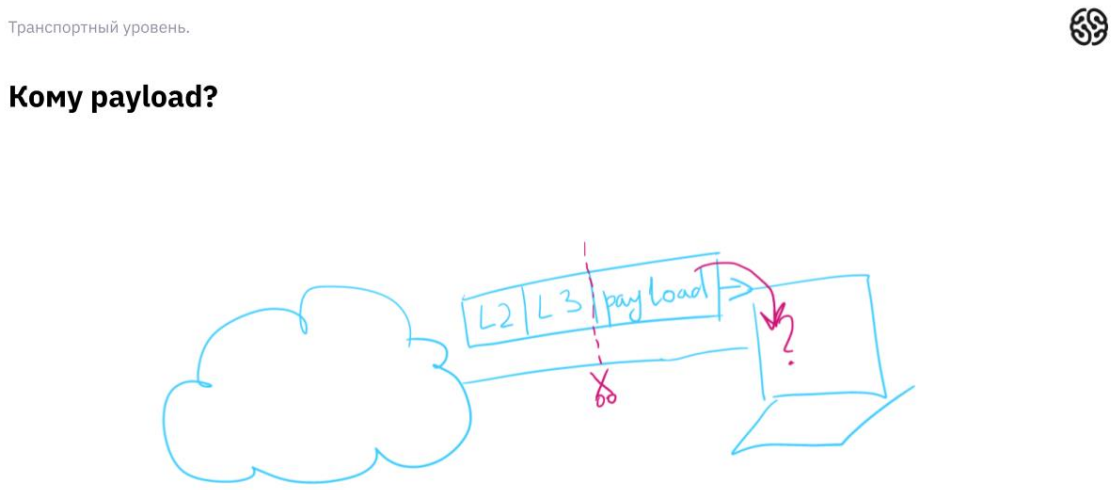
Лекция 4.

Добрый день, уважаемые студенты. Мы с вами уже на 4ой лекции по компьютерным сетям и давайте освежим в памяти, что было на прошлом уроке и на какой проблеме мы остановились.м

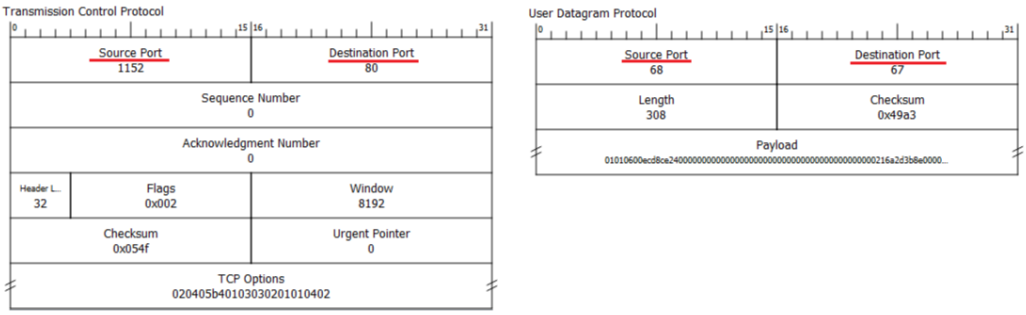


Мы научились строить IP-сети и передавать информацию от любого хоста в сети до любого другого хоста в сети. Правильно настроенные сетевые параметры хоста и маршрутизация на роутерах позволяют нам это делать. Но когда пакет с заголовками L2 и L3 доходит до получателя, он от них избавляется. И проблема в том, как хосту понять какому приложению передать дальше эти данные.

Для этого в стеке TCP/IP существуют два протокола TCP и UDP. О них мы сегодня и поговорим, также узнаем как TCP отвечает за гарантированную доставку данных в сети, где данные могут теряться, и как он же отвечает за скорость передачи этих данных. И в конце поговорим про сокеты.

Транспортный уровень.

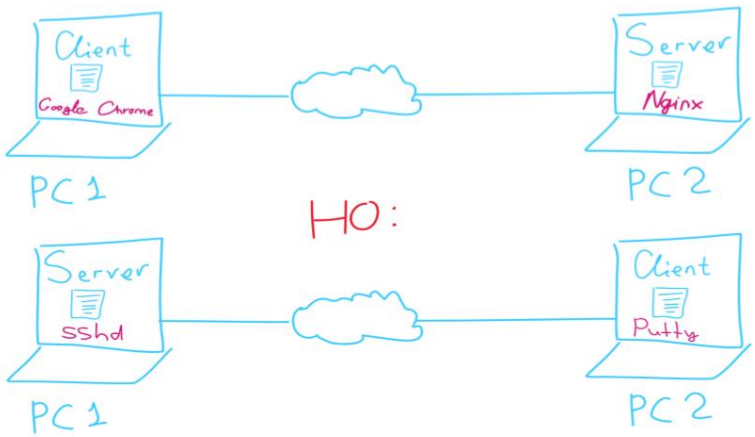
Заголовки TCP и UDP.



У каждого из них есть свои разные заголовки, но в них есть общие поля Port Source и Port Destination, которые занимают по 16 бит, и, следовательно, могут принимать значения от 0 до 65535. Этот еще один слой адресации в пакете и позволяет ОС определять какому приложению дальше передать payload. Не стоит путать физический порт компьютера или коммутатора с L4 портом.

Транспортный уровень.

Клиент-Сервер.

