O'REILLY®

Сложность масштабных систем неизбежно возрастает по мере того, как все больше компаний переходят на микросервисы и другие распределенные технологии. От сложности невозможно избавиться, но с помощью хаос-инжиниринга вы можете обнаружить уязвимости и предотвратить отказы до того, как они повлияют на ваших клиентов. Это практическое руководство рассказывает разработчикам и инженерам по эксплуатации, как лучше ориентироваться в сложных корпоративных системах, повышая их устойчивость для достижения бизнес-целей.

Двое выдающихся специалистов в этой области, К. Розенталь и Н. Джонс, стали пионерами в этой дисциплине во время совместной работы в Netflix. В данной книге они рассказывают о том, что такое хаос-инжиниринг, как и почему он появился, и в то же время организуют общение специалистов-практиков из разных отраслей. Многие главы книги написаны приглашенными авторами, чтобы расширить обзор как внутри, так и за пределами индустрии программного обеспечения.

«Сложные системы склонны проявлять хрупкость непредсказуемым образом. Авторы этой книги детально объясняют основы методики повышения устойчивости, приводят реальные примеры и дают практические советы о том, как использовать различные идеи, связанные с хаос-инжинирингом, чтобы сделать свои системы более устойчивыми».

Майкл Лопп,
бывший вице-президент
по инжинирингу,
компания Slack

- Узнайте, как хаос-инжиниринг позволяет вашей организации ориентироваться в сложных ситуациях.
- Изучите методологию, позволяющую избежать отказов в вашем приложении, сети и инфраструктуре.
- Перейдите от теории к практике, опираясь на реальные истории от отраслевых экспертов из Google, Microsoft, Slack, LinkedIn и других компаний мирового уровня.
- Воспользуйтесь поводом для размышлений о роли сложности в программных системах.
- Организуйте «игровые дни» для разработки и внедрения целенаправленных автоматизированных экспериментов хаос-инжиниринга в своей организации.

Кейси Розенталь – директор и соучредитель компании Verica. Ранее он был техническим руководителем команды хаос-инжиниринга в компании Netflix, имеет глубокий опыт работы с распределенными системами и искусственным интеллектом.
Нора Джонс – директор и соучредитель компании Jeli. Ее выступление на форуме AWS re:Invent в 2017 году послужило одним из мотиваторов массового внедрения хаос-инжиниринга.

ISBN 978-5-97060-796-1

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@alians-kniga.ru

www.дмк.рф

Хаос-инжиниринг

O'REILLY®

O'REILLY®

Хаос-инжиниринг

Революция в разработке устойчивых систем

Кейси Розенталь
Нора Джонс

издательство ДМК

Кейси Розенталь, Нора Джонс

Хаос-инжиниринг

Chaos Engineering

System Resiliency in Practice

Casey Rosenthal
Nora Jones

O'REILLY®

Хаос-инжиниринг

Революция в разработке устойчивых систем

Кейси Розенталь
Нора Джонс

ДМК Пресс
издательство
Москва, 2021

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

ISBN 978-1-492-04386-7 (англ.)
ISBN 978-5-97060-796-1 (рус.)

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.

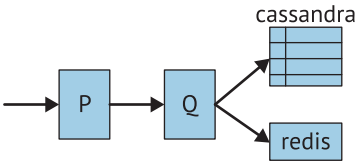

Рис. 1.1 ❖ Диаграмма компонентов микросервиса, показывающая поток запросов, поступающих в P и проходящих через хранилище

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.

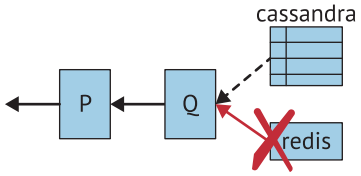

Рис. 1.2 ❖ В оперативном кеше T происходит сбой, в результате чего служба Q полагается на ответы из базы данных постоянного хранения S

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).

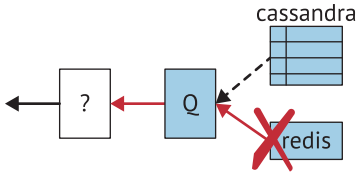

Рис. 1.3 ❖ Если T не отвечает, а S не в состоянии справиться с нагрузкой, требующей высокой скорости чтения, Q возвращает ответ по умолчанию на P

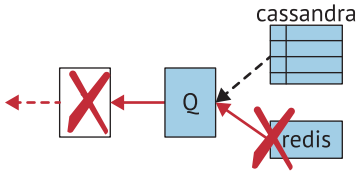

Рис. 1.4 ❖ Ответ по умолчанию `404[Not Found]` от Q выглядит логически невозможным для P, что приводит к катастрофическому сбою службы и обвалу системы

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].

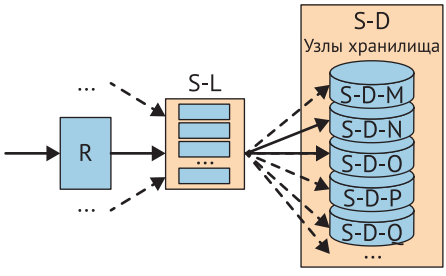

Рис. 1.5 ❖ Путь запроса данных о пользователе по имени Луи от R к S-D-N через S-L

[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.

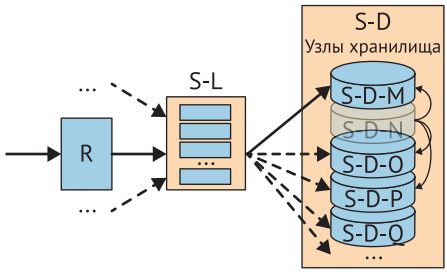

Рис. 1.6 ❖ Путь запроса от R к S-L к S-D-M для данных пользователя Луи после того, как S-D-N выключится и перебросит данные

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).

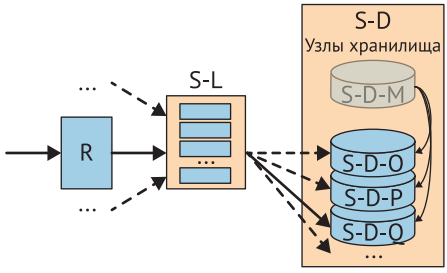

Рис. 1.7 ❖ Путь запроса от R к S-L и далее к S-D для пользовательских данных Луи после того, как S-D-M и S-D-N одновременно отключились и передали данные

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).

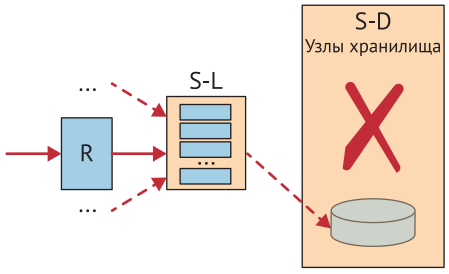

Рис. 1.8 ❖ Ситуация, когда подсистема S-D остановилась, а система R перегружена лавиной повторных запросов

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.

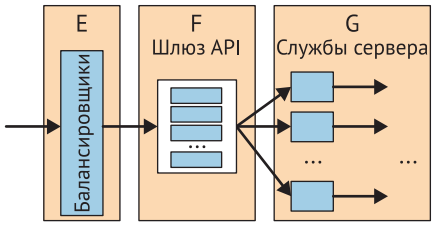

Рис. 1.9 ❖ Путь пользовательского запроса в крупном розничном интернет-магазине

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.

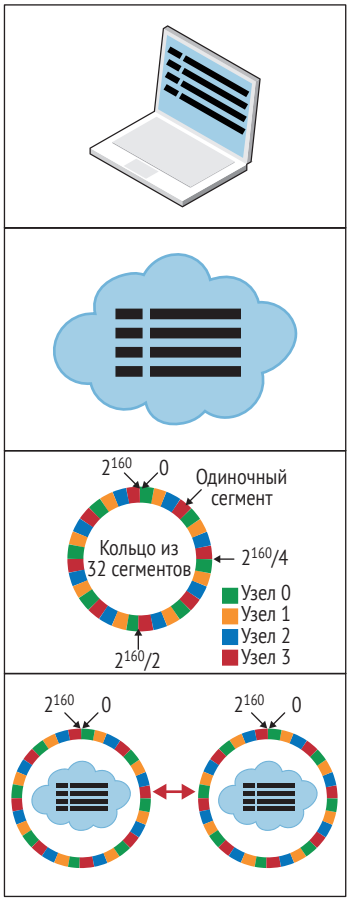

Рис. 1.10 ❖ Переход от простой базы данных ключ/значение к системе с высокой доступностью

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.

Глава 2

Навигация по сложным системам

В предыдущей главе мы говорили про сложные системы примерно так же, как если бы тонущий рассказывал про воду. Эта глава посвящена тому, как перестать тонуть и начать уверенно плыть по волнам. Мы представляем два метода, которые помогут вам лучше ориентироваться в сложных системах:

- динамическая модель безопасности;
- модель экономических факторов сложности.

Обе эти модели являются основой хаос-инжиниринга как дисциплины в разработке программного обеспечения.

2.1. Динамическая модель безопасности

Эта модель является адаптацией *динамической модели безопасности* Йенса Расмуссена[1], которая хорошо известна и высоко ценится в области технологий устойчивых систем. Представленная здесь адаптация просто помещает модель в контекст разработки программного обеспечения.

Динамическая модель безопасности имеет три свойства: экономика, рабочая нагрузка и безопасность (рис. 2.1). Представьте себе разработчика, которого поместили в центр между этих трех свойств и прикрепили к ним ре-

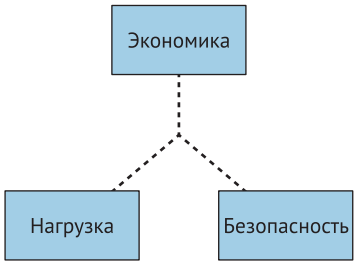

Рис. 2.1 ❖ Динамическая модель безопасности

[1] *Jens Rasmussen.* Risk Management in a Dynamic Society. Safety Science 27 (2–3), 1997.

зиновыми лентами одинаковой длины. В течение рабочего дня разработчик может двигаться в разные стороны, натягивая то одну, то другую ленту; однако если разработчик отклоняется слишком далеко, то какая-то резиновая лента рвется, и для этого разработчика игра окончена.

Захватывающая особенность данной модели заключается в том, что разработчики подсознательно оптимизируют свою работу, чтобы предотвратить разрывание резиновых лент. Мы коснемся каждого из трех свойств по очереди.

2.1.1. Экономика

Вероятно, у вас нет правила для разработчиков, согласно которому им не разрешается разворачивать миллион экземпляров в облаке в первый рабочий день. Почему у вас нет такого правила? Потому что в этом нет необходимости. Разработчики интуитивно понимают, что у них есть стоимость, у их команды есть стоимость и у вычислительных ресурсов есть стоимость. Это здравый смысл. Разработчики знают, что развертывание миллиона экземпляров в облаке будет стоить непомерно дорого и может обанкротить компанию. Есть бесчисленное множество других расходов, которые теоретически могут быть учтены разработчиком, но не учитываются, потому что нам хватает той формы сложности, с которой нам уже удобно ориентироваться. Мы можем сказать, что у разработчиков есть интуитивное понимание экономической выгоды, и они не пересекают ту границу, где эта резиновая лента порвется.

2.1.2. Нагрузка

Точно так же разработчики понимают, что они не могут работать 170 часов в неделю. У них есть предел рабочей нагрузки, с которой они могут справиться. Их команда имеет ограниченные возможности, которые они могут реализовать в спринте. Их инструменты и ресурсы тоже имеют ограничения допустимой нагрузки. Никто из разработчиков не надеется развернуть масштабную систему на ноутбуке, потому что они интуитивно знают, что нагрузка превысит возможности ноутбука. Есть бесчисленное множество других форм ограничений рабочей нагрузки, которые понимают разработчики, но, опять же, мы часто не учитываем их явно, потому что нам хватает той сложности, с которой нам удобно работать. Можно сказать, что у разработчиков есть интуитивное понимание запаса рабочей нагрузки, и они не пересекают эту границу.

2.1.3. Безопасность

Третье свойство модели – безопасность. В контексте информационно-вычислительных систем оно может относиться либо к доступности системы, либо к безопасности в смысле кибербезопасности[1]. Здесь мы наблюдаем от-

[1] См. главу 20 «Дело о разработке хаоса безопасности» Аарона Райнхарта для подробного исследования безопасности в Chaos Engineering.

личие от первых двух свойств. В то время как про экономику и рабочую нагрузку мы можем сказать, что у разработчиков есть интуитивное ощущение разумных границ, то же самое нельзя сказать о безопасности. Как правило, разработчики программного обеспечения не имеют интуитивного чутья для определения того, насколько близко они подошли к границе, за которой эта резиновая лента может лопнуть, что приведет к инциденту, будь то сбой или нарушение безопасности.

Мы считаем это широкое обобщение на всех разработчиков вполне обоснованным, потому что инциденты безопасности в информационной отрасли случаются регулярно и повсеместно. Если бы разработчики знали, что они находятся в опасной близости от границы безопасности, то они изменили бы свое поведение, чтобы предотвратить инцидент. Но это редкость, потому что инциденты почти всегда случаются неожиданно. Разработчики просто не имеют достаточно информации, чтобы оценить безопасность системы, и поэтому они переходят границу допустимого.

Что еще хуже, разработчики стремятся оптимизировать только то, что видят своими глазами. Поскольку у них есть интуиция для свойств «экономика» и «нагрузка», они видят эти свойства в своей повседневной работе и, таким образом, стремятся к этим полюсам на диаграмме (рис. 2.2), непреднамеренно уходя все дальше от безопасности. Этот эффект создает условия для медленного, незаметного дрейфа в сторону отказа системы, так как основные усилия разработчиков направлены на оптимизацию экономики и рабочей нагрузки, но не безопасности.

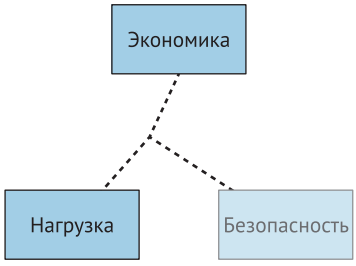

Рис. 2.2 ❖ В динамической модели безопасности, когда свойство безопасности не очевидно, разработчики стремятся отойти от него к свойствам экономики и рабочей нагрузки, которые более заметны

Одним из преимуществ идеологии хаос-инжиниринга для организации является то, что она помогает разработчикам развивать интуицию в области безопасности там, где ее не хватает по умолчанию. Эмпирические данные, полученные в результате экспериментов, дают представление о пробелах в интуиции разработчика. Конкретный результат может указать разработчикам на конкретную уязвимость, и они исправят недостаток. В этом есть некоторая польза, но гораздо важнее научить разработчиков вещам, которых они не видят, – показать, как механизмы безопасности взаимосвязаны со сложностью всей системы.

Более того, динамическая модель безопасности говорит нам, что как только у разработчиков появится интуитивное ощущение этой границы, они неявно изменят свое поведение, чтобы оптимизировать его. Они дадут себе запас прочности, чтобы не повредить резиновую ленту. Не надо твердить им, что следует избегать инцидента, или напоминать им быть осторожными, или поощрять их к применению лучших методов. Они сами будут неявно изменять свое поведение таким образом, что это приведет к разработке более устойчивой системы. Таким образом, динамическая модель безопасности служит основой хаос-инжиниринга.

2.2. Экономические факторы сложности

Эта модель является адаптацией модели Кента Бека[1], о которой рассказано в докладе профессора Энрико Занинотто, декана факультета экономики Университета Тренто. Адаптация, представленная в этой версии, специально предназначена для разработчиков программного обеспечения.

Как мы говорили в предыдущей главе, сложность может затруднить разработку, развертывание, эксплуатацию и обслуживание программного обеспечения. Сталкиваясь со сложностью, большинство разработчиков испытывают желание избежать или уменьшить ее. К сожалению, упрощение уменьшает полезность системы и в конечном итоге ограничивает ее ценность для бизнеса. Потенциал успеха возрастает только вместе со сложностью. Эта модель предоставляет один из способов, с помощью которого разработчики программного обеспечения справляются с этой сложностью.

В модели экономических факторов сложности есть четыре составляющие (рис. 2.3):

- состояния;
- отношения;
- окружающая обстановка;
- обратимость.

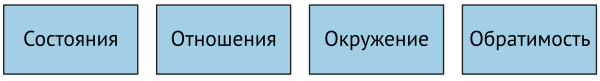

Рис. 2.3 ❖ Экономические факторы сложности

Модель утверждает, что степень, в которой организация может контролировать каждый фактор, соответствует успеху, с которым эта организация может ориентироваться в сложности конкурентного производственного процесса. Каждый из этих факторов лучше всего иллюстрируется на примере, и в качестве наиболее яркого примера обычно упоминают производителя автомобилей Ford середины 1910-х гг.

---

[1] *Kent Beck.* Taming Complexity with Reversibility. Facebook post, July 7, 2015, https://oreil.ly/jPqZ3.

2.2.1. Состояния

> «И вот в одно прекрасное утро 1909 года я объявил без всякого предварительного извещения, что в будущем мы станем выпускать лишь одну модель, а именно Model T, и что все машины будут иметь одинаковое шасси. Я заявил: “Каждый покупатель может заказать автомобиль любого цвета, если этот цвет черный”».
>
> – *Генри Форд*[1]

Как сказано в известной цитате Генри Форда, он радикально ограничил число *состояний* в процессе производства, продаж и технического обслуживания – с самого начала. Комплектующие были стандартизированы, а варианты кузова устранены. Это способствовало успеху компании в создании сложного и конкурентоспособного автомобильного бизнеса.

2.2.2. Отношения

Форд не остановился только на продукте. Реализуя собственный научный подход к управлению, компания также ограничила количество *отношений* в производственном процессе. У других автомобильных компаний были команды, которые собирали всю машину от начала и до конца. Сборка требовала сложных коммуникаций и координации усилий на протяжении всего производственного процесса. Форд разбил задачи на небольшие строго регламентированные этапы, которые превращали интеллектуальный процесс сборки в простой набор механических движений. Отпала всякая нужда в отношениях между людьми, собирающими автомобили. В то время такое решение стало революцией в управлении сложностью автомобильного бизнеса. Генри Форд знал это и долго колебался между ревностной охраной секрета и выставлением преимущества напоказ.

2.2.3. Окружение

Большинство организаций не имеют ресурсов для прямого воздействия на *окружение*. Форд потратил годы на борьбу с Ассоциацией лицензированных производителей автомобилей (ALAM), что в конечном итоге привело к распаду этой монополии. В течение нескольких десятилетий Ассоциация душила инновации в автомобильной промышленности патентами и судебными разбирательствами, и после этой победы Форд получил возможность свободно производить более передовые модели двигателей и продавать их без уплаты чрезмерных лицензионных сборов в пользу ALAM. Через несколько лет у самого Форда появилось достаточно власти, чтобы влиять на законодательство, буквально создавая окружающую обстановку под себя, дабы обеспечить более предсказуемую судьбу своего продукта в Соединенных Штатах. Это, очевидно, позволило компании еще лучше управлять сложностью производства автомобилей.

---

[1] *Форд Г.* Моя жизнь. Мои достижения. АСТ, 2019.

2.2.4. Обратимость

К огорчению Форда, обратить производственный процесс не так легко, как заставить автомобиль ехать задним ходом. Готовые автомобили очень трудно разобрать и переделать, чтобы отменить неудачные решения. Парадокс еще и в том, что внедренные руководством безупречно отлаженные и эффективные решения также очень затруднили импровизацию при разработке новых проектов. У Форда было не так уж много возможностей, чтобы влиять на этот фактор.

2.2.5. Экономические факторы сложности и программное обеспечение

В примере, который мы только что рассмотрели, Форд смог оптимизировать первые три фактора, но не четвертый. Как это относится к программному обеспечению?

Как правило, бизнес-цели стимулируют увеличение количества состояний. Это справедливо как в смысле создания и сохранения большего количества данных с течением времени (состояния приложения), так и в смысле увеличения функциональности и количества различных вариантов продукта. Люди обычно хотят больше функций, а не меньше. Поэтому при разработке программного обеспечения редко удается оптимизировать количество состояний.

Программное обеспечение также во многом связано с добавлением слоев абстракции. Эти новые слои требуют новых отношений. Даже по сравнению с несколькими годами ранее количество компонентов в среднем программном стеке значительно возросло. Возьмем в качестве примера типичную систему при масштабном развертывании. Здесь стало нормой наличие нескольких уровней виртуализации – от облачных экземпляров до контейнеров и функций в качестве службы. При общем переходе на микросервисные архитектуры отрасль, похоже, сработала против своих интересов в плане управления сложностью, целенаправленно увеличивая связи между частями в системе. Вдобавок многие компании-разработчики программного обеспечения переходят на удаленную работу, ослабляя контроль над человеческими ресурсами, влияющими на процесс.

Большинство разработчиков не имеют средств для управления взаимоотношениями, чтобы лучше управлять сложностью; на самом деле, похоже, все наоборот. И тем более очень немногие компании-разработчики программного обеспечения имеют такой размер или масштаб, чтобы оказывать существенное влияние на окружающую обстановку. Это делает данный фактор недоступным практически для всех разработчиков программного обеспечения.

У нас остался последний фактор: обратимость. Вот где программное обеспечение восседает на белом коне. Очевидно, что управлять развертыванием приложения намного проще, чем переделывать механические устройства. Значительные улучшения по части обратимости программного обеспечения начались с *экстремального программирования* (XP) и гибкого подхода (Agile).

Предыдущий общепринятый метод создания программного продукта, известный как *каскадная разработка*, предусматривал предварительное планирование, проектирование и длительное внедрение. Продукт когда-то будет представлен покупателю, возможно, после года напряженной работы, и если покупателю он не понравится, то ничего не поделаешь. Программное обеспечение тогда не имело большой обратимости. Короткие итерации экстремального программирования радикально изменили эту формулу. Буквально через пару недель после начала разработки клиенту можно представить первые наброски решения. Если ему это не понравилось, то отказ (обращение проектного процесса вспять) не станет приговором проекта. Следующая итерация будет ближе к тому, чего хочет клиент, а третья еще ближе, пока к четвертой итерации (хорошо, если так) разработчики не поймут наконец, чего на самом деле хочет клиент. Так выглядит метод улучшения обратимости, основанный на процессах.

Большим преимуществом информационных технологий является то, что вы можете применять простые проектные решения, которые также улучшают обратимость. Разработка приложений, работающих в интернете, где обновление окна браузера – это все, что требуется для развертывания новой кодовой базы, значительно улучшила обратимость приложений. Контроль версий, методы параллельного развертывания, автоматизированные канареечные релизы, флаги функций, непрерывная интеграция / непрерывная доставка (CI/CD) – все это примеры технических и архитектурных решений, которые явно улучшают обратимость. Любая техника, которая повышает способность инженера-программиста вводить код в производство, передумывать и отменять свое решение, улучшает обратимость. Это, в свою очередь, облегчает управление современной разработкой программного обеспечения.

С этой точки зрения возможность оптимизации обратимости является достоинством в современной разработке. Оптимизация обратимости приносит плоды при работе со сложными системами. Это еще один базовый подход хаос-инжиниринга. Эксперименты хаос-инжиниринга раскрывают свойства системы, которые препятствуют обратимости. Во многих случаях таким свойством может быть эффективность, на которой сосредоточились разработчики.

Мы согласны с тем, что эффективность может служить бизнес-целью, но эффективность – хрупкая штука. Эксперименты с хаосом могут выявить хрупкость, а затем нужно подумать, что делать дальше – продолжать бороться за эффективность или оптимизировать обратимость таким образом, чтобы система в целом не пострадала, если эта хрупкая деталь сломается. Хаос-инжиниринг действует как прожектор, высвечивая места, которые разработчики могут оптимизировать по части обратимости.

2.3. Системный подход

Целостный, системный подход хаос-инжиниринга – это одна из особенностей, которая отличает его от других практик. Наличие интуитивного восприятия границ безопасности неявно меняет поведение разработчиков

в сторону оптимизации безопасности. Наличие навыка выявления целей для улучшения обратимости помогает разработчикам оптимизировать методику обновления отказоустойчивых систем. Обе эти модели заложены в основу хаос-инжиниринга как подхода, который помогает разработчикам жить рядом со сложностью, а не бороться с ней. В следующих главах вы прочтете про несколько примеров хаос-инжиниринга, задействованного в различных компаниях. Изучая эти примеры, обратите внимание, как хаос-инжиниринг выявляет границы безопасности и определяет цели для улучшения обратимости. Это ключи для безопасной работы со сложными системами.

Глава 3

Обзор принципов хаос-инжиниринга

Первые дни создания хаос-инжиниринга в Netflix были хаотичными в прямом смысле этого слова. Регулярно слышались фразы о том, что надо выдернуть провода или как-то иначе уронить службы; было много заблуждений о том, как сделать сервисы надежными, и очень мало примеров реальных инструментов. Пришлось создать специальную команду Chaos Team и поставить перед ней задачу сформировать новую дисциплину, которая проактивно повышает надежность с помощью специальных инструментов. Мы потратили месяцы на исследования в области технологий устойчивости и других дисциплин в поисках единой концепции, и наконец составили документ, в котором определено понятие хаос-инжиниринга и предложен план привлечения участников в сообщество хаос-инженеров. Этот документ мы скромно озаглавили «Принципы» и опубликовали в сети как своего рода манифест новой идеологии (см. введение, в котором рассказано, как возникло сообщество хаос-инжиниринга).

Как обычно случается с новой идеологией, не все правильно понимают суть хаос-инжиниринга. В следующих разделах мы поясним, что относится к этой дисциплине, а что нет. Про стандарт применения на практике мы расскажем в разделе 3.3. Наконец, мы покажем, какие факторы могут повлиять на «Принципы» в будущем.

3.1. Что такое хаос-инжиниринг

«Принципы» для того и разработаны, чтобы, занимаясь хаос-инжинирингом, мы знали, как это делать и как делать это хорошо. Сегодня наиболее общее определение хаос-инжиниринга звучит так: «*проведение организованных экспериментов для выявления системных недостатков*». На веб-сайте «Принципов» изложены следующие этапы эксперимента:

1) начните с определения «устойчивого состояния» как некоторого *измеримого* результата системы, указывающего на нормальное поведение;
2) выдвиньте гипотезу, что это устойчивое состояние сохранится как для контрольной группы, так и для экспериментальной группы;

3) введите переменные факторы, которые отражают реальные события, такие как сбой серверов, сбой жестких дисков, разрыв сетевых подключений и т. п.;
4) попытайтесь опровергнуть гипотезу, находя разницу в установившемся состоянии между контрольной группой и экспериментальной группой.

В этом эксперименте заключаются основные принципы хаос-инжиниринга. В плане реализации таких экспериментов вы располагаете большой свободой действий.

3.1.1. Эксперименты или тестирование?

Одна из первых особенностей, которые мы обнаружили в Netflix, заключается в том, что хаос-инжиниринг – это разновидность эксперимента, а не тестирование. Возможно, оба понятия попадают под зонтик «контроль качества», но эта фраза часто имеет негативную коннотацию в индустрии программного обеспечения.

Другие команды в Netflix поначалу регулярно задавали команде Chaos Team вопросы наподобие «Разве не проще написать несколько комплексных тестов, которые ищут то же самое?». Это мнение разумно выглядит в теории, но на практике оказалось, что комплексные тесты не дают желаемого результата.

Строго говоря, тестирование не создает новых знаний. Тестирование требует, чтобы инженер, пишущий тест, заранее знал конкретные свойства системы, которые он ищет. Как сказано в предыдущей главе, сложные системы непрозрачны для такого типа анализа. Люди физически не способны соединить у себя в голове все потенциальные побочные эффекты от всех возможных взаимодействий частей в сложной системе. Это приводит нас к одному из ключевых свойств теста.

Тесты выдвигают утверждение, основанное на *существующих* знаниях, а затем запуск теста сводит утверждение к истинному или ложному значению. Тесты – это утверждения об известных свойствах системы.

Эксперименты, напротив, создают *новые* знания. Эксперименты предлагают гипотезу, и пока эта гипотеза не опровергнута, доверие к этой гипотезе возрастает. Если она будет опровергнута, то мы узнаем что-то новое. У нас появятся вопросы, на которые надо найти ответ, чтобы выяснить, почему наша гипотеза неверна. В сложной системе причина, по которой что-то происходит, зачастую не очевидна. Эксперименты либо укрепляют доверие, либо учат нас новым свойствам нашей собственной системы. Это исследование неизвестного.

Никакое количество прикладных тестов не может сравниться с пониманием, полученным в результате экспериментов, потому что тестирование требует, чтобы человек выдвигал утверждения заранее. Экспериментирование, по сути, дает нам способ открыть новые свойства. Потом вы можете преобразовать вновь обнаруженные свойства системы в тесты. Также вы можете превратить новые знания о системе в новые гипотезы, что создает

нечто вроде «регрессионного эксперимента», который исследует изменения системы с течением времени.

Поскольку хаос-инжиниринг родился при работе со сложными системами, важно, чтобы дисциплина включала в себя именно эксперименты, а не тестирование. Четыре этапа эксперимента, изложенных в «Принципах», примерно соответствуют общепринятым определениям с точки зрения изучения проблем доступности в масштабных системах.

Сочетание реального опыта применения ранее описанных шагов к масштабным системам, а также вдумчивый самоанализ позволили команде Chaos Team развить концепцию дальше простой схемы экспериментов. Эти идеи легли в основу «Продвинутых принципов» и служат золотым стандартом для команд, входящих в сообщество хаос-инженеров.

3.1.2. Функциональный контроль или аттестация?

Используя определения функционального контроля и аттестации, основанные на операционном управлении и логистическом планировании, мы можем сказать, что хаос-инжиниринг намного ближе к первому, чем ко второму.

Функциональный контроль

Функциональный контроль сложной системы – это процесс анализа результатов на границе системы. Домовладелец может проверить качество воды (выход), поступающей из кухонного крана (граница системы), проверив ее на наличие загрязняющих веществ, и при этом ничего не знать о том, как функционирует водопроводная сеть или коммунальное водоснабжение (детали системы).

Аттестация

Аттестация сложной системы – это процесс анализа частей системы и построения ментальных моделей, отражающих взаимодействие этих частей. Домовладелец может удостовериться в качестве воды косвенным образом, осмотрев все трубы и инфраструктуру (части системы), участвующие в сборе, очистке и доставке воды (интеллектуальная модель функциональных частей) в жилой район и в конечном итоге в рассматриваемый дом.

Обе эти практики потенциально полезны, и обе вырабатывают уверенность в результатах работы системы. Как разработчики программного обеспечения мы часто чувствуем необходимость погрузиться в код и подтвердить, что он отражает нашу ментальную модель того, как он должен работать. Вопреки этому предпочтению, хаос-инжиниринг настоятельно рекомендует контроль, а не аттестацию. Хаос-инжиниринг заботится о том, работает ли система *как надо*, а не о том, как устроен механизм ее работы.

Обратите внимание, что в метафоре сантехники мы могли бы проверить по отдельности все компоненты, которые входят в систему подачи чистой питьевой воды, и все же по какой-то причине получить из крана совсем не

то, чего ожидали. В сложной системе всегда есть непредсказуемые взаимодействия. Но если мы ставим перед собой задачу убедиться, что вода в кране чистая, то нам не обязательно заботиться о том, как она попала туда. В большинстве бизнес-случаев результат работы системы намного важнее, чем соответствие системы нашей личной ментальной модели. Хаос-инжиниринг больше заботится о бизнес-кейсе и результатах, чем о реализации или ментальной модели взаимодействующих частей.

3.2. Чем не является хаос

Есть две концепции, которые часто путают с хаос-инжинирингом, а именно разрушающие продакшн-тесты и стресс-тесты на устойчивость.

3.2.1. Разрушающее тестирование производства

Время от времени в комментариях блога или на конференциях специалисты упоминают хаос-инжиниринг как «брейк-тест продакшна». Хотя это очень круто звучит на языке пингвинов, но фактически не относится к масштабным предприятиям и другим операторам сложных систем, извлекающим наибольшую пользу из хаос-инжиниринга. Было бы правильнее называть хаос-инжиниринг *исправлением* неполадок в производстве. Не составит особого труда размахивать кувалдой, сокрушая систему; намного труднее обеспечивать минимальный радиус поражения, критически размышлять о безопасности, определяя, надо ли вообще что-то исправлять, принимать решение, стоит ли инвестировать в новые эксперименты, – список можно продолжать долго. «Сломать производство» можно бесчисленным количеством способов, затратив на это мало времени. На самом деле вопрос в другом: как мы можем судить о вещах, которые фактически *уже* сломаны, если мы даже не знаем, что они сломаны?

«Исправление неполадок в производстве» намного лучше отражает ценность хаос-инжиниринга, поскольку его цель заключается в *проактивном* (опережающем) улучшении доступности и безопасности сложной системы. На сегодняшний день существует множество методик и инструментов для реагирования на инциденты: инструменты оповещения, управление реагированием, инструменты наблюдения, планирование аварийного восстановления и т. д. Они направлены на то, чтобы сократить время обнаружения и время восстановления после неизбежного инцидента. Можно сказать, что *методика обеспечения надежности* (site reliability engineering, SRE) охватывает как реактивные (постфактум), так и проактивные (упреждающие) дисциплины, генерируя знания о прошлых инцидентах и помогая предотвратить будущие инциденты. Хаос-инжиниринг – единственная базовая дисциплина в программировании, которая сосредоточена исключительно на *опережающем* повышении безопасности в сложных системах.

3.2.2. Антихрупкость

Люди, знакомые с концепцией *антихрупкости*, представленной Нассимом Талебом[1], часто предполагают, что хаос-инжиниринг – это, по сути, программная реализация антихрупкости. Талеб утверждает, что таких терминов, как «гормезис», недостаточно, чтобы передать способность сложных систем адаптироваться, и поэтому он изобрел понятие «антихрупкость», обозначающее системы, которые становятся сильнее при случайном стрессовом воздействии. Важное, критическое различие между хаос-инжинирингом и антихрупкостью заключается в том, что хаос-инжиниринг рассказывает разработчикам о хаосе, уже присущем системе, чтобы они могли стать более гибкой командой. Антихрупкость, напротив, добавляет хаос в систему в надежде, что она станет сильнее, а не сломается.

В основе антихрупкости лежит идея, противоречащая современным исследованиям в области обеспечения устойчивости, человеческого фактора и систем безопасности. Например, антихрупкость подразумевает, что первым шагом в повышении надежности системы должно быть выявление слабых сторон и их устранение. Это предложение кажется интуитивно понятным, однако методика обеспечения устойчивости говорит нам, что определение *правильного* порядка вещей в безопасности гораздо более информативно, чем поиск того, что идет не так. Следующим шагом в антихрупкости является добавление избыточности. Это также кажется интуитивно понятным, но добавление избыточности может вызвать сбой так же легко, как и смягчить его, а в литературе по обеспечению устойчивости есть множество примеров, когда избыточность фактически провоцирует нарушение безопасности[2].

Есть множество других примеров расхождения между этими двумя школами мысли. Обеспечение устойчивости – это постоянная область исследований с десятилетиями опыта, в то время как антихрупкость – это теория, которая обитает в основном за пределами научных кругов и экспертной оценки. Можно подумать, что эти две концепции взаимосвязаны, поскольку обе имеют дело с хаосом и сложными системами, но антихрупкость не разделяет эмпиризм и фундаментальное обоснование хаос-инжиниринга. По этим причинам мы должны рассматривать их как принципиально разные занятия[3].

---

1 *Нассим Талеб.* Антихрупкость. Как извлечь выгоду из хаоса. КоЛибри, 2020.

2 Возможно, самый известный пример – катастрофа корабля «Челленджер» в 1986 году. Избыточность уплотнительных колец была одной из трех причин, по которым НАСА одобрило продолжение запусков, хотя о повреждении первичного уплотнительного кольца было хорошо известно после более чем пятидесяти предыдущих запусков в течение пятилетнего периода. См.: *Diane Vaughan.* The Challenger Launch Decision. Chicago: University of Chicago Press, 1997.

3 *Casey Rosenthal.* Antifragility Is a Fragile Concept. LinkedIn post, Aug. 28, 2018, https://oreil.ly/Lbult.

3.3. Ключевые принципы хаос-инжиниринга

Хаос-инжиниринг основан на эмпиризме, превалировании экспериментов над тестированием и функционального контроля над аттестацией. Но не все эксперименты одинаково полезны. Раздел «Ключевые принципы» в манифесте создателей хаос-инжиниринга содержит следующие стандартные этапы методики эксперимента:

- построение гипотезы о стабильном поведении;
- моделирование различных событий реального мира;
- выполнение экспериментов на производстве;
- автоматизация непрерывного запуска экспериментов;
- минимизация радиуса поражения.

3.3.1. Построение гипотезы о стабильном поведении

Каждый эксперимент начинается с гипотезы. Для экспериментов с доступностью гипотеза эксперимента обычно имеет вид:

В ситуации _____ клиенты все еще остаются довольны.

Для экспериментов с безопасностью, напротив, гипотеза эксперимента обычно имеет вид:

При обстоятельствах _____ уведомляется служба безопасности.

В обоих случаях пустое пространство заполняется переменными, упомянутыми в следующем разделе.

Ключевые принципы подчеркивают важность построения гипотезы о стабильном состоянии. Это означает, что необходимо сосредоточиться на том, как система должна вести себя в стабильном состоянии, и зафиксировать это в виде измеримых данных. В предыдущих примерах клиенты стабильно работающей системы, по-видимому, по умолчанию хорошо проводят время, и служба безопасности обычно получает уведомление, когда происходит подозрительное событие.

Эта нацеленность на стабильное поведение заставляет разработчиков отступить от кода и сосредоточиться на комплексном результате. Она отражает стремление хаос-инжиниринга к проверке достоверности гипотезы. У нас часто возникает желание погрузиться в проблему, найти «истинную причину» поведения и попытаться понять систему с помощью редукционизма. Глубокое погружение в проблему может помочь в освоении системы, но это отвлекает от опережающего анализа надежности, который может предложить хаос-инжиниринг. Самое большее, на чем фокусируется хаос-инжиниринг, – это ключевые показатели эффективности (KPI) или другие показатели, которые соответствуют четким бизнес-приоритетам и дают наилучшее определение стабильного поведения.

3.3.2. Моделирование различных событий реального мира

Этот этап требует, чтобы переменные факторы экспериментов отражали реальные события. Хотя в ретроспективе это может показаться очевидным, есть две веские причины настойчиво говорить об этом:

- переменные часто выбираются из того, что легко сделать, а не из того, что обеспечивает наилучшее обучение;
- разработчики склонны фокусироваться на переменных, которые отражают их личный опыт, а не опыт пользователей.

Не ищите легкие пути

Приложение Chaos Monkey[1] на самом деле устроено довольно тривиально для такой мощной программы. Это продукт с открытым исходным кодом, который случайным образом отключает экземпляры (виртуальные машины, контейнеры или серверы) примерно один раз в день для каждой службы. Вы можете использовать его как есть, но те же функции в большинстве организаций могут быть реализованы простым скриптом bash. По сути, это первый плод хаос-инжиниринга. Если вы используете развертывание в облаке, а теперь и развертывание контейнеров, то можете не сомневаться, что в масштабируемых системах экземпляры (виртуальные машины или контейнеры) будут периодически «отваливаться». Chaos Monkey воспроизводит этот фактор и просто ускоряет частоту события.

Это полезное мероприятие легко реализуется, если у вас есть привилегии корневого уровня для инфраструктуры и вы можете заставить эти экземпляры исчезать из системы. Если у вас есть привилегии корневого уровня, возникает соблазн делать и другие вещи, которые легко сделать от имени `root`. Рассмотрим следующие переменные-события, доступные для экземпляра:

- завершить экземпляр;
- перегрузить процессор экземпляра;
- переполнить оперативную память экземпляра;
- переполнить диск экземпляра;
- отключить сеть экземпляра.

Эти эксперименты имеют одну общую черту: они, как и ожидалось, приводят к тому, что экземпляр перестает отвечать. С точки зрения системы это выглядит так же, как если бы вы завершили экземпляр. Вы не узнали из последних четырех экспериментов ничего такого, чего вы не узнали из первого эксперимента. Остальные эксперименты, по сути, являются пустой тратой времени.

С точки зрения распределенной системы, почти все интересные эксперименты по доступности могут быть реализованы при помощи задержки или определенного типа ответа. Отключение экземпляра является частным случаем бесконечной задержки. В большинстве онлайн-систем сегодня тип ответа часто синонимичен коду состояния, например изменение ответа

[1] Chaos Monkey был первым инструментом хаос-инжиниринга. Во введении более подробно рассказано о том, как возник хаос-инжиниринг.

сервера HTTP 200 (OK) на HTTP 500 (внутренняя ошибка сервера). Из этого следует, что большинство экспериментов по доступности могут быть выполнены с помощью изменения задержки и/или кодов состояния.

Изменить задержку гораздо сложнее, чем просто перегрузить процессор или заполнить оперативную память экземпляра. Это требует координации на всех уровнях *межпроцессного взаимодействия* (inter-process communication, IPC), что, в свою очередь, может означать изменение внешних расширений, программно определяемых сетевых правил, оболочек клиентской библиотеки, служебных подсетей или даже применение облегченных балансировщиков нагрузки. Любое из этих решений требует нетривиальных инженерных усилий.

3.3.3. Выполнение экспериментов на производстве

Экспериментируя, вы изучаете систему, с которой работаете. Если вы экспериментируете в подготовительной среде (staging), то формируете уверенность только в этой среде. Однако подготовительная среда и рабочая среда производства (production) могут различаться самым непредсказуемым образом, поэтому вы не создаете уверенности в среде, которая вас действительно волнует. По этой причине самые передовые технологии хаос-инжиниринга применяются в производстве.

Этот принцип не лишен противоречий. Разумеется, в некоторых отраслях существуют нормативные требования, исключающие возможность воздействия на производственные системы. В иных ситуациях существуют непреодолимые технические барьеры для проведения экспериментов на производстве. Важно помнить, что цель хаос-инжиниринга состоит в том, чтобы выявлять хаос, присущий сложным системам, а не вызывать его. Если мы знаем, что эксперимент приведет к нежелательному результату, то не должны запускать этот эксперимент. Это особенно важно в производственной среде, где последствия опровержения гипотезы о стабильности могут стоить слишком дорого.

Мудрые разработчики не используют категоричный принцип «все или ничего» при планировании экспериментов в производстве. В большинстве ситуаций имеет смысл начать эксперименты на подготовительном этапе и постепенно перейти к производству, как только будет проработан безопасный инструментарий. Во многих случаях благодаря хаос-инжинирингу критическая информация о производстве выявляется еще на этапе подготовки и развертывания.

3.3.4. Автоматизация непрерывного запуска экспериментов

Этот принцип отражает практическое значение регулярного запуска экспериментов на сложных системах. Автоматизация должна быть введена по двум причинам:

1) чтобы охватить больший набор экспериментов, чем люди могут выполнить вручную. В сложных системах условия, которые могли бы способствовать инциденту, настолько многочисленны, что их невозможно запланировать. На самом деле они даже не могут быть учтены, потому что их невозможно узнать заранее. Это означает, что люди не могут надежно искать в пространстве решений возможные провоцирующие факторы в разумные сроки. Автоматизация предоставляет средства для масштабирования поиска уязвимостей, которые могут привести к нежелательным системным последствиям;
2) эмпирически проверять наши предположения с течением времени, так как неизвестные части системы изменяются. Представьте себе систему, в которой функциональность данного компонента зависит от других компонентов, находящихся вне области наблюдения оператора системы. Так бывает почти во всех сложных системах. Вполне возможно, что одна из зависимостей изменится таким образом, что создаст уязвимость. Непрерывные автоматические эксперименты могут отследить эти проблемы и показать операторам, как со временем меняется работа их системы. Это может быть изменение в производительности (например, сеть перегружается шумными соседями), или изменение в функциональности (например, ответы нисходящих служб содержат дополнительную информацию, которая может повлиять на порядок обработки), или изменения в человеческом факторе (например, разработчики покидают команду, а инженеры по эксплуатации не так хорошо знакомы с кодом).

Сама автоматизация тоже может иметь непредвиденные последствия. В главах 11 и 12 рассматриваются некоторые плюсы и минусы автоматизации. «Принципы хаос-инжиниринга» утверждают, что автоматизация является продвинутым механизмом для изучения пространства решений потенциальных уязвимостей и для уточнения институциональных знаний об уязвимостях путем проверки гипотезы с течением времени, зная, что сложные системы изменятся.

3.3.5. Минимизация радиуса поражения

Этот последний принцип добавили после того, как команда Chaos Team в Netflix обнаружила, что можно значительно снизить риск для трафика производства, разработав более безопасные способы проведения экспериментов. Используя небольшую подопытную группу, можно устроить эксперименты таким образом, чтобы влияние опровергнутой гипотезы на трафик клиентов в производстве было минимальным.

В целом метод минимизации радиуса поражения в значительной степени зависит от системы. В некоторых системах можно использовать параллельный трафик; или не трогать запросы, которые имеют большое значение для бизнеса, например транзакции свыше 100 долларов США; или внедрить логику автоматической повторной попытки для запросов, проваленных в ходе эксперимента. Например, команда Chaos Team из Netflix использова-

ла выборку запросов, привязку сессий и аналогичные функции, объединив их в платформу автоматизации Chaos (chaos automation platform, ChAP)[1], которая более подробно обсуждается в главе 16. Эти методы не только ограничивают радиус поражения; они дают дополнительное преимущество в обнаружении неполадок, поскольку показатели небольшой подопытной группы часто могут резко отличаться от небольшой контрольной группы. Как бы то ни было, этот принцип подчеркивает, что в действительно сложных системах потенциальное влияние эксперимента можно (и нужно) преднамеренно ограничивать.

Все эти принципы придуманы, чтобы направлять и вдохновлять разработчиков, а не диктовать им правила. Они воплощают прагматизм и должны быть приняты или отвергнуты исключительно из практических соображений.

Сосредоточьтесь на опыте пользователей

Улучшение *опыта разработчиков* (experience of developers) заслуживает отдельного внимания. DevUX – недооцененная дисциплина. Согласованные усилия по улучшению опыта разработчиков при написании, обслуживании и доставке кода в производство и отмене этих решений приносят осязаемую долгосрочную отдачу. Тем не менее основная польза от внедрения хаос-инжиниринга заключается в поиске уязвимостей *действующей* системы, а не в процессе разработки. Поэтому имеет смысл сосредоточиться на переменных, которые могут повлиять на *пользовательский* опыт.

Поскольку в экспериментах с хаосом выбором переменных обычно занимаются инженеры-программисты (а не пользователи), фокус внимания иногда смещается. Примером такого неправильного внимания является энтузиазм инженеров Chaos Team по поводу экспериментов с повреждением данных. Есть много мест, где эксперименты с повреждением данных оправданы и очень ценны. Ярким примером этого служит проверка надежности баз данных. Однако повреждение содержимого ответа при передаче клиенту, вероятно, не является хорошим примером.

Рассмотрим все более популярные эксперименты, в результате которых происходит искажение ответа – возвращается неправильно сформированный код HTML или поврежденный JSON. Скорее всего, это событие не произойдет в реальном мире, а если даже произойдет, то, вероятно, будет сглажено поведением пользователя (событие повтора), событием отката (другой тип события повтора) или поведением тонкого клиента (например, веб-браузерами).

Как инженеры мы часто сталкиваемся с несоответствием ожиданиям. Мы видим, что библиотеки, взаимодействующие с нашим кодом, ведут себя не так, как мы хотели. Мы тратим уйму времени на настройку взаимодействия, чтобы добиться от библиотек правильного поведения. Поскольку мы часто видим поведение, которое нам не нужно, то начинаем думать, что стоит проводить эксперименты, разоблачающие подобные случаи. Это заблуждение; в таких случаях эксперименты с хаосом не нужны. Устране-

[1] *Ali Basiri et al.* ChAP: Chaos Automation Platform // The Netflix Technology Blog, July 26, 2017, https://oreil.ly/U7aaj.

ние несоответствий является частью процесса разработки. Как только код заработает, как только взаимодействие будет настроено, крайне маловероятно, что космический луч или перегоревший транзистор исказит вывод библиотеки и повредит данные при передаче. Даже если мы полагаем, что это возможно из-за нестабильности библиотеки или чего-то в этом роде, это *известное* свойство системы. Известные свойства лучше всего исследовать с помощью тестирования.

Хаос-инжиниринг существует не настолько долго, чтобы успеть формализовать методы, используемые для генерации переменных. Некоторые методы очевидны, как, например, введение задержки. Некоторые требуют анализа, например подбор нужной величины задержки, чтобы вызвать эффекты очереди без фактического превышения порога, за которым срабатывает оповещение о сбое. Некоторые из них очень зависимы от контекста, например снижают производительность службы второго уровня таким образом, что другая служба второго уровня временно становится службой первого уровня.

Мы ожидаем, что по мере развития дисциплины эти методы либо предоставят обобщенные модели для генерации переменных, либо накопят модели по умолчанию, которые отражают обобщенный опыт всей отрасли. Тем временем избегайте простых путей, сосредоточьтесь на опыте пользователей и ориентируйтесь на события реального мира.

3.4. Будущее «Принципов»

За пять лет, прошедших с момента публикации «Принципов», мы стали свидетелями бурного развития хаос-инжиниринга, решающего новые задачи в новых отраслях. Принципы и основы этой методики, без сомнения, еще долго будут развиваться по мере внедрения в индустрию программного обеспечения и перехода на новые вертикали.

Когда Netflix в 2015 году начал активно продвигать идею хаос-инжиниринга, он получил много откликов, в частности от финансовых учреждений. В основном это были сомнения: «Конечно, это неплохо работает для развлекательной службы или интернет-рекламы, но у нас на кону реальные деньги». На это ребята из Chaos Team задавали вопрос: «У вас бывают перебои в работе?»

Разумеется, звучал ответ «да»; даже лучшие системы финансовых организаций испытывают дорогостоящие перебои в работе. По мнению Chaos Team, у обладателей масштабных систем есть два варианта: либо (а) продолжать испытывать перебои с непредсказуемым уровнем и серьезностью, либо (б) задействовать упреждающую стратегию, такую как хаос-инжиниринг, для исследования рисков с целью предотвращения разрушительных последствий. Идея нашла понимание у финансовых организаций, и многие крупнейшие банки мира в настоящее время разработали специальные программы хаос-инжиниринга.

Следующей отраслью, которая испытывала сомнения по поводу хаос-инжиниринга, было здравоохранение: «Конечно, это неплохо работает для он-

лайн-развлечений или финансового сектора, но у нас на кону человеческие жизни». И снова команда Chaos Team задавала вопрос: «У вас бывают перебои в работе?»

Но в этом случае можно проследить еще более прямую аналогию здравоохранения как системы. Прикладные эксперименты, заложенные в основу хаос-инжиниринга, – это прямое обращение к концепции фальсифицируемости Карла Поппера, которая лежит в основе современных представлений о науке и научном методе. Вершина практического воплощения понятий Поппера – это клиническое испытание.

В этом смысле феноменальный успех современной системы здравоохранения основан на хаос-инжиниринге. Современная медицина зависит от двойных слепых экспериментов с человеческими жизнями. Просто они называют это другим именем: клиническое испытание.

Различные формы хаос-инжиниринга давно существуют во многих других отраслях. Активное внедрение концепции экспериментирования, особенно в смежных с информатикой отраслях, дает нам бесценный практический опыт. Хаос-инжиниринг позволяет нам выработать стратегию по достижению цели, а также извлечь уроки из других областей и применить их к нашим собственным задачам.

В перспективе мы можем исследовать применение хаос-инжиниринга в отраслях и компаниях, которые сильно отличаются от первоначальных масштабных микросервисных систем. Финансовые технологии, автономные транспортные средства и состязательное машинное обучение могут рассказать нам много нового о возможностях и ограничениях хаос-инжиниринга. Машиностроение и авиация еще больше расширяют наш кругозор, выводя нас из сферы программного обеспечения в мир аппаратных и физических объектов. Хаос-инжиниринг перестал ограничиваться вопросами доступности и охватывает безопасность – другую сторону медали с точки зрения устойчивости системы. Все эти новые отрасли, варианты использования и среды послужат источником энергии для дальнейшего развития хаос-инжиниринга.

Часть II

ПРИНЦИПЫ ХАОСА В ДЕЙСТВИИ

Мы уверены, что в этой книге должны быть представлены разные мнения. Не существует универсального руководства по хаос-инжинирингу. Некоторые из представленных здесь мнений и советов не во всем совпадают, и это нормально. Мы не скрываем разногласия и не боимся противоположных точек зрения. Вы найдете здесь как общие темы, наподобие необходимости «большой красной кнопки» в инструментах, так и противоречивые мнения, например является ли хаос-инжиниринг формой тестирования или это своеобразные эксперименты[1].

Мы специально выбрали примеры из практики Slack, Google, Microsoft, LinkedIn и Capital One. Мы публикуем наиболее показательные истории и оставляем читателю возможность выбирать, какие из них наиболее соответствуют их собственному контексту. В сложных системах все решает контекст.

Мы начнем с главы 4 «Slack и островок спокойствия среди хаоса», в которой Ричард Кроули описывает особый подход к использованию методов хаос-инжиниринга в Slack. Оказавшись в ситуации, когда приходится сочетать устаревшие и новые системы, команда Slack успешно экспериментирует с новыми приемами хаос-инжиниринга. Ричард очень эффектно представил результаты своей работы на Game Days: «С помощью более двух десятков масштабных учений он обнаружил уязвимости, доказал безопасность новых и старых систем и указал верный путь многим командам разработчиков».

Джейсон Кахун в главе 5 «Google DiRT: Тестирование аварийного восстановления» раскрывает перед нами секреты методики под названием Google DiRT. Это одно из самых проработанных применений хаос-инжиниринга, поскольку Google уже давно использует программу DiRT. В этой главе рассматривается философия Google: «Просто надеяться, что система надежно сработает в экстремальных условиях, – далеко не лучшая стратегия. Вы должны предвидеть, что системы ломаются, проектировать их с учетом поломки и постоянно доказывать, что эти проекты остаются работоспособными».

---

[1] Авторы книги считают, что хаос-инжиниринг является разновидностью экспериментов. Некоторые из соавторов не согласны и используют термин «тестирование». См. раздел 3.1.1 в главе 3.

Джейсон также подчеркивает истинную ценность долгосрочной программы экспериментов, о которой мы уже говорили: «DiRT – это не способ ломать вещи только ради их разрушения; его ценность заключается в обнаружении потенциальных отказов, о которых вы еще не подозреваете».

«К сожалению, все пошло не так, как планировалось», – это характеристика большинства сюрпризов, с которыми мы сталкиваемся при эксплуатации масштабных систем. В главе 6 «Вариативность и приоритеты экспериментов в Microsoft» Олег Сурмачев дает очень структурированный взгляд на то, как расставить приоритеты в экспериментах. Потенциальные последствия инцидентов имеют первостепенное значение и сопровождаются множеством соображений, представленных в этой главе. Иногда, прежде чем искать «неизвестные события / неожиданные последствия», полезно как следует поработать над устранением известных слабостей, повысить надежность системы и избавить себя от ненужных экспериментов.

В главе 7 «LinkedIn и забота о пользователях» Логан Розен подчеркивает важность обслуживания клиентов. К счастью, существует множество стратегий по минимизации радиуса поражения и потенциального воздействия на клиентов во время экспериментов с хаосом. Логан знакомит нас с проектами LinkedIn, в которых реализовано несколько таких стратегий. «Хотя некоторые незначительные последствия обычно неизбежны, очень важно свести к минимуму дискомфорт, который эксперименты с хаосом могут причинить вашим конечным пользователям, и разработать простой план возвращения к нормальному состоянию».

Вторую часть книги завершает глава 8 «Capital One и адаптация хаос-инжиниринга к финансовым сервисам» Раджи Чокайяна. Банковский холдинг Capital One использует хаос-инжиниринг на протяжении многих лет. Раджи задокументировал развитие дисциплины от небольших ручных операций до сложных корпоративных инструментов, которые они теперь используют. Эволюция происходила в условиях жестко регламентированных операций и контроля результатов: «В банковском деле контроль и аудит важны ничуть не меньше, чем способность разрабатывать уникальные эксперименты».

Начиная рассказ о пяти вариантах практического применения, мы надеемся показать, что хаос-инжиниринг одновременно достаточно зрелый, чтобы работать с ценностями на отраслевом уровне, и достаточно молодой, чтобы сохранять гибкость применения и разнообразие интерпретаций.

Глава 4

Slack и островок спокойствия среди хаоса

Автор главы: **Ричард Кроули**

Откуда вдруг берется хаос-инжиниринг, если ваша компания никогда его не практиковала? Попытка ввести хаос в системы, разработанные с учетом того, что компьютеры могут и должны служить долго и бесперебойно, выглядит неразрешимой задачей. Сложные системы, созданные по принципу нерушимой надежности, как правило, менее приспособлены к хаотичным мимолетным изменениям в базовых компьютерах, чем их облачные потомки. Такие системы, как правило, очень хорошо работают в оптимальных условиях, но в случае сбоя рушатся с катастрофическими последствиями.

Вы можете быть счастливым обладателем именно такой системы. В ней не предусмотрено присутствие хаоса, но, нравится вам это или нет, хаос приходит сам по мере того, как увеличивается масштаб системы, и текущая бизнес-ситуация требует работать больше, быстрее и надежнее. Сейчас нет времени на переписывание – система и так находится под давлением. Применение новых методов хаос-инжиниринга к старым системам может лишь ухудшить ситуацию. Вам нужна другая стратегия.

В этой главе описана одна из возможных стратегий безопасного систематического тестирования сложных систем, которая использует контролируемые отказы и фрагментирование сети и вовсе не обязательно должна быть основана на хаос-инжиниринге. Продуманный процесс проверок, а не автоматизация как таковая, помогает вашей команде понять, насколько уязвимо ваше программное обеспечение, подталкивает к доработкам и позволяет предвидеть неполадки, которые случаются в системе. Этот процесс активно используется в Slack с начала 2018 года. Более двух десятков проведенных масштабных учений выявили уязвимости, укрепили безопасность новых и старых систем и показали хороший пример многим командам разработчиков.

Первый шаг, однако, заключается в том, чтобы убедиться, что ваша система хотя бы теоретически готова выдержать ошибки, с которыми вы ожидаете столкнуться…

4.1. Настройка методов хаоса под свои нужды

Инструменты и методы, которые вы используете для модернизации или создания собственных облачных сред и повышения надежности и доступности, подходят и для того, чтобы сделать систему более отказоустойчивой. Давайте их рассмотрим.

4.1.1. Подходы к проектированию старых систем

Существующие системы, особенно старые, в отличие от новых систем, предполагают, что отдельные компьютеры работают долго и надежно. Это простое предположение лежит в основе многих систем, которые не прощают сбои. Оно родилось в эпоху, когда приходилось избегать использования дорогостоящих запасных компьютеров, и с тех пор это отразилось на устройстве подобных систем.

Когда компьютеров было мало, их, как правило, оснащали операционной системой и всеми аксессуарами один раз, вскоре после приобретения, и обновляли на месте в течение всего срока службы. Процесс начальной установки программного обеспечения мог быть неплохо автоматизирован, особенно если к загрузочному доку одновременно подключали много компьютеров, но процесс обновления все равно выполняли вручную. В небольших системах доля ручного труда при начальном развертывании была намного больше.

Борьба с отказами, как правило, тоже происходила вручную, в виде действий оператора, который выбирал подходящий ответ на какой-то сбой или отклонение от нормальной работы. В самых старых системах период между отказом и аварийным переключением означал недоступность услуги для клиента. Считалось, что отказоустойчивые системы встречаются настолько редко, что не нуждаются в автоматизации, а в некоторых случаях даже в документации и обучении персонала.

Резервное копирование и восстановление – это еще одна область, в которой существующие системы могут отставать от современного уровня техники. К счастью, даже в самых старых системах обычно создают резервные копии. Но это не означает, что резервные копии гарантируют восстановление системы или что система будет восстановлена быстро. Как и в случае с отказами, восстановление из резервной копии когда-то было редким событием, которое явно не нуждалось в автоматизации.

Мы охотно миримся с возможными последствиями маловероятных событий – ведь они могут никогда не произойти! Существующим системам, созданным при попустительстве к редким рискам, трудно справиться с ситуацией, когда вместе с масштабом растет количество сбоев или последствия становятся неприемлемыми для бизнеса.

Для полноты картины я хочу кратко остановиться на *монолитной архитектуре*. Нет явной границы, после пересечения которой система считается монолитной – это относительное понятие. Монолитные системы по своей

природе не являются более или менее отказоустойчивыми, чем *сервис-ориентированные архитектуры*. Однако временами их сложнее модифицировать из-за большой удельной широты охвата, сложности постепенных изменений и склонности к большому радиусу поражения при сбое. Может быть, вы решите раздробить свой монолит, а может и нет. Отказоустойчивость достижима в обоих случаях.

4.1.2. Подходы к проектированию современных систем

Напротив, архитектура современных систем предполагает, что отдельные компьютеры постоянно подключаются, отключаются и обновляются. Из этого подхода вытекает много последствий, но, пожалуй, наиболее важным является то, что системы разрабатываются для одновременной работы на n компьютерах и продолжения работы в случае сбоя одного из них, когда остается только n – 1 компьютеров.

Критически важную роль играют проверки работоспособности, которые являются довольно глубокими для выявления проблем, но достаточно мелкими, чтобы избежать каскадных сбоев из-за зависимостей службы. Они удаляют неисправные компьютеры из службы и во многих случаях автоматически инициируют замену.

Замена экземпляров – провайдеры облачных сервисов обычно называют отдельные компьютеры экземплярами – это мощная стратегия современных систем. Она обеспечивает отказоустойчивость, которую я только что описал, а также стандартные режимы обновления, такие как параллельное *blue/green-развертывание*. А в системах хранения данных замена экземпляров является основанием для частого автоматического тестирования пригодности к восстановлению из резервных копий.

Еще раз хочу подчеркнуть, что даже монолитная система не исключает возможности использования современных приемов проектирования. Тем не менее существует проверенное архитектурное решение – представить новую функциональность как сервис, который взаимодействует с существующим монолитом.

4.1.3. Предварительная подготовка отказоустойчивости

В дополнение к средам разработки и отладки эксперименты по хаосу обязательно должны выполняться на производстве, и вы должны быть уверены, что влияние на клиентов будет незначительным, если оно вообще будет. Существует несколько важных изменений, которые вы можете внести, если какая-либо из ваших систем основана на старой архитектуре.

Прежде всего поддерживайте запасные мощности в сети. Наличие дополнительных компьютеров во время нормальной работы является первым

шагом к отказоустойчивости (и компенсирует больше видов аппаратных сбоев, чем RAID, который охватывает только жесткие диски, или постепенное снижение качества обслуживания, что может быть невозможно в вашем конкретном приложении). Используйте эту свободную мощность для обслуживания перенаправленных запросов, когда один или несколько компьютеров работают со сбоями.

Если у вас есть свободные ресурсы, подумайте, как автоматически отключить неисправные компьютеры из службы (прежде чем погрузиться в хаос-инжиниринг). Не останавливайтесь на автоматическом удалении, а перейдите к автоматической замене. Здесь облако дает некоторые явные преимущества. Легко (и весело) увлечься оптимизацией использования экземпляров, но большинству систем подойдет базовая реализация автоматического масштабирования, которая заменяет экземпляры по мере выпадения, просто поддерживая постоянное число работающих экземпляров. Автоматическая замена экземпляров должна быть надежной. Время запуска запасного экземпляра должно быть меньше, чем среднее время между отказами.

Некоторые системы, особенно те, которые хранят данные, могут различать лидера и зависимые службы. Опять же, легко (и весело) увлечься алгоритмами выбора лидера и достижения консенсуса, но и здесь вполне достаточно реализации, которая просто удерживает человека от критических действий. Внедрение автоматического перехода на другой ресурс – это идеальное время для проверки тайм-аута и политики повторных попыток в зависимых службах. Вам следует подбирать короткие, но разумные тайм-ауты, которые достаточно велики, чтобы завершить автоматический переход на другой ресурс, и использовать повторные попытки с экспоненциально нарастающим интервалом.

Мозговые штурмы, в которых ваша команда обсуждает детали ожидаемого поведения системы при наличии какой-либо ошибки, полезны для того, чтобы убедиться, что система готова к эксплуатации. Этой академической уверенности явно недостаточно в случае сложной системы, но единственный способ добиться истинной уверенности – это вызвать сбой в производстве. В оставшейся части этой главы я расскажу, как мы в команде Slack делаем это безопасно.

4.2. Disasterpiece Theater

Я называю этот процесс безопасного тестирования Disasterpiece Theater[1]. Когда вы конкурируете с другими проблемами за бесценное время ваших коллег-инженеров и просите их изменить способ разработки и эксплуатации программных систем, слегка безумный бренд помогает обратить на себя

[1] Disasterpiece Theater (театр бедствий) – это комедийная телевизионная программа, которая в начале 1980-х годов транслировалась на территорию США из Мексики. Большинство выпусков снято в стиле цирковой клоунады. – *Прим. перев.*

внимание. Disasterpiece Theater был впервые представлен на форуме про отказоустойчивые системы. Это постоянно действующая серия экспериментов, когда мы регулярно собираемся вместе и намеренно устраиваем сбой какой-то части Slack.

4.2.1. Цели экспериментов

Каждый эксперимент в Disasterpiece Theater немного отличается от предыдущих, он извлекает различные приемы из глубин нашего опыта, подогревает разные страхи и влечет за собой разные риски. Все эксперименты тем не менее сводятся к одним и тем же фундаментальным целям.

Если вынести за скобки самых преданных приверженцев программного обеспечения, предназначенного только для сбоев, большинство систем развертываются намного чаще, чем падают. Разрабатывая Disasterpiece Theater, мы уделяем пристальное внимание тому, насколько точно среда разработки соответствует производственной среде. Важно иметь возможность тестировать все изменения программного обеспечения в среде разработки, но намного важнее, чтобы там можно было отрабатывать сбои. Ценность Disasterpiece Theater в том, что он заставляет исправлять недостатки, и впоследствии эти усилия окупают себя при каждом запуске набора тестов и каждом цикле развертывания.

Более прямое и очевидное назначение управляемых сбоев заключается в обнаружении уязвимостей в наших рабочих системах. Планирование, которое предваряет эти эксперименты, помогает снизить (хотя и не полностью) риск того, что любая неизвестная уязвимость в конечном итоге повлияет на клиента. Мы ищем уязвимости в отношении доступности, корректности, управляемости, наблюдаемости и безопасности.

Disasterpiece Theater – это *непрерывная* серия экспериментов. Как только эксперимент выявляет уязвимость, мы планируем повторное выполнение, чтобы убедиться, что усилия по исправлению принесли результат, так же, как вы повторно запускаете набор тестов, чтобы убедиться, что вы исправили ошибку, из-за которой тест не прошел. В более общем плане эксперименты проверяют архитектуру системы и предположения, которые в нее заложены. Со временем сложная система развивается и может непреднамеренно опровергнуть предыдущее предположение, сделанное в зависимой части системы намного раньше. Например, время ожидания ответа службы может внезапно оказаться недостаточным, если эта служба развернута в нескольких облачных сервисах. Организационный и системный рост снижает точность любой модели системы; эти факторы сводят на нет предположения, сделанные при разработке системы. Регулярная проверка отказоустойчивости помогает организации сохранять актуальность предположений.

4.2.2. Антицели

Как видите, Disasterpiece Theater предназначен для обеспечения сопоставимой надежности в среде разработки и производства, побуждения к повыше-

нию надежности и демонстрации отказоустойчивости системы. Я считаю, что не помешает четко знать, какие эксперименты лучше не выполнять и какими инструментами лучше не пользоваться.

Один размер подходит не всем, поэтому я решил, что в случае Slack эксперименты Disasterpiece Theater должны быть нацелены на минимизацию вероятности возникновения производственного инцидента. Slack – это сервис, который малые и средние предприятия используют для ведения своего бизнеса; очень важно, чтобы сервис был для них постоянно доступен. Иными словами, у Slack нет права на ошибку, причиняющую клиенту ущерб в результате одного из запланированных экспериментов. Вы можете менее строго относиться к ошибкам и допускать более высокий риск, и если вы будете эффективно использовать свои возможности, в конечном итоге будете учиться все быстрее и быстрее благодаря широким возможностям для экспериментов.

Еще одним ключевым приоритетом является сохранность данных. Это не означает, что системы хранения нельзя подвергать испытаниям. Скорее, это просто означает, что планы экспериментов должны предусматривать непредвиденные обстоятельства и гарантировать безусловную сохранность данных. Возможно, вам следует выбрать особые методы провокации сбоя, постоянно хранить резервную реплику базы данных или вручную создать резервную копию данных перед экспериментом. Какие бы преимущества ни сулил Disasterpiece Theater, не стоит терять данные клиента.

Disasterpiece Theater не работает наобум. При организации даже небольшого сбоя с незнакомыми (или почти незнакомыми) условиями ключевым условием является планирование. У вас должна быть подробная, достоверная гипотеза о том, что произойдет *до того*, как вы спровоцируете сбой. Регулярно собирайте всех специалистов и заинтересованных лиц в одной комнате или видеоконференции. Это помогает упорядочить планы, знакомит широкий круг инженеров с устройством системы и распространяет информацию о самой программе Disasterpiece Theater. Следующий раздел детально описывает процесс организации эксперимента от идеи до результата.

4.3. Процесс проверки по шагам

Каждый эксперимент в Disasterpiece Theater начинается с идеи. Или, точнее, с ощущения беспокойства. Оно может исходить от разработчика и давнего владельца системы, из случайного открытия, сделанного во время какой-то другой работы, из воспоминаний после краха системы – на самом деле откуда угодно. Вооружившись этим беспокойством и помощью одного или нескольких экспертов по рассматриваемой системе, опытный экспериментатор проведет нас через весь процесс.

4.3.1. Подготовка эксперимента

Вы и ваши единомышленники должны собраться в одной комнате или на одной видеоконференции, чтобы подготовить план эксперимента. Мой по-

рядок действий по подготовке эксперимента содержит следующие пункты, каждый из которых я опишу подробно.

1. Выбираем сервер или службу, которая будет вызывать сбой, режим сбоя и стратегию имитации этого режима сбоя.
2. Изучаем сервер или службу в разработке и производстве; убеждаемся, что мы способны симулировать сбой в среде разработки.
3. Определяем предупреждения, информационные панели, журналы и метрики, которые, как мы предполагаем, обнаружат сбой; если таковых нет, все равно пытаемся разработать имитацию отказа, определив промежуточные показатели.
4. Определяем избыточности и автоматические исправления, которые должны смягчить последствия сбоя, и перечни действий, которые необходимо выполнить в ответ на сбой.
5. Приглашаем наблюдать за экспериментом весь причастный персонал, особенно тех, кто дежурит в это время, и объявляем о предстоящем эксперименте в #deathpiece-theatre (специальный канал в Slack).

Я обнаружил, что для начала хватает часа, проведенного вместе, а окончательная подготовка может выполняться асинхронно. (Да, мы сами используем Slack для этого.)

