

Компьютерные сети

лекция 2



Андрей
Вахутинский



Андрей Вахутинский

Зам.начальника ИТ отдела в АО "ИНТЕКО"





План модуля

1. Работа в терминале, лекция 1
2. Работа в терминале, лекция 2
3. Операционные системы, лекция 1
4. Операционные системы, лекция 2
5. Файловые системы
6. Компьютерные сети, лекция 1
- 7. Компьютерные сети, лекция 2**
8. Компьютерные сети, лекция 3
9. Элементы безопасности информационных систем



План занятия

1. [Адресация IPv4](#)
2. [Оптимизации TCP](#)
3. [DNS, Domain Name System](#)
4. [HTTP, HyperText Transfer Protocol](#)



Адресация IPv4

Проблема разграничения сетей

В предыдущей лекции мы говорили о необходимости наличия двух типов адресов:

- **L2 MAC** для адресации внутри локальной сети (шестнадцатеричная запись),
- **L3 IP** для адресации между сетями (десятичная запись).

Мы увидели, что хосты в сети имеют оба типа этих адресов одновременно.

```
vagrant@netology1:~$ ip addr show eth0 | egrep '(ether|inet )'  
link/ether 08:00:27:d2:23:ca brd ff:ff:ff:ff:ff:ff  
inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic eth0
```

На уровне L2 все несложно – формат Ethernet фрейма предполагает прямую доступность в локальной сети и адресацию по SRC + DST MAC.

Но мы видели в tcpdump, что при общении приложений по спецификациям TCP/IP, ОС не оперирует адресами *канального* уровня: TCP соединения устанавливаются между *сетевыми* адресами. Каким же образом хост “понимает”, что адрес сетевого уровня относится или не относится к его собственной сети? И если адрес “чужой”, то по какому пути до него можно добраться?

Подобное разграничение L3 сетей достигается дополнительной битовой маской, назначаемой хосту вместе с IP адресом, – **маской подсети**.

Двоичное представление

Чтобы понять, как работает маска подсети, нужно **научиться переводить десятичные числа для IPv4 (шестнадцатеричные для IPv6) в двоичные и использовать битовую операцию AND.**

Компьютеры работают с битами. Бит может быть “включен” (равен 1), или “выключен” (равен 0); то есть, бит представляет собой выбор из двух вариантов.

Вы наверняка слышали словосочетание “8-битная”, “32-битная” архитектура. Это такая архитектура, которая способна работать с набором бит длины 8 (32). Например:

- **8 бит:** 0010 1001
- **32 бит:** 0110 1101 1000 1000 0000 1111 1110 1001

Данная последовательность может представлять что угодно:
цвет индивидуальной точки в изображении, букву в таблице символов, число.

Так как выбор представлен только двумя значениями (0 или 1), можно посчитать что при 8 битах существует $2^8 = 256$ вариантов наборов установленных бит ($2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$);

$2^{32} = 4,294,967,296$ вариантов наборов установленных бит.

Таблица степеней двойки

У нас есть 8 бит данных (1 байт или октет). Что можно выразить 256 вариантами? Часто встречающееся **десятичное представление – положительные целые числа с отсчетом от 0, то есть 0, 1, 2 ... 255, или $2^n - 1$.**

Набор значений – предмет договоренности о том, как интерпретировать установленные биты. Это может быть и диапазон от -128 до 127, и дробные minifloat, и т.д.

Как представить десятичные числа от 0 до 255 в двоичной записи? Степени двойки:

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

$$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$$

Если все биты установлены в 0 – 0000 0000, получаем 0;

если младший регистр установлен в 1 – 0000 0001, получаем 1;

если 4 и младший 8 регистры установлены в 1 – 0001 0001, получаем 17 и т.д.

Конвертация: по таблице

Используя приведенную таблицу степеней двойки, очень легко конвертировать любое десятичное число из диапазона 0-255 в двоичное.

Алгоритм:

1. Какое будет двоичное представление числа 67?
2. Начиная со старшего разряда – составляет ли 128 часть числа 67? Нет. Старший 8 разряд равен 0 – 0xxx xxxx.
3. Составляет ли 64 часть числа 67? Да, устанавливаем бит в единицу: 01xx xxxx.
4. Остаются десятичные 3 единицы. Все разряды вплоть до второго нас не интересуют, так как 32, 16, 8 и 4 не составляют часть числа 3 – 0100 00xx.
5. Напротив, 2 составляет часть числа 3 – 0100 001x.
6. Остается единица, она в младшем разряде. Таким образом, десятичное число 67 может быть представлено в двоичном виде как 0100 0011.

Конвертация: python

Из двоичного представления в десятичное:

```
Python 3.7.7 (default, Mar 10 2020, 15:43:33)
[Clang 11.0.0 (clang-1100.0.33.17)] on darwin
>>> 0b00000000
0
>>> 0b00000001
1
>>> 0b000010001
17
```

Из десятичного в двоичное:

```
>>> bin(0)
'0b0'
>>> bin(255)
'0b11111111'
>>> bin(67)
'0b1000011'
```

Python опускает старшие разряды по умолчанию, если они нулевые.

```
>>> format(67, '08b')
'01000011'
```

IPv4 адрес – 32 бит или 4 октета

Адресное пространство IPv4 является 32-битным. Компьютер работает с двоичными данными, но для удобства их восприятия разработчиками могло быть выбрано любое представление. Для IPv4 выбрано десятичное представление **4 октетами по 8 бит - 0-255 в каждом октете.**

Десятичное представление адресного пространства:
от 0.0.0.0 до 255.255.255.255 (0-255.0-255.0-255.0-255)

Двоичное представление адресного пространства:
от 00000000.00000000.00000000.00000000 до 11111111.11111111.11111111.11111111

Говоря о представлении, можно привести в пример tcpdump*:

```
16:45:22.723690 IP 127.0.0.1.32862 > 127.0.0.1.8000...
```

*В примере номер порта записан через точку после IP адреса, т.е. в разных утилитах в реальной ОС можно встретить разные варианты одних и тех же величин (обычно номер порта указывается через двоеточие).