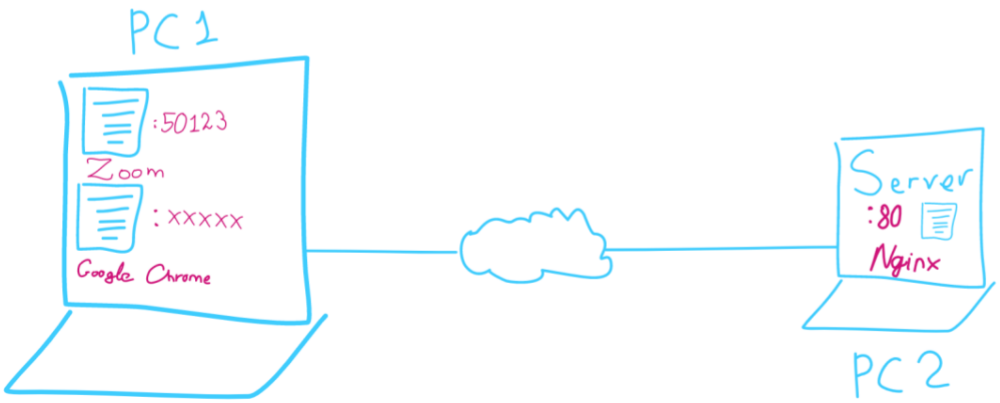


Давайте рассмотрим как эту проблему решает более простой протокол UDP, он же User Datagram Protocol. Представим, что у нас есть два хоста, между которыми есть сетевая связность. Всегда кто-то из этих двух будет

инициатором начала связи - он обычно называется клиентом, а тот с кем он связывается называется сервером. Обратите внимание, здесь не подразумевается железный физический сервер, а именно сервер - как программа, которая запущена на хосте и которая обрабатывает запросы клиентов. Например, если у нас с хоста 1 идет запрос на получение веб-страницы с хоста 2, тогда в рамках этого запроса, браузер на хосте 1 будет клиентом, а веб-сервер на хосте 2 сервером. Но, при этом допустим, если у нас админ захочет подключиться с хоста 2 с на хост 1 по SSH, то в таком контексте командная строка на хосте 2 будет клиентом, а процесс, обрабатывающий SSH подключения на хосте 1, будет сервером.

Транспортный уровень.

Порты.



ОС выдает приложению-клиенту каждый раз произвольный номер порта, который освобождается по завершению работы приложения. Т.е. допустим, мы открыли Zoom, Windows дала ему порт 50123. Если мы закроем Zoom и откроем Google Chrome, то Windows снова может дать для Chrome порт 50123. А если мы запустим снова Zoom, то ему может уже выделиться порт 50124.

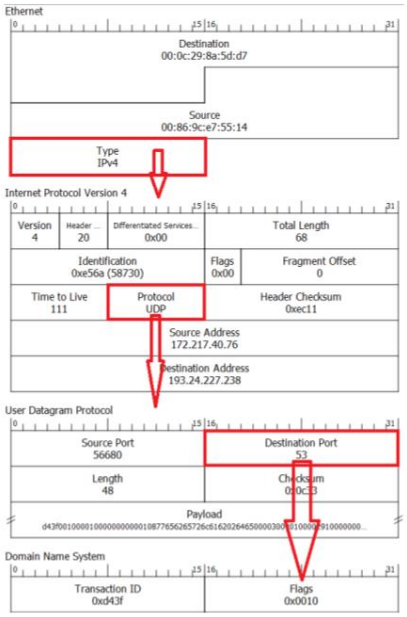
Но! Приложению-серверу ОС выдает постоянный номер порта. Т.е если мы запустим веб-сервер Nginx, ОС выдаст порт, который указан в настройках сервера, например 80. Если мы завершим работу Nginx и запустим снова, то ОС снова выдаст порт 80. При этом если мы завершим работу Nginx и запустим другой веб-сервер, например Apache, тоже на 80-

ом порту, то при запуске Nginx, будет выдана ошибка, что 80-й порт занят другим приложением и проложение не запуститься.

Недаром я указал порт клиента как 50124 а порт сервера как 80. Дело в том, что самые популярные протоколы, такие как SSH, FTP, HTTP и т.д., стандартизированы в диапазон от 0 до 1024 и называются системными портами (или так называемые “низкие порты”). Но это больше договоренность, вам ничто не мешает запустить свой веб-сервер на порту 8888 вместо 80, главное чтобы клиенты, которые подключались к нему знали об этом.

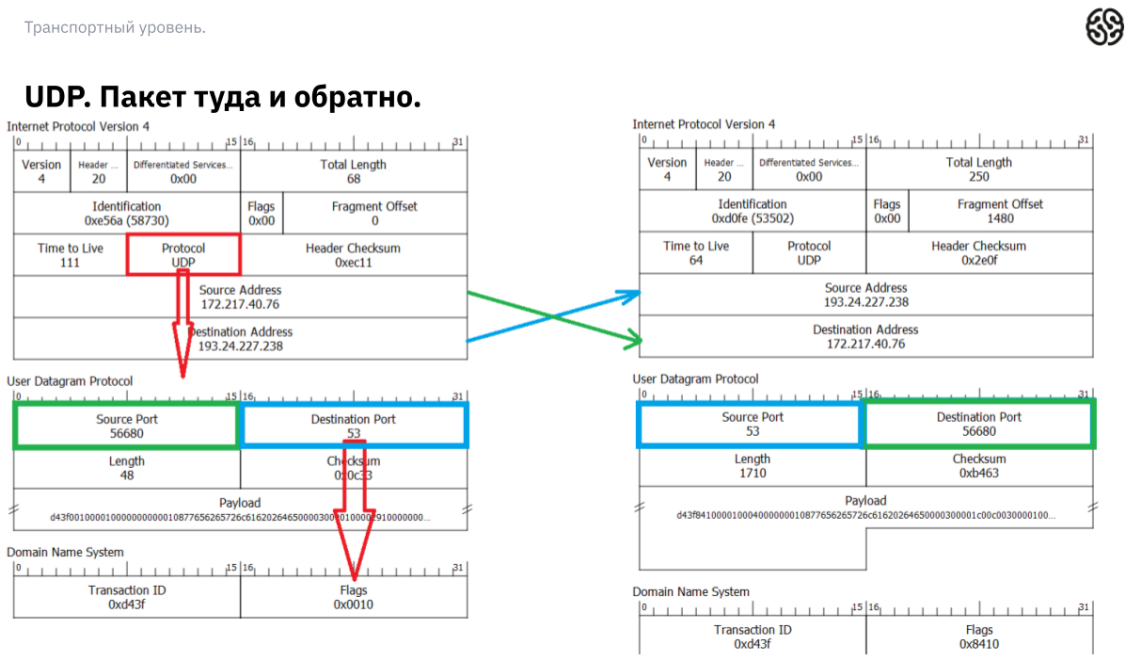
Транспортный уровень.

