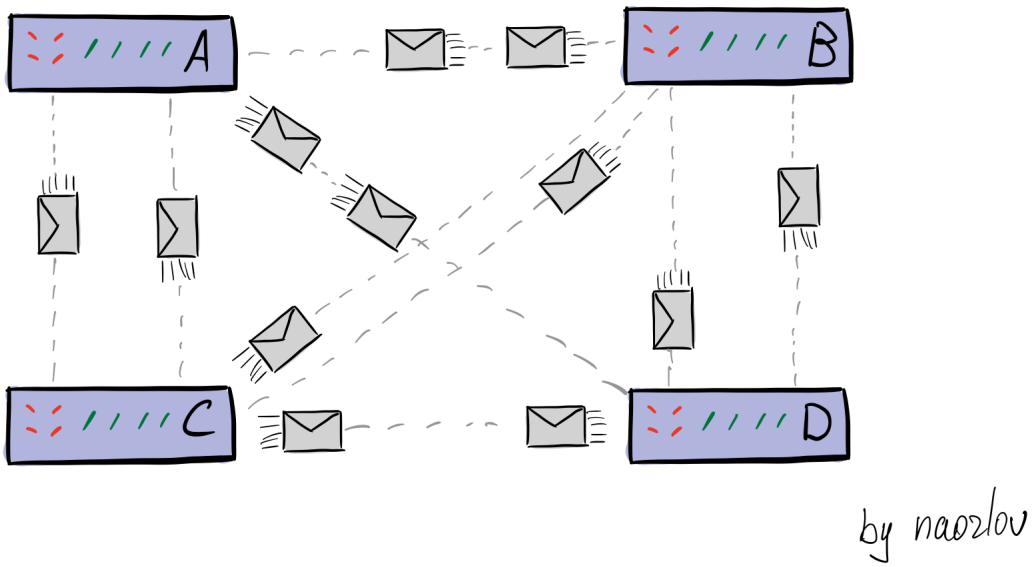
Сети, TCP/UDP, сокеты

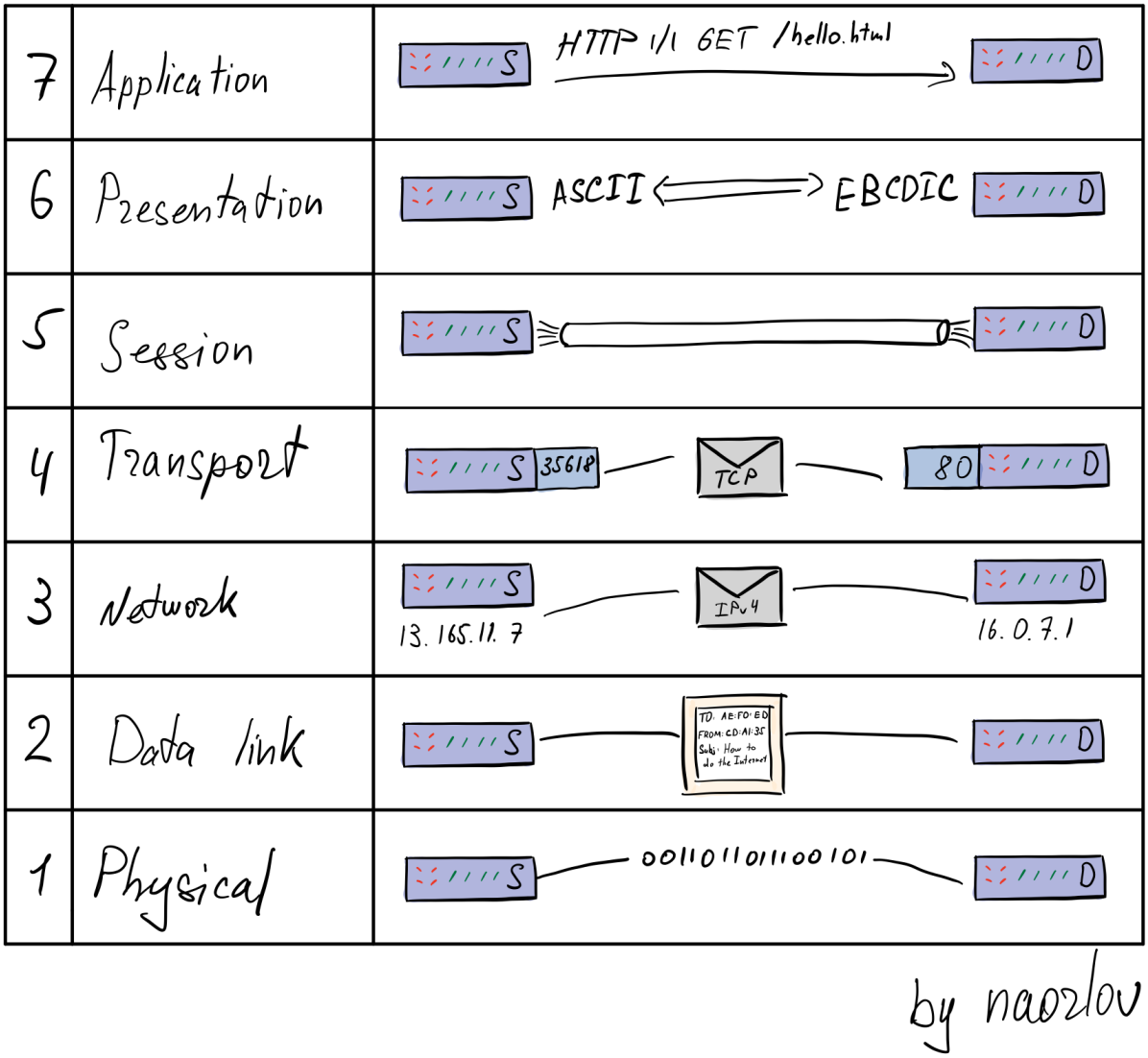
Здесь и далее PC == распределенная система

Для понимания работы распределенных систем необходимо чувствовать "кровеносную систему PC" -- сеть между узлами. Начнем мы с теоретической модели сети.



Модель OSI/ISO

Одной из моделей, с помощью которой пытались строить сети, является модель OSI (Open Systems Interconnection model). Задачей этой модели является разбиние межузлового взаимодействия по разным слоям абстракции.



