TCP报文结构



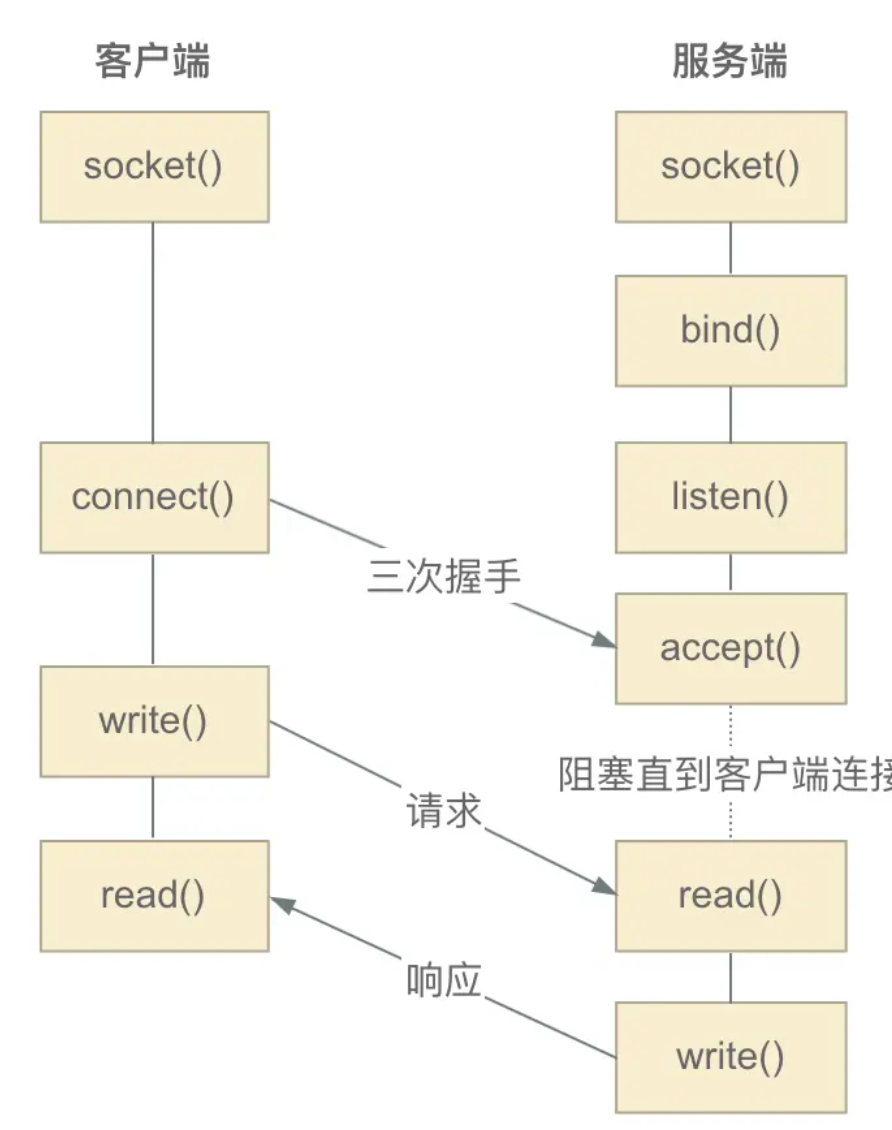
上图是TCP报文结构，以下是首部各个字段的描述：

* 占用16位的**源端口**号和**目的端口**号，标识了发送方和接收方的进程标识；
* 32位的**序号**字段表示数据报文的编号，应用层要发送一段应用数据，当传送到传输层后，如果应用数据大小超过MSS（最大报文段长度）时，就会对应用层数据字节流从头到尾按顺序进行编号，然后再切分为一个个MSS大小的TCP报文段，每个报文段的序号字段就是该分段的第一个字节的序号；
* 32位的**确认序号**字段表示希望对端发送的下一个报文段的序号。由于TCP是一种全双工的通讯协议，即发送方既发送数据也可能接收对端发送的数据，如源主机发送0-1000编号的字节段给目的主机，这时目的主机就会发送一个响应报文且确认序号为1001，表示希望收到下一个报文段编号为1001的数据，它和序号字段共同保证了TCP报文的可靠性；
* 4位的**首部长度**表示TCP报文首部的长度，该字段以4字节为单位。TCP首部的长度是可变的，在特殊情况下选项字段会有值，也计入首部长度，但一般情况下选项都会为空，这样首部长度就是20个字节，因此该字段的数值就为5，用4位二进制表示就是0101；
* 6位**保留位**未用到，可能以后TCP版本升级会被征用；
* 之后的6位是**标志位字段**，其中，SYN用于TCP连接的建立，FIN用于连接的关闭，RST是重新连接，ACK表示对发送方数据的确认，和确认序号结合使用；PSH标识⽴刻将数据交给上层处理；URG标识数据中需要被上层处理的紧急数据，而紧急数据部分是通过**紧急指针**字段来标识哪一段属于紧急数据；
* 16位**窗口大小**用于标识自己能接受的最大报文数，用于控制TCP连接的数据流量；
* 16位**校验和字段**用于接收端验证数据包的准确性，防止中途被篡改，这也是保证了TCP的可靠性。