IPv4 маска, форматы записи

По формату адреса IP видно, что в нем нет признаков, которые помогли бы с определением границ сети. Для этого существует **маска**.

В случае с IPv4 это такое же 32-битное число, как и сам адрес:

```
vagrant@vagrant:~$ ip -4 a s eth0
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group
default qlen 1000
    inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic eth0
        valid_lft 49201sec preferred_lft 49201sec
```

/24 - говорит о том, что из 32 бит адреса 24 бита являются адресом сети, остальные 8 бит – хостовой частью. Существует 2 нотации:

- устаревшая классовая;
- современная бесклассовая CIDR (Classless Inter-Domain Routing).

Маска /24 CIDR формата в ином варианте записи:

1111 1111.1111 1111.1111 1111.0000 0000 (24 + 8) или 255.255.255.0.

Число адресов в подсети легко прикинуть по CIDR: $2^8 = 256$.

```
vagrant@vagrant:~$ netmask -s 10.0.2.15/24
10.0.2.0/255.255.255.0
```

Логическое И

Мы узнали о характеристиках IPv4 адреса, о маске подсети, десятичном и двоичном представлении этих данных.

Далее - нужно **собрать всё воедино битовым логическим оператором AND (И), который возвращает:**

- **0** - если хотя бы один из битов равен 0;
- **1** - если оба бита равны 1.

Пример:

```
    00110 &
    01011
=    00010
>>> res = 0b00110 & 0b01011;
>>> format(res, '05b')
'00010'
```

Результат применения логического И – адрес сети.

IPv4 маска, применение на простом примере

Маска десятичная	255	255	255	0
Адрес десятичный	192	168	0	10
Адрес двоичный	11000000	10101000	00000000	00001010
Маска двоичная	11111111	11111111	11111111	00000000
Сетевая часть	nnnnnnnn	nnnnnnnn	nnnnnnnn	
Хостовая часть				hhhhhhhh
Адрес сети двоичный	11000000	10101000	00000000	00000000
Адрес сети десятичный	192	168	0	0

192.168.0.10/24 – это хост с адресом 192.168.0.10. Этот хост:

- имеет маску подсети /24 или 255.255.255.0,
- находится в подсети 192.168.0.0.

По стандарту маски должны обеспечивать смежность подсетей, поэтому они всегда будут выглядеть как последовательность из 1 в начале и 0 в конце. Адрес всегда принадлежит какой-то подсети.

IPv4 маска, применение на примере сложнее

Маска десятичная	255	255	224	0
Адрес десятичный	64	233	161	138
Адрес двоичный	01000000	11101001	10100001	10001010
Маска двоичная	11111111	11111111	11100000	00000000
Сетевая часть	nnnnnnnn	nnnnnnnn	nnn	
Хостовая часть			hhhhh	hhhhhhhh
Адрес сети двоичный	01000000	11101001	10100000	00000000
Адрес сети десятичный	64	233	160	0

Еще один пример на базе публичного адреса google:

```
vagrant@vagrant:~$ google_v4_addr=$(dig +short A google.com | head -n1)
vagrant@vagrant:~$ echo $google_v4_addr; whois $google_v4_addr | grep CIDR
64.233.161.138
CIDR:          64.233.160.0/19
```

Под хостовую часть отделен не просто целый последний октет (/24), а /19, таким образом в подсеть попадают адреса от 64.233.160.0 до 64.233.191.255.

IPv4, особенности нумерации

В первом примере 192.168.0.0 – *специальный зарезервированный адрес*, он называется **адресом подсети**. Последний адрес подсети (192.168.0.255) также является зарезервированным, – он предназначен для **широковещательных** сообщений на уровне L3 (по аналогии с адресом ff:ff:ff:ff:ff:ff L2 MAC).

Часто адресом шлюза (через который можно попасть в другие сети), выступает первый доступный адрес сети, в данном случае - 192.168.0.1.

Нетрудно подсчитать, что при маске 255.255.255.0 есть всего 256 адресов в последнем октете (хостовой части), из которых 2 недоступно (адрес подсети и широковещательный адрес в резерве), 1 уже занят нашим хостом 192.168.0.10, а, значит, в данной подсети может находиться еще 253 соседних адреса (или даже 252 если считать шлюз).

Автоматизированный калькулятор **ipcalc**:

```
vagrant@vagrant:~$ ipcalc 192.168.0.10/24
Address:    192.168.0.10      11000000.10101000.00000000. 00001010
Netmask:    255.255.255.0 = 24 11111111.11111111.11111111. 00000000
Network:    192.168.0.0/24    11000000.10101000.00000000. 00000000
HostMin:    192.168.0.1      11000000.10101000.00000000. 00000001
HostMax:    192.168.0.254    11000000.10101000.00000000. 11111110
Broadcast:  192.168.0.255    11000000.10101000.00000000. 11111111
```


IPv4, особенные диапазоны

В диапазоне IPv4 есть множество адресов, имеющих специальное назначение:

