OBSD.RU

Home

Оптимизация правил пакетного фильтра

printable page

PF: Оптимизация правил пакетного фильтра

Перевод: Сгибнев Михаил

В идеальном случае, пакетный фильтр не должен затрагивать валидный трафик. Запрещенные пакеты должны блокироваться, разрешенные должны проходить сквозь фильтр, как-будто его и нет.

Есть одна маленькая тонкость: на обработку пакета тратится время и ресурсы. Поскольку любое устройство обладает конечными ресурсами, то когда они будут потреблены полностью - пакеты будут теряться.

Большинство протоколов, таких как TCP, довольно хорошо переносят такую временную задержку. Вы можете достигнуть высоких скоростей передачи данных по TCP даже по каналам, с задержкой в несколько сотен миллисекунд. С другой стороны, в сетевых играх даже несколько десятков миллисекунд обычно бывает слишком много. Потеря пакета - вообще страшная проблема, так как при большой патере пакетов работа TCP серьезно ухудшается.

В этой статье я расскажу о том, как определить то, что PF становится ограничивающим фактором и что можно предпринять для повышения скорости обработки правил.
Смысл packet rate
Обычно для сравнения производительности сети используется параметр "бит/с". Но в случае pf данный параметр неприменим. Реальным ограничивающим фактором является не пропускная способность сети, а число пакетов (packet rate), обрабатываемых в единицу времени. Устройство, без какого-либо напряжения обрабатывающее 1500 байтные пакеты на скорости 100Мб/с может быть перегружено обычными 40 байтовыми пакетами на скорости всего-лишь 10 Мб/с. Если в первом случае обрабатывать приходится порядка 8 000 пакетов в секунду, то во втором случае трафик составляет около 32 000 пакетов в секунду, что в четыре реза больше.

Для детального понимания процесса давайте рассмотрим как трафик проходит через хост. Пакеты передаются на сетевую карту и сохраняются в буфере. Когда буфер наполняется, карта вызывает прерывание и драйвер сетевой карты копирует данные из буфера карты (mbufs) в память ядра. После того как пакет передан в mbuf, время выполнения всех дальнейших операций, проводимых стеком TCP/IP не зависит от размера пакета, так как производится анализ только его заголовка. То же верно и для pf, который принимает решение пропускать или блокировать пакет. Если пакет необходимо пропустить, то стек TCP/IP передаст его драйверу сетевой карты, который извлечет его из mbuf и передаст в линию.

Большинство этих операций имеет сравнительно высокую стоимость из расчета на один пакет, но очень низкую стоимость исходя из размера пакета. Следовательно, обработка большого пакета лишь ненамного дороже обработки маленького пакета.

Некоторые ограничения накладываются программными и аппаратными средствами вне pf. Например, машины

класса i386 не могут обработать более чем 10,000 прерываний в секунду, независимо от скорости процессора, что вызвано ограничениями архитектуры. Некоторые сетевые карты генерируют одно прерывание на каждый пакет. Следовательно, хост начнет терять пакеты, когда количество пакетов превысит приблизительно 10 000 пакетов в секунду. Другие карты, например, более дорогие, гигабитные имеют большие встроенные буферы, что позволяет им связывать несколько пакетов в одно прерывание. Следовательно, выбор аппаратных средств может наложить некоторые ограничения, которые не может преодолеть никакая оптимизация pf.

Когда pf является узким местом

Ядро передает пакеты pf последовательно, один за другим. В то время, когда pf решает судьбу пакета, поток пакетов через ядро останавливается и в течение этого короткого промежутка времени прочитанные с сетевой карты пакеты должны записываться в буфер. Если pf будет думать слишком долго, то буферы будут переполнены и пакеты начнут теряться. Цель оптимизации набора правил pf заключается в уменьшении времени, затрачиваемого на обработку каждого пакета.