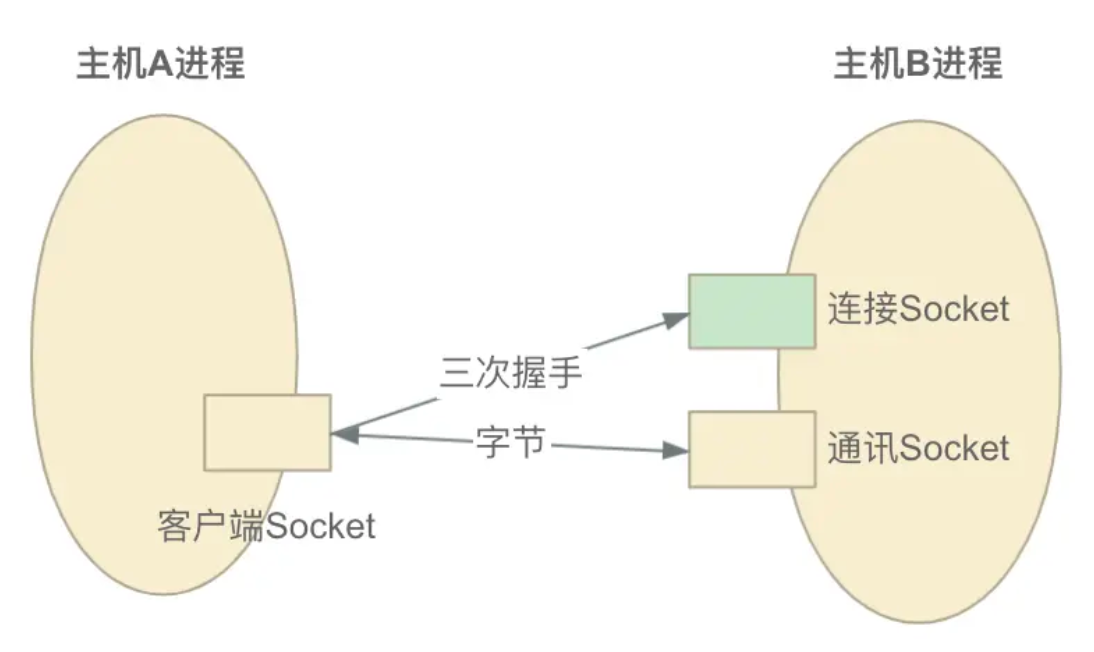
TCP连接

Socket连接的建立

服务端要先调用socket()函数创建一个Socket，然后通过bind()绑定端口号和IP地址，端口号是为了客户端发送数据过来时能找到对应要处理的进程，而绑定IP是由于一台主机可能有多个网卡，需要选择指定的网口去接收。当服务端有了 IP 和端口号，就可以调用 listen 函数进行监听，这个时候客户端就可以发起连接了。在服务端等待的时候，客户端可以通过 connect 函数发起连接。先在参数中指明要连接的 IP 地址和端口号，然后开始发起三次握手。内核会给客户端分配一个临时的端口。一旦握手成功，服务端的 accept 就会返回另一个 Socket。



服务端的连接Socket收到连接请求并确认连接可用后会为该主机创建一个独有的通讯Socket用于收发数据报（这个过程就是上图 accept 函数的处理过程）



三次握手

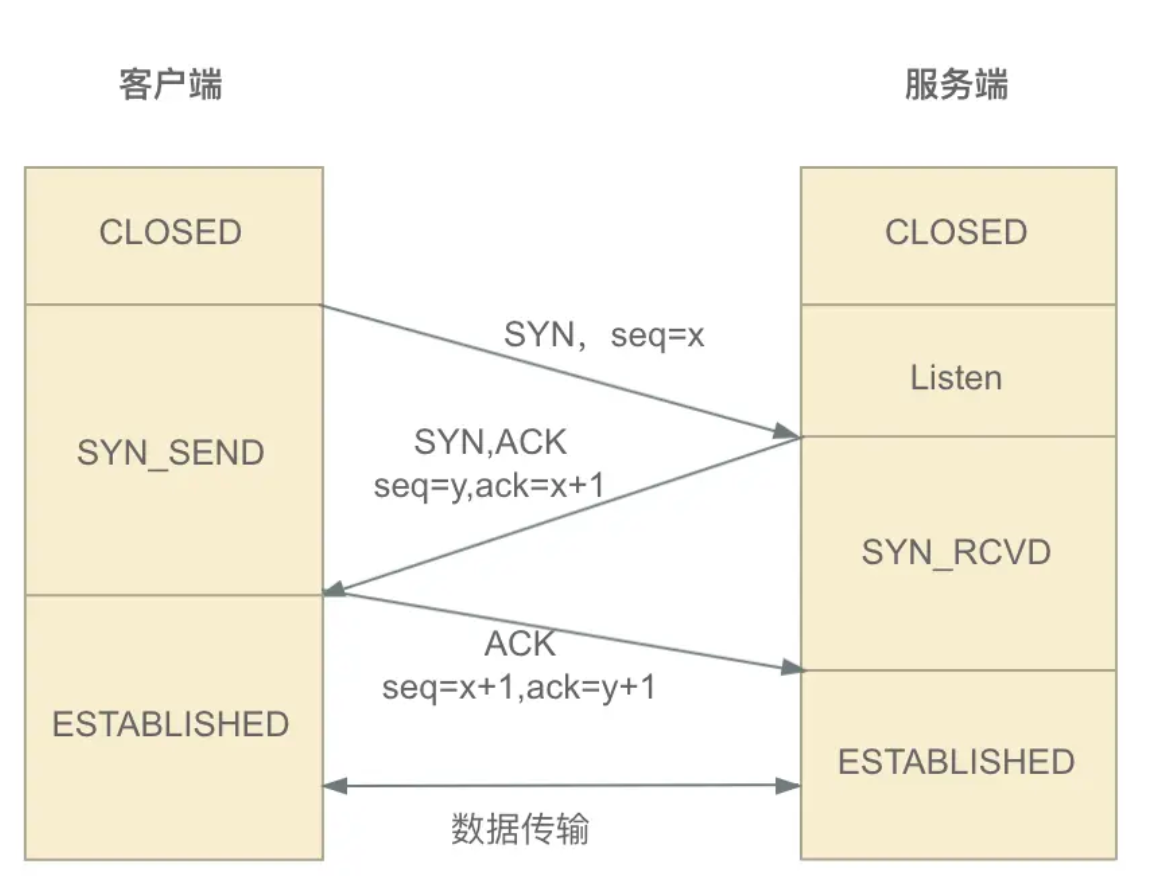
客户端调用connect()函数后会经过TCP三次握手的连接过程。这三次握手过程就是收发双方来回发送了三个TCP数据报后就确定了连接关系，可以开始发送数据了。那三次握手的规则是怎么样的呢？接下来详细分解：

首先，客户端和服务端原本互不相关，都处于连接关闭状态。现在客户端主动发起连接，就发送一个TCP报文段（称为SYN报文），SYN标志位置1，序号字段设为一个原始编号x，这个原始编号可以认为是一个32位计时器，每4微秒加1，这样是为了防止序号重复，该原始编号同时也是应用层数据包的起始编号；

服务端收到SYN报文后，接着会给客户端发送一个ACK置1、确认序号为x+1响应报文，客户端发送的SYN报文段会消耗一个序号，虽然SYN报文没有数据；与此同时，TCP是全双工通讯，服务端也会发送数据给客户端，因此响应报文也会将SYN置为1，序号设置为服务端的原始编号y，该编号同样是通过服务端的系统计时器得出的一个随机数；

