**Алгоритм медленного пуска**

Медленный старт — один из алгоритмов, который TCP использует для контроля над нагрузкой сети. Иногда его ещё называют фазой экспоненциального роста.

Во время фазы экспоненциального роста, алгоритм медленного старта работает за счёт увеличения окна TCP каждый раз, когда получено подтверждение, то есть увеличивает размер окна в зависимости от количества подтверждённых сегментов. Это происходит до тех пор, пока для какого-то сегмента не будет получено подтверждение или будет достигнуто какое-то заданное пороговое значение.

Если происходит потеря, TCP предполагает, что это связано с перегрузкой сети и принимает меры по сокращению нагрузки на сеть. Как только порог достигнут, TCP входит в фазу линейного роста (предотвращения перегрузки). В эту фазу окно увеличивается на один сегмент для каждого RTT. Это происходит до тех пор, пока не происходит потери пакета.

**Алгоритм тройного рукопожатия**

Все TCP-соединения начинаются с тройного рукопожатия. До того, как клиент и сервер могут обменяться любыми данными приложения, они должны «договориться» о начальном числе последовательности пакетов, а также о ряде других переменных, связанных с этим соединением. Числа последовательностей выбираются случайно на обоих сторонах ради безопасности.

SYN. Клиент выбирает случайное число Х и отправляет SYN-пакет, который может также содержать дополнительные флаги TCP и значения опций.

SYN ACK. Сервер выбирает свое собственное случайное число Y, прибавляет 1 к значению Х, добавляет свои флаги и опции и отправляет ответ.

АСК. Клиент прибавляет 1 к значениям Х и Y и завершает хэндшейк, отправляя АСК-пакет.

После того как хэндшейк совершен, может быть начат обмен данными. Клиент может отправить пакет данных сразу после АСК-пакета, сервер должен дождаться АСК-пакета, чтобы начать отправлять данные. Этот процесс происходит при каждом TCP-соединении и представляет серьезную сложность плане производительности сайтов. Ведь каждое новое соединение означает некоторую сетевую задержку.

Создание TCP-соединений оказывается «дорогим удовольствием», поэтому повторное использование соединений является важной возможностью оптимизации любых приложений, работающих по TCP.

**Обеспечение надёжности**

Для обеспечения надежности доставки можно использовать следующие методы: хэш-функции, кода Хэмминга, контрольные суммы.

Хэш-функция. Вычисляем хэш-функцию от данных, отправляем её вместе с данными, получатель вычисляет свою хэш-функцию от полученных данных, сравнивает её с полученной хэш-функцией, если всё передалось успешно - они совпадут.

Код Хэмминга. Состоит из двух частей. Первая часть кодирует исходное сообщение, вставляя в него в определённых местах контрольные биты (вычисленные особым образом). Вторая часть получает входящее сообщение и заново вычисляет контрольные биты (по тому же алгоритму, что и первая часть). Если все вновь вычисленные контрольные биты совпадают с полученными, то сообщение получено без ошибок.

Контрольная сумма. Поле UDP, используемое для проверки заголовка и данных на ошибки. Если сумма не сгенерирована передатчиком, то поле заполняется нулями. Поле не является обязательным для IPv4. При получении сообщения получатель считает контрольную сумму заново. Если сумма не сходится (данные были повреждены при передаче, либо контрольная сумма неверно посчитана на передающей стороне), то решение о дальнейших действиях принимает принимающая сторона.

**Таймеры TCP**

**1. Таймер повторной передачи**

Чтобы контролировать потери или удаления сегментов, TCP применяет таймер, который контролирует время повторной передачи и время ожидания подтверждения сегмента. Когда TCP посылает сегмент, он запускает таймер повторной передачи для этого конкретного сегмента. При этом могут возникнуть две ситуации:

1. Если подтверждение получено для этого конкретного сегмента прежде, чем истечет контрольное время, то таймер стирается.
2. Если контрольное время таймера истечет прежде, чем придет подтверждение, сегмент ретранслируется, и таймер включится повторно.

**Вычисление времени повторной передачи**

TCP не может использовать одно и то же время повторной передачи для всех соединений. Если время повторения передачи не соответствует времени для достижения сегментом пункта назначения и подтверждение за определённое время не достигнет исходного пункта, то это может привести к повторной передаче сегмента, который еще в пути. С другой стороны, если время повторения больше, чем необходимо для короткого пути, это может привести к задержке прикладных программ.

Даже для одиночного соединения время повторения передачи не должно быть фиксированным. Соединение может послать сегменты и получать подтверждения быстрее в период отсутствия трафика, чем в периоды перегрузки. TCP применяет динамическое время повторения, которое отличается для каждого соединения и которое может изменяться в течение одного и того же соединения.

Время повторной передачи может быть выполнено динамически на базе времени передачи "туда и обратно" (round-trip time — RTT). С этой целью используется несколько формул. Наиболее общая формула для вычисления времени повторения передачи есть двойное RTT:

Время повторения передачи = 2 x RTT.

