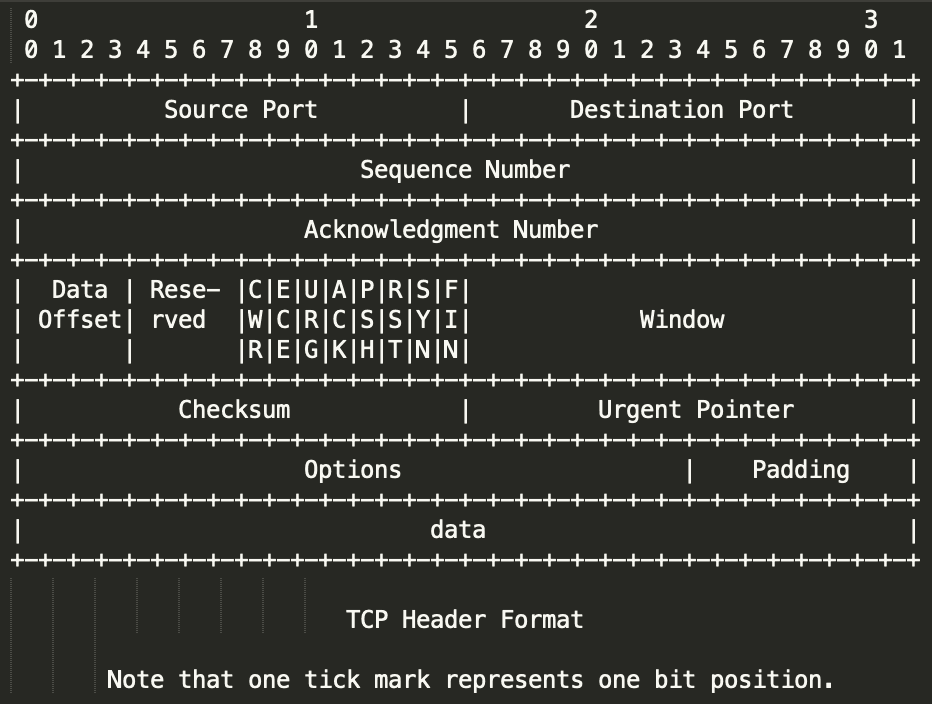
​https://mp.weixin.qq.com/s/KTKVu3uCC5MFlU5oylZPFA

TCP 拥塞控制详解

### 一、TCP 首部格式

在了解 TCP 的拥塞控制之前，先来看看 TCP 的首部格式和一些基本概念。

TCP 头部标准长度是 20 字节。包含源端口、目的端口、序列号、确认号、数据偏移、保留位、控制位、窗口大小、校验和、紧急指针、选项等。



#### 1.1 数据偏移（Data Offset）

该字段长 4 位，单位为 4 字节。表示为 TCP 首部的长度。所以 TCP 首部长度最多为 60 字节。

#### 1.2 控制位

目前的 TCP 控制位如下，其中 CWR 和 ECE 用于拥塞控制，ACK、RST、SYN、FIN 用于连接管理及数据传输。

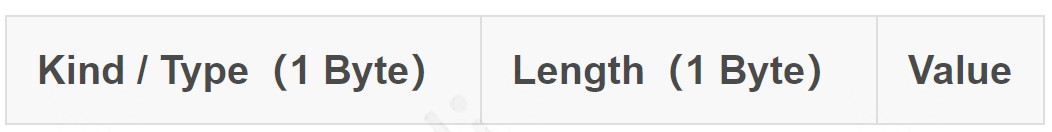
CWR：用于 IP 首部的 ECN 字段。ECE 为 1 时，则通知对方已将拥塞窗口缩小。  
ECE：在收到数据包的 IP 首部中 ECN 为 1 时将 TCP 首部中的 ECE 设置为 1，表示从对方到这边的网络有拥塞。  
URG：紧急模式  
ACK：确认  
PSH：推送，接收方应尽快给应用程序传送这个数据。没用到  
RST：该位为 1 表示 TCP 连接中出现异常必须强制断开连接。  
SYN：初始化一个连接的同步序列号  
FIN：该位为 1 表示今后不会有数据发送，希望断开连接。