Создать большой набор правил для подтверждения всего вышесказанного можно следующим образом:

```
$ i=0; while [ $i -lt 100 ]; do \
    printf "block from any to %d.%d.%d.%d\n" \
      `jot -r -s " " 4 1 255`; \
    let i=i+1; \
  done | pfctl -vf -

block drop inet from any to 151.153.227.25
block drop inet from any to 54.186.19.95
block drop inet from any to 165.143.57.178
...
```

Это самый плохой вариант, поскольку делает невозможным автоматическую оптимизацию. Поскольку каждое правило содержит случайный адрес, pf должен просмотреть весь набор правил и оценить соответствие пакета каждому правилу. Загрузка набора правил, который состоит исключительно из нескольких тысяч таких правил, и затем генерация устойчивого потока пакетов, которые должны быть отфильтрованы, создаст значительную нагрузку даже самой быстрой машине. В то время как хост находится под нагрузкой, проверьте количество прерываний:

```
$ vmstat -i
```

И загрузку CPU

```
$ top
```

Это даст вам возможность понаблюдать, как хост реагирует на перегрузку и поможет вам определять подобные признаки, когда используется ваш собственный набор правил. Вы можете использовать те же самые инструментальные средства, чтобы позже проверить эффекты оптимизации.

Теперь кинемся в другую крайность. Полностью отключим pf:

```
$ pfctl -d
```

После чего сравните значения vmstat и top.

Это позволит вам понять, чемо можно хотеть от оптимизации набора правил. Если ваша машина без pf уже показывает недостаточную производительность, то вряд-ли оптимизация приведет к фантастическому результату, если ее не подкрепить заменой или улучшением каких-либо других компонентов.

Если у вас уже есть наббор правил и вы задаетесь вопросом, нужно ли тратить время на их оптимизацию, повторите вышеприведенный тест и сравните результаты с обоими критическими случаями. Ниже приведены некоторые рекомендации, руководствуясь которыми можно повысить эффективность работы.

В некоторых случаях набор правил не оказывает существенного влияния на загрузку хоста, но возможны другие неприятные моменты, например когда соединение долго устанавливается или пропускная способность ниже ожидаемой.

И последнее: если ваш набор правил не нагружает машину, не блокирует нужные пакеты, то рекомендую оставить все как есть. Довольно часто наборы, написаные линейно, без всякой оптимизации, работают без потерь пакетов. Ручная оптимизация сделает набор правил значительно более трудночитаемым человеком, имея незначительное влияние на работу.
Фильтры "Stateful"
Работа pf, главным образом, состоит из двух операций: поиск соответствия в наборе правил и поиск в таблице состояний.

Для каждого пакета pf сперва выполняет поиск по таблице состояний и если соответствующая запись будет найдена, то пакет немедленно передается. В противном случае, начинается поиск подходящего правила, которым определяется, блокировать ли или передать пакет. Если правило разрешающее, то можно создать запись в таблице состояний, используя опцию 'keep state'.

При прямой фильтрации, без использования 'keep state', каждый раз выполняется операция поиска в наборе правил. Это единственная самая дорогостоящая по времени операция, выполняемая в этом случае. Каждый пакет все еще вызывает поиск в таблице состояний, но так как таблица пуста, стоимость поиска является нулевой.

Stateful фильтрация подразумевает использование опций 'keep state' в правилах, когда создается запись в таблице состояний. Все пакеты, имеющие подходящую запись в таблице состояний, будут пропускаться немедленно, без проверки набора правил. В этом случае, только первый пакет каждой сессии вызывает просмотр набора правил, а последующие пакеты только проходят только через таблицу состояний.

Поиск по таблице значительно менее затратен, чем поиск по набору правил. Поскольку набор правил обычно просматривается снизу доверху, то стоимость поиска растет примерно пропорционально количеству правил. Помимо этого, в одном правиле могут сравниваться различные параметры пакета. Таблица состояния представляет собой древовидную структуру и поэтому стоимость обработки увеличивается логарифмически с увеличением числа записей.

У операций удаления и добавления записей в таблицу состояний также имеется некоторая стоимость, но она довольно незначительна, если сравнивать со стоимостью проверки нескольких пакетов списком правил. Для определенных типов соединений, таких как DNS-запросы, где каждое соединение содержит два пакета, стоимость создания записи в таблице превысит стоимость поиска по списку правил, но если число пакетов значительно, что характерно для большинства TCP-сессий, то использование таблицы состояний становится оправданным.

Короче говоря, вы можете составлять свой набор правил исходя из стоимости или сессий или пакетов. Например я вижу следущие значения счетсиков:

```
$ pfctl -si
```

```
State Table                          Total             Rate
  searches                        172507978          887.4/s
  inserts                           1099936            5.7/s
  removals                          1099897            5.7/s
Counters
  match                             6786911           34.9/s
```

Это значит, что pf вызывается примерно 900 раз в секунду. Фильтрация у меня происходит на двух интерфейсах, таким образом у меня идет поток примерно в 450 пакетов с интерфейса, который проверяется на каждом интерфейсе. При этом проверка набора правил происходит 35 в секунду, а операции удаления и добавления записей в таблицу состояний происходит только 6 раз в секунду.

