Итак, давайте разберемся с протоколами TCP и UDP, с протоколами 4-го уровня. Это дает хорошее понимание того, каким образом устроен этот четвертый уровня, и каким образом работают протоколы более высоких уровней. И, возможно, это может вам помочь в вашей работе, потому что многое сетевое оборудование, например, работает только на очень низкоуровневых протоколах, не понимает никаких http запросов, никаких высоуровневых запросов, имеют любо свой какой-то специализированный механизм управления, либо как-то работает там путем передачи каких-то своих протоколов поверх TCP.

Соответственно, TCP и UDP — это 2 разных протокола. Их объединяет то, что они оба транспортные, они содержат непосредственно данные, которые передаются — вот эти самые потоки байт. Потоки байт, чем удобны - их можно разбить в любом месте, это на совести протоколов TCP и UDP, как эти потоки разбиваются. Соответственно, тоже, естественно, настраивается, но у них есть механизм для того, чтобы если вы, например, передаете файл в 2 гигабайта — это не один какой-то большой поток, большой пакет, который вы просто засовываете в сетевое оборудование и одним потоком передаете. Это какая-то, как правило, нарезанная колбаса из данных, и по кусочкам она передается. Потому что большие данные могут побиться, потеряться. Опять же, роутеры не рассчитаны на то, чтобы принимать огромные пакеты, и существует ещё несколько ограничений на более низких уровнях. Соответственно, вы эти пакеты разбиваете для того, чтобы передавать по сети.

TCP и UDP работают на уровне конкретных машин. То есть у вас, при отдаче, одна машина все собирает, и при приёме – другая машина эти все пакеты тоже собирает. Интернет про то, что вы записали в TCP и UDP, ничего не знает. То есть интернету, как системе передачи данных, важнее IP-адресация, это более низкий уровень. И, соответственно, важный концепт, который появляется именно на уровне транспортных протоколов, — это порты приложений. То есть в системе у

вас существует циферки от 1 до 65000, которые называются портами. То есть порты — это такая условная адресация того, какое приложение у вас будет принимать этот запрос. Например, у вас написано в протоколе, что я хочу попасть на порт там тысячу. Это значит, что при приеме на компьютер, система операционная будет искать, значит, приложение, которое слушает – так называется, то есть слушает этот порт 1000, и туда передаст данные. Если такого приложения на порту нет – операционная система скажет: «Не могу, ничего не знаю». У протоколов прикладного уровня уже есть стандартные определенные порты. У протоколов транспортного уровня порты могут быть любые. То есть вообще, они могут быть, в принципе, любые. Просто есть какие-то стандарты уже. И соответственно, вот этот первый уровень, на котором появляется такая информация о том, что приложение содержит в себе порты.

Мы поговорили о том, что есть в этих протоколах общего, а теперь давайте посмотрим на их отличия. Соответственно, самое главное отличие TCP и UDP в том, что TCP спроектирован таким образом, чтобы гарантировать доставку данных. То есть вот у вас есть какой-то объем информации, скажем, 10 пакетов этой информации. И они идут через сети. В сети может случиться, что угодно: между двумя роутерами нет связи, птицы нагадили на чердаке, у вас на последнем пакете отрубился коннект, у вас перебои в сети, не знаю, ну что угодно. То есть между вами и адресатом какое-то есть расстояние, и на этом расстоянии могут случаться всякие вещи.

TCP содержит в себе механизмы контроля, механизмы доставки данных. После получения пакета, другой компьютер отвечает, получил он пакет или нет. Если он его не получил, TCP доставляет пакет ещё раз. Следовательно, другой компьютер получит все пакеты, и они будут в том порядке, в котором вы их отправили.