UDP.



Итак, допустим у нас поступил запрос с хоста 1 на хост 2, на котором крутится DNS сервер. DNS работает поверх протокола UDP и позволяет сопоставлять доменные имена, типа vk.com, в конкретные IP адреса, о нем более подробно мы будем говорить на следующих уроках. Посмотрим внимательно на заголовок, который у нас придет на хост 2. Есть правильный L2, есть правильный L3, все хорошо, в L3 заголовке у как следующий протокол указан UDP, в котором есть Destination Port . По этому значению ОС должна найти программу, которая “слушает” на этом порту все входящие пакеты, т.е. программу, которая обработает payload из такого пакета. Если такая программа не запущена, то пакет просто уничтожается. Если есть, то отбрасывается L4 заголовок и данные передаются уже самой программе. Помните, мы говорили и том, что в

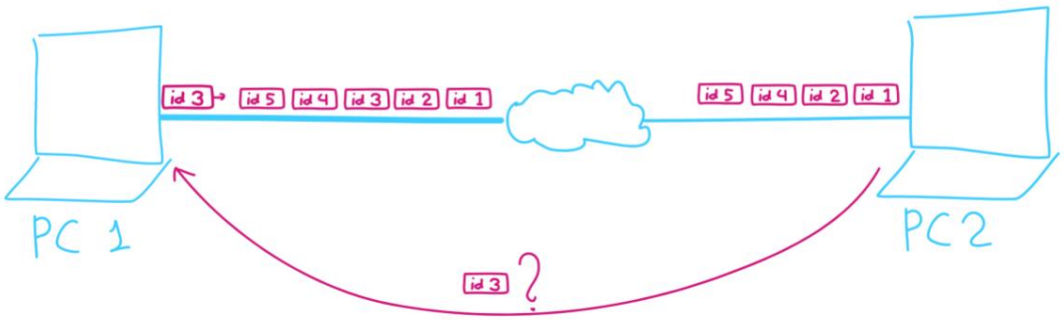
заголовке каждого уровня должен быть указан следующий протокол, чтобы обрабатывать пакеты можно было эффективно. Так вот, в этом плане Destination Port выполняет похожую роль в заголовке L4.



Программа, получив информацию, может ответить обратно, тогда создается L4 заголовок, в котором уже Destination Port будет с номером порта, с которого пришел запрос. Поэтому, когда клиент получит ответный пакет, он также без проблем поймет, какому приложению передать данные.

Помимо Destination Port и Source Port в UDP есть поле Length - это длина Payload’a + длина UDP заголовка. А также есть поле Checksum - в нём лежит значение, которое заполняет отправитель - это значение хэша, которое защищает UDP+Payload от случайных изменений, как и FCS в L2. Если какие-либо данные случайно изменятся, тогда хэш отправителя не совпадет с хэшем, который подсчитает получатель.

Повторная отправка.

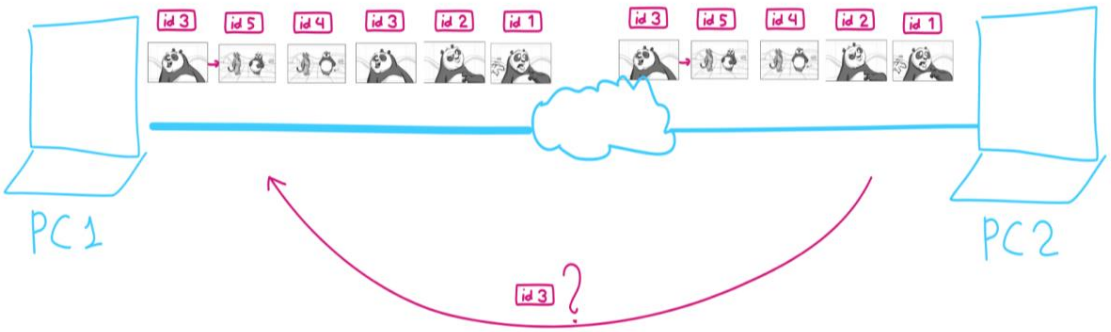


Как видим из заголовка L4, UDP достаточно прост. Но из-за такой простоты он не может гарантированно доставлять пакеты. Я напоминаю, что в сетях пакеты могут теряться, и это надо учитывать. Если мы будем использовать протокол UDP для доставки важных данных, то эти важные данные могут безвозвратно потеряться. Ведь ни сервер, ни клиент, не контролируют и не проверяют доставку пакетов. Для гарантированной доставки данных используется протокол TCP. Если упростить, то в TCP у нас используется нумерация пакетов и мы видим что у нас пакеты id1, id2, id3, id4, id5 вылетели от отправителя, а пришли пакеты id1, id2, id4, id5, и получатель понимает, что третьего пакета нет. Получатель запросит его снова и третий пакет придет после 5ого. В UDP отправитель же просто шлет данные и надеется, что они все долетят до получателя.

