Транспортный уровень. Часть 3. Работа UDP и TCP

Практика в Wireshark. UDP, TCP. Реализация TCP-соединения

Используемая литература

Введение

OSI/ISO	TCP/IP (DOD)
7. Прикладной уровень	4. Уровень приложений
6. Уровень представления	
5. Сеансовый уровень	
4. Транспортный уровень	3. Транспортный уровень
3. Сетевой уровень	2. Сетевой уровень
2. Канальный уровень	1. Уровень сетевых интерфейсов
1. Физический уровень	

Мы продолжаем изучать транспортный уровень, и сегодня рассмотрим, как работают транспортные протоколы, проследим формирование и отправку TCP-сегментов/UDP-дейтаграм.

Для того, чтобы разобраться с работой протоколов TCP и UDP, пронаблюдать использование прикладных протоколов в TCP/UDP, а также в целом проанализировать работу сетей на любом из уровней модели OSI/ISO, необходимо научиться использовать сетевые снифферы. Для этого подойдет сниффер с графическим интерфейсом — Wireshark или tcpdump (дамп из tcpdump можно открыть и в Wireshark).

В Cisco Packet Tracer для анализа трафика необходимо использовать режим симуляции либо устройство Sniffer. Сразу отметим, что Wireshark или tcpdump использовать с Cisco Packet Tracer не получится, так как Wireshark и tcpdump анализируют настоящий трафик. Но если вы используете GNS, то можно выбрать соответствующий сетевой интерфейс, и проанализировать трафик получится. Также можно анализировать трафик для Virtualbox или VMWare.

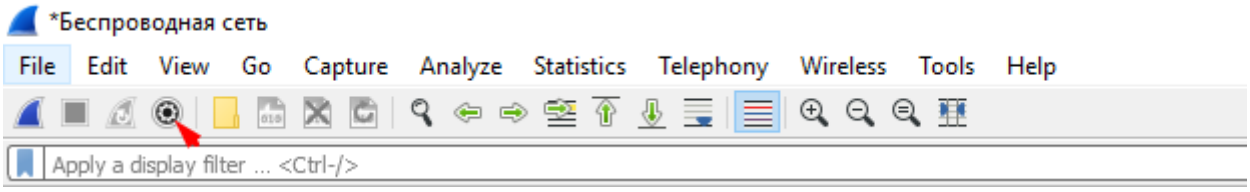
Чтобы установить программу Wireshark, необходимо перейти по адресу https://www.wireshark.org/

Программа работает под управлением Windows, GNU/Linux, Mac OS X и входит в состав Kali Linux. При установке будет предложено установить библиотеку Libpcap или Winpcap. Обязательно это сделайте, иначе программой не получится пользоваться.

Работа в Wireshark

Если вы изучаете сетевую безопасность и используете Kali Linux, можно использовать Wireshark из комплекта Kali Linux.

Открываем вкладку **Capture Interfaces** — кружок в панели инструментов, четвертый слева:



- **Input** — сетевые интерфейсы;
- **Output** — формат сохраняемого файла. Можно указать размер файла, например, чтобы сохранять трафик в файлы по 1 Гб;
- **Options** — дополнительные опции, возможность остановить захват после определенного числа пакетов или по истечении определенного времени.

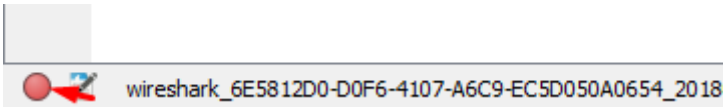
Меню: **Analyze > Enabled Protocols**: весь набор протоколов, которые понимает Wireshark.

Информация в основном окне:

1. Все пакеты.
2. Информация о пакете.
3. Байты пакета.

Если в списке кликнуть правой кнопкой мыши на захваченном пакете, станет доступно поле **Packet Comment**. Оно позволяет добавить произвольный комментарий, который будет отображаться в поле информации о пакете наряду с заголовком. Можно сделать Mark-пакет, без комментария, — это бывает полезно.

Чтобы найти прокомментированные пакеты, следует нажать на кружок в левом нижнем углу окна. Это пригодится для домашнего задания.



В **Statistics > Resolved Address** видим список встреченных IP-адресов и доменных имен, которые в них разрешаются. Фильтр устроен так, что в нем можно использовать все поля в заголовках.

К свойствам пакета обращаются через точку. Например:

```
frame.marked==true
```

или

```
frame.marked eq true
```

Синтаксис:

- **==** или **eq**;
- **!=** или **ne**;
- **>** или **gt**;
- **<** или **lt**;
- **>=** или **ge**;
- **<=** или **le**.

Фильтруем по ip-адресу:

```
ip.addr == 192.168.129.129
```

Рассмотрим, как исключить адрес. Такой способ не будет работать корректно:

```
ip.addr!=192.168.129.129
```

А такой будет:

```
!(ip.addr == 192.168.129.129)
```

И такой тоже:

```
(ip.src != 192.168.129.129)&&(ip.dst != 192.168.129.129)
```

Более сложные условия:

```
ip.src==192.168.129.19&& tcp.port==80
```

Возможен и такой вариант:

```
ip.src==192.168.129.19 and tcp.port==80
```

Чтобы использовать условие «ИЛИ», можно сделать так:

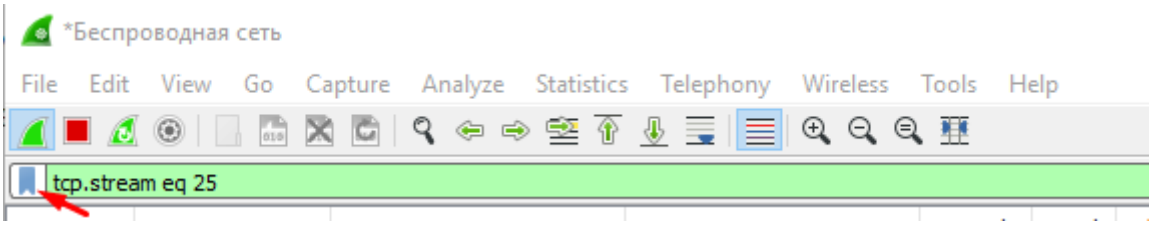
```
ip.src==192.168.129.19||192.168.129.20
```

Или так:

```
ip.src==192.168.129.19 or 192.168.129.20
```

Верхний регистр не работает: **WireShark Case Sensitive**.

Если кликнуть на значок закладки слева от фильтра, можно дать ему название и сохранить для дальнейшего использования.



Чтобы найти доступные фильтры, кликните на значок закладки.

Рассмотрим полезную функцию: **Analyze > Follow Stream > TCP**. Возьмем в качестве примера сайт lib.ru, который не поддерживает **HTTPS**.

Включаем в фильтре **HTTP**:

```
http
```

Видим запросы и ответы.

Зайдем на **samlib.ru** и найдем отправку данных с помощью метода **POST**:

```
http.request.method==POST
```

Можем искать по строке:

```
http contains "password"
```

Научимся выполнять поиск по произвольному слову:

```
data-text-lines contains "copyright"
```

То же можно сделать через меню **FInd > Find Packet.**

Дальше отследим сессию целиком. Нажимаем **Analyze > Follow Stream > TCP**.

Запросы в рамках данного потока будут отмечены красным, а ответы — синим.

Отфильтруются только те пакеты, которые относятся к данной сессии. Автоматически сформируется фильтр, например:

```
tcp.stream eq 17
```

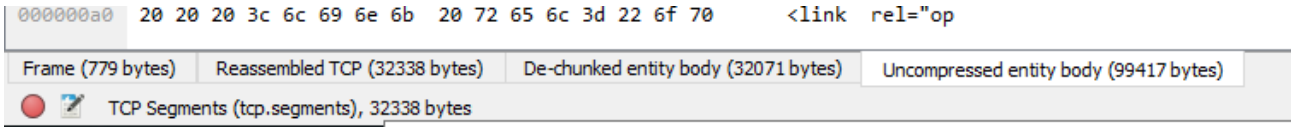
В потоке данных можно увидеть, что их тип — **text/html**, но в тексте идут кракозябры. Напоминает шифрование, но это компрессия. О ней свидетельствует заголовок с типом сжатия **gzip.**

Если кликнем на пакет, содержащий HTTP-ответ, то увидим, что Wireshark собрал целую сессию из фрагментов. Это помечается отдельной вложенностью: **[25 Reassembled TCP Segments].**

В таком случае внизу будут отображаться вкладки:

- **Reassembled TCP**, которая восстанавливает из сегментов поток данных;
- и **Uncompressed entity body**, если данные сжаты.

Увидим, что данные не зашифрованы.



Для любого заголовка в протоколе можно использовать в контекстном меню (правый клик мыши) **Apply As A Filter — Selected**. Тогда фильтр сформируется автоматически.

Если кликнем на заголовке **host: samlib.ru**, получим фильтр:

```
http.host == "lib.ru"
```

Опция **File > Save Objects > HTTP** позволяет сохранить данные, **HTML**, **CSS**, **JS**, картинки и загруженные файлы.

tcpdump

tcpdump — консольная утилита, позволяющая слушать трафик.

Для работы на удаленной машине **tcpdump** может оказаться удобнее. Она позволяет сохранить данные в файл **pcap**, который используется и в Wireshark благодаря общей библиотеке **libpcap**. Затем утилита передает данные по **SSH** и на анализ в Wireshark.

Рассмотрим для примера, как слушать трафик на сетевом интерфейсе **eth0**:

```
tcpdump -i eth0
```

Пример: слушаем TCP-трафик на сетевом интерфейсе **eth0**:

```
tcpdump -i eth0 -p tcp
```

Слушаем TCP- и ICMP-трафик на сетевом интерфейсе **eth0**:

```
tcpdump -i eth0 -p tcp or icmp
```

Слушаем TCP- и ICMP-трафик на сетевом интерфейсе **eth0**, не преобразуя доменные имена в IP-адреса

```
tcpdump -ni eth0 -np tcp or icmp
```

Разберемся с **tcpdump**:

```
man tcpdump
```

Запуск с сохранением дампа:

```
tcpdump -i any -nnvv dst port 23 -w /tmp/test.pcap
```

Анализ с регулярным выражением:

```
tcpdump port http or port ftp or port smtp or port imap or port pop3 -l -A |
egrep -i
'pass=|pwd=|log=|login=|user=|username=|pw=|passw=|passwd=|password=|pass:|use
r:|username:|password:|login:|pass |user ' --color=auto --line-buffered -B20
```

Перенаправление tcpdump-потока с помощью **SSH**:

```
tcpdump -i any ! host 192.168.1.2 -s 0 | ssh someone@192.168.1.2 "cat >
dump.txt"
```

Устройство протокола TCP и принципы его работы

Чтобы понимать, что влияет на скорость работы приложения через сеть, мы должны понимать теорию работы протокола TCP и факторы, на его работу влияющие.

Задержка (latency) и пропускная способность (bandwidth)

Скорость работы приложения — это одна из его отличительных черт, и пользователям нравится, когда приложение работает быстро. Для того, чтобы сделать скорость фичей, мы должны понимать множество факторов и фундаментальных ограничений, которые на нее влияют.

В этой главе мы сосредоточимся на двух важнейших компонентах, которые определяют производительность всего сетевого трафика: задержке (latency) и пропускной способности (bandwidth).

Latency — время, затраченное на передачу пакета отправителем до получения его получателем.

Bandwidth — максимальная пропускная способность логического или физического канала передачи данных.