Иногда беспокойство, которое побудило вас провести эксперимент, имеет вполне конкретную форму, и вы уже точно знаете, какую именно ошибку спровоцируете, например когда процесс проходит тест на проверку работоспособности, но тем не менее не отвечает на запросы. В остальных случаях есть много способов добиться желаемого отказа, и все они немного отличаются друг от друга. Как правило, самый простой способ сначала спровоцировать, а затем устранить неполадку – это остановить процесс. Следующим по сложности способом является прекращение работы экземпляра (особенно если он работает в облаке). Этот способ имеет смысл, если должна сработать автоматическая замена экземпляра. Для безопасной имитации отключения сетевого кабеля компьютера обычно используют `iptables(8)`. Этот способ отличается от остановки процесса и (иногда) завершения работы экземпляра, поскольку сбои проявляются в виде тайм-аутов вместо ошибки `ECONNREFUSED`. И тогда перед вами открывается дверь в бесконечный и порой ужасающий мир частичных и даже асимметричных нарушений связности сети, которые обычно моделируют с помощью `iptables(8)`.

Следом возникает вопрос о том, в каком месте системы применяется каждый из этих методов. Отдельные компьютеры – хорошее начало, но сразу начинайте думать над тем, как подобраться к целым стойкам, рядам, центрам обработки данных, зонам доступности или даже регионам. Масштабные сбои помогают обнаружить ограничения емкости и тесную связь между системами. Рассмотрите возможность сбоя на дистанции между балансировщиками нагрузки и серверами приложений, между некоторыми серверами приложений (но не всеми) и их базами данных резервного копирования и т. д. На этом этапе вы должны составить очень конкретный перечень шагов для выполнения или, что еще лучше, перечень команд для запуска.

Затем еще раз удостоверьтесь, что ваши действия действительно будут безопасными. Внимательно и беспристрастно взгляните на свою среду раз-

работки и подумайте, действительно ли в реальной жизни может возникнуть ошибка, которую вы хотите смоделировать. Также подумайте, хватит ли трафика в вашей среде разработки, чтобы проявить сбой и спровоцировать потенциально негативные эффекты, такие как исчерпание ресурсов в зависимой службе с плохо настроенными тайм-аутами, в оставшихся экземплярах пострадавшего сервиса или в связанных системах, таких как обнаружение сервисов или анализ журналов и агрегация данных.

Представьте на мгновение, что ваша среда разработки прекрасно переносит сбой. Уверены ли вы в том, что спровоцируете такую же ошибку в среде эксплуатации? Если нет, то вряд ли имеет смысл продолжать эксперимент – лучше придумать что-то новое в среде разработки. Если это так, то вы хорошо потрудились, чтобы создать среду, вызывающую доверие! Теперь найдите время, чтобы формализовать эту уверенность. Составьте перечень оповещений, которые должны сработать, когда вы провоцируете этот сбой, перечислите все панели мониторинга, журналы и/или метрики, которые, по вашему мнению, будут отображать сбой, а также те, которые должны остаться неизменными. Считайте, что это своего рода «заправка горючим» аварийного механизма реагирования на инциденты. Разумеется, вы не рассчитываете на худшее, но стоит убедиться, что если эксперимент пойдет не по плану, то время обнаружения опасности и восстановления системы будет близким к нулю. Я надеюсь, что в большинстве случаев эти журналы и метрики пригодятся вам только для подтверждения вашей гипотезы.

Но что это за гипотеза? Не поленитесь записать, чего именно вы и ваши единомышленники ожидаете. Опишите несколько точек зрения. Рассмотрите работу мониторинга исправности, балансировщиков нагрузки и обнаружения сервисов во время сбоя. Подумайте о судьбе запросов, которые были прерваны сбоем, а также запросов, которые поступили вскоре после этого. Как запрашивающее приложение узнает об ошибке? Сколько времени это займет? Повторяет ли свои запросы каждое приложение? Если да, то как агрессивно? Приведет ли комбинация этих тайм-аутов и повторных запросов к чему-то наподобие исчерпания ресурсов? Теперь расширьте охват моделируемой ситуации, чтобы включить в нее людей, и отметьте любые моменты, в которых вмешательство человека может быть необходимым или желательным. Перечислите любые перечни действий или документы, которые могут быть необходимы. (Это также играет роль «заправки топливом» процесса реагирования на инциденты.) Наконец, попробуйте количественно оценить возможное отрицательное воздействие на клиента, убедитесь, что оно будет достаточно маленьким и можно продолжать.

Завершите свою подготовку разработкой логистики эксперимента. Я рекомендую запланировать под этот этап как минимум три часа в большом конференц-зале. По моему опыту, на это редко уходят все три часа, но не помешает запас времени, если что-то пойдет не по плану. Если есть удаленные участники, используйте систему для видеоконференций с хорошим микрофоном, который охватывает всю комнату. Соберите единомышленников, всех доступных экспертов по используемой системе и ее клиентов, всех, кто находится на связи, и всех, кто хочет учиться. Эти упражнения отнимают драгоценные человеко-часы, что подчеркивает важность предварительной подготовки. Теперь, когда вы готовы, пришло время для Disasterpiece Theater.

4.3.2. Эксперимент

Я стараюсь придумать что-то особенное для каждого эксперимента, чтобы максимально прорекламировать значение программы Disasterpiece Theater внутри компании. Эта программа конкурирует за драгоценное время людей; они должны понимать, что время, потраченное на Disasterpiece Theater, приводит к созданию более надежной системы с вызывающей доверие средой разработки.

Вы должны назначить ответственного за ведение записей – своего рода стенографиста. (Исторически я сам играл эту роль во время учений в Slack's Disasterpiece Theater, но вам ничто не мешает принять иное решение.) Я рекомендую делать заметки в чате или каком-то подобном канале, который автоматически отмечает время каждого сообщения. Мы делаем заметки в собственном Slack-канале #disasterpiece-theater.

Если в какой-то момент во время учений вы обнаружите, что события развиваются не по плану или наблюдаются непредвиденные последствия для клиентов, немедленно прекратите эксперимент. Разберитесь в происходящем, перегруппируйтесь и попробуйте еще раз в другой день. Вы можете многому научиться, не превышая предел допустимой реакции на инцидент.

Дальше я перехожу к списку действий по проведению эксперимента, который еще длиннее и подробнее, чем план подготовки.

1. Убедившись, что никто не возражает, начинаю запись видеоконференции (если позволяет оборудование).
2. Просматриваю план подготовки и вношу в него необходимые изменения.
3. Объявляю разработчикам о начале эксперимента в #ops (канал в Slack, где мы обсуждаем изменения среды эксплуатации и инциденты).
4. Запускаю провокацию сбоя в среде разработки. Записываю время.
5. Получаю оповещения и проверяю информационные панели, журналы и метрики. Фиксирую время, когда они предоставляют убедительные доказательства отказа.
6. Если предусмотрено автоматическое восстановление, даю ему время на срабатывание. Записываю время срабатывания.
7. При необходимости выполняю инструкции по восстановлению сервиса. Регистрирую время и любые отклонения от инструкций, на которые пришлось пойти.
8. Принимаю решение о том, продолжать ли эксперимент в среде эксплуатации. Если нет, выкладываю сообщение в #ops, сворачиваю и останавливаю эксперимент. Если да, то перехожу к следующему этапу.
9. Сообщаю в канале #ops о переходе эксперимента в среду эксплуатации.
10. Провоцирую сбой в производстве. Записываю время.
11. Получаю оповещения и проверяю информационные панели, журналы и метрики. Фиксирую время, когда они предоставляют убедительные доказательства отказа.
12. Если предусмотрено автоматическое восстановление, даю ему время на срабатывание. Записываю время срабатывания.

13. При необходимости выполняю инструкции по восстановлению сервиса. Регистрирую время и любые отклонения от инструкций, на которые пришлось пойти.
14. Выкладываю краткое и понятное сообщение о результатах эксперимента в #ops.
15. Подвожу итоги.
16. Если велась запись, распространяю ее, как только она станет доступной.

Мне нравится иметь аудиозапись эксперимента, которую можно считать страховкой на случай, если стенографист упустил что-то важное или плохо отразил в заметках, сделанных в режиме реального времени. Тем не менее важно убедиться, что все участники эксперимента не возражают против записи. Сначала уточните и только потом включите запись, если это возможно.

Начните с тщательного анализа плана. Вполне вероятно, что некоторые участники впервые участвуют в эксперименте. Их уникальное видение может улучшить план. Обязательно учитывайте их мнение, особенно когда это повышает безопасность эксперимента или значимость результатов. Мы заранее публикуем планы в общем доступе и обновляем их с учетом изменений. Однако будьте осторожны, отклоняясь слишком далеко от плана, так как это может превратить безопасное и хорошо спланированное упражнение в полосу препятствий.

Когда план будет утвержден, объявите о проведении эксперимента в публичном месте, например в чате, где ожидаются обновления о работе системы, в списке рассылки для инженеров и т. д. В первом объявлении сообщите, что в среде разработки стартует эксперимент и непосредственные наблюдатели должны быть готовы принять участие. Взгляните на пример 4.1, в котором показано, как выглядит типичное объявление в Slack.

Пример 4.1. Типичное первое объявление о запуске Disasterpiece Theater в Slack

> **Ричард Кроули, 9:50 #disasterpiece-theater** стартует снова, и мы в основном планируем отключить сетевые кабели на 1/4 серверов каналов в среде разработки. Следите за новостями на канале или ждите моего объявления здесь, когда мы перейдем к среде эксплуатации.

Вот и наступил момент истины (по крайней мере, в среде разработки). Один из ваших единомышленников выполняет подготовленную команду, чтобы спровоцировать сбой. Ваш стенографист должен записать время.

С этого момента все участники команды (кроме стенографиста) вступают в действие. Соберите доказательства сбоя, восстановления и воздействия на смежные системы. Подтвердите или опровергните все детали вашей гипотезы. Отдельно отметьте, сколько времени заняло автоматическое исправление и что пережили ваши клиенты за это время. И если вам пришлось вмешаться, чтобы восстановить сервис, сделайте особенно подробные записи о действиях всех участников. Проследите за тем, чтобы стенографист фиксировал ваши наблюдения и публиковал скриншоты графиков, которые вы изучаете.

К этому времени ваша среда разработки должна была вернуться в устойчивое состояние. Подведите итоги. Если ваши инструменты автоматического

восстановления не обнаружили сбой или сами оказались неисправны, то, вероятно, здесь и нужно остановиться. Если сбой слишком заметен для клиентов (вы можете экстраполировать эффект из среды разработки) или длится слишком долго, остановитесь на этом этапе. Если вы оценили риск и решили прервать эксперимент, объявите об этом везде, где вы объявили о начале эксперимента. Взгляните на пример 4.2, где показано, как такое редкое отступление от плана выглядит в Slack.

Пример 4.2. Объявление о досрочном прекращении работы Disasterpiece Theater в Slack

Ричард Кроули 11:22 Disasterpiece Theater завершил работу в течение дня, даже не переходя в производство.

Когда эксперимент в вашей среде разработки идет по плану, вы можете объявить о переходе в среду эксплуатации. В примере 4.3 показано типичное объявление в Slack.

Пример 4.3. Типичное объявление о переходе Disasterpiece Theater в среду эксплуатации

Ричард Кроули 10:10 #disasterpiece-theater отработал два раунда в разработке. Теперь мы переходим к производству. В ближайшее время ожидается перераспределение группы каналов в кольце сервера каналов. Я сообщу еще раз, когда появятся подробности.

Это *настоящий* момент истины. Вся подготовка и эксперименты в среде разработки привели вас к тому моменту, когда один из ваших единомышленников должен спровоцировать сбой в среде эксплуатации, выполняя заранее подготовленные действия или команды. В одних случаях последствия эксперимента будут ничтожными, в других – ужасающими. Прислушивайтесь к своим ощущениям – они подсказывают вам, где расположена зона риска вашей системы.

И снова все участники (за исключением стенографиста) начинают собирать доказательства отказа, восстановления и воздействия на смежные системы. На этот раз доказательства, как правило, гораздо более интересны – ведь они отражают трафик реальных клиентов. Подтвердите или опровергните свою гипотезу в среде эксплуатации. Наблюдайте, как система реагирует на ошибку, засеките время, когда сработает автоматическое восстановление. И конечно, если нужно вмешаться, чтобы восстановить обслуживание, сделайте это быстро и решительно – ваши клиенты рассчитывают на вас! На этом этапе проследите, чтобы стенографист фиксировал наблюдения и публиковал скриншоты графиков, которые вы изучаете.

Когда система вернется в устойчивое состояние, сообщите об этом по тем же каналам, где сообщали о проведении эксперимента. Очень хорошо, если вы сможете сразу сделать какие-то выводы о результатах эксперимента, но, как минимум, объявление должно информировать товарищей по команде о внесении изменений в производство с учетом ситуации.

Прежде чем все участники разойдутся по своим делам, уделите время для небольшого обсуждения по горячим следам. Постарайтесь сразу понять либо, как минимум, задокументировать все неполадки или неоднозначности, обнаруженные в ходе эксперимента.

4.3.3. Подведение итогов

Вскоре после эксперимента, пока воспоминания еще свежи и точны, я подвожу итог эксперимента – только факты – для широкой аудитории. Подобный обзор итогов помогает сделать вывод, почему используемый режим отказа важен, как системы перенесли (или не допустили) отказ и что это значит для клиентов и бизнеса. А еще эти выводы хорошо убеждают остальную часть компании в необходимости продолжения экспериментов. Мой оригинальный опросный лист Disasterpiece Theater выглядит следующим образом:

- Сколько времени прошло между обнаружением и восстановлением?
- Заметили ли пользователи? Откуда мы это знаем? Как мы можем добиться ответа «нет»?
- Какую работу компьютеров пришлось выполнять людям?
- Что мы упустили?
- Где наши панели управления и документация сработали не так?
- Что нам нужно делать чаще?
- Что должны делать инженеры техподдержки, если это произошло неожиданно?

Мы размещаем ответы на эти вопросы в канале Slack или в итоговом документе, доступном в Slack. Совсем недавно мы начали делать аудиозаписи экспериментов и архивировать их для потомков.

Затем ведущий предлагает выводы и рекомендации, вытекающие из эксперимента. Ваша задача как хозяина ситуации состоит в том, чтобы сделать эти выводы и дать рекомендации по повышению надежности системы и качества среды разработки, основываясь на доказательствах, беспристрастно представленных в итоговом документе. Эти рекомендации приобретают особое значение, когда эксперимент прошел не по плану. Если даже самые опытные умы неправильно или не полностью поняли систему до начала эксперимента, вполне вероятно, что все остальные заблуждаются еще больше. Это ваша возможность улучшить коллективное знание.

Подведение итогов дает вам еще одну возможность рассказать людям о типах сбоев, которые *могут* произойти в системе, и о методах преодоления последствий, которые вы используете. Фактически это намного лучше, чем публиковать подробный анализ уже случившихся *реальных* инцидентов.

4.4. Как развивался Disasterpiece Theater

Disasterpiece Theater изначально задумывался как дополнение к процессу реагирования на инциденты и даже форум для практического реагирования на инциденты. Ранние наборы экспериментов включали довольно много

сбоев, которые были хорошо проработаны даже в то время, когда требовалось активное вмешательство человека. С наличием устаревших ситуаций, в принципе, можно было смириться, потому что они просто исключались естественным образом по мере развития систем.

Спустя год после внедрения Disasterpiece Theater мы в Slack перестали запускать эксперименты, в которых запланировано вмешательство человека, хотя тем не менее оставались случаи, когда вмешательство людей казалось неизбежным. Мы придумали интересный тренинг для выработки опыта управления сбоями: «Обед с управлением инцидентами». Это игра, в которой группа людей пытается прокормить себя, следуя процессу реагирования на инцидент. Они периодически достают карточки, на которых описан неожиданный поворот событий, такой как внезапное закрытие ресторана, аллергия или придирчивые едоки. Благодаря этой практике и предварительному обучению Disasterpiece Theater больше не оставляет пробелы в автоматизации процесса.

Disasterpiece Theater развивался и в других направлениях. Самые ранние варианты были полностью сосредоточены на результатах и упускали образовательные возможности. Подведение итогов и особенно письменные отчеты, выводы и рекомендации значительно повысили образовательную ценность методики. Добавление аудио- и видеозаписей помогает будущим разработчикам еще глубже и быстрее изучить предмет.

Удаленному участнику видеоконференции может быть сложно следить за тем, кто говорит, особенно если вместо изображения говорящего кто-то делится экраном своего компьютера. Вот почему в Disasterpiece Theater я не рекомендую включать проекцию экранов компьютеров. С другой стороны, иногда бывает чрезвычайно полезно вместе посмотреть на один и тот же график. Я продолжаю искать правильный баланс между общим доступом к экрану и видеоконференцией, которая создает эмоциональный комфорт для удаленных участников.

Наконец, мой исходный перечень действий требовал от владельцев системы придумать синтетические запросы, которые можно выполнять в коротком цикле, чтобы визуализировать ошибки и допущения. Этот подход оказался менее полезным по сравнению с хорошо настроенной приборной панелью, которая отображает долю сбойных запросов, гистограмму задержки и т. д. Я удалил это нерациональное требование из перечня, чтобы упростить процесс.

Это, конечно, не последняя доработка Disasterpiece Theater в Slack. Если вы применяете аналогичный процесс в своей компании, обратите внимание на шероховатости процесса и постарайтесь понять, кто не получает выгоды от более активного участия.

4.5. Как получить одобрение руководства

Повторю еще раз: ваше умение убеждать людей является ключевым навыком. Вы можете начать с бодрого риторического вступления: «Приветствую вас, технический директор и вице-президент по развитию. Разве вы не хо-

тите знать, насколько хорошо наша система выдерживает отказ носителя базы данных, фрагментацию сети и перепады питания?» Нарисуйте картину, которая включает в себя пугающую неизвестность.

А потом донесите до них неудобную правду. Единственный способ понять, как система справляется со сбоем в работе, – это испытать сбой в работе. Впрочем, я должен признать, что эту идею было невероятно легко продать руководителям Slack, потому что они уже верили, что это правда.

В целом, однако, любой ответственный руководитель захочет видеть доказательства того, что вы эффективно и ответственно управляете рисками. Disasterpiece Theater разработан специально для того, чтобы удовлетворить это пожелание. Подчеркните, что ваши эксперименты тщательно планируются и контролируются, чтобы максимизировать обучение и минимизировать (или, что еще лучше, полностью устранить) влияние на клиента.

Затем спланируйте первый эксперимент и покажите руководству результаты, наподобие тех, которые приведены дальше.

4.6. Результаты

Я провел десятки экспериментов в Slack. Большинство из них пошли относительно по плану, расширяя наше доверие к существующим системам и подтверждая правильную работу новых компонентов. Некоторые эксперименты, однако, выявили серьезные недостатки в отношении доступности или правильности архитектуры Slack и дали нам возможность исправить их, прежде чем пострадали клиенты.

4.6.1. Избегайте несогласованности кеша

В первый раз, когда Disasterpiece Theater обратил свое внимание на Memcached, он должен был продемонстрировать, что автоматическая замена экземпляра в производстве работает должным образом. Эксперимент был простым: мы отключили экземпляр Memcached от сети и наблюдали за тем, как сработала автоматическая замена. Затем восстановили сетевое подключение и дождались завершения замены экземпляра.

В ходе проверки плана мы обнаружили уязвимость в алгоритме замены экземпляра и вскоре подтвердили существование уязвимости в среде разработки. Как это было изначально реализовано, если экземпляр теряет аренду диапазона ключей кеша, а затем получает ту же самую аренду обратно, он не сбрасывает свои записи в кеше. Однако в нашем случае заменяющий экземпляр успевал обратиться к этому диапазону ключей кеша в период отключения исходного экземпляра, следовательно, данные в исходном экземпляре устарели и, возможно, неверны.

Во время эксперимента мы устранили возникшую неполадку, вручную очистив кеш в нужный момент, а затем, сразу после эксперимента, изменили алгоритм и протестировали его снова. Без этого мы бы, наверное, долго не подозревали о риске повреждения кеша.

4.6.2. Пробуйте и еще раз пробуйте

В начале 2019 года мы запланировали серию из десяти экспериментов, чтобы продемонстрировать устойчивость Slack к зональным сбоям и фрагментации сети в AWS. Один из этих экспериментов касался сервера каналов – системы, отвечающей за рассылку вновь отправленных сообщений и метаданных всем подключенным клиентским сокетам Slack. Цель состояла в том, чтобы просто отделить 25 % серверов каналов от сети, дабы убедиться, что сбой будет обнаружен, а экземпляры будут заменены запасными.

При первой попытке создать нарушение связности сети не удалось полностью учесть оверлейную сеть, которая обеспечивает прозрачное транзитное шифрование. По сути, мы изолировали каждый сервер каналов намного больше, чем предполагалось, создавая ситуацию, более близкую к полному отключению серверов от сети, чем к нарушению связности. Мы сразу остановили эксперимент, чтобы перегруппироваться и получить ожидаемое нарушение сети.

Вторая попытка показала многообещающие результаты, но также была прекращена до выхода на производство. Этот эксперимент, по сути, дал положительный результат. Он показал, что сервис Consul весьма искусно обходит нарушения целостности сети. Это вселило уверенность в сервисе, но провалило эксперимент, потому что ни один из серверов каналов так и не вышел из строя.

В третьей (и последней) попытке мы использовали весь арсенал правил iptables(8) и смогли выделить сегмент из 25 % серверов каналов. Consul быстро обнаружил неполадку, и их заменили. Самое главное, что нагрузка, которую эта масштабная автоматизированная реконфигурация взвалила на Slack API, находилась в пределах возможностей данной системы. В конце долгого пути мы получили только положительные результаты!

4.6.3. Невозможность как результат

Бывали у нас и отрицательные результаты. Однажды, отвечая на инцидент, мы были вынуждены внести и развернуть изменение кода, чтобы выполнить изменение конфигурации, потому что наша собственная система под названием Confabulator, предназначенная для этого изменения конфигурации, просто не сработала. Я подумал, что этот инцидент заслуживает дальнейшего изучения. Мы с коллегами спланировали эксперимент, напрямую имитирующий ситуацию, с которой столкнулись. Можно было бы отделить Confabulator от сервиса Slack, но в таком случае это не соответствовало бы реальности. Затем мы попытались внести изменения в конфигурацию без ручных операций.

Мы уверенно воспроизвели ошибку и начали анализировать наш код. Нам не потребовалось много времени, чтобы найти проблему. Разработчики системы предположили ситуацию, в которой сам Slack не работает и, таким образом, невозможно проверить предлагаемое изменение конфигурации; они предложили аварийный режим, который пропускает эту проверку. Однако

как в обычном, так и в аварийном режимах предпринималась попытка опубликовать уведомление об изменении конфигурации в канале Slack. У этого действия не было тайм-аута, но зато был тайм-аут в общем API конфигурации. В результате даже в аварийном режиме запрос никогда не сможет выполнить изменения конфигурации, если сам Slack не работает. С тех пор мы внесли много улучшений в процедуру развертывания кода и обновления конфигурации и провели аудит тайм-аута и политик повторных попыток в этих критических системах.

4.7. Вывод

Открытия, сделанные во время этих экспериментов, были возможны только потому, что Disasterpiece Theater обеспечил четкий процесс тестирования отказоустойчивости наших систем в среде эксплуатации.

Эксперименты Disasterpiece Theater – это тщательно спланированные сбои, которые группа экспертов вносит в среду разработки, а затем, если все в порядке, в среду эксплуатации. Это помогает минимизировать риск, связанный с тестированием отказоустойчивости, особенно когда она построена на допущениях, давным-давно сделанных в старых системах, которые, возможно, изначально не были разработаны с учетом отказоустойчивости.

Этот процесс предназначен для обоснования инвестиций в среду разработки, которая точно соответствует среде эксплуатации, и для повышения надежности во всех сложных системах.

Благодаря регулярному выполнению экспериментов Disasterpiece Theater ваша организация и ваши системы станут лучше. Вам следует иметь обоснованную уверенность в том, что решения, стабильно работающие в среде разработки, будут столь же стабильны в среде эксплуатации. Вы должны иметь возможность регулярно проверять старые предположения, чтобы предотвратить загнивание системы. И ваша организация должна лучше оценивать риски, особенно когда речь идет о системах, требующих вмешательства человека для восстановления после сбоя. Но самое главное, Disasterpiece Theater должен служить убедительным стимулом для постоянных вложений в отказоустойчивость.

Об авторе

Ричард Кроули – инженер, технический руководитель и менеджер по восстановлению систем. Он интересуется производственным обучением, распределенными системами и реалиями масштабного производства. Он терпимо относится к компьютерам, любит велосипеды, не очень хорошо работает с открытым исходным кодом и живет в Сан-Франциско со своей женой и детьми.

Глава 5

Google DiRT: тестирование аварийного восстановления

Автор главы: **Джейсон Кахун**

«Надежда – это не стратегия». Это девиз команды Google Site Reliability Engineering (SRE), прекрасно отражающий философию хаос-инжиниринга. Можно спроектировать сколь угодно устойчивую систему, но пока вы на практике не смоделируете условия масштабного отказа, будет оставаться риск того, что реальность не соответствует ожиданиям. Программа Google DiRT (disaster recovery testing, проверка восстановления после сбоя) была основана инженерами по надежности систем в 2006 году, чтобы преднамеренно провоцировать отказы в критических технологических системах и бизнес-процессах, тем самым выявляя неучтенные риски. Создатели программы DiRT привели убедительный аргумент, что анализ чрезвычайных ситуаций на производстве становится намного проще, *когда это на самом деле не является чрезвычайной ситуацией*.

Аварийное тестирование помогает доказать устойчивость системы, когда отказы обрабатываются спокойно и без последствий, и выявляет риски надежности управляемым образом, когда дела идут не столь гладко. Выявление рисков надежности во время контролируемого инцидента позволяет проводить тщательный анализ и упреждающее смягчение последствий, в отличие от ожидания, пока проблемы проявят себя по стечению обстоятельств, когда серьезность проблемы и нехватка времени усугубляют ошибки и вынуждают принимать рискованные решения на основе неполной информации.

DiRT начался с того, что инженеры Google выполняли ролевые упражнения[1] наподобие «игровых дней» (game days), которые проводятся в других компаниях. Они особенно сосредоточились на моделировании того, как катастрофы и стихийные бедствия могут нарушить работу Google. Корпорация

---

[1] *Andrew Widdowson*. Disaster Role Playing // *Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Murphy, eds*. Site Reliability Engineering. Sebastopol, CA: O'Reilly, 2016. Chapter 28.

Google, несмотря на распределение рабочей силы по всему миру, занимает непропорционально большую площадь в районе залива Сан-Франциско, в месте, особенно подверженном землетрясениям. Сосредоточение такой большой внутренней инфраструктуры в одном опасном районе вызывает непростые вопросы: «Что произойдет, если кампус Маунтин-Вью и его сотрудники станут полностью недоступными в течение нескольких дней? Как это может нарушить наши системы и процессы?»

Изучение возможных последствий недоступности служб, размещенных в Маунтин-Вью, вдохновило создателей многих оригинальных тестов Google, но по мере того, как росло понимание ситуации (или, возможно, чувство стыда…), команды, заинтересованные в повышении надежности, начали использовать эксперименты DiRT в масштабах всей компании как возможность глубоко исследовать свои собственные сервисы. Со временем чисто теоретические «настольные» игры научили владельцев сервисов вводить реальные, но управляемые сбои (добавление задержки, отключение связи с «некритическими» зависимостями, обеспечение непрерывной работы бизнеса в отсутствие ключевых лиц и т. д.). Чем дальше, тем больше команд принимали участие в экспериментах. По мере расширения охвата экспериментов стало ясно, как много нужно уточнить и улучшить в общей архитектуре Google: неизвестные жесткие зависимости, ненадлежащая работа резервных стратегий, неэффективность защитных мер, недостатки в планировании малых и больших модернизаций, которые становятся очевидными постфактум, но практически невидимы заранее или проявляют себя только при «правильной» (или неправильной, в зависимости от того, как вы на это смотрите) комбинации неблагоприятных условий.

Развитие программы DiRT с самого начала не останавливалось ни на минуту, и на данный момент различными командами Google проведены многие тысячи экспериментов DiRT. В течение года проводятся крупномасштабные скоординированные мероприятия, и команды активно испытывают на прочность свои системы и самих себя. Определенный уровень участия в DiRT является обязательным для команд SRE и настоятельно рекомендован для владельцев сервисов по всей компании. Среди участников не только разработчики программного обеспечения: такие вспомогательные службы, как физическая безопасность, информационная безопасность, техподдержка центра обработки данных, связь, инженерные коммуникации, отдел кадров и финансы, – все бизнес-подразделения разработали и выполнили свои тесты DiRT.

В последние годы основное внимание уделялось разработке стандартизированных наборов автоматических тестов для сетевых и программных систем. Инженеры могут использовать готовые автоматические тесты «из коробки» для проверки поведения своей системы с учетом сбоев в общей инфраструктуре и системах хранения данных. Непрерывные автоматические тесты зорко следят за снижением надежности и постоянно подтверждают уровень обслуживания в экстремальных или нестандартных условиях. Эти тесты снижают барьер для входа и служат стартовой площадкой для более сложного тестирования отказоустойчивости конкретной архитектуры. Выгода от автоматизации даже простейших задач проявляется в том, что число

выполнений автоматических тестов превысило общее количество традиционных тестов DiRT на порядок, достигнув нескольких миллионов тестовых прогонов всего за несколько лет.

Надежность, которой славится Google, – это не волшебный дар. Чтобы добиться надежности, нужно оспаривать гипотезы о вашей системе, а также изучать необычные комбинации сбоев и готовиться к ним (в масштабах Google сбои с коэффициентом вероятности один на миллион происходят несколько раз в секунду). Просто надеяться, что система поведет себя надежно в экстремальных обстоятельствах, – не очень хорошая стратегия. Вы должны ожидать, что системы потерпят неудачу, проектировать их с учетом неудач и постоянно доказывать, что эти проекты все еще актуальны.

Легенда: поиск не работает, когда начальник уходит в отпуск

Давным-давно один высокопоставленный программист в Google создал конвейер индексации и утилиту поиска для репозитория исходного кода Google вместе с несколькими специализированными внутренними инструментами, которые полагались на эту утилиту поиска, чтобы делать крутые вещи. Все были довольны классными вещами, которые делали инструменты, и они служили людям верой и правдой в течение нескольких лет, пока обстоятельства не изменились. Одним из последствий модификации системы безопасности в Google стало значительное сокращение времени ожидания для входа пользователя в систему. Программисты начали замечать, что всякий раз, когда ведущий специалист на несколько дней покидает офис, утилита поиска кода и связанные с ней инструменты перестают работать.

Оказывается, задания по индексированию, поддерживающие эти инструменты, выполнялись годами как регулярные задачи на рабочей станции начальника. Новые изменения в политиках безопасности привели к тому, что срок действия его токенов авторизации истекал, если они не обновлялись каждый день. Даже после перемещения этих конвейеров индексации с рабочей станции на резервные производственные экземпляры все еще оставались скрытые зависимости от сетевых каталогов, которые отобразились как недоступные, когда кампус Маунтин-Вью был переведен в автономный режим. В конечном итоге проблема была выявлена в результате аварийного теста, имитирующего сбой внутренней сети.

Это, казалось бы, тривиальный пример, но он показывает, как популярность небольшого, но полезного сервиса может быстро выйти за рамки старых допущений и что изменяющийся технологический ландшафт (в этом примере – новая политика безопасности) может привести к появлению новых критических зависимостей для хорошо настроенного и в остальном стабильного сервиса.

5.1. Жизненный цикл теста DiRT

Каждый неавтоматизированный тест DiRT начинается с документированного плана тестирования. Для этого Google использует стандартный шаблон

документа, который эволюционирует со временем и собирает наиболее важную информацию, необходимую для оценки рисков и потенциальных преимуществ данного теста. Документ включает в себя конкретные процедуры выполнения, процедуры отката, известные риски, потенциальные воздействия, зависимости и учебные цели теста. План разрабатывает команда, которая будет проводить тестирование, а затем как минимум два независимых рецензента проводят экспертизу проекта. По крайней мере, один рецензент должен иметь технические знания в области, связанной с тестируемой системой. Испытания с высокой степенью риска и высокой эффективностью подвергаются более тщательному изучению, и, как правило, к ним привлекают гораздо больше экспертов-рецензентов. Обычно проект проходит через несколько раундов обратной связи, поскольку рецензенты задают уточняющие вопросы и запрашивают более подробную информацию. Стандарт предусматривает подробное описание процедуры тестирования и этапов отката, чтобы любой, кто имеет средние знания о тестируемой системе, мог выполнять необходимые операции.

После того как рецензенты одобрили тест, его включают в график выполнения, рассылают необходимые сообщения (если это предусмотрено планом), а затем в назначенное время запускают тест. После того как тест закончен, в дело вступает другой шаблонный документ, отчет о результатах, который фиксирует краткую ретроспективу теста. Опять же, этот документ заполняет команда, которая провела тест. В документе обязательно отмечают все примечательные или неожиданные явления, записывают действия и отклики (также помещенные в нашу систему отслеживания проблем) и формулируют основную ценность полученного опыта.

Шаблоны как исходного плана, так и отчета о результатах представлены в полуструктурированном формате, который разработан для последующего программного анализа. После завершения теста эти документы получают цифровые индексы и становятся доступными через внутренний веб-интерфейс. Архив ранее выполненных тестов и результатов DiRT может быть найден и получен любым сотрудником Google. Этот репозиторий пользуется популярностью при подготовке новых тестов или просто в тех случаях, когда инженеры хотят повторно изучить тесты, которые их команда могла выполнять в прошлом.

5.1.1. Правила взаимодействия

Существует несколько инструкций по тестированию в случае неисправностей, к которым Google пришел ценой болезненного опыта и одной или двух серьезных ошибок. Мы обнаружили, что более качественные тесты получаются при соблюдении нескольких заранее установленных и хорошо продуманных правил. Одинаковый уровень ожиданий у тестировщиков и остальной части компании обеспечит более предсказуемый процесс тестирования для всех заинтересованных сторон. Разработчики тестов знают, какие испытания явно выходят за пределы возможностей системы, а владельцы тестируемых систем приблизительно знают, какие беды на них свалятся.

Эти правила взаимодействия легли в основу философии, которая определяет уникальные свойства методики хаос-инжиниринга Google, и на ее развитие ушли годы. Несмотря на внешнюю простоту правил, они вовсе не были очевидны в начале нашего пути. Если ваша организация намерена разработать программу аварийного тестирования, найдите время для написания руководящего документа, содержащего исчерпывающие правила тестирования. В следующих разделах представлена обобщенная версия правил, используемых в Google; я буду ссылаться на них, приводя практические примеры тестов.

Тесты DiRT не должны оказывать разрушающего воздействия на цель уровня обслуживания, на внешние системы или пользователей

Это правило *не* означает, что не должно быть *абсолютно* никакого намека на проблему для внешних пользователей, а только то, что *цель уровня обслуживания* (service-level objective, SLO) не должна опускаться ниже предела, установленного владельцем службы. Если ваша служба начала понижать SLO настолько сильно, что это повлияет на внешних пользователей, значит, тест прошел с огромным успехом – вы обнаружили проблему, – но тест не обязательно должен продолжаться. Тесты DiRT не являются оправданием неприятностей, которые испытывает клиент. Уровень обслуживания должен быть важным руководящим фактором при разработке и мониторинге теста.

Ключевым моментом подхода Google к хаос-инжинирингу является использование «контролируемого хаоса» всегда, когда это возможно. Это означает, что ваш тест должен иметь хорошо продуманную и быстро выполняемую стратегию отката. Наличие страховочных средств или «большой красной кнопки» позволит вам остановить тест в случае необходимости. Несмотря на то что план теста был тщательно продуман, досрочно прерванные тесты считаются самыми полезными. У вас есть чему поучиться, анализируя тесты, которые превосходят ваши ожидания и оказывают большее влияние, чем запланировано.

Инфраструктура и внутренние потребители могут столкнуться с падением SLO ниже допустимого даже при работе в пределах допустимой нагрузки. Запросы потребителей услуг неизбежно растут со временем и на момент тестирования могут перестать укладываться в старые допуски и спецификации. Мы проводим аварийное тестирование большой инфраструктуры и общих служб, чтобы выявить подобные узкие места. Что можно сделать, если тест транзитивно воздействует на другие системы, приводя к резкому снижению SLO? Полная остановка теста при первых признаках чрезмерного воздействия на одну службу помешает вам обнаружить других внутренних пользователей, которые также плохо настроены, но не успели отреагировать. Для подобных ситуаций можно предусмотреть безопасный список, то есть встроить в тест механизм, который позволяет пострадавшим полностью исключить влияние теста. Это дает возможность избежать постоянных проблем со службами, которые недопустимо снижают SLO, но в целом продолжить тестирование для всех остальных. Безопасный список не является заменой стратегии отката и «большой красной кнопки», но должен быть под рукой вместе с другими средствами отмены в случае необходимости. В раз-

деле 5.1.4 более подробно рассказано о получении дополнительной информации из безопасного списка теста.

Чрезвычайные ситуации в производстве всегда имеют приоритет перед «чрезвычайными ситуациями» DiRT

Если во время теста DiRT происходит реальный производственный инцидент, выполнение DiRT следует прекратить и отложить, чтобы сосредоточиться на реальном инциденте. Если ваш тест борется за внимание персонала во время реального инцидента, он неявно нарушает первое правило взаимодействия, даже если не является причиной неполадок. Организаторы проверки должны заранее неоднократно сообщить потенциальным заинтересованным сторонам о проведении теста DiRT и внимательно отслеживать обычные каналы оповещения об инцидентах до начала теста, во время выполнения теста и после его завершения. Если наблюдаются первые признаки инцидента в производстве, организаторы должны оценить возможную взаимосвязь с тестом и обсудить это с другими дежурными респондентами. Реальный инцидент может оказаться достаточно незначительным, чтобы требовать остановки теста, но для проблемы средней степени тяжести, даже если она не вызвана непосредственно тестом DiRT, скорее всего, стоит остановить тест, чтобы спокойно устранить реальную проблему. Неполадки могут возникнуть еще до начала теста, и в этом случае, возможно, будет лучше перенести тест на другой день.

Выполняйте тесты DiRT прозрачно для окружающих

Как можно более четко разъясните коллегам, что «чрезвычайные ситуации» DiRT являются частью теста. Вы можете считать это правило следствием второго правила взаимодействия, согласно которому реальные инциденты имеют приоритет над ложными чрезвычайными ситуациями. Оператор системы не может эффективно игнорировать «аварийные» сообщения DiRT, уделяя внимание только реальной аварии, если он не знает, какая из двух аварийных ситуаций является тестом DiRT.

Сообщения, которые являются частью вашего теста (как предварительные, так и во время выполнения теста), должны иметь предельно ясный признак принадлежности к тесту. Сообщайте о своем тесте заранее, часто и однозначно. Сотрудники Google начинают темы электронных писем, связанных с тестом, с токена «DiRT DiRT DiRT», а в теле сообщения обычно содержится преамбула с тем же токеном «DiRT DiRT DiRT», метаинформация о продолжительности теста, контактная информация организатора и напоминание о необходимости сообщать о любых возникающих проблемах по стандартным каналам эскалации инцидентов.

Google создает необычные темы для групп тестов: атаки зомби, разумные вирусы искусственного интеллекта, вторжение инопланетян и т. д. Эта тема будет постоянно фигурировать в коммуникациях между сотрудниками. Искусственно добавленные страницы, предупреждения, ошибки, некорректные изменения конфигурации и провокационные изменения исходного кода содержат явные ссылки на тему. Эта практика служит двум целям. Во-первых, это весело! Придумать для своего теста впечатляющие названия из научной

фантастики – это возможность пошутить, проявить творческий подход и позитивно оформить скучную документацию. Менее очевидное преимущество ярких тем заключается в том, что они предоставляют четкие индикаторы для причастных пользователей и тех, кто отслеживает тесты. С первого взгляда ясно, что сообщение является частью теста DiRT, а это, в свою очередь, снижает вероятность перепутать тест с реальным инцидентом.

Минимизируйте стоимость, максимизируйте отдачу

Доводилось ли вам работать над системой, которая вызывает у вас душевную боль, является источником повторяющихся инцидентов и имеет в вашей организации репутацию ненадежного компонента? Такие системы должны быть модернизированы; им не нужны тесты на неполадки – они сами по себе катастрофа. А вот после доработки системы тест DiRT будет отличным инструментом для проверки надежности новой архитектуры или для отказа от старых опасений, если тесты покажут, что прежние недостатки больше не актуальны.

При разработке теста вы должны тщательно оценить затраты, связанные с проблемами пользователей и потерей репутации, в сравнении с тем, что вы узнаете в результате. DiRT предназначен не для того, чтобы ломать вещи только ради того, чтобы ломать их; его ценность заключается в обнаружении сбоев, *о которых вы еще не знаете*. Если вы можете взглянуть на систему и мысленно увидеть очевидные недостатки, то вам не нужны прикладные тесты. Если вы уже знаете, что система сломана, достаточно расставить приоритеты технических работ, чтобы устранить известные риски, а затем провести аварийное тестирование для оценки эффекта. Тщательно обдумайте значимость вашего теста. Хорошей практикой будет регулярная переоценка этого фактора, поскольку работы над системой не стоят на месте. Вопрос в том, следует ли продолжать выполнять особо эффективный тест. Если из теста больше нечему учиться, и он оказывает большее, чем ожидалось, влияние на отдельные части вашей системы, наверное, лучше его отменить.

Относитесь к аварийным тестам так же, как к настоящим отказам

Люди ожидают, что Google всегда будет работать как обычно, и относятся к любым сбоям, связанным с тестами, как к реальным сбоям. В сообщениях о тестах DiRT часто повторяют напоминание о том, что если тест затронул ваш участок работы, необходимо «отрабатывать проблему как обычно». Люди склонны быть пассивными при работе с инцидентами, если они знают, что «чрезвычайная ситуация» является частью тренировки[1]. Есть много причин, чтобы настаивать на выполнении стандартного протокола в любой ситуации. Процессы эскалации инцидентов являются важным каналом для сбора информации о влиянии теста и часто выявляют непредвиденные связи между системами. Если кто-то игнорирует проблему, потому что знает о проведении теста, вы упустите эту ценную информацию; нельзя упускать шанс для прямого общения и обучения.

---

[1] Эта любопытная поведенческая тенденция проявляется и в реальных чрезвычайных ситуациях, таких как пожары в зданиях, см.: *Lea Winerman.* Fighting Fire with Psychology. APA Monitor on Psychology, Vol. 35, № 8 (сентябрь 2004 г.).

5.1.2. Что следует проверить

Разработка вашего первого теста может быть непростой задачей. Если в последнее время у вас случались неполадки, это может послужить хорошим поводом для выбора компонентов, которые нужно протестировать в первую очередь. Временное воссоздание провоцирующих условий прошлых инцидентов поможет доказать, что ваша система научилась корректно справляться с предыдущими проблемами.

Начните с наводящих вопросов, и вы быстро поймете, в каком направлении двигаться дальше. Итак, какие системы поддерживают ваш бизнес ночью? Все ли вы знаете об отдельных поставщиках внешних данных или услуг? Существуют ли процессы, которые полностью зависят от людей в одном месте или от одного поставщика? Уверены ли вы на 100 %, что ваши системы мониторинга и оповещения сработают, когда это ожидается? Когда вы в последний раз выполняли переход на резервные ресурсы? Когда в последний раз вы восстанавливали свою систему из резервной копии? Проверяли ли вы поведение вашей системы, когда ее «некритические» зависимости недоступны?

Вовлеченность бизнеса Google в программу тестирования DiRT со временем становится все глубже и охватывает множество систем и процессов. Проверка сетевых и программных систем по-прежнему занимает львиную долю тестов, но Google внедряет аварийное тестирование во многих других областях, и в разработке тестов хорошо помогает нестандартное видение различных аспектов бизнеса. В следующих разделах описаны некоторые общие категории и подсказки, которые помогут вам начать работу.

Выполнение на уровне службы

Сценарий: необычно большие пики трафика увеличивают среднюю задержку совместной внутренней службы. К чести службы, ее увеличенная задержка практически не приводит к снижению обещанного уровня SLO.

Ограничение работы службы на заявленном минимально допустимом уровне обслуживания в течение длительного периода времени – это великолепный способ проверки правильности архитектуры распределенной системы, а также правильности выбора уровня SLO и управления ожиданиями потребителей услуг. Со временем слишком хорошая работа системы воспринимается как должное. Если ваша служба постоянно работает *слишком* хорошо, пользователи начинают думать, что качество услуги, которое они получают в последнее время, – это то, что вы обязались предоставлять всегда. Предоставляемый вами уровень обслуживания со временем становится неявным договором, независимо от опубликованных спецификаций вашей системы. В более общей форме изречение, которое называется *правилом Хайрама* (https://www.hyrumslaw.com/) и нравится нам в Google, обычно сводится к следующему:

> При достаточно большом количестве пользователей не имеет значения, что вы обещаете в контракте: всегда найдутся те, кто будет требовать уровень сервиса, исходя из того, что видят своими глазами.

Это правило не ограничивается качеством работы интерфейсов или служб; оно распространяется даже на надежность и рабочие характеристики.

Выполнение при недоступных зависимостях

Сценарий: половина ваших «мягких» зависимостей начинает возвращать ошибки, в то время как другая половина сообщает об увеличении задержки на порядок.

Если один или несколько некритических компонентов вашей системы выходят из строя, должна происходить постепенная деградация, когда служба продолжает работать в пределах SLO. Снижение уровня обслуживания некритических служб не должно приводить к каскадным сбоям или необходимости вмешательства оператора. Если в вашей системе слабо выражено различие между жесткими и мягкими зависимостями, эти тесты помогут вам найти состояние, при котором различие становится более заметным. Прямые критические зависимости вашей системы могут быть очевидны, но по мере усложнения архитектуры становится все труднее отслеживать второй и третий уровни зависимости или то, как комбинации некритических сбоев в сервисах, скрытых в вашем стеке, могут повлиять на способность системы обслуживать клиентов[1].

Разовый тест на внедрение сбоев даст вам детальную и точную картину устойчивости вашей системы к отказам в зависимостях, но это всего лишь моментальный снимок. Вы должны проводить такое тестирование регулярно. Зависимости системы не будут оставаться статичными в течение всего срока ее службы, и, к сожалению, большинству систем свойственно со временем наращивать критические зависимости. Критические зависимости могут возникать в самых неожиданных местах[2] и проникать в сложные системы, словно вор через окно, несмотря на усилия ваших лучших разработчиков. В Google в течение многих лет мы постоянно проводим одни и те же тесты по внедрению сбоев в системы и по-прежнему обнаруживаем новые проблемы по мере развития этих систем; не думайте, что тест отслужил свое, только потому, что он успешно сработал раньше.

Начните медленно, с постепенного ввода сбоев; нет необходимости «давить на газ» с самого начала. Вы можете начать с введения небольшой задержки в несколько зависимостей. Затем начинайте увеличивать значения задержки, долю пострадавших зависимостей или то и другое. Рим не был построен за один день; создайте петлю обратной связи для улучшения вашей системы и изучите предмет тестирования до такой степени, чтобы легко ответить на вопросы «Что, если ...?» о любой системе, с которой вы работаете.

Отсутствие людей

Сценарий: критически важная внутренняя система начинает выходить из строя, но ее старшие разработчики уехали на конференцию и недоступны.

---

[1] Отличную статью по этой теме вы найдете в: *Ben Treynor et al.* The Calculus of Service Availability // Communications of the ACM, Vol. 60, № 9 (Sept. 2017).

[2] Одним из общеизвестных публичных примеров было удаление пакета left-pad из репозитория Node Package Manager (www.npmjs.com) в марте 2016 года, что привело к прекращению работы сборок бесчисленных проектов JavaScript по всему миру.

Насколько ваша система подвержена *фактору лотереи*[1]? Носители технических знаний, обладатели лидерских навыков или знатоки бизнес-решений не должны становиться точкой отказа. Продуманы ли у вас первичные и резервные способы подключения необходимого персонала в случае чрезвычайной ситуации? Попробуйте выполнить несколько упражнений «Колесо невезения»[2] и случайно выбрать членов команды, которые будут считаться недоступными.

Развертывание и откат

Сценарий: в какую-то часть вашей производственной инфраструктуры выгрузили ошибочную конфигурацию, сбойный исполняемый файл или неправильное API сервер/клиент. Инженер техподдержки срочно вызван на работу и должен изолировать проблему и откатить изменения.

Греческий философ Гераклит сказал: «Перемены – единственное, что постоянно в жизни». Заманчиво мечтать о том, что систему можно довести до устойчивого надежного состояния и оставить там навсегда, исключив риски, связанные с новым кодом, архитектурой и системами. Подход «не трогай, пока работает» несет свои риски: система, которая не адаптируется к изменяющейся среде, подвержена возможному отказу в силу изменившихся внешних условий. С другой стороны, в любом изменении системы заложен неявный риск; развертывание нового программного обеспечения или конфигурации часто является предвестником проблем, связанных с простоями. Процедуры развертывания и соответствующие (возможно, даже более важные) процедуры отката заслуживают тщательного изучения командами разработчиков, которые хотят повысить устойчивость системы к отказам.

Процедура отмены изменений в среде производства должна быть хорошо документирована и отрепетирована. Как вы думаете, какие ошибки будут предотвращены автоматическим тестированием в конвейере развертывания, прежде чем обновление достигнет производства? Проводите ли вы регулярные проверки эффективности этого механизма? Ваша платформа мониторинга должна отслеживать изменения как в программной, так и в аппаратной конфигурации, чтобы их можно было легко ассоциировать с отображаемыми метриками. Когда текущее обновление окажется непригодным, вам понадобится возможность остановить развертывание и быстро выявить уязвимые системы. Вы можете разработать аварийный тест, который выполняет обе эти важные процедуры в ходе обычного обновления. Докажите себе, что вы можете приостановить обновление, оценить его ход и выбрать возобновление или отмену. В какой точке насыщения становится невозмож-

---

[1] Фактор лотереи – это метафора риска, возникающего из-за того, что информацию о ситуации не доводят до членов команды. Например, член команды выиграл в лотерею миллион и уволился без уведомления. Он также известен как «фактор автобуса».

[2] Многие команды Google SRE регулярно планируют теоретические упражнения, в которых они направляют члена команды с помощью подсказок, основанных на ранее произошедших отключениях и интересных инцидентах. Эти упражнения являются отличным способом обмена знаниями и опытом в команде. См.: *Beyer et al.* Site Reliability Engineering, Chapter 28.

ным полное отключение всех случаев, когда произошел сбой? Отключение или откат обновления на целевом подмножестве уязвимых машин может оказаться чрезвычайно полезным и легко выполняется в рамках аварийного теста с использованием идентичных копий двоичного файла, которые отличаются только номером версии или меткой выпуска. Если ваши процессы сборки обновлений занимают значительное время, очень важно иметь простой способ вернуться непосредственно к предыдущему выпуску без необходимости инициировать повторную сборку. Умение манипулировать системой обновления может сэкономить вам много времени и избавить от лишних осложнений в кризисной ситуации.

Процедуры управления инцидентами

Сценарий: критическая внутренняя служба начала возвращать 100 % ошибок. Очевидно, что это серьезный инцидент. Ваша команда должна координировать отладку вашего собственного сервиса и связываться с другими командами для выяснения причин, а также для публичного и внутреннего общения.

У Google есть высокоразвитый протокол управления инцидентами[1], основанный на Системе управления инцидентами, которую FEMA (Federal Emergency Management Agency, американский эквивалент российского МЧС. – *Прим. перев.*) использует для управления и координации действий по ликвидации стихийных бедствий. Наличие понятного протокола и ролей для управления инцидентами проясняет обязанности каждого участника, обеспечивая более эффективное сотрудничество и использование ресурсов. Роли предусматривают строго определенные каналы для двунаправленного потока информации, и это имеет решающее значение для избавления от какофонии сигналов, которые требуют немедленной реакции. Убедитесь, что у вашей команды есть четкий план работы в критических обстоятельствах, а у вашей организации есть руководящие указания по эскалации инцидента наряду с процедурами передачи обслуживания при длительных инцидентах.

Работоспособность ЦОД

Сценарий: утечка воды на полу центра обработки данных рядом с серверными стойками и другим критичным электрооборудованием.

Коллективы центров обработки данных Google имеют богатый опыт тщательного тестирования отказов физических систем, а также разработки надежных процедур управления инцидентами и планов действий в случае аварий. Инженеры ЦОД были одними из первых и самых активных сторонников программы DiRT в Google и сыграли важную роль в продвижении и развитии программы.

Аварийное тестирование в центре обработки данных может быть простым, как отключение питания одной стойки, или сложным, как переключение на резервное питание для всего сегмента. Рекомендую сначала провести теоретические ролевые тесты и подробно изучить нюансы восстановления

---

[1] *Andrew Widdowson.* Disaster Role Playing // *Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Murphy, eds.* Site Reliability Engineering. Sebastopol, CA: O'Reilly, 2016. Chapter 14.

физического оборудования, прежде чем переходить к небольшим, а затем и более крупным практическим тестам. Знакомы ли вам ситуации, когда технические специалисты центра обработки данных должны взаимодействовать с инженерами техподдержки производства во время аварийных ситуаций, которые влияют на нормальную работу? И наоборот, знают ли клиенты, как обращаться к операторам центра обработки данных, чтобы помочь в устранении проблем, которые могут быть связаны с оборудованием? Это отличные идеи для разработки тестов. Убедитесь, что все участники внутри и снаружи центра обработки данных имеют опыт работы с аварийными процедурами и знают, чего ожидать и как общаться между собой.

Управление вычислительными ресурсами

Сценарий: произошла непредвиденная потеря выделенных вычислительных ресурсов в регионе.

Владельцы современных распределенных сервисов должны отслеживать и прогнозировать изменяющиеся потребности в ресурсах, чтобы поддерживать баланс между необходимым резервированием и потенциальными затратами на чрезмерное выделение ресурсов. Аварийное тестирование может быть хорошим способом оценки базовых предположений о распределении ресурсов и нагрузки. Насколько значительным должен быть всплеск трафика, чтобы вы задействовали резервы на случай чрезвычайных ситуаций? Как быстро вы сможете увеличить пропускную способность при таком скачке трафика? Что, если ваш облачный провайдер не сможет удовлетворить ваши потребности в ресурсах в нужном регионе? Может ли ваша команда оказать экстренную помощь другому региону? Вы должны стараться достичь состояния, при котором уменьшение вычислительной мощности в одном местоположении и увеличение ее в другом, временно или постоянно, не представляет никакой опасности. Вы можете сначала провести тестирование с неполными ресурсами, отключив часть сети или сократив ресурсы в каком-то одном месте, ожидая, что автоматическое распределение нагрузки сработает соответствующим образом.

Обеспечение бесперебойности бизнес-процессов

Сценарий: ваш главный офис пострадал от стихийного бедствия; дочерние офисы должны оценить последствия, пытаться установить связь с пострадавшими подразделениями и координировать нормальные бизнес-процессы.

Даже если вокруг все рушится, по-прежнему необходимо принимать руководящие решения, а жизненно важные бизнес-процессы должны оставаться неизменными. Как работают цепочки согласования решений в чрезвычайных ситуациях, когда руководство недоступно? Кто утверждает чрезвычайные расходы? Кто отвечает за общественные коммуникации? За юридические решения? Наличие хорошо продуманного *плана обеспечения бесперебойности бизнес-процессов* является лишь первым шагом; вы должны убедиться, что сотрудники осведомлены об этом плане и знают, как выполнять его в чрезвычайной ситуации. Вы можете проверить реакцию человека, сообщив ему, что респондент вынужден работать с ноутбука или не имеет прямого под-

ключения к вашей корпоративной сети. Все команды Google выполняют ролевые упражнения по методике DiRT для тренировки навыков обеспечения бесперебойных бизнес-процессов.

Целостность данных

Сценарий: повреждение данных требует восстановления из последней резервной копии системы.

Вы можете быть уверены в резервном копировании ровно настолько, насколько тщательно проверили процедуру восстановления в предыдущий раз. Если вы никогда не пробовали полностью восстановить данные из резервной копии в производственную среду, как вы можете утверждать, что ваши процедуры восстановления верны или ваши резервные копии вообще работают? Если ваши процессы резервного копирования по какой-то причине перестали работать, сколько времени потребуется, чтобы это заметить? Простые тесты, такие как постепенная задержка процесса резервного копирования или временная запись файла резервной копии в другое место, могут послужить хорошей проверкой достоверности ваших знаний в этой области.

Устойчивость приложения к повреждению данных является еще одним важным аспектом целостности данных. Применяете ли вы *нечеткое тестирование*[1] внутренних служб? Нечеткое тестирование конечных точек API в архитектуре микросервисов – это отличный способ защитить сервисы от непредвиденных входных данных, которые приводят к сбоям («запросы смерти»). Существует множество инструментов с открытым исходным кодом, которые помогут вам начать тестирование, в том числе несколько популярных приложений, выпущенных Google[2].

Насколько устойчивы к задержкам ваши реплицированные хранилища данных? Задержки распространения и сетевые разрывы в последовательно согласованных системах хранения данных могут привести к нарушениям целостности данных, которые трудно распознать. Искусственно вызванная задержка репликации поможет вам предсказать поведение системы при обработке очереди репликации. Добавление задержки репликации может также выявить условия, при которых возникают гонки данных, которые иначе не отображаются в менее экстремальных условиях. Если вы уже выполняете нагрузочное тестирование в подготовительной среде, аварийный тест на репликацию может быть очень простым – например, запуск обычного пакета нагрузочного тестирования с отложенной или временно отключенной репликацией хранилища.

Сети

Сценарий: значительная часть вашей сети отключается из-за перебоев в работе сети.

---

[1] Википедия дает такое определение: «Нечеткое тестирование – это метод автоматического тестирования программного обеспечения, который предусматривает предоставление недействительных, неожиданных или случайных данных в качестве входных данных для компьютерной программы», https://oreil.ly/Erveu.

[2] Google предлагает два открытых инструмента: libprotobuf-mutator и ClusterFuzz.

Проблемы с сетевой инфраструктурой могут привести к частичному или полному отключению службы, сайта либо группы. Команды Google, отвечающие за пропускную способность сети, особенно полюбили программу DiRT и разработали множество тестов, включающих временные конфигурации брандмауэра, которые перенаправляют или полностью блокируют трафик. Отказы сети следует тестировать в разных масштабах, охватывающих все уровни: от рабочего стола отдельного человека или здания до вашего географического региона или всего центра обработки данных. Региональные сбои – реальная проблема в облачных архитектурах, но с помощью нескольких хорошо продуманных правил брандмауэра вы сможете показать, что ваша система устойчива к любому отказу региона. Если ваша компания использует собственное сетевое оборудование, следует периодически проверять службы автоматического переключения и резервирования, чтобы убедиться, что в нужный момент они сработают должным образом.

Мониторинг и оповещение

Сценарий: внедряйте сбои в вашу систему и выключайте компоненты, пока не найдете действие, которое не вызывает оповещение.

Если у вас есть предупреждающие метрики, которые вы никогда не видели в действии, то вам приходится принимать на веру, что они работают, как задумано. Вы должны регулярно проверять, что ваши системы мониторинга и оповещения действительно выполняют свою работу. По этой же причине у детекторов дыма есть кнопка тестирования.

Телекоммуникации и IT-системы

Сценарий: оборудование для видеоконференций вашей компании становится недоступным в течение значительной части дня.

Внутренние системы видеоконференцсвязи и чата позволяют десяткам тысяч сотрудников Google общаться друг с другом каждый день во время обычной работы. Десятки тысяч настольных компьютеров используются для выполнения повседневной работы. Эти системы очень важны, но они не должны быть критическими в том смысле, что их временное отсутствие не должно доставлять никаких неприятностей, кроме легкого дискомфорта. На случай отказа системы видеоконференций необходимо настроить, согласовать и протестировать альтернативные способы связи.

Режим доступа специальных служб

Сценарий: работник сломал ногу в труднодоступном месте или в зоне с ограниченным доступом для посторонних.

Будут ли сотрудники специальных служб иметь доступ к закрытым зонам на ваших объектах в случае чрезвычайной ситуации? Центры обработки данных Google являются промышленными рабочими местами, и хотя Google гордится тем, что обеспечивает безопасные рабочие места, никто в мире не застрахован полностью от риска несчастных случаев. Важно иметь проверенную и отрепетированную процедуру реагирования на эти ситуации. Заранее определите, куда направлять экстренные службы, если они вызваны

на ваши объекты, и назначьте ответственных, которые должны сопровождать их в случае необходимости.

Полная перезагрузка

Сценарий: критическая уязвимость безопасности требует продолжительной перезагрузки каждой системы в вашей инфраструктуре. Все должно быть выключено, но все ли снова включится?

Отработайте полный перезапуск системы или набора связанных систем – не надейтесь, что эта процедура успешно сработает сама по себе. Вы должны быть знакомы с начальной загрузкой и холодным перезапуском любой службы, которая имеет решающее значение для вашего бизнеса.

5.1.3. Как выполнить тестирование

Надеемся, что на данный момент вы располагаете доступом к одной или двум системам, которые вас интересуют на предмет проведения аварийного тестирования, может быть, даже знаете конкретные аспекты, которые необходимо исследовать. Но вам придется еще немного потрудиться, чтобы превратить эти идеи в работоспособный тест. В Google команды часто обращаются к ветеранам DiRT за помощью, повторяя их первоначальные идеи, а затем совершенствуют их в практических тестах. Мы можем дать вам несколько полезных советов, которые вытекают из рассмотренных ранее правил взаимодействия. Ваш тест должен быть научным; рассматривайте тестирование в аварийных ситуациях как проведение научного эксперимента[1]. Внешние переменные в системе должны контролироваться в максимально возможной степени. Ваше воздействие на систему должно быть конкретным и целенаправленным, а способы измерения реакции должны быть определены заранее. Результаты вашего теста могут подтвердить ожидаемое поведение вашей системы в сценариях сбоев или, наоборот, выявить неожиданное поведение. Убедитесь, что ваш тест добавляет ценность; повторное тестирование хорошо известных или понятных режимов отказа – это напрасная трата времени и усилий.

Даже после того, как вы определились с воздействием вашего теста на систему, вы должны четко определить, *что именно* вы тестируете. Вы хотите, чтобы в этом тесте оценивалась реакция человека на неисправность или соблюдение и выполнение процедур управления инцидентами? Вы тестируете систему в целом, реакцию на сложные режимы сбоев, механизм автоматического восстановления? Уверены ли вы, что потребители вашей услуги все правильно понимают и не имеют необоснованно завышенных ожиданий? Старайтесь проверять только одну гипотезу за раз. Опасайтесь смешивать тестирование реакций автоматизированной системы и тестирование поведения человека.

Проверка одной гипотезы не означает, что вам нужно избегать одновременного внедрения нескольких сбоев. В реальном мире сочетание неза-

---

[1] «Принципы» хаос-инжиниринга не случайно отражают научный подход – Вселенная является самой первой крупномасштабной распределенной системой.

висимых сбоев, которые по отдельности относительно безопасны, может привести к катастрофе. Раскрытие петель отрицательной обратной связи и катастрофических комбинаций ошибок – один из самых полезных результатов тестирования DiRT. Всегда применяйте обычные меры безопасности: имейте возможность остановить ваш тест в любой момент и четко определите заранее, какое поведение системы вы считаете нормальным.

Перед началом тестирования полезно убедиться, что сотрудники техподдержки знают о вашем тесте. Отсутствие взаимодействия с командой техподдержки ограничивает вашу осведомленность о неожиданных проблемах, которые могут продолжаться и после завершения теста, и мешает отличать срабатывание автоматической системы восстановления от действий сотрудника техподдержки. Это хороший пример применения правил взаимодействия; заблаговременное уведомление персонала техподдержки о тесте не повлияет на его работу, а также поможет вам собрать информацию от команд, которые испытали непредвиденные трудности в результате запуска теста (обычно первый, кому сообщают о неполадках в системе, – это сотрудник техподдержки).

Если вы не можете сформулировать явную причину, по которой необходимо ограничить общение, то не бойтесь излишних коммуникаций. Если вы собираетесь проверить реакцию человека на инцидент, то небольшая неожиданность вполне уместна, но даже в этих случаях не помешает предварительно озвучить период времени, когда может состояться тест. Это полезно для исключения конфликтов планирования и послужит страховкой на тот случай, если реальный инцидент возникнет именно тогда, когда вы намеревались провести тестирование. Отправьте напоминание или два соответствующим участникам и потенциально затронутым командам за пару недель и несколько дней до фактического испытания; это призовет людей к большей бдительности и более спокойному отклику на тест.

Решение о том, когда и как долго проводить тест, может оказать огромное влияние на качественные и количественные показатели того, что вы изучаете. Запустите тест в течение неоправданно длительного периода времени, и вы можете обнаружить, что непреднамеренно отключили одного или нескольких клиентов. Запустите тот же тест на слишком короткий период времени, и вы можете не успеть получить какие-либо измеримые результаты; перерывы в работе останутся незамеченными или будут проигнорированы. В идеале продолжительность более крупных тестов совместно используемой инфраструктуры может увеличиться по мере выявления и устранения проблем системы. Стремление начинать с малого в целом выглядит разумно, но в случае тестирования запуск «мягких» тестов зачастую становится пустой тратой времени и усилий. Вы можете запустить нормальный «жесткий» тест, но сделать это в непиковые часы, чтобы собрать данные с целью постепенного увеличения интенсивности. Как правило, в Google мы избегаем тестирования в периоды пиковой нагрузки.

При планировании теста учитывайте корпоративный календарь вашей компании. Следует избегать совмещения аварийных тестов с крупными праздниками и значительными культурными событиями (крупными спортивными мероприятиями, праздничными распродажами в магазинах и т. д.). Остерегайтесь дат ежемесячных или ежеквартальных отчетов в вашей ком-

пании, если только это не то, что вы намерены протестировать. Отчеты о финансах, бухгалтерский учет и бизнес-аналитика, возможно, в последнюю очередь приходят вам на ум, когда вы заранее рассматриваете последствия, но вы явно не хотите быть человеком, который сломал систему начисления заработной платы компании в ночь перед выплатой.

Общий совет в Google – «смотри во все стороны», прежде чем начинать тест DiRT. Непосредственно перед началом теста постарайтесь следить за незапланированными случайными событиями, которые могут повлиять на ваш тест, особенно опасаясь текущих производственных проблем, которые могут усугубить ваш тест. Вы всегда можете отложить тест на несколько часов, дней или даже недель.

5.1.4. Сбор результатов

После того как планирование и оповещение завершено, а ваш тест спокойно работает, вы должны заняться регистрацией того, что обнаружили. Во время выполнения теста организаторам рекомендуется вести заметки о совершаемых ими действиях, а также обо всем, что покажется интересным или необычным. Для каждого запуска теста DiRT в Google составляется упрощенный аналитический отчет, вдохновленный «посмертными» документами, которые мы используем для реальных инцидентов. Любые проблемы, выявленные в ходе теста, специально помечаются как обнаруженные во время тестирования DiRT, и Google время от времени выделяет внутренние премии за закрытие ошибок DiRT, чтобы предотвратить стагнацию проблем с более низким приоритетом. Наконец, отчет содержит раздел для обратной связи по вопросам выполнения теста. Команда добровольцев, ответственных за развитие программы DiRT, использует эту обратную связь, чтобы многократно улучшать работу программы с каждым годом.

Менее известные подходы к аварийному тестированию дают любопытные результаты, которые заслуживают отдельного описания. При подготовке больших инфраструктурных тестов, предлагающих механизм безопасного списка, приносит пользу даже рассмотрение заявок на включение в список. В Google мы требуем, чтобы любая команда, запрашивающая включение в безопасный список, предоставила краткое объяснение, почему они не смогут принять участие в тесте, и определила перечень действий, которые устранят необходимость в следующем раунде тестирования. Бывают случаи, когда оповещение о предстоящем тесте вызывает настолько бурную реакцию у будущей «подопытной» команды, что фактически тест становится ненужным – одной угрозы теста оказывается достаточно, чтобы выполнить значительный объем срочной работы[1].

---

[1] В Google существует явление, известное как «проклятие DiRT», когда в случаях отмены или переноса особо важного теста вскоре случается реальный инцидент, необычайно похожий на первоначально запланированный тест. Были даже ситуации, когда реальный инцидент совпадал с первоначально запланированным временем для фальшивого инцидента. Один из неформальных принципов DiRT заключается в том, что «если вы не проведете проверку, Вселенная сделает это за вас». Приятно знать, что у Вселенной есть чувство юмора.

5.2. Объем тестов в Google

Внутренняя инфраструктура Google достаточно велика, и любой человек может знать о ней лишь в разумных пределах[1] – в конце концов, мы все только люди. Аварийное тестирование – это масштабируемое средство для глубокого изучения взаимодействия между системами. Со временем тесты могут наращивать интенсивность, продолжительность и объем. Большие тесты базовой инфраструктуры имеют дурную славу, но они опираются на фундамент из сотен и тысяч изолированных экспериментов с низким уровнем риска. Если вы копнете достаточно глубоко, то обнаружите, что к большинству тестов, упомянутых в предыдущем разделе, можно подойти с разными мерками.

Основные поставщики услуг для критически важной инфраструктуры, такой как вычисления, сеть, хранилище и управление доступом, имеют скрытый стимул выстраивать для своих клиентов надежные конфигурации. Для этих сервисов неправильная конфигурация клиента или недостаточное выделение ресурсов может привести к тому, что поставщик услуг будет завален претензиями, несмотря на то что сервис работает по назначению. Необходимость в снижении качества сервиса до наихудшего уровня должна быть хорошо понятна клиенту. Это понимание является условием спокойной и успешной работы как поставщика услуг, так и потребителей.

Иногда команды в Google искусственно ограничивают крупные сервисы минимальным SLO в пределах региона в течение продолжительных периодов времени. Эти тесты запланированы несколько раз в год, а выбранные регионы периодически меняются, чтобы гарантировать, что никто не останется без внимания. Точная продолжительность эксперимента определяется на основе «бюджета ошибок» SLO, который поставщик услуг готов потратить для его финансирования[2]. Если ваша служба постоянно превышает SLO, то вы накапливаете «бюджет ошибок», и аварийное тестирование – это возможность потратить его. Когда мы запускаем подобные тесты в Google, то стараемся продлевать их так долго, как только можем. Некоторые команды могут предвидеть, что их затронут, и примут стратегию пережидания шторма и окончания сбоя, а не эскалации проблемы. Затягивание теста на несколько дней подталкивает к энергичным действиям и долгосрочным решениям тех, кто был готов переждать более короткие перебои. Продление теста помогает выявить внутренних пользователей, которые случайно занизили свои требования к уровню обслуживания и надежности, необходимые для стабильной работы их приложений. Простого анализа работы системы недостаточно для защиты от этих типов проблем, поскольку некритические зависимости могут постепенно сползать в сторону критичности. Практическое тестирование

---

[1] Это мнение кратко выражено в теореме Вудса: «По мере того как возрастает сложность системы, точность модели этой системы у любого отдельного агента быстро снижается».

[2] Концепция бюджетов ошибок и их применение в Google подробно обсуждаются в статье: *Mark Roth.* (Un)Reliability Budgets: Finding Balance between Innovation and Reliability. Vol. 4, № 4 (August 2015), а также *Marc Alvidrez.* Embracing Risk // *Beyer et al.* Site Reliability Engineering, Chapter 3.