Теперь и вы понимаете все эти шутки про TCP и UDP)

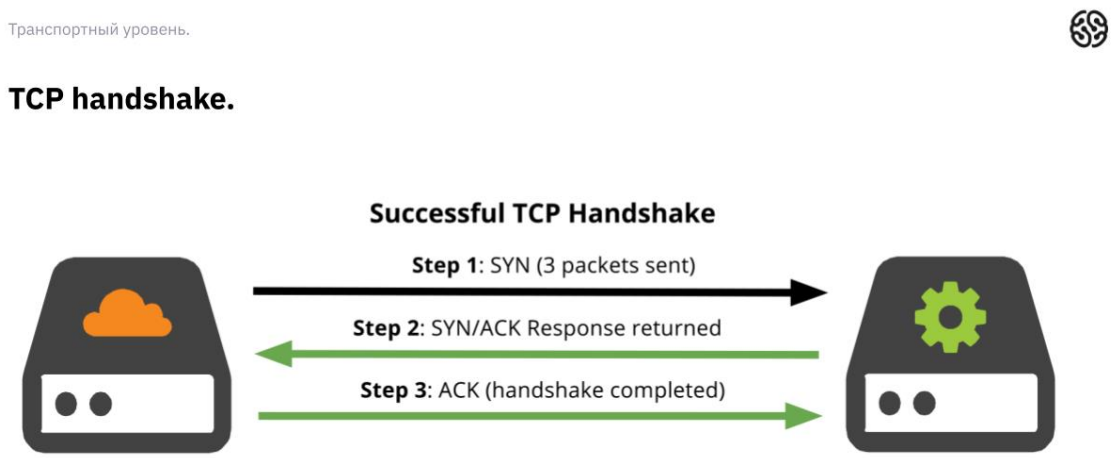
И вот на этом слайде можно видеть в чём принципиальная их разница. То что UDP не может доставлять гарантированно информацию - не означает что протокол плохой.

Повторный кадр с id3 здесь не нужен. Поэтому используем UDP.



Протокол UDP лучше всего использовать для доставки каких-то неважных данных, но которым нужна оперативность. То есть там, где не нужны заново отправленные пакеты, так как они быстро теряют актуальность. Например рассмотрим видеострим, который создает тот же zoom. Если при посылке кадров видео, у нас потеряется 1 кадр, то запрашивать его заново смысла нет. Иначе если мы начнем возвращать потерянный

видеокадр, то он у нас встроится в уже новую сцену, где будет лишним. У нас выскочит картинка из прошлого. Также UDP используется для пересылки телеметрии в онлайн играх например, где опять-таки позиционирование персонажа очень быстро меняется и если мы потеряем какие-либо координаты, то восстанавливать их нет смысла, персонаж уже будет в другой локации. Но такая доставка данных, например, для банковских платежей будет неприемлема, согласитесь. Под каждую задачу нужен свой инструмент. И тут нас выручает протокол TCP, который осуществляет гарантированную доставку данных.



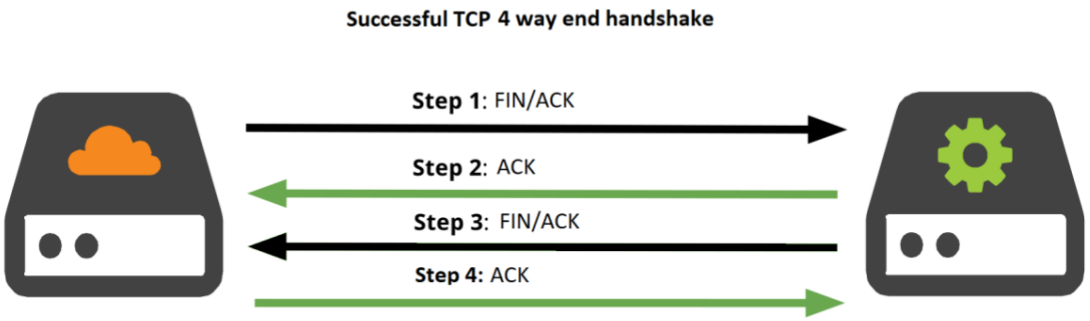
Давайте более подробно поговорим про протокол TCP. Прежде, чем начать передавать данные, клиенту надо убедиться что сервер доступен, и данные, которые он отправляет, приходят на сервер. И серверу надо убедиться что данные, которые он отправляет, приходят к клиенту. Чтобы проверить эту двустороннюю доступность устанавливается TCP-сессия.

В самом начале, клиент посылает специальный SYN-пакет, где в заголовке TCP есть специальный бит SYN (Synchronization), который он устанавливает в “1”. Сервер, получив SYN-пакет, должен отправить в ответ пакет с флагом ACK (Acknowledgment - подтверждение). При этом серверу также надо убедиться что его пакеты доходят до клиента, поэтому он также устанавливает флаг SYN. Клиент получив такой пакет посылает в ответ еще они пакет, с одним флагом ACK. После этого TCP-сессия

считается установленной, и данные могут передаваться между приложениями в рамках этой сессии.

Транспортный уровень.

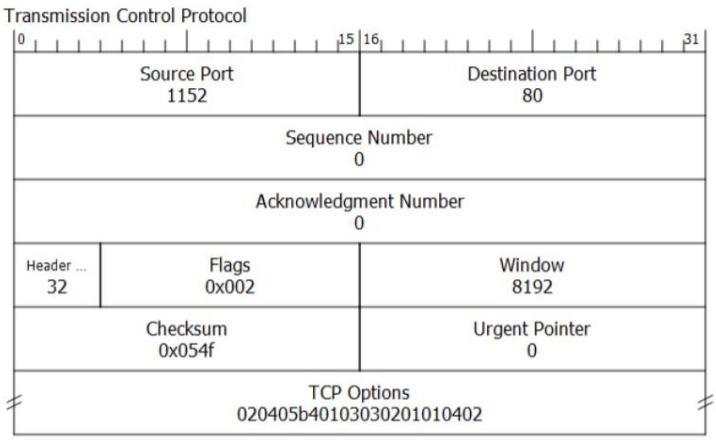
TCP 4-way handshake.