Соответственно, UDP не гарантирует доставку данных. UDP просто швыряет пакеты через сеть. И соответственно, это значит, что, если у вас какой-то один пакет потерялся или два, то UDP на это вообще-то плевать. Он будет продолжать кидать пакеты без всякой проверки доставки и прочего. Соответственно, за счёт того, что они реализованы по-разному, размеры пакетов тоже разные, в UDP гораздо меньше служебной информации для того, чтобы как раз гарантировать доставку данных. В TCP этой информации гораздо больше. Размеры, то есть размеры дополнительные в пакетах, помимо ваших данных, они отличаются. Соответственно, это, естественно, обуславливает разницу в скорости: TCP медленный, TCP просит подтверждение. То есть как минимум 2 раза у вас сигнал ходит туда-сюда, уже как минимум. Ещё не говоря о том, что это все надо расшифровать и обработать. UDP ничего не требует – UDP кинул в одном направлении, типа: «Петька, лови топор». Соответственно, он быстрый.

И все эти атрибуты как бы обуславливают их применение. TCP используют там, где разработчикам не нужно морочиться с…, не нужно, не хочется или невозможно морочиться с целостностью данных. Они это оставляют на уровне протокола. То есть, если вы, например, передаёте что-то в API, и у вас там какой-то бинарный формат – у вас все биты должны на месте стоять, иначе вы ничего не расшифруете. Если вы какой-то подписываете сертификат электронным ключом – в электронном ключе все биты должны стоять на месте, иначе он не расшифруется. Ну и так далее. То есть это электронная почта, это загрузка файлов, это работа с API, работа, соответственно, с http, загрузка сайтов и прочее.

UDP быстрый, UDP может потерять немножко пакетов, но это не страшно для UDP. Соответственно, UDP используется там, где вам нужно что-то стримить, где у вас есть какой-то поток данных. И не важно, если вы от этого потока потеряли пару байтов. Например, если вы стримите видео – не важно, если вы потеряли от этого видео пару пикселей. То есть вы можете сделать такой протокол, который

будет обеспечивать стриминг-видео даже с потерей какого-то количества пакетов. То же самое с голосом. То есть то, что что вы смотрите на (НРЗ 06:48), в YouTube-е, вот этот стриминг, он, как правило, тоже работает через UDP. Потому что через TCP – это слишком медленно.

Давайте разберемся с протоколом TCP. А потом посмотрим на протокол UDP. Соответственно, TCP работает по (НРЗ 07:04), поэтому вот он такой вот TCP IP. Соответственно, что здесь у нас есть? Здесь у нас протокол низкого уровня, поэтому все разбито-побито. Это не какие-то структуры данных, но вы можете их расценивать, как такой struct в Go, в котором есть какие-то определенные поля. Соответственно, порт источника из 16 бит, порт назначения из 16 бит, поэтому они у нас не могут быть в системе больше, чем 65000. Да, и в общем-то даже по современным меркам этого хватает. Потом у нас есть порядковый номер, который позволяет на другом конце восстановить, если вдруг один пакет пришел быстрее другого. У вас такое в сети может быть спокойно совершенно. Номер подтверждения, соответственно, под каждый пакет приемник должен отправить отправителю подтверждение. Флаги, длина пакета, соответственно, сами данные и контрольные суммы. Здесь мы можем видеть схему, значит, работы TCP, что у нас открывается здесь соединение. Сервер у нас слушает постоянно, поэтом, когда клиент что-то у него запрашивает, открывается соединение. После установки соединения, соответственно, передаются данные до пакетов FIN. То есть, если у вас длинная строка данных, у вас передается все это…, как бы она бьется по частям, передаются эти части, а в конце передаётся FIN. Это значит, что пакет кончился и надо его обрабатывать. Потом, значит, соответственно, после передачи всего этого пакета, соединение закрывается.

Все это здесь описано, значит, с флагами. Здорово, круто. И на самом, деле совершенно нам не нужно, потому что здесь как раз написано, что реализация TCP обычно встроена в ядра операционной системы. То есть в вашей работе

очень вряд ли вам понадобится имплементировать протокол. Поэтому давайте уже посмотрим непосредственно, как он реализуется в Go. И что вам нужно, собственно говоря, сделать, чтобы поднять TCP-сервер, и чтобы сделать TCP-клиента, который будет подключаться к этому TCP-серверу и что-то там передавать.

