Network: Часть 6. Транспортный уровень

24.10.2026 09:50 edited 24.10.2026 09:50

Эпоха адаптивных систем

В сентябре 1981 года Джон Постел выпустил RFC 793 — спецификацию протокола управления передачей (TCP), над которой к тому моменту работали почти десять лет. Документ описывал механизм, который должен был обеспечить «надежную потоковую передачу данных между парой процессов в разнородных компьютерных сетях», и не содержал ни единого слова о вебе, стриминге, облаках и мобильных сетях — ничего этого еще не существовало. Сорок пять лет спустя TCP переносит практически всю текстовую и значительную часть медиаинформации планеты: каждое открытие страницы, каждая покупка, каждый банковский перевод начинаются с трех пакетов рукопожатия, придуманного в эпоху, когда весь Интернет состоял из нескольких сотен хостов.

Парадокс транспортного уровня в том, что он строит порядок из хаоса. Сетевой уровень под ним — IP — не гарантирует ничего: ни доставки, ни порядка, ни отсутствия дублей. И поверх этой «почты, которая теряет письма» транспортный уровень возводит надежный байтовый поток, по которому приложения общаются так, будто между ними проложен идеальный провод. Более того, именно транспортный уровень — а не маршрутизаторы — управляет сегодня скоростью всего Интернета: алгоритмы контроля перегрузки в TCP-стеках миллиардов устройств коллективно решают, кто и с какой скоростью занимает общие каналы. Никакого центрального регулятора у мировой сети нет — его роль децентразованно играют оконные алгоритмы, изобретенные Вэном Джейкобсоном в 1988 году.

Эта архитектура порождает ряд критических вопросов:

1. **Архитектурный вопрос.** Почему на уровне оказалось ровно два «официальных» протокола — тяжеловесный надежный TCP и минималистичный UDP, — и почему второй стал «конструктором», из которого индустрия собрала RTP, DTLS, QUIC и WebRTC? Что потеряно и что выиграно в этой асимметрии?
2. **Вопрос надежности.** Как из сети, теряющей пакеты, собрать упорядоченный поток без единой потери? Ответ — скользящее окно, подтверждения и таймеры, — но у задачи есть и фундаментальная сторона: «проблема двух генералов» доказывает, что идеальная надежность в принципе недостижима, и вся инженерия уровня есть работа с этим пределом.
3. **Вопрос скорости.** Пропускная способность соединения равна окну, деленному на время кругового рейса (RTT). Шестнадцатибитное окно оригинального TCP ограничивало поток 64 килобайтами «в полете» — при современном RTT и гигабитных каналах этого

не хватило бы уже в девяностых. История «длинных толстых труб» — это история борьбы с собственным заголовком протокола.

4. **Вопрос перегрузки и справедливости.** TCP трактует потерю пакета как сигнал перегрузки и урезает скорость вдвое, из этого простого правила рождается глобальная справедливость раздела каналов — и масса патологий: несправедливость к дальним соединениям, «пила» пропускной способности, слепота к буферизации (bufferbloat), войны алгоритмов CUBIC и BBR.
5. **Вопрос эволюции.** TCP оказался заложником собственного успеха: он живет в ядрах операционных систем миллиардов устройств, а промежуточное оборудование «окостенело» вокруг его заголовков так, что новые опции протокола почти невозможно внедрить. Ответом стал QUIC — транспорт в пользовательском пространстве, поверх UDP, с полностью зашифрованными заголовками, — за десять лет прошедший путь от эксперимента Google до стандарта, на котором работает свыше 40% веб-сайтов.
6. **Вопрос безопасности.** Рукопожатие TCP — поверхность атаки: SYN-флуд исчерпывает очереди серверов, предсказуемость номеров последовательности позволяла перехватывать сессии (знаменитая атака Митника 1995 года), а «слепые» сбросы соединения используют даже государственные файрволы. У транспорта нет встроенной криптографии — ее добавили поверх, и этот шов до сих пор определяет облик протоколов.
7. **Вопрос производительности.** На скоростях 100 Гбит/с узким местом становится не сеть, а обработка: системные вызовы, копирование памяти, прерывания. Проблема C10K, событийные API, zero-copy, обход ядра и RDMA — вся эта индустрия существует потому, что цена доставки байта приложению в двадцать первом веке сравнялась с ценой его передачи по проводу.

Дальше мы пройдем весь транспортный уровень: от исторического разделения TCP и IP и берклиевских сокетов — через порты, UDP, устройство TCP-сегмента, рукопожатия и конечный автомат, надежность и таймеры, управление потоком и перегрузкой (от Tahoe 1988 года до BBR) — к производительности ядерных стеков, альтернативным протоколам (SCTP, MPTCP), QUIC и HTTP/3, реальному времени и RPC, безопасности и практическим ловушкам вроде TIME_WAIT и «склеенных пакетов». В финале проследим жизненный цикл одного соединения — от SYN до последнего ACK.

Рождение транспортного уровня

От единого протокола к паре TCP/IP

В статье Серфа и Кана 1974 года протокол TCP был единым целым: он отвечал и за адресацию, и за маршрутизацию дейтаграмм через границы сетей, и за надежную доставку между процессами. К 1977 году стало ясно, что один протокол решает слишком разные задачи — и, что хуже, делает это с разной необходимой скоростью. Разделение произошло в 1978 году, во многом под влиянием Дэнни Коэна, работавшего над передачей голоса через ARPANET: его эксперименты показали, что для речи надежная доставка с

повторными передачами вредна — просроченный пакет бесполезен, а задержка повторной передачи убивает интерактивность. Приложениям нужны были два разных сервиса: надежный поток для файлов и ненадежная датаграмма для голоса.

Так образовалась пара: IP взял на себя доставку дейтаграмм между хостами, TCP — надежный поток между процессами. В августе 1980 года вышел RFC 768 — спецификация UDP, протокола, который добавлял к IP ровно два свойства: порты и контрольную сумму. В сентябре 1981-го — RFC 793 (TCP) вместе с RFC 791 (IP) и RFC 792 (ICMP). А в 1989 году RFC 1122 под редакцией Роберта Брэйдена свел воедино требования к хостам: документ, который до сих пор формально определяет, что обязан делать корректный TCP/IP-стек.

Это разделение уровней — «сеть доставляет на машину, транспорт доставляет процессу» — и есть определяющая идея, с которой начинается весь разговор. IP не знает, сколько процессов работает на хосте, TCP и UDP не знают, сколько сетей лежит между хостами. Каждый уровень решает свою задачу и ничего сверх.

Проблема двух генералов: фундаментальный предел надежности

Прежде чем разбирать механизмы, стоит понять теоретическую границу, в которую они упираются. «Задача двух генералов» — мысленный эксперимент, формализованный в середине 1970-х (Акоюнлу, Эсваран и Хубер, 1975, классическую формулировку дал Джим Грей в 1978-м): два генерала, осаждающие город, должны согласовать одновременную атаку, но сообщения между ними проходят через долину, где их перехватывают. Сколько бы подтверждений ни посылали генералы друг другу, последнее подтверждение может быть потеряно — и ни один из них не может быть абсолютно уверен, что другой уверен. Формальный результат: надежное согласование состояния по ненадежному каналу в общем случае невозможно.

Для транспортного уровня это не философия, а инженерная константа. Когда клиент отправляет запрос и не получает ответа, он принципиально не может отличить три ситуации: запрос потерялся, сервер его не выполнил, запрос дошел, сервер выполнил, но ответ потерялся, сервер умер в процессе. Отсюда — вечная дилемма «ровно один раз» (exactly-once): без идеального канала достижимы лишь «хотя бы раз» (at-least-once, с риском повторов) и «не более раза» (at-most-once, с риском невыполнения). TCP выбирает at-least-once для данных внутри установленного соединения, а приложения, которым критична неповторяемость, вынуждены делать операции идемпотентными и вводить идентификаторы запросов. Вся теория распределенных систем — от повторных вызовов RPC до коммит-протоколов баз данных — вырастает из этой невозможности, открытой на заре сетевого проектирования.

Сокеты: берклиевский интерфейс, победивший всех

Механизмы уровня бессмысленны без интерфейса, через который приложения ими пользуются. В 1983 году в системе 4.2BSD команда Билла Джоя и Сэма Леффлера представила API сокетов: socket(), bind(), listen(), connect(), accept(), send(), recv(), close(). Гениальность интерфейса была не в функциях, а в метафоре: сокет выглядел как

дескриптор файла. Программист писал сетевой код теми же системными вызовами, что и файловый, — read/write работали и с диском, и с сетью, а Unix-философия «все есть файл» неожиданно покрыла и распределенные системы.

Конкурент был серьезный: AT&T продвигала Transport Layer Interface (TLI) в System V — «правильный», не привязанный к файловой модели API. Войну интерфейсов выиграл BSD: сначала потому, что 4.2BSD раздавали университетам дешево, затем потому, что на сокетах выросли все громкие приложения эпохи — telnet, FTP, позднее веб. TLI остался сноской в истории, а сокеты стали настолько всеобщими, что Winsock, POSIX-sockets и даже современные асинхронные API продолжают их семантику. Сокет — это точка, в которой абстракция «соединение» превращается в конкретную пару (IP-адрес, порт) для UDP и четверку (IP-адрес и порт источника, IP-адрес и порт назначения) для TCP.

Аналогия для запоминания классическая: IP-адрес — это дом, порт — квартира, сокет — конкретная квартира, в которой живет процесс. Почтальон (IP) доставляет письмо до дома, почтовый ящик с номером квартиры (порт) решает, какому жильцу (процессу) оно достанется.

Задачи транспортного уровня

Сформулируем уровень строго. Транспортный уровень обеспечивает передачу данных между процессами поверх сети, доставляющей пакеты между хостами. Единица его данных — сегмент (segment), для сравнения: канальный уровень оперирует кадрами (frames), сетевой — пакетами (packets). Цепочка инкапсуляции выглядит так:

```
данные приложения → TCP-сегмент → IP-пакет → Ethernet-кадр → сигнал
```

Сервис уровня описывается набором примитивов — операций, которые приложение может запросить у транспорта: LISTEN (ожидать входящего соединения), CONNECT (установить соединение), SEND (передать данные), RECEIVE (принять данные), DISCONNECT (завершить соединение). В Berkeley API этим примитивам почти один-в-один соответствуют listen(), connect(), send(), recv() и close() — редкий случай, когда абстракция стандарта и абстракция реализации совпали полностью.