客户端收到服务端的响应后，再对服务端的SYN发送一个ACK置1、确认序号为y+1的响应报文，而序号同样为x+1，这说明确认报文是没有消耗序号的。

自此，TCP连接就建立了。



四次挥手

· 客户端主动发送一个FIN标志位置1，序号为u的TCP报文段到服务端，这时客户端就进入了FIN-WAIT-1状态；

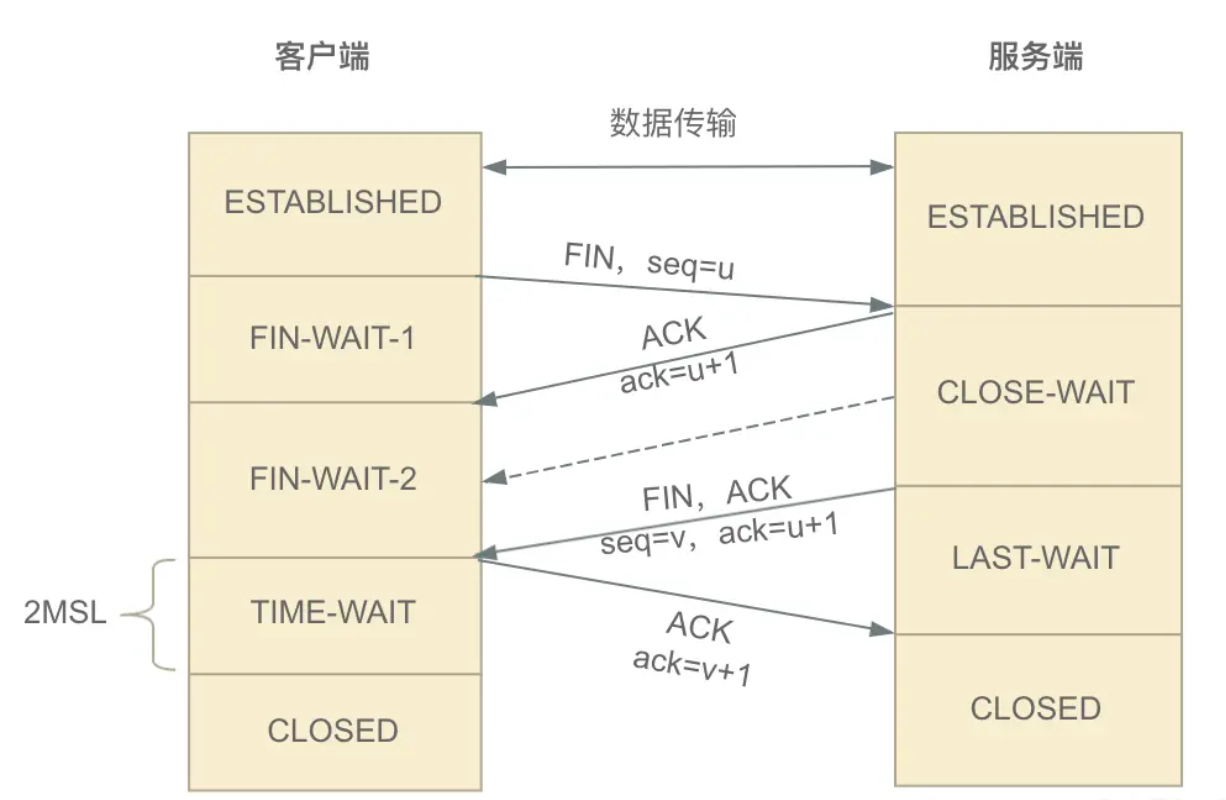
· 服务端收到客户端的FIN报文段后，会发送FIN报文的响应报文，响应报文将ACK置1，响应序号为u+1，说明FIN报文也和SYN报文一样会占用一个序号。服务端发送完响应报文后会进入CLOSE-WAIT状态；

· 客户端收到服务端的响应报文后进入FIN-WAIT-2状态，这时这条TCP连接已处于半关闭状态，从客户端到服务端已经不能再发送数据报了，但服务端还能继续向客户端发送数据报；

· 当服务端也没有数据要发送给客户端时，也主动发起了FIN报文，发送完后服务端自动进入LAST-WAIT状态，就等着收到该FIN的确认报文了；

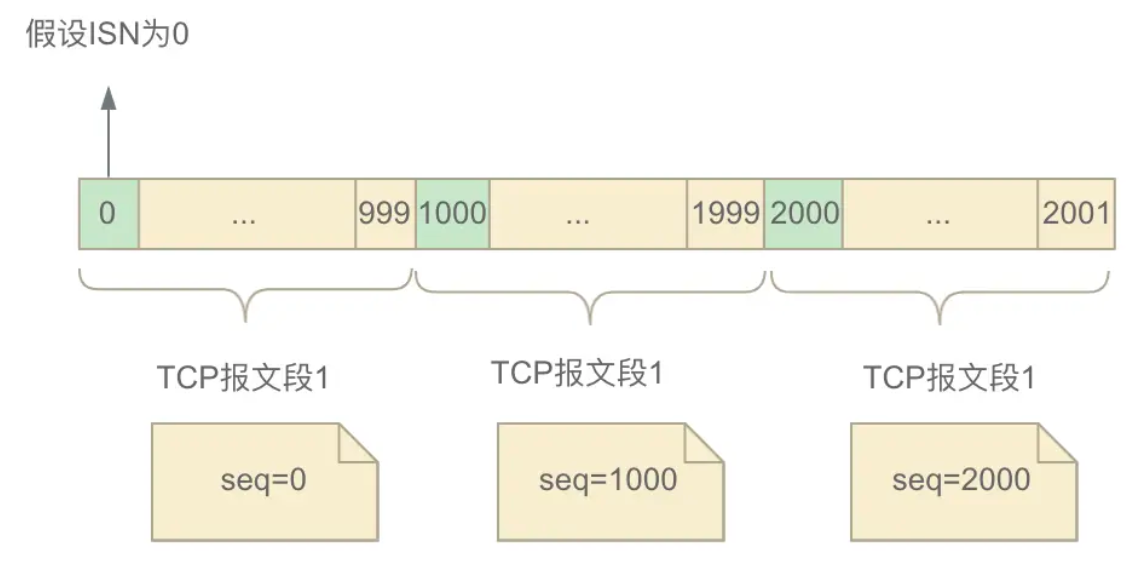
· 客户端收到服务端的FIN报文后，就发送了FIN的响应报文给服务端。照理说这个时候客户端已经断开操作了，可以释放IP和端口了。但这样还是会有问题，假如客户端发送响应报文后丢失了，服务端又重发了FIN报文，但这时如果客户端已经释放端口了，服务端就永远收不到ACK了，也就无法关闭。

