ADSM

Выпуск 0.1.0

дек. 03, 2022

Оглавление

1 Часть 0. Планирование 1

2 Часть 1. Виртуализация 19

3 Часть 2. Дизайн сети 59

4 Часть 3. IPAM/DCIM-системы 79

5 Часть 4. Архитектура системы автоматизации 119

6 Часть 5. История сетевой автоматизации 143

7 Часть 6. Интерфейсы взаимодействия с сетевым устройством 179

Глава 1

Часть 0. Планирование

СДСМ закончился, а бесконтрольное желание писать - осталось.

Долгие годы наш брат страдал от выполнения рутинной работы, скрещивал пальцы перед коммитом и недосыпал из-за ночных ролбэков.
Но тёмным временам приходит конец.

Этой статьёй я начну серию о том, как *мне* видится автоматизация.
По ходу дела разберёмся с этапами автоматизации, хранением переменных, формализацией дизайна, с RestAPI, NETCONF, YANG, YDK и будем очень много программировать.
Мне означает, что а) это не объективная истина, б) не безоговорочно лучший подход в) мой взгляд даже в ходе движения от первой к последней статье может поменяться - честно говоря, от стадии черновика до публикации я переписывал всё полностью дважды.

АДСМ я попробую вести в формате, немного отличном от СДСМ. По-прежнему будут появляться большие обстоятельные номерные статьи, а между ними я буду публиковать небольшие заметки из повседневного опыта. Постараюсь тут бороться с перфекционизмом и не вылизывать каждую из них.

> Как это забавно, что во второй раз приходится проходить один и тот же путь.
>
> Сначала пришлось писать самому статьи про сети из-за того, что их не было в рунете.
>
> Теперь я не смог найти всесторонний документ, который систематизировал бы подходы к автоматизации и на простых практических примерах разбирал вышеперечисленные технологии.
>
> Возможно, я ошибаюсь, поэтому, кидайте ссылки на годные ресурсы. Впрочем это не изменит моей решимости писать, потому что, основная цель - это всё-таки научиться чему-то самому, а облегчить жизнь ближнему - это приятный бонус, который ласкает ген распространения опыта.

NETWORK

Мы попробуем взять средних размеров дата-центр LAN DC и проработать всю схему автоматизации. Делать некоторые вещи я буду практически впервые вместе с вами.

> В описываемых тут идеях и инструментах я буду не оригинален. У Дмитрия Фиголя есть отличный канал со стримами на эту тему.
>
> Статьи во многих аспектах будут с ними пересекаться.

В LAN DC 4 ДЦ, около 250 коммутаторов, полдюжины маршрутизаторов и пара файрволов.
Не фейсбук, но достаточно для того, чтобы глубоко задуматься об автоматизации.
Бытует, впрочем, мнение, что если у вас больше 1 устройства, уже нужна автоматизация.
На самом деле тяжело представить, что кто-то сейчас может жить без хотя бы пачки наколеночных скриптов.
Хотя я слышал, что есть такие конторы, где учёт IP-адресов ведётся в экселе, а каждое из тысяч сетевых устройств настраивается вручную и имеет свою неповторимую конфигурацию. Это, конечно, можно выдать за современное искусство, но чувства инженера точно будут оскорблены.

1.1 Цели

Сейчас мы поставим максимально абстрактные цели:

- Сеть - как единый организм
- Тестирование конфигурации
- Версионирование состояния сети
- Мониторинг и самовосстановление сервисов

Позже в этой статье разберём какие будем использовать средства, а в следующих и цели и средства в подробностях.

1.1.1 Сеть - как единый организм

Определяющая фраза цикла, хотя на первый взгляд она может показаться не такой уж значительной: **мы будем настраивать сеть, а не отдельные устройства**.
Все последние годы мы наблюдаем сдвиг акцентов к тому, чтобы обращаться с сетью, как с единой сущностью, отсюда и приходящие в нашу жизнь *Software Defined Networking*, *Intent Driven Networks* и *Autonomous Networks*.
Ведь что глобально нужно приложениям от сети: связности между точками А и Б (ну иногда +В-Я) и изоляции от других приложений и пользователей.

И таким образом, наша задача в этой серии - **выстроить систему**, поддерживающую актуальную конфигурацию **всей сети**, которая уже декомпозируется на актуальную конфигурацию на каждом устройстве в соответствии с его ролью и местоположением.
Система управления сетью подразумевает, что для внесения изменений мы обращаемся в неё, а она уже в свою очередь вычисляет нужное состояние для каждого устройства и настраивает его.

Кто мы?
сетевики!
Чего мы хотим?
СВЯЗНОСТИ!
Когда мы хотим?
99,9999

Таким образом мы минимизируем почти до нуля хождение в CLI руками - любые изменения в настройках устройств или дизайне сети должны быть формализованы и документированы - и только потом выкатываться на нужные элементы сети.

> То есть, например, если мы решили, что с этого момента стоечные коммутаторы в Казани должны анонсировать две сети вместо одной, мы
>
> 1. Сначала документируем изменения в системах
> 2. Генерируем целевую конфигурацию всех устройств сети
> 3. Запускаем программу обновления конфигурации сети, которая вычисляет, что нужно удалить на каждом узле, что добавить, и приводит узлы к нужному состоянию.
>
> При этом руками мы вносим изменения только на первом шаге.

1.1.2 Тестирование конфигурации

Известно, что 80% проблем случаются во время изменения конфигурации - косвенное тому свидетельство - то, что в период новогодних каникул обычно всё спокойно.

Я лично был свидетелем десятков глобальных даунтаймов из-за ошибки человека: неправильная команда, не в той ветке конфигурации выполнили, забыли комьюнити, снесли MPLS глобально на маршрутизаторе, настроили пять железок, а на шестой ошибку не заметили, закоммитили старые изменения, сделанные другим человеком. Сценариев тьма тьмущая.

Автоматика нам позволит совершать меньше ошибок, но в большем масштабе. Так можно окирпичить не одно устройство, а всю сеть разом.

Испокон веков наши деды проверяли правильность вносимых изменений острым глазом, стальными яйцами и работоспособностью сети после их выкатки.

Те деды, чьи работы приводили к простою и катастрофическим убыткам, оставляли меньше потомства и должны со временем вымереть, но эволюция процесс медленный, и поэтому до сих пор не все предварительно проверяют изменения в лаборатории.

Однако на острие прогресса те, кто автоматизировал процесс тестирования конфигурации, и дальнейшего её применения на сеть. Иными словами - позаимствовал процедуру CI/CD Continuous Integration, Continuous Deployment.

В одной из частей мы рассмотрим как реализовать это с помощью системы контроля версий, вероятно, гитхаба.

> Как только вы свыкнитесь с мыслью о сетевом CI/CD, в одночасье метод проверки конфигурации путём её применения на рабочую сеть покажется вам раннесредневековым невежеством. Примерно как стучать молотком по боеголовке.

Органическим продолжением идей о **системе** управления сетью и CI/CD становится полноценное версионирование конфигурации.

1.1.3 Версионирование

Мы будем считать, что при любых изменениях, даже самых незначительных, даже на одном незаметном устройстве, вся сеть переходит из одного состояния в другое.

И мы всегда не выполняем команду на устройстве, мы меняем состояние сети.

Вот давайте эти состояния и будем называть версиями?

Допустим, текущая версия - 1.0.0.
Поменялся IP-адрес Loopback-интерфейса на одном из ToR'ов? Это минорная версия - получит номер 1.0.1.
Пересмотрели политики импорта маршрутов в BGP - чуть посерьёзнее - уже 1.1.0
Решили избавиться от IGP и перейти только на BGP - это уже радикальное изменение дизайна - 2.0.0.

При этом разные ДЦ могут иметь разные версии - сеть развивается, ставится новое оборудование, где-то добавляются новые уровни спайнов, где-то - нет, итд.

Про семантическое версионирование мы поговорим в отдельной статье.

Повторюсь - любое изменение (кроме отладочных команд) - это обновление версии. О любых отклонениях от актуальной версии должны оповещаться администраторы.

То же самое касается отката изменений - это не отмена последних команд, это не rollback силами операционной системы устройства - это приведение всей сети к новой (старой) версии.

1.1.4 Мониторинг и самовосстановление сервисов

Это самоочевидная задача в современных сетях выходит на новый уровень.
Зачастую у больших сервис-провайдеров практикуется подход, что упавший сервис надо очень быстро добить и поднять новый, вместо того, чтобы разбираться, что произошло.
«Очень» означает, что со всех сторон нужно обильно обмазаться мониторингами, которые в течение секунд обнаружат малейшие отклонения от нормы.
И здесь уже не достаточно привычных метрик, вроде загрузки интерфейса или доступности узла. Недостаточно и ручного слежения дежурного за ними.
Для многих вещей вообще должен быть Self-Healing - мониторинги зажглись красным и пошли сами подорожник приложили, где болит.

И здесь мы тоже мониторим не только отдельные устройства, но и здоровье сети целиком, причём как вайтбокс, что сравнительно понятно, так и блэкбокс, что уже сложнее.

Что нам понадобится для реализации таких амбициозных планов?

- Иметь список всех устройств в сети, их расположение, роли, модели, версии ПО. (*kazan-leaf-1.lmu.net, Kazan, leaf, Juniper QFX 5120, R18.3*)
- Иметь систему описания сетевых сервисов. (*IGP, BGP, L2/3VPN, Policy, ACL, NTP, SSH*)
- Уметь инициализировать устройство. (*Hostname, Mgmt IP, Mgmt Route, Users, RSA-Keys, LLDP, NETCONF*)
- Настраивать устройство и приводить конфигурацию к нужной (в том числе старой) версии.
- Тестировать конфигурацию
- Периодически проверять состояние всех устройств на предмет отхождения от актуального и сообщать кому следует. (*Ночью кто-то тихонько добавил правило в ACL*)

- Следить за работоспособностью.

1.2 Средства

Звучит достаточно сложно для того, чтобы начать декомпозировать проект на компоненты.

И будет их десять:

1. Инвентарная система
2. Система управления IP-пространством
3. Система описания сетевых сервисов
4. Механизм инициализации устройств
5. Вендор-агностик конфигурационная модель
6. Вендор-интерфейс специфичный драйвер
7. Механизм доставки конфигурации на устройство
8. CI/CD
9. Механизм резервного копирования и поиска отклонений
10. Система мониторинга

    Это, кстати, пример того, как менялся взгляд на цели цикла - в черновике компонентов было 4.

На иллюстрации я изобразил все компоненты и собственно устройство.
Пересекающиеся компоненты взаимодействуют другу с другом.
Чем больше блок, тем больше внимания нужно уделить этому компоненту.

1.2.1 Компонент 1. Инвентарная система

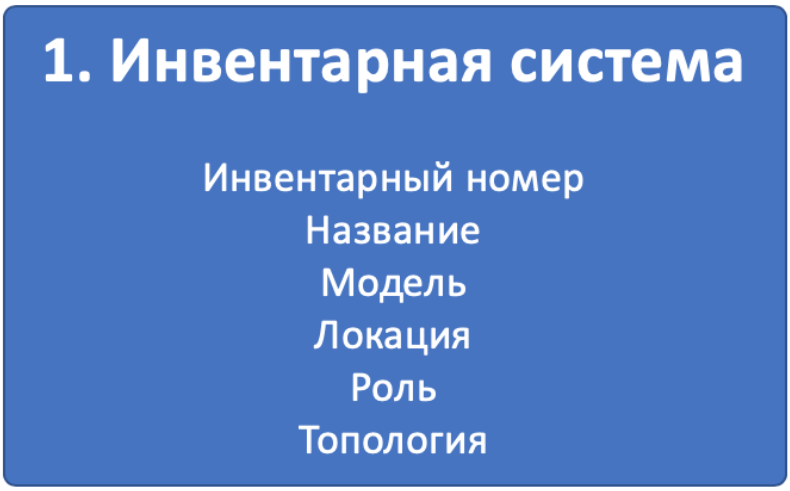
Очевидно, мы хотим знать, какое оборудование, где стоит, к чему подключено.
Инвентарная система - неотъемлемая часть любого предприятия.
Чаще всего для сетевых устройств предприятие имеет отдельную инвентарную систему, которая решает более специфичные задачи.
В рамках цикла статей мы будем называть это DCIM - Data Center Infrastructure Management. Хотя сам термин DCIM, строго говоря, включает в себя гораздо больше.

Для наших задач в ней мы будем хранить следующую информацию про устройство:

- Инвентарный номер
- Название/описание
- Модель (*Huawei CE12800, Juniper QFX5120 итд*)
- Характерные параметры (*платы, интерфейсы итд*)
- Роль (*Leaf, Spine, Border Router итд*)
- Локацию (*регион, город, дата-центр, стойка, юнит*)

Система мониторинга 10
1
Инвентарная система
2
Система управления IP-пространством
3
Система описания сетевых сервисов
4
Механизм инициализации устройства
5
Вендор-агностик конфигурационная модель
6
Вендор-интерфейс специфичный драйвер
7
Механизм доставки конфигурации на устройство
8
CI/CD
Сетевое устройство
9
Механизм резервного копирования и поиска отклонений

- Интерконнекты между устройствами
- Топологию сети



Прекрасно понятно, что нам самим хочется знать всё это.

Но поможет ли это в целях автоматизации?

Безусловно.

Например, мы знаем, что в данном дата-центре на Leaf-коммутаторах, если это Huawei, ACL для фильтрации определённого трафика должны применяться на VLAN, а если это Juniper - то на unit 0 физического интерфейса.

Или нужно раскатить новый Syslog-сервер на все бордеры региона.

В ней же мы будем хранить виртуальные сетевые устройства, например виртуальные маршрутизаторы или рут-рефлекторы. Можем добавить DNS-сервера, NTP, Syslog и вообще всё, что так или иначе относится к сети.

1.2.2 Компонент 2. Система управления IP-пространством

Да, и в наше время находятся коллективы людей, которые ведут учёт префиксов и IP-адресов в Excel-файле. Но современный подход - это всё-таки база данных, с фронтендом на nginx/apache, API и широкими функциями по учёту IP-адресов и сетей с разделением на VRF.

IPAM - IP Address Management.

Для наших задач в ней мы будем хранить следующую информацию:

- VLAN
- VRF
- Сети/Подсети
- IP-адреса
- Привязка адресов к устройствам, сетей к локациям и номерам VLAN

Опять же понятно, что мы хотим быть уверены, что, выделяя новый IP-адрес для лупбэка ToR'а, мы не споткнёмся о то, что он уже был кому-то назначен. Или что один и тот же префикс мы использовали дважды в разных концах сети.

Но как это поможет в автоматизации?

Легко.

Запрашиваем в системе префикс с ролью Loopbacks, в котором есть доступные для выделения IP-адреса - если находится, выделяем адрес, если нет, запрашиваем создание нового префикса.
Или при создании конфигурации устройства мы из этой же системы можем узнать, в каком VRF должен находиться интерфейс.
А при запуске нового сервера скрипт сходит в систему, узнает в каком сервер свитче, в каком порту и какая подсеть назначена на интерфейс - из него и будет выделять адрес сервера.

Напрашивается желание DCIM и IPAM объединить в одну систему, чтобы не дублировать функции и не обслуживать две похожие сущности.
Так мы и сделаем.

1.2.3 Компонент 3. Система описания сетевых сервисов

Если первые две системы хранят переменные, которые ещё нужно как-то использовать, то третья описывает для каждой роли устройства, как оно должно быть настроено.

Стоит выделить два разных типа сетевых сервисов:

- Инфраструктурные

- Клиентские

Первые призваны обеспечить базовую связность и управление устройством. Сюда можно отнести VTY, SNMP, NTP, Syslog, AAA, протоколы маршрутизации, CoPP итд.
Вторые организуют услугу для клиента: MPLS L2/L3VPN, GRE, VXLAN, VLAN, L2TP итд.
Разумеется, есть и пограничные случаи - куда отнести MPLS LDP, BGP? Да и протоколы маршрутизации могут использоваться для клиентов. Но это не принципиально.
Оба типа сервисов раскладываются на конфигурационные примитивы:

- физические и логические интерфейсы (тег/антег, mtu)
- IP-адреса и VRF (IP, IPv6, VRF)
- ACL и политики обработки трафика
- Протоколы (IGP, BGP, MPLS)
- Политики маршрутизации (префикс-листы, коммьюнити, ASN-фильтры).
- Служебные сервисы (SSH, NTP, LLDP, Syslog. . .)
- Итд.

Как именно мы это будем делать, я пока ума не приложу. Разберёмся в отдельной статье.

Если чуть ближе к жизни, то мы могли бы описать, что Leaf-коммутатор должен иметь BGP-сессии со всем подключенными Spine-коммутаторами, импортировать в процесс подключенные сети, принимать от Spine-коммутаторов только сети из определённого префикса. Ограничивать CoPP IPv6 ND до 10 pps итд. В свою очередь спайны держат сессии со всеми подключенными лифами, выступая в качестве рут-рефлекторов, и принимают от них только маршруты определённой длины и с определённым коммунити.

1.2.4 Компонент 4. Механизм инициализации устройства

Под этим заголовком я объединяю множество действий, которые должны произойти, чтобы устройство появилось на радарах и на него можно было попасть удалённо.

- Завести устройство в инвентарной системе.
- Выделить IP-адрес управления.
- Настроить базовый доступ на него: Hostname, IP-адрес управления, маршрут в сеть управления, пользователи, SSH-ключи, протоколы - telnet/SSH/NETCONF

Тут существует три подхода:

- Полностью всё вручную. Устройство привозят на стенд, где обычный органический человек, заведёт его в системы, подключится консолью и настроит. Может сработать на небольших статических сетях.
- ZTP - Zero Touch Provisioning. Железо приехало, встало, по DHCP получило себе адрес, сходило на специальный сервер, самонастроилось.
- Инфраструктура консольных серверов, где первичная настройка происходит через консольный порт в автоматическом режиме.

Про все три поговорим в отдельной статье.

1.2.5 Компонент 5. Вендор-агностик конфигурационная модель

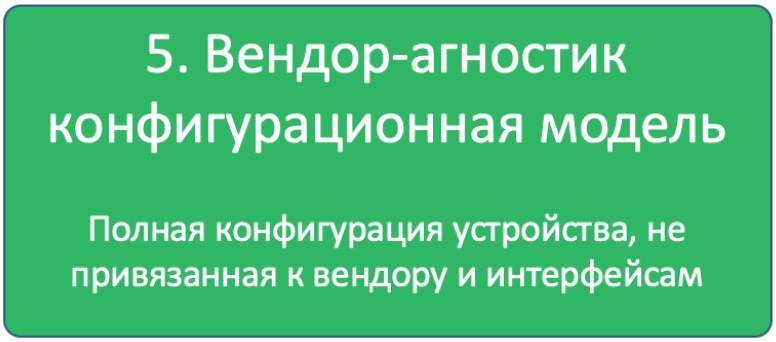
До сих пор все системы были разрозненными лоскутами, дающими переменные и декларативное описание того, что мы хотели бы видеть на сети. Но рано или поздно, придётся иметь дело с конкретикой.

На этом этапе для каждого конкретного устройства примитивы, сервисы и переменные комбинируются в конфигурационную модель, фактически описывающую полную конфигурацию конкретного устройства, только в вендоронезависимой манере.

Что даёт этот шаг? Почему бы сразу не формировать конфигурацию устройства, которую можно просто залить?

На самом деле это позволяет решить три задачи:

1. Не подстраиваться под конкретный интерфейс взаимодействия с устройством. Будь то CLI, NETCONF, RESTCONF, SNMP - модель будет одинаковой.
2. Не держать количество шаблонов/скриптов по числу вендоров в сети, и в случае изменения дизайна, менять одно и то же в нескольких местах.
3. Загружать конфигурацию с устройства (бэкапа), раскладывать её в точно такую же модель и непосредственно сравнивать между собой целевую конфигурацию и имеющуюся для вычисления дельты и подготовки конфигурационного патча, который изменит только те части, которые необходимо или для выявления отклонений.



В результате этого этапа мы получаем вендоронезависимую конфигурацию.

1.2.6 Компонент 6. Вендор-интерфейс специфичный драйвер

Не стоит тешить себя надеждами на то, что когда-то настраивать циску можно будет точно так же, как джунипер, просто отправив на них абсолютно одинаковые вызовы. Несмотря на набирающие популярность whitebox'ы и на появление поддержки NETCONF, RESTCONF, OpenConfig, конкретный контент, который этими протоколами доставляется, отличается от вендора к вендору, и это одно из их конкурентных отличий, которое они так просто не сдадут.

Это примерно точно так же, как OpenContrail и OpenStack, имеющие RestAPI в качестве своего NorthBound-интерфейса, ожидают совершенно разные вызовы.

Итак, на пятом шаге вендоронезависимая модель должна принять ту форму, в которой она поедет на железо.

И здесь все средства хороши (нет): CLI, NETCONF, RESTCONF, SNMP простихоспаде.

Поэтому нам понадобится драйвер, который результат предыдущего шага переложит в нужный формат конкретного вендора: набор CLI команд, структуру XML.

1.2.7 Компонент 7. Механизм доставки конфигурации на устройство

Конфигурацию-то мы сгенерировали, но её ещё нужно доставить на устройства - и, очевидно, не руками.

Во-первых, перед нами тут встаёт вопрос, какой транспорт будем использовать? А выбор на сегодняшний день уже не маленький:

- CLI (telnet, ssh)
- SNMP
- NETCONF
- RESTCONF
- REST API
- OpenFlow (хотя он из списка и выбивается, поскольку это способ доставить FIB, а не настройки)

Давайте тут расставим точки над ё. CLI - это легаси. SNMP... кхе-кхе.

RESTCONF - ещё пока неведомая зверушка, REST API поддерживается почти никем. Поэтому мы в цикле сосредоточимся на NETCONF.

На самом деле, как уже понял читатель, с интерфейсом мы к этому моменту уже определились - результат предыдущего шага уже представлен в формате того интерфейса, который был выбран.

Во-вторых, а какими инструментами мы будем это делать?

Тут выбор тоже большой:

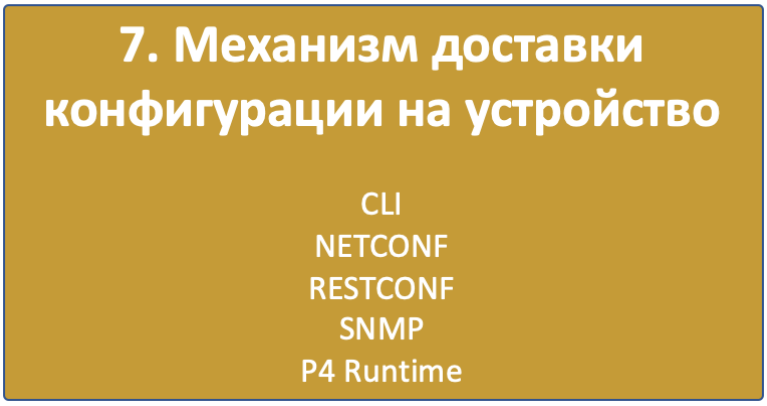
- Самописный скрипт или платформа. Вооружимся ncclient и asyncIO и сами всё сделаем. Что нам стоит, систему деплоймента с нуля построить?
- Ansible с его богатой библиотекой сетевых модулей.
- Salt с его скудной работой с сетью и связкой с Napalm.
- Собственно Napalm, который знает пару вендоров и всё, до свиданья.
- Nornir - ещё один зверёк, которого мы препарируем в будущем.

Здесь ещё фаворит не выбран - будем шупать.

Что здесь ещё важно? Последствия применения конфигурации.

Успешно или нет. Остался доступ на железку или нет.

Кажется, тут поможет commit с подтверждением и валидацией того, что в устройство сгрузили. Это в совокупности с правильной реализацией NETCONF значительно сужает круг подходящих устройств - нормальные коммиты поддерживают не так много производителей. Но это просто одно из обязательных условий в RFP. В конце концов никто не переживает, что ни один российский вендор не пройдёт под условие 32*100GE интерфейса. Или переживает?



1.2.8 Компонент 8. CI/CD

К этому моменту у нас уже готова конфигурация на все устройства сети.

Я пишу «на все», потому что мы говорим о версионировании состояния сети. И даже если нужно поменять настройки всего лишь одного свитча, просчитываются изменения для всей сети. Очевидно, они могут быть при этом нулевыми для большинства узлов.

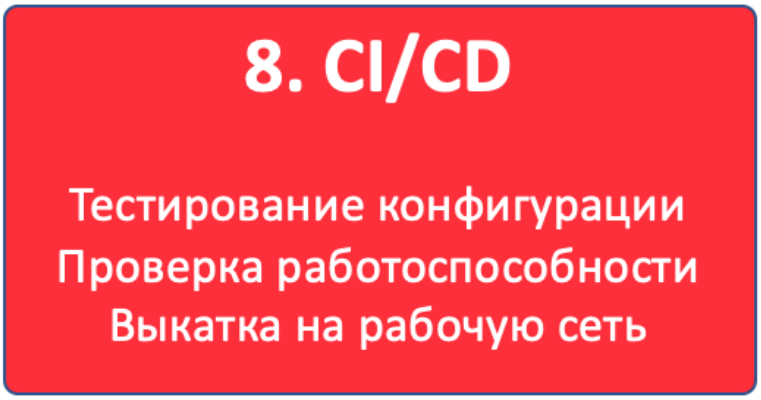
Но, как уже было сказано, выше, мы же не варвары какие-то, чтобы катить всё сразу в прод.

Сгенерированная конфигурация должна сначала пройти через Pipeline CI/CD.

> CI/CD означает Continuous Integration, Continuous Deployment. Это подход, при котором команда не раз в полгода выкладывает новый мажорный релиз, полностью заменяя старый, а регулярно инкрементально внедряет (Deployment) новую функциональность небольшими порциями, каждую из которых всесторонне тестирует на совместимость, безопасность и работоспособность (Integration).

Для этого у нас есть система контроля версий, следящая за изменениями конфигурации, лаборатория, на которой проверяется не ломается ли клиентский сервис, система мониторинга, проверяющая этот факт, и последний шаг - выкатка изменений в рабочую сеть.

За исключением отладочных команд, абсолютно все изменения на сети должны пройти через CI/CD Pipeline - это наш залог спокойной жизни и длинной счастливой карьеры.



1.2.9 Компонент 9. Система резервного копирования и поиска отклонений

Ну про бэкапы лишний раз говорить не приходится.

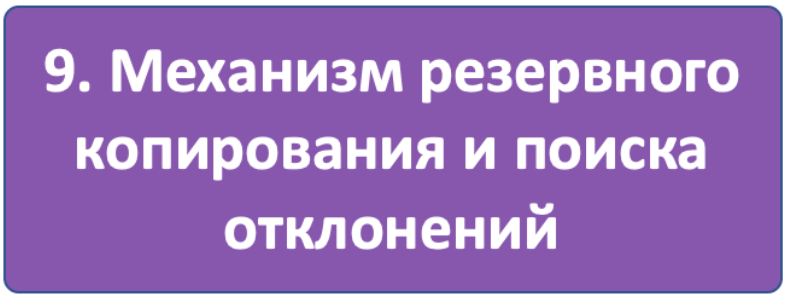
Будем просто их по крону или по факту изменения конфигурации в гит складывать.

А вот вторая часть поинтереснее - за этими бэкапами кто-то должен приглядывать. И в одних случаях этот кто-то должен пойти и вертать всё как было, а в других, мяукнуть кому-нибудь, о том, что непорядок.

Например, если появился какой-то новый пользователь, который не прописан в переменных, нужно от хака подальше его удалить. А если новое файрвольное правило - лучше не трогать, возможно кто-то просто отладку включил, а может новый сервис, растяпа, не по регламенту прописал, а в него уже люди пошли.

От некой небольшой дельты в масштабах всей сети мы всё равно не уйдём, несмотря на любые системы автоматизации и стальную руку руководства. Для отладки проблем всё равно никто конфигурацию не будет вносить в системы. Тем более, что их может даже не предусматривать модель конфигурации.

> Например, файрвольное правило для подсчёта числа пакетов на определённый IP, для локализации проблемы - вполне рядовая временная конфигурация.



1.2.10 Компонент 10. Система мониторинга

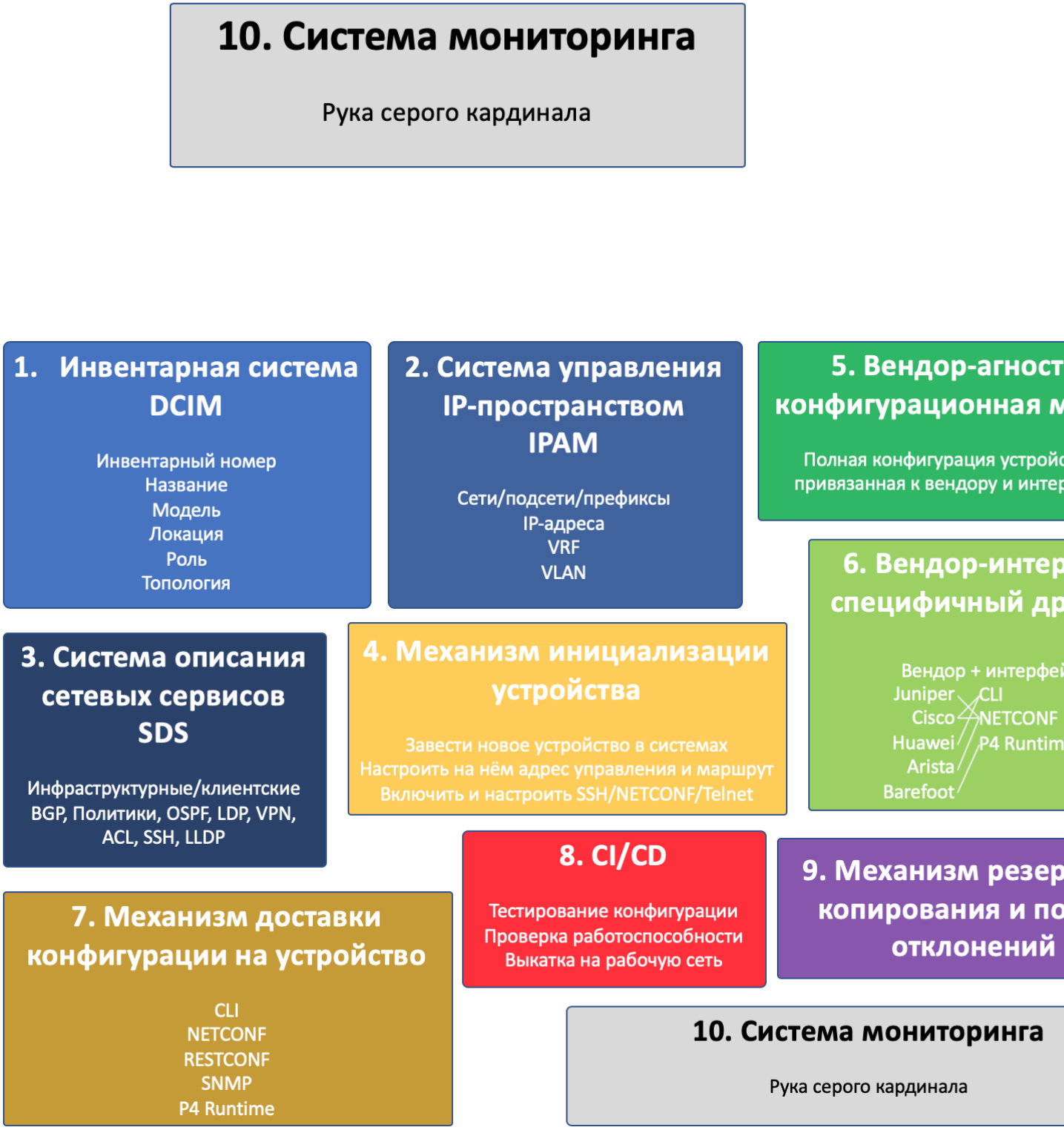
Сначала я не собирался освещать тему мониторинга - всё же объёмная, спорная и сложная тема. Но по ходу дела оказалось, что это неотъемлемая часть автоматизации. И обойти её стороной хотя бы даже без практики нельзя.

Развивая мысль - это органическая часть процесса CI/CD. После выкатки конфигурации на сеть, нам нужно уметь определить, а всё ли с ней теперь в порядке.

И речь не только и не столько о графиках использования интерфейсов или доступности узлов, сколько о более тонких вещах - наличии нужных маршрутов, атрибутов на них, количестве BGP-сессий, OSPF-соседей, End-to-End работоспособности вышележащих сервисов.

А не перестали ли складываться сислоги на внешний сервер, а не сломался ли SFlow-агент, а не начали ли расти дропы в очередях, а не нарушилась ли связность между какой-нибудь парой префиксов?

В отдельной статье мы поразмышляем и над этим.


10. Система мониторинга
Рука серого кардинала
1. Инвентарная система DCIM
Инвентарный номер
Название
Модель
Локация
Роль
Топология
2. Система управления IP-пространством IPAM
Сети/подсети/префиксы
IP-адреса
VRF
VLAN
5. Вендор-агност
конфигурационная м
Полная конфигурация устройс
привязанная к вендору и интер
6. Вендор-интер
специфичный др
Вендор + интерфей
Juniper
Cisco
Huawei
Arista
Barefoot
CLI
NETCONF
P4 Runtim
3. Система описания сетевых сервисов SDS
Инфраструктурные/клиентские
BGP, Политики, OSPF, LDP, VPN,
ACL, SSH, LLDP
4. Механизм инициализации устройства
Завести новое устройство в системах
Настроить на нём адрес управления и маршрут
Включить и настроить SSH/NETCONF/Telnet
8. CI/CD
Тестирование конфигурации
Проверка работоспособности
Выкатка на рабочую сеть
9. Механизм резер
копирования и по
отклонений
7. Механизм доставки конфигурации на устройство
CLI
NETCONF
RESTCONF
SNMP
P4 Runtime
10. Система мониторинга
Рука серого кардинала

1.3 Заключение

В качестве основы я выбрал один из современных дизайнов датацентровой сети - L3 Clos Fabric с BGP в качестве протокола маршрутизации.

Строить сеть мы будем на этот раз на Juniper, потому что теперь интерфейс JunOs - это ванлав.

Усложним себе жизнь использованием только Open Source инструментов и мультивендорной сетью - поэтому кроме джунипер по ходу дела выберу ещё одного счастливчика.

План ближайших публикаций примерно такой:

Сначала я расскажу про виртуальные сети. В первую очередь, потому что мне хочется, а во вторую, потому что без этого дизайн инфраструктурной сети будет не очень понятен.

Потом собственно про дизайн сети: топологию, маршрутизацию, политики.

Соберём лабораторный стенд.

Поразмышляем и, может, попрактикуемся в инициализации устройства в сети.

А дальше про каждый компонент в интимных подробностях.

И да, я не обещаю изящно закончить этот цикл готовым решением. :)

1.3.1 Полезные ссылки

- Перед тем, как углубляться в серию, стоит почитать книгу Наташи Самойленко Python для сетевых инженеров. А, возможно, и пройти курс.
- Полезным будет также почитать RFC про дизайн датацентровых фабрик от Фейсбука за авторством Петра Лапухова.
- О том, как работает Overlay'ный SDN вам даст представление документация по архитектуре Tungsten Fabric.

1.3.2 Спасибы

- Роман Горге. За комментарии и правки.
- Артём Чернобай. За КДПВ.

Глава 2

Часть 1. Виртуализация

Эту часть мы разбили на две:

- **Виртуализация сети**, которая рассказывает, как и зачем она появилась в виде SDN и NFV. И почему вообще мы рассматриваем её в серии статей про автоматизацию.
- **Основы виртуализации**, которая охватывает более общие вопросы о виртуальных хранилищах, процессорах, памяти и сети. Автор этой части - Роман Горге - бывший ведущий подкаста linkmeup.

2.1 Виртуализация сети

В предыдущем выпуске я описал фреймворк сетевой автоматизации. По отзывам у некоторых людей даже этот первый подход к проблеме уже разложил некоторые вопросы по полочкам. И это очень меня радует, потому что наша цель в цикле - не обмазать питоновскими скриптами анзибль, а выстроить систему.

Этот же фреймворк задаёт порядок, в котором мы будем разбираться с вопросом. И виртуализация сети, которой посвящён этот выпуск, не особо укладывается в тематику АДСМ, где мы разбираем автоматику.

Но давайте взглянем на неё под другим углом. Уже давно одной сетью пользуются многие сервисы. В случае оператора связи это 2G, 3G, LTE, ШПД и B2B, например. В случае ДЦ: связность для разных клиентов, Интернет, блочное хранилище, объектное хранилище. И все сервисы требуют изоляции друг от друга. Так появились оверлейные сети. И все сервисы не хотят ждать, когда человек настроит их вручную. Так появились оркестраторы и SDN.

Первый подход к систематической автоматизации сети, точнее её части, давно предпринят и много где внедрён в жизнь: VMWare, OpenStack, Google Compute Cloud, AWS, Facebook.

Вот с ним сегодня и поразбираемся.

adsm@linkmeup> automato it all

2.1.1 Причины

И раз уж мы об этом заговорили, то стоит упомянуть предпосылки к виртуализации сети. На самом деле этот процесс начался не вчера.

Наверно, вы не раз слышали, что сеть всегда была самой инертной частью любой системы. И это правда во всех смыслах. Сеть - это базис, на который опирается всё, и производить изменения на ней довольно сложно - сервисы не терпят, когда сеть лежит. Зачастую вывод из эксплуатации одного узла может сложить большую часть приложений и повлиять на много клиентов. Отчасти поэтому сетевая команда может сопротивляться любым изменениям - потому что сейчас оно как-то работает (*мы, возможно, даже не знаем как*), а тут надо что-то новое настроить, и неизвестно как оно повлияет на сеть.

Чтобы не ждать, когда сетевики прокинут VLAN и любые сервисы не прописывать на каждом узле сети, люди придумали использовать оверлеи - наложенные сети - коих великое многообразие: GRE, IPinIP, MPLS, MPLS L2/L3VPN, VXLAN, GENEVE, MPLSoverUDP, MPLSoverGRE итд.

Их привлекательность заключается в двух простых вещах:

- Настраиваются только конечные узлы - транзитные трогать не приходится. Это значительно ускоряет процесс, а иногда вообще позволяет исключить отдел сетевой инфраструктуры из процесса ввода новых сервисов.
- Нагрузка сокрыта глубоко внутри заголовков - транзитным узлам не нужно ничего знать о ней, об адресации на хостах, маршрутах наложенной сети. А это значит, нужно хранить меньше информации в таблицах, значит взять попроще/подешевле устройство.

В этом не совсем полноценном выпуске я не планирую разбирать все возможные технологии, а скорее описать фреймворк работы оверлейных сетей в ДЦ.

Вся серия будет описывать датацентр, состоящий из рядов однотипных стоек, в которых установлено одинаковое серверное оборудование. На этом оборудовании запускаются виртуальные машины/контейнеры/серверлесс, реализующие сервисы.

2.1.2 Терминология

В цикле **сервером** я буду называть программу, которая реализует серверную сторону клиент-серверной коммуникации.
Физические машины в стойках называть серверами **не** будем.

Физическая машина - x86-компьютер, установленный в стойке. Наиболее часто употребим термин **хост**. Так и будем называть её «**машина**» или **хост**.

Гипервизор - приложение, запущенное на физической машине, эмулирующее физические ресурсы, на которых запускаются Виртуальные Машины. Иногда в литературе и сети слово «гипервизор» используют как синоним «хост».

Виртуальная машина - операционная система, запущенная на физической машине поверх гипервизора. Для нас в рамках данного цикла не так уж важно, на самом ли деле это виртуальная машина или просто контейнер. Будем называть это «**ВМ**»

Тенант - широкое понятие, которое я в этой статье определю как отдельный сервис или отдельный клиент.

Мульти-тенантность или мультиарендность - использование одного и того же приложения разными клиентами/сервисами. При этом изоляция клиентов друг от друга достигается благодаря архитектуре приложения, а не отдельно-запущенным экземплярам.

ToR - Top of the Rack switch - коммутатор, установленный в стойке, к которому подключены все физические машины.

> Кроме топологии ToR, разные провайдеры практикуют End of Row (EoR) или Middle of Row (хотя последнее - пренебрежительная редкость и аббревиатуры MoR я не встречал).

Underlay network или подлежащая сеть или андэрлей - физическая сетевая инфраструктура: коммутаторы, маршрутизаторы, кабели.

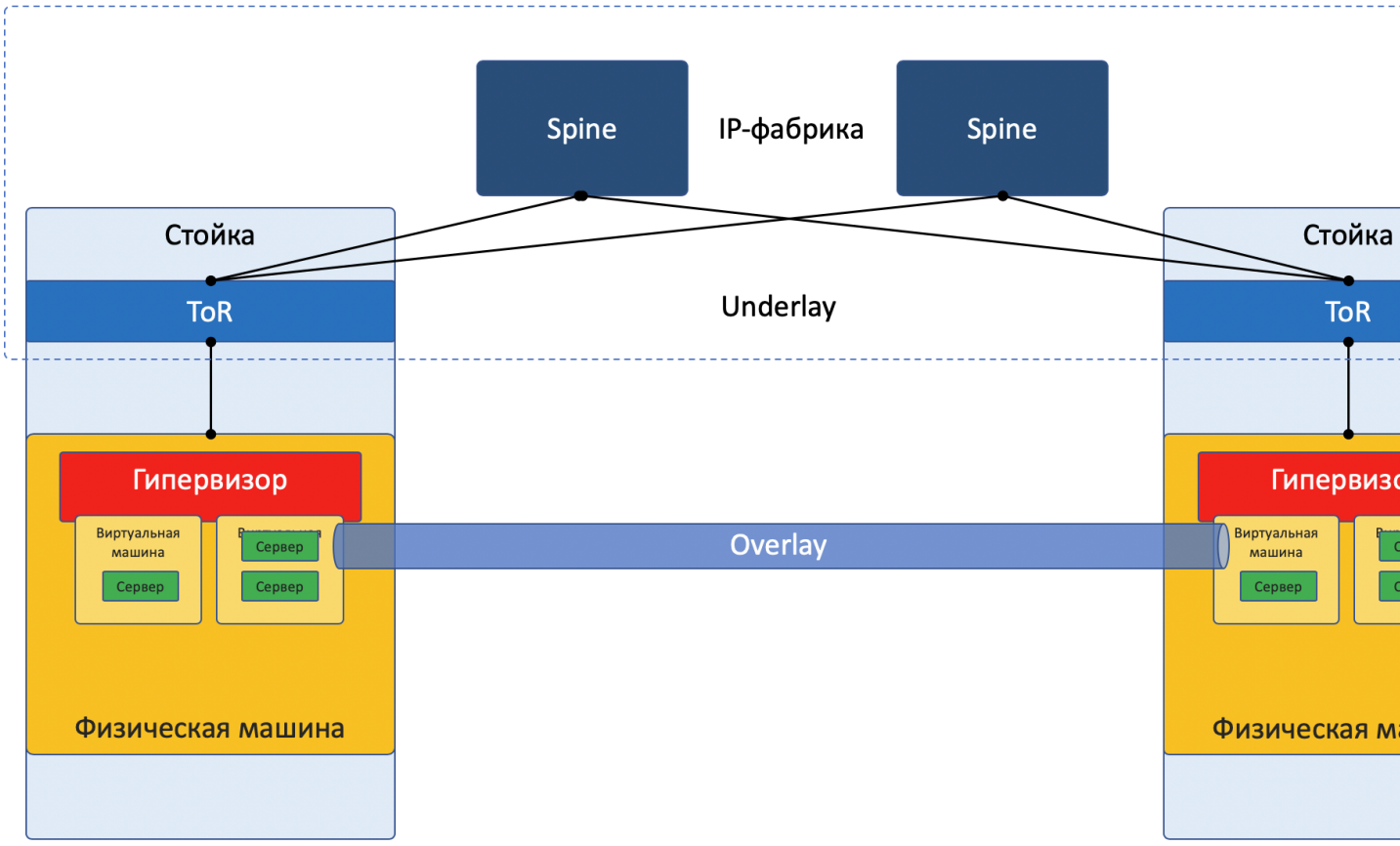
Overlay network или наложенная сеть или оверлей - виртуальная сеть туннелей, работающая поверх физической.

L3-фабрика или IP-фабрика - потрясающее изобретение человечества, позволяющее к собеседованиям не повторять STP и не учить TRILL. Концепция, в которой вся сеть вплоть до уровня доступа исключительно L3, без VLAN и соответственно огромных растянутых широковещательных доменов. Откуда тут слово «фабрика» разберёмся в следующей части.

SDN - Software Defined Network. Едва ли нуждается в представлении. Подход к управлению сетью, когда изменения на сети выполняются не человеком, а программой. Обычно означает вынесение Control Plane за пределы конечных сетевых устройств на контроллер.

NFV - Network Function Virtualization - виртуализация сетевых устройств, предполагающая, что часть функций сети можно запускать в виде виртуальных машин или контейнеров для ускорения внедрения новых сервисов, организации Service Chaining и более простой горизонтальной масштабируемости.

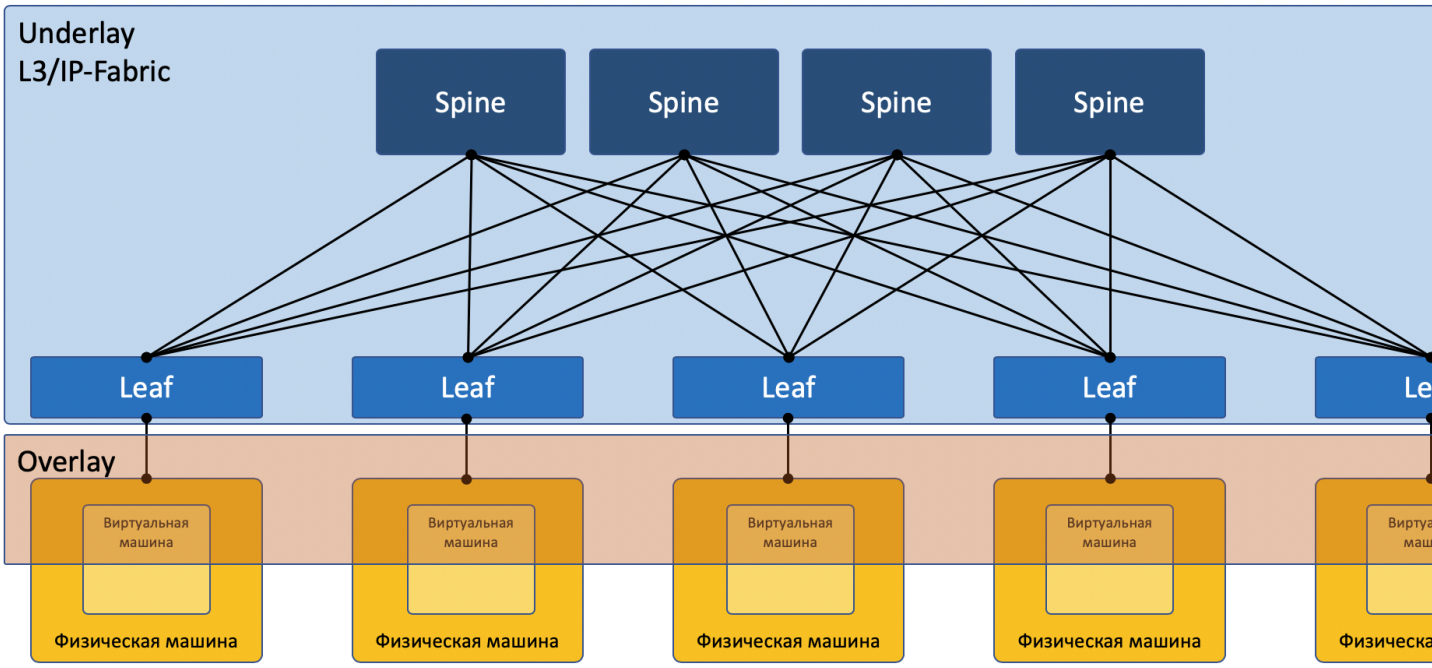
VNF - Virtual Network Function. Конкретное виртуальное устройство: маршрутизатор, коммутатор, файрвол, NAT, IPS/IDS итд. Хотя понятие функция более широкое, чем устройство. Так некоторые функции (например, v‘IMS <https://ru.wikipedia.org/wiki/IMS_(электросвязь)>‘_ - это система виртуальных машин/контейнеров/ПО, реализующая определённую функцию.



Большинство сетей сегодня можно явно разбить на две части:
Underlay - физическая сеть со стабильной конфигурацией.
Overlay - абстракция над Underlay для изоляции тенантов.

Это верно, как для случая ДЦ (который мы разберём в этой статьей), так и для ISP (который мы разбирать не будем, потому что уже было в СДСМ. С энтерпрайзными сетями, конечно, ситуация несколько иная.

Картинка с фокусом на сеть:



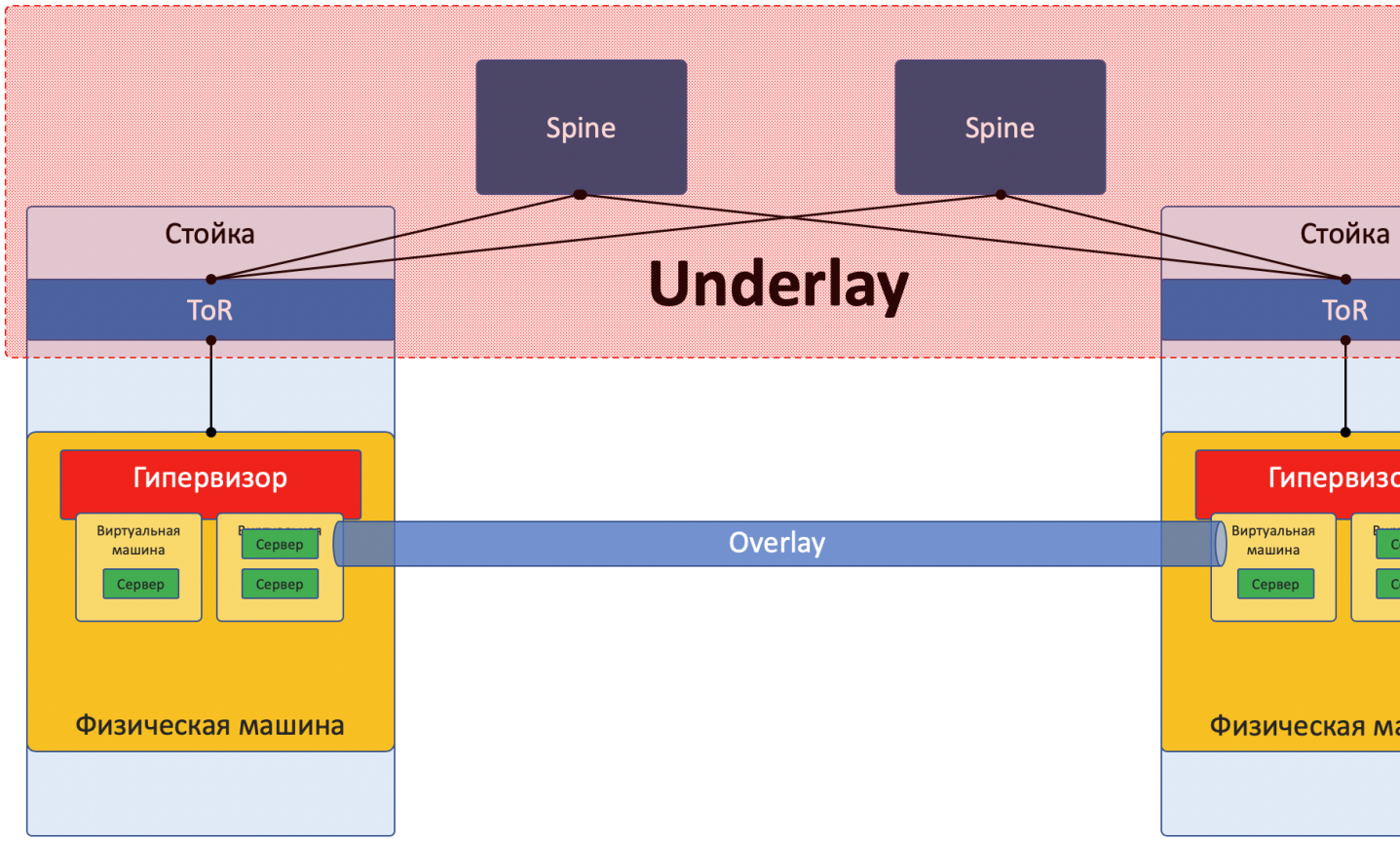
2.1.3 Underlay

Underlay - это физическая сеть: аппаратные коммутаторы и кабели. Устройства в андерлее знают, как добраться до физических машин.

Опирается он на стандартные протоколы и технологии. Не в последнюю очередь потому, что аппаратные устройства по сей день работают на проприетарном ПО, не допускающем ни программирование чипа, ни реализацию своих протоколов, соответственно, нужна совместимость с другими вендорами и стандартизация.

> А вот кто-нибудь вроде Гугла может себе позволить разработку собственных коммутаторов и отказ от общепринятых протоколов. Но LAN_DC не Гугл.

Underlay сравнительно редко меняется, потому что его задача - базовая IP-связность между физическими машинами. Underlay ничего не знает о запущенных поверх него сервисах, клиентах, тенантах - ему нужно только доставить пакет от одной машины до другой. Underlay может быть например таким:


Spine
Spine
Underlay
Стойка
ToR
Стойка
ToR
Гипервизор
Виртуальная машина
Сервер
Сервер
Сервер
Overlay
Виртуальная машина
Сервер
Физическая машина
Физическая ма

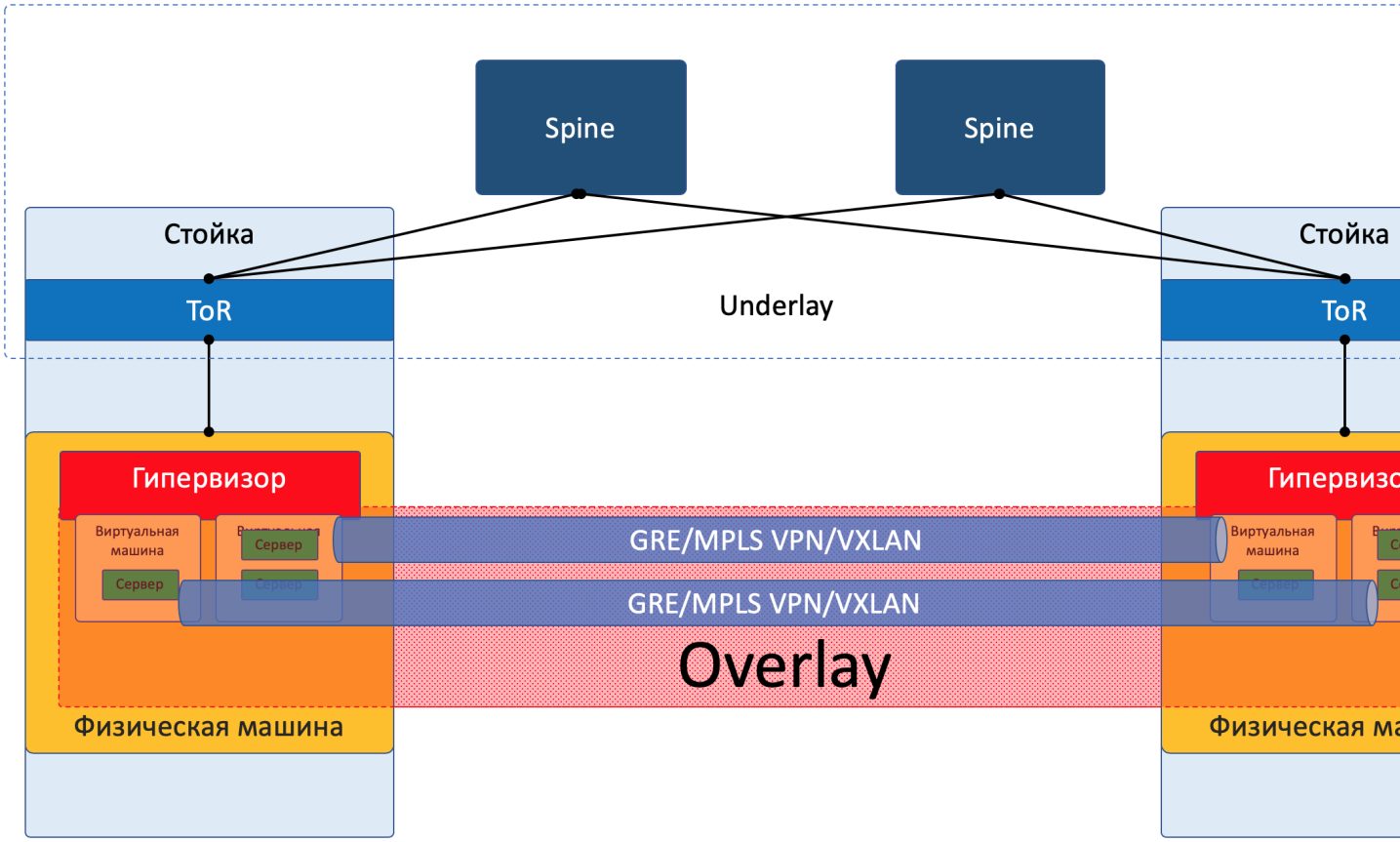
- IPv4+OSPF
- IPv6+ISIS+BGP+L3VPN
- IP+EBGP
- L2+TRILL
- L2+STP

Настраивается Underlay'ная сеть классическим образом: CLI/GUI/NETCONF. Вручную, скриптами, проприетарными утилитами.

Более подробно андерлею будет посвящена следующая статья цикла.

2.1.4 Overlay

Overlay - виртуальная сеть туннелей, натянутая поверх Underlay, она позволяет ВМ одного клиента общаться друг с другом, при этом обеспечивая изоляцию от других клиентов.
Данные клиента инкапсулируются в какие-либо туннелирующие заголовки для передачи через общую сеть.



Так ВМ одного клиента (одного сервиса) могут общаться друг с другом через Overlay, даже не подозревая какой на самом деле путь проходит пакет. Overlay может быть например таким, как уже я упоминал выше:

- GRE-туннель

- VXLAN
- EVPN
- L3VPN
- GENEVE

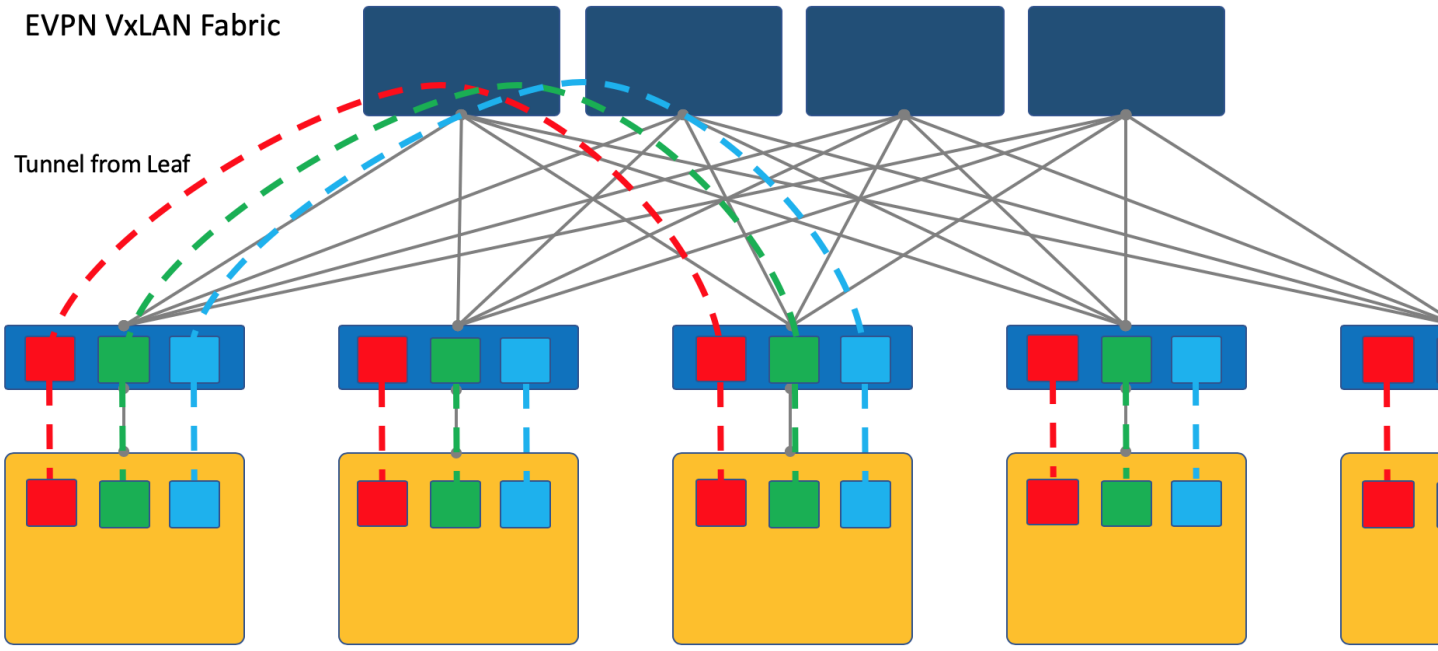
Overlay'ная сеть обычно настраивается и поддерживается через центральный контроллер. С него конфигурация, Control Plane и Data Plane доставляются на устройства, которые занимаются маршрутизацией и инкапсуляцией клиентского трафика. Чуть ниже разберём это на примерах. **Да, это SDN в чистом виде.**

Существует два принципиально различающихся подхода к организации Overlay-сети:

1. Overlay с ToR'a
2. Overlay с хоста

Overlay с ToR'a

Overlay может начинаться на коммутаторе доступа (ToR), стоящем в стойке, как это происходит, например, в случае VXLAN-фабрики.
Это проверенный временем на сетях ISP механизм и все вендоры сетевого оборудования его поддерживают.
Однако в этом случае ToR-коммутатор должен уметь разделять различные сервисы, соответственно, а сетевой администратор должен в известной степени сотрудничать с администраторами виртуальных машин и вносить изменения (пусть и автоматически) в конфигурацию устройств.



Тут я отошлю читателя к статье о VxLAN на хабре нашего старого друга bormoglotx.

В этой презентации с ENOG подробно описаны подходы к строительству сети ДЦ с EVPN VXLAN-фабрикой.

А для более полного погружения в реалии, можно почитать цискину книгу A Modern, Open, and Scalable Fabric: VXLAN EVPN.

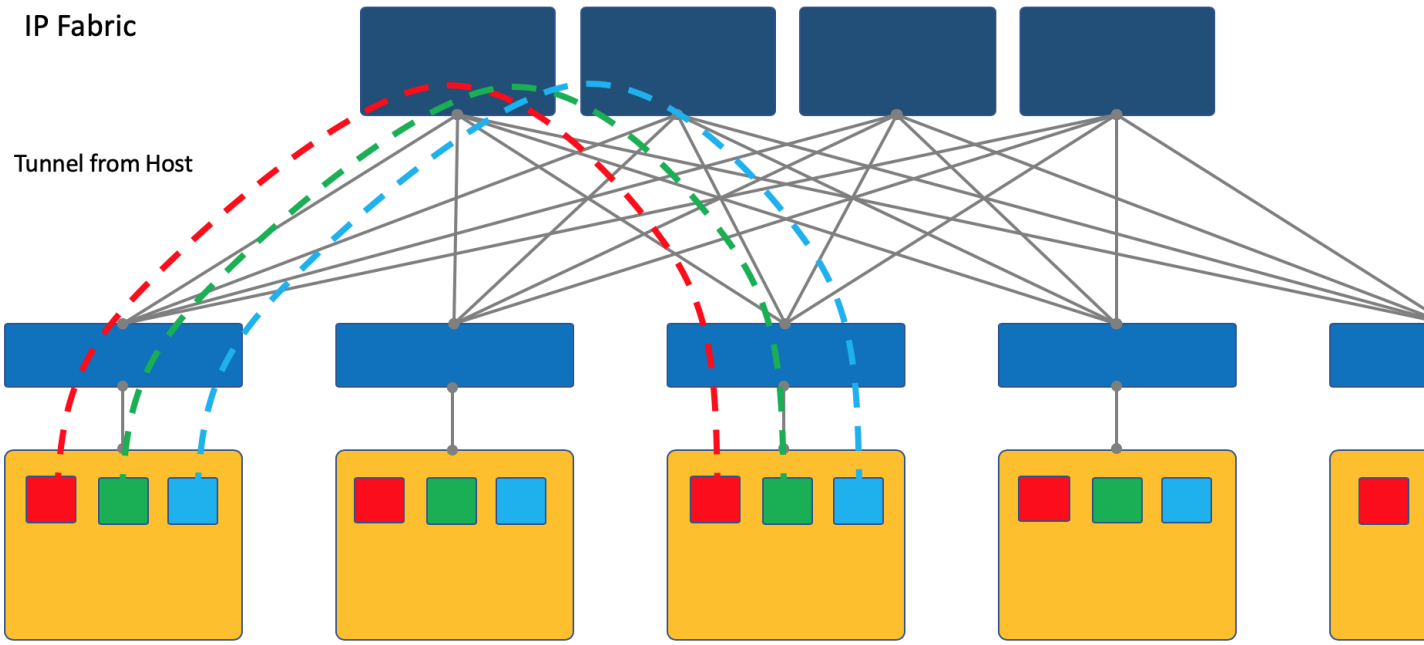
> Замечу, что VXLAN - это только метод инкапсуляции и терминация туннелей может происходить не на ToR'е, а на хосте, как это происходит в случае OpenStack'а, например. Однако, VXLAN-фабрика, где overlay начинается на ToR'е является одним из устоявшихся дизайнов оверлейной сети.

Overlay с хоста

Другой подход - начинать и терминировать туннели на конечных хостах.

В этом случае сеть (Underlay) остаётся максимально простой и статичной.

А хост сам делает все необходимые инкапсуляции.



Для этого потребуется, безусловно, запускать специальное приложение на хостах, но оно того стоит.

Во-первых, запустить клиент на linux-машине проще или, скажем так, - вообще возможно - в то время как на коммутаторе, скорее всего, придётся пока обращаться к проприетарным SDN-решениям, что убивает идею мультивендорности.

Во-вторых, ToR-коммутатор в этом случае можно оставить максимально простым, как с точки зрения Control Plane'а, так и Data Plane'а. Действительно - с SDN-контроллером ему тогда общаться не нужно, и хранить сети/ARP'ы всех подключенных клиентов - тоже - достаточно знать IP-адрес физической машины, что кратно облегчает таблицы коммутации/маршрутизации.

В серии АДСМ я выбираю подход оверлея с хоста - далее мы говорим только о нём и возвращаться к VXLAN-фабрике мы уже не будем.

Проще всего рассмотреть на примерах. И в качестве подопытного мы возьмём OpenSource'ную SDN платформу OpenContrail, ныне известную как Tungsten Fabric.

> В конце статьи я приведу некоторые размышления на тему аналогии с OpenFlow и OpenvSwitch.

На примере Tungsten Fabric

На каждой физической машине есть **vRouter** - виртуальный маршрутизатор, который знает о подключенных к нему сетях и каким клиентам они принадлежат - по сути - PE-маршрутизатор. Для каждого клиента он поддерживает изолированную таблицу маршрутизации (читай VRF). И собственно vRouter делает Overlay'ное туннелирование.

Чуть подробнее про vRouter - в конце статьи.

Каждая ВМ, расположенная на гипервизоре, соединяется с vRouter'ом этой машины через TAP-интерфейс.

TAP - Terminal Access Point - виртуальный интерфейс в ядре linux, которые позволяет осуществлять сетевое взаимодействие.

Если за vRouter'ом находится несколько сетей, то для каждой из них создаётся виртуальный интерфейс, на который назначается IP-адрес - он будет адресом шлюза по умолчанию.

Все сети одного клиента помещаются в один **VRF** (одну таблицу), разных - в разные.

Сделаю тут оговорку, что не всё так просто, и отправлю любознательного читателя в конец статьи.

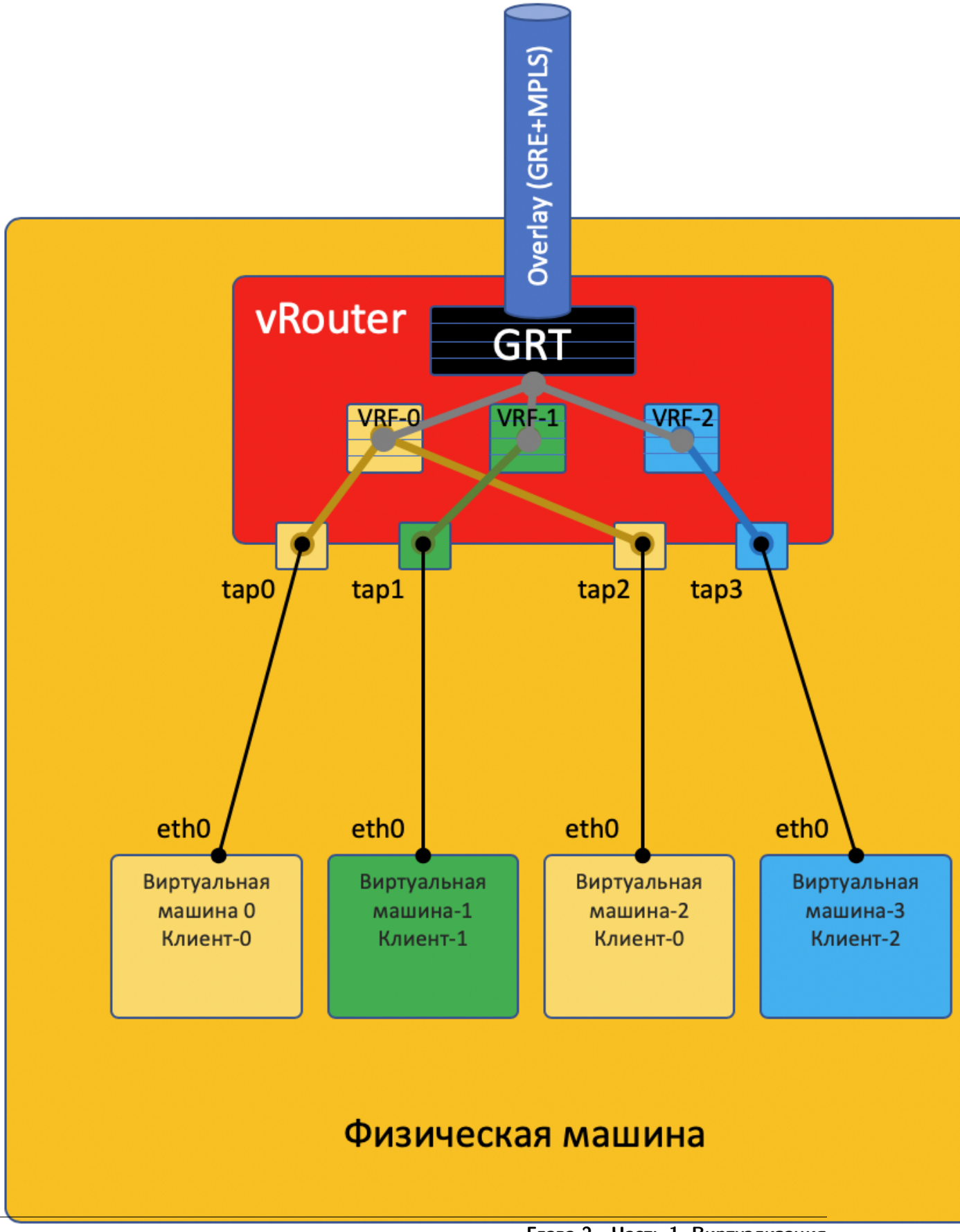
Чтобы vRouter'ы могли общаться друг с другом, а соответственно и ВМ, находящиеся за ними, они обмениваются маршрутной информацией через **SDN-контроллер**.

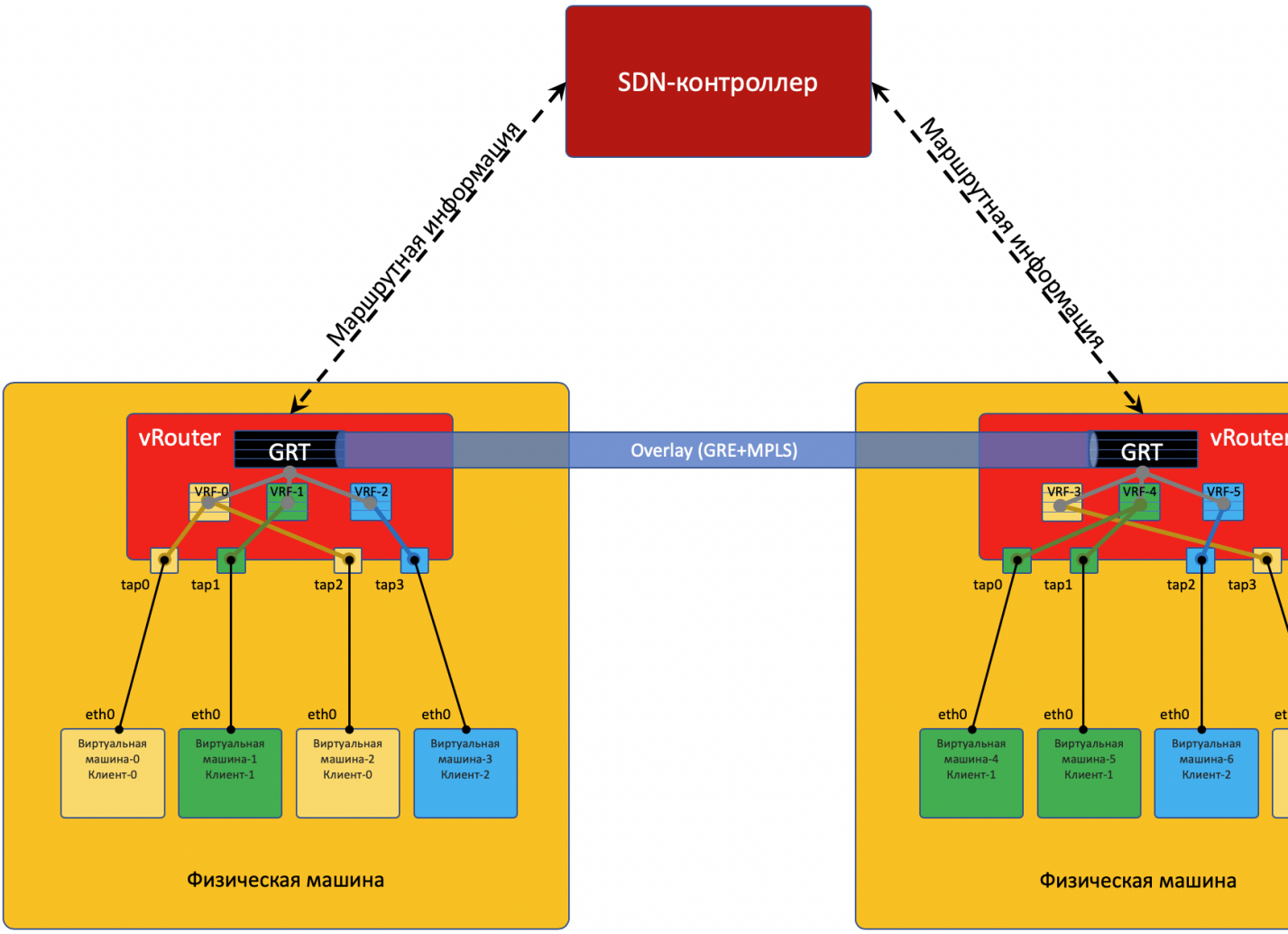
Чтобы выбраться во внешний мир, существует точка выхода из матрицы - шлюз виртуальной сети **VNGW** - Virtual Network GateWay (*термин мой*).

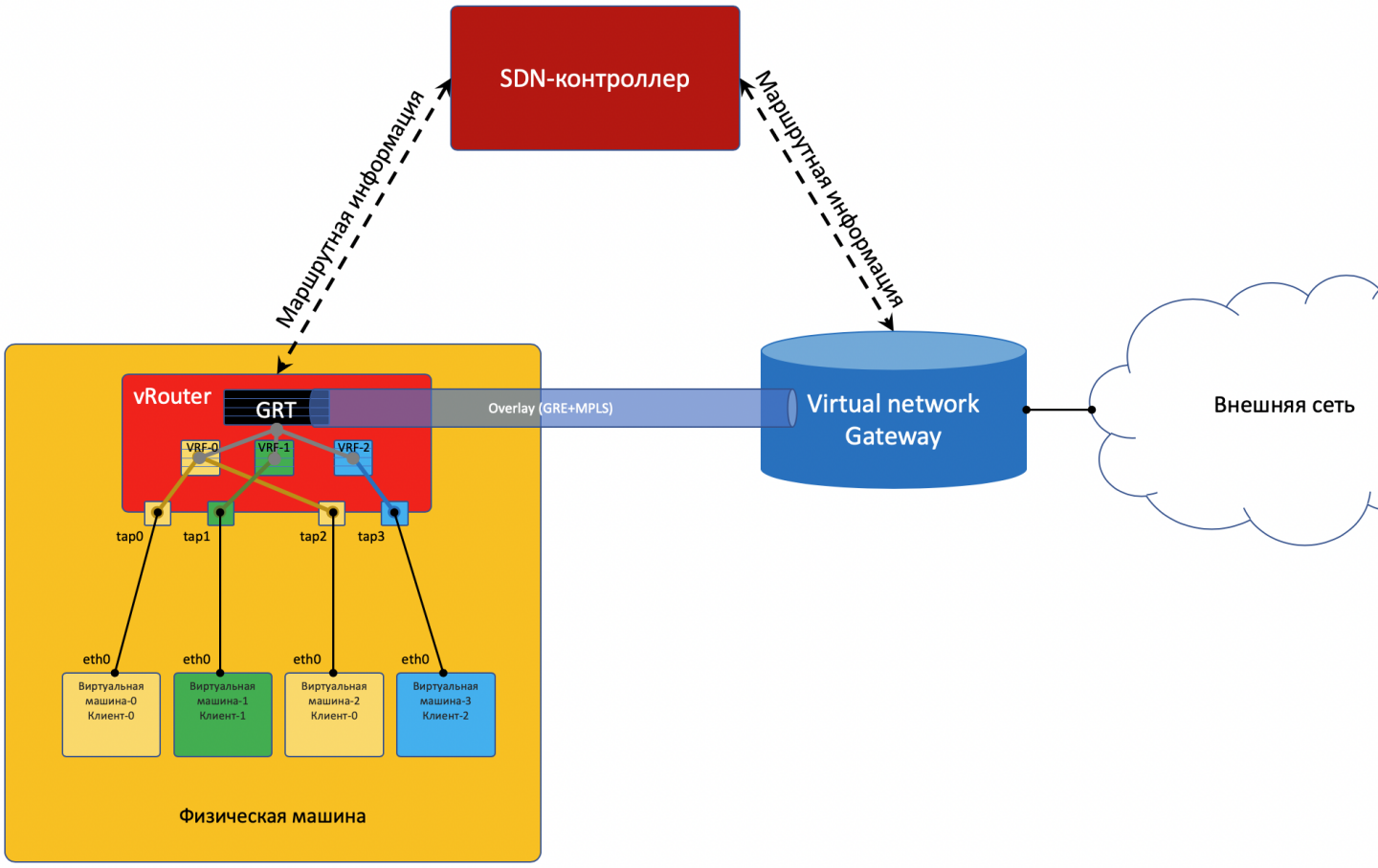
Теперь рассмотрим примеры коммуникаций - и будет ясность.

Коммуникация внутри одной физической машины

VM0 хочет отправить пакет на VM2. Предположим пока, что это ВМ одного клиента.

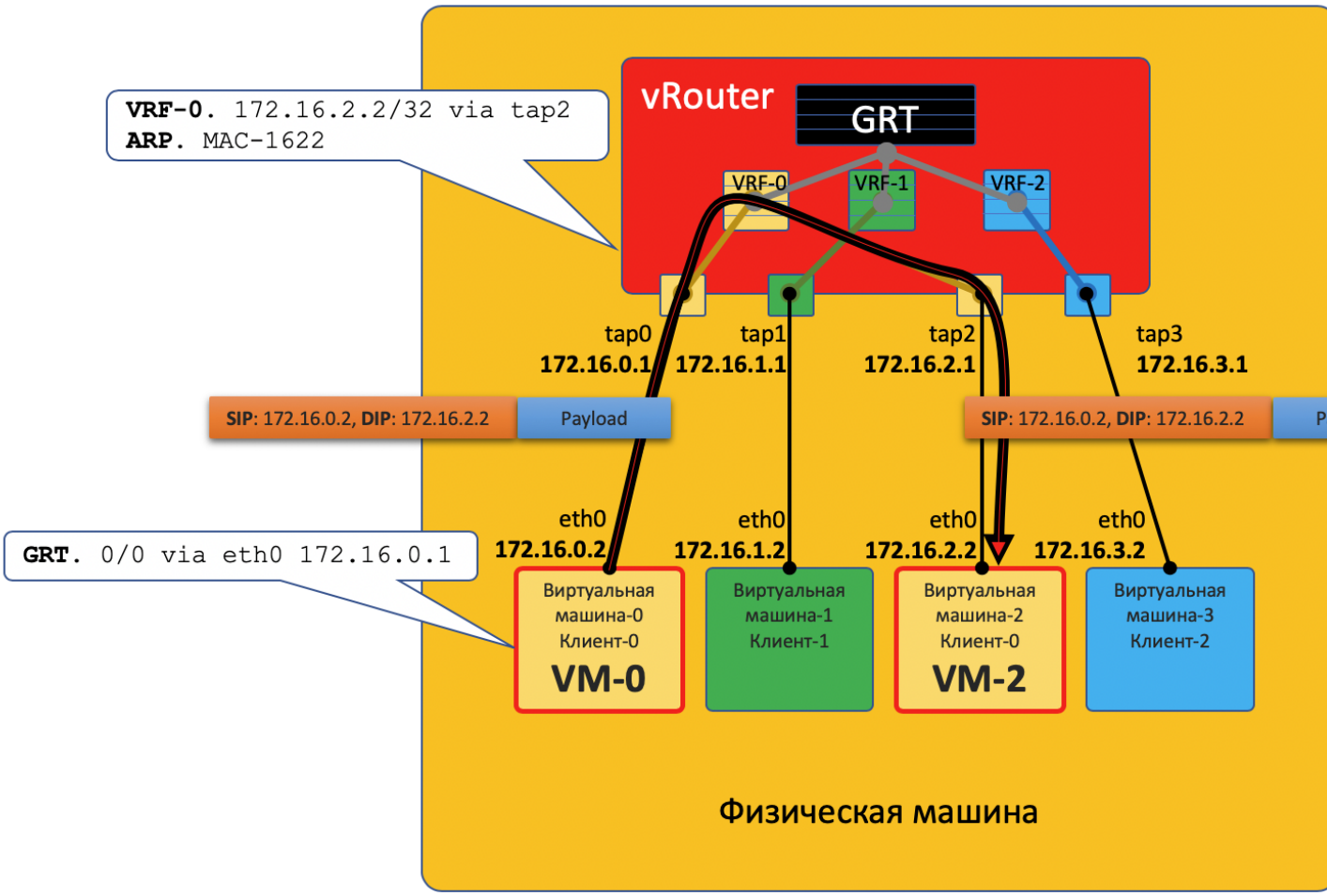

Overlay (GRE+MPLS)
vRouter
GRT
VRF-0
VRF-1
VRF-2
tap0
tap1
tap2
tap3
eth0
eth0
eth0
eth0
Виртуальная машина 0 Клиент-0
Виртуальная машина-1 Клиент-1
Виртуальная машина-2 Клиент-0
Виртуальная машина-3 Клиент-2
Физическая машина


SDN-контроллер
Маршрутная информация
Маршрутная информация
vRouter
GRT
Overlay (GRE+MPLS)
GRT
vRoute
VRF-0
VRF-1
VRF-2
VRF-3
VRF-4
VRF-5
tap0
tap1
tap2
tap3
tap0
tap1
tap2
tap3
eth0
eth0
eth0
eth0
eth0
eth0
eth0
Виртуальная машина-0 Клиент-0
Виртуальная машина-1 Клиент-1
Виртуальная машина-2 Клиент-0
Виртуальная машина-3 Клиент-2
Виртуальная машина-4 Клиент-1
Виртуальная машина-5 Клиент-1
Виртуальная машина-6 Клиент-2
Физическая машина
Физическая машина


SDN-контроллер
Маршрутная информация
Маршрутная информация
vRouter
GRT
VRF-0
VRF-1
VRF-2
Overlay (GRE+MPLS)
Virtual network Gateway
Внешняя сеть
tap0
tap1
tap2
tap3
eth0
eth0
eth0
eth0
Виртуальная машина-0 Клиент-0
Виртуальная машина-1 Клиент-1
Виртуальная машина-2 Клиент-0
Виртуальная машина-3 Клиент-2
Физическая машина

Data Plane

1. У VM-0 есть маршрут по умолчанию в его интерфейс eth0. Пакет отправляется туда. Этот интерфейс eth0 на самом деле виртуально соединён с виртуальным маршрутизатором vRouter через TAP-интерфейс tap0.
2. vRouter анализирует на какой интерфейс пришёл пакет, то есть к какому клиенту (VRF) он относится, сверяет адрес получателя с таблицей маршрутизации этого клиента.
3. Обнаружив, что получатель на этой же машине за другим портом, vRouter просто отправляет пакет в него без каких-либо дополнительных заголовков - на этот случай на vRouter'е уже есть ARP-запись.