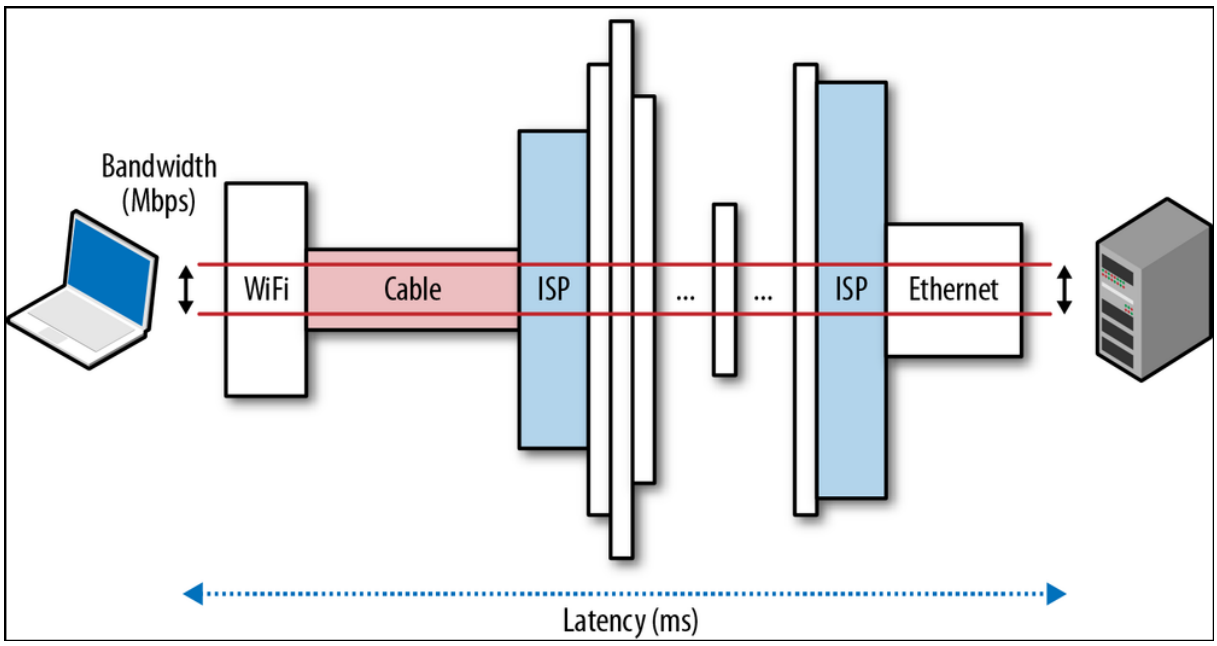


Рис. 1. Latency и bandwidth

На рис. 1 видно, что за пропускную способность канала принимается минимальное из значений пропускной способности каждого компонента сети на пути между двумя устройствами, тогда как latency (задержка) — это сумма задержек, вносимых каждым участником сетевого взаимодействия.

Поняв, как оба фактора (latency и bandwidth) работают вместе, мы сможем понять их влияние на архитектуру и характеристики производительности TCP, UDP и прикладных протоколов над ними (приложений).

Уменьшение задержки в трансатлантическом кабеле Hibernia Express

Latency (задержка) является важным фактором для многих алгоритмов высокочастотного трейдинга (High-Frequency Trading — HFT), где разница в несколько миллисекунд может привести к миллионным прибылям или убыткам.

В начале 2011 г. Huawei и Hibernia Atlantic начали укладку нового трансатлантического кабеля длинной 4600 км из Лондона в Нью-Йорк с целью уменьшить задержку для трейдеров на 5 мс — с ~64 мс до ~59 мс.

Строительство было завершено в 2015 году, и на него было потрачено ~600 млн USD — то есть 1 мс задержки стоит примерно 120 млн USD.

Из чего складывается задержка

Как уже говорилось, latency (задержка) представляет собой время, за которое сообщение или пакет доходит от отправителя до получателя. Любая система состоит из многих компонентов, и все они вносят свой вклад во время доставки сообщения.

Так как клиент зачастую не подключен напрямую к серверу, то между ними находится несколько различных сетевых устройств (маршрутизаторов, коммутаторов, балансировщиков, файрволов и т. д.), работа которых состоит в получении пакета через один из интерфейсов (входящий интерфейс), определение лучшего исходящего интерфейса и отправка пакета через исходящий интерфейс.

Давайте рассмотрим типовые компоненты, влияющие на скорость распространения сигнала от клиента до сервера (или наоборот):

- **Propagation delay** — задержка распространения. Время распространения сигнала в среде от отправителя до получателя. Зависит только от физических параметров, таких как расстояние между участниками и скорость движения частиц в проводнике (электронов или фотонов).
- **Transmission delay —** задержка передачи данных, то есть время, потраченное на передачу всего пакета из памяти устройства в кабель (эфир). Зависит от размера самого пакета, а также скорости интерфейса. Передать 1500 байт через интерфейс со скоростью 100 Gbps быстрее, чем через интерфейс 1 Gbps.
- **Processing delay** — задержка обработки пакета. Время, необходимое устройству на обработку заголовков пакета: проверку контрольных сумм, поиск получателя в таблице маршрутизации, определения исходящего интерфейса и т. д.
- **Queuing delay** — ожидание в очереди. Времени, в течение которого пакет может быть отправлен, но находится в очереди в ожидании отправки, например, если исходящий интерфейс перегружен.

Скорость распространения сигнала (Propagation time) определяется расстоянием и средой передачи данных, через которую проходит сигнал. Скорость распространения обычно колеблется в небольших пределах и зависит от коэффициента преломления кварцевого стекла в случае с оптическим кабелем, либо от скорости электромагнитной волны.

В качестве примера, давайте рассмотрим передачу файла размером 10 Мб по двум каналам связи: 1 Мбит/с и 100 Мбит/с. Чтобы поместить весь файл «в провод» (Transmission delay), в случае с каналом 1 Мбит/с потребуется 10 секунд, а с каналом 100 Мбит/с — всего 0,1 секунды.

Затем, как только пакет поступает в маршрутизатор, тот должен проверить заголовок пакета, чтобы определить, за каким интерфейсом находится получатель, проверить контрольную сумму, поменять TTL, а также может выполнить другие проверки данных. Всё это требует времени (Processing Delay). Большая часть этой логики Processing Delay зачастую реализована аппаратно, поэтому задержки очень малы (на современных коммутаторах/маршрутизаторах уровня дата центра — несколько сотен наносекунд), но они существуют. И, наконец, если пакеты поступают с большей скоростью, чем маршрутизатор способен обработать, то пакеты помещаются в очередь в исходящем/входящем (зависит от модели устройства) буфере (Queuing delay).

Любой пакет, путешествующий по сети, проходит через некоторое количество сетевых устройств, на каждом из которых он столкнется со всеми вышеперечисленными задержками, а из них уже складывается общая задержка (latency). Чем больше расстояние между отправителем и получателем, тем больше времени потребуется для распространения сигнала. Чем больше промежуточных

маршрутизаторов встретится на пути, тем выше задержки обработки и передачи для каждого пакета. Наконец, чем выше загрузка канала, тем выше вероятность задержки пакета в буфере.

Bufferbloat — излишняя буферизация

Bufferbloat — достаточно современный термин, который описывает ситуацию, когда слишком большой буффер и попытка не отбрасывать пакет, а хранить его в буфере максимально долго, ухудшает производительность сети по сравнению с подходом, при котором пакет отбрасывается и пересылается еще раз.

Проблема в том, что производители выпускают устройства со слишком большим размером входящего (исходящего) буфера, поддерживая миф о том, что пакет ни в коем случае не должен быть отброшен. Тем не менее, протокол TCP основан на том, что анализ пропускной способности соединения между отправителем и получателем происходит на основе отброшенных пакетов. То есть попытка излишне буферизировать ломает один из фундаментальных принципов TCP в плане контроля перегрузок (congestion avoidance): благое, казалось бы, намерение снизить потери приводит к еще большей деградации TCP-соединения (увеличению задержек и снижению пропускной способности).

Скорость света и задержка распространения сигнала (Propagation Delay)

Скорость света — постоянная величина, превзойти которую в случае передачи информации невозможно. Это накладывает ограничения на теоретически возможный минимум задержки распространения сигнала. Скорость света достаточно велика — 299 792 458 м/с, но это скорость света в вакууме, которого в кварцевом волоконно-оптическом кабеле нет.

Поэтому скоростью света в проводнике принято считать значение 200 000 000 м/с, так как в зависимости от типа кабеля коэффициент преломления лежит в диапазоне 1,4–1,6.

Показатели задержки между различными населенными пунктами приведены в таблице 1:

Направление	Расстояние	Время, в вакууме	Время, в оптоволокне	Round Trip Time (RTT)
СПб — Москва	750 км	3 мс	4 мс	8 мс

Москва — Иркутск	5585 км	19 мс	28 мс	56 мс
Нью-Йорк — Сидней	15993 км	53 мс	80 мс	160 мс

Таблица 1. Скорость распространения сигнала в оптическом кабеле и вакууме

В таблице приведена теоретически минимально возможная скорость распространения сигнала при условии, что кабель протянут строго по прямой. На деле же длина кабеля куда больше, так как он прокладывается рядом с железными дорогами или, например, ЛЭП.

Сети доставки контента (CDN) предоставляют множество преимуществ, один из основных — размещение контента как можно ближе к пользователю, чтобы до минимума сократить задержку распространения сигнала (Propagation Delay).

Мы не можем заставить сигнал распространяться быстрее, но в наших руках изменить расстояние, которое требуется пройти сигналу.

В случае с веб-трафиком, например, именно задержка (latency), а не пропускная способность (bandwidth) является узким местом, ограничивающим скорость отдачи контента пользователю.

Чтобы понять, почему это так, мы должны понять механизм работы протоколов TCP и HTTP. Несомненно, пропускная способность канала играет свою роль, но только в случае с потреблением «тяжелого» контента, например, скачиванием образа операционной системы, либо просмотром потокового видео. Если рассматривать обычный, ежедневный веб-серфинг, который включает в себя сотни небольших запросов к десяткам небольших ресурсов, таких как картинки, скрипты и т. д., ограничивающим фактором уже будет являться задержка:

- просмотр FHD-видео на youtube.com — зависит от ширины канала;
- открытие странички vk.com — зависит от задержки.

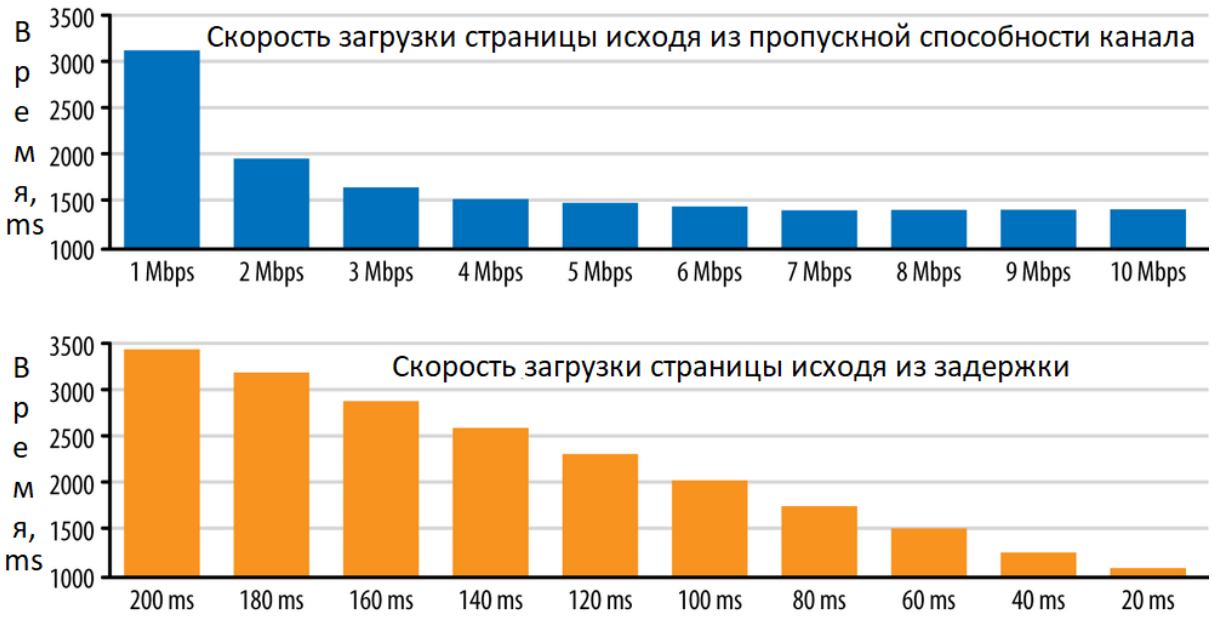


Рис. 2. Скорость загрузки сайта

Как видно на рисунке 2, при достижении скорости соединения в 4 Мб/с скорость загрузки страницы уже практически не повышается. Однако чем ниже RTT (круговая задержка, Round Trip Delay), тем быстрее загружается сайт.