является одним из немногих способов смягчения рисков от скрытого роста критичности.

Хорошо спроектированный, длительный, большой инфраструктурный тест включает в себя список безопасности и использует его в качестве стимула для улучшения любых приложений, которые запросили включение в список. Идеальный тест также предоставит пользователям возможность тестировать свои услуги изолированно, независимо от крупномасштабного теста, например возможность искусственного внедрения задержки или ошибок в клиентскую библиотеку службы. Это позволяет группам, которые зависят от тестируемой службы, запускать свои собственные изолированные тесты на аварийные ситуации, заблаговременно проверяя надежность приложения в тех же условиях, которые могут возникнуть в глобальном тесте. После обнаружения и устранения проблем в конкретном приложении полезно снова запустить большой инфраструктурный тест; он послужит доказательством, что проблемы были решены, а также поможет предотвратить регрессию, поскольку автоматизирован и предназначен для непрерывной работы.

Вот несколько масштабных тестов, которые мы проводим в Google:

- временное отключение служб хранения журналов в регионе, чтобы пользователи, которым требуется высокая доступность журналов, настроили свои системы для надлежащего переключения при сбое;
- время от времени отключаем нашу глобальную службу управления доступом[1];
- крупномасштабное засорение служб хаотичными данными. Google располагает автоматизированными инструментами, которые могут «засорять» ввод служб и отслеживать последующие сбои. Они работают постоянно, в отличие от разработки и отладки;
- масштабная фрагментация сети изолирует целый кампус;
- запрет доступа к учетным записям автоматизированных аккаунтов (учетные записи роботов и сервисов) на несколько часов.

В Google постоянно проводятся изолированные тесты с низким уровнем риска; команды используют автоматизированные инструменты для регулярного создания мини-катастроф в среде отладки. Хотя экстремальные условия применяются к средам, намного опережающим производство, они служат довольно жестким напоминанием о том, что ждет будущую систему в реальном мире. Изучение стандартного черного ящика промежуточной среды помогает выделить сочетание факторов, при которых провокация сбоя нарушает обслуживание.

Мы пытаемся автоматизировать все, что можем, в Google, и аварийные тесты не являются исключением. Для того чтобы идти в ногу с масштабным ростом Google как организации, программе DiRT пришлось превратиться в общую платформу готовых к использованию тестов на аварийные ситуации. Чтобы удовлетворить эту потребность, мы разработали набор тестов для внедрения ошибок «под ключ», которые легко настраиваются и могут выполняться из одного инструмента командной строки. Этот инструмент

---

[1] *Marc Alvidrez.* The Global Chubby Planned Outage // *Beyer et al.* Site Reliability Engineering, Chapter 4.

предоставляет готовые аварийные ситуации «из коробки» и снижает порог входа для команд, которые хотят использовать автоматическое аварийное тестирование, но не знают, с чего начать. Наша коллекция автоматизированных тестов позволяет легко собрать в одном месте большие инфраструктурные тесты с самообслуживанием и добавить несколько индивидуальных тестов. У нас есть автоматизированные тесты, связанные с балансировкой нагрузки, региональными отключениями, задержками репликации хранилища, полной и частичной очисткой кеша, а также задержкой RPC и внедрением ошибок. Стандартизация структуры автоматизированных аварийных тестов позволила нам реализовать унифицированную первоклассную поддержку важных функций, таких как глобальные и частичные прерывания всех автоматических тестов с помощью общего механизма «большой красной кнопки», хорошо продуманных откатов, аудита и предварительного автоматического оповещения.

Borg и показатель SLO: пример из жизни

Borg[1] – это система управления кластерами Google для планирования и управления приложениями в центрах обработки данных. Пользователи настраивают приоритет для своих заданий, которые должны быть включены в план Borg, при этом рабочие нагрузки в реальном времени имеют приоритет над пакетной обработкой[2]. Если задача досрочно завершается мастер-процессом Borg, происходит ее «исключение». Это может произойти по ряду причин, включая необходимость обновления ядер операционной системы, перемещения физического оборудования, замены дисков или просто потребность в ресурсах для задач с более высоким приоритетом. Borg SRE публикует уровень SLO, при котором происходит исключение (в зависимости от приоритета задачи), чтобы очертить верхнюю границу исключения, которую пользователи должны учитывать при разработке систем.

При нормальной работе задача никогда не приблизится к объявленному уровню SLO, после которого произойдет исключение, но во время некоторых плановых работ по техническому обслуживанию вероятность исключения увеличивается. Это обслуживание приблизительно равномерно распределено по ячейкам Borg, поэтому редко случается с точки зрения отдельной ячейки, но обязательно происходит в достаточно долгой перспективе. Разумеется, пользователи Borg удивлялись, когда значение SLO резко снижалось вплоть до исключения задачи.

Google помогает пользователям Borg SRE повысить надежность своих систем за счет повышения приспособленности к уровню исключения SLO. Несколько инициативных инженеров из команды Borg SRE разработали план ротации теста через все ячейки Borg и принудительного исключения производственных задач в соответствии с приоритетами. Тест выполняет-

---

1 *Niall Murphy, John Looney, Michael Kacirek, Beyer et al.* Site Reliability Engineering, Chapter 7; *A. Verma et al.* Large-Scale Cluster Management at Google with Borg // Proceedings of the European Conference on Computer Systems, 2015.

2 *Dan Dennison.* Data-Processing Pipelines // *Beyer et al.* Site Reliability Engineering, Chapter 25.

ся в течение нескольких дней, во время которых искусственно удаленные задачи помечаются предупреждающим сообщением о проведении теста и ссылкой на более подробную информацию на внутреннем веб-сайте. Пользователи могут включить себя в список безопасности во время выполнения теста. В дополнение к режиму тестирования всей ячейки Borg SRE предлагает режим самообслуживания, при котором пользователи могут запускать тестирование для отдельных групп задач. Пользователи Borg заранее проверяют свои задачи с помощью теста самообслуживания, чтобы подготовиться к запланированным внешним тестам, и точно знают, что их процессы не будут нарушены.

5.3. Вывод

Как лучше подготовиться к неожиданности? Этот, казалось бы, парадоксальный вопрос породил программу Google Disaster Recovery Testing. Постоянно растущие системы со временем неизбежно становятся хаотичными, и, вместо того чтобы отвергать очевидное, сторонники хаос-инжиниринга пользуются этим. Регулярное аварийное тестирование представляет собой средства проверки предположений и эмпирического подтверждения поведения, что дает разработчикам более глубокое понимание системы и в конечном итоге приводит к более стабильным системам. Мы часто не замечаем, что встраиваем в системы неявные предположения, пока эти предположения не будут разбиты в пух и прах непредвиденными обстоятельствами. Регулярные, формализованные и тщательно разработанные аварийные тесты позволят вам преднамеренно проверять систему на наличие таких вредных допущений. Несмотря на то что тестирование в аварийных ситуациях не полностью исключает риск, профиль риска можно настроить так, чтобы он соответствовал вашим ожиданиям, а наихудшие последствия были гораздо менее разрушительными по сравнению с риском сложного сбоя системы в условиях реальной жизни.

Представьте, что вы каким-то волшебным образом получили возможность точно запланировать следующий производственный инцидент вашей системы. Вы помещаете инцидент в календарь каждого сотрудника и старательно сообщаете о нем, чтобы никто не остался в неведении. Когда наступит назначенное время, у вас будут наготове лучшие разработчики, готовые записать и проанализировать проблему, как только она возникнет. Конечно, эта группа разработчиков будет заблаговременно расширять свои знания об источниках инцидентов и потенциально затронутых системах и хорошо подготовится к интерпретации поступающих данных. Насытившись данными, вы пускаете в ход еще одну новую суперспособность и отменяете инцидент. Система возвращается в нормальное состояние в течение нескольких минут. Анализируется множество собранных данных, решаются обнаруженные проблемы, и ваши системы становятся гораздо надежнее. Эти суперспособности быстро и резко повысят устойчивость вашей системы. Можно ли представить, что кто-то откажется от таких суперспособностей?

Об авторе

Джейсон Кахун – разработчик программного обеспечения и инженер по надежности подсистем в Google. Помимо написания программ и анализа технологических систем, он увлекается резьбой по дереву, проводит время со своими собаками и плохо играет в шахматы.

Глава 6

Вариативность и приоритеты экспериментов в Microsoft

Автор главы: **Олег Сурмачев**

В Microsoft мы разрабатываем и используем нашу собственную программу хаос-инжиниринга для масштабной облачной инфраструктуры. Мы полагаем, что выбор эксперимента, в частности, оказывает огромное влияние на то, как вы применяете хаос-инжиниринг в своей системе. Примеры различных сценариев отказов в реальном производстве иллюстрируют, как различные реальные события могут повлиять на вашу систему. Я предложу метод определения приоритетов экспериментов с вашими службами, а затем расскажу про правила выбора различных экспериментов. Моя цель в этой главе – предложить стратегии, которые вы можете применять в процессе разработки для повышения надежности ваших продуктов.

6.1. Почему все так сложно?

Современные программные системы сложны. Сотни, а часто и тысячи инженеров работают над созданием даже самого маленького программного продукта. Тысячи, а может и миллионы единиц оборудования и программного обеспечения соединяются в единую систему, которая становится вашим сервисом. Вспомните об инженерах, которые работают на поставщиков оборудования, таких как Intel, Samsung, Western Digital и другие компании, разрабатывающие и создающие серверное оборудование. Подумайте о Cisco, Arista, Dell, APC и других провайдерах сетевого и энергетического оборудования. В свою очередь, Microsoft и Amazon предоставляют вам облачную платформу.

Все эти зависимости, которые вы включаете в свою систему в явном или неявном виде, имеют свои собственные зависимости, вплоть до электросетей и оптоволоконных кабелей. Ваши зависимости объединяются, чтобы создать сложный черный ящик со множеством движущихся частей, поверх которых вы создаете свою собственную систему.

6.1.1. Пример неожиданных осложнений

В начале своей карьеры я работал над алгоритмами для гидроакустической аппаратуры подводных лодок. В те дни я мог буквально по памяти сказать, кто отвечает за каждый компонент, от которого зависит мой код, как за программное, так и за аппаратное обеспечение: каждую библиотеку, каждый драйвер, каждую машину, на которой мы работали. Подводная лодка – это очень замкнутая среда, поэтому для выполнения задачи можно задействовать лишь небольшое количество компьютеров, проводов и жестких дисков. У вас есть ограниченное количество датчиков и обычно только один пользователь в системе. Тем не менее я видел множество томов документации, состоящих из тысяч страниц каждый.

Каждое рабочее совещание собирало, по меньшей мере, дюжину человек: каждый компонент подвергался тщательному анализу, каждый случай использования документировался, а каждая ситуация внимательно изучалась. Модуль, над которым работала моя команда, выполнял предварительную обработку сигналов, и мы провели самое тщательное лабораторное тестирование, которое только могли себе представить. Мы воспроизводили записанные сигналы от других подводных лодок, применяли разные модели погоды и местоположения. Мы очень тщательно выполнили все испытания.

Когда я заказал билет на самолет для проведения первого испытания на борту настоящего судна, я был уверен, что поездка займет не более пары дней. Какие могут быть проблемы? Установить, проверить, отпраздновать успех, снять. К сожалению, все пошло не так, как планировалось. Это была моя первая поездка на Север России. Моя жена, которая выросла в тех краях, посоветовала мне взять больше теплой одежды, чем нужно по сезону. Конечно, я сказал ей: «Не беспокойся, это всего лишь пара дней. В конце концов, сейчас август, еще лето». Когда самолет приземлился, шел снег. Полдня пропало зря из-за ненастной погоды, а небольшая простуда была моим другом до конца поездки.

Когда я наконец смог установить наш модуль и опробовать его, он просто не работал. Первая попытка довести работу системы до конца заняла 40 дней. Мы столкнулись с десятками проблем, которых не предвидели, обнаружили сочетания факторов, которые мы не рассматривали, и встретились лицом к лицу с новыми проблемами, которые никогда не появлялись на чертежной доске. Соединительные провода от сонаров к процессору сигналов включали более 200 разъемов, ни один из которых не был сделан правильно. Нам пришлось пройтись вдоль всей проводки и изучить каждый разъем, чтобы программно компенсировать механические недостатки.

Мы провели тщательную подготовку и вложили огромные усилия в планирование, но во время первого испытания все равно столкнулись с хаосом. Что мы сделали не так? Мы не учли весь диапазон событий, которые могут произойти с нашей системой в реальном мире. Мы сосредоточились на математических и компьютерных проблемах обработки сигналов, но не учли плохую проводку, экстремальные температуры или качество пользовательского ввода, когда корабль болтается на волнах в плохую погоду. Мы узнали все это постфактум и ценой задержки проекта и времени, проведенного вдали от наших семей.

Один из ключевых принципов хаос-инжиниринга – это изменение реальных событий в экспериментах. Если бы мы посвятили время изучению реальных факторов как внутри, так и вне системы, вместо того чтобы тщательно планировать свой успех, возможно, мы провели бы более успешные испытания и раньше оказались бы дома с нашими близкими.

6.1.2. Простая система – лишь вершина айсберга

Давайте на минутку представим небольшой сайт, который я создал для художественной школы моей жены. Сайт размещен на Azure. Смотрите, от чего он зависит: я использовал IIS, Windows, виртуальные машины IaaS, Hyper-V, другую ОС на хосте, инфраструктуру Azure, хранилище данных Azure для моих данных, оборудование для машины разработки и развертывания, оборудование питания и сети в центрах обработки данных, и обслуживающие команды в центрах обработки данных. В общей сложности эти системы включают в себя несколько сотен программных компонентов и около двадцати тысяч разработчиков. Чтобы клиенты могли видеть веб-сайт, я также нуждаюсь в услугах CDN, DNS, глобальной магистральной сети, интернет-провайдера, браузера, который нравится посетителю сайта, и т. д. Даже очень простой сайт совсем не так прост, если принять во внимание все спрятанные за ним системы.

Все эти дополнительные уровни программного и аппаратного обеспечения, абстракций и поставщиков услуг являются частью общей картины, которая влияет на поведение интересующей нас системы. Даже в подводной лодке, в изолированной среде, где я мог точно сказать, от каких компонентов я зависел, и я мог буквально поднять трубку и позвонить каждому задействованному человеку, я все еще не контролировал ситуацию. Невозможно было предвидеть множество явлений, с которыми я столкнулся при испытаниях. Различные реальные события – это именно то, чем мы занимаемся в хаос-инжиниринге.

Проектирование и создание наших продуктов с учетом всех интерфейсов, контрактов на компоненты и соглашений об уровне обслуживания уже является сложной задачей. Сложность возрастает, когда мы охватываем всю нашу цепочку зависимостей. С нашей системой и зависимостями черного ящика будут происходить события, плохо влияющие на их работу: отказы оборудования, ошибки в программном обеспечении, плохо документированные или игнорируемые процедуры и стихийные бедствия. На эти события наклады-

ваются преднамеренные действия: обновления программного обеспечения, исправления ОС, регулярное обслуживание оборудования и сооружений. Есть много других событий, которые не упоминаются, не рассматриваются и даже не приходят в голову. Любое из них может (или не может) критически повлиять на производительность, доступность и успех нашего продукта. Используя методы из этой книги, вы можете устраивать эксперименты с неизведанным, чтобы понять, как факторы под вашим контролем и независимые факторы могут взаимодействовать в реальных сценариях.

6.2. Категории результатов эксперимента

При внедрении методов хаос-инжиниринга в производство вы столкнетесь с различными ситуациями, которые можно условно разбить на следующие категории:

Известные события / ожидаемые последствия
: Это все эксперименты, работающие именно так, как и ожидалось. Вы должны быть в состоянии контролировать ситуацию в любой момент. Именно здесь вы добиваетесь наибольшего успеха в своей программе хаос-инжиниринга, формируете правильный охват и осваиваете инструменты, позволяющие понять свойства вашей системы, будь то метрики, ведение журнала или инструменты наблюдения.

Известные события / неожиданные последствия
: Эксперимент пошел не по плану. Вы по-прежнему управляете начальным событием, но получаете неожиданный результат. Причина может быть простой – например, ваша стратегия мониторинга/отслеживания просто не отражает событие должным образом, или может быть сложной, иногда с необходимостью запуска процедуры аварийного восстановления. В любом случае вы должны прервать эксперимент в соответствии с планом аварийного завершения, записать событие и принять меры. Если есть инструменты для автоматического прерывания эксперимента, даже лучше. Из результатов этого эксперимента вы можете многое узнать о том, как происходит отказ в вашей системе и как восстанавливаться после таких отказов.

Неизвестные события / неожиданные последствия
: Эксперимент идет не по плану, ошибки начинают усугубляться, и вы рискуете потерять контроль над ситуацией. Вполне вероятно, что это тот случай, когда вам нужно привлечь помощников и использовать доступные планы восстановления и отката. Именно здесь вы узнаете самые ценные вещи о вашей системе в долгосрочной перспективе, потому что ваша программа хаос-инжиниринга показала, где находятся ваши слабые места и где вам нужно тратить больше времени и планировать защиту своей системы в будущем.

Первая ситуация является очевидным следствием планирования, но последние две заслуживают дополнительного обсуждения.

6.2.1. Известные события / непредвиденные последствия

Во время презентации хаос-инжиниринга руководству или пояснений заинтересованным сторонам часто упоминают только серьезные и масштабные проблемы. Действительно, существуют такие проблемы, которые впоследствии проявят себя как масштабная катастрофа. Однако есть много других сценариев отказа. Для примера возьмем события, которые регулярно и часто происходят в наших системах: развертывания новой версии, исправления ОС, ротация учетных данных, переход на летнее время и т. д. Эти действия могут вызывать запланированные или неожиданные отказы, такие как аварийное переключение или простой службы, даже когда все работает правильно. Однако такие события также создают два дополнительных риска:

1) как это часто бывает в жизни, на практике все может пойти не так. Даже тщательно запланированное событие может повредить систему, например если во время ротации учетных данных используется неправильный сертификат или новая версия драйвера сетевой карты вызывает проблемы с совместимостью. Любая такая проблема может вызвать инцидент, привести к более длительным простоям и снижению устойчивости к отказу или другим проблемам;
2) некоторые изменения не контролируются вашей системой или любой другой системой. Это может быть, например, переход на летнее время или истечение срока действия учетных данных. Переход на летнее время происходит независимо от вашего уровня готовности. Срок действия сертификатов истекает к определенной дате. GPS может считать недели только до 1024. Обязательные обновления безопасности могут устанавливаться внешними организациями. Эти независимые внешние события могут значительно ограничить реакцию вашей системы. Например, если вы уже работаете над проблемой, которая ограничивает пропускную способность вашей системы и, следовательно, возможность восстановления после отказа, вы можете приостановить процесс непрерывной интеграции и развертывания программного обеспечения. Вы не можете, однако, задержать истечение срока действия сертификата.

Давайте рассмотрим пример обычного развертывания службы. Вот некоторые из причин, по которым рутинное обновление может пойти не так:

- новая версия службы содержит ошибку и должна вернуться на шаг назад или перескочить на шаг вперед;
- служба содержит ошибку, которая нарушила исходное состояние, поэтому вы не можете откатить изменения и перейти к последнему известному исправному состоянию;
- процесс развертывания нарушен, и новая версия службы не установлена;
- процесс развертывания нарушен и слишком быстро применяет новую версию службы, что приводит к слишком высокому количеству аварийных переключений;
- любая комбинация вышеперечисленного.

Пример из моего недавнего опыта включает применение исправлений безопасности. При обнаружении любых угроз безопасности счет всегда идет на часы. Весь процесс проходит быстро, команда срезает углы на бегу, иногда пропуская тестирование. Вынужденно быстрое исправление безопасности может значительно увеличить вероятность отказа.

Во время события Meltdown/Spectre мы ожидали бесчисленных сбоев при развертывании исправления: неполадок при запуске, снижения производительности, нарушения функциональности и т. д. Нам удалось избежать многих из этих проблем, поскольку они уже ожидались. Однако если бы мы не смогли заранее поэкспериментировать с такими событиями, исправление безопасности не могло бы закончиться в разумные сроки. Специально для подобных случаев мы регулярно практикуем развертывание срочных изменений. Со временем мы достаточно хорошо усвоили навыки и адаптировались, и теперь клиенты нашей платформы не замечают существенные сдвиги в графиках обслуживания и доставки.

Как показывают эти примеры, регулярная практика делает нас лучше как специалистов. Даже крупномасштабные проблемы могут быть безвредны, если мы готовимся их встретить.

6.2.2. Неизвестные события / неожиданные последствия

Несколько лет назад я работал в команде инфраструктуры Bing. Однажды мое приложение Outlook перестало работать. Сначала я не слишком волновался, так как у нашего ИТ-отдела было свое собственное представление о том, когда следует устанавливать обновления, но Outlook не заработал после перезапуска. Потом я заметил, что перестал работать Skype и несколько других приложений. Явно что-то отключилось. Именно тогда я услышал, как кто-то пробежал по коридору.

Люди, бегающие в офисе, – обычно плохой знак. Хуже только визит высокопоставленного менеджера. Этот менеджер остановился у моей двери и сказал: «Возьми свой ноутбук и иди со мной». По дороге мы собрали других людей и уселись в конференц-зале, где наконец-то получили информацию о том, что происходит. Похоже, что мы не можем получить доступ ни к чему, даже к нашей собственной телеметрии. Рухнули все службы.

События такого типа создают стрессовую и требовательную среду. Рабочая группа мобилизовалась и провела мозговой штурм потенциальных решений: нестандартные каналы связи, точки ручного вмешательства, способы доступа к плоскости управления платформы. Двадцать очень напряженных минут, пара дюжин телефонных звонков и горстка замечательных идей чуть позже – и мы смогли правильно оценить происходящее и собрать подходящую команду для устранения отказа. Проблема была кульминацией множества мелких неполадок, которые в конечном итоге привели к нарушению конфигурации DNS. Штаб-квартира была изолирована от мира. Нам очень повезло, что трафик клиентов не пострадал.

Почему мы попали в эту ситуацию? Потому что мы этого не ожидали. Мы расслабились. Мы были настолько сосредоточены на симуляции событий внутри центра обработки данных, что полностью закрыли глаза на возможные проблемы в штаб-квартире. Сложность восприятия крупномасштабной инфраструктуры заключается в понимании взаимодействий и зависимостей между компонентами. Изменения в одном компоненте приводят к сложным кумулятивным эффектам у восходящих и нисходящих связей. Изоляция сети оставила нас слепыми и беспомощными. Понимание точного эффекта и масштабов проблемы было ключом к выявлению потенциальных провоцирующих факторов и в конечном итоге к разрешению ситуации.

Могли ли мы что-то сделать, чтобы предотвратить эту ситуацию, прежде чем люди начали бегать по коридорам? Конечно, легко ответить утвердительно задним числом, но это не помогает предотвратить будущие инциденты, для которых у нас никогда не будет совершенного знания наперед. Мы упустили определенные сценарии отказов за пределами нашей области разработки, связанные с отказом подключения к штаб-квартире. Урок здесь заключается в том, что мы должны быть готовы к неожиданным последствиям и как можно лучше отрабатывать восстановление.

6.3. Расстановка приоритетов отказов

Невозможно заранее определить все возможные инциденты, а тем более охватить их экспериментами. В любой программной системе разумного размера слишком много переменных. Чтобы ваши усилия по исключению отказов системы давали максимальный эффект, вы должны расставить приоритеты для разных классов инцидентов. Определите сценарии, которые наиболее важны для вашего продукта, и опишите их.

Давайте рассмотрим подходы к расстановке приоритетов, основанные на трех различных свойствах:

Как часто что-то происходит?

Что будет происходить регулярно? Посмотрите на события, которые гарантированно или с большей вероятностью произойдут первыми, например развертывание новой версии, ротация учетных данных, переход на летнее время, изменения схемы трафика и угрозы безопасности. Подумайте о разнообразии режимов отказа, которые соответствуют каждому такому событию. Вы хотите обнаружить отказ до того, как он случится, поэтому расставьте приоритеты для событий, которые происходят чаще других.

Давайте в качестве примера возьмем ротацию учетных данных. Вы все равно должны делать это время от времени, поэтому делайте это часто. Фактически делайте это так часто, как вы можете себе это позволить с точки зрения затрат, а не только из фактической потребности в безопасности. Ведь мы хорошо разбираемся в том, что делаем часто. Обратное тоже верно. Если вы не использовали ротацию учетных данных в течение полугода или более, как вы можете быть уверены, что она сработает, когда вам это нужно?

Какова вероятность, что вы успешно справитесь с событием?

С какими типами отказов вы можете справиться? Постарайтесь определить риски, которых вы не можете избежать. Существуют некоторые разновидности отказов, которые отключают ваш сервис в глобальном масштабе, и вы ничего не можете с этим поделать. Это будет зависеть от архитектуры вашего сервиса и его устойчивости. Для веб-сайта соседнего ресторана это может быть конфликт с местной властью. Для облака Microsoft это может быть глобальное природное бедствие. В любом случае будут случаться события, которые вам не нужно исследовать заранее, поскольку вы на них никак не влияете. С другой стороны, возможны события, которых вы наверняка не допустите. Расставьте приоритеты экспериментов, исходя из пределов своего влияния, чтобы регулярно проверять предположения, которые хоть в какой-то мере зависят от вас.

Какова вероятность, что это произойдет?

Знаете ли вы про неминуемые угрозы? Однократные запланированные события, такие как выборы или Суперкубок, а также известные уязвимости безопасности – они отличаются от обычных событий тем, что происходят недостаточно часто, чтобы получить приоритет для регулярной проверки. Однако, когда вы знаете, что событие неизбежно, вы должны отдавать приоритет тестированию именно таких событий, а не чему-либо другому. Примером может служить взлом шифра системы безопасности. В прошлом было несколько событий, когда известные шифры были взломаны злоумышленниками. Оглядываясь назад на подобные инциденты, вы можете спросить: насколько ваша система зависит от связи, защищенной определенными шифрами? Что потребуется для обновления шифров, используемых всеми компонентами вашей системы? Как бы вы сохранили функциональность при замене шифра? Как бы вы общались с клиентами? Сколько времени это займет? Или вы должны принять такой риск, как полная потеря шифрования?

Ваши ответы на эти вопросы должны основываться на бизнес-потребностях вашего продукта или услуги. Борьба за надежность системы должна соответствовать ожиданиям ваших клиентов. Всегда полезно четко оговорить с заинтересованными сторонами границы ожиданий.

6.3.1. Исследуйте зависимости

После того как составлен список событий, которые могут повлиять на вашу систему, сделайте то же самое для ваших зависимостей. Как и в случае с вашим продуктом, вы должны учитывать ожидаемые события, пределы допустимого и разовые неминуемые угрозы для всей цепочки зависимостей вашего сервиса. Очень хорошо, если вы можете связаться с владельцами ваших зависимостей и согласиться использовать аналогичные модели моделирования и проектирования угроз. Это не всегда возможно, но участие владельцев зависимостей идет на пользу делу.

Учтите, что все эти отказы могут затрагивать сразу несколько ваших зависимостей. Более того, некоторые зависимости также зависят друг от друга,

что может привести к пересекающимся и каскадным отказам в системе. Подобные отказы сложнее диагностировать с помощью мониторинга конечного уровня, вдобавок их часто упускают из виду на этапе проектирования. Выявление возможности таких отказов и составление схемы фактического воздействия на системы в производстве – одна из возможностей хаос-инжиниринга. Ранжирование событий, которые имеют потенциальное сложное влияние или неизвестное воздействие, – это еще один способ расставить приоритеты событий в хаос-инжиниринге.

6.4. Глубина варьирования

Хаос-инжиниринг служит инструментом для обнаружения *реального поведения* систем и построения графов *реальных зависимостей*. Даже если вы понимаете, как функционирует ваша система, и знаете ценность ваших зависимостей, вы можете упустить из виду полную картину сквозного взаимодействия с клиентами. В связи с нынешним ростом популярности микросервисных архитектур, часто формирующих облака зависимостей, слишком сложных[1] для понимания человеком, может оказаться тщетной попытка понять все возможные варианты зависимостей и планов восстановления после отказа. Решения, принимаемые разработчиками компонентов изолированно, могут привести к непредвиденным последствиям в масштабе системы. Стратегии классического моделирования предусматривают дискуссии об устройстве системы и заседания архитектурного совета; однако это никак не масштабируется, не говоря уже о том, что многие компоненты могут находиться вне контроля вашей организации в целом.

Это делает использование комбинированных и составных отказов важным инструментом при разработке версии хаос-инжиниринга для вашего продукта. Вводя различные режимы отказов, вы можете обнаружить неизвестные связи, зависимости и ошибочные решения. Организацию подобных отказов можно рассматривать как исследовательскую работу, когда вы изучаете чужие, недокументированные и недружественные системы, от которых зависите.

6.4.1. Вариативность отказов

Когда вы перечисляете и расставляете приоритеты событий, не забывайте учитывать степень *вариативности*, которую вы вводите в каждом эксперименте. Ограничение изменений одним компонентом или одним параметром конфигурации вносит ясность в понимание результатов. С другой стороны, внесение более широких изменений может пригодиться при изучении более сложных поведений системы.

Например, если ваша цель состоит в том, чтобы симулировать отказ сети для вашего компонента, вы можете представить такой отказ на уровне про-

[1] См. главу 2, где рассказано про последствия сложности.

токола приложения. Скажем, вы можете разорвать соединение на другой стороне или в рамках используемой вами библиотеки. Это дает вам изолированное понимание причин и следствий для вашей службы. Тем не менее если сбой сети случается в реальной жизни, он также может произойти из-за другого уровня модели ISO-OSI, поломки оборудования, ошибки в драйвере или изменения маршрутизации.

Сравните два разных пути ввода отказа: на транспортном уровне через отключение стека TCP на машине или обычное извлечение провода из сетевой карты. Последнее было бы более реалистичным с точки зрения имитации причинно-следственной связи. Однако это приведет к отключению *всех* компонентов на одном компьютере одновременно, что приведет к объединенному отказу нескольких компонентов. Например, в этом случае ваш компонент мониторинга может потерять соединение и не будет регистрировать телеметрию, которую вы пытаетесь отправить с этого компьютера, что затрудняет диагностику воздействия. Или ваш компонент балансировки нагрузки в этих условиях поведет себя непредсказуемо и т. д. Подобные *комбинированные отказы* создают большее негативное влияние в системе, но порой могут дать лучшее понимание фактического взаимодействия компонентов внутри системы.

В дополнение к комбинированным отказам мы должны рассмотреть *составные отказы*. Это неполадки, случившиеся выше или ниже по потоку от предмета вашего эксперимента. Составные отказы могут быть чрезвычайно опасными, так как они быстро увеличивают радиус поражения в вашем эксперименте. Последствия могут выйти за рамки эксперимента, нарушив первоначальные соображения безопасности и допущения при планировании. Не полагайтесь на то, что эксперимент сработает так, как задумано во время тренировки или игрового дня. Проанализируйте свой план исследования, предполагая, что любая его часть может потерпеть неудачу. Суть хаос-инжиниринга заключается в управляемых экспериментах, которые могут быть отменены с откатом к исходному состоянию. Если возможно, спланируйте и протестируйте алгоритм восстановления перед выполнением эксперимента.

Наконец, вы должны учитывать отказы за пределами вашей области влияния, с которыми вы можете только смириться. Вы не вправе ожидать, что система справится с таким отказом. Лучшее, что вы можете сделать в этом случае, – своевременно оповестить нужную аудиторию. Но вы вправе рассчитывать на быстрое восстановление. Возвращение в рабочий режим после отключения питания, возобновление соединений после ремонта оптоволокна, повторное создание копий хранилища – все эти события должны происходить правильно. Должны работать необходимые автоматизированные процедуры, персонал должен быть обучен ручному вмешательству по мере необходимости и т. д. Эти действия зачастую плохо поддаются автоматизации, поэтому их откладывают на потом. Тем не менее эксперименты в этой области важны, потому что урон из-за невозможности своевременного восстановления может превышать последствия самого инцидента. Эксперименты по глобальным катастрофическим отказам часто дороги и инвазивны, но возможность эффективного восстановления снижает ваш совокупный риск

и предоставляет большую свободу для экспериментов с комбинированными и составными отказами.

6.4.2. Объединение вариативности и расстановки приоритетов

При проектировании вашей методики хаос-инжиниринга и выборе между изолированными, комбинированными и составными отказами вы можете применять тот же подход приоритизации, что и при выборе сценариев для экспериментов с собственным сервисом и зависимостями:

Что произойдет, если вы выберете составные отказы? Производительность вашего компонента, безусловно, может выйти за допустимые пределы SLA. На что еще это может повлиять? Если вы зависите от конкретных аппаратных компонентов, готова ли ваша команда работать по ночам? В выходные или праздничные дни? Если вы зависите от вышестоящего сервиса, допускает ли он полные глобальные отказы? Если вы используете внешний CDN или формирователь трафика, какие дополнительные режимы отказов это влечет? Вписываются ли они в ваши режимы отказа? Придется ли вам реагировать на какие-то новые отказы? Только вы несете ответственность за свой продукт. Клиентов не волнуют особенности работы платформ Microsoft, Amazon или Akamai; они просто хотят получить свои услуги.

Как вы думаете, с какими проблемами вы можете справиться? Какие зависимости вы можете позволить себе полностью потерять? Что вы можете себе позволить удерживать в деградированном состоянии и как долго?

Что такое для вас неминуемая угроза? Предположим, что каждая из ваших зависимостей произвольно выйдет из строя хотя бы один раз, но вы можете изучить и ожидаемые события, такие как запланированные изменения зависимостей. Вот несколько примеров: смена поставщика облака или обновление до более новой версии операционной системы. Если вы решили перейти к другой зависимости, сначала вам следует проделать это в виде хаос-эксперимента.

6.4.3. Расширение вариативности до зависимостей

Научившись изолированно работать с локальными зависимостями, вы можете начать рассматривать всю систему в движении: каждый компонент внутри и вне вашего контроля работает по-разному и дает сбой в случайные моменты времени. Так устроен мир. Допустим, вы хорошо знаете свои зависимости. Попробуйте ответить, как все ваши зависимости поведут себя при следующих обстоятельствах внешнего мира:

- при стабильно высоком уровне SLA (нормальная ситуация). Очевидно, что ваши зависимости не испытают затруднений;
- при сниженной производительности, но все еще в пределах SLA (пониженная пропускная способность сети, повышенная частота отбра-

сывания запросов, иногда недоступные услуги). Вы должны быть в состоянии выдержать это в течение длительного периода времени и, возможно, принять навсегда;

- в точности SLA. Эта ситуация похожа на предыдущий пункт, но интересна с другой точки зрения: если такого качества обслуживания недостаточно, вам следует пересмотреть договор или перестроить свою систему. Например, если вы предоставляете услугу базы данных, а ваш вышестоящий провайдер обеспечивает доступность реплики на уровне 80 %, вы не сможете гарантировать доступность трех реплик системы лучше, чем 99,2 %. Вероятность одновременного отказа всех трех реплик – это умножение вероятностей отдельных отказов, поэтому *ваша* доступность достигает $1 - 0{,}2^3$. У вас есть возможность затребовать у провайдера более высокую доступность экземпляра (скажем, при доступности 90 % вы можете гарантировать 99,9 %) или добавить еще одну реплику (чтобы получить 99,84 %). Ваше решение может зависеть от цели и стоимости;
- хуже, чем SLA. Вы должны решить, является ли для вас критическим провалом выполнение вышестоящим провайдером операций хуже SLA. В зависимости от бизнес-ситуации вы можете решить, что это фатальная проблема, или разработать систему, способную выдержать ухудшение внешней услуги в течение некоторого времени. В примере базы данных вы можете использовать некоторое время локальный кеш, прежде чем сдаться.

Комбинируя несколько событий разной значимости, вы сможете изучить работоспособность системы в этих условиях и избежать неожиданностей в будущем.

6.5. Развертывание масштабных экспериментов

Теперь, когда ваш план готов, вы можете приступить к работе. Как отмечено в других главах, вы должны начинать с малого и минимизировать радиус поражения, как указано в «Принципах»[1]. Начните с выполнения экспериментов с наивысшим приоритетом. Подготовьте планы действий на случай чрезвычайных ситуаций. Помните, что вы не делаете ничего такого, чего бы не произошло в реальной жизни. Вы всего лишь ускоряете естественный ход событий, чтобы устранить сюрпризы.

Следует ясно понимать, с каким типом события вы имеете дело. Для этого крайне важно выстроить механизм мониторинга и отслеживания, учитывая два ключевых момента:

- понимание устройства системы, подвергаемой отказу. Благодаря этому вы сможете увидеть, дает ли эксперимент ожидаемый отклик. Ваша

[1] Читайте про «Принципы хаос-инжиниринга» в главе 3.

задача – добиться того, чтобы эксперимент оставался в заданных границах. Вы можете определить их как конкретную область отказа, определенный компонент или конкретного клиента (тестовая учетная запись);

- понимание общего состояния вашего продукта. Отказы могут комбинироваться и усугубляться за пределами области, которую вы наблюдаете, способами, которые вы не можете предсказать, и намного быстрее, чем вы ожидали. Любая аномалия заслуживает пристального изучения. Во время экспериментов нередко можно получать звонки и электронные письма об эскалации. Важно различать ожидаемые и неожиданные результаты.

В Microsoft наиболее безопасной стратегией развертывания новой серии экспериментов принято считать следующее: исходим из того, что, определив границу эксперимента, мы можем одновременно потерять все, что находится внутри этой границы. В то же время определение границы должно основываться на некотором допущении, учитывающем потенциальное каскадирование. Например, экспериментируя с системами хранения данных с несколькими репликами, вы должны рассмотреть сценарий полной недоступности данных, применяемый к учетным записям, включенным в эксперимент. Мы не собираемся попадать в такую ситуацию, но это не выходит за рамки возможного. Однако мы не ожидаем какого-либо воздействия на другие экземпляры хранилища.

Одним из подходов к изучению системных эффектов является преднамеренное подавление связанных вызовов (обращений к внешним компонентам) во время экспериментов, но я возражаю против подобной тактики. Выполнение процесса от начала до конца, без искусственных ограничений, дает возможность понять, насколько правильно он работает. Если связанные вызовы являются частью процесса (аварийного переключения или восстановления), они также должны работать.

Перенос хаос-экспериментов из среды отладки в производство создаст много новых рисков. Имейте в виду, что ваша система хаос-инжиниринга в первую очередь сама является системой и подвержена тем же недостаткам и сбоям, что и исследуемые системы. Вы рискуете нарушить стабильное состояние, спровоцировать нехватку мощности и, как правило, не выполнить свое обещание клиентам не быть слишком агрессивным. Подвергайте свою собственную систему хаос-инжиниринга самым строгим экспериментам из всех. Это не только поддержит качество и стабильность вашего продукта, но также установит высокую планку для последователей и станет достойным объяснением того, почему так важен хаос-инжиниринг.

6.6. Вывод

Нам не дано предвидеть все возможные будущие события. Эти события непредсказуемо влияют на наши продукты. Мы хорошо разбираемся в том, что делаем регулярно, поэтому регулярно повторяйте одни и те же экспери-

менты с небольшим интервалом. Составьте приоритетный список событий, которые будут происходить каждую неделю. Экспериментируйте на тех, кто в начале списка. Если что-то происходит ежемесячно, экспериментируйте с этим еженедельно. Если это происходит еженедельно, экспериментируйте ежедневно. Если обнаружите неполадку, у вас будет шесть дней, чтобы найти решение. Чем чаще вы используете какой-либо сценарий, тем меньше вероятность, что он случится в реальной жизни.

Известные события с ожидаемыми последствиями – лучшая ситуация, на которую мы можем надеяться. Целью различных экспериментов в хаос-инжиниринге является уменьшение сюрпризов в будущем. Может произойти потеря мощности? Ну так давайте попробуем ежедневно моделировать потерю мощности, чтобы сделать ее менее пугающей. Обновления ОС происходят ежемесячно, поэтому давайте попробуем менять версию вперед или назад несколько раз в день, чтобы посмотреть, как мы справимся с этим. Мы сделали все это в своей инфраструктуре облачных сервисов и благодаря этому стали более устойчивыми к отказам.

Рассмотрите все основные проблемы вашего продукта в трехмесячной перспективе; например, всплески трафика, атаки типа «отказ в обслуживании» или пользователи, использующие очень старую версию клиентского приложения. Попробуйте устроить эти проблемы сегодня. Планируйте все события, которые вы добавляете в систему. Планируйте откат к исходному состоянию. Составьте план действий, когда откат не сработал и вам необходимо восстановить исходное состояние.

Вариативность событий в вашей системе будет намного выше, чем вы ожидаете. То, как ваши зависимости будут вести себя перед лицом различных ключевых событий, можно выяснить только с помощью исследований и экспериментов. От правильной расстановки приоритетов событий зависит полезность хаос-инжиниринга для вашего продукта.

Об авторе

Олег Сурмачев много лет работает в команде облачной инфраструктуры Azure, разрабатывая наборы инструментов и поддержку глобальных сервисов в Microsoft. Он с первых дней помогает сообществу сделать хаос-инжиниринг промышленным стандартом. В Microsoft он отвечает за один из внутренних инструментов хаос-инжиниринга, обеспечивающий внедрение отказов и надежную экспериментальную платформу в облачном масштабе.

Глава 7

Как LinkedIn заботится о пользователях

Автор главы: **Логан Розен**

Всякий раз, когда вы проводите хаос-эксперимент на производстве, у вас есть риск отрицательно повлиять на пользователей вашего продукта. Без наших постоянных пользователей у нас не было бы систем для обслуживания, поэтому мы должны ставить их интересы на первое место при тщательном планировании экспериментов. Хотя совсем без воздействия обойтись не получится, очень важно минимизировать радиус поражения эксперимента и иметь простой план восстановления, следуя которому, вы сможете быстро вернуть все в нормальное состояние. Фактически ограничение радиуса поражения является одним из ключевых принципов хаос-инжиниринга (см. главу 3). В этой главе вы познакомитесь с лучшими примерами соблюдения этого принципа, а также узнаете о том, как он был реализован в индустрии программного обеспечения.

Чтобы рассмотреть тему безопасности пользователя в другом контексте, давайте ненадолго переключимся на автомобильную промышленность. Все современные транспортные средства проходят строгие краш-тесты со стороны производителей, независимых экспертов и правительственных организаций на предмет безопасности пассажиров в случае аварии. В ходе испытаний инженеры используют специальные манекены для краш-тестов, имитирующие человеческое тело и имеющие несколько датчиков, показывающих, как столкновение может повлиять на реального человека.

За последние десятилетия автомобильные краш-тесты претерпели значительные изменения. В 2018 году NHTSA (National highway traffic safety administration, Национальное управление безопасностью движения на трассах) выпустило Thor, который был признан самым реалистичным из когда-либо созданных манекенов для краш-тестов. Имея около 140 каналов данных, Thor передает инженерам обширные данные о том, как аварии влияют на реальных людей. Подобные манекены служат критерием безопасности транспортных средств, которые выпускаются на массовый рынок[1].

---

[1] Более подробно о манекене Thor рассказано в статье: *Eric Kulisch.* Meet Thor, the More Humanlike Crash-Test Dummy // Automotive News, Aug. 13, 2018, https://oreil.ly/tK8Dc.

Это выглядит очевидным: зачем подвергать живых людей краш-тестам, когда вместо них можно усадить манекены? Такой же подход применяется в хаос-инжиниринге при тестировании программного обеспечения.

Как и в случае с набором датчиков Thor, которые измеряют воздействие при столкновении, за последние годы разработаны методы измерения отклонений программной системы от нормального состояния. Даже при ограниченном внедрении отказов мы можем обнаружить влияние на показатели и пользовательский опыт. Эксперименты должны быть рассчитаны так, чтобы как можно меньше воздействовать на людей, по крайней мере до тех пор, пока вы не будете достаточно уверены в своей системе, чтобы справляться с этими отказами аналогичным образом в масштабе системы.

Даже если вы примете все необходимые меры предосторожности (и даже больше), чтобы минимизировать вред, нанесенный вашим пользователям во время эксперимента с хаосом, все еще остается вероятность непредвиденного воздействия. Как гласит закон Мерфи, «все, что может пойти не так, пойдет не так». Вам нужна большая красная кнопка, чтобы завершить эксперимент, если ваше приложение вдруг начинает работать неприемлемо для пользователей. Чтобы вернуться в устойчивое состояние, должно быть достаточно одного щелчка мыши.

7.1. Учитесь на примерах катастроф

Давайте оглянемся на известные случаи экспериментов с безопасностью, которые пошли наперекосяк, и сделаем выводы о том, как мы должны планировать и проводить наши собственные хаос-эксперименты. Даже если среда, в которой проводились неудачные эксперименты, отличается от нашей, мы можем понять, где эти эксперименты свернули не туда, чтобы попытаться не допустить аналогичные ошибки.

Чернобыльская катастрофа 1986 года – один из самых печально известных примеров грандиозной промышленной аварии. Работники атомной электростанции проводили эксперимент, чтобы выяснить, может ли ядро реактора оставаться достаточно охлажденным в случае потери мощности. Несмотря на возможность серьезных последствий, специалисты по безопасности отсутствовали во время эксперимента и не координировали действия операторов, чтобы минимизировать риск.

Во время эксперимента действия, предполагающие снижение мощности реакции, привели к обратному эффекту; мощность резко возросла и привела к нескольким взрывам и пожарам, которые вызвали радиоактивные осадки, имевшие катастрофические последствия для окружающей территории[1]. В течение нескольких недель после неудавшегося эксперимента погиб тридцать один человек, двое из которых были работниками на станции, а остальные были ликвидаторами аварии, пострадавшими от радиационного облучения[2].

---

[1] World Nuclear Association. Chernobyl Accident Appendix 1: Sequence of Events (updated June 2019), https://oreil.ly/CquWN.

[2] World Nuclear Association. Chernobyl Accident 1986 (updated February 2020), https://oreil.ly/Bbzcb.

Последующий анализ показал, что система находилась в нестабильном состоянии, без надлежащих мер безопасности, не было предусмотрено «адекватного оборудования и аварийных сигналов для предупреждения операторов об опасности»[1]. Это сочетание факторов привело к катастрофе, о которой сегодня знает весь мир и в которой наблюдается четкая аналогия с планированием и проведением экспериментов с программным обеспечением.

Даже когда ставки ниже и мы всего лишь вводим ошибки в веб-сайт, а не регулируем мощность атомной электростанции, мы все равно должны ставить интересы своих пользователей на первое место в каждом проводимом нами эксперименте. Когда мы проводим эксперименты с хаосом, мы должны извлечь уроки из того, что произошло в Чернобыле, и в первую очередь думать о безопасности наших пользователей.

7.2. Детализованные эксперименты

Скорее всего, особенно если вы только начинаете изучать внедрение хаос-инжиниринга в своей компании, вы ассоциируете эту методику с бравыми инженерами Netflix, отключающими серверы или целые центры обработки данных, с инструментами Chaos Monkey и Chaos Kong в руках. Тем не менее они достигли успеха, только разработав достаточно надежную инфраструктуру, способную справляться с этими типами отказов.

Когда вы впервые проводите эксперименты с отказами, важно начать с малого. Даже отключение одного сервера может оказаться катастрофическим, если ваши системы не рассчитаны на обеспечение высокой надежности. Постарайтесь спроектировать свои первые эксперименты таким образом, чтобы эксперименты имели высокий уровень детализации и нацеливались на наименьшую структурную единицу, какой бы она ни была в вашей системе.

Существует несколько возможных комбинаций подопытных единиц, и все они зависят от программного обеспечения, на котором вы экспериментируете. Давайте представим очень простой сайт интернет-магазина по продаже виджетов. Пользовательский интерфейс магазина отправляет вызовы REST к серверной части, которая, в свою очередь, общается с базой данных (рис. 7.1). Если вы хотите внедрить отказ на уровне приложения, то, вероятно, захотите посмотреть, что произойдет, когда у вашего веб-интерфейса возникнут проблемы с конкретными вызовами к серверной части.

Допустим, сервер магазина предоставляет API для продуктов, корзин и заказов, каждый из которых имеет свои собственные методы (GET для товаров, POST для добавления в корзину и т. д.). Для каждого из этих вызовов, которые веб-интерфейс отправляет серверу, обычно должен существовать контекст клиента, который может быть кем угодно, от человека до компании, покупающей виджеты.

Даже в этом очень простом приложении есть несколько способов вызвать отказ – комбинации сбоев вызовов API для разных пользователей. Однако,

---

[1] *Najmedin Meshkati.* Human Factors in Large-Scale Technological Systems' Accidents: Three Mile Island, Bhopal, Chernobyl. Industrial Crisis Quarterly, Vol. 5, № 2 (June 1991).

начав с малого, вы можете исследовать реакцию вашей системы на отказы, не вызывая массовые проблемы для всех ваших пользователей.

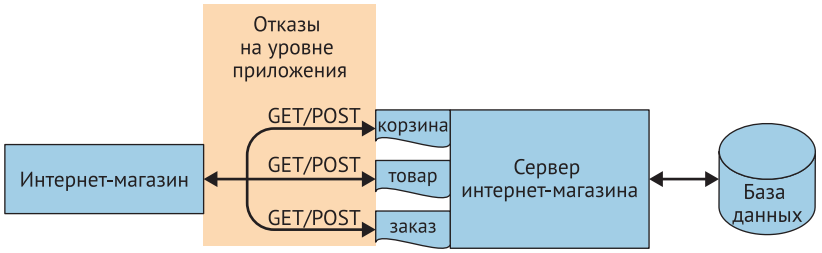

Рис. 7.1 ❖ Схема простого приложения для онлайн-торговли

В идеале вы можете начать экспериментировать с отказами только для вызовов API, создав для этой цели собственную учетную запись пользователя. Таким образом, вы можете проверять сбои в небольшом масштабе, не опасаясь массового воздействия[1]. Допустим, вы хотите увидеть, как будет выглядеть пользовательский интерфейс, если клиентская часть не может выполнить вызов GET к API корзины серверной части. Во-первых, это помогает задуматься о том, каким должен быть оптимальный опыт для клиента, когда это происходит; хотите ли вы показать страницу с сообщением об ошибке, которая просит пользователя повторить попытку? Возможно, вы захотите повторить вызов автоматически или воспользуетесь запасным вариантом и покажете клиенту страницу с распродажей других товаров, которые могут заинтересовать и отвлечь клиента, пока вы исправляете свой API.

Вы можете вызвать отказ для своего пользователя и посмотреть, что произойдет, когда вы попытаетесь добавить виджет в корзину, а время ожидания ответа API истечет. Результат может полностью соответствовать вашим ожиданиям на случай отказа или подскажет способ повысить надежность и качество обслуживания клиентов.

Это, конечно, надуманный пример, основанный на простой системе, которая позволяет прерывать вызовы REST на очень целевом уровне. Это не всегда возможно и зависит от того, сколько вы готовы инвестировать в разработку методики хаоса, или от ограничений системы, на которой вы экспериментируете.

Возможно, наименьшая детализация, которая вам доступна, – это все входящие или исходящие соединения определенного порта на данном хосте. Даже на этом уровне вы все еще избегаете более масштабных последствий своего эксперимента, поскольку худшее, что может случиться, – это когда один хост не отвечает на запросы. Пока вы остаетесь маленьким, у вас небольшой радиус поражения.

---

[1] Тед Стшалковски, SRE из команды Waterbear, представил информативную презентацию о том, как вызвать сбои в небольшом приложении Flask.

7.3. Масштабные, но безопасные эксперименты

Итак, вы проверили устойчивость вашей системы к определенным отказам или, может быть, просто к конкретному отказу, который вас особенно волновал. Что дальше? Естественно, имеет смысл продолжить эксперименты в более широком масштабе и посмотреть, будет ли ваша система по-прежнему нормально реагировать. В чем заключается влияние масштаба? Иногда, в зависимости от устройства системы, она может совершенно по-разному реагировать на одиночный отказ и множество одинаковых отказов.

Может быть, ваш кеш замедляется, когда вы провоцируете тайм-аут для обращений к базе данных, но вы хотите убедиться, что он может обработать *все* сбойные вызовы базы данных. Или, может быть, ваша система может поразному реагировать на однотипный отказ, происходящий в масштабе, по сравнению с несколькими отказами, воздействующими на разные ее части. Некоторые масштабные отказы могут привести к каскадному эффекту в вашей системе, который вы не смогли воспроизвести, просто вызвав отказ для своего экспериментального пользователя.

Ясно, что эксперименты в масштабе важны, но это сопряжено со значительными рисками, которые следует учитывать, прежде чем ваши эксперименты выйдут за пределы песочницы. Вы должны очень хорошо знать стабильное состояние своей системы, чтобы вовремя заметить любое отклонение. Это важное условие для экспериментов любого масштаба, особенно когда отказы могут привести к катастрофическим последствиям для пользователей вашей системы. Ключевые следствия из принципа устойчивого состояния заключаются в том, что вы всегда должны следить за показателями своей системы и иметь возможность быстро прекратить эксперимент, если что-то кажется неожиданным или подозрительным. Заранее назначив критерии досрочного прекращения эксперимента, вы можете быстрее прекратить воздействие и сократить время для восстановления после инцидента.

Если есть способ автоматизировать немедленное прекращение эксперимента при появлении отклонений в показателях, способных повлиять на пользователей, это значительно повысит безопасность системы. Хотя план эксперимента может и не предусмотреть все непредвиденные последствия, добавление логики завершения на основе четких критериев помогает снизить частоту и продолжительность инцидентов, негативно влияющих на клиента.

Независимо от автоматизации, встроенной в ваш набор инструментов, «большая красная кнопка» для немедленного прекращения экспериментов, как упоминалось ранее, является ключевым компонентом любого эксперимента с хаосом. Эта «кнопка» может быть чем угодно, в зависимости от того, как вы проводите эксперименты, от команды `kill -9` до серии вызовов API; до тех пор, пока кнопка способна вернуть ваше окружение в устойчивое состояние, она отвечает всем требованиям.

Если отказ, который вы ввели в вашу систему, вызывает реальные проблемы, вам необходимо успеть ликвидировать его, прежде чем он приведет к потере дохода или доверия пользователей. Некоторое влияние допустимо, но длительное воздействие может оттолкнуть людей от вашего сервиса или приложения.

Конечно, есть некоторые нюансы определения того, что считается достаточным ухудшением опыта пользователя, то есть основанием к прекращению ваших экспериментов. Это зависит от разных факторов, в том числе от размера базы пользователей, подвергшейся воздействию, и от того, насколько она утратила функциональность. Нахождение этого порога, вероятно, потребует сотрудничества между несколькими командами, включая представителей заказчика и разработчиков. Перед началом экспериментов важно провести эти совещания, чтобы вы знали, как сопоставить различные виды сбоев с пользовательским опытом.

7.4. На практике: LinkedOut

Как инженер по надежности систем[1] в LinkedIn, который работал над внедрением хаос-инжиниринга в нашу организацию, я потратил значительное время, пытаясь спроектировать нашу структуру экспериментов и стараясь придерживаться руководящих «принципов» хаос-инжиниринга, особенно сводя к минимуму радиус поражения с помощью детализации экспериментов. Наша инфраструктура внедрения отказов на уровне запросов, LinkedOut[2], в рамках проекта Waterbear[3] привела к значительным изменениям в подходах к тому, как разработчики в LinkedIn пишут и экспериментально проверяют свое программное обеспечение, но они охотно идут на это, потому что знают, что могут легко избежать негативного влияния на пользователей.

При создании инфраструктуры прерывания запросов для LinkedIn мы использовали стандартизированную среду REST с открытым исходным кодом Rest.li, используемую в большинстве производственных приложений компании. Предсказуемая структура межсервисных связей упростила конструкцию LinkedOut и позволила оказывать существенное влияние на наш стек. Также чрезвычайно полезна подключаемая архитектура Rest.li, поддерживающая настраиваемые цепочки фильтров, которые оцениваются при каждом запросе и ответе.

В этих цепочках разработчик или владелец сервиса может добавить фильтр, имеющий полный доступ ко всем входящим и исходящим соединениям клиента или сервера, и манипулировать запросом или ответом по мере необходимости. В этом контексте указывается, какая служба сделала вызов,

---

[1] Слово *site* здесь применяется в широком смысле и означает практическую деятельность на разных уровнях. Иногда SRE называют инженерами по безопасности эксплуатации. – *Прим. перев.*

[2] *Logan Rosen.* LinkedOut: A Request-Level Failure Injection Framework // LinkedIn Engineering, May 24, 2018, https://oreil.ly/KkhdS.

[3] *Bhaskaran Devaraj and Xiao Li.* Resilience Engineering at LinkedIn with Project Waterbear // LinkedIn Engi-neering, Nov. 10, 2017, https://oreil.ly/2tDRk.

какой ресурс предназначен для вызова, какой метод используется и другие соответствующие данные.

Учитывая это, мы решили написать и включить в цепочку фильтров по умолчанию *прерыватель* Rest.li, который может вводить отказы в запросы на стороне клиента. Прерыватель охватывает весь контекст, предоставляемый средой Rest.li, и позволяет выбирать, когда и как прерывать запросы, что помогает нам минимизировать радиус поражения за счет детализации экспериментов.

Мы решили начать со сбоя на стороне клиента, поскольку он снижает барьер входа в эксперименты с хаосом – вместо того чтобы вызывать отказ в нисходящем сервисе, у вас есть полный контроль над имитацией отказа на стороне клиента, и это также минимизирует риск. Если бы вы добавили задержку к нескольким запросам в нижестоящем сервлете, то могли бы связать все его потоки и вызвать его полное падение. Вместо этого имитация сбоев в работе клиента, хотя она, может, и не дает полной картины того, как отложенные запросы повлияют на все отложенные службы, позволяет проверить влияние на стороне клиента и одновременно снижает вероятность ущерба для служб, на которые вы полагаетесь.

Почему Waterbear?

Рис. 7.2 ❖ Логотип Waterbear («водяной медведь»)

Микроорганизм *Tardigrada* (также известный как *тихоходка*, или «водяной медведь», из-за своего внешнего вида) является уникальным существом, которое может жить практически в любых условиях, от горячих источников до горных вершин и открытого космоса. После воздействия экстремальных условий, таких как атмосферное давление или температура, тихоходки с легкостью приходят в норму[1].

Они служат примером надежности, которую мы бы хотели видеть у наших сервисов, тестируемых при помощи инструментов LinkedOut.

7.4.1. Режимы отказа

Мы выбрали для прерывателя режимы отказов, ежедневно наблюдаемых инженерами по эксплуатации и представляющих большинство проблем,

[1] *Cornelia Dean.* The Tardigrade: Practically Invisible, Indestructible 'Water Bears' // The New York Times, Sept. 7, 2015.

с которыми сталкиваются интернет-компании. Мы намеренно старались не модифицировать тела ответов (например, изменять коды ответов или создавать искаженный контент), поскольку знали, что такие ошибки лучше подходят для тестирования интеграции системы и представляют угрозу безопасности. Базовые режимы отказов, наоборот, просты, но эффективны и являются отличными предикторами того, как приложения LinkedIn будут реагировать на реальные сбои.

Исходя из этого, мы встроили в разрушитель три режима отказа:

Ошибка

Первый режим – *ошибка*. Существующие во всевозможных видах и формах, ошибки, вероятно, являются наиболее распространенными проблемами, с которыми сталкиваются специалисты по эксплуатации. Платформа Rest.li имеет несколько исключений по умолчанию, возникающих при проблемах связи или обмена данными с запрошенным ресурсом. Чтобы смоделировать недоступность ресурса, мы вызываем универсальное исключение Java внутри фильтра, которое всплывает как `RestliResponseException`.

Пользователи LinkedOut могут настроить величину задержки между внедрением отказа и выдачей исключения. Эта функциональность позволяет имитировать время, которое может потребоваться нисходящему сервису для обработки запроса перед выдачей ошибки, а также дает возможность избегать странных отклонений в показателях латентности на стороне клиента, когда нисходящий сервис неожиданно начинает обработку запросов в короткий промежуток времени.

Задержка

Второй режим отказа – задержка; вы можете настроить паузу, и фильтр задержит запрос на указанное время, прежде чем передать его в нисходящем направлении. Задержка ответа нисходящего сервиса – это также распространенная проблема, возникающая, как правило, из-за проблем с базой данных или других зависимых сервисов. Задержки могут вызвать каскадные сбои в сервис-ориентированной архитектуре, поэтому важно понимать их влияние на стек.

Время ожидания

Последним, но не менее важным является режим превышения *времени ожидания* (тайм-аут), когда прерыватель удерживает запрос в течение заданного периода времени, пока клиент Rest.li не сгенерирует исключение `TimeoutException`. В этом режиме отказа используется настроенный параметр тайм-аута для целевой конечной точки, что позволяет LinkedOut выставить чрезмерно большие тайм-ауты, посмотреть, что из этого выйдет, и улучшить взаимодействие с пользователем.

Возможность выбора между режимами ошибки, задержки и времени ожидания в LinkedOut дает разработчикам значительный простор для экспериментов, позволяя точно моделировать реальные производственные инциденты. Мы точно знаем, как работают эти режимы, и, следовательно, можем быть уверены в величине радиуса поражения.

7.4.2. Использование LiX для нацеливания экспериментов

Наличие фильтра прерывателя LinkedOut в цепочке фильтров Rest.li позволяет нам подключиться к нашей внутренней структуре нацеливания экспериментов под названием LiX (рис. 7.3). Эта структура дает возможность иметь очень высокую степень детализации при выборе цели эксперимента и предоставляет API-интерфейсы, которые обеспечивают высокоточное нацеливание с помощью пользовательского интерфейса, включая возможность досрочно завершать эксперименты с помощью «большой красной кнопки».

Мы начали внедрение масштабных хаос-экспериментов именно с этого механизма интеграции, поскольку среда LiX позволяет нам ограничить влияние эксперимента строго заданной целью. Мы создали схему, которая основана на концепции «воздействий» LiX и описывает, как именно должен произойти отказ и для кого (рис. 7.3).

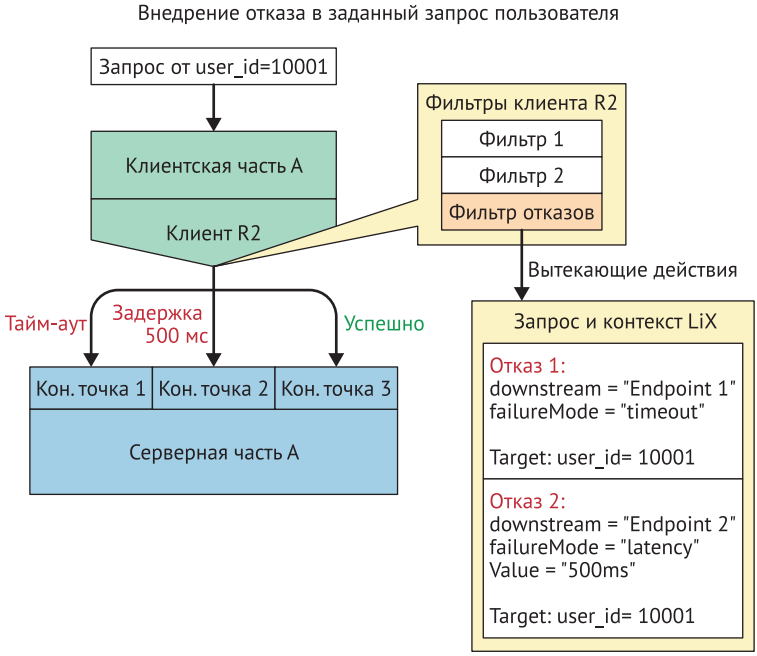

Рис. 7.3 ❖ Схема прерывателя на основе LiX в LinkedOut

LiX – чрезвычайно мощная платформа. Она может использовать так называемые селекторы для нацеливания на сегменты сообщества пользователей, от одного отдельного пользователя до, например, всех пользователей, которые живут в Соединенных Штатах и говорят по-английски, или просто на определенный процент от всех наших пользователей. С этой мощностью связаны большая ответственность и необходимость соблюдать предельную

осторожность при проведении эксперимента, поскольку неправильная цифра или буква может привести к последствиям, на которые вы совсем не рассчитывали.

Нам очень быстро стало ясно, что хороший пользовательский интерфейс очень важен с точки зрения безопасности и спокойствия пользователей нашей платформы. В начале 2018 года жители Гавайских островов ошибочно получили оповещение о ракетной атаке, что вызвало массовую панику. Вместо заурядной проверки систем оповещения было отправлено уведомление о реальной атаке ракет с ядерными боеголовками. И хотя в последующих отчетах говорилось, что сотрудник имел основания думать, что ракеты уже в пути[1], расследование инцидента выявило пользовательский интерфейс, спроектированный таким образом, чтобы легко провоцировать ложную тревогу.

На скриншотах, предоставленных правительственными экспертами, которые имитировали реальный пользовательский интерфейс, используемый для оповещения, ссылка на отправку проверочного оповещения была размещена прямо над ссылкой для реальных оповещений, в том же раскрывающемся меню[2]. Подобное устройство интерфейса не может обеспечить надежную работу и провоцирует ошибки. Независимо от того, намеренно ли работник нажал на ссылку, чтобы отправить реальное предупреждение жителям, или нет, здесь есть очевидный урок: пользовательские интерфейсы должны исключать неоднозначность для людей, особенно если они выполняют критически важную работу. Ясность в пользовательском опыте имеет большое значение.

Несмотря на то что у LiX был собственный пользовательский интерфейс для нацеливания экспериментов, мы решили, что лучший выход – это разработка специального интерфейса нацеливания для экспериментов, ориентированных на отказ (LinkedOut) (рис. 7.4). Он преднамеренно ограничивает возможности расширения эксперимента, ограничивая выбор целей только теми участниками, которые зарегистрировались на платформе и дали явное согласие на проведение целевых экспериментов с отказами.

Если кто-то решит, что он достаточно проверил устойчивость службы LinkedIn к конкретным отказам в небольшом масштабе и хотел бы расширить эксперимент до большего числа пользователей, ему потребуется ручное редактирование базового эксперимента LiX и тесное взаимодействие с командой Waterbear. Мы считаем, что здесь важно иметь систему сдержек и противовесов, особенно когда на карту поставлены интересы наших пользователей.

Мы сознательно ограничиваем возможности влияния сотрудников LinkedIn на пользовательский интерфейс, хотя базовая платформа LiX поддерживает нацеливание на реальных пользователей, потому что мы считаем, что владельцы служб могут получить достаточную пользу от внутренних

---

[1] *Cecelia Kang*. Hawaii Missile Alert Wasn't Accidental, Officials Say, Blaming Worker // The New York Times, Jan. 30, 2018.

[2] *Colin Lecher*. Here's How Hawaii's Emergency Alert Design Led to a False Alarm // The Verge, Jan. 28, 2018, https://oreil.ly/Nqb59.

экспериментов, не рискуя повлиять на производство. Кроме того, если бы мы позволили сотрудникам нацеливать эксперименты на пользователей, используя наш пользовательский интерфейс, нам бы пришлось встроить туда средства защиты для автоматического прекращения экспериментов, когда обнаруживаются явные отклонения от устойчивого состояния. Подобные технологии доступны в инструменте для проведения нагрузочных экспериментов Redliner в LinkedIn[1], поэтому в будущем может появиться возможность совместного использования инструментов, если это будет полезно для наших пользователей.

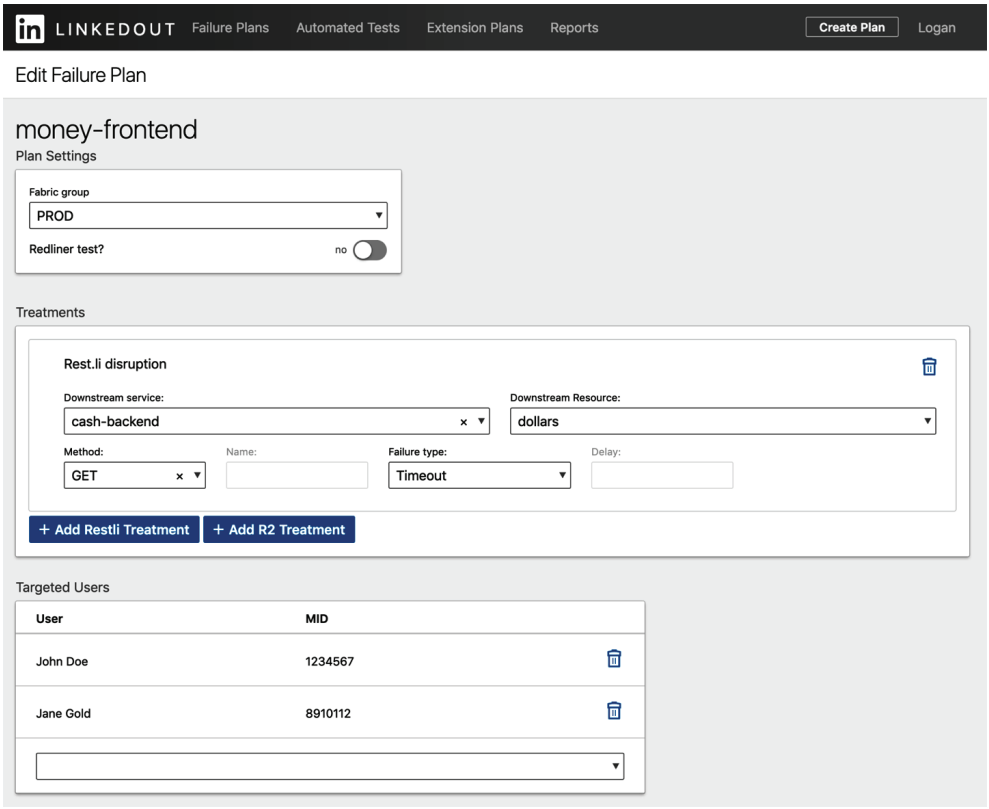

Рис. 7.4 ❖ Снимок экрана нашего пользовательского интерфейса для плана отказов LinkedOut

Конечно, есть вероятность, что даже небольшой эксперимент пойдет не так; поэтому мы используем способность платформы LiX немедленно прекратить любой эксперимент и предоставляем пользователям LinkedOut возможность сделать это в нашем пользовательском интерфейсе. Одним нажатием кнопки сотрудник может мгновенно отправить сигнал о прекращении

[1] *Susie Xia and Anant Rao.* Redliner: How LinkedIn Determines the Capacity Limits of Its Services // LinkedIn Engineering, Feb. 17, 2017, https://oreil.ly/QxSU.

эксперимента LiX через наш производственный стек, который выполняется в течение нескольких минут. Несмотря на то что радиус поражения уже сам по себе достаточно мал для разрешенных по умолчанию экспериментов, наша «большая красная кнопка» позволяет легко завершить работу при неожиданном исходе.

7.4.3. Браузерное расширение для быстрых экспериментов

Несмотря на то что наше приложение на основе LiX отлично подходит для множества пользователей и масштабных экспериментов, мы поняли, что отсутствует еще один компонент успешной среды экспериментов с отказами: возможность быстрого эксперимента в вашем собственном браузере без необходимости ждать распространения эксперимента по производственному стеку. Значительная часть хаос-инжиниринга – это исследование или попытка устроить различные отказы, чтобы посмотреть, как это повлияет на вашу систему, а наше решение на основе A/B-тестирования все-таки тяжеловато для подобных быстрых итераций.