Пакет в этом случае не попадает в физическую сеть - он смаршрутизировался внутри vRouter'а.

Control Plane

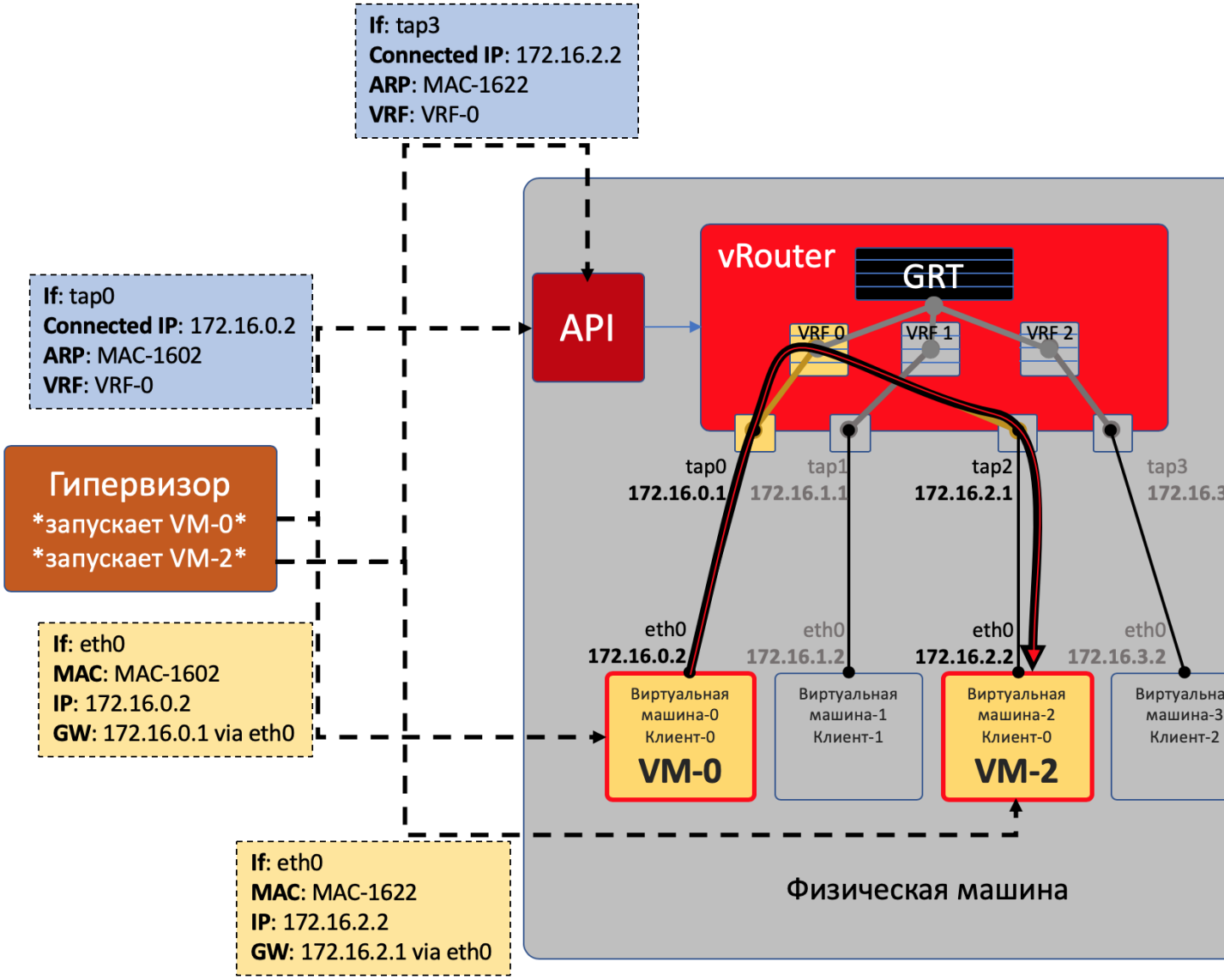
Гипервизор при запуске виртуальной машины сообщает ей:

- Её собственный IP-адрес.
- Маршрут по умолчанию - через IP-адрес vRouter'а в этой сети.

vRouter'у через специальный API гипервизор сообщает:

- Что нужно создать виртуальный интерфейс.
- Какой ей (ВМ) нужно создать Virtual Network.
- К какому VRF его (VN) привязать.
- Статическую ARP-запись для этой VM - за каким интерфейсом находится её IP-адрес и к какому MAC-адресу он привязан.

И снова, реальная процедура взаимодействия упрощена в угоду понимания концепции.



Таким образом все ВМ одного клиента на данной машине vRouter видит как непосредственно подключенные сети и может сам между ними маршрутизировать.

А вот VM0 и VM1 принадлежат разным клиентам, соответственно, находятся в разных таблицах vRouter'а.

Смогут ли они друг с другом общаться напрямую, зависит от настроек vRouter и дизайна сети.

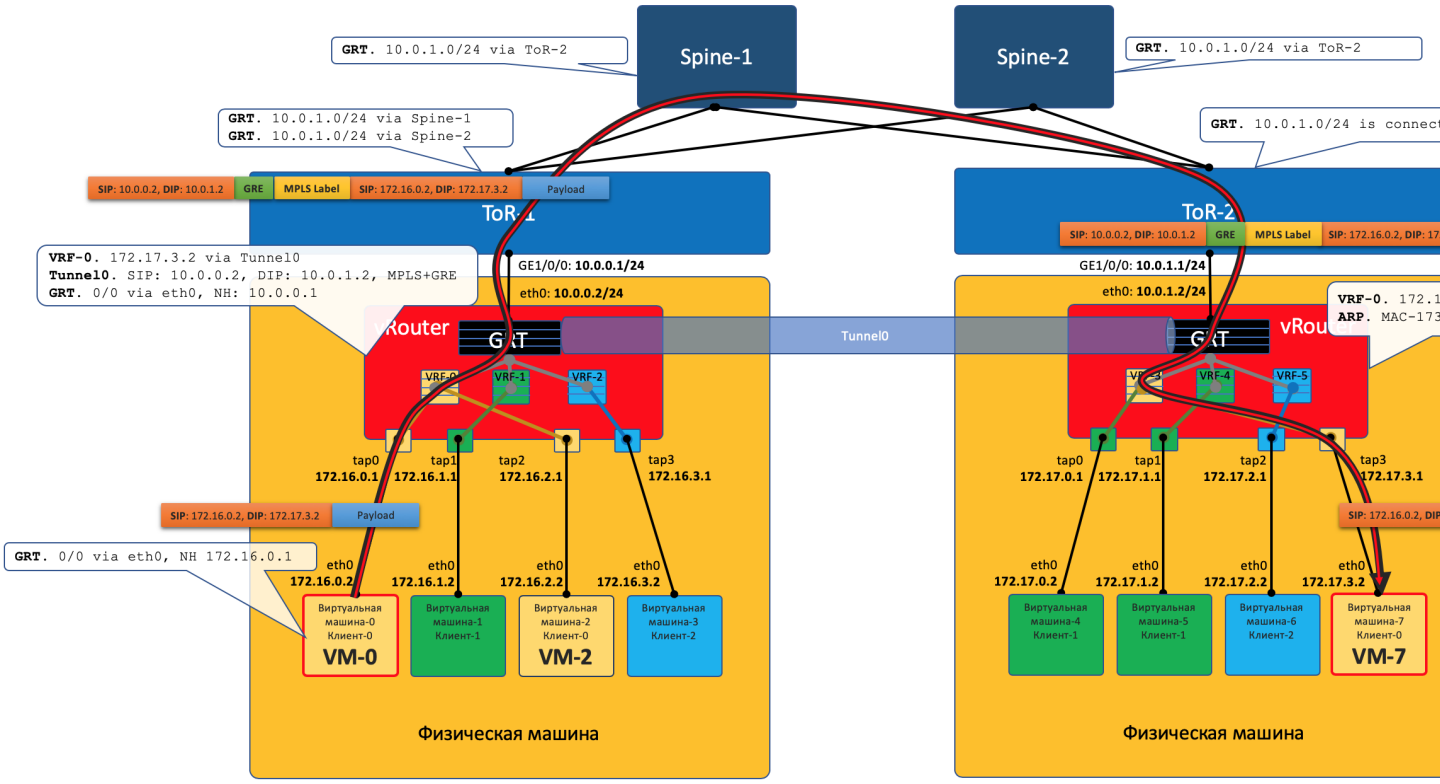
Например, если ВМ обоих клиентов используют публичные адреса, или NAT происходит на самом vRouter'е, то можно сделать и прямую маршрутизацию на vRouter.

В противной же ситуации возможно пересечение адресных пространств - нужно ходить через NAT-сервер, чтобы получить публичный адрес - это похоже на выход во внешние сети, о которых ниже.

Коммуникация между ВМ, расположенными на разных физических машинах

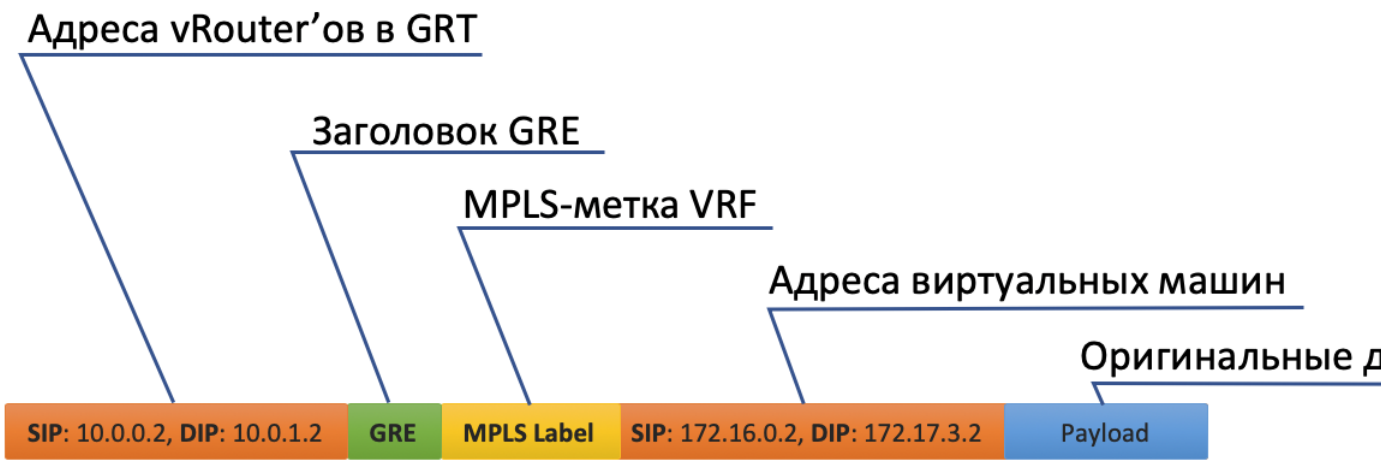
Data Plane

1. Начало точно такое же: VM-0 посылает пакет с адресатом VM-7 (172.17.3.2) по своему дефолту.
2. vRouter его получает и на этот раз видит, что адресат находится на другой машине и доступен через туннель Tunnel0.
3. Сначала он вешает метку MPLS, идентифицирующую удалённый интерфейс, чтобы на обратной стороне vRouter мог определить куда этот пакет поместить причём без дополнительных лукапов.



4. У Tunnel0 источник 10.0.0.2, получатель: 10.0.1.2.
   vRouter добавляет заголовки GRE (или UDP) и новый IP к исходному пакету.

5. В таблице маршрутизации vRouter есть маршрут по умолчанию через адрес ToR1 10.0.0.1. Туда и отправляет.



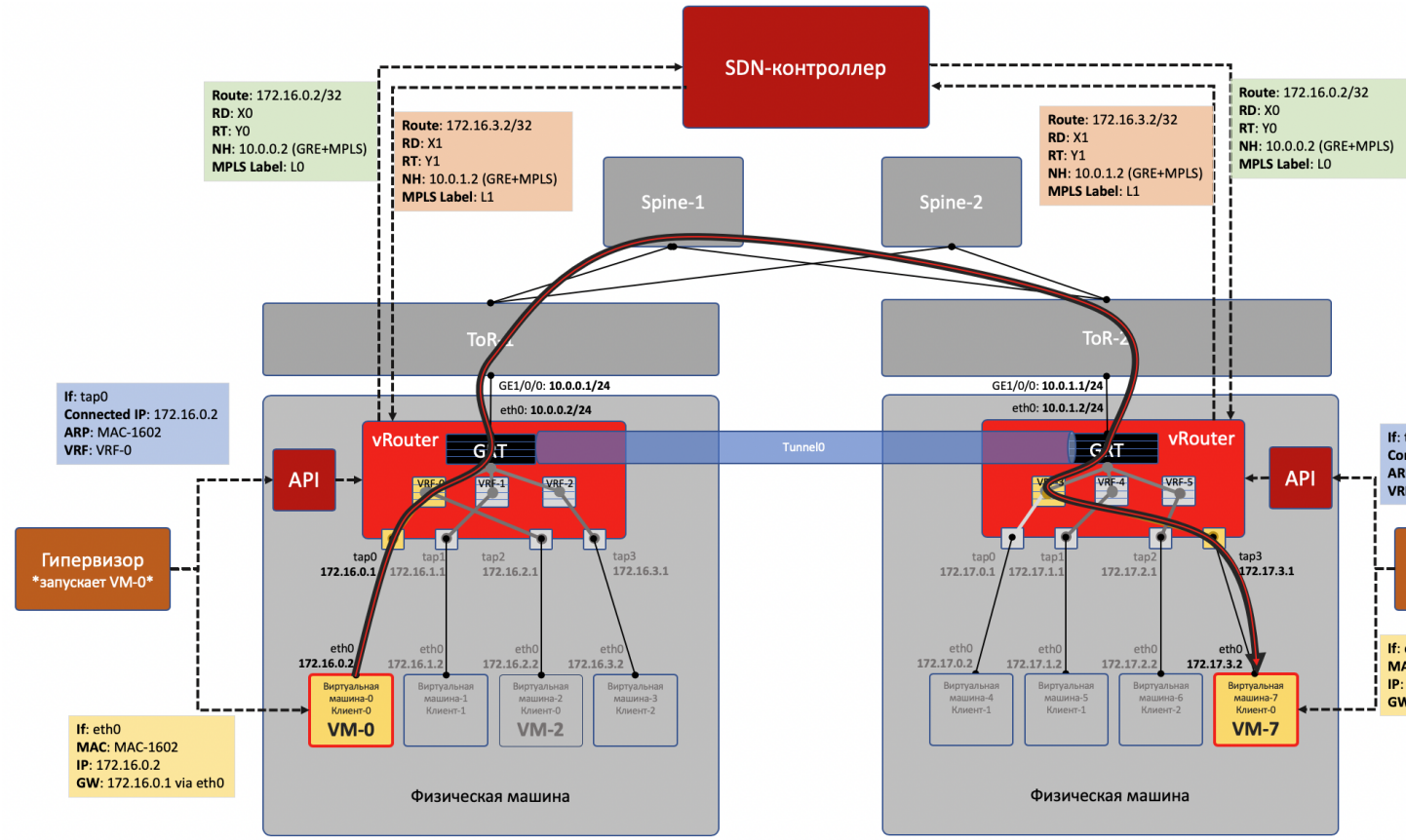
6. ToR1 как участник Underlay сети знает (например, по OSPF), как добраться до 10.0.1.2, и отправляет пакет по маршруту. Обратите внимание, что здесь включается ECMP. На иллюстрации два некстхопа, и разные потоки будут раскладываться в них по хэшу. В случае настоящей фабрики тут будет скорее 4 некстхопа.
   При этом знать, что находится под внешним заголовком IP ему не нужно. То есть фактически под IP может быть бутерброд из IPv6 over MPLS over Ethernet over MPLS over GRE over over over GREка.
7. Соответственно на принимающей стороне vRouter снимает GRE и по MPLS-метке понимает, в какой интерфейс этот пакет надо передать, раздевает его и отправляет в первоначальном виде получателю.

Control Plane

При запуске машины происходит всё то же, что было описано выше. И плюс ещё следующее:

- Для каждого клиента vRouter выделяет MPLS-метку. Это сервисная метка L3VPN, по которой клиенты будут разделяться в пределах одной физической машины.
  На самом деле MPLS-метка выделяется vRouter'ом безусловно всегда - ведь неизвестно заранее, что машина будет взаимодействовать только с другими машинам за тем же vRouter'ом и это скорее всего даже не так.
- vRouter устанавливает соединение с SDN-контроллером по протоколу BGP (или похожему на него - в случае TF -это XMPP 0_o).
- Через эту сессию vRouter сообщает SDN-контроллеру маршруты до подключенных сетей:
  - Адрес сети
  - Метод инкапсуляции (MPLSoGRE, MPLSoUDP, VXLAN)
  - MPLS-метку клиента
  - Свой IP-адрес в качестве nexthop
- SDN-контроллер получает такие маршруты ото всех подключенных vRouter'ов, и отражает их другим. То есть он выступает Route Reflector'ом.

То же самое происходит и в обратную сторону.



Overlay может меняться хоть каждую минуту. Примерно так это и происходит в публичных облаках, когда клиенты регулярно запускают и выключают свои виртуальные машины.

Центральный контроллер берёт на себя все сложности с поддержанием конфигурации и контролем таблиц коммутации/маршрутизации на vRouter.

Если говорить грубо, то контроллер запиривается со всеми vRouter'ами по BGP (или похожему на него протоколу) и просто передаёт маршрутную информацию. BGP, например, уже имеет Address-Family для передачи метода инкапсуляции MPLS-in-GRE или MPLS-in-UDP.

При этом не меняется никоим образом конфигурация Underlay-сети, которую кстати, автоматизировать на порядок сложнее, а сломать неловким движением проще.

Выход во внешний мир

Где-то симуляция должна закончиться, и из виртуального мира нужно выйти в реальный. И нужен таксофон^W шлюз.

Практикуют два подхода:

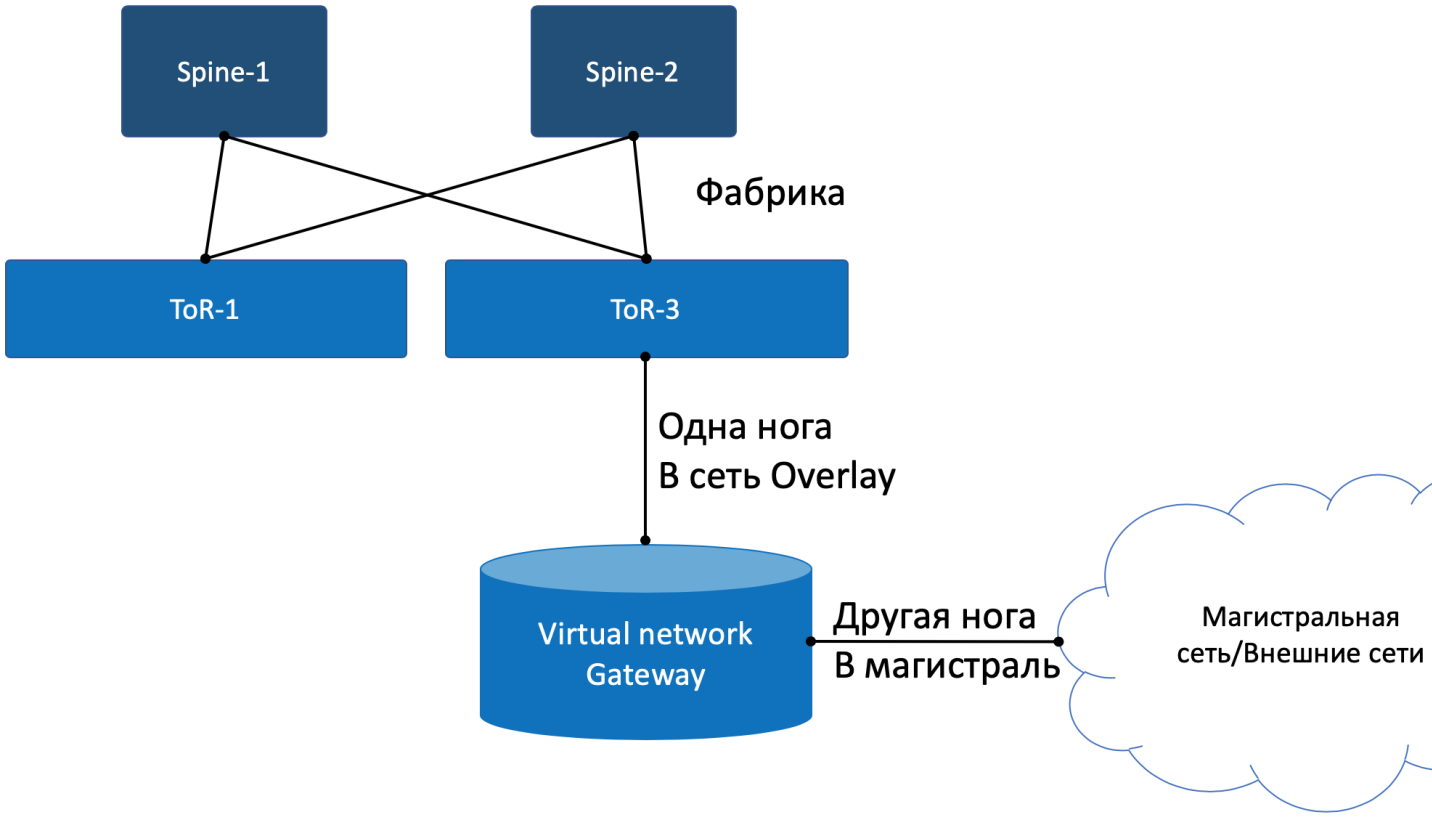
1. Ставится аппаратный маршрутизатор.
2. Запускается какой-либо appliance, реализующий функции маршрутизатора (да-да, вслед за SDN мы и с VNF столкнулись). Назовём его виртуальный шлюз.

   Преимущество второго подхода в дешёвой горизонтальной масштабируемости - не хватает мощности - запустили ещё одну виртуалку со шлюзом. На любой физической машине, без необходимости искать свободные стойки, юниты, вывода питания, покупать саму железку, везти её, устанавливать, коммутировать, настраивать, а потом ещё и менять в ней сбойные компоненты.

   Минусы же у виртуального шлюза в том, что единица физического маршрутизатора всё же на порядки мощнее многоядерной виртуалки, а его софт, подогнанный под его же аппаратную основу, работает значительно стабильнее (*нет*). Сложно отрицать и тот факт, что программно-аппаратный комплекс просто работает, требуя только настройки, в то время как запуск и обслуживание виртуального шлюза - занятие для сильных инженеров.

Одной своей ногой шлюз смотрит в виртуальную сеть Overlay, как обычная Виртуальная Машина, и может взаимодействовать со всеми другими ВМ. При этом она может терминировать на себе сети всех клиентов и, соответственно, осуществлять и маршрутизацию между ними.

Другой ногой шлюз смотрит уже в магистральную сеть и знает о том, как выбраться в Интернет.



Data Plane

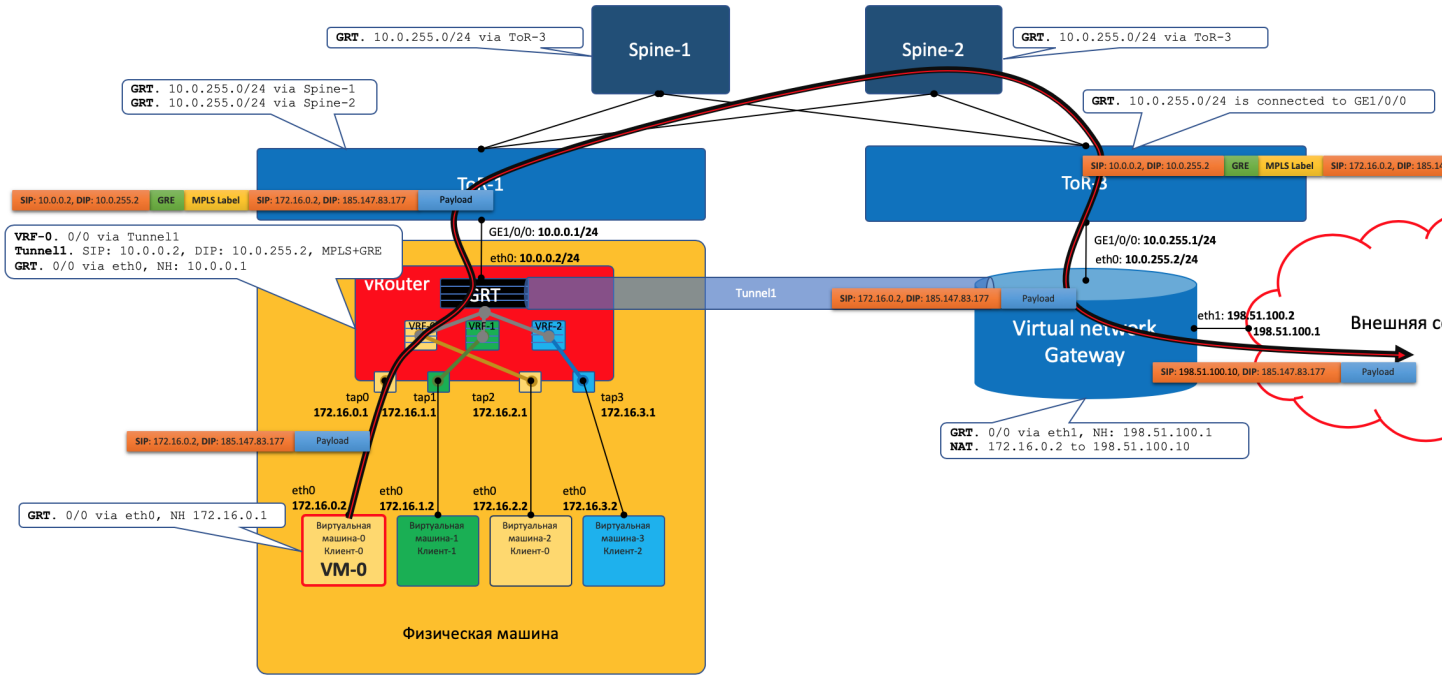
То есть процесс выглядит так:

1. VM-0, имея дефолт всё в тот же vRouter, отправляет пакет с адресатом во внешнем мире (185.147.83.177) в интерфейс eth0.
2. vRouter получает этот пакет и делает лукап адреса назначения в таблице маршрутизации - находит маршрут по умолчанию через шлюз VNGW1 через Tunnel 1.

   Также он видит, что это туннель GRE с SIP 10.0.0.2 и DIP 10.0.255.2, а ещё нужно сначала повесить MPLS-метку данного клиента, которую ожидает VNGW1.
3. vRouter упаковывает первоначальный пакет в заголовки MPLS, GRE и новый IP и отправляет на адрес ToR1 10.0.0.1 по дефолту.
4. Андерлейная сеть доставляет пакет до шлюза VNGW1.
5. Шлюз VNGW1 снимает туннелирующие заголовки GRE и MPLS, видит адрес назначения, консультируется со своей таблицей маршрутизации и понимает, что он направлен в Интернет - значит через Full View или Default. При необходимости производит NAT-трансляцию.
6. От VNGW до бордера может быть обычная IP-сеть, что вряд ли.

   Может быть классическая MPLS сеть (IGP+LDP/RSVP TE), может быть обратно фабрика с BGP LU или GRE-туннель от VNGW до бордера через IP-сеть.

   Как бы то ни было VNGW1 совершает необходимые инкапсуляции и отправляет первоначальный пакет в сторону бордера.



   Трафик в обратную сторону проходит те же шаги в противоположном порядке.

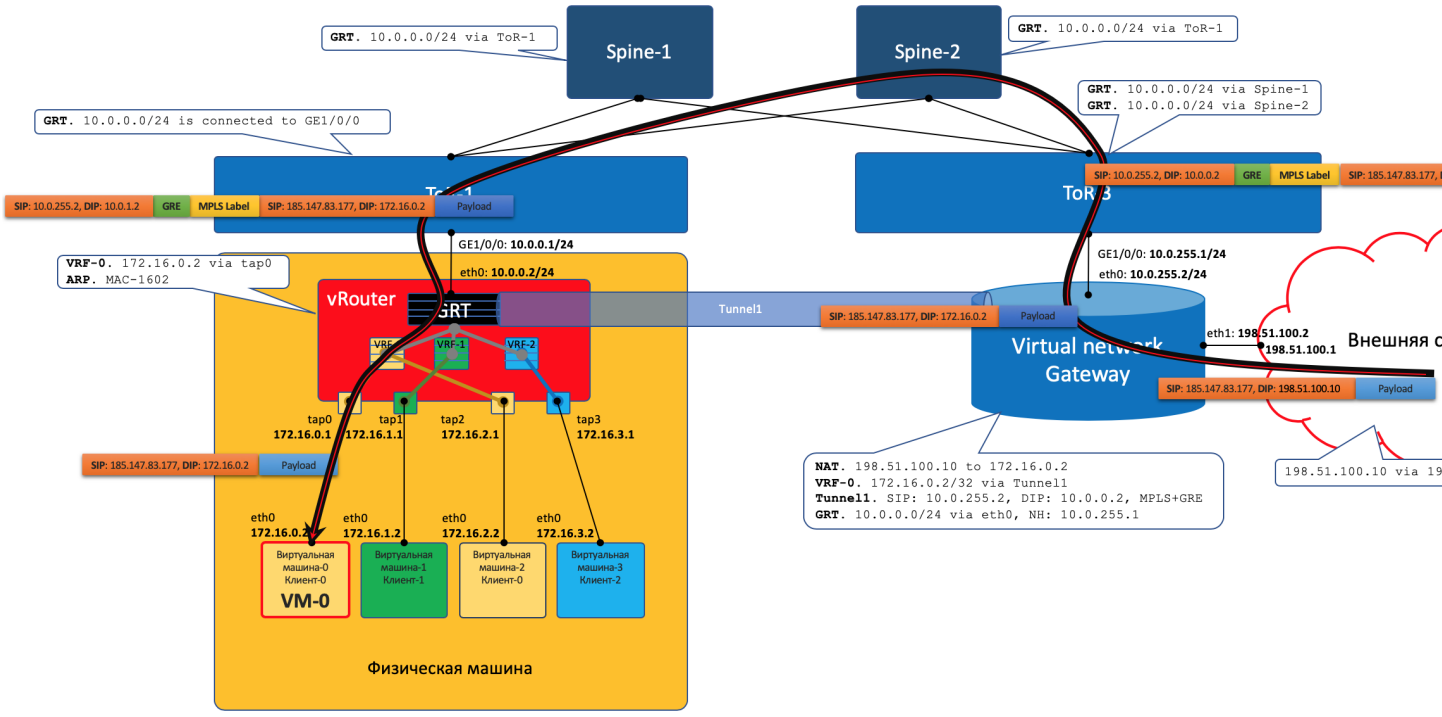
7. Бордер добрасывает пакет до VNGW1
8. Тот его раздевает, смотрит на адрес получателя и видит, что тот доступен через туннель Tunnel1 (MPLSoGRE или MPLSoUDP).

9. Соответственно, вешает метку MPLS, заголовок GRE/UDP и новый IP и отправляет на свой ToR3 10.0.255.1.

   Адрес назначения туннеля - IP-адрес vRouter'а, за которым стоит целевая ВМ - 10.0.0.2.

10. Андерлейная сеть доставляет пакет до нужного vRouter'а.

11. Целевой vRouter снимает GRE/UDP, по MPLS-метке определяет интерфейс и шлёт голый IP-пакет в свой TAP-интерфейс, связанный с eth0 ВМ.



Control Plane

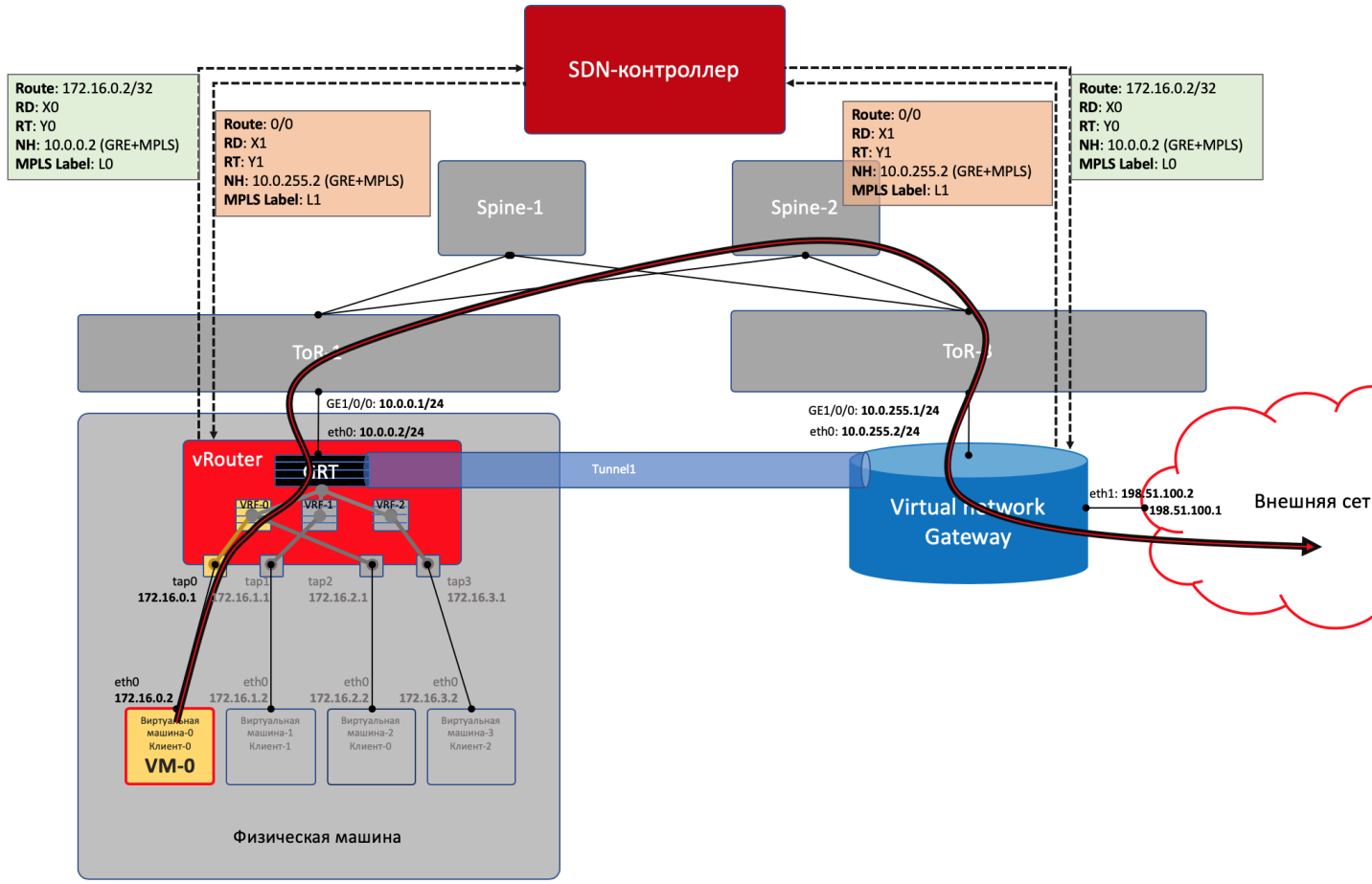
VNGW1 устанавливает BGP-соседство с SDN-контроллером, от которого он получает всю маршрутную информацию о клиентах: за каким IP-адресом (vRouter'ом) находится какой клиент, и какой MPLS-меткой он идентифицируется. Аналогично он сам SDN-контроллеру сообщает дефолтный маршрут с меткой этого клиента, указывая себя в качестве nexthop'а. А дальше этот дефолт приезжает на vRouter'ы.

На VNGW обычно происходит агрегация маршрутов или NAT-трансляция.
И в другую сторону в сессию с бордерами или Route Reflector'ами он отдаёт именно этот агрегированный маршрут. А от них получает маршрут по умолчанию или Full-View, или что-то ещё.

В плане инкапсуляции и обмена трафиком VNGW ничем не отличается от vRouter.
Если немного расширить область, то к VNGW и vRouter'ам можно добавить другие сетевые устройства, такие как файрволы, фермы очистки или обогащения трафика, IPS итд.

И с помощью последовательного создания VRF и правильного анонса маршрутов, можно заставлять трафик петлять так, как вам хочется, что и называется Service Chaining'ом.

То есть и тут SDN-контроллер выступает в роли Route-Reflector'а между VNGW, vRouter'ами и другими сетевыми устройствами.
Но фактически контроллер спускает ещё информацию об ACL и PBR (Policy Based Routing), заставляя отдельные потоки трафика ходить не так, как им велит маршрут.



2.1.5 FAQ

Зачем ты всё время делаешь ремарку GRE/UDP?

Ну, вообще, это, можно сказать, специфично для Tungsten Fabric - можно вообще не брать во-внимание.
Но если брать, то сам TF, ещё будучи OpenContrail'ом поддерживал обе инкапсуляции: MPLS in GRE и MPLS in UDP.
UDP хорош тем, что в Source Port в его заголовке очень легко закодировать хэш-функцию от изначальных IP+Proto+Port, что позволит делать балансировку.

В случае GRE, увы, есть только внешние заголовки IP и GRE, которые одинаковы для всего инкапсулированного трафика и речь о балансировке не идёт - мало кто может заглянуть так глубоко внутрь пакета.

До некоторого времени маршрутизаторы, если и умели в динамические туннели, то только в MPLSoGRE, и только совсем недавно, научились в MPLSoUDP. Поэтому приходится делать всегда ремарку о возможности двух разных инкапсуляций.

Справедливости ради, стоит отметить, что TF вполне поддерживает и L2-связность с помощью VXLAN.

Ты обещал провести параллели с OpenFlow.

Они и правда напрашиваются. vSwitch в том же OpenStack'е делает весьма похожие вещи, используя VXLAN, у которого, кстати, тоже UDP-заголовок.
В Data Plane они работают примерно одинаково, существенно различается Control Plane. Tungsten Fabric использует XMPP для доставки информации о маршрутах на vRouter, в то время, как в OpenStack'е работает Openflow.

А можно чуть больше про vRouter?

Он делится на две части: vRouter Agent и vRouter Forwarder.
Первый запускается в User Space хостовой ОС и общается с SDN-контроллером, обмениваясь информацией о маршрутах, VRF и ACL.
Второй реализует Data Plane - обычно в Kernel Space, но может запускаться и на SmartNIC'ах - сетевых картах с CPU и отдельным программируемым чипом коммутации, что позволяет снять нагрузку с CPU хостовой машины, а сеть сделать быстрее и более предсказуемой.
Ещё возможен сценарий, когда vRouter - это DPDK-приложение в User Space.

vRouter Agent спускает настройки на vRouter Forwarder.

Что за Virtual Network?

Я обмолвился в начале статьи о VRF, что мол каждый тенант привязывается к своему VRF. И если для поверхностного понимания работы оверлейной сети этого было достаточно, то уже на следующей итерации надо делать уточнения.
Обычно в механизмах виртуализации сущность Virtual Network (можно считать это именем собственным) вводится отдельно от клиентов/тенантов/виртуальных машин - вполне себе самостоятельная вещь. А этот Virtual Network через интерфейсы уже можно подключить в один тенант, в другой, в два, да хоть куда. Так, например, реализуется Service Chaining, когда трафик нужно пропустить через определённые ноды в нужной последовательности, просто в правильной последовательности создавая и привзявая Virtual Network'и.
Поэтому как такового прямого соответствия между Virtual Network и тенантом нет.

2.1.6 Заключение

Это весьма поверхностное описание работы виртуальной сети с оверлеем с хоста и SDN-контроллером. Но какую бы платформу виртуализации вы сегодня ни взяли, работать она будет похожим образом, будь то VMWare, ACI, OpenStack, CloudStack, Tungsten Fabric или Juniper Contrail. Они будут отличаться видами инкапсуляций и заголовков, протоколами доставки информации на конечные сетевые устройства, но принцип программно настраиваемой оверлейной сети, работающей поверх сравнительно простой и статичной андерлейной сети останется прежним. Можно сказать, что области создания приватного облака на сегодняшний день SDN на основе оверлейной сети победил. Впрочем это не значит, что Openflow нет места в современном мире - он используется в OpenStacke и в той же VMWare NSX, его, насколько мне известно, использует Google для настройки андерлейной сети.

> Чуть ниже я привёл ссылки на более подробные материалы, если хочется изучить вопрос глубже.

А что там наш Underlay? А в общем-то ничего. Он всю дорогу не менялся. Всё, что ему нужно делать в случае оверлея с хоста - это обновлять маршруты и ARP'ы по мере появления и исчезновения vRouter/VNGW и таскать пакеты между ними.

Давайте сформулируем список требований к Underlay-сети.

1. Уметь в какой-то протокол маршрутизации, в нашей ситуации - BGP.
2. Иметь широкую полосу, желательно без переподписки, чтобы не терялись пакеты из-за перегрузок.
3. Поддерживать ECMP - неотъемлемая часть фабрики.
4. Уметь обеспечить QoS, в том числе хитрые штуки, вроде ECN.
5. Поддерживать NETCONF - задел на будущее.

Работе самой Underlay-сети я посвятил здесь совсем мало времени. Это потому, что далее в серии я именно на ней и сосредоточусь, а Overlay мы будем затрагивать только вскользь.

Очевидно, я сильно ограничиваю нас всех, используя в качестве примера сеть ДЦ, построенную на фабрике Клоза с чистой IP-маршрутизацией и оверлеем с хоста. Однако я уверен, что любую сеть, имеющую дизайн, можно описать в формальных терминах и автоматизировать. Просто я здесь преследую целью разобраться в подходах к автоматизации, а не запутать всех вообще, решая задачу в общем виде.

В рамках АДСМ мы с Романом Горге планируем опубликовать отдельный выпуск про виртуализацию вычислительных мощностей и её взаимодействие с виртуализацией сети. Оставайтесь на связи.

Полезные ссылки

- Tungsten Fabric Archvitecture.
- 6 часов про Яндекс.Облако, где в том числе затрагивается виртуальная сеть на TF.
- What Is Open vSwitch?
- Введение в VxLAN.
- RFC 7348. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks.
- Scaleway approach to VXLAN EVPN Fabric. Тут рассказывается про всю сеть ДЦ, включая Underlay, Overlay, подходы к мульти-хоумингу и управлению.

Спасибы

- Роману Горге - бывшему ведущему подкаста linkmeup, а ныне эксперту в области облачных платформ. За комментарии и правки. Ну и ждём в скором будущем его более глубокой статьи о виртуализации.
- Александру Шалимову - моему коллеге и эксперту в области разработки виртуальных сетей. За комментарии и правки.
- Валентину Синицыну- моему коллеге и эксперту в области Tungsten Fabric . За комментарии и правки.
- Артёму Чернобаю - иллюстратору linkmeup. За КДПВ.
- Александру Лимонову. За мем «automato».

2.2 Основы виртуализации

Автор этой статьи - Роман Горге - бывший ведущий linkmeup.

Предыдущая статья рассматривала архитектуру виртуализированной сети, underlay-overlay, путь пакета между VM и прочее. В данной статье мы затронем (или попытаемся затронуть) вопросы а как собственно происходит виртуализация сетевых функций, как реализован backend основных продуктов, обеспечивающих запуск и управление VM, а также как работает виртуальный свитчинг (OVS и Linux bridge).

Тема виртуализации широка и глубока, объяснить все детали работы гипервизора невозможно (да и не нужно). Мы ограничимся минимальным набором знаний необходимым для понимания работы любого виртуализированного решения, не обязательно Telco.

2.2.1 Введение и краткая история виртуализации

История современных технологий виртуализации берет свое начало в 1999 году, когда молодая компания VMWare выпустила продукт под названием VMWare Workstation. Это был продукт обеспечивающий виртуализацию desktop/client приложений. Виртуализация серверной части пришла несколько позднее в виде продукта ESX Server, который в дальнейшем эволюционировал в ESXi (i означает integrated) - это тот самый продукт, который используется повсеместно, как в IT так и в Telco, как гипервизор серверных приложений.

На стороне Opensource два основных проекта принесли виртуализацию в Linux:

- KVM (Kernel-based Virtual Machine) - модуль ядра Linux, который позволяет kernel работать как гипервизор (создает необходимую инфраструктуру для запуска и управления VM). Был добавлен в версии ядра 2.6.20 в 2007 году.
- QEMU (Quick EMUlator - отсюда и страус на лого) - непосредственно эмулирует железо для виртуальной машины (CPU, Disk, RAM, что угодно, включая USB порт) и используется совместно с KVM для достижения почти «native» производительности.

  > На самом деле на сегодняшний момент вся функциональность KVM доступна в QEMU, но это не принципиально, так как большая часть пользователей виртуализации на Linux не использует напрямую KVM/QEMU, а обращается к ним как минимум через один уровень абстракции, но об этом позже.

virsh
FD KVM QEMU
VF TUN/TAP PCI-PT
VT-x VT-d NUMA SR-IOV
libvirt

Сегодня VMWare ESXi и Linux QEMU/KVM это два основных гипервизора, которые доминируют на рынке. Они же являются представителями двух разных типов гипервизоров:

- Type 1 - гипервизор запускается непосредственно на железе (bare-metal). Таковым является VMWare ESXi.
- Type 2 - гипервизор запускается внутри Host OS (операционной системы). Таковым является Linux KVM.

Обсуждение что лучше, а что хуже выходит за рамки данной статьи.

Производители железа также должны были сделать свою часть работы, дабы обеспечить приемлемую производительность.

Пожалуй, наиболее важной и самой широко используемой является технология Intel VT (Virtualization Technology) - набор расширений, разработанных Intel для своих x86 процессоров, которые используются для эффективной работы гипервизора (а в некоторых случаях необходимы, так, например, KVM не заработает без включенного VT-x и без него гипервизор вынужден заниматься чисто софтверной эмуляцией, без аппаратного ускорения). Наиболее известны два из этих расширений - VT-x и VT-d. Первое важно для улучшения производительности CPU при виртуализации, так как обеспечивает аппаратную поддержку некоторых ее функций (с VT-x 99.9% Gust OS кода выполняется прямо на физическом процессоре, делая выходы для эмуляции только в самых необходимых случаях)), второе для подключения физических устройств напрямую в виртуальную машину (для проброса виртуальных функций (VF) SRIOV, например, VT-d должен быть включен).

Следующей важной концепцией является отличие полной виртуализации (full virtualization) от пара-виртуализации (para-virtualization).

Полная виртуализация - это хорошо, это позволяет запускать какую угодно операционную систему на каком угодно процессоре, однако, это крайне неэффективно и абсолютно не подходит для высоконагруженных систем.

Пара-виртуализация, если коротко, это когда Guest OS понимает что она запущена в виртуальной среде и кооперируется с гипервизором для достижения большей эффективности. То есть появляется guest-hypervisor интерфейс.

Подавляющее большинство используемых операционных систем сегодня имеют поддержку пара-виртуализации - в Linux kernel это появилось начиная с ядра версии 2.6.20.

Для работы виртуальной машины нужны не только виртуальный процессор (vCPU) и виртуальная память (RAM), требуется также эмуляция PCI-устройств. То, есть по сути, требуется набор драйверов для управления виртуальными сетевыми интерфейсами, дисками и тд.

В гипервизоре Linux KVM данная задача была решена посредством внедрения **virtio** - фреймворка для разработки и использования виртуализированных устройств ввода/вывода.

Virtio представляет из себя дополнительный уровень абстракции, который позволяет эмулировать различные I/O устройства в пара-виртуализированном гипервизоре, предоставляя в сторону виртуальной машины единый и стандартизированный интерфейс. Это позволяет переиспользовать код virtio-драйвера для различных по своей сути устройств. Virtio состоит из:

- Front-end driver - то что находится в виртуальной машине
- Back-end driver - то что находится в гипервизоре
- Transport driver - то что связывает backend и frontend

Эта модульность позволяет изменять технологии, применяемые в гипервизоре, не затрагивая драйверы в виртуальной машине (этот момент очень важен для технологий сетевой акселерации и Cloud-решений в целом, но об этом позже).

То есть существует связь guest-hypervisor, когда Guest OS «знает» о том, что запущена в виртуальной среде.

> Если вы хоть раз писали вопрос в RFP или отвечали на вопрос в RFP «Поддерживается ли в вашем продукте virtio?» Это как раз было о поддержке front-end virtio драйвера.

2.2.2 Типы виртуальных ресурсов - compute, storage, network

Из чего же состоит виртуальная машина? Выделяют три основных вида виртуальных ресурсов:

- compute - процессор и оперативная память
- storage - системный диск виртуальной машины и блочные хранилища
- network - сетевые карты и устройства ввода/вывода

Compute

CPU

Теоретически QEMU способен эмулировать любой тип процессора и соотвествующие ему флаги и функциональность, на практике используют либо host-model и точечно выключают флаги перед передачей в Guest OS либо берут named-model и точечно включаютвыключают флаги.

По умолчанию QEMU будет эмулировать процессор, который будет распознан Guest OS как QEMU Virtual CPU. Это не самый оптимальный тип процессора, особенно если приложение, работающее в виртуальной машине, использует CPU-флаги для своей работы. Подробнее о разных моделях CPU в QEMU.

QEMU/KVM также позволяет контролировать топологию процессора, количество тредов, размер кэша, привязывать vCPU к физическому ядру и много чего еще.

Нужно ли это для виртуальной машины или нет, зависит от типа приложения, работающего в Guest OS. Например, известный факт, что для приложений, выполняющих обработку пакетов с высоким PPS, важно делать **CPU pinning**, то есть не позволять передавать физический процессор другим виртуальным машинам.

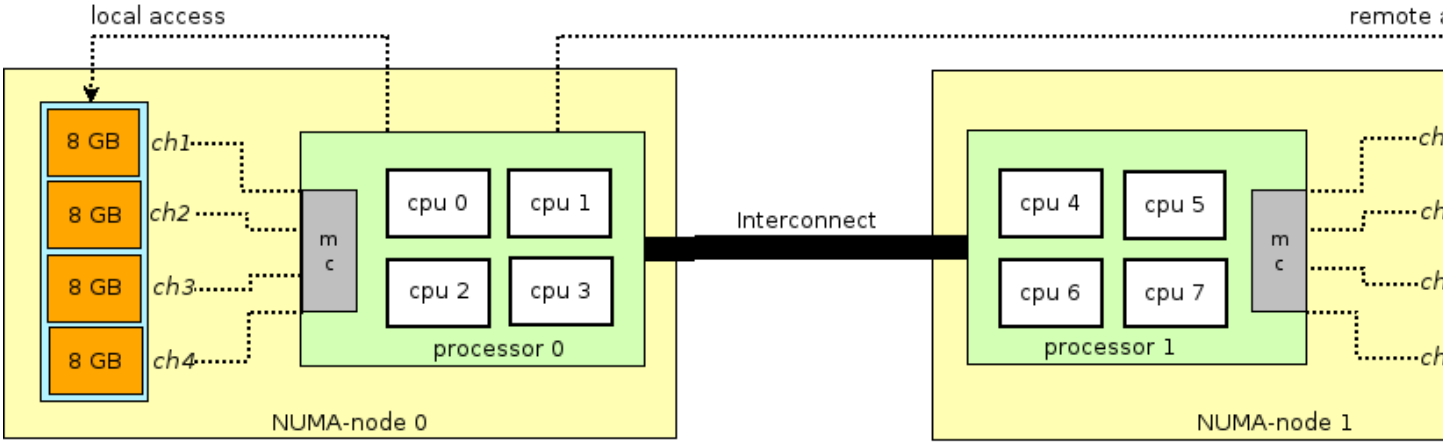
Memory

Далее на очереди оперативная память - RAM. С точки зрения Host OS запущенная с помощью QEMU/KVM виртуальная машина ничем не отличается от любого другого процесса, работающего в user-space операционной системы. Соотвественно и процесс выделения памяти виртуальной машине выполняется теми же вызовами в kernel Host OS, как если бы вы запустили, например, Chrome браузер.

> Перед тем как продолжить повествование об оперативной памяти в виртуальных машинах, необходимо сделать отступление и объяснить термин **‘NUMA <https://ru.wikipedia.org/wiki/Non-Uniform_Memory_Access>‘** - Non-Uniform Memory Access.

Архитектура современных физических серверов предполагает наличие двух или более процессоров (CPU) и ассоциированной с ней оперативной памятью (RAM). Такая связка процессор + память называется узел или нода (node). Связь между различными NUMA nodes осуществляется посредством специальной шины - **QPI** (QuickPath Interconnect)

Выделяют локальную NUMA node - когда процесс, запущенный в операционной системе, использует процессор и оперативную память, находящуюся в одной NUMA node, и удаленную NUMA node - когда процесс, запущенный в операционной системе, использует процессор и оперативную память, находящиеся в разных NUMA nodes, то есть для взаимодействия процессора и памяти требуется передача данных через QPI шину.



С точки зрения виртуальной машины память ей уже выделена на момент ее запуска, однако в реальности это не так, и kernel Host OS выделяет процессу QEMU/KVM новые участки памяти по мере того как приложение в Guest OS запрашивает дополнительную память (хотя тут тоже может быть исключение, если прямо указать QEMU/KVM выделить всю память виртуальной машине непосредственно при запуске).

Память выделяется не байт за байтом, а определенным размером - **page**. Размер page конфигурируем и теоретически может быть любым, но на практике используется размер 4kB (по умолчанию), 2MB и 1GB. Два последних размера называются **HugePages** и часто используются для выделения памяти для memory intensive виртуальных машин. Причина использования HugePages в процессе поиска соответствия между виртуальным адресом page и физической памятью в **Translation Lookaside Buffer** (**‘TLB <https://en.wikipedia.org/wiki/Translation_lookaside_buffer>‘_**), который в свою очередь ограничен и хранит информацию только о последних использованных pages. Если информации о нужной page в TLB нет, происходит процесс, называемый **TLB miss**, и требуется задействовать процессор Host OS для поиска ячейки физической памяти, соответствующей нужной page.

Данный процесс неэффективен и медлителен, поэтому и используется меньшее количество pages большего размера. QEMU/KVM также позволяет эмулировать различные NUMA-топологии для Guest OS, брать память для виртуальной машины только из определенной NUMA node Host OS и так далее. Наиболее распространенная практика - брать память для виртуальной машины из NUMA node локальной по отношению к процессорам, выделенным для виртуальной машины. Причина - желание избежать лишней нагрузки на **QPI** шину, соединяющую CPU sockets физического сервера (само собой, это логично если в вашем сервере 2 и более sockets).

Storage

Как известно, оперативная память потому и называется оперативной, что ее содержимое исчезает при отключении питания или перезагрузке операционной системы. Чтобы хранить информацию, требуется постоянное запоминающее устройство (ПЗУ) или **persistent storage**. Существует два основных вида persistent storage:

- Block storage (блоковое хранилище) - блок дискового пространства, который может быть использован для установки файловой системы и создания партиций. Если грубо, то можно воспринимать это как обычный диск.
- Object storage (объектное хранилище) - информация может быть сохранена только в виде объекта (файла), доступного по HTTP/HTTPS. Типичными примерами объектного хранилища являются AWS S3 или Dropbox.

Виртуальная машина нуждается в **persistent storage**, однако, как это сделать, если виртуальная машина «живет» в оперативной памяти Host OS? (кстати, именно поэтому невозможно запустить виртуальную машину с оперативной памятью меньше, чем размер ее qcow2 образа). Если вкратце, то любое обращение Guest OS к контроллеру виртуального диска перехватывается QEMU/KVM и трансформируется в запись на физический диск Host OS. Этот метод неэффективен, и поэтому здесь так же как и для сетевых устройств используется virtio-драйвер вместо полной эмуляции IDE или iSCSI-устройства. Подробнее об этом можно почитать здесь. Таким образом виртуальная машина обращается к своему виртуальному диску через virtio-драйвер, а далее QEMU/KVM делает так, чтобы переданная информация записалась на физический диск. Важно понимать, что в Host OS дисковый backend может быть реализован в виде CEPH, NFS или iSCSI-полки.

Наиболее простым способом эмулировать persistent storage является использование файла в какой-либо директории Host OS как дискового пространства виртуальной машины. QEMU/KVM поддерживает множество различных форматов такого рода файлов - raw, vdi, vmdk и прочие. Однако наибольшее распространение получил формат **qcow2** (QEMU copy-on-write version 2). В общем случае, qcow2 представляет собой определенным образом структурированный файл без какой-либо операционной системы. Большое количество виртуальных машин распространяется именно в виде qcow2-образов (images) и являются копией системного диска виртуальной машины, упакованной в qcow2-формат. Это имеет ряд преимуществ - qcow2-кодирование занимает гораздо меньше места, чем raw копия диска байт в байт, QEMU/KVM умеет изменять размер qcow2-файла (resizing), а значит имеется возможность изменить размер системного диска виртуальной машины, также поддерживается AES шифрование qcow2 (это имеет смысл, так как образ виртуальной машины может содержать интеллектуальную собственность).

Далее, когда происходит запуск виртуальной машины, QEMU/KVM использует qcow2-файл как системный диск (процесс загрузки виртуальной машины я опускаю здесь, хотя это тоже является интересной задачей), а виртуальная машина имеет возможность считать/записать данные в qcow2-файл через virtio-драйвер. Таким образом и работает процесс снятия образов виртуальных машин, поскольку в любой момент времени qcow2-файл содержит полную копию системного диска виртуальной машины, и образ может быть использован для резервного копирования, переноса на другой хост и прочее.

В общем случае этот qcow2-файл будет определяться в Guest OS как */dev/vda*-устройство, и Guest OS произведет разбиение дискового пространства на партиции и установку файловой системы. Аналогично, следующие qcow2-файлы, подключенные QEMU/KVM как */dev/vdX* устройства, могут быть использованы как **block storage** в виртуальной машине для хранения информации (именно так и работает компонент Openstack Cinder).

Network

Последним в нашем списке виртуальных ресурсов идут сетевые карты и устройства ввода/вывода. Виртуальная машина, как и физический хост, нуждается в **PCI/PCIe-шине** для подключения устройств

ввода/вывода. QEMU/KVM способен эмулировать разные типы чипсетов - q35 или i440fx (первый поддерживает - PCIe, второй - legacy PCI), а также различные PCI-топологии, например, создавать отдельные PCI-шины (PCI expander bus) для NUMA nodes Guest OS.

После создания PCI/PCIe шины необходимо подключить к ней устройство ввода/вывода. В общем случае это может быть что угодно - от сетевой карты до физического GPU. И, конечно же, сетевая карта, как полностью виртуализированная (полностью виртуализированный интерфейс e1000, например), так и пара-виртуализированная (virtio, например) или физическая NIC. Последняя опция используется для data-plane виртуальных машин, где требуется получить line-rate скорости передачи пакетов - маршрутизаторов, файрволов и тд.

Здесь существует два основных подхода - **PCI passthrough** и **SR-IOV**. Основное отличие между ними - для PCI-PT используется драйвер только внутри Guest OS, а для SRIOV используется драйвер Host OS (для создания **VF - Virtual Functions**) и драйвер Guest OS для управления SR-IOV VF. Более подробно об PCI-PT и SRIOV отлично написал Juniper.

> Для уточнения стоит отметить что, PCI passthrough и SR-IOV это дополняющие друг друга технологии. SR-IOV это нарезка физической функции на виртуальные функции. Это выполняется на уровне Host OS. При этом Host OS видит виртуальные функции как еще одно PCI/PCIe устройство. Что он дальше с ними делает - не важно.
>
> А PCI-PT это механизм проброса любого Host OS PCI устройства в Guest OS, в том числе виртуальной функции, созданной SR-IOV устройством

Таким образом мы рассмотрели основные виды виртуальных ресурсов и следующим шагом необходимо понять как виртуальная машина общается с внешним миром через сеть.

2.2.3 Виртуальная коммутация

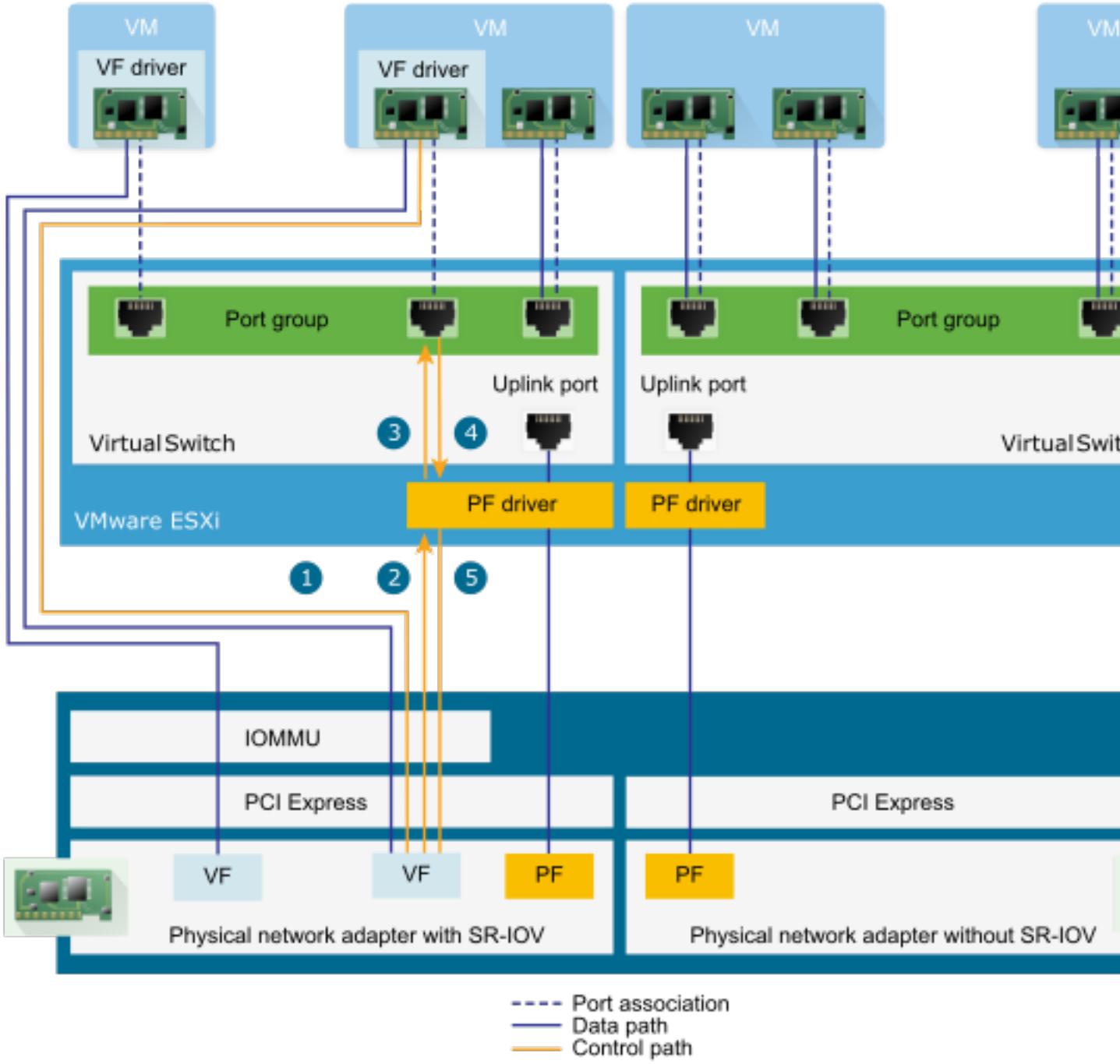
Если есть виртуальная машина, а в ней есть виртуальный интерфейс, то, очевидно, возникает задача передачи пакета из одной VM в другую. В Linux-based гипервизорах (KVM, например) эта задача может решаться с помощью Linux bridge, однако, большое распространение получил проект ‘Open vSwitch <https://www.openvswitch.org>‘_(OVS). Есть несколько основных функциональностей, которые позволили OVS широко распространиться и стать de-facto основным методом коммутации пакетов, который используется во многих платформах облачных вычислений(например, Openstack) и виртуализированных решениях.

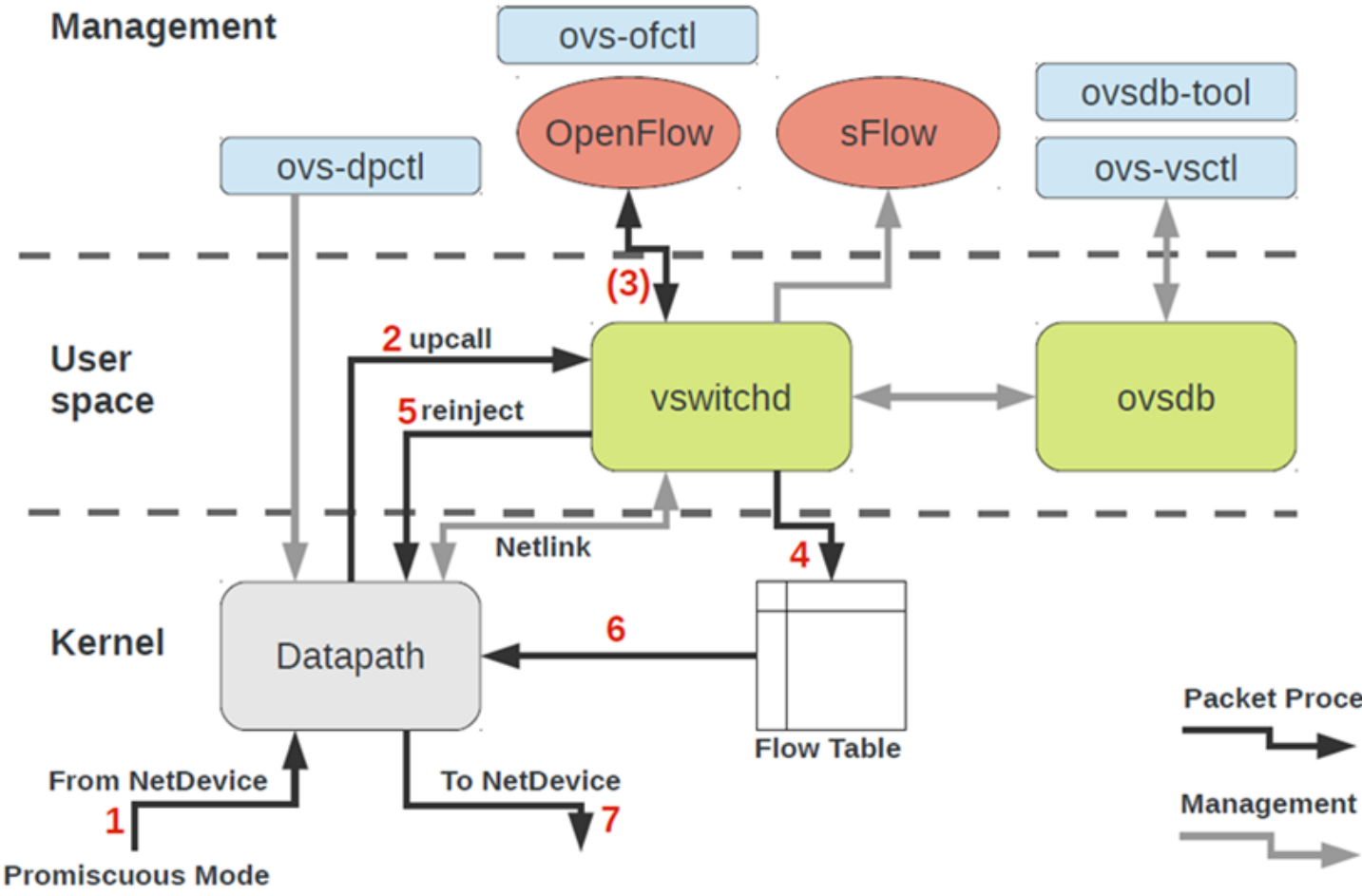
- Передача сетевого состояния - при миграции VM между гипервизорами возникает задача передачи ACL, QoSs, L2/L3 forwarding-таблиц и прочего. И OVS умеет это.
- Реализация механизма передачи пакетов (datapath) как в kernel, так и в user-space
- CUPS (Control/User-plane separation) архитектура - позволяет перенести функциональность обработки пакетов на специализированный chipset (Broadcom и Marvell chipset, например, могут такое), управляя им через control-plane OVS.
- Поддержка методов удаленного управления трафиком - протокол OpenFlow (привет, SDN).

Архитектура OVS на первый взгляд выглядит довольно страшно, но это только на первый взгляд.

Для работы с OVS нужно понимать следующее:

- **Datapath** - тут обрабатываются пакеты. Аналогия - switch-fabric железного коммутатора. Datapath включает в себя приём пакетов, обработку заголовков, поиск соответствий по таблице flow, который в Datapath уже запрограммирован. Если OVS работает в kernel, то выполнен в виде модуля ядра. Если OVS работает в user-space, то это процесс в user-space Linux.


VM
VF driver
VM
VF driver
VM
VM
Port group
Port group
Uplink port
Uplink port
Virtual Switch
Virtual Swit
3
4
VMware ESXi
PF driver
PF driver
1
2
5
IOMMU
PCI Express
PCI Express
VF
VF
PF
PF
Physical network adapter with SR-IOV
Physical network adapter without SR-IOV
Port association
Data path
Control path


Management
ovs-ofctl
OpenFlow
sFlow
ovsdb-tool
ovs-dpctl
ovs-vsctl
(3)
User space
2 upcall
vswitchd
ovsdb
5 reinject
Netlink
4
Kernel
Datapath
6
Flow Table
Packet Proce
From NetDevice
To NetDevice
Management
1
7
Promiscuous Mode

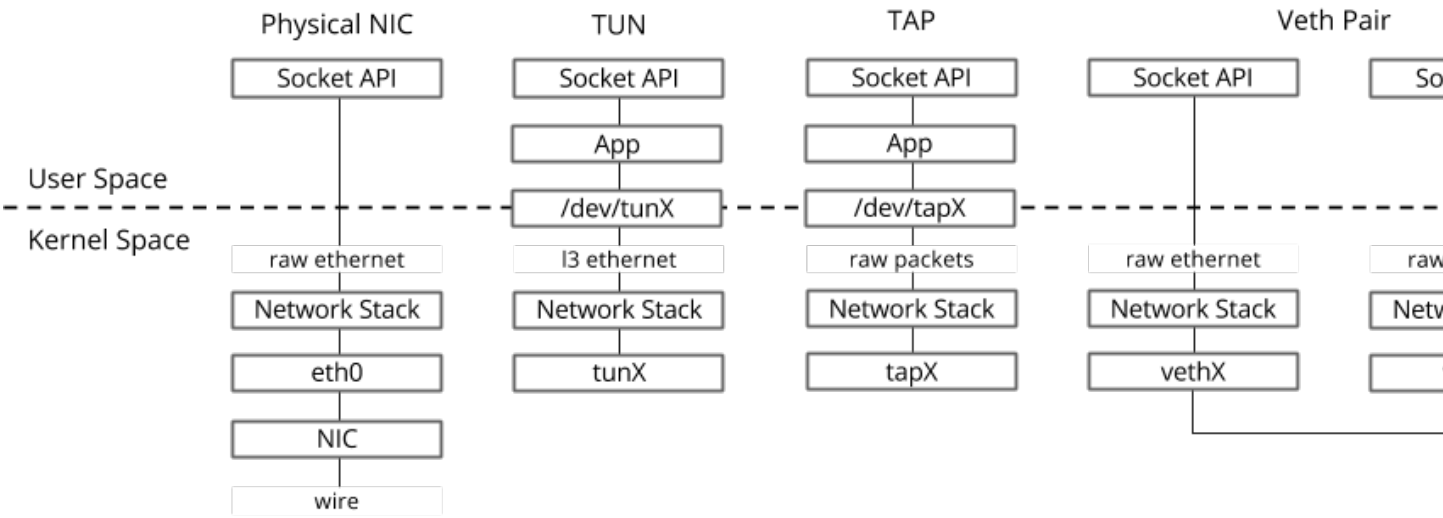
- **vswitchd** и **ovsdb** - демоны в user-space, то что реализует непосредственно сам функциональность коммутатора, хранит конфигурацию, устанавливает flow в datapath и программирует его.
- Набор инструментов для настройки и траблшутинга OVS - **ovs-vsctl, ovs-dpctl, ovs-ofctl, ovs-appctl**. Все то, что нужно, чтобы прописать в ovsdb конфигурацию портов, прописать какой flow куда должен коммутироваться, собрать статистику и прочее. Добрые люди написали статью по этому поводу.

Каким же образом сетевое устройство виртуальной машины оказывается в OVS?

Для решения данной задачи нам необходимо каким-то образом связать между собой виртуальный интерфейс, находящийся в user-space операционной системы с datapath OVS, находящимся в kernel.

В операционной системе Linux передача пакетов между kernel и user-space-процессами осуществляется посредством двух специальных интерфейсов. Оба интерфейса использует запись/чтение пакета в/из специальный файл для передачи пакетов из user-space-процесса в kernel и обратно - file descriptor (FD) (это одна из причин низкой производительности виртуальной коммутации, если datapath OVS находится в kernel - каждый пакет требуется записать/прочесть через FD)

- **TUN** (tunnel) - устройство, работающее в L3 режиме и позволяющее записывать/считывать только IP пакеты в/из FD.
- **TAP** (network tap) - то же самое, что и tun интерфейс + умеет производить операции с Ethernet-фреймами, т.е. работать в режиме L2.



Именно поэтому при запущенной виртуальной машине в Host OS можно увидеть созданные TAP-интерфейсы командой *ip link* или *ifconfig* - это «ответная» часть virtio, которая «видна» в kernel Host OS. Также стоит обратить внимание, что TAP-интерфейс имеет тот же MAC-адрес что и virtio-интерфейс в виртуальной машине.

TAP-интерфейс может быть добавлен в OVS с помощью команд *ovs-vsctl* - тогда любой пакет, скоммутированный OVS в TAP-интерфейс, будет передан в виртуальную машину через file descriptor.

> Реальный порядок действий при создании виртуальной машины может быть разным, т.е. сначала можно создать OVS bridge, потом указать виртуальной машине создать интерфейс, соединенный с этим OVS, а можно и наоборот.

Теперь, если нам необходимо получить возможность передачи пакетов между двумя и более виртуальными машинами, которые запущены на одном гипервизоре, нам потребуется лишь создать OVS bridge и добавить в него TAP-интерфейсы с помощью команд ovs-vsctl. Какие именно команды для этого нужны легко гуглится.

На гипервизоре может быть несколько OVS bridges, например, так работает Openstack Neutron, или же виртуальные машины могут находиться в разных namespace для реализации multi-tenancy.

А если виртуальные машины находятся в разных OVS bridges?

Для решения данной задачи существует другой инструмент - **veth pair**. Veth pair может быть представлен как пара сетевых интерфейсов, соединенных кабелем - все то, что «влетает» в один интерфейс, «вылетает» из другого. Veth pair используется для соединения между собой нескольких OVS bridges или Linux bridges. Другой важный момент что части veth pair могут находиться в разных namespace Linux OS, то есть veth pair может быть также использован для связи namespace между собой на сетевом уровне.

2.2.4 Инструменты виртуализации - libvirt, virsh и прочее

В предыдущих главах мы рассматривали теоретические основы виртуализации, в этой главе мы поговорим об инструментах, которые доступны пользователю непосредственно для запуска и изменения виртуальных машин на KVM-гипервизоре. Остановимся на трех основных компонентах, которые покрывают 90 процентов всевозможных операций с виртуальными машинами:

- libvirt
- virsh CLI
- virt-install

> Конечно, существует множество других утилит и CLI-команд, которые позволяют управлять гипервизором, например, можно напрямую пользоваться командами qemu_system_x86_64 или графическим интерфейсом virt manager, но это скорее исключение. К тому же существующие сегодня Cloud-платформы, Openstack, например, используют как раз libvirt.

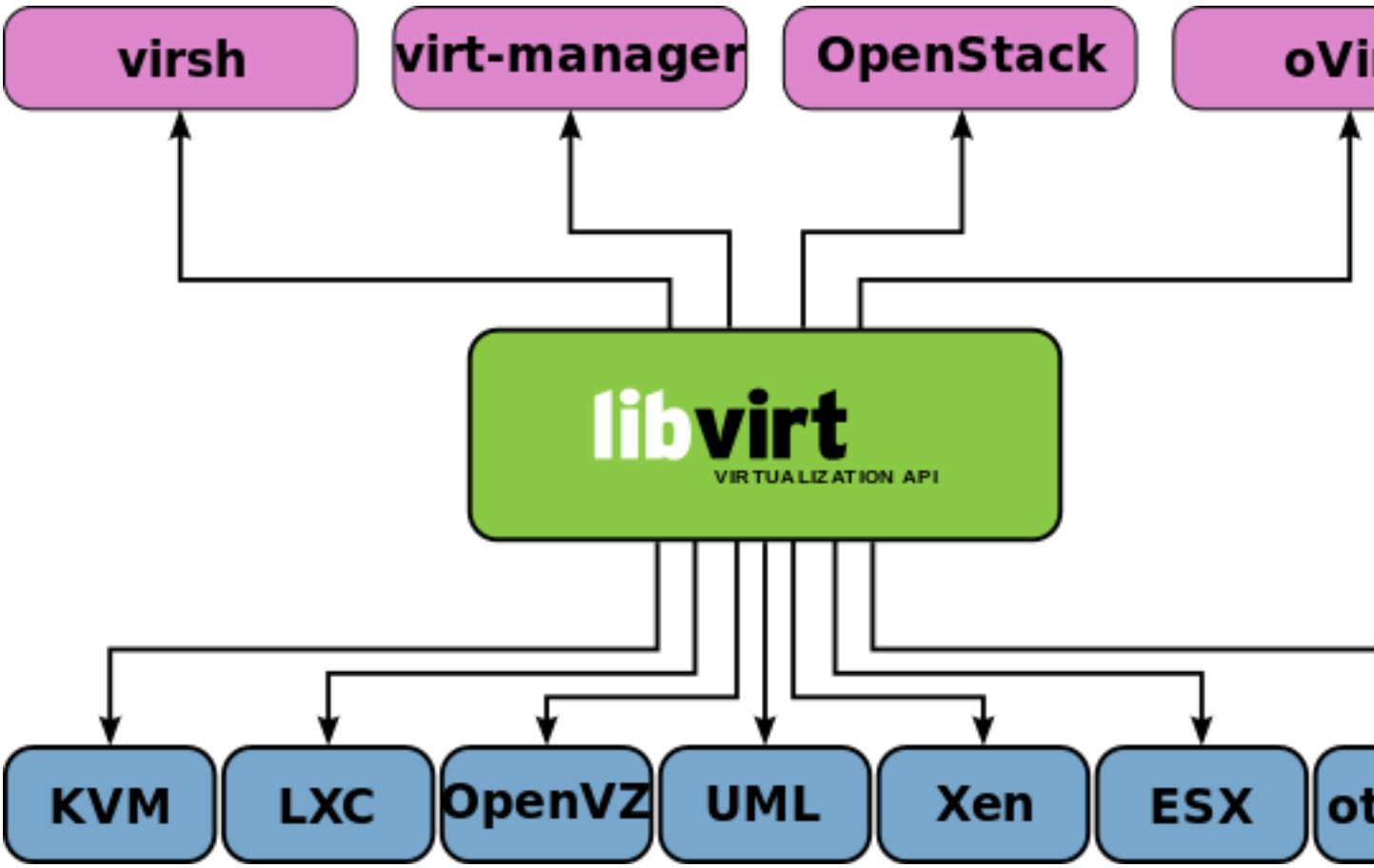
libvirt

libvirt - это масштабный open-source проект, который занимается разработкой набора инструментов и драйверов для управления гипервизорами. Он поддерживает не только QEMU/KVM, но и ESXi, LXC и много чего еще. Основная причина его популярности - структурированный и понятный интерфейс взаимодействия через набор XML-файлов плюс возможность автоматизации через API. Стоит оговориться что libvirt не описывает все возможные функции гипервизора, он лишь предоставляет удобный интерфейс использования **полезных**, с точки зрения участников проекта, функции гипервизора.

И да, libvirt это де-факто стандарт в мире виртуализации сегодня. Только взгляните на список приложений, которые используют libvirt.

Хорошая новость про libvirt - все нужные пакеты уже предустановлены во всех наиболее часто используемых Host OS - Ubuntu, CentOS и RHEL, поэтому, скорее всего, собирать руками нужные пакеты и компилировать libvirt вам не придется. В худшем случае придется воспользоваться соответствующим пакетным инсталлятором (apt, yum и им подобные).

При первоначальной установке и запуске libvirt по умолчанию создает Linux bridge virbr0 и его минимальную конфигурацию.


virsh
virt-manager
OpenStack
oVi
libvirt
VIRTUALIZATION API
KVM
LXC
OpenVZ
UML
Xen
ESX
ot

> Именно поэтому при установке Ubuntu Server, например, вы увидите в выводе команды ifconfig Linux bridge virbr0 - это результат запуска демона libvirtd

Этот Linux bridge не будет подключен ни к одному физическому интерфейсу, однако, может быть использован для связи виртуальных машин внутри одного гипервизора. Libvirt безусловно может быть использован вместе с OVS, однако, для этого пользователь должен самостоятельно создать OVS bridges с помощью соответствующих OVS-команд.

Любой виртуальный ресурс, необходимый для создания виртуальной машины (compute, network, storage) представлен в виде объекта в libvirt. За процесс описания и создания этих объектов отвечает набор различных XML-файлов.

Детально описывать процесс создания виртуальных сетей и виртуальных хранилищ не имеет особого смысла, так как эта прикладная задача хорошо описана в документации libvirt:

- Networking
- Storage

Сама виртуальная машина со всеми подключенными PCI-устройствами в терминологии libvirt называется domain. Это тоже объект внутри libvirt, который описывается отдельным XML-файлом.

Этот XML-файл и является, строго говоря, виртуальной машиной со всеми виртуальными ресурсами - оперативная память, процессор, сетевые устройства, диски и прочее. Часто данный XML-файл называют libvirt XML или dump XML. Вряд ли найдется человек, который понимает все параметры libvirt XML, однако, это и не требуется, когда есть документация.

В общем случае, libvirt XML для Ubuntu Desktop Guest OS будет довольно прост - 40-50 строчек. Поскольку вся оптимизация производительности описывается также в libvirt XML (NUMA-топология, CPU-топологии, CPU pinning и прочее), для сетевых функций libvirt XML может быть очень сложен и содержать несколько сот строк. Любой производитель сетевых устройств, который поставляет свое ПО в виде виртуальных машин, имеет рекомендованные примеры libvirt XML.

virsh CLI

Утилита virsh - «родная» командная строка для управления libvirt. Основное ее предназначение - это управление объектами libvirt, описанными в виде XML-файлов. Типичными примерами являются операции start, stop, define, destroy и так далее. То есть жизненный цикл объектов - life-cycle management.

Описание всех команд и флагов virsh также доступно в документации libvirt.

virt-install

Еще одна утилита, которая используется для взаимодействия с libvirt. Одно из основных преимуществ - можно не разбираться с XML-форматом, а обойтись лишь флагами, доступными в virsh-install. Второй важный момент - море примеров и информации в Сети.

Таким образом какой бы утилитой вы ни пользовались, управлять гипервизором в конечном счете будет именно libvirt, поэтому важно понимать архитектуру и принципы его работы.

2.2.5 Заключение

В данной статье мы рассмотрели минимальный набор теоретических знаний, который необходим для работы с виртуальными машинами. Я намеренно не приводил практических примеров и выводов команд, поскольку таких примеров можно найти сколько угодно в Сети, и я не ставил перед собой задачу написать «step-by-step guide». Если вас заинтересовала какая-то конкретная тема или технология, оставляйте свои комментарии и пишите вопросы.

