改进设定超时时间的算法：不仅需要根据链路时延，还需要根据时延抖动来确定超时时间，避免无谓重传。

RTT(Round Trip Time)：一个连接的往返时间，即数据发送时刻到接收到确认的时刻的差值。

RTO(Retransmission Time Out)：重传超时时间，即从数据发送时刻算起，超过这个时间便执行重传。

设发送端发出数据包时刻为T1，那么当T = T1+ RTO时还未收到这个数据包的 ACK 消息，那么发送端就认为数据包丢失，重传该数据包。这种方式依赖于接收端的ACK应答和RTO的准确测量，容易产生误判，主要有两种情况：

1. 对方收到了数据包，但是 ACK 在发送途中丢失。
2. ACK在途中正常传输，但是由于时延抖动，距离发送的时刻T1已经超过了一个 RTO。



图1 RTT与RTO

所以超时重传的重点和难点主要集中在 RTO 的计算上，计算RTO前，需要得到往返时延和时延抖动，具体计算方式如下：

**RTT估计：**

当数据包超时后就会自动重传数据。那么超时时间RTO怎么选取？若设置过短，数据包本来没丢却认为丢了，则会造成重传频繁，加剧网络拥塞程度；若设置过长，则会导致传输性能下降（失序的报文段等待时间过长，造成时延增大，吞吐率下降）。所以，RTO的算法需要正确反映当前网络的拥塞情况，而每个连接的RTT恰恰能够反映这一点，所以设计好的RTT估计器是计算 RTO 的第一步。

由于路由器的拥塞和端系统负载的变化，测量得到的RTT\_sample是在波动的，因此我们借用TCP协议的规范，使用滑动平均法来估计RTT。当获取到一份RTT\_sample时，根据下式来更新估计RTT（称之为RTT\_Est）：

RTT\_Est = (1 - a)\* RTT\_Est + a \* RTT\_sample

其中，a通常取值为0.125，即：

RTT\_Est = 0.875 \* RTT\_Est + 0.125 \* RTT\_sample

即每个新的估计值的87.5%来自前一个估计值，而12.5%则取自新的测量。

**时延抖动估计：**

尽量减小时延抖动对于实时应用程序非常重要。浏览网页和邮件对时延抖动相当有抵抗力，但任何一种流媒体(语音、视频、音乐)都很容易受到时延抖动的影响。抖动是拥塞或没有足够的带宽来处理流量的一种症状。另外，时延抖动可能会对传输造成影响，产生一些不必要的重传。例如，由于某种原因，链路RTT增大，新测量RTT\_sample的权值只占RTT\_Est的12.5%，即使RTT\_sample变化很大，但其所占比重小，最后RTT\_Est的变化也不大，导致RTO的变化不大，造成RTO过小，产生不必要的重传。因此对RTT的抖动估计很有必要。我们借用TCP协议的规范，定义RTT抖动量RTT\_Dev，用于估算RTT\_sample一般会偏离RTT\_Est的程度：

RTT\_Dev = (1 - B) \* RTT\_Dev + B \* |RTT\_sample - RTT\_Est|

其中,B通常取值为0.25，即：

RTT\_Dev = 0.75 \* RTT\_Dev + 0.25 \* |RTT\_sample - RTT\_Est|

当RTT波动很大的时候，RTT\_Dev就会很大。

**设定重传超时时间RTO：**

得到了当前链路的RTT\_Est和RTT\_Dev后，即可设置RTO。RTO应满足如下要求：

1. RTO应该大于等于RTT\_Est，否则会产生大量不必要的重传。

2. 时延抖动大时，RTO应较大，抖动小时，RTO应较小，但不小于RTT\_Est。

基于上述要求，则可设定超时重传时间RTO的计算公式为：

RTO = RTT\_Est + C \* RTT\_Dev

其中，C通常取值为4，即：

RTO = RTT\_Est + 4 \* RTT\_Dev

若链路刚初始化，尚未收到ACK报文，RTT参数未知，则设置初始RTO为1秒，当出现超时后，超时重传时间将以指数退避的方法加倍，以免即将被确认的后继报文段过早出现超时。一旦收到ACK并更新RTT\_Est后，RTO便会按上式计算。假设当前超时重传定时器溢出时，与最早的未被确认的报文段相关联的RTO为0.75s，程序就会重传报文段，并将新的RTO设置为1.5s，如果1.5s后又溢出了，则程序将再次重传报文段，并把RTO设置为3秒。超时间隔在每次重传后会呈指数上升，然而每当重传定时器在另外两个事件（收到上层应用的数据和收到ACK）中的任意一个启动时，RTO将利用最新的RTT\_Est和RTT\_Dev重新计算。

对于上述公式中的参数，我们可以根据不同的业务场景灵活调整。如果业务场景是一个对延迟敏感但对流量成本要求不高的场景，就可以将 RTO 设计得比较小，达到尽可能保证低时延的目的。例如：实时操作类网游、教育领域的书写同步，是典型的用成本换时延和质量的场景。