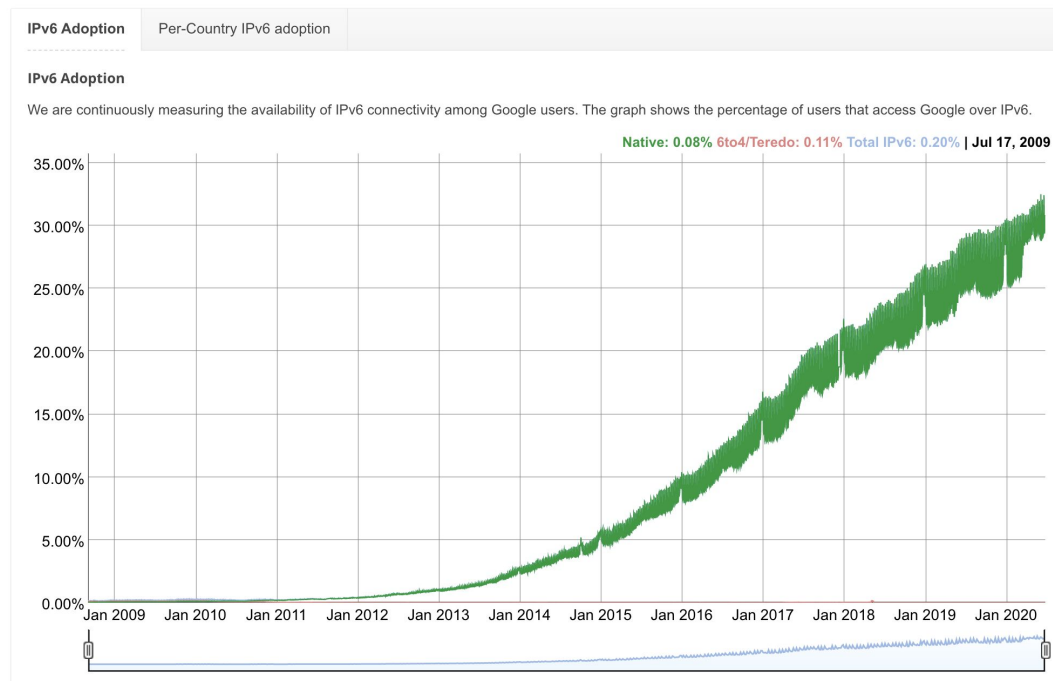
- всем известный 127.0.0.1, который на самом деле лишь 1 стандартный адрес loopback интерфейса из огромной (16+ млн) подсети 127.0.0.0/8,
- немаршрутизируемые частные сети 10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16,
- немаршрутизируемые link-local адреса 169.254.0.0/16, которые могут быть, например, назначены при неуспешной работе DHCP клиента,
- диапазон 224.0.0.0/4 для multicast (широковещательная групповая передача),
- 100.64.0.0/10 для carrier-grade NAT,
- и многие другие, ознакомьтесь с полным списком [на Wiki](#).

IPv6 уже здесь

Для системного администратора/DevOps внедрение IPv6 с современными ядрами не является проблемой.

Большинство приложений нормально работают с IPv6, решены проблемы dual-stack систем (с IPv4 и IPv6 одновременно).

Статистика [google](#):