Полезные ссылки

- Understanding QEMU Devices
- KVM/SR-IOV

Спасибы

- Александру Шалимову - моему коллеге и эксперту в области разработки виртуальных сетей. За комментарии и правки.
- Евгению Яковлеву - моему коллеге и эксперту в области виртуализации за комментарии и правки.

Глава 3

Часть 2. Дизайн сети

В первых двух статьях я поднял вопрос автоматизации и набросал её фреймворк, во второй сделал отступление в виртуализацию сети, как первый подход к автоматизации настройки сервисов. А теперь пришло время нарисовать схему физической сети.

Если вы не на короткой ноге с устройством сетей датацентров, то я настоятельно рекомендую начать со статьи о них.

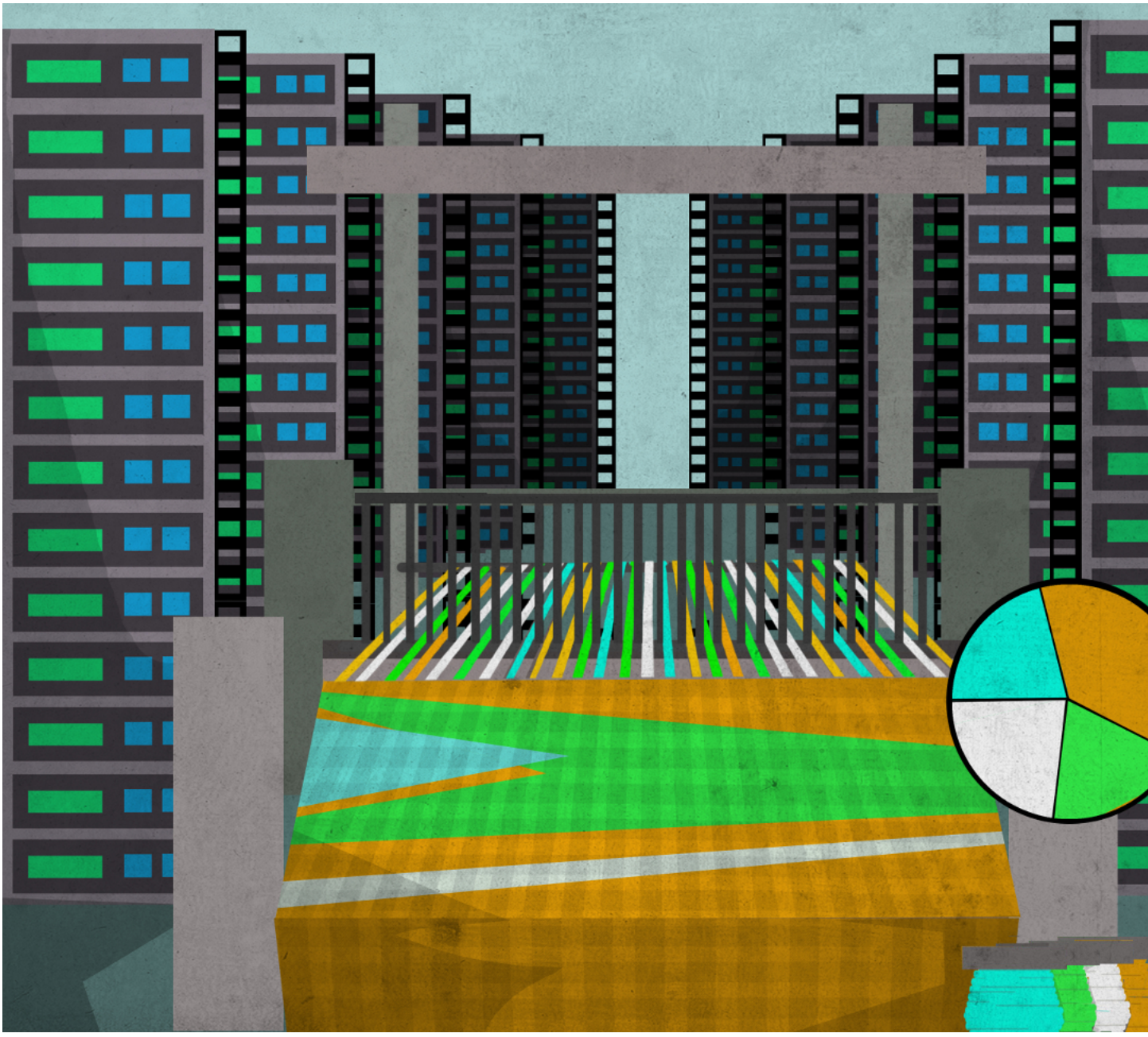
Описанные в этой серии практики должны быть применимы к сети любого типа, любого масштаба с любым многообразием вендоров (нет). Однако нельзя описать универсальный пример применения этих подходов. Поэтому я остановлюсь на современной архитектуре сети ДЦ: Фабрике Клоза. DCI сделаем на MPLS L3VPN. Поверх физической сети работает Overlay-сеть с хоста (это может быть VXLAN OpenStack'a или Tungsten Fabric или что угодно другое, что требует от сети только базовой IP-связности).

В этом случае получится сравнительно простой сценарий для автоматизации, потому что имеем много оборудования, настраивающегося одинаковым образом. Мы выберем сферический ДЦ в вакууме:

- Одна версия дизайна везде
- Два вендора, образующих две плоскости сети
- Один ДЦ похож на другой как две капли воды

Пусть наш Сервис-Провайдер LAN_DC будет, например, хостить обучающие видео о выживании в застрявших лифтах. В мегаполисах это пользуется бешенной популярностью, поэтому физических машин надо много.

Сначала я опишу сеть приблизительно такой, какой бы её хотелось видеть. А потом упрощу для лабы.

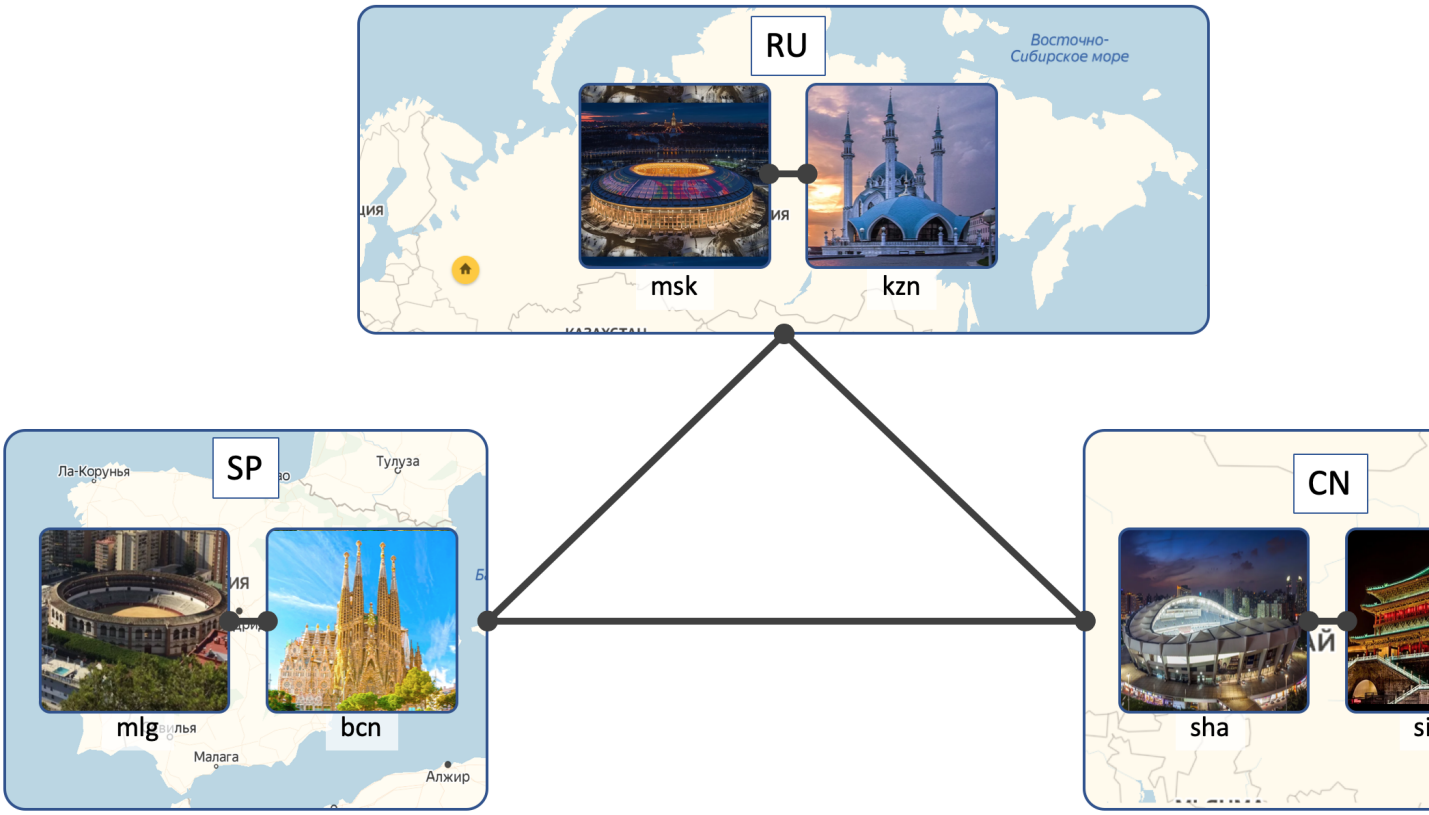


3.1 Физическая Топология

3.1.1 Локации

У LAN_DC будет 6 ДЦ:

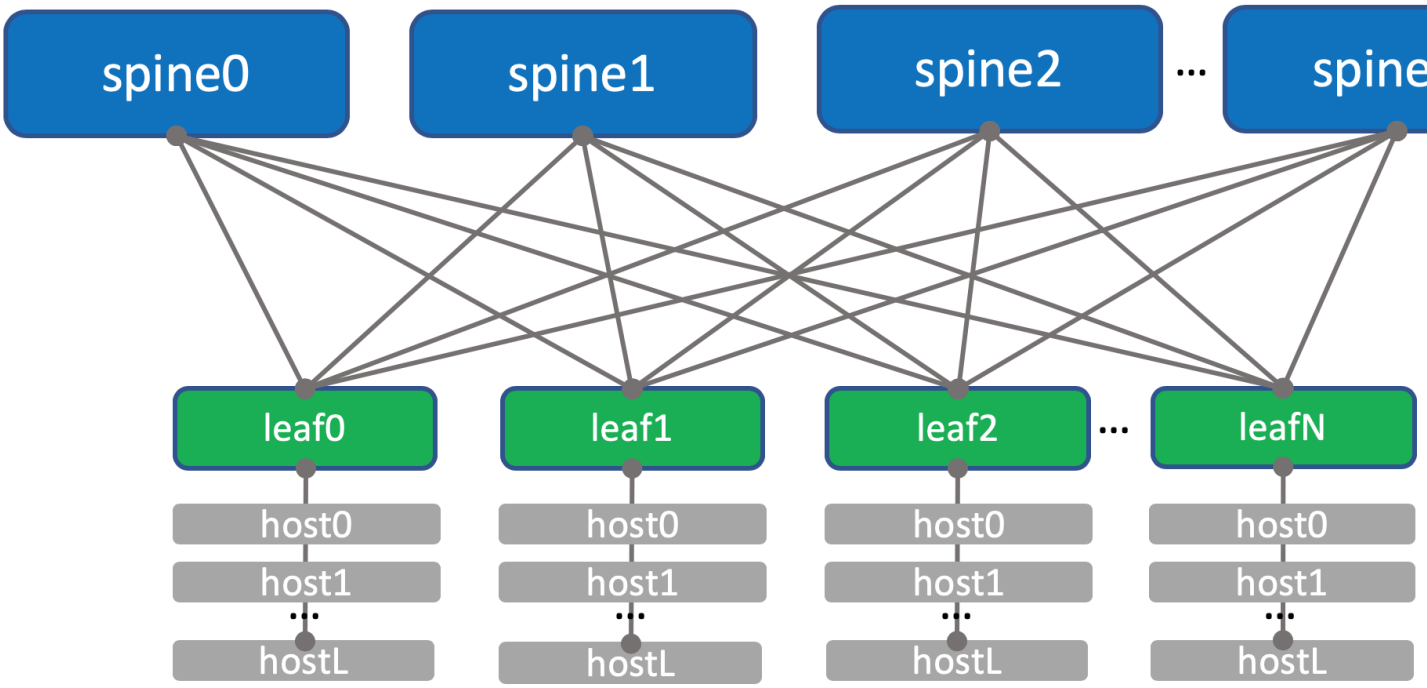
- **Россия (RU):**
  - Москва (**msk**)
  - Казань (**kzn**)
- **Испания (SP):**
  - Барселона (**bcn**)
  - Малага (**mlg**)
- **Китай (CN):**
  - Шанхай (**sha**)
  - Сиань (**sia**)



3.1.2 Внутри ДЦ (Intra-DC)

Во всех ДЦ идентичные сети внутренней связности, основанные на топологии Клоза. Что за сети Клоза и почему именно они - в отдельной статье.

В каждом ДЦ по 10 стоек с машинами, они будут нумероваться как **A**, **B**, **C** итд. В каждой стойке по 30 машин. Они нас интересовать не будут. Также в каждой стойке стоит коммутатор, к которому подключены все машины - это **Top of the Rack switch - ToR** или иначе в терминах фабрики Клоза будем называть его **Leaf**.



Общая схема фабрики

Именовать их будем *XXX-leafY*, где **XXX** - трёхбуквенное сокращение ДЦ, а **Y** - порядковый номер. Например, *kzn-leaf11*.

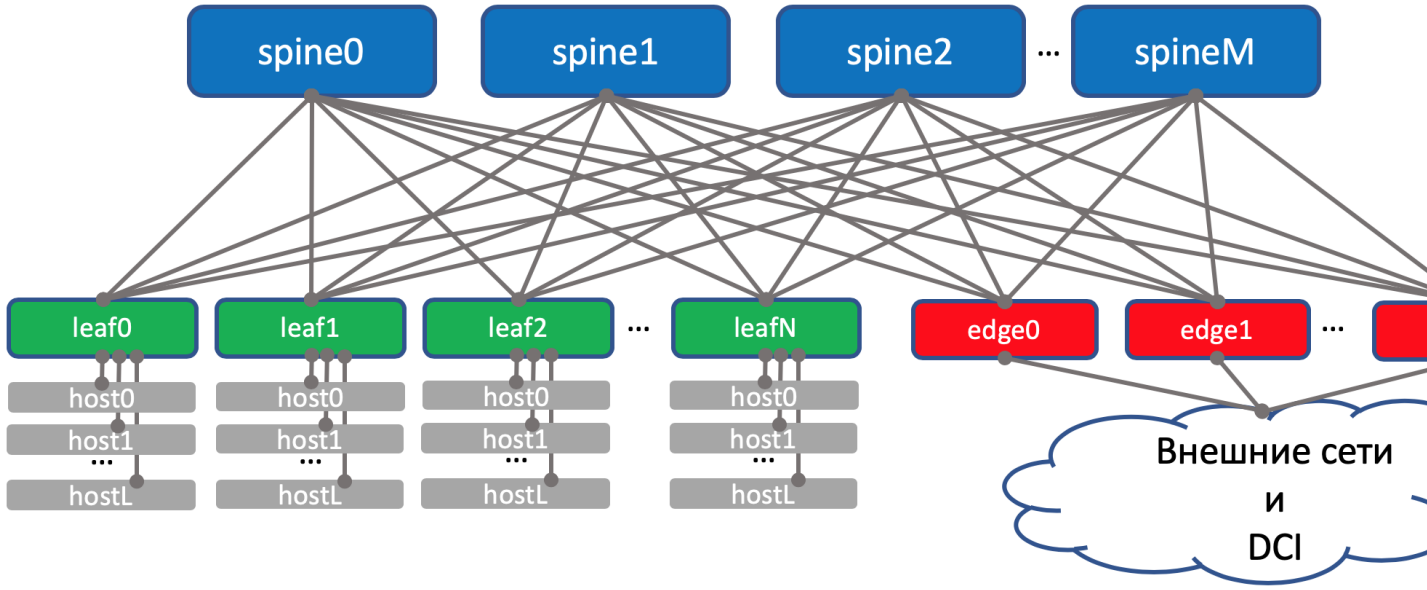
> Я в статьях позволю себе достаточно фривольно обращаться терминами Leaf и ToR, как синонимами. Однако нужно помнить, что это не так. ToR - это коммутатор, установленный в стойке, к которому подключаются машины. Leaf - это роль устройства в физической сети или свитч первого уровня в терминах топологии Клоза. То есть Leaf != ToR. Так Leaf'ом может быть End Of Raw-коммутатор, например. Однако в рамках этой статьи будем всё же обращаться ими как синонимами.

Каждый ToR-коммутатор в свою очередь соединён с четырьмя вышестоящими агрегирующими коммутаторами - **Spine**. Под Spine'ы выделено по одной стойке в ДЦ. Именовать будем аналогично: *XXX-spineY*.

В этой же стойке будет стоять сетевое оборудование для связности между ДЦ - 2 маршрутизатора с MPLS на борту. Но по большому счёту - это те же самые ToR'ы. То есть с точки зрения Spine-коммутаторов не имеет никакого значения обычный там ToR с подключенными машинами или маршрутизатор для DCI - один чёрт форвардить. Такие специальные ToR'ы называются **Edge-leaf**. Мы их будем именовать *XXX-edgeY*.

Выглядеть это будет так.

На схеме выше edge и leaf я действительно разместил на одном уровне. Классические трёхуровневые сети приучили нас рассматривать, аплинк (собственно отсюда и термин), как линки вверх. А тут



получается «аплинк» DCI идёт обратно вниз, что некоторым немного ломает привычную логику. В случае крупных сетей, когда датацентры делятся ещё на более мелкие единицы - **POD**'ы (Point Of Delivery), выделяют отдельные **Edge-POD**'ы для DCI и выхода во внешние сети.

Для удобства восприятия в дальнейшем я буду всё же рисовать Edge над Spine, при этом мы будем держать в уме, что никакого интеллекта на Spine и отличий при работе с обычными Leaf и с Edge-leaf нет (хотя тут могут быть нюансы, но в целом это так).

Схема фабрики с Edge-leaf'ами.

Троица Leaf, Spine и Edge образуют Underlay-сеть или фабрику.

Задача сетевой фабрики (читай Underlay), как мы уже определились в прошлом выпуске, очень и очень простая - обеспечить IP-связность между машинами как в пределах одного ДЦ, так и между. Оттого-то сеть и называется фабрикой, так же, например, как фабрика коммутации внутри модульных сетевых коробок, о чём подробнее можно почитать в СДСМ14.

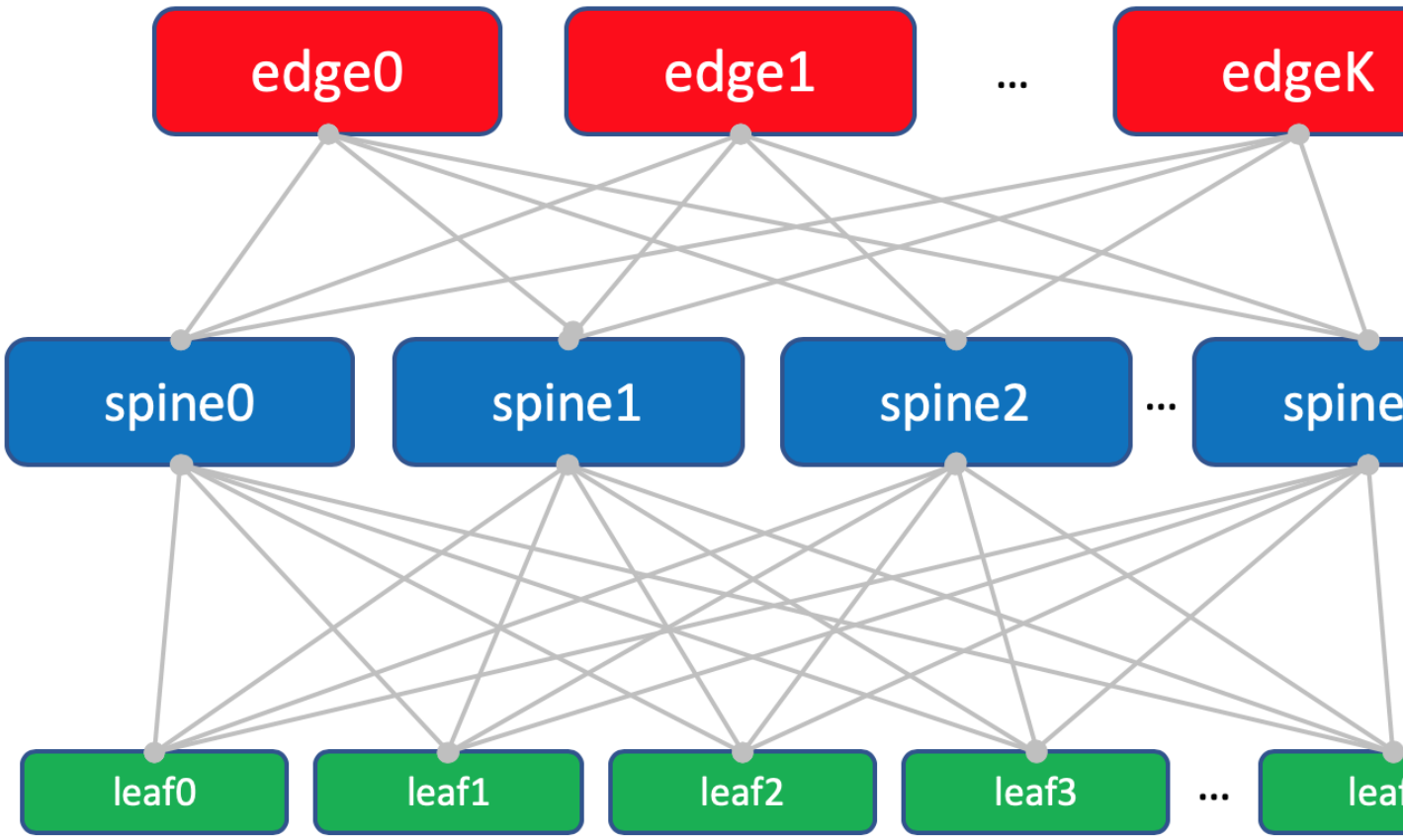
> А вообще такая топология называется фабрикой, потому что fabric в переводе - это ткань.
> И сложно не согласиться:

Фабрика полностью L3. Никаких VLAN, никаких Broadcast - вот такие у нас в LAN_DC замечательные программисты, умеют писать приложения, живущие в парадигме L3, а виртуальные машины не требуют Live Migration с сохранением IP-адреса.

И ещё раз: ответ на вопрос почему фабрика и почему L3 - в отдельной статье.

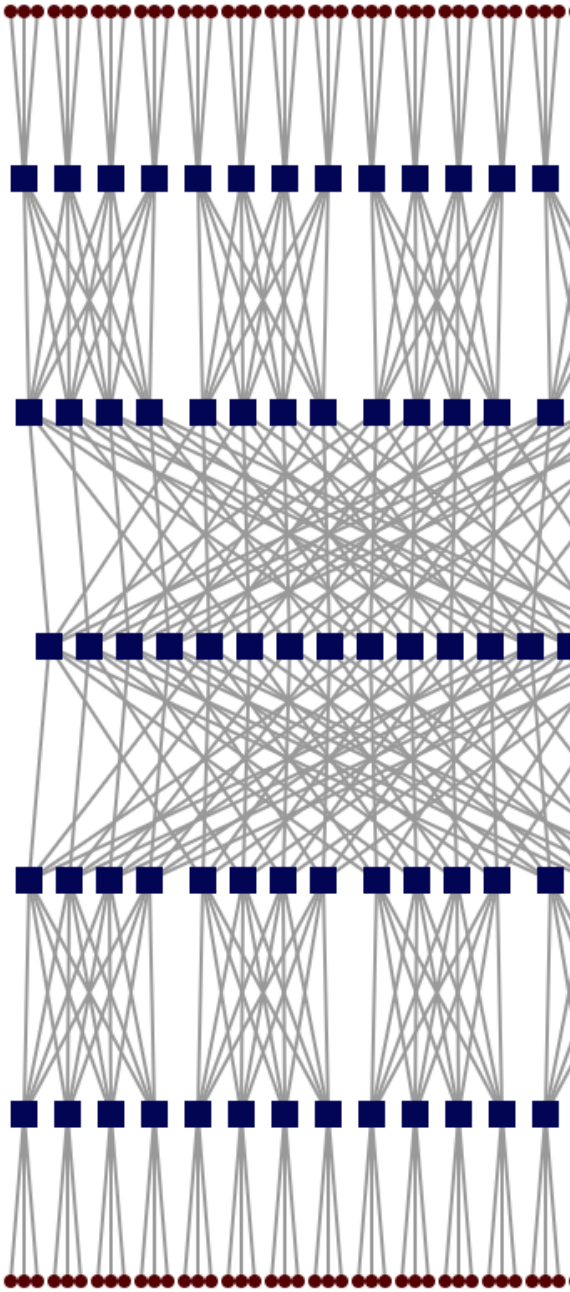
3.1.3 DCI - Data Center Interconnect (Inter-DC)

DCI будет организован с помощью Edge-Leaf, то есть они - наша точка выхода в магистраль. Для простоты предположим, что ДЦ связаны между собой прямыми линками. Исключим из рассмотрения внешнюю связность.


edge0
edge1
...
edgeK
spine0
spine1
spine2
...
spine
leaf0
leaf1
leaf2
leaf3
...
lea

Tree Depth	3
Ports per Switch	8
Hosts	128
Switches	80
Cables	384
Transmitters	768
Switch Tx's	640

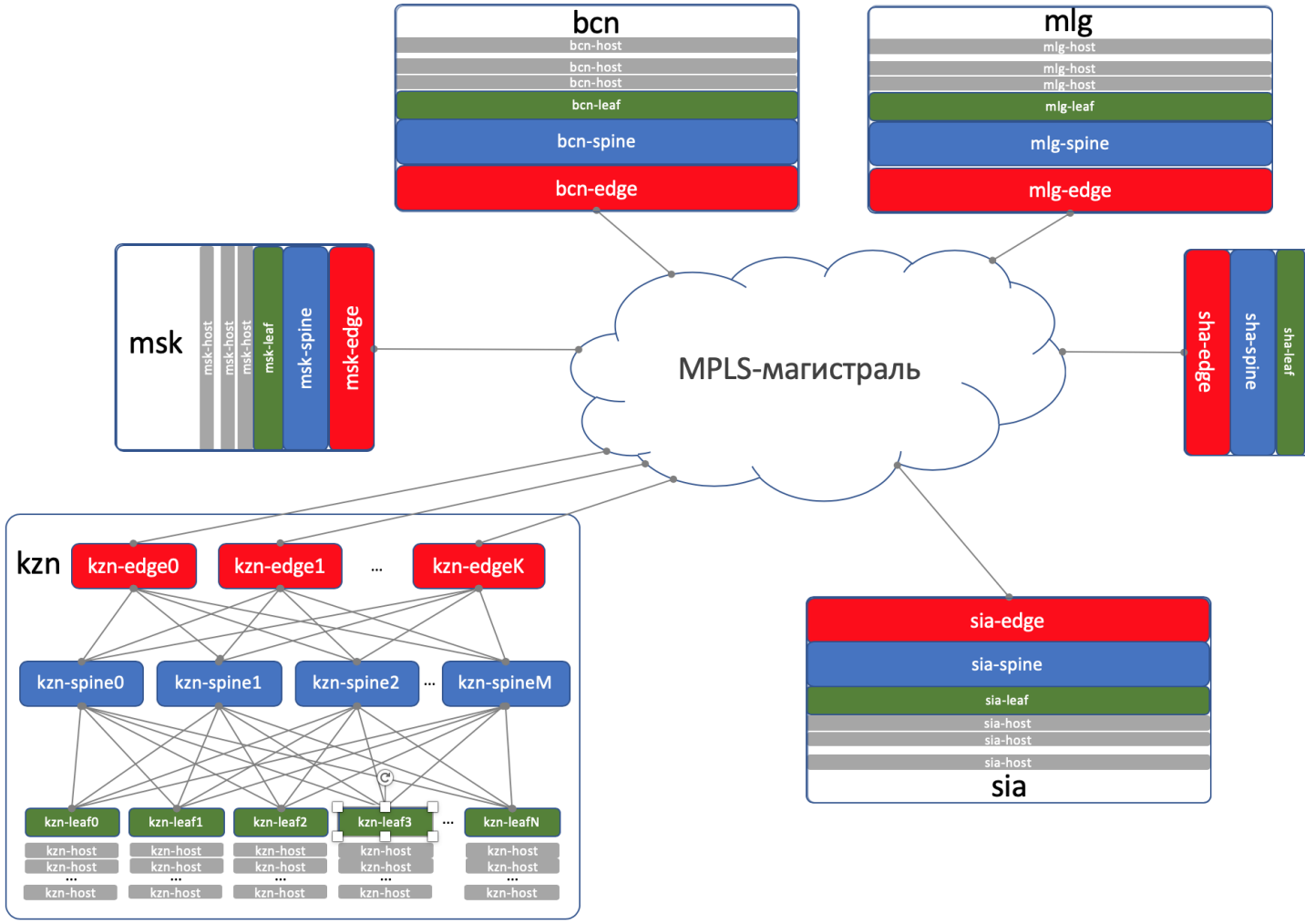
Fat Tree Visualizer
by He "Lonnie" Liu



> Я отдаю себе отчёт в том, что каждый раз, как я убираю какой-либо компонент, я значительно упрощаю сеть. И при автоматизации нашей абстрактной сети всё будет хорошо, а на реальной возникнут костыли. Это так. И всё же задача этой серии - подумать и поработать над подходами, а не героически решать выдуманные проблемы.

На Edge-Leaf'ах underlay помещается в VPN и передаётся через MPLS-магистраль (тот самый прямой линк).

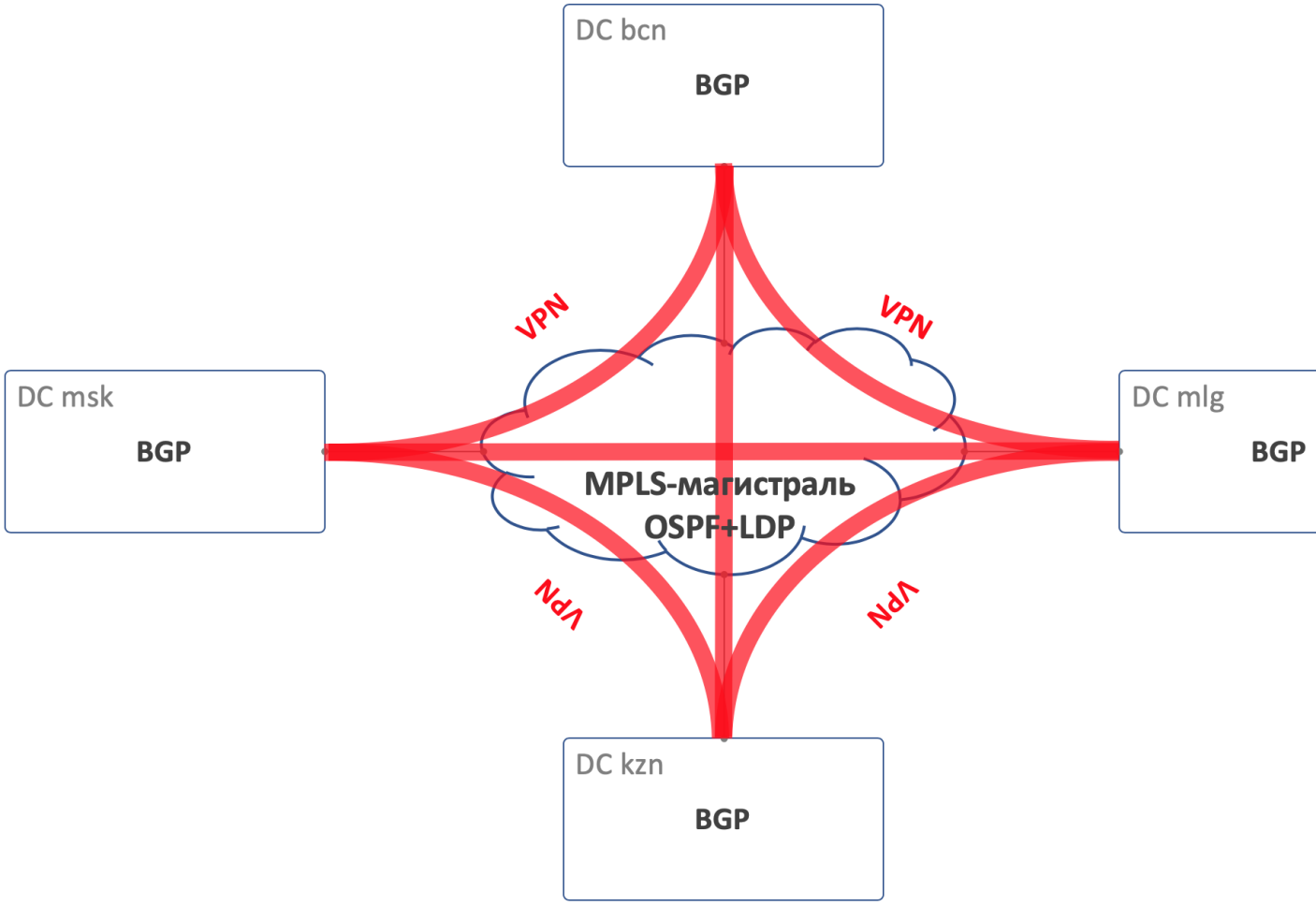
Вот такая верхнеуровневая схема получается.



3.2 Маршрутизация

Для маршрутизации внутри ДЦ будем использовать BGP. На MPLS-магистрали OSPF+LDP. Для DCI, то есть организации связности в андерлее - BGP L3VPN over MPLS.

Общая схема маршрутизации


DC bcn
BGP
VPN
VPN
DC msk
BGP
DC mlg
BGP
MPLS-магистраль
OSPF+LDP
VPN
VPN
DC kzn
BGP

На фабрике никаких OSPF и ISIS (запрещённый в Российской Федерации протокол маршрутизации). А это значит, что не будет Auto-discovery и вычисления кратчайших путей - только ручная (на самом деле автоматическая - мы же здесь про автоматизацию) настройка протокола, соседства и политик.

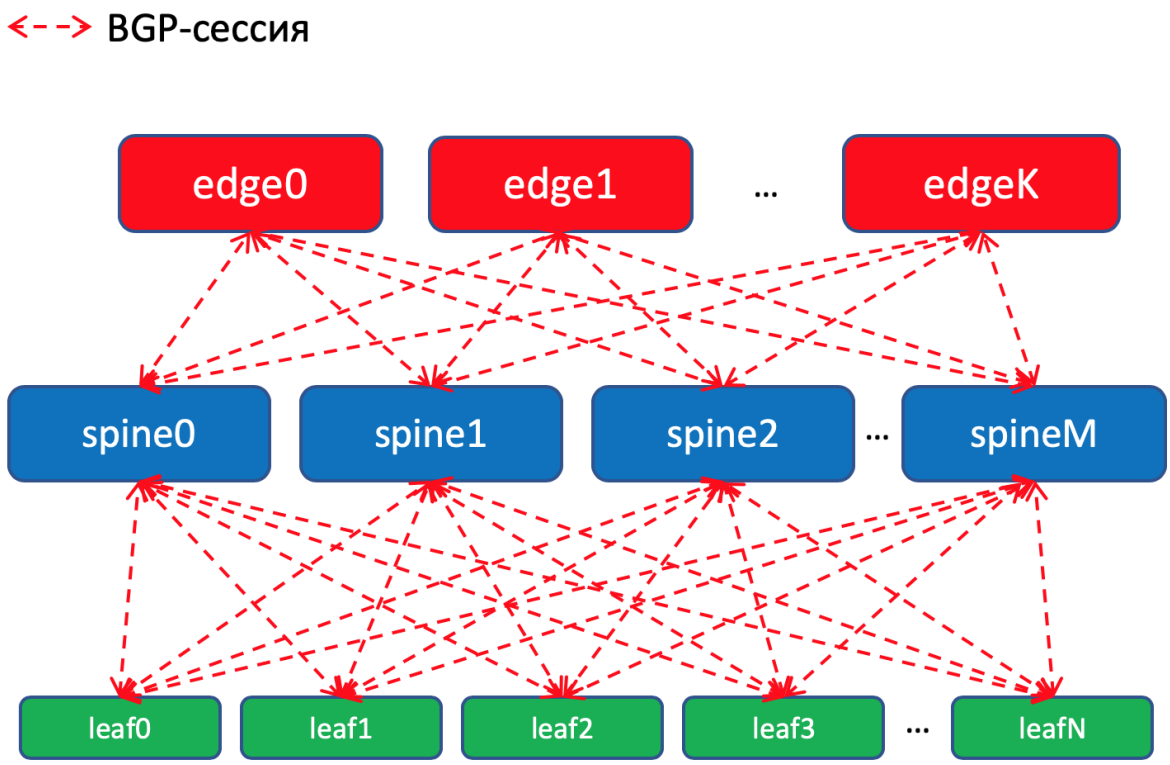


Схема маршрутизации BGP внутри ДЦ

Почему BGP? На эту тему есть целый RFC имени Facebook'а и Arista, где рассказывается, как строить **очень крупные** сети датацентров, используя BGP. Читается почти как художественный, очень рекомендую для томного вечера.

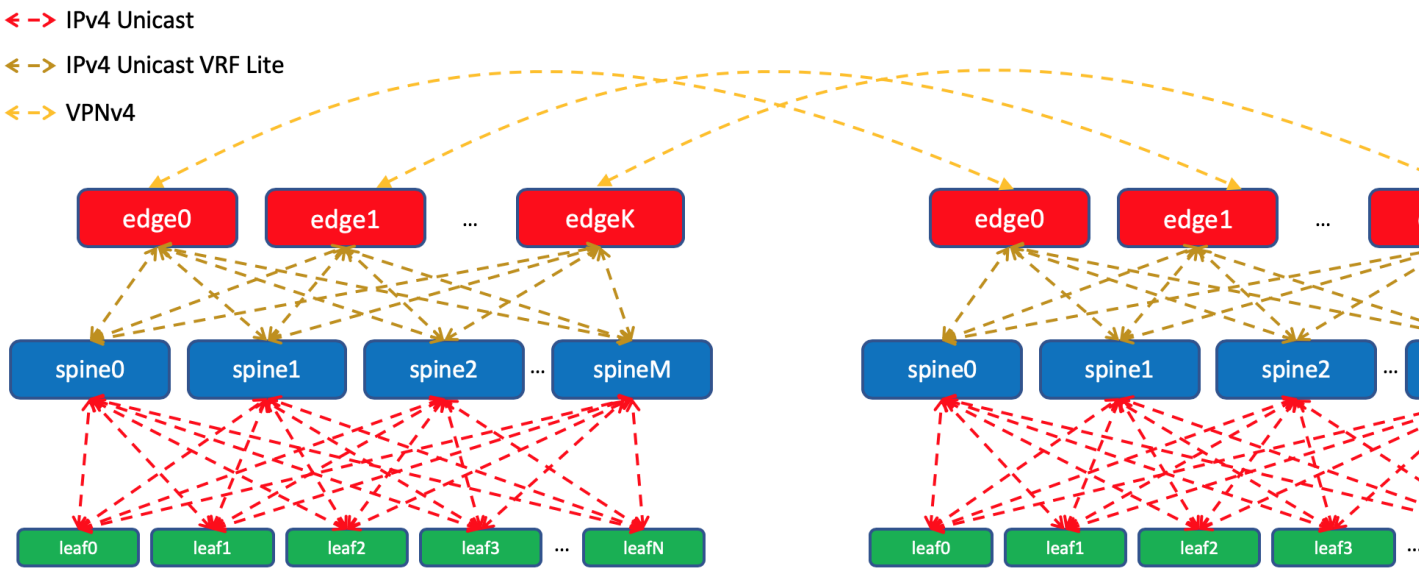
А ещё целый раздел в моей статье посвящён этому. Куда я вас и отсылаю.

Но всё же если коротко, то никакие IGP не подходят для сетей крупных датацентров, где счёт сетевым устройствами идёт на тысячи. Кроме того использование BGP везде позволит не распыляться на поддержку нескольких разных протоколов и синхронизацию между ними.

> Положа руку на сердце, на нашей фабрике, которая с большой долей вероятности не будет стремительно расти, за глаза хватило бы и OSPF. Это на самом деле проблемы мегаскейлеров и клауд-титанов. Но пофантазируем всего лишь на несколько выпусков, что нам это нужно, и будем использовать BGP, как завещал Пётр Лапухов.

3.2.1 Политики маршрутизации

На Leaf-коммутаторах мы импортируем в BGP префиксы с Underlay'ных интерфейсов с сетями. У нас будет BGP-сессия между **каждой** парой Leaf-Spine, в которой эти Underlay'ные префиксы будут анонсироваться по сети тудыть-сюдыть.



Внутри одного датацентра мы будем распространять специфики, которые импортировали на ToPe. На Edge-Leaf'ах будем их агрегировать и анонсировать в удалённые ДЦ и спускать до ToPов. То есть каждый ToP будет знать точно, как добраться до другого ToPa в этом же ДЦ и где точка входа, чтобы добраться до ToPa в другом ДЦ. В DCI маршруты будут передаваться, как VPNv4. Для этого на Edge-Leaf интерфейс в сторону фабрики будет помещаться в VRF, назовём его UNDERLAY, и соседство со Spine на Edge-Leaf будет подниматься внутри VRF, а между Edge-Leaf'ами в VPNv4-family.

А ещё мы запретим реанонсировать маршруты полученные от спайнов, обратно на них же.

На Leaf и Spine мы не будем импортировать Loopback'и. Они нам понадобятся только для того, чтобы определить Router ID. А вот на Edge-Leaf'ах импортируем его в Global BGP. Между Loopback-адресами Edge-Leaf'ы будут устанавливать BGP-сессию в IPv4 VPN-family друг с другом.

Между EDGE-устройствами у нас будет растянута магистраль на OSPF+LDP. Всё в одной зоне. Предельно простая конфигурация.

Вот такая картина с маршрутизацией.

3.2.2 BGP ASN

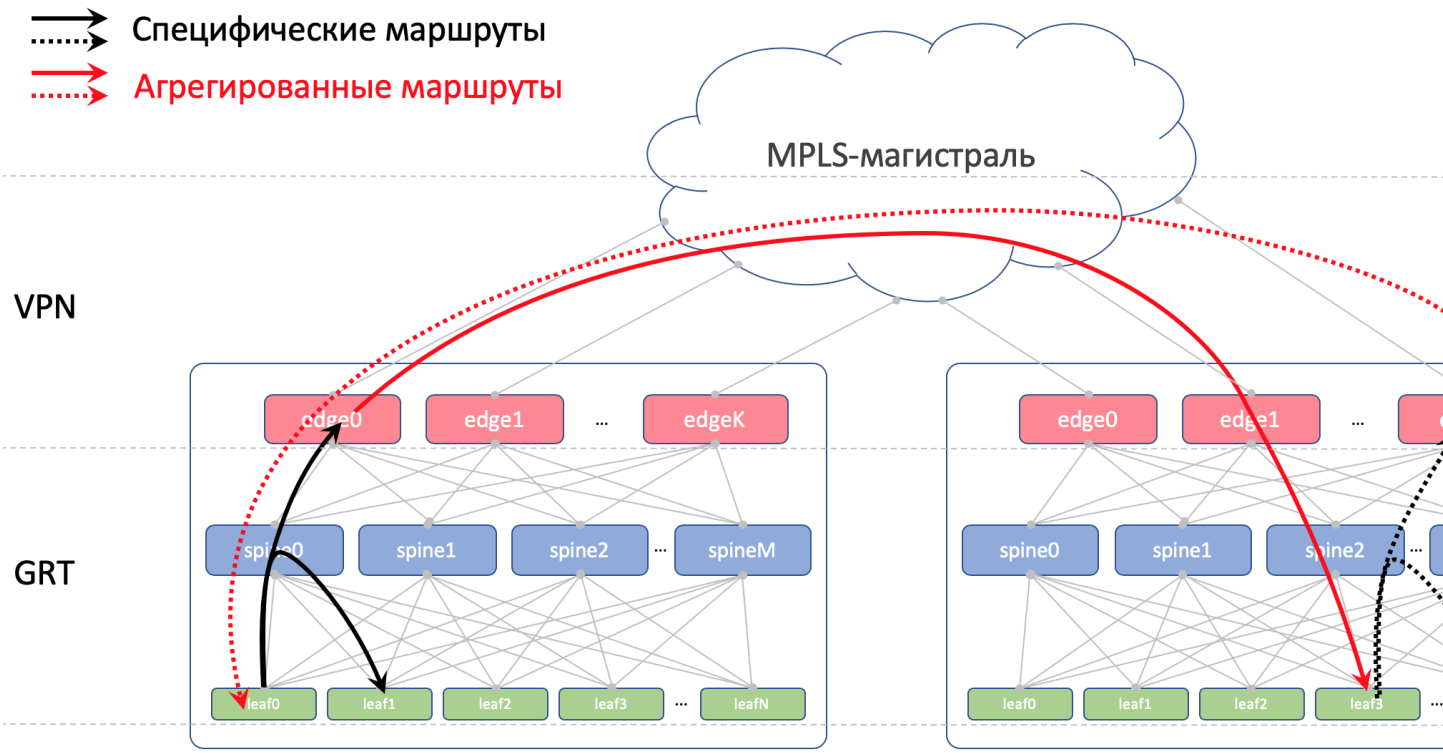
Edge-Leaf ASN

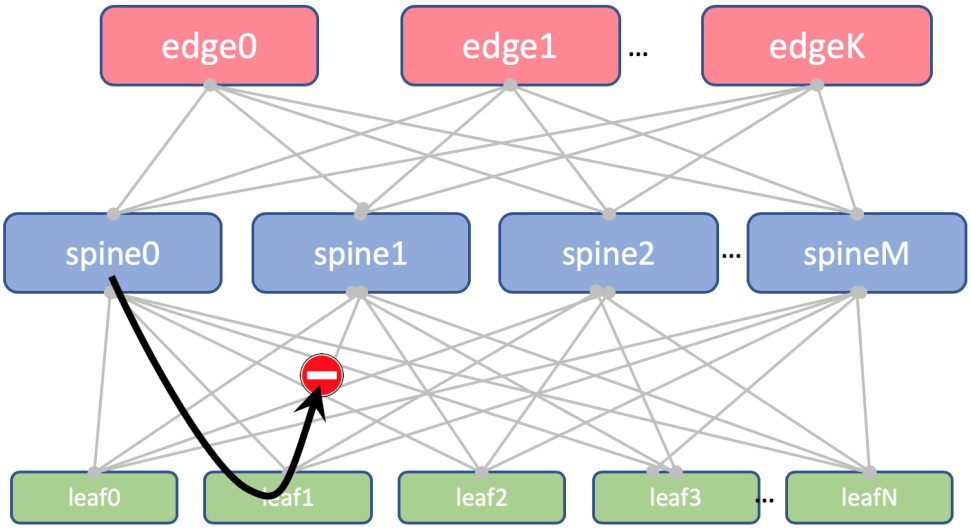
На Edge-Leaf'ах будет один ASN во всех ДЦ. Это важно, чтобы между Edge-Leaf'ами был iBGP, и мы не накололись на нюансы eBGP. Пусть это будет 65535. В реальности это мог бы быть номер публичной AS.

Spine ASN

На Spine у нас будет один ASN на ДЦ. Начнём здесь с самого первого номера из диапазона приватных AS - 64512, 64513 итд. Почему ASN на ДЦ? Декомпозируем этот вопрос на два:

- Почему одинаковые ASN на всех спайнах одного ДЦ?


Специфические маршруты
Агрегированные маршруты
MPLS-магистраль
VPN
GRT
edge0
edge1
edgeK
spine0
spine1
spine2
spineM
leaf0
leaf1
leaf2
leaf3
leafN


edge0
edge1
edgeK
spine0
spine1
spine2
spineM
leaf0
leaf1
leaf2
leaf3
leafN

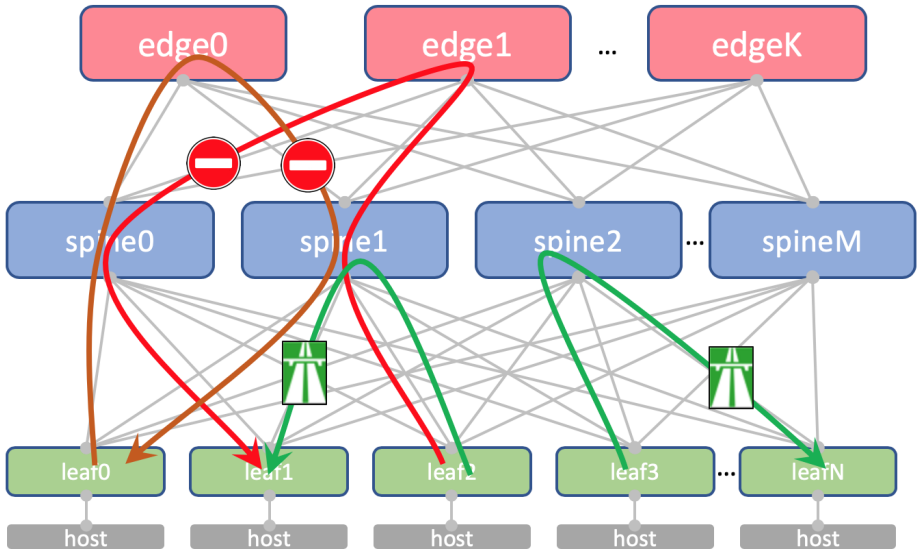
- Почему разные в разных ДЦ?

Почему одинаковые ASN на всех спайнах одного ДЦ Вот как будет выглядеть AS-Path Андерлейного маршрута на Edge-Leaf:

```
[leafX_ASN, spine_ASN, edge_ASN]
```

При попытке заанонсировать его обратно на Спайн, тот его отбросит потому что его AS (Spine_AS) уже есть в списке.

Однако в пределах ДЦ нас совершенно устраивает, что маршруты Underlay, поднявшиеся до Edge не смогут спуститься вниз. Вся коммуникация между хостами внутри ДЦ должна происходить в пределах уровня спайнов.



The Good AS path:
[leafX, spine, leafY]

The Bad AS path:
[leafX, **spine**, edge, **spine**

The Ugly AS path:
[leafX, **spine**, edge, **spine**

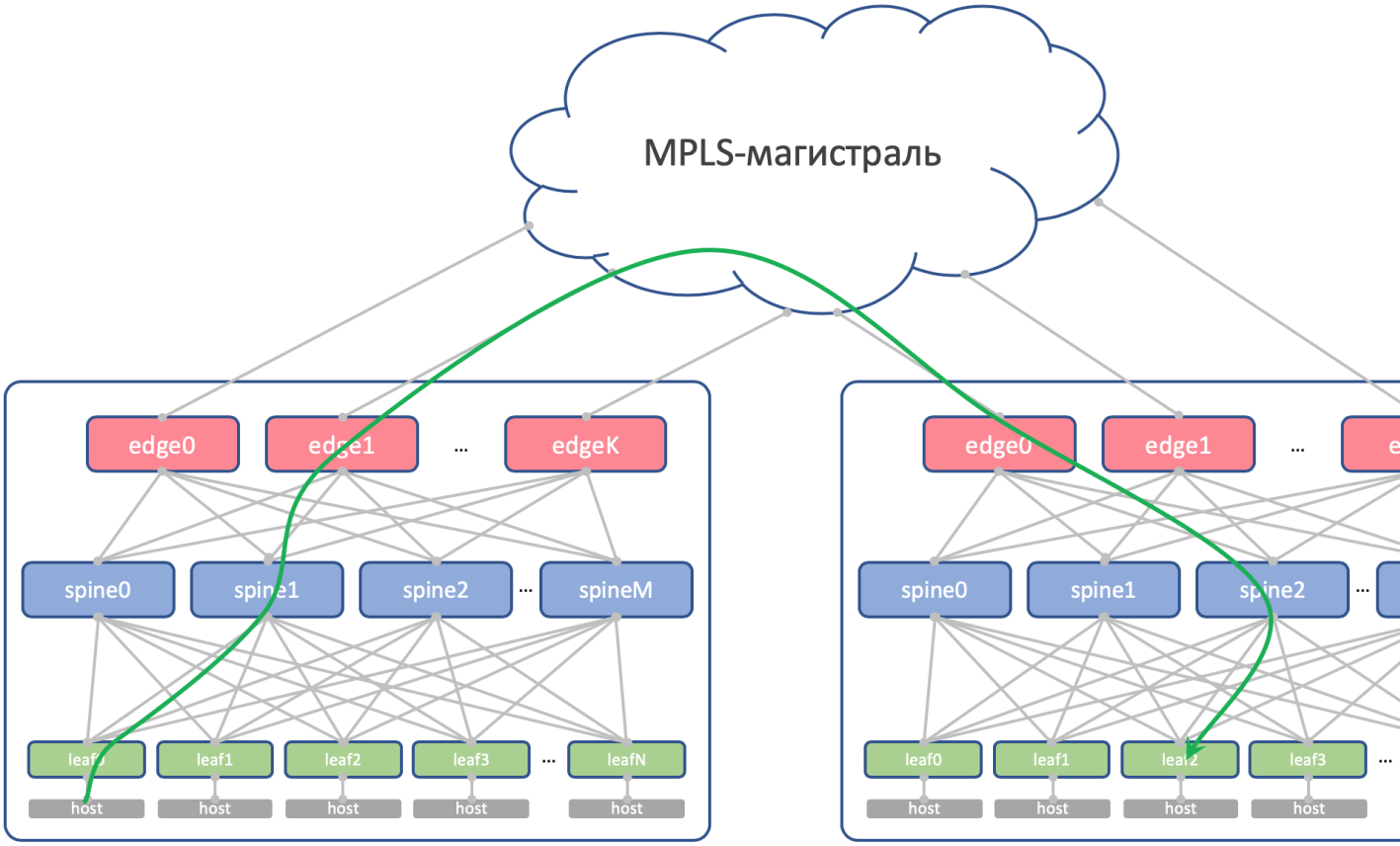
При этом агрегированные маршруты других ДЦ в любом случае беспрепятственно будут доходить до ToPов - в их AS-Path будет только ASN 65535 - номер AS Edge-Leaf'ов, потому что именно на них они были созданы.

Почему разные в разных ДЦ Теоретически нам может потребоваться протащить Loopback'и каких-нибудь сервисных виртуальных машин между ДЦ. Например, на хосте у нас запустится Route Reflector или тот самый VNGW (Virtual Network Gateway), который по BGP запирится с ToPом и проанонсирует свой лупбэк, который должен быть доступен из всех ДЦ. Так вот как будет выглядеть его AS-Path:

```
[VNF_ASN, leafX_DC1_ASN, spine_DC1_ASN, edge_ASN, spine_DC2_ASN, leafY_DC2_ASN]
```

И здесь нигде не должно быть повторяющихся ASN.

То есть Spine_DC1 и Spine_DC2 должны быть разными, ровно как и leafX_DC1 и leafY_DC2, к чему
Как вы, наверно, знаете, существуют хаки, позволяющие принимать маршруты с повторяющимися ASN вопреки механизму предотвращения петель (allowas-in на Cisco). И у этого есть даже вполне законные применения. Но это потенциальная брешь в устойчивости сети. И я лично в неё пару раз проваливался. И если у нас есть возможность не использовать опасные вещи, мы ей воспользуемся.



AS path: [VNF, leafX_DC1, **spine_DC1**, edge, **spine_DC2**, leafY_DC2]

Leaf ASN

У нас будет индивидуальный ASN на каждом Leaf-коммутаторе в пределах всей сети. Делаем мы так из соображений, приведённых выше: AS-Path без петель, конфигурация BGP без закладок. Чтобы маршруты между Leaf'ами беспрепятственно проходили, AS-Path должен выглядеть так:

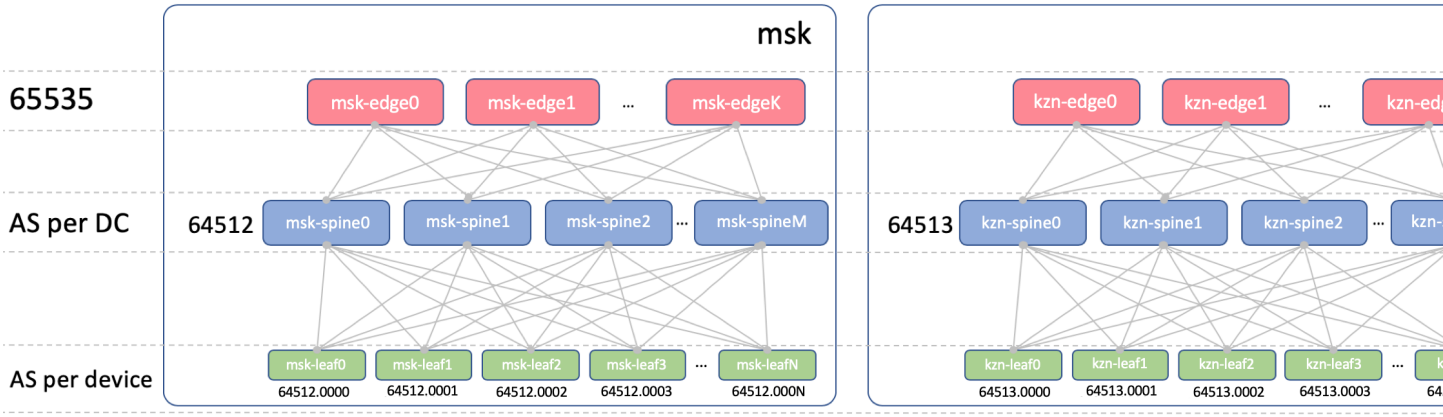
```
[leafX_ASN, spine_ASN, leafY_ASN]
```

где leafX_ASN и leafY_ASN хорошо бы, чтобы отличались. Требуется это и для ситуации с анонсом лупбэка VNF между ДЦ:

```
[VNF_ASN, leafX_DC1_ASN, spine_DC1_ASN, edge_ASN, spine_DC2_ASN, leafY_DC2_ASN]
```

Будем использовать 4-байтовый ASN и генерировать его на основе ASN Spine'а и номера Leaf-коммутатора, а именно, вот так: *Spine_ASN.0000X*.

Вот такая картина с ASN.



3.3 IP-план

Принципиально, нам нужно выделить адреса для следующих подключений:

1. Адреса сети Underlay между ToR и машиной. Они должны быть уникальны в пределах всей сети, чтобы любая машина могла связаться с любой другой. Отлично подходит **10/8**. На каждую стойку по /26 с запасом. Будем выделять по /19 на ДЦ и /17 на регион.
2. **Линковые адреса между Leaf/Tor и Spine.**

   Их хотелось бы назначать алгоритмически, то есть вычислять из имён устройств, которые нужно подключить.

   Пусть это будет... 169.254.0.0/16.

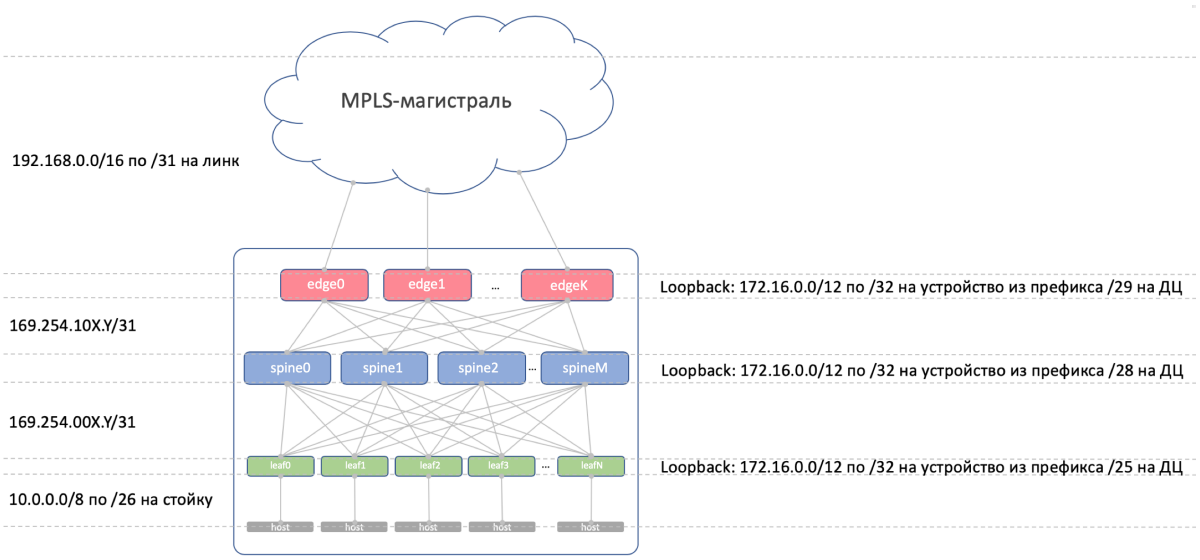
   А именно **169.254.00X.Y/31**, где **X** - номер Spine, **Y** - P2P-сеть /31.

   Это позволит запускать до 128 стоек, и до 10 Spine в ДЦ. Линковые адреса могут (и будут) повторяться из ДЦ в ДЦ.
3. Стык Spine - Edge-Leaf организуем на подсетях **169.254.10X.Y/31**, где точно так же **X** - номер Spine, **Y** - P2P-сеть /31.

4. Линковые адреса из Edge-Leaf в MPLS-магистраль. Здесь ситуация несколько иная - место соединения всех кусков в один пирог, поэтому переиспользовать те же самые адреса не получится - нужно выбирать следующую свободную подсеть. Поэтому за основу возьмём **192.168.0.0/16** и будем из неё выгребать свободные.
5. **Адреса Loopback. Отдадим под них весь диапазон 172.16.0.0/12.**
    - Leaf - по /25 на ДЦ - те же 128 стоек. Выделим по /23 на регион.
    - Spine - по /28 на ДЦ - до 16 Spine. Выделим по /26 на регион.
    - Edge-Leaf - по /29 на ДЦ - до 8 коробок. Выделим по /27 на регион.

Если в ДЦ нам не будет хватать выделенных диапазонов (а их не будет - мы же претендуем на гиперскейлероство), просто выделяем следующий блок.

Вот такая картина с IP-адресацией.



Loopback'и:

Префикс	Роль	Регион	ДЦ
172.16.0.0/23			
172.16.0.0/27			ru
172.16.0.0/29		ru	msk
172.16.0.8/29			kzn
172.16.0.32/27	edge		sp
172.16.0.32/29		sp	bcn
172.16.0.40/29			mlg
172.16.0.64/27			cn
172.16.0.64/29		cn	sha
172.16.0.72/29			sia
172.16.2.0/23			

(continues on next page)

(продолжение с предыдущей страницы)

```
+------------------+       +--------+-----+
| 172.16.2.0/26    |       |        |     |
| 172.16.2.0/28    |       |   ru   | msk |
| 172.16.2.16/28   |       |        | kzn |
+------------------+       +--------+-----+
| 172.16.2.64/26   | spine |        |     |
| 172.16.2.64/28   |       |   sp   | bcn |
| 172.16.2.80/28   |       |        | kzn |
+------------------+       +--------+-----+
| 172.16.2.128/26  |       |        |     |
| 172.16.2.128/28  |       |   cn   | sha |
| 172.16.2.144/28  |       |        | sia |
+------------------+-------+--------+-----+
| 172.16.8.0/21    |       |        |     |
+------------------+       +--------+-----+
| 172.16.8.0/23    |       |        |     |
| 172.16.8.0/25    |       |   ru   | msk |
| 172.16.8.128/25  |       |        | kzn |
+------------------+       +--------+-----+
| 172.16.10.0/23   | leaf  |        |     |
| 172.16.10.0/25   |       |   sp   | bcn |
| 172.16.10.128/25 |       |        | mlg |
+------------------+       +--------+-----+
| 172.16.12.0/23   |       |        |     |
| 172.16.12.0/25   |       |   cn   | sha |
| 172.16.12.128/25 |       |        | sia |
+------------------+-------+--------+-----+
```

Underlay:

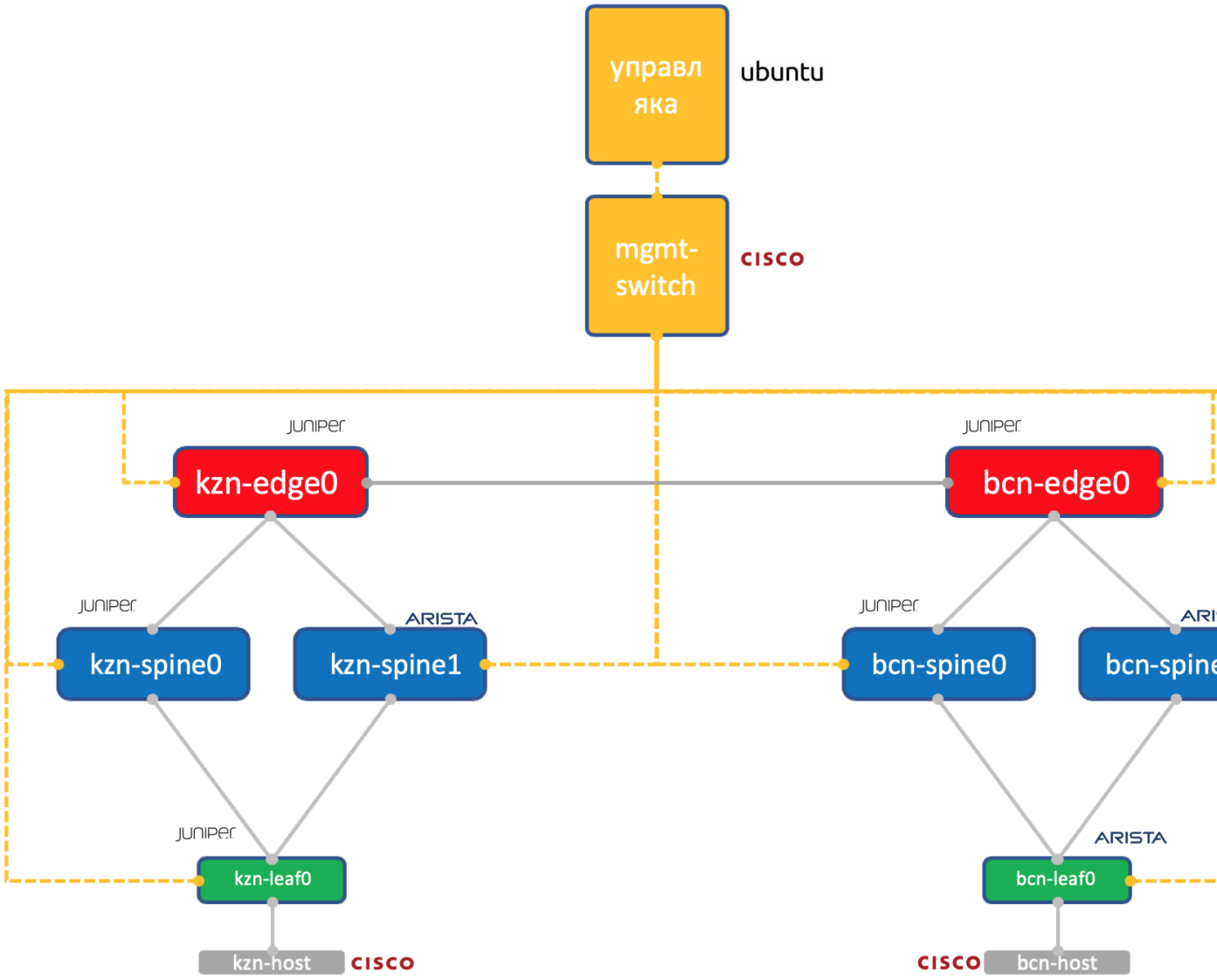
```
+------------------+--------+-----+
| Префикс          | Регион | ДЦ  |
+==================+========+=====+
| 10.0.0.0/17      |        |     |
| 10.0.0.0/19      |   ru   | msk |
| 10.0.32.0/19     |        | kzn |
+------------------+--------+-----+
| 10.0.128.0/17    |        |     |
| 10.0.128.0/19    |   sp   | bcn |
| 10.0.160.0/19    |        | mlg |
+------------------+--------+-----+
| 10.1.0.0/17      |        |     |
| 10.1.0.0/19      |   cn   | sha |
| 10.1.32.0/19     |        | sia |
+------------------+--------+-----+
```

3.4 Лаба

Два вендора. Одна сеть. АДСМ.

Juniper + Arista. Ubuntu. Старая добрая Ева.

Количество ресурсов на нашей виртуалочке в Миране всё же ограничено, поэтому для практики мы будем использовать вот такую упрощённую до предела сеть.


управл
яка
ubuntu
mgmt-
switch
cisco
juniper
kzn-edge0
juniper
bcn-edge0
juniper
kzn-spine0
ARISTA
kzn-spine1
juniper
bcn-spine0
juniper
kzn-leaf0
ARISTA
bcn-leaf0
kzn-host
cisco
cisco
bcn-host

- Два датацентра: Казань и Барселона.
- По два спайна в каждом: Juniper и Arista.
- По одному тору (Leaf'у) в каждом - Juniper и Arista, с одним подключенным хостом (возьмём легковесный Cisco IOL для этого).
- По одной ноде Edge-Leaf (пока только Juniper).
- One Cisco switch to rule them all.
- **Помимо сетевых коробок запущена виртуальная машина-управляка. Под управлением Ubuntu.** Она имеет доступ ко всем устройствам, на ней будут крутиться IPAM/DCIM-системы, букет питоновских скриптов, анзибль и что угодно ещё, что нам может понадобиться.

Полная конфигурация всех сетевых устройств, которую мы будем пробовать воспроизвести с помощью автоматики.

3.5 Заключение

Так же принято? Под каждой статьёй делать короткий вывод?

Итак мы выбрали трёхуровневую сеть Клоза внутри ДЦ, поскольку ожидаем много East-West трафика и хотим ECMP.

Разделили сеть на физическую (андерлей) и виртуальную (оверлей). При этом оверлей начинается с хоста - тем самым упростили требования к андерлею.

Выбрали BGP в качестве протокола маршрутизации анедрелейных сетей за его масштабируемость и гибкость политик.

У нас будут отдельные узлы для организации DCI - Edge-leaf.

На магистрали будет OSPF+LDP.

DCI будет реализован на основе MPLS L3VPN.

Для P2P-линков IP-адреса мы будем вычислять алгоритмически на основе имён устройств.

Лупбэки будем назначать по роли устройств и их расположению последовательно.

Андерлейные префиксы - только на Leaf-коммутаторы последовательно на основе их расположения.

Предположим, что прямо сейчас у нас ещё не установлено оборудование.

Поэтому наши следующие шаги будут - завести их в системах (IPAM, инвентарная), организовать доступ, сгенерировать конфигурацию и задеплоить её.

В следующей статье разберёмся с Netbox - системой инвентаризации и управления IP-пространством в ДЦ.

3.5.1 Спасибы

- Андрею Глазкову aka @glazgoo за вычитку и правки
- Александру Клименко aka @v00lk за вычитку и правки
- Артёму Чернобаю за КДПВ

Глава 4

Часть 3. IPAM/DCIM-системы

В предыдущих сериях АДСМ мы выработали фреймворк автоматизации, разобрались с тем, зачем появилась виртуализация и как она работает. В последней части мы выбрали и обосновали дизайн сети, роли устройств, производителей, определились с LLD (адресацией, маршрутизацией, номерами Автономных Систем).
Теперь мы готовы подумать о том, как всю эту гору информации хранить и в дальнейшем удобно извлекать.

Нет, есть, конечно, и сегодня компании, которые ведут учёт выделенных IP-адресов в таблице Excel. Но это не наш путь.
Даже для самой маленькой конторки размеров в пару филиалов наличие централизованной системы управления IP-пространством не повредит.
Необходимость системы инвентаризации очевидна без лишних слов.

Этот выпуск я посвящу неотъемлемым системам в сетевой автоматизации - системе управления адресным пространством и инвентарной системе.
Мы выберем и установим её, разберёмся с архитектурой, схемой БД, интерфейсами взаимодействия и наполним её. А в следующих частях начнём писать несложные скрипты, автоматизирующие повторяющиеся операции, такие как добавление новых стоек.
Кроме того, я уже опубликовал отдельную статью о RESTful API, в которой сделал короткий обзор его принципов и работы, это нам понадобится.

Сегодня рынок предлагает около дюжины инструментов, реализующих эту задачу: как платных, так и Open Source.

Для задач этой серии статей я выбрал NetBox по следующим причинам:

1. Это бесплатно

IPAM/DCIM
~ ❯ ./adsm.3

2. Он содержит в себе обе необходимые части - инвентаризацию и управление IP-пространством.
3. У него есть RESTful API-интерфейс.
4. Его разработал Digital Ocean (а конкретнее, любимый всеми Jeremy Stretch) для себя, то есть для дата-центров. Поэтому тут есть почти всё, что нужно, и почти ничего лишнего.
5. Он активно поддерживается (Slack, Github, Google-рассылки) и обновляется.
6. Это Open Source

> Для нужд АДСМ я развернул NetBox в виртуалочке на нашем сервере (спасибо Антону Клочкову и Мирану): http://netbox.linkmeup.ru:45127
>
> Кроме того я заполнил почти все необходимые нам в дальнейшем данные.
>
> Поэтому вы можете попробовать почти все примеры и изучать схему данных в режиме чтения, пока не развернули свою инсталляцию.

Немного полезного перед началом:

- Сам NetBox на github
- Контейнерная версия
- Полная инструкция по установке и вся документация по продукту
- SDK для работы с NetBox в Python
- Документация по API
- Mailing list
- Канал в Slack NetworkToCode
- Что такое REST
- Инсталляция NetBox для нужд АДСМ

4.1 Архитектура системы

- NetBox написан на Python3. Что хорошо, потому что ряд других решений написан на php и изменять их при необходимости не так уж просто.
- Фреймворк для самого сайта - Django.
- В качестве БД используется PostgreSQL.
- WEB-frontend (HTTP-сервис) - NGINX - он проксирует запросы в Gunicron.
- WSGI - Gunicorn - интерфейс между Nginx и самим приложением.
- Фреймворк для документации по API - Swagger.
- Чтобы демонизировать NetBox - Systemd.

> NetBox - проект молодой и быстро развивающийся. Например, в 2.7 отказались от supervisord и тянущегося за ним Python 2.7 в пользу systemd. Не так давно там не было ни кэширования, ни Webhooks. Поэтому меняется всё быстро и информация в статье может устареть к моменту чтения.

Иными словами все компоненты зрелые и проверенные.

По словам автора NetBox отражает не реальное состояние сети, а целевое. Поэтому ничего не подгружается в NetBox из сети - это сеть настраивается в соответствие с содержимым NetBox. Таким образом NetBox выступает единственным источником истины (калька с single source of truth). И изменения на

сети должны быть инициированы изменениями в NetBox. А это очень неплохо ложится на идеологию, которую я исповедую в этой серии статей - хочешь сделать изменения на сети - сначала внеси их в системы.

4.2 Схема данных NetBox

Две главные задачи, которые решает NetBox: управление адресным пространством и инвентаризация. NetBox едва ли станет единственной системой инвентаризации в компании, скорее, это будет специфическая дополнительная система для инвентаризации именно сети, забирающая данные из основной. Очевидно, в нашем случае для целей АДСМ будет только NetBox.

> К данному моменту большая часть начальных данных в NetBox уже внесена. На этих данных я буду демонстрировать различные примеры работы через API. Вы можете просто полазить и посмотреть: http://netbox.linkmeup.ru:45127 И эти же данные понадобятся в дальнейшем, когда мы перейдём к автоматизации.

В общих чертах схему данных можно увидеть по схеме БД в Postgres'е

```
+--------+-----------------------------------+-------+--------+
|        |               List of relations   |       |        |
| Schema |                Name               | Type  | Owner  |
+========+===================================+=======+========+
| public | auth_group                        | table | netbox |
| public | auth_group_permissions            | table | netbox |
| public | auth_permission                   | table | netbox |
| public | auth_user                         | table | netbox |
| public | auth_user_groups                  | table | netbox |
| public | auth_user_user_permissions        | table | netbox |
| public | circuits_circuit                  | table | netbox |
| public | circuits_circuittermination       | table | netbox |
| public | circuits_circuittype              | table | netbox |
| public | circuits_provider                 | table | netbox |
| public | dcim_cable                        | table | netbox |
| public | dcim_consoleport                  | table | netbox |
| public | dcim_consoleporttemplate          | table | netbox |
| public | dcim_consoleserverport            | table | netbox |
| public | dcim_consoleserverporttemplate    | table | netbox |
| public | dcim_device                       | table | netbox |
| public | dcim_devicebay                    | table | netbox |
| public | dcim_devicebaytemplate            | table | netbox |
| public | dcim_devicerole                   | table | netbox |
| public | dcim_devicetype                   | table | netbox |
| public | dcim_frontport                    | table | netbox |
| public | dcim_frontporttemplate            | table | netbox |
| public | dcim_interface                    | table | netbox |
| public | dcim_interface_tagged_vlans       | table | netbox |
| public | dcim_interfacetemplate            | table | netbox |
| public | dcim_inventoryitem                | table | netbox |
| public | dcim_manufacturer                 | table | netbox |
| public | dcim_platform                     | table | netbox |
| public | dcim_powerfeed                    | table | netbox |
| public | dcim_poweroutlet                  | table | netbox |
| public | dcim_poweroutlettemplate          | table | netbox |
| public | dcim_powerpanel                   | table | netbox |
| public | dcim_powerport                    | table | netbox |
| public | dcim_powerporttemplate            | table | netbox |
```

(continues on next page)

(продолжение с предыдущей страницы)

```
| public | dcim_rack                           | table | netbox |
| public | dcim_rackgroup                      | table | netbox |
| public | dcim_rackreservation                | table | netbox |
| public | dcim_rackrole                       | table | netbox |
| public | dcim_rearport                       | table | netbox |
| public | dcim_rearporttemplate               | table | netbox |
| public | dcim_region                         | table | netbox |
| public | dcim_site                           | table | netbox |
| public | dcim_virtualchassis                 | table | netbox |
| public | django_admin_log                    | table | netbox |
| public | django_content_type                 | table | netbox |
| public | django_migrations                   | table | netbox |
| public | django_session                      | table | netbox |
| public | extras_configcontext                | table | netbox |
| public | extras_configcontext_platforms      | table | netbox |
| public | extras_configcontext_regions        | table | netbox |
| public | extras_configcontext_roles          | table | netbox |
| public | extras_configcontext_sites          | table | netbox |
| public | extras_configcontext_tags           | table | netbox |
| public | extras_configcontext_tenant_groups  | table | netbox |
| public | extras_configcontext_tenants        | table | netbox |
| public | extras_customfield                  | table | netbox |
| public | extras_customfield_obj_type         | table | netbox |
| public | extras_customfieldchoice            | table | netbox |
| public | extras_customfieldvalue             | table | netbox |
| public | extras_customlink                   | table | netbox |
| public | extras_exporttemplate               | table | netbox |
| public | extras_graph                        | table | netbox |
| public | extras_imageattachment              | table | netbox |
| public | extras_objectchange                 | table | netbox |
| public | extras_reportresult                 | table | netbox |
| public | extras_tag                          | table | netbox |
| public | extras_taggeditem                   | table | netbox |
| public | extras_webhook                      | table | netbox |
| public | extras_webhook_obj_type             | table | netbox |
| public | ipam_aggregate                      | table | netbox |
| public | ipam_ipaddress                      | table | netbox |
| public | ipam_prefix                         | table | netbox |
| public | ipam_rir                            | table | netbox |
| public | ipam_role                           | table | netbox |
| public | ipam_service                        | table | netbox |
| public | ipam_service_ipaddresses            | table | netbox |
| public | ipam_vlan                           | table | netbox |
| public | ipam_vlangroup                      | table | netbox |
| public | ipam_vrf                            | table | netbox |
| public | secrets_secret                      | table | netbox |
| public | secrets_secretrole                  | table | netbox |
| public | secrets_secretrole_groups           | table | netbox |
| public | secrets_secretrole_users            | table | netbox |
| public | secrets_sessionkey                  | table | netbox |
| public | secrets_userkey                     | table | netbox |
| public | taggit_tag                          | table | netbox |
| public | taggit_taggeditem                   | table | netbox |
| public | tenancy_tenant                      | table | netbox |
| public | tenancy_tenantgroup                 | table | netbox |
| public | users_token                         | table | netbox |
```

(continues on next page)

(продолжение с предыдущей страницы)

```
| public | virtualization_cluster              | table | netbox |
| public | virtualization_clustergroup         | table | netbox |
| public | virtualization_clustertype          | table | netbox |
| public | virtualization_virtualmachine       | table | netbox |
+--------+-------------------------------------+-------+--------+
```

Функции NetBox:

- **IP address management (IPAM)** - IP-префиксы, адреса, VRF'ы и VLAN'ы
- **Equipment racks** - Стойки для оборудования, организованные по сайтам, группам и ролям
- **Devices** - Устройства, их модели, роли, комплектующие и расопложение
- **Connections** - Сетевые, консольные и силовые соединения между устройствами
- **Virtualization** - Виртуальные машины и вычислительные кластера
- **Data circuits** - Подключения к провайдерам
- **Secrets** - Зашифрованное хранилище учётных данных пользователей

В этой статье я коснусь следующих вещей: DCIM - Data Center Infrastructure Management, IPAM - IP Address Management, Виртуализация, дополнительные приятные вещи. Обо всём по порядку.

4.2.1 DCIM

Самая важная часть - это, несомненно, какое оборудование у нас стоит и как оно друг к другу подключено. Но начинается всё с того, **где** оно стоит.

Регионы и сайты (regions/sites)

В парадигме NetBox устройство устанавливается на сайт, сайт принадлежит региону, регионы могут быть вложены. При этом устройство не может быть установлено просто в регионе. Если такая необходимость есть, должен быть заведён отдельный сайт.

Для нашего случая это может (и будет) выглядеть так:

- Россия: Москва, Казань
- Испания: Барселона, Малага
- Китай: Шанхай, Сиань.

