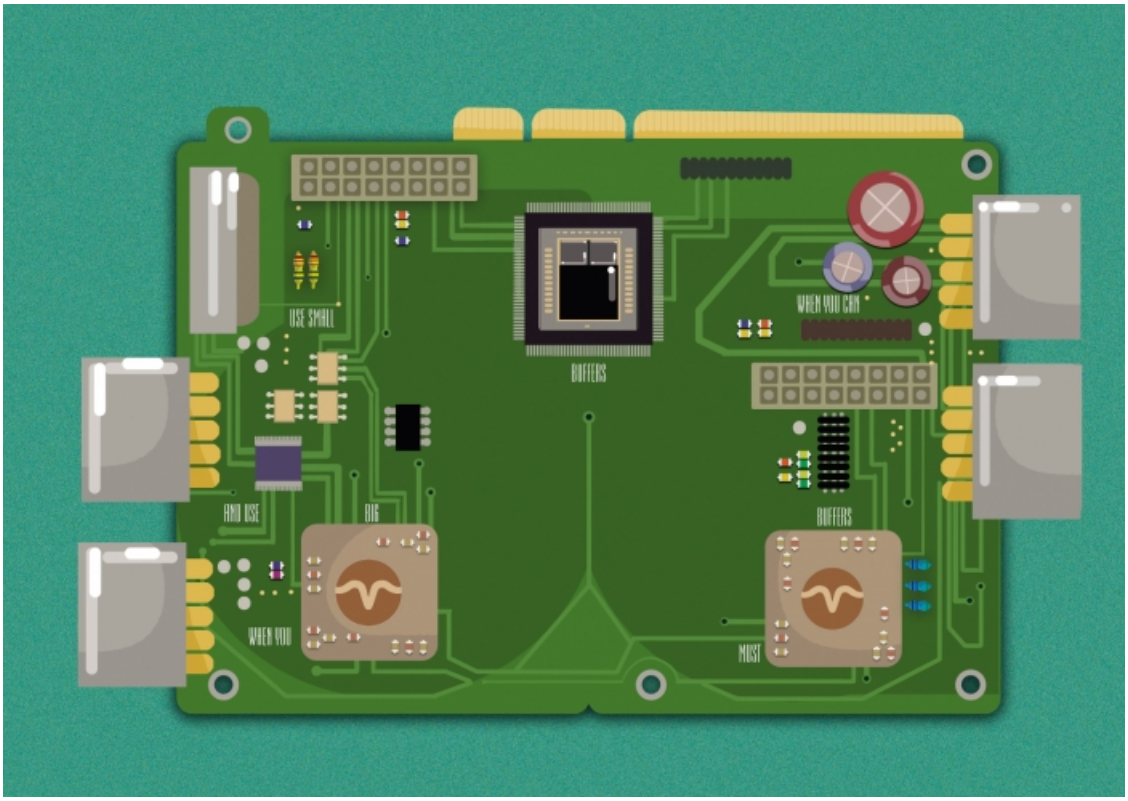
Память сети: как сохранить пакет

Эта статья едва ли для широкого круга читателей, но будет небезынтересна тем, кто хотел бы знать, сколько максимум пакет может полежать где-нибудь в сети, добираясь от точки А к точке Б, и как выбрать коммутатор так, чтобы через месяц от него не тянуло протухшими пакетами.
Для примера: на некоторых современных коммутаторах буферы по 4 гигабайта - спокойно можно четвёртый сезон Рика и Морти сохранить. Или вообще собрать кластер распределённого хранилища из коммутаторов.

Это вторая статья, копающаяся в кишочках сетевых микросхем - в первой я уже рассмотрел архитектуру сетевых устройств и чипов коммутации. А в этой взглянем на архитектуру памяти, выделение буферов и поднимем самый горячий вопрос современности - что лучше: большие или маленькие буферы?



Настоятельно рекомендую ознакомиться с 15-м выпуском СДСМ перед тем, как приступить к чтению.

Содержание

- **Терминология**
- **Назначение буферов**
- **Store-n-Forward vs Cut-Through**
- **Перегрузки: причины и места**
  - Причины перегрузок
  - Места образования перегрузок
- **Архитектура памяти**
  - Crossbar
  - Shared Buffers
    - Dedicated + Shared buffers
    - Headroom buffers
    - Admission Control
    - Alpha
  - Output queuing
  - Input queuing
  - Combined Input and Output queuing
  - Virtual Output Queueing
- **Shallow vs Deep buffers**
  - Hybrid Buffering
  - Большие буферы - это зло?
- **Lossless Low-Latency сети**

- **Полезные ссылки**

Терминология

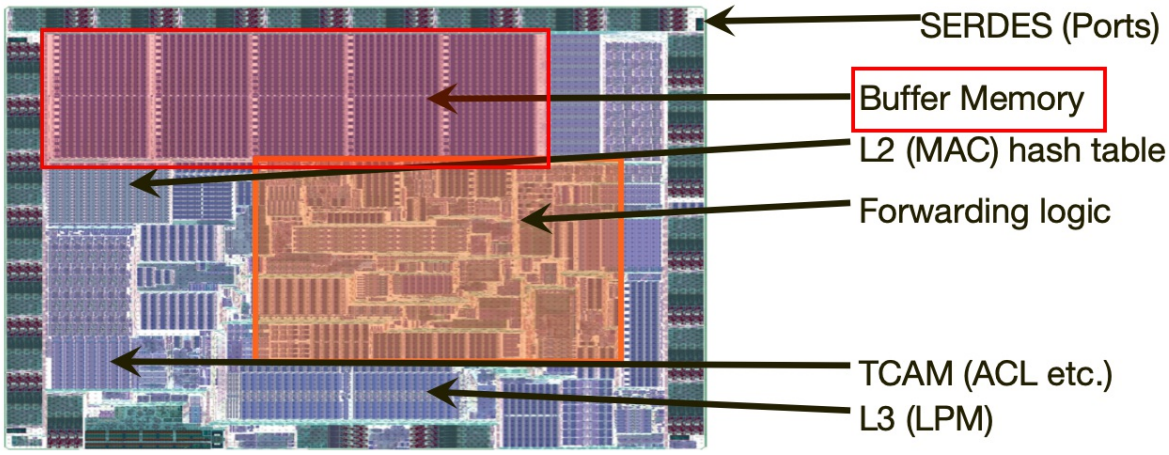
Задача этого параграфа не объять все непонятные слова, употребляемые в статье, а лишь внести некую ясность в неразбериху русско-английских терминов.

- **Чип коммутации**, **сетевой процессор**, **(Packet) Forwarding Engine**, **PFE** - микросхема, способная коммутировать пакет из входа в нужный выход с нужным набором заголовков.
- **Lookup** или **лукап** или **поиск** - поиск адресата в таблицах (FIB, LFIB, ARP Adjacencies, IPv6 ND Table и т.д.)
- **Pipeline** или **конвейер** - набор действий, которые происходят с пакетом по мере его продвижения от входа в чип до выхода из него.
- **Single-chip**, **одночиповый** - устройство, внутри которого только один чип.
- **Модульное сетевое устройство** - устройство, состоящее из шасси и отдельных карт, в него вставляющихся. В рамках статьи предполагаем, что каждая карта - это коммутатор с самостоятельным чипом коммутации, а друг с другом они связаны через фабрику коммутации.
- **Fixed**, **фиксированный**, **pizza-box** - немодульный коммутатор. Обычно внутри него нет фабрик коммутации. Часто эти термины используются как синоним Single-chip, хотя это не совсем верно - внутри может стоять два (back-to-back) чипа или даже больше.
- **Сериализация/Десериализация** - процесс перевода данных из параллельного низкоскоростного интерфейса (МГц) в последовательный высокоскоростной (ГГц) и наоборот. Например, из чипа в интерфейс или из чипа в фабрику.
- **Память** - физическая микросхема для хранения.
- **Буфер** - некий участок памяти, выделенный для хранения пакетов. Здесь и далее в производных словах, таких как "буферов", ударение на "У".
- **Очередь** - абстракция над буфером, позволяющая **виртуально** выстраивать пакеты в упорядоченную очередь. Фактически, в памяти они, конечно, хранятся "как попало".
- **OCB** - On-Chip Buffer. Память, встроенная в чип.
- **MMU** - Memory Management Unit - блок управления физической памятью.
- **TM** - Traffic Manager - блок, реализующий функции QoS, использует MMU для доступа к памяти.

Назначение буферов

Итак, в прошлый раз мы проследили судьбу пакета внутри чипа - какие именно микросхема предпринимает шаги, чтобы передать сигнал со входного интерфейса в выходной. Этот набор шагов называется конвейером или Pipeline"ом.
Ключевая часть процесса - это то, что чип отрезает от пакета значимую часть - а именно, первые сколько-то байтов, содержащих управляющие заголовки. Блок Parser эти заголовки разбирает, а блок Match-Action делает лукап по таблицам и определяет, что с пакетом нужно сделать (интерфейс, инкапсуляции, шейпинг, полисинг, ремаркировка и т.д.).
Все эти действия происходят только с заголовками пакетов.
А где же прохлаждаются их тела всё это время?

В сетевых чипсетах есть встроенная память (**on chip-OCB**) как раз для хранения тел. Её размер в силу физических ограничений очень мал (до 100Мб), но для большинства задач - это разумный компромисс.
Лишь в редких сценариях этого не хватает и добавляется объёмная внешняя память - **off-chip**.



Таким образом, на входе в чип парсер отделяет заголовки от тел. Первые отдаёт на анализ в блок Match-Action, вторые - складывает в буфер.
На выходе новые заголовки пришивают к извлечённым обратно телам и отправляют на выход.

Будучи неоднократно обвинённым при рецензировании этой статьи, чувствую необходимость ещё раз повторить это: внутри чипа коммутации заголовки отделяются от тела пакета. В то время, как изначальные заголовки анализируются, помогают сделать лукап, уничтожаются, формируются новые, тело находится в одном месте физической памяти, не перемещаясь. Даже в тот момент, когда Traffic Manager выстраивает пакеты в очередь согласно их приоритетам, производит их диспетчеризацию и Congestion Avoidance, фактически он работает со внутренними временными заголовками, не двигая пакеты по памяти.

Управляет доступом к физической памяти **MMU - Memory Management Unit**. Он довольно похож на MMU в компьютерах (по сути является им). Программа обращается к указателю, чтобы извлечь данные из памяти, MMU транслирует это в реальный адрес ячейки и возвращает данные.

MMU занимается размещением пакетов в буферах, их извлечением или отбрасыванием. Он же контролирует разделение памяти на области (dedicated, shared, headroom, voq) и их загрузку.

За более верхнеуровневое управление очередями и перегрузками отвечает блок **TM - Traffic Manager**.

Есть два подхода к размещению тел в буферах: **Store-and-Forward** и **Cut-Through**.

Store-and-Forward vs Cut-Through

SF - Store and Forward предполагает, что чип сначала получает полностью пакет, сохраняет в буфере, а уже потом занимается вивисекцией.
После анализа заголовков, пакет помещается в правильную выходную очередь.
Исторически, это первый метод коммутации. Его преимущество в том, что он может, получив пакет целиком, проверить контрольную сумму и выбросить побитые.