Доставка между процессами: порты

IP-адрес идентифицирует хост, но на хосте работают десятки процессов, и нескольким из них может быть нужно получать данные одновременно. Порт — шестнадцатибитный номер, идентифицирующий конечную точку внутри хоста, пара (IP, порт) называется endpoint'ом соединения. Знакомые каждому номера: 80 — HTTP, 443 — HTTPS, 22 — SSH, 53 — DNS, 25 — SMTP. Пространство портов делится на три диапазона (реестр ведет IANA, правила — RFC 6335): «известные» порты 0–1023, за которыми закреплены стандартные службы и которые в Unix-системах исторически может занимать только привилегированный процесс (root), «зарегистрированные» 1024–49151 для приложений, и «динамические» (эфемерные) — примерно 49152–65535, из которых ОС выделяет

исходящие соединения. В Linux диапазон эфемерных портов по умолчанию — 32768–60999, то есть около 28 тысяч портов, и эта цифра всплывет позже, когда речь пойдет об исчерпании.

Мультиплексирование и демультиплексирование

Мультиплексирование — способность многих приложений одновременно пользоваться одним транспортным стеком и одним IP-адресом. Демультиплексирование — обратная операция: доставить прибывший сегмент нужному процессу. И здесь кроется тонкость, которую часто пропускают: у UDP демультиплексирование идет по паре (IP назначения, порт назначения), а у TCP — по полной четверке (оба адреса и оба порта). Разница принципиальна: один UDP-сокет, привязанный к порту 53, принимает датаграммы от всех отправителей сразу (так работают DNS-серверы), а TCP-соединение однозначно определяется четверкой, поэтому тысячи клиентов могут быть подключены к одному и тому же серверному порту 443 — каждое соединение уникально своим клиентским адресом и портом. Именно это позволяет веб-серверу с одним listening-сокетом обслуживать десятки тысяч одновременных сессий.

Службы уровня: от надежного потока до «конструктора»

Транспортный уровень предоставляет два полярных сервиса. Надежный байтовый поток с установлением соединения, гарантиями доставки и порядка, управлением потоком и контролем перегрузки — TCP. И минимальную ненадежную датаграммную службу без соединений и гарантий — UDP. Все остальное многообразие (RTP, SCTP, QUIC, DTLS) построено между этими полюсами или поверх одного из них — и это, забегая вперед, главный сюжет сорокалетней эволюции уровня.

UDP: минимальный транспорт

Что делает UDP — и чего он принципиально не делает

UDP (User Datagram Protocol, RFC 768, август 1980 года) — самый маленький протокол семейства: его заголовок занимает восемь байт и содержит ровно четыре поля: порт источника, порт назначения, длину датаграммы и контрольную сумму. Никаких соединений, никаких подтверждений, никаких повторных передач, никакого порядка, никакого контроля перегрузки. Датаграмма отправляется одним системным вызовом и уходит в сеть как есть — «отправил и забыл» (send and forget). Если приложение послало сообщение в три килобайта, получатель получит ровно одно сообщение в три килобайта: границы сообщений сохраняются, в отличие от TCP.

За эти восемь байт приложение получает ровно два свойства, которых нет у голого IP: мультиплексирование по портам и (опционально в IPv4, обязательно в IPv6) контроль целостности. Все остальное UDP оставляет реализовать самому приложению — и это не недостаток, а архитектурная позиция: минимальный сервис, максимальная свобода. Недаром UDP называют «конструктором для своих протоколов»: почти все, что было изобретено в транспорте за последние сорок лет, строилось поверх него.

Контрольная сумма и псевдозаголовок

Контрольная сумма UDP устроена любопытно: она считается не только по полям UDP и данным, но и по «псевдозаголовку» — фрагменту IP-заголовка с адресами источника и назначения и номером протокола. Формально это нарушение уровнявания (транспорт заглядывает в сетевой уровень), практически — изящная страховка: датаграмма, доставленная не тому хосту из-за сбоя маршрутизации, будет отброшена, потому что адреса в псевдозаголовке не сойдутся. В IPv4 контрольную сумму разрешено не использовать (нулевое значение) — поблажка для медленных хостов восьмидесятых, которая обернулась источником трудноуловимых ошибок, в IPv6 она обязательна. Отдельная вариация — UDP-Lite (RFC 3828, 2004), позволяющая покрывать контрольной суммой только часть данных: кодек может защитить заголовок кадра и смириться с битами в младших разрядах изображения, что на зашумленных беспроводных каналах дает картинку с артефактами вместо пропажи кадра.

Где UDP — правильный выбор

Список приложений UDP читается как список всего, что чувствительно к задержке или просто не нуждается в надежности: DNS (короткий запрос — короткий ответ, повторная передача дешевле, чем установление соединения), VoIP и видеоконференции (просроченный пакет бесполезен, а потерянный кадр лучше заменить следующим), онлайн-игры (состояние мира обновляется десятки раз в секунду, старый апдейт не «чинят», а выбрасывают), потоковое вещание, DHCP, NTP, SNMP. Общая логика: когда стоимость ожидания потерянного пакета выше стоимости его потери, надежность становится антифункцией. К этому же классу относится все реальное время — о RTP, WebRTC и родственных протоколах речь пойдет отдельно.

Цена свободы: отсутствие контроля перегрузки и «окостенение»

У свободы есть и обратная сторона. UDP не тормозит при перегрузке: приложение, шлющее датаграммы без ограничений, способно залить общий канал — именно поэтому «честные» реализации (RTP, QUIC) обязаны добавлять свой контроль перегрузки поверх, а RFC 8085 (2017) «UDP Usage Guidelines» прямо предписывает прикладным протоколам либо ограничивать скорость, либо вести себя «TCP-дружественно». Вторая проблема — middleboxes: промежуточное оборудование десятилетиями «училось» на TCP и подозрительно относится к голому UDP, некоторые сети режут UDP-трафик или ограничивают его, часть операторских устройств ломает датаграммы, фрагментируя их небезопасным образом. Поэтому современный UDP редко ходит «голым»: поверх него почти всегда лежит еще один слой — DTLS (TLS для датаграмм), или QUIC, или RTP со своими заголовками, — который заодно защищает протокол от вмешательства middleboxes и делает трафик менее различимым для фильтров. Эта гонка между протоколами и промежуточным оборудованием — предвестник большой темы «окостенения» (ossification), к которой мы вернемся в разделе о QUIC.

TCP: протокол, на котором держится надежный Интернет

Сегмент TCP: заголовок по полям

TCP передает данные сегментами, каждый сегмент — заголовок (минимум 20 байт, до 60 с опциями) и полезная нагрузка. Разберем заголовок (RFC 793 с последующими дополнениями) в порядке следования полей.

Source Port / Destination Port (по 16 бит) — четверка вместе с IP-адресами однозначно идентифицирует соединение.

Sequence Number (32 бита) — номер первого байта данных этого сегмента в потоке, в сегменте с флагом SYN — начальный номер последовательности (ISN), с которого начинается отсчет.

Acknowledgment Number (32 бита) — номер байта, который отправитель подтверждения ожидает получить следующим, подтверждение кумулятивное: «все до этого байта получено».

Data Offset (4 бита) — длина заголовка в 32-битных словах (аналог IHL у IP).

Flags (8 бит) — управляющие биты: CWR и ECE (эхо ECN — сигналы перегрузки из RFC 3168), URG (срочные данные), ACK (поле подтверждения действительно), PSH (просьба передать данные приложению немедленно), RST (аварийный сброс), SYN (установка соединения), FIN (завершение). Из восьми флагов в повседневной жизни соединения участвуют четыре: SYN, ACK, FIN, RST. Два других — реликвии ранней эпохи. URG знаменует «срочные данные»: механизм унаследован от telnet, где так передавали команду прерывания, но реализации расходились даже в трактовке указателя срочности (RFC 793 и BSD понимали его по-разному), и сегодня он практически мертв. PSH — просьба к стеку получателя «отдать данные приложению немедленно, не дожидаясь заполнения буфера»: не правило, а подсказка (hint), которую современные реализации часто игнорируют — интерактивность обеспечивают настройками вроде TCP_NODELAY.

Window Size (16 бит) — размер окна получателя: сколько байт отправитель еще может прислать, не дожидаясь разгрузки. Шестнадцати бит хватает лишь на 64 килобайта — это ограничение стало одной из главных проблем TCP и породило опцию масштабирования окна, о ней в разделе об управлении потоком.

Checksum (16 бит) — контрольная сумма заголовка, данных и псевдозаголовка IP, как у UDP.

Urgent Pointer (16 бит) — смещение срочных данных, практически мертв.

Options — переменная часть: MSS (максимальный размер сегмента, который готов принимать отправитель опции), Window Scale (множитель окна), SACK Permitted (разрешение выборочных подтверждений), Timestamps (метки времени для точного измерения RTT и защиты от «зацикливания» номеров). Опции согласуются в рукопожатии — и именно с ними TCP приобрел большую часть своей современной функциональности.

MSS и связь с MTU

MSS (Maximum Segment Size) — максимальный объем данных в сегменте. Он вычисляется из MTU пути: для классического Ethernet MTU 1500 минус 20 байт заголовка IP минус 20 байт заголовка TCP дают MSS 1460. Согласование MSS в рукопожатии — первый эшелон защиты от фрагментации на сетевом уровне: стороны заранее договариваются не посылать друг другу сегменты, которые пришлось бы резать. Для туннелей (IPsec, GRE, PPPoE) MSS пересчитывают, а домашние роутеры часто применяют MSS clamping — «прищипывание» опции в проходящих рукопожатиях, чтобы клиенты не ломились пакетами, не пролезающими в туннель.