Напоминаю где и как мы планировали нашу сеть: АДСМ2. Дизайн сети

Давайте посмотрим, что позволяет API. Вот так можно вывести список всех регионов:

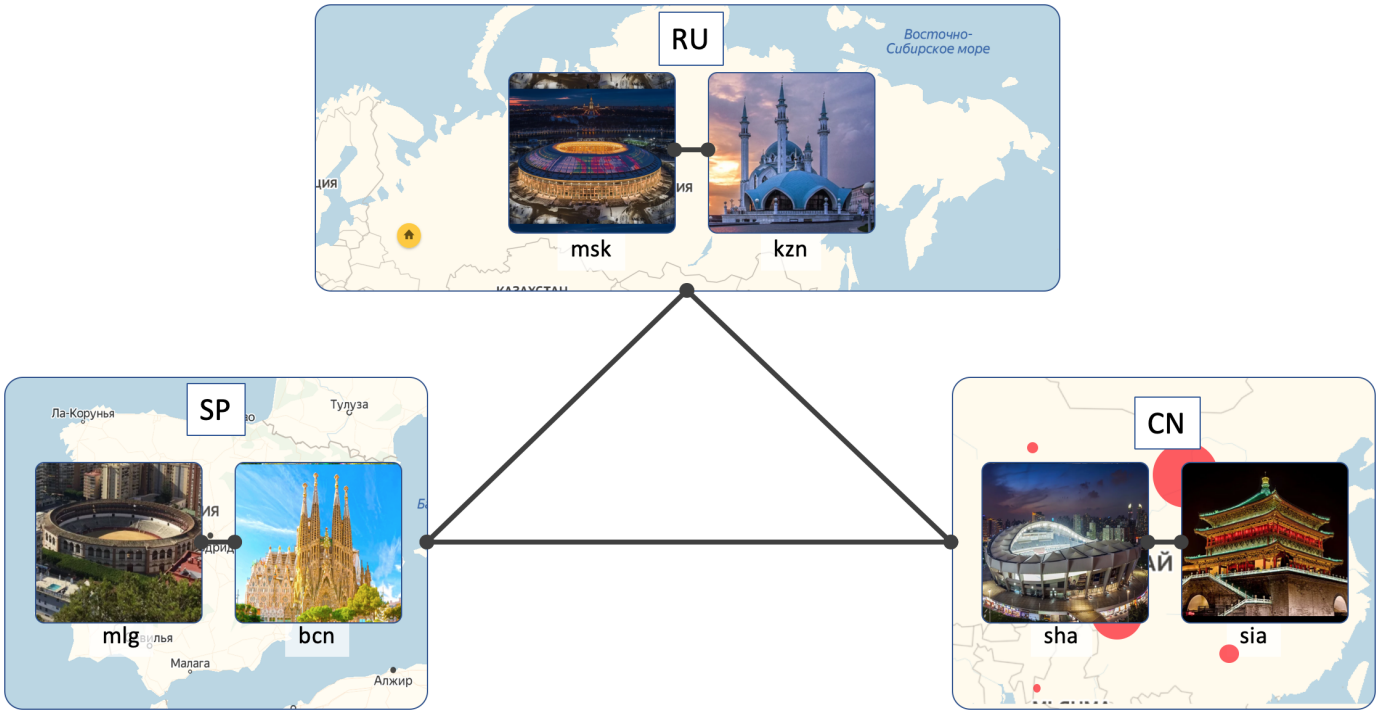
```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/regions/" -H "Accept: application/
↪json; indent=4"
```

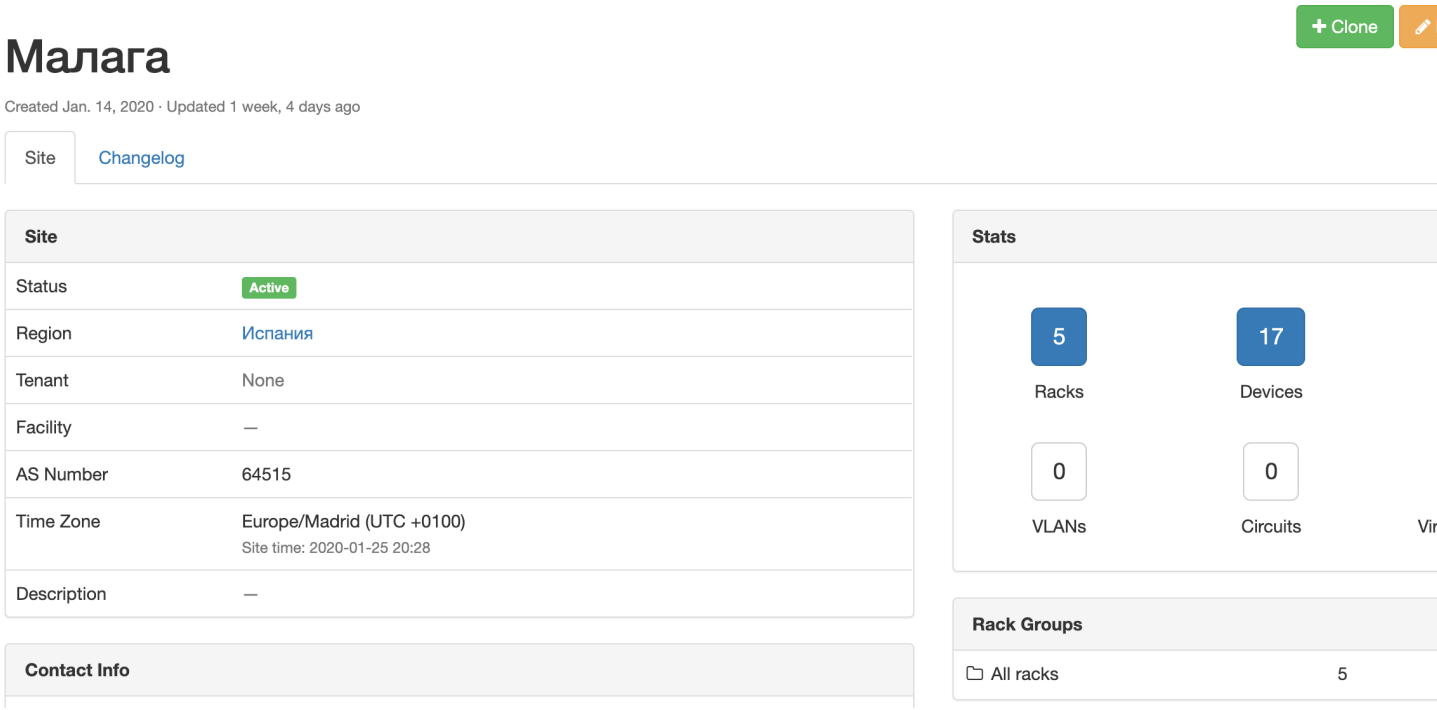
```
nb.dcim.regions.all()
```

Здесь и далее я буду приводить примеры curl и pynetbox без вывода результата. **Не забудьте** слэш в конце URL - без него не заработает. Как использовать pynetbox я рассказывал в статье про RESTful API.

Sites

Name	Status	Facility	Region	Tenant	ASN	Description
Барселона	Active	—	Испания	—	64514	—
Казань	Active	—	Россия	—	64513	—
Малага	Active	—	Испания	—	64515	—
Москва	Active	—	Россия	—	64512	—
Сиань	Active	—	Китай	—	64517	—
Шанхай	Active	—	Китай	—	64516	—





Получить список сайтов:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/sites/" -H "Accept: application/
↪json; indent=4"
```

```
nb.dcim.sites.all()
```

Список сайтов конкретного региона:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/sites/?region=ru" -H "Accept:␣
↪application/json; indent=4"
```

```
nb.dcim.sites.filter(region="ru")
```

Обратите внимание, что поиск идёт не по полному имени, а по так называемому slug. **Slug** - это идентификатор, содержащий только безопасные символы: [0-9A-Za-z-_], который можно использовать в URL. Задаётся он при создании объекта, например, «bcn» вместо «Барселона».

Устройства

Само устройство обладает какой-то ролью, например, leaf, spine, edge, border. Оно, очевидно, является какой-то моделью какого-то вендора. Например, Arista. Таким образом, сначала создаётся вендор, далее внутри него модели. Модель характеризуется именем, набором сервисных интерфейсов, интерфейсом удалённого управления, консольным портом и набором модулей питания.

Помимо коммутаторов, маршрутизаторов и хостов, обладающих Ethernet-интерфейсами, можно создавать консольные сервера.

Devices

Name	Status	Tenant	Site	Rack	Role	Type	IP Add
mlg-edge-0	Active	—	Малага	L	Edge	Juniper vMX	
mlg-edge-1	Active	—	Малага	L	Edge	Juniper vMX	
mlg-host-0	Active	—	Малага	A	Server	Hypermacro Server	
mlg-host-1	Active	—	Малага	A	Server	Hypermacro Server	
mlg-host-2	Active	—	Малага	A	Server	Hypermacro Server	
mlg-host-30	Active	—	Малага	B	Server	Hypermacro Server	
mlg-host-31	Active	—	Малага	B	Server	Hypermacro Server	
mlg-host-150	Active	—	Малага	F	Server	Hypermacro Server	
mlg-host-152	Active	—	Малага	F	Server	Hypermacro Server	
mlg-leaf-0	Active	—	Малага	A	Leaf	Arista vEOS	
mlg-leaf-1	Active	—	Малага	B	Leaf	Arista vEOS	
mlg-leaf-5	Active	—	Малага	F	Leaf	Arista vEOS	
mlg-leaf-6	Active	—	Малага	G	Leaf	Arista vEOS	
mlg-spine-0	Active	—	Малага	L	Spine	Arista vEOS	

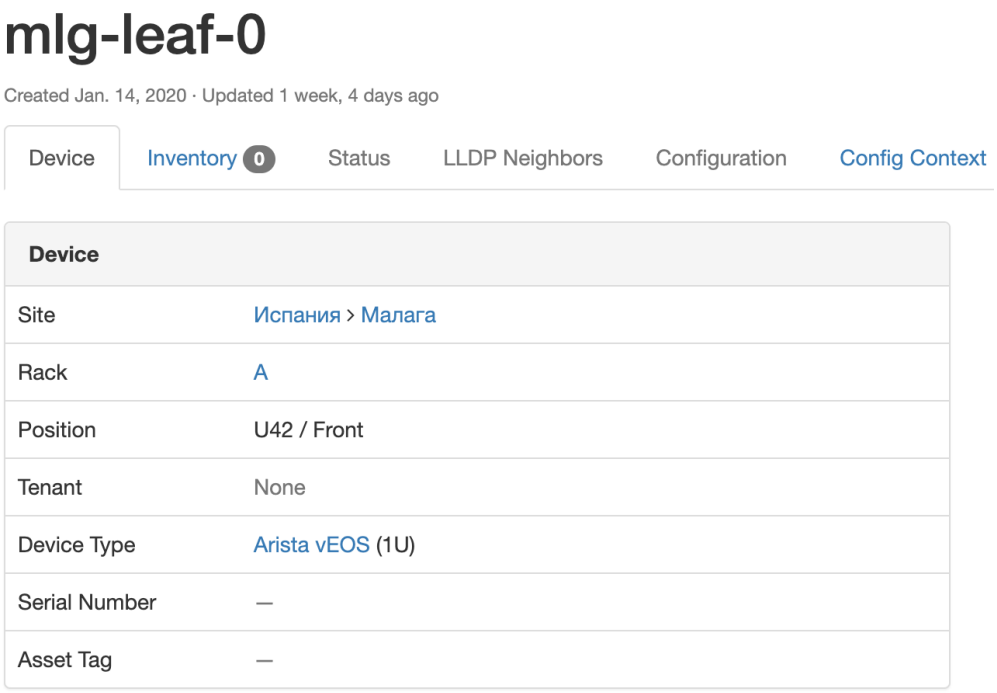
Получить список всех устройств:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/devices/" -H "Accept: application/
↪json; indent=4"
```

```
nb.dcim.devices.all()
```

Всех устройств конкретного сайта:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/devices/?site=mlg" -H "Accept:␣
↪application/json; indent=4"
```




```
nb.dcim.devices.filter(site="mlg")
```

Всех устройств определённой модели

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/devices/?model=veos" -H "Accept:␣
↪application/json; indent=4"
```

```
nb.dcim.devices.filter(device_type_id=2)
```

Всех устройств определённой роли:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/devices/?role=leaf" -H "Accept:␣
↪application/json; indent=4"
```

```
nb.dcim.devices.filter(role="leaf")
```

Устройство может быть в разных статусах: Active, Offline, Planned итд. Все активные устройства:

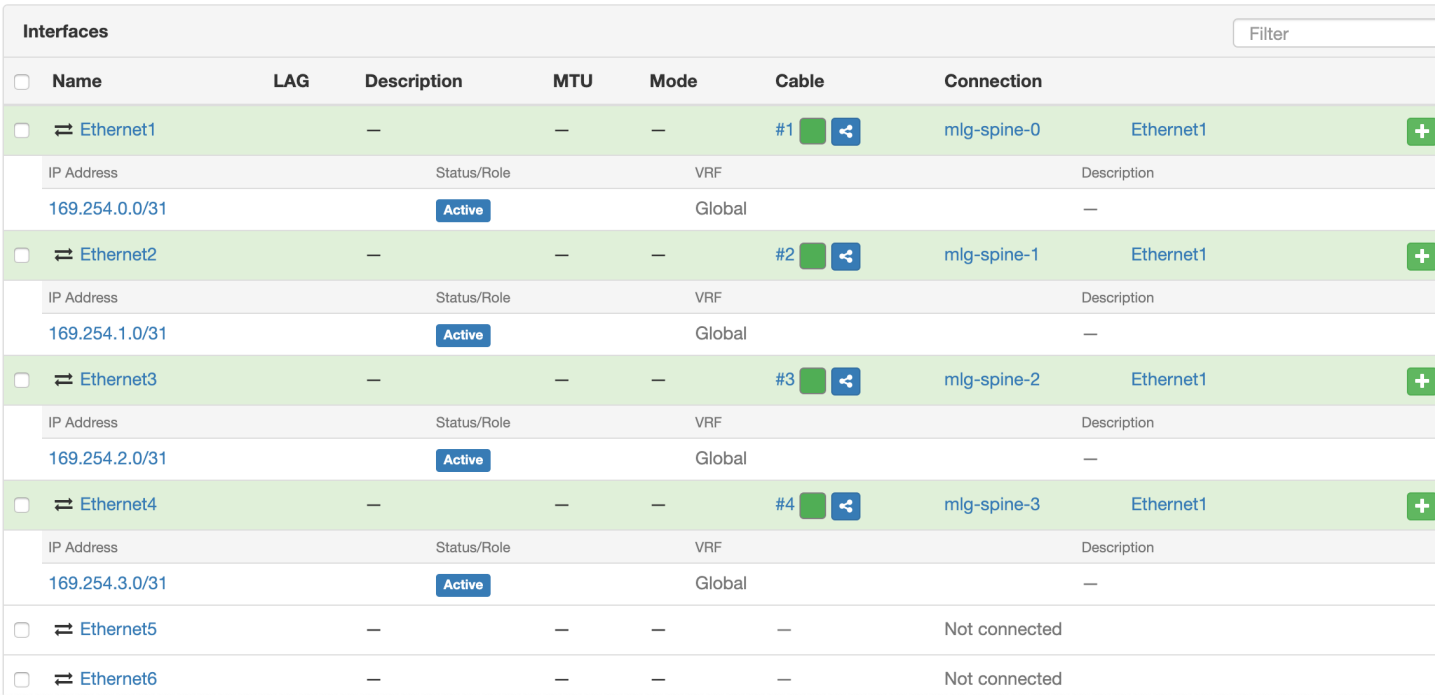
```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/devices/?status=active" -H
↪"Accept: application/json; indent=4"
```

```
nb.dcim.devices.filter(status="active")
```

Интерфейсы

NetBox поддерживает множество типов физических интерфейсов и LAG, однако все виртуальные, такие как Vlan/IRB и loopback объединены под одним типом - Virtual. Каждый интерфейс привязан к какому-либо устройству.

Интерфейсы устройств могут быть подключены друг к другу. Это будет отображаться как в интерфейсе, так и в ответах API (атрибут connected_endpoint).



Интерфейс может быть в различных режимах: Tagged или Access. Соответственно, в него могут быть спущены с тегом или без VLAN'ы - данного сайта или глобальные.

Получить список всех интерфейсов устройства:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/interfaces/?device=mlg-leaf-0" -H
↪"Accept: application/json; indent=4"
```

```
nb.dcim.interfaces.filter(device="mlg-leaf-0")
```

Получить список VLAN'ов конкретного интерфейса.

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/interfaces/?device=mlg-leaf-0&
↪name=Ethernet7" -H "Accept: application/json; indent=4"
```

```
nb.dcim.interfaces.get(device="mlg-leaf-0", name="Ethernet7").untagged_vlan.vid
```

> Обратите внимание, что тут я уже использую метод **get** вместо **filter**. Filter возвращает список, даже если результат - один единственный объект. Get - возвращает один объект или падает с ошибкой, если результатом запроса является список объектов. Поэтому get следует использовать только тогда, когда вы абсолютно уверены, что результат будет в единственном экземпляре. Ещё здесь же прямо после запроса я обращаюсь к атрибутам объекта. Строго говоря, это неправильно: если по запросу ничего не найдено, то pynetbox вернёт None, а у него нет атрибута «untagged_vlan». И ещё обратите внимание, что не везде pynetbox ожидает slug, где-то и name.

Выяснить к какому интерфейсу какого устройства подключен определённый интерфейс:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/interfaces/?device=mlg-leaf-0&
↪name=Ethernet1" -H "Accept: application/json; indent=4"
```

```
iface = nb.dcim.interfaces.get(device="mlg-leaf-0", name="Ethernet1")
iface.connected_endpoint.device
iface.connected_endpoint.name
```

Узнать имя интерфейса управления:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/interfaces/?device=mlg-leaf-0&
↪mgmt_only=true" -H "Accept: application/json; indent=4"
```

```
nb.dcim.interfaces.get(device="mlg-leaf-0", mgmt_only=True)
```

Консольные порты

Консольные порты не являются интерфейсами, поэтому вынесены как отдельные эндпоинты. Порты устройства можно связать с портами консольного сервера.

Выяснить к какому порту какого консольного сервера подключено конкретное устройство.

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/console-ports/?device=mlg-leaf-0"␣
↪-H "Accept: application/json; indent=4"
```

```
nb.dcim.console_ports.get(device="mlg-leaf-0").serialize()
```

Метод **serialize** в pynetbox позволяет преобразовать атрибуты экземпляра класса в словарь.

4.2.2 IPAM

VLAN и VRF

Могут быть привязаны к локации - полезно для VLAN. При создании VRF можно указать, допускается ли пересечение адресного пространства с другими VRF.

Получить список всех VLAN:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/ipam/vlans/" -H "Accept: application/
↪json; indent=4"
```

```
nb.ipam.vlans.all()
```

Получить список всех VRF:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/ipam/vrfs/" -H "Accept: application/
↪json; indent=4"
```

```
nb.ipam.vrfs.all()
```

IP-префиксы

Имеют иерархическую структуру. Может принадлежать какому-либо VRF (если не принадлежит - то Global).

Prefixes

Prefix	Status	VRF	Utilization	Tenant	Site	VLAN	Role	Description
10.0.0.0/8	Active	Global	0%	—	—	—	Underlay	Underlay-адреса дл подключения серве
10.0.0.0/17	Active	Global	0%	—	—	—	Underlay	Underlay для регио
10.0.0.0/19	Active	Global	0%	—	Москва	—	Underlay	Underlay для ДЦ ms
10.0.32.0/19	Active	Global	0%	—	Казань	—	Underlay	Underlay для ДЦ kzn
10.0.128.0/17	Active	Global	0%	—	—	—	Underlay	Underlay для регио
10.0.128.0/19	Active	Global	0%	—	Барселона	—	Underlay	Underlay для ДЦ bc
10.0.160.0/19	Active	Global	0%	—	Малага	—	Underlay	Underlay для ДЦ mlg
10.0.160.0/26	Active	Global	1%	—	Малага	—	Underlay	Underlay для mlg-lea
10.0.160.64/26	Active	Global	1%	—	Малага	—	Underlay	Underlay для mlg-lea
10.0.160.128/26	Active	Global	1%	—	Малага	—	Underlay	Underlay для mlg-lea
10.0.160.192/26	Active	Global	1%	—	Малага	—	Underlay	Underlay для mlg-lea

В NetBox очень удобное визуальное представление свободных префиксов:

Выделить можно просто кликом на зелёную строчку.

Может быть привязан к локации. Можно через API выделить следующий свободный под-префикс нужного размера или следующий свободный IP-адрес. Галочка/параметр «Is a pool» определяет, будет ли при автоматическом выделении выделяться 0-й адрес из этого префикса, или начнётся с 1-го.

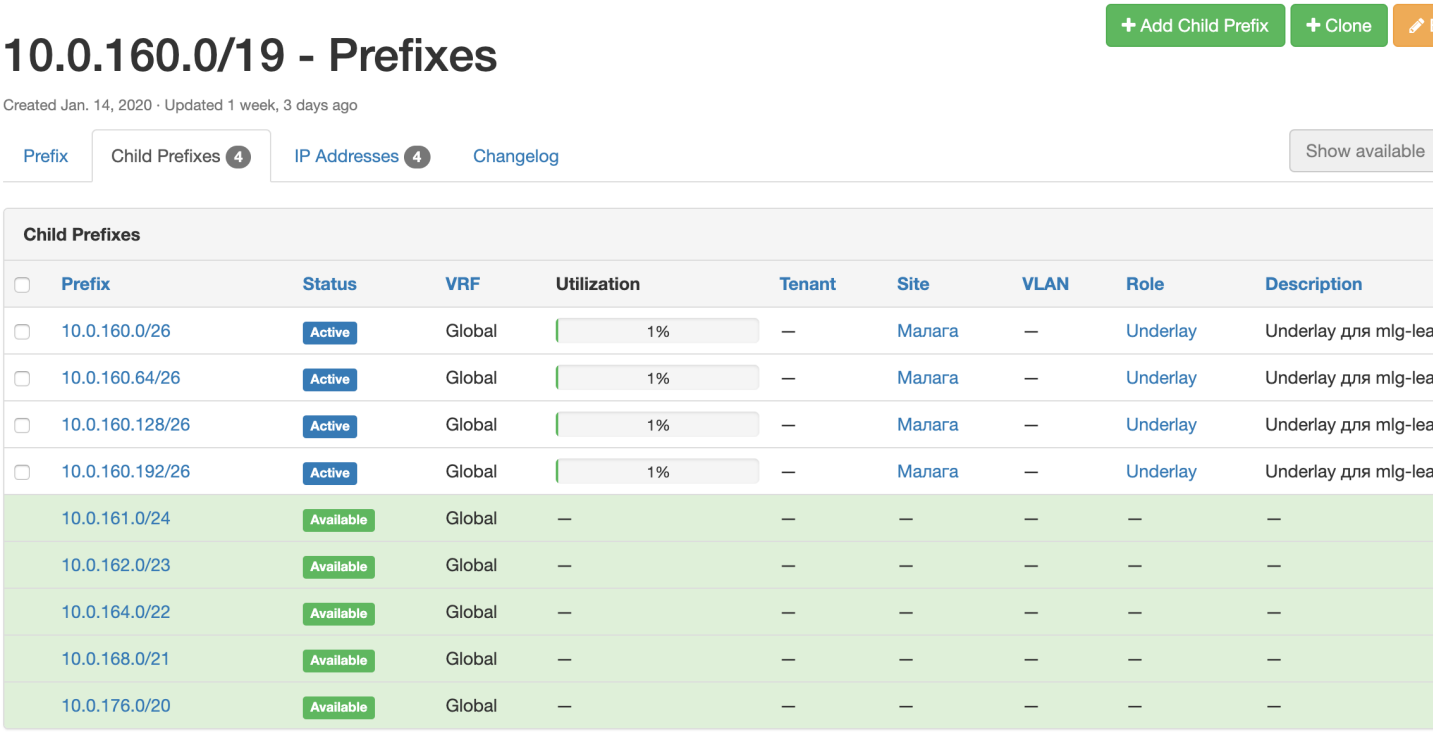
Получить список IP-префиксов сайта Малага с ролью Underlay и длиной 19:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/ipam/prefixes/?site=mlg&role=underlay&
↪mask_length=19" -H "Accept: application/json; indent=4"
```

```
prefix = nb.ipam.prefixes.get(site="mlg", role="underlay", mask_length="19")
```

Получить список свободных префиксов в регионе Россия с ролью Underlay:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/ipam/prefixes/40/available-prefixes/" -
↪H "Accept: application/json; indent=4"
```




```
prefix.available_prefixes.list()
```

Выделить следующий свободный префикс длиной в 24:

```
curl -X POST "http://netbox.linkmeup.ru:45127/api/ipam/prefixes/40/available-prefixes/"␣
↪\
-H "accept: application/json" \
-H "Content-Type: application/json" \
-H "Authorization: TOKEN a9aae70d65c928a554f9a038b9d4703a1583594f" \
-d "{\"prefix_length\": 24}"
```

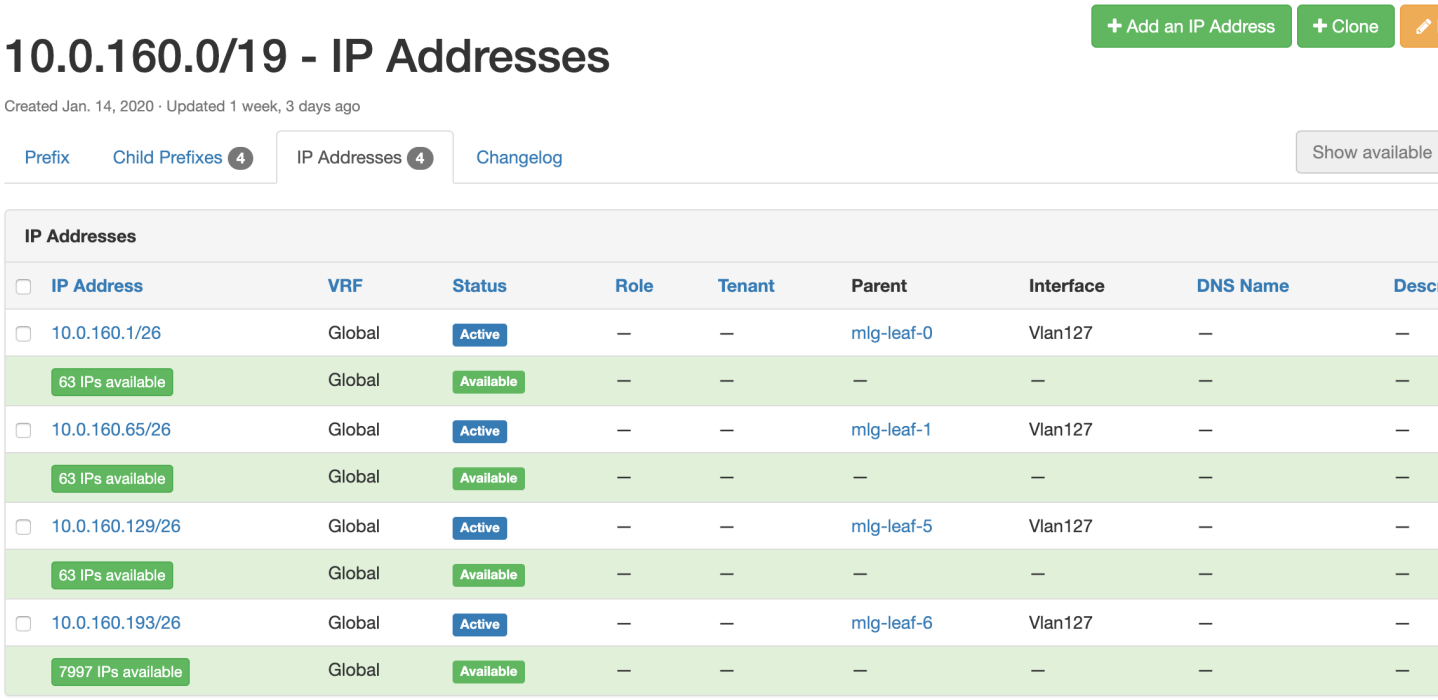
```
prefix.available_prefixes.create({"prefix_length":24})
```

Когда внутри одного объекта нам нужно выделить какой-то дочерний, используется метод POST и нужно указать ID родительского объекта - в данном случае - **40**. Его мы выяснили вызовом из предыдущего примера. В случае pynetbox мы сначала (в предыдущем примере) сохранили результат в переменную **prefix**, а далее обратились к его атрибуту **available_prefixes** и методу **create**. Этот пример у вас **не сработает**, поскольку токен с правом записи уже недействителен.

IP-адреса

Если есть включающий этот адрес префикс, то будут его частью. Могут быть и сами по себе. Могут принадлежать какому-либо VRF или быть в Global. Могут быть привязаны к интерфейсу, а могут висеть в воздухе. Можно выделить следующий свободный IP-адрес в префиксе.

Чтобы сделать это, просто нужно кликнуть по зелёной строчке.



+ Add an IP Address + Clone

10.0.160.0/19 - IP Addresses

Created Jan. 14, 2020 · Updated 1 week, 3 days ago

Prefix | Child Prefixes 4 | IP Addresses 4 | Changelog | Show available

IP Addresses

	IP Address	VRF	Status	Role	Tenant	Parent	Interface	DNS Name	Desc
☐	10.0.160.1/26	Global	Active	—	—	mlg-leaf-0	Vlan127	—	—
	63 IPs available	Global	Available	—	—	—	—	—	—
☐	10.0.160.65/26	Global	Active	—	—	mlg-leaf-1	Vlan127	—	—
	63 IPs available	Global	Available	—	—	—	—	—	—
☐	10.0.160.129/26	Global	Active	—	—	mlg-leaf-5	Vlan127	—	—
	63 IPs available	Global	Available	—	—	—	—	—	—
☐	10.0.160.193/26	Global	Active	—	—	mlg-leaf-6	Vlan127	—	—
	7997 IPs available	Global	Available	—	—	—	—	—	—

Получить список IP-адресов конкретного интерфейса:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/ipam/ip-addresses/?interface_id=8" -H
↪"Accept: application/json; indent=4"
```

```
nb.ipam.ip_addresses.filter(interface_id=8)
```

Или:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/ipam/ip-addresses/?device=mlg-leaf-0&
↪interface=Ethernet1" -H "Accept: application/json; indent=4"
```

```
nb.ipam.ip_addresses.filter(device="mlg-leaf-0", interface="Ethernet1")
```

Получить список всех IP-адресов устройства:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/ipam/ip-addresses/?device=mlg-leaf-0" -
↪H "Accept: application/json; indent=4"
```

```
nb.ipam.ip_addresses.filter(device="mlg-leaf-0")
```

Получить список доступных IP-адресов префикса:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/ipam/prefixes/28/available-ips/" -H
↪"Accept: application/json; indent=4"
```

```
prefix = nb.ipam.prefixes.get(site="mlg", role="leaf-loopbacks")
prefix.available_ips.list()
```

Здесь снова нужно в URL указать ID префикса, из которого выделяем адрес - на сей раз это 28.

Выделить следующий свободный IP-адрес в префиксе:

```
curl -X POST "http://netbox.linkmeup.ru:45127/api/ipam/prefixes/28/available-ips/" \
-H "accept: application/json" \
-H "Content-Type: application/json" \
-H "Authorization: TOKEN a9aae70d65c928a554f9a038b9d4703a1583594f"
```

```
prefix.available_ips.create()
```

4.2.3 Виртуализация

Мы же всё-таки боремся за звание современного ДЦ. Куда же без виртуализации. NetBox не выглядит и не является местом, где стоит хранить информацию о виртуальных машинах (даже о необходимости хранения в нём физических машин можно порассуждать). Однако нам это может оказаться полезным, например, можно занести информация о Route Reflector'ах, о служебных машинах, таких как NTP, Syslog, S-Flow-серверах, о машинах-управляках. ВМ обладает своим списком интерфейсов - они отличны от интерфейсов физических устройств и имеют свой отдельный Endpoint.

Так можно вывести список всех виртуальных машин:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/virtualization/virtual-machines/" -H
↪"Accept: application/json; indent=4"
```

```
nb.virtualization.virtual_machines.all()
```

Так - всех интерфейсов всех ВМ:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/virtualization/interfaces/" -H
↪"Accept: application/json; indent=4"
```

```
nb.virtualization.interfaces.all()
```

Для ВМ нельзя указать конкретный гипервизор/физическую машину, на котором она запущена, но можно указать кластер. Хотя не всё так безнадёжно. Читаем дальше.

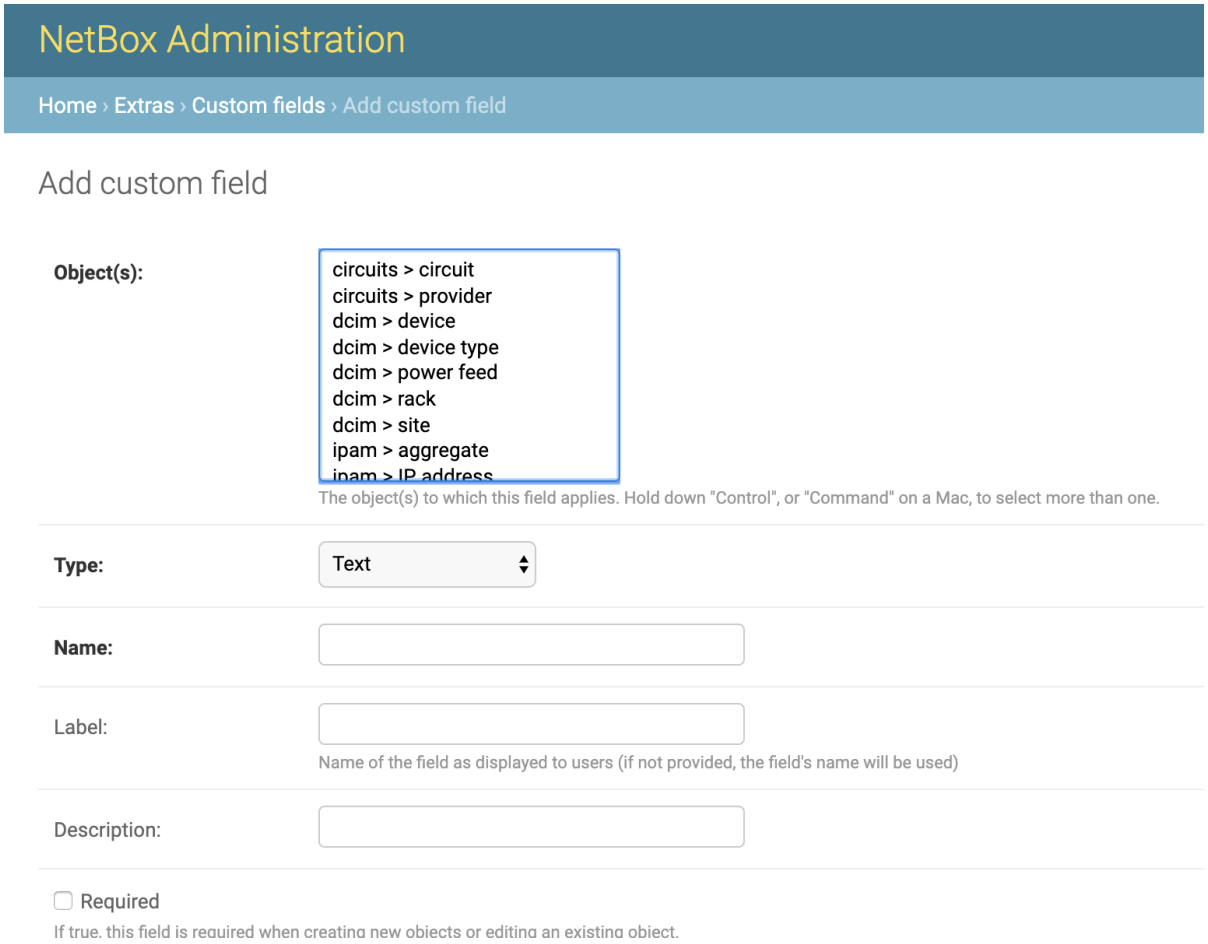
4.2.4 Дополнительные приятные вещи

Основная функциональность NetBox закрывает большинство задач многих пользователей, но не все. Всё-таки изначально продукт написан для решения задач конкретной компании. Однако он активно развивается и новые релизы выходят довольно часто. Соответственно появляются и новые функции. Так, например, с моей первой установки NetBox пару лет назад в нём появились теги, config contexts, webhooks, кэширование, supervisord сменился на systemd, внешние хранилища для файлов.

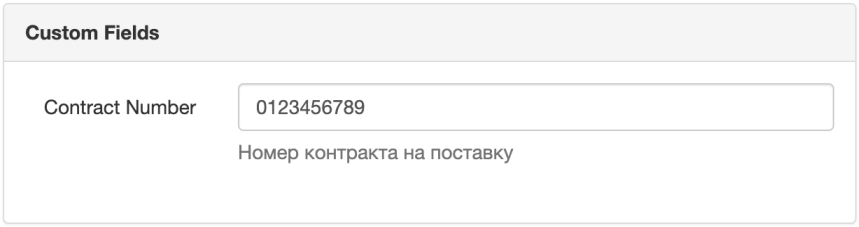
Custom fields

Иногда хочется к какой-либо сущности добавить поле, в которое можно было бы поместить произвольные данные. Например, указать номер договора поставки, по которому был приобретён коммутатор или имя физической машины, на которой запущена ВМ. Тут на помощь и приходит custom fields - как раз такое поле с текстовым значением, которое можно добавить почти к любой сущности в NetBox.

Создаётся Custom fields в админской панели


NetBox Administration
Home › Extras › Custom fields › Add custom field
Add custom field
Object(s):
circuits > circuit
circuits > provider
dcim > device
dcim > device type
dcim > power feed
dcim > rack
dcim > site
ipam > aggregate
ipam > IP address
The object(s) to which this field applies. Hold down "Control", or "Command" on a Mac, to select more than one.
Type:
Text
Name:
Label:
Name of the field as displayed to users (if not provided, the field's name will be used)
Description:
Required
If true, this field is required when creating new objects or editing an existing object.

Вот так это выглядит при редактировании устройства, для которого был создан custom field:



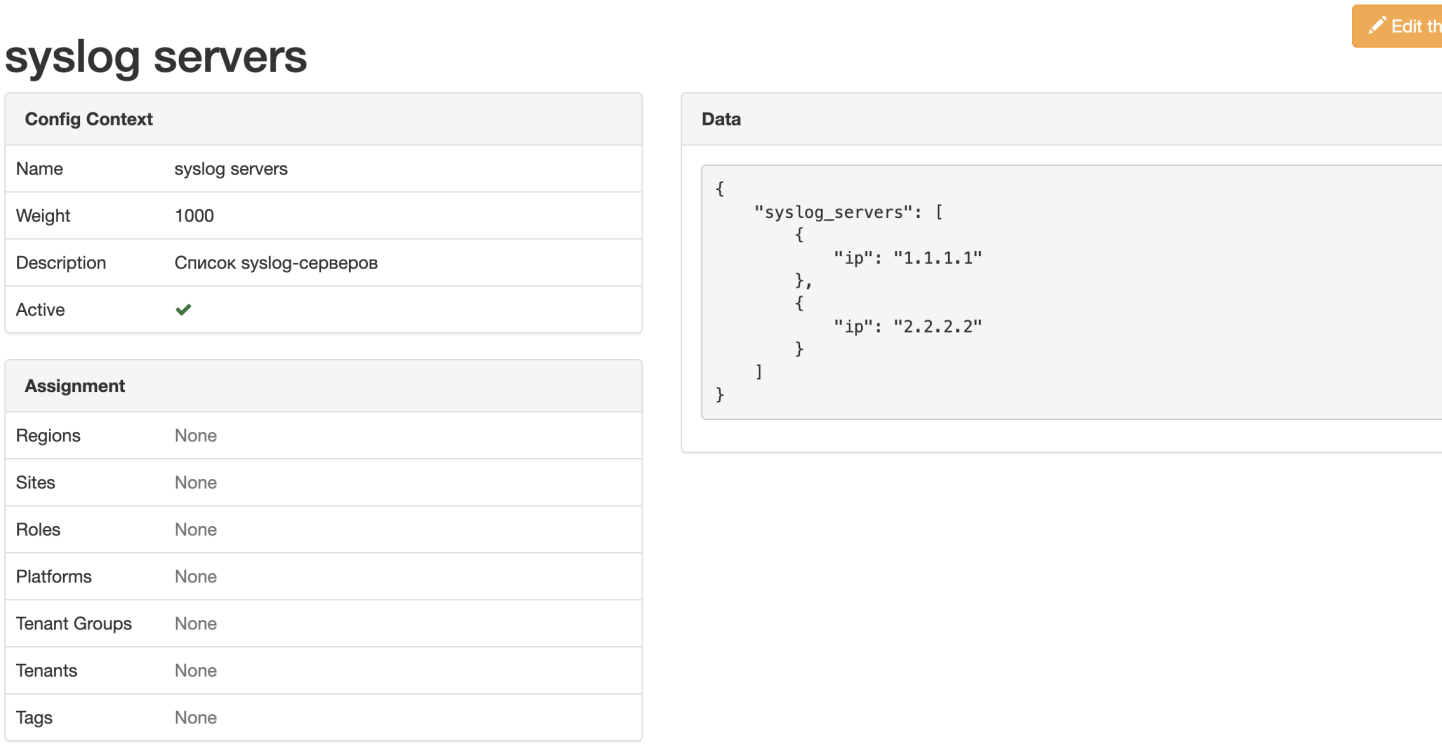
Запросить список устройств по значению custom_field

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/devices/?cf_contract_
↪number=0123456789" -H "Accept: application/json; indent=4"
```

```
nb.dcim.devices.filter(cf_contract_number="0123456789")
```

Config Context

Иногда хочется чего-то большего, чем неструктурированный текст. Тогда на помощь приходит Config Context. Это возможность ввести набор структурированных данных в формате JSON, который больше некуда поместить. Это может быть, например, набор BGP communities или список Syslog-серверов. Config Context может быть локальным - настроенным для конкретного объекта - или глобальным, когда он настраивается однажды, а затем распространяется на все объекты, удовлетворяющие определённым условиям (например, расположенные на одном сайте, или запущенные на одной платформе).



Config Context автоматически добавляется к результатам запроса. При этом локальные и глобальные контексты сливаются в один.

Например, для устройства just a simple russian girl, для которого есть локальный контекст, в выводе будет ключ «config_context»:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/devices/?q=russian" -H "Accept:␣
↪application/json; indent=4"
```

```
"config_context": {
    "syslog_servers": [
        {
            "ip": "1.1.1.1"
        },
        {
            "ip": "2.2.2.2"
        }
    ],
    "ntp_servers": [
        {
            "ip": "3.3.3.3"
        }
    ]
}
```

Теги

Про теги сложно сказать что-то новое. Они есть. Они удобны для добавления какого-либо признака. К примеру, можно пометить тегом «бяда» коммутаторы из партии, в которой сбоит память.

Webhooks

Незаменимая вещь, когда нужно, чтобы об изменениях в NetBox'е узнавали другие сервисы. Например, при заведении нового коммутатора отправляется хука в систему автоматизации, которая запускает процесс настройки устройства и ввода в эксплуатацию.

4.3 Некоторые нюансы установки NetBox

Я не буду описывать процесс инсталляции в деталях - он более чем классно описан в официальной документации.

Посмотреть на процесс запуска docker-образа NetBox и работу в GUI можно в видео Димы Фиголя (раз и два) и Эмиля Гарипова.

В целом, если следовать шагам установки/запуска неукоснительно, то всё получится.
Но вот какие есть нюансы, про которые случайно можно забыть.

- В файле configuration.py должен быть заполнен параметр ALLOWED_HOSTS:

```
ALLOWED_HOSTS = ['netbox.linkmeup.ru', 'localhost']
```

Тут нужно указать все возможные имена NetBox, к которым вы будете обращаться, например, может быть внешний IP-адрес или 127.0.0.1 или DNS-alias.
Если этого не будет сделано, сайт NetBox не откроется и будет показывать 400.

- В этом же файле должен быть указан SECRET_KEY, который можно выдумать самому или сгенерировать скриптом.
- Главная страница будет показывать 502 Bad Gateway, если что-то не так с настройкой базы PostgreSQL: проверьте хост(если ставили на другую машину), порт, имя базы, имя пользователя, пароль.
- С некоторых пор NetBox по умолчанию не даёт никаких прав на чтение без авторизации.
  Изменяется это всё в том же configuration.py:

  ```
  EXEMPT_VIEW_PERMISSIONS = ['*']
  ```

- А ещё API запросы будут возвращать 200 и не работать, если в API URL не будет слэша в конце.

  ```
  curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/devices" -H "Accept:␣
  ↪application/json; indent=4"
  ```

4.4 Немного о PostgreSQL

Для подключения к серверу:

```
psql -U *username* -h *hostname* *db_name*
```

Например:

```
psql -U netbox -h localhost netbox
```

Для вывода всех таблиц:

```
/dt
```

Для выхода:

```
/q
```

Для дампа БД:

```
pg_dump -U *username* -h *hostname* *db_name* > netbox.sql
```

Если не хочется каждый раз вводить пароль:

```
echo *:*:*:*username*:*password* > ~/.pgpass
chmod 600 ~/.pgpass
```

Если у вас есть своя инсталляция и не хочется вносить всё руками, можно просто сделать так, взяв дамп текущей БД NetBox тут:

```
psql -U *username* -h *hostname* *db_name* < netbox_initial_db.sql
```

Если предварительно нужно дропнуть все таблицы (а сделать это придётся), то можно подготовить заранее файл:

```
psql -U *username* -h *hostname* *db_name*
\o drop_all_tables.sql
select 'drop table ' || tablename || ' cascade;' from pg_tables;
\q
psql -U *username* -h *hostname* *db_name* -f drop_all_tables.sql
```

4.5 RESTful API

Эта статья - одна из обещанных коротких заметок по ходу АДСМ. Поскольку основным способом взаимодействия с NetBox будет RESTful API, я решил рассказать о нём отдельно.

Воздаю хвалы архитекторам современного мира - у нас есть стандартизированные интерфейсы. Да их много - это минус, но они есть - это плюс.

Эти интерфейсы взаимодействия обрели имя **API - Application Programming Interface**.

Одним из таких интерфейсов является RESTful API, который мы будем использовать для работы с нашей IPAM/DCIM-системой в будущем.

Если очень просто, то API даёт клиенту набор инструментов, через которые тот может управлять сервером. А клиентом может выступать по сути что угодно: веб-браузер, командная консоль, разработанное производителем приложение, или вообще любое другое приложение, у которого есть доступ к API.

Например, в случае NetBox, добавить новое устройство в него можно следующими способами: через веб-браузер, отправив curl'ом запрос в консоли, использовать Postman, обратиться к библиотеке requests в питоне, воспользоваться SDK pynetbox или перейти в Swagger.

Таким образом, один раз написав единый интерфейс, производитель навсегда освобождает себя от необходимости с каждым новым клиентом договариваться как его подключать (хотя, это самую малость лукавство).

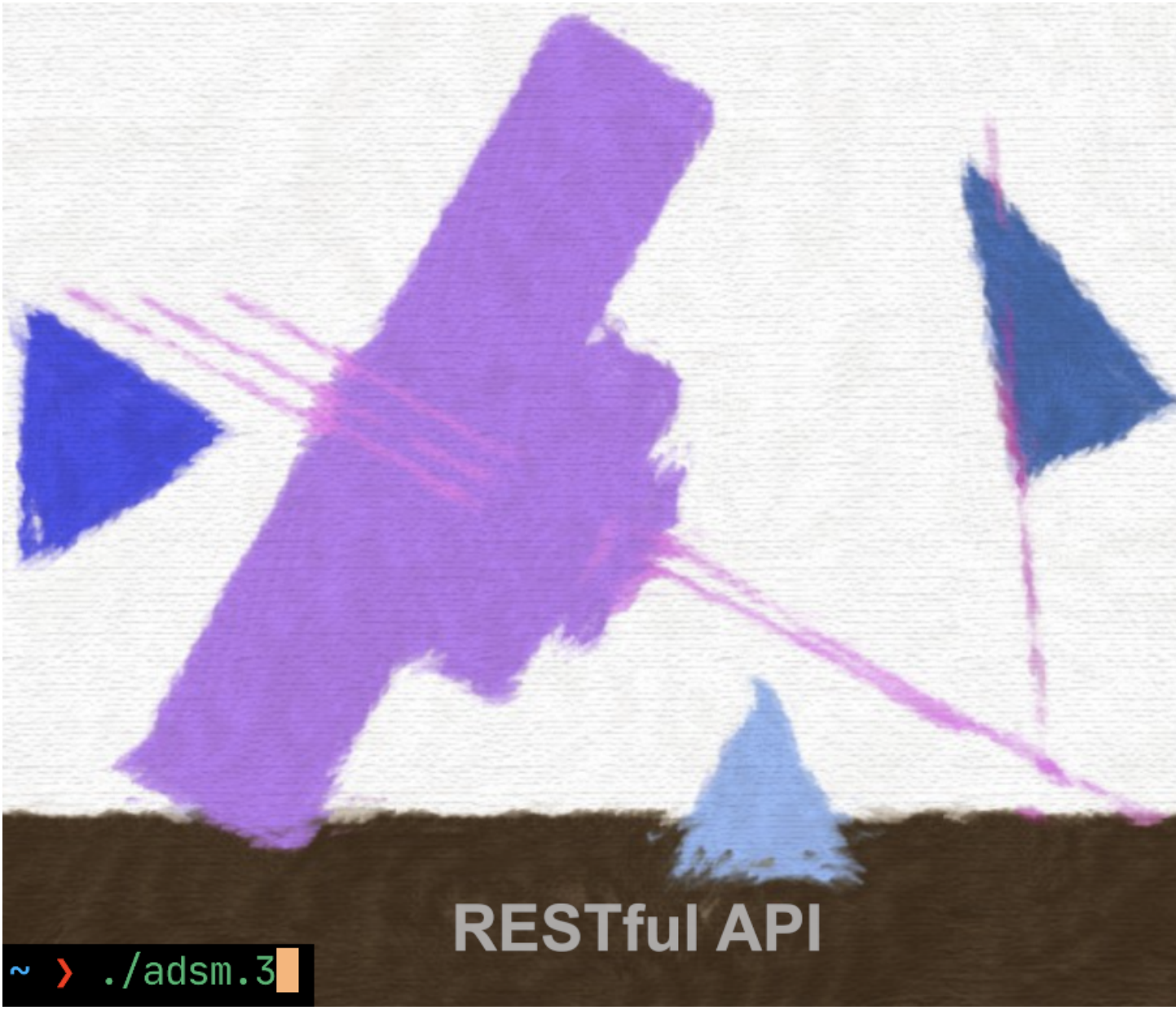
4.5.1 REST, RESTful, API

Ниже я дам очень упрощённое описание того, что такое REST.

Начнём с того, что **RESTful API** - это именно **интерфейс** взаимодействия, основанный на REST, в то время как сам **REST** (**REpresentational State Transfer**) - это набор ограничений, используемых для создания WEB-сервисов. тот самый VNGW

О каких именно ограничениях идёт речь, можно почитать в главе 5 диссертации Роя Филдинга Architectural Styles and the Design of Network-based Software Architectures. Мне же позвольте привести только три наиболее значимых (с моей точки зрения) из них:

- В REST-архитектуре используется модель взаимодействия Клиент-Сервер.
- Каждый REST-запрос содержит всю информацию, необходимую для его выполнения. То есть сервер не должен помнить ничего о предыдущих запросах клиента, что, как известно, характеризуется словом Stateless - не храним информацию о состоянии.


RESTful API
~ ❯ ./adsm.3

- Единый интерфейс. Реализация приложения отделена от сервиса, который оно предоставляет. То есть пользователь знает, что оно делает и как с ним взаимодействовать, но как именно оно это делает не имеет значения. При изменении приложения, интерфейс остаётся прежним, и клиентам не нужно подстраиваться.

WEB-сервисы, удовлетворяющие всем принципам REST, называются **RESTful WEB-services**.
А API, который предоставляют RESTful WEB-сервисы, называется RESTful API.

REST - не протокол, а, так называемый, стиль архитектуры (один из). Развиваемому вместе с HTTP Роем Филдингом, REST'у было предназначено использовать **HTTP 1.1**, в качестве транспорта.
Адрес назначения (или иным словом - объект, или ещё иным - **endpoint**) - это привычный нам URI.
Формат передаваемых данных - **XML** или **JSON**.

> Для этой серии статей на linkmeup развёрнута read-only (для вас, дорогие, читатели) инсталляция NetBox: http://netbox.linkmeup.ru:45127.
>
> На чтение права не требуются, но если хочется попробовать читать с токеном, то можно воспользоваться этим: *API Token: 90a22967d0bc4bdcd8ca47ec490bbf0b0cb2d9c8*.

Давайте интереса ради сделаем один запрос:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/devices/1/" -H "Accept:␣
↪application/json; indent=4"
```

То есть утилитой **curl** мы делаем **GET** объекта по адресу **http://netbox.linkmeup.ru:45127/api/dcim/devices/1/** с ответом в формате **JSON** и отступом в **4** пробела. Или чуть более академически: GET вовзращает типизированный объект **devices**, являющийся параметром объекта **DCIM**.

Этот запрос можете выполнить и вы - просто скопируйте себе в терминал.

> URL, к которому мы обращаемся в запросе, называется **Endpoint**. В некотором смысле это конечный объект, с которым мы будем взаимодействовать. Например, в случае netbox'а список всех endpoint'ов можно получить тут. И ещё обратите внимание на знак / в конце URL - он **обязателен**.

Вот что мы получим в ответ:

```
{
"id": 1,
"name": "mlg-host-0",
"display_name": "mlg-host-0",
"device_type": {
    "id": 4,
    "url": "http://netbox.linkmeup.ru/api/dcim/device-types/4/",
    "manufacturer": {
        "id": 4,
        "url": "http://netbox.linkmeup.ru/api/dcim/manufacturers/4/",
        "name": "Hypermacro",
        "slug": "hypermacro"
    },
    "model": "Server",
    "slug": "server",
    "display_name": "Hypermacro Server"
},
"device_role": {
```

(continues on next page)

(продолжение с предыдущей страницы)


```
    "id": 1,
    "url": "http://netbox.linkmeup.ru/api/dcim/device-roles/1/",
    "name": "Server",
    "slug": "server"
},
"tenant": null,
"platform": null,
"serial": "",
"asset_tag": null,
"site": {
    "id": 6,
    "url": "http://netbox.linkmeup.ru/api/dcim/sites/6/",
    "name": "Малага",
    "slug": "mlg"
},
"rack": {
    "id": 1,
    "url": "http://netbox.linkmeup.ru/api/dcim/racks/1/",
    "name": "A",
    "display_name": "A"
},
"position": 41,
"face": {
    "value": "front",
    "label": "Front",
    "id": 0
},
"parent_device": null,
"status": {
    "value": "active",
    "label": "Active",
    "id": 1
},
"primary_ip": null,
"primary_ip4": null,
"primary_ip6": null,
"cluster": null,
"virtual_chassis": null,
"vc_position": null,
"vc_priority": null,
"comments": "",
"local_context_data": null,
"tags": [],
"custom_fields": {},
"config_context": {},
"created": "2020-01-14",
"last_updated": "2020-01-14T18:39:01.288081Z"
}
```


Это JSON (как мы и просили), описывающий device с ID 1: как называется, с какой ролью, какой модели, где стоит итд.

Так будет выглядеть HTTP-запрос:

```
GET /api/dcim/devices/1/ HTTP/1.1
Host: netbox.linkmeup.ru:45127
User-Agent: curl/7.54.0
```

(continues on next page)

(продолжение с предыдущей страницы)

```
Accept: application/json; indent=4
```

Так будет выглядеть ответ:

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Date: Tue, 21 Jan 2020 15:14:22 GMT
Content-Type: application/json
Content-Length: 1638
Connection: keep-alive

Data
```

Дамп транзакции.

А теперь разберёмся, что же мы натворили.

4.5.2 Структура сообщений HTTP

HTTP-сообщение состоит из трёх частей, только первая из которых является обязательной.

- Стартовая строка
- Заголовки
- Тело сообщения

Стартовая строка

Стартовые строки HTTP-запроса и ответа выглядят по-разному.

HTTP-Запрос

```
METHOD URI HTTP_VERSION
```

Метод определяет, какое действие клиент хочет совершить: получить данные, создать объект, обновить его, удалить. **URI** - идентификатор ресурса, куда клиент обращается или иными словами путь к ресурсу/документу. **HTTP_VERSION** - соответственно версия HTTP. На сегодняшний день для REST это всегда 1.1.

Пример:

```
GET /api/dcim/devices/1/ HTTP/1.1
```

HTTP-Ответ

```
HTTP_VERSION STATUS_CODE REASON_PHRASE
```

HTTP_VERSION - версия HTTP (1.1).
STATUS_CODE - три цифры кода состояния (200, 404, 502 итд)

REASON_PHRASE - Пояснение (OK, NOT FOUND, BAD GATEWAY итд)

Пример:

```
HTTP/1.1 200 OK
```

Заголовки

В заголовках передаются параметры данной HTTP-транзакции.

Например, в примере выше в HTTP-запросе это были:

```
Host: netbox.linkmeup.ru:45127
User-Agent: curl/7.54.0
Accept: application/json; indent=4
```

В них указано, что

- Обращаемся к хосту **netbox.linkmeup.ru:45127**,
- Запрос был сгенерирован в **curl**,
- А принимаем данные в формате **JSON** с отступом **4**.

А вот какие заголовки были в HTTP-ответе:

```
Server: nginx/1.14.0 (Ubuntu)
Date: Tue, 21 Jan 2020 15:14:22 GMT
Content-Type: application/json
Content-Length: 1638
Connection: keep-alive
```

В них указано, что

- Тип сервера: **nginx на Ubuntu**,
- Время формирования ответа,
- Формат данных в ответе: **JSON**
- Длина данных в ответе: **1638 байтов**
- Соединение не нужно закрывать - ещё будут данные.

Заголовки, как вы уже заметили, выглядят как пары имя:значение, разделённые знаком «:».

Полный список возможных заголовков.

Тело HTTP-сообщения

Тело используется для передачи собственно данных. В HTTP-ответе это может быть HTML-страничка, или в нашем случае JSON-объект.

Между заголовками и телом должна быть как минимум одна пустая строка.

При использовании метода GET в HTTP-запросе обычно никакого тела нет, потому что передавать нечего. Но тело есть в HTTP-ответе. А вот например, при POST уже и в запросе будет тело. Давайте о методах и поговорим теперь.

4.5.3 Методы

Как вы уже поняли, для работы с WEB-сервисами HTTP использует методы. То же самое касается и RESTful API. Возможные сценарии описываются термином **CRUD - Create, Read, Update, Delete**. Вот список наиболее популярных методов HTTP, реализующих CRUD:

- HTTP GET
- HTTP POST
- HTTP PUT
- HTTP DELETE
- HTTP PATCH

Методы также называются **глаголами**, поскольку указывают на то, какое действие производится.

Полный список методов.

Давайте на примере NetBox разберёмся с каждым из них.

HTTP GET

Это метод для получения информации.

Так, например, мы забираем список устройств:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/devices/" -H "Accept: application/
↪json; indent=4"
```

Метод GET **безопасный** (**safe**), поскольку не меняет данные, а только запрашивает. Он **идемпотентный** с той точки зрения, что один и тот же запрос всегда возвращает одинаковый результат (до тех пор, пока сами данные не поменялись).

На GET сервер возвращает сообщение с HTTP-кодом и телом ответа (**response code** и **response body**).
То есть если всё OK, то код ответа - 200 (OK).
Если URL не найден - 404 (NOT FOUND).
Если что-то не так с самим сервером или компонентами, это может быть 500 (SERVER ERROR) или 502 (BAD GATEWAY).
Все возможные коды ответов.

Тело возвращается в формате JSON или XML.
Дамп транзакции.

Давайте ещё пару примеров. Теперь мы запросим информацию по конкретному устройству по его имени.

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/devices/?name=mlg-leaf-0" -H
↪"Accept: application/json; indent=4"
```

Здесь, чтобы задать условия поиска в URI я ещё указал атрибут объекта (параметр **name** и его значение **mlg-leaf-0**). Как вы можете видеть, перед условием и после слэша идёт знак **«?»**, а имя и значение разделяются знаком «=».

Так выглядит запрос.

```
GET /api/dcim/devices/?name=mlg-leaf-0 HTTP/1.1
Host: netbox.linkmeup.ru:45127
User-Agent: curl/7.54.0
Accept: application/json; indent=4
```

Дамп транзакции.

Если нужно задать пару условий, то запрос будет выглядеть так:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/devices/?role=leaf&site=mlg" -H
↪"Accept: application/json; indent=4"
```

Здесь мы запросили все устройства с ролью **leaf**, расположенные на сайте **mlg**. То есть два условия отделяются друг от друга знаком **«&»**.

Дамп транзакции.

Из любопытного и приятного - если через «&» задать два условия с одним именем, то между ними будет на самом деле не логическое «И», а логическое «ИЛИ».
То есть вот такой запрос и в самом деле вернёт два объекта: mlg-leaf-0 и mlg-spine-0

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/devices/?name=mlg-leaf-0&name=mlg-
↪spine-0" -H "Accept: application/json; indent=4"
```

Дамп транзакции.

Попробуем обратиться к несуществующему URL.

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/IDGAF/" -H "Accept: application/
↪json; indent=4"
```

Трамп транзакции.

HTTP POST

POST используется для создания нового объекта в наборе объектов. Или более сложным языком: для создания нового подчинённого ресурса.
То есть, если есть набор devices, то POST позволяет создать новый объект device внутри devices.

Выберем тот же Endpoint и с помощью POST создадим новое устройство.

```
curl -X POST "http://netbox.linkmeup.ru:45127/api/dcim/devices/" \
-H "accept: application/json"\
-H "Content-Type: application/json" \
-H "Authorization: TOKEN a9aae70d65c928a554f9a038b9d4703a1583594f" \
-d "{ \"name\": \"just a simple russian girl\", \"device_type\": 1, \"device_role\": 1,␣
↪\"site\": 3,       \"rack\": 3, \"position\": 5, \"face\": \"front\"}"
```

Здесь уже появляется заголовок **Authorization**, содержащий токен, который авторизует запрос на запись, а после директивы **-d** расположен JSON с параметрами создаваемого устройства:

```
{
    "name": "just a simple russian girl",
    "device_type": 1,
    "device_role": 1,
    "site": 3,
    "rack": 3,
    "position": 5,
    "face": "front"}
```

Запрос у вас **не сработает**, потому что токен уже не валиден - не пытайтесь записать в мой NetBox.

В ответ приходит HTTP-ответ с кодом 201 (CREATED) и JSON'ом в теле сообщения, где сервер возвращает все параметры о созданном устройстве.

```
HTTP/1.1 201 Created
Server: nginx/1.14.0 (Ubuntu)
Date: Sat, 18 Jan 2020 11:00:22 GMT
Content-Type: application/json
Content-Length: 1123
Connection: keep-alive

JSON
```

Дамп транзакции.

Теперь новым запросом с методом GET можно его увидеть в выдаче:

```
curl -X GET "http://netbox.linkmeup.ru:45127/api/dcim/devices/?q=russian" -H "Accept:␣
↪application/json; indent=4"
```

«q» в NetBox'е позволяет найти все объекты, содержащие в своём названии строку, идущую дальше.

POST, очевидно, не является **ни безопасным, ни идемпотентным** - он наверняка меняет данные, и дважды выполненный запрос приведёт или к созданию второго такого же объекта, или к ошибке.

HTTP PUT

Это метод для изменения существующего объекта. Endpoint для PUT выглядит иначе, чем для POST - в нём теперь содержится конкретный объект.
PUT может возвращать коды 201 или 200.

Важный момент с этим методом: нужно передавать все обязательные атрибуты, поскольку PUT замещает собой старый объект.
Поэтому, если например, просто попытаться добавить атрибут asset_tag нашему новому устройству, то получим ошибку:

```
curl -X PUT "http://netbox.linkmeup.ru:45127/api/dcim/devices/18/" \
-H "accept: application/json" \
-H "Content-Type: application/json" \
-H "Authorization: TOKEN a9aae70d65c928a554f9a038b9d4703a1583594f" \
-d "{ \"asset_tag\": \"12345678\"}"
```

Вот такую:

```
{"device_type":["This field is required."],"device_role":["This field is required."],
↪"site":["This field is required."]}
```

Но если добавить недостающие поля, то всё сработает:

```
curl -X PUT "http://netbox.linkmeup.ru:45127/api/dcim/devices/18/" \
-H "accept: application/json" \
-H "Content-Type: application/json" \
-H "Authorization: TOKEN a9aae70d65c928a554f9a038b9d4703a1583594f" \
-d "{ \"name\": \"just a simple russian girl\", \"device_type\": 1, \"device_role\": 1,␣
↪\"site\": 3,       \"rack\": 3, \"position\": 5, \"face\": \"front\", \"asset_tag\": \
↪"12345678\"}"
```

Трап транзакции.

Обратите внимание на URL здесь - теперь он включает ID устройства, которое мы хотим менять (**18**).

HTTP PATCH

Этот метод используется для частичного изменения ресурса.
WAT? Спросите вы, а как же PUT?

PUT - изначально существовавший в стандарте метод, предполагающий полную замену изменяемого объекта. Соответственно в методе PUT, как я и писал выше, придётся указать даже те атрибуты объекта, которые не меняются.

A PATCH был добавлен позже и позволяет указать только те атрибуты, которые действительно меняются.

Например:

```
curl -X PATCH "http://netbox.linkmeup.ru:45127/api/dcim/devices/18/" \
-H "accept: application/json" \
-H "Content-Type: application/json" \
-H "Authorization: TOKEN a9aae70d65c928a554f9a038b9d4703a1583594f" \
-d "{ \"serial\": \"BREAKINGBAD\"}"
```

Здесь также в URL указан ID устройства, но для изменения только один атрибут **serial**.

Труп транзакции.

HTTP DELETE

Очевидно, удаляет объект.

Пример.

```
curl -X DELETE "http://netbox.linkmeup.ru:45127/api/dcim/devices/21/" \
-H "accept: application/json" \
-H "Content-Type: application/json" \
-H "Authorization: TOKEN a9aae70d65c928a554f9a038b9d4703a1583594f"
```

Метод DELETE идемпотентен с той точки зрения, что повторно выполненный запрос уже ничего не меняет в списке ресурсов (но вернёт код 404 (NOT FOUND).

```
curl -X DELETE "http://netbox.linkmeup.ru:45127/api/dcim/devices/21/" \
-H "accept: application/json" \
-H "Content-Type: application/json" \
-H "Authorization: TOKEN a9aae70d65c928a554f9a038b9d4703a1583594f"
```

```
{"detail":"Not found."}
```

4.5.4 Способы работы с RESTful API

Curl - это, конечно, очень удобно для доблестных воинов CLI, но есть инструменты получше:

- Графическая утилита Postman
- Библиотека requests в Python
- Python SDK для NetBox Pynetbox
- API-фреймворк Swagger

Postman

Postman позволяет в графическом интерфейсе формировать запросы, выбирая методы, заголовки, тело, и отображает результат в удобочитаемом виде.
Кроме того запросы и URI можно сохранять и возвращаться к ним позже.

Скачать Postman на оф.сайте.

Так мы можем сделать GET:

Здесь указан Token в GET только для примера.

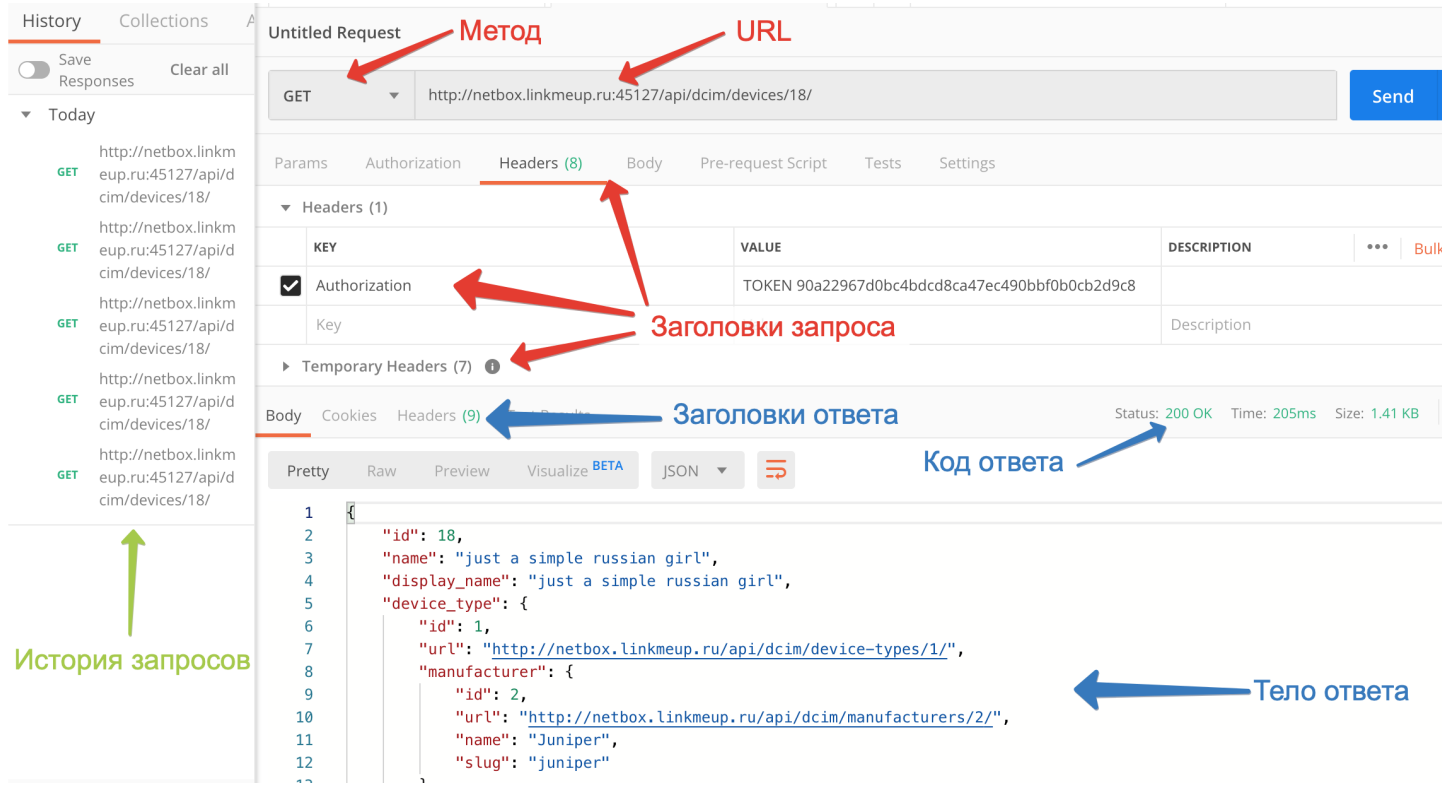
А так POST:

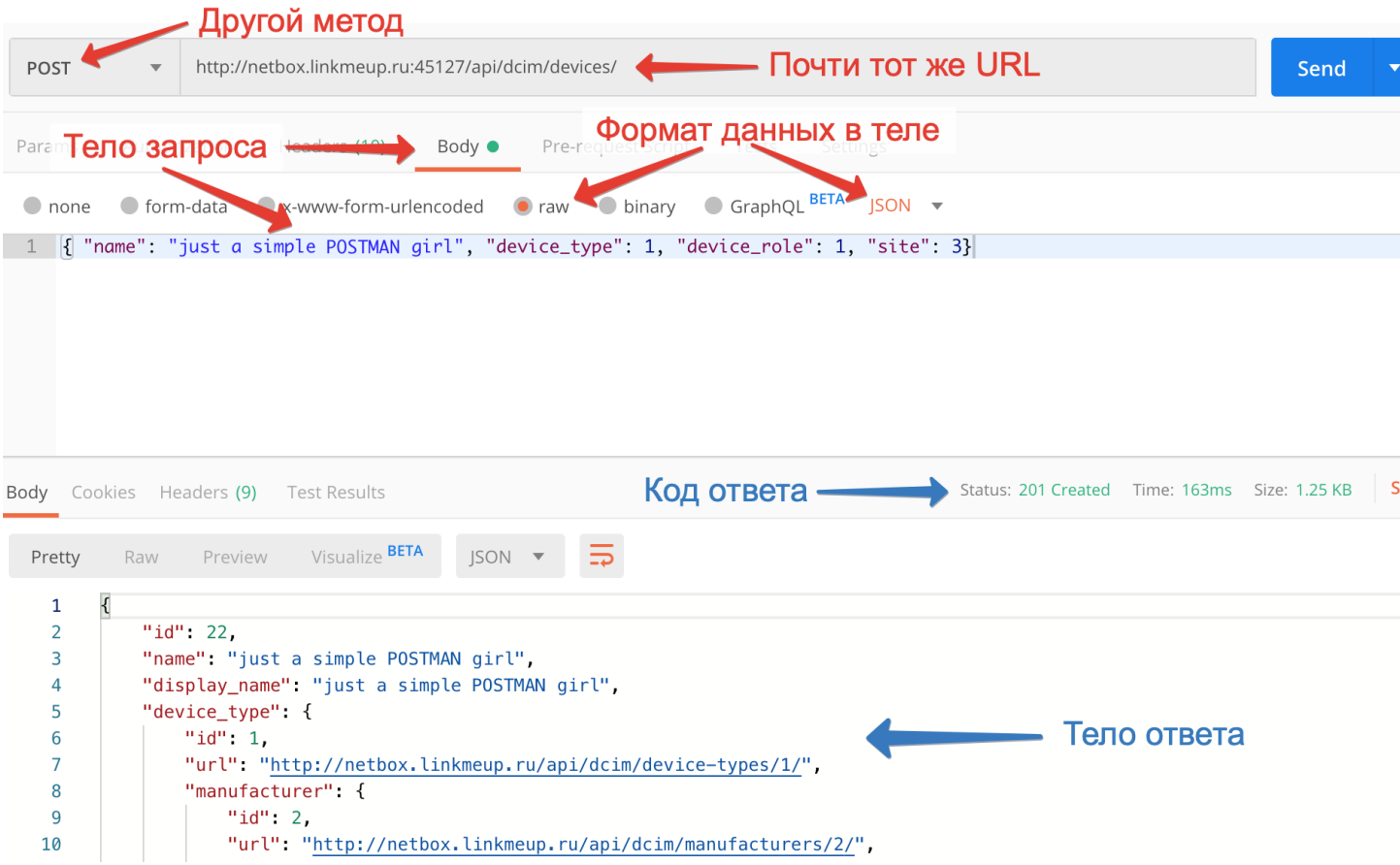
Postman служит только для работы с RESTful API.

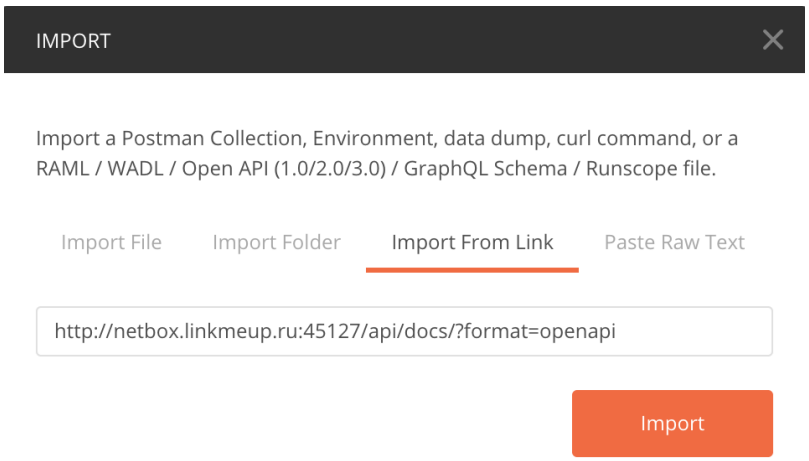
> Например, не пытайтесь через него отправить NETCONF XML, как это делал я на заре своей автоматизационной карьеры.

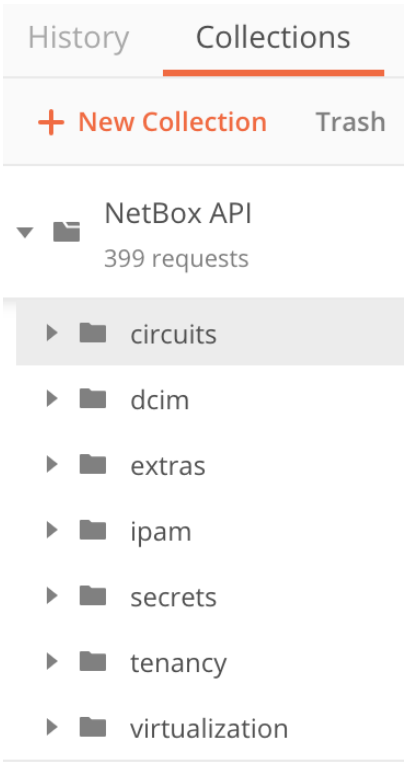
Один из приятных бонусов специфицированного API в том, что вы можете в Postman импортировать все эндпоинты и их методы как коллекцию.
Для этого в окне **Import** (File->Import) выберите **Import From Link** и вставьте в окно URL http://netbox.linkmeup.ru:45127/api/docs/?format=openapi.

Далее, всё, что только можно, вы найдёте в коллекциях.


History
Collections
Save Responses
Clear all
Today
GET http://netbox.linkmeup.ru:45127/api/dcim/devices/18/
Untitled Request
Метод
URL
GET
http://netbox.linkmeup.ru:45127/api/dcim/devices/18/
Send
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
Headers (1)
KEY
VALUE
DESCRIPTION
Authorization
TOKEN 90a22967d0bc4bdcd8ca47ec490bbf0b0cb2d9c8
Key
Description
Temporary Headers (7)
Заголовки запроса
Body
Cookies
Headers (9)
Заголовки ответа
Status: 200 OK
Time: 205ms
Size: 1.41 KB
Код ответа
Pretty
Raw
Preview
Visualize BETA
JSON
{
"id": 18,
"name": "just a simple russian girl",
"display_name": "just a simple russian girl",
"device_type": {
"id": 1,
"url": "http://netbox.linkmeup.ru/api/dcim/device-types/1/",
"manufacturer": {
"id": 2,
"url": "http://netbox.linkmeup.ru/api/dcim/manufacturers/2/",
"name": "Juniper",
"slug": "juniper"
История запросов
Тело ответа


Другой метод
POST
http://netbox.linkmeup.ru:45127/api/dcim/devices/
Почти тот же URL
Send
Тело запроса
Body
Формат данных в теле
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL BETA
JSON
{ "name": "just a simple POSTMAN girl", "device_type": 1, "device_role": 1, "site": 3}
Body
Cookies
Headers (9)
Test Results
Код ответа
Status: 201 Created
Time: 163ms
Size: 1.25 KB
Pretty
Raw
Preview
Visualize BETA
JSON
{
"id": 22,
"name": "just a simple POSTMAN girl",
"display_name": "just a simple POSTMAN girl",
"device_type": {
"id": 1,
"url": "http://netbox.linkmeup.ru/api/dcim/device-types/1/",
"manufacturer": {
"id": 2,
"url": "http://netbox.linkmeup.ru/api/dcim/manufacturers/2/",
Тело ответа


IMPORT
Import a Postman Collection, Environment, data dump, curl command, or a RAML / WADL / Open API (1.0/2.0/3.0) / GraphQL Schema / Runscope file.
Import File
Import Folder
Import From Link
Paste Raw Text
http://netbox.linkmeup.ru:45127/api/docs/?format=openapi
Import


History
Collections
New Collection
Trash
NetBox API
399 requests
circuits
dcim
extras
ipam
secrets
tenancy
virtualization

Python+requests

Но даже через Postman вы, скорее всего, не будете управлять своими Production-системами. Наверняка, у вас будут внешние приложения, которые захотят без вашего участия взаимодействовать с ними.

Например, система генерации конфигурации захочет забрать список IP-интерфейсов из NetBox.

В Python есть чудесная библиотека **requests**, которая реализует работу через HTTP.

Пример запроса списка всех устройств:

```
import requests

HEADERS = {'Content-Type': 'application/json', 'Accept': 'application/json'}
NB_URL = "http://netbox.linkmeup.ru:45127"

request_url = f"{NB_URL}/api/dcim/devices/"
devices = requests.get(request_url, headers = HEADERS)
print(devices.json())
```

Код скрипта на github.

Снова добавим новое устройство:

```
import requests

API_TOKEN = "a9aae70d65c928a554f9a038b9d4703a1583594f"
HEADERS = {'Authorization': f'Token {API_TOKEN}', 'Content-Type': 'application/json',
↪'Accept': 'application/json'}
NB_URL = "http://netbox.linkmeup.ru:45127"

request_url = f"{NB_URL}/api/dcim/devices/"

device_parameters = {
    "name": "just a simple REQUESTS girl",
    "device_type": 1,
    "device_role": 1,
    "site": 3,
}
new_device = requests.post(request_url, headers = HEADERS, json=device_parameters)
print(new_device.json())
```

Код скрипта на github.

Python+NetBox SDK

В случае NetBox есть также Python SDK - Pynetbox, который представляет все Endpoint'ы NetBox в виде объекта и его атрибутов, делая за вас всю грязную работу по формированию URI и парсингу ответа, хотя и не бесплатно, конечно.

Например, сделаем то же, что и выше, использую pynetbox. Список всех устройств:

```
import pynetbox

NB_URL = "http://netbox.linkmeup.ru:45127"
nb = pynetbox.api(NB_URL)

```

(continues on next page)

(продолжение с предыдущей страницы)

```
devices = nb.dcim.devices.all()
print(devices)
```

Кот скрипта на github.

Добавить новое устройство:

```
import pynetbox

API_TOKEN = "a9aae70d65c928a554f9a038b9d4703a1583594f"
NB_URL = "http://netbox.linkmeup.ru:45127"
nb = pynetbox.api(NB_URL, token = API_TOKEN)

device_parameters = {
    "name": "just a simple PYNETBOX girl",
    "device_type": 1,
    "device_role": 1,
    "site": 3,
}
new_device = nb.dcim.devices.create(**device_parameters)
print(new_device)
```

Скот скрипта на github.

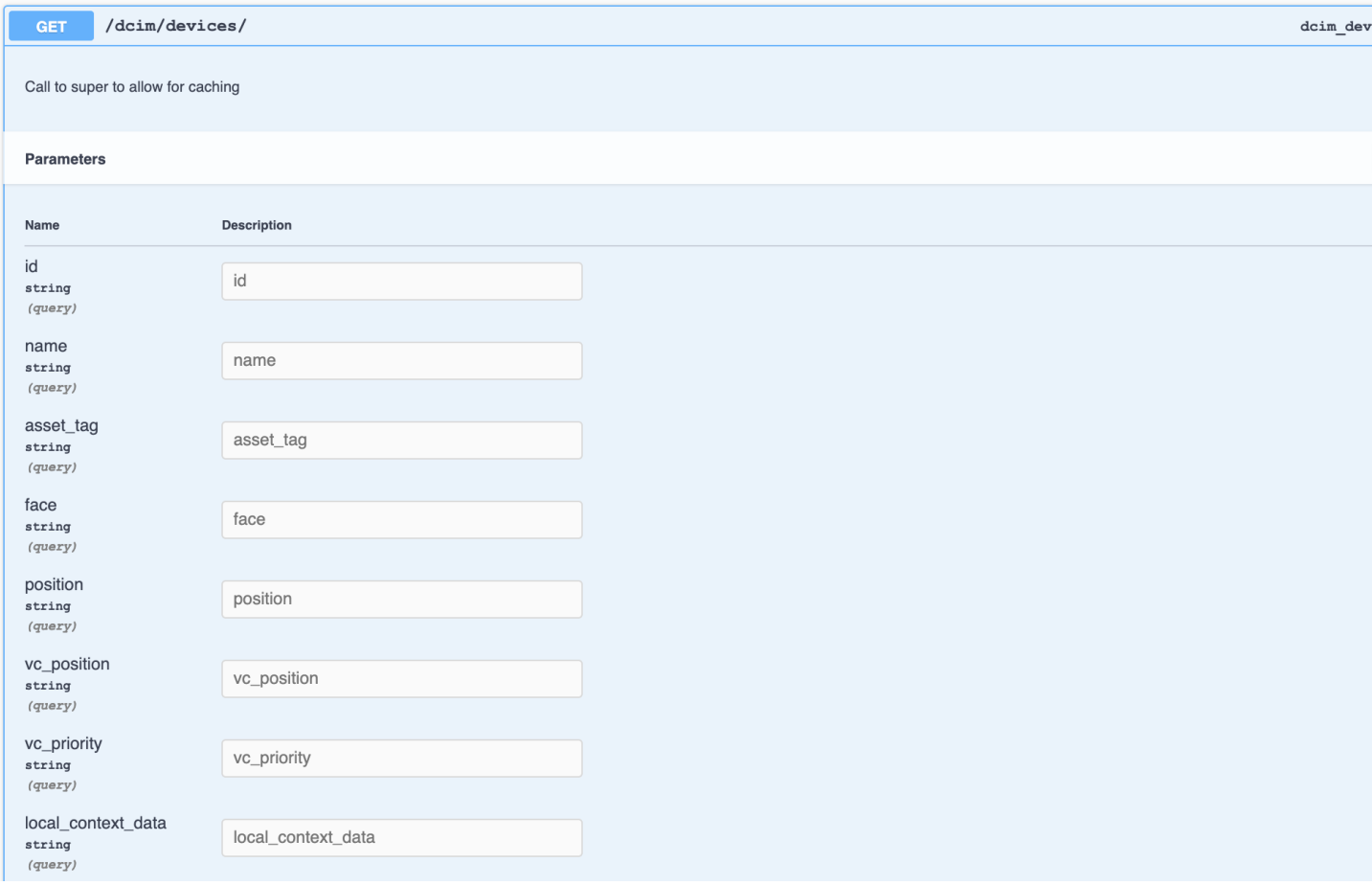
Документация по Pynetbox.

SWAGGER

За что ещё стоит поблагодарить ушедшее десятилетие, так это за спецификации API. Если вы перейдёте по этому пути, то попадёте в Swagger UI - документацию по API Netbox.