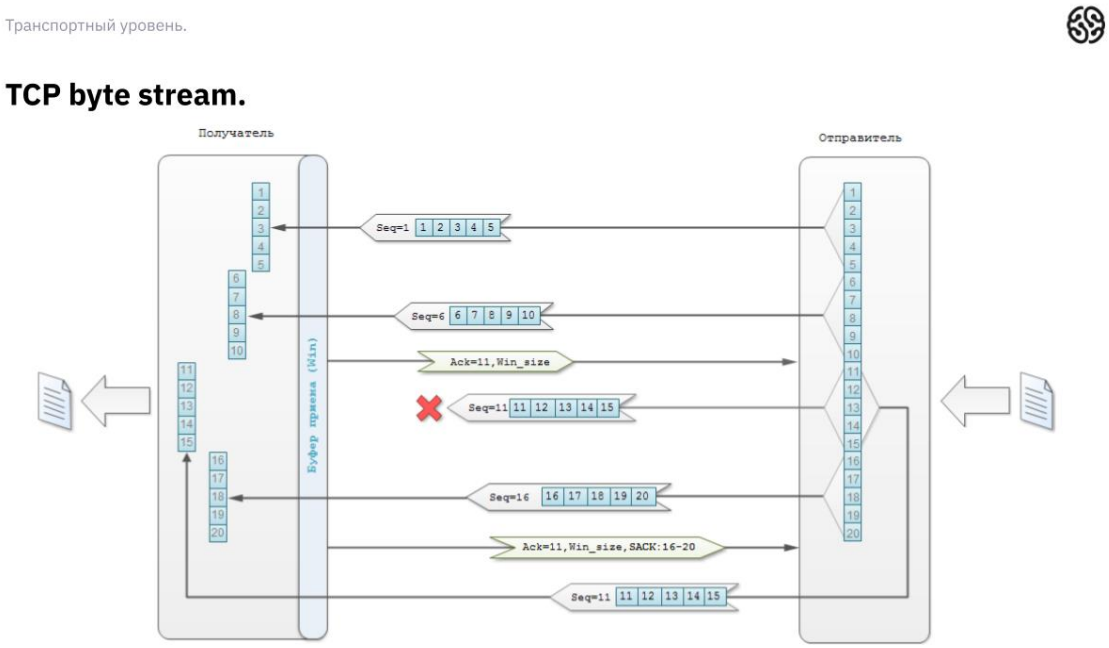
Закрывается сессия корректно уже в 4 шага и называется 4-этапным рукопожатием. Инициатор закрытия посылает пакет с флагами FIN/ACK и ждет на него ACK. Вторая сторона после этого также посылает флаги FIN/ACK и получает на него ACK. После этого сессия считается завершенной.

Транспортный уровень.

TCP header.



Как видим заголовок TCP уже более сложный, чем у UDP. Не будем сразу расписывать что к чему, обратим внимание что все так же присутствуют порты, появились поля Flags, а также поля Sequence и Acknowledgment number, о которых сейчас и поговорим.

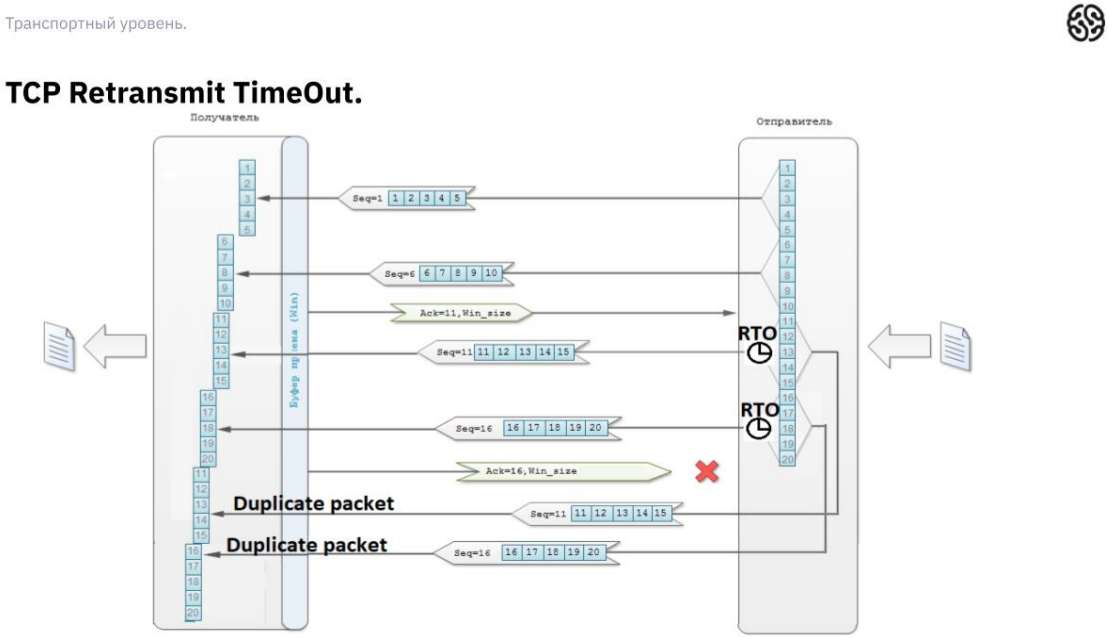


Давайте теперь рассмотрим то, как в TCP работает механизм гарантированной доставки. Прежде всего надо понимать, что данные, подлежащие отправке, на уровне TCP представляются потоком байт, где каждый байт последовательно пронумерован. TCP делит этот поток на части. Каждая часть инкапсулируется в пакет с L3 заголовком. Как я и говорил выше, каждому пакету нужен ID - уникальный номер пакета. И по нумерации пакетов, мы можем понимать, были ли пропущены какие-то пакеты во время передачи данных или нет. Этот id называется Sequence number и вставляется в TCP заголовок пакета, а сам id представляет из себя номер первого байта в пакете. Например, мы видим на слайде первые 5 байт передаются с Seq=1, затем вторые 5 байт передаются с Seq=6. Третьи 5 байт передаются с Seq=11, и т. д.

Принятые пакеты получатель собирает в поток байт и посылает в ответ отправителю Acknowledgment number, который равен количеству полученных байтов в пакете +1. То есть по сути, это ожидаемый следующий Seq номер пакета от отправителя. Acknowledgment number может посылаться в ответ не на каждый пакет, а сразу на серию пакетов.