Поток байтов, а не сообщений

TCP — байто-ориентированный протокол: приложение пишет в соединение последовательность байтов, и транспорт обязан доставить ее получателю в том же порядке, без пропусков и повторов. Границ сообщений при этом не существует. Два вызова send() с «Hello» и «World» могут прийти одним recv() как «HelloWorld», а могут — тремя кусками «He», «lloWo», «rld»: TCP вправе склеивать и дробить данные как угодно, лишь бы байты шли по порядку. Это не баг, а следствие оптимизаций (алгоритм Нейгла копит мелкие порции, стек объединяет сегменты), но для разработчиков — вечный источник ошибок, известный как «липкие пакеты» (sticky packets). Правило одно: framing — обязанность прикладного протокола. Границы задаются либо длиной (префикс: сначала 4 байта «сколько дальше читать»), либо разделителем (строки, заканчивающиеся \n, — так работают HTTP, Redis, IRC, SMTP), либо фиксированным размером записей. Протокол, который «просто шлет JSON в сокет» и полагается на границы send(), сломается при первой же фрагментации — рано или поздно, обычно в продакшене.

Номера последовательности и ISN: от атак Митника к рандомизации

Номера последовательности решают сразу три задачи: задают порядок байтов, позволяют кумулятивные подтверждения и — что важнее всего исторически — защищают от дублей. Пакет, задержавшийся в сети и пришедший повторно, будет отброшен, если его байты уже покрыты подтвержденным диапазоном: номер последовательности превращает «старый» сегмент в мусор. Счетчик 32-битный и на современных каналах зацикливается за секунды — при 1 Гбит/с примерно за полминуты, при 10 Гбит/с еще быстрее, — поэтому вместе с метками времени работает механизм PAWS (Protection Against Wrapped Sequences), отсекающий сегменты со «старыми» timestamps.

Отдельная драма связана с выбором начального номера (ISN). RFC 793 предписывал увеличивать счетчик на единицу каждые четыре микросекунды — предсказуемо. В рождество 1995 года Кевин Митник воспользовался этим для одной из самых знаменитых атак в истории: он «ослепил» трастовый хост (завалив его SYN-флудом), вычислил по алгоритму номер последовательности, который тот сгенерирует следующим, и от имени доверенной машины установил rlogin-сессию с компьютерами Цутому Симуры — специалиста по безопасности, который Митника и ловил. Атака не требовала перехвата трафика: достаточно было угадать ISN. Ответом стал RFC 1948 (1996) Стивена Белловина:

начальный номер должен быть криптографически стойкой функцией от четверки соединения и секретного ключа. Современные стеки генерируют ISN именно так — и атака предсказания последовательности, с которой началась карьера «борца с хакерами» Симуры и закат Митника, ушла в прошлое.

Контрольная сумма TCP: слабое звено, которое не меняют

Шестнадцатибитная сумма по дополнению до единицы (ones' complement) — дешевый, но слабый код: она не обнаруживает, например, перестановку двух 16-битных слов. В 2000 году исследование Стоуна и Партриджа «When The CRC and TCP Checksum Disagree» измерило реальность: сбои контрольных сумм встречались на порядки чаще, чем предсказывала теория (на отдельных маршрутах — вплоть до одного пакета из тысячи), а значит, память и аппаратура ошибаются чаще, чем считалось, и TCP-контрольная сумма регулярно ловит то, что не поймал канальный CRC. Заменить ее на что-то сильное не позволяют совместимость и стоимость, вместо этого индустрия пошла по пути разделения труда: целостность данных приложений обеспечивают TLS и прикладные протоколы, а контрольная сумма TCP осталась быстрым фильтром грубых ошибок — часто вычисляемым сетевой картой (checksum offload), чтобы не жечь циклы CPU.

Жизненный цикл соединения

Трехэтапное рукопожатие: почему именно три

Установка TCP-соединения — обмен тремя сегментами: клиент шлет SYN со своим начальным номером X, сервер отвечает SYN-ACK, подтверждая X+1 и сообщая свой номер Y, клиент завершает ACK-ом, подтверждающим Y+1. После этого обе стороны синхронизированы по номерам и соединение установлено — состояние ESTABLISHED, в котором и происходит передача данных.

Почему не два этапа? Именно из-за «задержавшихся дублей», тех самых, от которых защищают номера последовательности. Представьте старый SYN, пролежавший в сети минуты и пришедший на сервер, когда клиент уже давно передумал соединяться: при двухэтапной схеме сервер открыл бы соединение в никуда и держал бы ресурсы под фантомного клиента. Третий шаг дает клиенту возможность подтвердить (или, увидев неожиданное, сбросить) — соединение открывается только тогда, когда обе стороны убедились, что говорят с живым собеседником о текущем моменте. Одновременное открытие (обе стороны шлют SYN навстречу) протоколом тоже предусмотрено — оно проходит через состояние SYN-RECEIVED у обоих, — но на практике встречается редко.

В рукопожатии же согласуются опции: MSS, масштабирование окна, разрешение SACK, метки времени. Это единственный момент, когда стороны «договариваются о языке», — после него набор возможностей соединения заморожен на всю его жизнь. Отдельная модернизация — TCP Fast Open (RFC 7413, 2014): cookie, полученная при предыдущем

соединении, позволяет приложить данные прямо к SYN и сэкономить целый RTT на старте. Для веба, где соединения живут миллисекунды, это существенно — и ровно ту же мотивацию позже реализует QUIC со своими 0-RTT и 1-RTT.

Конечный автомат: одиннадцать состояний

TCP — не просто протокол, а автомат состояний, и знание его карты отличает инженера от пользователя API. Полный набор: CLOSED, LISTEN (сервер ждет), SYN-SENT (клиент послал SYN), SYN-RECEIVED (сервер получил SYN, послал SYN-ACK), ESTABLISHED (передача данных), а далее — каскад закрытия: FIN-WAIT-1 (послали FIN), FIN-WAIT-2 (наш FIN подтвержден, ждем FIN собеседника), CLOSE-WAIT (получили FIN собеседника, приложение еще не закрыло сокет), LAST-ACK (послали свой FIN, ждем последнего ACK), TIME-WAIT (ждем исчезновения старых сегментов) и снова CLOSED.

Нормальный жизненный цикл выглядит так. При открытии клиент проходит путь CLOSED — SYN-SENT — ESTABLISHED, сервер — LISTEN — SYN-RECEIVED — ESTABLISHED. Закрытие — «четырехэтапное рукопожатие»: каждая сторона независимо посылает FIN и получает на него ACK, потому что поток полудуплексен — закрыв направление «от себя», можно еще принимать данные. Активно закрывающая сторона (обычно клиент, но часто и сервер) проходит FIN-WAIT-1, FIN-WAIT-2 и TIME-WAIT до CLOSED, пассивная — CLOSE-WAIT и LAST-ACK до CLOSED. Диагностическое значение карты огромно: тысячи сокетов в CLOSE-WAIT — это не сетевая проблема, а программа, которая не вызывает close() (FIN получен, приложение «забыло» закрыть свою сторону), тысячи сокетов в TIME-WAIT — нормальное следствие активного закрытия, и лечится оно иначе.

TIME-WAIT: состояние, которое все ненавидят зря

TIME-WAIT длится 2×MSL (MSL — максимальное время жизни сегмента, RFC 793 предлагал две минуты, практические реализации обычно берут 30 секунд — отсюда фиксированные 60 секунд TIME-WAIT в Linux). Зачем оно нужно? Две причины. Первая — надежность последнего ACK: если он потеряется, собеседник повторит свой FIN, и сторона в TIME-WAIT сможет ответить повторно, — без этого тот застрял бы в LAST-ACK, а соединение закрылось бы «грязно». Вторая — защита от призраков: старые сегменты умершего соединения, блуждающие по сети, должны умереть раньше, чем появится новое соединение с той же четверкой адресов и портов. TIME-WAIT — карантин, гарантирующий, что прошлая жизнь четверки закончилась до начала следующей.

Практическая боль известна каждому, кто эксплуатировал высоконагруженный бэкенд: веб-сервер или прокси, активно закрывающий десятки тысяч коротких соединений в секунду, копит десятки тысяч сокетов в TIME-WAIT, эфемерные порты (те самые ~28 тысяч в Linux) исчерпываются, и новые исходящие соединения не создаются. Лекарства: закрывать соединение на стороне клиента, а не сервера (тогда TIME-WAIT живет у клиента), переиспользовать соединения (keep-alive — лучшая профилактика), SO_REUSEPORT (Linux 3.9, 2013), позволяющий нескольким сокетам слушать один порт и делить нагрузку, tcp_tw_reuse для исходящих соединений. А вот знаменитый tcp_tw_recycle — агрессивное сокращение TIME-WAIT по меткам времени — оказался опасен за NAT (одна

метка на множество клиентов ломала логику) и был удален из ядра Linux в 2017 году. Мораль: TIME-WAIT — не болезнь, а иммунитет, бороться надо с причиной его накопления, а не с самим состоянием.

RST: аварийный сброс и оружие цензуры

Флаг RST обрывает соединение немедленно, без рукопожатий: порт никто не слушает (классика — SYN на закрытый порт получает RST), сегмент пришел на несуществующее соединение, приложение требует аварийного закрытия (SO_LINGER с нулем) или данные пришли в недействительное окно. У RST есть и темная сторона: «слепая» инъекция сброса. Атакующий, не видя трафик, но зная (или угадав) четверку и приблизительный номер последовательности, может послать поддельный RST и разорвать чужое соединение. Это не теория: измерения 2005–2006 годов показали, что «Великий китайский файрвол» массово применяет RST-инъекции для разрыва «нежелательных» сессий, посылая сразу несколько пакетов с разными номерами последовательности — чтобы хотя бы один попал в окно. RFC 5961 (2010) затруднил такие атаки (на подозрительный RST теперь положено отвечать challenge ACK, а не рвать соединение), но полностью проблему «слепого» вмешательства не снял: транспорт без криптографии подделываем в принципе.