На этой странице перечислены все Endpoint'ы, методы работы с ними, возможные параметры и атрибуты и указано, какие из них обязательны. Кроме того описаны ожидаемые ответы.

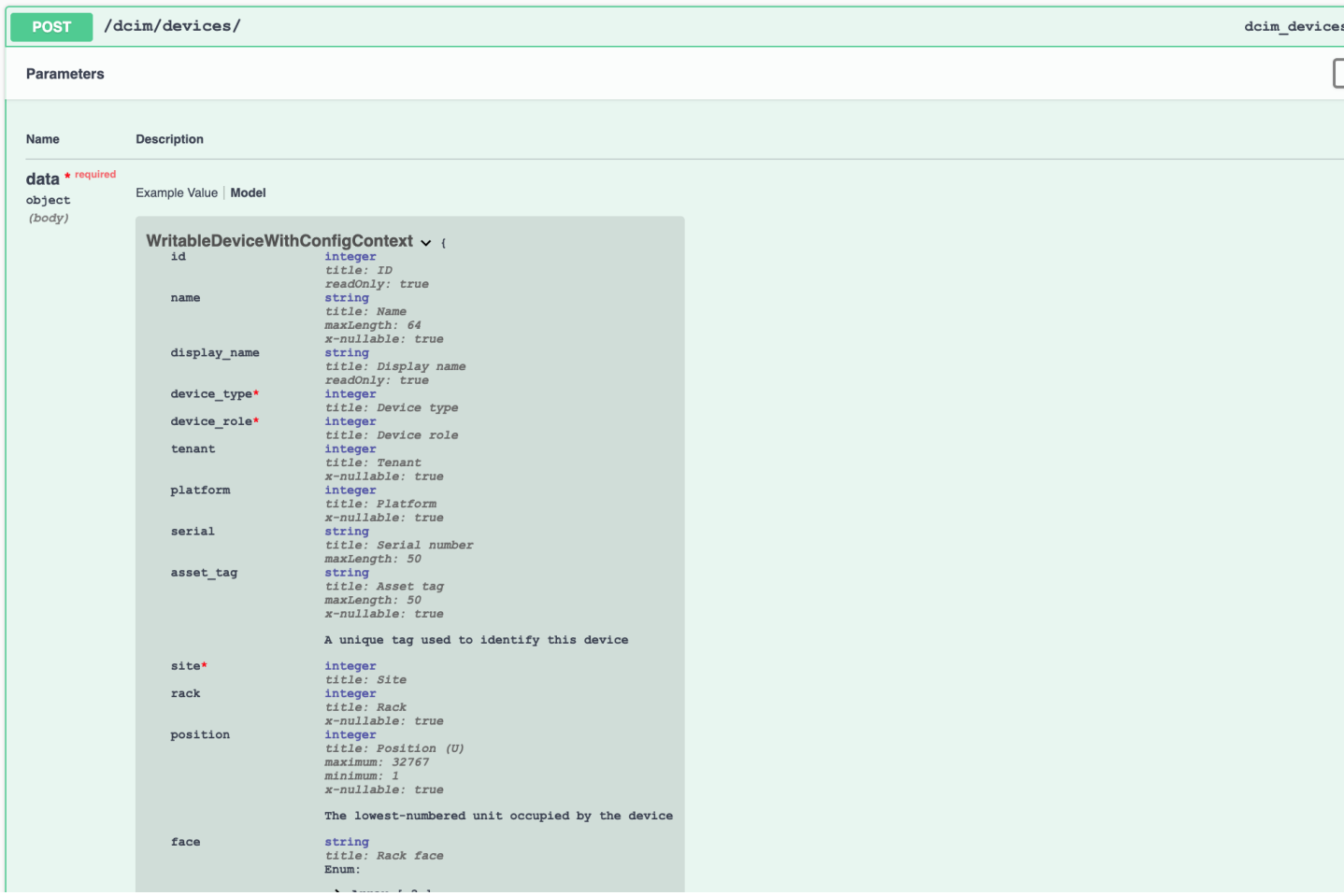
На этой же странице можно выполнять интерактивные запросы, кликнув на **Try it out**.


GET
/dcim/devices/
dcim_dev
Call to super to allow for caching
Parameters
Name
Description
id
string
(query)
id
name
string
(query)
name
asset_tag
string
(query)
asset_tag
face
string
(query)
face
position
string
(query)
position
vc_position
string
(query)
vc_position
vc_priority
string
(query)
vc_priority
local_context_data
string
(query)
local_context_data

По какой-от причине swagger в качестве Base URL берёт имя сервера без порта, поэтому функция Try it out не работает в моих примерах со Swagger'ом. Но вы можете попробовать это на собственной инсталляции.

При нажатии на **Execute** Swagger UI сформирует строку curl, с помощью которой можно аналогичный запрос сделать из командной строки.

В Swagger UI можно даже создать объект:



Для этого достаточно быть авторизованным пользователем, обладающим нужными правами.

То, что мы видим на этой странице - это Swagger UI - документация, сгенерированная на основе спецификации API.

С трендами на микросервисную архитектуру всё более важным становится иметь стандартизированный API для взаимодействия между компонентами, эндпоинты и методы которого легко определить как человеку, так и приложению, не роясь в исходном коде или PDF-документации.

Поэтому разработчики сегодня всё чаще следуют парадигме API First, когда сначала задумываются об API, а уже потом о реализации.

В этом дизайне сначала специфицируется API, а затем из него **генерируются** документация, клиентское приложение, серверная часть и необходимы тесты.

Swagger - это фреймворк и язык спецификации (который ныне переименован в OpenAPI 2.0), позволяющие реализовать эту задачу.

Углубляться в него я не буду.

За большими деталями сюда:

- Сайт Swagger
- Пример использования
- Wiki про Open API

4.5.5 Заключение

Не всё в том мире 2000-го года так уже радужно. REST по праву часто критикуют. Не являясь экспертом, не берусь предметно раскрывать вопрос, но дам ссылку на небесспорную статью на хабре.

Альтернативным интерфейсом взаимодействия компонентов системы сегодня является gRPC. Ему же пророчат большое будущее на ниве новых подходов к работе с сетевым оборудованием. Но о нём мы поговорим когда-то в будущем, когда придёт его черёд.

Можно также взглянуть на многообещающий GraphQL, но нам опять же нет нужды с ним работать пока, поэтому остаётся на самостоятельное изучение.

> **Важно** Токен a9aae70d65c928a554f9a038b9d4703a1583594f был использован только в демонстрационных целях и больше не работает. Прямое указание токенов в коде программы недопустимо и сделано здесь мной только в интересах упрощения примеров.

Полезные ссылки

- Доклад Роя Филдинга: Representational State Transfer (REST)
- Three Principles of API First Design
- HTTP-методы
- REST Architectural Constraints
- Сайт Swagger
- Пример использования Swagger
- Wiki про OpenAPI
- Критика REST

Спасибы

- Андрею Панфилову за вычитку и правки
- Александру Фатину за вычитку и правки

4.6 Заключение

В этой статье я не преследовал цель рассмотреть все возможности NetBox, поэтому всё остальное отдаю вам на откуп. Разбирайтесь, пробуйте.

Далее в рамках построения системы автоматизации я буду касаться только тех частей, которые нам действительно нужны.

Итак, выше я коротко рассказал о том, что из себя представляет NetBox, и как в нём хранятся данные. Повторюсь, что почти все необходимые данные я туда уже внёс, и вы можете утащить себе дамп БД

Всё готово к следующему этапу автоматизации: написанию системы (ахаха, просто скриптов) инициализации устройств и управления конфигурацией.

4.6.1 Полезные ссылки

- Сам NetBox на github
- Контейнерная версия
- Полная инструкция по установке и вся документация по продукту
- Documenting your network infrastructure in NetBox, integrating with Ansible over REST API
- IPAM NetBox and its API, Docker, Postman
- IPAM NetBox and its API, configuration templates with Python
- SDK для работы с NetBox в Python
- Документация по API
- Mailing list
- Канал в Slack NetworkToCode
- Что такое REST
- Инсталляция NetBox для нужд АДСМ

Глава 5

Часть 4. Архитектура системы автоматизации

Продолжаем наш забег по сетевой автоматизации.

Итак, сеть спроектирована, IPAM запущен. И вот-вот начнут съезжаться миллионы наших стоек. Будем готовиться к этому.

Мы всё дальше от фантазий и абстрактных разговоров и ближе к практике.

И всё же снова сделаем отступление. Большое дело начинается с большого перекура.

Сеть полезно представлять, как некое единое целое, которое мы переводим из одного состояния в другое. Сервис мы внедряем на всей сети. Не может быть такого, что он работает только на 3 устройствах из 4 необходимых. **Вся** сеть должна обеспечивать отказоустойчивость и достаточную полосу.

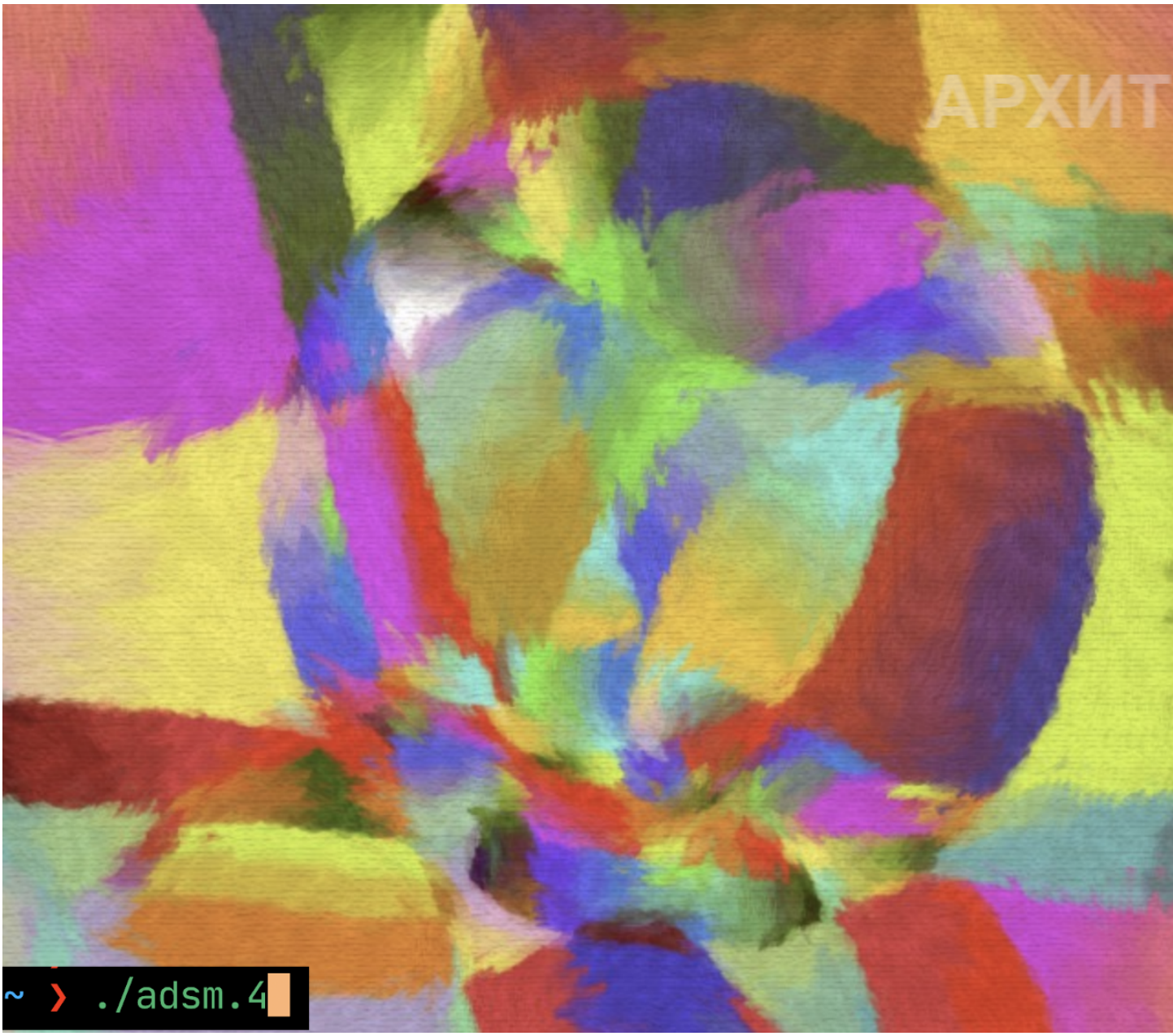
Однако рано или поздно всё равно любая задача декомпозируется до уровня отдельных сетевых коробок.

И если про сеть как единый организм мы уже поговорили в 0-й статье, то теперь пришло время разобраться отдельными органами.

В этой статье разберём жизненный цикл сетевого устройства и некоторые сценарии того, какие манипуляции с ним приходится порой делать.

Естественно, всё это интересует нас с точки зрения автоматизируемости. Поэтому ещё мы нарисуем архитектуру системы автоматизации.

Кстати, не так давно вышла просто восхитительная обзорная статья Дмитрия Тесля о процессе и инструментах сетевой автоматизации. Он смог лаконично изложить то, вокруг чего я пляшу уже несколько выпусков АДСМ. Настоятельно рекомендую прочитать её перед тем, как преступать к этой.


АРХИТ
~ ❯ ./adsm.4

5.1 Общий взгляд на жизненный цикл оборудования

Вот мы купили сетевую железку. Что теперь? Проследим её жизнь с первого и до последнего дня.

1. **Day 0** - железка только появилась в наших руках. Сейчас самое важное - базовая настройка:
   - Добавить IP-адрес управления и маршрут
   - Включить SSH
   - Создать пользователя с правами настройки

   Иными словами задача Day0 конфигурации - организовать доступ на устройство.

2. **Day 1** - Железка уже встала на позицию, определена его роль в сети и сервисы, которые она обслуживает.
   Теперь нужно настроить уже целевую конфигурацию, с которой устройство встанет в сеть под нагрузку.
3. **Day N** - Изменения конфигурации в процессе эксплуатации. | Бывает мы добавляем новый сервис, пересматриваем дизайн или на худой конец нужно ACL поправить. | Такие изменения нужно тоже уметь довозить до устройства.
4. **Обслуживание** - Помимо нормальной работы есть периоды, когда устройство нужно аккуратно вывести из под нагрузки, чтобы, например, поменять в нём плату, провести обновление ПО.
5. **Отслеживание изменений** - со всей сети следует собирать информацию о том, где и во сколько применялась новая конфигурация. Это позволит как скоррелировать жалобы клиентов с изменениями, так и знать, когда применялась новая конфигурация в обход процедуры.
6. Проверка соответствия эталонной конфигурации** - В течение всей жизни устройства нужно проверять, что его конфигурация не разошлась с целевой из-за сбоев в автоматике, обновлений ПО или прямого вмешательства чьих-то рук.
7. **Бэкапы** - Даже если мы в любой момент можем сгенерировать эталонную конфигурацию, чтобы применить её на устройство, бэкапы необходимы.
8. **The Last Day** - снятие нагрузки, удаление из всех систем, ритуальное сжигание. Под сжиганием я понимаю безопасную затирку конфигурации, чтобы хэши паролей (или, упаси Лейбниц, клиртекст), префикс-листы и ACL'и не оказались достоянием общественности.

Я намеренно обхожу вниманием в этой статье вопрос мониторингов операционного состояния и реакции на них, поскольку её лейтмотив - это всё же конфигурация.

Далее обсудим Day0 - DayN более детально.

5.1.1 Day0

Итак, поставщик привёз на склад новый свитч. Его нужно установить, настроить, проверить, запустить трафик, добавить во все системы: инвентарные, мониторинги, бэкапы, скрипты автоматизации всякой рутины.

Задачи Day 0 можно грубо разделить на две части:

- Завести устройство в системах

Day 0
Day 1
Day N
Обслуживание
Проверки
Бэкапы
Налоги
Семья
Боль в спине
The Last D

- Настроить базовый доступ

Говорить про них в отрыве друг от друга сложно, и делать мы так не будем.

Какие же есть способы?

1. **Бумер** - вручную завести устройство в инвентарной системе и выделить свободный IP-адрес. Пусть это будет даже экселька.

   Подключить свитч к компьютеру и через консольный порт настроить IP-адрес, маршрут, включить SSH, создать пользователя.

   Отвезти свитч на позицию.

   \+ Просто, не требует почти никакой инфраструктуры.

   - Склонно к ошибками, масштабируется человеко-часами.

2. **Бумер+** - автоматизируем заведение устройства в DCIM/IPAM. Мы только нажимаем кнопочку, а в системе появляется железка на правильной локации со всеми нужными портами, ей выделяется автоматически имя и следующий свободный IP-адрес. В итоге генерируется базовый конфиг в виде текстового файлика.

   Администратор подключает свитч к компьютеру и через консольный порт копипастит содержимое этого файлика в терминал.

   \+ Ниже вероятность ошибок, значительно меньше ручной работы

   - Требуется уже какая-никакая инфраструктура: IPAM/DCIM с API, скрипт, всё ещё ручная работа, всё ещё настраивать на стенде и потом везти устройство на позицию.

3. **Миллениал** - ZTP - Zero Touch Provisioning - подход, которому 100 лет в обед, но он почему-то всё ещё есть не везде. Идея в том, что устройство сразу же ставится на позицию и подключается в сеть управления, после чего по DHCP оно само получает свою конфигурацию.

   Для этого устройство должно быть уже заведено в IPAM/DCIM и предгенерирована конфигурация, которая и передаётся устройству.

   \+ Устройство можно сразу везти на позицию, минимум ручного труда

   - Нужна уже продуманная связная инфраструктура: IPAM/DCIM, DHCP, (T)FTP, автогенерация конфигов. Классическую вендорскую реализацию сложно применить для распределённых сетей, вроде ритейла.

4. **Зумеры** - SD-WAN. Кстати, как раз подходит для ритейлов, хотя в свою очередь не очень для датацентров. Подход разделяет идею ZTP - мы устройство включаем, а оно само настраивается.

   \+ Меньше вероятность ошибок. На первый взгляд меньше работы

   - Однако SD-WAN - это преимущественно проприетарные решения вендоров, требующие мощной инфраструктуры, причём иногда только в облаке вендора. У нас, кстати, был целый подкаст про SD-WAN: telecom №91. SD-WAN.

5. **Пост-хипстеры** - есть компании, где помимо Out of Band сети управления, есть ещё консольное соединение до абсолютно каждой железки. Для этого есть соответственно сеть консольных серверов внутри датацентров и точек присутствия.

   Каждое новое устройство после установки подключается отдельно в OOB-свитч по Ethernet и в консольный сервер консольным линком.

   Это позволяет реализовать схему, подобную описанной ниже:

   - Устройство добавляется в IPAM/DCIM
   - Устройство устанавливается и подключается по управлению
   - Инженер в ДЦ создаёт задачу на сервер наливки: настроить свитч за консольными сервером №7, порт 3

- Сервер наливки подключается на указанный порт, забирает серийный номер, с которым идёт в IPAM, генерирует базовый конфиг и обратно через тот же консольный порт применяет данную конфигурацию

+ Всегда есть консольный доступ на устройство, какие бы шторма ни гуляли в сети трафика и управления. Нет проблем с вендорскими особенностями - консольный протокол у всех реализован одинаково (с поправкой на параметры порта)
- Совсем непросто и в абсолютных цифрах недёшево реализовывать ещё одну сеть управления. Не подходит для географически распределённых сетей. Требуется серьёзная инфраструктура даже в минимальном варианте без использования сервера наливки.

Как видите, любые решения по автоматизации Day 0 требуют чего-то больше, чем просто скриптик на питоне. К этому процессу нужно подходить системно с точки зрения выстраивания инфраструктуры.
Кстати, вот классный доклад от фейсбука про их Вендинговые Машины по выдаче новых локаций: Scaling the Facebook backbone through Zero Touch Provisioning (ZTP)

Так или иначе эта часть автоматизирована у многих, потому что подходы понятны, инструменты в ассортименте.

5.1.2 Day 1

Дальше на железку нужно накатить уже рабочую конфигурацию и пустить на неё нагрузку.
Тут уже заметно интереснее. Одно дело - сгенерировать простейший конфиг на 20 строчек, одинаковый для всех типов устройств, как было в Day 0, и совсем другое - целевой конфиг на пару тысяч строк, который может радикально отличаться от железки к железке в зависимости от её роли и необходимых сервисов. Например, конфигурации двух экземпляров одной и той же модели свитча, установленных в качестве лифа и спайна, будут различаться как минимум настройками даунлинк интерфейсов.
Основная идея здесь в том, что мы описываем дизайн сети в том или ином формальном виде и отдаём его генераторам. Генераторы берут этот дизайн, роль устройства, локацию, переменные из IPAM/DCIM, всё это перемешивают, а на выходе получается специфический для данной коробки конфиг.
То есть основных компонента здесь три:

- Формализованный дизайн
- Заполненные данные в IPAM/DCIM
- Набор генераторов

Здесь подробно останавливаться не будем - формализации дизайна я посвящу отдельный (и скорее всего не один) выпуск.

Итак, имеем конфиг Day1. Осталось всего ничего - применить его на железку.
И тут все средства хороши в разных комбинациях: консоль, SSH, netmiko, NETCONF, GNMI, REST API, SNMP (я сейчас не шучу - лично видел), FTP, SCP.

В целом на нерабочую пока железку применить конфиг действительно можно разными способами:

- Ручной копипаст из файлика в терминал
- Применение команд последовательно через SSH из кода, используя тот же netmiko
- Копирование файла на флэшку устройства, установка его в качестве конфигурационного и ребут железки
- А-ля config replace
- Пульнуть через NETCONF весь конфиг в XML
- gNMI

Об этом тоже ещё поговорим.

С автоматизацией этой задачи большинство тоже справляются - один раз настроить железку без нагрузки - дело нехитрое.

Замечу, что если есть процесс и инструменты Configuration Management и версионирования конфигурации, то Day1 - это лишь частный случай DayN.

5.1.3 Day N

И вот теперь - ежедневная эксплуатация и периодические реконфигурации.
А вот с этим дела обстоят туго чуть менее, чем у всех. Говоря это, я не шучу. Тут всё плохо.
Дело в том, что нагенерить конфигурацию - действительно несложно. Пусть это будет даже циклопический jinja-шаблон с циклами и каунтерами.

А вот применить этот конфиг на железку ещё и под продуктивной нагрузкой - цель для инженеров со стальными нервами.

Тут целый ком проблем, как очевидных, так и неявных.

Во-первых, интерфейс: CLI, NETCONF, GNMI, SCP/FTP.
Если CLI - то как быть с особенностями реализации каждого вендора? Режимы контекстов, интерактивные диалоги, порядок выполнения команд.
Если NETCONF или gNMI - то его не все вендоры поддерживают. А те, кто поддерживает, делают это сильно по-разному, и зачастую не в полной мере. А если в полной мере, то, конечно, же в своей схеме, а не в OpenConfig.
А если файлик подложить - то не все на лету умеют заменять, а значит с ребутом - только кому он нужен при добавлении BGP-пира?

Во-вторых, инструмент доставки: netmiko, ncclient, ansible (какой модуль), SaltStack?

В-третьих, как заливать вслепую? Отправляя полную конфигурацию, мы не знаем, как она изменит состояние устройства. Даже если мы видим дифф между файлами или в ветке в гите, это не говорит о том, какие команды фактически применятся на железке.

В-четвёртых, даже если мы видим будущие изменения (кандидат-конфиг на самом устройстве, к примеру), то это не говорит о том, что мы ничего не разломаем по своей неосмотрительности. Тут уже напрашивается сетевой CI/CD.

В-пятых, весь ворох вопросов мультивендорной взрослой сети: разный синтаксис, семантика даже между версиями софта, где-то есть коммиты, где-то нет, где-то можно увидеть кандидат, где-то нет.

Это область компромиссов.

Но давайте будем честны сами с собой: восьми компаниям из десяти не нужен выстроенный процесс версионирования конфигурации, конвейер CI/CD, автоматическая выкатка, а возможно, и вообще весь этот ваш DevOps в сети.

Скорее всего, вам действительно достаточно залить первичный конфиг, а дальше изменения накатывать всю жизнь элементарными плейбуками, составленными вручную. И для этого, включая мониторинги и внутренние инструменты, достаточно 2-5 человек, а не целый штат разработчиков.

И большинство компаний именно так и делает.

Можно добавить GitLab, TeamCity, AWX, аппаратную лабораторию с набором специфических тестов (FIB, QoS). Это всё мощные улучшайзеры, которые сделают процесс выкатки новой конфигурации значительно безопаснее. Но они не переведут управление конфигурацией на принципиально новый уровень.

А мы ведь всё же хотим

- Полную автоматизацию
- Универсальное решение
- Минимизацию рутины
- Безопасные выкатки конфигурации
- Формализованный дизайн
- Версионирование
- Транзакционность, а если быть точнее, то соответствие требованиям ACID

Поэтому давайте составим схему системы автоматизации, которая позволит нам решить все задачи. Но прежде расширим понятие «Инфраструктура как код» на сетевую инфраструктуру.

5.2 IaC

Если вы не слышали о **IaC - Infrastructure as Code**, у меня для вас плохие новости. Очень плохие. Ладно, не напрягайтесь, сейчас всё расскажу.

Как было раньше: выдали вам пачку физических машин - на каждой из них вы запустили KVM/VMWare/WTF, подняли на них флот виртуалочек, настроили сеть, выкатили своё приложение. Нужна дев-среда? Давайте всё в той же последовательности теми же руками. И во второй раз может получиться *чуть-чуть* не то же самое.

Парадигма IaC предполагает, что конфигурация всей инфраструктуры описывается в текстовых файлах. Речь как про физические устройства, так и про виртуальные машины, контейнеры и прочее.

Далее эти текстовые файлы обрабатываются неким инструментарием, который настраивает инфраструктуру на основе этой информации. Это может быть Terraform, Ansible, SaltStack.

Как результат - вы всегда быстро и с минимальным участием человека получаете предсказуемый результат.

Например, вы декларативно записываете в yml-, txt-, tf-, wtf-файле, что на таких-то хостах нужно установить KVM, Open vSwitch, настроить IP-адреса и туннели. Далее поднять набор ВМ с убунтой, выдать им адреса, на них установить nginx, загрузить ваш сайт в указанный каталог и настроить nginx. И поставить всё это дело за свежезапущенный балансер.

И получить всё это становится возможным всего одним запуском terrform, если желаемое состояние описано в конфигурационном файле.

Данные в самом файле могут быть в форматах yml, json, это может быть набор объектов вашего любимого ЯП или что угодно иное, что может быть принято инструментом.

Строго говоря, даже если вы в bash-скрипт напихаете весь набор операций - это уже будет IaC. Просто пользоваться этим не очень удобно.

На схеме ниже изображена упрощённая процедура того, как изменения появляются на сети в парадигме IaC.

https://dteslya.engineer/network_automaiton_101/

То есть настройка инфраструктуры выглядит аналогично релизу новой версии приложения в DevOps - те же гиты, апрувы, CI/CD и прочая.

Что же до сетей?

Прежде в хорошей ситуации у нас для них был HLD (High Level Design), который руками превращался в LLD (Low Level Design) для каждой железки, готовился конфиг (скорее всего, руками) и заливался на железо (надо полагать, тоже руками). В плохой - инженер сразу на железке настраивал сервисы так, как ему казалось правильным.

Систему автоматизации сети мы будем рассматривать также в разрезе IaC. Давайте уже прекратим строить из себя особенных, потому лишь что у наших хостов проприетарная ОС. Сеть - такая же часть инфраструктуры, как физические машины, виртуалки, контейнеры. Ну да, прихлопнуть свитч с нерабочим чипом и пересоздать новый нельзя - но это просто добавляет красок в нашу работу.

Обновление конфигурации на сети и обновление прочей инфраструктуры после этого - тот же процесс деплоя.

В общем это именно то, о чём мы тут толкуем с самого 0-го выпуска. Дизайн описан в формализованном HLD, а конкретные данные берутся из нашего SoT - Netbox. Из них генерится конфигурация и складывается в репозиторий, где прогоняются авто-тесты (в аппаратной или виртуальной лабе или что-то а-ля Batfish), кто-то смотрит глазами и подтверждает изменения, далее они по всем правилам CD выезжают в прод.

IaC - и ничегошеньки не настраиваем руками.

ВЫ НЕ ПОНИМАЕТЕ
ЭТО
ДРУГОЕ

Нет, спасибо
Мы слишком заняты

5.3 Система автоматизации с высоты птичьего полёта

Один из принципов, который нужно заложить в систему - это Zero Touch Prod, то есть свести к минимуму прямое хождение инженера на устройства. Любые изменения конфигурации только через платформу, только через интерфейс.

Итого, какой список задач решаем?

1. Нужен интерфейс, через который можно создавать задачи. Он абстрагирует работу с сетевыми устройствами.
   Графический - для инженеров, API - для внешних сервисов.
2. Ввод новых устройств
3. Актуализация данных в инвентарной системе (LLDP, список интерфейсов, IP-адресов, версия ПО)
4. Генерация целевых конфигураций
5. Применение целевых конфигураций и временных патчей (тшут, костыль)
6. Сличение целевых и реальных конфигов
7. Снятие и возврат нагрузки
8. Обновление ПО
9. Сбор бэкапов, коммитов,
10. Диспетчеризация задач, выполняющихся на железе.

Соответственно схематично я бы изобразил это так:

1. **IPAM/DCIM** - система, являющаяся Source of Truth для всей системы автоматизации. В нашем случае - Netbox.
2. **NetAPI** - служба одного окна. Что бы ни вздумалось сделать с сетью - идём в него.
   Например, захотелось добавить новый свитч - идём в ручку NetAPI с нужным набором параметров (серийник, имя, локация) и создаём задачу на добавление свитча. А-ля: https://netapi.linkmeup.ru/api/adddevice.
   Захотелось собрать LLDP с устройств - идём в другую ручку со списком устройств. А-ля: https://netapi.linkmeup.ru/api/lldp.

   Исключительно как вариант: это может быть приложение на Django, FastAPI, Flask, запущенное как systemd-сервис.
3. **Набор приложений**, которые реализуют функционал ручек NetAPI. Например, клиент хочет получить список MAC-адресов со свитча - он идёт в ручку, а ручка дёргает модуль сбора MAC'ов, модуль идёт на свитч по SSH и собирает необходимую информацию (через CLI или NETCONF).
   Это может быть как интегральная часть NetAPI, так и отдельные сервисы, с которыми NetAPI взаимодействует по ещё одному API (REST, GRPC).

4. **Сервис NetGet**, выполняющий регулярные и разовые задачи на сбор данных с сетевых устройств, таких как бэкапы, коммиты, версии ПО итд.

   Это может быть systemd-сервис или просто набор скриптов, запускающихся по cron'у или триггеру.

5. **ConfMan - Configuration Manager** - это набор сервис и компонентов, выполняющий всю работу по управлению конфигурацией.

   Его составными частями являются:

   - HLD - формализованный дизайн сети (High Level Design). Это могут быть объекты того языка программирования, на котором написана система автоматизации, может быть набор YAML-файлов или что-то своё собственное.
   - Хранилище переменных, необходимых для конфигурации, которые по тем или иным причинам не получается хранить в IPAM/DCIM (например, префикс-листы или syslog-сервера).
   - Специфические компоненты, такие как система управления доступами - для ACL. Или система планирования нагрузки для генерирования конфигов QoS-очередей. Возможно, оркестраторы/контроллеры для инжиниринга трафика, тоже стоит рассматривать как часть ConfMan.
   - Набор генераторов конфигурации - то самое, что возьмёт HLD, обогатит его данными из IPAM/DCIM, хранилища, других систем и сформирует конечный вид конфигурации устройства.
   - Возможно, часть, которая вычисляет фактическую дельту конфига и формирует патч, то есть список команд для достижения целевого состояния. *Возможно* - потому что вместо применения только изменений, можно целиком конфигурацию заменять.
   - Модуль, отвечающий за сличение целевого и реального конфига.

   Отдельные компоненты ConfMan взаимодействуют друг с другом через тот или иной API.

6. **Carrier** - доставщик изменений на сеть. Например, ConfMan сгенерировал пачку конфигов и передал Carrier'у на применение.

   В зависимости от используемого интерфейса взаимодействия с сетевым устройством он выполняет разные функции.

   Так, для CLI он знает специфику взаимодействия с консолью конкретного вендора - интерактивные ответы, ошибки, информационные сообщения.

   Для NETCONF'а он умеет определять успешность или неуспешность применения конфигурации.

   > Можно было бы назвать его worker'ом, но Carrier - это функциональный компонент, тогда как Worker - это его экземпляр. То есть может быть несколько worker'ов, выполняющих задачу Carrier, настраивая одновременно две разные железки.

7. Над всем этим царит **Dispatcher** - этакий диспетчер задач, бригадир, который распределяет работу.

   Он ведёт учёт всех поступивших задач, отслеживает их статусы, составляет расписание на исполнение.

   Например, если стоит задача обновить 300 свитчей, то он знает, что нельзя это делать одновременно, поэтому он составит расписание. Так же он не выведет из эксплуатации больше двух спайнов одновременно, и не проведёт работы на двух бордерах.

   Если на конкретную железку уже есть задача или на ней CPU под сотку, это значит, что применение изменений нужно отложить.

В общем вот таким составлением расписания и занимается Dispatcher.
Все задачи связанные с доступом на сетевое устройство, проходят через него.

Вот такая получается система. Не очень простая, но не очень и сложная.

Давайте сразу отметим несколько важных характеристик этой системы.

5.3.1 Характеристики системы

Единый интерфейс

Во-первых, отметим здесь центральную роль **NetAPI**. Он является точкой входа для большинства задач: ввести новое железо, переконфигурить старое, обновить свитч. Внутри задачи могут быть подзадачи, требующие обращение к NetAPI, например, обновление ПО своей подзадачей имеет снятие нагрузки, которое тоже может являться ручкой NetAPI, а снятие нагрузки в свою очередь требует проверки наличия трафика на портах, что тоже подразумевает поход в NetAPI. И так далее.

Асинхронность

Во-вторых, нам необходим асинхронный режим работы API. Некоторые из запросов (тот же ввод нового оборудования в работу) может длиться продолжительное время, то есть ответ клиенту не вернётся в обозримое время. Поэтому нужна возможность создать заявку, получить её ID и вернуться позже за уточнением её статуса.
Для этого каждому запросу в API выделяется ID, данные о нём вносятся в базу данных, статус обновляется по мере поступления новых данных.

Соответственно должна существовать отдельная ручка (-и), в которую (-ые) можно прийти и узнать статус запроса по ID.

ACID

В-третьих, применение конфигурации на сеть должно соответствовать принципам ACID.

Давайте рассматривать выкатку новой конфигурации на сеть как транзакцию.

- **A - Atomicity**. Никакая конфигурация не должна примениться частично. Как в пределах устройства, так и в периметре сервиса - на наборе устройств. Применяется либо вся конфигурация, либо никакая. Соответственно, если на ряде устройств конфигурация применилась, она должна быть откачена. Либо средствами встроенного rollback-механизма, либо набором отменяющих изменения команд.
- **C - Consistency**. Именно в том виде, как понятие консистентность применяется к БД, к сети, пожалуй, не применима, но мы будем иметь в виду, что все сетевые сервисы после применения новой конфигурации остаются работоспособными.
  Факт консистенстности проверяется набором тестов, запускающимся после выкатки конфигурации. В зависимости от типа изменений могут быть разные наборы тестов. Иногда достаточно проверить CPU на паре коробок, в другой раз запустить пинги и проверить статусы BGP-сессий, а в третьем - всесторонние тесты всего, что настроено на сети.

- **I - Isolation**. Вполне понятный принцип применительно к сети - с того момента, как мы запланировали выкатку новой версии и до её применения, статус сети должен быть зафиксирован - никто не должен её менять. И уж тем более никто не должен настраивать что-то одновременно с запланированной выкаткой.

  Но это качество проще обозначить, чем обеспечить. Допустим, все таски внутри системы управляются Диспетчером, и он выстроит все задачи в правильном порядке. Однако как быть с тем, что кто-то может руками наадхочить на железке? Есть только один способ с этим справиться - **люди не ходят на оборудование напрямую** - Zero Touch Prod, помним. То есть на железе остаётся служебная учётка нашей системы автоматизации и аварийная для инженеров, которую используют только в ситуациях, когда система сложилась и надо срочно попасть на железо.

  Увы, это не отвечает на два вопроса: «А для тшута мы что делаем?» и «Что мешает инженеру пользоваться аварийной учёткой?». Вообще-то и на тот и на другой вопрос можно подобрать ответы, но не будем тут зацикливаться.

- **D - Durability**. Ну тут всё просто - что бы ни случилось на сети, после восстановления конфигурация должна быть прежней. Решается это сохранением конфигурации при каждом коммите (или изменении конфиги, если коммита нет). Но есть нюанс - идентичная конфигурация не говорит об идентичном поведении - дело может быть в консистентности FIB. Но это тоже уже за рамками данной статьи.

Взаимодействие компонент через API

В-четвёртых, взаимодействие между элементами системы. Очевидно на схеме выше лишь упрощённая схема. Фактически она будет значительно больше, а количество связей и сообщений между элементами превысит все мыслимые и немыслимые значения, а Васюки станут центром десяти губерний!.

К чему это я? Взаимодействие между частями системы должно быть реализовано через API, каким бы он ни был - gRPC, HTTP REST, да хоть SOAP (нет, не хоть).

А кроме того, в какой-то момент нам может понадобиться **очередь сообщений** (**Message Queue**). Мы всё это ещё потом в контейнеры сложим. И наступит полный микросервис.

Однако, сосредоточимся на важнейшей задаче - снизить нагрузку на инженера, а для этого надо исключить хождение инженеров на железо напрямую - нужно теперь разобрать сценарии, когда это требуется в обычной жизни.

5.4 Сценарии

В реальной жизни их, конечно, будет много. Я же опишу самые необходимые:

- *0. Проверка сети*
- *1. Ввод нового оборудования*
- *2. Переконфигурация из-за изменений переменных в инвентарной системе*
- *3. Переконфигурация из-за изменения дизайна*
- *4. Сбор информации с устройств*
- *5. Снятие и возврат нагрузки*
- *6. Обновление ПО*

- *7. Удаление устройства*
- *8. Замена устройства*

Распишем каждый из них детально.

5.4.1 0. Проверка сети

Наверно, совершенно оправданным будет нулевой приоритет отдать именно блоку ручек для проверки сети, поскольку к ним нужно будет обращаться почти в каждом последующем сценарии.
Часть из них будут реализовывать blackbox-проверки. Например, наличие e2e-связности, потерь, RTT.
Другая - whitebox - существование тех или иных маршрутов в RIB, состояние FIB итд.

1. Человек или сервис приходит в ручку NetAPI с запросом, в теле которого указаны параметры теста. Например, ICMP, устройство-источник, адрес назначения, VRF, число проб, размер пакета.
2. NetAPI формирует запрос в NetGet, чтобы тот собрал данные с сети/устройства. И тот собирает.
3. Результаты теста возвращаются клиенту.

5.4.2 1. Ввод нового оборудования

В зависимости от скорости роста, возможно, самый важный сценарий - быстро запускать новые узлы (стойки, филиалы, офисы), поскольку обычно занимает больше всего времени.

1. Человек или часть системы, реализующей нечто а-ля ZTP, приходит в NetAPI для инициализации устройства.
   Устройство идентифицируется по своему серийнику или инвентарному номеру, и ему должна быть задана роль, чтобы было понятно, с какой конфигурацией его наливать.

   1. Ручка дёргает конкретное приложение, отвечающее за этот шаг
   2. Приложение создаёт устройство в NetBox и прописывает его
      - Имя
      - Серийник
      - Локацию
      - Вендор/модель
      - Роль в сети
      - Присущие ему свойства: список интерфейсов, консольных портов, комментарии.
   3. Приложение определяет и при необходимости создаёт MGMT-интерфейс
   4. Приложение выделяет MGMT IP.

   На данном шаге устройство в минимальном виде заведено в инвентарной системе, и заполнены необходимые для первичной настройки параметры.

2. Далее другая часть процесса, а-ля ZTP, приходит в ручку NetAPI в поисках первоначального конфига

    1. Ручка дёргает конкретное приложение
    2. Приложение собирает данные из NetBox и, возможно, внешних систем
    3. Приложение рендерит конфиг, возвращает его клиенту и заодно складывает его в git-репозиторий.
    4. Клиент каким-то образом доставляет конфигурацию до устройства - это может быть ZTP или пропихивание конфига через консольный порт. Идентификатором устройства тут выступает серийник.

    После этого шага появляется удалённый SSH-доступ на устройство.

    Теперь по какому-то триггеру запускается конвейер ввода устройства в эксплуатацию.
    Триггером может быть:

    - Чьё-то ручное действие - например, нажатие кнопки в интерфейсе - и сигнал в NetAPI.
    - Обращение к ручке ввода в NetAPI от системы ZTP после завершения.
    - Факт появления доступа по SSH на устройство - например, кроняка пытается доступиться до железки, которая помечена как «для ввода».

3. Заполняются данные в NetBox, которые в дальнейшем будут служить переменными для генерации конфигурации.

    1. Система посылает в NetGet запрос на сбор данных о LLDP с данного свитча.
        1. Информация о соседях вносится в NetBox, порты связываются друг с другом.
        2. При необходимости создаются сабинтерфейсы или интерфейсы добавляются в LAG.
        3. Вычисляются (или выделяются) P2P IP-адреса.
           Необходимые изменения выполняются и на соседнем устройстве.
           Этот шаг позволяет, во-первых, подготовить данные для настройки IP-адресов, во-вторых, визуализировать топологию при необходимости, в-третьих, собрать в будущем информацию о BGP-соседях, если на узле используется BGP.
        4. Система создаёт набор виртуальных интерфейсов и выделяет IP-адреса. Например, loopback'и и VLAN-интерфейсы.
        5. Заполняет другие необходимые данные. Например, ASN, IS-IS Network Entity, настройки l2-интерфейсов.

4. Обновление данных в NetBox инициирует запрос в NetAPI на запуск конвейера для вычисления и деплоя новой конфигурации. Это может быть, например, Web-hook, отправленный самим Netbox'ом.

Речь здесь идёт обо всех устройствах, конфигурация которых меняется в результате ввода новых устройств. Добавляется новый Leaf - поменяется конфигурация Spine.

5. NetAPI через Диспетчера адресует задачу на ConfMan, который вычисляет вендор-агностик конфигурацию.

   Для этого система берёт формализованную модель конфигурации данных (питоновские объекты, yaml итд) и подставляет в неё данные из NetBox.

   Результатом может быть словарь, тот же yaml или питоновский объект.

6. Система генерит конфиг для списка устройств. Результатом может быть текст, содержащий последовательность CLI-команд, NETCONF XML, набор объектов для YANG, Protobuf для gNMI.

7. Выполняются лабораторные тесты CI/CD. Они могут быть в симуляторе, вроде Batfish, виртуальном стенде или всамделишной небольшой железной лабе, мимикрирующей под настоящую сеть.

   Даже для типовой операции, вроде описываемого ввода новых, серверов разумно их делать, ведь данные в SoT изменились - и выкатка может разломать сеть.

   Проходят ручные проверки и подтверждения.

   > Это немного сколькзий момент. С одной стороны я всё же не верю, что в обозримом будущем на сеть новый конфиг можно катить без человеческого подтверждения, как это давно происходит в мире WEB-приложений.
   >
   > С другой - когда изменения катятся на тысячу устройств, пойди глазами всё просмотри. Поэтому всё же CI/CD и канареечные деплои - это то, к чему мы будем стремиться.

   Опционально этот шаг может выполняться в git-репозитории. Хотя заставлять человека переходить во внешний относительно основной системы автоматизации сервис - негуманно. Впрочем как первые шаги разработки такой системы - вполне нормально.

   > Я всё же не верю, что в обозримом будущем на сеть новый конфиг можно катить без человеческого подтверждения, как это давно происходит в мире WEB-приложений.

8. По факту сгенерированного конфига или полученных апрувов формируется задача в Dispatcher для Carrier'а на доставку и применение конфигурации на сеть.

9. Диспетчер диспетчеризирует и следит за выполнением каждой конкретной задачи и всей транзакции целиком.

   Он несёт полную ответственность за то, когда выполняется задача и с каким статусом она завершается.

10. В случае успешной транзакции Диспетчер обращается в ручки NetAPI, чтобы провести ряд тестов, проверяющих две вещи:

    - Новое устройство готово к обслуживанию трафика,
    - Сеть при этом не сломалась.

Запускаются какие-то пинги. Проверяется маршрутная информация на сети - сравнивается с бейзлайном (например, состояние, как было до деплоя). Последнее предполагает, что мы либо собрали состояние перед обновлением, либо есть некая база данных с временными рядами (TSDB - Time Series Data Base), содержащая срезы исторических данных.

Есть тесты, падение которых вызовет аварию, но операция будет считаться завершённой. А есть те, после которых произойдёт автоматический откат всей транзакции. Лучше не сделать ничего, чем сделать хорошо, но наполовину.

11. В случае успешных тестов в NetBox и/или иных системах проставляются индикаторы успешного ввода, новое устройство заводится в мониторинги и другие системы.
12. С результатами Диспетчер идёт в ручку NetAPI и сообщает, что ввод завершён успешно, либо нет.

Конвейер завершён.

Это весьма упрощённый конвейер, конечно. Здесь опущены шаги, которые могут быть фактически необходимы в реальной жизни: подавления аварийных сообщений, отписывание комментариев в тикеты, возможные проверки и подтверждения целевой конфигурации живыми людьми, всевозможные валидации на каждом шаге.

5.4.3 2. Переконфигурация из-за изменений переменных в инвентарной системе

Допустим по какой-то причине данные в нашем SoT поменялись - человек руками дескрипшон на порту изменил или автоматика пересчитала LLDP-соседства или ещё что-то.

Это изменение, которое должно привести к запуску конвейера по вычислению и выкатке новой конфигурации, описанного выше.

Триггером может быть Web-hook от SoT или опять же кроняка, которая следит за изменениями в этом SoT.

NetAPI получает запрос на запуск конвейера для вычисления и деплоя новой конфигурации, как это уже было в предыдущем сценарии.

Далее повторяются все те же действия, за исключением специфики, присущей вводу новых стоек. Все те же тесты.

Не забываем про версионирование - изменения переменных в SoT фактически ведёт к изменению версии конфигурации сети. Мажорное, минорное или патч - это предмет жарких дискуссий, судьёй которому будет semver.

5.4.4 3. Переконфигурация из-за изменения дизайна

Это может быть как небольшое изменение политики маршрутизации или ACL, так и сравнительно масштабная вещь, такая как добавление нового типа сервиса на всю сеть.

В целом, что относить к дизайну, а что к переменным - вопрос не просто дискуссионный, думаю, он на данный момент не имеет точного ответа.

Так же вопрос без ответа, в каком виде дизайн должен храниться - питоновские объекты, словарь, yaml, json? Хотел бы знать.

Но допустим, что независимо от формы он хранится в гите. И тогда его изменение легко можно использовать как триггер для запуска конвейера для вычисления и деплоя новой конфигурации, который мы дважды уже тронули выше.

> Впрочем, тут возможна специфика: изменения дизайна несут риски, поэтому неплохо бы добавить шаг проведения тестов в лабе с помощью CI/CD.

С точки зрения версионирования - инженер, меняющий дизайн и коммитящий изменения в гит, сам определяет насколько это важное обновление.

5.4.5 4. Сбор информации с устройств

В целом сбором информации занимается NetGet. Как периодическим, так и разовым по запросу.

Поэтому, когда нужно собрать, например, MAC'и с конкретного устройства, клиент идёт в ручку NetAPI, а тот в свою очередь дёргает NetGet.

NetGet формирует задачу для Диспетчера, чтобы Carrier сходил на устройство и собрал необходимую информацию.

Учитывая, что для таких запросов клиент ожидает синхронный режим, Диспетчер должен по возможности прогнать его с высоким приоритетом и быстро вернуть ответ NetGet'у.

Из любопытных идей для оптимизации: NetGet видится очень активноиспользуемым компонентом - вплоть до того, что мониторинг будет ходить в него, чтобы собрать счётчики и состояние сети - и, возможно, ему стоило бы держать открытыми и прогретыми сессии со всем флотом сетевых устройств. С использованием asyncio данные будут собираться просто в мгновение ока. А шардирование сетевых элементов по разным worker'ам позволит не упираться в лимиты.

5.4.6 5. Снятие и возврат нагрузки

Этот сценарий не является самостоятельным, если мы говорим про окончательное решение вопроса автоматизации - это, скорее, ручка, к которой мы будем обращаться из других сценариев.

С одной стороны это задача, требующая ультра-много операций, занимающая много времени и склонная к человеческим ошибкам. Допустим какой-нибудь бордер вывести из эксплуатации, для замены контрол-бордов. Явно нужно автоматически это делать.

С другой - зачастую это работа, требующая весьма интеллектуальной деятельности - поди разбери в нужном порядке разные сервисы, линки, клиентов.

Но для сравнительно простых устройств, каковыми являются торы, спайны и суперспайны или один из маршрутизаторов в ISP на резервированном канале, сделать это выглядит несложным.

Это может быть реализовано как две ручки: для снятия нагрузки и для возврата - так и как одна: выполняющая полный цикл.

1. Клиент приходит в ручку NetAPI. А тот запускает конвейер увода нагрузки
2. Приложение определяет список сервисов, которые нужно погасить (L2/L3VPN, базовая маршрутизация, MPLS итд)
3. Приложение формирует список действий, которые нужно совершить.
   Например:
   1. Плавно увести трафик с помощью BGP gshut community или ISIS overload bit (или ещё чего-то
   2. Убедиться в отсутствии трафика на интерфейсах
   3. Выключить BGP-сессии в нужном порядке (сначала сервисные, потом транспортные
   4. Выключить интерфейсы
   5. Убедиться в отсутствии активных аварий по сервисам
4. Зафиксировать статус задачи.

Клиент может начинать выполнять запланированные работы. Клиентом может быть другой конвейер.

По завершении клиент дёргает ту же ручку для возврата нагрузки - и тогда в обратном порядке выполняются предыдущие действия.
Либо же это отдельная ручка, которая независимо описывает, каким образом для данного типа узлов происходит возврат нагрузки.

5.4.7 6. Обновление ПО

Обновление может быть двух видов - требующее прерывания сервисов, и нет.

Соответственно конвейеры для них будут разные.
Рассмотрим для сложного случая

1. Клиент приходит в ручку NetAPI
2. Запускается конвейер снятия нагрузки
3. Запускается конвейер обновления ПО:
   1. Залить файлы ПО
   2. Проверить контрольную сумму
   3. Обновить прошивку, указать загрузочные файлы, перезагрузить устройство и провести иные мероприятия
   4. После обновления проверить версию ПО
4. Запустить конвейер возврата нагрузки.

5.4.8 7. Удаление устройства

Это весьма частый сценарий. Особенно если рассматривать переезд старого устройства в новую роль или локацию, как удаление и создание нового.

1. Клиент приходит в NetAPI. Тот дёргает приложение, отвечающее за удаление устройства.
2. Приложение проверяет, что нагрузка на устройстве ниже определённого порога.
3. Приложение обращается в NetAPI в ручку снятия нагрузки.
4. Приложение ищет все зависящие от этого устройства объекты в SoT. Как пример:
    1. Интерфейсы
    2. IP-адреса
    3. Подсети
    4. Интерфейсы соседних устройств
    5. P2P-адреса соседних устройств
    6. Итд.
5. Приложение удаляет их все.
6. Изменения в SoT триггерят запуск уже известного нам конвейера. Как вы видите он весьма и весьма универсален.
   Как результат - настройки соседних устройств, относящиеся к удаляемому, так же удаляются в процессе деплоя новой конфигурации.

   Само же устройство затирается к заводским настройкам. Кроме того оно удаляется из всех мониторингов и других систем.

7. Устройство удаляется из БД или помечается каким-то образом, если нужно сохранить о нём информацию.

5.4.9 8. Замена устройства

Случается, что свитч ломается. Или нужно железку проапгрейдить на новую модель. В общем надо её снять, а новую поставить.
Теоретически это выглядит как два шага:

1. Удаление текущего устройства
2. Добавление нового

Но нам важны несколько вещей:

- Имя нового устройства должно быть таким же, как и у прежнего
- Сохранить MGMT IP
- Сохранить и другие атрибуты: лупбэки, вланы, ASN, итд
- Скорее всего, и конфигурацию

Не факт, что это всё необходимо, но, скорее всего, так.

Самым простым выглядит в существующей записи поменять минимум вещей - инвентарник, серийник, модель. Но это лишает гибкости и добавляет несколько щекотливых моментов при выводе старой железки.
Кроме того, мне импонирует мысль, что девайс в БД собой олицетворяет не место и роль, а вполне конкретное устройство. И при добавлении в сеть нового свитча или роутера, в DCIM появляется новая запись.

Поэтому я бы всё же рассматривал замену устройства на сети как

- Удаление старого устройства
- Добавление нового с определённым набором атрибутов, значение которых хотим зафиксировать, и которые в противном случае определялись/выделялись бы автоматически.

При этом процедура удаления, определённая шагом выше, берётся как есть: с удалением артефактов на других устройствах (пересоздадим на втором шаге) и вычисткой конфига с устройства (чтобы, например, оно случайно на новом месте неожиданно не запустилось со старыми адресами и не начало всасывать и блэкхолить трафик).
Естественно, сценарии этим не ограничиваются. Их количество, степень автоматизированности и результаты диктуются бизнес-логикой и рациональностью.

Опять же мы тут опускаем вопросы подавления аварийных сообщений, коммита изменений в репы и подобные.

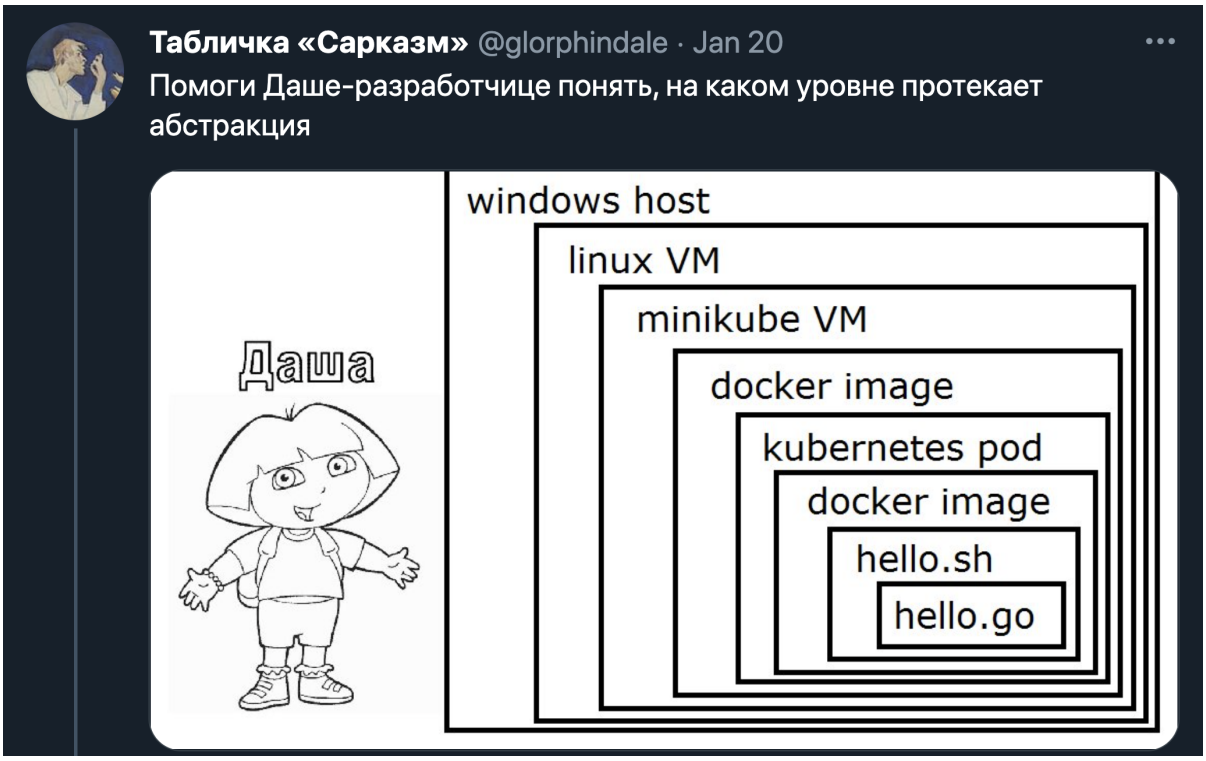
Но благодаря такому рассуждению мы приходим к пониманию, что здесь важно заложить наиболее общие и переиспользуемые конвейеры, которые станут впоследствии кирпичиками более сложных задач.
Сами конвейеры при этом декомпозируются на ещё более простые и универсальные атомы.

5.5 Полезные ссылки

- Awesome Network Automation
- Статья Network Automation 101 от Дмитрия Тесля
- Hands-on with NetDevOps
- Scaling the Facebook backbone through Zero Touch Provisioning (ZTP)

5.6 Заключение

Итак, скромной задачей этой статьи было спуститься на один уровень абстракции ниже по сравнению с нулевой публикацией и попытаться декомпозировать жизненный цикл оборудования на понятные блоки.



Повторюсь в очередной раз, что это лишь один из возможных подходов, который просто пришёл в голову мне. Он не только не претендует на оптимальность и проработанность, но и, хуже того, скорее всего, даже на таком описательном уровне уже местами переусложнён.

И ещё хуже то, что в рамках этой серии крайне маловероятно, что я сделаю что-то практическое, хотя бы отдалённое напоминающее описанную схему.
Хотя зарекаться не буду.

В следующих статьях мы поразбираемся с ZTP и какими-то базовыми скриптами автоматизации. А ещё рано или поздно углубимся в интерфейсы взаимодействия с сетевыми коробками: NETCONF/YANG, gNMI, GRPC.

5.6.1 Спасибы

- Роману Горге - бывшему ведущему подкаста linkmeup, а ныне эксперту в области облачных платформ - за комментарии про подход IaC и концепцию ACID применительно к сети.
- Михаилу Арефьеву - руководителю проектов по сетевой автоматизации в Яндексе - за анализ и критику архитектуры решения и сценариев.
- Дмитрию Фиголю - Network Automation Architect at Cisco Global Demo Engineering - за острые замечания и дискуссию.
- Никите Асташенко - моему другу и классному разработчику - за поездку на Алтай и долгие разговоры у костра, без которых эта идея не вызрела бы.

Особо благодарных просим задержаться и пройти на Патреон.

Become my patron on
patreon

Глава 6

Часть 5. История сетевой автоматизации

В этой части разбираемся, как мы оказались в том месте, где мы находимся, и куда ведёт нас этот путь.
Практическую пользу вам принесут только обе прочитанные части: 5 и 6. Вторая без первой будет непонятна. Первая без второй - беллетристика.

Когда началась история сетевой автоматизации?
С Ansible в 2018? С ним она явно получила ускорение благодаря безагентной природе.
Не. До него были голые языки на букву «P»: Python, Perl, PHP.

С NETCONF? Точно нет, CLI ещё мой дед парсил. А уж сколько expect'ов там поработало...

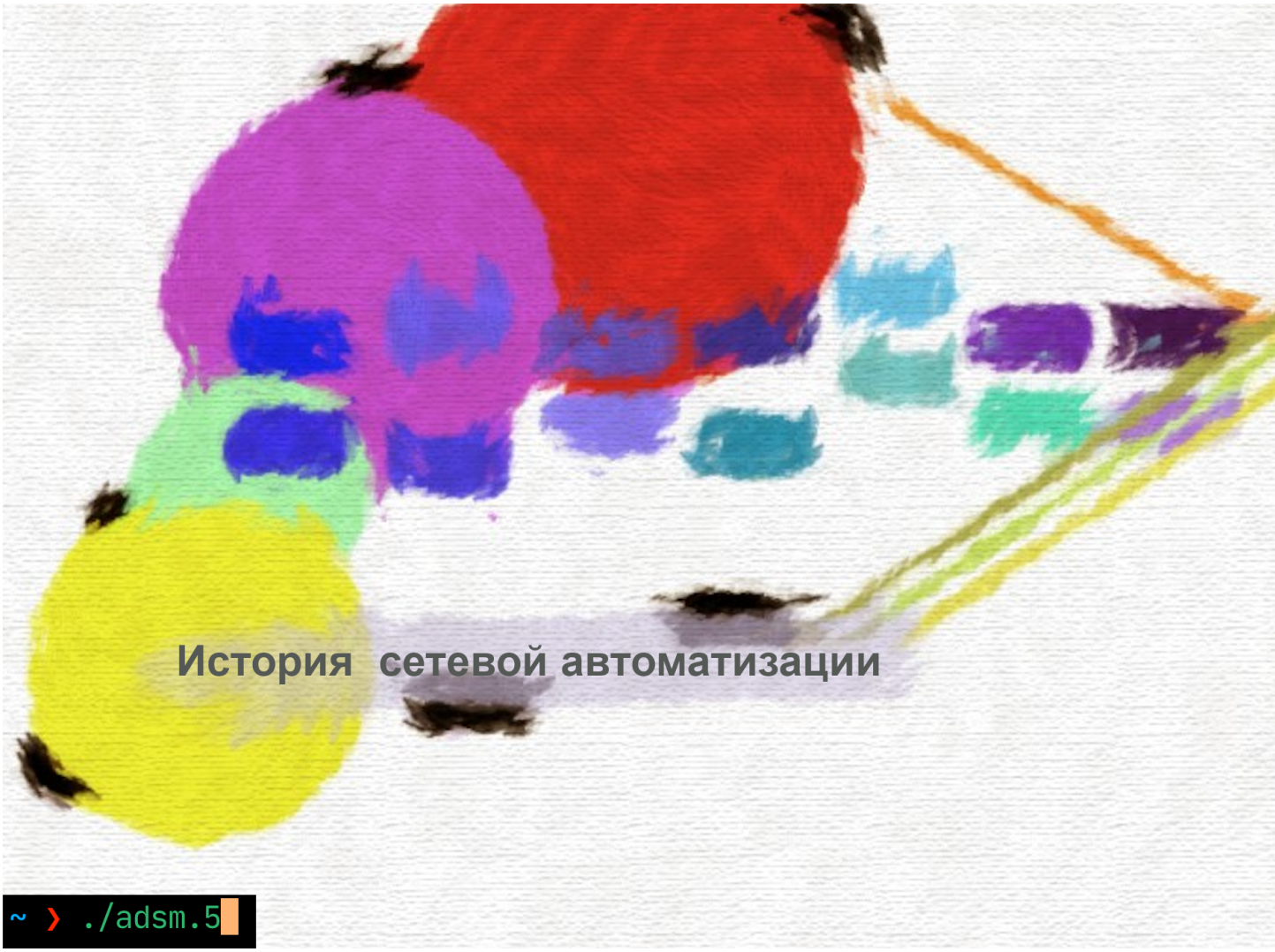
SNMP - вот что приходит на ум в качестве первого подхода - он родом из 90-х.

Однако как насчёт перехода от коммутации каналов к коммутации пакетов? Нельзя ли назвать динамическую сеть, не требующую мгновенного ручного вмешательства при обрывах, разновидностью автоматизации?

А первый декадно-шаговый искатель, разработанный Строуджером в 19-м веке и раз и навсегда избавивший мир от ручного труда телефонисток?

Да и в целом даже сам факт появления телефонных станций взамен почты, курьеров и гонцов?

Весь наш мир последние лет 300 безостановочно гонится за ускорением. Людей становится всё больше (Индия вон обогнала уже Китай), но их труд всё дороже (не в Индии). И в этом помогает автоматизация.
300 лет! Тем временем в сфере сетевых технологий сложилось мнение, что тут всё замерло, мы топчемся на месте, изобретая велосипеды. Но это лишь из-за того, что мы берём довольно короткий период времени и гораздо больший акцент делаем на текущем моменте, нежели на тенденции.


История сетевой автоматизации
~ ❯ ./adsm.5

Как говорится, мы склонны переоценивать краткосрочные последствия и недооценивать долгосрочные.

Источник: доклад на Cisco Live

В этой статье посмотрим, сколько всего в эти 30 лет уместилось. А уместилось немало.

6.1 One CLI to rule them all

С начала времён и до сего дня.

Сегодня бумажные деньги - это артефакты уходящей эпохи. Мы не можем в одночасье от них отказаться - даже в самых прогрессивных странах, потому что есть ещё огромный хвост людей, не готовых перейти на безналичный расчёт.

Но купюры были естественным и незаменимым средством ещё лет 10 назад.

Они пришли взамен монетам из золота и серебра, которые были естественны лет 300 назад. А те заменили собой натуральный обмен, который был естественным несколько тысяч лет назад.

Так и командная строка в своё время являлась совершенно естественным методом взаимодействия с сетевым железом. Вбивали же люди в nix'ах команды в терминал? Ну так и маршрутизатор - тот же nix.

Это самый понятный человеку-инженеру способ общаться с устройством.

Начал набирать команду, сделал табуляцию, посмотрел список доступных аргументов, продолжили. Возникла ошибка при выполнении - сразу же текст в терминале отображается - принимаешь меры.

Понятно, удобно, интерактивно. Нет ничего лучше для интерфейса человек-устройство.

Интерфейса, который сегодня стремительно теряет свои позиции в настройке сервисов.

И если в сфере серверов уже многие годы сумасшедшими темпами развивается RPC и IaC, то в области сетевых технологий всё как-то ни шатко ни валко.

На самом деле на сегодняшний день ситуация немногим менее печальная, чем лет 15 назад.

Командный интерфейс (**CLI**) реализует императивный интерфейс. А мы во всех сферах стремимся к декларативности.

CLI может быть структурированным и логичным, как на Juniper. Может быть как на Cisco/Huawei/Arista.

Он может поддерживать транзакции посредством коммитов, а может применять команды по мере их ввода.

Коммиты могут быть реализованы через подмену running конфигурации candidate, через применение только разницы между текущей и целевой или через последовательное применение введённых прежде команд.

Команды затем транслируются в вызовы какого-то внутреннего API, который может быть понятен и даже в каком-то смысле документирован (привет, Juniper) или представлять из себя чёрный ящик (йо, хуавэй, как жизнь?).

Но главная суть не меняется - если ты хочешь что-то настроить, ты подключаешься по ssh, вводишь команды - читаешь ответ, предпринимаешь дальнейшие действия.

То есть максимально понятно для человека и абсолютно неудобно для взаимодействия машина-машина.

Все эти приглашения, промпты, текстовые выводы, ошибки и предупреждения породили expect'ы, textfsm'ы, napalm'ы и триста способов заставить скрипт вести себя как человек.

Но самая главная проблема CLI - это фундаментальные различия в синтаксисе и семантике у различных вендоров. А порой даже в различных версиях.

Не существует такого способа, который позволил бы декларативно объявить - хочу вот именно такую конфигурацию BGP с такими пирами и таким набором политик, и не хочу вычислять, что мне нужно для этого добавить, а что удалить.

И кроме всего прочего долгое время не существовало надёжного инструмента доставки и применения конфигурации - каждый изгалялся в меру своей фантазии.

Спасибо Майклу Дехану, вендорам и сообществу, попытка создать такой инструмент была предпринята - Ansible. Да, к нему много (у кого-то очень много) вопросов. Да, он не решает большей части проблем, озвученных выше. Но по факту это лучший опенсорсный инструмент для применения конфигурации на сетевое железо (ну, кроме Nornir), и Ansible - спасение для многих сетевых инженеров. Сложить весь прод одним нажатием Enter никогда прежде не было так просто.

Ещё одна особенность CLI - отсутствие контроля состояния. CLI делает то, что ему сказали. Если ты не побеспокоился о том, чтобы удалить лишнего пира специальной командой - он и не почешется. Контроль состояния - ответственность инженера.

С другой стороны CLI - это на сегодняшний день (и долго ещё так будет) единственный вариант, который на 100% современных железок позволит настроить 100% предоставляемой ими функциональности.

Учитывая, что компаний, которые сейчас строят сеть с нуля и могут выбрать только то оборудование, которое поддерживает полноценный NETCONF/gNMI, очень и очень немного, всем остальным нужно уметь поддерживать зоопарк оборудования разного возраста и уровня.

Собственно в Яндексе именно тем, что CLI работает в буквальном смысле на всём, и воспользовались. Аннушка настраивает сетёвку именно через CLI.

Что можно утверждать однозначно - CLI как таковой никуда не денется **никогда**. Даже в эпоху полностью автоматизированных сетей, командная строка никуда не уйдёт из сферы диагностики.
Он, возможно, будет глубоко запрятан, или он трансформируется в полноценную линуксовую консоль, но как средство управления он останется.

Однако факт того, что все способы автоматизации чего-либо через CLI - это попытка написать скрипт, который будет прикидываться человеком и предугадать все возможные исключительные ситуации и варианты ответа операционной системы, заставляет писать очень изощрённые программы и постоянно их адаптировать под изменяющееся от версии к версии поведение и синтаксис новых вендоров.

А сколько радости представляет как крафтинг, так и парсинг текста, вы можете представить.

Желание унифицировать подходы и сделать стандартизированным интерфейс взаимодействия появилось не вчера. О нём думали уже в 80-е, что и породило очень удачное решение - SNMP.

6.2 SNMP - и не simple, и не management, и не short term

Бунтарские 80-е - лихие 90-е.

Это была достаточно смелая и, в некотором смысле, успешная попытка создать единый стандарт взаимодействия с сетевым устройством.
В 80-е годы было уже вполне понятно, что сети - бурнорастущая отрасль, в которой уже сейчас хаотически складываются подходы к управлению.

И вот умники в **IAB** (*Internet Activities Board*) крепко призадумались и 21-го марта 1988 года, собравшись в тогдашнем зуме (без шуток, не офлайн), постановили много важного стратегического про будущее систем управления интернетом. Ох они тогда напридумывали!

Результаты встречи они сели, записали и превратили в RFC1052 .

Они уже тогда действительно проектировали штуки, которые должны были не дать превратиться системам управления в то, во что они всё же превратились.

Как иронично теперь читать это послание из 80-х:

```
(i) Future Internet development is a joint interest of the R&D
community, the vendor community and the user community.
```

(continues on next page)

(продолжение с предыдущей страницы)

```
(ii) We still don't have a common understanding of what
[Inter]Network Management really is.
(iii) We will learn what [Inter]Network Management is by doing it.
(v) Define the Management Information Base for TCP/IP suite NOW!
(vi) Seek a seat for IETF on ANSI, ISO and/or CCITT

Удачи вам там, пацаны, в будущем...
```

Но работа закипела. RFC выходил за RFC. А количество рабочих групп не оставляло шансов для провала.

Что любопытно, так это то, что SNMP по их задумке был **временным протоколом**, решающим насущные нужды вендоров и операторов в перспективе нескольких лет. А в дальнейшем все должны были перейти на ISO CMIP/CMIS (RFC1095, RFC1189). Общими для них оставались MIB (RFC1066) - спецификации, описывающие формат данных.

> Уверен, что уже тогда не всем эта идея пришлась по душе.
>
> В те дни человечество ещё верило в ISO.

CMIP/CMIS - *Common Management Information Services/Common Management Information Protocol* - это такая же несостоявшаяся вещь, как OSI. Здорово всё напланировали, но временный SNMP заполонил всю планету.

Есть и другие слова, выдуманные в те дни, которые мы сейчас нигде в обиходе не используем: *HEMS, SGMP, NETVIEW, TNM, LANMANAGER, Network Computing Forum «Fat Document»*...

SNMP именно потому и был *Simple*, что на горизонте маячил *Common*. И его планировали держать «Simple», пока не откажутся. А вовсе не потому что он сам по себе был прост. Кажется, нам, как цивилизации, ещё повезло, что на смену SNMP не пришло что-то не столь Simple.

Итак, SNMP победил, если это можно считать победой. А другие рабочие группы распустили.

MIB - являл собой спецификацию, говорящую как системе управления, так и сетевому устройству, как собирать и интерпретировать SNMP-сообщения.

То есть на одной стороне есть инструкции того, как собрать пакет, на другой - как его прочитать. Остаётся только подставлять переменные и посылать через SNMP.

Ну какова красота?! Никакого чтения документации, никакой человеческой интерпретации - только строгое следование спецификации. И это начало 90-х!

Скажу вам больше - те же парни из IAB, ещё до того, как Cisco начала паровозить тему с SNMP, придумали **SMI** - *Structure of Management Information* - по сути язык моделирования данных - то, как именно данные будут структурированы в MIB.

Для нас не очень важно что и в течение какого времени нужно употреблять, чтобы написать и прочитать модели, написанные на языке ASN.1 - прародителе SMI. Для нас важно, что SMI дал жизнь **YANG**'у. Всё же не вся работа тогдашних групп была в стол. Но об этом позже.

Уже тогда, в 90-е, все хотели сделать что-то универсальное и отвязаться от вендорской специфики, но настойчивых попыток не предпринимали, а вендоры в погоне за time to market были ещё меньше заинтересованы вкладываться в стандартизацию того места, где им не придётся стыковаться друг с другом. Поэтому единой модели тогда не появилось.

Однако теперь следите за руками: к концу 90-х у нас уже были:

- Протокол - SNMP
- Спецификации - MIB
- Язык их моделирования - SMI
- Возможность стримить данные с железки на NMS - Trap'ы (ну серединка на половинку, конечно, но всё же)
- Целая пачка инструментов, утилит и NMS, работающих с MIB и SNMP - snmpwalk, MIB browser
- Желание вендоров поддерживать это и выпускать MIB'ы для каждой новой версии вовремя

По всей видимости мы были просто в шаге от дивного мира с единым фреймворком для сетевой автоматизации.

Но добавляя ещё один пункт:

- Никто из вендоров при этом так и не поддержал полноценное конфигурирование через SNMP

мы получаем ситуацию, в которой мы находимся прямо сейчас. Та-дам!

Но даже без этого в силу сложности (*S for Slozhnost*), вопросов к архитектуре, безопасности, транзакционности, нечитаемости спецификаций, непрогнозируемости результатов, невозможности проиграть изменения повторно, UDP в качестве транспорта и многим другим, SNMP нашёл применение лишь в задачах сбора данных с сетевых устройств и в крайне вырожденных случаях для настройки точечно тех или иных вещей.

Впрочем сегодня даже в вопросах мониторинга SNMP скромно уступает место NETCONF и gNMI.

Смахнули скупую слезу и забыли! И про SNMP и про CMIP/CMIS. Не забываем только про SMI.

Переходим к современности.

6.3 API

Давайте на парсек поднимемся над нашей ежедневной рутиной и взглянем на сеть с расстояния. И тогда можно увидеть, что программную работу с сетевым оборудованием следует рассматривать как взаимодействие с удалённой системой (чем она и является). А уже много лет назад для этого придумали термин - API.

То есть, если мы хотим чего-то от этой системы, мы используем известный интерфейс, для которого описано, как и что мы должны сделать, чтобы добиться желаемого результата.

API - это широкий и абстрактный термин, сам по себе не говорящий ничего о схеме данных, о формате и протоколах взаимодействия - просто программный интерфейс приложения - не человеческий, а программный.

И под одним этим зонтичным термином скрываются совершенно разные виды:

- REST API
- GraphQL
- XML RPC
- Linux Kernel API
- SOAP
- CORBA
- PCI шины
- JSON RPC
- Android API
- И сотни других

Хотя никто его так и не называл.

Одни из них, такие как REST, оперируют ресурсами и представляют набор операций над ними, в случае REST: CRUD - *Create, Read, Update, Delete*. То есть вы можете создать (Create) ресурс «билет на linkmeetup» и скачать (Read) его далее в любой момент. С REST мы уже разбирались как-то.

Другие виды API оперируют функциями, и позволяют на сервере запускать те или иные оговоренные программы. К последним относится класс API, который можно назвать RPC.

6.3.1 RPC - Remote Procedure Call

Этот термин родом из языков программирования. Ещё на рубеже 50-60-х годов языки вроде Fortran II и ALGOL ввели в обиход разработчиков процедуры (они же функции). С тех пор они везде - большинство языков - процедурные. Любое действие - это вызов процедуры - Procedure Call.

И когда-то эта процедура должна была находиться где-то в том же модуле или в соседних, но точно рядом и в том же окружении.

Но почему бы не слать вызов с параметрами на удалённую машину, где мы хотим что-то выполнить?

Например, мы могли бы по HTTP/FTP скачать несколько гигов данных sFlow с сервера и проанализировать их локально, а можем отправить сигнал на сервер, чтобы сложную статистику вычислил он сам и вернул результаты в ответе. Так вот второе - это удалённое исполнение кода.

Анналы истории говорят, что в в 1981 году Брюс Джей Нельсон, работая в Ксероксе, изобрёл концепцию и термин **RPC** - *Remote Procedure Call*.

RPC - удивительный клиент-серверный механизм, который позволяет запустить исполнение кода процедуры на другой машине так, словно бы он исполнялся локально. То есть разработчик просто привычным образом обращается к процедуре, не задумываясь о том, где и как она исполняется - главное, чтобы она вернула ответ.

А программа уже сама реализует взаимодействие с удалённой машиной.

Прелесть этого подхода в том, что он, во-первых, позволяет скрыть удалённый характер работы. А, во-вторых, на той, другой, стороне совершенно неважно, какая операционная система, архитектура, язык программирования и окружение - главное, чтобы они подчинялись одному протоколу.

Можно провести аналогию с TCP - не важно, какие операционные системы на хостах, желающих друг с другом общаться, - важно, чтобы они следовали спецификациям протокола TCP и его конкретной реализации - и тогда данные, отправленные одним хостом, будет возможно интерпретировать на другом.

Так в случае RPC, из-под винды в питоне, например, вы можете исполнить удалённую программу, написанную на Go, запущенную на линуксе. И никто вам не сможет помешать! (*Кроме сетевиков*)

Но что, по большому счёту, мы делаем, когда, зайдя по SSH, выполняем какую-то команду на коммутаторе или маршрутизаторе? Запускаем определённый код.

Например, сообщаем подсистеме BGP, что нужно теперь пробовать установить соединение с новым пиром.

Только представьте, как было бы восхитительно, если бы для вызова этого кода, не нужно было заходить на железку по SSH и вбивать команду?!

Взаимодействие между приложениями через RPC используется преимущественно в условиях, когда требуется обеспечить тесную связь между ними, когда они все формируют единую систему.

В то же время REST API наоборот требуется, когда компоненты должны быть достаточно изолированы и развиваться независимо.

Так REST обычно предоставляют внешним клиентам, B2B, смежным, но малосвязанным командам. Например, публикация постов в соц.сетях или агрегаторы авиабилетов при обращении к сервисам авиакомпаний. А RPC - там, где компоненты составляют часть чего-то большего, например, узлы банковской системы или микросервисные архитектуры.

В целом RPC - это концепция, не говорящая ничего о реализации.

Она постулирует, что на стороне клиента есть так называемый стаб (stub) - фрагмент кода, который реализует взаимодействие по RPC. Именно стабы делают для разработчика прозрачным вызов функции - из приложения вызывается этот стаб с набором параметров, а уже стаб делает удалённый вызов.

Ключевая часть RPC - спецификация - штука, которая на стороне сервера и клиента определяет, как работать с данными - как упаковать, как распаковать. Без участия человека, конечно же.

Язык, на котором пишется спецификация - **IDL** - *Interface Definition Language*.

Иными словами, на IDL пишется спецификация, на основе которой создаются и серверный интерфейс, и клиентский стаб. Это может быть, например, набор классов в питоне, имеющих функции для удалённого вызова, с которыми разработчик работает так, словно всё происходит локально - для клиента. И набор объектов Go - для сервера.

Наевшись с CLI и SNMP, сетевики придумали два протокола, которые используют под капотом RPC и при этом позволяют управлять сетевым железом:

- NETCONF
- gNMI (как фреймворк над gRPC)

6.4 NETCONF

Сытые 0-е и по ныне

Если вам по какой-то причине кажется, что стандарты рождаются где-то в недрах институтов, оторванных от жизни, то послушайте вот эту историю. Скорбно при этом помним про ISO.

В 1996 выходец из Ксерокса Прадип Синдху и Скот Кринс из StrataCom, купленной Циской, основали Juniper Networks. Идея создания мощного пакетного маршрутизатора пришла в голову Синдху, и он стал CTO компании, а второго наняли на роль CEO.

> Juniper M40. Источник

Вместе они создали легендарный M40 и лучший в мире интерфейс командной строки. До сих пор никто не сделал ничего лучшего - все только повторяют.

Операционка, предоставляющая клиенту обычный текстовый интерфейс, на самом деле перекладывает команды в XML, который используется для управления оборудованием.

Juniper
NETWORKS
M40

Так вот, их CLI и способ взаимодействия его с системой оказался настолько естественным и удачным, что его и положили в основу стандарта NETCONF в 2006-м году. Не без участия Juniper Networks, конечно же, появился RFC4741. Будем честны, один только джунипер там и постарался в практической части. И то тут, то там будут проскакивать его куски, начиная с set и заканчивая candidate config.

Вот как он был определён в нулевых:

```
Abstract
The Network Configuration Protocol (NETCONF) defined in this document
provides mechanisms to install, manipulate, and delete the
configuration of network devices.  It uses an Extensible Markup
Language (XML)-based data encoding for the configuration data as well
as the protocol messages.  The NETCONF protocol operations are
realized on top of a simple Remote Procedure Call (RPC) layer.
```

И определение с тех пор не менялось - вся суть NETCONF в этом параграфе.

Но как так получилось, с чего началось? Да с того, что в начале 2000-х IAB проснулся в одно недоброе утро и осознал, что все планы по CMIP мир провалил, SNMP прорастил свои корни глубоко и перестал быть Simple, никто из вендоров так и не реализовал на 100% его поддержку, в самой аббревиатуре SNMP «M» вместо *«Management»* стала обозначать *«Monitoring»*, и к тому же единой модели данных конфигурации не получилось. Хуже того - появился этот выскочка Juniper, который везде суёт свой нос.

В общем умники из IAB крепко призадумались. И собрались снова - на этот раз в офлайне 4-го июня 2002-го года в Рестоне (это в Штатах - любопытный городок - почитайте), чтобы *«продолжить важный диалог, начатый между операторами и протокол-девелоперами»*.

Сели, похаяли SNMP, покекали с аббревиатур COPS, SPPI, PIB, CIM, MOF и записали это всё в –тик-ток– RFC3535.

Выхлопом этой встречи стали 33 наблюдения и 8 рекомендаций. Среди них есть действительно важные, определившие наше настоящее.

```
1. Программные интерфейсы должны предоставлять полное покрытие,
иначе они не будут использоваться операторами, поскольку они
будут вынуждены использовать CLI.
5. Необходимо строгое разделение между конфигурационными и операционными
данными
8. Необходимо иметь возможность выгрузить и загрузить конфигурацию
в текстовом формате в единообразной манере между всеми вендорами
и типами устройства
9. Желательно иметь механизм доставки конфигурации в условиях
транзакционных ограничений.
14. Необходимо, чтобы устройства поддерживали как программный,
так и пользовательский интерфейс
15. Внутренние операции на устройстве должны быть одинаковы
как для программного, так и для пользовательского интерфейсов.
26. Должна быть возможность произвести операцию над указанной секцией
конфигурации.
27. Должна быть возможность выяснить возможности устройства
28. Необходимы безопасный транспорт, механизмы аутентификации
и авторизации, поддерживаемые текущей инфраструктурой.
30. Полная конфигурация устройства должна быть применима через один
протокол.
```

Часть из них мы воспринимаем сегодня как самоочевидное, мол, а как вы ещё иначе могли бы такое сделать? Но это не воспринималось так тогда. Просто вспомним как устроен SNMP :)

А ещё были явно полезные рекомендации:

- Рабочее совещание рекомендует прекратить форсить рабочие группы предоставлять конфигурационные MIB'ы
- Рабочее совещание рекомендует не тратить время на CIM, COPS-PR, SPPI PIB

В общем-то какие претензии к SNMP и его компании заставили уважаемых людей собраться на три дня?

- Проблемы масштабирования. Забирать большие объёмы данных с большого количества устройств он не был рассчитан.
- Транзакционность изменений на устройстве, и тем более на сети, должна была поддерживаться не протоколом и устройством, а системой инструментов.
- Откат также лежал на инструментах.
- Writable MIB не покрывали большей части задач по настройке устройства.
- Весь этот куст OID'ов был крайне сложночитаем для человека. Понять, что произойдёт после работы скрипта было очень сложно. *Сколькие из вас отчаялись, пытаясь его понять?*
- Не было никакого инструмента, который позволял бы повторно выполнить те же действия идемпотентно на этом же устройстве или на другом.
- Контроль состояния тоже отсутствовал.

