**目录：**

**重传策略**

**RTT测量及RTO设置**

**快速重传（结合SACK）**

**伪重传（DSACK、Eifel、F-RTO，Eifel响应）**

**重新组包**

**TCP重传相关攻击**

重传超时（RTO）：

TCP在发送数据时会设置一个计时器，若超时仍未收到ACK，则会引发对应的重传操作。

快速重传：

通常发生在没有延时的情况下，若TCP累计确认无法返回新的ACK，或SACK表明出现失序报文段时，则会认为丢包，导致重传。

TCP每次重传间隔时间都会x2，称为二进制指数退避。

TCP拥有两个值来决定如何重传同一个报文段：1.尝试重传的次数 2.等待时间

在Linux中

对于一般数据段来说，可以通过net.ipv4.tcp\_retries1（重传次数，默认值3）和net.ipv4.tcp\_retries2（等待时间，默认值15，约为13~30分钟）来设置TCP重传参数；对于SYN报文段来说，通过设置net.ipv4.tcp\_syn\_retries和net.ipv4.tcp\_synack\_retries来设置SYN报文段和SYNACK报文段的重传次数。

### 设置重传超时：

TCP时常工作在动态的网络环境中，其RTT也会动态变化，因此需要动态设置RTO（重传超时）。

若RTO过小则会在网络中引入不必要的重复数据，若RTO过大会降低网络的利用率。

每个TCP连接的RTT均独立估算，并且重传计时器会对任何占用序列号的在传数据计时，包括SYN和FIN报文段。

#### 设置RTO的方法：

1. 经典方法：

平滑的RTT估计值：**SRTT = α \* SRTT+（1-α）\* RTT**

α为平滑因子，推荐值为0.8~0.9，每次得到新的RTT就会更新SRTT，这样新估计值10~20%来自于新测量的RTT值。

RTO的设置：**RTO = min(ubound, max(lbound, (SRTT \* β ))**

β为时延离散因子，ubound为上边界，lbound为下边界。

简单来说RTO为SRTT \* β，β推荐值为1.3~2.0。

这种算法对于比较稳定的网络来说能够取得不错的性能，但是对于RTT变化较大的网络中则无法取得预期的效果。

1. 标准方法：

改善了经典方法，通过记录RTT的变化情况以及均值来进行估值。

因为计算标准差需要对方差及逆行平方根运算，对于TCP来说代价较大，所以选择使用平均偏差来计算。

**srtt = (1-g)(srtt) + (g)rtt**

**rttvar = (1-h)(rttvar) + (h)(|rtt - srtt|)**

**RTO = srtt + 4(rttvar)**

在经典方法的基础上加上了rttvar，RTT变化时，偏差量越大，RTO越大。

初始值：**srtt = rtt，rttvar = rtt/2。**

1. 因为重传报文的ACK存在二义性，无法区分是原始报文还是重传报文的ACK，所以出现超时重传时的ACK不进行RTT的计算。

#### 使用时间戳选项测量RTT：

由于可能出现多个报文段使用一个ACK、出现数据丢失失序等情况，TCP报文段与ACK并不是严格的一对一关系。

所以接收端每次需要记录上一个发送的ACK号(LAST ACK)，新的报文段到达时，比较last ack与序列号的值，若相等则证明是“相邻”的数据，此时将时间戳回传。发送端比较时间戳与当前时间，得到RTT。

分组失序或丢失时，发送端会接收到重复ACK，RTT会被过高的估算，导致RTO更大，此时在一定程度上可以避免不必要的重传（后续报文段未接收到ack，避免后续报文的的重传）。

