**QoS**

Метрики качества сети:

* Потери
* Задержки
* Джиттер

**Потери**

Эта метрика говорит о том, сколько из отправленных источником пакетов дошло до адресата.

Причиной потерь может быть проблема в интерфейсе/кабеле, перегрузка сети, битовые ошибки, блокирующие правила ACL.

Что делать в случае потерь решает приложение. Оно может проигнорировать их, как в случае с телефонным разговором, где запоздавший пакет уже не нужен, или перезапросить его отправку — так делает TCP, чтобы гарантировать точную доставку исходных данных.

**Задержки**

Это время, которое необходимо данным, чтобы добраться от источника до получателя. Совокупная задержка складывается из следующих компонентов.

* Задержка сериализации (Serialization Delay) — время, за которое узел разложит пакет в биты и поместит в линк к следующему узлу. Она определяется скоростью интерфейса. Так, например, передача пакета размером 1500 байтов через интерфейс 100Мб/с займёт 0,0001 с, а на 56 кб/с — 0,2 с.
* Задержка передачи сигнала в среде (Propagation Delay) — результат печально известного ограничения скорости распространения электромагнитных волн. Физика не позволяет добраться из Нью-Йорка до Томска по поверхности планеты быстрее чем за 30 мс (фактически порядка 70 мс).
* Задержки, вносимые QoS — это томление пакетов в очередях (Queuing Delay) и последствия шейпинга (Shaping Delay). Об этом мы сегодня будем говорить много и нудно в главе Управление скоростью.
* Задержка обработки пакетов (Processing Delay) — время на принятие решения, что делать с пакетом: lookup, ACL, NAT, DPI — и доставку его от входного интерфейса до выходного. Но в день, когда Juniper в своём M40 разделил Control и Data Plane, задержкой обработки стало можно пренебречь.

Связанный с задержкой термин, но не являющийся синонимом — RTT (Round Trip Time) — это путь туда-обратно. При пинге и трассировке вы видите именно RTT, а не одностороннюю задержку, хотя величины и имеют корреляцию.

**Джиттер**

Разница в задержках между доставкой последовательных пакетов называется джиттером.

**Неупорядоченная доставка**

Ряд приложений, таких как телефония, NAS, CES экстремально чувствительны к неупорядоченной доставке пакетов, когда они приходят к получателю не в том порядке, в котором были отправлены. Это может приводить к потере связности, ошибкам, повреждению файловой системы.

И хотя неупорядоченная доставка не является формально характеристикой QoS, но определённо относится к качеству сети.

**Полоса пропускания**

Её не выделяют, как метрику качества сети, поскольку фактически её недостаток выливается в три указанные выше. Однако в наших реалиях, когда некоторым приложениям она должна быть гарантирована или, наоборот, по договору должна быть ограничена, а например MPLS TE её резервирует на всём протяжении LSP, упомянуть её, хотя бы как слабую метрику, стоит.

**Три модели обеспечения QoS**

**Best Effort** — никакой гарантии качества. Все равны.

**IntServ** — гарантия качества для каждого потока. Резервирование ресурсов от источника до получателя.

**DiffServ** — нет никакого резервирования. Каждый узел сам определяет, как обеспечить нужное качество.

**Best Effort (BE)**

Самый простой подход к реализации QoS, с которого начинались IP-сети и который практикуется и по сей день — иногда потому что его достаточно, но чаще из-за того, что никто и не думал о QoS.

Кстати, когда вы отправляете трафик в Интернет, то он там будет обрабатываться как BestEffort. Поэтому через VPN, прокинутые поверх Интернета (в противовес VPN, предоставляемому провайдером), может не очень уверенно ходить важный трафик, вроде телефонного разговора.

В случае BE — все категории трафика равны, никакому не отдаётся предпочтение. Соответственно, нет гарантий ни задержки/джиттера, ни полосы.

**IntServ**

Заблаговременное резервирование ресурсов для потока на всём протяжении от источника до получателя.

Название отражает стремление в одной сети одновременно предоставлять услуги для реал-тайм и не-реал-тайм типов трафика, предоставив, при этом первым приоритетное право использования ресурсов через резервирование полосы. Возможность переиспользования полосы, на которой все и зарабатывают, и благодаря чему IP выстрелил, при этом сохранялась.

**DiffServ**

* Никакой сигнализации
* Основан на агрегированной классификации трафика, вместо акцента на потоках, клиентах итд.
* Имеет ограниченный и детерминированный набор действий по обработке трафика данных классов.