Для того, чтобы убедиться, что вы действительно создаете состояник для каждого сессии, ищите правила 'pass' в которых не используется 'keep state':

```
$ pfctl -sr | grep pass | grep -v 'keep state'
```

Удостоверьтесь, что используете по умолчанию политику 'block by default' для предотвращения прохода пакетов, не соответствующих ни одному правилу.
Обратная сторона фильтрации "Stateful"
Единственный недостаток использования таблицы состояний заключается в том, что она потребляет память. Примерно 256 байт на каждую запись. Когда pf не может выделить память для новой записи, то соответствующий пакет блокируется и увеличивается счетчик out-of-memory.

```
$ pfctl -si
Counters
memory                                0            0.0/s
```

Память для таблицы состояния выделяется из пула ядра, называемого 'pfstatepl'. Для простмотра состояния пула можно использовать команду vmstat(8):

```
$ vmstat -m
Memory resource pool statistics
Name        Size Requests Fail Releases Pgreq Pgrel Npage Hiwat Minpg Maxpg Idle
pfstatepl    256  1105099    0  1105062   183   114    69   127     0 625   62
```

Разница между 'Requests' и 'Releases' соответствует числу выделенной памяти для записей таблицы состояний, а количество записей можно посмотреть командой:

```
$ pfctl -si
State Table                          Total             Rate
  current entries                       36
```

Другие счетсики, показываемые pfctl можно сбросить командой pfctl -Fi.

Ядру доступна не вся память. Количество доступной ядру RAM зависит от архитектуры, опций ядра и его версии. При использовании OpenBSD 3.6 для архитектуры i386, ядро может использовать до 256 МБ памяти. До версии 3.6 этот предел был намного ниже в случае i386. Вы можете иметь 8GB оперативной памяти, но pf

будет доступна лишь малая часть этого пространства.

Если pf действительно выходит на тот предел, когда pool_get(9) не может выделить память, все выглятит не так уж весело. Вся система становится нестабильной и в конечном счете терпит крах. Это проблема не самого pf, а всего механизма управления пулом памяти ядра.

PF способен самостоятельно контролировать число записей в таблице, используя pool_sethardlimit (9). Значение этого параметра также можно увидеть по vmstat -m. По умолчанию это значение равно 10,000 записей, что является безопасным значением для любого хоста. Увидеть это значение можно с помощью команды:

```
$ pfctl -sm
states     hard limit  10000
src-nodes  hard limit  10000
frags      hard limit    500
```

Если вам необходимо иметь больше записей в таблице состояний, то установить это значение можно в pf.conf:

```
set limit states 10000
```

Для определения предела размера таблицы состояний, когда происходят ошибки выделения пула, не существует четкой формулы, поэтому можно предложить только экспериментальный путь, когда задав лимит вы создаете большое количество записей и проверяете стабильность работы системы.

Для того, чтобы не сгущать краски, скажу: если у вас имеется 512 МБ или больше оперативной памяти, то выделив ядру 256MB вы сможете иметь примерно 500,000 записей в таблице состояний. Только представьте, что если каждой записи будет соответствовать только один пакет в 10 секунд, вы получите поток в 50 000 пакетов в секунду.

Более вероятным случаем будет, когда вы не ожидаете большого когличества записей. Но установленный лимит может быть превышен, например в ходе DoS атаки. Злоумышленник может забить таблицу состояний только для того, чтобы легитимные пользователи не могли создать новые записи.