· 因此客户端在发送完FIN的ACK后，还要WAIT-TIME等待2MSL时间才能真正关闭。MSL为最长报文存活时间，即任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。那为什么要等待2倍的MSL才能真正关闭呢？这样可让TCP再次发送最后的ACK以防这个ACK丢失（另一端超时并重发最后的FIN）。



TCP的可靠性

TCP是一个可靠的通讯协议，它能保证数据的有序性、不丢失和无差错。那这些特性是如何保证的呢？前面我们已经有了解过序号和确认序号的使用了，就是这两个字段保证TCP的这些特性。应用层将要发送的数据不断地发送到Socket中，行程的数据流就不断地交给TCP处理，由于字节流本身是有顺序的，TCP会先将这些字节流编号，而起始编号是以ISN作为初始序号（ISN是根据系统计时器生成，防止和其他TCP连接重复）。编号后再将字节流切割成一个个MSS大小的TCP报文段，每个报文段的首字节编号作为此TCP报文首部的序号字段。



累积确认

对于客户端发送的每一个TCP报文段，服务端都要对应发送一个确认报文段，如果在一定时间内没有收到服务端的确认报文，客户端就会认为这个报文可能丢失了，就会重发，这样就保证了不丢失的特性。假设客户端发送0-999字节的报文段到服务端，服务端就会发送一个确认序号为1000的确认报文段，代表服务端接下来希望接收1000编号开始的报文段。但这样发送一个报文响应一个报文，一来一回的传送方式效率低下，真实的场景可能是客户端连续发送了多个TCP报文段，服务端只发送一个确认报文，就能确认所有客户端发来的报文，这叫做累积确认。如，客户端发送了0-999、1000-1999和2000-2999三个报文段，服务端连续收到了三个TCP报文段后，只发送一个确认序号为3000的报文段就能确认以上的三个报文。

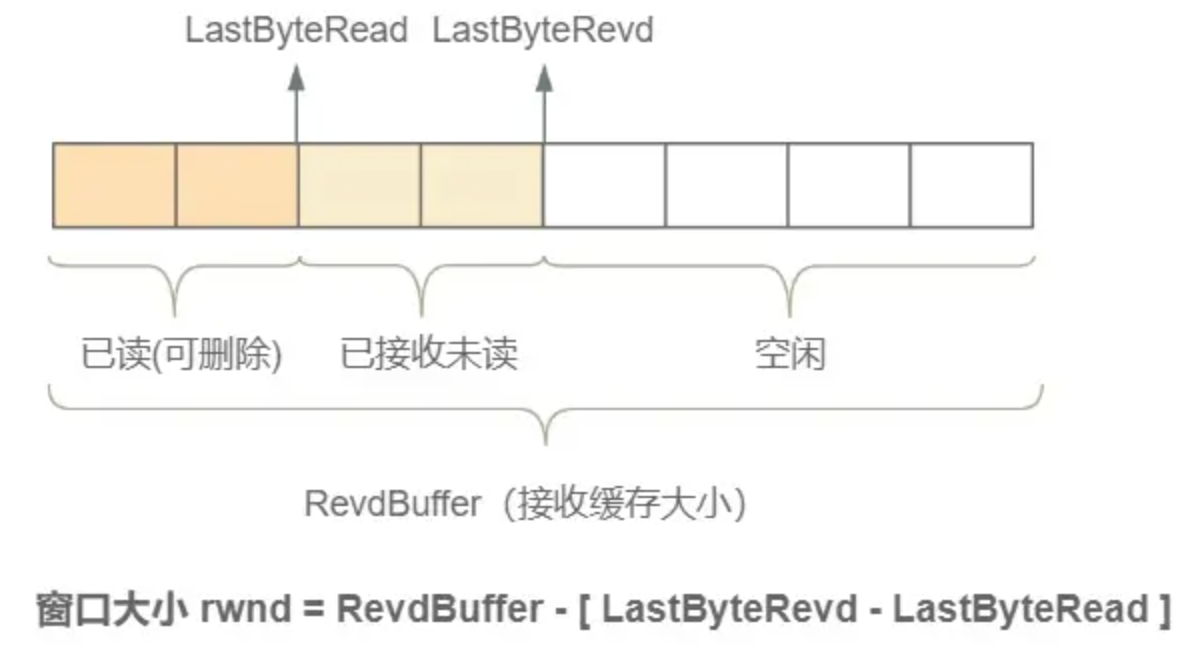
保证数据报有序性

客户端发送了多个报文后，每个报文都是独立地往底层传输，可能每个报文经过的网络路由都不一样，因此三个按照顺序发送的报文，可能会不按顺序到达服务端。那服务端是怎么解决的呢？它是通过只确认最后一个按序到达的TCP报文来解决的，比如，服务端刚发送完确认序号为3000的ACK报文段后，接着收到了一个4000-4999的报文段，这时由于还有一个3000-3999的报文段未收到，因此服务端不会发送确认序号为5000的ACK报文，而是先将4000-4999的报文段存放到接收缓存中，然后还是继续发送确认序号为3000的ACK报文段。等到失序的报文段收到后再发送一个确认序号5000的ACK来累积确认所有已收到的报文。