Поэтому мы добавили еще один механизм для внедрения сбоев в запросы через контекст вызова (invocation context, IC). IC является специфичным для LinkedIn внутренним компонентом структуры Rest.li, который позволяет передавать ключи и значения в запросы и распространять их на все службы, участвующие в их обработке. Мы создали новую схему для данных о сбоях, которые могут быть переданы через IC, и тогда для этого запроса мгновенно произойдут отказы.

Механизм внедрения IC открывает возможности для быстрых однократных экспериментов в браузере при помощи ввода данных об отказе через cookie-файлы. Но мы предположили, что никто не захочет вручную создавать cookie-файлы, придерживаясь нашей схемы JSON, только ради того, чтобы проводить эксперименты с отказами. Поэтому решили создать расширение для веб-браузера (рис. 7.5).

Мы предположили, что люди захотят проводить быстрые эксперименты с отказами в своем браузере, поэтому разработали простой процесс:

1) нажмите кнопку, чтобы получить перечень служб, задействованных в запросе;
2) выберите службы, для которых вы хотите внедрить отказы;
3) нажмите кнопку, чтобы обновить страницу с внедренными ошибками.

Чтобы обнаружить нисходящие потоки, мы используем трассировку запросов через внутреннюю среду под названием Call Tree, которая позволяет инженерам устанавливать ключ группировки в виде файла cookie со своими запросами, связывая вместе все обнаруженные нисходящие вызовы. Другими словами, мы разработали браузерное расширение, которое обновляет страницу с помощью набора файлов cookie ключа группировки Call Tree, обнаруживает задействованные нисходящие потоки и затем отображает их в пользовательском интерфейсе (рис. 7.6).

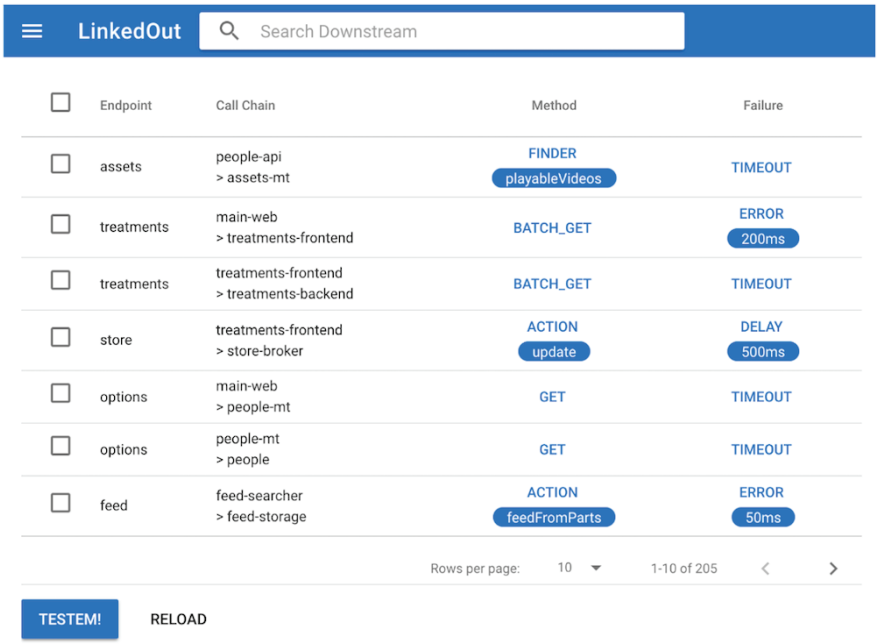

Рис. 7.5 ❖ Браузерное расширение LinkedOut

Рис. 7.6 ❖ Логическая диаграмма процессов в браузерном расширении

В LinkedIn в запросе может быть задействовано несколько служб – фактически их может быть несколько сотен. Поэтому мы добавили окно поиска, которое позволяет пользователю быстро фильтровать конечные точки/службы, которые ему нужны. Кроме того, из-за детализации фильтра прерывания пользователи могут для данной конечной точки вводить сбой только для определенного метода Rest.li.

Как только пользователь выбирает режимы отказа для всех подходящих ресурсов, расширение создает для этих отказов разрушающий BLOB-объект JSON, формирует cookie-файл для внедрения его в IC, а затем обновляет страницу с примененным отказом. Это простой «бесшовный» процесс, требующий небольших трудозатрат пользователя.

Главное достоинство браузерного расширения заключается в том, что радиус поражения практически не существует – сотрудники могут практиковать отказы на уровне запросов только в их собственном браузере, которым и ограничивается влияние запросов. Если они отклонят запрос от нашей домашней страницы до API, который обслуживает канал LinkedIn, это нарушит работу только в их сеансе, а не у кого-либо еще. Для отмены воздействия достаточно нажать кнопку в расширении.

7.4.4. Автоматизированные эксперименты

В дополнение к вышеупомянутым методам запуска отказов в LinkedOut наша платформа также предоставляет среду автоматических экспериментов, которая ограничивает радиус поражения аналогично браузерному расширению, с одним небольшим отличием. В то время как браузерное расширение подразумевает эксперименты в вашем собственном сеансе и влияет только на вашего пользователя, наши автоматические эксперименты проверяют последствия внедрения отказа лишь на служебной учетной записи или на синтетическом пользователе.

Среда автоматических экспериментов в LinkedOut позволяет пользователю ввести URL-адрес, с которым он хочет экспериментировать, отказы, которые он желает применить к каждому нисходящему потоку, вовлеченному в дерево вызовов, учетную запись службы, которую необходимо использовать для эксперимента (в случае если ему необходимо использовать ее со специальными разрешениями), и критерии для сопоставления отказов. Хотя LinkedOut использует критерии по умолчанию для сопоставления со страницами типа «oops» («ой, простите»), пустыми страницами и кодами неправильных ответов, которые указывают на серьезное ухудшение функциональности сайта, платформа также дает возможность с помощью селекторов DOM точно отобразить, в чем заключается ухудшение работы вашего продукта.

После запуска автоматического эксперимента реальные пользователи не испытают никакого воздействия. Сначала эксперимент обнаруживает все службы, задействованные в дереве вызовов для запрошенного URL-адреса в данный момент, так как граф вызовов может со временем меняться, а затем сообщает пулу воркеров (исполняющих процессов) Selenium, что нужно войти в систему как синтетический пользователь и внедрить в запрос отказ через cookie-файл.

По окончании эксперимента пользователю предоставляется отчет, содержащий внедренные отказы и информацию о том, вызвали ли они нарушение страницы в соответствии с заданными критериями (рис. 7.7). Отчет также содержит скриншоты нарушенных страниц, чтобы продемонстрировать, что увидел бы реальный пользователь в результате отказа в производстве; сотрудники считают эту функцию чрезвычайно полезной для проверки критериев ошибок, которые они выбрали для своих экспериментов.

В этом отчете ответственные за разработку приложения могут пометить определенные нисходящие потоки как критические: это делается, если для данного приложения недопустимо постепенное ухудшение качества, и любое несоответствие ожидаемому результату должно приводить к сообщению об ошибке. Иногда почти невозможно обеспечить пользователю хороший опыт в случае отказа критической службы. В подобной ситуации мы хотим дать разработчикам возможность пометить эти ребра в графе вызовов как критические и исключить их из перечня отказов в будущих отчетах.

Наши эксперименты также автоматизированы в том смысле, что они могут выполняться регулярно: ежедневно, еженедельно или ежемесячно. Запланировав эти эксперименты, разработчики могут узнавать о любых регрессиях или исправлениях, влияющих на стойкость их служб к нисходящим отказам. В планах – возможность запуска автоматических тестов после выпуска нового кода, что даст разработчикам дополнительный сигнал о том, как их изменения влияют на устойчивость.

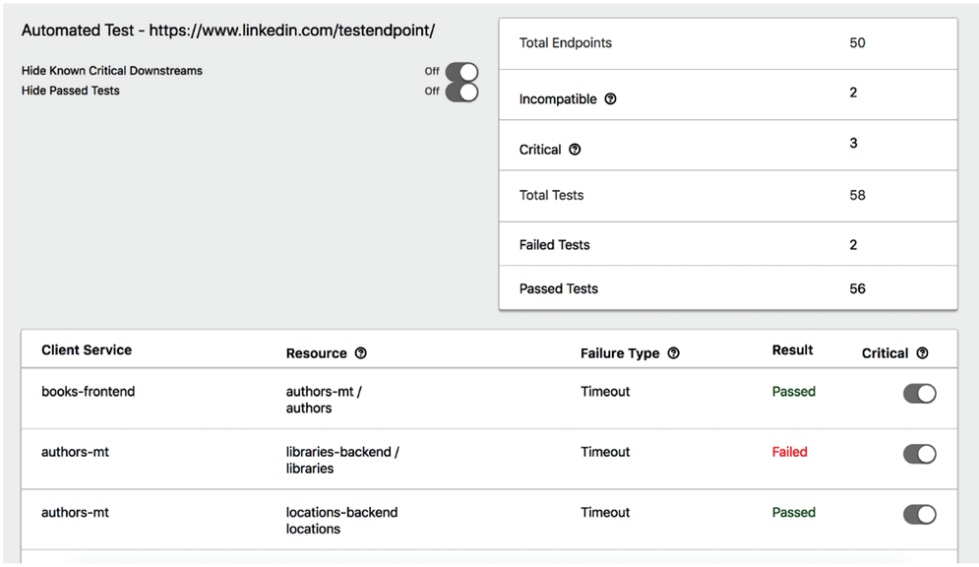

Рис. 7.7 ❖ Пример автоматического отчета об эксперименте

Автоматические эксперименты в LinkedOut – это ключевой компонент регулярной проверки надежности наших служб с помощью интуитивно понятных отчетов. Ограничивая эксперимент служебной учетной записью или

тестовым пользователем, он минимизирует радиус поражения и избегает воздействия на пользователей LinkedIn.

7.5. Вывод

Даже самые тщательно спланированные эксперименты могут иметь непредвиденные последствия, поэтому у вас должны быть предусмотрены меры предосторожности для прекращения эксперимента и ограничения воздействия на пользователей. Самый безопасный вариант – всегда экспериментировать в небольшом масштабе, особенно когда последствия ваших экспериментов на пользователях непредсказуемы.

Надеемся, что примеры безопасной работы в LinkedOut – ограниченное нацеливание с LiX, быстрые эксперименты в браузерном расширении и автоматические эксперименты со служебными учетными записями – вдохновят вас попробовать подобные идеи в вашей собственной системе. Ограничившись только внутренними экспериментами на сотрудниках, мы все еще можем получить высокую отдачу от хаос-экспериментов без риска испортить отношения с пользователями.

Простота использования и функциональность LinkedOut позволили сосредоточиться на надежности при разработке нового программного обеспечения для LinkedIn, и этот инструмент продолжает проверять отказоустойчивость, которую разработчики закладывают в свои приложения. Наши текущие и ожидаемые функции автоматизации экспериментов нацелены на проверку устойчивости приложений к отказам в стеке.

Как только вы научились проверять устойчивость системы к отказам с небольшим радиусом, переходите к более масштабным экспериментам – они тоже важны. Если ваш эксперимент способен затронуть многих пользователей, вы должны внимательно следить за показателями системы и иметь возможность быстро прекратить эксперимент при появлении признаков отклонения от устойчивого состояния. Если вы хотите провести несколько масштабных экспериментов, настоятельно рекомендуем встроить автоматическую защиту, которая прекращает эксперименты, когда это неожиданно сказывается на показателях вашего бизнеса и опыте участников.

Об авторе

Логан Розен – штатный инженер по эксплуатационной надежности в LinkedIn в Нью-Йорке. Являясь одним из основателей проекта Waterbear, он помог запустить LinkedOut, инфраструктуру внедрения отказов на уровне запросов, а также внедрил в организацию хаос-инжиниринг.

Глава 8

Развитие хаос-инжиниринга в Capital One

Автор главы: **Раджи Чокайян**

Хаос-инжиниринг дает представление о том, как бороться с отказами, и выявляет недостатки системы в свете этих отказов. В облачной среде данный подход к разработке программного обеспечения нельзя игнорировать. Облачные системы по сути своей допускают, что сбой может произойти в любое время, и нам приходится проектировать и строить надежные системы, способные противостоять всему спектру отказов. Нельзя забывать, что, в отличие от других отраслей, программное обеспечение в секторе финансовых услуг намного более жестко контролируется и имеет несколько уровней сложности. Во время сбоя системы один клиент может подавать заявку на кредит для своего первого дома, другой клиент может пытаться перевести средства с помощью мобильного телефона, а студент может подать заявку на получение своей первой кредитной карты, чтобы начать кредитную историю. В зависимости от масштаба сбоя воздействие будет варьироваться от легкого раздражения до неприязни к бренду, что негативно отражается на репутации финансового учреждения и потенциально влияет на бизнес.

Кроме того, существуют надзорные органы, наблюдающие за тем, как банки ведут свою деятельность. Банки обязаны формировать, хранить и регулярно представлять финансовым регуляторам подробные отчеты. Поэтому к любому изменению в производственной среде должна быть привязана четкая контрольная информация о том, что произошло, почему, как и когда. Банки давно внедрили процессы и инструменты для сбора и представления отчетов в соответствующие органы. Отчетность имеет важные юридические последствия и не должна зависеть от таких методов или инструментов, как хаос-инжиниринг.

С другой стороны, с ростом числа цифровых банков и необанков[1] меняется способ взаимодействия клиентов со своими деньгами. Финансовые техноло-

[1] Необанк – это полностью цифровой виртуальный банк, обслуживающий своих клиентов только через мобильные приложения и компьютерные платформы.

гии, основанные на блокчейне, искусственном интеллекте, машинном обучении и бизнес-аналитике, создали потребность в высоконадежных и масштабируемых системах, которые предоставляет облачная инфраструктура. Это стимулирует эволюцию методов разработки программного обеспечения и заставляет внедрять правильные методы проектирования. Наряду с тем, как автоматизированные развертывания обладают повышенной скоростью, а неизменная инфраструктура гарантирует, что развернутые серверы никогда не будут изменены, системы должны постоянно проверяться на надежность. В этот момент и возникает потребность в хаос-инжиниринге. Есть несколько подходов к началу работы. В одних случаях метод может быть внедрен извне, а в других – изнутри, как часть генотипа компании. В этой главе мы рассказываем об истории хаос-инжиниринга в банке Capital One, а также обсуждаем факторы, которые необходимо учитывать при внедрении; мы рассмотрим вещи, на которые следует обратить внимание при разработке эксперимента для финансовой организации, инструменты для создания контрольных журналов, а также показатели, за которыми надо следить во время проведения эксперимента.

8.1. Практический опыт Capital One

Capital One, крупнейший онлайн-банк[1] в Соединенных Штатах, известен тем, что он является технологическим лидером в индустрии финансовых услуг и последние семь лет совершает масштабную цифровую трансформацию. Он решает задачи, стоящие перед многими американскими предприятиями: использование цифровых технологий, аналитики в реальном времени и машинного обучения для повышения качества обслуживания клиентов и роста бизнеса. Банк полностью переосмыслил свою инфраструктуру, от персонала и культуры до технологий, создав собственные облачные приложения и инструменты для внедрения сложных методов проектирования, такие как конвейеры CI/CD[2], шаблонные структуры со встроенными стандартами соответствий и нормативов, скрытое управление и хаос-инжиниринг.

8.1.1. Слепое тестирование устойчивости

Хаос-инжиниринг практиковался в Capital One еще до начала перехода в облако. Одна из основных платформ API, обслуживающая мобильные и веб-порталы, переняла эту практику и образ мышления, когда размещалась в физических центрах обработки данных. По словам Гиты Гопал, технического директора Capital One, потребность во внедрении хаос-инжиниринга объясняется необходимостью полного понимания всех факторов – внутренних,

---

[1] Онлайн-банк – это банк, который предоставляет свои услуги удаленно. Его присутствие в интернете больше, чем сеть физических отделений.

[2] Continuous integration / continuous deployment, непрерывная интеграция / непрерывное развертывание. – *Прим. перев.*

внешних, приложений, оборудования, точек интеграции, – которые могут повлиять на время безотказной работы сервиса. Они назвали свой подход «Слепое тестирование устойчивости». В 2013 году они сформировали две группы, отвечающие за оркестровку процесса: *группу отказа* и *группу реагирования*.

Интересно, что ни одна из этих групп не являлась прикладной командой, поддерживающей основную платформу API. Для участия в этих учениях, проводимых через регулярные промежутки времени, один раз в месяц или после каждого обновления, также приглашали группы по кибербезопасности и мониторингу. Они должны были следить за тем, чтобы эксперименты выполнялись и анализировались надлежащим образом. Команды начали со списка, включающего почти 25 экспериментов. Группа отказа случайным образом выбрала два эксперимента и выполняла их, а группа реагирования наблюдала, отвечала и документировала результаты. Эксперименты запускали из DMZ (demilitarized zone, обособленный сегмент сети) с использованием нескольких синтетических учетных записей, генерирующих нагрузку. Все работы выполнялись в отладочной среде. За последующие несколько лет количество экспериментов значительно увеличилось, и группы отказа/реагирования стали демонстрировать ощутимую отдачу: меньшее количество обновлений, вызывающих инциденты, и ускорение отклика на внешние и аппаратные сбои. Они всегда фокусировались на серверных приложениях, где видели потенциальные точки отказа.

8.1.2. Переход к хаос-инжинирингу

Перенесемся в 2018 год. Теперь команда поддержки основного API запускает свою платформу исключительно в облаке, а эксперименты проводятся в производственной среде. На этот раз их называют хаос-экспериментами. Исследователи используют собственные инструменты, помогающие планировать повторяющиеся хаос-эксперименты и автоматически запускать их в назначенное время и с определенным интервалом. Они проводят каждый отдельный эксперимент в отдельной отладочной среде, чтобы убедиться, что все точки отказа обнаружены и исправлены до переноса обновления в производство. Благодаря этому большая часть полезных результатов извлекается из непроизводственных экспериментов.

Зачастую компании и группы разработчиков испытывают трудности с расстановкой приоритетов между инженерным мастерством и интересами бизнеса, которые напрямую влияют на рентабельность инвестиций. Одним из показателей успеха является уменьшение количества обращений в техподдержку. Чем больше экспериментов переносит платформа, тем меньше инцидентов на производстве, что снижает количество звонков от недовольных или растерянных клиентов. Иными словами, чем реже звонят телефоны колцентра, тем лучше для банка. Этот эффект легко переводится в денежное выражение (например, количество занятых сотрудников клиентского отдела, время, потраченное специалистами на поиск и устранение неисправностей и решение проблем) и служит показателем успеха, который можно отслеживать. В результате деловые партнеры в рамках Capital One теперь заинтере-

сованы в проведении этих экспериментов. Разработчики тоже ориентируются на снижение количества обращений в техподдержку, когда оценивают эффективность экспериментов и составляют приоритетный список. В этом случае имеют значение и другие показатели, такие как задержка, частота отказов в обслуживании, время отработки отказа, задержка автоматического масштабирования и насыщение ресурсов, таких как процессор и память.

Команда разработчиков платформы API использует интересный подход к предупреждению и мониторингу. Получение аварийного оповещения во время эксперимента означает, что их система провалила эксперимент, потому что в надежной системе запросы должны быть перенесены в альтернативную облачную область, или должно сработать автоматическое масштабирование.

Эти эксперименты и их результаты также периодически проверяются регуляторами и аудиторами в связи с критичностью возможных последствий. Внутри банка создано сообщество специалистов по хаос-инжинирингу. Это форум для всех команд, выполняющих хаос-эксперименты, где они делятся знаниями и опытом.

8.1.3. Хаос-эксперименты в CI/CD

Есть и другие команды, которые интегрировали хаос-инжиниринг в свой обычный жизненный цикл разработки программного обеспечения. Для них было недостаточно использовать эту новую классную технологию только для игр в изолированной отладочной среде. Некоторые сервисы в любой день обслуживают более десяти тысяч транзакций в секунду. Некоторые из них выступают в качестве уровней интеграции между двумя основными системами в Capital One. Банку больше всего нужны высокоэластичные, масштабируемые, доступные системы и API-интерфейсы. Хаос-инжиниринг – это хороший способ убедиться в том, что все системы банка соответствуют бизнес-целям.

Команды разработчиков начинают с углубленного анализа своих проектных решений в поисках потенциальных точек отказа. Затем они проверяют их при помощи хаос-экспериментов, которые запускаются как часть конвейеров CI/CD. У Capital One есть очень качественно проработанный конвейерный процесс с жесткими правилами и нормами безопасности. Группы, которые доказали свое соответствие этим требованиям, получают предварительное разрешение для выпуска своего кода в производство без какого-либо ручного тестирования.

Некоторые команды запускают хаос-эксперименты в своей инфраструктуре как часть процесса CI/CD. Они проверяют надежность нового стека инфраструктуры до переключения трафика на новую кодовую базу, что позволяет им убедиться в масштабируемости и устойчивости обновления. Другие команды используют эксперименты как часть инструментария для проверки надежности системы при пиковой нагрузке. В основном это проверка их проектов автоматического масштабирования и протоколов отработки отказа региона.

Команда разработчиков платформы Capital One оказывает необходимую поддержку этим группам и обеспечивает их необходимыми инструментами. Наш главный ориентир – это пользователи (как наши внутренние команды разработчиков, так и конечные потребители банковских услуг), и каждое решение принимается из расчета улучшить их жизнь. Каждая команда, по-своему внедряющая хаос-инжиниринг для создания устойчивых систем, в конечном счете работает в интересах клиентов.

8.2. Чего нужно остерегаться при разработке эксперимента

Разработка продуманного хаос-эксперимента часто требует больше времени и ресурсов, чем разработка функции. До тех пор, пока система не будет готова к проведению экспериментов с хаосом на производстве, выполнение их в среде более низкого уровня, имитирующей производство, также может повлечь дополнительные расходы. Настройка достаточно мощной отладочной среды и обеспечение доступности зависимых сервисов требуют времени и усилий. Этот момент важно учитывать на ранних этапах внедрения хаос-инжиниринга. Возможность создания шаблонов для основных отказов чрезвычайно помогает в последовательных прогонах, хотя и требует значительных предварительных вложений. Например, если отключение хост-компьютера для проверки срабатывания автоматического масштабирования является одним из наиболее частых отказов, то наличие настраиваемого шаблона для совместного использования группами сэкономит время и усилия.

На ранних стадиях разработки эксперимента хорошее понимание и документация по следующим вопросам помогут команде обрести уверенность:

- ясное понимание ожидаемого поведения;
- потенциальные/возможные сбои;
- влияние на выполняемые транзакции;
- мониторинг инфраструктуры и приложений;
- критичность точки отказа, которую вы пытаетесь проверить;
- оценка риска для каждого эксперимента.

После проведения экспериментов решающее значение имеет документирование наблюдаемого поведения.

В отладочной среде эти эксперименты удобно выполнять в течение дня, чтобы вся команда была на месте, если что-то пойдет не так. В компаниях, предоставляющих финансовые услуги, дорого обходится любой отказ, преднамеренный или нет. Если эксперименты выполняются на производстве, важно выбрать время, когда воздействие будет минимальным, особенно если в ходе эксперимента будет определена новая точка отказа.

Сохранение подробной аудиторской записи событий очень важно для любого эксперимента, связанного с синтетической информацией о клиенте или бизнес-транзакцией, например кто запланировал этот эксперимент, было ли

получено разрешение на его проведение в данной конкретной среде, произошел ли сбой в системе, подвергшейся эксперименту, какие процедуры уведомления и эскалации настроены для решения проблемы и во избежание воздействия на клиента и т. д. Это помогает получить целостную картину в дополнение к регистрации и/или отслеживанию, которые более информативны в отношении работоспособности приложения.

Когда команды приступают к экспериментам с хаос-инжинирингом, они начинают с малого, например с отключения экземпляра внутри группы автоматического масштабирования в отладочной среде. Обычно это делается вручную, при этом некоторые члены команды отслеживают и документируют весь процесс. По мере расширения масштаба воздействия ручная поддержка эксперимента может стать слишком трудоемкой. В этой ситуации помогут следующие меры:

- автоматизация экспериментов с помощью надежного инструмента;
- настройка правильного мониторинга и оповещения;
- разработка плана поддержки для команд по мере их роста.

8.3. Инструментарий

В хаос-инжиниринге есть три основных этапа:

1) разработка экспериментов, которые могут подтвердить очень специфическое предположение об архитектуре вашей системы;
2) выполнение этих экспериментов согласно плану;
3) изучение результатов экспериментов.

Все эти шаги вполне можно выполнить вручную. Но настоящую выгоду от хаос-инжиниринга приносит регулярное выполнение масштабных экспериментов. Для этого необходимо иметь соответствующий инструмент, который упрощает эти три процедуры.

Существуют как открытые, так и лицензионные корпоративные инструменты. Большинство из них специализируются на отказах инфраструктуры и сети, которые могут быть внедрены в выбранную среду. Некоторые из них также имеют приборную панель, чтобы легче было понять, что происходит после запуска эксперимента. Преимущество повторного использования существующего инструмента состоит не в изобретении велосипеда, а в том, чтобы сосредоточить энергию и время на создании новых возможностей для бизнеса.

Некоторые компании по ряду причин предпочитают создавать свои собственные инструменты. В этом случае не обойтись без бюджета и талантливых компетентных разработчиков. Capital One относится к числу тех, кто разработал свой собственный инструментарий для хаос-инжиниринга. Комплексный подход к архитектуре микросервисов требует более высокого уровня отказоустойчивости в масштабах предприятия, чем тот, который предлагается в сторонних инструментах. Устойчивость инфраструктуры важна для Capital One ничуть не меньше, чем безопасность и управление рисками. Собственная платформа хаос-инжиниринга полностью отвечает всем

требованиям банка. Планирование, выполнение и наблюдение ограничены рамками компании, что исключает риск утечки данных из сети. Кроме того, если после запуска эксперимента потребуется какая-либо особая аудиторская отчетность, ее можно встроить в инструмент.

Благодаря внутреннему инструменту SaaS данные не покидают предприятие. Все обмены данными происходят в пределах корпоративной сети. Это снижает некоторые риски безопасности, такие как раскрытие зашифрованных данных. Поверх стандартных отказов, таких как перегрузка сети трафиком, отключение сервера или блокировка базы данных, можно наложить заранее настроенные эксперименты, специфичные для банковской функциональности (например, те, которые могут нарушить некоторые основные транзакции). Поскольку и инструментарий, и группы приложений работают с одной и той же целью, подготовку и настройку можно выполнить быстро, с более коротким циклом обратной связи по сравнению с тем, когда приходится взаимодействовать с внешними сторонними компаниями.

В банковской сфере наблюдаемость и аудит столь же важны, как и способность разрабатывать индивидуальные эксперименты. На самом деле это крайне важно даже в менее зарегулированной среде. Отказ вследствие эксперимента должен быть отслежен и зафиксирован для последующего устранения неполадок и обучения других команд. Сбор необходимых данных из инструмента, журналов и трассировок приложений до, во время и после экспериментов помогает командам выполнить *условие адекватности* (способность различать реальные и искусственные отказы), а также создавать отчеты о работоспособности инфраструктуры и определять разницу между наблюдаемым и ожидаемым поведением и потенциальным воздействием на транзакции.

Иными словами, очень важно выбрать правильный инструмент хаос-инжиниринга, который соответствует бизнес-задачам именно вашей компании. Как только инструмент готов к работе, следующим критически важным фактором становится правильная структура команды, которая будет поддерживать этот инструмент.

8.4. Структура команды

Если организация решает создать свои собственные инструменты, крайне важно, чтобы структура команды хаос-разработчиков полностью соответствовала культуре и методам работы компании – от этого в значительной степени зависит успех внедрения новых инструментов. Наиболее распространенная модель представляет собой основную Agile-команду из штатных разработчиков и менеджера по продукту. Такая команда хорошо владеет концепциями хаос-инжиниринга и набором навыков для экспериментов. Однако у этой модели есть некоторые недостатки: например, за создание функциональности отвечает только основная команда, и может возникнуть отставание в разработке функций с низким приоритетом из-за банальной

нехватки человеко-часов. Кроме того, этой команде не хватает знаний в прикладной области и бизнес-логике системы. Хотя они могут читать журналы и трассировки, но плохо понимают бизнес-контекст того, как система отреагировала на сбой.

Создание постоянного рабочего канала с другими заинтересованными сторонами (включая группы по кибербезопасности, облачной инженерии, соответствию нормативам, аудиту и управлению, а также команды по контролю и наблюдаемости) имеет решающее значение для успешного внедрения хаос-инжиниринга. Этот момент часто упускается из виду, хотя взаимодействие должно быть реализовано с самого начала.

Некоторые компании, которые хорошо продвинулись во внедрении Agile-методики, пытаются экспериментировать и придумывать креативную, но при этом устойчивую структуру. Capital One, например, верит в совместное создание программного обеспечения. Помимо основной команды, у них также есть выделенный инженер или группа инженеров из команды разработчиков приложений, которые могут быть назначены для работы с основной командой, особенно на этапе прикладного использования платформы хаос-инжиниринга. Они будут работать с экспертами основной группы над выявлением точек отказа их архитектуры приложений и разработкой эксперимента в соответствии с тем, что они хотят раскрыть. Взаимодействие можно устроить разными способами, таким как игровые дни, собрания сообщества или простые встречи один на один. После запуска запланированных экспериментов члены специальной группы наблюдают за системой на предмет аномалий и документируют свои наблюдения. Если инструмент нуждается в техподдержке, то обращаются к группе, отвечающей за инструмент.

Если от инструмента ждут каких-то возможностей, специфичных для данной архитектуры, основная группа разработчиков может самостоятельно разработать нужную функцию и создать запрос на включение в код инструмента. Таким образом, команды разработчиков приложений не зацикливаются на ожидании обновления инструмента, и в то же время они могут вводить новшества и делать свой вклад доступным для корпоративного использования. Подобный подход предотвращает еще одну серьезную проблему: дублирование инструмента. На крупных предприятиях, какими бы хорошими ни были внутренние коммуникации, благодаря инновациям в современных методах разработки программного обеспечения одна и та же проблема должна решаться несколькими группами одновременно, что приводит к увеличению количества инструментов и влечет за собой дополнительные проблемы с затратами и управлением. Наличие команд, вносящих свои инновации, помогает в консолидации инструментов. Их работа также может включать создание шаблонов экспериментов, пригодных для других команд.

8.5. Продвижение хаос-инжиниринга

Любая успешная технология или практика разработки всегда вызывает интерес в первую очередь на низовом уровне, среди энтузиастов. Чтобы этот интерес превратился в масштабное явление, нужна четкая стратегия продвижения, как по вертикали, так и по горизонтали, инструменты для упрощения процесса и платформы для обмена историями успеха между командами разработчиков, представителями бизнеса и надзорными органами. Проведение целевых игровых дней, семинаров и вебинаров – это простые, но эффективные методы продвижения новой практики.

В сообществе прикладных специалистов, которое одна из команд в Capital One основала внутри организации, они собрали инженеров из разных подразделений, связанных общими интересами. Они создали базу знаний с набором информации, включая пособия для начинающих, доступные инструменты, примеры проведения игровых дней, передовые методы обеспечения устойчивости и идеи по автоматизации. Участники сообщества сформировали рабочие группы добровольцев, которые взяли на себя ответственность за каждое из этих направлений. Эта группа регулярно собирается, чтобы обменяться опытом, и значительно выросла в размерах. Благодаря такому обмену знаниями на низовом уровне растет интерес к хаос-инжинирингу и среди специалистов других компаний. Мы видели несколько примеров того, как участники сообщества убеждали менеджеров провести хаос-эксперименты на своем предприятии.

Наконец, в качестве рычагов продвижения хаос-инжиниринга можно использовать нормы и правила, которыми руководствуются финансовые организации. Одним из важнейших требований является поддержание целостности транзакций и данных. Хаос-инжиниринг помогает выявить дефекты устойчивости систем и инфраструктуры, которые могут потенциально повлиять на соблюдение норм. Если новый метод повышает уверенность в стабильной работе системы и соблюдении норм, он всегда найдет поддержку и понимание у высшего руководства компании.

8.6. Вывод

Как и любая крупная отрасль, сектор финансовых услуг предъявляет уникальные требования к разработке программного обеспечения. Это особенно справедливо для таких подходов, как хаос-инжиниринг, потенциально способных нарушить работу конечного пользователя. Разработчики надежных систем нуждаются в средствах проверки и подтверждения надежности. На эту роль идеально подходит хаос-инжиниринг с его богатым инструмен-

тарием, тщательно продуманным подходом к планированию экспериментов и выбором метрик, а также структурой команды, которая соответствует культуре организации. У себя в Capital One мы разработали и внедрили методику, которая наилучшим образом соответствует основным ценностям и миссии нашей компании. Наши инженеры находят наилучшие решения поставленных задач только благодаря передовым идеям, и другого пути у нас нет.

Об авторе

Раджи Чокайян – старший управляющий по разработке программного обеспечения в банке Capital One. Она возглавляет корпоративную инфраструктуру и платформы управления контейнерами, которые обеспечивают логистическую цепочку от ноутбука разработчика до конечного пользователя. Вместе со своей командой разрабатывает стратегии, планы и платформы для тестирования и повышения надежности микросервисов Capital One.

Часть III

ЧЕЛОВЕЧЕСКИЕ ФАКТОРЫ

Устойчивость создается людьми. Разработчики, создающие функциональный код, инженеры по эксплуатации, обслуживающие систему, и даже руководство, которое выделяет ресурсы, являются частями сложной системы. Мы называем ее *социотехнической системой*.

К концу этой части мы надеемся убедить вас в том, что для эффективного повышения устойчивости необходимо понимать взаимосвязи между техническими факторами и людьми, которые выдают разрешения, финансируют, наблюдают, строят, работают, обновляют и эксплуатируют системы. Хаос-инжиниринг поможет вам лучше понять социально-техническую границу между людьми и машинами.

Нора Джонс открывает эту часть книги главой 9 «Формирование предвидения». Акцентируя внимание на обучении как на способе повышения устойчивости, она объясняет, как иногда наиболее важная часть хаос-инжиниринга начинает работать еще до того, как запущен первый эксперимент. Она также раскрывает взаимосвязь между плановым экспериментом и незапланированными инцидентами: «Инциденты дают нам возможность сесть и посмотреть, как ментальная модель работающей системы одного человека отличается от ментальной модели работающей системы другого человека».

В главе 10 «Гуманистический хаос» Энди Флинер исследует применение хаос-инжиниринга к «социальной» части социотехнических систем. Он спрашивает: «А что, если применить хаос-инжиниринг не только к сложным распределенным техническим системам, которые мы знаем и любим, но и к таким сложным распределенным систем, как организации?» Что происходит, когда перемены в организации осуществляются при помощи хаос-экспериментов?

В главе 11 «Роль человека в системе» Джон Оллспоу исследует контекстно-зависимые отношения между людьми и инструменты для повышения безопасности и доступности систем. Что касается хаос-инжиниринга, в частности, он утверждает, что «мы должны рассматривать этот подход как способ развития гибких и контекстно-зависимых отношений, которыми обладают только люди, поскольку им приходится справляться с растущей сложностью, которая обязательно сопровождает современное программное обеспечение».

Сравните это мнение с позицией Питера Альваро в главе 12 «Проблема выбора эксперимента и ее решение», в которой он утверждает, что все больше полагается на автоматизацию. В частности, он предлагает оригинальный алгоритм исследования пространства потенциальных сбоев системы. Этот алгоритм был разработан, «чтобы заменить экспертов при выборе хаос-эксперимента».

Как и в предыдущих главах книги, мы представляем здесь разнообразные точки зрения на человеческий фактор, чтобы продемонстрировать гибкость и ширину охвата зарождающейся дисциплины и тем самым отметить ее ценность.

Глава 9

Формирование предвидения

Между прошлым, настоящим и будущим надежных компаний существует прочная взаимосвязь. Чтобы предоставлять потребителям компании стабильные и безопасные услуги, компания должна постоянно оглядываться в прошлое, не останавливаясь на достигнутом. Надежные компании не воспринимают прошлые успехи как повод для уверенности. Напротив, они используют их как возможность копать глубже, находить скрытые риски и выявлять истинные причины успехов и неудач.

Помимо построения платформ для тестирования устойчивости и проведения игровых дней, существуют и другие важные компоненты хаос-инжиниринга. У каждого человека свое понимание проблем, идей и ментальная модель системы, и такие вещи невозможно автоматизировать с помощью кода. В этой главе будут рассмотрены три различных этапа хаос-инжиниринга и скрытые цели в рамках каждого этапа, в совокупности представляющие хаос-инжиниринг как способ накопить опыт.

В нашей отрасли хронически обделены вниманием два этапа: «до» и «после», которые, как правило, выполняются одним человеком, обычно так называемым *фасилитатором* (facilitator, организатор-ведущий дискуссии). Это участник, который может выступать в качестве третьей стороны во время эксперимента, но прежде он должен научиться всему, с чем сталкивается команда. Если мы сосредоточимся исключительно на поиске проблем до того, как они станут инцидентами, мы упускаем возможность получить максимальную отдачу от основной цели хаос-инжиниринга, которая заключается в совершенствовании ментальных моделей наших систем.

В этой главе мы сфокусируемся на этапах «до» и «после» разработки экспериментов хаос-инжиниринга и сформулируем важные вопросы, которые необходимо задать себе и коллегам на каждом из этих этапов. Мы также рассмотрим определенную «иронию автоматизации»[1], присущую сегодня хаос-инжинирингу.

Как работники индустрии надежности мы уделяем слишком много внимания тому, как происходят отказы и поломки, и упускаем ценность, которую могут принести нам подготовка (этап «до») и распространение полученных

---

[1] *Lisanne Bainbridge.* Ironies of Automation // Automatica, Vol. 19, № 6 (1983).

знаний (этап «после»). Я буду опираться как на исследования по моделированию экспериментов в других отраслях, так и на личный опыт того, что сработало (и не сработало) при проведении хаос-экспериментов в нескольких компаниях с совершенно разными бизнес-целями. Важно отметить, что каждая фаза цикла требует различного набора навыков и разных типов мышления. Тому и другому вполне можно научить. Этим мы и займемся дальше.

9.1. Хаос-инжиниринг и отказоустойчивость

Прежде чем мы рассмотрим этапы цикла хаос-инжиниринга, давайте определимся с практическими целями. Как мы уже говорили в предыдущих главах, цель хаос-инжиниринга заключается не в том, чтобы ломать разные вещи. Но цель и не в разработке инструментов для борьбы с отказами. Давайте остановимся на минутку, потому что я хочу, чтобы эта мысль отложилась у вас в голове: *хаос-инжиниринг – это не поиск или обнаружение уязвимостей*.

Поскольку хаос-инжиниринг является новым и интересным предметом, основное внимание новичков привлекают необычные инструменты, а настоящие причины, по которым следует изучать хаос-инжиниринг, остаются в тени. Так почему же мы занимаемся хаос-инжинирингом? В чем его истинное назначение?

Хаос-инжиниринг – это построение культуры устойчивости в условиях непредсказуемого поведения системы. Если инструменты могут помочь в достижении этой цели, то это отлично, но они являются лишь одним из возможных средств для достижения цели.

> Инжиниринг устойчивости заключается в выявлении и последующем расширении положительных возможностей людей и организаций, которые позволяют им эффективно и безопасно адаптироваться в различных обстоятельствах. Устойчивость – это не уменьшение отрицательных последствий [ошибок].
>
> – *Сидни Деккер*[1]

Хаос-инжиниринг является лишь одним из способов улучшения наших адаптивных возможностей.

9.2. Этапы рабочего цикла хаос-инжиниринга

В этом разделе мы поговорим о различных этапах рабочего цикла хаос-инжиниринга (до, во время и после эксперимента). Мы специально сосредоточим наше внимание на этапах «до» и «после». Почему? Потому что на этапе «во время» в основном требуется много внимания; столько же (если не больше) поучительных данных и предметов исследования вы найдете на этапах «до» и «после».

---

[1] *Sidney Dekker.* Foundations of Safety Science: A Century of Understanding Accidents and Disasters. Boca Raton: Taylor & Francis, CRC Press, 2019.

Главная цель этапа «до» – заставить товарищей по команде обсудить их различные ментальные модели рассматриваемой системы. Вторичная цель заключается том, чтобы научиться разрабатывать стратегии, лежащие в основе *предсказательного проектирования* экспериментов, то есть уметь предсказывать результат эксперимента до его выполнения. Далее мы обсудим различные приемы проектирования и то, как эти ментальные модели появляются на свет.

9.2.1. Разработка эксперимента

> Все ментальные модели ошибочны, но некоторые полезны.
>
> – *Джордж Бокс*[1]

Инциденты дают нам возможность сесть и разобраться, как ментальная модель действующей системы одного человека отличается от ментальной модели действующей системы другого человека. Инциденты позволяют это сделать, потому что на определенном уровне они опровергают наши представления о том, как система работает и обрабатывает риски.

Допустим, некая гипотетическая Джози подумала, что она может ожидать стопроцентно безотказной работы от своей инфраструктуры обнаружения служб, в то время как Матео, разработчик этой инфраструктуры в вашей компании, уверен, что пользователи этой инфраструктуры никогда даже не сделают такого предположения.

Матео, учитывая его длительный опыт работы в качестве инженера-разработчика, предположил, что никто не будет ожидать стопроцентно безотказной работы системы, а как повторные попытки, так и аварийные восстановления будут обрабатываться службами, использующими инфраструктуру обнаружения служб. Он никогда не включал свое видение в документацию и не закладывал в инфраструктуру обнаружения служб такие вещи, которые могли бы снизить риск, если кому-то придет в голову *предполагать* стопроцентно безотказную работу.

Между тем Джози какое-то время использовала инфраструктуру обнаружения служб для разных надобностей, и ей это нравилось. Это сэкономило ей много времени и энергии на хранении важных пар ключ/значение, а также позволило быстро менять их при необходимости. Обнаружение служб работало именно так, как она хотела, – пока не перестало.

Произошло массовое отключение, частично в результате несоответствия взаимных ожиданий и отсутствия возможности обсудить предположения. Эти предположения были *полностью обоснованными* в контексте каждой стороны. Джози и Матео никогда не обсуждали свои представления. Зачем обсуждать то, что очевидно? И впрямь незачем, правда, пока у вас не случится инцидент.

---

[1] *G. E. P. Box*. Robustness in the Strategy of Scientific Model Building // *R. L. Launer and G. N. Wilkinson (eds.)*. Robustness in Statistics. New York: Academic Press, 1979, p. 201–236.

Вот здесь-то и начинаются эксперименты с хаосом. Хотя они действительно дают нам шанс найти уязвимости в наших системах, еще более важно, что они дают нам возможность увидеть, как наша ментальная модель и предположение о том, что случится, когда произойдет отказ компонента X, отличается от ментальной модели и представления нашего партнера по команде. На самом деле вы обнаружите, что на этапе подготовки люди более открыты для обсуждения этих различий, чем во время инцидента, потому что инцидент не произошел, поэтому присутствует элемент психологической безопасности, который невозможно переоценить. Предварительное обсуждение в безопасной обстановке исключает чувство стыда или незащищенности, которое могло бы возникнуть в результате поломки системы. Мы можем спокойно сосредоточиться на том, чему можем научиться, не беспокоясь, что кто-то может пострадать или оказаться виновником сбоя.

9.3. Инструменты для разработки хаос-экспериментов

Подход команд к выбору цели эксперимента столь же показателен, как и сам эксперимент. Когда я присоединилась к Netflix в начале 2017 года, моя команда работала над созданием инструмента под названием ChAP (рис. 9.1). На верхнем уровне платформа опрашивает конвейер развертывания для указанной пользователем службы. Потом она запускает экспериментальные и контрольные кластеры этой службы и направляет небольшое количество трафика каждой из них. Затем к экспериментальной группе применяется заданный сценарий внедрения отказа, и результаты эксперимента передаются разработчику.

ChAP воспользовался тем, что приложения использовали общий набор библиотек Java, и включал в метаданные входящего запроса уведомление о том, что конкретный вызов должен быть неудачным. Затем метаданные расходятся дальше по мере распространения вызова через систему. ChAP может искать эти метаданные по мере распространения запросов между службами и вычислять процент трафика – достаточно маленький, чтобы ограничить радиус поражения, но достаточно большой, чтобы получить статистически значимый отклик, разделить этот трафик пополам и направить на два кластера – экспериментальный (канареечный) и контрольный (основной).

Довольно аккуратная работа, не так ли? Многие организации выпускают свои собственные версии этого инструмента; вендоры тоже выпустили аналогичные инструменты вскоре после того, как мы объявили о ChAP.

Что интересно, спустя несколько месяцев, когда наши четыре бэкенд-разработчика ничего не делали, а только занимались программированием, я с удовольствием отметила, что наша команда приобрела уникальные знания о системе и ее режимах отказа. Потратив время на разработку архитектуры безопасных отказов в системе Netflix, вы, естественно, начинаете хорошо понимать, как все это работает в целом, не обладая «глубокими» знаниями в какой-либо конкретной части системы.

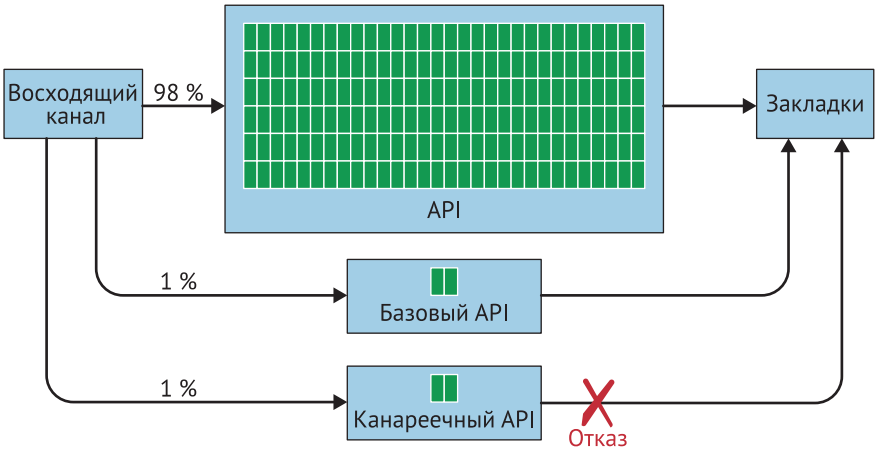

Рис. 9.1 ❖ Эксперимент ChAP внедряет отказ в закладки из API[1]

Наш опыт пригодился не только для создания и разработки экспериментов по хаосу, но и для обучения команд приемам наиболее эффективного использования инструмента; мы показывали, как им пользоваться, и задавали наводящие вопросы, заставляющие задуматься о том, как их службы могут испытать сбой. Хотя это было здорово и несомненно необходимо, вскоре я заметила, что большинство команд не склонны проводить эксперименты по собственному желанию. Они обычно запускали их, когда (а) мы просили или (б) перед крупным событием (например, праздничным сезоном в Netflix). Я начала задаваться вопросом: «А проводят ли члены сообщества Chaos Team большинство этих экспериментов? Что им не нравится?»

В состав пользовательского интерфейса ChAP входит страница «Запуски», где любой пользователь может просмотреть предыдущие эксперименты и узнать, когда они запускались, как долго, что не получилось и, что наиболее важно для моих целей, *кто* их запускал. Из любопытства я начала анализировать, кто проводил эксперименты. Что же я увидела? Прежде всего я поняла, что одно из моих предположений оказалось правильным: в основном только мы, хаос-инженеры, проводили эти эксперименты. Но хорошо ли это? В конце концов, экосистема Netflix была огромной; как мы можем знать все направления для экспериментов?

Мы так гордились удивительным инструментом, который разработали, – мы могли бы безопасно устроить отказ, добавляя задержку к вызовам служб в производстве, – но какой прок от этого инструмента, если единственными людьми, которые его используют, остаются четыре его создателя? Кроме того, хоть мы и были хорошими специалистами, но не могли получить экспертные знания в каждой области экосистемы Netflix, а значит, не могли индивидуально разрабатывать масштабные эксперименты, охватывающие всю систему.

---

[1] *Ali Basiri et al.* Automating Chaos Experiments in Production // International Conference on Software Engineering (ICSE), 2019, https://arxiv.org/abs/1905.04648.

Когда мы попытались уговорить людей проводить эксперименты, то обнаружили, что они переживают из-за возможных проблем с производством. В ответ на опасения мы восклицали «Но это так безопасно!»; однако надо признать, что эти люди находились в другой ситуации. Чтобы они чувствовали себя комфортно, используя инструмент, мы решили выделить время на разработку и запуск совместного «безопасного» эксперимента.

Это была довольно нелегкая работа в довесок к полному рабочему дню, занятому программированием, – ведь на работе от нас ожидали, что большую часть времени мы будем программировать. Мы поступили, как обычно делают настоящие программисты: автоматизировали процесс! Если уж нам пришлось разработать эксперименты для разных команд, то имеет смысл запускать их *автоматически*.

Чтобы сделать это, мне пришлось выработать совершенно новое понимание различных частей экосистемы, позволяющее разработать алгоритмы экспериментов, нацеленных на разные компоненты системы. Как я этого добилась? Мне пришлось буквально по крупицам собирать фрагменты информации о том, что значит безопасность хаос-эксперимента для каждой конкретной службы, как расставить приоритеты для экспериментов и где найти необходимую информацию для их разработки. Я вместе со своей командой опросила около 30 человек по всей организации и изучила все, *что* они знают о своих системах и *как* они это представляют. Как только у нас сформировалось понимание принципов работы разных частей системы, мы перешли к автоматическому сбору рабочей информации. Было бы совершенно неприемлемо использовать в экосистеме Netflix несколько разнотипных алгоритмов, поэтому мы разработали некий унифицированный алгоритм, охватывающий разные службы. Основываясь на результатах консультаций с экспертами Netflix, мы определили, что для разработки безопасного эксперимента, нацеленного на определенную службу, требуется *как минимум* следующая информация:

- длительность тайм-аутов;
- интервал повторных вызовов;
- назначен ли этим вызовам резерв;
- процент трафика, который обычно обрабатывает служба.

Это подводит нас к следующему пункту – многим командам, разрабатывающим хаос-эксперименты, нужно пойти и поговорить со своими пользователями. В этом может помочь менеджер по продукту или технический директор, но в конечном итоге хаос-инженеры должны обратиться к пользователям и сделать то же самое. Как я уже говорила, лучший способ собрать нужную информацию – это установить партнерские отношения с техническим персоналом служб и понять их точку зрения и соответствующие ментальные модели. Итак, давайте поговорим о том, как сделать это наиболее эффективно.

9.4. Эффективное внутреннее партнерство

В расследовании инцидентов в других отраслях (авиация, медицина, морское судоходство) обычно принимает участие третье лицо, *фасилитатор*, который

опрашивает членов комиссии и собирает сведения о том, что, по мнению товарищей по команде, произошло во время инцидента и привело к нему. Он не стремится получить одну каноническую историю; наоборот, профессиональные фасилитаторы отмечают, что очень важно собирать и фиксировать все мнения, потому что пробелы и нестыковки в ментальных моделях разных людей показывают, где кроется настоящая проблема. Аналогичным способом мы собираем информацию о системе и ее потенциально уязвимых местах.

Хотя это может быть трудоемким занятием, важно (по крайней мере, на этапе проектирования экспериментов) собрать ментальные модели разных членов рабочей группы и выработать общую гипотезу. Причем гипотеза не должна исходить от фасилитатора! Он должен лишь провести собеседование с группой в полном составе (примерно от 30 минут до часа) и дать каждому участнику возможность рассказать о своем понимании системы и о том, что происходит в случае сбоя.

Почему так важна совместная работа? Вы получаете *наибольшую* отдачу от хаос-инжиниринга, когда несколько человек из разных команд, участвующих в эксперименте, вместе обсуждают свои ожидания от системы. Обладание полной картиной происходящего в системе на этапе проектирования эксперимента столь же важно (если не важнее), чем выполнение эксперимента.

Дальше я расскажу про некоторые вопросы, которые можно задать участникам группы в качестве фасилитатора, чтобы извлечь максимальную пользу из хаос-эксперимента. Каждый вопрос служит инструментом для получения ментальной модели эксперта и заставляет членов рабочей группы противопоставить различные точки зрения.

9.4.1. Организация рабочих процедур

Фасилитатор должен направлять участников группы, задавая ряд вопросов. Начните с вопроса «Какое поведение и взаимодействие между компонентами вы ожидаете от системы?». Если позволяют корпоративные нормы и принятый у вас стиль общения, этот вопрос сначала нужно задать члену группы, не состоящему в штате компании, или младшему сотруднику, а потом лицу с более постоянным или высоким статусом (или большему количеству экспертов в предметной области). Затем каждый человек в комнате должен внести свой вклад, в конечном счете повышая уровень экспертизы. Здесь стоит упомянуть *закон компетентности Дэвида Вудса*, который гласит, что когда эксперты приобретают достаточный опыт в своей предметной области, они теряют способность оценивать свои навыки как вещь[1]. Для профессионала слишком многое выглядит очевидным; опыт скрывает усилия, необходимые для достижения результата. Ваша роль как фасилитатора, направляющего процесс, состоит в том, чтобы раскрыть эти усилия и пояснить каждому участнику, в чем они заключаются.

---

[1] *Robert R. Hoffman and David R. Woods.* Beyond Simon's Slice: Five Fundamental Trade-Offs That Bound the Performance of Macrocognitive Work Systems // IEEE Intelligent Systems (Nov./Dec. 2011).

Еще одна причина такого подхода заключается в том, что новые сотрудники смотрят на систему свежим взглядом и безразличны к ее прошлому. Когнитивный психолог Гари Кляйн подробно изучил различия между ментальными моделями новых и более опытных сотрудников:

> Новому сотруднику бросаются в глаза эпизоды, когда система *ломалась*, а не время, когда она работала исправно[1].

Фасилитатор должен сформировать свою ментальную модель системы на основе описания, которое предоставляют члены команды.

Фасилитатор также должен отметить, как оценки поведения системы и взаимодействия компонентов, которыми обеспокоены люди, варьируются среди товарищей по группе. Выделите общий опыт, а также расхождения и относитесь к этой части упражнения как к обучающему упражнению. Члены группы наверняка будут иметь разные ответы на этот вопрос.

Затем фасилитатор должен спросить: «Как вы думаете, что ваши коллеги знают о системе?» Этот вопрос задают, чтобы выявить отсутствие согласованной позиции по следующим пунктам:

- есть ли какие-либо нижестоящие службы, о работе которых вы беспокоитесь?
- что происходит с вышестоящими службами, если ваши службы работают неправильно?
- что произойдет, если неполадки с вашей службой случатся прямо сейчас? У вас есть запасные варианты (резерв или что-то еще)? Что делают эти запасные варианты? Как запасные варианты меняют поведение службы? Как запасные варианты могут повлиять на пользовательский опыт?
- есть ли недавние изменения в вашей системе, либо в вашей службе, либо в ваших восходящих/нисходящих каналах, которые могут привести к появлению новых уязвимостей в системе? Кто лучше всех знает об этих изменениях?
- может ли ухудшиться состояние вашей службы? Что может привести к этому?
- насколько вы разбираетесь в настройках различных параметров конфигурации вашей системы? То есть понимаете ли вы, на что влияют настройки тайм-аутов, интервалы повторных попыток и другие жестко заданные значения в вашей системе?
- что вас больше всего пугает в повседневной работе системы (в самых разных смыслах)?

Отсутствие уверенности у людей является признаком не до конца сформированной ментальной модели или, возможно, отсутствия согласованности рабочих процедур.

Только после того, как все участники команды эксперимента придут к единому мнению по этим вопросам, вы можете переходить к обсуждению пред-

---

[1] *Gary Klein and Robert Hoffman.* Seeing the Invisible: Perceptual-Cognitive Aspects of Expertise // *M. Rabinowitz (ed.).* Cognitive Science Foundations of Instruction. Mahwah, NJ: Lawrence Erlbaum Associates, 1993, p. 203–226.

мета эксперимента. Обратите внимание, что после обсуждения ответов на вопросы довольно часто принимают разумное решение не переходить к следующему этапу.

9.4.2. Обсуждение предмета эксперимента

Если вы решили проводить эксперимент, важно обсудить его свойства и взаимно согласовать ментальные модели каждого участника, задавая следующие вопросы:

- *как* вы определяете «нормальную» или «хорошую» работу службы?
- *какой* масштаб эксперимента мы установим (на каком уровне введем сбой)?
  - каковы причины для выбора именно этого масштаба?
  - все ли члены команды понимают ваши рассуждения – и откуда вы это знаете?
  - обусловлены ли ваши действия страхом? (Проводите ли вы этот эксперимент, потому что инцидент уже случался в прошлом? Если это так, изучите «посмертные» журналы и протоколы инцидента и свяжите их с сегодняшним обсуждением.)
  - вызван ли этот эксперимент отсутствием понимания того, что может произойти, если X потерпит сбой? (то есть случался ли этот сбой редко, если вообще когда-либо случался, и вы просто не уверены, что произойдет?)
- *что* вообще будет происходить в случае сбоя (формулируйте однозначно)?
  - чего мы ожидаем от отдельных компонентов?
  - чего мы ожидаем от системы в целом?
- *как* вы узнаете, что система испытывает проблемы?
  - какие показатели наиболее важны для вас и заслуживают измерения во время эксперимента? Задавая себе эти вопросы, вспоминайте следующую цитату из книги Холлнагеля и Вудса[1]: «*Наблюдаемость – это обратная связь, которая дает представление о процессе* и относится к работе, необходимой для извлечения смысла из имеющихся данных»;
  - учитывая это определение, как вы *наблюдаете* за системой?
- *как* вы ограничиваете радиус поражения?
- *какова* предполагаемая ценность эксперимента для бизнеса (для команды по эксплуатации)? Какова осязаемая ценность для остальной части организации?

Ответы на эти вопросы могут быть разными у разных людей, и это нормально.

---

[1] *Erik Hollnagel and David D. Woods.* Joint Cognitive Systems: Foundations of Cognitive Systems Engineering. Boca Raton, FL: Taylor & Francis, 2005.

9.4.3. Построение гипотезы

Теперь пришло время, чтобы сформулировать и переосмыслить вашу коллективную гипотезу об эксперименте:

> Если случится отказ в части X системы, то произойдет Y, и влияние будет Z.

- Запишите, что такое «устойчивое состояние» в контексте данной системы.
- Каково предполагаемое отклонение от устойчивого состояния? (Дайте определение для всех показателей, которые вы решили отслеживать.)
- Запишите, чего вы ожидаете (если чего-нибудь ожидаете) от пользовательского опыта.
- Запишите, какого воздействия вы ожидаете на нисходящий канал (если чего-нибудь ожидаете).
- Наконец, поделитесь документацией со всеми заинтересованными сторонами.

Распределение ролей

Если вы проводите эксперимент по типу игрового дня, определите участников и их роли в эксперименте. Имейте в виду, что не все эти роли необходимы для автоматизированных экспериментов, но в целом они очень полезны, если у вас достаточно времени и ресурсов для их временного укомплектования персоналом. Пока у вас нет автоматизации для ограничения радиуса поражения и внедрения отказа, каждая команда эксперимента должна иметь следующие роли:

- руководитель/фасилитатор (человек, ведущий дискуссию);
- исполнитель (человек, выполняющий команды);
- регистратор (делает записи о том, что происходит в ходе эксперимента в инструменте связи, таком как Slack);
- наблюдатель (следит за показателями системы и делится соответствующими графиками с остальной частью группы);
- корреспондент (следит за каналом #alerts-type-channel и делает так, чтобы дежурная смена техподдержки знала о проводимом эксперименте и ожидаемых последствиях).

Перечислите всех дополнительных людей, которые присутствуют при разработке и проведении эксперимента, а также их роль в эксперименте.

После того как на все эти вопросы даны ответы и роли распределены, разработчик/фасилитатор должен поделиться своим пониманием эксперимента и того, что обсуждалось и изучалось на этой встрече, с остальной частью группы в удобном для использования формате, в котором четко оговорены методы, которые следует использовать, и области, на которые следует обратить внимание.

Давайте кратко подытожим, о чем вы научились спрашивать и думать на этих этапах:

Перед экспериментом

Как вы узнаете о расхождениях в понимании и предположениях об устойчивом состоянии среди партнеров по команде, задавая вопросы?

Почему возникают эти расхождения, и что они означают? Как вы определяете «нормальную» или «хорошую» работу системы?

Какова осязаемая ценность экспериментов на этой части системы?

Как вы поощряете людей строить свои ментальные модели структурированным способом – посредством визуального представления или выработкой структурированной гипотезы о том, как, по их мнению, должна работать система?

Чтобы перейти к следующему этапу, проведению эксперимента, вы можете автоматизировать некоторые действия – как вы решаете, какими они должны быть? Как вы определяете, в каком масштабе вы можете безопасно проводить свои эксперименты? Для чего следует и для чего не следует использовать измерение эффективности платформы для проведения экспериментов на устойчивость? Как отделить сигнал от шума и определить, является ли ошибка результатом хаос-эксперимента или чем-то еще?

После эксперимента

Если в ходе эксперимента обнаружена проблема, вы можете задать следующие вопросы:

- что вы изучали?
- как вы используете полученную информацию для изменения структуры эксперимента и его повторения?
- как члены команды достигли согласия о проведении эксперимента до его запуска?
- как члены команды общались друг с другом о том, что происходило во время эксперимента?
- нужно ли пригласить в команду кого-либо, кто не присутствовал при разработке или проведении эксперимента, или расспросить его заранее, потому что у него есть полезный опыт?
- случились ли какие-либо инциденты во время эксперимента (как связанные, так и не связанные с экспериментом)?
- как вы определили, *что* стало основным результатом эксперимента?
- какие части эксперимента вели себя не так, как ожидалось?
- если исходить из того, что вы узнали и чего вы до сих пор не понимаете, какие эксперименты вы должны провести в следующем цикле?
- как распространить полученные знания таким образом, чтобы каждый в организации мог их увидеть и прочитать?
- на каком основании вы думаете, что последующие эксперименты будут приносить новые знания? И как связать эти знания с инцидентами и другими неожиданностями, которые переживает наша организация?

9.5. Вывод

Перечисленные вопросы являются инструментами, которые фасилитатор может использовать для ведения дискуссии. Однако цель подобных дискуссий – помочь команде экспертов обнаружить скрытые знания о компонентах системы и поделиться ими с окружающими. Помните: вы как фасилитатор не являетесь и не должны быть экспертом в этой ситуации – вы здесь для того, чтобы раскрыть чужой экспертный опыт, а не показать свой. Вся эта работа по выявлению опыта сводится к разработке лучших гипотез и вытекающих из них экспериментов, результаты которых могут вас удивить.

Мы ищем сюрпризы.

Давайте вернемся к истории Netflix, описанной ранее в этой главе. Все ключевые принципы, связанные с этапами экспериментов «до» и «после», основаны на опыте *автоматизации экспериментов на производстве и снятия «бремени» с пользователя*. Хотя нам удалось разработать алгоритм, который создавал, расставлял по приоритетам и выполнял хаос-эксперименты, самая большая ценность, которую мы извлекли из этого, заключается в том, что мы помогли людям усовершенствовать и отточить ментальные модели своих систем.

Собрав необходимую информацию для автоматического проектирования экспериментов, мы тем самым создали новый опыт и захотели поделиться им с миром. Всем, что мы узнали в процессе разработки автоматизированной платформы, мы поделились на динамической информационной панели. Конечно, это был побочный эффект от проекта по автоматизации экспериментов; разработка информационной панели не входила в число наших приоритетных задач. Тем не менее эта панель отражает наше глубокое понимание системы: она показывает пользователям, считает ли наш алгоритм, что их сервис защищен от сбоя, еще до того, как мы провели на нем эксперимент. Мне очень понравилось развлекаться с этой панелью – запускать ее перед разными командами и показывать, какие части их системы защищены от сбоя. Что в результате? Каждый раз, когда я показывала эту панель кому-то из специалистов компании, неизменной реакцией было удивление. Они всегда открывали для себя что-то новое, например что один из их сервисов «неожиданно» оказался критическим, или у них был слишком большой тайм-аут, или их повторные запросы не имели логического смысла. Я надолго запомню это удивленное выражение на лицах коллег каждый раз, когда показывала им информационную панель: я видела, как они старательно уклонялись от того, что мы на самом деле пытались делать все время, чтобы лучше понять *их* систему. И хотя мы запустили автоматизацию экспериментов, и она отлично работала, мы в конечном итоге отключили ее после нескольких недель работы. Почему? Что ж, по правде говоря, мы получили основную пользу от информационной панели и *процесса* автоматизации этих экспериментов, а не самой автоматизации, а одобрение нашей концепции в компании взлетело до небес, но не так, как мы изначально планировали. Попробуй тут угадать заранее…

Весь этот опыт побудил меня осознать, что совместная работа, когнитивное согласование и распространение опыта не только очень важны для успе-

ха хаос-инжиниринга в организациях, но и хронически обделены вниманием. Автоматизация упомянутых здесь показателей привела к уменьшению глубины обучения других команд и еще больше увеличила глубину обучения нашей команды. Только после того, как мы начали делиться своими методами разработки автоматизации и результатами, мы достигли того, что искали: более глубокого совместного понимания людьми принципов работы системы и причин сбоев. Команды, в конце концов, лучше всего учатся на примерах упрощенных экспериментов и вытекающего из них сравнения собственных ментальных моделей.

Глава 10

Гуманистический хаос

Автор главы: **Энди Флинер**

Уже довольно давно меня беспокоит один вопрос: как я могу применить свои знания о хаос-инжиниринге к человеческим системам? Впервые услышав о новой отрасли с необычным названием, я был, мягко говоря, очень заинтригован. Целенаправленно вводить в систему ошибки и отказы, чтобы лучше в ней разобраться? Я был покорен с первого слова. Как фанат «нового взгляда» на безопасность и системный мыслитель я придерживаюсь парадигмы, что системы, которые мы используем каждый день, по своей сути небезопасны. Мое исходное предположение заключалось в том, что если методы хаос-инжиниринга были разработаны для работы с распределенными веб-системами, то их можно применять и к другим распределенным системам, с которыми мы ежедневно взаимодействуем, – к системам, которые окружают нас, к системам, которые формируют нашу повседневную жизнь.

Что, если применить подходы хаос-инжиниринга не только к сложным техническим системам, которые мы знаем и любим, но и к сложным организациям? Организация – это одна гигантская система систем, так почему бы не применять к ней те же правила? В этой главе я расскажу о трех реальных примерах использования принципов хаос-инжиниринга в команде Platform Operations, которую я возглавляю, а также в большой организации по разработке продуктов SportsEngine, и надеюсь дать вам инструменты для применения этих же методов в вашей собственной организации.

10.1. Люди в системе

В организации фундаментальной единицей или действующим лицом системы является человек. По-настоящему сложные проблемы заключаются во взаимодействии между этими людьми. И даже когда мы говорим о программировании, в общем виде проблема звучит примерно так: «Нужно создать программное обеспечение, которое люди могут эффективно использовать для достижения цели». Обратите внимание, что люди занимают центральное место как в проблеме, так и в ее решении. Они одновременно являются причиной и решением этой проблемы.

10.1.1. Значение человека в социотехнических системах

Поскольку у нас инженерный склад ума, нас привлекают технические решения для человеческих проблем. Технология оказала огромное влияние на нашу жизнь. Мы этим живем. Истории трансформации «от монолита к микросервисам» встречаются на каждом шагу, и многие из этих превращений сосредоточены вокруг решения очень сложных проблем масштабирования организации. Но изменение технологии, которую мы используем, не будет волшебным образом создавать культуру, которую мы желаем. Мы постоянно игнорируем социальную сторону социотехнических систем, которые внедряем в повседневную жизнь. Понятно, что наиболее эффективным способом улучшения технических систем является их схематизация. Гэри Кляйн рассказывает о том, как мы получаем эти знания, в своей книге «Чего не заметили другие» («Seeing What Others Don't», в русском переводе пока не издана). Он утверждает, что истории являются «опорами» внутри организации, и эти истории – это то, как мы интерпретируем детали. Мы делаем это несколькими способами, такими как обзоры архитектуры, системные проекты и документация, и на более высоком уровне с ретроспективными показателями, такими как обзоры происшествий. Мы постоянно пытаемся скорректировать нашу ментальную модель того, как функционирует система. Но как мы это делаем с социальной частью системы? Когда вы в последний раз схематизировали канал эскалации инцидентов? Включает ли эта схема план действий на случай, когда человек 2-го уровня техподдержки находится в отпуске? Понятно ли из схемы, что делать, когда вы не получаете подтверждения того, что кто-то работает над проблемой? Содержит ли ваша схема точку принятия решения о том, является ли событие инцидентом? Допустим, все это у вас предусмотрено. Это великолепно, только вы когда-нибудь пытались проверить эти штуки в действии? Не забывайте о пропасти между «работой как хотелось» и «работой как получилось». То, что мы говорим, и то, что мы на самом деле делаем, – это совершенно разные вещи. Влияние этого разрыва можно описать как «скрытый долг», термин, впервые использованный в отчете команды STELLA в 2017 году[1].

Компоненты или участников системы можно рассматривать как единицы, обладающие ограниченной нагрузочной емкостью; последствия исчерпания этой мощности могут быть крайне неопределенными. В технической части системы это могут быть, например, каскадные сбои или другие неисправности. Когда мы говорим о социальной стороне системы, то часто подразумеваем под перегрузкой выгорание или разочарование в выборе жизненного пути. Но как можно измерить и описать воздействие на систему выгоревшего инженера или разочарованного менеджера, если невозможно определить, когда и как это происходит?

---

[1] SNAFUcatchers. Dark Debt // STELLA: Report from the SNAFUcatchers Workshop on Coping With Complexity, March 14–16, 2017, https://oreil.ly/D34nE.

10.1.2. Организация – это система систем

Что мне очень нравится в работе с организациями, так это то, что в них везде есть системы. Некоторые из этих систем формализованы, например у вас может быть политика отпусков, которая позволяет использовать *x* выходных дней в году. Или у вас есть недельный график дежурств, который расписан между шестью сотрудниками. Но другие системы вообще нигде не отражены, это просто общеизвестные неписаные правила, как общение с Джорджем только через Slack, потому что он никогда не проверяет свою электронную почту. Или Сьюзи знает больше всех о конкретной службе, потому что она написала большую часть ее кода. Системы неписаных знаний, как правило, менее надежны, чем явно прописанные аналоги. Но ничто так не делает систему ненадежной, как ошибочные представления. Всегда будет разрыв между работой, как вы себе ее представляете, и работой, которую вы реально делаете[1]. Способ создания действительно надежных систем – это активная работа по сокращению данного разрыва.

Поскольку организации представляют собой сложные системы, они соответствуют утверждениям доктора Ричарда Кука о том, как сложные системы выходят из строя[2]. Короче говоря, они опасны и содержат множество скрытых отказов за каждым углом; они хорошо защищены от отказов (катастрофа возможна только через череду отказов); и люди играют двойную роль как защитники системы, так и источники отказов. Каждое действие, предпринимаемое в сложной системе, является азартной игрой, но именно эти азартные игры создают безопасность в системе.