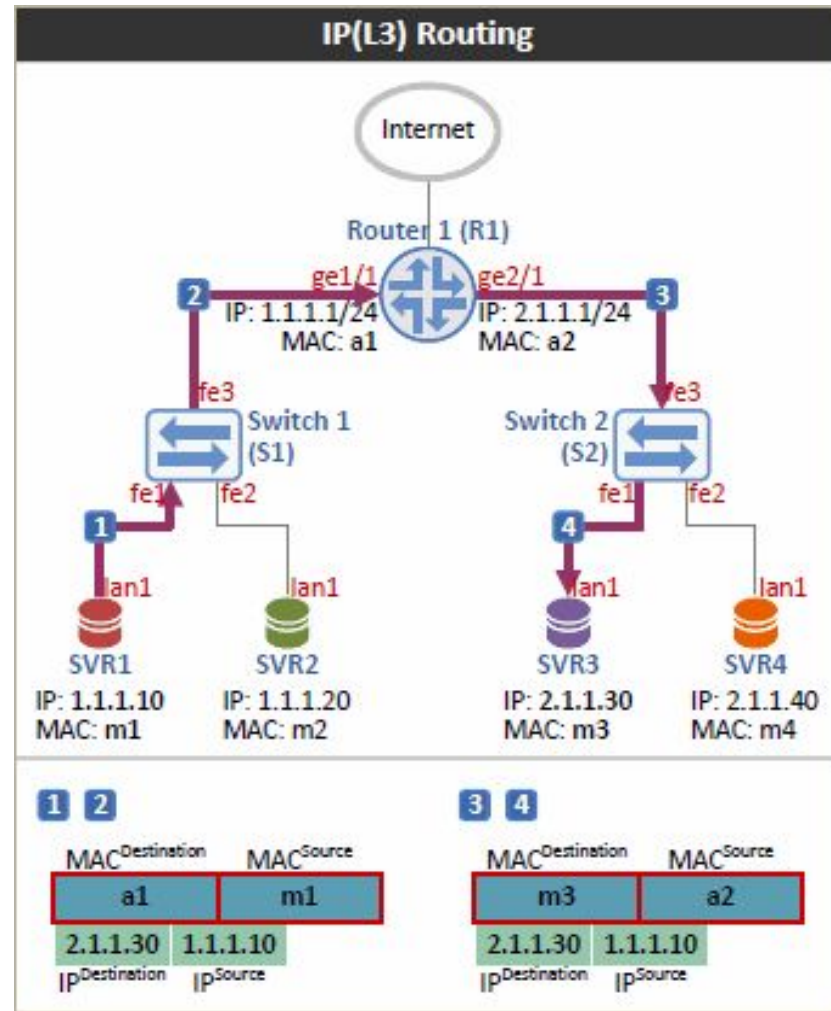
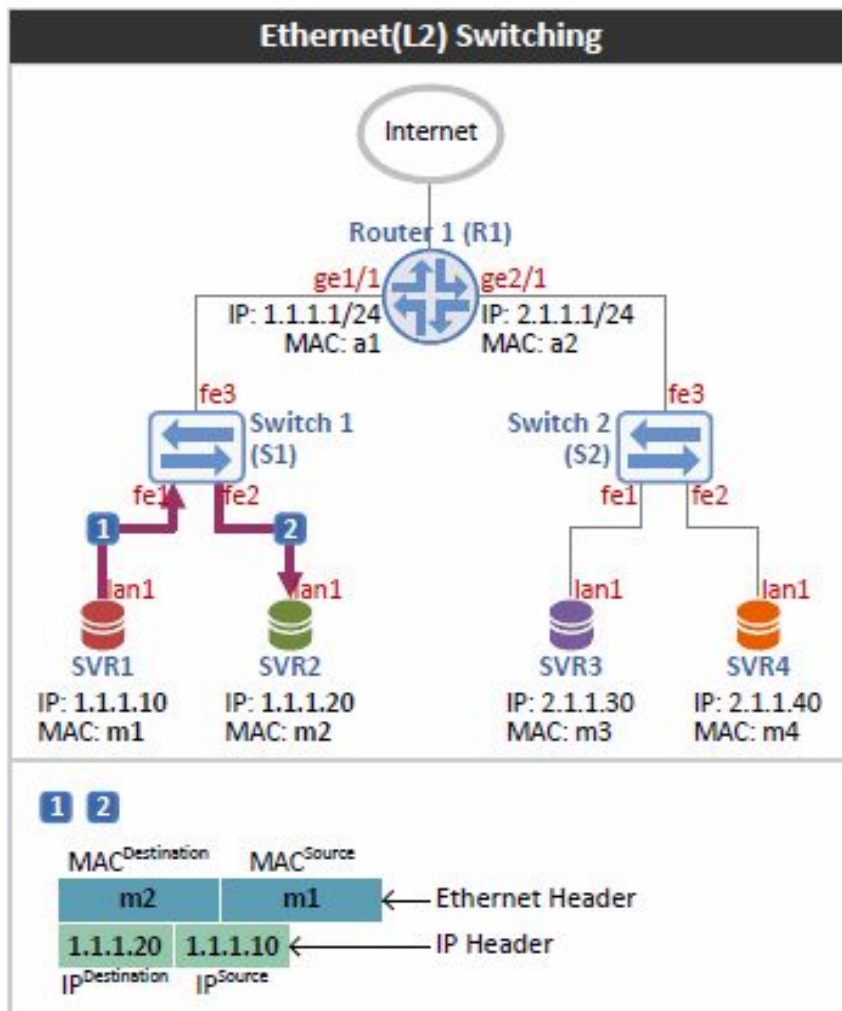


Утилиты:

- ping6
- traceroute6
- ip -6
- ...

/etc/gai.conf для настройки dual-stack

Двухуровневая адресация



взято с сайта: <https://netmanias.com>

ip, destination/policy-based routing

Маршрутизация в Linux основывается не только на ip назначениях, но и на политиках.

Как посмотреть маршруты в таблице main (r l = route list, пример сокращенных аргументов):

```
root@netology1:~# ip r l
default via 10.0.2.2 dev eth0 proto dhcp src 10.0.2.15 metric 100
# вместо default встречается 0.0.0.0/0
10.0.2.0/24 dev eth0 proto kernel scope link src 10.0.2.15
10.0.2.2 dev eth0 proto dhcp scope link src 10.0.2.15 metric 100
172.28.128.0/24 dev eth1 proto kernel scope link src 172.28.128.10
```

via – через адрес шлюза. В этом примере видно что он далеко не всегда 1-й в октете.

proto kernel – автоматически добавлено ядром при появлении адреса на интерфейсе.

Посмотреть таблицы (обратите внимание, rule вместо route):

```
root@netology1:~# ip rule list
0:          from all lookup local
32766:      from all lookup main
32767:      from all lookup default
```

ip route list без параметров аналогичен вызову **ip route show table main**.

Гибкость policy-based routing велика и проявляет себя при наличии нескольких каналов подключения, необходимости балансировке трафика и т.д., но для начала хорошо знать хотя бы о том какие таблицы вообще есть и как их можно посмотреть.

Как tcpdump поможет при проблемах с маской?

```
root@netology1:~# ip -4 a s eth1 | grep inet
inet 172.28.128.10/24 scope global eth1
root@netology2:~# ip -4 a s eth1 | grep inet
inet 172.28.128.60/24 scope global eth1
root@netology1:~# ip r get 172.28.128.60
172.28.128.60 dev eth1 src 172.28.128.10 uid 0
```

curl 172.28.128.60 работает. Ломаем на хосте netology2 сеть, устанавливая заведомо некорректную маску:

```
root@netology2:~# ip addr del 172.28.128.60/24 dev eth1
root@netology2:~# ip addr add 172.28.128.60/30 dev eth1 # вместо /24
root@netology2:~# ip -4 a s eth1 | grep inet
inet 172.28.128.60/30 scope global eth1
```

Если запустить в этот момент tcpdump на обоих хостах, будет видно, что SYN доходит до сервера, но он не отвечает клиенту SYN+ACK, и клиент перепосылает SYN:

```
root@netology1:~# tcpdump -nn -i eth1
10:34:38.393610 IP 172.28.128.10.50700 > 172.28.128.60.80: Flags [S]...
10:34:39.423744 IP 172.28.128.10.50700 > 172.28.128.60.80: Flags [S]...
root@netology2:~# tcpdump -nn -i eth1
10:34:37.621593 IP 172.28.128.10.50700 > 172.28.128.60.80: Flags [S]...
10:34:38.652240 IP 172.28.128.10.50700 > 172.28.128.60.80: Flags [S]...
```

Это только один из примеров, как tcpdump быстро покажет, на каком хосте случились проблемы с сетью (ведь зная как устанавливается TCP соединение мы сразу видим, на какой стадия происходит проблема и на каком хосте).