#### 1.3 窗口大小（Window）

该字段长度位 16 位，即 TCP 数据包长度位 64KB。可以通过 **Options** 字段的 **WSOPT** 选项扩展到 1GB。

#### 1.4 选项（Options）

受 Data Offset 控制，长度最大为 40 字节。一般 Option 的格式为 TLV 结构：



常见的 TCP Options 有，SACK 字段就位于该选项中：



#### 1.5 SACK 选项

SACK 包括了两个 TCP 选项，一个选项用于标识是否支持 SACK，是在 TCP 连接建立时发送；另一种选项则包含了具体的 SACK 信息。

1. SACK\_Permitted 选项，该选项只允许在 TCP 连接建立时，有 SYN 标志的包中设置，也即 TCP 握手的前两个包中，分别表示通信的两方各自是否支持 SACK。

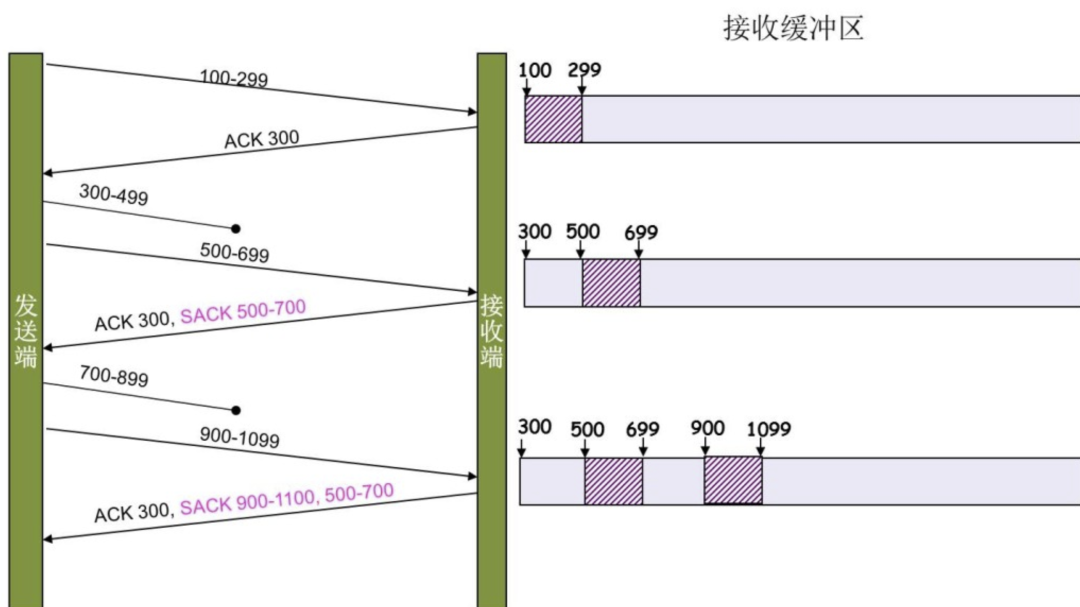
TCP SACK-Permitted Option:  
Kind: 4  
Length: Variable  
+----------+----------+  
| Kind=4 | Length=2 |  
+----------+----------+

2． SACK(选择性确认) 选项位于 Options 中。该选项参数告诉对方已经接收到并缓存的不连续的数据块，发送方可根据此信息检查究竟是哪些块丢失，从而发送相应的数据块。受 TCP 包长度限制，TCP 包头最多包含四组 SACK 字段。

TCP SACK Option:  
Kind: 5  
Length: Variable  
 +--------+--------+  
 | Kind=5 | Length |  
 +--------+--------+--------+--------+  
 | Left Edge Of lst Block |  
 +--------+--------+--------+--------+  
 | Right Edge Of lst Block |  
 +--------+--------+--------+--------+  
 | . . . |  
 +--------+--------+--------+--------+  
 | Left Edge Of nth Block |  
 +--------+--------+--------+--------+  
 | Right Edge Of nth Block |  
 +--------+--------+--------+--------+

1. SACK 的工作原理

如下图所示， 接收方收到 500-699 的数据包，但没有收到 300-499 的数据包就会回 SACK(500-700) 给发送端，表示收到 500-699 的数据。



