O'REILLY®

Kubernetes и сети

Многоуровневый подход

Джеймс Стронг,
Валлери Лэнси

Джеймс Стронг
Валлери Лэнси

Kubernetes и сети

Многоуровневый подход

Санкт-Петербург
«БХВ-Петербург»
2024

УДК 004.451
ББК 32.973.26-018.2
С86

Стронг, Дж.

С86 Kubernetes и сети. Многоуровневый подход: Пер. с англ. / Дж. Стронг, В. Лэнси. — СПб.: БХВ-Петербург, 2024. — 320 с.: ил.

ISBN 978-5-9775-1855-0

Книга посвящена интеграции Kubernetes в готовые компьютерные сети. Рассмотрено, как оркестратор Kubernetes вписывается в сетевую модель OSI. Раскрыты вопросы интеграции сетей предприятия с облачными мощностями и контейнерными архитектурами. Рассмотрены ключевые факторы и новые зоны ответственности, возникающие при взаимодействии Kubernetes с каждым из уровней модели OSI. Приведены примеры быстрого масштабирования нагрузок, рассказано, как обеспечивать целостность данных и высокую отказоустойчивость, при активном применении современной виртуализации и передаче больших объёмов данных по сети.

*Для Linux-разработчиков, DevOps-инженеров
и системных администраторов*

УДК 004.451
ББК 32.973.26-018.2

Группа подготовки издания:

Руководитель проекта *Олег Сивченко*
Зав. редакцией *Людмила Гауль*
Редактор *Анна Ардашева*
Компьютерная верстка *Ольги Сергиенко*
Оформление обложки *Зои Канторович*

© 2024 BHV
Authorized Russian translation of the English edition of ***Networking and Kubernetes*** ISBN 9781492081654
© 2021 Strongjz tech and Vallery Lancey
This translation is published and sold by permission of O'Reilly Media, Inc , which owns or controls all rights to publish and sell the same.

Авторизованный перевод с английского языка на русский издания ***Networking and Kubernetes***
ISBN 9781492081654 © 2021 Strongjz tech и Vallery Lancey.
Перевод опубликован и продается с разрешения компании-правообладателя O'Reilly Media, Inc.

Подписано в печать 05 07 23
Формат 70×100$^{1}/_{16}$ Печать офсетная Усл печ л 25,8.
Тираж 1000 экз Заказ № 4415
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул , 20

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул Полиграфистов, д 1
Сайт www chpd ru, E-mail sales@chpd ru, тел. 8(499)270-73-59

ISBN 978-1-492-08165-4 (англ.)
ISBN 978-5-9775-1855-0 (рус.)

© Strongjz tech, Vallery Lancey, 2021
© Перевод на русский язык, оформление
ООО "БХВ-Петербург", ООО "БХВ", 2024

Оглавление

https://t.me/it_boooks/2

Предисловие ... 9
Просто еще один пакет? ... 9
Для кого эта книга ... 9
Что вы узнаете ... 10
Обозначения, используемые в данной книге ... 10
Использование примеров программ ... 10
Благодарности ... 11

Глава 1. Введение в сетевые технологии ... 13
История сетевых технологий ... 13
Модель OSI ... 16
TCP/IP ... 20
Уровень приложения ... 22
HTTP ... 22
Транспортный уровень ... 25
TCP ... 25
TLS ... 36
UDP ... 38
Уровень межсетевого взаимодействия ... 39
Протокол Интернета ... 39
Движение по сети ... 44
ICMP ... 47
Уровень канала данных ... 49
Снова наш веб-сервер ... 54
Заключение ... 56

Глава 2. Поддержка сети в ОС Linux ... 59
Базовые понятия ... 59
Сетевой интерфейс ... 63
Интерфейс сетевого моста ... 64
Обработка пакетов в ядре Linux ... 66
Netfilter (межсетевой фильтр) ... 66
Conntrack ... 70
Маршрутизация ... 72
Высокоуровневая маршрутизация ... 73
Утилита iptables ... 74
Таблицы *iptables* ... 75
Цепочки *iptables* ... 75
Подцепочки ... 79

Правила *iptables* 80
Практическое применение *iptables* 82
IPVS 85
eBPF 87
Средства сетевой диагностики 90
Безопасность 90
ping 91
traceroute 92
dig 93
telnet 95
nmap 96
netstat 97
netcat 98
Openssl 99
cURL 100
Заключение 102

Глава 3. Основы работы с контейнерами 103
Введение в контейнеры 103
Приложения 103
Гипервизор 104
Контейнеры 105
OCI 108
LXC 109
runC 109
containerd 109
lmctfy 110
rkt 110
Docker 110
CRI-O 111
Примитивы контейнеров 113
Контрольные группы 113
Пространства имен 114
Задание пространств имен 116
Основы сетей контейнеров 123
Сетевая модель Docker 132
Оверлейная сеть 134
Сетевой интерфейс контейнера 135
Подключение контейнера к сети 137
Соединение контейнер-контейнер 142
Взаимодействие между контейнерами на разных хостах 144
Заключение 145

Глава 4. Сети в Kubernetes 147
Сетевая модель Kubernetes 147
Узел и конфигурация сети подов 150
Изолированные сети 151
Плоские сети 152
Островные сети 153
Конфигурация компонента kube-controller-manager 154

Kubelet 155
Готовность пода и ее проверка 156
Спецификация интерфейса CNI 162
Плагины CNI 163
  Интерфейс IPAM 164
  Распространенные плагины CNI 165
Компонент kube-proxy 170
  Режим *userspace* 171
  Режим *iptables* 171
  Режим IPVS 173
  Режим kernelspace 173
Сетевая политика 173
  Создание объекта *NetworkPolicy* с помощью Cilium 177
  Группировка подов 182
    Тип *LabelSelector* 183
  Правила 185
DNS 189
Двойной стек IPv4/ IPv6 194
Заключение 196

Глава 5. Сетевые абстракции в Kubernetes 197
StatefulSet 198
Конечные точки 200
Endpoint Slices 204
Сервисы Kubernetes 208
  NodePort 209
  ClusterIP 212
  Headless-сервис 219
  Сервис ExternalName 221
  Сервис LoadBalancer 222
  Сервисы Kubernetes — устранение проблем 228
Ингресс 229
Контроллеры и правила ингресса 230
    Задание правил ингресса 236
Технология service mesh 237
Заключение 249

Глава 6. Kubernetes и облачные сети 251
Amazon Web Services 251
  Сетевые сервисы AWS 251
    Виртуальное частное облако 252
    Регионы и зоны доступности 252
    Подсеть 253
    Таблицы маршрутизации 254
    Эластичный сетевой интерфейс 256
    Эластичный IP-адрес 256
    Средства обеспечения безопасности 257
    Устройства преобразования сетевых адресов 260
    Шлюз Интернета 260
    Эластичные балансировщики нагрузки 261

Эластичный сервис Kubernetes от Amazon 264
Узлы EKS 264
Режим EKS 265
Инструмент *eksctl* 268
CNI для виртуального облака в AWS 270
Ингресс-контроллер для AWS ALB 272
Развертывание приложения в кластере AWS EKS 274
Развертывание кластера EKS 274
Развертывание тестового приложения 276
Тестирование сервиса LoadBalancer для веб-сервера 277
Развертывание и тестирование ингресс-контроллера для ALB 278
Уборка мусора 281
Вычислительное облако Google (GCP) 282
Сетевые сервисы GCP 282
Регионы и зоны 283
Виртуальное частное облако 283
Подсеть 284
Маршруты и правила брандмауэров 285
Облачная балансировка нагрузки 285
Инстансы GCE 286
Google Kubernetes Engine (GKE) 286
GKE-узлы в облаке Google 287
Azure 290
Сетевые сервисы Azure 290
Базовая инфраструктура Azure 291
Подсети 292
Таблицы маршрутизации 292
Публичные и частные IP-адреса 295
Группы сетевой безопасности 295
Взаимодействие вне пределов виртуальной сети 297
Балансировщик нагрузки в Azure 297
Azure Kubernetes Service 300
Плагин CNI для Azure 302
Ингресс-контроллер для шлюза приложения 303
Развертывание приложения с помощью Azure Kubernetes Service 304
Развертывание кластера с помощью Azure Kubernetes Service 305
Соединение с кластером AKS и его конфигурирование 310
Развертывание веб-сервера 313
Заключительные замечания по AKS 315
Заключение 315

Об авторах 317

Об обложке 319

Предисловие

Просто еще один пакет?

С тех пор как два компьютера были соединены вместе через кабель, сети стали важнейшей частью инфраструктуры. Сегодня, чтобы поддерживать всю палитру возможных применений, сети становятся все более сложными многоуровневыми конструкциями, и появление контейнеров и проектов, таких как Mesosphere и Kubernetes, не изменило положения дел. Хотя создатели Kubernetes и старались «свернуть» как можно больше этих сложных понятий в предлагаемых объектах API (абстракциях), но информатика развивается таким образом, что иерархия уровней постоянно усложняется. Kubernetes и его API — это набор средств, позволяющих интегрировать определенный уровень сложности и развертывать рабочие приложения проще и быстрее. А как быть администраторам, которые управляют сетями, работающими на базе Kubernetes? Данная книга снимает налет таинственности с абстракций Kubernetes, знакомит с уровнями сложности и помогает понять, что Kubernetes — это не просто еще один пакет для разработчика.

Для кого эта книга

Данная книга предназначена для чтения от начала до конца теми, кто только начал заниматься администрированием сетей, кластеров и систем под Linux. Она также может быть полезна и уже опытным разработчикам и администраторам (DevOps), желающим освоить новые для них аспекты. Сегодня администраторам Linux, сетей и кластеров необходимо хорошо представлять себе, как работать с приложениями Kubernetes в больших масштабах.

В книге читатели найдут информацию, необходимую для понимания сложной многоуровневой сети, какой является сеть Kubernetes, а также управления ею. Уровень за уровнем книга иллюстрирует иерархию, заложенную в объектах Kubernetes, так что разработчики увидят, как развертывать приложения локально, в облаке и с управляемыми сервисами. Инженеры, ответственные за функционирование рабочих кластеров и сетей, могут использовать книгу, чтобы наверстать недостающие знания в вопросах функционирования отдельных объектов.

Что вы узнаете

Прочитав книгу, читатель будет понимать, что такое:

- Сетевая модель Kubernetes.
- Интерфейсы контейнерных сетей (CNI) и как выбирать CNI-интерфейс для своих кластеров.
- Сетевые примитивы и примитивы Linux, обеспечивающие работу Kubernetes.
- Взаимосвязь между объектами, обеспечивающими работу сетей Kubernetes.

Читатель также сможет:

- Развертывать сеть производственных масштабов для кластеров Kubernetes и управлять ею.
- Устранять проблемы в сетевых приложениях, работающих внутри кластеров Kubernetes.

Обозначения, используемые в данной книге

Этот элемент указывает на совет или предложение.

Этот элемент указывает на общее замечание.

Этот элемент указывает на предостережение или осторожность в использовании информации.

Использование примеров программ

Дополнительный материал (коды программ, упражнения и пр.) доступен для скачивания по адресу *https://github.com/strongjz/Networking-and-Kubernetes*.

Если у вас возникнут технические вопросы или проблемы, связанные с использованием образцов программ, пожалуйста, обращайтесь по электронной почте:

bookquestions@areilly.com.

Эта книга предназначена помочь вам в вашей работе. Предлагаемые в книге примеры вы можете использовать в ваших собственных программах или документах. Вам не нужно спрашивать нашего разрешения на использование кода, за исключением случаев, когда вы собираетесь репродуцировать его значительную часть. Например, использование фрагментов кода из книги при написании вашей программы не требует разрешения.

Благодарности

Авторы выражают благодарность сотрудникам «O'Reilly Media» за помощь в написании их первой книги. *Мелисса Поттер* обеспечила нам техническую поддержку в течение всего процесса работы над книгой. Мы также особо благодарны *Томасу Бенкену* за его советы по использованию Azure.

Джеймс: *Карин*, спасибо тебе за веру в мои силы и за то, что ты верила в меня, когда я сам был далек от этого. *Винк*, ты побудил меня работать в этой области, и я тебе за это вечно благодарен. *Энн*, я прошел долгий путь, поскольку считается, что изучение языка рано или поздно окупится. Я хочу также поблагодарить всех остальных своих учителей и преподавателей, которые поддерживали меня в течение всей жизни.

Валлери: я хотела бы поблагодарить дружелюбных сотрудников SIG-Network, которые помогли мне в освоении тонкостей Kubernetes.

Авторы благодарят также все сообщество пользователей Kubernetes — эта книга не появилась бы на свет без их участия. Мы надеемся, что она будет полезна всем разработчикам, которые намереваются применять Kubernetes в своей работе.

ГЛАВА 1

Введение в сетевые технологии

https://t.me/it_boooks/2

«Виновен, пока не доказано обратное». Это основной закон сетей и обслуживающих их инженеров. В этой вводной главе мы рассмотрим развитие сетевых технологий и стандартов, дадим краткое описание лидирующей на сегодня теории сетей и познакомимся с нашим веб-сервером на языке Go, который на протяжении всей книги будет использоваться для примеров работы Kubernetes и облачных сервисов.

Итак, начнем... с начала.

История сетевых технологий

Интернет, который мы знаем сегодня, — это скрытый от глаз гигантский клубок кабелей, идущих через моря и горы и соединяющих города и поселки между собой. Картина Баррета Лайона «Визуализация Интернета», приведенная на рис. 1.1, хорошо демонстрирует эту грандиозность — множество подсетей объединяются и образуют мировую сеть Интернет.

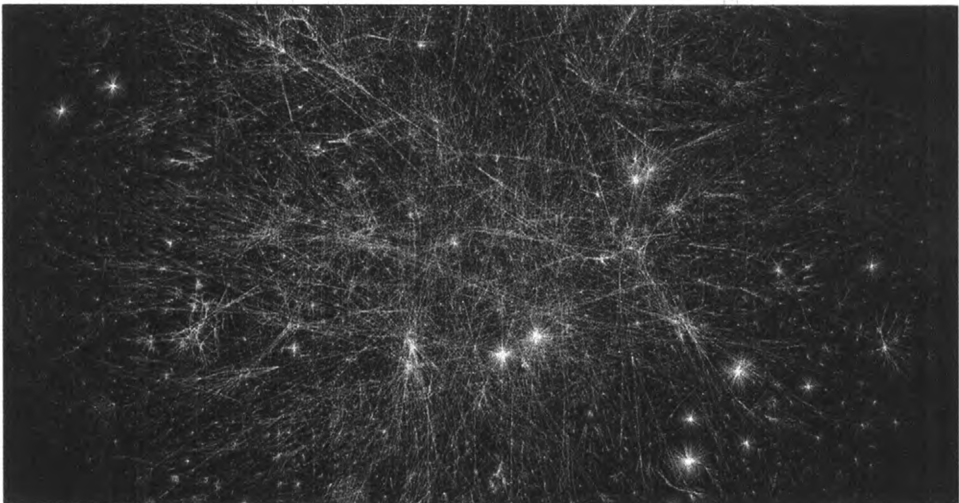


Рис. 1.1. Баррет Лайон, «Визуализация Интернета», 2003

Целью сетевой структуры является обмен информацией между отдельными ее системами. Это порождает запрос на глобальную распределенную систему, но Интернет не всегда был глобальным — он появился сначала как концептуальная модель,

а затем постепенно развился до сложной конструкции, великолепно показанной на картине Лайона.

Изучая работу с сетями, приходится рассматривать многие факторы, такие, например, как «последняя миля» — участок сети между домом пользователя и его провайдером Интернета, и вплоть до масштабов уровня всей планеты. Интернет плотно интегрирован в саму ткань нашего общества. В данной книге мы обсудим, как работают сетевые технологии и какую поддержку этой работы может дать Kubernetes.

Прежде чем мы приступим к рассмотрению отдельных важных деталей, приведем в табл. 1.1 краткую историю развития сетевых коммуникаций.

Таблица 1.1. *Краткая история развития сетевых коммуникаций*

Год	Событие
1969	Первое соединение в сети ARPANET
1969	Создание стандарта RFC 15 (Request for Comments — рабочее предложение)
1971	Стандарт RFC 114 для FTP
1973	Стандарт RFC 354 для FTP
1974	Стандарт RFC 675 T для CP, авторы Винт Сёрф, Йоген Далал, Карл Саншайн
1980	Начинается разработка модели взаимодействия открытых систем
1981	Стандарт RFC 760 для IP
1982	NORSAR и университетский колледж Лондона перестают пользоваться ARPANET и переходят на TCP/IP в рамках SATNET
1984	Публикация стандарта ISO 7498 для модели взаимодействия открытых систем OSI
1991	Альберт Гор способствует принятию Билля о национальной информационной инфраструктуре
1991	Выходит первая версия Linux
2015	Выходит первая версия Kubernetes

На заре существования компьютерные сети были либо государственными, либо создавались с финансовой поддержкой государства. Так, в США Министерство обороны (Department of Defence, DOD) поддерживало сеть Агентства передовых исследовательских проектов ARPPANET еще задолго до появления в политике Альберта Гора-младшего, будущего вице-президента США (о его роли расскажем ниже).

В 1969 году сеть ARPANET была развернута на базе Университета Калифорнии в Лос-Анджелесе, исследовательского центра Стэнфордского университета, Университета Калифорнии в Санта-Барбаре и Университета штата Юта. Однако коммуникация между этими узлами полностью начала функционировать только в 1970 г., когда они стали применять протокол сетевого управления NCP. Использование NCP привело к необходимости появления первых протоколов связи между отдельными компьютерами: Telnet и FTP (протоколы передачи файлов).

Успех ARPPANET в сочетании с первым сетевым протоколом NCP привел к тому, что NCP «ушел с рынка», поскольку не мог соответствовать ни требованиям отдельной сети, ни разнообразию подсетей, объединяемых в общую сеть.

В 1974 г. Винт Сёрф, Йоген Далал и Карл Саншайн начали разработку стандарта RFC 675 для протокола управления передачей данных TCP (подробнее о RFC будет сказано чуть позже). Постепенно TCP стал основным стандартом связи.

В 1981 г. с помощью протокола маршрутизации Интернета IP (Internet Protokol), созданном в RFC 791, некоторые функции TCP были выделены отдельно, что способствовало увеличению модульности сети. В последующие годы многие организации, включая и Министерство обороны США, приняли TCP в качестве стандарта. К январю 1983 г. протокол TCP/IP, заменивший NCP благодаря своей гибкости и модульности, стал единственным одобренным стандартом ARPANET.

Другая организация, также занимающаяся разработкой стандартов, — Международная организация по стандартизации ИСО (ISO), разработала и опубликовала в ISO 7498 свою Базовую модель взаимодействия открытых систем, названную моделью OSI. Одновременно с публикацией появились и поддерживающие модель протоколы. К сожалению, протоколы OSI не получили широкого использования и уступили место TCP/IP. Однако и сегодня модель OSI остается удобным примером для знакомства с многоуровневым подходом в сетевых коммуникациях.

В 1991 г. Альберт Гор изобрел Интернет (ну, на самом деле он помог провести Билль о национальной информационной инфраструктуре), что привело к созданию IETF — Инженерного совета Интернета. В настоящее время IETF является открытым консорциумом, куда входят ведущие эксперты и компании, работающие в сфере сетевых технологий, как, например, Cisco и Juniper, и занимается разработкой стандартов для Интернета. Стандарты RFC публикуются как службой IETF, так и отдельными специалистами или группами компьютерных инженеров и ученых, которые подробно описывают процессы, операции и приложения, поддерживающие работу Интернета.

Стандарты RFC IETF выходят в нескольких вариантах:

Предложенный стандарт (Proposed Standard)

Предложенный стандарт получил достаточную поддержку в среде разработчиков, чтобы рассматриваться как единый стандарт. Структура устоялась и хорошо понятна. Предлагаемый стандарт может быть развернут, успешно внедрен и протестирован. Однако он может быть и исключен из дальнейшего рассмотрения.

Стандарт Интернета (Internet Sandard)

Согласно RFC 2026: «Как правило, стандарт Интернета — это устоявшаяся, хорошо понимаемая и технически грамотная спецификация, имеющая многочисленные, независимые, совместимые реализации с достаточной историей работы, пользующаяся значительной общественной поддержкой и признанно полезная для определенных сегментов Интернета».

Проект стандарта (Draft Standard) был третьим вариантом, но прекратил использоваться в 2011 г.

Существуют тысячи стандартов Интернета, определяющих, как реализовывать протоколы для всех аспектов сетевых коммуникаций, включая среди прочего беспроводную связь, алгоритмы шифрования и форматы представления данных. Каждый их них реализуется либо как свободно распространяемый проект, либо как коммерческий, например, крупными частными фирмами.

За 50 лет, прошедших с установки первого сетевого соединения, случилось много событий. Значительно возросла сложность сетей и их формализация. Наше рассмотрение начнем с модели OSI.

Модель OSI

Модель OSI — концепция, описывающая взаимодействие сетевых устройств. Эта модель членит процесс передачи данных в Сети на отдельные уровни. Она наглядно показывает, как уровни взаимодействуют между собой и как данные проходят через сеть. Первоначально предполагалось, что протоколы модели станут сетевыми стандартами, но они проиграли конкуренцию протоколам TCP/IP.

Ниже приводятся стандарты ISO, которые описывают модель OSI и протоколы:

- ISO/IEC 7498-1, Базовая модель;
- ISO/IEC 7498-2, Архитектура безопасности;
- ISO/IEC 7498-3, Имена и адресация;
- ISO/IEC 7498-4, Администрирование.

Спецификация ISO/IEC 7498-1 (российский ГОСТ Р ИСО/МЭК 7498-1-99) описывает базовые понятия модели OSI:

> 5.2.2.1. Основной структурной единицей в Базовой модели взаимодействия открытых систем является уровень. Согласно данной модели каждая открытая система рассматривается как логически связанное упорядоченное множество (N) подсистем... Соседние (N)-подсистемы обмениваются данными через общую границу этих систем. (N)-подсистем одного и того же ранга (N) образуют (N)-уровень Базовой модели взаимодействия открытых систем. В открытой системе имеется одна и только одна (N)-подсистема для уровня N. (N)-подсистема состоит из одного или нескольких (N)-объектов. Объекты существуют на каждом (N)-уровне. Объекты одного и того же (N)-уровня называются равноправными объектами (N)-уровня (peer entity). Отметим, что высший уровень не имеет над собой уровня (N+1), а низший уровень не имеет под собой уровня (N-1).

Модель OSI представляет собой довольно сложное и формализованное описание конструкции сетей в виде луковичной структуры или слоеного пирога: функционал Сети разбивается на семь отдельных уровней, каждый уровень со своими собственными задачами в процессе передачи информации от одной системы к другой.

Структура модели показана на рис. 1.2. Уровни инкапсулируют информацию с предыдущего уровня; выделяются уровень приложения, уровень презентации, уровень

сеанса, транспортный уровень, сетевой уровень, уровень канала данных и физический уровень. На следующих страницах мы разберем функциональность каждого уровня и уточним, каким образом происходит пересылка данных между двумя системами.

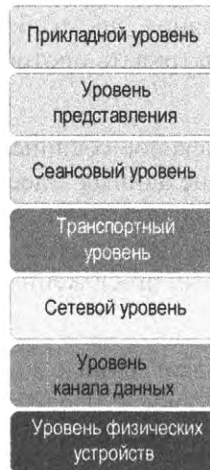


Рис. 1.2. Уровни модели OSI

Каждый уровень принимает данные предыдущего уровня и инкапсулирует их в своем PDU (Protocol Data Unit — блоке данных протокола). PDU являются также частью протоколов TCP/IP и служат для описания данных на каждом уровне.

Приложения сеансового уровня рассматриваются как данные для блока PDU, подготавливающего информацию приложения к передаче.

Транспортный уровень использует порты, чтобы определить, какой процесс в локальной системе отвечает за данные.

Блок PDU сетевого уровня — это пакет. Пакеты — это определенные фрагменты данных, пересылаемые между сетями.

Уровень канала данных — это кадр (frame) или сегмент. Каждый пакет разбивается на кадры, контролируется на ошибки и посылается по локальной сети.

На уровне устройства происходит передача кадра в битах по физической среде.

А теперь давайте рассмотрим каждый уровень более подробно.

Прикладной уровень

Самый верхний уровень модели OSI, и именно с этим уровнем взаимодействуют конечные пользователи, когда работают с приложениями. Это не какое-то место, где находятся реальные приложения, но та структура, которая предоставляет интерфейс для взаимодействия с ними, например веб-браузер или офисные программы. Самый известный пример — это HTTP. Другими протоколами прикладного уровня, с которыми мы часто контактируем, являются DNS, SSH и SMTP. Приложения этих протоколов отвечают за отображение и упорядочивание данных, запрашиваемых и пересылаемых по сетям.

Уровень представления

Этот уровень обеспечивает независимость от способа представления данных путем их перевода из формата приложения в сетевой формат. Его еще можно назвать *синтаксическим уровнем*. Он позволяет двум системам использовать различные кодировки и тем не менее обмениваться данными между собой. Здесь же происходит и процесс шифрования, но это более сложная процедура, которую мы опишем, когда будем рассматривать протокол безопасности TLS.

Сеансовый уровень

Этот уровень отвечает за дуплексное соединение, другими словами — за одновременную посылку и получение данных. Здесь также присутствуют процедуры, выполняющие проверку контрольных точек, прерывание, повторный старт и завершение сеанса. Уровень устанавливает, поддерживает и завершает соединения между локальными и удаленными приложениями.

Транспортный уровень

Транспортный уровень пересылает данные между приложениями. Контроль потока данных, сегментации и сборки, а также контроль ошибок обеспечивает надежную связь между верхними уровнями. Отдельные протоколы отслеживают состояние соединения. Здесь же проверяются сегменты и повторно передаются те, в которых возникла ошибка. А также подтверждается успешная передача данных и посылается следующий пакет. TCP/IP имеет два протокола на транспортном уровне: TCP и протокол пользовательской датаграммы UDP.

Сетевой уровень

Сетевой уровень реализует передачу потоков данных переменной длины от хоста одной сети к хосту другой сети при одновременной поддержке качества связи. Также уровень осуществляет маршрутизацию и может осуществлять фрагментацию и сборку, сообщая об ошибках доставки. На этом уровне задействуются маршрутизаторы, которые посылают данные через соседние сети. Сетевой уровень использует несколько протоколов, включая протоколы маршрутизации, управление многоадресными группами, информацию сетевого уровня, обработку ошибок, присвоение адресов — все это мы подробнее рассмотрим ниже в разделе «Протоколы TCP/IP».

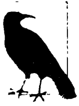
Уровень канала данных

Этот уровень отвечает за передачу данных между отдельными хостами одной и той же сети. Он определяет протоколы для установления и прекращения соединения между двумя устройствами. Уровень канала передает данные между точками сети и предоставляет средства для обнаружения и возможного исправления ошибок уровня физических устройств. Кадры данного уровня, т. е. его PDU, не пересекают границы локальной сети.

Физический уровень

Физический уровень наглядно представлен кабелем Ethernet, подключенным к вашему устройству. На этом уровне данные из цифровых битов преобразуются в электрические, радио- или оптические сигналы. Уровень состоит из физиче-

ских устройств таких, как кабели, коммутаторы и беспроводные точки доступа. Он также определяет протоколы обработки сигналов.



Существует много мнемонических правил, чтобы легче запомнить уровни модели OSI, наше любимое — это «All People Seem To Need Data Processing» («Все Люди нуждаются В Обработке Данных»). На русском, увы, эта фраза теряет свое мнемоническое значение, так что попробуйте заменить ее практическим советом «Предложи Представителю Сеанс Транспорта, и Сеть Канала Физически упадет».

В табл. 1.2 перечисляются все уровни модели OSI вместе с кратким описанием их функций.

Таблица 1.2. *Уровни модели OSI*

Номер уровня	Уровень	PDU	Функции
7	Прикладной уровень	Данные	Приложения высокого уровня и протоколы HTTP, DNS, SSH
6	Уровень представления	Данные	Кодировка символов, сжатие данных, шифрование/дешифровка
5	Сеансовый уровень	Данные	Управление непрерывным обменом данными между хостами: сколько данных передать, когда передать следующую порцию
4	Транспортный уровень	Датаграмма, сегмент	Передача сегментов данных по сети между конечными точками, включая сегментацию, подтверждение и мультиплексирование
3	Сетевой уровень	Пакет	Структурирование и управление адресацией, маршрутизация и управление трафиком для всех конечных точек сети
2	Уровень канала данных	Кадр	Передача кадров данных между двумя хостами в сети
1	Физический уровень	Бит	Посылка и получение потоков битов по физическим каналам

Модель OSI содержит все функции, необходимые для передачи по сети пакета данных между двумя хостами. В конце 1980-х — начале 1990-х она уступила место протоколам TCP/IP, которые стали использоваться Министерством обороны США и ведущими компаниями, работающими в сфере компьютерных сетей.

Стандарт, определяемый в документе ISO 7498, дает представление о некоторых особенностях модели, реализация которых многим специалистам казалась сложной, неэффективной и даже просто невозможной. Однако модель OSI позволяет пользователям, изучающим сетевые технологии, лучше понять базовые концепции и проблемы, возникающие при их реализации. Кроме того, терминология модели и ее функции используются и в протоколахTCP/IP, описываемых в следующем разделе, и в концепциях Kubernetes.

Сервисы Kubernetes используют функции в зависимости от уровня, на котором они применяются, например IP-адрес на уровне 3 или порт на уровне 4; подробно мы будем обсуждать это в *главе 4*. А в следующем разделе мы рассмотрим протоколы TCP/IP и приведем пример, как они работают.

TCP/IP

Семейство протоколов TCP/IP создает гетерогенную сеть с открытыми протоколами, которые не зависят от операционной системы и деталей архитектуры. Работают ли хосты под управлением Windows, Linux или другой ОС, протоколы TCP/IP дают им возможность связываться между собой, для TCP/IP не имеет значения, работает ли ваш веб-сервер на уровне приложений на Apache или Nginx. Это обеспечивается распределением функций, похожим на то, как они реализуются в модели OSI. Рис. 1.3 сопоставляет модель OSI и TCP/IP.

Рис. 1.3. Сравнение моделей OSI и TCP/IP

Разберем подробнее отличия между моделями OSI и TCP/IP.

Уровень приложения

В TCP/IP уровень приложения содержит все протоколы связи, которые используются для обмена информацией между процессами внутри IP-сети. Уровень приложения стандартизует связь и организует передачу данных между хостами в зависимости от протоколов нижнего транспортного уровня. Также транспортный уровень управляет обменом данными в межсетевых коммуникациях. Приложения определяются в документах RFC; в данной книге мы будем использовать HTTP, RFC 7231 для иллюстрации прикладного уровня.

Транспортный уровень

TCP и UDP являются основными протоколами транспортного уровня, которые предоставляют приложениям сервисы обмена информацией между хостами.

Транспортные протоколы отвечают за установление соединения, надежность передачи, контроль потока данных и мультиплексирование. В TCP поток данных управляется размером окна, в отличие от него, UDP не контролирует переполнение потока и поэтому считается ненадежным (подробнее будет сказано в разделе «UDP» ниже). Каждый порт идентифицирует процесс на хосте, отвечающий за обработку сетевой информации. Так, HTTP использует хорошо известный порт 80 для небезопасного соединения и порт 443 — для безопасного. Каждый порт на сервере идентифицирует свой трафик, отправитель для своей идентификации генерирует локальный случайный порт. Организация, управляющая присвоениями номера порта, — это IANA (Администрация адресного пространства Интернета). Всего существует 65 535 портов.

Уровень межсетевого взаимодействия (Интернета)

Уровень межсетевого взаимодействия, или Интернета, отвечает за передачу данных между сетями. Для исходящего пакета он определяет ближайший хост и пересылает пакет этому хосту путем передачи его соответствующему каналу. Как только пакет получен адресатом, уровень Интернета сообщает соответствующему протоколу транспортного уровня, какова была полезная нагрузка пакета.

Протокол IP обеспечивает фрагментацию или сборку пакета на основе MTU — максимального размера IP-пакета. Протокол IP не гарантирует успешное прибытие пакета. Поскольку процесс передачи пакета через разнообразные сети является в принципе ненадежным и неустойчивым к отказам, контроль прибытия пакета ложится на конечные точки коммуникационного пути, а не на саму сеть. Функции, обеспечивающие услуги надежности, относятся к транспортному уровню. Контрольная сумма гарантирует, что информация в полученном пакете передана верно, но этот уровень не верифицирует сохранность данных. Пакеты в сети идентифицируются через IP-адреса.

Уровень канала данных

В модели TCP/IP уровень канала содержит сетевые протоколы, действующие только в пределах локальной сети, к которой подсоединен хост. Пакеты не направляются в нелокальные сети — эту функцию выполняет уровень Интернета. Ведущим протоколом данного уровня является Ethernet, хосты идентифицируются через их адреса на этом уровне или в общем случае через MAC-адреса их сетевых карт. После идентификации данных хостом с использованием протокола ARP 9 (протокол разрешения адреса) переданные их локальной сети данные обрабатываются на уровне Интернета. Этот уровень также включает в себя протоколы для пересылки пакетов между двумя хостами уровня Интернета.

Физический уровень

Физический уровень определяет сетевые устройства и компоненты. Он задает физические характеристики коммуникационной среды, например стандарт IEEE 802.3, специфицирующий сетевую среду для Ethernet. Некоторые положения документа RFC 1122 для физического уровня реализованы на других уровнях, мы приводим это здесь просто для полноты описания.

На протяжении данной книги мы будем использовать минимальный веб-сервер на языке Go из примера 1.1 для иллюстрации различных сетевых компонентов из tcpdump, системного вызова Linux, чтобы показать, как Kubernetes работает с системными вызовами. В данной главе мы обратимся к нему, чтобы продемонстрировать, что происходит на уровнях приложения, транспортном, межсетевом и канала данных.

Уровень приложения

Как уже было сказано, уровень приложения — это самый верхний уровень иерархии TCP/IP, то место, где пользователь взаимодействует с данными перед тем, как передать их в Сеть. В нашем примере мы будем использовать протокол передачи гипертекста HTTP и простую HTTP-транзакцию, чтобы посмотреть, что происходит на каждом уровне модели TCP/IP.

HTTP

Протокол HTTP отвечает за пересылку и получение HTML-документов, т. е. обычных веб-страниц. Большинство того, что мы делаем в Интернете — покупки на маркетплейсах, посты в социальных сетях, общение в мессенджерах, — осуществляется через HTTP.

Клиент сгенерирует HTTP-запрос на наш минимальный Go веб-сервер из примера 1.1, и тот пошлет HTTP-ответ с текстом "Hello". Веб-сервер запущен локально на виртуальной машине под Ubuntu. Этот пример продемонстрирует нам работу всех уровней модели TCP/IP.

Полный листинг можно посмотреть в библиотеке кодов.

Пример 1.1. Минимальный веб-сервер на языке Go

```
package main
import (
        "fmt"
        "net/http"
)
func hello(w http.ResponseWriter, _ *http.Request) {
        fmt.Fprintf(w, "Hello")
}
func main() {
        http.HandleFunc("/", hello)
        http.ListenAndServe("0.0.0.0:8080", nil)
}
```

На нашей виртуальной машине с Ubuntu нам надо запустить минимальный веб-сервер или, если у вас уже установлен Go, выполнить команду:

```
Go run web-server .Go
```

Рассмотрим, как запрос проходит через все уровни TCP/IP.

Утилита cURL является запрашивающим клиентом для нашего HTTP-запроса. Как правило, клиентом будет веб-браузер, но мы используем cURL для простоты и чтобы показать командную строку.

cURL предназначена для загрузки и скачивания данных, располагающихся по указанному URL. Это программа стороны клиента (на это указывает буква с в начале) для запроса данных от URL и возврата ответа на запрос.

В примере 1.2 мы увидим все части HTTP-запроса, формируемого клиентом cURL, и ответ на запрос. Рассмотрим выдачи и опции подробно.

Пример 1.2. Запрос стороны клиента

```
o → curl localhost:8080 -vvv 1
      Trying ::1...
  TCP_NODELAY set
* Connected to localhost (::1) port 8080 2
* GET / HTTP/1.1 3
* Host: localhost:8080 4
> User-Agent: curl/7.64.1 5
> Accept: */* 6
>
> HTTP/1.1 200 OK 7
< Date: Sat, 25 Jul 2020 14:57:46 GMT 8
< Content-Length: 5 9
< Content-Type: text/plain; charset=utf-8 10
   <
* Connection #0 to host localhost left intact 11
Hello* Closing connection 0
```

Пояснение к листингу:

1. `curl localhost:8080 -vvv`: это команда `curl`, которая открывает соединение с локально запущенным веб-сервером `localhost` на TCP порте 8080. `-vvv` устанавливает полную выдачу, чтобы мы могли проследить все, что происходит с запросом. `TCP_NODELAY` дает указание TCP соединению посылать данные без задержки — это одна из опций, которые можно установить на клиенте.
2. `Connected to localhost (::1) port 8080`: Наша программа сработала! `cURL` соединился с веб-сервером на `localhost` через порт 8080.
3. `Get / HTTP/1/1`: протокол HTTP предоставляет несколько возможностей для получения или обновления данных. В нашем запросе мы используем HTTP-

команду `GET`, чтобы получить ответ "Hello" на наш запрос. Прямой слеш — это следующая часть, унифицированный указатель ресурса URL, показывающий, откуда мы посылаем запрос на сервер. Последняя часть строки указывает, какую версию HTTP использует сервер 1.1.

4 `Host: localhost:8080`: HTTP предоставляет несколько опций для посылки информации о запросе. В нашем запросе процесс `cURL` установил значение HTTP- заголовка `Host`. Клиент и сервер могут обмениваться информацией, используя HTTP-запрос или ответ на него. HTTP-заголовок содержит название протокола, затем следует двоеточие и значение.

5 `User-Agent: curl/7.64.1`: user agent — это строка, которая идентифицирует компьютерную программу, формирующую HTTP-запрос от конечного пользователя, — в нашем контексте это `curl`. Строка часто указывает на браузер, номер его версии и операционную систему его хоста.

6 `Accept: */*`: этот заголовок сообщает веб-серверу, какие типы содержимого понимает клиент. В табл. 1.3 приводятся примеры типов, которые могут быть переданы.

7 `HTTP/1.1 200 OK`: это ответ сервера на наш запрос. Сервер отвечает, указывая версию HTTP и код состояния ответа. Возможны несколько типов ответов от сервера. Код состояния 200 показывает, что ответ успешный. 1XX означает информирующий ответ, 2XX — успешный, 3XX — переадресацию, 4XX — проблемы с ответом, 5XX — проблемы с сервером.

8 `Date: Sat, July 25, 2020, 14:57:46 GMT`: поле заголовка `Date` содержит дату и время сообщения. Сервер генерирует значение, примерно соответствующее дате и времени генерации сообщения.

9 `Content-Length:5`: заголовок `Content-Length` указывает размер посланного получателю тела сообщения в байтах, в нашем случае — 5 байт.

10 `Content-Type: text/plain; charset=utf-8`: заголовок `Content-Type` используется, чтобы указать тип содержимого. В нашем случае возвращается обычный текстовый файл вкодировке UTF-8.

11 `Hello* Closing connection 0`: выводится ответ от веб-сервера и завершается HTTP-соединение.

Таблица 1.3. *Стандартные типы содержимого для HTTP данных*

Тип	Описание
Приложение	Любые двоичные данные, которые явно не попадают в другие типы. Примерами могут служить файлы с расширением json, pdf, pkcs8, zip
Аудио	Аудио или музыка. Примеры: * mpeg, * vorbis
Шрифты	Примеры. *.woff, *.ttf, *.otf
Картинки	Картинки или графика, включая растровую и векторную, анимация GIF или APNG. Примеры: * jpg, *.png, *.svg+xml

***Таблица 1.3** (окончание)*

Тип	Описание
Модели	Модельные данные для 3D-объекта или сцены. Примеры. * 3mf, * vrml
Текст	Тип данных text-only включает себя как тексты в обычном понимании, так это может быть и текст программы. Примеры: *.plain, *.csv, *.html
Видео	Видеоданные или файлы Пример. *.mp4

Мы рассмотрели на простом примере, что происходит при выполнении HTTP-запроса. Сегодня одна веб-страница посылает огромное число запросов при каждой своей загрузке — и все это за доли секунды! Этот пример демонстрирует системным администраторам, как работает HTTP (и другие приложения всех уровней). Мы продолжим углублять наши знания о том, как запрос обрабатывается на каждом из уровней TCP/IP и как с теми же самыми запросами обращается Kubernetes. Форматирование данных и установка опций происходят на уровне 7, но основная работа выполняется на более низких уровнях TCP/IP, которые мы рассмотрим в следующих разделах.

Транспортный уровень

Протоколы транспортного уровня отвечают за передачу данных и ее надежность, осуществляют управление потоком данных и мультиплексирование — по большей части это относится и к протоколу TCP. Его особенности будут описаны ниже. Наш Go веб-сервер — приложение 7-го уровня, использующее HTTP; HTTP при этом опирается на протокол TCP.

TCP

Как сказано выше, TCP — это ориентированный на соединение, надежный протокол, обеспечивающий контроль потока и мультиплексирование. Протокол контролирует состояние соединения в течение всего цикла соединения. В TCP ширина канала управляет потоком данных — в отличие от UDP, который не проводит контроль перегрузки канала. Кроме того, UDP не является надежным, и последовательность данных может нарушаться. Каждый порт идентифицирует на хосте процесс, отвечающий за обработку сетевой информации. И TCP является именно протоколом коммуникации между хостами. Чтобы определить процесс, отвечающий за соединение, TCP идентифицирует сегменты с помощью 16-битового номера порта. HTTP-сервер использует хорошо известный порт 80 для небезопасного соединения и 443 — для безопасного, используя протокол безопасности транспортного уровня TLS. Клиенты, запрашивающие новое соединение, локально создают порт источника с диапазоном значений от 0 до 65 534.

Чтобы понять, как TCP выполняет мультиплексирование, рассмотрим вызов простой HTML-страницы:

1. В веб-браузере набрать адрес веб-страницы.
2. Браузер устанавливает соединение, чтобы передать страницу.
3. Браузер устанавливает соединения для каждой картинки на странице.
4. Браузер открывает соединение для внешних CSS (каскадные таблицы стилей)
5. Каждое из этих соединений использует различный набор виртуальных портов.
6. Все составляющие страницы загружаются одновременно.
7. Браузер представляет страницу пользователю.

Посмотрим, как TCP осуществляет мультиплексирование, используя информацию, содержащуюся в заголовках сегментов TCP:

Source port (16 бит)

Идентификатор исходного порта.

Destination port (16 бит)

Идентификатор порта назначения.

Sequence number (32 бита)

Если флаг SYN установлен, то это будет начальный порядковый номер. Порядковый номер первого байта данных и порядковый номер подтверждения в соответствующем сообщении ACK будет начальный номер плюс 1. Используется также для сборки данных, если они приходят с нарушением последовательности.

Acknowledgment number (32 бита)

Если флаг ACK установлен, то значение этого поля есть следующий номер подтверждения ACK, который ожидает передающая сторона. Это подтверждает получение всех предыдущих байтов (если они были). Первое ACK каждой конечной точки подтверждает начальный порядковый номер другой конечной точки, хотя данные при этом не пересылались.

Data offset (4 бита)

Задает размер заголовка сегмента TCP в 32-битовых словах.

Reserved (3 бита)

Предназначено для будущего использования, устанавливается в 0.

Flags (9 бит)

Для TCP-заголовка определены девять однобитовых полей:

- NS: ECN-nonce — защита сокрытия.
- CWR: окно перегрузки уменьшено — передающая сторона уменьшает скорость передачи.
- ECE: ECN-echo — передающая сторона ранее получила сообщение о перегрузке сети.

- URG: флаг срочности — активирует указатель поля Urgent, используется редко.
- ACK: подтверждение — активирует поле Acknowledgment number, флаг остается установленным на протяжении всей длительности соединения.
- PSH: получающая сторона должна передать данные приложению как можно быстрее.
- RST: переустановка или прерывание соединения, обычно из-за ошибки.
- SYN: синхронизировать порядковые номера, чтобы инициировать соединение.
- FIN: передающая сторона закончила передачу данных.

Битовое поле NS объясняется в спецификации RFC3540, «Robust Explicit Congestion Notification (ECN) Signaling with Nonces» («Устойчивый механизм сигнализации насыщения с помощью ECN-nonce»). Документ описывает опциональные расширения ECN, предназначенные для повышения устойчивости к намеренному или случайному переполнению потока данных для маркетплейсов.

Window size (16 бит)

Задает размер окна принимающей стороны.

Checksum (16 бит)

Поле контрольной суммы используется для проверки на ошибки TCP-заголовка.

Urgent pointer (16 бит)

Сдвиг от порядкового номера, указывающий на последний байт данных Urgent (Важно).

Options

Переменные 0–320 бита, в словах по 32 бита.

Padding

Параметр Padding заголовка TCP используется для указания, что заголовок закончился, и после 32 бита пойдут данные.

Data

Данные приложения, пересылаемые в указанном сегменте.

На рис. 1. 4 показаны все TCP-заголовки сегмента, которые содержат всю необходимую информацию по TCP-потокам.

Указанные поля помогают управлять потоком данных между двумя системами. Рис. 1.5 иллюстрирует, как каждый уровень иерархии TCP/IP участвует в передаче данных от приложения на одном хосте на другой хост, используя межсетевое взаимодействие на уровнях 1 и 2.

В следующем разделе мы покажем, как TCP использует эти поля, чтобы установить соединение, используя механизм тройного квитирования.

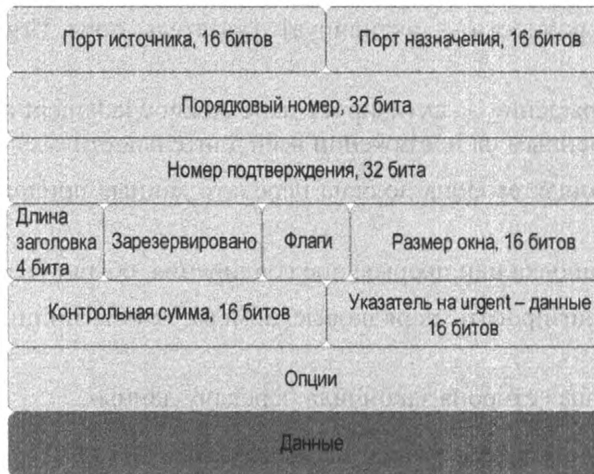


Рис. 1.4. Заголовок TCP-сегмента

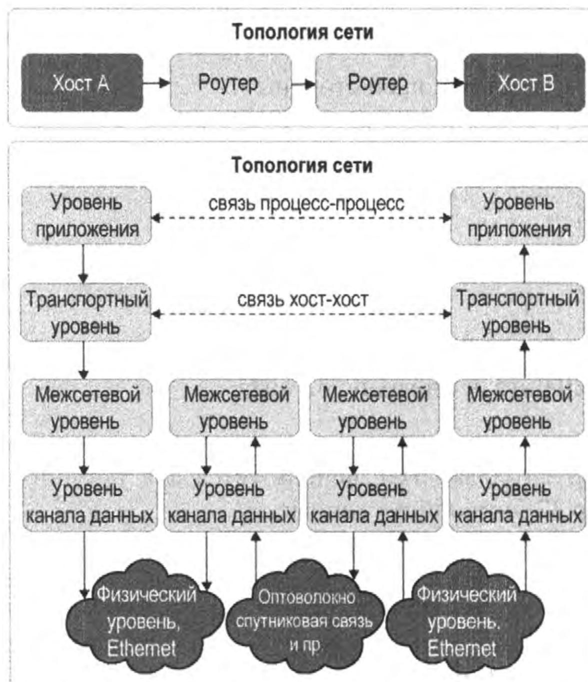


Рис. 1.5. Поток данных в модели TCP/IP

Квитирование в протоколе TCP

TCP использует процедуру тройного квитирования, показанную на рис. 1.6, для установления соединения. Обмен информацией регулируется с помощью различных опций и флагов:

Рис. 1.6. Механизм тройного квитирования в протоколе TCP

1. Запрашивающий хост посылает запрос на соединение путем передачи пакета SYN, чтобы инициировать процесс передачи данных.
2. Если на принимающем хосте запрашиваемый отправителем порт является активным, то принимающий хост отвечает пакетом SYN-ACK, подтверждая установление контакта с запрашивающим хостом.
3. Запрашивающий хост возвращает пакет ACK, подтверждая тем самым, что хосты «слышат» друг друга и могут обмениваться информацией.

Итак, соединение установлено. Теперь данные могут передаваться по физическим каналам и маршрутизироваться между различными сетями, чтобы достичь адресата, — но каким образом конечная точка узнает, как обрабатывать полученную информацию?

Для этого на локальном и удаленном хостах создаются специальные файлы — сокеты. Сокет — это виртуальная (логическая) конечная точка, предназначенная для взаимодействия между процессами. Подробнее работа клиентской ОС Linux с сокетами будет рассмотрена ниже в *главе 2*.

TCP осуществляет мониторинг соединения с момента его установления и до тех пор, пока оно не будет разорвано. Состояние соединения зависит от согласия как отправителя, так и получателя начать процесс обмена данными. Далее состояние соединения отслеживает, кто передает и кто получает информацию в TCP-потоке. Для того чтобы понимать, когда и где происходит соединение, TCP оперирует сложной системой маркировки состояний, используя для этого 9-битовые флаги в заголовке TCP-сегмента, как иллюстрируется на рис. 1.7.

Состояниями TCP-соединения являются:

LISTEN (сервер)

Ожидание на запрос соединения от любого удаленного TCP или порта.

SYN-SENT (клиент)

Ожидание ответного запроса на соединение после посылки запроса на соединение.

SYN-RECEIVED (сервер)

Ожидание подтверждения на запрос соединения после посылки запроса на соединение и получение ответного запроса.

ESTABLISHED (как сервер, так и клиент)

Соединение открыто; полученные данные могут быть переданы пользователю — это промежуточное состояние соединения на этапе обмена данными.

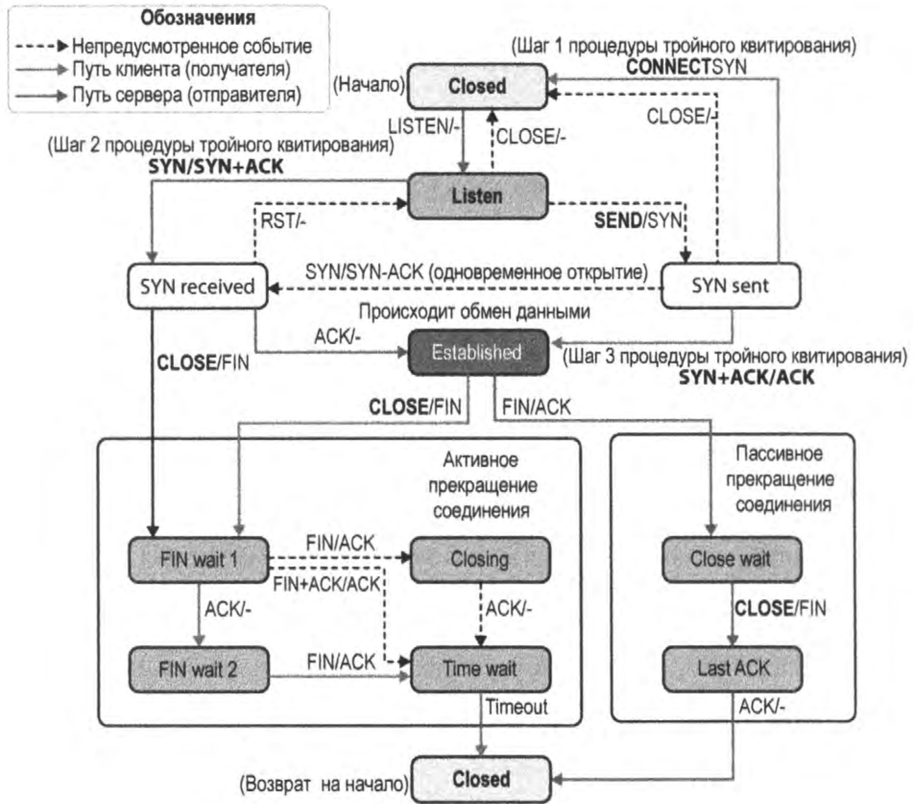


Рис. 1.7. Диаграмма состояний соединения в протоколе TCP

FIN-WAIT-1 (как сервер, так и клиент)

Ожидание запроса на прекращение соединения от удаленного хоста.

FIN-WAIT-2 (как сервер, так и клиент)

Ожидание запроса на прекращение соединения от удаленного TCP.

CLOSE-WAIT (как сервер, так и клиент)

Ожидание запроса на прекращение соединения от локального пользователя.

CLOSING (как сервер, так и клиент)

Ожидание запроса на прекращение соединения от удаленного TCP.

LAST-ACK (как сервер, так и клиент)

Ожидание подтверждения на запрос о прекращении соединения, ранее посланный удаленному хосту.

TIME-WAIT (как сервер, так и клиент)

Время ожидания, чтобы гарантировать, что удаленный хост получил подтверждение своего запроса на прекращение соединения.

CLOSED (как сервер, так и клиент)

Отсутствие какого-либо состояния соединения.

В примере 1.3 приводится таблица TCP-соединений (для Macintosh), их состояние и адреса конечных точек соединения.

Пример 1.3. Состояния соединения в модели TCP

```
o → netstat -ap TCP
Active internet connections  including servers
Proto Recv-Q Send-Q Local Address          Foreign Address        (state)
tcp6       0      0 2607:fcc8:a205:c.53606 g2600-1407-2800-.https  ESTABLISHED
tcp6       0      0 2607:fcc8:a205:c.53603 g2600-1408-5c00-.https  ESTABLISHED
tcp4       0      0 192.168.0.17.53602     ec2-3-22-64-157..https  ESTABLISHED
tcp6       0      0 2607:fcc8:a205:c.53600 g2600-1408-5c00-.https  ESTABLISHED
tcp4       0      0 192.168.0.17.53598     164.196.102.34.b.https  ESTABLISHED
tcp4       0      0 192.168.0.17.53597     server-99-84-217.https  ESTABLISHED
tcp4       0      0 192.168.0.17.53596     151.101.194.137.https   ESTABLISHED
tcp4       0      0 192.168.0.17.53587     ec2-52-27-83-248.https  ESTABLISHED
tcp6       0      0 2607:fcc8:a205:c.53586 iad23s61-in-x04..https  ESTABLISHED
tcp6       0      0 2607:fcc8:a205:c.53542 iad23s61-in-x04..https  ESTABLISHED
tcp4       0      0 192.168.0.17.53536     ec2-52-10-162-14.https  ESTABLISHED
tcp4       0      0 192.168.0.17.53530     server-99-84-178.https  ESTABLISHED
tcp4       0      0 192.168.0.17.53525     ec2-52-70-63-25..https  ESTABLISHED
tcp6       0      0 2607:fcc8:a205:c.53480 upload-lb.eqiad..https  ESTABLISHED
tcp6       0      0 2607:fcc8:a205:c.53477 text-lb.eqiad.wi.https  ESTABLISHED
tcp4       0      0 192.168.0.17.53466     151.101.1.132.https     ESTABLISHED
tcp4       0      0 192.168.0.17.53420     ec2-52-0-84-183..https  ESTABLISHED
tcp4       0      0 192.168.0.17.53410     192.168.0.18.8060       CLOSE_WAIT
tcp6       0      0 2607:fcc8:a205:c.53408 2600:1901:1:c36:.https  ESTABLISHED
tcp4       0      0 192.168.0.17.53067     ec2-52-40-198-7..https  ESTABLISHED
tcp4       0      0 192.168.0.17.53066     ec2-52-40-198-7..https  ESTABLISHED
tcp4       0      0 192.168.0.17.53055     ec2-54-186-46-24.https  ESTABLISHED
tcp4       0      0 localhost.16587        localhost.53029         ESTABLISHED
tcp4       0      0 localhost.53029        localhost.16587         ESTABLISHED
tcp46      0      0 *.16587                *.*                     LISTEN
tcp6      56      0 2607:fcc8:a205:c.56210 ord38s08-in-x0a..https  CLOSE_WAIT
tcp6       0      0 2607:fcc8:a205:c.51699 2606:4700::6810:.https  ESTABLISHED
tcp4       0      0 192.168.0.17.64407     do-77.lastpass.c.https  ESTABLISHED
tcp4       0      0 192.168.0.17.64396     ec2-54-70-97-159.https  ESTABLISHED
tcp4       0      0 192.168.0.17.60612     ac88393aca5853df.https  ESTABLISHED
tcp4       0      0 192.168.0.17.58193     47.224.186.35.bc.https  ESTABLISHED
tcp4       0      0 localhost.63342        *.*                     LISTEN
tcp4       0      0 localhost.6942         *.*                     LISTEN
tcp4       0      0 192.168.0.17.55273     ec2-50-16-251-20.https ESTABLISHED
```

Теперь, когда мы знаем, каким образом TCP устанавливает и отслеживает соединения, давайте рассмотрим HTTP-запрос на наш веб-сервер на транспортном уровне,

используя протокол TCP. Воспользуемся для этого системным вызовом `tcpdump` из командной строки.

tcpdump

> Утилита `tcpdump` распечатывает описание содержимого пакетов на сетевом интерфейсе, который совпадает со значением булевского выражения.
>
> *Страница руководства по* `tcpdump`

Утилита `tcpdump` предоставляет администраторам и пользователям заголовки всех пакетов, обработанных системой, и выдает их на экран. Выбор осуществляется на основе параметров, задаваемых в заголовке TCP-сегментов. В нашем запросе мы выбираем все пакеты с портом назначения 8080 на сетевом интерфейсе с меткой lo0 — это локальный петлевой интерфейс на Macintosh. Наш веб-сервер запущен на 0.0.0.0:8080. Рис. 1.8 показывает, на каких уровнях TCP/IP команда `tcpdump` осуществляет перехват пакета — это происходит между драйвером сетевой карты (NIC) и уровнем 2.

Рис. 1.8. Перехват пакета (packet capture) командой `tcpdump`

Петлевой интерфейс (loopback) — это логический виртуальный интерфейс устройства, не связанный с какими-то физическими устройствами, как, например, Ethernet. Петлевой интерфейс — всегда активен, доступен и работает, даже если все другие интерфейсы на хосте остаются неактивированными.

В общем случае формат выдачи `tcpdump` будет содержать следующие поля: `tos`, `TTL`, `id`, `offset`, `flags`, `proto`, `length` и `options`. Рассмотрим их подробнее:

`tos`

Тип поля сервиса.

`TTL`

Время жизни; не отображается, если установлен 0.

`id`

Поле IP-идентификатора.

`offset`

Поле смещения фрагмента; печатается, чтобы показать, является ли он частью фрагментированной датаграммы или нет.

`flags`

Флаг DF (не фрагментировать) указывает, что при передаче пакет нельзя разбивать на фрагменты. Если флаг не установлен, то пакет фрагментировать можно. Флаг MF (еще есть фрагменты) указывает, что идет фрагментированная передача; если не установлен, то указывает, все фрагменты переданы или же пакет передавался без фрагментирования.

`proto`

Поле, содержащее идентификатор протокола.

`length`

Поле, указывающее общую длину.

`options`

Опции протокола IP.

В системах, поддерживающих режим checksum offload (вычисление контрольной суммы передается другим объектам), а также в протоколах IP, TCP и UDP контрольные суммы вычисляются на сетевой карте перед тем, как начать пересылку данных по физическому каналу. Поскольку перехват пакета производится командой `tcpdump` перед сетевой картой, то в выдаче, подобно приведенной в примере 1.4, могут появиться ошибки вроде `cksum 0xfe34 (incorrect ->0xb4c1)`.

Чтобы получить выдачу, как в примере 1.4, откройте другой терминал и запустите `tcpdump` на петлевом интерфейсе только для TCP и порта 8080 — в противном случае вы увидите в результате множество других пакетов, не относящихся непосредственно к примеру. Чтобы отслеживать пакеты, вам потребуется повысить свой приоритет, так что используйте `sudo`.

Пример 1.4. Системный вызов `tcpdump`

```
o → sudo tcpdump -i lo0 tcp port 8080 -vvv  1

tcpdump: listening on lo0, link-type NULL (BSD loopback),
capture size 262144 bytes 2

08:13:55.009899 localhost.50399 > localhost.http-alt: Flags [S],cksum 0x0034
(incorrect -> 0x1bd9), seq 2784345138, win 65535, options [mss 16324,nop,wscale
6,nop,nop,TS val 587364215 ecr 0,sackOK,eol], length 0 3

08:13:55.009997 localhost.http-alt > localhost.50399: Flags [S.],
cksum 0x0034 (incorrect -> 0xbe5a), seq 195606347,
ack 2784345139, win 65535, options [mss 16324,nop,wscale 6,nop,nop,TS val 587364215
ecr 587364215,sackOK,eol], length 0 4

08:13:55.010012 localhost.50399 > localhost.http-alt: Flags [.],
cksum 0x0028 (incorrect -> 0x1f58), seq 1, ack 1,
win 6371, options [nop,nop,TS val 587364215 ecr 587364215],
length 0 5
```

```
08:13:55.010021 localhost.http-alt > localhost.50399: Flags [.],
cksum 0x0028 (incorrect -> 0x1f58), seq 1, ack
1, win 6371, options [nop,nop,TS val 587364215 ecr 587364215],
length 0 6

08:13:55.010079 localhost.50399 > localhost.http-alt: Flags [P.],
cksum 0x0076 (incorrect -> 0x78b2), seq 1:79,
ack 1, win 6371, options [nop,nop,TS val 587364215 ecr 587364215],
length 78: HTTP, length: 78 7

GET / HTTP/1.1
Host: localhost:8080
User-Agent: curl/7.64.1
Accept: */*
08:13:55.010102 localhost.http-alt > localhost.50399: Flags [.],
cksum 0x0028 (incorrect -> 0x1f0b), seq 1,
ack 79, win 6370, options [nop,nop,TS val 587364215 ecr 587364215],
length 0 8

08:13:55.010198 localhost.http-alt > localhost.50399: Flags [P.],
cksum 0x00a1 (incorrect -> 0x05d7), seq 1:122,
ack 79, win 6370, options [nop,nop,TS val 587364215 ecr 587364215],
length 121: HTTP, length: 121 9

HTTP/1.1 200 OK
Date: Wed, 19 Aug 2020 12:13:55 GMT
Content-Length: 5
Content-Type: text/plain; charset=utf-8
Hello[!http]

08:13:55.010219 localhost.50399 > localhost.http-alt: Flags [.], cksum 0x0028
(incorrect -> 0x1e93), seq 79,
ack 122, win 6369, options [nop,nop,TS val 587364215 ecr 587364215], length 0 10

08:13:55.010324 localhost.50399 > localhost.http-alt: Flags [F.],
cksum 0x0028 (incorrect -> 0x1e92), seq 79,
ack 122, win 6369, options [nop,nop,TS val 587364215 ecr 587364215],
length 0 11

08:13:55.010343 localhost.http-alt > localhost.50399: Flags [.],
cksum 0x0028 (incorrect -> 0x1e91), seq 122,
\ack 80, win 6370, options [nop,nop,TS val 587364215 ecr 587364215],
length 0 12

08:13:55.010379 localhost.http-alt > localhost.50399: Flags [F.],
cksum 0x0028 (incorrect -> 0x1e90), seq 122,
ack 80, win 6370, options [nop,nop,TS val 587364215 ecr 587364215],
length 0 13
```

```
08:13:55.010403 localhost.50399 > localhost.http-alt: Flags [.],
cksum 0x0028 (incorrect -> 0x1e91), seq 80, ack
123, win 6369, options [nop,nop,TS val 587364215 ecr 587364215],
length 0 14

12 packets captured, 12062 packets received by filter
0 packets dropped by kernel. 15
```

1. Запуск tcpdump с опциями. Программа sudo делегирует требуемый высокий приоритет. tcpdump — выполняемый файл. -i lo0 — интерфейс, с которого мы собираемся перехватывать пакеты. dst port 8080 — маска фильтра (описана в инструкции к команде), в нашем случае мы отбираем все пакеты с назначениями на порт 8080 — это порт, запросы с которого отслеживает веб-сервер. -v — опция текстовой выдачи, позволяющая увидеть подробности работы команды tcpdump.
2. Ответ от tcpdump, информирующий о том, что фильтр активирован.
3. Это первый пакет, полученный в рамках процедуры тройного квитирования. Мы можем утверждать, что это SYN. Флаговый бит установлен в [S.], порядковый номер задан cURL как 2784345138, а номер процесса на локальном хосте есть 50399.
4. Пакет SYN-ACK перехвачен командой tcpdump от процесса localhost.http -alt, это Go веб-сервер. Значение флага есть [S.], т. е. это SYN-ACK. Пакет посылает 195606347 в качестве следующего порядкового номера, устанавливается ACK 2784345139 для подтверждения предыдущего пакета.
5. Пакет подтверждения с клиента cURL посылается обратно на сервер с флагом ACK, установленным в [.], и порядковыми номерами ACK и SYN, установленными в 1, что означает, что клиент готов к пересылке данных.
6. Номер подтверждения установлен в 1, чтобы сигнализировать получение флага SYN от клиента; открывается data push («проталкивание данных»).
7. TCP-соединение установлено: клиент и сервер готовы к обмену данными. Следующие пакеты — это данные HTTP-запроса с установленным флагом data push, [.P]. Предыдущие пакеты имели нулевую длину, но HTTP-запрос имеет длину 78 байт и порядковый номер 1:79.
8. Флаг ACK установлен в [.]: сервер подтверждает получение данных, посылая номер подтверждения 79.
9. Этот пакет является ответом HTTP-сервера на запрос клиента cURL. Флаг data push установлен ([.P]), он подтверждает предыдущий пакет с номером ACK, равным 79. В процессе передачи данных устанавливается новый порядковый номер 122, данные длиной 121 байт.
10. Клиент cURL подтверждает получение пакета, устанавливая флаг ACK, устанавливает номер подтверждения в 122 и порядковый номер в 79.
11. Начинается процедура закрытия TCP-соединения. Клиент посылает пакет FIN-ACK, [F.], подтверждая получение предыдущего пакета с порядковым номером 122 и устанавливая новый порядковый номер, равный 80.

12 Сервер задает номер подтверждения, равный 80, и устанавливает флаг ACK.

13 TCP требует, чтобы для прекращения соединения как клиент, так и сервер исполняли пакет FIN — это пакет, в котором устанавливаются флаги FIN и ACK.

14 Это финальное подтверждение ACK от клиента, номер подтверждения равен 123. Теперь соединение прекращено.

15 По окончании работы команда `tcpdump` сообщает число пакетов, перехваченных в данном цикле, общее число пакетов, перехваченных `tcpdump`, и сколько пакетов были сброшены операционной системой.

Системный вызов `tcpdump` представляет собой удобный инструмент для поиска и устранения проблем как для инженеров, так и для администраторов сети. Вообще очень полезно уметь контролировать сетевые соединения на разных уровнях. С другими важными свойствами `tcpdump` мы познакомимся в *главе 6*.

Рассмотренный пример представлял собой простое HTTP-приложение с использованием TCP. Все данные пересылались по сети в обычном текстовом виде. Этого достаточно для примеров типа «Hello World!», но пересылка банковских паролей уже требует некоего уровня безопасности. Транспортный уровень сам по себе не обеспечивает защиту для пересылаемых по сети данных. Дополнение TCP протоколом TLS позволяет осуществлять защищенную передачу данных. Рассмотрим это в следующем разделе.

TLS

Протокол TLS добавляет к TCP функцию шифрования. Протокол TLS является добавочным к TCP/IP и не рассматривается как составная часть TCP. HTTP-транзакции могут осуществляться и без TLS, но не будут при этом защищены от перехвата злоумышленниками. TLS представляет собой комбинацию протоколов, используемых для контроля трафика между отправителем и получателем. Подобно TCP, TLS применяет квитирование, чтобы осуществить шифрование и обменяться ключами. Следующие шаги описывают процедуру TLS-квитирования между клиентом и сервером, которая представлена также на рис. 1.9.

1. `ClientHello`: сообщение содержит наборы шифрования (cipher suite), поддерживаемые клиентом, а также случайное число.
2. `ServerHello`: Сообщение содержит поддерживаемый шифр и случайное число.
3. `ServerCertificate`: Сообщение содержит сертификат сервера и его открытый ключ шифрования.
4. `ServerHelloDone`: Конец сообщения `ServerHello`. Если клиент получает запрос на свой сертификат, он отсылает сообщение `ClientCertificate`.
5. `ClientKeyExchange`: пользуясь случайным числом, переданным сервером, клиент генерирует предварительное случайное секретное число (pre-master secret), шифрует его с помощью открытого ключа сервера и отсылает серверу.
6. `Key Generation`: клиент и сервер генерируют общее секретное число (master secret), используя присланное ранее случайное секретное число, и обмениваются соответствующими случайными числами.

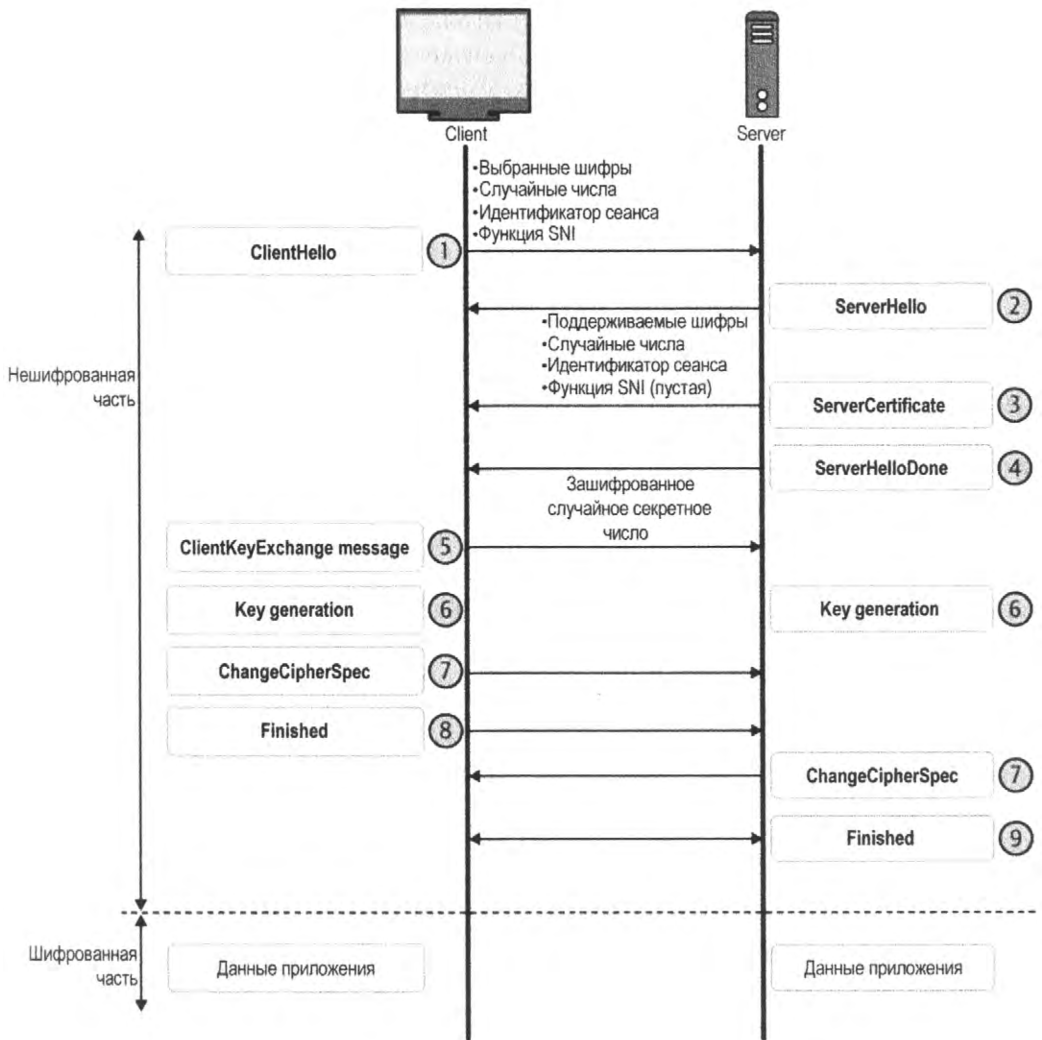


Рис. 1.9. Процедура квитирования в протоколе TLS

7. `ChangeCipherSpec`: клиент и сервер обмениваются сообщениями `ChangeCipherSpec`, чтобы начать использовать новые ключи шифрования.

8. `Finished Client`: Клиент посылает завершающее сообщение, чтобы подтвердить, что обмен ключами и аутентификация проведены успешно.

9. `Finished Server`: Сервер посылает завершающее сообщение клиенту, чтобы закончить квитирование.

Приложения и компоненты Kubernetes берут на себя работу с TLS. Тем не менее полезно познакомиться с некоторыми базовыми понятиями этого протокола. *Глава 5* дает более подробную информацию о TLS и Kubernetes.

Как мы уже показали в примере с нашим веб-сервером, клиентом cURL и системным вызовом `tcpdump`, TCP является гибким и надежным протоколом для обмена

данными между различными хостами. Благодаря использованию флагов и подтверждений этот протокол обеспечивает надежность для тысяч сообщений, посылаемых по незащищенным сетям по всему земному шару. Понятно, что надежность имеет свою цену — из 12 посланных пакетов только 2 были заняты непосредственной пересылкой данных. Для приложений, которые не требуют особой надежности, например для передачи голосовых сообщений, протокол UDP может оказаться более выгодным в смысле генерации дополнительного трафика. Теперь, когда мы разобрали, каким образом TCP обеспечивает надежность коммуникации, посмотрим, чем протокол UDP отличается от TCP.

UDP

Протокол UDP может использоваться для приложений, которым не требуется надежность, обеспечиваемая TCP. UDP выгодно применять в приложениях, в которых потеря некоторого количества пакетов не является критичной, — это, например, голосовые коммуникации или DNS. С точки зрения сетевого трафика UDP генерирует меньшую нагрузку, в нем всего четыре поля и отсутствуют подтверждения, в отличие от его «болтливого» собрата TCP.

Данный протокол является ориентированным на транзакции, он подходит для протоколов простых запросов и ответов, как, например, система доменных имен DNS или протокол управлениями простыми сетями SNMP. UDP разбивает запрос на отдельные датаграммы, тем самым приспосабливая его для использования совместно с другими протоколами для передачи через сети типа VPN (виртуальная частная сеть). При этом генерируется небольшой трафик, что особенно выгодно, если приложение работает с протоколом динамической настройки узла DHCP. Отсутствие проверки состояний при обмене данными делает UDP идеальным для приложений типа голосовой коммуникации, которые устойчивы к возможной потере пакетов. То, что UDP не занимается повторной трансляцией данных, обеспечивает его выгоду при применении в видеостримингах.

Рассмотрим немногочисленные заголовки UDP-датаграммы (рис. 1.10):

Source port number (2 байта)

Идентифицирует порт источника. Хост источника является клиентом, номер порта — эфемерный. UDP-порты имеют хорошо известные значения как, например, DNS на 53 или DHCP на 67/68.

Destination port number (2 байта)

Идентифицирует порт назначения, является обязательным для задания.

Length (2 байта)

Задает длину заголовка UDP и UDP-данные в байтах. Минимальная длина 8 байтов — длина заголовка.

Checksum (2 байта)

Используется для проверки на ошибки в заголовке и данных. Является опциональной в версии IPv4 и обязательной в IPv6; по умолчанию значение «ноль», если не используется.

Рис. 1.10. Заголовок датаграммы протокола UDP

UDP и TCP — это стандартные транспортные протоколы, помогающие передавать данные между хостами. Kubernetes поддерживает оба протокола, и его сервисы дают возможность пользователям распределять нагрузку на многих подах. Также важно отметить, что в каждом сервисе разработчик должен явно указать транспортный протокол — если это не сделано, то по умолчанию используется TCP.

Следующий уровень в иерархии TCP/IP — это уровень межсетевого взаимодействия, или уровень Интернета. Это пакеты, которые могут посылаться во все точки земного шара через огромное множество сетей, составляющее современный Интернет. Рассмотрим, каким образом это происходит.

Уровень межсетевого взаимодействия

В модели TCP/IP все данные протоколов TCP и UDP пересылаются как IP-пакеты на уровне межсетевого взаимодействия (сети Интернет). Этот уровень отвечает за передачу данных между отдельными сетями. Исходящие пакеты выбирают ближайший хост и посылают на него данные одновременно с посылкой параметров, необходимых для уровня канала данных; хост получает пакеты, производит распаковку и отправляет их соответствующему протоколу транспортного уровня. В версии IPv4 протокол IP производит фрагментацию и сборку пакетов на основе MTU — максимального размера IP-пакета.

Протокол IP не гарантирует безошибочную доставку пакета. Поскольку прохождение пакета через различные сети является весьма ненадежной и неотказоустойчивой операцией, то проверка целостности пакета ложится на конечные точки. Как было показано в предыдущем разделе, обеспечение надежности приема-передачи — это функция транспортного уровня. У каждого пакета есть контрольная сумма, которая позволяет проверить, нет ли ошибок в полученной пакетной информации. Однако этот уровень не контролирует целостность полных данных. Пакеты в сети идентифицируются через IP-адреса отправителя и получателя, как будет рассмотрено в следующем разделе.

Протокол Интернета

Этот всемогущий протокол определяется в документе RFC 791 и используется для передачи данных между сетями. На рис. 1.11 показан формат заголовка в версии протокола IPv4.

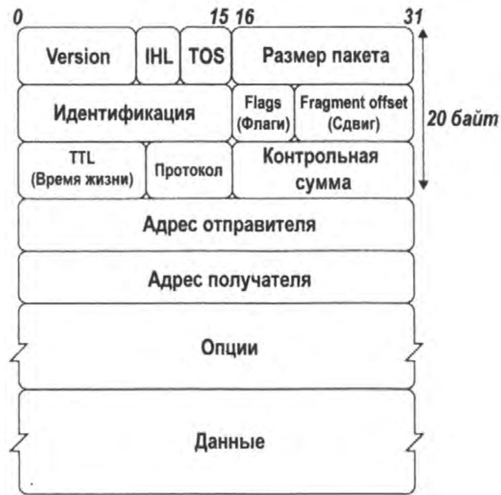


Рис. 1.11. Формат заголовка в протоколе IPv4

Рассмотрим его более подробно:

Версия (Version)

Первое поле заголовка IP-пакета есть четырехбитовое поле версии. Для протокола IPv4 оно всегда равно 4.

Длина интернет-заголовка (Internet header length, IHL)

Заголовок в IPv4 имеет переменную длину благодаря наличию опционального 14-го поля.

Тип сервиса (Type of Service, TOS)

Первоначально определяемое как тип сервиса (TOS), сейчас как кодовая точка дифференцированных сервисов (DSCP), это поле задает дифференцированные услуги. DSCP позволяет сетям и маршрутизаторам задавать приоритетность передачи пакетов при перегрузке сети. Технологии передачи голоса по Интернету используют DSCP, чтобы установить приоритет передачи звонков над остальными видами трафика.

Размер пакета (Total length)

Задает полный размер пакета в байтах.

Идентификация (Identification)

Поле идентификации, используется для однозначной идентификации группы фрагментов данной IP-датаграммы.

Флаги (Flags)

Используются для управления фрагментами и их идентификации. В порядке от старшего бита к младшему:

- Бит 0: зарезервирован, устанавливается в 0.
- Бит1: не фрагментировать.
- Бит 2: еще есть фрагменты.

Сдвиг (Fragment offset)

Задает офсет (сдвиг) определенного фрагмента относительно первого нефрагментированного IP-пакета. Первый пакет всегда имеет сдвиг 0.

Время жизни (Time to Live — TTL)

8-битовый промежуток, задающий время жизни датаграммы, предназначен для защиты от зацикливания датаграммы в сети.

Protocol

Используется в секции данных IP-пакета. Ассоциация IANA предоставляет список номеров IP-протоколов в документе RFC 790. Некоторые хорошо известные протоколы приводятся в табл. 1.4.

Таблица 1.4. *Номера IP-протоколов*

Номер	Название	Аббревиатура
1	Протокол управления сообщениями в Интернете	ICMP
2	Протокол управления интернет-группами	IGMP
6	Протокол управления передачей данных	TCP
17	Протокол передачи датаграмм	UDP
41	Инкапсуляция в IPv4	ENCAP
89	Открыть кратчайший путь первым	OSPF
132	Протокол управления передачей потоков	SCTP

Контрольная сумма заголовка (Header Checksum)

Поле контрольной суммы заголовка в IPv4 используется для контроля ошибок. При прибытии пакета роутер вычисляет контрольную сумму заголовка, если оба числа не совпадают, то роутер сбрасывает пакет. Инкапсулированный протокол обрабатывает ошибки в поле данных. Оба протокола UDP и TCP имеют поля контрольной суммы.

Когда роутер получает пакет, он уменьшает поле TTL на единицу. Как следствие — роутер должен вычислить новую контрольную сумму.

Адрес отправителя (Source address)

Адрес отправителя пакета в стандарте IPv4.

При прохождении по сети адрес источника может быть изменен устройством трансляции сетевого адреса NAT; эти устройства будут рассмотрены ниже в данном разделе и позже в *главе 3*.

Адрес получателя (Destination address)

Адрес получателя пакета в IPv4. Аналогично адресу источника данный адрес может быть изменен NAT-устройством.

Опции (Options)

Возможными опциями заголовка являются Copied (скопировано), Option Class (класс опции), Option number (номер опции), Option Length (опциональная длина) и Option data (опциональные данные)

Самым важным компонентом в протоколе является адрес, именно по нему осуществляется идентификация сетей. Адрес одновременно идентифицирует хост в сети и саму сеть (более подробно рассмотрим ниже, в разделе «Движение по сети»). Компьютерные специалисты должны четко понимать структуру IP-адреса. Сначала мы рассмотрим, как происходит формирование адреса в версии IPv4, а затем познакомимся с изменениями, — весьма значительными, — предпринятыми в версии IPv6.

В IPv4 адреса задаются с помощью десятичных цифр и точек — для человекочитаемости, компьютер же воспринимает их как двоичные строки. На рис. 1.12 показано представление адреса как в десятичном формате с точками, так и в виде двоичных строк. Каждая часть имеет длину в 8 битов, всего 4 части, полная длина получается 32 бита. IPv4-адрес делится на 2 части: первая указывает на сеть, а вторая служит для идентификации компьютера в данной сети.

Адрес в десятичной форме с точками-разделителями.

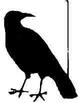
Рис. 1.12. Адрес в версии протокола IPv4

В примере 1.5 показана выдача IP-адреса компьютера на его сетевую карту, этот адрес 192.168.1.2. IP-адрес также имеет связанную с ним маску подсети (netmask), чтобы понимать, к какой сети он принадлежит. В примере маска подсети есть `netmask 0xffffff00`, или в десятичном формате: 255.255.255.0.

Пример 1.5. IP-адрес

```
o → ifconfig en0
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
     options=400<CHANNEL_IO>
     ether 38:f9:d3:bc:8a:51
     inet6 fe80::8f4:bb53:e500:9557%en0 prefixlen 64 secured scopeid 0x6
     inet 192.168.1.2 netmask 0xffffff00 broadcast 192.168.1.255
     nd6 options=201<PERFORMNUD,DAD>
     media: autoselect
     status: active
```

Использование подсети вызывает в памяти классовую адресацию. Первоначально, IP-адрес рассматривался как комбинация 8-, 16- или 24-битового префикса сети и 24-, 16- или 8-битового идентификатора хоста. Класс A использовал 8 битов для хоста, класс B — 16 и класс C — 24. Соответственно, в классе A были доступны 2^{16} хостов — 16 777 216; в классе B — 65 536 и в классе C — 256. Каждый класс определял адреса хостов, включая сюда и широковещательные адреса. Классовая адресация иллюстрируется на рис. 1.13.



Существует еще 2 класса, но, как правило, они не применяются в IP-адресации. Адреса класса D используются при многоадресной передаче (multicasting), а адреса класса E зарезервированы для экспериментальных целей.

Сетевая часть и хостовая часть

Класс и маска подсети

	Octer 1	Octer 2	Octer 3	Octer 4	Маска подсети
Class A	Network	Host	Host	Host	255.0.0.0 or /3
Class B	Network	Network	Host	Host	255.255.0.0 or /16
Class C	Network	Network	Network	Host	255.255.255.0 or /124

Рис. 1.13. IP-класс

С развитием Интернета стало приходить понимание того, что классовая адресация довольно негибкая и неэффективная. Для решения проблемы пришлось выйти за границы классов и перейти к бесклассовой междоменной маршрутизации CIDR. Вместо 16 миллионов адресов внутри одного класса, интернет-объект теперь пользуется только частью этого пространства. Механизм CIDR позволяет сетевым разработчикам при необходимости сдвигать границы подсети внутри адресного пространства класса, обеспечивая тем самым необходимую гибкость IP-адресации.