Оптимизации ТСР

TCP Window Size – размер окна rwnd

Надежность в TCP гарантирована подтверждающими сообщениями о доставленных сегментах. Нетрудно представить, что при известных нам значениях MSS (пусть даже 8960 байт при 9K jumbo-frame), ожидать подтверждения каждого переданного сегмента – крайне неэффективно; скорость передачи равнялась бы MSS / RTT (отправили данные, за $\frac{1}{2} RTT$ они дошли, за $\frac{1}{2} RTT$ дошло подтверждение) в лучшем случае.

Тогда при 30 мс RTT (round-trip time, время пинга), даже имея MTU 9000 байт, мы бы не смогли утилизировать и 10 Мбит/с линк:

если за 30 мс мы подтверждаем передачу 8960 байт, за 1000 мс – 298667 байт ~ 292 Кб/с, тогда как 10 Мбит/с линк способен пропускать до ~1,2 Мб/с.

Решение – отправлять данных без подтверждения больше, чем один сегмент. Для этого в TCP заголовках предусмотрено поле Window Size – 16 битное число, которое обозначает размер окна в байтах (0-65535).

Анимация для иллюстрации: ccs-labs.org

Flow Control – контроль потока

Более полная утилизация соединения – не единственное предназначение окна.

Представим, что к коммутатору подключены 2 участника сетевого обмена. Один из них обладает современной сетевой картой с поддержкой стандарта 10GBASE-T – она позволяет передавать данные со скоростью 10 Гбит/с.

Другой хост располагает скромной 1 Гбит/с сетевой картой. Эти компьютеры ничего не знают о физических характеристиках линков друг друга. Тем не менее, скорость передачи данных от более быстрой сетевой карты к медленной будет не выше 1 Гбит/с. За счет чего так происходит?

Окно не является статически заданной величиной, а регулируется в процессе соединения.

```
19:31:43.029422 IP 192.168.88.187.60729 > 87.250.250.242.443: Flags [P.], seq 1:205,
ack 1, win 2053, options [nop,nop,TS val 965883632 ecr 3037463738], length 204
19:31:43.036066 IP 87.250.250.242.443 > 192.168.88.187.60729: Flags [.], ack 205, win
169, options [nop,nop,TS val 3037463749 ecr 965883632], length 0
...
19:31:43.038373 IP 192.168.88.187.60729 > 87.250.250.242.443: Flags [.], ack 2797, win
2026, options [nop,nop,TS val 965883639 ecr 3037463750], length 0
```

Вне зависимости от того, сколько данных готов отправить инициатор соединения, он не отправит сегментов больше, чем размер окна приемника. В примере видно что окно клиента уменьшается при передаче.

Congestion Control – контроль заторов, размер окна cwnd

Помимо важного протокола скользящего окна и Flow Control, призванных не допустить переполнения буфера на клиенте, в TCP существует множество реализаций концепции Congestion Control для справедливого распределения производительности самой сети в которой работает TCP.

Далеко не всегда есть техническая или экономическая возможность обеспечить ПС промежуточных узлов без переподписки: например стоечный коммутатор обеспечивает честные 10 Гбит/с каждому из 48 клиентских портов, но это не всегда значит что его собственный аплинк имеет 480 Гбит/с, а главное, что устройство за ним эти 480 Гбит/с при максимальной активности хостов сможет переварить. По этой причине TCP контролирует не только окно на прием, но и окно при передаче, стремясь одновременно максимально утилизировать возможности сети, приемника, но и не помешать при этом по возможности соседям.

Посмотреть, какой из алгоритмов используется сейчас: `net.ipv4.tcp_congestion_control`
В сетях с разными физическими характеристиками (сочетания ПС, значения и вариативности RTT) оптимальный алгоритм будет свой.

Подробнее:

[TCP Slow Start](#)

[Cubic TCP](#)



DNS, Domain Name System

DNS, система доменных имен

В отличие от протоколов TCP/IP, которые легли в основу современного Интернета, DNS появился не сразу.

```
vagrant@netology1:~$ head -n1 /etc/hosts
127.0.0.1 localhost
```

Именно так в простых текстовых файлах раньше распространялись соответствия IP адресов доменным именам и до сих пор этот файл имеет приоритет перед прочими способами разрешения имени по-умолчанию. Кстати, порядок резолва определен здесь:

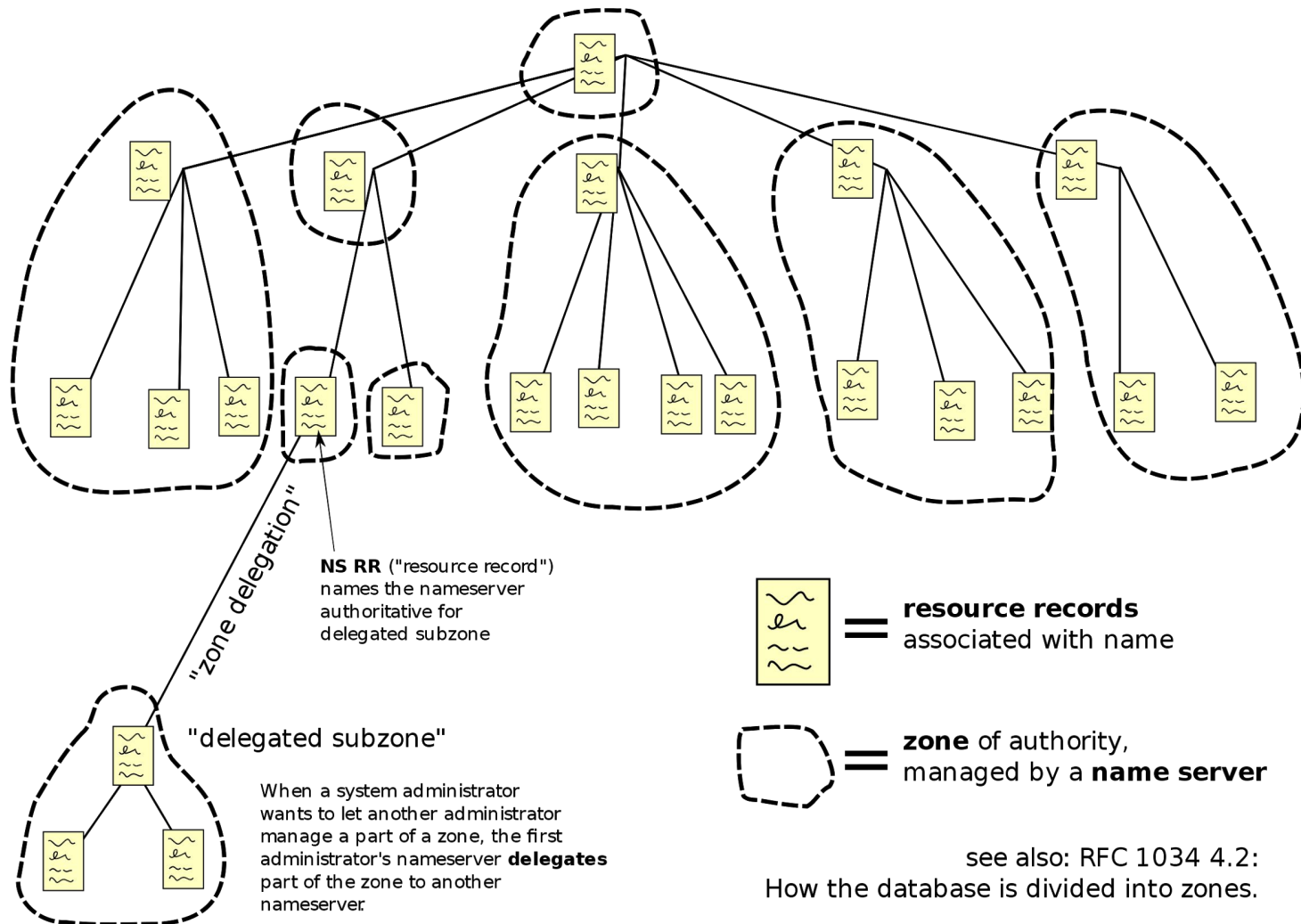
```
vagrant@netology1:~$ grep hosts /etc/nsswitch.conf
hosts:                files dns
```

files – **/etc/hosts**, dns – правила из **/etc/resolv.conf**.

В зависимости от дистрибутива, конфигурация и способ создания resolv.conf сильно отличается. Это может быть и статический текстовый файл, и динамически создаваемый специализированными службами, такими как **systemd-resolved** в случае с нашей демонстрационной Ubuntu 20.04.

DNS, пространство доменных имен

Domain Name Space



Разрешение DNS

- root, корневые серверы

DNS имя заканчивается точкой (пустой суффикс), которую обычно опускают: `www.ru[.]`

Эту зону обслуживают 13 корневых серверов, которые знают о...

```
vagrant@netology1:~$ dig NS . @1.1.1.1 +noall +answer | head -n1
.                510166      IN      NS      a.root-servers.net.
```

- TLD – top level domain серверах. Это серверы, которым делегированы зоны верхнего уровня, такие как ru, com и т.д.

```
vagrant@netology1:~$ dig NS ru. @a.root-servers.net. +noall +authority | head -n1
ru.              172800      IN      NS      a.dns.ripn.net.
```

Мы спрашиваем один из корневых серверов о зоне ru – т.к. он отвечает за знания о ней, он дает *авторитативный* ответ (в прошлом примере Cloudflare не является первоисточником о зоне . поэтому его ответ не является авторитативным).

- Авторитативные DNS серверы для подзон. `a.dns.ripn.net.` все еще не имеет знаний о конкретном адресе, в чем можно убедиться:

```
vagrant@netology1:~$ dig A www.ru. @a.dns.ripn.net. +noall +authority | head -n1
www.ru.          345600      IN      NS      ns3.nic.ru.
```

А вот ns3.nic.ru. в свою очередь уже умеет:

```
vagrant@netology1:~$ dig A www.ru. @ns3.nic.ru. +answer | egrep '(^www.ru.|^;; flags)' | head -n2
;; flags: qr aa rd; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1
www.ru.          3600 IN      A       31.177.76.70
```

Итеративные и рекурсивные запросы

Разрешение имени подобным образом, когда сначала у корневого сервера мы узнаем TLD серверы, у TLD сервера – NS для искомой зоны, у них – адрес, называется *рекурсивным* запросом. Обычно нас не интересуют эти многократные запросы между уровнями, мы хотим просто получить ответ:

```
vagrant@netology1:~$ dig +short A netology.ru | head -n1  
104.26.9.143
```

Поэтому в resolv.conf будет присутствовать сервер, который делает эту работу за нас, – рекурсивный DNS сервер. Не все серверы такими являются; если прийти с рекурсивным запросом к TLD серверу, он явно ответит что такие запросы обслуживать не будет:


```
vagrant@netology1:~$ dig +recurse A netology.ru @a.root-servers.net  
...  
;; WARNING: recursion requested but not available  
;; AUTHORITY SECTION:  
ru.                172800      IN      NS      a.dns.ripn.net.
```

и предложит обратиться к авторитативным серверам.