TCP的流量控制

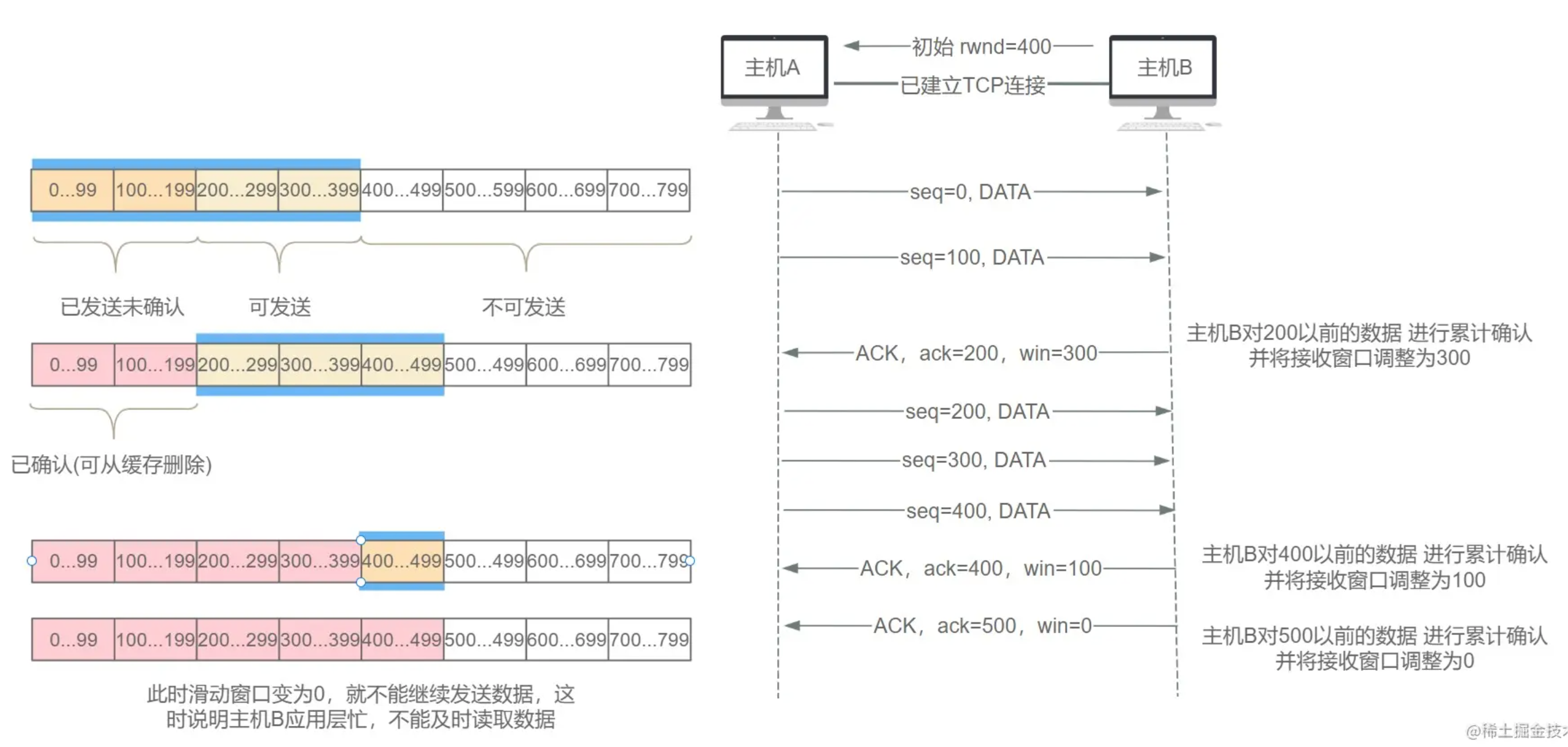
TCP连接建立后，每一侧的主机都为该连接建立了一个接收缓存。当连接的一端收到有序、正确的报文段后就将数据存放到接收缓存中，然后应用程序再从缓存中读取数据，被读取过数据就会从缓存中删除。但有时候应用程序处理速度较慢时，就会导致接收缓存占满空间，这时发送方继续发送报文时就会因为缓存溢出而被丢弃，因此TCP需要提供一个流量控制服务来防止缓存溢出的情况。那TCP是如何实现流量控制呢？

TCP通过在每次双方收发数据时，都给对方通告一个窗口大小，该值是接收方可用缓存的大小，用来告诉发送方我最多只能接收这么多数据。该窗口大小是通过TCP报文段结构中的16位窗口大小字段来传给对方的。由于TCP是全双工协议，两端主机都会各自维护一个接收窗口大小的变量rwnd，这个变量是发给对端的，同时自己也会维护一个发送窗口大小，这个窗口大小是控制自己发送数据的快慢。下图是关于接收窗口的计算：



滑动窗口

不管是流量控制还是等下要分析的拥塞控制，其最终实施方案都是通过控制发送窗口的大小来实现的。TCP协议规定发送窗口的大小是取拥塞窗口和接收窗口的较小者。但现在我们先忽略拥塞情况，假设发送窗口大小只受接收端接收窗口的影响，且发TCP连接数据传输是单向的，发送端只发送数据，接收端只接收数据，如图所示是主机A的发送窗口随着主机B响应报文通报的窗口大小而动态滑动的示意图：



主机A和主机B建立TCP连接后，主机B通报其接收窗口大小为400字节，因此主机A将初始发送窗口设置为400，通过图左边的蓝色背景代表发送主机A的发送窗口大小。这时主机A将seq为0和100的两个报文段发送出去，随即在左边的缓存中用深黄色标注出已发送待确认的报文。之后从主机B收到响应报文，ack为200代表主机B对序号为200以前的两个报文段进行累积确认，同时接收窗口减小为300，这时有可能是主机B虽然已收到两个确认报文，都将两个报文存入接收缓存，但应用层只读取了一个，导致接收窗口减小了100；主机A收到响应报文后，发现0-99和100-199的报文段已经确认，于是将蓝色窗口向右移动到200-299报文段的左边缘，由于此时接收窗口减小为300了，因此右边缘向左移动一个报文段的位置。这时包含在蓝色窗口内的是200-299、300-399和400-499三个可发送未发的报文，因此主机A就可以同时将三个报文段都发送出去并将其标注为已发送未确认的深黄色...

