4 Антипаттерны стабильности

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

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

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

Дональд Норман описывает разрыв между ментальной моделью пользователей и моделью реализации, возникающий, когда принцип реализации непонятен, а внешний вид неочевиден. Норман рассказал про регуляторы в холодильнике, которые, как ему показалось, давали возможность напрямую задавать температуру в холодильной и морозильной камерах. Руководствуясь возникшей у него ментальной моделью, он получил замороженное молоко и растаявшее мясо, так как на самом деле механизм регулировал долю охлажденного воздуха, посылаемую в каждую из камер. В проявляющих чрезвычайно сложные взаимодействия системах интуитивные действия оператора в лучшем случае будут неэффективными, а в худшем окажут пагубное влияние. Из самых благих побуждений оператор может предпринять основанные на его представлениях о функционировании системы действия, которые запустят неожиданную цепочку связанных друг с другом событий. Подобные цепочки вносят вклад в *нарастание проблемы* (problem inflation), превращая небольшую неисправность в серьезный инцидент. Именно такая цепочка в системах охлаждения, наблюдения и управления частично стала причиной повреждения активной зоны реактора на атомной станции Три-Майл-Айленд[1]. Зачастую эти скрытые взаимосвязи выглядят совершенно очевидными при анализе последствий аварии, но на самом деле предусмотреть их практически невозможно.

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

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

Антипаттерны создают, увеличивают и множат трещины в системе.

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

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

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

4.1. Точки интеграции

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

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

ЧИСЛО ВЕБ-КАНАЛОВ

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

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

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

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

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

Сокет-ориентированные протоколы

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

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

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

Любая архитектурная схема рисуется при помощи прямоугольников и стрелок, как показано на рис. 3. Подобно множеству других вещей, с которыми нам приходится иметь дело, каждая такая стрелка является абстракцией для сетевого соединения. При этом, по сути, это абстракция абстракции. Сетевое «соединение» само по себе является логической структурой — абстракцией. Все, что мы можем увидеть в сети, — это пакеты[1]. Это работа маршрутизируемого протокола сетевого уровня (IP), входящего в стек TCP/IP. Протокол управления передачей (Transmission Control

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

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

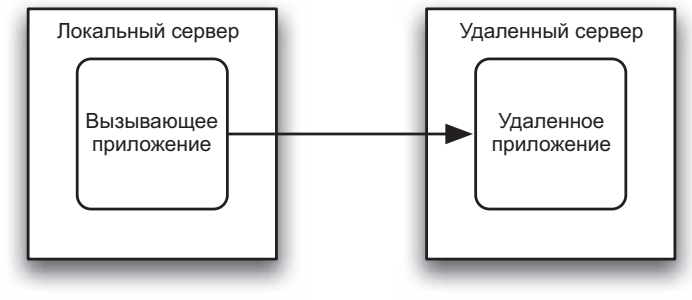

Рис. 3. Простейшая топология: прямое соединение

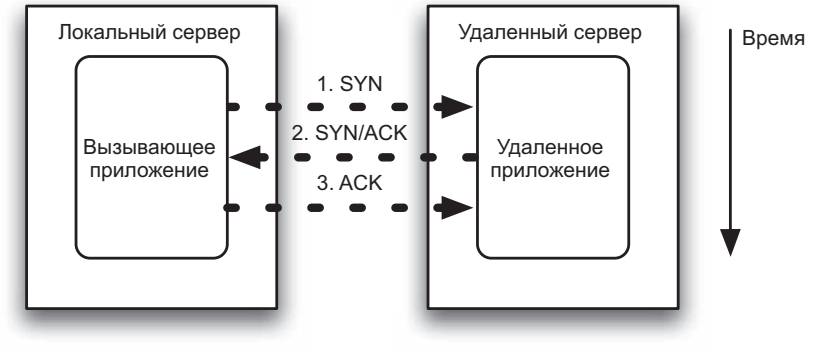

Рис. 4. Трехстороннее квитирование

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

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

отправляет собственный пакет с флагом ACK. Эти три пакета устанавливают «соединение», что дает приложениям возможность пересылать данные в обоих направлениях[1].

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

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

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

Проблема пяти часов утра

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

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