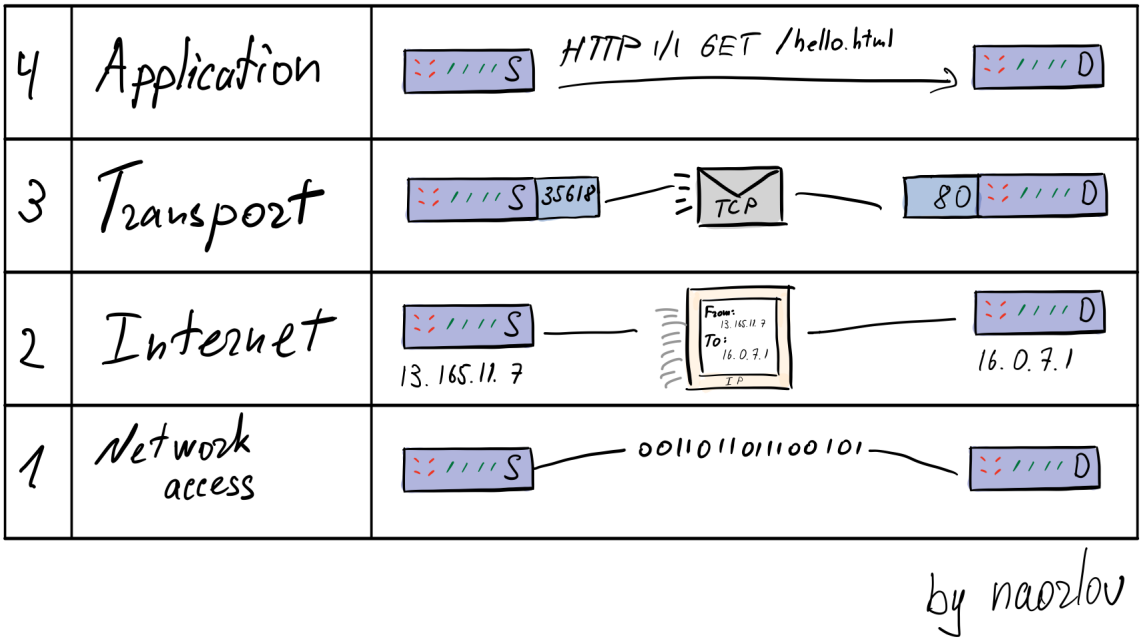
Модель принято описывать начиная с верхнего уровня, поэтому мы не откажем себе в этом удовольствии.

- 7. **Application layer** (прикладной уровень). Обеспечивает взаимодействие конечных приложений с сетью. Пример такого протокола -- HTTP
- 6. **Presentation layer** (уровень представления). Обеспечивает в первую очередь кодирование/декодирование данных между разными форматами и кодировками, может сжимать/разжимать, шифровать/расшифровывать данные и заниматься в целом такого рода преобразованиями.
- 5. **Session layer** (сеансовый уровень). Уровень управляет созданием/завершением сеанса, обменом информацией, синхронизацией задач, определением права на передачу данных и поддержанием сеанса в периоды неактивности приложений.
- 4. **Transport layer** (транспортный уровень). Самые известные его представители -- TCP и UDP, обеспечивает доставку данных от отправителя к получателю.
- 3. **Network layer** (сетевой уровень). Занимается логической адресацией узлов сети. Именно на этом уровне работает IPv4/v6
- 2. **Data link layer** (канальный уровень). Обеспечивает физическую адресацию компьютеров сети.
- 1. **Physical layer** (физический уровень). Работа с физической средой, отправка битов

информации через провод, оптический или радиоканал.

Стек TCP/IP

В современных сетях (где-то с 80 годов 20 века) используется стэк протоколов TCP/IP.