CT - Cut-Through, напротив, сразу после получения первых нескольких десятков байтов пакета, позволяющих принять решение о его судьбе, отправляет его в выходную очередь, не дожидаясь его полной доставки.
Это позволяет сэкономить до нескольких микросекунд на обработке пакета внутри коробки, ценою, однако, отсутствия проверки целостности - ведь FCS-то в конце. Эта задача перекладывается на протоколы более высокого уровня (или на следующий хоп).
Такой режим используется для приложений, требующих ультра-коротких задержек.

> К слову, на сегодняшний день для коммутаторов 2-го уровня время обработки в пределах устройства порядка 200нс, 3-го - 250нс. Для коммутаторов 1-го уровня (фактически патч-панель) это время - около 5 нс.

Многие производители сегодня по умолчанию устанавливают режим Cut-Through, поскольку ошибки на Ethernet сегодня явление сравнительно нечастое, а приложение обычно само может обнаружить проблему и запросить переотправку (или не запрашивать, кстати).

Дальнейшее чтение:

- Весьма глубокий подкоп под режимы коммутации от циски: Cisco Nexus 5000 Series Switches

Перегрузки: причины и места

Буфер нужен на сетевом устройстве не только для того, чтобы похранить тело пакета, пока его заголовки перевариваются в кишках чипа.
Как минимум нужно сгладить поток пакетов до скорости выходного интерфейса. Грубо говоря, пока один пакет сериализуется для передачи в интерфейс, второму нужно где-то подождать.
Для этой задачи обычно хватит совсем небольшой FIFO-очереди.
Другая важнейшая сетевая задача - перегрузки (congestion). Если в один интерфейс одновременно сваливается много пакетов, их тоже нужно сохранить.

Причём перегрузки - это наша повседневная легитимная реальность, а не что-то крайне редкое, что нужно перетерпеть и станет попроще.
Во времена коммутации каналов такой проблемы не стояло - для любой общающейся пары всегда было зарезервировано строго необходимое количество ресурсов.
Сегодня совершенно законно на порт может прийти больше трафика, чем тот готов сиюминутно пропустить.

Причины перегрузок

Самая простая - из высокоскоростного интерфейса трафик должен слиться в менее скоростной - из **10G в 1**, например.
Другая причина, очень распространённая в сетях крупных ДЦ, особенно в тех, где развёрнуты кластеры Map-Reduce - это **Incast**. Это ситуация, в которой одна машина отправляет запрос на десятки/сотни/тысячи, а те все разом начинают отвечать, и пакеты стопорятся на узком интерфейсе в сторону машины-инициатора.
Более общий случай - трафик с нескольких входящих портов должен влиться в один исходящий - **Backpressure**. Прочие типы всплесков трафика, которые ещё называют бёрстовыми или просто бёрстами (**Bursts**).

Поэтому однозначно нужно побольше памяти для буферизации. Но фактически это место, где пакеты обрастают задержкой. Так на заре 1G буферизация вызывала массу головной боли у трейдеров, чьи приложения получали свой бесценный трафик с задержкой и джиттером.

И поэтому тут уже недостаточно везде FIFO. Это задача, во благо которой трудится QoS.

Если случилась конгестия, то нужно иметь возможность требовательный к задержкам трафик, пропустить первым, чувствительный к потерям - не дропнуть, а наименее ценным пожертвовать. Ещё нужно уметь пополисить и пошейпить.

Но в каком месте располагать эту память и где реализовывать QoS?

Места возникновения перегрузок

Их по большому счёту четыре:

1. **на входном чипе** - если со стороны интерфейсов на него поступает больше, чем он способен обработать;
2. **на фабрике коммутации** (если коробка модульная) - если линейные карты пытаются отправить на фабрику больше, чем она способна обработать;
3. **на выходной линейной карте**, если фабрика пытается передать на линейную карту больше, чем её чип способен обработать;
4. **на выходном интерфейсе** - если чип шлёт в интерфейс больше, чем тот способен сериализовать.

   [Память сети: как сохранить пакет]

Но никто не хочет бороться с перегрузками в четырёх местах.
Поэтому обычно:

А) Чип делают такой производительности, чтобы он смог обработать весь трафик, даже если тот начал одновременно поступать со всех портов на этой линейной карте. Поэтому для устройства со 128 портами 100Гб/с используется чип с производительностью 12,8Тб/с.
Очевидно, бывают и исключения. Тогда или имеем непредсказуемые потери, или (чаще)

невозможность использовать часть портов.
Б) Фабрику также делают без переподписки, чтобы она могла провернуть весь трафик, который пытаются в неё передать все линейные карты, даже если они делают это одновременно на полной скорости. Таким образом, не нужно буферизировать трафик и перед отправкой на фабрику.
В) Управление перегрузками на выходном чипе и выходном интерфейсе сводят в одно место.

На самом деле фабрика без переподписки (или неблокируемая) - это та ещё спекуляция, к которой нередко прибегают маркетологи.
Для некоторых сценариев, например, CIOQ, даже со speedup фабрики в пару раз от необходимого, есть строгие результаты, показывающие, при каких условиях она будет неблокируемой.
Можно почитать у достопочтенных выпускников MIT и Стэнфорда: On the speedup required for combined input and output queued switching.

Архитектура буферов

И вот тут на сцену выходит TM - Traffic Manager, который реализует функции QoS (и некоторые другие). Он может быть частью чипа коммутации, а может быть отдельной микросхемой - для нас сейчас важно то, что он заправляет буферами.

Буфер - это, с некоторыми оговорками, обычная память, используемая в компьютерах. В ней в определённой ячейке хранится пакет, который чип может извлечь, обратившись по адресу.

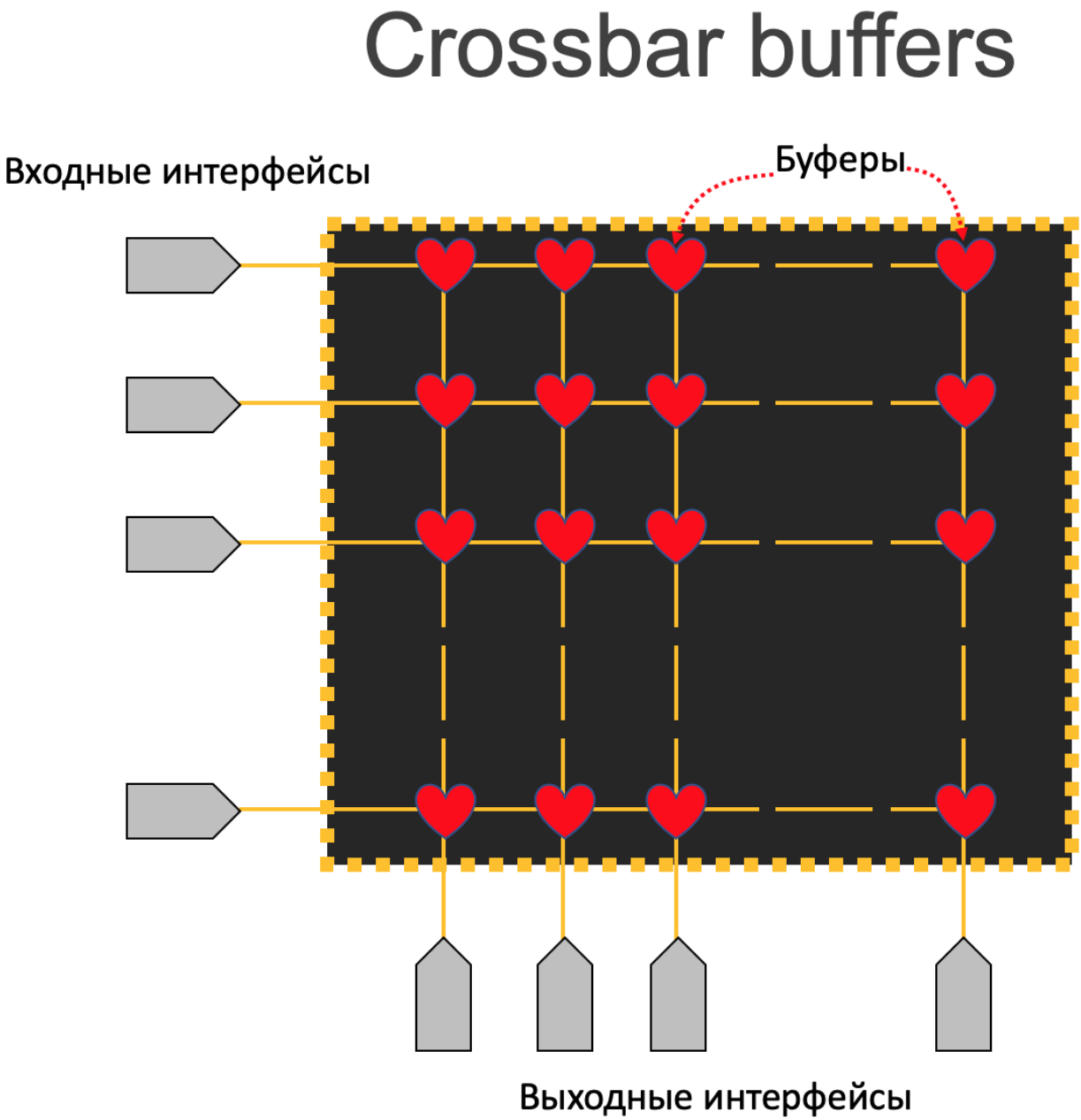
Любой сетевой ASIC или NP обладает некоторым объёмом встроенной (on-chip) памяти (порядка десятков МБ).
Так называемые Deep-Buffer свитчи имеют ещё внешнюю (off-chip) память, исчисляемую уже гигабайтами.
И той и другой управляет модуль чипа - MMU.

В целом, для нас пока местонахождение не имеет значения - взглянем на это попозже. Важно то, как имеющейся памятью чип распоряжается, а именно, где и какие очереди он создаёт и какие AQM использует.
И тут практикуют:

- Crossbar.
- Shared Buffer.
- Output Queuing.
- Input Queueing.
- Combined Input and Output Queueing.
- Virtual Output Queueing.