**Механизмы DiffServ**

Трафик делится на классы. Пакет на входе в каждый узел классифицируется и к нему применяется набор инструментов, который по-разному обрабатывает пакеты разных классов, таким образом обеспечивая им разный уровень сервиса.

В основе DiffServ лежит концепция PHB — Per-Hop Behavior. Каждый узел по пути трафика самостоятельно принимает решение о том, как вести себя относительно пришедшего пакета, на основе его заголовков.

Модель поведения определяется набором инструментов и их параметров: Policing, Dropping, Queuing, Scheduling, Shaping.

Используя имеющиеся модели поведения, сеть может предоставлять различные классы сервиса (Class of Service).

То есть разные категории трафика могут получить разный уровень сервиса в сети путём применения к ним разных PHB.

Соответственно прежде всего нужно определить к какому классу сервиса относится трафик — классификация (**Classification**). Каждый узел самостоятельно классифицирует поступающие пакеты.

После классификации происходит измерение (**Metering**) — сколько битов/байтов трафика данного класса пришло на маршрутизатор.

На основе результатов пакеты могут окрашиваться (**Coloring**): зелёный (в рамках установленного лимита), жёлтый (вне лимита), красный.

Если необходимо, далее происходит полисинг (**Policing**). Полисер на основе цвета пакета назначает действие по отношению к пакету — передать, отбросить или перемаркировать.

После этого пакет должен попасть в одну из очередей (**Queuing**). Для каждого класса сервиса выделена отдельная очередь, что и позволяет их дифференцировать, применяя разные PHB. Но ещё до того, как пакет попадёт в очередь, он может быть отброшен (**Dropper**), если очередь заполнена.

На выходе из очереди работает шейпер (**Shaper**), задача которого очень похожа на задачу полисера — ограничить трафик до заданного значения.

Все очереди в итоге должны слиться в единый выходной интерфейс. Поэтому есть специальный диспетчер (**Scheduler**), который циклически вынимает пакеты из разных очередей и отправляет в интерфейс (**Scheduling**).

Далее пакеты уже выходят на интерфейс, где происходит преобразование пакетов в поток битов — сериализация (**Serialization**) и далее сигнал среды.

В DiffServ поведение каждого узла независимо от остальных, нет протоколов сигнализации, которые бы сообщили, какая на сети политика QoS. При этом в пределах сети хотелось бы, чтобы трафик обрабатывался одинаково. Для этого, во-первых, на всех маршрутизаторах, настраиваются одинаковые классы и PHB для них, а во-вторых, используется маркировка (Marking) пакета — его принадлежность определённому классу записывается в заголовок (IP, MPLS, 802.1q). И красота DiffServ в том, что следующий узел может довериться этой маркировке при классификации.

Такая зона доверия, в которой действуют одинаковые правила классификации трафика и одни модели поведения, называется домен DiffServ (**DiffServ-Domain**).

Таким образом на входе в домен DiffServ мы можем классифицировать пакет, промаркировать (Remark/Rewrite) его согласно правилам домена, и дальнейшие узлы будут доверять этой маркировке и не делать сложную классификацию. То есть явной сигнализации в DiffServ нет, но узел может сообщить всем следующим, какой класс нужно обеспечить этому пакету, ожидая, что тот доверится.

**Классификация и маркировка**

Существует три способа:

**Behavior Aggregate (BA)**

Просто довериться имеющейся маркировке пакета в его заголовке. Например, полю IP DSCP.

Называется он так, потому что под одной меткой в поле DSCP агрегированы различные категории трафика, которые ожидают одинакового по отношению к себе поведения. Например, все SIP-сессии будут агрегированы в один класс.

Количество возможных классов сервиса, а значит и моделей поведения, ограничено. Соответственно нельзя каждой категории (или тем более потоку) выделить отдельный класс — приходится агрегировать.

**Interface-based**

Всё, что приходит на конкретный интерфейс, помещать в один класс трафика. Например, мы точно знаем, что в этот порт подключен сервер БД и больше ничего. А в другой рабочая станция сотрудника.

**MultiField (MF)**

Проанализировать поля заголовков пакета — IP-адреса, порты, MAC-адреса. Вообще говоря, произвольные поля. Наиболее часто встречающийся тип классификации на входе в DS-домен. Не доверяем имеющейся маркировке, а на основе заголовков пакета присваиваем класс.

