O'REILLY®

Все получилось – вы успешно перевели приложение на микросервисную архитектуру. Но теперь, когда ваши сервисы выполняются в разных средах – общедоступных, частных, в виртуальных машинах и контейнерах – ваше облачное решение начинает сталкиваться с проблемами надежности.

Как справиться с усложняющейся ситуацией? Сервисная сетка Istio поможет вам организовать управление трафиком, контроль доступа, мониторинг, отчетность, передачу телеметрии, квотирование, трассировку и др., и одновременно обеспечить надежную работу микросервисов.

В этой книге авторы объясняют, для чего сервисам нужна сервисная сетка, и показывают шаг за шагом, как Istio вписывается в жизненный цикл распределенного приложения. Здесь вы познакомитесь с инструментами и API поддержки многих функций Istio и управления ими.

«Эта книга будет иметь огромную ценность для сообщества».
Майлз Штайнхаузер, старший разработчик, специалист по облачным сервисам

«По мере выхода сервисных сеток на облачный уровень эта выдающаяся книга многим поможет в изучении современных технологий управления приложениями».
Гириш Ранганатан, специалист по поддержке Layer5, сообщество пользователей сервисных сеток

Также в этой книге вы:

- исследуете проблемы наблюдаемости, которые решает Istio;
- познакомитесь с приемами маршрутизации запросов, переключения трафика, имитации ошибок и другими возможностями, обеспечивающими стабильную работу сервисной сетки;
- научитесь генерировать и собирать телеметрию;
- попробуете использовать различные шаблоны развертывания, включая A/B, сине-зеленое;
- увидите примеры разработки и развертывания реальных приложений при поддержке Istio.

Ли Калькот – лидер в области инновационных продуктов и технологий, стремящийся обеспечить инженеров эффективными и действенными решениями. Будучи основателем Layer5, Ли находится в авангарде движения за развитие облачных технологий.

Зак Бутчер – основатель компании Tetrate и основной разработчик проекта Istio. В Tetrate Зак выполняет широкий круг обязанностей, в том числе системного архитектора, коммерсанта, автора и оратора.

ISBN 978-5-97060-863-0

Интернет-магазин: www.dmkpress.com

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

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

Istio: приступаем к работе

O'REILLY®

O'REILLY®

Istio: приступаем к работе

Использование сервисной сетки для связи, безопасности, управления и наблюдения

Ли Калькот
Зак Бутчер

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Ли Калькот и Зак Бутчер

Istio:
приступаем к работе

Istio: Up and Running

Using a Service Mesh to Connect, Secure, Control, and Observe

Lee Calcote
Zack Butcher

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Istio: приступаем к работе

Использование сервисной сетки для связи, безопасности, управления и наблюдения

Ли Калькот
Зак Бутчер

ДМК Пресс
издательство
Москва, 2020

УДК 004.451Istio
ББК 32.972.1
К17

Калькот Л., Бутчер З.
К17 Istio: приступаем к работе / пер. с англ. А. Л. Бриня. – М.: ДМК Пресс, 2020. – 236 с.: ил.

ISBN 978-5-97060-863-0

Вне зависимости от того, управляете ли вы микросервисами или модернизируете существующие неконтейнерные сервисы, рано или поздно вы все равно окажетесь перед необходимостью организации сервисной сетки. Этот момент наступит тем быстрее, чем больше будет развернуто микросервисов.

В предлагаемой вашему вниманию книге Ли Калькот и Зак Бутчер показывают, как сервисная сетка Istio вписывается в жизненный цикл распределенного приложения. Вы изучите ее архитектуру, узнаете об инструментах и API для управления многими функциями Istio, рассмотрите вопросы безопасности и управления трафиком. Особое внимание уделяется устранению неисправностей и отладке.

Книга предназначена для IT-специалистов, в задачу которых входит обеспечение безопасной, быстрой и надежной связи между сервисами.

УДК 004.451Istio
ББК 32.972.1

ISBN 978-1-492-04378-2 (англ.)

ISBN 978-5-97060-863-0 (рус.)

Содержание

От издательства

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.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com на странице с описанием соответствующей книги.

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 – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

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.

Пожалуйста, свяжитесь с нами по адресу dmkpress@gmail.com со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Об авторах

Ли Калькот – лидер в области инновационных продуктов и технологий, стремящийся обеспечить инженеров эффективными и действенными решениями. Являясь основателем Layer5, Ли находится в авангарде движения за развитие облачных технологий. Работая в компаниях SolarWinds, Seagate, Cisco и Schneider Electric, он постоянно уделяет внимание открытому исходному коду, передовым и новейшим технологиям. Советник, автор и оратор, Ли активно работает в сообществе как Docker Captain, Cloud Native Ambassador и Google Summer of Code Mentor.

Зак Бутчер – инженер-основатель компании Tetrate и основной разработчик проекта Istio. Его всегда привлекали трудные задачи, начиная с разработки веб-приложений для IE6 и заканчивая управлением сервисами, контролем доступа и иерархической организацией ресурсов Google Cloud Platform. Tetrate – небольшая компания, и Зак выполняет в ней широкий круг обязанностей, в том числе как системного архитектора, коммерсанта, автора и оратора.

Колофон

На обложке книги «*Istio: запуск и работа*» изображена вилохвостая чайка (*Xema sabini*). После завершения сезона размножения в арктической тундре на севере Аляски, в Нунавате и Гренландии популяции этой маленькой чайки рассеиваются по прибрежным районам на севере Южной Америки и Юго-Западной Африки, богатым кормом.

У взрослых особей темно-серая голова с черным кольцом в основании шеи; удлиненные крылья сверху бледно-серые с черным передним краем и треугольником белого цвета в центре. У взрослых особей красные глаза, темно-серые лапки, а также черный клюв с желтым кончиком. Молодые птицы первого года жизни имеют более тусклое оперение с коричневым окрасом. Длина птиц в среднем чуть больше 30 см (маленькая по сравнению с более привычной серебристой чайкой со средней длиной полметра).

В период гнездования эта маленькая чайка питается насекомыми, но в остальное время года, проводимое в прибрежных водах, ее рацион состоит в основном из мелких рыб, ракообразных и планктона. Эти птицы тянутся к местам, где восходящие холодные течения, такие как Перуанское течение у северо-западного побережья Перу и Бенгела, поднимающиеся к юго-западу от Африки, выносят на поверхность глубоководные питательные вещества, создавая питательный рай для многих видов морской флоры и фауны.

Название этой птице дал английский ученый Уильям Элфорд Лич в честь Эдварда Сабина, впервые описавшего ее. Лич посчитал эту птицу достаточно уникальной, чтобы отнести ее к отдельному роду (она остается единственным членом рода *Xema*), однако некоторые до сих пор оспаривают это мнение. Будучи важной фигурой в британской зоологии и таксономии начала XIX в., Лич был известен экстравагантным и нестандартным подходом к выбору названий, иногда применяя анаграммы из имен коллег и знакомых. Хотя он также использовал имена из классической мифологии, что в бóльшей степени соответствует культурным и научным традициям.

Многие животные, изображенные на обложках O’Reilly, находятся под угрозой полного исчезновения; все они важны для мира.

Иллюстрацию для обложки нарисовала Карен Монтгомери, основываясь на черно-белых гравюрах из каталога *British Birds*.

Введение

Кому стоить прочесть эту книгу?

Сервисная сетка (service mesh) является важным инструментом организации любой облачной инфраструктуры. Эта книга предназначена для тех, кто хочет начать работать с Istio. Хорошо, если читатель уже знаком с Docker и Kubernetes, но для изучения Istio по этой книге вполне достаточно базовых знаний о работе сети и Linux. Знание языка программирования Go или другого не требуется и не ожидается.

Здесь описаны многие облачные инструменты итехнологии, такие как Prometheus, Jaeger, Grafana, Meshery, Envoy и OpenTracing. Знакомство читателя с ними было бы идеальным, но для усвоения содержимого этой книги достаточно упомянутых выше базовых знаний.

Почему мы написали эту книгу?

Эпоха сервисных сеток выводит на новый уровень интеллектуальные сетевые сервисы, меняющие архитектуру современных приложений и увеличивающие надежность обслуживания. Istio – лишь одна из многих сервисных сеток, но, обладая огромным набором функций и возможностей, нуждается в исчерпывающем руководстве.

Цель этой книги – рассказать шаг за шагом, как начать работать с Istio. Она проведет вас за собой, крепко держа за руку. Все понятия описываются в четком логическом порядке, когда объяснение каждого следующего понятия основывается на предыдущих. Учитывая сложность обсуждаемой темы и активность сообщества, книга просто не в состоянии охватить все возможные варианты использования, поэтому акцент сделан на основных структурных элементах и неустаревающих аспектах проекта. По мере необходимости указываются дополнительные ресурсы.

Прочитав «*Istio: запуск и работа*», вы познакомитесь с основными возможностями Istio и сможете уверенно развертывать ее в своих облачных окружениях.

Соглашения по оформлению

В этой книге используются следующие соглашения по оформлению:

Курсив
: Используется для обозначения новых терминов, адресов URL и электронной почты, имен файлов и расширений имен файлов.

`Моноширинный шрифт`

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.

`Моноширинный полужирный`

Обозначает команды или другой текст, который должен вводиться пользователем.

`Моноширинный курсив`

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.

Так выделяются советы и предложения.

Так обозначаются советы, предложения и примечания общего характера.

Так обозначаются предупреждения и предостережения.

Благодарности

Спасибо Никки Макдональд (Nikki McDonald), Джону Дэвинсу (John Devins), Вирджинии Уилсон (Virginia Wilson), Корбину Коллинзу (Corbin Collins), Деборе Бейкер (Deborah Baker) и остальной команде O'Reilly.

И отдельное спасибо всем, кто рецензировал нашу рукопись по мере сборки этой книги, особенно нашим техническим рецензентам Майлсу Штайнхаузеру (Myles Steinhauser), Гириш Ранганатан (Girish Ranganathan) и Джесс Мэйлс (Jess Males).

Ли хотел бы лично сказать:

«Джилл, твои стойкость и любовь – моя опора в жизни. Ты подарила мне величайшую ценность – наших двойняшек.

Доктор Джи, мой друг, путешествие только началось. Спасибо, что сопровождали меня.

Кит, я жажду встречи с тобой, чтобы вновь испытать радость настоящей мужской дружбы».

Глава 1

Введение в сервисные сетки

Что такое сервисная сетка?

Сервисные сетки обеспечивают обслуживание сетевых рабочих нагрузок на основе политик, гарантируя требуемое поведение сети при постоянном изменении условий и топологии. Изменяются нагрузки, конфигурации, ресурсы (включая те, которые влияют на инфраструктуру и топологию приложений, в том числе внутри- и межкластерные ресурсы, то появляющиеся, то исчезающие), и развертываются новые рабочие нагрузки.

Основы

Сервисные сетки – это адресуемый инфраструктурный уровень, позволяющий управлять как модернизацией существующих монолитных (или иных) рабочих нагрузок, так и разрастанием микросервисов. Сервисные сетки – это задействованный в полную силу адресуемый инфраструктурный уровень. Они выгодны в монолитных средах, но главной причиной их появления является быстрое развитие микросервисов и контейнеров – истинно облачного подхода к организации масштабируемых и независимых сервисов. Микросервисы превратили внутренние коммуникации приложений в сетку, сплетенную из вызовов удаленных процедур (RPC) между сервисами, передаваемых по сетям. Среди преимуществ микросервисов – демократизация выбора языка и технологий в независимых сервисных группах – командах, быстро создающих новые функции, итеративно и непрерывно поставляющих программное обеспечение (как правило, в виде сервисов).

Область сетевых взаимодействий очень обширна. Неудивительно, что существует много тонких, почти незаметных различий между похожими понятиями. По своей сути сервисная сетка – это сеть, конструируемая разработчиками исключительно для взаимодействий между сервисами: она избавляет разработчиков от необходимости заниматься сетевыми проблемами (например, обеспечением надежности) и дает администраторам возможность декларативно определять поведение сети, идентификацию узлов и политики управления трафиком.

Это может выглядеть как реинкарнация программно определяемой сети (SDN), но сервисные сетки отличаются в первую очередь ориентацией на разработчика, а не на администратора сети. По бóльшей части сегодняшние сер-

висные сетки полностью основаны на программном обеспечении (хотя аппаратные реализации могут появиться в будущем). Термин «*целевой нетворкинг*» (intent-based networking) используется в основном в физических сетях, но, учитывая декларативный контроль на основе политик, предоставляемый сервисными сетками, справедливо сравнить их с истинно облачными SDN. На рис. 1.1 показана обобщенная архитектура сервисной сетки (в главе 2 мы описываем, что значит быть истинно облачным).

Рис. 1.1 ❖ Если нет уровня управления, это не сервисная сетка

Сервисные сетки строятся с использованием прокси сервисов. Прокси сервисов находятся на уровне данных и передают трафик. Трафик прозрачно перехватывается с помощью правил iptables в пространстве имен подов (pod – группа контейнеров).

Такой унифицированный слой инфраструктуры в сочетании с развернутыми сервисами обычно называют *сервисной сеткой* (*service mesh*). Istio превращает разрозненные микросервисы в интегрированную сервисную сетку, внедряя прокси сервисов во все сетевые пути, устанавливая соединения между прокси и ставя их под централизованный контроль, таким образом формируя сервисную сетку.

Путешествие в сервисную сетку

Не важно, чем вы занимаетесь: управляете ли флотилией микросервисов или модернизируете существующие неконтейнерные сервисы, рано или поздно вы все равно окажетесь перед необходимостью организации сервисной сетки. Чем больше будет развернуто микросервисов, тем быстрее вы окажетесь в этой ситуации.

Клиентские библиотеки: первые сервисные сетки?

Чтобы справиться со сложной задачей управления микросервисами, некоторые компании в качестве основы для стандартизации разработки начали использовать *клиентские библиотеки*. Некоторые считают эти библиотеки первыми сервисными сетками. Использование библиотеки требует, чтобы в архитектуре был прикладной код, расширяющий или использующий примитивы выбранных библиотек, как показано на рис. 1.2. Кроме того, архитектура должна учитывать потенциальное использование фреймворков и/или серверов приложений для разных языков программирования.

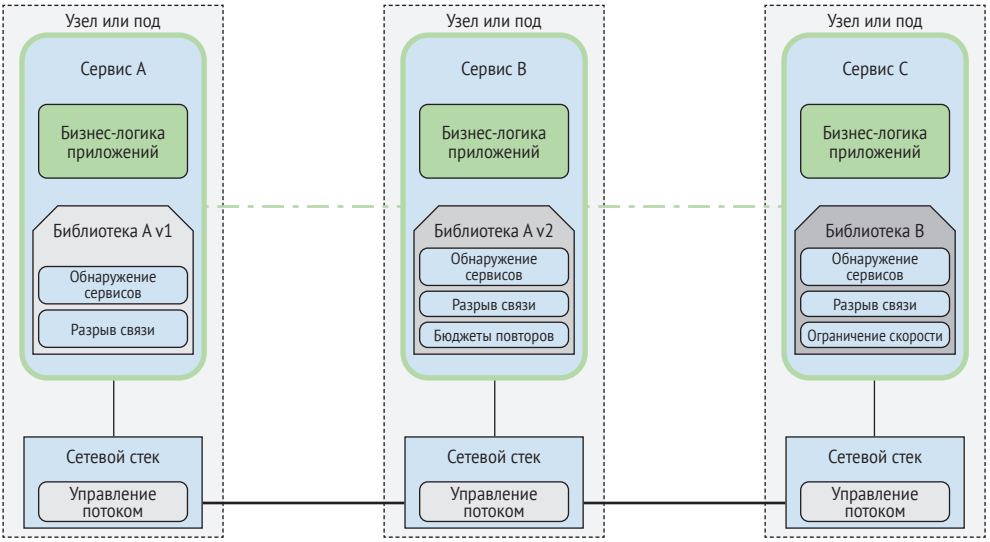

Рис. 1.2 ❖ Сервисная архитектура использует клиентские библиотеки, связанные с логикой приложений

Выгода создания клиентских библиотек, во-первых, состоит в том, что потребляемые ресурсы учитываются локально по каждому конкретному сервису, и, во-вторых, разработчики могут самостоятельно выбрать существующую или создать новую библиотеку для определенного языка. Однако недостатки использования клиентских библиотек со временем привели к появлению сервисных сеток. Наиболее значительным недостатком библиотек является тесная связь инфраструктуры с прикладным кодом. Неоднородный дизайн клиентских библиотек, часто зависящий от языка, делает их функциональность и поведение непоследовательными, что сужает возможности мониторинга, требует применения специфических методов для расширения сервисов, часто сильно зависящих друг от друга, и влечет потенциальные риски безопасности. Такие специфические *библиотеки отказоустойчивости* (*resilience libraries*) могут оказаться слишком дорогостоящими для широкого использования в организациях из-за сложности внедрения в действующие приложения или полной нецелесообразности интеграции в существующие архитектуры.

Сетевые взаимодействия – это сложно. Создание клиентской библиотеки, устраняющей конфликты между клиентами путем добавления случайных задержек и использования экспоненциального алгоритма расчета времени следующей повторной попытки – сложная задача, и не всегда удается обеспечить одинаковое поведение различных клиентских библиотек (на разных языках и в разных версиях этих библиотек). Координация обновления клиентских библиотек затруднена в больших окружениях, поскольку обновления требуют внесения изменений в код, установки новой версии приложения и, возможно, простоя приложения.

Рисунок 1.3 показывает, как размещение прокси возле каждого экземпляра приложения избавляет от необходимости иметь специфические библиотеки отказоустойчивости для разрыва цепи, тайм-аутов, повторных попыток, обнаружения сервисов, балансировки нагрузки и т. д. Сервисные сетки дают организациям, внедряющим микросервисы, возможность использовать лучшие фреймворки и языки и избавляют от хлопот с выбором библиотек и шаблонов проектирования для каждой конкретной платформы.

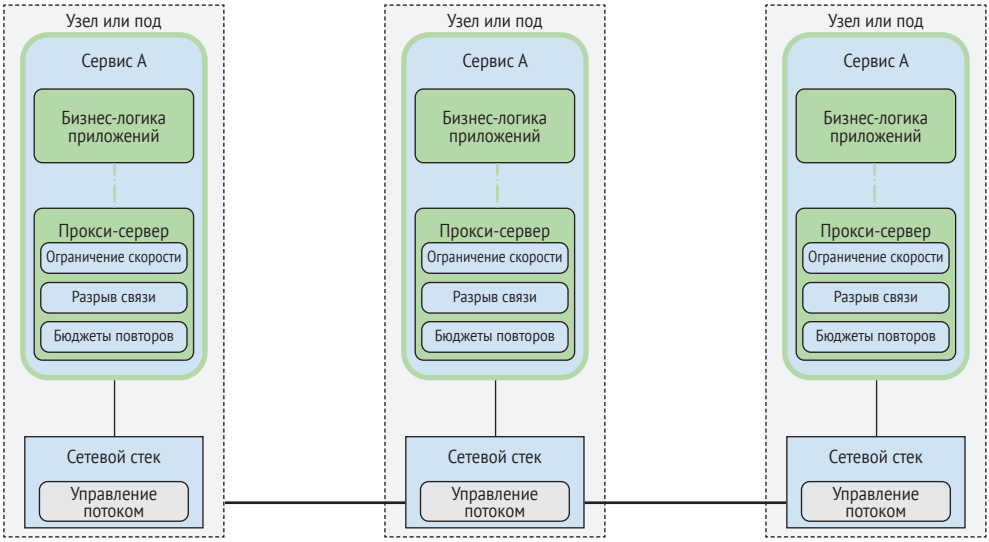

Рис. 1.3 ❖ Архитектура сервисов, использующих прокси, отделенные от логики приложений

Зачем они нужны?

Здесь можно задаться вопросом: «Есть же оркестратор контейнеров. Зачем городить еще один инфраструктурный уровень?» Сегодня, при массовом использовании микросервисов и контейнеров, оркестраторы контейнеров обеспечивают бóльшую часть потребностей кластера (узлов и контейнеров). Они осуществляют функции планирования, обнаружения и поддержания работоспособности главным образом на уровне инфраструктуры (что и требуется), оставляя неудовлетворенными потребности на уровне сервисов. Сервисная

сетка – это выделенный слой инфраструктуры для обеспечения безопасной, быстрой и надежной связи между сервисами, иногда зависящий от оркестратора контейнера или интеграции с другой системой обнаружения сервисов. Сервисные сетки могут развертываться отдельным уровнем поверх оркестраторов контейнеров, но не требуют их, поскольку компоненты уровней управления и данных могут быть развернуты независимо от инфраструктуры контейнеров. В главе 3 мы увидим, что агент узла (включая прокси), как компонент уровня данных, часто используется в неконтейнерных окружениях.

Сервисная сетка Istio обычно адаптируется под конкретные потребности. Сотрудники организаций, с которыми мы беседовали, внедряют сервисные сетки в первую очередь для контроля с применением средств мониторинга сетевого трафика. Многие учреждения, особенно финансовые, используют сервисные сетки прежде всего для управления шифрованием межсервисного трафика.

Что бы ни было катализатором, организации внедряют их сломя голову... Сервисные сетки полезны не только в истинно облачных окружениях, они помогают решать сложные задачи эксплуатации микросервисов. Многие организации, использующие монолитные сервисы (работающие на физических или виртуальных машинах, локально или за пределами организации), остро желают внедрения сервисных сеток, потому что это позволит ускорить модернизацию существующих архитектур.

На рис. 1.4 показаны возможности оркестраторов контейнеров (звездочками отмечены наиболее важные из них). Сервисные сетки, как правило, полагаются на нижележащие слои. В данном случае нижний уровень образуют оркестраторы контейнеров.

Разве на контейнерных платформах такого еще нет?

Контейнеры предлагают простой и универсальный механизм упаковки приложений, не зависящий от выбранного языка и обеспечивающий управление их жизненным циклом . Будучи универсальными по своей природе, *оркестраторы контейнеров* отвечают за формирование кластеров, эффективное распределение своих ресурсов и высокоуровневое управление приложениями (развертывание, обслуживание, оценка близости/удаленности, проверка исправности, масштабирование и т. д.). Как показано на рис. 1.4, оркестраторы имеют механизмы обнаружения сервисов и балансировки нагрузки со встроенными виртуальными IP-адресами. Поддерживаемые алгоритмы балансировки нагрузки, как правило, просты по своей природе (циклические, случайные) и действуют под одним виртуальным IP-адресом для взаимодействия с внутренними подами.

Kubernetes занимается регистрацией/вытеснением экземпляров в группе на основании их работоспособности и соответствия предикату группы (меткам и селекторам). Далее, сервисы могут использовать DNS для обнаружения сервисов и балансировки нагрузки вне зависимости от их реализации. Нет необходимости в специальных библиотеках, зависящих от языка, или в регистрации. Контейнерные оркестраторы позволили переместить рутинные сетевые задачи из приложений в инфраструктуру, освободив общую технологическую экосистему инфраструктуры и переместив акцент на более высокие уровни.

Рис. 1.4 ❖ Возможности и фокус оркестраторов контейнеров в сравнении с потребностями уровня сервисов

Теперь ясно, как сервисные сетки дополняют нижележащие слои. Перейдем к другим слоям.

Ландшафт и экосистема

Ландшафт сервисных сеток (https://oreil.ly/57P0j) представляет собой растущую экосистему инструментов, не относящуюся к истинно облачным приложениям; на самом деле он также обеспечивает большое преимущество для неконтейнеризированных, немикросервисных нагрузок. По мере осмысления преимуществ и роли, которую сервисная сетка играет в развертывании, можно начинать выбор сетки и ее интеграцию с имеющимися инструментами.

Ландшафт

Как выбрать сервисную сетку? Большое разнообразие доступных в настоящее время сервисных сеток не позволяет людям легко определить, что на самом

деле является сервисной сеткой, а что – нет. Со временем они обретают все больше похожих возможностей, что облегчает их описание и сравнение.

Интересно, но не удивительно, что многие сервисные сетки основаны на одних и тех же прокси, таких как Envoy и NGINX.

Экосистема

Как сервисная сетка соотносится с другими технологическими экосистемами, мы уже видели на примере клиентских библиотек и оркестраторов контейнеров. API-шлюзы удовлетворяют ряд схожих потребностей и обычно развертываются в оркестраторах в качестве пограничного прокси. Пограничные прокси предоставляют управление уровнями с 4 (L4) по 7 (L7) и используют оркестраторы контейнеров для обеспечения надежности, доступности и масштабируемости контейнерной инфраструктуры.

API-шлюзы взаимодействуют с сервисными сетками способом, озадачивающим многих, поскольку API-шлюзы (и прокси, на которых они основаны) варьируются от традиционных до облачных API-шлюзов и API-шлюзов микросервисов. Последние могут быть представлены коллекцией API-шлюзов с открытым исходным кодом для микросервисов, которые обертывают существующие прокси уровня L7, интегрированные с оркестраторами контейнеров и средствами самообслуживания разработчика (например, HAProxy, Traefik, NGINX или Envoy).

Главной задачей API-шлюзов в сервисных сетках является прием трафика извне и его распределение внутри. API-шлюзы образуют управляемый API для доступа к сервисам и ориентированы на передачу вертикального трафика (входящего и исходящего из сервисной сетки). Они не так хорошо подходят для управления горизонтальным трафиком (внутри сервисной сетки), так как им требуется, чтобы трафик проходил через центральный прокси, а это добавляет лишний сетевой переход. Сервисные сетки, напротив, предназначены в первую очередь для управления горизонтальным трафиком внутри сервисной сетки.

Учитывая их взаимодополняющий характер, API-шлюзы и сервисные сетки часто устанавливаются совместно. API-шлюзы используют другие функции системы управления для работы с аналитикой, бизнес-данными, вспомогательными сервисами и механизмами управления версиями. Сегодня существуют как перекрытие, так и разрыв между возможностями сервисных сеток, API-шлюзов и систем управления. Сервисные сетки по мере развития получают новые возможности, и перекрытие областей применения увеличивается.

Критическая, ненадежная сеть

Как уже отмечалось, в комплексе действующих микросервисов сеть непосредственно вовлечена в каждую транзакцию, в каждое обращение к бизнес-логике и в каждый запрос, сделанный к приложению. Надежность сети и задержки являются одними из главных проблем современных облачных приложений. Одно истинно облачное приложение может включать сотни микросервисов,

со множеством экземпляров каждого, постоянно меняющихся оркестратором контейнеров по расписанию.

Учитывая центральную роль сети, желательно, чтобы она была как можно более интеллектуальной и отказоустойчивой.

Сеть должна:

- маршрутизировать трафик в обход отказов для повышения совокупной надежности кластера;
- избегать нежелательных издержек, возникающих, например, при выборе маршрутов с высокой задержкой или серверов с холодным кешем;
- обеспечить защиту межсервисного трафика от тривиальной атаки;
- помогать в выявлении проблем, выделяя неожиданные зависимости и первопричины сбоев в коммуникациях;
- разрешать определять политики не только на уровне соединений, но и на уровне поведения сервисов.

Однако вряд ли разработчик будет гореть желанием вносить всю эту логику в свое приложение.

Необходимо управление на уровне L5, то есть сеть, ориентированная на сервисы; проще говоря – сервисная сетка.

Преимущества сервисной сетки

Сервисные сетки предлагают единообразие способов подключения, защиты, управления и мониторинга микросервисов.

Наблюдаемость

Сервисные сетки обеспечивают видимость, отказоустойчивость и контроль трафика, а также контроль безопасности распределенных сервисов приложений. Это весомые преимущества. Сервисные сетки разворачиваются прозрачно и обеспечивают видимость и контроль трафика без необходимости внесения каких-либо изменений в код приложения (более подробно см. главу 2).

В текущем, первом поколении сервисные сетки имеют большой потенциал, в том числе и Istio. Остается подождать и посмотреть, какие появятся возможности у второго поколения, когда сервисные сетки будут использоваться так же широко, как контейнеры и их оркестраторы.

Управление трафиком

Сервисные сетки обеспечивают детальный, декларативный контроль над сетевым трафиком, например позволяя определить направление запроса на выполнение канареечного развертывания. В число функций поддержки надежности обычно входят: разрыв цепи, балансировка нагрузки с учетом задержек, согласованное обнаружение сервисов, повторы, тайм-ауты и критические сроки (более подробно см. главу 8).

Безопасность

В лице сервисных сеток организации получают мощный инструмент управления безопасностью, политиками и требованиями. Большинство сервисных

сеток предоставляют центр сертификации (CA, Certificate Authority) для управления ключами и сертификатами в обеспечение связи между сервисами. Присвоение каждому сервису в сетке проверяемой идентичности является ключом к определению клиентов, имеющих право выполнять запросы к различным сервисам, а также для шифрования трафика, порождаемого этими запросами. Сертификаты генерируются для каждого сервиса и представляют его уникальную идентичность. Обычно для идентификации сервисов и управления жизненным циклом сертификатов (генерация, распределение, обновление и отзыв) от их имени используются соответствующие прокси (подробнее об этом см. главу 6).

Модернизация существующей инфраструктуры (реновация развертывания)

Многие считают, что при небольшом количестве сервисов нет смысла добавлять сервисную сетку в свою архитектуру. Это ошибочное суждение. Сервисные сетки остаются ценным инструментом независимо от числа сервисов. Просто их ценность увеличивается с ростом числа запущенных сервисов и мест их развертывания.

Некоторые новые проекты имеют роскошную возможность внедрить сервисную сетку с самого начала, но большинству организаций придется перестраивать существующие сервисы (монолитные или другие) для этого. Имеющиеся сервисы могут работать не в контейнерах, а на виртуальных или физических машинах. Сервисные сетки помогают в модернизации, позволяя организациям обойтись без переписывания приложений, внедрения микросервисов и новых языков или перехода на облако.

Для разбиения монолитов можно использовать *фасадные сервисы*. Также можно использовать *шаблон Заслонка* (Strangler), чтобы окружить устаревший монолит сервисами с более дружественным API.

Внедрение сервисной сетки дает организациям поддержку наблюдаемости (например, метрики, журналы и трассировку), а также оптимизацию зависимостей и обслуживания каждого сервиса (микросервиса или обычного). Например, для трассировки в сервисы потребуется добавить только пересылку определенных HTTP-заголовков. Сервисные сетки полезны для дооснащения существующих инфраструктур единообразными и универсальными средствами мониторинга с минимальным количеством изменений кода.

Развязка на уровне 5

Важными факторами, которые необходимо учитывать при оценке ценности сервисной сетки, являются устранение зависимости между командами разработчиков и увеличение скорости доставки, как показано на рис. 1.5.

Подобно тому как микросервисы помогают разделить разработчиков на команды по функциональным направлениям, сервисные сетки помогают отделить администраторов от процессов разработки и выпуска приложений, предоставляя администраторам декларативный контроль над работой сервисов. Сервисная сетка не просто отделяет команды друг от друга – она исклю-

чает размывание ответственности между ними и обеспечивает единообразие рабочих стандартов между организациями в одной отрасли.

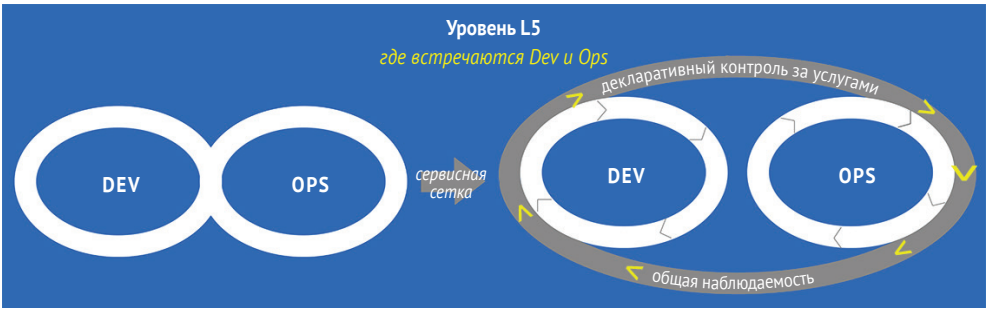

Рис. 1.5 ❖ Уровень 5 (L5), где встречаются Dev (разработчики) и Ops (операторы)

Рассмотрим следующий список заданий:

- определить, когда разрывать связь и упростить процесс;
- установить предельные сроки обслуживания;
- гарантировать генерирование распределенных трассировок и их передачу в системы мониторинга;
- запретить пользователям аккаунта «Рога и копыта» доступ к бета-версиям сервисов.

Кто несет ответственность за их реализацию – разработчик или администратор? Ответы, вероятно, будут зависеть от организации; в настоящее время нет общепринятой практики. Сервисные сетки помогают исключить распыление обязанностей или ситуации, когда одна команда обвиняет другую в недостаточной ответственности.

Сервисная сетка Istio

Начнем рассмотрение сервисной сетки Istio.

Этимология проекта

Поскольку фреймворк Kubernetes является значимой частью облачной экосистемы, его греческое этимологическое наследие привело к распространению проектов с названиями, начинающимися с «K», и других букв, ассоциирующихся с известными греческими совами. *Kubernetes* (греч. κυβερνήτης (https://oreil.ly/azC7B)) – греческое слово «рулевой» или «пилот», однокоренное со словами «кибернетика» и «губернаторский», имеющими важное значение в теории управления, а контроль находится в центре внимания Kubernetes.

Istio (греч. ιστίο) – это греческое слово «парус» и произносится как *ис-ти-о*. И конечно же, название интерфейса командной строки (CLI) Istio, `istioctl`, произносится как *ис-ти-о-си-ети-эль*, потому что он используется для управления Istio, а не для шуток.

Происхождение Istio

Istio – это реализация сервисной сетки с открытым исходным кодом, созданная компаниями Google, IBM и Lyft. Начавшийся как совместный проект этих компаний, он быстро обогатился вкладом многих других организаций и отдельных лиц. Istio – обширный проект; в истинно облачной экосистеме по масштабам задач он занимает второе место после Kubernetes. Он включает в себя ряд проектов, разрабатываемых энтузиастами под эгидой Cloud Native Computing Foundation (CNCF) , таких как Prometheus, OpenTelemetry, Fluentd, Ambassador, Jaeger, Kiali и многие другие.

Подобно другим сервисным сеткам, Istio помогает повысить отказоустойчивость и прозрачность архитектуры сервисов. Сервисные сетки не требуют, чтобы приложения знали о существовании сетки, и в этом отношении дизайн Istio ничем не отличается от других сервисных сеток. Работая с входящим, внутренним и исходящим трафиками, Istio прозрачно перехватывает и обрабатывает сетевой трафик от имени приложения.

Используя Envoy в качестве компонента уровня данных, Istio поможет настроить приложение так, чтобы рядом с ним разворачивался экземпляр прокси. Уровень управления Istio состоит из нескольких компонентов, управляющих конфигурацией прокси уровня данных, API для администраторов, настройками безопасности, проверкой политик и многим другим. Эти компоненты уровня управления мы рассмотрим в последующих главах данной книги.

Хотя изначально Istio создавалась для работы в Kubernetes, ее дизайн не зависит от платформы развертывания. То есть сервисную сетку на основе Istio можно развернуть и на платформах OpenShift, Mesos и Cloud Foundry, а также в таких традиционных окружениях, как виртуальные и физические машины. Взаимодействие Consul с Istio может помочь при развертывании на виртуальных и физических машинах. Istio принесет пользу в любом случае, используется ли она для поддержки монолита или группы микросервисов, и *чем больше сервисов, тем больше пользы*.

Текущее состояние Istio

Как развивающийся проект, Istio имеет разумную частоту выпуска новых версий, что является одним из показателей активности проекта с открытым исходным кодом. На рис. 1.6 представлена статистика сообщества за период с мая 2017 г., когда было публично объявлено об образовании проекта Istio, до февраля 2019 г. За этот период сделано порядка 2400 *форков* (копий проекта с целью участия в разработке проекта или для использования его кода в качестве основы для других проектов) и получено около 15 000 *звезд* (проект получает звезду, когда пользователь помещает его в свои закладки, чтобы получать информацию о его обновлениях в своих новостях).

Простое количество звезд, форков и коммитов может служить индикатором жизнеспособности проекта, отражая скорость его развития, уровень интереса и поддержки. Каждую из этих исходных метрик можно усовершенствовать. От-

ношение числа коммитов, обзоров и слияний ко времени отражает скорость развития проекта. При определении жизнеспособности проекта следует обращать внимание на то, как изменяется активность, насколько регулярно выходят новые версии и как часто появляются исправления ошибок и улучшения между выходами версий.

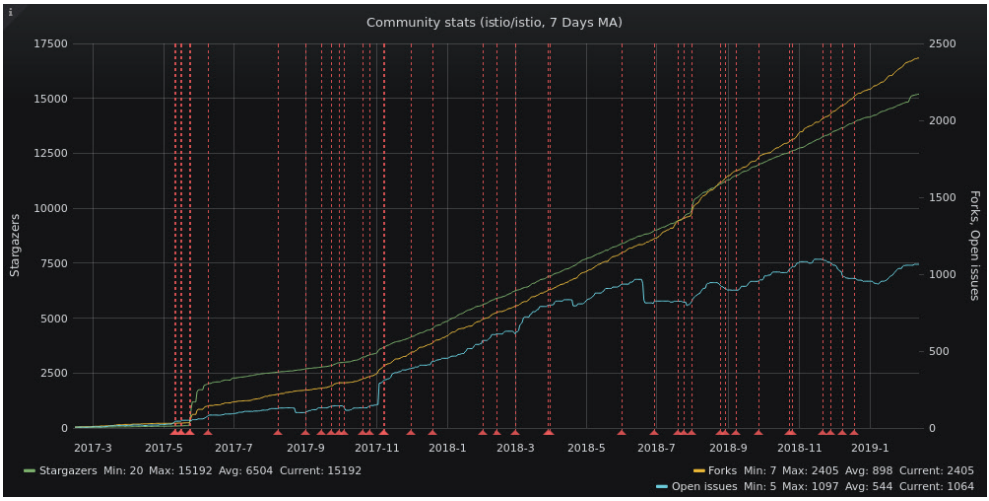

Рис. 1.6 ❖ Статистика проекта Istio

Активность развития

Нумерация версий Istio осуществляется с использованием привычной семантики (https://semver.org/lang/ru/): например, версия 1.1.1. Как и другие проекты, Istio учитывает множество нюансов, выбирая частоту выпуска новых версий и устанавливая срок поддержки (см. табл. 1.1). Несмотря на доступность ежедневных и еженедельных выпусков, они не поддерживаются и могут быть ненадежными. Как показывает табл. 1.1, ежемесячные выпуски относительно безопасны и, как правило, содержат новые функции. Если вы хотите использовать Istio в производстве, ищите выпуски, отмеченные как LTS (Long-Term Support – долгосрочная поддержка). На момент написания этой книги последней LTS-версией была версия 1.2.x.

Для справки: минорные версии Kubernetes выходят примерно каждые три месяца, поэтому каждая такая версия поддерживается в течение примерно девяти месяцев.

К примеру, разработчики дистрибутива Ubuntu во главу угла ставят стабильность, а не скорость добавления новых функций, поэтому выпускают LTS-версии раз в два года в апреле. Стоит отметить, что LTS-версии используются более интенсивно (около 95 % всех установок Ubuntu — LTS-версии).

Таблица 1.1

Тип	Уровень поддержки	Качество и рекомендованное использование
Ежедневные сборки	Не поддерживаются	
Промежуточные (snapshot) версии	Поддержка предоставляется только для последней snapshot-версии	Предположительно стабильные версии, но их использование в производстве должно быть ограничено. Обычно принимаются только начинающими или пользователями, требующими специфических возможностей
LTS-версии	Поддержка предоставляется в течение трех месяцев после выхода следующей LTS-версии	Безопасно использовать в производстве. Пользователям рекомендуется как можно скорее обновляться до этих версий
Исправления	Такой же, как LTS/Snapshot-версий	Пользователям рекомендуется применять исправления по мере их выхода для данной версии

В проекте Docker используется немного другой график выпуска версий:

- версии Docker CE Edge выходят ежемесячно;
- версии Docker CE Stable выходят ежеквартально, а исправления к ним выпускаются по мере необходимости;
- версии Docker EE выпускаются дважды в год, а исправления к ним выпускаются по мере необходимости.

Обновления и исправления выпускаются в следующем режиме:

- исправления и обновления вносятся в версию Docker EE в течение не менее одного года после выпуска;
- исправления и обновления вносятся в версию Docker CE Stable в течение одного месяца после выхода следующей версии Docker CE Stable;
- исправления и обновления вносятся в версию Docker CE Edge только до выхода следующей версии Docker CE Edge или Stable.

Выпуски

Первоначально планировалось, что ежеквартально будет выходить один точечный выпуск Istio, за которым последуют *n* выпусков с исправлениями. Snapshot-версии, качественно не уступающие точечным выпускам, предполагалось выпускать раз в месяц, но их не планировалось поддерживать, и они могли включать изменения, нарушающие обратную совместимость. История всех релизов доступна на странице истории выпусков проекта Istio в GitHub (https://oreil.ly/2fZ4x). В табл. 1.2 представлены данные о выходе новых версий Istio за 10-месячный период.

Таблица 1.2. Скорость выпуска новых версий Istio с августа 2018 г. по апрель 2019 г.

Дата выпуска	Номер выпуска	Дней с предыдущего выпуска
4/16/19	1.1.3	11
4/5/19	1.1.2	0
4/5/19	1.0.7	11
3/25/19	1.1.1	6
3/19/19	1.1.0	3
3/16/19	1.1.0-rc.6	2

***Таблица 1.2** (окончание)*

Дата выпуска	Номер выпуска	Дней с предыдущего выпуска
3/14/19	1.1.0-rc.5	2
3/12/19	1.1.0-rc.4	4
3/8/19	1.1.0-rc.3	4
3/4/19	1.1.0-rc.2	5
2/27/19	1.1.0-rc.1	6
2/21/19	1.1.0-rc.0	8
2/13/19	1.1.0-snapshot.6	0
2/13/19	1.0.6	19
1/25/19	1.1.0-snapshot.5	0
1/25/19	1.1.0-snapshot.4	48
12/8/18	1.0.5	11
11/27/18	1.1.0-snapshot.3	6
11/21/18	1.0.4	26
10/26/18	1.0.3	3
10/23/18	1.1.0-snapshot.2	26
9/27/18	1.1.0-snapshot.1	21
9/6/18	1.0.2	8
8/29/18	1.1.0-snapshot.0	5

Классификация версий

В полном соответствии с принципами гибкой разработки каждый выпуск Istio проходит свой жизненный цикл (разработка / альфа-версия / бета-версия / стабильная версия). Одни выпуски достигают стабильного состояния, а другие только появляются или совершенствуются, как показано в табл. 1.3.

Таблица 1.3. Характеристики разных версий Istio (см. https://oreil.ly/qV8b0)

	Альфа-версия	Бета-версия	Стабильная версия
Цель	Демонстрация возможностей; работает от начала до конца, но имеет ограничения	Готова к использованию в производстве, больше не игрушка	Надежная, проверенная в производстве
API	Без гарантий обратной совместимости	Поддерживается версионирование API	Надежный, пригодный к использованию в производстве. Поддерживается версионирование API с автоматическим преобразованием версий для обратной совместимости
Производительность	Не измеряется и не гарантируется	Не измеряется и не гарантируется	Производительность (задержка/масштаб) измерима, документирована, с гарантиями от регрессии
Политика устаревания	Нет	Слабая: 3 месяца	Надежная, стабильная. Уведомления рассылаются за год до изменений

Будущее

Рабочие группы разрабатывают проекты архитектуры v2 с учетом уроков, извлеченных из массового применения Istio и отзывов пользователей об удоб-

стве применения. В будущем все больше и больше людей узнают о сервисных сетках, поэтому простота внедрения будет ключевым фактором, помогающим массам успешно достичь третьей фазы их собственного облачного путешествия → контейнеры → оркестраторы → сетки.

То, чем Istio не является

Istio не учитывает специфические возможности, имеющиеся в других сервисных сетках, или предлагаемые программным обеспечением уровня управления. Это связано с изменяемостью и с использованием дополнительного стороннего программного обеспечения.

Istio лишь упрощает распределенную трассировку, но не является решением для мониторинга производительности *white box* (полностью известных) приложений (application performance monitoring, APM). Способы генерации дополнительной телеметрии, сопровождающей и анализирующей сетевой трафик и сервисные запросы, доступные в Istio, обеспечивают дополнительную видимость *black box*, объектов с неизвестными/необъявленными параметрами. Из всех метрик и журналов, доступных в Istio, эта телеметрия позволяет получить представление о потоках сетевого трафика, включая источник, приемник, задержки и ошибки; метрики сервисов высшего уровня, но нестандартные метрики приложений, которые генерируют рабочие нагрузки по отдельности, а также журналы уровня кластера остаются недоступными.

В Istio имеются плагины, интегрирующие журналы сервисов в систему мониторинга, которая используется для протоколирования на уровне кластера (например, Fluentd, Elasticsearch, Kibana). Кроме того, Istio может использовать метрики и оповещения, которые собираются уже применяемыми утилитами, такими как Prometheus.

Речь идет не только о микросервисах

Kubernetes этого всего не делает. Будет ли инфраструктура будущего полностью базироваться на использовании Kubernetes? Вряд ли. Не все приложения, особенно разработанные для работы вне контейнеров, хорошо подходят для Kubernetes (во всяком случае, пока). Хвост информационных технологий достаточно длинный – вспомним, что созданные десятилетия назад мэйнфреймы используются и по сей день.

Никакие технологии не являются панацеей. Монолиты легче понять, потому что бóльшая часть приложения находится в одном месте. Можно проследить взаимодействие различных его частей в рамках одной системы (либо более или менее ограниченного набора систем). Однако монолиты не масштабируются с точки зрения команд разработчиков и строк кода.

Нераспределенные монолиты будут существовать еще долгое время. Сервисные сетки помогают их модернизировать, позволяя создавать фасады, упрощающие эволюционное развитие архитектур. Прием развертывания перед монолитом сервисной сетки, играющей роль интеллектуального фасада, используется многими для постепенного замещения монолита путем перехвата запросов по адресам назначения (или иным способом). Этот подход позволяет постепенно перенести функции монолита в современные микросервисы,

а также может использоваться в качестве временной меры в ожидании полной реорганизации структуры облачных вычислений.

Терминология

Некоторые важные термины, связанные с Istio, следует знать и помнить:

Облако (*cloud*)
: Специализированный провайдер облачных сервисов.

Кластер (*cluster*)
: Набор узлов Kubernetes с общим API.

Хранилище конфигурации (*config store*)
: Система, хранящая конфигурацию вне уровня управления, например etcd в сервисной сетке Istio, развернутой в Kubernetes, или даже в простой файловой системе.

Управление контейнерами (*container management*)
: Программные стеки виртуализации операционных систем, такие как Kubernetes, OpenShift, Cloud Foundry, Apache Mesos и другие.

Окружение (e*nvironment*)
: Вычислительная среда от поставщиков инфраструктуры как услуги (IaaS), таких как Azure Cloud Services (https://oreil.ly/mDkY5), AWS (https://aws.amazon.com/), Google Cloud Platform (https://oreil.ly/39fXT), IBM Cloud (https://oreil.ly/N-1xd), Red Hat Cloud Computing (https://oreil.ly/ZCMj_), или группа виртуальных/физических машин, работающих в локальных/удаленных центрах обработки данных.

Сетка (*mesh*)
: Ряд рабочих нагрузок с общим административным управлением в рамках одного и того же руководящего органа (например, уровня управления).

Мультисреда (*гибрид*) (*Multienvironment, hybrid*)
: Неоднородный набор окружений, каждое из которых может отличаться от других реализацией и способом развертывания следующих инфраструктурных компонентов:

  Границы сети (*network boundaries*)
  : Пример: один компонент доступен в местной сети, а другой – в облаке.

  Системы идентификации (*Identity systems*)
  : Пример: один компонент использует LDAP, другой – учетные записи сервисов.

  Системы разрешения имен, такие как DNS (*Naming systems*)
  : Пример: локальный DNS, DNS на базе Consul.

  VM / контейнер / фреймворки управления процессами (*VM / container / process orchestration frameworks*)
  : Пример: один компонент имеет локально управляемые VM, а другой – контейнеры, управляемые Kubernetes.

Множественная аренда (*Multitenancy*)
: Логически изолированные, физически интегрированные сервисы, работающие под одним уровнем управления сервисной сетки Istio.

Сеть (*Network*)
: Набор непосредственно связанных между собой конечных точек (может включать виртуальную частную сеть [VPN]).

Безопасное разрешение имен (*Secure naming*)
: Обеспечивает сопоставление между именем сервиса и субъектами рабочих нагрузок, уполномоченными на выполнение рабочих нагрузок, реализующих сервис.

Сервис (*Service*)
: Определенная группа взаимосвязанных линий поведения в рамках сервисной сетки. Сервисы имеют имена, а политики Istio, такие как балансировка нагрузки и маршрутизация, применяются к именам сервисов. Обычно сервис имеет одну или несколько конечных точек, доступных извне.

Конечная точка сервиса (*Service endpoint*)
: Достижимое по сети представление сервиса. Конечные точки экспортируются рабочими нагрузками. Не все сервисы имеют конечные точки, доступные извне.

Сервисная сетка (*Service mesh*)
: Общий набор имен и идентификационных данных, позволяющий обеспечить применение общих политик и сбор телеметрии. Имена сервисов и субъект рабочей нагрузки уникальны в пределах одной сетки.

Имя сервиса (*Service name*)
: Уникальное имя сервиса, идентифицирующее его в пределах сети сервисов. Сервис нельзя переименовать, он сохраняет свою идентичность: каждое имя сервиса уникально. Сервис может иметь несколько версий, но имя сервиса не зависит от версии. Имена сервисов доступны в конфигурации Istio в виде атрибутов `source.service` и `destination.service`.

Прокси сервиса (*Service proxy*)
: Компонент уровня данных, управляющий трафиком от имени прикладных сервисов.

Шаблон «Прицеп» (*Sidecar*)
: Методология совместного планирования служебных и прикладных контейнеров, сгруппированных в одну логическую единицу планирования. В случае с Kubernetes – *под* (pod).

Рабочая нагрузка (*Workload*)
: Процесс / двоичный код, развернутый в Istio, обычно представленный такими объектами, как контейнеры, поды или виртуальные машины. Рабочая нагрузка может содержать ноль или более конечных точек; рабочая нагрузка может потреблять ноль или более сервисов. Каждая рабочая нагрузка имеет одно каноническое имя сервиса, связанное с ней, но может также иметь дополнительные имена сервисов.

Имя рабочей нагрузки (*Workload name*)
: Уникальное имя для рабочей нагрузки, идентифицирующее ее в пределах сервисной сетки. В отличие от имени сервиса и субъекта рабочей нагрузки, имя рабочей нагрузки не является строго контролируемым свойством и не должно использоваться в определениях списков управления доступом

(ACL). Имена рабочих нагрузок доступны в конфигурации Istio в виде атрибутов `source.name` и `destination.name`.

Субъект (*учетная запись*) *рабочей нагрузки* (*Workload principal*)

Определяет контролируемые полномочия, под которыми выполняется *рабочая нагрузка*. Для проверки субъектов рабочих нагрузок в Istio используется аутентификация сервис–сервис. По умолчанию субъекты рабочих нагрузок соответствуют формату SPIFFE ID (http://spiffe.io/). Множественные рабочие нагрузки могут совместно использовать одного и того же субъекта, но каждая рабочая нагрузка имеет один субъект. Они доступны в конфигурации Istio в виде атрибутов `source.user` и `destination.user`.

Зона, уровень управления Istio (*Zone, Istio control plane*)

В набор компонентов, необходимых для работы сервисной сетки Istio, входят Galley, Mixer, Pilot и Citadel.

- Одна зона представлена одним логическим хранилищем Galley.
- Все компоненты Mixer и Pilot, подключенные к одному хранилищу Galley, считаются частью одной и той же зоны, независимо от того, где они работают.
- Одна зона может работать независимо, даже если все другие зоны находятся вне сети или недоступны.
- Одна зона может содержать только одно *окружение*.
- Зоны не используются для идентификации *сервисов* или *рабочих нагрузок*. Каждое *имя сервиса* и *каждый субъект рабочей нагрузки* принадлежит сервисной сетке в целом, а не отдельной зоне.
- Каждая зона относится к одной сервисной сетке. Сервисная сетка охватывает одну или несколько зон.
- В отношении кластеров (например, кластеров Kubernetes) и поддержки мультисред одна зона может иметь несколько экземпляров таких кластеров. Однако пользователям Istio лучше выбрать более простые конфигурации. Запуск компонентов уровня управления в каждом кластере или окружении и ограничение конфигурации зоны единственным кластером являются относительно простой задачей.

Администраторам необходимы независимый контроль и гибкий набор инструментов для обеспечения безопасности, совместимости, доступности и отказоустойчивости микросервисов. Разработчикам требуются свобода от проблем с инфраструктурой и возможность экспериментировать с различными производственными функциями, а также выполнять канареечное развертывание новых версий без ущерба для всей системы. Istio добавляет в микросервисы поддержку управления трафиком и создает основу для важнейших функций, таких как безопасность, мониторинг, маршрутизация, управление связью и управление политиками.

Глава 2

Истинно облачный подход к равномерной наблюдаемости

В этой главе мы рассмотрим истинно облачный подход к обеспечению равномерной наблюдаемости через призму сервисных сеток и исследуем три понятия: *истинно облачный* (*cloud native*), *наблюдаемость* (*observability*) и *равномерная наблюдаемость* (*uniform observability*). В первом разделе мы разберем аморфное понятие *истинно облачный*, охарактеризовав некоторые его аспекты. Затем попробуем увидеть разницу между мониторингом и наблюдаемыми свойствами сервиса. В последнем разделе поразмышляем о возможностях автоматической телеметрии, обеспечивающей всеобъемлющее и непротиворечивое представление действующих сервисов. Поскольку сервисные сетки являются продуктом облачных вычислений, начнем с определения, что на самом деле означает *истинно облачный*.

Что значит быть *истинно облачным*?

«Истинно облачный» – это совокупное название *технологий* и *процессов*. С целью увеличения эффективности машин и людей истинно облачные технологии охватывают всю архитектуру приложений, упаковку и инфраструктуру. Истинно облачный процесс представляет собой полный жизненный цикл программного обеспечения. Зачастую, но не всегда истинно облачный процесс сокращает исторически разделенные организационные функции и этапы жизненного цикла (например, архитектура, обеспечение качества, безопасность, документация, разработка, эксплуатация, сопровождение и т. д.) до двух функций: *разработка* и *эксплуатация*. Разработка и эксплуатация являются двумя главными функциями людей, поставляющих программное обеспечение как услугу, обычно применяя практики и культуру DevOps. Истинно облачное программное обеспечение обычно, но не всегда является непрерывно предоставляемой услугой.

Чем больше будет развернуто сервисов, тем выше окупаемость инвестиций при использовании сервисной сетки. Истинно облачная архитектура позволяет запускать большое количество сервисов, поэтому очень важно понимать, что значит быть истинно облачным. Сервисные сетки дают дополнительные выгоды также неконтейнеризированным рабочим нагрузкам и монолитным сервисам. Примеры такой дополнительной выгоды приводятся в этой книге.

Истинно облачные приложения обычно работают в общедоступных или частных облаках. Как минимум они работают поверх программно адресуемой инфраструктуры. Поэтому *подъем и перенос* приложения в облако не делают его истинно облачным.

Ниже приведены характеристики истинно облачных приложений:

- работают на программно адресуемой инфраструктуре, динамичны и отделены от физических ресурсов одним или несколькими уровнями абстракций вычислительных и сетевых ресурсов, а также хранилищ;
- распределены и децентрализованы, при этом основное внимание часто уделяется поведению приложения, а не месту его работы. Учитывают события жизненного цикла программного обеспечения, что позволяет регулярно применять обновления, плавно изменяя сервисы без перебоев в обслуживании;
- устойчивы и масштабируемы, предполагают возможность запуска избыточных экземпляров, чтобы не имелось единой точки отказа и для повышения живучести;
- поддерживают возможность наблюдения с помощью собственных механизмов и/или механизмов нижележащих уровней. Учитывая их динамический характер, распределенные системы относительно сложнее в отладке, поэтому следует учитывать их наблюдаемость.

Путь к истинной облачности

Для большинства организаций путь к истинной облачности – это эволюционный акт применения принципов истинной облачности к существующим сервисам путем их модернизации или реализации заново. Другим посчастливилось начать проекты после того, как принципы и инструменты облачных вычислений стали широко доступны и известны. В любом случае – и при модернизации существующих, и при разработке новых коллекций сервисов – сервисные сетки предлагают значительный выигрыш, *увеличивающийся с ростом числа сервисов*. Сервисные сетки являются следующим логическим шагом после развертывания системы оркестровки контейнерами. На рис. 2.1 показаны различные пути к истинной облачности.

Одни сервисные сетки проще других в развертывании, другие дают бóльший выигрыш; в зависимости от устанавливаемой сетки может потребоваться определенное количество микросервисов, чтобы она стала полезной. Со временем сервисные сетки (и расширения к ним) помогут разработчикам решать общие вопросы прикладного уровня (такие как учет затрат и планирование цен), поскольку сервисные сетки просто закрывают подобные общие проблемы.

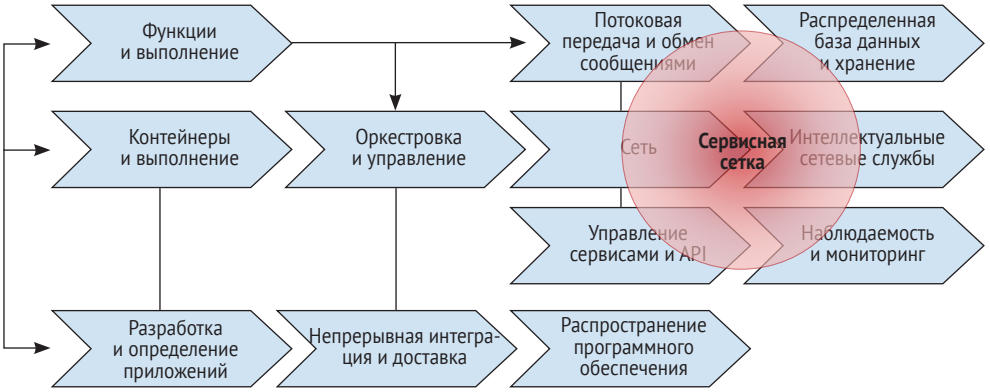

Рис. 2.1 ❖ Пути к истинной облачности разнообразны и многочисленны

В зависимости от уровня опыта команды и особенностей конкретных проектов путь к истинной облачности подразумевает использование различных комбинаций из процессов разработки программного обеспечения, методов эксплуатации, архитектур приложений, способов упаковки и развертывания, а также инфраструктур приложений. Команды, создающие истинно облачные приложения, используют один, несколько или все подходы, выделенные на рис. 2.2.

Рис. 2.2 ❖ Эволюция к истинной облачности с использованием различных процессов и технологий (архитектура, упаковка и инфраструктура)

Упаковка и развертывание

Истинно облачные технологии часто принимают форму микросервисов (через конечные точки сервисов), встроенных контейнеров (через планировщик) и бессерверных функций (через оповещения о событиях). Эволюционные изменения в практике упаковки обусловлены потребностью инженеров в эффективном использовании машин и скорости поставки. Путь к истинному облаку требует все меньших и меньших единиц внедрения, что достигается за счет высокой степени изоляции ресурсов. Изоляция с помощью виртуализации и контейнеризации обеспечивает более высокий уровень эффективности, поскольку мелкие пакеты позволяют более плотно загрузить серверы.

На каждом этапе эволюции, судя по общей статистике количества внедрений – от физических серверов до виртуальных машин, контейнеров, ядер и функций, – наблюдалась разная плотность загрузки. Некоторые способы упаковки обеспечивают лучшие гарантии переносимости, совместимости, изоляции, эффективности и т. д. Например, контейнеры обеспечивают бóльшую переносимость и совместимость, чем виртуальные машины. Функции же, напротив, несмотря на компактность, изолированность и практически бесконечную масштабируемость, проигрывают в переносимости и демонстрируют самую высокую степень обособленности среди всех способов упаковки. Независимо от выбранного способа упаковки – будь то развертывание сервисов непосредственно в операционной системе хоста, в виртуальной машине, в контейнере или в виде бессерверной функции – сервисные сетки обеспечат связь, контроль, наблюдаемость и безопасность.

Архитектура приложений

Еще более важной характеристикой истинной облачности является архитектура приложений. Центральное место в истинно облачной среде занимают такие качества, как эфемерность, активно планируемая рабочая нагрузка, слабая связанность с четко описанными зависимостями, ориентированность на события, горизонтальная масштабируемость и четкое разделение на сервисы с состоянием и без. Истинно облачные приложения часто являются примером декларативного архитектурного подхода, включающего устойчивость, доступность и наблюдаемость в первоочередные задачи проектирования.

Истинно облачные технологии позволяют организациям создавать и запускать масштабируемые приложения в динамических окружениях, таких как публичные, частные и гибридные облака, где приложения концентрируются вокруг декларативных API для взаимодействия с инфраструктурой. Такие методы позволяют создавать гибкие, управляемые, наблюдаемые и слабо связанные системы. Istio и другие сервисные сетки с открытым исходным кодом реализуют следующее поколение сетевых технологий, предназначенных для истинно облачных приложений.

Процессы разработки и эксплуатации

Опыт разработчиков и администраторов также занимает центральное место в философии истинной облачности, что способствует повторному использованию кода и компонентов, а также высокой степени автоматизации. В сочетании с подходом *инфраструктура как код* (*infrastructure as code*, IaC) администраторы активно автоматизируют методы развертывания, мониторинга и масштабирования облачных приложений и их инфраструктуры. В сочетании с надежной автоматизацией микросервисы позволяют инженерам часто и предсказуемо вносить существенные изменения с минимальными затратами труда, обычно используя несколько конвейеров *непрерывной интеграции* (*continuous integration*, CI) и *непрерывной поставки* (*continuous delivery*, CD) для создания и развертывания микросервисов.