Итеративный тип запросов как раз применяется рекурсором, когда он производит поэтапное разрешение имени.

Некоторые особенности DNS

- FQDN и PQDN имена
- Разные типы записей: A (IPv4), AAAA (IPv6), PTR (обратная запись), SRV (предоставляемый сервис), TXT (текстовое поле для любых целей) и т.д.
- Обратная зона [in-addr.arpa](#) (56.34.12.10.in-addr.arpa. IN PTR host1.example.net.)
- Кеширование: в современных системах число DNS запросов может быть очень велико, поэтому присутствует кеширование на разных уровнях – не забывайте об этом при отладке.
- Реализация DNS может отличаться в разных языках программирования – если классические приложения используют библиотечные функции вроде `getaddrinfo`, приложения на языке `golang`, например, часто используют собственный многопоточный резолвер с отличным поведением.
- Множественные опции при резолве: секция `options` в `man resolv.conf`, далеко не только сами `nameservers` указываются в `resolv.conf`.
- `resolv.conf` на хостовой системе и в контейнерах – разные вещи! Различные системы контейнеризации могут проводить разные манипуляции для работы DNS в контейнерах, на это можно влиять опциями.
- Полезная опция `dig` – `+trace`, покажет поэтапный резолв имени.



HTTP, HyperText Transfer Protocol

HTTP – базовый протокол веба

Несколько поколений:

- 1.1 – текстовый, поверх TCP
- 2 – бинарный, поверх TCP
- 3 – бинарный, поверх QUIC и UDP, не является стандартом по состоянию на июль 2020

Принцип работы HTTP одинаковый для всех поколений: это клиент-серверный протокол, не имеющий концепции состояния, но имеющий концепцию соединения.

В HTTP несколько ключевых моментов, один из которых – методы. По сути, это управляющие веб-сервером команды от клиента:

- **GET** – запрос ресурса
- **HEAD** – запрос ресурса без тела ответа (только заголовки)
- **POST** – передача данных с целью изменения ресурса
- **PUT** – передача данных с целью замены ресурса
- **DELETE** – удаление ресурса
- **OPTIONS** – параметры для контроля соединения
- ... и некоторые другие.

HTTP, коды ответов

Вторая важная часть протокола – коды ответов. В ответ на запрос с методом, веб-сервер отвечает сообщением с кодом одной из групп:

1xx информационный ответ – запрос был получен, ожидается обработка

- 100 continue в ответ на PUT, продолжайте

2xx успешно – запрос был получен, корректно обработан и исполнен

- 200 ok – на успешный GET
- 201 created – на успешный PUT

3xx перенаправление – необходимы дальнейшие действия со стороны клиента для успешного выполнения запроса

- 301 moved permanently – постоянное перенаправление, например, с http на https

4xx ошибка на стороне клиента – проблемы в запросе со стороны клиента

- 400 некорректный синтаксис запроса
- 404 запрошен несуществующий ресурс
- 418 i am a teapot – сервер не может приготовить кофе, потому что он чайник

5xx ошибка на стороне сервера – сервер не смог выполнить корректный запрос от клиента

- 502 bad gateway – прокси-сервер, к которому вы обратились, получил ошибочный ответ от его риалов (бэкендов)
- 504 gateway timeout – прокси-сервер не дождался ответа от бэкенда за установленный промежуток времени



HTTP, заголовки

Третья важная часть протокола – заголовки. Они могут дополнять как запрос клиента, так и ответ сервера.

Например, при 3xx редиректах, веб-сервер сообщает клиенту именно заголовком Location о том, по какому адресу тому следует перепослать запрос.

Клиент же, когда делает GET, дополняет запрос заголовком Host чтобы дать понять веб-серверу, к какому виртуальному хосту он хочет обратиться (на одном веб-сервере могут обслуживаться разные домены).

В заголовках передаются Set-Cookie и Cookie, разнообразные поля для управления кэшированием статических ресурсов на странице и т.д.

HTTP на примерах GET и POST

```
vagrant@netology1:~$ telnet netology.ru 80
Trying 172.67.75.22...
Connected to netology.ru.
Escape character is '^]'.
GET / HTTP/1.1
Host: netology.ru
```

```
HTTP/1.1 301 Moved Permanently
Connection: keep-alive
Expires: Sun, 05 Jul 2020 09:41:20 GMT
Location: https://netology.ru/
```

Веб-сервер сообщает, что клиент должен проследовать на HTTPS ресурс <https://netology.ru> вместо запрошенного изначально ресурса <http://netology.ru>