零窗口和持续计时器

在主机B发送最后一个响应报文后，通报接收窗口为0，这时主机A的发送窗口也随着变为0，这时就是零窗口状态，这时主机A就不能继续发送数据了，只能等主机B将接收窗口更新为大于0后才能继续发送，但如果接收方发送更新窗口的确认报文在中途丢失了，这时双方就处于假死状态。因此当窗口为0时，客户端会启动持续计时器，在计时器触发时主动发送零窗口探测报文段，服务端收到探测报文后就会同步发送一个响应报文并更新窗口大小，但也有可能探测报文也会丢失，还是回到原来的问题了。因此零窗口探测报文也有一个重传计时器，为了避免该探测报文段也丢失的情况。

TCP的拥塞控制

对网络中某一资源的需求超过了该资源所能提供的可用部分，网络性能就要变坏，这种情况称为拥塞，若出现拥塞而不进行控制，整个网络的吞吐量将随输入负载的增大而下降。因此发送端引入了一个拥塞窗口变量cwnd来动态了解网络的拥塞情况，前面滑动窗口大小rwnd是怕数据把接收窗口塞满，而拥塞窗口cwnd是怕数据把网络塞满，因此需要根据两个值来决定发送窗口的大小。

那怎么判断发送多少数据能把整个网络塞满呢？其实，相对于网络层来讲，TCP是上层协议压根就不知道底下的网络状态，而发送端在发送数据，就像往一个瓶子里倒水，不能一下子往里面灌进太多水，这样会溅出来，同样拥塞控制也是使发送的数据不出现丢包、超时重发前提下，尽量发挥带宽作用、尽快地发送数据。

因此，TCP设计了几个四个算法用来控制拥塞窗口慢开始、拥塞避免、快重传和快恢复，我们通过一个例子来说明四个算法。首先，假设接收方有足够大的接收缓存，因而发送窗口仅受拥塞窗口影响。

· 发送方维护一个拥塞窗口cwnd的变量，其值取决于网络的拥塞情况，且动态变化；

* · 拥塞窗口cwnd的维护原则：只要网络没有出现拥塞，cwnd就再增大一些，如果出现拥塞时，cwnd就再减小一些；
* 出现拥塞的判断依据：没有按时收到应当到达的确认报文（即出现超时重传的情况）；

· 发送方将拥塞窗口作为发送窗口swnd，即swnd = cwnd；

· · 发送方维护一个慢开始门限的变量ssthresh的状态变量：

* · 当swnd < ssthresh时，使用慢开始算法；
* 当swnd > ssthresh时，停止使用慢开始算法而改用拥塞避免算法；
* 当swnd = ssthresh时，既可以使用慢开始算法也可以使用拥塞避免算法；

慢开始

慢开始的算法是先设置发送窗口大小cwnd为1（1代表一个MSS大小报文段），发送完报文后，在不发生超时重发情况下每收到一个TCP报文段的确认后，cwnd就加1，即确认了几个报文就加几。刚开始发送1个报文段，当收到这个报文段的确认后，cwnd就加1变为2，这时就可以同时发送两个报文段，当收到这两个报文段的累计确认后，cwnd就加2变为4，继续同时发送四个报文出去，当收到4个报文段的累计确认后，swnd就加4变为8了。由此可知，慢开始算法是一个指数增长的关系，cwnd增长得很快，之所以叫慢开始其实是指一开始的发送窗口比较小。由于发送窗口是指数方式增长，迟早会把整个网络带宽塞满，那什么时候会结束这种指数增长呢？一般会有两种情况：

当有一个发送的报文出现超时未收到响应时，就代表出现了拥塞了。这时就将swnd重新设置为1，同时将慢开始门限变量设为出现拥塞时cwnd的一半，即ssthresh=cwnd/2，随后继续进行慢开始算法。结束慢启动算法的第二种情况是与慢开始门限ssthresh有关了，当cwnd增大到超过或等于ssthresh时，就要停止慢开始算法而改用拥塞避免算法。