Crossbar

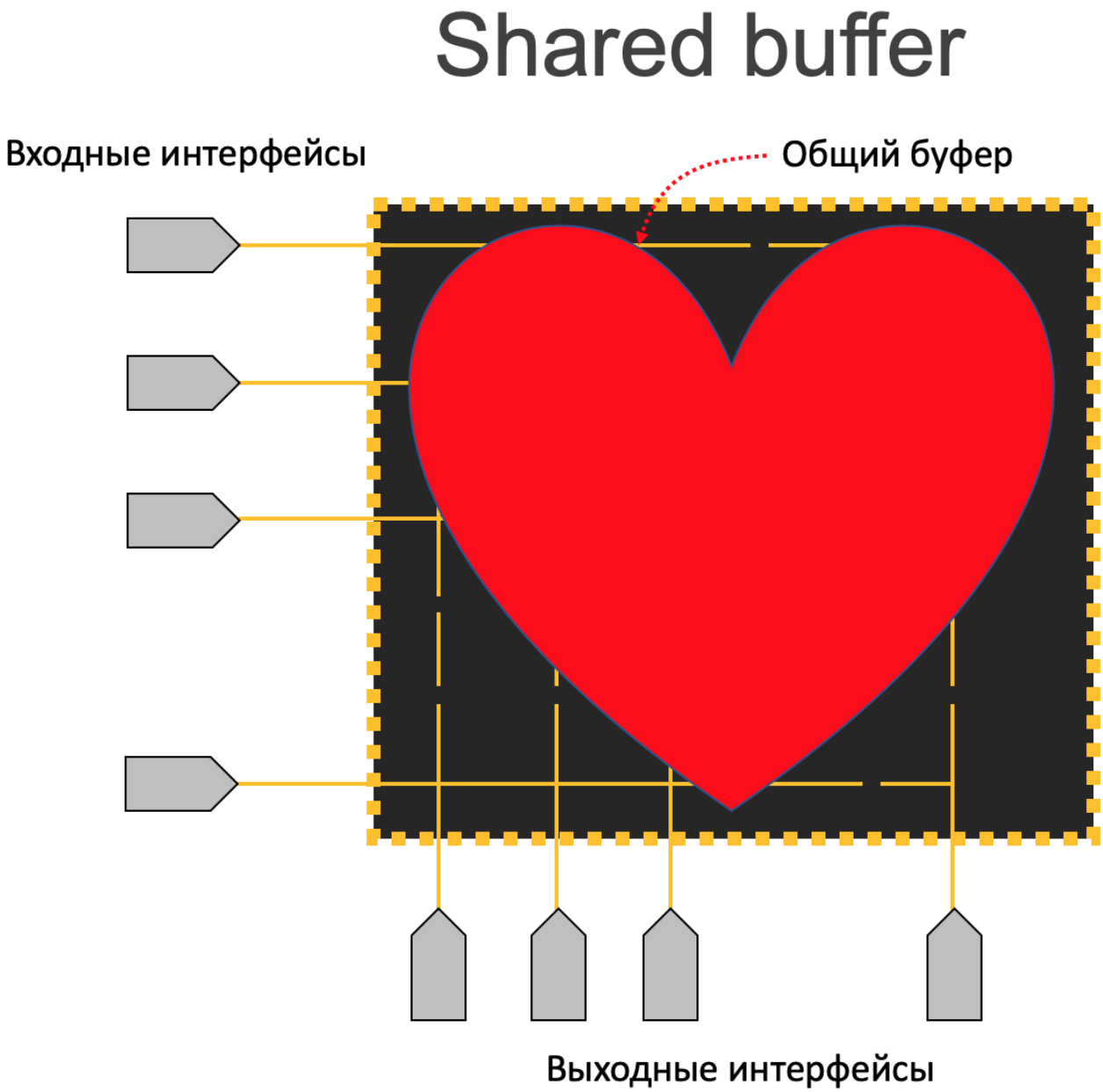
Идея в том, чтобы для каждой пары (входной интерфейс - выходной интерфейс) выделить аппаратный буфер.



Это, скорее, умозрительный эксперимент, потому что в плане сложности, стоимости реализации и эффективности это проигрышный вариант.

Shared Buffer

По числу существующих в мире коробок этот вариант, однозначно, на первом месте.



Используется Shared Buffer на немодульных устройствах без фабрики коммутации, в которых установлен один чип (обычно, но может быть больше).

Аппаратно - это память (обычно SRAM), встроенная прямо в чип - она так и называется **on-chip** (OCB).
Много туда не засунешь, поэтому объём до 100 МБ.
Зачастую, это единственная память, которая в одночиповых устройствах используется для буферизации.
Пусть, однако, эта кажущаяся простота не вводит вас в заблуждение - для того, чтобы в десятки мегабайтов поместить трафик сотни портов 100Гб/с, да ещё и обеспечить отсутствие потерь, за ними должны скрываться годы разработок и нетривиальная архитектура.
А так оно и есть - я чуть ниже неглубоко вас окуну.

Итак, есть соблазн эту память взять и просто равномерно разделить между всеми портами. Такой статический дизайн имеет право на жизнь, но сводит на нет возможность динамически абсорбировать всплески трафика.

Гораздо более привлекательным выглядит следующий вариант:

Dedicated + Shared

Из доступной памяти каждому порту выделяется определённая небольшая часть - это **Dedicated Buffer**. За каждым портом кусочек памяти законодательно закреплён и не может быть использован другими портами. То есть, при любых обстоятельствах у порта будет свой защищённый кусочек.
Минимальный размер Dedicated Buffer где-то настраивается, где-то нет. Но лучше без основательного понимания в дефолты не лезть.

Доля каждого порта в абсолютных цифрах очень маленькая - порядка единиц кБ.
Гарантируемый минимум выделяется для хранения как входящих пакетов, так и выходящих.

Остальная часть памяти как раз общая - **Shared Buffer** - может быть использована любым портом по мере необходимости. Из неё динамически выделяются куски для тех интерфейсов, которые испытывают перегрузку.
Например, если чип пытается на один из интерфейсов передать больше трафика, чем тот способен отправлять в единицу времени, то эти пакеты сначала заполняют выделенный для этого порта буфер, а когда он заканчивается, автоматически начинают складываться в динамически выделенный буфер из общей памяти. Как только все пакеты обработаны, память освобождается.
Под общий буфер может быть отдано 100% той памяти, что осталась после вычитания из неё выделенных для портов кусочков (Dedicated). Но она так же может быть перераспределена - за счёт общего буфера можно увеличить выделенные. Так, если выделить 80% под Shared, то оставшиеся 20% равномерно распределятся по Dedicated.

Наличие Shared Buffer"а решает огромную проблему, позволяя сглаживать всплески трафика, когда перегрузку испытывает один или несколько интерфейсов.

Однако, вместе с тем, за общую память начинаются соревноваться разные порты одновременно. И серьёзная перегрузка на одном порту может вызвать потери на другом, которому нужно было всего лишь несколько килобайтов общей памяти, чтобы не дропнуть пакет.
Одним из способов облегчить эту ситуацию является увеличение выделенных буферов за счёт уменьшения общего. Но это всегда зона компромиссных решений - сокращая размер общей памяти, мы уменьшаем и объёмы всплесков, которые чип может сгладить.
Кроме того, Lossless трафик требует к себе ещё более щепетильного отношения.

Поэтому, зачастую, помимо Dedicated и Shared буферов, резервируют ещё **Headroom buffers**.

Headroom buffers

Это последний способ сохранить пакеты, когда даже общий буфер уже забит. Естественно, он тоже отрезается от общей памяти, поэтому на первый взгляд выглядит не очень логичным откусить от общей памяти кусок, назвать его по-другому и сказать, мол, мы всё оптимизировали.
На самом деле, Headroom-буферы решают довольно специфическую задачу - помочь lossless приложениям с **PFC** - Priority-based Flow Control.

PFC - это механизм Ethernet Pause, который умеет притормаживать не всю отправку, а только по конкретным приоритетам Ethernet CoS.
Например, два приложения на отправителе: RoCE и репликация БД. Первое - чувствительная к задержкам и потерям вещь, второе - массивные данные.

Коммутатор, заметив заполнение общего буфера, отправляет Pause для более низкого приоритета, тем самым притормаживая репликацию, но не RoCE.

Задача буфера Headroom здесь в том, чтобы сохранить те пакеты приоритетной очереди, что сейчас в кабеле, пока Pause летит к отправителю с просьбой притормозить. То есть пакеты репликации начнут дропаться, когда заполнится общий буфер, а пакеты RoCE будут складываться в Headroom.

> Помимо lossless headroom бывает и headroom для обычного трафика, чтобы помочь сохранить более приоритетный. Но это на домашнее задание.

> [Память сети: как сохранить пакет]

При наступлении перегрузки буферы будут задействованы в следующем порядке.
Для входящего best-effort трафика:

1. Dedicated buffers.
2. Shared buffers.

Для входящего lossless трафика:

1. Dedicated buffers.
2. Shared buffers.
3. Lossless headroom buffers.

Для всего исходящего трафика:

1. Dedicated buffers.
2. Shared buffers.

Разумеется, описанное выше лишь частный пример, и от вендора к вендору ситуация может различаться (разительно).

Например, бродкомовские чипы (как минимум, Trident и Tomahawk) имеют внутреннее разделение памяти по группам портов. Общая память делится на порт-группы по 4-8 портов, которые имеют свой собственный кусочек общего буфера. Порты из одной группы, соответственно буферизируют пакеты только в своём кусочке памяти и не могут занимать другие. Это тоже один из способов снизить влияние перегруженных портов друг на друга. Такой подход иногда называют **Segregated Buffer**.

Admission Control

Admission Control - входной контроль - механизм, который следит за тем, можно ли пакет записывать в буфер. Он не является специфичным для Shared-буферов, просто в рамках статьи - это лучшее место, чтобы о нём рассказать.

Формально Admission Control делится на Ingress и Egress.
Задача **Ingress Admission Control** - во-первых, вообще убедиться, что в буфере есть место, а, во-вторых, обеспечить справедливое использование памяти.
Это означает, что у каждого порта и очереди всегда должен быть гарантированный минимальный буфер. А ещё несколько входных портов не оккупируют целиком весь буфер, записывая в него всё новые и новые пакеты.

Задача **Egress Admission Control** - помочь чипу абсорбировать всплески, не допустив того, чтобы один или несколько выходных портов забили целиком весь буфер, получая всё новые и новые пакеты с кучи входных портов.

В случае Shared Buffer оба механизма срабатывают в момент первичного помещения пакета в буфер. То есть, никакой двойной буферизации и проверки не происходит.

Как именно понять, сколько буфера занято конкретным портом/очередью и главное, сколько ещё можно ему выдать? Это может быть статический порог, одинаковый для всех портов, а может быть и динамически меняющийся, регулируемый параметром **Alpha**.

Alpha

Итак, почти во всех современных чипах память распределяется динамически на основе информации о том, сколько общей памяти вообще свободно и сколько ещё можно выделить для данного порта/очереди.

На самом деле, минимальной единицей аккаунтинга является не порт/очередь, а регион (в терминологии Мелланокс). Регион - это кортеж (входной порт, Priority Group на входном порту, выходной порт, Traffic Class на выходном порту).

Каждому региону назначается динамический порог - сколько памяти он может под себя "подмять". При его превышении, очевидно, пакеты начинают дропаться, чтобы не влиять на другие регионы. Этот порог вычисляется по формуле, множителями которой являются объём свободной на данный момент памяти и параметр **alpha**, специфичный для региона и настраиваемый:
Threshold [Bytes] = alpha * free_buffer [Bytes]

Его значение варьируется от 1/128 до примерно 8 с шагом x2. Чем больше эта цифра, тем больший объём свободной памяти доступен региону.
Например, если на коммутаторе 32 региона, то:

- при alpha=1/64 каждому региону будет доступна 1/64 часть свободной памяти, и даже при максимальной утилизации они все смогут использовать только половину буфера.
- при alpha=1/32 вся память равномерно распределится между регионами, ни один из них не сможет влиять на другие, а при полной утилизации 100% памяти будет занято.
- при alpha=1/16 каждый регион может претендовать на больший объём памяти. И если все регионы разом начнут потреблять место, то им всем не хватит, потому что памяти потребовалось бы 200%. То есть это своего рода переподписка, позволяющая сглаживать всплески.