### 二、滑动窗口和包守恒原则

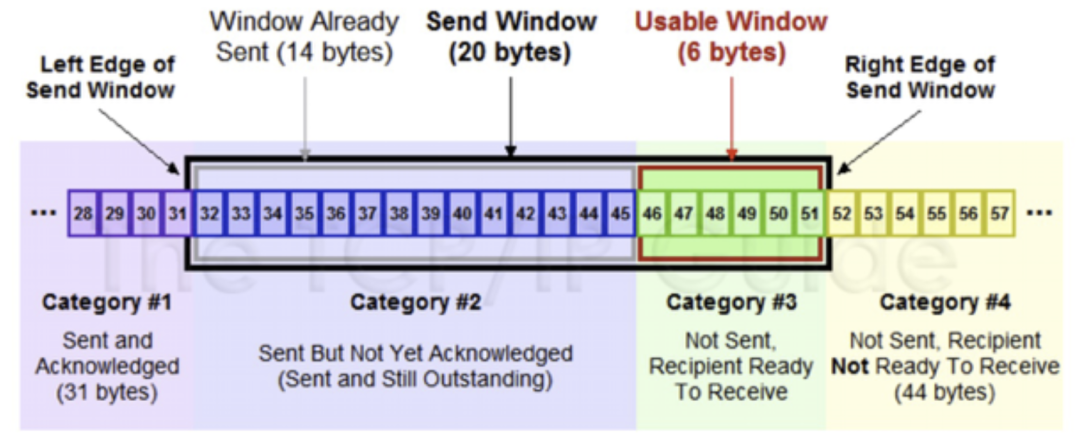
#### 2.1 滑动窗口

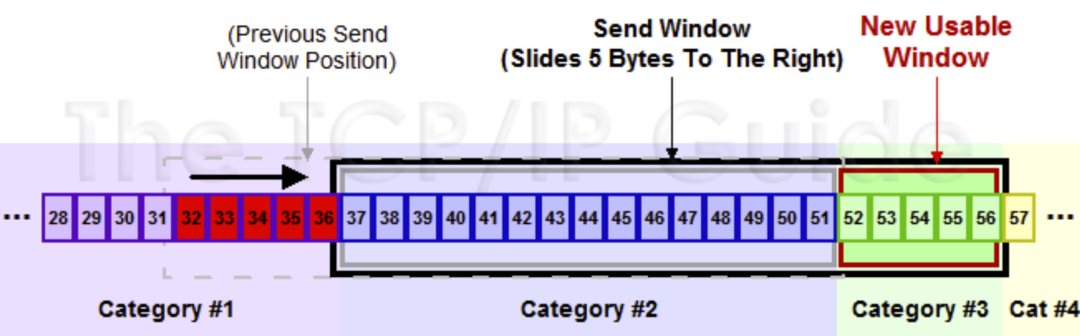
为了解决可靠传输以及包乱序的问题，TCP 引入滑动窗口的概念。在传输过程中，client 和 server 协商接收窗口 rwnd，再结合拥塞控制窗口 cwnd 计算滑动窗口 swnd。在 Linux 内核实现中，滑动窗口 cwnd 是以包为单位，所以在计算 swnd 时需要乘上 mss（最大分段大小）。



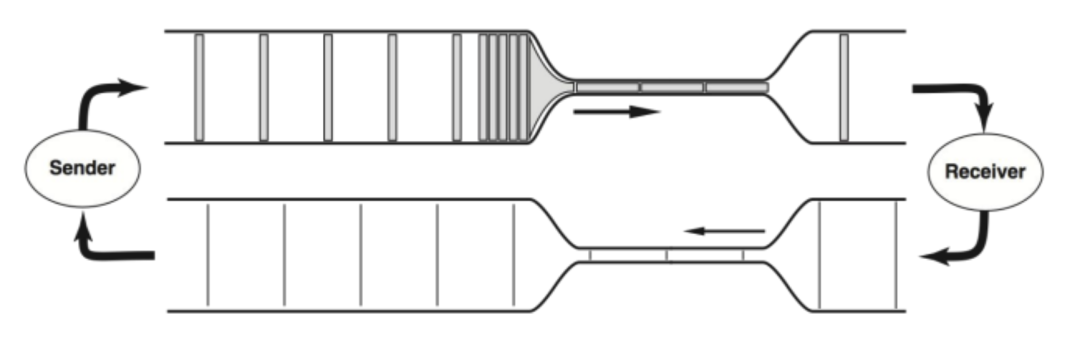
如下图所示滑动窗口包含 4 部分：

* 已收到 ack 确认的数据；
* 已发还没收到 ack 的；
* 在窗口中还没有发出的（接收方还有空间）；
* 窗口以外的数据（接收方没空间）。



滑动后的示意图如下（收到 36 的 ack，并发出了 46-51 的数据）：

#### 2.2 包守恒原则



TCP 维护一个发送窗口，估计当前网络链路上能容纳的数据包数量，希望在有数据可发的情况下，回来一个确认包就发出一个数据包，总是保持发送窗口那么多包在网络中流动。

传输的理想情况是要同时达到最大的吞吐量和最小的往返延迟，要达到这个目的，连接必须同时满足两个条件：

* 以链路瓶颈带宽 BtlBw 发包 （带宽利用率最高）
* 保证链路中没有缓存队列（延迟最低）

包守恒原则是拥塞控制的基础。

### 三、TCP 重传机制

本文重点介绍 TCP 拥塞控制相关，传输流程不在该范围之内，有兴趣的同学可以查阅相关文档。不过 TCP 重传逻辑和拥塞控制中的快速重传有关，所以在真正介绍拥塞控制算法之前，先来了解下 TCP 重传逻辑。