В итоге протокол, призванный решать вопрос автоматизации, не особо-то для этого подходил.

Короткий итог встречи: IETF всё это время что-то там придумывал, разрабатывал, чтобы сделать жизнь операторов проще, а те не будь дураками, пришли и наконец сказали, что, мол, вы тут штаны просиживаете, а ничего полезного для нас не делаете, а делаете вы бесполезное! И ISO туда же!

И в этот момент Juniper из-за угла приоткрывает полу своего XML-API.

И он оказывается настолько более лаконичным (это XML-то!) и удобным, что рабочая группа внезапно решает принять его концепции в качестве стандарта **NET**work **CONF**iguration protocol - RFC4741. Упор на Configuration в названии - это, видимо, гиперкомпенсация отсутствия режима конфигурации в SNMP.

Вот так в итоге скромно упомянут джунипер в этом RFC:

```
In the late 1990's, some vendors started to use the Extensible Markup
Language (XML) [XML] for describing device configurations and for
protocols that can be used to retrieve and manipulate XML formatted
configurations.
```

А через 5 лет, в 2011, исправленное и дополненное издание вышло под номером RFC6241. Там уже потрудились несколько университетов и компаний. Одной из них стала восходящая звезда сетевой автоматизации Tail-f, купленная и погубленная в 2014-м году циской. Нет, формально она, конечно, осталась внутри как отдельный Business Unit, но в большой мир они отсвечиваюь теперь только Cisco NSO, хотя могли бы приносить большую пользу. Впрочем, может, я зря наговариваю? Надо будет потрогать его.

И вот в операторские сети на белом коне въезжает NETCONF.

- Работает по SSH (и не только),
- Представляет данные в структурированном виде,
- Разделяет конфигурационные и операционные данные,

- Имеет несколько операций над данными: create, merge, replace, delete, remove,
- Может обеспечить контроль целевого состояния конфигурации,
- Поддерживает концепцию нескольких версий конфигурации (datastores),
- Может поддерживать commit конфигурации. Обеспечивает транзакционность,
- И вообще красавчик.

Причём Juniper его поддерживает с нулевого дня. И в полной мере, потому что для него это максимально естественно - это и есть его API.
А вот внутренний API той же Циски или Хуавэя не ложится так гладко на XML и какую-либо простую схему. Для них поддержка NETCONF - это большая работа, которую они выполняют с переменным успехом. Коммиты, операция *replace* - это всё даётся тяжело. А именно в них вся сила.

Datastores - это различные версии конфигурации на устройстве: `running`, `candidate`, `saved` и, возможно, другие. Они позволяют не менять на лету работающую конфигурацию.
Commit обеспечивает три буквы **ACID** - *Атомарность*, *Консистентность* и *Изолированность*.
Операция Replace - мощнейшая штука - позволяет заменять всю или часть конфигурации на новую.
Мы привыкли, что в CLI нам нужно сформировать список команд, добавляющих новую конфигурацию, и команд - удаляющих старую - ненужную. Довольно простая операция для человека, но чудовищно сложная для автоматики. Мы настолько привыкли, что это даже не вызывает раздражения у нас.
А с NETCONF replace - мы просто суём ту конфигурацию, которую хотели бы видеть, а коробка сама считает, что нужно сделать, чтобы к ней прийти из текущего состояния. Это и есть тот самый декларативный путь, к которому мы так стремимся.

Для работы с NETCONF есть библиотеки для питона (и синхронные, и асинхронные), для го, плагины для Ансибл.
Вроде бы всё - бери и пользуйся. Но не все производители его поддерживают. И совсем немногие поддерживают его в полной мере. Где-то нельзя настроить DHCP-Relay, где-то нет секций IPv6-vpn AF в BGP, где-то `replace` не поддерживается или поддерживается, но работает через delete/create - ух, неспасибо за это.
В итоге пара пунктов из вышеупомянутого RFC3535 нарушены: не всё можно настроить через этот новый протокол, а для настройки всех возможных функций нужен как минимум CLI.

Но своё место NETCONF уже прочно занял и будет дальше только расширять и углублять. Несколько вендоров действительно его поддерживают в полной мере. А на других точечные операции всё равно многократно удобнее через программный интерфейс со структурированными данными выполнять. Плюс своё давление оказывают крупные заказчики, требующие его поддержки.

6.5 RESTCONF

Буйные 10-е и забыли

```
The workshop recommends, with strong consensus from the operators
and rough consensus from the protocol developers, that the
IETF/IRTF should not spend resources on developing HTML-based or
HTTP-based methods for configuration management.
```

RFC3535. Recommendation 6

SSH - это хорошо. Но на сетевую автоматизацию случился спрос, а за ним наплыв сил разработчиков. А вот эти разработчики хорошо шарят в REST, но на курле крутили все эти наши SSH и парсинг текста.

В компаниях, где начинают заниматься автоматизацией сети, обычно уже есть свой штат разработчиков, инструменты, практики. И они в лучшем случае рассматривают сеть, как ещё одни сервера, а то и ещё один сервис.

И вот REST API с CRUD им очень знаком.

Вот и решили парни из циски, джунипера и tail-f: *«а почему бы не запилить REST API в сетевые коробки?»*. Ну пошли и запилили - делов-то. И назвали RESTCONF - всё ещё отзываются боли SNMP.

Драфт был опубликован в 2014м, а в 2017 мир увидел RFC8040.

Это помесь RESTAPI и NETCONF, которая была призвана упростить управление сетью для WEB-приложений.

Внутри идеологически это NETCONF с его datastores и способами работать с конфигурацией, однако в качестве транспорта - HTTP с набором операций CRUD, реализованных через стандартные методы (*GET*, *POST*, *PUT*, *PATCH*, *DELETE*).

Конфигурационные данные передаются в формате JSON или XML.

В качестве модели данных используется только те, что написаны на языке YANG - тут уже никакой самодеятельности.

С самого начала RESTCONF не затевался как замена NETCONF, а только как более удобный для WEB-приложений способ работать с сетевыми устройствами. То есть они должны сожительствовать.

При этом обычно на устройстве реализуется один бэкенд, обрабатывающий запросы на работу с конфигурацией и опер.данными от разных фронтов - NETCONF или RESTCONF. То есть в основе одни и те же datastores, один и тот же движок, вычисляющий конфигурационные дельты, но сложность транзакционности и нескольких разных видов конфигураций (`running`, `candidate`, `saved`) от пользователя скрыта в случае NETCONF.

С другой стороны отсутствие в выдаче поисковиков хоть сколько-то серьёзных работ по автоматизации с помощью RESTCONF и даже популистских статей от больших игроков говорит о том, что это всё не более чем баловство. И я намеренно не пишу слово «пока». Лично я в него не верю.

При этом CRUD не очень гладко ложится на RPC-подход, да и в идее держать открытым на сетевом железе HTTP есть что-то противоестественное, согласитесь? Нет? Ну ладно.

Просто жаль сил, вложенных в этот протокол. Потому что на пятки ему наступает gRPC/gNMI.

6.6 Модели и языки

Однако вернёмся к NETCONF: в чём его фундаментальная проблема? Да в том, что он вышел в мир один одинёшенек. Не было предложено никаких схем, языка, стандартов для семантики. И всё пошло вразнос.

Модели были нужны, но языка для их описания не было. До 2010 (на самом деле больше) каждый вендор писал их кто во что горазд.

6.6.1 YANG, который (по-)меняет мир

Очень странно это, конечно, вышло. Для SNMP IETF много думали, работали и выпустили сначала язык спецификации SMI, а потом даже замахнулись на SMIng - nextgen, так сказать.
То есть необходимость языка описания спецификации была очевидна уже тогда - в 90-е, однако к NETCONF язык не приложили почему-то.

Впрочем это всё-таки довольно быстро стало понятно - в 2008 из осколков рабочих групп по SNMP слепили рабочую группу IETF NETMOD, которая в срочном порядке занялась разработкой языка. Не мудрствуя лукаво, они взяли синтаксис SMIng и «адаптировали» его. Уже в 2010 они выпускают YANG 1.0, а в 2016 - 1.1.
YANG - *Yet Another Next Generation* - по сути - это язык описания моделей. То есть это не данные и даже не конкретные модели - только язык. Как русский - это не произведение и не слова.

А уже с помощью этого языка создаются непосредственно модели, которые обычно так и называют - YANG-модели.
Модели на языке YANG далее могут преобразовываться в XML/JSON-схемы или в gRPC Protobuf'ы или во что угодно другое, что станет спецификацией для протокола.
И уже на основе этой спецификации можно генерировать конфигурации или проверять их валидность.

Четырёх лет задержки оказалось достаточно, чтобы вендоры понаделали кучу своего, на что завязали инструменты и они сами, и их заказчики.
Четыре года задержки откинули внедрение Model-Driven подхода лет на десять. Только сегодня хоть что-то похожее на практическое применение начинает выходить за пределы гуглов и фейсбуков.

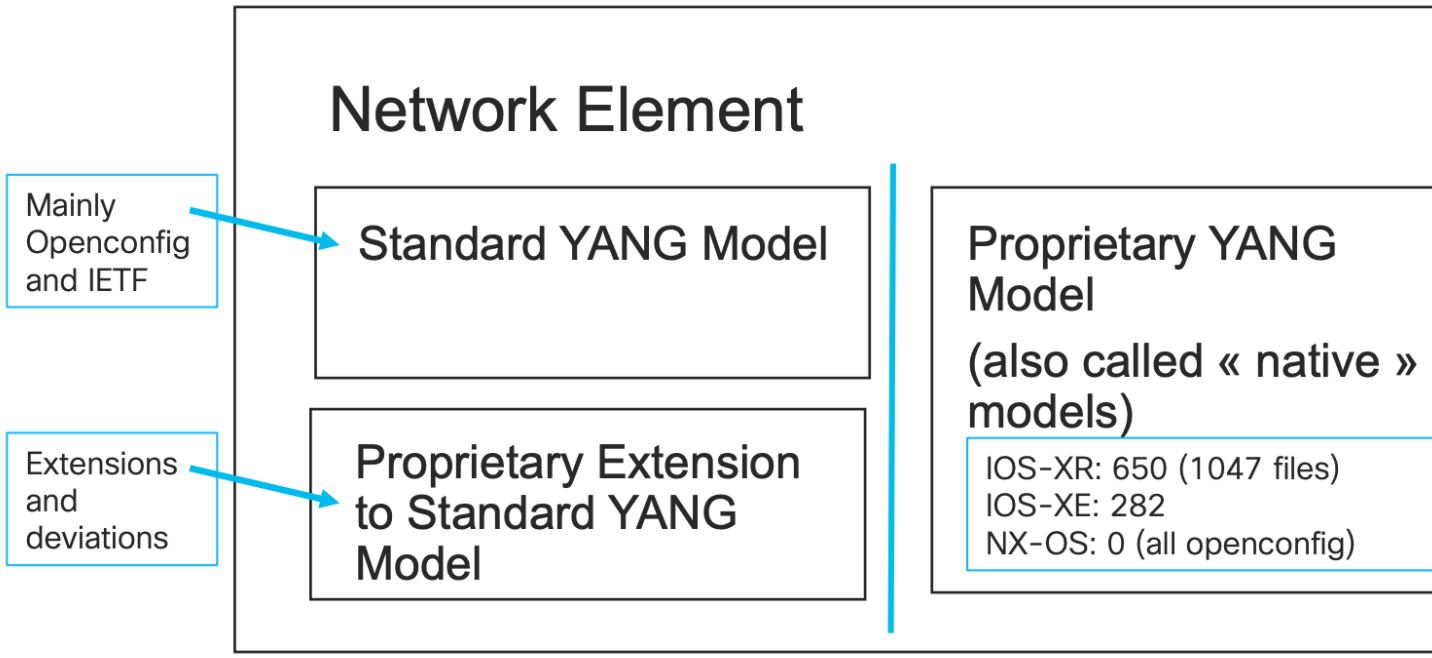
Кстати, будьте аккуратнее, когда ищете *«yang models»* в интернетах, серьёзно вам говорю.

6.6.2 Виды моделей

Вендоры очень быстро сориентировались в ситуации на самом деле - и довольно скоро насоздавали YANG-модели для своих устройств.

Проприетарные, они же Native

У каждого производителя набор моделей свой собственный, никоим образом несовместимый с любыми другими. А зачастую просто даже ничего похожего между ними нет.



Но это уже большое дело - теперь вся ваша автоматика может полагаться на них при генерации и валидации конфигурации, при сборе статистических данных, при разборе телеметрии. Известно какого типа и в какой ветке иерархии вернутся те или иные значения.

Где их можно взять?

Говорят, что можно прям запросить с устройства YANG-модель через операцию `<get-schema>`, но не все вендоры это поддерживают.

Говорят, некоторые вендоры выкладывают модели в GitHub, но не все и не всё.

Говорят, что можно скачать модели с сайта производителя.

Пусть говорят: но универсального пути тут нет, увы.

Главное: с этим уже можно было жить.

Инженерам стало нужно чуть меньше думать об интерфейсах и форматах сообщений, но с глубоким вниманием подходить к содержимому сообщений всё ещё приходилось, оказывая разные знаки почтения разным вендорам.

При этом казалось бы - вся сеть - это конечный набор одинаковых сервисов, если выбросить всякие IGRP, HSRP, RRPP и прочие проприетарные выдумки. Ну, всем же нужен IP, OSPF, BGP? Всем нужна аутентификация на устройствах и SSH? Они не могут иметь очень уж принципиальные отличия, как минимум из-за необходимости поддерживать совместимость друг с другом и соответствовать RFC.

Так почему мы делаем это сотней разных способов?

Сделать по отдельности у каждого вендора Configuration State Management - одноразовая, решаемая (а много где и решённая) задача. А вот договориться между всеми производителями, как должна выглядеть универсальная модель - так же сложно, как и любая другая задача, где людям нужно договориться.

Но ни один из зарождавшихся и выживших стандартов или не ставил целью унификацию вообще, или пытался поднять этот вопрос, но был выброшен в окно штаб-квартиры вендора.

Хотя вру. IETF предприняли отчасти успешную попытку написать универсальную модель.

IETF-модели

Разработав язык YANG, инженеры IETF поняли, что напрашивается и мультивендорная модель.
Ещё в 2014-м году были сделаны первые коммиты с этой моделью в репозиторий YANG.
С тех пор много накоммичено, но мало фактически сделано. Общепризнанно, что IETF-модель очень медленно развивается, у неё низкое покрытие, а схема не выдерживает критики.
С IETF-модели рекомендуют начинать, потому что она якобы проще, а уже потом переходить на OpenConfig, но как по мне - это напрасная трата времени.
Будущее её туманно, если не сказать непроглядно. Однако вендоры её поддерживают. Ну, кстати, Openconfig-модель из IETF-модели тоже кое-что импортирует, например, частично описание интерфейсов.

Заказчиков и пользователей беспокоили ущербность модели и инертность IETF, но один в поле не воин - тысячи разрозненных автоматизаторов по всему миру не могли ничего с этим сделать. А вот большие компании могли.
Когда надо настроить тысячу свитчей, а каждый месяц запускать новый датацентр, когда на сети пять разных поколений дизайна, а катить изменения нужно дважды в день, начинаешь несколько иначе смотреть на все этим ваши сиэлаи и вендор-специфичные эксэмэли.

Могу только предположить, что в недрах гугла это происходило примерно так:
Вот была сеть из дюжины вендоров, были некие драйверы, которые могут доставлять конфигурацию на сетевые коробки. А ещё была где-то далеко стоящая база данных с переменными. А между ними 2 миллиметра антивещества.
Скорее всего, сначала появился некий дизайн сети, которые в суперпозиции с БД давал вендор-нейтральную конфигурацию.
Этот дизайн сети уже опирался на разработанную внутри модель данных - ведь в нём нужно было описать все нюансы конфигурации. То есть или уже была или параллельно с дизайном появлялась модель данных.
А вместе с тем набирал обороты gRPC. И на каком-то из удачно расположенных кофе-поинтов пересеклись парни из соседних отделов и подумали:

> - Слышь, а зачем вам эти полумеры? Давай из вашей модели сразу же в коробку перекладывать? Мы вам поможем агента написать
> - Да, но у нас циски, проприетарная ось.
> - Да это фигня. О, Джон, здоров. Давай парням линукс на свитчи вкорячим?
> - Так давай, изян. Через сколько месяцев надо?
> - Подождите, подождите, там типа чип, SDK, памяти маловато
> - Хей, Рони, алло! Нам нужен свитч, на который мы можем свою операционку поставить
> - Без базы, ща, в R&D запустим.

Ну как-то так я себе представляю рождение OpenConfig.

OpenConfig - мечта, становящаяся явью

Возможно, впервые за шестидесятилетнюю историю телекоммуникаций у нас появился шанс изобрести свой USB Type C. Представьте мир, в котором Cisco, Juniper, Nokia и Mikrotik настраиваются одними и теми же командами и это к тому же приводит к одинаковому результату?

Я не могу.

OpenConfig - это открытая YANG-модель, которая предполагается единой для всех вендоров. Одна стандартизированная модель для сбора операционных данных с устройств, управления конфигурацией и анализа телеметрии.

Итак, OpenConfig появился в Google, как они сами сказали на наноге в 2015, как ответ на следующие вызовы:

- 20+ ролей сетевых устройств
- Больше полудюжины вендоров
- Множество платформ
- 4M строк в конфигурационных файлах
- 30K изменений конфигураций в месяц
- Больше 8M OIDs опрашиваются каждые 5 минут
- Больше 20к CLI-команд выполняется каждые 5 минут
- Множество инструментов и поколений софта, куча скриптов
- Отсутствие абстракций и проприетарные CLI
- SNMP не был рассчитан на столь большое количество устройств и на столько большие объёмы данных (RIB)

Это всё настолько знакомые ежедневные трудности, что любой может приписать их себе, просто уменьшив цифры.

Вскоре после этого в том же 2015м был сделан первый коммит в публичную репу openconfig/public.

Так начал своё шествие по индустрии OpenConfig.
Вот тут все модели данных, разработанные и опубликованные в OpenConfig.

> Картинка побольше

Никаким стандартом он не стал, в RFC не превратился, но вендоры его подхватили. Ещё бы они его не подхватили - очень быстро к гуглу подтянулись и другие гиганты - за ОС теперь топят десятки компаний.

Есть только пара проблем - карта старовата и некоторые ссылки на сайте ведут на 404 :)
Но репозиторий живёт насыщенной жизнью.
Есть и ещё пара проблем посерьёзнее, но о них в конце главы.

OpenConfig сегодня даёт возможность настройки стандартных сервисов, таких как интерфейсы, IP-адреса, NTP, OSPF и прочее. Безусловно, речь не идёт про вещи, завязанные на аппаратные особенно-

сти: QoS, управление буферами и ресурсами чипа, сплиты портов, работа с трансиверами. И в каком-то хоть сколько-то обозримом будущем этого ждать не стоит.

Хуже того, на сегодняшний день многие вендоры, ввязавшиеся в поддержку OC, не реализуют все 100%, а лишь часть - ту, которая нужна им, а точнее, их заказчикам.
Но BGP с OSPF настроить точно можно.

И что же делать, если брать 5 разных несвязанных Native-моделей не хочется, а OC-модель не покрывает всех необходимых функций?

И есть два пути.
Один из них - брать OC и видоизменять его с помощью добавления или убирания каких-либо его частей.
Когда вендор хочет расширить покрытие модели - он делает **augmentation** (расширение, дополнение), встраивая его в нужное место.
Если он хочет поменять какое-то поведение или удалить функционал - он описывает **deviation** (отклонение) к базовой модели.

Этот способ, конечно, не покрывает все потребности.

Другой - совмещать OC и Native. В целом рекомендуют (даже сами вендоры), использовать OC там, где это возможно, а где нет - прибегать к Native. Главное - не настраивать одно и то же с помощью разных моделей.

> Источник: доклад на Cisco Live

Если вам всё ещё кажется, что так можно жить, то пришло время сказать, что разные вендоры, оборудование и даже версии ПО могут использовать разные версии OC-модели и быть не полностью совместимыми. Вам всё ещё придётся думать о том, что и куда вы деплоите.

OpenConfig входит в наш мир в ногу с **gNMI**, как это и задумывал Google.
Но в качестве транспорта может быть как gNMI, так и NETCONF и RESTCONF - это не принципиально, потому что OC - это только YANG-модель, которая далее может быть переложена уже хоть в XSD, хоть в JSON-схему, хоть в gRPC protobuf'ы.

6.7 gRPC/gNMI

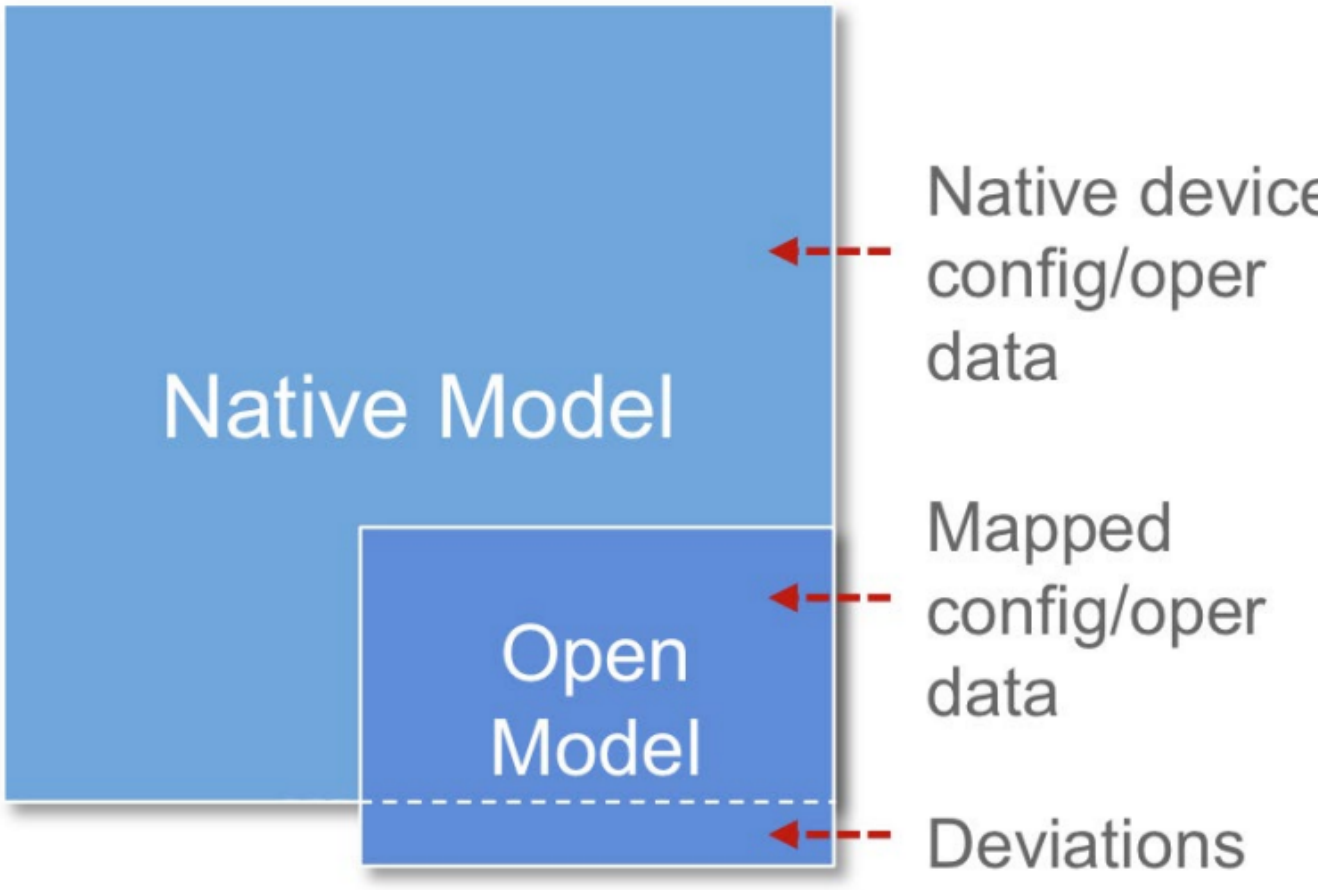
Сверхлихие 20-е

За последние лет семь gRPC уже всем уши прожужжали. И только самые ловкие разработчики могли избежать реализации взаимодействия с какой-нибудь системой по gRPC.

> «g» в *gRPC*, кстати, означает вовсе не «google».

6.7.1 gRPC

Вообще-то RPC вроде бы как начал давным давно уходить в тень, уступая место REST и ему подобным. Но в недрах гугла он цвёл, эффективно связывая между собой микросервисы, и назывался


Native Model
Open Model
Native device config/oper data
Mapped config/oper data
Deviations

Stubby. Ровно до тех пор пока, в 2015 они не решили его переписать и заопенсорсить, чтобы нанести непоправимую пользу миру.

Долгое время в изученной Вселенной не существовало никаких общедоступных библиотек, позволяющих реализовать какой-то типовой RPC. Разработчики сами описывали и сообщения, и формат данных в них, и как их интерпретировать. Поэтому и популярности особой он не сыскал.
А тут вот, пожалте: готовый протокол, стек, формат данных и библиотеки для кучи языков.
Что же он из себя представляет?

Это фреймворк, позволяющий приложениям, запущенным в совершенно разных окружениях, взаимодействовать друг с другом посредством RPC.
Делает gRPC концептуально ровно то, что предполагается самой идеологией RPC, но есть несколько вещей, которые обусловили его успех и популярность:

1. Строгий IDL (Interface Definition Language), диктующий то, как именно описывать спецификации - protocol buffers или protobufs.
2. Готовый формат данных и механизм их маршалинга и демаршалинга - тоже protocol buffers (protobufs).
3. Библиотеки для разных языков программирования, которые на основе спецификации генерируют объекты языка (классы, методы итд) - разработчику остаётся только использовать их. Как для сервера, так и для клиента.

То есть. Поставил себе пакет grpc: перед тобой сразу язык спецификации, генераторы кода, интерфейсы, форматы данных, транспорт. Красота-тра-та-та!

Мы не знаем сколько лет внутри гугла gRPC набирал популярность и проникал всё глубже в межсервисное взаимодействие. Но что теперь известно точно, так это то, что у них менеджмент с яйцами, а сетевые инженеры достаточно гибки и пытливы, чтобы и к сети адаптировать этот единый протокол.
При этом не забываем, что на проприетарные джуносы, иосы и врп никто не притащит свой бинарничек, чтобы удобный для себя интерфейс реализовать. Это значит, что white-box коммутаторы с собственной linux OS у гугла появились задолго до того как их увидел мир.
Что и неудивительно - с железом они работать умеют, с Linux и подавно - дело было за малым - собрать команду Network R&D, в которой будут ребята, которые занимались разработкой своих серверов и адаптацией интерфейсов и инструментов, и найти достаточно гибкого вендора. А за последним дело не встанет, когда вы закупаете килограмм свичтей в секунду.

Так по мнению поисковых систем выглядят крутые сетевики

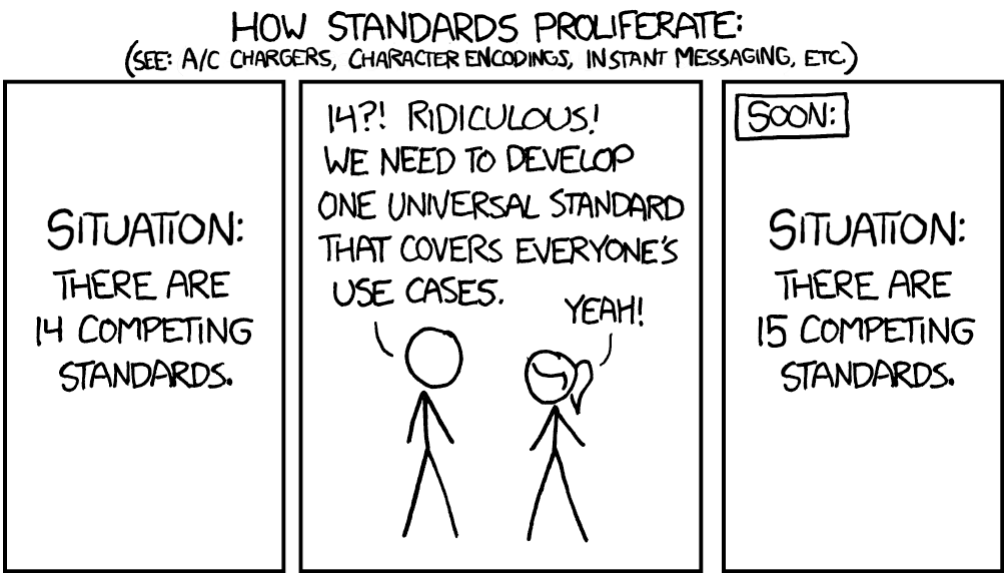
Вообще для обывателей всё началось 24 сентября 2015, когда OpenConfig consortium выпустил OpenConfig в мир. Весь FANG (кроме Amazon) поучаствовал в этом консорциуме. Но начал всю заварушку и продолжает её паровозить гугл. Естественно, среди них и крупные телекомы, вроде Level3, AT&T, Verizon, Bell.

И пока OpenConfig прокладывал себе дорогу, раскидывая в сторону вендорские и IETF модели, гугл сделал следующий шаг - как раз таки реализовал gNMI.

Итак, в 2016-м мир увидел плод труда инженеров гугл - протокол **gNMI**, реализующий весь стек технологий для программного взаимодействия с железом.

И что с того?! Ведь к тому времени буйным цветом шёл NETCONF и к тому же почти одновременно с gNMI уже почти сформировался RFC 8040, описывающий RESTCONF со вполне ещё модным на тот момент REST.

Как в таких условиях пробиться ещё одному протоколу и не стать героем известной картинки?



Источник

Так вот, рассказываю: собрались как-то сетевики гугл вместе, пришли на встречу IETF 98 в Чикаго на секцию Routing Area Working

Group и прямым текстом им заявили, что то, что те навыдумывали, пора пришла заменить на молодёжные технологии.

Шёл 2017-й год. Марат устроился в Яндекс.

И... Ничего не изменилось.

В 2018 они, видимо, поняли, что их не услышали и на IETF 101 снова пришли с рассказом про gNMI, и уже более явно сообщали, что он пришёл на замену этим вашим x-CONF'ам. Слышите вы, старпёры? Ало?! gNMI пришёл!

И тут завертелось! Сообщество сетевых автоматизаторов из вендоров, телекомов и просто одиноких пассионариев понесло благую весть в народ.

Как вы видите, gNMI молодой и дерзкий протокол. Про него нет страницы на вики, довольно скромное количество материалов и мало кто рассказывает о том, как его использует в своём проде.

Он не является стандартом согласно любым организациям и RFC, но его спецификация описана на гитхабе.

Однако свою дорогу в мир прокладывает. Медленно, но, похоже, что верно.

Что нам важно знать о нём? gRPC Network Management Interface.

Это протокол управления сетевыми устройствами, использующий gRPC как фреймворк: транспорт, режимы взаимодействия (унарный и все виды стриминга), механизмы маршаллинга данных, proto-файлы для описания спецификаций.

В качестве модели данных он может использовать YANG-модели, а может и не использовать - protobuf'ы можно сгенерировать на основе чего угодно, и даже просто написать вручную.

Как того требует gRPC, на сетевом устройстве запускается сервер, а на системе управления - клиент. На обеих сторонах должна быть одна спецификация, одна модель данных.

gNMI в мир пришёл под руку с OpenConfig, но неразрывно они друг с другом не связаны.

А ещё, что немаловажно, gNMI приводит с собой стриминг телеметрии. Впервые в истории хоть кто-то наконец подумал о том, что push-модель на сетевом устройстве может быть эффективнее pull, как делали системы мониторинга на основе CLI, SNMP и NETCONF. Можно подписаться на рассылку и хоть несколько раз в секунду получать метрики и даже анализировать утилизацию буфера на чипе. И для всех этих данных есть модели, позволяющие удобно с ними работать.

В этой статье я не копаю глубоко в каждый протокол и фреймворк, не разбираюсь, как они устроены, а даю только взгляд на историю развития автоматизации. За деталями приглашаю во шестую часть.

6.8 Настоящее сетевой автоматизации

Ну и пришло время подводить итоги?

Итого, что же творится в мире сетевой автоматизации сейчас?

Тут на самом деле вопрос, где вопрошающий и отвечающий находятся на спектре от чед-инженера «руками фигакну тыщу свичей, хорошо, если есть Excel» до непорочного инженера, кругом обложившегося gRPC и OpenConfig'ом с сетевым CI/CD-пайплайном.

А посередине спектра и ансибль, и питон, и перл, и баш, и го. Где-то CLI, куда-то NETCONF уже внедрили, кто-то по gNMI что-то настраивает и телеметрию снимает. Да добрая половина и SNMPv2 ещё не выключила.

У многих конфигурация Day 0 и 1 плюс-минус автоматизирована - это правда легко.

Day-N у кого-то решается в полностью ручном режиме, иные раскатывают обновления скриптом или простым Ansible playbook'ом, кто-то даже готовит развесистые плейбуки, основанные на модулях, «поддерживающих» состояние.

У кого-то есть крутая CLI-автоматизация , умеющая по-настоящему поддерживать состояние.

Самые продвинутые укладывают всё в NETCONF, собрав полноценный замкнутый цикл релиза конфигурации.

А совсем оторванные от земли заставляют производителей поддерживать новую функциональность в OpenConfig.

В целом описать всё многообразие проявлений автоматизаторской фантазии сегодня просто невозможно. Мы сейчас в мире, в котором чёрный параллелепипед ещё не был признан стандартом в области форм-факторов смартфонов.

6.9 Видимое будущее

Тут на самом деле два направления развития.

Во-первых, совершенно очевидно будут стремительно набирать популярность **программные интерфейсы** взаимодействия с железом и набивший оскомину на конференциях model-driven programmability.

Рано или поздно он сойдёт со слайдов и начнёт свой путь в каждое домохозяйство.

gNMI слишком хорош, чтобы пройти мимо него. Да и NETCONF, настоенный на YANG'e, тоже.

Будем видеть всё больше статей, больше лаб, разборов. Всё больше вендоров будет нормально поддерживать их и увеличивать покрытие.

Такие требования начнут появляться в RFI.

И это очень-очень-очень хорошо.

Другое направление более интересное и многообещающее.

Whitebox-оборудование. Лет 5 назад это было таким же шумом, как сейчас gNMI. Это - где-то там, у них - у больших и сильных - есть и железо, и софт, и штат R&D для этого интересного.

Сегодня Cumulus, Switchdev, Onos, SONiC - уже вполне зрелые операционные системы, на которых работает прод. Broadcom опубликовал свой SDK на github - это вообще из разряда «чё творится-то?!»

Выпуском железа занимаются уже давно не только новоиспечённые игроки, вроде mellanox'a (у нас с ними был подкаст), delta Xpliant, Barefoot (у нас и с ними был подкаст), но и вендоры - так называемые Britebox - Branded white box - cisco, huawei итд.

Почему это важно? И почему эта информация вообще получила место в статье про историю автоматизации?

А всё просто. С появлением Whitebox в мир сетевых операционных систем приходит Linux.

«Да он там и до этого был» - скажут некоторые. Не секрет, что почти все проприетарные сетевые ОС основаны на Linux или FreeBSD. И что с того, если доступа к консоли обычно нет, не говоря уж о руте?

А на Whitebox'ах стоит самый что ни на есть честный Linux, на который можно установить пакеты из репозитория, притащить любой файл, любой бинарник, любой скрипт.

Он превращается в обычный сервер с о-о-очень модными сетевыми карточками с чипом на 12,8 тера.

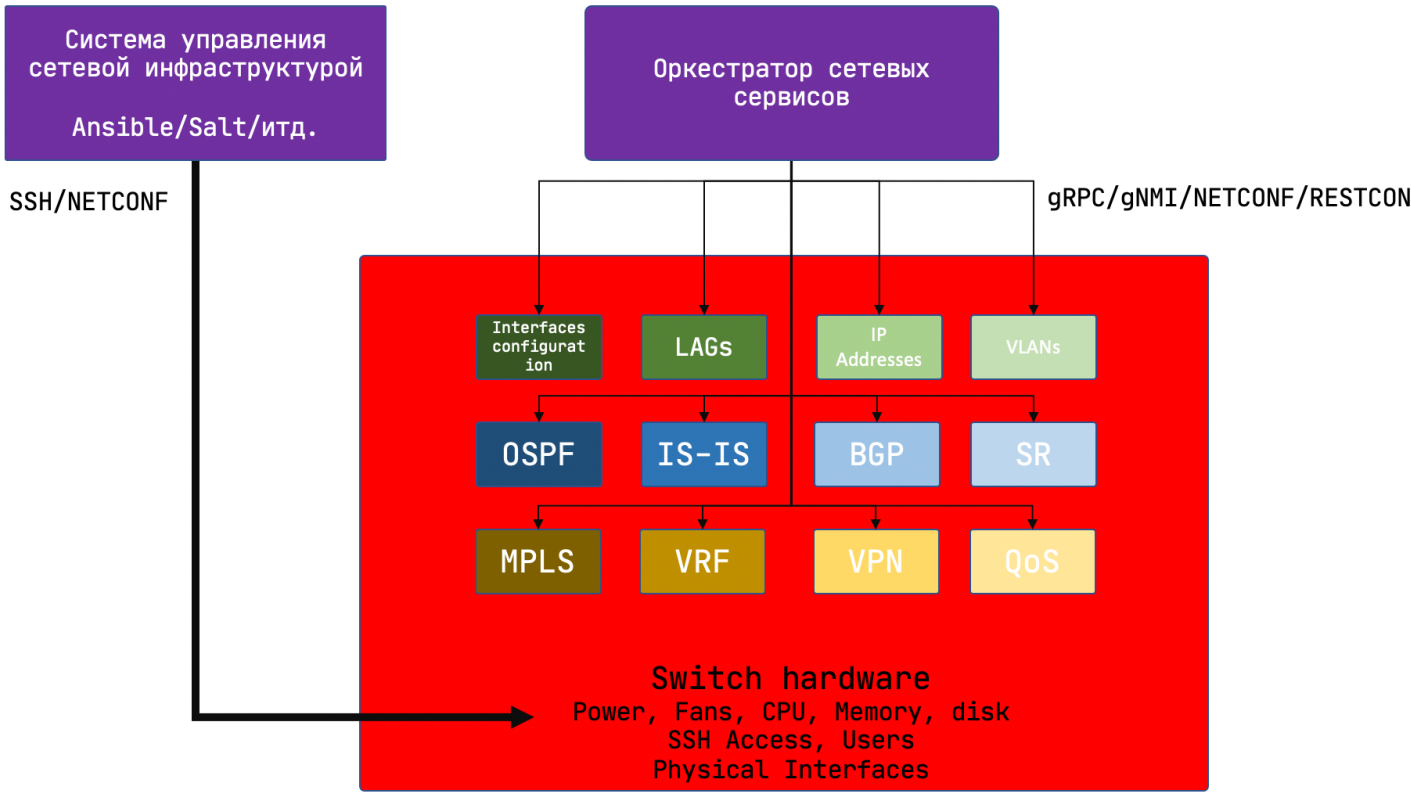
На нём есть файлы конфигурации, systemd, cron.

А это означает, что обслуживать его можно как обычный Linux-сервер.

Можно поднять nginx и совершенно любое REST-приложение за ним. Ну или gRPC-сервер. Поставить телеграф-агент - и сливать метрики в коллекторы.

И здесь совершенно чудесно ложится идея разделения управления устройством и сервисами, которые на нём запущены.

Есть Linux-тачка. На ней CPU, память, интерфейсы, пользователи, мониторинги системы и всё такое прочее: но весьма ограниченный набор. Доступ - по SSH. Инструмент - любой, используемый для управления физическими машинами - Ansible, Salt, Chef.



А есть сервисы - BGP, VPN, VxLAN, который на физической машине запущен. И вот они могут управляться через REST, gRPC, ну или хотя бы путём подсовывания конфигурационных файлов.

И менеджмент конфигурации сети становится задачей других, уже существующих отделов: серверной инфраструктуры и config-плейна. Без размазывания компетенций и накладных расходов. **Сетевики всё ещё нужны**, очевидно - для определения того, что именно нужно настраивать, как будет выглядеть архитектура сети, набор протоколов, модель данных.

Сюда же можно подтащить версионирование работающих приложений, в которое можно включить и конфигурацию, запуск двух версий side-by-side и где-то не за горизонтом даже полноценный Continious Integation маячит.

А все заморочки с ансиблом, питон-скриптами, притворяющимися человеком с руками и какой-никакой головой, поддержанием стейта конфигурационной портянки текста - они просто испаряются в пред-рассветном тумане.

И попробуйте после этого взглянуть на существующую модель работы с сетевым железом - какой кривой и инертной она выглядит.

Но есть «Но».

Во-первых, это требует глобальной перестройки всех процессов и вообще-то сознания. Такие вещи за 5 лет не происходят в индустрии - мы ещё долго будем развивать NETCONF.

Во-вторых, это требует всё же некий R&D - в штате должны быть крепкие инфраструктурщики и разработчики. В то время как идеально вылизанное вендорское железо и софт - требуют весьма ограниченное количество человеческих ресурсов на обслуживание.

В-третьих, это всё сейчас и ещё долгое время будет касаться ДатаЦентров со сравнительно простой сетью, построенной преимущественно на свитчах. В гораздо меньшей степени это применимо к энтерпрайзной сети. А уж когда это докатится до операторов нам будут рассказывать наши внуки. При том что NETCONF там тоже плюс-минус состоялся.

И всё же этот мир прекрасен.

6.10 Шаг за горизонт

А что если я вам теперь скажу, что мы делаем всё неправильно?

Мысль не свежая и принадлежит не мне.

А людям, которые в сети приходят из разработки или других смежных профессий.

У сети в целом очень простая задача - доставить данные из точки А в точку Б.

Концепция пакетной маршрутизации в основе своей понятна и красива - глядя на IP-адрес назначения, принимать решение на каждой новой точке. От конца до конца одна логика.

Но сразу же появляется необходимость в протоколах маршрутизации. И тут первый раз становится сложно: Тут - OSPF, там - BGP, где-то - статика. Ещё OSPF не всегда подходит - нужен какой-то IS-IS.

У каждого протокола триллион параметров: помним про virtual-links, MED, sham-links, DN bit, Local Preference, Overload bit, никому не нужный стек OSI.

Мы придумываем RPKI и болезненно его внедряем, потому что все сетевики в прошлом джентльмены - и мы безоговорочно доверяем друг другу.

А потом кончаются IPv4-адреса, потому что кто-то когда-то не подумал заранее. Ломается принцип end-to-end связности, появляется триста видов NAT, часть из которых призвана обеспечить хождение данных между IPv4 и IPv6 мирами.

А в маленьких сегментах сети, где находятся компьютеры, нужно ещё не забыть, что в Ethernet забыли заложить TTL и организовали широковещательные рассылки. Для этого придумали пяток несовместимых друг с другом протоколов, каждый из которых устраивал в своё время инцидент масштаба как минимум города, а то и региона.

Кстати, спасибо ребятам, в L2-сегменте мы не можем утилизировать все линки одновременно, поэтому придумываем ещё несколько протоколов защиты дефолтного шлюза, которые нет-нет да страдают от сплит-брейна.

Учитывая, что L2-пространство плоское, пришлось придумать сначала dot1q, потом, когда оказалось, что 4094 мало, qinq и разные виды маппингов и стекингов.

Всё это призвано решить только задачи транспорта. Но есть ещё сервисы. А там пересекающиеся IP-пространства, данные, которые нужно спрятать, или даже вовсе не IP, а Ethernet или

какой-нибудь IPX или ATM.

Появляются оверлеи. Просто перечислим их виды? IPinIP, GREinIP, BGP MPLS L3VPN, BGP MPLS L2VPN, VPLS Compella, VPLS Martini, MPLS over GRE, MPLS over UDP, VxLAN, несколько режимов работы IPSEC. И ещё не вспоминаем тыщу разных видов VPN, как L2TP, PPTP, SSTP, OpenVPN, SSH-туннели.

Каждый раз, строя сеть, пробуем разобраться, что из этого поддерживается какими вендорами и какими железками. А поддерживается только в Control Plane или в Data Plane тоже? А через CPU или через ASIC? А как настроить?

Как вы знаете, недостаточно положить IP в GRE и снова в IP - мы используем MPLS. Технология, нашедшая себя в совершенно не в той сфере, в которую её сватали.

Идея в Data Plane выглядит ещё более или менее понятной, пока мы не касаемся FRR, bandwidth allocation и Affinity.

Но тут у нас есть LDP в разных режимах, RSVP-TE, remote LDP, BGP - всё только чтобы распространять метки. И это рождает монстров вроде LDPoverTE, MPLS TE Hotstandby с Fast Reroute и ручного назначения и слежения за bandwidth.

Кому-то ведь даже приходится хорошо помнить (на собесах, например), сдвиги октетов, в которых находится RRO объект в RSVP-TE Path message.

BGP вот, кстати, стал козлом отпущения, сам того не желая (как будто это бывает иначе). В его NLRI насовали всего, что только могли - BGP всё стерпит. Отсюда же у нас и в некоторых случаях отсутствие обязательного атрибута NextHop.

Ещё сетям свойственно перегружаться и ронять пакеты на пол. С такими потерями имеют дело дюжина механизмов Congestion Avoidance и ещё столько же Congestion Management. Tail-drop, RED, WRED, RR, WRR, PQ, CBWFQ, ECN, CWR. Мы знаем, что такое HoLB, VoQ, Pause Frames, PFC, CAM, TCAM, SerDes, Traffic Manager - и как они все поступают с пакетами. И вообще мы эксперты по аббревиатурам - найдите, в какой профессии их больше - устроим батл!

И давайте не будем трогать мультикаст?

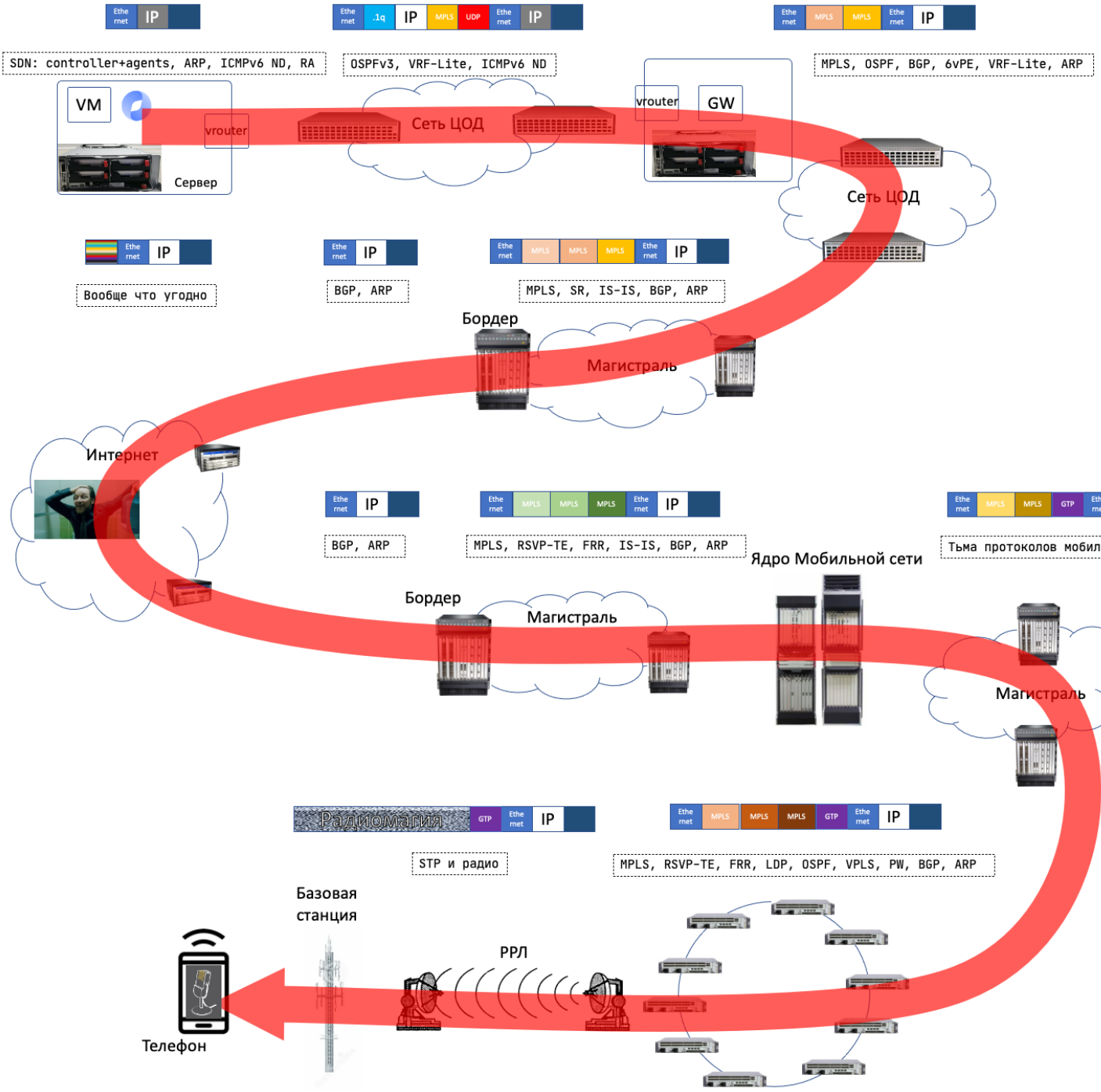
В итоге как может выглядеть жизнь одного байта, который вы отправили из виртуальной машины в облаке клиенту с телефоном?

Выходя из виртуалки, он содержится внутри TCP/IP/Ethernet - в качестве адреса назначения - клиентский IPv4-адрес. Далее его перехватывает виртуальный рутер (vRouter) на хосте-гипервизоре, снимает Ethernet, навешивает на него метку MPLS потом UDP, и ещё один IP-заголовок - теперь IPv6 и новый Ethernet - и отправляет на свой Gateway. Но фактически пакет попадет на стоечный коммутатор, внутри которого создан отдельный VRF, далее он идёт по маршруту, изученному по OSPFv3, перескакивая с хопа на хоп через выделенные для него VLAN'ы внутри одного физического линка. Пока не достигнет последнего коммутатора, который должен из своих таблиц ND и MAC-адресов извлечь в какой порт и в какой VLAN отправить этот пакет с какими MAC'ами.

На хосте его перехватит vRouter, снимет заголовки Ethernet, IP, UDP и изначальный пакет доставит до Gateway, который суть другая виртуальная машина. Уже этот Gateway знает, что для 0/0 у него некстхоп - это адрес бордера, но нужно упаковать данные в MPLS. Эта информация изучена по BGP.

Gateway навешивает сервисную метку MPLS и ещё сверху транспортную и отправляет в свитч.

Цепочка свитчей выполняет MPLS-коммутацию, на основе информации, изученной с помощью BGP Labeled Unicast, и доносит пакет до MPLS-магистрали, где существующий пакет упакуется в ещё


SDN: controller+agents, ARP, ICMPv6 ND, RA
OSPFv3, VRF-Lite, ICMPv6 ND
MPLS, OSPF, BGP, 6vPE, VRF-Lite, ARP
VM
vrouter
Сервер
Сеть ЦОД
vrouter
GW
Сеть ЦОД
Вообще что угодно
BGP, ARP
MPLS, SR, IS-IS, BGP, ARP
Бордер
Магистраль
Интернет
BGP, ARP
MPLS, RSVP-TE, FRR, IS-IS, BGP, ARP
Тьма протоколов мобил
Ядро Мобильной сети
Бордер
Магистраль
Магистраль
Радиомагия
STP и радио
MPLS, RSVP-TE, FRR, LDP, OSPF, VPLS, PW, BGP, ARP
Базовая станция
РРЛ
Телефон

несколько заголовков MPLS, которые должен донести пакет через магистраль до бордера. Информация, о том, какие именно MPLS-метки проставить и как с ними дальше быть появляется из Segment Routing, работающего поверх IS-IS.

Далее на магистрали за одно устройство до точки выхода снимаются все MPLS-метки, потому что PHP.

Бордер шлёт пакет в одного из своих транзитов. Далее побитый судьбой, пройдя через множество оптических каналов, маршрутизирующих систем, обросший неизвестным количеством меток он приходит на бордер оператора связи, в котором находится клиентский терминал.

Бордер теперь должен доставить пакет до ядра мобильной сети в том регионе, где находится клиент. Через магистраль оператора связи. На этот раз она поддерживается механизмом RSVP-TE. Тут у оператора могут быть десятки разных каналов, которые управляются Affinity, Bandwidth constraints и даже Explicit routes.

Пакет обрастает одной, двумя или тремя метками MPLS и отправляется в долгий путь. А по пути случается обрыв трассы между городами, срабатывает Fast Reroute и пакет обзаводится ещё одной MPLS-меткой.

Так или иначе он добирается до ядра сети в регионе, где принимается решение о том, как доставить его до базовой станции. Чтобы это случилось, пакет сначала весь раздевается почти до изначального состояния, а потом должен обратно обрасти новыми заголовками, один из которых ключевой для мобильных сетей - это GTP - ещё один туннель. В плане опорной сети у операторов тут тоже без фантазий - оптическое кольцо по области, сверху которого накручены OSPF или IS-IS, MPLS как транспорт или BGP L3VPN для выделения сервиса мобильного интернета в отдельный VRF. Пакет попадает в это кольцо, направляется нагруженный метками к точке назначения. И тут рвётся один из линков, размыкая кольцо. Срабатывает снова FRR, который перенаправляет трафик к тому же некстхопу, но через противоположное направление по кольцу, опять же навешав MPLS-метку. И доходит.

Но на этом жизнь его не кончается. В этом месте от кольца начинается цепочка старинных радиорелеек, не поддерживающая MPLS, которая ведёт к глухой деревне. Скидываются все метки, пакет идёт по проброшенному VLAN'у. А чтобы не случилось петли коммутации на них включен STP.

А на одном из сегментов для организации канала используется E1, где нужно Ethernet конвертировать в E1 тайм-слоты, а потом обратно расконвертировать.

И вот базовая станция клиента, где пакет, полученный из одной радиосреды, нужно передать в другой - в LTE, при этом не забыв все его заголовки и радио-маркеры привести в требуемый вид.

После этого терминал наконец получает пакет, потрёпанный, поцарапанный, побитый, раздавленный, и ... дропает его, потому что истекло время ожидания.

А сколько радости доставляет каждому сетевику и разработчику вендора интероп разных вендоров?

И всё это только потому, что сеть - это самый фундамент инфраструктуры. Нужно поддерживать как новейшие технологии, так и чудовищные ископаемые.

Новенькие облака должны смочь дотянуться до старинных Token Ring. Недовытравленные сети SDH должны жить одновременно с 400Гб/с Ethernet. Серые IPv4 сети должны иметь возможность добраться до IPv6-only сервисов.

Сетевые устройства и технологии вынуждены тащить за свой хвост легаси длинной в 60 лет, продолжая наращивать его всё больше и больше. И это всё лишь для того, чтобы доставить байты из точки А в точку Б.

Расскажите сетевикам про проклятое легаси и обратную совместимость?

Самое обидное, что тащить его через ворох такого старья приходится не только глобально по миру, но и внутри вполне себе контролируемого контура. Мы можем выдумывать какие угодно изящные схемы, но это не имеет смысла, если их нельзя реализовать в железе.

И тут проприетарный вендорский софт всячески вставляет палки в колёса.

А вот программируемые конвейеры обещают небывалую гибкость. Whitebox и P4 могут стать теми, кто поменяет правила игры. Если у вас есть десятки коробок с запущенным Linux, и вы можете программировать обработчики в чипе, то перед вами открываются совершенно иные возможности.

Центральный контроллер, принимающий решение о том, как направить трафик в магистрали, учитывая утилизацию, как нагружать внешние линки, программируя нужные инкапсуляции на источнике - это ли не мечта?

И какой-нибудь протокол, как Segment Routing v6, может тут сыграть свою достойную роль. В пределах своего домена используем контроллер и SRv6 для транспорта трафика, на конечных узлах с помощью P4 программируем обработку инкапсуляций на мощном и гибком ASIC'е, а Control Plane сгружаем на обычные сервера. И больше не нужны гигантские вендорские мангалы, не нужно платить за 80% функциональности, которой так никогда и не воспользуешься, этот проприетарный софт, и бесконечные баги.

А потом вы начинаете задавать себе и другим вопросы, вроде - а нужен ли Ethernet?

6.11 Полезные ссылки

- **Главные RFC:**
  - RFC1052: Рекомендации IAB по инструментам управления сетью. Результаты встречи (Апрель 1988)
  - RFC3535: Результаты встречи IAB по инструментам управления сетью (май 2003)
  - NETCONF: RFC4741, RFC6241
  - RESTCONF: RFC8040
  - YANG: RFC6020, RFC7950
- Репозиторий с опубликованными YANG-моделями
- Слайды NANOG 2015
- Слайды IETF 98
- Слайды IETF 101
- SDK Broadcom на github
- Концепция RPC: Remote Procedure Call (RPC)
- Сайт OpenConfig: openconfig.net
- Опубликованные модели OpenConfig
- Хорошая вводная в NETCONF и немного YANG: YANG Data Modelling and NETCONF
- Продолжение про YANG и немного про NETCONF: The Road to Model Driven Programmability
- На русском про NETCONF. Начало

- Если жить не можете, хочется на русском про YANG, и вы воспринимаете художественную речь: YANG — это имя для вождя
- Спецификация gNMI в его же репозитории
- RESTCONF Tutorial - Everything you need to know about RESTCONF in 2020
- Серия статей про RESTCONF, но рекомендую я её из-за того, что там хорошо разобраны примеры с YANG-моделями: RESTCONF, NETCONF and YANG
- Документация по gRPC
- Блог Романа Додина
- Блог Антона Карнелюка
- Блог Михаила Кашина
- Яндекс Nextop. Про использование CLI для Configuration Management

6.12 Заключение

Конечно, я очень размазал последнюю часть, непонятно как это сделать, непонятно даже что я имел в виду, возможно. Но это совершенно удивительный мир не просто автоматизации сети, а новой парадигмы строительства и управления сетью.
Я думаю, что об этом я напишу ещё не одну статью.

И, надеюсь, этой статьёй мне удалось немного раскрыть длинную историю сетевой автоматизации, показать, что мы живём в эпоху Кембрийского взрыва инструментов и интерфейсов. И перед нами сейчас открыты разные пути, каждый из которых сулит как минимум интересное развитие событий.

Рекомендую к прочтению шестую часть АДСМ.

6.12.1 Благодарности

- Роману Додину за дельные комментарии как по теоретической, так и по практической частям. А так же за полезный блог и инструменты. GitHub.
- Кириллу Плетнёву за наведение порядка с NETCONF и YANG - язык, модели, спецификации, форматы данных. И за уместные и остроумные замечания по языкам и библиотекам. GitHub, fb.
- Александру Лимонову за несколько идеологических замечаний и исправлений фактических ошибок.

Глава 7

Часть 6. Интерфейсы взаимодействия с сетевым устройством

В этой части мы раскрываем дерево XML, пробуем на вкус капабилити NETCONF, шлём первые RPC и наконец уже расставим в правильном порядке буквы YANG, OpenConfig, gNMI.

Практическую пользу вам принесут только обе прочитанные статьи. Вторая без первой будет непонятна. Первая без второй - беллетристика.

Сразу предупреждаю, что это будет большая и нудная статья, потому что автор в очередной раз решил разобраться в чём-то, и опубликовать это разом. И вам, клянусь, ещё повезло, когда на двухсоттысячном символе я придумал, как её можно разделить на две части.

Тут разберём по косточкам все возможные способы взаимодействия с сетевым железом.

Лишь вскользь мы заденем CLI и SNMP, как не имеющие практической значимости в контексте этой статьи, разберёмся достаточно глубоко с NETCONF - это новый SNMP или всё же у него есть будущее хотя бы с YANG'ом, продолжим RESTCONF'ом и закончим на интригующем - gRPC.

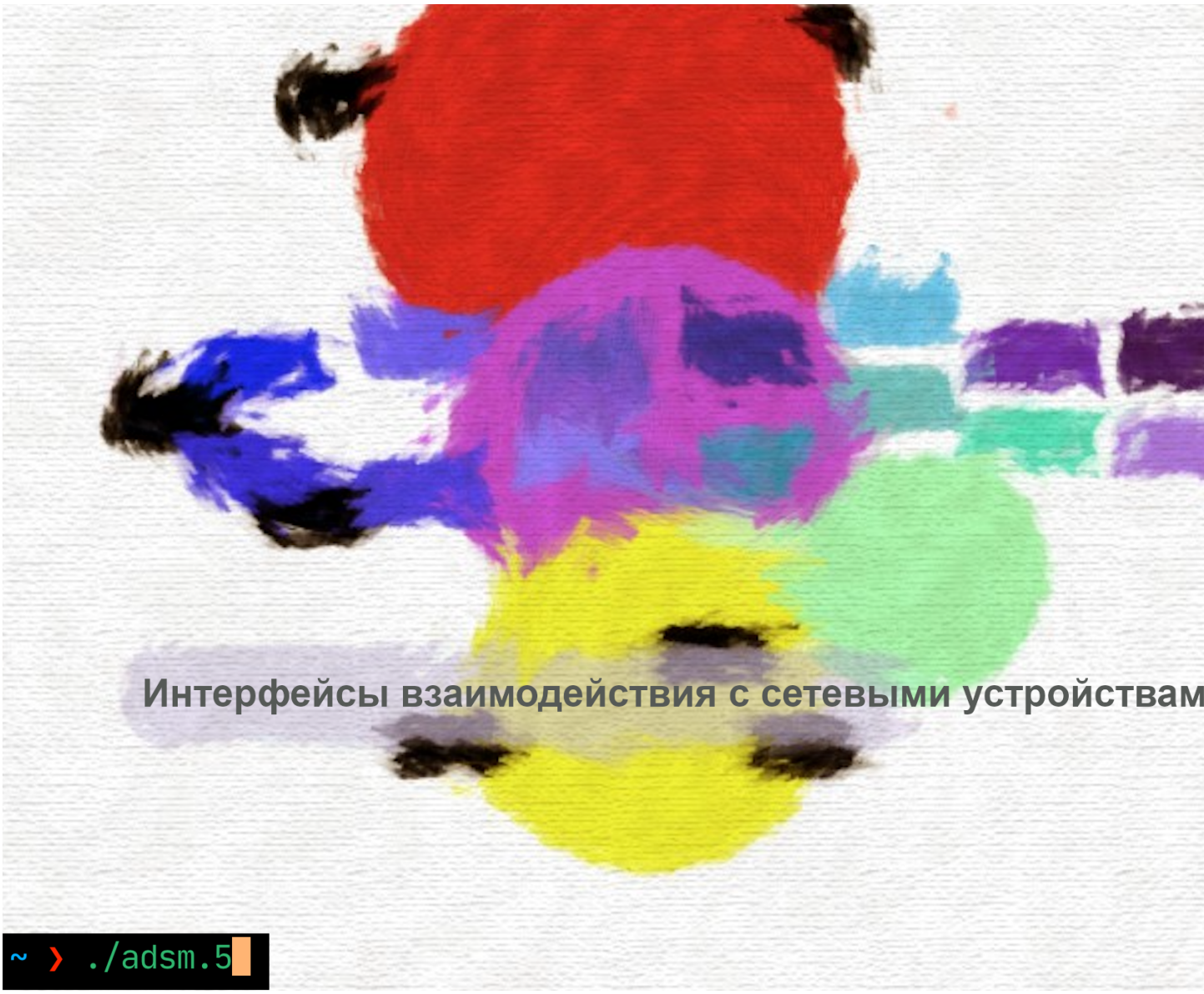
Ну а по ходу неминуемо разберёмся с тем, за что с нашими глазами так поступает XML, с концепцией RPC, моделями данных и успеем посмотреть на OpenConfig.

7.1 CLI - Command Line Interface

CLI - *сиэлай, кли, сли, слай, слаи, консоль, терминал, командная строка.* Этому механизму уже лет 60. И он никуда не делся. Он живее всех живых - где-то для отладки, где-то для эксплуатации, зачастую для конфигурации и даже для ежедневной работы.

На компьютерах, серверах, виртуальных машинах, коммутаторах, маршрутизаторах, фаерволах, АТС, базовых станциях. Трудно найти такое оборудование, где нет CLI, пусть даже хорошо спрятанного.

И в этом его сила - 100% функциональности на 100% сетевых устройств можно настроить через CLI. Ладно 99,9% - придётся выкинуть некоторое альтернативное оборудование.


Интерфейсы взаимодействия с сетевыми устройствам
~ ❯ ./adsm.5

Это породило миллионы строк кода на Perl, PHP, Python, Go, Ruby, развесистые джинджа-шаблоны и по 300 экспектов в каждом скрипте.

И дало работу тысячам кодеров, выросших из сетевиков и админов.

Вот уже лет 30, а то и больше мы старательно пишем скрипты, которые с той или иной степенью успеха прикидываются человеком перед сетевой коробкой.

И ещё долго мы не останемся без дела - выпускают всё новые версии софта, ещё более другие модели железа, постоянно меняется CLI, и там, где вчера был string, завтра будет integer. И там, где вчера было `no some shitty service enable`, завтра будет `some shitty service disable`. И там, где вчера на вопрос интерфейса надо было ответить `yes`, завтра вылезет ошибка.

Клянусь, это увлекательное путешествие продлится ещё десятилетия.

А чем же оно увлекательно?

1. Модели конфигурации не формализованы
2. Модель и поведение не зафиксированы
3. CLI интерактивен
4. Формат данных не структурированный
5. Нет явного признака успешности операции
6. Сложно вычислять разницу между целевой и текущей конфигурацией
7. Сложно считать конфигурационный патч
8. Транзакционность не всегда доступна
9. Поддержание целевого состояния – задача инженера

Выше 9 смертных грехов CLI, которые обусловили рождение моделей данных конфигурации, языков их описания, протоколов, как SNMP, NETCONF и gNMI.

Если всё по каждому из них понятно, просто пропускайте следующую секцию.

7.1.1 9 грехов CLI

1. Модели конфигурации не формализованы

Есть такое? Есть такое.

Как мы узнаём, какие команды с какими аргументами в каком порядке надо дать?

Правильно - идём в Command reference guide на сайте производителя и дальше методом проб и ошибок разбираемся в терминале. Или в обратном порядке. Но эти два способа (и ещё помощь друга) - это то, как мы узнаём модель данных конфигурации.

И скажем так: она наверняка есть - ведь каждый раз одна и та же команда приводит к одному и тому же результату (правда ведь? Правда?)

Знаем как настраивать интерфейсы, знаем как они должны называться, где будет IPv4, а где IPv6 адреса? Если мы введём что-то неправильно, CLI ругнётся, но мы не отправим OS в kernel panic или ASIC в рестарт?

Просто эта модель не формализована. Или по крайней мере нам об этом не говорят.

И да, своим естественным интеллектом мы рано или поздно такую модель в своей голове выстраиваем и научаемся ею пользоваться.
Но для того, чтобы написать код, нам придётся её хорошо или плохо самим формализовать - иерархия, порядок ввода команд, последовательность аргументов, типы значений. И скорее плохо, конечно.

2. Модель и поведение не зафиксированы

Всё, что мы изучили на предыдущем этапе, может поменяться в новой версии - и мы сначала переобучаем себя, потом переписываем код.

3. CLI интерактивен

```
expect("Вы точно хотите выключить bgp-сессию, mpls на всей коробке [Y/n]?"] Yes!
expect("Вы точно хотите выключить электричество в серверной [Y/n]?"] No!
```

4. Формат данных не структурированный

Мы засылаем туда неструктурированный текст.
Мы получаем оттуда неструктурированный текст.
Мы его крафтим, мы его парсим. Мы пишем jinja-шаблоны и regexp'ы. Мы прожигаем свою жизнь.
Лучше json'ы перекладывать.

> Строго говоря, будь-то json или вывод `show version`, в итоге это всё равно поток байтов и по сути текст. Только в одном случае в нём есть структура, а в другом - это просто набор символов.

5. Нет явного признака успешности операции

Вывод CLI не означает ни успех, ни провал.
Warning ещё не означает, что что-то пошло не так.
Отсутствие вывода - ещё не признак успешности.

6. Сложно вычислять разницу между целевой и текущей конфигурацией

Казалось бы нужно просто подифать два текста.
Вот только в этих текстах имеет значение, порядок строк.
Одна и та же конфигурация может быть (и будет) разной для разных версий ПО.
Различающийся же регистр, напротив, может ничего не значить в одном месте, а в другом значить.
Одна и та же команда, в разных контекстах может означать разное (выключите MPLS на интерфейсе или в глобальном режиме - посмотрите на последствия).
Даже IPv6-адреса могут быть записаны в сокращённом или полном виде.

7. Сложно считать конфигурационный патч

Как следствие предыдущего пункта - выяснить, какие команды нужно применить - тоже нетривиально.

Но не только это.

Дело в том, что нужно уметь не только правильно добавлять, но и правильно удалять - а способов - не один и не два. Обратная команда не всегда формируется как отрицание прямой. Часто нужны не все её параметры.

В каком порядке отменять - и не поломает ли это чего-то ещё?

Даже не всегда команды после применения выглядят так же, как их применяли.

8. Транзакционность не всегда доступна

В целом мы уже избалованы коммитами - многие вендоры его поддерживают. Но многие ещё нет. А те, кто поддерживает, может это делать тоже собственным уникальным способом, как например валидация ввода только при коммите, или коммит заключается в последовательном применении всех команд без вычисления дельты.

9. Поддержание целевого состояния – задача инженера

Ооо, это самое интересное - а как собственно привести конфигурацию к тому состоянию, которое мы желаем увидеть, а не просто применить новую конфигу?

Кто и как должен посчитать конфигурационный патч, применить только его и проверить за собой, что рантайм соответствует эталону?

> Но тут стоит быть чуть более честным - не всегда CLI настолько плох. Некоторые вендоры генерируют CLI-интерфейс из YANG-модели, что гарантирует чёткое соответствие между тем, что и как конфигурируется через CLI или любые другие интерфейсы.
>
> Например, в Nokia SR Linux интерфейс командной строки, а так же gNMI, JSON-RPC и внутренние приложения работают с единым API - *mgmt_srv* - поэтому не только формализованы из одной и той же YANG-модели, но и имеют одинаковые возможности по чтению/записи конфигурации.
>
> Дифы, коммиты, датасторы и прочее, тоже могут быть сделаны с умом - как у той же Nokia или у Juniper.
>
> Но это всё, конечно, не отменяет факта работы с неструктурированным текстом.

Этого всего, как мне кажется, достаточно для того, чтобы даже не приступать к написанию полноценной системы автоматизации, основанной на CLI.

Далее был опыт с SNMP и всеми связанными протоколами (приглашаю пройти в пятую часть книги).

Признаем его удачным лишь по той простой причине, что он позволил сформулировать требования к новым интерфейсам и протоколам.

Не исчерпывающий список можно сформулировать так:

- Представление данных в структурированном виде,
- Разделение конфигурационных и операционных данных,
- Читаемость для человека исходных данных и самой конфигурации,
- Воспроизводимость - задачу на исходных данных можно запустить повторно - проиграть,

- Механизм основан на формальных моделях,
- Транзакционность изменений и их откат,
- Поддержание целевого состояния.

Не все они появились сразу. Не все они появились. Но это понятная и приятная цель.

И на замену SNMP, в подмогу CLI зародился NETCONF, эксплуатирующий идею RPC - Remote Procedure Call.
Что за RPC, какое у него отношение с API вы так же можете узнать из пятой части.
Ну, только если коротко.

7.2 Концепция RPC - Russian Pravoslavnaya Church

RPC - клиент-серверный механизм, который позволяет запустить исполнение кода процедуры на другой машине так, словно бы он исполнялся локально. То есть разработчик просто привычным образом обращается к процедуре, не задумываясь о том, где и как она исполняется - главное, чтобы она ответ вернула.
А программа уже сама реализует взаимодействие с удалённой машиной.
Прелесть этого подхода в том, что он, во-первых, позволяет скрыть удалённый характер работы. А, во-вторых, на той, другой, стороне совершенно неважно, какая операционная система, архитектура, язык программирования и окружение - главное, чтобы они подчинялись одному протоколу.
Например из-под винды в exe-шнике, написанном на Delphi, вы можете исполнить удалённую программу, написанную на го, запущенную на линуксе. И никто вам не сможет помешать!

Но что, по большому счёту, мы делаем, когда, зайдя по SSH, выполняем какую-то команду на коммутаторе или маршрутизаторе? Запускаем определённый код.
Например, сообщаем подсистеме BGP, что нужно теперь пробовать установить соединение с новым пиром.

Но только представьте, как было бы восхитительно, если бы для вызова этого кода, не нужно было заходить на железку по SSH и вбивать команду?!

Постойте! Да ведь именно об этом мы и говорим в данном разделе. | Большую оставшуюся часть статьи мы посвятим именно RPC.

7.2.1 Пример

Абстрактно взаимодействие с сетевым оборудованием выглядеть может примерно так.
Для начала мы определяем спецификацию - это некий контракт, который гарантирует, что у клиента и сервера одинаковое понимание процедуры: имя, параметры, типы данных итд.

1. Наша убер-платформа автоматизации вызывает некую функцию `add_bgp_peer_stub(ip="10.1.1.1", as="12345")`.
2. Функция `add_bgp_peer_stub` открывает спецификацию для протокола, реализующего RPC, и согласно ей упаковывает полученные параметры, которые станут payload'ом для сообщения. Такая упаковка называется **маршалинг**.

3. Далее формирует пакет и передаёт его вниз по стеку и - в сеть.
4. На другой стороне - на устройстве - приложение получает пакета.
5. Функция, принявшая сообщение, вытаскивает из него параметры процедуры, согласно той же самой спецификации и формирует список параметров. Это называется **демаршалинг**.
6. Приложение выполняет функцию - настраивает BGP-соседа 10.1.1.1 с AS 12345. Проверяет успешность выполнения.
7. Далее функция формирует на основе всё той же спецификации сообщение-ответ и передаёт его в ответном пакете.
8. Наша локальная сторона, с которой мы инициировали выполнение RPC, получает ответ, словно бы его вернула локальная функция.
9. Воаля

Поподробнее про RPC.

В целом RPC - это концепция, не говорящая ничего о реализации.
Она постулирует, что на стороне клиента есть так называемый стаб (stub) - фрагмент кода, который реализует взаимодействие по RPC. Именно стабы делают для разработчика прозрачным вызов функции - из приложения вызывается этот стаб с набором параметров, а уже стаб делает удалённый вызов.
Ключевая часть RPC - спецификация - штука, которая на стороне сервера и клиента определяет, как работать с данными - как упаковать, как распаковать. Без участия человека, конечно же.
Язык, на котором пишется спецификация - **IDL** - *Interface Definition Language*.
Иными словами, на IDL пишется спецификация, на основе которой создаются и серверный интерфейс, и клиентский стаб. Это может быть, например, набор классов в питоне, имеющих функции для удалённого вызова, с которыми разработчик работает так, словно всё происходит локально - для клиента. И набор объектов Go - для сервера.

Мы дальше разберём два протокола, которые используются под капотом RPC и при этом позволяют управлять сетевым железом.

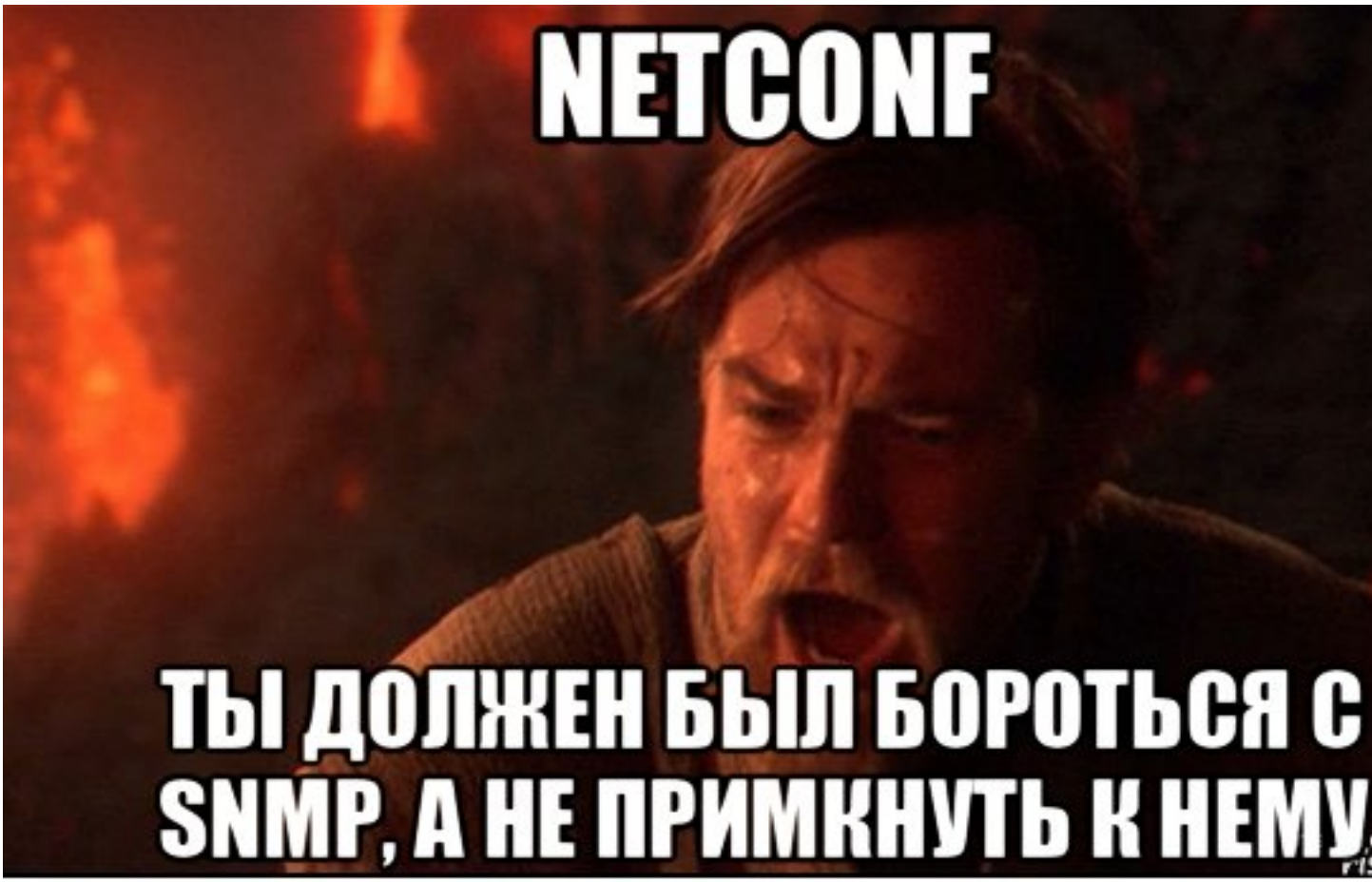
- NETCONF
- gNMI (использующий gRPC)

7.3 NETCONF

Ох, как я вился вокруг этого нетконфа в своё время, ожидая, что это серебряная пуля, решающая если не все, то 99,99% всех проблем сетевых инженеров.
Спойлер: это не так.

Если вам по какой-то причине кажется, что стандарты рождаются где-то в недрах институтов, оторванных от жизни, то вот вам контр-пример.

В 1996 был основан Juniper Networks, в недрах которого создали легендарный M40 и лучший в мире интерфейс командной строки. До сих пор никто не сделал ничего лучшего - все только повторяют.


NETCONF
ТЫ ДОЛЖЕН БЫЛ БОРОТЬСЯ С
SNMP, А НЕ ПРИМКНУТЬ К НЕМУ

Операционка, предоставляющая клиенту обычный текстовый интерфейс, на самом деле перекладывает команды в XML, который фактически является интерфейсом для управления устройством.

Если вы сейчас к любой show-команде на джуне добавите `| display xml`, то увидите ответ в формате XML

```
eucariot@kzn-spine-0> show system uptime | display xml
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.3R3/junos">
    <multi-routing-engine-results>
        <multi-routing-engine-item>
            <re-name>localre</re-name>
            <system-uptime-information xmlns="http://xml.juniper.net/junos/18.3R3/junos
↪">
                <current-time>
                    <date-time junos:seconds="1641211199">2022-01-03 14:59:59 MSK</date-
↪time>
                </current-time>
                <time-source> LOCAL CLOCK </time-source>
                <system-booted-time>
                    <date-time junos:seconds="1614866046">2021-03-04 16:54:06 MSK</date-
↪time>
                    <time-length junos:seconds="26345153">43w3d 22:05</time-length>
                </system-booted-time>
                <protocols-started-time>
                    <date-time junos:seconds="1614866101">2021-03-04 16:55:01 MSK</date-
↪time>
                    <time-length junos:seconds="26345098">43w3d 22:04</time-length>
                </protocols-started-time>
                <last-configured-time>
                    <date-time junos:seconds="1638893962">2021-12-07 19:19:22 MSK</date-
↪time>
                    <time-length junos:seconds="2317237">3w5d 19:40</time-length>
                    <user>scamp</user>
                </last-configured-time>
                <uptime-information>
                    <date-time junos:seconds="1641211200">3:00PM</date-time>
                    <up-time junos:seconds="26345160">304 days, 22:06</up-time>
                    <active-user-count junos:format="1 users">1</active-user-count>
                    <load-average-1>0.20</load-average-1>
                    <load-average-5>0.17</load-average-5>
                    <load-average-15>0.20</load-average-15>
                    <user-table></user-table>
                </uptime-information>
            </system-uptime-information>
        </multi-routing-engine-item>
    </multi-routing-engine-results>
    <cli>
        <banner>{master:0}</banner>
    </cli>
</rpc-reply>
```

В корне вы можете видеть `<rpc-reply>`, что означает, что был какой-то `<rpc>`-request. И вот так вы можете увидеть, каким RPC-запросом можно получить такие данные:

```
eucariot@kzn-spine-0> show version | display xml rpc
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/18.3R3/junos">
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    <rpc>
        <get-software-information>
        </get-software-information>
    </rpc>
    <cli>
        <banner>{master:0}</banner>
    </cli>
</rpc-reply>

*Внимание, работает только для Juniper!*
```

Так вот, их CLI и способ взаимодействия его с системой оказался настолько естественным и удачным, что его и положили в основу стандарта. Не без участия Juniper Networks, конечно же, появился RFC4741. Будем честны, один только джунипер там и постарался. И то тут, то там будут проскакивать его куски, начиная с `commit confirmed` и заканчивая `candidate config`.

Вот как NETCONF был определён в 2006-м году:

```
Abstract
The Network Configuration Protocol (NETCONF) defined in this document
provides mechanisms to install, manipulate, and delete the
configuration of network devices.  It uses an Extensible Markup
Language (XML)-based data encoding for the configuration data as well
as the protocol messages.  The NETCONF protocol operations are
realized on top of a simple Remote Procedure Call (RPC) layer.
```

И определение с тех пор не менялось - вся суть NETCONF в этом параграфе.

А теперь давайте разбираться с очень непростым NETCONF и его составными частями.

7.3.1 NETCONF и его команды

Если совсем коротко, NETCONF - это четырёхуровневый стек, согласно которому через SSH передаётся RPC, где указана операция и конкретный набор действий (контент).