Предполагаем тут, что значение alpha одинаково для всех регионов, хотя оно может быть настроено отдельно для каждого.

При получении каждого пакета, механизм Admission Control вычисляет актуальный порог для региона, которому принадлежит пакет. Если порог меньше размера пакета, тот отбрасывается. Если же больше, то он помещается в буфер и уже не будет отброшен никогда, даже если регион исчерпал все лимиты. Объём свободной памяти уменьшается на размер пакета.
Это происходит для каждого приходящего на чип пакета.

Написанное выше об Admission Control и Alpha может быть справедливо не только для Shared Buffers, но и для других архитектур, например, VoQ.

Дальнейшее чтиво:

- Если в жизни не хватает страданий: Design and Implementation of a Shared Memory Switch Fabric.
- Understanding the Alpha Parameter in the Buffer Configuration of Mellanox Spectrum Switches.
- Programming Guide"ы коммерческих микросхем (NDA кровью, помним, да?).

Crossbar и Shared Buffer - это архитектуры, которые могут использоваться для устройств фиксированной конфигурации, но не подходят для модульных.
Взглянем же теперь на них.

Дело в том, что они состоят из нескольких линейных карт, каждая из которых несёт как минимум один самостоятельный чип коммутации.
И этот чип, будь то ASIC, NP или даже CPU не может в своей внутренней памяти динамически выделять буферы для тысяч очередей выходных интерфейсов - кишка тонка.

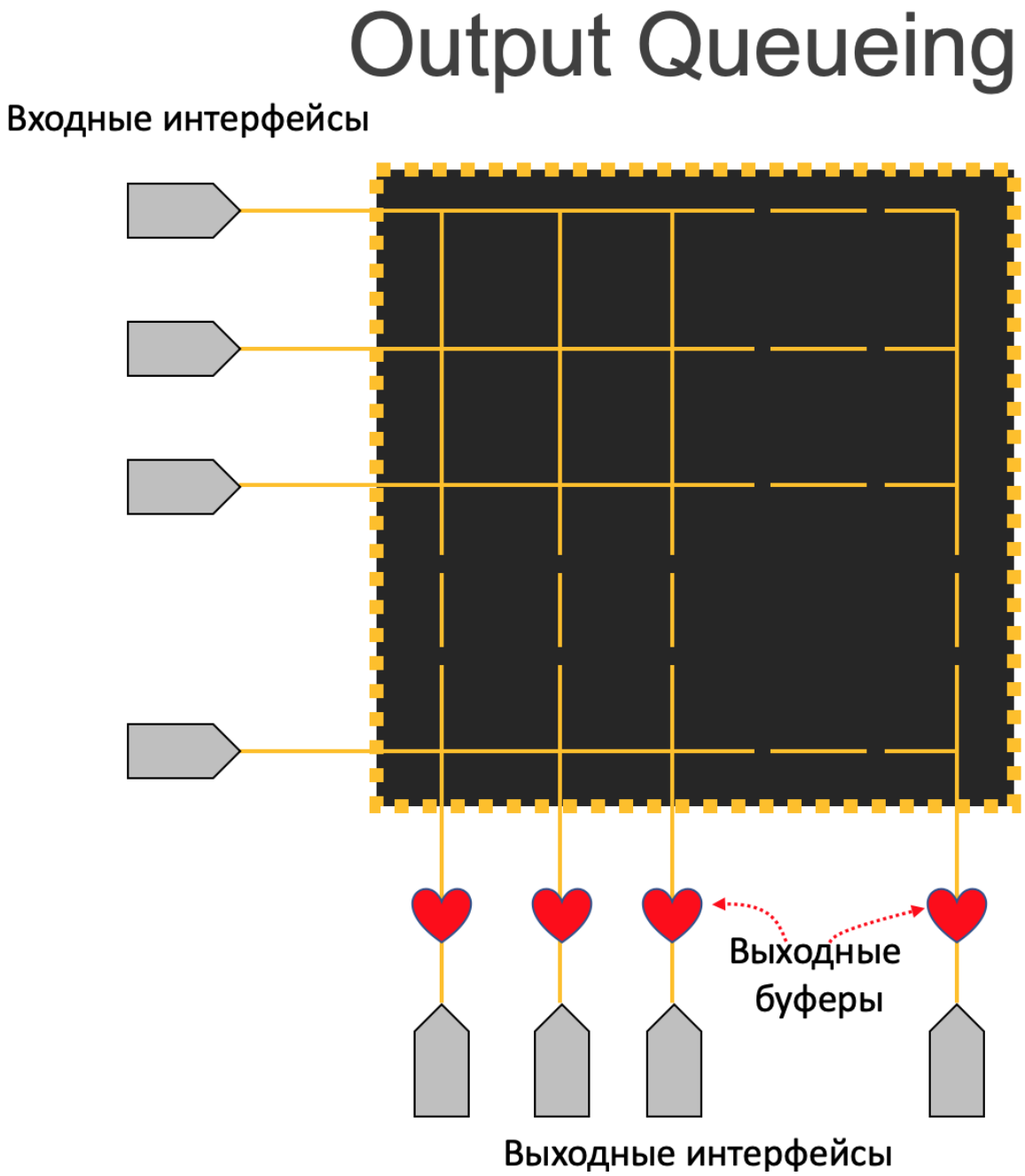
> [Память сети: как сохранить пакет]

Далее поговорим про архитектуры памяти для модульных шасси:

- Output Queueing.
- Input Queueing.
- Combined Input and Output Queueing.
- Virtual Output Queueing.

Output Queueing

Наиболее логичным кажется буферизировать пакеты как можно ближе к месту возможного затора - около выходных интерфейсов. Кому как не выходному чипу знать о здоровье своих подопечных интерфейсов, обслуживать по несколько QoS-очередей для каждого и бороться с перегрузками?



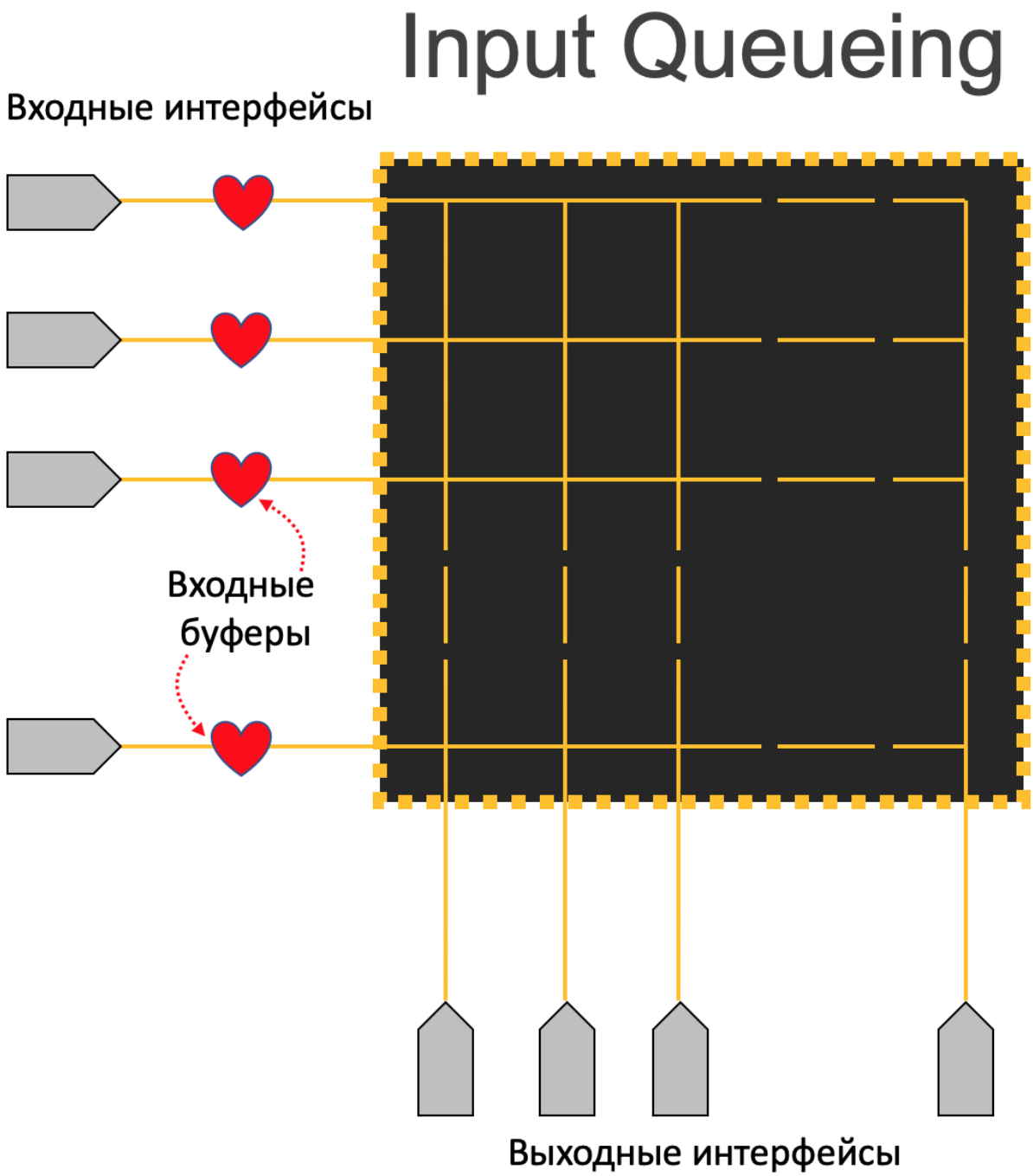
И это правда так.
Но есть одна фундаментальная проблема - в случае перегрузок пакеты будут приходить на Egress PFE, чтобы умирать. Они проделают весь огромный путь от входного интерфейса через фабрику коммутации до выходного буфера через фабрику для того, чтобы узнать, что мест нет и быть печально дропнутыми.
Это бессмысленная и бесполезная утилизация полосы пропускания фабрики.

> [Память сети: как сохранить пакет]

И вот уже вырисовывается следующая логичная мысль - выбросить пакет нужно как можно раньше. Как было бы здорово, если бы мы могли это сделать на входной плате.

Input Queuing

Более удачным вариантом оказывается буферизировать пакеты на входной плате после лукапа, когда уже становится понятно, куда пакет слать. Если выходной интерфейс заведомо занят, то и смысла гнать камикадзе на фабрику нет.