Есть несколько способов решения данной проблемы:

Вы можете ограничить число записей, создаваемых определенным правилом:

```
pass in from any to $ext_if port www keep state (max 256)
```

Это позволит ограничить число создваемых записей этим правилом 256 записями, при этом сохранив возможность изменять таблицу состояний другими правилами. Можно также ограничить число записей с одного адреса:

```
pass keep state (source-track rule, max-src-states 16)
```

Определить время жизни записи можно следущим способом:

```
$ pfctl -st
tcp.opening                30s
```

В данном случае оно составляет 30 секунд. Установить свое значение можно с помощью pf.conf:

```
set timeout tcp.opening 20
```

Также, можно задать срок жизни записи для конкретного правила:

```
pass keep state (tcp.opening 10)
```

Есть несколько готовых наборов значений таймаутов, которые можно выставить в pf.conf:

```
set optimization aggressive
```

Кроме того, есть адаптивные таймауты, что означает, что они не являются константами, они изменяются в зависимости от количества записей в таблице состояний. Например:

```
set timeout { adaptive.start 6000, adaptive.end 12000 }
```

pf будет использовать постоянные значения таймаутов, пока число записей в таблице меньше чем 6 000. Когда количество записей между 6 000 и 12 000, все таймауты линейно масштабируются от 100% при 6 000 до 0% при 12 000 записях, то есть при 9 000 все таймауты уменьшаются на 50%.

В данный момент вы должны определить, сколько записей будет поддерживать ваша таблица. Учтите, что данный лимит может быть превышен в ходе некоторых атак и для этого случая вы должны определить стратегию таймаутов. В самом крайнем случае, pf отбросит пакет и увеличит счетчик out-of-memory.
Оценка набора правил
Набор правил - это линейный список, просматриваемый сверху вниз для данного пакета. Каждое правило соответствует или не соответствует пакету, в зависимости от критериев в правиле и соответствующих значений в пакете.

Поэтому, в первом приближении, стоимость набора правил оценки растет с числом правил. Это не совсем верно по причинам, которые мы скоро разберем, но в целом соответствует истине. Набор из 10 000 правил почти наверняка вызовет намного больше нагрузки на хост, чем 100 правил. Самая очевидная оптимизация должна уменьшить число правил.
Построение набора правил с максимальным пропуском шагов
Первая причина, по которой набор правил может быть дешевле, чем правила поотдельности - пропущенные правила. Когда правила загружены, pf производит прозрачную и автоматическую оптимизацию, что можно рассмотреть на следующем примере:

```
1. block in all
2. pass in on fxp0 proto tcp from any to 10.1.2.3 port 22 keep state
3. pass in on fxp0 proto tcp from any to 10.1.2.3 port 25 keep state
4. pass in on fxp0 proto tcp from any to 10.1.2.3 port 80 keep state
5. pass in on fxp0 proto tcp from any to 10.2.3.4 port 80 keep state
```

В данном случае разрешается TCP пакеты с fxp0 на адрес 10.2.3.4 по некоему порту. pf начнет оценку пакета с данным набором правил, ища первое, полностью соответствующее, правило. Сравнение начинается со второго правила, в котором совпадают 'in', 'on fxp0', 'proto tcp', 'from any' и не совпадает 'to 10.1.2.3'. Таким образом мы переходим на третье правило.

Но pf знает, что третье и четвертое правило также определяет тот же самый критерий 'to 10.1.2.3', нарушающий соответствие, поэтому сразу переходит к пятому, экономя несколько сравнений.

Вообразите, что пакет был UDP вместо TCP. Первое правило соответствовало бы, после чего мы бы перешли на второе правило. В нем не соответствует критерий критерий 'proto tcp'. Так как последующие правила также определяют тот же самый критерий 'proto tcp', все они могут быть безопасно пропущены, не влияя на конечный результат.

Давайте посмотрим, как pf анализирует набор правил. Каждое правило содержит список критериев, таких как 'to 10.1.2.3', определяющих совпадение пакета с адресом назначения. Для каждого критерия в каждом правиле, pf считает количество правил, находящихся сразу ниже текущего, имеющих тоже самое значение соответствующего критерия. Это может быть ноль, если критерии не совпадают. Полученные значения сохраняются в памяти для последующего использования. Они называются "шагами пропуска", поскольку они указывают pf, сколько последующих правил можно пропустить, если критерий в текущем правиле не соответствует пакету.

Порядок подсчета критериев следующий:

1. **interface ('on fxp0')**
2. **direction ('in', 'out')**
3. **address family ('inet' or 'inet6')**
4. **protocol ('proto tcp')**
5. **source address ('from 10.1.2.3')**
6. **source port ('from port < 1024')**
7. **destination address ('to 10.2.3.4')**
8. **destination port ('to port 80')**

Если правило соответствует, то мы продвигаемся к следующему, в противном случае берется первый несовпадающий критерий и определяется, сколько правил мы можем пропустить.