Зачастую это способ вообще «включить» QoS, в случае, когда отправители не проставляют маркировку.

На входе в DS-домен мы никому не доверяем, поэтому проводится классификация вторым или третьим способом: на основе адресов, протоколов или интерфейсов определяется класс сервиса и соответствующее цифровое значение.

На выходе из первого узла эта цифра кодируется в поле DSCP заголовка IP (или другое поле Traffic Class: MPLS Traffic Class, IPv6 Traffic Class, Ethernet 802.1p) — происходит ремаркировка.

Внутри DS-домена принято доверять этой маркировке, поэтому транзитные узлы используют первый способ классификации (BA) — наиболее простой. Никакого сложного анализа заголовков, смотрим только записанную цифру.

**Рекомендации IETF (категории трафика, классы сервиса и модели поведения)**

**DF — Default Forwarding**

Стандартная пересылка. Если классу трафика не назначена модель поведения специально, он будет обрабатываться именно по Default Forwarding.

Это Best Effort — устройство сделает всё возможное, но ничего не гарантирует. Возможны отбрасывания, разупорядочивание, непредсказуемые задержки и плавающий джиттер, но это не точно.

Такая модель подходит для нетребовательных приложений, вроде почты или загрузки файлов.

**AF — Assured Forwarding**

Гарантированная пересылка. Это улучшенный BE. Здесь появляются некоторые гарантии, например, полосы. Отбрасывания и плавающие задержки всё ещё возможны, но уже в гораздо меньшей степени.

**EF — Expedited Forwarding**

Экстренная пересылка.

Все ресурсы и приоритеты бросаются сюда. Это модель для приложений, которым нужны отсутствие потерь, короткие задержки, стабильный джиттер, но они не жадные до полосы. Как, например, телефония или сервис эмуляции провода (CES — Circuit Emulation Service).

Потери, разупорядочивание и плавающие задержки в EF крайне маловероятны.

**CS — Class Selector**

Это модели поведения, призванные сохранить обратную совместимость с IP-Precedence в сетях, умеющих в DS.

В IPP существуют следующие классы CS0, CS1, CS2, CS3, CS4, CS5, CS6, CS7.

Не всегда для всех них существует отдельный PHB, обычно их два-три, а остальные просто транслируются в ближайший DSCP класс и получают соответствующий ему PHB.

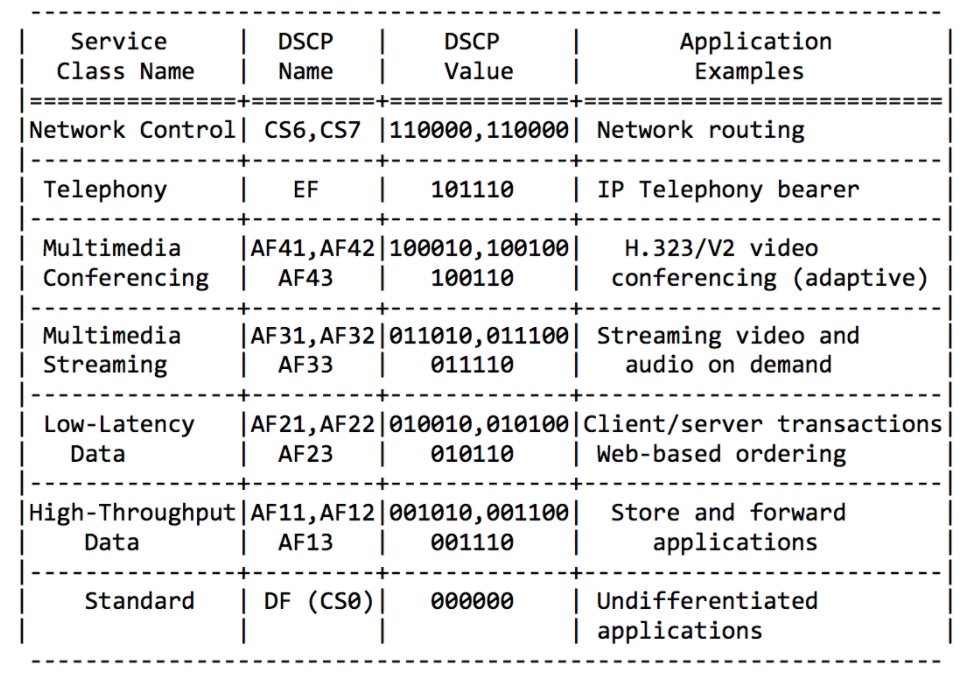
Так например, пакет, помеченный CS 011000, может классифицироваться как 011010.