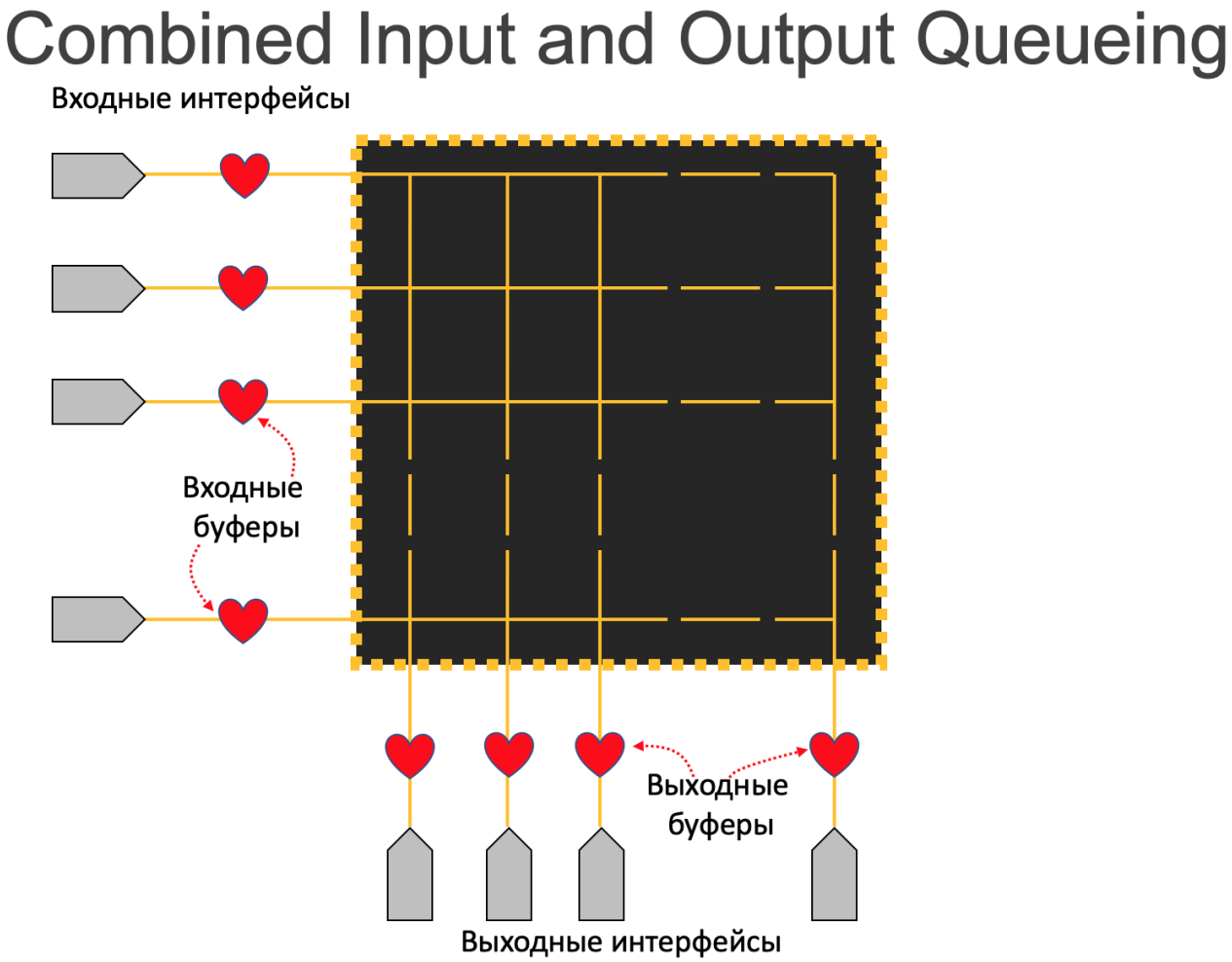


Постойте! Как же входной чип узнает, что выходной интерфейс не занят?

С точки зрения Data Plane никакой обратной связи, от выходного чипа входному, очевидно, нет. Распространение между ними информации, необходимой для лукапа (некстхопы, интерфейсы, заголовки) производится средствами медленного Control Plane - тоже не подойдёт.
Так вот, для сигнализации такой информации между линейными платами появляется арбитр. У разных вендоров он может быть реализован по-разному, но суть его в следующем - входной чип регулярно запрашивает у выходного разрешение на отправку нового блока данных. И пока он его не получит - держит пакеты в своём буфере, не отправляя их в фабрику.
Соответственно, выходной чип, получив такой запрос, смотрит на утилизацию выходного интерфейса и решает, готов ли он принять пакет. Если да - отправляет разрешение (**Grant**).
Это на первый взгляд контринтуитивное поведение - каковы же накладные расходы на такой арбитраж, насколько это увеличивает задержки, если на отправку пакета данных нужно дождаться RTT в пределах коробки - пока запрос улетит на выходной чип, пока тот обработает, пока ответ вернётся назад.
Тут для меня начинается область магического искусства, но вендоры эту революцию успели совершить и есть масса платформ, на которых арбитр прекрасно со своей задачей справляется. Хотя обычно он применяется не для Input Queueing в описанном виде.
Дело в том, что эффективность Input Queueing не очень высокая - очень часто придётся ждать, пока интерфейс освободится. Эх, прям вспоминается старый добрый Ethernet CSMA/CD.

Combined Input and Output Queueing

Гораздо выгоднее в этом плане разрешить буферизацию и на выходе.
Тогда арбитр будет проверять не занятость интерфейса, а степень заполненности выходного буфера - вероятность, что в нём есть место, гораздо выше.



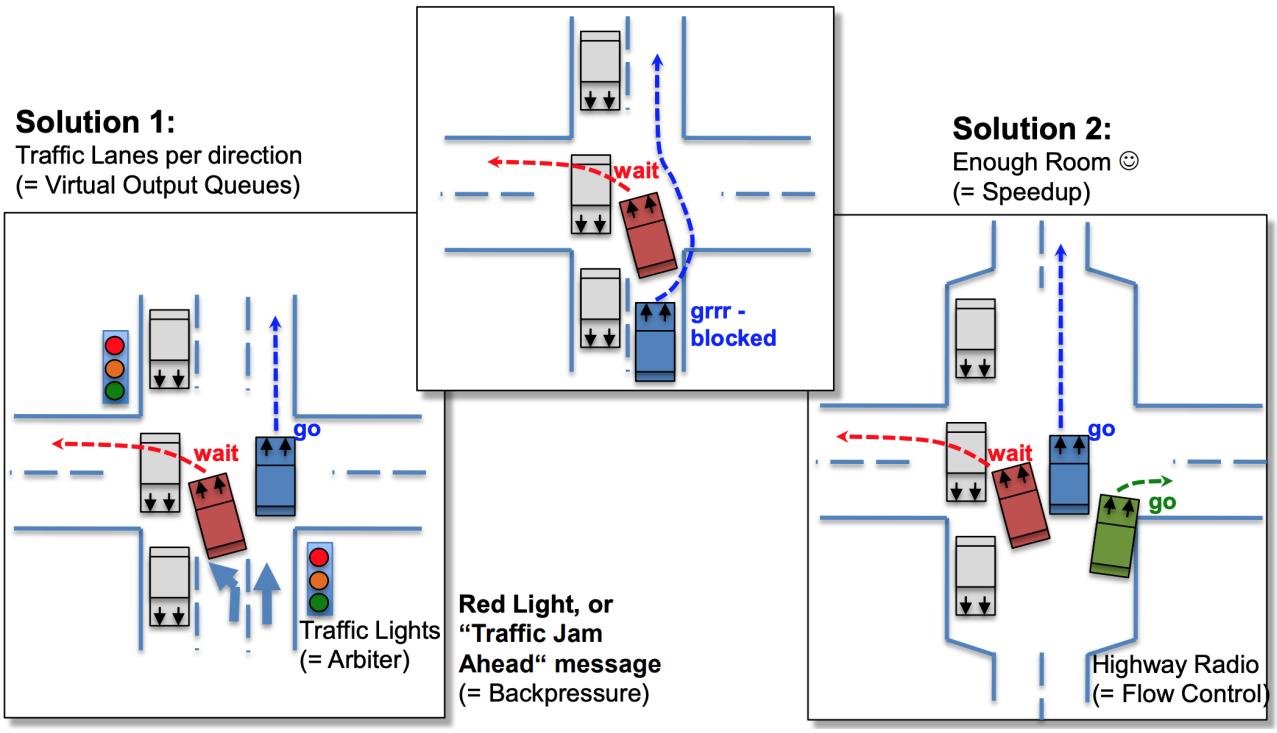
Но такие вещи не даются даром. Очевидно, это и увеличенная цена из-за необходимости реализовывать дважды буферизацию, и увеличенные задержки - даже в отсутствие заторов этот процесс не бесплатный по времени.

Кроме того, для обеспечения QoS придётся хоть какой-то минимум его функций реализовывать в двух местах, что опять же скажется на цене продукта.

Но у CIOQ (как и у IQ) есть фундаментальный недостаток, заставивший в своё время немало поломать голову лучшим умам - **Head of Line Blocking**.

Представьте себе ситуацию: однополосная дорога, перекрёсток, машине нужно повернуть налево, сквозь встречный поток. Она останавливается, и ждёт, когда появится окно для поворота. А за ней стоит 17 машин, которым нужно проехать прямо. Им не мешает встречный поток, но им мешает машина, которая хочет повернуть налево.

HoLB (Head of Line Blocking) problem



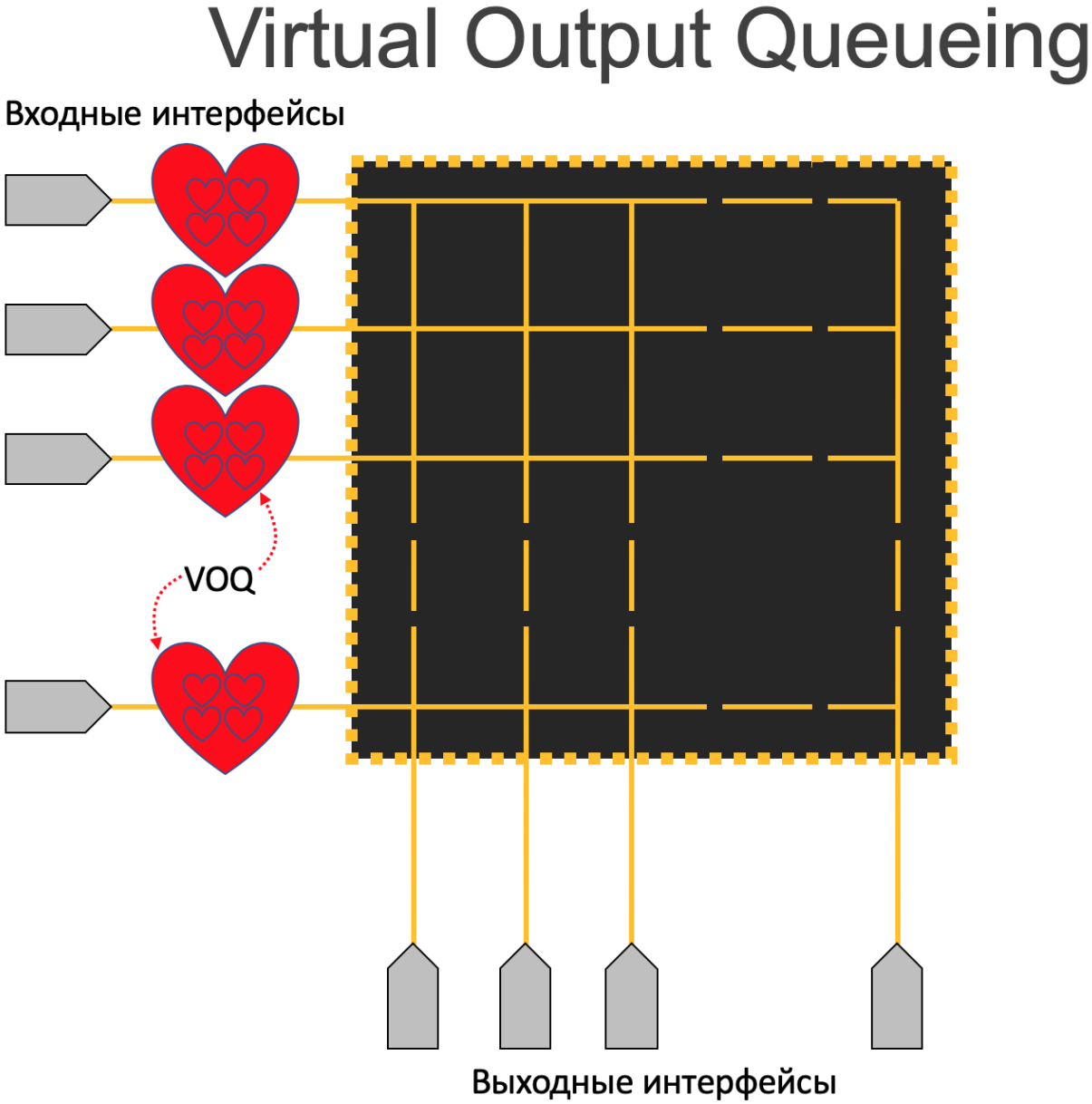
Этот избитый пример иллюстрирует ситуацию HoLB. Входной буфер - один на всех. И если всего лишь один выходной интерфейс начинает испытывать затор, он блокирует полностью очередь отправки на выходном чипе, поскольку один пакет в начале этой очереди не получает разрешение на отправку на фабрику.