#### 3.1 超时重传 [RFC2988]

RTT（Round Trip Time）由三部分组成：链路的传播时间（propagation delay）、末端系统的处理时间、路由器缓存中的排队和处理时间（queuing delay）。TCP 发送端得到了基于时间变化的 RTT 测量值，就能据此设置 RTO。

当一个重传报文段被再次重传时，则增大 RTO 的退避因子  。通常情况下  值为 1，多次重传  加倍增长为 2，4，8 等。通常  不能超过最大退避因子，Linux 下 RTO 不能超过 TCP\_RTO\_MAX（默认为 120s）。一旦收到相应的 ACK，  重置为 1。

下面介绍几种常用的 RTT 算法。

##### 3.1.1 rtt 经典算法 [RFC793]

1）首先，先采样 RTT，记下最近几次的 RTT 值。2）然后做平滑计算 SRTT（ Smoothed RTT）。公式为：（其中的 α 取值在 0.8 到 0.9 之间，这个算法英文叫 Exponential weighted moving average，中文叫：加权移动平均）



2）开始计算 RTO。公式如下：



其中：

* UBOUND 是最大的 timeout 时间，上限值；
* LBOUND 是最小的 timeout 时间，下限值；
* β 值一般在 1.3 到 2.0 之间。

该算法的问题在于**重传时，是用重传的时间还是第一次发数据的时间和 ACK 回来的时间计算 RTT 样本值，另外，delay ack 的存在也让 rtt 不能精确测量**。

该算法的问题在于**重传时，是用重传的时间还是第一次发数据的时间和 ACK 回来的时间计算 RTT 样本值，另外，delay ack 的存在也让 rtt 不能精确测量**。

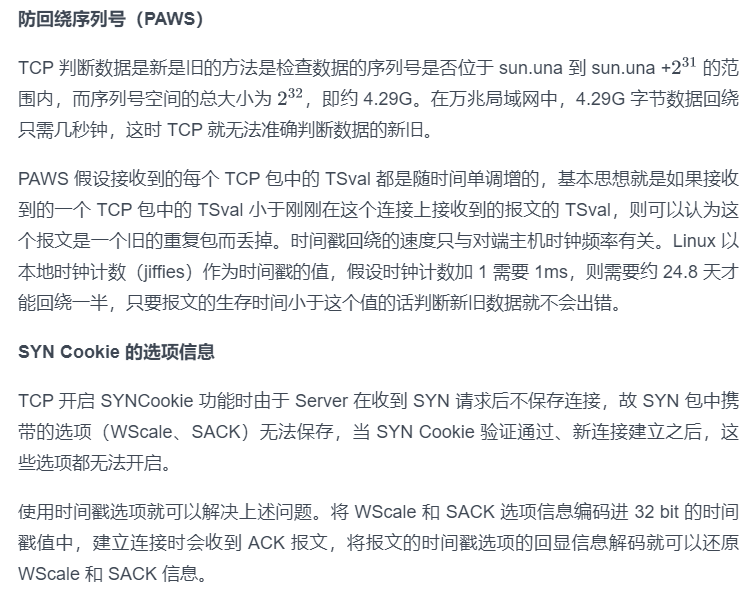
##### 3.1.4 TCP 时间戳选项（TSOPT）

根据 [RFC1323]，TSOPT 主要有两个用途，一个是 RTTM(round-trip time measurement)，即根据 ACK 报文中的这个选项测量往返时延；另外一个用途是 PAWS(protect against wrapped sequence numbers)，即防止同一个连接的系列号重叠。另外还有一些其他的用途，如 SYN-cookie、Eifel Detection Algorithm 等等。TSOPT 为 Options 选项中一种，格式如下：

Kind: 8  
 Length: 10 bytes  
 +-------+-------+---------------------+---------------------+  
 Kind=8 | 10 | TS Value (TSval) |TS Echo Reply (TSecr)|  
 +-------+-------+---------------------+---------------------+  
 1 1 4 4

**RTT 测量（RTTM）**

当使用这个选项的时候，发送方在 TSval 处放置一个时间戳，接收方则会把这个时间通过 TSecr 返回来。因为接收端并不会处理这个 TSval 而只是直接从 TSecr 返回来，因此不需要双方时钟同步。这个时间戳一般是一个单调增的值，[RFC1323]建议这个时间戳每秒至少增加 1。其中在初始 SYN 包中因为发送方没有对方时间戳的信息，因此 TSecr 会以 0 填充，TSval 则填充自己的时间戳信息。