10.2. Инженерно-адаптивный потенциал

В статье доктора Сидни Деккера о человеческой деятельности[3] представлены два взгляда на безопасность. Старый подход «рассматривает людей как проблему для контроля (через процедуры, соблюдение, стандартизацию, санкции). Безопасность измеряется в основном отсутствием негативных явлений». Новый подход стремится «воспринимать людей как решение проблемы, а не как проблему, требующую решения, и рассматривать безопасность больше как наличие положительных способностей». Вооружившись именно этим новым подходом, я сосредоточился на создании надежной, продвинутой и, как мы надеемся, устойчивой команды инженеров. Как руководитель команды, которая занимается техническим обслуживанием по вызову, я вижу свою задачу в поиске сигналов, говорящих о том, что команда

---

1 *Steven Shorrock.* The Varieties and Archetypes of Human Work. Safety Synthesis: The Repository for Safety-II (website), https://oreil.ly/6ECXu.

2 *Richard I. Cook.* How Complex Systems Fail // Cognitive Technologies Laboratory, 2000, https://oreil.ly/kvlZ8.

3 *Sidney Dekker.* Employees: A Problem to Control or Solution to Harness? // Professional Safety Vol. 59, № 8 (Aug. 2014).

достигла границы возможностей, и одновременно ищу способы отодвинуть эту границу. Что я могу сделать как участник системы, чтобы уберечь систему от настоящей проблемы, такой как увольнение важного специалиста, потеря большого числа клиентов или утечка данных?

10.2.1. Обнаружение слабых сигналов

Необходимость постоянно поддерживать дистанцию между нормальной «безопасной» работой и серьезным отказом подтолкнула меня к поиску *слабых сигналов*. Слабые сигналы – это небольшие, едва заметные признаки нового тревожного поведения системы. В мире веб-систем мы используем такие методы, как USE, который представляет использование (utilization), степень насыщения (saturation) и ошибки (errors). Это мониторы, установленные на ключевых узких местах системы. Они могут не указывать на непосредственные проблемы, но как только значения пересекают заданную границу, система начинает выходить из строя (сильные сигналы). В контексте организации доктор Тодд Конклин описывает сигналы в своем подкасте[1] «Предаварийные наблюдения» как слабые индикаторы, которые сообщают нам, когда проблема зародилась, а не когда она уже произошла: «Вы никогда не услышите слабый сигнал при отказе; у отказа очень громкий сигнал». Он приводит несколько примеров, таких как «дверь, которая открывается не в ту сторону» или «светофор, который переключается слишком быстро, чтобы успеть безопасно перейти улицу». Организации настолько высоконадежны, что слабые сигналы часто являются единственным типом сигнала, который вы можете использовать для увеличения устойчивости системы.

Давайте рассмотрим некоторые примеры, с которыми я столкнулся лично. Я заметил, что окончание дежурной смены в понедельник более утомительно, чем окончание в пятницу. Само по себе это не проблема, пока никак себя не проявляет. Но если инженеры устали, они будут менее эффективны и могут допустить серьезную ошибку. В то время мы также использовали общедоступный канал Slack, чтобы принимать заявки и отвечать на вопросы других команд. В основном это работает хорошо, но если вся команда дружно отвечает на вопрос, когда нужен ответ только одного человека, или, что еще хуже, никто не отвечает, это влияет на ход работы моей команды и других команд. Другой очень распространенный слабый сигнал может выглядеть так: «Я ничего об этом не знаю, нам нужно поговорить с Эммой». Этот сигнал «нам нужно поговорить с *X*» встречается постоянно; он означает, что информация не распределена между всеми членами команды. Но это сигнализирует, что система приближается к границе устойчивости. Что произойдет, если Эмма внезапно решит уволиться? Ваша система постоянно о чем-то сигнализирует. Нахождение баланса между тем, на какие сигналы реагировать, а за какими продолжать следить, – это действительно сложная задача.

---

[1] *Todd Conklin.* Safety Moment: Weak Signals Matter // Pre-Accident Investigation (podcast), July 8, 2011, https://oreil.ly/6rBbw.

10.2.2. Неудача и успех, две стороны одной монеты

Для меня откровением в поиске слабых сигналов стало то, что успех и неудача не являются выбором по нашей воле. Как описывает доктор Йенс Расмуссен в своей динамической модели безопасности, успех или неудача зависит от того, когда рабочая точка системы пересекает границу, переломный момент неудачи в работе. Расмуссен называет это «переломным усилием»[1]. Стабильно работающая система может внезапно и неожиданно погрузиться в состояние отказа. Если мы намеренно ищем состояния отказа наших организационных систем, мы лучше знаем эту границу. Это либо сделает границу более явной, либо даже улучшит систему и отодвинет границу еще дальше. Что я заметил у себя в SportsEngine, так это то, что с помощью принципов хаос-инжиниринга возможно и то, и другое: мы можем сделать границу более четкой и отодвинуть ее дальше.

10.3. Применение принципов хаос-инжиниринга на практике

Много сказано о применении принципов хаос-инжиниринга для больших распределенных систем. Есть замечательные инструменты, разработанные ребятами из Netflix, ChaosIQ и других компаний, но эти инструменты предназначены для технических систем. Что же нам делать с хаосом в социальных системах?

Что интересно в социальных системах, так это то, что ваша позиция в системе сильно влияет на вашу способность изменить ее. Как участник системы вы получаете обратную связь таким путем, который невозможен в зависимых системах[2], таких как программные продукты, которые мы создаем и используем. Если вы хотите увидеть механизм социальной обратной связи в действии, попробуйте просто переставить рабочие столы по всему отделу. Если у вас *хоть самая малость* пойдет не так, вам непременно скажут.

Как и любые эксперименты с хаосом, социальный хаос-инжиниринг должен основываться на основных принципах. В этом тематическом исследовании я остановлюсь на трех основных принципах. Сначала построим гипотезу, основанную на установившемся состоянии системы. Вам необходимо определить устойчивое состояние системы, прежде чем вы сможете контролировать результат. Затем определите переменную, которую вы планируете намеренно изменить. Это этап внедрения отказа. И наконец, следите за

---

[1] *R. Cook and Jens Rasmussen.* 'Going Solid': A Model of System Dynamics and Consequences for Patient Safety. BMJ Quality and Safety Healthcare, 14 (2005).

[2] SNAFUcatchers. The Above-the-Line/Below-the-Line Framework // STELLA: Report from the SNAFUcatchers Workshop on Coping with Complexity, March 14–16, 2017, https://oreil.ly/V7M98.

результатом. Найдите новое тревожное состояние системы и сравните его с предыдущим состоянием.

10.3.1. Построение гипотезы

Как мы знаем, первым шагом к проведению эксперимента хаоса является построение гипотезы об устойчивом поведении. В классическом смысле речь идет об основных показателях уровня обслуживания, таких как пропускная способность, частота ошибок, задержка и т. д. В социальной системе наше понимание устойчивого состояния сильно отличается. Метрики ввода/вывода не столь очевидны, петли обратной связи могут быть короче, а видимость системы кардинально отличается от картины, которую рисует нам мониторинг технических систем. Это ситуация, когда качественный анализ может быть гораздо более эффективным. Люди – это машины обратной связи; это то, что мы делаем каждый день. Мы находимся в постоянном состоянии обратной связи. Если вы собираетесь провести эксперимент, очень важно создать эффективные петли обратной связи в рамках эксперимента. Это не должно вас удивлять, поскольку петли обратной связи лежат в основе того, как вы создаете успешные изменения в организации.

Выдвигая первую гипотезу, посмотрите на свои организационные ограничения. Где у вас есть конкретные точки отказа, узкие места в коммуникации, длинные очереди рабочих заданий? Определите места системы, в которых, как говорит ваша интуиция, вы наиболее близки к границе пропускной способности, поскольку именно здесь вы получите наибольшую отдачу от эксперимента.

10.3.2. Варьирование событий реального мира

После того как вы определили предел возможностей системы, следующим шагом будет попытка оттолкнуться от края. Вы, вероятно, столкнетесь с тремя вариантами сценария.

Сценарий 1

Вы вносите изменения в систему, и она немедленно рушится под тяжестью изменений. Не забудьте заранее разработать план отката. Вы можете не многому научиться из этого сценария, но поздравляю, вы нашли жесткое ограничение! Подобные ограничения должны быть явными, и ценность эксперимента в том и состоит, что вы теперь сделали это явным. Примером такого сценария может быть изменение, в результате которого важная работа остается без исполнителя – работа, которая слишком важна, чтобы ее потерять.

Сценарий 2

Вы внесли изменение, и теперь предел, который вы отслеживали, изменился, но влияние на нагрузочную способность системы слишком велико, чтобы поддерживать это изменение в течение неопределенного времени. Эта обратная связь может оказать влияние на понимание вашей выгоды и того,

какие новые свойства система приобретет в этом состоянии. Возникающее поведение иногда помогает распознать слабые сигналы и дать опережающие индикаторы на будущее. Это явление больше относится к конкуренции приоритетов и выражено намного тоньше, чем работа, оставшаяся без исполнителя. Допустим, вы переместили сотрудников из одной команды в другую, и это работает какое-то время, но когда возрастет нагрузка на первую команду, вам, возможно, придется вернуть их обратно.

Сценарий 3

Вы вносите изменения в систему, и все указывает на то, что она не испытала существенного влияния. Это самый сложный сценарий. Скорее всего, это означает, что ваша гипотеза была полностью неверной, или вы измеряете не то, что нужно, и у вас нет правильной обратной связи, чтобы понять влияние. В этой ситуации, как правило, вы должны вернуться к чертежной доске. Как и в случае неудачного научного эксперимента, вам может не хватать ключевой информации об устойчивом состоянии системы. В качестве примера представьте, что вы нашли единственную точку отказа в системе – конкретного человека. Но, несмотря на ваши попытки заставить его поделиться своими знаниями, он по-прежнему остается единственным и незаменимым экспертом в данной области.

10.3.3. Минимизация радиуса поражения

Как и в случае технических систем, каскадные сбои являются самым большим риском при проведении хаос-экспериментов. Постоянно имейте в виду, какое влияние может оказать конкретный эксперимент. Например, внесение радикальных изменений, ограничивающих возможности команды, на которую полагаются многие другие команды, может привести к нарастающему каскаду проблем в организации и усложнить будущие эксперименты. Этот уровень воздействия, скорее всего, приведет к сценарию № 1, и вам быстро придется отменить все внесенные изменения.

Вы быстро узнаете, как функционирует система и где рабочее поле эксперимента меньше, чем вы ожидали. Но прежде чем снова попытаться провести этот эксперимент, вам определенно нужно поработать над расширением границ рабочего поля. Влияние радиуса поражения сильно зависит от контекста вашего бизнеса. Например, бизнес SportsEngine имеет очень сезонный характер. Попытка ввести значительные изменения, влияющие на доступность наших продуктов для клиентов в течение года, очень заметно влияет на наш бизнес. Но если мы сможем уменьшить охват эксперимента, затрагивая только продукты, для которых в настоящее время не сезон, это не повлияет на наших клиентов или нашу прибыль.

Этап планирования эксперимента должен включать разработку планов действий в чрезвычайных ситуациях. Очевидно, что конкретные планы будут зависеть от характера эксперимента, но «откат», аварийный люк, «выдерни шнур, выдави стекло» – как бы вы это ни называли – необходимо сформулировать предельно понятно. Каждый участник эксперимента должен обладать способностью и полномочиями справиться с аварийным планом.

Игра на этом уровне, как и при экспериментах с техническими системами, означает принятие правильных деловых решений. Не нарушайте систему только для того, чтобы доказать свою точку зрения. Вам необходимо внести свой вклад в общее дело вашей организации, а это означает, что нужно четко осознавать полезность того, что вы делаете.

Вы также должны помнить, что все действующие лица в этой системе – живые люди. У всех них есть свои собственные перспективы, жизненный опыт и цели. Это относится и к вам, человеку, управляющему экспериментом. Ваши личные предубеждения могут оказать существенное влияние на эксперимент, на людей в эксперименте и на бизнес. Задавайте вопросы всем, кого встретите, слушайте отзывы и будьте готовы изменить свои взгляды или даже отказаться от эксперимента. Не позволяйте вашему желанию увидеть успех эксперимента перекрыть отрицательные отзывы.

10.3.4. Пример 1: игровые дни

Давайте пока оставим в стороне культуру и системное мышление и поищем новые возможности системы. Исходя из своего увлечения «новым взглядом» на безопасность, я пришел к убеждению, что инциденты – это незапланированные обучающие события в моей организации[1]. Это осознание вдохновило меня на создание игровых дней, которые полностью имитировали наш жизненный цикл реакции на инциденты. Это возможность бросить какого-нибудь новичка в водоворот работы организации и дать ему возможность понять, каково это – быть частью системы в безопасной среде с минимальными рисками. В этой среде нет прямого ущерба клиентам или бизнесу, если не считать потраченное время. Если вы еще не проводите игровые дни в своей организации, займитесь этим.

Игровые дни в SportsEngine состоят из четырех этапов.

1. *Распределение и планирование*: вы выбираете день в календаре или часть дня и формируете разнообразную группу людей, которые будут вашими «игроками».
2. *Выполнение игры*: это мой любимый этап. Ваша задача как участника игры состоит в том, чтобы целенаправленно внедрить отказ в систему, а затем отойти в сторону и посмотреть, как он распространяется по вашей системе.
3. *Реакция на инцидент*: как участник игры вы должны внимательно следить за происходящим; у вас есть преимущество в знании начального события. Внимательное наблюдение за реакцией может дать вам бездну понимания пробелов в ваших рабочих процессах и механизмах мониторинга и оповещения.
4. *Разбор инцидентов*: точно так же, как мы проводим разборы производственных инцидентов, мы должны проводить более неформальные, короткие обзоры после каждой игры.

---

[1] *John Allspaw.* How Your Systems Keep Running Day After Day // IT Revolution, April 30, 2018, https://oreil.ly/vHVIW.

Затем цикл повторяется: вы начинаете новую игру на этапе 2, пока не закончите запланированные игры или не закончится время.

Игровые дни оказали значительное влияние на надежность платформы SportsEngine, но они оказали еще большее влияние на нашу способность выполнять эффективный план реагирования на инциденты. Успешное реагирование на инциденты включает в себя много разнообразной работы: постановка задач экспертам в данной области, эффективное устранение неполадок, общение с заинтересованными сторонами и клиентами через информационные каналы, такие как страницы состояния. Мы рассматриваем реагирование на инциденты как командный вид спорта, а это значит, что вам нужно много тренироваться, чтобы команда работала эффективно.

Гипотеза

Если реакция на инцидент – командный вид спорта, как будет работать команда, если выбывает один из игроков? Если вы расспросите членов профессиональной спортивной команды о действиях в случае травмы звездного игрока, они дадут вам своего рода автоматический ответ «замена резервным игроком». Я выявил в нашей платформе такие места, где у нас работают один или два предметных эксперта. Что, если такой эксперт окажется единственной точкой критического отказа системы? Это может быть до боли очевидно, но чаще ситуация скорее серая, чем черно-белая. Два или три человека могут обладать общими знаниями о системе, но при этом может быть единственный человек, который знает конкретный подкомпонент. Я выдвинул очень узкую гипотезу о намеренном создании игровой ситуации, в которой не могут участвовать «правильные люди». Я предполагал, что исключение некоторых экспертов сильно повлияет на количество времени, которое потребуется, для решения учебной проблемы. Я хотел преднамеренно найти отдельные точки отказа в нашем процессе реагирования на инциденты, но при этом дать людям реальный опыт борьбы с этим отказом.

Переменный фактор

В традиционных хаос-экспериментах принято организовывать контрольные и экспериментальные группы. В этом сценарии сложно обеспечить контрольную группу, но самый эффективный способ, который я нашел, – это запустить несколько игр, как правило, с полным участием команды. Затем, начиная очередную игру, внезапно объявите: «[эксперт, предположительно представляющий единственную точку отказа] находится в отпуске». Запустите игру и сломайте подсистему, о которой лучше всего знает отсутствующий специалист. Вам также нужно, чтобы этот специалист наблюдал со стороны за всеми коммуникациями, устранением неисправностей и действиями, предпринимаемыми командой. Это даст ему совершенно новый взгляд на то, как и с кем следует делиться знаниями.

Еще один несколько менее неожиданный способ провести этот эксперимент – привлечь одного из «подозреваемых» в точке отказа к участию в игре. Это не станет неожиданностью для остальных игроков, и вплоть до запуска игры никто не сможет задавать экспертам вопросы, но это дает вам возможность использовать знания эксперта об этой системе, чтобы внедрить в нее

сбой. Это не только дает вам преимущество от участия эксперта в реакции на инцидент, но также помогает ему критически подумать о том, где находятся слабые места в его части системы.

Результат

Эта концепция определения единой точки отказа в нашей организации оказалась удачной для нас. Лучшее, что мы еще можем сделать, – это активно искать новые точки отказа. Мы пытаемся выявить системные недостатки. Эти недостатки постоянно меняются, значит, наша работа никогда не будет завершена. Мой самый наглядный критерий успеха – видеть, как люди уезжают в длительный отпуск без риска быть отозванными для устранения какого-нибудь инцидента. Вы также можете устроить простой опрос, чтобы оценить отношение дежурного персонала к механизму реагирования на заявки и уровень комфорта посменной работы. Эта культура организации работы сменного персонала – одна из вещей, которыми я больше всего горжусь в SportsEngine.

10.3.5. Коммуникации и сетевая задержка в организациях

В мире распределенных систем мы проводим бесчисленные часы, зациклившись на разработке эффективных коммуникационных протоколов и систем. Если на технической конференции вы достаточно громко произнесете слова «теорема CAP» (consistency-availability-partition, теорема Брюэра), то услышите стон и увидите, как кто-нибудь закатывает глаза, но вы также увидите таких людей, как я, которые прислушиваются к разговору. Распределенные системы постоянно находятся в состоянии одновременно как успеха, так и сбоя и функционируют благодаря согласованным алгоритмам, таким как Raft или Paxos, и надежным механизмам доставки, таким как TCP. Но даже эти высоконадежные системы начинают разваливаться, если в систему добавляется неожиданная задержка сети. Вам не понадобится много времени, чтобы отправиться в мир каскадных сбоев, выборов лидера, раздвоения ведущих узлов и потери данных.

Отказы, возникающие по причине сетевой задержки, имеют аналог в распределенной человеческой системе – организации. Мы создаем различные надежные системы для отслеживания процесса работы, такие как системы трекинга тикетов, асинхронные чаты и платформы Lean, Scrum или Kanban. Эти системы предназначены для эффективной работы, когда связь работает быстро и бесперебойно, и быстро начинают рушиться, как только замедляется связь. Обычно это случается, когда общение начинает выходить за рамки команды или отдела. Эффективность многих структур основана на коротких петлях обратной связи между людьми. Фактически многие из этих структур предназначены для сокращения петель обратной связи, но, как правило, они не предназначены для создания новых коммуникационных структур. Отсутствие гибких коммуникаций может привести к остановившейся работе, несоблюдению сроков и неудовлетворенным ожиданиям. Автономность и эффективность достигается благодаря доверительным отношениям. Ожи-

даемые прерывания эффективно обрабатываются в системе, но неожиданные прерывания могут потеряться в электронных письмах, журналах протоколов и т. д., никогда не передавая необходимый контекст нужным людям.

Поиск новых каналов коммуникации

Проблема в том, что все эти потерянные электронные письма и просроченные тикеты являются обещанием для внешнего участника системы. Последствия небольшого сбоя в коммуникации могут очень негативно отразиться на всей организации. Это могут быть, например, недовольные клиенты, потерянный доход или утрата репутации. Подобные каскадные сбои слишком распространены, особенно в индустрии программного обеспечения. Достаточно одной неверной оценки размера и объема проекта. Именно поэтому инструменты управления рабочими процессами так популярны и эффективны. Организации находятся в постоянном поиске точек соприкосновения между участниками системы.

Размышляя о том, как уменьшить задержки в сети и создать больше маршрутов коммуникации, я искал способы выстраивать новые каналы связи. Очевидно, что лучше всего использовать каналы, которые уже построены. Просто нужно превратить грунтовую дорогу в асфальтированное шоссе. С одной стороны, каналы коммуникации между людьми часто прерываются, проходя через промежуточные узлы. Но с другой – отработанное взаимодействие двух участников системы является наиболее эффективной формой общения. Так как же избежать ловушек прерванной коммуникации, но сохранить скорость и эффективность?

10.3.6. Пример 2: связь между точками

Глядя на риск, связанный со стандартными каналами связи внутри организации, я задумался о том, как расширить сеть. Число Данбара[1] говорит нам, что существует ограничение на количество людей, где-то около 150 человек, с которыми вы можете установить и поддерживать стабильные отношения. Это означает, что существует жесткое ограничение на количество новых связей, которые вы можете установить в организации. Многие организации, которые зависят от высокой связности, часто нуждаются в связывании более чем 150 человек. Как же нам справиться с этими противоречиями? Теоретически решение состоит в том, чтобы объединить людей в команды, а затем установить коммуникации между командами через связи между несколькими членами этих команд.

Гипотеза

Когда я размышлял над этой концепцией, наткнулся на сообщение в блоге[2] от ребят из Etsy об их концепции «тренировочного лагеря» (bootcamp). Каж-

---

[1] *R. I. M. Dunbar.* Co-evolution of Neocortex Size, Group Size and Language in Humans // Behavioral and Brain Sciences 16 (4), 1993.

[2] *Marc Hedlund and Raffi Krikorian.* The Engineer Exchange Program // Etsy Code as Craft, Sept. 10, 2012, https://oreil.ly/sO7fK.

дый новый сотрудник проводит непродолжительное время, от нескольких недель до нескольких месяцев, с другими командами в организации, прежде чем официально присоединится к команде, в которую он был нанят. Подобная стажировка помогает им увидеть организацию с разных точек зрения. Но это также создает естественные каналы связи через установленные доверительные отношения. В SportsEngine мы всегда находимся в состоянии роста, будь то наём новых людей или приобретение новых компаний (более 15 с тех пор, как я начал работать в 2011 году), каждая из которых имеет собственное программное обеспечение и людей. Это почти постоянное расширение требует формирования новых связей. Вопросы, которые мы себе задали, были такими: что, если сотрудники целенаправленно и регулярно будут проводить время с разными командами? Какое влияние это окажет на доставку, доступность и масштабируемость продуктов, при условии что у нас их довольно много?

Переменный фактор

Довольно сложно найти способы вывести людей из своих зон комфорта и повседневного уклада жизни. В нашем случае мы преследовали две цели: создать более эффективную рабочую обстановку в процессе разработки продукта и лучше соответствовать будущим проектам и новым производственным проблемам. Общение на эти темы, конечно же, обычно происходило через встречи, электронную почту и тикеты. Но это было очень нерегулярное общение. Мы нуждались в новом средстве коммуникаций, где общение было бы более регулярным.

Опираясь на концепцию тренировочного лагеря Etsy, операционная команда (отвечающая за развертывание и эксплуатацию) нашей компании начала программу ротации. Вместо того чтобы отправлять людей из операционной команды в группы разработчиков, мы решили, что один инженер-программист из каждой группы разработчиков проведет спринт (две недели) в операционной команде. Первоначальная группа состояла из пяти инженеров, поэтому каждые несколько месяцев каждый инженер проводил спринт в команде, формируя систему, в которой операционная команда получала бы сотрудника, работающего полный рабочий день, и каждая продуктовая команда теряла только одного инженера на 20 % времени. Цель состояла в том, чтобы расширить операционную осведомленность каждой команды. Со временем команды разработчиков получат инженера, который сможет выступать экспертом как в области продукта, так и в управлении продуктом платформы.

Другой эксперимент, который был доработан за последний год, был инверсией этого подхода. Вместо того чтобы обучать разработчика в операционной группе, я назначил по одному операционному инженеру для каждой группы разработчиков, который будет своего рода прикрепленным операционным адвокатом. Цель здесь – повысить прозрачность работы как для операционного инженера, так и для группы разработчиков. Оказавшись в группе разработчиков продукта, инженер по эксплуатации с большей вероятностью поднимет руку и задаст вопросы. Предполагается, что этот человек также будет присутствовать на сеансах разработки, развертывания и т. д., по сути, выступая в качестве заинтересованного лица в работе команды.

Результат

Эти инициативы дали нам смесь успеха и неудачи. С точки зрения хаос-инжиниринга, мы внедряли сбои в систему, забирая сотрудников из родных команд и отправляя их в другие команды на короткое время. Это компромисс между освоением новой предметной области и тем, какую работу делают эти специалисты в обычный день.

Как и следовало ожидать, реалии производства всегда работают против таких инициатив. «Нам нужен этот программист, чтобы срочно завершить работу над новым релизом». Или, что еще хуже, программист так важен для успеха своей родной команды, что теперь он внезапно потерял возможность эффективно работать в любой команде, будучи постоянно вовлеченным в разные направления.

Интересные результаты принесла концепция тренировочного лагеря. Несколько разработчиков решили задержаться подольше в операционной команде. Один даже присоединился к команде на полный рабочий год, прежде чем решил вернуться к своей первоначальной команде разработчиков. Благодаря дополнительным возможностям и навыкам разработки программного обеспечения операционная команда смогла выполнить важные проекты, которые без них заняли бы гораздо больше времени. Привлеченные разработчики практически в одиночку превратили внутреннюю службу развертывания SportsEngine из того, что было просто идеей, в настоящего монстра, выполняющего в среднем около 100 развертываний в день в 75 различных службах, и составили общее руководство для разработчиков, развертывающих службы всех типов и размеров. Этот эксперимент также заставил операционную команду научиться эффективно привлекать людей в команду. Когда в течение нескольких месяцев вы регулярно привлекаете в команду одного нового человека, это совсем другой опыт, нежели использование пяти разработчиков, практически не имеющих опыта работы в операционной команде.

Во втором варианте эксперимента, с операционными адвокатами в команде разработчиков, систему стало труднее совершенствовать. Этот вариант не настолько отличается от структуры со встроенными контурами обратной связи, как концепция тренировочного лагеря. Трудно понять, какую информацию упускает адвокат, не будучи членом команды. Что было интересно, так это то, что разработчики часто сами распознают, когда должен присутствовать адвокат, и привлекают его. Самый большой компромисс, который я заметил из концепции адвоката, касается распределения ресурсов. Если у меня есть оперативный специалист по важному проекту, и команда разработчиков вовлекает его в дискуссию или в новый проект, в котором им нужна помощь, это влияет на то, что я пытаюсь сделать.

Эти целенаправленные попытки увеличить количество связей внутри организации во многом помогли нам создать платформу, необходимую нашим сотрудникам. У нашей организации нет неограниченных ресурсов, для того чтобы нанять еще больше специалистов для решения этой проблемы, и, честно говоря, я даже не уверен, что от этого был бы толк. Вопрос эксплуатации очень важен для успеха программного обеспечения. Я постоянно

думаю о новых способах распространения операционных навыков по всей компании. Я думаю, что в более общем смысле то же самое можно сказать об общем контексте и ценности поиска общих точек соприкосновения.

10.3.7. Лидерство как новое свойство системы

Что меня привлекает в системном мышлении и теории сложности, так это то, что она применима ко всему. Когда я понял, что лидерство в организации можно рассматривать через объектив теории сложности, это буквально взорвало мой мозг. Важно отметить, что, когда я произношу слово «лидерство» в данном контексте, я не говорю о людях или группе людей, которые «руководят другими». Скорее, я говорю о лидерстве как о явлении, которое существует так же, как общение между субъектами в системе является свойством этой системы. Например, в мире алгоритмов достижения консенсуса есть «лидер» и «выборы лидера». Это свойства системы, которые меняются со временем. Это не значит, что кто-то на «руководящей должности» не способствует изменениям, на самом деле они делают это чаще, чем другие. Почему? Многие из проблем, с которыми мы сталкиваемся в распределенных системах, могут быть сопоставлены с проблемами внутри организации, и в какой-то момент ответственность в конечном итоге оказывается на плечах лидера системы.

Продвижение организации вперед

Доктор Джеймс Баркер – профессор и научный сотрудник Университета Далхаузи, специализирующийся в области организационной безопасности, теории лидерства и теории сложности. Он описывает лидерство как *независимое свойство системы, которое двигает организацию вперед*. Мне нравится это определение. В нем лидерство рассматривается как принятие решений в рамках ограниченного контекста. Лидер – это тот, кто делает определенный выбор, за который впоследствии отвечает. Это определение хорошо сочетается с выталкиванием контекста и принятия решений за рамки повседневности, к «острым краям», к практикующим специалистам, к людям, лучше всех понимающим компромиссы, на которые приходится идти каждый день.

Используйте сигналы, чтобы определить направление

Когда вы со своими решениями доходите до грани возможного, как и следовало ожидать, достижение консенсуса становится все более и более трудным. В дело вступает *локальная рациональность*. Это явление можно описать как то, что имеет смысл с точки зрения одного человека, но может не иметь смысла с точки зрения другого человека. Однако взамен вы даете свободу и ответственность тем, кто отчитывается перед вами. Те, кто взаимодействует с данной системой, являются экспертами этой системы. Они способны видеть и понимать сигналы или обратную связь от этой системы. Когда им доверяют действовать, руководствуясь этими сигналами, могут произойти удивительные вещи.

10.3.8. Пример 3: изменение базового предположения

Как технологи мы часто подвергаем сомнению саму основу, на которой стоим. Для нас это естественное состояние. Мы часто подвергаем сомнению архитектурные решения (например, «Мы выбрали правильную базу данных?», «Мы выбрали правильную топологию сети?», «Это должен быть наш собственный сервис?»). Как часто вы задаете себе подобные основополагающие вопросы, от ответов на которые зависит работа вашей организации? Что меня всегда привлекало в SportsEngine, так это моя способность менять систему. Чаще всего это происходит медленно, но это происходит. Нередко изменение системы начинается в момент, про который Эли Голдратт в «Размышлениях о теории ограничений»[1] говорил: «Измените базовое предположение, и вы измените саму систему». Основное предположение о любой системе в организации – это то, что она работает. Именно здесь расширение возможностей действующих лиц по внесению изменений в систему может оказать огромное влияние. Помогите им изменить базовое предположение организации о системе, выдав слабые сигналы, которые они могут заметить только со своей точки зрения.

Гипотеза

Сотрудники – это решение, которое нужно использовать, а не проблема, которую нужно решать. Сидни Деккер в своей статье на эту тему[2] говорит о безопасности, определяемой работником. Люди на переднем крае работы являются экспертами системы. Во-вторых, вовлеченность в общее дело является движущим фактором человеческой деятельности. Создавая место, где люди хотят работать, вы получаете счастливых и продуктивных сотрудников. Данный набор экспериментов позволил людям, которые были отдельными участниками, инженерами-разработчиками, внести существенные изменения в систему организации.

Переменный фактор

Когда вы начинаете говорить о таких абстрактных вещах, как организационные изменения и эффективное руководство, трудно перейти к разговору о конкретных действиях. Я сужу об этом по собственному опыту, поэтому лучше приведу два примера действий, предпринятых сотрудниками компании, которые превратились в фундамент нашей корпоративной культуры.

Первым был программист, испытывающий горячее желание совершенствоваться. Он был буквально одержим идеей личного роста и повышения квалификации. Будучи разработчиком-управленцем, вы можете только раз-

---

[1] *Eliyahu Goldratt*. Essays on the Theory of Constraints. Great Barrington, MA: North River Press, 1998.

[2] *Sidney Dekker*. Employees: A Problem to Control or Solution to Harness?

давать направо и налево советы о том, как другие специалисты могут улучшить свою повседневную работу, но если вы действующий член команды, то погружены в такой контекст, что поиск путей улучшения может стать вашей естественной потребностью. Именно это стремление породило программу наставничества. Наша первая итерация программы включала всего лишь одну пару разработчиков, снабженных лишь несколькими свободными рекомендациями о том, как быть эффективными, и регулярными циклами обратной связи, чтобы наблюдать положительный эффект. Эта структура сначала выросла в две дополнительные программы наставничества, а сегодня это шесть активных программ. Но эти программы родились только благодаря желанию одного программиста постоянно совершенствоваться и получать обратную связь и понимания руководства, что если эта система может принести пользу одному специалисту, она может оказать еще большее влияние на культуру обучения в организации.

Второй пример начался с предположения, что можно целенаправленно сформировать культуру совместного обучения на основе коллективного опыта. Один из наших сотрудников прочитал сообщение в блоге Spotify[1] о том, как проводить эффективные проверки работоспособности команды, и захотел попробовать применить этот метод в наших командах разработчиков в SportsEngine. Мы начали с ограниченного эксперимента над одной командой в течение года и получили хороший результат. Теперь каждая команда разработчиков проводит регулярные проверки «состояния здоровья», с оценочными карточками и увлекательной дискуссией о том, где команда эффективна и где у нее есть возможности для роста. Это еще одна внутренняя инициатива, которая из однократного эксперимента превратилась в чрезвычайно продуктивное упражнение, приносящее реальную пользу.

Результат

Описанные выше примеры не являются универсальным лекарством от всех болезней. Не каждый способен стать хорошим наставником, и не все, кто хочет наставника, способны усвоить предложенные знания. Несмотря на хорошие рабочие отношения, сиюминутные производственные задачи все равно будут вступать в противоречие с потребностью роста и развития. Если в вашей команде не сложилось близкое человеческое общение, или нет лидера, который открыт для изменений и новых идей, проверки работоспособности создадут больше проблем, чем исправят. Эффективные проверки работоспособности не должны быть обзорами эффективности управления или содержать хоть малейшую каплю вины и стыда. Они применяются только для того, чтобы узнать, как команды могут стать еще лучше, чем они уже есть.

Именно здесь минимизация радиуса поражения[2] имеет решающее значение для успеха. Найдите безопасное место, чтобы опробовать идеи с бо-

---

[1] *Henrik Kniberg*. Squad Health Check Model–Visualizing What to Improve // Spotify Labs, Sept. 16, 2014, https://oreil.ly/whbiH.

[2] См. главу 3 о минимизации радиуса поражения.

лее высоким уровнем доверия или меньшим риском для бизнеса. Вы будете поражены идеями, которые люди генерируют, когда они чувствуют, что их слушают, им доверяют и позволяют экспериментировать и ошибаться. В предыдущих примерах новые идеи были опробованы в безопасных местах; к ним относились как к экспериментам с незначительными последствиями в случае неудачи.

В этих примерах нет ничего революционного, но, в отличие от большинства подобных инициатив, они были созданы с нуля. Созданные на основе опыта практикующих специалистов, они не были нисходящими «управленческими инициативами». Если для вас лидерство – это способ действовать на основании внешних сигналов, чтобы развивать свою организацию, то вам не составит труда распознавать и поощрять инициативы персонала.

Я достигал наибольшего успеха в своей деятельности, когда прямо заявлял, что хочу «попробовать что-то». Вот парадокс: четко понимая, что ваша инициатива может потерпеть неудачу, и это нормально, в результате вы получите более высокую вероятность успеха.

10.3.9. Безопасная организация хаоса

Сейчас вы, возможно, хотите спросить меня: как вы привлекли людей на свою сторону и убедили сотрудничать? Ответ на этот вопрос, как любит говорить Эндрю Клэй Шафер: «Рубите дрова и подносите воду»[1]. Вы должны сосредоточиться на создании правильной культуры, где поощряется инициативное поведение, а дальше все будет работать само. Мне нравится использовать модель, созданную социологом Роном Веструмом[2]. Вы можете использовать это, чтобы понять состояние вашей организации. Организация относится к одному из трех типов: патологическая, бюрократическая и генерирующая. Классификация основана на том, как в организации выстроено общение и сотрудничество. Например, в патологической организации новые идеи подавляются; в бюрократической организации новые идеи пробиваются с трудом; в генерирующей организации новинки приветствуются. Эта модель может служить тестом на готовность вашей организации к экспериментам. Если вы не уверены, что ваша организация относится к числу генерирующих, я бы на вашем месте даже не пытался проводить масштабные эксперименты наподобие тренировочного лагеря. Вы можете проводить эти эксперименты только в условиях автономии, которая способствует обучению и совершенствованию, а не в жестких рамках догм и бюрократии. Соблюдение корпоративных правил никогда не было моей сильной стороной, но кое-что я делаю хорошо – это привожу аргументы о необходимости системных изменений.

---

[1] «Chop wood and carry water» – фраза из известной буддистской притчи. В переносном смысле означает, что нужно спокойно и упорно делать свое дело, отринув личные амбиции. – *Прим. перев.*

[2] *Ron Westrum*. A Typology of Organizational Cultures // BMJ Quality and Safety 13 (2004), ii22–ii27.

10.3.10. Все, что вам нужно, – это высота и направление

Любой эксперимент, который вы проводите, требует от вас достаточного запаса надежности, чтобы избежать катастрофических последствий. В авиации есть буквальное определение этого запаса – высота над уровнем моря. Если вы еще не упали, время есть. Но этот запас бесполезен, если вы не двигаетесь. Любой эксперимент, который вы запускаете, должен иметь заявленные цели, желаемые результаты и запас надежности, чтобы успеть закончить эксперимент, прежде чем он нанесет серьезный вред участникам или бизнесу. Во всех предыдущих экспериментах, которые я описал, был явно неприемлемый результат. Если в течение игрового дня команда потратила порядка часа, пытаясь решить проблему «специалиста в отпуске», хотя ее можно было решить сразу, мы не стеснялись говорить об этом. Мы не упустили ни одной возможности обучения, мы узнали, что нам нужно. В примере с тренировочным лагерем мы довольно гибко относились к перемещениям временных специалистов; если они были слишком загружены на основной работе, они пропускали один цикл или подменялись другим сотрудником. То же самое касается операционных адвокатов: если они оказывались недоступны по какой-либо причине, мы заменяли их другим человеком или привлекали позднее. Когда подобные нестыковки и подмены становятся обычным явлением, вам нужно переосмыслить саму концепцию. В конце концов, мы отказались от идеи тренировочного лагеря, потому что все наши сотрудники были слишком заняты, или переехали, или поменялись ролями. Все перемешалось, так что пришла пора эксперименту закончиться.

10.3.11. Замыкайте петли обратной связи

Системное мышление вращается вокруг петель обратной связи. Они подобны чувствам, нервной системе – они несут сигнал к органам чувств. Без обратной связи вы не сможете понять, в каком направлении движетесь и движетесь ли вообще. Когда «Принципы хаос-инжиниринга» говорят о понимании устойчивого состояния системы, речь идет о ваших петлях обратной связи. У вас достаточно сигналов от системы, чтобы знать, как она работает нормально. Само собой разумеется, если вы меняете систему, вам может потребоваться реализовать новые петли обратной связи, чтобы ощутить влияние ваших изменений. Что касается участников тренировочного лагеря, мы постоянно следили, ощущают ли они ценность своих действий, находят ли свою работу интересной и чувствуют ли, что рационально расходуют свое время. Эта ретроспектива оказалась критически важной при выявлении проблемы с разработчиками, выполняющими роль «технического лидера», которым было трудно оставаться вдали от своей команды в течение длительного периода времени. Создание петель обратной связи в человеческих системах обычно не составляет большого труда. Проведите интервью с заинтересованными людьми – и получите качественный анализ ситуации. Если они ощущают ценность идеи, вы движетесь в правильном направлении.

10.3.12. Если вы не ошибаетесь, вы не учитесь

Если у вас есть эти полезные петли обратной связи, если все «просто работает», значит, вы не проявляете должного усердия. Помните, что все системы по своей природе скрывают в себе как успехи, так и неудачи. Глядя на ваш эксперимент через розовые очки, вы не получите нужный вам результат. Ожидайте, что вы ошибетесь, внесите коррективы и продолжайте двигаться. Неудача – это обучение в действии. Эксперименты часто никогда не воплощаются в жизнь, потому что они либо слишком дороги, либо не сулят достаточной пользы. У нас считается нормой что-то попробовать в течение некоторого времени, а затем отказаться почти так же быстро, как это приняли. Команда, полная энергичных учеников, создает удивительную культуру. Если у вас появляется все больше идей и находится время, чтобы опробовать их, значит, вы движетесь в правильном направлении.

Об авторе

Энди Флинер – гуманист и идеолог «нового взгляда», который считает, что программное обеспечение – это как люди, разрабатывающие и поддерживающие его, так и люди, использующие его. Он является старшим менеджером по эксплуатации платформы в SportsEngine, где с 2011 года разрабатывает и запускает приложения класса SaaS (software-as-a-service, «программное обеспечение как услуга») для молодежных и любительских спортивных организаций.

Глава 11

Роль человека в системе

Автор главы: **Джон Алспоу**

Сама идея о том, что проведение экспериментов на программной системе служит хорошим способом понять ее поведение, исходит из соображения о том, что поведение системы не может быть полностью предопределено во время ее разработки. Большинство современных разработчиков программного обеспечения понимают, что *написание* программного обеспечения и *понимание того, как оно работает* (и как может случиться отказ), – две совершенно разные вещи.

Достижение и сохранение понимания поведения системы зависят от способности людей перестраивать «ментальные модели» ее поведения[1]. Процесс экспериментов в производственных системах можно рассматривать (наряду с анализом после инцидента) как плодотворную возможность для перестройки или «обновления ментальной модели».

Большая часть этой книги посвящена подходам, нюансам, деталям и взглядам на то, *как* работает хаос-инжиниринг. Впечатляет сам факт существования этой книги и наличие этой темы в сегодняшних разговорах о разработке и эксплуатации программного обеспечения. Вероятно, эта тема будет развиваться и дальше, а значит, *доверие* к системе можно будет развивать более сложными и современными способами.

Как и следовало ожидать от индустрии программного обеспечения, в ней появляются все новые и новые идеи относительно автоматизации хаос-инжиниринга. Эта тенденция понятна (с учетом того, что создание «автоматизации» является главным смыслом для разработчиков программного обеспечения). Однако не все задумываются об иронии введения все большей автоматизации, предназначенной главным образом для обеспечения уверенности в неопределенном поведении … существующей автоматизации.

Как сказано в «Принципах хаос-инжиниринга»[2]:

---

[1] *David D. Woods*. STELLA: Report from the SNAFUcatchers Workshop on Coping with Complexity. Columbus, OH: The Ohio State University, 2017.

[2] Principles of Chaos Engineering, June 10, 2019, https://principlesofchaos.org.

> Хаос-инжиниринг – это дисциплина экспериментов над системой, призванных укрепить уверенность в способности системы противостоять турбулентным условиям окружающего мира.

Как выглядит «укрепление уверенности» в реальных условиях?

- люди *чувствительны к контексту*, в котором чувствуют себя уверенно, то есть их уверенность зависит от ответов на вопросы *что*, *когда* и *как* происходит;
- процесс эксперимента также зависит от контекста в том, что движет целью (целями) и деталями данного эксперимента, обоснования того, когда и как он должен быть выполнен, и как интерпретируются результаты.

Это те вопросы, которые нам необходимо изучить, чтобы понять, что на самом деле означает «роль человека в системе».

11.1. Эксперименты: почему, как и когда

Подход, который хаос-инжиниринг предлагает разработчикам программного обеспечения, достаточно прост: повысить *уверенность* в поведении системы при различных условиях и различных параметрах, которые могут выйти за пределы оптимистического «счастливого пути» функционирования.

11.1.1. Почему

Этот *modus operandi* помещает хаос-инжиниринг в хорошую компанию с другими методами и технологиями, укрепляющими уверенность в поведении программного обеспечения, например:

- различные формы тестирования (к примеру, модульное, интеграционное, функциональное и т. д.);
- стратегии проверки кода (несколько точек зрения на одно и то же изменение кода);
- «темное» развертывание на производстве[1];
- «наращивание» новых функциональных возможностей по группам (например, только для сотрудников) или в процентах (например, 50 % трафика клиентов).

Эти методы достигают своей цели разными путями: как функции во фрагментах кода, как взаимодействующие между собой группы функций или как взаимосвязанные системы либо компоненты. В любом случае, разработчикам доступен широкий спектр методов укрепления доверия, который включает эксперименты хаос-инжиниринга.

---

[1] Развертывание изменений в программном обеспечении с указанием только некоторой части пользователей называется «темным» развертыванием. См.: *Justin Baker*. The Dark Launch: How Google and Facebook Release New Features. April 20, 2018, Tech.co, https://oreil.ly/jKKRX.

Эти методы укрепления доверия, как правило, связаны с относительно современной парадигмой, известной как «непрерывная доставка»[1,2], и было бы разумно включить в эту парадигму хаос-инжиниринг как надежный подход[3].

Некоторые эксперименты обусловлены не отсутствием доверия со стороны команды, ответственной за «цель» эксперимента, а отсутствием доверия *других* сторон. Например, представьте команду разработчиков, которая вносит существенные изменения в свою службу и хочет продемонстрировать заинтересованным сторонам должную осмотрительность или «готовность к производству». Таким образом, эксперимент может служить неявным доказательством того, что были предприняты определенные усилия.

11.1.2. Как

Как упоминалось в предыдущих рассуждениях, эти методы *зависят от контекста*, то есть они могут быть более или менее продуктивными в зависимости от обстоятельств.

Например, в отсутствие модульного или функционального тестирования некоторые изменения кода могут потребовать более тщательного изучения путем проверки кода. Или в определенный период (например, во время пиков трафика в праздничные дни, испытываемых сайтами электронной коммерции) можно применить более консервативные и медленные способы развертывания кода. Изменения схемы базы данных, перестройка поисковых индексов, маршрутизация сетей нижнего уровня и «косметические» изменения разметки страниц – все это примеры действий, которые несут в себе различные неопределенности, различные последствия, если все идет «не туда», и различные непредвиденные обстоятельства, на которые необходимо реагировать по мере возникновения.

В этом смысле хаос-инжиниринг ничем не отличается; эффективно экспериментировать – значит быть контекстно-зависимым. Необходимо учитывать следующее:

- над чем вы экспериментируете (а над чем нет);
- когда вы экспериментируете (и как долго нужно это делать, чтобы быть уверенным в результатах);
- как вы экспериментируете (и все подробности о конкретных реализациях эксперимента);
- для кого предназначен эксперимент;
- какова может быть более широкая цель эксперимента, помимо укрепления доверия к *разработчику* эксперимента?

---

[1] Непрерывная доставка – это «способность безопасно и быстро получать изменения всех типов – включая новые функции, изменения конфигурации, исправления ошибок и эксперименты – в производство или в руки пользователей». См. https://continuousdelivery.com.

[2] *Jez Humble and David Farley.* Continuous Delivery. Upper Saddle River, NJ: Addison-Wesley, 2015.

[3] См. также главу 16, которая рассказывает в том числе про непрерывную проверку.

На практике часто звучит несколько вызывающий ответ «когда как», и *только люди* могут рассказать вам больше о том, что значит «когда как» в разных обстоятельствах. Эта глава исследует «когда как».

11.1.3. Когда

Что влияет на решение, *когда* начинать тот или иной эксперимент? На практике команды разработчиков показали, что различные обстоятельства могут влиять не только на то, когда *начинать* эксперимент, но и когда его *заканчивать*. Когда одного разработчика спросили, как часто можно отменять или приостанавливать эксперимент, он ответил:

> В некоторых случаях эти эксперименты часто откладывались … потому что была другая команда, которая хотела скоординировать наши упражнения или эксперимент с их циклом развертывания. Представьте, мы подготовимся к отказу всего региона, и тут они говорят: «О нет, пожалуйста, подождите! Мы находимся в середине процесса передислокации»[1].

Условия, которые могут повлиять на сроки проведения экспериментов, включают:

- реагирование на неожиданную динамику или поведение, наблюдаемое в инциденте;
- запуск новой функции/продукта/подсистемы с потенциально неожиданным поведением;
- интеграция с другими частями системы (то есть добавление новой части инфраструктуры);
- подготовка к высокому спросу, например «черная пятница» или «киберпонедельник»;
- адаптация к внешним обстоятельствам (колебания цены акций, внезапные новости и т. д.);
- подтверждение теорий о неизвестном или редком поведении системы (например, гарантия, что запасная реализация не разрушит критически важную для бизнеса инфраструктуру).

Во время инцидентов: «Это связано с выполнением эксперимента?»

Неопределенность и двусмысленность являются отличительными чертами инцидентов, происходящих в программном обеспечении. Обычная эвристика, при помощи которой инженеры по эксплуатации пытаются понять, что на самом деле происходит с их системой, заключается в сокращении числа потенциальных участников почти наобум (без предварительного всестороннего обоснования того, на какие из них стоит обратить внимание) путем остановки процессов, отключения служб и т. д.[2]

---

[1] Personal communication, 2019.

[2] *John Allspaw.* Trade-Offs Under Pressure: Heuristics and Observations of Teams Resolving Internet Service Outages. Master's thesis, Lund University, Lund, Sweden, 2015.

Хаос-эксперименты в производстве не исключены из перечня потенциальных источников инцидентов, что вполне объяснимо, если вспомнить, как часто команды впадают в растерянность от непонимания того, что происходит с их системами.

Но как насчет автоматизации и исключения людей из рабочего цикла?

Как я уже говорил, регулярно возникают вопросы о том, какие части хаос-инжиниринга можно «автоматизировать»[1]. Один из аспектов, на которые стоит обратить внимание, – это описание видов деятельности, из которых состоит хаос-инжиниринг, как сказано в «Принципах»[2].

1. Начните с определения «устойчивого состояния» как некоторого измеримого результата системы, указывающего на нормальное поведение.
2. Предположите, что это устойчивое состояние сохранится как в контрольной группе, так и в экспериментальной группе.
3. Введите переменные факторы, которые отражают реальные события, такие как отказ серверов, отказ жестких дисков, разрыв сетевых подключений и т. д.
4. Попытайтесь опровергнуть гипотезу об устойчивом состоянии, ища разницу в установившемся состоянии между контрольной группой и экспериментальной группой.

Какие из этих действий *могут* или *должны* быть автоматизированы?

Какие из них *не могут* быть автоматизированы и *не должны* быть автоматизированы? Как должна помочь автоматизация?

Чтобы критически подойти к этим вопросам, стоит немного остановиться на теме распределения функций.

11.1.4. Распределение функций, или Каждый хорош по-своему

Распределение функций (function allocation) было впервые упомянуто в документе 1951 года, озаглавленном «Роль человека в эффективной системе аэронавигации и управления движением»[3], где представлены указания относительно того, какие «функции» (задачи, рабочие места) следует «распределять» людям, а какие – выделять машинам (сегодня это чаще всего компьютер). Эта идея была оформлена в виде так называемого *списка Фиттса* (Fitts list). См. рис. 11.1 и табл. 11.1.

---

[1] Этот термин заслуживает жирных кавычек, когда речь заходит о компьютерных программах, которые и без того представляют собой способ записи автоматизации. Действительно, все вычисления компьютера выполняются за счет цепочек автоматических действий. Не может быть никаких «неавтоматизированных» разновидностей современных вычислений; даже такие рутинные действия, как ввод с клавиатуры, невозможно выполнить в буквальном смысле слова вручную.

[2] Principles of Chaos Engineering, retrieved June 10, 2019, https://principlesofchaos.org.

[3] *P. M. Fitts, ed.* Human Engineering for an Effective Air-Navigation and Traffic-Control System. 1951, Washington, DC: National Research Council.

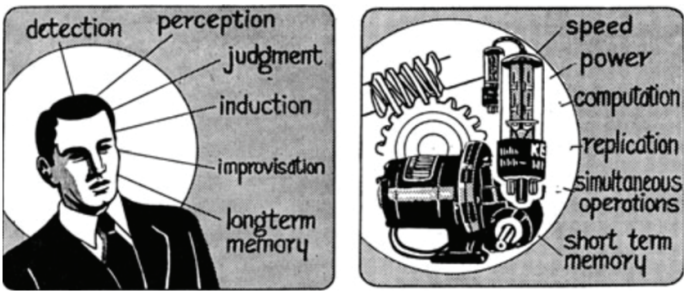

Рис. 11.1 ❖ Иллюстрации списка Фиттса взяты из оригинального отчета[1] 1951 года

Таблица 11.1 Оригинальный список Фиттса

В чем люди сильнее машин	**В чем машины превосходят людей**
Способность распознавать слабые звуки и неясные изображения	Способность быстро реагировать на сигналы управления и применять большую силу плавно и точно
Способность воспринимать упорядоченные структуры света и звука	Способность выполнять повторяющиеся, рутинные задачи
Умение импровизировать и использовать гибкие процедуры	Возможность воспринимать информацию короткое время, а затем стирать ее полностью
Способность хранить очень большие объемы информации в течение длительного времени и вспоминать соответствующие факты в надлежащее время	Способность мыслить дедуктивно, включая вычислительные способности
Способность мыслить индуктивно	Способность выполнять очень сложные операции (делать много разных вещей одновременно)
Способность выносить суждение	

Источник: *P. M. Fitts, ed.* Human Engineering for an Effective Air-Navigation and Traffic-Control System.

Многим программистам строки из списка Фиттса покажутся знакомыми. В каком-то смысле это философские основы программирования. Хотя распределение функций, начавшееся со списка Фиттса и позднее получившее название «HABA-MABA»[2] (humans-are-better-at/machines-are-better-at, «люди справляются лучше / машины справляются лучше»), в течение десятилетий являлось основным подходом к изучению человеческого фактора, существуют некоторые относительно свежие критические замечания в отношении этого подхода, вытекающие из эмпирических исследований в области когнитивных систем.

---

1 *P. M. Fitts*. Human Engineering for an Effective Air-Navigation and Traffic-Control System.

2 Это современная версия. В оригинале было сокращение «MABA-MABA» (*men*-are-better-at/machines-are-better-at, «мужчина справляется лучше / машины справляются лучше»), которое отражает гендерные стереотипы того времени.

Детальное изучение этих критических замечаний выходит за рамки главы. Однако мы можем кратко перечислить здесь основные проблемы парадигмы HABA-MABA.

11.1.5. Миф замещения

Идея о том, что работу можно разложить на изолированные задачи, которые затем «распределяются» между соответствующими агентами (человек и машина), вызывает сомнение по ряду причин[1]:

> Именно это Холлнагель (1999) называл «распределением функций путем замены». Идея заключается в том, что автоматизация может быть представлена как простая замена людей машинами, сохраняющая исходную систему при одновременном улучшении некоторых ее показателей (меньшая нагрузка, лучшая экономия, меньшее количество ошибок, более высокая точность и т. д.).
> В основе замены функций лежит идея о том, что люди и компьютеры (или любые другие машины) имеют фиксированные сильные и слабые стороны и что задача автоматизации заключается в том, чтобы извлечь выгоду из сильных сторон, одновременно заменяя или компенсируя слабые стороны. Проблема в том, что иногда компьютерная сила не заменяет человеческую слабость, а лишь раскрывает новые сильные и слабые стороны человека – часто в непредсказуемой форме[2].

Когда предлагаются новые формы автоматизации на основе логики «доверить компьютерам работу, которую они делают хорошо», часто возникает предположение, что[3]:

> …новая автоматизация может заменить участие человека, не воздействуя в исходной системе больше ни на что, кроме выходных данных. Эта точка зрения опирается на представление о том, что сложную систему можно разложить на ряд по существу независимых задач.
> Тем не менее исследования влияния новых технологий показали, что эти предположения являются необоснованными (их можно назвать мифом о замещении). В реальных сложных системах задачи и действия тесно взаимосвязаны или взаимозависимы.

Актуальна ли эта точка зрения в случае «автоматизации» методики хаос-инжиниринга?

В следующей главе этой книги Питер Альваро предлагает автоматизировать *выбор* эксперимента[4], обосновывая это тем, что существенное препятствие, которое необходимо преодолеть в хаос-инжиниринге, – это не просто наличие «комбинаторного пространства» возможных сбоев для экспери-

---

1 *Sidney W. A. Dekker*. Safety Differently. Second Edition. CRC Press: Boca Raton, FL, 2015.

2 *Lisanne Bainbridge*. Ironies of Automation», Automatica, Vol. 19, № 6, 1983.

3 *Nadine B. Sarter, and David D. Woods*. Team Play with a Powerful and Independent Agent: Operational Experiences and Automation Surprises on the Airbus A-320. Human Factors, 39 (4), 1997.

4 Здесь необходимо подчеркнуть, что в данном случае эксперименты именно *выбирают* из предложенного списка вариантов, а *не разрабатывают* по «указанию» приложения.

ментов[1]. Необходимо выбрать эксперимент, который наиболее эффективно выявляет нежелательное поведение системы.

На первый взгляд идея кажется разумной. Если бы мы могли разработать и запустить приложение, которое «обнаруживает» слабые стороны или предметы, заслуживающие экспериментальной проверки, и представляет людям их перечень на выбор, то это приложение потенциально «решило» бы задачу выбора действий в хаос-инжиниринге. Приложение, выполняющее эту «выборочную» работу, может проанализировать программную систему и затем представить результаты человеку, который в свою очередь выберет, какой эксперимент выполнить. Возможно, самая очевидная польза от такого приложения заключается в создании повода для диалога между разработчиками.

Однако здесь вызывает сомнение фундаментальное представление о том, что такое приложение «автоматизирует» мешающие факторы. Как отмечает Бейнбридж в своей оригинальной статье «Ирония автоматизации»[2], ирония ситуации заключается в следующем:

> Ошибки разработчика [автоматизации] могут стать основным источником проблем в эксплуатации системы.

Будет ли алгоритм, выполняющий этот автоматический «выбор» эксперимента, полностью безошибочным? Если нет, то сколько внимания придется уделять самой автоматизации? Будет ли приложение требовать наличия опыта для эффективного использования?

Ирония (используя термин Бейнбриджа) может заключаться в том, что автоматизация, призванная облегчить жизнь людям, сама по себе может стать источником новых проблем, которые необходимо рассмотреть и решить. Действительно, автоматизация не достается бесплатно.

Вторая ирония, которую отмечает Бейнбридж:

> Разработчик [автоматизации], который пытается исключить оператора, по-прежнему оставляет оператора выполнять задачи, которые не сумел автоматизировать.

Хотя теоретически мы освободили разработчиков от исследования пространства потенциально возможных сценариев эксперимента, мы ввели новые задачи, которых раньше у них не было:

- они должны решить, *как часто* следует запускать этот автоматический искатель экспериментов, учитывая, что система постоянно изменяется и используется;
- они будут отвечать за остановку или приостановку автоматизации, работающей на производстве при определенных условиях (что само по себе является экспериментом);
- теперь у них появилось еще одно приложение, которое требует поддержки и развития; в идеале нужно вводить в него улучшения и исправлять ошибки по мере обнаружения.

---

[1] *Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein.* Lineage-Driven Fault Injection», Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data-SIGMOD 15, 2015, doi: 10.1145/2723372.2723711.

[2] *Bainbridge*. Ironies of Automation.

11.2. Вывод

Появление хаос-инжиниринга раскрывает перед нами потрясающие возможности, которые нельзя упускать. Несмотря на заявления об обратном, хаос-инжиниринг *не* является инструментом для уменьшения количества ошибок и/или инцидентов в программных системах.

На самом деле хаос-инжиниринг – это новый и продуктивный способ выработать глубокое понимание истинного устройства и поведения системы как у разработчиков, так и у операционных сотрудников; способ выстроить продуктивный диалог специалистов по всем видам деятельности (генерация гипотез, определение устойчивого состояния, выражение озабоченности и опасений по поводу неопределенности или двусмысленности). Поэтому мы должны рассматривать хаос-инжиниринг как подход, развивающий гибкость и контекстное восприятие, присущие только людям. Ведь именно людям приходится справляться с возрастающей сложностью, которая приходит вместе с новыми вычислительными системами.

Люди являются конечными бенефициарами всех технологий, включая программирование. Причина, по которой важно *укреплять доверие* к программным системам с помощью экспериментов, заключается в том, что эти программные системы *важны для людей*, иногда по разным причинам.

Люди всегда должны быть в курсе дел с программным обеспечением, потому что люди несут ответственность, а программы – нет:

> Неотъемлемой частью человеческого бытия является способность брать на себя обязательства и нести ответственность за свои действия. Компьютер не способен принять на себя обязательства[1].

В конечно счете хаос-инжиниринг по-своему помогает людям, ответственным за разработку и эксплуатацию сложных систем.

Об авторе

Джон Алспоу более двадцати лет работает в области разработки и эксплуатации сложных систем в самых разных средах. Публикации Джона включают «Искусство планирования ресурсов» и «Веб-операции» (O'Reilly), а также предисловие к «Руководству по DevOps» (IT Revolution Press). Его совместное выступление с Полом Хаммондом на конференции Velocity в 2009 году на тему «10+ развертываний в день: интеграция разработки и эксплуатации» способствовало рождению движения DevOps как самостоятельной профессии. Джон работал техническим директором в Etsy и получил степень магистра по специальности «Человеческие факторы и безопасность систем» в Университете Лунда.

---

[1] *Terry Winograd, Fernando Flores.* Understanding Computers and Cognition. Reading, MA: Addison-Wesley, 1986.

Глава 12

Проблема выбора эксперимента и ее решение

Автор главы: **Питер Альваро**

Трудно представить масштабную, реально существующую систему, которая не предполагает взаимодействия людей и машин. Когда мы проектируем такую систему, зачастую самой сложной (и самой важной) частью работы является выяснение того, как наилучшим образом использовать два разных вида ресурсов. В этой главе я утверждаю, что сообщество специалистов по отказоустойчивости должно переосмыслить подход к использованию людей и компьютеров в качестве ресурсов. В частности, я утверждаю, что формирование представления о режимах сбоев системы с использованием наблюдаемой инфраструктуры и в конечном итоге реализацию этих представлений в форме хаос-экспериментов лучше доверить компьютеру, чем человеку. Завершая главу, я приведу доказательства того, что сообщество готово двигаться в этом направлении.

12.1. Выбор экспериментов

Независимо от методологий, обсуждаемых в остальной части книги (и в дополнение к ним), всегда существует проблема *выбора эксперимента*, а именно: выбор отказа и места для его внедрения в систему. Как мы уже видели, правильно подобранный эксперимент способен выявить отказ до того, как это сделают пользователи, и рассказать много нового о поведении масштабной распределенной системы. К сожалению, из-за сложности, присущей таким системам, число возможных отдельных экспериментов, которые мы могли бы провести, является астрономическим – степенная функция от количества взаимодействующих случаев. Например, предположим, что мы хотим полностью протестировать влияние каждой возможной комбинации отказов

узлов в приложении, включающем 20 различных сервисов. Существует 2^{20} – более миллиона! – способов, которыми даже такая небольшая распределенная система может испытать отказы узлов!

«Это не баг, это функция!»

Вы заметите, что в этом разделе основной целью хаос-экспериментов я считаю поиск «багов» (ошибок в коде). Хотя я использую это слово в неформальном смысле, оно имеет две одинаково правильные интерпретации, одну узкую и одну довольно широкую:

- некоторые из наиболее пагубных ошибок в распределенных системах – это малозаметные ошибки в логике отказоустойчивости (например, связанные с репликацией, повторными попытками, откатами, восстановлением и т. д.). Эти ошибки часто проявляют себя только во время интеграционного тестирования, в котором происходят конкретные отказы (например, отключение машины), – следовательно, они могут долгое время скрываться в коде и вызывать катастрофические проблемы при стечении обстоятельств в производстве. Большая часть моих собственных исследований сосредоточена именно на этом классе ошибок, представляющих собой «отложенные отказы»[1];
- как я подробно поясню позже в этом разделе, имеет смысл проводить только такие хаос-эксперименты, которые, по предположению, должны *допускаться системой*. Если эксперимент приводит к неожиданному результату (такому как заметная аномалия, потеря данных, недоступность системы и т. д.), то очевидно, что предположение было неверным – мы где-то допустили ошибку! Это может быть проявлением «спящей ошибки» в коде, о которой говорилось выше, но может быть следствием ошибочной конфигурации, чрезмерно консервативной политики безопасности, или правила брандмауэра, или неправильного понимания на уровне архитектуры того, как должна работать эта часть инфраструктуры. Чаще всего проблема заключается в сочетании этих логических ошибок. Обычные хаос-эксперименты помогают выявить подобные ошибки.

Читая этот раздел, вы можете выбрать любую интерпретацию на свое усмотрение.

Как из этого огромного комбинаторного пространства выбрать эксперименты, на которые придется тратить время и ресурсы? Поиск перебором невозможен – солнце погаснет раньше, чем мы выполним все возможные эксперименты в распределенной системе даже скромных размеров. Современное состояние технологии дает два обоснованных, хотя и неудовлетворительных ответа, которые мы уже встречали в данной книге.

---

[1] *Haopeng Liu*. FCatch: Automatically Detecting Time-of-fault Bugs in Cloud Systems // Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, Williamsburg, VA, 2018.

12.1.1. Случайный поиск

Ранние методы хаос-инжиниринга (например, *метод обезьяны*, у которой один из первых инструментов, Chaos Monkey, позаимствовал свое название) исследовали пространство отказов *случайным* образом (рис. 12.1). Случайный подход имеет много преимуществ. Во-первых, его просто реализовать: как только мы перечислили пространство неисправностей (например, все возможные комбинации аварийных ситуаций), мы можем просто выбирать эксперименты, делая единообразные и случайные выборки. Случайный подход также не требует знания предметной области: он одинаково хорошо работает в любой распределенной системе.

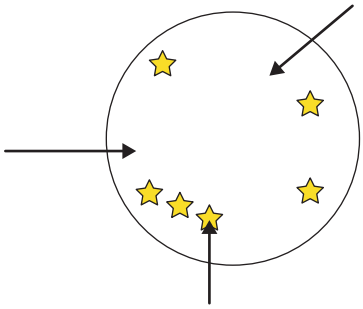

Рис. 12.1 ❖ Круг представляет собой пространство возможных ошибок, которые может внедрить инструмент хаоса. При случайном выборе экспериментов некоторые хаос-эксперименты (стрелки) выявляют программные ошибки (звездочки), которые приводят к критическому отказу системы

К сожалению, случайный подход не особенно эффективен по отношению к распределенным системам. Комбинаторное пространство отказов достаточно велико, и подходы, которые просто наобум проникают в него, вряд ли позволят выявить серьезные ошибки в коде и неправильные конфигурации, которые обнаруживаются только при определенном сочетании независимых неполадок (например, почти одновременный выход из строя коммутатора серверной стойки и сбой узла репликатора данных). Случайный подход также не дает нам никакого представления о том, насколько хорошо проведенные нами эксперименты «охватили» пространство возможных экспериментов. Он ничего не говорит нам о том, как долго следует проводить случайные хаос-эксперименты, прежде чем сделать вывод, что тестируемая система достаточно надежна для безопасного развертывания нового кода.

12.1.2. Настало время экспертов

Значительная часть этой книги посвящена альтернативе случайному выбору экспериментов: использованию знаний системных экспертов. Выбор эксперимента под руководством эксперта (рис. 12.2) расположен на проти-

воположном конце спектра относительно случайного тестирования. Этот подход расходует значительные ресурсы и время (эксперты стоят дорого), зато может сосредоточиться на слабых сторонах системы или на граничных случаях, а не применять везде одну и ту же стратегию. В отличие от случайного тестирования, которое не может учиться на своих ошибках и в каком-то смысле никогда не добивается прогресса, выбор эксперимента под руководством эксперта может опираться на результаты предыдущих экспериментов (положительные или отрицательные) для формулирования новых гипотез и постепенного их уточнения. Если процесс уточнения проводится достаточно тщательно, то постепенно он выявляет пробелы в ментальной модели эксперта исследуемой системы, нацеливает на эти пробелы эксперименты и в конечном итоге заполняет их путем либо выявления отказа, либо получения новых доказательств устойчивости системы. Это звучит великолепно! Давайте же скорее займемся этим.

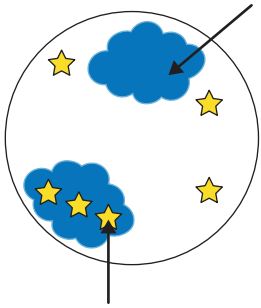

Рис. 12.2 ❖ Выбор эксперимента под руководством эксперта, основанный на инфраструктуре системы, может определить области (облака) пространства эксперимента, в которых результаты уже выполненных экспериментов могут говорить о том, что в этой области эксперименты больше не нужны. В этом примере некоторые эксперименты (верхняя область в виде облака) не выявляют ошибку, а вместе эти эксперименты определяют область, в которой нам нечего искать. Другие эксперименты (нижнее облако) выявляют ошибки и предупреждают нас о существовании иных ошибок в этой области (опять же, нам незачем запускать здесь новые эксперименты)

К сожалению, для того чтобы использовать экспертный отбор экспериментов, предприятию необходимо нанять и подготовить *экспертов*.

Роль эксперта

Частью работы эксперта является *выбор* отказов: они должны решить, на какие эксперименты потратить время, деньги и усилия. Но более интересно рассмотреть эту проблему с другой стороны: эксперт должен решить, какие эксперименты *не нужно* проводить. На основании чего мы решаем пропустить какую-то точку в комбинаторном пространстве возможных экспериментов? Это справедливо, если мы знаем о системе достаточно много, чтобы точно предсказать, что произойдет, если внедрить отказ в данной точке. Как уже было сказано в данной книге, если мы заранее знаем, что конкретный

эксперимент вызовет отказ, нет необходимости (не говоря уже о неуважении к окружающим!) проводить этот эксперимент. Вместо этого мы должны направить ресурсы на исправление ошибки, прежде чем она затронет наших пользователей. Например, предположим, что эксперимент выявил критический недостаток в логике обработки ошибок конкретной службы. В первую очередь мы должны устранить найденную ошибку, и только потом проводить эксперимент со службами верхнего уровня! Если мы не устраним текущую ошибку, то вряд ли узнаем что-то полезное из нового эксперимента.

Опять же, обратная сторона здесь еще более интересна: если мы знаем, что эксперимент не вызовет отказ, мы также не должны его проводить. Например, возьмем службу X, которую мы определили, как мягкую зависимость, то есть все службы, которые к ней обращаются, продолжают нормально функционировать, даже когда эта служба не работает. Теперь рассмотрим службу Y, от которой зависит X (но не другая служба). Любые сбои, которые мы могли бы внести в Y, в худшем случае приведут к выходу из строя службы X, которая, как мы уже знаем, не нарушает работу системы! Следовательно, незачем исследовать эту область пространства экспериментов.

Этот второй пример говорит нам, что помимо выбора экспериментов не менее важная задача эксперта – решить, *в каком порядке* их запускать (в этом случае наше решение тщательно поэкспериментировать на X, прежде чем перейти к Y, позволило нам обойтись без большого числа ненужных экспериментов). Однако выбор правильного порядка требует от нас оптимизации различных целей, которые могут противоречить друг другу. Во-первых, как я уже говорил ранее, мы должны изучить семантику исследуемой системы, чтобы понять, какие эксперименты дают новые знания, позволяющие нам пропускать будущие эксперименты. То есть мы должны использовать имеющиеся знания о системе, чтобы выбирать эксперименты, которые быстрее всего дают новые знания. Во-вторых, поскольку мы хотим выявить ошибки до того, как это сделают наши пользователи, мы также хотим провести эксперименты, которые соответствуют *вероятным событиям* (например, одновременный отказ реплики и сетевого раздела между двумя оставшимися репликами), прежде чем исследовать редкие события (например, одновременная потеря трех отдельных центров обработки данных).

Если подвести краткий промежуточный итог, эксперт по подбору экспериментов должен:

- начать с семантического знания тестируемой системы (например, топологии графа вызовов);
- использовать имеющиеся знания, чтобы избежать лишних экспериментов, результаты которых можно предсказать;
- среди интересных экспериментов выбрать такие, которые лучше всего увеличивают знания, одновременно исследуя вероятные неисправности в первую очередь.