Трагическая история, как в реальной жизни, так и на сетевом оборудовании.

Virtual Output Queueing

Как можно исправить эту дорожную ситуацию? Например, сделав три полосы - одна налево, другая прямо, третья направо.

Ровно то же самое сделали разработчики сетевого оборудования.
Они взяли входной буфер побольше и подробили его на множество очередей.
Для каждого выходного интерфейса они создали по 8 очередей на каждом чипе коммутации. То есть перенесли все задачи по обеспечению QoS на входной чип. На выходном же при этом остаётся самая базовая FIFO очередь, в которой никогда не будет заторов, потому что их контроль взял на себя входной чип.



Если взять, грубо говоря, коробку со 100 интерфейсами, то на каждой плате в буферах нужно будет выделить 800 очередей. Если в коробке всего 10 линейных карт, то общее число очередей на ней будет 100*8*10 = 8000.

Однако V в VOQ означает виртуальный, не потому, что они как бы выходные, но на самом деле находятся на входных платах, а потому что Output Queue для каждого выходного интерфейса распределён между всеми линейными картами. То есть, сумма десяти физических очередей для одного интерфейса на десяти чипах составляет 1 виртуальную.
Собственно, из-за распределённого характера этой виртуальной очереди от арбитра и здесь избавиться не получится - разным входным чипам всё же нужно знать, состояние выходной очереди. Поэтому, даже несмотря на то, что выходная очередь - это FIFO, выходной чип всё ещё должен давать добро на отправку трафика.

Кстати, что касается трафика, который должен вернуться в интерфейс той же карты, на которую он пришёл изначально, то здесь никаких исключений - он томится в VOQ, пока чип не даст "добро" переложить его в выходную очередь.
Причём, зачастую, такому трафику придётся даже выйти на фабрику, потому что Ingress и Egress части полностью разделены. Поэтому перед лицом перегрузок все равны.

На сегодняшний день подавляющее большинство модульных сетевых устройств используют архитектуру VOQ.

Дальнейшее чтиво:

- An Update on Router Buffering.
- What is VOQ and why you should care?
- Strategies of packet buffering inside Routers.

- Understanding CoS Virtual Output Queues (VOQs) on QFX10000 Switches.
- High Performance Switches and Routers - если у вас есть лишних 14 к₽.

Shallow vs Deep Buffers

Буфер - это такое место, где пакеты можно похранить, вкачав в них смертельную порцию задержки. Как сказали в видео Packet Pushers, буферы - это религия. Хотя, скорее всего, неортодоксальная, а возможно даже секта.

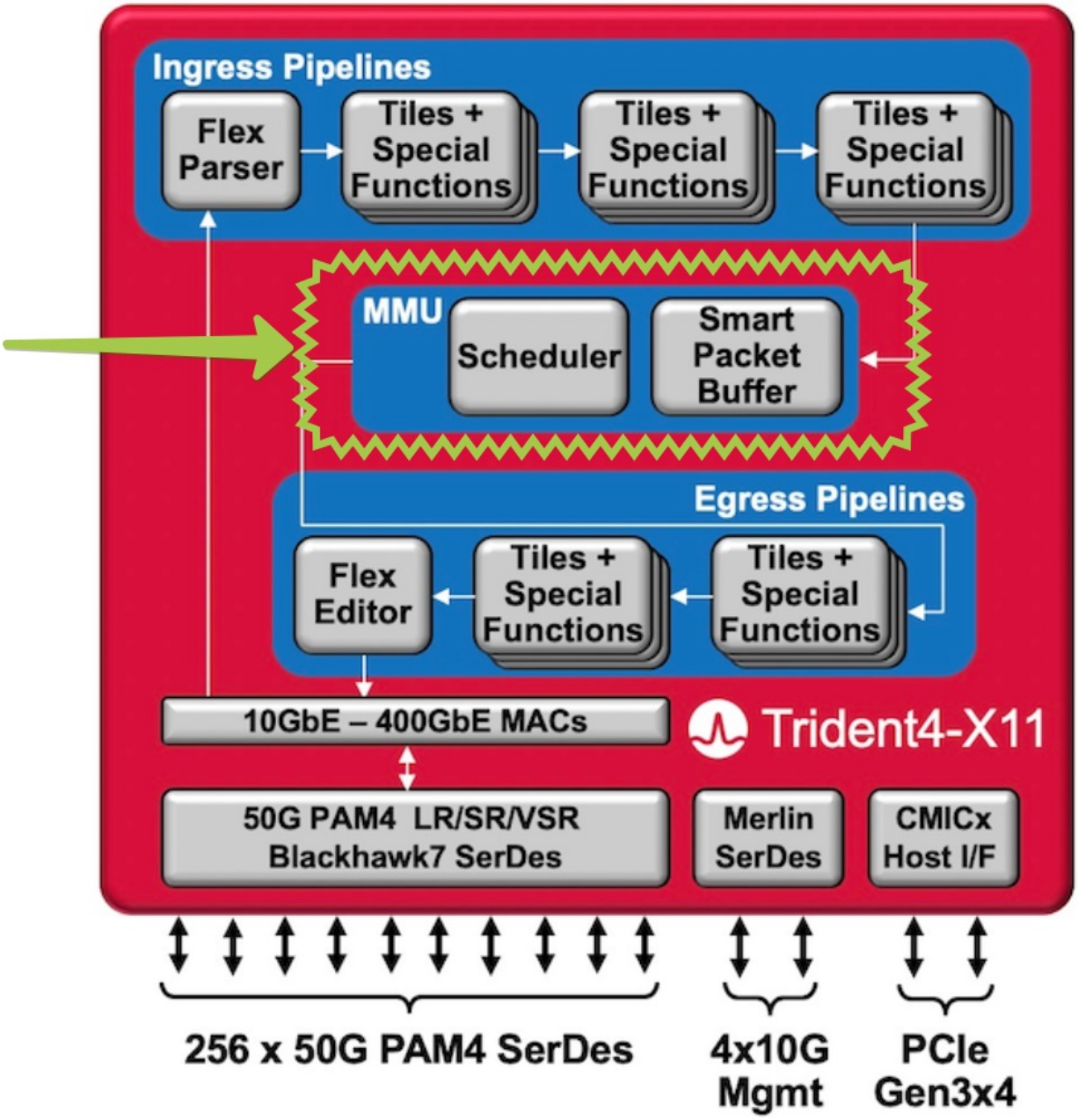
Чуть позже мы поговорим о том, что такое хорошо, а что такое плохо. А пока посмотрим на реализации.

Shallow - неглубокие - это буферы размером до 100МБ. Обычно это встроенная в кристалл **on-chip** память - **OCB** - On-Chip Buffer.
Deep - счёт уже идёт на гигабайты. Обычно **off-chip** и подключается к чипу по отдельной шине. И нет ничего посередине.

За последние лет десять производительность чипов выросла на порядки, трафика они теперь перемалывают терабиты в секунду вместо единиц гигабит. А размер памяти не то что не поспевает за этим ростом, он фактически почти стоит на месте. Давайте грубо прикинем: если для гигабитного порта буфер размером 16 мегабайт мог абсорбировать всплеск трафика длительностью примерно 100 мс, то для 100Гб/с - всего лишь 1мс. И это только один порт, фактически же плотность портов тоже растёт и максимальная комплектация для одночипового устройства сегодня - 64 порта 400Гб/с - или 25,6 Тб/с полосы пропускания.
Используя только 64 МБ буфер, такой чип сможет хранить трафик 0.000005 с или 5 мкс.

Такие буферы порой даже называют Extremely shallow buffers.

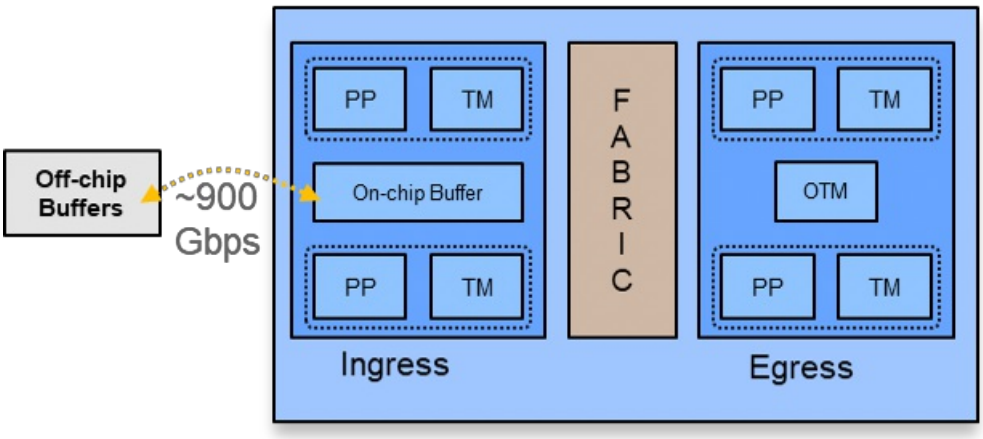
Их воистину миниатюрный объём обусловлен, в первую очередь тем, что они в прямом смысле встроены в чип. Такая память является составной частью микросхемы, и каждый дополнительный мегабайт, разумеется, будет обходиться в лишнюю тысячу долларов, больший размер и тепловыделение. Для справки: Broadcom Trident 4 содержит 21 миллиард транзисторов, изготовленных по 7нм техпроцессу на нескольких квадратных сантиметрах.
Логично вытекающим следствием, является скорость работы с этой памятью - она должна соответствовать производительности чипа.



Очевидно, что не для всех задач такие маленькие буферы подходят. В частности, модульные коробки с VOQ явно не могут позволить себе дробить 64 Мб на несколько тысяч очередей (на самом деле могут).

