1. Обзор существующих решений

1.1. PQ

Самый простой метод обеспечения дифференцированного обслуживания. Эта техника использует множество очередей. Очереди обслуживаются с разными видами приоритетов и очереди с более высоким приоритетом обслуживаются в первую очередь. Пакеты помещаются в одну из очередей в соответствии с их классификацией. Пакеты обрабатываются из одной очереди, только если очереди с более высокими приоритетами пусты. Внутри очередей пакеты обрабатываются в порядке FIFO. В случае переполнения отбрасываются пакеты из очереди с более низким приоритетом. [packethandling.pdf]

[image]

Приемущества.

- Для software-based маршрутизаторов PQ предоставляет относительно небольшую вычислительную нагрузку на систему в сравнении с более сложными ДО.
- PQ позволяет маршрутизаторам организовывать буферизацию пакетов и обслуживать один класс трафика отдельно от других.

Недостатки.

- Если количество высокоприоритизированного трафика стало избыточным, низкоприоритезированный трафик будет отбрасываться.
- Избыточный (misbehaving) высокоприоритетный трафик может значительно увеличивать задержку (delay) и джиттер (jitter) для другого высокоприоритетного трафика.
- Не решается проблема с TCP и UDP, когда TCP-трафику даётся высокий приоритет и он пытается поглотить всю пропускную способность (bandwidth).

1.2. CBQ

Class Based Queueing (CBQ) – это классовая дисциплина обслуживания, которая реализует богатое иерархическое разделение канала между классами (rich link sharing hierarchy of classes). [man tc-cbq]

Главная идея за CBQ – это планировка пакетов в очередях и гарантия определённой скорости передачи. Если в очереди нет пакетов, её пропускная способность становится доступной для других очередей. Сила этого метода состоит в том, что он позволяет справляться (among flows) со значительно различными требованиями к пропускной способности канала (bandwidth). Это сделано путём назначения определённого процента Link bandwidth каждой очереди. CBQ также избегает проблему застоя пропускной способности (bandwidth

starvation) с помощью PQ метода, так как по крайней мере один пакет обслуживается из каждой очереди в течение service round.

Алгоритм CBQ состоит в следующем. Сначала пакеты классифицируются в классы обслуживания в соответствии с определёнными критериями и сохраняются в соответствующей очереди. Очереди обслуживаются циклически (round robin). Различное количество пропускной способности может быть назначено для каждой очереди двумя различными способами: с помощью позволения очереди отправлять более чем один пакет на каждый цикл обслуживания или с помощью позволения очереди отправлять только один пакет за цикл, но при этом очередь может быть обслужена несколько раз за цикл.

Преимущества.

- Позволяет контролировать количество пропускной способности для каждого класса обслуживания.
- Каждый класс получает обслуживание, вне зависимости от других классов. Это помогает избегать проблемы PQ, когда при избытке высокоприоритезированного трафика низкоприоритезированный не обслуживался вообще.

Недостатки.

- Честное выделение пропускной способности (fair allocation bandwidth) происходит, только если пакеты из всех очередей имеют сравнительно одинаковый размер. Если один класс обслуживания содержит пакет, который длиннее остальных, этот класс обслуживания получит большую пропускную способность (bandwidth), чем сконфигурированное значение.
- Судя по словам реализовавших её в Linux, она не очень Linux-way, в Linux реализована довольно приближёно (судя по комментариям в sch_cbq.c).
- Похожа на HTB (по крайней мере в реализации), но хуже.

1.3. HTB

Hierarchical Token Bucket (HTB) – ДО с иерархическим разделением канала между классами. HTB состоит из произвольного числа иерархически организованных TBF (token bucket filter). HTB позволяет создать динамическую структуру разделения канала основанного на дереве, где узел определяет класс, а его потомки – некоторый подкласс; каждому классу выделяется пропускная способность, которая разделяется между подклассами. Если подкласс не использует назначенную пропускную способность, она распределяется между его соседями; если на более высоком уровне иерархии класс не использует всю пропускную способность, она распределяется уже между соседями этого класса. Сами очереди содержатся в листьях дерева.

[Что-нибудь ещё, информация доступна]
Приемущества.