### 基于计时器的重传：

每个TCP连接的发送端会随着报文段的发送，不断的设定和取消一个重传计时器。

若在RTO时间内未收到计时报文段的ACK，则会触发超时重传，此时会降低发送速率来对此进行快速响应。

一种方法是基于拥塞控制机制减少发送窗口的大小。

另一种是暂时增大RTO的退避因子γ，RTO=γRTO。γ通常为1，随着多次重传，γ=2，4，8加倍增长。一旦接收到对应的ACK，γ会重置为1。

### 快速重传：

TCP发送端在接收到至少dupthresh（重复ACK阈值）个重复ACK后，立即重传可能丢失的数据分组，而不必等到重传计时器超时。

TCP接收端在接收到失序报文段时，需要立即生成重复ACK并返回。当采用SACK时，重复ACK通常也包含SACK信息，可以通过SACK获知多个空缺。

### 带选择确认（SACK）的重传：

使用SACK选项可以在一个RTT内获知多个空缺，减少不必要的重传。

1. SACK：接收端行为：

接收端收到SACK许可选项即可生成SACK，通常在出现失序数据时生成SACK。SACK记录的是已接收的序列号的范围。

1. SACK：发送端行为：

发送端除了记录累计ACK信息，也需要记录SACK信息，并利用该信息来避免重传已经正确接收的数据。

执行重传时最常用的方法是，首先填补接收端的空缺，然后再继续发送新数据。（若拥塞控制机制允许）

### 伪超时与重传：

伪超时：过早判定超时，导致即使没有出现数据丢失也引发了重传。

包失序、重复或ACK丢失也可能导致伪重传。此处仅讨论伪超时导致的伪重传。

#### 伪重传的判定：

1. 重复SACK（DSACK）扩展：

DSACK作用：可以在第一个SACK块中告知接收端接收到的重复报文段序列号。发送端通过DSACK推断是否发生了包失序、ACK丢失、包重复或伪重传。

1. Eifel检测算法：

通过时间戳来判断是否为伪重传，比DSACK能更早检测到伪重传行为。

超时重传触发后，Eifel等到相应分组的第一个ACK到达后，通过时间戳判断，若为原始传输的确认，则判定该重传是伪重传。

而DSACK需要等到重传报文段的ACK到达后才能判断出是否为伪重传。

1. 前移RTO恢复（Forward-RTO Recovery，F-RTO）：

是检测伪重传的标准方法，只检测由重传计时器超时引发的伪重传。

在出现一次基于计时器超时的重传后，这类行为针对的是没有收到ACK的最小序列号，会触发回退N（Back-N）行为。即TCP会继续按序发送相邻的分组。

而F-RTO会修改TCP的行为。

在超时重传后接收到相应分组的第一个ACK后，会发送新的（非重重更换）数据，之后再响应后一个到达的ACK。若两个ACK不为重复ACK则判定该重传为伪重传。

#### 伪重传的响应：

Eifel响应算法：判定出现伪重传时，引发的一套标准操作。

原则上超时重传和快速重传都可使用此响应算法，但目前只针对超时重传做了相关规定。

对于重传再次发生超时的情况，不会执行响应算法。

1. 伪超时：

对于Eifel检测算法或F-RTO，通过检查ACK或原始传输来实现，称为伪超时。

对于伪超时，TCP会将下一个报文段的序列号修改为最新的未发送过的报文段，以此来避免“BACK-N”行为。

迟伪超时：

对于DSACK等基于重传返回ACK来检测的方式，称为迟伪超时。

对于迟伪超时，TCP则进行“BACK-N”行为。

1. RTO的设置：

在任何一次计时器超时后都会指定，但只有出现伪重传时才会使用：

**srtt\_prev = srtt + 2G**

**rttvar\_prev = rttvar**

G代表TCP时钟粒度。

接收到重传计时器超时后发送的报文段的ACK后，需要按一下方式更新RTO：  
**srtt = max (srtt\_prev, m)**

**rttvar = max (rttvar\_prev, m/2)**

**RTO = srtt + max (G, 4(rttvar))**

m为RTT，基于超时后首个发送数据收到的ACK计算得到。

### 包失序：

在IP网络中出现包失序的原因在与IP层不能保证包传输是有序的。IP层可以选择传输速度更快的路径；在硬件方面一些高性能路由器会采用多个并行数据链路；不同的处理延时也会导致包失序。

失序问题可能存在于TCP的正向或反向（ACK）链路中。

若发生在反链路，则会使发送端窗口快速前移，并且收到一些重复而应被丢弃的ACK。由于TCP的拥塞控制行为，会导致发送端出现不必要的流量突发（瞬时的高速发送）行为，影响可用网络带宽。

如果失序发生在正向链路，TCP可能无法正确识别失序和丢包。由于快速重传机制的存在，当出现严重失序时，会导致伪重传出现。

网络中任何一个失序的数据报都会生成重复ACK。若接收到重复ACK就立即启动快速重传，就会导致大量不必要的重传。因此快速重传仅在达到重复阈值后才会触发。

区分丢包与失序不是一个很重要的问题。互联网中严重的失序并不常见，因此将重复阈值（dupthresh）设为相对较小（默认3）就能处理绝大部分情况。

### 包重复：

### 目的度量：