Чтобы разобраться, почему так происходит, мы должны понять, как работает TCP.

Составные части TCP

В основе Интернета лежат два протокола, IP и TCP. IP, или Internet Protocol, отвечает за адресацию устройств, а также за маршрутизацию пакетов от отправителя до получателя, а TCP, или Transmission Control Protocol, работает поверх IP и гарантирует доставку данных приложению.

Первоначальные RFC датированы 1981 годом:

- https://tools.ietf.org/html/rfc791 RFC 791 — Internet Protocol;
- https://tools.ietf.org/html/rfc793 RFC 793 — Transmission Control Protocol;

С тех пор TCP претерпел ряд усовершенствований, но сама логика работы не изменилась. TCP обеспечивает «гарантированную» доставку данных, работая через канал связи, который такие гарантии предоставить не может. TCP скрывает сложность сетевого взаимодействия между приложениями, беря на себя такие вещи, как:

- повторная передача потерянных данных;
- доставка данных в том же порядке, в каком они были отправлены;
- контроль и предотвращение перегрузки канала;
- обеспечение целостности данных и многое другое.

Когда приложение работает по протоколу TCP, ему гарантируется, что все отправленные байты будут идентичны полученным байтам с другой стороны и что они будут доставлены клиенту в том же порядке. Таким образом, TCP оптимизирован для точной, а не своевременной и быстрой доставки данных.

Если говорить о вебе, то стандарт HTTP не определяет TCP как единственный транспортный протокол. То есть никто нам не запрещает доставлять HTTP-запросы через UDP (User Datagram Protocol) или любой другой транспортный протокол, но на практике подавляющее большинство HTTP-трафика в Интернете сегодня доставляется через TCP. Поэтому понимание основных механизмов TCP критически важно для создания оптимизированного веб-интерфейса или приложения. Скорее всего, вы не будете работать с сокетами TCP непосредственно в своем приложении, но сам дизайн приложения будет определять производительность TCP и сети, через которую оно взаимодействует.

Three-Way Handshake

Все TCP-соединения начинаются с 3-Way Handshake (трехстороннего рукопожатия). До того, как начать обмениваться какими-либо данными, отправитель и получатель должны согласовать несколько параметров, такие как порядковый номера пакетов (sequence number), максимальный размер передаваемого сегмента данных (MSS) и некоторые другие. Это необходимо TCP, чтобы понимать, сколько данных было отправлено от клиента к серверу, сколько данных сервер получил и подтвердил, что он их получил, и т. д. Параметр Sequence number в целях безопасности выбирается случайно из некоего диапазона и является просто точкой отсчета, с которой TCP внутри сессии начинает отсчитывать количество переданных данных.

Также следует отметить, что TCP-сессия является двунаправленной, а значит, и клиент и сервер должны иметь разные порядковые номера для отправляемых с их стороны данных.

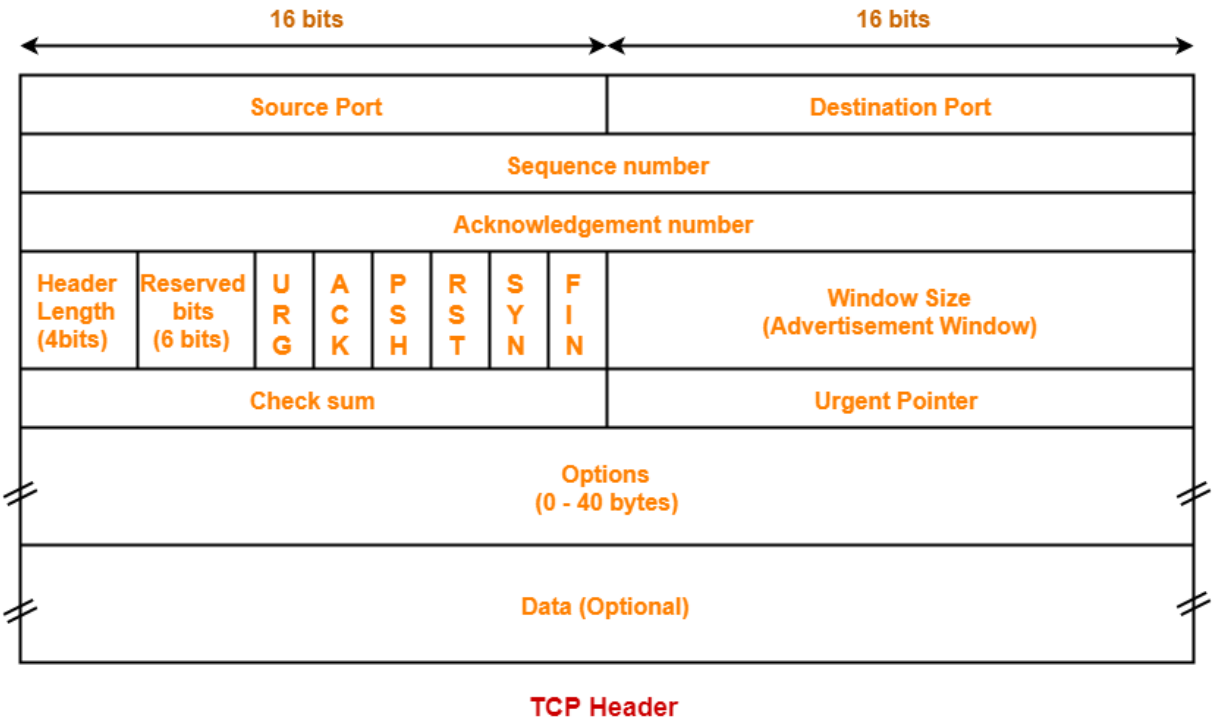


Рис. 3. TCP-заголовок

SYN

Клиент выбирает случайное значение sequence number **X** (порядковый номер), то есть это значение является точкой отсчета переданных байт от клиента к серверу, а также отмечает несколько флагов (один из которых — SYN) и опций в TCP заголовке и отправляет TCP-пакет серверу

SYN ACK

Так как TCP обеспечивает гарантированную доставку, то сервер при получении SYN-пакета от клиента должен проделать сразу несколько действий:

1. Надо подтвердить, что отправленный SYN от клиента с номером **X** был получен.
2. Необходимо начать сессию от сервера в сторону клиента и назначить на эту сессию отдельный sequence number **Y**, так как количество информации переданной от сервера к клиенту также должно быть посчитано, а доставка гарантирована.

Эти два действия могут быть совмещены в одном пакете который называется SYN + ACK (SYN ACK) и имеет два отмеченных флага — SYN и ACK. В нем сервер при помощи флага ACK и значения Acknowledge number **X+1**, подтверждает, что пакет с sequence number **X** был получен, а также при помощи флага SYN и с sequence number **Y** извещает клиента о своем sequence number для передачи данных от сервера к клиенту.

ACK

Полученный SYN ACK от сервера должен быть аналогично подтвержден клиентом. Так как sequence number от сервера к клиенту имеет значение Y, то клиент отвечает ACK сообщением со значением Acknowledge number **Y + 1** в сторону сервера, обозначая то, что SYN от сервера был получен клиентом.

После чего и клиент и сервер убеждаются, что у них есть возможность пересылать и получать данные друг от друга, узнают точки отсчета переданных байт, и теперь приложение может начать передавать свои данные.

Обратите внимание, что со стороны клиента sequence number стал уже X + 1, так как sequence number является счетчиком переданных байт, и в каждом новом пакете он увеличивается на количество переданных байт.

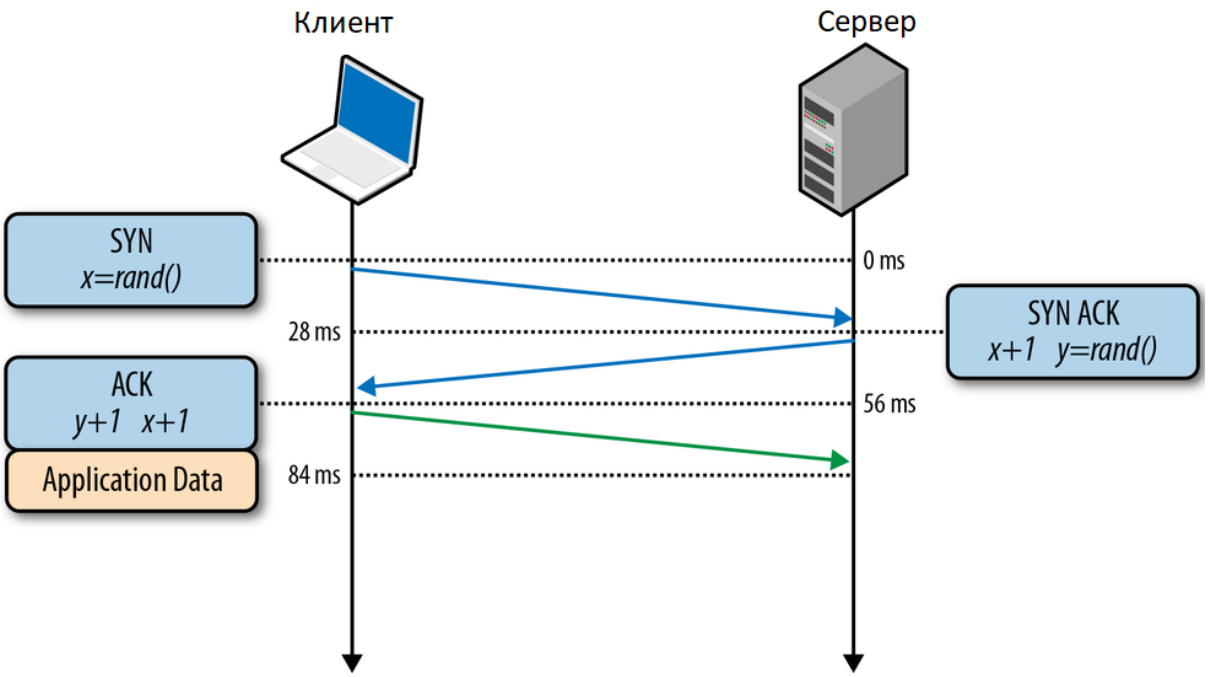


Рис. 4. 3-way handshake

На рисунке показан пример с 3-way handshake, в котором RTT = 56 ms.

Так как данные от приложения (Application Data) могут начать передаваться только после завершения 3-way handshake, клиент может начать отправку сразу после получения ACK от сервера (на 56 ms), а сервер должен дождаться ACK от клиента, прежде чем сможет отправлять какие-либо данные в сторону клиента (на 84 ms).

Этот процесс обязателен для новой TCP-сессии и оказывает сильное влияние на производительность приложений, работающих поверх TCP.

Например, если клиент находится в Москве, а сервер в Иркутске, и мы запускаем новое TCP-соединение по оптоволоконному каналу, то 3-way handshake займет не менее 56 миллисекунд (по 28 мс в каждом направлении).

Обратите внимание, что пропускная способность сети здесь не играет никакой роли. Вместо этого задержка определяется задержкой распространения сигнала между клиентом и сервером, которая, в свою очередь, определяется временем, затраченным на движение фотонов между Москвой и Иркутском.

Задержка, вносимая 3-way handshake, делает создание новых TCP-соединений дорогостоящим процессом, особенно если значение RTT велико, и является одной из основных причин, по которым повторное использование соединений является желаемой оптимизацией для любого приложения, работающего поверх TCP.