Ну что ж, начнём с TCP-сервера. Все, связанное с интернетом, у нас находится в пакете NET. И соответственно, здесь, что происходит? Здесь установится соединение сервера. То есть занимается в данном случае порт. То есть как это происходит на более низком уровне - на самом деле, здесь можно сейчас, если мы углубимся в исходный код, то здесь как раз будет вызов VIN API в данном случае. Здесь мы, наверное, остановимся. Дальше идут системы специфические штуки. То есть система ваша операционная – не важно, это Windows или Linux какой-то, ну если вы только не на чем-то совсем экзотическим сидите, чего я не знаю, имеет в себе socket-ы. То есть это такие как бы почтовые ящики, в которые вы кладете сообщения. И из них забираете сообщения.

И система имеет эти socket-ы разных типов. То есть, если в системе есть имплементированный стек TCP/UDP, значит, будет TCP socket, UDP socket. Если нету, то будет IP socket. То есть socket более низкого уровня, в которой вы уже засовываете данные, обернутые вашим протоколом. А он их оборачивает в IP протокол и отправляет дальше. Таким образом вам, как разработчику, не нужно думать, что вам там…, когда что-то по сети надо передать, то значит: «Так-так, что там у нас, MAC-адреса, это надо драйвер опросить». Вам нужно просто положить в TCP- socket. И TCP- socket сам обернёт во всю эту периферию, которую только, что перечислил. Вот это из Википедии, вот это вот все обернет в порты источника, порты назначения, разобьет сам данные по размеру. Этот размер обычно в системе настраивается. И все, что вам нужно сделать, это передать адрес, который вы хотите выслать, и указать тип, то есть sock_stream — это тип TCP. Это

значит, что у вас открывается соединение. И туда передаются данные с подтверждениями. И соответственно, указываете ему режим. Режим у вас может быть лишь Listen или Write. В принципе, здесь дальше…, да, уже идут вызовы socket-ов. Если интересно, можно с этим разобраться. Я не вижу какое для этого есть применение, когда нам просто нужно этим пользоваться.

Соответственно, после вызова Listen, мы получаем ссылку на Listener. Тут написано, что это такой стандартный сетевой Listener, для протоколов ориентированных стримы. То есть на потоке данных туда и обратно. То есть TCP — это протокол, ориентированный на стримы. Значит, что мы туда можем запихнуть стрим байтов и прочитать оттуда стрим байтов. То есть мы не делим то, что мы засовываем туда какие-то пакеты, мы не делим их на части. Мы туда что-то впихиваем и что-то оттуда забираем. Этот метод будет работать у нас для обоих протоколов, забегая вперед. Дальше мы проверяем на ошибки, закрываем Listener, иначе у нас будет на порте висеть наш хук. То есть в нашем…, тут я…, здесь вижу сейчас в IDE-шке, и здесь не страшно. Если вы сидите в реальной системе, у вас может такое остаться, что у вас приложение висит на хуке и не закрывается. Придется его убивать, если вы не закроете соединение.

И дальше у нас бесконечный цикл. То есть в данном случае этот цикл, если мы смотрим на эту схему, то это вот этот вот переход из closed в closed. Слушание, так называемое. То есть мы данном случае, как сервер, мы в бесконечном цикле пассивно ждем, пока к нам кто-то подключится. Это, в принципе, основа любого сервера. То есть любой сервер в бесконечном цикле опрашивает порт свой и ждет, пока к нему что-то придет.