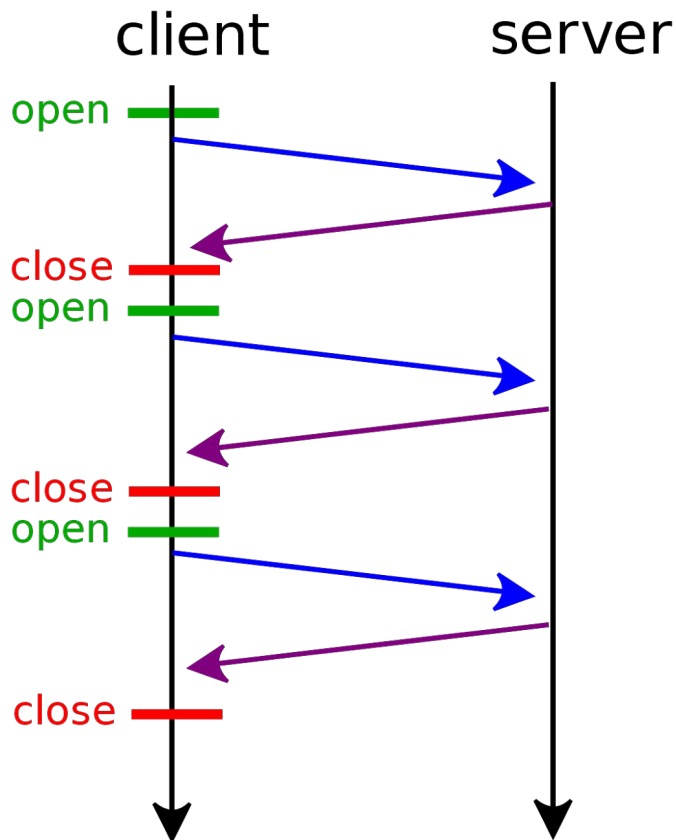
HEAD запрос заголовков curl: curl -I (по-умолчанию подразумевается GET, -I – HEAD)
POST запрос:

```
root@netology2:~/postdemo# curl -d "@data.json" -X POST -H "Content-Type:
application/json" http://localhost:3000/data
{ test_key1: 'test_datapoint1', test_key2: 'test_datapoint2' }

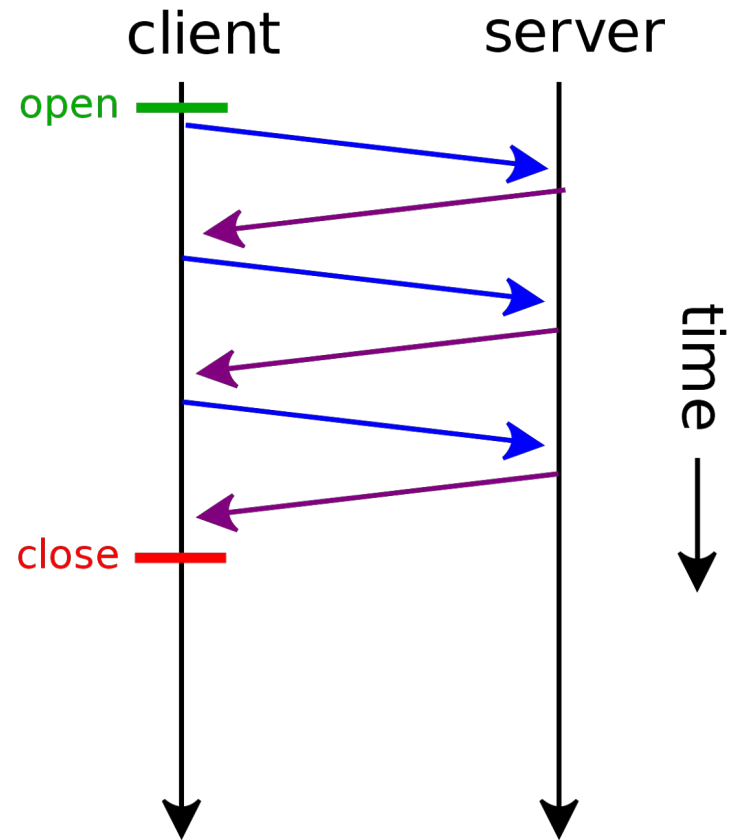
root@netology2:~/postdemo# curl -d '{"test_key1":"'test_datapoint1'",
"test_key2":"'test_datapoint2'"}' -H "Content-Type: application/json" -X POST
http://localhost:3000/data
```

HTTP/1.1 keep-alive

Multiple Connections



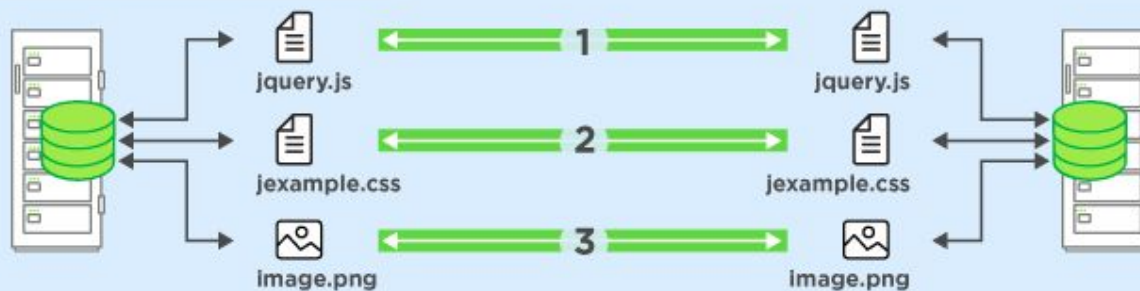
Persistent Connection



HTTP/2 – мультиплексирование соединений

HTTP 1.1

3 TCP CONNECTIONS



HTTP/2

1 TCP CONNECTION



Итоги

- Разобрались в том, как работает маска сети IP,
- в разных вариантах записи адресов,
- познакомились с калькулятором ipcalc,
- узнали о некоторых важных оптимизациях TCP,
- разобрались в основах DNS и HTTP.



Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задавайте **в чате** мессенджера Slack.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

**Задавайте вопросы и
пишите отзыв о лекции!**

Андрей Вахутинский