На рис. 1.14 показана структура IPv4-адреса 208.130.29.33 и заложенная в нем иерархия. Организация IANA предоставила диапазон CIDR 208.128.0.0/11 Американскому реестру интернет-адресов ARIN. ARIN, в свою очередь, разбивает свою сеть на все более мелкие подсети, что приводит нас в конце концов к конкретному хосту в сети 208.130.29.33/32.

Организация IANA осуществляет глобальную координацию доменных имен (DNS), IP-адресацию и распределение ресурсов Интернета.

Однако со временем даже то расширение адресного пространства, что было достигнуто при помощи механизма CIDR, стало недостаточным, что привело инженеров к разработке стандарта IPv6.

Из рис. 1.15 видно, что IPv6, в отличие от IPv4, использует шестнадцатеричную адресацию. При этом, как это было и в IPv4, адрес состоит из сетевого префикса и адреса хоста.

Рис. 1.14. Пример бесклассовой междоменной маршрутизации CIDR

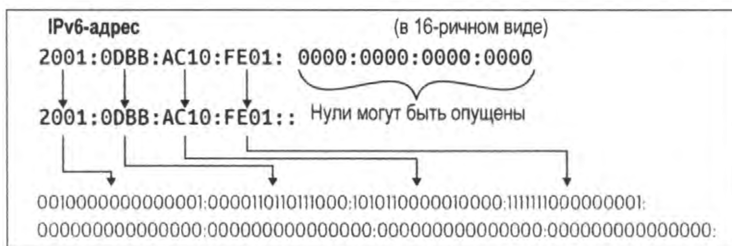


Рис. 1.15. Адреса в версии протокола IPv6

Самым значительным отличием IPv6 от IPv4 является размер адресного пространства: IPv4 использует 32-битовую адресацию, а IPv6 — 128-битовую. Для наглядности приведем соответствующие числа в явном виде:

Объем адресного пространства в IPv4: 4 294 967 296 адреса.

Объем в IPv6: 340 282 366 920 938 463 463 374 607 431 768 211 456 адреса.

Теперь, когда мы разобрали, каким образом идентифицируется каждый конкретный хост и сеть, к которой он принадлежит, рассмотрим, как сети обмениваются информацией между собой, используя для этого протоколы маршрутизации.

Движение по сети

Пакеты снабжены адресом, и данные подготовлены к отправке. Но каким образом наши пакеты попадут с нашего хоста в нашей сети к хосту-адресату в другой сети, находящейся на другом конце света? Это и есть задача маршрутизации. Существует несколько протоколов маршрутизации, но в Интернете используется протокол граничного шлюза BGP — протокол динамической маршрутизации, используемый для управления движением пакетов между конечными точками в Интернете. Этот протокол нам понадобится, поскольку некоторые сетевые реализации Kubernetes используют BGP для маршрутизации трафика между узлами. Между каждым узлом в отдельной сети стоит последовательность маршрутизаторов.

В рамках протокола BGP каждой сети в Интернете приписывается номер автономной системы ASN, чтобы обозначить определенную административную или корпо-

ративную организационную единицу, поддерживающую общую и четко определенную политику маршрутизации в Интернете. Протокол BGP и номера ASN позволяют сетевым администраторам сохранять контроль над своими внутренними сетями, одновременно обеспечивая трафик по всему Интернету. В табл. 1.5 перечисляются доступные номера ASN, назначаемые организацией IANA и ее региональными отделениями[1].

Таблица 1.5. *Доступные номера ASN*

Номер	Размер (в битах)	Значение	Спецификация
0	16	Зарезервировано	RFC1930, RFC 7607
1-23455	16	Публичные ASN	
233456	16	Зарезервировано	RFC6793
23457-64495	16	Публичные ASN	
64496-64511	16	Зарезервировано	RFC65398
64512-65534	16	Зарезервировано	RFC1930, RFC6996
65535	16	Зарезервировано	RFC7300
65536-65551	32	Зарезервировано	RFC4893, RFC5398
65552-131071	32	Зарезервировано	
131072-4199999999	32	Публичные 32-битовые ASN	
42000000000-4294967294	32	Зарезервировано	RFC6996
4294967295	32	Зарезервировано	RFC7300

На рис. 1.16 показаны 5 номеров ASN, 100-500. Хост с адресом 130.10.1.200 пытается связаться с хостом по адресу 150.10.2.300. Как только локальный роутер или шлюз по умолчанию для хоста 130.10.1.200 получает пакет, он начинает искать интерфейс или путь к 150.10.2.300, который протокол BGP задал для маршрута.

На основе таблицы маршрутизации, приведенной на рис. 1.17, роутер на AS 100 определил, что пакет предназначен для AS300 и оптимальный путь — это внешний интерфейс 140.10.1.1. Теперь он будет стучаться в шлюз AS200, пока локальный роутер 150.10.2.300 на AS300 не получит пакет. Возникающий трафик представлен на рис. 1.6, который иллюстрирует поток данных в модели TCP/IP. Желательно иметь хотя бы базовое понимание BGP, т. к. некоторые сетевые проекты с контейнерами, как, например, Calico, используют этот протокол для маршрутизации между узлами (подробнее будет рассмотрено в *главе 3*).

На рис. 1.17 представлена таблица локальной маршрутизации. Видно, что интерфейс, куда должен быть отослан пакет, определяется IP-адресом точки назначения.

[1] Номера автономных систем (AS), IANA.org.2018-12-07.

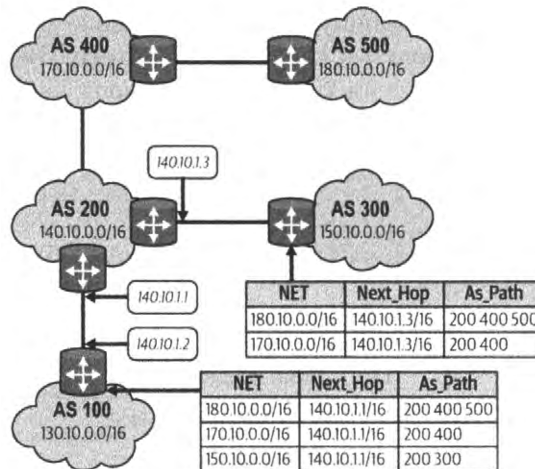


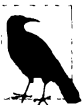
Рис. 1.16. Пример маршрута в протоколе BGP

```
o → netstat -nr
Routing tables

Internet:
Destination        Gateway            Flags        Netif Expire
default            192.168.1.254      UGSc           en8
127                127.0.0.1          UCS            lo0
127.0.0.1          127.0.0.1          UH             lo0
169.254            link#11            UCS            en8      !
192.168.1          link#11            UCS            en8      !
192.168.1.153/32   link#11            UCS            en8      !
192.168.1.254/32   link#11            UCS            en8      !
192.168.1.254      10:93:97:6e:6b:60  UHLWIir        en8   1186
224.0.0/4          link#11            UmCS           en8      !
224.0.0.251        1:0:5e:0:0:fb      UHmLWI         en8
239.255.255.250    1:0:5e:7f:ff:fa    UHmLWI         en8
255.255.255.255/32 link#11            UCS            en8      !
```

Рис. 1.17. Таблица локальной маршрутизации

Например, пакет с назначением в 192.168.1.153 будет послан через шлюз link#11, который является локальным по отношению к сети, так что маршрутизация не требуется. Адрес 192.168.1.254 является роутером сети, включающей наше интернет-соединение. Если сеть назначения неизвестна, то отсылка использует маршрут по умолчанию.



В ОС Linux и BSD подробную информацию можно получить в инструкции к команде net stat (`man netstat`). В ОС Apple команда `netstat` заимствована из версии BSD. См. Руководство по FreeBSD.

Роутеры в Интернете постоянно связываются друг с другом, обмениваясь данными о маршрутах и информацией об изменениях в их собственных сетях. Большой объем этого обмена данными проходит под управлением BGP, но сетевые инжене-

ры и администраторы могут также использовать для проверки соединения между хостами и роутерами протокол ICMP и команду ping.

ICMP

Команда ping — это сетевая утилита, использующая протокол ICMP для тестирования соединения между хостами сети. В примере 1.6 приводится листинг положительного пингования — теста на соединение с 192.168.1.2, когда все 5 пакетов возвращают эхо-ответ.

Пример 1.6. Эхо-запрос в протоколе ICMP

```
o → ping 192.168.1.2 -c 5
PING 192.168.1.2 (192.168.1.2): 56 data bytes
64 bytes from 192.168.1.2: icmp_seq=0 ttl=64 time=0.052 ms
64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.089 ms
64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.142 ms
64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=0.050 ms
64 bytes from 192.168.1.2: icmp_seq=4 ttl=64 time=0.050 ms
--- 192.168.1.2 ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.050/0.077/0.142/0.036 ms
```

Рис. 1.7, наоборот, показывает неудачную попытку команды ping связаться с хостом 1.2.3.4. Администраторы сети используют эту команду для проверки соединения, но она также полезна и для проверки соединений в контейнерных приложениях. Мы рассмотрим это в *главах 2* и *3*, где развернем наш минимальный Go веб-сервер в контейнер и модуль.

Пример 1.7. Неудачная попытка получить эхо-ответ на запрос ICMP

```
o → ping 1.2.3.4 -c 4
PING 1.2.3.4 (1.2.3.4): 56 data bytes
Request timeout for icmp_seq 0
Request timeout for icmp_seq 1
Request timeout for icmp_seq 2
--- 1.2.3.4 ping statistics ---
4 packets transmitted, 0 packets received, 100.0% packet loss
```

Как и в протоколах TCP и UDP, в ICMP-пакетах присутствуют заголовки, данные и опции, они рассматриваются ниже и показаны также на рис. 1.18.

Type

Тип ICMP.

Code

Подтип ICMP.

Checksum

Контрольная сумма для проверки на ошибки, рассчитывается на основе ICMP-заголовка и данных, которые могут быть нулевыми.

Rest of Header (4 байта)

Содержание варьируется в зависимости от значений полей `Type` и `Code`.

Data

Сообщения об ошибках в ICMP содержат секцию данных, которая включает в себя полную копию IPv4-заголовка.

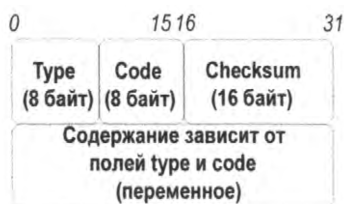
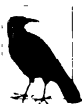


Рис. 1.18. Заголовок пакета в протоколе ICMP



Некоторые рассматривают ICMP как отдельный протокол транспортного уровня, поскольку он не использует TCP или UDP. В спецификации RFC 792 протокол ICMP определяется как протокол, обеспечивающий маршрутизацию, диагностику и обработку ошибок для протокола IP. Хотя ICMP-сообщения инкапсулированы в IP-датаграммы, обработка по протоколу ICMP рассматривается и реализуется как составляющая часть уровня Интернета. ICMP — это IP-протокол 1, в то время как TCP — 6, а UDP — 17.

Контрольные сообщения определяются значением поля `Type`. С помощью поля `code` задается дополнительная информация для сообщения. В табл. 1.6 приводятся некоторые значения поля `Type` для ICMP.

Таблица 1.6. *Значения поля* `Type` *в заголовке ICMP-пакета*

Значение	Расшифровка	Спецификация
0	Эхо-ответ	RFC 792
3	Адрес назначения недоступен	RFC 792
4	Перенаправлен	RFC 792
8	Эхо	RFC 792

Теперь, когда наши пакеты знают, откуда и куда они должны идти, пришло время физической передачи данных по сети — это обеспечивается функциями уровня канала данных.

Уровень канала данных

HTTP-запрос разбит на сегменты, которые снабжены адресами. Для них проложен маршрут их путешествия по Интернету, и теперь нам остается только переслать данные по физическим носителям.

Уровень канала данных модели TCP/IP подразделяется на два подуровня:

- Управление доступом к среде передачи данных (MAC).
- Управление логическим каналом (LLC).

Вместе они обеспечивают функциональность 1 и 2 OSI-уровней: уровня канала данных и физического уровня.

Уровень канала данных отвечает за соединения с локальной сетью. Первый подуровень, MAC, обеспечивает доступ к физическому носителю.

Подуровень LLC управляет потоком и мультиплексированием при отправке и демультиплексированием — при получении пакета, это показано на рис. 1.19.

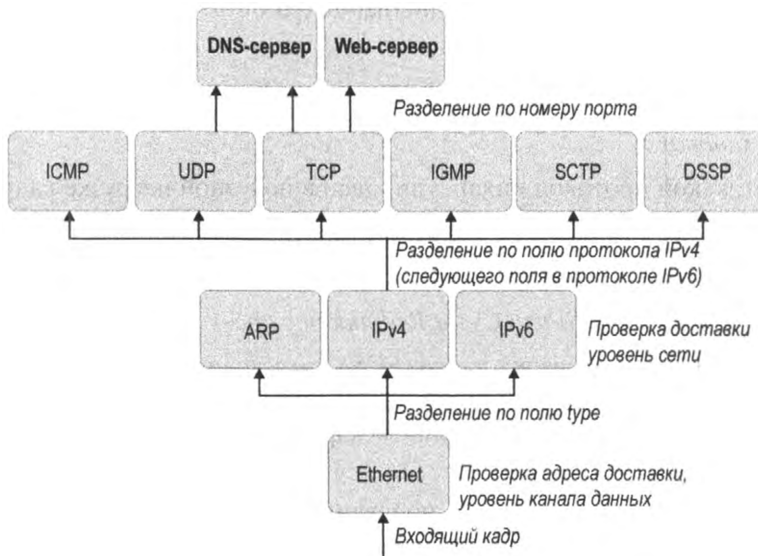


Рис. 1.19. Пример демультиплексирования в Ethernet

Стандарт IEEE 802.3 Ethernet определяет протоколы приема и передачи кадровов для инкапсуляции IP-пакетов. Стандарт IEEE 802 является обобщающим для LLC (802.2), беспроводной коммуникации (802.11) и Ethernet/MAC (802.3).

Как и другие пакеты протоколов, пакет Ethernet имеет заголовок и данные (рис. 1.20).

Рассмотрим все это подробнее:

Preamble (8 байтов)

Строка из нулей и единиц сообщает принимающему хосту, что кадр находится на входе.

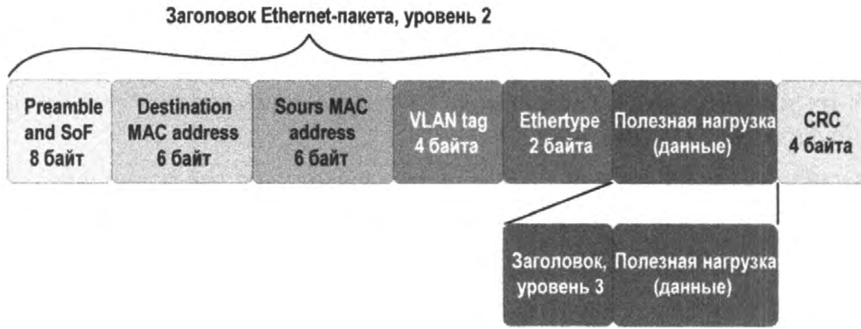


Рис. 1.20. Структура Ethernet-пакета

Destination MAC-address (6 байтов)

МАС-адрес назначения, получатель Ethernet-кадра.

Source MAC address (6 байтов)

МАС-адрес источника, источник Ethernet-кадра.

VLAN tag (4 байта)

Опциональный тег 802.1Q для маркировки трафика на сетевых сегментах.

Ether-type (2 байта)

Указывает, какой протокол инкапсулирован в полезной нагрузке кадра.

Payload (переменная длина)

Инкапсулированный IP-пакет.

Frame Check Sequence (FCS) или Cycle Redundancy check (CRC) (4 байта)

Последовательность проверки кадра (FCS) представляет собой четырехоктетную циклическую проверку избыточности (CRC), которая позволяет обнаруживать ошибки передачи в кадре, полученном адресатом. CRC является частью блока данных кадра Ethernet.

Из рис. 1.21 видно, что МАС-адреса присваиваются сетевым картам (NIC) при их изготовлении. МАС-адреса подразделяются на две части: уникальный идентификатор организации (OUI) и часть, уникальная для каждой сетевой карты.

Кадр сообщает получателю тип пакета. В табл. 1.7 приведены обычно используемые стандартные протоколы. В рамках Kubernetes мы в основном имеем дело с IPv4- и ARP-пакетами. Версия IPv6 недавно появилась в Kubernetes 1.19.

Таблица 1.7. *Стандартные EtherType-протоколы*

EtherType	Протокол
0x0800	Протокол Интернета версия 4 (IPv4)
0x0806	Протокол разрешения адреса (ARP)
0x8035	Протокол реверсивного разрешения адреса (RARP)

***Таблица 1.7** (окончание)*

EtherType	Протокол
0x86DD	Протокол Интернета версия 6 (IPv6)
0x88E5	Защита MAC (EEE802 14E)
0x9100	Кадр с VLAN-тегом (EEE802.1Q), двойной тег

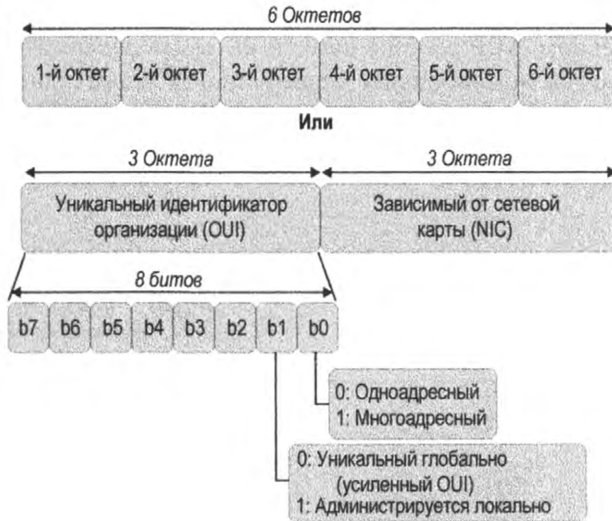


Рис. 1.21. Структура MAC-адреса

Когда IP-пакет приходит в сеть назначения, IP-адрес назначения, в случае IPv4, отображается с помощью протокола разрешения адреса (или протокола розыска соседа в IPv6) в MAC-адрес хоста назначения. Протокол отображения адреса обеспечивает перевод интернет-адреса в адресную систему канального уровня Ethernet-сети. ARP-таблица предназначена для быстрого просмотра известных хостов, поэтому ей не требуется посылать ARP-запрос на каждый кадр, который хочет передать хост. В примере 1.8 показана выдача локальной ARP-таблицы. Для этой цели все сетевые устройства кешируют ARP-адреса.

Пример 1.8. Таблица протокола разрешения адреса ARP

```
o → arp -a
? (192.168.0.1) at bc:a5:11:f1:5d:be on en0 ifscope [ethernet]
? (192.168.0.17) at 38:f9:d3:bc:8a:51 on en0 ifscope permanent [ethernet]
? (192.168.0.255) at ff:ff:ff:ff:ff:ff on en0 ifscope [ethernet]
? (224.0.0.251) at 1:0:5e:0:0:fb on en0 ifscope permanent [ethernet]
? (239.255.255.250) at 1:0:5e:7f:ff:fa on en0 ifscope permanent [ethernet]
```

Рис. 1.22 иллюстрирует обмен данными между хостами одной локальной сети. Браузер посылает HTTP-запрос веб-странице, размещенной на сервере назначения.

Через DNS он определяет, что сервер имеет IP-адрес `10.0.0.1`. Чтобы продолжить отсылку HTTP-запроса, ему требуется еще MAC-адрес сервера. Сначала запрашивающий компьютер просматривает кешированную ARP-таблицу, чтобы попытаться найти `10.0.0.1` среди всех существующих MAC-адресов сервера. Если MAC-адрес найден, то отсылается Ethernet-кадр с адресом назначения, равным MAC-адресу сервера. Кадр содержит IP-пакет, адресованный на `10.0.0.1`. Если в кеше адреса `10.0.0.1` не нашлось, то запрашивающий компьютер посылает широковещательное ARP-сообщение с MAC-адресом назначения `FF:FF:FF:FF:FF:FF`, которое принимается всеми хостами локальной сети, с просьбой сообщить о наличии адреса `10.0.0.1`. Сервер отвечает своим ARP-сообщением, в котором содержатся его MAC- и IP-адреса. В качестве части сообщения сервер иногда может запросить MAC-адрес компьютера с тем, чтобы внести его в свою ARP-таблицу для последующего использования. Запрашивающий компьютер получает и кеширует ответную информацию в своей ARP-таблице и таким образом получает возможность передавать HTTP-пакеты.

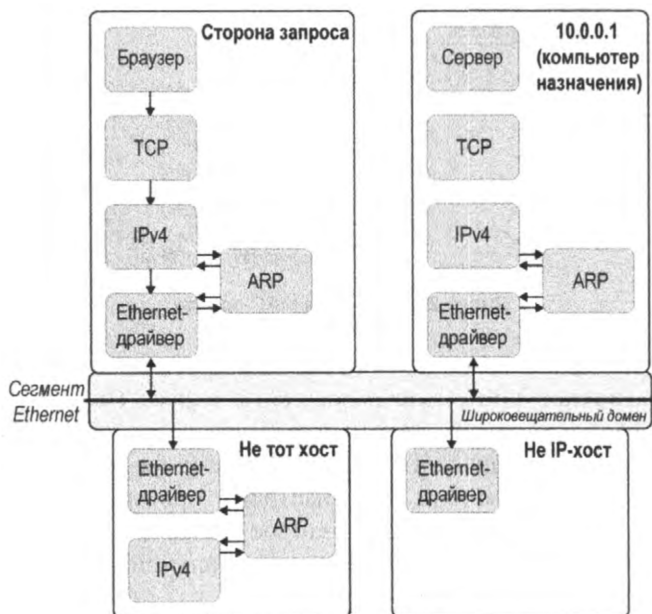


Рис. 1.22. ARP-запрос

Здесь мы встречаемся с одним из основных понятий в мире локальных сетей — широковещательным доменом (broadcast domain). Все пакеты в широковещательном домене получают все ARP-сообщения от хостов. Кроме того, все кадры посылаются всем узлам домена, хост сравнивает MAC-адрес назначения с своим собственным адресом. Хост отклоняет кадры, если они предназначены не для него. С ростом числа хостов в сети естественным образом растет и широковещательный трафик.

Мы снова можем воспользоваться системным вызовом `tcpdump`, чтобы отследить ARP-запросы в локальной сети (пример 1.9). Перехват дает информацию об ARP-

пакетах, типе используемого Ethernet, Ethernet (len 6), и протоколе высокого уровня IPv4. Выдача также позволяет узнать, кто запрашивает MAC-адрес IP-адреса, `Request who-has 192.168.0.1 tell 192.168.0.12`.

Пример 1.9. Выдача команды `tcpdump` для протокола ARP

```
o → sudo tcpdump -i en0 arp -vvv
tcpdump: listening on en0, link-type EN10MB (Ethernet), capture size 262144 bytes
17:26:25.906401 ARP, Ethernet (len 6), IPv4 (len 4),
Request who-has 192.168.0.1 tell 192.168.0.12, length 46
17:26:27.954867 ARP, Ethernet (len 6), IPv4 (len 4),
Request who-has 192.168.0.1 tell 192.168.0.12, length 46
17:26:29.797714 ARP, Ethernet (len 6), IPv4 (len 4),
Request who-has 192.168.0.1 tell 192.168.0.12, length 46
17:26:31.845838 ARP, Ethernet (len 6), IPv4 (len 4),
Request who-has 192.168.0.1 tell 192.168.0.12, length 46
17:26:33.897299 ARP, Ethernet (len 6), IPv4 (len 4),
Request who-has 192.168.0.1 tell 192.168.0.12, length 46
17:26:35.942221 ARP, Ethernet (len 6), IPv4 (len 4),
Request who-has 192.168.0.1 tell 192.168.0.12, length 46
17:26:37.785585 ARP, Ethernet (len 6), IPv4 (len 4),
Request who-has 192.168.0.1 tell 192.168.0.12, length 46
17:26:39.628958 ARP, Ethernet (len 6), IPv4 (len 4),
Request who-has 192.168.0.1 tell 192.168.0.13, length 46
17:26:39.833697 ARP, Ethernet (len 6), IPv4 (len 4),
Request who-has 192.168.0.1 tell 192.168.0.12, length 46
17:26:41.881322 ARP, Ethernet (len 6), IPv4 (len 4),
Request who-has 192.168.0.1 tell 192.168.0.12, length 46
17:26:43.929320 ARP, Ethernet (len 6), IPv4 (len 4),
Request who-has 192.168.0.1 tell 192.168.0.12, length 46
17:26:45.977691 ARP, Ethernet (len 6), IPv4 (len 4),
Request who-has 192.168.0.1 tell 192.168.0.12, length 46
17:26:47.820597 ARP, Ethernet (len 6), IPv4 (len 4),
Request who-has 192.168.0.1 tell 192.168.0.12, length 46

13 packets captured
233 packets received by filter
0 packets dropped by kernel
```

Дальнейшая сегментация сетевого уровня 2 возможна с использованием тегов виртуальной локальной сети (VLAN). В заголовке кадра Ethernet есть опциональный тег VLAN, который позволяет регулировать трафик по сети LAN. Выгодно использовать несколько виртуальных сетей VLAN в пределах одной LAN и управлять сетями с одной или другой точки. Маршрутизаторы между различными VLAN отфильтровывают широковещательный трафик, что повышает безопасность сети и предотвращает перегрузку. Такой подход часто используется сетевыми администраторами, а администраторы сетей на базе Kubernetes могут обратиться и к расши-

ренной версии технологии VLAN, называемой виртуальной расширяемой локальной сетью (VXLAN).

Рис. 1.23 показывает, каким образом VXLAN производит инкапсуляцию кадров уровня 2 в UDP-пакеты уровня 4. VXLAN повышает масштабируемость до 16 миллионов логических сетей и обеспечивает смежность для уровня 2 на всем пространстве IP-сетей. В Kubernetes эта технология используется для конструирования оверлейных сетей, о чем мы расскажем в следующих главах.

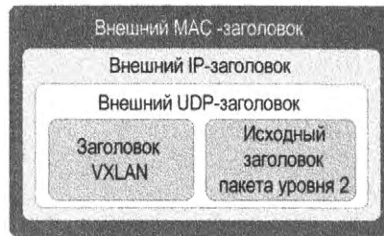


Рис. 1.23. Пакет VXLAN

Ethernet также задает характеристики среды для передачи кадров, как, например, витая пара, коаксиальный кабель, оптоволокно, радиочастоты или любая другая среда передачи, которой еще только суждено быть изобретенной — сеть на гамма-лучах или мгновенный коммуникатор Philotic Parallax из фильма «Игра Эндера». Ethernet даже определяет протоколы кодирования и обработки электрических сигналов, однако это находится за рамками данной публикации.

Уровень канала использует и другие протоколы, которые были разработаны для сетевой коммуникации. Здесь мы только слегка коснулись этих вопросов — данные ограничения связаны с тем, что мы приводим только то, что необходимо для базового понимания работы уровня канала в сетевой модели на основе Kubernetes.

Снова наш веб-сервер

Итак, наше путешествие по уровням модели TCP/IP завершено. На рис. 1.24 показаны все заголовки и блоки данных для каждого уровня, которые генерирует TCP/IP для того, чтобы пересылать пакеты по всему Интернету.

Давайте еще раз пройдем по этапам нашего путешествия и вспомним, что происходит на каждом из уровней. Пример 1.10 снова показывает наш веб-сервер, а пример 1.11 — запрос на него от клиента cURL, который уже был рассмотрен в этой главе.

Пример 1.10. Минимальный веб-сервер на языке Go

```
package main
import (
        "fmt"
        "net/http"
)
```

```
func hello(w http.ResponseWriter, _ *http.Request) {
        fmt.Fprintf(w, "Hello")
}
func main() {
      http.HandleFunc("/", hello)
      http.ListenAndServe("0.0.0.0:8080", nil)
}
```

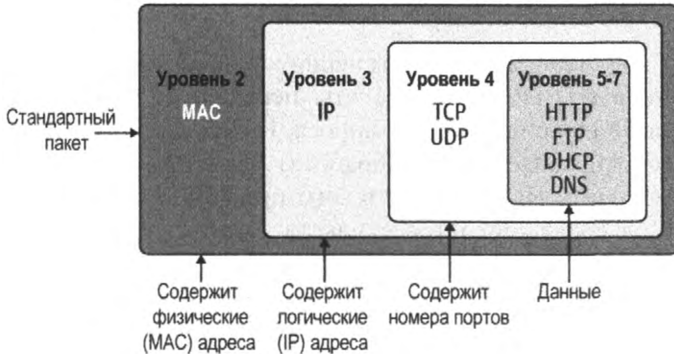


Рис. 1.24. Структура полного пакета в модели TCP/IP

Пример 1.11. Запрос со стороны клиента

```
o → curl localhost:8080 -vvv
*
Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080
> GET / HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Sat, 25 Jul 2020 14:57:46 GMT
< Content-Length: 5
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host localhost left intact
Hello* Closing connection 0
```

Мы начинаем с нашего веб-сервера из примера 1.10, который находится в режиме ожидания соединения. Клиент cURL запрашивает HTTP-сервер по адресу `0.0.0.0` на порте 8080. Клиент cURL определяет IP-адрес и номер порта из идентификатора URL и переходит к установлению TCP-соединения с сервером. Как только соеди-

нение — с использованием процедуры квитирования, — установлено, клиент cURL посылает HTTP-запрос. Веб-сервер активируется, на HTTP-сервере создается сокет 8080, который соответствует TCP-порту 8080, то же происходит на стороне клиента cURL, где генерируется случайный номер порта. Далее эта информация пересылается на сетевой уровень, где IP-адреса отправителя и получателя присоединяются к заголовку IP-пакета. На стороне клиента на уровне канала к Ethernet-кадру присоединяется MAC-адрес сетевой карты источника. Если MAC-адрес назначения неизвестен, то генерируется ARP-запрос для его поиска. Затем сетевая карта пересылает Ethernet-кадры на веб-сервер.

Когда веб-сервер получает запрос, он генерирует пакеты данных, содержащих ответную информацию (HTTP-ответ). Пакеты передаются обратно процессу cURL через Интернет на IP-адрес источника запроса. Как только процесс cURL получает ответный пакет, он отправляет его на обработку. На уровне канала данных из пакета убирается MAC-адрес. На уровне сетевого протокола происходит верификация IP-адреса, и затем он также убирается из пакета. Поэтому, если приложение требует доступа к IP-адресу клиента, то адрес должен запоминаться на уровне приложения, лучше всего это делать в HTTP-запросах и заголовке пакета Forward (переадресация). На следующем шаге из TCP-данных определяется сокет и также исключается. Пакет переадресуется на клиентское приложение, создавшее этот сокет. Клиент читает ответ и обрабатывает полученные данные. В данном случае идентификатор сокета был случайным числом, соответствующим процессу cURL. Все пакеты отправляются на cURL и собираются вместе в единый HTTP-ответ. Если бы мы использовали при выдаче опцию `-o`, то он бы был записан в файл, в противном случае выдача отображается на стандартном терминале.

Вау, почти 50 страниц текста и 50 лет развития сетей мы сжали всего в два абзаца! Но надо понимать, что базовые основы сетевых технологий, с которыми мы познакомились, — это начальные, но абсолютно необходимые знания, если вы хотите в широких масштабах использовать сети и кластеры Kubernetes.

Заключение

HTTP-транзакции, рассмотренные в этой главе, производятся в глобальном Интернете по миллиарду раз в день. Это число стоит иметь в виду, чтобы оценить всю пользу сетевых приложений на основе Kubernetes, которые позволяют свернуть сложные взаимодействия в довольно простые описания на языке YAML. Понимание масштаба проблем — это первый шаг на пути овладения навыками управления сетями Kubernetes. Рассматрев простой пример веб-сервера на Go и ознакомившись с его помощью с основами сетевых технологий, вы сможете начать управлять потоком пакетом внутрь и наружу ваших кластеров.

Итак, в данной главе мы рассмотрели следующие темы:

- Историю сетевых технологий.
- Модель OSI.
- Модель TCP/IP.

По ходу изложения мы касались многих аспектов функционирования сетей, выбор которых в первую очередь был обусловлен их связью с абстракциями в Kubernetes. В издательстве «O-Reilly» было издано несколько книг о TCP/IP, «TCP/IP Network Administration», написанные Крейгом Хантом и представляющие собой великолепное и глубокое руководство по всем аспектам TCP.

Мы рассказали, как развивались сетевые технологии, рассмотрели модель OSI, сравнили ее функциональность с моделью TCP/IP и на основе последней подробно изучили пример HTTP-запроса. В следующей главе мы познакомимся с тем, как это реализуется в архитектуре клиент-сервер в сетях на базе ОС Linux.

ГЛАВА 2

Поддержка сети в ОС Linux

https://t.me/it_boooks/2

Чтобы освоить работу с сетью в Kubernetes, нам понадобится познакомиться с основами поддержки сети в ОС Linux. Вообще говоря, Kubernetes — это мощный инструмент администрирования для компьютеров под Linux (или Windows!), что наглядно демонстрирует сетевой стек Kubernetes. В данной главе мы рассмотрим средства поддержки сети в Linux, обращая особое внимание на их связь с Kubernetes. Если вы уже хорошо знакомы с инструментами Linux по работе с сетью, то эту главу можете пропустить.

В главе приводится много системных программ Linux. Подробное описание этих программ можно получить, выполняя с консоли команду `man <program>`.

Базовые понятия

Снова обратимся к нашему веб-серверу на языке Go, который мы рассматривали в *главе 1*. Этот веб-сервер слушает порт 8080 и, получив HTTP-запрос, выводит "Hello" на консоль (см. пример 2.1).

Пример 2.1. Минимальный веб-сервер на языке Go

```
package main

import (
	"fmt"
	"net/http"
)

func hello(w http.ResponseWriter, _ *http.Request) {
	fmt.Fprintf(w, "Hello")
}

func main() {
	http.HandleFunc("/", hello)
	http.ListenAndServe("0.0.0.0:8080", nil)
}
```

Для соединения с портами 1-1023 (называемыми также «зарезервированные порты») требуются привилегии системного администратора — root.

Программам рекомендуется давать низший возможный приоритет, достаточный им для работы, так что в общем случае веб-сервис не имеет приоритета root. Поэтому многие программы слушают порты с номерами 1024 и выше (в частности, порт 8080 — это стандартное назначение для HTTP-запросов). По возможности старайтесь указывать непривилегированный порт и используйте перенаправления (переадресация в балансировщиках нагрузки, сервисы Kubernetes и т. п.), чтобы переориентировать доступный извне привилегированный порт на программу, слушающую непривилегированный порт.

В этом случае хакеры, воспользовавшиеся потенциальной уязвимостью вашего сервиса, не смогут получить высокий приоритет.

Предположим, что эта программа запущена на сервере под управлением Linux и внешний клиент делает запрос на /. Что происходит на сервере? Для начала наша программа должна прослушать адрес и порт. Программа создает сокет для этих адреса и порта и устанавливает с ним связь. Сокет будет получать запросы, направляемые с любого IP-адреса на указанный адрес и на порт 8080.

`0.0.0.0` в IPv4 и `[::]` в IPv6 — это универсальные адреса. Они дают совпадение со всеми адресами указанных протоколов и соответственно обеспечивают мониторинг всех возможных IP-адресов, если задаются как адреса связи в сокетах.

Это позволяет предоставить доступ к сервису в случае, если IP-адреса компьютеров, на которых он работает, заранее не известны. Большинство сетевых сервисов используют универсальные адреса.

Есть несколько способов просмотра сокетов. Например, команда `ls -lah /proc/<server proc>/fd` выдает список всех сокетов. Мы подробнее поговорим о программах для просмотра сокетов в конце данной главы.

Ядро отображает данный пакет на указанное соединение и использует внутренний модуль мониторинга состояний Linux для управления состоянием соединения. Как и в случае сокетов, есть несколько инструментов просмотра соединений, которые мы обсудим ниже в данной главе. На каждое соединение в Linux предусмотрен отдельный файл. Если соединение устанавливается, то ядро посылает уведомление нашей программе, которая таким образом получает возможность потоком передать данные в файл и из файла.

Воспользовавшись командой `strace`, мы может посмотреть, какие конкретно действия совершает наш веб-сервер:

```
$ strace ./main
execve("./main", ["./main"], 0x7ebf2700 /* 21 vars */) = 0
brk(NULL)                                = 0x78e000
uname({sysname="Linux", nodename="raspberrypi", ...}) = 0
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
= 0x76f1d000
[Content cut]
```

Поскольку `strace` перехватывает все системные вызовы, выполняемые сервером, выдача получается довольно обширная. Мы ее подсократили, ограничившись толь-

ко тем, что имеет отношение к сетевым системным вызовам. Мы пометили ключевые точки, чтобы выделить их из множества системных вызовов, совершаемых HTTP-сервером во время своего запуска:

```
openat(AT_FDCWD, "/proc/sys/net/core/somaxconn",
O_RDONLY|O_LARGEFILE|O_CLOEXEC) = 3
epoll_create1(EPOLL_CLOEXEC)        = 4 1
epoll_ctl(4, EPOLL_CTL_ADD, 3, {EPOLLIN|EPOLLOUT|EPOLLRDHUP|EPOLLET,
{u32=1714573248, u64=1714573248}}) = 0
fcntl(3, F_GETFL)            = 0x20000 (flags O_RDONLY|O_LARGEFILE)
fcntl(3, F_SETFL, O_RDONLY|O_NONBLOCK|O_LARGEFILE) = 0
read(3, "128\n", 65536)           = 4
read(3, "", 65532)                = 0
epoll_ctl(4, EPOLL_CTL_DEL, 3, 0x20245b0) = 0
close(3)                               = 0
socket(AF_INET, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, IPPROTO_TCP) = 3
close(3)                               = 0
socket(AF_INET6, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, IPPROTO_TCP) = 3 2
setsockopt(3, SOL_IPV6, IPV6_V6ONLY, [1], 4) = 0 3
bind(3, {sa_family=AF_INET6, sin6_port=htons(0),
        inet_pton(AF_INET6, "::1", &sin6_addr),
       sin6_flowinfo=htonl(0), sin6_scope_id=0}, 28) = 0
socket(AF_INET6, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, IPPROTO_TCP) = 5
setsockopt(5, SOL_IPV6, IPV6_V6ONLY, [0], 4) = 0
        bind(5, {sa_family=AF_INET6,
        sin6_port=htons(0), inet_pton(AF_INET6,
        "::ffff:127.0.0.1", &sin6_addr), sin6_flowinfo=htonl(0),
        sin6_scope_id=0}, 28) = 0
close(5)                      = 0
close(3)                      = 0
socket(AF_INET6, SOCK_STREAM|SOCK_CLOEXEC|SOCK_NONBLOCK, IPPROTO_IP) = 3
setsockopt(3, SOL_IPV6, IPV6_V6ONLY, [0], 4) = 0
setsockopt(3, SOL_SOCKET, SO_BROADCAST, [1], 4) = 0
setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
bind(3, {sa_family=AF_INET6, sin6_port=htons(8080),
        inet_pton(AF_INET6, "::", &sin6_addr),
        sin6_flowinfo=htonl(0), sin6_scope_id=0}, 28) = 0 4
listen(3, 128)                                       = 0
epoll_ctl(4, EPOLL_CTL_ADD, 3,
{EPOLLIN|EPOLLOUT|EPOLLRDHUP|EPOLLET, {u32=1714573248,
        u64=1714573248}}) = 0
       getsockname(3, {sa_family=AF_INET6, sin6_port=htons(8080),
        inet_pton(AF_INET6, "::", &sin6_addr), sin6_flowinfo=htonl(0),
sin6_scope_id=0},
[112->28]) = 0
accept4(3, 0x2032d70, [112], SOCK_CLOEXEC|SOCK_NONBLOCK) = -1 EAGAIN
    (Resource temporarily unavailable)
```

```
epoll_wait(4, [], 128, 0)                        = 0
epoll_wait(4, 5
```

Пояснения:

1. Открыть дескриптор файла.
2. Создать TCP-сокет для соединения по протоколу IPv6.
3. Отключить опцию `IPV6_V6ONLY` на сокете. Теперь он может работать как с IPv4, так и с IPv6.
4. Связать IPv6-сокет с портом 8080 (все адреса).
5. Ожидание запроса.

Как только сервер запустится, мы увидим вывод от `strace` при вызове команды `epoll_wait`.

В этой точке сервер слушает сокет и ждет уведомления от ядра о прибытии пакетов. Если мы пошлем запрос на наш сервер, то увидим сообщение "Hello":

```
$ curl <ip>:8080/
Hello
```

Если вы попытаетесь выполнить глубокую отладку веб-сервера с помощью `strace`, то не стоит использовать для этого веб-браузер. Дополнительные запросы или метаданные, посылаемые на сервер, могут увеличить нагрузку на него, либо браузер не станет посылать ожидаемые запросы. Например, многие браузеры автоматически запрашивают фавикон (значок веб-сайта или страницы). Они также пытаются кешировать файлы, повторно используют соединения и выполняют много других действий, которые затрудняют спрогнозировать, каким будет само сетевое взаимодействие. Если вам не нужна детальная информация, то используйте средства типа `curl` или `telnet`.

Программа `strace` показывает следующую информацию по нашему серверному процессу:

```
[{EPOLLIN, {u32=1714573248, u64=1714573248}}], 128, -1) = 1
accept4(3, {sa_family=AF_INET6, sin6_port=htons(54202), inet_pton(AF_INET6,
        "::ffff:10.0.0.57", &sin6_addr), sin6_flowinfo=htonl(0),sin6_scope_id=0},
         [112->28], SOCK_CLOEXEC|SOCK_NONBLOCK) = 5
epoll_ctl(4, EPOLL_CTL_ADD, 5, {EPOLLIN|EPOLLOUT|EPOLLRDHUP|EPOLLET,
        {u32=1714573120, u64=1714573120}}) = 0
getsockname(5, {sa_family=AF_INET6, sin6_port=htons(8080),
        inet_pton(AF_INET6, "::ffff:10.0.0.30", &sin6_addr), sin6_flowinfo=htonl(0),
sin6_scope_id=0}, [112->28]) = 0
setsockopt(5, SOL_TCP, TCP_NODELAY, [1], 4) = 0
setsockopt(5, SOL_SOCKET, SO_KEEPALIVE, [1], 4) = 0
setsockopt(5, SOL_TCP, TCP_KEEPINTVL, [180], 4) = 0
setsockopt(5, SOL_TCP, TCP_KEEPIDLE, [180], 4) = 0
accept4(3, 0x2032d70, [112], SOCK_CLOEXEC|SOCK_NONBLOCK) = -1 EAGAIN
         (Resource temporarily unavailable)
```

После просмотра сокета наш сервер записывает ответ ("Hello" вместе с параметрами HTTP-протокола) в файловый дескриптор. Оттуда ядро Linux (а также некоторые другие системы пользовательского пространства) переводит запрос в пакеты и передает эти пакеты обратно нашему клиенту cURL.

Резюмируем действия сервера, которые он выполняет после получения запроса:

- Возвращается значение `Epoll`, работа программы возобновляется.
- Сервер видит соединение с `::ffff:10.0.57`, в нашем примере это IP-адрес клиента.
- Сервер просматривает сокет.
- Сервер изменяет опции `KEEPALIVE`: он активирует `KEEPALIVE` и устанавливает 180-секундный интервал между проверками `KEEPALIVE`.

Мы в самом общем виде рассмотрели работу сетевого приложения в Linux. На самом деле выполняется еще множество действий, обеспечивающих бесперебойное функционирование систем и сетей. Рассмотрим более подробно работу сетевых модулей, что представляется наиболее важным именно для пользователей Kubernetes.

Сетевой интерфейс

Компьютеры используют *сетевой интерфейс* для взаимодействия с внешним миром. Сетевые интерфейсы могут быть физическими (как, например, сетевой контроллер Ethernet) или виртуальными. Виртуальный сетевой интерфейс не связан с физическим устройством, это абстрактный интерфейс, предоставляемый компьютером-хостом или гипервизором.

Сетевым интерфейсам назначаются IP-адреса. Типичный интерфейс может иметь один адрес типа IPv4, один — IPv6, возможна также ситуация, когда одному интерфейсу назначается несколько адресов.

В Linux сетевой интерфейс может быть физическим (сетевая карта Ethernet или порт) или виртуальным. Команда `ifconfig` выдает список всех сетевых интерфейсов и их конфигурации, включая IP-адреса.

Петлевой интерфейс (он же loopback-интерфейс) — это специальный интерфейс для коммуникации внутри одного и того же хоста. Стандартный IP-адрес для такого интерфейса — `127.0.0.1`. Посылаемые на loopback-интерфейс пакеты не покидают данный хост, а прослушивающие адрес `127.0.0.1` процессы доступны только процессам на том же хосте. Имейте в виду, что разрешать процессу прослушивать `127.0.0.1` не есть действие внутри границ безопасности. В Kubernetes была задокументированная уязвимость CVE-2020-8558, при которой правила kube-`proxy` позволяли некоторым удаленным системам получить доступ к `127.0.0.1`. Петлевой интерфейс обычно обозначается `lo`.

Команда `ip` тоже может использоваться для просмотра сетевых интерфейсов.

В примере 2.2 мы рассмотрим типичный результат команды ifconfig.

Пример 2.2. Выдача команды ifconfig для компьютера с одним физическим сетевым интерфейсом (ens4) и интерфейсом типа loopback

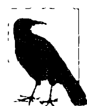
```
$ ifconfig
ens4: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1460
      inet 10.138.0.4 netmask 255.255.255.255 broadcast 0.0.0.0
      inet6 fe80::4001:aff:fe8a:4 prefixlen 64 scopeid 0x20<link>
      ether 42:01:0a:8a:00:04 txqueuelen 1000 (Ethernet)
      RX packets 5896679 bytes 504372582 (504.3 MB)
      RX errors 0 dropped 0 overruns 0 frame 0
      TX packets 9962136 bytes 1850543741 (1.8 GB)
      TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
      inet 127.0.0.1 netmask 255.0.0.0
      inet6 ::1 prefixlen 128 scopeid 0x10<host>
      loop txqueuelen 1000 (Local Loopback)
      RX packets 352 bytes 33742 (33.7 KB)
      RX errors 0 dropped 0 overruns 0 frame 0
      TX packets 352 bytes 33742 (33.7 KB)
      TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

При работе контейнеров на хосте создается виртуальный сетевой интерфейс для каждого пода, так что для узла Kubernetes этот список будет существенно длиннее. Мы рассмотрим подробно сетевые операции с контейнерами в *главе 3*.

Интерфейс сетевого моста

Интерфейс типа мост (Bridge), рис. 2.1, позволяет системным администраторам создавать несколько сетей второго уровня на одном и том же хосте. Другими словами, *мостовой интерфейс* работает на хосте как коммутатор сетевых интерфейсов, обеспечивая их бесперебойное переключение. Такие интерфейсы позволяют подам, имеющим собственные сетевые интерфейсы, взаимодействовать с остальной сетью через сетевой интерфейс узла.



Более подробно о работе интерфейсов типа мост в ОС Linux можно прочитать в документации.

В примере 2.3 показывается, как создать мост с именем br0 и связать виртуальное Ethernet-устройство (veth), veth, с физическим устройством, eth0, используя ip.

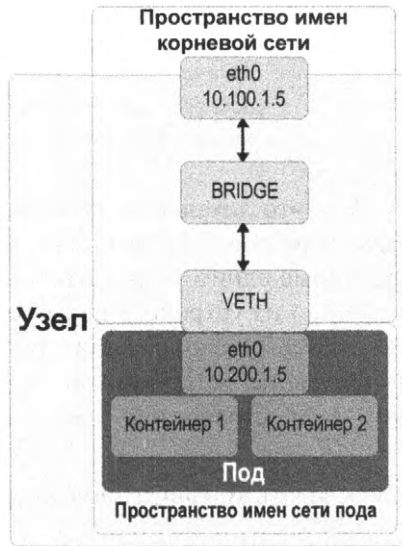


Рис. 2.1. Интерфейс сетевого моста

Пример 2.3. Создание интерфейса сетевого моста и подключение к паре виртуальных устройств veth

```
## Добавить новый bridge-интерфейс с именем br0.
# ip link add br0 type bridge
## Подключить eth0 к нашему мосту.
# ip link set eth0 master br0
## Подключить veth к нашему мосту.
# ip link set veth master br0
```

Для создания мостовых интерфейсов и управления ими также можно использовать команду `brctl`. В примере 2.4 показаны некоторые опции этой команды.

Пример 2.4. Опции команды `brctl`

```
$ brctl
$ commands:
addbr          <мост>                   добавить мост
delbr          <мост>                   удалить мост
addif          <мост> <устр-во>         добавить мосту интерфейс
delif          <мост> <устр-во>         удалить интерфейс из моста
setageing      <мост> <время>           задать время старения
setbridgeprio  <мост> <приор>           задать приоритет моста
setfd          <мост> <время>           задать задержку переадресации моста
sethello       <мост> <время>           задать интервал hellotime
setmaxage      <мост> <время>           задать макс. возраст сообщения
setpathcost    <мост> <порт> <стоим>    задать стоимость пути
```

```
setportprio    <мост> <порт> <приор>   задать приоритет порта
show                                    показать список мостов
showmacs       <мост>                   показать список mac-адресов
showstp        <мост>                   показать данные о протоколе stp
stp            <мост> <сост>            вкл/выкл поддержку stp
```

Виртуальное устройство veth — это локальный туннель Ethernet. Устройства veth создаются попарно, как показано на рис. 2.1, где под получает доступ к интерфейсу `eth0` через veth. Пакеты, переданные одним устройством в паре, сразу же получаются другим устройством. Если одно из устройств пары не функционирует, вся пара тоже отключена. В Linux создание сетевого моста производится командами `brcl` и ip. Используйте конфигурирование устройства `veth`, если пространствам имен требуется взаимодействовать с пространством имен главного хоста или между собой.

Пример 2.5 демонстрирует, как задать конфигурацию устройства veth.

Пример 2.5. Создание виртуального устройства veth

```
# ip netns add net1
# ip netns add net2
# ip link add veth1 netns net1 type veth peer name veth2 netns net2
```

В примере 2.5 мы показываем, как создать два сетевых пространства имен (не путать с пространствами имен в Kubernetes), `net1` и `net2`, и пару виртуальных устройств `veth`, где `veth1` назначается пространству `net1`, а `veth2` — пространству `net2`. Эти два пространства имен соединены с указанной парой интерфейсов `veth`. Если паре назначить IP-адреса, то можно будет использовать `ping` и осуществлять взаимодействие между двумя пространствами имен.

В контейнерных сетевых интерфейсах CNI и в Kubernetes интерфейсы сетевого моста используются для управления интерфейсами, IP-адресами и пространством имен сетевых контейнеров. Более подробно мы рассмотрим эти вопросы в *главе 3*.

Обработка пакетов в ядре Linux

Ядро Linux отвечает за взаимодействие между пакетами и согласованный поток данных для программ. В частности, мы рассмотрим, каким образом ядро работает с соединениями, поскольку маршрутизация и брандмауэры — ключевые функции Kubernetes — базируются именно на процедурах пакетной обработки Linux.

Netfilter (межсетевой фильтр)

Межсетевые фильтры netfilter, появившиеся в версиях Linux старше 2.3, являются важной компонентой процедуры обработки пакетов. Межсетевые фильтры — это набор перехватов ядра, которые позволяют программам пространства пользователя

обрабатывать пакеты от имени ядра. Если говорить коротко, то программа регистрирует себя на определенном перехвате системы Netfilter, а затем ядро вызывает эту программу при обработке соответствующих пакетов. Программа может сообщить ядру, что делать с пакетом (например, сбросить его), или она может послать ядру модифицированный пакет. С помощью Netfilter разработчик может создавать программы, которые работают в пространстве пользователя и обрабатывают пакеты. Межсетевые фильтры создаются с помощью утилиты `iptables`, чтобы отделить программы, относящиеся к ядру, от программ пространства пользователя.

Страница netfilter.org содержит прекрасное руководство по использованию системы Netfilter и утилиты iptables.

Система Netfilter имеет 5 перехватов, которые перечислены в табл. 2.1.

Netfilter выполняет каждый перехват на определенной стадии движения пакета через ядро. Понимание перехватов в Netfilter является ключевым для понимания работы утилиты `iptables`, поскольку эти перехваты являются прямым отображением концепции цепочек из `iptables`.

Таблица 2.1. Перехваты системы Netfilter

Перехват Netfilter	Имя цепочки Iptables	Описание
NF_IP_PRE_ROUTING	PREROUTING	Активируется при приходе пакета из внешней системы
NF_IP_LOCAL_IN	INPUT	Активируется, когда IP-адрес назначения пакета совпадает с адресом данного устройства
NF_IP_FORWARD	NAT	Активируется для пакетов, для которых ни адрес источника, ни адрес получателя не совпадают с IP-адресом устройства (другими словами, пакетов, для которых данное устройство прокладывает маршрут от имени других устройств)
NF_IP_LOCAL_OUT	OUTPUT	Активируется, когда пакет, имеющий источником данное устройство, его покидает
NF_IP_POST_ROUTING	POSTROUTING	Активируется, когда любой пакет (независимо от происхождения) покидает устройство

Таким образом, Netfilter активирует каждый перехват на определенной стадии обработки пакета и при соответствующих условиях. Перехваты Netfilter показаны на потоковой диаграмме на рис. 2.2.

Глядя на эту диаграмму, мы видим, что для каждого конкретного пакета возможны вызовы только определенных перехватов Netfilter. Например, пакет, имеющий источником локальный процесс, всегда активирует перехват `NF_IP_LOCAL_OUT`, а затем `NF_IP_POST_ROUTING`. Набор перехватов зависит от двух вещей: является ли хост ис-

точником пакета и является ли хост назначением пакета. Отметим, что если процесс посылает пакет, назначенный на тот же самый хост, то он активирует перехват `NF_IP_LOCAL_OUT`, а затем `NF_IP_POST_ROUTING` перед тем, как повторно зайти в систему и активировать `NF_IP_PRE_ROUTING` и `NF_IP_LOCAL_IN`.

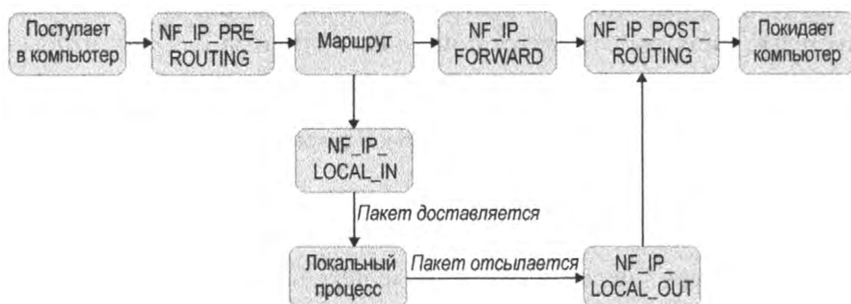


Рис. 2.2. Прохождение пакетов через перехваты системы Netfilter

В некоторых системах есть возможность подмены (спуфинга) такого пакета путем задания поддельного адреса (т. е. задать пакету адрес источника и адрес назначения как `127.0.0.1`). Обычно при попадании такого пакета на внешний интерфейс он отфильтровывается системой Linux. Вообще говоря, Linux отфильтровывает пакеты, если пакет попадает на интерфейс, а адрес источника пакета в той сети не существует. Пакет с «невозможным» IP-адресом источника называется *марсианским пакетом*. Фильтрация марсианских пакетов в Linux может быть отключена. Но это влечет за собой существенный риск, когда любой сервис на хосте будет считать, что трафик от локального хоста заслуживает большего доверия, чем трафик извне. Так может произойти, если интерфейс API или база данных получают доступ к хосту без надежной аутентификации.

В Kubernetes была как минимум одна уязвимость, CVE-20020-8558, когда пакеты с другого хоста, но с IP-адресом источника, установленным в `127.0.0.1`, получали доступ к портам, которые должны были быть доступны только локально. Среди прочего это означало, что если узел в панели управления Kubernetes запускал `kube-proxy`, то другие машины сети узла могли использовать «доверенную аутентификацию» для соединения с сервером API, тем самым получая права собственности на кластер.

Технически это не было связано с отсутствием фильтрации марсианских пакетов, поскольку пакеты-нарушители поступали от устройства с петлевым интерфейсом, которое находилось в той же сети и имело адрес `127.0.0.1`. Подробности можно прочитать на GitHub.

В табл. 2.2 показан порядок перехватов системы Netfilter для пакетов, имеющих различные источники и назначения.

Отметим, что пакеты, адресованные устройством ему самому, активируют `NF_IP_LOCAL_OUT` и `NF_IP_POST_ROUTING` и «покидают» сетевой интерфейс. Затем они снова «появятся на входе» и будут восприниматься как пакеты от любого другого источника.

Таблица 2.2. *Перехваты системы Netfilter в зависимости от происхождения и назначения пакетов*

Источник пакета	Назначение пакета	Перехваты (по порядку)
Локальное устройство	Локальное устройство	NF_IP_LOCAL_OUT, NF_IP_LOCAL_IN
Локальное устройство	Локальное устройство	NF_IP_LOCAL_OUT, NF_IP_POST_ROUTING
Внешнее устройство	Локальное устройство	NF_IP_PRE_ROUTING, NF_IP_LOCAL_IN
Внешнее устройство	Локальное устройство	NF_IP_PRE_ROUTING, NF_IP_FORWARD, NF_IP_POST_ROUTING

Преобразование сетевых адресов (NAT) затрагивает только локальную маршрутизацию в перехватах `NF_IP_PRE_ROUTING` и `NF_IP_LOCAL_OUT` (например, ядро не выполняет маршрутизацию после того, как пакет прошел через `NF_IP_LOCAL_IN`). Это отражается в структуре `iptables`, где преобразование сетевых адресов источника и назначения может быть выполнено только в определенных перехватах/цепочках.

Программа может отследить перехват через вызов `NF_REGISTER_NET_HOOK` (`NF_REGISTER_HOOK` в версиях раньше Linux 4.13) вместе с функцией обработки. Перехват будет вызываться каждый раз, когда параметры пакета совпадут с заданными. Таким образом программы типа `iptables` интегрируются с Netfilter, хотя весьма вероятно, что вам не придется делать это самим.

Перечислим действия, которые перехват Netfilter может запустить в зависимости от возвращаемой величины:

Accept

Продолжить обработку пакета.

Drop

Сбросить пакет без дальнейшей обработки.

Queue

Передать пакет программе пространства пользователя.

Stolen

Дальнейшие перехваты не выполняются, пакет становится собственностью программы пространства пользователя.

Repeat

Пакет снова попадает на перехват и переобрабатывается.

Перехваты могут возвращать измененные пакеты. Это позволяет программам выполнять такие действия, как повторная маршрутизация, маскировка пакетов, изменение времени жизни пакета (TTL) и т. д.

Conntrack

Conntrack — это компонент системы Netfilter, предназначенный для контроля состояния соединения (внешнего и внутреннего) с компьютером. При этом пакет непосредственно связывается с конкретным соединением. Данная функции позволяет сделать поток пакетов более прозрачным. Conntrack может быть как обязательным инструментом, так и опцией — в зависимости от способа его использования. В общем случае Conntrack особенно рекомендован для систем с брандмауэрами и преобразованием сетевых адресов (NAT).

Контроль соединения позволяет брандмауэрам различать пакеты-ответы на запрос и остальные пакеты. Брандмауэр может быть сконфигурирован таким образом, чтобы пропускать входящие пакеты, являющиеся частью существующего соединения, и не пропускать входящие пакеты, которые не являются частью соединения. Например, программе может быть разрешено устанавливать выходящее соединение и выполнять HTTP-запрос, но без того, чтобы удаленный сервер в любом другом случае мог бы пересылать данные или устанавливать входящие соединения.

Механизм NAT требует для своей работы Conntrack. Утилита `iptables` использует NAT двух типов: SNAT (NAT источника, когда `iptables` переписывает адрес источника) и DNAT (NAT назначения, `iptables` переписывает адрес назначения). Преобразование сетевых адресов — широко используемая процедура, даже ваш домашний роутер наверняка использует SNAT и DNAT, чтобы распределить трафик между вашим публичным IPv4-адресом и локальным адресом каждого из сетевых устройств. С помощью функции контроля соединения пакеты автоматически привязываются к своим соединениям и легко модифицируются через изменение SNAT/DNAT. Это позволяет выполнять согласованную маршрутизацию, как, например, привязку соединения к определенному бэкенду или компьютеру в баласировщике нагрузки. Это особенно актуально для Kubernetes, где балансировщик нагрузки для сервисов реализован в `kube-proxy` на базе `iptables`. Без функции контроля соединения каждый пакет пришлось бы в явном виде переадресовывать на один и тот же адрес назначения, что представляется едва ли выполнимым (представим себе, что список адресов назначения изменился...).

Для идентификации соединений Conntrack использует *кортеж (tuple)*, который состоит из адреса источника, порта источника, адреса назначения, порта назначения и протокола 4-го уровня. Этот набор из 5 компонентов данных является минимальным набором, позволяющим идентифицировать каждое конкретное соединение 4-го уровня. Все такие соединения имеют адрес и порт на каждой стороне соединения: Интернет использует адреса для маршрутизации, а компьютеры используют номера портов для определения соответствия между ними и приложениями. Последний компонент — протокол 4-го уровня — присутствует в кортеже, поскольку программа привязывается к порту в режиме TCP или UDP (привязка к одному порту не исключает привязку к другому). Conntrack называет эти соединениями *потоками*. Поток содержит метаданные, касающиеся соединения и его состояния.

Conntrack запоминает потоки в хеш-таблице, как показано на рис. 2.3, используя при этом кортеж в качестве ключа. Размер пространства ключей конфигурируемый.

Большое пространство требует больше памяти для хранения соответствующих массивов, но зато уменьшит количество обращений потоков к одному и тому же ключу и, будучи оформлено в связанный список, приведет к уменьшению времени просмотра потоков. Максимальное число потоков также задаваемо. Серьезной проблемой является, когда Conntrack достигает границ доступного пространства, так что новые соединения некуда запоминать. Есть и другие конфигурационные опции, например время жизни соединения. В стандартной системе достаточно значений по умолчанию. Но система с большим количеством соединений может натолкнуться на пределы расширения. Если ваш хост имеет прямой доступ к Интернету, то переполнение Conntrack короткоживущими или не полностью установленными соединениями может легко привести к отказу от обслуживания (DOS).

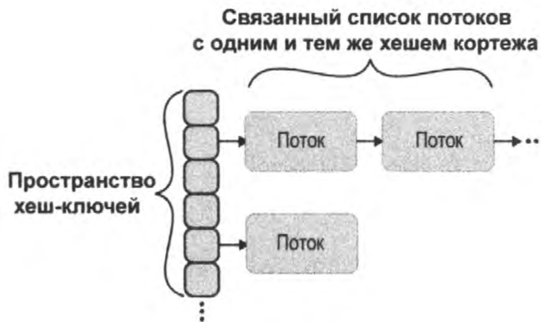


Рис. 2.3. Структура потоков в Conntrack

Максимальный размер для Conntrack обычно задается в `/proc/sys/net/nf_conntrack_max`, а размер хеш-таблицы — в `/sys/module/nf_conntrack/parameters/hashsize`.

Conntrack возвращает состояние соединения, одно из четырех. Важно отметить, что Conntrack является инструментом 3-го (сетевого) уровня, поэтому его состояния отличаются от состояний 4-го уровня (уровень протокола). Состояния Conntrack представлены в табл. 2.3.

Таблица 2.3. *Состояния Conntrack*

Состояние	**Описание**	**Пример**
NEW	Пакет отослан или получен, ответ не получен	Получен TCP SYN
ESTABLISHED	Пакеты пересылаются в обоих направлениях	Получен TCP SYN, передан TCP SYN/ACK
RELATED	Открыто дополнительное соединение, метаданные которого указывают, что оно имеет отношение к основному соединению. Обработка таких связанных соединений довольно сложна	FTP-программа со статусом соединения ESTABLISHED открывает дополнительные соединения для обмена данными
INVALID	Пакет испорчен или не соответствует другим состояниям Conntrack	Получен TCP RST, без предварительного соединения

Хотя модуль Conntrack встроен в ядро, он может быть неактивирован в вашей системе. Для его установки необходимо загрузить определенные модули ядра, и вам необходимо иметь соответствующие правила в утилите iptables (вообще говоря, Conntrack в нормальном состоянии неактивен, если в его работе нет надобности). Conntrack требует, чтобы был активирован модуль ядра nf_conntrack_ipv4. Команда lsmod | grep nf_conntrack покажет, загружен ли модуль, а загрузить его можно с помощью sudo modprobe nf_conntrack. Может понадобиться установить интерфейс командной строки для conntrack, чтобы отслеживать состояния.

Если Conntrack активен, то команда conntrack -L показывает все текущие потоки. Путем задания дополнительных флагов можно осуществлять фильтрацию показываемых потоков.

Посмотрим внимательнее на поток Conntrack:

```
tcp   6 431999 ESTABLISHED src=10.0.0.2 dst=10.0.0.1
sport=22 dport=49431 src=10.0.0.1 dst=10.0.0.2 sport=49431 dport=22 [ASSURED]
mark=0 use=1

<протокол> <номер протокола> <TTL потока> [состояние потока>]
<ip источника> <ip назначения> <порт источника> <порт назн.> [] <ожидаемый ответный
пакет>
```

Ожидаемый ответный пакет приходит в формате <ip источника> <ip назн> <порт источника> <порт назн>. Это идентификатор, который мы ожидаем увидеть, когда удаленная система посылает пакет. Отметим, что в нашем примере в данных источника и назначения адрес и порты идут в обратном порядке. Это бывает часто, хотя и не всегда. Например, если вычислительная машина расположена после роутера, то пакеты с назначением на эту машину будут адресованы на роутер, в то время как пакеты, посылаемые машиной, будут иметь в качестве источника адрес машины, а не адрес роутера.

В предыдущем примере адрес 10.0.0.1 установил TCP-соединение с порта 49431 на порт 22 на машине 10.0.0.2. Это функция SSH, хотя Conntrack не имеет возможности показывать информацию о программах прикладного уровня.

Команды, подобные grep, позволяют отслеживать состояния Conntrack и получать статистику:

```
grep ESTABLISHED /proc/net/ip_conntrack | wc -l
```

Маршрутизация

Обрабатывая пакет, ядро должно принять решение, куда его отправить. В большинстве случаев адрес назначения принадлежит устройству, находящемуся в другой сети. Например, представим, что вы с вашего персонального компьютера пытаетесь установить соединение с адресом 1.2.3.4. Этот адрес не принадлежит вашей сети, в лучшем случае ваш компьютер передаст вас другому компьютеру, с которого будет ближе до 1.2.3.4. Делается это с помощью *таблицы маршрутизации*, которая сопоставляет известные подсети с IP-адресом и интерфейсом шлюза. Список мар-

шрутов выдается с помощью команды route (или `route -n`, если хотите получить просто IP-адреса, а не имена хостов). В общем случае компьютер получает маршрут по локальной сети и маршрут на `0.0.0.0/0`. Напомним, что подсети могут адресоваться по методу бесклассовой доменной маршрутизации CIDR (например, `10.0.0.0/24`) или заданием IP-адреса и маски (например, `10.0.0.0` и `255.255.255.0`).

Ниже приводится типичная таблица маршрутизации для компьютера в локальной сети с доступом в Интернет:

```
#route
Kernel IP routing table
Destination   Gateway      Genmask        Flags  Metric   Ref   Use    Iface
0.0.0.0       10.0.0.1     0.0.0.0        UG     303      0     0      eth0
10.0.0.0      0.0.0.0      255.255.255.0  U      303      0     0      eth0
```

В приведенном выше примере запрос на `1.2.3.4` будет послан на `10.0.0.1` с интерфейсом `eth0`, поскольку `1.2.3.4` находится в подсети, описываемой по первому методу (`0.0.0.0/0`), а не в подсети, описываемой по второму (`10.0.0.0/24`). Подсети задаются через адрес назначения и значение `genmask`.

Linux предпочитает маршрутизировать пакеты по «специфичности» (насколько «маленькой» является совпадающая подсеть), а затем по «весу» (значение `Metric` в выдаче `route`). В нашем примере пакет, адресованный на 10.0.0.1, всегда будет пересылаться на шлюз `0.0.0.0`, т. к. этот маршрут совпадает с меньшим набором адресов. Если есть два маршрута с одинаковой специфичностью, то предпочтительным выбором будет маршрут, у которого меньшая метрика.

Некоторые плагины контейнерных сетевых интерфейсов CNI активно используют таблицу маршрутизации.

После рассмотрения некоторых базовых концепций обработки пакетов ядром Linux мы можем перейти к знакомству с тем, как работает высокоуровневая маршрутизация для пакетов и соединений.

Высокоуровневая маршрутизация

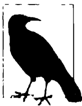
Linux располагает обширными средствами администрирования пакетов. Они помогают пользователям Linux создавать брандмауэры, отслеживать трафик, осуществлять маршрутизацию пакетов и даже реализовывать балансировку нагрузки. Kubernetes использует некоторые из этих средств для поддержки связности узла и пода, а также для управлениями сервисами Kubernetes. В данной книге мы рассмотрим три инструмента, которые широко представлены в Kubernetes. Все конфигурации Kubernetes используют так или иначе утилиту `iptables`, но существует много путей, как это конкретно происходит. Мы также рассмотрим виртуальный сервер IPVS (который имеет встроенную поддержку в `kube-proxy`) и функцию ядра eBPF, используемую Cilium (альтернатива `kube-proxy`).

Сведения из данного раздела понадобятся нам в *главе 4*, когда мы будем разбирать сервисы и `kube-proxy`.

Утилита iptables

Утилита iptables является основным рабочим инструментом системных администраторов Linux. Она может использоваться для создания брандмауэров и системных журналов, для модификации и переадресации пакетов и даже для грубой реализации разветвителей подключения. iptables использует систему Netfilter, что позволяет iptables перехватывать и изменять пакеты.

Правила применения iptables могут быть чрезвычайно сложными. Есть много средств, предоставляющих упрощенный интерфейс для работы с правилами iptables, например брандмауэры типа ufw и firewalld. Компоненты Kubernetes (конкретно kubelet и kube-proxy) тоже относятся к подобным генераторам правил iptables. Понимание работы iptables важно для понимания того, как осуществляется доступ и маршрутизация для подов и узлов в кластерах.



Большинство дистрибутивов Linux постепенно заменяют iptables на `nftables` — похожую, но более мощную программу также на основе Netfilter. Некоторые дистрибутивы уже поставляют версию `iptables`, которая работает на основе `nftables`.

В Kubernetes уже известны многие проблемы, связанные с конверсией `iptables/nftables`. Мы настоятельно не рекомендуем использовать версии `iptables` на базе `nftables` ни сейчас, ни в ближайшем будущем.

В утилите `iptables` присутствуют три ключевые составляющие: таблицы, цепочки и правила. Они представляют собой иерархию: таблица содержит цепочки, а цепочка содержит правила.

Таблицы организуют правила согласно типу действия, который они вызывают. iptables имеет широкий спектр функций, которые в таблицах упорядочиваются по группам. Три наиболее используемые таблицы — это Фильтры (для правил, относящихся к брандмауэрам), NAT (для правил, связанных с преобразованием сетевых адресов NAT) и Mangle (правила модификации пакетов помимо NAT). `Iptables` производит вычисления, используя таблицы в определенном порядке, который мы разберем позднее.

Цепочки содержат список правил. Когда пакет доходит до цепочки, то вычисления на основе правил цепочки производятся в определенном порядке. Цепочки работают внутри таблицы и упорядочивают правила в соответствии с перехватами Netfilter. Всего есть пять встроенных цепочек верхнего уровня, каждая из которых соответствует перехвату Netfilter (напомним, что Netfilter разрабатывался вместе с `iptables`). Таким образом, выбор цепочки, в которую включается правило, определяет, будет ли это правило приниматься во внимание при обработке данного пакета.

Правила представляют собой комбинацию условия и действия (называемого цель, target). Например, «если пакет адресован на порт 22, то сбросить его». `Iptables` оценивает состояние отдельных пакетов, при этом цепочки и таблицы определяют, по каким правилам будет проводиться эта оценка.

Процедура исполнения последовательности «таблица → цепочка → действие» весьма сложна, на этот счет существует множество хитрых диаграмм, описываю-

щих всю совокупность состояний системы. Далее мы будем рассматривать эту процедуру по частям.

В процессе продвижения по теме может оказаться полезным обратиться к ранее рассмотренным материалам. Структуры таблиц, цепочек и правил тесно переплетены, поэтому невозможно понять одно, не понимая при этом, как работают другие.

Таблицы *iptables*

Таблица в `Iptables` есть отображение определенного набора действий, при этом каждая таблица «отвечает» за выполнение определенных операций. Более конкретно: таблица может содержать только заданные типы действий, и многие типы действий могут использоваться только в заданных таблицах. Всего `iptables` имеет пять таблиц, которые перечислены в табл. 2.4.

Таблица 2.4. *Таблицы* `Iptables`

Таблица	Назначение
FILTER	Определяет, будет ли пакет принят или отфильтрован
NAT	Используется для изменения IP-адресов источника или назначения
MANGLE	Может производить изменения в заголовках пакета, не относящихся к NAT. Также может отличать пакеты с метаданными, доступными только для `iptables`
RAW	Позволяет модифицировать пакет перед тем, как будет произведен контроль состояния соединения и перед вычислениями на базе остальных таблиц. Обычно используется для выключения функции контроля состояния соединения для определенных пакетов
SECURITY	Эту таблицу использует модуль SELinux. Не работает на устройствах, на которых не установлен SELinux

В данной книге таблица Security обсуждаться не будет. Но если вы используете SELinux, то вам полезно понимать, как она функционирует.

Программа `Iptables` исполняет таблицы в заданном порядке: Raw, Mangle, NAT, Filter. Однако данный порядок может быть нарушен цепочками. Обычно пользователи Linux считают, что «таблицы содержат цепочки», но это может оказаться не совсем верным. На самом деле порядок исполнения «сначала цепочки, затем таблицы». Так, например, пакет может активировать `Raw PREROUTING`, `Mangle PREROUTING`, `NAT PREROUTING`, а затем запустить таблицу Mangle в `INPUT` или `FORWARD` цепочке (в зависимости от пакета). Подробнее мы рассмотрим эти вопросы в следующем разделе, посвященном цепочкам.

Цепочки *iptables*

Цепочки `iptables` — это списки правил. Когда пакет активирует (или проходит через) цепочку, то последовательно производятся вычисления на основе правил,

пока пакет не попадет на завершающее действие (например, DROP) или не достигнет конца цепочки.

Встроенные цепочки верхнего уровня — это PREROUTING, INPUT, NAT, OUTPUT и POSTROUTING. Все они функционируют на основе перехватов Netfilter, каждая цепочка соответствует одному перехвату. Эти пары показаны в табл. 2.5. Существуют также пользовательские подцепочки, назначение которых — помогать в формировании правил.

Таблица 2.5. *Цепочки* `iptables` *и соответствующие им перехваты Netfilter*

Цепочка `Iptables`	**Перехват Netfilter**
PREROUTIN	NF_IP_PRE_ROUTING
INPUT	NF_IP_LOCAL_IN
NAT	NF_IP_FORWARD
OUTPUT	NF_IP_LOCAL_OUT
POSTROUTING	NF_IP_POST_ROUTING

Возвращаясь к нашей диаграмме перехватов Netfilter, мы можем изобразить аналогичную диаграмму, представляющую для конкретного пакета порядок выполнения цепочки программой iptables (см. рис. 2.4).

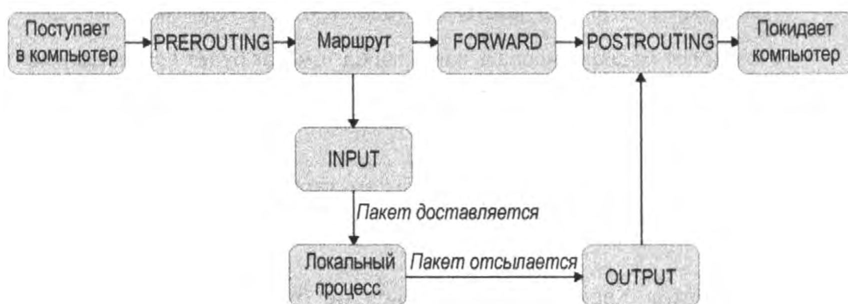


Рис. 2.4. Прохождение пакета через цепочки программы `Iptables`

Как это было и в Netfilter, число возможных вариантов движения пакета по цепочкам невелико (предполагая, что пакет не отвергается на входе и не отфильтровывается по пути). Рассмотрим пример с тремя устройствами, имеющими IP-адреса 10.0.0.1, 10.0.0.2 и 10.0.0.3 соответственно. Покажем несколько сценариев маршрутизации с точки зрения устройства 1 (IP-адрес 10.0.0.1) — см. табл. 2.6.

Вы можете потренироваться в работе с цепочками, используя правила LOG. Например, команда

```
Iptables -A OUTPUT -p tcp -dport 22 -j LOG
--log-level info --log-prefix "ssh-output"
```

отправляет TCP-пакеты на порт 22, где они обрабатываются цепочкой OUTPUT, с log-префиксом `"ssh-output"`. Учтите, что размер log-файла очень быстро может

стать неприемлемым. При удаленном соединении с важными хостами будьте особенно внимательны.

Таблица 2.6. Цепочки `Iptables`, исполняемые в различных сценариях маршрутизации

Описание пакета	Источник пакета	Назначение пакета	Обрабатываемые таблицы
Входящий пакет, с другого устройства	10.0.0 2	10.0.0 1	PREROUTING, INPUT
Входящий пакет, не предназначенный для данного устройства	10.0.0.2	10.0.0.3	PREROUTING, NAT, POSTROUTING
Выходящий пакет, сгенерирован локально, назначен на другое устройство	10.0.0.1	10.0.0 2	OUTPUT, POSTROUTING
Пакет от локальной программы, назначен на то же самое устройство	127.0 0.1	127.0.0.1	OUTPUT, POSTROUTING, (затем PREROUTING, INPUT, когда пакет снова попадает на вход через петлевой интерфейс)

Напомним, что когда пакет активирует цепочку, `iptables` исполняет таблицы внутри этой цепочки (конкретные правила в каждой таблице) в следующем порядке:

1. Raw.
2. Mangle.
3. NAT.
4. Filter.

Большинство цепочек не содержит все типы таблиц, однако порядок исполнения таблиц сохраняется. Такая конструкция выбрана для того, чтобы избежать избыточности. Например, таблица Raw предназначена для модификации пакетов перед их заходом на `iptables` и имеет только PREROUTING- и OUTPUT-цепочки — в соответствии со схемой потоков пакетов в Netfilter. Как соотносятся таблицы и цепочки, показывает табл. 2.7.

Таблица 2.7. Таблицы `iptables` (столбцы) и цепочки (колонки)

	Raw	Mangle	NAT	Filter
PREROUTING	x	x		x
INPUT	x		x	x
FORWARD	x		x	x
OUTPUT	x	x	x	x
POSTROUTING	x		x	

Получить список цепочек, входящих в соответствующую таблицу, можно командой `iptables -L -t <table>`:

```
$ iptables -L -t filter
Chain INPUT (policy ACCEPT)
target prot opt source         destination

Chain FORWARD (policy ACCEPT)
target prot opt source         destination

Chain OUTPUT (policy ACCEPT)
target prot opt source         destination
```

При использовании таблицы NAT надо учитывать следующее: DNAT может исполняться в цепочках `PREROUTING` или `OUTPUT`, а SNAT исполняется только в `INPUT` или `POSTROUTING`.

Для примера предположим, что у нас есть входящий пакет с назначением на наш хост. Тогда порядок исполнения будет таким.

- PREROUTING:
  - Raw.
  - Mangle.
  - NAT.
- INPUT:
  - Mangle.
  - NAT.
  - Filter.

Теперь, когда мы рассмотрели перехваты Netfilter, цепочки и таблицы, посмотрим на потоковую диаграмму обработки пакетов в `iptables` (рис. 2.5).

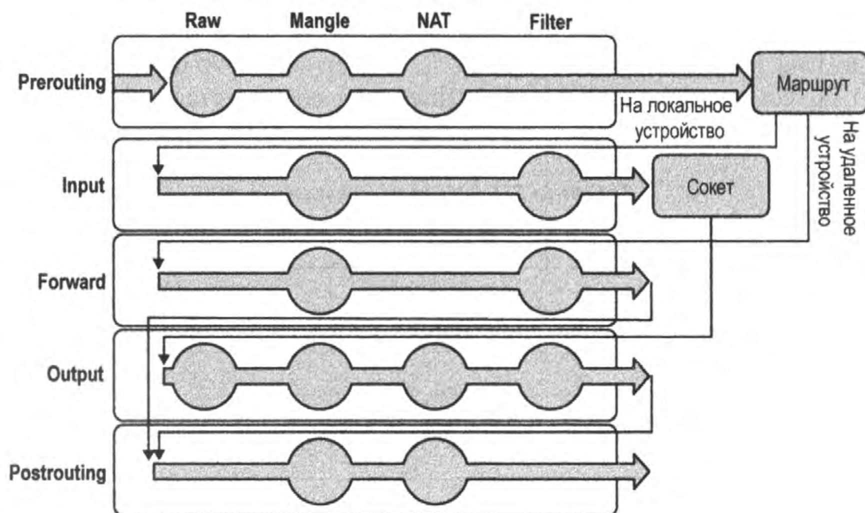


Рис. 2.5. Прохождение пакетов через таблицы и цепочки программы `iptables`. Кружок обозначает комбинацию таблица-перехват, присутствующую в `iptables`

Все правила iptables относятся к таблице и цепочке, их возможные комбинации показаны на потокой диаграмме в виде кружков. iptables исполняет цепочки (и правила в них по порядку) в зависимости от порядка активируемых пакетом перехватов Netfilter. Для заданной цепочки, программа iptables исполняет ее в каждой таблице, где эта цепочка присутствует (отметим, что некоторые комбинации цепочка-таблица не существуют, например Filter/POSTROUTING). Если мы проследим за прохождением пакета, исходящего от локального хоста, то увидим следующие исполняемые комбинации таблица-цепочка по порядку:

1. Raw/OUTPUT.
2. Mangle/OUTPUT.
3. NAT/OUTPUT.
4. Filter/OUTPUT.
5. Mangle/POSTROUTING.
6. NAT/POSTROUTING.

Подцепочки

Рассмотренные выше цепочки являются *цепочками верхнего уровня* (уровень точек входа). Пользователи могут определить свои собственные *подцепочки* и работать с ними с помощью JUMP. iptables производит вычисление состояний по этим цепочкам таким же образом — действие за действием, пока не будет достигнута конечная точка. Пользовательские цепочки могут быть полезны для объединения действий в группы с целью их повторного использования в разных контекстах (подобно тому, как в программах организуются процедуры-функции). Группировка правил в цепочках может оказывать существенное влияние на производительность. Программа iptables выполняет десятки, сотни и тысячи операторов if для каждого пакета, покидающего вашу систему или вошедшего в него. Все это имеет измеряемый эффект на загрузку ЦПУ, пропускную способность сети и время обработки пакета. Хорошо организованная последовательность цепочек уменьшает избыточную нагрузку путем исключения лишних проверок или действий. Несмотря на это, в сервисах с многими подами производительность iptables все еще недостаточна, что является проблемой для Kubernetes и делает более привлекательными решения, которые минимизируют использование iptables, как, например, IPVS или eBPF.

Посмотрим, как в примере 2.6 создается новая цепочка.

Пример 2.6. Цепочка iptables для брандмауэра SSH

```
# Создать цепочку с именем incoming-ssh
$ iptables -N incoming-ssh

# Разрешить пакеты с заданных IP-адресов.
$ iptables -A incoming-ssh -s 10.0.0.1 -j ACCEPT
$ iptables -A incoming-ssh -s 10.0.0.2 -j ACCEPT
```

```
# Регистрация пакета
$ iptables -A incoming-ssh -j LOG --log-level info --log-prefix "ssh-failure"