- Иерархическая структура предоставляет гибкую возможность конфигурировать трафик.
- Не зависит от характеристик интерфейса и не нуждается в знании о лежащей в основе пропускной способности выходного интерфейса [man tc-htb].

Недостатки.

- Видимо, связаны с TBF.

1.4. HFSC

HFSC – Hierarchical Fair-Service Curve – иерархический алгоритм планирования пакетов, основанный на математической модели честных кривых обслуживания (Fair Service Curve), где под термином “кривая обслуживания” подразумевается зависящая от времени неубывающая функция, которая служит нижней границей количества обслуживания, предоставляемого системой [из сетевого исчисления, ссылка на какой-либо ресурс такого рода].

HFSC ставит перед собой цели:

- гарантировать точное выделение пропускной способности и задержки для всех листовых классов (критерий реального времени);
- честно выделять избыточную пропускную способность как указано классовой иерархией (критерий разделения канала);
- минимизировать несоответствие кривой обслуживания идеальной модели и действительного количество обслуживания. [man 7 tc-hfsc]

Алгоритм планировки основан на двух критериях: критерий реального времени (real-time) и критерий разделения канала (link-sharing). Критерии реального времени используются для выбора пакета в условиях, когда есть потенциальная опасность, что гарантия обслуживания для листового класса нарушается. В ином случае используется критерий разделения канала.

HFSC использует три типа временных параметров: время крайнего срока (deadline time), «подходящее» время (eligible time) и виртуальное время (virtual time). Время крайнего срока назначается таким образом, чтобы, если крайние сроки всех пакетов сессии выполнены, его кривая была гарантирована. «Подходящее» время используется для выбора критерия планировки для следующего пакета. Виртуальное время показывает нормализованное количество обслуживания, которое получил класс. Виртуальное время присуще всем вершинам дерева классов, так как является важным параметром при критерии разделение

канала, при котором должно минимизироваться несоответствие между виртуальным временем класса и временами его соседей (так как в идеальной модели виртуальное время соседей одинаково); при выборе критерия разделения канала алгоритм рекурсивно, начиная с корня, обходит всё дерево, переходя в вершины с наименьшим виртуальным временем. Время крайнего срока и «подходящее» время используются дополнительно в листьевых классах, так как в этих вершинах непосредственно содержатся очереди. [SIGCOM97.pdf, описать более вразумительно].

Преимущества.

- Алгоритм основан на формальной модели с доказанными нижними границами.

Недостатки.

- Сложность.

1.5. Flow-based WFQ

WFQ (Weighted Fait Queueing) – динамический метод планировки пакетов, который предоставляет честное разделение пропускной способности всем потокам трафика. WFQ применяет вес, чтобы идентифицировать и классифицировать трафик в поток и определить, как много выделить пропускной способности каждому потоку относительно других потоков. WFQ на месте планирует интерактивный трафик в начало очереди, уменьшая там самым время ответа, и честно делит оставшуюся пропускную способность между остальными потоками. [link]

[Картинка]

Планировщик не нарушает порядка обработки пакетов, принадлежащих одному потока, даже в том случае, если они имеют различный приоритет. С этой целью поток реализуется в виде хэша, определяемого IP-адресом источника, IP-адресом цели назначения, полем протокола IP, номерами портов TCP/UDP и пятью битами байта ToS (Type of Service).Очередь потока обслуживается в соответствии с алгоритмом FIFO.

WFQ на основе потока использует для обработки каждого трафика так называемые очереди диалога (conversation queue). Поскольку память – конечный ресурс, число очередей диалога по умолчанию ограничено 256. Если число потоков превысит число очередей, допускается использование одной очереди для обработки нескольких потоков.[вегешн]

В целях планировки в WFQ длина очереди измеряется не в пакетах, а во времени, которое заняла бы передача всех пакетов в очереди. WFQ адаптирует количество потоков и выделяет одинаковое количество полосы пропускания каждому потоку. Поток с маленькими пакетами, которые обычно являются интерактивными потоками, получают лучшее обслуживание, потому что они не

нуждаются в большой полосе пропускания; также они получаются низкую задрежку, потому что у меньших пакетов меньшее время отправки (finish time). Время отправки – это сумма текущего времени и время, которое заняла бы отправка пакета. Текущее время ноль, если в очереди нет пакетов. WFQ поместит пакет в аппаратную очередь, основываясь на времени отправки в порядке возрастания.