Высокая степень гранулированной наблюдаемости является ключевым аспектом систем и сервисов, отслеживаемых и управляемых инженерами, которые обеспечивают надежную работу сайта. Istio генерирует метрики, журналы и трассировки, относящиеся к запросам, отправляемым через сетку, облегчая инструментирование сервисов, чтобы дать возможность генерировать метрики, журналы и трассировки без изменения кода (кроме распространения контекста в трассировках). Как Istio, так и сервисные сетки в целом вставляют специальный инфраструктурный уровень между Dev (разработкой) и Ops (эксплуатацией), разделяя общие проблемы связи сервисов и обеспечивая независимый контроль над сервисами. Без сервисной сетки администраторы по-прежнему оставались бы тесно связанными с разработчиками множеством вопросов, поскольку им нужны были бы новые сборки приложений для контроля сетевого трафика, формирования и влияния на управление доступом и упорядочивания взаимодействий между сервисами. Развязка Dev и Ops является ключом к обеспечению автономности и независимости итераций.

Истинно облачная инфраструктура

Публичные, гибридные и частные облака являются основой определения истинной облачности. В трех словах: облако – это программно-определяемая инфраструктура. Использование API в качестве основного интерфейса к инфраструктуре является основной концепцией облачных вычислений. Истинно облачные интегрированные рабочие нагрузки используют эти API (или их абстракции) вместо неистинных рабочих нагрузок, не знающих своей инфраструктуры. По мере распространения определения «истинно облачный» распространяются и сами облачные сервисы. В широком смысле облачные сервисы эволюционировали из IaaS в управляемые сервисы и бессерверные функции. Учитывая, что большинство систем FaaS (*functions-as-a-service*) выполняются внутри контейнеров, платформы FaaS могут работать в сервисной сетке и выиграть от *равномерной наблюдаемости*.

Истинно облачные технологии и процессы радикально повышают эффективность использования аппаратных ресурсов при одновременном снижении затрат на обслуживание и эксплуатацию, а также значительно повышают об-

щую гибкость и удобство сопровождения приложений. Хотя использование оркестратора контейнеров удовлетворяет целому ряду потребностей инфраструктуры, оно не отвечает всем требованиям, предъявляемым к приложениям и сервисам. Сервисные сетки обеспечивают необходимый инструментарий для решения задач обслуживания истинно облачных приложений.

Что такое наблюдаемость?

Правильное определение любых новых терминов важно не только для создания общей системы обозначений (и облегчения понимания), но также чтобы избежать ненужных споров. Понятие *наблюдаемости* системы (*observability*), в отличие от *мониторинга* (*monitoring*), давно и подробно обсуждается в отрасли. Для наглядности определим *мониторинг* как выполняемую функцию, деятельность, а *наблюдаемость* – как атрибут системы.

Наблюдаемость описывает, насколько хорошо и каким образом система предоставляет сигналы для мониторинга. Наблюдаемое программное обеспечение обычно оснащается инструментами для сбора и передачи информации (телеметрии/измерений), позволяющей понять особенности его работы.

Мониторинг, напротив, определяет действия, связанные с наблюдением и проверкой поведения и результатов работы системы и ее компонентов во времени и оценкой состояния системы. Чем больше наблюдаемых атрибутов (шире наблюдаемость), тем больше возможностей для мониторинга. Мониторинг помогает проверить состояние системы (например, ухудшилась эффективность работы системы или нет).

Рассматривайте возможность мониторинга как условие, необходимое для контроля состояния. Мониторинг осуществляется с целью выявления сбоев, как правило, посредством опросов наблюдаемых конечных точек. Проще говоря, системы раннего мониторинга рассматривают время бесперебойной работы (*uptime*) как один из ключевых показателей устойчивости. Современные инструменты мониторинга ориентированы на такие метрики, как задержка, ошибки (количество неудачных запросов), объем трафика (запросов в секунду для веб-сервисов или транзакций в секунду для хранилищ) и насыщенность (степень использования ресурсов). Современные системы мониторинга часто оснащены логикой для выявления аномального поведения, прогнозирования нарушений работоспособности и т. д. Сервисные сетки объединяют наблюдаемость и мониторинг, частично обеспечивая и то, и другое, генерируя, комбинируя и анализируя данные телеметрии. Различные сервисные сетки включают в себя инструменты мониторинга в виде встроенных функций или простых дополнений и расширений.

Обсуждение затихло бы, если бы использовались термины «мониторинг» и «возможность мониторинга». Эти термины можно оставить в покое, в компании синонимов «наблюдение» и «наблюдаемость». Однако продавцам нужно предложить новый термин, чтобы они могли презентовать, жонглировать, владеть и позировать. Отсутствие термина и определения для дебатов похоже на кинсеаньеру без пиньяты (совершеннолетие без подарка) – ритуал оказывается неполным.

Наблюдаемость для разработчиков и мониторинг для администраторов? Может быть, но дело не в этом. До появления сервисных сеток было не ясно, кто должен обеспечить наблюдаемость системы, а кто – осуществлять ее мониторинг. Большинство команд по-разному отвечают на вопрос, кто несет ответственность за определение и реализацию задач обслуживания данного сервиса. Такие обязанности часто распылены. Сервисная сетка разделяет команды разработки и эксплуатации, вводя уровень менеджмента – уровень 5 (L5) – между инфраструктурой нижнего уровня и сервисами приложений верхнего уровня.

Источники телеметрии

Под наблюдаемостью обычно подразумевается наличие журналов с событиями и ошибками; трассировок со связями и аннотациями; метрик в виде гистограмм, индикаторов, сводок и счетчиков, как показано на рис. 2.3.

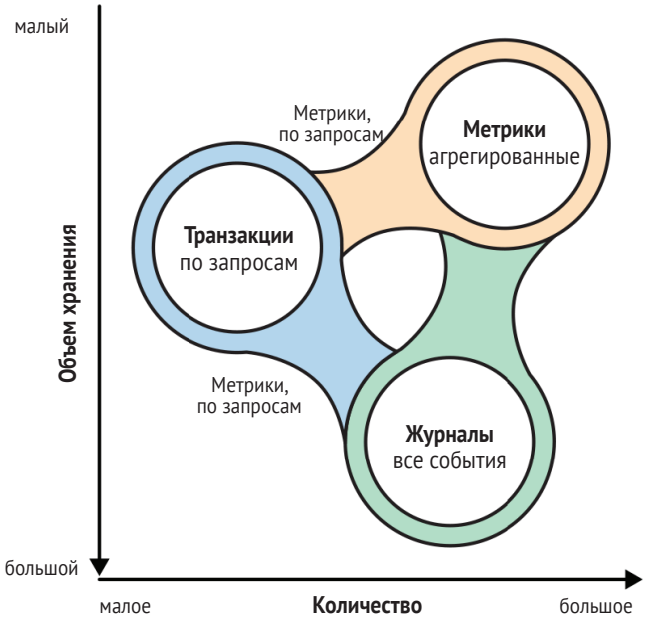

Рис. 2.3 ❖ Три источника наблюдаемости: ключевые типы телеметрии

Журналы

Журналы предоставляют дополнительный контекст для таких данных, как метрики. Они хорошо подходят для отладки, но для них трудно подобрать правильный баланс настроек – какие журналы должны храниться долго, а какие можно затирать. Также журналы дорого обходятся с точки зрения производительности, поскольку они, как правило, имеют довольно большой объем и за-

нимают больше всего места в хранилище. Структурированное журналирование не имеет недостатков, присущих чисто текстовым журналам, но получающиеся журналы занимают намного больше места и требуют гораздо больше времени для выборки и обработки (как это видно из цен на программные продукты мониторинга на основе журналов). К числу передовых методов журналирования относятся принудительное соблюдение квот и динамическая регулировка скорости генерации журналов.

Метрики

Метрики, в отличие от журналов, имеют постоянное потребление ресурсов и хороши для оповещений. Журналы и метрики вместе дают представление об отдельных системах, но затрудняют просмотр всего жизненного цикла запроса, прошедшего через несколько систем. Это довольно характерно для распределенных систем. Метрики – мощный инструмент, а их совокупности позволяют вникнуть в суть происходящего. Хорошая сжимаемость метрик снижает требования к объему хранилища (хорошая реализация Gorilla позволяет снизить выборку до 2,37 байта) и дает возможность длительного хранения исторических данных.

Трассировка

Трассировка позволяет детально отслеживать порядок обработки запроса различными сервисами. Трассировку сложно вводить задним числом, так как (помимо прочего) сторонние библиотеки, используемые приложением, также нуждаются в оснащении средствами трассировки. Распределенная трассировка может быть накладной, поэтому большинство систем трассировки для сервисных сеток используют различные формы выборки данных для извлечения лишь части наблюдаемых событий. При сборе трассировки снижаются потребление ресурсов и затраты на хранение, но также снижается и наглядность. Частота выборки уравновешивается частотой регистрации (обычно выражаемой в процентах от объема запросов к сервисам).

Алгоритмы сбора трассировки

Все алгоритмы выборки трассировки имеют свои компромиссы. Они делятся на две большие категории: *действующие с опережением* (*head-based*) и *действующие с отставанием* (*tail-based*)). Алгоритм трассировки с опережением осуществляет выборку непрерывно и последовательно с самого начала. Алгоритм с отставанием принимает решение о сохранении данных трассировки в конце выполнения запроса, когда есть возможность учесть дополнительные критерии. Вот несколько различных алгоритмов выборки:

Вероятностная выборка
: Решение основывается на бросании монеты с определенной вероятностью.

Выборка с ограничением скорости
: При принятии решения учитывается ограничение скорости, обеспечивающее выборку фиксированного количества трасс за определенный промежуток времени.

Адаптивная выборка
: В процессе выборки производится динамическая корректировка параметров для приведения фактического количества трасс в соответствие с заданной/желаемой частотой генерации.

Контекстно-зависимая выборка
: Используется для специализированной или отладочной выборки (например, с помощью специального заголовка, сигнализирующего системам отслеживания, что данный запрос должен быть отобран).

В то время как в большинстве современных систем трассировки используется выборка с опережением, некоторые новые системы применяют метод выборки с отставанием. Все эти подходы имеют свои накладные расходы и издержки, поэтому важно оценить эффективность затрат на организацию телеметрии (например, нужна ли многолетняя аналитика по этому сигналу?). Можно применять различные алгоритмы для настройки поведения выборки и для уменьшения нагрузки на серверы уровня администрирования.

Комбинирование источников телеметрии

Максимально наблюдаемая система использует каждый внутренний сигнал, включая синтетические проверки, мониторинг взаимодействий с конечным пользователем (мониторинг действий реального пользователя) и инструменты для распределенной отладки. Все еще необходимы и тестирование методом *black box*, и синтетические проверки, поскольку с их помощью осуществляется сквозная проверка всего, что могло быть не замечено.

На рис. 2.4 показаны варианты, как при сборе рабочей телеметрии достигается компромисс между стоимостью хранения, производительностью (процессор, память и задержка запросов) и ценностью информации с точки зрения ее выразительности или полезности для исправления медленных или ошибочных ответов.

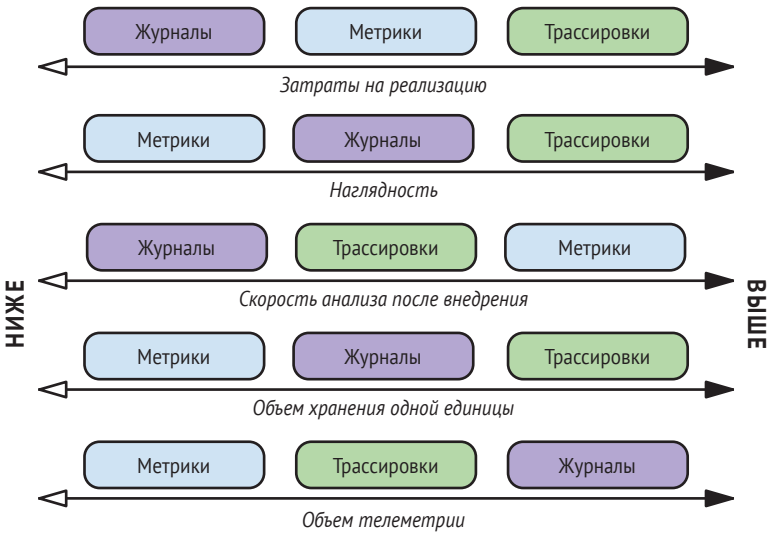

Рис. 2.4 ❖ Сравнение показывает ценность каждого источника в сравнении с затратами на его организацию

Определенно метрики имеют наибольшую эффективность. Некоторые сервисные сетки облегчают распределенную трассировку, поэтому можно утверждать, что распределенная трассировка дает наибольшую отдачу при наименьших вложениях (с точки зрения наглядности). В идеале инструментарий должен позволять регулировать уровень детализации и частоту выборки, чтобы можно было снижать накладные расходы и иметь желаемый уровень наблюдаемости.

Сегодня многие организации уже используют индивидуальные решения мониторинга для распределенной трассировки, журналирования, безопасности, контроля доступа и т. д. Сервисные сетки централизуют и помогают решать проблемы наблюдаемости. Istio генерирует и отправляет телеметрию на основе посылаемых в сетку запросов, как показано на рис. 2.5.

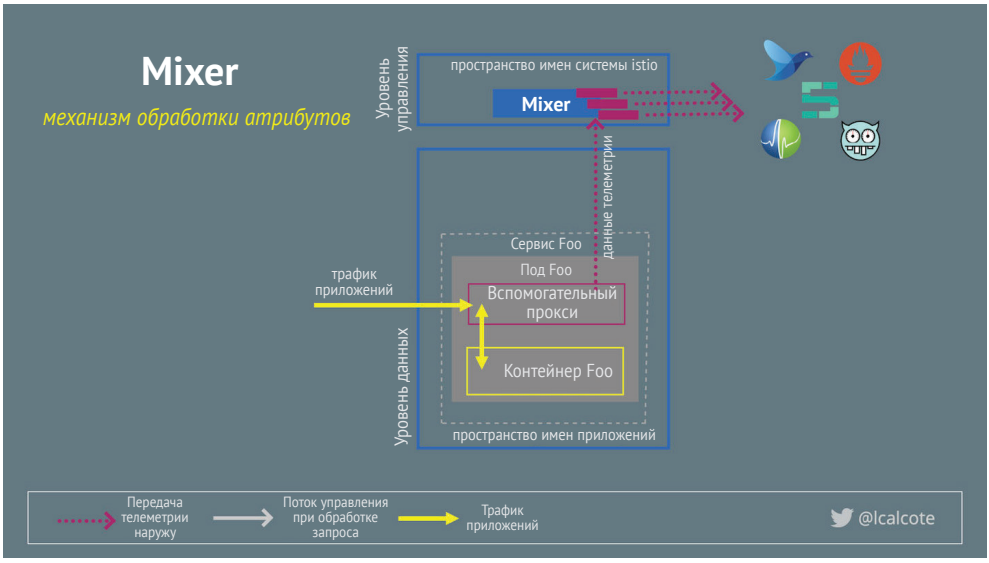

Рис. 2.5 ❖ Istio Mixer осуществляет сбор и отправку телеметрии в системы мониторинга, аутентификации и квотирования через адаптеры

Почему так важна наблюдаемость в распределенных системах?

Способность агрегировать и сопоставлять журналы, метрики и трассировки при работе распределенных систем важна для анализа происходящего внутри приложения, работающего в различных инфраструктурах. Мы знаем, что при работе распределенной системы неизбежно *будут* возникать сбои, и можем учесть какие-то из них. Однако все виды сбоев заранее не известны, поэтому система должна быть максимально наблюдаемой (и отлаживаемой), чтобы можно было ставить новые вопросы и объяснять поведение приложения (в контексте его инфраструктуры). Какие же из множества доступных сигналов являются наиболее важными для мониторинга?

Владельцам сервисов нужно изучить сложные и взаимосвязанные системы и уметь объяснять аномалии, основываясь на телеметрии, полученной с помощью инструментария. Именно благодаря комбинации внутренних наблюдений и внешнего мониторинга сервисные сетки освещают работу сервиса, где в противном случае наблюдатель оказался бы слепым.

Какие KPI являются наиболее значимыми?

Популярные методологии по-разному описывают, какие ключевые показатели эффективности (KPI) следует измерять и каким образом:

- USE означает «использование, загрузка и ошибки». Это объем ресурсов (например, процессор, память и т. д.);
- RED означает «скорость, ошибки и длительность». Это объем запросов. Длительность явно подразумевает распределение, а не усредненные значения;
- четыре золотых сигнала – это задержка, запросы, загрузка и ошибки.

Рассматривая популярные методики USE, RED и четыре золотых сигнала, можно обнаружить, что общими для них являются *запросы*, *задержки* и *ошибки*.

Запросы
Показывает, какая нагрузка приходится на вашу систему, и измеряется в запросах в секунду.

Задержка
Отражает время, необходимое для обслуживания запроса, обычно отдельно для успешных и неудачных запросов.

Ошибки
Дает скорость завершения запросов неудачей.

Мониторинг – это деятельность, выражающаяся в простом наблюдении за состоянием системы в течение определенного периода времени. Наблюдаемость (условие наблюдаемости) – это мера того, насколько точно можно определить внутреннее состояние системы, опираясь на внешние проявления ее работы, мера наблюдаемости чего-либо.

Вместо того чтобы пытаться преодолеть проблемы распределенных систем, вписывая инфраструктурную логику в код приложения, можно прибегнуть к помощи сервисной сетки. Сервисная сетка помогает обеспечить централизацию управления сервисами, избежать переоснащения их избыточными инструментами и сделать наблюдаемость сервисов исчерпывающей и равномерной.

Равномерная наблюдаемость с сервисной сеткой

Наглядность (наблюдаемость) является причиной номер один, почему люди используют сервисные сетки. Сервисные сетки обеспечивают не просто наблюдаемость, но равномерную и исчерпывающую наблюдаемость. Возможно, вы привыкли пользоваться индивидуальными решениями мониторинга для распределенной трассировки, журналирования, безопасности, контроля доступа, учета и т. д. Сервисные сетки централизуют и помогают консолидировать подобные разрозненные «экраны», генерируя метрики, журналы и трассировки

запросов, проходящих через сетку. Используя преимущества автоматически генерируемых идентификаторов связей из уровня данных, Istio предоставляет основу распределенной трассировки для визуализации зависимостей, объема запросов и частоты отказов. Стандартный шаблон атрибутов Istio (подробнее об атрибутах – в главе 9) выделяет метрики глобального объема запросов, глобального коэффициента успешности и индивидуальных ответов сервисов по версиям, источникам и времени. Когда метрики распределены в кластере повсеместно, они дают новое понимание, а также освобождают разработчиков от необходимости писать код, генерирующий эти метрики.

Важность наличия исчерпывающей и единообразной информации (особенно информации о поведении запросов) легко проиллюстрировать на примере проблем, возникающих в клиентских библиотеках.

Клиентские библиотеки

Клиентские библиотеки (иногда их называют фреймворками микросервисов) давно стали привычным инструментарием разработчиков, стремящихся повысить устойчивость своих микросервисов. Существует ряд (https://layer5.io/landscape/) популярных клиентских библиотек, предлагающих такие функции отказоустойчивости, как ограничение времени ожидания обработки запроса и выполнение повторных попыток в случае несвоевременного ответа сервиса.

Клиентские библиотеки набрали популярность по мере того, как микросервисы закрепились в истинно облачном дизайне приложений, что позволило избежать необходимости переписывать одну и ту же инфраструктуру и логику работы каждого сервиса. Одна из проблем с микросервисами заключается в том, что они объединяют ту же самую инфраструктуру и функциональные возможности с имеющимся кодом. Это приводит к дублированию кода в сервисах и несоответствию в возможностях и поведении различных библиотек. Как показано на рис. 2.6, при запуске нескольких версий одной и той же (или разных) библиотеки процесс их обновления группами обслуживания может быть затруднен. Когда задачи, свойственные распределенным системам, встроены в код сервиса, нужно искать инженеров, чтобы обновить и исправить их библиотеки (которых может быть несколько, используемых в разной степени). Развертывание свежей согласованной версии может занять некоторое время. Обеспечение слаженности является довольно сложной задачей.

Взаимодействие с системами мониторинга

С точки зрения приложения сервисные сетки в основном обеспечивают *black-box-мониторинг* (наблюдение со стороны) взаимодействий сервис–сервис, оставляя *white-box-мониторинг* (наблюдение изнутри с помощью внутренних механизмов) приложений в ведении микросервиса. Прокси, составляющие уровень данных, благодаря своему расположению прекрасно подходят для создания метрик, журналов и трассировок и способны обеспечить равномерную и исчерпывающую наблюдаемость по всей сетке в целом. Istio предоставляет адаптеры для получения и передачи этой телеметрии в систему/системы мониторинга по выбору.

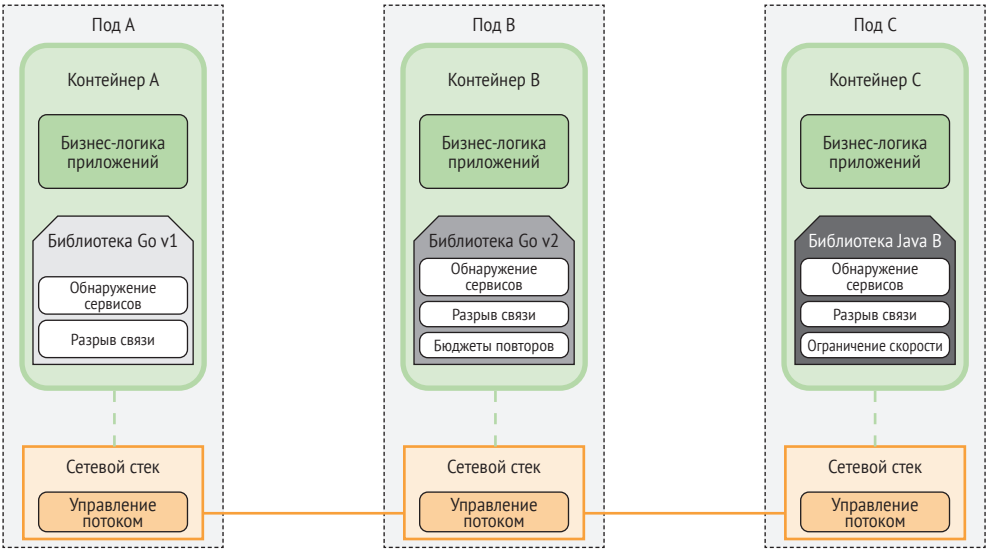

Рис. 2.6 ❖ Приложения, тесно связанные с логикой управления инфраструктурой

В связи с необходимостью ускорения поставки, потенциально в глобальном масштабе, и разумного использования ресурсов истинно облачные приложения запускаются как неизменные, изолированные, эфемерные пакеты в некоторой общей инфраструктуре.

Использование клиентских библиотек и фреймворков микросервисов порождает определенные проблемы. Сервисные сетки перемещают эти проблемы в прокси и отделяют их от кода приложения.

Легко ли контролировать приложение в работе? Многие приложения контролируются легко, но, к сожалению, при проектировании некоторых из них наблюдаемость расценивалась как второстепенная задача. В идеале следует заранее продумывать возможность наблюдения, поскольку это один из важных факторов масштабируемости приложений наряду с резервным копированием, безопасностью, возможностью проверки и т. д. В таком случае выбор можно сделать осознанно. Независимо от того, рассматривалась ли заранее возможность наблюдения в вашем окружении или нет, сервисная сетка предоставляет множество выгодных преимуществ.

Телеметрия обходится дорого. Поэтому на практике используются различные методы и алгоритмы, помогающие собирать только самые показательные сигналы.

Глава 3

Istio на первый взгляд

По мере совершенствования операций по развертыванию контейнеров многие организации придут к использованию сервисных сеток в своей среде. Эта тема занимает значительное место в истинно облачной экосистеме. В настоящее время многие администраторы, операторы, архитекторы и разработчики стремятся узнать, как, когда и зачем использовать сервисные сетки. Поэтому давайте рассмотрим Istio.

Как вы узнали из главы 2, Istio, как и другие сервисные сетки, вводит новый уровень в современную инфраструктуру, создавая потенциал для реализации надежных и масштабируемых приложений с детальным контролем. При использовании микросервисов проблемы усугубляются по мере развертывания все бóльшего их числа. Возможно, вы не пользуетесь микросервисами. Даже несмотря на то что преимущества сервисных сеток наиболее ярко проявляются при развертывании микросервисов, Istio также может принести выгоды при развертывании сервисов непосредственно в операционной системе, работающей на виртуальной машине и физическом сервере.

Архитектура сервисной сетки

В общем и целом архитектура сервисных сеток, включая Istio, делится на два уровня: уровень *управления* и уровень *данных*, а третий уровень (уровень администрирования) может располагаться в существующих/инфраструктурных системах. Архитектура Istio тоже следует этой парадигме. На рис. 3.1 представлено распределение задач по уровням.

Более подробное описание моделей развертывания сервисных сеток и подходов к эволюционным архитектурам см. в книге *The Enterprise Path to Service Mesh Architectures* (https://oreil.ly/TTO4p).

Источник: Layer5 Meshery

Рис. 3.1 ❖ Istio и другие сервисные сетки состоят из двух уровней. Третий уровень обычно используется для обеспечения дополнительной сетевой интеллектуальности и простоты управления в неоднородных окружениях

Уровни

Уровень данных Istio перехватывает все запросы и отвечает за проверку работоспособности, маршрутизацию, балансировку нагрузки, аутентификацию, авторизацию и генерацию наблюдаемых сигналов. Прокси добавляются к сервисам автоматически и действуют незаметно для них, в результате чего приложения не знают о существовании уровня данных. Уровни данных отвечают за связь внутри кластера, а также за обработку входящего и исходящего трафиков кластера. Любой трафик, входящий в сетку или исходящий из нее, если это трафик прикладного сервиса, сначала направляется для обработки в прокси. В Istio трафик прозрачно перехватывается с помощью правил iptables и перенаправляется в прокси. Уровень данных Istio анализирует каждый пакет/запрос в системе и отвечает за обнаружение сервисов, проверку работоспособности, маршрутизацию, балансировку нагрузки, аутентификацию, авторизацию и наблюдаемость.

Уровень управления Istio предоставляет единую точку администрирования прокси для эффективного управления их настройками в режиме реального времени по мере изменения расписания выполнения сервисов в вашем окружении (например, в кластере контейнеров). Уровни управления отвечают за определение политик и настройку сервисов в сетке, объединяя вместе изолированные прокси и превращая их в сервисную сетку. Уровни управления не анализируют сетевые пакеты непосредственно; они работают вне сервисной сетки. Для взаимодействия с ними обычно предоставляется графический интерфейс и интерфейс командной строки , каждый из которых обеспечивает до-

ступ к централизованному API комплексного управления поведением прокси. Используя API уровня управления, можно автоматизировать изменение его конфигурации (например, с помощью конвейера CI/CD), что особенно удобно, когда конфигурация зависит от версии и обновляется вместе с ней.

Уровень управления Istio:

- предоставляет политику и конфигурацию сервисам в сетке через API и позволяет администраторам задать требуемое поведение маршрутизации/отказоустойчивости;
- объединяет набор *изолированных прокси* в сервисную сетку и предоставляет уровню данных:
  - API для передачи локализованной конфигурации;
  - абстракцию обнаружения сервисов;
- использует API для определения политик через назначение квот и ограничений;
- обеспечивает безопасность посредством выдачи и ротации сертификатов;
- назначает идентификатор рабочей нагрузке;
- обрабатывает конфигурацию маршрутизации:
  - не анализирует никакие пакеты/запросы в системе;
  - определяет границы сетей и способы доступа к ним;
  - унифицирует сбор телеметрических данных.

Компоненты уровня управления Istio

В этом разделе мы в общих чертах рассмотрим функциональность каждого компонента уровня управления. А более подробно о поведении, конфигурации и возможности отладки каждого компонента мы поговорим в следующих главах.

Pilot

Pilot – это, так сказать, капитан корабля в сетке Istio. Он находится в постоянном контакте с базовой платформой (например, Kubernetes), наблюдает за состоянием и местоположением запущенных сервисов и предоставляет эту информацию уровню данных. Pilot взаимодействует с системой обнаружения сервисов и отвечает за настройку прокси (компонент уровня данных `istio-proxy` мы рассмотрим позже).

По мере развития Istio компонент Pilot все больше будет сосредоточен на масштабируемом обслуживании конфигураций прокси и все меньше – на взаимодействии с нижележащими платформами. Pilot обслуживает Envoy-совместимые конфигурации путем объединения информации о конфигурации и конечных точках из различных источников и преобразования ее в объекты xDS. А ответственность за взаимодействие с нижележащими платформами возьмет на себя другой компонент – Galley.

Galley

Galley является компонентом агрегации и распределения конфигурации Istio. По мере развития своей роли он будет изолировать другие компоненты Istio от

нижележащих платформ и поставляемых пользователем конфигураций, принимая и проверяя их. Galley использует *протокол настройки сетки* (*Mesh Configuration Protocol*, MCP) в качестве механизма обслуживания и распределения конфигураций.

Mixer

Mixer может работать самостоятельно и представляет собой компонент уровня управления, разработанный для абстрагирования других инфраструктурных компонентов Istio, таких как Stackdriver или New Relic. Mixer отвечает за предварительную проверку условий, управление квотами, передачу телеметрии, а также:

- обеспечивает мобильность платформы и окружения;
- обеспечивает тщательный контроль за операционной политикой и телеметрией, отвечая за оценку политики и передачу телеметрических данных;
- имеет обширные настройки;
- абстрагирует большинство инфраструктурных задач с помощью *конфигурации на основе намерений* (*intent-based configuration*).

Прокси сервисов и шлюзы обращаются к Mixer для выполнения предварительных проверок, чтобы определить, разрешено ли выполнение запроса (проверка), разрешена ли связь между вызывающим абонентом и сервисом, не превышены ли квоты, а также для передачи телеметрии после обработки запроса (отчет). Mixer взаимодействует с компонентами инфраструктуры через набор встроенных и сторонних адаптеров. Конфигурация адаптера определяет, когда и какому компоненту отправлять телеметрию. Администраторы сервисных сеток могут использовать адаптеры Mixer в качестве точки интеграции и взаимодействия со своими инфраструктурными компонентами, поскольку он работает как механизм обработки атрибутов и маршрутизации.

Проект Mixer v2, в настоящее время находящийся в стадии разработки, предлагает существенно иную архитектуру. Тем не менее его цели и задачи предполагается оставить практически такими же, как и в Mixer v1.

Citadel

Citadel позволяет Istio обеспечить надежную аутентификацию между сервисами и конечными пользователями с помощью протокола двусторонней защиты транспортного уровня (*mutual Transport Layer Security*, mTLS) со встроенной идентификацией и управлением учетными данными. Компонент центра сертификации (CA) Citadel утверждает и подписывает запросы, отправляемые агентами Citadel, и выполняет генерацию, развертывание, ротацию и аннулирование ключей и сертификатов. Citadel имеет возможность взаимодействовать с идентификационным каталогом в процессе сертификации.

Citadel имеет расширяемую архитектуру, позволяя использовать различные центры сертификации вместо самосозданных, самоподписанных ключей и сертификатов для подписи сертификатов рабочей нагрузки. Возможность подключения центра сертификации Istio позволяет и облегчает:

- интеграцию с системой открытых ключей (PKI) в вашей организации;
- защиту связи между доверенными сервисами Istio и не-Istio (используя один и тот же корень доверия (root of trust));
- защиту ключа подписи CA, храня его в хорошо защищенной среде (например, HashiCorp Vault, аппаратный модуль безопасности, или HSM).

Прокси сервисов

Прокси в сервисных сетках можно использовать для обработки входящего сетевого трафика, трафика между сервисами и исходящего трафика. Istio использует прокси для передачи трафика между сервисами и клиентами и обычно развертывает их в подах, используя шаблон «Прицеп» (Sidecar). (Примеры других моделей развертывания можно найти в книге *The Enterprise Path to Service Mesh Architecture* (https://oreil.ly/Up2H7). Сетку на самом деле образуют связи прокси–прокси. Из этого следует, что для включения приложения в сетку между приложением и сетью следует установить прокси, как показано на рис. 3.2.

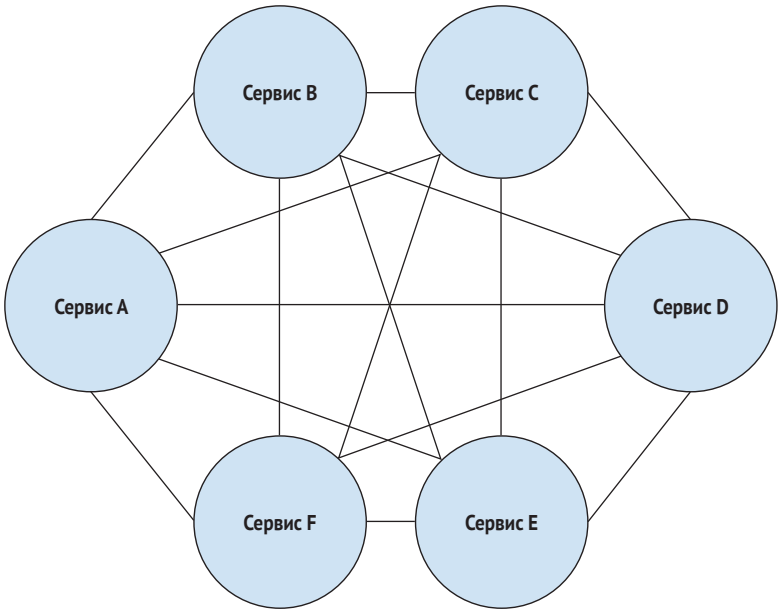

Рис. 3.2 ❖ Сетку образуют связи между прокси

Шаблон «Прицеп» (Sidecar) разработан с целью упростить добавление в контейнер новых возможностей без его изменения. В этом смысле «прицеп» и сервис действуют как модуль с расширенными возможностями. Поды вмещают «прицеп» и сервис как одно целое.

Компоненты уровня данных Istio

Istio использует расширенную версию прокси Envoy, разработанного на C++ высокопроизводительного прокси, для обработки входящего и исходящего трафиков для всех сервисов в сетке. Istio использует такие функции Envoy, как динамическое обнаружение сервисов, балансировка нагрузки, инкапсуляция прикладных протоколов в TLS, проксирование HTTP/2 и GRPC Remote Procedure Call (gRPC), прерывание цепи, проверка состояния, поэтапное развертывание с процентным разделением трафика, имитация ошибок и детальные метрики.

Envoy разворачивается как «прицеп» к соответствующему сервису в том же поде Kubernetes. Это позволяет Istio извлекать большое количество сигналов о поведении трафика в виде атрибутов, которые, в свою очередь, могут использоваться в Mixer для соблюдения политик и передачи информации о работе всей сетки в системы мониторинга.

Внедрение

Такая модель развертывания прокси также позволяет внедрить Istio в существующую систему без ее переделки. Это значительное преимущество использования Istio. Появляются возможности для немедленного просмотра метрик сервисов верхнего уровня, детального контроля трафика, автоматической аутентификации и шифрования между всеми сервисами без изменения кода приложения *или* манифестов развертывания.

Используем канонический пример Bookinfo, приложения Istio, чтобы разобраться, как прокси формируют сетку и какие функции выполняют. На рис. 3.3 показано приложение Bookinfo без прокси (мы развернем и исследуем это приложение более подробно в главе 4).

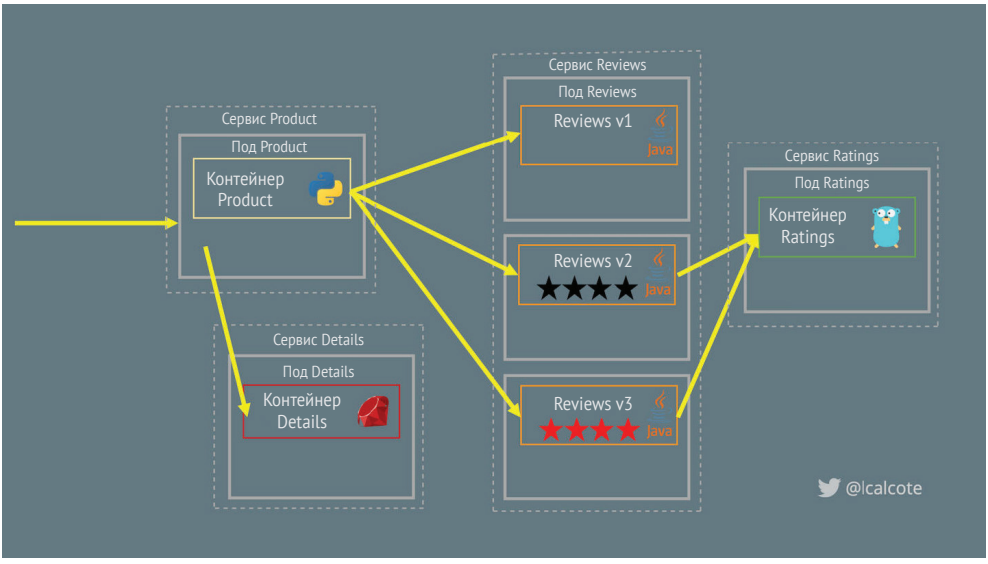

Рис. 3.3 ❖ Приложение Bookinfo без прокси

В Kubernetes автоматическое внедрение прокси реализовано с использованием Kubernetes API Server и входного контроллера изменяющих вебобработчиков. Он не имеет состояния и зависит только от шаблона внедрения и *конфигурационных карт сетки*, а также от объекта пода, в который осуществляется внедрение. Таким образом, его легко масштабировать по горизонтали вручную, с помощью объекта Deployment, или автоматически, с помощью механизма автоматического *горизонтального масштабирования подов* (*Horizontal Pod Autoscaler*, HPA).

Собственно внедрение вспомогательного прокси во вновь созданный под занимает в среднем 1,5 мкс (на микротест) – это время выполнения вебобработчика. Общее время внедрения будет чуть больше, если учитывать задержки в сети и время обработки запроса в API Server.

Развертывая легковесные прокси между прикладными контейнерами и сетью, Istio решает известную проблему распределенных систем, связанную с отсутствием однородных, надежных и неизменных сетей. Рисунок 3.4 показывает полную архитектуру Istio с уровнями управления и данных, а также со всеми внутренними компонентами. Полное развертывание сетки также включает входные и выходные сервисные шлюзы.

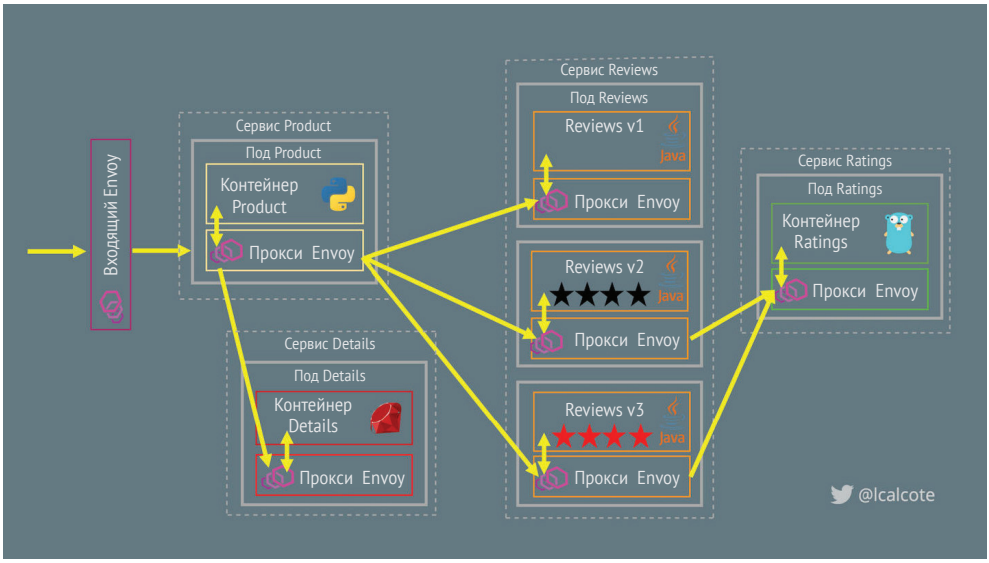

Рис. 3.4 ❖ Приложение Bookinfo с внедренными прокси

Шлюзы

В Istio v0.8 было введено понятие входных и выходных шлюзов. Симметрично похожие, входные и выходные шлюзы играют роль обратного и прямого прокси соответственно, для входящего и исходящего трафиков. Поведение Istio Gateway, как и других компонентов Istio, определяется конфигурацией, в которой можно определить, какой трафик пропускать, с какой скоростью и т. д.

Входной шлюз

Настройка входных шлюзов позволяет определить точки в сервисной сетке, через которые будет проходить входящий трафик. Обработка входящего в сетку трафика – это задача обратного проксирования, похожая на традиционную балансировку нагрузки веб-сервера. Настройка обработки исходящего из сетки трафика – это задача прямого проксирования, при решении которой определяется, какому трафику разрешено покинуть сетку и куда он должен направляться.

Например, следующая конфигурация шлюза настраивает прокси для выполнения функций балансировки нагрузки и открывает доступ к портам 80 и 9080 (HTTP), 443 (HTTPS) и 2379 (TCP). Настройки шлюза будут применены к прокси, запущенному в поде с меткой `app: my-gateway-controller`. Даже если Istio настроит прокси на прослушивание этих портов, пользователь обязан убедиться, что внешний трафик в эти порты разрешен в сетке (более подробно см. документацию по шлюзу Istio (https://oreil.ly/e5plQ)):

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: my-gateway
spec:
    selector:
      app: my-gateway-controller
    servers:
   - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
   - uk.bookinfo.com
   - eu.bookinfo.com
    tls:
      httpsRedirect: true # отправляет ответ 301 для переадресации HTTP-запросов
   - port:
      number: 443
      name: https
      protocol: HTTPS
    hosts:
   - uk.bookinfo.com
   - eu.bookinfo.com
    tls:
      mode: SIMPLE # разрешает передачу HTTPS-трафика в этот порт
      serverCertificate: /etc/certs/servercert.pem
      privateKey: /etc/certs/privatekey.pem
- port:
    number: 9080
    name: http-wildcard
    protocol: HTTP
  hosts:
  - "*"
- port:
    number: 2379 # открывает доступ к сервису через порт 2379
```

```
    name: mongo
    protocol: MONGO
  hosts:
- "*"
```

Выходной шлюз

Трафик может выходить из сервисной сетки Istio двумя путями: непосредственно из вспомогательных прокси или через выходной шлюз, в котором можно применить дополнительные политики для управления исходящим трафиком.

По умолчанию Istio-совместимые приложения не имеют доступа к URL вне кластера.

Непосредственно из прокси сервисов

Если требуется, чтобы исходящий трафик миновал выходной шлюз, это можно сделать настройкой `istio-sidecar-injector в ConfigMap`. Добавьте в конфигурацию следующий параметр, идентифицирующий локальные сети кластера и удерживающий локальный трафик в пределах сетки, пересылая трафик для всех остальных адресатов во внешнюю сеть:

```
--set global.proxy.includeIPRanges="10.0.0.1/24"
```

После применения этой настройки и обновления прокси Istio этой конфигурацией запросы вовне будут миновать выходной шлюз и отправляться адресату непосредственно. Прокси в Istio перехватывают и обрабатывают только внутренние запросы, рассылаемые внутри кластера.

Через выходной шлюз

Выходной шлюз может понадобиться для поддержки соединений, устанавливаемых из приватного пространства IP-адресов в кластере, мониторинга или взаимодействий между кластерами.

Выходной шлюз позволяет применять правила мониторинга и маршрутизации Istio к трафику, исходящему из сетки. Облегчает коммуникацию между приложениями, работающими в кластере, где узлы не имеют публичных IP-адресов, закрывая доступ в интернет для приложений в сетке. Определение выходного шлюза, передача через него всего исходящего трафика и назначение публичных IP-адресов только узлам выходного шлюза позволяют контролировать доступ к внешним сервисам из внутренних узлов сетки (и работающих на них приложений), как показано на рис. 3.5.

Почему шлюзы Istio предпочтительнее, чем ресурс Ingress в Kubernetes?

Шлюзы в Istio v1alpha3 используются для расширения функциональности, а возможностей Ingress в Kubernetes оказалось недостаточно для приложений Istio. По сравнению с Ingress в Kubernetes шлюзы Istio могут работать как чистые TCP-прокси 4-го уровня (L4) и поддерживать все протоколы, поддерживаемые Envoy.

Еще один важный аспект – разделение доверенных доменов между подразделениями в организации. Kubernetes Ingress объединяет спецификации с L4 по L7, что затрудняет различным подразделениям в организации с раздельными доверенными доменами (таким как отдел безопасности, отдел администрирования сети, отдел сопровождения кластеров и отдел разработки) управление своим входным трафиком.

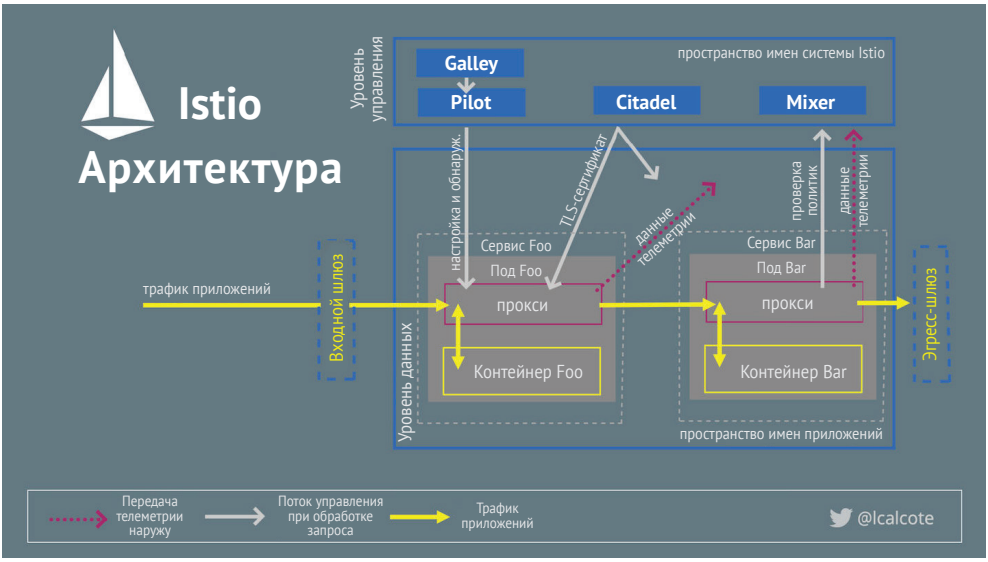

Рис. 3.5 ❖ Архитектура и компоненты Istio

Расширяемость

Хотя это не является самоцелью для некоторых сервисных сеток, Istio поддерживает возможность индивидуальной настройки. Как расширяемая платформа она позволяет настраивать два основных аспекта интеграции: использовать другие прокси и другие адаптеры телеметрии/авторизации.

Замена прокси

По умолчанию в Istio применяются прокси Envoy, но точно так же можно использовать другие прокси. В экосистеме имеется множество других прокси, кроме Envoy, но в настоящее время только два поддерживают интеграцию с Istio: Linkerd и NGINX. Linkerd2 сейчас не подходит как прокси общего назначения; скорее, он ориентирован на легкость. Расширяемость в нем реализуется через плагин gRPC.

Возможно, вы выберете другую сервисную сетку, но в случае с Istio лучше использовать один из следующих прокси:

Linkerd
: Можно использовать, если вы уже применяете Linkerd и хотите использовать Istio API, например `CheckRequest`, для получения одобрения/неодобрения перед выполнением действия.

NGINX
: Исходя из практического опыта и необходимости использования проверенных временем прокси, можно выбрать NGINX. Возможно, вам пригодятся

кеширование, файервол веб-приложений или другие функции, доступные в NGINX Plus.

Consul Connect

Легковесный прокси Consul Connect не вызывает сложностей при развертывании и прекрасно подходит для простых задач.

Появление нескольких прокси для Istio вызвало большой ажиотаж. Интеграция с Linkerd была реализована еще в версии Istio v0.1.6. Аналогично на раннем этапе развития Istio появилась возможность использования NGINX в качестве прокси в рамках проекта nginMesh (https://oreil.ly/axVOR).

Хотя nginMesh больше не развивается, статья «How to Customize an Istio Service Mesh» (https://oreil.ly/p72P8) и лекция (https://oreil.ly/KHPOG) помогут лучше понять проблемы расширяемости Istio относительно прокси.

Прокси не могут выполнять свои задачи без настройки. Pilot является, так сказать, капитаном корабля в сетке Istio, поддерживая синхронизацию с базовой платформой путем отслеживания и предоставления своих сервисов в `istio-proxy`. Как прокси по умолчанию, `istio-proxy` включает расширенную версию Envoy. Обычно в роли вспомогательных прокси, входного и выходного шлюзов используется один и тот же Docker-образ `istio-proxy`. `istio-proxy` содержит не только прокси, но и агента Istio Pilot, регулярно извлекающего настройки из Pilot в прокси через короткие интервалы времени, чтобы каждый прокси знал, куда направлять трафик. В этом случае агент nginMesh выполняет задачу по настройке NGINX в качестве `istio-proxy`. Pilot отвечает за жизненный цикл `istio-proxy`.

Замена адаптеров

Компонент Mixer уровня управления Istio отвечает за контроль доступа и применение политик ко всей сервисной сетке и за сбор телеметрических данных из прокси. Как основная точка расширяемости Istio, Mixer классифицирует адаптеры в соответствии с типом потребляемых ими данных, как показано на рис. 3.6.

Модель разработки *внутрипроцессных* адаптеров Mixer в Istio теперь считается устаревшей. Как и предыдущие проекты с открытым исходным кодом, разработчики Istio ориентировались на удобство включения адаптеров в дерево. По мере развития и становления Istio от этой модели перешли на другую, предусматривающую отделение адаптеров от основного проекта, чтобы снять нагрузку с разработчиков основного проекта и стимулировать ответственность разработчиков, занимающихся интеграцией с другими системами.

В будущем возможности расширения могут быть распространены на поддержку надежных хранилищ ключей для HSM и более эффективных распределенных инфраструктур трассировки. Кроме того, ожидается, что уровни администрирования будут играть более заметную роль по мере распространения Istio и других сервисных сеток.

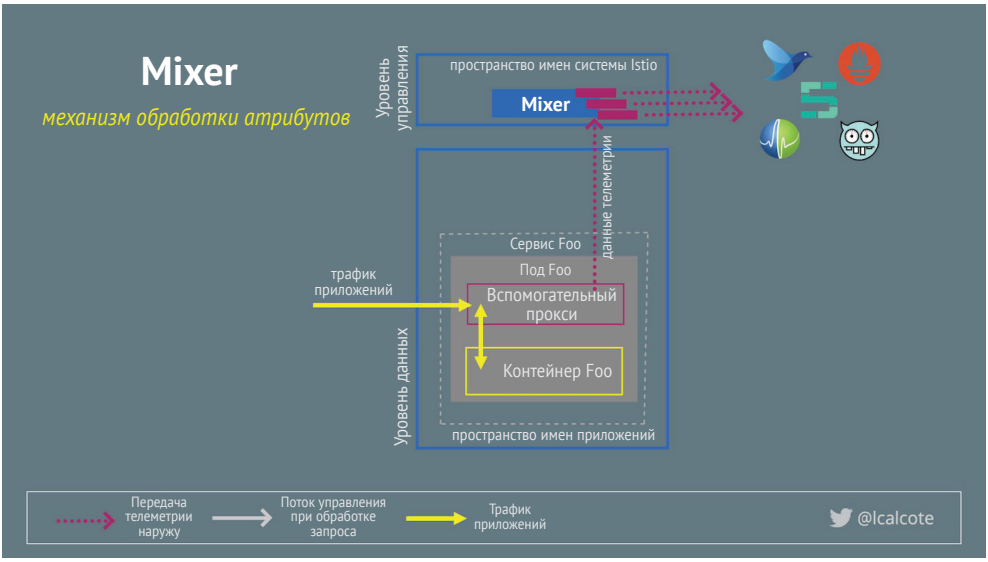

Рис. 3.6 ❖ Mixer выступает в роли движка для обработки атрибутов, сбора, преобразования и передачи данных телеметрии

Масштабируемость и производительность

Как и многие, вы можете задаться вопросом: «Эти функции великолепны, но какие накладные расходы связаны с запуском сервисной сетки?» Да, эти функции стоят своих денег. Работа прокси возле каждого сервиса и перехват каждого пакета будут постоянно потреблять ресурсы. Затраты можно анализировать по уровням данных и управления. И ответы на вопросы о производительности всегда начинаются с «Это зависит от...». Здесь, в зависимости от количества используемых функций Istio, потребуются различные ресурсы. Некоторые затраты/ресурсы можно оценить следующим образом:

- 1 виртуальный процессор (vCPU) на пиковую тысячу запросов в 1 с на прокси с журналированием доступа (который отключен по умолчанию в версии 1.1) и 0,5 – без него. Fluentd на узле вносит большой вклад в эту стоимость, так как он собирает и выгружает журналы;
- предполагая типичное соотношение попаданий в кеш (> 80 %) для проверок Mixer: 0,5 vCPU на пиковую тысячу запросов в 1 с для подов Mixer;
- в 90 % случаев добавляется дополнительная задержка около 8 мс;
- взаимные затраты на TLS (mTLS) незначительны для оборудования с поддержкой AES-NI как с точки зрения CPU, так и с точки зрения задержек.

Накладные расходы для уровней данных и управления являются общей проблемой для тех, кто ставит на вооружение сервисную сетку. Что касается уровня данных, разработчики Envoy понимают, что этот прокси находится в критической точке, и работают над оптимизацией его производительности. Envoy обладает первоклассной поддержкой протоколов HTTP/2 и TLS в любом

направлении и минимизирует потребление ресурсов за счет мультиплексирования запросов и ответов по одиночным, долговечным TCP-соединениям. Таким образом, поддержка HTTP/2 достигает меньших задержек, чем это было бы при использовании HTTP/1.1.

Хотя разработчики проекта Envoy в настоящее время не публикуют никаких официальных контрольных показателей эффективности, пользователей призывают замерять их в своих собственных условиях со структурой, аналогичной запланированной к работе. Чтобы заполнить эту пустоту, в сообществе открытого исходного кода появились такие инструменты, как Meshery (https://oreil.ly/MNUQC). Meshery – это мультисервисный уровень администрирования, предоставляющий различные сервисные сетки с примерами приложений и сравнивающий их производительность. Он облегчает сравнение различных конфигурационных сценариев Istio, а также производительности сервисов (приложений) в сетке, вне ее и в разных сетках. Он проверяет конфигурацию сетки и сервисов на соответствие лучшим практикам развертывания. Отдельные проекты сервисных сеток в качестве инструмента сравнения релизов используют Meshery. Он дополняется другими инструментами генерации нагрузки (https://oreil.ly/pZxU9), обычно используемыми для тестирования производительности сервисной сетки.

В крупных приложениях, по мере увеличения числа прокси (Envoy), уровень управления, играющий центральную роль, может стать узким местом или источником задержек. Например, в зависимости от типа инструментария и частоты выборки объем данных трассировки может превысить объем фактического делового трафика, получаемого приложением. Сбор этой телеметрии и отправка ее в систему трассировки (непосредственно или через уровень управления) могут оказать значительное влияние на задержку и пропускную способность приложения.

Модели развертывания

Istio поддерживает различные модели развертывания; некоторые из них устанавливают только отдельные компоненты архитектуры. На рис. 3.7 показано, как выглядит полное развертывание Istio во всей его красе.

Сервисные сетки бывают различных форм и размеров. Информацию о других моделях развертывания сеток можно почерпнуть в *The Enterprise Path to Service Mesh Architectures* (https://oreil.ly/70pu7). Теперь вы готовы перейти на следующий уровень и узнать больше о сервисных сетках в целом и об Istio в частности как инструментах, упрощающих управление сервисами. Istio стремится напрямую решать проблемы управления сервисами, обеспечивая новый уровень видимости, безопасности и контроля.

В этой главе мы рассмотрели, как уровень управления Istio обеспечивает единую точку видимости и контроля, а уровень данных облегчает маршрутизацию трафика. Кроме того, Istio является примером сервисной сетки, разработанной с учетом настраиваемости. Наконец, мы показали, как новый уровень разделения управления и унификации ответственности за микросервисы по-

зволяет избежать поиска виновных между командами разработчиков и администраторов.

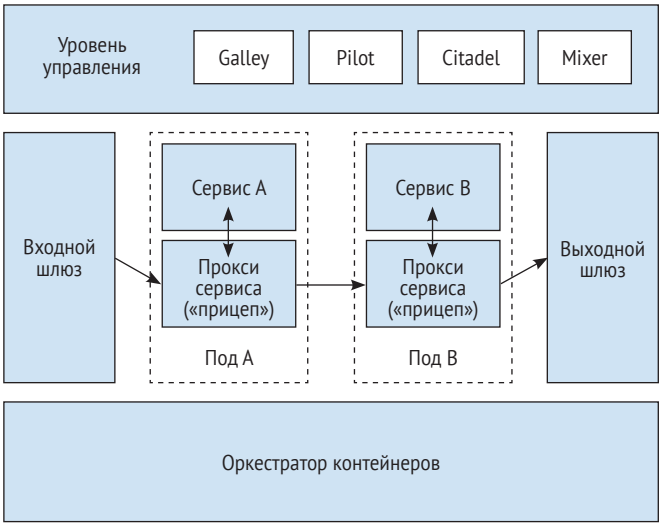

Рис. 3.7 ❖ Развертывание Istio в Kubernetes

Глава 4

Развертывание Istio

Для знакомства с полным набором возможностей сетки Istio нужно запустить ее в работу. Давайте начнем с изучения поддерживаемых платформ и подготовки к развертыванию. Istio – это крупный проект, предоставляющий массу возможностей и вариантов развертывания. В этой главе мы выполним базовую установку на локальную машину и развернем несколько сервисов в сетке. В последующих главах более подробно рассмотрим различные аспекты работы Istio.

Подготовка окружения для Istio

Кроме Istio развернем также пример приложения Bookinfo. В ходе развертывания Istio и Bookinfo будет размещено несколько контейнеров. Мы используем Kubernetes (https://kubernetes.io/) в качестве платформы для управления контейнерами. Kubernetes – надежная система управления контейнерами, способная формировать кластеры (наборы узлов) и распределять контейнеры по узлам в пределах парка машин (узлов), входящих в кластер. Узлы – это серверы Linux или Windows, способные работать с контейнерами, на которых установлен kubelet, агент Kubernetes. Являясь одной из нескольких поддерживаемых платформ, Kubernetes – первая и наиболее полно поддерживаемая платформа. Итак, в примерах мы используем Kubernetes. Понятно, что Istio не зависит от Kubernetes. Будучи платформонезависимой, Istio поддерживает множество платформ развертывания, включая платформы без оркестратора контейнеров.

Docker Desktop как среда установки

Существует множество вариантов развертывания Kubernetes. В этой книге для удобства мы используем Docker Desktop. Docker Desktop – простое в установке приложение для Mac или Windows, позволяющее запускать Kubernetes и Istio на локальной машине.

Чтобы установить Docker Desktop и убедиться в работоспособности окружения Docker, в командной строке запустите `docker run hello-world`. Сообщение «`Hello from Docker!`» подтвердит, что Docker смог извлечь образы, создать их экземпляры и работает как положено.

Мы выбрали Docker Desktop (https://oreil.ly/HRKvu) в качестве платформы Kubernetes из-за удобства и необходимости устанавливать Istio вручную, чтобы показать некоторые внутренние тонкости и при этом не суетиться с установкой кластера Kubernetes. Также можно использовать Meshery (https://oreil.ly/0Oa9X), быстро развертывающий Istio и пример приложения Bookinfo. Независимо от того, какой инструмент вы используете для развертывания Istio, примеры, приводимые здесь, должны действовать в любой среде Istio, работающей на Kubernetes. Список поддерживаемых платформ см. в документации Istio (https://oreil.ly/LeGw5).

По состоянию на июль 2018 г. Docker Desktop для Mac и Windows включает поддержку работы автономного сервера и клиента Kubernetes, а также интеграцию с Docker CLI. Мы используем Docker Desktop для запуска Kubernetes и Kubernetes в качестве платформы для развертывания Istio. Сервер Kubernetes, управляемый из Docker Desktop, работает локально в вашем экземпляре Docker, не конфигурируется и представляет собой одноузловой кластер Kubernetes.

Интеграция Docker Desktop с Kubernetes для Mac обеспечивается утилитой командной строки */usr/local/bin/kubectl*. Интеграция Docker Desktop с Kubernetes для Windows обеспечивается утилитой командной строки *C:\Program Files\Docker\Docker\Resources\bin\kubectl\kubectl.exe*. Этот путь может быть не прописан в переменной окружения `PATH`; если это так, введите в командной строке полный путь к команде или добавьте его в `PATH`. Более подробную информацию о `kubectl` смотрите в официальной документации (https://oreil.ly/BaEUI).

Конфигурирование Docker Desktop

Чтобы убедиться, что в виртуальной машине Docker Desktop VM достаточно памяти для запуска Kubernetes, Istio и приложения Bookinfo, виртуальной машине Docker Desktop VM нужно выделить в настройках как минимум 4 Гб памяти. Этот объем необходим для работы всех сервисов Istio и Bookinfo. В частности, у программы Pilot могут возникнуть проблемы с работой, так как она требует 2 Гб памяти при установке Istio с настройками по умолчанию (для быстрого ознакомления с предназначением программы Pilot см. главу 3). По умолчанию Docker Desktop требует еще 2 Гб, поэтому, если объем выделенной памяти окажется меньше 4 Гб, Pilot может отказать в запуске из-за нехватки ресурсов.

Вместо увеличения объема памяти, выделяемой для Docker Desktop, как показано на рис. 4.1, можно уменьшить объем, запрашиваемый Pilot у кластера Kubernetes. Есть несколько способов это сделать, в зависимости от того, используется ли диспетчер пакетов, такой как Helm, или файлы спецификации Kubernetes.

В примере 4.1 показан фрагмент из файла манифеста *install/kubernetes/istio-demo.yaml* с настройками для Pilot, где объем запрашиваемой памяти уменьшен с 2 Гб до 512 Мб.

Пример 4.1 ❖ Фрагмент istio-demo.yaml с настройками объема памяти, запрашиваемого для контейнера Pilot

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: istio-pilot
```

```
  namespace: istio-system
...
    resources:
      requests:
      cpu: 500m
      memory: 2048Mi
...
```

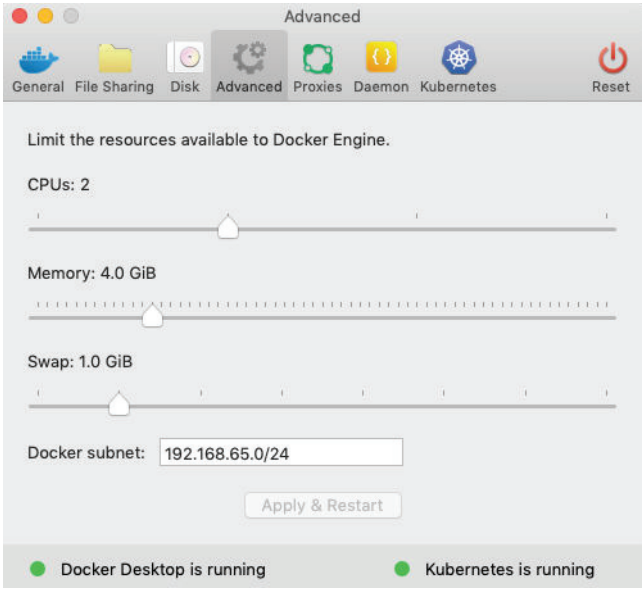

Рис. 4.1 ❖ Увеличение лимита памяти Docker в панели Advanced

Если развертывание сетки Istio осуществляется с помощью Helm (диспетчера пакетов Kubernetes), необходимые настройки можно выполнить с его помощью. Для этого передайте команде Helm один или несколько параметров `--set <key>=<value>` с нужными настройками, как показано в примере 4.2.