#### 3.2 Fast Retransmit（快速重传）

快速重传算法概括为：TCP 发送端在观测到至少 dupthresh（一般为 3） 个重复 ACK 后，即重传可能丢失的数据分组，而不需等待重传计时器超时。

##### 3.2.1 SACK 重传

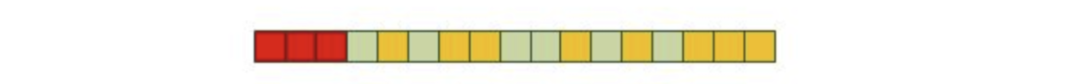
1. 未启用 SACK 时，TCP 重复 ACK 定义为收到连续相同的 ACK seq。[RFC5681]
2. 启用 SACK 时，携带 SACK 的 ACK 也被认为重复 ACK。[RFC6675]

如下图所示（绿色为已发送并且被 ack 的数据包，黄色表示已发送还未确认的数据包，浅绿色为被 Sack 确认的数据包，蓝色表示还未发送的数据包），设 dupthresh = 3，SACKed\_count = 6，从 unAcked 包开始的 SACKed\_count - dupthresh 个数据包，即 3 个数据包会被标记为 LOST。



拥塞窗口状态

记分板状态如下，红色表示该数据包丢失：



记分板状态

##### 3.2.2 FACK 重传

FACK 是 SACK 的一个激进版本，它拥有标准 SACK 算法的一切性质，除此之外，它**假设网络不会使数据包乱序**，因此收到最大的被 SACK 的数据包之前，FACK 均认为是丢失的。FACK 模式下，重传时机为 **被 SACKed 的包数 + 空洞数 > dupthresh**。

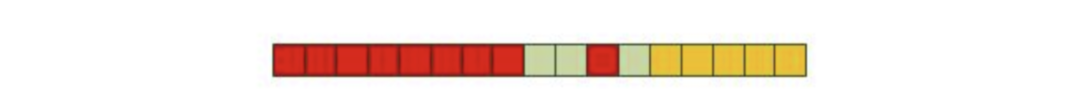
如下图所示，设 dupthresh = 3，FACKed\_count = 12，从 unACKed 包开始的 FACKed\_count

* dupthresh 个数据包，即 9 个包会被标记为 LOST。



拥塞窗口状态

记分板状态如下，红色表示该数据包丢失：



记分板状态

##### 3.2.3 RACK 重传

**基本思路** 如果数据包 p1 在 p2 之前发送，没有收到 p1 的确认，当收到 p2 的 Sack 时，推断 p1 丢包。

**算法简介** 每一个 skb 记录发送时间 xmit\_time，传输控制块维护全局变量：rack.xmit\_time，rack.reo\_wnd。rack.xmit\_time 是接收方已经收到的最新的那个 skb 的发送时间，rack.reo\_wnd 是乱序的时间窗口大小。

1.每次收到新的 ACK 后，更新 reo\_wnd，其中 rtt\_min 为固定时间窗口的 rtt 最小值。



2.每当收到一个 ACK 或者 SACK 的时候，更新 rack.xmit\_time。再去遍历发送队列上已经发送但还没有收到确认的数据包（skb），如果满足如下条件，那么标记这个数据包丢了。



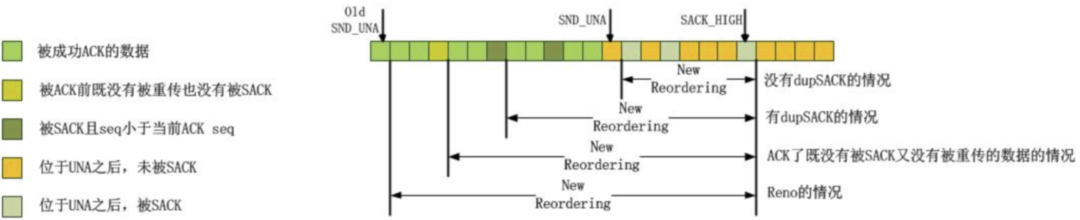
3.如果没有收到确认，那么就用定时器每个一段时间看看有哪些包丢了，如果满足如下条件，那么把这个 skb 标记为已经丢了：



注：目前 linux 内核中只实现了第一种判断方法，定时器还没有实现，这样的话就还没有解决对于尾部包丢失的问题。

##### 3.2.4 乱序检测

乱序检测的目的是探测网络是否发生重排导致的丢包，并以此来更新 dupthresh 值。只要能收到一个 ACK 或者 SACK，其序列号位于当前已经被 ACK 或 SACK 的数据包最大的序列号之前，就说明网络发生了重排造成了乱序，此时如果涉及的数据包大小大于当前能容忍的网络乱序值，即 dupthresh 值，就说明网络乱序加重了，此时应该更新 dupthresh 值。之所以保持 dupthresh 的值递增，是考虑其初始值 3 只是一个经验值，既然真实检测到乱序，如果其值比 3 小，并不能说明网络的乱序度估计偏大，同时 TCP 保守地递增乱序度，也是为了让快速重传的进入保持保守的姿态，从而增加友好性。