# Сбрасывать пакеты со всех остальных IP-адресов.
$ iptables -A incoming-ssh -j DROP

# Вычислять состояние по цепочке incoming-ssh,
# если пакет является входящим TCP-пакетом, адресованным на порт 22.
$ iptables -A INPUT -p tcp --dport 22 -j incoming-ssh
```

В примере создается новая цепочка, `incoming-ssh`, которая задействуется для любого TCP-пакета, направленного на порт 22. Цепочка разрешает вход пакетов от двух указанных IP-адресов, а пакеты с других адресов регистрируются и сбрасываются.

Цепочки с условиями фильтрации по умолчанию сбрасывают пакет, если он не соответствует ни одному из условий. Если условия по умолчанию явно не заданы, то цепочки выполняют по умолчанию ACCEPT. Командой iptables `-P <chain> <target>` можно задать условия по умолчанию.

Правила *iptables*

Правила состоят из двух частей — условие и действие (target). В условиях задаются определенные параметры, если они совпадают с параметрами пакета, то выполняется действие. Если параметры пакета не соответствуют заданным, то `iptables` переходит к проверке следующего правила.

Условия в правиле нужны для проверки пакета на соответствие определенным критериям, например имеет ли пакет заданный адрес источника. Важно запомнить порядок выполнения операций при обработке таблиц/цепочек, поскольку предшествующая операция может модифицировать пакет, сбросить его или сразу отвергнуть на входе. В табл. 2.8 перечисляются некоторые часто используемые условия.

Таблица 2.8. *Некоторые часто используемые критерии выбора в* `iptables`

Для чего используется	Флаги	Описание
Источник	`-s`, `--src`, `--source`	Пакеты с адресом источника, совпадающим с заданным
Назначение	`-d`, `--dest`, `--destination`	Пакеты с адресом назначения, совпадающим с заданным
Протокол	`-p`, `--protocol`	Пакеты с протоколом, совпадающим с заданным
Входной интерфейс	`-i`, `--in-interface`	Пакеты, входящие через заданный интерфейс
Интерфейс выхода	`-o`, `--out-interface`	Пакеты, покидающие систему через заданный интерфейс

Таблица 2.8 (окончание)

Для чего используется	Флаги	Описание
Состояние	`-m state -state <states>`	Пакеты от соединений, состояние которых совпадает с одним из указанных через запятую. Используются состояния Conntrack (NEW, ESTABLISHED, RELATED, INVALID)

Опции -m или -match позволяют программе iptables использовать расширения для формулирования условий. Расширения могут покрывать широкий диапазон условий — от задания нескольких портов в одном правиле (мультипорт) до сложных взаимодействий в eBPF. Подробную информацию можно получить командой man iptables-extension.

Существуют два типа действий — прекращающие и непрекращающие. Если действие задано как прекращающее, то iptables прекращает проверку следующих действий в цепочке, т. е. это действие представляет собой окончательное решение. Непрекращающие действия позволяют iptables продолжить движение по цепочке. ACCEPT, DROP, REJECT, RETURN — это все прекращающие действия. Обратите внимание, что ACCEPT и RETURN являются прекращающими только *внутри своей цепочки*. То есть если пакет доходит до ACCEPT в подцепочке, то обработка возвращается в родительскую цепочку, которая, в свою очередь, в принципе может сбросить пакет или отказать в его приеме. В примере 2.7 показан набор правил, согласно которым пакеты с назначением на порт 80 отвергаются, хотя в одной точке выполняется ACCEPT. Для простоты мы сократили выдачу отдельных команд.

Пример 2.7. Последовательность правил, в результате выполнения которых некоторые из предварительно принятых пакетов будут отброшены

```
$ iptables -L --line-numbers
Chain INPUT (policy ACCEPT)
num target prot opt source      destination
1 accept-all all - anywhere     anywhere
2 REJECT tcp - anywhere         anywhere
   tcp dpt:80 reject-with icmp-port-unreachable

Chain accept-all (1 references)
num target    prot opt source   destination
1             all - anywhere    anywhere
```

В табл. 2.9 приводятся широко используемые типы действий и их описание.

Действие относится как к таблице, так и к цепочке, контролирующим, когда (или если) iptables выполнит указанное действие для данного пакета. А теперь обобщим все, что мы узнали, и посмотрим, как команды iptables используются на практике.

Таблица 2.9. *Стандартные действия (target) программы* `itables`

Тип действия	В каких таблицах используется	Описание
AUDIT	все	Записывает данные о принятых, сброшенных или отклоненных пакетах
ACCEPT	Filter	Позволяет дальнейшую обработку пакета без каких-либо модификаций в нем
DNAT	NAT	Изменяет адрес назначения
DROPs	Filter	Сбрасывает пакет. Для внешнего наблюдателя все выглядит так, как будто пакет никогда не был получен
JUMP	все	Переход на выполнение другой цепочки. Как только выполнение этой цепочки завершится, выполнение родительской цепочки возобновится
LOG	все	Регистрация содержимого пакета, используется соответствующая функция ядра
MARK	все	Устанавливает для пакета специальный маркер — целое число, которое используется как идентификатор в Netfilter. Это число может также использоваться в других блоках `iptables`, непосредственно в пакет оно не записывается
MASQUERADE	NAT	Изменяет адрес источника пакета, заменяя его на адрес заданного сетевого интерфейса Действие, аналогичное SNAT, но не требует заранее известного IP-адреса устройства
REJECT	Filter	Отклоняет пакет и посылает уведомление о причине
RETURN	все	Прекращает обработку текущей цепочки (или подцепочки). Отметим, что это *не есть* прекращающее действие, и если присутствует родительская цепочка, то обработка переходит на нее
SNAT	NAT	Изменяет адрес источника пакета, заменяя его фиксированным адресом. См. также `MASQUERADE`

Практическое применение *iptables*

Цепочки `iptables` можно показать с помощью команды `iptables -L`:

```
$ iptables -L
Chain INPUT (policy ACCEPT)
target    prot opt source                        destination

Chain FORWARD (policy ACCEPT)
target   prot opt source                         destination

Chain OUTPUT (policy ACCEPT)
target   prot opt source                         destination
```

Существует отдельная, но почти полностью идентичная программа, ip6tables, для работы с правилами в IPv6. Правила iptables и ip6tables — совершенно различные вещи. Например, сброс всех пакетов с назначением на TCP 0.0.0.0:22 в правилах iptables не запрещает соединения с TCP [::]:22 в ip6tables и наоборот.

В данном разделе для простоты мы будем использовать только iptables и адреса протокола IPv4.

Команда --line-numbers показывает для каждого правила его номер в цепочке. Это может быть полезным, если потребуется вставить или удалить правило. Команда -I <chain> <line> вставляет правило в строку с заданным номером перед предыдущим правилом в этой строке.

Стандартный формат команды для работы с правилами iptables :

```
iptables [-t table] {-A|-C|-D} chain rule-specification
```

где -A означает *append* (добавить), -C — *check* (проверить) и -D — *delete* (удалить).

Правила iptables не сохраняются при перезапуске системы. Утилита iptables имеет опции iptables -save и iptables -restore, которые могут использоваться в ручном или автоматическом режиме для запоминания и повторной загрузки правил. Большинство брандмауэров автоматически создают свои собственные правила iptables при каждом запуске системы.

Утилита iptables может маскировать соединения, представляя так, как будто пакеты пришли со своего собственного IP-адреса. Это полезно, если есть задача предъявить внешнему миру упрощенную схему системы. Например, направить трафик на известный хост, который действует как защитный бастион, или предоставить третьим лицам предсказуемый набор IP-адресов. Используя маскирование в Kubernetes, можно заставить поды использовать IP-адреса их узлов, несмотря на то, что поды имеют свои уникальные IP-адреса. Во многих конфигурациях к этому приходится прибегать, чтобы обеспечить передачу данных из кластера, в котором поды имеют внутренние IP-адреса, не предназначенные для прямого выхода в Интернет. MASQUERADE действует аналогично SNAT, он, однако, не требует, чтобы адрес --source-address был известен и указан заранее. Вместо этого используется адрес заданного интерфейса. Это немного менее производительно, чем SNAT, в случаях, когда новый адрес источника является статическим, поскольку функция iptables будет постоянно запрашивать адрес:

```
$iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
```

Программа iptables может выполнять балансировку нагрузки на уровне соединения, или, более точно, разветвление соединения. Эта процедура базируется на правилах DNAT и случайном выборе (чтобы избежать случаев, когда каждое соединение будет маршрутизироваться на первое действие (target) в DNAT):

```
$ iptables -t nat -A OUTPUT -p tcp -dport 80 -d $FRONT_IP statistic\
--mode random -probability 0.5 -j DNAT -to-destination$BACKEND_IP:80
$ iptables -t nat -A OUTPUT -p tcp -dport 80 -d $FRONT_IP \
-j DNAT --to-destination $BACKEND2_IP:80
```

В этом примере вероятность отправки на первый бэкенд равна 50%. В противном случае пакет переходит к следующему правилу, которое гарантирует отправку на второй бэкенд. Если добавляются еще бэкенды, то вероятности пересчитываются. Чтобы вероятность была одинаковой для всех бэкендов, n-й бэкенд должен иметь вероятность 1/n того, что соединение будет передано ему. Если бы было 3 бэкенда, то вероятности были бы 0.3 (с повтором), 0.5 и 1:

```
Chain KUBE-SVC-I7EAKVFJLYM7WH25 (1 references)

target  prot opt source destination
KUBE-SEP-LXP5RGXOX6SCIC6C all - anywhere           anywhere
    statistic mode random probability 0.25000000000
KUBE-SEP-XRJTEP3YTXUYFBMK all - anywhere           anywhere
    statistic mode random probability 0.33332999982
KUBE-SEP-OMZR4HWUSCJLN33U all - anywhere           anywhere
    statistic mode random probability 0.50000000000
KUBE-SEP-EELL7LVIDZU4CPY6 all - anywhere           anywhere
```

Когда Kubernetes использует iptables для балансировки нагрузки, он создает цепочку, как показано выше. Если посмотрите внимательнее, то увидите ошибки округления в одном из значений вероятности.

Использование разветвления на основе DNAT для балансировки нагрузки не всегда безопасно. Обратная связь по нагрузке от заданного бэкенда отсутствует, что приводит к ситуации, когда приложения из очереди назначаются на одно и то же соединение с одним и тем же бэкендом. Поскольку результат работы DNAT действует в течение всего срока жизни соединения, то в случае большого числа долгоживущих соединений многие нисходящие клиенты могут оказаться связанными с одним и тем же бэкендом, если этот бэкенд живет дольше других. Чтобы привести пример из Kubernetes, предположим, что сервис gRPC имеет только две реплики, а затем дополнительные реплики масштабируются. gRPC использует то же самое HTTP/2-соединение, так что клиенты, расположенные ниже (использующие сервис Kubernetes, а не балансировку нагрузки gRPC), будут оставаться связанными с первыми двумя репликами, искажая тем самым профиль распределения нагрузки между бэкендами gRPC. По этой причине многие разработчики используют «умный» клиент (например, задействуя балансировку нагрузки со стороны клиента), предусматривают периодическое переподключение к серверу и/или клиенту или переходят на инфраструктуру service mesh. Более подробно мы рассмотрим балансировку нагрузки в *главах 4 и 5*.

Хотя утилита `iptables` широко используется в Linux, в присутствии огромного количества правил она начинает работать медленно и к тому же предоставляет довольно ограниченный функционал в смысле регулирования нагрузки. В следующем разделе мы рассмотрим виртуальный сервер IPVS — альтернативную технологию, которая является более удобной для балансировки нагрузки.

IPVS

Виртуальный сервер IP (IPVS) — это балансировщик нагрузки 4-го уровня в Linux. Диаграмма на рис. 2.6 иллюстрирует роль IPVS в маршрутизации пакетов.

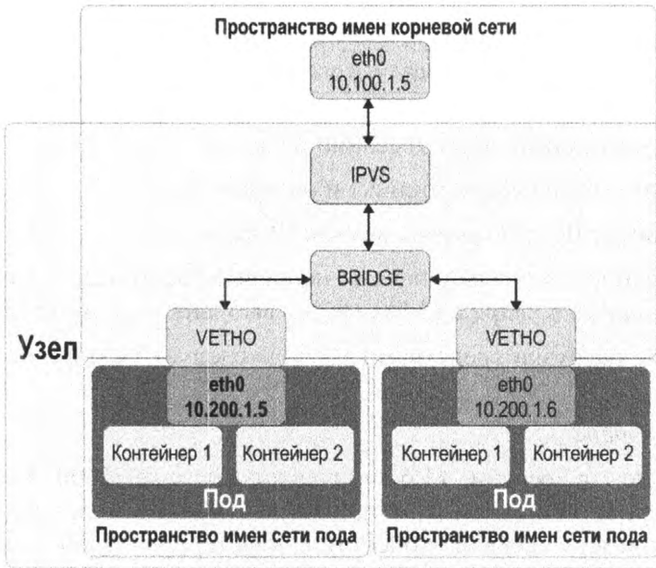


Рис. 2.6. Виртуальный сервер IPVS

Утилита iptables может выполнять несложную балансировку нагрузки 4-го уровня путем случайной маршрутизации соединений, при этом вероятности определяются через веса отдельных правил в DNAT. Виртуальный сервер IPVS поддерживает несколько режимов балансировки нагрузки (в отличие от всего одного в iptables), эти режимы перечислены в табл. 2.10. В зависимости от конфигурации IPVS и типов трафика это позволяет распределять нагрузку более эффективно по сравнению с iptables.

Таблица 2.10. *Режимы IPVS, поддерживаемые в Kubernetes*

Название	**Сокращение**	**Описание**
Round-robin	`rr`	Передает последовательно поступающие соединения на следующий хост в цикле. По сравнению со случайной маршрутизацией в iptables это увеличивает время между последовательными соединениями, передаваемыми на данный хост
Least connection	`lc`	Передает соединения на хост, имеющий в данный момент наименьшее число подключений
Destination hashing	`dh`	Передает соединения на заданный хост в зависимости от адреса назначения соединения
Source hashing	`sh`	Передает соединения на заданный хост в зависимости от исходного адреса соединения

Таблица 2.10 (окончание)

Название	Сокращение	Описание
Shortest expected delay	`sed`	Передает соединения на хост с наименьшим отношением число соединений / весовой коэффициент сервера
Never queue	`nq`	Передает соединение на любой хост без актуальных соединений, в противном случае использует принцип наименьшей ожидаемой задержки

IPVS поддерживает режимы переадресации пакетов:

- NAT модифицирует адреса источника и назначения.
- DR инкапсулирует IP-датаграммы внутри IP-датаграмм.
- IP-туннелирование направляет пакеты непосредственно на бэкенд-сервер путем замены MAC-адреса в кадре на MAC-адрес выбранного бэкенд-сервера.

При балансировке нагрузки средствами `iptables` следует учитывать следующие три фактора:

Число узлов в кластере

Хотя Kubernetes в версии v1.6 поддерживает уже 5000 узлов, kube-`proxy` с iptables остаются узким местом при развертывании кластера на 5000 узлов. Возьмем, к примеру, сервис NodePort в кластере на 5000 узлов: если у нас 2000 сервисов и каждый сервис имеет 10 подов, это потребует как минимум 20 000 записей в `iptables` на каждый работающий узел, что сильно влияет на работу ядра.

Время

Время добавления одного правила в условиях, когда есть 5000 сервисов (40 000 правил), составляет 11 минут. Для 20 000 сервисов (160 000 правил) это уже будет 5 часов.

Время ожидания

Существует время ожидания доступа к сервису (задержка маршрутизации): каждый пакет должен просматривать весь список в `iptables`, пока не будет найдено совпадающее условие. Есть также задержка в добавлении/удалении правил, что тоже связано с работой с длинными списками.

IPVS также поддерживает режим привязки сеанса (session affinity), который доступен через опции `Service.spec.sessionAffinity` и `Service.spec.sessionAffinityConfig`.

Повторные соединения, происходящие в пределах задаваемого в session affinity временного окна, будут направляться на один и тот же хост. В некоторых случаях это может быть полезным, например если хотят минимизировать так называемые промахи кеша (cache misses). Данный режим позволяет также сделать маршрутизацию зависимой от состояния: путем передачи соединений с одного и того же адреса на один и тот же хост, но такая «липкость» маршрутизации в Kubernetes не является особенно принципиальной, поскольку индивидуальные поды все время приходят и уходят.

Чтобы создать простейший балансировщик нагрузки для двух адресов назначения с одинаковыми весовыми коэффициентами, запустите команду `run ipvsadm -A -t <address> -s <mode>`.

Здесь `-A`, `-E` и `-D` используются, чтобы добавить, отредактировать и удалить виртуальные сервисы. Соответствующие опции, обозначаемые маленькими буквами `-a`, `-e` и `-d`, используются для добавления, редактирования и удаления бэкендов на хосте:

```
# ipvsadm -A -t 1.1.1.1:80 -s lc
# ipvsadm -a -t 1.1.1.1:80 -r 2.2.2.2 -m -w 100
# ipvsadm -a -t 1.1.1.1:80 -r 3.3.3.3 -m -w 100
```

Список хостов IPVS можно получить, используя параметр `-L`. Показан каждый виртуальный сервер (уникальный IP-адрес и порт) вместе со своими бэкендами:

```
# ipvsadm -L
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
-> RemoteAddress:Port  Forward Weight ActiveConn InActConn
TCP 1.1.1.1.80:http lc
-> 2.2.2.2:http         Masq    100      0        0
-> 3.3.3.3:http         Masq    100      0        0
```

Параметр `-L` поддерживает различные опции, например `--stats`, которая выдает дополнительную статистику по соединениям.

eBPF

eBPF — это система программирования, которая позволяет специальным автономным программам работать в ядре без постоянных переходов из ядра в пространство пользователя и обратно, как это происходит в Netfilter и `iptables`.

Прежде чем появилась eBPF, была просто BPF — Berkeley Packet Filter (берклиевская фильтрация пакетов). BPF является технологией, используемой ядром для анализа сетевого трафика. BPF поддерживает фильтрацию пакетов, что дает возможность процессу пространства пользователя формировать фильтр для задания свойств, по которым пакеты будут отбираться для анализа данным процессом. Одна из программ, где используется BPF, — это `tcpdump`, показанная на рис. 2.7. Если вы задали фильтр программе `tcpdump`, она скомпилирует его как BPF-программу и передаст ее BPF. Процедуры, реализованные в BPF, позже были перенесены в другие процессы и операции, выполняемые ядром.

eBPF-программа имеет прямой выход на системные вызовы. eBPF-программы могут отслеживать и блокировать системные вызовы без того, чтобы добавлять какие-либо перехваты ядра в программу пространства пользователя. Благодаря своим характеристикам система eBPF хорошо подходит для написания программ сетевого взаимодействия.

```
sudo tcpdump -n -i any
15:19:45.157203 IP 192.168.1.152.58128 > 192.168.1.140.8009: Flags [P.], seq 2927356224:2927356334, ack
15:19:45.157284 IP 192.168.1.152.58130 > 192.168.1.140.42593: Flags [P.], seq 2696314214:2696314324, ack
15:19:45.157351 IP 192.168.1.152.58129 > 192.168.1.135.8009: Flags [P.], seq 2157251184:2157251294, ack
15:19:45.170544 IP 192.168.1.140.8009 > 192.168.1.152.58128: Flags [P.], seq 1:111, ack 110, win 277, op
15:19:45.170547 IP 192.168.1.140.42593 > 192.168.1.152.58130: Flags [P.], seq 1:111, ack 110, win 277, o
15:19:45.170586 IP 192.168.1.152.58128 > 192.168.1.140.8009: Flags [.], ack 111, win 2046, options [nop,
15:19:45.170604 IP 192.168.1.152.58130 > 192.168.1.140.42593: Flags [.], ack 111, win 2046, options [nop
15:19:45.180631 IP 192.168.1.135.8009 > 192.168.1.152.58129: Flags [P.], seq 1:111, ack 110, win 277, op
15:19:45.180677 IP 192.168.1.152.58129 > 192.168.1.135.8009: Flags [.], ack 111, win 2046, options [nop,
15:19:45.265532 STP 802.1d, Config, Flags [none], bridge id 0fa0.10:93:97:6e:6b:62.8001, length 43
15:19:45.271989 IP 172.217.195.189.443 > 192.168.1.153.59925: UDP, length 40
15:19:45.273088 IP 172.217.195.189.443 > 192.168.1.153.59925: UDP, length 18
15:19:45.279601 IP 192.168.1.153.59925 > 172.217.195.189.443: UDP, length 29
15:19:45.280925 IP 192.168.1.153.59925 > 172.217.195.189.443: UDP, length 318
15:19:45.317150 IP 172.217.195.189.443 > 192.168.1.153.59925: UDP, length 21
15:19:45.318595 IP 172.217.195.189.443 > 192.168.1.153.59925: UDP, length 187
15:19:45.318597 IP 172.217.195.189.443 > 192.168.1.153.59925: UDP, length 38
15:19:45.326571 IP 192.168.1.153.59925 > 172.217.195.189.443: UDP, length 29
15:19:45.327238 IP 192.168.1.153.57942 > 172.217.9.130.443: UDP, length 23
15:19:45.349641 IP 192.168.1.153.60186 > 172.217.14.174.443: UDP, length 23
```

Рис. 2.7. Выдача программы `tcpdump`

Более подробные сведения о eBPF можно получить на специализированных сайтах.

Помимо фильтрации сокетов другими точками сопряжения с ядром являются следующие:

Kprobes

Динамическое отслеживание внутренних компонентов ядра.

Uprobes

Отслеживание пространства пользователя.

Tracepoints

Статическое отслеживание компонентов ядра. Запрограммировано в ядре разработчиками и является более устойчивым по сравнению с Kprobes, который может меняться в зависимости от версии ядра.

perf_events

Выборочный по времени контроль данных и событий.

XDP

Специализированные eBPF-программы, способные работать ниже пространства ядра, используются для доступа к драйверам, чтобы воздействовать непосредственно на пакеты.

Снова вернемся к рассмотрению примера с `tcpdump`. На рис. 2.8 показана упрощенная схема взаимодействия `tcpdump` с eBPF.

Предположим, мы запустили `tcpdump -i any`.

С помощью `Pcap_compile` строка компилируется в BPF-программу. Ядро будет использовать эту BPF-программу для фильтрации всех пакетов, проходящих через все указанные сетевые устройства, в нашем случае все с `-I`.

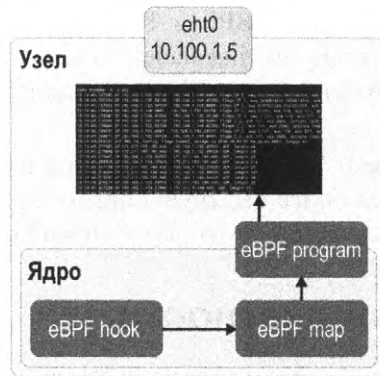


Рис. 2.8. Пример работы системы eBPF

Для `tcpdump` эти данные ядро предоставит через карту. Карты — это структуры данных, состоящие из пар величин, используемых BPF-программами для обмена данными.

Использование в Kubernetes системы eBPF имеет много преимуществ:

Производительность (хеш-таблица по сравнению со списком `iptables`)

Для каждого добавляемого в Kubernetes сервиса список просматриваемых правил `iptables` растет экспоненциально. Список правил должен заменяться каждый раз при добавлении нового правила. Требуется 5 часов времени, чтобы инсталировать 160 000 правил iptables, представляющих 20 000 сервисов Kubernetes.

Отслеживание

Используя BPF, мы можем собирать сетевую статистику на уровне пода и контейнера. BPF-фильтр для сокетов хорошо известен, чего не скажешь о BPF-фильтре для сокетов контрольных групп. Впервые появившийся в Linux 4.10, `cgroup-bpf` позволяет присоединять eBPF-программы к контрольным группам. После этого программа будет исполняться для всех пакетов, входящих в любой процесс в контрольной группе или покидающих его.

Контроль `Kuberctl exec` *с помощью eBPF*

С помощью eBPF вы можете присоединить программу, которая будет запоминать любые команды, исполняемые в сеансе `Kuberctl exec`, и передавать эти команды в программу пространства пользователя, которая следит за этими событиями.

Безопасность

Seccomp

Защищенный режим, который ограничивает разрешенные системные вызовы. Фильтры Seccomp можно написать средствами BPF.

Falco

Процедура безопасности с открытым кодом, которая использует eBPF.

Наиболее широкое использование eBPF в Kubernetes — это Cilium, реализация сетевого интерфейса CNI. Cilium предназначен заменить kube-proxy, который записывает правила iptables, чтобы связать IP-адрес сервиса и соответствующие ему поды.

Используя eBPF, Cilium может перехватывать пакеты и отправлять их непосредственно ядру, что обеспечивает более быструю обработку и позволяет балансировать нагрузку на 7-м уровне (приложения). Мы рассмотрим kube-proxy в *главе* 4.

Средства сетевой диагностики

Сетевая диагностика в Linux — это довольно сложная и разветвленная тема, которая сама по себе может стать отдельной книгой. В данном разделе мы познакомимся с некоторыми базовыми средствами диагностики и дадим рекомендации по их использованию (табл. 2.11 дает список этих средств). Информация данного раздела может рассматриваться как стартовая для перехода к средствам диагностики непосредственно Kubernetes. Более подробную информацию можно найти, как обычно, в мануалах, --help, и Интернете. Многие средства дублируют друг друга, поэтому может показаться, что изложение избыточно. Однако некоторые средства лучше работают в определенных условиях (например, многие диагностируют ошибки протокола TLS, но наиболее полную диагностическую информацию предоставляет OpenSSL). Какое именно средство и в каких условиях использовать, является, вообще говоря, вопросом личных предпочтений, привычки или просто наличия этих средств в системе.

Таблица 2.11. Средства отладки и условия их применения

Проблема	Инструменты
Проверка соединения	traceroute, ping, telnet, netcat
Сканирование порта	nmap
Проверка записей DNS	dig, команды, перечисленные в Проверке соединения
Проверка HTTP/1	cURL, telnet, netcat
Проверка HTTPS	OpenSSL, cURL
Проверка программ, активных в данный момент	netstat

Некоторые из описываемых программ могут отсутствовать в ваших дистрибутивах, но их легко получить по запросу. В выдачах мы иногда будем использовать #Truncated в местах, где мы опустили текст, чтобы избежать повторений и не делать примеры слишком длинными.

Безопасность

Прежде чем мы двинемся дальше, необходимо обсудить проблемы, связанные с безопасностью. Хакеры могут воспользоваться любым из перечисленных средств,

чтобы получить доступ к системе. Есть разные алгоритмы действий, но мы считаем лучшей практикой иметь на каждом устройстве только необходимый минимум инсталлированных программ.

Злоумышленники могут попробовать самостоятельно загрузить диагностические программы. В этом случае вы должны предусмотреть возможные препятствия для несанкционированного проникновения в вашу систему. Отсутствие заранее инсталлированных программ диагностики все-таки сужает поле действий для хакеров.

Права доступа к файлу в Linux содержат бит, называемый *setuid*, который иногда используется сетевыми утилитами. Если этот бит установлен, то исполнение соответствующего файла будет разрешено только *владельцу файла*, а не текущему пользователю. Об этом вам скажет присутствие в атрибутах файла `s` (а не `x`):

```
$ ls -la /etc/passwd
-rwsr-xr-x 1 root root 68208 May 28 2020 /usr/bin/passwd
```

Такое задание права доступа позволяет программе ограничить привилегии, предоставляемые внешнему пользователю (например, `passwd` использует данное свойство, чтобы позволить пользователю обновлять свой пароль, но не давая ему возможности осуществлять произвольные записи в файл паролей). Некоторые сетевые программы (`ping`, `nmap` и др.) используют бит setuid для посылки «сырых» пакетов, пакетов-анализаторов (sniffer) и пр. Если хакер загружает свою собственную копию программы и не может получить приоритет суперпользователя, то его возможности использовать данную программу будут ограничены, если она была инсталлирована в системе с установленным битом setuid.

ping

`ping` — простая утилита, которая рассылает пакеты `ECHO_REQUEST` протокола ICMP сетевым устройствам. Это стандартный способ проверки соединения между двумя хостами.

Межсетевой протокол управления сообщениями ICMP является протоколом 4-го уровня, как TCP и UDP. Сервисы Kubernetes поддерживают TCP и UDP, но не ICMP. То есть посылка `ping` на сервисы Kubernetes *не принесет результата*. Для проверки соединения с сервисом необходимо использовать `telnet` или программы более высокого уровня вроде `cURL`. Однако индивидуальные поды могут реагировать на ping — это зависит от конфигурации конкретной сети.

Брандмауэры и программы маршрутизации умеют распознавать ICMP-пакеты и могут быть сконфигурированы так, чтобы отфильтровывать или определенным образом маршрутизировать ICMP-пакеты. Обычно желательно (но не обязательно) иметь права доступа к ICMP-пакетам. Некоторые сетевые администраторы, сетевые программы и облачные провайдеры разрешают ICMP-пакеты по умолчанию.

Базовое использование `ping` — это просто `ping <адрес>`. Адресом может быть IP-адрес или домен. `ping` пошлет пакет, подождет и сообщит статус запроса, когда придет ответ или истечет заданное время.

По умолчанию ping будет посылать пакеты бесконечно, поэтому его надо прерывать вручную (например, через Ctrl-C). Заданием -c <число> можно заставить ping выполнять ограниченное число запросов и после этого завершать работу. По окончании работы ping выдаст протокол:

```
$ ping -c 2 k8s.io
PING k8s.io (34.107.204.206): 56 data bytes
64 bytes from 34.107.204.206: icmp_seq=0 ttl=117 time=12.665 ms
64 bytes from 34.107.204.206: icmp_seq=1 ttl=117 time=12.403 ms

--- k8s.io ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 12.403/12.534/12.665/0.131 ms
```

В табл. 2.12 приводятся стандартные опции ping.

Таблица 2.12. *Полезные опции программы ping*

Опция	Описание
с<число>	Посылает заданное количество пакетов Завершает работу, когда получен последний пакет или истекло время
-i<сек>	Задает интервал между последовательными посылками пакетов По умолчанию 1 сек. Очень маленькие значения не рекомендуются, поскольку ping может заполнить сеть
-o	Выход после получения 1 пакета. Эквивалентно –с 1
-S<адрес источника>	Использует для адресации пакета заданный адрес источник
-W<млсек>	Устанавливает интервал ожидания на получение пакета Если ping получит пакет после времени ожидания, он все равно учтет его в финальной выдаче

traceroute

traceroute показывает сетевой маршрут от одного хоста к другому. Это позволяет пользователям контролировать маршрут и исправлять ошибки (если маршрут оказался невыполненным).

traceroute посылает пакеты, задавая определенное время жизни. Напомним информацию из *главы 1*, что каждый хост, обрабатывающий пакет, уменьшает время жизни пакета (TTL) на 1, тем самым ограничивая число хостов, на которых может быть обработан запрос. Когда хост получает пакет и устанавливает его TTL в 0, он посылает ответный пакет TIME_EXCEEDED и сбрасывает исходный пакет. Ответный пакет TIME_EXCEEDED содержит адрес устройства, на котором произошло обнуление TTL. Начиная работу с TTL, равным 1, и увеличивая TTL на 1 для каждого пакета, traceroute способен получать ответы от каждого хоста на маршруте вплоть до адреса назначения.

traceroute отображает хосты строчка за строчкой, начиная с первого внешнего устройства. Каждая строчка содержит имя хоста (если есть), IP-адрес и время отклика:

```
$traceroute k8s.io
traceroute to k8s.io (34.107.204.206), 64 hops max, 52 byte packets
1 router (10.0.0.1)  8.061 ms  2.273 ms  1.576 ms
2 192.168.1.254 (192.168.1.254)  2.037 ms  1.856 ms  1.835 ms
3 adsl-71-145-208-1.dsl.austtx.sbcglobal.net (71.145.208.1)
4.675 ms 7.179 ms 9.930 ms
4 * * *
5 12.122.149.186 (12.122.149.186)  20.272 ms  8.142 ms  8.046 ms
6 sffca22crs.ip.att.net (12.122.3.70)  14.715 ms  8.257 ms  12.038 ms
7 12.122.163.61 (12.122.163.61)  5.057 ms  4.963 ms  5.004 ms
8 12.255.10.236 (12.255.10.236)  5.560 ms
12.255.10.238 (12.255.10.238)    6.396 ms
12.255.10.236 (12.255.10.236)    5.729 ms
9 * * *

8310 206.204.107.34.bc.googleusercontent.com (34.107.204.206)
64.473 ms 10.008 ms 9.321 ms
```

Если traceroute не получает отклик от очередной точки на маршруте в течение заданного времени, то он печатает *. Такая ситуация возникает, когда некоторые хосты отказываются посылать пакет TIME_EXCEEDED либо брандмауэры по маршруту препятствуют доставке пакета.

В табл. 2.13 приводятся стандартные опции traceroute.

Таблица 2.13. *Полезные опции программы* traceroute

Опция	**Синтаксис**	**Описание**
Первое TTL	-f <TTL>, -M <TTL>	Устанавливает TTL на первом IP (по умолчанию 1) Установка TTL в n приведет к тому, что traceroute не будет сообщать о первых n-1 точках на маршруте
Макс TTL	-m<TTL>	Устанавливает максимальное TTL, т. е. максимальное число хостов, через которое пройдет traceroute
Протокол	-P<протокол>	Посылает пакеты, соответствующие заданным протоколам (TCP, UDP, ICMP, есть другие опции), по умолчанию UDP
Исходный адрес	-s <адрес>	Задает IP-адрес источника для исходящих пакетов
Ожидание	-w <секунд>	Задает время ожидания отклика

dig

Команда dig предназначена для просмотра сервера доменных имен DNS. Вы можете использовать ее для формирования запросов к DNS из командной строки и для отображения результатов.

Общий формат команды dig [опции] <домен>. По умолчанию dig отображает записи CNAME, A и AAA:

```
$ dig kubernetes.io

; <<>> DiG 9.10.6 <<>> kubernetes.io
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 51818
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1452
;; QUESTION SECTION:
;kubernetes.io.            IN     A

;; ANSWER SECTION:
kubernetes.io.       960   IN     A  147.75.40.148

;; Query time: 12 msec
;; SERVER: 2600:1700:2800:7d4f:6238:e0ff:fe08:6a7b#53
(2600:1700:2800:7d4f:6238:e0ff:fe08:6a7b)

;; WHEN: Mon Jul 06 00:10:35 PDT 2020
;; MSG SIZE rcvd: 71
```

Чтобы отобразить определенный тип записи DNS, используйте dig <домен> <тип> (или dig -t <тип> <домен>). Данный формат dig является наиболее широко используемым:

```
$ dig kubernetes.io TXT

; <<>> DiG 9.10.6 <<>> -t TXT kubernetes.io
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 16443
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;kubernetes.io.              IN           TXT

;; ANSWER SECTION:
kubernetes.io.        3599   IN           TXT
"v=spf1 include:_spf.google.com ~all"
kubernetes.io.        3599   IN           TXT
"google-site-verification=oPORCoq9XU6CmaR7G_bV00CLmEz-wLGOL7SXpeEuTt8"
```

```
;; Query time: 49 msec
;; SERVER: 2600:1700:2800:7d4f:6238:e0ff:fe08:6a7b#53
(2600:1700:2800:7d4f:6238:e0ff:fe08:6a7b)
;; WHEN: Sat Aug 08 18:11:48 PDT 2020
;; MSG SIZE rcvd: 171
```

В табл. 2.14 приводятся опции команды dig.

Таблица 2.14. *Полезные опции команды* dig

Опция	Синтаксис	Описание
IPv4	-4	Использовать только IPv4
IPv6	-6	Использовать только IPv6
Адрес	-b<адрес> [#порт]	Задает адрес для запроса DNS. Можно включить в запрос порт (после знака #)
Порт	-p <порт>	Запрос по имени домена. Имя задается как аргумент в строке
Домен	-q <домен>	Запрос по имени домена. Имя задается как аргумент в строке
Тип	-t <тип>	Запрос по типу записи в DNS. Тип записи задается как аргумент

telnet

telnet — это одновременно сетевой протокол и инструмент использования данного протокола. Ранее telnet использовался для удаленного доступа наравне с SSH. Сейчас SSH стала основным средством удаленного доступа благодаря своей лучшей защищенности, но telnet по-прежнему остается чрезвычайно полезным инструментом для отладки серверных приложений, использующих протоколы на основе текста. Например, с помощью telnet вы можете соединиться с HTTP/1 сервером и вручную формировать на него запросы.

Базовый синтаксис telnet <адрес> <порт>. Эта команда устанавливает соединение и предлагает интерактивный интерфейс командной строки. Посылка команды осуществляется двойным нажатием Enter, что позволяет писать многострочные команды. Сеанс завершается нажатием Ctrl-J:

```
$ telnet kubernetes.io
Trying 147.75.40.148...
Connected to kubernetes.io.
Escape character is '^]'.
> HEAD / HTTP/1.1
> Host: kubernetes.io
>
HTTP/1.1 301 Moved Permanently
Cache-Control: public, max-age=0, must-revalidate
```

```
Content-Length: 0
Content-Type: text/plain
Date: Thu, 30 Jul 2020 01:23:53 GMT
Location: https://kubernetes.io/
Age: 2
Connection: keep-alive
Server: Netlify
X-NF-Request-ID: a48579f7-a045-4f13-af1a-eeaa69a81b2f23395499
```

Чтобы использовать все возможности `telnet`, вы должны хорошо понимать, как работает используемый вами протокол. `telnet` является классическим средством отладки серверов, использующих HTTP, HTTPS, POP3, IMAP и др.

nmap

`nmap` — это сканер портов, который дает вам возможность контролировать сервисы в вашей сети.

Общий синтаксис `nmap [опции] <назначение>`, где назначение — это домен, IP-адрес или IP CIDR. Устанавливаемые по умолчанию опции `nmap` позволяют быстро получить информацию по открытым на хосте портам:

```
$ nmap 1.2.3.4
Starting Nmap 7.80 ( https://nmap.org ) at 2020-07-29 20:14 PDT
Nmap scan report for my-host (1.2.3.4)
Host is up (0.011s latency).
Not shown: 997 closed ports
PORT STATE SERVICE

22/tcp   open   ssh
3000/tcp open   ppp
5432/tcp open   postgresql

Nmap done: 1 IP address (1 host up) scanned in 0.45 seconds
```

В данном примере nmap обнаруживает три открытых порта и идентифицирует работающие на этих портах сервисы.

Поскольку программа `nmap` способна быстро обнаружить и показать, какой сервис доступен с удаленной машины, она также может быть эффективным средством разведывания сервисов, которые *не должны быть* доступны. По этой причине `nmap` является любимым инструментом хакеров.

Программа `nmap` имеет огромное количество опций, которые определяют параметры сканирования и выдачи. В табл. 2.15 мы приводим несколько самых употребительных опций и настоятельно рекомендуем читателю обращаться к соответствующим руководствам по `nmap` на страницах man и help.

Таблица 2.15. Полезные опции команды nmap

Опция	Синтаксис	Описание
Дополнительная детекция	-A	Дает возможность определять ОС, версию и т. д.
Уменьшить детализацию	-d	Уменьшает детализацию вывода Использование множественных d (например, -dd) усиливает эффект
Увеличить детализацию	-v	Увеличивает детализацию. Множественные v (-vv) усиливают эффект

netstat

Команда netstat выдает детальную информацию о сети и соединениях:

```
$ netstat
Active internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0    164 my-host:ssh             laptop:50113            ESTABLISHED
tcp        0      0 my-host:50051           example-host:48760      ESTABLISHED
tcp6       0      0 2600:1700:2800:7d:54310 2600:1901:0:bae2::https TIME_WAIT
udp6       0      0 localhost:38125         localhost:38125         ESTABLISHED
Active UNIX domain sockets (w/o servers)
Proto RefCnt      Flags Type    State         I-Node  Path
unix  13          [ ]   DGRAM                 8451    /run/systemd/journal/dev-log
unix  2           [ ]   DGRAM                 8463    /run/systemd/journal/syslog
[Cut for brevity]
```

При вызове netstat без дополнительных аргументов мы получаем выдачу всех участвующих в соединениях сокетов. В нашем примере мы видим три TCP сокета, один сокет UDP и несколько сокетов UNIX. Информация включает адреса (IP-адрес и порт) на обеих сторонах соединения.

Вызов netstat с параметром -a позволяет увидеть все соединения, с параметром -l — только соединения в режиме прослушивания:

```
$ netstat -a
Active internet connections (servers and established)
Proto Recv-Q   Send-Q Local Address         Foreign Address       State
tcp        0        0 0.0.0.0:ssh           0.0.0.0:*             LISTEN
tcp        0        0 0.0.0.0:postgresql    0.0.0.0:*             LISTEN
tcp        0      172 my-host:ssh           laptop:50113          ESTABLISHED
[Content cut]
```

Обычно netstat используется для определения процесса, слушающего заданный порт. Для этого запускаем команду sudo netstat -lp, где -l относится к «прослушиванию» и p — к «программе». sudo может понадобиться для получения полной информации по программе. Выдача по опции -l показывает адрес, который прослушивает сервис (например, 0.0.0.0 или 127.0.0.1).

Можно использовать простые средства типа grep, чтобы получить детальную выдачу от netstat, если у нас есть конкретный объект поиска:

```
$ sudo netstat -lp | grep 3000
tcp6    0       0 [::]:3000     [::]:*  LISTEN    613/grafana-server
```

В табл. 2.16 приводятся стандартные опции netstat.

Таблица 2.16. *Полезные опции команды* netstat

Опция	Синтаксис	Описание
Показать все сокеты	-a	Показ всех сокетов, а не только открытых соединений
Показать статистику	-s	Показ сетевой статистики. По умолчанию netstat показывает статистику для всех протоколов
Показать слушающие сокеты	-l	Показ сокетов, находящихся в режиме прослушивания. Удобно для определения работающих сервисов
TCP	-t	Флаг задает показ только TCP данных. Может использоваться с другими флагами, например -lt показывает сокеты, слушающие TCP
UDP	-u	Флаг задает показ только UDP данных Может использоваться с другими флагами, например -lu показывает сокеты, слушающие UDP

netcat

Команда netcat — мощное средство для установления соединений, пересылки данных или мониторинга сокетов. Она может быть полезна для «ручного» запуска клиента или сервера, чтобы получить детальную информацию по их работе. В этом плане netcat похожа на telnet, хотя функционал netcat гораздо более широкий.

nc является сокращением для netcat в большинстве систем.

netcat может соединяться с сервером, будучи вызвана как netcat <адрес> <порт>. У команды есть интерактивный интерфейс stdin, который позволяет вручную вводить данные или передавать их в netcat. Очень похоже на telnet:

```
$ echo -e "GET / HTTP/1.1\nHost: localhost\n" > cmd
$ nc localhost 80 < cmd
HTTP/1.1 302 Found
Cache-Control: no-cache
Content-Type: text/html; charset=utf-8
[Content cut]
```

Openssl

Технология OpenSSL используется в подавляющем большинстве HTTPS-соединений. Основная часть работы OpenSSL падает на языковые привязки, но у библиотеки есть также интерфейс CLI для решения текущих задач и для отладки. openssl способна выполнять такие функции, как создание ключей и сертификатов, сертификатов электронной подписи и пр., в нашем случае наиболее важным представляется тестирование TLS/SSL-соединений. Вообще говоря, это могут выполнять и другие программы, включая описанные в данной главе, но по количеству опций и детализации диагностики openssl стоит на первом месте.

Стандартный формат команды openssl [подкоманда] [аргументы] [опции]. openssl имеет огромное число подкоманд (например, openssl rand позволяет генерировать псевдослучайные данные). Подкоманда list дает возможность получить список функций, имеется также опция поиска (например, openssl list --commands). Подробную информацию по отдельным подкомандам можно получить, задавая openssl <подкоманда> --help, или на странице руководства (man openssl -<подкоманда> или просто man <подкоманда>).

Команда openssl s_client -connect осуществляет соединение с сервером и выдает подробную информацию о серверном сертификате. Ниже приводится выдача:

```
openssl s_client -connect k8s.io:443
CONNECTED (00000003)
depth=2 O = Digital Signature Trust Co., CN = DST Root CA X3
verify return:1
depth=1 C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
verify return:1
depth=0 CN = k8s.io
verify return:1
---
Certificate chain
0 s:CN = k8s.io
i:C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
1 s:C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
i:O = Digital Signature Trust Co., CN = DST Root CA X3
---
Server certificate
-----BEGIN CERTIFICATE-----
[Content cut]
-----END CERTIFICATE-----
subject=CN = k8s.io
issuer=C = US, O = Let's Encrypt, CN = Let's Encrypt Authority X3
---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: RSA-PSS
Server Temp Key: X25519, 253 bits
```

```
SSL handshake has read 3915 bytes and written 378 bytes
Verification: OK
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 2048 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 0 (ok)
```

Если вы используете самоподписывающийся код CA, то можете выполнять для этого CAfile `<путь>`. Это позволит устанавливать и верифицировать соединения с таким сертификатом.

cURL

Команда cURL используется для передачи данных, поддерживает многие протоколы, в том числе HTTP и HTTPS.

`wget` **похожа по функциям на команду** `curl`. **Некоторые дистрибутивы предлагают ее, а не** `curl`.

Команды cURL имеют формат `curl [опции] <URL>`. cURL распечатывает содержимое URL, иногда вместе с соответствующими сообщениями, на `stdout`. По умолчанию генерируется запрос HTTP GET:

```
$ curl example.org
<!doctype html>
<html>
<head>
<title>Example Domain</title>
# Усечено
```

По умолчанию cURL не отслеживает переадресацию, как, например, HTTP 301 или обновления протокола.

Флаг -L (или --location) позволит выполнить следующее перенаправление:

```
$ curl kubernetes.io
Redirecting to https://kubernetes.io

$ curl -L kubernetes.io
<!doctype html><html lang=en class=no-js><head>
# Усечено
```

Используйте опцию `-X` для выполнения действий HTTP, например команда `curl -X DELETE foo/bar` выполняет запрос `DELETE`.

Передавать данные (для POST, PUT и пр.) можно несколькими способами:

- Через URL : `-d "key1=value1&key2=value2"`.
- JSON: `-d '{"key1":"value1", "key1":"value2"}'`.
- Как файл формата `-d @data.txt`.

Опция `-H` добавляет явный заголовок, хотя базовые заголовки типа `Content -Type` добавляются автоматически:

```
-H "Content-Type: application/x-www-form-urlencoded"
```

Приведем несколько примеров:

```
$ curl -d "key1=value1" -X PUT localhost:8080
$ curl -H "X-App-Auth: xyz" -d "key1=value1&key2=value2"
-X POST https://localhost:8080/demo
```

cURL иногда полезна при отладке деталей, связанных с TLS, но мы рекомендуем пользоваться более специализированными средствами типа `openssl`.

cURL может помочь при диагностике TLS-соединений. Как каждый уважающий себя браузер, cURL верифицирует цепочку сертификата, возвращаемую HTTP-сайтом, и проверяет CA-сертификаты хоста:

```
curl https://expired-tls-site
curl: (60) SSL certificate problem: certificate has expired
More details here: https://curl.haxx/se/docs/sslcerts.html

Curl failed to verify the legitimacy of the server and therefore could not establish
a secure connection to it. To learn more about this situation and how to fix it,
please visit the web page mentioned above.

 //Curl не получила подтверждение сертификата сервера и поэтому не смогла установить
с ним надежное соединение. Чтобы узнать больше о проблеме и путях ее решения,
обращайтесь на указанную выше веб-страницу
```

Как многие другие программы, cURL имеет опцию детальной выдачи `-v`, с помощью которой распечатывается подробная информация о запросе и ответе на него. Опция чрезвычайно полезна при отладке протоколов 7-го уровня, таких как HTTP:

```
$ curl https://expired-tls-site -v
*  Trying 1.2.3.4...
*  TCP_NODELAY set
*  Connected to expired-tls-site (1.2.3.4) port 443 (#0)
*  ALPN, offering h2

*  ALPN, offering http/1.1
*  successfully set certificate verify locations:
*   CAfile: /etc/ssl/cert.pem
   CApath: none
```

```
* TLSv1.2 (OUT), TLS handshake, Client hello (1):
* TLSv1.2 (IN), TLS handshake, Server hello (2):
* TLSv1.2 (IN), TLS handshake, Certificate (11):
* TLSv1.2 (OUT), TLS alert, certificate expired (557):
* SSL certificate problem: certificate has expired
* Closing connection 0
curl: (60) SSL certificate problem: certificate has expired
More details here: https://curl.haxx.se/docs/sslcerts.html
# Truncated
```

cURL имеет множество дополнительных опций, которых мы не коснулись, как, например, использование временных задержек, пользовательских CA сертификатов, пользовательского DNS и т. д.

Заключение

В данной главе мы рассмотрели функции поддержки сети в Linux. В первую очередь наше внимание было обращено на концепции, необходимые для понимания приложений Kubernetes, ограничений на конфигурацию кластеров и устранение проблем, связанных с реализацией сетей на базе Kubernetes. Приведенная информация отнюдь не является исчерпывающей, и вам могут потребоваться дополнительные источники.

В следующей главе мы перейдем к изучению контейнеров в Linux и тому, как контейнеры взаимодействуют с сетями.

ГЛАВА 3

Основы работы с контейнерами

После того как мы рассмотрели основы сетевых технологий и поддержку сети в ОС Linux, пришло время познакомиться с сетевыми операциями с контейнерами. Как и сами сети, контейнеры тоже имеют длинную историю. В данной главе мы расскажем о развитии концепции контейнеров, обсудим различные возможности их реализации и изучим имеющиеся конфигурации. В настоящее время промышленным стандартом считаются контейнеры проекта Docker. Поэтому мы рассмотрим сетевую модель Docker, объясним, чем модель CNI отличается от модели Docker и закончим главу примерами сетевых операций с контейнерами Docker.

Введение в контейнеры

В этом разделе мы обсудим эволюцию методов запуска приложений, которая привела в конце концов к появлению контейнеров. Некоторые пользователи обоснованно считают, что контейнеры — это не есть нечто реальное, они представляют собой просто еще одну абстракцию функций ядра операционной системы. Однако ограничиваясь только чисто технической стороной дела, мы не сможем понять базовые идеи этой технологии, что не будет способствовать решению сложных задач управления и развертывания прикладных программ.

Приложения

Старт приложений всегда сопровождался теми или иными проблемами. В настоящее время существует много способов запустить программу — на локальном компьютере, используя облачный сервис или как контейнеризованное приложение. Разработчики приложения и системные администраторы сталкиваются с необходимостью решать многие вопросы — такие как работа с разными версиями библиотек, развертывание приложений и учет их старых версий. Эти вопросы занимают разработчиков с незапамятных времен. И bash-скрипты, и инструменты развертывания имеют свои недостатки и узкие места. Каждая компания имеет свои собственные процедуры запуска приложений, под которые подстраиваются все разработчики этой организации.

Разделение функций, контроль доступа, поддержка устойчивости системы приводят к тому, что системные администраторы ограничивают доступ разработчиков к работающим приложениям. Кроме того, администраторы имеют дело с многими приложениями на одной и той же вычислительной машине и должны обеспечить их

общую эффективную работу. Это тоже нередко приводит к конфликтам между разработчиками, желающими реализовать еще одну программную функцию, и администраторами, обязанными следить за устойчивостью всей системы.

Операционная система общего назначения поддерживает столько типов приложений, сколько возможно, так что ее ядро включает в себя всевозможные драйверы, библиотеки протоколов и планировщики. На рис. 3.1 представлена одна вычислительная машина с одной операционной системой, но позволяющая развертывать приложения многими способами. Развертывание приложений — это задача, которую решают все организации.

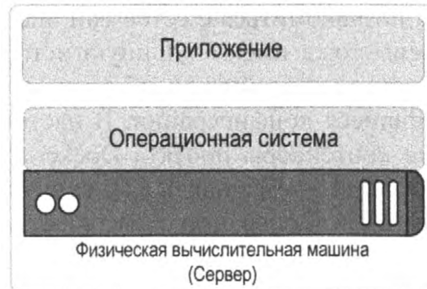


Рис. 3.1. Сервер приложений

С точки зрения сетевых технологий при наличии одной ОС есть один стек протоколов TCP/IP. Этот единственный стек может привести к проблемам, связанным с конфликтом портов на хосте. Системные администраторы размещают несколько приложений на одном и том же устройстве, чтобы повысить коэффициент его использования, поэтому каждое приложение должно иметь свой собственный порт. То есть теперь свои действия должны согласовывать разработчик, системный администратор и сетевой инженер. К заданиям по полному развертыванию приложений добавляется еще создание инструкции по устранению возможных проблем, и реализация всех требований по части ИТ. Гипервизоры позволяют увеличить эффективность вычислительной машины и снять проблемы, связанные с наличием одной ОС и одного стека сетевых протоколов.

Гипервизор

Гипервизор эмулирует ресурсы, ЦПУ и память хост-компьютера, создавая гостевые операционные системы, или виртуальные машины.

В 2001 г. фирма VMware выпустила свой гипервизор x86, ранние версии включали z-архитектуру от IBM и FreeBSD Jail. В 2003 г. вышел Xen, первый гипервизор с открытым кодом, в 2006-м появилась KVM (Kernel-based Virtual Machine) — виртуальная машина на основе ядра. Гипервизор дает возможность системным администраторам распределять ресурсы машины между несколькими гостевыми ОС, рис. 3.2 демонстрирует, как это делается. Такое совместное использование ресурсов повышает эффективность хост-компьютера, решая тем самым одну из задач, стоящих перед системными администраторами.

Рис. 3.2. Гипервизор

Также гипервизоры предоставляют каждому из разработчиков отдельный сетевой стек, снимая тем самым вопросы, связанные с возможным конфликтом портов в системах коллективного пользования. Например, группа А разработчиков запускает свое приложение на порту 8080, а группа В тоже может использовать порт 8080, т. к. теперь каждое приложение может работать с отдельной гостевой учетной записью в ОС, имеющей собственный сетевой стек. Библиотеки, развертывание приложений и другие вопросы по-прежнему остаются в ведении разработчика. Как им запустить свое приложение и обеспечить его ресурсами, сохраняя при этом эффективность, обеспечиваемую гипервизором и виртуальными машинами? Ответ на этот вопрос привел к появлению концепции контейнера.

Контейнеры

Глядя на рис. 3.3, мы видим преимущества контейнеризованных приложений — каждый контейнер является независимым. Разработчики приложений могут использовать для запуска своих программ все, что нужно, не беспокоясь при этом о библиотеках или операционных системах хост-компьютера. У каждого контейнера есть свой собственный сетевой стек. Контейнер дает возможность разработчикам упаковывать и развертывать приложения, эффективно используя при этом ресурсы хост-компьютера.

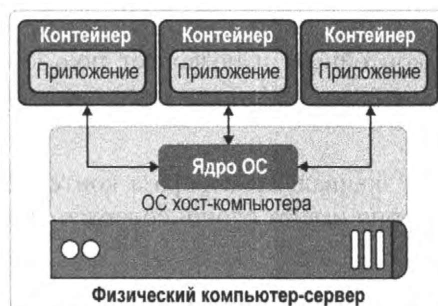


Рис. 3.3. Контейнеры, работающие под управлением операционной системы хост-компьютера

Каждая технология — это история изменений, конкурирующих решений и инноваций, и контейнеры здесь не исключение. Ниже мы приводим список терминов, которые применительно к контейнерам сначала могут показаться не совсем понятными. Мы перечисляем различия между средами запуска (runtime) контейнеров, обсуждаем функциональность каждой среды и показываем, как они соотносятся с Kubernetes. Функциональность среды запуска контейнера разбивается на «низкоуровневую» и «высокоуровневую».

Контейнер

Запущенный образ контейнера.

Образ

Образ контейнера — это файл, загруженный с сервера регистрации и используемый локально как точка сборки при запуске контейнера.

Движок контейнера (container engine)

Программное обеспечение, имеющее интерфейс командной строки. Принимает запросы пользователя и осуществляет загрузку образов и запуск контейнера.

Среда запуска контейнера (container runtime)

Низкоуровневое программное обеспечение внутри container engine, которое обеспечивает работу контейнера.

Базовый образ

Начальная точка формирования образа контейнера. Чтобы уменьшить размер и сложность построенного контейнера, пользователь может начать с базового образа и затем нарастить на его основе требуемые функции.

Слои образа

Репозитории обычно считаются набором образов (контейнеров), но, вообще говоря, они состоят из одного или нескольких слоев. Слои образов в репозитории связаны отношениями наследования. Каждый слой образа отличается изменениями между ним и родительским слоем.

Формат образа

Движки контейнера имеют свои собственные форматы образа, такие как LXD, RKT и Docker.

Реестр

Реестр хранит образы контейнера и позволяет пользователю скачивать, загружать и модифицировать образы.

Репозиторий

Репозитории могут быть эквивалентом образа контейнера. Важное отличие состоит в том, что репозитории имеют слои и содержат метаданные образа.

Тег

Тег — это определяемое пользователем имя для различных версий образа контейнера.

Хост контейнера

Хост контейнера — это система, в которой с помощью движка контейнера запущен контейнер.

Оркестрация контейнеров

Это то, чем занимается Kubernetes! Он динамически распределяет рабочую нагрузку контейнера в кластере контейнерных хостов.

Контрольные группы cgroups и пространства имен — это примитивы Linux для создания контейнеров, они будут рассмотрены в следующем разделе.

Пример низкоуровневой функциональности — создание контрольных групп и пространств имен, что является минимальным набором для запуска контейнера. Однако разработчикам требуется, как правило, больше функций. Им необходимо формировать, тестировать и развертывать контейнеры — это называется «высокоуровневой» функциональностью. Каждая среда запуска контейнера предлагает различные уровни функциональности. Рассмотрим их подробно:

Низкоуровневые функции среды запуска контейнера

- Создание контейнеров.
- Запуск контейнеров.

Высокоуровневые функции среды запуска контейнера

- Форматирование образов контейнеров.
- Формирование образов контейнеров.
- Управление образами контейнеров.
- Управление экземплярами контейнеров (container instance).
- Общее использование образов контейнера.

На следующих страницах мы рассмотрим среды запуска, в которых реализованы все эти функции. Каждый из проектов имеет свои сильные и слабые стороны в способах реализации высокоуровневых и низкоуровневых функций. Некоторые из них имеют сегодня только историческое значение, т. к. больше не существуют или влились в другие проекты.

Низкоуровневые среды запуска контейнеров

- *LXC*

  API на C для Linux.
- *runC*

  Интерфейс CLI для OCI-совместимых контейнеров.

Высокоуровневые среды запуска контейнеров

- *containerd*

  Среда запуска контейнеров, выделившаяся из Docker, проект CNCF.

- *CRI-O*

  Интерфейс среды запуска контейнеров, использующий спецификации открытой контейнерной инициативы (OCI), вышел из «инкубатора» CNCF.

- *Docker*

  Контейнерная платформа программ с открытым кодом.

- *lmctfy*

  Платформа контейнеризации Google.

- *rkt*

  Спецификация контейнера CoreOS.

OCI

Проект OCI предоставляет общие минимальные открытые стандарты и спецификации для технологии контейнеров.

Разработка формальной спецификации для сред запуска и форматов образа контейнера позволяет развертывать контейнеры на всех основных платформах и операционных системах. В основе проекта OCI лежат три концепции:

Структурность

Средства управления контейнерами должны предоставлять четкие и ясные интерфейсы. Они не должны быть связаны с конкретными проектами, клиентами или стандартами и должны работать на всех платформах.

Децентрализация

Спецификации форматов и сред запуска должны быть разработаны всем сообществом, а не одной организацией. Другой целью проекта OCI является независимая программная реализация инструментов для запуска одного и того же контейнера.

Минимализм

Проект OCI стремится выполнять задания качественно, с минимальными ресурсами и добиваться устойчивых решений, он приветствует инновации и эксперимент.

Проект Docker предоставил шаблон для формата и среду запуска. Он также предоставил OCI программный код для базовой реализации. Docker взял содержимое библиотеки libcontainer, сделал ее независимой от Docker и передал проекту OCI. Среда запуска — это runC, который можно найти на GitHub.

Давайте рассмотрим несколько более ранних проектов по контейнеризации и их возможности. Мы закончим раздел как раз на том месте, где в дело вступает Kubernetes со своими средами запуска и средствами управления.

LXC

Контейнеры Linux, LXC, появились в 2008 г. LXC объединяет контрольные группы и пространства имен, чтобы предоставить изолированную среду для запуска приложений. Целью LXC является создание среды, максимально приближенной к стандартному Linux, без необходимости создавать отдельное ядро системы. LXC состоит из отдельных компонент: библиотеки `liblxc`, несколько интерфейсов программирования для разных языков (Python версий 2 и 3, Lua, Go, Ruby, Haskell), набора стандартных инструментов разработки и шаблонов контейнеров.

runC

runC — наиболее широко используемая среда запуска контейнеров, первоначально разработанная в рамках проекта Docker, а позднее выделенная как отдельная программа и библиотека. runC представляет собой интерфейс командной строки, реализованный в соответствии со спецификациями OCI, для запуска приложений, упакованных в соответствии с форматом OCI. runC использует `libcontainer`, являющейся той же библиотекой, которая обеспечивает работу движка Docker. Движок Docker ранее версии 1.11 использовался для управления объемами, сетями, контейнерами, образами и пр. В настоящее время архитектура Docker включает в себя различные компоненты, runC обеспечивает следующие функции:

- Полная поддержка пространства имен Linux, включая пространство имен пользователя.
- Изначальная поддержка всех процедур безопасности, имеющихся в Linux:
- SELinux, AppArmor, seccomp, контрольных групп, исключения привилегий, `pivot_root`, сброс UID/GID и т. д.
- Изначальная поддержка контейнеров Windows 10.
- Запланированная изначальная поддержка всех аппаратных средств экосистемы.
- Специфицированный конфигурационный формат, разрабатываемый OCI под управлением Linux Foundation.

containerd

containerd — это высокоуровневая среда запуска, выделившаяся из Docker. containerd является фоновым сервисом, который предлагает услуги API для различных сред запуска контейнеров и ОС. containerd состоит из различных компонент, обеспечивающих ему высокоуровневую функциональность. containerd представляет собой сервис для Linux и Windows, который управляет полным жизненным циклом контейнера, передачей образа, хранением, развертыванием контейнера, соединением с сетью. CLI-интерфейс для containerd — это ctr, с его помощью можно выполнять разработку и отладку. Компонент containerd-shim предназначен для работы с контейнерами без демона. Для процесса контейнера он является родительским и выполняет определенные действия. containerd дает возможность средам запуска, например runC, завершать работу после запуска контейнера. Тем самым мы избав-

ляемся от долгоживущих процессов при работе контейнера. Другим преимуществом этого подхода является то, что стандартный I/O и другие файловые дескрипторы остаются открытыми для контейнера, когда процессы containerd и Docker умирают. Если shim не запущен, тогда родительская сторона канала межпроцессного взаимодействия была бы закрыта и контейнер прекратил бы работу. containerd-shim позволяет также посылать сообщение о прекращении работы контейнера на высокоуровневые инструменты типа Docker, при этом реальный родительский процесс контейнера этого не делает.

lmctfy

Google заявил lmctfy в качестве контейнерной технологии на базе Linux в 2013 г. lmctfy представляет собой высокоуровневую среду запуска контейнеров, имеющую функции создания и удаления контейнеров, однако в настоящее время она больше не поддерживается и была передана libcontainer, который сегодня стал containerd (самостоятельное решение, реализующее исполняемую среду для запуска контейнеров). lmctfy предоставляла конфигурацию под управлением API, так что разработчики могли не беспокоиться о деталях реализации контрольных групп и пространства имен.

rkt

Проект rkt стартовал как CoreOS в 2014, будучи альтернативой Docker. Он написан на языке Go, использует поды в качестве базовой вычислительной единицы и предлагает автономную среду для приложений. Исходный формат образа в rkt был App Container Image (ACI), определяемый в спецификации App Container, позднее предпочтение было отдано поддержке формата OCI. Есть поддержка спецификации CNI, могут запускаться образы Docker и OCI. Проект rkt был архивирован в феврале 2020 г. его разработчиками.

Docker

Проект Docker, запущенный в 2013 г., смог решить многие проблемы, с которыми сталкивались разработчики контейнеризованных приложений. Он предлагает функциональность, которая требуется разработчикам для создания, поддержки и развертывания контейнеров:

- Форматирование образов контейнеров.
- Формирование образов контейнеров.
- Управление образами контейнеров.
- Управление экземплярами контейнеров.
- Совместное использование образов контейнеров.
- Запуск контейнеров.

На рис. 3.4 показана архитектура движка (engine) Docker и его компоненты. Docker начинал как монолитное приложение, объединяя все имеющиеся функции в едином

бинарном файле, известном как *Docker engine*. Движок содержал также клиента Docker, или CLI-интерфейс, который позволял разработчикам формировать, запускать и развертывать контейнеры и образы. Сервер Docker работает как процесс-демон и управляет объемами данных и сетями для запущенных контейнеров. Клиент взаимодействует с сервером посредством Docker API.

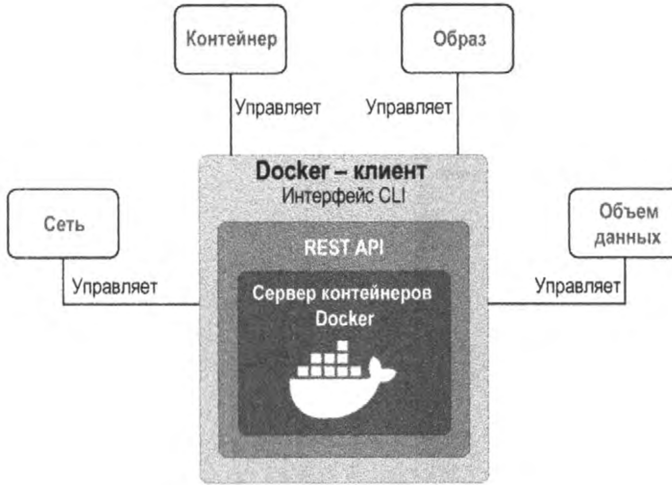


Рис. 3.4. Движок Docker

В последующие годы Docker разбил этот монолит на отдельные компоненты. Чтобы запустить контейнер, движок Docker создает образ и передает его containerd. containerd вызывает containerd-shim, который использует runC для запуска контейнера. После того как контейнер запущен, containerd-shim прекращает работу среды запуска (в данном случае runC). Таким образом мы можем запускать контейнеры без процесса-демона, поскольку теперь для работы контейнера нам не нужны долгоживущие процессы.

Docker позволяет разделить обязанности между разработчиками приложений и системными администраторами: разработчики прилагают усилия, чтобы написать свои приложения, а системные администраторы думают над тем, как их развернуть. Docker обеспечивает быстрый цикл разработки: чтобы протестировать новые версии Go для наших веб-приложений, мы можем обновить базовый образ и прогнать с ним все тесты. Docker гарантирует переносимость приложений, которые одинаково работают на локальном компьютере, в облаке или в любом другом дата-центре. Девизом Docker является создавать, поставлять и запускать программы во всех имеющихся средах. Новый контейнер на хост-компьютере быстро масштабируется, и в нем могут быть запущены еще больше приложений, что повышает эффективность использования вычислительной техники.

CRI-O

CRI-O является реализацией интерфейса Kubernetes CRI на базе OCI — набора спецификаций, определяющих работу среды запуска контейнеров. Red Hat начала про-

ект CRI в 2016 г. и в 2019 г. передала его CNCF. CRI — это плагин-интерфейс, который позволяет Kubernetes через его сервис `kubelet` взаимодействовать с любой средой запуска контейнеров, которая совместима с CRI-интерфейсом. Разработка CRI-O началась в 2016 г. после того, как проект Kubernetes представил CRI; версия CRI –O 1.0 вышла в 2017 г. CRI –O — это облегченная среда запуска CRI, выполненная как Kubernetes-специфичная высокоуровневая среда запуска, построенная с использованием gRPC и Protobuf на сокете UNIX. Рис. 3.5 показывает, каким образом CRI встроен в архитектуру Kubernetes. CRI-O добавляет устойчивости в проект Kubernetes, он прошел все тесты Kubernetes.

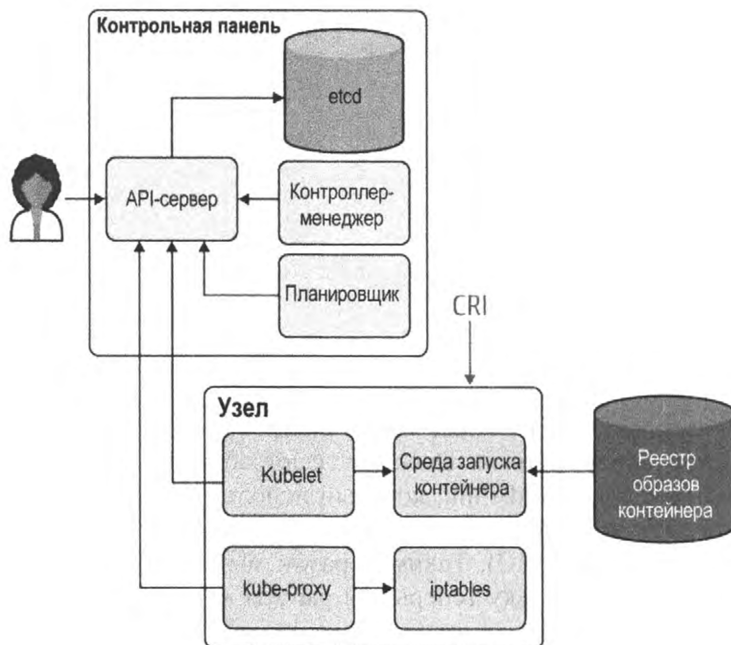


Рис. 3.5. Интерфейс CRI в Kubernetes

Контейнеры сегодня — это продукт технологий, инноваций и деятельности компаний разработчиков. Выше мы изложили краткую историю развития контейнеров. На сегодняшний день устоявшимся мнением является то, что контейнерные технологии продолжат развиваться как OCI-проект с открытым кодом для самых различных применений. Kubernetes также внес свой вклад в это развитие, предоставив интерфейс CRI-O. Понимание компонентов контейнера является первостепенной задачей для администраторов, развертывающих контейнеры в своих системах, и для разработчиков контейнеризованных приложений. Важность этого понимания демонстрирует, например, Kubernetes 1.20, в котором было признано нежелательным поддерживать dockershim. Хотя среда запуска Docker, использующая dockershim, для администраторов больше не входит в список рекомендаций, разработчики все еще могут использовать Docker для создания и запуска OCI-совместимых контейнеров.

Первой реализацией CRI-интерфейса был dockershim, который предоставлял уровень абстракции перед движком Docker.

А теперь перейдем к более подробному рассмотрению контейнерных технологий.

Примитивы контейнеров

Независимо от того, используется Docker или containerd, запуск и управление контейнером осуществляется утилитой runC. В данном разделе мы рассмотрим, чем runC помогает программисту с точки зрения работы с контейнером. Каждый из наших контейнеров имеет в основе примитивы Linux, известные как *контрольные группы* (cgroups) и *пространства имен*. Рис. 3.6 показывает пример этих структурных единиц. Контрольные группы управляют для контейнера доступом к ресурсам ядра, а пространства имен — это часть ресурсов, управляемых отдельно от корневого пространства имен, т. е. хост-компьютера.

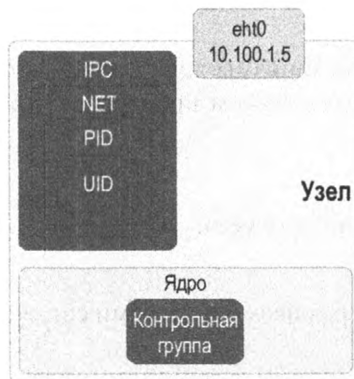


Рис. 3.6. Пространства имен и контрольные группы

Чтобы лучше понять эти концепции, рассмотрим контрольные группы и пространства имен подробнее.

Контрольные группы

Если говорить коротко, то *контрольная группа* — это механизм ядра Linux, который отвечает за использование ресурсов. Впервые представленные в версии Linux 2.6.24, контрольные группы дают возможность администраторам управлять ресурсами процессора и памяти для различных процессов. Механизм контрольных групп предоставляется через псевдофайловые системы и программно реализован в ядре в модуле cgroups. Следующие подсистемы ядра поддерживают различные контрольные группы:

ЦПУ

Процессу гарантируется, что доля ресурса ЦПУ, предоставляемая другим процессам, будет минимальной.

Память

Задает лимиты на память для процесса.

Дисковый I/O

Это и другие устройства управляются через контрольную группу для устройств.

Сеть

Поддерживается подсистемой `net_cls` и маркирует пакеты, покидающие контрольную группу.

С помощью утилиты `lscgroup` можно получить список всех контрольных групп, присутствующих на данный момент в системе.

runC генерирует контрольные группы для контейнера при его создании. Контрольная группа задает объем ресурсов, которые может использовать контейнер, в то время как пространства имен определяют, какие внутренние процессы может видеть контейнер.

Пространства имен

Пространства имен — это механизмы ядра Linux, которые изолируют и виртуализируют системные ресурсы для набора процессов. Ниже приведены примеры виртуализированных ресурсов:

Пространство имен PID

ID процессов, для изоляции процесса.

Сетевое пространство имен

Управляет сетевыми интерфейсами и стеками сетевых протоколов.

Пространство имен IPC

Управляет доступом к ресурсам межпроцессного взаимодействия (IPC).

Пространство имен точек монтирования

Управляет точками монтирования файловой системы.

Пространство имен UTS

Режим разделения времени в UNIX, позволяет отдельным хостам иметь различные имена хоста и домена для различных процессов.

Пространство имен UID

ID пользователя, изолирует процессы с назначениями различным пользователям и группам.

Пользователи процесса и ID-групп могут различаться внутри и вне пространства имен пользователя. Процесс может иметь ID непривилегированного пользователя вне пространства имен пользователя и в то же время иметь UID = 0 внутри пользовательского пространства имен в контейнере. Процесс имеет приоритет корня при исполнении внутри пространства имен пользователя, но является непривилегированным при действиях вне пространства имен.

Пример 3.1 показывает, как просмотреть пространства имен для процесса. Вся информация по процессу находится в каталоге /proc файловой системы Linux. PID 1-го пространства имен есть 4026531836, листинг всех пространств имен показывает, что ID-пространства имен процессов совпадают.

Пример 3.1. Пространства имен отдельного процесса

```
vagrant@ubuntu-xenial:~$ sudo ps -p 1 -o pid,pidns
PID       PIDNS
1 4026531836

vagrant@ubuntu-xenial:~$ sudo ls -l /proc/1/ns
total 0
lrwxrwxrwx 1 root root 0  Dec 12  20:41  cgroup -> cgroup:[4026531835]

lrwxrwxrwx 1 root root 0  Dec 12  20:41   ipc -> ipc:[4026531839]
lrwxrwxrwx 1 root root 0  Dec 12  20:41   mnt -> mnt:[4026531840]
lrwxrwxrwx 1 root root 0  Dec 12  20:41   net -> net:[4026531957]
lrwxrwxrwx 1 root root 0  Dec 12  20:41   pid -> pid:[4026531836]
lrwxrwxrwx 1 root root 0  Dec 12  20:41   user -> user:[4026531837]
lrwxrwxrwx 1 root root 0  Dec 12  20:41   uts -> uts:[4026531838]
```

Рис. 3.7 показывает, что рассмотренные примитивы Linux позволяют разработчикам приложений контролировать свои приложения и управлять ими независимо от хост-компьютеров и других приложений, запущенных как в контейнерах, так и непосредственно на компьютере.

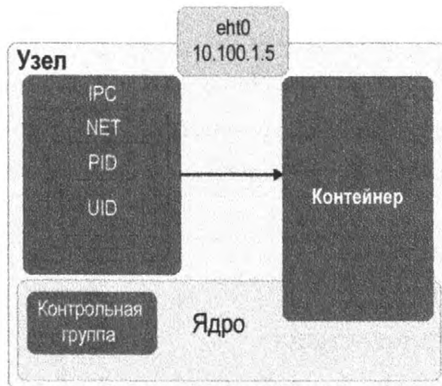


Рис. 3.7. Объединенная функциональность контрольных групп и пространств имен

Приводимые ниже примеры используют сборку Ubuntu 16.04 LTS Xenial Xerus. Если вы хотите сами выполнить их на вашей системе, то более подробную информацию можно найти в библиотеке программ данной книги. Библиотека содержит инструменты и конфигурации для запуска виртуальной машины Ubuntu и контейнеров Docker. Начнем с задания и тестирования наших пространств имен.

Задание пространств имен

На рис. 3.8 показана конфигурация контейнерной сети. На следующих страницах мы разберем все команды Linux, которые выполняет низкоуровневая среда запуска при создании контейнерной сети.

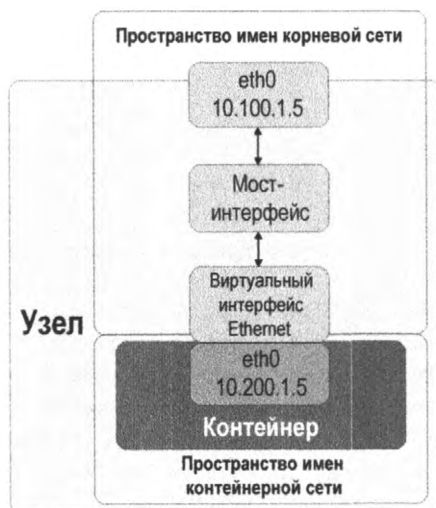


Рис. 3.8. Пространства имен корневой сети и контейнерной сети

Чтобы создать сетевую конфигурацию, показанную на рис. 3.8, необходимо выполнить следующие шаги:

1. Создать хост с пространством имен корневой сети.
2. Создать пространство имен новой сети.
3. Создать пару виртуальных интерфейсов `veth`.
4. Включить одну сторону пары `veth` в пространство имен новой сети.
5. Присвоить адрес одной стороне пары `veth` внутри пространства имен новой сети.
6. Создать интерфейс типа мост.
7. Присвоить адрес мосту.
8. Связать мост с интерфейсом хоста.
9. Связать одну сторону пары `veth` с мостом.
10. Готово!

Далее перечисляются команды Linux, необходимые для создания сетевого пространства имен, моста, пары `veth` и для связки всего в единое целое:

```
$ echo 1 > /proc/sys/net/ipv4/ip_forward
$ sudo ip netns add net1
$ sudo ip link add veth0 type veth peer name veth1
```

```
$ sudo ip link set veth1 netns net1
$ sudo ip link add veth0 type veth peer name veth1
$ sudo ip netns exec net1 ip addr add 192.168.1.101/24 dev veth1
$ sudo ip netns exec net1 ip link set dev veth1 up
$ sudo ip link add br0 type bridge
$ sudo ip link set dev br0 up
$ sudo ip link set enp0s3 master br0
$ sudo ip link set veth0 master br0
$ sudo ip netns exec net1 ip route add default via 192.168.1.100
```

Команда Linux `ip` служит для задания и управления сетевым пространством имен.

Подробную информацию о команде ip можно получить на странице ее мануала (man).

В примере 3.2 мы использовали Vagrant и VirtualBox, чтобы инсталлировать Ubuntu для проведения наших тестов.

Пример 3.2. Виртуальная машина Ubuntu

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/xenial64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box 'ubuntu/xenial64' version '20200904.0.0' is up to date...
==> default: Setting the name of the VM:
advanced_networking_code_examples_default_1600085275588_55198
==> default: Clearing any previously set network interfaces...
==> default: Available bridged network interfaces:
1) en12: USB 10/100 /1000LAN
2) en5: USB Ethernet(?)
3) en0: Wi-Fi (Wireless)
4) llw0
5) en11: USB 10/100/1000 LAN 2
6) en4: Thunderbolt 4
7) en1: Thunderbolt 1
8) en2: Thunderbolt 2
9) en3: Thunderbolt 3
==> default: When choosing an interface, it is usually the one that is
==> default: being used to connect to the internet.
==> default:
    default: Which interface should the network bridge to? 1
==> default: Preparing network interfaces based on configuration...

    default: Adapter 1: nat
    default: Adapter 2: bridged
```

```
==> default: Forwarding ports...
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default: Warning: Connection reset. Retrying...
    default:
    default: Vagrant insecure key detected. Vagrant will automatically replace
    default: this with a newly generated keypair for better security.
    default:
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if it's present...
    default: Key inserted! Disconnecting and reconnecting using new SSH key...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Configuring and enabling network interfaces...
==> default: Mounting shared folders...
    default: /vagrant =>
    /Users/strongjz/Documents/code/advanced_networking_code_examples
```

Обращайтесь к библиотеке программ данной книги и файлу Vagrantfile, чтобы воспроизвести пример.

Vagrant — это менеджер локальной виртуальной машины, созданный HashiCorp

После того как Vagrant загружает нашу виртуальную машину, мы можем использовать его для удаленного доступа (ssh) к этой машине:

```
$± |master U:2 ?:2 ✗| → vagrant ssh
Welcome to Ubuntu 16.04.7 LTS (GNU/Linux 4.4.0-189-generic x86_64)

vagrant@ubuntu-xenial:~$
```

Переадресация IP (IP forwarding) — это способность операционной системы принимать входящие сетевые пакеты на одном интерфейсе, распознавать их как предназначенные для другого интерфейса и соответственно отправлять их в эту другую сеть. Если опция IP forwarding включена, то Linux компьютер может получать входящие пакеты и производить их переадресацию. Для Linux компьютера, действующего как обычный хост, эта опция не нужна, поскольку он генерирует и получает IP трафик для своих собственных целей. По умолчанию опция выключена, но на нашей виртуальной машине мы ее включим:

```
vagrant@ubuntu-xenial:~$ sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 0
vagrant@ubuntu-xenial:~$ sudo echo 1 > /proc/sys/net/ipv4/ip_forward
vagrant@ubuntu-xenial:~$ sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 1
```

Установив виртуальную машину Ubuntu, мы видим, что у нас нет никаких дополнительных пространств имен для сети, так что создадим такое пространство:

```
vagrant@ubuntu-xenial:~$ sudo ip netns list
```

Команда `ip netns` позволяет нам контролировать пространства имен на сервере. Пространство создается простой операцией `ip netns add net1`:

```
vagrant@ubuntu-xenial:~$ sudo ip netns add net1
```

Продвигаясь по примеру дальше, мы можем увидеть сетевое пространство имен, которое мы только что создали:

```
vagrant@ubuntu-xenial:~$ sudo ip netns list
net1
```

Теперь, когда у нас есть новое сетевое пространство имен для нашего контейнера, нам понадобится пара `veth` для взаимодействия между пространством имен корневой сети и пространством имен контейнерной сети net1.

Команда `ip` снова поможет нам создать пару `veth`. Напомним материал из *главы 2*, что интерфейсы `veth` создаются парами и работают как канал между сетевыми пространствами имен, т. е. пакеты с одного конца автоматически переадресуются на другой.

```
vagrant@ubuntu-xenial:~$ sudo ip link add veth0 type veth peer name veth1
```

Интерфейсы 4 и 5 — это пары veth в консольной выдаче. Мы также можем увидеть, что с чем соединено, `veth1@veth0` и `veth1@veth0`.

Команда `ip link list` подтверждает создание пары veth:

```
vagrant@ubuntu-xenial: ~$ ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state
UNKNOWN mode DEFAULT group default qlen 1
      link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
pfifo_fast state UP mode DEFAULT group default qlen 1000
      link/ether 02:8f:67:5f:07:a5 brd ff:ff:ff:ff:ff:ff
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
pfifo_fast state UP mode DEFAULT group default qlen 1000
      link/ether 08:00:27:0f:4e:0d brd ff:ff:ff:ff:ff:ff
4: veth1@veth0: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc
noop state DOWN mode DEFAULT group default qlen 1000
      link/ether 72:e4:03:03:c1:96 brd ff:ff:ff:ff:ff:ff
```

```
5: veth0@veth1: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc
noop state DOWN mode DEFAULT group default qlen 1000
      link/ether 26:1a:7f:2c:d4:48 brd ff:ff:ff:ff:ff:ff
vagrant@ubuntu-xenial:~$
```

Теперь перенесем veth1 в пространство имен новой сети, созданное ранее:

```
vagrant@ubuntu-xenial:~$ sudo ip link set veth1 netns net1
```

Команда ip netns exec позволяет нам проверить конфигурацию пространства имен сети. Выдача подтверждает, что veth1 находится в пространстве имен сети net:

```
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1 ip link list
4: veth1@if5: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state
DOWN mode DEFAULT group default qlen 1000
    link/ether 72:e4:03:03:c1:96 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Пространства имен представляют собой полностью автономные стеки протоколов TCP/IP в ядре Linux. Будучи новым интерфейсом и находясь в новом сетевом пространстве имен, интерфейс veth требует присвоения ему IP-адреса, чтобы переносить пакеты из пространства имен net1 в корневое пространство имен и за пределы хоста:

```
vagrant@ubuntu-xenial:~$ sudo ip netns exec
net1 ip addr add 192.168.1.100/24 dev veth1
```

Что же касается сетевых интерфейсов хост-компьютера, то их требуется «включить»:

```
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1 ip link set dev veth1 up
```

Состояние теперь перешло в LOWERLAYERDOWN. Флаг NO-CARRIER указывает в правильном направлении. Требуется кабель для подсоединения Ethernet, наша первичная пара veth тоже еще не активна. Интерфейс veth1 уже активирован, и ему присвоен адрес, но реально он еще «не подключен к розетке»:

```
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1 ip link list veth1
4: veth1@if5: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500
qdisc noqueue state LOWERLAYERDOWN mode DEFAULT
group default qlen 1000 link/ether 72:e4:03:03:c1:96
brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Рассмотрим сторону veth0 парного интерфейса:

```
vagrant@ubuntu-xenial:~$ sudo ip link set dev veth0 up
vagrant@ubuntu-xenial:~$ sudo ip link list
5: veth0@if4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
qdisc noqueue state UP mode DEFAULT group default qlen 1000
link/ether 26:1a:7f:2c:d4:48 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Теперь парный интерфейс veth внутри пространства имен net1 активирован:

```
vagrant@ubuntu-xenial:~$ sudo ip netns exec net1 ip link list
4: veth1@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
```

```
qdisc noqueue state UP mode DEFAULT group default qlen 1000
link/ether 72:e4:03:03:c1:96 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Обе стороны пары veth сообщают, что они активны; теперь нам требуется связать сторону veth корневого пространства имен с мост-интерфейсом. Внимательно следите, чтобы был выбран именно интерфейс, с которым вы работаете, — в данной случае это enp0s8; в других ситуациях он может быть другим:

```
vagrant@ubuntu-xenial:~$ sudo ip link add br0 type bridge
vagrant@ubuntu-xenial:~$ sudo ip link set dev br0 up
vagrant@ubuntu-xenial:~$ sudo ip link set enp0s8 master br0
vagrant@ubuntu-xenial:~$ sudo ip link set veth0 master br0
```

Мы видим, что enp0s8 и veth0 сообщают, что входят в состав bridge-интерфейса br0, master br0 находится в активном состоянии.