Очевидно, что порядок следования правил сильно влияет на количество шагов пропуска. Например:

1. **pass on fxp0**
2. **pass on fxp1**
3. **pass on fxp0**
4. **pass on fxp1**

Здесь шаг пропуска будет равен 0, так как нет повторяющихся параметров.

Как несложно догадаться, правила можно упорядочить следующим образом:

1. **pass on fxp0**
2. **pass on fxp0**
3. **pass on fxp1**
4. **pass on fxp1**

В этом случае шаг пропуска составит 1 для первого и третьего правила.

Тут есть маленькое различие, когда происходит сравнение пакета для интерфейса fxp2. Перед тем, как переупорядочить, оцениваются все четыре правила, так как ни одно из них нельзя пропустить. После переупорядовачивания оцениваются только правила 1 и 3, а 2 и 4 можно пропустить.

В данном примере разница в быстродействии может быть незначительной, но представьте себе набор, состоящий из 1,000 правил, применяемым к двум различным интерфейсам. В случае, когда правила сгруппированы по интерфейсам, pf может смело пропустить половину, при этом стоимость оценки уменьшится на 50%, вне зависимости от того, какой тип трафика оценивается.

Следовательно, вы можете помочь pf эффективно использовать шаги пропуска упорядочивая правила в соответствии с критериями, в том порядке, каком они указаны выше, начиная с номера интерфейса.

Для проверки эффекта выполните команду:

```
$ pfctl -gsr
```

pfctl выдаст расчетные значения шага пропуска для каждого критерия в каждом правиле, например:

```
@18 block return-rst in quick on kue0 proto tcp from any to any port = 1433
[ Skip steps: i=38 d=38 f=41 p=27 sa=48 sp=end da=43 ]
```

В этом выводе "i" означает интерфейс, "d" - назначение,"f" - семейство адресов и т.д. 'i=38', как несложно догадаться, указывает на то, что можно пропустить 38 правил, если пакет не соответствует данному интерфейсу.

Также можно оценить число сравнений для каждого правила:

```
$ pfctl -vsr
```

pfctl рассчитывает, сколько времени оценивалось каждое правило, сколько пакетов и байтов этому правилу соответствовало и сколько записей в таблице состояний было создано. Когда правило было пропущено, в соответствии с шагами пропуска, счетчик оценки не увеличивался.
Использование таблицы для списка адресов
Использование списков в фигурных скобках позволяет писать очень компактные правила в pf.conf, например:

```
pass proto tcp to { 10.1.2.3, 10.2.3.4 } port { ssh, www }
```

Написание списков не приводит к загрузке в ядро одного правила. Вместо этого, pfctl развертывает одно правило в несколько. В данном случае:

```
$ echo "pass proto tcp to { 10.1.2.3, 10.2.3.4 } port { ssh, www }" |
    pfctl -nvf -
pass inet proto tcp from any to 10.1.2.3 port = ssh keep state
pass inet proto tcp from any to 10.1.2.3 port = www keep state
pass inet proto tcp from any to 10.2.3.4 port = ssh keep state
pass inet proto tcp from any to 10.2.3.4 port = www keep state
```

Короткий синтаксис в pf.conf не предает реальную стоимость оценки этого правила. Если у вас имеется всего 10 правил, но в ядре они разворачиваются в 100, то стоимость оценки будет как для 100 правил. Чтобы увидеть, какие правила действительно оцениваются, выполните:

```
$ pfctl -sr
```

Для определенного типа списков, адресов, есть контейнер в ядре, называемый таблицей. Например:

```
pass in from { 10.1.2.3, 10.2.3.4, 10.3.4.5 }
```

Список адресов может быть выражен как таблица:

```
table  const { 10.1.2.3, 10.2.3.4, 10.3.4.5 }
pass in from
```

Эта конструкция может быть загружена как единственное правило (и таблица) в ядро, тогда как версия без использования таблицы расширилась бы до трех правил.

В течение оценки правила, ссылающегося на таблицу, pf сделает поиск исходного адреса пакета в таблице, чтобы определить, соответствует ли правило пакету. Этот поиск очень дешев, и стоимость не увеличивается с числом записей в таблице.