Из CS наверняка в оборудовании сохранились только CS6, CS7, которые рекомендованы для NCP — Network Control Protocol и требуют отдельного PHB.

Как и EF, PHB CS6,7 предназначены для тех классов, что имеют очень высокие требования к задержкам и потерям, однако до некоторой степени толерантны к дискриминации по полосе.

Задача PHB для CS6,7 обеспечить уровень сервиса, который исключит дропы и задержки даже в случае экстремальной перегрузки интерфейса, чипа и очередей.

**Классы сервиса**



Классом DF (или BE) маркируется абсолютно нетребовательный трафик — он получает внимание по остаточному принципу.

PHB AF обслуживает классы AF1, AF2, AF3, AF4. Им всем нужно обеспечить полосу, в ущерб задержкам и потерям. Потери управляются битами Drop Precedence, поэтому они и называются AFxy, где x — класс сервиса, а y — Drop Precedence.

EF нужна некая минимальная гарантия полосы, но что важнее — гарантия задержек, джиттера и отсутствия потерь.

CS6, CS7 требуют ещё меньше полосы, потому что это ручеёк служебных пакетов, в котором всё же возможны всплески (BGP Update, например), но в ней недопустимы потери и задержки.

**Короткий итог по классификации**

На входе в узел пакет классифицируется на основе интерфейса, MF или его маркировки (BA).

Маркировка — это значение полей DSCP в IPv4, Traffic Class в IPv6 и в MPLS или 802.1p в 802.1q.

Выделяют 8 классов сервиса, которые агрегируют в себе различные категории трафика. Каждому классу назначается свой PHB, удовлетворяющий требованиям класса.

Согласно рекомендациям IETF, выделяются следующие классы сервисов, это CS1, CS0, AF11, AF12, AF13, AF21, CS2, AF22, AF23, CS3, AF31, AF32, AF33, CS4, AF41, AF42, AF43, CS5, EF, CS6, CS7 в порядке возрастания важности трафика.

Из них можно выбрать комбинацию из 8, которые реально можно закодировать в поля CoS.

Наиболее распространённая комбинация: CS0, AF1, AF2, AF3, AF4, EF, CS6, CS7 с 3 градациями цвета для AF.

Каждому классу ставится в соответствие PHB, которых существует 3 — Default Forwarding, Assured Forwarding, Expedited Forwarding в порядке возрастания строгости. Немного в стороне стоит PHB Class Selector. Каждый PHB может варьироваться параметрами инструментов, но об этом дальше.

**Инструменты PHB**

Существует три группы инструментов обеспечения QoS, которые активно манипулируют пакетами:

* **Предотвращение перегрузок** (Congestion Avoidance) — что делать, чтобы не было плохо.
* **Управление перегрузками** (Congestion Management) — что делать, когда уже плохо.
* **Ограничение скорости** (Rate Limiting) — как не пустить в сеть больше, чем положено, и не выпустить столько, сколько не смогут принять.

**Очереди**

После классификации пакеты помещаются в соответствующую данному классу очередь.

А далее одна очередь (с голосовыми данными) будет двигаться быстро, но с ограниченной полосой, другая помедленнее (потоковое вещание), зато с широкой полосой, а какой-то ресурсы достанутся по остаточному принципу.

Но в пределах каждой отдельной очереди действует то же правило — нельзя выдернуть пакет из середины — только из его изголовья.

Каждая очередь обладает определённой ограниченной длиной. С одной стороны это диктуется аппаратными ограничениями, а с другой тем, что нет смысла держать пакеты в очереди слишком долго. VoIP пакет не нужен, если он задержался на 200мс. TCP запросит переотправку, условно, после истечения RTT (настраивается в sysctl). Поэтому отбрасывание — не всегда плохо.

Разработчикам и дизайнерам сетевого оборудования приходится находить компромисс между попытками сохранить пакет как можно дольше и напротив не допустить бесполезной траты полосы, пытаясь доставить уже никому не нужный пакет.

В нормальной ситуации, когда интерфейс/чип не перегружен, утилизация буферов около нуля. Они амортизируют кратковременные всплески, но это не вызывает их продолжительного заполнения.