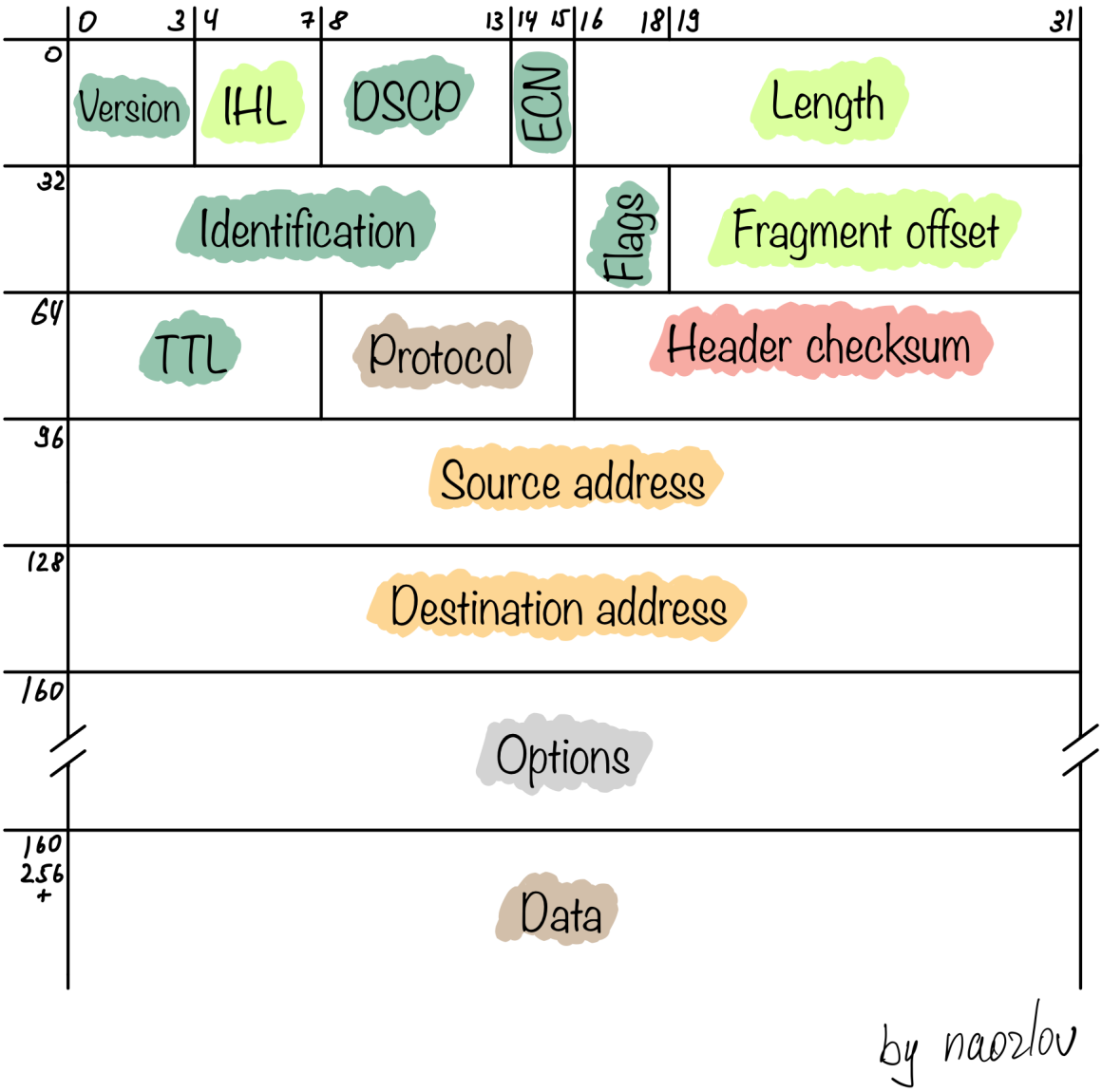


Состоит из 4 уровней:

- 4. ***Application*** -- уровень приложений
- 3. ***Transport*** -- уровень протокола передачи данных (TCP, UDP)
- 2. ***Internet*** -- уровень логический и физической адресации (IP)
- 1. ***Network access*** -- физический уровень передачи информации.

IP протокол

Протокол, выполняющий адресацию в сетях.



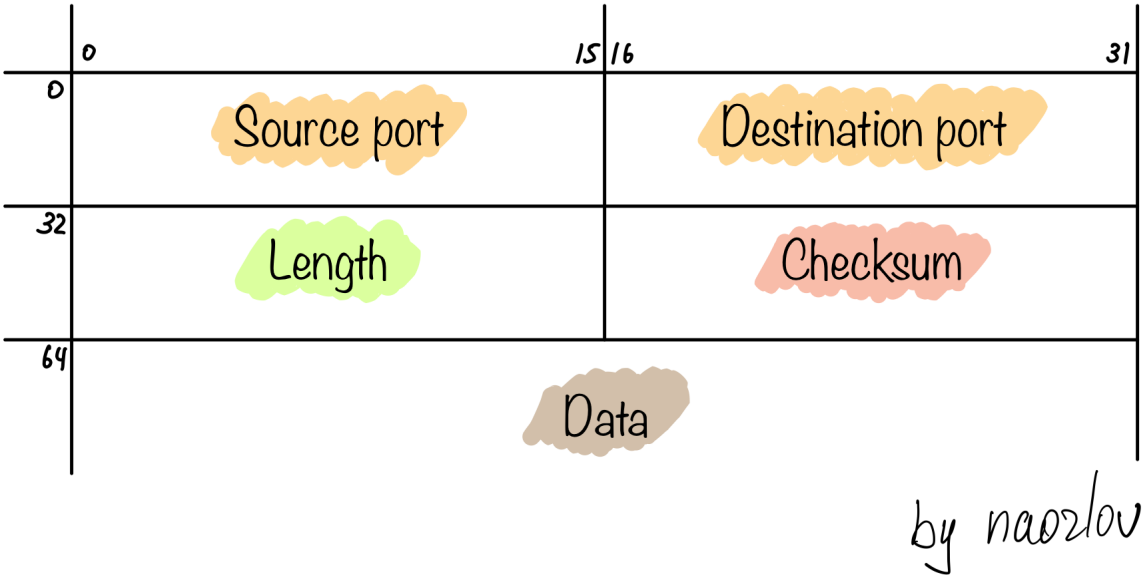
Цветами обозначены различные группы полей. Зеленый -- управляющие поля, необходимые для работы протокола, салатовый -- различные длины, оранжевый относится к адресации, красный -- к проверке целостности и коричневый -- к данным, содержащимся внутри пакета.

UDP/TCP протоколы

Из всей модели OSI нас сейчас интересует 4 уровень. Именно на нем работают TCP и UDP -- основные протоколы, через которые идет передача данных в сетях. Задача этих протоколов -- доставить данные от отправителя к получателю, возможно предоставив какие-то гарантии по этой самой доставке. Для программиста интерфейс этих протоколов -- *соединение*, с помощью которого происходит отправка данных. Полезная нагрузка протоколов это *байты* -- им не важна семантика данных. Оперируемая единица -- *датаграмма* или *сегмент* (подробнее об этом ниже).

UDP

User Datagram Protocol -- более простой из упомянутой пары протокол. Перед ним не стоит задачи доставить данные *надежно* или *в том же порядке*, в котором данные отправлены. Его цель -- попытаться доставить пакет, и, если пакет был доставлен, проверить целостность. Оперирует UDP *датаграммами* -- специальными сообщениями.