Если теряется один пакет с данными, то по Seq номеру следующего пакета, который равен 16, получатель понимает, что пакета с ожидаемым Seq=11 не было и в ответном пакете оставляет Ack=11. У отправителя происходит “Duplicate Ack”, т.к. у него два Ack пакет с одним и тем же номером. И по умолчанию начинает пересылать пакеты начиная с Seq=11, т.е. и Seq=11 и Seq=16. Это не совсем эффективно, поэтому в TCP есть опция Selective ACK или SACK, то есть выборочное подтверждение, если её поддерживают и отправитель и получатель, то это позволяет в ACK пакете указывать выборочно подтверждать полученные пакеты. Например в нашем случае когда придет Seq=11 без Seq=16, получатель ответит Ack=11+SACK: 16-20 и отправитель поймет, что переслать необходимо только один пакет с Seq=11.

Помимо SACK есть и другие дополнительные опции. Вообще опции нужны, чтобы расширять возможности TCP. В ранних реализациях TCP работал проще, но со временем предлагались дополнительные функции, которые и объявляются в опциях. Об опциях и клиент и сервер договариваются во время установления сессии (когда происходит обмен SYN-SYN,ACK-ACK), они идут сразу после заголовка TCP.



Есть второй вариант потери, когда теряются ответные Ack’и. В таком случае отправитель не знает и не может узнать были ли доставлены пакеты Seq=11 и Seq=16. На этот случай при посылке пакетов

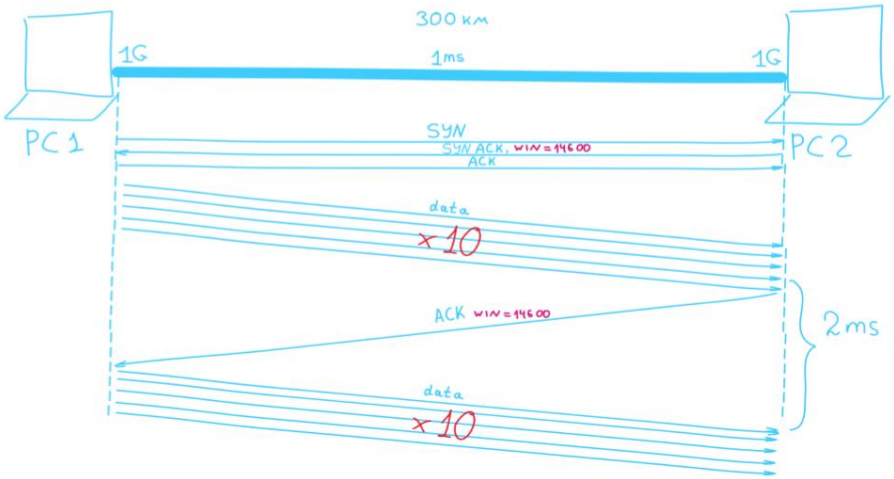
отправителем запускается таймер RTO - Retransmit TimeOut. По его истечению пакеты пересылаются снова (а сама передача приостанавливается). Seq=11 у получателя встает на свое место, а Seq=16 просто повторяется, Ack=21 посылается отправителю и он продолжает передачу.

При этом важно отметить, что у нас в пакетах в обе стороны передаются и Seq и Ack, а значит и сервер и клиент могут и принимать данные и передавать в рамках одной TCP сессии. И здесь слова “отправитель” и “получатель” не надо путать с “сервером” и “клиентом”. Здесь и далее мы специально это упростим, чтобы понять, как это работает в одну сторону.

Протокол TCP также ответствен за скорость передачи данных между отправителем и получателем. Для того чтобы понять, как это реализовывается, рассмотрим упрощенный случай сферического коня в вакууме . Предположим, что у нас есть два компьютера соединенных между собой гигабитным оптическим линком на расстоянии 300км. Как мы помним из школы, скорость света равна 300 000 км/с. Получается, что если с компа A вылетит пакет компу B, он долетит за 1мс (300000/300), ответ о подтверждении получения пакета вернется еще за 1 мс. 2 мс - это так называемая “круговая задержка пакета” - Round Trip Time - RTT. Если мы будем ждать ответа на каждый отправленный пакет, то в итоге пропускная способность будет равна 1460 байт * (1с/2мс) = 730 000 байт в секунду, или, умножая на 8, получим 5 840 000 бит/с = +- 6 Мбит/с, что довольно мало для гигабитного соединения. Почему 1460? Потому что мы помним что MTU = 1500 байтам, минус 20 байт на заголовок TCP и минус 20 байт на заголовок IP. Это так называемый Maximum Segment Size - MSS. MSS кстати тоже объявляется в TCP опциях и может быть специально уменьшен или увеличен, но стандартный MSS именно такой.

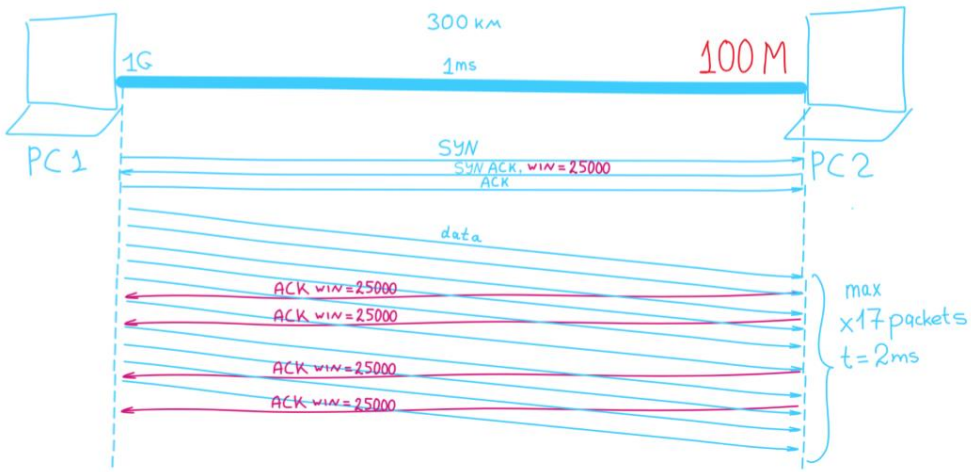
Согласитесь, если бы так работал TCP было бы не очень эффективно, поэтому в TCP используются различные ухищрения, чтобы эффективно передавать данные.