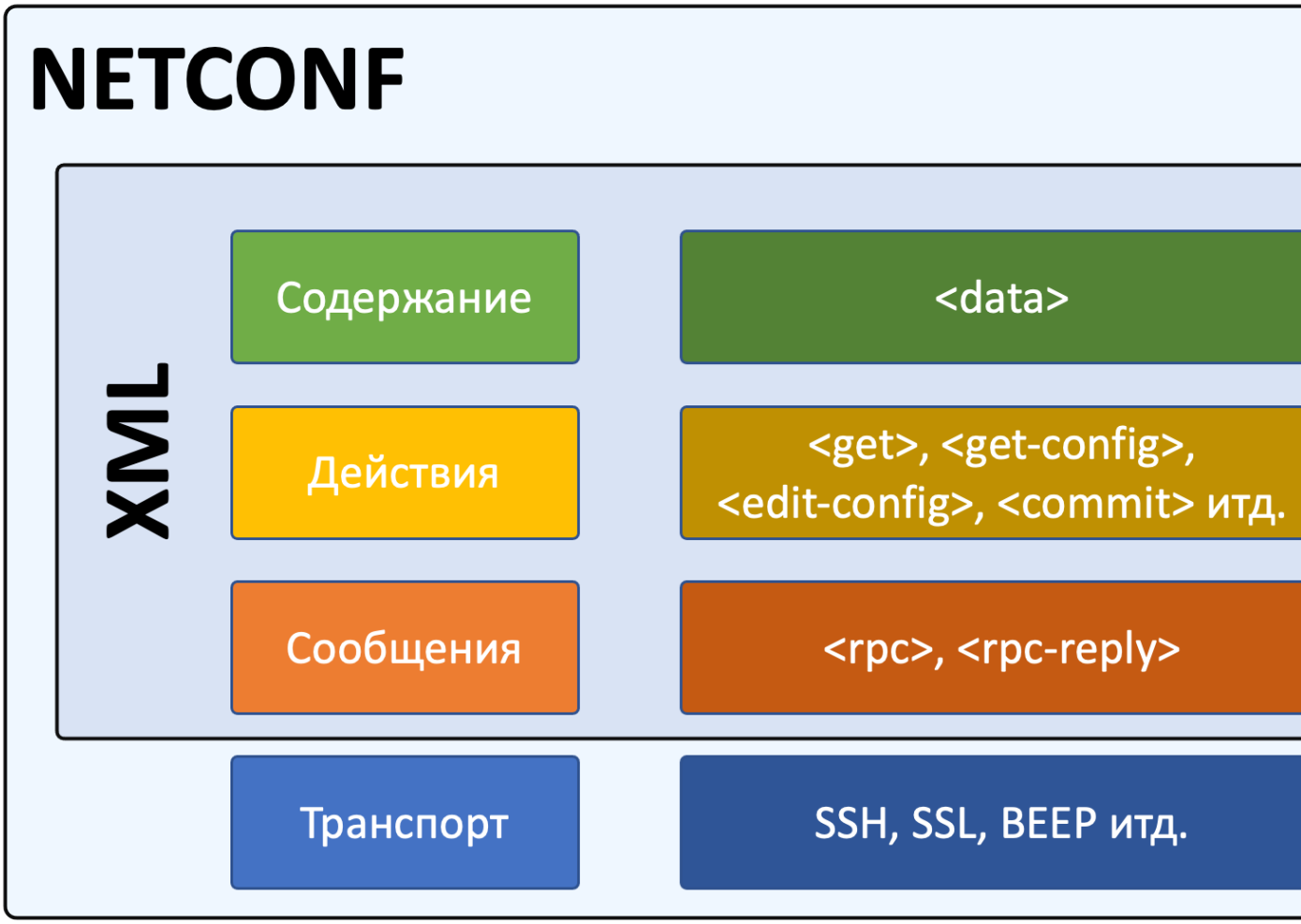
Стек NETCONF

Итак, в качестве транспорта NETCONF использует SSH. На самом деле, там есть и другие протоколы: SSH, SOAP, BEEP, TLS - но мы их опустим - SSH стал де-факто стандартом.

Каждый NETCONF запрос содержит элемент (или сообщение):

- `<rpc>` - это собственно запрос на вызов процедуры с необходимыми параметрами.
- `<rpc-reply>` - ответ на RPC.
  - `<rpc-error>` - очевидно, ответная ошибка, когда RPC некорректен.
  - `<ok>` - rpc корректен и отработал.
- `<notification>` - сообщение о событии, инициированное сетевой коробкой - аналог трапа в snmp. (из RFC6241)

Это всё сообщения, внутри которых определённым образом сформированные XML.


NETCONF
XML
Содержание
<data>
Действия
<get>, <get-config>,
<edit-config>, <commit> итд.
Сообщения
<rpc>, <rpc-reply>
Транспорт
SSH, SSL, BEEP итд.

Внутри сообщения определяется какая операция (действие) исполняется.
Ниже полный их список, определённый в RFC:

- `<get>` - retrieve running configuration and device state information
- `<get-config>` - retrieve all or part of a specified configuration datastore
- `<edit-config>` - edit a configuration datastore by creating, deleting, merging or replacing content
- `<copy-config>` - copy an entire configuration datastore to another configuration datastore
- `<delete-config>` - delete a configuration datastore
- `<lock>` - lock an entire configuration datastore of a device
- `<unlock>` - release a configuration datastore lock previously obtained with the <lock> operation
- `<close-session>` - request graceful termination of a netconf session
- `<kill-session>` - force the termination of a netconf session

Каждый вендор может расширять список операций хоть до бесконечности. Так, у кого-то, например, есть `<copy-config>`.

И далее уже сам контент. Это самая сложная часть.
Но забегая вперёд - он никак не формализован, не описан, и, возможно, это величайшая претензия к нетконф, как стандарту, позволившему благую идею превратить в очередного зомби.
Даже удивительно, что после опыта с SNMP, где необходимость языка моделирования стала очевидна со временем, NETCONF родился сам по себе без какого-либо языка спецификации для данных. Уже много позже для этого подтянули YANG.

Установка сессии и Capabilities

Так, сначала включаем SSH NETCONF. На примере джунипер.

```
set system services netconf
```

> Это значит, что SSH будет использоваться как транспорт для указанной подсистемы.
> Для netconf IANA установила специальный порт 830, хотя часто используется и обычный для SSH 22.

И пробуем подключиться.
Для того, чтобы указать, что это не просто подключение по SSH, мы используем вызов подсистемы:

```
ssh kazan-spine-0.juniper -s netconf

<!-- No zombies were killed during the creation of this user interface -->
<!-- user eucariot, class j-super-user -->
<hello xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <capabilities>
    <capability>urn:ietf:params:netconf:base:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:candidate:1.0</capability>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    <capability>urn:ietf:params:netconf:capability:confirmed-commit:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:validate:1.0</capability>
    <capability>urn:ietf:params:netconf:capability:url:1.0?scheme=http,ftp,file</
↪capability>
    <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0</
↪capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0</capability>
    <capability>urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file
↪</capability>
    <capability>http://xml.juniper.net/netconf/junos/1.0</capability>
    <capability>http://xml.juniper.net/dmi/system/1.0</capability>
  </capabilities>
  <session-id>15420</session-id>
</hello>
]]>]]>
```

Мы ещё ничего не успели сделать, а железка нам уже насыпала в терминал.

Это сообщение NETCONF Hello, которое заставляет на берегу договориться, что поддерживается в данной сессии, а что нет.

Внутри - список капабилитей - возможностей, поддерживаемых коробкой.

RFC4741 определял базовый набор функций, который должен поддерживаться каждым клиентом и каждым сервером.

При этом базовые могут расширяться другими стандартизированными capability и даже проприетарными. Давайте рассмотрим сначала стандартные, а потом самые интересные расширенные. Ну и будем называть их «способностями», а то капабилитя - это почти как капибара.

NETCONF Standard Capabilities (стандартные способности)

- **Candidate configuration** Эта способность говорит о том. Что коробка поддерживает отдельный кандидат-конфиг, содержащий полную конфигурацию, с которой можно работать без влияния на фактически применённую конфигурацию. Аналоги candidate-config на Juniper.
- **Confirmed commit** Опять же аналог джуниперовоского commit confirmed - откат изменений после коммита, если не было подтверждения коммита.
- **Validate** Способность проверить желаемую конфигурацию до её применения.
- **Rollback-on-error** Способность отмены изменений при ошибке. Работает, если поддерживается способность candidate configuration.
- **Writable-running** Такая способность говорит о том, что устройство позволяет писать непосредственно в running-конфигурацию, в обхода candidate.
- **Distinct startup** Способность задавать startup конфигурацию отличную от running и candidate.
- **Notification** Аналог SNMP-trap. Коробка может слать аварии и события клиенту.

И ещё несколько более других способностей, которыми грузить вас не хочу, ибо в лучшем виде они описаны в RFC.

Посмотрите, кстати, какие способности отдал джунипер, а какие нет.

NETCONF Extended Capabilities (сверх-способности)

Их тьма. Из самых интересных:

- **YANG push** Способность отсылать данные с коробки на клиент - периодически или по событию.
- **YANG-library** Способность сервера сообщить клиенту о поддерживаемых параметрах относительно YANG: версия, модель, нейспейсы итд.
- **Commit-description** Самоговорящее название.

Формат названия capability строго регламентирован:
`urn:ietf:params:netconf:capability:{name}:1.0`.
Последние два значения - это имя и версия - и только они могут меняться.
Так `urn:ietf:params:netconf:base:1.1` - это имя базовой капабилити для версии 1.1.

В ответ на `<hello>` сервера клиент в свою очередь должен послать свои capability:

```
<hello>
 <capabilities>
  <capability>urn:ietf:params:xml:ns:netconf:base:1.0</capability>
  <capability>urn:ietf:params:xml:ns:netconf:capability:candidate:1.0</capability>
  <capability>urn:ietf:params:xml:ns:netconf:capability:confirmed-commit:1.0</
↪capability>
  <capability>urn:ietf:params:xml:ns:netconf:capability:validate:1.0</capability>
  <capability>urn:ietf:params:xml:ns:netconf:capability:url:1.0?protocol=http,ftp,file</
↪capability>
  <capability>xml.juniper.net/netconf/junos/1.0</capability>
  <capability>xml.juniper.net/dmi/system/1.0</capability>
 </capabilities>
</hello>
]]>]]>
```

Чего почти нигде не пишут, но что очень важно: если вы пробуете взаимодействовать с коробкой по нетконф руками, то нужно обязательно вручную отослать такую последовательность `]]>]]>`, сообщающую, что ввод закончен. Она называется **Framing Marker** или **Message Separator Sequence**.

> Есть важный нюанс, описанный в RFC6242, `]]>]]>` - это старый **End-of-Message Framing Marker**, который был выбран из соображений, что такая последовательность не должна встречаться в well-formed XML. Однако жизнь показала, что она встречается. Поэтому в NETCONF 1.1 придумали новый механизм, который делит данные на блоки - чанки - и нумерует их. Так он и называется: **Chunked Framing Mechanism**.
>
> Каждый чанк данных начинается с `##X`, где `X` - это число октетов в нём.
>
> Это одно из фундаментальных отличий между 1.0 и 1.1 :). Другие менее значительны.

Сейчас NETCONF-сессия установлена и можно заслать какой-то RPC.

Посылаем свой первый RPC

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
   <source>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    <running/>
   </source>
   <filter type="subtree">
     <configuration>
       <system>
          <host-name/>
       </system>
     </configuration>
   </filter>
  </get-config>
</rpc>
]]>]]>
```

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos" message-id="100"␣
↪xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
   <configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:commit-seconds=
↪"1644510087" junos:commit-localtime="2022-02-10 16:21:27 UTC" junos:commit-user=
↪"eucariot">
    <system>
        <host-name>kzn-spine-0</host-name>
    </system>
</configuration>
</data>
</rpc-reply>
```

Мы отправили элемент `<rpc>`, в котором запросили `<running>`-конфигурацию с помощью операцию `<get-config>`. И ещё на сервере отфильтровали по интересной ветке.

А в ответ пришёл `<rpc-reply>` с ответом. И в запросе, и в ответе можете найти `message-id` - по ним можно отслеживать на что именно ответ - ведь режим работы NETCONF асинхронный и можно засылать следующее сообщение, пока предыдущее ещё не было обработано.

Здесь вы видите некоторую структуру XML. Её легко можно скормить XML-парсеру, который превратит его в JSON или python dict или что угодно другое, с чем удобно работать в скриптах и программах. И далее извлечь по ключам нужные данные.
Но почему XML? За что? Как вообще с этим быть?

Ох. Зря вы спросили.
В общем дальше 10 000 знаков про XML. Если вы не готовы это выдержать, милости прошу дальше. Но будьте готовы, что практика NETCONF тогда пройдёт мимо вас. Или вы мимо неё. В общем разминётесь.

Так за что же так с нами?

7.4 <XML>

По всей видимости наиболее точный и честный ответ на вопрос «за что же так с нами поступает XML» - «исторически сложилось».

Судьба XML в чём-то похожа на MPLS - оба были созданы для одной задачи, а популярность снискали в другой.

XML намеревался стать метаязыком для создания языков разметки документов. Но очень быстро его адаптировали под формат сериализации данных при передаче. И к моменту, когда Juniper выбирал формат, в котором API будет принимать запросы, XML стал уже проверенным, зрелым кандидатом.

Сегодня, вероятно, победил бы JSON, но тогда он только начинал свой путь к славе.

XML

json

YAML и protobuf тогда ещё не существовали. Ну и вообще YAML подходит лучше для описания конфигураций, которые редактируются руками, нежели как формат обмена данными.

Прелюбопытная историческая справка по XML, JSON и YAML: YAML: The Missing Battery in Python.

В общем выбор в те дни был предопределён - XML был сверхсовременным и суперудобным,

Сложность читаемости XML компенсируется простотой его программной обработки. Чёткая иерар-

хическая структура, понятные начало, конец и значение. В том же питоне xmltodict изящно любой валидный XML разворачивает в словарь. А вообще вот годная статья про то, как предполагается работать с XML средствами стандартной библиотеки.

Но давайте разбираться с тем, что же в себе интересного таит XML.

У меня нет задачи подвергнуть читателя пыткам и мучительной смерти через зачитывание стандартов, поэтому сильно глубоко мы погружаться не будем, но какую-то скучную базу дать придётся.

XML сам по себе не делает ничего - это только формат представления информации, в отличие от HTML, который как раз таки призван отрисовать содержимое.

XML описывает что за данные внутри, а его теги не определены заранее, опять же в отличие от HTML.

То есть это два брата, похожих друг на друга внешне, но очень разных внутри.

Давайте сначала на отвлечённом примере поразбираемся?

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book>
    <title>Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <instock>
    </instock>
  </book>
  <book>
    <title>Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <instock />
  </book>
</bookstore>
```

Тут у нас XML, описывающий книжный магазин и имеющиеся в нём книги. У каждой книги есть свой набор атрибутов - название, автор, год выпуска, наличие в магазине.

Всё начинается с

7.4.1 XML Prolog

```
<?xml version="1.0" encoding="UTF-8"?>
```

Он опционален, однако обычно присутствует и должен идти первой строкой. Версия всегда строго 1.0, кодировка по умолчанию - UTF-8.

Коль скоро он опциональный, далее мы его опускаем.

7.4.2 Дерево элементов

XML представляет из себя дерево, состоящее из отдельных элементов. Оно может быть произвольной вложенности.

Самый первый элемент называется корневым - `root`, все последующие - его дети.

В примере выше это `<bookstore>`. Элемент представляет из себя открывающий и закрывающий теги и содержимое.

Теги заключены в угловые скобки и чувствительны к регистру. `<bookstore>` и `<Bookstore>` - это разные теги.

Соответственно между каждой парой определены отношения - родитель-ребёнок или собратья (`siblings`).

Детьми корневого элемента являются элементы `<book>`. Разные элементы `<book>` друг для друга являются собратьями.

Как такового понятия списка в XML нет, но по имени элемента мы (и код) понимаем, что они представляют из себя именно список.

У элемента `<book>` есть дочерние элементы. Их состав совсем не обязательно должен быть одинаковым - XML этого не требует, однако этого может (и скорее всего будет) требовать приложение.

Главное правило XML - каждый открывшийся тег должен быть закрыт: сказал `<a>` - говори и `</a>`. Элемент может быть пустым, просто выражая факт своего существования, тогда запись `<instock></instock>` можно заменить на просто `<instock/>`.

7.4.3 Атрибуты

Взглянем на другой пример:

```
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
  </book>
</bookstore>
```

Теперь внутри тега появилась строка вида `category="cooking"`. Она описывает дополнительные данные об элементе. Своего рода метаданные.

При этом вот эти две записи абсолютно равноправны **с точки зрения XML**:

```
<book category="cooking">
  <title lang="en">Everyday Italian</title>
  <author>Giada De Laurentiis</author>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  <year>2005</year>
</book>
```

и

```
<book>
  <category>cooking</category>
  <title>Everyday Italian
    <lang>en</lang>
  </title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
</book>
```

То есть XML в терминах ни синтаксиса, ни семантики понятия дочерний элемент и атрибут не разделяет. Это остаётся исключительно решением составителя/разработчика.

В целом к этому следует относиться именно как к метаданным - информации об информации. То есть если это не является неотъемлемым свойством объекта или нужно в служебных целях, то его можно вынести в атрибуты.

Чтобы далеко не уходить, вот пример из netconf:

```
<rpc message-id=”101>
<get-config>
  <source>
    <running/>
  </source>
</get-config>
</rpc>
```

Здесь `message-id` - это атрибут элемента RPC, который не имеет непосредственного отношения к передаваемым далее данным, но позволяет отследить по `message-id` ответ сервера (он вставит его в `<rpc-reply>`).

Ещё один пример, который мы будем разбирать дальше: `<interfaces operation="replace">`. Атрибут `operation="replace"` не является частью конфигурации интерфейса, он лишь говорит, что то, что существует сейчас на коробке в ветке `<interfaces>`, нужно заменить на то, что описано в данном XML.

Ну и замечу, что пусть с точки зрения XML атрибут и дочерний элемент взаимозаменяемы, когда вы придумываете свою схему обмена или хранения, однако NETCONF вам такого не простит. Да и любой другой интерфейс, в который вы встраиваетесь - ведь в нём уже определена схема XML.

7.4.4 Namespaces

Хух. Я откладывал много лет момент, когда придётся разобраться с неймспейсами в XML.

На самом деле ничего тут нет хитрого.

Если мы определили два разных элемента с одинаковыми именами, то появляется неоднозначность - какой именно элемент мы имеем в виду, обращаясь к нему по имени?

Например, элемент <name> может быть как у интерфейса, так и у пользователя и у влана итд. Их можно разнести в разные NS, хотя это не обязательно, потому что они находятся под разными родителями.

А если на одном уровне могут оказаться совпадающие имена - это уже настоящая проблема. Например,

```
<root>
<address>
  <city>
    <name>Moscow</name>
    <street>Novocheremushkinskaya, 50</street>
  </city>
</address>
<address>
  <ipv6>2a01:ba80:e:20::32</ipv6>
  <ipv4>185.127.149.137</ipv4>
</address>
</root>
```

В первом случае имеется в виду почтовый адрес, во втором - IP.

Здесь уже однозначно будет конфликт. Надо решать.

Сделать это можно несколькими способами.

1. Прямо объявляем неймспейсы с префиксами:

   ```
   <root>
   <postal:address xmlns:postal="https://www.linkmeup.ru/postal_address/">
     <postal:city>
       <postal:name>Moscow</postal:name>
       <postal:street>Novocheremushkinskaya, 50</postal:street>
     </postal:city>
   </postal:address>
   <ip:address xmlns:ip="https://www.linkmeup.ru/ip/">
     <ip:ipv6>2a01:ba80:e:20::32</ip:ipv6>
     <ip:ipv4>185.127.149.137</ip:ipv4>
   </ip:address>
   </root>
   ```

   Теперь это полное, fully qualified, имя безо всяких ограничений. Обращаемся из приложений, соответственно, по полному имени. `postal` и `ip` - это короткие префиксы. Само имя namespace - это произвольная строка. Но негласная договорённость, что все используют URI. Он может вести на страницу с описанием этого неймспейса, а может и не вести. Но указание префикса в каждом теге может показаться не очень удобным, тогда есть второй способ.

2. Определяем `default namespace`

   ```
   <root>
   <address xmlns="https://www.linkmeup.ru/postal_address/">
     <city>
       <name>Moscow</name>
       <street>Novocheremushkinskaya, 50</street>
     </city>
   ```

(continues on next page)

(продолжение с предыдущей страницы)

```
</address>
<address xmlns="https://www.linkmeup.ru/ip/">
  <ipv6>2a01:ba80:e:20::32</ipv6>
  <ipv4>185.127.149.137</ipv4>
</address>
</root>
```

Область действия дефолтного неймспейса - сам элемент и все его потомки, если он нигде не переопределяется.

Концепция namespace с одной стороны проста, с другой стороны и там есть место тёмным пятнам. Если хочется подетальнее изучить, то есть пара полезных FAQ про них.

7.4.5 Xpath - XML Path

Сначала правильно, но непонятно: `XPath` - это способ выбрать ноды или множество нод из XML документа.
Теперь неправильно, но понятно: это способ представить иерархию XML в виде «привычного» нам пути, где элементы отделены друг от друга знаком «/».

Например, в XML из примера выше путь к элементу `<title>` будет записан в виде `/bookstore/book/title`

Ну а теперь и правильно, и понятно, но долго.
XPath - это очень гибкий и мощный инструмент, позволяющий внутри XML делать разнообразные запросы. Он поддерживает различные функции: `sum`, `count`, `avg`, `min`, `starts-with`, `contains`, `concat`, `true`, `false` - над разными типами данных: числа, строки, булевы.
Так с помощью XPath можно выбрать названия всех книг с ценою выше 35:
`/bookstore/book[price>35]/title`

XPath оперирует нодами, которыми являются элементы, атрибуты, текст, неймспейсы и другое.

Соответственно помимо того, что мы можем запросить часть XML по конкретному пути, можно делать разные хитрые запросы.
Например:

- Вернуть BGP-группу, в которой есть peer 10.1.1.1
- Вернуть интерфейс, на котором число ошибок больше 100
- Вернуть список интерфейсов, на которых native-vlan 127
- Вернуть количество интерфейсов, в имени которых есть «Ethernet».

В контексте NETCONF вы можете его встретить, но это не самая популярная capability. В общем, знать про него полезно, но глубоко копать не будем. Если хочется поподробнее почитать, то это можно сделать например, тут.

7.4.6 Схема

Что такое XML - это удобный способ передавать структурированные данные между приложениями. Но это лишено какого-либо смысла, есть нет контракта о том, как данные в этих файлах должны храниться - где какие элементы и какого они типа.

Представьте, что информацию об IP-адресах мы будем помещать непосредственно в элемент `<interface>`, а читать его пытаются из элемента `<unit>`?

Или дату мы передаём в формате `YYYY-MM-DD`, а читать её пытаются в `MM-DD-YYYY` (больные ублюдки).

При этом сам XML будет абсолютно «Well Formed», что называется, - то есть соответствовать синтаксису XML.

Для этого и существует **Схема**. В отдельном XML-файле описывается схема данных для основного XML.

Это позволяет

- двум сторонам использовать один и тот же способ хранения и распаковки данных.
- описывать содержимое документа
- определять ограничения на данные
- проверять корректность XML

Называется это хозяйство **XML Schema Definition** - или коротко **XSD**.

Поскольку это тот же самый XML, он должен как-то обозначать себя, что является схемой. Для этого есть ключевой элемент `<schema>`. Вот так будет выглядеть XSD для кусочка XML выше:

```
<xs:schemaxmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:elementname="address">
    <xs:complexType>
      <xs:sequence>
        <xs:elementname="country_name" type="xs:string"/>
        <xs:elementname="population" type="xs:decimal"/>
      </xs:sequence>
    </xs:complexType>
</xs:element>
</xs:schema>
```

При этом в самом XML можно дать ссылку на XSD

```
<note
xmlns="https://www.linkmeup.ru"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="https://www.linkmeup.ru/404.xsd">
```

Самостоятельное продолжение изучения XSD.

Лучшая сторона XSD - это то, что на его основе можно автоматически генерировать объекты в языках программирования. То есть XSD описывает, какие именно объекты и структуры должны быть созданы, а конкретный XML - наполняет экземпляр, пользоваться которым значительно удобнее, чем крафтить XML. Со схемами и моделями мы будем разбираться дальше.

Надеюсь получилось, не утопая в деталях, дать понимание, что из себя представляет XML. Далее для нас это будет важным.

</XML>

7.5 NETCONF Again

И вот теперь время взглянуть на операции NETCONF и попрактиковаться.
Один из принципов NETCONF - это отделение конфигурационных данных от операционных.
Поэтому отдельными операциями он позволяет управлять конфигурацией, а отдельными - забирать информацию о состоянии.
Вот базовый неполный список операций NETCONF:

- `<get>`
- `<get-config>`
- `<edit-config>`
- `<copy-config>`
- `<delete-config>`
- `<lock>`
- `<unlock>`
- `<close-session>`
- `<kill-session>`

Но зачастую вендоры определяют свои собственные операции.

7.5.1 Действия, операции

<get>

Эта операция возвращает текущие (running) операционные и конфигурационные данные.
Выполните просто

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get/>
</rpc>
]]>]]>
```

И в ответ получите несколько экранов XML.
Ответ приходит в `<rpc-reply>`. В случае ошибки внутри `<rpc-reply>` сервер вернёт `<rpc-error>` с текстом ошибки.
Для получения ошибки можно просто сформировать некорректный XML.
Например, забудем закрывающий тег `</get>`:

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
</rpc>
]]>]]>
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos" message-id="100"␣
↪xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
```

(continues on next page)

(продолжение с предыдущей страницы)

```
<rpc-error>
<error-type>protocol</error-type>
<error-tag>operation-failed</error-tag>
<error-severity>error</error-severity>
<error-message>syntax error, expecting <filter> or </get></error-message>
<error-info>
<bad-element>interfaces</bad-element>
</error-info>
</rpc-error>
</rpc-reply>
```

Или запросить несуществующую ветку:

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
    <interfaces/>
  </get>
</rpc>
]]>]]>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos" message-id="100"␣
↪xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<rpc-error>
<error-type>protocol</error-type>
<error-tag>operation-failed</error-tag>
<error-severity>error</error-severity>
<error-message>syntax error, expecting <filter> or </get></error-message>
<error-info>
<bad-element>interfaces</bad-element>
</error-info>
</rpc-error>
</rpc-reply>
```

В зависимости от вендора в ответе на `<get>` будет содержаться либо вообще всё, что вам может дать устройство - полный конфиг и вся информацию по состоянию, либо какую-то часть.

Так, Juniper возвращает конфиг и совсем немного данных сверху. Для того, чтобы забрать операционные данные, нужно использовать специальные операции, например `<get-interface-information>`:

```
<rpc>
    <get-interface-information/>
</rpc>
```

Вот такой будет ответ: https://pastebin.com/2xTpuSi3.

> Этому, кстати, сложно найти объяснение. Довольно неудобно для каждой ветки операционных данных иметь собственный RPC. И более того, непонятно как это вообще описывается в моделях данных.

Очевидно, это не всегда (никогда) удобно. Хотелось бы пофильтровать данные. NETCONF позволяет не просто отфильтровать результат, а указать NETCONF-серверу, какую именно часть клиент желает запросить. Для этого используется элемент `<filter>`.

<filter>

С его помощью можно указать какую именно часть информации вы хотите получить. Можно указывать атрибут фильтрации, поддерживаются subtree и xpath.
По умолчанию используется subtree, но обычно его задают явно, дабы избежать двусмысленности.

Давайте на примере get пофильтруем ответ.
Без фильтра совсем данные вернутся полностью.

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get/>
</rpc>
]]>]]>
```

Вот такой будет ответ: https://pastebin.com/MMWXM2eT.

С пустым фильтром не вернётся никаких данных.

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
   <filter type="subtree">
   </filter>
  </get>
</rpc>
]]>]]>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos" message-id="100"␣
↪xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<database-status-information>
<database-status>
<user>eucariot</user>
<terminal></terminal>
<pid>31101</pid>
<start-time junos:seconds="1644636396">2022-02-12 03:26:36 UTC</start-time>
<edit-path></edit-path>
</database-status>
</database-status-information>
</data>
</rpc-reply>
]]>]]>
```

Вот таким запросом можно вытащить конфигурационные данные по всем интерфейсам

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
   <filter type="subtree">
     <configuration>
       <interfaces/>
     </configuration>
   </filter>
  </get>
</rpc>
]]>]]>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos" message-id="100"␣
↪xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:changed-seconds=
↪"1644510087" junos:changed-localtime="2022-02-10 16:21:27 UTC">
    <interfaces>
        <interface>
            <name>ge-0/0/0</name>
            <unit>
                       <name>0</name>
                       <family>
                   <inet>
                             <address>
                          <name>169.254.0.1/31</name>
                             </address>
                   </inet>
                       </family>
            </unit>
        </interface>
        <interface>
            <name>ge-0/0/2</name>
            <unit>
                       <name>0</name>
                       <family>
                   <inet>
                             <address>
                          <name>169.254.100.1/31</name>
                             </address>
                   </inet>
                       </family>
            </unit>
        </interface>
        <interface>
            <name>em0</name>
            <unit>
                       <name>0</name>
                       <family>
                   <inet>
                             <address>
                          <name>192.168.1.2/24</name>
                             </address>
                   </inet>
                       </family>
            </unit>
        </interface>
    </interfaces>
</configuration>
<database-status-information>
<database-status>
<user>eucariot</user>
<terminal></terminal>
<pid>31101</pid>
<start-time junos:seconds="1644636721">2022-02-12 03:32:01 UTC</start-time>
<edit-path></edit-path>
</database-status>
</database-status-information>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
</data>
</rpc-reply>
]]>]]>
```

Если вы хотите выбрать не все элементы дерева, а только интересующую вас часть, то можно указать, какие именно нужны:

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
   <filter type="subtree">
     <configuration>
       <interfaces>
         <interface>
           <name/>
           <description/>
         </interface>
       </interfaces>
     </configuration>
   </filter>
  </get>
</rpc>
]]>]]>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos" message-id="100"␣
↪xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:changed-seconds=
↪"1644637011" junos:changed-localtime="2022-02-12 03:36:51 UTC">
    <interfaces>
        <interface>
            <name>ge-0/0/0</name>
            <description>kzn-leaf-0</description>
        </interface>
        <interface>
            <name>ge-0/0/2</name>
            <description>kzn-edge-0</description>
        </interface>
        <interface>
            <name>em0</name>
            <description>mgmt-switch</description>
        </interface>
    </interfaces>
</configuration>
<database-status-information>
<database-status>
<user>eucariot</user>
<terminal></terminal>
<pid>31316</pid>
<start-time junos:seconds="1644637103">2022-02-12 03:38:23 UTC</start-time>
<edit-path></edit-path>
</database-status>
</database-status-information>
</data>
</rpc-reply>
]]>]]>
```

При этом если хочется забрать данные только по конкретному интерфейсу:

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
   <filter type="subtree">
     <configuration>
       <interfaces>
         <interface>
           <name>ge-0/0/0</name>
         </interface>
       </interfaces>
     </configuration>
   </filter>
  </get>
</rpc>
]]>]]>


<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos" message-id="100"␣
↪xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:changed-seconds=
↪"1644637011" junos:changed-localtime="2022-02-12 03:36:51 UTC">
    <interfaces>
        <interface>
            <name>ge-0/0/0</name>
            <description>kzn-leaf-0</description>
            <unit>
                 <name>0</name>
                 <family>
                     <inet>
                         <address>
                             <name>169.254.0.1/31</name>
                         </address>
                     </inet>
                 </family>
            </unit>
        </interface>
    </interfaces>
</configuration>
<database-status-information>
<database-status>
<user>eucariot</user>
<terminal></terminal>
<pid>31316</pid>
<start-time junos:seconds="1644637321">2022-02-12 03:42:01 UTC</start-time>
<edit-path></edit-path>
</database-status>
</database-status-information>
</data>
</rpc-reply>
]]>]]>
```

Соответственно можно совместить запрос конкретного интерфейса и только тех его полей, которые интересны.

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get>
   <filter type="subtree">
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    <configuration>
      <interfaces>
        <interface>
          <name>ge-0/0/0</name>
          <description/>
        </interface>
      </interfaces>
    </configuration>
  </filter>
 </get>
</rpc>
]]>]]>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos" message-id="100"␣
↪xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:changed-seconds=
↪"1644637011" junos:changed-localtime="2022-02-12 03:36:51 UTC">
    <interfaces>
        <interface>
            <name>ge-0/0/0</name>
            <description>kzn-leaf-0</description>
        </interface>
    </interfaces>
</configuration>
<database-status-information>
<database-status>
<user>eucariot</user>
<terminal></terminal>
<pid>31316</pid>
<start-time junos:seconds="1644637396">2022-02-12 03:43:16 UTC</start-time>
<edit-path></edit-path>
</database-status>
</database-status-information>
</data>
</rpc-reply>
]]>]]>
```

Ещё немного про subtree filtering.

В случае Juniper `<get>` ничем практически не отличается от `<get-config>`. Для того, чтобы забрать операционные данные, нужно воспользоваться другими операциями - специфическими под каждую задачу.

Узнать их можно достаточно просто: `show version | display xml rpc`

С помощью операций `<get>` удобно забирать операционные данные с устройства. Например, для мониторинга. Или для отладки. Можно выбрать всех BGP-соседей в состоянии Idle, или все интерфейсы с ошибками, данные по маршрутам.

Да, понятно, что для всего этого есть и более удобные способы, но всё же такой путь есть.

<get-config>

Позволяет забрать конфигурацию устройства.
Могло показаться, что `<get-config>` - это поддерево `<get>`, но это всё-таки не так.

С помощью `<get-config>` можно указать из какого источника мы хотим получить конфигу - `running`, `candidate`, `startup` итд.

Ну и можно быть уверенным, что в ответе будут только конфигурационные данные.
Хотя по своему опыту вам скажу, что вендоры тут могут отличаться изобретательностью, подмешивая оперданные к конфиге.

Забираем текущий конфиг:

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
   <source>
     <running/>
   </source>
  </get-config>
</rpc>
]]>]]>
```

`<get-config>` так же, как и `<get>` позволяет использовать элемент `<filter>`. Например:

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
   <source>
     <running/>
   </source>
   <filter type="subtree">
     <configuration>
       <system>
          <host-name/>
       </system>
     </configuration>
   </filter>
  </get-config>
</rpc>
]]>]]>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos" message-id="100"␣
↪xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:commit-seconds=
↪"1644637011" junos:commit-localtime="2022-02-12 03:36:51 UTC" junos:commit-user=
↪"eucariot">
    <system>
        <host-name>kzn-spine-0</host-name>
    </system>
</configuration>
</data>
</rpc-reply>
]]>]]>
```

В запросе самые внимательные обратили внимание на элемент `<source>`.

Configuration Datastores

Это место для хранения полной конфигурации. Хотя слово «хранения», возможно, и не очень точное. Обязательным является только `<running>` - это текущая актуальная конфигурация.

В зависимости от вендора и поддерживаемых капабилитей могут быть так же `<candidate>`, `<startup>` и какие-то другие.

Соответственно запросить конфигурацию можно из разных Datastores при их наличии, указывая соответствующий элемент внутри `<source>`.

Как увидим далее, менять конфигурацию так же, можно в разных datastores через `<target>`.

И тут разные вендоры ведут себя по-разному, кто-то разрешает менять сразу в `<running>`, а кто-то только `<candidate>` с последующим `<commit>`.

<edit-config>

ЕЙ богу, самая интересная штука во всём NETCONF! Операция, с помощью которой можно привести конфигурацию к нужному состоянию. Серебряная пуля, панацея, окончательное решение конфигурационного вопроса. Ага, щаз! Идея в теории прекрасна: мы отправляем на устройство желаемую конфигурацию в виде XML, а оно само шуршит и считает, что нужно применить, а что удалить. Давайте идеальный случай и разберём сначала.

`<edit-config>` позволяет загрузить полную конфигурацию устройства или его часть в указанный datastore. При этом устройство сравнивает актуальную конфигурацию в datastore и передаваемую с клиента и предпринимает указанные действия.

А какие действия могут быть указаны? Это определяется атрибутом `operation` в любом из элементов поддерева `<configuration>`. Operation может встречаться несколько раз в XML и быть при этом разным. Атрибут может принимать следующие значения:

- **Merge** - новая конфига вливается в старую - что необходимо заменить - заменяется, новое - добавляется, ничего не удаляется.
- **Replace** - заменяет старую конфигурацию новой.
- **Create** - создаёт блок конфигурации. Однако, если он уже существует, вернётся `<rpc-error>`
- **Delete** - удаляет блок конфигурации. Однако, если его не существует, вернётся `<rpc-error>`
- **Remove** - удаляет блок конфигурации. Однако, если его не существует, проигнорирует. Определён в RFC6241.

Если тип операции не задан, то новая конфигурация будет вмёржена в старую. Задать операцию по умолчанию можно с помощью параметра `<default-operation>`: `merge`, `replace`, `none`.

В дереве `<configuration>` задаётся собственно целевая конфигурация в виде XML.

Безусловно, самая интересная операция внутри `<edit-config>` - это replace. Ведь она предполагает, что устройство возьмёт конфигурацию из RPC и заменит ею ту, что находится в datastore. А где-то там под капотом и крышкой блока цилиндров система сама просчитает дельту, которую нужно отправить на чипы.

Практика edit-config

Давайте сначала что-то простое: поменяет hostname:

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
   <target>
     <candidate/>
   </target>
   <config>
     <configuration>
       <system>
          <host-name>just-for-lulz</host-name>
       </system>
     </configuration>
   </config>
  </edit-config>
</rpc>
]]>]]>
```

Проверяем, что в кандидат-конфиге эти изменения есть, а в текущем - нет

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
   <source>
     <candidate/>
   </source>
   <filter type="subtree">
     <configuration>
       <system>
          <host-name/>
       </system>
     </configuration>
   </filter>
  </get-config>
</rpc>
]]>]]>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos" message-id="100"␣
↪xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:changed-seconds=
↪"1644719855" junos:changed-localtime="2022-02-13 02:37:35 UTC">
    <system>
        <host-name>just-for-lulz</host-name>
    </system>
</configuration>
</data>
</rpc-reply>
]]>]]>
```

Проверяем running:

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
   <source>
     <running/>
   </source>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
   <filter type="subtree">
     <configuration>
       <system>
          <host-name/>
       </system>
     </configuration>
   </filter>
  </get-config>
</rpc>
]]>]]>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos" message-id="100"␣
↪xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:commit-seconds=
↪"1644637011" junos:commit-localtime="2022-02-12 03:36:51 UTC" junos:commit-user=
↪"eucariot">
    <system>
        <host-name>kzn-spine-0</host-name>
    </system>
</configuration>
</data>
</rpc-reply>
```

Значит, надо закоммитить изменения.

```
<rpc>
  <commit/>
</rpc>
]]>]]>

<rpc-reply xmlns="urn:ietf:params:xml:ns:netconf:base:1.0" xmlns:junos="http://xml.
↪juniper.net/junos/14.1R1/junos">
<ok/>
</rpc-reply>
```

Проверяем running:

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
   <source>
     <running/>
   </source>
   <filter type="subtree">
     <configuration>
       <system>
          <host-name/>
       </system>
     </configuration>
   </filter>
  </get-config>
</rpc>
]]>]]>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos" message-id="100"␣
↪xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
```

(continues on next page)

(продолжение с предыдущей страницы)

```
<data>
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:commit-seconds=
↪"1644720065" junos:commit-localtime="2022-02-13 02:41:05 UTC" junos:commit-user=
↪"eucariot">
    <system>
        <host-name>just-for-lulz</host-name>
    </system>
</configuration>
</data>
</rpc-reply>
```

На Juniper доступны в NETCONF те же функции коммитов, что и в CLI. Например, `commit confirmed` и `confirmed-timeout`.

А теперь что-то посложнее и с операцией `replace`: заменим список BGP-пиров:

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
   <target>
     <candidate/>
   </target>
   <config>
     <configuration>
    <protocols>
            <bgp operation="replace">
                <group>
                    <name>LEAFS</name>
                    <type>external</type>
                    <import>ALLOW</import>
                    <family>
                        <inet>
                            <unicast>
                            </unicast>
                        </inet>
                    </family>
                    <export>EXPORT</export>
                    <neighbor>
                        <name>169.254.0.0</name>
                        <peer-as>64513.00000</peer-as>
                    </neighbor>
                </group>
                <group>
                    <name>EDGES</name>
                    <type>external</type>
                    <import>ALLOW</import>
                    <family>
                        <inet>
                            <unicast>
                            </unicast>
                        </inet>
                    </family>
                    <export>EXPORT</export>
                    <neighbor>
                        <name>222.222.222.0</name>
                        <peer-as>65535</peer-as>
                    </neighbor>
                </group>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            </bgp>
        </protocols>
    </configuration>
   </config>
  </edit-config>
</rpc>
]]>]]>
```

Коммит

```
<rpc>
  <commit/>
</rpc>
]]>]]>
```

Проверяем running

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<get-config>
<source>
    <running/>
</source>
<filter type="subtree">
    <configuration>
    <protocols>
        <bgp>
            <group>
            <neighbor/>
            </group>
        </bgp>
    </protocols>
    </configuration>
</filter>
</get-config>
</rpc>
]]>]]>

<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos" message-id="100"␣
↪xmlns="urn:ietf:params:    xml:ns:netconf:base:1.0">
<data>
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:commit-seconds=
↪"1644720678"        junos:commit-localtime="2022-02-13 02:51:18 UTC" junos:commit-
↪user="eucariot">
    <protocols>
        <bgp>
            <group>
                <name>LEAFS</name>
                <neighbor>
                    <name>169.254.0.0</name>
                    <peer-as>64513.00000</peer-as>
                </neighbor>
            </group>
            <group>
                <name>EDGES</name>
                <neighbor>
                    <name>222.222.222.0</name>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                    <peer-as>65535</peer-as>
                </neighbor>
            </group>
        </bgp>
    </protocols>
</configuration>
</data>
</rpc-reply>
```

Всё сработало)

А теперь попробуем операцию *merge* при добавлении нового пира.

```
<rpc message-id="100" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <edit-config>
   <target>
     <candidate/>
   </target>
   <config>
     <configuration>
    <protocols>
            <bgp operation="merge">
                <group>
                    <name>LEAFS</name>
                    <type>external</type>
                    <import>ALLOW</import>
                    <family>
                        <inet>
                            <unicast>
                            </unicast>
                        </inet>
                    </family>
                    <export>EXPORT</export>
                    <neighbor>
                        <name>169.254.0.0</name>
                        <peer-as>64513.00000</peer-as>
                    </neighbor>
                </group>
                <group>
                    <name>EDGES</name>
                    <type>external</type>
                    <import>ALLOW</import>
                    <family>
                        <inet>
                            <unicast>
                            </unicast>
                        </inet>
                    </family>
                    <export>EXPORT</export>
                    <neighbor>
                        <name>222.222.222.0</name>
                        <peer-as>65535</peer-as>
                    </neighbor>
                    <neighbor>
                        <name>169.254.100.0</name>
                        <peer-as>65535</peer-as>
                    </neighbor>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            </group>
          </bgp>
        </protocols>
      </configuration>
    </config>
  </edit-config>
</rpc>
]]>]]>
```

Коммит

```
<rpc>
  <commit/>
</rpc>
]]>]]>
```

Проверка:

```
<rpc-reply xmlns:junos="http://xml.juniper.net/junos/14.1R1/junos" message-id="100"␣
↪xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
<data>
<configuration xmlns="http://xml.juniper.net/xnm/1.1/xnm" junos:commit-seconds=
↪"1644721481" junos:commit-localtime="2022-02-13 03:04:41 UTC" junos:commit-user=
↪"eucariot">
    <protocols>
        <bgp>
            <group>
                <name>LEAFS</name>
                <neighbor>
                    <name>169.254.0.0</name>
                    <peer-as>64513.00000</peer-as>
                </neighbor>
            </group>
            <group>
                <name>EDGES</name>
                <neighbor>
                    <name>222.222.222.0</name>
                    <peer-as>65535</peer-as>
                </neighbor>
                <neighbor>
                    <name>169.254.100.0</name>
                    <peer-as>65535</peer-as>
                </neighbor>
            </group>
        </bgp>
    </protocols>
</configuration>
</data>
</rpc-reply>
]]>]]>
```

Вот он новенький пир, и старые на месте.
То есть достаточно очевидна разница между работой `replace` и `merge`.

Operation replace

С `replace` следует иметь в виду некоторые нюансы. Например, что нужно передавать полную конфигурацию того или иного сервиса или функциональности - не просто новые параметры - ведь железка натурально заменит то, что было, тем, что прилетело. Едва ли вы хотите создав один интерфейс в OSPF Area, удалить остальные?

Некоторые сущности не могут быть удалены, такие, например, как физические интерфейсы. Поэтому при формировании соответствующего блока конфигурации нужно быть аккуратнее - в целевой конфигурации должны все они присутствовать, иначе в лучшем случае вернётся *<rpc-error>*, а в худшем вы чего-то поудаляете.

Использовать *replace* можно как на уровне отдельных частей конфигурации, так и на верхнем уровне, требуя заменить всё поддерево.

Однако ещё один нюанс заключается в том, что в зависимости от реализации вычисление дельты может занять много ресурсов CPU. Поэтому, если собираетесь кинуть диф на 13 000 строк политик BGP, то дважды подумайте и трижды оттестируйте, что после этого происходит с коробкой.

<commit>

Ещё одно свидетельство того, что модель NETCONF скалькирована с API Juniper - это возможность commit'a candidate-конфигурации в running. Доступна она, конечно, только в том случае, если при обмене capability сервер сообщил, что поддерживает candidate datastore.

`<commit>` не замещает running на candidate, как это делает `<copy-config>`, а выполняет именно применение конфигурационной дельты, как это происходит в CLI.

Как и в CLI у `commit` может быть параметр `confirmed`, заставляющий откатить изменения, если commit не был подтверждён. За это отвечает отдельная capability: `confirmed-commit`.

`<commit>` не входит в число базовых операций, поскольку как раз зависит от поддерживаемых возможностей сервера.

<copy-config>

Операция заменяет одну конфигурацию другой. Имеет два параметра: `source` - откуда - и `target` - куда. Может использоваться как для применения новой конфигурации на коробку, так и для бэкапа активной. Если коробка поддерживает capability `:url`, то в качестве `source` и/или `target` может быть указан URL.

<delete-config>

Очевидно, удаляет конфигурацию из target datastore. Без хитростей.

<lock/unlock>

Аналогично Juniper CLI ставит блок на target datastore от совместного редактирования, чтобы не было конфликта. Причём блок должен работать как на NETCONF, так и на другие способы изменения конфигурации - SNMP, CLI, gRPC итд.

<close-session>

Аккуратно закрывает существующую NETCONF-сессию, снимает локи, высвобождает ресурсы.

<kill-session>

Грубо разрывает сессиию, но снимает локи. Если сервер получил такую операцию в тот момент, когда он дожидается confirmed commit, он должен отменить его и откатить изменения к состоянию, как было до установки сессии.

7.5.2 Инструменты разработчика для NETCONF

Ну вот как будто бы необходимый базис по NETCONF набрали.
Я в этой статье не ставлю перед собой задачу выстроить какую-то систему автоматизации. Просто хочу показать разные интерфейсы в теории и на практике.

И я думаю, к этому моменту вам уже очевидно, что отправка XML через SSH с ручным проставлением Framing Marker (`]]>]]>`) - не самый удобный способ. Давайте посмотрим на существующие библиотеки.

netconf-console

Прежде чем писать какой-то код, обычно стоит проверить всё руками. Но вот руками крафтить XML и проставлять framing marker'ы тоскливо. Тут отца русской автоматизации спасёт `netconf-console` - главный и, возможно, единственный CLI-инструмент для работы с NETCONF.

Может работать в режиме команды:

```
netconf-console --host 192.168.1.2 --port 22 -u eucariot -p password --get-config
```

А может в интерактивном:

```
netconf-console2 --host 192.168.1.2 --port 22 -u eucariot -p password -i
netconf> hello
```

Чуть больше про библиотеку у Романа Додина.

NCclient

Это, пожалуй, самая известная библиотека для работы с NETCONF. Она для питона и достаточно зрелая.
Начать пользоваться очень легко:

```
from ncclient import manager


if __name__ == "__main__":
    with manager.connect(
        host="kzn-spine-0.juniper",
        ssh_config=True,
        hostkey_verify=False,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        device_params={'name': 'junos'}
    ) as m:
        c = m.get_config(source='running').data_xml

    print(c)
```

Дабы уберечь читателя от многочасовых мук с отладкой аунтентификации, небольшая подсказка тут. Текущая версия `paramiko` на момент написания статьи (>=2.9.0), которую подтягивает `ncclient`, в ряде случае не может работать с OpenSSH-ключами и падает с ошибкой «Authentication failed». Рекомендую в этом случае устанавливать 2.8.0. На гитхабе открыта куча issue на эту тему. И, кажется, его даже починили, но я не проверял. И вроде бы даже есть решение, но и это я не проверял.

Так же работают *filter*:

```
from ncclient import manager

rpc = """
     <filter>
     <configuration>
       <system>
          <host-name/>
       </system>
     </configuration>
     </filter>
     """

if __name__ == "__main__":
    with manager.connect(
        host="kzn-spine-0.juniper",
        ssh_config=True,
        hostkey_verify=False,
        device_params={"name": "junos"}
    ) as m:
        c = m.get_config("running", rpc).data_xml

    print(c)
```

С таким вот результатом:

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc-reply message-id="urn:uuid:864dd143-7a86-40ca-8992-5a35f2322ea0">
  <data>
          <configuration commit-seconds="1644732354" commit-localtime="2022-02-13␣
↪06:05:54 UTC" commit-user="eucariot">
      <system>
        <host-name>
        kzn-spine-0
        </host-name>
      </system>
    </configuration>
  </data>
</rpc-reply>
```

На текстовый XML смотреть не надо - парсим библиотечкой *xmltodict*:

```
from ncclient import manager
import xmltodict

rpc = """
     <filter>
     <configuration>
       <system>
          <host-name/>
       </system>
     </configuration>
     </filter>
     """

if __name__ == "__main__":
    with manager.connect(
        host="kzn-spine-0.juniper",
        ssh_config=True,
        hostkey_verify=False,
        device_params={"name": "junos"}
    ) as m:
        result = m.get_config("running", rpc).data_xml
    result_dict = xmltodict.parse(result)
    print(f'hostname is {result_dict["rpc-reply"]["data"]["configuration"]["system"][
↪"host-name"]}')
```

С уже таким результатом:

```
hostname is kzn-spine-0
```

При работе с сетевыми коробками по NETCONF xmltodict, пожалуй, самая практичная библиотека, преобразующая XML-данные в объект Python. Она использует C-шный парсер pyexpat, так что недостатков у такого подхода фактически нет.

Точно так же можно обновить конфигурацию в два действия: `<edit-config>` в `<candidate>` и `<commit>`:

```
from ncclient import manager
import xmltodict

rpc = """
     <config>
       <configuration>
         <interfaces>
           <interface>
             <name>ge-0/0/0</name>
             <description>Mit der Dummheit kämpfen Götter selbst vergebens.</
↪description>
           </interface>
         </interfaces>
       </configuration>
     </config>
     """

if __name__ == "__main__":
    with manager.connect(
        host="kzn-spine-0.juniper",
        ssh_config=True,
        hostkey_verify=False,
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        device_params={"name": "junos"}
    ) as m:
        result = m.edit_config(target="candidate", config=rpc).data_xml
        m.commit()
    result_dict = xmltodict.parse(result)
    print(result_dict)

OrderedDict([('rpc-reply', OrderedDict([('@message-id', 'urn:uuid:93bde991-81f9-42d6-
↪a343-b4fc267646c2'), ('ok', None)]))])
```

Дальше пока копать не будем. Тем более, бытует мнение *«без всяких сомнений, самый ублюдочно написанный Python код, что я видел в opensource»*

scrapli-netconf

NCclient был первым и классным, но отсутствие поддержки async в нём сильно ограничивает его использование.

Тут нас выручает Карл Монтанари, который уже подарил миру scrapli.

Но для тех, кто достаточно смел, чтобы использовать на своей сети NETCONF, создали scrapli-netconf.

Давайте взглянем на пару примеров работы.

```
from scrapli_netconf.driver import NetconfDriver

rpc = """
     <filter>
     <configuration>
       <system>
          <host-name/>
       </system>
     </configuration>
     </filter>
     """

device = {
        "host": "kzn-spine-0.juniper",
        "auth_strict_key": False,
        "port": 22
        }

if __name__ == "__main__":
    with NetconfDriver(**device) as conn:
        response = conn.get_config("running", rpc)

    print(response.result)
```

Scrapligo и scrapligo-netconf

Для Go тоже не придумано ничего лучше, чем scrapligo, в котором есть модуль для работы через netconf.

Так что если вы сетевик, осваивающий Го, путь для вас уже проложен.

7.5.3 Как это использовать

Мониторинг

NETCONF предоставляет возможность собирать операционные данные:

- Состояния протоколов (OPSF, BGP-пиринги)
- Статистику интерфейсов
- Утилизацию ресурсов CPU
- Таблицы маршрутизации
- Другое

При этом возвращаются структурированные данные, с которыми легко работать без сложных процедур парсинга.
Поэтому NETCONF вполне можно использовать для целей мониторинга.
Тут вы спросите: а зачем, если есть SNMP? А я отвечу. Точнее постараюсь.

- Используем безопасный SSH, не используем SNMP
- Не несём дополнительные протоколы в сеть
- Полная свобода того, какие данные мы собираем, без необходимости разбираться в OID'ах и MIB'ах
- При этом есть возможность собирать данные в соответствии с YANG-моделью
- Гипотетическая возможность оформить подписку на события в системе

Выполнение отдельных операций

Используя NETCONF, можно выполнять какие-то конкретные задачи: собрать данные с сети или изменить какую-то часть конфигурации.
Например, вы хотите периодически собирать MAC-адреса с сети или список коммитов.
Или вам нужно переключать порт коммутатора в другой VLAN.
Или например, у вас есть скрипт, который проверяет, что устройство в порядке - правильные настройки сислог-сервера, корректное время и пинги, куда полагается, работают.
Это всё можно сделать и на парсинге CLI, безусловно, но структурированные данные - это структурированные данные, а regexp - это regexp.

Configuration Management

Да, это тоже возможно, если

1. Оборудование поддерживает 100% конфигурации через NETCONF. Увы, я на своём веку повидал ситуаций, когда некоторые секции просто-напросто отсутствовали в NETCONF и никакого способа настроить нужную функцию нет.
2. Оборудование честно поддерживает операцию «replace», без этого вычисление конфигурационной дельты ложится вновь на сетевиков.

Однако, в том виде, в котором мы познакомились с темой на данный момент, дальше начинается Jinja-программирование. Каждому, кто этим занимался, обычно неловко, и он стыдливо избегает разговора на эту тему.

Задача решается примерно следующим образом:

3. Пишем циклопические развесистые jinja-шаблоны с ифами и форами, внутри которых XML. Шаблоны под каждого вендора, конечно, свои собственные, поскольку и схемы данных у них разные. Но при этом они универсальные в плане ролей устройств - не нужно для свитчей доступа и маршрутизаторов ядра писать разные шаблоны - просто в зависимости от роли будут активироваться те или иные их части.

   Здесь в нужных местах сразу описаны типы операций - где merge, где replace.

4. Каким-то образом формируем под каждое устройство файлы переменных, в которых указаны хостнеймы, IP-адреса, ASN, пиры и прочие специфические вещи. Эти файлы переменных в свою очередь, напротив, вендор-нейтральны, но будут отличаться от роли к роли.

5. Рендерим конфигурацию в формате XML, накладывая переменные на шаблоны. Получаем целевую конфигурацию в виде дерева XML, где в нужных местах проставлена операция `replace`.

6. Этот XML с помощью ncclient, ansible, scrapli-netconf или чего-то ещё подпихиваем на коробку.

7. NETCONF-сервер на коробке получает RPC и вычисляет конфигурационный патч, который фактически применит. То есть он находит разницу между целевой конфигурацией в RPC и текущей в `<running>`. Применяет эту конфигурацию.

Как бы это могло выглядеть я уже показывал в предыдущем выпуске АДСМ.

Источник: dteslya.engineer/network_automaiton_101/

Ручная правка файлов переменных - это очень неудобно, конечно же. Просто мрак, если мы говорим про какие-то типовые вещи, как например датацентровые регулярные топологии. Новая пачка стоек - сотни и тысячи строк для копипащения и ручного изменения. Но на самом деле их можно создавать автоматически на основе данных из централизованной базы данных - DCIM/IPAM.

Почему я об этом говорю так уверенно?
Потому что мы у себя (в Яндексе) полностью построили весь жизненный цикл отдельного сегмента сети на основе описанной схемы. И любые изменения на сеть могут применяться только через подобный конвейер и NETCONF. Любые временные конфигурации на железе перетрутся следующим же релизом.

Что тут хорошо:

1. Изменения в Jinja-шаблонах версионируются через git и проходят проверку другими инженерами перед применением. Это систематические изменения, влияющие на большое количество устройств.

2. Изменения в переменных - точно так же. Это точечное изменение конкретного устройства.

3. Только после согласования изменений в пунктах выше, можно сгенерировать новую конфигурацию и далее уже её отправить на проверку в git.

4. Если соблюдать процесс, то отсутствует конфигурационный дрейф.

Что тут плохо?

1. Ну, очевидно, Jinja-программирование

2. Работа с текстом, вместо объектов языка.

3. Отсутствие возможности взглянуть на конфигурационный диф до его применения.

На этом на самом деле заканчивается первая большая часть этой статьи, которая позволяет просто уже взять и получать пользу от NETCONF в задачах автоматизации.

Я вот прям серьёзно сейчас, ей богу! Не туманные абстракции - берём NETCONF - и на многих вендорах уже можно с ним работать выстраивая автоматизацию того или иного объёма.

Как вам ощущения от составления XML? А представьте, что вам нужно всю конфигурацию на несколько тысяч строк описать? А приправить это всё Jinja-программированием? А описывать в ямлах переменные?
Но абсолютное большинство тех, кто использует сегодня NETCONF, именно так и делают. (!)
Мнение автора. Change my mind!
В то время как есть YANG и набор инструментов вокруг него?

Хух. Давайте просто не будем об этом сейчас? Просто не сейчас? Попозже. После RESTCONF и gRPC?

7.6 RESTCONF

Просто пара слов об этом мертворожденном протоколе.
Это помесь RESTAPI и NETCONF, которая была призвана упростить управление сетью для WEB-приложений.
Внутри идеологически это NETCONF с его datastores и способами работать с конфигурацией, однако в качестве транспорта - HTTP с набором операций CRUD, реализованных через стандартные методы (*GET*, *POST*, *PUT*, *PATCH*, *DELETE*).
Данные передаются в формате JSON или XML.
В качестве модели данных используется YANG.

Описан в RFC8040.

Не могу отказать себе в удовольствии попробовать.
Возьмём на этот раз Arista veos-4.21.

Что нужно настроить, чтобы заработал restconf:

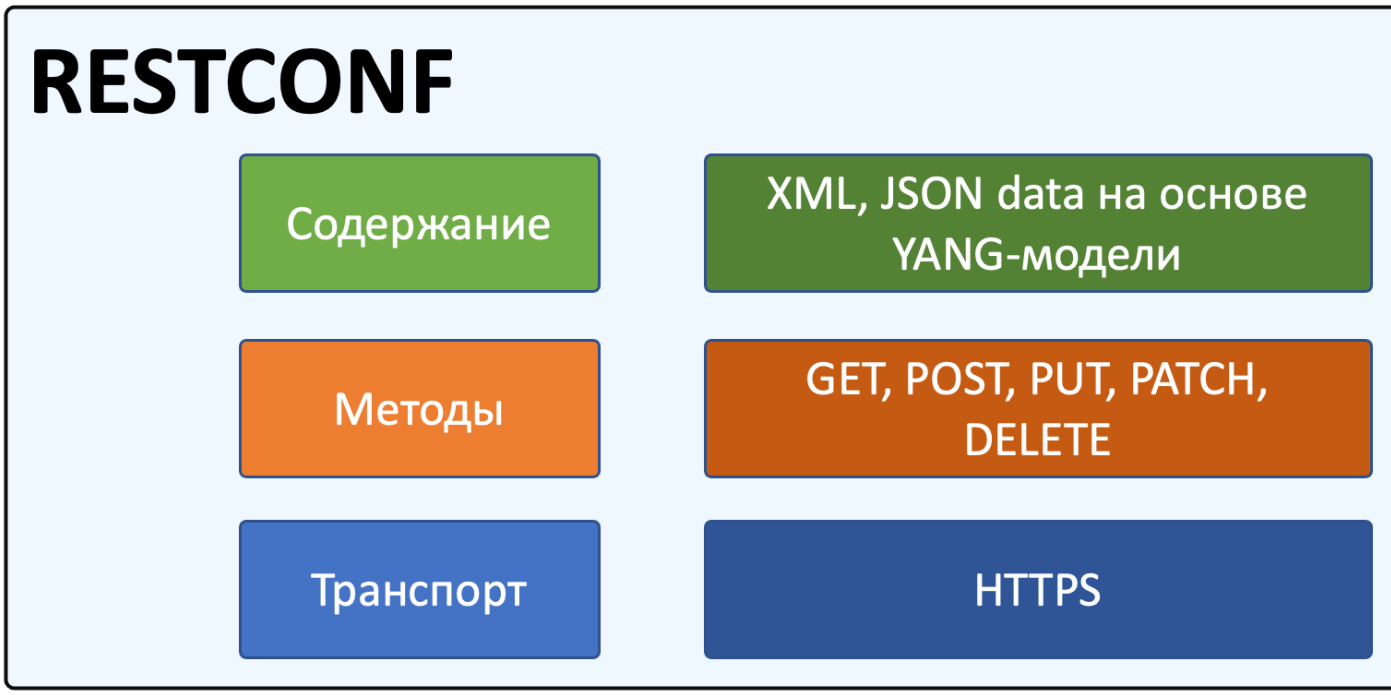
1. Выпускаем самоподписанный сертификат

```
security pki certificate generate self-signed restconf.crt key restconf.key␣
↪generate rsa 2048 parameters common-name restconf
certificate:restconf.crt generated
```

2. Разрешаем доступ на устройство по порту 6020 - правим *control-plane acl*

   Смотрим то, что разрешено сейчас - это readonly acl.

```
show ip access-lists default-control-plane-acl
```



Копируем правила и создаём копию ACL. Добавляем правило, разрешающее доступ по порту 6020

```
ip access-list control-plane-acl-with-restconf
9 permit tcp any any eq 6020
30 permit udp any any eq bfd ttl eq 255
40 permit udp any any eq bfd-echo ttl eq 254
50 permit udp any any eq multihop-bfd
60 permit udp any any eq micro-bfd
70 permit ospf any any
80 permit tcp any any eq ssh telnet www snmp bgp https msdp ldp
90 permit udp any any eq bootps bootpc snmp rip ntp ldp
100 permit tcp any any eq mlag ttl eq 255
110 permit udp any any eq mlag ttl eq 255
120 permit vrrp any any
130 permit ahp any any
140 permit pim any any
150 permit igmp any any
160 permit tcp any any range 5900 5910
170 permit tcp any any range 50000 50100
180 permit udp any any range 51000 51100
190 permit tcp any any eq 3333
200 permit tcp any any eq nat ttl eq 255
210 permit tcp any eq bgp any
220 permit rsvp any any
```

Применяем ACL на Control-Plane

```
control-plane
   ip access-group control-plane-acl-with-restconf in
```

3. Включаем сервис RESTCONF

```
management api restconf
    transport https test
    ssl profile restconf
```

4. Настраиваем SSL

```
management security
    ssl profile restconf
    certificate restconf.crt key restconf.key
```

5. Вы божественны

Теперь проверяем, что порт открыт

```
nc -zv bcn-spine-1.arista 6020
Connection to bcn-spine-1.arista 6020 port [tcp/*] succeeded!
```

И собственно курлим:

```
curl -k -s GET 'https://bcn-spine-1.arista:6020/restconf/data/openconfig-
↪interfaces:interfaces/interface=Management1' \
     --header 'Accept: application/yang-data+json' \
     -u eucariot:password
```

Так мы извлекли информацию про интерфейс `Management1`.

А вот так можно получить данные по BGP:

```
curl -k -s GET 'https://bcn-spine-1.arista:6020/restconf/data/network-instances/network-
↪instance/config/protocols' \
     --header 'Accept: application/yang-data+json' \
     -u eucariot:password | jq
```

Строка URL формируется следующим образом:

```
https://<ADDRESS>/<ROOT>/data/<[YANG-MODULE]:CONTAINER>/<LEAF>/[?<OPTIONS>]
```

- <**ADDRESS**> - адрес RESTCONF-сервера.
- <**ROOT**> - Точка входа для запросов RESTCONF. Можно найти тут: `https://<ADDRESS>/.well-known/`
- **data** - прям так и остаётся
- <**[YANG MODULE:]CONTAINER**> - Базовый контейнер YANG. Наличие YANG Module - не обязательно.
- <**LEAF**> - Отдельный элемент в контейнере
- <**OPTIONS**> - Опциональные параметры, влияющие на результат.

Пробуем выяснить `<ROOT>`:

```
curl -k https://bcn-spine-1.arista:6020/.well-known/host-meta
<XRD xmlns='http://docs.oasis-open.org/ns/xri/xrd-1.0'>
    <Link rel='restconf' href='/restconf'/>
</XRD>
```

Ну можно и настроить что-нибудь:
К примеру hostname.

```
curl -k -X PUT https://bcn-spine-1.arista:6020/restconf/data/system/config \
     -H 'Content-Type: application/json' -u eucariot:password \
     -d '{"openconfig-system:hostname":"vika-kristina-0"}'

{"openconfig-system:hostname":"vika-kristina-0"}
```

Проверим?

```
curl -k -X GET https://bcn-spine-1.arista:6020/restconf/data/system/config \
     --header 'Accept: application/yang-data+json' \
     -u eucariot:password

{"openconfig-system:hostname":"bcn-spine-1","openconfig-system:login-banner":"",
↪"openconfig-system:motd-banner":""}
```

Что? Не поменялось?! И оно действительно не поменялось. Я не смог заставить это работать.

В общем знакомство с RESTCONF пока скорее травматично: документации исчезающие мало, большая часть ссылок - на космические корабли, бороздящие просторы неизученной Вселенной, примеры работы с RESTCONF все как один однообразны, а некоторые просто не работают. С той же аристой использование разных моделей - ietf, openconfig приводит к одному ответу в виде OpenConfig.
В конце концов отсутствие в выдаче хоть сколько-то серьёзных работ по автоматизации с помощью RESTCONF говорит о том, что это всё не более чем баловство. И я намеренно не пишу слово «пока». Лично я в него не верю

Хотя ощутимые удобства присутствуют - это использование чуть более привычного интерфейса и существующих библиотек. И с точки зрения разработчика несколько проще - он теперь имеет дело со знакомым с пелёнок WEB-сервисом.
При этом CRUD не очень гладко ложится на RPC-подход, да и в идее держать открытым на сетевом железе HTTP есть что-то противоестественное, согласитесь?

Просто жаль сил, вложенных в этот протокол. Потому что на пятки ему наступает gRPC/gNMI.

На самостоятельное изучение: RESTCONF intro with Postman - Part 1.

7.7 Call-Home

RFC8071.

Это, что называется, звонок домой - способ инициировать соединение с NETCONF/RESTCONF-сервера к клиенту, то есть с сетевой коробки на систему управления.

На устройстве настраивается IP-адрес NETCONF/RESTCONF-клиента, куда оно отсылает периодически данные по своему состоянию. Либо обращается для того, чтобы зарегистрироваться в системе и забрать свою конфигурацию.

Применимо для сценариев, когда

- Новое устройство должно сообщить о себе в систему управления
- Устройство находится за NAT или фаерволом
- Администратор считает, что безопаснее иметь закрытые порты на сетевых элементах и открывать только well-known порт на системе управления

Подробно тут останавливаться не будем.

7.8 gRPC/gNMI

За последние лет семь gRPC уже всем уши прожужжали. И только самые ловкие разработчики могли избежать реализации взаимодействия с какой-нибудь системой по gRPC.

g в gRPC, кстати, означает вовсе не *«google»*.

Реализация фреймворка поверх gRPC в мире сетевой автоматизации получила название **gNMI** - *gRPC Network Management Interface*.
В основе gNMI лежит gRPC, для моделирования данных использует YANG (но не обязательно), внутри уже определяются конкретные RPC. Кроме того gNMI изначально предоставляет возможность естественным образом реализовать telemetry - потоковую передачу телеметрических данных.

В любом случае я не я, если перед gNMI я не разберу gRPC. Поэтому простите за отступление, но без него статья превратится в бесполезное поверхностное хауту.

7.8.1 gRPC

Без теории - за ней прошу в пятую часть АДСМ.
В любом случае после голой теории вот только такие ощущения:

> Есть, правда, и более последовательная инструкция.

Хотя точно стоит сказать о том, что gRPC использует **Protocol Buffers** (или **коротко protobuf**). Термин этот довольно нагруженный:

- это и спецификация, в которой описано, как данные должны выглядеть. Ещё это называется прото-спека.
- это и IDL (Interface Definition Language), позволяющим разным системам друг с другом на одном языке общаться
- это и формат сериализованных данных, в котором информация передаётся между системами
- То есть всего лишь один proto-файл (или их набор), определяет сразу все эти три вещи

- То есть когда вы пишете gRPC-приложение, формирование protobuf - это важнейший шаг.

Пишем ping!

Спецификация

Описываем protobuf:

```
service Ping {
  rpc SendPingReply (PingRequest) returns (PingReply) {}
}
```

Сначала определяем сервис - `Ping`. А в нём есть метод - `SendPingReply` - это собственно и есть RPC - та самая процедура, которую мы дёрнем удалённо - процедура отправить `Ping Reply`.
В качестве атрибута она принимает параметр *PingRequest*, а вернёт ответ `PingReply`.

А что такое эти `PingRequest` и `PingReply`??

```
message PingRequest {
  string payload = 1;
}
```

`PingRequest` - это одно из пересылаемых сообщений между клиентом и сервером.

Так объявляется факт его существования, и его содержимое. В этом случае внутри сообщения передаётся одно поле `payload` типа `string`.

`payload` - это произвольное имя, которое мы можем выбрать, как хотим.

`string` - определение типа.

`1` - позиция поля в сообщении - для нас не имеет значения.

```
message PingReply {
  string message = 1;
}
```

Всё точно то же самое. Именем поля может быть даже слово message.

Вот так будет выглядеть полный proto-файл:

```
syntax = "proto3";

option go_package = "go-server/ping";

package ping;

// The ping service definition.
service Ping {
  // Sends a ping reply
  rpc SendPingReply (PingRequest) returns (PingReply) {}
}

// The request message containing the ping payload.
message PingRequest {
  string payload = 1;
}

// The response message containing the ping replay
message PingReply {
  string message = 1;
}
```

То есть именно вот так и выглядит спецификация, описывающая схему данных на обеих сторонах. И сервер и клиент будут использовать один и тот же proto-файл и всегда знать, как разобрать то, что отправила другая сторона. Даже если они написаны на разных языках.

Сохраняем как `protos/ping.proto` - он будет один для всех.
Ну ладно спецификация есть. И что с ней теперь делать?

А теперь мы напишем пинг-клиент на Python, а пинг-сервер на Go.

gRPC Client

Сгенерированный Код

Создадим директорию `python-client`.
Далее на основе спецификации сгенерируем код.

Для этого нужно будет установить `grpcio-tools`.

```
pip install grpcio-tools
```

И используя его уже нагенерить нужные классы:

```
python3 -m grpc_tools.protoc \
        -I protos \
        --python_out=python-client \
        --grpc_python_out=python-client \
        protos/ping.proto
```

Сразу после этого в каталоге, где мы это выполнили, появятся два файла: `ping_pb2.py` и `ping_pb2_grpc.py` - это сгенерированный код.
Если вы зяглянете вовнутрь, то обнаружите там кучу классов. Это классы, реализующие сообщения, сервисы для сервера (`PingServicer`) и для клиента (`PingStub`). Там же у класса `Ping` есть и метод `SendPingReply`. И куча других штуковин.
Эти файлы нам никогда не придётся менять вручную - мы будем их только импортировать и использовать.

Очевидно, что эти py-файлы это только реализация интерфейса взаимодействия. Ровным счётом ничего тут не говорит, как этот сервис будет работать.
Бизнес-логика описывается уже отдельно - и вот она делается нами.

Пока структура выглядит так:

```
.
├── ping_client.py
├── ping_pb2.py
└── ping_pb2_grpc.py
```

Давайте писать gRPC-клиент.

Клиент будет совсем бесхитростным. В цикле он будет пытаться выполнить RPC `SendPingReply` на удалённом хосте `84.201.157.17:12345`. В качестве аргумента передаём payload, который считали из аргументов запуска скрипта.

В функции `run` мы устанавливаем соединение к серверу, подключаем `stub` и выполняем RPC `SendPingReply`, которому передаём сообщение `PingRequest` с тем самым `payload`.

```
import sys
import time
from datetime import datetime

import grpc

import ping_pb2
import ping_pb2_grpc

server = "84.201.157.17:12345"


```

(continues on next page)

(продолжение с предыдущей страницы)

```
def run(payload) -> None:
    with grpc.insecure_channel(server) as channel:
        stub = ping_pb2_grpc.PingStub(channel)
        start_time = datetime.now()
        response = stub.SendPingReply(ping_pb2.PingRequest(payload=payload))
        rtt = round((datetime.now() - start_time).total_seconds()*1000, 2)
    print(f"Ping response received: {response.message} time={rtt}ms")


if __name__ == "__main__":
    payload = sys.argv[1]

    while True:
        run(payload)
        time.sleep(1)
```

Если запустить его сейчас, клиент вернёт `StatusCode.UNAVAILABLE` - сервера пока нет, порт 12345 никто не слушает.

Давайте теперь писать

gRPC-сервер

на Go. Я его развернул на облачной виртуалочке, поэтому какое-то время он будет доступен и читателям.

Всё, что делает сервер - получает какую-то строку в `payload`, добавляет к нему *«-pong»* и возвращает это клиенту.

Сгенерированный Код

Тут нам тоже понадобится дополнительный код, реализующий интерфейс.
Создаём рабочую директорию *go-server*, внутри которой ещё `ping` - для хранения спецификации и кода интерфейса.

```
protoc --go_out=. --go-grpc_out=.  protos/ping.proto
```

И получается так:

```
.
├── go.mod
├── go.sum
└── ping
    ├── ping_grpc.pb.go
    ├── ping.pb.go
    └── ping.proto
```

Дальше сам код сервера. Я его тоже взял из примеров для go.
Мы тут опускаем часть про установку go, protoc, потому что это всё есть в документации grpc.io.