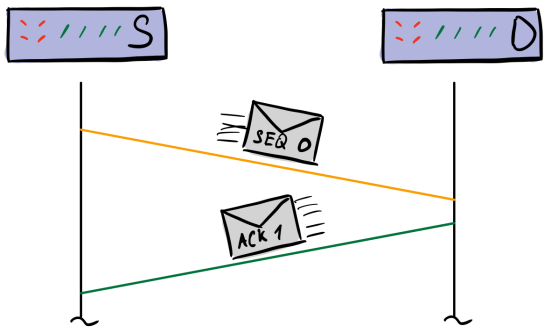


Датаграмма состоит из небольшого количества полей

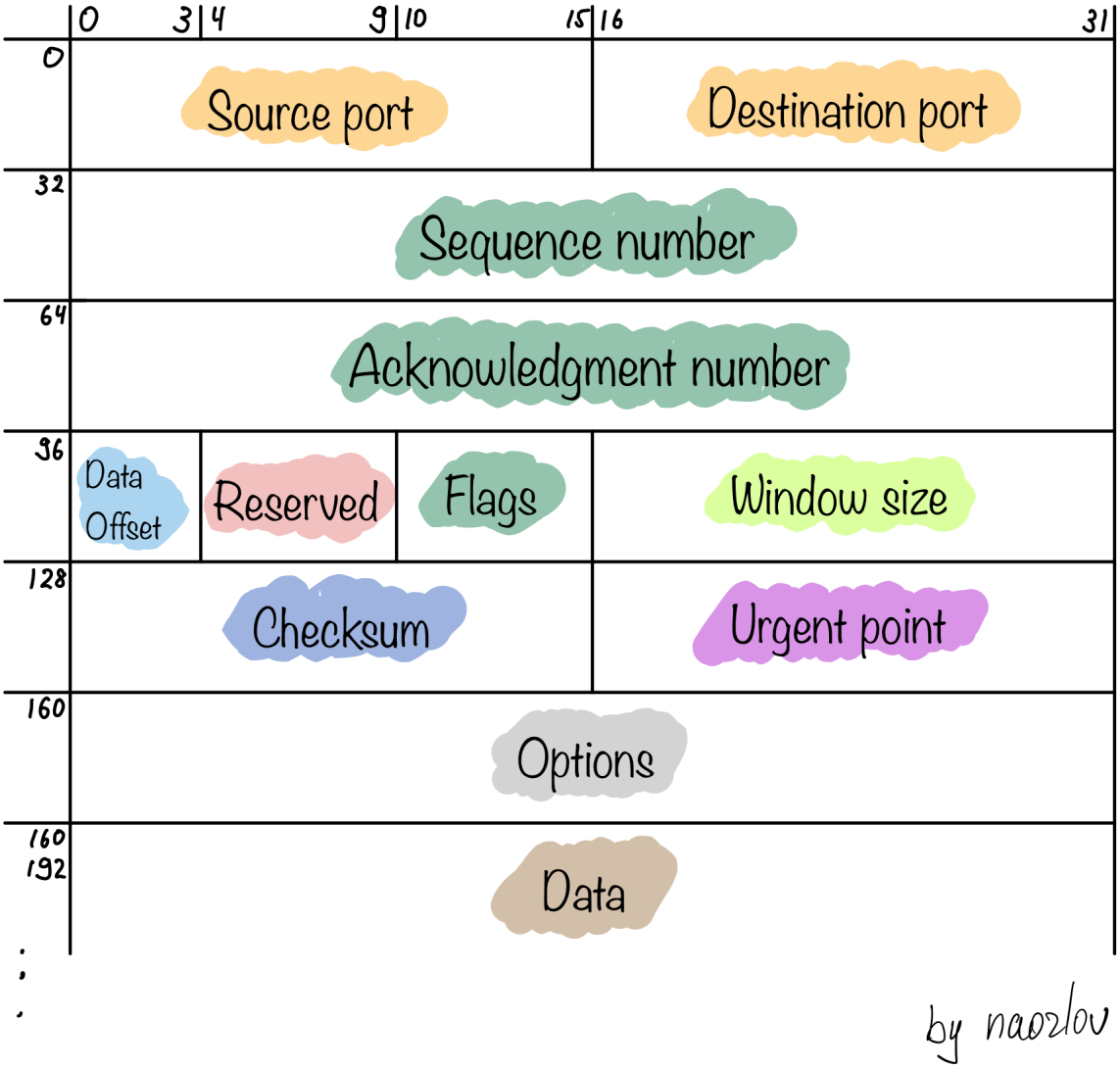
- Source port и Destination port -- два 16 битных числа, отвечающих за порт отправителя и получателя соответственно.
- Length -- 16 битное число, задает длину всей датаграммы (включая заголовок) в байтах. Минимальное корректное значение -- 8, теоретически максимальное -- 65535, фактический максимум, если на 3 уровне используется IPv4 -- 65507 (8 байт уходит на заголовок UDP, еще 20 уходит на заголовок IP пакета)
- Checksum -- контрольная сумма. При использовании IPv4 не является обязательным полем.

При отправке датаграммы нет никаких гарантий доставки -- **пакет может потеряться по дороге**. Взамен, протокол обеспечивает очень быструю отправку данных -- как мы увидим дальше, не происходит никаких лишних телодвижений. В UDP ничего особенно интересного нет -- интересующиеся могут почитать соответствующий RFC.

TCP



Transmission Control Protocol уже является протоколом надежной доставки сообщений -- он гарантирует, что *когда-нибудь* данные придут в том же порядке, в котором они были отправлены.



Мы не ставим себе задачей объяснить полностью протокол TCP -- заинтересованные могут обратиться к первоисточнику. Здесь же мы дадим описание протокола в первом приближении, а именно -- его основные идеи, позволяющие ему быть более надежным протоколом передачи данных.

Поля имеют следующее описание:

1. Source и Destination port -- 16-битное беззнаковое число, определяющее номер порта отправителя и получателя, на котором происходит обмен данными
2. Sequence number -- номер по порядку первого октета (байта) пакета во всем потоке данных для этого пакета. Для наших целей мы предположим, что размер данных в каждом пакете -- 1 байт, так будет проще следить за sequence номерами пакетов.
3. Acknowledgment number -- номер октета в потоке данных, который был принят отправителем пакета с выставленным *флагом* ACK
4. Flags -- 6-битное поле, в котором выставляются управляющие флаги
5. Data -- данные, передаваемые в этом пакете
6. Window size -- количество октетов, которые готова принять сторона протокола. Имеет значение только в ACK-пакетах (с проставленным Acknowledgment number и флагом ACK).

Остальные поля сейчас не имеют для нас особого значения, их назначение можно узнать более подробно в первоисточнике.

Протокол поддерживает *полнодуплексное* соединение -- данные можно как и отправлять, так и принимать каждой стороной соединения. Для простоты мы обозначим двух участников как *отправитель (инициатор, клиент, sender, source)* и как *получатель (сервер, destination)*.