Проверим соединение с нашим сетевым пространством имен:

```
vagrant@ubuntu-xenial:~$ ping 192.168.1.100 -c 4
PING 192.168.1.100 (192.168.1.100) 56(84) bytes of data.
From 192.168.1.10 icmp_seq=1 Destination Host Unreachable
From 192.168.1.10 icmp_seq=2 Destination Host Unreachable
From 192.168.1.10 icmp_seq=3 Destination Host Unreachable
From 192.168.1.10 icmp_seq=4 Destination Host Unreachable

--- 192.168.1.100 ping statistics ---
4 packets transmitted, 0 received, +4 errors, 100% packet loss, time 6043ms
```

Наше новое сетевое пространство имен не имеет маршрута по умолчанию, поэтому оно не знает, куда отправлять пакеты в ответ на ping запросы:

```
$ sudo ip netns exec net1
Ip route add default via 192.168.1.100
$ sudo ip netns exec net1 ip r
default via 192.168.1.100 dev veth1
192.168.1.0/24 dev veth1 proto kernel scope link src 192.168.1.100
```

Теперь попробуем еще раз:

```
$ ping 192.168.2.100 -c 4
PING 192.168.2.100 (192.168.2.100) 56(84) bytes of data.
64 bytes from 192.168.2.100: icmp_seq=1 ttl=64 time=0.018 ms
64 bytes from 192.168.2.100: icmp_seq=2 ttl=64 time=0.028 ms
64 bytes from 192.168.2.100: icmp_seq=3 ttl=64 time=0.036 ms
64 bytes from 192.168.2.100: icmp_seq=4 ttl=64 time=0.043 ms

--- 192.168.2.100 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2997ms

$ ping 192.168.2.101 -c 4
PING 192.168.2.101 (192.168.2.101) 56(84) bytes of data.
64 bytes from 192.168.2.101: icmp_seq=1 ttl=64 time=0.016 ms
```

```
64 bytes from 192.168.2.101: icmp_seq=2 ttl=64 time=0.017 ms
64 bytes from 192.168.2.101: icmp_seq=3 ttl=64 time=0.016 ms
64 bytes from 192.168.2.101: icmp_seq=4 ttl=64 time=0.021 ms

--- 192.168.2.101 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 2997ms
rtt min/avg/max/mdev = 0.016/0.017/0.021/0.004 ms
```

Получилось! Мы создали мост-интерфейс и парные интерфейсы veth, перенесли один из них в новое сетевое пространство имен и проверили возможность подключения. В примере 3.3 мы снова перечисляем все команды, которые потребовались для выполнения задачи.

Пример 3.3. Создание сетевого пространства

```
$ echo 1 > /proc/sys/net/ipv4/ip_forward
$ sudo ip netns add net1
$ sudo ip link add veth0 type veth peer name veth1
$ sudo ip link set veth1 netns net1
$ sudo ip link add veth0 type veth peer name veth1
$ sudo ip netns exec net1 ip addr add 192.168.1.101/24 dev veth1
$ sudo ip netns exec net1 ip link set dev veth1 up
$ sudo ip link add br0 type bridge
$ sudo ip link set dev br0 up
$ sudo ip link set enp0s3 master br0
$ sudo ip link set veth0 master br0
$ sudo ip netns exec net1 ip route add default via 192.168.1.100
```

Для разработчика, не знакомого со всеми этими командами, потребуется очень многое запоминать, — и очень легко выбрать не тот способ! Например, если мост-интерфейс окажется некорректно сконфигурирован, то сетевые петли могут вывести из строя часть сети. Это ситуация, которую системные администраторы всеми силами стремятся предотвратить, поэтому они не позволяют разработчикам производить такого рода изменения в системе. К счастью, контейнеры избавляют разработчиков от необходимости знать все вышеперечисленные команды и тем самым убирают опасения системных администраторов, что разработчики неумелым использованием команд обрушат сеть.

Указанные команды требуются для сетевого пространства имен при каждом создании и удалении контейнера. Создание пространства имен в примере 3.3 — это как раз то, что выполняется при запуске контейнера. Эту работу берет на себя Docker. Проект CNI предлагает единый для всех систем стандарт создания сети. CNI, так же как и OCI, — это возможность для разработчиков стандартизировать и расставлять приоритеты для задач по управлению жизненным циклом контейнера. В следующем разделе мы рассмотрим CNI.

Основы сетей контейнеров

В предыдущем разделе мы рассмотрели команды, требующиеся, чтобы создать пространства имен для наших сетевых коммуникаций. Здесь мы познакомимся с тем, как это реализовано в Docker. Выше мы использовали только режим моста, однако есть и другие способы реализации сетей контейнеров. В данном разделе мы развернем несколько контейнеров Docker и посмотрим их работу с сетью, а также объясним, каким образом контейнеры взаимодействуют с внешним хостом и друг с другом.

Начнем с рассмотрения ряда сетевых «режимов», используемых при работе с контейнерами:

Автономный (None)

Никакая внешняя сеть не нарушает работу сети контейнера. Используйте данный режим, если контейнерам не требуется доступ к сети.

Мост

В режиме «мост» контейнер работает в частной сети, внутренней по отношению к хосту. Взаимодействие с другими контейнерами сети открыто. Взаимодействие с сервисами, внешними по отношению к хосту, происходит через трансляцию сетевых адресов (NAT) перед тем, как покинуть хост. Режим мост является сетевым режимом по умолчанию, если опция --net не задана.

Хост

В этом режиме контейнер имеет тот же IP-адрес и пространство имен, что и хост. Процессы, работающие внутри контейнера, имеют такие же сетевые возможности, как и сервисы, запущенные непосредственно на хост-компьютере. Данный режим выгоден, если контейнеру требуется доступ к сетевым ресурсам хоста. Но при этом контейнер теряет свои преимущества, связанные с изоляцией сети. При развертывании такого контейнера необходимо следить за портами сервисов, запущенных на сетевом узле.

Режим «хост» работает только на хост-компьютерах под управлением Linux. Docker Desktop для Mac и Windows или Docker EE для Windows Server данный режим не поддерживают.

Macvlan

Macvlan использует родительский интерфейс. Это может быть хост-интерфейс, такой как eth0, суб-интерфейс или даже связанный хост-адаптер, который объединяет Ethernet-интерфейсы в один логический интерфейс. Как и все сети Docker, сети Macvlan изолированы друг от друга, есть взаимодействие внутри каждой сети, но не между сетями. Режим Macvlan позволяет физическому интерфейсу иметь множественные MAC и IP-адреса через использование субинтерфейсов. Сеть Macvlan бывает 4 типов: частная, VEPA, мост (по умолчанию в Docker) и транзитная (passthrough). Если используется мост, то для внешнего соединения требуется NAT. Поскольку в Macvlan хосты напрямую связываются

с физической сетью, внешние соединения могут осуществляться с помощью тех же DHCP сервера и коммутатора, что и используемые хостом.

Большинство облачных провайдеров блокируют сети Macvlan. Требуется разрешение администратора на доступ к сетевому оборудованию.

IPvlan

Сеть IPvlan похожа на Macvlan, но с существенным отличием: IPvlan не присваивает MAC-адреса созданным суб-интерфейсам. Все суб-интерфейсы имеют MAC-адрес родительского интерфейса, но используют различные IP-адреса. IPvlan имеет два режима: L2 и L3. Режим L2, или 2-го уровня, аналогичен режиму моста в сети Macvlan. Режим L3, или 3-го уровня, работает как устройство 3-го уровня между суб-интерфейсами и родительским интерфейсом.

Оверлей

Режим позволяет распространить сеть на другие хосты в кластере контейнеров. Оверлейная сеть виртуально располагается поверх базовых/физических сетей. Несколько проектов с открытым программным кодом занимаются созданием таких сетей, мы рассмотрим их подробнее ниже в данной главе.

Пользовательский

Пользовательский сетевой мост — это то же самое, что сетевой мост, но используется мост, специально созданный для данного контейнера. В качестве примера приведем контейнер, запущенный в bridge-сети базы данных. Контейнер может иметь интерфейс с мостом по умолчанию и мостом базы данных, что позволяет ему при необходимости взаимодействовать с обеими сетями.

В сети контейнеров адрес и конфигурация сети могут быть общими для нескольких контейнеров. Это дает возможность изолировать процессы внутри контейнера, когда в каждом контейнере запускается один сервис, но эти сервисы могут взаимодействовать друг с другом через адрес `127.0.0.1`.

Чтобы протестировать все перечисленные режимы, нам по-прежнему понадобится хост Vagrant Ubuntu, но на этот раз с установленным Docker. Docker для Mac и Windows не поддерживает режим хостовой сети, поэтому используем Linux. Вы можете воспользоваться машиной из примера 1.1 или взять версию Docker Vagrant из библиотеки программ данной книги. Инсталляция Ubuntu Docker происходит, как показано ниже — если вы хотите выполнить ее вручную:

```
$ vagrant up
Bringing machine 'default' up with 'virtualbox' provider...
==> default: Importing base box 'ubuntu/xenial64'...
==> default: Matching MAC address for NAT networking...
==> default: Checking if box
    'ubuntu/xenial64' version '20200904.0.0' is up to date...
==> default: Setting the name of the VM:
    advanced_networking_code_examples_default_1600085275588_55198
```

```
==> default: Clearing any previously set network interfaces...
==> default: Available bridged network interfaces:
    1) en12: USB 10/100 /1000LAN
    2) en5: USB Ethernet(?)
    3) en0: Wi-Fi (Wireless)
    4) llw0
    5) en11: USB 10/100/1000 LAN 2
    6) en4: Thunderbolt 4
    7) en1: Thunderbolt 1
    8) en2: Thunderbolt 2
    9) en3: Thunderbolt 3
==> default: When choosing an interface, it is usually the one that is
==> default: being used to connect to the internet.
==> default:
    default: Which interface should the network bridge to? 1
==> default: Preparing network interfaces based on configuration...
    default: Adapter 1: nat
    default: Adapter 2: bridged
==> default: Forwarding ports...
    default: 22 (guest) => 2222 (host) (adapter 1)
==> default: Running 'pre-boot' VM customizations...
==> default: Booting VM...
==> default: Waiting for machine to boot. This may take a few minutes...
    default: SSH address: 127.0.0.1:2222
    default: SSH username: vagrant
    default: SSH auth method: private key
    default: Warning: Connection reset. Retrying...
    default:
    default: Vagrant insecure key detected. Vagrant will automatically replace
    default: this with a newly generated keypair for better security.
    default:
    default: Inserting generated public key within guest...
    default: Removing insecure key from the guest if it's present...
    default: Key inserted! Disconnecting and reconnecting using new SSH key...
==> default: Machine booted and ready!
==> default: Checking for guest additions in VM...
==> default: Configuring and enabling network interfaces...
==> default: Mounting shared folders...
    default: /vagrant =>
    default: + sudo docker run hello-world
    default: Unable to find image 'hello-world:latest' locally
    default: latest: Pulling from library/hello-world
    default: 0e03bdcc26d7:
    default: Pulling fs layer
    default: 0e03bdcc26d7:
    default: Verifying Checksum
    default: 0e03bdcc26d7:
```

```
default: Download complete
default: 0e03bdcc26d7:
default: Pull complete
default: Digest:
sha256:4cf9c47f86df71d48364001ede3a4fcd85ae80ce02ebad74156906caff5378bc
default: Status: Downloaded newer image for hello-world:latest
default:
default: Hello from Docker!
default: This message shows that your
default: installation appears to be working correctly.
default:
default: To generate this message, Docker took the following steps:
default: 1. The Docker client contacted the Docker daemon.
default: 2. The Docker daemon pulled the "hello-world" image
default: from the Docker Hub.
default:(amd64)
default: 3. The Docker daemon created a new container from that image
default: which runs the executable that produces the output you are
default: currently reading.
default: 4. The Docker daemon streamed that output to the Docker
default: client, which sent it to your terminal.
default:
default: To try something more ambitious, you can run an Ubuntu
default: container with:
default: $ docker run -it ubuntu bash
default:
default: Share images, automate workflows, and more with a free Docker ID:
default: https://hub.docker.com
default:
default: For more examples and ideas, visit:
default: https://docs.docker.com/get-started
```

Теперь, когда мы запустили хост, начнем рассматривать различные сетевые конфигурации, с которыми нам придется работать в Docker. Пример 3.4 показывает, что Docker при инсталляции создает три типа сетей: мост, хост и автономную.

Пример 3.4. Сети Docker

```
vagrant@ubuntu-xenial:~$ sudo docker network ls
NETWORK ID        NAME      DRIVER    SCOPE
1fd1db59c592      bridge    bridge    local
eb34a2105b0f      host      host      local
941ce103b382      none      null      local
vagrant@ubuntu-xenial:~$
```

Сеть по умолчанию — мост Docker, контейнер присоединяется к ней и получает IP-адрес в подсети по умолчанию `172.17.0.0/16`. В примере 3.5 рассматриваются

интерфейсы по умолчанию в Ubuntu и инсталляция Docker, которая создает bridge-интерфейс docker0 для хоста.

Пример 3.5. Bridge-интерфейс в Docker

```
vagrant@ubuntu-xenial:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc
noqueue state UNKNOWN group default qlen 1 1
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
   valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
   valid_lft forever preferred_lft forever
2: enp0s3:
<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
default qlen 1000 2
   link/ether 02:8f:67:5f:07:a5 brd ff:ff:ff:ff:ff:ff
   inet 10.0.2.15/24 brd 10.0.2.255 scope global enp0s3
   valid_lft forever preferred_lft forever
   inet6 fe80::8f:67ff:fe5f:7a5/64 scope link
   valid_lft forever preferred_lft forever
3: enp0s8:
<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
default qlen 1000 3
   link/ether 08:00:27:22:0e:46 brd ff:ff:ff:ff:ff:ff
   inet 192.168.1.19/24 brd 192.168.1.255 scope global enp0s8
   valid_lft forever preferred_lft forever
   inet 192.168.1.20/24 brd 192.168.1.255 scope global secondary enp0s8
   valid_lft forever preferred_lft forever
   inet6 2605:a000:160d:517:a00:27ff:fe22:e46/64 scope global mngtmpaddr dynamic
   valid_lft 604600sec preferred_lft 604600sec
   inet6 fe80::a00:27ff:fe22:e46/64 scope link
   valid_lft forever preferred_lft forever
4: docker0:
<NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group
Default 4
   link/ether 02:42:7d:50:c7:01 brd ff:ff:ff:ff:ff:ff
   inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
   valid_lft forever preferred_lft forever
   inet6 fe80::42:7dff:fe50:c701/64 scope link
   valid_lft forever preferred_lft forever
```

Пояснения к примеру:

1. Это петлевой (loopback) интерфейс.
2. enp0s3 — виртуальный интерфейс с NAT.

3 enp0s8 — хост-интерфейс, он принадлежит той же сети, что и наш хост, и использует DHCP, чтобы получить адрес 192.168.1.19 моста Docker.

4 Интерфейс контейнера Docker использует сетевой мост.

В примере 3.6 происходит запуск контейнера busybox с помощью команды docker run и делается запрос, в ответ на который Docker должен вернуть IP-адрес контейнера. NAT-преобразованный Docker адрес по умолчанию есть `172.17.0.0/16`, наш busybox-контейнер получает `172.17.0.2`.

Пример 3.6. Мост в Docker

```
vagrant@ubuntu-xenial:~$ sudo docker run -it busybox ip a
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
df8698476c65: Pull complete
Digest:sha256:d366a4665ab44f0648d7a00ae3fae139d55e32f9712c67accd604bb55df9d0a
Status: Downloaded newer image for busybox:latest
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
valid_lft forever preferred_lft forever
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
valid_lft forever preferred_lft forever
```

Хост-сеть в примере 3.7 показывает, что контейнер и хост используют одно и то же пространство имен. Мы видим, что интерфейсы те же, что и у хоста; команда `ip a` для контейнера выдает интерфейсы `enp0s3`, `enp0s8` и `docker0`.

Пример 3.7. Хост-сеть в Docker

```
vagrant@ubuntu-xenial:~$ sudo docker run -it --net=host busybox ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
   valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
   valid_lft forever preferred_lft forever`
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
   link/ether 02:8f:67:5f:07:a5 brd ff:ff:ff:ff:ff:ff
   inet 10.0.2.15/24 brd 10.0.2.255 scope global enp0s3
   valid_lft forever preferred_lft forever
   inet6 fe80::8f:67ff:fe5f:7a5/64 scope link
   valid_lft forever preferred_lft forever
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
   link/ether 08:00:27:22:0e:46 brd ff:ff:ff:ff:ff:ff
```

```
    inet 192.168.1.19/24 brd 192.168.1.255 scope global enp0s8
    valid_lft forever preferred_lft forever
    inet 192.168.1.20/24 brd 192.168.1.255 scope global secondary enp0s8
    valid_lft forever preferred_lft forever
    inet6 2605:a000:160d:517:a00:27ff:fe22:e46/64 scope global dynamic
    valid_lft 604603sec preferred_lft 604603sec
    inet6 fe80::a00:27ff:fe22:e46/64 scope link
    valid_lft forever preferred_lft forever
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
    link/ether 02:42:7d:50:c7:01 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
    valid_lft forever preferred_lft forever
    inet6 fe80::42:7dff:fe50:c701/64 scope link
    valid_lft forever preferred_lft forever
```

Давайте сравним с приведенным ранее примером интерфейса veth и посмотрим, насколько все упростится, если будет использоваться Docker. Для этого нам понадобится процесс, который будет поддерживать работу контейнера. Следующая команда запускает busybox-контейнер:

```
vagrant@ubuntu-xenial:~$ sudo docker run -it --rm busybox /bin/sh
/#
```

У нас есть петлевой интерфейс, `lo`, и интерфейс Ethernet `eth0`, связанный с `veth12`, с Docker IP-адресом по умолчанию `172.17.0.2`. Поскольку наша предыдущая команда выдала только результат вызова `ip a`, а затем контейнер прекратил работу, то Docker снова воспользовался IP-адресом `172.17.0.2` для запуска busybox-контейнера:

```
/# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
    valid_lft forever preferred_lft forever
11: eth0@if12: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
    valid_lft forever preferred_lft forever
```

Задав команду `ip r` внутри пространства имен контейнера, мы можем увидеть, что таблица маршрутизации контейнера сконфигурировалась автоматически:

```
/ # ip r
default via 172.17.0.1 dev eth0
172.17.0.0/16 dev eth0 scope link src 172.17.0.2
```

Если мы откроем новый терминал и соединимся через `vagrant ssh` с нашей машиной Vagrant Ubuntu, а затем зададим команду `docker ps`, то увидим всю информацию по процессам в запущенном busybox-контейнере:

```
vagrant@ubuntu-xenial:~$ sudo docker ps
CONTAINER ID   IMAGE        COMMAND
3b5a7c3a74d5   busybox      /bin/sh"

CREATED          STATUS          PORTS        NAMES
47 seconds ago   Up 46 seconds               competent_mendel
```

Мы видим интерфейс `veth veth68b6f80@if11`, **который Docker сконфигурировал для контейнера в пространстве имен хоста. Он является частью моста для** `docker0` **и активируется командой** `master docker0 state UP`:

```
vagrant@ubuntu-xenial:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default qlen 1
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
valid_lft forever preferred_lft forever
inet6 ::1/128 scope host
valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
group default qlen 1000
link/ether 02:8f:67:5f:07:a5 brd ff:ff:ff:ff:ff:ff
inet 10.0.2.15/24 brd 10.0.2.255 scope global enp0s3
valid_lft forever preferred_lft forever
inet6 fe80::8f:67ff:fe5f:7a5/64 scope link
valid_lft forever preferred_lft forever
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
group default qlen 1000
  link/ether 08:00:27:22:0e:46 brd ff:ff:ff:ff:ff:ff
  inet 192.168.1.19/24 brd 192.168.1.255 scope global enp0s8
  valid_lft forever preferred_lft forever
  inet 192.168.1.20/24 brd 192.168.1.255 scope global secondary enp0s8
  valid_lft forever preferred_lft forever
inet6 2605:a000:160d:517:a00:27ff:fe22:e46/64 scope global mngtmpaddr dynamic
  valid_lft 604745sec preferred_lft 604745sec
  inet6 fe80::a00:27ff:fe22:e46/64 scope link
  valid_lft forever preferred_lft forever
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default
link/ether 02:42:7d:50:c7:01 brd ff:ff:ff:ff:ff:ff
inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
valid_lft forever preferred_lft forever
inet6 fe80::42:7dff:fe50:c701/64 scope link
valid_lft forever preferred_lft forever
12: veth68b6f80@if11: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
master docker0 state UP group default
link/ether 3a:64:80:02:87:76 brd ff:ff:ff:ff:ff:ff link-netnsid 0
inet6 fe80::3864:80ff:fe02:8776/64 scope link
valid_lft forever preferred_lft forever
```

Таблица маршрутизации хоста показывает маршруты к контейнеру Docker, запущенному на хосте:

```
vagrant@ubuntu-xenial:~$ ip r
default via 192.168.1.1 dev enp0s8
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.15
172.17.0.0/16 dev docker0 proto kernel scope link src 172.17.0.1
192.168.1.0/24 dev enp0s8 proto kernel scope link src 192.168.1.19
```

По умолчанию Docker не добавляет создаваемое им пространство имен к каталогу /var/run, где команда `ip netns list` будет искать пространства имен вновь созданной сети. Давайте рассмотрим, каким образом можно все же увидеть эти пространства имен. Для того чтобы получить список пространств имен сети Docker с помощью команды `ip`, необходимо сделать следующее:

1. Получить PID работающего контейнера.
2. Установить символическую (мягкую) ссылку пространства имен сети с /proc/PID/net на /var/run/netns.
3. Выдать список пространства имен сети.

С помощью команды `docker ps` получаем ID контейнера, требуемый для просмотра PID работающих процессов в пространстве имен хоста:

```
vagrant@ubuntu-xenial:~$ sudo docker ps
CONTAINER ID       IMAGE             COMMAND
1f3f62ad5e02       busybox           "/bin/sh"

CREATED            STATUS            PORTS NAMES
11 minutes ago     Up 11 minutes     determined_shamir
```

Команда `docker inspect` позволяет нам проанализировать выдачу и получить PID хостового процесса. Если теперь мы выполним `ps -p` с полученным PID, то увидим, что этот процесс соответствует программе sh, которая отслеживает нашу команду `docker run`:

```
vagrant@ubuntu-xenial:~$ sudo docker inspect -f '{{.State.Pid}}' 1f3f62ad5e02
25719
vagrant@ubuntu-xenial:~$ ps -p 25719
PID TTY         TIME CMD
25719 pts/0     00:00:00 sh
```

`1f3f62ad5e02` — это ID контейнера, а `25719` — PID процесса sh, соответствующего контейнеру busybox, так что теперь мы можем создать символическую ссылку для пространства имен сети контейнера на область, где ее будет искать команда `ip`:

```
$ sudo ln -sfT /proc/25719/ns/net /var/run/netns/1f3f62ad5e02
```

Имейте в виду, что ID контейнера и ID процесса в вашей системе будут отличаться от значений, использованных в нашем примере.

Теперь команды ip netns exec возвращают тот же IP-адрес, 172.17.9.2, что и команда docker exec:

```
vagrant@ubuntu-xenial:~$ sudo ip netns exec 1f3f62ad5e02 ip a
1: lo:
<LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
   valid_lft forever preferred_lft forever
13: eth0@if14:
<BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
   link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff link-netnsid 0
   inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
   valid_lft forever preferred_lft forever
```

С помощью docker exec и ip мы можем проверить параметры контейнера busybox. IP-адрес, MAC-адрес и сетевые интерфейсы выдаются корректно:

```
vagrant@ubuntu-xenial:~$ sudo docker exec 1f3f62ad5e02 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
13: eth0@if14: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
        valid_lft forever preferred_lft forever
```

Docker инициирует наш контейнер, создает пространство имен, пару veth и мост docker0 (если он еще не существует) — и сворачивает все действия, связанные с созданием и удалением контейнера, всего в одну команду! С точки зрения разработчика приложений, это мощно! Не нужно помнить все соответствующие команды Linux и нет опасности порушить сеть на хост-компьютере. Пока мы рассматривали задачу с одним хостом; в следующем разделе познакомимся, как Docker управляет взаимодействием между хостами в кластере.

Сетевая модель Docker

Проект Libnetwork — это вклад Docker в сети контейнеров, концепция получила отражение в сетевой модели контейнера CNM. Libnetwork реализует CNM и имеет три компонента:

Песочница (sandbox), конечной точки и сетевой. Задачей песочницы является управление пространствами имен сети Linux для всех запущенных на хосте контейнеров. Сетевой компонент — это набор конечных точек одной и той же сети. Конечные точки, в свою очередь, это хосты сети. Всем управляет сетевой контроллер с помощью API-интерфейсов движка Docker.

Для изоляции сети в конечной точке Docker использует iptables. Контейнер сообщает порт, через который он может быть доступен извне. Контейнеры не получают

публичные IPv4-адреса — им присваиваются частные адреса системы RFC 1918. Запущенные в контейнерах сервисы должны иметь свои порты, во избежание конфликтов порты контейнеров должны привязываться к порту хоста. При запуске Docker создает на хост-компьютере виртуальный интерфейс моста, `docker0`, и присваивает ему случайный IP-адрес из набора RFC 1918. Этот мост передает пакеты между двумя связанными устройствами — как это делает физический мост в обычной жизни. Каждый новый контейнер получает интерфейс, автоматически связанный с мостом `docker0`. Схематично это представлено на рис. 3.9 и похоже на подход, который мы уже разбирали в предыдущих разделах.

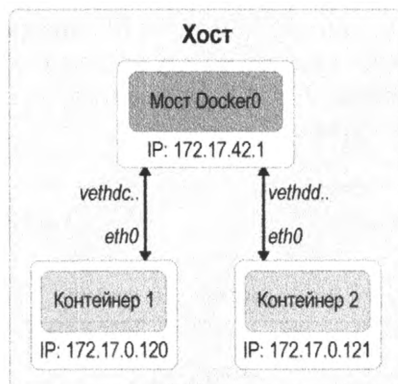


Рис. 3.9. Мост-интерфейс в Docker

Модель CNM привязывает сетевые режимы к драйверам, которые мы уже рассматривали. Ниже приводим сетевые режимы и их эквиваленты из движка Docker:

Мост

По умолчанию мост Docker (см. рис. 3.9 и наши предыдущие примеры).

Пользовательский или удаленный

Определяемый пользователем мост-интерфейс, или пользователям разрешается создавать и использовать свои собственные плагины.

Оверлей

Оверлейная сеть.

Ноль (null)

Нет сетевых опций.

Сети с мостами предназначены для контейнеров, запущенных на одном и том же хост-компьютере. Взаимодействие с контейнерами, запущенными на различных хостах, может происходить через оверлейную сеть. Docker использует концепцию глобальных и локальных драйверов. Локальные драйверы, например мост, привязаны к хосту и не участвуют во взаимодействии между узлами. Такое взаимодействие является задачей глобальных драйверов, таких как оверлеи. Для осуществления координации между отдельными машинами глобальные драйверы используют libkv — абстракцию хранилища «ключ-значение». Модель CNM не предоставляет

хранилища «ключ-значение», поэтому используются внешние программы типа Consul, etcd и Zookeeper.

В следующем разделе мы подробно рассмотрим технологию оверлейных сетей.

Оверлейная сеть

Пока все наши примеры запускались на одном и том же хост-компьютере, однако масштабируемые промышленные приложения не работают на единственном хосте. Чтобы обеспечить взаимодействие между приложениями, запущенными в контейнерах на разных узлах, необходимо решить ряд задач: управление маршрутизацией между хостами, конфликты портов, контроль IP-адресов и др. Технология, помогающая контейнерам в маршрутизации между хостами, — это VXLAN. На рис. 3.10 показана созданная с помощью VXLAN оверлейная сеть 2-го уровня, работающая поверх физической сети 3-го уровня.

Рис. 3.10. Туннель VXLAN

Сети VXLAN коротко уже упоминались в *главе 1*, в данном разделе мы дадим более подробную информацию, объясняющую, каким образом осуществляется обмен данными, обеспечивающий взаимодействие между контейнерами.

VXLAN — это расширение протокола VLAN, создающее 16 миллионов уникальных идентификаторов. Согласно спецификации IEEE 802.1Q максимальное число VLAN на данной сети Ethernet равно 4094. Транспортный протокол в физической сети дата-центра — это IP плюс UDP. VXLAN задает схему инкапсуляции MAC-в-UDP, в которой к кадру 2-го уровня добавляется VXLAN-заголовок, упакованный в UDP-IP пакет. На рис. 3.11. показан IP-пакет, инкапсулированный в UDP-пакет, и его заголовки.

Пакет VXLAN является пакетом, инкапсулированным по схеме MAC-в-UDP. Кадр 2-го уровня имеет добавленный к нему VXLAN-заголовок и помещается в пакет UDP-IP. Идентификатор VXLAN содержит 24 бита. Это объясняет, каким образом VXLAN может поддерживать 16 миллионов сегментов.

Рис. 3.11 показывает более подробную версию VXLAN по сравнению с рассмотренной в *главе 1*. Здесь мы видим на обоих хостах конечные точки туннеля VXLAN, VTEP, они привязаны к мостам-интерфейсам хоста, а контейнеры привязаны к мосту. Конечные точки VTEP осуществляют инкапсуляцию и деинкапсуля-

цию кадров данных. Взаимодействие между VTEP обеспечивает отправку данных в соответствующие контейнеры. Покидающие контейнер данные инкапсулируются с помощью VXLAN и передаются по VXLAN-туннелям с последующей деинкапсуляцией в парной конечной точке.

Оверлейная сеть обеспечивает взаимодействие между хостами через сеть контейнеров. В модели CNM присутствуют некоторые детали, которые делают ее несовместимой с Kubernetes. Разработчики Kubernetes приняли решение использовать проект CNI, начинавшийся в рамках CoreOs. Он проще, чем CNM, не требует процессов-демонов и является кросс-платформенным.

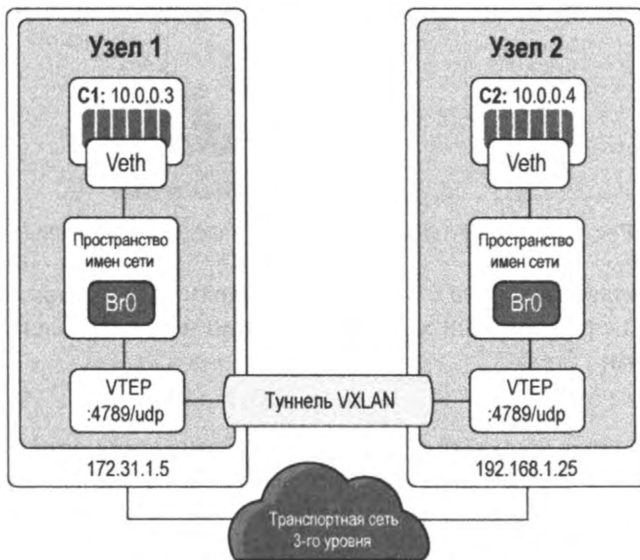


Рис. 3.11. Подробная схема туннеля VXLAN

Сетевой интерфейс контейнера

Сетевой интерфейс контейнера CNI — это программный интерфейс между средой запуска контейнера и сетью. Программирование CNI представляет собой выбор из многих опций, ниже мы рассмотрим несколько наиболее важных. CNI стартовал в рамках CoreOs как часть проекта rkt, в настоящее время он является проектом CNCF. Проект CNI состоит из спецификации и библиотек для разработки плагинов, которые используются для конфигурирования сетевых интерфейсов контейнеров Linux. Задачей CNI является обеспечение сетевой связанности контейнеров, для этого контейнеру предоставляются ресурсы при его создании и освобождаются — при его удалении. CNI-плагин отвечает за привязку сетевого интерфейса к пространству имен сети контейнера и за необходимые изменения на стороне хоста. Затем он присваивает интерфейсу IP-адрес и задает маршрут к нему. На рис. 3.12 представлена схема архитектуры CNI. Среда запуска контейнера использует конфигурационный файл для получения информации, касающейся хостовой сети;

в Kubernetes модуль Kubelet тоже использует конфигурационный файл. CNI и среда запуска контейнера взаимодействуют друг с другом через команды, посылаемые сконфигурированному CNI-плагину.

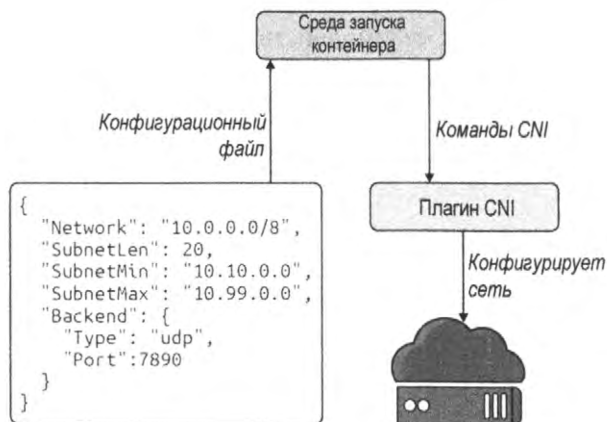


Рис. 3.12. Архитектура CNI-сетевого интерфейса контейнера

Существует несколько проектов с открытым программным кодом, которые предлагают CNI-плагины с различными характеристиками и функционалом. Некоторые из них мы перечислим:

Cilium

Программный пакет с открытым кодом для обеспечения сетевого взаимодействия между контейнерами для запуска приложений. Cilium является интерфейсом CNI 7-го уровня (HTTP) и может усилить сетевую безопасность на уровнях 3–7 благодаря использованию модели безопасности на основе идентификации, независимой от сетевой адресации. В основе лежит технология eBPF Linux.

Flannel

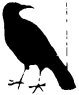
Пакет Flannel — это простой метод конфигурирования сети 3-го уровня, разработанный для Kubernetes. Flannel решает задачи сетевого взаимодействия. Для хранения своей установочной информации Flannel использует хранилище данных `etcd` для кластеров Kubernetes.

Calico

Согласно документации Calico «благодаря сочетанию гибкости работы в сети с безопасностью запуска приложений в любом окружении предоставляются решения с производительностью ядра Linux и настоящей облачной масштабируемостью». Имеется полная поддержка политики сетевой безопасности, возможна работа в сочетании с другими модулями CNI. Calico не использует оверлейную сеть. Для маршрутизации пакетов между хостами Calico конфигурирует сеть 3-го уровня, использующую протокол BGP. Calico может интегрироваться с Istio, так называемой service mesh (сервисная ячейка), чтобы регулировать рабочую нагрузку внутри кластера как на уровне service mesh, так и на уровне сетевой инфраструктуры.

AWS

AWS имеет приложение с открытым кодом, реализующее CNI, — AWS VPC CNI. Оно предоставляет высокую пропускную способность и доступность благодаря прямому выходу на AWS-сеть. Данная реализация CNI имеет малую задержку (латентность) благодаря низкой дополнительной нагрузке, поскольку AWS-сеть не использует оверлеи и имеет минимальный сетевой джиттер. Администраторы сетей могут использовать существующие AWS VPC и лучшие практики безопасности для построения сетей Kubernetes на базе AWS. Эти сети имеют возможность реализовать лучшие практики, поскольку AWS VPC используют исходные AWS-сервисы типа журналов потоков VPC для анализа сетевых событий и паттернов, политики маршрутизации VPC — для управления трафиком, группы безопасности и контрольные списки доступа к сетям — для изоляции сетевого трафика. Подробнее AWS VPC CNI будет обсуждаться в *главе 6*.



Список имеющихся опций CNI можно посмотреть на сайте Kubernetes.io.

Проект CNI предоставляет большое число опций, так что администраторам сетей и разработчикам приложений приходится решать, какой из пакетов CNI наилучшим образом отвечает их задачам. В дальнейших главах мы рассмотрим конкретные задачи и их решения, чтобы помочь администраторам сделать правильный выбор.

В следующем разделе мы разберем примеры работы сетей контейнеров, используя Docker и наш веб-сервер на языке Go.

Подключение контейнера к сети

Как мы это делали в предыдущих главах, снова обратимся к нашему веб-серверу на Go — на этот раз, чтобы проработать процедуру подключения к сетям. Мы посмотрим, что происходит на уровне контейнера, если мы разворачиваем вер-сервер в контейнере на хосте под Ubuntu.

Разберем два возможных варианта соединения:

- Контейнер с контейнером на хосте Docker.
- Контейнер с контейнером на различных хостах.

Веб-сервер запрограммирован на запуск на порту 8080, `http.ListenAndServe("0.0.0.0:8080", nil)`, как видно из примера 3.8.

Пример 3.8. Минимальный веб-сервер на языке Go

```
package main
import (
"fmt"
"net/http"
)
```

```
func hello(w http.ResponseWriter, _ *http.Request) {
fmt.Fprintf(w, "Hello")
}

func main() {
http.HandleFunc("/", hello)
http.ListenAndServe("0.0.0.0:8080", nil)
}
```

Чтобы инициировать наш веб-сервер, мы должны создать его из Dockerfile. Пример 3.9 показывает Dockerfile нашего сервера. Dockerfile содержит команды, задающие порядок действий по созданию образа. Файл начинается с команды FROM и указывает, каким должен быть базовый образ. Команда RUN задает выполняемую последовательность. Комментарии следуют после знака #. Не забывайте, что каждая строка в Dockerfile создает новый слой, если она изменяет состояние образа. Разработчикам приходится находить разумный баланс между количеством созданных для образа слоев и читабельностью Dockerfile.

Пример 3.9. Dockerfile для минимального веб-сервера на языке Go

```
FROM golang:1.15 AS builder 1
WORKDIR /opt 2
COPY web-server.go . 3
RUN CGO_ENABLED=0 GOOS=linux go build -o web-server . 4

FROM golang:1.15 5
WORKDIR /opt 6
COPY --from=0 /opt/web-server . 7
CMD ["/opt/web-server"] 8
```

Пояснения:

1. Поскольку наш веб-сервер написан на Go, мы можем скомпилировать сервер в контейнере, чтобы уменьшить размер образа до размера бинарного файла Go. Мы начинаем с базового образа веб-сервера, используя Go версии 1.15.
2. `WORKDIR` задает рабочую директорию, из которой будут запускаться все последующие команды.
3. `COPY` копирует файл `web-server.go`, который определяет наше приложение как рабочую директорию.
4. `RUN` указывает Docker скомпилировать наше Go приложение в контейнер сборки (builder).
5. Чтобы запустить наше приложение, мы задаем `FROM` для базового образа приложения, это снова `golang:1.15`, мы можем еще уменьшить окончательный размер образа путем использования других минимальных образов типа alpine.

6 Будучи новым контейнером, снова устанавливаем рабочую директорию как /opt.

7 COPY копирует скомпилированный бинарный файл Go из контейнера сборки в контейнер приложений.

8 CMD указывает Docker, что команда запуска нашего приложения — это запуск веб-сервера.

Существуют лучшие практики для Dockerfile, которых рекомендуется придерживаться администраторам и разработчикам, когда они работают с контейнеризованными приложениями.

- Использовать один ENTRYPOINT на один Dockerfile. ENTRYPOINT или CMD сообщают Docker, что начало процесса находится внутри запущенного контейнера, так что будет только один работающий процесс — идея контейнеров как раз и состоит в изоляции процессов.
- Чтобы сократить слои контейнера, разработчикам рекомендуется объединять одинаковые команды в одну, используя && и \. Каждая новая команда в Dockerfile добавляет слой к образу контейнера Docker, тем самым увеличивая объем хранимой информации.
- Использовать кеширование для уменьшения времени создания контейнера. Если это не изменяет слой, то оно должно стоять в начале Dockerfile. Кеширование — один из аргументов, что порядок следования операторов является определяющим.
- Добавлять файлы, которые с наименьшей вероятностью будут изменены в первую очередь, и файлы, которые с наибольшей вероятностью будут изменены в последнюю очередь.
- Использовать многоэтапную сборку (multistage build), чтобы значительно уменьшить размер окончательного образа.
- Не устанавливать ненужные пакеты и инструменты. Это позволит уменьшить размер и уязвимость контейнера, а также сократит время прохода по сети между реестром и хостами с запущенными на них контейнерами.

Давайте соберем наш веб-сервер и рассмотрим необходимые для этого команды Docker.

Команда docker build дает задание Docker собрать наши образы согласно командам в Dockerfile:

```
$ sudo docker build
Sending build context to Docker daemon
Step 1/8 : FROM golang:1.15 AS builder
1.15: Pulling from library/golang

57df1a1f1ad8: Pull complete
71e126169501: Pull complete
1af28a55c3f3: Pull complete
03f1c9932170: Pull complete
```

```
f4773b341423: Pull complete
fb320882041b: Pull complete
24b0ad6f9416: Pull complete
Digest:
sha256:da7ff43658854148b401f24075c0aa390e3b52187ab67cab0043f2b15e754a68
Status: Downloaded newer ımage for golang:1.15
---> 05c8f6d2538a
     Step 2/8 : WORKDIR /opt
---> Running in 20c103431e6d
     Removing intermediate container 20c103431e6d
---> 74ba65cfdf74
     Step 3/8 : COPY web-server.go .
---> 7a36ec66be52
     Step 4/8 : RUN CGO_ENABLED=0 GOOS=linux go build -o web-server .
---> Runnıng in 5ea1c0a85422
     Removing intermediate contaıner 5ea1c0a85422
---> b508120db6ba
     Step 5/8 : FROM golang:1.15
---> 05c8f6d2538a
     Step 6/8 : WORKDIR /opt
---> Using cache
---> 74ba65cfdf74
     Step 7/8 : COPY --from=0 /opt/web-server .
---> dde6002760cd
     Step 8/8 : CMD ["/opt/web-server"]
---> Runnıng in 2bcb7c8f5681
     Removing intermediate contaıner 2bcb7c8f5681
     Removing ıntermediate container 2bcb7c8f5681
---> 72fd05de6f73
     Successfully buılt 72fd05de6f73
```

Наш тестовый веб-сервер имеет ID контейнера 72fd05de6f73, что не очень удобно читать, поэтому воспользуемся командой docker tag, чтобы присвоить ему более «дружественное» имя:

```
$ sudo docker tag 72fd05de6f73 go-web:v0.0.1
```

Команда `docker images` возвращает список локально доступных образов. У нас есть один образ из примера по инсталляции Docker и образ busybox, который мы использовали для тестирования нашей сетевой конфигурации. Если контейнер не доступен локально, то он загружается из реестра; чтобы уменьшить время загрузки, необходимо делать размеры образов как можно более малыми:

```
$ sudo docker ımages
REPOSITORY      TAG         IMAGE ID          SIZE
<none>          <none>      b508120db6ba      857MB
go-web          v0.0.1      72fd05de6f73      845MB
golang          1.15        05c8f6d2538a      839MB
```

```
busybox        latest    6858809bf669    1.23MB
hello-world    latest    bf756fb1ae65    13.3kB
```

Команда docker ps показывает нам запущенные на хосте контейнеры. Вспомнив пример с сетевым пространством имен, узнаем, что контейнер busybox из этого примера все еще работает:

```
$ sudo docker ps
CONTAINER ID IMAGE     COMMAND   STATUS   PORTS       NAMES
1f3f62ad5e02 busybox   "/bin/sh" Up      11 minutes   determined_shamir
```

С помощью docker logs можно распечатать любую стандартную выдачу контейнера; в настоящий момент, как мы видим, от образа busybox никакой информации не поступает:

```
vagrant@ubuntu-xenial:~$ sudo docker logs 1f3f62ad5e02
vagrant@ubuntu-xenial:~$
```

Выполняемые команды внутри контейнера Docker запускаются через docker exec. Мы уже ранее использовали эту команду, когда рассматривали сетевые конфигурации Docker:

```
vagrant@ubuntu-xenial:~$ sudo docker exec 1f3f62ad5e02 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
inet 127.0.0.1/8 scope host lo
valid_lft forever preferred_lft forever
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
inet 172.17.0.2/16 brd 172.17.255.255 scope global eth0
valid_lft forever preferred_lft forever
vagrant@ubuntu-xenial:~$
```

Команды для Docker CLI приводятся в документации.

В предыдущем разделе мы запустили веб-сервер в контейнере. Чтобы проверить соединение с сетями, будем использовать образ dnsutils, обычно применяемый в Kubernetes для сквозного тестирования. Образ находится по адресу: gcr.io/kubernetes-e2e-test-images/dnsutils:1.3.

Заданием имени образа мы запускаем копирование образов Docker из реестра контейнеров Google в нашу локальную файловую систему Docker:

```
$ sudo docker pull gcr.io/kubernetes-e2e-test-images/dnsutils:1.3
1.3: Pulling from kubernetes-e2e-test-images/dnsutils
5a3ea8efae5d: Pull complete
7b7e943444f2: Pull complete
59c439aa0fa7: Pull complete
3702870470ee: Pull complete
```

```
Digest: sha256:b31bcf7ef4420ce7108e7fc10b6c00343b21257c945eec94c21598e72a8f2de0
Status: Downloaded newer image for gcr.io/kubernetes-e2e-test-images/dnsutils:1.3
gcr.io/kubernetes-e2e-test-images/dnsutils:1.3
```

Теперь, запустив наше приложение в контейнере, посмотрим, как контейнеры общаются друг с другом через сеть.

Соединение контейнер-контейнер

Сначала рассмотрим взаимодействие между контейнерами, запущенными на одном и том же хосте. Начнем с запуска образа `dnsutils` и последующего перехода в shell:

```
$ sudo docker run -it gcr.io/kubernetes-e2e-test-images/dnsutils:1.3 /bin/sh
/ #
```

По умолчанию, конфигурация сети Docker обеспечивает образу dnsutils соединение с Интернетом:

```
/ # ping google.com -c 4
PING google.com (172.217.9.78): 56 data bytes
64 bytes from 172.217.9.78: seq=0 ttl=114 time=39.677 ms
64 bytes from 172.217.9.78: seq=1 ttl=114 time=38.180 ms
64 bytes from 172.217.9.78: seq=2 ttl=114 time=43.150 ms
64 bytes from 172.217.9.78: seq=3 ttl=114 time=38.140 ms

--- google.com ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 38.140/39.786/43.150 ms
/ #
```

Наш веб-сервер запускается с мостом Docker, предусмотренным по умолчанию. Через SSH-соединение с хостом Vagrant мы запустим веб-сервер с помощью следующей команды:

```
$ sudo docker run -it -d -p 80:8080 go-web:v0.0.1
a568732bc191bb1f5a281e30e34ffdeabc624c59d3684b93167456a9a0902369
```

Опция -it используется для интерактивных процессов (таких как shell); с помощью `-it` мы предоставляем контейнерному процессу утилиту TTY. Опция `-d` запускает контейнер в фоновом режиме; это даст нам возможность использовать терминал и выдать на него полный ID контейнера Docker. Для сетевой работы нам важна опция `-p` — она создает соединение через порт между хостом и контейнером. Наш веб-сервер запущен на порту 8080 и связывает этот порт с портом 80 на хосте.

Выполнив docker ps, убеждаемся, что у нас два запущенных контейнера: веб-сервер на порту 8080, связанном с портом 80 на хосте, и shell, работающий внутри нашего контейнера dnsutils:

```
vagrant@ubuntu-xenial:~$ sudo docker ps
CONTAINER ID IMAGE          COMMAND            CREATED       STATUS
906fd860f84d go-web:v0.0.1  "/opt/web-server" 4 minutes     ago Up 4 minutes
25ded12445df dnsutils:1.3   "/bin/sh"          6 minutes     ago Up 6 minutes
```

```
PORTS                                     NAMES
0.0.0.0:8080->8080/tcp                    frosty_brown
                                          brave_zhukovsky
```

Используем команду `docker inspect`, чтобы получить Docker IP-адрес контейнера веб-сервера:

```
$ sudo docker inspect
-f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}'
906fd860f84d
172.17.0.2
```

Для образа dnsutils мы можем использовать сетевой Docker адрес веб-сервера 172.17.0.2 и порт контейнера 8080 :

```
/ # wget 172.17.0.2:8080
Connecting to 172.17.0.2:8080 (172.17.0.2:8080)
index.html      100% |*******************************************|
                5      0:00:00 ETA
/ # cat index.html
Hello/ #
```

Каждый контейнер может взаимодействовать с другим через мост docker0 и порты контейнера, поскольку они размещены на одном и том же хосте Docker и в той же сети. Хост Docker получает доступ к контейнеру через IP-адрес контейнера и соответствующий порт:

```
vagrant@ubuntu-xenial:~$ curl 172.17.0.2:8080
Hello
```

Однако этого не происходит, если использовать порт хоста и Docker IP-адрес, полученные из команды docker run:

```
vagrant@ubuntu-xenial:~$ curl 172.17.0.2:80
curl: (7) Failed to connect to 172.17.0.2 port 80: Connection refused
vagrant@ubuntu-xenial:~$ curl 172.17.0.2:8080
Hello
```

И наоборот — используя петлевой интерфейс, мы видим, что хост может получить доступ к веб-серверу только через порт, указанный для связи, т. е. 80, а не через Docker порт 8080:

```
vagrant@ubuntu-xenial:~$ curl 127.0.0.1:8080
curl: (7) Failed to connect to 127.0.0.1 port 8080: Connection refused
vagrant@ubuntu-xenial:~$ curl 127.0.0.1:80
Hellovagrant@ubuntu-xenial:~$
```

Это же относится и к dnsutils: образ dnsutils в сети Docker, используя Docker IP-адрес контейнера нашего веб-сервера, может использовать только Docker порт 8080, а не связанный с ним порт хоста 80:

```
/ # wget 172.17.0.2:8080 -qO-
Hello/ #
/ # wget 172.17.0.2:80 -qO-
wget: can't connect to remote host (172.17.0.2): Connection refused
```

А теперь попробуем задать для dnsutils его адрес от петлевого интерфейса и оба порта — порт Docker и связанный с ним порт хоста:

```
/ # wget localhost:80 -qO-
wget: can't connect to remote host (127.0.0.1): Connection refused
/ # wget localhost:8080 -qO-
wget: can't connect to remote host (127.0.0.1): Connection refused
```

Ни одна команда не сработала так, как ожидалось: образ dnsutils имеет свой отдельный сетевой стек и у него нет общего пространства имен с нашим веб-сервером. Понимание этого обстоятельства чрезвычайно важно для работы в Kubernetes, т. к. *под* — это набор контейнеров, имеющих общее сетевое пространство имен. Ниже мы рассмотрим, как взаимодействуют между собой контейнеры, запущенные на разных хостах.

Взаимодействие между контейнерами на разных хостах

В предыдущем примере вы рассмотрели работу сети контейнеров, запущенных на одном и том же хосте, а как взаимодействуют между собой контейнеры на разных хостах? В данном разделе мы развернем контейнеры на разных хостах и изучим, как происходит их взаимодействие.

Запустим второй хост с Vagrant Ubuntu, host-2, и соединимся с ним через SSH, как мы ранее делали с хостом Docker. Мы видим, что IP-адрес отличается от адреса хоста Docker, где работает наш веб-сервер:

```
vagrant@host-2:~$ ifconfig enp0s8
enp0s8 Link encap:Ethernet HWaddr 08:00:27:f9:77:12
       inet addr:192.168.1.23 Bcast:192.168.1.255 Mask:255.255.255.0
       inet6 addr: fe80::a00:27ff:fef9:7712/64 Scope:Link
       UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
       RX packets:65630 errors:0 dropped:0 overruns:0 frame:0
       TX packets:2967 errors:0 dropped:0 overruns:0 carrier:0
       collisions:0 txqueuelen:1000
       RX bytes:96493210 (96.4 MB) TX bytes:228989 (228.9 KB)
```

Мы можем получить доступ к веб-серверу через IP-адрес хоста Docker, 192.168.1.20, и порт 80, указанный в опциях команды docker run. Порт 80 открыт на хосте Docker, но он недоступен через порт контейнера 8080 с IP-адресом хоста:

```
vagrant@ubuntu-xenial:~$ curl 192.168.1.20:80
Hellovagrant@ubuntu-xenial:~$
vagrant@host-2:~$ curl 192.168.1.20:8080
curl: (7) Failed to connect to 192.168.1.20 port 8080: Connection refused
vagrant@ubuntu-xenial:~$
```

То же самое произойдет, если host-2 попытается получить доступ к контейнеру через его IP-адрес и порт либо Docker, либо хоста. Напомним, что Docker использует диапазон частных адресов, 172.17.0.0/16:

```
vagrant@host-2:~$ curl 172.17.0.2:8080 -t 5
curl: (7) Failed to connect to 172.17.0.2 port 8080: No route to host
vagrant@host-2:~$ curl 172.17.0.2:80 -t 5
curl: (7) Failed to connect to 172.17.0.2 port 80: No route to host
vagrant@host-2:~$
```

Для соединения с IP-адресом хост Docker использует оверлейную сеть или внешнюю по отношению к Docker маршрутизацию. Маршрутизация является внешней и в Kubernetes, многие CNI помогают в этом вопросе, подробно мы рассмотрим этот аспект в главе 6.

В предыдущих примерах использовался сетевой мост Docker, конфигурируемый по умолчанию, с открытыми портами на хостах. Он обеспечил взаимодействие host-2 с контейнером Docker, запущенном на хосте Docker. В данной главе мы весьма поверхностно рассмотрели контейнерные сети. Существует еще множество аспектов этих сетей, например входящий и исходящий трафик в кластере, поиск сервисов, внешняя и внутренняя маршрутизация в кластере, которые мы постепенно будем изучать в последующих главах.

Заключение

В данной главе, начальной в нашем изучении сетевых контейнеров, мы познакомились с тем, как контейнеры помогают развернуть приложение и повысить эффективность использования вычислительной техники путем запуска нескольких изолированных приложений на одном хосте. Мы также рассмотрели историю развития контейнерных технологий с ее различными проектами. Контейнеры используют в своей работе контрольные группы и пространства имен — механизмы ядра Linux. Мы также коснулись абстракций, предоставляемых разработчикам средой запуска контейнеров, и научились их применять на практике. Понимание этих абстракций ядра Linux очень важно для выбора подходящего CNI на основе баланса его достоинств и недостатков. Изложенный материал поможет администраторам сетей в понимании того, каким образом среда запуска контейнера управляет сетевыми абстракциями Linux.

Итак, мы закончили знакомство с основами сетей контейнеров. Мы прошли путь от использования простого сетевого стека до запуска нескольких не связанных между собой стеков внутри контейнеров. Понимание того, как работают пространства имен, как открываются порты, как осуществляется взаимодействие, помогает администраторам быстро находить и устранять проблемы, уменьшая тем самым время простоя для приложений, запущенных в кластерах Kubernetes. Исправление ошибок, связанных с портами, и проверка, является ли порт открытым на данном хосте, в контейнере или в сети, являются обязательными навыками для любого сетевого инженера и разработчика контейнеризованных приложений. Эти основы являются также и основами, на которых построен Kubernetes и который абстрагирует их для разработчиков. В следующей главе мы рассмотрим, как Kubernetes создает соответствующие абстракции и включает их в свою сетевую модель.

ГЛАВА 4

Сети в Kubernetes

Теперь, когда мы рассмотрели основные сетевые компоненты Linux и контейнеров, настало время для более подробного знакомства с тем, как доступ к сети обеспечивается в Kubernetes. В данной главе мы обсудим, каким образом поды связываются с кластером извне и изнутри. Также увидим, как осуществляется соединение между внутренними компонентами Kubernetes. Высокоуровневые абстракции (ингрессы), относящиеся к обнаружению сервисов и балансировке нагрузки, будут рассмотрены в следующей главе.

Сети в Kubernetes решают следующие задачи:

- Взаимодействие между сильносвязанными контейнерами.
- Взаимодействие между подами.
- Взаимодействие между подом и сервисом.
- Взаимодействие между внешним компонентом и сервисом.

Сетевая модель Docker по умолчанию использует виртуальную мостовую сеть, определяемую для хоста и являющуюся частной сетью, к которой присоединяются контейнеры. IP-адрес контейнера — это частный IP-адрес, в результате чего контейнеры, запущенные на различных машинах, не могут взаимодействовать друг с другом. Поэтому в Docker разработчики должны связывать порты хоста с портами контейнера, а затем проксировать трафик, чтобы обеспечить взаимодействие между узлами. В этой схеме ответственными за отсутствие конфликтов между портами контейнеров выступают администраторы Docker, обычно это задача системных администраторов. Kubernetes предлагает свое решение данной проблемы.

Сетевая модель Kubernetes

Сетевая модель Kubernetes изначально поддерживает многохостовую кластерную сеть. Поды взаимодействуют друг с другом по умолчанию, вне зависимости от того, на каком хосте они развернуты. Kubernetes базируется на проекте CNI с целью выполнения следующих требований:

- Все контейнеры должны взаимодействовать друг с другом без использования NAT.
- Узлы могут взаимодействовать с контейнерами без использования NAT.
- IP-адрес контейнера является тем же, каким его видят извне.

Рабочий объект в Kubernetes называется *под* (от *англ.* pod — автономный блок какого-либо устройства). Под содержит один или несколько контейнеров, запущенных «совместно» на одном и том же узле. Такая связность позволяет отдельным функциям сервиса размещаться в разных контейнерах. Например, разработчик может выбрать для запуска сервиса один контейнер, а для переадресовки журналов — другой. Запуск процессов в разных контейнерах дает им возможность использовать различные ограничения по ресурсам (например, «переадресовщик журналов не может использовать больше 512 MB памяти»). Это также позволяет разделить механизмы сборки и развертывания контейнера путем уменьшения ресурса, требуемого для сборки.

Код ниже представляет собой минимальное определение для пода — мы опустили в нем многие опции. Kubernetes имеет возможность устанавливать различные поля, как, например, статус пода, который является неизменяемым (read-only):

```
apiVersion: v1
kind: Pod
metadata:
   name: go-web
   namespace: default
spec:
   containers:
   - name: go-web
   image: go-web:v0.0.1
   ports:
     - containerPort: 8080
     protocol: TCP
```

Пользователь Kubernetes обычно не создает под впрямую. Вместо этого он создает рабочую нагрузку (workload), например развертывание, которое и работает с подом в зависимости от установок в поле `spec`. В случае с развертыванием пользователь задает *шаблон* для подов (см. рис. 4.1), а также сколько он хочет, чтобы существовало подов (часто называемых *репликами*). Есть несколько соответствующих процедур, такие как `ReplicaSet` и `StatefulSet`, которые мы рассмотрим в следующей главе. Некоторые из них представляют собой абстракции промежуточных типов, в то время как другие работают непосредственно с подами. Существуют также сторонние приложения в форме пользовательских определений ресурса (CDR). Приложения в Kubernetes — это довольно сложная тема, и мы рассмотрим только ее основы и только в частях, относящихся к сетевому стеку.

Можно сказать, что поды как таковые являются эфемерными — они удаляются и заменяются новыми версиями себя самих. Краткость существования подов — это один из основных вызовов для разработчиков и операторов сетей, привыкших к более долговременным физическим или виртуальным вычислительным машинам. Статус локального диска, назначение на узлы и IP-адреса постоянно обновляются на протяжении всего жизненного цикла пода.

Под имеет уникальный IP-адрес, который является общим для всех контейнеров в поде. Первичной мотивацией для присвоения каждому поду IP-адреса было

стремление убрать ограничения по числу портов. В Linux только одна программа может связываться с заданным адресом, портом и протоколом. Если бы у подов не было их уникальных IP-адресов, то два пода на узле могли бы конкурировать за один и тот же порт (например, два веб-сервера, оба слушающих порт 80). Если бы адреса были одинаковыми, это потребовало бы задания определенной конфигурации при запуске, например ключа `-port`. Либо надо было бы писать специальный скрипт для модификации конфигурационного файла в случае использования стороннего программного обеспечения.

Рис. 4.1. Взаимосвязь между приложением Kubernetes Развертывание и подами

В некоторых случаях стороннее ПО вообще не может работать с указанными пользователем портами, что требует более сложной настройки вроде задания DNAT-правил в `iptables` для узла. Веб-серверы еще добавляют проблему, поскольку их программы обычно используют стандартные номера портов, как, например, 80 для HTTP и 443 для HTTPS. Чтобы обойти этот стандарт, требуется обратное проксирование через балансировщик нагрузки или явное задание портов со стороны пользователей сервиса (что гораздо проще осуществляется для внутренних систем, чем для внешних). Эта модель используется в некоторых системах, например в Berg от Google. Kubernetes выбрал модель «один под — один IP-адрес», чтобы облегчить разработчикам адаптацию и запуск стороннего ПО. К сожалению для нас всех, присвоение IP-адреса и маршрутизация для каждого пода *значительно* усложняют работу с кластером Kubernetes.

По умолчанию Kubernetes разрешает любой трафик внутрь или наружу пода. Такая пассивная связность означает среди прочего, что каждый под в кластере может соединиться с любым другим подом в том же кластере. Это легко может привести к перегрузке доступа, особенно если сервисы не используют аутентификацию или если злоумышленник завладеет паролями. См. более подробно в разделе «Распространенные CNI-плагины».

Поды, создаваемые и удаляемые вместе с их IP-адресами, могут представлять определенные проблемы для начинающих разработчиков, еще не привыкших к данной модели. Представим, что у нас есть маленький Kubernetes-сервис, развернутый в форме реплик на трех подах. Когда производится модификация образа контейнера, Kubernetes осуществляет автоматическое обновление, удаляя старые поды и создавая новые с использованием нового образа контейнера. Эти новые поды, ско-

рее всего, будут иметь новые IP-адреса, что сделает доступ к старым IP-адресам невозможным. Обычной ошибкой новичков является задание вручную IP-адреса пода в конфигурационном файле или записи DNS, что приводит к сбоям в работе системы. Предотвращение именно этой ошибки является задачей сервисов и конечных точек, которые будут рассмотрены в следующей главе.

Если под создается в явном виде, то можно задать его IP-адрес. StatefulSets — это встроенное приложение, предназначенное для работы с базами данных, которое поддерживает концепцию идентичности подов, т. е. задает новому поду тот же самый IP-адрес, какой был у старого пода. Есть еще примеры сторонних приложений, как правило в форме CRD, можно также написать программу CRD для конкретной задачи.

Пользовательские ресурсы (CR) — это определяемые пользователем расширения API в Kubernetes. Они позволяют разработчикам ПО индивидуализировать установку их программ в среде Kubernetes. Подробные инструкции по написанию CRD можно найти в документации.

На каждом узле Kubernetes запускается *Kubelet* — компонент, управляющий подами узла. Сетевой функционал в Kubelet обеспечивается взаимодействием между API и плагином CNI. Именно CNI-плагин управляет IP-адресами подов и обеспечивает сетевое соединение для индивидуальных контейнеров. Мы уже упоминали одноименный интерфейс CNI в предыдущей главе — он является стандартным сетевым интерфейсом для контейнеров. Причиной сделать CNI интерфейсом является желание иметь стандарт интероперабельности, в котором присутствуют многочисленные CNI-плагины. Плагин CNI отвечает за присвоение подам IP-адресов и маршрутизацию между всеми подами. Kubernetes не поставляется с CNI-плагинами по умолчанию, поэтому в стандартной установке Kubernetes поды не могут пользоваться сетью.

Начнем с рассмотрения, каким образом CNI делает возможным сетевое подключение подов и какие бывают конфигурации сети.

Узел и конфигурация сети подов

Кластер должен иметь контролируемую им группу IP-адресов, например 10.1.0.0/16, из которой будет назначаться IP-адрес для пода. В данном IP-адресном пространстве узлы и поды должны иметь третий уровень связности. Вспомним материал *главы 3*: связность третьего уровня, уровня Интернета, означает, что пакеты с IP-адресом маршрутизируются на хост, имеющий этот IP-адрес. Важно отметить, что способность доставлять *пакеты* является более фундаментальной, чем способность устанавливать соединение (функция 4-го уровня). Брандмауэры на уровне 4 могут разрешить соединение хоста A с хостом B, но запретить соединения с хостом A, исходящие от хоста B. Соединения уровня 4, A с B, соединения уровня 3, A с B и B с A, должны быть разрешены. Без связности на уровне 3 невозможно TCP-квитирование, поскольку сигналы SYN-ACK не могут быть получены.

В общем случае поды не имеют MAC-адресов, т. е. связность на уровне 2 для подов невозможна. Эту функцию для подов берет на себя CNI.

Kubernetes не предоставляет связь с внешним миром на уровне 3. Хотя большинство кластеров имеют выход в Интернет, некоторые из них все же предпочитают изолировать из соображений безопасности.

Мы подробно обсудим как ингресс (трафик, выходящий из кластера или хоста), так и игресс (egress) — входной трафик хоста или кластера. В данном случае использование понятия «ингресс» не следует путать с одноименным ресурсом в Kubernetes, являющимся HTTP-механизмом для маршрутизации трафика к сервисам Kubernetes.

Есть три основных подхода с многочисленными вариациями к структурированию кластерной сети: изолированная сеть, плоская сеть и островная сеть. Мы сначала дадим общее описание этих подходов, а затем рассмотрим конкретные реализации, когда будем изучать CNI-плагины в данной главе ниже.

Изолированные сети

В *изолированной кластерной сети* узлы доступны для компонентов более широкой сети (т. е. хосты, не являющиеся частью кластера, могут получать доступ к узлам в кластере), но этот доступ отсутствует для подов. Такой кластер показан на рис. 4.2. Обратите внимание, что поды не могут связываться с другими подами (или другими хостами), находящимися вне кластера.

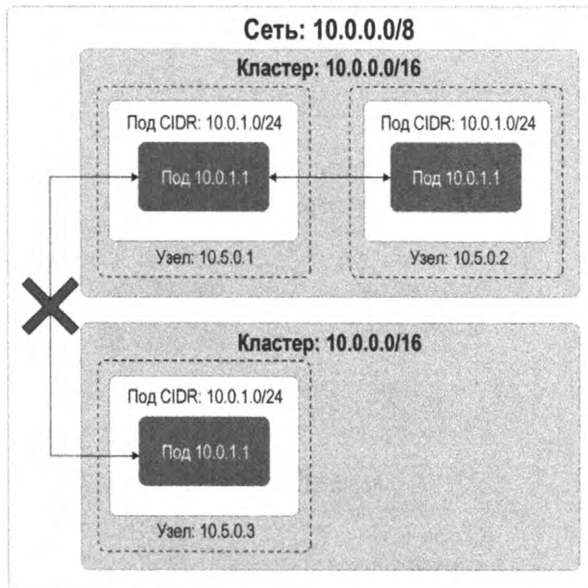


Рис. 4.2. Два изолированных кластера в одной сети

Поскольку более широкая сеть не маршрутизирует кластер, то разные кластеры могут использовать одно и то же пространство IP-адресов. Отметим, что API-сервер Kubernetes должен быть маршрутизируем из широкой сети, если внешним системам или пользователям разрешается доступ к API Kubernetes. Многие провайдеры

Kubernetes имеют опцию «безопасный кластер», при которой прямой трафик между кластером и Интернетом закрыт.

Такое изолирование локального кластера очень выгодно с точки зрения безопасности, если рабочие приложения кластера позволяют либо требуют данную конфигурацию сети, например кластеры для пакетной обработки. Однако эта конфигурация не является оптимальной для всех кластеров. Большинству кластеров все же требуется доступ к внешним системам, или они сами должны быть доступны извне, как, например, кластеры с поддержкой сервисов, которые зависят от Интернета. Чтобы преодолеть эти барьеры и обеспечить трафик внутрь и наружу изолированного кластера, используются балансировщики нагрузки и прокси.

Плоские сети

В *плоской сети* все поды имеют IP-адрес, к которому есть доступ из более широкой сети. Независимо от правил брандмауэров, любой хост сети может связываться с любым подом внутри или вне кластера. Такая конфигурация имеет множество плюсов с точки зрения простоты и производительности. Поды могут соединяться с любыми хостами сети.

Обратите внимание на рис. 4.3, что CIDR подов никаких двух узлов не пересекаются, поэтому никаким двум подам не может быть назначен один и тот же IP-адрес. Поскольку более широкая сеть может маршрутизировать IP-адрес каждого пода на узел этого пода, то любой хост сети доступен для пода, а под имеет доступ к любому хосту.

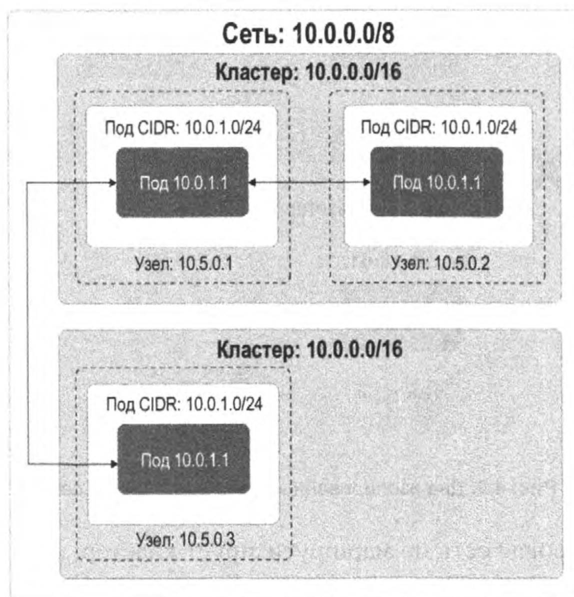


Рис. 4.3. Два кластера в плоской сети

Такая открытость позволяет хостам с достаточным объемом данных по обнаружению сервисов (service discovery) решать, какой под получит пакеты. Балансировщик нагрузки снаружи кластера может распределять нагрузку на поды.

Выходной трафик пода (а также входной трафик — если назначением соединения является IP-адрес конкретного пода) имеет низкую задержку и малые расходы на обработку. Любой тип проксирования или модификации пакета влечет за собой повышение задержки и увеличение обработки, которые хотя и малы, но не пренебрегаемы (особенно в приложениях, которые включают в себя много бэкенд-сервисов, где каждая задержка суммируется).

К сожалению, данная модель требует большого непрерывного пространства IP-адресов для каждого кластера (т. е. диапазона IP-адресов, в котором вы контролируете каждый IP-адрес). Kubernetes требует один CIDR для IP-адресов пода (для каждого семейства IP-адресов). Это можно реализовать с помощью частной подсети (такой как 10.0.0.0/8 или 172.16.0.0/12), но довольно сложно и дорого, если использовать публичные IP-адреса, особенно IPv4. Администраторам придется прибегнуть к NAT, чтобы связать с Интернетом кластер, работающий в частном IP-адресном пространстве.

Помимо большого пространства IP-адресов, администраторам также понадобится легко программируемая сеть. CNI-плагин должен назначать IP-адреса пода и обеспечивать маршрутизацию на узел данного пода.

Плоские сети, использующие частную подсеть, легко реализовать в облачной среде. Подавляющее большинство облачных сетей предоставляют большие частные сети и имеют API (или даже уже существующие CNI-плагины) для назначения IP-адресов и управления маршрутизацией.

Островные сети

Кластерные островные сети — это, по существу, комбинация изолированных и плоских сетей.

В конфигурации островной кластерной сети, показанной на рис. 4.4, узлы имеют связность уровня 3 с более широкой сетью, а поды ее не имеют. Внешний и внутренний трафик подов проходит через узлы с использованием тех или иных прокси. Наиболее часто это реализуется через трансформацию (NAT) адреса источника в iptables для формируемых подом пакетов, которые покидают узел. Данная концепция, называемая *маскированием* (masquerading), использует SNAT, чтобы переписать источники пакетов с IP-адреса пода на IP-адрес узла (см. *главу 2*, чтобы освежить в памяти, что такое SNAT). Другими словами, пакеты кажутся исходящими из узла, хотя на самом деле они идут из пода.

Совместное использование IP-адреса и обращение к NAT скрывают IP-адреса отдельных подов. Тем самым усложняется реализация межкластерных брандмауэров, основанных на распознавании IP-адресов. Внутри кластера вполне понятно, какой IP-адрес к какому поду относится (и соответственно приложению). Для подов в других кластерах или на других хостах сети такое соответствие совсем не очевидно. Таким образом, брандмауэры на основе IP-адреса и списки допуска не

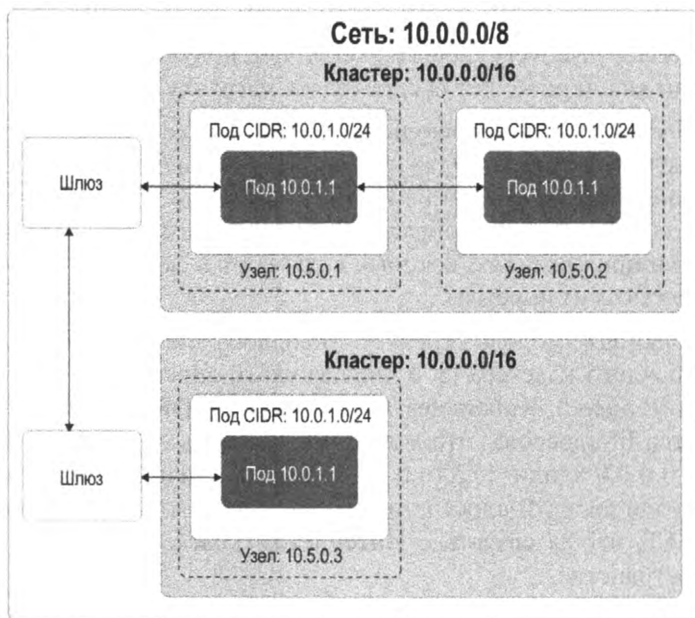


Рис. 4.4. Два кластера в конфигурации островной сети

являются достаточной защитной мерой, хотя в некоторых случаях они хорошо работают и даже являются необходимыми.

Посмотрим, каким образом каждая из рассмотренных сетей может быть сконфигурирована с помощью `kube-controller-manager`. *Управляющий уровень (Control plane)* объединяет все функции и процессы, определяющие, какой путь использовать для передачи пакета или кадра. К *передающему уровню (Data plane)* относятся все функции и процессы, которые пересылают пакеты/кадры от одного интерфейса на другой на основе логики управляющего уровня.

Конфигурация компонента kube-controller-manager

Компонент `kube-controller-manager` запускает большинство контроллеров Kubernetes в виде исполняемого файла и процесса, в котором содержится основная логика Kubernetes. Контроллер в Kubernetes — это высокоуровневая программа, отслеживающая ресурсы и выполняющая действия по синхронизации или достижению определенного состояния (желаемого состояния или отображая текущее состояние как статус). Как правило, многочисленные контроллеры Kubernetes работают с объектами определенного типа или выполняют специальные функции.

Компонент `kube-controller-manager` включает в себя несколько контроллеров, управляющих сетевым стеком Kubernetes. Отметим, что администраторы сетей задают здесь CIDR кластера.

Поскольку `kube-controller-manager` запускает несколько контроллеров, он имеет большое количество ключей. В табл. 4.1 перечислены наиболее употребительные опции для конфигурирования сети.

Таблица 4.1. *Опции компонента* `kube-controller-manager`

Опция	Значение по умолчанию	Описание
`--allocate-node-cidrs`	true	Задает, должны ли CIDR подов устанавливаться в облаке
`--CIDR-allocator-type`	RangeAllocator	Какой тип аллокатора CIDR использовать
`--cluster-CIDR`		Диапазон CIDR, из которого назначаются IP-адреса подов. Требует, чтобы `--allocate-node-cidrs` был установлен в true. Если в `kube-controller-manager` ключ `IPv6DualStack` установлен, то `--cluster-CIDR` принимает разделенную запятой пару `CIDR` типов IPv4 и IPv6
`--configure-cloud-routes`	true	Устанавливает, должны ли CIDR быть назначены через `--allocate-node-cidrs` и сконфигурированы в облаке
`--node-CIDR-mask-size`	24 для кластеров IPv4, 64 — для IPv6	Размер маски для CIDR-узла в кластере. Выделяет каждому узлу `2^(node-CIDR-mask-size)` IP-адресов
`--node-CIDR-mask-size-ipv4`	24	Размер маски для CIDR-узла в кластере. Предназначен для кластеров с параллельным использованием IPv4 и IPv6
`--node-CIDR-mask-size-ipv6`	64	Размер маски для CIDR-узла в кластере. Предназначен для кластеров с параллельным использованием IPv4 и IPv6
`--service-cluster-ip-range`		Диапазон CIDR для сервисов в кластере для назначения сервисам IP-адресов. Требует установки `--allocate-node-cidrs` в true. Если в `kube-controller-manager` активирован ключ `IPv6DualStack`, то `--service-cluster-ip-range` принимает разделенную запятой пару `CIDR` типов IPv4 и IPv6

Теперь, когда мы познакомились с архитектурой и конфигурированием сети на управляющем уровне Kubernetes, рассмотрим подробнее, каким образом узлы Kubernetes осуществляют сетевое взаимодействие.

Kubelet

Kubelet — это исполняемый файл, который запускается на каждом рабочем узле кластера. Kubelet отвечает за управление всеми назначенными на узел подами и за обновление статуса узла и его подов. Однако в первую очередь Kubelet на узле работает как координатор выполнения других программ. Kubelet контролирует сетевое взаимодействие контейнеров (через CNI) и среду запуска контейнеров (через CRI).

Мы определяем рабочие узла в Kubernetes как узлы, на которых могут запускаться поды. Некоторые кластеры по техническим причинам запускают API-серверы и etcd только на определенных рабочих узлах. Такая конфигурация позволяет контролировать компоненты управляющего уровня по тем же схемам, что и стандартные приложения Kubernetes, но несет с собой повышенный риск отказов и ухудшает безопасность.

Когда контроллер (или пользователь) создает под в Kubernetes API, то первоначально этот под существует только как объект API. Планировщик (scheduler) Kubernetes отслеживает такой под и пытается назначить его на подходящий узел. Эта операция имеет ограничения. Требования пода по ресурсам ЦПУ и памяти не должны превышать свободные ресурсы узла. Имеются также многочисленные выборные опции, такие как аффинность/анти-аффинность к помеченным (labeled) узлам и другим отмеченным подам или «неподходящие» (tainted) узлы. Когда планировщик находит узел, отвечающий параметрам пода, то он записывает имя этого узла в поле `nodeName` пода. Предположим, Kubernetes назначил под на узел `node-1`:

```
apiVersion: v1
kind: Pod
metadata:
  name: example
spec:
  nodeName: "node-1"
  containers:
    - name: example
      image: example:1.0
```

Kubelet на узле `node-1` отслеживает все назначенные на узел поды. Эквивалентной командой `kubectl` является `kubectl get pod -w -field-selector spec.nodeName=node-1`. Если Kubelet обнаруживает, что под существует, но не присутствует на узле, то он его создает. Здесь мы не будем касаться деталей интерфейса CRI и создания самого контейнера. Когда контейнер уже существует, то Kubelet выполняет ADD-вызов на интерфейс CNI, который дает задание плагину CNI создать сеть для пода. Эти интерфейсы и плагины рассматриваются в следующем разделе.

Готовность пода и ее проверка

Готовность пода — это индикатор, показывающий, может ли под обслуживать трафик. Готовность пода определяется тем, что его адрес в объекте `Endpoints` виден внешнему источнику. Другие управляющие подами ресурсы Kubernetes, например развертывание, учитывают готовность пода при выполнении своих операций, таких как advancing при плавающем (непрерывном) обновлении (rolling update). При плавающем развертывании новый под оказывается готовым, однако по каким-либо причинам сервис, сетевая политика или балансировщик нагрузки для него пока отсутствуют. Это может повлечь за собой сбой в выполнении сервиса или потерю функциональности бэкенда. Нужно отметить, что если в поле `spec` пода не предусмотрено какого-либо типа проверки, то Kubernetes по умолчанию требует положительного результата для всех трех типов.

Пользователи могут задать проверки готовности пода в поле `spec` пода. На этой основе Kubelet будет выполнять указанные тесты и обновлять статус пода в зависимости от результатов тестов.

Результаты тестов влияют на поле `.Status.Phase` пода. Ниже приводится список возможных значений состояния пода и их описание:

Pending (в ожидании)

Под существует в кластере, но один или несколько контейнеров еще не сконфигурированы и не готовы к запуску. Сюда включается время, которое под проводит в ожидании своего назначения на узел, а также время, требуемое на загрузку образа контейнера через сеть.

Running (в работе)

Под назначен на узел, и все контейнеры созданы. Как минимум один контейнер еще работает или находится в процессе запуска/перезапуска. Отметим, что некоторые контейнеры могут выдавать ошибку, например CrashLoopBackoff.

Succeeded (удачное завершение)

Все контейнеры в поде безаварийно завершились и не будут перезапускаться.

Failed (неудачное завершение)

Все контейнеры в поде завершились и как минимум один контейнер выдал ошибку. То есть контейнер или завершился с ненулевым статусом, или был остановлен системой.

Unknown (неизвестное состояние)

По каким-либо причинам состояние пода не может быть определено. Эта событие обычно возникает при ошибке взаимодействия с Kubelet на узле, где должен быть запущен под.

Kubelet выполняет несколько типов тестов для контейнеров в поде: проверка живучести (`livenessProbe`), проверка готовности (`readinessProbe`), тест на запуск (`startupProbe`). Kubelet (и сам узел) должны быть способны устанавливать соединение со всеми контейнерами, запущенными на данном узле, чтобы выполнять любое тестирование HTTP.

Результаты тестов бывают трех типов:

Success (Успешно)

Контейнер прошел тест.

Failure (неуспешно)

Контейнер не прошел тест.

Unknown (неизвестно)

Тест прерван, дальнейшие действия не могут выполняться.

Тестами могут выступать исполняемые файлы, которые запускаются в контейнере, проверки TCP или проверки HTTP. Если проверка оканчивается неудачей

`failureThreshold` раз, то Kubernetes будет считать, что тест провалился. Что будет дальше, зависит от вида теста.

Если не прошел тест на готовность контейнера, то Kubelet не будет прекращать его работу, а сделает запись об отрицательном результате в статусе пода.

Если не прошел тест на живучесть, то Kubelet прекращает работу контейнера. Проверка живучести легко может привести к неожиданным сбоям, если она неправильно используется или неправильно сконфигурирована. Данный тест не предназначен для того, чтобы сообщать Kubelet, когда производить повторный запуск контейнера. Однако с опытом мы начинаем понимать, что принцип «если что-то работает неправильно, то сделай перезагрузку» может быть довольно опасным. Например, предположим, что мы сконфигурировали тест на живучесть, который загружает главную страницу нашего веб-приложения. Далее предположим, что какие-то изменения в системе — вне кода нашего контейнера — приводят к тому, что главная страница возвращает ошибку 404 или 500. Причинами такого поведения могут быть, например, отказы базы данных на бэкенде, сбой в запрашиваемом сервисе или задание некого условия, приводящего к ошибке в программе. В каждом из этих случаев проверка живучести будет приводить к перезапуску контейнера. В лучшем случае это просто не поможет: перезапуск контейнера не решит проблему, возникающую где-то в системе. В Kubernetes есть механизмы отсрочки перезапуска контейнера (`CrashLoopBackoff`), которые добавляют все увеличивающиеся задержки перед повторным запуском сбойных контейнеров. При достаточном количестве подов или при коротких промежутках между сбоями приложение может перейти из состояния ошибки на домашней странице в состояние полного отказа от работы. В зависимости от приложения при перезагрузке поды могут потерять кешированные данные, восстановление которых будет сложно осуществить. Поэтому планируйте тесты на живучесть с большой осторожностью. Когда поды используют данные тесты, они находятся в зависимости только от тестируемого контейнера и больше ни от чего. Многие инженеры имеют свои наработки по тестированию, направленные на применение более мягких критериев типа «PHP работает и обслуживает мой API».