Если же трафика приходит больше, чем может обработать чип коммутации или выходной интерфейс, очереди начинают заполняться. И хроническая утилизация выше 20-30% — это уже ситуация, к которой нужно принимать меры.

**Предотвращение перегрузок (Congestion Avoidance)**

**Tail Drop** — наиболее простой механизм управления очередью — отбрасываем все вновь пришедшие пакеты, не помещающиеся в буфер.

**Head Drop** отбрасывает пакеты, которые стоят в очереди уже очень долго. Их уже лучше выбросить, чем пытаться спасти, потому что они, скорее всего, бесполезны. Зато у более актуальных пакетов, пришедших в конец очереди, будет больше шансов прийти вовремя. Плюс к этому Head Drop позволит не загружать сеть ненужными пакетами. Естественным образом самыми старыми пакетами оказываются те, что в самой голове очереди, откуда и название подхода.

Оба механизма работают с дифференциацией по очередям. То есть на самом деле не обязательно, чтобы весь буфер переполнился. Если 2-ая очередь пустая, а нулевая под завязку, то отбрасываться буду только пакеты из нулевой.

Tail Drop и Head Drop могут работать одновременно.

Две проблемы TCP трафика:

* Global TCP Synchronization - глобальной синхронизацией TCP
* TCP Starvation — истощение TCP.

**RED — Random Early Detection**

Early Detection — фиксируем потенциальную перегрузку;

Random — отбрасываем пакеты в случайном порядке.

Условно говоря, до заполнения буфера на 80% пакеты не отбрасываются совсем — вероятность 0%.

Начинаем отбрасывать случайные пакеты, когда очередь заполняется на 80%, заставляя некоторые TCP-сессии уменьшить окно и соответственно, скорость.

А если очередь заполнена на 90%, начинаем отбрасывать случайным образом 50% пакетов.

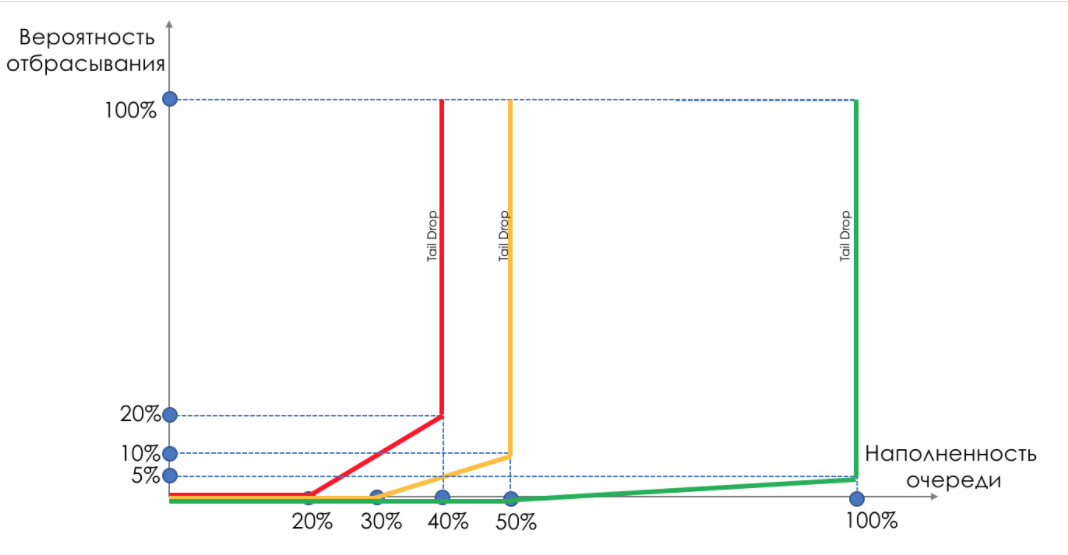
90% — вероятность растёт вплоть до Tail Drop (100% новых пакетов отбрасывается).

Побочным эффектом RED является и то, что агрессивные TCP-сессии скорее начнут притормаживать, просто потому что их пакетов много и они с бо́льшей вероятностью будут отброшены.

Неэффективность использования полосы RED решает тем, что притупляет гораздо меньшую часть сессий, не вызывая такую серьёзную просадку между зубьями. Ровно по этой же причине UDP не может оккупировать всё.