Как только к нему что-то пришло – с помощью метода Accept мы получаем соединение. Соединение – этого интерфейс с методами read и write, и пачкой других. Соответственно, методы read и write нам позволяют предположить, что это

соединение имплементирует в себя как раз стрим. Как раз тот самый reader и writer, которые в golang-е обозначают поточное чтение и поточную запись. Так получается, соответственно. После того, как мы получили это соединение успешно, мы вызываем goroutine-у, чтобы наш основной цикл продолжал слушать, чтобы наш сервер мог работать в мультиплексном режиме, то есть принимать несколько соединений. И в нашей goroutine-е мы делаем буфер, опять же, стрим, поток байтов — значит, куда-то его надо считать, куда считать? Естественно, в slice.

И с помощью метода read, мы читаем какую-то длину этих самых байтов в буфер. То есть метод read, ну стандартный метод reader-а любого. И поскольку у нас есть метод write, опять же, мы можем сразу же здесь ответить как-то. И можем закрыть соединение. А можем не закрывать соединение, можем ещё что-то туда дописать. Или ещё что-то оттуда прочитать. Это зависит уже от нашего протокола.

То есть, в принципе, ну как вы видите, это довольно простая штука, она довольно простая, потому что у протокола TCP, на самом деле, не настолько уж много настроек. Это реально протокол транспортный. Его не интересует, что вы туда положите, он никак этим не управляет. Если вы, допустим, скажете, что у меня такой формат, что мой сервер слушает первое сообщение, на него как-то отвечает, а, по второму сообщению, он, значит, не знаю…, лезет в базу данных, что-то там меняет и ещё что-то отвечает - то вы уже пишете протокол прикладного уровня, состоящий из этих двух сообщений. И таким образом мы можем видеть, что работа с TCP со стороны сервера довольно простая.

Давайте посмотрим, как это работает в реальной жизни. Для проверки моего сервера, я использую программку Hercules, «Геркулес», «Херкулес» – не знаю. В общем, эта программа написана какими-то ребятами, которые тестируют много низкоуровневого оборудования. Соответственно, у них здесь есть поддержка

TCP-клиента и TCP-сервера, сериализованных данных каких-то. Достаточно удобная штука для того, чтобы третировать ваши сервера. Соответственно, я запустил свой сервер. Что я от него буду ожидать?

Соответственно, когда к нему приходит какое-то сообщение, он должен мне напечатать, сюда напечатать его длину. И соответственно, написать, что я принял это сообщение в connection. Написать, сколько бат, pipe-ов записано туда, и закрыть соединение. Давайте посмотрим, как это работает.

Я указываю localhost и порт 1234 — это мой порт здесь, для TCP. И, соответственно, я подключился. Сюда я могу что-то написать. Но лучше я напишу это (НРЗ 16:53). Соответственно, это однобайтовые какие-то сообщения. Если у вас есть какая-то мультибайтовая кодировка — это уже на вашей совести, как это делать, как это засовывать в протокол, в сами данные. И посылаем. Соответственно, мое сообщение пришло, оно получено. И я вижу, что вот этот messege received у меня занял 17 байтов для записи. То есть здесь у нас простое конвертирование в байты и простой вывод. Больше ничего. Особых никаких настроек у TCP-сервера нет.

Теперь давайте посмотрим на TCP-клиент к моему серверу, к вот этому TCP-серверу. И соответственно, разберемся теперь, как работать, с другой стороны, как вот эту программульку сэмулировать средствами go. Но клиент у нас вообще ещё проще. То есть здесь мы создаём наш slice из байтов. Мы используем команду NET.Dial. Это команда, устанавливающая соединение, открывающая соединение со стороны клиента. То есть мы таким образом можем обратиться к любому API. А если оно находится в рамках определенного протокола и stack-а. И соответственно, можем напечатать какое-то сообщение. Поскольку соединение, которое нам отдается в результате DIAL — это тот же самый connection с методами read и write, мы можем использовать по-другому. Можем вместо conn

write, написать print с помощью команды fnt, потому что print принимает writer. И io writer - ему достаточно интерфейса write.