Пример 4.2 ❖ Настройка конфигурации Istio с помощью Helm

```
$ helm template install/kubernetes/helm/istio --name istio --namespace istio-system
     --set pilot.resources.requests.memory="512Mi" | kubectl apply -f -
```

Развертывание Kubernetes

Если Kubernetes не установлен на вашем компьютере, включите его в настройках Docker Desktop. Проверьте правильность установки `kubectl`, как показано в примере 4.3.

Пример 4.3 ❖Получение версий клиента (бинарный kubectl) и сервера (кластер Kubernetes)

```
$ kubectl version -short
Client Version: v1.13.0
Server Version: v1.13.0
```

Если эта команда вывела номера версий клиента и сервера, значит, путь к клиенту `kubectl` прописан в `PATH`, и кластер Kubernetes доступен. Проверьте установку Kubernetes и текущий контекст, выполнив команду `$ kubectl get nodes` (см. пример 4.4), и убедитесь, что *kubeconfig* (обычно находящийся в *~/.kube/config*) правильно настроен в контексте `docker-desktop` и кластер с единственным узлом запущен и работает.

Пример 4.4 ❖ Список узлов Kubernetes, полученный с помощью kubectl

```
$ kubectl get nodes
NAME             STATUS   ROLES    AGE   VERSION
docker-desktop   Ready    master   32m   v1.13.0
```

Эта установка создает в кластере примеры учетных записей, ключей, приложений и других объектов. Созданные объекты предназначены только для иллюстративных и образовательных целей. Не используйте эту конфигурацию в рабочей среде.

Установка Kubernetes Dashboard

Kubernetes Dashboard – это веб-интерфейс пользователя для управления кластером и его ресурсами. Его можно использовать для развертывания и сопровождения контейнеризированных приложений. Kubernetes Dashboard также предоставляет информацию о состоянии ресурсов Kubernetes в кластере и о любых возникающих ошибках. Dashboard полезен для лучшего понимания работы Istio. Самый простой и распространенный способ получить доступ к кластеру – создать локальный веб-сервер командой `kubectl proxy`, безопасно передающий данные в Dashboard через Kubernetes API Server. Разверните Kubernetes Dashboard, выполнив следующую команду:

```
$ kubectl create -f https://raw.githubusercontent.com/kubernetes/dashboard/master/aio/
    deploy/recommended/kubernetes-dashboard.yaml
```

Затем запустите Dashboard, выполнив следующую команду:

```
$ kubectl proxy
```

Она запустит локальный веб-сервер, безопасно передающий данные Dashboard через Kubernetes API Server. Помните, что доступ к Dashboard возможен только с компьютера, на котором была выполнена команда. Дополнительную информацию о поддерживаемых параметрах можно узнать с помощью команды `kubectl proxy --help` или в документации Kubernetes Dashboard (https://oreil.ly/El-cx). На рис. 4.2 показано, как выглядит интерфейс Dashboard.

Чтобы защитить данные кластера, Dashboard по умолчанию развертывается с настройками роли, имеющей минимально возможные привилегии (*Role-based access control*, RBAC). В настоящее время Dashboard поддерживает вход, только с использованием токена на предъявителя. Можно создать свою учетную запись (https://oreil.ly/qwiNm) и использовать ее токен, либо использовать существующий токен, предоставляемый установкой Docker Desktop, как показано здесь:

```
$ kubectl -n kube-system describe secret $(kubectl -n kube-system get secret |
      awk '/default-token/ {print $1}')
```

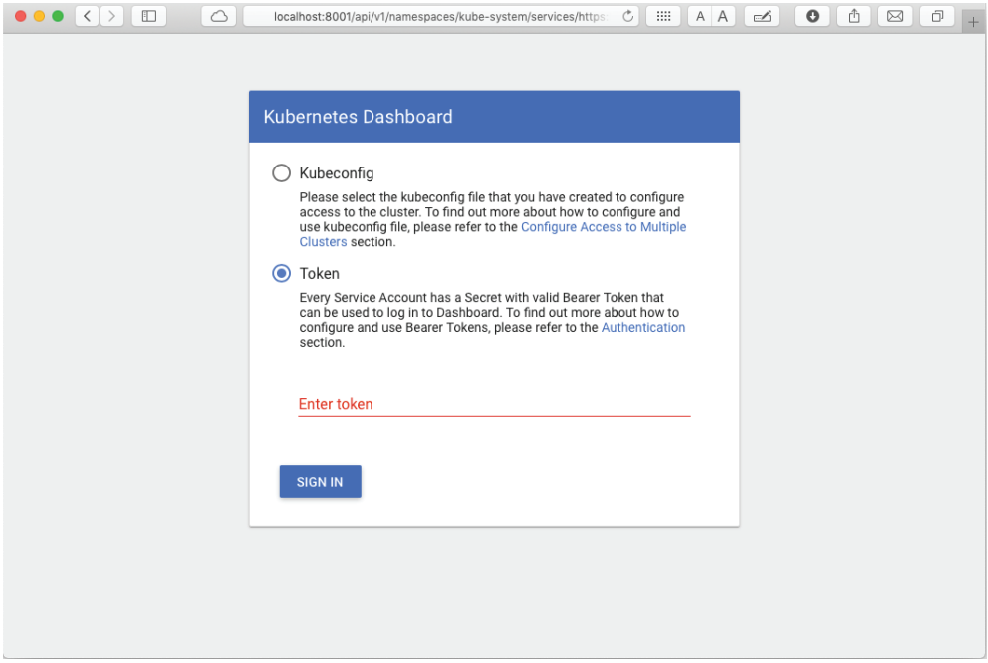

Рис. 4.2 ❖ Аутентификация в Kubernetes Dashboard

В выводе этой команды вы найдете требуемый токен, как показано в примере 4.5.

Пример 4.5 ❖ Вывод описания секрета Kubernetes

```
Name:         default-token-tktcn
Namespace:    kube-system
Labels:       <none>
Annotations: kubernetes.io/service-account.name: default
             kubernetes.io/service-account.uid: 3a0a68b1-4abd-11e9-8561-025...

Type: kubernetes.io/service-account-token

Data
====
ca.crt:     1025 bytes
namespace:  11 bytes
token:      eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZp...
```

Скопируйте токен для авторизации в Dashboard.

Установка Istio

После развертывания Kubernetes и запуска Dashboard можно приступать к установке сервисной сетки. Последнюю версию Istio можно загрузить командой:

```
$ curl -L https://git.io/getLatestIstio | sh - Sh -
```

Она загрузит и распакует последнюю версию Istio.

Пользователям Mac и Linux команда `curl` доступна по умолчанию. Пользователям Windows может понадобиться загрузить версию curl для Windows (https://oreil.ly/2QiWB).

Чтобы получить определенную версию Istio, укажите желаемый номер версии, как показано здесь:

```
$ curl -L https://git.io/getLatestIstio | ISTIO_VERSION=1.1.0 sh -
```

Проект Istio обеспечивает обратную совместимость между младшими версиями (если иное не утверждается в журнале изменений), тем не менее, чтобы гарантировать правильное функционирование всех примеров в книге, можно указать `ISTIO_VERSION=1.1.0`.

Скачать Istio также можно со страницы со списком всех выпускавшихся версий (https://oreil.ly/6GzRG). Здесь вы сможете выбрать версию для Windows, MacOS и Linux. Независимо от используемой операционной системы, после загрузки дистрибутива распакуйте его и ознакомьтесь с содержимым.

Каждый релиз включает `istioctl` (двоичный файл, соответствующий выбранной ОС), примеры конфигурации, пример приложения и установочные ресурсы для конкретной платформы. Кроме того, `istioctl` – важный инструмент командной строки для администраторов, помогающий в отладке и диагностике сервисных сеток Istio, – можно установить с помощью диспетчера пакетов.

После загрузки не забудьте перенести дистрибутив в папку, где он не будет удален.

Например, чтобы исследовать содержимое дистрибутива в MacOS или Linux, перейдите в каталог `istio-x.x.x`, как показано здесь (на момент написания этой книги последней была версия Istio 1.1.7):

```
$ cd istio-1.1.0
```

В этом каталоге (см. пример 4.6) находятся файлы, необходимые для установки Istio, примеры файлов, а также `istioctl`.

Пример 4.6 ❖ Содержимое папки верхнего уровня в дистрибутиве Istio

```
$ ls -l
total 48
-rw-r--r--   1 lee   staff   11343   Mar 18 16:08 LICENSE
-rw-r--r--   1 lee   staff    5921   Mar 18 16:08 README.md
drwxr-xr-x   3 lee   staff      96   Mar 18 16:08 bin
drwxr-xr-x   7 lee   staff     224   Mar 18 16:08 install
-rw-r--r--   1 lee   staff     602   Mar 18 16:08 istio.VERSION
drwxr-xr-x  16 lee   staff     512   Mar 18 16:08 samples
drwxr-xr-x  21 lee   staff     672   Mar 18 16:08 tools
```

В каталоге install/ находятся установочные инсталляционные YAML-файлы Istio для Kubernetes, в *samples/* – примеры приложений и в bin/ – двоичный

файл клиента `istioctl`. Конфигурационный файл *istio.VERSION* содержит список компонентов Istio с номерами версий, включенных в дистрибутив.

Утилиту `istioctl` можно использовать для создания правил и политик маршрутизации, а также для внедрения экземпляров прокси Envoy вручную. Другие области применения включают создание, перечисление, изменение и удаление ресурсов конфигурации в системе Istio. Давайте добавим путь к этой утилите в переменную окружения `PATH`:

```
$ export PATH=$PWD/bin:$PATH
```

Правильность установки `istioctl` можно проверить, запустив:

```
$ istioctl version
```

Ответ должен подтвердить правильность настройки пути и параметров команды `istioctl` (см. пример 4.7). Если нет, смотрите раздел «*Istio: установка, обновление и удаление*» в главе 11.

Пример 4.7 ❖Проверка правильности установки istioctl на данной машине

```
version.BuildInfo{
Version:"1.1.0", GitRevision:"82797c0c0649a3f73029b33957ae105260458c6e",
User:"root",
Host:"996cd064-49c1-11e9-813c-0a580a2c0506", GolangVersion:"go1.10.4", DockerHub:"docker.
io/istio",
BuildStatus:"Clean",
GitTag:"1.1.0-rc.6"
}
```

Загрузив дистрибутив Istio и убедившись, что `istioctl` доступна из командной строки, давайте выполним базовую установку.

Параметры установки Istio

Существуют различные методы установки и архитектуры развертывания. Как правило, установки делятся на следующие категории.

По уровню безопасности

Со строгой mTLS-аутентификацией

Рекомендуется для нового кластера Kubernetes. Этот метод по умолчанию обеспечивает аутентификацию между вспомогательными прокси.

С разрешительной mTLS-аутентификацией между вспомогательными прокси:

- рекомендуется, если у вас есть действующие кластеры и сервисы;
- используйте этот способ, если сервисам вашего приложения со вспомогательными прокси Istio необходимо взаимодействовать с сервисами, действующими вне Istio и Kubernetes.

Нестандартная, включающая или исключающая некоторые компоненты Istio:

- рекомендуется, если функция одного из компонентов Istio не нужна или нежелательна в вашем окружении (например, можно исключить компонент Citadel, если не будет использоваться mTLS).

В зависимости от выбранной утилиты развертывания

При выборе утилиты установки учитывайте следующее:

- генерируются ли манифесты Kubernetes прямым вызовом kubectl;
- имеется ли четкое понимание основ Istio;
- будут ли генерироваться манифесты Kubernetes с помощью таких систем управления пакетами/конфигурацией, как Helm или Ansible;
- предполагается ли выполнять развертывание в производственном окружении с шаблонной конфигурацией.

Этот список не является исчерпывающим, однако, независимо от выбранного подхода к установке Istio, каждый из них будет включать установку определений нестандартных ресурсов (*custom resource definitions, CRD*) Kubernetes для Istio. CRD позволяют определять и регистрировать нестандартные ресурсы Kubernetes. При развертывании Istio CRD-объекты регистрируются как объекты Kubernetes, что обеспечивает высокую степень интеграции с Kubernetes как платформой развертывания.

Что такое CRD?

Поддержка CRD, появившаяся в Kubernetes 1.7, является расширением Kubernetes API. Определения CRD позволяют добавлять нестандартные объекты в кластер Kubernetes. *Ресурс* – это конечная точка в Kubernetes API, хранящая коллекцию объектов определенного типа. Например, встроенный ресурс **pods** содержит коллекцию объектов `Pod`.

Стандартный дистрибутив Kubernetes включает множество встроенных ресурсов (объектов). Поддержка определения нестандартных ресурсов – это расширение Kubernetes API, надстройка, необязательно доступная в стандартной установке Kubernetes. Однако многие основные функции Kubernetes сегодня собраны с использованием CRD, что делает Kubernetes более модульным.

CRD широко используются, когда требуется внедрить свои объекты в кластер Kubernetes. После применения CRD в кластере Kubernetes они хранятся в кластере `etcd`, обеспечивающем их сохранность за счет репликации и управления жизненным циклом. В Kubernetes API (`kube-api`) эти новые ресурсы доступны как конечные точки, которые можно использовать подобно любым другим встроенным объектам Kubernetes (таким как `Pod`). Таким образом, для взаимодействия со своими ресурсами и управления ими можно использовать все возможности Kubernetes, такие как инструменты командной строки, средства поддержки безопасности, интерфейс доступа к сервисам, RBAC и многие другие, экономя силы и время на их реализации.

Дополнительные сведения см. в документации Kubernetes Custom Resources (https://oreil.ly/rKU4N).

Регистрация нестандартных ресурсов Istio

Теперь, понимая силу и удобство определений CRD, имеющихся в нашем распоряжении, давайте зарегистрируем определения `CustomResourceDefinitions` объектов Istio в кластере Kubernetes:

```
for i in install/kubernetes/helm/istio-init/files/crd*yaml;
    do kubectl apply -f $i; done
```

В этом примере не используется диспетчер пакетов Helm. Однако при развертывании Istio в производственном окружении желательно использовать Helm или Ansible; оба включены в только что загруженный дистрибутив. Helm или Ansible дают гибкость выбора устанавливаемых компонентов и позволяют настраивать все детали.

Пример 4.8 ❖ CRD Istio, загруженные в виде ресурсов в cube-api и доступные для взаимодействия

```
$ kubectl api-resources | grep istio
meshpolicies        authentication.istio.io   false   MeshPolicy
policies            authentication.istio.io   true    Policy
adapters            config.istio.io           true    adapter
apikeys             config.istio.io           true    apikey
attributemanifests  config.istio.io           true    attributemanifest
authorizations      config.istio.io           true    authorization
bypasses            config.istio.io           true    bypass
checknothings       config.istio.io           true    checknothing
circonuses          config.istio.io           true    circonus
cloudwatches        config.istio.io           true    cloudwatch
...
```

Istio регистрирует новые CRD (см. пример 4.8), которыми можно манипулировать (создавать/обновлять/удалять), как и любыми другими объектами Kubernetes:

```
$ kubectl get crd | grep istio
adapters.config.istio.io              2019-03-24T03:17:08Z
apikeys.config.istio.io               2019-03-24T03:17:07Z
attributemanifests.config.istio.io    2019-03-24T03:17:07Z
authorizations.config.istio.io        2019-03-24T03:17:07Z
bypasses.config.istio.io              2019-03-24T03:17:07Z
checknothings.config.istio.io         2019-03-24T03:17:07Z
circonuses.config.istio.io            2019-03-24T03:17:07Z
cloudwatches.config.istio.io          2019-03-24T03:17:08Z
clusterrbacconfigs.rbac.istio.io      2019-03-24T03:17:07Z
deniers.config.istio.io               2019-03-24T03:17:07Z
destinationrules.networking.istio.io  2019-03-24T03:17:07Z
dogstatsds.config.istio.io            2019-03-24T03:17:08Z
edges.config.istio.io                 2019-03-24T03:17:08Z
envoyfilters.networking.istio.io      2019-03-24T03:17:07Z
...
```

После установки CRD к ресурсам Istio можно обратиться с помощью `istioctl` или `kubectl`, как показано в примере 4.9.

Пример 4.9 ❖ Использование istioctl для отображения информации о шлюзах Istio

```
$ istioctl get gateway
Command "get" is deprecated, Use `kubectl get` instead (see https://kubernetes.io/docs/
tasks/tools/install-kubectl)

No resources found.
```

Обратите внимание на сообщение о нежелательности использования команд, определяемых утилитой `istioctl`. Имейте в виду, что хотя `istioctl` все еще поддерживается и совершенствуется в качестве инструмента командной строки для управления Istio, для взаимодействий с нестандартными ресурсами Istio лучше использовать `kubectl` (см. пример 4.10). `istioctl` предоставляет некоторые специфические для Istio утилиты, не поддерживаемые в `kubectl`.

Пример 4.10 ❖ Использование kubectl для отображения информации о шлюзах Istio

```
$ kubectl get gateway
NAME               AGE
No resources found.
```

В качестве альтернативы циклу по манифестам *install/kubernetes/helm/istio-init/files/crd*yaml* можно использовать *istio-demo.yaml*, подключающий те же CRD-манифесты. Файл *istio-demo.yaml* также содержит все компоненты уровня управления Istio (не только CRD). В папке *install/*, в каталоге дистрибутива, находятся установочные файлы Istio для различных поддерживаемых платформ. Учитывая, что в этой книге выбрана платформа Kubernetes, откройте папку *install/kubernetes/*. Здесь находится файл *istio-demo.yaml*, содержащий все необходимые компоненты Istio (CRD, определения ролей, конфигурационные карты, сервисы, HPA, настройки развертывания и т. д.) и несколько полезных адаптеров, таких как Grafana и Prometheus.

После регистрации нестандартных ресурсов Istio в Kubernetes можно установить компоненты уровня управления Istio.

Установка компонентов уровня управления

Мы используем файл спецификации *istio-demo.yaml*, содержащий конфигурации Istio, позволяющие сервисам работать в разрешительной аутентификации mTLS. Использование разрешительного режима mTLS рекомендуется, если в кластере Kubernetes уже есть сервисы или приложения. Однако, если начинать с нового кластера, рекомендуется перейти на `istio-demo-auth.yaml` для принудительного шифрования трафика между машинами.

```
$ kubectl apply -f install/kubernetes/istio-demo.yaml
```

Подождите несколько минут, пока завершатся установка, загрузка образов Docker и успешное развертывание. Применение этого обширного YAML-файла позволяет Kubernetes реализовать множество новых CRD.

Установка Istio с принудительной аутентификацией mTLS

При настройке Istio для работы в разрешительном режиме mTLS, сервис может принимать как текстовый, так и mTLS-трафик в зависимости от типа трафика клиента. Разрешительный mTLS-режим способствует миграции существующих сервисов в сервисную сетку. Итак, при использовании существующей конфигурации Kubernetes с запущенными сервисами можно выбрать установку Istio с разрешительным режимом mTLS.

В качестве альтернативы, и особенно в рамках данной локальной установки, можно использовать *istio-demo-auth.yaml*, принудительно устанавливающий mTLS-аутентификацию

между всеми клиентами и серверами. Можно подумать, что начальное развертывание Istio со строгой аутентификацией mTLS наиболее успешно используется в свежем кластере Kubernetes, где все рабочие нагрузки будут действовать под управлением Istio. Чтобы применить конфигурацию Istio с принудительной mTLS-аутентификацией, выполните следующую команду:

```
$ kubectl apply -f install/kubernetes/istio-demo-auth.yaml
```

Обратите внимание на применение `tls.mode: ISTIO_MUTUAL` в файле *istio-demo-auth.yaml*.

Используя разрешительный режим mTLS, администраторы могут постепенно устанавливать и настраивать прокси сервисов в Istio для взаимной аутентификации запросов. Завершив настройку всех сервисов, администратор сможет настроить Istio на принудительное использование mTLS-режима. Дополнительные сведения приведены в разделе «*mTLS*».

Уровень управления Istio установлен в собственном пространстве имен `istio-system` и из этого пространства управляет сервисами, работающими во всех других пространствах имен, имеющих сервисы с прокси, или, другими словами, во всех других пространствах имен, имеющих сервисы в сетке. Уровень управления действует в масштабах всего кластера, что означает, что он ведет себя как один пользователь, как показано в примере 4.11.

Пример 4.11 ❖ Пространство имен istio-system, созданное для компонентов уровня управления в Istio

```
$ kubectl get namespaces
NAME           STATUS   AGE
default        Active   49d
docker         Active   49d
istio-system   Active   2m15s
kube-public    Active   49d
kube-system    Active   49d
```

Проверить правильность установки уровня управления в пространство имен `istio-system` можно с помощью команд, приведенных в примере 4.12. Если установка прошла успешно, вы увидите аналогичный вывод.

Пример 4.12 ❖ Сервисы и блоки управления Istio, работающие в пространстве имен системы istio

```
$ kubectl get svc -n istio-system
NAME          TYPE      CLUSTER-IP     EXTERNAL-IP PORT(S)           AGE
grafana       ClusterIP 10.108.237.105 <none>   3000/TCP             11d
istio-citadel ClusterIP 10.108.165.14  <none>   8060/TCP,15014/TCP   11d
istio-egressgateway ClusterIP 10.107.148.169 <none>   80/TCP,443/TCP,15443/TCP 11d
...

$ kubectl get pod -n istio-system
NAME                                  READY   STATUS      RESTARTS  AGE
grafana-57586c685b-jr2pd              1/1     Running     0         5m45s
istio-citadel-645ffc4999-8j4v6        1/1     Running     0         5m45s
istio-cleanup-secrets-1.1.0-4c9pc     0/1     Completed   0         5m48s
istio-egressgateway-5c7fd57fdb-85g26  1/1     Running     0         5m46s
istio-galley-978f9447f-mj5xj          1/1     Running     0         5m46s
```

```
istio-grafana-post-install-1.1.0-g49gh    0/1     Completed   0          5m48s
istio-ingressgateway-8ccdc79bc-8mk4p      1/1     Running     0          5m46s
istio-pilot-649455846-klc8c               2/2     Running     0          5m45s
istio-policy-7b7d7f644b-sqsp8             2/2     Running     4          5m45s
istio-security-post-install-1.1.0-v4ffp   0/1     Completed   0          5m48s
istio-sidecar-injector-6dcc9d5c64-tklqz   1/1     Running     0          5m45s
istio-telemetry-6d494cd676-n6pkz          2/2     Running     4          5m45s
istio-tracing-656f9fc99c-nn9hd            1/1     Running     0          5m44s
kiali-69d6978b45-7q7ms                    1/1     Running     0          5m45s
prometheus-66c9f5694-2xzpm                1/1     Running     0          5m45s
```

Но почему каждый компонент уровня управления в Istio создан в единственном экземпляре? Разве при этом не образуется единственная точка отказа? Да, верно. Наличие единственного экземпляра компонента в производственном окружении представляет определенную проблему. Уровень управления Istio должен развертываться в архитектуре с высокой доступностью (с несколькими экземплярами каждого компонента) в любой среде, где любые задержки недопустимы. Вас не беспокоит, что уровень данных является встроенным? Что делать, если он выйдет из строя или потеряет связь с уровнем управления? К счастью, уровень данных имеет встроенные механизмы поддержки отказоустойчивости, способные сохранять конфигурацию и восстанавливать работоспособность компонентов в случае сбоя или отключения от уровня управления. По мере дальнейшего чтения книги будет видно, что Istio спроектирована очень отказоустойчивой. Как и должно быть, потому что сбои в работе распределенных систем почти гарантированы.

До сих пор мы развернули только половину сервисной сетки – уровень управления. Прежде чем устанавливать пример приложения и, следовательно, уровень данных, можно подумать, что прокси еще не запущены, и не заметить, что два из них уже действуют. Входной и выходной шлюзы настроены и работают как экземпляры прокси. Давайте проверим.

Этот простой демонстрационный пример включает в себя выходной шлюз для ознакомления и облегчения исследования Istio. Этот дополнительный шлюз отключен по умолчанию, начиная с версии v1.1. Если нужно проконтролировать и защитить исходящий трафик, следует вручную включить настройку `gateways.istio-egressgateway.enabled=true` в любых недемонстрационных конфигурационных профилях развертывания.

Использование команды `istioctl proxy-status` позволяет получить обзор сетки; это пример утилиты `istioctl`, отсутствующей в `kubectl`, как показано в примере 4.13. Если есть подозрение, что один из прокси не настроен или не синхронизирован, команда `proxy-status` сообщит об этом. Мы продолжим использовать `istioctl` для отладки Istio в главе 11.

Пример 4.13 ❖ Подтверждение того, что входной и выходной шлюзы Istio имеют вспомогательные прокси, установленные и синхронизированные с использованием istioctl proxy-status

```
$ istioctl proxy-status
NAME                    CDS    LDS    EDS       RDS       PILOT           VERSION
istio-egressgateway-... SYNCED SYNCED SYNCED... NOT SENT  istio-pilot-... 1.1.0
istio-ingressgateway-... SYNCED SYNCED SYNCED... NOT SENT  istio-pilot-... 1.1.0
```

Понимание, как Istio управляет конфигурацией прокси Envoy, развернутых в качестве шлюзов в уровне управления, проливает свет на то, как осуществляется управление экземплярами Envoy в уровне данных. Как вы помните, уровень данных состоит из интеллектуальных прокси, развернутых с использованием шаблона «Прицеп» (Sidecar) рядом с сервисами приложения. Давайте развернем уровень данных. Для этого установим пример приложения Bookinfo.

Развертывание образца приложения Bookinfo

Развернем первый набор сервисов (приложения) в сервисной сетке. Для этого используем образец приложения Bookinfo, разработанного для демонстрации многих важных аспектов сервисных сеток. Файлы манифестов Kubernetes для Bookinfo находятся в папке дистрибутива: *samples/bookinfo/*. Раз уж в книге в качестве примера приложения используется Bookinfo, давайте познакомимся с ним поближе.

На рис. 4.3, слева направо: пользователи вызывают микросервис `productpage`, который, в свою очередь, вызывает `details` и просматривает микросервисы для заполнения страницы. Микросервис `details` содержит информацию о книге.

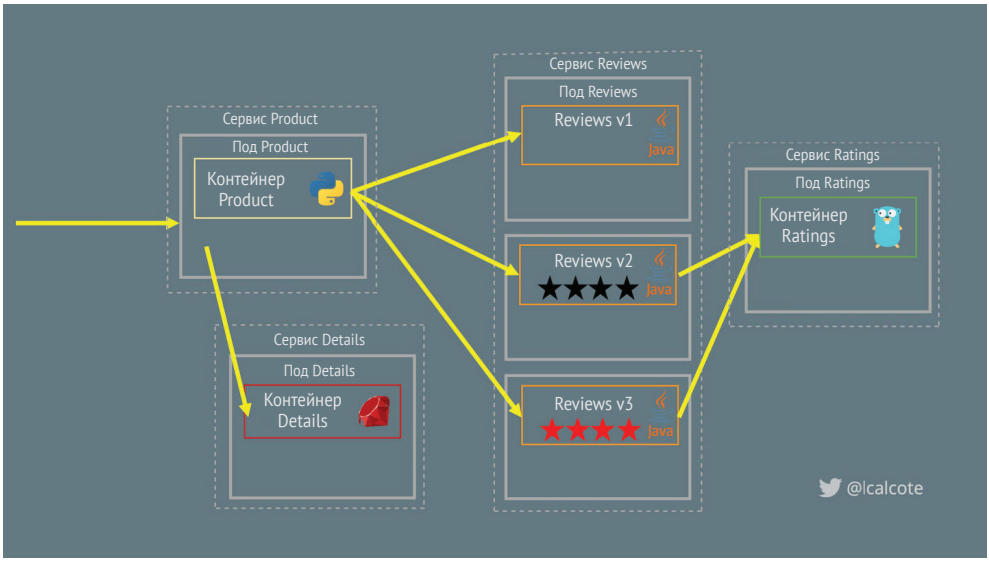

Рис. 4.3 ❖ Bookinfo: показаны отдельные микросервисы, иллюстрирующие различные возможности Istio

Микросервис `reviews` содержит обзоры книг и, чтобы их получить, вызывает микросервис `ratings`. Микросервис `ratings` содержит рейтинг книг в виде обзоров с рейтингом от 1 до 5 звезд. Микросервис `reviews` имеет три версии:

- `reviews` v1 не имеет рейтингов (не вызывает сервис `ratings`);
- `reviews` v2 имеет рейтинги от 1 до 5 черных звезд (вызывает сервис `ratings`);

- `reviews` v3 имеет рейтинги от 1 до 5 красных звезд (вызывает сервис `ratings`).

Все эти сервисы написаны на разных языках – Python, Ruby, Java и Node.js, что и демонстрирует ценность сервисной сетки.

Большинство проектов сервисных сеток включают пример приложения. Это делается для быстрого освоения пользователями сервисной сетки и демонстрации ее преимуществ.

При работе с примером в Istio не требуется вносить изменений в приложение. Достаточно просто настроить и запустить сервисы в сетке Istio и внедрить прокси во все сервисы. В сетке Istio прокси можно внедрять вручную или автоматически (см. рис. 4.4). Давайте посмотрим, как работает автоматическое внедрение прокси при установке примера приложения (внедрение вручную мы рассмотрим в главе 5).

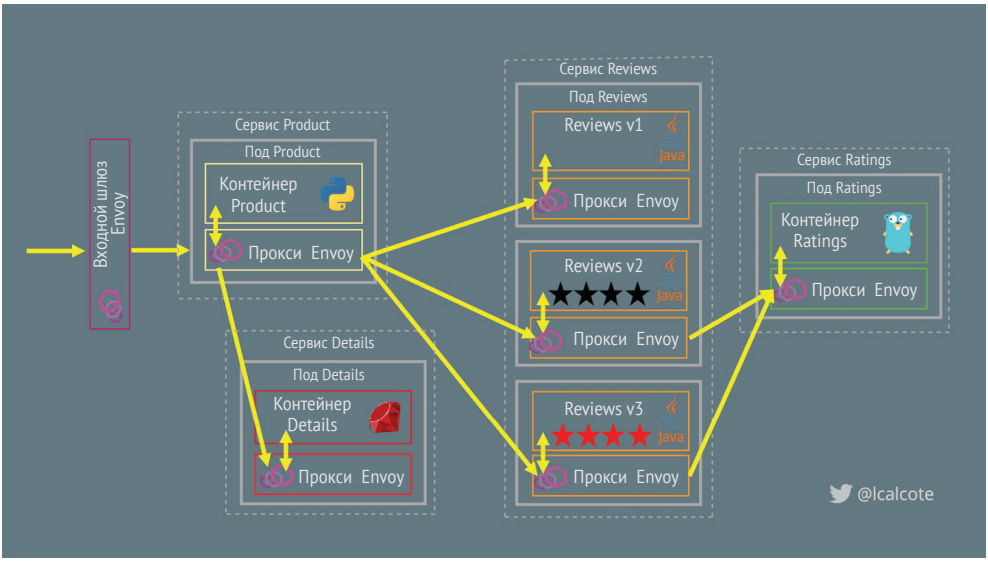

Рис. 4.4 ❖ Приложение Bookinfo развернуто в сетке Istio вместе со вспомогательными прокси

Развертывание примера приложения с автоматическим внедрением прокси

Для внедрения прокси Envoy в каждый сервис Istio развертывает специальный механизм внедрения. Более подробно этот механизм рассматривается в главе 5. А пока, как показано в примере 4.14, проверим наличие работающего механизма внедрения и метку его пространства имен, эта метка определяет пространство имен с подами, в которые прокси будут вставляться автоматически.

Пример 4.14 ❖ Проверка наличия механизма внедрения Istio

```
$ kubectl -n istio-system get deployment -l istio=sidecar-injector
NAME                     READY   UP-TO-DATE   AVAILABLE   AGE
istio-sidecar-injector   1/1     1            1           82m
```

Необходимость запуска веб-обработчика для объекта определяется соответствием селектора namespaceSelector пространству имен этого объекта (https://oreil.ly/rncCW).

Определяем метку пространства имен по умолчанию, как `istio-injection=enabled`:

```
$ kubectl label namespace default istio-injection=enabled
```

Теперь посмотрим, какие пространства имен имеют метку `istio-injection`:

```
$ kubectl get namespace -L istio-injection
NAME           STATUS    AGE       ISTIO-INJECTION
default        Active    1h        enabled
Docker         Active    1h        enabled
istio-system   Active    1h        disabled
kube-public    Active    1h
kube-system    Active    1h
```

Использованный нами файл манифеста *istio-demo.yaml* настроен на автоматическое внедрение.

Теперь, имея механизм внедрения с *изменяющим входным веб-обработчиком* (*mutating admission webhook*) и пространство имен с меткой для автоматического внедрения прокси, можно развернуть пробное приложение. Подробное описание того, что такое изменяющий входной веб-обработчик в Kubernetes, см. в разделе «*Автоматическое внедрение прокси*». Теперь для развертывания пробного приложения передадим в Kubernetes манифест Bookinfo:

```
$ kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml
```

После развертывания пробного приложения можно убедиться, что автоматическое внедрение прокси прошло успешно, осмотрев любой из подов Bookinfo и отметив наличие нового контейнера `istio-proxy` в поде с прикладным сервисом:

```
$ kubectl describe po/productpage-v1-....
...
istio-proxy:
    Container ID:  docker://f28abdf1f0acf92687711488f7fcca8cc5968e2ed39d…
    Image:         docker.io/istio/proxyv2:1.1.7
    Image ID:      docker-pullable://istio/proxyv2@sha256:e6f039115c7d5e…
    Port:          15090/TCP
    Host Port:     0/TCP
    Args:
      proxy
      sidecar
...
```

Прокси можно удалить из подов приложения, даже если пространство имен включает метку `sidecar-injection`. Мы рассмотрим эту процедуру в главе 5.

Работа примера приложения в сети

После запуска сервисов Bookinfo нужно открыть доступ к приложению из-за пределов кластера Kubernetes, например из браузера. Для этого используем Istio Gateway (https://oreil.ly/iUlwY). Определим входной шлюз для приложения:

```
$ kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml
```

Проверим, что шлюз создан:

```
$ kubectl get gateway

NAME               AGE
bookinfo-gateway   7m
```

Чтобы взаимодействовать с только что развернутым приложением, определите адрес сервиса `productpage`, на который можно отправлять запросы из-за пределов кластера. Как это сделать, показано в примере 4.15.

Пример 4.15 ❖ Определение IP-адреса и номера порта входного шлюза Istio для доступа к приложению

```
$ echo "http://$(kubectl get nodes -o template --template='{{range.items}}
    {{range.status.addresses}}{{if eq.type "InternalIP"}}{{.address}}{{end}}{{end}}
    {{end}}'):$(kubectl get svc istio-ingressgateway
    -n istio-system -o jsonpath='{.spec.ports[0].nodePort}')/
    productpage"

http://x.x.x.x:31380/productpage
```

Нужно также убедиться, что другие ваши приложения не используют те же сетевые порты, что и компоненты Istio. В табл. 4.1 перечислены порты, используемые Istio.

Таблица 4.1. Сетевые порты, используемые Istio

Port	Protocol	Used by	Description
8060	HTTP	Citadel	GRPC-сервер
9090	HTTP	Prometheus	Prometheus
9091	HTTP	Mixer	Политики/телеметрия
9093	HTTP	Citadel	TCP-сервер
15000	TCP	Envoy	Порт администрирования Envoy (команды/диагностика)
15001	TCP	Envoy	Envoy
15004	HTTP	Mixer, Pilot	Политики/телеметрия – mTLS
15010	HTTP	Pilot	Сервис Pilot – XDS-сервер – обнаружение сервисов
15011	TCP	Pilot	Сервис Pilot – mTLS – прокси – обнаружение сервисов
15014	HTTP	Citadel, Mixer, Pilot	Мониторинг уровня контроля
15030	TCP	Prometheus	Prometheus
15090	HTTP	Mixer	Прокси
42422	TCP	Mixer	Телеметрия – Prometheus

Запустив приложение и открыв к нему доступ, можно начать манипуляции сервисным трафиком. Доступность приложения – самый наглядный способ подтвердить возможность влияния на него, но использование сетки не является обязательным условием для манипулирования его трафиком.

Деинсталляция Istio

Удаление пространства имен `istio-system` *не* приведет к удалению Istio. Распространенная ошибка – думать, что это так, но удаление istio-system лишь удаляет компоненты уровня управления, оставляя CRD, прокси и другие артефакты в вашем кластере. Логика заключается в том, что CRD содержат конфигурацию времени выполнения, заданную администраторами. Из этого следует, что администраторам лучше удалять конфигурации времени выполнения, чем терять контроль над отдельными компонентами сетки. В данном случае мы не связаны с нашей инсталляцией, и поэтому для удаления Istio достаточно выполнить следующую команду в каталоге установки:

```
$ kubectl delete -f install/kubernetes/istio-demo.yaml
```

Однако это не приведет к удалению всех определений CRD, настроек сетки и примера приложения. Чтобы удалить их, выполните следующие действия:

```
$ for i in install/kubernetes/helm/istio-init/files/crd*yaml;
    do kubectl delete -f $i; done
$ kubectl delete -f samples/bookinfo/platform/kube/bookinfo.yaml
$ kubectl delete -f samples/bookinfo/networking/bookinfo-gateway.yaml
```

Убедитесь, что Istio и Bookinfo были успешно удалены, выполнив команды:

```
$ kubectl get crds
$ kubectl get pods
```

Если CRD останутся после попытки деинсталляции, некоторые предложения по отладке см. в главе 11.

Установка с помощью Helm

Для упрощения процедуры развертывания Istio в производственном окружении обычно рекомендуется использовать диспетчеры пакетов, такие как Helm (без Tiller), или диспетчеры конфигурации, такие как Ansible. В настоящее время в проекте Istio ведутся работы по его адаптации для использования Helm Tiller в качестве утилиты для управления обновлениями.

Установка Helm

Чтобы воспользоваться диспетчером пакетов Helm, его нужно установить. Для этого достаточно поместить двоичный файл в PATH, или прибегнуть к помощи системного диспетчера пакетов.

1. Загрузите последнюю версию Helm (https://oreil.ly/61_6b) для вашей ОС.
2. Распакуйте загруженный файл.
3. Найдите Helm и переместите его в нужное место (например, */usr/local/bin/helm*).

Или можно добавить путь к распакованному каталогу в PATH, перейдя в этот каталог и выполнив команду:

```
export PATH=$PWD:$PATH
```

Для установки в локальную систему (MacOS) можно также использовать Homebrew, как показано в примере 4.16.

Пример 4.16 ❖ Установка Helm на MacOS с помощью Homebrew

```
$ brew install kubernetes-helm
```

Информацию по установке в другие системы см. в документации Helm (https://oreil.ly/UvQm2).

После установки Helm можно продолжить работу с Tiller или без Tiller. Tiller – это серверный компонент Helm, работающий в вашем кластере. Он напрямую взаимодействует с Kubernetes API Server для установки, обновления, получения и удаления ресурсов Kubernetes. Он также хранит объекты, представляющие релизы.

Установка с помощью Helm

Давайте рассмотрим развертывание сетки с помощью Helm (без Tiller). Создайте пространство имен для компонентов уровня управления Istio, а затем установите все определения CRD Istio. Добавьте в Kubernetes все основные компоненты Istio, перечисленные в манифесте *istio.yaml*, выполнив следующие команды в каталоге установки Istio:

```
$ kubectl create namespace istio-system
$ helm template install/kubernetes/helm/istio-init --name istio-init
    --namespace istio-system | kubectl apply -f -
$ helm template install/kubernetes/helm/istio --name istio
    --namespace istio-system | kubectl apply -f -
```

Одним из преимуществ развертывания с помощью Helm является простота изменения конфигурации Istio с использованием параметров установки `--set.<key>=<value>`, которые можно добавить в команду Helm. Вот так:

```
$ helm install install/kubernetes/helm/istio --name istio
    --namespace istio-system \
--set global.controlPlaneSecurityEnabled=true \
--set mixer.adapters.useAdapterCRDs=false \
--set grafana.enabled=true --set grafana.security.enabled=true \
--set tracing.enabled=true \
--set kiali.enabled=true
```

Параметры установки Istio (https://oreil.ly/UvQm2) помогают легко настроить развертывание Istio. При развертывании в производственном окружении обычно используется диспетчер пакетов или конфигураций, такой как Helm

или Ansible. Сетки Istio, первоначально развернутые с помощью этих инструментов, должны обновляться с использованием тех же инструментов.

Проверка сетки после установки

По аналогии с `kubectl`, Helm устанавливает Istio в собственное пространство имен Kubernetes, `istio-system`. Чтобы проверить, развернута ли система Istio, а также увидеть все развернутые части, выполните команду:

```
$ kubectl get svc -n istio-system
```

Проверьте также, запущены ли сервисы:

```
$ kubectl get pods -n istio-system
```

Деинсталляция с помощью Helm

Чтобы удалить сетку Istio, установленную с помощью Helm, выполните следующие команды:

```
$ helm template install/kubernetes/helm/istio --name istio
    --namespace istio-system | kubectl delete -f -
$ kubectl delete -f install/kubernetes/helm/istio-init/files
$ kubectl delete namespace istio-system
```

Другие окружения

Сервисную сетку можно развернуть не только в Kubernetes, но и на множестве других платформ. *Расширение сетки* – это термин, используемый сообществом Istio для описания переноса существующих сервисов в сервисную сетку. В этой книге мы не будем рассматривать все способы расширения сервисной сетки Istio, но отметим, что в число поддерживаемых платформ входят:

- физические и виртуальные машины;
- Apache Mesos и Cloud Foundry;
- Nomad и Consul;
- Eureka.

Кроме того, многие поставщики услуг облачных вычислений и управляемых сервисов предлагают поддержку Istio в своих окружениях.

Глава 5

Прокси для сервисов

Вероятно, вы знаете отличия между *прямым* и *обратным* прокси. Напомним, что прямые прокси управляют исходящим трафиком для повышения производительности и фильтрации запросов и обычно используются в качестве интерфейса между пользователями в локальных сетях и ресурсами в интернете. Прямые прокси обычно улучшают производительность, кешируя статический веб-контент, и обеспечивают некоторый уровень безопасности, предотвращая доступ пользователей к определенным категориям веб-сайтов. В большой организации может быть настроен прямой прокси между локальными машинами и интернетом, фильтрующий протоколы и веб-сайты в соответствии с политикой использования сети данной организации.

Обратные прокси, напротив, управляют входящим трафиком, поступающим из интернета в локальные сети. Они обычно используются для фильтрации HTTP-запросов, защиты и балансировки нагрузки на серверы. То есть прямые прокси служат олицетворением пользователей для внешних серверов, а обратные прокси – олицетворением реальных серверов для внешних пользователей (клиентов).

Как показано на рис. 5.1, обратные прокси олицетворяют реальные серверы. В зависимости от типа и конфигурации, клиент практически не видит разницы между обратным прокси и сервисом, которому он посылает запрос. Обратные прокси пересылают запросы одному или нескольким реальным серверам, обрабатывающим запросы. Ответ прокси выглядит так, будто он пришел непосредственно с реального сервера, и клиент даже не подозревает, что взаимодействует с обратным прокси, а не с реальным сервером.

Рис. 5.1 ❖ Обратные прокси выступают в качестве промежуточного звена между клиентом и сервером

Эта концепция ничем не отличается от концепции разработки высоконадежных трехуровневых приложений, когда для каждого уровня требуется обеспечить непрямой доступ, высокую доступность и балансировку нагрузки. Обычно эти уровни можно масштабировать по вертикали, а прокси-серверы повышают устойчивость приложений, поскольку они встраиваются во взаи-

модействия между клиентами и серверами и реализуют дополнительные возможности, такие как балансировка нагрузки, как показано на рис. 5.2.

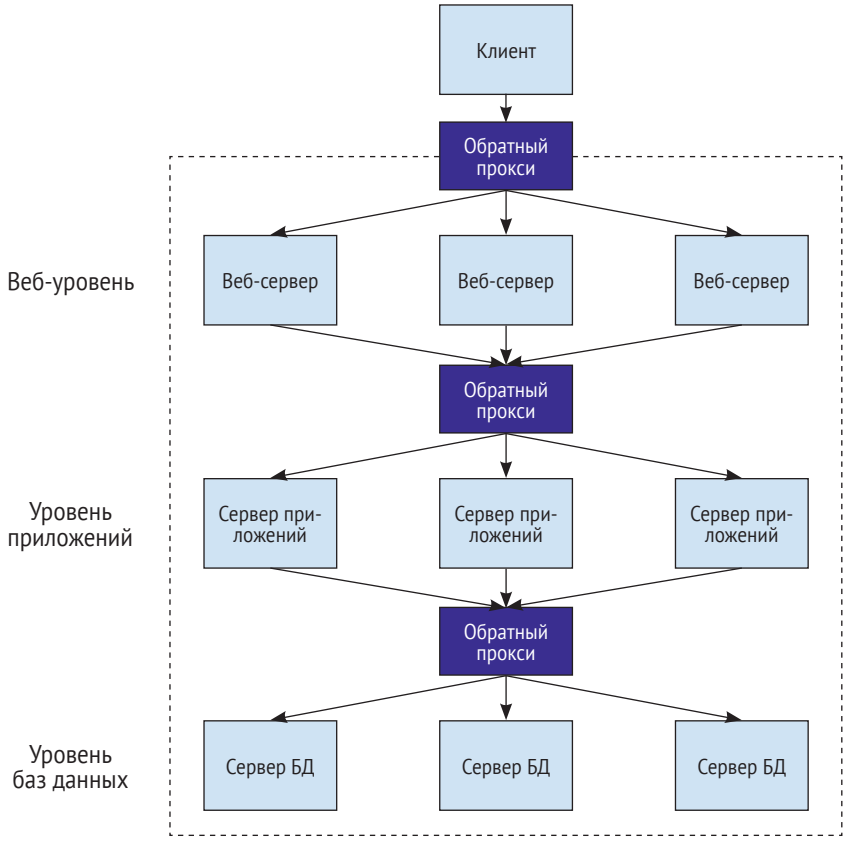

Рис. 5.2 ❖ Трехуровневое приложение с обратными прокси-серверами / межуровневыми балансировщиками нагрузки

Прокси играют роль представления сервиса и контролируют доступ к нему, вводя дополнительный уровень косвенности.

Что такое прокси для сервисов?

Подобно обратным прокси, *прокси для сервиса* – это транзитный клиент-посредник, передающий запросы от имени сервиса. Прокси сервиса позволяет приложениям отправлять и принимать сообщения по каналу как вызовы методов. Подключения через прокси могут создаваться по мере необходимости или использоваться для поддержания открытых соединений. Прокси добавляются в сервисы прозрачно, и поскольку приложения совершают вызовы «сервис–сервис», они не знают о существовании уровня данных. Уровни данных отвечают за внутрикластерную связь, а также за входящий и исходящий сетевой

трафик кластера. Входит ли трафик в сеть или выходит из сети, трафик сервиса сначала направляется на обработку в прокси. В случае Istio трафик прозрачно перехватывается правилами iptables и перенаправляется в прокси.

Коротко о iptables

iptables – это инструмент командной строки для управления файерволом на стороне сервера и манипулирования пакетами в Linux. Netfilter – это модуль ядра Linux, содержащий таблицы, цепочки и правила. Обычно среда iptables содержит несколько таблиц: *Filter*, *NAT*, *Mangle* и *Raw*. Вы можете определить свои собственные таблицы; в противном случае таблица *Filter* используется по умолчанию, как показано на рис. 5.3.

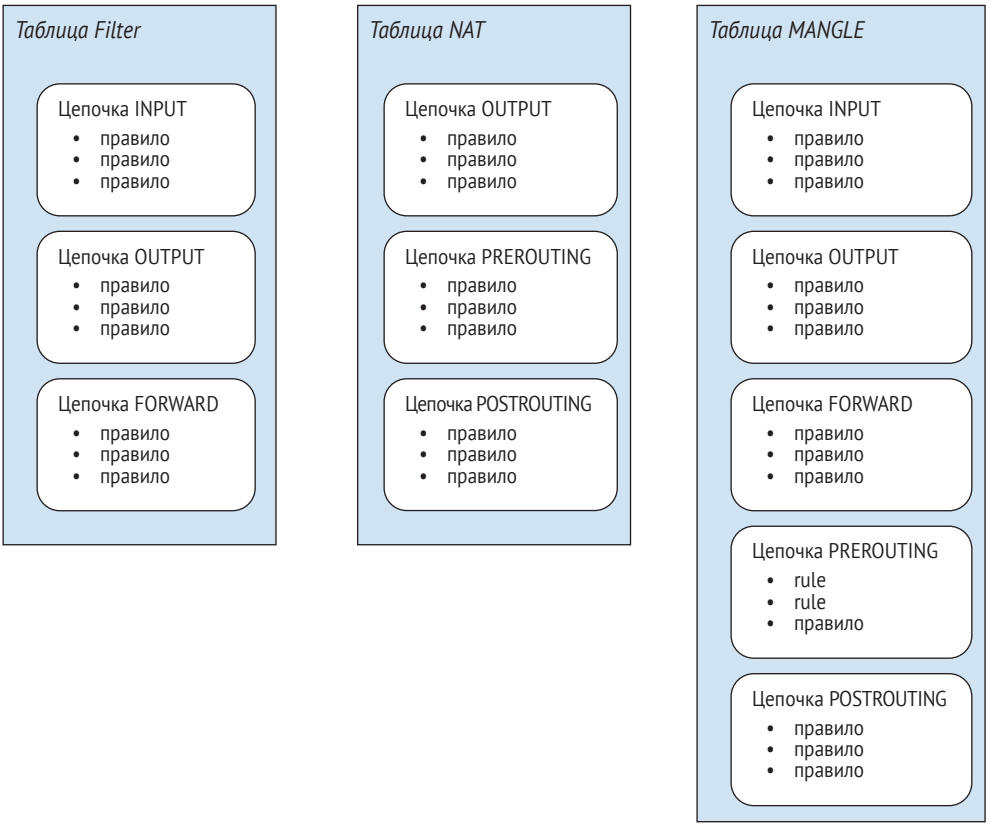

Рис. 5.3 ❖ Таблицы, цепочки и правила в iptables

Таблицы могут содержать несколько цепочек. Цепочки могут быть встроенными или определяться пользователем, а также содержать ряд правил. Правила выбирают пакеты, соответствующие критериям, и выполняют некоторые операции с ними. С помощью команды `iptables` можно просмотреть цепочки, которые использует Istio (https://oreil.ly/jA55x) для перенаправления трафика

в прокси Envoy. Цепочки iptables работают в сетевом пространстве имен, поэтому изменения, сделанные внутри пода, не влияют на другие поды или узел, на котором работает под.

Цепочки iptables, создаваемые Istio, можно исследовать и даже изменять. Эти цепочки можно увидеть в действии и проверить недостающие возможности NET_ADMIN вашего приложения и контейнеров-прицепов при выполнении exec в одном из контейнеров пода, как показано в примере 5.1.

Пример 5.1 ❖ Пример вывода команды iptables в контейнере с подключенным прокси Envoy

```
# iptables -t nat --list
Chain ISTIO_REDIRECT (2 references)
target     prot opt source               destination
REDIRECT   tcp  --  anywhere             anywhere             redir ports 15001
```

Напомним, что политика управления трафиком настраивается в компоненте Pilot и реализуется прокси. Коллекция прокси называется уровнем данных. Прокси перехватывают каждый пакет в запросе и отвечают за проверку состояния, маршрутизацию, балансировку нагрузки, аутентификацию, авторизацию и генерацию наблюдаемых сигналов. Прокси создают дополнительный уровень косвенности, чтобы клиенты могли обращаться по одному и тому же адресу (например, proxy.example.com), а сервис мог бы переноситься с одного узла на другой; таким образом, прокси предоставляют постоянные ссылки, повышая отказоустойчивость распределенных систем.

Обзор Envoy Proxy

В соответствии со своим девизом универсального API уровня данных многоцелевой и эффективный Envoy (https://envoyproxy.io/) появился как прокси уровня приложений с открытым кодом. Envoy был разработан в Lyft, когда было необходимо решить проблему крупных распределенных систем. Envoy получил широкую известность и часто используется в облачных окружениях. Страница сообщества проекта (https://oreil.ly/-g_eV) освещает наиболее значимые области его применения.

Лингвистика сервисной сетки

Безусловное лидерство в облачной экосистеме принадлежит языку программирования Go, поэтому большинство сервисных сеток написаны именно на нем. По характеру решаемых задач уровни данных должны максимально эффективно перехватывать, анализировать и пересылать сетевой трафик. Хотя Go обеспечивает высокую производительность, никто не будет отрицать, что машинный код – это священный Грааль производительности.

Как компонент уровня данных Envoy написан на языке C++ (точнее на C++11), отчасти по соображениям производительности. Также применение в сервисных сетках нашел Rust – новый и перспективный язык (способный конкурировать даже с C++). Благодаря эффективности (выше, чем у Go) и безопасному управлению памятью без сборки мусора, на Rust был написан компонент уровня данных Linkerd2 и модуль Mixer в nginMesh.

Если вы интересуетесь использованием языков, тогда обзор ландшафта сервисных сеток (https://oreil.ly/57P0j) покажется вам весьма любопытным. Дополнительные сведения по этому вопросу можно также найти в статье *The Enterprise Path to Service Mesh Architectures* (https://oreil.ly/2Jw6u).

Почему Envoy?

Почему бы не использовать NGINX, широко известный и проверенный прокси-сервер? Или Linkerd v1, Conduit, HAProxy, или Traefik? В то время Envoy был малоизвестен и не выглядел очевидным кандидатом. Виртуальный прокси-сервер Linkerd v1 на базе Java Virtual Machine (JVM) с его уровнем потребления ресурсов хорошо подходит для развертывания агентов узлов, но не для развертывания в контейнерах (в Linkerd v2 эту проблему исправили и реализовали прокси для сервисов на Rust). Первоначально Envoy предназначался на роль пограничного прокси и не предусматривал возможности развертывания в контейнерах-прицепах. Но со временем компания Lyft реализовала поддержку шаблона «Прицеп» (Sidecar) в Envoy.

Помимо модели развертывания, главным аргументом в пользу использования в Istio прокси-сервера Envoy вместо NGINX (который первоначально рассматривался на эту роль) стала идея горячей *перезагрузки* вместо горячих *рестартов*. С самого начала управление настройками Envoy осуществлялось посредством его API, который мог *осушать* (*draining*) и замещать собственный процесс новым процессом с новой конфигурацией. Envoy выполняет горячую перезагрузку процессов, используя общую память и взаимодействие через Unix Domain Socket (UDS, https://oreil.ly/YpkZ8), – похожий подход используется в GitHub (https://oreil.ly/SNdNG) для перезагрузки HAProxy с нулевым временем простоя.

Кроме того, Envoy предлагает агрегированный сервис обнаружения (*Aggregate Discovery Service*, ADS) для доставки данных на каждый xDS API (подробнее об этих API позже).

HTTP/2 и gRPC

В свое время функция поддержки протоколов HTTP/2 и gRPC выделяла Envoy среди других прокси-серверов. HTTP/2 значительно лучше HTTP/1.1 в том, что HTTP/2 позволяет *мультиплексирование* запросов по одному TCP-соединению. Прокси-серверы, поддерживающие HTTP/2, могут значительно снизить потребление ресурсов за счет объединения множества отдельных соединений в одно. HTTP/2 позволяет клиентам отправлять несколько параллельных запросов и загружать ресурсы заранее по инициативе сервера.

Envoy совместим с HTTP/1.1 и HTTP/2 и способен передавать оба протокола как вверх, так и вниз. Это означает, что Envoy может принимать входящие соединения HTTP/2 и передавать их в вышестоящие кластеры HTTP/2, а также принимать соединения HTTP/1.1 и проксировать их в HTTP/2 (и наоборот).

gRPC – это протокол RPC, использующий буферы протоколов (*protobufs*) поверх HTTP/2. Envoy изначально поддерживает gRPC (по HTTP/2), а также позволяет соединить клиент HTTP/1.1 по gRPC. Более того, Envoy способен работать

как транскодер gRPC-JSON. Функциональность транскодера gRPC-JSON позволяет клиенту отправлять запросы HTTP/1.1 с полезной нагрузкой JSON в Envoy, который преобразует запрос в соответствующий вызов gRPC и затем переведет ответное сообщение обратно в JSON. Эти мощные возможности (и их трудно правильно реализовать в процессе разработки) отличают Envoy от других сервис-прокси.

Envoy в Istio

В качестве внепроцессного прокси Envoy прозрачно формирует базовую единицу сетки. Подобно прокси в других сервисных сетках, эта рабочая лошадка Istio развертывается в дополнительных контейнерах, прицепляемых к сервисам приложений, как показано на рис. 5.4.

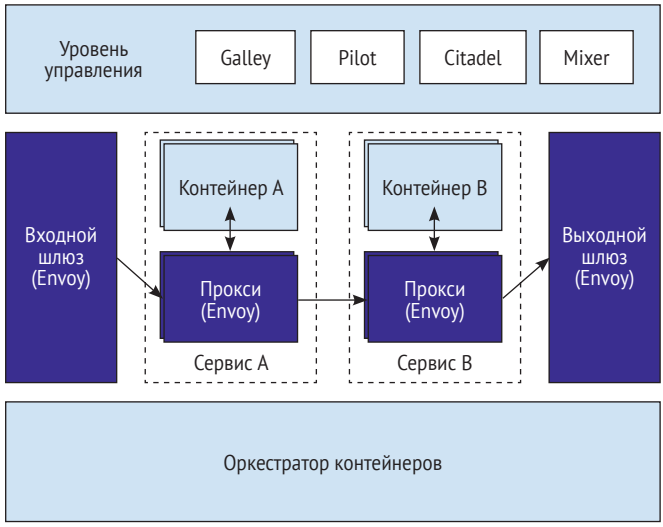

Рис. 5.4 ❖ Envoy в качестве прокси в сетке Istio

Обозначенный в файлах развертывания как `istio-proxy`, Envoy не требует привилегий root для запуска, но работает как пользователь 1337 (не root).

Внедрение в сетку

Добавление прокси в сервис выполняется в два этапа: *внедрение* и *захват сети*. На этапе внедрения, или «подцепления», прокси добавляется («подцепляется») к заданному приложению. На этапе захвата сети осуществляется перенаправление входящего трафика в прокси (вместо приложения) и исходящего трафика в прокси (вместо прямого возврата клиенту или последующим сервисам приложения).

Внедрение вручную

Можно использовать istioctl в качестве инструмента для внедрения вручную определений прокси Envoy в манифесты Kubernetes. Например, вот как с помощью команды istioctl kube-inject можно внедрить прокси в манифесты установки:

```
$ istioctl kube-inject -f samples/sleep/sleep.yaml | kubectl apply -f -
```

Изменять настройки Kubernetes можно даже на лету, применяя их к Kubernetes для планирования с использованием все той же утилиты istioctl kube-inject, например:

```
$ kubectl apply -f <(istioctl kube-inject -f <resource.yaml>)
```

Если у вас нет доступных исходных манифестов, можно обновить существующую установку Kubernetes, чтобы включить сервисы в сетку:

```
$ kubectl get deployment <deployment_name> -o yaml | istioctl kube-inject -f -
    | kubectl apply -f –
```

Рассмотрим пример добавления существующего приложения в сетку. Возьмем только что установленную копию Bookinfo, приложения для Istio, в качестве примера, уже работающего в Kubernetes, но развернутого не в сервисной сетке. Начнем с просмотра подов Bookinfo в примере 5.2.

Пример 5.2 ❖ Приложение Bookinfo, работающее вне сервисной сетки

```
$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
details-v1-69658dcf78-nghss       1/1     Running   0          43m
productpage-v1-6b6798cb84-nzfhd   1/1     Running   0          43m
ratings-v1-6f97d68b6-v6wj6        1/1     Running   0          43m
reviews-v1-7c98dcd6dc-b974c       1/1     Running   0          43m
reviews-v2-6677766d47-2qz2g       1/1     Running   0          43m
reviews-v3-79f9bcc54c-sjndp       1/1     Running   0          43m
```

Атомарной единицей развертывания в Kubernetes является объект pod (под) – группа, включающая один или несколько контейнеров, развертываемых вместе и атомарно. Как показано в примере 5.2, все поды в приложении Bookinfo сдержат по одному контейнеру. Когда команда istioctl kube-inject применяется к манифестам Bookinfo, она просто добавляет в определение ресурса Pod еще один контейнер, но фактически ничего не *развертывает*. istioctl kube-inject поддерживает модификацию объектов Kubernetes на базе Pod (Job, DaemonSet, ReplicaSet, Pod и Deployment), которые могут быть встроены в длинные YAML-файлы, содержащие другие объекты Kubernetes. Остальные объекты Kubernetes утилита istioctl kube-inject оставит без изменений. Неподдерживаемые ресурсы остаются неизменными, поэтому kube-inject можно без опаски применять к файлам, содержащим несколько Service, ConfigMap, Deployment и других определений для сложных приложений. Лучше всего это делать при первоначальном создании ресурса.