Если список адресов является большим, выгода работы одной оценки правила с одним поиском в таблице против одной оценки правила для каждого адреса довольно существенна. В качестве эмпирического правила, таблицы более дешевы, когда список содержит шесть или больше адресов.
Использование quick для прерывания сравнения при соответствии правила
Когда встречается правило, соответствующее пакету, pf не прекращает оценку набора правил (в отличие от некоторых других пакетных фильтров), а доходит до конца списка. В результате к пакету применяется последнее подходящее правило.

Опция 'quick' может применяться для прекращения оценки набора при совпаденни правила. Когда 'quick' используется на каждом правиле, pf начинает вести себя как пакетный фильтр, работающий по первому совпавшему правилу, но это не является поведением по умолчанию.

Например, pf фильтрует пакеты, проходящие через любой интерфейс, включая виртуальные интерфейсы, типа loopback. Если, как и большинство людей, вы не намереваетесь фильтровать трафик с loopback, приведенное правило может сохранить немало времени:

```
set skip on { lo0 }
```

Набор правил может содержать сотни строк, не соответствующих loopback и такой трафик мог бы пройти только как разрешенный по умолчанию в самом конце списка. В результате для каждого пакета с loopback проводилось бы сравнение со всем списком.

Обычно, вы помещаете правило 'quick' в верхней части набора правил, полагая, что это позволит сэкономить на проверки нижележащих правил. Но в тех случаях, когда правило не соответствует пакету, размещение правила сверху приводит к лишней оценке. Короче говоря, при размещении правил стоит учитывать объем и тип проходящего трафика, помещая наиболее часто востребуемые правила в верхней части списка.

Вместо того, чтобы гадать, с какой частотой сравниваются правило и в каком порядке их разместить, вы можете использовать оценку правила и соответствующие счетчики, которые можно посмотреть командой:

```
$ pfctl -vsr
```

Когда вы увидите правила, находящиеся в верхней части списка, но имеющие мало совпадений, то это первые кандидаты на перемещение вниз.

Якоря с условной оценкой
Якорь - это поднабор правил. Вы можете загрузить весь набор правил в якоря, и заставить их оцениваться из главного набора.

Другой способ смотреть на них состоит в том, чтобы сравнить правила фильтрации с языком программирования. Без якорей, весь ваш код находится в единственной основной функции, главном наборе правил. Якоря, тогда тогда являются подпрограммами, храня свой код в отдельных функциях, которые вы можете вызывать из основной функции.

Начиная с OpenBSD 3.6, вы можете также создавать вложенные структуры якорей, строя иерархию подпрограмм, и вызывать одну подпрограмму из другой. В OpenBSD 3.5 и более ранних, иерархия могла быть только одного уровня вложенности, то есть вы могли иметь много подпрограмм, но вызвать их могли только из главного набора правил.

Например:

```
pass in proto tcp from 10.1.2.3 to 10.2.3.4 port www
pass in proto udp from 10.1.2.3 to 10.2.3.4
pass in proto tcp from 10.1.2.4 to 10.2.3.5 port www
pass in proto tcp from 10.1.2.4 to 10.2.3.5 port ssh
pass in proto udp from 10.1.2.4 to 10.2.3.5
pass in proto tcp from 10.1.2.5 to 10.2.3.6 port www
pass in proto udp from 10.1.2.5 to 10.2.3.6
pass in proto tcp from 10.1.2.6 to 10.2.3.7 port www
```

Вы можете разделить набор на два поднабора, один из которых будет содержать правила для UDP и называться "udp-only":

```
pass in proto udp from 10.1.2.3 to 10.2.3.4
pass in proto udp from 10.1.2.4 to 10.2.3.5
pass in proto udp from 10.1.2.5 to 10.2.3.6
```

И второй поднабор для TCP, называемый "tcp-only":

```
pass in proto tcp from 10.1.2.3 to 10.2.3.4 port www
pass in proto tcp from 10.1.2.4 to 10.2.3.5 port www
pass in proto tcp from 10.1.2.4 to 10.2.3.5 port ssh
pass in proto tcp from 10.1.2.5 to 10.2.3.6 port www
pass in proto tcp from 10.1.2.6 to 10.2.3.7 port www
```

И оба их можно вызвать из главного набора правил:

```
anchor udp-only
anchor tcp-only
```

Быстродействия это все же не прибавит. Фактически есть даже некоторые потери, связанные с вызовом этих поднаборов.