TCP Fast Open

Как уже было сказано, фаза 3-way handshake — причина существенной задержки при просмотре веб-страниц, так как веб-трафик представляет собой множество коротких TCP-сессий, необходимых для получения небольшого количества контента на множестве различных хостов.

TCP Fast Open (TFO) — это механизм, направленный на снижение задержки при создании новых TCP-соединений. На основе анализа трафика и эмуляции сети, проведенного в Google, исследователи показали, что TFO, который позволяет передавать данные приложения уже в первом SYN-пакете, может снизить задержку сети для HTTP-транзакции на 15%, время загрузки целой страницы — в среднем на 10%, а в случае каналов с высоким значением latency — на 40%.

Поддержка как клиентского, так и серверного TFO доступна в ядре Linux от версии 3.7 и выше, что делает ее жизнеспособной для новых клиентов и серверов. Но для применения в Интернете нужна поддержка всех «промежуточных» (middleware) устройств — таких как файрволы или балансировщики, которые зачастую зависят от вендоров и от того, с какой скоростью они добавляют новый функционал.

Подробности, а также в каких случаях TFO не может работать, можно почитать тут: https://tools.ietf.org/html/rfc7413.

Congestion Avoidance and Control

Так как сам по себе протокол TCP не имеет понятия о пропускной способности канала между клиентом и сервером, то любая передача данных может привести к перегрузке этого канала. Для решения данной проблемы в TCP существует несколько механизмов: контроль потока (flow control), управление перегрузкой (congestion control), предотвращение перегрузки (congestion avoidance).

Flow Control

Flow control (контроль потока) — это механизм, предотвращающий перегрузку получателя данных отправителем, если получатель по какой-то причине не может обработать полученные данные — например, находится под большой нагрузкой или выделил фиксированный объем буферного пространства для приема данных, которое вот-вот закончится или уже закончилось.

Для решения этой проблемы каждая сторона TCP-соединения объявляет (см. рис. 5) свое собственное окно приема (receive window — rwnd), которое является размером доступного буферного пространства (в байтах) для хранения входящих данных. В TCP-заголовке (рис 3.) это поле Window Size.

Когда соединение устанавливается впервые, обе стороны задают свои собственные значения rwnd, используя системные настройки операционной системы или приложения, и отправляют эти значения друг другу. То есть клиент получает значение rwnd от сервера, а сервер знает rwnd клиента. Это позволяет им знать, сколько байт данных получатель может принять.

В случае с веб-трафиком сервер будет передавать бóльшую часть данных клиенту, что делает окно клиента (rwnd) вероятным узким местом. Однако, если клиент передает большие объемы данных на сервер, например, в случае загрузки изображения или видео, уже окно приема сервера может стать ограничивающим фактором.

Если по какой-либо причине одна из сторон не в состоянии обрабатывать столько данных, сколько она обещала изначально в rwnd, есть возможность сообщить отправителю меньшее значение rwnd.

Если rwnd достигает нуля, это рассматривается как сигнал о том, что необходимо совсем остановить передачу данных, пока данные, уже находящиеся в буфере получателя, не будут обработаны приложением.

Этот процесс продолжается в течение всего времени жизни каждого TCP-соединения: каждый отправленный ACK пакет содержит последнее значение rwnd для каждой стороны, что позволяет обеим сторонам динамически регулировать скорость потока данных в соответствии с емкостью и скоростью обработки данных отправителем и получателем.

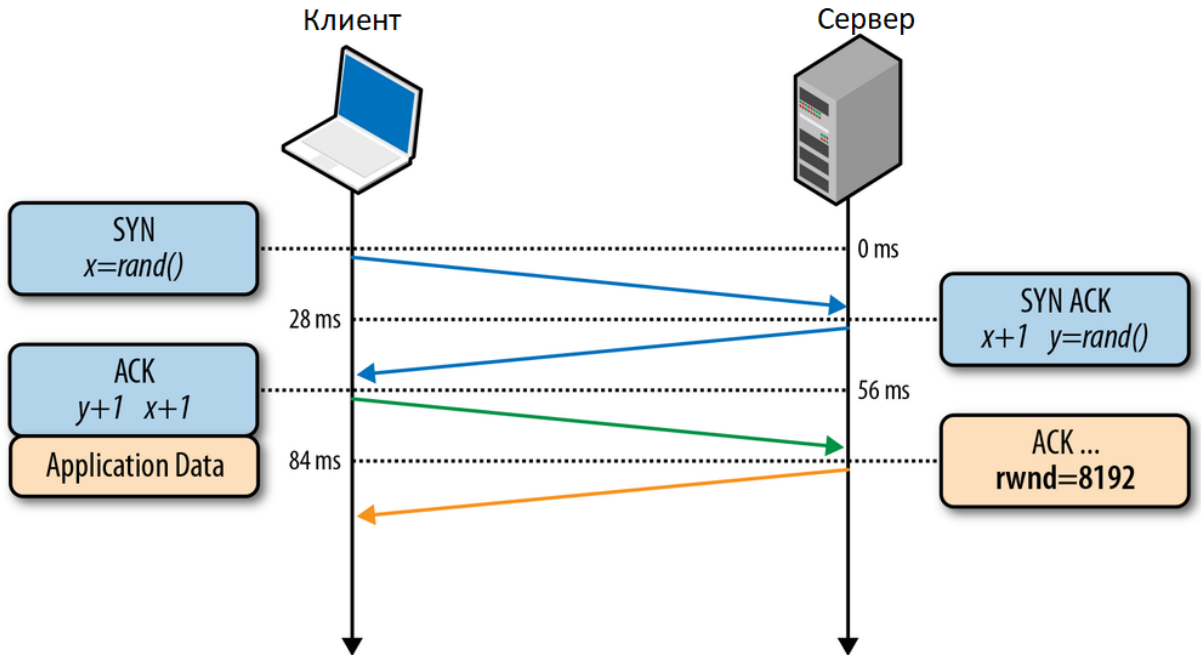


Рис. 5. Receive WIndow (rwnd)

Window Scaling (RFC 1323)

Оригинальная спецификация TCP выделила всего 16 бит для поля receive window в TCP заголовке (рис 3, поле Window Size), что устанавливает жесткую верхнюю границу для максимального размера rwnd (2^16, или 65 535 байт), которое может быть объявлено отправителем или получателем. Как впоследствии оказалось, такого размера часто недостаточно для достижения максимальной производительности, особенно в случае с каналами, имеющими высокую скорость, но и высокую задержку.

Решение этой проблемы описано в RFC 1323 путем добавления опции Window Scaling (масштабирование окна), которая позволяет увеличить максимальный размер receive window с 65 535 байт до 1 гигабайта, не меняя размера поля в заголовке.

Опция Window Scaling сообщается во время 3-way handshake и является числом, которое сообщает количество бит для сдвига влево 16-битного поля receive window.

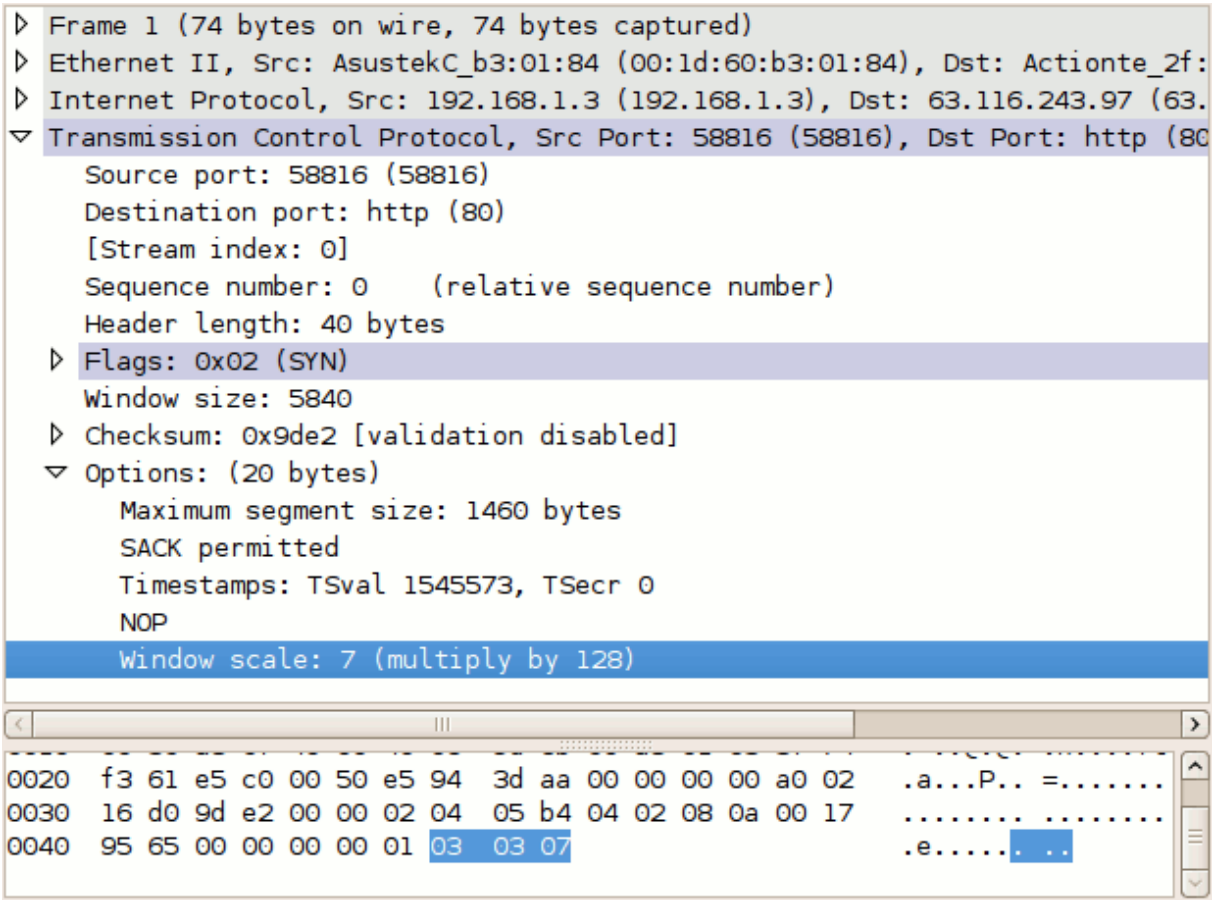
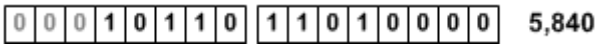
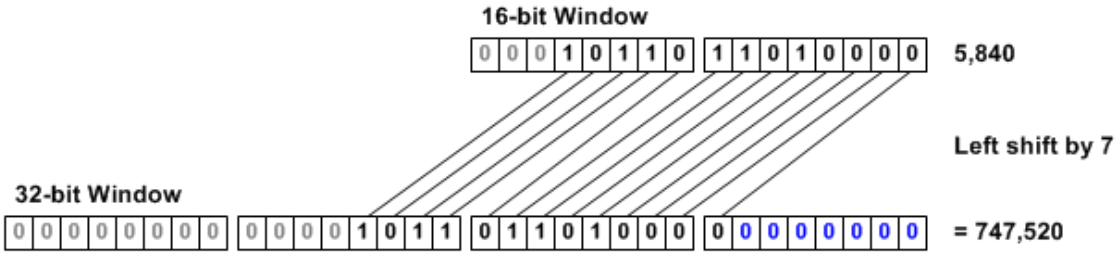


Рис. 6. Window Scaling

Это означает, что берется текущее значение отправленное в rwnd (в случае на рис. 6 это 5840 байт), и переводится в двоичный формат исходя из 16-битного поля:



Дальше представляется, что поле стало размером в 32 бита, и мы сдвигаем значение rwnd влево на 7 нулей, так как Window Scale = 7:



В результате мы получаем новое значение в двоичном формате, равное 747 520 байт.