Можно взять YAML-файл, созданный командой kube-inject, и развернуть его напрямую. Для переноса существующего приложения можно выполнить

`istioctl kube-inject` для каждого `Deployment` и получить *последовательное обновление* `Deployment`, инициированное Kubernetes, как показано в примере 5.3. Давайте начнем с сервиса `productpage`.

Пример 5.3 ❖ Внедрение определения прокси в настройки сервиса productpage приложения Bookinfo

```
$ kubectl get deployment productpage-v1 -o yaml | istioctl
     kube-inject -f - | kubectl apply -f -
deployment.extensions/productpage-v1 configured
```

Если теперь вывести список подов Bookinfo, можно заметить, что под `productpage` содержит два контейнера. Внедрение прокси сервисной сетки Istio прошло успешно. Остальные сервисы приложения Bookinfo еще не включены в сетку, как показано в примере 5.4, и мы должны повторить ту же операцию с ними.

Пример 5.4 ❖ Сервис productpage приложения Bookinfo включен в сервисную сетку

```
$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
details-v1-69658dcf78-nghss       1/1     Running   0          45m
productpage-v1-64647d4c5f-z95dl   2/2     Running   0          64s
ratings-v1-6f97d68b6-v6wj6        1/1     Running   0          45m
reviews-v1-7c98dcd6dc-b974c       1/1     Running   0          45m
reviews-v2-6677766d47-2qz2g       1/1     Running   0          45m
reviews-v3-79f9bcc54c-sjndp       1/1     Running   0          45m
```

Чтобы не повторять снова и снова операции по включению работающего приложения в сервисную сетку, можно выполнить внедрение вручную один раз и сохранить новый файл манифеста с вставленным `istio-proxy` (Envoy). Для этого достаточно сохранить результаты работы `istioctl kube-inject` в файл:

```
$ istioctl kube-inject -f deployment.yaml -o deployment-injected.yaml
```

Либо так:

```
$ istioctl kube-inject -f deployment.yaml > deployment-injected.yaml
```

По мере развития Istio конфигурация прокси по умолчанию может изменяться (без объявления или объявление может быть скрыто в подробных примечаниях к релизу, где его легко пропустить).

istioctl kube-inject не является идемпотентной

Нельзя повторно применять операцию `istioctl kube-inject` к результатам предыдущей операции `kube-inject`. Операция `kube-inject` не является идемпотентной. Поэтому перед выполнением внедрения вручную сохраните исходный YAML-файл, чтобы в будущем можно было обновить прокси, составляющие уровень данных.

Параметры `--injectConfigFile` и `--injectConfigMapName` позволяют изменить шаблон с определением внедряемых прокси, встроенный в `istioctl`. Оба позволяют переопределить любые другие параметры конфигурации в шаблоне по умолчанию (например, `--hub` и `--tag`). Обычно эти параметры используются с файлами/картами конфигурации, распространяемыми с новой версией Istio:

```
# Создание хранимой версии развертывания с конфигурацией Envoy,
# внедренной из configmap Kubernetes 'istio-inject'
istioctl kube-inject -f deployment.yaml -o deployment-injected.yaml
     --injectConfigMapName istio-inject
```

Выборочное внедрение

Этап внедрения определения прокси подготавливает настройки для следующего этапа – захвата сети. Для постепенного внедрения Istio можно выборочно применять внедрение и захват сети. Вновь воспользуемся примером приложения Bookinfo и исключим его сервис `productpage` (только этот сервис из четырех) из сервисной сетки, оставив его в роли внешнего сервиса. Сначала убедимся, что у этого сервиса есть свой прокси:

```
$ kubectl get pods productpage-8459b4f9cf-tfblj
     -o jsonpath="{.spec.containers[*].image}"
layer5/istio-bookinfo-productpage:v1 docker.io/istio/proxyv2:1.0.5
```

Как видите, здесь имеется два контейнера: контейнер с сервисом `productpage` из нашего приложения и контейнер istio/proxy с прокси (Envoy), который Istio внедрила в под. Вручную включать сервисы в сервисную сетку и исключать из нее можно, управляя аннотацией в определении развертывания Kubernetes Deployment, как показано в примере 5.5.

Пример 5.5 ❖ Исключение определения развертывания сервиса из сетки

```
$ kubectl patch deployment nginx --type=json --patch='[{"op": "add", "path":
     "/spec/template/metadata/annotations", "value":
     {"sidecar.istio.io/inject": "false"}}]'
deployment.extensions/productpage-v1 patched
```

Откройте в браузере приложение `productpage`. Оно все еще обслуживается входным шлюзом Istio, но в его подах больше нет вспомогательных прокси. Таким образом, приложение `productpage` было удалено из сетки:

```
UNAVAILABLE:upstream connect error or disconnect/reset before headers
```

Автоматическое внедрение

Автоматическое внедрение прокси при введении сервисов в строй выглядит волшебством. Автоматическое внедрение избавляет от необходимости менять код и манифесты Kubernetes. В зависимости от конфигурации приложения могут потребоваться или не потребоваться изменения каких-либо аспектов приложения. Автоматическое внедрение прокси в Kubernetes основано на *изменяющих входных веб-обработчиках*. Когда Istio устанавливается в Kubernetes, она добавляет в конфигурацию изменяющий входной веб-обработчик `istio-sidecar-injector`, как показано в примерах 5.6 и 5.7.

Пример 5.6 ❖ Кластер Kubernetes с изменяющими веб-обработчиками Istio и Linkerd, зарегистрированными для механизма внедрения каждой сервисной сетки

```
$ kubectl get mutatingwebhookconfigurations
NAME                                  CREATED AT
```

```
istio-sidecar-injector                  2019-04-18T16:35:03Z
linkerd-proxy-injector-webhook-config  2019-04-18T16:48:49Z
```

Пример 5.7 ❖ Конфигурация изменяющего веб-обработчика istio-sidecar-injector

```
$ kubectl get mutatingwebhookconfigurations istio-sidecar-injector -o yaml
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration
metadata:
  creationTimestamp: "2019-04-18T16:35:03Z"
  generation: 2
  labels:
    app: sidecarInjectorWebhook
    chart: sidecarInjectorWebhook
    heritage: Tiller
    release: istio
  name: istio-sidecar-injector
  resourceVersion: "192908"
  selfLink: /apis/admissionregistration.k8s.io/v1beta1/
            mutatingwebhookconfigurations/istio-sidecar-injector
  uid: eaa85688-61f7-11e9-a968-00505698ee31
webhooks:
- admissionReviewVersions:
 - v1beta1
  clientConfig:
  caBundle: <redacted>
  service:
    name: istio-sidecar-injector
    namespace: istio-system
    path: /inject
failurePolicy: Fail
name: sidecar-injector.istio.io
namespaceSelector:
  matchLabels:
    istio-injection: enabled
rules:
apiGroups:
 - ""
  apiVersions:
 - v1
  operations:
 - CREATE
  resources:
 - pods
  scope: '*'
sideEffects: Unknown
timeoutSeconds: 30
```

После регистрации этого изменяющего веб-обработчика Kubernetes будет отправлять все события создания подов в сервис `istio-sidecar-injector` (в пространстве имен `istio-system`), если в этом пространстве имеется метка `istio-injection=enabled`. Этот сервис изменит PodSpec, добавив *два* дополнительных контейнера: `init-container` для настройки правил управления трафиком и `istio-proxy` (Envoy) с прокси для сервиса. Добавление дополнительных контей-

неров производится с помощью шаблона, находящегося в конфигурационной карте (*configmap*) `istio-sidecar-injector`.

Жизненный цикл Kubernetes позволяет настраивать ресурсы до их передачи в хранилище etcd – «источник истины», хранящий конфигурацию Kubernetes. Когда создается отдельный под (командой `kubectl` или при развертывании ресурса Deployment), он проходит эти этапы жизненного цикла и попадает в изменяющий входной веб-обработчик, модифицирующий его до применения.

Метки Kubernetes

Автоматическое внедрение прокси зависит от меток, определяющих, в какие поды будут внедряться прокси, образующие уровень данных в Istio. Объекты Kubernetes, такие как поды и пространства имен, могут иметь пользовательские метки. Метки, по сути, являются парами ключ/значение, как и в других системах, поддерживающих идею тегов. Контроллер изменяющих входных веб-обработчиков полагается на метки, выбирая пространства имен для применения. Istio использует специальную метку `istio-injection`. Познакомиться с автоматическим внедрением прокси можно, снабдив пространство имен `default` меткой `istio-injection=enabled`:

```
$ kubectl label namespace default istio-injection=enabled
```

В примере 5.8 показано, какие пространства имен имеют метку `istio-injection`.

Пример 5.8 ❖ Какие пространства имен Kubernetes имеют метку istio-injection?

```
$ kubectl get namespace -L istio-injection
NAME           STATUS    AGE       ISTIO-INJECTION
default        Active    1h        enabled
Docker         Active    1h        enabled
istio-system   Active    1h        disabled
kube-public    Active    1h
kube-system    Active    1h
```

Обратите внимание, что только пространство имен `istio-system` имеет метку `istio-injection`. Так как пространство имен istio-system имеет метку `istio-injection` со значением `disabled`, в его поды с сервисами не будут автоматически внедряться прокси. Однако из этого не следует, что поды в данном пространстве имен не могут иметь прокси; это просто означает, что прокси не будут внедряться автоматически.

Одно замечание: при использовании `namespaceSelector` убедитесь, что выбранные пространства имен действительно имеют используемую метку. Имейте в виду, что встроенные пространства имен, такие как `default` и `kube-system`, не имеют меток по умолчанию. И наоборот, пространство имен в разделе метаданных – это фактическое название пространства имен, а не метка:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
...
```

Init-контейнеры в Kubernetes

Подобно `cloud-init`, *init-контейнеры в Kubernetes* (https://oreil.ly/vopoy) позволяют использовать временные контейнеры для выполнения задачи до подключения основного контейнера (контейнеров). Init-контейнеры часто используются для выполнения задач инициализации ресурсов, таких как объединение активов, миграция баз данных или клонирование Git-репозитория в том. В случае Istio init-контейнеры используются для настройки сетевых фильтров – iptables, – управляющих трафиком.

Выделение ресурсов для прокси

Istio v1.1 определяет лимиты ресурсов по умолчанию для своих прокси. Определение лимитов необходимо для автоматического масштабирования прокси. Заглянув в YAML-определение контейнера с прокси, можно заметить, что том монтируется независимо от того, используется mTLS или нет, как показано в примере 5.9.

Пример 5.9 ❖Фрагмент описания прокси в определении пода Kubernetes

```
...
  --controlPlaneAuthPolicy
   MUTUAL_TLS
...
  Mounts:
    /etc/certs/ from istio-certs (ro)
    /etc/istio/proxy from istio-envoy (rw)
...
```

Функциональные возможности Envoy

Как и другие прокси, Envoy использует сетевые приемники для приема входящего трафика. Термины *восходящий* (*upstream*) и *нисходящий* (*downstream*) описывают направление цепочки связанных между собой запросов на обслуживание (см. рис. 5.5). Куда ведет каждое из этих направлений?

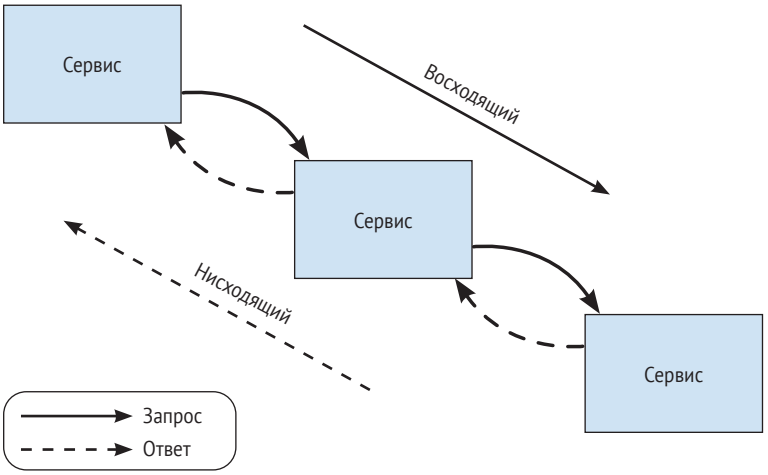

Рис. 5.5 ❖ Клиенты находятся ниже серверов; серверы выше клиентов

Нижестоящий
Нижестоящий сервис инициирует запросы и получает ответы.

Вышестоящий
Вышестоящий сервис получает запросы и возвращает ответы.

Основные конструкции

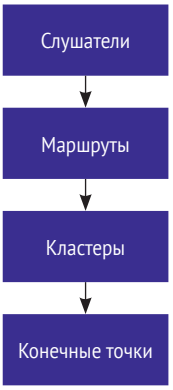

Рис. 5.6 ❖ Взаимоотношения между основными структурами Envoy

Приемник (*слушатель*) – это именованное сетевое местоположение (например, порт, сокет домена Unix и т. д.), принимающее соединения от нижестоящих клиентов. Envoy выставляет один или несколько приемников, что во многих случаях является открытым наружу портом, с которым внешние клиенты могут установить соединение. Приемник привязывается к определенному порту; физические приемники привязываются просто к порту. Виртуальные приемники используются для пересылки. Приемники также могут быть опционально настроены с помощью цепочки принимающих фильтров, каждый из которых может быть использован для работы с метаданными соединения или для лучшей системной интеграции без необходимости внесения изменений в ядро.

Приемники, маршруты, кластеры и конечные точки можно настраивать статически, используя файлы конфигурации, или динамически через соответствующие API: сервис обнаружения приемников (*listener discovery service*, LDS), сервис обнаружения маршрутов (*route discovery service*, RDS), сервис обнаружения кластеров (*cluster discovery service*, CDS) и сервис обнаружения конечных то-

чек (*endpoint discovery service*, EDS). Статические конфигурационные файлы могут определяться в формате JSON или YAML. Собирательно набор обнаружения сервисов для API Envoy называется xDS. Конфигурационный файл определяет объекты `listener`, `route`, `cluster` и `endpoint`, а также специфические настройки сервера, такие как включение Admin API, куда должны отправляться журналы доступа, и конфигурацию механизма трассировки.

Важно отметить, что в справочной документации Envoy (https://www.envoyproxy.io/docs) проводится четкое различие между документацией v1 и v2.

Существуют различные варианты конфигурации Envoy. Первоначальный вариант (v1) был отозван в пользу варианта v2. API v1 Envoy и его интеграция с Istio потребовали, чтобы Envoy для получения обновлений конфигурации опрашивал Pilot. В версии v2 Envoy открывает долгосрочное потоковое gRPC-соединение с Pilot, по которому Pilot может передавать последние обновления, «как есть», открытым текстом. Envoy поддерживал некоторую обратную совместимость с API конфигурации v1; однако, учитывая, что в какой-то момент она будет удалена, будет лучше сразу сосредоточиться на v2.

Istio Pilot использует ADS Envoy для динамической конфигурации и централизованно хранит таблицы маршрутизации и определения приемников и кластеров. Pilot может применять одни и те же правила к нескольким прокси, что упрощает рассылку обновлений конфигурации прокси в кластере. Во время работы Pilot использует эти API для рассылки конфигурации. Pilot эффективно рассчитывает конфигурацию для каждого сервиса. По умолчанию Pilot рассылает конфигурацию каждые 10 с; этот период определяется параметром `PILOT_DEBOUNCE_MAX`.

Раньше конфигурации рассылались по запросам, но теперь новые API Envoy используют принудительную рассылку, которая лучше масштабируется и, что особенно важно, позволяет передавать конфигурацию в Envoy в определенном порядке. Используя gRPC, Envoy устанавливает долговременную связь с Pilot. Pilot рассылает данные по мере вычисления изменений. Система AggregateADS Envoy гарантирует порядок доставки, позволяя последовательно обновлять прокси в сервисах. Это одно из ключевых свойств, обеспечивающих живучесть сервисной сетки.

Сертификаты и защита трафика

Итак, каковы установки безопасности по умолчанию? По умолчанию Pilot запрещает исходящий трафик к неопределенным конечным точкам (хотя недавно появилась возможность изменить эту настройку). Стандартная конфигурация безопасности Istio требует, чтобы по умолчанию Pilot был проинформирован, на какие конечные точки вне кластера можно отправлять трафик. Как только Pilot узнает об изменении топологии окружающей среды, ему необходимо перенастроить каждый затронутый прокси уровня данных.

В зависимости от типа вносимых в конфигурацию изменений может потребоваться или не потребоваться закрыть приемники Envoy (связь может быть либо прервана, либо нет). Примером намеренного закрытия соединений в Istio является ротация учетной записи (сертификата) сервиса. Хотя это и не требу-

ется, Istio прерывает соединения при перезагрузке сертификата сервиса. Secret Discovery Service (SDS) Envoy обеспечивает механизм передачи секретов (сертификатов) каждому прокси. В главе 6 содержится более подробная информация о SDS.

Перезапуск Envoy при ротации сертификатов (1 раз в час) осуществляет `pilot-agent` (показан в примерах 5.10 и 5.11). Хотя существующее открытое соединение будет повторно использовать просроченный сертификат, Istio намеренно закроет соединение.

Пример 5.10 ❖ istio-proxy – это контейнер, в котором выполняется два процесса: pilot-agent и Envoy

```
$ kubectl exec ratings-v1-7665579b75-2qcsb -c istio-proxy ps
  PID TTY          TIME CMD
    1 ?        00:00:10 pilot-agent
   18 ?        00:00:32 envoy
   70 ?        00:00:00 ps
```

Пример 5.11 ❖ Проверка действительности сертификата productpage

```
$ kubectl exec -it $(kubectl get pod | grep productpage | awk '{ print $1 }') -c
     istio-proxy -- cat /etc/certs/cert-chain.pem |
     openssl x509 -text –noout
```

Такое поведение Envoy при работе с соединениями можно изменить, и можно изучить его конфигурацию, как показано в примере 5.12.

Пример 5.12 ❖ Как узнать имя файла конфигурации Envoy в контейнере istio-proxy

```
$ kubectl exec ratings-v1-7665579b75-2qcsb -c istio-proxy ls /etc/istio/proxy
Envoy-rev0.json
```

Между собой прокси сервисов устанавливают соединения mTLS, используя для этого сертификаты. Сервисные сетки со вспомогательными прокси, такие как Istio, обычно создают локальные нешифрованные TCP-соединения между сервисами и прокси, действующими в одном поде. Это означает, что сервис (контейнер приложения) и Envoy используют для связи локальный сетевой интерфейс (loopback). Из этого следует, что сетевая политика Kubernetes и переадресация из прокси в приложение совместимы (и пересекаются). Кроме того, применение сетевой политики Kubernetes к взаимодействиям между приложением и прокси невозможно. Сетевой трафик приложения столкнется с сетевой политикой Kubernetes, только покинув пределы пода.

Консоль администрирования

Envoy предоставляет возможность просматривать конфигурацию, статистику, журналы и другие внутренние данные Envoy. Чтобы получить доступ к административной консоли прокси в уровне данных развернутой сетки Istio, следуйте инструкциям из главы 11. Если с административной консолью Envoy вне развернутой сетки, используйте Docker, как показано в примере 5.13.

Пример 5.13 ❖ Использование Envoy в контейнере Docker

```
$ docker run --name=proxy -d \
  -p 80:10000 \
  -v $(pwd)/envoy/envoy.yaml:/etc/envoy/envoy.yaml \
  envoyproxy/envoy:latest
```

Открыв в браузере адрес: http://localhost:15000, вы увидите список конечных точек:

`/certs`
: Сертификаты данного экземпляра Envoy.

`/clusters`
: Кластеры, для работы с которыми настроен Envoy.

`/config_dump`
: Фактическая конфигурация Envoy.

`/listeners`
: Настроенные приемники в Envoy.

`/logging`
: Просмотр и изменение настроек ведения журнала.

`/stats`
: Статистика Envoy.

`/stats/prometheus`
: Статистика Envoy в формате записей Prometheus.

В списке сертификатов, которые использует прокси в поде `productpage`, вы найдете три файла (см. пример 5.14). Один из них содержит приватный ключ сервиса `productpage` (`key.pem`).

Пример 5.14 ❖ Проверка правильности ключа и сертификата в прокси productpage

```
$ kubectl exec -it $(kubectl get pod | grep productpage | awk '{ print $1 }')
     -c istio-proxy -- ls /etc/certs
cert-chain.pem key.pem root-cert.pem
```

Теоретически Envoy можно использовать для управления состоянием окружения, однако за настройку прокси, конфигурацию Istio и хранение данных о сервисах обнаружения отвечает Pilot.

Глава 6

Безопасность и идентичность

Безопасность приложений и систем на протяжении длительного времени сконцентрирована на сети. Исторически для защиты от атак создавались жесткие внешние оболочки (брандмауэры, VPN и т. д.), однако после проникновения внутрь периметра взломщик может легко получить доступ ко многим системам. Но мы придумали идею глубокоэшелонированной защиты и применили приемы сетевой изоляции внутри своего доверительного домена, требуя от администраторов безопасности открывать узкие проходы в сети, достаточные для беспрепятственных взаимодействий наших приложений, и не более. Такой подход к безопасности работает хорошо при малой скорости изменения системы. Когда изменения происходят в пределах нескольких дней, легче выполнять действия вручную или автоматизировать настройку и обслуживание сетей.

Однако в случае систем на базе контейнеров темпы изменений измеряются не днями, а секундами. В высокодинамичных средах традиционные модели сетевой безопасности ломаются. Ключевая проблема заключается в том, что традиционная сетевая безопасность делает акцент на единственном доступном для сети идентификаторе – IP-адресе. IP-адрес не является сильным индикатором приложения, и поскольку динамические среды, такие как Kubernetes, легко могут повторно использовать IP-адрес для различных рабочих нагрузок, его недостаточно для использования в политике или безопасности.

Для решения этой проблемы в Istio реализована возможность присваивания идентификационных данных каждой рабочей нагрузке в сервисной сетке. Идентификационные данные привязаны к рабочей нагрузке, а не к конкретному хосту (или некоторой сетевой идентичности). Это означает, что можно определять политику взаимодействий сервис–сервис, устойчивую к изменениям в топологии и развертывании системы и не зависящую от конкретных особенностей сети.

В этой главе мы рассмотрим понятия *идентификации*, *авторизации* и *аутентификации* применительно к взаимодействиям сервис–сервис. Также разберем, как Istio обеспечивает идентификацию рабочих нагрузок, выполняет аутентификацию этих данных в режиме реального времени и использует их для авторизации взаимодействий между сервисами. Давайте начнем с контроля доступа.

Контроль доступа

Системы *контроля доступа* (*access control*) отвечают на следующий фундаментальный вопрос: «*Может* ли сущность *совершать* действия *над* объектом?» Эту сущность называют *субъектом*. *Действие* – это некоторая операция, определенная системой. *Объект* – это то, на что воздействует субъект. Например, в файловой системе Unix субъект пользователя может совершить *действия* «чтение», «запись» или «выполнение» над *объектом* файл.

Аутентификация

Главная цель аутентификации – субъект. Аутентификация – это процесс получения некоторого удостоверения (например, сертификата), проверки действительности удостоверения (т. е. подлинности удостоверения) и что *сущность*, представившая удостоверение, действительно является той сущностью, за которую себя выдает. Главная цель авторизации – определить, какие *действия* может или не может совершать *сущность* над *объектом*.

Аутентификация— это акт получения удостоверения из запроса и проверки его аутентичности. В Istio роль удостоверения, используемого сервисами при взаимодействии друг с другом, играет *сертификат X.509*. Прокси аутентифицируют вызывающую сторону (а вызывающая сторона – отвечающую) проверкой сертификата X.509, предоставленного другой стороной, используя обычный процесс проверки сертификата. Если сертификат действителен, то идентичность, закодированная в сертификате, считается аутентифицированной. После выполнения аутентификации мы говорим, что субъект *аутентифицирован*, называя его *аутентифицированным субъектом*, чтобы отличить его от *субъекта*, который может быть неаутентифицирован.

Авторизация

Авторизация – это акт ответа на вопрос «*Имеет ли право данная* сущность *выполнить* действие *над* объектом?». Например, прежде чем запустить сценарий, система Unix проверяет – имеет право текущий пользователь (*аутентифицированный субъект*) выполнить его. В Istio авторизация сервисов настраивается с помощью политик RBAC, о которых мы подробно расскажем далее в этой главе.

Когда мы размышляем над вопросом контроля доступа: «*Может* ли сущность *совершать* действия *над* объектом?», становится ясно, что необходимы обе процедуры – и аутентификация, и авторизация – и одна без другой бесполезна. Если аутентифицировать только удостоверение, любой пользователь сможет выполнить любое действие с любым объектом. Аутентификация лишь подтверждает, что данный пользователь на самом деле тот, кем он представился! Аналогично, если авторизовать только запросы, любой пользователь сможет притвориться другим и выполнить действия с его объектами; авторизация лишь проверяет, что *у кого-то* есть разрешение на выполнение данного действия, необязательно у вызывающего пользователя. И последнее: хочется

отметить, что ответы на вопрос авторизации Istio несколько конкретнее, чем «*Может* ли сущность *совершать* действия *над* объектом?». Говоря точнее, Istio отвечает на вопрос: «Может ли *сервис А* выполнить действие с *сервисом Б*?» Другими словами, как *сущность*, так и *объект* являются идентификаторами сервисов в сетке.

После знакомства с понятиями аутентификации и авторизации, естественно, возникают следующие вопросы: какие идентификационные данные Istio назначает сервисам? как сервисная сетка управляет этими идентификационными данными во время работы? как определить политику, разрешающую одному сервису выполнять действия в отношении другого, и как Istio осуществляет эти политики во время работы? В оставшейся части этой главы даны ответы на каждый из вопросов.

Идентичность

Сервисные сетки охватывают несколько кластеров – сервисы в сетке и вне ее способны взаимодействовать друг с другом. Где начинается сервисная сетка и где заканчивается? Что назвать границей сервисной сетки? Обычно ответы ограничиваются концепцией *административного домена*, при этом административным доменом являются либо все, что настраивается одним оператором сервисов, либо все, что может взаимодействовать друг с другом в рамках одной и той же сетки. Оба ответа популярны. На наш взгляд, подобные ответы не соответствуют действительности. Например, несколько команд могут управлять различными сегментами сетки, и различным частям сетки может быть запрещено взаимодействовать друг с другом. Напротив, мы считаем, что лучшим ответом будет «сервисная сетка – это единый домен идентификации». Другими словами, это единое пространство имен, из которого каждому сервису в системе присваивается идентификатор.

Идентификация формирует границу сервисной сетки. Идентификация является фундаментальной функцией сервисной сетки в том смысле, что все коммуникации исходят из нее. Управление трафиком и телеметрические функции сервисной сетки зависят от понимания того, как идентифицировать сервисы. Без знания, что измеряется, метрики бесполезны.

SPIFFE

Для создания идентификационных данных Istio использует спецификацию *Secure Production Identity Framework for Everyone* (SPIFFE). Короче говоря, Istio создает сертификат X.509, содержащий в поле альтернативного имени субъекта (SAN) уникальный идентификатор ресурса (URI), описывающий сервис. Istio обращается к платформе за атрибутами идентичности. В Kubernetes Istio использует учетную запись сервиса в поде и встраивает ее в URI: `spiffe://ClusterName/ns/Namespace/sa/ServiceAccountName`.

В Kubernetes под (pod) будет использовать учетную запись по умолчанию для пространства имен, в котором он установлен, если поле `ServiceAccount` не задано в спецификации пода. Это означает, что все сервисы в одном пространстве имен будут иметь общую идентификацию, если не настроить отдельные учетные записи для каждого сервиса.

SPIFFE – это спецификация фреймворка, способного генерировать и выпускать идентификационные данные. SPIRE (среда выполнения SPIFFE) является эталонной реализацией сообщества SPIFFE, а Citadel (бывшая Istio Auth) является второй реализацией. Спецификация SPIFFE описывает три понятия:

- идентификаторы в форме URI, используемые сервисами для связи;
- стандарт преобразования этого идентификатора в криптографически проверяемый документ (Verifiable Identity Document SPIFFE, SVID);
- API для выпуска и извлечения SVID (Workload API).

SPIFFE требует, чтобы идентификатор сервиса был закодирован как URI, начинающийся со схемы `spiffe`, например: `spiffe://trust-domain/path`. Доверенный домен (trust-domain) – это корень доверия идентичности (например, организация, подразделение или команда). Доверенный домен – это поле полномочий в URI (в частности, часть записи, соответствующая имени хоста). Спецификация разрешает в разделе пути в URI указывать все, что угодно, – универсальный уникальный идентификатор (UUID), иерархию доверия или вообще ничего не указывать. В Kubernetes Istio кодирует учетную запись `ServiceAccount` сервиса, используя имя локального кластера в качестве домена доверия, и создает путь, используя имя `ServiceAccount` и пространство имен. Например, учетная запись `ServiceAccount` по умолчанию закодирована как `spiffe://cluster.local/ns/default/sa/default` (где «ns» – сокращение от «namespce» (пространство имен), а «sa» – сокращение от «service account» (учетная запись сервиса)).

SPIFFE также описывает, как преобразовать этот идентификатор в SVID X.509. Сертификат X.509 можно проверить, чтобы убедиться в идентичности предъявителя. Спецификация требует, чтобы идентификатор в форме URI передавался в поле SAN сертификата. Чтобы убедиться в подлинности SVID, необходимо выполнить три проверки:

1) обычную проверку X.509;
2) подтвердить, что сертификат *не* является сертификатом подписи. Сертификаты подписи не могут использоваться для идентификации в соответствии со спецификацией;
3) убедиться, что в сертификате присутствует ровно одно поле SAN со схемой SPIFFE.

SPIFFE определяет Workload API для выпуска и извлечения SVID, однако именно здесь Istio отклоняется от SPIFFE. Istio реализует предоставление сертификатов с помощью специального протокола, *CA Service*. Citadel Node Agent выпускает CSR через этот API в момент появления новой рабочей нагрузки; Citadel выполняет проверку запроса и возвращает SVID для рабочей нагрузки. И SPIFFE Workload API, и сервис CA преследуют одну и ту же цель (подтверждают некоторую информацию о рабочей нагрузке, чтобы получить для нее идентификатор).

Наконец, хотя спецификация SPIFFE не требует этого, SPIRE и Istio выпускают краткосрочные X.509 SVID – они истекают примерно через час после их выпуска. Это противоречит традиционному использованию сертификатов X.509, обычно используемых для инкапсуляции в HTTPS TLS и заканчивающихся через год или более после выпуска.

Преимущество краткосрочных сертификатов заключается в том, что атаки ограничиваются в рамках этого срока без необходимости отзыва сертификата (и облегчает отзыв, если вы на это решитесь). Предположим, что взломщик скомпрометировал рабочую нагрузку и украл SVID; он действует только в части доверенного домена в течение короткого времени. Если для выполнения атаки требуется продолжительный период времени, придется постоянно извлекать действительные идентификационные данные из рабочей нагрузки. Как только станет известно об атаке, можно будет использовать политику, запрещающую доступ к другим сервисам, прекратить перевыпуск сертификатов для этой сущности и даже поместить данный сертификат в список аннулированных. Поскольку сертификаты являются эфемерными, управление списком отзывов является простой процедурой – удаление просроченных сертификатов из списка отозванных сертификатов считается стандартной практикой. Список остается небольшим, так как срок действия сертификатов быстро истекает.

Однако такое использование недолговечных сертификатов имеет один большой недостаток: выпуск и ротация сертификатов для всего парка рабочих нагрузок в короткий промежуток времени – сложная задача. О том, как Istio решает эту проблему, мы поговорим в следующем разделе.

Архитектура управления ключами

Citadel, агенты узлов и Envoy – это три ключевых компонента архитектуры управления ключами. Все они участвуют в выпуске и ротации SVID в сетке Istio (см. также рис. 6.1):

Citadel
: Citadel выдает идентификационные данные рабочим нагрузкам в сетке, действуя в качестве CA, подписывая запросы сертификатов, формирующие X.509 SVID.

Агент узла
: Доверенный агент, развернутый на каждом узле и действующий как посредник между Citadel и прокси Envoy на узле.

Envoy (*прокси*)
: В процессе работы Envoy обращается к агенту узла, используя локальное соединение, чтобы получить удостоверение и передать его другим сторонам.

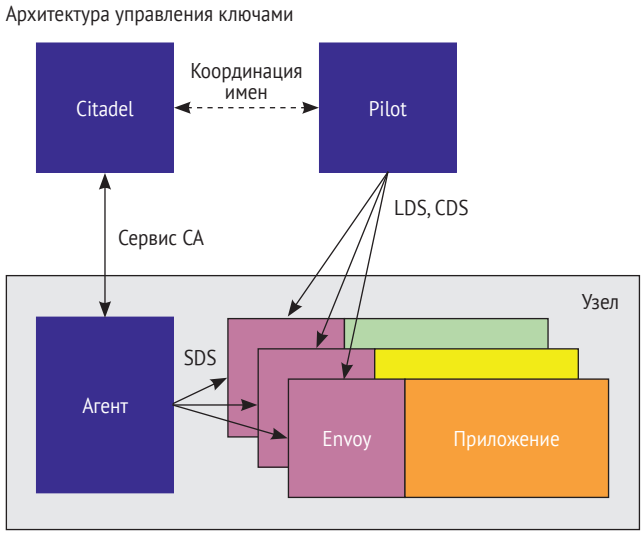

Рис. 6.1 ❖ Архитектура управления ключами в Istio и взаимодействие компонентов (см. главу 5 для получения дополнительной информации об Envoy xDS API)

Citadel

Citadel отвечает за прием запросов на идентификацию, их аутентификацию, авторизацию и в конечном счете выпуск сертификата для данной сущности. Сам компонент Citadel состоит из нескольких логических компонентов, как показано на рис. 6.2.

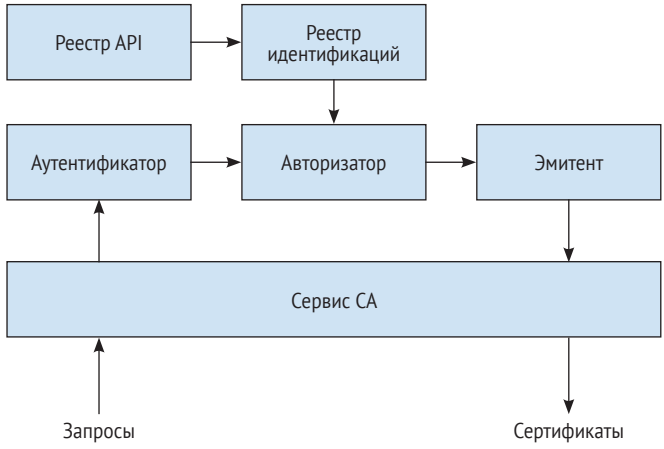

Рис. 6.2 ❖ Архитектура Citadel и взаимодействие внутренних компонентов

Пройдем по пути выпуска сертификатов сервисом CA, снизу вверх и слева направо, как показано на рис. 6.2:

1) Citadel открывает доступ к сервису CA через общедоступный API. Чтобы запросить идентификацию, абонент соединяется с сервисом CA и посылает запрос CSR на подпись сертификата в Citadel, который преобразуется в сертификат (X.509 SVID);
2) после получения запроса он передается *аутентификатору* (*authenticator*) для проверки. Метод аутентификации зависит от способа развертывания Citadel. Например, в Kubernetes компонент Pilot присваивает каждой рабочей нагрузке имя ее сервиса;
3) после аутентификации запроса *авторизатор* (*authorizer*) определяет, имеет ли право запрашивать удостоверение аутентифицированная сущность. Авторизатор консультируется с реестром идентификаций (*Identity Registry*), отображающим рабочие нагрузки через их аутентифицированные сущности в идентификационные данные;
4) как только приложение будет авторизовано на получение удостоверения, производится его выпуск методом подписания сертификата. Авторизатор запрашивает у *эмитента* (*issuer*) создание сертификата и делает его доступным для запрашивающей стороны. В настоящее время в Citadel в роли эмитентов могут использоваться центры сертификации (CA), действующие в оперативной памяти, и HashiCorp Vault (*Vault – open-source* инструмент компании HashiCorp для управления секретами).

Агенты узлов

Агенты узлов развернуты на каждом узле с приложениями, для которых Citadel будет выдавать удостоверения. На них возложены две обязанности: во-первых, они действуют как простой адаптер протокола между Envoy и Citadel. Envoy использует SDS API, конфигурирующий секреты, предоставляемые Envoy во время работы. Этот API отлично подходит для выпуска наборов ключей – самих сертификатов, но не поддерживает верификацию. Другими словами, Citadel не может использовать SDS API для проверки подлинности владельца удостоверения. Вместо этого для аутентификации запросов Citadel использует свой API сервиса CA, как мы описали в предыдущем разделе. Агенты узлов соединяют сервисы SDS и CA от имени приложений, развернутых на узлах. Это подводит ко второй обязанности агентов узлов: быть доверенными агентами на узлах, способными проверять среду приложений для Citadel от имени приложения и распределять ключи между приложениями, выполняющимися на данном узле.

Агент узла хранит в памяти секреты, полученные от Citadel, и когда срок их действия подходит к концу (например, осталось 25 % времени жизни), агент связывается с Citadel и пытается обновить сертификат. Это оставляет немного пространства для маневра на случай, если Citadel будет временно недоступен. Агент узла после остановки и перезапуска диспетчером контейнеров попытается получить новые учетные данные для всех приложений узла. В этом смысле агент узла не имеет хранимого состояния. После получения нового SVID от Citadel агент узла через SDS передает сертификат прокси-серверу Envoy. Как

описано в главе 5, это побуждает Envoy установить новые связи с целевыми приложениями. Envoy переключает существующие соединения на новые и прекращает использование старых (а теперь и потенциально истекших) сертификатов[1].

Расчет на агента узла, таким образом, оправдан в модели безопасности Istio, поскольку агент уже обрабатывает все секреты узла; именно поэтому он *должен* работать на более высоком уровне доверия, чем другие компоненты узла. Следовательно, разумным будет расширить доверие, чтобы, кроме прочего, заставить агента проверять среду выполнения. Даже если бы Envoy связывался с Citadel напрямую, нельзя было бы доверять ответам среды от Envoy, поскольку они относятся к той же области доверия, что и само приложение (поэтому злоумышленник, скомпрометировав приложение, легко сможет запутать Envoy, давая различные или неправильные ответы на вопросы аттестации среды).

Наконец, для поддержания масштабируемости Citadel применено архитектурное решение, обязывающее приложения взаимодействовать с локальным агентом узла, а агента узла – с Citadel. Такой дизайн ограничивает количество подключений к экземплярам Citadel в системе количеством узлов в сетке (обычно это число невелико), а не количеством рабочих нагрузок (как правило, их намного больше). Часто в таких средах, как Kubernetes, приложений значительно бóльше, чем узлов, поэтому такая конструкция дает реальные преимущества.

Envoy

Последний участник всего этого карнавала – прокси Envoy. Envoy настроен на связь с локальным агентом узла в качестве носителя SDS API. Координаты SDS-сервера Envoy может динамически получить от Pilot; однако сетка Istio обычно настраивает эту информацию статически, так как это менее подвержено ошибкам во время работы. Envoy может взаимодействовать с SDS-сервером (агентом узла), разрешая определенный адрес в адрес локального узла или, чаще всего, связываясь локально через UDS.

Envoy не будет перезаписывать статическую конфигурацию версией из API, поэтому вы не сможете случайно передать конфигурацию Envoy, делающую невозможной связь с конфигурационным сервером.

При инициализации подключений к другим сервисам в сетке Envoy использует SVID-сертификат. Взаимодействующие приложения в сетке могут при необходимости использовать протокол *mTLS*. Это гарантирует клиенту и серверу возможность аутентификации и авторизации другой стороны соединения

---

[1] Восстанавливать соединения с использованием новых учетных данных не нужно – TLS-сессия остается действительной, если она была инициирована до истечения срока действия сертификата. Установленное соединение будет успешно продолжать использовать просроченный сертификат. Мы решили восстановить связи для предотвращения некоторых видов атак, связанных с перехватом учетных данных. Плюс небольшие встряски, как правило, полезны для системы!

и обеспечивает шифрование при передаче. Однако одного mTLS недостаточно, поскольку для выполнения авторизации соединения между двумя сущностями нужно что-то еще. Далее в этой главе мы расскажем как о mTLS, так и об авторизации связи.

Pilot

Pilot также играет незначительную роль в управлении ключами. Когда Pilot передает конфигурацию в Envoy, включающую конфигурацию о сервисах назначения и способах приема трафика, он должен ссылаться на сертификаты. Pilot ссылается на эти сертификаты по именам; поэтому он должен координировать свою работу со SDS-службой, предоставляемой агентом узла. Было бы нежелательно заставлять агентов узлов общаться с Pilot в дополнение к Citadel. Поэтому все компоненты Istio заранее настроены на использование общей схемы именования секретов, чтобы Pilot мог однозначно ссылаться на секреты, обслуживаемые сервисом SDS. Кроме того, первичный сертификат во всех прокси Envoy, идентификатор SVID, хранится в хорошо известном месте (в */etc/certs/*, как описано в главе 5).

mTLS

Поскольку все идентификационные сертификаты (SVID) передаются всем приложениям в системе, как же использовать их для проверки идентичности серверов, с которыми мы взаимодействуем, и выполнения аутентификации и авторизации? Именно здесь вступает в игру mTLS. Но сначала немного предыстории.

Говоря о TLS или SSL (TLS – это новая версия SSL), обычно на ум приходит использование протокола HTTPS. Допустим, пользователь решил использовать свой браузер для подключения к какому-либо веб-серверу (например, http://wikipedia.org). Браузер (или ОД) выполняет DNS-запрос, чтобы определить IP-адрес сайта. Затем отправляет HTTPS-запрос на этот адрес и ждет ответа от сервера (сайта). Когда браузер (клиент) пытается подключиться к серверу, в ответ сервер возвращает сертификат с идентификатором (например, wikipedia.org), подписанный каким-либо корнем доверия, которому доверяет клиент. Клиент проверяет сертификат, идентифицирует сервер и позволяет установить соединение. Затем генерируется набор ключей для шифрования данных, отправляемых как клиентом, так и сервером. Другими словами, TLS – это протокол, позволяющий клиенту убедиться, что сервер действительно контролируется wikipedia.org и что никто не подслушивает и не подменяет данные, посылаемые сервером.

mTLS – это TLS, в котором обе стороны, клиент и сервер, предоставляют друг другу сертификаты. Это позволяет клиенту проверить идентичность сервера, как и обычный TLS, но также дает серверу возможность проверить идентичность клиента, пытающегося установить соединение. Мы используем mTLS в Istio, где обе стороны предоставляют друг другу свои SVID. Это позволяет обеим сторонам аутентифицировать SVID, предоставленный другой стороной,

и выполнять авторизацию соединения. В реальности же авторизация в Istio выполняется только на стороне сервера. Учитывая то, как написана политика авторизации, это имеет смысл; о ней мы поговорим в следующем разделе.

Настройка политик аутентификации и авторизации в Istio

Istio разделяет политику аутентификации и авторизации на два набора конфигураций. Первый, политика аутентификации, управляет взаимодействиями прокси в сетке друг с другом (независимо от того, требуется им SVID или нет). Второй, политика авторизации, требует сначала применить политику аутентификации и определяет, каким сущностям разрешено взаимодействовать.

Политика аутентификации: конфигурирование mTLS

Внедрение mTLS в существующую систему является сложной задачей, поскольку нужно убедиться, что и клиент, и сервер одновременно обеспечены сертификатами (традиционный протокол TLS намного проще, поскольку требует настройки только на стороне сервера). В результате Istio предоставляет несколько инструментов развертывания без использования mTLS и его постепенного включения без прерывания работы клиентов.

Политика аутентификации (`authentication.istio.io/v1alpha1.Policy`) является основным CRD, используемым для настройки взаимодействия сервисов в сетке. Политика аутентификации позволяет требовать, делать необязательными или отключать mTLS на основе подхода «сервис за сервисом», «пространство имен за пространством имен». Кластерный вариант, MeshPolicy, применяет политику по умолчанию для каждого пространства имен и каждого сервиса в сетке.

Чтобы включить mTLS для одного сервиса, достаточно создать политику в его пространстве имен и указать обязательное использование mTLS для целевого целевого сервиса, например:

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: foo-require-mtls
  namespace: default
spec:
  targets:
 - name: foo.default.svc.cluster.local
  peers:
 - mtls:
      mode: STRICT
```

Эта политика применяется к пространству имен `default` и требует обязательного использования TLS для общения с сервисом `foo`. Обратите внимание, что поскольку для mTLS по умолчанию настроен режим `STRICT`, можно немного упростить эту конфигурацию, опустив избыточные поля:

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: foo-require-mtls
  namespace: default
spec:
  targets:
 - name: foo.default.svc.cluster.local
  peers:
 - mtls: {}
```

Многие политики в istio.io определяются в этой форме, опуская объект mTLS, поскольку по умолчанию требуется режим `STRICT`.

Конечно, простое создание такой конфигурации в кластере может привести к сбоям, если у клиентов еще нет сертификатов для работы с mTLS. Поэтому Istio включает для mTLS режим `PERMISSIVE`, позволяющий клиентам подключаться *либо* открытым текстом, *либо* через mTLS. Следующая конфигурация позволяет клиентам связаться с сервисом `bar`, используя mTLS и открытый текст:

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: bar-optional-mtls
  namespace: default
spec:
  targets:
 - name: bar.default.svc.cluster.local
  peers:
 - mtls:
      mode: PERMISSIVE
```

Аналогичным образом можно сделать mTLS необязательным для всего пространства имен, опустив поле `targets`:

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: default-namespace-optional-mtls
  namespace: default
spec:
peers:
 - mtls:
    mode: PERMISSIVE
```

Эта конфигурация позволяет приложениям в сетке связываться с любым сервисом в пространстве имен default, используя mTLS или открытый текст. Также можно включить или отключить mTLS для каждого порта сервиса. Примером того, где выгодна политика на каждый порт, является проверка работоспособности, выполняемая с помощью kubelet в кластере Kubernetes. Предоставление отдельных сертификатов для mTLS-соединений с kubelet может быть слишком трудоемкой задачей. Описав два объекта `policy`, можно исключить порт проверки работоспособности из mTLS, требуя mTLS для всех остальных портов, и тем самым упростить интеграцию с существующими системами, например:

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: bar-require-mtls-no-port-81
  namespace: default
spec:
  targets:
 - name: bar.default.svc.cluster.local
peers:
 - mtls:
      mode: STRICT
---
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: bar-require-mtls-no-port-81
  namespace: default
spec:
  targets:
 - name: bar.default.svc.cluster.local
    port:
      name: http-healthcheck
   peers:
  - mtls:
       mode: PERMISSIVE
```

Используя такой же подход, администраторы могут исключить обязательность mTLS для подключения к порту `http-healthcheck` через пространство имен, опустив в поле targets список любых конкретных имен сервисов.

Чтобы применить одну и ту же конфигурацию ко всем пространствам имен, используется ресурс `MeshPolicy`. Он идентичен ресурсу `policy` в схеме, но существует на уровне кластера. Также обратите внимание, что стандартная политика `MeshPolicy` *должна* называться *default*, иначе Istio не распознает ее правильно.

```
apiVersion: authentication.istio.io/v1alpha1
kind: MeshPolicy
metadata:
  name: mesh-wide-optional-mtls
spec:
  peers:
 - mtls:
      mode: PERMISSIVE
```

И конечно же, можно сделать mTLS обязательным для всей сетки, установив режим `mode: STRICT` или опуская объект mTLS полностью:

```
apiVersion: authentication.istio.io/v1alpha1
kind: MeshPolicy
metadata:
  name: mesh-wide-mtls
spec:
  peers:
 - mtls: {}
```

Istio также поддерживает аутентификацию конечных пользователей через JSON Web Tokens (JWT). Политика аутентификации Istio поддерживает установку широкого набора ограничений на данные в JWT и позволяет проверять достоверность практически всех полей JWT. Следующая политика предписывает Envoy требовать mTLS; также она требует, чтобы учетные данные конечного пользователя, хранящиеся как JWT в заголовке «`x-goog-iap-jwt-assertion`», выпущенном Google (https://securetoken.google.com), проверялись по открытым ключам Google (https://www.googleapis.com/oauth2/v1/certs) для аудитории `bar`:

```
apiVersion: authentication.istio.io/v1alpha1
kind: Policy
metadata:
  name: end-user-auth
  namespace: default
spec:
  target:
 - name: bar
  peers:
 - mtls: {}
  origins:
 - jwt:
      issuer: "https://securetoken.google.com"
      audiences:
     - "bar"
      jwksUri: "https://www.googleapis.com/oauth2/v1/certs"
      jwt_headers:
     - "x-goog-iap-jwt-assertion"
  principalBinding: USE_ORIGIN
```

Политика авторизации: настройка разрешений

Наличие политики аутентификации позволяет в рамках всей системы использовать идентификационные данные для контроля того, какие сервисы могут взаимодействовать. Предположим, надо описать коммуникационную политику сервис–сервис. Политика авторизации в Istio описывается с использованием системы RBAC. Как и большинство систем RBAC, она определяет два объекта, совместно используемых для записи политики:

ServiceRole
: Описывает набор действий, которые могут быть выполнены любым субъектом, обладающим данной ролью, с набором сервисов.

ServiceRoleBinding
: Назначает роли субъектам. В этом контексте главными субъектами являются удостоверения, выпускаемые сеткой Istio. Напомним, что в кластере Kubernetes субъекты определяются через `ServiceAccount`.

Во-первых, нужно создать объект `ClusterRBACConfig` (бывший `RBACConfig` в версиях до v1.1), включающий RBAC в Istio:

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: RBACConfig
```

```
metadata:
  name: default
  namespace: istio-system
spec:
  mode: ON
```

Такая конфигурация разрешает RBAC для связи сервис–сервис по всей сетке. Подобно включению mTLS, это потенциально опасно делать в работающей системе, поэтому Istio поддерживает постепенное включение RBAC для связи сервис–сервис путем изменения режима `RBACConfig`. Istio поддерживает четыре режима:

`OFF`
: Для связи RBAC не требуется. Если объект `ClusterRBACConfig` не существует, то это поведение системы по умолчанию.

`ON`
: Политики RBAC обязательны для связи, а связь, не разрешенная политикой, запрещена.

`ON_WITH_INCLUSION`
: Политики RBAC обязательны для связи с любым сервисом из набора пространств имен, перечисленных в политике.

`ON_WITH_EXCLUSION`
: Политики RBAC необходимы для связи с любым сервисом в сетке, за исключением сервисов из набора пространств имен, перечисленных в этой политике.

Для постепенного развертывания RBAC в системе сначала включите его в режиме `ON_WITH_INCLUSION`. При определении политик для каждого сервиса или пространства имен добавьте этот сервис или пространство имен в список включений. Это позволяет включить RBAC сервис за сервисом (или пространство имен за пространством имен), как показано в примере 6.1.

Пример 6.1 ❖ Пошаговое внедрение политики RBAC

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ClusterRBACConfig
metadata:
  name: default
  namespace: istio-system
spec:
  mode: ON_WITH_INCLUSION
  inclusion:
    services:
   - bar.bar.svc.cluster.local
    namespaces:
   - default
```

Политика в примере 6.1 не требует использования RBAC для связи с любыми сервисами в пространстве имен default, кроме как с сервисом `bar`. В какой-то момент пространств имен и сервисов в системе с политикой RBAC окажется больше, чем без нее; в этот момент можно переключиться на политику `ON_WITH_EXCLUSION`.

После включения RBAC для сервиса `bar` нужно написать политики. Начнем с выбора пространства имен или сервиса и описания ролей для этого сервиса. Например, создадим ресурс `ServiceRole`, разрешающий доступ на чтение (запросы HTTP GET) к сервису `bar`:

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRole
metadata:
  name: bar-viewer
  namespace: default
spec:
rules:
- services:
 - bar.default.cluster.local
  methods:
 - GET
```

Затем определим ресурс `ServiceRoleBinding`, чтобы связать эту роль с учетной записью сервиса `bar`и позволить ей вызывать сервис `foo`:

```
apiVersion: "rbac.istio.io/v1alpha1"
kind: ServiceRoleBinding
metadata:
  name: bar-bar-viewer-binding
  namespace: default
spec:
  subjects:
 - properties:
      # SPIFFE ID аккаунта сервиса bar в пространстве имен bar
     source.principal: "cluster.local/ns/bar/sa/bar"
  roleRef:
    kind: ServiceRole
    name: "bar-viewer"
```

В отличие от RBAC в приложениях, разрешающих или запрещающих пользователям определенные операции, Istio RBAC ориентирован на взаимодействия сервис–сервис, определяя, какие сервисы могут связываться и взаимодействовать друг с другом. Для этого включите Citadel, систему управления ключами, чтобы обеспечить идентификацию каждого сервиса в сетке, и разрешите сервису самоаутентификацию.

Границу сетки формирует идентификация. Благодаря прокси в сетке Istio, передающим идентификационные данные и обрабатывающим весь трафик от/к сервисам, можно использовать взаимно доверенные сертификаты для защиты соединений и авторизации этих соединений. Istio способствует постепенному внедрению mTLS и RBAC между сервисами.

Глава 7

Pilot

Pilot отвечает за программирование уровня данных, входных и выходных шлюзов и прокси в сетке Istio. Pilot моделирует среду развертывания, комбинируя конфигурацию Istio от Galley и информацию о сервисах из реестра, такого как Kubernetes API Server или Consul. Pilot использует эту модель для создания конфигурации уровня данных и передает созданную конфигурацию в парк подключенных к нему прокси.

Настройка Pilot

Чтобы лучше понять все аспекты сетки, касающиеся Pilot, сделаем обзор его конфигурации. Отметим лишь, что по мере развития проекта Istio будет расти зависимость Pilot от Galley относительно нижележащей платформы и информации о среде. Pilot использует три основных источника конфигурации:

Конфигурация сетки
: Набор глобальных конфигураций для сервисной сетки.

Конфигурация сети
: Конфигурации `ServiceEntrie`, `DestinationRule`, `VirtualService`, `Gateway` и прокси.

Механизм обнаружения сервисов
: Местоположение и метаданные из реестров о каталоге сервисов, размещенных на одной или нескольких нижележащих платформах.

Конфигурация сетки

Конфигурация сетки – это статический набор глобальных конфигураций для ее установки. Конфигурация сетки распределена по трем объектам:

`MeshConfig (mesh.istio.io/v1alpha1.MeshConfig)`
: `MeshConfig` описывает настройку взаимодействий между компонентами Istio, местонахождение источников конфигурации и т. д.

`ProxyConfig (mesh.istio.io/v1alpha1.ProxyConfig)`
: `ProxyConfig` описывает параметры инициализации Envoy: местоположение начальной конфигурации, привязки портов и т. д.

`MeshNetworks` (`mesh.istio.io/v1alpha1.MeshNetworks`)

`MeshNetworks` описывает набор сетей, охватываемых сервисной сеткой, с адресами входных шлюзов каждой сети.

`MeshConfig` в основном используется для настройки следующих параметров: включена ли политика и/или телеметрия, откуда загружать конфигурацию, и настройки балансировки нагрузки в зависимости от местоположения. `MeshConfig` содержит следующий исчерпывающий набор настроек:

- использование Mixer:
  - адреса серверов политики и телеметрии;
  - включена ли проверка политик во время выполнения?
  - отказывать ли в открытии или закрытии, если Mixer Policy недоступен или возвращает ошибку;
  - проверять ли политику на стороне клиента?
  - использовать ли сходство сеансов для направления в один и тот же экземпляр Mixer Telemetry? Сходство сеансов всегда включено для Mixer Policy (от нее зависит производительность системы!);
- настройка прокси на прием:
  - порты приема трафика (т. е. порты, куда трафик перенаправляется с помощью iptables) и запросов HTTP PROXY;
  - настройки keepalive и тайм-аута TCP-соединения;
  - формат журнала доступа, имя файла и способ кодирования (JSON или текст);
  - разрешить ли весь исходящий трафик или только из сервисов, известных Pilot?
  - где принимать секреты от Citadel (SDS API) и как настроить проверку идентичности (в среде с токенами на локальных машинах)?
- поддерживать ли ресурсы Kubernetes Ingress?
- набор источников конфигурации для всех компонентов Istio (например, локальная файловая система или Galley) и как с ними взаимодействовать (адреса, использовать ли Transport Layer Security (TLS), какие секреты и т. д.);
- настройки балансировки нагрузки в зависимости от местоположения – настройка аварийного переключения и разделения трафика между зонами и регионами (подробнее об этом будет сказано в главе 8).

`ProxyConfig` в основном используется для создания пользовательской конфигурации загрузки для Envoy. `ProxyConfig` определяет следующие аспекты:

- местонахождение файла с загрузочной конфигурацией Envoy, а также местонахождение самого двоичного файла Envoy;
- имя сервиса, к которому будет «прицепляться» Envoy;
- настройки выключения (порядок остановки соединения и горячего перезапуска);
- местонахождение сервера xDS Envoy (Pilot) и способы связи с ним;
- настройки тайм-аута соединения;
- какие порты прокси должны использоваться для администрирования и получения статистики?
- параллелизм Envoy (число рабочих потоков);

- как Envoy привязывает сокет для перехвата трафика (через iptables REDIRECT или TPROXY)?
- местонахождение сборщика телеметрии (т. е. куда отправлять данные телеметрии).

`MeshNetworks` определяет набор именованных сетей, способ передачи трафика в эту сеть и местоположение этой сети. Каждая сеть представляет собой либо диапазон бесклассовой междоменной маршрутизации (*Classless Inter-Domain Routing*, CIDR), либо набор конечных точек, возвращаемый реестром сервисов (например, Kubernetes API Server). Объект `ServiceEntry`, используемый для определения сервисов в Istio, имеет набор конечных точек. У каждой конечной точки может быть метка сети, чтобы `ServiceEntry` мог описать сервис, развернутый в нескольких сетях (или кластерах). Об этом мы поговорим в разделе «*Обнаружение сервисов*».

Большинство значений в `MeshConfig` не могут обновляться динамически, и для вступления их в силу необходимо перезапустить уровень управления. Аналогичным образом обновление значений в `ProxyConfig` вступает в силу только после повторного развертывания Envoy (например, когда в Kubernetes происходит перепланирование пода). `MeshNetworks` можно динамически обновлять во время работы без перезапуска каких-либо компонентов уровня управления.

В Kubernetes бóльшая часть конфигураций `MeshConfig` и `ProxyConfig` определяется параметрами установки Helm, хотя не все они доступны через интерфейс Helm. Чтобы полностью контролировать процесс установки, потребуется дополнительно обработать файл, генерируемый диспетчером пакетов Helm.

Сетевая конфигурация

Сетевая конфигурация – хлеб и масло Istio – это конфигурация, используемая для управления потоком трафика через сетку. Мы подробно рассмотрим каждый объект в главе 8 и обсудим, как эти конструкции используются вместе для управления потоком трафика через сетку. Здесь будет рассмотрен каждый объект, но только на высоком уровне, чтобы связать конфигурацию Istio с Envoy xDS API (обсуждается в главе 5), помочь понять конфигурационный сервер Pilot и показать отладку системы (об этом поговорим в последующих разделах).

`ServiceEntry` является центральным элементом сетевых API Istio. `ServiceEntry` определяет сервис по именам – как набор имен хостов, используемых клиентами для вызова сервиса. Более подробно об этом мы поговорим в следующем разделе. Правила `DestinationRule` управляют взаимодействиями клиентов с сервисом, а именно: стратегии балансировки нагрузки, обнаружения отклонений, обрыва цепи и организации пула используемых соединений; настройки TLS и т. д. `VirtualService` задают конфигурацию потока трафика к сервису: маршрутизация L7 и L4, формирование трафика, повторные попытки, тайм-ауты и т. д. Шлюзы определяют доступность сервисов из-за пределов сетки: какие имена хостов каким сервисам соответствуют, как обслуживать сертификаты для этих хостов и многое другое. Прокси управляют доступностью сервисов внутри сетки: какие сервисы доступны и каким клиентам.

Обнаружение сервисов

Для получения информации о сервисах, конечных точках и локальном окружении Pilot интегрируется с различными системами обнаружения сервисов, такими как Kubernetes API Server, Consul и Eureka. Адаптеры Pilot получают информацию о сервисах из своих источников и синтезируют из этих данных объекты `ServiceEntry`. Например, интеграция с Kubernetes использует Kubernetes SDK для получения событий создания сервисов и обновления конечных точек. Используя эти данные, адаптер реестра Pilot синтезирует объект `ServiceEntry`. Далее `ServiceEntry` используется для обновления внутренней модели Pilot и генерации обновленной конфигурации уровня данных.

Исторически сложилось так, что адаптеры реестра Pilot были реализованы внутри него на Golang. С появлением Galley эти адаптеры можно выделить из Pilot. Адаптер обнаружения сервисов может работать как отдельное задание (или автономный процесс, скажем, выполняемый системой CI), читать существующий реестр сервисов и создавать из него набор объектов `ServiceEntry`. Затем полученные `ServiceEntry` можно загрузить в Galley в виде файлов, передать в Kubernetes API Server или самостоятельно реализовать сервер *Mesh Config Protocol* и передать `ServiceEntry` в Galley. *Mesh Config Protocol* и сбор конфигурации в целом описаны в главе 11. В статических окружениях (например, в традиционных виртуальных машинах с редко меняющимися IP-адресами) генерация статических `ServiceEntry` может быть эффективным способом включения Istio.

Обслуживание конфигурации

Из трех источников конфигурации — сети, сетки и обнаружения сервисов – Pilot создает модель окружения и состояния развертывания. Асинхронно, по мере развертывания в кластере, прокси подключаются к Pilot. Pilot группирует прокси на основе их меток и подключенных к ним сервисов. Используя эту модель, Pilot генерирует ответы *Discovery Service* (xDS) для каждой группы подключенных прокси (подробнее о Discovery Service API далее). Когда подключается прокси, Pilot посылает текущее состояние окружения и конфигурацию. Учитывая в целом динамический характер нижележащей платформы (или платформ), модель обновляется с определенной частотой. Обновление модели требует обновления текущего набора конфигураций xDS. Когда изменяется конфигурация xDS, Pilot определяет, какие прокси эти изменения затрагивают, и передает им обновленную конфигурацию.

xDS API подробно обсуждается в главе 5, но давайте повторно рассмотрим некоторые высокоуровневые понятия, чтобы описать, как сетевая конфигурация Istio проявляется в xDS. Конфигурацию прокси (Envoy) можно разделить на две основные группы:

- приемники и маршруты;
- кластеры и конечные точки.

Приемники настраивают набор фильтров (например, поддержка HTTP в Envoy определяется фильтром HTTP) и связь между этими фильтрами и портами. Есть два типа приемников: *физический* и *виртуальный*. Физический приемник – тот, к которому Envoy привязывается через указанный порт. Виртуальный приемник получает трафик от физического приемника, но не привязан к порту (он получает трафик от физических приемников). Маршруты сопутствуют приемникам и определяют, как приемники направляют трафик в определенный кластер (например, сопоставляя HTTP-путь или посредством *Service Name Indication* (SNI). Кластер – это группа конечных точек с информацией о том, как с ними связаться (настройки TLS, стратегия балансировки нагрузки, настройки пула соединений и т. д.). Кластер аналогичен «сервису» (например, один сервис Kubernetes может объявляться как один кластер). Наконец, конечными точками являются отдельные сетевые узлы (IP-адреса или доменные имена), куда Envoy направит трафик.

В рамках этой конфигурации элементы ссылаются друг на друга по имени. Так, приемник направляет трафик именованному маршруту, маршрут – именованному кластеру, а кластер – набору конечных точек. Pilot обеспечивает непротиворечивость этих имен. Посмотрим в следующем разделе, чем эти имена полезны для отладки системы.

Замечание про «x»

Envoy API мы называем *xDS API*, так как каждый конфигурационный примитив – приемник, маршрут, кластер, конечная точка – имеют свой собственный Discovery Service, названный в его честь. Каждый Discovery Service позволяет обновлять свой ресурс. Вместо того чтобы ссылаться по отдельности на LDS, RDS, CDS и EDS, мы используем обобщенное название *xDS API*.

Сетевая конфигурация Istio почти напрямую отображается в Envoy API:

- шлюзы настраивают физические приемники;
- `VirtualService` конфигурируют как виртуальные приемники (совпадения имен хостов кодируются как отдельные приемники, а обработка протокола конфигурируется приемниками с помощью специальных фильтров), так и маршруты (условия соответствия HTTP/TLS, конфигурация повторной передачи и тайм-аутов и т. д.);
- `ServiceEntry` создают кластеры и заполняют их конечные точки;
- `DestinationRule` конфигурируют взаимодействие с кластерами (секреты, стратегия балансировки нагрузки, разрыв связи, объединение соединений в пулы и т. д.) и создают новые кластеры, когда они используются для определения подмножеств.

Последним элементом сетевой конфигурации Istio являются настройки прокси для сервисов. Они не относятся напрямую к примитиву Envoy; Istio использует эти настройки для выбора конфигурации, посылаемой каждой группе Envoy.

Теперь давайте рассмотрим, как обычная конфигурация Istio объявляется конфигурацией Envoy xDS, и дадим несколько советов по отладке сетевой конфигурации Istio.

Отладка и устранение неисправностей в Pilot

Данный раздел посвящен устранению неисправностей в Pilot и может служить дополнением к главе 11, посвященной отладке. Istio – сложная система с большим количеством подвижных частей. Вплоть до полного изучения Istio может быть сложно понять, почему система ведет себя так или иначе (проблема усугубляется тем, что зачастую система вообще перестает обслуживать трафик!). К счастью, растет число инструментов, помогающих понять и разобрать состояние системы. В этом разделе мы даем обзор некоторых инструментов, особенно полезных для понимания и поиска неисправностей в сетях Istio.

istioctl

`istioctl` содержит множество полезных инструментов, помогающих понять состояние развертывания Istio, включая `istioctl authn` для проверки состояния mTLS в сетке, получения метрик для каждого пода и проверки конфигурации Pilot и Envoy. Еще два, `istioctl proxy-config` и `istioctl proxy-status`, просто бесценны для понимания текущего состояния конфигурации сети.

К сожалению, несколько следующих инструментов (в частности, `proxy-config` и `proxy-status`) специфичны для Kubernetes, поскольку в настоящее время их реализация полагается на Kubernetes. Например, `istioctl proxy-config` использует `kubectl exec` для получения данных с удаленной машины.

В будущем аналогичные инструменты будут созданы и для других платформ. Где это возможно, мы описываем, как реализован инструмент, чтобы помочь тем, кто работает не на платформах Kubernetes. Более подробно о том, как `istioctl proxy-config` взаимодействует с Kubernetes, см. в главе 11.

Для поддержки других платформ (и других сервисных сеток) мы также указываем, какие инструменты могут заполнить эти пробелы. Meshery (https://oreil.ly/c_eJo) – одно из решений, графически представляющее ту же информацию о статусе сетки, что и `istioctl proxy-config` и `proxy-status` (для Istio и других сервисных сеток). Meshery позволяет сопоставить текущее состояние в сравнении с *запланированным состоянием* конфигурации Istio, что упрощает управление пробными прогонами развертывания и гарантирует, что внесенные изменения принесут желаемый результат.