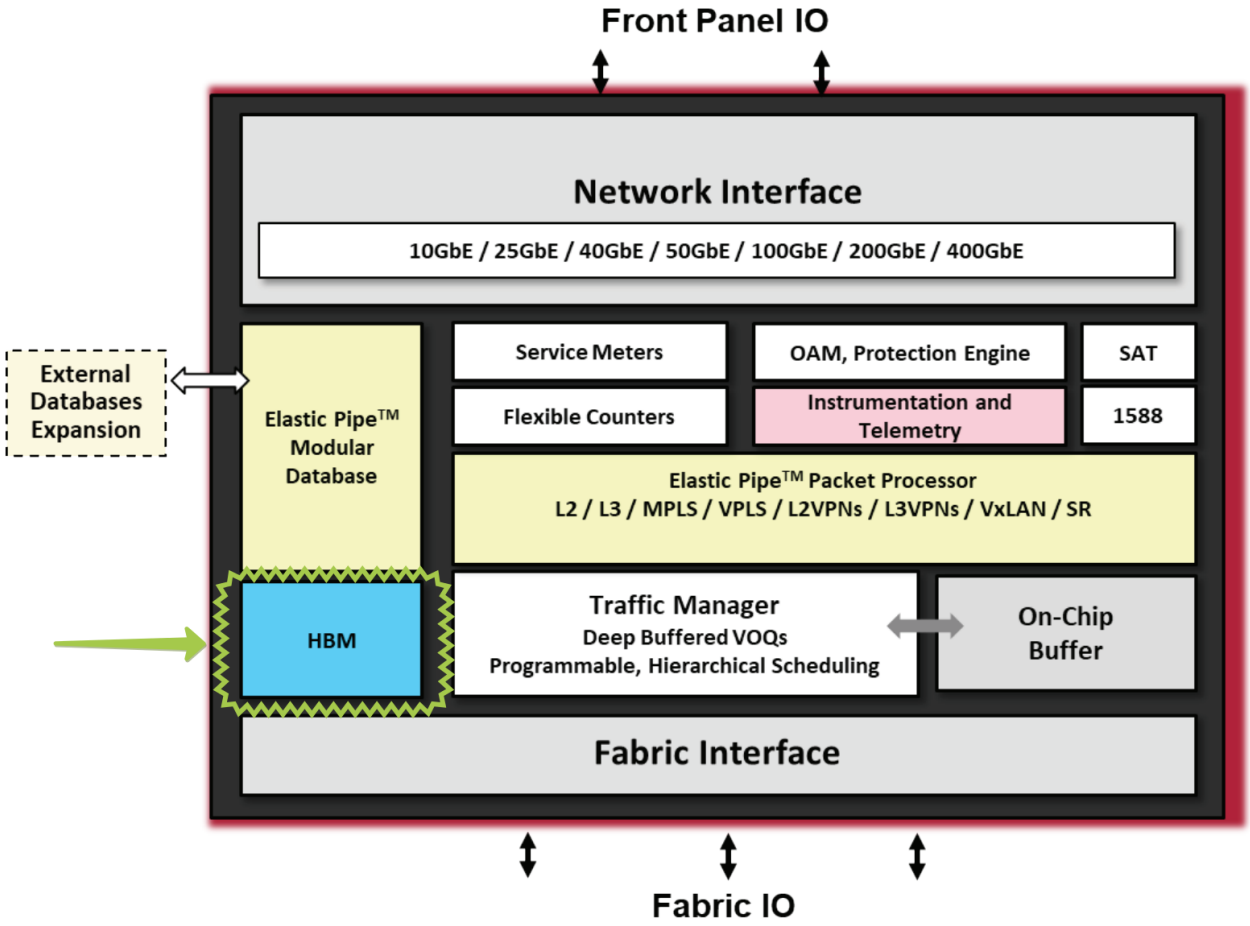
Поэтому рынок предлагает решения с большой внешней памятью (Deep Buffers), размер которой начинается от 1ГБ (обычно от 4ГБ).
Согласно этой таблице существуют коммутаторы (Arista 7280QR-C48) с фантастическими 32-гигабайтовыми буферами - это уже все сезоны Рика и Морти в неплохом качестве. Но это уже история про VOQ - всё-таки это память не одного чипа. На моём первом ПК такого объёма был жёсткий диск.

Как такая память реализована зависит уже от чипа и коробки.
Например, Broadcom **Jericho+** сгружает пакеты во внешнюю память в размере 4ГБ. Это обычная широко известная **GDDR5**, использующаяся в видеокартах.

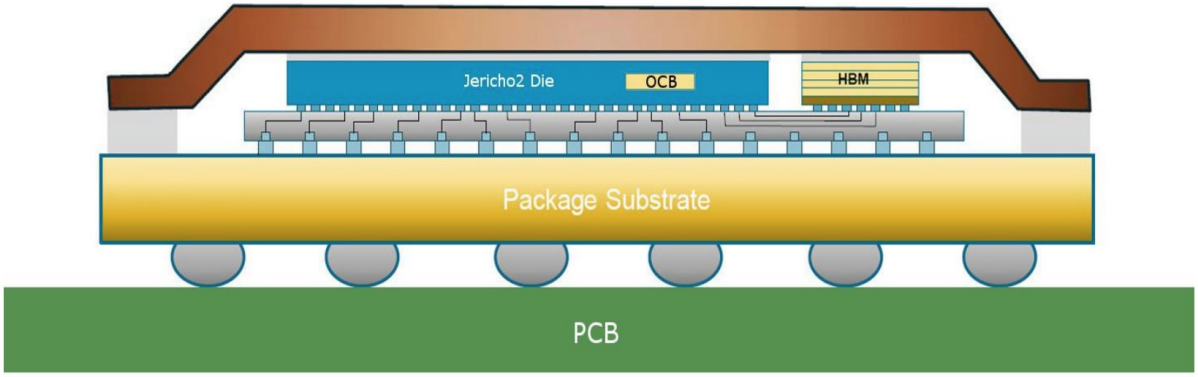


Источник.

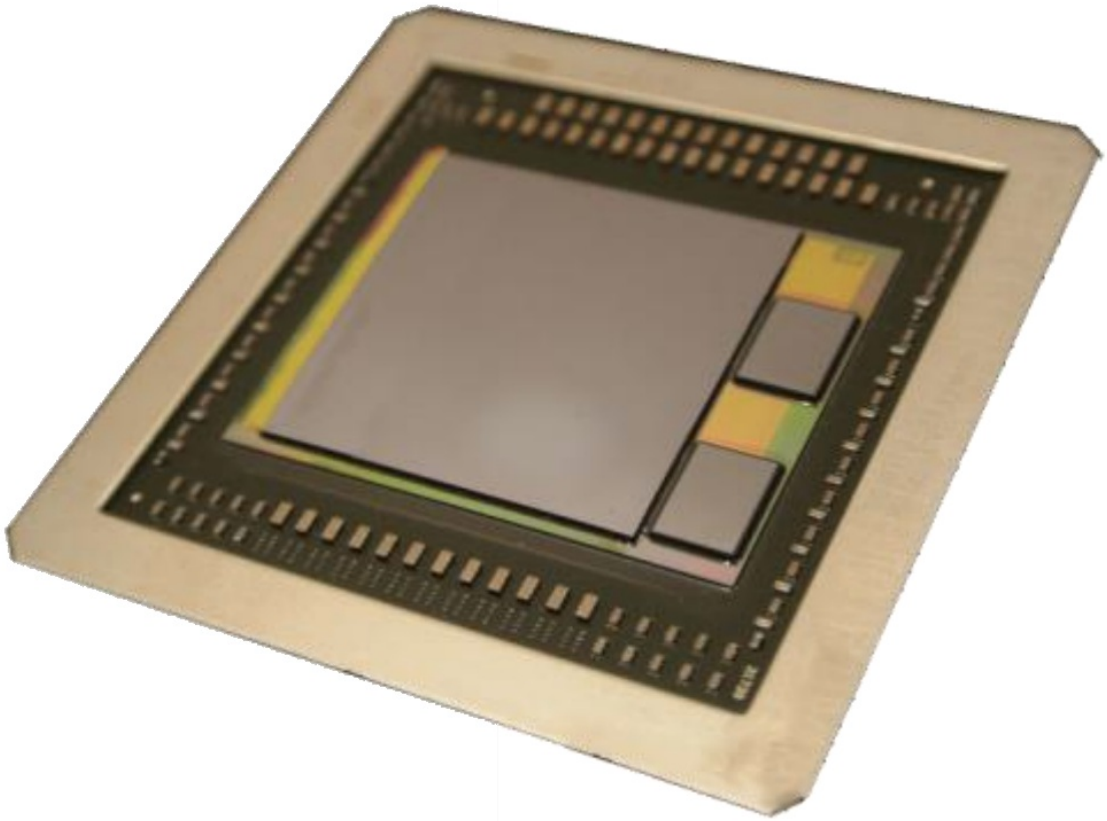
Jericho2 несёт на борту новейшую память **HBM2** - High Bandwidth Memory - размером 8ГБ.



Источник.



А вот и фото Jericho2:



Источник.

Juniper QFX10000 использует чип **Q5** собственного производства с внешней памятью - **HMC** - Hybrid Memory Cube - в размере 4ГБ. HMC - это коммерческая память производства Micron, от которой ныне отказались в пользу HBM и GDDR6.



Источник.

Фото чипа ZX EXPRESS:



Источник.

А вот так выглядит сетевой процессор Cisco с внешней памятью:

> [Память сети: как сохранить пакет]

Источник.

Перечислять можно и дальше.
Важно отметить, что внешняя память тоже не даётся бесплатно. Во-первых, цена таких решений значительно выше. Во-вторых, пропускная способность обычно ниже. И основное ограничение - канал между чипом коммутации и чипом памяти. Для производимых массово чипов, вроде GDDR5 полоса не превышает 900ГБ в режиме half-duplex. Но это чип, явно не заточенный под задачи сетевых сервисов.

Кастомный джуниперовский HMC обещает 1,25 Тб/с в обоих направлениях.

Если верить вики, то HBM 2-го поколения, используемый в последнем чипе Broadcom Jericho2, выдаёт порядка 2Тб/с.

Но это всё ещё далеко от реальной производительности сетевого ASIC. Фактически, шины до этой внешней памяти является узким местом, которое и определяет производительность чипа.

Hybrid Buffering

Поэтому почти все вендоры сегодня практикуют **гибридную буферизацию**, или, если хотите - динамическую. В нормальных условиях используется только on-chip память, предоставляющая line-rate производительность. А в случае перегрузки пакеты автоматически начинают буферизироваться во внешней памяти.
Это позволяет уменьшить стандартные задержки, энергопотребление и в большинстве случаев вписаться в ограниченную полосу пропускания до памяти.

> [Память сети: как сохранить пакет]

Источник.

> Данный параграф отменяет сказанное выше о том, что on-chip памяти не хватит для VOQ. Фактически, в случае гибридной буферизации она всё же дробится на тысячи очередей очень маленькой длины, чтобы обеспечить VOQ. Просто в нормальных условиях этой длины хватает, чтобы пропускать трафик мимо внешней памяти. При этом, в первую очередь начнёт офлоадиться на внешнюю память массивный трафик, идущий в низкоприоритетных очередях, а требовательный к задержками будет по-прежнему пролетать фаст-пасом.