С точки зрения десятичных значений и математики мы умножаем rwnd на 2^(window scale). В нашем случае:

- rwnd_oring = 5840;
- wscale = 7;
- rwnd = 5840 * 2^7 = 5840 * 128 = 747520.

Следует помнить, что изменить значение window scale невозможно, так как сам множитель передается только в момент установки сессии.

Сегодня Window Scaling TCP включено по умолчанию во всех основных операционных системах. Тем не менее любые промежуточные устройства, например файрволы, могут полностью переписать или даже удалить эту опцию.

Если соединение между сервером и клиентом не может в полной мере использовать доступную полосу пропускания, то можно начать искать причину с проверки receive window и window scaling.

Slow-start

Рассмотренный механизм управления потоком (flow control) работает только между двумя участниками TCP-соединения, что, соответственно, решает проблему с перегрузкой либо клиента, либо сервера. Но ни один из них не знает о том, чему равна пропускная способность канала между ними.

Следовательно, они нуждаются в механизме для оценки полосы пропускания, а также для адаптации передачи данных к постоянно меняющимся условиям в сети. Для для этого у TCP есть несколько алгоритмов: slow-start, congestion avoidance, fast retransmit и fast recovery.

Чтобы понять slow-start, лучше всего увидеть его в действии. Итак, давайте вернемся к нашему клиенту, который находится в Москве и пытается получить файл с сервера в Иркутске.

Сначала выполняется 3-way handshake, во время которого обе стороны объявляют свои соответствующие размеры receive window (rwnd) в пакетах ACK (рис 5).

Единственный способ оценить ширину доступного канала между клиентом и сервером — это измерить ее путем обмена данными, и именно для этого предназначен slow-start.

Для начала сервер инициализирует новую локальную переменную congestion window (окно перегрузки, cwnd) для каждого TCP-соединения и устанавливает ее в начальное системное значение.

Значение переменной cwnd не передается между клиентом и сервером и не является полем в TCP-заголовке. Это локальная переменная для сервера в Иркутске и другая локальная переменная у клиента в Москве.

Далее вводится новое правило: максимальный объем данных «в пути» (не подтвержденных через ACK) между сервером и клиентом должно быть наименьшим значением из rwnd и cwnd. То есть значение rwnd клиент получил от сервера, а значение cwnd — локально, исходя из настроек операционной системы.

Однако это все еще не ответ на вопрос, каким образом сервер и клиент определяют оптимальные значения для их cwnd.

Решение заключается в том, чтобы начать медленный обмен данными, приняв небольшое, консервативное значение как начальное, и увеличивать размер окна при каждом получении ACK. Это и есть механизм slow-start.

Изначальное значение cwnd составляло 1 сегмент; RFC 2581 обновил это значение максимум до 4 сегментов в апреле 1999 года, и совсем недавно, в RFC 6928, вышедшем в апреле 2013 года, это значение было увеличено еще раз — до 10 сегментов.

Так как максимальный объем данных «в пути» для каждого нового TCP-соединения — это минимальное из значений rwnd и cwnd, сервер может отправить клиенту до четырех (или до десяти) сегментов, после чего он должен остановить передачу данных и ждать ACK от клиента.

Затем для каждого полученного ACK алгоритм slow-start указывает серверу, что он может увеличить размер своего окна cwnd на еще один сегмент.

Эта фаза в росте переданных данных для TCP-соединения обычно называется алгоритмом экспоненциального роста (exponential growth) (рис. 7), поскольку клиент и сервер пытаются быстро сойтись на доступной полосе пропускания между ними

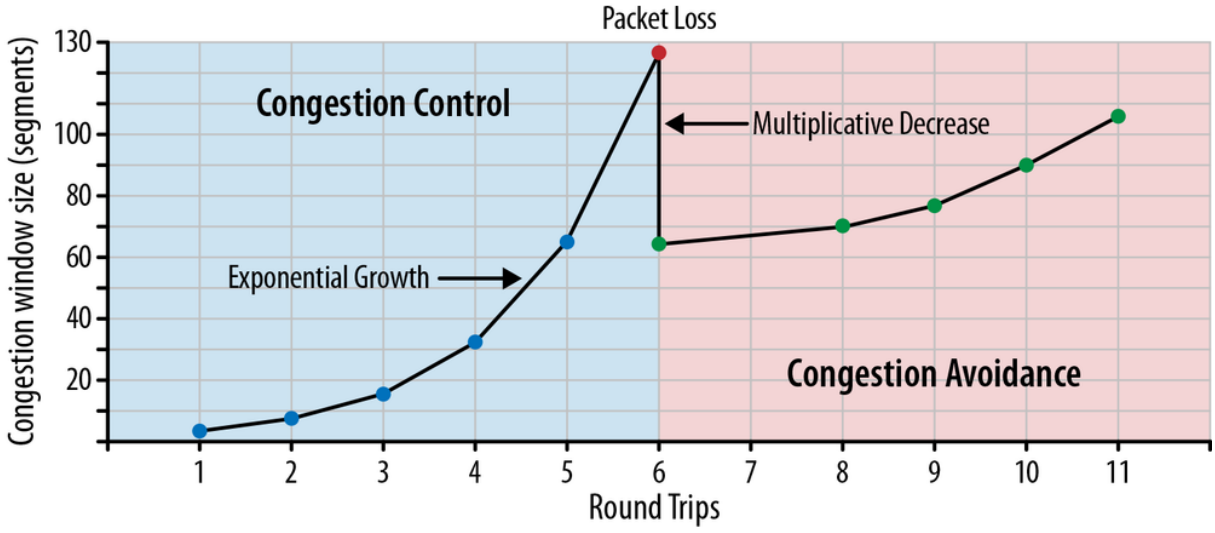


Рис. 7. Фаза экспоненциального роста окна

Вне зависимости от доступной пропускной способности каждое TCP-соединение будет проходить фазу slow start — то есть приложению не будет доступна вся ширина канала сразу же.

Вместо этого TCP начинает с небольшого значения cwnd и удваивает его для каждого RTT. В результате время, требуемое для достижения конкретной пропускной способности **N**, является функцией зависимости от RTT:

$$\text{Время} = RTT \times log_2(\frac{N}{initial\ cwnd}).$$

В нашем случае, если ни клиент ни сервер не используют window scale:

- rwnd и клиента и сервера — 65 535 байт (64 Кб);
- начальное окно cwnd — 4 сегмента (RFC 2581);
- RTT: 56 мс (Москва — Иркутск).

$$\frac{65535\ \text{байт}}{1460\ \text{байт}} \approx 45\ \text{сегментов},$$

$$56\ \text{мс} \times [log_2(\frac{45}{4})] = 224\ \text{мс}.$$

Фактически, чтобы достичь предела в 64 Кб, нам нужно увеличить размер окна cwnd до 45 сегментов, что займет 224 миллисекунды:

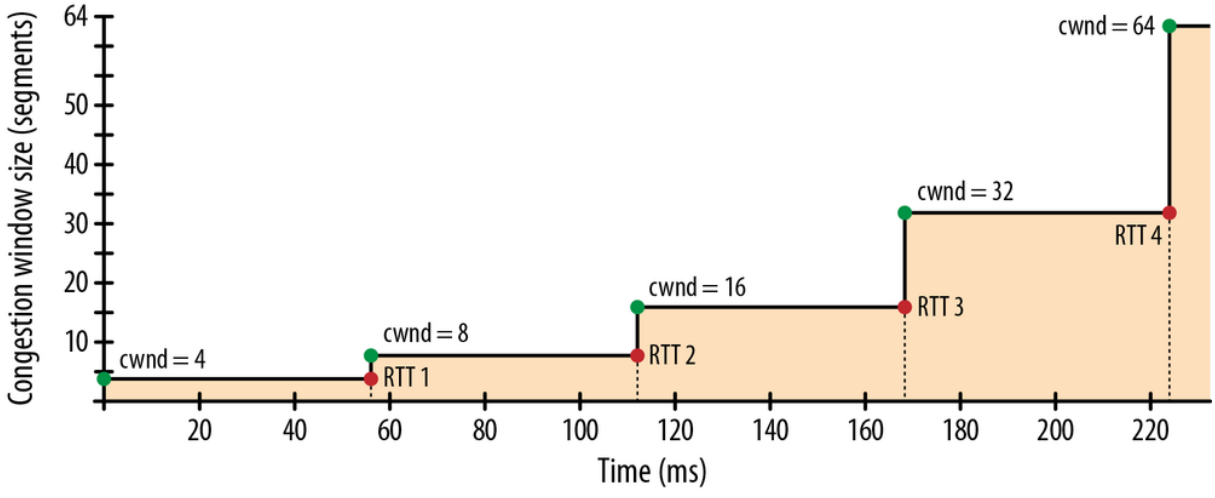


Рис. 7. Рост congestion window исходя из RTT

Slow-start не столь важен для больших потоковых загрузок, поскольку клиент и сервер все равно достигнут максимальных размеров окна через несколько сотен миллисекунд и продолжат передачу с максимальными на их канале передачи данных скоростями.

Однако многие HTTP-соединения короткие и представляют собой лишь несколько пакетов в виде запроса и ответа, и нередки случаи, когда весь обмен информацией завершается еще до достижения максимального размера окна, в результате чего производительность многих веб-приложений часто ограничивается величиной RTT между сервером и клиентом.

Чтобы проиллюстрировать влияние 3-way handshake и slow-start на передачу данных по протоколу HTTP, давайте предположим, что наш клиент в Москве запрашивает файл размером 20 Кб с сервера в Иркутске через новое TCP-соединение (рис. 8) со следующими параметрами соединения:

- RTT — 56 мс;
- пропускная способность канала — 5 Мбит/с;
- rwnd клиента и сервера — 65 535 байт;
- начальный размер cwnd — 4 сегмента по 1460 байт = 5,7 Кб;
- время обработки сервером HTTP запроса — 40 мс;
- отсутствие потерь.

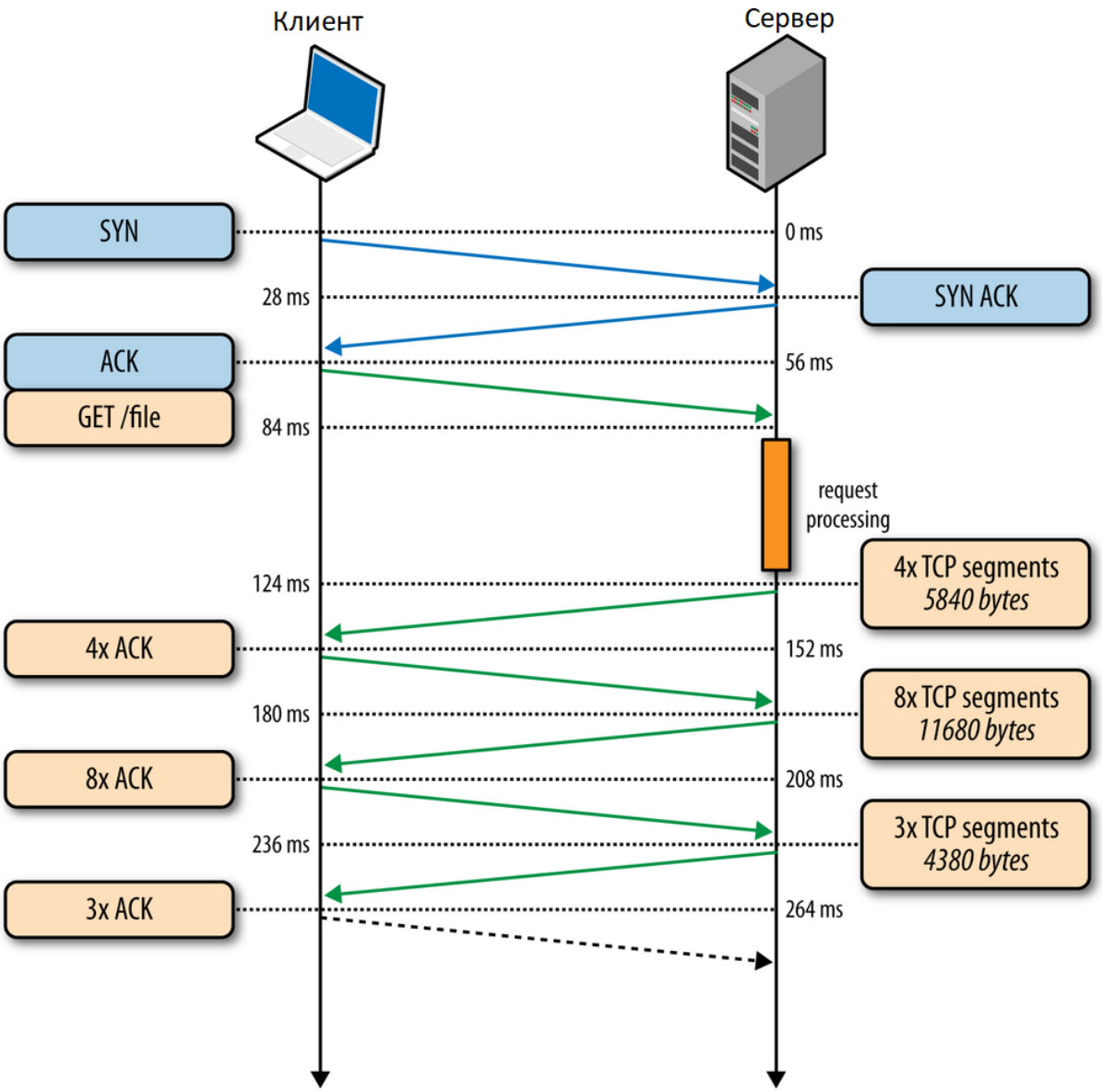


Рис. 8. Получение данных через новое TCP-соединение

0 ms. Клиент начинает TCP 3-way handshake с пакета SYN.