Тест на запуск может предоставлять период времени перед тем, как проявится результат теста на живучесть, а именно: тест на живучесть не будет завершать работу контейнера прежде, чем не закончится тест на запуск. Пример использования — предоставить контейнеру несколько минут для запуска, но быстро прекратить его работу, если после запуска он становится сбойным.

В примере 4.1 наш веб-сервер на Go включает в себя тест на живучесть, выполняющий команду HTTP GET с порта 8080 на путь /healthz, а проверка готовности использует путь / на том же порте.

Пример 4.1. Поле spec пода в Kubernetes для минимального веб-сервера на языке Go

```
apiVersion: v1
kind: Pod
```

```
metadata:
  labels:
    test: liveness
  name: go-web
spec:
  containers:
  - name: go-web
    image: go-web:v0.0.1
    ports:
    - containerPort: 8080
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
      initialDelaySeconds: 5
          periodSeconds: 5
    readinessProbe:
      httpGet:
        path: /
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 5
```

Модифицированный в зависимости от результата теста статус не влияет на сам под, но другие механизмы Kubernetes на него реагируют. Примером является объект `ReplicaSet` (или, более широко, компонент «развертывание»). Отрицательный результат теста на готовность указывает контроллеру `ReplicaSet` на неготовность пода к приему трафика, что может привести к прекращению развертывания, если достаточное количество подов оказываются дефектными. Контроллеры `Endpoint/EndpointsSlice` также реагируют на провалы тестов готовности. Если данный тест для пода отрицателен, то IP-адрес пода не будет присутствовать в объекте конечных точек `Endpoint` и соответственно сервис не будет направлять на него трафик. Мы обсудим сервисы и конечные точки в следующей главе.

Тест `StartupProbe` информирует Kubelet, запущено ли приложение в контейнере. Этот тест является привилегированным по сравнению с другими. Если он задан в поле `spec` пода, то все остальные тесты деактивируются. Если тест `StartupProbe` проходит, то Kubelet начинает запускать остальные тесты. Но если этот тест не проходит, то Kubelet уничтожает контейнер, и выполняется перезагрузка контейнера. Как и в прочих случаях, если выполнение теста `StartupProbe` не задано, то по умолчанию его результат считается положительным.

Конфигурируемые опции тестов:

initialDelaySeconds

Количество секунд после запуска контейнера, через которые инициируются проверки на живучесть и готовность. По умолчанию 0, минимальное значение 0.

periodSeconds

Как часто выполняются проверки. По умолчанию 10, минимальное значение 1.

timeoutSeconds

Количество секунд, после которых тест прерывается. По умолчанию 1, минимальное значение 1.

successThreshold

Минимальное количество последовательных положительных — после отрицательного исхода — результатов теста, чтобы он считался выполненным. По умолчанию 1, для тестов на живучесть и на запуск должна устанавливаться 1, минимальное значение 1.

failureThreshold

Если тест не проходит, то Kubernetes будет повторять его несколько раз, прежде чем прекратить операцию. Прекращение операции в случае теста на живучесть означает, что контейнер будет перезапущен. В случае теста на готовность контейнер будет помечен как неготовый (Unready). По умолчанию 3, минимальное значение 1.

Разработчики приложений могут использовать индикаторы готовности, чтобы определять готовность приложения внутри пода. Опция имеется, начиная с версии Kubernetes 1.14, для ее реализации необходимо добавить в поле `spec` пода команду `readiness gates`, после которой задать список дополнительных условий, проверяемых модулем Kubelet для определения готовности пода. Они задаются через атрибут `ConditionType` — типом условия в списке условий пода. Значение `readiness gates` контролируется текущим статусом поля пода `status.condition`. Если Kubelet не может найти данное условие в указанном поле пода, то статус условия по умолчанию устанавливается в False.

Как можно видеть из следующего примера, индикатор готовности `feature-Y` установлен в true, в то время как индикатор `feature-X` установлен в `false`, таким образом статус пода также оказывается `false`:

```
kind: Pod
...
spec:
readinessGates:
- conditionType: www.example.com/feature-X
- conditionType: www.example.com/feature-Y
...
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: 2021-04-25T00:00:00Z
    status: "False"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: 2021-04-25T00:00:00Z
```

```
    status: "False"
    type: www.example.com/feature-X
  - lastProbeTime: null
    lastTransitionTime: 2021-04-25T00:00:00Z
    status: "True"
    type: www.example.com/feature-Y

containerStatuses:
  - containerID: docker://xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
    ready : true
```

Балансировщики нагрузки типа AWS ALB могут использовать индикатор готовности в качестве параметра жизненного цикла пода и учитывать его при маршрутизации трафика.

Kubelet должен быть способен связываться с API-сервером Kubernetes. На рис. 4.5 показаны все соединения, устанавливаемые всеми компонентами в кластере:

CNI

Сетевой плагин в Kubelet, который дает возможность сети получать IP-адреса для подов и сервисов.

gRPC

API для взаимодействия между API-сервером и etcd.

Kubelet

Все узлы Kubernetes запускают Kubelet, который обеспечивает, что все назначенные на узел поды работают и сконфигурированы в желаемом состоянии.

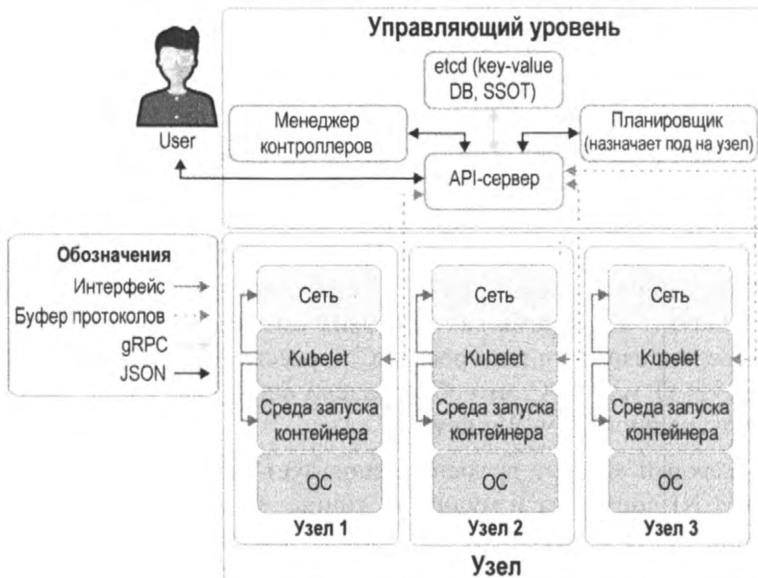


Рис. 4.5. Поток данных между компонентами в кластере

CRI

Скомпилированный в Kubelet интерфейс gRPC API, который обеспечивает взаимодействие между Kubelet и средой запуска контейнера. Провайдер среды запуска должен адаптировать ее к CRI API, чтобы сделать возможной связь между Kubelet и контейнером по стандарту OCI (runC). CRI состоит из буферов протоколов, библиотек и gRPC API.

Взаимодействие между подами и Kybelet обеспечивается интерфейсом CNI. В следующем разделе мы рассмотрим спецификацию CNI с примерами, взятыми из различных CNI-проектов.

Спецификация интерфейса CNI

Спецификация CNI сама по себе довольно простая. Согласно спецификации CNI-плагин должен поддерживать четыре операции:

ADD

Добавить контейнер к сети.

DEL

Удалить контейнер из сети.

CHECK

Вернуть ошибку, если возникла проблема с контейнерной сетью.

VERSION

Сообщает версию документации плагина.

Полная спецификация CNI доступна на GitHub.

На рис. 4.6 показано, как Kubernetes (или *среда запуска*, поскольку CNI обращается к оркестраторам контейнеров) вызывает операции CNI-плагина путем выполнения бинарного файла. Kubernetes передает произвольную конфигурацию в `stdin` через команду JSON и получает результат (JSON) из `stdout`. Плагины CNI обычно имеют очень простые исполняемые файлы, представляющие собой вызываемую Kubernetes оболочку, в то время как сам файл вызывает HTTP или RPC API с постоянного сервера. Разработчики проекта CNI рассматривали переход от такой модели к модели HTTP или RPC, что обусловлено проблемами в производительности при частом запуске процессов Windows.

Kubernetes в каждый момент времени использует только один CNI-плагин, хотя спецификация CNI допускает и мультиплагинные установки (т. е. назначение контейнеру нескольких IP-адресов). Multus — это плагин CNI, позволяющий обойти это ограничение в Kubernetes и работающий как разветвитель для других CNI-плагинов.

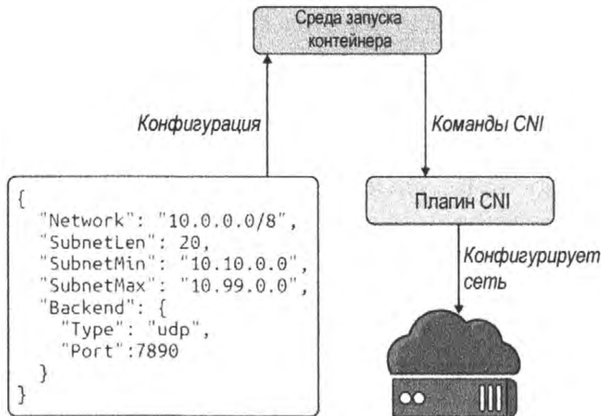
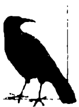


Рис. 4.6. Конфигурация CNI



Ко времени написания данной книги спецификация CNI имела версию 0.4. Она не сильно изменилась за прошедшие годы и, скорее всего, мало изменится в будущем — разработчики спецификации вскоре планируют выпустить версию 1.0.

Плагины CNI

Плагин CNI имеет две основные функции: назначать подам уникальные IP-адреса и обеспечивать внутри Kubernetes маршруты к этим адресам. Это означает, что общая сеть, в которой находится кластер, определяет поведение CNI-плагина. Например, если имеется мало IP-адресов или невозможно выделить узлу достаточное число адресов, то администраторам кластера придется воспользоваться CNI-плагином, поддерживающим оверлейную сеть. Обычно, имеющиеся вычислительные мощности или возможности облачного провайдера диктуют, какие требуются опции CNI. В главе 6 мы поговорим об основных облачных платформах и каким образом архитектура сети определяет выбор компонентов CNI.

Чтобы использовать CNI, добавьте `–network-plugin=cni` в аргументы файла запуска Kubelet. По умолчанию Kubelet считывает конфигурацию CNI из директории `/etc/cni/net.d/` и ожидает найти исполняемый CNI-файл в директории `/opt/cni/bin/`. Изменить место нахождения конфигурации можно командой `--cni-config-dir=<directory>`, а место нахождения бинарного файла — командой `--cni-bin-dir=<directory>`.

Многие дистрибутивы Kubernetes поставляются с предварительно сконфигурированным CNI.

Существуют две основные категории сетевых CNI-моделей: плоские сети и оверлейные сети. В случае плоской сети CNI-драйвер использует IP-адреса кластерной сети, что в общем случае требует большого числа доступных в кластере IP-адресов. В случае оверлейной сети CNI-драйвер создает внутри Kubernetes вторичную сеть,

которая использует для передачи пакетов кластерную сеть (называемую *базовой сетью)*. То есть оверлейные сети образуют внутри кластера виртуальную сеть. В оверлейной сети CNI-плагин инкапсулирует пакеты. Мы подробно обсуждали этот тип сетей в *главе 3*. Оверлейные сети сильно усложняют конструкцию и не позволяют хостам кластерной сети непосредственно связываться с подами. Однако оверлейные сети делают кластерную сеть гораздо компактнее, поскольку IP-адреса присваиваются только узлам.

Плагины CNI обычно требуют какие-то средства для обеспечения взаимодействия между узлами. Плагины используют при этом весьма отличающиеся подходы, такие как сохранение данных в API Kubernetes или в специальной базе данных.

Функцией CNI-плагина также является вызов IPAM-плагинов для управления IP-адресацией.

Интерфейс IPAM

Спецификация CNI предусматривает второй интерфейс — интерфейс управления IP-адресом, IPAM, чтобы не повторять в плагинах CNI- код, описывающий выделение адресов. Как показано в примере 4.2, плагин IPAM определяет и выдает IP-адрес интерфейса, шлюз и маршруты. Интерфейс IPAM похож на интерфейс CNI: он является исполняемым файлом с JSON-вводом через `stdin` и JSON-выводом через stdout.

Пример 4.2. Вывод плагина IPAM (взято из документации по CNI спецификации 0.4)

```
{
"cniVersion": "0.4.0",
"ips": [
        {
         "version": "<4-or-6>",
         "address": "<ip-and-prefix-in-CIDR>",
         "gateway": "<ip-address-of-the-gateway>"      (optional)
        },
...
],
"routes": [                                            (optional)
        {
         "dst": "<ip-and-prefix-in-cidr>",
         "gw": "<ip-of-next-hop>"                      (optional)
        },
...
]
"dns": {                                               (optional)
        "nameservers": <list-of-nameservers>           (optional)
        "domain": <name-of-local-domain>               (optional)
        "search": <list-of-search-domains>             (optional)
        "options": <list-of-options>                   (optional)
 }
}
```

Далее мы рассмотрим опции, имеющиеся в распоряжении администраторов кластера при установке CNI.

Распространенные плагины CNI

Cilium — это программа с открытым кодом, предназначенная для обеспечения безопасности сетевого взаимодействия между контейнерами приложений. Cilium является плагином CNI с поддержкой HTTP (7-й уровень) и может обеспечивать сетевые политики на уровнях 3–7, реализуя модель безопасности на основе идентификации без сетевой адресации. Базой Cilium служит технология Linux eBPF, которую мы рассматривали в *главе 2*. Ниже в данной главе мы подробно обсудим объекты `NetworkPolicy`, сейчас же отметим только, что они фактически представляют собой брандмауэры для подов.

Flannel работает с сетью и предлагает простой и доступный способ для конфигурирования сетевых компонентов третьего уровня, предназначенных для Kubernetes. Если кластер требует функций вроде сетевых политик, то администратор должен инсталлировать другие плагины CNI, например Calico. Flannel использует имеющийся в Kubernetes-кластере `etcd` для хранения данных о состоянии кластера и, таким образом, не обращается к внешним хранилищам.

Согласно описанию плагина Calico «он соединяет гибкую работу в сети с универсальной поддержкой безопасности, предоставляя решения с производительностью ядра Linux и масштабируемостью облака». Calico не использует оверлейную сеть, а конфигурирует сеть 3-го уровня, которая для передачи пакетов между хостами использует протокол маршрутизации BGP. Calico может интегрироваться с Istio, реализацией service mesh, чтобы обеспечивать сетевую политику для приложений внутри кластера на уровне service mesh и инфраструктурном уровне сети.

В табл. 4.2 приводится список основных плагинов CNI.

***Таблица 4.2.** Краткая характеристика основных плагинов CNI*

Название	**Поддержка сетевой политики**	**Хранение данных**	**Конфигурация сети**
Cilium	Да	etcd или consul	Ipvlan(beta), veth, поддержка 7-го уровня
Flannel	Нет	etcd	Оверлейная сеть 3-го уровня, IPv4
Calico	Да	etcd или Kubernetes API	Сеть 3-го уровня с использованием BGP
Weave Net	Да	Нет внешнего кластерного хранилища	Ячеистая оверлейная сеть

Полная документация по KIND, Helm или Cilium содержится в публикациях на GitHub.

В примере 4.3 мы устанавливаем Cilium, чтобы протестировать наш веб-сервер. Для установки Cilium нам понадобится кластер Kubernetes. Наиболее простой способ развернуть кластер для локальной работы — использовать KIND, который является аналогом Kubernetes в Docker. Он позволит нам создать кластер с помощью конфигурационного файла на YAML, а затем, используя Helm, установить в этом кластере плагин Cilium.

Пример 4.3. Конфигурация KIND для установки Cilium в локальном кластере

```
kind: Cluster ❶
apiVersion: kind.x-k8s.io/v1alpha4 ❷
nodes: ❸
- role: control-plane ❹
- role: worker ❺
- role: worker ❻
- role: worker ❼
networking: ❽
disableDefaultCNI: true ❾
```

Пояснения:

❶ Указывает, что мы создаем кластер KIND.

❷ Версия KIND.

❸ Список узлов в кластере.

❹ Узел управляющей панели.

❺ Рабочий узел 1.

❻ Рабочий узел 2.

❼ Рабочий узел 3.

❽ Сетевые опции KIND.

❾ Деактивирует сетевую опцию по умолчанию, чтобы можно было установить Cilium.

Инструкции по конфигурированию кластера KIND и другую информацию можно найти в соответствующей документации.

Используя YAML-файл для конфигурирования кластера, мы можем создать кластер с помощью следующей команды. Если она выполняется в первый раз, то потребуется определенное время для загрузки всех образов Docker для рабочей и управляющей панели:

```
$ kind create cluster --config=kind-config.yaml
Creating cluster "kind" ...
✓ Ensuring node image (kindest/node:v1.18.2)
```

```
Preparing nodes
✓ Writing configuration Starting control-plane
Installing StorageClass Joining worker nodes Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Have a question, bug, or feature request?
Let us know! https://kind.sigs.k8s.io/#community ⍰ ---

Always verify that the cluster is up and running with kubectl.

$ kubectl cluster-info --context kind-kind
Kubernetes master -> control plane is running at https://127.0.0.1:59511
KubeDNS is running at
https://127.0.0.1:59511/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
To further debug and diagnose cluster problems, use 'kubectl cluster-info dump.'
```

Узлы кластера остаются в состоянии NotReady все время, пока Cilium будет устанавливать сеть. Это нормальное поведение для кластера.

Запустив локальный кластер, перейдем к установке Cilium, используя для этого Helm — установочный инструмент Kubernetes. Согласно документации Helm является рекомендуемым инструментом для установки плагина Cilium. В первую очередь нам надо добавить библиотеку Helm для Cilium. Либо можно загрузить образы Docker для Cilium, а затем выполнить команду KIND, чтобы загрузить образы Cilium в кластер:

```
$ helm repo add cilium https://helm.cilium.io/
# Pre-pulling and loading container images is optional.
$ docker pull cilium/cilium:v1.9.1
kind load docker-image cilium/cilium:v1.9.1
```

Теперь, когда все предварительные операции для Cilium завершены, мы можем установить его в нашем кластере, используя Helm. Для Cilium существует множество конфигурационных опций, Helm задает опции с помощью команды `--set NAME_VAR=VAR`:

```
$ helm install cilium cilium/cilium --version 1.10.1 \
--namespace kube-system

NAME: Cilium
LAST DEPLOYED: Fri Jan 1 15:39:59 2021
NAMESPACE: kube-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

```
NOTES:
You have successfully installed Cilium with Hubble.

Your release version is 1.10.1.

For any further help, visit https://docs.cilium.io/en/v1.10/gettinghelp/
```

Cilium устанавливает в кластере несколько компонентов: агента, клиента, оператора и плагин `cilium-cni`:

Агент

Агент Cilium, `cilium-agent`, запускается на каждом узле кластера. Конфигурирование агента производится с помощью Kubernetes API, при этом задаются параметры сети, балансировка нагрузки сервисов, сетевые политики, видимость и параметры управления.

Клиент (CLI)

Клиент Cilium (CLI) — это инструмент командной строки, инсталлируемый вместе с агентом Cilium. Он взаимодействует с REST API на том же узле. CLI дает возможность разработчикам отслеживать состояние и статус локального агента. Он также предоставляет инструменты для доступа к картам eBPF, чтобы напрямую контролировать их состояние.

Плагин CNI

Плагин CNI (`cilium-cni`) взаимодействует на узле с Cilium API, чтобы предоставить подам доступ к сети, балансировку нагрузки и сетевые политики.

С помощью команды `kubectl -n kube-system get pods -watch` можно отслеживать установку в кластере всех этих компонентов:

```
$ kubectl -n kube-system get pods --watch
NAME                                          READY   STATUS
cilium-65kvp                                  0/1     Init:0/2
cilium-node-init-485lj                        0/1     ContainerCreating
cilium-node-init-79g68                        1/1     Running
cilium-node-init-gfdl8                        1/1     Running
cilium-node-init-jz8qc                        1/1     Running
cilium-operator-5b64c54cd-cgr2b               0/1     ContainerCreating
cilium-operator-5b64c54cd-tblbz               0/1     ContainerCreating
cilium-pg6v8                                  0/1     Init:0/2
cilium-rsnqk                                  0/1     Init:0/2
cilium-vfhrs                                  0/1     Init:0/2
coredns-66bff467f8-dqzql                      0/1     Pending
coredns-66bff467f8-r5nl6                      0/1     Pending
etcd-kind-control-plane                       1/1     Running
kube-apiserver-kind-control-plane             1/1     Running
kube-controller-manager-kind-control-plane    1/1     Running
kube-proxy-k5zc2                              1/1     Running
kube-proxy-qzhvq                              1/1     Running
```

```
kube-proxy-v54p4                              1/1     Running
kube-proxy-xb9tr                              1/1     Running
kube-scheduler-kind-control-plane             1/1     Running
cilium-operator-5b64c54cd-tblbz               1/1     Running
```

После того как мы установили Cilium, можно провести проверку сетевого соединения, чтобы убедиться в правильности работы Cilium:

```
$ kubectl create ns cilium-test
namespace/cilium-test created

$ kubectl apply -n cilium-test \
-f \
https://raw.githubusercontent.com/strongjz/advanced_networking_code_examples/
master/chapter-4/connectivity-check.yaml

deployment.apps/echo-a created
deployment.apps/echo-b created
deployment.apps/echo-b-host created
deployment.apps/pod-to-a created
deployment.apps/pod-to-external-1111 created
deployment.apps/pod-to-a-denied-cnp created
deployment.apps/pod-to-a-allowed-cnp created
deployment.apps/pod-to-external-fqdn-allow-google-cnp created
deployment.apps/pod-to-b-multi-node-clusterip created
deployment.apps/pod-to-b-multi-node-headless created
deployment.apps/host-to-b-multi-node-clusterip created
deployment.apps/host-deployment.apps/pod-to-b-multi-node-nodeport created
deployment.apps/pod-to-b-intra-node-nodeport created
service/echo-a created
service/echo-b created
service/echo-b-headless created
service/echo-b-host-headless created
ciliumnetworkpolicy.cilium.io/pod-to-a-denied-cnp created
ciliumnetworkpolicy.cilium.io/pod-to-a-allowed-cnp created
ciliumnetworkpolicy.cilium.io/pod-to-external-fqdn-allow-google-cnp created
```

При проверке сетевого соединения происходит установка ряда объектов Kubernetes deployment, которые используют различные пути подключения. Пути подключения тестируются с балансировкой нагрузки на сервисы и без нее, а также с различными комбинациями сетевых политик. Имя пода показывает вариант сетевого соединения, а индикаторы живучести и готовности указывают на положительный или отрицательный результат тестирования:

```
$ kubectl get pods -n cilium-test -w
NAME                                          READY   STATUS
echo-a-57cbbd9b8b-szn94                       1/1     Running
echo-b-6db5fc8ff8-wkcr6                       1/1     Running
echo-b-host-76d89978c-dsjm8                   1/1     Running
```

```
host-to-b-multi-node-clusterip-fd6868749-7zkcr            1/1     Running
host-to-b-multi-node-headless-54fbc4659f-z4rtd            1/1     Running
pod-to-a-648fd74787-x27hc                                 1/1     Running
pod-to-a-allowed-cnp-7776c879f-6rq7z                      1/1     Running
pod-to-a-denied-cnp-b5ff897c7-qp5kp                       1/1     Running
pod-to-b-intra-node-nodeport-6546644d59-qkmck             1/1     Running
pod-to-b-multi-node-clusterip-7d54c74c5f-4j7pm            1/1     Running
pod-to-b-multi-node-headless-76db68d547-fhlz7             1/1     Running
pod-to-b-multi-node-nodeport-7496df84d7-5z872             1/1     Running
pod-to-external-1111-6d4f9d9645-kfl4x                     1/1     Running
pod-to-external-fqdn-allow-google-cnp-5bc496897c-bnlqs 1/1     Running
```

Итак, теперь Cilium управляет нашей сетью в кластере, ниже мы еще раз обратимся к нему при рассмотрении объекта `NetworkPolicy`. Не все плагины CNI поддерживают `NetworkPolicy`, этот важный фактор надо учитывать, когда выбирается плагин для использования.

Компонент kube-proxy

Компонент `kube-proxy` — еще одна фоновая программа (демон) Kubernetes после Kubelet, работающая в каждом узле. `kube-proxy` предоставляет основные функции балансировки нагрузки внутри кластера. Она реализует сервисы и основывается на объектах конечных точек `Endpoints/EndpointSlices` — двух объектах API, которые мы подробно будем разбирать в следующей главе при изучении сетевых абстракций. Здесь же ограничимся просто кратким описанием:

- Сервисы определяют балансировщик нагрузки для набора подов.
- Конечные точки (и сечения конечных точек) представляют собой список IP-адресов готовых подов. Они автоматически создаются на базе сервисов, с тем же самым селектором подов, что и сервис.

Большинство типов сервисов имеют IP-адрес сервиса, называемый кластерный IP-адрес, маршрут к которому вне кластера отсутствует. `kube-proxy` отвечает за маршрутизацию запросов на кластерный IP-адрес сервиса. `kube-proxy` представляет собой наиболее распространенную реализацию сервисов Kubernetes, альтернативой ему является Cilium. Значительная часть задач маршрутизации, рассмотренных в *главе 2*, реализована в kube-proxy, в том числе отладка сетевых соединений и производительность.

Кластерный IP-адрес адрес обычно не доступен извне кластера.

Компонент `kube-proxy` имеет четыре режима, которые определяют его работу и характеристики: `userspace`, `iptables`, `ipvs`, `kernelspace`. Режим задается командой `-proxy-mode <mode>`. Отметим, что все режимы в той или иной форме основываются на `iptables`. Рассмотрим их ниже.

Режим *userspace*

Первым и наиболее старым является режим userspace. В этом режиме kube-proxy запускает веб-сервер и маршрутизирует все IP-адреса сервисов на этот сервер, используя iptables. Веб-сервер прекращает соединение и проксирует запрос на под, указанный в конечных точках сервиса. Режим userspace сейчас, как правило, не используется, и мы тоже рекомендуем прибегать к нему только при наличии особых оснований.

Режим *iptables*

Режим iptables полностью основывается на использовании iptables. Это режим, включаемый по умолчанию, и он наиболее популярный (это отчасти связано с тем, что режим IPVS только недавно приобрел GA-устойчивость, а iptables является привычной технологией Linux).

Режим iptables выполняет разветвление соединений, а не реальную балансировку нагрузки. Другими словами, этот режим маршрутизирует соединение на под бэкенда, и все запросы, использующие это соединение, будут попадать на один и тот же под, пока соединение не будет прекращено. Это простая схема с предсказуемым — в идеальном случае — поведением (например, последовательные запросы по одному и тому же соединению получают преимущества благодаря кешированию информации в подах бэкенда).

Однако поведение может стать непредсказуемым, если соединение является долгоживущим, как, например, HTTP/2 (HTTP/2 является протоколом транспортного уровня для gRPC). Предположим, у вас есть два пода, X и Y, которые обслуживают сервис, и вы заменяете X на Z при обычном плавающем обновлении. Старый под Y еще поддерживает все установленные соединения плюс половину соединений, которые должны быть заново установлены, когда удаляется под X, таким образом, трафик через Y существенно возрастает. Подобных сценариев может быть много, и все они приводят к несбалансированным нагрузкам.

Вспомним наши примеры из *раздела «Практика применения iptables» главы 2*. Там мы показывали, что iptables могут конфигурироваться со списком IP-адресов и случайными вероятностями для маршрутов, в соответствии с которыми и устанавливается соединение с IP-адресами. Возьмем сервис, имеющий в бэкенде рабочие поды 10.0.0.1, 10.0.0.2, 10.0.0.3 и 10.0.0.4, тогда kube-proxy сгенерирует последовательность правил, которые будут маршрутизировать соединения следующим образом:

- 25% соединений направляется на 10.0.0.1.
- 33.3% немаршрутизированных соединений направляется на 10.0.0.2.
- 50% немаршрутизированных соединений направляется на 10.0.0.3.
- Все немаршрутизированные соединения направляются на 10.0.0.4.

Это может показаться не слишком логичным и заставляет некоторых инженеров предполагать, что kube-proxy плохо маршрутизирует трафик (поскольку мало кто

смотрит на kube-proxy, когда сервисы работают без сбоев). Ключевым положением здесь является то, что каждое правило маршрутизации применяется для соединений, которые *не были маршрутизированы* по предыдущем правилу. Последнее правило отправляет все соединения на 10.0.0.4 (т. к. в конце концов соединение должны быть с чем-то установлено), предшествующее ему правило дает 50%-ную вероятность направления соединения на 10.0.0.3 как выбор между двумя IP-адресами и т. д. Вероятность маршрутизации рассчитывается как 1/${оставшееся число IP-адресов}.

Ниже показаны Iptables-правила переадресации для сервиса kube-dns в кластере. В данном примере кластерный IP-адрес сервиса kube-dns есть 10.96.9.10. Для наглядности мы сократили и переформатировали выдачу:

```
$ sudo iptables -t nat -L KUBE-SERVICES
Chain KUBE-SERVICES (2 references)
Target    prot opt source  destination

/* kube-system/kube-dns:dns cluster IP */ udp dpt:domain
KUBE-MARK-MASQ udp -- !10.217.0.0/16      10.96.0.10
/* kube-system/kube-dns:dns cluster IP */ udp dpt:domain
KUBE-SVC-TCOU7JCQXEZGVUNU udp -- anywhere 10.96.0.10
/* kube-system/kube-dns:dns-tcp cluster IP */ tcp dpt:domain
KUBE-MARK-MASQ tcp -- !10.217.0.0/16      10.96.0.10
/* kube-system/kube-dns:dns-tcp cluster IP */ tcp dpt:domain
KUBE-SVC-ERIFXISQEP7F7OF4 tcp -- anywhere 10.96.0.10 ADDRTYPE
    match dst-type LOCAL
/* kubernetes service nodeports; NOTE: this must be the
     last rule in this chain */
KUBE-NODEPORTS all – anywhere             anywhere
```

Для kube-dns существует несколько правил UDP и TCP. Мы рассмотрим правила UDP.

Первое UDP-правило помечает любое соединение с сервисом, которое не имеет источником IP-адрес пода (10.217.0.0/16 — это сетевой CIDR пода по умолчанию), как маскированное.

В следующем UDP-правиле в качестве назначения рассматривается цепочка KUBE-SVC-TCOU7JCQXEZGVUNU. Рассмотрим подробнее:

```
$ sudo iptables -t nat -L KUBE-SVC-TCOU7JCQXEZGVUNU
Chain KUBE-SVC-TCOU7JCQXEZGVUNU (1 references)
target     prot opt source     destination

/* kube-system/kube-dns:dns */
KUBE-SEP-OCPCMVGPKTDWRD3C all -- anywhere anywhere     statistic mode
random probability 0.50000000000
/* kube-system/kube-dns:dns */
KUBE-SEP-VFGOVXCRCJYSGAY3 all -- anywhere anywhere
```

Мы имеем цепочку с 50%-ной вероятностью выполнения и цепочку, которая будет выполнена в остальных случаях. Если посмотрим на первую из этих цепочек, то увидим, что она направляет на 10.0.1.141 — один из двух IP-адресов для наших CoreDNS подов:

```
$ sudo iptables -t nat -L KUBE-SEP-OCPCMVGPKTDWRD3C
Chain KUBE-SEP-OCPCMVGPKTDWRD3C (1 references)
target  prot opt source                destination

/* kube-system/kube-dns:dns */
KUBE-MARK-MASQ all -- 10.0.1.141       anywhere
/* kube-system/kube-dns:dns */ udp to:10.0.1.141:53
DNAT    udp - anywhere                 anywhere
```

Режим IPVS

Режим ipvs использует для балансировки нагрузки IPVS (рассмотренный в *главе 2*), а не `iptables`. Данный режим поддерживает шесть методов балансировки, которые задаются с помощью `-ipvs-scheduler`:

- rr: круговой;
- lc: наименьшее число соединений;
- dh: хеширование назначения;
- sh: хеширование источника;
- sed: наименьшая ожидаемая задержка;
- nq: никогда не ожидать очереди.

Круговой метод (rr) балансировки нагрузки устанавливается по умолчанию. Его действие очень похоже на работу режима `iptables` (в котором соединения распределяются равномерно вне зависимости от состояния пода), хотя на самом деле режим `iptables` и не выполняет круговую маршрутизацию.

Режим kernelspace

Режим kernelspace — это новый режим, предназначенный только для Windows. Он является альтернативой режиму userspace Kubernetes под Windows, а то время как iptables и ipvs работают под Linux.

Итак, мы рассмотрели, каким образом в Kubernetes реализуется соединение между подами. Перейдем теперь к изучению сетевой политики и обеспечению безопасности трафика в кластере.

Сетевая политика

По умолчанию Kubernetes разрешает трафик между любыми двумя подами в кластере. Это выгодное решение с точки зрения простоты и гибкости конфигурации, но на практике оно оказывается крайне нежелательным. Позволять любой системе

принимать (или посылать) любой трафик — это риск. Система может подвергнуться атаке хакеров, которые взломают коды доступа или обнаружат слабую или вообще отсутствие авторизации. Разрешение произвольных соединений облегчает также кражу данных из системы через специальным образом модифицированные приложения. В общем, мы *категорически* не рекомендуем запускать кластеры без сетевой политики. Наоборот, мы рекомендует, чтобы все собственники приложений в обязательном порядке использовали объекты `NetworkPolicy` — наряду с другими средствами защиты прикладного уровня, такими как токены авторизации или взаимная авторизация транспортного уровня (mTLS) — для обеспечения безопасности любых сетевых соединений.

`NetworkPolicy` — это ресурс Kubernetes, содержащий правила брандмауэров. Пользователи могут прибегать к объектам `NetworkPolicy`, чтобы ограничить трафик внутрь и извне пода. Ресурс `NetworkPolicy` подключается через конфигурирование плагинов CNI, основной задачей которых и является обеспечение соединений между подами. Kubernetes API заявляет, что поддержка `NetworkPolicy` является необязательной для CNI-драйверов, как следствие — не все CNI-драйверы поддерживают сетевые политики — см. табл. 4.3. Если разработчик создает объекты `NetworkPolicy`, а затем использует CNI-драйвер, не поддерживающий эти объекты, то безопасность трафика между подами оказывается негарантированной. Некоторые CNI-драйверы, например внутренние продукты компаний, могут использовать свои собственные версии сетевых политик. Также некоторые CNI-драйверы по-разному интерпретируют установки в поле spec объекта `NetworkPolicy`.

Таблица 4.3. *Распространенные плагины CNI и поддержка сетевой политики*

CNI-плагин	Поддержка сетевой политики
Calico	Да, дополнительно поддерживает собственные политики
Cilium	Да, дополнительно поддерживает собственные политики
Flannel	Нет
Kubenet	Нет

В примере 4.4 показан объект `NetworkPolicy`, который содержит селектор пода, правила ингресса (входного трафика) и правила игресса (выходного трафика). Политика действует для всех подов с совпадающими метками и находящимися в том же пространстве имен, что и объект `NetworkPolicy`. Использование меток для выбора подов характерно и для других Kubernetes API: согласно установкам в поле `spec` группировка подов осуществляется на основе их меток, а не имен или родительских объектов.

Пример 4.4. Структура объекта NetworkPolicy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
```

```
metadata:
  name: demo
  namespace: default
spec:
  podSelector:
     matchLabels:
      app: demo
  policyTypes:
  - Ingress
  - Egress
  ingress: []NetworkPolicyIngressRule # не развернуты
  egress:  []NetworkPolicyEgressRule # не развернуты
```

Перед тем как погрузиться в изучение деталей API, рассмотрим простой пример: создадим объект `NetworkPolicy`, чтобы ограничить доступ к некоторым подам. Предположим, что у нас есть два компонента: `demo` и `demo-DB`. Поскольку на рис. 4.7 объект `NetworkPolicy` отсутствует, то все поды могут взаимодействовать со всеми остальными подами (включая и не относящиеся к системе — на рисунке не показаны).

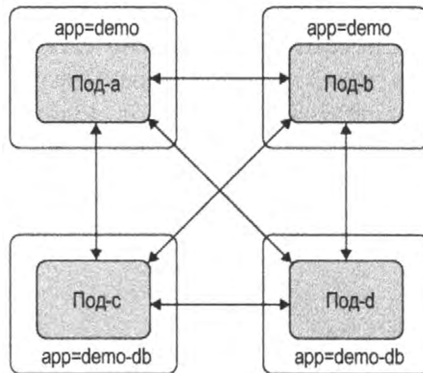


Рис. 4.7. Поды при отсутствии объектов сетевой политики NetworkPolicy

Давайте ограничим уровень доступа к `demo-DB`. Если мы создадим показанный ниже объект `NetworkPolicy`, который отбирает поды с запущенным приложением demo-DB, то эти поды не смогут посылать или получать трафик:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
   name: demo-db
   namespace: default
spec:
   podSelector:
     matchLabels:
     app: demo-db
```

```
policyTypes:
- Ingress
- Egress
```

Из рис. 4.8 мы видим, что поды с меткой app=demo_db больше не могут посылать или получать трафик.

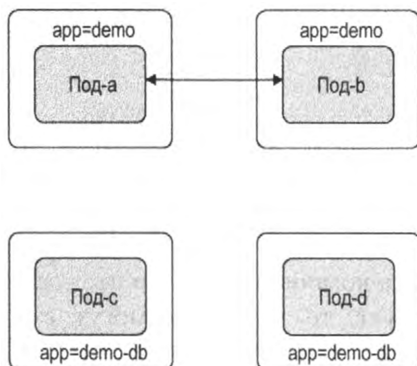


Рис. 4.8. Поды с меткой app:demo_db не могут получать или посылать трафик

Совсем не иметь доступа к сети нежелательно для большинства приложений, включая и нашу базу данных demo_db из примера. Сделаем так, чтобы приложение demo_db имело возможность получать трафик от подов с приложением demo. Для этого добавим к объекту NetworkPolicy правило для входного трафика — ингресс:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: demo-db
  namespace: default
spec:
  podSelector:
    matchLabels:
     app: demo-db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
     - podSelector:
         matchLabels:
           app: demo
```

Теперь поды с demo_db могут получать трафик только от подов с demo. Более того, сами поды с demo_db не могут устанавливать соединения (см. рис. 4.9).

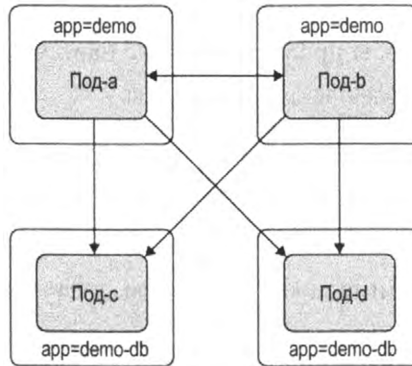


Рис. 4.9. Поды с меткой app:demo_db не могут генерировать соединения, они только могут получать трафик от подов с меткой app:demo

Если пользователь случайно или намеренно изменил метки, то он тем самым изменил и воздействие сетевой политики на поды. В нашем примере, если хакер получил возможность изменить метку app:demo_db пода в том же самом пространстве имен, то созданный нами объект NetworkPolicy больше не будет применим к этому поду. Аналогично, хакер может получить доступ из другого пода пространства имен, если добавит метку app:demo к «захваченному» поду.

Рассмотренный пример — это всего лишь пример. В следующем разделе мы создадим объекты `NetworkPolicy` для нашего веб-сервера на языке Go с помощью плагина Cilium.

Создание объекта *NetworkPolicy* с помощью Cilium

Наш веб-сервер соединяется с базой данных Postgres без TLS. Пока нет объектов `NetworkPolicy`, любой сетевой под может проследить трафик между веб-сервером и базой данных, что представляет собой потенциальный риск. Ниже мы установим наше веб-серверное приложение и базу данных, а затем создадим объекты `NetworkPolicy`, которые ограничат возможный трафик только соединениями между базой данных и нашим веб-сервером. Используя кластер KIND из установки Cilium, развернем базу данных Postgres с помощью следующих YAML-команд и `kubectl`:

```
$ kubectl apply -f database.yaml
service/postgres created
configmap/postgres-config created
statefulset.apps/postgres created
```

Развернем наш веб-сервер в кластере KIND, используя Kubernetes-объект `deployment`:

```
$ kubectl apply -f web.yaml
deployment.apps/app created
```

Для проверки соединений внутри кластерной сети создадим под `dnsutils`, в котором присутствуют базовые сетевые инструменты типа ping и curl:

```
$ kubectl apply -f dnsutils.yaml
pod/dnsutils created
```

Поскольку мы развернули сервис без ингресса, то можно воспользоваться командой `kubectl port-forward`, чтобы проверить связь с нашим веб-сервером:

```
kubectl port-forward app-5878d69796-j889q 8080:8080
```

Подробная информация по kubectl port-forward содержится в документации.

Теперь с локального терминала мы получаем доступ к API:

```
$ curl localhost:8080/
Hello
$ curl localhost:8080/healthz
Healthy
$ curl localhost:8080/data
Database Connected
```

Проверим связь с нашим веб-сервером со стороны других подов в кластере. Для этого нам понадобится IP-адрес пода, где развернут веб-сервер:

```
$ kubectl get pods -l app=app -o wide
NAME                    READY STATUS   RESTARTS AGE  IP           NODE
app-5878d69796-j889q    1/1   Running  0        87m  10.244.1.188 kind-worker3
```

Теперь протестирует соединение с нашим веб-сервером со стороны пода dnsutils (уровни 3 и 7):

```
$ kubectl exec dnsutils -- nc -z -vv 10.244.1.188 8080
10.244.1.188 (10.244.1.188:8080) open
sent 0, rcvd 0
```

Теперь проверим доступ от пода dnsutils к HTTP API (уровень 7):

```
$ kubectl exec dnsutils -- wget -qO- 10.244.1.188:8080/
Hello

$ kubectl exec dnsutils -- wget -qO- 10.244.1.188:8080/data
Database Connected

$ kubectl exec dnsutils -- wget -qO- 10.244.1.188:8080/healthz
Healthy
```

То же самое можно проделать и с подом базы данных. Сначала нужно получить IP-адрес для пода базы данных — это 10.244.2.189. Используем для этого kubectl с комбинацией меток и опций:

```
 $ kubectl get pods -l app=postgres -o wide
NAME            READY STATUS   RESTARTS AGE  IP           NODE
postgres-0      1/1   Running  0        98m  10.244.2.189 kind-worker
```

Снова воспользуемся подом dnsutils, чтобы проверить соединение с базой данных Postgres через порт 5432, заданный по умолчанию:

```
$ kubectl exec dnsutils -- nc -z -vv 10.244.2.189 5432
10.244.2.189 (10.244.2.189:5432) open
sent 0, rcvd 0
```

Использование порта открыто для всех, поскольку сетевая политика отсутствует. Попробуем ограничить это использование, создав с помощью Cilium сетевую политику. Ниже приводятся команды, которые устанавливают сетевую политику. Сначала разрешим доступ к поду с базой данных только со стороны пода с веб-сервером. Применим сетевую политику, разрешающую только трафик от пода с веб-сервером к базе данных:

```
$ kubectl apply -f layer_3_net_pol. yaml
ciliumnetworkpolicy.cilium.io/l3-rule-app-to-db created
```

Установка объектов Cilium с помощью плагина Cilium создает ресурсы, которые могут быть просмотрены с помощью kubectl. Задав команду `kubectl describe ciliumnetworkpolicies.cilium.io l3-rule-app-to-db`, мы увидим полную информацию о созданном правиле:

```
$ kubectl describe ciliumnetworkpolicies.cilium.io l3-rule-app-to-db
Name:         l3-rule-app-to-db
Namespace:    default
Labels:       <none>
Annotations:  API Version: cilium.io/v2
Kind:         CiliumNetworkPolicy
Metadata:
Creation Timestamp: 2021-01-10T01:06:13Z
Generation:         1
Managed Fields:
API Version: cilium.io/v2
Fields Type: FieldsV1
fieldsV1:
f:metadata:
f:annotations:
.:
f:kubectl.kubernetes.io/last-applied-configuration:
f:spec:
.:
f:endpointSelector:
.:
f:matchLabels:
.:
f:app:
f:ingress:
Manager:      kubectl
Operation:    Update
Time:         2021-01-10T01:06:13Z
Resource Version: 47377
```

```
Self Link:
/apis/cilium.io/v2/namespaces/default/ciliumnetworkpolicies/l3-rule-app-to-db
UID:          71ee6571-9551-449d-8f3e-c177becda35a
Spec:
Endpoint Selector:
Match Labels:
App: postgres
Ingress:
From Endpoints:
Match Labels:
App: app
Events:       <none>
```

С применением этой сетевой политики под dnsutils уже не сможет связаться с подом базы данных — это видно по прерыванию операции в связи с истечением времени:

```
$ kubectl exec dnsutils -- nc -z -vv -w 5 10.244.2.189 5432
nc: 10.244.2.189 (10.244.2.189:5432): Operation timed out
sent 0, rcvd 0
command terminated with exit code 1
```

Пока под веб-сервера остается связанным с подом базы данных, маршрут `/data` соединяет веб-сервер с базой данных, и сетевая политика разрешает это соединение:

```
$ kubectl exec dnsutils -- wget -qO- 10.244.1.188:8080/data
Database Connected

$ curl localhost:8080/data
Database Connected
```

Теперь применим политику для 7-го уровня. Cilium поддерживает 7-й уровень, так что мы можем разрешать или блокировать запрос, задавая пути для HTTP URI. В нашем примере мы разрешаем команды HTTP GET на `/` и на `/data`, но блокируем их на `/healthz`:

```
$ kubectl apply -f layer_7_netpol.yml
ciliumnetworkpolicy.cilium.io/l7-rule created
```

Примененную политику можно посмотреть, как и все остальные объекты Kubernetes, с помощью API:

```
$ kubectl get ciliumnetworkpolicies.cilium.io
NAME          AGE
l7-rule       6m54s

$ kubectl describe ciliumnetworkpolicies.cilium.io l7-rule
Name:         l7-rule
Namespace:    default
Labels:       <none>
Annotations:  API Version: cilium.io/v2
```

```
Kind:          CiliumNetworkPolicy
Metadata:
Creation Timestamp: 2021-01-10T00:49:34Z
Generation:        1
Managed Fields:
API Version: cilium.io/v2
Fields Type: FieldsV1
fieldsV1:
   f:metadata:
     f:annotations:
  .:
  f:kubectl.kubernetes.io/last-applied-configuration:
   f:spec:
    .:
    f:egress:
    f:endpointSelector:
.:
f:matchLabels:
  .:
  f:app:
Manager:        kubectl
Operation:      Update
Time:           2021-01-10T00:49:34Z
Resource Version: 43869
Self Link:/apis/cilium.io/v2/namespaces/default/ciliumnetworkpolicies/l7-rule
UID:            0162c16e-dd55-4020-83b9-464bb625b164
Spec:
  Egress:
   To Ports:
    Ports:
      Port:     8080
      Protocol: TCP
    Rules:
      Http:
         Method: GET
         Path:   /
         Method: GET
         Path:   /data
Endpoint Selector:
  Match Labels:
App: app
Events: <none>
```

Как видим, доступны пути / и /data, но не /healthz — в точном соответствии с правилами объекта NetworkPolicy:

```
$ kubectl exec dnsutils -- wget -qO- 10.244.1.188:8080/data
Database Connected
```

```
$kubectl exec dnsutils -- wget -qO- 10.244.1.188:8080/
Hello

$ kubectl exec dnsutils -- wget -qO- -T 5 10.244.1.188:8080/healthz
wget: error getting response
command terminated with exit code 1
```

Этот простой пример показывает, насколько эффективны сетевые политики Cilium для безопасности коммуникаций внутри кластера. Мы настоятельно рекомендуем администраторам выбирать плагины CNI, поддерживающие сетевые политики, и мотивировать разработчиков использовать сетевые политики. Сетевые политики работают в пространстве имен, и если коллективы имеют похожие конфигурации систем, то администраторы кластеров могут и должны требовать, чтобы разработчики применяли сетевые политики для обеспечения дополнительной безопасности трафика.

Мы познакомились с двумя инструментами Kubernetes API — метками и селекторами. В следующем разделе мы рассмотрим на примерах, как еще их можно использовать в кластере.

Группировка подов

Трафик на поды ничем не ограничен до тех пор, пока они не подпадут под действие правил отбора в объекте `NetworkPolicy`. После этого плагин CNI разрешит входящий (ingress) или исходящий (egress) трафик только для тех подов, которые отвечают указанным правилам. Объект `NetworkPolicy` включает в себя поле `.spec.policyTypes`, которое содержит список типов политики (ingress или egress). Например, если мы выбрали под, у которого в `Networkpolicy` указаны правила для ingress, но ничего не указано для egress, то входной трафик будет ограничен, а выходной — нет.

Поле `spec.podSelector` задает, к какому поду будет применена сетевая политика. Пустое поле `label selector.(podSelector: {})` означает, что выбираются все поды в пространстве имен. Более подробно мы рассмотрим селекторы меток ниже.

Объекты `NetworkPolicy` существуют, а сетевая политика применяется только в пределах определенного пространства имен. Поле `spec.podSelector` определяет группировку подов, только если они относятся к тому же пространству имен, что и `NetworkPolicy` . Это значит, что отбор по метке app: demo будет выполняться только в текущем пространстве имен, а поды в другом пространстве имен, также имеющие метку app: demo, не будут затронуты.

Есть несколько способов реализации брандмауэра по умолчанию, среди них:

- В каждом пространстве имен создать объект `NetworkPolicy`, полностью закрывающий трафик. Чтобы разрешить желаемый трафик, необходимо будет создать дополнительные объекты `NetworkPolicy`.
- Добавить пользовательский CNI-плагин, который нарушает поведение открытого по умолчанию API. Некоторые CNI-плагины имеют дополнительную конфигурацию, которая позволяет это проделать.

- Создать политики доступа, требующие, чтобы приложения имели свои объекты `NetworkPolicy`.

В объектах `NetworkPolicy` большое значение имеют метки и селекторы, поэтому ниже мы рассмотрим более сложные примеры.

Тип *LabelSelector*

Впервые в этой книге мы встречаемся с типом `LabelSelector` в ресурсе Kubernetes. Он является часто встречающимся элементом конфигурации, и в следующей главе нам неоднократно придется иметь с ним дело. Так что, когда вы доберетесь до нее, может оказаться полезным еще раз вернуться в настоящий раздел и посмотреть некоторые детали.

Каждый объект Kubernetes имеет поле `metadata` типа `ObjectMeta`. В числе прочих тип содержит поле меток. Метки представляют собой набор пар ключ-значение:

```
metadata:
  labels:
    colour: purple
    shape: square
```

Элемент `LabelSelector` задает группировку ресурсов в зависимости от имеющейся (или отсутствующей) метки. Только очень немногочисленные ресурсы Kubernetes обращаются к другим ресурсам по именам. Большинство же (объекты `NetworkPolicy`, сервисы, развертывания и др.) используют идентификацию по меткам с помощью `LabelSelector`. `LabelSelector` может также встречаться в вызовах API и kubectl, что позволяет обеспечить возвращение только тех объектов, которые нужны. `LabelSelector` содержит два поля: `matchExpressions` и `matchLabels`. Если значения полей в `LabelSelector` не заданы, то выбираются все объекты в области поиска, например все поды, находящиеся в общем пространстве имен с `NetworkPolicy`. `matchLabels` является более простым полем — оно содержит набор пар ключ-значение. Объект считается отвечающим критериям выбора, если он содержит все поля указанных ключей и все ключи имеют указанные значения. Часто `matchLabels` используется с единственным ключом (например, `app=example-thing`), этого, как правило, достаточно для выбора.

В примере 4.5 показано поле `matchLabels` с двумя заданными метками: `colour=purple` и `shape=square`.

Пример 4.5. Пример задания поля matchLabels

```
matchLabels:
  colour: purple
  shape: square
```

Поле `matchExpressions` более мощное, но и более сложное. Оно содержит список условий `LabelSelectorRequirements`. Чтобы объект отвечал критериям отбора, все условия должны быть `true`. В табл. 4.4 перечислены все требуемые для `matchExpressions` условия.

Таблица 4.4. Поля LabelSelectorRequirement

Поле	Описание
key	Задает ключ
operator	Одно из значений `Exists`, `DoesNotExist`, `In`, `NotIn`. `Exists`: выбирается объект с меткой, совпадающей с ключом, вне зависимости от значения ключа. `NotExists`: выбирается объект, у которого нет совпадающей с ключом метки. `In`: Выбирается объект с совпадающей с ключом меткой и значением ключа в заданных пределах. `NotIn`: выбирается объект либо не имеющий совпадающей с ключом метки, или значение ключа лежит вне заданных пределов
Values	Список строчных значений для заданного ключа. Поле пустое, если поле operator имеет значения `In` или `NotIn`. Может быть не пустое, если поле operator установлено в `Exists` или `NotExists`

Рассмотрим два коротких примера на применение `matchExpressions`. В примере 4.6 показано использование `matchExpressions`, эквивалентное нашему предыдущему примеру с `matchLabels`.

Пример 4.6. Использование поля matchExpressions

```
matchExpressions:
- key: colour
  operator: In
  values:
    - purple
- key: shape
  operator: In
  values:
    - square
```

Установки в поле `matchExpressions` в примере 4.7 производят выбор объектов с цветом, отличным от красного, оранжевого или желтого, и имеющих метку `shape`.

Пример 4.7. Еще одно использование поля matchExpressions

```
matchExpressions:
- key: colour
  operator: NotIn
  values:
    - red
    - orange
    - yellow
- key: shape
  operator: Exists
```

Рассмотрев метки, перейдем к рассмотрению правил. Правила обеспечивают реализацию сетевых политик для выбранных с помощью меток объектов.

Правила

Объекты `NetworkPolicy` содержат поля, в которых конфигурируются правила для входного и выходного трафика. Эти правила действуют как исключения, т. е. как список разрешений для выбранных сетевой политикой подов, снимающий устанавливаемую по умолчанию блокировку трафика. Правила не могут блокировать сетевой доступ — они могут только добавить доступ. Если несколько объектов `NetworkPolicy` выбирают под, то действуют все правила, заданные в каждом из этих объектов. Иногда имеет смысл использовать несколько объектов `NetworkPolicy` для одного и того же набора подов (например, определяя условия трафика для приложений в одной политике и условия для инфраструктуры типа передачи телеметрических данных — в другой). Однако имейте в виду, что совсем не обязательно иметь несколько объектов `NetworkPolicy` и что с несколькими политиками труднее работать.

Чтобы поддерживать проверки на готовность и живучесть в Kubelet, CNI-плагин всегда должен разрешать трафик из узла, где сгенерирован под. Метка может злонамеренно использоваться при хакерских атаках, если хакеры получат доступ к узлу (даже без привилегий администратора). Хакеры могут перехватить IP-адрес узла и отправлять пакеты с IP-адресом узла в качестве источника.

Правила для входного и выходного трафика — это два разных типа в `NetworkPolicy` API (`NetworkPolicyIngressRule` и `NetworkPolicyEgressRule`), но функционально их структура совпадает. Каждое правило `NetworkPolicyIngressRule`/ `NetworkPolicyEgressRule` содержит список портов и список полей `NetworkPolicyPeer`.

Поле `NetworkPolicyPeer` допускает четыре способа обращения к сетевым компонентам: `ipBlock`, `namespaceSelector`, `podSelector` и их комбинация.

Установка поля `ipBlock` используется, чтобы разрешить входной трафик от внешних систем и выходной трафик на них. Поле используется само по себе, `namespaceSelector` или `podSelector`. `ipBlock` содержит CIDR или как опцию `except` CIDR. При задании except CIDR будет исключена подгруппа CIDR-адресов (они должны быть в пределах CIDR). В примере 4.8 мы разрешаем трафик от всех IP-адресов в диапазоне от 10.0.0.0 до 10.0.0.255 за исключением 10.0.0.10. В примере 4.9 разрешается трафик от всех подов в пространстве имен, имеющих метку `group:x`.

Пример 4.8. Разрешаем трафик, первый тип

```
from:
  - ipBlock:
    - cidr: "10.0.0.0/24"
    - except: "10.0.0.10"
```

Пример 4.9. Разрешаем трафик, второй тип

```
#
from:
  - namespaceSelector:
     - matchLabels:
       group: x
```

В примере 4.10 мы разрешаем трафик от всех подов в пространстве имен, имеющих метку `service`, при этом поле `x. .podSelector` действует аналогично полю `spec.podSelector`, которое было рассмотрено ранее. Если поле `namespaceSelector` не установлено, то выбор подов происходит в том же пространстве имен, в котором применяется сетевая политика (объект `NetworkPolicy`).

Пример 4.10. Разрешаем трафик, третий тип

```
from:
 - podSelector:
   - matchLabels:
     service: y
```

Если мы зададим поля `namespaceSelector` и `podSelector`, то согласно правилу будут выбраны все поды с указанной меткой пода во всех пространствах имен, имеющих указанную метку пространства. В целях безопасности рекомендуется держать пространство имен маленьким, обычно оно ограничивается приложением или предоставляется для определенного коллектива. Существует еще один способ, показанный в примере 4.11, в котором устанавливаются и `namespaceSelector`, и `podSelector`. Такой комбинированный селектор действует как логическое условие AND: поды должны иметь указанную метку и быть в пространстве имен с указанной меткой.

Пример 4.11. Разрешаем трафик, четвертый тип

```
from:
- namespaceSelector:
  - matchLabels:
    group: monitoring
podSelector:
  - matchLabels:
    service: logscraper
```

Заметьте — это отдельный тип в API, хотя синтаксис YAML выглядит *очень* похоже. Поскольку разделы to и from могут иметь несколько селекторов, один-единственный символ может превратить AND в OR, так что пишите политики очень внимательно.

Наши предыдущие замечания по поводу безопасности при доступе к API применимы и в данном случае. Если пользователь сможет менять метки в пространствах

имен, то вероятны нарушения работы сетевой политики. В нашем предыдущем примере, если пользователь сможет установить метку `group: monitoring` на произвольное пространство имен, то он в принципе получит несанкционированный доступ к входящему и выходящему трафику. Если сетевая политика задана таким образом, что указано только поле `namespaceSelector`, то тогда метки пространства имен будет достаточно, чтобы соответствовать этой политике. Если же в сетевой политике задается еще и метка пода, то для соответствия политике потребуется совпадение и этой метки. Однако в стандартной конфигурации владельцы сервисов предоставляют право на обновление подам в пространстве имен сервиса (или напрямую ресурсам пода, или косвенно через ресурс, подобный развертыванию, который может генерировать поды).

Типичный объект `NetworkPolicy` мог бы выглядеть следующим образом:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
   name: store-api
   namespace: store
spec:
   podSelector:
      matchLabels: {}
   policyTypes:
   - Ingress
   - Egress
   ingress:
   - from:
      - namespaceSelector:
          matchLabels:
             app: frontend
        podSelector:
          matchLabels:
         app: frontend
   ports:
   - protocol: TCP
     port: 8080
egress:
- to:
  - namespaceSelector:
       matchLabels:
          app: downstream-1
    podSelector:
      matchLabels:
          app: downstream-1
  - namespaceSelector:
       matchLabels:
          app: downstream-2
```

```
    podSelector:
      matchLabels:
         app: downstream-2
ports:
- protocol: TCP
  port: 8080
```

В этом примере все поды в нашем пространстве имен `store` могут получать трафик только от подов с меткой app: frontend в пространстве имен с меткой `app: frontend`. Эти поды могут устанавливать соединение только с подами в пространствах имен, в которых и под, и пространство имен имеют метки `app: downstream-1` или `app: downstream-2`. В каждом из этих случаев разрешен только трафик на порт 8080. Наконец, вспомним, что такая сетевая политика не гарантирует, что она совпадет с политикой, действующей для `app: downstream-1` или `app: downstream-2` (см. следующий пример). Установление этих соединений не исключает действие других политик нашего пространства имен, добавляющих дополнительные исключения:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
   name: store-to-downstream-1
   namespace: downstream-1
spec:
   podSelector:
     app: downstream-1
   policyTypes:
   - Ingress
   ingress:
     - from:
       - namespaceSelector:
           matchLabels:
             app: store
       ports:
       - protocol: TCP
       port: 8080
```

Хотя объекты `NetworkPolicy` являются «стабильным» ресурсом (т. е. частью сетевого API v1), мы считает, что они представляют собой только первую ступень обеспечения сетевой безопасности в Kubernetes. Из опыта работы с этими объектами можно заключить, что они не очень удобны для конфигурирования, а их действие по умолчанию оказывается крайне нежелательным. В настоящее время действует рабочая группа, обсуждающая будущее объектов `NetworkPolicy` и возможное содержание второй версии API.

Плагины CNI используют метки и селекторы, чтобы ограничить сетевой трафик для определенных подов. Как мы увидели во многих примерах выше, эти плагины являются существенной часть Kubernetes API, и как администраторы, так и разработчики должны хорошо разбираться в деталях их применения.

Объекты `NetworkPolicy` относятся к важным инструментам работы администратора кластера. Это единственный доступный инструмент для контроля внутрикластерного трафика, исходно присутствующий в Kubernetes API. Другой инструмент для безопасности и управления приложениями, называемый service mesh, мы рассмотрим в соответствующем разделе *главы 5*.

В следующем разделе мы обсудим еще один инструмент администрирования сетей и дадим понимание того, как он работает внутри кластера: систему доменных имен DNS.

DNS

Система доменных имен DNS — ключевой элемент инфраструктуры любой сети. Это относится и к Kubernetes, поэтому краткое описание DNS здесь вполне уместно. В следующих разделах при описании сервисов мы увидим, насколько сильно они зависят от DNS и почему любой дистрибутив Kubernetes обязательно должен предоставлять сервис DNS, соответствующий его спецификации. Но сначала познакомимся с тем, как DNS работает внутри сетей Kubernetes.

В данной книге мы не будем подробно рассматривать спецификацию DNS. Интересующиеся могут обратиться на GitHub, где она доступна в полном виде.

В ранних версиях Kubernetes использовалась система KubeDNS. Она предусматривает несколько контейнеров в одном поде: kube-dns, dnsmasq и sidecar. Контейнер kube-dns отслеживает Kubernetes API и обслуживает записи DNS на базе спецификации Kubernetes DNS. Контейнер dnsmasq обеспечивает кеширование и поддержку тупикового домена, контейнер sidecar отвечает за показатели состояния и тесты работоспособности. Версии Kubernetes старше 1.13 перешли к использованию компонента CoreDNS.

Между CoreDNS и KubeDNS существует несколько отличий:

- Для простоты CoreDNS запускается в одном контейнере.
- CoreDNS — это процесс Go, который реплицирует и усиливает функциональность KubeDNS.
- CoreDNS является универсальным DNS-сервером, обратно совместимым с Kubernetes. Его плагины позволяют расширить функциональность за границы спецификации Kubernetes DNS.

На рис. 4.10 представлены компоненты CoreDNS. Запущено приложение с двумя (по умолчанию) репликами. Чтобы эта конфигурация работала, системе CoreDNS требуется доступ к API-серверу, ConfigMap должен содержать свой файл Corefile, должна быть служба, чтобы предоставлять функции DNS в кластере, и должно работать развертывание, которое запускает поды и управляет ими. Все перечисленные компоненты запускаются в пространстве имен kube-system наряду с другими критическими компонентами кластера.

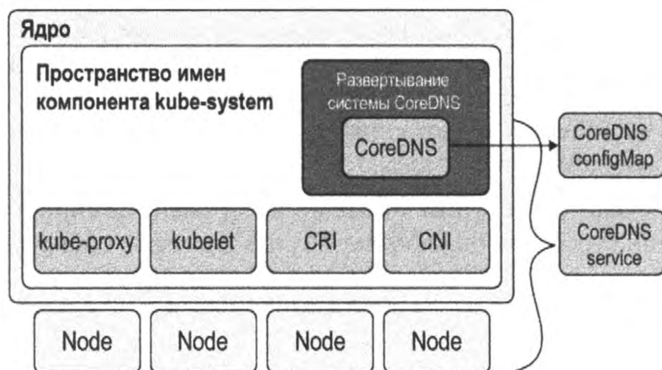


Рис. 4.10. Компоненты системы CoreDNS

Как и в большинстве конфигураций, способ, которым под осуществляет запросы к DNS, задается в поле `spec` объекта под атрибутом `dnsPolicy`.

В примере 4.12 поле spec указывает `ClusterFirstWithHostNet` в качестве политики `dnsPolicy`.

Пример 4.12. Установки поля spec пода, задающие конфигурацию DNS

```
apiVersion: v1
kind: Pod
metadata:
   name: busybox
   namespace: default
spec:
   containers:
   - image: busybox:1.28
     command:
       - sleep
       - "3600"
     imagePullPolicy: IfNotPresent
     name: busybox
   restartPolicy: Always
   hostNetwork: true
   dnsPolicy: ClusterFirstWithHostNet
```

Для поля `dnsPolicy` существуют четыре опции, определяющие то, каким образом распознавание имен DNS работает внутри пода:

Default

Под наследует распознавание имен от узла, на котором он запущен.

ClusterFirst

Запрос DNS, который не соответствует суффиксу домена кластера, типа www.kubernetes.io, посылается на вышестоящий сервер имен, унаследованный от узла.

ClusterFirstWithHostNet

Для подов, запущенных в сети hostNetwork, администраторы кластера должны установить DNS-политику как ClusterFirstWithHostNet.

None

Все установки DNS используют поле dnsConfig в установке spec объекта.

Если указана опция None, то разработчик должен задать серверы имен в поле spec пода. Поле nameservers: содержит список IP-адресов, которые под будет использовать как DNS-серверы. Допускается указать не больше трех адресов. Поле searches: содержит список поисковых доменов DNS для поиска имени хоста в поде. Kubernetes допускает задание не более шести поисковых доменов. Ниже приводится пример с соответствующими установками в поле spec:

```
apiVersion: v1
kind: Pod
metadata:
  namespace: default
  name: busybox
spec:
  containers:
  - image: busybox:1.28
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  dnsPolicy: "None"
  dnsConfig:
    nameservers:
      - 1.1.1.1
    searches:
      - ns1.svc.cluster-domain.example
      - my.dns.search.suffix
```

Дополнительные свойства задаются в поле `Options`, которое также является списком объектов, при этом каждый объект может иметь установки `name` и `value`.

Все заданные свойства записываются в конфигурацию resolv.conf политики DNS. При выполнении запроса система CoreDNS проходит по следующему пути:

```
<service>.default.svc.cluster.local
                ↓
         svc.cluster.local
                ↓
         cluster.local
                ↓
       Путь поиска хоста
```

Путь поиска хоста определяется политикой DNS или политикой CoreDNS для пода.

Запрашивание записи DNS в Kubernetes может повлечь за собой генерирование нескольких запросов, что для приложения увеличивает время возвращения ответа. CoreDNS предлагает решать данную проблему с помощью плагина Autopath, который разрешает автодополнение поискового пути со стороны сервера. Плагин укорачивает клиентский поисковый путь через удаление из него домена кластера и выполнение просмотров на сервере CoreDNS. Когда будет найден подходящий ответ, результат записывается как CNAME и возвращается в одном запросе вместо пяти.

Однако обращение к Autopath увеличивает использование памяти со стороны CoreDNS. Поэтому масштабируйте память для реплик CoreDNS в соответствии с ростом размера кластера. Обеспечьте также адекватность запросов на предоставление памяти и процессора со стороны подов CoreDNS. Для мониторинга состояния CoreDNS можно использовать несколько экспортируемых этой системой показателей:

coredns build info

Информация о самой системе CoreDNS.

dns request count total

Общее число запросов.

dns request duration seconds

Время на обработку каждого запроса.

dns request size bytes

Размер запроса в байтах.

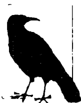
coredns plugin enabled

Показывает, активирован ли плагин для сервера и зоны.

Используя комбинацию показателей работы пода и CoreDNS, администраторы могут контролировать работоспособность CoreDNS в кластере.

Мы перечислили только некоторые из возможных показателей. Полный список доступен по ссылке https://coredns.io/plugins/metrics/.

Autopath и функции, вычисляющие показатели, — это плагины. Такая конструкция позволяет CoreDNS сосредоточиться на своей основной работе — DNS, но при этом расширить свою функциональность, как это было с плагинами CNI. В табл. 4.5 приводится список имеющихся на сегодня плагинов. Поскольку они создаются в рамках проектов с открытым программным кодом, то любой разработчик может внести свой вклад. Есть несколько специальных облачных плагинов типа router53, который дает возможность обслуживать данные зоны из AWS route 53.



Полный список плагинов CoreDNS доступен по ссылке https://coredns.io/explugins/

Таблица 4.5. Плагины системы CoreDNS

Название	Описание
auto	Дает возможность обслуживать данные зоны из RFC 1035 мастер-файла, который автоматически выбирается с диска
autopath	Автодополнение поискового пути стороны сервера. `autopath [ZONE...] RESOLV-CONF`
bind	Меняет хост, с которым должен связываться сервер
cache	Обеспечивает кеш фронтенда. `cache [TTL] [ZONES...]`
chaos	Дает возможность отвечать на запросы TXT в классе CH
debug	Выключает автоматическое восстановление после сбоя, так что вы получите симпатичную трассировку стека. text2pcap
dnssec	Обеспечивает текущее DNSSEC-подписание обслуживаемых данных
dnstap	Разрешает доступ к dnstap
erratic	Полезный плагин для тестирования поведения клиента
errors	Активирует журналирование ошибок
etcd	Дает возможность читать данные зоны с помощью экземпляра etcd версии 3
federation	Дает возможность разрешать объединенные запросы с помощью плагина kubernetes
file	Разрешает обслуживание данных зоны из RFC 1035 мастер-файла
forward	Обеспечивает проксирование сообщений DNS к вышестоящему распознавателю
health	Активирует проверку работоспособности
host	Активирует обслуживание данных зоны из файла директории /etc/hosts
kubernetes	Дает возможность читать данные зоны из кластера Kubernetes
loadbalancer	Рандомизирует порядок записей A, AAAA и MX
log enables	Журналирование запросов к стандартному выводу
loop detect	Простые петли переадресации и остановка сервера
metadata	Активирует сборщик метаданных
nsid	Добавляет идентификатор данного сервера к каждому ответу. RFC 5001
pprof	Публикует параметры работы в конечных точках в /debug/pprof
proxy	Обеспечивает обратный прокси-сервер и надежный балансировщик нагрузки
reload	Обеспечивает автоматическую загрузку модифицированного файла Corefile. Мягкая перезагрузка
rewrite	Выполняет переписывание внутренних сообщений. `rewrite name foo.example.com foo.default.svc.cluster.local`
root	Задает корневую директорию: где искать файлы зоны
router53	Дает возможность обслуживать данные зоны из AWS route53
secondary	Дает возможность обслуживать зону, полученную с первичного сервера

Таблица 4.5 (окончание)

Название	Описание
template	Динамическая генерация ответов на основе входящего запроса
tls	Конфигурирование серверных сертификатов для серверов TLS и gRPC
trace	Позволяет трассирование DNS-запросов на основе OpenTracing (открытое трассирование)
whoami	Возвращает локальный IP-адрес распознавателя, порт и транспортный протокол

CoreDNS имеет многочисленные конфигурационные опции и совместима с моделью Kubernetes. Мы рассмотрели ее возможности только вкратце, если вы хотите больше узнать о CoreDNS, мы рекомендуем книгу издательства O'Reilly «Learning CoreDNS» от авторов John Belamaric, Cricket Liu.

С помощью CoreDNS поды узнают IP-адреса, чтобы получать доступ к приложениям и серверам внутри и вне кластера. В следующем разделе мы подробно рассмотрим, каким образом происходит управление адресами систем IPv4 и 6 в кластере.

Двойной стек IPv4/ IPv6

Kubernetes поддерживает и развивает режим *двойного стека протоколов IPv4/IPv6*, что позволяет кластеру использовать как IPv4, так и IPv6 адреса. Kubernetes имеет уже завершенный блок поддержки кластеров, работающих только в системе IPv6, однако такой режим не позволяет взаимодействовать с клиентами и хост-серверами, работающими только в IPv4. Поэтому режим двойного стека протоколов является мостиком, обеспечивающим адаптацию к IPv6. Ниже мы описываем текущее состояние режима IPv4/IPv6 в Kubernetes на примере Kubernetes 1.20, поэтому имейте в виду, что в последующих версиях могут быть изменения. Полное описание расширенных предложений Kubernetes (KEP) по поддержке двойного стека протоколов доступно на GitHub.

В Kubernetes компонент считается альфа-версией, если его структура еще не окончательно разработана, тестирование не закончено или если он еще недостаточно показал себя при использовании в реальных системах.

Расширенные предложения Kubernetes (KEP) устанавливают параметры, по достижении которых компонент переходит на стадию бета-версии, а затем его структура признается установившейся. Во всех альфа-версиях компонентов Kubernetes поддержка двойного стека отключена, и если она необходима, то должна активироваться вручную.

Режим IPv4/IPv6 делает доступными следующие опции:

- Уникальные адреса IPv4 и IPv6 для каждого пода.
- Сервисы IPv4 и IPv6.
- Маршрутизирование выходного трафика кластера с помощью интерфейсов IPv4 и IPv6.

Поскольку режим стека IPv4/IPv6 является альфа-версией, то администраторы должны активировать данный компонент вручную. Для этого в кластере надо сконфигурировать сетевой интерфейс `IPv6DualStack`. Ниже приводится список опций данного компонента:

```
kube-apiserver
• feature-gates="IPv6DualStack=true"
• service-cluster-ip-range=<IPv4 CIDR>, <IPv6 CIDR>

kube-controller-manager
• feature-gates="IPv6DualStack=true"
• cluster-cidr=<IPv4 CIDR>, <IPv6 CIDR>
• service-cluster-ip-range=<IPv4 CIDR>, <IPv6 CIDR>
• node-cidr-mask-size-ipv4|--node-cidr-mask-size-ipv6
```

По умолчанию установка /24 для IPv4 и /64 для IPv6:

```
kubelet
• feature-gates="IPv6DualStack=true"

kube-proxy
• cluster-cidr=<IPv4 CIDR>, <IPv6 CIDR>
• feature-gates="IPv6DualStack=true"
```

Когда режим IPv4/IPv6 в кластере активирован, то сервисы получают дополнительное поле, в котором разработчик может выбрать ipFamilyPolicy для своего приложения:

`SingleStack`

Сервис использует только одну систему адресов. Управляющий уровень назначает сервису IP-адрес кластера, используя диапазон IP-адресов первого сконфигурированного сервиса.

`PreferDualStack`

Используется, только если в кластере активирован режим двойного стека протоколов. Принцип работы данной опции совпадает с `SingleStack`.

`RequiredDualStack`

Назначает сервису кластерные IP-адреса как из диапазона IPv4, так и из IPv6.

`ipFamilies`

Массив, определяющий, какое семейство IP-адресов использовать в опции SingleStack или порядок следования семейств в опции DualStack. В сервисе семейство IP-адресов задается установкой данного поля. Разрешенными значениями являются ["IPv4"], ["IPv6"] и ["IPv4","IPv6"] (для двойного стека).

Начиная с версии 1.21, двойной стек протоколов активируется по умолчанию.

Ниже приводится пример описания сервиса, в котором атрибут ipFamilyPolicy установлен в PreferDualStack:

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

Заключение

Сетевая модель Kubernetes — это основа функционирования кластерной сети. Принципы данной модели реализуются в компонентах CNI, запускаемых на узлах. Сама модель не определяет сетевую безопасность, но благодаря расширяемости Kubernetes функции безопасности можно реализовывать с помощью плагинов CNI путем задания сетевых политик.

CNI, DNS и безопасность представляют собой основные составляющие кластерной сети, они являются мостом между сетями Linux, описанными в *главе 2*, и контейнерными сетями и сетями Kubernetes, рассматриваемыми в *главах 3 и 5* соответственно.

Выбор соответствующего плагина CNI требует экспертизы как разработчика, так и администратора. Требования должны быть четко сформулированы, а плагины CNI — протестированы. По нашему мнению, кластер не может считаться полноценным, если не сформулированы требования по безопасности трафика и не выбраны поддерживающие их плагины CNI.

Система DNS чрезвычайно важна, для конфигурирования и полноценной работы сети администраторы должны уметь устанавливать CoreDNS в своих кластерах. Большинство сбоев и ошибок Kubernetes связаны с DNS и происходят от неправильного конфигурирования CoreDNS.

Информация данной главы будет востребована в *главе 6*, когда мы перейдем к описанию облачных сетей и тех опций, которые эти сети предоставляют администраторам кластеров.

В следующей главе мы рассмотрим, каким образом все вышеизложенное интегрируется в абстракциях Kubernetes.

ГЛАВА 5

Сетевые абстракции в Kubernetes

В предыдущих главах мы рассмотрели ряд базовых сетевых технологий и описали, каким образом в Kubernetes трафик попадает из точки A в точку B. В данной главе мы обсудим сетевые абстракции Kubernetes, в первую очередь обнаружение сервисов и балансировку нагрузки. Основное внимание здесь мы уделим сервисам и входному трафику (ингрессам). Оба ресурса Kubernetes довольно сложны вследствие большого количества опций, которые должны обслуживать многочисленные практические реализации. Эти ресурсы — наиболее видимая часть сетевого стека Kubernetes, поскольку они определяют основные характеристики работы в сети Kubernetes. Именно с их помощью разработчики подключают сети к своим приложениям, развернутым в среде Kubernetes.

В данной главе на конкретных примерах будут рассмотрены сетевые абстракции Kubernetes и детали их функционирования. Чтобы следовать за изложением, вам понадобятся следующие инструменты:

- Docker.
- KIND.
- Linkerd.

Вы также должны быть знакомы с командами `kubectl exec` и `Docker exec`. Если же это не так, то не стоит беспокоиться — в библиотеке программ данной книги все обсуждаемые команды присутствуют. Помимо этого, мы еще будем использовать команды `ip` и `netns` из *глав 2 и 3*. Отметим, что большинство названных инструментов предназначены для отладки и вывода деталей работы программы, так что в условиях нормального функционирования программ они не требуются.

Пакеты для установки Docker, KIND и Linkerd доступны на соответствующих сайтах, дополнительная информация содержится в библиотеке программ книги.

Для примеров данной главы `kubectl` является ключевым инструментом, он также является стандартом для операторов при взаимодействии с кластерами и кластерными сетями. Вы должны быть знакомы с употреблением команд `kubectl create`, `apply`, `get`, `delete` и `exec`. Подробности смотрите в документации Kubernetes или запускайте `kubectl [команда] - help`.

В данной главе мы разберем следующие сетевые абстракции Kubernetes:

- StatefulSets (объекты, сохраняющие состояние после выполнения).
- Endpoints (конечные точки):
  - Endpoint slices.

- Services (сервисы):
  - Nodeport.
  - Cluster.
  - Headless.
  - External.
  - LoadBalancer.
- Ingress (ингресс):
  - Ingress controller.
  - Ingress rules.
- Sevice meshes:
  - Linkerd.

При изучении этих абстракций мы будем разворачивать примеры в нашем кластере Kubernetes, выполняя следующие шаги:

1. Развернуть KIND-кластер с разрешенным входным трафиком.
2. Изучить работу объектов StatefulSets.
3. Развернуть сервисы Kubernetes.
4. Развернуть ингресс-контроллер.
5. Развернуть Linkerd service mesh.

Упомянутые абстракции представляют собой основу того, что Kubernetes API предлагают разработчикам и администраторам для программного управления потоком трафика внутрь и наружу кластера. Понимание принципов работы абстракций чрезвычайно важно для успешной работы приложений Kubernetes внутри кластера. После знакомства с примерами данной главы вы будете ориентироваться в том, какие абстракции наиболее подходят для решения вашей конкретной задачи.

С помощью конфигурационного файла (YAML) для KIND-кластера мы можем использовать KIND и команду из следующего раздела, чтобы создать кластер. Если вы проделываете это в первый раз, то потребуется некоторое время для загрузки всех требуемых образов Docker.

Примеры данной главы предполагают, что у вас уже есть запущенный локальный KIND-кластер из предыдущей главы, наряду с веб-сервером на Go и образами `dnsutils` для тестирования.

StatefulSet

Абстракция StatefulSet — это API-интерфейс Kubernetes для управления подами, он похож по действиям на ресурс deployment. Однако, в отличие от deployment, интерфейс StatefulSet предоставляет приложениям дополнительные возможности:

- Стабильные уникальные сетевые идентификаторы.
- Постоянно существующее (энергонезависимое) хранилище.
- Упорядоченное мягкое развертывание и масштабирование.
- Упорядоченное автоматическое плавное обновление.

Ресурс deployment больше подходит для приложений, которым эти возможности не требуются (например, сервис хранит данные во внешней базе данных).

База данных для нашего минимального веб-сервера использует StatefulSet. База данных содержит сервис, файл ConfigMap для пользовательского имени Postgres, пароль, имя тестовой базы данных и объект StatefulSet для контейнеров, в которых запущен Postgres.

Итак, развернем нашу базу данных:

```
kubectl apply -f database.yaml
service/postgres created
configmap/postgres-config created
statefulset.apps/postgres created
```

Изучим DNS и сетевые разветвления вследствие использования StatefulSet.

Чтобы протестировать DNS внутри кластера, можно воспользоваться образом dnsutils; он находится по адресу gcr .io/kubernetes-e2e-test-images/dnsutils:1.3 и используется для тестирования Kubernetes:

```
kubectl apply -f dnsutils.yaml

pod/dnsutils created

kubectl get pods
NAME         READY        STATUS     RESTARTS          AGE
dnsutils      1/1         Running    0                 9s
```

Имея реплику приложения, сконфигурированную на два пода, мы видим, что `StatefulSet` разворачивает `postgres-0` и `postgres-1` — в таком порядке — и присваивает им соответственно IP-адреса 10.244.1.3 и 10.244.2.3:

```
kubectl get pods -o wide

NAME          READY      STATUS RESTARTS  AGEIP   NODE
Dnsutils       1/1       Running 0          15m  10.244.3.2   kind-worker3
postgres-0     1/1       Running 0          15m  10.244.1.3   kind-worker2
postgres-1     1/1       Running 0          14m  10.244.2.3   kind-worker
```

Имя нашего автономного сервиса Postgres клиент может использовать для запрашивания IP-адресов конечных точек:

```
kubectl get svc postgres

NAME       TYPE        CLUSTER-IP       EXTERNAL-IP        PORT(S)        AGE
Postgres               ClusterIP        <none>             5432/TCP       23m
```

Используя наш образ dnsutils, можно увидеть, что имена DNS для объектов `StatefulSets` возвращают эти IP-адреса вместе с IP-адресом кластера, в котором размещен сервис Postgres:

```
kubectl exec dnsutils -- host postgres-0.postgres.default.svc.cluster.local.
postgres-0.postgres.default.svc.cluster.local has address 10.244.1.3

kubectl exec dnsutils -- host postgres-1.postgres.default.svc.cluster.local.
postgres-1.postgres.default.svc.cluster.local has address 10.244.2.3

kubectl exec dnsutils -- host postgres
postgres.default.svc.cluster.local has address 10.105.214.153
```

Объекты StatefulSets имитируют фиксированную группу постоянно существующих машин. Будучи стандартным решением для задач, требующих сохранения состояния после завершения работы, поведение этих объектов в некоторых случаях может быть неоптимальным.

Общей проблемой, с которой сталкиваются пользователи, является обновление, требующее ручного исправления, что происходит при использовании `.spec.updateStrategy.type: RollingUpdate` и `.spec.podManagementPolicy: OrderedReady` (установки по умолчанию). При этих установках пользователь вынужден вмешиваться вручную, если обновленный под не проходит тест готовности.

Дополнительно объекты `StatefulSets` требуют сервис, лучше автономный, который будет отвечать за сетевую идентификацию подов, и конечный пользователь должен будет создать такой сервис.

Объекты `StatefulSets` имеют многочисленные конфигурационные опции, кроме того, существуют независимые альтернативы (как стандартные контроллеры рабочих приложений с сохранением состояния, так и программно-специфичные контроллеры).

В Kubernetes объекты `StatefulSets` предоставляют функционал для отдельных специальных случаев, и их не рекомендуется использовать при развертывании обычных приложений. Сетевые абстракции, более подходящие для таких приложений, будут рассмотрены ниже.