一旦发现 dupthresh 值更新的情形，FACK 的假设便不成立，必须在连接内永久禁用 FACK 算法。

#### 3.3 Early Retransmit for TCP（ER 机制）

**要解决的问题:** 当无法收到足够的 dupack 时，TCP 标准的 Fast Retransmit 机制无法被触发，只能等待 RTO 超时才能进行丢包的重传。而 RTO 超时不管是时间等待代价，还是性能损耗代价都很大。

**解决方法:** 检测出无法收到足够 dupack 的场景，进而降低 dupack threshold 来触发快速重传。从而避免等待 RTO 超时重传，对性能造成较大的损耗。

总结出现 dupack 不够的情况：

a. cwnd 较小

b. 发送窗口里大量的数据包都被丢失了

c.在数据发送的尾端发生丢包时

但是，上面各种极端的 case 有共同的特点：

m. 无法产生足够的 dupack

n.没有新的数据包可以发送进入网络

ER 机制就是在判断条件 m 和 n 都成立后，选择降低触发 Fast Retransmit 的阈值，来避免只能通过 RTO 超时重传的问题。

#### 3.4 TCP Tail Loss Probe（TLP 算法）

ER 算法解决了 dupack 较少时无法触发快速重传的问题，但当发生尾丢包时，由于尾包后没有更多数据包，也就无法触发 dupack。TLP 算法通过发送一个 loss probe 包，以产生足够的 SACK/FACK 数据包来触发重传。TLP 算法会在 TCP 还是 Open 态时设置一个 Probetimeout（PTO），当链路中有未被确认的数据包，同时在 PTO 时间内未收到任何 ACK，则会触发 PTO 超时处理机制。TLP 会选择传输序号最大的一个数据包作为 tail loss probe 包，这个序号最大的包可能是一个可以发送的新的数据包，也可能是一个重传包。TLP 通过这样一个 tail loss probe 包，如果能够收到相应的 ACK，则会触发 FR 机制，而不是 RTO 机制。

#### 3.5 伪超时与重传

在很多情况下，即使没有出现数据丢失也可能引发重传。这种不必要的重传称为伪重传，其主要造成原因是伪超时，即过早判定超时，其他因素如包失序、包重复，或 ACK 丢失也可能导致该现象。在实际 RTT 显著增长，超过当前 RTO 时，可能出现伪超时。在下层协议性能变化较大的环境中（如无线环境），这种情况出现得比较多。

TCP 为处理伪超时问题提出了许多方法。这些方法通常包含检测算法与响应算法。检测算法用于判断某个超时或基于计时器的重传是否真实，一旦认定出现伪超时则执行响应算法，用于撤销或减轻该超时带来的影响。检测算法包含 DSACK 、Eifel 检测算法、迁移 RTO 恢复算法(F-RTO) 三种。

##### 3.5.1 DSACK 扩展

DSACK 的主要目的是判断何时的重传是不必要的，并了解网络中的其他事项。通过 DSACK 发送端至少可以推断是否发生了包失序、 ACK 丢失、包重复或伪重传。D-SACK 使用了 SACK 的第一个段来做标志， a. 如果 SACK 的第一个段的范围被 ACK 所覆盖，那么就是 D-SACK。b.如果 SACK 的第一个段的范围被 SACK 的第二个段覆盖，那么就是 D-SACK。RFC2883]没有具体规定发送端对 DSACK 怎样处理。[RFC3708]给出了一种实验算法，利用 DSACK 来检测伪重传，响应算法可采用 Eifel 响应算法。

##### 3.5.2 Eifel 检测算法 [RFC3522]

实验性的 Eifel 检测算法利用了 TCP 的 TSOPT 来检测伪重传。在发生超时重传后，Eifel 算法等待接收下一个 ACK，若为针对第一次传输（即原始传输）的确认，则判定该重传是伪重传。

与 DSACK 的比较：利用 Eifel 检测算法能比仅采用 DSACK**更早检测到伪重传行为**，因为它判断伪重传的 ACK 是在启动丢失恢复之前生成的。相反， DSACK 只有在重复报文段到达接收端后才能发送，并且在 DSACK 返回至发送端后才能有所响应。及早检测伪重传更为有利，它能使发送端有效避免“回退 N”行为。