Полураскрытые соединения и keepalive

Если хост внезапно умирает (крах, выдернутый кабель), его собеседник об этом не узнает: соединение останется «полураскрытым» — одна сторона считает его живым, другой больше нет. Данные, посланные в никуда, будут копиться в буферах, пока не исчерпаются. Средство — keepalive: если в соединении долго (по умолчанию в Linux — два часа) нет трафика, сторона шлет пустой зонд, несколько зондов без ответа подряд объявляют соединение мертвым. Двухчасовой таймаут почти никем не устраивает, поэтому в продакшене его уменьшают (или, что правильнее, используют прикладные heartbeats — протоколы вроде WebSocket ping/pong, gRPC keepalive или AMQP heartbeat, которые живут в потоке данных и не зависят от настроек стека). Keepalive TCP — последний рубеж, а не рабочий инструмент.

Надежность: подтверждения, таймеры, повторные передачи

Кумулятивные ACK и дублирующие подтверждения

Основа надежности TCP — подтверждения: получатель в каждом ACK сообщает «следующий ожидаемый байт N», что эквивалентно «все байты до N получены». Подтверждение кумулятивное — компактное и устойчивое к потерям самих ACK (потерянное подтверждение заменяется следующим), но грубое: если из середины потока пропал один сегмент, получатель продолжает слать одно и то же число — дублирующие ACK. Для отправителя три дубликата подряд — почти наверняка признак потери (а не задержки): ждать таймаута незачем, сегмент переотправляется немедленно. Так работает быстрая повторная передача (fast retransmit), добавленная в стек в 1988 году.

У кумулятивной схемы есть слепое пятно: отправитель не знает, что получено после дыры. Закрыл его SACK (Selective Acknowledgment, RFC 2018, 1996) — опция, позволяющая ACK'у перечислять дополнительно принятые блоки вне порядка: «жду байт 1000, но 2000–3000 и 4000–5000 уже получил». С SACK повторная передача становится точечной — переотправляется только реально потерянное. Совместно они дают поведение, близкое к модели selective repeat из учебников: базово TCP — это go-back-N с быстрым восстановлением, а с SACK — почти выборочный повтор. Современное дополнение — механизм RACK (стандартизован RFC 8985 в 2021 году): потерю определяет не счет дубликатов, а время — сегмент считается потерянным, если более поздний уже подтвержден, а этот «просрочен» относительно измеренного RTT. RACK работает там, где счет дубликатов бессилен (короткие потоки, малое окно), и стал стандартом в современных ядрах.

Таймеры и оценка RTT: адаптивность как принцип

Главный таймер TCP — RTO (Retransmission Timeout): сколько ждать ACK перед повторной передачей. Слишком короткий таймер плодит ложные ретрансмиссии (дубли грузят сеть и сбивают контроль перегрузки), слишком длинный — оставляет соединение простаивать после реальной потери. Фиксированное значение невозможно: RTT в Интернете варьируется от миллисекунд в ЦОД до секунд на спутниковых каналах и «дышит» в течение одной сессии. Решение — адаптивная оценка: скользящее среднее SRTT плюс оценка разброса RTTVAR, по формуле Джейкобсона — Карелса (1988) RTO = SRTT + 4·RTTVAR (стандартизована в RFC 6298, нижняя граница в Linux — 200 мс). Отдельная тонкость — алгоритм Карна (Фил Карн, 1987): измерять RTT по сегментам, переданным повторно, нельзя — непонятно, подтверждение это первой передачи или второй, такие отсчеты отбрасываются, а при повторных потерях RTO растет экспоненциально (backoff).

Таймеров у соединения несколько: RTO — для данных, persist timer — для дозора при нулевом окне (см. управление потоком), keepalive — для обнаружения смерти собеседника, FIN-WAIT-2 таймер — чтобы сторона, уже закрывшая свое направление, не висела вечно в ожидании чужого FIN. Совокупность этих механизмов и делает TCP «самоуправляемым»: соединение без вмешательства человека переживает потери, задержки, паузы получателя и частично — смерть одной из сторон.

Буферизация на обоих концах

Надежность стоит памяти. Отправитель хранит посланные, но еще не подтвержденные байты (их может потребоваться передать заново) и данные, принятые от приложения, но еще не уместившиеся в окно. Получатель хранит принятые вне порядка сегменты (дыра в середине потока обязывает держать «хвост» до ее заполнения) и данные, которые приложение еще не забрало. Отсюда — прямая связь транспорта с производительностью системы: размеры сокетных буферов ограничивают достижимую скорость (окно не может превысить буфер), а патологические сценарии (медленное приложение на приеме, глубокая переупорядоченность в сети) превращают буферы в источник задержек и

давления на память. Большой буфер не равен хорошей производительности — эта максима, доказанная на сетевых очередях явлением bufferbloat, в полной мере относится и к конечным узлам.

Порядок доставки и блокировка во главе очереди

TCP гарантирует порядок: сегменты, пришедшие раньше времени, буферизуются и отдаются приложению только сплошным префиксом потока. Расплата — head-of-line blocking (блокировка во главе очереди): если сегмент №2 потерян, байты сегментов №3 и №4, уже принятые и лежащие в буфере, приложению не отдаются — поток «замерз» в ожидании дыры, даже если потерянные данные нужны не срочно. Для передачи файла это правильная цена, для мультиплексированных приложений — катастрофа: в HTTP/2 десятки логических потоков делят одно TCP-соединение, и потеря одного сегмента останавливает их все. Это фундаментальное противоречие «порядок против независимости» — главный мотив появления QUIC, где каждый поток надежен сам по себе и потеря в одном не блокирует другие.

Управление потоком: не перегрузить получателя

Скользящее окно и окно получателя

Управление потоком (flow control) решает задачу, отличную от перегрузки: отправитель не должен слать данные быстрее, чем получатель успевает их обрабатывать. Механизм — скользящее окно. Получатель в каждом сегменте (в поле Window Size) сообщает, сколько байт он готов принять сверх уже подтвержденного, отправитель держит «в полете» не более этого объема. По мере того как приложение забирает данные из буфера, окно растет, по мере отправки — «скользит» вперед фактически признанная граница. Фактический лимит передачи — минимум двух окон: отправитель вправе держать в полете не больше, чем разрешают одновременно окно получателя (rwnd — flow control) и окно перегрузки (cwnd — congestion control). Два независимых тормоза, и в каждый момент работает более строгий.

Сама идея «отправить N сегментов, не дожидаясь подтверждений» — обобщение трех учебных схем надежной передачи. Stop-and-wait (послал один пакет, ждешь ACK, посылаешь следующий) проста и безнадежно неэффективна: ее пропускная способность равна размеру пакета, деленному на RTT, — на канале 1 Гбит/с с задержкой 100 мс полуторакилобайтный пакет дает около 120 кбит/с — десятитысячную долю доступного. Go-Back-N (окно без выборочного повтора: потерялся пакет — переотправляй все после него) тратит полосу на лишние передачи. Selective Repeat (переотправляется ровно потерянное) оптимален по полосе, но требует буферов вне порядка у получателя. TCP, как уже говорилось, — практичный гибрид: кумулятивные ACK тяготеют к go-back-N, а fast retransmit и SACK приближают поведение к selective repeat.

Нулевое окно и persist-таймер: анатомия взаимной блокировки

Приложение на приеме может замедлиться — и окно схлопнется в ноль. Отправитель останавливается. Возникает опасная ситуация: когда получатель освободит буфер и пошлет обновление окна («window update»), это сообщение может потеряться — и обе стороны будут вечно ждать друг друга: получатель считает, что окно объявил, отправитель — что оно по-прежнему нулевое. Взаимная блокировка (deadlock) лечится persist-таймером: замерев на нулевом окне, отправитель периодически шлет крошечные window probes — однобайтовые зонды, вынуждающие получателя повторно объявить окно. Обновление может потеряться снова, но зонд будет послан еще — кольцо разрывается. Это классический пример того, как TCP латает дыры протокола таймерами: любой управляющий сегмент может пропасть, поэтому каждое «молчание» обязано иметь свой дозорный таймер.

Silly Window Syndrome и алгоритм Нейгла

Еще одна патология — «синдром глупого окна» (silly window syndrome, SWS). Быстрое приложение-приемник, забирающее данные по чайной ложке, объявляет крошечные окна, отправитель набивает сегменты одним-двумя байтами — и по сети идет лавина пакетов с 40-байтовыми заголовками и байтом полезной нагрузки: КПД падает до единиц процентов. Лечится синдром с двух сторон. Получатель (решение Дэвида Кларка) не объявляет окно, пока не сможет принять хотя бы сегмент или половину буфера, — лучше помолчать, чем открыть окно на байт. Отправитель (алгоритм Нейгла, RFC 896, январь 1984) копит мелкие порции данных, пока не наберется полный сегмент или не придет ACK на ранее посланное, — и склеивает их в один пакет. Джон Нейгл придумал алгоритм, борясь с коллапсом перегрузки в сети Ford Aerospace: telnet-сессии генерировали по пакету на нажатие клавиши, и сеть тонула в однобайтовых сегментах.

Ирония в том, что сам Нейгл позже признал: его алгоритм в паре с delayed ACK (получатель держит подтверждение до ~40 мс, надеясь piggyback'нуть его на ответные данные) создает злую связку — отправитель ждет ACK, чтобы послать накопленное, получатель ждет данных, чтобы послать ACK, результат — периодические 40–500-миллисекундные «залипания» интерактивных приложений. Поэтому в низколатентных системах (торговые терминалы, игры, SSH-подобные сессии) алгоритм Нейгла отключают флагом TCP_NODELAY — и сам автор рекомендует делать это везде, где интерактивность важнее экономии пакетов. Linux и BSD предлагают и обратный инструмент — TCP_CORK / TCP_NOPUSH: «не отправлять ничего, кроме полных сегментов, пока я не сниму пробку», удобный для пакетной рассылки заголовка и тела одним движением.