Но якоря могут содержать критерии фильтрации, совсем как правила pass/block:

```
anchor udp-only in on fxp0 inet proto udp
anchor tcp-only in on fxp0 inet proto tcp
```

В результате мы получаем, что якорь вызывается только в случае совпадения с критерием и стоимость вычисляется исходя из оценки правил в основном наборе.
Пусть pfctl работает за вас
В OpenBSD 3.6 можно произвести оптимизацию правил используя pfctl -o. Оптимизатор анализирует набор правил и производит необходиые модификации, которые не изменяют функционал набора правил.

Сначала, pfctl разбивает набор на блоки смежных правил таким способом, что переупорядочение правил в пределах одного блока не может затронуть результат оценки для любого пакета.

Например, правила в следующем блоке могут быть произвольно переупорядочены:

```
pass proto tcp to 10.1.2.3 port www keep state
pass proto udp to 10.1.2.3 port domain keep state
pass proto tcp to 10.1.0.0/16 keep state
```

Но так бывает редко. В большинстве случаев порядок необходим, например:

```
block log all
block from 10.1.2.3
pass from any to 10.2.3.4
```

Изменение порядка правил в этом наборе каждый раз будет приводить к разным эффектам. Если поменять 1 и 2 правила, то пакеты с 10.1.2.3 будут блокироваться и запись об этом будет занесена в журнальный файл. Если поменять местами последние два правила, то будут блокироваться пакеты с 10.1.2.3 на 10.2.3.4.

pfctl стремится переупорядочить правила таким образом, чтобы шаг пропуска был максимальным:

```
$ cat example
pass proto tcp from 10.0.0.3 to 10.0.0.8
pass proto udp from 10.0.0.1
pass proto tcp from 10.0.0.2
pass proto tcp from 10.0.0.4
pass proto udp from 10.0.0.6
pass proto tcp from 10.0.0.3 to 10.0.0.7

$ pfctl -onvf example
pass inet proto tcp from 10.0.0.3 to 10.0.0.8
pass inet proto tcp from 10.0.0.3 to 10.0.0.7
pass inet proto tcp from 10.0.0.2 to any
pass inet proto tcp from 10.0.0.4 to any
pass inet proto udp from 10.0.0.1 to any
pass inet proto udp from 10.0.0.6 to any
```

Когда встречаются дублирующие правила, они будут удалены:

```
$ cat example
pass proto tcp from 10.0.0.1
pass proto udp from 10.0.0.2
pass proto tcp from 10.0.0.1

$ pfctl -onvf example
pass inet proto tcp from 10.0.0.1 to any
pass inet proto udp from 10.0.0.2 to any
```

Избыточные правила также удаляются:

```
$ cat example
pass proto tcp from 10.1/16
pass proto tcp from 10.1.2.3
pass proto tcp from 10/8

$ pfctl -onvf example
pass inet proto tcp from 10.0.0.0/8 to any
```

Несколько правил объединяются в одно правило, используя таблицу где только возможно и выгодно:

```
$ cat example
pass from 10.1.2.3
pass from 10.2.3.4
pass from 10.3.4.5
pass from 10.4.5.6
pass from 10.5.6.7
pass from 10.8.9.1

$ pfctl -onvf example
table <_automatic_0> const { 10.1.2.3 10.2.3.4 10.3.4.5 10.4.5.6
                             10.5.6.7 10.8.9.1 }
pass inet from <_automatic_0> to any
```

Если вызвать pfctlc параметром -oo, то он также сверяется со счетчиками оценки, которые показывает команда pfctl -vsr, чтобы переупорядочить правила 'quick' согласно соответствующей частоте.

Эта опция очень консервативна в выполнении любых изменений, выполняя только те изменения, которые не затронут результат прохождения набора правил при любых обстоятельствах для любого пакета. Преимуществом этого является то, что оптимизатор может использоваться безопасно с любым набором правил. Недостаток метода состоит в том, что pfctl не смеет изменять то, что вы могли бы изменить сами, меняя сами правила. Если вы сначала вручную переупорядочите правила, то можно потенциально улучшить выгоду, которую может дать оптимизатор.

Самый легкий способ видеть, что опции -o или -oo сделают с вашим набором правил, состоит в том, чтобы сравнить результат с оригиналом, выполнив следующие команды:

```
$ pfctl -nvf /etc/pf.conf >before
$ pfctl -oonvf /etc/pf.conf >after
```

```
$ diff -u before after
```

Если вы вручную оптимизировали набор правил, то изменения вряд-ли произведут впечатление.