Этот процесс является итеративным: новые знания определяют будущий выбор и порядок экспериментов.

Но откуда приходит знание? Выполняя свою работу, эксперт требует, чтобы исследуемая система раскрывала информацию о себе во время экспериментов, а также в стабильном состоянии. То есть им требуется какая-то *наблюдае-*

мость системы – чем больше, тем лучше. Без наблюдаемости не может быть ни новых знаний, ни уточнения пространства экспериментов. Мы вернемся к этой теме позже.

Человеческий разум может быть невероятно хорош в умелом преодолении огромного пространства возможностей, умудряясь одновременно стремиться к разным, иногда противоречивым целям. Хорошие хаос-инженеры действительно заслуживают уважения. Однако можем ли мы точно описать процесс, при помощи которого становятся хорошими экспертами? Если мы этого не можем, как мы собираемся обучать новых специалистов?

Интуиция и коммуникация

Как вы, вероятно, поняли из предыдущих глав этой книги, продолжаются дебаты относительно правильной роли автоматизации в хаос-инжиниринге и надлежащего баланса между автоматизацией и человеческим опытом. Я думаю, все согласятся, что нам не нужна ни полностью человекозависимая, ни полностью автоматизированная система. Вместо этого мы хотели бы автоматизировать все отвлекающие действия, чтобы люди могли спокойно делать то, что они делают лучше всего. Я надеюсь, что с этим утверждением никто не станет спорить.

В чем именно люди уникальны? Оказывается, на это сложно ответить простыми словами, по крайней мере для меня. Иногда может показаться, что наше призвание – это синтез, создание совершенно новой идеи путем объединения существующих идей. Но с каждым годом компьютеры все больше удивляют нас тем, насколько они хороши в синтезе. А как насчет абстракции – обобщения от конкретных примеров к понятиям? Это, безусловно, одна из тех вещей, в которых люди великолепны. Но и здесь машины сулят большие перспективы. Возможно, лучший ответ – это сотворение *ex nihilo*: придумать совершенно новую идею, которая не состоит из сочетания существующих, более мелких идей. Здесь компьютерам далеко до людей, и это не должно удивлять, поскольку мы сами не понимаем, как мы это делаем. У нас мало шансов научить компьютер тому, чего мы не понимаем.

Заманчиво думать, что интуиция – способность «мыслить душой» без четких рассуждений – это исключительное свойство людей. Люди обладают удивительной способностью быстро устанавливать связи на основе наблюдений и впоследствии использовать эти связи для эффективного принятия решений. К сожалению, люди плохо объясняют свою интуицию другим. Когда мы принимаем решение, основанное на интуиции, то часто не можем вспомнить наблюдения, на которых основано это решение (как бы это ни было полезно!). Из-за этого рискованно полагаться на человеческую интуицию в качестве ключевой части бизнес-процесса. Если инженер по безопасной эксплуатации (SRE) после беглого взгляда на приборную панель сужает свое расследование до конкретной подозрительной службы, мы, конечно, можем спросить его, почему этот набор сигналов привел к такому действию. Но только сможет ли он ответить? Возможно, именно поэтому SRE так эффективно учатся на новых инцидентах, но хуже обучаются на основе накопленных знаний прошлых SRE.

В публикации 2015 года в техническом блоге Netflix[1] Кейси Розенталь (соавтор этой книги) и его коллеги описывают желание создать «целостное понимание» поведения сложной системы, используя мысленный эксперимент, называемый «болевой костюм». Этот костюм преобразует системные предупреждения и другие сигналы, выдаваемые инфраструктурой мониторинга, в болезненные ощущения на коже пользователя.

Они пишут:

> Теперь представьте, что вы носите болевой костюм в течение нескольких дней. Вы просыпаетесь однажды утром и чувствуете боль в плече. «Ну конечно, – думаете вы, – микросервис X снова барахлит». Вам не понадобится много времени, чтобы сформировать интуитивное представление о целостном состоянии системы. Очень быстро у вас будет интуитивное понимание всего сервиса, без каких-либо числовых фактов о каких-либо событиях или явных предупреждений.

Я полагаю, что болевой костюм, если бы он существовал, был бы действительно таким эффективным, как описывают авторы поста. Более того, я думаю, что здесь уместна аналогия с современной методикой обучения SRE в режиме реального времени. Будущих опытных экспертов-экспериментаторов также обучают, помещая их перед огромным набором разнообразных сигналов и показывая, например, как переходить от сигнала к потенциальной причине. Однако я твердо уверен, что это катастрофически неверное решение проблемы выбора эксперимента, и надеюсь вас в этом убедить.

Как бы могущественна ни была человеческая интуиция, как правило, она противоречит *объяснимости*. Решения, которые мы принимаем исходя из своей интуиции, не являются рациональными в том смысле, что, хотя они могут основываться на наблюдениях, мы утратили способность их объяснять. Объяснения очень важны: они делают обоснование наших решений доступным для *коммуникации*. Если мы сможем объяснить решение, мы можем научить кого-то другого принимать аналогичное решение при сходных обстоятельствах, не зацикливаясь на деталях конкретного примера. Если мы сможем объяснить решение, мы можем закодировать логику в компьютерную программу и автоматизировать некоторые этапы принятия решения. Общая цель хаос-инжиниринга – расширить наши знания о системе, а знания подразумевают коммуникабельность. В некотором важном смысле SRE, натренированные в болевом костюме или в его реальном эквиваленте, не обладают знаниями – у них просто есть реакции или инстинкт.

Я рискну предположить, что одна из вещей (возможно, самая важная вещь), в которых люди уникально хороши, – это предоставление и интерпретация объяснений. Если нам достаточно тела, на которое можно надеть болевой костюм, если это тело может обучиться на примерах, но не может обучить других, то не проще ли надеть болевой костюм на компьютер? Это звучит как работа для глубокой нейронной сети, а не опытного SRE с хорошей зарплатой.

---

[1] *Casey Rosenthal et al.* Flux: A New Approach to System Intuition // The Netflix Tech Blog, Oct. 1, 2015, https://oreil.ly/dp_-3.

12.2. Наблюдаемость системы

Проведение хаос-экспериментов в соответствии с принципами, отстаиваемыми в этой книге, предусматривает наличие не только инфраструктуры внедрения отказов, но и инфраструктуры наблюдаемости. В конце концов, какой смысл поджигать что-то, если вы не видите, как оно горит? Во многих случаях те же механизмы, которые выявляют отклонения тестовой группы от устойчивого состояния, могут применяться для объяснения успешного выполнения отдельных запросов в тестовой группе, несмотря на отказы, введенные в эксперименте.

Например, трассировка графа вызовов объясняет отдельные срабатывания распределенной системы, раскрывая причинно-следственную связь между компонентами. Сегодня большинство инфраструктур распределенной трассировки на уровне запросов в значительной степени основаны на Google Dapper[1], опубликованном в 2010 году. К 2012 году разработчики Twitter внедрили версию с открытым исходным кодом под названием Zipkin; к 2015 году клоны Zipkin распространились по многим микросервисным системам (например, Salp в Netflix, Jaeger в Uber). Сегодня OpenTracing предлагает открытый стандарт для трассировки графа вызовов.

Инженеры по эксплуатации и разработчики обычно недооценивают возможность использования трассировки графов вызовов. Наиболее часто встречаются следующие варианты применения:

- графы вызовов используются для диагностики того, *что пошло не так* при аномалиях в работе системы, – например, чтобы понять, почему конкретный запрос провален или занял намного больше времени, чем среднее время выполнения. В этом нет ничего удивительного: причина, по которой был разработан Dapper, заключалась в диагностике проблем с задержкой при массовом разветвлении поиска в Google, задача, которую не удалось полностью решить при помощи инфраструктуры мониторинга;
- графы вызовов используются конечными пользователями (проблема взаимодействия человека с компьютером) и, следовательно, выходят за рамки автоматизации. В последнее время много усилий направлено на улучшение визуализации графов вызовов, но мало сделано для их массового анализа.

Эти варианты использования, однако, почти не касаются потенциальной области применения богатых информацией сигналов трассировки. Не мешает подумать о пользе обучения SRE тому, *что работает правильно*, а не диагностике того, что пошло не так[2]. Чтобы сформулировать проблемы, определенные в разделе 12.1 (а именно выбрать, какие эксперименты проводить и в каком порядке их запускать), нам необходимо понять, как выглядит устойчивое состояние тестируемой системы. Хотя мы склонны считать, что

---

[1] *Benjamin H. Sigelman, et al.* Dapper, a Large-Scale Distributed Systems Tracing Infrastructure // Technical Report (Google), 2010.

[2] В технологии устойчивых систем эта концепция известна под названием Safety-II; см. пояснения Эрика Холландера: https://oreil.ly/sw5KR.

устойчивое состояние распределенной системы наилучшим образом представлено массивом совокупных показателей работоспособности, с вершины этой горы не открывается весь обзор. Понимание устойчивого состояния системы также включает в себя знание того, как выглядят отдельные успешные взаимодействия (например, индивидуальные взаимодействия запрос/ответ). Как я уже говорил, выбор правильных экспериментов требует семантического понимания тестируемой системы – в частности, понимания того, как она выходит из строя в ответ на сбои, которые должна была перенести. Отсюда следует понимание того, как система продолжает функционировать, несмотря на сбои. Особый интерес для SRE представляют графы вызовов этих действий, помогающие выяснить, как система допустила конкретную ошибку или комбинацию ошибок.

12.2.1. Наблюдаемость и интуиция

Хотя на первый взгляд наблюдаемость нужна для помощи в расследовании, когда что-то идет не так, мы видим, что в нашем случае ее можно использовать для помощи экспертам в построении модели исследуемой системы. Возникает соблазн спросить – особенно учитывая вопрос, который я задавал выше («В чем именно люди уникальны?»), – можем ли мы использовать наблюдаемость для обучения компьютеров навыкам выбора эксперимента. «Болевой костюм» не является истинно автоматическим: он лишь автоматизирует представление информации человеку-эксперту, чья работа заключается в том, чтобы создать (а затем и применить) интуицию. Давайте подумаем о том, как автоматизировать этот процесс, чтобы драгоценные человеческие ресурсы остались для тех задач, которые мы никогда не смогли бы доверить компьютеру.

Внедрение связанных сбоев

Хотя это далеко не единственный путь к цели, исследования Disorderly Labs по *внедрению отказов линейным методом* (lineage-driven fault injection, LDFI) показывают, что при наличии достаточно творческого подхода интуитивный выбор экспериментов можно автоматизировать от начала и до конца. В оставшейся части этой главы я расскажу о том, как LDFI играет роль эксперта по безопасной эксплуатации.

Первая ключевая идея, лежащая в основе LDFI, заключается в том, что трассировки часто показывают, как распределенная система компенсирует сбои за счет *избыточности* в различных формах. С моей точки зрения, отказоустойчивость и избыточность – это одно и то же: система отказоустойчива *именно в том случае*, если она обладает достаточными ресурсами или способами для успешного выполнения распределенных вычислений, чтобы частичный сбой некоторых компонентов не нарушал работу системы! Эта избыточность существует в различных измерениях: репликация, откаты, повторные попытки, контрольные точки и резервные копии, а также журналы с упреждающей записью. Вторая ключевая идея заключается в том, что эта избыточность очень часто проявляется в *структуре* объяснений, предо-

ставляемых трассировкой системы. Например, тайм-аут и аварийное переключение на резервный экземпляр службы без сохранения состояния или откат к альтернативной службе раскрывают новую «ветвь» в объяснении. Точно так же, когда мягкие зависимости терпят сбой, граф вызовов дает нам доказательство того, что этот подграф не так уж необходим для успешной работы. Со временем история успешных графов вызовов может выявить множество *альтернативных* вычислений, которые по отдельности достаточны для успешного выполнения, а также *критические* вычисления, общие для всех успешных выполнений. Как правило, первые дают нам подсказки об экспериментах, которые можно пропустить, а вторые говорят о том, какие эксперименты должны быть приоритетными.

Все, что остается сделать дальше, – только моделирование и проектирование. Мы выбрали довольно упрощенный способ моделирования информации, содержащейся в системных трассировках: в виде булевых формул. Коллекция трассировок успешных выполнений относительно просто преобразуется в булеву формулу; как только мы это сделаем, проблема выбора «хорошего эксперимента» (такого, который может выявить либо ошибку, либо избыточность, которая у нас пока что не смоделирована) может быть представлена как проблема выбора эффективного, готового к использованию решения для задачи логической выполнимости булевых формул (Boolean satisfiability, SAT). Точно так же, чтобы расставить наши эксперименты по приоритету, мы можем закодировать информацию о ранжировании на основе топологии графов или вероятности отказа определенного сервиса либо оборудования. Используя это ранжирование, мы можем представить ту же самую логическую формулу решателю целочисленного линейного программирования (integer linear programming, ILP) и попросить его найти *лучшее* решение (эксперимент), которое максимизирует ранжирование. Путем точной настройки функции ранжирования мы можем закодировать эвристику, например «сначала исследовать наиболее вероятные отказы, но среди одинаково вероятных отказов предпочесть те эксперименты, которые (на основе топологической информации), скорее всего, исключат большинство других экспериментов». Чтобы LDFI работал с различными участниками информационной отрасли (включая Netflix, Huawei и eBay), пришлось потрудиться над интеграцией. Например, извлечение релевантной информации из конкретного развертывания трассировки никогда не бывает одинаковым заданием два раза подряд, равно как не бывают одинаковыми любые две пользовательские инфраструктуры для внедрения отказов. Тем не менее процесс «склеивания» этих частей вместе можно рассматривать как создание набора абстрактных отображений. Мы описали это в недавней статье[1].

Булевы формулы служат лишь одним из возможных способов построения наших моделей, и можно легко представить себе другие варианты. Возможно, исходя из принципиально неопределенной природы распределенных систем, наиболее подходящими окажутся вероятностные модели или модели машин-

[1] *Peter Alvaro, et al.* Automating Failure Testing Research at Internet Scale. Proceedings of the 7th Annual Symposium on Cloud Computing (SoCC 2016), Santa Clara, CA: ACM (October 2016).

ного обучения, такие как глубокие нейронные сети. Это выходит за рамки моей собственной компетенции, но я бы очень хотел найти сотрудников и нанять аспирантов, которые заинтересованы в работе над этими проблемами!

В этом разделе я не ставил перед собой задачу рекламировать подход LDFI как таковой, а лишь привел пример, показывающий, что на практике возможна некая сквозная «автоматизация интуиции», целесообразность которой я отстаивал во врезке в основной текст. Надеюсь, я убедил вас, что если ваша организация уже имеет зрелую культуру и инфраструктуру SRE, то все готово для автоматизации интуиции. Вместо того чтобы обучать экспертов SRE навыкам выбора экспериментов, попробуйте выделить время для автоматизации процесса выбора. Затем некоторое время автоматический процесс может работать самостоятельно, и когда он выявляет ошибки, этих экспертов можно привлечь для работы над проблемой, которая выглядит (по крайней мере, на первый взгляд)[1] более сложной для автоматизации: выслеживание и исправление ошибок.

12.3. Вывод

Современное состояние интуитивного инжиниринга оставляет желать лучшего. Нам нужна «автоматизация интуиции» – обучение компьютеров, чтобы заменить экспертов в выборе хаос-экспериментов. Хотя, конечно, можно обучать опытных экспертов по выбору экспериментов, на основе постоянно улучшающейся инфраструктуры наблюдаемости нашего сообщества, это дорого, ненадежно и плохо повторимо. Если мы можем научить машины выполнять эту работу хотя бы так же хорошо, как человеческие эксперты, мы должны начать делать это поскорее, чтобы сохранить драгоценные человеческие ресурсы для чисто человеческих задач (например, реагировать на новые явления и давать объяснения этим реакциям).

Об авторе

Питер Альваро – доцент кафедры информатики в Калифорнийском университете в Санта-Крус, где он возглавляет исследовательскую группу Disorderly Labs (disorderly-labs.github.io). Его исследования сосредоточены на использовании языков, ориентированных на данные, и методов анализа для построения и объяснения распределенных систем, интенсивно использующих данные, чтобы сделать их масштабируемыми, предсказуемыми и устойчивыми к сбоям и недетерминизму, характерным для крупномасштабных проектов. Питер получил степень доктора философии в университете Беркли, где он учился у Джозефа М. Хеллерстайна. Он является лауреатом премий NSF CAREER Award и Facebook Research Award.

[1] *Lennart Oldenburg et al.* Fixed It For You: Protocol Repair Using Lineage Graphs. Proceedings of the 9th biennial Conference on Innovative Data Systems Research (CIDR 2019), Asilomar, CA, 2019.

Часть IV

ФАКТОРЫ БИЗНЕСА

Хаос-инжиниринг предназначен для решения реальных бизнес-задач. Он родился в Netflix и в настоящее время прижился в тысячах компаний, большая часть которых не является специализированными разработчиками программного обеспечения. Эта часть книги более обстоятельно рассказывает о том, как хаос-инжиниринг вписывается в более широкий контекст проблем бизнеса.

Глава 13 «Рентабельность хаос-инжиниринга» посвящена наиболее важному практическому вопросу с точки зрения бизнеса, а именно: как мы докажем, что внедрение хаос-инжиниринга выгоднее, чем его отсутствие? «Демонстрировать рентабельность инвестиций в хаос-инжиниринг нелегко. В большинстве случаев вы почувствуете ценность экспериментов почти сразу, раньше, чем успеете сформулировать доказательство». В этой главе представлена модель практического применения ROI.

Расс Майлз в главе 14 «Открытые умы, открытая наука и открытый хаос» рассматривает деловые соображения с другой точки зрения, подчеркивая взаимосвязь между бизнесом и научными задачами. «Как и любая наука, хаос-инжиниринг наиболее продуктивен в атмосфере тесного сотрудничества: где каждый может увидеть, какие эксперименты проводятся, когда они происходят и какие результаты получаются». Он доказывает, что для извлечения максимальной пользы из хаос-инжиниринга нужны инструменты с открытым исходным кодом, открытые эксперименты и сообщество единомышленников.

Один из наиболее частых вопросов у организаций, внедряющих хаос-инжиниринг, – с чего начать или как действовать. Глава 15 «Модель зрелости хаоса» предоставляет методику оценки существующей программы хаос-инжиниринга. Наряду с методикой оценки в этой главе предложены пути совершенствования и распространения технологии: «Высокотехнологичный, глубоко проработанный и широко распространенный хаос-инжиниринг является лучшим *проактивным* методом для повышения доступности и безопасности в информационной индустрии».

Независимо от того, планируете ли вы внедрение хаос-инжиниринга в своей организации или хотите улучшить существующие навыки, вам будет полезно узнать о том, как объединить научный дух хаос-инжиниринга с прагматизмом бизнеса.

Глава 13

Рентабельность хаос-инжиниринга

«Никто не рассказывает историю инцидента, который не произошел».

– Джон Алспоу

Хаос-инжиниринг – это прагматичная дисциплина, призванная приносить пользу бизнесу. Одной из главных трудностей успешного применения хаос-инжиниринга на предприятии является поиск доказательств ценности для бизнеса. В этой главе перечисляются трудности, связанные с представлением осязаемой ценности для бизнеса, описывается модель методического достижения *рентабельности инвестиций* (return of investing, ROI), называемая *моделью Киркпатрика*, и дается объективный пример определения ROI, основанного на опыте Netflix с их платформой автоматизации хаоса под названием ChAP.

13.1. Краткосрочный эффект хаос-инжиниринга

Представьте, что вы постоянно измеряете время безотказной работы вашего сервиса и обнаруживаете, что у вас есть две девятки[1] времени безотказной работы. Вы внедряете методику хаос-инжиниринга, и через какое-то время система демонстрирует три девятки безотказной работы. Как вы докажете, что это заслуга хаос-инжиниринга, а не случайное отклонение? Это очень сложная проблема.

---

[1] «Девятки» – это обычный способ измерения времени безотказной работы. Две девятки соответствуют услуге, работающей в течение 99 % времени, три девятки соответствуют 99,9 % времени и т. д. Чем больше девяток, тем лучше. Пять девяток соответствуют менее чем 5,5 минуты простоя в год. Обратите внимание, что это небольшое количество чрезвычайно трудно измерить, и любое количество девяток выше этого значения не имеет смысла для большинства систем, даже если оно измеряется регулярно в течение длительного времени.

Есть еще одно препятствие: наиболее очевидные положительные эффекты от применения хаос-инжиниринга краткосрочны по своей природе. Вместо того чтобы создавать долговременные преимущества для безопасности системы, хаос-инжиниринг предпочитает открывать врата для других проблем бизнеса. Если хаос-инжиниринг улучшит доступность сервиса, велика вероятность, что бизнес отреагирует, выпустив более быстрые функции. Это, в свою очередь, поднимет планку сложности системы, командам станет труднее поддерживать достигнутый уровень доступности, поэтому потребуются новые улучшения.

Так что выгода от успешного хаос-инжиниринга может быть невидимой. И даже если положительный эффект заметен, он быстро будет сглажен внешним противодействием. На первый взгляд это может показаться безвыходной ситуацией, но не теряйте надежду. Необходимо приложить дополнительные усилия для явного определения значимости. Иногда ценность может быть напрямую связана с бизнес-результатом; иногда это невозможно. Зачастую это зависит от методики измерений и количества усилий, которые нужно приложить для измерения отдачи.

13.2. Модель Киркпатрика

Одним из способов оценки ROI является *модель Киркпатрика*. Она существует с 1950-х годов. Наиболее распространенная итерация модели может быть разбита на следующие четыре уровня:

- *уровень 1*: реакция;
- *уровень 2*: обучение;
- *уровень 3*: перенос;
- *уровень 4*: результаты.

Это прогрессия, где *уровень 1* является относительно простым и малозначащим, тогда как *уровень 4* часто сложен для реализации, но имеет высокую ценность.

Модель используется в академических и корпоративных образовательных учреждениях для оценки эффективности программы обучения или тренинга. Поскольку хаос-инжиниринг рассказывает людям об их собственных системах, мы можем считать его образовательным мероприятием. Следовательно, модель Киркпатрика в основном подходит и для оценки эффективности хаос-инжиниринга.

13.2.1. Уровень 1: реакция

Мы начинаем работу с моделью с очень простой оценки: как ученики отреагировали на тренинг? Мы можем оценить ответ слушателей с помощью анкет, интервью или даже неформальной беседы. В контексте хаос-инжиниринга наши ученики – это владельцы и операторы системы, которым мы хотим рассказать о системных уязвимостях. Они являются стажерами, а так-

же заинтересованы в безопасности системы. Мы хотим, чтобы эти заинтересованные стороны оценили полезность экспериментов с их личной точки зрения. Удалось ли им узнать что-то новое из экспериментов? Понравились ли им упражнения в игровой день? Что они думают о текущей реализации программы хаос-инжиниринга? Будут ли они рекомендовать продолжить или расширить программу? Если ответы в целом положительные, мы считаем положительной оценку *уровня 1* в модели Киркпатрика, и это является минимальной демонстрацией рентабельности хаос-инжиниринга. Если ответы в целом отрицательные, то модель Киркпатрика говорит нам о том, что программа, вероятно, неэффективна, и программа хаос-инжиниринга должна быть прекращена или значительно изменена. Обучение редко идет на пользу, когда заинтересованные стороны не видят смысла в программе обучения.

13.2.2. Уровень 2: обучение

Если заинтересованные стороны ощущают полезность программы, то все хорошо и можно идти дальше. Следующий уровень в модели оценивает доказательства того, что участники чему-то научились. В течение игрового дня составлением оценки может заниматься непосредственно фасилитатор. Он может записывать открытия, например «Команда узнала, что производство имеет непредвиденную зависимость от кластера Kafka». Или «Команда узнала, что отказ двух сервисов уровня 2 приведет к отключению сервиса уровня 1 и последующему катастрофическому отказу системы». (Надеемся, что подобные предметы изучаются с должной минимизацией радиуса поражения.) Труднее оценивать эксперименты, не связанные с игровыми днями. Если у вас существует форма для записи опровергнутых гипотез, это обеспечит отправную точку для перечисления новых знаний, предоставленных программой. Перечень полученных уроков формирует второй уровень ROI для хаос-инжиниринга в соответствии с моделью Киркпатрика.

13.2.3. Уровень 3: перенос

Перечень уроков, полученный на втором уровне, является хорошим началом для определения ROI учебной программы, и еще лучше, если за этим последует его практическая реализация. Третий уровень в модели Киркпатрика оценивает перенос информации в поведение. Если хаос-инжиниринг информирует операторов и заинтересованные стороны о наличии уязвимостей в системе, мы ожидаем увидеть их ответные действия. Мы можем искать изменения в поведении операторов, инженеров, других заинтересованных сторон и даже руководства:

- исправляют ли они уязвимости?
- меняют ли они стратегию использования в отношении своих систем?
- повышают ли они надежность предоставления услуг?
- улучшают ли они адаптационные возможности системы?

- выделяют ли они больше ресурсов для обеспечения безопасности, таких как проверки кода, более безопасные инструменты непрерывной доставки и дополнительные хаос-эксперименты?

Если мы сможем обнаружить изменения в поведении в результате применения методов хаос-инжиниринга, то в соответствии с моделью Киркпатрика у нас есть веские доказательства того, что программа хаос-инжиниринга работает.

13.2.4. Уровень 4: результаты

Это наиболее сложный уровень оценки эффективности программы обучения в модели Киркпатрика. Сопоставление одного конкретного эффекта с бизнес-результатом не может служить корректной оценкой, поскольку на бизнес-результаты одновременно влияет очень много факторов. Независимо от того, является ли целью меньшее время простоя, меньшее количество инцидентов безопасности или меньшее количество времени, проведенного в режиме пониженного обслуживания, у бизнеса всегда есть приоритетная причина для запуска программы. Получает ли организация результат, за который она заплатила? Программа хаос-инжиниринга стоит денег. Можно ли связать сокращение времени простоя, меньшее количество инцидентов безопасности или менее длительное ухудшение качества услуги с реализацией программы хаос-инжиниринга? Если это так, то экономическое обоснование готово, и остается лишь сравнить стоимость программы со стоимостью результата. Это последний и самый важный уровень модели.

Уровни в модели Киркпатрика формируют все более убедительные оценки ROI от *уровня 1* до *уровня 4*. Но чем выше уровень, тем труднее его выполнять. Модель Киркпатрика предполагает, что степень, в которой вам нужно продемонстрировать ROI, будет соответствовать усилиям, которые вы затрачиваете, и, следовательно, будет определять уровень, к которому вы стремитесь. Например, если у вас очень легкая программа и она обходится организации не слишком дорого, то для оценки рентабельности инвестиций можно обойтись первым уровнем. Возможно, будет достаточно отправить вопросник участникам игрового дня, чтобы узнать оценку полезности их опыта. И наоборот, если у вас есть большая команда разработчиков, полностью погруженных в программу, то вам наверняка придется перейти на уровень 4. Возможно, вам потребуется придумать способ классификации инцидентов и показать, что для некоторых классов инцидентов удалось сократить их частоту, продолжительность или последствия.

13.3. Альтернативный вариант оценки рентабельности

Программа хаос-инжиниринга в Netflix описана в разных местах этой книги. В частности, внедрение ChAP (глава 16) было большим капиталовложением

и требовало соответствующей демонстрации ROI. В Netflix и без того постоянно предпринимались усилия по улучшению доступности, поэтому для доказательства эффективности хаос-инжиниринга было недостаточно просто отметить улучшение доступности. Вклад ChAP должен быть изолирован от других усилий.

ChAP выдвигал гипотезы типа: «В условиях X запросы клиентов все равно будут хорошо удовлетворены». Затем он пытался опровергнуть эти гипотезы. Большинство из них не могли быть опровергнуты, что придало больше уверенности в системе. Однако иногда гипотеза опровергалась. Каждая опровергнутая гипотеза служила уроком. Заинтересованные стороны считали, что система во всех случаях будет вести себя одинаково; ChAP показал им, что это не так. В этом и заключалась основная ценность приложения.

Каждая опровергнутая гипотеза соответствовала некоторому набору ключевых показателей эффективности бизнеса (key performance indicators, KPI), которые различались между экспериментальной группой и контрольной группой. В случае с ChAP основным показателем KPI было количество запусков потоковой передачи видео в секунду (starts per second, SPS). Для каждой опровергнутой гипотезы фиксировали влияние на SPS. Это могло быть влияние 5 % или даже 100 % и т. д.

В то же время регистрировали влияние реальных инцидентов на SPS. Например, инцидент, который вызывает снижение SPS на 20 %, может длиться в среднем пятнадцать минут. Используя это распределение вероятностей, можно грубо объединить опровергнутые гипотезы с продолжительностью реальных инцидентов. Суммирование потерянного SPS на определенном интервале времени дает нам предположение о влиянии конкретной уязвимости, если бы она проявилась в производстве.

Используя данный метод для каждой опровергнутой гипотезы, мы могли вычислить гипотетическое влияние на SPS, которое удалось выявить и устранить благодаря ChAP, прежде чем сбои проникли в производство и повлияли на всех клиентов. Суммируя эти «сэкономленные SPS» с течением времени, мы получили график, которая показывал, что этот показатель растет каждый месяц, при условии что опровергнута хотя бы одна гипотеза. Таким образом, мы показали, что ChAP постоянно приносит выгоду бизнесу, и определили ROI для ChAP в рамках программы хаос-инжиниринга в Netflix.

Судя по модели Киркпатрика, эта оценка рентабельности инвестиций ChAP достигла только второго уровня, потому что охватывала лишь прямое обучение. Мы не пытались показать, что специалисты вели себя как-то иначе из-за обнаруженных уязвимостей, и не было найдено никакой корреляции с улучшением доступности. Чтобы найти более веские аргументы в пользу рентабельности инвестиций, нам бы пришлось перейти на третий уровень и отметить изменения в поведении инженеров, а затем каким-то образом соотнести уроки, извлеченные из опровергнутой гипотезы, с улучшением доступностью или другими бизнес-целями на четвертом уровне. Но как показала практика, наш метод определения рентабельности инвестиций через SPS и без того оказался достаточно убедительным.

13.4. Побочная отдача от инвестиций

Рассмотренный выше пример с ChAP демонстрирует ощутимый результат, измеряемый фундаментальной метрикой с точки зрения бизнес-цели. Однако многие уроки, извлекаемые из хаос-инжиниринга, не зависят от основных целей эксперимента. Это явления, обнаруженные в непосредственной близости от хаос-инжиниринга, и часто даже не во время фактической оценки гипотезы. Мы называем это *побочной отдачей от инвестиций* (collateral ROI): дополнительные выгоды, которые отличаются от основной цели, но тем не менее способствуют «возврату» инвестиций.

Побочную отдачу проще всего заметить, если прислушаться к разговорам во время игрового дня. На этапе разработки игрового дня происходит отбор участников. Заинтересованные стороны соответствующей системы определены и собраны вместе. Один из первых вопросов, задаваемых этой группе, должен звучать так: что, по вашему мнению, должно произойти в условиях X?

В этом диалоге раскрываются две побочные выгоды для бизнеса. Наиболее очевидной выгодой является формирование общей точки зрения. Выявляются изолированные очаги знаний: то, что кажется очевидным результатом для одного человека, часто является откровением для другого. Многие игровые дни на этом этапе приходится перестраивать на ходу, потому что гипотезы, которые некоторым фасилитаторам кажутся заслуживающими анализа, с большой вероятностью проваливаются при обсуждении. В подобных случаях нет смысла проводить соответствующий эксперимент. Исходная уязвимость может быть устранена, а затем протестирована в один из следующих игровых дней, а сегодня можно рассмотреть другую гипотезу.

Иное преимущество этого процесса труднее охарактеризовать. Часто люди, собравшиеся на игровой день, ранее не сотрудничали. Иногда они даже не встречались. Это дает возможность выявить совместные островки знаний, а также закладывает основу для взаимодействия людей, работающих над достижением общей цели. Опыт совместной работы позволяет командам лучше работать вместе во время реальных инцидентов.

Представьте, что эти люди впервые собрались вместе во время сложного инцидента в производстве. А теперь представьте, сколько времени сэкономит слаженная команда просто на знакомстве и адаптации к индивидуальным особенностям общения. Это дополнительный источник «побочной рентабельности».

Реальные инциденты часто пугают и сбивают с толку участников. Поскольку поведение людей во многом определяется привычками, практика – это лучший метод подготовки специалистов к отказу системы. По такому же принципу работают пожарные тренировки, отработка поведения во время землетрясения или спортивная тренировка – неоднократное прохождение человека через смоделированную ситуацию помогает ему сохранить спокойствие, когда событие происходит в реальности. Далеко не всегда можно написать исчерпывающие инструкции. Непредсказуемое сочетание инструментов и процесса инцидента создает неопределенный контекст. В этих об-

стоятельствах способность понимать друг друга без слов ценится дороже, чем любая письменная инструкция.

Как и всегда в хаос-инжиниринге, цель здесь не в том, чтобы вызвать хаос или дискомфорт. Реакция на инцидент может быть хаотичной сама по себе. Целью хаос-инжиниринга является создание контролируемой ситуации, в которой люди могут научиться направлять хаос к желаемому результату. В нашем случае это быстрое и надежное исправление инцидента.

13.5. Вывод

Демонстрировать рентабельность инвестиций в хаос-инжиниринг нелегко. В большинстве случаев вы почувствуете ценность экспериментов почти сразу, раньше, чем успеете сформулировать доказательство. Если этого чувства достаточно для обоснования программы, то нет смысла искать более объективную метрику. Если же требуется объективная и обоснованная оценка, воспользуйтесь моделью Киркпатрика, как описано в этой главе. Четыре последовательных уровня представляют собой этапы создания все более убедительных доказательств рентабельности инвестиций соразмерно усилиям по выявлению ценности. Это непростая работа, но зато в ваших руках есть все инструменты, чтобы создать надежный фундамент окупаемости инвестиций в хаос-инжиниринг.

Глава 14

Открытые умы, открытая наука и открытый хаос

Автор главы: **Расс Майлз**

Хаос-инжиниринг – это наука. Вы ищете слабые места, которые обусловлены невероятной сложностью современных, быстро развивающихся систем. Это делается с помощью эмпирических экспериментов, которые вводят контролируемые отказы или сочетания факторов в комбинацию вашей инфраструктуры, платформ, приложений и даже ваших людей, процессов и методов.

Как у любой науки, главная ценность хаос-инжиниринга раскрывается в совместной работе, где каждый может увидеть, какие эксперименты проводятся, когда они проводятся и какие результаты были обнаружены. Но стоит нарушить взаимную открытость, и все это рухнет, как карточный домик.

14.1. Совместное мышление

Представьте себе два образа мышления. Во-первых, люди с инженерным мышлением думают так: «Мне просто нравится находить слабые места. Я горю желанием добраться до какой-нибудь системы и нанести ей вред различными интересными способами, чтобы показать владельцу, почему он должен улучшить свою систему».

Команда с таким мышлением сосредоточена на создании хаоса в системах других людей. Цель все еще состоит в том, чтобы учиться на неудачах, однако она легко теряется в конфликтных отношениях, которые наверняка установятся, когда команда хаос-инженеров начнет проводить эксперименты с системами других людей. Вот в чем ключевая проблема – это делается для *систем других людей*. Сфера полномочий этой команды заканчивается словами «Вот проблема!», и даже если они дойдут до «Вот несколько потенциальных решений», то все равно встретят сильное сопротивление со

стороны команд, которые на самом деле владеют этими системами[1]. Вы слышите ворчание: «Эти хаос-инженеры мешают нам работать...»? С подобным стилем отношений положение дел может быстро ухудшиться до такой степени, что хаос-инжиниринг начнут рассматривать как еще одну проблему, которую нужно обойти, и в конце концов даже могут отвергнуть, как показано на рис. 14.1.

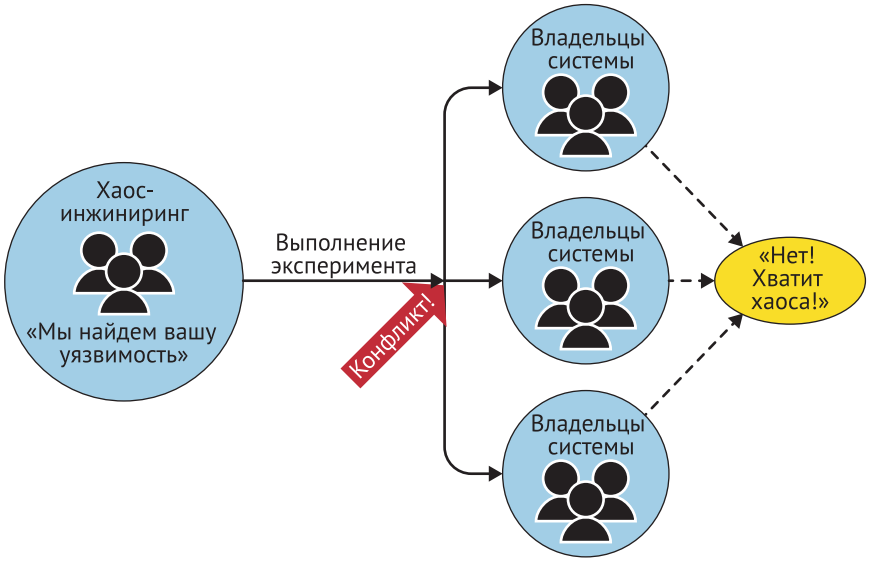

Рис. 14.1 ❖ Токсичные отношения «Хаос-инженеры против нас»

Сравните это со следующим мышлением: «Я горю желанием помочь командам улучшить свои системы. Управлять их системами сложно, и поэтому я очень хочу помочь им найти слабые места в своих системах, чтобы они извлекли уроки из этих открытий и смогли создавать более устойчивые системы, которые сделают их жизнь немного проще».

Во втором примере хаос-инженеры работают *вместе* с командами разработчиков. Они изучают слабые места *своей* собственной системы, а хаос-инжиниринг служит инструментом исследований. Вероятность токсичных отношений «Они против нас» значительно снижается, когда все участвуют в поиске скрытых недостатков и вместе принимают участие в решении проблемы, как показано на рис. 14.2.

---

[1] Именно такого рода конфликтные отношения возникают при разделении на команду качества и команду разработчиков. Отделяя качество и превращая его в «чужую работу», разработчики фокусируются на функциях, а качеством занимаются тестеры. В такой ситуации непременно кто-то или что-то пострадает (как правило, вовлеченные люди, а также качество и функциональность кода). Вот почему обеспечение качества должно быть обязанностью команды разработчиков, поскольку это одна из основных составляющих разработки, а не дополнение к разработке приложений.

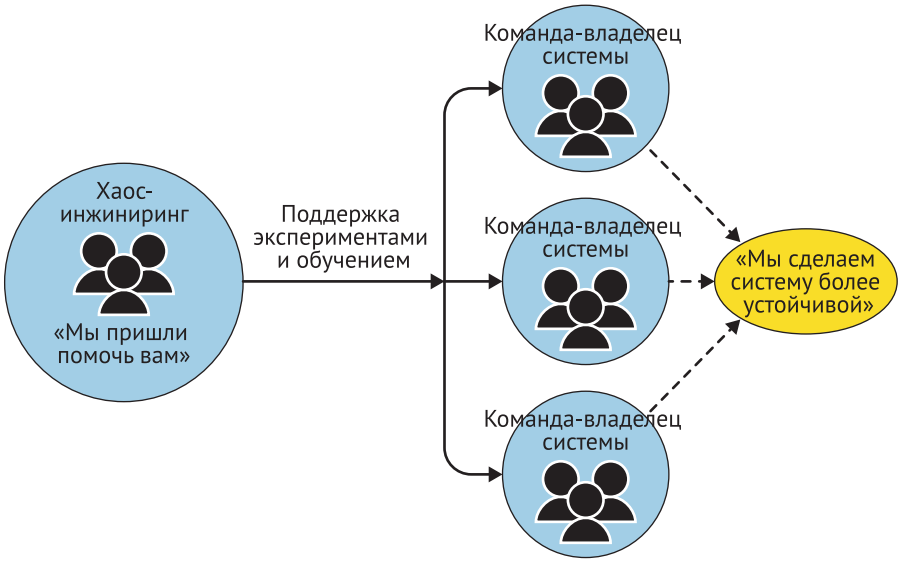

Рис. 14.2 ❖ Продуктивные отношения между специалистами по хаос-инжинирингу и командами, которые вместе берут на себя ответственность за поиск и устранение слабых мест в своих системах

Два совершенно разных подхода, и на первый взгляд разница между ними может показаться надуманной. Какой из этих подходов вы хотели бы использовать в вашей компании? Так уж вышло, что первый может легко провалить внедрение хаос-инжиниринга в вашей организации, а второй приведет к успеху. При внедрении программы хаос-инжиниринга вам не обойтись без сотрудничества и совместного владения системой.

14.2. Открытая наука, открытый исходный код

Сотрудничество выходит за рамки совместного владения[1]. Даже если вам удастся накопить знания о слабых местах системы, от них мало пользы, если информация заперта в проприетарных системах, проприетарных протоколах или хранится в проприетарных форматах. Для продуктивной работы хаос-инжиниринга в организации и процветания этого метода в обществе недостаточно иметь квалифицированных специалистов.

---

[1] Под *владением* (ownership) в индустрии программного обеспечения подразумевают не юридически оформленное право собственности, а *вовлеченность*, то есть участие плюс личную заинтересованность. Иными словами, владельцы системы – это ее персонал и менеджмент, а собственники бизнеса, которому юридически принадлежит система, – это учредители и акционеры. – *Прим. перев.*

Без открытости хаос-инжиниринг лишен возможности реализовать свой истинный потенциал. Вся ценность хаос-инжиниринга заключается в открытом изучении экспериментов, которые выбраны и спланированы разными командами, а также в свободном обмене результатами этих экспериментов внутри команды, отдела, организации и, может быть, даже публично.

Наука испытывает такую же потребность. Борьба с коммерческими и социальными ограничениями имеет долгую историю, и периодически на этом пути встречаются обнадеживающие достижения. Движение Open Science («Открытая наука») «делает научные исследования (включая публикации, данные, физические образцы и программное обеспечение) и их распространение доступными для всех уровней исследовательского сообщества, любителей или профессионалов»[1] и добивается этого с помощью следующих принципов:

- открытые образовательные ресурсы;
- открытый доступ;
- открытая экспертная оценка;
- открытая методология;
- открытые источники;
- открытые данные.

Вы и ваша организация вполне можете следовать этим принципам, извлекая максимальную выгоду из открытости хаос-инжиниринга.

14.2.1. Открытые хаос-эксперименты

Для того чтобы успешно делиться экспериментами между всеми заинтересованными сторонами, необходимо иметь согласованное определение эксперимента. Рабочая группа под названием Open Chaos Initiative, в создании которой я принимал участие, дает следующее определение основных составляющих хаос-эксперимента:

Описание эксперимента
: Эксперимент – это концепция верхнего уровня в хаос-инжиниринге. Эксперимент включает в себя список объектов, гипотезу о стабильном состоянии, метод и (необязательно) откаты к исходному состоянию. Описание может также содержать заголовок, теги для идентификации и любую исходную конфигурацию, полезную для выполнения эксперимента.

Объект эксперимента
: Объект определяет, какая важная система нуждается в проверке и укреплении доверия при помощи хаос-эксперимента.

Гипотеза о стабильном состоянии
: Гипотеза о стабильном состоянии описывает «нормальное состояние» вашей системы, чтобы эксперимент мог выявить аномальные отклонения по сравнению с заявленными «нормальными» допусками измеряемой вели-

[1] *Ruben Vicente-Saez and Clara Martinez-Fuentes.* Open Science Now: A Systematic Literature Review for an Integrated Definition // Journal of Business Research, Vol. 88 (2018), p. 428–436.

чины. Обычно гипотеза ссылается на *зонды*, которые наряду с мерой допуска гарантируют, что параметры системы измеримы в пределах допуска.

Метод
Метод вводит турбулентные условия, которые исследуются экспериментом. Турбулентные условия могут быть заданы в явном виде или определены случайным образом. Условия вводятся в систему при помощи последовательности *действий*.

Зонды
Зонд – это инструмент или способ для наблюдения за определенными параметрами системы, которая подвергается экспериментам.

Действия
Действие – это конкретная операция, которую необходимо выполнить, чтобы воздействовать на экспериментальную систему.

Откаты
Откаты содержат последовательность действий, возвращающих систему к исходному состоянию до запуска эксперимента.

Реализация этих определений зависит от контекста и может сильно различаться, однако в открытом доступе существуют эталонные примеры реализаций[1].

Порядок выполнения эксперимента показан на рис. 14.3.

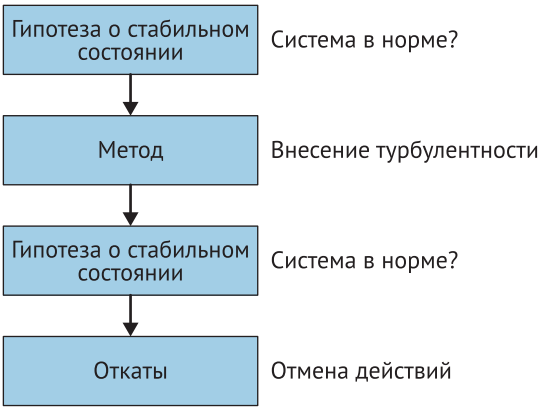

Рис. 14.3 ❖ Порядок выполнения открытого хаос-эксперимента

Примечательная деталь этого процесса выполнения эксперимента состоит в том, что гипотеза о стабильном состоянии используется дважды. Первый раз к ней обращаются в начале эксперимента, чтобы убедиться, что система функционирует «нормально», прежде чем продолжить. Затем гипотезу используют снова, когда применяют турбулентные условия, изучаемые с по-

[1] Практический пример использования этой концепции можно найти в открытом каталоге «Open Chaos Public Experiment Catalog»: https://oreil.ly/dexkU.

мощью эксперимента. Эта вторая оценка системы на основании гипотезы имеет решающее значение, поскольку она показывает, либо что система «пережила» турбулентные условия и поэтому устойчива к ним, либо что «отклонилась» от нормальности, и в этом случае обнаружена потенциальная скрытая уязвимость.

14.2.2. Обмен результатами и выводами

Определение экспериментов для их выполнения и совместного использования – это полдела. Если вы не можете поделиться своими экспериментальными данными, потенциальными доказательствами уязвимостей, значит, полноценное открытое сотрудничество не сложилось.

По этой причине сообщество Open Chaos Initiative также определяет перечень информации, которая должна присутствовать в журнале эксперимента. Ключевые понятия:

- информация о самом эксперименте;
- статус и продолжительность выполнения эксперимента.

Пример реального журнала здесь не приводится, но даже простые записи о ходе эксперимента будут способствовать открытому сотрудничеству, хотя каждый случай зависит от контекста и реализации.

Мы действительно можем делиться информацией и совместно извлекать уроки из недостатков, обнаруженных в различных системах. Хаос-инжиниринг – это хороший пример того, чего можно добиться в открытом сообществе.

14.3. Вывод

Хаос-инжиниринг нуждается в свободе и открытости, чтобы каждый желающий мог исследовать и выявлять слабые места в своих системах. Для этого нужны открытые инструменты и открытые стандарты, и здесь может пригодиться концепция Open Science. С помощью стандартов Open Chaos мы сможем сотрудничать и делиться нашими экспериментами и результатами, возможно, даже через открытые API. Опираясь на эксперименты друг друга, мы можем преодолеть системные уязвимости, прежде чем они нанесут вред нашим пользователям. Надежность, обеспечиваемая хаос-инжинирингом, не должна быть чьей-то исключительной особенностью. Работая вместе, мы все будем сильнее.

Об авторе

Расс Майлз работает генеральным директором ChaosIQ.io, где он и его команда создают продукты и услуги, помогающие их клиентам проверять надежность своих систем. Расс – международный консультант, тренер, докладчик и автор. В его последней книге Learning Chaos Engineering (O'Reilly) рассказывается о том, как сформировать доверие и уверенность в современных сложных системах, применяя хаос-инжиниринг для выявления слабых мест системы до того, как они затронут ваших пользователей.

Глава 15

Модель зрелости хаоса

Когда члены команды Netflix написали первую книгу о хаос-инжиниринге[1], они предложили «Модель зрелости хаоса» (chaos maturity model, CMM). Первоначально это была просто шутка, отсылавшая к популярной в конце 80-х – начале 90-х годов «Модели зрелости возможностей» (capability maturity model, CMM), разработанной в Университете Карнеги-Меллона для анализа процесса разработки программного обеспечения. Эта модель представляла очень жесткий процесс, который резко противоречил свободной культуре в Netflix, где «процесс» считался плохим словом.

Опыт использования модели командой Netflix показал, что она не лишена смысла. Оказывается, в этой шутке велика доля истины. Модель зрелости хаоса действительно приносит пользу, особенно для организаций, которые ищут способы оценить и увеличить свои вложения в хаос-инжиниринг.

Индустрия программного обеспечения в целом недостаточно однородна, чтобы поддерживать отраслевые стандарты, связанные с хаос-инжинирингом. Инфраструктура, культура, ожидания и уровень зрелости слишком различны, чтобы говорить о едином решении, которое обеспечивало бы сопоставимый уровень функциональности в разных компаниях. Вместо отраслевых стандартов в модели зрелости хаоса представлены скользящие шкалы, которые можно использовать для сравнения и оценки различных методов хаос-инжиниринга.

В этой главе мы рассматриваем *модель зрелости хаоса* (CMM) как шаблон. Его можно использовать для составления карты текущего положения команды или организации. Карта наглядно показывает, в каком направлении может совершенствоваться команда, если на примере других команд видит возможности для развития. На карте CMM есть две оси: *внедрение* и *освоение*. Оба аспекта можно исследовать по отдельности.

15.1. Внедрение

Один из наиболее часто задаваемых вопросов о хаос-инжиниринге – как убедить руководство принять концепцию. Уинстон Черчилль сказал: «Никогда не позволяйте хорошему кризису пропасть даром». Это очень хорошо

---

[1] *Ali Basiri et al.* Chaos Engineering. Sebastopol, CA: O'Reilly, 2017.

относится к принятию хаос-инжиниринга руководством. Как мы рассказали во введении, сама дисциплина родилась из кризиса в Netflix. Инструмент Chaos Monkey был изобретен во время отключений, которые затруднили миграцию из центра обработки данных в облако в 2008 году. Chaos Kong был создан после инцидента в канун Рождества в 2012 году.

Это похоже на погоню за машиной скорой помощи с предложением пациенту купить лекарства, но иногда действительно лучшая возможность помочь кому-то – обратиться сразу после того, как он ощутит отсутствие необходимой помощи. Мы знаем несколько случаев, когда руководство неохотно занималось хаос-инжинирингом до тех пор, пока не случался инцидент с доступностью или безопасностью. Сразу после этого резко менялась точка зрения и появлялся бюджет, чтобы исключить повторение подобных происшествий. Зачастую лучший способ представить хаос-инжиниринг руководству – это продемонстрировать его проактивность.

По мере развития дисциплины в целом мы в конечном итоге достигнем точки, когда применение хаос-инжиниринга будут вписывать в дорожную карту еще на этапе создания организации. Но прежде, чем мы дождемся этого, внедрение, как правило, будет основано на традиционной модели.

Внедрение концепции в традиционную организацию состоит из четырех частей:

- появление идеи внедрения;
- определение задействованной части организации;
- соблюдение начальных условий;
- преодоление препятствий.

15.1.1. От кого исходит идея внедрения

Инициаторами внедрения хаос-инжиниринга, скорее всего, будут специалисты, которые непосредственно сталкиваются с последствиями отказа или инцидента безопасности и по очевидным причинам ищут новые инструменты. Кроме них, инициаторами могу быть локальные лидеры, получающие поддержку со стороны команд DevOps, SRE и реагирования на инциденты. В более традиционных организациях инициаторами часто выступают службы эксплуатации или IT-отделы. Это команды, которые понимают, насколько сильно может пострадать организация от инцидента доступности.

Конечно, необходимость быстро вернуть систему в рабочий режим создает препятствия для обучения. По традиции, много усилий направляют на оптимизацию процесса анализа инцидентов или обучения, и лишь немногие организации нашли оптимальный путь к повышению отказоустойчивости: учиться на практике, чтобы развивать адаптационные возможности людей в социотехнической системе.

Вместо этого мы обычно видим постоянный акцент на сокращении времени обнаружения и устранения неполадок. Работа над сокращением этих интервалов – это большое и нужное дело, но это *реактивный* подход (реакция на событие). В конце концов, благодаря множеству успешных примеров и под

давлением разумных аргументов руководство компаний все чаще внедряет *проактивную* (опережающую) стратегию борьбы с инцидентами.

В некоторых продвинутых организациях использование хаос-инжиниринга предписано правилами. Подобное внедрение сверху вниз, как правило, исходит от вице-президента, директора по информационным технологиям или директора по информационной безопасности и служит сильным стимулом для внедрения в тех частях организации, которые отвечают за техническое обеспечение работоспособности.

Так выглядит наиболее распространенный процесс принятия: от тех, кто первыми сталкиваются с инцидентами на переднем крае, до управленцев, а затем новая стратегия становится частью политики организации.

15.1.2. Какая часть организации участвует в хаос-инжиниринге

Независимо от способа внедрения хаос-инжиниринга в организации, еще одним показателем принятия является распространение корпоративной практики. Приверженность организации какой-либо методике отражается в распределении ресурсов.

Хаос-инжиниринг может начинаться с отдельного сотрудника в группе приложений или в централизованной группе эксплуатации. Возможно, поначалу это даже не работа на полный рабочий день, а эпизодические мероприятия, призванные улучшить эксплуатационные свойства конкретной системы. По мере роста популярности хаос-инжиниринг будет использоваться в нескольких приложениях или системах. Это потребует участия большего количества людей.

В конце концов, имеет смысл ввести в штатное расписание отдельную должность хаос-инженера либо создать специальную команду с таким названием. Тогда функция хаос-инжиниринга сможет неявно охватывать всю систему, в противном случае она будет встроена только в одну команду. Выделение ресурсов этой команде сигнализирует остальной организации о том, что проактивное устранение инцидентов является приоритетом, а это довольно высокий уровень принятия.

Поскольку методы хаос-инжиниринга применимы к инфраструктуре, межпрограммным интерфейсам, безопасности и другим уровням социотехнической системы, перед вами открывается множество возможностей для распространения подхода на различные части организации. Окончательное принятие происходит, когда ответственность за применение хаос-инжиниринга возлагается на всех отдельных участников в рамках организации, на каждом уровне иерархии, даже если имеется централизованная команда со своими инструментами. Это похоже на то, как в организации, ориентированной на эксплуатацию, каждая группа несет ответственность за эксплуатационные свойства своего программного обеспечения, даже если централизованная группа разработчиков может внести свой вклад в улучшение некоторых из этих свойств.

15.1.3. Обязательные условия

Для внедрения хаос-инжиниринга требуется меньше начальных условий, чем думает большинство людей. Первый вопрос, который следует задать организации, планирующей использовать эту методику, заключается в том, знают ли они, когда оказываются в опасном состоянии. Не составит труда заметить, что организация отключилась от сети и осталась в автономном режиме. Однако не все организации могут вовремя определить, что упало качество их услуг или они собираются отключиться.

Если организация не способна уверенно различать уровни деградации системы, то любые достижения хаос-инжиниринга будут мимолетными. Если экспериментаторы не могут сравнить контрольную и экспериментальную группы, то ценность любого эксперимента сомнительна с самого начала. Не менее важно, что в таких условиях становится невозможной любая расстановка приоритетов результатов. Без уровней деградации, позволяющих различать влияние экспериментов, совершенно не ясно, указывает ли хаос-эксперимент на что-то важное.

В данном случае противоядием являются мониторинг и наблюдаемость системы. К счастью, решение этой проблемы может привести к неожиданному и значительному улучшению качества инструментов анализа системы и ее окружения.

Совершенствование инструментария служит хорошим поводом больше рассказать о хаос-инжиниринге как о повседневной практике. Не каждый эксперимент должен служить предупреждением. В идеале эксперименты становятся настолько рутинными, что никто не думает о них, пока они не нарушат гипотезу. Эксперименты должны происходить практически без влияния на KPI системы и учить операторов чему-то новому в отношении системы.

При первом внедрении в любую систему или подсистему хаос-инжиниринг нуждается в активном обсуждении. Важно четко пояснить всем заинтересованным сторонам, что делается, с какой целью и каковы ожидаемые результаты. Хаос-инжиниринг часто учит нас новым вещам, которых мы не ожидали, но сама по себе способность удивлять не является достоинством. Удивление заинтересованной стороны с помощью эксперимента только спровоцирует вражду и разногласия.

Другой способ вызвать разногласия – устроить эксперимент, если заранее известно, что результат нежелателен. Хаос-инжиниринг не может быть продуктивным, если гипотеза не проверяется с честным ожиданием того, что она будет подтверждена. Если нельзя сделать заявление типа «эта служба будет соответствовать всем SLO даже в условиях высокой задержки на уровне данных», то не имеет смысла проводить эксперимент с участием этой службы. Если процесс лишь подтверждает, что подозреваемый неисправный компонент действительно неисправен, вы не получите новые знания. Исправьте неполадки, которые вам известны, до того, как в игру вступит хаос-инжиниринг.

Наконец, дисциплина требует, чтобы в организации были согласованы действия в ответ на новую информацию, предоставляемую хаос-инжини-

рингом. Если найдены уязвимости и никто ничего не делает, то это не информация, а бесполезный шум.

Краткое изложение начальных условий:

- инструменты, способные обнаружить ухудшение состояния системы;
- социальная осведомленность;
- ожидания, что гипотеза должна быть подтверждена;
- согласованность ответных действий.

15.1.4. Препятствия для внедрения

Нас часто спрашивают о том, не мешает ли внедрению новой дисциплины слово «хаос» в названии и не отпугивает ли оно руководителей. По нашему опыту, это случается редко, если вообще когда-либо происходит. Люди, кажется, интуитивно понимают, что слово «хаос» относится к раскрытию сложности, уже присущей системе, а не к созданию дополнительного безумия.

Однако существуют и некоторые обоснованные препятствия для внедрения. Основное возражение заключается в том, что бизнес-модель не допускает побочные эффекты экспериментов в производственном трафике. Это веский аргумент с психологической точки зрения, потому что перемены и ощущение риска доставляют людям дискомфорт. Если система не страдает от инцидентов с доступностью или безопасностью, то, возможно, это вполне резонный аргумент. Не нужно связываться с системой, которая никогда не ломается.

Если в системе наблюдались отказы или инциденты безопасности, то было бы неразумно делать вид, что все в порядке. Возможны следующие варианты: (а) продолжать использовать реактивные методы, периодически подвергаться неизбежным инцидентам и повторять процесс устранения или (б) принять упреждающий метод, контролировать потенциальный ущерб, ограничивая радиус поражения, и избегать инцидентов. Последнее решение, несомненно, лучше. Хаос-инжиниринг, как минимум, меняет неуправляемый риск на управляемый.

Хаос-инжиниринг не всегда должен применяться в производственной среде. В сложных случаях хаос-инжиниринг обычно доходит до производства, но это удел продвинутых систем. Многие команды узнали о критических неполадках из первых экспериментов в промежуточной или тестовой среде. Если проблему можно решить более безопасным способом, то лучше так и поступить.

Еще одним потенциальным препятствием является соблюдение нормативов. Устаревшие правила не всегда учитывают преимущества предотвращения неконтролируемых крупных инцидентов за счет введения небольших управляемых рисков. Ситуация с нормами осложняется еще больше, когда средства контроля безопасности тестируются в рабочей среде.

Нередким препятствием является устоявшееся эксплуатационное состояние системы. Люди часто шутят, что они не нуждаются в хаос-инжиниринге, потому что система обладает достаточным количеством собственного хаоса. Если система сама по себе нестабильна, то изучение новых способов соз-

дания нестабильности не выглядит полезным применением ресурсов. Это создает путаницу в отношении приоритетов и может негативно сказаться на моральном духе команды. Если очередь проблем растет быстрее, чем персонал успевает обрабатывать элементы из очереди, то нет причин усугублять эту проблему.

Одним из самых сложных препятствий для внедрения является определение отдачи от инвестиций в программу хаос-инжиниринга (более подробно об этом говорилось в главе 13). Никто не спешит услышать историю об инциденте, который никогда не случался. Точно так же бывает трудно получить одобрение руководства и ресурсы для практики, которая способна значительно улучшить ситуацию, но делает это молча.

Краткий перечень препятствий, о которых мы говорили:

- эксперименты в производстве несут определенный риск;
- соответствие нормативам в некоторых случаях препятствует экспериментам;
- непреодолимая нестабильность существующей системы;
- сложность измерения ROI.

15.1.5. Освоение

Ось измерения сложности практики хаос-инжиниринга в организации аналогична определению места, где она находится на шкале консультативной службы, а не набора инструментов. Первое время после рождения в Netflix команда хаос-инжиниринга существовала в рамках консультативной организации. Руководитель команды Кейси Розенталь принял волевое решение ориентировать команду на разработку инструментов. Это отражено в «Принципах хаос-инжиниринга», таких как автоматизация экспериментов.

И консалтинг, и разработка инструментария могут быть инициированы небольшой централизованной командой. Программная инфраструктура отрасли настолько неоднородна, что не может быть универсального инструмента, который подходит для всех сценариев использования хаос-инжиниринга во всех этих разнородных средах. Вполне естественно, что освоение методики начинается с активного участия человека, а со временем разрабатываются индивидуальные решения.

Процесс освоения часто выглядит следующим образом:

1) игровые дни;
2) консультации по внедрению отказа;
3) свои инструменты внедрения отказов;
4) экспериментальные платформы;
5) автоматизация платформ.

Игровые дни

Игровые дни – отличный способ для организации окунуть свои пальцы в воды хаос-инжиниринга. Их легко организовать с технической точки зрения, и не нужно особо изощряться в плане реализации. Фасилитатор или менеджер проекта могут провести игровой день, выполнив следующие действия:

1) соберите группу людей, которые отвечают за систему или набор систем;
2) отключите компонент, по отношению к которому система должна быть достаточно устойчивой, чтобы продолжить работу без него;
3) запишите результаты эксперимента и верните компонент в рабочий режим.

Преимущества этого типа упражнений могут быть огромными, особенно в первые несколько выполнений. Но бремя ответственности почти полностью ложится на людей. Заинтересованные стороны – специалисты, владеющие системой. Если что-то пойдет не так во время эксперимента, то именно они, по-видимому, могут исправить систему. Координация целиком зависит от фасилитатора: запланировать мероприятие, предварительно продумать процесс, записать полученные знания, распространить эту информацию и т. д. Все это очень ценная деятельность, но она потребляет один из самых ценных ресурсов любой организации – человеческое время – и не масштабируется на большое количество сервисов.

Инструменты внедрения отказа

Если первые игровые дни увенчались успехом, следующим шагом будет создание инструментов, которые можно повторно использовать во всей организации для проведения ручных экспериментов. Такими инструментами часто являются средства внедрения отказа. В идеале инструмент внедрения отказа может создавать помехи на уровне *межпроцессного взаимодействия* (inter-process communication, IPC) системы. Накопление и задержка запросов, прерывание соединений и прекращение передачи сообщений в нисходящий канал – все это отличные примеры экспериментов, которые могут быть построены вокруг манипуляций с IPC.

Способы усложнения экспериментов

Другие способы развития экспериментов включают в себя возврат ошибок, изменение кодов состояния в заголовках ответов, изменение порядка запросов и изменение полезных данных запросов. Использование этих методов не всегда приносит пользу и зависит от назначения системы. Обычный набор факторов сбоя, связанных с доступностью, – это увеличенная задержка, ошибка и отсутствие ответа, что можно рассматривать как предельный случай задержки.

Существует также класс экспериментов, которые не вписываются в область внедрения отказов, что является одной из причин, по которой проводится такое строгое различие между внедрением отказов и хаос-инжинирингом. Отправка большего количества трафика, чем обычно, одному экземпляру в кластере – отличный тому пример. В реальном мире это может быть вызвано недостаточным выделением ресурсов, неудачным хешированием, необычной балансировкой нагрузки, непоследовательной стратегией разделения, неправильно понятой бизнес-логикой или даже внезапным всплеском трафика клиентов. В каждом из этих случаев было бы неправильно называть аномальное увеличение трафика «внедрением отказа».

Как и в случае с игровым днем, консультирующий фасилитатор будет сидеть с командой, запускать систему внедрения отказов, чтобы начать эксперимент, и записывать полученные знания. Если в надежности системы обнаруживаются уязвимости, процесс, как правило, циклически повторяют, испытывая новые решения, пока уязвимость не будет достоверно устранена. Это столь же эффективно, как игровой день, и вдобавок позволяет использовать один инструмент для проведения экспериментов с несколькими командами. Здесь мы видим более развитое кросс-функциональное обучение, поскольку эксперименты, которые оказались плодотворными у одной команды, можно легко воспроизвести с использованием инструмента внедрения отказов в других командах.

Автоматизация внедрения отказов

Добившись успеха с ручным внедрением отказов, можно приступать к процессу автоматизации. Если инструмент внедрения отказов оснащен интерфейсом самообслуживания, то несколько команд могут использовать его одновременно, не прибегая к помощи фасилитатора. На данном этапе обычно продолжаются консультации по настройке экспериментов и интерпретации результатов, но теперь они лучше масштабируются, поскольку фасилитаторы не обязаны сидеть с каждой командой во время эксперимента.

Платформы для экспериментов

Нарастание сложности может идти одним из двух путей: увеличением автоматизации или совершенствованием экспериментов. В большинстве случаев инструмент развивается преимущественно в направлении усложнения экспериментов. Эксперименты проводятся с контрольной группой и подопытной группой. На эти две группы направляют выборочный трафик. Радиус поражения сужают до подопытной группы. Эффект, который можно упустить во время эксперимента на целой системе, часто удается выделить за счет выполнения на небольшой подвыборке, которая напрямую сравнивается с контрольной группой.

На этом этапе команда все еще может консультироваться с фасилитатором по поводу настройки экспериментов и толкования результатов. Мощь экспериментальной платформы позволяет проводить несколько экспериментов одновременно, даже в производственной среде. Количество проводимых экспериментов теперь может увеличиваться в геометрической прогрессии от количества консультантов. Каждый эксперимент должен быть создан и проверен.

Автоматизация платформ

Автоматизация экспериментальной платформы ставит несколько новых технических проблем. Такие функции, как мониторинг KPI, часто содержат автоматический «выключатель робота» (dead robot's switch), поэтому эксперименты, которые обнаружили интересную аномалию, могут быть немедленно прекращены. С одной стороны, это устраняет необходимость контролировать каждый эксперимент силами специалистов, но с другой – требует от них просмотра результатов впоследствии и интерпретации любых аномалий. Система самоанализа позволяет платформе строить эксперименты без

вмешательства фасилитатора; например, когда новый сервис подключается к сети, самоанализ автоматически обнаруживает его и ставит в очередь для экспериментов. Конечно, следует проводить эксперименты только в тех случаях, когда существует измеримое ожидание для гипотезы, например потенциально опасная функция в коде. Это устраняет необходимость создавать каждый эксперимент вручную.

Наконец, ваши эвристические механизмы запрограммированы таким образом, чтобы время и ресурсы были потрачены на эксперименты, которые с наибольшей вероятностью расскажут что-то новое о системе. На этом этапе автоматизация является достаточно полной, чтобы платформа могла работать полностью автоматически: создание, установление приоритетов, выполнение и завершение экспериментов. Мы считаем, что это очень высокий уровень сложности современных инструментов хаос-инжиниринга.

Пример неудачного выбора приоритета эксперимента

Важный момент, о котором следует помнить при создании алгоритма определения приоритетов и проведения экспериментов, состоит в том, чтобы команды четко понимали, что выполняется и как определяется приоритет. Хорошая наблюдаемость помогает командам понять, где их ментальные модели отклоняются от того, что на самом деле происходит в их системе.

Давайте рассмотрим пример. Представьте, что вы работаете в компании, которая транслирует потоковые телепередачи и фильмы. Мы сделаем следующие предположения:

- ключевым показателем эффективности компании является способность удовлетворять запросы на потоковую передачу видео;
- этот показатель измеряется в запусках потока в секунду;
- трафик является достаточно предсказуемым – если количество запусков потока в секунду отклоняется на порядок выше или ниже, чем ожидалось, следует вызов аварийной команды, которая разбирается в ситуации независимо от того, является ли это проблемой.

Представьте, что вы работаете в команде, которая отвечает за эксплуатацию сервиса «Закладки». Этот сервис отвечает за отслеживание текущей позиции видеопотока на тот случай, если человек временно покидает видео и потом хочет вернуться туда, где он остановился. Если бы кто-то спросил вас, влияет ли сервис «Закладки» на количество запусков потока в секунду (то есть считаете ли вы «Закладки» критическим сервисом), вы бы ответили отрицательно. Фактически вы настолько уверены в этом, что реализовали запасной вариант на случай, когда вашему сервису не удается получить правильную позицию потока, где остановился пользователь. Если это не удается, он просто перезапускает поток с начала видео.

Теперь давайте вернемся к алгоритму, который определяет приоритеты экспериментов. Этот алгоритм должен проверить, является ли исследуемый объект «защищенным от сбоя», чтобы правильно спланировать эксперимент. Чтобы упростить пример, давайте предположим, что этот алгоритм просматривает все сервисы, существующие в системе, и эмпирически определяет, есть ли у них запасной вариант. Если не существует запасного варианта, можно предположить, что такой сервис небезопасен.

Итак, у нас есть сервис «Закладки» и алгоритм расстановки приоритетов. Алгоритм определяет, что сервис «Закладки» имеет резерв, поэтому защищен для сбоя и успешно проектирует, а затем запускает эксперимент. Во время эксперимента важно, чтобы его успех сравнивался с KPI в стабильном состоянии, числом запуска потоков в секунду.

Эксперимент автоматически разрабатывается, запускается и заканчивается провалом. Что произошло?

Помните, что у «Закладок» был запасной вариант, который перенаправлял пользователей к началу видео? Оказывается, многих пользователей это не устраивает, и они начинают искать старое место в потоке вручную. Каждая попытка найти нужное место учитывается как запуск потока. В результате число запусков потока в секунду взлетает до небес.

Это прекрасный пример несоответствия ментальной модели, и важно, чтобы команда, разрабатывающая автоматизацию хаоса, учитывала возможную необходимость перестройки ментальной модели. Это может быть как сбор данных сеанса, которые показывают поведение пользователя в ходе эксперимента, так и фиксация места, где владелец сервиса замечает, как изменились его предположения.

Это не означает, что наш путь окончен. Впереди у нас расширение и совершенствование платформы разными способами. Например, вы можете:

- автоматически определять сотрудника, которого надо уведомить, если найдена уязвимость;
- реализовать «бюджет хаоса», чтобы за определенный период времени можно было утратить только определенное количество KPI, гарантируя, что эксперименты никогда не выйдут за рамки допустимого ущерба для бизнеса. Это повысит уверенность руководства в том, что платформа контролируется и фокусируется на выявлении хаоса, а не на его создании;
- построить эксперименты с комбинаторными переменными. Хотя такие эксперименты срабатывают очень редко, потому что большинству компонентов видны только узкие «окна» отказов, они могут выявить комбинаторные уязвимости, вызванные, например, истощением общих ресурсов;
- автоматически устранять уязвимости. Теоретически это возможно. На практике это тема для отдельной книги.