Флаги

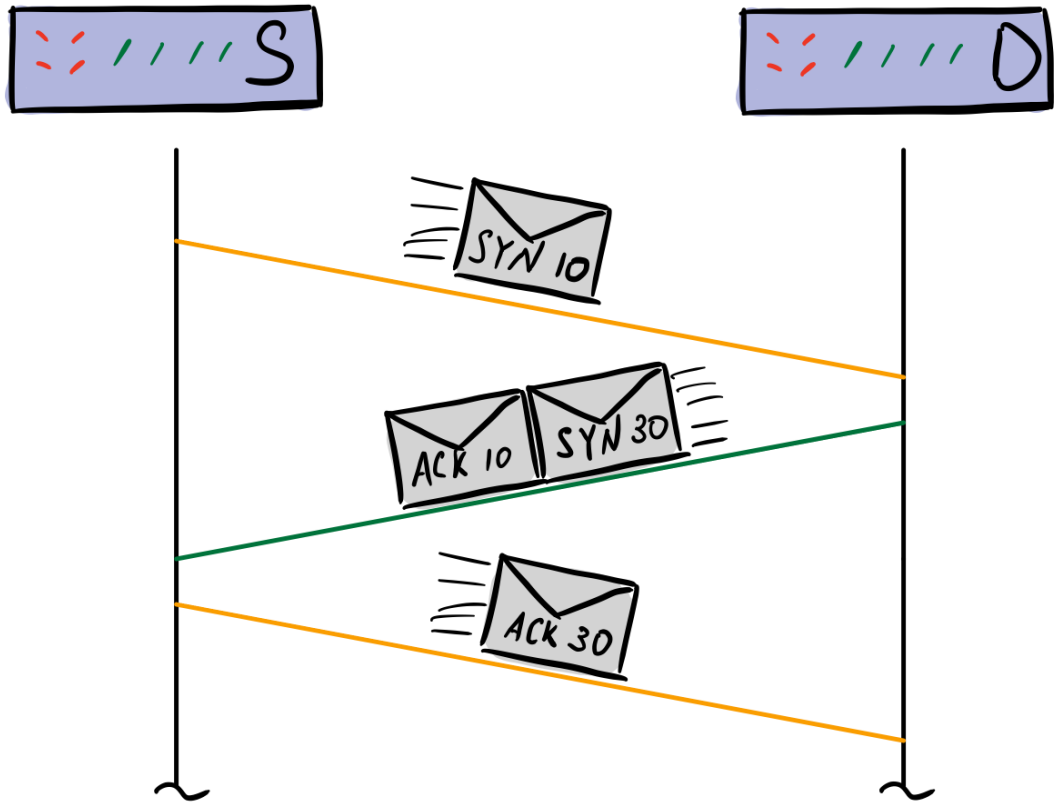
Всего есть 6 флагов, среди которых нас интересуют два: SYN и ACK флаг.

SYN (*synchronize sequence numbers*) -- флаг, используемый в начале соединения. Пакет, в котором установлен SYN-флаг назовем SYN-пакетом. Запись SYN 10 означает пакет, в котором в поле Sequence number стоит число 10, а в поле флагов установлен бит "synchronize sequence numbers".

ACK (Acknowledgment field significant) -- флаг, говорящий о том, что поле "Acknowledgment number" является значимым в этом пакете, и что получатель должен обработать получение пакета с таким флагом. Назовем такой пакет ACK-пакетом. Запись ACK 10 означает пакет, в котором в поле Acknowledgment number стоит число 10, а в поле флагов установлен флаг "acknowledgment field significant". Запись ACK 10 WIN 20 означает ACK 10 пакет, в котором дополнительно в поле Window size проставлено значение 20.

Пакет, в котором не проставлен ни один из этих флагов мы назовем SEQ-пакетом. Запись SEQ 10 означает пакет, в котором в поле Sequence number стоит число 10 и в поле флагов ничего не установлено.

Инициализация соединения



Для инициализации соединения используется "трехстороннее рукопожатие" (three-way handshake) -- алгортим, при котором происходит синхронизация sequence number у обоих участников соединения.

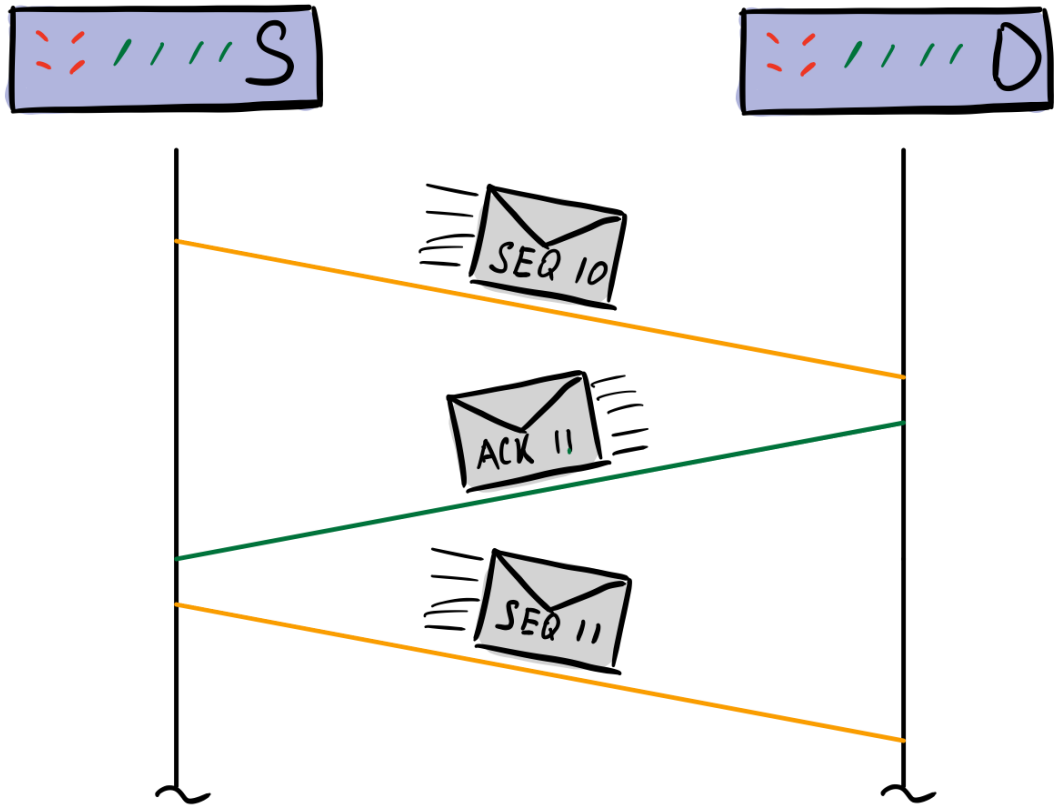
Отправитель (инициатор соединения) выбирает начальное значение поля sequence number (это число называется initial sequence number -- ISN), и отправляет пакет с этим числом получателю. Для простоты примеров все ISN положим равными 10 для отправителя и 30 для получателя.