28 ms. Сервер отвечает SYN-ACK и указывает его размер rwnd.

56 ms. Клиент подтверждает получение SYN-ACK, определяет свой размер rwnd и немедленно отправляет HTTP-запрос GET.

84 ms. Сервер получает HTTP-запрос.

124 ms. Сервер завершает создание ответа размером 20 Кб и отправляет 4 TCP-сегмента, прежде чем сделать паузу для ожидания получения ACK (начальный размер cwnd равен 4 и максимальный размер пакета составляет 1460 байт).

152 ms. Клиент получает четыре сегмента и посылает подтверждение ACK на каждый.

180 ms. Сервер увеличивает свой cwnd для каждого полученного ACK и отправляет теперь уже восемь сегментов.

208 ms. Клиент получает восемь сегментов и отправляет ACK на каждый.

236 ms. Сервер увеличивает свой cwnd для каждого ACK и отправляет оставшиеся сегменты.

264 ms. Клиент получает оставшиеся сегменты, и подтверждает каждый при помощи ACK.

Получается, что необходимо как минимум 264 мс для передачи файла 20 Кб через новое TCP-соединение с 56 мс RTT. И это несмотря на то, что пропускная способность канала — 5 Мбит/с.

Давайте теперь предположим, что клиент может повторно использовать уже существующее TCP-соединение (рисунки 2–6) и снова отправляет тот же запрос.

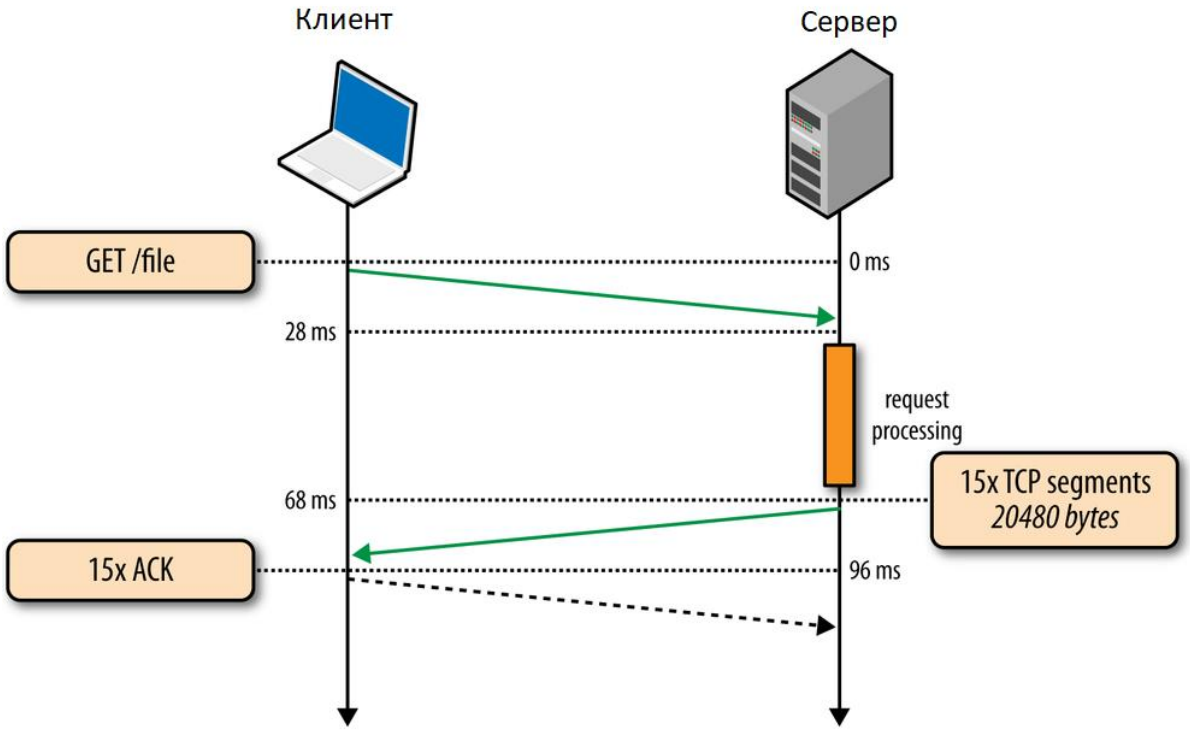


Рис 9. Получение данных через существующее TCP-соединение

0 ms. Клиент отправляет HTTP-запрос.

28 ms. Сервер получает HTTP-запрос.

68 ms. Сервер завершает создание ответа размером 20 Кб, но значение cwnd уже превышает 15 сегментов, необходимых для отправки файла; следовательно, он отправляет все сегменты за раз (15 сегментов по 1460 байт).

96 ms. Клиент получает все 15 сегментов, и подтверждает каждый при помощи ACK.

Тот же самый запрос, сделанный для того же соединения, но без учета 3-way handshake и фазы slow start, теперь занял 96 миллисекунд, что означает повышение производительности на 275%!

Заметьте, в обоих случаях и сервер, и клиент имеют доступ к пропускной способности 5 Мбит/с, которая никоим образом не повлияла на скорость передачи данных в этом соединении. Вместо этого ограничивающими факторами были задержка и размеры окна cwnd.

Congestion Avoidance

Помните, что в TCP потеря пакетов — важнейший механизм обратной связи, который позволяет определить пропускную способность канала и помочь регулировать его производительность. Вопрос не в том, как избежать потери пакета, а в том, когда произойдет эта потеря. Slow-start инициализирует соединение с достаточно консервативным размером окна cwnd, и для каждого RTT удваивает его размер, пока он не превысит размер окна rwnd (окно управления потоком) у получателя, либо пока cwnd не превысит настроенное системой окно порога перегрузки (ssthresh), либо пока пакет не будет потерян: в этот момент алгоритм slow start сменяется алгоритмом congestion avoidance (рис. 7).

В синей области графика размер cwnd растет по экспоненте, и в какой-то момент наступит перенасыщение канала, то есть какой-то маршрутизатор или коммутатор в пути не сможет передать этот пакет и будет вынужден его отбросить, что приведет к потере пакета.

В это же мгновение происходит смена алгоритма экспоненциального роста slow-start на механизм предотвращения перегрузок (красная часть графика) который вначале уменьшает размер cwnd, а затем опять начинает увеличивать cwnd, но уже не столь агрессивно. Как только окно cwnd уменьшено, функция congestion avoidance (предотвращения перегрузки) задействует свои собственные алгоритмы для увеличения окна, чтобы минимизировать дальнейшие потери. В определенный момент произойдет новая потеря пакета, и процесс повторится еще раз. Если посмотреть на график роста cwnd, то мы увидим «пилу» (отражение того, как алгоритм congestion avoidance пытается за некоторое количество итераций найти оптимальный размер cwnd).

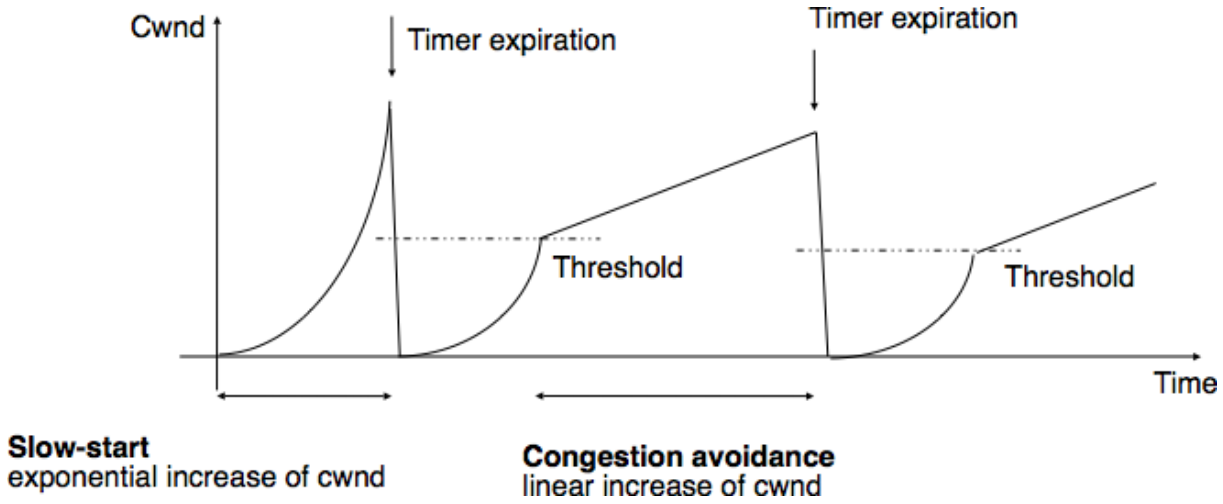


Рис. 10. Влияние потери пакета на рост cwnd

Не существует универсального алгоритма congestion avoidance. Сегодня, в зависимости от операционной системы, вы можете выбрать из множества вариантов: TCP Tahoe, TCP Reno, TCP Vegas, TCP New Reno, TCP BIC, TCP CUBIC, DC-TCP, TCP BBR и так далее. Все они используют разные алгоритмы роста cwnd в зависимости от характеристик канала, и какие-то работают лучше, когда

задержки очень малы, а каналы широкие (например, внутри дата-центра), а какие-то работают лучше в случае с большими задержками и наличием потерь (3G-сети). Тем не менее основные характеристики производительности по контролю и предотвращению перегрузок сохраняются независимо от алгоритма.

Bandwidth-Delay Product (BDP)

Встроенные механизмы управления перегрузкой (congestion control) и предотвращением перегрузки (congestion avoidance) в TCP сильно влияют на производительность соединения: оптимальные размеры окон отправителя и получателя находятся в зависимости от RTT и скорости канала передачи данных между ними.

Максимальный объем неподтвержденных данных (данные «в пути», на которые еще получен ACK) между отправителем и получателем определяется как минимум размеров окон приема (rwnd) и перегрузки (cwnd): текущий размер rwnd передается в каждом ACK, а окно перегрузки является локальным значением для каждого из участников и никуда не передается, оно динамически настраивается отправителем на основе алгоритмов контроля и предотвращения перегрузки.