В следующем разделе мы обсудим API конечных точек и ресурс EndpointSlice — «становой хребет» сервисов Kubernetes.

Конечные точки

Конечные точки помогают идентифицировать поды, на которых запущен сервис. Конечные точки создаются и управляются сервисами. Сервисы мы рассмотрим позже, чтобы не громоздить слишком много новых объектов в одном месте. Пока же отметим, что сервис содержит стандартный селектор меток (описанный в *главе 4*), который определяет, какие поды относятся к конечной точке.

На рис. 5.1 показан трафик, направляемый на конечную точку на узле 2, под 5.

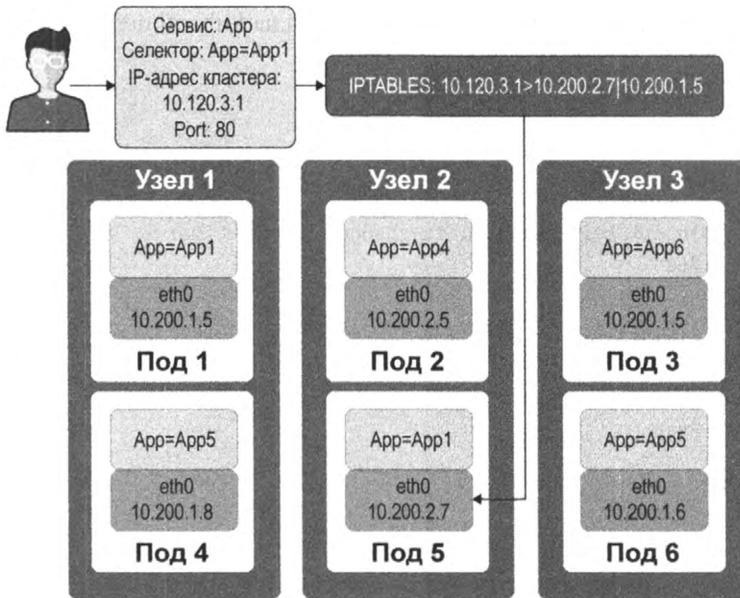


Рис. 5.1. Конечные точки сервиса

Давайте рассмотрим, каким образом данная конечная точка создается и поддерживается в кластере.

Каждая конечная точка содержит список портов (которые относятся ко всем подам) и два списка адресов — готовых и неготовых:

```
apiVersion: v1
kind: Endpoints
metadata:
   labels:
     name: demo-endpoints
subsets:
- addresses:
  - ip: 10.0.0.1
- notReadyAddresses:
  - ip: 10.0.0.2
ports:
- port: 8080
  protocol: TCP
```

Если адреса проходят тесты на готовность пода, то они перечисляются в поле. `addresses`, если же нет — то в поле `.notReadyAddresses`. Тем самым конечные точки становятся инструментом *обнаружения сервиса*. Наблюдая за объектом `Endpoints`, можно оценить готовность к работе и получить адреса всех подов:

```
kubectl get endpoints clusterip-service

NAME                 ENDPOINTS
clusterip-service    10.244.1.5:8080,10.244.2.7:8080,10.244.2.8:8080 + 1 more...
```

Удобнее просматривать адреса с помощью команды kubectl describe:

```
kubectl describe endpoints clusterip-service

Name:          clusterip-service
Namespace:     default
Labels:        app=app
Annotations:endpoints.kubernetes.io/last-change-trigger-time:
2021-01-30T18:51:36Z
Subsets:
Addresses:       10.244.1.5,10.244.2.7,10.244.2.8,10.244.3.9
NotReadyAddresses: <none>
Ports:
Name       Port  Protocol
----       ----  ----
<unset> 8080  TCP

Events:
Type       Reason       Age       From       Message
----       ------       ----      ----       -------
```

Попробуем убрать метку с приложения и посмотрим, как на это отреагирует Kubernetes. С отдельного терминала запустите указанную ниже команду, она даст нам возможность отслеживать все изменения в подах в реальном времени:

```
kubectl get pods -w
```

С другого терминала запустите ту же команду для конечных точек:

```
kubectl get endpoints -w
```

Теперь нам надо получить имя пода, чтобы удалить его из объекта Endpoints:

```
kubectl get pods -l app=app -o wide

NAME                   READY  STATUS    RESTARTS  AGE  IP          NODE
app-5586fc9d77-7frts   1/1    Running   0         19m  10.244.1.5  kind-worker2
app-5586fc9d77-mxhgw   1/1    Running   0         19m  10.244.3.9  kind-worker3
app-5586fc9d77-qpxwk   1/1    Running   0         20m  10.244.2.7  kind-worker
app-5586fc9d77-tpz8q   1/1    Running   0         19m  10.244.2.8  kind-worker
```

С помощью команды kubectl label мы можем изменить метку пода app-5586fc9d77-7frts app=app:

```
kubectl label pod app-5586fc9d77-7frts app=nope --overwrite
pod/app-5586fc9d77-7frts labeled
```

Когда с пода будет удалена метка, то поды и обе отслеживающие команды заметят изменения. Контроллер конечных точек заметит изменения в поде с меткой app=app, это же относится и к контроллеру развертывания. Таким образом, Kubernetes сделал то, что и должен был, — теперь реальное состояние отражает желаемое состояние:

```
kubectl get pods -w

NAME                    READY   STATUS              RESTARTS   AGE
app-5586fc9d77-7frts    1/1     Running             0          21m
app-5586fc9d77-mxhgw    1/1     Running             0          21m
app-5586fc9d77-qpxwk    1/1     Running             0          22m
app-5586fc9d77-tpz8q    1/1     Running             0          21m
dnsutils                1/1     Running             0          3h1m
postgres-0              1/1     Running             0          3h
postgres-1              1/1     Running             0          3h
app-5586fc9d77-7frts    1/1     Running             0          22m
app-5586fc9d77-7frts    1/1     Running             0          22m
app-5586fc9d77-6dcg2    0/1     Pending             0          0s
app-5586fc9d77-6dcg2    0/1     Pending             0          0s
app-5586fc9d77-6dcg2    0/1     ContainerCreating   0          0s
app-5586fc9d77-6dcg2    0/1     Running             0          2s
app-5586fc9d77-6dcg2    1/1     Running             0          7s
```

Развернуто четыре пода, но наш под с измененной меткой еще существует — это `app-5586fc9d77-7frts`:

```
kubectl get pods

NAME                    READY   STATUS    RESTARTS    AGE
app-5586fc9d77-6dcg2    1/11    Pending   0           4m51s
app-5586fc9d77-7frts    1/11    Pending   0           27m
app-5586fc9d77-mxhgw    1/11    Pending   0           27m
app-5586fc9d77-qpxwk    1/11    Pending   0           28m
app-5586fc9d77-tpz8q    1/11    Pending   0           27m
dnsutils                1/11    Pending   3           3h6m
postgres-0              1/11    Pending   0           3h6m
postgres-1              1/11    Pending   0           3h6m
```

Под `app-5586fc9d77-6dcg2` теперь является частью объекта конечных точек с IP-адресом `10.244.1.6`:

```
kubectl get pods app-5586fc9d77-6dcg2 -o wide

NAME                 READY  STATUS    RESTARTS  AGE   IP          NODE
app-5586fc9d77-6dcg2 1/1    Running     0       3m6s  10.244.1.6  kind-worker2
```

Как обычно, все детали процесса мы можем посмотреть с помощью `kubectl describe`:

```
kubectl describe endpoints clusterip-service

Name:         clusterip-service
Namespace:    default
Labels:       app=app
Annotations: endpoints.kubernetes.io/last-change-trigger-time:
2021-01-30T19:14:23Z
```

```
Subsets:
   Addresses:    10.244.1.6,10.244.2.7,10.244.2.8,10.244.3.9
   NotReadyAddresses: <none>
   Ports:
     Name  Port Protocol
    ----   ----  ----
    <unset> 8080 TCP
Events:
   Type     Reason    Age    From   Message
   ----     -------   --     ---    ------
```

При больших развертываниях объект конечных точек может стать довольно большим, так что возможно замедление изменений в кластере. Чтобы преодолеть эту проблему, разработчики Kubernetes предложили концепцию сечений множества конечных точек (endpoint slices).

Endpoint Slices

Вы можете спросить, чем этот ресурс отличается от конечных точек? Вот с этого места как раз и начинается наше путешествие в дебри Kubernetes.

В стандартном кластере Kubernetes запускает `kube-proxy` на каждом узле. `kube-proxy` отвечает за работу сервиса в пределах узла путем маршрутизации и балансировки *выходящей* нагрузки для всех подов сервиса. Для этого `kube-proxy` отслеживает все конечные точки в кластере, чтобы определить готовые к работе поды, на которые можно маршрутизировать сервисы.

Теперь представим, что у нас *большой* кластер с тысячами узлов и десятками тысяч подов.

Это значит, что тысячи `kube-proxy` ведут наблюдение за конечными точками. Если в объекте `Endpoints` меняется один какой-нибудь адрес (вследствие выполнения плавного обновления, масштабирования, уничтожения, непрохождения теста на работоспособность или по любой другой причине), то обновленный объект пересылается на все слушающие `kube-proxy`. Проблема еще более обостряется при большом числе подов, поскольку чем больше подов, тем больше объекты `Endpoints` и соответственно более частые обновления. Это приводит к увеличению нагрузки на `etcd` — Kubernetes API-сервер и на саму сеть. Масштабирование в Kubernetes довольно сложно и зависит от многих причин, но отслеживание конечных точек является стандартной проблемой в кластерах с тысячами узлов. Интересно отметить, что многие пользователи Kubernetes считают именно эту проблему основным фактором, ограничивающим размер кластера.

Проблема обусловлена архитектурой `kube-proxy` и ожиданием того, что каждый под должен быть способен связываться с любым сервисом без предуведомления. Сечения множества конечных точек (Endpoint slices) — это подход, который позволяет сохранить архитектуру `kube-proxy`, но при этом снимает проблему с отслеживанием конечных точек в больших кластерах с большими сервисами.

Объекты `EndpointSlices` имеют содержание, сходное с объектами `Endpoint`, но дополнительно включают в себя массив конечных точек:

```
apiVersion: discovery.k8s.io/v1beta1
kind: EndpointSlice
metadata:
  name: demo-slice-1
  labels:
    kubernetes.io/service-name: demo
addressType: IPv4
ports:
  - name: http
    protocol: TCP
    port: 80
endpoints:
  - addresses:
      - "10.0.0.1"
    conditions:
        ready: true
```

Смысловая разница между конечными точками и сечениями заключается в том, каким образом их обрабатывает Kubernetes. В случае «обычных» конечных точек сервис Kubernetes создает одну конечную точку для всех подов в сервисе. Сервис же создает *множество* сечений для конечных точек, каждое из которых содержит набор подов — на рис. 5.2 такой набор изображен. Объединение всех сечений для конечных точек содержит все поды сервиса. Таким образом, изменение IP-адреса (вследствие появления нового пода, удаления пода, изменения статуса готовности пода) будет приводить к гораздо меньшему трафику на отслеживающие объекты.

Рис. 5.2. Сравнение объектов `Endpoints` и `Endpointslice`

Поскольку в Kubernetes отсутствует транзакционный API, то один и тот же адрес может временно появляться в разных сечениях. Все программы, работающие с сечениями конечных точек (такие как kube-proxy), должны учитывать этот факт.

Максимальное число адресов в сечении множества контрольных точек задается опцией --maxendpoints-per-slice в менеджере kube-controller-manager. На данный момент установка по умолчанию 100, максимум 1000. Контроллер сечений заполняет существующие сечения прежде, чем создать новые, но не изменяет баланс сечения.

Контроллер сечений множества контрольных точек зеркально отображает контрольные точки на сечение, чтобы система могла продолжить записывать контрольные точки, воспринимая сечение как «источник истины». Будущее такой схемы и вообще контрольных точек пока окончательно не определено (как для ресурса первой версии, для конечных точек может быть объявлен ограниченный период использования). Есть четыре исключительных случая, в которых отзеркаливание не производится:

- Отсутствует соответствующий сервис.
- Ресурс соответствующего сервиса выбирает поды.
- Объект Endpoints имеет метку endpointslice.kubetnetes.io/skıpmirror: true.
- Объект Endpoint аннотируется как control-plane.alpha.kubernetes.ıo/leader.

Вы можете получить все сечения конечных точек для заданного сервиса путем задания его имени в .metadata.labels."kubernetes.io/service-name".

Концепция сечений множества конечных точек присутствует в качестве бета-версии, начиная с Kubernetes 1.17. В этом же состояния она находится в Kubernetes 1.20 — текущей версии на дату написания книги. Бета-версии обычно не претерпевают особых изменений и могут превращаться в окончательный API, но это не обязательно. Если вы используете сечения, то имейте в виду, что следующие версии Kubernetes могут без специального уведомления внести в упоминаемую концепцию существенные изменения.

Посмотрим на конечные точки в кластере, используя команду kubectl et.endpointslıce:

```
kubectl get endpointslice

NAME                     ADDRESSTYPE PORTS  ENDPOINTS
clusterip-service-l2n9q  IPv4        8080   10.244.2.7,10.244.2.8,10.244.1.5
```

Если мы хотим получить больше информации о сечениях конечных точек сервиса clusterıp-service-l2n9q, то можно воспользоваться командой kubectl describe:

```
kubectl descrıbe endpointslice clusterip-service-l2n9q

Name:         clusterip-service-l2n9q
Namespace:    default
Labels:
endpointslice.kubernetes.io/managed-by=endpointslice-controller.k8s.io
```

```
kubernetes.io/service-name=clusterip-service
Annotations: endpoints.kubernetes.io/last-change-trigger-time:
2021-01-30T18:51:36Z
AddressType: IPv4
Ports:
   Name     Port   Protocol
   ----     ----   --------
   <unset>  8080   TCP
Endpoints:
- Addresses:        10.244.2.7
  Conditions:
     Ready:         true
  Hostname:         <unset>
  TargetRef:        Pod/app-5586fc9d77-qpxwk
  Topology:         kubernetes.io/hostname=kind-worker
- Addresses:        10.244.2.8
  Conditions:
     Ready:         true
  Hostname:         <unset>
  TargetRef:        Pod/app-5586fc9d77-tpz8q
  Topology:         kubernetes.io/hostname=kind-worker
- Addresses:        10.244.1.5
  Conditions:
     Ready:         true
  Hostname:         <unset>
  TargetRef:        Pod/app-5586fc9d77-7frts
  Topology:         kubernetes.io/hostname=kind-worker2
- Addresses:        10.244.3.9
  Conditions:
     Ready:         true
  Hostname:         <unset>
  TargetRef:        Pod/app-5586fc9d77-mxhgw
Topology:           kubernetes.io/hostname=kind-worker3
Events:             <none>
```

В выдаче мы видим под, входящий в сечение множества конечных точек из `TargetRef`. Информация в поле `Topology` сообщает нам имя хост-сервера рабочего узла, на котором развернут под. Самое важное — поле `Addresses` возвращает IP-адрес объекта конечных точек.

Важно понимать концепции конечных точек и сечений, поскольку с их помощью осуществляется идентификация подов, отвечающих за сервисы, независимо от типа развернутого сервиса. Ниже в данной главе мы рассмотрим, как использовать конечные точки и метки для устранения ошибок. В следующем разделе мы изучим все типы сервисов Kubernetes.

Сервисы Kubernetes

Сервис в Kubernetes — это абстракция, выполняющая балансировку нагрузки в кластере. Существуют четыре типа сервисов, которые задаются через поле `.spec.Type`. Каждый тип реализует отдельную форму балансировки нагрузки или обнаружения сервиса. Эти типы — `ClusterIP`, `NodePort`, `LoadBalancer` и `ExternalName`.

Для выбора пода сервисы используют стандартный селектор пода. Сервис включает в себя все выбранные поды. Для обнаружения пода сервисы создают конечную точку (или сечение конечных точек):

```
apiVersion: v1
kind: Service
metadata:
  name: demo-service
spec:
  selector:
    app: demo
```

Во всех примерах на сервисы мы будем использовать наш минимальный веб-сервер на Go. Мы добавили приложению функциональность путем отображения IP-адресов пода и хост-сервера в запросе REST.

На рис. 5.3 показан сетевой статус нашего пода как единственного пода в кластере. В одних случаях сетевые объекты, которые мы собираемся исследовать, будут экспонировать поды нашего приложения за пределами кластера, в других они позволят нам масштабировать приложения, чтобы соответствовать новым требованиям. Напомним материал из *глав 3 и 4*, что запущенные внутри подов контейнеры имеют

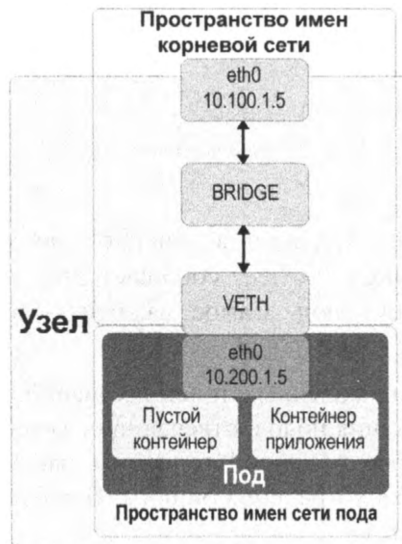


Рис. 5.3. Под на хост-сервере

общее пространство имен. Кроме того, для каждого пода также создается пустой (pause) контейнер. Пустой контейнер управляет пространствами имен для пода.

Пустой контейнер — это родительский контейнер для всех контейнеров, запущенных на поде. Он содержит все общие пространства имен для пода. Более подробно о пустом контейнере можно прочесть в блоге Яна Льюиса (Ian Lewis).

Прежде чем мы развернем сервисы, мы сначала должны развернуть веб-сервер, на который сервисы будут направлять трафик:

```
kubectl apply -f web.yaml
deployment.apps/app created

kubectl get pods -o wide

NAME                  READY  STATUS   RESTARTS  AGE  IP          NODE
app-9cc7d9df8-ffsm6    1/1    Running   0       49s  10.244.1.4  kind-worker2
dnsutils               1/1    Running   0       49m  10.244.3.2  kind-worker3
postgres-0             1/1    Running   0       48m  10.244.1.3  kind-worker2
postgres-1             1/1    Running   0       48m  10.244.2.3  kind-worker2
```

Начнем рассмотрение сервисов с сервиса NodePort.

NodePort

Сервис Nodeport облегчает внешним программам, например, балансировщику нагрузки, направление трафика на поды. Программе требуется только знать IP-адреса узлов и порты сервиса. Сервис Nodeport экспонирует на всех узлах фиксированный порт, который относится к рабочим подам. Сервис использует для задания такого открытого на всех узлах порта поле `.spec.ports[].nodePort`:

```
apiVersion: v1
kind: Service
metadata:
  name: demo-service
spec:
  type: NodePort
  selector:
    app: demo
  ports:
   - port: 80
     targetPort: 80
     nodePort: 30000
```

Поле `nodePort` может быть оставлено пустым, в этом случае Kubernetes автоматически выбирает уникальный порт. Опция `–service-node-port-range` в `kube-controller-manager` устанавливает действующий диапазон для портов 30000–32767. Порт, задаваемый вручную, должен быть из этого диапазона.

С помощью сервиса NodePort внешние пользователи могут соединяться с портом на любом узле и направлять трафик на тот под в узле, который поддерживает требуемый сервис (рис. 5.4). На этом рисунке сервис направляет трафик на узел 3, а правила iptables перенаправляют трафик на узел 2, который является хостом для пода. Такая процедура не очень эффективна, поскольку в общем случае соединение устанавливается с подом на другом узле.

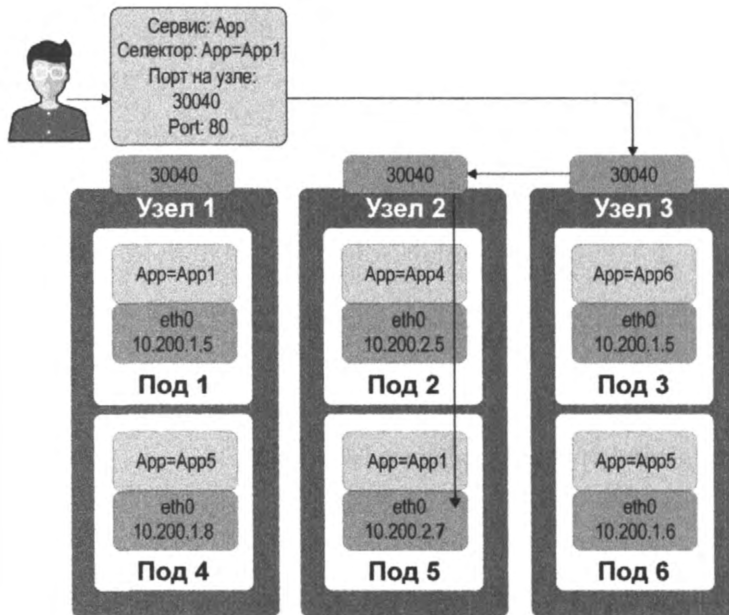


Рис. 5.4. Поток трафика в сервисе NodePort

Здесь стоит обсудить один атрибут сервиса, а именно externalTrafficPolicy. Данный атрибут устанавливает, будет ли сервис направлять внешний трафик на конечные точки, локальные по отношению к узлу, или на точки, доступные везде в кластере. Если задано Local, то сохраняется IP-адрес клиента и не происходит вторичного перенаправления на сервисы LoadBalancer и NodePort, но при этом есть риск распространения потенциально несбалансированного трафика. При задании Cluster IP-адрес клиента скрывается и возможно перенаправление на другой узел, но распределение нагрузки в целом равномерное. Установка Cluster означает, что для каждого рабочего узла правила iptables в kube-proxy определяют маршрут к поддерживающим сервис подам везде в кластере — как и было показано на рис. 5.4.

Если же установлено Local, то правила iptables в kube-proxy действуют только на рабочих узлах с запущенными на них соответствующими подами, и трафик маршрутизируется локально по отношению к рабочему узлу. Задание Local также позволяет разработчикам приложений сохранять IP-адрес источника пользовательского запроса. Когда атрибут externalTraffiPolicy установлен в Local, то kube-proxy будет проксировать запросы только к конечным точкам, локальным к узлу, и не будет направлять трафик на другие узлы. Если локальные конечные точки отсутствуют, то направленные на узел пакеты сбрасываются.

Чтобы выполнить некоторые дополнительные тесты, увеличим количество реплик нашего веб-приложения:

```
kubectl scale deployment app --replicas 4
deployment.apps/app scaled

kubectl get pods -l app=app -o wide

NAME                   READY  STATUS   IP           NODE
app-9cc7d9df8-9d5t8    1/1    Running  10.244.2.4   kind-worker
app-9cc7d9df8-ffsm6    1/1    Running  10.244.1.4   kind-worker2
app-9cc7d9df8-srxk5    1/1    Running  10.244.3.4   kind-worker3
app-9cc7d9df8-zrnvb    1/1    Running  10.244.3.5   kind-worker3
```

Запущены четыре пода — по одному поду в каждом узле кластера:

```
kubectl get pods -o wide -l app=app

NAME                    READY STATUS    IP           NODES
app-5586fc9d77-7frts 1/1    Running   10.244.1.5   kind-worker2
app-5586fc9d77-mxhgw 1/1    Running   10.244.3.9   kind-worker3
app-5586fc9d77-qpxwk 1/1    Running   10.244.2.7   kind-worker
app-5586fc9d77-tpz8q 1/1    Running   10.244.2.8   kind-worker
```

Теперь развернем наш сервис `NodePort`:

```
kubectl apply -f services-nodeport.yaml
service/nodeport-service created

kubectl describe svc nodeport-service
Name:                     nodeport-service
Namespace:                default
Labels:                   <none>
Annotations:              Selector: app=app
Type:                     NodePort
IP:                       10.101.85.57
Port:                     echo 8080/TCP
TargetPort:               8080/TCP
NodePort:                 echo 30040/TCP
Endpoints:                10.244.1.5:8080,10.244.2.7:8080,10.244.2.8:8080
+ 1 more...
Session Affinity:         None
External Traffic Policy: Cluster
Events:                   none>
```

Чтобы протестировать сервис `NodePort`, нам нужно получить IP-адрес рабочего узла:

```
kubectl get nodes -o wide

NAME                 STATUS  ROLES   INTERNAL-IP  OS-IMAGE
kind-control-plane   Ready   master  172.18.0.5   Ubuntu 19.10
kind-worker          Ready   <none>  172.18.0.3   Ubuntu 19.10
```

```
kind-worker2        Ready    <none>  172.18.0.4   Ubuntu 19.10
kind-worker3        Ready    <none>  172.18.0.2   Ubuntu 19.10
```

Трафик, внешний к кластеру, будет использовать номер порта `NodePort` 30040, открытого на каждом рабочем узле, и IP-адрес рабочего узла.

Мы видим, что каждый хост в кластере имеет доступ к нашим подам:

```
kubectl exec -it dnsutils -- wget -q -O-  172.18.0.5:30040/host
NODE: kind-worker2, POD IP:10.244.1.5

kubectl exec -it dnsutils -- wget -q -O-  172.18.0.3:30040/host
NODE: kind-worker, POD IP:10.244.2.8

kubectl exec -it dnsutils -- wget -q -O-  172.18.0.4:30040/host
NODE: kind-worker2, POD IP:10.244.1.5
```

Сервис NodePort имеет свои ограничения. Так, он не будет развернут, если не сможет обнаружить запрашиваемый порт. Сервис должен просматривать порты всех приложений. Использование портов, выбранных вручную, повышает опасность конфликта, особенно в случае, когда рабочая нагрузка распределяется на несколько кластеров, в которых не обязательно могут быть свободными одинаковые порты на узлах.

Другим ограничением на использование сервиса `NodePort` является то, что балансировщик нагрузки или программа клиента должны знать IP-адреса узлов. Статическая конфигурация (например, когда оператор вручную копирует IP-адреса узлов) может легко устареть (особенно у облачного провайдера), поскольку IP-адреса изменяются либо происходит замена узлов. Надежная система должна автоматически пополнять IP-адреса узлов или путем отслеживания виртуальных машин, назначаемых кластеру, или через прослушивание узлов, используя Kubernetes API.

Сервис `NodePort` относится к самым ранним сервисам. Далее мы увидим, что другие типы сервисов используют `NodePort` в качестве базовой структуры своей архитектуры. Данный сервис не стоит применять в одиночку, т. к. для выполнения запросов на соединение он требует от клиента знания IP-адресов хостов и узла. Когда мы будем рассматривать облачные сети, то увидим, каким образом `NodePort` используется для активации балансировщиков нагрузки.

В следующем разделе обсудим сервис по умолчанию — `ClusterIP`.

ClusterIP

Жизненный цикл IP-адресов подов совпадает с жизненным циклом самих подов и поэтому не подходит для использования в запросах. Сервисы помогают преодолеть эту особенность сетевой конфигурации подов. Сервис `ClusterIP` предоставляет внутренний балансировщик нагрузки с единственным IP-адресом, который распространяется на все обеспечивающие сервис (или готовые к работе) поды.

IP-адрес сервиса должен находиться в пределах CIDR, устанавливаемых командой service-cluster-ip-range API-сервера. Можно задать IP-адрес вручную или оставить поле .spec.cluster-IP пустым, чтобы адрес был присвоен автоматически.

Адрес сервиса ClusterIP — это виртуальный IP-адрес, который маршрутизируется только внутри кластера.

kube-proxy отвечает за маршрутизацию адреса сервиса ClusterIP ко всем подам приложения. В «нормальных» конфигурациях kube-proxy выполняет балансировку нагрузки на 4-м уровне, чего может быть недостаточно. Например, более старые поды могут испытывать большую нагрузку вследствие накопления долгоживущих соединений с клиентами. Или несколько клиентов, но выполняющих большое количество запросов, могут вызвать неравномерность распределения нагрузки.

Сервис `ClusterIP` в основном предназначен для случаев, когда работа приложений в кластере требует, чтобы балансировщик нагрузки был внутри того же кластера.

На рис. 5.5 показан развернутый сервис `ClusterIP`. Имя сервиса — это `App` с селектором, т. е. `App=App1`. Сервис развернут на двух подах. Под 1 и Под 5 отвечают условиям выбора для сервиса.

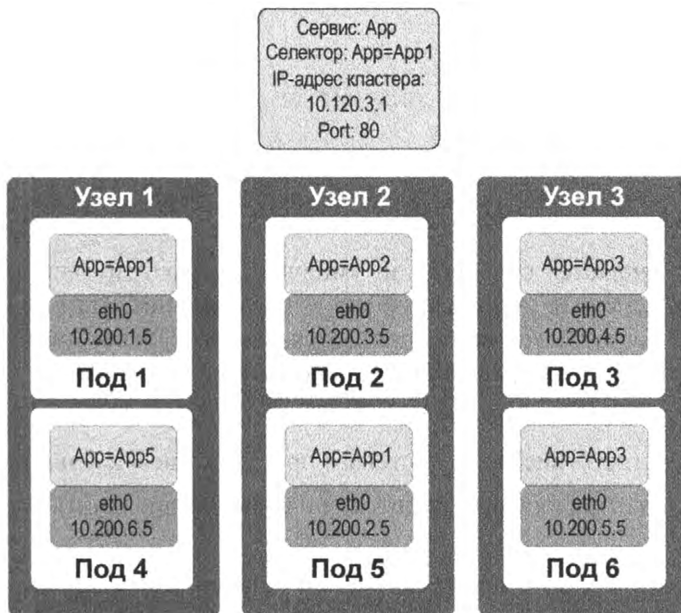


Рис. 5.5. Пример сервиса ClusterIP

Разберем соответствующие команды на примере нашего KIND-кластера.

Развернем сервис `ClusterIP` для использования с нашим веб-сервером на Go:

```
kubectl apply -f service-clusterip.yaml
service/clusterip-service created

kubectl describe svc clusterip-service
Name:              clusterip-service
Namespace:         default
Labels:            app=app
Annotations:       Selector: app=app
```

```
Type:             ClusterIP
IP:               10.98.252.195
Port:             <unset> 80/TCP
TargetPort:       8080/TCP
Endpoints:        <none>
Session Affinity: None
Events:           <none>
```

Имя сервиса `ClusterIP` распознаваемо в сети:

```
kubectl exec dnsutils -- host clusterip-service
clusterip-service.default.svc.cluster.local has address 10.98.252.195
```

Теперь мы можем получить доступ к конечной точке хостового API, используя кластерный IP-адрес 10.98.252.195, имя сервиса clusterip-service или IP-адрес пода 10.244.1.4 и порт 8080:

```
kubectl exec dnsutils -- wget -q -O- clusterip-service/host
NODE: kind-worker2, POD IP:10.244.1.4

kubectl exec dnsutils -- wget -q -O- 10.98.252.195/host
NODE: kind-worker2, POD IP:10.244.1.4

kubectl exec dnsutils -- wget -q -O- 10.244.1.4:8080/host
NODE: kind-worker2, POD IP:10.244.1.4
```

Сервис `ClusterIP` является сервисом по умолчанию. Поэтому вполне резонно рассмотреть, какие действия абстрагируются в этом сервисе. На самом деле список действий похож на тот, который выполняется в сетях Docker (см. *главы 2 и 3*), но сейчас у нас еще есть таблицы iptables для сервиса на всех узлах:

- Установить соответствие между парой VETH и подом.
- Установить соответствие между сетевым пространством имен и подом.
- Подтвердить идентификаторы процессов на узле и определить соответствующие поды.
- Установить соответствие между сервисами и правилами iptables.

Чтобы выполнить требуемые команды, нам нужно узнать, на каком рабочем узле развернут под, в нашем случае это kind-worker2:

```
kubectl get pods -o wide --field-selector spec.nodeName=kind-worker2 -l app=app

NAME                  READY STATUS  RESTARTS AGE   IP         NODE
app-9cc7d9df8-ffsm6   1/1   Running 0        7m23s 10.244.1.4 kind-worker2
```

Идентификаторы и имена контейнеров будут для вас разными.

Поскольку мы используем KIND, то можем вызвать docker ps и docker exec, чтобы получить информацию из рабочего узла kind-worker-2:

```
docker ps
CONTAINER ID  COMMAND                      PORTS                   NAMES
df6df0736958 "/usr/local/bin/entr..."                              kind-worker2
e242f11d2d00 "/usr/local/bin/entr..."                              kind-worker
a76b32f37c0e "/usr/local/bin/entr..."                              kind-worker3
07ccb63d870f "/usr/local/bin/entr..."  0.0.0.0:80->80/tcp,         kind-control-plane
                                       0.0.0.0:443->443/tcp,
                                       127.0.0.1:52321->6443/tcp
```

Идентификатор контейнера kind-worker2 есть df6df0736958. KIND был достаточно любезен (от *англ.* kind — любезный, милый), чтобы пометить каждый контейнер его именем, так что теперь мы можем обращаться к каждому рабочему узлу, используя его имя kind-worker2.

Посмотрим IP-адрес и таблицу маршрутизации для нашего пода app-9cc7d9df8-ffsm6:

```
kubectl exec app-9cc7d9df8-ffsm6 ip r
default via 10.244.1.1 dev eth0
10.244.1.0/24 via 10.244.1.1 dev eth0 src 10.244.1.4
10.244.1.1 dev eth0 scope link src 10.244.1.4
```

IP-адрес пода 10.244.1.4, по умолчанию доступный на интерфейсе eth0@if5 с адресом 10.244.1.1. Это соответствует интерфейсу 5 на поде veth45d1f3e8@if5:

```
kubectl exec app-9cc7d9df8-ffsm6 ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
qlen 1000
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
   inet 127.0.0.1/8 scope host lo
     valid_lft forever preferred_lft forever
   inet6 ::1/128 scope host
     valid_lft forever preferred_lft forever
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN
group default qlen 1000
   link/ipip 0.0.0.0 brd 0.0.0.0
3: ip6tnl0@NONE: <NOARP> mtu 1452 qdisc noop state DOWN group default qlen 1000
   link/tunnel6 :: brd ::
5: eth0@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group
default
   link/ether 3e:57:42:6e:cd:45 brd ff:ff:ff:ff:ff:ff link-netnsid 0
   inet 10.244.1.4/24 brd 10.244.1.255 scope global eth0
     valid_lft forever preferred_lft forever
   inet6 fe80::3c57:42ff:fe6e:cd45/64 scope link
     valid_lft forever preferred_lft forever
```

Проверим также пространство имен сети, ниже приведена выдача с узла ip:

```
docker exec -it kind-worker2 ip a
<trimmerd>
```

```
5: veth45d1f3e8@if5: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP group default
   link/ether 3e:39:16:38:3f:23 brd <>
   link-netns cni-ec37f6e4-a1b5-9bc9-b324-59d612edb4d4
   inet 10.244.1.1/32 brd 10.244.1.1 scope global veth45d1f3e8
      valid_lft forever preferred_lft forever
```

Команда `netns list` подтверждает, что сетевые пространства имен включают в себя наши поды, интерфейс к хостовому интерфейсу cni-ec37f6e4-a1b5-9bc9-b324-59d612edb4d4:

```
docker exec -it kind-worker2 /usr/sbin/ip netns list
cni-ec37f6e4-a1b5-9bc9-b324-59d612edb4d4 (id: 2)
cni-c18c44cb-6c3e-c48d-b783-e7850d40e01c (id: 1)
```

Посмотрим, какие процессы запущены внутри этого пространства имен. Для этого воспользуемся `docker exec` для выполнения команд внутри узла `kind-worker2`, являющегося хостом для пода и его пространства имен:

```
docker exec -it kind-worker2 /usr/sbin/ip netns pid
cni-ec37f6e4-a1b5-9bc9-b324-59d612edb4d4
4687
4637
```

Теперь можно запустить grep для каждого идентификатора процесса и посмотреть, что эти процессы выполняют:

```
docker exec -it kind-worker2 ps aux | grep 4687
root      4687  0.0   0.0  968     4 ?    Ss    17:00   0:00 /pause

docker exec -it kind-worker2 ps aux | grep 4737
root      4737  0.0   0.0 708376 6368 ?  Ssl 17:00 0:00 /opt/web-server
```

4737 — это идентификатор процесса, соответствующего запуску контейнера нашего веб-сервера на узле `kind-worker2`.

4687 — это родительский (пустой) контейнер, поддерживающий все наши пространства имен.

Посмотрим, что на рабочем узле случилось с iptables:

```
docker exec -it kind-worker2 iptables -L
Chain INPUT (policy ACCEPT)
Target                 prot  opt   source         destination
/* kubernetes service portals */
KUBE-SERVICES          all   -     anywhere       anywhere   ctstate NEW
/* kubernetes externally-visible service portals */
KUBE-EXTERNAL-SERVICES all   --    anywhere       anywhere   ctstate NEW
KUBE-FIREWALL          all   --    anywhere       anywhere

Chain FORWARD (policy ACCEPT)
Target                 prot  opt   source         destination
```

```
/* kubernetes forwarding rules */
KUBE-FORWARD           all    --    anywhere        anywhere
/* kubernetes service portals */
KUBE-SERVICES          all    --    anywhere        anywhere   ctstate NEW

Chain OUTPUT (policy ACCEPT)
Target                 prot   opt   source          destination
/* kubernetes service portals */
KUBE-SERVICES          all    --    anywhere        anywhere   ctstate NEW
KUBE-FIREWALL          all    --    anywhere        anywhere

Chain KUBE-EXTERNAL-SERVICES (1 references)
Target                 prot   opt   source          destination

Chain KUBE-FIREWALL (2 references)
Target                 prot   opt   source          destination
/* kubernetes firewall for dropping marked packets */
DROP                   all    --    anywhere        anywhere  mark match 0x8000/0x8000

Chain KUBE-FORWARD (1 references)
target                 prot   opt   source      destination
DROP                   all    --    anywhere    anywhere   ctstate INVALID
/*kubernetes forwarding rules*/
ACCEPT                 all    --    anywhere    anywhere   mark match 0x4000/0x4000
/*kubernetes forwarding conntrack pod source rule*/
ACCEPT                 all    --    anywhere    anywhere   ctstate RELATED,ESTABLISHED
/*kubernetes forwarding conntrack pod destination rule*/
ACCEPT                 all    --    anywhere    anywhere   ctstate RELATED,ESTABLISHED

Chain KUBE-KUBELET-CANARY (0 references)
Target                   prot   opt    source      destination

Chain KUBE-PROXY-CANARY (0 references)
Target                   prot  opt     source      destination

Chain KUBE-SERVICES (3 references)
Target                   prot  opt     source      destination
```

Kubernetes работает с множеством таблиц.

Рассмотрим немного подробнее таблицы iptables, отвечающие за развернутые нами сервисы. Попробуем узнать IP-адрес объекта clusterip-service — он понадобится для поиска соответствующих правил в iptables:

```
kubectl get svc clusterip-service

NAME                   TYPE       CLUSTER-IP     EXTERNAL-IP  PORT(S)  AGE
clusterip-service      ClusterIP  10.98.252.195  <none>       80/TCP   57m
```

Теперь с помощью cluster IP-адреса сервиса, 10.98.252.195, найдем правило в iptables:

```
docker exec -it kind-worker2 iptables -L -t nat | grep 10.98.252.195
/* default/clusterip-service: cluster IP */
KUBE-MARK-MASQ tcp -- !10.244.0.0/16   10.98.252.195 tcp dpt:80

/* default/clusterip-service: cluster IP */
KUBE-SVC-V7R3EVKW3DT43QQM tcp - anywhere 10.98.252.195 tcp dpt:80
```

Список всех правил в цепочке KUBE-SVC-V7R3EVKW3DT43QQM:

```
docker exec -it kind-worker2 iptables -t nat -L KUBE-SVC-V7R3EVKW3DT43QQM
Chain KUBE-SVC-V7R3EVKW3DT43QQM (1 references)
Target                    prot opt  source     destination
/* default/clusterip-service: */
KUBE-SEP-THJR2P3Q4C2QAEPT all   --  anywhere   anywhere
```

Конечные точки сервисов содержатся в цепочке KUBE-SEP-THJR2P3Q4C2QAEPT.

Посмотрим в iptables правила для этой цепочки:

```
docker exec -it kind-worker2 iptables -L KUBE-SEP-THJR2P3Q4C2QAEPT -t nat
Chain KUBE-SEP-THJR2P3Q4C2QAEPT (1 references)
Target          prot  opt   source          destination
/* default/clusterip-service: */
KUBE-MARK-MASQ all    --     10.244.1.4     anywhere
/* default/clusterip-service: */
DNAT            tcp    --     anywhere       anywhere  tcp to:10.244.1.4:8080
```

10.244.1.4:8080 — это одна из конечных точек сервиса, она же является поддерживающим сервис подом, что подтверждается выдачей команды kubectl get ep clusterip-service :

```
kubectl get ep clusterip-service

NAME                 ENDPOINTS           AGE
clusterip-service  10.244.1.4:8080     62m

kubectl describe ep clusterip-service
Name:           clusterip-service
Namespace:      default
Labels:         app=app
Annotations:    <none>
Subsets:
Addresses:               10.244.1.4
NotReadyAddresses:       <none>
Ports:
         Name    Port      Protocol
         ----    ----      --------
       <unset>   8080      TCP
Events: <none>
```

Рассмотрим теперь ограничения сервиса `ClusterIP`. Данный сервис регулирует внутренний трафик кластера, и он сталкивается с теми же проблемами, что имеют место для конечных точек. С увеличением размера сервиса все обновления замедляются. В *главе 2* мы обсуждали, что эту проблему можно частично решить путем использования виртуального сервера IPVS как прокси-режима для kube-proxy. Ниже в данной главе мы познакомимся с тем, как направить трафик в кластер, используя ингресс и сервис `LoadBalancer`.

Сервис ClusterIP — это сервис по умолчанию, кроме него есть и другие типы сервисов, например «безголовый» (headless) и `ExternalName`. `ExternalName` представляет собой специальный тип сервиса, предназначенный для доступа к сервисам вне кластера. Мы коротко уже касались безголового сервиса, когда рассматривали StatefulSet, в следующем разделе мы обсудим его в деталях.

Headless-сервис

Headless-сервис («безголовый», автономный) не является формальным типом сервиса (т. е. спецификации `.spec.type:Headless`). Headless-сервис имеет спецификацию .spec.clusterIP: «None». Это отличается от просто *незадания* IP-адреса кластера, когда Kubernetes автоматически просто назначает кластеру IP-адрес.

Если ClusterIP устанавливается в None, то сервис не поддерживает никакой балансировки нагрузки. Вместо этого он предоставляет всем подам, которые выбраны в группу и готовы к работе, только объект Endpoints и указатель на запись DNS для сервиса.

Headless-сервис является общепринятым методом наблюдения за конечными точками, не прибегая к взаимодействию с Kubernetes API. Получить запись из DNS гораздо проще интеграции с Kubernetes API и может быть вообще невозможно в случае программного обеспечения третьей стороны.

«Безголовый» сервис дает возможность разработчику создать в развертывании несколько копий пода. Вместо возвращения единственного IP-адреса, как это происходит в сервисе `ClusterIP`, здесь все IP-адреса конечных точек возвращаются через запрос. Их дальнейшее использование остается на усмотрение клиента. Чтобы посмотреть, как это все работает, реплицируем наше веб-приложение:

```
kubectl scale deployment app --replicas 4
deployment.apps/app scaled

kubectl get pods -l app=app -o wide
NAME                    READY  STATUS   IP          NODE
app-9cc7d9df8-9d5t8      1/1   Running  10.244.2.4  kind-worker
app-9cc7d9df8-ffsm6      1/1   Running  10.244.1.4  kind-worker2
app-9cc7d9df8-srxk5      1/1   Running  10.244.3.4  kind-worker3
app-9cc7d9df8-zrnvb      1/1   Running  10.244.3.5  kind-worker3
```

Теперь давайте развернем «безголовый» сервис:

```
kubectl apply -f service-headless.yml
service/headless-service created
```

Запрос DNS вернет нам все четыре IP-адреса для подов. Используя наш образ dnsutils, убедимся, что это действительно так:

```
kubectl exec dnsutils -- host -v -t a headless-service
Trying "headless-service.default.svc.cluster.local"
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id:45294
;; flags: qr aa rd; QUERY: 1, ANSWER: 4, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;headless-service.default.svc.cluster.local. IN A

;; ANSWER SECTION:
headless-service.default.svc.cluster.local.  30 IN A  10.244.2.4
headless-service.default.svc.cluster.local.  30 IN A  10.244.3.5
headless-service.default.svc.cluster.local.  30 IN A  10.244.1.4
headless-service.default.svc.cluster.local.  30 IN A  10.244.3.4

Received 292 bytes from 10.96.0.10#53 in 0 ms
```

Возвращенные в ответе на запрос IP-адреса соответствуют конечным точкам сервиса. Чтобы убедиться в этом, воспользуемся командой `kubectl describe` для конечных точек:

```
kubectl describe endpoints headless-service
Name:           headless-service
Namespace:      default
Labels:         service.kubernetes.io/headless
Annotations: endpoints.kubernetes.io/last-change-trigger-time:
2021-01-30T18:16:09Z
Subsets:
    Addresses:          10.244.1.4,10.244.2.4,10.244.3.4,10.244.3.5
    NotReadyAddresses: <none>
    Ports:
      Name     Port    Protocol
      ----     ----    --------
     <unset>   8080    TCP
Events:  <none>
```

Применение безголового сервиса ограничено особыми случаями, и обычно он не используется при развертываниях. Как мы уже упоминали в разделе `StatefuSets`, если разработчик предоставляет клиенту право решать, какую конечную точку использовать, то обращение к headless-сервису представляется разумным. В качестве примеров безголового сервиса можно привести кластеризованные базы данных и приложения, в которых уже на уровне кода реализована балансировка нагрузки на стороне клиента.

Следующим типом сервиса, который мы рассмотрим, будет `ExternalName`, который помогает в миграции внешних по отношению к кластеру сервисов. Он также предлагает некоторые преимущества в работе с DNS внутри кластера.

Сервис ExternalName

ExternalName — это специальный тип сервиса, который не имеет селекторов, а вместо них использует имена DNS.

Если просмотреть хост `ext-service.default.svc.cluster.local`, то служба DNS кластера вернет запись CNAME для database.mycompany.com:

```
apiVersion: v1
kind: Service
metadata:
   name: ext-service
   spec:
      type: ExternalName

      externalName: database.mycompany.com
```

Если разработчики осуществляют миграцию приложения в Kubernetes, но при этом в приложении остаются обращения к источникам вне кластера, то сервис `ExternalName` позволит определить запись DNS, внутреннюю по отношению к кластеру, вне зависимости от того, где на самом деле запущен сервис.

DNS будет осуществлять поиск, как показано в следующем примере:

```
  kubectl exec -it dnsutils -- host -v -t a github.com
Trying "github.com.default.svc.cluster.local"
Trying "github.com.svc.cluster.local"
Trying "github.com.cluster.local"
Trying "github.com"
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 55908
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;github.com.                    IN      A

;; ANSWER SECTION:
github.com.         30          IN      A        140.82.112.3

Received 54 bytes from 10.96.0.10#53 in 18 ms
```

Также сервис `ExternalName` позволяет разработчикам отображать сервис на DNS-имя.

Развернем внешний сервис:

```
kubectl apply -f service-external.yml
service/external-service created
```

Запись A для github.com возвращается через запрос `external-service query`:

```
kubectl exec -it dnsutils -- host -v -t a external-service
Trying "external-service.default.svc.cluster.local"
```

```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 11252
;; flags: qr aa rd; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;external-service.default.svc.cluster.local. IN A

;; ANSWER SECTION:
external-service.default.svc.cluster.local. 24 IN CNAME github.com.
github.com.            24       IN      A         140.82.112.3

Received 152 bytes from 10.96.0.10#53 in 0 ms
```

Запрос CNAME для внешнего сервиса возвращает github.com:

```
kubectl exec -it dnsutils -- host -v -t cname external-service
Trying "external-service.default.svc.cluster.local"

;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 36874
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;external-service.default.svc.cluster.local. IN CNAME

;; ANSWER SECTION:
external-service.default.svc.cluster.local. 30 IN CNAME github.com.

Received 126 bytes from 10.96.0.10#53 in 0 ms
```

Направлять трафик на безголовый сервис через DNS-запись возможно, но не рекомендуется. Использование DNS — это весьма неэффективный подход с точки зрения балансировки нагрузки, поскольку ПО использует разные (и часто упрощенные или неинтуитивные) способы работы с A и AAAA DNS-записями, которые возвращают множественные адреса. Например, обычным делом для программ является выбор первого IP-адреса из ответа на запрос или из кеша и бесконечное использование одного и того же IP-адреса. Если вы хотите направлять трафик на адрес DNS-сервиса, то используйте (стандартный) сервис ClusterIP или сервис LoadBalancer.

«Правильный» способ применения безголового сервиса — это запрос записи A/AAAA DNS-сервера и последующее использование полученных данных в балансировщиках нагрузки на стороне сервера или стороне клиента.

Большинство рассмотренных типов сервисов предназначаются для управления внутренним трафиком кластерной сети. В следующем разделе мы обсудим, каким образом направлять запросы в кластер с помощью сервисов `LoadBalancer` и ингресс.

Сервис LoadBalancer

Сервис LoadBalancer экспонирует сервисы, являющиеся внешними по отношению к кластерной сети. Данный сервис сочетает в себе функции сервиса `NodePort` с интеграцией внешних компонентов, таких как, например, балансировщик нагрузки об-

лачного провайдера. Сервис `LoadBalancer` обслуживает трафик на 4-м уровне (в отличие от ингресса, который предназначен для трафика 7-го уровня), так что он будет работать с любыми службами TCP или UDP — при условии, что выбранный балансировщик нагрузки поддерживает трафик 4-го уровня.

Конфигурации и опции балансировщика нагрузки сильно зависят от облачного провайдера. Например, некоторые будут поддерживать спецификацию `.spec.loadBalancerIP` (с требуемыми установками), а некоторые будут ее игнорировать:

```
apiVersion: v1
kind: Service
metadata:
  name: demo-service
spec:
  selector:
    app: demo
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
clusterIP: 10.0.5.1
type: LoadBalancer
```

Как только балансировщик нагрузки предоставлен, его IP-адрес записывается в поле `.status.loadBalancer.ingress.ip`.

Сервисы LoadBalancer удобно использовать для внешнего экспонирования TCP или UDP-служб. Трафик поступает на балансировщик нагрузки по его публичному IP-адресу и TCP порту 80, определяемому в `spec.ports[*].port`, и направляется на IP-адрес кластера, 10.0.5.1, и далее на порт 8080 контейнера, `spec.ports[*].targetPort`. В примере не показан `.spec.ports[*].nodePort`, если он не установлен, то Kubernetes сам выберет порт для сервиса.

Порт spec.ports[*].targetPort для сервиса должен согласовываться с портом контейнера в поде, spec.container[*].ports.containerPort, а также с протоколом.

В противном случае это будет воспринято как отсутствие точки с запятой в Kubernetes.

Схема на рис. 5.6 демонстрирует, каким образом сервис `LoadBalancer` выстраивается на основе других типов сервиса. Облачный балансировщик нагрузки задает распределение трафика — мы подробно обсудим этот вопрос в следующей главе.

Продолжим работу с нашим веб-сервером на Go и развернем для него сервис LoadBalancer.

Поскольку мы работаем на локальной машине, а не используем специальные службы типа AWS, GCP или Azure, то можно в качестве примера балансировщика нагрузки взять MetalLB. Проект MetalLB предоставляет пользователям средства, позволяющие развернуть в кластерах bare-metal балансировщики нагрузки.

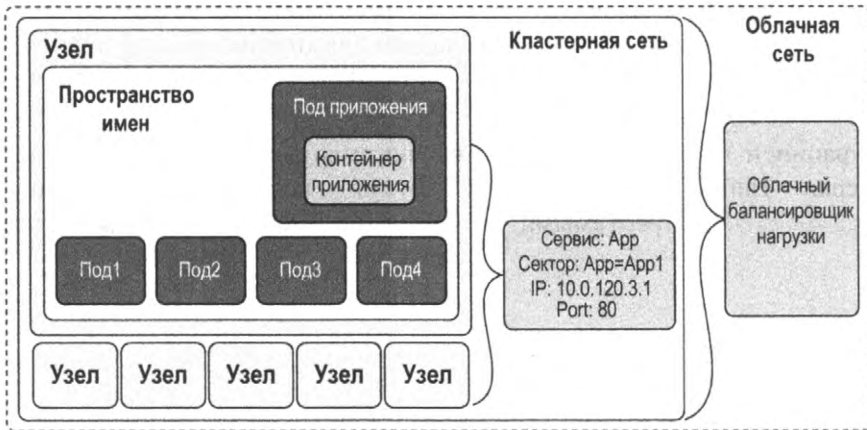


Рис. 5.6. Сервис LoadBalancer

Рассматриваемый ниже пример является модификацией примера на развертывание KIND.

Первым шагом будет создание отдельного пространства имен для MetalLB:

```
kubectl apply -f mlb-ns.yaml
namespace/metallb-system created
```

Для входа в кластер балансировщика нагрузки участникам MetalLB требуется секретный ключ — создадим его для использования в нашем кластере:

```
kubectl create secret generic -n metallb-system memberlist
--from-literal=secretkey="$(openssl rand -base64 128)"
secret/memberlist created
```

Теперь мы можем развернуть MetalLB!

```
kubectl apply -f ./metallb.yaml
podsecuritypolicy.policy/controller created
podsecuritypolicy.policy/speaker created
serviceaccount/controller created
serviceaccount/speaker created
clusterrole.rbac.authorization.k8s.io/metallb-system:controller created
clusterrole.rbac.authorization.k8s.io/metallb-system:speaker created
role.rbac.authorization.k8s.io/config-watcher created
role.rbac.authorization.k8s.io/pod-lister created
clusterrolebinding.rbac.authorization.k8s.io/metallb-system:controller created
clusterrolebinding.rbac.authorization.k8s.io/metallb-system:speaker created
rolebinding.rbac.authorization.k8s.io/config-watcher created
rolebinding.rbac.authorization.k8s.io/pod-lister created
daemonset.apps/speaker created
deployment.apps/controller created
```

Как видите, происходит развертывание многих объектов, и поэтому приходится подождать, пока процесс завершится. Мы можем следить за разворачиванием ресурсов, используя опцию `-watch` в пространстве имен `metallb-system`:

```
kubectl get pods -n metallb-system --watch

 NAME                           READY   STATUS              RESTARTS   AGE
controller-5df88bd85d-mvgqn 0/1     ContainerCreating     0          10s
speaker-5knqb                 1/1     Running               0          10s
speaker-k79c9                 1/1     Running               0          10s
speaker-pfs2p                 1/1     Running               0          10s
speaker-sl7fd                 1/1     Running               0          10s
controller-5df88bd85d-mvgqn 1/1     Running               0          12s
```

Чтобы завершить конфигурирование, нужно представить балансировщику MetalLB диапазон IP-адресов, которые он контролирует. Этот диапазон должен быть в сети Docker KIND:

```
docker network inspect -f '{{.IPAM.Config}}' kind
[{172.18.0.0/16 172.18.0.1 map[]} {fc00:f853:ccd:e793::/64
fc00:f853:ccd:e793::1 map[]}]
```

Адреса 172.18.0.0/16 — это наша локальная сеть Docker.

Мы хотим, чтобы диапазон IP-адресов нашего балансировщика происходил из этого подмножества. Задавая объект `ConfigMap`, мы можем сконфигурировать MetalLB на использование, например, адресов от 172.18.255.200 до 172.18.255.250.

Объект ConfigMap будет выглядеть примерно так:

```
apiVersion: v1
kind: ConfigMap
metadata:
   namespace: metallb-system
   name: config
data:
   config: |
     address-pools:
     - name: default
       protocol: layer2
       addresses:
       - 172.18.255.200-172.18.255.250
```

Применим его для MetalLB:

```
kubectl apply -f ./metallb-configmap.yaml
```

Теперь развернем балансировщик нагрузки для нашего веб-приложения:

```
kubectl apply -f services-loadbalancer.yaml
service/loadbalancer-service created
```

Чтобы потренироваться, попробуем масштабировать наше веб-приложение на 10 реплик — конечно, если у вас есть для этого ресурсы:

```
kubectl scale deployment app --replicas 10

kubectl get pods -o wide
```

```
NAME                   READY STATUS   RESTARTS AGE  IP          NODE
app-7bdb9ffd6c-b5x7m   2/2   Running  0        26s  10.244.3.15 kind-worker
app-7bdb9ffd6c-bqtf8   2/2   Running  0        26s  10.244.2.13 kind-worker2
app-7bdb9ffd6c-fb9sf   2/2   Running  0        26s  10.244.3.14 kind-worker
app-7bdb9ffd6c-hrt7b   2/2   Running  0        26s  10.244.2.7  kind-worker2
app-7bdb9ffd6c-l2794   2/2   Running  0        26s  10.244.2.9  kind-worker2
app-7bdb9ffd6c-l4cfx   2/2   Running  0        26s  10.244.3.11 kind-worker2
app-7bdb9ffd6c-rr4kn   2/2   Running  0        23s  10.244.3.10 kind-worker
app-7bdb9ffd6c-s4k92   2/2   Running  0        26s  10.244.3.13 kind-worker
app-7bdb9ffd6c-shmdt   2/2   Running  0        26s  10.244.1.12 kind-worker3
app-7bdb9ffd6c-v87f9   2/2   Running  0        26s  10.244.1.11 kind-worker3
app2-658bcd97bd-4n888 1/1    Running  0        35s  10.244.2.6  kind-worker3
app2-658bcd97bd-mnpkp 1/1    Running  0        35s  10.244.3.7  kind-worker
app2-658bcd97bd-w2qkl 1/1    Running  0        35s  10.244.3.8  kind-worker
dnsutils               1/1   Running  0        75s  10.244.1.2  kind-worker3
postgres-0             1/1   Running  0        75s  10.244.1.4  kind-worker3
postgres-1             1/1   Running  0        75s  10.244.3.4  kind-worker
```

Теперь можно протестировать развернутый балансировщик нагрузки.

Поскольку несколько реплик развернуты для нашего приложения вне балансировщика, нам понадобится внешний IP-адрес 172.18.255.200 этого сервиса:

```
kubectl get svc loadbalancer-service
NAME                 TYPE         CLUSTER-IP    EXTERNAL-IP
PORT(S)      AGE
loadbalancer-service LoadBalancer  10.99.24.220  172.18.255.200
80:31276/TCP  52s

kubectl get svc/loadbalancer-service o=jsonpath='{.status.loadBalancer.ingress[0].ip}'
172.18.255.200
```

Так как Docker для Mac или Windows не экспонирует хосту сеть KIND, то мы не можем получить прямой доступ к IP-адресу 172.18.255.200 балансировщика нагрузки в частной сети Docker. Чтобы обойти это препятствие, мы можем присоединить контейнер Docker к сети KIND и выполнить cURL.

Мы запустим локально образ Docker с именем nicolaka/netshoot, присоединим к KIND сеть Docker и будем посылать запросы на наш балансировщик нагрузки MetalLB.

Если мы проделаем это несколько раз, то сможем увидеть, как балансировщик выполняет свою работу, направляя трафик на различные поды:

```
docker run --network kind -a stdin -a stdout -i -t nicolaka/netshoot
curl 172.18.255.200/host
NODE: kind-worker, POD IP:10.244.2.7

docker run --network kind -a stdin -a stdout -i -t nicolaka/netshoot
curl 172.18.255.200/host
NODE: kind-worker, POD IP:10.244.2.9
```

```
docker run --network kind -a stdin -a stdout -i -t nicolaka/netshoot
curl 172.18.255.200/host
NODE: kind-worker3, POD IP:10.244.3.11

docker run --network kind -a stdin -a stdout -i -t nicolaka/netshoot
curl 172.18.255.200/host
NODE: kind-worker2, POD IP:10.244.1.6

docker run --network kind -a stdin -a stdout -i -t nicolaka/netshoot
curl 172.18.255.200/host
NODE: kind-worker, POD IP:10.244.2.9
```

Каждый новый запрос сервис MetalLB направляет на другой под. Балансировщик нагрузки, как и другие сервисы, использует для подов селекторы и метки — мы можем посмотреть это, вызвав команду `kubectl describe endpoints loadbalancer-service`. IP-адреса подов совпадают с нашими результатами выполнения cURL:

```
 kubectl describe endpoints loadbalancer-service
Name:         loadbalancer-service
Namespace:    default
Labels:       app=app
Annotations:  endpoints.kubernetes.io/last-change-trigger-time:
2021-01-30T19:59:57Z
  Subsets:
    Addresses:
    10.244.1.6,
    10.244.1.7,
    10.244.1.8,
    10.244.2.10,
    10.244.2.7,
    10.244.2.8,
    10.244.2.9,
    10.244.3.11,
    10.244.3.12,
    10.244.3.9
    NotReadyAddresses: <none>
    Ports:
        Name         Port    Protocol
        ----         ----    --------
      service-port 8080    TCP
Events: <none>
```

Важно напомнить, что сервисы балансировки нагрузки требуют особой интеграции и не будут работать без поддержки облачного провайдера или без установленного специального ПО типа MetalLB.

Как правило, эти балансировщики не работают на уровне 7, т. е. они не могут эффективно распределять HTTP(S) запросы. Существует взаимно однозначное соответствие между балансировщиком нагрузки и приложением Kubernetes, озна-

чающее, что все посылаемые на конкретный балансировщик запросы должны обрабатываться соответствующим приложением.

Хотя он и не является сетевым, но важно упомянуть здесь сервис Horizontal Pod Autoscaler, который масштабирует поды в контроллере репликаций, развертывании, ReplicaSet или Stateful на базе использования процессора.

Мы можем масштабировать наше приложение в зависимости от требований пользователей, не производя каких-либо изменений в конфигурациях отдельных компонентов. Kubernetes и сервис LoadBalancer выполняют все необходимые операции, оказывая существенную помощь разработчикам и администраторам систем. В следующей главе мы рассмотрим, как такие ситуации решаются с использованием облачных сервисов автоматического масштабирования.

Сервисы Kubernetes — устранение проблем

Приведем несколько рекомендаций по устранению проблем, возникающих в связи с конечными точками или сервисами:

- Снятие метки с пода позволяет ему продолжать функционировать, но при этом также обновляется конечная точка и сервис. Контроллер конечных точек исключит данный непомеченный под из объектов конечных точек, а компонент развертывания запустит другой под. Это позволит вам устранить проблемы, возникшие в связи с подом, с которого сняли метку, но при этом не затронуть сервис для конечных пользователей. Авторы проделывали эту операцию несчетное число раз, она также была разобрана в примерах предыдущего раздела.
- Есть два теста, которые посылают информацию о работоспособности пода на Kubelet и другие компоненты Kubernetes.
- В YAML-декларации могут быть допущены ошибки, обязательно следите за соответствием портов сервиса и подов.
- Сетевые политики обсуждались в *главе 3* — они также могут привести к прерыванию взаимодействия подов между собой и с сервисами. Если ваша кластерная сеть использует сетевые политики, то убедитесь, что их установки соответствуют потоку трафика приложения.
- Не забывайте использовать диагностики типа `dnsutils` для пода; `netshoot` для подов в кластерной сети является полезным инструментом отладки.
- Если время синхронизации конечных точек с кластером слишком большое, т. е. несколько опций, которые могут быть установлены в Kubelet, чтобы контролировать скорость реакции на изменение в среде Kubernetes:

  `--kube-api-qps`

  Устанавливает число запросов в секунду, которые Kubelet будет использовать при взаимодействии с Kubernetes API-сервером, по умолчанию 5.

  `--kube-api-burst`

  Временно разрешает числу API-запросов увеличиться до этого значения, по умолчанию 10.

`--iptables-sync-period`

Максимальный интервал, показывающий частоту обновления правил iptables (например, 5 сек., 1 мин, 2 часа 22 мин). Должен быть больше 0, по умолчанию 30 сек.

`--ipvs-sync-period duration`

Максимальный интервал обновления правил IPVS. Значение должно быть положительным, по умолчанию 30 сек.

- Для больших кластеров рекомендуется увеличить значения указанных опций, но не забывайте, что это приводит к повышенной нагрузке на ресурсы как Kubelet, так и API-сервера.

Эти рекомендации помогут устранить проблемы, их стоит иметь в виду, когда число подов и сервисов в кластере увеличивается.

Различные типы сервисов демонстрируют, насколько мощными являются сетевые абстракции Kubernetes. Мы подробно рассмотрели, как работает вся линейка инструментов. Разработчики приложений в Kubernetes теперь знают, какой сервис надлежит выбрать для оптимального удовлетворения их требований. А также сетевые администраторы, ранее вручную вносившие IP-адреса в балансировщики нагрузки, могут теперь передать Kubernetes выполнение этой работы.

Мы перечислили далеко не все, что могут делать сервисы. Каждая новая версия Kubernetes содержит дополнительные опции и настройки для запуска сервисов. Старайтесь тестировать каждый сервис на ваших задачах, чтобы убедиться, что используется именно тот тип, который приводит к оптимальной работе вашего приложения в сети Kubernetes.

Сервис `LoadBalancer` — это только один из типов сервисов, который управляет входящим трафиком кластера, экспонируя для внешних пользователей HTTP(S) сервисы, находящиеся «позади» балансировщика нагрузки. Ингрессы поддерживают маршрутизацию на основе пути, которая позволяет различным сервисам обслуживать различные HTTP-пути. В следующем разделе мы обсудим ингресс и каким образом он управляет доступом к ресурсам кластера.

Ингресс

Ингресс — это собственный балансировщик нагрузки Kubernetes уровня 7 (HTTP), который доступен извне — в отличие от сервиса `ClusterIP` уровня 4, который является внутренним к кластеру. Он считается стандартным выбором при экспонировании приложений HTTP(S) внешним пользователям. Ингресс может быть единственной точкой входа для API или архитектуры на базе микросервисов. Трафик может быть направлен на сервисы на основе HTTP-данных в запросе. Ингресс — это конфигурируемая спецификация (с многими реализациями), предназначенная для передачи HTTP-трафика на сервисы Kubernetes. Рис. 5.7 показывает компоненты ингресса.

Рис. 5.7. Архитектура ингресса

Чтобы управлять трафиком в кластере с помощью ингресса, требуются два компонента: контроллер и правила. Контроллер управляет подами ингресса, а установленные правила определяют, каким образом осуществляется маршрутизация трафика.

Контроллеры и правила ингресса

Мы будем называть реализации ингресса — *ингресс-контроллерами*. В Kubernetes под контроллером понимается программа, отвечающая за управление определенным типом ресурса и обеспечивающая желаемое функционирование системы.

Существуют два основных типа контроллеров:

- Контроллеры внешних балансировщиков нагрузки.
- Контроллеры внутренних балансировщиков нагрузки.

Контроллеры внешних балансировщиков создают балансировщик нагрузки, существующий «снаружи» кластера как, например, продукт, предоставляемый облаком.

Контроллер внутренних балансировщиков разворачивает балансировщик, работающий внутри кластера и не решающий напрямую задачу направления пользователей на балансировщик нагрузки. Есть огромное число вариантов, как администратор сети может использовать внутренние балансировщики нагрузки, например запуск балансировщика на группе определенных узлов и направление трафика на эти узлы. Основной мотивацией к выбору внутреннего балансировщика нагрузки является снижение издержек. Внутренний балансировщик нагрузки на входе может направлять трафик на многие объекты ингресса, тогда как контроллер внешнего балансировщика нагрузки обычно требует одного балансировщика на один вход. Поскольку большинство облачных сервисов требуют оплату в зависимости от баланса нагрузки, то будет дешевле поддерживать один облачный балансировщик нагрузки, который потом развернется в кластере, чем несколько облачных балансировщиков. Однако учтите, что такой подход увеличивает операционные издержки,

ухудшает прозрачность функционирования и повышает стоимость расчетов, так что смотрите, стоит ли игра свеч. У многих компаний есть плохая привычка экономить на нестандартных облачных решениях.

Рассмотрим спецификацию ингресс-контроллера. Как и в сервисах `LoadBalancer`, большинство атрибутов спецификации являются универсальными, однако разные ингресс-контроллеры имеют разные функции и воспринимают разные конфигурации. Начнем с базового описания:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
   name: basic-ingress
spec:
  rules:
  - http:
      paths:
# Посылать все запросы /demo на demo-service
       - path: /demo
        pathType: Prefix
        backend:
          service:
            name: demo-service
            port:
              number: 80
# Посылать все остальные запросы на main-service.
defaultBackend:
   service:
     name: main-service
     port:
       number: 80
```

Пример иллюстрирует типичные черты ингресса. Он направляет трафик `/demo` на один сервис, а весь остальной трафик — на другой. Ингрессы имеют «бэкенд по умолчанию», на который направляется запрос, если ни одно правило не нашло соответствия. Для многих ингресс-контроллеров это может быть описано в самой конфигурации контроллера, а многие поддерживают поле `.spec.defaultBackend`. Ингрессы имеют различные возможности задания пути. На данный момент их три:

Exact

Совпадают только указанный путь и данный путь (включая полный путь / путь не задан).

Prefix

Совпадают все пути, начинающиеся с заданного.

ImplementationSpecific

Определяется в зависимости от текущего ингресс-контроллера.

Если путь в запросе совпадает с несколькими путями, то из них выбирается наиболее полный. Например, если заданы правила для /first и /first/second, то все запросы, начинающиеся с /first/second, будут направлены на бэкенд для /first/second. Если путь совпадает с полным путем и с префиксом, то запрос пойдет на бэкенд для полного пути.

Ингресс-контроллеры могут использовать в правилах имена хостов:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
   name: multi-host-ingress
spec:
  rules:
  - host: a.example.com
    http:
       paths:
       - pathType: Prefix
         path: "/"
         backend:
           service:
             name: service-a
             port:
               number: 80
  - host: b.example.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
           service:
              name: service-b
              port:
                number: 80
```

В этом примере мы обслуживаем трафик на a.example.com от одного сервиса и трафик на b.example.com — от другого. Это похоже на виртуальные хосты веб-серверов. Если захотите, то можете установить хостовые правила, чтобы использовать один балансировщик нагрузки, и IP-адрес, когда надо обслуживать несколько уникальных доменов.

Ингресс-контроллеры поддерживают TLS:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
   name: demo-ingress-secure
spec:
   tls:
```

```
  - hosts:
      - https-example.com
    secretName: demo-tls
  rules:
    - host: https-example.com
      http:
        paths:
        - path: /
          pathType: Prefix
          backend:

          service:
             name: demo-service
             port:
               number: 80
```

Конфигурация TLS содержит прямую ссылку на секретное имя Kubernetes в команде `.spec.tls.[*].secretName`. Как показано ниже, ингресс-контроллеры получают серфикат TLS и ключ через установки полей `.data."tls.crt"` и `.data."tls.key"`:

```
apiVersion: v1
kind: Secret
metadata:
    name: demo-tls
type: kubernetes.io/tls
data:
   tls.crt: cert, encoded in base64
   tls.key: key, encoded in base64
```

Если вы не хотите задавать вручную данные сертификатов, то можете использовать cert-manager для автоматического получения и обновления сертификатов.

Мы уже упоминали в начале, что ингресс — это просто спецификация с очень отличающимися друг от друга реализациями. В одном кластере можно использовать несколько ингресс-контроллеров — через разные установки в классе `IngressClass`. Класс представляет собой ингресс-контроллер и соответственно является реализацией конкретного ингресса.

Аннотациями в Kubernetes должны быть строки. Поскольку `true` и `false` имеют заданные нестрочные значения, вы не можете установить аннотацию в `true` или `false` без использования кавычек. Допускаются `"true"` и `"false"`. Это давно известная ошибка, с которой часто встречаются при задании приоритета класса по умолчанию.

Класс `IngressClass` впервые появился в версии Kubernetes 1.18. В более ранних версиях аннотация ингрессов как `kubernetes.io/ingress.class` была принятым стандартом, но это требовало поддержки данного стандарта всеми инсталлированными

ингресс-контроллерами. Ингрессы могут выбирать себе класс путем задания имени класса в поле `.spec.ingressClassName`.

Если по умолчанию задано более одного ингресс-класса, то Kubernetes не позволит вам создать ингресс без ингресс-класса или удалить ингресс-класс из существующего ингресса. Вы можете использовать контроль доступа, чтобы отслеживать ситуации, когда несколько ингресс-классов будут помечены как «default».

Ингресс поддерживает только HTTP(S)-запросы, чего недостаточно, если ваш сервис использует другой протокол (например, большинство баз данных используют свои собственные протоколы). Некоторые ингресс-контроллеры, такие как NGINX, поддерживают TCP и UDP, но это не является нормой.

А теперь давайте развернем ингресс-контроллер, чтобы добавить правила ингресса в наш веб-сервер на Go.

Когда мы разворачивали KIND-кластер, то должны были добавить несколько опций, чтобы обеспечить развертывание ингресс-контроллера:

- extraPortMapping позволяет локальному хосту делать запросы на ингресс-контроллер через порты 80/443.
- Метки узлов (node-labels) позволяют ингресс-контроллеру запускаться только на определенном узле (узлах), отвечающих условиям группирования.

Есть много ингресс-контроллеров на выбор. Система Kubernetes не имеет ингресс-контроллера по умолчанию, в отличие от остальных компонентов. Пользователи Kubernetes поддерживают ингресс-контроллеры AWS, GCE и Nginx. В табл. 5.1 приведено несколько распространенных ингрессов.

Таблица 5.1. *Краткий список ингресс-контроллеров*

Название	Коммерческая поддержка	Движок	Поддержка протокола	SSL-терминация
Ambassador ingress controller	есть	Envoy	gRPC, HTTP/2, WebSockets	есть
Community ingress Nginx	нет	NGINX	gRPC, HTTP/2, WebSockets	есть
NGINX Inc. ingress	есть	NGINX	HTTP, Websocket, gRPC	есть
HAProxy ingress	есть	HAProxy	gRPC, HTTP/2, WebSockets	есть
Istio Ingress	нет	Envoy	HTTP, HTTPS, gRPC, HTTP/2	есть
Kong ингресс-контроллер для Kubernetes	есть	Lua поверх NGINX	gRPC, HTTP/2	есть
Traefik Kubernetes ingress	есть	Traefik	HTTP/2, gRPC, WebSockets	есть

При выборе ингресса для ваших кластеров рекомендуется учитывать следующие факторы:

- Поддержка протокола: нужно ли вам что-либо кроме TCP/UDP, например интеграция gRPC или Websocket?
- Коммерческая поддержка: нужна ли она вам?
- Дополнительные опции: требуется ли для вашего приложения идентификация JWT/oAuth2 или шаблон circuit-breaker?
- Опции API-интерфейса: нужен ли специальный функционал типа ограничения скорости передачи?
- Распределение трафика: требует ли ваше приложение поддержки специализированного распределения трафика типа отзеркаливания или canary A/B-тестирования?

Для нашего примера мы выбрали Community-версию ингресс-контроллера NGINX.

Список ингресс-контроллеров для выбора поддерживается в Kubernetes.io.

Развернем ингресс-контроллер NGINX в нашем кластере KIND:

```
kubectl apply -f ingress.yaml
namespace/ingress-nginx created
serviceaccount/ingress-nginx created
configmap/ingress-nginx-controller created
clusterrole.rbac.authorization.k8s.io/ingress-nginx created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created
role.rbac.authorization.k8s.io/ingress-nginx created
rolebinding.rbac.authorization.k8s.io/ingress-nginx created
service/ingress-nginx-controller-admission created
service/ingress-nginx-controller created
deployment.apps/ingress-nginx-controller created
validatingwebhookconfiguration.admissionregistration.k8s.io/
ingress-nginx-admission created
serviceaccount/ingress-nginx-admission created
clusterrole.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
role.rbac.authorization.k8s.io/ingress-nginx-admission created
rolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
job.batch/ingress-nginx-admission-create created
job.batch/ingress-nginx-admission-patch created
```

Как и с остальными развертываниями, мы должны подождать, пока контроллер не будет готов к работе. С помощью следующей команды мы можем убедиться, что наш контроллер установлен:

```
kubectl wait --namespace ingress-nginx \
>       --for=condition=ready pod \
>       --selector=app.kubernetes.io/component=controller \
>       --timeout=90s
pod/ingress-nginx-controller-76b5f89575-zps4k condition met
```

Контроллер развернут в кластере, и теперь мы готовы написать правила ингресса для нашего приложения.

Задание правил ингресса

YAML-манифест определяет несколько правил ингресса для использования в нашем веб-сервере:

```
kubectl apply -f ingress-rule.yaml
ingress.extensions/ingress-resource created

kubectl get ingress
NAME                    CLASS    HOSTS   ADDRESS   PORTS   AGE
ingress-resource        <none>   *                 80      4s
```

С помощью describe мы можем увидеть все бэкенды, относящиеся к сервису ClusterIP и подам:

```
kubectl describe ingress
Name:          ingress-resource
Namespace:      default
Address:
Default backend: default-http-backend:80 (<error:
endpoints "default-http-backend" not found>)
Rules:
   Host     Path   Backends
   ----     ----   --------
   *
/host clusterip-service:8080 (
10.244.1.6:8080,10.244.1.7:8080,10.244.1.8:8080)
Annotations: kubernetes.io/ingress.class: nginx
Events:
   Type   Reason Age  From                         Message
   ----   ------ ---- ----                         -------
   Normal Sync   17s  nginx-ingress-controller     Scheduled for sync
```

Наше правило ингресса относится только к маршруту `/host` и будет направлять запросы на наш сервис `clusterip-service:8080`:

```
curl localhost/host
NODE: kind-worker2, POD IP:10.244.1.6
curl localhost/healthz
```

Теперь мы видим, насколько мощным инструментом являются ингрессы. Выполним еще одно развертывание и развернем также сервис `ClusterIP`.

Это развертывание и сервис будут отвечать на запросы к /data:

```
kubectl apply -f ingress-example-2.yaml
deployment.apps/app2 created
service/clusterip-service-2 configured
ingress.extensions/ingress-resource-2 configured
```

Итак, работают и /host, и /data, но отправляются на разные сервисы:

```
curl localhost/host
NODE: kind-worker2, POD IP:10.244.1.6

curl localhost/data
Database Connected
```

Поскольку ингресс работает на уровне 7, то существует множество возможностей маршрутизации трафика, например через заголовок хоста или путь URI.

Для более сложных схем маршрутизации трафика требуется развертывание в кластерной сети технологии service mesh (сеть сервисов). В следующем разделе мы рассмотрим эту технологию подробно.

Технология service mesh

Новый кластер, запущенный с опциями по умолчанию, имеет определенные ограничения. Давайте посмотрим, что это за ограничения и каким образом service mesh может снять некоторые из них. *Service mesh* — это инфраструктурный слой со своим API, предназначенный для управления взаимодействием между сервисами.

Если посмотреть с точки зрения безопасности, то весь трафик между подами внутри кластера не шифруется и для каждого сервиса его создатели должны предусмотреть отдельную процедуру мониторинга. Ранее мы рассмотрели типы сервисов, но мы не обсуждали, каким образом можно модифицировать поды, на которых запущены сервисы.

Технология Service mesh поддерживает несколько типов развертывания, она также поддерживает плавное обновление и повторное создание подов (аналогично тому, как это делает Canary). С точки зрения разработчика, возможность внесения помех (ошибок) в сеть является полезной опцией, но она напрямую не поддерживается в сетях Kubernetes, развернутых с установками по умолчанию. Используя Service mesh, разработчики могут добавить к приложениям тестирование на ошибки, не удаляя при этом поды, а внося с помощью Service mesh задержки — так или иначе, но любое приложение должно иметь встроенное тестирование на ошибки или функцию автоматического выключения.

Service mesh в кластерных сетях Kubernetes, развернутых с установками по умолчанию, предоставляет следующие функции:

Обнаружение сервиса

Вместо использования DNS для обнаружения сервиса эту задачу выполняет Service mesh, тем самым отменяя необходимость прописывать обнаружение сервиса в каждом отдельном приложении.

Балансировка нагрузки

Service mesh использует более эффективные алгоритмы для балансировки нагрузки, такие как минимизация числа запросов, последовательное хеширование и учет зоны.

Коммуникационная устойчивость

Service mesh может повысить коммуникационную устойчивость приложений благодаря отсутствию необходимости реализовывать в коде приложения процедуры входа, истечения времени ожидания, автоматического выключения или ограничения скорости передачи.

Безопасность

Service mesh предоставляет сквозное шифрование с mTLS между сервисами и политики авторизации, которые определяют, какие сервисы имеют право взаимодействовать с другими — и не только на уровнях 3 и 4, как это происходит в сетевых политиках Kubernetes.

Мониторинг

Service mesh усиливают возможности мониторинга путем добавления диагностик на уровне 7, а также трассирования и предупреждений.

Управление маршрутизацией

Сдвиг и отзеркаливание трафика в кластере.

API

Все вышеуказанное контролируется с помощью API, предоставляемого реализацией Service mesh.

Рассмотрим компоненты Service mesh, показанные на рис. 5.8.

Трафик обрабатывается различными способами — в зависимости от компонента или назначения трафика. Входной и выходной трафик кластера управляется шлю-

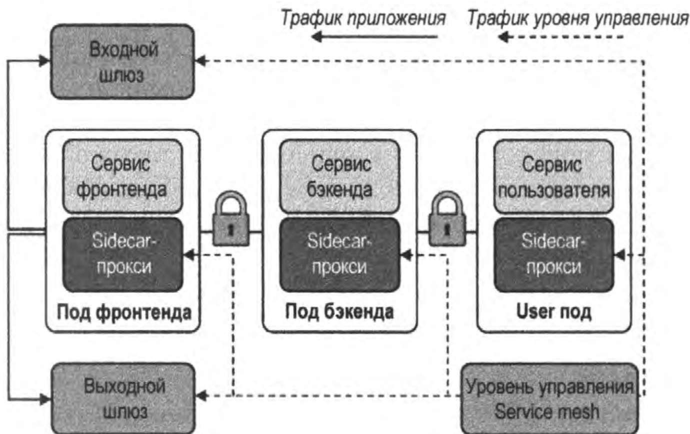


Рис. 5.8. Компоненты технологии Service mesh

зами. Трафик между фронтендом, бэкендом и пользовательским сервисом шифруется с помощью mTLS и обрабатывается service mesh. Весь трафик на поды фронтенда, бэкенда и пользователя в service mesh проксируется sidecar-прокси, развернутым внутри подов. Даже если уровень управления не функционирует и обновления service mesh не выполняются, трафик сервисов и приложения это не затрагивает.

Существует довольно много систем, с помощью которых можно развернуть service mesh; ниже мы перечислим только несколько из них:

- *Istio*
  - Использует уровень управления на Go и Envoy прокси.
  - Представляет собой «родное» решение Kubernetes, изначально выпущенное Lyft.
- *Consul*
  - Уровень управления на основе HashiCorp Consul.
  - ConsulConnect использует агента, устанавливаемого на каждом узле, в качестве DaemonSet, который взаимодействует с Envoy sidecar-прокси для маршрутизации и переадресации трафика.
- *AWS App Mesh*
  - Является решением под управлением AWS, имеет собственный управляющий уровень
  - Не имеет mTLS или сетевой политики.
  - Для уровня данных использует Envoy прокси.
- *Linkerd*
  - Тоже использует Go для уровня управления наряду с Linkerd прокси.
  - Отсутствует сдвиг трафика и распределенное трассирование.
  - Является решением только для Kubernetes, что отражается в меньшем количестве мобильных компонентов и означает, что Linkerd везде имеет меньшую сложность.

По нашему мнению, наиболее оптимально использовать service mesh с применением mTLS между сервисами. Другие варианты высокоуровневого использования включают в себя автоматическое выключение (circuit breaking) и тестирование на сбои для API. Также на базе service mesh сетевые администраторы могут реализовать более эффективные алгоритмы и политики маршрутизации.

Рассмотрим пример на использование service mesh. Первый шаг, который надо выполнить, если вы еще этого не сделали, — инсталлировать Linkerd CLI.

На выбор предоставляются cURL, bash или brew, если вы работаете на Mac:

```
curl -sL https://run.linkerd.io/install | sh
```

либо

```
brew install linkerd

linkerd version
Client version: stable-2.9.2
Server version: unavailable
```

Проведенная проверка компонентов позволяет убедиться в том, что кластер готов к запуску Linkerd:

```
linkerd check --pre
kubernetes-api
--------------
√ can initialize the client
√ can query the Kubernetes API

kubernetes-version
------------------
√ is running the minimum Kubernetes API version
√ is running the minimum kubectl version

pre-kubernetes-setup
--------------------
√ control plane namespace does not already exist
√ can create non-namespaced resources
√ can create ServiceAccounts
√ can create Services
√ can create Deployments
√ can create CronJobs
√ can create ConfigMaps
√ can create Secrets
√ can read Secrets
√ can read extension-apiserver-authentication configmap
√ no clock skew detected

pre-kubernetes-capability
-------------------------
√ has NET_ADMIN capability
√ has NET_RAW capability

linkerd-version

√ can determine the latest version
√ cli is up-to-date

Status check results are √
```

С помощью интерфейса Linkerd CLI установим Linkerd в нашем KIND-кластере:

```
linkerd install | kubectl apply -f -
namespace/linkerd created
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-identity created
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-identity created
serviceaccount/linkerd-identity created
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-controller created
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-controller created
serviceaccount/linkerd-controller created
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-destination created
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-destination created
serviceaccount/linkerd-destination created
role.rbac.authorization.k8s.io/linkerd-heartbeat created
rolebinding.rbac.authorization.k8s.io/linkerd-heartbeat created
serviceaccount/linkerd-heartbeat created
role.rbac.authorization.k8s.io/linkerd-web created
rolebinding.rbac.authorization.k8s.io/linkerd-web created
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-web-check created
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-web-check created
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-web-admin created
serviceaccount/linkerd-web created
customresourcedefinition.apiextensions.k8s.io/serviceprofiles.linkerd.io created
customresourcedefinition.apiextensions.k8s.io/trafficsplits.split.smi-spec.io
created
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-proxy-injector created
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-proxy-injector
created
serviceaccount/linkerd-proxy-injector created
secret/linkerd-proxy-injector-k8s-tls created
mutatingwebhookconfiguration.admissionregistration.k8s.io
/linkerd-proxy-injector-webhook-config created
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-sp-validator created
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-sp-validator
created
serviceaccount/linkerd-sp-validator created
secret/linkerd-sp-validator-k8s-tls created
validatingwebhookconfiguration.admissionregistration.k8s.io
/linkerd-sp-validator-webhook-config created
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-tap created
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-tap-admin created
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-tap created
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-tap-auth-delegator
created
serviceaccount/linkerd-tap created
rolebinding.rbac.authorization.k8s.io/linkerd-linkerd-tap-auth-reader created
secret/linkerd-tap-k8s-tls created
apiservice.apiregistration.k8s.io/v1alpha1.tap.linkerd.io created
```

```
podsecuritypolicy.policy/linkerd-linkerd-control-plane created
role.rbac.authorization.k8s.io/linkerd-psp created
rolebinding.rbac.authorization.k8s.io/linkerd-psp created
configmap/linkerd-config created
secret/linkerd-identity-issuer created
service/linkerd-identity created
service/linkerd-identity-headless created
deployment.apps/linkerd-identity created
service/linkerd-controller-api created
deployment.apps/linkerd-controller created
service/linkerd-dst created
service/linkerd-dst-headless created
deployment.apps/linkerd-destination created
cronjob.batch/linkerd-heartbeat created
service/linkerd-web created
deployment.apps/linkerd-web created
deployment.apps/linkerd-proxy-injector created
service/linkerd-proxy-injector created
service/linkerd-sp-validator created
deployment.apps/linkerd-sp-validator created
service/linkerd-tap created
deployment.apps/linkerd-tap created
serviceaccount/linkerd-grafana created
configmap/linkerd-grafana-config created
service/linkerd-grafana created
deployment.apps/linkerd-grafana created
clusterrole.rbac.authorization.k8s.io/linkerd-linkerd-prometheus created
clusterrolebinding.rbac.authorization.k8s.io/linkerd-linkerd-prometheus created
serviceaccount/linkerd-prometheus created
configmap/linkerd-prometheus-config created
service/linkerd-prometheus created
deployment.apps/linkerd-prometheus created
secret/linkerd-config-overrides created
```

Видим, что в кластере происходит установка большого количества компонент, раньше мы наблюдали похожую ситуацию при инсталляции ингресс-контроллера и MetalLB.

Установку Linkerd можно проверить с помощью команды `linkerd check`. В результате будут протестированы/верифицированы многочисленные компоненты Linkerd: версия Kubernetes API, контроллеры, поды, конфигурационные файлы, а также все сервисы, версии и API, необходимые для работы Linkerd:

```
linkerd check
kubernetes-api
--------------
√ can initialize the client
√ can query the Kubernetes API
```

```
kubernetes-version
------------------
√ is running the minimum Kubernetes API version
√ is running the minimum kubectl version

linkerd-existence
-----------------
√ 'linkerd-config' config map exists
√ heartbeat ServiceAccount exists
√ control plane replica sets are ready
√ no unschedulable pods
√ controller pod is running
√ can initialize the client
√ can query the control plane API

linkerd-config
--------------
√ control plane Namespace exists
√ control plane ClusterRoles exist
√ control plane ClusterRoleBindings exist
√ control plane ServiceAccounts exist
√ control plane CustomResourceDefinitions exist
√ control plane MutatingWebhookConfigurations exist
√ control plane ValidatingWebhookConfigurations exist
√ control plane PodSecurityPolicies exist

linkerd-identity
----------------
√ certificate config is valid
√ trust anchors are using supported crypto algorithm
√ trust anchors are within their validity period
√ trust anchors are valid for at least 60 days
√ issuer cert is using supported crypto algorithm
√ issuer cert is within its validity period
√ issuer cert is valid for at least 60 days
√ issuer cert is issued by the trust anchor

linkerd-webhooks-and-apisvc-tls
-------------------------------
√ tap API server has valid cert
√ tap API server cert is valid for at least 60 days
√ proxy-injector webhook has valid cert
√ proxy-injector cert is valid for at least 60 days
√ sp-validator webhook has valid cert
√ sp-validator cert is valid for at least 60 days
```

```
linkerd-api
-----------
√ control plane pods are ready
√ control plane self-check
√ [kubernetes] control plane can talk to Kubernetes
√ [prometheus] control plane can talk to Prometheus
√ tap api service is running

linkerd-version
---------------
√ can determine the latest version
√ cli is up-to-date

control-plane-version
---------------------
√ control plane is up-to-date
√ control plane and cli versions match

linkerd-prometheus
------------------
√ prometheus add-on service account exists
√ prometheus add-on config map exists
√ prometheus pod is running

linkerd-grafana
---------------
√ grafana add-on service account exists
√ grafana add-on config map exists
√ grafana pod is running

Status check results are √
```

Теперь, убедившись, что установка Linkerd прошла успешно, мы может добавить наше приложение к service mesh:

```
kubectl -n linkerd get deploy

NAME                     READY UP-TO-DATE   AVAILABLE      AGE
linkerd-controller       1/1    1            1             3m17s
linkerd-destination      1/1    1            1             3m17s
linkerd-grafana          1/1    1            1             3m16s
linkerd-identity         1/1    1            1             3m17s
linkerd-prometheus       1/1    1            1             3m16s
linkerd-proxy-injector   1/1    1            1             3m17s
linkerd-sp-validator     1/1    1            1             3m17s
linkerd-tap              1/1    1            1             3m17s
linkerd-web              1/1    1            1             3m17s
```

Перейдем на консоль Linkerd, чтобы изучить только что развернутую систему. Консоль запускается командой `linkerd dashboard &`.

Консоль будет доступна на нашей локальной машине по адресу http://localhost:5075:

```
linkerd viz install | kubectl apply -f -
linkerd viz dashboard
Linkerd dashboard available at:
http://localhost:50750
Grafana dashboard available at:
http://localhost:50750/grafana
Opening Linkerd dashboard in the default browser
```

Если вы столкнулись с проблемами при доступе к консоли, то можете запустить inkerd viz check и найти помощь в документации к Linkerd

На рис. 5.9 представлены все объекты, которые мы разворачивали в предыдущих примерах.

Наш сервис ClusterIP не является частью сети сервисов Linkerd. Чтобы добавить его к сети, придется обратиться к прокси-инжектору. Он выполнит задачу, обращаясь к соответствующей аннотации, которая может быть добавлена либо командой Linkerd inject, либо вручную в спецификации пода.

Рис. 5.9. Консоль Linkerd

Для большей наглядности уберем некоторые ресурсы, оставшиеся от старых примеров:

```
kubectl delete -f ingress-example-2.yaml
deployment.apps "app2" deleted
service "clusterip-service-2" deleted
ingress.extensions "ingress-resource-2" deleted

kubectl delete pods app-5586fc9d77-7frts
pod "app-5586fc9d77-7frts" deleted

kubectl delete -f ingress-rule.yaml
ingress.extensions "ingress-resource" deleted
```

Для добавления аннотаций, необходимых, чтобы сервис стал частью сети сервисов, воспользуемся Linkerd CLI.

Сначала нам надо получить манифест приложения, `cat web.yaml`, и использовать Linkerd, чтобы вставить аннотации, `linkerd inject -` , а затем применить их в Kubernetes API командой `kubectl apply -f -`:

```
cat web.yaml | linkerd inject - | kubectl apply -f -

deployment "app" injected

deployment.apps/app configured
```

Если теперь вызовем `describe` для нашего развернутого приложения, то увидим, что Linkerd вставил новые аннотации, `Annotations: linkerd.io/object: enabled`:

```
kubectl describe deployment app
Name:                   app
Namespace:              default
CreationTimestamp:      Sat, 30 Jan 2021 13:48:47 -0500
Labels:                 <none>
Annotations:            deployment.kubernetes.io/revision: 3
Selector:               app=app
Replicas:               1 desired | 1 updated | 1 total | 1 available |
0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
   Labels:          app=app
   Annotations:     linkerd.io/inject: enabled
   Containers:
    go-web:
     Image:      strongjz/go-web:v0.0.6
     Port:       8080/TCP
     Host Port: 0/TCP
```

```
    Liveness: http-get http://:8080/healthz delay=5s timeout=1s period=5s
    Readiness: http-get http://:8080/ delay=5s timeout=1s period=5s
    Environment:
      MY_NODE_NAME:               v1:spec.nodeName)
      MY_POD_NAME:                (v1:metadata.name)
      MY_POD_NAMESPACE:           (v1:metadata.namespace)
      MY_POD_IP:                  (v1:status.podIP)
      MY_POD_SERVICE_ACCOUNT:     (v1:spec.serviceAccountName)
      DB_HOST:                    postgres
      DB_USER:                    postgres
      DB_PASSWORD:                mysecretpassword
      DB_PORT:                    5432
    Mounts:                       <none>
  Volumes:                        <none>
Conditions:
Type             Status     Reason
----             ------     ------
Available        True       MinimumReplicasAvailable
Progressing      True       NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   app-78dfbb4854 (1/1 replicas created)
Events:
Type        Reason       Age    From                   Message
----        ------       ----   ----                   -------
Normal ScalingReplicaSet 4m4s deployment-controller  Scaled down app-5586fc9d77
Normal ScalingReplicaSet 4m4s deployment-controller  Scaled up app-78dfbb4854
Normal Injected          4m4s linkerd-proxy-injector  Linkerd sidecar injected
Normal ScalingReplicaSet 3m54s deployment-controller  Scaled app-5586fc9d77
```

Если перейдем на наше приложении на консоли, то увидим, что наше развертывание теперь является частью сети сервисов Linkerd (рис. 5.10).

Также интерфейс CLI показывает нам некоторые характеристики, относящиеся к нашему развертыванию:

```
linkerd stat deployments -n default
NAME MESHED SUCCESS    RPS LATENCY_P50 LATENCY_P95 LATENCY_P99 TCP_CONN
App     1/1 100.00% 0.4rps         1ms         1ms         1ms     1
```

Снова увеличим число реплик в нашем развертывании:

```
kubectl scale deploy app --replicas 10
deployment.apps/app scaled
```

На рис. 5.11 мы переходим в веб-браузер и открываем показанную ссылку, тем самым мы можем следить за характеристиками работы в реальном времени. Выберите параметр namespaces (пространство имен) по умолчанию, а в ресурсах (Resources) выберите наше приложение deployment/app. Затем нажмите start for the web (начать работу в Интернете), чтобы запустить выдачу характеристик.

Рис. 5.10. Консоль Linkerd с развернутым веб-приложением

Рис. 5.11. Веб-приложение на консоли Linkerd

На отдельном терминале давайте воспользуемся образом netshoot, но на этот раз запущенным внутри нашего KIND-кластера:

```
kubectl run tmp-shell --rm -i --tty --image nicolaka/netshoot -- /bin/bash
If you don't see a command prompt, try pressing enter.
bash-5.0#
```

Отправим несколько сотен запросов и посмотрим статистику:

```
bash-5.0#for i in `seq 1 100`;
do curl http://clusterip-service/host && sleep 2;
done
```

На терминале мы можем видеть результаты всех проверок на готовность и активность, а также наши запросы на /host.

tmp-shell — это наш bash-терминал для запущенного образа netshoot.

10.244.2.1 , 10.244.3.1 и 10.244.2.1 — это адреса Kubelet на хостах, выполняющих проверки:

```
linkerd viz stat deploy

NAME    MESHED    SUCCESS   RPS     LATENCY_P50  LATENCY_P95  LATENCY_P99 TCP_CONN
App   1/1         100.00%  0.7rps   1ms          1ms              1ms        3
```

Это только один пример, показывающий, как можно наблюдать за работой сети сервисов. Linkerd, Istio и другие системы предоставляют большое количество опций, позволяющих разработчикам и сетевым администраторам управлять, контролировать и устранять ошибки в сервисах, запущенных внутри кластерной сети. Выбор этих опций осуществляется в зависимости от того, какая функциональность и какие характеристики являются важными для ваших сетей.

Заключение

Сети Kubernetes предоставляют разработчикам большое количество опций для развертывания, тестирования и управления работой приложений в кластерах. Каждая новая опция добавляет сложности и приводит к задержкам при выполнении операций. Мы рассказали разработчикам, сетевым и системным администраторам об абстракциях, которые предлагает им Kubernetes.

Эти специалисты должны решать, какая из абстракций будем наиболее эффективной для функционирования каждой данной системы. Это большое дело, и теперь вы достаточно вооружены знаниями, чтобы начать его обсуждение.

В следующей главе мы перенесем наши сервисы Kubernetes в облако. Мы рассмотрим сетевые сервисы, предлагаемые различными облачными провайдерами, и как их можно интегрировать в сервисы под управлением Kubernetes.

ГЛАВА 6

Kubernetes и облачные сети

Использование облаков и предлагаемых ими сервисов показывает гигантский рост: 77% предприятий используют публичные облака в той или иной степени, а 81% могут быстрее внедрять инновации, основываясь на публичном облаке, а не на облаке предприятия. Такая популярность облаков и доступность в них технологического обновления обусловливают применение Kubernetes в облачных сетях. Каждый крупный облачный провайдер предлагает свой сервис для Kubernetes, основываясь на имеющихся в его распоряжении сетевых возможностях.

В данной главе мы рассмотрим сетевые сервисы, предлагаемые основными облачными провайдерами AWS, Azure и GCP, обращая особое внимание на процедуры, необходимые для запуска кластера Kubernetes внутри конкретного облака. Все провайдеры также поддерживают проект CNI, который благодаря интеграции с облачными API позволяет обеспечить плавную работу кластеров Kubernetes. Поэтому вполне логично, что мы подробно изучим эти CNI. После прочтения главы администраторы будут лучше понимать, каким образом облачные провайдеры реализуют функции Kubernetes поверх своих облачных сетевых сервисов.

Amazon Web Services

Облачные сервисы Amazon Web Services (AWS) выросли из Simple Queue Service (SQS) и Simple Storage Service (s3) и в настоящее время объединяют в себе свыше 200 сервисов. В 2020 г. фирма Gartner Research поместила AWS в лидирующем сегменте своего списка облачных и платформенных сервисов. Многие сервисы формируются поверх других базовых сервисов. Так, Lambda использует S3 для хранения программ, а DynamoDB — для метаданных. AWS CodeCommit также использует S3 для хранения программ. EC2, S3 и CloudWatch интегрированы в сервис Amazon Elastic MapReduce, создавая платформу управляемых данных. Аналогично работают и сетевые сервисы AWS. Современные сервисы типа пиринга и конечных точек реализованы на базе ядерных структурных блоков. Хорошее понимание работы этих блоков, которые позволяют AWS обеспечивать развернутую поддержку Kubernetes, необходимо для администраторов и разработчиков.

Сетевые сервисы AWS

AWS предоставляет множество сервисов, позволяющих пользователю расширить свои облачные сети и сделать их безопасными. Amazon Elastic Kubernetes Service

(EKS) широко использует сетевые компоненты, доступные в AWS облаке. Мы рассмотрим основные сетевые компоненты AWS и их использование при развертывании кластерной сети EKS. Также в данном разделе мы обсудим несколько инструментов из проектов с открытым кодом, которые помогают в управлении кластером и развертывании приложений. В первую очередь это `eksctl` — CLI- инструмент для развертывания и управления кластерами EKS. Как мы уже видели в предыдущих главах, для запуска кластера требуется большое количество компонентов, и это в полной мере относится к сети AWS. Задачей `eksctl` как раз и является установка всех компонент AWS для работы кластера и сети. Далее мы изучим CNI для AWS VPC, который дает возможность кластеру использовать базовые сервисы AWS, чтобы масштабировать поды и управлять пространством их IP-адресов. Наконец, мы проработаем ингресс-контроллер AWS Application Load Balancer, облегчающий разработчикам сетевых приложений AWS развертывание балансировщиков нагрузки и ингрессов.

Виртуальное частное облако

Основой сети AWS является *виртуальное частное облако VPC*. Большинство ресурсов AWS локализуется внутри VPC. VPC — это защищенная виртуальная сеть, определяемая администраторами только для их адреса и пользующаяся только своими ресурсами. На рис. 6.1 показана VPC, заданная единственным CIDR 192.168.0.0/16. Все ресурсы внутри VPC будут использовать данный диапазон для своих частных IP-адресов. AWS постоянно расширяет свою линейку сервисов, в настоящее время сетевые администраторы могут использовать множественные неперекрывающиеся CIDR в одном облаке VPC. IP-адреса подов тоже назначаются из диапазона CIDR облака VPC и IP-адресов хоста, более подробно это будет рассмотрено в разделе «CNI для облака AWS VPC» ниже. VPC устанавливается на регион AWS, вы можете иметь в регионе несколько VPC, они все будут привязаны только к данному региону.

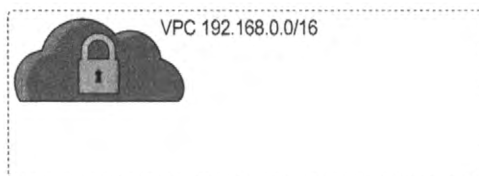


Рис. 6.1. Виртуальное частное облако AWS VPC

Регионы и зоны доступности

Ресурсы в AWS определяются по *границам доступности* — есть глобальные, региональные или доступные только в зоне. Сеть AWS включает в себя несколько регионов, каждый регион состоит из нескольких изолированных и физически разделенных зон доступности внутри географической области. Зона доступности может содержать несколько центров обработки данных — см. рис. 6.2. Некоторые

регионы могут содержать 6 зон доступности, а более новые регионы могут иметь только две. Каждая зона напрямую связана с другими, но защищена от сбоев, происходящих в других секторах. Важно хорошо понимать данную архитектуру, поскольку именно она определяет высокую доступность, балансировку нагрузки и функционирование подсетей. Балансировщик нагрузки в регионе направляет трафик через многие зоны, имеющие отдельные подсети, обеспечивая тем самым высокую доступность для приложений.

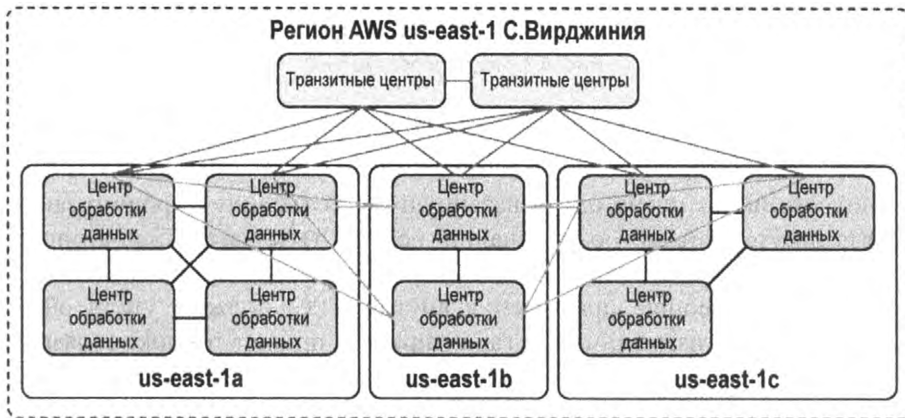
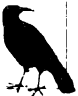


Рис. 6.2. Топология сети в регионе AWS



Актуальный на сегодня список регионов AWS и зон доступности имеется в документации.

Подсеть

В облаке VPC могут существовать несколько подсетей с идентификаторами из заданного диапазона CIDR, развернутых в одной зоне доступности. Приложения, требующие высокой доступности, должны быть запущены в нескольких зонах, а распределение нагрузки должно производиться любым из имеющихся балансировщиков.

Подсеть является публичной, если таблица маршрутизации содержит путь к шлюзу Интернета. На рис. 6.3. показаны три публичные и три частные подсети. Частные подсети не имеют прямого выхода в Интернет. Эти подсети предназначены для внутреннего сетевого трафика, например баз данных. Диапазон CIDR вашего облака VPC и число частных и публичных подсетей определяются установками при развертывании вашей сетевой архитектуры. Последние улучшения в структуре VPC, например разрешение иметь несколько диапазонов CIDR, помогают сгладить недостатки неудачных начальных установок, поскольку теперь сетевые инженеры просто могут добавить новый диапазон CIDR к имеющемуся в их распоряжении облаку VPC.

Рис. 6.3. Подсети облака VPC

Таблицы маршрутизации

Каждая подсеть имеет ровно одну связанную с ней таблицу маршрутизации. Если явная связь отсутствует, то по умолчанию берется головная таблица маршрутизации. Здесь могут проявиться проблемы, обусловленные связностью сетей: разработчики, разворачивающие приложения внутри VPC, должны уметь обращаться с таблицами маршрутизации, чтобы гарантировать приход трафика в назначенную точку.

Ниже приводятся правила для головной таблицы маршрутизации:

- Головная таблица маршрутизации не может быть удалена.
- Таблица маршрутизации шлюза не может быть установлена как головная.
- Головная таблица маршрутизации может быть заменена пользовательской таблицей маршрутизации.
- Администраторы могут добавлять, удалять и изменять маршруты в головной таблице маршрутизации.
- Локальный маршрут является наиболее предпочтительным.
- Подсети могут быть явно связаны с головной таблицей маршрутизации.

Существуют специализированные таблицы маршрутизации; ниже приводится их список и описание их характеристик:

Головная таблица маршрутизации

Эта таблица автоматически управляет маршрутизацией для всех подсетей, которые явно не связаны с любой другой таблицей.

Пользовательская таблица маршрутизации

Таблица, создаваемая сетевыми инженерами и предназначенная для потока трафика конкретного приложения.

Edge Association

Таблица маршрутизации, направляющая входящий трафик облака VPC на пограничные устройства (edge appliance).

Таблица маршрутизации для подсети

Таблица маршрутизации, соответствующая подсети.

Таблица маршрутизации шлюза

Таблица маршрутизации, связанная со шлюзом Интернета или шлюзом виртуальной частной сети.

Каждая таблица маршрутизации имеет несколько компонентов, определяющих ее функционирование:

Соответствие таблицы маршрутизации

Соответствие между таблицей маршрутизации и шлюзом подсети, Интернета или виртуальной частной сети.

Правила

Список задающих таблицу маршрутов; каждое правило содержит назначение, шлюз/интерфейс, статус и флаг распространения маршрута.

Назначение

Диапазон IP-адресов, куда направляется трафик (CIDR назначения).

Шлюз

Шлюз, сетевой интерфейс или соединение, через которое посылается назначенный трафик, например шлюз Интернета.

Статус

Состояние маршрута в таблице: активный или черная дыра. Состояние «черная дыра» указывает, что шлюз маршрута недоступен.

Флаг распространения маршрута

Функция распространения маршрута позволяет шлюзу виртуальной частной сети автоматически распространять маршруты на таблицу маршрутизации. Установленный флаг означает, что маршрут был добавлен через распространение.

Локальный маршрут

Маршрут по умолчанию для коммуникации внутри облака VPC.

На рис. 6.4 показаны два маршрута из таблицы маршрутизации. Любой трафик, адресованный на `11.0.0.0/16`, остается в локальной сети внутри VPC. Любой другой

Edit

View: All rules

Destination	Target	Status	Propagated
11.0.0.0/16	local	Active	No
0.0.0.0/0	igw-f43c4690	Active	No

Рис. 6.4. Таблица маршрутизации

трафик, `0.0.0.0/0`, идет на шлюз Интернета, igw-f43c4690, делая его публичной подсетью.

Эластичный сетевой интерфейс

Эластичный сетевой интерфейс (ENI) — это логический сетевой компонент облака VPC, являющийся экземпляром виртуальной сетевой карты. Интерфейсы ENI содержат IP-адрес, один на экземпляр, и их эластичность проявляется в том, что они могут быть поставлены в соответствие или снова разъединены с данным экземпляром, полностью сохраняя свои свойства.

Свойствами ENI являются:

- Частный IPv4-адрес первого уровня.
- Частные IPv4-адреса второго уровня.
- Один эластичный (EIP) адрес на один частный IPv4-адрес.
- Один публичный IPv4-адрес, который может автоматически ставиться в соответствие с сетевым интерфесом `eth0` при запуске экземпляра сетевой карты.
- Один или несколько IPv6-адресов.
- MAC-адрес.
- Флаг проверки источника/назначения.
- Описание.

Обычно ENI используется для создания управляющих сетей, которые доступны только из сети предприятия. Сервисы AWS, подобные Amazon WorkSpace, используют эластичные интерфейсы, чтобы разрешить доступ к пользовательскому облаку VPC и облакам под управлением AWS. С помощью Lambda можно получить доступ к ресурсам внутри VPC, например базам данных, путем предоставления эластичного интерфейса ENI или связывания с ним.

Ниже в данном разделе мы рассмотрим, каким образом CNI для облаков AWS VPC используют и управляют ENI наряду с IP-адресами подов.

Эластичный IP-адрес

Эластичный IP-адрес (EIP) — это статичный публичный IPv4-адрес, используемый для динамической адресации в облаке AWS. EIP связывается с виртуальным или обычным сетевым интерфейсом в любом облаке VPC. Используя EIP, разработчик приложений может купировать сбой интерфейса путем переназначения адреса на другой виртуальный интерфейс.

EIP-адрес принадлежит эластичному интерфейсу ENI и связывается с виртуальной картой путем обновления соответствующего ENI. Преимущество связывания EIP с интерфейсом ENI, а не с самой картой заключается в том, что все атрибуты сетевого интерфейса переносятся с одной карты на другую за одну операцию.

Действуют следующие правила:

- В каждый данный момент времени EIP-адрес может быть связан либо с одной виртуальной картой, либо с сетевым интерфейсом.

- EIP-адрес может мигрировать с одной виртуальной карты или сетевого интерфейса на другую (другой).
- Имеется (нежесткий) лимит на 5 EIP-адресов.
- Система IPv6 не поддерживается.

Сервисы, подобные NAT, и шлюз Интернета используют EIP, чтобы обеспечить согласованность между зонами доступности. Другие шлюзовые службы типа bastion также получают преимущества от применения EIP. Подсети могут автоматически присваивать публичные IP-адреса экземплярам EC2, но такой адрес может измениться, в то время как использование EIP позволяет это предотвратить.

Средства обеспечения безопасности

Базовые средства обеспечения безопасности в сетях AWS делятся на две категории: группы безопасности и списки контроля доступа к сети (NACL). Как показывает наш опыт, неправильное конфигурирование групп безопасности и списков NACL может привести к большим проблемам. Разработчики и сетевые инженеры должны хорошо понимать различия между указанными двумя категориями и эффектами, к которым может привести внесение изменений в соответствующие компоненты.

Группы безопасности

Группы безопасности действуют на уровне виртуального сервера (instance) или сетевого интерфейса и функционируют как брандмауэр для связанных с ними устройств. Группа безопасности — это группа сетевых устройств, требующих общего доступа друг к другу и к другим устройствам сети. Из рис. 6.5 видно, что группа безопасности работает в нескольких зонах доступности. Группы безопасности имеют две таблицы — для входящего трафика и для выходящего трафика. Группы безопасности действуют в режиме сохранения состояния (stateful), так что если трафик разрешен для входного потока, то он разрешен и для выходного. Каждая группа безопасности имеет список правил, определяющих фильтры для трафика. Прежде чем принимается решение на переадресацию трафика, происходит проверка на выполнение всех установленных правил.

Рис. 6.5. Группы безопасности

Ниже приводится список компонентов для правил групп безопасности:

Источник/назначение

Источник (правила входа) или назначение (правила выхода) рассматриваемого трафика:

- Индивидуальные адреса или диапазон IPv4/IPv6.
- Другая группа безопасности.
- Другие ENI, шлюзы или интерфейсы.

Протокол

Какой протокол 4-го уровня отфильтровывается: 6 (TCP), 17 (UDP), 1(ICMP).

Диапазон портов

Задаются порты для отфильтрованного протокола.

Описание

Задаваемое пользователем поле для информации о целях данной группы.

Группы безопасности по функциям похожи на сетевые политики Kubernetes, которые мы рассматривали в предыдущих главах. Они являются базовыми компонентами сетевых технологий и всегда должны использоваться для обеспечения безопасности ваших приложений в облаке VPC AWS. Система EKS предоставляет несколько групп безопасности для взаимодействия между уровнем управления данными AWS и вашими рабочими узлами.

Списки контроля доступа к сети (NACL). Принципы работы данных списков аналогичны используемым на других брандмауэрах, так что сетевым инженерам они хорошо знакомы. Из рис. 6.6 видно, что каждая подсеть имеет свой список по умолчанию, связанный с ней и ограниченный одной зоной доступности, в отличие от группы безопасности. Правила фильтрования должны быть явно заданы для обоих направлений трафика. Правила по умолчанию весьма мягкие и разрешают любой трафик в любом направлении. Поэтому если пользователи считают, что группа безопасности «слишком открытая», то они могут определить свои собственные списки контроля доступа применительно к подсети. По умолчанию пользовательские списки NACL закрывают любой трафик, и поэтому при развертывании

Рис. 6.6. Список контроля доступа к сети (NACL)

приложений к ним необходимо добавлять правила — в противном случае приложения потеряют доступ к сетям.

Ниже приводится список компонентов для NACL:

Номер правила

Правила выполняются, начиная с правила с наименьшим номером.

Тип

Тип трафика, например SSH или HTTP.

Протокол

Любой протокол, имеющий стандартный номер: TCP/UDP или ALL.

Диапазон портов

Слушающий порт или диапазон портов для трафика, например 80 для HTTP-трафика.

Источник

Правила только для входного трафика, диапазон CIDR-источника трафика.

Назначение

Правила только для выходящего трафика, адрес назначения трафика.

Разрешить/Запретить

Разрешить или запретить указанный трафик.

Списки NACL предоставляют дополнительную безопасность для подсетей, что может помочь в их защите в случае, если группы безопасности сконфигурированы неправильно.

Табл. 6.1 суммирует основные различия между группами безопасности и списками NACL.

Таблица 6.1. *Сравнение групп безопасности и списков контроля доступа к сети (NACL)*

Группа безопасности	Список NACL
Работает на уровне виртуального сервера	Работает на уровне подсети
Поддерживает только разрешительные правила	Поддерживает разрешительные и запретительные правила
Режим Stateful: возвратный трафик автоматически разрешается вне зависимости от любых правил	Режим Stateless: возвратный трафик должен быть явно разрешен правилами
Перед перенаправлением трафика проверяется выполнение всех правил	Правила обрабатываются по порядку, начиная с правила с наименьшим номером
Правила применяются к одной виртуальной карте или сетевому интерфейсу	В интерфейсах правила действуют для всех подсетей, с которыми он ассоциирован

Очень важно понимать различия между списками NACL и группами безопасности. Проблемы с сетевой доступностью часто возникают из-за того, что группа безопасности не разрешает трафик на конкретный порт или кто-то не добавил правило для

выходящего трафика в список NACL. Так что при поиске причин сбоев в сетях AWS разработчики и сетевые инженеры обязательно должны проверять данные компоненты.

Все рассмотренные компоненты так или иначе управляют потоком трафика внутри облака VPC. Службы, которые мы обсудим в следующем разделе, управляют входящим трафиком VPC, т. е. трафиком, имеющим в качестве источника клиентские запросы и направляемым на приложения, запущенные внутри кластера Kubernetes: это устройства преобразования сетевых адресов, шлюз Интернета и балансировщики нагрузки. Перейдем к более подробному их описанию.

Устройства преобразования сетевых адресов

Устройства преобразования сетевых адресов (NAT) применяются в случаях, когда виртуальные серверы внутри VPC требуют связи с Интернетом, однако сетевое взаимодействие не должно осуществляться непосредственно с серверами. Примерами таких систем являются экземпляры баз данных или любое связующее ПО, необходимое для запуска приложений.

При использовании AWS сетевые инженеры получают несколько вариантов работы с NAT-устройствами. Они могут управлять своими собственными NAT-устройствами, развернутыми как экземпляры EC2, или прибегнуть к управляемому сервису AWS Managed Service для шлюза NAT (NAT GW). Оба подхода требуют наличия публичных подсетей, действующих в нескольких зонах доступности, чтобы обеспечить высокую доступность и эластичные адреса EIP. Ограничением при работе с NAT GW является невозможность изменить IP-адрес после развертывания шлюза. Таким образом, этот IP-адрес будет адресом источника, используемым для взаимодействия со шлюзом Интернета.

Рис. 6.7 показывает, как с помощью двух таблиц маршрутизации осуществляется соединение с Интернетом. Главная таблица содержит два правила: локальный маршрут для взаимодействия внутри VPC и маршрут для 0.0.0.0/0 с назначением на ID шлюза NAT GW. Серверы баз данных частной подсети будут направлять трафик в Интернет, используя правило для NAT GW в своих таблицах маршрутизации.

Поды и виртуальные серверы в системе EKS должны будут работать с трафиком, выходящим из VPC, так что придется развернуть NAT-устройство. Выбор этого устройства зависит от операционной нагрузки, стоимостных показателей или требований по доступности, предъявляемых конструкцией вашей сети.

Шлюз Интернета

Шлюз Интернета является управляемым AWS сервисом и сетевым устройством в облаке VPC, которое обеспечивает связь с Интернетом для всех устройств облака. Ниже приводится последовательность шагов для обеспечения доступа в Интернет из VPC:

1. Развернуть и связать шлюз Интернета с VPC.
2. Прописать маршрут в таблице маршрутизации подсети, который будет адресовать направляемый в Интернет трафик на шлюз Интернета.

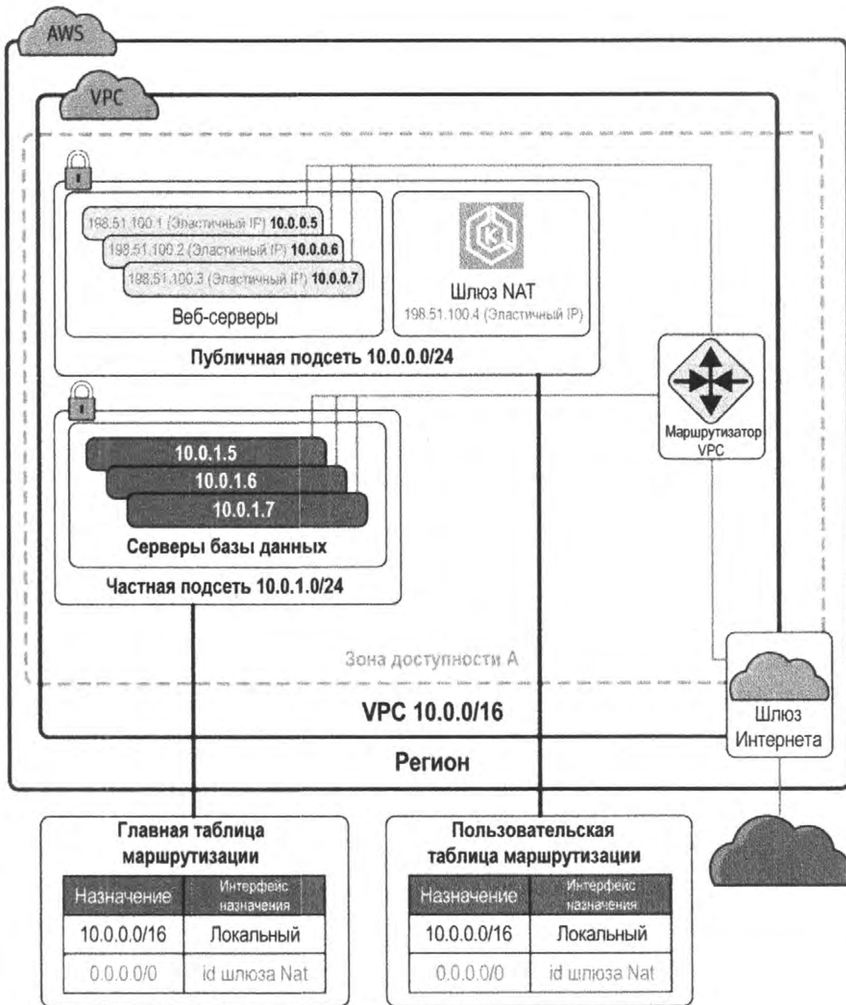


Рис. 6.7. Диаграмма маршрутизации для виртуального облака VPC

3. Проверить правила в списках NACL и группах безопасности, чтобы обеспечить поток трафика к виртуальному серверу и из него.

Данная технология показана на рис. 6.7. Мы видим развернутый в VPC шлюз Интернета и установки в пользовательской таблице маршрутизации, направляющие весь трафик, `0.0.0.0/0`, на этот шлюз. Веб-серверы имеют интернет-маршрутизируемые адреса системы IPv4, `198.51.100.1-3`.

Эластичные балансировщики нагрузки

Теперь, когда трафик идет из Интернета и клиенты могут запрашивать приложения, запущенные внутри облака VPC, нам потребуется управлять нагрузкой и распределять ее. AWS предлагает разработчикам несколько опций в зависимости от типа приложения и требований к сетевому трафику.

Эластичный балансировщик нагрузки доступен в четырех вариантах:

Классический

Классический балансировщик нагрузки обеспечивает базовую балансировку для EC2-серверов. Действует на уровнях запроса и соединения. Данные балансировщики имеют ограниченную функциональность и не должны использоваться с контейнерами.

Приложение

Балансировщики нагрузки для приложений работают на уровне 7. Маршрутизация выполняется на основе данных запроса, таких как HTTP-заголовок или HTTP-путь. Данный балансировщик используется вместе с контроллером балансировщика нагрузки приложения. Контроллер позволяет разработчикам автоматизировать развертывание приложения и распределять нагрузку на него, не прибегая к консоли или API, а написав всего несколько строк на YAML.

Сетевой

Работает на уровне 4. Основой маршрутизации являются порты TCP/UDP входящего трафика, трафик направляется на хосты, на которых запущены сервисы на указанных портах. Сетевой балансировщик может быть развернут с присвоением эластичного IP-адреса — этим свойством обладают балансировщики только данного типа.

Шлюз

Балансировщик нагрузки шлюза управляет трафиком на устройства на уровне VPC. Данный тип балансировщика может использоваться с сетевыми устройствами типа DPI (проверка сетевых пакетов по их содержимому) или прокси-серверы. Мы описываем здесь этот тип балансировщика для полноты представления сервисов AWS, на самом деле он не используется внутри экосистемы EKS Kubernetes.

Балансировщики нагрузки AWS имеют несколько атрибутов, которые важно понимать, когда работа в облаке VPC ведется не только с контейнерами, но и с другими приложениями:

Правило

(Только для балансировщиков нагрузки на приложения.) Правила, которые вы задаете для слушающего компонента, определяют, каким образом балансировщик будет направлять запросы на назначенные компоненты из групп назначения.

Слушающий компонент

Проверяет, поступили ли запросы от клиентов. Поддержка HTTP и HTTPS на портах 1–65535.

Назначенный компонент

EC2-сервер, IP-адрес, поды или Lambda с запущенным на нем кодом приложения.

Группа назначения

Используется для маршрутизации запросов на отслеживаемый компонент назначения.

Тест работоспособности

Тест, чтобы убедиться, что назначенный компонент способен принимать запросы клиента.

Эти атрибуты балансировщика нагрузок на приложение показаны на рис. 6.8. Слушающий компонент постоянно тестирует поступающие запросы на соответствие заданному протоколу и портам. Каждый слушающий компонент имеет свой набор правил, определяющий, куда направить запрос. Правила содержат тип действия, которое должно быть совершено применительно к запросу:

Cognito-идентификация

(HTTPS-прослушка.) Использовать Amazon Cognito для идентификации пользователей.

oidc-идентификация

(HTTPS-прослушка.) Использовать для идентификации пользователей систему идентификации, совместимую с OpenID Connect.

Фиксированный ответ

Возвратить заданный HTTP-ответ.

Перенаправление

Перенаправить запросы на заданные группы назначений.

Переадресация

Переадресовать запросы с одного URL на другой.

Действие с наименьшим порядковым номером выполняется первым. Каждое правило должно содержать ровно одно из следующих действий: перенаправить, пере-

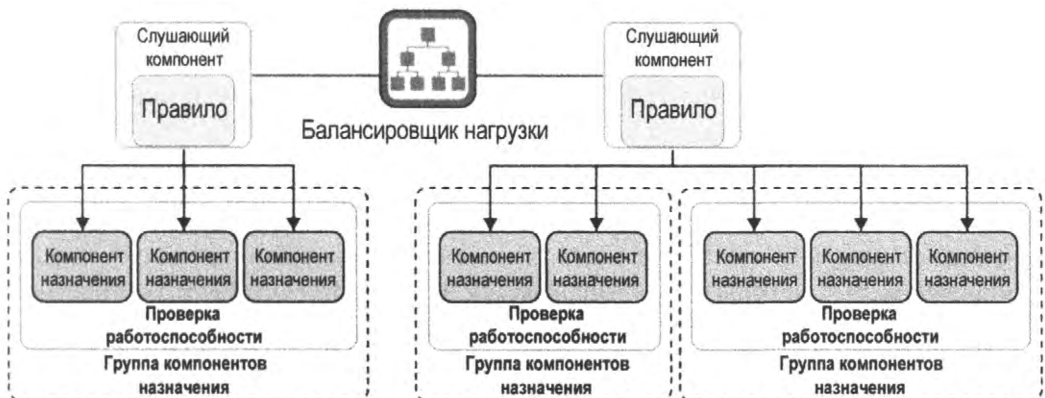


Рис. 6.8. Компоненты балансировщика нагрузки

адресовать или возвратить заданный ответ. На рис. 6.8 показаны группы назначений, которые являются адресатами правил по перенаправлению трафика. Каждый компонент группы проходит тест на работоспособность, так что балансировщик нагрузки точно знает, какие компоненты готовы к работе и могут получать запросы.

Теперь, когда мы познакомились с сетевыми компонентами AWS, мы можем начать рассмотрение того, каким образом сервис EKS строит на основе этих компонентов сеть и обеспечивает функционирование кластеров и сетей Kubernetes.

Эластичный сервис Kubernetes от Amazon

Эластичный сервис Kubernetes (EKS) от Amazon — это управляемый AWS сервис Kubernetes. Он дает возможность разработчикам, администраторам кластеров и сетей быстро развертывать кластеры Kubernetes производственного уровня. Благодаря масштабированию в сетевых и облачных сервисах AWS для развертывания многих сервисов со всеми их компонентами часто достаточно одного запроса API.

Каким образом это осуществляется в EKS? Подобно другим сервисам AWS, EKS с каждой новой версией приобретал все новые свойства и становился более простым в использовании. В настоящее время EKS поддерживает развертывания на уровне предприятия с помощью EKS Anywhere, бессерверную работу — через EKS Fargate и даже работу с узлами под Windows. Кластеры EKS можно развертывать стандартно через AWS CLI или с консоли через `eksctl` — инструмент командной строки, разработанный Weaveworks. На данный момент — это самый простой способ установки всех компонентов, необходимых для запуска EKS. В следующем разделе мы подробно рассмотрим требования, выполнение которых необходимо для работы кластера EKS, и каким образом `eksctl` обеспечивает выполнение этих требований.

Итак, перейдем к изучению сетевых компонентов кластера EKS.

Узлы EKS

В EKS существуют три типа рабочих узлов: управляемые EKS группы узлов, самоуправляемые узлы и AWS Fargate. Выбор между ними осуществляет администратор кластера в зависимости от того, насколько глубокий контроль и за какую цену он хочет получить.

Управляемая группа узлов

Управляемые EKS группы узлов создают экземпляры EC2 и управляют ими. Все управляемые узлы предоставляются как часть группы автоматического масштабирования EC2, которая тоже управляется через AMAZON EKS. Все ресурсы, включая экземпляры EC2 и группы автоматического масштабирования, доступны внутри вашего аккаунта в AWS. Группа автоматического масштабирования внутри вашей группы узлов покрывает все подсети, которые вы определяете при создании группы.

Самоуправляемая группа узлов

Узлы EKS работают внутри вашего аккаунта в AWS и связываются с уровнем управления кластера через конечную точку API. Вы разворачиваете узлы в группу узлов. Группа узлов — это набор экземпляров EC2, которые развернуты в группе автоматического масштабирования. Все экземпляры в группе узлов должны удовлетворять следующим условиям:

- Быть одного типа.
- Иметь одну и ту же конфигурацию Amazon Machine Image.
- Использовать одну и ту же IAM роль узла Amazon EKS.

AWS Fargate

Amazon EKS интегрирует Kubernetes в AWS Fargate, используя контроллеры, которые сформированы AWS с помощью высокоуровневой расширяемой модели, предоставляемой Kubernetes. Каждый под, запущенный через Fargate, имеет свою собственную изолирующую границу и не имеет общих ресурсов ядра, процессора, памяти и общего эластичного сетевого интерфейса с другими подами. Также поды, запущенные через Fargate, не могут использовать группы безопасности для подов.

Тип запущенного экземпляра виртуального сервера влияет также на кластерную сеть. В EKS число подов, которые могут быть запущены на узлах, определяется числом IP-адресов, назначаемым для данного типа. Подробнее мы рассмотрим это ниже в разделах «CNI для AWS VPC» и «Инструмент `eksctl`».

Узлы должны иметь возможность взаимодействовать с управляющим уровнем Kubernetes и других сервисов AWS. Для работы кластера EKS критичным является пространство IP-адресов. Узлы, поды и все остальные сервисы используют диапазон адресов CIDR для компонентов VPC. Облако EKSVPC требует наличия NAT-шлюза для частных подсетей и маркировки данных подсетей для использования совместно с EKS:

```
Key - kubernetes.io/cluster/<cluster-name>
Value - shared
```

Расположение каждого узла, т. е. топология ваших подсетей и маршрутизация трафика через Kubernetes API, будет определять «режим» функционирования EKS.

Режим EKS

На рис. 6.9 показаны компоненты EKS. Управляющий уровень Amazon EKS создает для каждого кластера в вашем облаке VPC до четырех эластичных сетевых интерфейсов с перекрестным доступом. EKS использует два облака VPC, одно для управляющего уровня Kubernetes, включая сюда Kubernetes мастер-API, API балансировки нагрузки и `etcd`; другое — это пользовательское облако, где на рабочих узлах EKS будут запускаться ваши поды. При загрузке экземпляров EC2 запускается также Kubelet. Kubelet получает доступ к конечной точке кластера Kubernetes, чтобы зарегистрировать узел. Он соединяется либо с конечной точкой публичной

сети вне VPC, либо с конечной точкой частной сети внутри VPC. Команда `kubectl` получает доступ к конечной точке API в EKS VPC. Конечные пользователи получают доступ к приложениям, запущенным в пользовательском облаке.

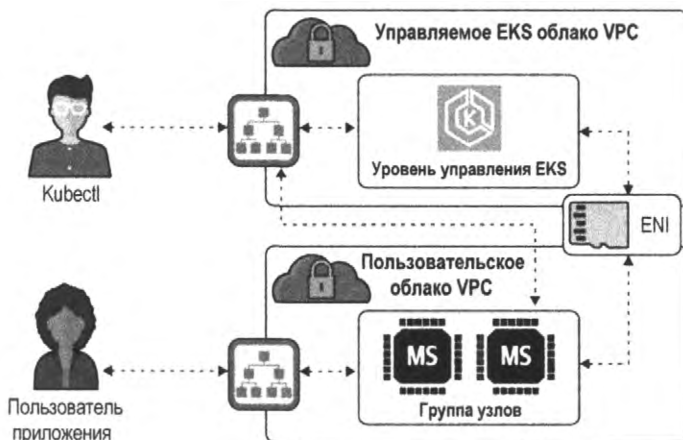


Рис. 6.9. Схема взаимодействия в EKS

Существуют три способа сконфигурировать для EKS управляющий трафик кластера и конечную точку Kubernetes API; они зависят от того, где работают уровень управления и уровень данных для компонентов Kubernetes.

Есть следующие режимы функционирования:

Только публичные сети

Все запускается в публичной подсети, включая сюда и рабочие узлы.

Только частные сети

Запуск только в частной подсети, Kubernetes не может реализовать балансировку нагрузки для интернет-трафика.

Смешанный

Комбинация частных и публичных подсетей.

Конечная точка в публичной подсети — это опция по умолчанию; это сделано потому, что балансировщик нагрузки для конечной точки API находится в публичной подсети — см. рис. 6.10. Запросы Kubernetes API, исходящие из облака VPC кластера, например когда рабочий узел обращается к уровню управления, покидают пользовательское виртуальное облако, но не сеть Amazon. С точки зрения безопасности надо иметь в виду, что когда конечные точки API находятся в публичной подсети, то они доступны через Интернет.

Рис. 6.11 иллюстрирует режим «только частные сети»: весть трафик к API вашего кластера должен поступать только из облака, где развернут кластер. API-сервер не имеет доступа к Интернету, любые команды `kubectl` должны приходить изнутри VPC или из связанной с ним сети. Конечная точка API кластера распознается сервером DNS как частный IP-адрес в виртуальном облаке.

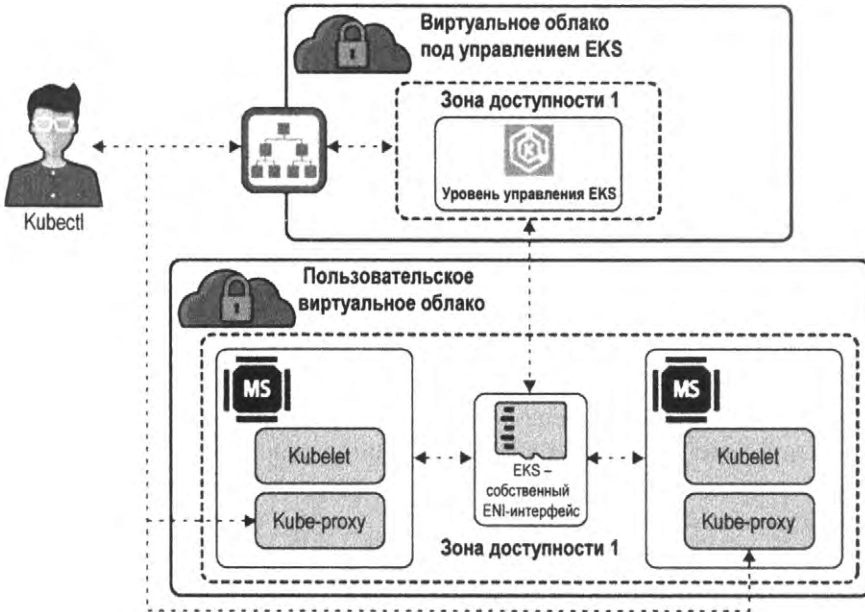


Рис. 6.10. EKS в режиме «только публичные сети»

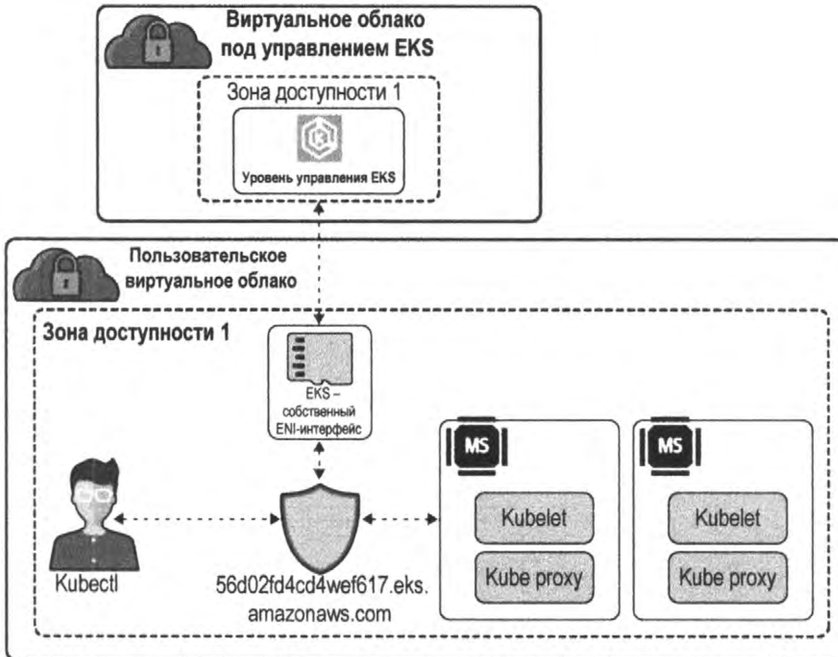


Рис. 6.11. EKS в режиме «только частные сети»

Если разрешены как частные, так и публичные конечные точки, то запросы Kubernetes API, исходящие из облака VPC, взаимодействуют с уровнем управления посредством EKS — управляемых ENI-интерфейсов внутри пользовательского VPC, как показано на рис. 6.12. API кластера доступен через Интернет, но этот доступ может быть ограничен использованием групп безопасности и списков NACL.

Для информации о других способах развертывания EKS смотрите документацию AWS.

Выбор режима работы — это критически важное решение, которое должны принять администраторы кластера. Это решение затронет трафик приложения, маршрутизацию для балансировщиков нагрузки и безопасность кластера. Помимо этих существуют и другие требования к развертыванию кластера в EKS. Для того чтобы обеспечить удовлетворение этих требований, предлагается инструмент `eksctl`. Посмотрим, каким образом он работает.

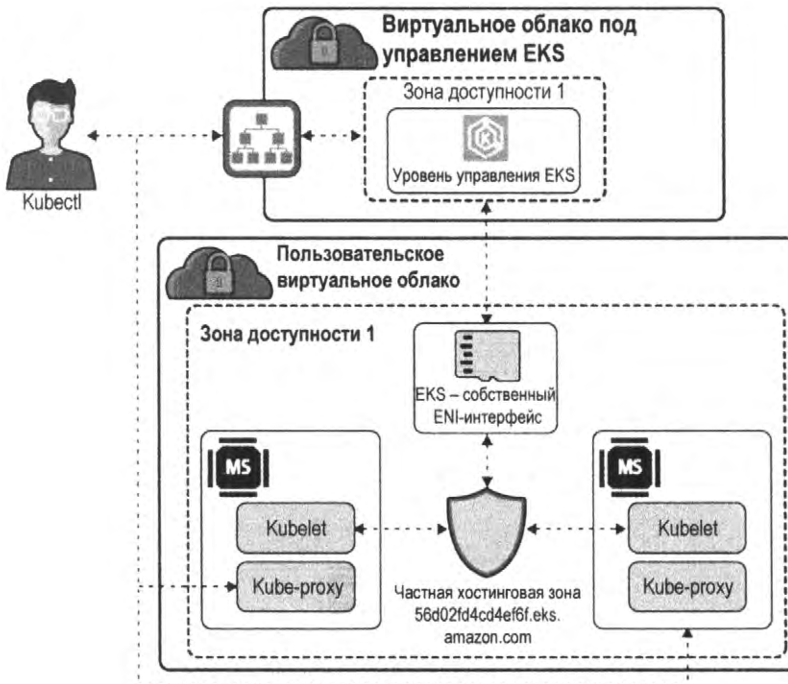


Рис. 6.12. Смешанный (частные и публичные сети) режим EKS

Инструмент *eksctl*

`eksctl` — это инструмент командной строки, разработанный Weaveworks, он предлагает наиболее простой способ развертывания всех компонент, необходимых для работы EKS.

Полная информация по `eksctl` имеется на соответствующей веб-странице.

По умолчанию `eksctl` создает кластер со следующими параметрами:

- Автоматически сгенерированное имя кластера.
- Два рабочих узла размером m5.large.
- Использование официального AWS EKS AMI.
- Регион AWS по умолчанию Us-west-2
- Выделенное виртуальное облако VPC.

Для выделенного VPC с диапазоном CIDR 192.168.0.0/16 `eksctl` по умолчанию создает 8/19 подсетей: три частные, три публичные и две резервные. `eksctl` также разворачивает NAT-шлюз, который делает возможным взаимодействие между узлами в частных подсетях и шлюзом Интернета; это делается для обеспечения доступа к требуемым образам контейнеров и связи с API Amazon S3 и Amazon ECR.

Для кластера EKS устанавливаются две группы безопасности:

Группа безопасности для связи между узлами

Дает возможность узлам взаимодействовать друг с другом, используя любые порты.

Группа безопасности уровня управления

Разрешает взаимодействие между уровнем управления и группами рабочих узлов.

Для групп узлов в публичных подсетях SSH выключается. Экземпляры (инстансы) EC2 в начальной группе узлов получают публичный IP-адрес и могут быть доступны через высокоуровневые порты.

По умолчанию eksctl создает группу, содержащую два узла размером m5.large. А сколько подов можно запустить на этом узле? AWS предлагает формулу, основываясь на типе узла и числе интерфейсов и IP-адресов, которые он может поддерживать. Формула следующая:

$$(\text{Число сетевых интерфейсов для типа экземпляра сервера} \times \times (\text{число IP-адресов на сетевой интерфейс} - 1)) + 2.$$

Используя эту формулу и размер экземпляра по умолчанию, получаем, что узел m5.large может поддерживать максимум 29 подов.

Системные поды уменьшают максимальное число доступных подов. Плагин CNI и `kube-proxy` запускаются на каждом узле кластера, так что в вашем распоряжении на экземпляре сервера размером m5.large останется только 27 подов. CoreDNS запускается на узлах кластера, что тоже уменьшает максимальное число подов, которые могут быть запущены на узле.

Команды, управляющие кластерами, должны выбирать размеры кластеров и типы инстансов таким образом, чтобы развертывания не сталкивались с проблемами,

связанными с ограниченным числом узлов или IP-адресов. Поды будут находиться «в режиме ожидания», если не будет доступных узлов с IP-адресом пода. Масштабирование для группы узлов EKS также может столкнуться с ограничениями у инстансов EC2 и привести к цепочке сбоев.

Все эти сетевые опции задаются в конфигурационном файле `eksctl`.

Опции `eksctl` для виртуального облака можно посмотреть в документации по `eksctl`.

Мы узнали, что размер узла важен для выполнения адресации пода и для определения числа подов, которые могут быть запущены на узле. После развертывания узла IP-адресация подов управляется через CNI-интерфейс для AWS VPC. Посмотрим подробнее, как это работает.

CNI для виртуального облака в AWS

AWS в рамках проекта с открытым кодом реализовал поддержку интерфейсов CNI. Плагин AWS VPC CNI для Kubernetes обеспечивает высокую пропускную способность и доступность, низкую задержку и минимальный джиттер в сети AWS. Сетевые инженеры могут использовать существующие для AWS VPC сетевые решения и политики безопасности для построения в AWS кластеров Kubernetes. Это подразумевает применение исходных AWS-сервисов типа журналирования потоков в виртуальном облаке, политик маршрутизации и групп безопасности для защиты сетевого трафика.

Документация по CNI для AWS VPC доступна на GitHub.

Система сетевых интерфейсов CNI для виртуального облака в AWS имеет два компонента:

CNI-плагин

CNI-плагин, будучи вызванным, отвечает за связывание сетевых стеков хоста и пода. Он также конфигурирует интерфейсы и виртуальные пары Ethernet.

ipamd

Долгоживущий локальный по отношению к узлу демон ipam отвечает за сохранение пула доступных IP-адресов и назначение поду IP-адреса.

На рис. 6.13 показаны функции плагина CNI для VPC. Пользовательское облако VPC с подсетью 10.200.1.0/24 в AWS предоставляет нам для использования в данной подсети 250 адресов. В кластере есть два узла. В управляемых EKS узлах AWS интерфейсы CNI запускаются как демон-процессы. В нашем примере каждый узел имеет только один запущенный под со вторичными IP-адресами на ENI, 10.200.1.6 и 10.200.1.8, для каждого пода. Когда рабочий узел впервые появляется в кластере, имеется только один ENI, и все адреса относятся к этому ENI. Когда под 3 назначается на узел 1, то ipamd присваивает IP-адрес интерфейсу ENI данного пода. То же самое относится к адресу 10.200.1.7 на узле 2 с подом 4.

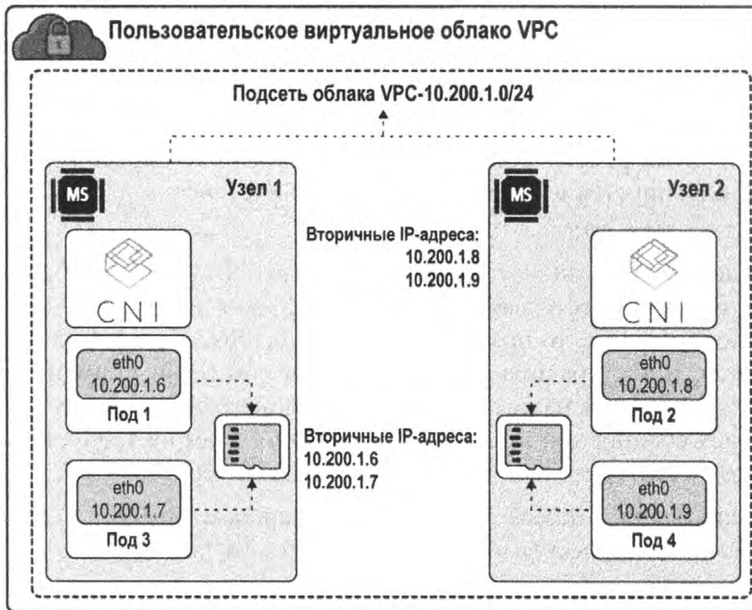


Рис. 6.13. Интерфейсы CNI для виртуального облака в AWS

Когда рабочий узел впервые появляется в кластере, имеется только один ENI, и все адреса относятся к этому ENI. Если конфигурационный файл отсутствует, то ipamd всегда старается поддержать один дополнительный ENI. Если несколько запущенных на одном узле подов выходят на рамки предельного числа адресов на одном ENI, то бэкенд CNI начинает создавать новые ENI. CNI-плагин создает несколько ENI для экземпляров EC2, а затем связывает с этими ENI вторичные IP-адреса. CNI-плагин позволяет назначать на экземпляр произвольное число IP-адресов.

Система CNI для AWS VPC имеет множество конфигурационных опций. Перечислим некоторые из них:

`AWS_VPC_CNI_NODE_PORT_SUPPORT`

Показывает, активированы ли сервисы NodePort на интерфейсе первичной сети рабочего узла. Это требует дополнительных правил в `iptables` и активации проверки обратного адреса на первичном интерфейсе.

`AWS_VPC_K8S_CNI_CUSTOM_NETWORK_CFG`

Рабочие узлы могут быть сконфигурированы в публичных подсетях, так что вам понадобятся установки для подов, позволяющие разворачивать поды в частных подсетях. Или если требования по безопасности для подов отличаются от прочих, запущенных на данном узле, то установка данной опции в `true` позволит решить задачу.

`AWS_VPC_ENI_MTU`

По умолчанию устанавливается 9001. Используется для задания максимального размера блока данных пакета (MTU) для подключенных ENI. Допускается диапазон от 576 до 9001.

`WARM_ENI_TARGET`

Задает число свободных эластичных сетевых интерфейсов (и все их доступные IP-адреса), которые демон `ipamd` должен поддерживать в состоянии готовности для подов, назначаемых на узел. По умолчанию `ipamd` предоставляет подам один эластичный интерфейс и все его IP-адреса. Число IP-адресов на один сетевой интерфейс варьируется в зависимости от типа инстанса.

`AWS_VPC_K8S_CNI_EXTERNALSNAT`

Задает, будет ли использоваться внешний шлюз NAT, чтобы обеспечить преобразование исходных сетевых адресов (SNAT) для вторичных IP-адресов ENI. Если установлен в true, то правило в `iptables` для SNAT и правило для VPC IP не применяются, и эти правила удаляются, если они были применены ранее. Выключайте SNAT, если надо разрешить входной трафик для подов от внешних VPN, прямых соединений и внешних VPC и при этом не требуется, чтобы поды имели выход в Интернет напрямую через шлюз Интернета.

Например, если вашим подам с частным IP-адресом требуется взаимодействие с другими частными адресными пространствами, то активируйте опцию `AWS_VPC_K8S_CNI_EXTERNALSNAT` с помощью команды:

```
kubectl set env daemonset
-n kube-system aws-node AWS_VPC_K8S_CNI_ EXTERNALSNAT=true
```

Полная информация по сетевым взаимодействиям подов в EKS содержится в документации EKS.

Система CNI для виртуального облака в AWS предлагает широкий набор возможностей для управления сетевым взаимодействием в EKS.

В AWS существует ингресс-контроллер для балансировщика нагрузки приложения (ALB), который позволяет упростить и автоматизировать управление развертыванием и работой приложения в облачной сети AWS. В следующем разделе мы разберем его работу.

Ингресс-контроллер для AWS ALB

На примере, показанном на рис. 6.14, посмотрим, каким образом балансировщик нагрузки AWS ALB работает с Kubernetes. Чем занимается ингресс-контроллер, было рассказано в *главе 5*.

Обсудим механизм работы ингресс-контроллера для балансировщика ALB:

1. Ингресс-контроллер для ALB отслеживает входной трафик от API-сервера. Когда будут выполнены заданные требования, то контроллер запускает процесс создания балансировщика ALB.
2. Для нового ресурса ингресса в AWS создан ALB. Ресурс может быть как внутренним, так и внешним по отношению к кластеру.
3. Для каждого отдельного сервиса Kubernetes, описанного в ресурсе ингресса, в AWS создаются целевые группы.

4. Для каждого порта, указанного в аннотации к ресурсу ингресса, создаются слушающие компоненты. Если не указаны явно, то для HTTP и HTTPS трафика указываются порты по умолчанию. Сервисы NodePort для каждого сервера создают порты узлов, которые используются для проверки работоспособности.
5. Для каждого пути, указанного в ресурсе ингресса, создаются правила. Тем самым обеспечивается, что трафик к заданному адресу направляется к соответствующему сервису Kubernetes.

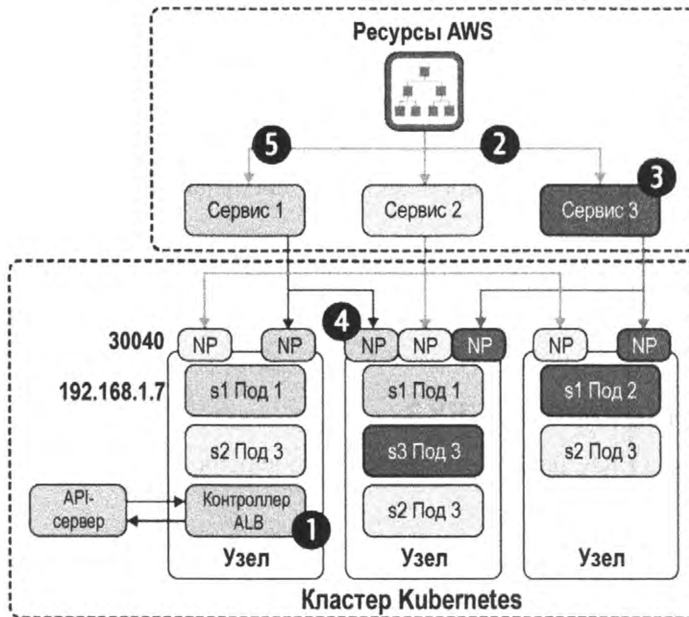


Рис. 6.14. Балансировщик нагрузки приложения (ALB) в AWS

Способ, которым трафик попадает на узлы и поды, зависит от одного из двух режимов, поддерживаемых балансировщиком ALB:

Режим Instance

Входной трафик начинается от ALB и доходит до узлов Kubernetes через порты отдельных сервисов. Это означает, что сервисы, указываемые в ресурсах ингресса, должны объявляться как `type:NodePort`, чтобы обеспечить доступ к ним со стороны балансировщика ALB.

Режим IP

Входной трафик начинается от ALB и напрямую попадает на поды Kubernetes. Интерфейсы CNI должны поддерживать для подов IP-адреса прямого доступа через вторичные IP-адреса на ENI.

Ингресс-контроллер для AWS ALB дает возможность разработчикам управлять своими сетевыми компонентами таким же образом, как это происходит с компонентами приложения, т. е. нет необходимости инсталлировать еще какие-либо инструменты контроля и управления.

Сетевые компоненты AWS плотно интегрированы с EKS. Понимание основ их работы важно для всех, кто намеревается развернуть свои приложения в Kubernetes, используя для этого AWS EKS. Размер ваших подсетей, расположение узлов в этих подсетях и, конечно, размер узлов будут определять, насколько большую сеть подов и сервисов вы сможете запустить на базе AWS. Использование управляемых сервисов, подобных EKS, наряду с инструментами типа `eksctl` позволит значительно снизить операционные издержки, связанные с работой кластера Kubernetes в AWS.

Развертывание приложения в кластере AWS EKS

Перечислим шаги, необходимые для развертывания кластера EKS, чтобы продолжить работу с нашим веб-сервером на языке Go:

1. Развернуть кластер EKS.
2. Развернуть веб-серверы для приложения и для балансировщика нагрузки.
3. Протестировать.
4. Развернуть ингресс-контроллер балансировщика нагрузки приложения и протестировать.
5. Удалить лишнее.

Развертывание кластера EKS

Развернем кластер EKS, используя последнюю версию EKS 1.20:

```
export CLUSTER_NAME=eks-demo
eksctl create cluster -N 3 --name ${CLUSTER_NAME} --version=1.20
eksctl version 0.54.0
using region us-west-2
setting availability zones to [us-west-2b us-west-2a us-west-2c]
subnets for us-west-2b - public:192.168.0.0/19 private:192.168.96.0/19
subnets for us-west-2a - public:192.168.32.0/19 private:192.168.128.0/19
subnets for us-west-2c - public:192.168.64.0/19 private:192.168.160.0/19
nodegroup "ng-90b7a9a5" will use "ami-0a1abe779ecfc6a3e" [AmazonLinux2/1.20]
using Kubernetes version 1.20
creating EKS cluster "eks-demo" in "us-west-2" region with un-managed nodes
will create 2 separate CloudFormation stacks for cluster itself and the initial
nodegroup
if you encounter any issues, check CloudFormation console or try
'eksctl utils describe-stacks --region=us-west-2 --cluster=eks-demo'
CloudWatch logging will not be enabled for cluster "eks-demo" in "us-west-2"
you can enable it with
'eksctl utils update-cluster-logging --enable-types={SPECIFY-YOUR-LOG-TYPES-HERE
(e.g. all)} --region=us-west-2 --cluster=eks-demo'
Kubernetes API endpoint access will use default of
{publicAccess=true, privateAccess=false} for cluster "eks-demo" in "us-west-2"
2 sequential tasks: { create cluster control plane "eks-demo",
3 sequential sub-tasks: { wait for control plane to become ready, 1 task:
{ create addons }, create nodegroup "ng-90b7a9a5" } }
```

```
building cluster stack "eksctl-eks-demo-cluster"
deploying stack "eksctl-eks-demo-cluster"
waiting for CloudFormation stack "eksctl-eks-demo-cluster"
<truncate>
building nodegroup stack "eksctl-eks-demo-nodegroup-ng-90b7a9a5"
--nodes-min=3 was set automatically for nodegroup ng-90b7a9a5
deploying stack "eksctl-eks-demo-nodegroup-ng-90b7a9a5"
waiting for CloudFormation stack "eksctl-eks-demo-nodegroup-ng-90b7a9a5"
<truncated>
waiting for the control plane availability...
saved kubeconfig as "/Users/strongjz/.kube/config"
no tasks
all EKS cluster resources for "eks-demo" have been created
adding identity
"arn:aws:iam::1234567890:role/
eksctl-eks-demo-nodegroup-ng-9-NodeInstanceRole-TLKVDDVTW2TZ" to auth ConfigMap
nodegroup "ng-90b7a9a5" has 0 node(s)
waiting for at least 3 node(s) to become ready in "ng-90b7a9a5"
nodegroup "ng-90b7a9a5" has 3 node(s)
node "ip-192-168-31-17.us-west-2.compute.internal" is ready
node "ip-192-168-58-247.us-west-2.compute.internal" is ready
node "ip-192-168-85-104.us-west-2.compute.internal" is ready
kubectl command should work with "/Users/strongjz/.kube/config",
try 'kubectl get nodes'
EKS cluster "eks-demo" in "us-west-2" region is ready
```

Из этой распечатки мы видим, что EKS сгенерировал группу узлов, eksctl-eks-demo-nodegroup-ng-90b7a9a5, с тремя узлами:

```
ip-192-168-31-17.us-west-2.compute.internal
ip-192-168-58-247.us-west-2.compute.internal
ip-192-168-85-104.us-west-2.compute.internal
```

Они все находятся внутри виртуального облака с тремя публичными и тремя частными подсетями, действующими в трех зонах доступности:

```
public:192.168.0.0/19  private:192.168.96.0/19
public:192.168.32.0/19  private:192.168.128.0/19
public:192.168.64.0/19  private:192.168.160.0/19
```

Мы использовали для `eksctl` установки по умолчанию и развернули k8s API как конечную точку публичной сети `{publicAccess=true, privateAccess=false}`.

Теперь развернем наше веб-приложение в кластере и обеспечим доступ к нему, используя сервис балансировки нагрузки LoadBalancer.

Развертывание тестового приложения

Приложения можно развертывать поодиночке или все сразу. *dnsutils.yml* — это наш под для проверки dnsutils, *database.yml* — база данных Postgres для тестирования сетевой связности пода, *web.yml* — наш веб-сервер:

```
kubectl apply -f dnsutils.yml,database.yml,web.yml
```

Выполним команду kubectl get pods, чтобы убедиться, что все поды работают нормально:

```
kubectl get pods -o wide
NAME                     READY  STATUS  IP              NODE
app-6bf97c555d-5mzfb 1/1    Running 192.168.15.108 ip-192-168-0-94
app-6bf97c555d-76fgm 1/1    Running 192.168.52.42  ip-192-168-63-151
app-6bf97c555d-gw4k9 1/1    Running 192.168.88.61  ip-192-168-91-46
dnsutils             1/1    Running 192.168.57.174 ip-192-168-63-151
postgres-0           1/1    Running 192.168.70.170 ip-192-168-91-46
```

Теперь проверим сервис LoadBalancer:

```
kubectl get svc clusterip-service
NAME                 TYPE          CLUSTER-IP
EXTERNAL-IP                                                          PORT(S)      AGE
clusterip-service    LoadBalancer  10.100.159.28
a76d1c69125e543e5b67c899f5e45284-593302470.us-west-2.elb.amazonaws.com 80:32671/TCP 29m
```

У сервиса также есть конечные точки:

```
kubectl get endpoints clusterip-service
NAME                 ENDPOINTS                                                    AGE
clusterip-service    192.168.15.108:8080,192.168.52.42:8080,192.168.88.61:8080 58m
```

Мы должны убедиться, что внутри кластера к приложению есть доступ через ClusterIP и порт 10.100.159.28:8080, имя сервиса и порт, clusterip-service:80 и, наконец, через IP-адрес пода и порт 192.168.15.108:8080:

```
kubectl exec dnsutils -- wget -qO- 10.100.159.28:80/data
Database Connected

kubectl exec dnsutils -- wget -qO- 10.100.159.28:80/host
NODE: ip-192-168-63-151.us-west-2.compute.internal, POD IP:192.168.52.42

kubectl exec dnsutils -- wget -qO- clusterip-service:80/host
NODE: ip-192-168-91-46.us-west-2.compute.internal, POD IP:192.168.88.61

kubectl exec dnsutils -- wget -qO- clusterip-service:80/data
Database Connected

kubectl exec dnsutils -- wget -qO- 192.168.15.108:8080/data
Database Connected

kubectl exec dnsutils -- wget -qO- 192.168.15.108:8080/host
NODE: ip-192-168-0-94.us-west-2.compute.internal, POD IP:192.168.15.108
```

Порт базы данных доступен через dnsutils, если задать IP-адрес пода и порт 192.168.70.170:5432 или имя сервиса и порт — postgres:5432:

```
kubectl exec dnsutils -- nc -z -vv -w 5 192.168.70.170 5432
192.168.70.170 (192.168.70.170:5432) open
sent 0, rcvd 0

kubectl exec dnsutils -- nc -z -vv -w 5 postgres 5432
postgres (10.100.106.134:5432) open
sent 0, rcvd 0
```

Итак, приложение внутри кластера запущено и работает. Теперь проверим доступ к нему от источников вне кластера.

Тестирование сервиса LoadBalancer для веб-сервера

Воспользовавшись kubectl, получим информацию, необходимую для тестирования, а именно ClusterIP, внеший IP-адрес и все порты:

```
kubectl get svc clusterip-service
NAME                TYPE            CLUSTER-IP
EXTERNAL-IP                                                              PORT S       AGE
clusterip-service   LoadBalancer    10.100.159.28
a76d1c69125e543e5b67c899f5e45284-593302470.us-west-2.elb.amazonaws.com  80:32671/TCP 29m
```

Используем внешний IP-адрес балансировщика нагрузки:

```
wget -qO-
a76d1c69125e543e5b67c899f5e45284-593302470.us-west-2.elb.amazonaws.com/data
Database Connected
```

Протестируем балансировщик нагрузки и выполним несколько запросов на наши бэкенды:

```
wget -qO-
a76d1c69125e543e5b67c899f5e45284-593302470.us-west-2.elb.amazonaws.com/host
NODE: ip-192-168-63-151.us-west-2.compute.internal, POD IP:192.168.52.42

wget -qO-
a76d1c69125e543e5b67c899f5e45284-593302470.us-west-2.elb.amazonaws.com/host
NODE: ip-192-168-91-46.us-west-2.compute.internal, POD IP:192.168.88.61

wget -qO-
a76d1c69125e543e5b67c899f5e45284-593302470.us-west-2.elb.amazonaws.com/host
NODE: ip-192-168-0-94.us-west-2.compute.internal, POD IP:192.168.15.108

wget -qO-
a76d1c69125e543e5b67c899f5e45284-593302470.us-west-2.elb.amazonaws.com/host
NODE: ip-192-168-0-94.us-west-2.compute.internal, POD IP:192.168.15.108
```

Команда `kubectl get pods -o wide` снова даст нам возможность проверить, что данные нашего пода соответствуют данным в запросах балансировщика нагрузки:

```
kubectl get pods -o wide
NAME                     READY STATUS   IP               NODE
app-6bf97c555d-5mzfb     1/1   Running  192.168.15.108   ip-192-168-0-94
app-6bf97c555d-76fgm     1/1   Running  192.168.52.42    ip-192-168-63-151
app-6bf97c555d-gw4k9     1/1   Running  192.168.88.61    ip-192-168-91-46
dnsutils                 1/1   Running  192.168.57.174   ip-192-168-63-151
postgres-0               1/1   Running  192.168.70.170   ip-192-168-91-46
```

Мы также можем проверить порт узла, поскольку утилита `dnsutils` запущена внутри нашего облака на экземпляре EC2. Она может просмотреть DNS на хосте частной сети, `ip-192-168-0-94.us-west-2.compute.internal` , а команда `kubectl get service` сообщила нам номер порта, 32671:

```
kubectl exec dnsutils -- wget -qO-
ip-192-168-0-94.us-west-2.compute.internal:32671/host
NODE: ip-192-168-0-94.us-west-2.compute.internal, POD IP:192.168.15.108
```

Итак, мы убедились, что в нашем кластере как на локальном, так и на внешнем к нему уровне все функционирует нормально.

Развертывание и тестирование ингресс-контроллера для ALB

На некоторых стадиях развертывания нам понадобится ID аккаунта в AWS. Запишем его в переменную среды. Чтобы получить ID аккаунта, можно выполнить следующие команды:

```
aws sts get-caller-identity
{
"UserId": "AIDA2RZMTHAQTEUI3Z537",
"Account": "1234567890",
"Arn": "arn:aws:iam::1234567890:user/eks"
}
export ACCOUNT_ID=1234567890
```

Если это еще не сделано, то нам надо установить OIDC-провайдера для кластера.

Данная процедура требуется, чтобы дать IAM-разрешения поду, запущенному в кластере, который использует IAM для сервис-аккаунта:

```
eksctl utils associate-iam-oidc-provider \
--region ${AWS_REGION} \
--cluster ${CLUSTER_NAME} \
--approve
```

Для роли сервис-аккаунта нам нужно создать IAM-политику, чтобы задать разрешения для ALB-контроллера в AWS:

```
aws iam create-policy \
--policy-name AWSLoadBalancerControllerIAMPolicy \
--policy-document iam_policy.json
```

Теперь нам надо создать сервис-аккаунт и связать его с уже имеющейся IAM-ролью:

```
eksctl create iamserviceaccount \
> --cluster ${CLUSTER_NAME} \
> --namespace kube-system \
> --name aws-load-balancer-controller \
> --attach-policy-arn
arn:aws:iam::${ACCOUNT_ID}:policy/AWSLoadBalancerControllerIAMPolicy \
> --override-existing-serviceaccounts \
--approve
eksctl version 0.54.0
using region us-west-2
i iamserviceaccount (kube-system/aws-load-balancer-controller) was included
(based on the include/exclude rules)
metadata of serviceaccounts that exist in Kubernetes will be updated,
as --override-existing-serviceaccounts was set
1 task: { 2 sequential sub-tasks: { create IAM role for serviceaccount
"kube-system/aws-load-balancer-controller", create serviceaccount
"kube-system/aws-load-balancer-controller" } }
building iamserviceaccount stack
deploying stack
waiting for CloudFormation stack
waiting for CloudFormation stack
waiting for CloudFormation stack
created serviceaccount "kube-system/aws-load-balancer-controller"
```

Все параметры сервис-аккаунта можно посмотреть с помощью следующей команды:

```
kubectl get sa aws-load-balancer-controller -n kube-system -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
annotations:
eks.amazonaws.com/role-arn:
arn:aws:iam::1234567890:role/eksctl-eks-demo-addon-iamserviceaccount-Role1
creationTimestamp: "2021-06-27T18:40:06Z"
labels:
app.kubernetes.io/managed-by: eksctl
name: aws-load-balancer-controller
namespace: kube-system
resourceVersion: "16133"
uid: 30281eb5-8edf-4840-bc94-f214c1102e4f
secrets:
- name: aws-load-balancer-controller-token-dtq48
```

Функция CRD (определение пользовательского ресурса) для TargetGroupBinding позволяет контроллеру связать конечную точку сервиса Kubernetes с TargetGroup в AWS:

```
kubectl apply -f crd.yml
customresourcedefinition.apiextensions.k8s.io/ingressclassparams.elbv2.k8s.aws
configured
```

```
customresourcedefinition.apiextensions.k8s.io/targetgroupbindings.elbv2.k8s.aws
configured
```

Теперь все готово для развертывания ALB-контроллера с помощью Helm.

Установим версию среды для развертывания:

```
export ALB_LB_VERSION="v2.2.0"
```

Развернем ее, добавим репозиторий Helm для eks, получим идентификатор облака VPC, в котором запущен кластер, и наконец развернем контроллер с помощью Helm.

```
helm repo add eks https://aws.github.io/eks-charts
export VPC_ID=$(aws eks describe-cluster \
--name ${CLUSTER_NAME} \
--query "cluster.resourcesVpcConfig.vpcId" \
--output text)

helm upgrade -i aws-load-balancer-controller \
eks/aws-load-balancer-controller \
-n kube-system \
--set clusterName=${CLUSTER_NAME} \
--set serviceAccount.create=false \
--set serviceAccount.name=aws-load-balancer-controller \
--set image.tag="${ALB_LB_VERSION}" \
--set region=${AWS_REGION} \
--set vpcId=${VPC_ID}

Release "aws-load-balancer-controller" has been upgraded. Happy Helming!
NAME: aws-load-balancer-controller
LAST DEPLOYED: Sun Jun 27 14:43:06 2021
NAMESPACE: kube-system
STATUS: deployed
REVISION: 2
TEST SUITE: None
NOTES:
AWS Load Balancer controller installed!
```

Протоколы развертывания можно посмотреть дополнительно.