Чтобы ввести вес в расчёт то, в каком порядке будут обслуживаться очереди, WFQ использует время окончания и приоритет IP (IP precedence). Вес расчитывается как время окончания, делённое на приоритет IP плюс один (во избежание деления на ноль). Однако для увеличения производительности в маршрутизаторах Cisco взамен времени отправки (finish time) используется размер пакета, так как он пропорционален времени; к тому же деление на приоритет IP заменяется на умножение фиксированного значения, просчитанного заранее (это сделано из-за того, что деление более трудная операция для CPU, чем умножение).

WFQ использует два метода отбрасывания пакетов: ранее (Early Dropping) и агрессивное (Aggressive Dropping) отбрасывания. Ранее отбрасывание срабатывает тогда, когда достигается congestive discard threshold (CDT); CDT – это количество пакетов, которые могут находиться в системе WFQ перед тем, как начнётся отбрасывание новых пакетов из самой длиной очереди; используется, чтобы начать отбрасывание пакетов из наиболее агрессивного потока, даже перед тем, как достигнется предел hold queue out (HQO). HQO – это максимальное количество пакетов, которое может быть во всех выходящих очередях в интерфейсе в любое время; при достижении HQO срабатывает агрессивный режим отбрасывания.

[Схема алгоритма отбрасывания (src/images)]

[`http://blog.globalknowledge.com/2010/02/12/quality-of-service-part-10`

Приемущества.

- Простая конфигурация.
- Гарантированная полоса пропускания для всех потоков.
- Отбрасывание пакетов из более агрессивных потоков.

WFQ страдает от нескольких недостатков.

- Трафик не может регулироваться на основе пользовательски определённых классов.
- WFQ не может предоставить фиксированную пропускную способность.
- WFQ поддерживается только на медленных каналах.

Эти ограничения были исправлены CBWFQ.

1.6. Class-Based WFQ

CBWFQ (Class-based weighted fair queueing) – основанный на классах взвешенный алгоритм равномерного обслуживания очередей[вагешен?]; является расширением функциональности дисциплины обслуживания WFQ, основанной на потоках, для предоставления определяемых пользователями классов трафика.

Class-Based WFQ – это мехнизм, использующийся для гарантировании пропускной способности для класса. Для CBWFQ класс трафика определяется на основе заданных криетриев соответствия: список контроля доступа (ACL), протокол, входящий интерфейс и т.п. Пакеты, удовлетворяющие криетриям класса, составляют трафика для этого класса. Дисциплина позволяет задавать до 64-х пользовательских классов.

После определения класса, ему назначаются характеристики, которые определяют политику очереди: пропускная способность, выделенная классу, максимальная длина очереди и так далее.[link] Алгоритм CBWFQ позволяет явано указать требуемую минимальную полосу пропускания для каждого класса трафика.[вегешн] Полоса пропускания используется в качестве веса класса. Вес можно задать в абсолютной (опция `bandwidth`), в процентной (опция `bandwidth percent`) и в доле от оставшиейся полосы пропускания (опция `bandwidth remaining precent`) величинах.

Кроме пользовательских классов CBWFQ предоставляет стандартный класс (default class), в который попадает весь трафик, который не был классифицирован. В стандартном классе управление очередью может осуществляться с помощью алгоритмов FIFO и FQ (Fair Queueing).

В случае переполнения очередей начинает работать алгоритм отбрасывания пакетов. В качестве политики отбрасывания пакетов по умолчанию используется отбрасывание конца очереди (Tail Drop), однако допускается сконфигуривать работу алгоритм взвешенного произвольного раннего обнаружения (Weighted Random Early Detection, WRED) для каждого класса.

[вставить картинку]

Больше Cisco-реализации.

2. Реализация Class-Based WFQ в ядре Linux

2.1. Управление трафиком

Расскать про traffic control (tc) и как он связан с конфигурацией qdisc + Netlink.