```
istioctl proxy-config <bootstrap | listener | route | cluster> <kubernetes pod>
```

Подключается к указанному поду и запрашивает административный интерфейс прокси для получения текущей конфигурации прокси. Можно получить загрузочную конфигурацию прокси (обычно содержащую настройки, необходимые для общения с Pilot), его приемники, маршруты и кластеры. `proxy-config` поддерживает флаг вывода (`--output` или просто `-o`), используемый для печати в формате JSON всего тела конфигурации Envoy. В разделе «Трассировка конфигурации» он используется для анализа настроек, добавляемых сеткой Istio в конфигурацию прокси.

```
istioctl proxy-status <Istio service>
```

Подключается к отладочному интерфейсу Pilot, получает xDS-статус каждого экземпляра прокси (если предоставлено имя сервиса, анализируется только прокси этого сервиса) и сообщает, соответствует ли конфигурация прокси последним настройкам Pilot, и если нет, то насколько отстает. Эта информация особенно ценна для выявления проблемной конфигурации, влияющей только на подмножество прокси.

Отладка Pilot

Pilot предоставляет разнообразные конечные точки для анализа своего состояния. К сожалению, они документированы крайне плохо. На момент написания не было открытых документов, детально описывающих конечные точки. Эти конечные точки имеют префикс `/debug/`, возвращают JSON фрагменты различных конфигураций, хранящиеся в Pilot.

Для проверки состояния подключенных к Pilot прокси используйте следующие конечные точки:

`/debug/edsz`
: Выводит все предварительно рассчитанные EDS-ответы Pilot (т. е. конечные точки, посылаемые каждому подключенному прокси).

`/debug/adzz`
: Выводит список приемников, маршрутов и кластеров, передаваемых каждому прокси, подключенному к Pilot.

`/debug/cdsz`
: Выводит набор кластеров, передаваемых каждому прокси, подключенному к Pilot.

`/debug/synz`
: Выводит состояние подключений ADS, CDS и EDS всех прокси, подключенных к Pilot. В частности, показывает текущий одноразовый номер в Pilot и последние одноразовые номера, для которых получено подтверждение от Envoy. Это позволяет увидеть, какие Envoy не принимают обновления конфигурации.

Следующие конечные точки показывают картину мира Pilot (его реестры сервисов):

`/debug/registryz`
: Выводит список сервисов из всех реестров, известных Pilot.

`/debug/endpointz[?brief=1]`
: Выводит конечные точки для каждого сервиса, известного Pilot, включая их порты, протоколы, учетные записи сервисов, метки и т. д. С флагом `brief` получится удобочитаемая таблица (без этого флага выводится трудно читаемый блок в формате JSON). Это устаревшая конечная точка, вместо нее рекомендуется использовать `/debug/endpointShardz`, которая предоставляет более подробную информацию.

`/debug/endpointShardz`
: Выдает конечные точки для каждого сервиса, известного Pilot, сгруппированные по реестрам (*сегментам* (*shard*), где зарегистрированы конечные

точки, с точки зрения Pilot). Например, если один и тот же сервис существует и в Consul, и в Kubernetes, его конечные точки будут сгруппированы в два сегмента, по одной для Consul и Kubernetes. Эта конечная точка предоставляет те же данные, что и `/debug/endpointz`, а также сеть конечной точки, расположение, вес для балансировщика нагрузки, представление в конфигурации Envoy xDS и многое другое.

`/debug/workloadz`
: Выдает набор конечных точек (рабочих нагрузок), подключенных к Pilot, и их метаданные (например, метки).

`/debug/configz`
: Выдает весь набор конфигураций Istio, известных Pilot. Возвращаются только проверенные конфигурации, используемые Pilot при построении своей модели. Полезно для понимания ситуаций, когда Pilot самостоятельно не обрабатывает новую конфигурацию.

Далее перечислены различные конечные точки с отладочной информацией более высокого уровня:

`/debug/authenticationz [?proxyID=pod_name.namespace]`
: Показывает состояние политики аутентификации Istio целевого прокси для каждого обслуживаемого им хоста и порта, включая: имя политики аутентификации; имя затрагивающего его `DestinationRule`; как осуществляется обмен через этот порт: с использованием mTLS, стандартного TLS или в открытом виде и наличие конфликтов в настройках этого порта (общая причина 500-й ошибки в новых установках Istio).

`/debug/config_dump [?proxyID=pod_name.namespace]`
: Показывает приемники, маршруты и кластеры данного узла; выдачу можно сравнить, используя `diff` с результатами вывода `istioctl proxy-config`.

`/debug/push_status`
: Показывает статус каждой подключенной конечной точки на момент последней рассылки конфигураций из Pilot; включает статус каждого подключенного прокси, когда начался (и закончился) период рассылки, и идентичности, назначенные каждому порту каждого хоста.

ControlZ

Каждый компонент уровня управления Istio предоставляет административный интерфейс для настройки журналирования, просмотра информации о процессе и окружении, а также для просмотра метрик этого экземпляра. ControlZ, чаще всего используемый для настройки уровня детальности журналов, позволяет во время работы независимо и динамически изменять параметры журналирования для каждого *сегмента* (*scope*). Компоненты Istio используют общую систему журналирования с концепцией сегментов. Например, Pilot определяет сегменты для записи в журнал соединений Envoy API; скажем, один сегмент для ADS-соединений, другой – для EDS-соединений и третий – для CDS-соединений. Дополнительные сведения о ControlZ см. в разделе «Изучение компонентов Istio» главы 11.

Prometheus

Pilot наряду с другими компонентами уровня управления Istio обслуживает конечную точку Prometheus с подробными метриками их внутреннего состояния. Стандартная конфигурация Grafana по умолчанию включает индикаторы, использующие эти метрики для отображения состояния каждого компонента уровня управления Istio. Метрики можно использовать для отладки внутреннего состояния Pilot. По умолчанию Pilot размещает конечную точку Prometheus на порту 8080 в `/metrics` (т. е. `kubectl exec -it PILOT_POD -n istio-system -c discovery - curl localhost:8080/metrics`).

Трассировка конфигурации

Отслеживание этапов создания и распространения конфигурации, начиная с Pilot и заканчивая прокси, может оказаться сложной задачей. Конечные точки отладки Pilot (описанные выше) вместе с `istioctl` являются удобными инструментами для исследования Pilot и любых изменений в нем. В данном разделе описывается применение этих инструментов для исследования конфигурации Istio до и после, а также результирующей xDS-конфигурации, передаваемой в прокси.

Конфигураций слишком много, чтобы продемонстрировать все их проявления. Далее мы покажем примеры основных типов конфигурации Istio и результирующие конфигурации Envoy; подчеркнем основные сходства и объясним, как другие изменения в той же конфигурации Istio проявятся в Envoy, чтобы вы могли проверить и убедиться в этом сами и использовать полученные знания для диагностики и решения большинства проблем, с которыми придется иметь дело.

Приемники

Определения ресурсов `Gateway` и `VirtualService` описывают *приемники* (*listeners*) в Envoy. `Gateway` предоставляет физические приемники (привязанные к порту в сети), а `VirtualService` – виртуальные приемники (*не* привязанные к порту, но получающие трафик от физических приемников). Примеры 7.1 и 7.2 демонстрируют, как определение Gateway в конфигурации Istio проявляется в xDS-конфигурации (см. foo-gw.yaml (https://oreil.ly/8SW3s) в GitHub-репозитории этой книги).

Пример 7.1 ❖ Gateway для обслуживания HTTP-трафика поддоменов в домене http://foo.com (с использованием чистой установки Istio)

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: foo-com-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
```

```
- hosts:
  - "*.foo.com"
   port:
     number: 80
     name: http
     protocol: HTTP
```

Это определение создает один HTTP-приемник на порту 80 входного шлюза (см. пример 7.2).

Пример 7.2 ❖ Конфигурация приемника Envoy (LDS) для шлюза, созданного в примере 7.1

```
$ istioctl proxy-config listener istio-ingressgateway_PODNAME
-o json -n istio-system
[
    {
        "name": "0.0.0.0_80",
        "address": {
            "socketAddress": {
                "address": "0.0.0.0",
                "portValue": 80
            }
        },
        "filterChains": [
            {
                "filters": [
                    {
                        "name": "envoy.http_connection_manager",
...
                            "rds": {
                                "config_source": {
                                    "ads": {}
                                },
                                "route_config_name": "http.80"
                            },
...
```

Обратите внимание, что только что созданный фильтр обслуживает адрес 0.0.0.0. Этот приемник пропускает весь HTTP-трафик, поступающий в порт 80, независимо от адреса хоста-получателя. Если в этом определении `Gateway` настроить поддержку TLS, появится новый приемник, пересылающий трафик только хостам с поддержкой TLS, остальной трафик будет обслуживаться этим общим приемником. Привяжем `VirtualService` к этому `Gateway`, как показано в примере 7.3 (см. foo-vs.yaml в репозитории GitHub этой книги (https://oreil.ly/OZqjU)).

Пример 7.3 ❖ VirtualService привязывается к шлюзу из примера 7.1 и создает виртуальные приемники в Envoy

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
```

```
metadata:
  name: foo-default
spec:
  hosts:
 - bar.foo.com
  gateways:
 - foo-com-gateway
  http:
 - route:
   - destination:
       host: bar.foo.svc.cluster.local
```

Как это определение проявляется в виде виртуальных приемников, см. пример 7.4:

Пример 7.4 ❖ Конфигурация приемника Envoy (LDS) для VirtualService из примера 7.3 – никаких изменений по сравнению с конфигурацией, приведенной в примере 7.2; все происходит в маршрутах!

```
$ istioctl proxy-config listener istio-ingressgateway_PODNAME -o json
[
    {
        "name": "0.0.0.0_80",
        "address": {
            "socketAddress": {
                "address": "0.0.0.0",
                "portValue": 80
            }
        },
        "filterChains": [
            {
                "filters": [
                  {
                      "name": "envoy.http_connection_manager",
...
                   "rds": {
                     "config_source": {
                       "ads": {}
                     },
                     "route_config_name": "http.80"
                   },
...
```

В примере 7.4 не видно никаких изменений в настройке приемника. Это происходит потому, что приемник на IP 0.0.0.0 обслуживает весь HTTP-трафик, поступающий в порт 80. Однако TLS в приемнике будет настроен не так. Если в определение `Gateway` добавить настройки TLS, появится новый приемник, созданный только для хостов в секции с TLS. Остальной трафик будет обслуживаться приемником по умолчанию. В случае HTTP все действия выполняются в маршрутах. Другие протоколы, например TCP, передают на приемник больше логики. Поэкспериментируйте, создав несколько `Gateway` с различными прото-

колами, чтобы увидеть их проявление в качестве приемников. Идеи и примеры можно найти в репозитории GitHub книги (https://oreil.ly/istio-up-and-running).

Также нужно обратить внимание на конфигурацию Mixer в приемниках. Конфигурация Mixer в Envoy отображается как в приемниках (где устанавливаются атрибуты источника), так и в маршрутах (где устанавливаются атрибуты назначения). Использование `MeshConfig` для отключения *проверок* Mixer приведет к несколько иной конфигурации, как и отключение *отчетов* Mixer. Если отключить как *проверки*, так и *отчеты*, то конфигурация Mixer полностью исчезнет из Envoy.

Мы также рекомендуем попробовать разные протоколы для портов (или определить один `Gateway` с несколькими портами с различными протоколами), чтобы увидеть, какие фильтры получатся в результате. Настройка различных TLS-параметров внутри `Gateway` также приводит к изменениям в конфигурации сгенерированного приемника. Вы всегда увидите специфический фильтр протокола, настроенный в приемнике для каждого используемого протокола (для HTTP – это агент `http_connection_man` и его `router`; для MongoDB – другой; для TCP – еще один и т. д.). Хорошо также попробовать различные комбинации хостов в `Gateway` и `VirtualService`, чтобы увидеть их влияние. Мы подробно рассмотрим, как они работают вместе, – как *связать* `VirtualService` с `Gateway` – в главе 8.

Маршруты

Мы видели, как `VirtualService` приводит к созданию приемников (или не приводит, как в нашем примере!). Бóльшая часть конфигурации `VirtualService` на самом деле в Envoy проявляется в виде маршрутов. Маршруты бывают разных типов и группируются по протоколам.

В примере 7.5 показан список маршрутов, получившихся из конфигурации `VirtualService` в примере 7.3. В данном случае маршрут получился довольно простым, потому что `VirtualService` просто переправляет трафик единственному сервису. Пример показывает политику Retry Policy (повтор по умолчанию) и встроенную конфигурацию Mixer (используемую для передачи телеметрии обратно в Mixer).

Пример 7.5 ❖ Конфигурация маршрута Envoy (RDS) для VirtualService в примере 7.3

```
$ istioctl proxy-config route istio-ingressgateway_PODNAME -o json
[
    {
        "name": "0.0.0.0_80",
        "virtualHosts": [
            {
                "name": "bar.foo.com:80",
                "domains": [
                    "bar.foo.com",
                    "bar.foo.com:80"
                ],
                "routes": [
```

```
                {
                    "match": {
                        "prefix": "/"
                    },
                    "route": {
                        "cluster": "outbound|8000||bar.foo.svc.cluster.local",
                        "timeout": "0s",
                        "retryPolicy": {
                            "retryOn": "connect-failure,refused-stream,
                            unavailable,cancelled,resource-exhausted,
                            retriable-status-codes","numRetries": 2,
                            "retryHostPredicate": [
                                {
                                    "name": "envoy.retry_host_predicates.previous_hosts"
                                }
                            ],
                            "hostSelectionRetryMaxAttempts": "3",
                            "retriableStatusCodes": [
                                503
                            ]
                        },
...
```

Обновим маршрут, включив некоторые условия совпадений. В результате получим различные маршруты для Envoy, как показано в примере 7.6 (см. foo-routes.yaml в репозитории GitHub книги (https://oreil.ly/bbevw)).

Пример 7.6 ❖ VirtualService связывает /whiz с сервисом whiz

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-default
spec:
hosts:
- bar.foo.com
gateways:
- foo-com-gateway
http:
 - match:
   - uri:
       prefix: /whiz
    route:
   - destination:
       host: whiz.foo.svc.cluster.local
- route:
 - destination:
     host: bar.foo.svc.cluster.local
```

Аналогичным образом можно добавить повторы, разделить трафик между несколькими адресатами, ввести неисправности и многое другое. Все эти настройки в `VirtualService` проявляются как маршруты в Envoy (см. пример 7.7).

Пример 7.7 ❖ Конфигурация маршрута Envoy (RDS) для VirtualService из примера 7.6

```
$ istioctl proxy-config route istio-ingressgateway_PODNAME -o json
[
    {
        "name": "http.80",
        "virtualHosts": [
            {
                "name": "bar.foo.com:80",
                "domains": [
                    "bar.foo.com",
                    "bar.foo.com:80"
                ],
                "routes": [
                    {
                        "match": {
                            "prefix": "/whiz"
                        },
                        "route": {
                            "cluster": "outbound|80||whiz.foo.svc.cluster.local",
...
                    {
                        "match": {
                            "prefix": "/"
                        },
                        "route": {
...
```

Как видите, настройка сопоставления URI проявляется в виде дополнительного маршрута с указанным префиксом. Предыдущий маршрут для «/» тоже остался, но он следует за определением нового маршрута. Сверки в Envoy проводятся по порядку, и этот порядок совпадает с порядком в `VirtualService`.

Кластеры

Если использовать `istioctl` для просмотра кластеров, можно заметить, что Istio генерирует кластер для каждого сервиса и порта в сетке. Можно создать новый `ServiceEntry`, подобный описанному в примере 7.8, чтобы увидеть новый кластер в Envoy, как показано в примере 7.9 (см. some-domain-se.yaml в репозитории на GitHub книги (https://oreil.ly/8F4cu)).

Пример 7.8 ❖ ServiceEntry для сайта some.domain.com со статическим IP-адресом

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: http-server
spec:
  hosts:
 - some.domain.com
```

```
  ports:
 - number: 80
    name: http
    protocol: http
resolution: STATIC
endpoints:
- address: 2.2.2.2
```

Пример 7.9 ❖ Конфигурация кластера Envoy (CDS) для ServiceEntry из примера 7.8

```
$ istioctl proxy-config cluster istio-ingressgateway_PODNAME -o json [
...
    {
        "name": "outbound|80||some.domain.com",
        "type": "EDS",
        "edsClusterConfig":
            { "edsConfig": {
                "ads": {}
            },
            "serviceName": "outbound|80||some.domain.com"
        },
        "connectTimeout": "10s", "circuitBreakers": {
            "thresholds": [
                {
                    "maxRetries": 1024
                }
            ]
        }
    },
...
```

В результате образуется единый кластер `outbound|80||some.domain.com`. Обратите внимание, как Istio кодирует вход и выход в имени кластера вместе с портом.

Добавление новых портов (с различными протоколами) в `ServiceEntry` приводит к созданию новых кластеров. Другим инструментом, используемым для создания и обновления кластеров в Istio, является `DestinationRule`. Определение подмножеств создает новые кластеры (см. примеры 7.10 и 7.11), а обновление настроек балансировки нагрузки и TLS влияет на конфигурацию внутри самого кластера (см. файл some-domain-dest.yaml в репозитории GitHub книги (https://oreil.ly/fh1tx)).

Пример 7.10 ❖ DestinationRule для сайта some.domain.com, разделяющий его на две подгруппы

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: some-domain-com
spec:
  host: some.domain.com
  subsets:
 - name: v1
```

```
    labels:
      version: v1
    name: v2
- labels:
    version: v2
```

Пример 7.11 ❖ Конфигурация кластера Envoy (CDS) для DestinationRule в примере 7.10

```
$ istioctl proxy-config cluster istio-ingressgateway_PODNAME -o json
[
...
    {
        "name": "outbound|80||some.domain.com",
...
    },
    {
        "name": "outbound|80|v1|some.domain.com",
...
        "metadata": {
            "filterMetadata": {
                "istio": {
                    "config": "/apis/networking/v1alpha3/namespaces/default
                    /destination-rule/some-domain-com"
                }
            }
        }
    },
    {
        "name": "outbound|80|v2|some.domain.com",
...
    },
...
```

Отметим, что сохранился оригинальный кластер `outbound|80||some.domain.com` и появился новый кластер для каждого подмножества Subset. Кроме того, Istio добавляет в конфигурацию Envoy свое правило, оказывающее помощь при отладке.

В этой главе мы рассмотрели компонент Pilot: его базовую модель, источники конфигураций, которые он использует для создания модели сетки, порядок использования этой модели для передачи конфигурации в Envoy, отладку и, наконец, преобразования, которые выполняет Pilot, чтобы из конфигурации Istio получить конфигурацию для Envoy. Этой информации достаточно для отладки и решения подавляющего большинства вопросов, с которыми сталкиваются начинающие пользователи Istio.

Глава 8

Управление трафиком

Среди основных возможностей всех сервисных сеток – управление трафиком, что само по себе – обширная функциональная область. Безусловно, это относится и к Istio. Рассматривая в данной главе в качестве темы исследования управление трафиком, мы начинаем изучение возможностей Istio в контексте прохождения запросов в системе, по мере продвижения знакомясь с сетевыми API Istio. Мы рассмотрим, как использовать API для настройки потока трафика, выполнять канареечное развертывание, устанавливать политики тайм-аутов и повторных попыток, совместимых со всеми сервисами, и, наконец, тестировать режимы отказов приложения с контролируемой, повторяемой имитацией ошибок.

Как движется трафик в Istio?

Для понимания работы сетевых API Istio важно понимать, как на самом деле запросы проходят через Istio. Pilot, как мы узнали из предыдущей главы, понимает топологию сервисной сетки и использует эти знания наряду с дополнительными конфигурациями сети Istio, предоставленными *пользователем*, для настройки прокси в сетке. В главе 7 приведены дополнительные сведения о конфигурации, рассылаемой компонентом Pilot в прокси.

Выполняя функции прокси уровня данных, Envoy перехватывает все входящие и исходящие запросы в режиме реального времени (по мере прохождения трафика по сервисной сетке). Перехват осуществляется *прозрачно* с помощью правил iptables или пакетного фильтра Berkeley Packet Filter (BPF), направляющих весь входящий и исходящий сетевой трафик через Envoy. Envoy анализирует запрос и использует имя хоста запроса, SNI или виртуальный IP-адрес сервиса, чтобы определить *цель* запроса (сервис, которому клиент *намеревается* отправить запрос). Envoy применяет правила маршрутизации для определения *адресата* запроса (сервиса, которому прокси *на самом деле* собирается отправить запрос). Определив пункт назначения, Envoy применяет правила пункта назначения. Правила назначения включают в себя стратегию балансировки нагрузки, используемую для выбора *конечной точки* (конечной точкой является адрес исполнителя, поддерживающего сервис назначения). Как правило, у сервисов для обработки запросов имеется более одного исполнителя; запросы могут балансироваться между ними. Наконец, Envoy перенаправляет перехваченный запрос в конечную точку.

Ряд моментов заслуживают дальнейшего освещения. Во-первых, желательно, чтобы приложения работали *открытым текстом* (взаимодействовали без шифрования) с прокси и позволяли ему управлять безопасностью транспорта. Например, приложение работает по HTTP-протоколу с прокси и позволяет ему инкапсулировать трафик в протокол HTTPS. Это позволяет прокси собирать метаданные L7 о запросах, на основе которых Istio генерирует метрики L7 *и* манипулирует трафиком на основе политики L7. Если прокси не поддерживает TLS, Istio сможет генерировать метрики и применять политику только к сегменту L4 запроса, ограничиваясь применением политик лишь к содержимому IP-пакета и TCP-заголовкам (по сути, к адресам отправителя/получателя и номерам портов). Во-вторых, нужно выполнять балансировку нагрузки на стороне клиента, а не полагаться на традиционную балансировку через обратные прокси. Балансировка нагрузки на стороне клиента означает, что можно устанавливать сетевые подключения напрямую от клиентов к серверам, при этом поддерживая надежную, хорошо работающую систему. Это, в свою очередь, позволяет использовать более эффективные сетевые топологии с меньшим количеством переходов, чем в традиционных системах, зависящих от обратных прокси.

Как правило, Pilot имеет подробную информацию о конечных точках сервисов, зарегистрированных в реестре, которую он передает непосредственно в прокси. Если не настроить прокси на другое действие, во время работы он выбирает конечную точку из статического набора, переданного компонентом Pilot, и не выполняет динамическое разрешение адреса (например, через DNS). Поэтому единственное, куда Istio может маршрутизировать трафик, – это имена хостов в реестре сервисов Istio. В более новых версиях Istio есть возможность (по умолчанию отключена в версии 1.1) изменить это поведение и позволить Envoy направлять трафик неизвестным сервисам, не зарегистрированным в Istio, при условии что приложение предоставляет IP-адрес.

В следующем разделе мы поговорим об *именах хостов* (*hostnames*), лежащих в основе сетевой модели Istio, и о том, как сетевые API Istio позволяют создавать имена хостов для описания приложений и управлять потоками трафика к ним.

Работа сетевых API Istio

Приложения обращаются к сервисам по именам (например, по именам хостов, которые разрешаются через DNS), чтобы избежать проблем с IP-адресами (которые первоначально неизвестны, могут измениться в любое время, трудно запоминаются и могут транслироваться между v4 и v6 в зависимости от окружения). Поэтому в сетевой конфигурации Istio принята *модель адресации на основе имен* (*name-centric model*), в которой:

- `Gateway` (https://oreil.ly/uPLZa) экспортирует имена;
- `VirtualService` (https://oreil.ly/_qE97) определяет имена и маршруты к ним;
- `DestinationRule` (https://oreil.ly/rj42r) описывает, как взаимодействовать с именованными приложениями;
- `ServiceEntry` (https://oreil.ly/tyvAq) позволяет создавать новые имена.

Запросы приложений инициируются вызовом сервиса по имени, как показано на рис. 8.1.

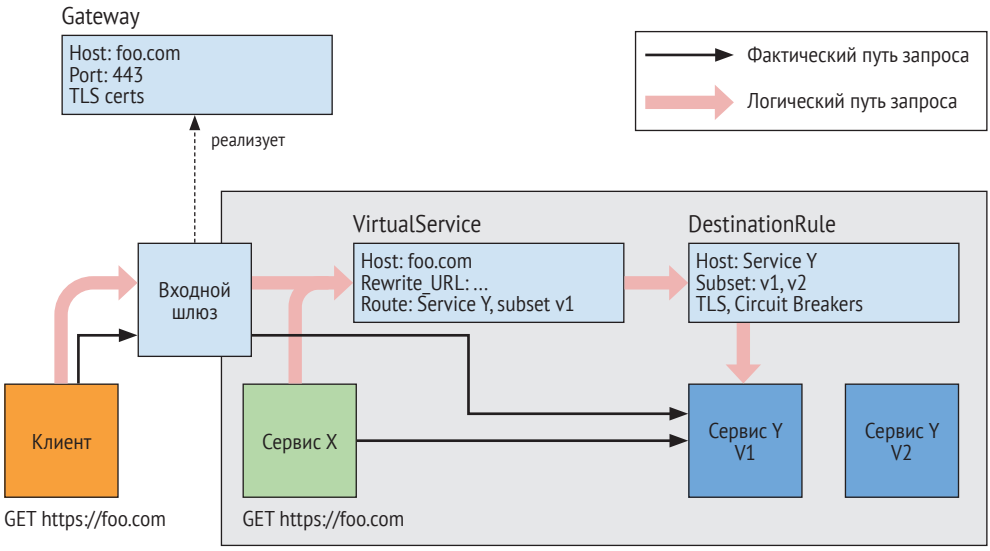

Рис. 8.1 ❖ Ключевые сетевые концепции Istio, связанные с прохождением трафика через систему

ServiceEntry

`ServiceEntry` – это способ ручного добавления/удаления сервисов в реестр (из реестра) Istio. К сервисам из реестра можно обращаться по именам и ссылаться на них в других конфигурациях Istio. В простейшем случае ServiceEntry можно использовать для связывания имен с IP-адресами, как показано в примере 8.1 (см. static-se.yaml в репозитории GitHub книги (https://oreil.ly/Us7jl)).

Пример 8.1 ❖Пример определения ServiceEntry

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: http-server
spec:
  hosts:
  - some.domain.com
  ports:
  - number: 80
    name: http
  protocol: http
resolution: STATIC
endpoints:
- address: 2.2.2.2
```

На основе ServiceEntry из примера 8.1 прокси сетки будут направлять запросы на имя some.domain.com по IP-адресу 2.2.2.2. Как показывает пример 8.2, ServiceEntry можно использовать для преобразования имен, используемых в Istio, в имена, разрешаемые через DNS (см. dns-se.yaml в репозитории GitHub книги (https://oreil.ly/cEc5k)).

Пример 8.2 ❖ ServiceEntry Istio с разрешением имени через DNS

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: external-svc-dns
spec:
  hosts:
 - foo.bar.com
  location: MESH_EXTERNAL
  ports:
 - number: 443
    name: https
    protocol: HTTP
  resolution: DNS
  endpoints:
 - address: baz.com
```

Определение ServiceEntry в примере 8.2 заставит прокси пересылать запросы к foo.bar.com на имя baz.com, используя DNS для его разрешения. В данном примере заявлено, что сервис находится вне сетки (location: MESH_EXTERNAL), поэтому прокси не будут пытаться использовать mTLS.

Все реестры сервисов, с которыми интегрируется Istio (Kubernetes, Consul, Eureka и т. д.), преобразуют свои данные в ServiceEntry. Например, сервис Kubernetes с одним подом (и, следовательно, одной конечной точкой) отображается непосредственно в ServiceEntry с именем хоста и IP-адресом конечной точки, как показано в примере 8.3 (см. svc-endpoint.yaml в репозитории GitHub книги (https://oreil.ly/TNaWH)).

Пример 8.3 ❖ Сервисы базовой платформы, пример Kubernetes

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
 - protocol: TCP
    port: 80
---
apiVersion: v1
kind: Endpoints
metadata:
  name: my-service
```

```
subsets:
 - addresses:
   - ip: 1.2.3.4
  ports:
   - port: 80
```

Это определение преобразуется в ServiceEntry из примера 8.4 (см. k8s-se.yaml в репозитории GitHub книги (https://oreil.ly/LLawn)).

Пример 8.4 ❖ Сервис Kubernetes и конечная точка объявлены как ServiceEntry

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: k8s-my-service
spec:
  hosts:
  # Имена для обращения к сервису в Kubernetes
 - my-service
 - my-service.default
 - my-service.default.svc.cluster.local
  ports:
 - number: 80
    name: http
    protocol: HTTP
resolution: STATIC
endpoints:
- address: 1.2.3.4
```

Записи ServiceEntry, созданные адаптерами платформы, не отображаются непосредственно в конфигурации Istio (т. е. их нельзя получить через istioctl get). Команда istioctl get ServiceEntry может вернуть только записи, созданные вами.

Обратите внимание, что Istio не заполняет DNS-записи на основе ServiceEntry. Это означает, что определение в примере 8.1, связывающее IP-адрес 2.2.2.2 с именем some.domain.com, не обеспечит преобразование имени some.domain.com в адрес 2.2.2.2 через DNS. Этим Istio отличается от таких систем, как Kubernetes, в которых объявление сервиса также создает DNS-запись сервиса, используемую приложением во время работы. Впрочем, существует DNS-плагин для Istio, генерирующий DNS-записи из определений ServiceEntry, который можно использовать для заполнения DNS именами сервисов Istio за пределами Kubernetes или когда нужно смоделировать вещи, не являющиеся сервисами Kubernetes.

Наконец, как показывает пример 8.5, можно использовать ServiceEntry для создания виртуальных IP-адресов (VIP), отображая IP-адрес в имя, которое можно настроить с помощью других сетевых API Istio.

Пример 8.5 ❖ Использование ServiceEntry для создания виртуального IP-адреса

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: http-server
```

```
spec:
  hosts:
 - my-tcp-service.internal
  addresses:
 - 1.2.3.4
  ports:
 - number: 975
  name: tcp
  protocol: TCP
resolution: DNS
endpoints:
- address: foo.com
```

В этом примере `1.2.3.4` объявлен как VIP с именем `my-tcp-service.internal`. Весь трафик, посылаемый на этот виртуальный адрес в порт `975` будет пересылаться на IP-адрес `foo.com`, разрешаемый через DNS. Конечно, можно настраивать конечные точки для VIP-адресов так же, как и любой другой `ServiceEntry`, обращаясь к DNS или явно настраивая набор адресов. Другие конфигурации Istio могут использовать имя `my-tcp-service.internal` для описания трафика этого сервиса. Однако опять же Istio не будет настраивать DNS-записи, внешние по отношению к сервисной сетке (или, в случае с Kubernetes, внешние по отношению к кластеру), так что `my-tcp-service.internal` для приложений будет преобразовываться в адрес `1.2.3.4`. Для этого необходимо настроить DNS, иначе приложение должно само обратиться непосредственно к `1.2.3.4`.

DestinationRule

Правила назначения (`DestinationRule`), названные немного нелогично, предназначены для настройки клиентов. Они позволяют администратору сервиса описать, как клиент в сетке должен вызывать сервис, включая:

- подмножества сервисов (например, v1 и v2);
- стратегию балансировки нагрузки, которую должен использовать клиент;
- условия признания конечных точек сервиса неисправными;
- настройки пула соединений L4 и L7;
- TLS-настройки сервера.

Подробнее о балансировке нагрузки со стороны клиента, стратегии балансировки нагрузки и обнаружении аномалий см. в разделе «*Устойчивость*» данной главы.

Настройки пула соединений

С помощью `DestinationRule` можно настроить низкоуровневые параметры пула соединений, такие как число TCP-соединений, разрешенных для каждого узла назначения, максимальное число незавершенных запросов HTTP1, HTTP2 или gRPC для каждого узла назначения и максимальное число повторных попыток для всех конечных точек. Пример 8.6 показывает определение `DestinationRule`, разрешающее максимум четыре TCP-соединения на конечную точку назначения и максимум 1000 одновременных запросов HTTP2 через эти четыре TCP-соединения.

Пример 8.6 ❖ Параметры пула соединений в DestinationRule

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: foo-default
spec:
  host: foo.default.svc.cluster.local
  trafficPolicy:
    connectionPool:
      tcp:
        maxConnections: 4
      http:
        http2MaxRequests: 1000
```

Настройки TLS

Правила `DestinationRule` могут описать, как прокси должен защищать соединение с конечной точкой назначения. Поддерживаются четыре режима:

`DISABLED`
: Отключает TLS для TCP-соединения.

`SIMPLE`
: Инициирует TLS-соединение с конечной точкой назначения.

`MUTUAL`
: Устанавливает mTLS-соединение с конечной точкой назначения.

`ISTIO_MUTUAL`
: Запрашивает, использует ли mTLS сертификаты, предоставлямые сеткой Istio.

Включение mTLS по всей сетке в конфигурации Istio – короткий путь к настройке mTLS для всех пунктов назначения в сетке. Например, можно использовать `DestinationRule` для подключения к веб-сайту HTTPS за пределами сетки, как показано в примере 8.7 (см. egress-destrule.yaml в репозитории GitHub книги (https://oreil.ly/o7aNe)).

Пример 8.7 ❖ Разрешение покидать сетку исходящему трафику в домен http://google.com

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: google.com
spec:
  host: "*.google.com"
  trafficPolicy:
    tls:
      mode: SIMPLE
```

Также можно описать подключение к другому серверу с помощью mTLS, как показано в примере 8.8.

Пример 8.8 ❖ DestinationRule, предписывающее использование mTLS

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: remote-a-ingress
spec:
  host: ingress.a.remote.cluster
  trafficPolicy:
    tls:
      mode: MUTUAL
      clientCertificate: /etc/certs/remote-cluster-a.pem
      privateKey: /etc/certs/client_private_key_cluster_a.pem
      caCertificates: /etc/certs/rootcacerts.pem
```

Можно использовать DestinationRule, как в примере 8.7, вместе с ServiceEntry кластера ingress.a.remote.cluster для маршрутизации трафика между доверенными доменами (например, отдельными кластерами) через интернет, безопасно, без VPN или других оверлейных сетей (overlay networks). Мы рассмотрим работу без VPN и другие темы в главе 13.

Подмножества

Наконец, DestinationRule позволяет разделить один сервис на подмножества с помощью меток. Также можно для каждого подмножества отдельно конфигурировать все описанные выше функции, которые позволяют настраивать DestinationRule. Например, можно разделить сервис на два подмножества на основе версии и использовать VirtualService для канареечного развертывания новой версии, постепенно переводя весь трафик на новую версию. Как показано в примере 8.9, foo имеет две версии: v1 и v2. Каждая версия сервиса foo имеет свою собственную четко определенную политику балансировки нагрузки.

Пример 8.9 ❖ Разделение трафика с использованием подмножеств

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: foo-default
spec:
  host: foo.default.svc.cluster.local
  subsets:
 - name: v1
    labels:
     version: v1
    trafficPolicy:
      loadBalancer:
        simple: ROUND_ROBIN
 - name: v2
    labels:
     version: v2
    trafficPolicy:
     loadBalancer:
       simple: LEAST_CONN
```

Более подробно рассмотрим VirtualService в следующем разделе.

VirtualService

VirtualService описывает направление именованного трафика нескольким получателям, как показано в примере 8.10.

Пример 8.10 ❖ Простой VirtualService

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-identity
spec:
  hosts:
 - foo.default.svc.cluster.local
  http:
 - route:
   - destination:
       host: foo.default.svc.cluster.local
```

VirtualService в примере 8.10 перенаправляет трафик, адресованный кластеру foo.default.svc.local, получателю с именем foo.default.svc.cluster.local. Pilot неявно генерирует VirtualService (как в этом примере) для сопряжения с ServiceEntry каждого сервиса.

Конечно, VirtualService позволяет решать гораздо более интересные задачи. Например, определить для сервиса конечные точки HTTP и поручить Envoy выдавать ошибку 404 (на стороне клиента) при обнаружении неверных путей без вызова удаленного сервера, как показано в примере 8.11.

Пример 8.11 ❖ VirtualService с сопоставлением путей

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-apiserver
spec:
  hosts:
 - foo.default.svc.cluster.local
  http:
 - match:
 - uri:
     prefix: "/api"
   route:
   - destination:
       host: apiserver.foo.svc.cluster.local
```

Запросы, отправленные на адрес foo.default.svc.cluster.local/api/..., будут переданы набору API-серверов на целевом узле apiserver.foo.svc.cluster.local, а попытка обратиться к любому другому URI приведет к тому, что Envoy, не найдя адресата, вернет приложению ошибку 404. Вот почему Pilot создает неявный VirtualService для каждого ServiceEntry. Хотя адрес для остального трафика явно

не задан, любое несовпадение с путями в DestinationRule приведет к ошибке 404, что можно считать аналогом адреса для необслуживаемого трафика.

VirtualService можно использовать для работы с довольно специфическими типами трафика и направлять в различные пункты назначения. Например, VirtualService может сопоставлять запросы по значениям заголовков; порту, к которому пытается подключиться вызывающая сторона; или меткам рабочей нагрузки клиента (например, меткам клиентского пода в Kubernetes) и отправлять только совпадающий трафик на другой адрес (например, новой версии сервиса). Подробнее эти случаи будут рассмотрены в разделе «*Управление трафиком и маршрутизация*» данной главы. Простым примером может служить отправка части трафика в новую версию сервиса (см. пример 8.12). Такой подход позволяет быстро откатиться в случае неудачного развертывания.

Пример 8.12 ❖ Разделение трафика между подмножествами VirtualService

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-apiserver
spec:
  hosts:
 - foo.default.svc.cluster.local
  http:
 - match:
   - uri:
      prefix: "/api"
    route:
   - destination:
      host: apiserver.foo.svc.cluster.local
     subset: v1
   weight: 90
    destination:
      host: apiserver.foo.svc.cluster.local
    subset: v2
   weight: 10
```

Важно отметить, что в рамках VirtualService условия совпадения проверяются во время работы в порядке их появления. Это означает, что первыми должны следовать более строгие условия, а менее строгие – позднее. Для безопасности должен быть предусмотрен маршрут «по умолчанию» для трафика, не соответствующего ни одному условию. Потому что, опять же, запрос, не соответствующий какому-либо условию в VirtualService, приведет к ошибке 404 на стороне отправителя (или «соединение отклоненено» для протоколов, отличных от HTTP).

Хосты

Мы говорим, что VirtualService *объявляет* имя: одно и то же имя хоста может появиться не более чем в одном определении VirtualService, при этом VirtualService может объявить несколько имен хостов. Это может вызвать проблемы, когда одно имя, например apis.foo.com, используется для размещения нескольких сервисов, а их маршрутизация выполняется по пути – например, apis.foo.com/

bars или apis.foo.com/basz, поскольку многие команды/сотрудники вынуждены будут редактировать одно и то же определение VirtualService apis.foo.com. Одно из решений этой проблемы состоит в использовании набора многоуровневых VirtualService. VirtualService верхнего уровня разделяет запросы на логические сервисы по префиксу пути и является ресурсом, общим для каждой команды (подобно ресурсу Kubernetes Ingress). Затем VirtualService для каждого логического сервиса описывает трафик для своего блока запросов. Можно повторно применять эту схему, делегируя управление меньшими и меньшими частями трафика.

Рассмотрим общий VirtualService с бизнес-логикой для нескольких команд, как в примере 8.13.

Пример 8.13 ❖ Монолитное определение VirtualService

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-apiserver
spec:
  hosts:
 - apis.foo.com
  http:
 - match:
   - uri:
       prefix: "/bars/newMethod"
    route:
   - destination:
       host: bar.foo.svc.cluster.local
       subset: v2
 - match:
   - uri:
      prefix: "/bars"
    route:
   - destination:
       host: bar.foo.svc.cluster.local
       subset: v1
 - match:
   - uri:
       prefix: "/bazs/legacy/rest/path"
    route:
   - destination:
       host: monolith.legacy.svc.cluster.remote
    retries:
      attempts: 3
      perTryTimeout: 2s
 - match:
   - uri:
       prefix: "/bazs"
  route:
 - destination:
     host: baz.foo.svc.cluster.local
```

Это определение VirtualService можно разделить на отдельные VirtualService (показанные в примере 8.14), принадлежащие соответствующим командам.

Пример 8.14 ❖ VirtualService разделены на независимые определения для облегчения независимого управления изменениями

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-svc-shared
spec:
  hosts:
 - apis.foo.com
  http:
 - match:
   - uri:
       prefix: "/bars"
    route:
   - destination:
       host: bar.foo.svc.cluster.local
 - match:
    uri:
      prefix: "/bazs"
    route:
   - destination:
       host: baz.foo.svc.cluster.local
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-bars-svc
spec:
  hosts:
 - bar.foo.svc.cluster.local
  http:
 - match:
   - uri:
       prefix: "/bars/newMethod"
    route:
   - destination:
       host: bar.foo.svc.cluster.local
       subset: v2
  route:
 - destination:
     host: bar.foo.svc.cluster.local
     subset: v1
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  Name: foo-bazs-svc
spec:
  hosts:
```

```
- baz.foo.svc.cluster.local
 http:
- match:
  - uri:
      prefix: "/bazs/legacy/rest/path"
   route:
  - destination:
      host: monolith.legacy.svc.cluster.remote
   retries:
     attempts: 3
     perTryTimeout: 2s
   route:
  - destination:
      host: baz.foo.svc.cluster.local
```

Как описано в разделе «*Развязка на уровне 5*» главы 1, сервисные сетки значительно уменьшают зависимость разных групп друг от друга (разработчиков, администраторов и т. д.) и как таковые являются ключевым инструментом повышения скорости работы групп, сокращения рисков, с которыми сталкиваются группы при управлении изменениями, уточнения границ ответственности между ролями и облегчения отчетности по конкретным аспектам развертывания сервисов.

В примере 8.14 специально показано, как можно подойти к уточнению границ ответственности и убрать зависимости между группами в рамках конфигурации сервисной сетки на L5.

Наконец, `VirtualService` может заявить свои права на несколько хостов, описав их с использованием символов подстановки. Другими словами, `VirtualService` может претендовать на хост типа `*.com`. При выборе конфигурации всегда будет применяться наиболее специфический хост: для запроса на `baz.foo.com` используется `VirtualService` для `baz.foo.com`, а `VirtualService` – для `*.foo.com` и `*.com` игнорируются. Обратите внимание, однако, что ни один `VirtualService` не может претендовать на `"*"` (любой хост).

Gateway

Ресурс `Gateway` предназначен для экспортирования имен через границы доверия. Предположим, что есть сервис `webserver.foo.svc.cluster.local`, развернутый в сетке, обслуживающей сайт `foo.com`. Вы можете открыть публичный доступ к веб-серверу из интернета, использовав Gateway для отображения внутреннего имени `weberver.foo.svc.cluster.local` в публичное имя `foo.com`. Необходимо также указать порт и протокол для доступа к этому имени, как показано в примере 8.15.

Пример 8.15 ❖ Простое определение шлюза, которое открывает HTTP/80

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: foo-com-gateway
spec:
  selector:
```

```
    app: gateway-workloads
  servers:
 - hosts:
   - foo.com
  port:
    number: 80
    name: http
    protocol: HTTP
```

Однако для безопасной передачи недостаточно простого отображения имен. Gateway должен быть в состоянии доказать вызывающим абонентам, что он является истинным владельцем имени. Это можно сделать, настроив Gateway на обслуживание сертификата http://foo.com, как показано в примере 8.16 (см. gwhttps.yaml в репозитории GitHub книги (https://oreil.ly/DLkkP)).

Пример 8.16 ❖ Gateway обслуживает сертификат foo.com

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: foo-com-gateway
spec:
  selector:
    app: gateway-workloads
  servers:
 - hosts:
   - foo.com
    port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE # Разрешает HTTPS на этом порту
      serverCertificate: /etc/certs/foo-com-public.pem
      privateKey: /etc/certs/foo-com-privatekey.pem
```

В примере 8.16 foo-com-public.pem и foo-com-privatekey.pem являются долгоживущими сертификатами для foo.com, которые можно получить в центре сертификации, таком как *Let's Encrypt*. К сожалению, сегодня Istio не работает с такими типами сертификатов, поэтому любые сертификаты, которые Gateway должен обслуживать, необходимо монтировать в файловую систему рабочей нагрузки. Также необходимо соответствующим образом изменить порт и протокол. Если необходимо, можно и далее обслуживать foo.com по HTTP через порт 80 в дополнение к HTTPS/443, как показано в примере 8.17.

Пример 8.17 ❖ Gateway, обслуживающий одновременно протоколы HTTP/80 и HTTPS/443

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: foo-com-gateway
spec:
  selector:
```

```
    app: gateway-workloads
  servers:
 - hosts:
 - foo.com
    port:
      number: 80
      name: http
      protocol: HTTP
 - hosts:
   - foo.com
    port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE # Разрешает HTTPS на этом порту
      serverCertificate: /etc/certs/foo-com-public.pem
      privateKey: /etc/certs/foo-com-privatekey.pem
```

Согласно лучшим практикам безопасности, рекомендуется настроить Gateway на инкапсуляцию входящего трафика HTTP в протокол HTTPS, как показано в примере 8.18 (см. gw-https-upgrade.yaml в репозитории GitHub книги (https://oreil.ly/2lf6g)).

Пример 8.18 ❖ Gateway сконфигурирован для преобразования входящих HTTP/80 соединений в защищенные соединения HTTPS/443

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: foo-com-gateway
spec:
  selector:
    app: gateway-workloads
  servers:
 -  hosts:
   - foo.com
    port:
      number: 80
      name: http
      protocol: HTTP
    tls:
      httpsRedirect: true # Отправляет 301 редирект для http-запросов
 - hosts:
   - foo.com
    port:
      number: 443
      name: https
      protocol: HTTPS
    tls:
      mode: SIMPLE # Разрешает HTTPS на этом порту
      serverCertificate: /etc/certs/foo-com-public.pem
      privateKey: /etc/certs/foo-com-privatekey.pem
```

Примеры демонстрируют обычно используемые HTTP(S) и порты 80 и 443; однако Gateway может открыть доступ для любого протокола и к любому порту. Когда Istio контролирует установку Gateway, будут прослушиваться все порты, перечисленные в его конфигурации.

До сих пор ни один из Gateway не отображал foo.com ни в один сервис в сетке! Для этого нужно *привязать* VirtualService к Gateway, как показано в примере 8.19 (см. foo-vs.yaml в репозитории GitHub книги (https://oreil.ly/aRbTM)).

Пример 8.19 ❖ Привязка VirtualService foo.com к Gateway

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-com-virtual-service
spec:
  hosts:
 - foo.com
  gateways:
 - foo-com-gateway
  http:
 - route:
 - destination:
     host: webserver.foo.svc.cluster.local
```

Правила привязки VirtualService к Gateway описаны в разделе «*Привязка VirtualService к Gateway*», но остается важный момент: Gateway настраивает поведение L4, а не L7. Здесь подразумевается, что Gateway описывает порты и их протоколы, а также имена (и подтверждение имен посредством сертификатов), которые будут обслуживать эти порты. Но VirtualService описывает поведение L7. Поведение L7 состоит в сопоставлении некоторых имен (т. е. foo.com) с различными приложениями и рабочими нагрузками.

Одной из целей разработки Istio было отделение поведения L4 от L7. Это позволяет создать единое определение Gateway, которое разные команды смогут использовать совместно, как показано в примере 8.20 (см. gw-to-vses.yaml в репозитории GitHub книги (https://oreil.ly/n3YT0)).

Пример 8.20 ❖ Один Gateway, используемый несколькими VirtualService

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: foo-com-gateway
spec:
  selector:
    app: gateway-workloads
  servers:
 - hosts:
   - *.foo.com
    port:
      number: 80
      name: http
      protocol: http
```

```
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-com-virtual-service
spec:
  hosts:
 - api.foo.com
  gateways:
 - foo-com-gateway
  http:
 - route:
 - destination:
      host: api.foo.svc.cluster.local
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-com-virtual-service
spec:
  hosts:
 - www.foo.com
  gateways:
 - foo-com-gateway
  http:
 - route:
   - destination:
       host: webserver.foo.svc.cluster.local
```

Более важно то, что разделение поведения L4 и L7 позволяет использовать `Gateway` для моделирования сетевых интерфейсов в Istio (например, сетевые устройства или неплоские сети L3). Наконец, можно использовать `Gateway` для построения mTLS-туннелей между частями сетки, расположенными в разных сетях L3. Например, можно их применять для создания защищенных соединений между разными сетками Istio, развернутыми в разных зонах доступности облачных провайдеров, через общедоступный интернет без использования VPN.

Наконец, `Gateway` можно применять для моделирования произвольных сетевых интерфейсов, независимо от того, находится этот интерфейс под контролем Istio или нет. Таким образом, даже если сетевой интерфейс управляется сеткой Istio, поведение сетевого сервиса за `Gateway`, представляющим интерфейс, может находиться под контролем Istio, а может и нет. Например, если `Gateway` используется в облаке в роли наружного балансировщика нагрузки, конфигурация Istio не сможет влиять на решения, принимаемые этим балансировщиком. Рабочие нагрузки, принадлежащие `Gateway`, описываются полем `selector` объекта `Gateway`. Рабочие нагрузки с метками, соответствующими выбранному селектору, рассматриваются в Istio как объекты `Gateway`. Когда Istio контролирует реализацию `Gateway` (т. е. когда в роли шлюза выступает Envoy), можно *привязать* `VirtualService` к `Gateway`, чтобы использовать возможности `VirtualService` в точках входа и выхода нашего кластера.

Привязка VirtualService к Gateway

Мы говорим, что VirtualService *привязывается* к Gateway, если верно следующее:

- VirtualService включает имя Gateway в своем поле gateways;
- по крайней мере один хост, *заявленный* VirtualService, *экспортируется объектом* Gateway.

Хосты в конфигурации Gateway объявляются так же, как в VirtualService, но с одним важным отличием. Gateway не объявляет имена хостов, как это делают VirtualService. Вместо этого Gateway *экспортирует* имя, позволяя VirtualService настроить передачу трафика к этому имени путем привязки к Gateway. Например, любое количество Gateway может содержать имя foo.com, но только один VirtualService должен настроить трафик для него через все Gateway. В поле hosts в определении Gateway допускается использовать групповые имена хостов так же, как в VirtualService, но, кроме того, в Gateway *допускается* использовать групповое имя "*".

Начнем с небольшого обзора двух Gateway, отличающихся конфигурацией хостов: foo-gateway и wildcard-Gateway (см. gw-examples.yaml в репозитории GitHub книги (https://oreil.ly/AuyOE)). Сначала пример foo-gateway:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: foo-gateway
spec:
  selector:
    app: my-gateway-impl
  servers:
 - hosts:
   - foo.com
    port:
      number: 80
      name: http
      protocol: HTTP
```

И теперь пример wildcard-gateway:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: wildcard-gateway
spec:
  selector:
    app: my-gateway-impl
  servers:
 - hosts:
   - *.com
    port:
      number: 80
      name: http
      protocol: HTTP
```

Теперь рассмотрим, как следующие VirtualService привязываются (или не привязываются, в зависимости от обстоятельств) к этим Gateway (см. vs-examples.yaml в репозитории GitHub книги (https://oreil.ly/eSWbt)).

Следующий пример привязывается к *foo-gateway*, так как имя `Gateway` и имя шлюза в `VirtualService` совпадают, и потому что `VirtualService` объявляет хост *foo.com*, открытый через *foo-gateway*. Поэтому запросы к *foo.com*, полученные этим `Gateway` на порт 80, будут перенаправлены на порт 7777 сервиса *foo* в пространстве имен *default*.

Привязка к шлюзу wildcard-gateway не происходит; хосты совпадают, но `VirtualService` не перечисляет wildcard-gateway в своем списке шлюзов:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-default
spec:
  hosts:
 - foo.com
  gateways:
 - foo-gateway
  http:
 - route:
   - destination:
       host: foo.default.svc.cluster.local
```

Следующий пример привязывается к *foo-gateway*, потому что имя `Gateway` и имя шлюза в VirtualService совпадают и потому что `VirtualService` объявляет хост *foo.com*, открытый через *foo-gateway*. Вызывающим абонентам `Gateway` видно только имя *foo.com*, хотя `VirtualService` объявляет имя *foo.super.secret.internal.name*.

Привязка к шлюзу wildcard-gateway не происходит; хосты совпадают, но `VirtualService` не перечисляет wildcard-gateway в своем списке шлюзов:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-default
spec:
  hosts:
 - foo.com
 - foo.super.secret.internal.name
  gateways:
 - foo-gateway
  http:
 - route:
   - destination:
       host: foo.default.svc.cluster.local
```

Следующий пример не привязывается ни к одному `Gateway`: хотя `VirtualService` и перечисляет оба шлюза, имя хоста *foo.super.secret.Internal.name*, объявленное в `VirtualService`, не экспортируется ни одним из шлюзов, поэтому они не будут принимать запросы к этим именам:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-internal
```

```
spec:
  hosts:
 - foo.super.secret.internal.name
  gateways:
 - foo-gateway
 - wildcard-gateway
  http:
 - route:
   - destination:
       host: foo.default.svc.cluster.local
```

Последний пример привязывается к *foo-gateway*, потому что имя `Gateway` и имя шлюза в VirtualService совпадают и потому что `VirtualService` объявляет хост *foo.com*, открытый через *foo-gateway*.

Также он привязывается к wildcard-gateway, потому что совпадает имя `Gateway` и имя шлюза в VirtualService `VirtualService` и потому что `VirtualService` объявляет хост *foo.com*, открытый через wildcard-gateway (*foo.com* соответствует групповому имени `"*.com"`):

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-internal
spec:
  hosts:
 - foo.com
  gateways:
 - foo-gateway
 - wildcard-gateway
  http:
 - route:
   - destination:
       host: foo.default.svc.cluster.local
```

Шлюз сетки

В каждом развернутом экземпляре Istio есть специальный, неявный `Gateway`, называющийся *шлюзом сетки*. Шлюз такого типа владеет рабочими нагрузками, представленными всеми прокси в сетке, и открывает доступ ко всем портам хоста с групповым именем (*). Когда `VirtualService` не привязывается ни к одному `Gateway`, происходит автоматическая привязка к шлюзу сетки, т. е. ко всем прокси в сетке. `VirtualService` всегда привязывается *либо* к шлюзу сетки, *либо* к шлюзам, указанным в его поле `gateways`. Обычная опасность аварийного сбоя при использовании `VirtualService` возникает, если попытаться обновить `VirtualService`, используемый внутри сетки, для привязки к определенному `Gateway`, заменяющему шлюз сетки. При передаче этого ресурса его конфигурация больше не пересылается в прокси сервисов, что приводит к ошибкам. Для такого рода обновления следует специально включить *шлюз сетки* в список шлюзов, к которым необходимо привязываться.

Управление трафиком и маршрутизация

Вышеописанный API можно использовать самыми разными способами, чтобы влиять на поток трафика в системе. В этом разделе рассмотрим некоторые наиболее распространенные случаи применения, например использование `VirtualService` для принятия решений о маршрутизации на основе:

- атрибутов запроса, таких как URI;
- заголовков;
- схемы запроса;
- целевого порта запроса.

Также можно использовать `VirtualService` для реализации *канареечных* и *сине-зеленых* стратегий развертывания сервисов.

Как безопасно развернуть новые версии сервиса?

Kubernetes будет распределять 50/50 трафика между старой и новой версиями вашего сервиса. Правила назначения для прокси в сетке Istio позволяют администратору сервиса осуществлять гранулированное управление трафиком (или использовать переменные окружения в виртуальной машине).

Istio игнорирует сервисы Kubernetes, поскольку они могут выполнять только циклическую (*round-robin*) балансировку нагрузки. Когда приложение пытается это сделать, оно находит две подгруппы сервиса, идентифицируемые по ярлыкам, которые можно использовать для выделения сервиса. Есть набор переменных окружения, которые посылает Istio.

Для Envoy безразлично, работает ли одна подгруппа в контейнерах, а другая в виртуальных машинах. Управление трафиком на основе содержимого осуществляется в Envoy локально.

Маршрутизация с метаданными запроса

Одной из самых мощных функций Istio является возможность маршрутизации трафика на основе метаданных запроса, таких как URI запроса, его заголовки, IP-адреса источника/назначения и других. Единственным ключевым ограничением является то, что Istio не будет выполнять маршрутизацию, основанную на *содержимом* запроса.

Раздел «*VirtualService*» ранее в этой главе подробно описывал маршрутизацию на основе префиксов URI. Можно выполнять аналогичную маршрутизацию на точных совпадениях URI и регулярных выражениях, как показано в примере 8.21.

Пример 8.21 ❖ Маршрутизация в VirtualService с сопоставлением путей

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  Name: foo-bars-svc
spec:
  hosts:
 - bar.foo.svc.cluster.local
  http:
 - match:
```

```
    - uri:
        exact: "/assets/static/style.css"
      route:
    - destination:
        host: webserver.frontend.svc.cluster.local
  - match:
    - uri:
        # Совпадение запросов вида “/foo/132:myCustomMethod”
        regex: "/foo/\\d+:myCustomMethod"
      route:
    - destination:
        host: bar.foo.svc.cluster.local
        subset: v3
  - route:
    - destination:
        host: bar.foo.svc.cluster.local
        subset: v2
```

Также можно маршрутизировать, используя заголовки или значения cookie, как показано в примере 8.22.

Пример 8.22 ❖ Перенаправление запросов на основе наличия значения в cookie-файле

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  Name: dev-webserver
spec:
  hosts:
  - webserver.company.com
  http:
  - match:
  - headers:
      cookie:
        environment: "dev"
    route:
    - destination:
        host: webserver.dev.svc.cluster.local
  - route:
    - destination:
        host: webserver.prod.svc.cluster.local
```

Конечно, Istio также поддерживает маршрутизацию запросов для TCP-сервисов, используя L4 метаданные запросов, такие как целевая подсеть и целевой порт (см. пример 8.23). Для сервисов TLS TCP для маршрутизации можно использовать SNI так же, как и HTTP-заголовок Host.