Получатель получает пакет с ISN=10 от отправителя и в ответ отправляет ему два пакета: один это ACK-пакет со значением 10 в поле Ack number, и SYN-пакет с ISN=30. Клиент, получив ACK 10, понимает, что с его стороны соединение установлено, но теперь он должен сам подтвердить SYN 30 пакет от сервера. Делает он это отправкой ACK 30.

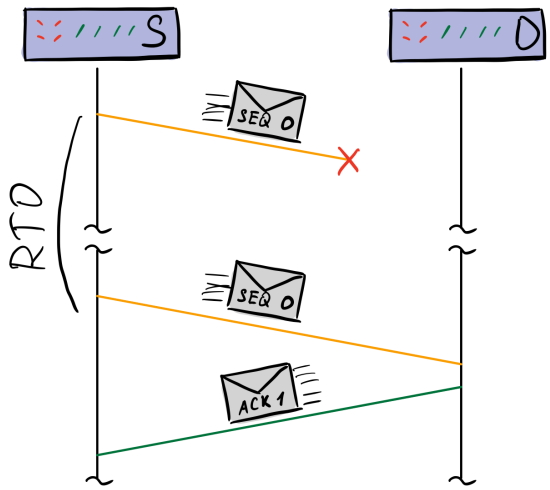
После отправки этого сообщения клиент может начать отправлять данные серверу и наоборот.

Примечание: в настоящем TCP ACK 10 и SYN 30 объединяются в один пакет, что помогает снизить нагрузку на сеть и также объясняет, почему three-way handshake так называется.

Еще примечание: в наших примерах стороны обмениваются сообщением размером в байт, так как это упрощает понимание протокола.



В случае отсутствия отказов отправка данных является очень простой операцией. Клиент отправляет пакеты, постоянно увеличивая SEQ-номер. Сервер в момент получения упорядочивает пришедшие пакеты в порядке возрастания номера, отдавая пользовательской программе данные в нужном порядке. Также, на каждый пакет сервер отправляет ACK пакет с SEQ номером, следующим за последним полученным от клиента.



Теперь допустим, что клиент отправил сообщение серверу, а сообщение потерялось где-то по дороге. Тогда клиент ждет некоторое время и повторно отправляет тот же пакет серверу. Это время называется *retransmission timeout, RTO*. Это время постоянно обновляется, в зависимости от текущего состояния сети.

Одна из формул пересчета RTO, предложенная в стандарте:

$$T_{SRTT} = \alpha \times T_{SRTT} + (1 - \alpha) * T_{RTT}$$
$$T_{RTO} = \min\big(T_{upper}, \max(T_{lower}, \beta \times T_{SRTT})\big)$$

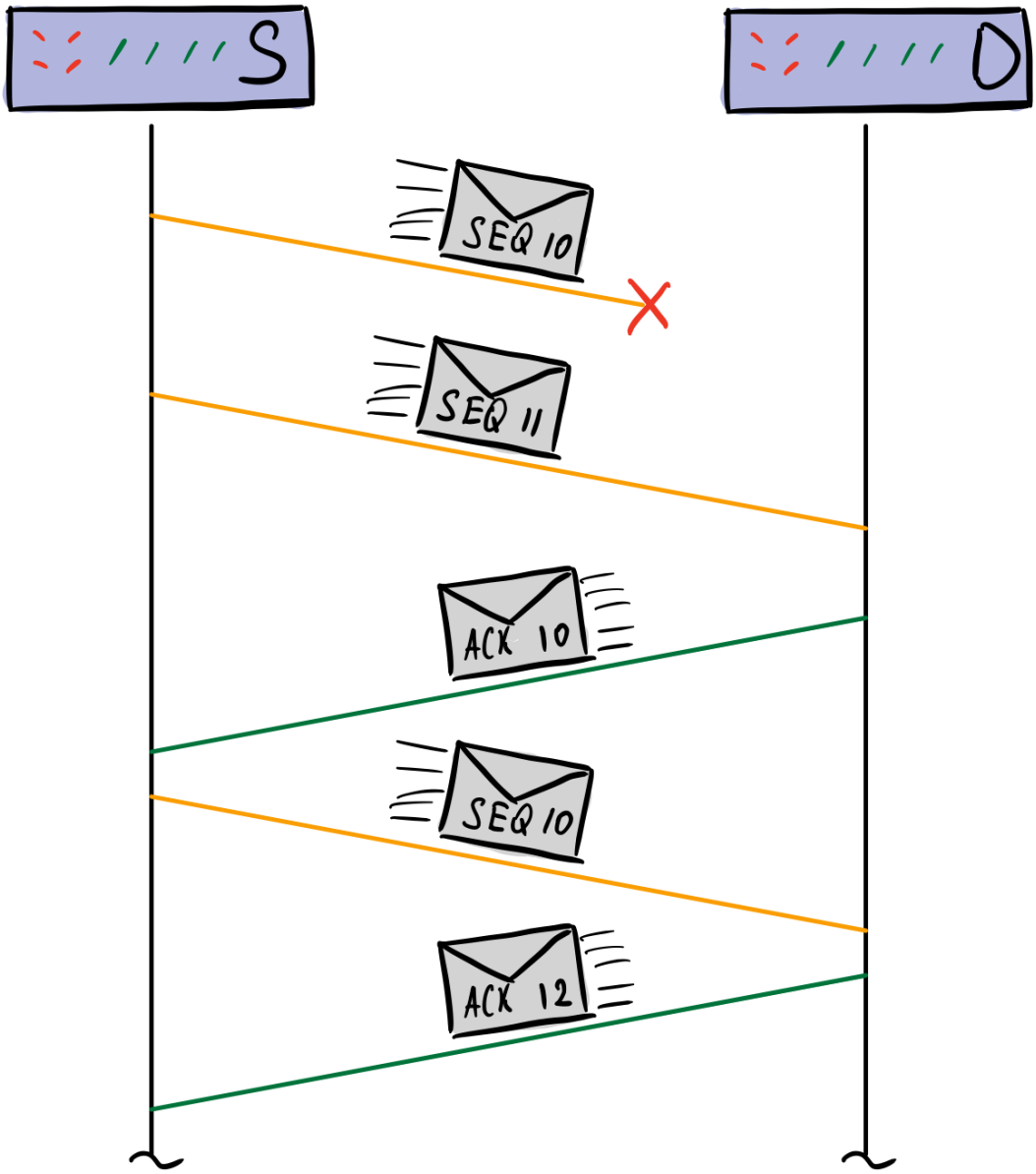
Здесь:

- T_{RTT} -- *Round-trip time*, время, прошедшее между отправкой последнего пакета и получения подтверждения
- T_{SRTT} -- *Smoothed Round Trip Time* , скользящее среднее время
- α -- степень сглаживания
- β -- величина задержки RTT
- T_{upper} -- верхняя граница на величину RTO
- T_{lower} -- нижняя граница на величину RTO
- T_{RTO} -- величина RTO.

Конкретный метод пересчета является implementation-defined.

Для отправки данных можно не дожидаться ACK от получателя. В этом случае, отправитель просто получит все подтверждения на все отправленные сегменты.

Представим ситуацию на картинке:



Отправитель отправляет два пакета, первый из которых теряется. Получив пакет SEQ 11, получатель отправит в ответ ACK 10 пакет, так как он ожидает, что следующий пакет придет с SEQ 10. Отправитель видит этот пакет, досылает SEQ 10 пакет, и в ответ получает ACK 12 -- подтверждение обоих пакетов.

В базовом TCP, описанном в стандарте RFC 793, ACK-сегменты отправляются в ответ на каждый полученный сегмент. В октябре 1989 года вышел RFC 1122, описывающий спецификации к интернет хостам, работающим на стеке TCP/IP. Там было предложено несколько оптимизаций протокола, направленных на уменьшение количества отправляемых ACK-сегментов. Информацию об этом можно почитать в первоисточнике. Вкратце -- можно подтверждать получение либо после каждого второго пакета, либо по истечение 0.5 секунд с последнего отправленного пакета.

Первая редакция TCP содержала внутри себя механизм для снятия нагрузки с получателя данных (flow control). Работает это следующим образом: при отправке ACK пакета соответствующая сторона сообщает другой стороне размер *окна* -- количество октетов, которое она может принять и спокойно обработать. Получатель же должен уважать данное поле -- не отправлять в сеть данных больше, чем текущий window size.

В случае, если отправитель получил ACK x WIN 0 пакет (то есть получатель не может обрабатывать больше данных), отправитель переходит в ожидание. Раз в несколько минут он отправляет однобайтовый пакет, проверяя, увеличился ли window size. После успешного увеличения, отправитель продолжает отправку данных.

Тут будет красивая картиночка, но я пока не придумал

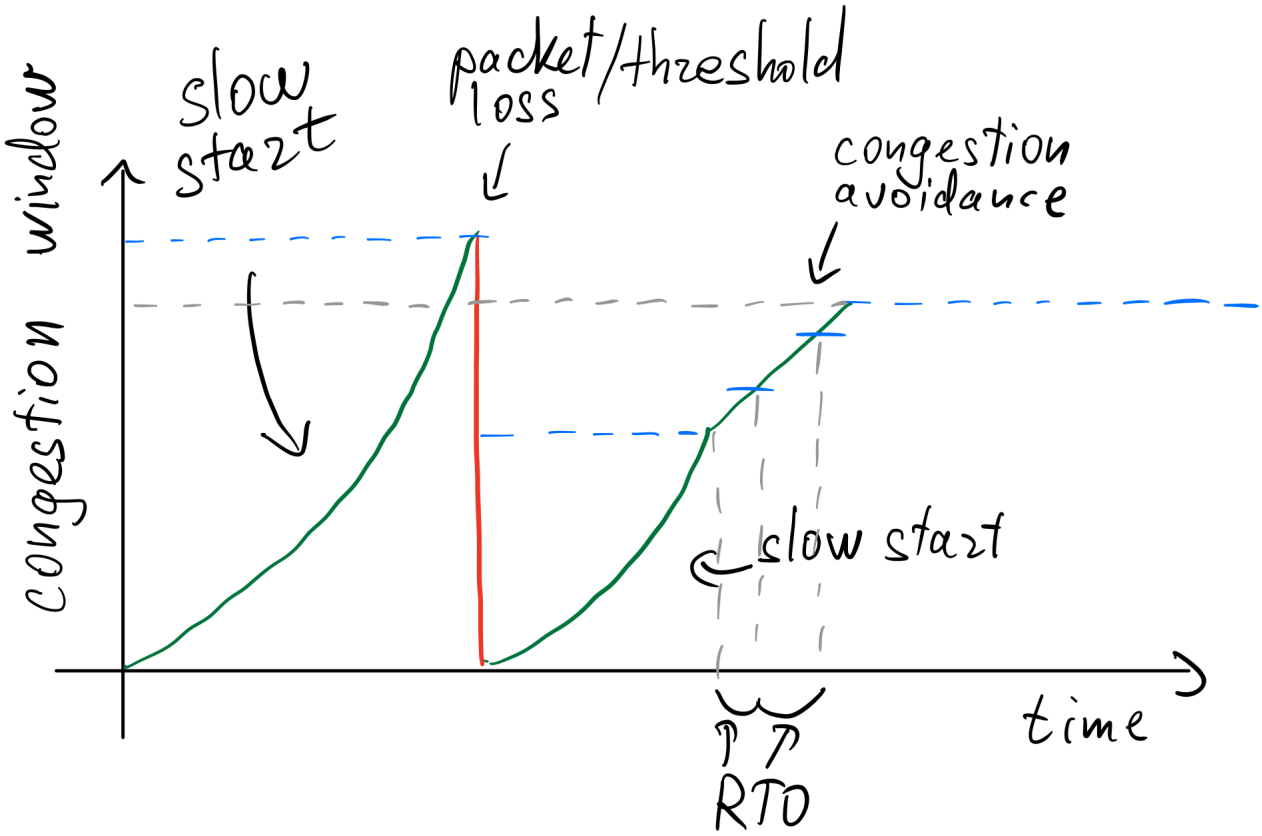
В какой-то момент к TCP также добавился механизм снятия нагрузки *с сети* (congestion control). Есть несколько алгоритмов для поддержания нормального уровня загрузки сети, здесь мы расскажем о slow start. У всех них одна задача -- не отправлять в сеть пакетов больше, чем она может через себя пропустить.