Масштабирование окна и «длинные толстые трубы»

Шестнадцатибитное поле окна — бомба замедленного действия, заложенная в 1981 году. Максимум 65 535 байт «в полете» означает предельную пропускную способность window/RTT: при RTT 100 мс — 5,2 Мбит/с, при спутниковом RTT 600 мс — меньше мегабита. Каналы росли быстрее, чем задержки падали, и к концу 1980-х возник класс «длинных толстых труб» (long fat networks, LFN) — каналов с огромным произведением пропускной

способности на задержку. Произведение это называется bandwidth-delay product (BDP) и показывает, сколько байт должно находиться в пути, чтобы заполнить канал целиком: 1 Гбит/с при 100 мс — это 12,5 мегабайт в полете, трансатлантический 10 Гбит/с при 80 мс — 100 мегабайт.

Ответом стали расширения TCP Extensions for High Performance: сперва RFC 1072 (1988), затем RFC 1323 (1992) Вэна Джейкобсона, Роберта Брэйдена и Дейва Бормана — они добавили опцию масштабирования окна (window scale — множитель до 2^{30}, окно растягивается до гигабайта), метки времени (точный RTTM и PAWS) и, как фундамент, — концепцию LFN. Именно эти расширения позволили TCP пережить переход Интернета в гигабитную эру: без window scaling современный Интернет физически не смог бы наполнить свои магистрали. Сегодня окна в десятки мегабайт — рутина, а стеки автоматически подстраивают размеры буферов под измеренный BDP (receive buffer autotuning). Обратная сторона — требование к памяти: «толстая труба» обязывает держать мегабайты непрочитанных данных в буферах обоих концов.

Контроль перегрузки: как TCP управляет Интернетом

Наследие 1988 года: Tahoe, Reno и четыре алгоритма

Сетевой уровень сознательно не управляет скоростью отправителей — у дейтаграммной сети нет ни состояния потоков, ни обратного канала к источникам. Поэтому контроль перегрузки живет в транспорте: каждый TCP-стек самостоятельно нащупывает свою долю общей полосы. Алгоритмы, которые для этого придумал Вэн Джейкобсон после коллапса 1986 года (и которые вошли в релиз 4.3BSD-Tahoe 1988 года), до сих пор — скелет любого контроля перегрузки.

Медленный старт (slow start): соединение начинает с окна в один сегмент (современный стандарт — начальные десять, RFC 6928) и на каждый пришедший ACK увеличивает окно еще на сегмент, то есть удваивает его за каждый RTT — экспоненциальный разгон, пока окно не упрется в порог ssthresh или не появится потеря. Название иронично: «медленным» старт казался на фоне стоп-энд-вейта. **Избегание перегрузки** (congestion avoidance): после порога окно растет линейно — прибавка примерно одного сегмента на RTT, осторожное приближение к границе, за которой начинаются потери. **Быстрая повторная передача**: три дублирующих ACK трактуются как потеря одного сегмента — переотправить немедленно, не дожидаясь RTO. **Быстрое восстановление** (добавлено в 4.3BSD-Reno, 1990): после быстрой ретрансмиссии окно не обнуляется, а урезается вдвое и продолжается работа — потеря одного пакета еще не означает коллапса сети. Комбинация Reno-поведения с SACK получила имя NewReno и была канонизирована в RFC 2581 (1999) и действующем RFC 5681 (2009).

AIMD: пила, которая приводит к справедливости

Суммарная динамика окна — аддитивный рост и мультипликативное снижение (AIMD) — рисует знаменитую «пилу»: cwnd медленно ползет вверх, при потере мгновенно падает вдвое, растет снова. Пропускная способность одного соединения в среднем составляет примерно 0,75·cwnd/RTT. Пила выглядит расточительством (канал периодически недогружен), но у нее есть глубокое свойство, доказанное в классической работе Чью и Джейна (1989): из всех пар «увеличить/уменьшить» только аддитивно-мультипликативная схема сходится к справедливому и эффективному разделу канала между конкурентами. Геометрически: вектор аддитивного роста направлен к диагонали справедливости под 45°, мультипликативное снижение проецирует систему обратно на линию эффективного использования — итерация за итерацией все потоки стягиваются к равным долям. Десятки тысяч независимых TCP-потоков делят Интернет без единого координатора — распределенный алгоритм, работающий потому, что каждая его часть эгоистично соблюдает AIMD. Это, вероятно, самый масштабный пример самоорганизующейся системы управления в истории техники.

Формула Мэтиса и цена потерь

Поведение этой пилы описывается формулой Мэтиса и соавторов (1997): пропускная способность соединения приблизительно равна MSS / (RTT·√p), где p — вероятность потери. Следствия безжалостны: скорость падает обратно пропорционально корню из потерь (потери в 1% режут скорость в десять раз относительно потерь в 0,01%) и обратно пропорционально RTT. Отсюда — все беды дальних соединений: спутниковые и трансокеанские каналы с высоким RTT и малейшими потерями дают мизерную скорость даже на толстой трубе, отсюда же — стимул приближать контент к пользователям (CDN режут RTT) и беречься от потерь (буферы, FEC). Формула также объясняет, почему «одна потеря на гигабит» для loss-based алгоритма — катастрофа: окно восстанавливается до прежних значений минуты.

Нечестность и войны алгоритмов

Справедливость AIMD имеет оговорки. Во-первых, RTT-несправедливость: соединения с коротким RTT получают ACK чаще и растут быстрее — в общем канале ближний поток отъедает больше, чем дальний. Во-вторых, несовместимость разных алгоритмов: потоки, агрессивнее реагирующие на полосу (или вообще не реагирующие на потери), вытесняют классический AIMD. Обе проблемы — не баги, а свойства децентрализованной игры без правил: «справедливость» в Интернете есть равновесие, а не гарантия.

История знает попытку сыграть по другим правилам: TCP Vegas (Бракмо и Питерсон, 1995) ориентировался не на потери, а на рост задержки — увеличивающийся RTT означал, что очереди растут, и Vegas притормаживал заранее, не доводя до потерь. Алгоритм был элегантнее и дружелюбнее… и проиграл в конкурентной борьбе: в канале, который делят Vegas и loss-based Reno, второй занимает больше полосы (он не тормозит до потерь), и

оператору невыгодно включать Vegas в одиночку. Классическая дилемма заключенного в масштабе протокола: индивидуальная вежливость наказуема, пока вежливы не все. Vegas остался в истории, а его идеи возродились в BBR — при других стартовых условиях.

Эра гигабитных каналов породила семейство «высокоскоростных» алгоритмов. BIC (2004) и его преемник CUBIC (Ха, Ри и Сюй, 2008) растят окно не линейно, а кубической функцией времени, прошедшего с последней потери: быстрое возвращение к прежней скорости, плато в ее окрестности, осторожный зондаж выше. CUBIC с 2006 года (ядро 2.6.19) — алгоритм по умолчанию в Linux и фактический стандарт Интернета, в 2018 году он был канонизирован как RFC 8312. Microsoft в те же годы продвигала Compound TCP (сумма loss- и delay-составляющих) — он жил в Windows и аккуратно проигрывал CUBIC в смешанном трафике.

BBR: модель вместо потерь

В 2016 году Нил Кардуэлл и команда Google предложили радикальную альтернативу: BBR (Bottleneck Bandwidth and Round-trip propagation time). Вместо того чтобы реагировать на потери как на единственный сигнал, BBR строит модель канала: непрерывно оценивает пропускную способность узкого места (BtlBw) и минимальную «пропагационную» задержку (RTprop) и держит количество данных в полете около их произведения — того самого BDP. Идея: перегрузка наступает не когда пакеты начинают теряться, а когда in-flight превышает BDP, до этой точки можно ехать быстро, за ней — бессмысленно (данные стоят в очередях, скорость та же, задержка выше). BBR циклически зондирует обе величины (фаза ProbeBW перебирает коэффициенты скорости, фаза ProbeRTT периодически сливает окно до минимума, чтобы измерить незанятую задержку).

Результаты оказались впечатляющими: на трансатлантических каналах с фоновыми потерями BBR давал кратный выигрыш против CUBIC, Google развернула его на своем междугороднем каркасе и в YouTube еще до публикации. Но первая версия породила и конфликты: BBR игнорировал потери, «продавливая» свою оценку, и в общих бутылках с loss-based потоками вел себя агрессивно, а на буферах малого размера сам становился источником потерь. BBRv2 (2019) и BBRv3 (2023) добавили реакцию на потери и ECN, ограничения «верхней полки» — протокол повзрослел. Показательна и другая деталь: с 2021 года ядро Linux позволяет подгружать алгоритмы контроля перегрузки как eBPF-программы (BPF struct_ops) — исследователи и операторы экспериментируют с новыми алгоритмами без патчей ядра, контроль перегрузки из монолита превращается в плагин.

ECN и дата-центровый TCP

Альтернатива потерям как сигналу — явная маркировка: ECN (RFC 3168, 2001) позволяет маршрутизатору поставить бит «здесь тесно», а стеку TCP — сигнализировать партнеру флагами ECE/CWR и снизить окно без единой потерянной капли. В магистральном Интернете ECN приживался вяло (нужна поддержка всей цепочки), но в дата-центрах стал фундаментом: алгоритм DCTCP (Ализаде и соавторы, SIGCOMM 2010) считывает не факт маркировки, а ее долю и пропорционально уменьшает окно — при маркировке всего нескольких процентов очереди держатся крошечными, задержки микросекундными, а

пропускная способность почти полной. DCTCP показал, что транспорт может использовать сетевую телеметрию куда тоньше, чем «потеря/не потеря», — и задал направление, в котором развиваются L4S и ECN-режимы BBRv2/v3.

Беспроводные сети: когда потеря — не перегрузка

Фундаментальное допущение loss-based контроля — «потеря значит, сеть переполнена» — рушится в беспроводных средах: Wi-Fi и сотовые каналы теряют пакеты из-за помех, замираний и роуминга, при совершенно пустых очередях. TCP «пугается» и режет скорость в разы, хотя тормозить не перед кем. Решения делятся на классы: специальные алгоритмы, учитывающие природу потерь (Westwood+ оценивает доступную полосу по темпу ACK), прокси с разделением соединения (split-TCP, PEP — операторские шлюзы, терминирующие TCP ближе к абоненту и держащие радиосегмент под своим контролем), и переход на транспорты, которые меряют, а не угадывают (BBR, QUIC с настраиваемым контролем). Проблема стара как мобильный Интернет и не решена до конца — просто научились ее обходить.