Пример 8.23 ❖ Маршрутизация запросов на основе информации L4

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  Name: dev-api-server
spec:
```

```
  hosts:
- api.company.com
  tcp:
- match:
  - port: 9090
     destinationSubnets:
    - 10.128.0.0/16
   route:
  - destination:
       host: database.test.svc.cluster.local
- match:
  - port: 9090
   route:
  - destination:
       host: database.prod.svc.cluster.local
  tls:
- match:
  - sniHosts:
    - example.api.company.com
   route:
  - destination:
       host: example.prod.svc.cluster.local
```

Полную информацию обо всех доступных способах сопоставления и синтаксисе смотрите на сайте Istio (https://istio.io/).

Сине-зеленое развертывание

В методике сине-зеленого развертывания бок о бок устанавливаются две версии приложения, старая и новая, и пользовательский трафик переключается со старого набора на новый. Это позволяет быстро вернуться к предыдущей рабочей версии, если что-то пойдет не так, – нужно лишь переключить пользовательский трафик с нового набора на старый (в противоположность стратегии скользящего обновления, в которой для отката к предыдущей версии сначала нужно заново развернуть двоичный файл предыдущей версии).

Сетевые API Istio позволяют легко выполнять *сине-зеленые* развертывания. Объявляются две *подгруппы* сервиса с помощью `DestinationRule`, а затем `VirtualService` используется для направления трафика к одной или другой подгруппе, как показано в примере 8.24.

Вместо использования *меток "синий" и "зеленый"* в `DestinationRule` мы будем использовать номера версий. Разработчикам это легче понять (потому что речь идет о контролируемых ими частях сервиса), и они меньше подвержены ошибкам (избегая заминки типа «Эй, я готов к развертыванию. Сейчас работает синий или зеленый?»). Подобная формулировка облегчает и переход к другим стратегиям развертывания, таким как канареечное, или пробное, развертывание.

Пример 8.24 ❖ Определение подмножеств с помощью функции DestinationRule

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: foo-default
```

```
spec:
  host: foo.default.svc.cluster.local
  subsets:
 - name: v1
      labels:
        version: v1
 - name: v2
      labels:
        version: v2
```

Теперь можно написать `VirtualService`, направляющий весь трафик в кластере, предназначенный для данного сервиса, в единственное подмножество сервиса, как показано в примере 8.25.

Пример 8.25 ❖ Направление запросов по метке подмножества v1

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-blue-green-virtual-service
spec:
  hosts:
 - foo.default.svc.cluster.local
  http:
 - route:
   - destination:
       host: foo.default.svc.cluster.local
       subset: v1
```

Чтобы перейти к другому набору, достаточно обновить `VirtualService`, направив трафик в целевое подмножество v2, как показано в примере 8.26.

Пример 8.26 ❖ Направление запросов по метке подмножества v2

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-blue-green-virtual-service
spec:
  hosts:
 - foo.default.svc.cluster.local
  http:
 - route:
   - destination:
       host: foo.default.svc.cluster.local
       subset: v2
```

Конечно, можно комбинировать этот подход с `Gateway` для выполнения *сине-зеленых* развертываний для пользователей, использующих сервис через `Gateway`, в дополнение к сервисам в сетке.

Канареечные развертывания

Канареечное развертывание – это практика отправки небольшой части трафика в новые рабочие нагрузки и постепенное его увеличение, пока весь трафик не

будет обслуживаться новыми рабочими нагрузками. Цель состоит в проверке работоспособности новой рабочей нагрузки (запущена, работает и не возвращает ошибок) перед отправкой ей всего трафика. Это похоже на *сине-зеленое* развертывание, поскольку позволяет быстро вернуться к известным рабочим нагрузкам, только лучше, потому что в новые рабочие нагрузки посылается не весь трафик, а только часть. В целом это сокращает *бюджет ошибок* (метрика, выделяющая определенное число перерывов в обслуживании, допустимое в течение определенного периода времени), который можно потратить на выполнение развертывания.

Канареечное развертывание, как правило, требует ресурсов для обновления. Настоящее *сине-зеленое* развертывание требует вдвое больше ресурсов, чем стандартное (для *полного* синего и *полного* зеленого развертываний). Канареечное развертывание можно комбинировать со стратегиями бинарного развертывания на месте, чтобы обеспечить безопасность отката *сине-зеленого* развертывания , оно позволяет ограничиться постоянным количеством дополнительных ресурсов (резервных мощностей для планирования небольшого числа дополнительных рабочих нагрузок).

Выполнить канареечное развертывание новой рабочей нагрузки можно разными способами. Можно определить полный набор сопоставлений, как описывается в разделе «*Маршрутизация с метаданными запроса*», для отправки небольших объемов трафика новому сервису. Однако самый простой способ – разделить трафик по процентам. Для начала можно отправить 5 % трафика в новую версию и, постепенно продвигая новые конфигурации `VirtualService`, увеличить трафик в новую версию до 100 %, как показано в примере 8.27 (см. canary-shift.yaml в репозитории GitHub книги (https://oreil.ly/fwGh-)).

Пример 8.27 ❖ Канареечное развертывание с использованием процентного переноса трафика

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-canary-virtual-service
spec:
  hosts:
 - foo.default.svc.cluster.local
  tcp:
 - route:
   - destination:
       host: foo.default.svc.cluster.local
       subset: v2
     weight: 5
   - destination:
       host: foo.default.svc.cluster.local
       subset: v1
     weight: 95
```

Другой распространенной схемой является канареечное развертывание новой системы для доверенных тестировщиков, таких как команда обслуживания или несколько клиентов, выбравших экспериментальные функции. Мож-

но использовать Istio для установки cookie *trusted-tester*, который, например, во время маршрутизации может перенаправлять запросы в этом конкретном сеансе на разные рабочие нагрузки, в отличие от рабочих нагрузок, обслуживаемых запросами без этого cookie, как показано в примере 8.28 (см. файл canary-cookie.yaml в репозитории GitHub книги (https://oreil.ly/8mCg9)).

Пример 8.28 ❖ Канареечное развертывание с использованием cookie-файла

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-canary-virtual-service
spec:
  hosts:
 - foo.default.svc.cluster.local
  http:
 - match:
   - headers:
       cookie:
       trusted-tester: "true"
  route:
 - destination:
     host: foo.default.svc.cluster.local
     subset: test
 - route:
   - destination:
       host: foo.default.svc.cluster.local
       subset: v1
```

Конечно, будьте осторожны при использовании значений, предоставляемых вызывающим абонентом (например, cookies), для маршрутизации: в идеале все сервисы кластера должны выполнять аутентификацию и авторизацию всех запросов. Это гарантирует, что даже если вызывающий абонент подделает данные, чтобы изменить поведение маршрутизации, он не сможет получить доступ к данным, недоступным иначе (и на самом деле, внедрение аутентификации и авторизации через Istio является мощным способом убедиться, что все сервисы кластера делают это правильно).

Устойчивость

Устойчивая система – это система, поддерживающая хорошую производительность для пользователей (т. е. остающаяся в пределах соглашений об уровне обслуживания), одновременно справляющаяся со сбоями в нижестоящих системах, от которых она зависит. Istio предоставляет множество функций, помогающих создавать более отказоустойчивые приложения; наиболее важными из них являются балансировка нагрузки со стороны клиента, разрыв цепи при обнаружении аномалий, автоматические повторные попытки и тайм-аут запроса. Istio также предоставляет инструменты для имитации неисправностей в приложениях, позволяющие создавать программные, воспроизводимые тесты отказоустойчивости системы.

Стратегия балансировки нагрузки

Балансировка нагрузки со стороны клиента является невероятно ценным инструментом для построения отказоустойчивых систем. Позволяя клиентам напрямую взаимодействовать с серверами без использования обратных прокси, мы устраняем точки сбоя, сохраняя при этом правильное поведение системы. Кроме того, опираясь на ответы серверов, клиенты могут корректировать свое поведение, например прекращать отправку запросов конечным точкам, возвращающим больше ошибок, чем другие конечные точки того же сервиса (подробнее об этой функции, обнаружении аномалий, см. в следующем разделе). `DestinationRule` позволяет определить стратегию балансировки нагрузки, используемую клиентами для выбора экземпляров сервисов. Пример 8.29 показывает, как настроить клиентов на использование простой стратегии циклической балансировки нагрузки (см. round-robin.yaml в репозитории GitHub книги (https://oreil.ly/cQC7Y)).

Пример 8.29 ❖ Простая конфигурация циклической балансировки нагрузки

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
  metadata:
name: foo-default
  spec:
  host: foo.default.svc.cluster.local
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
```

Это правило `DestinationRule` циклически направляет трафик конечным точкам сервиса `foo.default.svc.cluster.local`, а `ServiceEntry` определяет, что это за конечные точки (или как их обнаружить во время работы; например, через DNS). Важно отметить, что `DestinationRule` применяется только к хостам в реестре сервисов Istio. Если `ServiceEntry` не существует для хоста, то `DestinationRule` игнорируется.

Поддерживаются также более сложные стратегии балансировки нагрузки, такие как последовательная балансировка на основе хеша. Следующее правило `DestinationRule` настраивает балансировку нагрузки на основе хеша IP-адреса вызывающего абонента (также можно использовать HTTP-заголовки и cookies), как показано в примере 8.30.

Пример 8.30 ❖ Создание закрепленных сеансов на основе IP-адреса источника

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: foo-default
spec:
  host: foo.default.svc.cluster.local
  trafficPolicy:
    loadBalancer:
      consistentHash:
        useSourceIp: true
```

Обнаружение аномалий

Прерывание цепи – это схема защиты вызовов (например, сетевых вызовов удаленного сервиса) позади *прерывателя цепи* (*circuit breaker*). Если защищенный вызов возвращает слишком много ошибок, «щелчок» прерывателя возвращает ошибки вызывающему абоненту без выполнения самого защищенного вызова. Это применимо для смягчения последствий нескольких классов отказов, включая каскадные. В балансировке нагрузки *под отключением конечной точки понимается ее удаление* из «активного» набора балансировщика нагрузки, чтобы в течение определенного периода времени трафик на нее не посылался. Отключение – один из методов, который можно использовать для реализации схемы прерывателя цепи.

Обнаружение аномалий – это способ отключения конечных точек, возвращающих плохие ответы. Можно определить, когда отдельная конечная точка ведет себя аномально по сравнению с остальными в «активном» наборе балансировщика нагрузки (т. е. возвращает больше ошибок, чем другие конечные точки сервиса), и удалить плохую конечную точку из «активного» набора балансировщика нагрузки, как показано ниже:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: foo-default
spec:
  host: foo.default.svc.cluster.local
  trafficPolicy:
    outlierDetection:
      consecutiveErrors: 5
      interval: 1m
      baseEjectionTime: 3m
```

Здесь `DestinationRule` настраивает прокси на исключение из набора балансировки нагрузки любой конечной точки, давшей подряд пять ошибок, на период не менее трех минут. Каждую минуту прокси сканирует набор всех конечных точек, чтобы решить, нужно ли исключать какие-то конечные точки и можно ли вернуть исключенные точки обратно в набор для балансировки. Помните, что обнаружение аномалий производится для каждого клиента отдельно, так как любой сервер может возвращать плохие результаты только определенному клиенту (например, если связь с этим клиентом нарушилась, а с другими нет).

Повторные попытки

В любой системе случаются кратковременные сбои: переполнение сетевых буферов; выключение серверов с потерей запросов; отказ нижестоящей системы и т. д. Мы используем повторные попытки – отправку одного и того же запроса другой конечной точке того же сервиса – для смягчения воздействия переходных сбоев. Однако неэффективная политика повторных попыток часто оказывается причиной вторичных отказов: «Что-то пошло не так, и повторные попытки клиента только ухудшили ситуацию», – это обычный рефрен (напев).

Часто это связано с тем, что повторные попытки жестко закодированы в приложениях (например, как цикл `for` вокруг сетевого вызова), и поэтому их трудно изменить. Istio дает возможность глобально настраивать повторные попытки для всех сервисов в сетке. Более того, она позволяет управлять стратегиями повторных попыток во время работы через конфигурацию, так что можно изменять поведение клиента на лету, как показано ниже:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-default
spec:
  hosts:
 - foo.com
  gateways:
 - foo-gateway
  http:
 - route:
   - destination:
       host: foo.default.svc.cluster.local
    retries:
      attempts: 3
      perTryTimeout: 500ms
```

Политика повторных попыток, определенная в `VirtualService`, согласуется с настройками пула соединений, определенными в `DestinationRule` получателя, чтобы контролировать общее количество одновременных незавершенных попыток получить ответ от этого получателя. Подробнее об этом было сказано в разделе «*DestinationRule*» данной главы.

Тайм-ауты

Тайм-ауты важны для построения систем с устойчивым поведением. Ограничивая сроки, можно прерывать обработку запросов, занимающих слишком много времени, и освободить ресурсы сервера. Также можно гораздо точнее контролировать конечную задержку, зная, как долго ожидать подготовки ответа для клиента. Тайм-аут можно присвоить любому HTTP-маршруту в `VirtualService`, как показано в примере 8.31 (см. timeout.yaml в репозитории GitHub книги (https://oreil.ly/cQC7Y)).

Пример 8.31 ❖ Простой тайм-аут для VirtualService

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-default
spec:
  hosts:
 - foo.com
  gateways:
 - foo-gateway
```

```
  http:
 - route:
   - destination:
       host: foo.default.svc.cluster.local
    timeout: 1s
```

Если задано число повторных попыток, тайм-аут определяет общее время ожидания результата от сервера. Пример 8.32 демонстрирует конфигурацию тайм-аут-на-попытку (*per-try-timeout*), контролирующую тайм-аут каждой отдельной попытки (см. per-try-timeout.yaml в репозитории GitHub книги (https://oreil.ly/gzixs)).

Пример 8.32 ❖ VirtualService со сконфигурированным тайм-аутом повторных попыток

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-default
spec:
  hosts:
 - foo.com
  gateways:
 - foo-gateway
  http:
 - route:
   - destination:
       host: foo.default.svc.cluster.local
    timeout: 2s
    retries:
      attempts: 3
      perTryTimeout: 500ms
```

`VirtualService` в примере 8.32 настраивает клиента на максимальное ожидание в 2 с, повторяя попытку трижды, с тайм-аутом 500 мс каждый. Общее время ожидания указано чуть больше, чтобы учесть случайные задержки между повторными попытками.

Имитация ошибок

Имитация ошибок – это невероятно мощный способ тестирования и построения надежных распределенных приложений. Такие компании, как Netflix, довели его до идеала, введя термин «инжиниринг хаоса» для описания практики имитации неисправностей в действующих производственных системах с целью обеспечения надежности и устойчивости систем к сбоям окружающей среды.

Istio позволяет настраивать имитацию ошибок для HTTP-трафика, вводить произвольные задержки или возвращать определенные коды ответа (например, 500) для определенного процента трафика:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
```

```
metadata:
  name: foo-default
spec:
  hosts:
 - foo.default.svc.cluster.local
  http:
 - route:
   - destination:
       host: foo.default.svc.cluster.local
    fault:
      delay:
        fixedDelay: 5s
        percentage: 100
```

Например, в предыдущем примере `VirtualService` вводит 5-секундную задержку для всего трафика, вызывающего сервис `foo`. Это отличный способ надежно протестировать такие моменты, как поведение пользовательского интерфейса при неустойчивой связи с удаленными серверами. Этот способ также важен для проверки обработки тайм-аутов в приложениях.

При тестировании еще может пригодиться возможность отвечать на запросы клиентов определенными кодами, такими как 429 или 500. Например, иногда сложно программно протестировать поведение приложения при неустойчивой работе стороннего сервиса, от которого оно зависит. С помощью Istio можно написать набор надежных сквозных тестов поведения приложения, имитирующих сбои в его зависимостях, таких как следующий:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: foo-default
spec:
  hosts:
 - foo.default.svc.cluster.local
  http:
 - route:
   - destination:
       host: foo.default.svc.cluster.local
    fault:
      abort:
        httpStatus: 500
        percentage: 10
```

Например, можно смоделировать ситуацию, когда некоторый сервер отвечает на 10 % запросов кодом ошибки 500.

Входные и выходные шлюзы

Ресурсы `Gateway` в развертывании определяют *границы доверия в сети*. Другими словами, обычно определения Gateway используются для моделирования прокси на границе сети, контролирующих входящий и исходящий трафики (в среде типа Kubernetes, обеспечивающей плоскую сеть для подов, сеть охва-

тывает весь кластер). Вместе Gateway и VirtualService могут точно контролировать, как трафик входит и выходит из сетки. Более того, когда сетка Istio разворачивается с включенным определением Policy, появляется возможность применять политику к трафику на входе и выходе из сетки.

Входной шлюз

В разделе «*Gateway*» ранее было показано, как имена хостов «экспортируются» с помощью Gateway и как происходит привязка VirtualService к Gateway. После привязки VirtualService к шлюзу весь обычный функционал VirtualService, описанный в предыдущих разделах, – повторные попытки, внедрение ошибок или управление трафиком – автоматически применяется к входящему трафику. Во многих отношениях входной шлюз действует как «внешний для кластера прокси на стороне клиента».

Единственное, чего Istio не может контролировать, – как клиентский трафик попадает *во входной шлюз*. Обычной практикой в средах Kubernetes является моделирование прокси на входе в Istio в виде сервиса NodePort и передача платформе права на назначение публичных IP-адресов, создание DNS-записей и т. д.

Выходной шлюз

По аналогии с входным шлюзом, который мы рассматриваем как «внешний для кластера прокси на стороне клиента», выходной шлюз можно считать «внутренним для кластера прокси на стороне сервера». Используя комбинацию ServiceEntry, DestinationRule, VirtualService и Gateway, можно перехватывать исходящий трафик и перенаправлять его в выходной шлюз, где свободно можно применять политику для исходящего трафика.

Рассмотрим пример определения выходного шлюза. Предположим, что Istio была развернута с прокси istio-egressgateway.istio-system.svc.cluster.local на выходе. Для начала смоделируем внешнего получателя, до которого мы пытаемся добраться. Пример 8.33 использует *https://wikipedia.org* в качестве ServiceEntry (см. se-egress-gw.yaml в репозитории GitHub книги (https://oreil.ly/8NYdX)).

Пример 8.33 ❖ Определение ServiceEntry, отображаемое в выходной шлюз

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: https-wikipedia-org
spec:
  hosts:
 - wikipedia.org
  ports:
 - number: 443
    name: https
    protocol: HTTPS
location: MESH_EXTERNAL
resolution: DNS
```

```
endpoints:
- address: istio-egressgateway.istio-system.svc.cluster.local
  ports:
    http: 443
```

Как показано в примере 8.34, далее можно настроить выходной шлюз, принимающий трафик, отправляемый в `wikipedia.org` (см. egress-gwiki.yaml в репозитории GitHub книги (https://oreil.ly/_bYos)).

Пример 8.34 ❖ Выходной шлюз настроен на прием исходящего трафика, адресованного wikipedia.org

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: https-wikipedia-org-egress
spec:
  selector:
    istio: egressgateway
  servers:
 - port:
      number: 443
      name: https-wikipedia-org-egress-443
      protocol: TLS # Пометить как TLS, потому что передается трафик HTTPS.
    hosts:
   - wikipedia.org
    tls:
      mode: PASSTHROUGH
```

Однако есть проблема. Нужно, чтобы выходной шлюз получал адрес `wikipedia.org` с помощью DNS и пересылал запрос на этот адрес, а у нас все прокси в сетке настроены на пересылку запросов на имя `wikipedia.org` в выходной шлюз (поэтому прокси на выходе перешлет сообщение себе же или отбросит его). Чтобы исправить ситуацию, воспользуемся возможностью привязки `VirtualService` к `Gateway` и направим трафик для wikipedia.org на некоторое фиктивное имя, например `egress-wikipedia-org`:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: egress-wikipedia-org
spec:
  hosts:
 - wikipedia.org
  gateways:
 - https-wikipedia-org-egress
  tls:
 - match:
   - ports: 443
     sniHosts:
     - wikipedia.org
    route:
   - destination:
       host: egress-wikipedia-org
```

Затем определим ServiceEntry для разрешения egress-wikipedia-org через DNS как wikipedia.org:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: egress-https-wikipedia-org
spec:
  hosts:
 - egress-wikipedia-org
  ports:
 - number: 443
    name: https
    protocol: HTTPS
  location: MESH_EXTERNAL
  resolution: DNS
  endpoints:
 - address: wikipedia.org
    ports:
      http: 443
```

Теперь трафик будет пересылаться на внешний сайт с помощью выделенного выходного шлюза. По умолчанию Istio позволяет передавать трафик адресатам, не имеющим записей ServiceEntry. Однако для большей безопасности эту настройку по умолчанию следует отключить, а сервисы вне сетки внести в белый список, создав для них записи ServiceEntry. Чтобы разрешить доступ к внешнему сервису в обход выходного шлюза, достаточно создать для нее идентификатор ServiceEntry:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: egress-https-wikipedia-org
spec:
  hosts:
 - wikipedia.org
  ports:
 - number: 443
    name: https
    protocol: HTTPS
  location: MESH_EXTERNAL
  resolution: DNS
  endpoints:
 - address: istio-egressgateway.istio-system.svc.cluster.local
    ports:
      http: 443
```

Мы показали силу сетевых API Istio, и возможностей здесь поразительно много. Важно помнить, что все и вся в сетке Istio можно настраивать постепенно. Выберите одну функцию, важную сегодня. Примените небольшую конфигурацию к своему сервису и используйте на практике. *Затем* перейдите к следующей функции, решающей следующую проблему.

Глава 9

Mixer и политика в сетке

Различные способы использования Mixer можно разделить на две категории обязанностей: телеметрия и обеспечение соблюдения политик. Взглянув на публичные API компонента Mixer, деление обязанностей на категории становится более очевидным, поскольку Mixer имеет два основных API: `check` для предварительной проверки условий и `report` для сбора данных телеметрии. Отражением этих областей внимания является тот факт, что по умолчанию в установку Istio входят два модуля Mixer, работающих в уровне управления: модуль телеметрии и модуль обеспечения политик.

В роли точки сбора телеметрии Mixer часто описывается как механизм обработки атрибутов, поскольку он принимает атрибуты телеметрии от прокси и трансформирует и передает их (через адаптеры) во внешние системы. В роли инструмента применения политик Mixer также описывается как кеш (второго уровня), отвечающий на запросы по проверке политики трафика и кеширующий результаты оценки. Mixer потребляет различные конфигурации из разных источников и объединяет их вместе.

Архитектура

Располагаясь в уровне управления, Mixer осуществляет связь между уровнем данных и уровнем администрирования. Вопреки схеме на рис. 9.1, Mixer не является единственной точкой отказа, поскольку конфигурация Istio по умолчанию включает набор реплик подов для механизма горизонтального масштабирования подов `HorizontalPodAutoscaler`. Mixer не имеет состояния, использует методы кеширования и буферизации и разработан с повышенными требованиями к надежности и отказоустойчивости с целью обеспечения доступности на уровне 99,999 %.

На Mixer ссылаются как на единую сущность, поскольку, несмотря на разделение пространства API по функциональности, обе функции выполняет один и тот же двоичный файл в одном и том же образе Docker. Они просто настроены на разное поведение в зависимости от функции, выполняемой данным экземпляром, – *применение политик* или *сбор телеметрии*. Разделение на несколько развертываний позволяет масштабировать каждую функцию независимо, не влияя на производительность другой. Характеристики нагрузки при

применении политики и сбора телеметрии различаются, поэтому полезно оптимизировать их среду выполнения. Это позволит не только масштабировать их по отдельности, но и контролировать использование ресурсов в зависимости от выполняемой функции – сбор телеметрии или применение политик. Не совсем соседи, эти двойняшки могут мешать друг другу; от вас зависит, объединять и устанавливать их как одно целое или нет, если вы хотите оптимизировать использование ресурсов в зависимости от нагрузки среды на Mixer. Можно объединить их в одно развертывание.

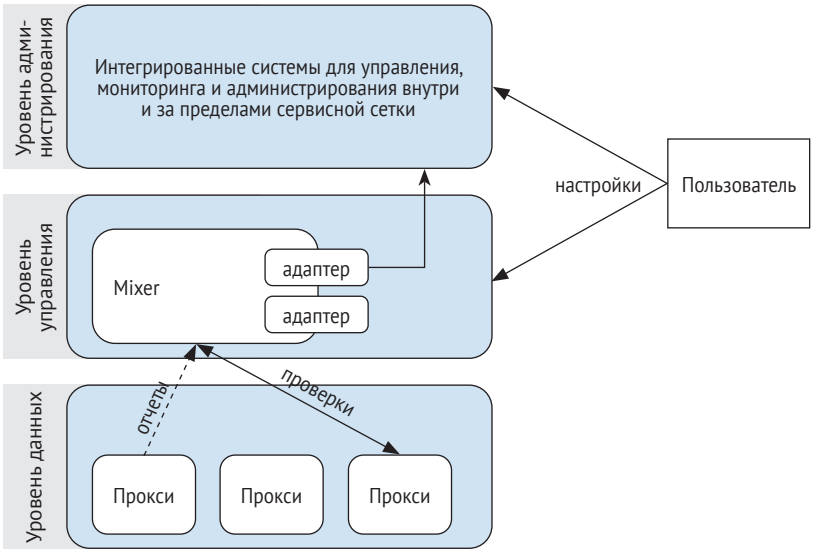

Рис. 9.1 ❖ Обзор архитектуры Mixer

Mixer выступает в качестве центрального узла обработки телеметрии, оценки политики, *а также* точки расширения. Высокая расширяемость Mixer достигается за счет реализации универсальной модели плагинов. Плагины для Mixer известны как *адаптеры*. В сетке Istio может работать любое количество адаптеров. Адаптеры расширяют две сферы ответственности Mixer:

Оценка политики (*check*)

Адаптеры могут выполнять проверку условий (например, ACL, аутентификацию) и управлять квотами (например, ограничивать частоту запросов).

Сбор телеметрии (*report*)

Адаптеры могут добавлять метрики (например, запрашивать статистику трафика), журналы и данные трассировки (т. е. производительность или другой контекст, передаваемый между сервисами).

Прокси взаимодействуют с Mixer через клиентскую библиотеку. В зависимости от API, получающего атрибуты запроса – `check` или `report`, – Mixer авторизует запрос на выполнение операции (предварительная проверка) или интерпретирует атрибуты как данные телеметрии, направляемые на анализ после завершения запроса.

Обеспечение политики

API check, экспортируемый как istio-policy, обрабатывает политики различных типов, такие как аутентификация и квотирование. Производительность и доступность API check важны при его использовании в поточном режиме (синхронно), когда прокси обрабатывает каждый запрос. На основании представленных атрибутов запроса API check проверяет, соответствует ли данный запрос активным политикам, настроенным в Mixer. В конечном счете адаптеры Mixer определяют, выполняются ли условия политики. Некоторые адаптеры проверяют условия, обращаясь к другим системам, в то время как иные выполняют проверки внутри себя (например, сверяясь с черными списками и квотами).

Квотироваться могут произвольные аспекты запроса. Например, можно ограничить частоту запросов по API-токену или IP-адресу.

Действуя в роли механизма обработки атрибутов, Mixer преобразует их в запросы к определенным службам, используя адаптеры для форматирования атрибутов в представление, понятное этим службам. Например, служба, с которой взаимодействует адаптер, может быть движком политики или системой управления. Такие системы оценивают запрос и, в зависимости от условий, дают положительный или отрицательный ответ. И без того длинный список сторонних адаптеров для Mixer постоянно пополняется разработчиками различных служб.

Давайте еще раз рассмотрим архитектуру Mixer – на этот раз в контексте того, как istio-policy подходит к приему запросов проверки, оценке политики и выдаче результатов. istio-policy предоставляет check API с фиксированным набором параметров, как показано на рис. 9.2.

Прокси обращаются в Mixer перед каждым запросом на выполнение проверки предварительных условий и после каждого запроса на отдачу результатов телеметрии. Результаты проверки из Mixer кешируются в прокси. Выступая в качестве кеша первого уровня, прокси могут отвечать на довольно большой процент предварительных проверок, выбирая результаты из своего локального кеша. Для всех остальных запросов функцию кеша выполняет Mixer. Его роль кеша второго уровня для результатов оценки политики имеет решающее значение для уменьшения объема трафика проверок политики и накладных расходов на оценку в Mixer.

Хорошо спроектированные кеши играют ключевую роль в обеспечении добросовестного подхода к безопасности в распределенных системах. В идеале запросы между сервисами аутентифицируются и авторизуются на каждом этапе в восходящей цепочке сервисов (для получения ответа на запрос может посылаться любое количество запросов к другим сервисам). Традиционно аутентификация и авторизация выполняются на границе сервиса чем-то вроде API-шлюза. Обычная практика такова, что после аутентификации и авторизации запроса на границе запросы к другим сервисам в цепочке считаются безопасными и впоследствии не проверяются.

В идеале распределенные системы должны иметь политику, применяемую на каждом шаге в цепочке сервисов, а не только на границе. Таким образом достигается последовательность в обеспечении безопасности всей распределенной системы. Это реальная проблема для сервиса аутентификации Istio Mixer: например, если каждый отдельный сервис обращается в Mixer, то один запрос,

вовлекающий восемь сервисов, направит на рассмотрение в Mixer восемь различных запросов на авторизацию. Mixer и прокси должны эффективно использовать распределенный кеш.

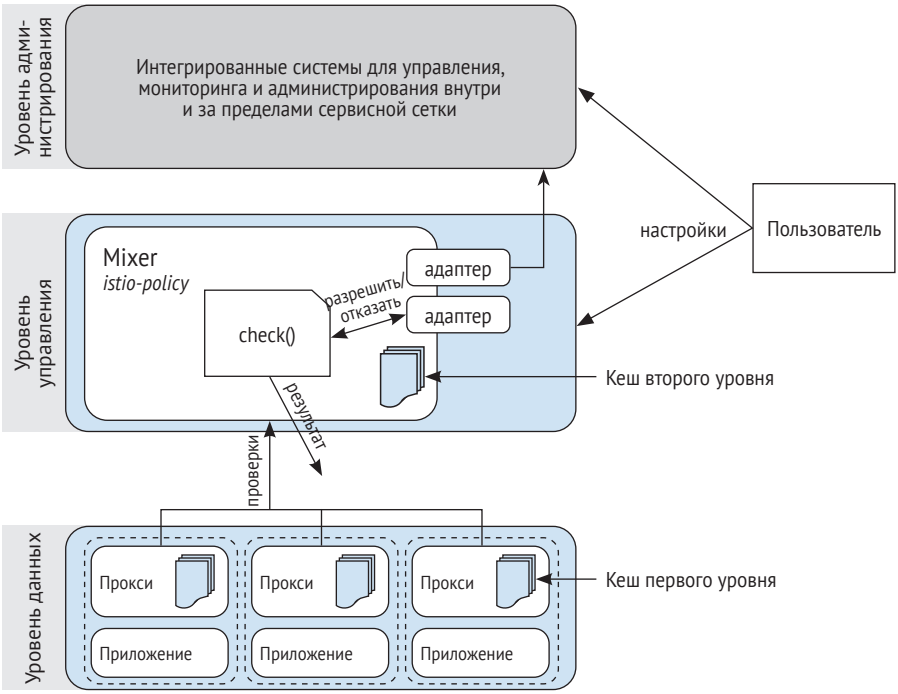

Рис. 9.2 ❖ Обзор архитектуры Mixer istio-policy

Как работают политики Mixer

Возникает вопрос: включено ли применение политики? Чтобы избежать ненужного потребления ресурсов, стандартный профиль установки Istio v1.1 и более поздних версий по умолчанию отключает принудительное применение политик. Политика контролируется в двух местах:

- в настройках Mixer Policy имеется параметр `mixer.policy.enabled`. По умолчанию он выключен. Только после его включения вступает в силу второй элемент конфигурации;
- в `global.disablePolicyChecks` управляет проверкой политик в Mixer. Если присвоить ему значение `true`, проверка политик в Mixer будет отключена. Для того чтобы вступили в силу любые изменения в этом элементе конфигурации, необходимо перезапустить Pilot.

Чтобы установить Istio с включенным применением политик, используйте параметр установки Helm `--set global.disablePolicy Checks=false`. Если сервисная сетка Istio уже развернута, сначала необходимо проверить, включено или отключено применение политики:

```
$ kubectl -n istio-system get cm istio -o jsonpath="{@.data.mesh}" |
    grep disablePolicyChecks
```

Конфигурация Mixer перечисляет используемые адаптеры и их настройки. Различные адаптеры будут отображать атрибуты запроса во входные данные адаптеров в зависимости от конкретных случаев использования и интеграции. Каждому адаптеру передаются определенные входные данные.

Передача телеметрии

Телеметрия генерируется при поступлении запроса (сетевого трафика) в прокси уровня данных. Каждому полученному запросу соответствует массив метаданных. Эти метаданные обеспечивают контекст и детали каждого запроса и фиксируются в виде атрибутов. Mixer постоянно получает и обрабатывает эти атрибуты запроса (телеметрию). На рис. 9.3 изображена конфигурация `istio-telemetry`, экспортирующая `report` API с потенциально длинным списком атрибутов, меняющимся в зависимости от адаптера.

Отчеты составляются по мере обработки запросов в Envoy и направляются асинхронно в API `report` Mixer (экспортируется компонентом `istio-telemetry`). Envoy буферизует исходящую телеметрию таким образом, что Mixer вызывается только после обработки большого количества запросов (размер буфера конфигурируется). Именно в рамках `istio-telemetry` Mixer синтезирует атрибуты и через один или несколько адаптеров передает серверу сбора телеметрии.

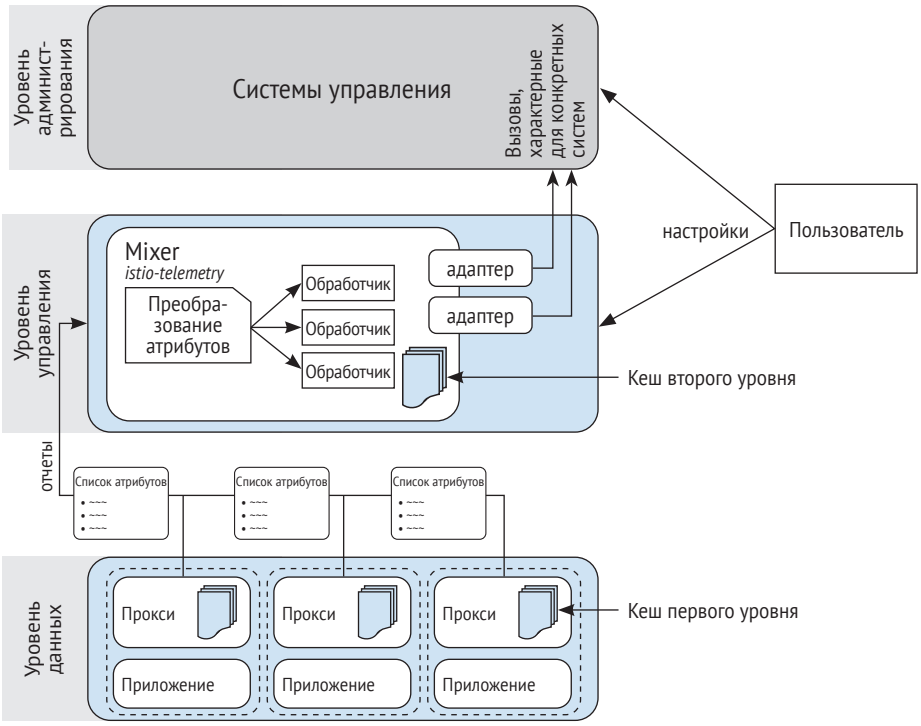

Рис. 9.3 ❖ Обзор архитектуры istio-telemetry Mixer: каждый прокси имеет буфер отчетов, где копится телеметрия и затем периодически передается в report API компонента Mixer

В архитектуре Istio v1.x адаптеры встроены в бинарный файл Mixer. Они могут включаться и выключаться в настройках.

Как уже отмечалось, отчеты по телеметрии генерируются по мере обработки запросов в прокси. Запросы поступают в форме взаимодействия клиент–сервер. Отметим, что запросы создаются клиентом и отправляются в сервис, который, в свою очередь, может инициировать запрос (в качестве клиента) к другому сервису (серверу). Как клиентские, так и серверные сервисы могут отправлять телеметрию о любом обработанном запросе. Версии Istio выше версии 1.1 настроены по умолчанию так, чтобы только сервер посылал телеметрический отчет и отправлялся лишь один отчет. Это настраивается желаемой степенью детализации как клиентских, так и серверных отчетов о запросах.

Атрибуты

Атрибуты являются ключевой концепцией Mixer и, по сути, представляют собой набор типизированных кортежей имя/значение. Они обеспечивают гибкий и расширяемый механизм передачи информации от прокси к Mixer. Атрибуты могут описывать трафик запроса и его контекст, давая администратору сетки детальный контроль в определении, какую информацию о запросе следует учитывать при проверке политики и далее преобразовывать в собираемую телеметрию. Атрибуты совершенно необходимы для правильного восприятия Istio администраторами, постоянно встречаясь в конфигурациях и журналах.

Значения атрибутов многочисленны и разнообразны в зависимости от того, какие адаптеры присутствуют и включены. Значения могут быть строковыми, целыми, вещественными, булевыми, метками времени, интервалами, IP-адресами, необработанными байтами, картами строк и т. д. Существует расширяемый словарь известных атрибутов. В табл. 9.1 показан небольшой пример атрибутов, отправляемых в Mixer во время работы. Полный набор атрибутов, известных Istio, фиксирован на момент установки, но версионируется от релиза к релизу. Для получения исчерпывающего списка нужно обратиться к вебстранице *Istio's Attribute Vocabulary* (https://oreil.ly/zGWf5).

Таблица 9.1. Примеры атрибутов, отправляемых в Mixer

Имя	Тип	Описание	Пример Kubernetes
source.uid	string	Платформозависимый уникальный идентификатор экземпляра исходного приложения	kubernetes://redis-master-2353460263-1ecey.my-namespace
source.ip (http://source.ip/)	ip_address	IP-адрес экземпляра исходного приложения	10.0.0.117
source.labels	map[string, string]	Массив пар ключ/значение, присоединенных к экземпляру источника	version ⩾ v1
source.name (http://source.name/)	string	Имя экземпляра исходного приложения	redis-master-2353460263-1ecey
source.namespace	string	Пространство имен экземпляра исходного приложения	my-namespace

Таблица 9.1 *(окончание)*

Имя	Тип	Описание	Пример Kubernetes
source.principal	string	Рабочие полномочия экземпляра исходной рабочей нагрузки	service-account-foo
source.owner	string	Ссылка на приложение, управляющее экземпляром исходной рабочей нагрузки	kubernetes://apis/ extensions/v1beta1/ namespaces/istio-system/ deployments/ istio-policy

Отправка отчетов

Прокси передают в Mixer информацию о запросах и ответах в виде атрибутов (типизированные пары ключ/значение). Mixer преобразует наборы атрибутов в структурированные значения в соответствии с конфигурацией, предоставленной администратором, а затем рассылает их адаптерам, указанным в конфигурации. Адаптеры передают данные телеметрии в управляющие системы для дальнейшего использования и анализа. Атрибуты в основном генерируются в прокси; адаптеры Mixer также могут их генерировать.

Кеширование результатов проверок

Как только Mixer вынесет первоначальный вердикт политики, в силу вступает протокол атрибутов. Envoy и Mixer используют хорошо известные атрибуты для описания политики оценки запросов к сервисам. При вынесении вердикта по запросу Mixer вычисляет хеш-ключ с использованием этих атрибутов. Envoy использует эти же атрибуты и сокращает задержку обработки запросов, поскольку после вынесения вердикта и Mixer, и Envoy работают как кеш. С помощью ключей атрибутов Mixer создает хеш-ключ. Цель конфигурации Envoy – сбалансировать попадания в кеш с помощью избыточных наборов хеш-ключей.

Каждый кеш с результатами проверок включает TTL-значение, определяющее максимальное время, в течение которого можно доверять кешированному результату. Кеши необходимо обновлять, поскольку конфигурация Mixer может измениться; точно так же могут измениться настройки системы управления, с которой сверяется Mixer при оценке результатов проверки, что приведет к необходимости обновления кеша проверок в прокси. Таким образом, прокси работают как кеши первого уровня, а Mixer – как общий кеш второго уровня.

Адаптеры

В архитектуре Istio v1.x адаптеры встроены в бинарный файл Mixer; включены они или нет, зависит от конфигурации. Конфигурация Istio Mixer активирует адаптеры. Единственные накладные расходы, связанные с неактивными адаптерами, – это размер самих адаптеров. Несколько адаптеров разных типов могут работать одновременно. Для определения сложных политик адаптеры могут связываться в цепочки. Можно скомпилировать свою версию Mixer с нужными адаптерами. В настоящее время архитектура Mixer учитывает *внут-*

рипроцессные адаптеры (*in-process adapters*), хотя gRPC-интерфейс для *внепроцессных адаптеров* (*out-of-process adapters*) был выпущен в альфа-версии в v1.0.

Адаптеры позволяют Mixer экспортировать единый согласованный API, не зависящий от используемых инфраструктурных систем управления. Большинство адаптеров обслуживают внешние, удаленные системы, в то время как другие являются автономными, обеспечивая функциональность полностью в рамках Mixer (также известные как *baby backends – внутренние потребители*). Внутренние потребители настраиваются в Istio через конфигурацию уровня адаптера. Например, адаптер `list` предоставляет простой белый или черный список проверок. Адаптеры `list` можно настроить непосредственно для проверки списка либо предоставить им URL (или путь к файлу), из которого следует извлечь список. Списки проверяются во время выполнения на соответствие спискам строк, IP-адресам или шаблонам регулярных выражений (*regex*) как включительно (инклюзивно), так и исключительно.

Внутрипроцессные адаптеры

Внутрипроцессные адаптеры пишутся на языке Go и компилируются в процесс Mixer. Опять же, вовлеченность адаптера зависит от того, настроен ли он для использования через обработчик, экземпляр и правила. Все следующие адаптеры встроены в бинарный файл Mixer и, соответственно, во все версии Istio:

Проверка предварительных условий

`denier`, `listchecker`, `memquota`, `opa`, `rbac`, `redisquota`

Телеметрия

`circonus`, `cloudwatch`, `dogstatsd`, `fluentd`, `prometheus`, `solarwinds`, `stack driver`, `statsd`, `stdio`

Генерирование атрибутов

`kubernetesenv`

Для создания нового внутрипроцессного адаптера разработчики реализуют набор интерфейсов Golang и передают свой адаптер на рассмотрение для включения в проект Istio.

Внепроцессные адаптеры

Первоначально адаптеры встраивались в бинарный файл Mixer как внутрипроцессные адаптеры, но в настоящее время осуществляется переход на внепроцессную модель, в которой код адаптера не сохраняется в проекте Istio, а хранится и управляется разработчиком отдельно. Внепроцессные адаптеры взаимодействуют через gRPC-сервис, реализующий внутренний протокол инфраструктуры шаблонов, и работают вне процесса Mixer (либо в виде сопроцесса, либо в виде сервиса). Переходя на модель внепроцессных адаптеров, Istio устраняет необходимость в пользовательских сборках Mixer для включения или исключения определенных адаптеров. Разработчики адаптеров получили возможность писать адаптеры на языке по своему выбору с учетом абстракций gRPC. При переходе на внепроцессную модель Mixer больше не будет разделять общую судьбу своих адаптеров, которые станут работать независимо в собственных процессах.

Создание политики Mixer и использование адаптеров

Для администратора последовательность шагов, выполняемых для применения политики к потокам данных в Istio, выглядит следующим образом:

- применить политику в отношении Kubernetes. Политики передаются серверу kube-api;
- Galley извлечет их и либо:
  - передаст в Pilot для реализации в виде конфигурации Envoy;
  - передаст в Mixer для динамической рассылки при подготовке прокси-серверов, вызывающих адаптеры для получения и реализации соответствующих политик.

Конфигурация Istio располагается в Kubernetes API Server. С точки зрения Mixer Kubernetes API Server представляет собой базу данных конфигураций. С точки зрения Pilot это механизм обнаружения сервисов. Один и тот же источник истины используется как для конфигурации, так и для обнаружения сервисов – это артефакт Kubernetes, а не требование Istio. Точно так же для обнаружения сервисов Istio может использовать Consul, а в роли хранилища конфигураций – `kube-api`.

Конфигурация Mixer

Администраторы сервисов контролируют все операционные и политические аспекты развертывания Mixer, управляя ресурсами конфигурации. Настройка Mixer включает описание используемых адаптеров, как они должны работать, какие атрибуты запрашивать для отображения входов того или иного адаптера и когда конкретный адаптер вовлекается с помощью определенных входных сигналов. Конфигурация Mixer управляется и представляется через CRD Kubernetes – `rule`, `handler`, `instance` и `adapter`.

Логику, необходимую для взаимодействия с управляющими системами, инкапсулирует `adapter`. Схема конфигурации адаптера определяется упаковкой адаптеров. Конфигурация содержит рабочие параметры, необходимые адаптеру для работы. `handler` создает конкретный экземпляр `adapter`, применяя соответствующие настройки. Обработчики `handler` могут получать данные (`attributes`). `instance` – это объект с данными для запроса, набор структурированных атрибутов запроса с хорошо известными полями. `Instance` отображает атрибуты запроса в значения, передаваемые в код адаптера. Отображение атрибутов контролируется с помощью выражений атрибутов.

В качестве варианта выражения атрибута в примере 9.1 рассмотрим `instance` Prometheus для `requestduration` (длительности запроса). В этом примере атрибут `requestduration` настроен так, что в случае отсутствия `response.code` будет возвращаться код 200. Если сайт `destination.service` отсутствует, отчет просто не будет отправлен.

Пример 9.1 ❖ Пример экземпляра Prometheus

```
$ kubectl -n istio-system get metrics requestduration -o yaml
apiVersion: config.istio.io/v1alpha2
kind: metric
metadata:
```

```
  name: requestduration
  namespace: istio-system
spec:
  dimensions:
    destination_app: destination.labels["app"] | "unknown"
    destination_principal: destination.principal | "unknown"
    destination_service: destination.service.host | "unknown"
    destination_service_name: destination.service.name | "unknown"
    request_protocol: api.protocol | context.protocol | "unknown"
    response_code: response.code | 200
...
```

Правила определяют, когда конкретный `handler`, обработчик, вызывается с определенным `instance`, экземпляром, и сопоставляют обработчиков с экземплярами. Правила предписывают, что когда данное условие истинно, ему присваивается зависящий от `handler` конкретный `instance` (атрибут запроса).

Правила содержат предикат соответствия для выражения атрибута и список необходимых к выполнению действий, если предикат вернет истинное значение. Если условие соответствия не указано, правила автоматически оцениваются как истинные. Такое поведение особенно удобно при учете соблюдения квот, как показано в примере 9.2, где приводится фрагмент правила для адаптера `memquota` (относящегося к типу *проверка предварительных условий*). В следующем примере правило не имеет условия соответствия, поэтому всегда оценивается как истинное при каждой проверке и, следовательно, увеличивает значение `requestcount.quota`:

Пример 9.2 ❖ Правило без условий соответствия

```
...
spec:
  actions:
 - handler: handler.memquota
    instances:
   - requestcount.quota
...
```

Адаптер открытого агента политик

Открытый агент политик (Open Policy Agent – OPA) – это универсальный механизм применения политик, используемый для снятия нагрузки с решений авторизации. Для декларативного описания политик в OPA используется язык Rego. Он реализован на языке Go и может быть развернут как библиотека или демон.

OPA-адаптер в Mixer – это адаптер *проверяющего* типа. Модель безопасности Mixer для адаптеров *проверяющего* типа требует закрывать их в случае сбоя, чтобы обеспечить безопасность. Любую политику, которую можно реализовать с помощью OPA, можно реализовать и с помощью адаптера OPA Mixer. Адаптер использует среду выполнения OPA. Атрибуты передаются в движок языка Rego для обработки полным экземпляром OPA в адаптере.

Рассмотрим следующие примеры конфигураций правила OPA-адаптера, обработчика и экземпляра (см. примеры 9.3–9.5).

Пример 9.3 ❖ Пример конфигурации правила OPA

```
apiVersion: config.istio.io/v1alpha2
kind: rule
metadata:
  name: authz
spec:
  actions:
 - handler: opa-handler
  instances:
 - authz-instance
```

Пример 9.4 ❖ Пример конфигурации OPA instance

```
apiVersion: config.istio/v1alpha2
kind: authz
metadata:
  name: authz-instance
spec:
  subject:
    user: source.uid | ""
  action:
    namespace: target.namespace | "default"
    service: target.service | ""
    path: target.path | ""
    method: request.method | ""
```

Пример 9.5 ❖ Пример конфигурации OPA handler

```
apiVersion: config.istio.io/v1alpha2
kind: opa
metadata:
  name: opa-handler
spec:
  checkMethod: authz.allow
  policy: |
    package authz
    default allow = false
    allow { is_read }
    is_read { input.action.method = "GET" }
```

Какие политики приходят из Pilot, а какие из Mixer?

Политики, влияющие на трафик, определяются в Pilot. Политики, требующие принудительной аутентификации и авторизации запросов, применяются в Mixer. Если политике для принятия решения необходимо проконсультироваться с внешней системой, она применяется в Mixer.

Адаптер Prometheus

Адаптер Prometheus встроен в бинарный файл Mixer и включен по умолчанию с метриками, срок действия которых составляет 10 мин. Адаптер Prometheus определяет пользовательский ресурс, *metrics*, как показано в примере 9.6.

Пример 9.6 ❖ Перечень метрик, отслеживаемых в компонентах Istio, доступных Prometheus

```
$ kubectl -n istio-system get metrics
NAME                    AGE
requestcount            25h
requestduration         25h
requestsize             25h
responsesize            25h
tcpbytereceived         25h
tcpbytesent             25h
tcpconnectionsclosed    25h
tcpconnectionsopened    25h
```

Обработчик Prometheus должен знать специфические размерности и типы метрик. В общем случае это и есть объект `instance`, но есть несколько особых `instance`, имеющих собственное имя; `metric` – один из них. Как правило, адаптер строится с расчетом на определенные объекты `instance`. В примере 9.7 перечислены объекты `instance`, которые потребляет адаптер Prometheus.

Пример 9.7 ❖ Пример набора объектов instance, которые потребляет адаптер Prometheus

```
$ kubectl -n istio-system get metrics tcpbytereceived -o yaml
apiVersion: config.istio.io/v1alpha2
kind: metric metadata:
  labels:
    app: mixer
    release: istio
  name: tcpbytereceived
  namespace: istio-system
spec:
  dimensions:
    connection_security_policy: conditional((context.reporter.kind | "inbound") ==
      "outbound", "unknown", conditional(connection.mtls | false, "mutual_tls",
      "none"))
    destination_app: destination.labels["app"] | "unknown"
    destination_principal: destination.principal | "unknown"
    destination_service: destination.service.host | "unknown"
    destination_service_name: destination.service.name | "unknown"
    destination_service_namespace: destination.service.namespace | "unknown"
    destination_version: destination.labels["version"] | "unknown"
    destination_workload: destination.workload.name | "unknown"
    destination_workload_namespace: destination.workload.namespace | "unknown"
    reporter: conditional((context.reporter.kind | "inbound") == "outbound",
      "source", "destination")
    response_flags: context.proxy_error_code | "-"
    source_app: source.labels["app"] | "unknown"
    source_principal: source.principal | "unknown"
    source_version: source.labels["version"] | "unknown"
    source_workload: source.workload.name | "unknown"
    source_workload_namespace: source.workload.namespace | "unknown"
  monitored_resource_type: '"UNSPECIFIED"'
  value: connection.received.bytes | 0
```

Mixer должен знать, когда генерировать эти метрики и отправлять в Prometheus. Это определяется как правило. Каждое правило имеет условие соответствия; если условие оценивается как истинное, то правило срабатывает. Например, можно определить условие, соответствующее только HTTP-данным, только TCP-данным и т. д. Prometheus делает именно это, как показано в примере 9.8, и определяет правило для каждого набора протоколов, для которых у него есть описания метрик.

Пример 9.8 ❖ Список правил Mixer

```
$ kubectl -n istio-system get rules
NAME                        AGE
kubeattrgenrulerule         25h
promhttp                    25h
promtcp                     25h
promtcpconnectionclosed     25h
promtcpconnectionopen       25h
stdio                       25h
stdiotcp                    25h
tcpkubeattrgenrulerule      25h
```

Давайте извлечем одно правило и посмотрим, как оно выглядит:

```
$ kubectl -n istio-system get rules promtcpconnectionopen -o yaml
```

```
apiVersion: config.istio.io/v1alpha2
kind: rule
metadata:
  annotations:
 ...
  generation: 1
  name: promtcpconnectionopen
  namespace: istio-system
spec:
  actions:
 - handler: prometheus
    instances: tcpconnectionsopened.metric
  match: context.protocol == "tcp" && ((connection.event | "na") == "open")
```

В главе 10 все перечисленные метрики открыты управляющим системам для анализа, визуализации, оповещения и т. д.

Mixer: ревизия дизайна

Сильными сторонами архитектуры Mixer v1 являются гибкая модель адаптеров, широкие возможности, изолирование сетки от особенностей и неполадок внешних систем. Архитектура v1 выступает в качестве кеша второго уровня для результатов предварительной проверки. Несмотря на то что Mixer увеличивает уровень обслуживания, повышая доступность сетки и сокращая время ожидания за счет агрессивного использования общих многоуровневых кешей, существует мнение, что этого недостаточно. Были высказаны сомнения в отношении потребления ресурсов на управление, накладных расходов, связанных с задержкой запросов, а также опасения по поводу того, что это единственная точка отказа. Простота использования Mixer также была поставлена под сомнение. С самого на-

чала проекта (v0.3) разработчики определили эти соображения и говорили о них (https://oreil.ly/BQexz).

Еще предстоит определить, произойдет ли в архитектуре Mixer v2 перемещение бóльшей части функциональности Mixer в фильтры Envoy, которые обычно пишутся на C++. Однако уже сейчас доступна поддержка HTTP-фильтров на Lua, позволяющая выполнять сценарии Lua (с использованием LuaJIT) в потоках запросов и ответов. Фильтры могут быть написаны на других языках с использованием gRPC-расширений.

Являясь многопользовательским компонентом уровня управления, Mixer позволяет администратору сервиса управлять решениями в области политики и телеметрии на основе конфигурации и выступает в качестве точки интеграции между Istio и другими компонентами инфраструктуры. С помощью двух сервисов (`istio-policy` и `istio-telemetry`) Mixer предоставляет следующие основные возможности:

- проверка предварительных условий (ACL, аутентификация);
- управление квотами (ограничение частоты запросов);
- телеметрическая отчетность (метрики, логи, трассировки).

Mixer может потреблять достаточно много ресурсов, и это зависит от конфигурации Istio. Mixer обеспечивает агрессивное кеширование и сокращает наблюдаемые задержки, а также помогает администраторам контролировать соблюдение политик и сбор данных телеметрии. Благодаря абстракции компонентов инфраструктуры снижает сложность систем, а модель адаптеров обеспечивает мобильность инфраструктуры.

Глава 10

Телеметрия

При работе с микросервисами важнейшее значение имеет способность анализировать их поведение, которая обеспечивается не только триумвиратом журналов, метрик и трассировок, но также инструментами визуализации, поиска и устранения неисправностей и отладки. В главе 2 было показано, как сервисные сетки в целом и Istio в частности обеспечивают равномерную наблюдаемость. В этой главе мы рассмотрим особенности различных сигналов и инструментов, доступных для мониторинга сервисов, работающих в Istio. Устранение неисправностей и отладка будут обсуждаться в главе 11.

Mixer (описанный в главе 9) играет ключевую роль в сборе и объединении данных телеметрии, полученных с помощью прокси. Во время работы прокси генерируют телеметрию на основе обрабатываемого трафика и буферизуют ее перед выдачей в Mixer для дальнейшей обработки. Половина работы Mixer заключается в сборе, переводе и передаче этих важных сигналов (вторая половина – авторизация). Маршрутизация этих различных сигналов полностью зависит от того, какие адаптеры и какого типа используются в Mixer. Давайте детально рассмотрим адаптеры Mixer.

В главе 4 мы познакомились с приложением Bookinfo – каноническим примером применения Istio, и в этой главе мы вновь используем его в качестве испытуемого.

Модели адаптеров

Как описано в главе 9, адаптеры интегрируют Mixer с различными компонентами инфраструктуры, обеспечивающими следующие основные функции: протоколирование, мониторинг, квотирование, проверка списка контроля доступа и многое другое. Администраторы могут выбирать количество и тип используемых адаптеров, обеспечивающих интеграцию с существующими системами или ценных самих по себе. Mixer поддерживает одновременное включение нескольких адаптеров одного и того же типа: например, два адаптера для передачи журналов в две разные системы. Существует особый случай, когда адаптеры, производящие атрибуты, всегда запускаются первыми перед адаптерами телеметрии или адаптерами политики. Наиболее ярким примером одного из таких адаптеров является `kubernetesenv`: он извлекает информацию из среды Kubernetes и создает атрибуты, которые можно использовать в следующих адаптерах.

Телеметрические адаптеры также работают параллельно. В отношении пакетной обработки существует немного больше сложностей, но логично, что Mixer параллельно отправляет вызовы в адаптеры и ждет их завершения. Одновременно можно развернуть два одинаковых типа адаптеров.

Отчеты телеметрии

В главе 2 мы говорили, что Istio поддерживает три формы телеметрии (метрики, журналы, трассировки), позволяющие с разных сторон взглянуть на происходящее. Телеметрия передается из уровня данных в уровень управления. Отчеты прокси содержат *атрибуты* (подробнее об атрибутах см. в главе 9). Контекстные атрибуты обеспечивают возможность различать протоколы HTTP и TCP внутри политик. Свои отчеты с атрибутами прокси-серверы отправляют в три разных момента времени:

- сразу после установки соединения (начальный отчет);
- через определенные интервалы времени, пока соединение остается открытым (периодические отчеты);
- перед закрытием соединения (завершающий отчет).

По умолчанию периодические отчеты посылаются через каждые 10 секунд, однако предпочтительнее, чтобы интервал измерялся не единицами секунд.

Метрики

Метрики сервисов собирают соответствующие им прокси и отправляют их в сервис Mixer `istio-telemetry`. В Mixer может быть загружено любое количество любых типов адаптеров. Адаптер Mixer, реализованный на базе шаблона адаптера `metric`, может использоваться для сбора и обработки метрик, переданных ему компонентом Mixer. Рассмотрим, как настраиваются адаптеры в целом, а затем разберем пример использования адаптера Prometheus Mixer.

Настройка Mixer для сбора метрик

Телеметрия (и политика) конфигурируется с использованием трех типов ресурсов:

Обработчики (*handlers*)
: Определяют набор используемых адаптеров и их работу. Примером конфигурации обработчика может служить предоставление удаленному syslog-серверу адаптера журналирования с IP-адресом.

Экземпляры (*instances*)
: Описывают соответствие атрибутов запросов (генерируемых прокси-серверами) входам адаптера, на которые адаптер будет получать сгенерированную телеметрию. Экземпляры представляют собой фрагмент данных, с которыми будут работать один или несколько адаптеров. Например, администратор может принять решение сгенерировать экземпляры метрики `request_bytes` из атрибута `destination_workload`.

Правила (*rules*)

Определяют, когда вызывается определенный адаптер и какие экземпляры ему передаются (какая телеметрия к нему поступает). Правила состоят из выражения соответствия и действий. Выражение соответствия определяет, когда вызывать адаптер, а действия – набор экземпляров для передачи адаптеру.

Чтобы использовать адаптер Prometheus Mixer, нужен сервер Prometheus, установленный либо в том же кластере Kubernetes, либо в другом месте, способный извлекать метрики из адаптера Prometheus Mixer. Существует множество способов развертывания Prometheus в Kubernetes или снаружи; эти подробности выходят за рамки данной книги.

Настройка сбора и запроса метрик

Для настройки и использования адаптера Prometheus Mixer необходимо выполнить следующие действия.

1. Создайте экземпляр `metric`, настраивающий генерируемые и собираемые метрики Istio, и сконфигурируйте обработчик Prometheus для сбора метрик. Назначьте соответствующие метки и предоставьте их экземпляру Prometheus для обработки, как показано в примере 10.1 (сетевой трафик; см. полную конфигурацию в репозитории GitHub книги (https://oreil.ly/GcDZ5)).

Пример 10.1 ❖ Шаг 1: фрагмент обработчика Prometheus и метрика `requests_total`, отслеживаемая им по разным меткам

```
apiVersion: "config.istio.io/v1alpha2"
kind: handler
metadata:
  name: prometheus
  namespace: istio-system
spec:
  compiledAdapter: prometheus
  params:
    metrics:
   - name: requests_total
      instance_name: requestcount.metric.istio-system
      kind: COUNTER
      label_names:2
     - reporter
     - source_app
     - source_namespace
     - source_principal
     - source_workload
     - source_workload_namespace
     - source_version
     - destination_app
     - destination_namespace
     - destination_principal
     - destination_workload
     - destination_workload_namespace
     - destination_version
```

```
    - destination_service
    - destination_service_name
    - destination_service_namespace
    - request_protocol
    - response_code
    - connection_mtls
```

2. Обновите Prometheus для извлечения метрик из адаптера Prometheus Mixer и создайте правила, передающие собранные метрики в адаптер Prometheus Mixer с настроенными метками, как показано в примере 10.2.

Пример 10.2 ❖ Шаг 2: правило для сопоставления HTTP-трафика и выполнения действий по передаче метрик обработчику Prometheus