**WRED — Weighted Random Early Detection**

В пределах одной очереди у пакетов с разным приоритетом отбрасывания будут разные кривые. Чем ниже приоритет, тем больше вероятность того, что его прихлопнут.



Здесь есть три кривые:

Красная — менее приоритетный трафик (с точки зрения отбрасывания), жёлтая — более, зелёная — максимально.

Красный трафик начинает отбрасываться уже при заполнении буфера на 20%, с 20 до 40 он дропается вплоть до 20%, далее — Tail Drop.

Жёлтый стартует попозже — с 30 до 50 он отбрасывается вплоть до 10%, далее — Tail Drop.

Зелёный наименее подвержен: с 50 до 100 плавно растёт до 5 %. Далее — Tail Drop.

В случае с DSCP это могли бы быть AF11, AF12 и AF13, соответственно зелёная, жёлтая и красная.

Очень важно здесь то, что это работает с TCP и это абсолютно неприменимо к UDP. Либо приложение использующее UDP игнорирует потери, как в случае телефонии или потокового видео, и это отрицательно сказывается на том, что видит пользователь. Либо приложение само осуществляет контроль доставки и попросит повторно отправить тот же самый пакет. При этом оно совсем не обязано просить источник снизить скорость передачи. И вместо уменьшения нагрузки получается увеличение из-за ретрансмитов.

Именно поэтому для EF применяется только Tail Drop.

Для CS6, CS7 тоже применяется Tail Drop, поскольку там не предполагается больших скоростей и WRED ничего не решит.

Для AF применяется WRED. AFxy, где x — класс сервиса, то есть очередь, в которую он попадает, а y — приоритет отбрасывания — тот самый цвет.

Для BE решение принимается на основе преобладающего в этой очереди трафика.

В пределах одного маршрутизатора используются специальная внутренняя маркировка пакетов, отличная от той, что несут заголовки. Поэтому MPLS и 802.1q, где нет возможности кодировать Drop Precedence, в очередях могут обрабатываться с различными приоритетами отбрасывания.

**Управление перегрузками (Congestion Management)**

Когда всё плохо, приоритет обработки следует отдать более важному трафику. Важность каждого пакета определяется на этапе классификации.

Необязательно все буферы должны быть забиты, чтобы приложения начали испытывать проблемы.

Самый простой пример — голосовые пакетики, которые толпятся за большими пачками крупных пакетов приложения, скачивающего файл.

Это увеличит задержку, испортит джиттер и, возможно, вызовет отбрасывания. То есть мы имеем проблемы с обеспечением качественных услуг при фактическом отсутствии перегрузок.

Эту проблему призван решить механизм управления перегрузками (Congestion Management).

Трафик разных приложений разделяется по очередям, как мы уже видели выше.

Вот только в результате всё снова должно слиться в один интерфейс. Сериализация всё равно происходит последовательно.

аким же образом разным очередям удаётся предоставлять различный уровень сервисов?

По-разному изымать пакеты из разных очередей.

Занимается этим диспетчер.

Мы рассмотрим большинство существующих сегодня диспетчеров, начиная с самого простого:

* FIFO — только одна очередь, все в BE.
* PQ — приоритетная очерель.
* FQ — все равны.
* DWRR — все равны, но некоторые ровнее.

**FIFO — First In, First Out**

Простейший случай, по сути отсутствие QoS, — весь трафик обрабатывается одинаково — в одной очереди.

Пакеты уходят из очереди ровно в том порядке, в котором они туда попали, отсюда и название: первым вошёл — первым и вышел.

В FIFO все классы сливаются в CS0. Однако несмотря на все эти недостатки именно так работает сейчас Интернет.

**PQ — Priority Queuing**

Второй по сложности механизм и попытка разделить сервис по классам — приоритетная очередь.

Трафик теперь раскладывается в несколько очередей согласно своему классу — приоритету (например, хотя не обязательно, те же BE, AF1-4, EF, CS6-7). Диспетчер перебирает одну очередь за другой.

Сначала он пропускает все пакеты из самой приоритетной очереди, потом из менее, потом из менее. И так по кругу.

Диспетчер не начинает изымать пакеты низкого приоритета, пока не пуста высокоприоритетная очередь.

Он отлично подходит для таких видов трафика, как протокольные пакеты и голос, где задержки имеют критическое значение, а общий объём не очень большой.

Но тут и кроется недостаток PQ — если приоритетная очередь нагружена трафиком, диспетчер вообще никогда не переключится на другие.