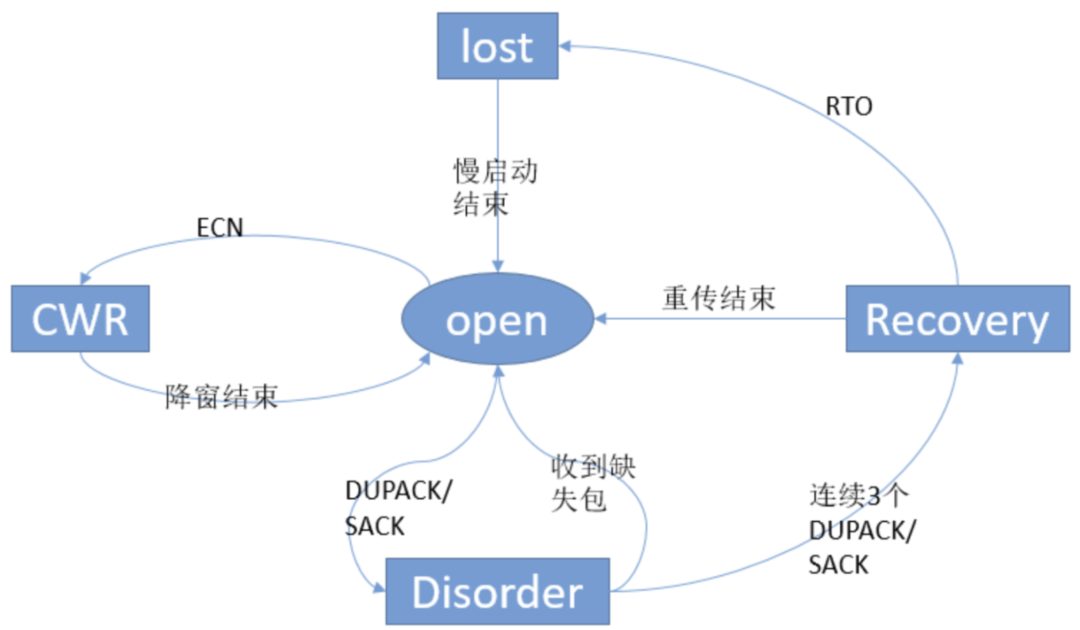
##### 3.5.3 迁移 RTO 恢复算法(F-RTO)

前移 RTO 恢复（Forward-RTO Recovery，F-RTO）[RFC5682]是检测伪重传的标准算法。它不需要任何 TCP 选项，因此只要在发送端实现该方法后，即使针对不支持 TSOPT 的接收端也能有效地工作。该算法**只检测由重传计时器超时引发的伪重传**，对之前提到的其他原因引起的伪重传则无法判断。

F-RTO 的工作原理如下：1. F-RTO 会修改 TCP 的行为，在超时重传后收到第一个 ACK 时，TCP 会发送新（非重传）数据，之后再响应后一个到达的 ACK。2.如果其中有一个为重复 ACK，则认为此次重传没问题。3. 如果这两个都不是重复 ACK，则表示该重传是伪重传。4.重复 ACK 是在接收端收到失序的报文段产生的。这种方法比较直观。如果新数据的传输得到了相应的 ACK，就使得接收端窗口前移。如果新数据的发送导致了重复 ACK，那么接收端至少有一个或更多的空缺。这两种情况下，接收新数据都不会影响整体数据的传输性能。

### 四、拥塞状态机

TCP 通过拥塞状态机来决定收到 ACK 时 cwnd 的行为（增长或者降低）。TCP 拥塞状态机有 Open,Disorder,Recovery,Lost和CWR五种状态。



#### 4.1 Open

当网络中没有发生丢包，也就不需要重传，sender 按照慢启动或者拥塞避免算法处理到来的 ACK。

#### 4.2 Disorder

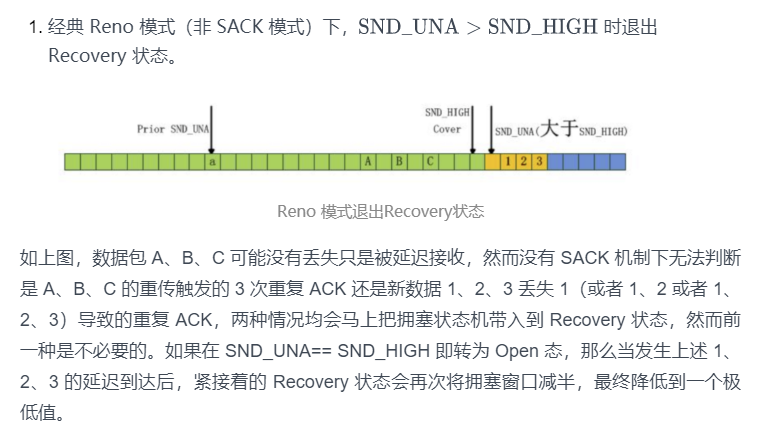
当 sender 检测到 dupack 或者 SACK，将会转移到 Disorder 状态，当处在这个这个状态中时，cwnd 将维持不变。每收到一个 dupack 或 SACK，发送方将发送一个新包。