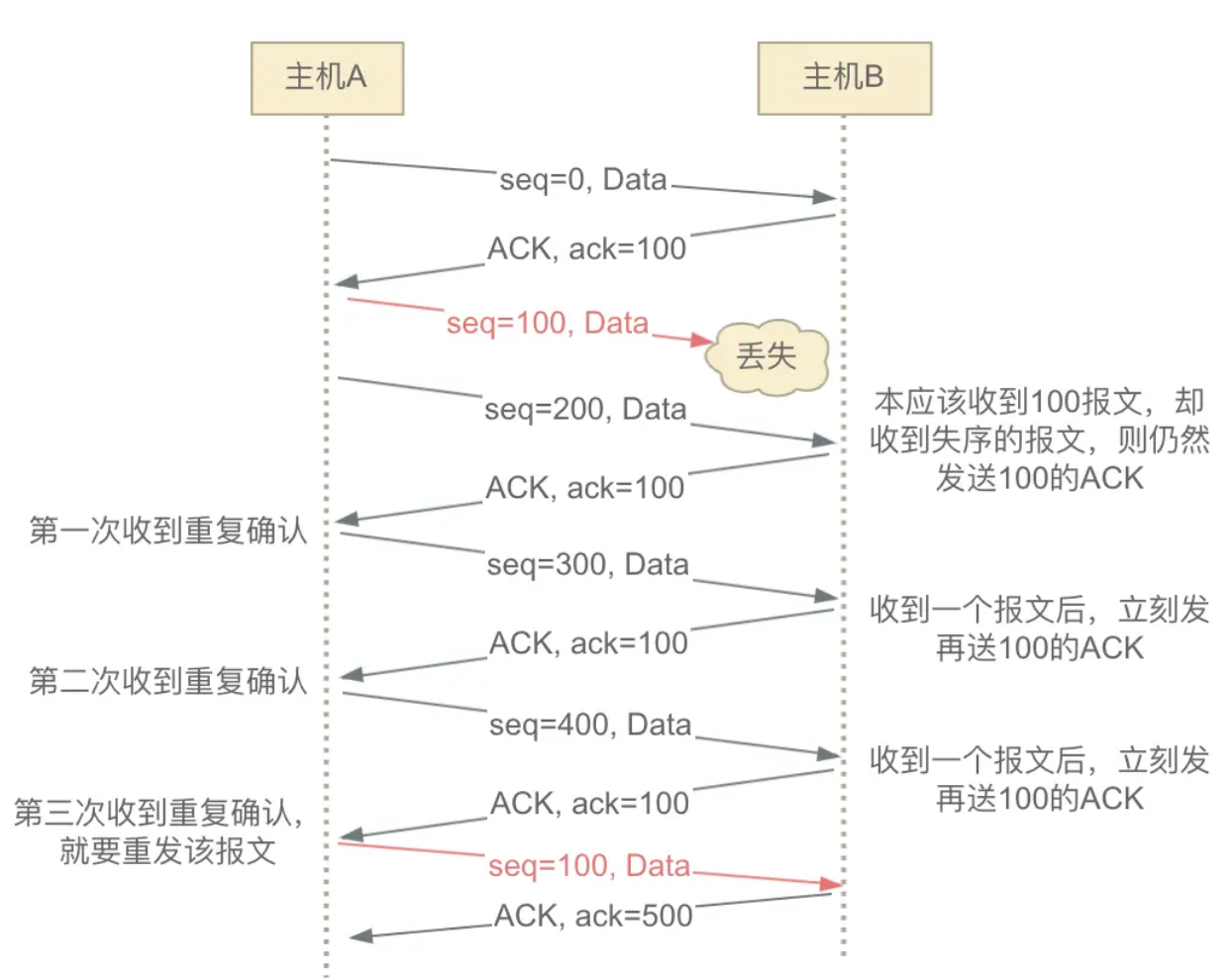
拥塞避免

当使用拥塞避免算法时，此时的拥塞窗口刚好是使用慢开始出现拥塞时的一半，这个时候可能举例出现拥塞也不远了。因此，就不能像慢开始一样快速地增加拥塞窗口大小了，而是每次收到一个数据报文的确认时，就将cwnd增加1/cwnd。假如现在的cwnd为8，于是就同时将8个TCP报文段发送出去，当收到所有8个报文段的累计确认时，就将cwnd增加8 \* (1/8)，即增加1个报文段为9，然后继续发送9个报文段出去，又收到9个报文段的累计确认时，又将cwnd增加1变为10，由此可知，拥塞避免算法是以线性方式增长的。那什么时候结束这种线性方式增长呢？和慢开始算法碰到拥塞情况时一样，当出现拥塞时，就将即ssthresh设为原来的一半，cwnd重新设置为1，又重新开始执行慢开始算法，然后算法流程还是和原来的一样重复执行。

快重传

在使用慢开始和拥塞避免算法时，当出现数据报超时时，就认为是出现了网络拥塞。但这并不是绝对的，当网络还是很流畅时，传输过程中有个别数据报由于各种原因导致丢失了，这时发送方就发现此数据报超时未确认，如果因此就认为是网络拥塞导致的，就会误用拥塞算法而将cwnd减小为1，这无疑就降低了传输效率了。因此，就出现了快重传算法来解决这种个别数据报丢失的情况。

快重传的原则是发送端在发现个别报文丢失时，就应该尽快重传，而不是等到超时了再重传。那发送端怎么知道是个别报文丢失呢？如下图所示，A主机和B主机已建立连接情况下，A主机先发送一个数据报文0，正常收到报文，接着发送第二个报文100，该报文在传输工程中丢失了，主机B就收不到报文100的数据，接着主机A继续发送其他报文200，该报文在主机B正常收到，但主机B发现报文100还没收到，先收到报文200，于是将报文200存放到接收缓存，继续发送确认序号为100的ACK报文，代表希望从主机A收到报文100。接着主机A继续发送报文300和400，两个报文都正常到达主机B，主机B由于设置不使用经受时延的确认，对每个收到的报文都要及时发送ACK报文，也就连续一共发送了三次确认序号为100的响应报文。当客户端连续收到三个连续的相同确认序号的报文端时，就知道这个报文可能中间丢失了，于是就要重发报文100，当服务端收到这个空缺的报文段后，就已收到的失序报文一起累计确认。



快恢复

当发送方连续收到三个重复的ACK报文时，就知道现在只是个别报文端丢失了，于是不启用慢开始算法，而改用快恢复算法。快恢复算法一般有两种实现方案：

发送方将慢开始门限ssthresh值和拥塞窗口cwnd都调整为当前窗口的一半，然后开始执行拥塞避免算法；

将快恢复开始时的拥塞窗口值再增大一些，即等于新的ssthresh+3，这样做有几个原因：

既然发送方收到三个重复的ACK报文，就表明有三个数据报文段已经离开了网络；

这三个报文段不再消耗网络资源而是停留在接收方的接收缓存中；

可见现在网络中不是堆积了报文而是减少了三个报文段，因此拥塞窗口可以适当再扩大些。

经受时延的确认

TCP为了减少TCP数据单元的交互，增强效率。在收到TCP连接远程一端的数据报时，不会马上发送确认报文，而是等待系统的定时器，在定时器一个200ms周期内，如果本地端有要发送的数据，则连同确认报文一起发送出去，如果没有的话在200ms的触发时机才会发送单独的确认报文。这种现象也被称为数据捎带确认，它相当于TCP的一个功能，可开启也可被关闭，如前文介绍的快重传算法就没有启动该功能。