В какой-то момент этого путешествия имеет смысл включить в платформу эксперименты по безопасности, а также традиционные эксперименты по доступности. Хаос-инжиниринг безопасности подробно рассматривается в главе 20.

ПРИМЕЧАНИЕ Есть и отрицательные примеры развития. Наиболее распространенным антипримером является разработка новых способов вывести из строя экземпляр. Независимо от того, сгорел ли блок питания экземпляра, закончилась ли его память, перегружен ли центральный процессор или переполнен диск, это может проявляться только в увеличении времени ожидания, ошибки или отказа в ответе. Как правило, вы не увидите ничего нового, если будете выводить экземпляр из строя новыми изощренными способами. Это общая особенность сложных программных систем, и поэтому таких экспериментов лучше избегать.

Другой способ понять, в чем суть развития сложности хаос-инжиниринга, – рассмотреть уровень, на котором вводится переменный фактор. Как правило, эксперименты начинаются на уровне инфраструктуры. Chaos Monkey классно начал с выключения виртуальных машин. Chaos Kong использовал подобный подход на макроскопическом уровне, отключая целые регионы. По мере того как инструменты становятся все более утонченными, они переходят в логику приложения, влияя на запросы между службами. Кроме того, мы видим еще более сложные эксперименты, когда переменная влияет на бизнес-логику, например предоставляя сервису правдоподобный, но неожиданный ответ. Естественное развитие экспериментов идет по пути *инфраструктура → приложение → бизнес-логика*.

15.2. Карта состояния хаос-инжиниринга

Используя два свойства, принятие и развитие, в качестве ортогональных осей, мы получаем карту состояний (рис. 15.1). Мы начинаем в левом нижнем квадранте этой карты с проведения игровых дней, организованных отдельными SRE или другими заинтересованными людьми. В зависимости от того, какая ось развивается в первую очередь, дисциплина движется в сторону либо повышения сложности, возможно, с помощью системы внедрения отказов, либо более широкого внедрения с выделением новых ресурсов для хаос-инжиниринга. Как правило, внедрение и усложнение происходят одновременно. Существуют естественные ограничения, которые препятствуют чрезмерному усложнению инструментария без широкого применения, а также широкому внедрению без развития инструментов.

Рис. 15.1 ❖ Пример карты состояний

Определение позиции вашей организации на карте поможет вам понять ваш контекст в рамках дисциплины и определить направление для инвестиций, чтобы сделать следующий скачок. Конечная цель всего этого – максимальная отдача для организации. Как видно на карте (рис. 15.1), движение вверх и вправо приводит к состоянию, когда хаос-инжиниринг обеспечивает наибольшую отдачу. Высокотехнологичный и широко внедренный хаос-инжиниринг – лучший проактивный метод повышения доступности и безопасности в индустрии программного обеспечения.

Часть V

ЭВОЛЮЦИЯ

Человек по определению не может понять сложную систему достаточно хорошо, чтобы делать точные прогнозы относительно ее работы. Хаос-инжиниринг, безусловно, обитает в сложной системе взаимодействующих практик, потребностей и бизнес-сред. Тем не менее появились четкие тенденции, которые определяют будущие направления этой методики и ее место в более широкой отрасли. Далее мы расскажем об этих тенденциях.

Эта часть книги начинается с главы 16 «Непрерывная проверка», которая относит хаос-инжиниринг к более широкой категории программных методов. «Подобно CI/CD (непрерывная интеграция / непрерывная доставка), данный метод возник из-за необходимости ориентироваться во все более сложных системах. Организации не имеют времени или других ресурсов для проверки того, что внутренние механизмы системы работают, как предполагалось; вместо этого они проверяют, что выходные данные системы соответствуют ожиданиям». Многие компании уже используют термин «непрерывная проверка» (continuous verification, CV), и интерес к полному набору технологий «CI/CD/CV» постоянно растет, особенно в компаниях, которые эксплуатируют масштабные программные системы.

Глава 17 «Поговорим о киберфизических системах» уводит нас на полшага в сторону от программного обеспечения, в область аппаратных средств с *киберфизическими системами* (cyber-physical system, CPS). «Оказывается, если вы собрали достаточно много опытных специалистов по смежным дисциплинам и довольно долго держите их вместе, чтобы они занимались такой деятельностью, как FMEA, они на самом деле очень неплохо извлекают контекст из собственного опыта и за несколько подходов выжимают из системы пугающее количество неопределенностей». Натан Ашбахер исследует факторы, с которыми CPS должны бороться в ситуациях, когда результат оказывает непосредственное влияние на физический мир вокруг нас и может буквально быть вопросом жизни и смерти.

В главе 18 «Методика HOP и Chaos Monkey» Боб Эдвардс выводит нас из мира программного обеспечения в мир промышленного производства. Концепция *личной и корпоративной эффективности* (human and organizational performance, HOP), направленная на улучшение систем в производстве, имеет много общего с хаос-инжинирингом, и это совпадение поможет нам взглянуть на программирование с другой точки зрения. «Хаос-инжиниринг учит нас изменять параметры и программное обеспечение имитационной модели

таким образом, чтобы симуляция лучше отражала фактическое ухудшение рабочей среды».

Если говорить о привычной работе программистов, то большинство применений хаос-инжиниринга сосредоточены на прикладном уровне. Обзор другого уровня стека приведен в главе 19 «Хаос-инжиниринг и базы данных» Лю Тана и Хао Вэна из компании по производству баз данных PingCap. Эта глава содержит самое глубокое погружение в технологию в данной книге и рассказывает о применении хаос-инжиниринга к базе данных TiDB для повышения ее отказоустойчивости. «В TiDB мы применяем хаос-инжиниринг для наблюдения за устойчивым состоянием нашей системы, выдвигаем гипотезы, проводим эксперименты и сверяем эти гипотезы с реальными результатами».

Последняя глава этой части книги посвящена кибербезопасности. В главе 20 Аарона Райнхарта «Хаос-инжиниринг и кибербезопасность» говорится о применении методики хаос-инжиниринга в сфере кибербезопасности, которая с точки зрения безопасности системы является другой стороной медали доступности. Когда люди обсуждают сегодняшнее состояние хаос-инжиниринга, они в основном говорят о надежности и работоспособности. По мнению Аарона, это может скоро измениться. В ближайшем будущем хаос-инжиниринг может сосредоточиться на безопасности.

Хаос-инжиниринг находит применение в разных отраслях и везде служит источником более глубокого понимания сложных систем. Главы в пятой части книги освещают некоторые из этих применений.

Глава 16

Непрерывная проверка

> Непрерывная проверка – это методика опережающих экспериментов в программном обеспечении, реализуемая как инструмент, который проверяет поведение системы.
>
> *– Кейси Розенталь*

Проблемы, создаваемые сложными системами, стимулировали естественный переход от непрерывной интеграции к непрерывной доставке и непрерывной проверке. Последняя и является темой этой главы, описывающей новое пространство возможностей, а также реальный пример одной из работающих в Netflix систем под названием ChAP. Широта применения непрерывной проверки ничем не ограничена, но области, на которые нужно обратить внимание в первую очередь, рассматриваются в конце этой главы.

16.1. Происхождение непрерывной проверки

Если между двумя или более разработчиками, пишущими код, существует расхождение ожиданий или ментальных моделей, то объединение этого кода может привести к неожиданным и неприятным результатам. Чем быстрее вы это обнаружите, тем больше вероятность того, что в следующем коде будет меньше расхождений. И наоборот, если это расхождение не будет обнаружено на раннем этапе, то следующий написанный код, скорее всего, будет расходиться еще больше, увеличивая вероятность инцидентов и прочих нежелательных явлений.

Один из наиболее эффективных методов выявления этого расхождения в ожиданиях – собрать код и запустить его. Отсюда и появилась *непрерывная интеграция* (continuous integration, CI) как способ достижения этой цели. В настоящее время CI является общепринятой отраслевой нормой. Конвейеры CI предусматривают разработку интеграционных тестов, которые специально проверяют работоспособность объединенных функций кода, написанного отдельными разработчиками или командами.

При каждом редактировании кода, публикуемого в общем хранилище, конвейер CI будет компилировать новое объединение и запускать комплект интеграционных тестов, чтобы подтвердить, что не было внесено никаких критических изменений. Этот цикл обратной связи способствует обратимо-

сти в программном обеспечении: способность быстро развернуть код, передумать и отменить это изменение. Обратимость[1] – это полезная опция при работе со сложными программными системами.

Технология *непрерывной доставки* (continuous delivery, CD) выросла на успехе CI и автоматизирует этапы подготовки кода и его развертывания в среде. Инструменты CD позволяют инженерам выбирать сборку, которая прошла этап CI, и продвигать ее дальше по конвейеру для запуска в производство. Это дает разработчикам дополнительный цикл обратной связи (новый код работает в производстве) и способствует частым развертываниям. Частые развертывания реже выходят из строя, потому что они с большей вероятностью обнаруживают дополнительные расхождения.

Относительно недавно появилась новая технология, основанная на преимуществах CI/CD. *Непрерывная проверка* (continuous verification, CV) – это дисциплина проактивного экспериментирования, реализованная как инструмент, который проверяет поведение системы. Этот подход отличается от предыдущих общепринятых методов обеспечения качества программного обеспечения, которые основаны на реактивном тестировании[2], то есть проверяют[3] заранее известные свойства программного обеспечения. Это не означает, что привычные методики не работают или устарели. Оповещение, тестирование, проверка кода, мониторинг, SRE и т. д. – все это отличные приемы, и их следует поощрять. В свою очередь, CV успешно использует традиционные методы на новый лад для решения уникальных проблем, присущих только сложным системам:

- существует очень мало *проактивных* методов, направленных на оптимизацию системных свойств;
- сложные системы нуждаются в методике, которая предпочитает метод исследования неизвестного (эксперимент) перед известным (тестирование);
- для масштабирования требуются инструменты, в то время как популярные методологии (Agile, DevOps, SRE и т. д.) требуют цифровой трансформации и культурных изменений наряду с дорогостоящими инвестициями в персонал для их реализации;
- бизнес больше нуждается в прагматичной проверке конечного результата, чем в проверке, сфокусированной на правильности программного обеспечения;
- свойства сложных систем находятся в состоянии постоянного изменения и требуют иных подходов, чем известные свойства программного обеспечения, такие как выходные ограничения.

CV не ставит своей целью разработку новой парадигмы в разработке программного обеспечения. Скорее, это название случившегося естественным образом слияния методов разработки и методов проверки, которые имеют много общего. Мы лишь констатируем факт, что появилась новая методика,

---

[1] См. раздел 2.2, в котором рассказано о преимуществах обратимости в разработке программного обеспечения.

[2] См. раздел 3.1.1.

[3] См. раздел 3.1.2.

которая существенно отличается от того, как мы обычно думали о разработке и эксплуатации программного обеспечения до этого момента (рис. 16.1).

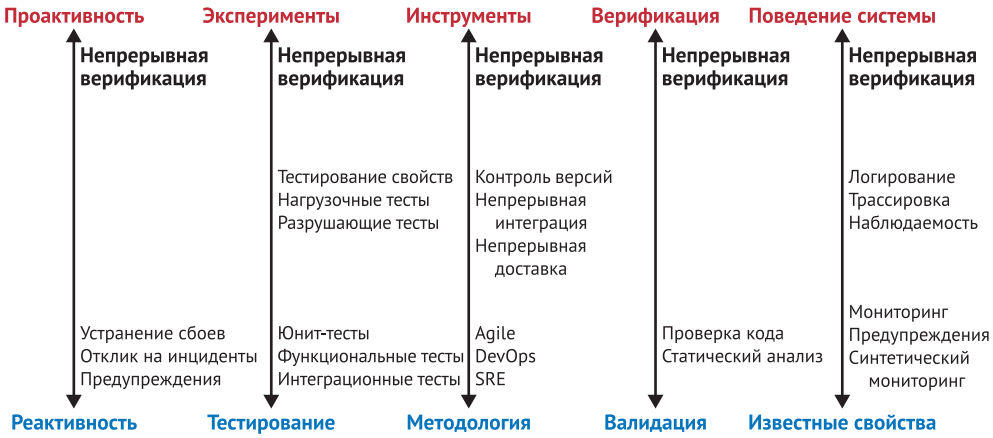

Рис. 16.1 ❖ Непрерывная проверка (вверху) в сравнении с обычными методиками обеспечения качества программного обеспечения (внизу)

Подобно хаос-инжинирингу, CV-платформы могут включать компоненты доступности или безопасности (см. главу 20) и часто выражают их как гипотезы. Подобно CI/CD, новая методика происходит из необходимости ориентироваться во все более сложных системах. Организации не имеют времени или других ресурсов для проверки того, что внутренние механизмы системы работают, как предполагалось; вместо этого они проверяют, что выходные данные системы соответствуют ожиданиям. Это приоритет *верификации* (проверка конечного результата) над *валидацией* (проверка правильности механизма), характерный для успешного управления сложными системами.

16.2. Разновидности систем непрерывной проверки

Инструменты в этой категории все еще зарождаются. На одном конце спектра у нас есть сложные платформы автоматизации хаос-инжиниринга. Они проводят прямые эксперименты, обычно в рабочее время. Конкретный пример ChAP обсуждается в следующем разделе.

Автоматизированные канареечные развертывания также попадают в эту категорию и могут рассматриваться как подкатегория автоматизированных платформ хаос-инжиниринга. Автоматизированные канареечные развертывания проводят эксперимент, в котором переменный фактор представляет собой новую ветвь кода, а контрольный объект – развернутую в данный момент статичную ветвь. Гипотеза заключается в следующем: клиенты получат

хороший опыт даже в условиях нового кода. Если гипотеза не опровергнута, CD автоматически доставляет новый код в развертывание, чтобы заменить текущий код, работающий в производстве.

На другом конце спектра CV у нас есть инструменты, которые обеспечивают целостное представление о системе. Некоторые из них относятся к необычной дисциплине *обзорного инжиниринга* (intuition engineering). Инструмент Netflix Vizceral (рис. 16.2) является отличным примером инструмента, который обеспечивает мгновенное обзорное восприятие общего состояния системы.

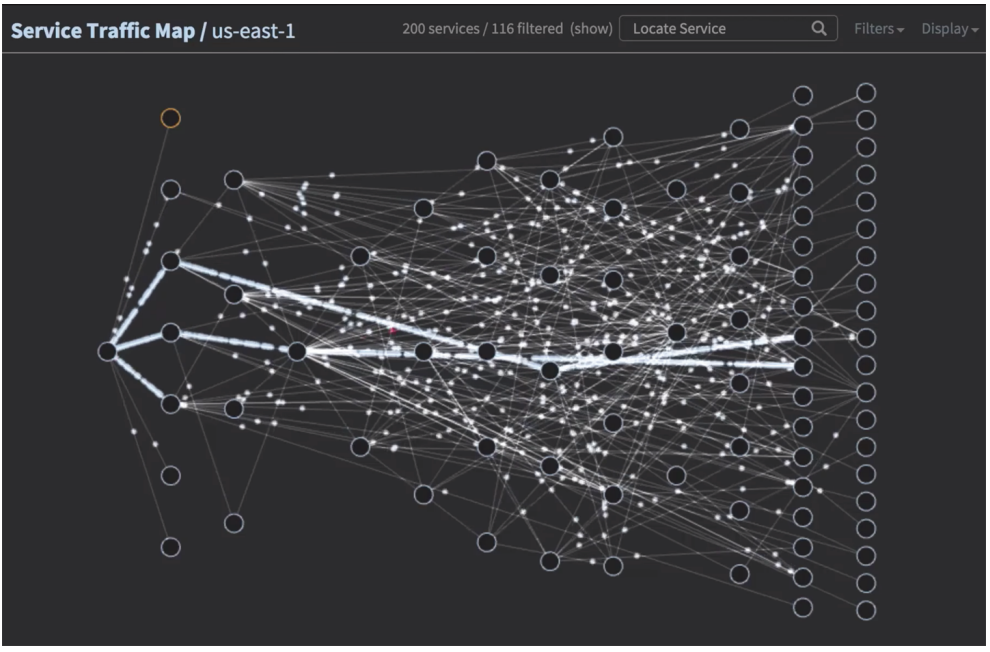

Рис. 16.2 ❖ Снимок экрана Vizceral[1]

В случае Vizceral сам визуальный интерфейс обеспечивает постоянно обновляемое представление системы, которая целостно дает представление о ее работоспособности или выходных данных. Это позволяет человеку с первого взгляда проверить предположения о состоянии системы и потенциально получить дополнительные знания о текущих эксплуатационных свойствах. Vizceral сам по себе не проводит эксперименты, но он может взаимодействовать с симуляциями и, безусловно, предоставляет проактивный инструмент для проверки поведения системы.

Итак, на одном конце спектра CV у нас есть эмпирические инструменты, которые обеспечивают проверку посредством экспериментов. На другом конце спектра у нас есть качественные инструменты, которые обеспечивают

[1] *Vlad Shamgin.* Adobe Contributes to Netflix's Vizceral Open Source Code // Adobe Tech Blog, Dec. 6, 2017, https://oreil.ly/pfNlZ.

проверку за счет участия человека в интерпретации. Между этими полюсами лежит обширное поле для исследований и разработки прикладных инструментов в индустрии программного обеспечения.

16.3. CV в реальной жизни: ChAP

Самым ярким примером CV на сегодняшний день является также самый сложный пример хаос-инжиниринга в отрасли. Платформа автоматизации хаос-экспериментов (chaos automation platform, ChAP)[1] была разработана в Netflix, в то время как Нора была инженером, а Кейси – руководителем команды разработчиков Chaos Team. Она прекрасно иллюстрирует передовые принципы, освещенные в этой книге, а также цикл обратной связи, который мы ожидаем от CV.

Платформа ChAP полностью автоматизирована. Она анализирует микросервисную архитектуру, выбирает микросервисы для проверки, строит эксперименты для этого микросервиса и проводит эксперимент в рабочее время. Гипотезы эксперимента принимают следующую форму: в условиях X для микросервиса Y клиенты по-прежнему смотрят нормальное количество видеопотоков. X обычно представляет собой величину задержки в нисходящем направлении, а нормальность определяется контрольной группой.

16.3.1. Выбор экспериментов в ChAP

В рамках внедрения ChAP был создан продукт под названием Monocle. Он расставляет приоритеты экспериментов, а также позволяет пользователям наблюдать за процессом расстановки приоритетов.

Monocle отправляет службам Netflix запрос на предоставление информации о своих зависимостях. В данном случае *зависимость* относится либо к настроенному RPC-клиенту, либо к команде Hystrix. Monocle объединяет данные из нескольких источников: телеметрической системы, системы трассировки (собственная система, концептуально основанная на Google Dapper), а также напрямую запрашивает работающие серверы о параметрах конфигурации, таких как предельное время ожидания.

Затем Monocle раскрывает собранные данные через пользовательский интерфейс, который обобщает информацию о зависимостях. Для каждой команды Hystrix Monocle выдает ответ, который описывает время ожидания и поведение повторных попыток, а также показывает, считает ли ChAP безопасным этот компонент в случае сбоя. В частности, это говорит о том, нужно ли владельцу системы что-то исправить, прежде чем он сможет безопасно провести хаос-эксперимент. Если сбой безопасен, то этот микросервис может быть добавлен в список и расставлен по приоритетам для экспериментов.

---

[1] *Ali Basiri et al.* ChAP: Chaos Automation Platform // The Netflix Tchnology Blog, https://oreil.ly/Yl_Q-.

16.3.2. Запуск экспериментов в ChAP

Благодаря интеграции со средством непрерывной проверки Netflix Spinnaker ChAP может анализировать код, развернутый для выбранной микросервисной службы, непосредственно в процессе определения приоритетов, предоставляемом Monocle. Для этого микросервиса в кластере работает несколько экземпляров. ChAP запускает два дополнительных экземпляра. Один действует как контрольный, другой как экспериментальный. Небольшое количество запросов рабочего трафика отбирается и равномерно распределяется между этими двумя экземплярами.

Экспериментальный объект подвергается воздействию. Чаще всего это внесение задержки в зависимые потоки.

Поскольку ответы отправляются из системы обратно клиентам, их предварительно проверяют на соответствие контрольному или экспериментальному экземпляру. Если данный ответ участвует в эксперименте, то соответствующий запрос участвует в подсчете KPI. По умолчанию KPI для Netflix – это количество запусков видео в секунду (starts per second, SPS).

Если KPI для контрольного и экспериментального экземпляра достаточно близки, то ChAP выдает положительный сигнал, который поддерживает гипотезу. Эксперименты обычно проводятся в течение 45 минут (минимум 20 минут).

Если же наблюдается значительное расхождение KPI, то эксперимент немедленно прекращается, новые запросы из рабочего трафика больше не отбираются. Контрольные и переменные экземпляры отключаются. Самое главное, что команда, ответственная за этот микросервис, получает уведомление о том, что в условиях X для их микросервиса клиенты испытывают трудности. Это открытие информирует команду о достижении предела прочности, и они сами должны выяснить, почему клиенты плохо проводят время в таких условиях и что с этим делать.

16.3.3. ChAP и принципы хаос-инжиниринга

Давайте посмотрим, как ChAP применяет передовые принципы хаос-инжиниринга, описанные в главе 3.

Построение гипотезы о стабильном поведении

В Netflix устойчивое поведение моделируется с использованием KPI количества потоков видео в секунду. Это показатель, к которому все разработчики могут легко получить доступ, и он тесно связан с работоспособностью сервиса и деловой ценностью. Клиенты, которые часто смотрят Netflix, рекомендуют его другим. Гипотеза «В условиях X для микросервиса Y клиенты по-прежнему смотрят нормальное количество видеопотоков» отражает это представление о желаемом стационарном поведении.

Моделирование различных событий реального мира

Большинство инцидентов в сложной распределенной системе можно смоделировать с использованием задержки, потому что в реальном мире

именно так проявляется большинство отказов узлов. Сравните это с загрузкой центрального процессора на 100 % или ошибки OOM. Такие модели не очень полезны, потому что на системном уровне они будут все равно проявляться как задержка или отсутствие ответа.

Выполнение экспериментов на производстве
ChAP работает в производственной среде. Распределенная система Netflix слишком велика, сложна и быстро меняется, чтобы иметь точную альтернативу в промежуточной среде. Запуск эксперимента в производстве обеспечивает уверенность в том, что мы строим систему, которая нам нужна.

Автоматизация непрерывного запуска экспериментов
ChAP работает непрерывно в рабочее время, не требуя вмешательства человека. Эксперименты можно добавить вручную, но это не обязательно, поскольку описанный ранее Monocle может ежедневно генерировать список приоритетных экспериментов.

Минимизация радиуса поражения
Механизмы, которые запускают новые контрольные и подопытные экземпляры и перенаправляют на них небольшой объем производственного трафика, уменьшают радиус поражения до безопасного значения. Если гипотеза опровергнута и с подопытными экземплярами случилось что-то ужасное, эксперимент не только немедленно прекратится, но и затронет лишь небольшой объем трафика. Дополнительное преимущество методики заключается в том, что одновременно можно проводить много разных экспериментов, поскольку каждый из них надежно изолируется от остальных.

16.3.4. ChAP как непрерывная проверка

Есть несколько важных нюансов, которые доказывают, что ChAP тоже является инструментом непрерывной проверки:

- ChAP работает непрерывно, в рабочее время, генерируя новые знания всякий раз, когда гипотеза опровергается;
- ChAP работает автономно, отдавая приоритет экспериментам с использованием эвристики и статического анализа кода в Monocle, и проводит эксперименты в порядке, предназначенном для максимизации скорости, с которой генерируются идеи;
- после небольших изменений ChAP может быть встроен как этап в конвейер инструментов CI/CD, образуя глубокую интеграцию CI/CD/CV.

16.4. Непрерывная проверка в системах рядом с вами

Существует как минимум три варианта использования CV в ближайшем будущем: проверка производительности, артефактов данных и корректности.

16.4.1. Проверка производительности

Существует множество инструментов для нагрузочного тестирования по нескольким параметрам производительности (скорость, отклонение задержки, параллелизм и т. д.). Большинство из них привязаны к одному конкретному шаблону использования, определяющему, сколько трафика может выдержать вся система. Лишь немногие инструменты работают на платформах, которые регулярно сталкивают их с часто меняющимся состоянием мира. Редкие инструменты изменяют схему использования в реальном времени, исходя из наблюдений за реальным производственным трафиком. Не многие из них тестируют производительность подсистем или выполняют стресс-тесты отдельных микросервисов для определения данных о выделении ресурсов и емкости. На подходе более интеллектуальные инструменты, которые дадут операторам больше информации о своих системах.

16.4.2. Артефакты данных

Базы данных и приложения хранения данных делают громкие заявления о надежности записи и извлечения данных и предлагают обширные гарантии для своих продуктов. Jepsen является отличным примером экспериментальной платформы, которая проверяет эти утверждения. Jepsen устанавливает платформу для экспериментов, генерирует нагрузку на базу данных, а затем ищет различные вещи, такие как нарушения коммутативности в последовательно согласованных базах данных, нарушения линеаризации, неверные гарантии уровня изоляции и т. д. Для таких систем, как критически важные службы обработки платежей, в которых необходимо сохранить транзакционные свойства, важно постоянно следить за наличием побочных эффектов данных.

16.4.3. Корректность

Не все формы «правильности» проявляются как состояние или желательные свойства. Иногда разные части системы нуждаются в согласовании либо по назначению, либо по логике, либо даже в некоторых случаях по здравому смыслу. В главе 1 в разделе 1.2.1 «Несоответствие между бизнес-логикой и логикой приложения» мы видели пример нарушения корректности, когда служба P получила ответ 404 для объекта, который, как она знала, существует. Это нарушение произошло, потому что логика была внутренне согласованной для каждого слоя, но не согласованной между слоями. Три распространенных уровня корректности в программном обеспечении:

Инфраструктура
: Иногда этот уровень называют «общим охватом». Распространенные сценарии поддержания корректности инфраструктуры включают распределение ресурсов, автоматическое масштабирование и предоставление в рамках SLA.

Приложение

В большинстве сценариев инфраструктура не знает, какие приложения работают поверх нее. Корректность приложения может быть зафиксирована в контрактах API между службами, в типах языка программирования или в языке описания интерфейса буферов протокола. Эти спецификации корректности могут быть строго применены к логике приложения, чтобы показать, что фрагмент кода доказуемо корректен с использованием решателя. К сожалению, даже если все компоненты сложной системы доказуемо корректны, система в целом все еще может проявлять нежелательное поведение.

Бизнес

Под этим уровнем часто подразумеваются предположения, сделанные в пользовательском интерфейсе. Чтобы бизнес был конкурентоспособным, ему часто приходится вводить новшества. Если бизнес-логика является инновационной, то не имеет смысла ее строго формализовать, поскольку она, вероятно, скоро изменится в ответ на условия в неопределенной среде. Следовательно, бизнес-логику сложнее всего проверить, и в любой ситуации с ограниченными ресурсами неизбежны несоответствия между бизнес-логикой, логикой приложений и инфраструктурой.

Если корректность трех перечисленных уровней не обеспечивается одновременно, то со временем проявятся проблемы на уровне системы. Это может быстро привести к инцидентам в плане как безопасности, так и доступности.

По мере развития этих категорий во всей отрасли найдутся новые возможности использования непрерывной проверки. Во многих случаях CV будет основываться на инструментах, предназначенных для хаос-инжиниринга.

Глава 17

Поговорим о киберфизических системах

Автор главы: **Натан Ашбахер**

Когда мы говорим о хаос-инжиниринге, то чаще всего подразумеваем создание, развертывание и управление инфраструктурой и приложениями, заложенные в основу различных интернет-ориентированных продуктов и сервисов, таких как Netflix, AWS и Facebook. В ландшафте этой области доминируют инновационные, масштабные, сложные и взаимосвязанные программно управляемые системы.

Предназначением хаос-инжиниринга в этих информационных и облачных экосистемах является обнаружение скрытых проблем и поиск ответов на вопросы о поведении системы. Применяя методы хаос-инжиниринга к сложным системам, вы пытаетесь узнать, где у вас есть слепые зоны. Иногда вы находите что-то такое, чего не могли найти во время реального сбоя. В других случаях вы обнаруживаете, что ваши предположения о поведении вашей системы полностью ошибочны. Возможно, вы даже проведете реверс-инжиниринг своей системы и узнаете, насколько далека ее реализация от первоначального вами проекта.

В этой главе мы рассмотрим четыре темы:

1) традиционная методика функциональной безопасности;
2) совпадения между функциональной безопасностью и хаос-инжинирингом (например, отработка режима отказа и анализ последствий);
3) методика функциональной безопасности и обширные возможности для совершенствования новых поколений программно-интенсивных систем, разрабатываемых в настоящее время;
4) применение принципов хаос-инжиниринга для заполнения пробелов в безопасности программно-интенсивных систем.

Мы рассматриваем эти темы в контексте киберфизических систем.

17.1. Происхождение и развитие киберфизических систем

Существует множество экосистем, незнакомых большинству разработчиков программного обеспечения, в которых программное обеспечение попрежнему является главным компонентом технологии. Одной из таких экосистем являются *киберфизические системы* (cyber-physical system, CPS). CPS – это взаимосвязанная аппаратно-программная система, которая развернута в окружающем физическом мире и взаимодействует с ним. Примеры таких систем варьируются от систем авионики или автономных транспортных средств до традиционных прикладных технологий, таких как промышленные системы управления на химическом заводе. Как правило, они состоят из целого зоопарка датчиков, встроенных устройств, средств связи и протоколов, полдюжины различных непонятных операционных систем и часто имеют критические зависимости от многочисленных компонентов фирменного черного ящика.

Современные вычислительные и облачные системы уже сейчас регулярно испытывают трудности из-за необходимой сложности. Это случается даже тогда, когда план развертывания ограничен серверами x86, которые согласованно обмениваются данными через обычные интерфейсы TCP/IP только с одной или двумя точками операционной системы. CPS добавляет несколько дополнительных уровней сложности благодаря разнообразию и непрозрачности своей концепции. Как будто этого еще недостаточно, существует растущая потребность в более тесной интеграции CPS в корпоративные информационные и облачные операции. Итак, возьмите всю сложность, которую вы знаете сейчас, добавьте немного безумной путаницы, а затем разверните систему в физическом мире, где риск неудачи в буквальном смысле бывает вопросом жизни или смерти.

Как вы уже знаете, цель хаос-инжиниринга состоит в том, чтобы дать ответы на вопросы и понять влияние сложности системы на ее устойчивость и безопасность для создания более надежных и отказоустойчивых систем, поэтому CPS идеально подходит для применения хаос-инжиниринга по всем фронтам.

Некоторые из постулатов хаос-инжиниринга настолько хорошо согласуются с необходимостью удаления «неизвестных неопределенностей» из систем, взаимодействующих с окружающей средой, что их определения уже фигурируют в старой инженерной дисциплине, так называемой функциональной безопасности. *Функциональная безопасность* стремится устранить неприемлемый риск причинения физического вреда людям и имуществу от машин, процессов и систем. Проектировщики и создатели CPS, особенно развернутых в критических средах, часто используют методы функциональной безопасности для оценки и управления рисками, связанными с их продуктами.

17.2. Слияние функциональной безопасности с хаос-инжинирингом

Стандарты и практика функциональной безопасности представлены в разнообразных формах. Существует множество отраслевых стандартов. Например, я профессионал в области функциональной безопасности автомобилей с сертификатом SGS-TÜV. Это означает, что у меня есть опыт работы по стандарту ISO 26262. Как вы уже догадались из длинного названия, это стандарт для автомобильной промышленности. Этот конкретный стандарт специально разработан для автомобильных электрических и электронных систем в пассажирских транспортных средствах. Упоминание «электронных систем» объясняет, каким образом этот стандарт имеет отношение к программному обеспечению: многие из этих электронных систем представляют собой небольшие встроенные компьютеры, выполняющие такие задачи, как координация автоматического экстренного торможения (AEB), или развлекают ваших детей в долгих поездках с помощью мультимедийного оборудования автомобиля.

Другие отрасли также имеют свои собственные стандарты, такие как авиация (DO-178C и DO-254) и ядерная энергетика (IEC 61513). Все эти бесчисленные стандарты происходят от общего предка, известного как «IEC 61508: Функциональная безопасность электрических, электронных и программируемых электронных систем, связанных с безопасностью». Здесь важно отметить, что в этих стандартах нет какой-то магии или науки. По большому счету, они кодифицируют передовые технические разработки, накопленные за многие десятилетия, иногда ценой трагедий, споров или того и другого. Поскольку это продукты передового опыта, из этих стандартов можно позаимствовать вещи, которые вы могли бы использовать в ходе обычной разработки CPS независимо от того, подлежит ли ваш продукт отраслевому регулированию.

Одна из таких полезных вещей называется «Анализ типов отказов и их последствий» (failure mode and effects analysis, FMEA). Судя по названию, это очень похоже на хаос-инжиниринг. В конце концов, занимаясь хаос-инжинирингом, вы вызываете отказ своей системы, а затем пытаетесь проанализировать последствия. Давайте рассмотрим, что относится к FMEA, а что нет.

Для методики FMEA характерны следующие шаги:

1) определите область анализируемой информации (например, всю систему, подкомпонент, проект, процесс разработки и т. д.);
2) определите все компоненты функциональности, которые находятся в этой области;
3) проведите мозговой штурм и составьте исчерпывающий список всех элементов, которые могут потерпеть сбой при отказе определенного компонента функциональности;
4) перечислите все последствия каждого потенциального режима отказа для каждого элемента функциональности;
5) присвойте числовые значения (рейтинги) серьезности, вероятности и обнаруживаемости (до отказа) каждому режиму потенциального отказа;

6) перемножьте между собой рейтинги каждого режима отказа, чтобы вычислить то, что называется числами приоритета риска;
7) запишите все полученные значения в электронную таблицу и вернитесь к ней при выполнении итерации в своей системе.

Теория гласит, что чем выше число приоритета риска, тем больше внимания вы должны уделять этим режимам сбоев, пока вы не устранили их все в своем проекте и/или реализации. Тем не менее вы можете столкнуться с непредвиденными препятствиями, которые могут серьезно снизить достоверность и полезность FMEA:

- весьма вероятно, что в любой области, которую вы выбрали для анализа, на самом деле может отсутствовать какое-либо критическое допущение;
- очень сложно выделить каждый элемент функциональности в заданной области;
- за исключением тривиальных случаев, очень маловероятно, что вы полностью исчерпали список вещей, которые могут пойти не так;
- еще менее вероятно, что вы полностью предусмотрели все последствия возможного отказа, не говоря уже о режимах отказа, о которых вы даже не думали;
- даже вычисляемые по нормам рейтинги по своей природе несколько произвольны и, следовательно, не обязательно отражают реальность;
- произвольный характер рейтингов означает, что рассчитанный приоритет также может быть искаженным;
- электронная таблица, полная ваших лучших идей и намерений, на самом деле не связана с вашей системой. Нет никакой гарантии, что содержимое таблицы соответствует реальной системе.

Этот процесс сам по себе очень восприимчив к различным сбоям из-за ошибочного определения того, что нужно анализировать, недостатка воображения в отношении того, что может пойти не так, ограниченной возможности точно оценить масштаб и глубину последствий отказа, довольно произвольного характера рейтингов и немного более тонкого вопроса, связанного с многоточечными сбоями (мы обсудим его позже в этой главе). По большей части, когда вы занимаетесь FMEA, вы делаете целую кучу эмпирических догадок (educated guesses)[1].

Но если в этом процессе таится такое количество потенциальных ловушек, почему он является настолько важной частью наших многолетних попыток создания надежных и отказоустойчивых систем? Во-первых, для действительно критически важных систем выполнение чего-то вроде FMEA – это лишь один из множества различных процессов. Для соответствия некоторым стандартам требуется прохождение двойных и тройных проверок. Во-вторых, трудно недооценить одну из самых значительных ценностей такого процесса, как этот, – он ставит вас в тупик.

Оказывается, если вы собрали достаточно много опытных специалистов по смежным дисциплинам и достаточно долго держите их вместе, чтобы они

[1] На русский язык *educated guesses* иногда переводят как «метод научного тыка». – *Прим. перев.*

занимались такой деятельностью, как FMEA, они на самом деле довольно неплохо извлекают контекст из собственного опыта и за несколько подходов выжимают из системы пугающее количество неопределенностей.

17.2.1. FMEA и хаос-инжиниринг

Если методики функциональной безопасности, такие как FMEA, уже проделали хорошую работу, то какую выгоду можно получить от внедрения хаос-инжиниринга в этих системах? Наиболее очевидное применение состоит в том, что эксперименты хаос-инжиниринга можно использовать для проверки или опровержения каждого предположения, сделанного в вашем FMEA:

- вы можете собрать несколько разных групп экспертов для самостоятельного выполнения FMEA и поиска расхождений в их результатах;
- вы можете вводить сбои в области, не входящие в сферу действия FMEA, и посмотреть, не распространить ли на них FMEA;
- вы можете вызвать перечисленные вами отказы и попытаться измерить их реальные, а не воображаемые последствия;
- вы можете использовать результаты своих экспериментов в хаос-инжиниринге, чтобы напрямую переносить изменения в ваш FMEA.

В более формализованных процессах разработки вам, как правило, потребуется составить план тестирования и показать доказательства прохождения тестов FMEA перед выпуском вашего продукта в самостоятельную жизнь. В настоящее время можно приобрести чрезвычайно сложные средства тестирования компонентов системы. Существует множество поставщиков оборудования, продающих различные устройства для ввода отказов, таких как имитация коротких замыканий в электрических интерфейсах, неисправные механические разъемы и многое другое.

Хаос-инжиниринг добавляет важные детали к процессу создания надежных и отказоустойчивых CPS, помогая как функциональным специалистам по безопасности, так и системным инженерам лучше понимать эффекты взаимодействия сложного программного и аппаратного обеспечения, объединяемых в еще более сложные системы. Это особенно верно для систем, которые являются высокоавтоматизированными, где операторы-люди либо полностью удаляются из цикла, либо взаимодействуют на гораздо большем расстоянии, часто через несколько абстракций системы, вдали от критических интерфейсов и операций.

17.3. Программное обеспечение в киберфизических системах

Программное обеспечение обладает некоторыми уникальными свойствами, которые делают его особенно проблематичным, когда имеются остаточные ошибки в его реализации (известные проблемы), скрытые ошибки в его реализации (ошибки, о которых вы не знаете) или систематические ошибки

в его разработке (проблемы, которые требуют от вас принципиально пересмотреть что-нибудь в спецификациях вашей системы).

Использование программного обеспечения совершенно не похоже на использование механических или электрических компонентов. Вы можете вызывать программную функцию тысячу раз в тысяче разных мест, и это всегда будет одна и та же функция. В отличие от этого, один физический экземпляр резистора не используется тысячей разных способов на тысяче различных электронных плат. Если у вас один бракованный резистор, значит, у вас один отказ. Если у вас бракованная программная функция, значит, у вас будет отказ везде, где она используется.

Этот довольно уникальный аспект программного обеспечения является фантастически удобным, когда функция реализована и ведет себя так, как вы ожидаете. И возникает совершенно противоположная ситуация, когда что-то идет не так. Вы получаете неожиданное или неправильное поведение во всей вашей системе одновременно. Чтобы окончательно добить ваш оптимизм, последствия неправильной работы функции в остальной системе могут сильно зависеть не только от места события, но и от почти бесконечного пространства возможностей, основанного на состоянии системы и сочетании факторов окружающей среды.

Мы взаимодействуем с программным обеспечением на очень высоком уровне абстракции. Чтобы рассуждать о том, что на самом деле будет делать программное обеспечение при работе в реальной системе, приходится задействовать огромное количество базовых моделей. Каждая абстракция увеличивает сложность ради удобства, но эта сложность также добавляет новые измерения неопределенности. В программном обеспечении можно с легкостью случайно создавать тесно связанные зависимости и отношения между различными частями функциональности, даже не осознавая этого.

Наконец, программное обеспечение может провоцировать поведение системы, которое напоминает неисправное оборудование. Рассмотрим общий случай взаимосвязанной системы со сбойным стеком TCP/IP или неправильно настроенным сетевым интерфейсом. В зависимости от характера проблемы для других частей вашей системы это может выглядеть как поломка оборудования или обрыв сетевого кабеля. Такие ситуации, когда программный сбой скрывается за маской аппаратной неисправности, являются скрытыми отказами и легко приводят к возникновению нескольких одновременных режимов отказа, также известных как *многоточечные отказы* (multipoint failures).

Стоит подчеркнуть небольшую, но тревожную особенность традиционного процесса FMEA: обычно, работая с FMEA, вы не учитываете в своем анализе одновременные многоточечные сбои. Для каждого перечисленного аспекта функциональности вы выполняете анализ, предполагая, что остальная часть системы вообще не дает сбоев, и за один раз в области, которую вы анализируете, происходит отказ только одного элемента. Если учесть, что FMEA разработали для того, чтобы люди достигли лучшего – не идеального – понимания системы, это можно считать вполне приемлемым допущением. В любой нетривиальной системе с многоточечными отказами борьба с комбинаторной сложностью режимов и последствий отказов может оказаться неразрешимой.

Это не означает отрицание типичных подходов, таких как FMEA, для оценки рисков в вашей системе. В конце концов, имейте в виду, что даже в области функциональной безопасности ожидается, что «удастся устранить недопустимый риск», а с точки зрения электромеханических систем *крайне маловероятно*, что система будет испытывать независимые многоточечные (то есть одновременные) отказы. С этой точки зрения добровольные ограничения FMEA полностью обоснованы. Возникновение множественных одновременных независимых сбоев в системе представляется маловероятным, поскольку у вас, вероятно, нет двух полностью независимых механических или электрических компонентов, которые могут случайно сломаться в одно и то же время. Кроме того, последствия подобных перекрывающихся отказов считаются настолько неустранимыми и катастрофическими, что не заслуживают детального анализа. Это немного похоже на то, как если бы вы попытались оценить влияние отказа вашей системы аутентификации из-за удара молнии по серверам, наводнения в центре обработки данных и удара астероида по Земле. Тот факт, что ваша система аутентификации находится в автономном режиме, просто не имеет значения в сложившихся обстоятельствах, и, вероятно, вы ничего не можете с этим поделать.

Однако в *программно-интенсивных системах*[1] (software-intensive systems) дела обстоят иначе. В отличие от одного неисправного электрического компонента, легкость, с которой проблемы программного обеспечения распространяются по системе как эпидемия, имеет неприятный побочный эффект – трудность или невозможность точной оценки режимов отказов, их последствий, серьезности, вероятности и обнаруживаемости. Именно здесь эмпирическая проверка при помощи инструментов хаос-инжиниринга открывает новые возможности для экспериментов и исследований.

17.4. Хаос-инжиниринг как следующий шаг после FMEA

Некоторые из процедур FMEA коррелируют с принципами хаос-инжиниринга:

Определение области и функциональности == Гипотеза об устойчивом состоянии.

Мозговой штурм о возможных неполадках == Различные события в реальном мире.

Присвоение баллов значимости и вероятности == Минимизация радиуса поражения.

---

[1] Программно-интенсивные системы или преимущественно программные системы – это системы, которые преимущественно состоят из программного обеспечения, но имеют и неотъемлемую физическую часть для взаимодействия с внешним миром. – *Прим. перев.*

Процесс FMEA помогает расставить приоритеты и подсказывает, где нужно копать глубже и, возможно, провести эксперименты для проверки предположений. Принимая во внимание элементы с наивысшим числовым значением риска, можно извлечь максимальную пользу из хаос-экспериментов и исследовать реальные риски, скрытые как в ваших предположениях, так и в ваших системах.

Тем не менее этот процесс сопряжен с огромным количеством накладных расходов и процедур. Для определения целей экспериментов существуют и другие эффективные подходы. Для начала попробуйте поразмышлять о том, как вы проектируете и создаете систему и от каких допущений и гарантий вы критически зависимы при этом.

Например, невероятно сложно создать что-то на основе неопределенного или недетерминированного поведения. Скажем, я поставил перед вами задачу разработать систему на языке программирования, где единственными операторами, которые работают правильно и детерминистически, являются «сложение» и «присваивание», а все остальные ведут себя непредсказуемо. Если вы не откажетесь от работы по причине ее безумного технического задания, то сделаете все возможное, чтобы как можно больше использовать только сложение и присваивание для абсолютно всего, что может сойти вам с рук. В конце концов, для любого полезного действия у вас будет наготове нагромождение хитрых трюков и абстракций, воздвигнутое на фундаменте из двух кирпичиков сложения и присваивания, потому что это единственные элементы, в которые вы можете верить.

Когда люди начинают планировать хаос-эксперименты, у них возникает стремление сначала взяться за самый большой «черный ящик». Сначала это кажется разумным: с одной стороны, у вас есть множество вещей, которым вы безоговорочно доверяете; а с другой – у вас есть аморфная куча подозрительной неопределенности. Но взгляните на это решение в контексте только что рассмотренного надуманного примера с программированием. Исходя из ограничений языка программирования, который вы использовали, код вашей системы, скорее всего, в конечном итоге будет использовать сложение и присваивание везде, где для системы важно предсказуемое поведение, и вы будете использовать другие операторы как удобство, когда результаты или последствия менее критичны. По сути, вы бы неосознанно склоняли некоторые части вашей системы к устойчивости, а другие – к отказу.

Учитывая это, не имеет смысла начинать с хаос-экспериментов там, где у вас уже есть подозрение в ненадежности. Вместо этого начните с проверки того, существует ли скрытая неопределенность в вещах, которые вы считаете абсолютно надежными. *Известные определенности, которые не проверены, на самом деле являются просто неизвестными неопределенностями наихудшего вида*. Вы не только не знаете об их существовании, но и критически зависите от них повсюду. В нашем примере отличный начальный эксперимент с хаосом может выглядеть как преднамеренное нарушение гарантий, которые, как вам сказали, сохранятся. Если вы построили систему на основе сложения и присваивания, которые всегда работают так, как вы ожидаете, то внесите ошибку в вашу систему в нескольких местах и на нескольких уровнях, чтобы предположение больше не работало. Испортите исходный код. Прервите ра-

боту компилятора. Смоделируйте процессор, который делает что-то невозможное. Тогда вы увидите, насколько высок риск, которому вы подвергаетесь только из-за потенциальных неизвестных факторов в ваших основных предположениях и критических зависимостях.

Чтобы создать что-то действительно надежное, разработчики стремятся построить как можно больше вспомогательных вещей поверх того, что мы считаем надежным в наших системах. В нашем надуманном примере, настолько глубоко используя только два надежных элемента языка программирования, мы тем самым внесли в систему огромную уязвимость. И теперь, чтобы все работало, нам *действительно* нужны безукоризненно работающие сложения и присваивания. Для других частей нашей системы мы предвидели отказ и неявно включали дополнительные факторы надежности в наш проект. В итоге мы сталкиваемся с парадоксом: части системы, которым мы доверяем за их надежность, становятся огромными векторами риска при малейшем сбое, а наименее доверенные части оказываются самыми надежными.

В мире CPS эта проблема лишь усугубляется, поскольку здесь многие вещи должны работать в рамках по-настоящему жестких и продуманных допусков, чтобы все, что построено сверху них, могло выполнять свое предназначение. Одним из факторов CPS, в отношении которого действуют очень жесткие ограничения, является время, в частности в физических системах реального времени. В отличие от большинства ИТ-систем, в которых допуски времени могут иметь относительно широкие пределы, в CPS часто существуют строгие эксплуатационные границы, потому что недопустимо, чтобы определенные вещи происходили не вовремя. В физическом мире вы не сможете просто попробовать еще раз через некоторое время; к тому времени вы можете буквально оказаться в огне или под водой.

Хорошая новость в том, что инженеры-электрики умеют строить очень надежные встроенные часы и схемы синхронизации. Плохая новость заключается в том, что инженеры настолько преуспели в этом, что полностью полагаются на ожидаемое поведение часов и принимают его за основу, на которой проектируются и строятся системы. Когда я проектирую свою систему, я начинаю думать о времени точно так же, как думал о сложении двух чисел в моем примере языка программирования. Я склоняю свой проект к зависимости от времени, потому что больше уверен в надежности этого компонента.

Количество путей, которыми это предположение может привести к катастрофическим каскадным проблемам даже при незначительных отклонениях в реальном поведении, воистину огромно. Требования к интервалам времени связаны с контурами управления. Небольшие ошибки во времени могут теперь всплывать по всей моей системе и влиять на функции, которые отвечают за активацию физических частей CPS. Проблемы могут быть банальными, как запуск незначительного процесса в неподходящее время, но также могут означать, что вы не выполняете абсолютно необходимую функцию именно тогда, когда вам это нужно. Ошибки синхронизации могут вызвать ложное срабатывание сторожевых таймеров или запустить аварийное поведение, такое как выключение и включение питания компонента либо всей

системы. Постоянные проблемы с синхронизацией могут привести к тому, что подсистема отключит сама себя, вернется, снова отключится и будет делать это бесконечно.

Инженеры по распределенным системам не доверяют настенным часам. Они понимают, что в распределенной системе маловероятно идеальное совпадение часов по времени и тактовой частоте. Это глубокое недоверие и скептицизм не особо распространено среди других системных инженеров, поэтому они строят очень большие и сложные системы, исходя из предположения, что гарантировано точное совпадение времени. Раньше было так: системы были меньше, функции были проще, а локальные часы были довольно надежными. Глобальное время редко приходилось распространять на большие расстояния или в ответ на разные запросы. Инженеры по встраиваемым системам и системные инженеры, ответственные за безопасность, постепенно приходят к пониманию того, что они больше не являются разработчиками локальной функциональности в замкнутой коробке. Теперь мы все разработчики распределенных систем. Происхождение хаос-инжиниринга из распределенных программных систем идет на пользу CPS, поскольку разработчики распределенных систем изначально привыкли беспокоиться о вещах, которые только сейчас попадают в поле зрения инженеров встраиваемых систем.

Сотрудничая с инженерами в области разработки автономных транспортных средств, я часто предлагаю использовать ограничения синхронизации в качестве первой цели для экспериментов хаос-инжиниринга. Я еще не встретил никого, кто хотел бы получить интересную, а в некоторых случаях и экзистенциальную информацию о том, как ведет себя их система, когда их часы обманывают их. Мой совет для тех, кто читает эту книгу, кто работает над критически важными программно-интенсивными системами, кто ищет место, где можно начать применять методы хаос-инжиниринга: начните с *тайминга* – распределения и синхронизации процессов во времени. Многие из компонентов, с которыми вы, вероятно, будете работать, уже достаточно хороши в обработке или, по крайней мере, в обнаружении таких вещей, как повреждение данных, переворачивание битов, удаление сообщений и т. д. Проблемы тайминга могут быть очень тонкими, часто каскадными, но иногда являются вопросом жизни и смерти киберфизической системы.

17.5. Эффект щупа

Чрезвычайные узкие допуски и высокие ожидания надежности во встроенных и жизненно важных системах сами по себе являются проблемой при применении методов хаос-инжиниринга к устройствам и системам. Внедрение чего-либо для создания неисправности в системе и чего-то еще для проведения измерений обычно не проходит бесследно. Рассмотрим ситуацию, когда вы работаете с чрезвычайно чувствительным электрическим устройством. Уровень сигнала очень низкий по сравнению с уровнем шума. Вы хотите про-

вести измерение параметров этой системы, чтобы что-то проверить, поэтому подключаете свой осциллограф к тракту сигнала. К сожалению, идеального щупа не существует. Ему присущи сопротивление и емкость наконечника, а провод и зажим заземления будут иметь некоторую индуктивность. Само присутствие щупа осциллографа повлияет на ваши измерения.

Это влияние обычно называют *эффектом щупа*, или *эффектом зонда*: появление непреднамеренных побочных эффектов при попытке выполнить измерение. В области хаос-инжиниринга этот побочный эффект мешает нам дважды. Первый эффект щупа возникает при проведении измерений, а второй, на уровне системы, возникает при внедрении сбоя или другого переменного фактора. Рассмотрим программную систему, в которой вы хотите провести хаос-эксперименты в ядре Linux на каком-то интерфейсе ввода-вывода с малой задержкой. Вы вставляете программный код где-то на «пути прохождения сигнала» интерфейса ввода-вывода, чтобы снимать сигнал через посредника. Вы хотите инвертировать несколько битов на выходе, чтобы «щуп» хаоса воздействовал на любое вышестоящее приложение, зависящее от этого интерфейса, но вы не хотите, чтобы он был активным все время, поэтому вам нужно иметь возможность включать и выключать его. Поскольку щуп может быть с равной вероятностью как подключен, так и отключен, теперь для каждого бита, выходящего из этого интерфейса, вам приходится задавать вопрос: «Я сейчас провожу хаос-эксперимент или нет?» – с последующим условным переходом по ветви алгоритма. Какую вычислительную стоимость это имеет, каковы побочные эффекты в системе, и каковы побочные эффекты этих побочных эффектов? Взаимосвязано ли это условие с аппаратной оптимизацией работы процессора, такой как предсказатель ветвлений? Если это так, то влияет ли это на быстродействие других участков кода, поскольку конвейер нужно было очистить?

Чем больше слоев и абстракций в системе, тем заметнее могут быть эффекты щупа. Напомню, что мы еще даже не вводили ошибки. Мы только закладываем фундамент. Теперь представьте, что мы начинаем вводить ошибки, инвертируя каждый третий бит. Мы получили не только хаос в виде инверсии каждого третьего бита, но также постоянно считаем биты, значит, эта ветвь кода становится еще длиннее, что безусловно влияет на тайминг и кешпамять процессора. Я решил внедрить только одну ошибку, чтобы посмотреть, как это повлияет на поведение системы, но на самом деле я внедрил несколько видов непреднамеренных ошибок в другие ресурсы в системе, помимо этого интерфейса ввода-вывода.

17.5.1. Решение проблемы щупа

Чем ближе ваша система к пределу возможностей и чем больше ваш проект и реализация зависят от этих действительно жестких допусков, тем больше вероятность, что эффект щупа сам по себе может стать серьезной проблемой. К счастью, есть несколько методов, чтобы справиться с этим; к сожалению, ни один из них не идеален, если вы стремитесь к абсолютной точности в понимании нюансов поведения вашей системы.

Решение этой проблемы звучит одновременно банально и нереально. Вы должны постараться понять, каким будет воздействие вашего щупа. В ситуации с осциллографом мы работаем с физическим щупом, и нам помогает знание электрических характеристик оборудования. В многослойном нагромождении абстракций от микроархитектуры ЦП вплоть до стека ваших приложений, потенциально распределенных по N-числу устройств, определить воздействие программного «щупа» чрезвычайно сложно.

Есть много способов оценить начальное влияние, которое вносит ваш щуп. Например, вы можете написать тщательно выверенный код, который гарантирует, что ваш программный щуп всегда будет занимать строго определенный объем памяти, или что ваш щуп никогда не будет внезапно падать, или что для выполнения функции щупа всегда требуется ровно четыре команды процессора. Это все достоверные величины. Вам точно известна степень локального воздействия вашего зонда. Вы также можете выполнить эмпирические измерения эффекта вашего щупа, чтобы получить аналогичную уверенность в его поведении без верификации соответствующего кода. Чего вы не знаете, так это степень влияния щупа на систему, в то время когда вы не вносите преднамеренный хаос. Чтобы решить эту проблему, вы можете создать фиктивные щупы, которые будут загружать систему аналогичным образом, даже если вы не вносите хаос. Как только вы создадите фиктивные щупы, то сможете с некоторой степенью достоверности понять, как щуп сам по себе влияет на поведение вашей системы. Затем вы можете ввести в систему настоящие щупы для проведения реальных хаос-экспериментов и проведения реальных измерений. Это существенно облегчит анализ результатов.

Другой подход меняет точность на удобство. Рассмотрим свойство вашей системы, в оценке которого вы наиболее заинтересованы. Например, если вы наиболее обеспокоены надежностью вашей системы с точки зрения проблем со связью, то, вероятно, не имеет значения, что щупы могут влиять на загрузку процессора.

Это может обеспечить интересную возможность для экспериментов, если вы правы насчет того, что ваша система не чувствительна к изменениям нагрузки на процессор. Или, например, вы хотите понять, что происходит, когда ваши процессы не могут получить больше памяти. В этом случае вряд ли имеет значение, заполняет ли ваша измерительная функция пропускную способность сети. Сосредоточив внимание на величинах или эксплуатационных ограничениях, к которым ваша система наиболее чувствительна, можно провести триангуляцию того, влияет ли эффект щупа на результаты измерений. В каком-то углу действительно может быть выраженный эффект щупа, но это не тот угол, который вы хотите оценить. В таком случае, проводя хаос-эксперименты, вы имеете все основания полагать, что эффекты самого щупа в достаточной степени отделены от эффектов, которые вы пытаетесь измерить с помощью щупа после введения переменных факторов в систему.

Иными словами, у вас всегда есть простой путь: собирать свою систему со всеми встроенными щупами, постоянно развертывать ее, всегда внося пассивную нагрузку щупов в систему, даже если они не предпринимают никаких действий, чтобы изменить поведение системы. Разместите щупы в нужных местах, установите для них статическое выделение ОЗУ, сделайте так, чтобы

все ветки кода выполняли реальную работу или фиктивную работу, которая является подходящей имитацией реальной работы, и просто сделайте так, чтобы вся система работала в стабильном режиме, эквивалентном наихудшему случаю с точки зрения использования ресурсов. Это похоже на постоянную работу всей системы в режиме отладки и, очевидно, не подходит для систем, где приоритетом является эффективность.

Как говорится, бесплатного обеда не бывает. Это относится и к тому случаю, когда вы подаете щупы в систему и спрашиваете ее мнение о вашей кулинарии.

17.6. Вывод

Наверняка вы заметили мою предвзятость как бывшего инженера-механика и производственника, но одна из вещей из моей прошлой жизни, по которой я всегда ностальгирую, – это идея «свойств материала». Это такие вещи, как прочность на растяжение, магнитосопротивление, поверхностное натяжение, температура вспышки и т. д. В мире программирования нет ничего подобного. Поэтому мы очень ограничены в разнообразии вариантов построения и интеграции программных систем. Эта идея посетила меня, когда я прослушал два разных доклада о хаос-инжиниринге: один от Норы Джонс, когда она работала в Netflix, и другой от Хизер Накамы, когда она работала в Microsoft.

Слушая этих двух ораторов, я понял, что они накопили очень ценные знания о своих системах. Помимо некоторых идей о том, как извлекать эти знания с помощью хаос-инжиниринга, у них не было особого способа напрямую обмениваться знаниями из систем, с которыми Нора и Хизер имели дело изо дня в день. Конечно, в этих системах есть некоторые похожие компоненты, но если Нора захочет взять подсистему Netflix и добавить ее в инфраструктуру Хизер в Microsoft, чтобы внести туда новую функциональность, им будет очень трудно знать наперед, будет ли это работать. Какое влияние это окажет на систему? Какое влияние это окажет на другие аспекты инфраструктуры Хизер? Понимание программных систем, с которыми мы имеем дело сегодня, во многом зависит от контекста.

Свойства материала, с другой стороны, одинаковы во всех случаях и не зависят от контекста. Мы измеряем поведение титана в испытательных лабораториях и подвергаем его разным воздействиям – сгибаем, ломаем, быстро нагреваем и охлаждаем. Мы проводим ряд измерений и получаем новые знания о титане, которые можем использовать не только для того, чтобы доказать жизнеспособность конкретного применения. Обладая достаточным количеством знаний, мы можем смоделировать различные воздействия и их эффект без затрат времени и средств на их фактическое выполнение.

Так инженеры-конструкторы применяют инструменты анализа в физическом мире. Они могут спроектировать физическую вещь в виде виртуальной модели и определить параметры отдельных деталей и материал, из которого они сделаны. Они могут применить к модели нагрузки и напряжения, кото-

рые ожидаются при практическом использовании. Поскольку мы измерили так много различных свойств материалов, и эти свойства обладают безусловной повторяемостью, можно предсказать, как вещь будет вести себя задолго до физического производства. Благодаря знанию повторяющихся свойств мы можем проектировать фюзеляжи самолетов, моделировать деформацию сминаемой зоны при столкновениях транспортных средств и проектировать подводные турбины, которые минимально возмущают окружающую среду, в которой они вращаются. Это существенно подняло планку сложности физических вещей, создаваемых инженерами-конструкторами.

Я считаю, что методика хаос-инжиниринга – это путь в такой же мир для программно-интенсивных систем. Постепенно в программных системах появляются аналоги свойств материалов. Мы рассматриваем такие показатели программной системы, как доступность, загрузка ЦП и пропускная способность сети. Мы можем подключать щупы и проводить измерения. Формализация хаос-инжиниринга создаст эквивалент «свойств материала» для программно-интенсивных систем. У нас появятся эмпирические методы, позволяющие выполнять контекстно-независимые измерения для всех видов систем. Инструменты хаос-инжиниринга похожи на лаборатории тестирования материалов, в которых мы исследуем их предельные параметры. Если все сделать правильно, то хаос-инжиниринг переносит инциденты с поздней стадии эксплуатации на раннюю стадию проектирования системы. Программно-интенсивные системы достигли такого уровня сложности, когда сама сложность сдерживает дальнейшие инновации в программных системах. Хаос-инжиниринг, как минимум, указывает нам путь, ведущий за пределы этой проблемы.

Об авторе

Натан Ашбахер начал свою карьеру в области программирования станков с ЧПУ, где ошибки и упущения приводили к искореженным грудам металла, сломанным инструментам и неизгладимым впечатлениям от громкой аварии. С тех пор он начал разрабатывать отказоустойчивые распределенные платформы данных и глобальные сети обработки транзакций. Сначала Натан применил принципы хаос-инжиниринга к проблемам в финтехе, затем перенес эту методику на разработку платформы автономных транспортных средств, а теперь занимается разработкой продуктов и технологий в своей компании Auxon, которая использует формализованные методы и сложное внедрение отказов для проверки высокоавтоматизированных систем.

Глава 18

HOP с точки зрения хаос-инжиниринга

Автор главы: **Боб Эдвардс**

18.1. Что такое HOP?

Что такое *личная и корпоративная эффективность* (human and organizational performance, HOP)? Это подход к совершенствованию организационных структур и функциональных процессов для оптимизации критически важных для бизнеса свойств, таких как безопасность. Возможно, из-за прошлых связей с производством HOP часто ошибочно принимают за процесс, но это не описание порядка действий. В нем есть гибкость и творчество, основанные на пяти принципах, изложенных в этой главе.

Поскольку HOP широко применяется в производственном мире, нам было бы полезно учиться у хаос-инжиниринга и внедрять методы, которые показали ценность в программных системах. И хаос-инжиниринг, и HOP имеют философские корни в так называемой философии «нового взгляда»[1] на науку о безопасности. Это фундаментальный сдвиг в нашем понимании несчастных случаев, человеческих факторов. HOP – это применение моделей нового взгляда к расследованиям аварий и организационным изменениям для создания более безопасных систем.

---

[1] «Новый взгляд» – это приблизительная (не строго определенная) философия, которая контрастирует со «старым взглядом» на науку о безопасности, пропагандируемая во многих книгах Сиднея Деккера и его соавторов. Эта философия отражает последние исследования и тенденции в области науки о безопасности и устойчивости с начала 2000-х годов, включая такие концепции, как «Безопасность-II» и «Безопасность по-другому», популяризируемые в научной и популярной литературе по этому вопросу. См.: *Sidney Dekker.* Reconstructing Human Contributions to Accidents // Journal of Safety Research 33 (2002), p. 371–385, – где приведено первое описание «нового взгляда» на безопасность.

18.2. Ключевые принципы HOP

Мы взяли пять ключевых принципов из исследований в области *технологии повышения продуктивности персонала*[1] (human performance technology, HPT) и адаптировали их для применения в индустриальном мире. Эти принципы служат нашим руководством по повышению эксплуатационной надежности и устойчивости в таких отраслях, как машиностроение, коммунальное хозяйство, химическая промышленность, нефтегазовая отрасль и даже медицина. Эти пять принципов применимы практически к любой организации:

1) ошибка – это норма;
2) вина ничего не исправляет;
3) контекст определяет поведение;
4) обучение и улучшение имеют жизненно важное значение;
5) важны осмысленные ответы.

18.2.1. Принцип 1: ошибка – это норма

Все люди ошибаются. Зачастую люди, которые делают больше всего ошибок, – это те, кто больше всех работает. Однажды я увидел в машинном зале плакат: «Не ошибается лишь тот, кто ничего не делает». Если мы ожидаем, что люди на своем рабочем месте будут безупречно выполнять свою работу, то обрекаем себя на неудачу. Мы должны предотвращать возможные ошибки; тем не менее нам также необходимо предусмотреть возможность безопасного сбоя. Этот подход широко распространен у производителей автомобилей: они строят системы безопасности, чтобы предотвратить аварии, и одновременно создают автомобили, которые удивительно безопасны при попадании в аварию. Автомобильные конструкторы стремятся максимально обезопасить водителей, но они также знают, что всех аварий не избежать. В автомобили встроены системы предупреждения и даже автоматическая коррекция руления и помощь в торможении, что помогает предотвратить несчастные случаи. В дополнение к стратегии предотвращения производители автомобилей также проектируют и делают сминаемые зоны, подушки безопасности и пространство для выживания, на случай, когда не сработает предотвращение. Они рассчитывают на несчастные случаи и стараются делать так, чтобы машина была одновременно трудно разрушаемой и безопасно разрушаемой.

18.2.2. Принцип 2: вина ничего не исправляет

Вина не только ничего не исправляет – она заставляет избегать важных разговоров и скрывать необходимую информацию. Если человеку кажется, что

---

[1] Вспомните о методах управления процессами и улучшения процессов, таких как Lean, «Шесть сигм», управление знаниями, обучение и т. д. В мире программного обеспечения им соответствуют XP, Agile и DevOps, но термин HPT обычно применяется к производству и не применяется к программному обеспечению.

его могут в чем-то обвинить, он будет очень неохотно говорить об этом, особенно если думает, что и правда что-то испортил. Это не означает, что мы не отвечаем за свои действия; мы должны отвечать. Просто настало время других подходов к ответственности. Слишком часто, когда люди говорят «Мне нужно привлечь сотрудников к ответственности», они на самом деле говорят, что им нужен кто-то, на кого можно свалить вину. Когда мы перестаем зацикливаться на вине и вместо этого задумываемся об изучении, улучшении и восстановлении, то формируем рабочий климат, в котором работники не боятся говорить о проблеме, даже если они спровоцировали эту проблему. Трудно не обвинять кого-нибудь, потому что мы люди, а люди исключительно хороши в поиске связей между вещами, даже если этот вывод не подкреплен объективными доказательствами. HOP не может предотвратить вину; но вина не сделает вас или вашу организацию лучше. Если ваша цель заключается в развитии, вам необходимо предпринять целенаправленные усилия, чтобы сместить фокус организации на обучение и совершенствование, избегая обвинений и наказаний.

18.2.3. Принцип 3: контекст определяет поведение

Окружающий работу контекст состоит из множества различных условий и компонентов. Некоторые из этих компонентов – системы, с которыми мы работаем ежедневно, такие как безопасность, качество, производство, окружающая среда, метрики и т. д. Выберите любую из них и посмотрите, какое поведение они навязывают. Если безопасность заставляет стремиться к нулевому количеству травм на производстве, это провоцирует на занижение отчетности. Чем больше система стремится снизить это число, тем меньше людей будут сообщать о своих травмах. Если контекст вашего производства требует сделать тысячу деталей к концу смены, то будет менее важно, как вы достигли этой цели, и более важно, сколько деталей вы сделали. Окружающий работу контекст часто определяет поведение, не связанное с целью бизнеса. Нам нужны эти системы, и нам нужны метрики; однако очень важно быть открытым и честным в отношении того, какое поведение навязывают эти системы. Если мы обнаружим, что они ведут себя правильно, это замечательно, если нет, мы должны придумать что-то другое. Нам также следует помнить, что контекст работы, который привел к нужному поведению вчера или в прошлом году, возможно, провоцирует нежелательное поведение сегодня. Например, мы реализуем программу наблюдения за производством, чтобы руководители подразделений всегда были в курсе того, что происходит. Очевидно, это хорошая вещь. Мы можем добавить метрику, чтобы следить, что они проводят хотя бы несколько наблюдений в неделю, то есть начнем наблюдать за наблюдающими. Мы сами не заметим, как в центре внимания окажется именно эта метрика, а не реальная работа.

18.2.4. Принцип 4: обучение и улучшение имеют жизненно важное значение

У многих из нас есть какой-то непрерывный процесс улучшения нашей организации. Он может работать хорошо; он может нуждаться в доработке. Методы, которые мы используем для обучения и совершенствования, должны быть полезны участникам и эффективны на практике, а не только в теории. Один из методов, которые мы создали для этого процесса обучения, называется «Учебная команда». Часть процесса обучения и совершенствования предусматривает проверку эффективности путем сверки результатов с заранее запланированными целями. В некоторых случаях это может быть даже не полное исправление недостатков, а скорее улучшение, основанное на том, как сейчас выглядит работа. Чтобы узнать, работает ли то, что мы создали, нам необходимо провести глубокое тестирование или анализ улучшений. Вот где подход хаос-инжиниринга действительно может пригодиться: проверка того, что система производит желаемый результат.

18.2.5. Принцип 5: важны осмысленные ответы

Последний из пяти принципов говорит о том, каким должен быть наш ответ. Когда случается или почти случается инцидент, когда мы не понимаем какой-либо процесс, или когда мы сомневаемся, не является ли успех результатом везения, мы должны дать тот или иной ответ. По сути, в этот момент мы должны выполнить оперативное обучение, которое приведет нас к более глубокому пониманию нашей работы и побудит нас к совершенствованию. Это нужно делать вдумчиво и рационально. Одновременно с поиском ответа необходимо удержать вашу организацию от обвинений в адрес работника, руководителя или менеджера. Помните: «Вина ничего не исправляет». Когда мы генерируем здравый ответ на какое-либо событие или проблему, тем самым мы задаем тон для окружающих. Это имеет большое значение на всех уровнях организации!

18.3. Хаос-инжиниринг в мире НОР

В моем производственном мире, когда возникает какая-то неполадка, будь то безопасность, качество или сбой рабочего процесса, мы хотим исправить ситуацию и сделать ее лучше. Более того, мы хотим быть уверены, что наши действия предприняты вовремя и они сработали. Обычно мы отслеживаем предпринятые действия, чтобы закрыть проблему, фиксируем имена исполнителей и даты выполнения, проводим совещания, на которых перечисляем незакрытые пункты. Звучит знакомо, не так ли? Я не говорю, что это плохо,

просто я часто наблюдаю, как основное внимание фокусируют на закрытии пунктов, а не на эффективности и отказоустойчивости, особенно если это считается вопросом соблюдения правил.

Этот процесс работает в большинстве организаций и приносит определенную пользу. Тем не менее я хочу сместить акцент на эффективность и отказоустойчивость решений, исправлений, улучшений и действий. Я хочу, чтобы моя команда и мое руководство были уверены, что сделанные улучшения действительно добавят больше надежности и устойчивости на рабочем месте. Действительно ли наши действия изменяют рабочую среду? Этот подход в значительной степени совпадает с методикой хаос-инжиниринга.