```
kubectl logs -n kube-system -f deploy/aws-load-balancer-controller
```

Теперь развернем наш ингресс, используя ALB:

```
kubectl apply -f alb-rules.yml
ingress.networking.k8s.io/app configured
```

Выдача команды `kubectl describe ing app` показывает, что балансировщик ALB развернут.

В распечатке присутствует публичный DNS-адрес для ALB, правила для экземпляров и конечные точки сервиса.

```
kubectl describe ing app
Name:          app
Namespace:     default
Address:
k8s-default-app-d5e5a26be4-2128411681.us-west-2.elb.amazonaws.com
Default backend: default-http-backend:80
(<error: endpoints "default-http-backend" not found>)
Rules:
Host   Path    Backends
----   ----    --------
*
/data   clusterip-service:80 (192.168.3.221:8080,
192.168.44.165:8080,
192.168.89.224:8080)
/host   clusterip-service:80 (192.168.3.221:8080,
192.168.44.165:8080,
192.168.89.224:8080)
Annotations: alb.ingress.kubernetes.io/scheme: internet-facing
kubernetes.io/ingress.class: alb
Events:
Type   Reason                  Age                    From     Message
----   ------                  ----                   ----     -------
Normal SuccessfullyReconciled  4m33s (x2 over 5m58s)  ingress  Successfully reconciled
```

А теперь протестируем наш балансировщик ALB!

```
wget -qO- k8s-default-app-d5e5a26be4-2128411681.us-west-2.elb.amazonaws.com/data
Database Connected

wget -qO- k8s-default-app-d5e5a26be4-2128411681.us-west-2.elb.amazonaws.com/host
NODE: ip-192-168-63-151.us-west-2.compute.internal, POD IP:192.168.44.165
```

Уборка мусора

После того как работа с EKS и тестирование завершены, удалите поды приложений и сервис:

```
kubectl delete -f dnsutils.yml,database.yml,web.yml
```

Удалите ALB:

```
kubectl delete -f alb-rules.yml
```

Удалите IAM-политику для контроллера ALB:

```
aws iam delete-policy
--policy-arn arn:aws:iam::${ACCOUNT_ID}:policy/AWSLoadBalancerControllerIAMPolicy
```

Убедитесь, что в PVC не осталось EBS-томов от тестового приложения. Удалите все EBS-тома, оставшиеся в PVC от тестирования базы данных Postgres:

```
aws ec2 describe-volumes --filters
Name=tag:kubernetes.io/created-for/pv/name,Values=*
--query "Volumes[].{ID:VolumeId}"
```

Проверьте, не остались ли работающие балансировщики нагрузки — ALB или другие:

```
aws elbv2 describe-load-balancers --query "LoadBalancers[].LoadBalancerArn"

aws elb describe-load-balancers --query "LoadBalancerDescriptions[].DNSName"
```

Проверим также, удалили ли мы кластер, чтобы не пришлось платить за кластер, который ничего не делает:

```
eksctl delete cluster --name ${CLUSTER_NAME}
```

Итак, мы развернули балансировщик нагрузки, который для каждого сервиса будет в AWS развертывать классический балансировщик ELB. Контроллер ALB позволяет разработчикам использовать ингресс совместно с ALB или NLB, чтобы приложение стало доступно извне кластера. Если бы мы расширили наше приложение до предоставления разнообразных бэкендовых сервисов, то благодаря ингрессу мы использовали бы один балансировщик нагрузки и маршрут, определенный на базе информации протокола 7-го уровня.

В следующем разделе мы изучим GCP по той же схеме, как мы это делали с AWS.

Вычислительное облако Google (GCP)

В 2008 г. Google объявил о выпуске App Engine — модели «платформа как услуга», предназначенной для развертывания приложений на Java, Python, Ruby и Go. Как и все прочие поставщики услуг, GCP постоянно расширяет линейку предлагаемых сервисов. Провайдеры облачных вычислений стремятся к тому, чтобы их продукты отличались друг от друга, так что двух одинаковых не встретишь. Несмотря на это, многие продукты все равно оказываются похожими. Например, GCP Compute Engine является «инфраструктурой как услуга» и предназначена для запуска виртуальных машин. Сеть GCP состоит из 25 облачных регионов, 76 зон и 144 локаций сопряжения сетей с Интернетом. Соединив сеть GCP с Compute Engine, GCP выпустило Google Kubernetes Engine — платформу «контейнер как услуга».

Сетевые сервисы GCP

Управляемые и неуправляемые кластеры Kubernetes в GCP работают по одному и тому же принципу. Узлы в этих кластерах запускаются как инстансы (экземпляры виртуальных серверов) Google Compute Engine. Сети в GCP — это облачные сети VPC. VPC-сети в GCP аналогично сетям в AWS содержат функционал, включающий в себя управление IP-адресами, маршрутизацию, брандмауэры и пиринг.

Для пользователей сеть с предлагает на выбор разные уровни, есть уровни премиум и стандартные. Они различаются по набору функций, типам маршрутизации и производительности, сетевые инженеры сами должны решать, какой уровень больше походит для решения их задач. Уровень премиум предоставляет наиболее высокую производительность. Весь трафик между Интернетом и инстансами в сети VPC осуществляется по возможности внутри сети Google. Если вашим сервисам требу-

ется глобальная доступность, то используйте уровень премиум. Также стоит учесть, что этот уровень устанавливается по умолчанию, если только не сделаны специальные изменения в конфигурации.

Стандартный уровень — это уровень с оптимизацией по стоимости, когда трафик между Интернетом и виртуальными машинами облачной сети идет, как правило, через Интернет. Сетевым инженерам рекомендуется выбирать этот уровень, если их сервисы предполагается разместить в пределах одного региона. Стандартный уровень не может гарантировать высокую производительность, поскольку его производительность не отличается от той, какую демонстрируют обычные интернет-приложения.

Сеть GCP отличается от сетей других провайдеров тем, что она предоставляет так называемые *глобальные* ресурсы. Глобальными они называются потому, что пользователи в рамках определенного проекта имеют к ним доступ из любой зоны. Эти ресурсы включают в себя такие составляющие, как виртуальные облака, брандмауэры и маршруты к ним.

Более подробная информация по уровням содержится в документации по GCP.

Регионы и зоны

Регионы — это независимые географические области, которые содержат несколько зон. Региональные ресурсы предоставляются с избыточностью, поскольку они развертываются сразу в нескольких зонах данного региона. Зоны — это области развертывания ресурсов внутри региона. Одна зона — это, как правило, центр обработки данных внутри региона, и администраторы могут рассматривать его как единственный домен, где могут случаться сбои. Чтобы обеспечить отказоустойчивость развернутых приложений, рекомендуется развертывать их в нескольких зонах региона, а для того, чтобы обеспечить высокую доступность — в нескольких регионах. Если зона становится недоступной, то и все ее ресурсы тоже будут вне доступа, пока собственники не восстановят сервисы.

Виртуальное частное облако

Виртуальное частное облако VPC — это виртуальная сеть, которая обеспечивает связность ресурсов внутри проекта GCP. Подобно аккаунтам и подпискам проекты могут содержать несколько VPC-сетей, по умолчанию все новые проекты начинаются с облачной виртуальной сети, содержащей одну подсеть на каждый регион. Пользовательские облачные сети могут не содержать ни одной подсети. Как уже упоминалось ранее, облачные сети являются глобальными ресурсами и не связаны с определенными регионами или зонами.

Облачная сеть содержит одну или более региональных подсетей. Подсети имеют регион, CIDR и уникальное в глобальном масштабе имя. CIDR для подсети может быть любой, включая и пересекающийся с другим частным адресным пространст-

вом. Выбор CIDR для подсети влияет на то, к каким IP-адресам будет доступ и с какими сетями будет возможна связь.

Google создает VPC сеть по умолчанию путем случайной генерации подсетей для каждого региона. Некоторые подсети могут перекрываться с другой сетью в пределах одного облака (например, с VPC-сетью по умолчанию в рамках другого GCP-проекта), что сделает невозможным пиринг.

Облачные сети поддерживают *пиринг* и общую конфигурацию облака VPC. Пиринг в облачной сети позволяет облаку одного проекта связываться с облаком другого, формируя тем самым сеть 3-го уровня. Пиринг невозможен в перекрывающихся сетях одного облака, поскольку одинаковые IP-адреса присутствуют в обеих сетях. Облако общего пользования дает возможность другому проекту использовать определенные подсети, в частности создавать виртуальные машины, являющиеся частью данной подсети. Более подробная информация содержится в документации по VPC.

Пиринг облачных сетей является стандартной операцией, поскольку организации часто включают в свои проекты GCP различные команды разработчиков, различные приложения и компоненты. Пиринг позволяет улучшить контроль доступа, квоты и сообщения. Некоторые администраторы создают в рамках одного проекта несколько облачных сетей — по тем же соображениям.

Подсеть

Подсеть — это компонент внутри облачной сети с диапазоном первичных IP-адресов и с возможностью иметь ноль или больше диапазонов вторичных адресов. Подсети являются региональными ресурсами, каждая подсеть задает свой диапазон IP-адресов. В регионе может быть больше одной подсети. Конфигурирование подсети может проходить в двух режимах: автоматическом и пользовательском. При создании облачной сети в автоматическом режиме для каждого региона генерируется одна подсеть с IP-адресами в наперед заданном диапазоне. Если создается облачная сеть в пользовательском режиме, то GCP не предоставляет никаких подсетей, диапазон адресов задают сетевые администраторы. Пользовательские облачные сети применяются в организациях и на производственных предприятиях.

GCP дает возможность «резервировать» статические адреса для внутренних и внешних IP-адресов. Пользователи могут применять зарезервированные адреса для инстансов GCE, балансировщиков нагрузки и других аналогичных продуктов. Зарезервированные внутренние IP-адреса имеют имя и могут генерироваться автоматически или назначаться вручную. Резервирование внутреннего статического IP-адреса предотвращает его случайное назначение в период, когда он не используется.

Резервирование внешних IP-адресов действует аналогично. Вы можете запросить автоматически назначаемый IP-адрес, но не можете выбирать, какой IP-адрес зарезервировать. Поскольку вы резервируете IP-адрес, доступный глобально, то при некоторых обстоятельствах с вас могут потребовать за это плату. Вы не можете закрепить за собой внешний IP-адрес, который назначен вам автоматически в качестве эфемерного.

Маршруты и правила брандмауэров

При развертывании облака вы можете сформулировать правила для брандмауэров, чтобы разрешать или запрещать входящий и выходящий трафик ваших приложений. Каждое правило брандмауэра может относиться к входящим или выходящим соединениям, но не одновременно к обоим. GCP применяет эти правила на уровне экземпляров приложения, при этом конфигурация правил привязана к облачной сети, так что правила нельзя сделать общими для VPC-сетей, включая сюда и пиринговые. Облачные правила для брандмауэров действуют в режиме stateful (с запоминанием состояния), так что при запуске TCP-сеанса правила разрешают двунаправленный трафик аналогично тому, как это делают группы безопасности в AWS.

Облачная балансировка нагрузки

Балансировщик нагрузки Google Cloud Load Balancer (GCLB) предоставляет полностью распределенный высокопроизводительный масштабируемый сервис по балансировке нагрузки в GCP, управляемый большим количеством опций. Используя GCLB, вы получаете единственный Anycast IP-адрес, который распространяется на все ваши инстансы по всему земному шару. Дополнительно программно-задаваемые сервисы балансировки нагрузки дают вам возможность применять балансировку к трафику HTTP(S), TCP/SSL и UDP. Вы также можете завершить ваш трафик с помощью SSL прокси и балансировки HTTPS. Внутренняя балансировка нагрузки позволяет выстраивать высокодоступные внутренние сервисы для ваших внутренних инстансов, не требуя при этом доступа к балансировщикам в Интернете.

Большинство пользователей GCP используют GCP балансировщики нагрузки совместно с ингресс-контроллером Kubernetes. GCP предоставляет балансировшики внутренней и внешней нагрузки с поддержкой сетей уровней 4 и 7. Кластеры GKE по умолчанию создают GCP-балансировщик для входного трафика и сервис типа `type:LoadBalancer`.

Чтобы обеспечить доступ к приложению извне GKE-кластера, GKE предоставляет встроенные ингресс-контроллер и сервис-контроллер, которые разворачивают балансировщик GCLB для пользователей GKE. GKE имеет три различных балансировщика, которые управляют доступом и распределяют входящий трафик как можно более равномерно по кластеру. Вы можете сконфигурировать один сервис на одновременное использование нескольких типов балансировщиков:

Внешние балансировщики нагрузки

Управляют внешним трафиком кластера и облачной сети. Для направления трафика на узел Kubernetes внешние балансировщики используют правила адресации, установленные в облачной сети Google.

Внутренние балансировщики нарузки

Управляют трафиком, приходящим из той же облачной сети. Аналогично внешним балансировщикам внутренние также используют правила адресации, установленные в облачной сети Google, для направления трафика на узел Kubernetes.

Балансировщики нагрузки HTTP

Специализированные внешние балансировщики, используемые для HTTP-трафика. Направление трафика на узел Kubernetes происходит путем обращения к ингрессу, а не к правилам адресации.

Когда вы создаете объект ингресса в облаке Google, ингресс-контроллер GKE конфигурирует балансировщик нагрузки HTTP(S) согласно манифесту ингресса и соответствующему манифесту правил сервисов Kubernetes. Клиент посылает запрос на балансировщик нагрузки. Балансировщик — это прокси-сервер, он выбирает узел и адресует запрос на NodeIP:NodePort данного узла. Узел использует свою таблицу NAT в `iptables`, чтобы выбрать под. Как мы узнали в предыдущих главах, правилами `iptables` на узле управляет сервер `kube-proxy`.

Когда ингресс создает балансировщик нарузки, этот балансировщик действует на уровне подов, а не направляет запросы на все узлы (т. е. направляет на отдельные поды). Делается это путем обращения к объекту `Endpoints/EndpointSlice` (см. *главу 5*) и использования IP-адресов отдельных подов в качестве адресов назначения.

Администраторы кластеров могут использовать провайдеры внутрикластерных сервисов ингресса, такие как Nginx или Contour. В такой конфигурации балансировщик нагрузки указывает на узлы, на которых запущен прокси ингресса, который уже направляет запросы на конкретные поды. Данная конфигурация представляется более выгодной для кластеров со многими ингрессами, но она приводит к дополнительным издержкам в производительности.

Инстансы GCE

Инстансы GCE имеют один или более сетевых интерфейсов. Сетевой интерфейс связан с сетью и подсетью, имеет частный IP-адрес и публичный IP-адрес. Частный IP-адрес относится к подсети. Частные IP-адреса могут быть автоматическими и эфемерными, назначаемыми пользователем, и эфемерными, а также статическими. Внешние IP-адреса могут быть автоматическими и эфемерными либо статическими. Вы можете добавлять сетевые интерфейсы к инстансам GCE. Дополнительные сетевые интерфейсы не обязательно должны быть в той же облачной сети. Например, инстанс может быть мостом между двумя виртуальными облаками с разными уровнями защищенности. Рассмотрим, каким образом Google Kubernetes Engine (GKE) использует инстансы и управляет сервисами, обеспечивающими его работу.

Google Kubernetes Engine (GKE)

Система Google Kubernetes Engine (GKE) — это сервис Kubernetes под управлением Google. GKE имеет свой скрытый уровень управления, который напрямую не виден и доступ к которому невозможен. Вы получаете доступ только к определенным конфигурационным файлам уровня управления и к Kubernetes API.

GKE дает доступ только к некоторым конфигурационным параметрам вроде типа машины и масштабирования кластера. Доступны также несколько сетевых установок. На момент написания книги пользователи имеют возможность устанавливать

сетевые политики (через Calico), максимальное число подов на узле (`maxPods` в kubelet, `--node-CIDR-mask-size` в `kube-controller-manager`) и диапазон адресов для пода (`-cluster-CIDR` в `kube-controller-manager`). Прямая установка флагов `apiserver/kube-controller-manager` невозможна.

GKE поддерживает публичные и частные кластеры. Частные кластеры не назначают узлам публичные IP-адреса, т. е. они доступны только в пределах вашей частной сети. Частные кластеры также дают возможность разрешить доступ к Kubernetes API только определенным IP-адресам. GKE управляет рабочими узлами, используя автоматические GCE инстансы и создавая пулы узлов (node pools).

GKE-узлы в облаке Google

Сетевой взаимодействие GKE-узлов сравнимо с работой самоуправляемых кластеров Kubernetes в GKE. Кластеры GKE определяют пулы узлов — наборы узлов, имеющих одинаковую конфигурацию. В эту конфигурацию включены как облачные установки Google, так и общие установки Kubernetes. Пулы узлов задают тип (виртуальной) машины, автомасштабирование и сервис-аккаунт GCE. Вы также можете установить для пула узлов пользовательские метки.

Кластер существует внутри ровно одной виртуальной облачной сети. Отдельные узлы могут иметь сетевые дескрипторы для создания своих собственных правил брандмауэров. Все кластеры GKE версий 1.16 и старше снабжены демоном `kube-proxy`, который автоматически запускается на всех новых узлах в кластере. Размер кластера определяется сконфигурированными установками для подсети — учитывайте это при развертывании масштабируемых кластеров. Есть формула для вычисления максимального числа узлов, N, которое может поддерживать данная сетевая маска. Если S — это размер сетевой маски, диапазон значений которой от 8 до 29, то имеем:

```
N = 2(32- S) - 4
```

Подсчитаем размер сетевой маски, S, требуемый для поддержки максимум N узлов:

```
S = 32 - log2(N+4)
```

В табл. 6.2 приводится число узлов в кластере и как оно меняется в зависимости от размера подсети.

***Таблица 6.2.** Число узлов в кластере в зависимости от размера подсети*

Диапазон первичных IP-адресов	Максимальное число узлов адресов
/29	Минимальный размер для IP-диапазона первичной сети: 4 узла
/28	12 узлов
/27	28 узлов
/26	60 узлов
/25	124 узла
/24	252 узла

Таблица 6.2 (окончание)

Диапазон первичных IP-адресов	Максимальное число узлов адресов
/23	508 узлов
/22	1020 узлов
/21	2044 узла
/20	Размер по умолчанию диапазона первичных IP-адресов подсети в автоматически сконфигурированных сетях· 4092 узла
/19	8188 узлов
/8	Максимальный размер диапазона первичных IP-адресов подсети. 16 777 212 узлов

Если вы используете CNI для системы GKE, то одним концом парный виртуальный интерфейс veth связан с подом в его пространстве имен, а другой конец соединяется с мостовым устройством Linux cbr0.1 — все так, как мы рассматривали в *главах 2* и *3*.

Кластеры расположены либо в зоне, либо на границе регионов; у зональных кластеров имеется только один уровень управления. Региональные кластеры располагают многими репликами уровня управления. Когда вы разворачиваете кластер, то GKE предлагает вам 2 режима: собственный облачный и основанный на маршрутах. Кластер, использующий псевдонимы (alias) IP-адресов, рассматривается как собственный кластер виртуального облака. Кластер, использующий определяемые пользователем статические маршруты в виртуальном облаке, называется кластером на основе маршрутов. Таблица 6.3 показывает, как связаны метод создания кластера и сетевой режим.

***Таблица 6.3.** Сетевой режим кластера и метод создания кластера*

Метод создания кластера	Сетевой режим кластера
Google Cloud Console	Собственный облака VPC
REST API	На основе маршрутов
gcloud версия 256.0.0 и выше или версия 250.0.0 и ниже	На основе маршрутов
gcloud v251.0.0–255.0.0	Собственный облака VPC

При использовании собственных кластеров облака администраторы могут обращаться к группам конечных точек, которые представляют собой группы бэкендов, обслуживаемых балансировщиком нагрузки. Эти группы являются списками IP-адресов, управляемых контроллером группы и используемых балансировщиками нагрузки облака Google. IP-адреса в группе могут быть первчными или вторичными IP-адресами виртуальной машины, что означает, что они могут быть адресами подов. Тем самым делается возможной балансировка нагрузки прямо в контейнере,

при которой трафик от облачного балансировщика Google направляется напрямую к подам.

Собственные облачные кластеры имеют определенные преимущества:

- IP-адреса подов являются маршрутизируемыми внутри облачной кластерной сети.
- IP-адреса подов резервируются в сети еще до момента создания пода.
- Диапазоны IP-адресов подов зависят от пользовательских статических маршрутов.
- Правила брандмауэров применяются к диапазону IP-адресов пода, а не к любому IP-адресу на узлах кластера.
- Связь облачной сети Google с (частной) сетью предприятия распространяется на диапазоны IP-адресов пода.

Рис. 6.15 демонстрирует соответствие компонентов GKE и GCE.

Рис. 6.15. Компоненты группы конечных точек сети и Google Computer Engine (GCE)

Использование групп конечных точек в сети GKE имеет ряд преимуществ:

Повышение производительности сети

Собственный балансировщик нагрузки контейнера напрямую общается с подами, для установления соединения требуется меньше межсетевых переходов, тем самым уменьшаются задержки и улучшается пропускная способность.

Увеличение видимости

Используя собственный балансировщик нагрузки контейнера, вы можете контролировать задержку при передаче HTTP-трафика от балансировщика нагрузки на поды. Становится видимой задержка от балансировщика на каждый из подов, которая суммируется по всем IP-адресам узла. Такая видимость облегчает отладку ваших сервисов на уровне конечных точек.

Поддержка улучшенных механизмов балансировки нагрузки

Собственный балансировщик нагрузки контейнера предлагает в GKE поддержку различных опций балансировки HTTP-трафика, например интеграцию с облачными сервисами Google типа Google C;oud Armor, Cloud CDN и Identity-Aware Proxy. Есть также поддержка алгоритмов балансировки для равномерного распределения трафика.

Как и большинство провайдеров решений на базе Kubernetes, система GKE тесно интегрирована с облачными функциями Google. Хотя большинство ПО для GKE закрыто, система тем не менее использует стандартные ресурсы типа инстансов GCE, которые могут просматриваться и контролироваться так же, как и другие облачные ресурсы Google. Если вы хотите управлять кластером, используя только свои собственные средства, вы потеряете в функциональности, например у вас не будет балансировки нагрузки на уровне контейнеров.

Стоит отметить, что облачные сервисы Google пока не поддерживают IPv6 — в отличие от AWS и Azure.

И наконец, мы рассмотрим работу Kubernetes в облаке Azure.

Azure

Azure от Microsoft, подобно другим облачным провайдерам, предлагает линейку готовых решений для сервисов и работы в сетях. Прежде чем мы приступим к описанию работы Azure AKS, рассмотрим модели развертывания в Azure. В течение нескольких лет Azure разрабатывал и усовершенствовал свои решения, в результате чего в настоящее время предлагаются две модели развертывания. Эти модели различаются в методах развертывания и использования ресурсов, и соответственно выбор одной или другой может сказаться на том, как ресурсы будут эксплуатироваться пользователями.

Одна модель — это классическое развертывание, исторически она была первой, которую использовал Azure. Все ресурсы существовали независимо друг от друга, логическая группировка их отсутствовала. Это было довольно неудобно: пользователи должны были создавать, обновлять и удалять каждый компонент приложения, что приводило к ошибкам, недостатку ресурсов и лишним затратам времени и труда. Отсутствовала также возможность присваивать ресурсам метки, что не позволяло производить поиск и дополнительно осложняло разработку.

В 2014 г. Microsoft предложил Azure Resource Manager (ARM) в качестве новой модели. Эта модель является на сегодня рекомендованной, более того, рекомендуется даже переустановить ваши ресурсы с использованием ARM. Основное новшество данной модели заключалось во введении понятия «группа ресурсов». Группы ресурсов — это логическое объединение ресурсов, которое позволяет проводить трассировку, назначать метки и конфигурировать ресурсы совместно, а не по отдельности.

Теперь, когда мы познакомились с развертыванием и управлением ресурсами в Azure, мы можем перейти к рассмотрению сетевых сервисов Azure и их взаимодействию с Azure Kubernetes Service (AKS) и другими (не Azure) решениями для Kubernetes.

Сетевые сервисы Azure

Основой сетевых сервисов в Azure является виртуальная сеть, известная под именем Azure Vnet. Данная сеть предлагает защищенную виртуальную сетевую инфра-

структуру для связывания между собой ваших развернутых ресурсов Azure, таких как виртуальные машины и кластеры AKS. Благодаря дополнительным возможностям сети Vnet связывают ваши ресурсы с Интернетом и с инфраструктурой вашего предприятия. Если конфигурация не изменялась, то все сети Azure Vnet взаимодействуют с Интернетом по маршруту, устанавливаемому по умолчанию.

На рис. 6.16 сеть Azure Vnet имеет CIDR `192.168.0.0/16`. Как и другие ресурсы Azure, сети Vnet требуют подписки, чтобы быть помещенными в группу ресурсов для Vnet. Безопасность сети Vnet может устанавливаться через опции, некоторые опции, например разрешения IAM, наследуются от группы ресурсов и подписки. Сеть Vnet ограничена определенным регионом. В одном регионе может существовать несколько сетей Vnet, но одна Vnet может работать только внутри одного региона.

Рис. 6.16. Сеть Azure Vnet

Базовая инфраструктура Azure

Microsoft Azure располагает глобально распределенной сетью зон и центров обработки данных. Базовой единицей является регион Azure, который объединяет несколько дата-центров внутри области, определяемой по величине задержки, дата-центры связаны между собой через выделенную сетевую инфраструктуру. В регионе может быть произвольное число дата-центров, но обычно развернуты два-три на регион. Любая область земного шара, в которой присутствует хотя бы один регион Azure, включена в географию Azure.

Регион подразделяется на зоны доступности. Зоны доступности — это физические области, в которых могут существовать один или более дата-центров, имеющих независимое электропитание, охлаждение и сетевую инфраструктуру. Связь между регионом и зонами доступности конструируется таким образом, чтобы выбытие одной зоны не приводило к выбытию всего региона. Каждая зона доступности в регионе связана с другими зонами региона, но не зависит от других зон. Благодаря концепции зон доступности Azure способен предлагать свои сервисы в течение 99.99% времени. Как показано на рис. 6.17, регион состоит из нескольких зон доступности, в которых, в свою очередь, имеется несколько центров обработки данных.

Поскольку сеть Vnet находится в регионе, а регионы разделены на зоны доступности, то доступ к сетям Vnet возможен из нескольких зон данного региона. Как пока-

зано на рис. 6.18, благодаря этому обеспечивается высокая доступность сетевой инфраструктуры, что позволяет сервисам Azure работать с минимальными отказами. Azure также дает возможность использовать балансировщики нагрузки при работе в системах с избыточными ресурсами.

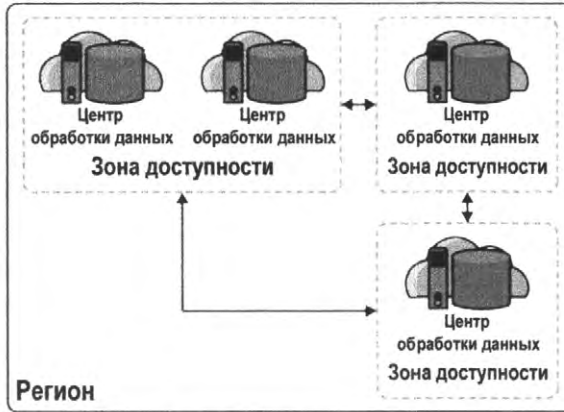


Рис. 6.17. Регион Azure

Рис. 6.18. Сеть Vnet и зоны доступности

В документации Azure содержится актуализированный список географических локаций Azure, регионов и зон доступности.

Подсети

IP-адреса ресурсов не назначаются напрямую сетью Vnet. Наоборот, Vnet определяется через подсети. Подсети получают свои пространства адресов от Vnet. Затем частные IP-адреса присваиваются ресурсам, предоставленным внутри каждой из подсетей. Здесь происходит и адресация кластеров AKS и подов. Как и сети Vnet. подсети Azure также существуют в зонах доступности, как показано на рис. 6.19.

Таблицы маршрутизации

Как было упомянуто выше, таблица маршрутизации управляет взаимодействием подсетей или определяет набор маршрутов, по которым следует направлять сетевой трафик. Каждая вновь предоставляемая подсеть приходит с уже заполненной по

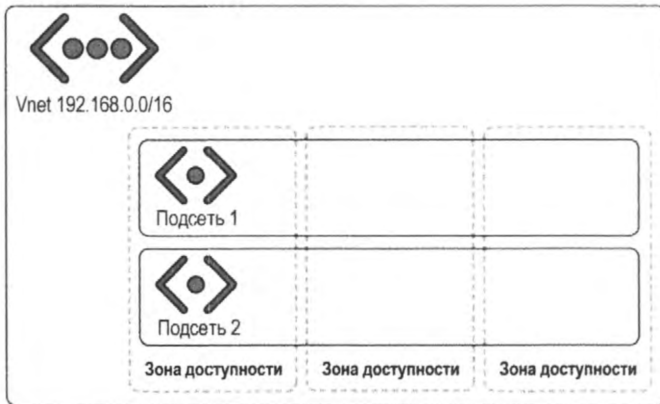


Рис. 6.19. Подсети в зонах доступности

умолчанию таблицей маршрутизации, содержащей некоторые системные маршруты. Эти маршруты не могут быть удалены или изменены. Системные маршруты включают в себя маршрут к сети Vnet, подразделением которой является данная подсеть, маршруты на 10.0.0.0/8 и 192.168.0.0/16, по умолчанию устанавливаемые как идущие в никуда, и самое важное — маршрут по умолчанию в Интернет. Этот маршрут позволяет любому вновь предоставленному ресурсу с IP-адресом AZure сразу же получать доступ к Интернету. Маршрут в Интернет, предоставляемый по умолчанию, является принципиальным различием между Azure и другими облачными провайдерами; это предъявляет также дополнительные требования по обеспечению безопасности сетей Vnet.

На рис. 6.20 показана стандартная таблица маршрутизации для вновь предоставленного объекта AKS. Присутствуют маршруты к пулу агентов с их CIDR, а также IP-адрес следующего перехода. Данный адрес является маршрутом, который таблица задает для пути, а устройство следующего перехода устанавливается как виртуальное, в данном случае это балансировщик нагрузки. Системные маршруты по умолчанию в таблице не показываются, хотя они присутствуют в конфигурации. Понимание сетевого поведения AZure весьма важно с точки зрения обеспечения

aks-agentpool-55103786-routetable | Routes
Route table

Search (Cmd+/)
Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
Settings
Configuration
Routes
Subnets

+ Add

Search routes

Name	Address prefix	Next hop type	Next hop IP address
aks-agentpool-55103786-vmss000000	10.244.0.0/24	Virtual appliance	10.240.0.4
aks-agentpool-55103786-vmss000001	10.244.2.0/24	Virtual appliance	10.240.0.5
aks-agentpool-55103786-vmss000002	10.244.1.0/24	Virtual appliance	10.240.0.6

Рис. 6.20. Таблица маршрутизации

безопасности коммуникаций, а также с точки зрения планирования и устранения ошибок.

Некоторые системные маршруты, называемые опциональными маршрутами по умолчанию, работают только в случаях, когда разрешаются определенные действия — например, пиринг в сети Vnet. Этот пиринг позволяет сетям Vnet независимо от их расположения в мире устанавливать взаимодействие между собой, пользуясь для этого глобальной инфраструктурой Azure.

Таблицы маршрутизации могут содержать маршруты, определяемые пользователем, которые создаются также протоколом граничного шлюза (Border Gateway Protocol). Пользовательские маршруты чрезвычайно важны, поскольку с их помощью сетевые администраторы задают маршруты помимо тех, которые устанавливаются Azure по умолчанию, например маршруты к прокси-серверам или брандмауэрам. Пользовательские маршруты могут затрагивать маршруты по умолчанию. Хотя маршруты по умолчанию неизменяемы, но, установив пользовательский маршрут с высоким приоритетом, вы сможете отклонить маршрут по умолчанию. В качестве примера приведем использование пользовательского маршрута для направления трафика, назначенного в Интернет, сначала на виртуальный брандмауэр, а не напрямую в Интернет. На рис. 6.21 показан пользовательский маршрут, называемый Google, с типом следующего перехода, заданным как Интернет. Если приоритеты выставлены корректно, то данный пользовательский маршрут будет посылать трафик в Интернет, минуя установленный по умолчанию системный маршрут, даже если другое правило будет перенаправлять оставшийся интернет-трафик.

aks-agentpool-55103786-routetable | Routes
Route table

Search (Cmd+/)
Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
Settings
Configuration
Routes
Subnets

+ Add

Search routes

Name	Address prefix	Next hop type	Next hop IP address
aks-agentpool-55103786-vmss000000	10.244.0.0/24	Virtual appliance	10.240.0.4
aks-agentpool-55103786-vmss000001	10.244.2.0/24	Virtual appliance	10.240.0.5
aks-agentpool-55103786-vmss000002	10.244.1.0/24	Virtual appliance	10.240.0.6
Google	8.8.8.8/32	Internet	-

Рис. 6.21. Таблица маршрутизации с маршрутом, определяемым пользователем

Таблицы маршрутизации могут создаваться отдельно, а затем использоваться для конфигурирования подсети. Это удобно, если надо сохранить одну таблицу для нескольких подсетей, особенно в случаях, когда есть много пользовательских маршрутов. Подсеть может иметь только одну таблицу маршрутизации, но одна таблица может быть связана с несколькими подсетями. Правила конфигурирования таблицы маршрутизации с пользовательскими маршрутами и таблицы, создаваемой как часть конфигурируемой по умолчанию подсети, одинаковы. В них входят те же

самые системные маршруты по умолчанию, и при необходимости они будут обновляться с использованием тех же опциональных маршрутов по умолчанию.

В настоящее время большинство маршрутов в таблице маршрутизации используют IP-адрес в качестве адреса источника, однако Azur уже начал внедрение концепции, в которой источником служит метка сервиса. Метка сервиса — это строка, за которой скрывается набор IP-адресов сервисов внутри бэкенда Azur, например строка `SQL.EastUs`, представляющая собой метку сервиса, включающую в себя диапазон IP-адресов для платформы Microsoft SQL, доступной на востоке США. С помощью новой концепции будет возможно задать маршрут между одним сервисом Azure, например AzureDevOp, в качестве источника и другим сервисом, например Azure AppService, в качестве назначения без задания IP-адресов для каждого из них.

Документация по Azure содержит список имеющихся меток сервисов.

Публичные и частные IP-адреса

Azure предоставляет IP-адреса как самостоятельный ресурс, т. е. пользователь может создать публичный или частный IP-адрес, не связывая его с чем-либо. Такие адреса могут быть отнесены к группе ресурсов, из которой они будут браться для последующих назначений. Это надо иметь в виду, если вы хотите иметь масштабируемый AKS-кластер, т. е. вы должны обеспечить резервирование достаточного количества частных IP-адресов для подов, если в качестве сетевого интерфейса буде использоваться Azure CNI. Работу Azure CNI мы рассмотрим немного позднее.

IP-адреса ресурсов, как частные, так и публичные, определяются как динамические или статические. Статический IP-адрес задается неизменяемым, в то время как динамический IP-адрес может меняться, если он не назначен на ресурс, такой как виртуальная машина или AKS-под.

Группы сетевой безопасности

Группы сетевой безопасности (NSG) используются для конфигурирования сетей Vnet, подсетей и карт сетевых интерфейсов (NIC) путем установки правил для входящего и выходящего трафика. Трафик фильтруется в соответствии с правилами, в результате принимается решение, пропустить ли его дальше или сбросить. Правила для групп сетевой безопасности являются весьма гибкими, они могут учитывать IP-адреса источника и назначения, сетевые порты и сетевые протоколы. В одном правиле могут быть объединены условия, касающиеся нескольких компонентов, также одно правило может входить в несколько групп.

Фильтрации трафика может осуществляться на базе следующих компонентов:

Приоритет

Число в диапазоне от 100 до 4096. Низшие номера обрабатываются в первую очередь, и используется правило, в котором произошло первое выполнение

заданного условия. Как только условие выполнено, остальные правила уже не рассматриваются.

Источник/назначение

Проверяются правила для источника (входящий трафик) и назначения (выходящий трафик). Источник/назначение может быть:

- Индивидуальным IP-адресом.
- Блоком CIDR (например, 10.2.0.0/24).
- Меткой сервиса в Microsoft Azure.
- Группами безопасности для приложений.

Протокол

TCP, UDP, ICMP, ESP, AH или любой.

Направление

Правила для входящего или выходящего трафика.

Диапазон портов

Задается номер порта или диапазон номеров.

Действие

Разрешить или запретить трафик.

На рис. 6.22 показан пример сетевых групп безопасности.

Filter by name | Port == all | Protocol == all | Source == all | Destination == all | Action == all

Priority ↑	Name ↑↓	Port ↑↓	Protocol ↑↓	Source ↑↓	Destination ↑↓	Action ↑↓
∨ Inbound Security Rules						
65000	AllowVnetInBound	Any	Any	VirtualNetwork	VirtualNetwork	Allow
65001	AllowAzureLoadBalance···	Any	Any	AzureLoadBalancer	Any	Allow
65500	DenyAllInBound	Any	Any	Any	Any	Deny
∨ Outbound Security Rules						
65000	AllowVnetOutBound	Any	Any	VirtualNetwork	VirtualNetwork	Allow
65001	AllowInternetOutBound	Any	Any	Any	Internet	Allow
65500	DenyAllOutBound	Any	Any	Any	Any	Deny

Рис. 6.22. Сетевая группа безопасности в Azure

При конфигурировании групп сетевой безопасности в Azure надо учитывать следующее. Во-первых, не могут существовать два или более правил с одним и тем же приоритетом и направлением. Может совпадать приоритет или направление, но не оба параметра сразу. Во-вторых, диапазон портов может использоваться, только если развертывание было произведено с помощью Resource Manager, а не по классической модели. Это ограничение касается также диапазона IP-адресов и меток сервиса для параметра источник/назначение. В-третьих, при задании IP-адреса для ресурса Azure в параметре источник/назначение, если ресурс имеет как публичный,

так и частный IP-адрес, то используйте частный. Azure выполняет преобразование IP-адресов из публичных в частные вне процесса обработки, поэтому частный IP-адрес во время обработки будет как раз к месту.

Взаимодействие вне пределов виртуальной сети

Описанные выше концепции относились в основном к взаимодействиям в пределах одной сети Vnet Azure. Этот тип коммуникации очень важен для Azure, но является далеко не единственным. Большинство приложений в Azure требуют взаимодействия с другими сетями вне рамок Vnet, включая сюда локальные сети предприятий, другие виртуальные сети Azure и Интернет. Способы этих взаимодействий во многом схожи с внутренними сетевыми взаимодействиями и используют те же ресурсы, хотя и с некоторыми отличиями. В данном разделе мы рассмотрим эти отличия.

Пиринг между сетями Vnet может связать сети Vnet в разных регионах, но есть ограничения на определенные сервисы, например балансировщики нагрузки.

Полный список ограничений см. в документации по Azure.

Доступ в Интернет вне рамок Azure работает с другим набором ресурсов. Как уже было сказано ранее, публичные IP-адреса могут быть созданы и присвоены ресурсу в Azure. Ресурс использует свой частный IP-адрес для коммуникации внутри Azure. Когда трафик от ресурса выходит из внутренней сети в Интернет, то Azure преобразует частный IP-адрес в присвоенный ресурсу публичный IP-адрес. После этого трафик выходит в Интернет. Публичный адрес ресурса Azure, являющийся назначением для входящего трафика, преобразуется на границе сети Vnet в частный адрес данного ресурса, и именно этот адрес используется все остальное время, пока трафик идет до места своего назначения. Такой маршрут и является причиной, почему в подсетях правила, например, для групп сетевой безопасности формулируются на базе частных IP-адресов.

Преобразование сетевых адресов (NAT) тоже может быть сконфигурировано на подсети. В этом случае ресурсам подсети, где активировано NAT, не требуется публичный IP-адрес для доступа в Интернет. NAT активируется в подсети, чтобы сделать возможным выходящий трафик в Интернет с публичным IP-адресом из пула предоставленных IP-адресов. NAT позволяет ресурсам направлять трафик в Интернет для выполнения запросов, таких как обновления или установки, и возвращать ответный трафик, но при этом препятствует доступу к этим ресурсам через Интернет. Важно отметить, что если NAT сконфигурировано, то оно получает приоритет над всеми остальными правилами для выходящего трафика и заменяет в подсети все установленные по умолчанию назначения в Интернете. По умолчанию NAT также использует преобразование адресов портов (PAT).

Балансировщик нагрузки в Azure

Теперь, когда вы можете посылать трафик из вашей сети и получать его из внешних сетей на вашу Vnet, нужны средства, которые будут обеспечивать устойчивую

работу этих линий коммуникации. Балансировщики нагрузки, предлагаемые Azure, при обработке запроса стараются распределить трафик по пулу ресурсов, а не передать его на единственный ресурс. В Azure есть два основных типа балансировщиков: стандартный и шлюз приложения.

Стандартный балансировщик Azure — это система уровня 4, которая распределяют входящий трафик на основе протоколов 4-го уровня, таких как TCP и UDP, т. е. трафик маршрутизируется на базе IP-адреса и порта. Такие балансировщики занимаются фильтрацией трафика, поступающего из Интернета, но они также могут направлять трафик от одного из ресурсов Azure на набор его других ресурсов. Стандартный балансировщик использует модель сети с нулевым доверием. Данная модель требует наличия сетевых групп безопасности, на основе которых происходит «открытие» трафика на балансировщик. Если соответствующая группа безопасности не разрешает трафик, то балансировщик нагрузки и не будет пытаться его маршрутизировать.

Шлюз приложения Azure работает аналогично стандартному балансировщику в том, что он также фильтрует входящий трафик, но, в отличие от последнего, он делает это на уровне 7. Фильтрация входящих HTTP-запросов производится на основе URI или хостовых заголовков. Шлюзы приложений также могут использоваться в качестве брандмауэров для веб-приложений, чтобы повысить защищенность трафика. Кроме того, шлюз приложения может работать как ингресс-контроллер для кластеров AKS.

Балансировщики нагрузки, как стандартные, так и шлюзы приложений, имеют общие базовые параметры:

IP-адрес фронтенда

Публичный или частный, в зависимости от ситуации этот адрес используется для назначения балансировщика нагрузка или, более широко, ресурсов бэкенда, которые он балансирует.

SKU

Как и для других ресурсов Azure, параметр определяет «тип» балансировщика нагрузки и соответственно имеющиеся конфигурационные опции.

Пул бэкендов

Это набор ресурсов, на которые балансировщик направляет трафик, например виртуальные машины или поды внутри кластера AKS.

Тесты на работоспособность

Методы, используемые балансировщиком нагрузки, чтобы убедиться в доступности ресурсов бэкенда для трафика, например проверка конечной точки, которая возвращает состояние OK:

Слушающий компонент

Конфигурация, сообщающая балансировщику, какого типа трафик ожидать, например HTTP-запросы.

Правила

Задают, каким образом направлять входящий трафик на слушающий компонент.

На рис. 6.23 показаны некоторые из базовых компонентов архитектуры балансировщика нагрузки Azure. Трафик попадает на балансировщик и посылается на слушающее устройство, чтобы определить, происходит ли балансировка трафика. Затем трафик анализируется на базе правил и в зависимости от результата посылается на пул бэкендов. Ресурсы пула, прошедшие тесты на работоспособность, производят обработку трафика.

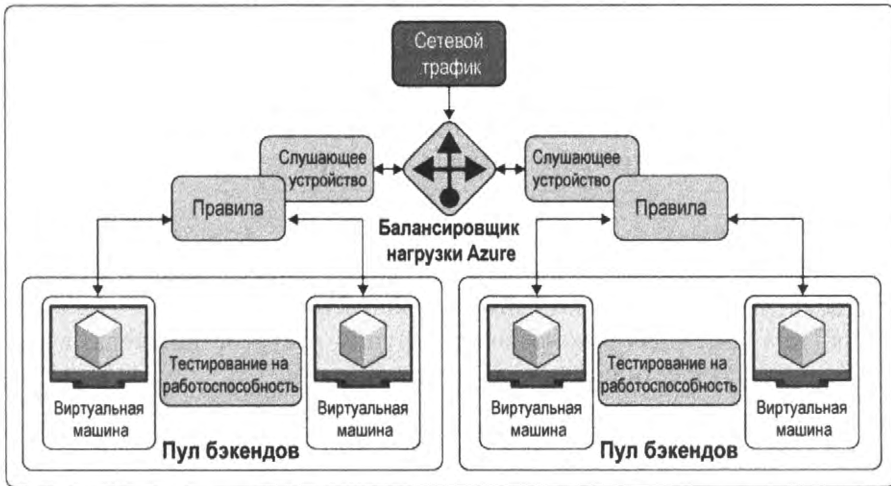


Рис. 6.23. Компоненты балансировщика нагрузки в Azure

На рис. 6.24 показано использование балансировщика нагрузки в AKS.

Теперь, когда мы познакомились с основами сетей Azure, мы можем перейти к рассмотрению, каким образом это используется в сервисе Azure для Kubernetes, Azure Kubernetes Service (AKS).

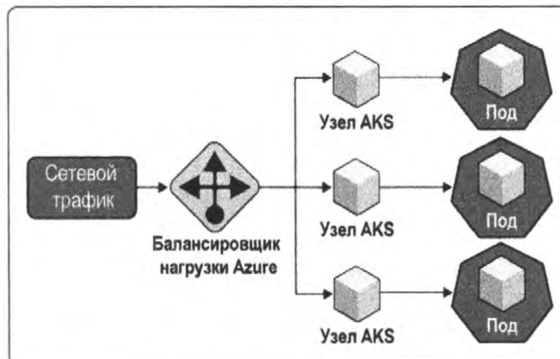


Рис. 6.24. Балансировка нагрузки в кластере AKS

Azure Kubernetes Service

Подобно другим облачным провайдерам, Microsoft оценил большие возможности Kubernetes и предложил свой продукт Azure Kubernetes Service, интегрирующий Azure с Kubernetes. AKS — это аутсорсинговый сервис от Azure, выполняющий значительную часть работы по управлению приложениями Kubernetes. Azure берет на себя функции поддержания и тестирования работоспособности, тем самым оставляя разработчикам больше времени на решение более сложных задач в рамках их решений на базе Kubernetes.

AKS позволяет создавать кластеры и управлять ими на основе интерфейсов Azure CLI, таких как Azure PowerShell и Azure Portal, а также других систем развертывания на основе шаблонов типа ARM Templates и Terraform от HashiCorp. Используя AKS, Azure управляет компонентами Kubernetes masters, так что пользователю остается только работа с процессами, запущенными на узлах. Это позволяет Azure предлагать ядро AKS в качестве бесплатного сервиса, а оплату требовать за программы на узлах и периферийные сервисы типа хранения данных и сетевая поддержка.

Интерфейс Azure Portal дает возможность управлять и конфигурировать среду AKS. На рис. 6.25 показана информационная страница, на которой перечислены компоненты среды AKS. Помимо данных, приведены также ссылки на важные ресурсы. В разделе Essentials можно видеть группу ресурсов кластера, адрес DNS, версию Kubernetes, тип сети и ссылку на пулы узлов.

Connect Delete Refresh

Essentials JSON View

Resource group (change) : tjbakstest
Status : Succeeded
Location : West US 2
Subscription (change) : tjb_azure_test_2
Subscription ID : 7a0e265a-c0e4-4081-8d76-aafbca9db45e
Tags (change) : createdby : tjb

Kubernetes version : 1.18.14
API server address : tjbakstest-dns-3e15111c.hcp.westus2.azmk8s.io
Network type (plugin) : Kubenet
Node pools : 1 node pool

Properties Capabilities

Kubernetes services
Kubernetes version 1.18.14
Azure AD integration Not enabled

Node pools
Node pools 1 node pool
Kubernetes versions 1.18.14
Node sizes Standard_B2s
Virtual node pools Not enabled

Networking
API server address tjbakstest-dns-3e15111c.hcp.westus2.azmk8s.io
Network type (plugin) Kubenet
Private cluster Not enabled
Pod CIDR 10.244.0.0/16
Service CIDR 10.0.0.0/16
DNS service IP 10.0.0.10
Docker bridge CIDR 172.17.0.1/16
HTTP application routing Not enabled

Рис. 6.25. Информационная страница Azure Portal для AKS

На рис. 6.26 отдельно представлен раздел Properties информационной страницы, где пользователь может найти дополнительную информацию и ссылки на соответствующие компоненты. Большинство данных совпадают с теми, что показаны в раз-

деле Essentials. Однако здесь можно также увидеть CIDR различных подсетей для таких компонентов, как мост Docker и подсеть пода.

Поды Kubernetes, создаваемые AKS, связываются с виртуальными сетями и могут получать доступ к сетевым ресурсам через абстракцию. Эту абстракцию создает kube-proxy, запускаемый на каждом узле; благодаря данному компоненту становится возможным входящий и выходящий трафик. Кроме того, AKS стремится еще больше упростить управление Kubernetes путем автоматизации изменений, вызванных изменениями в виртуальных сетях. Когда происходят соответствующие изменения, сетевые сервисы в AKS автоматически переконфигурируются. Например, открытие сетевого порта на поде активирует обновление связанных с подом групп сетевой безопасности, чтобы открыть эти порты для трафика.

Properties Capabilities

Kubernetes services

Kubernetes version	1.18.14
Azure AD integration	Not enabled

Node pools

Node pools	1 node pool
Kubernetes versions	1.18.14
Node sizes	Standard_B2s
Virtual node pools	Not enabled

Networking

API server address	tjbakstest-dns-3e15111c.hcp.westus2.azmk8s.io
Network type (plugin)	Kubenet
Private cluster	Not enabled
Pod CIDR	10.244.0.0/16
Service CIDR	10.0.0.0/16
DNS service IP	10.0.0.10
Docker bridge CIDR	172.17.0.1/16
HTTP application routing	Not enabled

Integrations

Container insights	Not enabled
Workspace resource ID	N/A

Рис. 6.26. Свойства AKS, показанные в Azure Portal

По умолчанию AKS создает запись в Azure DNS, имеющую публичный IP-адрес. Однако сетевые правила, задаваемые по умолчанию, не разрешают допуск с публичного адреса. Кластер, создаваемый в частном режиме, не использует публичные IP-адреса и блокирует доступ с них. В таком режиме доступ к кластеру будет возможен только из виртуальной сети Vnet. По умолчанию стандартный SKU создаст балансировщик нагрузки AKS. Такая конфигурация может быть изменена при развертывании, если развертывание осуществляется с помощью CLI. Ресурсы, не включенные в кластер, помещаются в отдельную, автоматически создаваемую группу ресурсов.

При использовании сетевой модели kubenet в AKS действуют следующие правила:

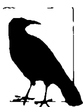
- Узлы получают IP-адреса от подсети виртуальной сети Azure.
- Поды получают IP-адреса в адресном пространстве, логически отличном от адресного пространства узлов.
- IP-адрес источника трафика переключается на первичный адрес узла.
- Преобразование сетевых адресов для подов конфигурируется так, чтобы получить доступ к ресурсам Azure через сеть Vnet.

Важно отметить, что только узлы получают маршрутизируемые IP-адреса, у подов таких адресов нет.

Хотя использование Kubenet в AKS сильно упрощает работу с сетью в Kubernetes, это не единственный способ сетевого взаимодействия. Как и другие облачные провайдеры, Azure для управления инфраструктурой Kubernetes предоставляет интерфейс CNI. В следующем разделе мы рассмотрим плагин CNI для Azure.

Плагин CNI для Azure

Microsoft предоставляет свой собственный CNI-плагин для Azure и AKS — Azure CNI. Первым существенным различием между ним и kubenet является то, что поды получают маршрутизируемый IP-адрес, что обеспечивает к ним прямой доступ. Это различие влечет за собой необходимость в планировании пространства IP-адресов. Каждый узел имеет максимальное число подов, которое может использовать, и для этого использования резервируется достаточно большое число IP-адресов.



Более подробная информация по Azure Container Networking (контейнерные сети в Azure) находится на GitHub.

Используя Azure CNI, трафик внутри сети Vnet адресуется не на IP-адреса узлов, а непосредственно на IP-адреса подов, как показано на рис. 6.27. Внешний трафик, например в Интернет, все еще адресуется на IP-адреса узлов. Azure CNI управляет адресацией бэкендов и направлением к ним трафика, поскольку все ресурсы в одной и той же сети Azure Vnet по умолчанию могут взаимодействовать друг с другом.

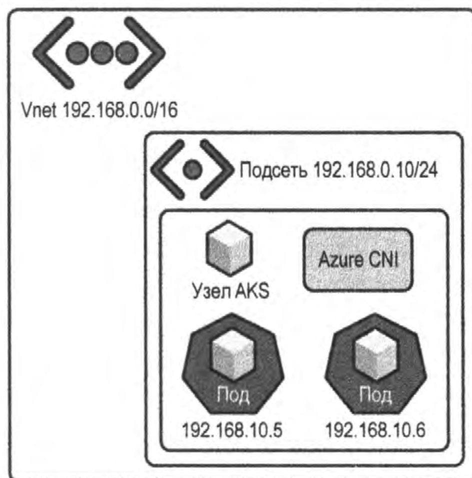


Рис. 6.27. CNI-плагин для Azure

Azure CNI может применяться для развертывания приложений Kubernetes вне AKS. Хотя при этом приходится выполнять дополнительную работу, которую обычно берет на себя Azure, однако вы получаете возможность объединить сетевые воз-

можности Azure и другие ресурсы при сохранении контроля над нестандартными аспектами работы Kubernetes с использованием AKS.

Azure CNI предоставляет дополнительное преимущество, делая возможным разделение задач при сохранении инфраструктуры AKS. Для сетевых ресурсов Azure CNI создает отдельную группу ресурсов. Такое разделение ресурсов позволяет лучше контролировать доступ к ресурсам на уровне групп, когда развертывание осуществляется по модели Azure Resource Management. Различные команды пользователей могут получать доступ к одним компонентам AKS, например сетям, без необходимости иметь доступ к другим, например развернутым приложениям.

Azure CNI — это не единственный способ использовать сервисы Azure с целью улучшить сетевую инфраструктуру Kubernetes. В следующем разделе мы обсудим применение шлюза приложений в Azure как средства контроля входного трафика в вашем кластере Kubernetes.

Ингресс-контроллер для шлюза приложения

Azure позволяет создать щлюз приложения, развернутого внутри кластера AKS, который будет служить как ингресс-контроллер (AGIC). При такой модели развертывания пропадает необходимость иметь вторичный балансировщик нагрузки вне инфраструктуры AKS, что уменьшает операционные издержки и вероятность отказов. AGIC разворачивает свои поды в кластере. Он также осуществляет мониторинг других компонентов кластера на предмет изменений в конфигурации. Если фиксируется какое-либо изменение, то AGIC обновляет шаблон Azure Resource Management (ARM), предназначенный для конфигурирования балансировщика нагрузки, а затем применяет эту обновленную конфигурацию (см. рис. 6.28).

Рис. 6.28. Ингресс-контроллер для шлюза приложения (AGIC) в Azure

В AKS существуют SKU-ограничения на использование AGIC, заключающиеся в поддержке только `Standard_v2` и `WAF_v2`, однако данные SKU имеют масштабируемые функции. Такие формы ингресса применяются в случаях, где есть потребность

в высокой масштабируемости. В качестве опций для развертывания AGIC Microsoft поддерживает как Helm, так и дополнительные функции AKS. Между этими опциями имеются существенные различия:

- Параметры развертывания в Helm не могут редактироваться с использованием дополнений AKS.
- Helm поддерживает конфигурацию Prohibited Target (запрещенные назначения). Ингресс-контроллер AGIC может конфигурировать шлюз приложения с назначениями только на инстансы AKS, не затрагивая каких-либо иных компонентов бэкендов.
- Добавления AKS, будучи управляемым сервисом, автоматически обновляются до последней и более безопасной версии. Развертывания через Helm требуют ручного обновления.

Даже если AGIC сконфигурирован как ингресс-ресурс Kubernetes, он все равно имеет все преимущества кластерного шлюза приложения. Службы шлюза приложения, например завершение по TLS, маршрутизация по URL и брандмауэр для веб-приложений предоставляются в кластере как часть функций AGIC.

В то время как базовая поддержка Kubernetes и сетевого взаимодействия у большинства облачных провайдеров примерно одинакова, Azure предлагает свой собственный подход к Kubernetes на основе распределения ресурсов и управления, особенно удобного для использования на уровне предприятия. Требуется ли вам единственный кластер с базовыми установками и kubenet или масштабируемое развертывание с современными сетевыми решениями, использующими балансировщики нагрузки и шлюзы приложений, — в любом случае сервис Microsoft Azure Kubernetes Service поможет вам сформировать надежную и управляемую инфраструктуру Kubernetes.

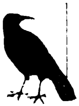
Развертывание приложения с помощью Azure Kubernetes Service

Конфигурирование кластера с помощью Azure Kubernetes Service — это базовый навык, необходимый для освоения данного сервиса. В данном разделе мы пройдем все этапы, начиная от конфигурирования кластера до развертывания в нем нашего веб-сервера из *главы 1*. Для выполнения действий мы будем использовать Azure Portal, Azure CLI и `kubectl`.

Прежде чем мы займемся конфигурированием и развертыванием кластера, нам надо обсудить реестр контейнеров Azure (ACR). Реестр контейнеров — это место, где в Azure хранятся образы контейнеров. В нашем примере будем использовать данный реестр в качестве места, где будет находиться образ контейнера, который хотим развернуть. Чтобы передать образ в ACR, вам нужно иметь его в вашем компьютере. После того как образ в компьютере появился, нам нужно подготовить его для ACR.

Сначала надо задать репозиторий ACR, в котором вы хотите хранить образ, и зайти в него через Docker CLI командой `docker login <acr_repository>.azurecr.io`. В данном

примере будем использовать репозиторий ACR с именем tjbakstestcr, так что команда будет docker login tjbakstestcr.azurecr.io. Затем командой <acr_repository>. azurecr.io\<imagetag> присвойте тег локальному образу, который вы хотите передать в ACR. В данном примере возьмем для образа тег aksdemo, т. е. получим tjbakstestcr.azurecr.io/aksdemo. Чтобы присвоить тег образу, используйте команду docker tag <local_image_tag> <act_image_tag>. В нашем примере она будет docker tag aksdemo tjbakstestcr. azurecr.io/aksdemo. Наконец, загрузите образ в реестр командой docker push tjbakstestcr. azurecr.io/aksdemo.



Более подробная информация по Docker и реестру контейнеров в Azure содержится в официальной документации.

После того как образ попал в ACR, останется только установить субъект-службу (Sevice Principal). Проще это сделать перед началом создания кластера AKS, но можно и в процессе. Субъект-служба в Azure — это объект приложения Azure Active Directory. Субъект-службы обычно используются для взаимодействия с Azure через автоматизацию приложений. С помощью субъект-службы мы дадим возможность кластеру загружать образ aksdemo из реестра ACR. То есть субъект-службе потребуется доступ к реестру, где хранится образ. Поэтому нужно будет запомнить ID клиента и секрет субъект-службы, которую вы собираетесь использовать.

Дополнительная информация по Azure Active Directory и субъект-службам содержится в документации.

Теперь, когда у нас есть образ в реестре, клиентский ID и секрет для субъект-службы, мы можем приступить к развертыванию кластера AKS.

Развертывание кластера с помощью Azure Kubernetes Service

Итак, пришло время развернуть наш кластер. Запустим Azure Portal: перейдите на portal.azure.com и зарегистрируйтесь. После этого вы увидите на экране панель с поисковой строкой вверху, которую мы будем использовать для поиска сервисов. В этой строке наберите **kubernetes** и выберите пункт Kubernetes Service из выпадающего меню (см. рис. 6.29).

Мы находимся на странице Azure Kubernetes Service. Используя фильтры и запросы, мы можем просмотреть на ней все развернутые кластеры AKS. Выберем пункт Create (Создать), как показано на рис. 6.30. Появится выпадающее меню, в котором выберем пункт Create a Kubernetes Cluster (Создать кластер Kubernetes).

На следующем шаге зададим свойства кластера на странице Create a Kubernetes Cluster. Сначала заполним раздел Project Details (Параметры проекта), выбрав подписку, куда будет разворачиваться кластер. Для этой цели существует выпадающее меню, облегчающее поиск и выбор. Для нашего примера выберем подписку tjb_azure_test_2, но подойдет и любая другая, к которой у вас будет доступ. Теперь надо выбрать группу ресурсов для нашего кластера. Это может быть уже суще-

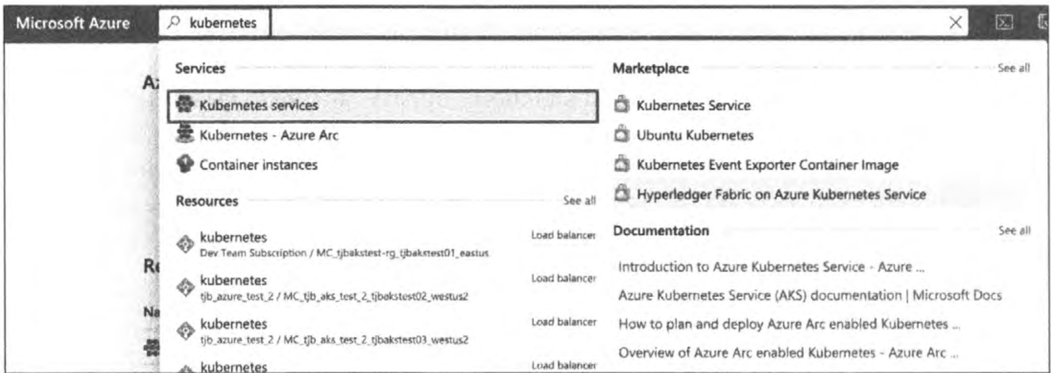


Рис. 6.29. Поиск Kubernetes в Azure Portal

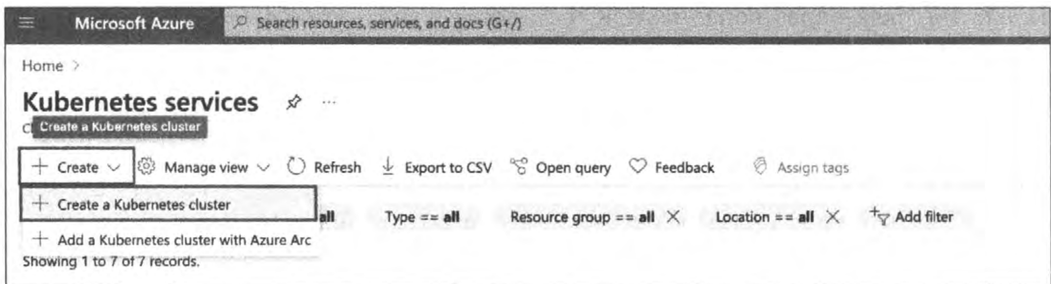
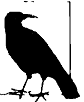


Рис. 6.30. Создание кластера Kubernetes в Azure

ствующая группа или созданная заново. Мы создадим новую группу ресурсов с именем `go-web`.

После того как раздел Project Details заполнен, переходим в раздел Cluster Details (Параметры кластера). Здесь надо задать имя кластера, в нашем случае это будет go-web. Регион, зоны доступности и версия Kubernetes также задаются на данной странице, они имеют предустановленные значения, которые могут быть изменены. Для нашего примера оставим регион по умолчанию «(US) West 2» без зон доступности и версию Kubernetes по умолчанию 1.19.11.



Не все регионы Azure имеют выбираемые зоны доступности. Если зоны доступности являются частью развертываемой архитектуры AKS, то следует использовать соответствующие регионы. Более подробную информацию по регионам AKS можно получить из документации по зонам доступности.

Наконец, мы заполним раздел Primary Node Pool (Пул первичных узлов) на странице Create Kubernetes Cluster, выбрав размер узла и число узлов. Для нашего примера возьмем установленный по умолчанию размер узла DS2 v2 и число узлов 3. Учтите, что AKS имеет определенные ограничения на размер узлов. На рис. 6.31 показаны заполненные нами поля параметров.

Информация по ограничениям AKS, включая и ограничения на размеры узлов, содержится в документации.

Рис. 6.31. Страница создания кластера Kubernetes в Azure

Нажмите кнопку Next: Node Pools (Следующий шаг: пулы узлов), чтобы попасть на страницу Node Pools. На ней можно задать дополнительные пулы узлов для кластера. В нашем случае мы оставим на данной странице установки по умолчанию и перейдем на страницу Authentification (Аутентификация), нажав кнопку Next: Authentification (Следующий шаг: аутентификация) внизу экрана.

Рис. 6.32 показывает страницу Authentification, на которой мы зададим метод аутентификации, используемый кластером для связи с присоединенными сервисами Azure типа ACR, рассмотренного выше в данной главе. По умолчанию метод аутентификации установлен как System-Assigned Managed Identity (Назначаемая системой управляемая идентичность), но мы выберем кнопку Service Principal (Субъект-служба).

Если вы еще не создали субъект-службу, как было показано в начале раздела, то вы можете сделать это сейчас. При этом вам надо будет вернуться назад и дать субъект-службе разрешения на доступ к реестру ACR. Но поскольку мы используем созданную ранее субъект-службу, то жмем на ссылку Configure Service Principal (Конфигурирование субъект-службы) и вводим клиентский ID и секрет.

Все остальные параметры оставим предустановленными. Для завершения создания кластера AKS нажмем кнопку Review + create (Обзор + создание). Это приведет

Рис. 6.32. Страница Authentification в Kubernetes Azure

нас на проверочную страницу. Как показано на рис. 6.33, если все установлено правильно, то вверху экрана появляется сообщение Validation Passed (Проверка прошла успешно). Если же что-то сконфигурировано с ошибками, то появляется Validation Failed (Проверка не пройдена). После того как проверка успешно завершена, мы еще раз посмотрим на все установки и нажмем Create (Создать).

В каком состоянии находится развертывание, покажет колокольчик вверху страницы Azure. Рис. 6.34 показывает, что развертывание нашего кластера еще продолжается. На данной странице также находится информация, которая может быть полезна для поиска ошибок — если они произойдут, — такая как имя развертывания, время начала и Correlation ID.

Как показано на рис. 6.35, развертывание нашего кластера произошло без ошибок. Теперь нам надо установить связь с кластером и сконфигурировать его для использования совместно с нашим веб-сервером.

Рис. 6.33. Проверочная страница Kubernetes Azure

Рис. 6.34. Развертывание кластера Kubernetes Azure в процессе

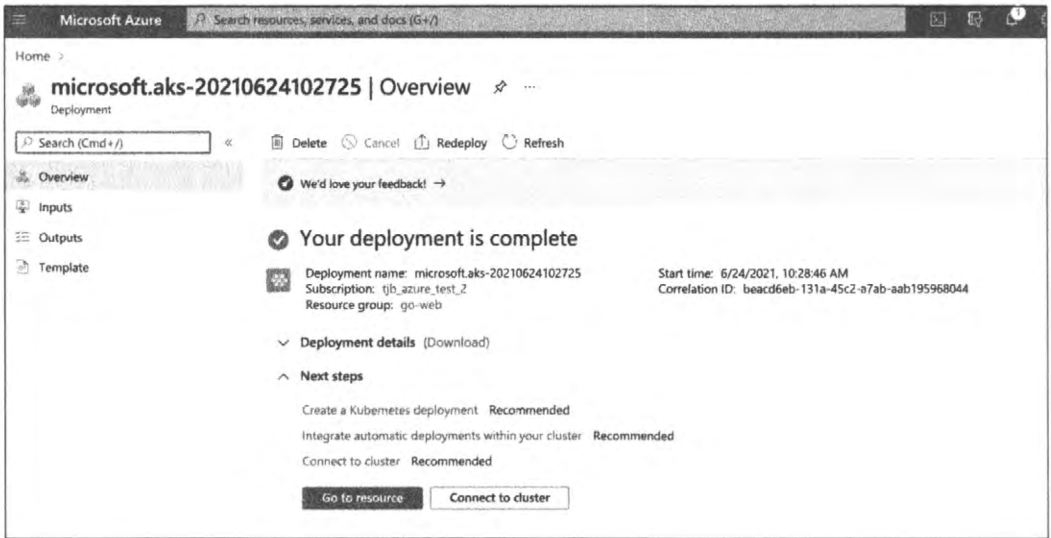


Рис. 6.35. Развертывание кластера Kubernetes Azure завершено

Соединение с кластером AKS и его конфигурирование

В дальнейшем мы будем работать с экземпляром go-web кластера AKS из командной строки. Для управления кластерами из командной строки используется преимущественно команда kubectl. Для установки kubectl интерфейс Azure CLI предлагает простую команду az aks install-cli. Однако прежде чем использовать kubectl, нам нужно получить доступ к кластеру. Для этого существует команда az aks get-credentials -resource-group<resource_group_name> --name <aks_cluster_name>. В нашем случае выполняем az aks get-credentials -resource-group go-web -name go-web, чтобы получить доступ к go-web кластеру в группе ресурсов go-web.

Далее мы установим связь с реестром контейнеров Azure, в котором находится наш образ aksdemo. Командой az aks update -n <aks_cluster_name> -g <cluster_resource_group_name> --attach-acr <acr_repo_name> устанавливается соединение существующего кластера AKS с именованным репозиторием ACR. В нашем случае команда будет az aks update -n tjbakstest -g tjbakstest --attach-acr tjbakstestcr. Команда выполняется какое-то время, а затем появляется выдача, показанная в примере 6.1.

Пример 6.1. Выдача после команды, устанавливающей связь с реестром ACR

```
{- Finished ..
  "aadProfile": null,
  "addonProfiles": {
     "azurepolicy": {
        "config": null,
        "enabled": false,
        "identity": null
  },
```

```
"httpApplicationRouting": {
  "config": null,
  "enabled": false,
  "identity": null
},
"omsAgent": {
  "config": {
     "logAnalyticsWorkspaceResourceID":
     "/subscriptions/7a0e265a-c0e4-4081-8d76-aafbca9db45e/
     resourcegroups/defaultresourcegroup-wus2/providers/
     microsoft.operationalinsights/
     workspaces/defaultworkspace-7a0e265a-c0e4-4081-8d76-aafbca9db45e-wus2"
},
    "enabled": true,
    "identity": null
  }
},
"agentPoolProfiles": [
{
"availabilityZones": null,
"count": 3,
"enableAutoScaling": false,
"enableNodePublicIp": null,
"maxCount": null,
"maxPods": 110,
"minCount": null,
"mode": "System",
"name": "agentpool",
"nodeImageVersion": "AKSUbuntu-1804gen2containerd-2021.06.02",
"nodeLabels": {},
"nodeTaints": null,
"orchestratorVersion": "1.19.11",
"osDiskSizeGb": 128,
"osDiskType": "Managed",
"osType": "Linux",
"powerState": {
   "code": "Running"
},
"provisioningState": "Succeeded",
"proximityPlacementGroupId": null,
"scaleSetEvictionPolicy": null,
"scaleSetPriority": null,
"spotMaxPrice": null,
"tags": null,
"type": "VirtualMachineScaleSets",
"upgradeSettings": null,
"vmSize": "Standard_DS2_v2",
```

```
"vnetSubnetId": null
   }
],
"apiServerAccessProfile": {
   "authorizedIpRanges": null,
   "enablePrivateCluster": false
},
"autoScalerProfile": null,
"diskEncryptionSetId": null,
"dnsPrefix": "go-web-dns",
"enablePodSecurityPolicy": null,
"enableRbac": true,
"fqdn": "go-web-dns-a59354e4.hcp.westus.azmk8s.io",
"id":
"/subscriptions/7a0e265a-c0e4-4081-8d76-aafbca9db45e/
resourcegroups/go-web/providers/Microsoft.ContainerService/managedClusters/go-web",
"identity": null,
"identityProfile": null,
"kubernetesVersion": "1.19.11",
"linuxProfile": null,
"location": "westus",
"maxAgentPools": 100,
"name": "go-web",
"networkProfile": {
   "dnsServiceIp": "10.0.0.10",
   "dockerBridgeCidr": "172.17.0.1/16",
   "loadBalancerProfile": {
     "allocatedOutboundPorts": null,
     "effectiveOutboundIps": [
       {
"id":
"/subscriptions/7a0e265a-c0e4-4081-8d76-aafbca9db45e/
resourceGroups/MC_go-web_go-web_westus/providers/Microsoft.Network/
publicIPAddresses/eb67f61d-7370-4a38-a237-a95e9393b294",
"resourceGroup": "MC_go-web_go-web_westus"
}
],
"idleTimeoutInMinutes": null,
"managedOutboundIps": {
  "count": 1
},
"outboundIpPrefixes": null,
"outboundIps": null
  },
"loadBalancerSku": "Standard",
"networkMode": null,
"networkPlugin": "kubenet",
```

```
"networkPolicy": null,
"outboundType": "loadBalancer",
"podCidr": "10.244.0.0/16",
"serviceCidr": "10.0.0.0/16"
},
"nodeResourceGroup": "MC_go-web_go-web_westus",
"powerState": {
  "code": "Running"
},
"privateFqdn": null,
"provisioningState": "Succeeded",
"resourceGroup": "go-web",
"servicePrincipalProfile": {
  "clientId": "bbd3ac10-5c0c-4084-a1b8-39dd1097ec1c"
  "secret": null
},
"sku": {
"name": "Basic",
"tier": "Free"
},
"tags": {
"createdby": "tjb"
},
"type": "Microsoft.ContainerService/ManagedClusters",
"windowsProfile": null
}
```


Данная выдача содержит полную информацию по кластеру, что означает, что мы успешно установили с ним связь. Теперь, когда доступ к кластеру существует и установлено соединение с реестром, мы можем перейти к развертыванию в кластере нашего веб-сервера.

Развертывание веб-сервера

Разворачиваем программу, показанную в примере 6.2. Как было сказано ранее, мы встроили код программы в образ Docker и сохранили его в реестре ACR в репозитории `tjbakstestcr`. Для развертывания приложения будем использовать YAML-файл.

Пример 6.2. Спецификация Kubernetes PodSpec для минимального веб-сервера

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: go-web
```

```
spec:
   containers:
   - name: go-web
     image: go-web:v0.0.1
     ports:
     - containerPort: 8080
     livenessProbe:
       httpGet:
         path: /healthz
   port: 8080
       initialDelaySeconds: 5
       periodSeconds: 5
     readinessProbe:
  httpGet:
   path: /
   port: 8080
initialDelaySeconds: 5
periodSeconds: 5
```

Разбирая этот YAM-файл, мы видим, что задаются два ресурса AKS: развертывание и сервис. В развертывании создается контейнер с именем `go-web` и порт контейнера 8080. Строка `image: tjbakstestcr.azurecr.io/aksdemo` ссылается на образ `aksdemo` в реестре ACR, поскольку этот образ будет развертываться в контейнере. Сервис тоже создается с именем `go-web`. YAML-файл определяет его как балансировщик нагрузки, слушающий порт 8080 и имеющий в качестве назначения приложение `go-web`.

Теперь нам надо перенести приложение в кластер. Это делается командой `kubectlapply -f <yaml_file_name>.yaml`. Из выдачи мы видим, что создаются два объекта: deployment.apps/go-web и service/go-web. Задав команду `kubectl get pods`, увидим следующее:

```
o   kubectl get pods

NAME                         READY    STATUS     RESTARTS    AGE

go-web-574dd4c94d-2z5lp      1/1      Running      0         5h29m
```

Теперь соединимся с развернутым приложением, чтобы убедиться, что оно корректно работает. Когда создавался кластер AKS с установками по умолчанию, то вместе с ним был развернут балансировщик нагрузки с публичным IP-адресом. Мы могли бы вернуться в портал и найти этот балансировщик и его IP-адрес, но это проще выполнить с помощью kubectl. Команда kubectl get [.keep-together]# service go-web выдает следующую информацию:

```
o   kubectl get service go-web

NAME    TYPE          CLUSTER-IP  EXTERNAL-IP   PORT(S)         AGE
go-web  LoadBalancer 10.0.3.75   13.88.96.117  8080:31728/TCP  21h
```

В данной выдаче мы видим внешний IP-адрес 13.88.96.117. Таким образом, если все установлено корректно, то мы можем связываться с ресурсом 13.88.96.117 на порту 8080 с помощью команды `curl 13.88.96.117:8080`. Как видно из приводимой ниже выдачи, наше развертывание действительно прошло успешно:

```
o → curl 13.88.96.117:8080 -vvv
*   Trying 13.88.96.117...
* TCP_NODELAY set
* Connected to 13.88.96.117 (13.88.96.117) port 8080 (#0)
> GET / HTTP/1.1
> Host: 13.88.96.117:8080
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Date: Fri, 25 Jun 2021 20:12:48 GMT
< Content-Length: 5
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host 13.88.96.117 left intact
Hello* Closing connection 0
```

Переход на веб-браузер и задание в нем `http://13.88.96.117:8080` приведет к тому же результату, как показано на рис. 6.36.

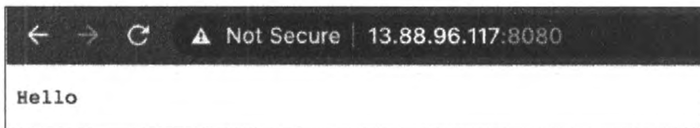


Рис. 6.36. Приложение Hello в Kubernetes Azure

Заключительные замечания по AKS

В данном разделе мы развернули наш веб-сервер, написанный на Go, в кластере, созданном средствами Azure Kubernetes Service. Чтобы сконфигурировать кластер и развернуть в нем наше приложение, мы использовали AzurePortal, интерфейс командной строки az cli и программу kubectl. Образ нашего веб-сервера сохранялся в реестре контейнеров Azure. Для развертывания приложения мы использовали YAML-файл и проверили результат, связываясь с приложением командой cURL и через веб-браузер.

Заключение

Каждый облачный провайдер предлагает свои собственные продукты, касающиеся сетевых сервисов для кластеров Kubernetes. В табл. 6.4 приводятся некоторые характеристики этих продуктов. При выборе поставщика облачных услуг приходится принимать во внимание множество факторов и еще больше — когда речь

идет о выборе платформы для Kubernetes. Целью данной главы было дать информацию администраторам и разработчикам о возможностях, из которых они будут выбирать, когда придется принимать решение о развертывании приложений Kubernetes.

Таблица 6.4. Сводная информация по облачным сетям и Kubernetes

	AWS	Azure	GCP
Виртуальная сеть	VPC	Vnet	VPC
Область действия сети	Регион	Регион	Глобально
Граница подсети	Зона	Регион	Регион
Область маршрутизации	Подсеть	Подсеть	VPC
Контроль безопасности	NACL/SecGroup	Группы сетевой безопасности/ SecGroup приложений	Брандмауэр
IPv6	Да	Да	Нет
Сервис для Kubernetes	eks	aks	gke
Ингресс-контроллер	AWS ALB	Nginx-Ingress	GKE ингресс-контроллер
Облачный CNI	AWS VPC CNI	Azure CNI	GKE CNI
Поддержка балансировки	ALB L7, L4w/NLB, Nginx	Балансировщик L4 Azure, L7 w/Nginx	L7, HTTP(S)
Сетевые политики	Да (Calico/Cilium)	Да (Calico/Cilium)	Да (Calico/Cilium)

Мы рассмотрели множество вопросов — от базовых концепций модели OSI до запуска кластеров в облачных сетях. Администраторам кластеров, сетевым инженерам и разработчикам приложений постоянно приходится выбирать размер подсети, сетевой интерфейс CNI, тип балансировщика, нагрузку — это только некоторые из множества факторов, по которым принимаются решения. Объяснение того, как делать этот выбор и как он может повлиять на работу кластерной сети, составляет основную задачу данной книги. И это только начало вашего пути к эффективному управлению большими кластерами. Мы рассмотрели только сетевые возможности, предоставляемые для кластеров Kubernetes. Хранение информации, вычисления и способы развертывания приложений в кластерах — это технологии, которые вам еще предстоит освоить. В этом вам помогут многочисленные книги библиотеки «O'Reilly», такие как Production Kubernetes (Rosso и др.), из которой вы узнаете, как разрабатывать промышленные приложения с помощью Kubernetes или Hacking Kubernetes (Martin и Hausenblas), как повысить защищенность Kubernetes и тестировать кластеры Kubernetes на слабые места в смысле безопасности.

Мы надеемся, что наши рекомендации помогут вам в выборе сетевых решений. Мы с восхищением смотрим на достижения всех разработчиков Kubernetes и с интересом ждем ваших разработок на основе абстракций, предоставляемых вам Kubernetes.

Об авторах

Джеймс Стронг (James Strong) начал работать с сетями еще в школе, когда посещал занятия в Академии сетей Cisco. Затем последовала работа сетевым инженером в Университете Дейтона и фирме «Дженерал электрик». Будучи участником программы «Дженерал электрик — лидеры информационных технологий», Джеймс Стронг познакомился со многими производственными проблемами, с которыми сталкиваются системные администраторы и разработчики приложений. В качестве директора облака в фирме Contino он помогает многим крупным предприятиям и финансовым организациям в разработке и размещении их облачных решений. Джеймс также вовлечен в деятельность местной облачной «общественности» в рамках сообщества пользователей AWS и пользователей облачных сервисов Луисвилля. Джеймс имеет степень мастера в компьютерных технологиях, полученную в Университете Луисвилля, и обладает шестью сертификатами AWS, включая Certified Advanced Networking Specialty, а также CKA от CNCF.

Валлери Лэнси (Vallery Lancey) является IT-инженером со специализацией в области надежности, инфраструктуры и распределенных систем. Она начала использовать Kubernetes в 2017 г., пройдя через все трудности работы с еще не отработанным и быстро развивающимся продуктом. Валлери разрабатывала платформы на базе Kubernetes и широко использовала Kubernetes в деятельности таких компаний, как Lyft. Она занимается также решениями Kubernetes для сетей SIG. Также она внесла свой вклад в разработку kube-proxy и поддержку двойного сетевого стека IPv4/IPv6.

Об обложке

Птица на обложке книги — это чернозобая гагара (gavia arctica), водоплавающая птица, обитающая в Северном полушарии.

Чернозобая гагара может иметь размер в длину до 70 см и весить от полутора до трех килограммов. Взрослые особи в основном черные с белыми отметинами, белыми полосами и белым брюшком. Гагара обычно откладывает два яйца, которые она высиживает 27–29 дней. Гнездо устраивается прямо на земле, хищники и затопления приводят к тому, что обычно выживает только один птенец.

Хотя общая численность популяции гагар снижается, сохранение вида не считается проблематичным, поскольку популяция все еще очень многочисленна и широко распространена. Отметим, что многие другие представители животного мира, изображаемые на обложках книг издательства «O'Reilly», действительно находятся в опасности, а их существование важно для всего мира.

Иллюстрация на обложке выполнена Сьюзен Томпсон на основе черно-белой гравюры из книги «Птицы Британии».

O'REILLY®

Kubernetes и сети

Kubernetes превратился в важнейший повседневный инструмент для большинства системных администраторов, управляющих сетями и кластерами. Но для эффективной эксплуатации Kubernetes в промышленном масштабе все специалисты должны найти общий язык. Эта книга позволяет четко сориентироваться в различных уровнях сложности и абстракциях, с которыми приходится иметь дело при встраивании Kubernetes в компьютерную сеть.

Авторы помогут вам быстро познакомиться со всеми тонкостями, возникающими при развертывании Kubernetes в больших контейнерных конфигурациях. Если хотите преуспеть в поддержке продакшен-кластера и эффективно устранять в нем неполадки, то нужно уверенно различать абстракции, предоставляемые на каждом из уровней OSI. Как — читайте в этой книге.

У Джеймса и Валлери получилась интересная и практичная книга, в которой доступно излагаются сложные вопросы встраивания Kubernetes в сложившиеся и новые компьютерные сети.

— Брэд Топол,
заслуженный инженер компании IBM, активно пропагандирующий открытые технологии и свободную разработку

- **Изучите сетевую модель Kubernetes**
- **Выберите для ваших кластеров наилучший интерфейс, соответствующий требованиям CNCF**
- **Исследуйте примитивы сетевого уровня и операционной системы Linux, лежащие в основе Kubernetes**
- **Научитесь быстро устранять сетевые проблемы и предотвращать простои**
- **Узнайте, как при помощи Kubernetes организуются и поддерживаются облачные сети**
- **Взвесьте достоинства и недостатки различных сетевых инструментов и научитесь наилучшим образом формировать стек для обслуживания сети**

Джеймс Стронг — директор по облачной интеграции в компании Contino, имеет опыт руководящей работы и консультирования многих крупных предприятий и финансовых институтов, участвует в работе конференций AWS User Group и Cloud-Native Louisville. Ранее работал системным администратором в Дейтонском университете (штат Огайо) и в компании GE Appliances.

Валлери Лэнси — инженер по распределенным системам, ранее отвечала за поддержку мультикластера Kubernetes в компании Lyft, работала ведущим DevOps-инженером в компании CheckFront (Британская Колумбия, Канада).

ISBN 978-5-9775-1855-0

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

bhv®