**FQ — Fair Queuing**

Честность заключается в том, что диспетчер оперирует числом не пакетов, а числом битов, которые можно передать из каждой очереди.

Так агрессивный TCP-поток не может затопить интерфейс, и все получают равные возможности. FQ так и не был реализован на практике как механизм диспетчеризации очередей в сетевом оборудовании.

Недостатка тут три:

Первый — очевидный — это очень дорого — заводить очередь под каждый поток, считать вес каждого пакета и всегда беспокоиться о пропускаемых битах и размере пакета.

Второй — менее очевидный — все потоки получают равные возможности в плане пропускной способности. А если я хочу неравные?

Третий — неочевидный — честность FQ абсолютная: задержки у всех тоже равные, но есть потоки которым задержка важнее, чем полоса.

Например, среди 256 потоков присутствуют голосовые, это значит, что до каждого из них дело будет доходить только раз из 256-и.

**WFQ — Weighted Fair Queuing**

Каждая очередь наделялась весом и соответственно правом за один цикл отдавать трафика кратно весу.

**CBWFQ — Class-Based WFQ**

Weight в CBWFQ приобрел уже другой смысл. Вес назначался классам (не потокам) вручную в конфигурации по желанию администратора, потому что поле DSCP уже использовалось для классификации.

То есть DSCP определял в какую очередь помещать, а настроенный вес — сколько полосы доступно данной очереди.

**CBWFQ+LLQ — Low-Latency Queue**

Одна из очередей становится так называемой LLQ (очередь с низими задержками), и в то время, пока все остальные очереди обрабатываются диспетчером CBWFQ, между LLQ и остальными работает диспетчер PQ.

То есть пока в LLQ есть пакеты, остальные очереди ждут, растят свои задержки. Как только пакеты в LLQ кончились — пошли обрабатывать остальные. Появились пакеты в LLQ — про остальные забыли, вернулись к нему.

Внутри LLQ работает также FIFO, поэтому не стоит туда пихать всё, что ни попадя, увеличивая утилизацию буфера и заодно задержки.

**RR — Round-Robin**

RR перебирал очереди, извлекая из них равное число пакетов. Подход более примитивный, чем FQ, и оттого нечестный по отношению к различным потокам. Агрессивные источники легко могли затопить полосу пакетами размером в 1500 байтов.

**WRR — Weighted Round Robin**

**DWRR — Deficit Weighted Round Robin**

Каждая очередь имеет отдельную кредитную линию в битах.

При проходе из очереди выпускается столько пакетов, на сколько хватает кредита.

Из суммы кредита вычитается размер того пакета, что в голове очереди.

Если разность больше нуля, этот пакет изымается, и проверяется следующий. Так до тех пор, пока разность не окажется меньше нуля.

Для разных очередей квант разный — чем большую полосу нужно дать, тем больше квант.

Таким образом все очереди получают гарантированную полосу, независимо от размера пакетов в ней.

**PB-DWRR — Priority-Based DWRR**

Собственно практически мейнстримом сегодня становится PB-DWRR — Priority Based Deficit Weighted Round Robin.

Это тот же DWRR, в который добавлена ещё одна очередь — приоритетная, пакеты в которой обрабатываются с более высоким приоритетом. Это не значит, что ей отдаётся бóльшая полоса, но то, что оттуда пакеты будут забираться чаще.

Существует несколько подходов к реализации PB-DWRR. В одних из них, как в PQ, любой пришедший в приоритетную очередь пакет изымается сразу. В других, обращение к ней происходит каждый раз при переходе диспетчера между очередями. В третьих и для неё вводится кредит и квант, чтобы приоритетная очередь не могла отжать всю полосу.

**Короткий итог про механизмы диспетчеризации**

Нет некоего универсального диспетчера, который все классы обрабатывал так, как они того требуют. Это всегда комбинация диспетчеров, один из которых решает задачу обеспечения задержек, джиттера и отсутствия потерь, а другой распределения полосы.

CBWFQ+LLQ или PB-WDRR или WDRR+PQ.

На реальном оборудовании можно указать какие именно очереди каким диспетчером обрабатывать.

CBWFQ, WDRR и их производные — это сегодняшние фавориты.

PQ, FQ, WFQ, RR, WRR — не скорбим и не помним (если, конечно, не готовимся к CCIE Клиппер).