Я узнал об этом от Норы Джонс, Кейси Розенталя и их коллег. В своей программе хаос-инжиниринга они применяют несколько ключевых принципов изучения надежности их продуктов. Эти же принципы я хочу применить, чтобы лучше понять надежность и потенциал моих усилий по улучшению работы организации. Если вы хотите взять принципы хаос-инжиниринга и широко применить их к процессам целой организации, подумайте о том, сможет ли этот подход лучше, чем другие методы, охарактеризовать устойчивость и потенциал, которые вы формируете в своем офисе.

Давайте начнем с моего понимания философии экспериментов хаос-инжиниринга. Цель заключается не в том, чтобы покончить со стандартным тестированием продукта или процесса на соответствие нормативным требованиям и основным эксплуатационным критериям. Хаос-инжиниринг используется для того, чтобы вывести продукт или процесс за рамки обычных условий и посмотреть, узнаем ли мы что-то новое о надежности продукта или процесса, или, возможно, выявим отсутствие надежности, увидев, что система и впрямь погрузилась в «хаос». Мы экспериментируем, чтобы узнать что-то новое.

Экспериментировать, чтобы узнать что-то новое, а не просто проверить известное, – это великолепно.

Мне напомнили о простом примере применения этого принципа на практике. Допустим, в рамках программы по обучению и совершенствованию я решил, что нужно установить страховочное ограждение на рабочем месте. Я окрашиваю ограждение в оранжево-желтый цвет. Я закрепляю его анкерами в бетонном полу через анкерные пластины в нижней части каждой ножки (следя за тем, чтобы на каждую пластину было четыре болта). Ограждение должно быть 1 м в высоту и иметь среднюю перекладину и защитную накладку для ног. Оно должно выдерживать определенную боковую нагрузку и т. д. Ограждение должно быть проверено на соответствие всем перечисленным требованиям. Это «известные» требования. Хаос-инжиниринг, в свою очередь, спрашивает, насколько хорошо работает это ограждение при его использовании. Мешает ли оно? Блокирует ли оно доступ к клапанам или датчикам? Заставляет ли рабочих выходить в другие опасные зоны, чтобы обойти заграждение? А может, замедляет рабочий процесс? Что мы знаем о том, насколько хорошо работает ограждение в этом месте, когда происходят частые поломки оборудования, срабатывает аварийная сигнализация или проводится пожарная тренировка? Это вещи, которые более интересны,

чем просто соответствие нормативным требованиям или закрытие пунктов в трекере активности в течение 30 дней.

Когда мы вооружаемся подходом хаос-инжиниринга к системным экспериментам, защите и наращиванию потенциала, это сразу меняет положение дел. Когда я начал задумываться об этом новом подходе к пониманию продуктов и процессов, то понял, что это может стать настоящим переломным моментом в работе, которую мы выполняем с НОР. С помощью хаос-инжиниринга можно проверить улучшения и решения, придуманные учебными командами. После сессий нашей учебной группы мы знаем намного больше о хаосе работы. Вооружившись этой информацией, мы внедряем идеи по улучшению работы и можем их проверить. Если внесенные нами улучшения приведут процесс, продукт или операции к более устойчивому состоянию, наши знания о системе станут намного глубже. По общему признанию, во время работы над этой главой мы находимся лишь на ранних стадиях этого нового мышления, однако я хотел бы поделиться тем, что мы уже делаем, и идеями о том, что мы планируем делать.

18.3.1. Практический пример хаос-инжиниринга в мире НОР

Вот пример применения хаос-инжиниринга в мире НОР. Многие из компаний, с которыми мы работаем, имеют учебные залы по моделированию ситуаций. Они обычно располагаются как можно ближе к реальному залу с пультом управления. Операторы проводят бесчисленные часы в комнате моделирования ситуаций как во время обучения, так и в повседневной жизни. Тренеры могут вбросить новому оператору проблему и посмотреть, как он ее решает. Операторы терпят неудачу, учатся и пробуют снова. Это отличный способ развить компетентность и укрепить доверие к работе, которую они собираются выполнять. Допустим, на плате управления моделью генерируется учебная проблема, и оператор закрывает клапан 3 и направляет поток через обводную трубу в переливной бак. Оператор справился с учебной проблемой. В этой ситуации есть неочевидная обучающая ценность. Дело в том, что у нас есть симулятор, который делает то, что *должен* делать. Клапан 3 закрывается, и поток отводится в переливной бак.

Проблема в *реальной* жизни заключается в том, что клапан 3 заклинивает и закрывает только 80 % сечения трубы, что приводит к полному переключению потока, и значительная часть потока продолжит движение вниз по участку трубы, где возникла проблема. Это реальность, которой нет в симуляторе. Хаос-инжиниринг учит нас изменять параметры и программное обеспечение пульта управления имитацией, чтобы симуляция лучше представляла реальное положение дел. Теперь, когда новый оператор дает команду закрыть клапан 3, пульт показывает, что клапан закрыт; однако оператор все еще видит поток в трубе за клапаном. Теперь ему приходится в режиме реального времени искать другие варианты решения. Такая симуляция намного больше соответствует решению проблем в режиме реального времени,

когда система не делает то, что должна делать. Это хаос, присущий работе. Мы изучаем эту информацию на занятиях наших учебных групп и можем реализовать ее в программном обеспечении симулятора.

Я думаю, что мы сможем даже создать код, который будет рандомизировать проблемы для различных симуляций. Возможно, это будет какой-то алгоритм «износа», который имитирует износ разных компонентов и не дает им функционировать так, как изначально задумано. Это добавит ощущение хаоса реальной жизни в диспетчерской энергетической компании, или на химическом заводе, или в любом месте, где процесс контролируется компьютерными системами.

Теперь рассмотрим, как этот пример соотносится с принципами HOP.

Ошибка – это нормально

Мы ожидаем, что в смоделированных ситуациях стажеры будут принимать неоптимальные решения. Решать проблемы в реальном времени очень сложно, особенно когда вы еще нарабатываете опыт. Стажеры будут ошибаться, и это нормально.

Вина ничего не исправляет

В симуляции легче подавить стремление обвинять людей в решениях, которые не приводят к желаемому результату. Ставки ниже, потому что симуляция не оказывает существенного влияния на бизнес-результаты.

Контекст управляет поведением

Реалистичный симулятор формирует качественный контекст, и на основе взаимодействия между интерфейсом симуляции и поведением человека получается отличная учебная платформа. Достоверный контекст также помогает дизайнерам *пользовательского опыта* (user experience, UX) лучше понять взаимодействие между операторами и техническими частями системы. Несоответствие предпринятых действий (закрыть клапан 3) и видимого результата (поток все еще движется через проблемную трубу) информирует оператора о проблеме, а также говорит о возможности улучшить инструмент.

Обучение и улучшение имеют жизненно важное значение

Весь смысл моделирования заключается в обеспечении безопасной среды обучения. Это основа подхода HOP.

Важны осмысленные ответы

В реалистичной симуляции тренеры могут искать осмысленное поведение, поощрять его и поддерживать более глубокое обсуждение процесса принятия решений.

Цели хаос-инжиниринга и HOP прекрасно сочетаются друг с другом. Фокус внимания на эмпирических данных и стремление эксплуатировать систему в турбулентных условиях прямо соответствуют принципам HOP.

18.4. Вывод

Методика HOP разработана для того, чтобы собрать лучшее из того, что мы знаем о совместной работе людей и организаций, а затем создать лучшее рабочее место. HOP побуждает организации подвергать сомнению существующий порядок вещей, чтобы застраховаться от ложного чувства безопасности. Хаос-инжиниринг дополняет этот подход с философской и практической точек зрения, проверяя, что выходное состояние системы соответствует ожиданиям.

Об авторе

Боб Эдвардс – специалист по личной и корпоративной эффективности (HOP). Боб работает с персоналом на всех уровнях организации, преподавая основы HOP, формируя и тренируя обучающие команды. Боб получил степень бакалавра в области машиностроения в Технологическом университете штата Теннесси и степень магистра в области управления промышленной безопасностью в Университете Алабамы в Бирмингеме. Его опыт работы включает в себя должности специалиста по техническому обслуживанию, солдата в армии США, инженера-конструктора, руководителя службы технического обслуживания и технической поддержки, руководителя службы безопасности и помощника руководителя завода.

Глава 19

Хаос-инжиниринг и базы данных

Авторы главы: **Лю Тан** и **Хао Вэн**

19.1. Зачем нам нужен хаос-инжиниринг?

С тех пор как в 2011 году Netflix открыла исходный код Chaos Monkey, эта программа становится все более популярной. Если вы строите распределенную систему, то, позволяя Chaos Monkey слегка пошалить на вашем кластере, вы создадите более отказоустойчивую, отлаженную и безопасную систему[1].

TiDB – это распределенная база данных типа Hybrid Transactional/Analytical Processing (HTAP)[2] с открытым исходным кодом, разработанная главным образом PingCAP. Она хранит то, что мы считаем самым важным активом для любых пользователей базы данных: сами данные. Одним из основных и важнейших требований нашей системы является отказоустойчивость. Традиционно мы проводим модульные и интеграционные тесты, чтобы гарантировать, что система готова к производству, но они охватывают только верхушку айсберга, поскольку масштабы кластеров, сложность и объем данных на уровне производства значительно увеличиваются. Хаос-инжиниринг является для нас естественным инструментом. В этой главе мы подробно опишем нашу методологию и конкретные причины, по которым распределенная система, такая как TiDB, нуждается в хаос-инжиниринге.

19.1.1. Надежность и стабильность

Чтобы укрепить доверие пользователей к недавно выпущенной распределенной базе данных, такой как TiDB, где данные сохраняются в нескольких узлах, обменивающихся данными друг с другом, необходимо непрерывно бороться с потерями или повреждениями данных. Но в реальном мире отказы могут

---

[1] Частичное содержание этой главы было ранее опубликовано в блоге PingCAP.

[2] HTAP означает способность одной базы данных выполнять как оперативную обработку транзакций (OLTP), так и оперативную аналитическую обработку (OLAP) для обработки оперативных данных в реальном времени.

произойти в любое время и в любом месте, и мы не можем подготовиться заранее. Так как мы можем пережить их? Один из распространенных способов – сделать нашу систему отказоустойчивой. В случае сбоя одного сервиса другой сервис-фолловер может немедленно взять на себя ответственность, не влияя на онлайн-сервисы. На практике нам следует опасаться, что отказоустойчивость увеличивает сложность распределенной системы.

Как мы можем гарантировать, что наша отказоустойчивость заслуживает доверия? Типичные способы проверки нашей устойчивости к отказам включают написание модульных и интеграционных тестов. С помощью внутренних инструментов генерации тестов мы выполнили более 20 млн тестовых циклов. Мы также использовали большое количество тестовых примеров с открытым исходным кодом, таких как тесты MySQL и тесты фреймворка ORM. Тем не менее даже 100%-ное дублирование элементов не создает отказоустойчивую систему. Аналогично система, выдерживающая хорошо спроектированные интеграционные тесты, не обязательно будет работать достаточно устойчиво в реальной производственной среде. В реальном мире может произойти все, что угодно, например сбои диска или рассинхронизация протокола сетевого времени (NTP). Чтобы повысить устойчивость системы распределенных баз данных, такой как TiDB, нам нужен метод моделирования непредсказуемых отказов и проверки наших ответов на эти отказы.

19.1.2. Пример из реального мира

В TiDB мы используем алгоритм консенсуса Raft для репликации данных от лидера к фолловерам, чтобы гарантировать согласованность данных между репликами. Когда фолловер добавляется в группу реплик, велика вероятность, что он будет отставать от лидера на несколько версий. Для обеспечения согласованности данных лидер отправляет фолловеру снимок текущего состояния базы данных. Это место, где все может пойти не так. На рис. 19.1 показан типичный случай, с которым мы столкнулись в производственной среде.

```
[root@10-180-0-22 data]# ls -lt snap/* | grep 16986
-rw-r--r-- 1 ops ops       58 Jul 18 23:42 snap/rev_1129386_18_16986.meta
-rw-r--r-- 1 ops ops        0 Jul 18 23:42 snap/rev_1129386_18_16986_write.sst
-rw-r--r-- 1 ops ops        0 Jul 18 23:42 snap/rev_1129386_18_16986_lock.sst
-rw-r--r-- 1 ops ops  8499200 Jul 18 23:42 snap/rev_1129386_18_16986_default.sst
```

Рис. 19.1 ❖ Реальная ошибка, найденная в снимке TiDB. Файлы *_write.sst* и *_lock.sst* не должны иметь размер 0 байт в соответствии с информацией в файле *.meta*

Как видно на рис. 19.1, снимок состоит из четырех частей: одного мета-файла (с расширением .meta) и трех файлов данных (с расширением .sst). Мета-файл содержит информацию о размере всех файлов данных и соответствующие контрольные суммы, которые мы можем использовать, чтобы проверить, действительны ли полученные файлы данных.

На рис. 19.2 показано, как согласованность снимков вновь созданной реплики проверяется лидером и фолловерами Raft. Как вы можете видеть в журнале, некоторые размеры равны нулю, но на самом деле они не равны нулю в метафайле. Это означает, что снимок поврежден.

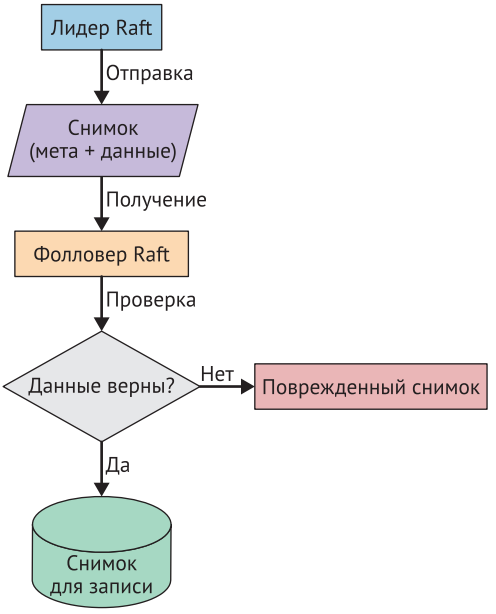

Рис. 19.2 ❖ Проверка согласованности снимка в группе Raft. Лидер отправляет снимок фолловеру, и далее метафайл сравнивается с файлами данных, чтобы определить, соответствуют ли они друг другу

Так как же произошла эта ошибка? В отладочном сообщении Linux мы обнаружили ошибку в ядре Linux:

```
[17988717.953809] SLUB: Unable to allocate memory on node -1 (gfp = 0x20)]
```

Ошибка произошла, когда Linux работал с кешем страниц[1], работающим как основной кеш диска, на который ссылается ядро при чтении или записи на диск. Когда мы записываем данные в файл Linux без использования режима прямого ввода-вывода, данные сначала будут сохраняться в кеш страницы, а затем записываться на диск через фоновый поток. В случае сбоя процесса очистки из-за отказов системы или отключения электроэнергии мы можем потерять записанные данные.

Этот сбой был хитрым, потому что он не был изолирован от самого TiDB. С ним можно столкнуться только в полном контексте производственной среды, которая имеет большую сложность и непредсказуемость. Решить эту конкретную проблему легко, но независимо от того, сколько модульных или

[1] *Marco Cesati and Daniel P. Bovet.* Understanding the Linux Kernel. Third Edition. Sebastopol, CA: O'Reilly, 2005. Chapter 15.

интеграционных тестов мы напишем, мы все равно не можем охватить все случаи. Нам нужен лучший способ: хаос-инжиниринг.

19.2. Применение хаос-инжиниринга

Netflix не только изобрел инструмент Chaos Monkey, но и представил концепцию хаос-инжиниринга – методику обнаружения скрытых отказов. Мы объединили с нашим особым подходом следующие рекомендации по хаос-экспериментам (с сайта «Принципов хаоса»):

- определите «устойчивое состояние» как некоторый измеримый выход системы, указывающий на нормальное поведение;
- разработайте гипотезу, основанную на устойчивом состоянии;
- введите переменные факторы, которые отражают реальные инциденты;
- опровергните гипотезу, идентифицировав отклонения от стационарного состояния как отказы.

19.2.1. Наш особый подход к хаос-инжинирингу

В TiDB мы применяем хаос-инжиниринг для наблюдения за устойчивым состоянием нашей системы, выдвигаем гипотезу, проводим эксперименты и проверяем нашу гипотезу на реальных результатах[1]. Вот наша пятиэтапная методология хаос-инжиниринга, основанная на базовых принципах.

1. Определите устойчивое состояние на основе показателей. Мы используем Prometheus в качестве монитора и определяем устойчивое состояние системы, наблюдая и собирая критические метрики стабильного кластера. Обычно мы используем QPS и время ожидания (P99/P95), нагрузку процессора и выделение памяти. Это ключевые показатели качества обслуживания для распределенной базы данных, такой как TiDB.
2. Составьте список гипотез определенных сценариев отказов и того, что вы ожидаете; например, если мы изолируем узел TiKV (слой распределенного хранилища значений ключей TiDB) от кластера с тремя репликами, QPS должен сначала уменьшиться, но вскоре восстановиться до иного стабильного состояния. Другой пример – мы увеличиваем количество регионов (единица сегментации хранилища в TiKV) до 40 000 на одном узле. Загрузка процессора и памяти должна оставаться нормальной.
3. Выберите гипотезу для проверки.
4. Внедрите отказы в систему и проверяйте, наблюдается ли изменение показателей. Если есть значительные отклонения от стационарного состояния, должно быть, что-то не так. Возвращаясь к гипотезе QPS (рис. 19.3), в случае отказа узла TiKV, если QPS никогда не возвращается к нормальному уровню, это означает, что неисправные лидеры, вызванные разделом кластера, никогда не переизбираются, либо клиент постоянно запрашивает ответ от пропавшего лидера. Оба случая указывают на ошибки или даже конструктивные дефекты в системе.

[1] Более подробная информация о шаблоне для экспериментов приведена в главе 3.

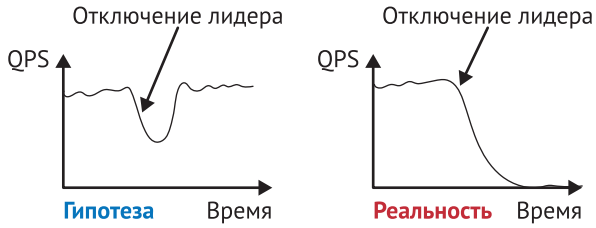

Рис. 19.3 ❖ Гипотеза и реальность

Вернитесь к исходному состоянию и проверьте еще одну гипотезу из вашего списка. Автоматизируйте процесс с помощью тестовой среды под названием Schrodinger.

19.2.2. Внедрение отказов

Внедрение отказов – это метод улучшения охвата теста путем введения ошибок в пути выполнения кода теста, в частности в пути кода обработки ошибок. Работая с TiDB, мы накопили много способов выполнить внедрение отказов, чтобы нарушить работу системы и лучше понять ее сложность. Выполняя внедрение отказа, важно изолировать части системы и правильно определить компоненты, чтобы минимизировать радиус поражения. Исходя из области воздействия, мы разделяем наши методы внедрения отказов на четыре основные категории:

- отказы приложений;
- ошибки процессора и памяти;
- отказы сети;
- ошибки файловой системы.

19.2.3. Отказы приложений

На уровне приложения уничтожение или приостановка процесса – это хороший метод для проверки возможностей отказоустойчивости и параллельной обработки (табл. 19.1).

Таблица 19.1 Внедрение отказа в приложение

Назначение	Метод/шаг
Отказоустойчивость/восстановление	Произвольно устранить процесс принудительно (с помощью SIGKILL) или деликатно (с помощью SIGTERM) и перезапустить его. Остановить процесс с помощью команды SIGSTOP, а затем возобновить его с помощью SIGCONT
Ошибки параллелизма	Используйте `renice`, чтобы изменить приоритет процесса. Используйте `pthread_setaffinity_np`, чтобы изменить привязку потока

19.2.4. Ошибки процессора и памяти

Поскольку ЦП и память тесно связаны друг с другом и оба имеют непосредственное влияние на работу потоков и производительность, вполне логично, что мы помещаем ЦП и память в одну категорию. В табл. 19.2 перечислены шаги и цели внедрения отказов ЦП и памяти.

Таблица 19.2 Ошибки процессора и памяти

Назначение	Метод/шаг
Проблемы с насыщенностью и производительностью	Запустите такие приемы, как цикл `while (true){}`, чтобы максимально использовать процессор (загрузка 100 %)
Производительность в ограниченных условиях	Используйте `cgroup` для управления процессором и выделением памяти для определенного процесса

Процессор и память – это общая цель для внедрения отказа в любой системе. Поскольку мы создаем распределенную базу данных, в которой данные сохраняются на нескольких машинах и перемещаются между ними, мы уделяем основное внимание внедрению отказов в сеть и файловую систему.

19.2.5. Отказы сети

Проанализировав 25 известных систем с открытым исходным кодом, Ахмед Алкураан и др.[1] выявили 136 неисправностей, связанных с нарушением связности сети. При этих неисправностях 80 % являются катастрофическими, причем потеря данных выступает наиболее распространенной категорией (27 %). Это особенно актуально для распределенной базы данных. К нарушению связности сети не следует относиться легкомысленно. Благодаря внедрению отказов сети мы можем заранее обнаружить как можно больше сетевых проблем, чтобы укрепить наше понимание и доверие к базе данных до ее развертывания в производстве.

Существует три типа нарушения связности сети, как показано на рис. 19.4:

a) полная сегментация: связь между группой 1 и группой 2 полностью разорвана;
b) частичная сегментация: группы 1 и 2 не могут общаться напрямую, но они могут общаться через группу 3;
c) симплексная сегментация: группа 1 может подключаться к группе 2, но группа 2 не может подключаться к группе 1.

Для проверки TiDB мы не только используем базовые типы нарушения связности для имитации сетевых сбоев, но также добавляем несколько других вариантов:

- применение `tc`[2] для увеличения задержки в сети;

---

[1] *Ahmed Alquraan et al.* An Analysis of Network-Partitioning Failures in Cloud Systems // 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), USENIX Association.

[2] tc (traffic control) – служебная программа пользовательского пространства, исполь-

- использование `tc` для изменения порядка сетевых пакетов;
- запуск специального приложения, чтобы выйти из полосы пропускания;
- применение прокси для управления определенным TCP-соединением;
- использование `iptable` для ограничения определенных соединений.

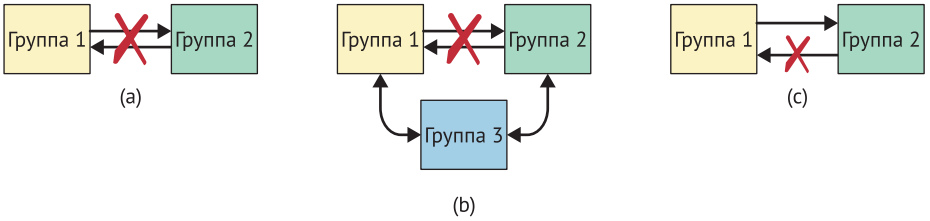

Рис. 19.4 ❖ Типы дробления сети

С помощью этих методов для ввода неисправностей в сеть мы можем обнаружить основные проблемы в сети, в основном связанные с распределенной базой данных, такие как задержка, потеря пакетов, нарушение связности сети и т. д. Конечно, это не все условия, которые мы исследуем; например, иногда мы отключаем сетевой кабель, чтобы вызвать немедленное отключение от сети в течение определенного периода времени.

19.2.6. Внедрение ошибок в файловую систему

Пиллаи с коллегами[1] обнаружили, что файловая система может вызывать нарушение целостности данных в результате сбоев, например проблему снимка, о которой мы упоминали ранее. Чтобы лучше понять файловую систему и защитить данные от сбоев файловой системы, нам также необходимо провести эксперименты с этим хаосом.

Поскольку трудно внедрить ошибки в файловую систему напрямую, мы используем Fuse (рис. 19.5) для монтирования каталога и позволяем нашему приложению обрабатывать данные в этом каталоге. Любая операция ввода-вывода вызовет срабатывание триггера, поэтому мы можем выполнить внедрение отказа, чтобы вернуть ошибку или просто передать операцию в реальный каталог.

В блоке внедрения отказа мы определяем правила, например для пути /a/b/c устанавливаем задержку 20 мс на каждую операцию чтения/записи; или для пути /a/b/d выполняем действие `return NoSpace error` для каждой операции записи. Отказы вводятся в соответствии с этими правилами через

зуемая для настройки планировщика пакетов ядра Linux.

[1] *Thanumalayan Sankaranarayana Pillai et al.* All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications // Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, October 2014.

смонтированный каталог. Операции, которые не соответствуют отказу, обходят эти правила и взаимодействуют с реальным каталогом.

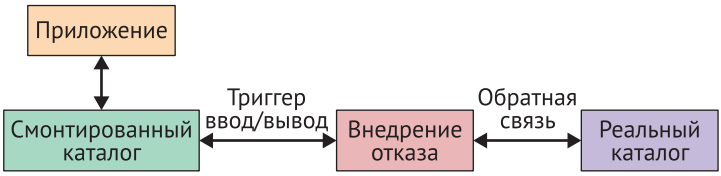

Рис. 19.5 ❖ Архитектура Fuse

19.3. Обнаружение сбоев

Разработка экспериментальных гипотез и внедрение ошибок в системы высвечивают проблемы, но они – только начало процесса, направленного на понимание сложности и непредсказуемости системы. Чтобы эксперимент действительно работал, нам нужны методы для эффективного и точного обнаружения сбоев в производстве. Иначе говоря, нам нужен инструмент, способный автоматически обнаруживать сбои в работе.

Простой способ обнаруживать сбои – использовать механизм оповещения Prometheus. Мы можем настроить некоторые правила и получать оповещения, когда что-то идет не так и вызывает срабатывание правила. Например, когда заданное количество ошибок в секунду превышает предварительно определенный порог, срабатывает предупреждение, и мы можем реагировать соответствующим образом.

Другой способ – извлечь уроки из истории. У нас есть показатели пользовательской нагрузки, накопленные на большом интервале времени. Основываясь на этих исторических данных, мы можем сделать вывод о нормальности текущих показателей, таких как скачок в продолжительности сохранения, потому что пользовательская нагрузка фиксируется большую часть времени (рис. 19.6).

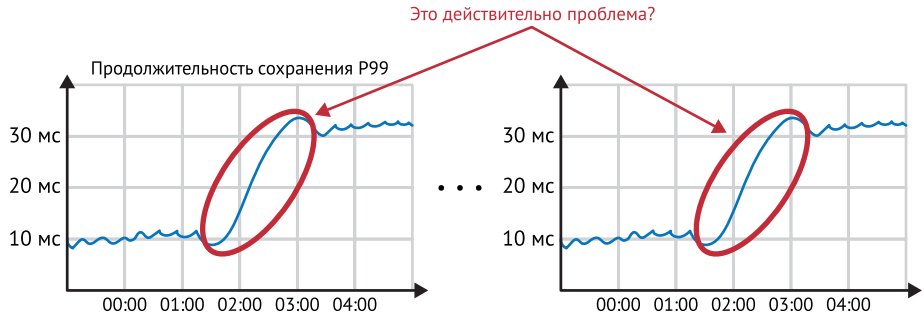

Рис. 19.6 ❖ Индикатор на основе истории

Для критических ошибок мы используем Fluent Bit, процессор журналов с открытым исходным кодом и сервер пересылки для сбора журналов между компонентами TiDB и анализа журналов перед дальнейшей обработкой и отладкой в Elasticsearch. Чтобы представить сбор, анализ и запрос журналов в структурированном виде, мы определили унифицированный формат журналов, называемый TiDB Log Format, который структурирован следующим образом:

```
Log header: [date_time] [LEVEL] [source_file:line_number]
Log message: [message]
Log field: [field_key=field_value]
```

Вот пример журнала:

```
[2018/12/15 14:20:11.015 +08:00] [WARN] [session.go:1234]
[“Slow query”]
[sql=”SELECT * FROM TABLE WHERE ID=\”abc\””] [duration=1.345s]\n
[client=192.168.0.123:12345] [txn_id=123000102231]
```

В этом примере присутствует ключ «`Slow query`». Из этой части сообщения журнала мы можем узнать соответствующий оператор SQL и его `txn_id` (уникальный идентификатор SQL-запроса), на основании которого можем получить все связанные журналы и узнать, почему SQL работает медленно.

19.4. Автоматизация хаоса

В 2015 году, когда мы впервые начали разрабатывать TiDB, каждый раз, добавляя функцию, мы делали следующее:

1) создавали двоичный файл TiDB;
2) просили администратора выделить несколько машин для тестирования;
3) развертывали двоичные файлы TiDB и запускали их;
4) запускали инструменты тестирования;
5) внедряли отказы;
6) удаляли файлы и освобождали машины после завершения всех испытаний.

Хотя этот механизм нормально работал, он включал утомительные ручные операции. По мере роста кодовой базы TiDB и количества пользователей приходилось проводить все больше и больше экспериментов одновременно. Ручной способ просто не мог масштабироваться. Нам понадобился автоматический конвейер, который решает эту проблему.

19.4.1. Автоматизированная платформа для экспериментов Schrodinger

Знаменитый мысленный эксперимент с котом Шредингера представляет нам воображаемого кота, который может считаться одновременно живым и мертвым, поскольку его жизнь связана с субатомным событием, которое

может происходить или не происходить. Мы решили, что заложенная в эксперимент непредсказуемость и устройство, которое ее вызывает, прекрасно применимы к нашей методике хаос-инжиниринга. Вдохновившись этой идеей, мы создали Schrodinger – экспериментальную платформу, которая автоматически выполняет хаос-эксперименты. Все, что от нас требуется, – это написать эксперименты и настроить Schrodinger для выполнения конкретных задач тестирования, а он сделает все остальное.

Одной из самых больших проблем при проведении этих экспериментов было создание среды на физических машинах буквально с чистого листа. Нам требовалось техническое решение для проведения наших хаос-экспериментов, которое могло бы покончить с этой проблемой, чтобы мы могли сосредоточиться на том, что важно: на понимании наших систем. Schrodinger основан на Kubernetes (K8s), поэтому мы независим от физических машин. K8s скрывает детали на уровне машины и помогает нам планировать правильную работу для правильных машин.

Как показано на рис. 19.7, Schrodinger состоит из следующих компонентов:

Кот (Cat)
: Кластер TiDB с заданной конфигурацией.

Коробка (Box)
: Шаблон для генерации конфигураций кластера и связанных с ним экспериментов; это инкапсуляция эксперимента или тест для запуска.

Немезида (Nemesis)
: Инжекторы отказов, которые вводят отказы, чтобы нарушить работу системы, с целью «убить кота» или провалить тест.

Форматы теста
: Определяет процедуру тестирования, входные данные и ожидаемые результаты.

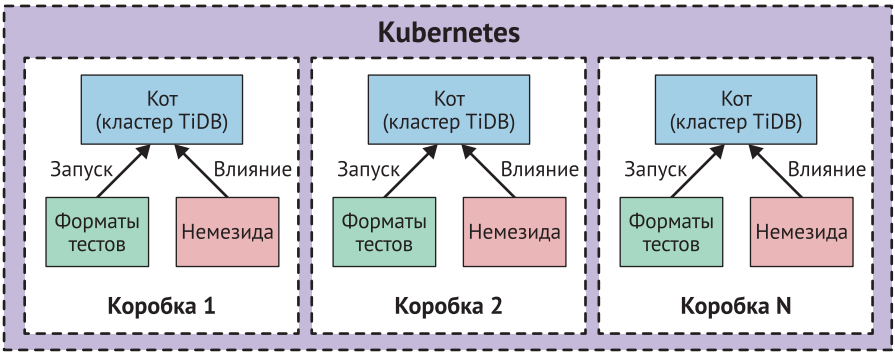

Рис. 19.7 ❖ Архитектура Шредингера для K8c

Платформа Schrodinger оптимизирует методологию хаос-инжиниринга и позволяет автоматически масштабировать эксперименты по мере необходимости.

19.4.2. Рабочий процесс на платформе Schrodinger

Чтобы запустить эксперименты с недавно разработанными функциями на платформе Schrodinger, нам нужно выполнить несколько простых действий.

1. Подготовьте контрольный пример:
   а) загрузите тестовый код с помощью `git clone`, скомпилируйте и укажите рабочие параметры;
   б) укажите последовательность выполнения для нескольких экспериментов, например последовательных или параллельных, и определите, следует ли вводить ошибки с использованием Немезиды случайно или на основе сценариев эксперимента.
2. Создайте Кота. В нашем случае Кот – это кластер TiDB, который мы хотим протестировать. Установите количество различных компонентов TiDB в кластере, ветвь кода и конфигурации компонента в файле конфигурации.
3. Добавьте Коробку и поместите в нее настроенный кластер TiDB и контрольные примеры.

После того как мы завершим эти шаги, Schrodinger начинает готовить ресурс, собирать соответствующий релиз, развертывать и запускать кластер. Затем он запустит эксперимент (с внедрением ошибок Немезиды или нет) и в конце даст нам отчет.

До появления платформы Schrodinger даже для такого простого эксперимента, как перенос учетной записи, нам нужно было вручную развернуть кластер TiDB, настроить тест, внедрить отказы и, наконец, обнаружить ошибки. Благодаря платформе Schrodinger, будь то простой эксперимент, подобный этому, или гораздо более сложный, эти шаги могут выполняться автоматически в несколько кликов мышью. Теперь Schrodinger может проводить эксперименты в семи разных кластерах одновременно, круглосуточно и без остановки.

19.5. Вывод

Наша платформа хаос-экспериментов Schrodinger помогает нам более эффективно определять проблемы во всех компонентах TiDB, включая сторонние приложения, такие как RocksDB. Мы твердо верим, что хаос-инжиниринг – это отличный способ выявления систематической неопределенности в распределенной системе и повышения уверенности в надежности системы.

Мы планируем и дальше постоянно расширять реализацию, делая нашу платформу более универсальной, умной и автоматизированной. Например, мы хотим иметь возможность внедрять ошибки кода на уровне ядра и использовать машинное обучение, чтобы Schrodinger «изучил» журнал истории кластера и выяснил, как интеллектуально внедрить ошибку. Кроме того, мы также рассматриваем возможность предоставления инструментов

Schrodinger в качестве службы экспериментов с помощью хаос-оператора Chaos-Mesh[1] или хаоса как *определяемого пользователем ресурса* (customer resource definition, CRD) через K8s, чтобы больше пользователей за пределами PingCAP могли находить свои проблемы с помощью наших методологий, просто предоставляя свои собственные шаблоны K8s.

Об авторах

Лю Тан работает главным инженером в PingCAP. Он был руководителем группы и ответственным за эксплуатацию проекта TiKV с момента его запуска в 2015 году. Он также давний сторонник и практик хаос-инжиниринга. Помимо работы в PingCAP, является энтузиастом открытого исходного кода и автором библиотек go-ycsb и ledisdb.

Хао Вэн – контент-стратег и координатор проекта по международному развертыванию проекта TiKV в PingCAP. Он имеет многолетний опыт работы техническим писателем в таких технологических компаниях, как Spirent и Citrix. Помимо рассказов о сложных технологиях понятным языком, он увлекается марафонами и мюзиклами.

[1] К моменту публикации книги компания PingCAP уже опубликовала открытый исходный код Chaos Mesh, облачной платформы хаос-инжиниринга для оркестровки хаос-экспериментов на K8s.

Глава 20

Хаос-инжиниринг в информационной безопасности

Автор главы: **Аарон Райнхарт**

Определение хаос-инжиниринга в информационной безопасности: выявление сбоев контроля безопасности с помощью упреждающих экспериментов, чтобы укрепить уверенность в способности системы защищаться от вредоносного воздействия в производственной среде[1].

Согласно Privacy Rights Clearinghouse, организации, которая отслеживает утечки данных, частота инцидентов безопасности, а также количество скомпрометированных учетных записей пользователей растут экспоненциально. Неправильное применение базовых конфигураций и соответствующих технических средств управления служит причиной множества инцидентов[2]. Организациям приходится прикладывать большие усилия, просто чтобы сохранить статус-кво безопасности. Между методологией обеспечения безопасности и подходом к построению систем существует постоянное противоречие.

Необходимость найти другие подходы к информационной безопасности приобрела первостепенное значение, поскольку безопасность не успевает за движением в сторону сложных распределенных систем. Мы достигли уровня развития технологий, при котором человеческий ум не в состоянии мысленно моделировать системы, которые мы разрабатываем. Новые технологические прорывы, такие как облачные вычисления, микросервисы и непрерывная доставка (CD), привели к появлению новой ценности для клиентов, но, в свою очередь, и к новой веренице проблем. Главной проблемой является наша неспособность понять наши собственные системы.

Если мы плохо понимаем, как ведут себя наши системы в целом, как мы можем добиться от них приемлемого уровня безопасности? При помощи

---

1 *Aaron Rinehart.* Security Chaos Engineering: A New Paradigm for Cybersecurity // Opensource.com, Jan. 24, 2018, https://oreil.ly/Vqnjo.

2 2018 Cost of a Data Breach Report / IBM/Ponemon Institute, 2018, https://oreil.ly/sEt6A.

запланированных познавательных экспериментов. В этой главе мы говорим о хаос-инжиниринге с позиций кибербезопасности. Мы называем это *хаос-инжинирингом безопасности* (security chaos engineering, SCE).

SCE служит культурным фундаментом для обучения тому, как строить, эксплуатировать, оборудовать и защищать свои системы. Цель этих экспериментов – перевести прикладную безопасность из субъективного ощущения в объективную оценку. Как и в мире DevOps, хаос-эксперименты позволяют командам по безопасности сократить количество «неизвестных неизвестных» и заменить «известные неизвестные» полезной информацией, которая способствует улучшению безопасности.

Преднамеренно вводя режим отказа или другое событие, команды по безопасности могут оценить подлинную оснащенность, наблюдаемость и измеримость систем безопасности. Команды могут видеть, работают ли функции безопасности так хорошо, как задумано, то есть объективно оценить способности и слабости, а затем укрепить первое и устранить второе.

SCE предполагает, что единственный способ понять эту неопределенность – объективно противостоять ей путем введения управляемых отказов. Вводя в систему объективный контролируемый сигнал, вы получаете возможность измерить такие вещи, как качество работы команды с различными типами инцидентов, насколько эффективна технология, насколько согласованно работают модули или процессы обработки инцидентов безопасности и т. д. Теперь вы можете по-настоящему понять, когда произошел инцидент, измерить, отследить и сравнить результаты за разные периоды времени и даже заранее подготовить разные команды к атаке.

20.1. Современный подход к безопасности

Хаос-инжиниринг – единственный общепринятый проактивный механизм для обнаружения инцидентов доступности до того, как они произойдут. В свою очередь, SCE позволяет заблаговременно и безопасно обнаруживать слабые места в системе, *прежде* чем они навредят бизнесу. Это требует принципиально нового подхода к кибербезопасности, идущего в ногу с быстро развивающимся миром разработки программного обеспечения.

20.1.1. Человеческий фактор и отказы

В кибербезопасности определение «первопричины» по-прежнему является широко распространенной культурной нормой.

> То, что вы называете «первопричиной», – это просто место, где вы перестаете искать дальше.
>
> – *Сидни Деккер*[1]

[1] *Sydney Dekker.* The Field Guide to Understanding «Human Error». 3rd ed. Abingdon and New York, NY: Routledge, 2014.

Не существует единой причины отказа, так же, как нет единой причины успеха. Подход *анализа первопричин* (root cause analysis, RCA) приводит к ненужному и бесполезному распределению ответственности, изоляции занятых специалистов и в конечном итоге к культуре страха в организации.

Без ошибок не обойтись. Вместо того чтобы устраивать быстрые и познавательные эксперименты, традиционный подход RCA фокусируется на «посмертном» анализе инцидентов. Такой подход мешает более масштабному пониманию того, какие события и действия могли способствовать инциденту. В конце концов, RCA не уменьшает количество или серьезность дефектов безопасности в наших продуктах. Наше текущее мышление и процессы лишь усугубляют проблему, а не решают ее.

Стремление анализировать события постфактум означает, что мы охотно используем «первопричину» в качестве объекта для приписывания и перекладывания вины. Предвзятость часто подменяет истину нашим личным представлением, а истина – это объективный факт, который мы, исследователи, никогда не сможем полностью познать. Склонность к саморефлексии, различные человеческие слабости и вечная нехватка ресурсов еще больше стимулируют эту порочную модель.

Большинство известных «первопричин» утечки данных[1] не связаны со злонамеренной или преступной деятельностью. Институт Ponemon/IBM определяет «злонамеренные атаки» как «действия, совершаемые хакерами или инсайдерами (сотрудниками, подрядчиками или другими третьими лицами)». Другими словами, определение «злонамеренных или криминальных атак» может быть довольно широким и включать атаки, проводимые союзниками и/или вражескими национальными государствами, хактивизм, организованную преступность, кибертерроризм, корпоративный шпионаж и другие акты криминального свойства. Тем не менее если «человеческие факторы» и «системные сбои» находятся в центре внимания и часто упоминаются как первопричины утечки данных, то криминальную деятельность редко называют первопричиной. Почему?

В одной из статей, рассуждая о продвинутости киберпреступников, BBC пишет[2]:

> Подобные атаки случаются. Но чаще всего хакеры и киберпреступники, попавшие в заголовки газет, не делают ничего особенного. На самом деле они нередко просто коварные приспособленцы – как и все преступники.

Реальность такова, что подавляющее большинство вредоносного кода, такого как вирусы, вредоносные программы, вымогатели и т. п., обычно использует в своих интересах так называемые «низко висящие фрукты»: слабые пароли, пароли по умолчанию, устаревшее программное обеспечение, незашифрованные данные, слабые меры безопасности в системах, но прежде всего они пользуются отсутствием у ничего не подозревающих людей

---

1 2018 Cost of a Data Breach Report / IBM/Ponemon Institute.

2 *Chris Baraniuk*. It's a Myth that Most Cybercriminals Are 'Sophisticated // BBC.com, July 26, 2017, https://oreil.ly/qA1Dw.

понимания того, как на самом деле работает сложная система перед ними. Наша отрасль нуждается в новом подходе.

20.1.2. Устраните легкодоступные цели

Если большая часть вредоносного кода предназначена для того, чтобы охотиться на ничего не подозревающих, плохо подготовленных или необразованных людей, которых в мире кибербезопасности называют «низко висящими фруктами», то имеет смысл спросить: сколько криминальных атак было бы успешными, не будь такого изобилия легкодоступных целей?

Что, если «низко висящий фрукт» и правда самый сладкий? Возьмем только что описанную ситуацию, в которой преступники охотятся за слабостями в виде неудачного стечения обстоятельств, ошибок и непонимания. Не является ли этот «фрукт» ключом к опережающему пониманию того, как ведут себя наши системы и люди, которые их строят и эксплуатируют?

Если мы будем исходить из предположения, что люди и системы ведут себя непредсказуемо, то, возможно, будем действовать иначе и иметь более полезные взгляды на поведение системы. Может оказаться так, что присущие социотехническим экосистемам ошибки, с которыми мы работаем, можно обратить на пользу системе. Предположите, что произошел отказ, и спроектируйте систему, готовую к указанным отказам.

Мы должны сосредоточиться на нашей способности учиться на ошибках, потому что теперь ожидаем, что это наше новое рабочее состояние. Благодаря этому изменению мышления мы можем начать понимать, что нужно для построения более устойчивых систем. Создавая более отказоустойчивые системы, вместо того чтобы пытаться поймать все ошибки, мы заставляем неискушенных преступников и злоумышленников больше работать за меньшие деньги.

Институт Ponemon/IBM упоминает системные сбои в качестве одного из способствующих факторов: «Системные сбои включают сбои приложений, непреднамеренные потери данных, логические ошибки при передаче данных, ошибки идентификации или аутентификации (неправильный доступ), ошибки восстановления данных и многое другое».

Неумолимая реальность заключается в том, что сбои – это *нормальное* поведение наших систем. Несомненно, неудачи становятся неприятным сюрпризом, но по-настоящему удивительно то, что наши системы вообще работают с самого начала. Если системные сбои, отказы и прочие сюрпризы случаются настолько часто, что считаются нормой, значит, работоспособность наших систем балансирует на грани хаоса каждый день.

Занимаясь реагированием на сбои постфактум, индустрия безопасности упускает возможность использовать инциденты для упреждающего повышения устойчивости системы. Что, если бы можно было заранее распознать инцидент до того, как он произойдет? Что, если бы мы не полагались только на надежду, а вместо этого активно и целенаправленно подходили к безопасности?

20.1.3. Петли обратной связи

Даже несмотря на то, что современное программное обеспечение становится все более распределенным и с короткими итерациями, подход к безопасности остается преимущественно превентивным и зависит от момента времени. В современных практиках безопасности отсутствуют быстрые итеративные петли обратной связи, которые успешно работают в доставке современного продукта. Должны существовать такие же петли обратной связи между изменениями в средах продукта и механизмами, используемыми для их защиты.

Меры безопасности должны быть итеративными и достаточно гибкими, чтобы изменять свое поведение так же часто, как и программная экосистема, в которой они работают. Элементы безопасности обычно разрабатываются с учетом конкретного состояния (то есть выпуска продукта в «нулевой день»). Между тем экосистема, которая окружает эти элементы безопасности, быстро меняется каждый день. Микросервисы, машины и другие компоненты живут своей жизнью. Благодаря непрерывной доставке изменения компонентов происходят несколько раз в день. Внешние API-интерфейсы постоянно меняются в зависимости от их собственных графиков доставки и т. д.

> Чтобы улучшить безопасность, важно оценить, что вы делаете хорошо, и научиться *делать меньше, но лучше*.
>
> – *Чарльз Нвату*,
> инженер по безопасности Netflix (бывший CISO Stitch Fix)

Чарльз Нвату описывает SCE как механизм, позволяющий организациям активно оценивать эффективность своих мер безопасности. Независимо от того, происходит ли это из-за растущих нормативных требований или изменяющегося ландшафта атак, специалистов по обеспечению безопасности просят создавать, эксплуатировать и поддерживать постоянно растущее число мер безопасности. Как бывшему начальнику управления информационной безопасности (chief information security officer, CISO) в компании Stitch Fix Чарльзу было поручено создать надежную систему кибербезопасности. В процессе создания аппарата безопасности компании он заявил о необходимости «делать меньше, но лучше», вместо того чтобы слепо вводить меры безопасности буква за буквой. Его девиз «делай меньше, но лучше» отражает его желание проактивно и постоянно проверять, что разрабатываемые меры безопасности («делай меньше») действительно эффективны при выполнении намеченных функций («лучше»).

Чарльз прекрасно понимал, что инструменты и методы безопасности должны быть достаточно гибкими, чтобы соответствовать постоянным изменениям и итерациям окружающей среды. Без петли обратной связи система безопасности рискует дрейфовать в состояние неизвестной уязвимости, так же как система разработки без петли обратной связи может перейти в состояние неготовности к эксплуатации.

Наиболее распространенный способ обнаружения дефектов безопасности – это наблюдение за реальными инцидентами. Однако инциденты

безопасности не являются эффективными сигналами, потому что в этот момент уже слишком поздно. Ущерб уже нанесен. Если мы стремимся активно обнаруживать сбои в системе безопасности, нам придется найти инструменты получше, чем простое наблюдение.

SCE обеспечивает наблюдаемость и строгие эксперименты, подтверждающие безопасность системы. Наблюдаемость имеет решающее значение для создания петли обратной связи. Тестирование – это валидация или бинарная оценка ранее известного результата. Мы знаем, что мы ищем, прежде чем искать это. Эксперименты извлекают новые идеи и информацию, которые ранее были неизвестны. Эти новые идеи завершают цикл обратной связи и продолжают обучение. Это более высокий уровень развития безопасности.

Внедрение событий безопасности помогает командам глубже понять функционирование своих систем и найти новые способы повышения устойчивости. Ожидается, что SRE[1], группы разработчиков продуктов и группы безопасности будут вместе внедрять систему безопасности. Постоянно проводя эксперименты по безопасности, мы можем развить наше понимание новых уязвимостей, прежде чем они превратятся в кризис. При правильной реализации SCE становится петлей обратной связи, информирующей о состоянии системы безопасности.

20.2. Хаос-инжиниринг и новая методология безопасности

SCE устраняет ряд пробелов в современных методологиях безопасности, таких как упражнения Red and Purple Team («красная» и «фиолетовая» команды). Мы не намерены упускать из виду ценность командных упражнений Red and Purple или других методов тестирования безопасности. Эти методы остаются ценными, но различаются с точки зрения целей и методов. В сочетании с SCE они обеспечивают более объективный и упреждающий механизм обратной связи для подготовки системы к нежелательному событию, чем при ее реализации в одиночку.

Методология Red Teaming[2] родилась в вооруженных силах США[3]. За прошедшие годы ей давали разное определение, но сегодня ее можно охарактеризовать как «состязательный подход, который максимально реалистично имитирует поведение и приемы злоумышленников». На предприятиях встречаются две распространенные формы Red Teaming: этические взломы и тестирование на проникновение, которые часто используют сочетание внутреннего и внешнего взаимодействий. В этих упражнениях синяя команда обороняется, а красная – атакует.

---

1 *Betsy Beyer, Chris Jones, Jennifer Petoff and Niall Richard Murphy, eds.* Site Reliability Engineering. Sebastopol: O'Reilly, 2016.

2 *Margaret Rouse.* What Is Red Teaming // WhatIs.com, July 2017, https://oreil.ly/Lmx4M.

3 Red Team // Wikipedia, https://oreil.ly/YcTHc.

Упражнения Purple Team[1] были задуманы как развитие упражнений Red Team путем слияния опыта наступательной и оборонительной команд. Фиолетовый (purple) цвет в Purple Teaming отражает смешение или слияние красных и синих команд.

Целью этих учений является объединение наступательной и оборонительной тактик для повышения эффективности обеих групп в случае попытки компромисса. Идея заключается в том, чтобы повысить прозрачность работы механизма безопасности и узнать, насколько эффективна его подготовка, когда он оказывается на линии огня.

20.2.1. Проблемы с Red Teaming

Проблемы с Red Teaming выглядят следующим образом:

- результаты обычно состоят из отчетов. Эти отчеты, если они вообще публикуются, редко предлагают практические действия. Они также не обеспечивают согласованность работы и не побуждают команды разработчиков менять свои приоритеты;
- в первую очередь методика ориентирована на злоумышленников и эксплойты, а не на более распространенные системные уязвимости;
- команды мотивированы перехитрить противостоящую синюю команду, а не участвовать в общем понимании работы системы:
  - успех красной команды часто выглядит как большой страшный отчет, указывающий на обширную уязвимость;
  - успех синей команды часто выглядит как правильные оповещения, указывающие на то, что все профилактические меры сработали;
- для синих команд многие предупреждения могут быть неправильно истолкованы как эффективная работа средств обнаружения, хотя на самом деле ситуация может быть гораздо сложнее.

20.2.2. Проблемы с Purple Teaming

Проблемы с Purple Teaming выглядят следующим образом:

- выполнение упражнений Purple Team очень ресурсоемко, что означает:
  - в упражнении участвует только небольшой процент приложений из бизнес-портфеля;
  - упражнения выполняются нечасто, обычно ежегодно или ежемесячно;
- у полученных артефактов отсутствует механизм повторного применения прошлых результатов с целью регрессионного анализа.

---

[1] *Robert Wood and William Bengtson*. The Rise of the Purple Team. RSA Conference (2016), https://oreil.ly/VyVO0.

20.2.3. Преимущества хаос-инжиниринга в кибербезопасности

SCE устраняет перечисленные проблемы и предлагает ряд преимуществ, включая следующие:

- SCE имеет более целостный охват системы. Основная цель состоит не в том, чтобы обмануть другого человека или проверить предупреждения; скорее, это проактивное выявление сбоев безопасности системы, вызванных природой сложных адаптивных систем, и формирование уверенности в операционной безопасности;
- SCE использует простые изолированные и контролируемые эксперименты вместо сложных цепочек атак, включающих сотни или даже тысячи изменений. Трудно контролировать радиус поражения и отделить сигнал от шума, когда вы делаете большое количество одновременных изменений. SCE значительно снижает шум;
- SCE дает совместный опыт обучения, который сфокусирован на создании более устойчивых систем, а не на реакции на инциденты. В хаос-инжиниринге принято не проводить эксперименты во время активных текущих инцидентов или простоев. Нетрудно понять, что это может помешать усилиям группы реагирования, но, кроме того, важно понимать, что люди работают по-разному во время активных инцидентов из-за нехватки времени, когнитивной нагрузки, стресса и других факторов. Ситуация реального инцидента не является идеальной учебной средой, поскольку основное внимание, как правило, уделяется восстановлению работоспособности бизнеса, а не изучению того, что послужило причиной неблагоприятного события. Мы проводим эксперименты SCE при отсутствии неблагоприятных событий и отключений, когда считается, что система работает оптимально. Поэтому мы получаем лучшую среду совместного обучения, в которой команды сосредоточены на создании более устойчивых систем, а не на борьбе с инцидентом.

SCE не обязательно конкурирует с выводами или намерениями Red/Purple Teaming, однако добавляет эффективность, прозрачность и воспроизводимость, которые могут значительно повысить ценность этих методов. Упражнения Red and Purple Team просто не в состоянии идти в ногу с CI/CD и сложными распределенными вычислительными средами. Команды разработчиков программного обеспечения теперь поставляют несколько обновлений продукта в течение 24 часов. Результаты упражнений Red/Purple Teaming быстро теряют актуальность, потому что за это время система может существенно измениться. Благодаря SCE информация о состоянии безопасности следует по пятам за изменениями, которые разработчики программного обеспечения постоянно вносят в основную систему.

20.3. Игровые дни в кибербезопасности

Резервное копирование почти всегда работает; вам нужно беспокоиться о восстановлении. Аварийное восстановление и тестирование резервного копирования/восстановления предоставляют классические примеры разрушительного потенциала процесса, который не сработал. То же самое относится и к остальной части нашего контроля безопасности. Вместо того чтобы дожидаться, пока не всплывет какая-нибудь неполадка, активно вводите в систему факторы неисправности, чтобы убедиться, что наша безопасность настолько эффективна, насколько мы думаем.

Распространенный способ начать работу – использовать упражнение «игровой день» для планирования, создания и проведения экспериментов. Упражнения игрового дня, как правило, занимают от двух до четырех часов. В них принимает участие многофункциональная команда, которая разрабатывает, использует, отслеживает и/или защищает приложение. В идеале она включает работающих совместно специалистов из разных областей.

Цель упражнения игрового дня – ввести отказ в контролируемом эксперименте по безопасности, чтобы определить:

- насколько эффективно ваши инструменты, методы и процессы обнаружили сбой;
- какие инструменты и данные привели к обнаружению сбоя;
- насколько полезными были данные для выявления проблемы;
- работает ли система, как задумано.

Вы не можете предсказать, в какой форме проявят себя будущие события, но вы можете развить свою способность понимать, насколько хорошо вы реагируете на ситуации, которые вы не можете предсказать. SCE предоставляет инструментарий и процесс для отработки реагирования на инциденты.

20.4. Пример инструмента безопасности: ChaoSlingr

Сообщество профессионалов в области кибербезопасности, которые одновременно выступают за SCE и проводят эксперименты с помощью открытого исходного кода и других инициатив сообщества, становится шире день ото дня. По мере развития универсальных инструментов хаос-инжиниринга эти библиотеки будут включать все больше экспериментов, связанных с безопасностью. Однако сегодня специалисты по безопасности должны быть готовы разрабатывать и создавать свои собственные эксперименты с помощью сценариев или брать за основу существующие наборы инструментов с открытым исходным кодом, такие как ChaoSlingr.

20.4.1. История ChaoSlingr

ChaoSlingr, как показано на рис. 20.1, представляет собой эксперимент в области безопасности и систему генерации отчетности, созданные командой UnitedHealth Group во главе с Аароном Райнхартом (автором этой главы). Это был первый программный инструмент с открытым исходным кодом, продемонстрировавший практическую ценность хаос-инжиниринга для кибербезопасности. Он разработан, представлен и выпущен в виде открытого исходного кода.

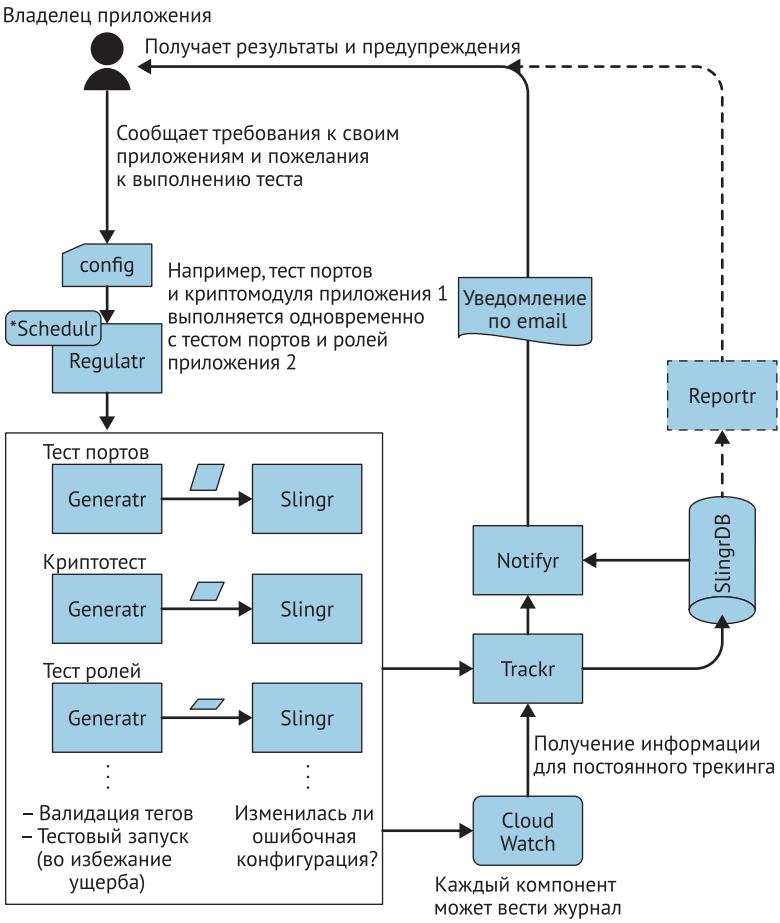

Рис. 20.1 ❖ Обобщенное представление устройства инструмента ChaoSlingr

Один из экспериментов в UnitedHealth Group включал неправильную настройку порта. Гипотеза для этого эксперимента состояла в том, что неправильно настроенный порт должен быть обнаружен и заблокирован брандмауэром, а инцидент должен быть надлежащим образом зарегистрирован

и доведен до группы безопасности. В половине случаев именно так и происходило. В остальное время брандмауэр не мог обнаружить и заблокировать дефектный порт. Но инструмент настройки облака всегда его обнаруживал и блокировал. К сожалению, этот инструмент не фиксировал нужную информацию, чтобы группа безопасности могла легко определить, где произошел инцидент.

Представьте, что вы оказались в этой команде. Ваша вера в безопасность собственной системы будет изрядно подорвана этим открытием. Сила ChaoSlingr в том, что эксперименты доказывают, верны ваши предположения или нет. У вас не остается места для догадок или предположений о своих инструментах безопасности.

Фреймворк состоит из четырех основных функций:

Generatr
: Идентифицирует объект, в который нужно внедрить отказ, и вызывает Slingr.

Slingr
: Вносит сбой.

Trackr
: Записывает подробную информацию о ходе эксперимента.

Описание эксперимента
: Предоставляет документацию по эксперименту вместе с применимыми параметрами ввода и вывода для лямбда-функций.

Первоначально ChaoSlingr был разработан для использования в Amazon Web Services (AWS). Он проактивно вводит известные условия отказа системы безопасности с помощью серии экспериментов, чтобы определить, насколько эффективно реализована защита. Основным бизнес-мотиватором для разработки инструмента послужило повышение способности компании быстро поставлять высококачественные продукты и услуги, поддерживая при этом максимально возможный уровень безопасности и защищенности.

Критически важные для безопасности системы, которые создаются сегодня, становятся настолько сложными и распределенными по своей природе, что никто не может в одиночку понять и объяснить их реальное функционирование. Даже когда все отдельные сервисы в распределенной системе функционируют должным образом, взаимодействие между этими сервисами может привести к непредсказуемым результатам. Эти непредсказуемые результаты, усугубляемые редкими, но разрушительными событиями реального мира[1], которые воздействуют на производственную среду, сделали эти распределенные системы по своей природе хаотичными. ChaoSlingr был разработан для активного выявления, обсуждения и устранения существенных недостатков, прежде чем они успеют повлиять на клиентов.

---

[1] *David Woods, Emily S. Patterson.* How Unexpected Events Produce an Escalation of Cognitive and Coordinative Demands // Stress Workload and Fatigue, Hancock and Desmond, eds. Hillsdale, NJ: Lawrence Erlbaum, 2000.

ChaoSlingr обладает следующими особенностями:

- открытый исходный код;
- большая красная кнопка: автоматически отключает ChaoSlingr, если что-то пошло не так или во время реального инцидента;
- настраиваемые сроки и частота запуска экспериментов;
- написан на Python;
- работает как лямбда-функции;
- автоконфигурация для настройки, написанная в форме скрипта Terraform.

ChaoSlingr наглядно демонстрирует, как хаос-эксперименты повышают безопасность в распределенных системах. Большинство организаций, использующих ChaoSlingr, с тех пор развили проект и создали свои собственные серии экспериментов по хаос-безопасности, исходя из базовой структуры проекта.

20.5. Вывод

Поскольку предприятия используют собственные облачные стеки и модель DevOps, их программы безопасности должны непрерывно развиваться, чтобы поспевать за нарастающей частотой изменений системы вследствие непрерывного развертывания. Для решения этих новых задач уже недостаточно традиционного тестирования безопасности.

Необходимо переосмыслить и сам подход к безопасности. «Системные ошибки» – это нормальное состояние работающих сложных систем. Сосредоточив внимание на «человеческой ошибке», «первопричине» или искушенных злоумышленниках, вы не достигнете того уровня безопасности, который дает глубокое понимание фундаментального состояния безопасности, достигнутое с помощью инструментальных петель обратной связи. SCE создает эти петли обратной связи и может открывать неизвестные ранее факторы, ограничивая область деятельности для злоумышленников.

Инструмент ChaoSlingr наглядно доказывает, что хаос-инжиниринг может быть применен и к кибербезопасности. Опыт использования ChaoSlingr в UnitedHealth Group доказывает, что этот подход имеет бизнес-ценность. При применении в области безопасности хаос-инжиниринг может раскрыть ценную и объективную информацию о том, как работают средства управления безопасностью, что позволяет организациям более эффективно расходовать средства, выделенные на безопасность. Учитывая это преимущество, все организации должны задуматься о том, когда и как применять эту дисциплину; особенно это касается тех, кто работает с масштабными сложными системами.

Соавторы и рецензенты

- Чарльз Нвату, Netflix
- Прима Вирани, Pinterest
- Джеймс Уикетт, Verica

- Майкл Чжоу, Verica
- Грейсон Брюэр
- Чэньси Ван, доктор философии, Rain Capital
- Джейми Льюис, Rain Capital
- Дэниел Уолш, Rally Health
- Роб Фрай, вице-президент по технологиям JASK (бывший руководитель по безопасности Netflix)
- Герхард Эшельбек, CISO, Google (в отставке)
- Тим Прендергаст, директор по облачным технологиям Palo Alto (основатель Evident.io)
- Ленни Малый, вице-президент по безопасности, TransUnion
- Д. Дж. Шлин, Aetna
- Энрике Салем, Bain Capital Ventures
- Роб Дюар, Cardinal Health
- Майк Фрост, HERE Технологии

Об авторе

Аарон Райнхарт – технический директор и соучредитель @Verica.io. Он работает над расширением применения хаос-инжиниринга в других критических с точки зрения безопасности частях ИТ-сферы, особенно в области кибербезопасности. Он стал пионером в применении хаос-инжиниринга в кибербезопасности во время своего пребывания на посту главного архитектора системы безопасности в крупнейшей частной медицинской компании в мире, UnitedHealth Group (UHG). В UHG Аарон выпустил ChaoSlingr, один из первых релизов программного обеспечения с открытым исходным кодом, сфокусированный на использовании хаос-инжиниринга в кибербезопасности для создания более отказоустойчивых систем. Аарон живет в Вашингтоне, округ Колумбия, и является частым автором, консультантом и лектором в этой отрасли.

Глава 21

Заключение

Устойчивость создается людьми. Разработчики, которые программируют функциональные возможности, те, кто управляет и обслуживает систему, и даже руководство, которое выделяет ресурсы для нее, являются частью сложной системы. Каждый из нас играет определенную роль в создании устойчивости, привносит свой опыт и уделяет внимание этому свойству системы.

Для работы нужны инструменты. Хаос-инжиниринг – это инструмент, который мы можем использовать для повышения устойчивости сложных систем. Наш успех как специалистов в этой отрасли зависит не от устранения сложности, а от того, как мы научимся жить с ней, ориентироваться в ней и оптимизировать ее для других критически важных для бизнеса свойств, несмотря на изначальную сложность.

Обычно, иллюстрируя различие между программным обеспечением и людьми, которые его создают и используют, проводят горизонтальную линию. Программы и инструменты помещают под линией. Люди и организация, которые пользуются инструментами, располагаются над линией, как надстройка. Как профессионалы в области программного обеспечения мы слишком часто фокусируемся на том, что происходит ниже линии. Там легче увидеть проблемы и указать на них. Возможность свести инцидент к одной строке кода, а затем просто исправить эту строку приносит психологическое удовлетворение. Есть искушение на этом и остановиться, но мы должны противостоять данному искушению.

На протяжении всей этой книги мы углублялись в детали работы как выше, так и ниже линии. Мы показали, как эта работа способствует созданию более устойчивых систем. Мы говорили о людях и технологиях, объединившихся в «социотехническую» систему, которую невозможно полностью понять, не исследуя обе стороны медали и не понимая, как они взаимодействуют.

Отсюда вытекают неожиданные побочные эффекты. Например, не всегда возможно сделать наши системы более надежными, написав больше кода. Часто лучшая стратегия повышения надежности системы заключается в повышении согласованности подходов к реагированию на опасности. Эту согласованность невозможно спроектировать заранее, или, по крайней мере, не получается спроектировать так же, как программное обеспечение.

После десятилетий исследований, охватывающих широкий спектр социологии, теории принятия решений, организационной социологии и психологии, человеческих факторов и инженерии, Йенс Расмуссен написал:

> Наиболее многообещающим общим подходом к усовершенствованному управлению рисками, по-видимому, является четкое определение границ безопасной эксплуатации вместе с работой, направленной на то, чтобы сделать эти границы видимыми для участников и научить их справляться с этими границами. В дополнение к повышению безопасности визуализация границ помогает повысить устойчивость системы, поскольку работа внутри известных границ надежнее, чем необходимость ориентироваться в неясной области, которая может непредсказуемо ухудшиться под влиянием внешних факторов[1].

Следствием из этих слов является то, что понимание контекста инцидентов и степени сопротивляемости системы более прагматично и действенно, чем поиск «первопричины» или сочинение надуманных правил.

Более того, применение стандартных правил для повышения надежности может сбить вас с пути, например:

- интуитивно понятно, что добавление избыточности в систему делает ее более безопасной. К сожалению, опыт показывает, что эта интуиция неверна. Одно лишь резервирование не делает систему более безопасной, а во многих случаях повышает вероятность сбоя системы. Вспомните про избыточные уплотнительные кольца на твердотопливном ракетном ускорителе космического челнока. Из-за наличия дублирующего уплотнительного кольца инженеры, работающие над твердотопливным ракетным ускорителем, со временем стали считать нормой выход из строя основного уплотнительного кольца, что позволило «Челленджеру» выйти за пределы допустимого риска и в конечном итоге привело к катастрофе в 1986 году[2];
- интуитивно понятно, что удаление сложности из системы делает ее более безопасной. К сожалению, опыт показывает, что эта интуиция неверна. Создавая систему, мы можем оптимизировать ее по разным критериям. Одним из свойств, которое мы стараемся улучшить, является безопасность. Чтобы добиться повышения безопасности, приходится добавлять в систему определенные компоненты. Если вы возьметесь удалять сложность из стабильной системы, то рискуете удалить функциональность, которая делает систему безопасной;
- интуитивно понятно, что эффективная эксплуатация системы делает ее более безопасной. К сожалению, опыт показывает, что эта интуиция неверна. Идеально эффективные системы хрупкие и неустойчивые. Более правильно будет учитывать и понимать неэффективность своей системы. Неэффективность позволяет системе поглощать потрясения и оставляет людям пространство для решений, направленных на исправление сбоев, которых никто не мог ожидать.

Список интуитивных, но ошибочных правил построения безопасных систем можно продолжить. Это длинный список. К сожалению, существу-

---

[1] *Jens Rasmussen*. Risk Management in a Dynamic Society: A Modelling Problem // Safety Science, Vol. 27, № 2/3 (1997), https://lewebpedagogique.com/audevillemain/files/2014/12/maint-Rasmus-1997.pdf.

[2] *Diane Vaughan*. The Challenger Launch Decision. Chicago: University of Chicago Press, 1997.

ет совсем немного универсальных правил, применимых в разных обстоятельствах.

Вот почему в рецепте безопасной системы так важны люди. Только люди могут охватить контекст каждой ситуации. Только люди могут импровизировать во время инцидента и находить творческие решения в непредсказуемых обстоятельствах. Это человеческие факторы, которые мы должны учитывать в наших сложных системах.

Нам, программистам, нравится думать, что мы ведем себя логично. Но люди не логичны. В лучшем случае мы рациональны. Но большую часть времени мы просто действуем по привычке, повторяя модели, которые работали для нас в прошлом. В этой книге исследованы и проиллюстрированы разнообразные привычки, шаблоны, взаимодействия и внутренняя работа социотехнической системы, на которых основан хаос-инжиниринг.

Мы надеемся, что убедили вас, что для повышения устойчивости вашей системы вам необходимо понять взаимодействие между людьми, которые принимают решения, финансируют, наблюдают, создают, эксплуатируют, поддерживают и предъявляют требования к системе и техническим компонентам «ниже линии», составляющим систему. С хаос-инжинирингом вы лучше поймете социально-техническую границу между людьми и машинами, оцените запас прочности, оставшийся между вашим текущим положением и катастрофическим отказом, а также улучшите обратимость вашей архитектуры. С хаос-инжинирингом вы можете существенно повысить показатели вашей социотехнической системы, которая формирует ценность вашей организации или бизнеса.

Инструменты не создают отказоустойчивость. Это делают люди, для которых и предназначены инструменты. Хаос-инжиниринг является важным инструментом для повышения отказоустойчивости систем.

Предметный указатель

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38;
тел.: **(499) 782-38-89**, электронная почта: **books@alians-kniga.ru**.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.

Кейси Розенталь, Нора Джонс

Хаос-инжиниринг

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Яценков В. С.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70 × 100 1/16.
Гарнитура PT Serif. Печать офсетная.
Усл. печ. л. 23,08. Тираж 200 экз.

Отпечатано в ООО «Принт-М»
142300, Московская обл., Чехов, ул. Полиграфистов, 1

Веб-сайт издательства: **www.dmkpress.com**