TCP RWND.



Ухищрение №1 - Receive Window (RWND), оно же поле Window в заголовке TCP. Оно позволяет слать пакеты отправителем без подтверждения. Измеряется в байтах, и означает, сколько байт может отправить отправитель без подтверждения, то есть с момента получения последнего ACK. Получается, если в нашем случае, при установке соединения, когда компы обменяются SYN-SYN,ACK-ACK, в последнем ACK будет окно 14600 байт, то отправитель может послать 10 пакетов без подтверждения. Итого, у нас уже скорость будет 10* 1460 байт * (1с/2мс) = 7 300 000 байт в секунду, или, умножая на 8, получим 58 400 000 бит/с = +-60 Мбит/с, что уже лучше. Если окно будет 29200 байт, то скорость будет порядка 120 МБит/с, и так далее. В данном случае у нас данные шлются пачками что не очень хорошо. Получателю необходимо чаще слать АСКи чтобы не было пустых мест и простоя в сети!

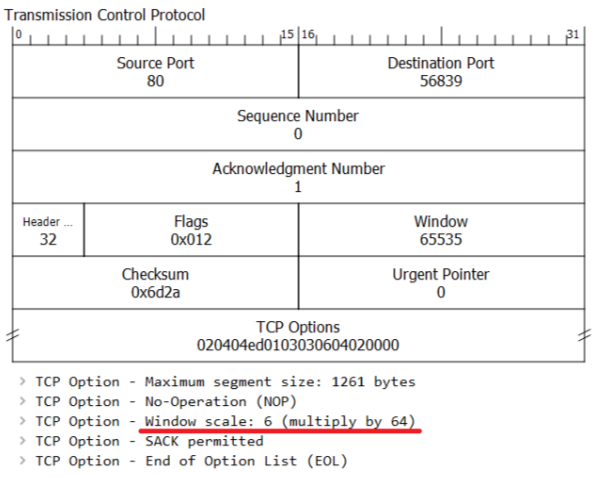
TCP RWND.



А ещё с момента получения последнего АСКа (и окна в нем), отправитель заново начинает отсчёт количество посланных данных без подтверждения. Важно понять, что чем больше окно, тем с большей частотой, а значит и скоростью, шлет данные отправитель.

А еще именно с помощью объявления своего окна, получатель может управлять входящим на него трафиком. Если скорость его интерфейса не 1Gb/s, а, например 100Mb/s, то посылая окно меньшее, чем 25000 байт при RTT равным 2 мс, он защитится от перегрузки своего интерфейса.

Window Scale.



Также можно заметить, что поле окно занимает 16 бит в заголовке TCP, что означает, что максимальный размер окна может быть 2^16=65535 байт. Что даст нам в нашем примере с RTT = 2 мс максимальную скорость 262,140,000 бит/с. Поэтому есть Ухищрение №2 - TCP опция Window Scale, которая позволяет получать более большие значения для окон, благодаря умножению числа в поле Window на общее число о котором договорятся отправитель с получателем.

Важно понимать, что при слишком большом RTT и маленьком окне, может случиться, что данные все равно будут ходить пачками, и сеть не будет эффективно использоваться. Для анализа данных ситуаций используется Wireshark - программа, которая перехватывает пакеты с интерфейса и предоставляет их в удобочитаемом виде. С ней мы будем работать на семинаре и будем анализировать данные ситуации, пытаясь скачать файл из какой-нибудь далекой страны, например из Чили.

Рассмотрим еще один вариант развития событий - предположим, что наши два компьютера с гигабитными линками, соединены теперь через какую-то сеть, и в этой сети есть узкое место с пропускной способностью 50 Мбит/с. Если в нашем случае отправитель получив большое окно от получателя сразу нагрузит сеть, то в середине пакеты начнут дропаться и пойдут потери. Для избежания этого существует алгоритм плавного разгона TCP - Slow Start. При таком алгоритме отправитель посылает N пакетов, затем получив ACK на эти пакеты посылает Nx2 пакетов, снова получает ACK, потом уже посылает 4xN пакетов, и так далее, посылая каждый раз в два раза больше пакетов. Это несмотря на то что получатель шлёт нам большое окно и готов сразу принять весь гигабитный поток. Это число N называется Congestion Window или CWND, и чем оно больше, тем быстрее разгоняется передача файлов. В нашем случае, когда на 50 Мбитах/с начнутся потери, отправитель сразу станет слать в несколько раз меньше пакетов и уже более плавно начнет подбираться к нашему потолку 50 Мбит, пока опять не начнутся потери, таким образом “нащупывая” свой поток в плане скорости передачи данных.

Стоит отметить, что каждый раз при отправке данных отправитель сравнивает CWND с RWND, и если CWND < RWND, продолжает разгон,

как только RWND достигнут, то CWND перестает увеличиваться и разгон останавливается. Что тут важно отметить, что при больших задержках, скорость будет расти долго, поэтому CWND чем больше, тем лучше. Ведь если мы грузим большой файл, рано или поздно скорость будет достигнута, а если файл маленький (например веб-страничка), то более-менее существенная скорость не сможет быть достигнута. В современной реализации TCP и на современных ОС, это значение равно 10. Если на вашей ОС это значение меньше, и вы испытываете проблемы с отдачей файлов вашим клиентам, то имеет смысл увеличить это значение.