Для начала клиент выбирает congestion window size (cwnd), равный какому-то значению от 1 до 10 MSS (maximum segment size). Затем, клиент "забивает" весь доступный ему канал пакетами, отправляя сразу CWND пакетов. Например, пусть клиент выбрал 4 MSS в качестве CWND, тогда он отправит сразу пакетов столько, сколько влезет в выбранное количество байт.

На каждый ACK пакет клиент увеличивает CWND на 1. Так как он одновременно отправил несколько пакетов, и на каждый (или каждый второй) придет ACK, за примерно один RTT произойдет удвоение CWND.

Самое интересное происходит, когда происходит потеря пакета. Здесь есть два алгоритма: Tahoe и Reno.

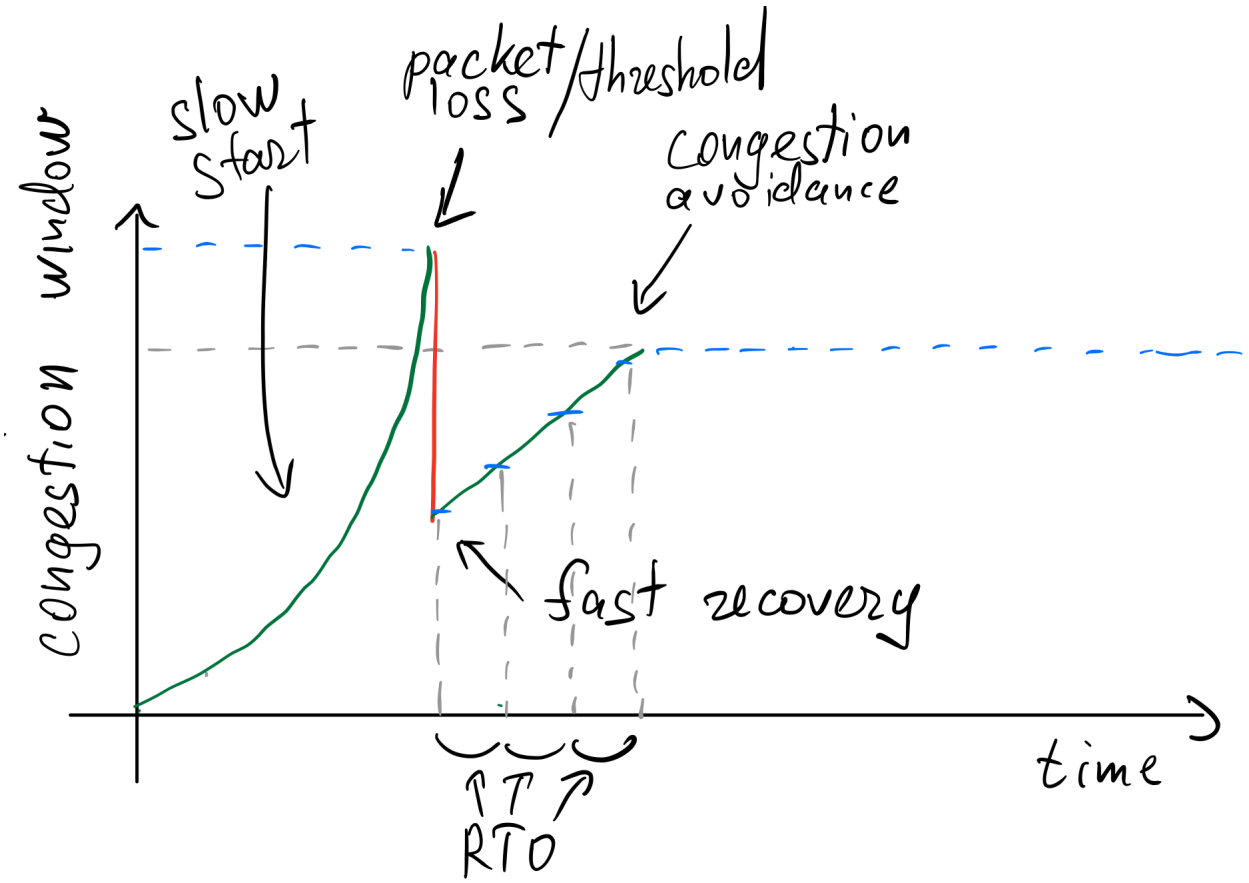
Tahoe работает следующим образом:



Сначала удовение каждый RTT, затем обнуление CWND, уполовинивание верхнего порога CWND, и повторение процедуры с начала.

После этого, если нет потерь пакетов, после достижения верхнего порога (ssthresh), каждый RTO ssthresh увеличивается на 1 до следующего момента потери пакетов, и тогда все начинается с начала.

Reno работает чуть быстрее:



После потери пакета вместо сбрасывания CWND в ноль, Reno устанавливает CWND в половину предыдущего ssthresh, и сразу начинает применять congestion avoidance.

Более полное описание этих механизмов можно встретить либо в стандарте, либо в материале [Understanding TCP internals step-by-step]

Полезные ссылки

- https://www.quora.com/What-is-synchronous-and-asynchrounous-in-distributed-systems
- https://en.wikipedia.org/wiki/OSI_model
- Grigorik I. High Performance Browser Networking. O'Reilly, 2013. (главы 1-3)
- Kleppmann M. Designing Data-Intensive Applications. O'Reilly, 2017. (глава 8, Unreliable Networks)
- Dordal P. An Introduction to Computer Networks
- Peterson L., Davie B. Computer Networks: A Systems Approach
- Kurose J., Ross L. Computer Networking: a Top Down Approach
- Networking tutorial
- Network Protocols (for anyone who knows programming language)
- TCP Puzzlers (Dave Pacheco, 2016)
- Understanding TCP internals step-by-step