Производительность: когда узким местом становится сам стек

Цена байта: системные вызовы, копии, прерывания

На скоростях 10–100 Гбит/с и RTT в единицы миллисекунд ограничение перемещается из сети в хост: процессор не успевает обрабатывать пакеты. Путь сегмента до приложения включает прерывание сетевой карты (или опрос в polling-режиме), подъем по стеку в ядре, несколько копирований данных между буферами, системные вызовы на прием и отправку. Каждый пакет стоит микросекунды CPU, на 100 Гбит/с при стандартном MSS это десятки миллионов пакетов в секунду — больше, чем способно обработать ядро общего назначения. Отсюда — целая индустрия оптимизаций конца 1990-х и далее.

Первый класс — перенос работы на железо: checksum offload (контрольная сумма считается сетевой картой), TSO/GSO (сегментация крупного блока на MSS-размерные пакеты откладывается до карты или до последнего момента в ядре), GRO/LRO (обратная операция на приеме — склейка мелких пакетов в крупные до подъема в стек), interrupt coalescing (одно прерывание на группу пришедших кадров вместо прерывания на каждый кадр). Второй класс — уменьшение копирований: sendfile() и splice() (Linux, конец девяностых) позволяют передать данные из файла в сокет, минуя пользовательское пространство, — статический контент и медиа разгружаются без лишних тактов, zero-copy-прием отдает приложению страницу ядра напрямую. Третий класс — обход ядра целиком: DPDK и родственные фреймворки забирают очередь сетевой карты в пользовательский процесс, который опрашивает ее в busy-poll цикле без прерываний и системных вызовов, пользовательские стеки (mTCP, Seastar, F-Stack) реализуют поверх этого собственный TCP. Цена — выделенное ядро CPU на опрос и потеря универсальности, выигрыш — десятки миллионов пакетов в секунду на одном сервере. Крайняя точка этой эволюции — RDMA (RoCE в дата-центрах): транспорт, работающий мимо и ядра, и CPU, перекладывающий доставку памяти от процесса к процессу на сетевую карту.

Проблема C10K и событийные API

Отдельная ось производительности — не байты, а соединения. Классическая модель «поток на соединение» упирается в пределы тысяч одновременных потоков, веб-серверу же нужны десятки и сотни тысяч. В 1999 году Дэн Кегель сформулировал «проблему C10K» — десять тысяч одновременных клиентов, — и вокруг нее выросла событийная модель: один поток обслуживает тысячи сокетов, переключаясь только туда, где есть готовность. select/poll (линейный опрос, O(n)) сменились kqueue (FreeBSD, 2000), epoll (Linux, 2002) и IOCP (Windows) — регистрами событий с O(1)-доступом, nginx (2004) доказал, что на такой архитектуре один сервер держит сотни тысяч соединений. Современное продолжение — io_uring (Linux 5.1, 2019, Йенс Аксбое): очередь заявок и очередь завершений в разделяемой памяти, позволяющая обмениваться с ядром пакетами операций почти без системных вызовов, — шаг в сторону того же kernel bypass, но без потери универсальности.

Инкарст и специфика дата-центров

Внутри ЦОД у транспорта своя патология — TCP incast: десятки и сотни серверов по запросу отвечают одному получателю одновременно, суммарный всплеск мгновенно переполняет буфер коммутатора, десятки пакетов теряются, десятки соединений синхронно уходят в backoff — полезная пропускная способность падает на порядок. Ответы: DCTCP с его микроскопическими очередями на ECN, приоритизация и точный контроль очередей, и радикальные альтернативы вроде Homa (Монтазери и соавторы, SIGCOMM 2018) — транспорта с приоритетом коротких сообщений и приемником, управляющим отправкой, — которые показывают: универсальный TCP не является оптимальным даже в условиях, где сеть идеально управляема. Дата-центровый транспорт — отдельная экосистема, живущая по другим законам, чем Интернет.

Сжатие заголовков

Для крупных пакетов 40 байт заголовков IP+TCP — незаметная доля, для телеметрии IoT и голосовых пакетов (полезная нагрузка 20–160 байт) — до половины кадра. Механизм сжатия заголовков (ROHC, RFC 3095, 2001) эксплуатирует избыточность потока: большинство полей между соседними пакетами меняется предсказуемо или не меняется вовсе, поэтому по каналу передаются дельты, а полный заголовок восстанавливается на приеме. В сотовых сетях ROHC обязателен для VoLTE — без него разговорный трафик не влезал бы в радиоэфир. В вебе ту же роль играет QPACK (сжатие HTTP-заголовков в HTTP/3) — заголовки прикладного уровня, а транспортные QUIC шифрует целиком.

Альтернативы и расширения TCP

SCTP: транспорт для телекома

SCTP (Stream Control Transmission Protocol, RFC 2960, 2000, действующая версия — RFC 4960, 2007) проектировался для телефонных сетей: перенос сигнализации SS7 поверх IP требовал надежного транспорта с функциями, которых не было у TCP. SCTP ориентирован

на сообщения (сохраняет границы сообщений), поддерживает мультистриминг (несколько независимых подпотоков в одном соединении — потеря в одном не блокирует другие: head-of-line blocking решен задолго до QUIC) и мультихоуминг (одно соединение с несколькими адресами на каждом конце — резервирование путей). Рукопожатие четырехэтапное, с cookie-механизмом защиты от флуда (в отличие от TCP, состояние на сервере до завершения рукопожатия не хранится). Судьба SCTP поучительна: в телекоме (DIAMETER, Sigtran) он прижился, в WebRTC канал данных работает поверх SCTP-поверх-DTLS-поверх-UDP, но общецелевым транспортом он не стал — промежуточное оборудование и NAT его не понимали, а экосистема уже была заточена под TCP и UDP. Урок выучил QUIC: строить надо поверх протокола, который middleboxes точно не трогают.

DCCP и другие эксперименты

DCCP (Datagram Congestion Control Protocol, RFC 4340, 2006) предлагал датаграммный транспорт со встроенным согласуемым контролем перегрузки (профили CCID: TCP-подобный и «уравнительный» TFRC — для потоков, которым нужна скорость «как у TCP», но без повторных передач). Идея была здравой, внедрение — почти нулевым: middleboxes задушили еще один новый транспорт, и этот урок позже усвоит QUIC — маскируйся под то, к чему сеть уже привыкла. Из той же серии — UDP-Lite. Все они остались в реестрах IANA как памятники главной проблеме транспорта: новый протокол бесполезен, пока его трафик не проходит через существующий Интернет.

MPTCP: несколько путей одновременно

MPTCP (Multipath TCP, архитектура RFC 6182, протокол RFC 8684, 2020) расширяет TCP до работы по нескольким путям сразу: одно соединение для приложения, несколько под-соединений (subflows) поверх разных интерфейсов и маршрутов, общая нумерация байтов и единый контроль перегрузки. Мотивация — мобильность: телефон с Wi-Fi и LTE одновременно может агрегировать полосу и бесшовно переживать уход одной из сетей. Apple включила MPTCP в iOS 7 (2013) для Siri, позже — для карт и ассистента, операторы (проксирование трафика) и ЦОДы используют его для балансировки. В общий Интернет MPTCP не вышел — опять middleboxes: оборудование, не знающее опций MPTCP, ломало рукопожатия, помог fallback-режим (при малейшей проблеме соединение деградирует до обычного TCP). MPTCP — важное свидетельство: даже расширение, стандартизованное IETF и реализованное гигантами, упирается в «иммунную систему» сети.

Почему TCP так трудно менять

Сложим наблюдения в диагноз. TCP живет в ядрах миллиардов устройств, обновления которых растянуты на годы и десятилетия (встроенные системы, IoT, старые ОС). Его заголовки видны промежуточному оборудованию, которое за сорок лет научилось их читать, модифицировать и блокировать «непонятное» — любая новая опция рискует быть отброшенной файрволом по дороге. Это явление назвали окостенением (ossification): сеть «костенеет» вокруг текущего профиля протоколов, и эволюция становится невозможной без маскировки. Попытки расширить TCP (новые опции, Accurate ECN, альтернативные

алгоритмы) продвигаются с трудом десятилетий, зато алгоритмы внутри существующего заголовка (CUBIC, BBR, RACK) меняются относительно свободно — их не видно снаружи. Вывод, который сделала индустрия к середине 2010-х: чтобы транспорт снова стал эволюционируемым, его нужно (а) перенести из ядра в пространство, где обновления измеряются неделями, (б) спрятать за уже неприкосновенным протоколом (UDP 443 проходит почти везде), и (в) зашифровать, чтобы middleboxes не могли окостенеть вокруг нового формата. Все три пункта — про QUIC.

QUIC: транспорт из пользовательского пространства

От эксперимента Google к стандарту IETF

История началась в 2012 году в Google: Джим Роскинд спроектировал экспериментальный протокол, призванный ускорить поиск и YouTube. Публично QUIC был анонсирован в 2013-м, развернут в Chrome и на серверах Google, в 2016-м IETF учредила рабочую группу для стандартизации, и в мае 2021 года вышли RFC 9000 (транспорт), RFC 9001 (интеграция TLS 1.3), RFC 9002 (обнаружение потерь и контроль перегрузки). В июне 2022-го стандартизирован HTTP/3 (RFC 9114) — прикладной профиль поверх QUIC. Путь от черновика до глобального стандарта с миллиардами пользователей занял меньше десятилетия — беспрецедентно быстро для транспорта, для сравнения: MPTCP от первой спецификации (2011) до итогового стандарта протокола (2020) шел около десяти лет — и все равно остался нишевым решением.