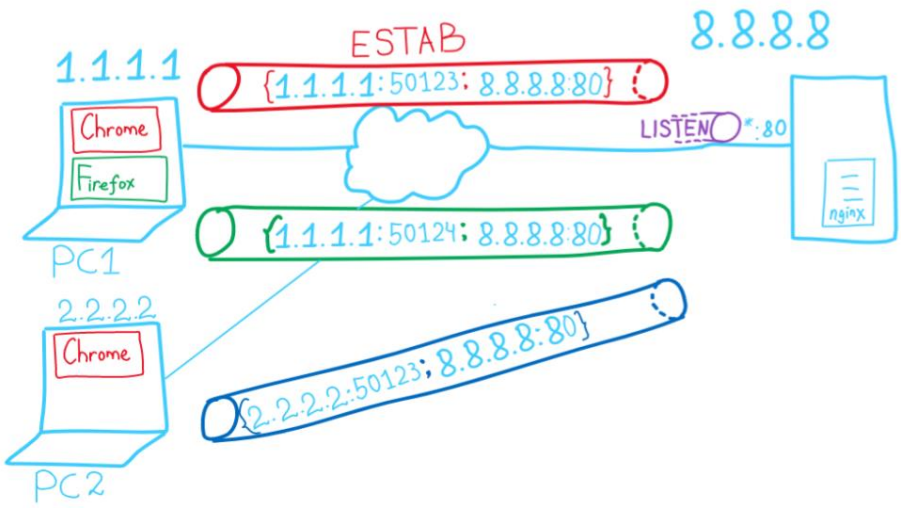
У TCP много дополнительных опций и реализаций. В настоящее время нельзя сказать, что TCP “завершенный” протокол как IP например. Он открыт к новым алгоритмам и решениям разгона скорости передачи трафика. Смотря на какой ОС вы работает, используется один из многих вариантов TCP:
- TCP Tahoe and Reno (исходная реализация),
- TCP CUBIC (по умолчанию на Linux),
- Compound TCP (по умолчанию на Windows) и многие другие.
И все они могут обновляться и новых версиях ОС и приложений. Но сам формат заголовка и основные принципы не меняются.

Итого, чтобы ваше приложение работало максимально эффективно по сети, старайтесь:

- уменьшать RTT и располагаться ближе географически к клиентам,
- снимайте дамп трафика и проверяйте CWND,
- обновляйте ОС.

В IETF - организации поддерживающей протоколы TCP и IP, есть отдельный документ на эту тему:
https://datatracker.ietf.org/doc/html/draft-stenberg-httpbis-tcp
Если в будущем ваше приложение будет испытывать проблемы со скоростью передачи по сети, вспомните этот урок!

Sockets.



Итак, поговорив про TCP и UDP, мы понимаем, что клиент с сервером устанавливают соединение и начинают передавать по нему поток байтов. Такое соединение называется сокетом и оно уникально идентифицируется четверкой параметров + протокол (TCP/UDP):

- IP source
- Port source
- IP destination
- Port destination

В сети не может возникнуть два одинаковых сокета. Если у нас, как на этом слайде PC1 подключится к веб-серверу по порту 80, то у нас получится сокет со следующими характеристиками {1.1.1.1:50123 , 8.8.8.8:80}. Если PC1 откроет еще одно соединение, например из другого браузера, то откроется сокет {1.1.1.1:50124 , 8.8.8.8:80}, который уже не будет равен первому сокету. Сервер не будет путать эти сокеты, и если запрос прилетит через первый сокет, он не отправит данные во второй, он не перепутает их.

Если у нас на PC2 установится соединение с нашим веб-сервером, и, предположим, что там ОС также выберет Port Source 50123, то получится следующий сокет {2.2.2.2:50123 , 8.8.8.8:80}, который также будет отличаться от первого сокеты, из-за того, что у нас в сети не должно быть одинаковых IP адресов.

Сокет может иметь несколько состояний. Например на нашем сервере есть сокет {0.0.0.0:* , 0.0.0.0:80}, который называется сокет в состоянии LISTEN (т.е. прослушивающий сокет). Если мы переведем слово сокет с английского языка, то это будет гнездо (не птичье, а как гнездо для наушников). Собственно сокет в состоянии LISTEN это и есть такое гнездо, в которое не воткнули шнур (т.е. не устанавили TCP или UDP сессию). Но надо понимать, что из сокета в состоянии LISTEN рождаются сокеты в состоянии ESTABLISHED, когда сессия устанавливается. И таких сокетов может установиться много.

С точки зрения приложения сокет можно представить как трубу, в которую можно слать данные (делать write) и получать данные (делать read). А как там работает сеть, маршрутизация и прочее - приложение не волнует, главное сокет установился.

В различных ОС можно посмотреть текущие сокеты. Этим мы займемся на семинаре.

Итак, на сегодня все и давайте подытожим. На этом уроке мы поняли:

- как решается проблема доставки трафика от конкретного приложения до конкретного приложения,
- чем отличается TCP от UDP,
- как TCP гарантирует доставку данных в сетях, где данные могут теряться
- как TCP отвечает за скорость передачи данных
- что такое сокет и как их смотреть

На семинаре мы будем работать с программой Wireshark и учиться перехватывать данные, изучать их, строить графики и даже попробуем перехватить пароль в незащищенном соединении. Также будем смотреть как образуются сокеты. Приходите, будет интересно!

Глоссарий:

L4 - Транспортный уровень - уровень в стеке TCP/IP, который отвечает за передачу данных между приложениями.

TCP - Transmission Control Protocol - протокол передачи данных транспортного уровня с гарантированной доставкой данных. Используется для передачи данных, где их потеря неприемлема (передача файлов, банковских транзакций и т. д.)

UDP - User Datagram Protocol - протокол передачи данных транспортного уровня без гарантированной доставки данных. Используется для передачи таких данных, в которых нет необходимости повторной отправки при потерях.

Port - адрес приложения внутри ОС. По нему определяется какому приложению необходимо передать данные.

TCP handshake - процесс установления сессии для передачи данных по протоколу TCP.

Sequence Number - уникальный номер пакета, номер первого байта в сегменте (если представлять передаваемые данные как поток нумерованных байт).

Acknowlegment Number - номер ожидаемого пакета.

Recieve Window - RWND - количество байт, которое может отправить отправитель без получения подтверждения.

Congestion Window - CWND - количество пакетов, которое сначала отправляет отправитель получателю и ждет на них подтверждения. Нужно для механизма Slow Start. Увеличивается в 2 раза с каждой итерацией, пока не достигнет RWND либо пока не начнутся потери.

Socket - канал данных, устанавливаемый между приложениями. Идентифицируется парой IP+порт клиента и IP+port сервера.