```
package main

import (
    "context"
    "flag"
    "fmt"
    "log"
    "net"

    "google.golang.org/grpc"
    pb "ping-server/ping"
)

var (
    port = flag.Int("port", 12345, "The server port")
)

type server struct {
    pb.UnimplementedPingServer
}

func (s *server) SendPingReply(ctx context.Context, in *pb.PingRequest) (*pb.PingReply,␣
↪error) {
    log.Printf("Received: %v", in.GetPayload())
    return &pb.PingReply{Message: in.GetPayload() + "-pong"}, nil
}

func main() {
    flag.Parse()
    lis, err := net.Listen("tcp", fmt.Sprintf("10.128.0.6:%d", *port))
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }
    s := grpc.NewServer()
    pb.RegisterPingServer(s, &server{})
    log.Printf("server listening at %v", lis.Addr())
    if err := s.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}
```

Вся бизнес логика описана в функции `SendPingReply`, ожидающей `PingRequest`, а возвращающей `PingReply`, в котором мы отправляем сообщение `message`: `payload` + *«-pong»* (`GetPayload`). Естественно, там может быть более изощрённая логика.

Ну, а в `main` мы запускаем сервер на адресе `10.128.0.6`.

Почему не на `84.201.157.17`, на который стучится клиент? Тут без хитростей - это внутренний адрес ВМ, на который натируются все запросы к публичному адресу.

Я положу его в

```
.
└── ping-server
    └── main.go
```

```
$ go run ping-server/main.go
2022/01/30 04:26:11 server listening at 10.128.0.6:12345
```

Всё, сервер готов слушать.

Пример сервера на питоне, если хочется попробовать.

Используем сразу asyncio, это же сервер, нельзя тут блочиться.

Для того, чтобы запустить сервер, нужно доставить пакет grpcio

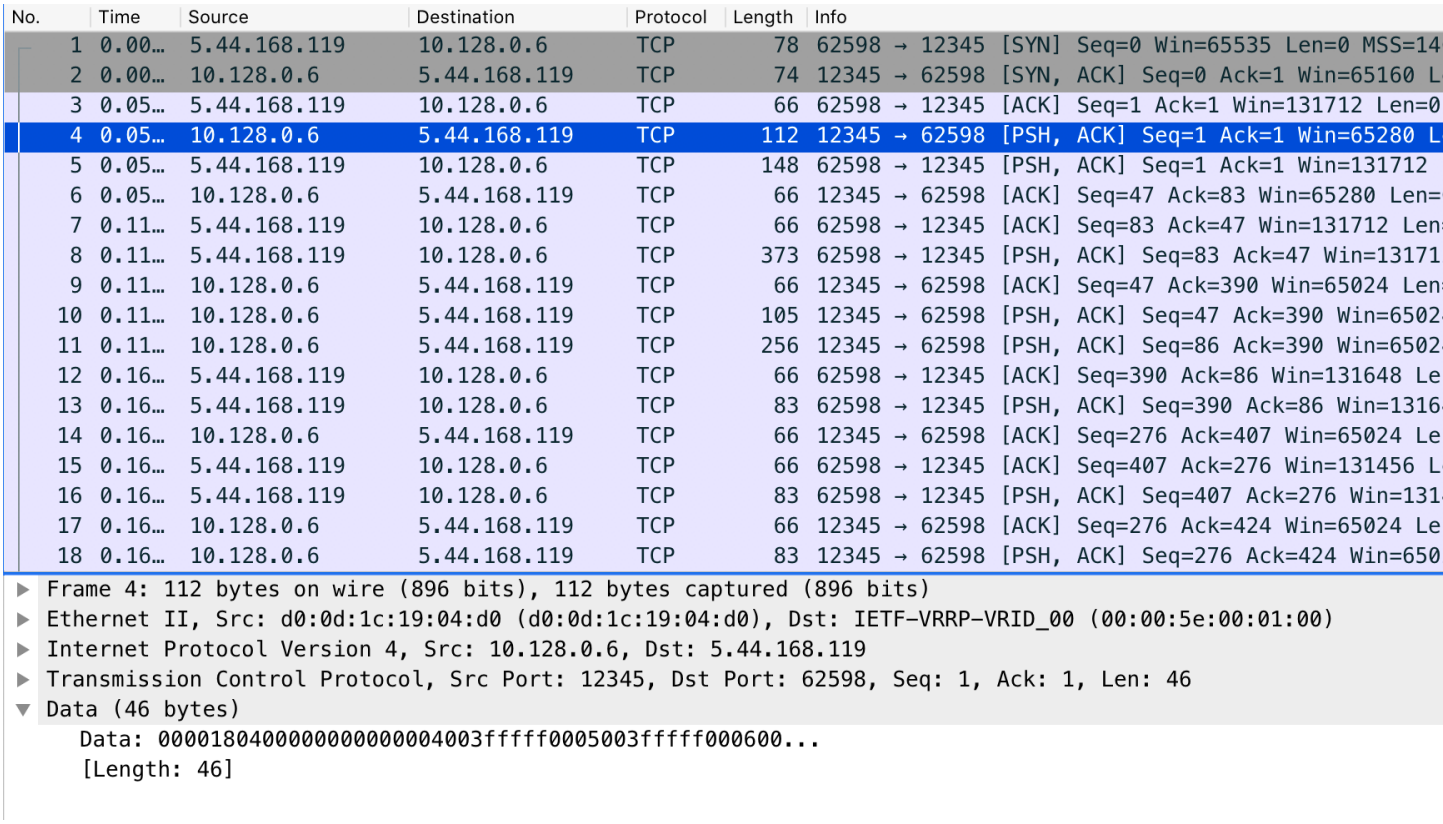
```
python -m pip install grpcio
```

Запускаем?

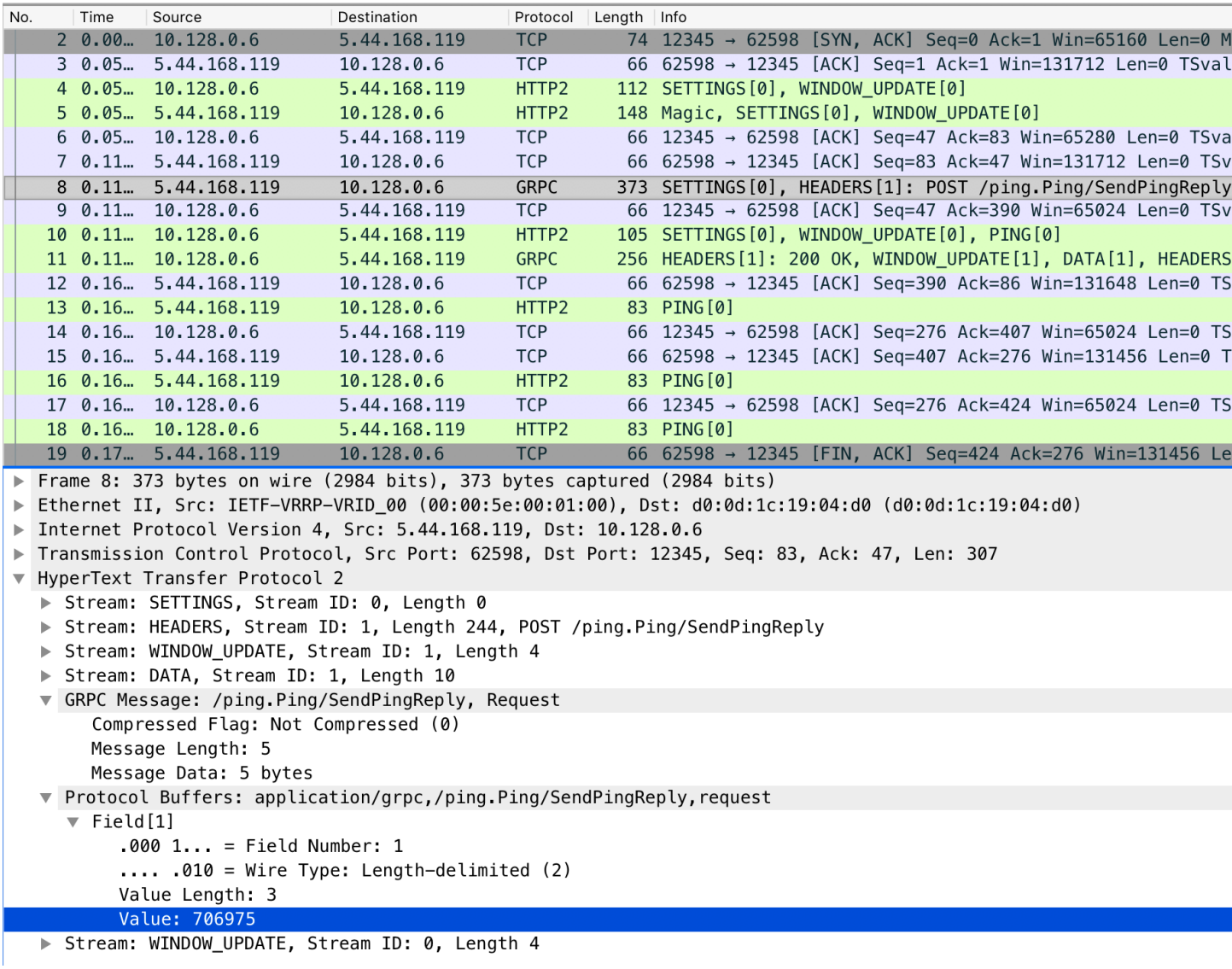
```
 python ping_client.py piu
Ping response received: piu-pong time=208.13ms
Ping response received: piu-pong time=165.62ms
Ping response received: piu-pong time=162.89ms
```

У-хууу, ё-моё, grpc-заработал!!!!

А давайте теперь попробуем подампать трафик? Я запустил сервер удалённо и снял трафик.



По умолчанию, Wireshark не декодирует HTTP2, давайте научим его?
Analyze -> Decode As.



No.	Time	Source	Destination	Protocol	Length	Info
2	0.00…	10.128.0.6	5.44.168.119	TCP	74	12345 → 62598 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 M
3	0.05…	5.44.168.119	10.128.0.6	TCP	66	62598 → 12345 [ACK] Seq=1 Ack=1 Win=131712 Len=0 TSval
4	0.05…	10.128.0.6	5.44.168.119	HTTP2	112	SETTINGS[0], WINDOW_UPDATE[0]
5	0.05…	5.44.168.119	10.128.0.6	HTTP2	148	Magic, SETTINGS[0], WINDOW_UPDATE[0]
6	0.05…	10.128.0.6	5.44.168.119	TCP	66	12345 → 62598 [ACK] Seq=47 Ack=83 Win=65280 Len=0 TSva
7	0.11…	5.44.168.119	10.128.0.6	TCP	66	62598 → 12345 [ACK] Seq=83 Ack=47 Win=131712 Len=0 TSv
8	0.11…	5.44.168.119	10.128.0.6	GRPC	373	SETTINGS[0], HEADERS[1]: POST /ping.Ping/SendPingReply
9	0.11…	10.128.0.6	5.44.168.119	TCP	66	12345 → 62598 [ACK] Seq=47 Ack=390 Win=65024 Len=0 TSv
10	0.11…	10.128.0.6	5.44.168.119	HTTP2	105	SETTINGS[0], WINDOW_UPDATE[0], PING[0]
11	0.11…	10.128.0.6	5.44.168.119	GRPC	256	HEADERS[1]: 200 OK, WINDOW_UPDATE[1], DATA[1], HEADERS
12	0.16…	5.44.168.119	10.128.0.6	TCP	66	62598 → 12345 [ACK] Seq=390 Ack=86 Win=131648 Len=0 TS
13	0.16…	5.44.168.119	10.128.0.6	HTTP2	83	PING[0]
14	0.16…	10.128.0.6	5.44.168.119	TCP	66	12345 → 62598 [ACK] Seq=276 Ack=407 Win=65024 Len=0 TS
15	0.16…	5.44.168.119	10.128.0.6	TCP	66	62598 → 12345 [ACK] Seq=407 Ack=276 Win=131456 Len=0 T
16	0.16…	5.44.168.119	10.128.0.6	HTTP2	83	PING[0]
17	0.16…	10.128.0.6	5.44.168.119	TCP	66	12345 → 62598 [ACK] Seq=276 Ack=424 Win=65024 Len=0 TS
18	0.16…	10.128.0.6	5.44.168.119	HTTP2	83	PING[0]
19	0.17…	5.44.168.119	10.128.0.6	TCP	66	62598 → 12345 [FIN, ACK] Seq=424 Ack=276 Win=131456 Le

```
▶ Frame 8: 373 bytes on wire (2984 bits), 373 bytes captured (2984 bits)
▶ Ethernet II, Src: IETF-VRRP-VRID_00 (00:00:5e:00:01:00), Dst: d0:0d:1c:19:04:d0 (d0:0d:1c:19:04:d0)
▶ Internet Protocol Version 4, Src: 5.44.168.119, Dst: 10.128.0.6
▶ Transmission Control Protocol, Src Port: 62598, Dst Port: 12345, Seq: 83, Ack: 47, Len: 307
▼ HyperText Transfer Protocol 2
  ▶ Stream: SETTINGS, Stream ID: 0, Length 0
  ▶ Stream: HEADERS, Stream ID: 1, Length 244, POST /ping.Ping/SendPingReply
  ▶ Stream: WINDOW_UPDATE, Stream ID: 1, Length 4
  ▶ Stream: DATA, Stream ID: 1, Length 10
  ▼ GRPC Message: /ping.Ping/SendPingReply, Request
      Compressed Flag: Not Compressed (0)
      Message Length: 5
      Message Data: 5 bytes
  ▼ Protocol Buffers: application/grpc,/ping.Ping/SendPingReply,request
    ▼ Field[1]
        .000 1... = Field Number: 1
        .... .010 = Wire Type: Length-delimited (2)
        Value Length: 3
        Value: 706975
  ▶ Stream: WINDOW_UPDATE, Stream ID: 0, Length 4
```

Вот тут уже видно почти все наши объекты, которые передаются между клиентом и сервером. pcap-файл.

Кайф!!

Давайте ещё раз проговорим, что мы сделали.

1. Описали спецификацию сервиса - какие методы доступны, какими сообщениями с какими полями они обмениваются.
2. Сгенерировали из этой спецификации код как для сервера на Go, так и для клиента на Python.
3. Написали логику сервера и клиента
4. Клиент сделал вызов удалённого метода на сервере. Список доступных методов мы знаем из proto-файла.
5. Сервер вернул результат работы процедуры клиенту.

Весь код в репозитории.

Итак, разобрались с gRPC. Ну, будем так считать, по крайней мере.

Внутри гугла gRPC удалось адаптировать даже к задачам сети. То есть gRPC стал единым интерфейсом взаимодействия между разными компонентами во всей компании (или одним из - мы не знаем).

7.8.2 gNMI

gNMI довольно новый протокол. Про него нет страницы на вики, довольно мало материалов и мало кто рассказывает о том, как его использует в своём проде.

Он не является стандартом согласно любым организациям и RFC, но его спецификация описана на гитхабе.

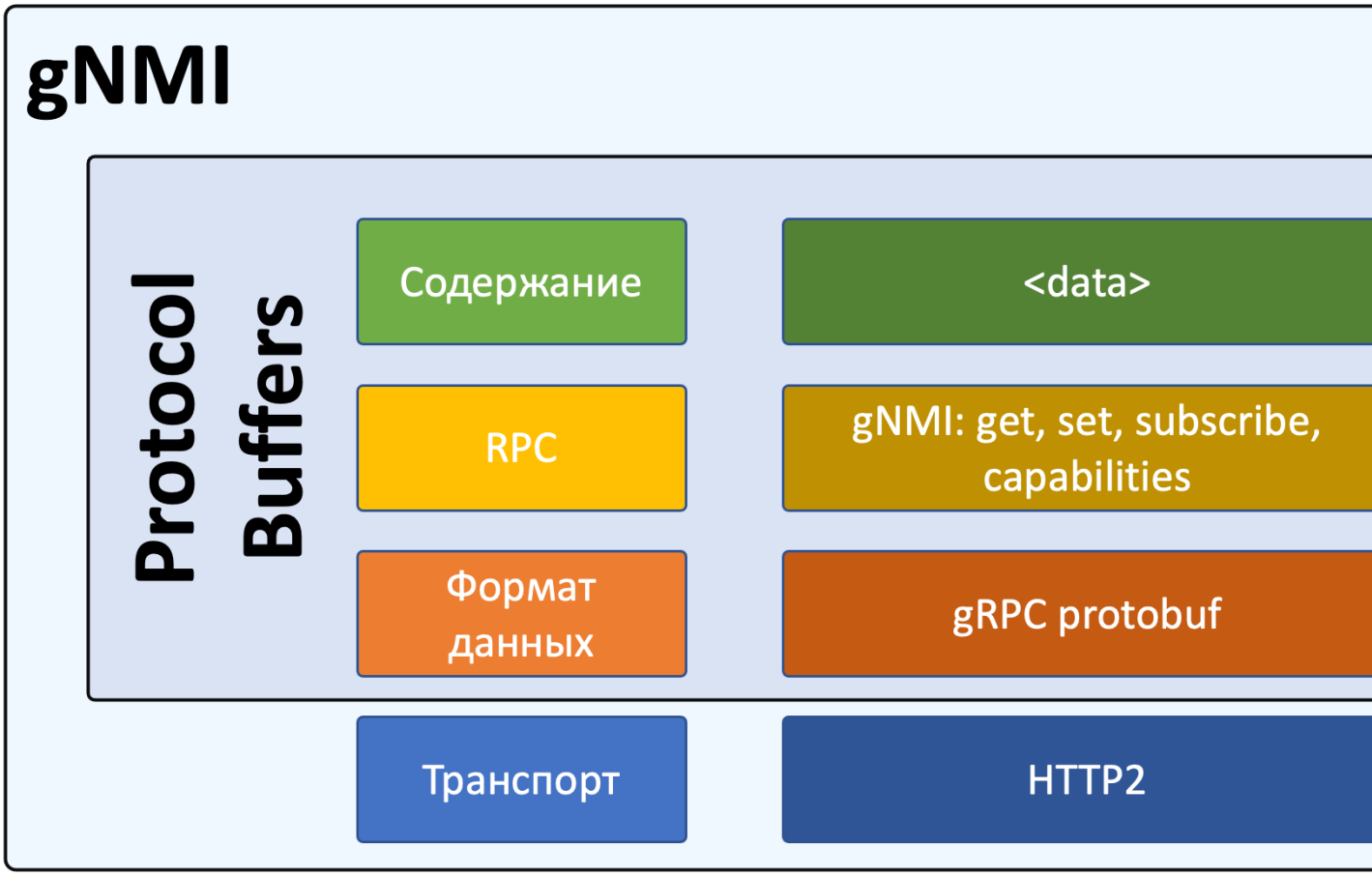
Что нам важно знать о нём для начала? *gRPC Network Management Interface.*
Это протокол управления сетевыми устройствами, использующий gRPC как фреймворк: транспорт, режимы взаимодействия (унарный и все виды стриминга), механизмы маршаллинга данных, прото-файлы для описания спецификаций.

В качестве модели данных он может использовать YANG (а может и не использовать - в протобафы можно же засунуть всё, что угодно).

Как того требует gRPC, на сетевом устройстве запускается сервер, на системе управления - клиент. На обеих сторонах должна быть одна спецификация, одна модель данных.

Поскольку это конструкт над gRPC, в нём определены конкретные сервисы и RPC:

```
servicegNMI{
  rpcCapabilities(CapabilityRequest) returns(CapabilityResponse);
  rpcGet(GetRequest) returns(GetResponse);
  rpcSet(SetRequest) returns(SetResponse);
  rpcSubscribe(streamSubscribeRequest) returns(streamSubscribeResponse);
}
```


gNMI
Protocol Buffers
Содержание
<data>
RPC
gNMI: get, set, subscribe, capabilities
Формат данных
gRPC protobuf
Транспорт
HTTP2

Более наглядное представление полного прото-файла можно увидеть на интерактивной карте, которую нарисовал Роман Додин:

> Картинка побольше

Здесь каждый RPC расписан на сообщения и типы данных, и указаны ссылки на прото-спеки и документацию. Не могу сказать, что это удобное место для того, чтобы начать знакомиться с gNMI, но вы точно к нему ещё много раз вернётесь, если сядете на gNMI.

Предлагаю попробовать на практике вместо теорий.

Вообще gNMI, как плоть от плоти gRPC не очень удобен для использования человеком.
Прото-файлы пиши, код пиши, исполняй. Нельзя как в REST API просто curl отправить и получить текстовый ответ - это вообще известная боль.
Но для gNMI напридумывали клиентов.

И тут google в лучших традициях делает прекрасные инфраструктурные вещи и ужасный пользовательский интерфейс. gNXI, OpenConfig gNMI CLI client.

gNMIc

Нас и тут спасает Роман Додин, поучаствоваший в создании классного клиента gNMI, совместно с Karim Radhouani - gNMIc.

Устанавливаем по инструкции:

```
bash -c "$(curl -sL https://get-gnmic.kmrd.dev)"
```

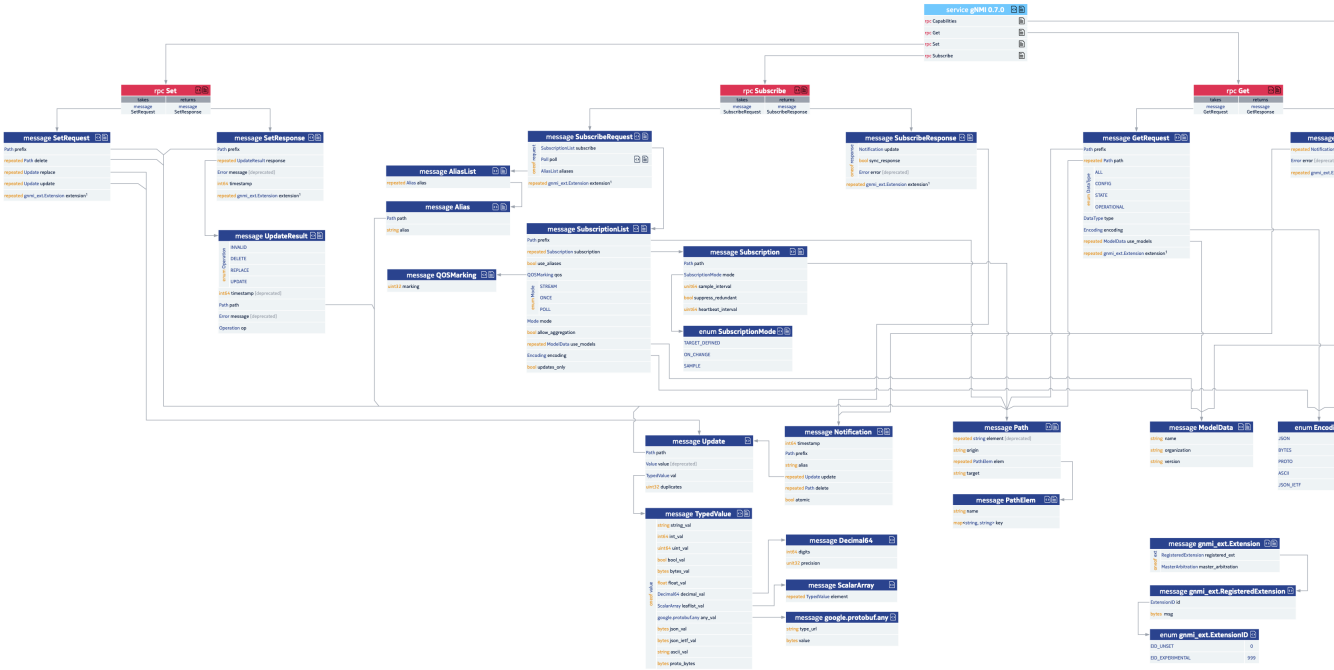
Теперь надо настроить узел.

```
interface Management1
   ip address 192.168.1.11/24

username eucariot secret <SUPPASECRET>

management api gnmi
   transport grpc default

ip access-list control-plane-acl-with-restconf-and-gnmi
   8 permit tcp any any eq 6030
...

control-plane
   ip access-group control-plane-acl-with-restconf-and-gnmi in
```

Попробуем выяснить capabilities:

```
gnmic capabilities \
      -a bcn-spine-1.arista:6030 \
      -u eucariot \
      -p password \
      --insecure
```

А в ответ пара экранов текста, полного возможностей:


service gNMI 0.7.0
rpc Capabilities
rpc Get
rpc Set
rpc Subscribe
rpc Set
rpc Subscribe
rpc Get
message SetRequest
message SetResponse
message UpdateResult
message SubscribeRequest
message SubscribeResponse
message AliasList
message Alias
message QOSMarking
message SubscriptionList
message Subscription
enum SubscriptionMode
message GetRequest
message Update
message TypedValue
message Notification
message Decimal64
message ScalarArray
message google.protobuf.any
message Path
message PathElem
message ModelData
message gnmi_ext.Extension
message gnmi_ext.RegisteredExtension
enum gnmi_ext.ExtensionID

```
gNMI version: 0.6.0
supported models:
  - arista-exp-eos-multicast, Arista Networks <http://arista.com/>,
  - arista-exp-eos, Arista Networks <http://arista.com/>,
  - openconfig-if-ip, OpenConfig working group, 2.3.0
...
supported encodings:
  - JSON
  - JSON_IETF
  - ASCII
```

Тут видно, что устройство поддерживает три вида кодирования. Нам интересен JSON.
А так же, несколько десятков моделей данных, как OpenConfig, так и IETF и проприетарные.
Дальше нет времени объяснять, откуда я это взял, просто пробуем собрать IP-адреса всех интерфейсов:

```
gnmic get \
      --path "/interfaces/interface/subinterfaces/subinterface/ipv4/addresses/address/
↪config"\
      -a bcn-spine-1.arista:6030 \
      -u eucariot \
      -p password \
      --insecure
```

```
[
  {
    "source": "bcn-spine-1.arista:6030",
    "time": "1969-12-31T16:00:00-08:00",
    "updates": [
      {
                "Path": "interfaces/interface[name=Management1]/subinterfaces/
↪subinterface[index=0]/ipv4/addresses/address[ip=192.168.1.11]/config",
        "values": {
          "interfaces/interface/subinterfaces/subinterface/ipv4/addresses/address/config
↪": {
            "openconfig-if-ip:ip": "192.168.1.11",
            "openconfig-if-ip:prefix-length": 24
          }
        }
      },
      {
                "Path": "interfaces/interface[name=Ethernet3]/subinterfaces/
↪subinterface[index=0]/ipv4/addresses/address[ip=169.254.101.1]/config",
        "values": {
          "interfaces/interface/subinterfaces/subinterface/ipv4/addresses/address/config
↪": {
            "openconfig-if-ip:ip": "169.254.101.1",
            "openconfig-if-ip:prefix-length": 31
          }
        }
      },
      {
                "Path": "interfaces/interface[name=Ethernet2]/subinterfaces/
↪subinterface[index=0]/ipv4/addresses/address[ip=169.254.1.3]/config",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        "values": {
          "interfaces/interface/subinterfaces/subinterface/ipv4/addresses/address/config
↪": {
            "openconfig-if-ip:ip": "169.254.1.3",
            "openconfig-if-ip:prefix-length": 31
          }
        }
      }
    ]
  }
]
```

Из ответа видно полный путь к каждому интерфейсу, запрошенный путь и результат в модели OpenConfig.

Один ультра-полезный аргумент в gNMIc, это `--path "/"` - он вернёт просто всё, что может.
Полезен он тем, что можно из вывода пореверсинжинирить где что искать.

```
gnmic get \
      --path "/" \
      -a bcn-spine-1.arista:6030 \
      -u eucariot \
      -p password \
      --insecure
```

Ответа будет много.

И оттуда можно понять, что посмотреть конфигурацию BGP-пиров можно, используя путь `"/network-instances/network-instance/protocols/protocol/bgp/neighbors/neighbor/config"`:

```
gnmic get \
      --path "/network-instances/network-instance/protocols/protocol/bgp/neighbors/
↪neighbor/config" \
      -a bcn-spine-1.arista:6030 \
      -u eucariot \
      -p password \
      --insecure
```

```
[
  {
    "source": "bcn-spine-1.arista:6030",
    "time": "1969-12-31T16:00:00-08:00",
    "updates": [
      {
                "Path": "network-instances/network-instance[name=default]/protocols/
↪protocol[identifier=BGP][name=BGP]/bgp/neighbors/neighbor[neighbor-address=169.254.1.
↪2]/config",
        "values": {
          "network-instances/network-instance/protocols/protocol/bgp/neighbors/neighbor/
↪config": {
            "openconfig-network-instance:auth-password": "",
            "openconfig-network-instance:description": "",
            "openconfig-network-instance:enabled": true,
            "openconfig-network-instance:local-as": 0,
```

(continues on next page)

(продолжение с предыдущей страницы)


```
            "openconfig-network-instance:neighbor-address": "169.254.1.2",
            "openconfig-network-instance:peer-as": 4228186112,
            "openconfig-network-instance:peer-group": "LEAFS",
            "openconfig-network-instance:route-flap-damping": false,
            "openconfig-network-instance:send-community": "NONE"
          }
        }
      },
      {
              "Path": "network-instances/network-instance[name=default]/protocols/
↪protocol[identifier=BGP][name=BGP]/bgp/neighbors/neighbor[neighbor-address=169.254.
↪101.0]/config",
        "values": {
          "network-instances/network-instance/protocols/protocol/bgp/neighbors/neighbor/
↪config": {
            "openconfig-network-instance:auth-password": "",
            "openconfig-network-instance:description": "",
            "openconfig-network-instance:enabled": true,
            "openconfig-network-instance:local-as": 0,
            "openconfig-network-instance:neighbor-address": "169.254.101.0",
            "openconfig-network-instance:peer-as": 0,
            "openconfig-network-instance:peer-group": "EDGES",
            "openconfig-network-instance:route-flap-damping": false,
            "openconfig-network-instance:send-community": "NONE"
          }
        }
      }
    ]
  }
]
```


А такой, чтобы проверить состояние пира: `"/network-instances/network-instance/protocols/protocol/bgp/neighbors/neighbor/state/session-state"`

```
gnmic get \
          --path "/network-instances/network-instance/protocols/protocol/bgp/neighbors/
↪neighbor/state/session-state" \
      -a bcn-spine-1.arista:6030 \
      -u eucariot \
      -p password \
      --insecure
```


```
[
  {
    "source": "bcn-spine-1.arista:6030",
    "time": "1969-12-31T16:00:00-08:00",
    "updates": [
      {
              "Path": "network-instances/network-instance[name=default]/protocols/
↪protocol[identifier=BGP][name=BGP]/bgp/neighbors/neighbor[neighbor-address=169.254.1.
↪2]/state/session-state",
        "values": {
              "network-instances/network-instance/protocols/protocol/bgp/neighbors/
↪neighbor/state/session-state": "ACTIVE"
        }
      },
```


(continues on next page)

(продолжение с предыдущей страницы)

```
      {
              "Path": "network-instances/network-instance[name=default]/protocols/
↪protocol[identifier=BGP][name=BGP]/bgp/neighbors/neighbor[neighbor-address=169.254.
↪101.0]/state/session-state",
        "values": {
              "network-instances/network-instance/protocols/protocol/bgp/neighbors/
↪neighbor/state/session-state": "ACTIVE"
        }
      }
    ]
  }
]
```

И получается, вполне очевидное деление на конфигурационные и операционные данные.

Вот данные по конфигурации ветки system:

```
gnmic get \
      --path "/system/config" \
      -a bcn-spine-1.arista:6030 \
      -u eucariot \
      -p password \
      --insecure
```

```
[
  {
    "source": "bcn-spine-1.arista:6030",
    "time": "1969-12-31T16:00:00-08:00",
    "updates": [
      {
        "Path": "system/config",
        "values": {
          "system/config": {
            "openconfig-system:hostname": "bcn-spine-1",
            "openconfig-system:login-banner": "",
            "openconfig-system:motd-banner": ""
          }
        }
      }
    ]
  }
]
```

А вот по состоянию:

```
gnmic get \
      --path "/system/state" \
      -a bcn-spine-1.arista:6030 \
      -u eucariot \
      -p password \
      --insecure
```

```
[
  {
    "source": "bcn-spine-1.arista:6030",
    "time": "1969-12-31T16:00:00-08:00",
```

(continues on next page)

(продолжение с предыдущей страницы)


```
    "updates": [
      {
        "Path": "system/state",
        "values": {
          "system/state": {
            "openconfig-system:boot-time": "164480684820",
            "openconfig-system:current-datetime": "2022-02-19T13:24:54Z+00:00",
            "openconfig-system:hostname": "bcn-spine-1",
            "openconfig-system:login-banner": "",
            "openconfig-system:motd-banner": ""
          }
        }
      }
    ]
  }
]
```


Ну, и последний практический пример в этой секции: настроим чего полезного на железке, `Set RPC`.

Сначала посмотрим значение AS у одного из BGP-пиров:

```
gnmic get \
          --path "/network-instances/network-instance/protocols/protocol/bgp/neighbors/
↪neighbor[neighbor-address=169.254.1.2]/config/peer-as" \
      -a bcn-spine-1.arista:6030 \
      -u eucariot \
      -p passowrd \
      --insecure
```


```
[
  {
    "source": "bcn-spine-1.arista:6030",
    "time": "1969-12-31T16:00:00-08:00",
    "updates": [
      {
               "Path": "network-instances/network-instance[name=default]/protocols/
↪protocol[identifier=BGP][name=BGP]/bgp/neighbors/neighbor[neighbor-address=169.254.1.
↪2]/config/peer-as",
        "values": {
               "network-instances/network-instance/protocols/protocol/bgp/neighbors/
↪neighbor/config/peer-as": 4228186112
        }
      }
    ]
  }
]
```


Теперь поменяем значение:

```
gnmic set \
          --update-path "/network-instances/network-instance[name=default]/protocols/
↪protocol[name=BGP]/bgp/neighbors/neighbor[neighbor-address=169.254.1.2]/config/peer-as
↪" \
      --update-value "4228186113" \
      -a bcn-spine-1.arista:6030 \
      -u eucariot \
```

(continues on next page)

(продолжение с предыдущей страницы)

```
      -p passowrd \
      --insecure
```


```
{
  "source": "bcn-spine-1.arista:6030",
  "timestamp": 1645281264572566754,
  "time": "2022-02-19T06:34:24.572566754-08:00",
  "results": [
    {
      "operation": "UPDATE",
      "path": "network-instances/network-instance[name=default]/protocols/
↪protocol[name=BGP]/bgp/neighbors/    neighbor[neighbor-address=169.254.1.2]/config/
↪peer-as"
    }
  ]
}
```


Проверяем ещё раз:

```
gnmic get \
           --path "/network-instances/network-instance/protocols/protocol/bgp/neighbors/
↪neighbor[neighbor-address=169.254.1.2]/config/peer-as" \
      -a bcn-spine-1.arista:6030 \
      -u eucariot \
      -p password \
      --insecure
```


```
[
  {
    "source": "bcn-spine-1.arista:6030",
    "time": "1969-12-31T16:00:00-08:00",
    "updates": [
      {
                "Path": "network-instances/network-instance[name=default]/protocols/
↪protocol[identifier=BGP][name=BGP]/bgp/neighbors/neighbor[neighbor-address=169.254.1.
↪2]/config/peer-as",
        "values": {
                "network-instances/network-instance/protocols/protocol/bgp/neighbors/
↪neighbor/config/peer-as": 4228186113
        }
      }
    ]
  }
]
```


Уиии! Я чуть не вскочил с места, когда получилось.

А ещё у gNMIc есть автокомплишн.

Ну нам бы сейчас полезно было бы посмотреть на примеры работы с кодом?
Но вместо того, чтобы всё делать руками, воспользуемся готовым инструментом.

Сам gNMIc может быть импортирован как зависимость в Go-программу, поскольку имеет зрелую подсистему API.

pyGNMI

Эта библиотека написана Антоном Карнелюком (и снова русский след). Заметно удобнее всего остального и активно развивается.

Да на неё даже ссылается Arista из своей Open Management.

Соберём capabilities:

```
#!/usr/bin/env python

from pygnmi.client import gNMIclient
import json

host = ("bcn-spine-1.arista", 6030)

if __name__ == "__main__":
    with gNMIclient(target=host, username="eucariot",
                    password="password", insecure=True) as gc:

        result = gc.capabilities()

    print(json.dumps(result))
```

По-get-аем что-нибудь:

```
#!/usr/bin/env python

from pygnmi.client import gNMIclient
import json

host = ("bcn-spine-1.arista", 6030)

if __name__ == "__main__":
          paths = ["openconfig-interfaces:interfaces/interface/subinterfaces/
↪subinterface/ipv4/addresses/address/config"]

    with gNMIclient(target=host, username="eucariot",
                    password="password", insecure=True) as gc:

        result = gc.get(path=paths, encoding='json')

    print(json.dumps(result))
```

Ну и осталось теперь что-то поменять, например, тот же hostname:

```
#!/usr/bin/env python

from pygnmi.client import gNMIclient
import json

host = ("bcn-spine-1.arista", 6030)

set_config = [
(
    "openconfig-system:system",
    {
            "config": {
```

(continues on next page)

(продолжение с предыдущей страницы)

```
                "hostname": "bcn-spine-1.barista-karatista"
            }
    }
)
]
if __name__ == "__main__":

    with gNMIclient(target=host, username="eucariot",
                    password="fpassword", insecure=True) as gc:

        result = gc.set(update=set_config)

    print(json.dumps(result))
```

```
python gc_set.py | jq
{
  "timestamp": 1645326686451002000,
  "prefix": null,
  "response": [
    {
      "path": "system",
      "op": "UPDATE"
    }
  ]
}
```

В репе ADSM можно найти пример по изменению BGP peer-as.

gNMIc и pyGNMI - это всего лишь частные инструменты для работы через gNMI. Ничто не мешает вам самим реализовать набор методов удобным образом.

Важно здесь заметить то, что у gNMI нет концепции Data Stores и как следствие функционала коммитов конфигурации - мы работаем с сервисом.

gNMI заставляет нас вывернуть привычный взгляд на сеть иголками внутрь. Мы к ней теперь должны относиться как к ещё одному сервису, которым можно легко управлять через единообразный интерфейс. Сам же gNMI обеспечивает транзакционность всех изменений, передаваемых в одном RPC.

Представьте себе, что вы пишите в базу данных и нужно потом сделать ещё коммит, чтобы эти изменения сохранить - звучит нелогично. Вот так и с сетью - транзакционность есть, коммитов - нет.

Для инфраструктурной команды сеть - это больше не какой-то свой собственный особенный мир, находящийся где-то там за высокой стеной CLI, окружённый рвами, заполненными проприетарным синтаксисом.

Нам следует разделить сетевое устройство, к которому мы всю жизнь относились как к чему-то в целостному, потому что покупаем сразу всё это в сборе, на следующие части:

- **железный хост** - коммутаторы и маршрутизаторы, со всеми их медными и оптическими проводочками, куском кремния под вентилятором и трансиверами,
- **операционная система** - софт, который управляет жизнью железа и запускаемыми приложениями,
- **приложения**, реализующие те или иные сервисы или доступ к ним - аутентификация, интерфейсы, BGP, VLAN'ы, или gNMI, дающий доступ к ним ко всем.

Да, влияние проблем на сетевом устройстве имеет больший охват. Да, можно оторвать себе доступ одним неверным движением. Да, поддержка целевого состояния на все 100% - всё ещё сложная задача.

Но чем, в конце концов это отличается от обычного Linux'а, на котором крутится сервис?

То есть сервисный интерфейс (gNMI, gRPC, REST, NETCONF) следует рассматривать как способ управления собственно сервисами, в то время как для обслуживания хоста никуда не девается SSH+CLI - для отладки, обновления, управления приложениями. Впрочем и тут есть Ansible, Salt. Вот только идеально для этого, чтобы сетевая железка стала по-настоящему открытой - с Linux'ом на борту.

Кроме того есть

7.8.3 gNOI

gRPC Network Operations Interface от OpenConfig - набор микросервисов, основанных на gRPC, позволяющих выполнять операционные команды на хостах.
Если проще, то можно запустить ping, traceroute, почистить разные таблицы, сделать Route Refresh BGP-соседу, скопировать файл - всё то, что относится не к конфигурации, а скорее к отладке и эксплуатации.

На самом деле там на сегодняшний день уже достаточно внушительный список операций.

А ещё по аналогии с gNMIc существует и gNOIc.

7.8.4 Telemetry

Начнем с того что под словом «телеметрия» каждый вендор может понимать свое. У Huawei своя реализация поверх gRPC (местами даже платная!), у Cisco есть Model-driven Telemetry (например Cisco Model-Driven Telemetry (MDT) Input Plugin), у Juniper тоже есть своя реализация - JTI. Последние два еще параллельно поддерживают gNMI.

Отсюда уже возникает некоторая путаница, с которой разберёмся пониже. Главное - суть одна - устройство само рассылает данные тем клиентам, которые оформили подписку на обновления. Таким образом система мониторинга не тратит такты и RTT на опрос (polling), а всего лишь получает данные на свой интерфейс.
И если частота опроса ограничена единицами-десятками раз в минуту, то стриминг телеметрии вообще нет - устройство может слать хоть по каждому обновлению.
Тут, конечно, нужно быть аккуратным с машстабированием - в случае опроса мы сами управляем тем, сколько данных нужно обработать системой. Объём работы всегда предсказуемый и линейно зависит от числа опрашиваемых узлов. Видим увеличение нагрузки - добавляем ещё экземпляров попрошаек и шардируем нагрузку.
И всё иначе в случае стриминга телеметрии - тут узлы сети могу напихать в коллектор столько, что он не в состоянии обработать. Думать про масштабирование и отказоустойчивость тут придётся до усиленного потовыделения.

Перечисленные выше вендорские телеметрии поддерживают "сжатый формат" протобафов - когда данные представлены в виде структуры, зависимой от вендора, модели и ПО. Для декодирования таких данных нужен специальный вендорский прото-файл. Например, такой. В gNMI же данные в универсальном виде. Это хорошо с точки зрения discovery, так как для запроса нужно найти только путь подписки, но сильно сказывается на размере данных и отсутствии структуры данных (*теоретически можно вывести из моделей).

gNMI чересчур свободен в выборе типа кодирования данных. Можно и Proto и ASCII и даже два вида JSON.

Не предусмотрена “дешевая” отправка. У джуна и у аристы в их собственных протоколах телеметрии есть отправка протобафов в UDP. Это очень дешево для устройства и коллектора и даже может быть реализовано прям на линейной карте (мониторинг микробёрстов, например). При столь частой отправке обновлений разовые потери UDP не страшны.

В gNMI такого нет, но что в нём сделано классно, так это возможность подписаться только на изменения. Рисовать графики только по изменениям так себе идея конечно, но в gNMI можно реализовать такой сценарий: подписываешься на определенные данные, получаешь все значения и записываешь их в свой кеш. Дальше получаешь изменения и опционально периодические полные срезы. Теперь можно периодически отсылать весь кеш в БД и рисовать аккуратные графики.

Если говорить в целом про сбор метрик, то есть смысл использовать то, что поддерживается вендором в первую очередь. SNMP - надежный как швейцарские часы, тонны библиотек для работы с ним, MIBы устоялись и его не стыдно использовать даже в 2022. gNMI крутой, но реализация может подкачать - неизбежны детские болячки типа отсутствия поддержки IPv6 и требования админских прав для получения метрик.

7.9 Модели данных

Обратили, кстати, внимание, что в примерах выше в вызовах не было ничего специфичного для вендора?

На самом деле некая неявная привязка есть - это пути, они могли бы отличаться для Аристы и Хуавэя. Но внимание на слово «openconfig» в этих путях. Что это? Что за Открытый конфиг?

Сложность с автоматизацией сети - она ведь в чём? В том, что прежде чем отправлять конфигурацию на устройство, человек должен сесть и прям-таки разобраться в структуре CLI или XML и руками накидать шаблоны для конфигурации.

Даже просто для того, чтобы настроить IP-адрес на интерфейсе, нужно знать иерархию секций конфигурации или конкретное поддерево XML.

А ещё выяснить, в каком формате надо передавать адрес: `fe80::1/64`, `fe80::1 64`, `fe80::1 link-local`, `address: fe80::1, mask: 64`, `FE8:0:0:0:0:0:0:1`, `0000111111101000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000001` или там вообще не поддерживается IPv6. И надо ли сначала как-то энейблить IPv6, а MTU заимствуется с интерфейса или для IPv6 отдельный?

И так для каждого вендора по отдельности. Знаете, сетевых автоматизаторов спасает только то, что они до этого лет 10 ели на завтрак циски да джуниперы - и как свои два пальца знают все тонкости CLI.

Оно же их и губит.

NETCONF поел овса из-за того, что не предложил никакой стандартизации для моделирования данных. Именно поэтому вендоры успели наплодить своих собственных, несовместимых моделей, про которые мы и поговорим ниже.

Собственно то, в какой иерархии представлена конфигурация - и есть модель данных. Говорится об этом или нет, но такая модель есть всегда и у любого интерфейса. Она может быть плоской или

иерархической, может быть простой или запутанной. Если бы её не было, то вы бы просто не смогли настроить устройство, а команды конфигурации могли бы видоизвиняться случайным образом. Говорят, в Router OS 7 подвезли такую функцию.

Так, мы знаем, что например, в случае Juniper нужно войти в контекст `system->login`, чтобы настроить нового пользователя, а формат команды будет `set <USERNAME> <OTHER PARAMETERS>`.
А настройка IP-адреса управления при этом будет происходить в контексте `interface -> em0 -> unit 0 -> family inet`. И так будет всегда. Во всяком случае на этой железке и этой версии софта.

То есть модель данных - это контракт между пользователем и операционной системой - как она интерпретирует переданные команды в зависимости от контекста.

Это верно для CLI, SNMP, NETCONF, gNMI и даже прямых вызовов чипового SDK.

Просто бОльшую часть истории нам не нужно было знать об этих моделях. Есть аксиома - у каждого вендора она своя. А мы в голове, сознательно или нет, её выстраивали, воссоздавали.
И вендор может менять эту модель по своему усмотрению от версии к версии. А мы как люди к этому адаптируем свою внутреннюю модель, приспосабливаемся - по законам эволюции.

7.9.1 Native

Так было всегда, но это поменялось с приходом автоматизации. Вендоры, как будто бы думали, что рост сетей можно поддерживать постоянным докидыванием людей на их настройку. Но людям это не нравилось, они начали писать инструменты автоматизации на perl'ax, php, python'ax с expect'ами, попытками отловить все возможные ответы CLI, правильно на них среагировать. Но количество скорби в этом мире только множилось. Все рано или поздно приходили к пониманию, что долго притворяться робот человеком не может.

Так и появились NETCONF и RESTCONF (так появлялся и SNMP). Они дали возможность работать со структурированными данными, а также создавать более явные контракты между клиентом и сервером.
Автор утилиты/библиотеки, опираясь на этот контракт, пишет код, а вендор обязуется принять данные, которые ему прислали. Если вы присылаете соответствующие контракту данные, а вендор говорит, что вы ерунду прислали, вы идёте в суд (в TAC).

Первые реализации NETCONF были настолько же закрытыми, как и сам CLI. У джуна - меньше, у циски - больше. У кого-то RPC перекладывались собственно в вызове CLI.
Но необходимость приводить это всё к каким-то явным схемам становилась всё очевиднее с каждым днём. К этому же подталкивал и расцвет NMS, берущих на вооружение NETCONF.

И так появились первые модели данных - NATIVE. У каждого вендора своя, но уже модель, и уже открытая. Вендоры с достаточно высокой социальной ответственностью выкладывают свои модели в публичный репозиторий.

Наличие модели позволяет уже как минимум не рыскать в попытках вслепую нащупать, как составить XML, а пойти, посмотреть, в каком виде ожидает данные коробка.

А при желании модель читать программно и руками даже ничего не делать.

7.9.2 Вендор-нейтральные модели

С этим уже можно было жить.
Инженерам нужно было чуть меньше думать об интерфейсах и форматах сообщений, но с глубоким вниманием подходить к содержимому сообщений всё ещё приходилось, оказывая разные знаки почтения разным вендорам.

При этом казалось бы - вся сеть - это конечный набор одинаковых сервисов, если выбросить всякие IGRP, HSRP, RRPP и прочие проприетарные выдумки. Ну, всем же нужен IP, OSPF, BGP? Всем нужна аутентификация на устройствах и SSH? Они не могут иметь очень уж принципиальные отличия, как минимум из-за необходимости поддерживать совместимость друг с другом и соответствия RFC.
Так почему мы делаем это сотней разных способов?

Настройка интерфейса:

Juniper:

```
configure
set interfaces ge-0/0/0 unit 0 family inet address 10.0.0.1/30
commit and-quit
```

Nokia:

```
router
interface "test"
address 10.0.0.1
port 1/1/1
no shutdown
exit
```

Cisco:

```
conf t
interface gigabitethernet1
ip address 10.0.0.1 255.255.255.252
no shut
exit
```

Настройка BGP:

Juniper:

```
configure
set routing-options router-id 10.0.0.1
set routing-options autonomous-system 65000
set protocols bgp group test type internal
set protocols bgp group test peer-as 65000
set protocols bgp group test neighbor 10.0.0.2 redistribute-connected
set policy-options policy-statement redistribute-connected from protocol direct
set policy-options policy-statement redistribute-connected then accept
commit and-quit
```

Nokia:

```
router
autonomous-system 6500
router-id 10.0.0.1
bgp group "ibgp"
 type internal
 neighbor 10.10.10.2
exit
```

Cisco:

```
conf t
router bgp 65000
 bgp router-id 10.0.0.1
 neighbor 10.0.0.2 remote-as 65000
 redistribute connected
exit
```

Сложность ведь не в транспорте и не в интерфейсе, а в модели данных. Сделать у каждого вендора Configuration State Management - одноразовая решаемая (а много где и решённая) задача. А вот договориться между всеми производителями, как должна выглядеть модель - так же сложно, как и любая другая задача, где людям нужно договориться.

Но ни один из зарождавшихся и выживших стандартов или не ставил целью унификацию вообще, или пытался поднять этот вопрос, но был выброшен в окно штаб-квартиры вендора.

Хотя вру. IETF предприняли отчасти успешную попытку написать универсальную модель.

IETF-модель

Ещё в 2014-м году были сделаны первые коммиты в её репозиторий.

С тех пор много накоммичено, но мало фактически сделано. Общепризнанно, что IETF -модель очень медленно развивается, у неё низкое покрытие, а архитектура - так себе.

С IETF-модели рекомендуют начинать, потому что она якобы проще, а уже потом переходить на OpenConfig, но как по мне - это напрасная трата времени.

Она мертворождённая и никому особо не нужна. Хотя вендоры поддерживают.

Заказчиков и пользователей беспокоила обрезанность модели и инертность IETF.

Но один в поле не воин - тысячи разрозненных автоматизаторов по всему миру не могли ничего с этим сделать. А вот большие компании могли.

Когда надо настроить тысячу свитчей, а каждый месяц запускать новый датацентр, когда на сети 5 разных поколений дизайна, а катить изменения нужно дважды в день, начинаешь несколько иначе смотреть на все этим ваши сиэлаи и вендор-специфичные эксэмали.

Так гугл придумал OpenConfig. Он не стал размениваться на IETF-модели и торги со стариканами из института.

OpenConfig - мечта, становящаяся явью

Возможно, впервые за шестидесятилетнюю историю телекоммуникаций у нас появился шанс изобрести свой USB Type C. Представьте мир, в котором Cisco, Juniper, Arista и Mikrotik настраиваются одними и теми же командами и это к тому же приводит к одинаковому результату?

Я не могу.

OpenConfig - это открытая YANG-модель, которая предполагается единой для всех вендоров. Одна стандартизированная модель для управления конфигурацией, сбора операционных данных с устройства и телеметрии. Одна для всех поддерживающих ОС вендоров.

Итак, OpenConfig появился в 2015 году в Google как ответ на следующие вызовы:

- 20+ ролей сетевых устройств
- Больше полудюжины вендоров
- Множество платформ
- 4M строк в конфигурационных файлах
- 30K изменений конфигураций в месяц
- Больше 8M OIDs опрашиваются каждые 5 минут
- Больше 20K CLI-команд выполняется каждые 5 минут
- Множество инструментов и поколений софта, куча скриптов
- Отсутствие абстракций и проприетарные CLI
- SNMP не был рассчитан на столь большое количество устройств и на столько большие объёмы данных (RIB)

Как работать с openconfig мы уже немного попрактиковались выше.
Полезным было бы взглянуть на структуру этой модели. Но это мы сделаем в следующей главе про YANG.

OpenConfig сегодня даёт возможность настройки базовых сервисов. Безусловно речь не идёт про вещи, завязанные на аппаратные особенности: QoS, управление буферами и ресурсами чипа, сплиты портов, работа с трансиверами. И в каком-то хоть сколько-то обозримом будущем этого ждать не стоит.

Хуже того, на сегодняшний день многие вендоры, ввязавшиеся в поддержку ОС, не реализуют все 100%, а лишь часть.

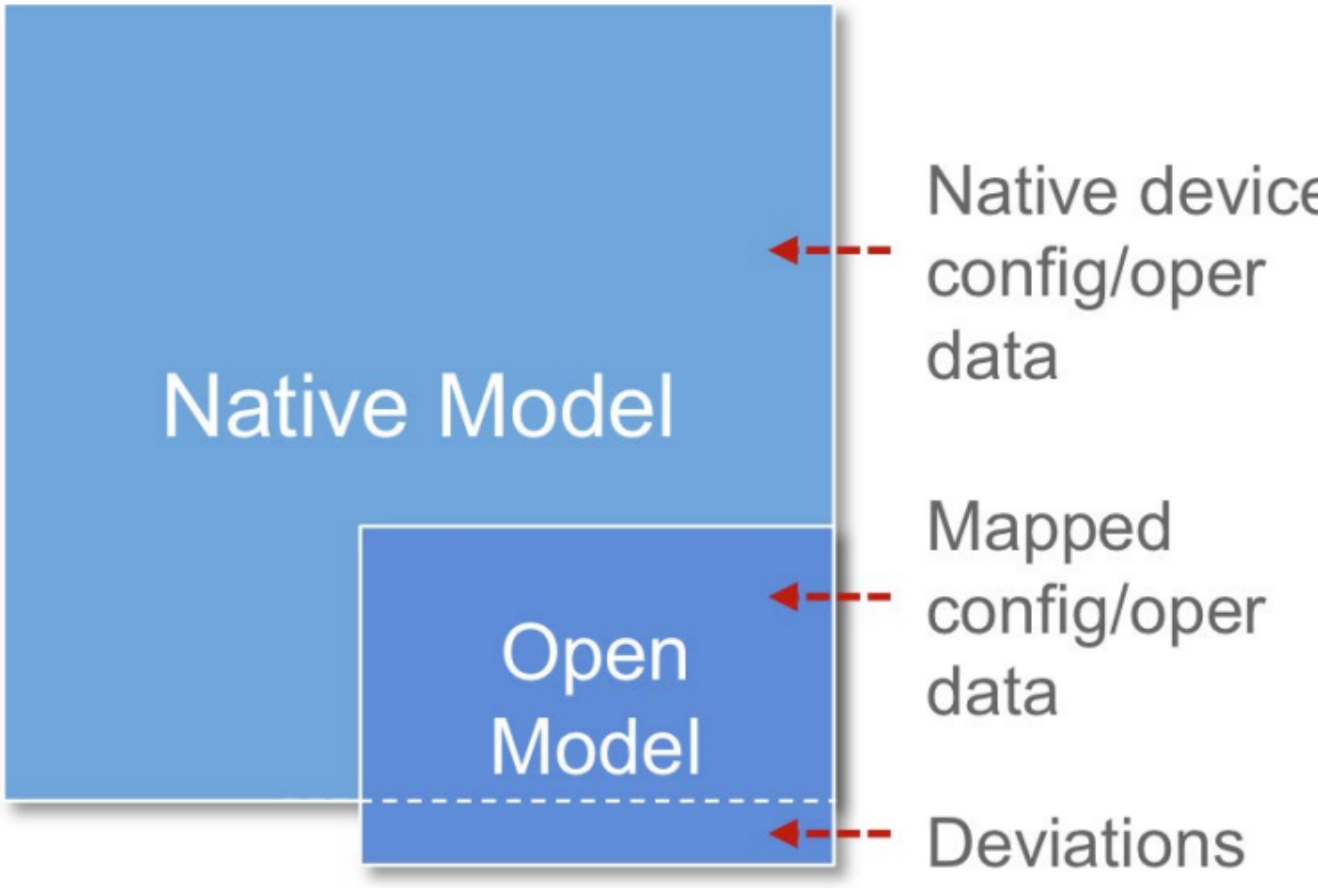
Но BGP с OSPF настроить точно можно.

Что делать в этом случае?

И есть два пути.
Один из них - брать ОС и видоизменять его с помощью добавления или убирания каких-либо его частей.
Когда вендор хочет расширить покрытие модели - он делает augmentation, встраивая его в нужное место.
Если он хочет поменять какое-то поведение или удалить функциональность - он описывает deviation к базовой модели.
Этот способ, конечно, не покрывает все потребности.

Другой - использовать вендорские Native модели, покрытие которых намного больше.

Абсолютно нормально совмещать OC и Native - главное, не настраивать одно и то же с помощью разных моделей. В целом рекомендуют (даже сами вендоры), использовать OC там, где это возможно, а где нет - прибегать к native.



Источник: доклад на Cisco Live

Google привёл в наш мир OpenConfig в одной руке, а gNMI - в другой.
Но в качестве транспорта для OC может быть как gNMI, так и NETCONF и RESTCONF - это не принципиально. В то же время, для gNMI OpenConfig в частности и YANG вообще не единственные возможные модели и языки.

Так что же это за мифический YANG?

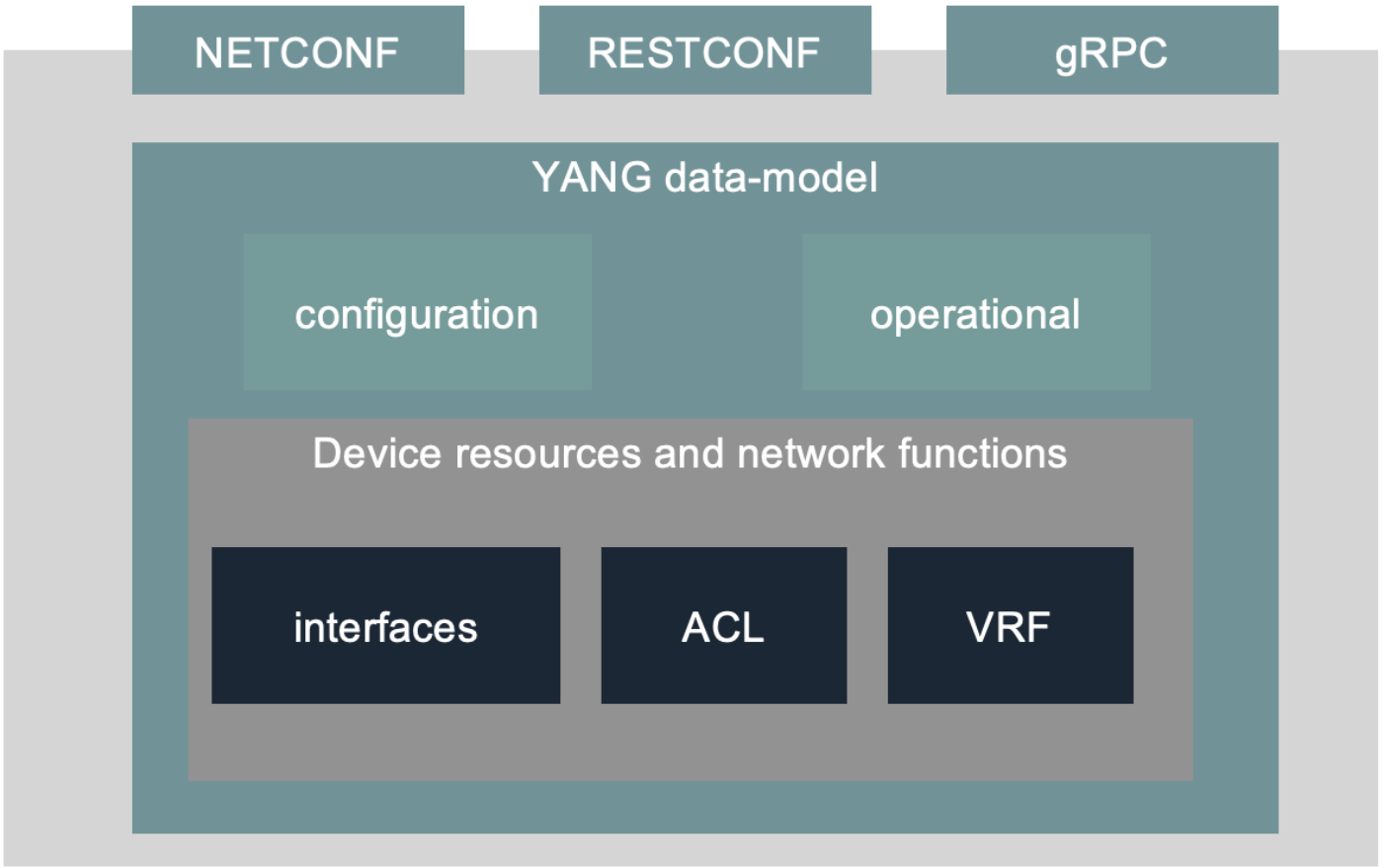
7.10 YANG

Оооо, как долго я шёл к этому, как долго я ждал, чтобы разобраться с этой темой.

Такой манящий и такой недоступный - **YANG** - *Yet Another Next Generation*, который решит все мои дилеммы автоматизатора, который снимет с меня груз парсинга CLI и приведёт нас всех в новый дивный мир.

Почему-то казалось, что стоит только понять, что такое YANG - и дальше самой сложной задачей останется оттраблшутить Ансибль и высадиться на Марсе.

YANG, а точнее модели, написанные на нём, не стали серебряной пулей, как не стали ей (пока) NETCONF, OpenConfig, gNMI.



И вообще YANG - вещь весьма академическая. Это просто язык описания моделей. Модели у каждого производителя и под каждую задачу могут быть совершенно разными, но, учитывая, что они все написаны на одном языке, мы можем применять одни и те же подходы и инструменты для работы с ними, и не отращивать ещё новые нейронные связи.

Вообще-то модели может не быть вовсе, или она может быть описана по-английски или по-русски, вместо YANG. Но при этом в JunOS/VRP/IOS по-прежнему будет какая-то структура данных. Просто в этом случае у вас не будет контракта, и в суд вы обратиться не сможете. Это собственно то, как мы и жили прежде.

Вообще-то YANG пришёл на помощь NETCONF'у, когда стало понятно, что <s>достаточно разврата - откапывайте SNMP SMI</s> без моделей дальше никуда ни шагу - и уже достаточно ошибок было совершено.

YANG - достойный сын SMIng. Когда парни из Network Working Group поняли, что с SNMP у них как-то не выгорело, весь накопленный багаж знаний они пустили на пользу обществу.

В общем-то, не мудрствуя лукаво, ребята из IETF взяли синтаксическую структуру и базовые типы из SMIng и запилили YANG.

При разработке YANG решили не совершать ошибок SMIng, который должен был стать универсальным языком под общие задачи, отчего немало страдал - нет, YANG нацеливался исключительно на NETCONF.

И какова ирония: RESTCONF и gNMI тоже решили использовать YANG - как язык моделирования. Ну логично ведь - не выдумывать 13-й стандарт же (хотя, подождите)?

Но гугл пошёл ещё дальше - gNMI может работать как с YANG-моделями, так и нет. Свободу вариативности! Что, впрочем, вполне логично, ведь в основе gNMI - protobuf'ы gRPC. А они могут как быть созданы на основе YANG-модели, так и просто придуманы из головы, или модель может быть написана не на YANG.

Как видно, благими намерениями уже тогда - был устлан путь к хьюман-ридабл, мэшин-парсибл.

Давайте мы не будем закапываться в сам язык - пожалуй, это нужно не очень большому числу людей. Нам интереснее практическая сторона - конкретные модели, как в них найти глазами нужные вещи, как с ними работать программно, какая вообще от них польза.

Дальше в качестве практики возьмём модель OpenConfig.

7.10.1 Препарируем YANG-модель

Давайте ещё раз вспомним пример, как мы собирали данные по конфигурации интерфейсов.

```
gnmic get --path "/interfaces/interface/subinterfaces/subinterface/ipv4/addresses/
↪address/config" \
      -a bcn-spine-1.arista:6030 \
      -u eucariot \
      -p password \
      --insecure
```

Откуда взялся этот замысловатый путь?

Для этого нам понадобится взглянуть на открытый репозиторий OpenConfig.

Пристально смотрим...
Ещё немного...
Понятно?

И мне не очень. Чтобы такое читать, надо всё же разбираться с самим языком.
Нам лень.
Поэтому воспользуемся помощью библиотеки `pyang`.

7.10.2 Pyang

Продолжим с примером про интерфейсы.

```
sudo pip install pyang
```

В рабочий каталог склоним репу:

```
git clone https://github.com/openconfig/public
```

И дадим вот такую команду:

```
pyang -f tree -p yang/oc/public/release/models/ \
      yang/oc/public/release/models/interfaces/openconfig-interfaces.yang
```

И дальше вываливается много текста:

```
module: openconfig-interfaces
  +--rw interfaces
     +--rw interface* [name]
        +--rw name             -> ../config/name
        +--rw config
        |  +--rw name?            string
        |  +--rw type             identityref
        |  +--rw mtu?             uint16
        |  +--rw loopback-mode?   boolean
        |  +--rw description?     string
        |  +--rw enabled?         boolean
        +--ro state
        |  +--ro name?            string
        |  +--ro type             identityref
        |  +--ro mtu?             uint16
        |  +--ro loopback-mode?   boolean
        |  +--ro description?     string
        |  +--ro enabled?         boolean
        |  +--ro ifindex?         uint32
        |  +--ro admin-status     enumeration
        |  +--ro oper-status      enumeration
        |  +--ro last-change?     oc-types:timeticks64
        |  +--ro logical?         boolean
        |  +--ro management?      boolean
        |  +--ro cpu?             boolean
        |  +--ro counters
        |     +--ro in-octets?             oc-yang:counter64
        |     +--ro in-pkts?               oc-yang:counter64
        |     +--ro in-unicast-pkts?       oc-yang:counter64
        |     +--ro in-broadcast-pkts?     oc-yang:counter64
        |     +--ro in-multicast-pkts?     oc-yang:counter64
        |     +--ro in-discards?           oc-yang:counter64
        |     +--ro in-errors?             oc-yang:counter64
        |     +--ro in-unknown-protos?     oc-yang:counter64
        |     +--ro in-fcs-errors?         oc-yang:counter64
        |     +--ro out-octets?            oc-yang:counter64
        |     +--ro out-pkts?              oc-yang:counter64
        |     +--ro out-unicast-pkts?      oc-yang:counter64
        |     +--ro out-broadcast-pkts?    oc-yang:counter64
        |     +--ro out-multicast-pkts?    oc-yang:counter64
        |     +--ro out-discards?          oc-yang:counter64
```

(continues on next page)

(продолжение с предыдущей страницы)

```
|     +--ro out-errors?            oc-yang:counter64
|     +--ro carrier-transitions?   oc-yang:counter64
|     +--ro last-clear?            oc-types:timeticks64
+--rw hold-time
|  +--rw config
|  |  +--rw up?     uint32
|  |  +--rw down?   uint32
|  +--ro state
|     +--ro up?     uint32
|     +--ro down?   uint32
+--rw subinterfaces
   +--rw subinterface* [index]
      +--rw index     -> ../config/index
      +--rw config
      |  +--rw index?         uint32
      |  +--rw description?   string
      |  +--rw enabled?       boolean
      +--ro state
         +--ro index?          uint32
         +--ro description?    string
         +--ro enabled?        boolean
         +--ro name?           string
         +--ro ifindex?        uint32
         +--ro admin-status    enumeration
         +--ro oper-status     enumeration
         +--ro last-change?    oc-types:timeticks64
         +--ro logical?        boolean
         +--ro management?     boolean
         +--ro cpu?            boolean
         +--ro counters
            +--ro in-octets?             oc-yang:counter64
            +--ro in-pkts?               oc-yang:counter64
            +--ro in-unicast-pkts?       oc-yang:counter64
            +--ro in-broadcast-pkts?     oc-yang:counter64
            +--ro in-multicast-pkts?     oc-yang:counter64
            +--ro in-discards?           oc-yang:counter64
            +--ro in-errors?             oc-yang:counter64
            +--ro in-unknown-protos?     oc-yang:counter64
            +--ro in-fcs-errors?         oc-yang:counter64
            +--ro out-octets?            oc-yang:counter64
            +--ro out-pkts?              oc-yang:counter64
            +--ro out-unicast-pkts?      oc-yang:counter64
            +--ro out-broadcast-pkts?    oc-yang:counter64
            +--ro out-multicast-pkts?    oc-yang:counter64
            +--ro out-discards?          oc-yang:counter64
            +--ro out-errors?            oc-yang:counter64
            +--ro carrier-transitions?   oc-yang:counter64
            +--ro last-clear?            oc-types:timeticks64
```

Неплохо. С такими аргументами pyang представляет модель в виде дерева, выбрасывая несущественные данные.

Здесь сразу видно, на каком уровне иерархии что находится, какой у него тип и режим - rw или ro.

Постойте, но где же ipv4, который в запросе gnmic? Тут его явно нет. А в запросе и ответе явно был - то есть где-то он должен существовать и в модели.

Взглянем ещё раз на директорию. И повторим pyang:

```
pyang -f tree  -p yang/oc/public/release/models/ \
      yang/oc/public/release/models/interfaces/openconfig-if-ip.yang | head -n 10

module: openconfig-if-ip

  augment /oc-if:interfaces/oc-if:interface/oc-if:subinterfaces/oc-if:subinterface:
    +--rw ipv4
       +--rw addresses
       |  +--rw address* [ip]
       |     +--rw ip        -> ../config/ip
       |     +--rw config
       |     |  +--rw ip?              oc-inet:ipv4-address
       |     |  +--rw prefix-length?   uint8
```

Вот и он во всей красе. И тут видно, что это аугментация к модели `/oc-if:interfaces/oc-if:interface/oc-if:subinterfaces/oc-if:subinterface`.

А что такое *oc-if*?

```
less yang/oc/public/release/models/interfaces/openconfig-interfaces.yang | grep '^␣
↪*prefix'
prefix "oc-if";
```

Итак, у модели есть короткий префикс для более удобного обращения к ней. Он используется в другой модели, чтобы сделать его аугментацию.

Так можно просто проверить корректность

```
pyang -p yang/oc/public/release/models/ \
      yang/oc/public/release/models/interfaces/openconfig-interfaces.yang
```

Ключ `-f` позволяет конвертировать в разные форматы: `tree`, `yin`, `yang`, `jstree`, `uml` и другие.

Для нас интереснее всего `tree` и `uml`, потому что вот такие крутые картинки можно рисовать для визуалов

> Чтобы конвертировать uml в png можно воспользоваться пакетом plantuml. Ссылка на картинку побольше

С помощью pyang, конечно, можно работать не только с моделями OpenConfig, но и с любыми другими, написанными на языке YANG.

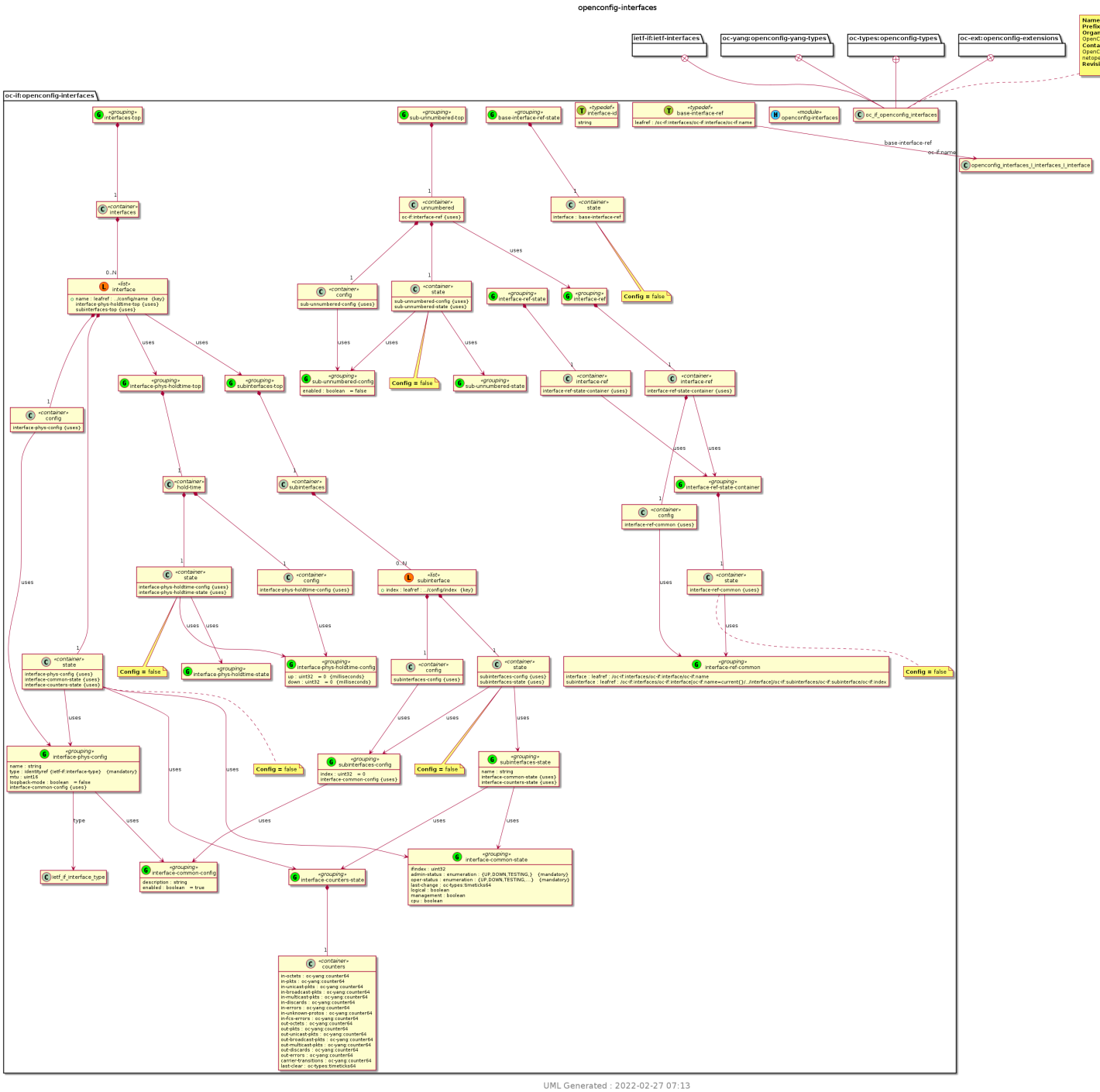
Место, где неплохо описан pyang.

А тут бравые парни из Австралии пишут свою модель. С этой работой будет полезно ознакомиться тем, кто хочет разобраться поглубже в языке YANG.

На сегодняшний день многие вендоры поддерживают YANG-схему для своих NETCONF/gNMI-интерфейсов.

Есть несколько мест, где их можно раздобыть:

- Официальный репозиторий YANG. Тут есть ссылки на гитхабы вендоров, которые публикуют свои модели.
- Но Huawei, например, хранит свои YANG-и в нескольких местах (https://github.com/Huawei/yang/ и https://github.com/HuaweiDatacomm/yang)


openconfig-interfaces
ietf-if:ietf-interfaces
oc-yang:openconfig-yang-types
oc-types:openconfig-types
oc-ext:openconfig-extensions
oc-if:openconfig-interfaces
«grouping» interfaces-top
«container» interfaces
«list» interface
«grouping» interface-phys-holdtime-top
«grouping» subinterfaces-top
«container» hold-time
«container» subinterfaces
«container» config
«container» state
«grouping» interface-phys-config
«grouping» interface-common-config
«grouping» interface-counters-state
«container» counters
«grouping» sub-unnumbered-top
«container» unnumbered
«container» config
«container» state
«grouping» sub-unnumbered-config
«grouping» sub-unnumbered-state
«grouping» base-interface-ref-state
«typedef» interface-id
«typedef» base-interface-ref
«module» openconfig-interfaces
oc_if_openconfig_interfaces
openconfig_interfaces_|_interfaces_|_interface
«grouping» interface-ref-state
«grouping» interface-ref
«container» interface-ref
«container» interface-ref-state-container
«grouping» interface-ref-state-container
«container» config
«container» state
«grouping» interface-ref-common
«container» subinterfaces
«list» subinterface
«container» config
«container» state
«grouping» subinterfaces-config
«grouping» subinterfaces-state
«container» config
«container» state
«grouping» interface-phys-holdtime-config
«grouping» interface-phys-holdtime-state
Config = false
base-interface-ref
oc-if:name
uses
type
1
0..N
UML Generated : 2022-02-27 07:13

- Отдельно так же лежат схемы аристы: https://github.com/aristanetworks/yang
- Некоторые вендоры могут хранить их на своих сайтах.

В общем, собираем с репы по коммиту.

Замечательная новость в том, что все коробки, заявляющие своё соответствие RFC6022 должны уметь возвращать всю YANG-схему по запросу с операцией `<get_schema>`.
Отвратительная новость в том, что не все вендоры эту операцию поддерживают.

Что нужно знать про YANG?

Это способ описать структуру данных, но не сами данные.
Сами данные могут быть представлены в любом формате, поддерживающем структуры: XML, JSON, Protobuf, объекты Python.
YANG придумывали не для того, чтобы решить общую задачу, он нацелен на конкретно NETCONF и конкретно XML-кодирование. Но его смогли присобачить и к другим интерфейсам.

Я бы взял на себя смелость сказать, что NETCONF/YANG - это как TCP/IP. То есть там и про NETCONF, и про YANG. Однако не только NETCONF.

Очевидно, YANG - огромная тема, которой будет тесно даже на страницах отдельной книги. В этой статье мне хотелось только приоткрыть первый её разворот, на котором ещё нет пугающих выкладок.
Возможно, когда-то, если я осознаю неизбежность его повсеместного проявления, я напишу отдельную огромную статью и о нём. А пока положим достаточным эти едва ли заметные тропинки.

7.11 Model Driven Programmability

Так что же это такое? Ведомая моделью программируемость? Теперь, после двух статей, нам хватит пары минут, чтобы разобраться что это такое.

Давайте вернёмся к 4-й части АДСМ, где я использовал позаимствованную у Дмитрия Тесля картинку.

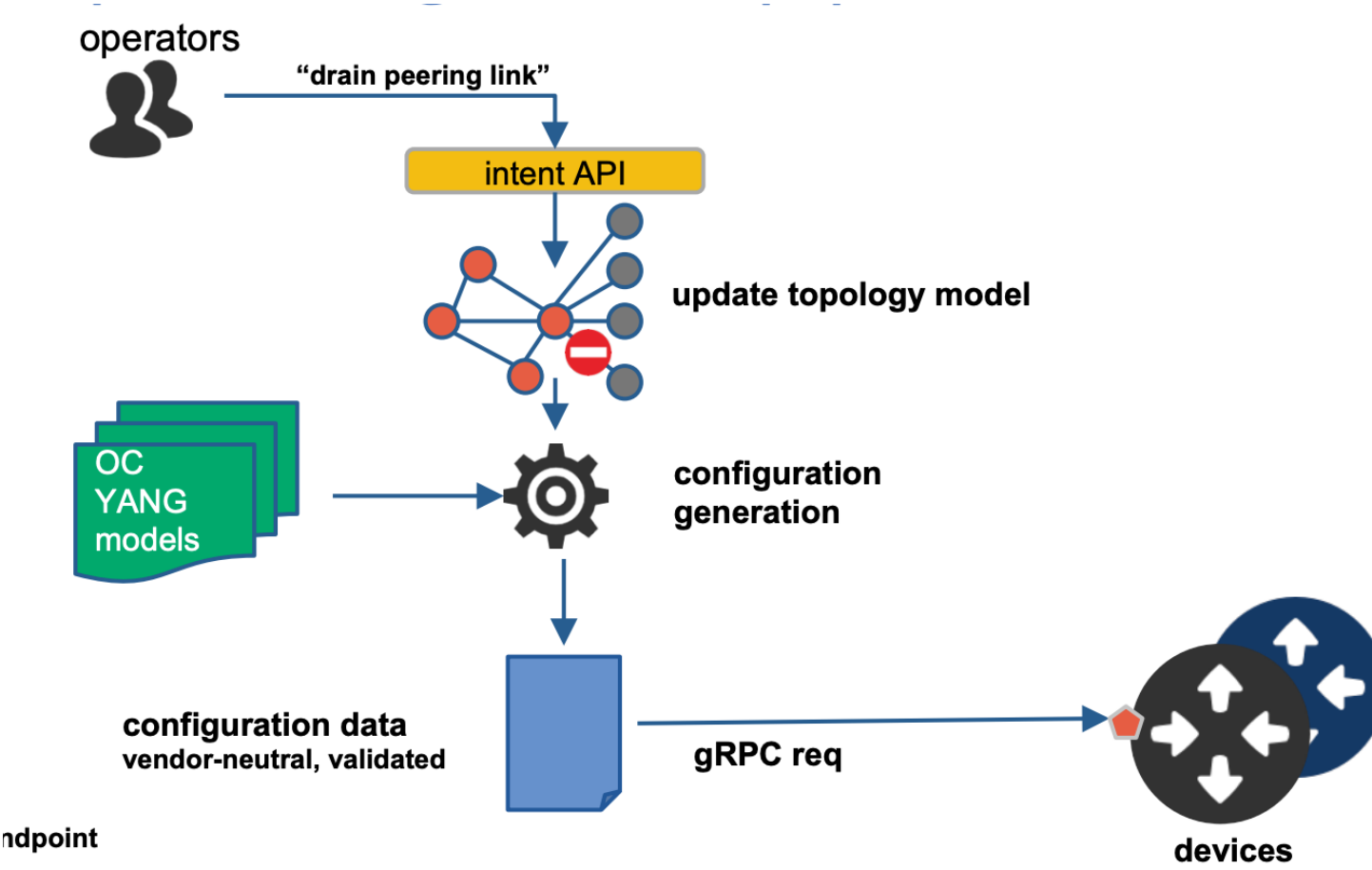
Источник: dteslya.engineer/network_automaiton_101

Она ведь очень понятная? Inventory, Git с шаблонами конфигурации, рендер, валидация, применение.

Где закопаны два мешка с человеко-неделями? Под шаблонами с генераторами и под системами применения конфигурации.
Со вторым пытаются бороться NETCONF, RESTCONF, gNMI.
А с первым - модели.

Проблема в том, что шаблоны мы составляем руками на основе изучения документации, интерфейса коробки и действуем методом проб и ошибок, вообще-то. Если нужна проверка типов, диапазонов, если меняется иерархия - будьте добры сами всё это написать и обработать. И, окончив, уехать в сумасшедший дом, учить друзей джинджа-программированию.

Model Driven меняет картину следующим образом:



Не могу найти, откуда брал эту картинку.

Здесь шаблоны конфигурации заменяются на YANG-модель (в данном случае OpenConfig).
Из инвентарки (топологии) и этих моделей рендерится конфиг, который с помощью RPC (тут gRPC) прогружается на коробку.

Model Driven означает тут, что мы

А) не думаем (или думаем меньше) про иерархию, типы данных. Перестаём мыслить тегами XML.
Б) Модель определяет, как будет выглядеть конфигурация, как с ней работать.
В) Использование точно такой же модели на устройстве гарантирует, что отправленное нами, будет принято и валидно на той стороне, коль скоро оно валидно на этой.

Иными словами именно модель управляет тем, как мы и железо будет работать с конфигурацией.
Вот и вся суть.

7.12 Всё вместе

Ух, какую же большую кучу из разных технологий и идей мы свалили с начала статьи. Пора бы её разобрать и по коробкам разложить.
Итак,

Транспорт

- SSH,
- HTTP,
- HTTP/2
- SNMP тоже, конечно же, возможен, но не нужен.

Интерфейс

- CLI
- SNMP
- NETCONF
- RESTCONF
- gRPC

Формат данных

- Текст
- XML
- JSON
- Protocol Buffers

Способ описания спецификации - он же может называться схемой

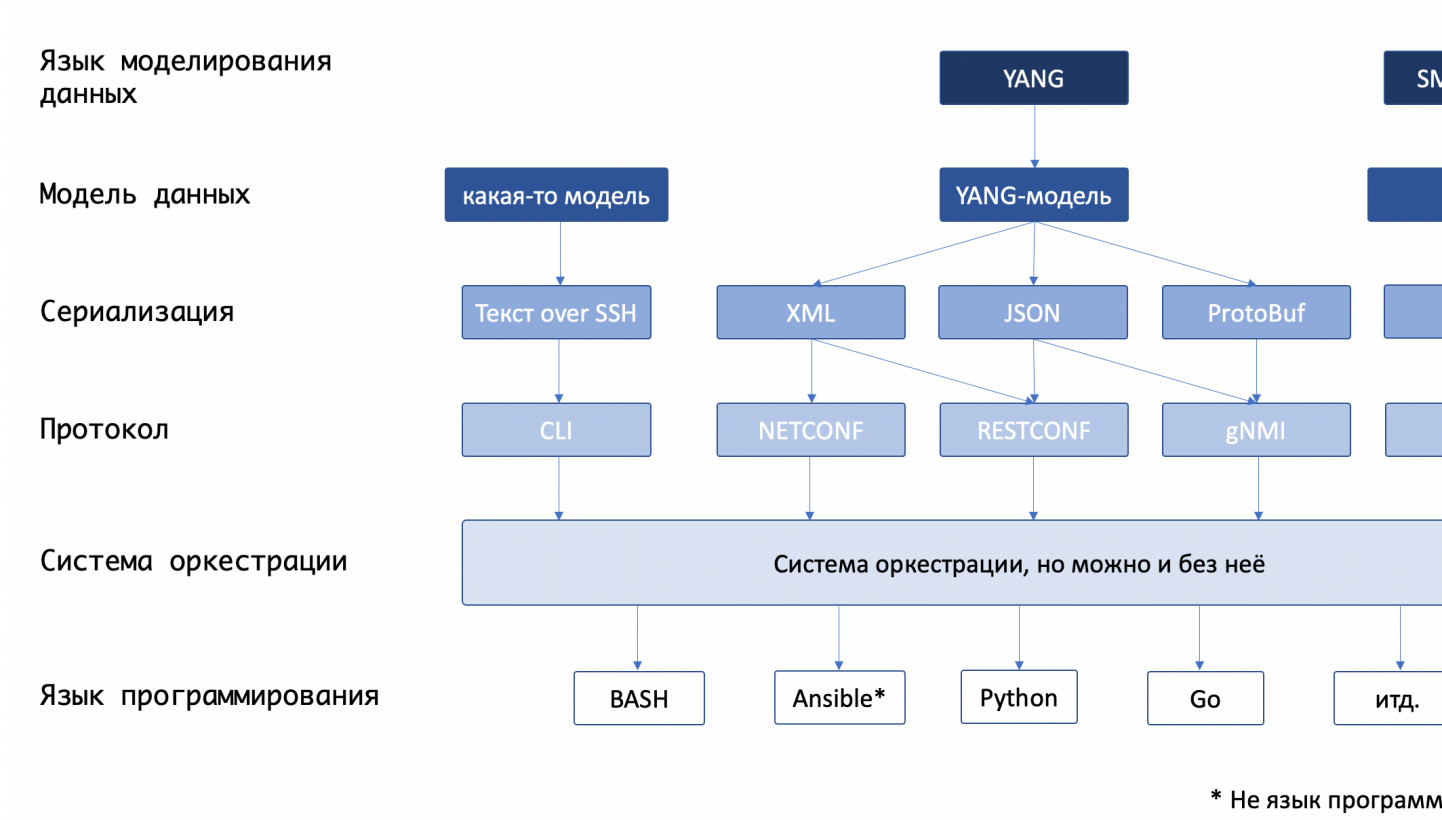
- XSD
- JSON schema
- Protocol Buffers
- MIB
- Проприетарный способ, придуманный вендором и описанный в документации.

YANG-модели данных конфигурации

- OpenConfig
- Проприетарные модели
- IETF
- Проприетарная модель, придуманная вендором и неописанная в документации

Языки описания моделей

- YANG
- SMI/SMIng
- Проприетарный язык, придуманный вендором и не описанный в документации



7.12.1 И ещё другими словами

- **YANG** - язык моделирования данных, но не сами модели,
- **YANG-модели** - конкретные модели, написанные на языке YANG, но ещё не сами данные и не их схема,
- **OpenConfig** - вендор-независимая YANG-модель данных конфигурации сетевого оборудования,
- **Native-модели** - вендорские проприетарные YANG-модели данных сетевой конфигурации,
- **XML, JSON, Protobuf** - синтаксис по представлению структур данных в виде, пригодном для передачи (например, строка), иными словами - сериализация,
- **XML-схемы (XSD), JSON-схемы, proto-спецификации** - репрезентация YANG-модели в соответствующем формате, схема
- **NETCONF** - протокол взаимодействия с сетевым железом, работающий поверх SSH. В качестве формата данных использует XML. Структура XML может быть основана на YANG-модели, но не обязательно,
- **RESTCONF** - аналог NETCONF, но работающий через HTTP. Данные представляются в JSON или XML на основе какой-либо YANG-модели,
- **gRPC** - фреймворк для межсервисного взаимодействия, которые реализует интерфейс, протокол, формат данных и спецификации (protocol buffers). Непосредственно к сетям отношения не имеет,

- **Protobuf** - он же protocol buffers - спецификация для gRPC, а так же формат передачи данных в нём,
- **gNMI** - протокол на основе gRPC для взаимодействия с сетевым оборудованием. Всегда основан на модели, представленной в формате protobuf-спецификации, но это не обязательно должна быть YANG-модель.

И чтобы окончательно разобраться в терминах, давайте разложим по полочкам: схема, спецификация, IDL.
Схема - это широкий термин. Это то, что описывает, как данные должны быть представлены и чему соответствовать: структура, иерархия, типы итд.

Думаю, что слова «схема» и «спецификация» мы можем считать синонимами.

Для каждого формата данных будет так же и свой формат написания схем.

Для XML - это XSD, для JSON - JSON-schema, для gRPC - protobuf.

А уже конкретный файл/текст, описывающий какие-либо данные - это и будет сама схема.

Соответственно данные можно провалидировать по схеме - убедиться, соответствуют ли они ей.

Из схемы/спецификации можно создать объекты языка программировани, чтобы было удобнее работать с ними.

То есть из XML-схемы создаём классы, например, питона, работаем с ними привычным образом, далее преобразуем в XML, который можно уже проверить на соответствие изначальной схеме. Или данные, полученные из какой-то внешней системы, можно проверить на такое соответствие, прежде чем начинать обрабатывать.

IDL - его назначение прямо следует из названия - язык определения интерфейса. Если схема описывает как данные выглядят вообще, то IDL определяет, как две системы должнц представлять данные, чтобы взаимодействовать друг с другом. То есть это уже контракт между ними, а схема - это инструмент, позволяющий этого добиться.

Таким образов в gRPC protobuf является и IDL и способом описания спецификацией. В случае NETCONF формат данных - это XML, способ описания спецификации - это XSD, а в качестве IDL выступает сам NETCONF - ведь именно он и определяет интерфейс.

Модель же определяет то, как будет выглядеть сама спецификация/схема. То есть это ещё более абстрактная конструкция. И нужна модель для того, чтобы на её основе была возможность создать как proto-спеку, так и JSON-схему, так и XSD.

7.13 Полезные ссылки

- Как я собирал лабу для данной статьи
- Репозиторий с примерами кода из этой статьи
- Главные RFC:
  - NETCONF: RFC4741, RFC6241
  - RESTCONF: RFC8040
  - YANG: RFC6020, RFC7950
- Концепция RPC: Remote Procedure Call (RPC)

- Про форматы данных: YAML: The Missing Battery in Python
- Детальный разбор XML во всём его многообразии: XML Tutorial
- XML разобран по кусочкам и мило иллюстрирован (русский): Что такое XML
- Understanding XML Namespaces
- Сайт OpenConfig: openconfig.net
- Опубликованные модели OpenConfig
- Хорошая вводная в NETCONF и немного YANG: YANG Data Modelling and NETCONF
- Продолжение про YANG и немного про NETCONF: The Road to Model Driven Programmability
- На русском про NETCONF. Начало
- Если жить не можете, хочется на русском про YANG, и вы воспринимаете художественную речь: YANG — это имя для вождя
- Спецификация gNMI в его же репозитории
- Совершенно чудесный и обязательный к употреблению сайт от Аристы с описанием и примерами разных интерфейсов и инструментов: Open Management
- RESTCONF Tutorial - Everything you need to know about RESTCONF in 2020
- Серия статей про RESTCONF, но рекомендую я её из-за того, что там хорошо разобраны примеры с YANG-моделями: RESTCONF, NETCONF and YANG
- Достаточно обстоятельный разбор телеметрии поверх gNMI: Три года Телеметрии в IOS XR
- Документация по gRPC
- Python Microservices With gRPC
- Блог Романа Додина в общем. И в частности:
  - gNMIc
  - Пример взаимодействия с Arista через gNMI
- Блог Антона Карнелюка в общем. И в частности:
  - OpenConfig. Part 1. How open is OpenConfig
  - Серия Automation
  - Pygnmi блог-пост и репо.
    Но на сайте поехала вёрстка - до чего-то приходится докапываться самому.
- Блог Михаила Кашина в общем. И в частности:
  - Getting Started With NETCONF and YANG on Cisco IOS XE
- Репозиторий с PyangBind, позволяющим генерировать классы Python из YANG-моделей
- Для любителей изучать Питон по Луцу: NETCONF XML Management Protocol Developer Guide
- gRPC Network Operation Interface
- Сложный, но полезный джуниперовский документ: Understanding OpenConfig and gRPC on Junos Telemetry Interface. Я не читал ;)
- Про кишки YANG для новчиков

7.14 Заключение

Думаю, что хорошее заключение было в пятой части книги, к которой я и отсылаю читателя. | Путь нас ожидает неблизкий. Туман медленно рассеивается, открывая новые развилки дорожек, из которых нужно выбирать перспективную. | Но вот что следует держать в голове. Нам обо всём этом рассказывают на конференциях и пишут в длинных статьях как о свершившемся факте, в то время, как многие вещи всё ещё не работают, а в конце обычно есть приписка *«Adding support of OpenConfig gNMI paves the way for future network automation»*.

7.14.1 Благодарности

- Роману Додину за дельные комментарии как по теоретической, так и по практической частям. А так же за полезный блог и инструменты. GitHub.
- Кириллу Плетнёву за наведение порядка с NETCONF и YANG - язык, модели, спецификации, форматы данных. И за уместные и остроумные замечания по языкам и библиотекам. GitHub, fb.
- Александру Лимонову за несколько идеологических замечаний и исправлений фактических ошибок.
- Александру Балезину за написанную часть про Telemetry.