**Вычисление RTT**

RTT тоже вычисляется динамически. Имеются два метода. Первый метод — TCP использует опцию метки времени, когда при посылке сегмента и его подтверждении указывается время отправления. При втором методе TCP посылает сегмент, запускает таймер и ждет подтверждения. Это измерение времени между посланным сегментом и полученным подтверждением дает значение RTT.

Значение RTT используется в вычислениях времени повторения передачи — в соответствии со следующей формулой:

RTT = a \* предыдущее RTT + (1 – a) текущее RTT.

Значение "a" обычно 90 процентов. Это означает, что новое RTT – это 90 процентов предыдущего значения RTT плюс 10 процентов значения, текущего RTT.

**Вычисление RTT с учетом повторной передачи**

Предположим, что сегмент не подтверждён в течение времени повторения, и он все-таки передан повторно. Когда TCP передатчика получает подтверждение на этот сегмент, он не знает — подтверждение это на исходный сегмент или на повторный. Значение RTT поэтому вычисляется, базируясь на прибытии сегмента. Однако если исходное сообщение было потеряно и подтверждение пришло на повторную единицу, значение RTT будет вычислено от времени повтора сегмента. Это дилемма, которая решается следующим образом:

1. не определять RTT повторного сегмента в вычислении нового RTT;
2. не корректировать значения RTT, пока вам не послан сегмент и не получено подтверждение без необходимости повторной передачи.

**2. Время настойчивости**

Для того чтобы иметь дело с объявлением окна с нулевым размером, TCP необходим другой таймер. Предположим, TCP приёмника получил извещение о нулевом размере окна. TCP передатчика затем останавливает передачу сегментов, пока TCP приёмника не пришлёт подтверждение, извещающее о ненулевом размере окна. Это подтверждение может быть потеряно. Если TCP приёмника думает, что он отработал, и ждет, что TCP передатчика пришлет еще сегменты, TCP передатчика, не приняв подтверждение, ждёт другое сообщение TCP для того, чтобы послать извещение о размере окна. И оба TCP могут продолжать ждать друг друга бесконечно.

Чтобы скорректировать этот тупик, TCP использует время настойчивости для каждого соединения. Когда TCP передатчика получает подтверждение с размером окна нуль, он запускает таймер настойчивости. Когда время таймера настойчивости истечёт, TCP передатчика посылает специальный сегмент, называемый проба. Этот сегмент содержит только 1 байт данных.

Значение таймера настойчивости устанавливается по времени повторения передачи. Однако если отклик не получен от приёмника, посылается другой пробный сегмент и значение таймера настойчивости дублируется и переустанавливается. Передатчик продолжает передавать пробные сегменты и переустанавливать значение таймера настойчивости, пока не достигнет порогового значения (обычно 60 с.). После этого передатчик посылает один пробный сегмент каждые 60 с., пока окно не откроется снова.

**3. Дежурный таймер**

Дежурный таймер используется в некоторых реализациях для предотвращения длительных не работающих соединений двух TCP. Предположим, клиент открыл соединение TCP к серверу, потом передал некоторые данные и затих. Может быть, клиент "упал", в этом случае соединение останется открытым навсегда.

Чтобы избавиться от этой ситуации, в большинстве реализаций оборудуется сервер с дежурным таймером. Каждый раз, когда сервер "слышит" клиента, он переустанавливает таймер. Отсчёт времени обычно — два часа. Если нет никакого отклика после 10 проб, каждая по 75 раз, он предполагает, что клиент "не в порядке", и заканчивает соединение.

**4. Таймер времени ожидания**

Таймер времени ожидания используется после окончания соединения. Когда TCP закрывается, он реально не разъединяется. Соединение удерживается в неопределённости на период времени ожидания. Если что-либо прибудет к пункту назначения, то оно будет удалено. Значение для этого таймера равно двум временам ожидаемого времени существования сегмента.

**Алгоритм четверного рукопожатия**

Протокол SCRIPT ориентирован на установление соединения, подобно TCP, поэтому он также имеет собственные процедуры рукопожатия и завершения. Минимальное количество пакетов для установления ассоциации SCRIPT равно четырём, поэтому описанная процедура называется четырёхэтажным рукопожатием SCRIPT. При установлении ассоциации SCRIPT выполняется приведённая далее последовательность действий, подобная трёхэтапному рукопожатию TCP.

1. Сервер должен быть готов к приему входящего соединения. Подготовка обычно осуществляется последовательным вызовом функций socket, bind и listen и называется пассивным открытием.

2. Клиент начинает активное открытие, вызывая функцию connect или сразу отправляя сообщение. При этом клиент SCRIPT передаёт сообщение INIT, в котором серверу отправляется список IP-адресов клиента, начальный порядковый номер, идентификационная метка, позволяющая отличать пакеты данной ассоциации от всех прочих, количество исходящих потоков, запрашиваемых клиентом, и количество входящих потоков, поддерживаемых клиентом.