Таким образом, мы можем использовать несколько методов записи в поток. Это все примерно тоже самое, что вызов conn write. И здесь какой-то форматированной строки. Только print F сам все переведет в байты. Ну и, соответственно, можем прочитать ответ после того, как записали приложение. Здесь нет никаких асинхронных вызовов. То есть, если вы пришли из другого языка, у которого есть поддержка, значит async, всякие future-ы и прочее – здесь нет. Вот это приложение, вот этот кусок кода, он выполняется строго последовательно. То есть между вот этим и вот этим у вас будет довольно продолжительное ожидание по меркам компьютера. Потому что у вас в этот момент уйдет сообщение по сети, будет принято сервером, сервер его обработает, прочитает и, соответственно, даст ответ.

Фишка в том, что, в DIAL-е у вас есть, так называемый, cancel context. То есть каждое соединение по умолчанию устанавливает у себя default-ный KeepAlive. То есть, если вы пишете сообщение, и ваш сервер не отвечает в течение 15 секунд, то TCP считает, что, значит, ваш…, вот этот DIAL считает, что соединение неудачно и его разорвёт, вернув ошибку. Если, соответственно, оно удачное, то вы прочитаете этот набор данных и закроете соединение. Соответственно, если у вас есть какой-то супер-специфический протокол, с большим объемом данных в пакетах – вам KeepAlive нужен несколько побольше. В принципе, стандартных 15 секунд для TCP-соединения обычно достаточно, потому что он считает на соединение. И если у вас всё там работает нормально, то у вас этого никогда не будет. Ну и соответственно, важно опять же не висеть на порту после этого, а закрыть соединение.

То есть, как мы видим, со стороны клиента это вообще супер просто. Нам здесь даже думать не надо особо ни о чем. Мы просто открываем, читаем наш поток байтов, пишем. Вернее, пишем, потом читаем наш поток байтов, закрываем соединение. Смотрим, что у нас получилось, что у нас клиент выдаст. Он выдает ответ сервера, то же самое, что у нас было здесь. После этого идёт куча пробелов. Куча пробелов у нас идет, потому что это такой padding go-шный для нашего массива байт. То есть у нас здесь куча нулей, стандартный наш slice. И соответственно, вот после последнего message read go вот так вот нам отобьет, отформатирует строку. Вот как-то так.

Давайте теперь посмотрим, в чем разница с UDP. Как эта разница выражается в коде. И продолжим. Давайте теперь разбираться с UDP, с тем самым, быстрым протоколом. Здесь вот наш пакет. Как вы можете видеть тут внезапно почти ничего нет. Это как раз то, чем я говорил вначале. Что у нас, значит, TCP больше, UDP меньше. Вот можно видеть. Здесь из дополнительных данных только два порта, длина дейтаграммы и, значит, контрольная сумма. Всё. Все остальное — это данные. И никаких тебе флагов подтверждения, никаких тебе опций - ничего вообще здесь нету. Только данные, и всё. И можно увидеть, что называется это не пакет, как в TCP, а дейтаграмма. Уж не знаю, откуда вообще это повелось, но смысл в том, что дейтаграмму мы просто кидаем в сеть без даже какого-то, собственно говоря, подтверждения.

То есть сами socket-ы, которые это делают, то есть которые устанавливают одностороннее соединение, они называются дейтаграммными socket-ами. То есть они как бы принимают в себя сразу, выплевывают в интернет, и даже никаких подтверждений они не возвращают. Соответственно, в Go UDP-сервер выглядит на порядок проще TCP-сервера. Можно сравнить как бы. Ну даже ладно, порядок — это сильное слово, оно несколько проще. То есть в (HP3 23:17) происходит гораздо меньше, да. Вот здесь какие-то методы попроще.