Большие буферы - добро или зло?

В целом, это довольно старая дилемма. Подольше похранить или пораньше дропнуть?

Конечно, всем приложениям хочется lossless low-latency-сеть. Даже жирному некрасивому торренту. Но так не бывает и кем-то нужно жертвовать.
И мы долгое время живём с инертной мыслью, что часть приложений могут потерпеть задержки, а вот терять трафик совсем не хочется. Не в малой степени этому способствовало и то, что потери - это измеримая характеристика с более или менее понятными границами - потерь быть не должно. А что задержка? Вроде можно чётко сказать, что единицы миллисекунд - это хорошо, а секунды - это плохо. А между ними - зона спекуляций. Как оценить влияние вариаций задержки для рядового TCP-трафика?
Поэтому и спрос на устройства с большими буферами есть - никто не хочет терять трафик.

А теперь я выскажу не самое популярное мнение: потери - это хорошо.
Так уж вышло, что один из транспортных протоколов, фиксирует перегрузки, опираясь на потери. Дроп в очереди на сетевом устройстве означает, что на нём случился затор - это он не по своему капризу. И будет совсем не лишним, если отправители немного приуменьшат свои congestion window. Именно так и работают все классические (и не очень) реализации TCP Congestion Control. Соответственно, на устройствах с глубокими буферами во время заторов пакеты будут долго копиться, не отбрасываясь. Когда они всё-таки дойдут до получателя и тот их ACKнет, отправитель не только не снизит скорость, но может даже её увеличить, если у него сейчас режим Slow Start или Congestion Avoidance.
Можно взглянуть и дальше: растущая очередь взвинчивает RTT, что соответственно влечёт за собой

увеличение RTO таймеров на отправителях, тем самым замедляя обнаружение потерь.
То есть, сеть лишается своего практически единственного инструмента управления перегрузками.
И таким образом, архитекторы, пытающиеся решить вопрос заторов на сети путём увеличения буферов, усугубляют ситуацию ещё больше.
Ситуация, описанная выше, называется **bufferbloat** - распухание буфера.
Википедия иронично замечает:

> Проект www.bufferbloat.net иронично определил этот термин, как "ухудшение производительности Интернета, вызванное предыдущими попытками её улучшения".

Отбросы - санитары сети. Ко всеобщему удивлению, уменьшение очереди до одного пакета зачастую может кардинально улучшить ситуацию, особенно в условиях датацентра (только не сочтите это за дельный совет).

> Справедливости ради, следует заметить, что современные реализации TCP - BBR2, TIMELY ориентируются не только и не столько на потери, сколько на RTT и BDP. Гугловый QUIC - надстройку над UDP - следует отнести сюда же.

Внутри фабрики датацентра RTT ультракороткий - зачастую меньше 1 мс. Это позволяет среагировать на потерю очень быстро и купировать перегрузку в её зачатке.
Собственно поэтому практически все ASIC"и для датацентровых коммутаторов имеют только крохотную on-chip память. Хотя и появилась в последние годы тенденция к глубоких буферам и тут.
И этому даже находится объяснение.

Особая история на границе датацентра (или на устройствах доступа в сети провайдера или на магистральных сетях).
Во-первых, это места, которые обычно заведомо строятся с переподпиской, поскольку WAN-линки дорогие, что автоматически означает, что ситуации, в которых трафика приходит больше, чем способен переварить интерфейс, ожидаемы. А значит нужна возможность пакеты хранить и обрабатывать их в соответствии с приоритетами. Большие буферы позволяют сгладить всплески.
Во-вторых, обычно приложения настолько чувствительные к задержкам, никто не будет пытаться растягивать на этот сегмент. Например, RoCE или распределённое хранилище. Для чуть менее чувствительных, таких как телефония, в больших буферах выделяется приоритетная очередь.
В-третьих, тут задержки на устройстве всё ещё делают основной вклад в общее время доставки, но уже не настолько драматический.

Итак, устройства с большим объёмом памяти годятся в места где заложена переподписка или могут появиться заторы.

Что стоит отметить, так это то, что в датацентрах тоже есть ситуации, в которых 16-64 МБ буферов может не хватить, даже несмотря на отсутствие переподписки.
Два типичных примера - это обработка Big Data и Storage.

Анализ Big Data. Кластера Map-Reduce - это сотни и тысячи машин, которые перемалывают параллельно огромные массивы данных по заданию Master-узла, заканчивают примерно одинаково и все разом начинают возвращать ответы на Master-узел. Ситуация называется Incast. Длится она порядка нескольких десятков миллисекунд и потом исчезает.
On-chip память неспособна вместить эти данные - значит будет много дропов, значит ретрансмиты, значит общее снижение производительности.

Storage. Это штука крайне чувствительная к потерям и тоже гоняющая массивные объёмы данных. В случае хранилки тоже лучше не терять ничего. Но обычно она при этом и к задержкам предъявляет строгие требования, поэтому такие приложения обсудим пониже.

Однако, при этом крайне редко они единственные потребители сети в датацентрах, другим приложениям нужна низкая задержка.

Впрочем, это легко решается выделением очередей QoS с ограничением максимальной доступной глубины. И весь вопрос заключается тогда только в том, готова ли компания заплатить за глубокие буферы, возможно, не использовать их и поддерживать конфигурацию QoS.

Но в любой ситуации лучше следовать правилу: use shallow ASIC buffers when you can and use deep buffers when you must.

Критика глубоких буферов:

- Arista’s Big Buffer B.S.
- Incast.
- Speeding Applications in Data Center Networks. The Interaction of Buffer Size and TCP Protocol Handling and its Impact on Data-Mining and Large .Enterprise IT Traffic Flows.

Кстати, показательная таблица типичных задержек:

> [Память сети: как сохранить пакет]

Источник.

Low-latency lossless сети

С развитием RoCE, RDMA, nVME over Fabric к сети стали появляться другие, неслыханные доселе требования: и каждый пакет ценен, и времени терять нельзя. Одновременно. И хуже того, мы хотим по максимуму утилизировать имеющиеся линки, чтобы они не простаивали.
И все требования оправданы - осуждать их мы здесь не будем.

Так что же можно сделать? На самом деле, как я уже говорил выше, условия сети датацентра хороши тем, что информация о перегрузке очень быстро доносится до отправителей. Поэтому и реакция на потерянный пакет не заставит себя долго ждать.

Но что если нам не отбрасывать пакет, а каким-то другим способом сообщить отправителю, что нужно замедлиться? Здесь есть несколько конкурирующих подходов.

- ECN-Based.
- Bandwidth-Delay Product Based.
- RTT-BASED.

ECN-Based

Транзитное устройство, готовящееся испытать перегрузку, явным образом сообщает отправителям о том, что нужно охладить свою страсть.

Прозорливые инженеры заложили в IP целых 8 бит под QoS, и только 6 мы задействовали под DSCP, а 2 бита были зарезервированы для целей **ECN - Explicit Congestion Notification**.

Надолго забытый механизм сегодня извлекают из ящика стола, сдувают с него пыль, и вставляют в шкатулку, на которой нацарапано или DCTCP.

Транзитное устройство при заполнении буфера больше, чем до определённого порога, выставляет в заголовках IP обрабатываемых пакетов бит CE (Congestion Encountered) и отправляет пакет дальше. Получатель, увидев в пришедшем пакете этот флаг, сообщает отправителю о перегрузке, и о том, что нужно снизить скорость.

Классический TCP может обнаружить только уже существующую перегрузку, а DCTCP, используя ECN, узнаёт о том, что она только приближается, и пробует её избежать.

Есть и другие реализации TCP, поддерживающие ECN, например, HTCP.

Нюанс с ECN-based Congestion Control механизмами в том, что до поры до времени они ничего не знают о надвигающейся перегрузке, а потом должен пройти ещё целый RTT, чтобы отправитель узнал, что какое-то транзитное устройство к ней близко. К тому времени, как отправители начнут снижать скорость, перегрузка уже может или рассосаться или наоборот дойти до уровня, когда начнутся дропы.

Bandwidth-Delay Product Based

Другие реализации замеряют эффективную полосу сети совместно с RTT, то есть сколько можно ещё напихать в трубу до того, как это создаст затор и увеличит задержку.

Примерами таких протоколов являются BBR и H-TCP.

RTT-BASED

В конце концов есть элегантные механизмы, которые замеряют время прохода трафика туда-обратно. Идея провальная для MAN/WAN-сегментов, и, честно говоря, при попытке программной вычисления RTT тоже. TIMELY от Google с аппаратным offload"ом вычисления RTT один из наиболее

удачных примеров.

На самом деле, если бы не видео с прекрасной девушкой, рассказывающей про технические детали TIMELY, не знаю даже стал ли бы я упоминать про него. Наслаждайтесь, но берегите уши: TIMELY: RTT-based Congestion Control for the Datacenter.

Полезные ссылки

В этот раз хоть под кат убирай. Но, поверьте, я оставил тут только самые хорошие источники, прочитанные лично моими глазами и отобранные лично моими руками.

- **Архитектура памяти**
  - Packet Buffers. Отсюда по ссылкам разворачиваются разнообразные материалы очень глубоко.
  - An Update on Router Buffering.
  - NCS 5500 Buffering Architecture.
  - Cisco Nexus 5000 Series Switches. Cut-through and Store-and-Forward.
  - Understanding CoS Flow Control (Ethernet PAUSE and PFC).
  - Quality of Service. Headroom buffers.
- **Реализации очередей**
  - Strategies of packet buffering inside Routers.
  - Understanding CoS Virtual Output Queues (VOQs) on QFX10000 Switches.
  - What is VOQ and why you should care?
- **Deep Buffers**
  - Not all deep buffer switches are created equal.
  - Arista’s Big Buffer B.S.
  - Speeding Applications in Data Center Networks. The Interaction of Buffer Size and TCP Protocol Handling and its Impact on Data-Mining and Large Enterprise IT Traffic Flows.
- Data Center TCP (DCTCP): TCP Congestion Control for Data Centers.
- Understanding Microburst.

Заключение

Я предпринял несколько попыток написать короткое резюме всей статье, но каждый раз получалось почти полное её переписывание иными словами.
Поэтому просто замечу, что буферы - это неотъемлемая часть сетевого устройства. Но к тому же это место, где ко времени жизни пакетов могут добавиться лишние миллисекунды. И чем больше объём памяти, тем дольше трафик может в ней протомиться. Но даже в экстремальных случаях (8-гиговый буфер) нужно с большим подозрением и узкими глазами относиться к претензиям к сети о том, что, мол, у кого-то секундные задержки в пределах датацентра - ну негде там столько их хранить.
Вопрос "большие или маленькие буферы" обычно решается легко, когда отталкиваешься от задач и от позиции устройства в сети, но есть, конечно, места, где это вопрос философии или даже религии.

Спасибы

- Андрею Глазкову за рецензию и дельные замечания, Shared Buffers и коммерческих чипах.
- Михаилу Соколову за разъяснения об устройстве чипов, SerDes и Silicon Photonics.
- Александру Клименко за обнаружение точек роста в вопросах Admission Control, Alpha, Pipeline"ов и минусов DCTCP.
- Александру Азимову за комментарии о Lossless Ethernet.
- Артёму Чернобаю за КДПВ.