Следовательно, если отправитель или получатель превышает максимальный объем неподтвержденных данных, он должен остановиться и дождаться, пока получатель не подтвердит полученные пакеты, прежде чем продолжить и это время ожидания будет равно RTT между клиентом и сервером.

Если отправителю или получателю часто приходится останавливаться и ждать ACK для предыдущих пакетов, это приведет к появлению пробелов в потоке данных (рис. 11), что, следовательно, ограничит максимальную пропускную способность соединения.

Если выбрать слишком маленький размер окна, это будет ограничивать пропускную способность соединения независимо от реально доступной пропускной способности между узлами.

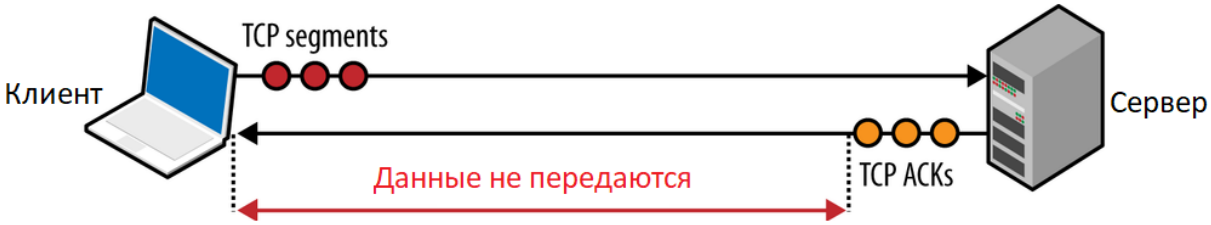


Рис. 11. Из-за небольшого размера окна данные не передаются, пока ACK не получен

Давайте рассчитаем максимальную пропускную способность канала. Предположим, что:

- минимум от cwnd и rwnd составляет 16 Кб;
- RTT = 100 мс (0,1 с):

$$16\ \text{Кб} = 16 \times 1024 \times 8 = 131072\ \text{бит},$$

$$\frac{131072 \text{ бит}}{0{,}1 \text{ с}} = 1310720 \text{ бит/с},$$

$$1310720 \text{ бит/с} = \frac{1310720}{1000000} = 1{,}31 \text{ Мбит/с}.$$

Получается, что независимо от доступной пропускной способности между отправителем и получателем (то есть даже если скорость соединения между ними будет 100 Гб/с) скорость TCP-соединения между ними не будет превышать скорость передачи данных 1,31 Мбит/с!

Чтобы достичь более высокой пропускной способности, нам нужно либо увеличить минимальный размер окна, либо уменьшить RTT.

Давайте рассчитаем размер окна исходя из пропускной способности канала.

Предположим, что RTT остается тем же самым (100 мс), но отправитель имеет 10 Мбит/с доступной полосы пропускания между ними:

$$10 \text{ Мбит/с} = 10 \times 1000000$$

$$= 10000000 \text{ бит/с},$$

$$10000000 \text{ бит/с} = \frac{10000000}{8 \times 1024} = 1221 \text{ Кб/с},$$

$$1221 \text{ Кб/с} \times 0{,}1 \text{ с} = 122{,}1 \text{ Кб}.$$

То есть чтобы максимально эффективно использовать канал 10 Мбит/с необходим размер окна не менее 122,1 Кб. Следует помнить, что максимальный размер окна rwnd в поле TCP заголовка составляет 64 Кб, если не указана опция Window Scale. Соответственно, если вы хотите достигнуть высоких скоростей в TCP-сессии, без wscale не обойтись.

Head-of-Line Blocking

TCP обеспечивает абстракцию надежной сети, работающей по ненадежному каналу, которая включает в себя в том числе доставку пакетов по порядку и повторную передачу потерянных пакетов. Однако эти

функции не всегда необходимы для приложения и могут привести к ненужным задержкам и негативно повлиять на его производительность.

Каждый TCP-пакет имеет уникальный порядковый номер, и данные должны быть переданы получателю по порядку (рис. 12). Так как само приложение передает в TCP последовательность байт, зачастую не имеющую четких границ разбиения на пакеты, то все эти байты должны быть переданы приложению со стороны получателя как единый блок. Если один из пакетов был потерян на пути к получателю, то все последующие пакеты должны храниться в буфере TCP получателя до тех пор, пока потерянный пакет не будет повторно передан и получен.

Поскольку эта работа выполняется на уровне TCP, приложение не знает, что что-то было потеряно, и должно дождаться полной последовательности данных, прежде чем сможет получить к ним доступ. Поэтому приложение просто видит задержку доставки, когда пытается прочитать данные из сокета. Этот эффект известен как блокировка типа TCP head of line (HOL).

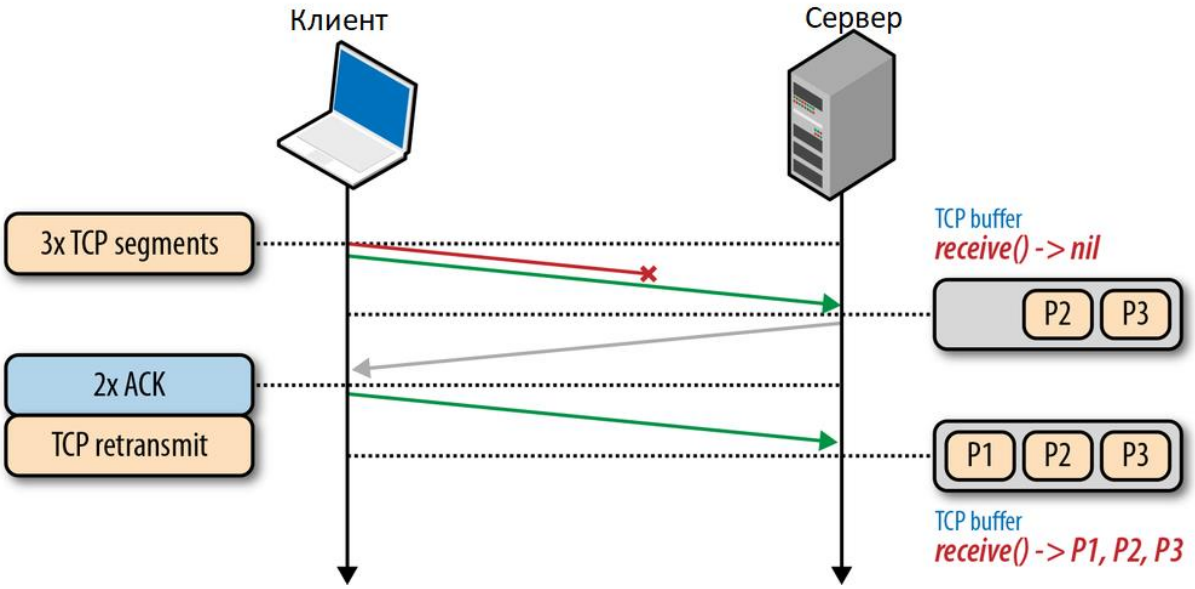


Рис. 12. Блокировка соединения из-за потери пакета

Задержка, вызванная HOL, позволяет приложениям избегать переупорядочивания и повторной сборки пакетов со стороны приложения, что значительно упрощает код самого приложения. Однако это делается за счет введения непредсказуемого изменения времени ожидания доставки пакета в приложение (обычно оно называется jitter), которое может отрицательно повлиять на производительность приложения.

Кроме того, некоторым приложениям может даже не потребоваться ни надежная доставка, ни доставка по порядку, например, если каждый пакет является автономным сообщением. К сожалению, TCP не может обеспечить такой функционал — все пакеты упорядочены и должны быть доставлены по порядку.

Приложениям, которые могут иметь дело с неупорядоченной доставкой или потерей пакетов и которые чувствительны к задержке или jitter, вероятно, лучше использовать альтернативный транспортный протокол, такой как UDP.

Потеря пакета — это норма

Фактически потеря пакетов необходима для получения максимальной производительности от TCP. Отброшенный пакет действует как механизм обратной связи, который позволяет получателю и отправителю регулировать свои скорости отправки, чтобы избежать перегрузки сети и минимизировать задержку. Кроме того, некоторые приложения могут допускать потерю пакетов без неблагоприятных последствий: передача аудио, видео и игровых состояний не требует ни надежной, ни упорядоченной доставки — кстати, именно поэтому WebRTC использует UDP в качестве транспорта.

Если пакет потерян, то аудиокодек может просто вставить незначительный перерыв в аудио и продолжить обработку входящих пакетов. Если промежуток небольшой, пользователь может даже не заметить, при этом если бы доставка гарантировалась, то ожидание потерянного пакета могло бы ввести переменные паузы в выводе звука, что уже воспринимается слушателем как значительно худшее качество звучания.

Аналогично, если мы доставляем обновления местоположения игрового персонажа в трехмерном мире, ожидание пакета, описывающего его положение во время T – 1, совершенно бесполезно, когда у нас уже есть пакет с позицией персонажа во время T.

TCP Selective Acknowledgments (SACK)

Так как потеря пакетов является нормой в TCP, давайте рассмотрим, как происходит пересылка пакета. На рисунке 13 клиент отправляет некоторый запрос на сервер, и сервер формулирует ответ, разбитый на четыре сегмента TCP (4 пакета). Сервер передает все четыре пакета в ответ на запрос. Однако второй пакет от сервера к клиенту отбрасывается где-то в сети и никогда не достигает клиента. Давайте посмотрим на то, как с этим справляется TCP.

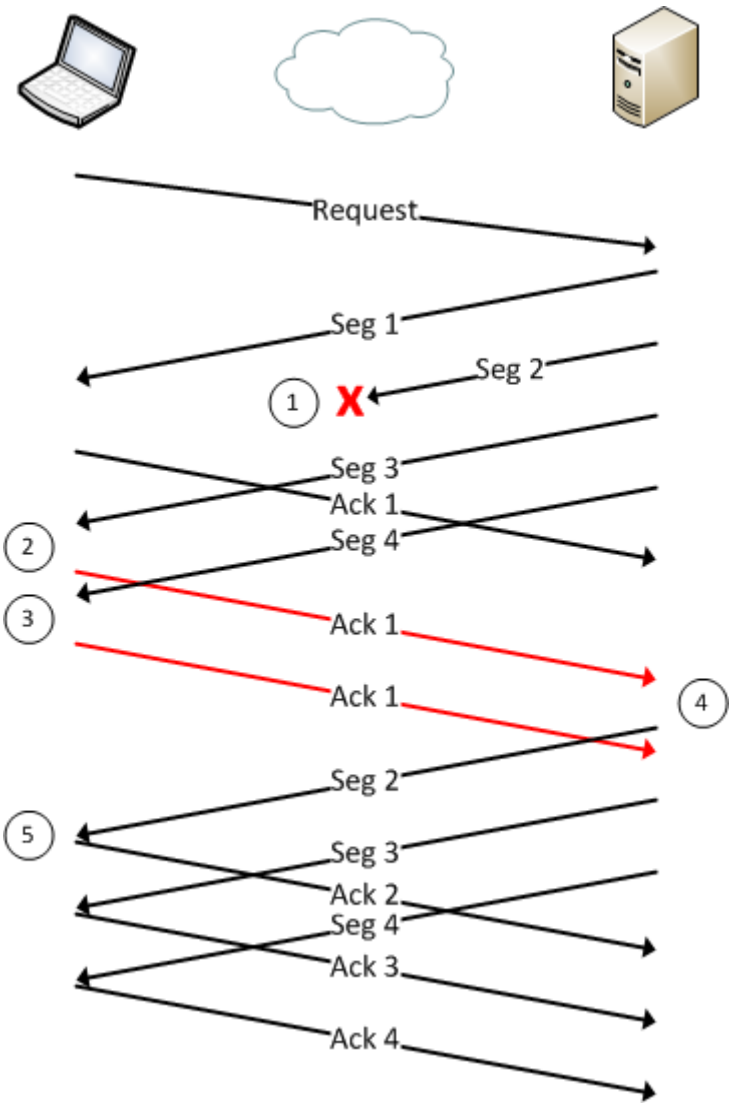


Рис. 13. Потеря пакета номер 2

1. Пакет под номером Seg2 утерян где-то в сети.
2. Клиент получает сегмент номер Seg3. После проверки sequence number сегмента клиент понимает, что данные пришли не по порядку — отсутствуют данные между последним полученным пакетом (Seg 1 был получен) и текущим полученным — Seg3.
    a. Клиент еще раз передает в сторону сервера ACK1 для пакета 1, чтобы проинформировать сервер о том, что он не получил никаких последовательных данных после Seg1.
3. Поскольку сервер еще не знает, что что-то пошло не так (так как он еще не получил дубликат ACK1), он продолжает отправку Seg4. Клиент, получив Seg4 и проверив его sequence number понимает, что данные все еще в неверном порядке и не подходят к последнему полученному и подтвержденному пакету Seg1. Клиент повторяет свое поведение на третьем шаге, отправляя еще одно, дублирующее подтверждение для пакета 1 (ACK1), чтобы проинформировать сервер о том, что последним полученным и подтвержденным по очереди сообщение было Seg1.
4. Сервер получает первое дублированное подтверждение клиента для пакета 1. Поскольку клиент подтвердил только получение первого из четырех отправленных сегментов, сервер

должен повторно передать все три оставшихся сегмента, даже несмотря на то, что Seg3 и Seg4 фактически были получены клиентом. Второе подтверждение ACK1, полученное от клиента, игнорируется.