То есть мы здесь, что делаем? Мы здесь составляем адрес. Вызываем команду listen UDP. Это, на самом деле, то же самое, что вызвать команду listen network UDP, вот таким адресом. Здесь просто показывают другой вариант, как это можно сделать. Передать, значит, соответственно, структуру. В структуре есть TCP-адрес, есть и UDP-адрес. Они поддерживают порты и IP.

То есть в нашем случае это, соответственно, наш опять же localhost. И соответственно, этот адрес, уже специфический, передается в метод listen UDP. Существует метод, соответственно, listen… Так, он существует не в методе UDP sock, он существует в методе TCP sock.

Соответственно, есть метод listen TCP. Вот, он здесь, соответственно. Точно также слушает, но принимает TCP-адрес. Это такой alias для удобства. Можете пользоваться и так, и так, как хотите. Мне нравится больше метод с listen. Он проще что ли выглядит. И соответственно после того, как мы передали listen UDP, теперь мы получаем connection. Но в данном случае получаем опять же обертку над connection, UDP connection. Для того, чтобы удобнее было работать с UDP. Здесь, что важно понимать? Да, здесь и вот уже говорил про socket-ы. Вот они, socket-ы, socket datagram – то есть это другой socket. Это socket опять же дейтаграммный, который выкидывает просто данные.

Соответственно, мы когда слушаем, если у нас прошло все без ошибки, мы опять же в бесконечном цикле просто читаем из socket-а. Нет у нас никакого акцепта соединения. Вот этого всего, что у нас в TCP было. Мы просто читаем из socket-а. Все время что-то читаем. Потом мы можем вызвать Response. Если что-то в socket-е есть, то есть здесь у нас есть такая, в некотором смысле проверочка о том, что сообщение прошло. То есть здесь мы прочитали что-то из socket-а, получили адрес и, соответственно, можем в этот адрес что-то написать. Адрес — это, соответственно, удаленный адрес, машиной собираем из IP-адреса и

UDP-порта назначения. И здесь мы можем просто ответить, что у нас сервер получил ваше сообщение, и отправить его по конкретному адресу с помощью write to UDP. Никаких проверок соединения, ничего нету. Если что-то не получилось - у вас сообщение просто не уйдет.

И давайте посмотрим теперь, как это выглядит с помощью Hercules. Хотя, на самом деле, зачем нам Hercules, когда клиент UDP-клиент от TCP-клиента не сильно отличается. То есть там под капотом, да, у вас (НРЗ 26:23-26:24), вот эти все машины состояния, TCP в UDP – ничего нету. Но какая, в принципе, разница для потребителя? Вы просто пишете сообщения в байтах, читаете сообщения в байтах. Всё. Соответственно, та же команда DIAL, тот же network UDP. Ну в смысле, не тот же. A network UDP, вместо TCP, адрес порт, те же ошибки, тот же reader и так далее.

То есть здесь мы опять же видим, что у нас отбивок больше. Сообщение, соответственно, от сервера - hello i got your message. Потому что сервер у нас его, соответственно, пишет вот здесь, при приходе сообщения. Значит, при отправке сообщения, мы точно так же prinr-уем в соединение. То есть здесь вообще никакой нет разницы, здесь у нас опять же, в принципе, все в порядке. И можно увидеть, что клиент UDP-шный выбирает рандомный порт. Система ему назначает порт — это не важно, потому что до той поры, до какой сервер знает, на каком порту висит клиент, он может к нему обратиться. Соответственно, адрес сервера у нас статичный.

В принципе, вот так как-то. Я надеюсь, что вам не часто придется работать с настолько низким уровнем протоколов. То есть здесь у вас есть, по сути, только одна функция — это передать массив байт внутрь. И это очень как бы низкий уровень. То есть все, что (НРЗ 28:05) – все эти протоколы, все кодирования, кэширования, безопасность, сертификаты какие-то там — это все будет делаться

вами на прикладном, так называемом, уровне. И очень надеюсь, что оборудование, которое понимает только TCP, IP, UDP запросы, у вас не будет.