```
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
  name: promhttp
  namespace: istio-system
  labels:
    app: mixer
    chart: mixer
    heritage: Tiller
    release: istio
spec:
  match: (context.protocol == "http" || context.protocol == "grpc") &&
         (match((request.useragent | "-"), "kube-probe*") == false)
  actions:
 - handler: prometheus
    instances:
   - requestcount.metric
   - requestduration.metric
   - requestsize.metric
   - responsesize.metric
```

Трассировка

Распределенная трассировка, вероятно, является наиболее глубокой телеметрической информацией, получаемой из сервисной сетки, дающей подсказки в трудных вопросах типа «Почему сервис работает медленно?». В состав Istio входят Zipkin и Jaeger – популярные распределенные системы трассировки с открытым кодом для хранения, агрегирования и интерпретации данных.

Формирование диапазонов трассировки

Прокси Envoy в сетке Istio отвечает за создание первоначальных заголовков трассировки и делает это в OpenTelemetry-совместимом виде. OpenTelemetry (ранее OpenTracing) – это не зависящая от языка спецификация распределенной трассировки. Заголовок `x-request-id` формируется и используется Envoy для однозначной идентификации запроса, а также для журналирования и трассировки. Envoy рассылает `x-request-id` всем сервисам, с которыми взаимодействует запрос, а также включает уникальный идентификатор запроса в генериру-

мые им регистрационные записи. Таким образом, если понадобится найти уникальный `request-id` в такой системе, как Kibana, вам придется выполнить поиск этого запроса в журналах всех сервисов.

Рассылка заголовков трассировки

В этой области возможности Istio могут быть преувеличены. Прокси является воротами в приложение и имеет массу информации о поступающих и исходящих запросах. Но он не может полностью освободить приложение от необходимости использовать дополнительные инструменты. Ваше приложение должно использовать библиотеку тонкого клиента для сбора и рассылки небольшого набора HTTP-заголовков, включая следующие:

- `x-request-id`;
- `x-b3-traceid`;
- `x-b3-spanid`;
- `x-b3-parentspanid`;
- `x-b3-sampled`;
- `x-b3-flags`;
- `x-ot-span-context`.

Каждый сервис в примере приложения Bookinfo поддерживает рассылку HTTP-заголовков трассировки. Благодаря этому имеется возможность использовать Jaeger (например) в качестве распределенной системы трассировки для изучения задержек запросов внутри и между узлами. В примере 10.3 представлена простая программа на Go с функцией для приема HTTP-запросов, извлечения заголовков трассировки и вывода их в stdout (см. пример кода (https://oreil.ly/zg-4Q) в репозитории GitHub книги).

Пример 10.3 ❖ Простая программа Go для вывода заголовков трассировки

```
package main

import (
    "fmt"
    "log"
    "net/http"
)

func tracingMiddleware(next http.HandlerFunc) http.HandlerFunc {
    incomingHeaders := []string{
        "x-request-id",
        "x-b3-traceid",
        "x-b3-spanid",
        "x-b3-parentspanid",
        "x-b3-sampled",
        "x-b3-flags",
        "x-ot-span-context",
    }

    return func(w http.ResponseWriter, r *http.Request) {
        for _, th := range incomingHeaders {
            w.Header().Set(th, r.Header.Get(th))
        }
        next.ServeHTTP(w, r)
```

```
    }
}

func main() {
    http.HandleFunc("/", tracingMiddleware(func(w http.ResponseWriter,
                        r *http.Request) {
        fmt.Fprintf(w, "Hello headers, %v", r.Header)
    }))

    log.Fatal(http.ListenAndServe(":8081", nil))
}
```

Отключение трассировки

Отбор трассировки запроса действительно является накладным с точки зрения производительности. Как показывает Meshery (https://oreil.ly/WsTUW) на рис. 10.1, существует значительная разница между отбором трассировки с частотой 1 % и 100 %.

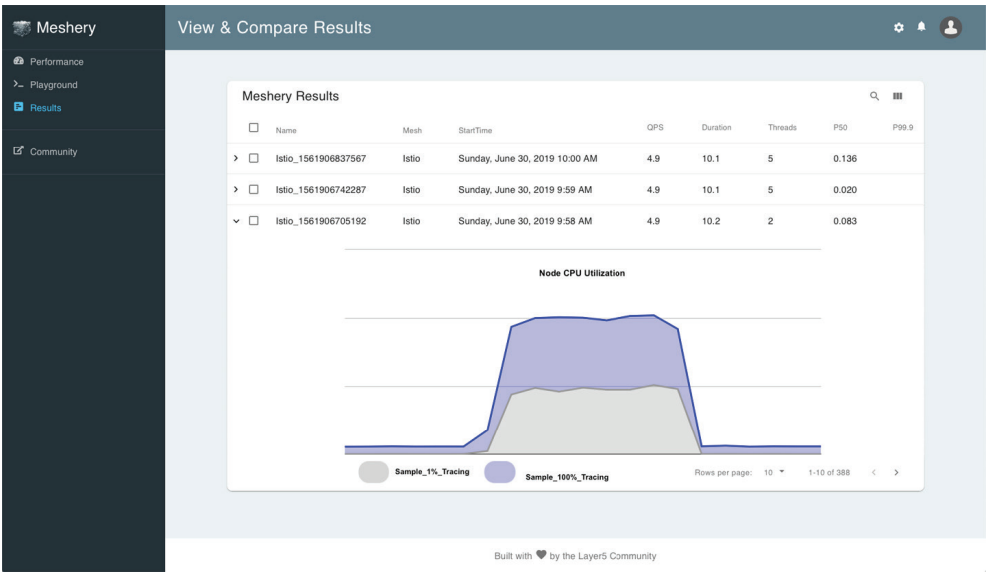

Рис. 10.1 ❖ Разница в среднем использовании CPU узла между двумя тестами производительности

Самый простой способ запустить сетку Istio без трассировки – не включать ее при установке сервисной сетки. То есть при установке с помощью Helm следует добавить параметр:

```
--set tracing.enabled=false
```

Конфигурационные профили Istio `default` и `minimal` не включают трассировку при установке сетки. Если Istio развернута с функцией трассировки, а теперь ее нужно отключить (предположим, что уровень управления установлен в пространстве имен `istio-system`), выполните следующее:

```
$TRACING_POD=`kubectl get po -n <istio namespace> | grep istio-tracing
              | awk `{print $1}`
$ kubectl delete pod $TRACING_POD -n <istio namespace>
$ kubectl delete services tracing zipkin  -n <istio namespace>
```

Удалите ссылки на Zipkin из конфигурации Mixer:

```
$ kubectl -n istio-system edit deployment istio-telemetry
```

Достаточно вручную удалить вхождения `trace_zipkin_url` из файла и сохранить его.

Журналы

Журналы доступа к сервисам являются важнейшим источником информации. Главенствующую роль в их формировании играет адаптер `logentry`, встроенный в Mixer. Также для сбора и обработки журналов, передаваемых компонентом Istio Mixer, можно использовать адаптеры, созданные сообществом на основе шаблона logentry. Среди них популярностью пользуется адаптер Fluentd Mixer.

Для использования адаптера Fluentd понадобится демон Fluentd, работающий в том же кластере Kubernetes или в другом месте (существует множество способов развертывания Fluentd как в Kubernetes, так и за его пределами; эти подробности выходят за рамки данной книги). Как и в случае с другими адаптерами Mixer, для настройки и использования адаптера Fluentd Mixer необходимо выполнить следующие действия:

1) создать экземпляр `logentry`, настраивающий поток журналов, генерируемый и собираемый Istio;
2) настроить обработчик Fluentd на передачу собранных журналов принимающему демону Fluentd;
3) создать правило Istio, пересылающее собранные журналы в адаптер Fluentd Mixer.

Вариант конфигурации в примере 10.4 предполагает, что демон Fluentd доступен на `localhost:24224`.

Пример 10.4 ❖ Конфигурация журналирующего адаптера Mixer

```
apiVersion: "config.istio.io/v1alpha2"
kind: logentry
metadata:
  name: istiolog
  namespace: istio-system
spec:
  severity: '"warning"'
  timestamp:  request.time
  variables:
    source: source.labels["app"] | source.service | "unknown"
    user: source.user | "unknown"
    destination: destination.labels["app"] | destination.service | "unknown"
    responseCode: response.code     | 0
    responseSize: response.size | 0
    latency: response.duration | "0ms"
```

```
  monitored_resource_type: '"UNSPECIFIED"'
---
# Конфигурация обработчика fluentd
apiVersion: "config.istio.io/v1alpha2"
kind: fluentd
metadata:
  name: handler
  namespace: istio-system
spec:
  address: "localhost:24224"
  integerDuration: n
---
# Правило для отправки экземпляров logentry обработчику fluentd
apiVersion: "config.istio.io/v1alpha2"
kind: rule
metadata:
  name: istiologtofluentd
  namespace: istio-system
spec:
  match: "true" # Совпадение для всех запросов
  actions:
 - handler: handler.fluentd
    instances:
   - istiolog.logentry
```

Здесь установлен глобальный уровень журналирования по умолчанию *Info*, но можно установить желаемый уровень в конфигурации экземпляров. Помимо уровней важности, можно настроить условия совпадения, например регистрировать только запросы, завершившиеся неудачей. Для ответов, отличных от 200, можно отредактировать условие совпадения, как показано в примере 10.5.

Пример 10.5 ❖ Пример настройки условий совпадения

```
apiVersion: "config.istio.io/v1alpha2"
kind: instance
metadata:
  name: requestcount
  namespace: {{.Release.Namespace }}
  labels:
    app: {{ template "mixer.name". }}
    chart: {{ template "mixer.chart". }}
    heritage: {{.Release.Service }}
    release: {{.Release.Name }}
spec:
  compiledTemplate: metric
  params:
    value: "1"
    dimensions:
      reporter: conditional((context.reporter.kind | "inbound") == "outbound",
           "source", "destination")
      source_workload: source.workload.name | "unknown"
      source_workload_namespace: source.workload.namespace | "unknown"
```

```
    source_principal: source.principal | "unknown"
    source_app: source.labels["app"] | "unknown"
    source_version: source.labels["version"] | "unknown"
    destination_workload: destination.workload.name | "unknown"
    destination_workload_namespace: destination.workload.namespace | "unknown"
    destination_principal: destination.principal | "unknown"
    destination_app: destination.labels["app"] | "unknown"
    destination_version: destination.labels["version"] | "unknown"
    destination_service: destination.service.host | "unknown"
    destination_service_name: destination.service.name | "unknown"
    destination_service_namespace: destination.service.namespace | "unknown"
    request_protocol: api.protocol | context.protocol | "unknown"
    response_code: response.code | 200
    response_flags: context.proxy_error_code | "-"
    permissive_response_code: rbac.permissive.response_code | "none"
    permissive_response_policyid: rbac.permissive.effective_policy_id | "none"
    connection_security_policy: conditional((context.reporter.kind | "inbound")
        == "outbound", "unknown", conditional(connection.mtls | false,
        "mutual_tls", "none"))
monitored_resource_type: '"UNSPECIFIED"'
```

Разработчики Istio предварительно добавили поддержку выборки (и экспериментировали с вариантами типа протоколирования только ошибок), но в данный момент из этого нельзя сделать никаких реальных рекомендаций. Похоже, этот вопрос отложен до будущих версий.

Метрики

Производительность сервисов легко оценить с использованием инструментов визуализации метрик в виде графиков и диаграмм. Grafana – это популярный инструмент визуализации метрик с открытым исходным кодом, используемый для опроса, анализа и оповещения. Grafana действует не как адаптер Mixer, а добавляется в сетку Istio как дополнение и настраивается на чтение метрик из Prometheus. Prometheus – это база данных временных рядов и набор инструментов для сбора данных. Версия Grafana в Istio поставляется с уже готовыми дашбордами. Перечень метрик, отображаемых в дашбордах Grafana, зависит от конфигурации Prometheus. Вот некоторые дашборды, входящие в комплект:

Обзор сетки
: Предоставляет глобальный сводный обзор сервисной сетки и показывает рабочие нагрузки HTTP/gRPC и TCP.

Обзор отдельных сервисов
: Предоставляет метрики по запросам и ответам по каждому отдельному сервису в сетке (HTTP/gRPC и TCP). Также отображает метрики рабочих нагрузок рассматриваемого сервиса.

Обзор по отдельным рабочим нагрузкам
: Предоставляет метрики о запросах и ответах по каждой отдельной рабочей нагрузке в сетке (HTTP/gRPC и TCP). Кроме того, дает метрики входящей нагрузки и исходящих сервисов для рассматриваемой рабочей нагрузки.

Визуализация

Возможность визуализации топологии на основе телеметрических данных является (подобно включению освещения) ключевым аспектом для понимания происходящего в сетке. Ранее Istio предлагала рудиментарное решение этой задачи под названием ServiceGraph со следующими конечными точками:

- */force/forcegraph.html*, интерактивная *D3.js*-визуализация;
- */dotviz*, статическая визуализация Graphviz;
- */dotgraph*, DOT-сериализация;
- */d3graph*, JSON-сериализация для визуализации D3;
- */graph*, общая JSON-сериализация.

Впоследствии решение ServiceGraph было заменено дополнением Kiali (https://www.kiali.io/) с веб-интерфейсом для просмотра графа сервисов в сетке и объектов конфигурации Istio. Другое приложение, обрабатывающее поток трафика в реальном времени, – это Vistio (https://oreil.ly/YpvqZ), помогающее визуализировать трафик кластера на основе данных Prometheus.

Сервисные сетки занимают уникальное положение основополагающего компонента при построении наблюдаемой системы. Прокси уровня данных располагаются на пути запросов, могут отслеживать и сообщать важные параметры системы. Телеметрия стоит дорого, поэтому к ней следует относиться взвешенно. Проекты, такие как Kiali Istio, помогают визуализировать конфигурацию сетки Istio и трафик, протекающий через нее.

Глава 11

Отладка Istio

Работа с Istio, как и с любым другим программным обеспечением, время от времени требует устранения неисправностей и отладки. Istio и другие инструменты с открытым исходным кодом обеспечивают протоколирование, интроспекцию и отладку, помогая управлять компонентами.

Поддержка интроспекции в компонентах Istio

При разработке компонентов Istio изначально учитывалась возможность поддержки широко известного пакета под названием ControlZ (ctrlz – https://oreil.ly/9yY2K). ControlZ – гибкий фреймворк интроспекции, позволяющий исследовать и изменять внутреннее состояние компонентов Istio. ControlZ предлагает административный интерфейс, которому компоненты предоставляют доступ, открывая порт (9876 по умолчанию), доступный из веб-браузера или через REST для использования внешними инструментами. Простой интерфейс фреймворка ControlZ дает интерактивный доступ к внутреннему состоянию компонентов Istio.

Mixer, Pilot, Citadel и Galley уже собраны с поддержкой пакета `ctlz`, но это не относится к шлюзам. Шлюзы не реализуют административный интерфейс ControlZ. Являясь экземплярами Envoy, они предлагают административную консоль Envoy. При запуске компонентов Mixer, Pilot, Citadel и Galley в журнал записывается сообщение с указанием IP-адреса и порта подключения для взаимодействия с ControlZ. ControlZ основан на понятии «темы». Каждому разделу пользовательского интерфейса соответствует своя тема. Существует набор встроенных тем, представляющих основную функциональность интроспекции, и каждый компонент уровня управления, поддерживающий ControlZ, может добавлять свои темы в соответствии с их назначением.

По умолчанию ControlZ использует порт 9876. Этот порт можно переопределить с помощью параметров командной строки `--ctrlz_port` и `--ctrlz_address` при запуске компонентов, указав в них конкретный адрес и порт, который будет открыт для ControlZ.

Для доступа к интерфейсу ControlZ компонентов уровня управления выполните переадресацию порта локального хоста в удаленный порт ControlZ с помощью `kubectl`.

```
$ kubectl port-forward -n istio-system istio-pilot-74cb7cd5f9-lbndc 9876:9876
```

Для удаленного доступа к ControlZ откройте в браузере адрес: http://localhost:9876, как показано на рис. 11.1.

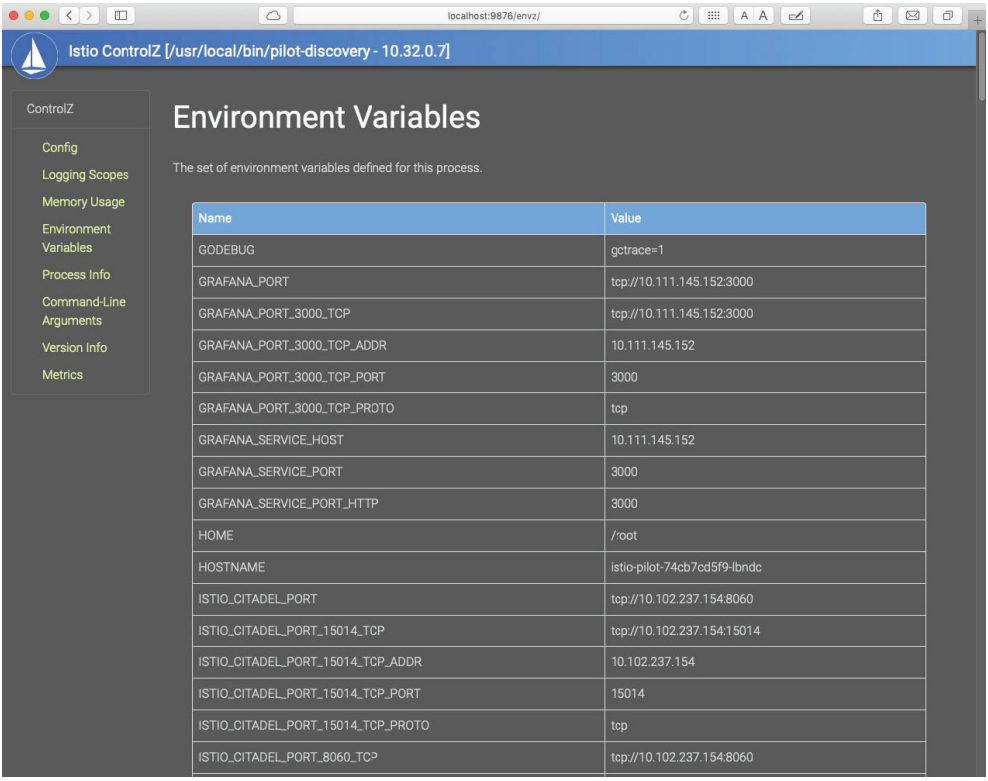

Рис. 11.1 ❖ Интерфейс ControlZ с состоянием компонента Pilot

ControlZ реализует возможность интроспекции Istio. При интеграции с ControlZ компоненты автоматически получают IP-порт, позволяющий администраторам просматривать и контролировать ряд аспектов каждого процесса, включая управление уровнем журналирования, просмотр параметров командной строки, использование памяти и многое другое. Кроме того, на порту реализован REST API, обеспечивающий доступ к тем же параметрам и управление ими.

Административный интерфейс ControlZ в первую очередь полезен для включения более подробного уровня журналирования во время отладки. Другие инструменты, такие как Meshery, располагающиеся на уровне администрирования, обычно используются в качестве более эффективной точки контроля для управления жизненным циклом Istio и рабочими нагрузками сетки. Давайте рассмотрим пример уровня администрирования.

Отладка с использованием уровня администрирования

Как уже рассматривалось в главе 1, уровень администрирования находится выше уровня управления и работает с несколькими однородными и разнород-

ными кластерами сервисных сеток. Помимо прочего, уровень администрирования может выполнять проверку рабочих нагрузок и конфигурации сетки при подготовке к включению рабочей нагрузки в сетку или при постоянной проверке их конфигурации по мере обновления версий компонентов, работающих в уровнях управления и данных или приложений. Проведем предварительный анализ, выполнив ряд проверок существующей конфигурации рабочей нагрузки (и конфигурации сервисной сетки). Для этого установим и запустим Meshery (https://oreil.ly/xubaa), загрузив инструмент командной строки mesheryctl, как показано в примере 11.1.

Пример 11.1 ❖ Локальная установка уровня администрирования Meshery

```
$ sudo curl -L https://git.io/mesheryctl -o /usr/local/bin/mesheryctl
$ sudo chmod a+x /usr/local/bin/mesheryctl
$ mesheryctl start
```

Для постоянной проверки конфигурации установите Meshery в своем кластере (кластерах). Если пользовательский интерфейс Meshery не загружается автоматически, откройте его по адресу: http://localhost:9081. В зависимости от параметров окружения Meshery автоматически подключится к кластеру (кластерам), проанализирует сетку и конфигурацию рабочей нагрузки, а также выявит отличия от лучших вариантов конфигурации или предложит устранение проблемных моментов, как показано на рис. 11.2.

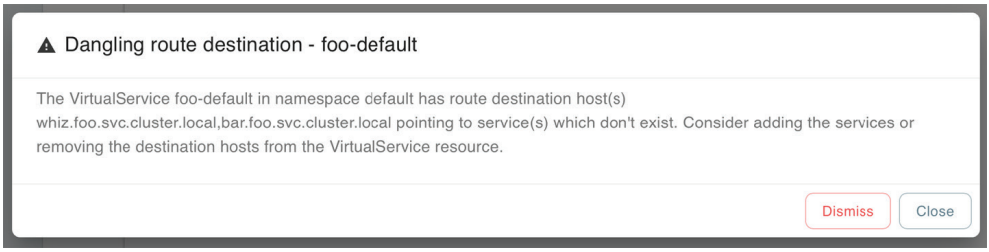

Рис. 11.2 ❖ Meshery определяет проблемы в конфигурациях сервисных сеток[1]

С использованием kubectl

Как отмечалось в контексте отладки Pilot в главе 7, Meshery и istioctl являются мощными инструментами для оценки синхронизации и координации между компонентами уровня управления Istio и прокси уровня данных. Для управления кластерами Kubernetes инструменты Meshery и istioctl используют `kubectl exec`. В своей работе `kubectl exec` применяет один из двух потоковых протоколов HTTP для передачи данных от локально выполняемой команды `kubectl` на стороне Kubernetes API Server команде `kubelet` на узле, где запущен прокси, опрашиваемый Meshery или командой `istioctl`.

---

[1] На рисунке: «**Подвисший маршрут назначения.** VirtualService foo-default в пространстве имен default определяет маршрут к хосту(ам) whiz.foo.svc.cluster.local, bar.foo.svc.cluster.local, ведущий к несуществующим сервисам. Добавьте сервисы или удалите целевые хосты из ресурса VirtualService». – *Прим. перев.*

Конкретная механика работы двух потоковых HTTP-протоколов зависит от версии Kubernetes и используемого контейнера; Kubernetes API Server поддерживает протокол SPDY (ныне устаревший) и HTTP/2 WebSockets (если вы не знакомы с WebSockets, просто рассматривайте WebSockets как протокол, преобразующий HTTP в двунаправленный протокол потоковой передачи байтов). Поверх этого потока Kubernetes API Server вводит дополнительный мультиплексированный протокол потоковой передачи. Причина этого в том, что API Server часто может обслуживать несколько независимых потоков, что весьма полезно. Рассмотрим, например, выполнение команды внутри контейнера. На самом деле в этом случае необходимо поддерживать три потока: `stdin`, `stderr` и `stdout`.

При вызове `kubectl exec` выполняет следующую последовательность действий. Сначала он отправляет HTTP POST-запрос в Kubernetes API Server

```
/api/v1/namespaces/$NAMESPACE/pods/$NAME/exec
```

со строкой запроса, определяющей, в каком контейнере выполнять команду (или команды) и устанавливать ли мультиплексированный двунаправленный поток `stdin`, `stdout` и `stderr`:

```
?command=<command-syntax>&container=<name>&stderr=true&stdout=true
```

Параметры запроса понятны без пояснений: они определяют команду для выполнения, необходимость включения `stdin`, `stdout`, `stderr` и имя контейнера. WebSocket-соединение устанавливается с этими параметрами строки запроса, и `kube-apiserver` начинает потоковую передачу данных между Meshery/istioctl и соответствующим `kubelet`, как показано на рис. 11.3.

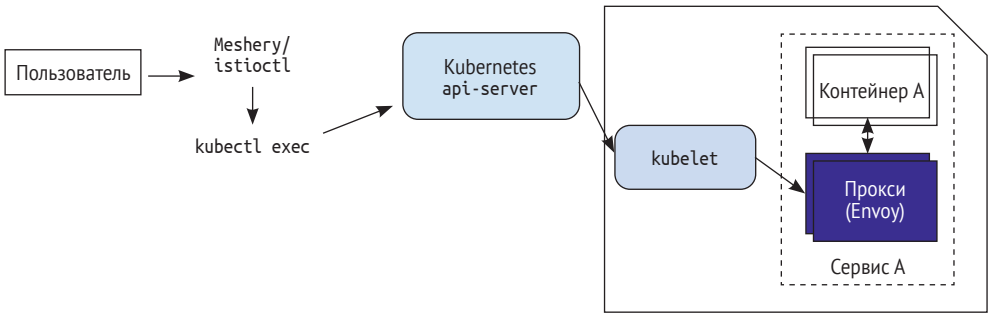

Рис. 11.3 ❖ Как Meshery и istioctl используют kubectl для извлечения конфигурации сетки из прокси

При записи в WebSocket данные посылаются в стандартный ввод (`stdin`), а на принимающей стороне WebSocket извлекаются из стандартных потоков вывода (`stdout`) и ошибок (`stderr`). API определяет простой протокол для мультиплексирования `stdout` и `stderr` через одно соединение. Каждое сообщение, проходящее через WebSocket, имеет однобайтовый префикс, определяющий, к какому потоку относится данное сообщение (см. табл. 11.1).

Таблица 11.1. Основные каналы протокола потоковой передачи данных, используемые для kubectl exec/attach/logs/proxy

Канал	Назначение	Описание
0	stdin	Поток stdin для записи в процесс. Этот поток недоступен для чтения
1	stdout	Поток вывода stdout для чтения stdout из процесса. Этот поток недоступен для записи
2	stderr	Выходной поток stderr для чтения stderr из процесса. Этот поток недоступен для записи

kube-apiserver устанавливает соединение с kubelet, расположенным на узле, где находится рассматриваемый под. Утилита kubelet на этом узле генерирует короткоживущий маркер и пересылает запрос в интерфейс времени выполнения контейнера (*container runtime interface*, CRI). CRI обрабатывает запрос kubectl exec и выдает API-вызов docker exec. Основные каналы протокола stdin, stdout и stderr задаются как соответствующие потоки ввода, вывода и ошибок API-вызова docker exec. kubectl exec/attach/log/proxy могут потребовать долгоживущих соединений с kube-apiserver, поскольку любая из этих команд может потребовать передачи данных в течение всего времени выполнения, а не мгновенного, однократного ответа.

Готовность рабочих нагрузок

Проблемы могут возникнуть не только с компонентами Istio, но и с вновь добавляемым в сетку приложением. Поскольку в сетке уже могут иметься действующие сервисы, необходимо убедиться в совместимости добавляемого приложения с Istio. Ниже приводятся некоторые соображения относительно совместимости.

Конфигурация приложения

Избегайте UID 1337. Убедитесь, что поды не запускают приложения с идентификатором пользователя (UID) 1337. Этот UID зарезервирован для прокси Istio. Нужно избегать подобного конфликта.

Сетевой трафик и порты

Необходим протокол HTTP/1.1 или HTTP/2.0. Приложения должны использовать протокол HTTP/1.1 или HTTP/2.0 для всего HTTP-трафика; HTTP/1.0 не поддерживается.

Портам сервисов должны быть присвоены имена. Чтобы использовать механизм маршрутизации трафика в Istio, для каждого сервиса должны быть определены пары ключ/значение, включающие имя порта и протокол, согласно синтаксису: name: <protocol>[-<suffix>]. Для <protocol> можно использовать одно из следующих значений (в виде строки):

- grpc;
- http;
- http2;

- `https`;
- `mongo`;
- `redis`;
- `tcp`;
- `tls`;
- `udp`.

По умолчанию Istio будет рассматривать трафик как TCP. Если порт явно не использует `udp` для обозначения протокола UDP или если начало имени порта сервиса не совпадает с одним из этих префиксов, Istio будет рассматривать трафик в соответствующий порт как TCP. Как следствие трафик на неименованных портах также рассматривается как TCP. Примерами допустимых имен портов являются `http2-myservice` или `http2`; но имя `http2myservice` не будет правильно опознано.

Такое поведение схоже с поведением Kubernetes, сервисы которого по умолчанию используют протокол TCP, но есть возможность использовать любой другой поддерживаемый протокол (TCP, UDP, HTTPS, proxy, SCTP). Поскольку многие сервисы требуют открытия более чем одного порта, Kubernetes поддерживает возможность определения нескольких портов для одного сервиса. Каждое определение порта может иметь один и тот же или разные протоколы. Однако Kubernetes не требует, чтобы два разных сервиса, ссылающихся на один и тот же порт пода, использовали один и тот же протокол.

Поды должны включать явный список портов, которые прослушивают все контейнеры в нем. Используйте в спецификации контейнера `containerPortconfiguration` для каждого порта. Любые не указанные в списке порты будут использоваться в обход прокси Istio.

Сервисы и развертывание

Привяжите все поды хотя бы к одному сервису. Открывается порт или нет, все поды должны принадлежать, по крайней мере, одному сервису Kubernetes. Для подов, принадлежащих нескольким сервисам Kubernetes, убедитесь, что каждый сервис определяет один и тот же тип протокола при обращении к одному и тому же номеру порта (например, HTTP и TCP).

Содержательная телеметрия с метками Kubernetes. При установке меток приложений и версий мы рекомендуем добавлять явные метки приложений и метки версий. Добавьте метки в спецификацию развертывания подов, устанавливаемых с помощью Kubernetes Deployment. Метки приложений и версий добавляют контекстную информацию к телеметрии, собираемой Istio:

Метка приложения
: Каждая спецификация развертывания должна иметь отдельную метку приложения с осмысленным значением.

Метка версии
: Указывает версию приложения, соответствующую конкретному развертыванию.

Эти метки полезны в качестве контекста, распространяемого в распределенной трассировке.

Поды

Конфигурация пода должна допускать использование NET_ADMIN. Если кластер требует соблюдения политики безопасности подов, поды должны разрешать функцию NET_ADMIN. Как описано в главе 5, при внедрении прокси сетка Istio использует контейнер init, устанавливающий правила iptables в поде для перехвата запросов к контейнерам приложений. Хотя прокси не требует привилегий root для выполнения, но контейнеру init нужны привилегии `cap_net_admin` для установки правил iptables в каждом поде непосредственно перед запуском основных контейнеров пода в сервисной сетке.

Управление правилами iptables требует повышения привилегий через функцию NET_ADMIN в ядре Linux. Поды с включенной поддержкой NET_ADMIN могут управлять сетевой конфигурацией других подов, а также своего узла. Большинство администраторов Kubernetes избегают предоставления такой возможности для арендуемых подов или, по крайней мере, для подов в общедоступных кластерах.

Если используется Istio CNI Plugin (https://oreil.ly/G4TK0), требование к поддержке функции NET_ADMIN не применяется, поскольку плагин CNI (привилегированный под) будет выполнять административные функции от имени контейнера `istio-init`. Ниже показана проверка, включены ли политики безопасности пода в кластере:

```
$ kubectl get psp
No resources found.
```

Здесь показан кластер, для которого не определены политики безопасности подов. Если в кластере определены политики безопасности подов, ищите NET_ADMIN или * в списке функций разрешенных политик для данной учетной записи сервиса. Если учетная запись сервиса Kubernetes не указана в настройках развертывания подов, они запускаются с учетной записью `default` в пространстве имен, в котором установлены поды. Проверку разрешенных возможностей учетной записи сервиса подов можно выполнить следующей командой:

```
$ for psp in $(kubectl get psp -o
    jsonpath="{range.items[*]}{@.metadata.name}{'\n'}{end}");
    do if [ $(kubectl auth can-i use psp/$psp --as=system:serviceaccount:
    <your namespace>:<your service account>) = yes ]; then kubectl get psp/$psp
    --no-headers -o=custom-columns=NAME:.metadata.name,CAPS:.spec.allowed
    Capabilities; fi; done
```

Более подробную информацию о NET_ADMIN см. на странице Istio *Required Pod Capabilities* (https://oreil.ly/33bCH).

Istio: установка, обновление и удаление

Как и в случае со многими механизмами установки и настраиваемыми параметрами, перед началом установки необходимо рассмотреть множество вопросов. Более того, со временем понадобится обновить системы, а в какой-то

момент придется и удалить, учитывая, что развернутые сервисные сетки следуют жизненному циклу обслуживаемых приложений.

Установка

Просто перезапустите сценарий установки. Первоначальные проблемы с установкой иногда решаются путем повторного применения установочного YAML-файла к кластеру Kubernetes (повторный запуск ваших команд установки). Иногда не все компоненты Istio успешно устанавливаются с первой попытки либо из-за потери сетевого трафика, либо из-за отсутствия запланированных ресурсов (например, CRD), которые еще не были полностью подготовлены утилитой `kube-api`.

В таком случае можно увидеть подобное сообщение:

```
unable to recognize "install/kubernetes/istio-demo-auth.yaml":
      no matches for kind
```

Необходимо повторно применить CRD, выполнив следующую команду:

```
$ kubectl apply -f install/kubernetes/helm/istio/templates/crds.yamlt
```

Обновление

Обновления Istio могут проводиться различными способами. Давайте посмотрим на два варианта: Helm с Tiller и без Tiller.

Helm с Tiller

Если установка производилась с помощью `helm install` (с Tiller), как показано ниже:

```
$ helm install install/kubernetes/helm/istio --name istio
       --namespace istio-system
```

то для обновления развертывания Istio можно использовать `helm upgrade`, например:

```
$ helm upgrade istio install/kubernetes/helm/istio
       --namespace istio-system
```

Helm без Tiller

Если Istio была установлена с помощью шаблона Helm (без Tiller), обновить ее с помощью команды `upgrade` можно будет только после установки Tiller. Затем на выбор появляются два варианта. Можно использовать команду `install` для установки исходной версии, а затем команду `upgrade` для обновления. Либо использовать тот же процесс установки с командой `template`, чтобы попутно обновить Istio:

```
$ helm template install/kubernetes/helm/istio --name istio -namespace
     istio-system > istio.yaml
$ kubectl apply -f istio.yaml
```

Процесс `helm template install (upgrade)` использует преимущества процесса скользящего обновления Kubernetes и обновит все компоненты и конфигурационные карты до новой версии. Благодаря такому подходу при необходимости можно откатиться назад, применив YAML-файлы из старой версии.

Удаление

Деинсталляция Istio может быть неполной и оставить некоторые артефакты, которые перечислены далее.

Остаточные CRD

Если после удаления Istio остались CRD, их можно итеративно удалить по одному, как показано ниже:

```
$ for i in install/kubernetes/helm/istio-init/files/crd*yaml;
     do kubectl delete -f $i; done
```

В зависимости от профиля установки развертывание может включать разное число CRD. Вот как можно проверить удаление CRD, принадлежащих Istio:

```
$ kubectl get crds | grep istio
```

После успешного удаления всех CRD результирующий набор должен быть пустым.

Отладка Mixer

Ниже приведена команда для включения журнала отладки Mixer:

```
$ kubectl edit deployment -n istio-system istio-mixer
# Добавить в список аргументов:
       - --log_output_level=debug
```

Получить доступ к журналам Mixer можно с помощью команды `kubectl logs`, как описано ниже.

Для сервиса `istio-policy`:

```
kubectl -n istio-system logs $(kubectl -n istio-system get pods -lapp=policy
     -o jsonpath='{.items[0].metadata.name}') -c mixer
```

Для сервиса `istio-telemetry`:

```
kubectl -n istio-system logs $(kubectl -n istio-system get pods
    -lapp=telemetry -o jsonpath='{.items[0].metadata.name}') -c mixer
```

Использование ControlZ (ctrlz)

Другой вариант – включить отладку Mixer с помощью ControlZ, открыв порт 9876. Для этого перенаправьте порт в ControlZ:

```
$ kubectl --namespace istio-system port-forward istio-[policy/telemetry]-<pod#> 9876:9876
```

Откройте в браузере адрес http://localhost:9876.

Отладка Pilot

Обратившись к API регистрации в компоненте Pilot, можно запросить список хостов и IP-адресов, чтобы получить общую конфигурацию сетки и информацию о конечных точках. Например, следующая команда должна вернуть большой фрагмент JSON:

```
$ kubectl run -i --rm --restart=never dummy --image=tutum/curl:alpine -n
    istio-system --command \
-- curl -v 'http://istio-pilot.istio-system:8080/v1/registration'
```

В репозитории GitHub книги содержится пример (https://oreil.ly/iJD2W) вывода результатов из API регистрации Pilot.

Чтобы собрать журналы Pilot, выполните команду:

```
$ kubectl logs -n istio-system -listio=pilot -c discovery
```

Примеры журналов из контейнера с сервисом discovery компонента Pilot см. в репозитории GitHub книги (https://oreil.ly/1b5W6).

Убедитесь, что сервис `pilot-discovery` `discovery` работает:

```
$ kubectl -n istio-system exec -it istio-pilot-644ff8f78d-p757j -c discovery sh -
# ps -ax
PID TTY   STAT   TIME  COMMAND
  1 ?     Ssl   72:49  /usr/local/bin/pilot-discovery discovery...
```

Отладка Galley

Начиная с Istio v1.1 две основные области ответственности Galley – это синтаксическая и семантическая проверка конфигураций Istio, создаваемых пользователем, и выполнение функций главного реестра конфигураций. Для взаимодействия с другими компонентами Galley использует MCP (https://oreil.ly/DkCL9).

В качестве основного механизма приема и распределения конфигурации в Istio Galley осуществляет проверку пользовательских конфигураций и использует Kubernetes Admission Controller, как показано на рис. 11.4.

```
$ kubectl get validatingwebhookconfigurations
NAME                CREATED AT
istio-galley        2019-06-11T15:33:21Z
```

В сервисе `istio-galley` есть два веб-обработчика, прослушивающих порт 443:

`/admitpilot`
: Отвечает за проверку конфигурации, используемой в Pilot (например, `VirtualService`, настройки аутентификации).

`/admitmixer`
: Отвечает за проверку конфигурации, используемой в Mixer.

Оба веб-обработчика относятся ко всем пространствам имен, и поэтому определение `namespaceSelector` в `pilot.validation.istio.io` и `mixer.validation.istio.io` должно быть пустым.

```
$ kubectl get validatingwebhookconfiguration istio-galley -o yaml
```

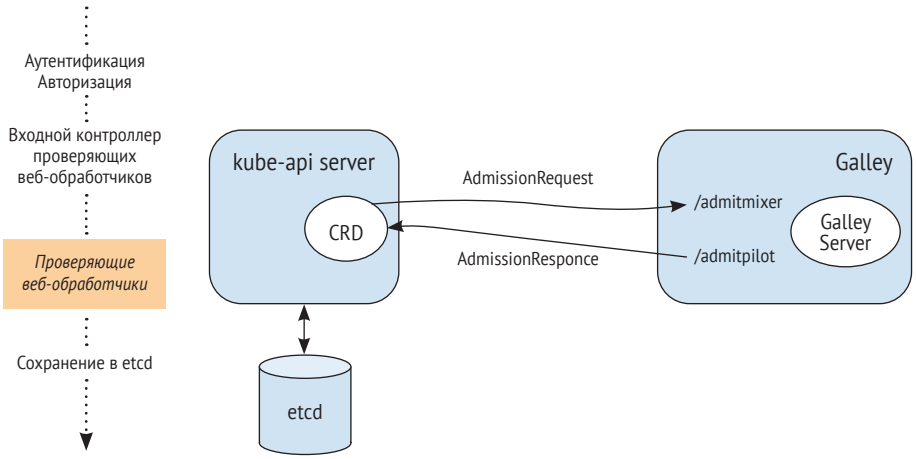

Рис. 11.4 ❖ Два проверяющих веб-обработчика в Galley: pilot.validation.istio.io и mixer.validation.istio.io

Пример вывода `istio-galley`, сгенерированного веб-обработчиком с пустым пространством имен, вы найдете в репозитории GitHub книги (https://oreil.ly/6PWBG).

Отсутствуют хосты с сервисом istio-galley. Если не удается создать или обновить конфигурацию, может оказаться, что Galley работает неправильно. Приступая к устранению неполадок с Galley, прежде всего нужно проверить состояние его пода:

```
$ kubectl -n istio-system get pod -listio=galley
NAME                            READY   STATUS    RESTARTS   AGE
istio-galley-74c6547b94-4vw58   1/1     Running   0          14h
```

Отсутствуют конечные точки для сервиса *istio-galley*. А затем провести аналогичную проверку его конечных точек:

```
$ kubectl -n istio-system get endpoints istio-galley
NAME           ENDPOINTS                                            AGE
istio-galley   10.32.0.17:15014,10.32.0.17:443,10.32.0.17:9901   14h
```

Если поды или конечные точки не готовы, проверьте журналы и состояние подов на наличие признаков, объясняющих, почему поды веб-обработчиков не запускаются и не обслуживают трафик:

```
$ kubectl logs -n istio-system istio-galley-755f8df6cb-zq4p8
```

Пример вывода пода `istio-galley` приведен в репозитории GitHub книги (https://oreil.ly/rpU8s).

Отладка Envoy

Налаживание сетевого взаимодействия является сложной задачей. Слои абстракции и перенаправления еще больше затрудняют отладку сетевых проблем.

Административная консоль Envoy

Административный интерфейс Envoy использует порт 15000 (исчерпывающий список портов см. в главе 4). Доступ к административному интерфейсу Envoy можно получить, настроив с помощью `kubectl` переадресацию портов локальной машины в любой под, обслуживаемый Envoy:

```
$ kubectl port-forward <pod> 15000:15000 &
```

Чтобы остановить фоновое задание переадресации портов, выполните команду `kill %1` (параметр %1 подразумевает отсутствие других фоновых заданий). Для получения списка активных заданий реализуйте команду `jobs`.

Для просмотра конфигурации Envoy в YAML-формате выполните команду:

```
$ curl http://localhost:15000/config_dump | yq r -
```

Также можно открыть полноценную административную консоль в браузере по адресу: http://localhost:15000. Откройте http://localhost:15000/help для ознакомления с описанием имеющихся административных действий. Полное описание всех административных функций Envoy см. в документации по работе и администрированию (https://oreil.ly/7DY65).

Ответы 503 или 404

Если при попытке обратиться к сервису вы получите один из этих двух кодов ошибок, имейте в виду, что они могут быть вызваны разными причинами, в том числе:

- отсутствие связи между Envoy и Pilot. Вот шаги по исправлению положения:
  1) убедитесь, что Pilot запущен. См. раздел «*Отладка Pilot*» главы 7;
  2) используйте `istioctl proxy-status` для подтверждения статуса связи между Pilot и Envoy. Во время нормальной работы каждый xDS будет показывать статус `SYNCED`;
- в манифесте сервисов Kubernetes отсутствует или неправильно указан сетевой протокол. Чтобы исправить проблему, настройте описание сервиса с соответствующим именем, добавив в него определения открытых портов. Список названий протоколов см. в разделе «*Конфигурация приложения*»;
- конфигурация Envoy получает ошибочные настройки маршрутов. Конфигурация `VirtualService` определена неверно. Вот шаги по исправлению положения:
  1) определите пограничный сервис (соседствующий с входным шлюзом);
  2) проверьте журналы Envoy в нем или воспользуйтесь таким инструментом, как Jaeger, чтобы проверить, где происходят сбои.

Внедрение прокси

Затруднения с внедрением прокси могут свидетельствовать о наличии других проблем в окружении. Вот три фактора, способных влиять на возможность внедрения прокси:

- настройка `namespaceSelector` в определении веб-обработчика;

- политика по умолчанию;
- в каждом поде присутствуют переопределяющие аннотации.

Проверим следующие пункты.

Поды не должны запускать приложения с UID 1337

Контейнеры приложений не должны работать с UID 1337, поскольку этот UID использует прокси Envoy. В настоящее время Envoy *должен* работать как `istio-proxy` с UID 1337, и этот аспект не является централизованно конфигурируемым параметром развертывания.

Входной контроллер изменяющих веб-обработчиков включен

Ошибка, такая как:

```
error: unable to recognize "istio.yaml": no matches for
      admissionregistration.k8s.io/, Kind=MutatingWebhookConfiguration
```

говорит о том, что у вас, скорее всего, используется версия Kubernetes 1.9 или более ранняя, которая не поддерживает входной контроллер изменяющих веб-обработчиков, либо он выключен.

Присутствует метка istio-injection

Как обсуждалось в главе 5, чтобы поды извлекали выгоду из автоматического внедрения прокси, их пространство имен должно иметь метку `istio-injection`, потому что веб-обработчик, осуществляющий внедрение, вызывается только для подов, созданных в пространствах имен с меткой `istio-injection=enabled`. Проверим, что затрагиваемые поды находятся в пространстве имен с этой меткой:

```
$ kubectl get namespace -L istio-injection
NAME              STATUS   AGE   ISTIO-INJECTION
default           Active   39d   enabled
istio-system      Active   13h
kube-node-lease   Active   39d
kube-public       Active   39d
kube-system       Active   39d
```

Правильная настройка namespaceSelect в веб-обработчике

`namespaceSelect` в веб-обработчике определяет, будет ли он вызываться для целевого пространства имен. Вот как выглядит определение `namespaceSelector` для *включения* веб-обработчика:

```
$ kubectl get mutatingwebhookconfiguration istio-sidecar-injector -o yaml
       | grep "namespaceSelector:" -A5

  namespaceSelector:
    matchLabels:
      istio-injection: enabled
  rules:
 - apiGroups:
   - ""
```

Чтобы *исключить пространство имен из обработки, определение* `namespaceSelector` должно выглядеть следующим образом:

```
namespaceSelector:
    matchExpressions:
   - key: istio-injection
      operator: NotIn
      values:
     - disabled
  rules:
 - apiGroups:
   - ""
```

Отладка Citadel

Как и при отладке других проблем, просмотр журналов и событий компонента Citadel может оказаться полезным для диагностики возможных проблем.

```
$ kubectl logs -l istio=citadel -n istio-system
$ kubectl describe pod -l istio=citadel -n istio-system
```

Может случиться так, что под `istio-citadel` не запущен, поэтому первым делом следует проверить его статус:

```
$ kubectl get pod -l istio=citadel -n istio-system
NAME                              READY   STATUS    RESTARTS   AGE
istio-citadel-678b7c5cd4-ndn4n   1/1     Running   0          13h
```

Повторно разверните под `istio-citadel`, если он не находится в состоянии `Running`.

Совместимость версий

Нельзя смешивать компоненты Istio, утилиту `istioctl` и приложение-образец Bookinfo разных версий. Например, нельзя одновременно запустить Istio v1.1, `istioctl` v.1.0 и Bookinfo v1.2. То же обычно относится к номерам исправлений, например нужно избегать запуска Istio v1.1.4 при использовании `istioctl` v1.1.3. Чтобы проверить версии всех компонентов уровня управления в Istio, достаточно выполнить команду:

```
$ istioctl version --remote -o yaml
```

В качестве альтернативы можно посмотреть тег изображения в любом из компонентов уровня управления. Вот как это сделать, если использовать компонент Pilot:

```
$ kubectl get deployment istio-pilot -o yaml -n istio-system | grep image:
    | cut -d ':' -f3 | head –1
```

Приведенные инструменты отладки и примеры их использования должны быть полезны, но это далеко не полный список. Существуют и другие сценарии неисправностей и их устранения, а также, к счастью, другие инструменты. С увеличением количества сервисных сеток растущий спрос на них привел к появлению ряда полезных утилит и инструментов управления (https://oreil.ly/U6Vwa). Ожидается, что будет расти количество предлагаемого программного обеспечения уровня управления.

Глава 12

Вопросы развертывания приложений

Инженеры используют сервисные сетки по многим причинам, не в последнюю очередь для повышения надежности сервисов, предоставляемых пользователям. Ключевым фактором повышения надежности приложений в Istio является понимание характеристик надежности *самой* Istio. Логично, что их надежность взаимосвязана. Хотя использование Istio (или любой другой сервисной сетки) позволяет резко облегчить работу всех приложений (в сетке и вне ее), введение дополнительных компонентов в систему, таких как прокси, образующих сетку сервисов, порождает новые виды потенциальных сбоев. В этой главе рассматриваются вопросы, связанные с *надежным* развертыванием Istio и рабочих нагрузок в ней.

В предыдущих главах рассказывалось, как Istio увеличивает надежность приложений (в частности, за счет обнаружения аномалий, исключения из работы аварийных сервисов, повторов и т. д.). Было показано, как Istio позволяет управлять трафиком в сетке и помогает получить наглядное представление о развертывании, генерируя телеметрию приложений. Рассмотрены особенности Istio, предотвращающие определенные классы сбоев, но мы не разобрали подробно, как Istio помогает ослабить влияние наиболее распространенных источников сбоев: развертывание новых версий ваших приложений. К счастью, способность контролировать трафик и отслеживать его поведение является *именно тем*, что нужно для минимизации рисков при (повторном) развертывании приложений.

Далее мы представим некоторые соображения по развертыванию компонентов уровня контроля, а затем рассмотрим конкретный пример *канареечного развертывания* (*canary deployment*) приложения. Знакомясь с методами безопасного развертывания приложений, мы попутно посмотрим, как взаимодействуют друг с другом компоненты Istio во время работы и как их поведение влияет на надежность.

Соображения об уровне управления

Разные компоненты Istio могут испытывать разные проблемы, по-разному отражающиеся на работе сетки. Чтобы лучше понять эти проблемы и их проявления, для каждого компонента уровня управления мы рассмотрим общий

набор возможных отказов (недоступность рабочих нагрузок для компонентов, невозможность взаимодействий между самими компонентами и т. д.), любые отказы, характерные для данного компонента, и их отражения на работе сетки. Мы охватим только самые основные виды отказов, потому что перечислить их все просто невозможно. Наша цель – вооружить вас знаниями о моделях поведения и зависимостях и объединить их с информацией, содержащейся в главах, посвященных конкретным компонентам, в надежде, что это поможет вам понять причины появления других отказов.

В этом разделе рассматриваются в основном отказы, обусловленные сегментированием *сети*, которое действительно является источником многих видов сбоев. Например, недоступность хранилища конфигураций для Galley может вызываться многими причинами, в том числе: наличие фактической границы между сетями; хранилище генерирует ошибку 500; хранилище вообще не принимает соединений или обслуживает запросы с недопустимо большими задержками. Независимо от первопричины, с точки зрения надежности компонентов уровня управления все эти сбои похожи тем, что один компонент системы не может вовремя получить необходимые данные.

Также стоит обсудить обновление компонентов Istio на высоком уровне. Исторически обновление Istio всегда было болезненным процессом, чреватым ошибками (простоями). Начиная с версии Istio 1.0 разработчики начали предпринимать некоторые шаги по обеспечению бесперебойного процесса обновления, но работы в этом направлении еще не завершены. Например, после выпуска следующей версии обнаружилось, что для некоторых пользовательских конфигураций, развернутых небольшим количеством людей, обновление с 1.0 до 1.1 привело к нарушению нормального движения трафика в сетке. По результатам этого инцидента и других исследований в рамках проекта было начато несколько долгосрочных инициатив, касающихся возможности обновления компонентов уровня управления и сетки Istio в целом (уровень управления, агенты узлов и уровень данных). Эти усилия в первую очередь сосредоточены на способности компонентов уровня управления осуществлять самопроверку. На момент написания данного текста эта работа еще не завершена. Для каждого компонента, рассмотренного в текущей главе, мы представим свои рекомендации по обновлению, но конкретные известные проблемы перехода с одной версии на другую приводить не будем.

Galley

Galley отвечает за рассылку конфигурации другим компонентам уровня управления в Istio. Когда компонент Galley испытывает проблемы (не имеет доступа к своим источникам данных и другим компонентам уровня управления в Istio, снова и снова сталкивается с ошибками или другими проблемами), основным симптомом является невозможность добавления новых конфигураций в сетку. Несмотря на то что сетка способна продолжать устойчиво функционировать в текущем состоянии, она не позволяет вносить изменения в конфигурацию, пока не будет восстановлена работоспособность Galley.

Типичная сетка Istio имеет небольшое количество экземпляров Galley, обычно один или два (если используется пара высокой доступности) на каж-

дый развернутый уровень управления. Galley не обязательно должен быть «рядом» (с точки зрения сетевых задержек) с остальными компонентами уровня управления. Более высокая задержка между Galley и остальной частью уровня управления означает более длительное время, в течение которого пользовательские конфигурации вступают в силу; даже один глобальный экземпляр Galley, управляющий экземпляром уровня управления на другом конце мира, будет иметь достаточно низкую задержку, чтобы сетка функционировала правильно.

Недоступность хранилища конфигураций

Когда Galley не может получить доступ к хранилищу конфигураций, никакая новая конфигурация не поступит из Galley в другие компоненты Istio. Все компоненты Istio кешируют свое текущее состояние в памяти – Galley не является исключением; пока сам Galley функционирует, он продолжит обслуживать текущую конфигурацию остальной части уровня управления, пытаясь восстановить соединение с хранилищем конфигураций. Если в течение этого времени Galley прекратит работу и будет перезапущен, он не сможет обслуживать остальные части уровня управления до восстановления соединения с хранилищем конфигураций.

Одним из способов ослабить влияние этой категории сбоев является кеширование конфигураций в локальном хранилище. Galley может принимать конфигурации из локальной файловой системы в дополнение к удаленным источникам (таким как Kubernetes API Server). Базовый набор конфигураций может храниться в локальной файловой системе (устойчивой к перезагрузкам Galley), которую Galley может использовать в периоды попыток установить соединение с удаленными хранилищами конфигураций. В крайнем случае, в системах, где изменения происходят редко, вполне можно запустить Galley, используя *только* источник конфигураций в файловой системе.

Когда Galley развернут в Kubernetes, он также выступает в роли *входного контроллера проверяющих веб-обработчиков* (*Validating Admission Controller*), то есть Galley отвечает за проверку конфигураций, отправляемых в Kubernetes API Server. В этом случае конфигурации, переданные в Kubernetes, будут отклонены в момент передачи (т. е. `kubectl apply` даст отказ).

Недоступность для других компонентов Istio

Когда Galley не имеет доступа к другим компонентам уровня управления, сам он продолжит работать, но другие компоненты не будут получать обновлений конфигурации. (См. предыдущий раздел для получения подробной информации о неисправностях, когда Galley не может получить доступ к ним.)

Недоступность для рабочих нагрузок сетки

Galley не взаимодействует напрямую с рабочими нагрузками или узлами, развернутыми в сетке. Он взаимодействует только с другими компонентами уровня управления Istio и собственным хранилищем конфигураций. Недоступность рабочих нагрузок в сетке для компонента Galley не является проблемой, если сохраняется связь с остальными компонентами уровня управления в Istio.

Обновления

В связи с характером отказов Galley обновление Galley на месте или скользящее обновление протекают без особых проблем (обновление на месте, по сути, то же самое, что и временная недоступность в данном контексте, поскольку предыдущее задание отменяется, а вместо него создается новое). Другие компоненты Istio найдут Galley по DNS и попытаются снова подключиться к нему после разрыва соединения. В рамках проекта Istio выполняется перекрестное тестирование (в ходе которого проверяется совместимость компонентов уровней управления и данных двух соседних версий), чтобы убедиться, что обновление (например, обновление версии 1.0 до 1.1) не нарушит работоспособность. Однако обновление через версию может быть небезопасно (например, обновление версии 1.0 до 1.2).

Pilot

Pilot отвечает за настройку уровня данных во время работы сервисной сетки. Когда он недоступен, изменить текущую конфигурацию сетки не получится; вы не сможете запустить новые рабочие нагрузки, а существующие будут продолжать работать в конфигурации, которая была перед потерей связи с Pilot. Прокси будут хранить прежнюю конфигурацию до восстановления соединения с Pilot или до их перезапуска. Любые изменения конфигурации уровня данных, выполняемые во время работы (не во время загрузки), например обновление настроек политики или телеметрии по всей сетке, также не будут применяться до восстановления связи с Pilot.

Типичная сервисная сетка содержит несколько экземпляров Pilot. Pilot, как и другие компоненты уровня управления Istio, является сервисом без состояния, который по мере необходимости можно горизонтально масштабировать. На самом деле это рекомендуемая практика при развертывании в производственном окружении. Нижележащие платформы, такие как Kubernetes, делают подобные рабочие конфигурации относительно простыми со стандартной поддержкой горизонтального автоматического масштабирования подов. Задержка между реестром сервисов, Pilot и прокси, находящимися под управлением Pilot, критична при обновлении конечных точек сетки, поскольку рабочие нагрузки могут перераспределяться и запускаться на разных узлах. Удержание этой задержки на низком уровне обеспечивает лучшую общую производительность сетки. Как правило, Pilot должен быть «рядом» (с малой задержкой доступа) с прокси, которым он предоставляет конфигурацию. Производительность Pilot менее чувствительна к удаленности от *его собственных* источников конфигураций.

Стоит отметить один неприятный момент, связанный с масштабированием Pilot. Поскольку Envoy для передачи конфигурации использует gRPC-поток, распределение нагрузки по запросу между экземплярами Pilot отсутствует. Вместо этого каждый Envoy, находящийся в сетке Istio, старается взаимодействовать с первоначально настроенным экземпляром Pilot и не будет связываться с другими экземплярами, пока его экземпляр Pilot не разорвет связь (или не исчезнет, разорвав ее). По этой причине масштабирование Pilot может оказаться непростым делом. Обычно на практике запускают несколько экземпляров, а затем останавливают перегруженный экземпляр Pilot, чтобы

заставить прокси-серверы Envoy перераспределить нагрузку между вновь развернутыми экземплярами Pilot. Эта проблема известна разработчикам и будет решена в следующих выпусках Istio, где Pilot будет сбрасывать нагрузку, закрывая некоторые соединения и таким способом вынуждая Envoy переподключиться или найти новый экземпляр.

Недоступность хранилища конфигураций

Как и другие компоненты Istio, Pilot кеширует свое текущее состояние в памяти. Конфигурация в Pilot разделена на две категории: сетевая конфигурация Istio и состояние окружающей среды из реестров сервисов.

Когда Pilot не может связаться со своим хранилищем конфигураций для получения конфигурации Istio (с Galley или Kubernetes API Server в старых версиях Istio), он будет продолжать использовать текущее кешированное состояние. Новые рабочие нагрузки можно будет планировать, а их прокси будут получать конфигурации, основанные на текущей конфигурации Pilot. Если Pilot сам перезапустится и не будет иметь возможности связаться с хранилищем конфигураций (или запустится новый экземпляр Pilot), он не сможет обслуживать конфигурации взаимодействующих с ним прокси.

Если Pilot не сможет связаться со своими реестрами сервисов, он точно так же будет использовать свое текущее состояние из памяти, пытаясь подключиться к источнику. В течение этого времени новые сервисы, введенные в сетку (например, в результате создания нового ресурса Service в Kubernetes), не будут доступны рабочим нагрузкам в сетке. Аналогично маршруты к новым конечным точкам не будут передаваться в прокси Envoy – это означает, что если Pilot потеряет связь со своим механизмом обнаружения сервисов, рабочие нагрузки, перераспределяемые между узлами по расписанию или иным образом, не будут удалены из наборов нагрузок других прокси-серверов в сетке, в результате чего они будут продолжать отправлять трафик в эти, уже нерабочие, конечные точки. Настройка автоматических повторов с обнаружением аномалий по всей сетке поможет сохранить прохождение трафика приложений во время подобных переходных сбоев. И как и прежде, если Pilot сам перезапустится в момент недоступности реестра сервисов, *никакие* сервисы из этого реестра в сетке не будут доступны.

Недоступность для других компонентов Istio

Pilot получает свое сетевое удостоверение от Citadel, как и другие компоненты Istio; невозможность связаться с Citadel, когда компоненту Pilot понадобится новое удостоверение (например, при запуске нового экземпляра или по истечении срока действия текущего удостоверения), приведет к недоступности Pilot. Устранение неполадок при невозможности связи с Galley было описано в предыдущем разделе и разделе «Galley» (см. также раздел «*Citadel*» ниже). Pilot не связывается напрямую с Mixer, поэтому возможность связаться с сервисом телеметрии или политикой Mixer не влияет на работу Pilot.

Недоступность рабочих нагрузок сетки

Когда рабочие нагрузки в сетке не могут связаться с Pilot, их прокси не могут получать новые конфигурации. В частности, они не смогут получать обновле-

ния сетевой конфигурации, политики авторизации сервис–сервис и маршруты к новым сервисам и конечным точкам. Как и все остальные компоненты Istio, прокси кешируют свою текущую конфигурацию и будут продолжать использовать ее до тех пор, пока не восстановят соединение с одним из экземпляров Pilot. Пока Pilot недоступен, новые запланированные рабочие нагрузки не получат никакой конфигурации и поэтому вообще ни с кем не смогут взаимодействовать (Istio настраивает прокси так, что они закрываются при отказе). Новые запланированные рабочие нагрузки также не смогут получить удостоверения от Citadel, так как их удостоверения пересылаются через Pilot. Существующие рабочие нагрузки будут по-прежнему выполняться с текущими удостоверениями (имевшимися на момент потери связи с Pilot) и смогут получать от Citadel новые удостоверения, даже если Pilot не будет доступен.

Как обсуждалось ранее, настройка повторных попыток, обрыва цепи и обнаружения аномалий по умолчанию по всей сетке может помочь уменьшить влияние переходных сбоев в работе Pilot, из-за которых возникают устаревшие рабочие конфигурации. Одним из ключевых преимуществ балансировки нагрузки на стороне клиента является возможность выбора серверов для взаимодействий, в зависимости от их доступности со стороны клиента. Сетка продолжит функционировать даже после потери связи с Pilot, если конфигурация остальной части развертывания изменяется редко и в сетке используются продуманные политики поддержки надежности.

Обновления

Обновление Pilot, как и обновление Galley, в действующей сетке выглядит как сбой. Уровень данных продолжает работать, но обновления (например, запуск новых и остановка старых рабочих нагрузок) не будут передаваться в экземпляры Envoy. Как отмечалось в начале раздела, Envoy не выполняют балансировку запросов между экземплярами Pilot, поэтому одного лишь развертывания нового Pilot рядом со старым недостаточно: действительно необходимо выполнить скользящее обновление (в котором новая версия заменяет старую, забирая управление Envoy на себя) или вручную остановить старые экземпляры Pilot по мере запуска новых.

Эти ограничения отчасти объясняются ограничениями Envoy. Envoy принимает загрузочную конфигурацию, которая является неизменной; частью этой конфигурации является адрес Pilot. Полный комплект конфигурации Envoy не содержит настроек, определяющих порядок взаимодействий с Pilot, а чтобы обновить загрузочную конфигурацию, прокси нужно перезапустить. Например, невозможно использовать собственную конфигурацию Envoy (или Istio), чтобы заставить Envoy переключиться на новый экземпляр Pilot. (Это осознанное проектное решение, исключающее целый класс отказов; уровень управления конфигурирует Envoy так, что он не может снова выйти на связь для получения корректирующей конфигурации). Поэтому методы, доступные для постепенного внедрения новых версий Pilot, ограничены.

Mixer

Mixer может работать в двух разных режимах, и в каждом из них могут возникать свои проблемы. В режиме *Policy* (*применения политик*) Mixer обслуживает

запросы, поэтому сбои в нем оказывают прямое влияние на пользовательский трафик. Это связано с тем, что сбои в применении политик всегда вызывают блокировку запросов. В режиме *Telemetry* (*сбора телеметрии*) Mixer не обслуживает запросы, и сбои в нем влияют только на способность сетки производить телеметрию (это, конечно, может привести к срабатыванию всех видов оповещений о сбоях, поскольку телеметрия не приходит от некоторых частей сетки).

В следующих разделах обсуждаются только отказы, общие для обоих режимов, и отдельно описываются особые соображения для каждого режима.

Фактически Mixer действует как маршрутизатор: его конфигурация просто описывает, как создавать значения из набора данных и куда пересылать эти значения. В результате в обоих режимах работы Mixer обычно связывается с набором удаленных сервисов для каждого полученного запроса. Это означает, что Mixer особенно чувствителен к границам в сети и способствует увеличению сетевых задержек намного больше, чем другие компоненты уровня управления. Также важно отметить, что в современной модели развертывания Mixer сетка Istio предполагает, что удаленные сервисы, с которыми взаимодействует Mixer, находятся не в самой сетке. Istio делает это предположение по ряду причин, например чтобы избежать рекурсивных вызовов. (Mixer посылает данные телеметрии коллектору, который запускает свой прокси для отправки телеметрии в Mixer; Mixer посылает эту телеметрию коллектору, который запускает свой прокси.) Спасение заключается в том, что Mixer, в отличие от Pilot, сам находится за прокси. Это позволяет использовать конфигурацию Istio для управления взаимодействием Mixer с удаленными сервисами (включая конфигурацию отказоустойчивости, такую как разрыв цепи и автоматические повторные попытки).

Недоступность хранилища конфигураций

Mixer, как и другие компоненты Istio, хранит свою текущую конфигурацию в памяти. Отсутствие связи с Galley означает, что он не получит новые конфигурации, но это не помешает Mixer продолжать работать в текущей конфигурации. Если Mixer остановится и будет перезапущен в период недоступности Galley, он будет обслуживать только свою конфигурацию по умолчанию. Далее мы обсудим поведение Mixer при использовании конфигурации по умолчанию отдельно для каждого режима. Для каждого режима можно определить для Mixer свою конфигурацию по умолчанию, сохранив ее в локальной файловой системе.

Политики Mixer. Политику по умолчанию для Mixer можно определить при установке и потребовать открывать или закрывать соединения, отсутствующие в конфигурации. Istio поставляется с конфигурацией открыто-по-умолчанию, не применяющей никакой политики. При такой конфигурации проверка вызова прокси всегда будет разрешать трафик. Сделано это для того, чтобы установка Istio в существующий кластер не вызвала нарушений в передаче трафика. Если Mixer используется для применения обязательных политик, следует установить конфигурацию закрыто-по-умолчанию. Авторизация является хорошим примером политики, которую многие администраторы считают обязательной. Другие администраторы вполне терпимо относятся к частоте следования запросов и даже к некоторым злоупотреблениям в течение короткого

времени, предпочитая бесперебойно обслуживать пользовательский трафик, поэтому в данном контексте такие политики могут рассматриваться как «необязательные».

Телеметрия Mixer. В этом режиме Mixer не обслуживает запросы, поэтому сбои в нем не влияют на трафик в сетке. В отсутствие настроек телеметрии Mixer будет принимать данные в API *report* от прокси в сетке, но сам не будет генерировать телеметрию (в отсутствие настроек API *report в* Mixer ничего не делает). Такая ситуация может вызвать «информационный шторм», потому что из-за отсутствия метрик одновременно могут рассылаться оповещения для каждого затронутого сервиса.

К сожалению, Mixer не поддерживает возможности чтения конфигурации одновременно из локальной файловой системы и с удаленного конфигурационного сервера. Поэтому при использовании удаленного сервера (т. е. Pilot) в настоящее время нельзя установить конфигурацию по умолчанию для Mixer, кроме флагов высокого уровня (например, политику открыто или закрыто по умолчанию). Разработчики знают об этих проблемах и планируют заняться ими вплотную в следующих версиях Istio.

Недоступность для других компонентов Istio

Другие компоненты Istio практически не взаимодействуют с Mixer и не зависят от него. Недоступность Mixer отражается на компонентах Istio (например, Pilot), которые работают за прокси, применяющими политику, точно так же, как на любой другой рабочей нагрузке в сетке. В остальном между компонентами и Mixer не существует специфических зависимостей во время выполнения.

Недоступность для рабочих нагрузок сетки

Потеряв возможность взаимодействовать с Mixer, рабочие нагрузки становятся неработоспособными, что вполне ожидаемо. Прокси будут принудительно применять политику по умолчанию, заданную во время установки (открыто или закрыто при сбое). В отношении телеметрии прокси будут стараться сохранить как можно больше данных, ожидая возможности переслать их в Mixer. Для хранения телеметрии прокси используют кольцевой буфер фиксированного размера, поэтому при большом объеме накопленных данных часть из них неизбежно будет потеряна. Размер буфера настраивается флагами, которые передаются в прокси при запуске. Этот элемент конфигурации в настоящее время не поддерживается в Helm.

Обновления

Для связи с Mixer прокси используют унарный протокол, а для взаимодействий с интерфейсом xDS Pilot – потоковый протокол gRPC. То есть сообщения отправляются по отдельности, а не в потоковом режиме. Поэтому каждый запрос может подвергаться балансировке. Это значительно облегчает развертывание новых версий Mixer, позволяя использовать Envoy и обычные примитивы Istio для автоматического развертывания новых версий.

При обновлении политик Mixer остерегайтесь всплеска задержек, который появляется при переходе на новый экземпляр. Mixer довольно агрессивно ке-

ширует решения политик, и когда трафик переходит на новый экземпляр, появляется всплеск задержек, поскольку запросы проверок не попадают в кеш и для принятия решений приходится вызывать сервисы, реализующие политики. Это означает, что сервисы политик также получат рост трафика в период прогрева кеша нового экземпляра. Это вряд ли повлияет на 50-й процентиль задержек, но определенно окажет влияние на 99-й процентиль.

Для телеметрии Mixer нет никаких особых соображений, хотя, учитывая, что конфигурация времени выполнения загружается в отложенном режиме, можно ожидать высокой задержки для нескольких первых запросов к API *report*. Поскольку *report* работает асинхронно и вне полосы пропускания трафика, это не должно вызвать замедление трафика пользователей.

Citadel

Citadel отвечает за выдачу и ротацию удостоверений в сервисной сетке. Когда компонент Citadel окажется недоступен, ничего не произойдет до истечения срока действия сертификатов. Затем по всей сетке будут наблюдаться сбои при попытке установить связь. Пока Citadel находится в нерабочем состоянии, существующие соединения будут работать, но новые рабочие нагрузки не смогут включиться в работу, и не будут устанавливаться новые соединения (ни с новыми, ни с существующими рабочими нагрузками), если сроки действия сертификатов рабочих нагрузок истекут в период отсутствия связи с Citadel. При использовании mTLS для взаимодействий между компонентами уровня управления Istio, что является настройкой по умолчанию, запуск всех остальных компонентов уровня управления зависит от запуска Citadel. Это обусловлено тем, что прежде чем установить соединение, каждый компонент уровня управления должен получить удостоверение от Citadel.

Недоступность хранилища конфигураций

Citadel, как и другие компоненты, будет продолжать работать в текущем состоянии, пока не сможет установить связь с хранилищем конфигурации. В отличие от большинства других компонентов уровня управления, Citadel получает мало настроек от Galley и намного теснее связан со своими источниками конфигураций (в частности, с Kubernetes API Server), используемыми для обнаружения сущностей, которым он будет присваивать сертификаты. При недоступности этих источников Citadel не сможет генерировать сертификаты для новых рабочих нагрузок и выполнять ротацию сертификатов в существующих.

Недоступность для других компонентов Istio

С точки зрения Citadel другие компоненты Istio выглядят так же, как и обычные рабочие нагрузки сетки: ни один из компонентов уровня управления не является особенным. В следующем разделе это обсуждается подробнее.

Недоступность для рабочих нагрузок

При невозможности связи с Citadel рабочие нагрузки в сетке не смогут получить новые удостоверения. Это означает, что запускаемые новые рабочие нагрузки

не смогут взаимодействовать в сетке ни с чем, требующим mTLS, поскольку Citadel не может выдать им удостоверения. Существующие рабочие нагрузки с истекающими сертификатами не смогут установить новые соединения, однако существующие соединения останутся открытыми и действительными до получения нового сертификата или их закрытия. Этот сбой проявится как ошибка установки TCP-соединения, вызывающая пугающее сообщение «Соединение сброшено удаленной стороной» (*Connection reset by peer*). Если установить сертификаты с более коротким сроком действия, скажем в несколько часов, в моменты ротации сертификатов эта проблема может проявляться как ошибка 503 (из-за ошибки сброса соединения). Разработчики Istio делают все возможное, чтобы устранить такие ошибки с кодом 503, тем не менее некоторые крайние случаи ротации сертификатов остаются последним источником этих ошибок.

Также следует отметить, что во избежание проблем «громоподобного стада» отдельные рабочие нагрузки будут запрашивать обновленные сертификаты через случайные промежутки времени до истечения срока действия сертификата (чтобы предотвратить запрос нового сертификата каждым приложением, например, каждый час). Это означает, что при сбое связи с Citadel различные рабочие нагрузки одного и того же сервиса окажутся недоступными, в то время как другие будут оставаться на связи.

Обновления

Случайная природа запросов на обновление сертификатов не дает простого способа «запланировать время простоя» компонента Citadel, даже несмотря на то, что он очень редко вызывается приложениями в сетке. Однако рядом с существующим экземпляром Citadel можно развернуть новый, а затем осушить существующий (или вообще остановить его, например, в Kubernetes), чтобы принудительно переключить трафик на новый экземпляр. Учитывая, что Citadel при запуске усердно пытается создавать сертификаты для всех сущностей в сетке и не может выдавать сертификаты, не завершив этот процесс, вероятно, лучше установить новую версию Citadel и дождаться его прогрева перед переключением трафика.

Пример из практики: канареечное развертывание

Информация, изложенная в предыдущих разделах о взаимодействии компонентов уровня управления Istio, должна помочь построить мысленную модель взаимодействий в сетке в целом отражения на приложениях отказа каждого компонента. С этими знаниями можно приступать к планированию защищенной и надежной сетки Istio. Теперь возникает вопрос: как использовать возможности Istio для повышения надежности приложений в рабочей системе?

Практически каждый простой является результатом некоторых изменений. Контроль за внедрением изменений в работу и их действием имеет решающее значение для контроля простоев. Если говорить о сервисах, то развертывание нового двоичного файла является наиболее распространенным изменением, сопровождающимся коротким этапом обновления конфигурации данного

сервиса. Мы рекомендуем одинаково внимательно относиться к изменениям в конфигурации и в двоичных файлах. В настоящее время развертывание новых двоичных файлов может вызывать больше сбоев, но по мере развития приложения, скорее всего, основной причиной большинства сбоев станет развертывание новых конфигураций. Обобщая подходы к развертыванию тех и других, можно создать единый набор практических методов и процессов для смягчения последствий сбоев в работе независимо от их первопричины. Хочется надеяться, что ситуация повторения сбоев, с которыми вы уже сталкивались, весьма нетипична (потому что проблема, вызвавшая этот сбой, решена и известна). Из этого следует, что не существует универсального подхода к решению проблем; однако в чрезвычайных ситуациях, когда известна последовательность действий, это экономит время, деньги и бюджет ошибок. Логично? Итак, как же безопасно развернуть новый двоичный файл в Istio?

Канареечное развертывание – это процесс постепенного развертывания изменений, тщательного контроля их воздействия на сущности, которые они затрагивают. Например, во многих компаниях следующая версия продукта тестируется на ее сотрудниках, прежде чем открыть к ней доступ для клиентов; это и называется канареечным развертыванием. В Istio имеется широкий спектр параметров маршрутизации трафика к любым группам экземпляров сервиса, многие из которых рассматривались в главе 8. В этой же главе мы разберем пример отвода доли трафика для проверки новой версии сервиса. Для подготовки тестовой среды, позволяющей увидеть канареечную (новую) версию в реальном времени, в примере 12.1 сначала создается простое развертывание в Kubernetes (см. httpbin-svc-depl.yaml (https://oreil.ly/lmmyS) в репозитории GitHub книги).

Обратите внимание на метку `apiVersion: v1`. В сообществе Kubernetes принято использовать метку `version` для обозначения версий развертывания, а метку приложения – для выбора набора установок сервиса. Многие инструменты, включая собственные дашборды Istio, при построении графиков обслуживания используют метку `version`. Применим эту же метку для управления маршрутизацией трафика.

Пример 12.1 ❖ Сервис Kubernetes и описание развертывания приложения httpbin