5. Клиент успешно получает и подтверждает три оставшихся сегмента Seg2, Seg3, Seg4, отправив ACK2, ACK3 и ACK4.

Как видно из происходящего, потеря одного лишь пакета вызвала пересылку всех последующих, что не совсем эффективно. Поэтому для увеличения эффективности TCP во время потери пакетов существует опция selective acknowledgment (SACK), описанная в RFC 2018: https://tools.ietf.org/html/rfc2018.

SACK работают путем добавления к дублирующему ACK пакету опции TCP, содержащей диапазон полученных несмежных данных. Другими словами, это позволяет клиенту сказать: «У меня есть только до пакета № 1 по порядку, но я также получил пакеты № 3 и № 4». Это дает возможность серверу понять, какие данные следует повторно переслать клиенту.

Поддержка SACK оговаривается во время 3-way handshake в начале соединения TCP обе стороны соединения должны поддерживать SACK. Давайте посмотрим, как будет выглядеть предыдущий пример с включенным SACK (рис. 14):

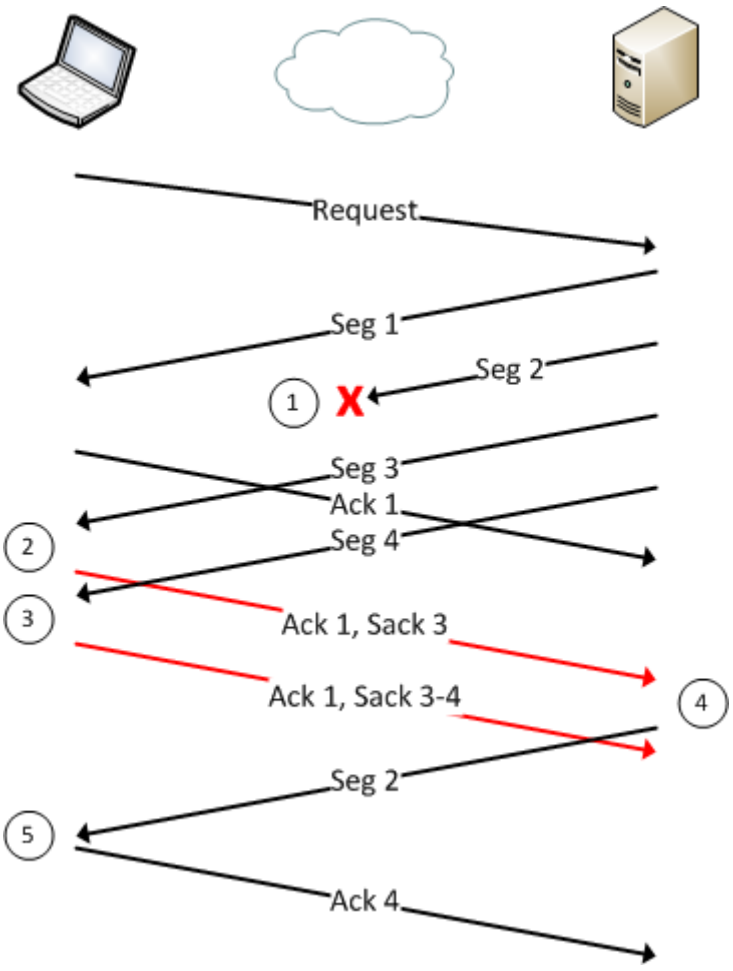


Рис. 14. Потеря пакета номер 2 при включенном SACK

1. Пакет под номером Seg2 утерян где-то в сети.
2. Получив пакет под номером Seg3, и проверив seq number, клиент понимает, что отсутствует сегмент между пакетами Seg1 и Seg3. Он отправляет дублированное подтверждение для пакета номер 1 ACK1 и добавляет туда опцию SACK, указывающую, что он получил пакет Seg 3.
3. Клиент получает пакет Seg4 и отправляет еще одно дублирующее подтверждение для пакета 1 — ACK1, но на этот раз расширяет опцию SACK, чтобы показать, что он получил пакеты Seg3 и Seg4.
4. Сервер получает дубликат ACK клиента для пакета 1 ACK1 , внутри которого указан SACK для пакета номер 3 Seg3 (эти данные находятся внутри этого ACK-сообщения). Из этого сервер делает вывод, что у клиента отсутствует пакет номер 2 Seg2, поэтому пакет Seg2 передается повторно. Следующий SACK, полученный сервером, указывает, что клиент также успешно принял пакет Seg4, поэтому больше ничего передавать не нужно.
5. Клиент получает пакет Seg2 и отправляет подтверждение ACK, указывающее, что он получил все данные, покрывая диапазон Seg2 — Seg4.

Как видно, SACK позволяет гораздо более эффективно использовать ресурсы участников сетевого взаимодействия и ускоряет взаимодействие по TCP.

UDP

Слова «датаграмма» и «пакет» часто употребляются как синонимы, но эти понятия не совсем тождественны. Хотя термин «пакет» применяется к любому отформатированному блоку данных, термин «датаграмма» часто зарезервирован для пакетов, доставляемых через ненадежный сервис — без гарантий доставки и уведомлений об ошибках.

Протокол UDP инкапсулирует пользовательские сообщения в свою собственную структуру пакета (рис. 15), которая добавляет только четыре дополнительных поля: порт отправителя, порт получателя, длину пакета и контрольную сумму. Таким образом, когда протокол IP доставляет пакет на хост назначения, хост смотрит на UDP-заголовок и идентифицирует приложение по порту назначения, после чего доставляет сообщение в это приложение.

Bit	+0..7	+8..15	+16..23	+24..31
0	Source Port		Destination Port	
32	Length		Checksum	
...	*Payload*			

Рис. 15. Заголовок UDP

Фактически и порт источника, и поля контрольной суммы являются необязательными полями в дейтаграммах UDP. Другие особенности протокола UDP:

- нет гарантии доставки сообщений;
- нет подтверждений, повторных передач или тайм-аутов;
- нет гарантии упорядоченной доставки сообщений;
- нет порядковых номеров пакетов, нет переупорядочения, нет head of line blocking;
- нет отслеживания состояния соединения;
- нет протокола установки и закрытия соединения;
- нет механизмов предотвращения и контроля перегрузок.

TCP — это протокол, ориентированный на поток байтов, способный передавать сообщения от приложений, разбитые по нескольким пакетам, без каких-либо явных границ сообщений внутри самих пакетов.

Для достижения этого TCP имеет понятие о состоянии соединения между клиентом и сервером, и каждый пакет имеет свой порядковый номер и отправляется по очереди, повторно передается при потере и доставляется по порядку. Дейтаграммы UDP, с другой стороны, имеют четкие границы: каждая дейтаграмма передается в одном IP-пакете, и каждое сообщение от приложения должно поместиться в одну дейтаграмму. UDP-дейтаграммы не могут быть фрагментированы: одно сообщение от приложения не может быть передано в двух UDP-дейтаграммах.

UDP — это простой протокол без сохранения состояния, подходящий для начальной загрузки других протоколов приложений поверх: право принимать практически все решения по проектированию протоколов предоставляется приложению над ним. Однако прежде чем приступить к реализации собственного протокола для замены TCP, вы должны тщательно продумать такие сложности, как взаимодействие UDP со многими уровнями развернутых промежуточных ящиков (обход NAT), а также общие рекомендации по проектированию сетевых протоколов. Без тщательного проектирования и планирования разработчики нередко начинают с яркой идеи нового протокола, а в итоге получают плохо реализованную версию TCP. Алгоритмы и конечные автоматы в TCP оттачивались и совершенствовались на протяжении десятилетий, и в них учитывались десятки механизмов, которые легко поддаются хорошей репликации.

Практическое задание

Установить программу Wireshark: https://www.wireshark.org/.

Программа работает под управлением Windows, GNU/Linux, Mac OS X. Входит в состав Kali Linux. При установке будет предложено установить библиотеку Libpcap или Winpcap. Обязательно это сделайте, иначе программой не получится пользоваться.

Далее выполните задания и составьте отчет (можно со скриншотами) с описанием ваших действий и выводами.

Важно: в качестве практического задания сдается отчет в формате .docx или .pdf. Не присылайте дампы: это не только не может быть зачтено как практическое задание, но и небезопасно.

1.Найти нешифрованный HTTP-сайт, где есть регистрация и логин. Отправить фейковые данные. Сможет ли злоумышленник перехватить пароль?

2. Найти нешифрованный HTTP-сайт со множеством картинок. Рекомендуется использовать Google Chrome. Сколько TCP-соединений будет открыто и почему?

3. Повторите п.1 с TLS. Вопрос тот же.

4 ** Какие интересные протоколы можно обнаружить, если зайти при помощи Google Chrome на YouTube? (Сработать может не у всех.)

5 ** Если хочется совсем сложное задание, то попробуйте изучить трафик при подключении к FTP-серверу (достаточно любого публичного нешифрованного FTP-сервера, например Yandex.Mirror)

Дополнительные материалы

1. https://www.wireshark.org/docs/wsug_html/.
2. Лихтциндер Б. Я., Поздняк И. С. Анализ трафика мультисервисных сетей / Учебное пособие для проведения лабораторно-практических занятий / Под ред. Карташевского В. Г., рецензия Васина Н. Н. Поволжский государственный университет телекоммуникаций и информатики. Кафедра МСИБ. Самара, 2013. — 95 с. — Электронный ресурс [Режим доступа] URL: http://msib.psuti.ru/content/metod/Анализ_трафика_мультисервисных_сетей.pdf.
3. Таненбаум Э., Уэзеролл Д. Т18 Компьютерные сети. 5-е изд. — СПб.: Питер, 2012. — 960 с. (Главы 6, 7.)
4. High Performance Browser Networking. by Ilya Grigorik. Publisher: O'Reilly Media, Inc. Release Date: September 2013. ISBN: 978144934475.
5. TCP/IP Illustrated, Volume 1: The Protocols by W. Richard Stevens, Kevin R. Fall Publisher: Addison-Wesley Professional Release Date: November 2011 ISBN: 9780321336316.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. https://www.wireshark.org/docs/wsug_html/.
2. Лихтциндер Б. Я., Поздняк И. С. Анализ трафика мультисервисных сетей / Учебное пособие для проведения лабораторно-практических занятий / Под ред. Карташевского В. Г., рецензия Васина Н. Н. Поволжский государственный университет телекоммуникаций и информатики. Кафедра МСИБ. Самара, 2013. — 95 с. — Электронный ресурс [Режим доступа] URL: http://msib.psuti.ru/content/metod/Анализ_трафика_мультисервисных_сетей.pdf.