Архитектура QUIC: что именно перепроектировано

QUIC работает поверх UDP: для сети это обычные датаграммы на порт 443, для middleboxes — непрозрачный поток (а после окостенения вокруг QUIC — поток, который опасно трогать). Внутри — пересобранный транспорт.

Криптография встроена, а не надстроена. Рукопожатие TLS 1.3 совмещено с транспортным: 1-RTT на полное соединение с нуля и 0-RTT на повторное (данные уходят в первом пакете, ценой — риск replay-атак, поэтому 0-RTT разрешен только для идемпотентных операций). Практически весь заголовок QUIC зашифрован — наблюдатель видит минимум метаданных, TCP, где все флаги и окна открыты, на этом фоне выглядит протоколом из восьмидесятых, каковым и является.

Потоки вместо одного байтового потока. Соединение QUIC содержит множество логических потоков (streams), каждый — надежный и упорядоченный сам по себе, но независимый от остальных. Потеря сегмента блокирует только свой поток: head-of-line blocking на транспортном уровне устранен (внутри потока — нет, и это осознанная граница: приложениям, которым нужен порядок, он обеспечен, которым нет — не навязан). HTTP/3 кладет каждый запрос/ответ в свой поток.

Connection ID вместо четверки. Соединение идентифицируется случайным ID, а не четверкой адресов-портов: телефон, перешедший с Wi-Fi на LTE, сохраняет сессию (включая видео) без переподключения — connection migration из коробки. TCP так не умеет в принципе: смена адреса для него означает смерть соединения.

Номера пакетов вместо номеров последовательности. В TCP один счетчик обслуживает и порядок байтов, и идентификацию сегментов — отсюда вечная двусмысленность при повторных передачах (на какой именно сегмент пришел ACK?) и алгоритм Карна как заплатка. QUIC развел понятия: поток байтов нумеруется внутри потока, а каждый пакет получает уникальный, невозобновляемый номер — ретрансмиссия это новый пакет, подтверждения однозначны, оценка RTT точна без специальных оговорок.

Обнаружение потерь без RTO. RFC 9002 определяет потери по подтверждениям и времени (родственник RACK): таймер повторной передачи как первичный механизм не нужен — меньше ложных срабатываний и быстрее восстановление. Контроль перегрузки — плагин: базовая реализация loss-based (ньюрено-подобная), Google использует BBR, алгоритм можно менять, не трогая сеть.

Пользовательское пространство. QUIC живет в библиотеке приложения, а не в ядре: обновления выходят вместе с Chrome и nginx, а не с дистрибутивом ОС, A/B-эксперименты с алгоритмами — часть обычного цикла разработки. Цена — CPU: шифрование и обработка в user space дороже, чем в оптимизированном ядре (исследования 2022 года — «QUIC Is Not Quick Enough over Fast Internet» — показали, что на коротких RTT быстрых каналов TCP иногда быстрее, пока не появились GSO/GRO для QUIC и аппаратные офлоуды), и совместимость: часть сетей блокирует UDP, поэтому клиенты обязаны уметь откат на TCP (стратегия Happy Eyeballs для QUIC).

Внедрение в цифрах

Данные о развертывании QUIC публикуются с 2017 года, когда статья Лэнгли и соавторов (SIGCOMM) подвела первые итоги: задержка Google Search снизилась на 8% для десктопов и 3,6% для мобильных пользователей, повторные буферизации на YouTube — примерно на 15–18%, а на долю QUIC приходилось уже около 7% мирового интернет-трафика. К 2026 году картина иная: HTTP/3 поддерживает более 40% веб-сайтов (по измерениям W3Techs, октябрь 2026-го), доля HTTP/3 в мировых веб-запросах колеблется около 20%, а в трафике крупных CDN достигает 35–45%. QUIC перестал быть «протоколом Google»: его реализуют Cloudflare, Fastly, Akamai, Microsoft (MsQuic), Meta, его используют для API мобильные приложения по всему миру. Следующий шаг уже виден: MASQUE — стандартизация проксирования поверх HTTP/3 (на нем построен iCloud Private Relay), и Media over QUIC (MoQ) — попытка пересобрать live-стриминг на транспортных принципах QUIC.

Что QUIC меняет концептуально

Главный сдвиг даже не в производительности, а в модели эволюции. TCP обновлялся «всем Интернетом сразу» — через ядра, через middleboxes, через десятилетия, QUIC обновляется как приложение — библиотекой в одном процессе. Транспорт впервые стал software в полном смысле: с релизными циклами, флагами экспериментов и откатом. Оборотная сторона — фрагментация (у каждого приложения «свой TCP» со своими багами и настройками), рост стоимости CPU и потеря видимости для операторов: то, что middleboxes больше не могут вмешиваться, — одновременно и защита приватности, и конец целой индустрии оптимизации трафика. Спор о том, хорошо это или плохо, — по сути, спор о том, кому принадлежит сеть: конечным точкам или инфраструктуре между ними. Архитектура end-to-end, сформулированная в 1981 году, одержала запоздалую, но решительную победу.

Реальное время и удаленные вызовы

Почему реальное время отказывается от надежности

Требования живого трафика противоположны файловым: важны низкая задержка, стабильность джиттера и синхронизация, а полнота данных — вторична. Кадр видеозвонка, пришедший на 300 мс позже, не нужен никому — беседа ушла вперед, лучше потерять кадр и интерполировать, чем остановить поток ради повторной передачи. Отсюда — RTP (Real-time Transport Protocol, RFC 3550, 2003, изначально 1996), надстройка над UDP, добавляющая ровно то, чего не хватает датаграммам для медиа: номера последовательности (обнаружить потерю и переставить пришедшее вразнобой), метки времени (восстановить ритм воспроизведения и синхронизировать аудио с видео) и идентификаторы источников, управляющий протокол RTCP параллельно обменивается статистикой качества. Потерянный пакет RTP не переотправляется — получатель скрывает потерю (packet loss concealment): повторяет предыдущий фрагмент, интерполирует, вставляет тишину. Между RTP и приложением лежит jitter-буфер: он сглаживает неравномерность прихода пакетов ценой небольшой фиксированной задержки — баланс, который adaptive-буферы подстраивают на лету.

Вершина этой эволюции — WebRTC (развивается с 2011 года под эгидой W3C и IETF): стек, в котором медиа идет по SRTP поверх UDP, управляющие каналы — по DTLS, а произвольные данные — по SCTP-поверх-DTLS, ICE прокладывает путь сквозь NAT, а контроль перегрузки (GCC) реализован поверх — по задержкам и потерям. Звонки в браузере без плагинов стали возможны именно потому, что транспортный слой разрешили собрать заново из UDP-«конструктора». К той же семье относятся стриминговые протоколы нового поколения — SRT и RIST в вещании, MoQ для live-доставки поверх QUIC.

RPC: вызов функции через океан

Вторая большая прикладная абстракция транспорта — удаленный вызов процедур (RPC). Идея, предложенная еще в диссертации Брюса Нельсона (1981) и популяризованная Sun RPC (1984), на котором работает NFS: вызов result = remote_function() должен выглядеть как локальный, а транспорт, сериализация аргументов (XDR, позже protobuf и другие), ожидание ответа и повторные попытки — скрываться внутри. За элегантностью стоит уже знакомая нам проблема двух генералов: таймаут вызова принципиально неотличим от «запрос не дошел» и «запрос выполнен, ответ потерян». Повторный вызов может выполнить операцию дважды — дважды списать деньги. Поэтому зрелые RPC-системы вводят идентификаторы запросов и идемпотентность на стороне сервера (повторный запрос с тем же ID возвращает прежний результат, а не исполняется заново), а семантику честно документируют как at-least-once. Современный стандарт — gRPC (Google, 2015): protobuf-сериализация поверх HTTP/2 (мультиплексированные потоки, двунаправленный стриминг), с поддержкой HTTP/3. Микросервисная архитектура — это, по сути, индустриализация RPC, и ее главные головные боли (таймауты, ретраи, каскадные отказы) — прямые следствия фундаментальной ненадежности транспорта, на котором она стоит.

QoS против QoE: метрики и ощущения

Качество транспорта измеряют в двух системах координат. QoS (Quality of Service) — технические метрики: пропускная способность, задержка (RTT), джиттер, потери. QoE (Quality of Experience) — что чувствует пользователь: время открытия страницы, количество и длительность «залипаний» видео, разборчивость речи в звонке. Две системы не сводятся друг к другу: при формально хорошем QoS видео может лагать (адаптивный плеер неправильно оценил канал), а при плохом QoS пользователь может быть доволен (потери скрыты кодеком). Инженерная практика XX века оптимизировала QoS, индустрия потоковых сервисов XXI-го перешла на QoE — потому что именно частота зависаний (rebuffering), а не средний битрейт, предсказывает отток зрителей. Измерение и того и другого — самостоятельная наука: сеть нестационарна, данные шумны, а результат зависит от точки измерения (активные зонды искажают картину, пассивные не видят всей картины), RTT, измеренный ICMP-пингом, и RTT, который видит TCP-рукопожатие, могут различаться из-за приоритизации трафика.

Безопасность транспортного уровня

SYN-флуд и SYN-cookies

Рукопожатие — самая уязвимая фаза соединения: получив SYN, сервер выделяет состояние (буферы, запись в очереди полуоткрытых соединений) еще до того, как убедился, что клиент существует. SYN-флуд забивает очередь поддельными SYN'ами с несуществующих адресов — легитимные клиенты не могут соединиться. Атака стара как сам TCP (один из первых громких случаев — атака Митника 1995 года, ослепившая хост флудом перед подменой соединения). Контрмера — SYN-cookies (Дэниел Бернштейн и Эрик Шенк, середина 1990-х): при переполнении очереди сервер перестает хранить

состояние и кодирует его прямо в свой ISN — криптографический хэш от параметров соединения и секретного ключа, состояние «восстанавливается из пакета», когда приходит третий шаг рукопожатия. SYN-cookies вошли в Linux еще в 2.0 и включены по умолчанию, современные варианты (SYNPROXY) терминируют рукопожатие на файрволе, вообще не пуская флуд к серверу. SYN-флуд сегодня — рядовой инструмент DDoS-арсенала, нейтрализуемый на уровне инфраструктуры.