```
apiVersion: v1
kind: Service
metadata:
  name: httpbin
  labels:
    app: httpbin
spec:
  ports:
 - name: http
    port: 8000
    targetPort: 80
  selector:
    app: httpbin
---
apiVersion: extensions/v1beta1
kind: Deployment
```

```
metadata:
  name: httpbin-v1
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: httpbin
        version: v1
    spec:
      containers:
     - image: docker.io/kennethreitz/httpbin
        imagePullPolicy: IfNotPresent
        name: httpbin
        ports:
       - containerPort: 80
```

Внутри кластера трафик можно отправлять в сервис `httpbin`, в результате чего должны появиться метрики. Для простоты можно открыть доступ к `httpbin` на `Gateway`, как показано в примере 12.2, чтобы получить возможность обращаться к ней извне кластера (т. е. с локальной машины). См. httpbin-gw-vs.yaml в репозитории GitHub книги (https://oreil.ly/PwWAI) для следующего примера:

Пример 12.2 ❖ Istio Gateway и VirtualService, открывающие доступ к сервису httpbin на входном шлюзе istio-ingressgateway

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: httpbin-gateway
spec:
  selector:
    istio: ingressgateway
    servers:
   - port:
        number: 80
        name: http
        protocol: HTTP
      hosts:
     - "*"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
 - "httpbin.svc.default.cluster.local"
  gateways:
 - httpbin-gateway
  http:
 - route:
```

```
  - destination:
      host: httpbin
    port:
      number: 8000
```

Настройка `hosts: "*"` в этом примере выбрана с целью упростить проверку шлюза с помощью `curl` и произвольного IP-адреса. Если известно доменное имя шлюза `istio-ingessgateway` или IP-адрес, его можно использовать в поле `hosts` в определениях `Gateway` и `VirtualService` (вместо `"*"`).

Для проверки можно выполнить команду `curl` на локальной машине, как показано в примере 12.3.

Пример 12.3 ❖ Команда curl направляет запрос в сервис httpbin

```
$ curl ${ISTIO_INGRESS_IP}/status/200
```

Далее, для канареечной проверки новой версии приложения `httpbin` можно просто развернуть ее. Это приведет к равномерному распределению нагрузки между всеми экземплярами `httpbin` в кластере. При наличии большого числа экземпляров `httpbin`, обслуживающих большой объем трафика, такое распределение может быть приемлемо. Но чаще всего это не так. Поэтому мы настроим Istio так, чтобы она отправляла трафик действующей версии, а затем постепенно будем переводить трафик на новую версию.

Для этого создадим несколько ресурсов: во-первых, `DestinationRule` для сервиса `httpbin`, описывающий подмножества сервисов. Далее используем эти подмножества в `VirtualService`, чтобы направить трафик в v1 на время развертывания v2, как показано в примере 12.4 (см. httpbin-destination-v2.yaml в репозитории GitHub книги (https://oreil.ly/vzcPD)).

Пример 12.4 ❖ Ресурс Istio DestinationRule для сервиса httpbin, объявляющий два подмножества

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: httpbin
spec:
  host: httpbin
  subsets:
 - name: v1
    labels:
      version: v1
 - name: v2
    labels:
      version: v2
```

Объявлены два подмножества: v1 – для уже установленных рабочих нагрузок и v2 – для рабочих нагрузок, которые планируется развернуть.

Обратите внимание, что метка `version: v1` обозначает первоначально развернутое приложение. Также отметьте, что метка `v2` соответствует подмножеству еще *не* развернутых сервисов. Это нормально. Когда рабочие нагрузки с меткой `version: v2` вступят в строй, они будут подхвачены правилом `DestinationRule`.

Но до этого момента трафик, направленный в подмножество `httpbin v2`, будет вызывать ошибку 500 (потому что в подгруппе v2 нет ни одного действующего сервера).

Теперь обновим `VirtualService` и задействуем подмножество, как показано в примере 12.5 (см. httpbin-vs-v1.yaml в репозитории GitHub книги (https://oreil.ly/a80aT)).

Пример 12.5 ❖ VirtualService из примера 12.2 с добавленным разделом destination, описывающим подмножество

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
  - "*"
  gateways:
  - httpbin-gateway
  - mesh # Также прямой трафик в сетке с тем же VirtualService
http:
- route:
  - destination:
      host: httpbin
      subset: v1
      port:
        number: 8000
```

Это определение гарантирует, что весь трафик, как внутри сетки (благодаря `gateways: mesh`), так и на входе (`gateways: httpbin-gateway`), будет привязан к подмножеству v1 сервисов `httpbin`. Теперь можно без опаски развернуть новую версию `httpbin`, которая точно не будет получать пользовательский трафик, как показано в примере 12.6 (см. httpbin-depl-v2.yaml в репозитории GitHub книги (https://oreil.ly/uXK71)).

Пример 12.6 ❖ Второе развертывание httpbin с меткой version: v2

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: httpbin-v2
spec:
  replicas: 1
template:
  metadata:
    labels:
      app: httpbin
      version: v2
  spec:
    containers:
    - image: docker.io/kennethreitz/httpbin
      imagePullPolicy: IfNotPresent
      name: httpbin
```

```
    ports:
    - containerPort: 80
```

Можно продолжить отправку трафика сервису `httpbin` извне или изнутри кластера, и часть его должна поступить в новую версию. Чтобы это проверить, используйте метрики Istio.

Проведем канареечное испытание новой версии. Как показано в примере 12.7, направим 5 % трафика в новую версию и понаблюдаем за кодами ответов сервиса и задержкой через метрики Istio, чтобы убедиться в его работоспособности, а затем постепенно увеличим процент (см. httpbin-vs-v2-5.yaml в репозитории GitHub книги (https://oreil.ly/rB9cH)).

Пример 12.7 ❖ VirtualService из примера 12.6 с дополнительными настройками для отправки 5 % трафика в подмножество v2 httpbin

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
  - "*"
  gateways:
  - httpbin-gateway
  - mesh # Также прямой трафик в сетке с тем же VirtualService
  http:
  - route:
    - destination:
        host: httpbin
        subset: v1
        port:
          number: 8000
      weight: 95
    - destination:
        host: httpbin
        subset: v2
        port:
          number: 8000
      weight: 5
```

Продолжим процесс, постепенно увеличивая вес для `subset: v2` и уменьшая для `subset: v1`. Имейте в виду, что сумма всех весов должна составлять 100 (%), при этом число подмножеств, одновременно получающих трафик, не ограничивается (в этом примере использованы лишь два таких подмножества).

После развертывания и опробования новой версии нужно сделать выбор: либо убрать из `DestinationRule` не используемые сейчас подмножества, либо оставить старые на месте. Рекомендуется придерживаться промежуточной позиции: держать `DestinationRule` и `VirtualService` фиксированными для работы с двумя подмножествами, текущими и следующими. В данном примере можно оставить подгруппы v1 и v2 без изменений до тех пор, пока не придет время развернуть v3. Затем можно заменить v1 на v3 и повторить всю процедуру развертывания, постепенно перейдя с v2 на v3. Имея на руках конфигурации

(DestinationRule и VirtualService) для подгрупп v2 и v3, когда настанет время v4, останется только заменить v2 на v4 и выполнить такой же канареечный переход с v3 на v4, и т. д. Кроме того, при таком подходе значительно уменьшается количество конфигураций, подлежащих изменению в чрезвычайной ситуации, когда придется вернуться к предыдущему работоспособному варианту развертывания: необходимая для этого конфигурация уже есть, останется только заново развернуть двоичный файл и изменить веса в VirtualService.

Кросс-кластерное развертывание

Как обсуждалось в главе 8, возможности маршрутизации трафика не ограничены использованием только сервисов в пределах одного кластера. Каждая компания, осуществляющая серьезные инвестиции в Kubernetes, рано или поздно столкнется с необходимостью развертывания и управления сервисами в нескольких кластерах. Множество кластеров обычно используют для создания изолированных отказоустойчивых доменов. Если для создания отказоустойчивых доменов используются кластеры, для обхода отказов во время работы очень важно уметь перемещать трафик между кластерами. Используя те же техники разделения трафика, применяемые для канареечного развертывания, можно также постепенно (или весь сразу) перевести трафик с одного кластера на другой. Istio поддерживает такую возможность (подробнее см. главу 13). Покажем, как это работает. Представим себе удаленный кластер с IP-адресом входного шлюза 1.2.3.4, на котором работает сервис httpbin. В первом кластере с сервисом httpbin, показанном в примере 12.8, можно создать новую запись ServiceEntry, ссылающуюся на вход нового кластера (см. httpbin-cross-cluster-svcentry.yaml в репозитории GitHub книги (https://oreil.ly/PGfU5)).

Пример 12.8 ❖ ServiceEntry для httpbin.remote.global

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: httpbin-remote
spec:
  hosts:
 - httpbin.remote.global # удаленный постфикс используется для DNS-плагина Istio
  location: MESH_INTERNAL # требуем использовать mTLS
  ports:
 - name: http
    number: 8000
    protocol: http
resolution: DNS
addresses:
# Не обязательно маршрутизируемый, но должен быть уникальным
# для каждого сервиса, маршрутизируемого между кластерами;
# используется DNS-модулем Istio
- 127.255.0.2
endpoints:
- address: 1.2.3.4 # адрес входного шлюза удаленного кластера
  ports:
```

```
# Не изменяйте это значение порта, если используется мультикластерная установка Istio
    http: 15443
```

В локальном кластере можно обновить `VirtualService` для принудительной отправки трафика сервису `httpbin` в удаленном кластере, как показано в примере 12.9 (см. httpbin-cross-cluster-vs.yaml в репозитории GitHub книги (https://oreil.ly/j7X6l)).

Пример 12.9 ❖ Обновленная версия примера 12.5

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
 - "*"
  gateways:
 - httpbin-gateway
 - mesh # Также прямой трафик в сетке с тем же VirtualService
  http:
 - route:
   - destination:
       host: httpbin.remote.global
       port:
         number: 8000
```

Это определение обеспечит маршрутизацию внешнего трафика, направленного внутрь локального кластера, а также внутреннего трафика, адресованного сервису `httpbin` в удаленном кластере. Поскольку в `ServiceEntry` задекларировано, что конечная точка имеет тип `MESH_INTERNAL`, гарантируется, что mTLS будет использоваться для обмена данными между кластерами, поэтому нет необходимости настраивать VPN-соединение между кластерами. Если понадобится, маршрут можно проложить даже через интернет.

В этой главе мы кратко рассмотрели возможные проблемы компонентов уровня управления и их влияние на сервисную сетку, начиная от соседних компонентов уровня управления и заканчивая прокси уровня данных с рабочими нагрузками, к которым они подключены. Приведен пример безопасного развертывания новой версии существующего сервиса, где показан прием переключения пользователей на новую версию с сохранением возможности отката пользовательского трафика на старую версию. Наконец, приведен очень краткий пример использования примитивов маршрутизации трафика, использованных при работе с канареечными развертываниями, для поддержки конфигураций с несколькими кластерами. Istio блистательно показала себя в этой области, не требуя больших усилий для организации развертывания в вариантах с высокой отказоустойчивостью (active/active) и высокой доступностью (active/passive).

Глава 13

Продвинутые сценарии

В некоторых средах бывает достаточно развертывания однокластерной сервисной сетки. Но в других может понадобиться создать несколько кластеров и развернуть единую глобальную сервисную сетку или объединить несколько независимых сеток. Такие ситуации предполагают участие существующих монолитов или других внешних сервисов. И хотя мы все будем рады тому моменту, когда микросервисы начнут править миром и монолитные приложения уйдут на пыльные страницы истории, он еще не наступил. Проект Istio это учитывает и поддерживает различные модели развертывания и настройки.

В данной главе будет рассмотрено несколько наиболее распространенных продвинутых топологий. Передовые топологии полезны в средах, где внимание уделяется географически близким микросервисам или средам, где сервисные сетки распределены по регионам или поставщикам.

Типы продвинутых топологий

Существует множество топологических конфигураций, но здесь мы рассмотрим только самые основные, на базе которых строятся другие конфигурации. Давайте разобьем эти немногие топологии на две основные категории, развернутые в рамках одного или нескольких кластеров.

Однокластерные сетки

Продвинутая однокластерная топология подразумевает *расширение сетки*. Расширение сетки – это топология, включающая традиционные, немикросервисные рабочие нагрузки (монолитные приложения) на физических или виртуальных машинах (или на тех и других) в сервисную сетку Istio. Хотя такие приложения не получают всех преимуществ сервисных сеток, их включение в нее дает возможность получить представление о взаимодействии сервисов друг с другом и управлении их взаимодействием. Это закладывает основу для миграции в облачную архитектуру или для распределения рабочей нагрузки между Kubernetes- и не-Kubernetes-узлами.

Об этом мы чуть подробнее поговорим в данной главе немного позже. Речь идет о включении существующих приложений в сетку, где можно наблюдать за трафиком, начать разрушение монолита, отсекая трафик с использованием правил маршрутизации, или тестировать сервисы, когда они больше *не* работают на физической или виртуальной машине.

Мультикластерные сетки

К двум другим типам продвинутых топологий относятся *мультикластерные* и *кросс-кластерные* модели Istio. Мы относим эти две топологии в одну и ту же мультикластерную (федеративную) категорию сеток, поскольку, по сути, они направлены на решение одной и той же проблемы межкластерной коммуникации. Они обеспечивают связь между разрозненными кластерами Kubernetes, на которых работают отдельные сервисные сетки, объединяя их либо на одном уровне управления, либо на двух. Но именно здесь терминология становится слишком каверзной, и может начаться неразбериха.

Для простоты сейчас мы кратко изложим эти два подхода, а более подробно рассмотрим их далее.

Мультикластер Istio (единая сетка)

Односеточный *мультикластер Istio* представляет собой централизованный подход к объединению нескольких сервисных сеток в одну. Выбирается кластер, выступающий в роли мастера, остальные же становятся удаленными кластерами. Термины *локальные* и *удаленные* обозначают локальность или удаленность уровня данных того или иного кластера по отношению к централизованному уровню управления.

Развертывание Istio с одним уровнем управления может охватывать несколько кластеров, при условии что между ними есть сетевое соединение и нет перекрытия диапазона IP-адресов. Рисунок 13.1 иллюстрирует, как Istio v1.0 поддерживает плоскую сетевую модель с несколькими кластерами. Istio v1.0 можно расширить до неплоских сетей, в которых существует конфликт IP-адресов между кластерами.

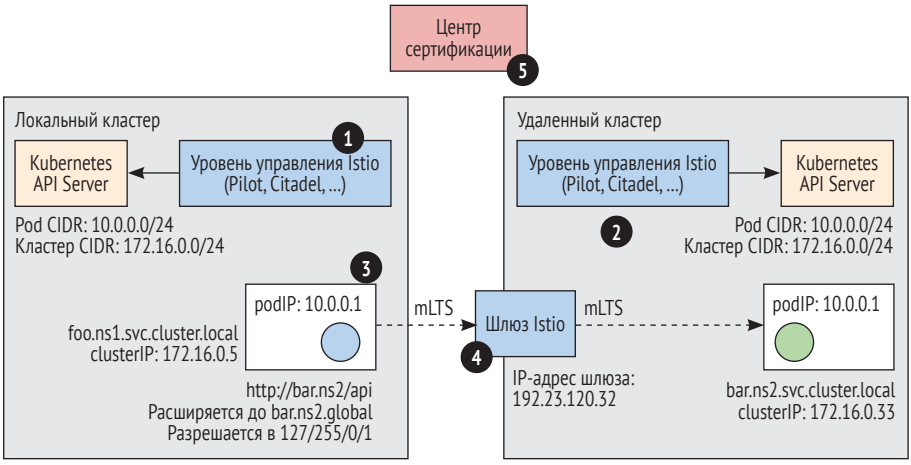

Рис. 13.1 ❖ Мультикластер Istio: развертывание Istio с одним уровнем управления при прямом соединении между кластерами (плоская сеть)

Используя преобразование сетевых адресов (NAT) и комбинацию VPN, Istio `Gateway` или другие сетевые сервисы, можно объединять кластеры в один адми-

нистративный домен (с общим уровнем управления). В версии 1.0 можно объединить несколько уровней управления, но в Istio это потребует длительной ручной настройки и конфигурирования. Независимо от подхода к объединению кластеров, для разрешения имен сервисов и проверки сертификатов все пространства имен, сервисы и учетные записи сервисов должны быть одинаково определены в каждом кластере.

В Istio v1.1 плоское сетевое окружение больше не является необходимостью. Добавлены две дополнительные возможности, позволяющие реализовать различные мультикластерные сценарии:

Расщепление горизонта (*EDS*)
: Pilot реализует поддержку Envoy EDS API и использует его для настройки прокси в уровнях данных с информацией о сервисах и конечных точках, локальных только для данного кластера, где находится прокси. С функцией расщепления горизонта (EDS) Pilot представляет конечные точки кластера, где расположен прокси, подключенный к сервису, что позволяет Istio направлять запросы к разным конечным точкам, в зависимости от расположения запрашивающего источника. Istio `Gateway` перехватывают и разбирают TLS-обмен и используют SNI-данные для определения конечных точек целевого сервиса.

Маршрутизация на основе SNI
: SNI-расширение TLS используется для принятия решений о маршрутизации между кластерами.

Как отмечалось в главе 5, EDS является частью API Envoy. Расщепление горизонта – это сетевая концепция, в которой зацикливание маршрутов исключается запретом маршрутизатору анонсировать маршрут обратно на интерфейс, с которого он был получен. В случае с Istio компонент Pilot настраивает прокси уровня данных, используя информацию о сервисах и конечных точках, с учетом сведений о конечных точках, находящихся в кластере, где работают прокси.

Имена для сервисов, учетных записей сервисов и т. д. должны выбираться единообразно для всех кластеров, но в версии 1.1 улучшена осведомленность Istio о кластерах и расположении. Метки и аннотации Kubernetes используются для получения дополнительной информации о кластерах как на уровне отдельных сетей (ассоциирование каждой сети с определенным кластером), так и на уровне географии для более разумной балансировки нагрузки на основе местоположения.

После инициализации прокси присваивает каждой сети метку кластера, которая связывает данный экземпляр сервиса с кластером. Обычно каждому кластеру даются свои значения меток, однако это настраиваемо до степени принадлежности нескольких кластеров к одной логической сети, и они должны маршрутизироваться напрямую и иметь малую задержку (в идеале). У каждого кластера также будет свой входной шлюз, имеющий то же значение метки сети, что и другие рабочие нагрузки кластера. Совпадающие значения меток связывают конечные точки сервисов кластера с входным шлюзом кластера. Поскольку он используется только для межкластерной связи, входной шлюз идеально отделен от входящего потока в кластер и недоступен конечным пользователям.

Мультикластерное развертывание объединяет несколько кластеров в одну логическую единицу с общим уровнем управления Istio, имеющую единое

логическое представление всех сервисов по кластерам. Это может быть набор уровней управления, синхронизированных с общей конфигурацией (обычно с использованием дополнительных инструментов, управляемых общими CI/CD-конвейерами и/или методами GitOps) или одним «физическим» уровнем управления, работающим с несколькими уровнями данных как с одной сервисной сеткой. В любом случае имеющийся набор кластеров является частью одной и той же сетки.

Istio должна знать, к какому кластеру относятся рабочие нагрузки. Подобно использованию меток Kubernetes для локальной балансировки нагрузки, Istio присваивает каждому кластеру «сетевую» метку. Они также используются для оповещения кластера Istio о том, с каким из них связана данная сеть. Обычно используются разные значения меток для каждого кластера, но это можно изменить, если известно, что несколько кластеров являются частью одной логической сети (например, маршрутизируемые напрямую, с низкой задержкой).

Кросс-кластер Istio (объединение сеток)

Кросс-кластер Istio (объединение сеток) – это децентрализованный подход к объединению сервисных сеток. Каждый кластер использует свой собственный уровень управления и уровень данных. Два или более кластеров могут участвовать в сервисной сетке независимо от региона или провайдера облачных вычислений. Кросс-кластерное развертывание облегчает работу сервисных сеток с относительно разными конфигурациями в различных административных доменах, работающих в разных регионах. Имея в виду отдельные административные домены, преимуществом модели объединения сеток является возможность выборочно устанавливать связи между кластерами и, в свою очередь, предоставлять сервисы одного кластера другим.

Улучшение межкластерной балансировки нагрузки

При эксплуатации сервиса в одном кластере обычно работает несколько экземпляров его подов, чтобы сбалансировать входящие запросы. В первую очередь это помогает масштабируемости (и доступности в контексте масштабирования, но не в контексте проблемных доменов). При эксплуатации сервиса на нескольких кластерах можно балансировать нагрузку входящих запросов между подами в обоих кластерах. Это помогает не только с масштабируемостью, но и с доступностью (отказоустойчивостью). Однако во время балансировки запросов между кластерами обычно требуется балансировка нагрузки с учетом местной специфики, т. е. балансировка запросов к определенному сервису с переадресацией в удаленный кластер нежелательна, *если* запросы можно обработать локально (в подах локального кластера).

В общем случае желательно сохранить локальность трафика. Балансировка нагрузки на основе местоположения поддерживает такое поведение, что при исправности конечных точек информация о местоположении используется для удержания сбалансированного трафика в непосредственной близости. С выходом Istio версии v1.1 появилась локальная балансировка нагрузки (согласно практике обкатки новых функций, она экспериментально поддерживается в 1.1 и выключена по умолчанию). Istio определяет локальность как географическое местоположение, используя комбинацию концепций региона, зоны и подзоны для приоритезации пулов распределения нагрузки для контроля географического местоположения, куда направляются запросы.

Используя свойства конкретной среды развертывания, Istio получает информацию о местоположении из нижележащей платформы. Рассматривая Kubernetes как пример платформы, обратим внимание, что Kubernetes имеет набор зарезервированных меток и аннотаций в пространстве имен kubernetes.io. При развертывании в общедоступном облаке сервисы kubelet на каждом узле кластера будут заполнять зарезервированные метки, такие как `failure-domain.beta.kubernetes.io/region`, из которых Istio может получить информацию о местоположении региона, зоны и подзоны.

Эта новая функция позволяет использовать два режима балансировки нагрузки: *distribute* (распределение) и *failover* (обход отказа). Режим распределения облегчает балансировку нагрузки в зависимости от местоположения. При сбое конечных точек в том или ином пункте прокси Envoy скорректирует вес пункта назначения для отражения ситуации, используя взвешенный круговой график (*round-robin schedule*). Режим обхода отказа гарантирует, что когда отказ происходит в конечных точках в локальном регионе, трафик перенаправляется в удаленный регион (регионы). Следует настроить привязку обходных регионов, чтобы при сбое перенаправлять на них трафик других регионов.

Даже с улучшением поддержки мультикластеров в версии 1.1 в Istio появились не все возможные алгоритмы балансировки нагрузки. В настоящее время поддерживаются такие режимы балансировки нагрузки, как круговой, случайный, взвешенный и «меньше запросов» (*least-requests*). Однокластерная балансировка нагрузки все еще совершенствуется, и Envoy поддерживает несколько других сложных алгоритмов.

Рабочие примеры

Как видно из расширенных конфигураций, описанных выше, существует множество рабочих примеров. Помня мантру о необходимости обеспечить безопасность, наблюдение, контроль и подключение сервисов независимо от того, где или на чем они работают (публичное, частное или гибридное облако), давайте рассмотрим, *как использовать* эти продвинутые модели развертывания.

HA (*межрегиональная*)
: Как с помощью мультикластера, так и кросс-кластера можно решить межрегиональную задачу. То есть можно разместить кластер Kubernetes в двух отдельных регионах с безопасной маршрутизацией трафика между ними. Также можно обходить отказы в этих регионах с помощью кросс-кластерных конфигураций, чтобы при отказе одного региона не обязательно происходил отказ всего приложения.

Кросс-провайдерная
: Продолжая межрегиональную, обе топологии поддерживают многооблачные конфигурации, однако требования для них существенно различаются. Подробнее об этом мы поговорим в следующем разделе. Говоря кратко, Istio позволяет развернуть сервисную сетку, охватывающую несколько облаков.

Стратегии развертывания
: Мультикластерная установка позволяет размещать канареечные версии на недорогих экземплярах у дешевого провайдера. Представьте себе запуск канареечной версии в DigitalOcean, на которую можно передавать 1 % трафика из рабочей среды, функционирующей в IBM Cloud. Это возможно при использовании обеих топологий. Подобная логика применима для таких стратегий, как A/B-тестирование и *сине-зеленое развертывание*.

Распределенная трассировка монолита

На базе расширения сервисной сетки монолитное приложение становится более прозрачным. После включения в сетку традиционных приложений, работающих на виртуальных или физических машинах, можно собирать данные трассировки и многое другое.

Миграция

Кросс-кластер позволяет перемещать сервисы между регионами или провайдерами, используя Istio для управления маршрутизацией трафика. Одним из наиболее интересных сценариев миграции является возможность перемещения старых приложений в Kubernetes по частям. Это поможет придать старому приложению немного облачного шарма. Теперь «старичок» сможет, если необходимо, взаимодействовать с размещаемыми в кластере новыми сервисами.

Перечисленные сценарии начнут приобретать смысл по мере углубления в каждый из них далее в этой главе. После упражнений должно появиться базовое понимание того, как настроить каждое из них и как они работают. Для иллюстрации различий используем пример приложения Bookinfo.

Выбор топологии

Дизайн каждой топологии развертывания сопровождается компромиссами реализации. Весьма вероятно, что выбор подхода будет и, вероятно, должен диктоваться тем, где находятся данные (или выполняются вычисления). Если используются только кластеры в общедоступном облаке, более целесообразным может оказаться межкластерный подход. Если используются несколько собственных кластеров с одним или двумя кластерами в общедоступных облаках, может иметь смысл мультикластер. Это не означает, что нельзя использовать кросс-кластер между локальными и общедоступными провайдерами облачных вычислений, тем более что локальные провайдеры начинают брать пример с того, как предоставляются облачные сервисы, например NetApp HCI, Azure Stack, GKE On-Prem и др.

Кросс-кластер или мультикластер?

Погрузимся в глубокую часть бассейна. Как говорилось выше, рассуждения о мультикластере Istio в сравнении с кросс-кластером лучше всего основывать на сравнении *централизованных* и *децентрализованных* уровней управления соответственно. С течением времени они стали двумя доминирующими подходами к соединению нескольких кластеров Kubernetes, совместно использующих сервисную сетку. У обоих есть свои преимущества и недостатки. Давайте рассмотрим эти плюсы и минусы, начав с мультикластера.

Каждый уровень данных, локальный или удаленный от центрального уровня управления, должен иметь подключение к компонентам управления, таким как Pilot и Mixer. Локальные и удаленные уровни данных также должны иметь возможность передавать телеметрию в центральный уровень управления. Все кластеры, участвующие в сервисной сетке, должны иметь уникальные сетевые

диапазоны и маршрутизироваться между собой. Общим подходом к упрощению связи между провайдерами или между регионами является использование приватных туннелей между кластерами. В зависимости от конкретной среды это может быть VPN между локальным кластером (кластерами) и провайдером (провайдерами) или между регионами провайдера. Rancher Submariner (https://submariner.io/) или Amazon VPC – лишь два примера используемых технологий. Все больше и больше людей используют присущие самой Istio возможности, пользуясь преимуществами безопасной связи шлюз–шлюз.

Мультикластерная среда Istio v1.1 требует, чтобы основные компоненты уровня управления находились только в одном кластере и все удаленные кластеры взаимодействовали с ними. Удаленные кластеры должны настраиваться на автоматическое внедрение прокси и компонента Citadel (все они должны быть связаны с общим корневым центром сертификации). Сеть можно расширить для поддержки неплоских сетей с помощью NAT или VPN.

Один из примеров использования такого типа развертывания – соединение локальной сервисной сетки Istio с общедоступным облаком через VPN. Это позволяет разработчикам проверять с помощью канареечного развертывания, или A/B-тестирования, или чего-то индивидуального свои сервисы у облачного провайдера, снижая потребность попадания трафика в локальную рабочую среду. Такой подход упрощает постепенную миграцию в общедоступное облако или переход к гибридному решению, если нормативы требуют привязки к конкретному месту. Это один из многих вариантов, возникающих при наличии гибридного подключения к общедоступному облаку из частной среды.

Также необязательно, чтобы приложения работали в одном кластере с уровнем управления. В некоторых случаях имеет смысл использовать централизованный уровень управления, предназначенный для тех компонентов и удаленных уровней управления, где выполняется рабочая нагрузка (хотя топология такого типа и повышает риск потери связи между уровнями управления и данных из-за неустойчивой работы сети).

Второй вариант развертывания – кросс-кластер Istio, как показано на рис. 13.2.

Кросс-кластерные модели развертывания позволяют объединять децентрализованные группы сервисных сеток Istio с помощью правил маршрутизации, установленных в каждой сервисной сетке. В этом сценарии каждый кластер Kubernetes запускает свой собственный экземпляр уровня управления. Оба используются для выполнения рабочих нагрузок.

Давайте посмотрим, как выполняются кросс-кластерные вызовы. Понимание того, какие системы задействованы и какие шаги поддерживают этот процесс, позволяет понять кросс-кластерное поведение:

- клиентская рабочая нагрузка разрешает имя внешнего сервиса в конечную точку сети (например, с помощью Kubernetes DNS или другого реестра сервисов, такого как Consul). По сути, это означает, что для того чтобы клиентская рабочая нагрузка успешно разрешила имя удаленного сервиса в конечную точку, удаленный сервис должен быть зарегистрирован в локальном сервере имен (DNS) или реестре сервисов;

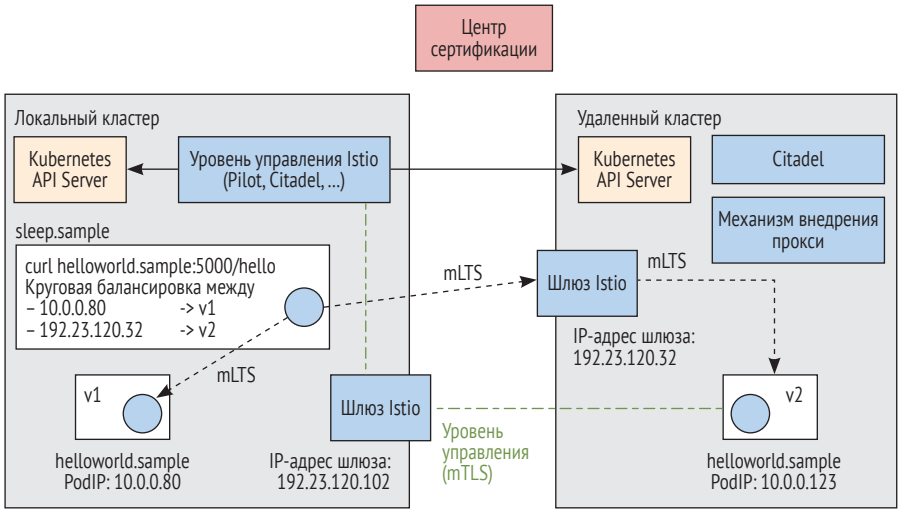

Рис. 13.2 ❖ Кросс-кластерная топология Istio v1.1 с маршрутизацией сервисов с учетом кластеров: один уровень управления охватывает несколько кластеров Kubernetes

- определив адрес конечной точки, клиент вызывает удаленный сервис. Эти запросы перехватываются локальным прокси. Затем запрос отображается в определенную конечную точку и передается ей для обработки. В зависимости от топологии и конфигурации безопасности прокси клиентского сервиса может подключаться непосредственно к удаленной конечной точке. Или же он может подключаться через входной и/или выходной шлюз;
- удаленный прокси принимает соединение и проверяет идентичность с помощью mTLS (это неявно требует, чтобы все сервисы во всех кластерах использовали общий корень доверия посредством одного и того же или разных компонентов Citadel);
- в зависимости от политики авторизации может потребоваться обратиться к API check компонента Mixer. В этом случае идентификаторы клиента и внешнего сервиса (из разных кластеров) передаются компоненту Mixer для оценки.

С точки зрения администратора требования для кросс-кластера проще, чем для мультикластера, – здесь не нужно настраивать прямое соединение, VPN, VPC или что-то подобное. Но необходимо определить некоторую входную конечную точку для связи с кластером и открыть порт для взаимодействия через этот туннель.

Каждый кластер должен быть в состоянии взаимодействовать с каждым целевым кластером через выбранный порт, например 80 или 443. В общедоступном облаке это требование предъявляется к обеим сторонам. Например, нужно убедиться, что исходный кластер может взаимодействовать с целевым кластером посредством гибкой балансировки нагрузки, и наоборот. Входная конечная точка определяется как `ServiceEntry` внутри Istio.

Давайте сделаем паузу и вспомним определение ServiceEntry из главы 8. ServiceEntry содержит множество свойств – хост, адрес, порт, протокол и конечную точку. ServiceEntry используется для информирования Istio о еще не обнаруженном сервисе. ServiceEntry может идентифицировать сервисы за пределами сетки или расположенные внутри нее.

Кросс-кластерная топология может принести пользу, устраняя необходимость использования VPN для соединения кластеров, как это необходимо сделать с мультикластером (если только вы не используете маршрутизацию с поддержкой кластера в версии 1.1 с помощью Istio Gateway). Она также защищает от появления единой точки отказа. Недостатком является необходимость определения политики, уникальной для каждой среды. Если нужно применить одну и ту же политику глобально, придется в каждом кластере сделать это индивидуально. Надеемся, что со временем в таких решениях, как Galley, появятся средства управления конфигурацией Istio.

Кроме того, поскольку это разрозненные кластеры Kubernetes, все еще нужно искать решение для репликации объектов, которые должны быть в обеих средах. В Kubernetes это можно решить через механизм Kubernetes Federation, который позволяет гарантировать создание большинства объектов в каждом кластере, участвующем в объединении.

Учитывая незрелость Kubernetes Federation v2, было бы разумно использовать альтернативные подходы, основанные на GitOps, не дожидаясь, пока v2 достигнет требуемого уровня надежности.

Еще один момент, заслуживающий внимания, – балансировка нагрузки в облаке, если вы решите пойти по пути запуска сервисов в нескольких разных облачных окружениях, чтобы получить подлинную избыточность.

Настройка кросс-кластера

Давайте рассмотрим пример развертывания двух или более кластеров с использованием кросс-кластерной топологии. В этом упражнении нам не нужно обеспечивать существование уникальных сетей во всех кластерах, участвующих в каждой сетке. Их шлюзы должны иметь возможность беспрепятственно подключаться друг к другу. С кросс-кластерной топологией прокси будут взаимодействовать со своим локальным уровнем управления для администрирования, авторизации, телеметрии и т. д. Далее мы будем предполагать, что вы уже знакомы с установкой и настройкой кластера Kubernetes. Если нет желания проходить все эти сложности вручную, можно разместить два или более кластеров в управляемом кластере Kubernetes для ускорения прохождения этого упражнения (не обязательно для управления Kubernetes в целом).

Для этого упражнения необходимо выполнить несколько предварительных условий:

- к каждому кластеру Kubernetes должен иметься доступ с привилегиями ClusterAdmin, и нужно проверить, что утилита kubectl способна подключаться к обоим кластерам. Доступ к командной оболочке не понадобится;
- шлюзы в каждом кластере обеспечивают связь между кластерами через TLS, необходимую для взаимодействий сервисов. IP-адрес сервиса istio-

`ingressgateway` в каждом кластере должен быть доступен из всех остальных кластеров;

- вышеупомянутая межкластерная коммуникация требует использования mTLS между сервисами, который сам по себе требует общего корневого центра сертификации (*shared root CA*). Чтобы удовлетворить это требование, компонент Citadel в каждом кластере должен быть настроен с промежуточными учетными данными CA, сгенерированными общим корневым CA;
- все кластеры Kubernetes, действующие под управлением Istio, должны иметь одну и ту же версию (1.12 или выше) с Istio версии 1.0 или выше. Можно использовать управляемый сервис Kubernetes, такой как NetApp Kubernetes Services (NKS), Google Kubernetes Engine (GKE) или Azure Kubernetes Service (AKS), – это может сократить время выполнения упражнения. Конечно, необязательно использовать именно эти сервисы, так как любого CNCF-совместимого дистрибутива Kubernetes будет вполне достаточно.

Удобно, что есть Helm-схемы, автоматизирующие бóльшую часть этой установки. Давайте пройдемся по каждому шагу, чтобы увидеть «происходящее под капотом».

1. Разверните два или более кластеров Kubernetes. Достаточно следовать процессу установки, описанному в главе 4.
2. Как показано в документации Istio (https://istio.io/docs/), сгенерируйте мультикластерный конфигурационный файл Istio-шлюзов с помощью Helm, используя его команду `template`. Убедитесь, что `helm` установлен локально и из каталога пакетов Istio, выполнив следующую команду:

   ```
   $ helm template install/kubernetes/helm/istio --name Istio
   --namespace istio-system -f \ install/kubernetes/helm/Istio
   /example-values/values-istio-multicluster-gateways.yaml >
   $HOME/istio.yaml
   ```

   В этой установке подразумевается необходимость использования общего корня доверия для получения сертификатов, даже если они подписаны разными экземплярами Citadel. Для корректной работы mTLS в кластерах нужно использовать общий корневой CA. Пока в каждом кластере используются одни и те же сертификаты и Citadel может выпускать и передавать их прокси-серверам, межкластерное взаимодействие будет защищено с помощью mTLS. Citadel мы подробно описали в главе 6.
3. Установите ресурсы CRD Istio в каждый кластер:

   ```
   $ kubectl apply -f install/kubernetes/helm/istio-init/files/crd/
   ```

4. В каждом кластере необходимо создать уровень управления. Все они должны быть настроены одинаково. Начните с создания пространства имен `istio-system` в кластере Kubernetes:

   ```
   $ kubectl create ns istio-system
   ```

5. В каждом кластере выполните следующие действия:

   ```
   $ kubectl create namespace istio-system
   $ kubectl create secret generic cacerts -n istio-system \
   ```

```
    --from-file=samples/certs/ca-cert.pem \
    --from-file=samples/certs/ca-key.pem \
    --from-file=samples/certs/root-cert.pem \
    --from-file=samples/certs/cert-chain.pem
```

6. Затем примените шаблон Helm, сгенерированный ранее:

```
$ kubectl apply -f $HOME/istio.yaml
```

7. Убедитесь, что во всех кластерах включено автоматическое внедрение прокси (что уже должно быть сделано):

```
$ kubectl label namespace default istio-injection=enabled
```

Настройка DNS и развертывание Bookinfo

В этом примере в обоих кластерах мы развернем Bookinfo, стандартное демонстрационное приложение Istio. Убедимся, что CoreDNS настроен на кросс-кластерное разрешение имен, и в каждом кластере применим обновление `ConfigMap`, показанное в примере 13.1 (см. istiocoredns.yaml в репозитории GitHub книги (https://oreil.ly/Dmy1v)).

Пример 13.1 ❖ Кросс-кластерный ConfigMap для CoreDNS

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
   .:53 {
        errors
        health
        kubernetes cluster.local in-addr.arpa ip6.arpa {
            pods insecure
            upstream
            fallthrough in-addr.arpa ip6.arpa
        }
        prometheus :9153
        proxy. /etc/resolv.conf
        cache 30
        loadbalance
        loop
        reload

    }
    global:53 {
        errors
        cache 30
        proxy. $(kubectl -n istio-system get svc istiocoredns
             -o jsonpath={.spec.clusterIP})
    }
```

Настроим и сконфигурируем правила Istio для `Gateway`, `ServiceEntries` и `VirtualService`. Обратитесь к главе 8 за подробным объяснением основных сетевых конструкций Istio. Сначала настроим входные шлюзы для обоих кластеров, A и B, как показано в примере 13.2. Используйте конфигурацию шлюза для обоих кластеров (смотрите ingress-gw.yaml в репозитории на GitHub книги (https://oreil.ly/MCnpt)).

Пример 13.2 ❖ Создание шлюза для каждого кластера

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  generation: 1
  name: ingress-gateway
  namespace: "default"
  resourceVersion: ""
  selfLink: /apis/networking.istio.io/v1alpha3/namespaces/default/gateways
            /ingress-gateway
  uid: ""
spec:
  selector:
    istio: ingressgateway
  servers:
 - hosts:
   - '*'
    port:
      name: http
      number: 80
      protocol: HTTP
 - hosts:
   - '*'
  port:
    name: https
    number: 443
    protocol: HTTPS
  tls:
    caCertificates: /etc/istio/ingressgateway-ca-certs/ca-chain.cert.pem
    mode: MUTUAL
    privateKey: /etc/istio/ingressgateway-certs/tls.key
    serverCertificate: /etc/istio/ingressgateway-certs/tls.crt
```

Использование публичного облачного провайдера в качестве инфраструктуры для данного примера может (или не может) способствовать быстрому открытию публичного входного шлюза для обоих кластеров, по сравнению с локальным развертыванием. Кросс-кластерная топология требует определить внешний IP-адрес, чтобы разрешить транзит сервисного трафика через публичный интернет. Это также иллюстрирует, как Istio поможет элегантно решить проблему многооблачной кластерной коммуникации.

Имея под рукой внешний IP-адрес, подключитесь к кластеру *A* и примените следующие настройки с помощью `kubectl`. Укажите адрес входного шлюза кластера B в разделе `endpoints` и имя удаленного сервиса в разделе `hosts`, как

показано в примере 13.3 (см. egress-serviceentry-a.yaml в репозитории GitHub книги (https://oreil.ly/ongFN)).

Пример 13.3 ❖ Определение ServiceEntry для доступа к кластеру B

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  generation: 1
  name: egress-service-entry
  namespace: "default"
  resourceVersion: ""
  selfLink: /apis/networking.istio.io/v1alpha3/namespaces/default
            /serviceentries/egress-service-entry
  uid: ""
spec:
  endpoints:
 - address: # <укажите внешний IP-адрес>
  hosts:
 - svc.cluster-b.remote
  location: MESH_EXTERNAL
  ports:
 - name: https
    number: 443
    protocol: HTTPS
  resolution: DNS
```

Теперь подключитесь к кластеру *B* и создайте `ServiceEntry` из примера 13.4. Укажите в разделах `endpoints` и `hosts` данные, соответствующие кластеру *A*. Помните, что нужно иметь возможность добраться до конечных точек изнутри кластера (см. egress-serviceentry-b.yaml в репозитории GitHub книги (https://oreil.ly/5Rq1o)).

Пример 13.4 ❖ Определение ServiceEntry для доступа к кластеру A

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  generation: 1
  name: egress-service-entry
  namespace: "default"
  resourceVersion: ""
  selfLink: /apis/networking.istio.io/v1alpha3/namespaces/default/serviceentries
            /egress-service-entry
  uid: ""
spec:
  endpoints:
 - address: # < укажите внешний IP-адрес >
  hosts:
 - svc.cluster-a.remote
  location: MESH_EXTERNAL
  ports:
 - name: https
    number: 443
```

```
    protocol: HTTPS
  resolution: DNS
```

На данный момент все готово к распределению трафика между кластерами. В данном примере мы распределим трафик между кластерами *A* и *B*, определив правило `DestinationRule`. В качестве примера используем *Reviews*, сервис Bookinfo. В этой схеме два экземпляра сервиса работают в обоих кластерах (но в этом нет необходимости, учитывая, что конфигурация CoreDNS была обновлена ранее).

Правило `DestinationRule` сообщает Istio, куда направлять трафик. Такие правила могут определять самые разные конфигурации. В следующем примере определяется правило `DestinationRule`, описывающее поддержку mTLS для исходящего трафика, посылаемого в порт 443.

Подключитесь к кластеру *A* и примените правило, показанное в примере 13.5. Атрибут `host` определяет удаленный кластер *B*. Наш трафик будет направлен в порт 443 внешнего кластера (см. review-destinationrule.yaml в репозитории GitHub книги (https://oreil.ly/MQYBN)).

Пример 13.5 ❖ DestinationRule направляет трафик для Reviews в кластер B

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  generation: 1
  name: reviews-tls-origination
  namespace: "default"
  resourceVersion: ""
  selfLink: /apis/networking.istio.io/v1alpha3/namespaces/default/destinationrules
            /reviews-tls-origination
  uid: ""
spec:
  host: svc.cluster-b.remote
  trafficPolicy:
    portLevelSettings:
   - port:
        number: 443
      tls:
      caCertificates: /etc/certs/cert-chain.pem
      clientCertificate: /etc/certs/cert-chain.pem
      mode: MUTUAL
      privateKey: /etc/certs/key.pem
```

Теперь создадим *VirtualService* в том же кластере *A*. Напомним нашу цель: распределить трафик между *A* и *B*. В примере 13.6 мы направляем 50 % трафика, предназначенного для сервиса Reviews, в кластер *B*, а оставшиеся 50 % оставляем для сервиса, работающего локально в кластере *A* (смотрите reviews-virtualservice.yaml в репозитории GitHub книги (https://oreil.ly/hYYtb)).

Пример 13.6 ❖ Определение VirtualService для распределения трафика Reviews между кластерами

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
```

```
metadata:
  generation: 1
  name: reviews-egress-splitter-virtual-service
  namespace: "default"
  resourceVersion: ""
  selfLink: /apis/networking.istio.io/v1alpha3/namespaces/default/virtualservices
            /reviews-egress-splitter-virtual-service
  uid: ""
spec:
  hosts:
 - reviews.default.svc.cluster.local
  http:
 - rewrite:
      authority: reviews.default.svc.cluster-b.remote
    route:
   - destination:
       host: svc.cluster-b.remote
       port:
         number: 443
     weight: 50
   - destination:
       host: reviews
     weight: 50
```

Теперь добавим дополнительные `VirtualService`, чтобы входящий трафик действительно мог попасть в сервис `Reviews`; другими словами, чтобы пользователи могли обращаться к приложению из публичного интернета. Пример 13.7 определяет настройки для кластера A. Здесь мы использовали шлюз, созданный ранее в этой главе, а также URI сервиса (смотрите bookinfo-vs.yaml в репозитории GitHub книги (https://oreil.ly/NkXU9)).

Пример 13.7 ❖ VirtualService для кластера A отображает входящий трафик в ProductPage

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  generation: 1
  name: bookinfo-vs
  namespace: "default"
  resourceVersion: ""
  selfLink: /apis/networking.istio.io/v1alpha3/namespaces/default
            /virtualservices/bookinfo-vs
  uid: ""
spec:
  gateways:
 - ingress-gateway
  hosts:
 - '*'
  http:
 - match:
   - uri:
       prefix: /productpage
    route:
```

```
  - destination:
      host: productpage
```

Для кластера *B* определим еще один `VirtualService`, чтобы трафик мог попадать в сервис напрямую. В примере 13.8 определяется удаленный сервис в атрибуте `hosts` и маршрут назначения (см. reviews-ingress-virtual-service.yaml в репозитории GitHub книги (https://oreil.ly/QZTLq)).

Пример 13.8 ❖ VirtualService для кластера B отображает входящий трафик в Reviews

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  generation: 1
  name: reviews-ingress-virtual-service
  namespace: "default"
  resourceVersion: ""
  selfLink: /apis/networking.istio.io/v1alpha3/namespaces/default/virtualservices
            /reviews-ingress-virtual-service
  uid: ""
spec:
  gateways:
 - ingress-gateway
  hosts:
 - reviews.default.svc.cluster-b.remote
  http:
 - route:
   - destination:
       host: reviews
```

К этому моменту 50 % трафика должно попасть в сервис в кластере *A* и 50 % – в сервис в кластере *B*. Мы объединили в сервисную сетку два разных кластера.

Используя пример из этой главы, можно поиграть с различными сценариями, не ограничиваясь равномерным разделением трафика. Кросс-кластер позволяет использовать и другие сценарии, в том числе обход отказов, включая разрывы связи между провайдерами или регионами, канареечное тестирование на менее дорогих кластерах в других местах и многое другое.

При планировании передовых топологий сетевое взаимодействие выходит на первый план. Стоит отметить, что мультикластер Istio не равнозначен мультикластеру из коробки. Самое основное требование заключается в том, что все кластеры Kubernetes должны иметь возможность пересылать трафик друг другу без перекрытия сети. Без должного планирования администраторы рискуют столкнуться с массой проблем, когда решат объединить свою инфраструктуру в единую сервисную сетку, такую как Istio. Поэтому перед внедрением мультикластера Istio проанализируйте текущую конфигурацию и топологию сети. Использовалось ли при создании каждого нового кластера одно и то же сетевое адресное пространство? Если да, то, возможно, понадобится реорганизовать кластеры. Достижимы ли все сервисы в обоих кластерах? Если нет, необходимо убедиться в правильной маршрутизации трафика.

Предметный указатель

Т

У

Ф

Х

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**.

Ли Калькот и Зак Бутчер

Istio: приступаем к работе

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Редактор *Киселев А. Н.*
Перевод *Бринь А. Л.*
Корректоры *Ганюшкина Е. В., 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. 19,18. Тираж 200 экз.

Отпечатано в ПАО «Т8 Издательские Технологии»
109316, Москва, Волгоградский проспект, д. 42, корпус 5

Веб-сайт издательства: **www.dmkpress.com**