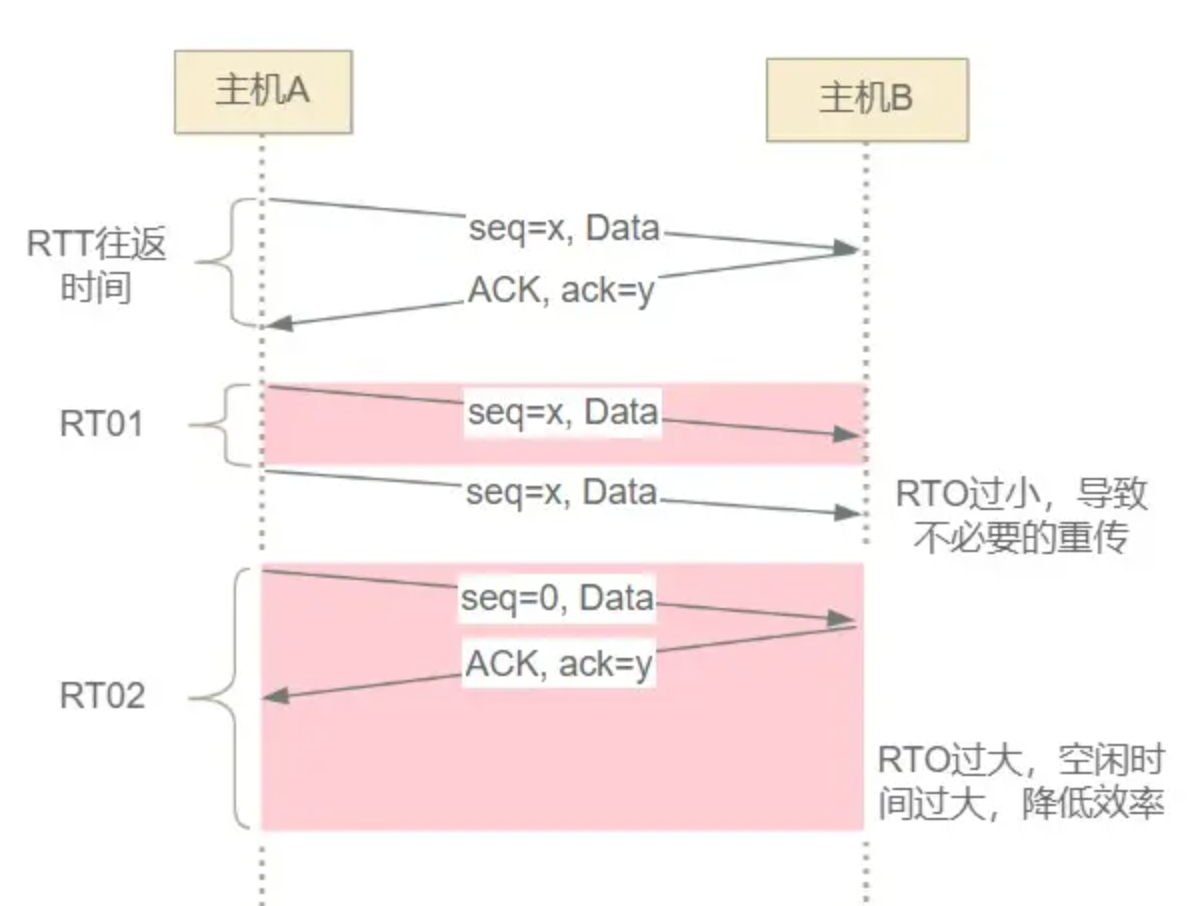
Nagle算法

Nagle算法为了避免发送小的数据包，规定同一个时间段内只能有一个未被确认的数据分组发送出去，在一个数据包发送出去还没收到确认报文之前，所有待发送的数据报都要在缓存区等待，等上一个数据报收到确认后，再将缓冲区内积累的所有数据当做一个数据分组发送出去（前提是不能超过MSS）。这样能提高端对端的数据传输效率，假设发送端每次发送的数据只有一个字节，如果没有启用Nagle算法的话，每次发送的一个字节就作为一个数据包发送出去，每个数据包就会有41字节（20字节的IP首部和20字节的TCP首部，加上一个字节的业务数据），这样负荷数据就增加了40倍。

Nagle算法虽然能提高发送效率，因为其他数据包都要等到确认到达时才能发送出去，这对于经常发送细小数据包的应用层业务很有好处的，比如Rlogin这种命令式交互的应用，而对于数据实时性要求比较高的业务，如射击类游戏这显示是不合适的，因为Nagle算法也增加了延迟，这时就需要禁用Nagle算法来提供传输速率。Nagle算法在大部分系统是默认开启的，但也是可以关闭的，如Linux提供了TCP\_NODELAY的选项来禁用Nagle算法。

TCP超时重传

在拥塞控制的介绍中，已经了解到一个TCP报文发送出去后，如果在一定时间内没有收到确认报文段，就会导致发送端超时重传。那这个超时时间是怎么定的呢？其实计算TCP超时重传时间是个很复杂的过程，我们先通过一个图了解一下RTT往返时间和超时时间的关系：



这里，RTT是指TCP数据报文的从发送数据报到接收响应报文的往返时间，RTO是指TCP报文超时重传时间。如上图所示，当网络不出现拥塞时，TCP报文的往返时间为RTT，假如超时重传时间RTO定得远小于RTT，就会出现不必要的重传，而如果将RTO定得远大于RTT，当出现拥塞时就会等待过长时间才重发报文，降低效率。因此，RTO就应该设置得比RTT大一点才更合理。但是，网络环境很复杂，在一个TCP连接中，可能上一个报文段的RTT小一些，下一个报文段就会因为网络问题而变得很大，那怎么通过RTT来计算RTO的值？

通过上面的分析，显然不能直接使用某一次的RTT来计算超时重传时间RTO。因此，TCP利用每次测量得到的RTT样本，来计算加权平均往返时间RTTs（又称平滑往返时间），再RTTs来计算RTO，它们计算公式如下：

RTTs1 = RTT1 (第一个报文段的RTT作为RTTs)

RTTs = (1 - α) × 旧的RTTs + α × 新的RTT样本 (RFC6298标准推荐α值为0.125)

RTO = RTTs + 4 × RTTd

这里我们看到了一个新的变量RTTd，这个叫做RTT偏差的加权平均，计算公式如下；

RTTd1 = RTT1 ÷ 2 (第一个报文段的RTTd值)

新的RTTd = (1 - β) × 旧的RTTd + β × (RTTs - 新的RTT样本) (RFC6298标准推荐β值为0.25)

Karn算法

由于网络环境的复杂性，当主机A在时间T1将TCP报文段发送出去后由于网络拥塞导致超时，因此在时间T2重发，之后在时间T3收到了响应报文，那这时RTT的值T3-T2还是T3-T1呢？这时会有两种情况，如果第一次报文段在发送过程中丢失了，而第二次重发后正常到达主机B并收到响应报文，因此RTT=T3-T2，第二种情况是第一次报文段正常到达主机B了，并发回了响应报文，但响应报文发回主机A时发生延迟了，等第二次重发报文后发送端才收到此确认报文，这时RTT就应该为T3-T1。但客户端主机A这种情况是无法得知的。

因此，针对出现超时重传时无法推测往返时间RTT的问题，Karn提出了一个算法：在计算加权平均往返时间RTTs时，只要报文段重传了，就不采用其往返时间RTT样本。也就是出现重传时，不重新计算RTTs，进行RTO也不会重新计算。

但这又引发了另一个问题，假设网络突然拥塞，一个报文段的往返时间RTT突然增大很多，这就导致了该报文段超时重传，之后的很多报文段都是保持这样大的RTT，于是都超时重传了。但根据Karn算法，超时重传的报文不计入更新RTO，就会导致RTO比实际情况偏小，大量的报文段都反复被重发。因此，Karn算法有了更新，方法是：报文段每重传一次，就把超时重传时间RTO增大一些，普遍的做法就将RTO取值为旧RTO的2倍。