**Ограничение скорости**

Существует два принципиально разных подхода к ограничению скорости: полисинг и шейпинг. Решают они одну задачу, но по-разному.

**Traffic Policing**

Полисинг ограничивает скорость путём отбрасывания лишнего трафика.

Всё, что превышает установленное значение, полисер срезает и выбрасывает.

Из-за строгости принимаемых мер это называется **Hard Policing**.

Однако есть и другие возможные действия.

Полисер обычно работает совместно с измерителем трафика. Измеритель красит пакеты, как вы помните, в зелёный, жёлтый или красный.

А на основе уже этого цвета полисер может не отбросить пакет, а поместить в другой класс. Это мягкие меры — **Soft Policing**.

Применяться может как ко входящему трафику, так и к исходящему.

Отличительная черта полисера — способность абсорбировать всплески трафика и определять пиковую допустимую скорость благодаря механизму **Token Bucket**.

То есть на самом деле не срезается всё, что выше заданного значения — разрешается немного за него выходить — пропускаются кратковременные всплески или небольшие превышения выделенной полосы.

**Traffic Shaping**

Шейпинг ограничивает скорость путём буферизации лишнего трафика.

Весь приходящий трафик проходит через буфер. Шейпер из этого буфера с постоянной скоростью изымает пакеты.

Если скорость поступления пакетов в буфер ниже выходной, они в буфере не задерживаются — пролетают насквозь.

А если скорость поступления выше выходной, они начинают скапливаться.

Выходная скорость при этом всегда одинакова.

Таким образом всплески трафика складываются в буфер и будут отправлены, когда до них дойдёт очередь.

Поэтому наряду с диспетчеризацией в очередях, шейпинг — второй инструмент, делающий вклад в совокупную задержку.

Шейпер обычно применяется к исходящему трафику.

Главное достоинство такого подхода — оптимальное использование имеющейся полосы — вместо дропа чрезмерного трафика, мы его откладываем.

Главный недостаток — непредсказуемая задержка — при заполнении буфера, пакеты будут томиться в нём долго. Поэтому не для всех типов трафика шейпинг хорошо подходит.

Shaping использует механизм **Leaky Bucket**.

Шейпер старается не отбрасывать пакеты, пока они помещаются в буфер ценою увеличения задержки.

Полисер не вносит задержек, но с большей готовностью отбрасывает пакеты.

Приложения, нечувствительные к задержкам, но для которых нежелательны потери, лучше ограничивать шейпером.

Для тех же, для которых опоздавший пакет всё равно, что потерянный, лучше его отбросить сразу — тогда полисинг.

**Механизмы Leaky Bucket и Token Bucket**

**Алгоритм Leaky bucket**

Leaky Bucket — это протекающее ведро.

Имеем ведро с отверстием заданного размера внизу. В это ведро сверху сыпятся/льются пакеты. А снизу они вытекают с постоянной битовой скоростью.

В плане реализации Leaky Bucket — это обычный буфер на основе SD-RAM.

Даже если явным образом шейпинг не настроен, при наличии всплесков, которые не проходят в интерфейс, пакеты временно складываются в буфер и передаются по мере освобождения интерфейса. Это тоже шейпинг.

Leaky Bucket используется только для шейпинга и не подходит для полисинга.

**Алгоритм Token Bucket**

Задача Token Bucket — пропускать трафик, если он укладывается в ограничение по времени и отбрасывать/красить в красный, если нет.

При этом важно разрешить всплески трафика, поскольку это нормальное явление.

И если в Leaky Bucket всплески амортизировались буфером, то Token Bucket ничего не буферизирует.

**Короткий итог по ограничению скорости**

В то время как шейпинг откладывает трафик при превышении, полисинг его отбрасывает.

Шейпинг не подходит для приложений, чувствительных к задержкам и джиттеру.

Для реализации полисинга в железе используется алгоритм Token Bucket, для шейпинга — Leaky Bucket.

Token Bucket может быть:

Single Rate — Two Color Marking. Позволяет допустимые всплески.

Single Rate — Three Color Marking (sr-TCM). Позволяет допустимые и избыточные всплески.

Two Rate — Three Color Marking (tr-TCM). Позволяет пиковую скорость и допустимые и избыточные всплески.

Полисинг может использоваться на входе и на выходе с устройства. Шейпинг преимущественно на выходе.

Для PHB CS и EF используется Single Rate Two Color Marking.

Для AF — sr-TCM или tr-TCM.