2.2. Описание устройства подсистемы планировки в ядре Linux

[Схема пути пакета]

В ядре сетевое устройство описывается структурой `struct net_device`, которая содержит указатель на ассоциированную с устройством дисциплину обслу-

живания, описываемая структурой `struct Qdisc`.

Опишем важные для понимания подсистемы поля `Qdisc`.

- enqueue

В общем, дисциплина обслуживания – это чёрный ящик, который может ставить пакеты в очереди и вынимать их из очереди, когда устройство готово к отправке, в порядка и во время, определёнными спрятанным в ящике алгоритмом. В ядре Linux дисциплины обслуживания представляются в качестве модулей ядра, которые реализуют предоставляемый ядром интерфейс.

API предосавтляет две функции: `register_qdisc(struct Qdisc_ops *ops)` и обратную – `unregister_qdisc(struct Qdisc_ops *ops)`, которые регистрируют и разрегистрируют дисциплину обслуживания. Важно отметить, что обе эти функции принимают в качестве аргумента структуру `struct Qdisc_ops`, которая явным образом идентифицирует дисциплину обслуживания в ядре.

Структура `Qdisc_ops` помимо метаинформации содержит указатели на функции, которые должен реализовывать модель дисциплины обслуживания для корректной работы в ядре.

Следующие функции определяют работу алгоритма.

- `enqueue`
  `int enqueue(struct sk_buff *skb, struct Qdisc *sch, struct sk_buff **to_free );`
  Фукнкция добавляет пакет в очередь. Если пакет был отброшен, функция возвращает код ошибки, говорящий о том, был отброшен пришедший пакет или иной, чьё место занял новый.

- `dequeue`
  `struct sk_buff *dequeue(struct Qdisc * sch);` Функция, возвращающая пакет из очереди на отправку. Дисциплина может не передавать пакет при вызове этой функции по решению алгоритма, в таком случае вернув нулевой указатель; однако то же значение алгоритм возвращает в случае, если очередь пуста, поэтому в таком случае дополнительно проверяется длина очереди.

- `peek`
  `struct sk_buff *peek(struct Qdisc * sch);` Функция возвращает пакет из очереди на отправку, не удаляя его из реальной очереди, как это делает функция `dequeue`.

Следующие функции определяют настройки алгоритма.

- `init`
  `int init(struct Qdisc *sch, struct nlattr *arg);`
  Функция инициализирует вновь созданный экземпляр дисциплины обслуживания `sch`. Вторым аргументом функции является конфигурация

дисциплины обслуживния, передаваемая в ядро с помощью подсистемы Netlink.

- `change`
  `int change(struct Qdisc *sch, struct nlattr *arg);`
  Функция изменяет текущие настройки дисциплины обслуживания.

Дополнительно рассказать про классификаторы.

Таким образом при составлении алгоритма CBWFQ явный упор должен делаться на том, как выражается ДО в ядре через `enqueue` и `dequeue`.

2.3. Алгоритм CBWFQ

2.3.1. Стркутура хранения данных Class-Based WFQ

2.3.2. Псевдокод

Описать enqueue, classify и drop. Все реализуют drop двумя способами: через qdisc_drop() и через if net_xmit_drop_count() { qdisc_qstat_drop(sch) }.

function dequeue(Q)
    sort_queues_by_time(Q)
    q ← Q_1
    return qdisc_dequeue(q)
end function

function enqueue(Q, pkt)
    // Сначала нужно классифицировать пакет в очередь
    q ← classify(Q, pkt)
    //Если количество пакетов в очереди не превосходит пороговых значений,
    //то мехнизм отбрасывания не запускается.
    //Иначе следует запустить соответствующий мехнизм отбрасывания.
    //Так как мехнизмы довольно схожи, их вычисление можно объединить.
    if q.count < HQO ∧ q.count < CDT **then**
        qdisc_enqueue(q, pkt)
    else if pkt.time has the worst finish time **then**
        qdisc_drop(pkt)
    else
        if q.count > HQO **then**
            p_1 ← get_worst_finish_time(q)
            qdisc_drop(p_1)
        end if
        qdisc_enqueue(q, pkt)
    end if
end function

function classify(Q, pkt)

```
    // TODO
end function
```