Предсказуемость и подделка: от Митника к RFC 5961

Атака предсказания номеров последовательности, прославившая Митника, закрыта рандомизацией ISN (RFC 1948), но родственный класс жив: «слепые» атаки в окно (blind in-window attacks) — инъекция данных, SYN или RST от имени одной из сторон атакующим, который не видит трафик, но угадал четверку и номер в пределах приемного окна. RFC 5961 (2010) ужесточил правила: RST вне точного окна игнорируется, на подозрительный — отвечают challenge ACK, данные в окне вне ожидаемого — отбрасываются. Полностью проблему решает только криптоография канала — TLS делает инъекцию бессмысленной (поддельные байты не пройдут проверку MAC), а QUIC закрывает и метаданные. Для управляющего трафика инфраструктуры (BGP-сессии) существует отдельный механизм — TCP-AO (RFC 5925, 2010), наследник TCP MD5 (RFC 2385, 1998): аутентификация каждого сегмента общим ключом.

Порты как поверхность атаки

Порты — не только адресация, но и уязвимость. Эфемерные порты исчерпываются на прокси и балансировщиках (десятки тысяч TIME-WAIT против ~28 тысяч портов на пару адресов), коллизии эфемерных портов позволили в свое время атаковать DNS, предсказывая исходный порт запроса, — в паре с угадыванием transaction ID это дало знаменитую атаку Каминского (2008), после которой DNS-резолверы рандомизировали и порты, и идентификаторы (рекомендации — RFC 6056). Сканирование портов — базовая разведка любого взлома. Отдельный реликт — привилегированные порты ниже 1024 в Unix: исторически занимать их мог только процесс с правами суперпользователя, и это служило механизмом доверия — например, rlogin принимал всерьез лишь соединения, исходящие с низкого порта, механизм давно стал анахронизмом, но по инерции живет в системах.

Транспорт без шифрования как норма прошлого

Стоит зафиксировать сдвиг, случившийся на наших глазах. До середины 2010-х транспортные метаданные — кто, куда, какого размера пакетами, с какой скоростью — были открыты всем наблюдателям по пути, на этом строились и оптимизация (прокси, кэши, PEP), и слежка (анализ трафика выявляет сервисы даже внутри шифрованного TLS, по размерам и ритму пакетов). TLS 1.3 (2018) спрятал рукопожатие, QUIC — почти весь транспортный заголовок. Сеть теряет наблюдаемость, конечные точки получают приватность, операторы отвечают попытками классифицировать трафик по косвенным признакам и блокировать «непонятное» (история с замедлением UDP-трафика в

отдельных сетях — эпизод этой борьбы). Безопасность транспорта перестала быть надстройкой и стала его свойством — и это, возможно, самое важное изменение уровня за тридцать лет.

Путь одного соединения: синтез

Соберем все механизмы в одну историю — жизнь типичной HTTPS-сессии 2026 года, в которой пользователь открывает страницу.

Все начинается до транспорта: DNS (датаграмма UDP на порт 53 — короткая жизнь без соединений) возвращает адрес сервера. Клиент выбирает эфемерный порт (например, 52431 из диапазона 32768–60999) и формирует четверку — будущий идентификатор соединения. Первый пакет — SYN с ISN, сгенерированным как криптографический хэш четверки и секретного ключа, и опциями: MSS 1460, window scale (множитель 128 — базовых 64 КБ на гигабитных каналах давно мало), SACK permitted, timestamps. Сервер отвечает SYN-ACK со своим ISN, клиент шлет ACK — три пакета, один RTT, соединение ESTABLISHED. Если включен TCP Fast Open и cookie на руках — данные пойдут уже в первом SYN. Поверх немедленно стартует TLS 1.3 (еще один RTT) — и только потом первый байт HTTP-запроса.

Передача данных — работа двух окон. Поток режется на сегменты по MSS, каждый уходит как IP-пакет (при необходимости — через TUN/туннели провайдера, где MSS ужат clamping'ом), сетевая карта считает контрольные суммы и склеивает сегменты (TSO). Медленный старт удваивает окно каждый RTT — первые 10 сегментов (IW10) уходят сразу, окно упирается в ssthresh, рост становится линейным: начинается фаза избегания перегрузки, «пилы» AIMD. Получатель подтверждает байты кумулятивно, откладывая ACK до 40 мс в надежде piggyback'а, приложение-получатель объявляет окно — flow control страхует от переполнения его буферов. Где-то в середине загрузки маршрутизатор провайдера дропает сегмент: получатель трижды подтверждает прежнюю границу — fast retransmit отправляет копию, не дожидаясь RTO (который все равно адаптирован под измеренный SRTT + 4·RTTVAR), SACK сообщает, что «хвост» за дырой уже принят, — переотправлен только пропавший сегмент, fast recovery режет cwnd вдвое и продолжает работу. Если сервер — за CDN с BBR на борту, потерю он перенесет иначе: алгоритм сверит in-flight со своей оценкой BDP и скорректирует скорость по модели, а не по факту дропа.

Запрос обслуживается, страница получена. Клиент вызывает close() — уходит FIN, сервер, отдав последние байты, отвечает ACK, затем своим FIN, клиент подтверждает и входит в TIME-WAIT на 60 секунд — карантин против заблудших сегментов. На сервере, закрывшемся пассивно, состояние CLOSE-WAIT продлится ровно столько, сколько приложение не вызовет close(). Через минуту клиентская четверка свободна для новой жизни — а в логах провайдера от соединения остались несколько тысяч пакетов, каждый из которых прошел через всю машинерию, описанную в этой статье: номера последовательности и кумулятивные подтверждения, два окна и «пилу», таймеры и зонды, офлоуды сетевой карты и таблицу состояний. Если же и клиент, и сервер понимают

HTTP/3 — всей этой сцены не было вовсе: вместо нее — одно соединение QUIC, чье TLS-рукопожатие заняло один (а повторное — ноль) RTT, чьи потоки не блокируют друг друга, и которое переживет переход клиента с Wi-Fi на сотовую сеть, не заметив смены IP-адреса.

Итоги

Транспортный уровень — место, где Интернет становится пригодным для приложений, и место, где его фундаментальные противоречия проявляются острее всего.

Надежность собирается из ненадежности — и упирается в теоретический предел. Скользящее окно, кумулятивные подтверждения, адаптивные таймеры, выборочный повтор — из этих механизмов TCP строит гарантированный байтовый поток поверх почты, которая теряет письма. Но проблема двух генералов напоминает: абсолютная недостижима, приложения обречены на идемпотентность и «хотя бы раз» — и зрелые системы проектируются с этим знанием, а не вопреки ему.

Минимализм UDP оказался стратегией. Восьмибайтовый протокол, «ничего не гарантирующий», стал платформой эпохи: RTP и WebRTC, DTLS и, главное, QUIC — все построено из UDP-конструктора. Когда транспорт застыл в ядрах и middleboxes, единственным свободным слоем оказался самый простой.

Контроль перегрузки — это и есть управление Интернетом. AIMD Джейкобсона превратил миллиарды эгоистичных потоков в самоорганизующуюся систему с доказуемой сходимостью к справедливости, формула Мэтиса — ее паспорт: скорость упирается в корень из потерь и обратно пропорциональна RTT. История алгоритмов — Tahoe, Reno, Vegas, CUBIC, BBR — это история поиска сигнала о перегрузке: от потерь к задержкам, от задержек к измеряемой модели канала.

Практика уровня полна ловушек, которые понимаются только из устройства. TIME-WAIT — иммунитет, а не болезнь, CLOSE-WAIT — баг приложения, а не сети, «липкие пакеты» — не баг TCP, а обязанность прикладного протокола делать framing, Nagle с delayed ACK — известная связка, ломающая интерактивность (лечится TCP_NODELAY), производительность — это BDP против размеров буферов и окон, а на современных скоростях — еще и цена системных вызовов, копий и прерываний (от epoll и io_uring до DPDK и RDMA).

Окостенение победило TCP — и породило QUIC. Протокол, сорок лет служивший становым хребтом Интернета, утратил способность меняться: ядра обновляются десятилетиями, middleboxes блокируют незнакомое, опции не разворачиваются. Ответ — перенос транспорта в пользовательское пространство, поверх неприкосновенного UDP, под полной криптографией: QUIC обновляется как библиотека, мигрирует между сетями по connection ID, устраняет межпоточную блокировку и уже поддерживает более 40% сайтов. Транспорт впервые стал обычным software — со всеми выгодами быстрых релизов и всеми рисками фрагментации.

Безопасность переехала внутрь транспорта. От SYN-cookies и рандомизации ISN, родившихся из атак девяностых, — к TLS 1.3 и шифрованным заголовкам QUIC: уровень, который полвека был прозрачным по проекту, стал непрозрачным по необходимости. Приватность против наблюдаемости — новый водораздел, вдоль которого будет идти следующая эволюция сетей.

Итоговая картина уровня такова: TCP — адаптивная система управления, которая наблюдает сеть через ACK и потери, принимает решения об окнах и ретрансмиссиях и непрерывно корректирует собственное поведение, UDP — минимальный каркас, из которого каждое приложение строит транспорт под себя, а QUIC — доказательство, что и сорокалетний стандарт можно пересобрать заново, если спрятать его там, где сеть уже не достанет. Следующий шаг вверх — уровень приложений: протоколы, ради которых вся эта машинерия существует. HTTP и DNS, почта и стриминг, CDN и P2P — все они говорят на портах и сокетах, в окнах и сегментах, описанных здесь, и, как гласит двенадцатая «истина сетевого проектирования» из первоапрельского RFC 1925 (восходящая к Сент-Экзюпери): совершенство достигается не тогда, когда уже нечего добавить, а тогда, когда уже нечего убрать. Транспортный уровень — лучшее в Интернете доказательство этой максимы: и TCP 1981 года, и UDP из восьми байт, и QUIC, убравший из поля зрения все, что можно было убрать.