3. Сервер подтверждает получение сообщения INIT от клиента сообщением INIT-ACK, которое содержит список IP-адресов сервера, начальный порядковый номер, идентификационную метку, количество исходящих потоков, запрашиваемых сервером, количество входящих потоков, поддерживаемых сервером, и cookie с данными о состоянии. Cookie содержит все сведения о состоянии, которые нужны серверу для того, чтобы гарантировать действительность ассоциации. В cookie включается цифровая подпись, подтверждающая аутентичность.

4. Клиент отсылает cookie обратно серверу сообщением COOKIE-ECHO. Это сообщение уже может содержать пользовательские данные.

5. Сервер подтверждает правильность приема cookie и установление ассоциации сообщением COOKIE-ACK. Это сообщение также может включать полезные данные.

Во многих отношениях четырехэтапное рукопожатие SCRIPT подобно трёхэтапному рукопожатию TCP, за исключением всего, что связано с cookie. Сообщение INIT включает контрольную метку Та и начальный порядковый номер J. Метка Та должна присутствовать во всех пакетах, отправляемых собеседнику по данной ассоциации. Начальный порядковый номер используется для нумерации сообщений DATA. Собеседник тоже выбирает собственную метку Tz, которая должна присутствовать во всех его пакетах. Помимо контрольной метки и начального порядкового номера K получатель сообщения INIT отправляет cookie С. Пакет cookie содержит все сведения о состоянии, необходимые для установления ассоциации SCRIPT, так что стеку SCRIPT сервера не приходится хранить сведения о клиенте, с которым устанавливается ассоциация.

В заключение рукопожатия каждая сторона выбирает основной адрес назначения. На этот адрес передаются все данные в отсутствие неполадок в сети.

Четырехэтапное рукопожатие SCRIPT с использованием cookie формализует метод защиты от атак типа «отказ в обслуживании». Многие реализации TCP используют аналогичный метод. Отличие в том, что при работе с TCP данные cookie приходится кодировать в начальный порядковый номер, длина которого составляет всего 32 разряда. В SCRIPT используется поле произвольной длины и криптографическая защита.

**Многопоточность (SCTP)**

Позволяет передавать несколько потоков в рамках одной ассоциации (ассоциацией в SCTP называется соединение между двумя хостами). Например, в TCP данные и служебная информация передаются по одному соединению. Поэтому задержка в получении служебной информации может быть вызвана текущей передачей данных (например, ACK может не прийти вовремя, поэтому мы пошлём дубликат). Многопоточность позволяет передавать эти данные независимо, что приведёт к лучшему использованию доступных ресурсов.

**Мультихоуминг (SCTP)**

SCTP позволяет устанавливать соединение к одному серверу по разным линиям связи (например, по Wi-Fi и по Ethernet). Таким образом, если одна линия связи отвалится (скорей всего Wi-Fi), то соединение не разорвётся. Это так же позволяет передавать данные сразу по нескольким линиям, что увеличивает скорость передачи. Если в TCP одна линия связи оборвётся, то соединение будет потеряно, придётся устанавливать новое.

**TCP SYN Flood атака**

Злоумышленник шлёт много пакетов в короткое время для запроса TCP соединения (запрос на соединение помечен битом SYN в заголовке, поэтому и SYN-flood). Но при этом не подтверждает установление соединения. Таким образом на сервере образуются полуоткрытые соединения. Они закрываются сами по истечению таймаута. Но цель злоумышленника состоит в том, чтобы создать как можно больше таких соединений, не давая тем самым создавать новые валидные соединения из-за ограничения в их количестве на стороне сервера (сервер не может иметь бесконечное число соединений, а если сделать таймаут на обрыв слишком маленьким, то валидные соединения могут отваливаться раньше времени, что тоже нехорошо, и это всё делает syn-атаки возможными). В SCTP используется не тройное рукопожатие, а четверное (из разряда: "я хочу установить соединение - ты точно хочешь установить соединение? - да, я точно хочу установить соединение - ну тогда ладно"). Таким образом за короткий промежуток времени нельзя создать много новых соединений. В SCTP не бывает полузакрытых соединений, как в TCP. Если мы закрываем соединение, то сразу в обе стороны.

Противодействие

Предложенным решением было использование SYN cookie, либо ограничение запросов на новые подключения от конкретного источника за определённый промежуток времени.

SYN cookie — техника противодействия SYN-флуд-атаке. Изобретатель техники определил SYN cookie как «особый выбор начальной TCP-последовательности со стороны сервера». Использование SYN cookie позволяет серверу избегать сброса новых соединений, когда очередь TCP-соединений переполнена. Сервер отправляет обратно клиенту правильную последовательность SYN+ACK, но не сохраняет новое соединение в очереди. Если сервер затем получит ACK ответ от клиента, то он сможет восстановить своё значение SYN последовательности по принятому от клиента значению.