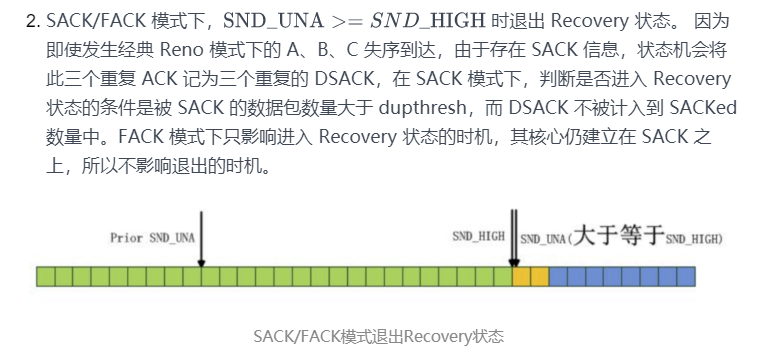
#### 4.3 CWR

当 sender 收到 ACK 包含显示拥塞通知（ECN），这个 ECN 由路由器写在 IP 头中，告诉 TCP sender 网络拥塞，sender 不会立马降低 cwnd，而是根据快速恢复算法进行降窗，直到减为之前的一半。当 cwnd 正在减小 cwnd，网络中有没有重传包时，这个状态就叫 CWR，CWR 可以被 Recovery 或者 Loss 中断。

#### 4.4 Recovery

当 sender 因为快速重传机制触发丢包时，sender 会重传第一个未被 ACK 的包，并进入 Recovery 状态。在 Recovery 状态期间，cwnd 的处理同 CWR 大致一样，要么重传标记了 lost 的包，要么根据保守原则发送新包。直到网络中所有的包都被 ACK，才会退出 Recovery 进入 Open 状态，Recovery 状态可以被 loss 状态打断。



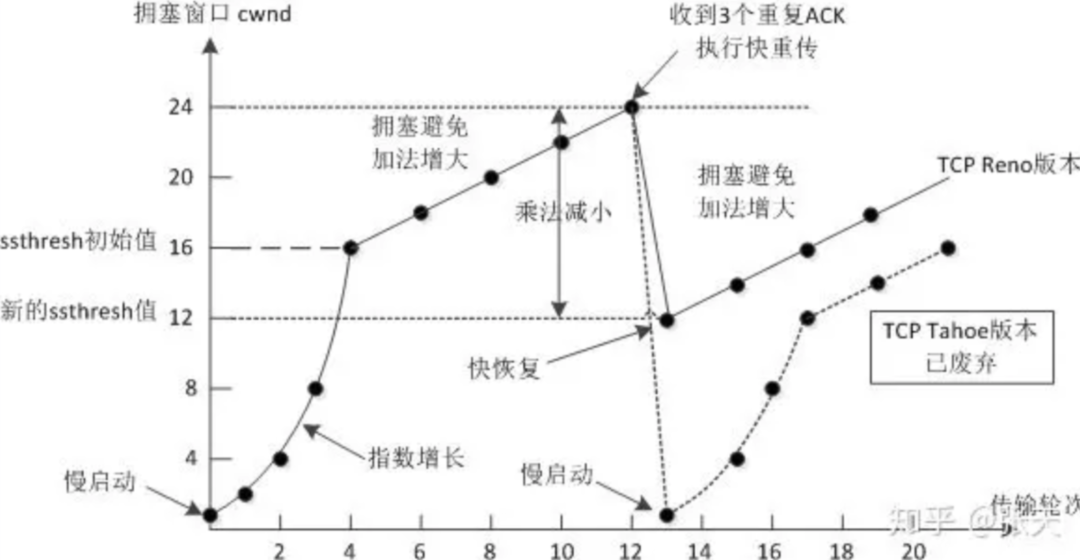


#### 4.5 Loss

当 RTO 后，TCPsender 进入 Loss 状态，所有在网络中的包被标记为 lost，cwnd 重置为 1，通过 slow start 重新增加 cwnd。Loss 与 Recovery 状态的不同点在于，cwnd 会重置为 1，但是 Recovery 状态不会，Recovery 状态下拥塞控制通过快速恢复算法逐步降低 cwnd 至 sshthresh。Loss 状态不能被其它任何状态中断，只有当网络中所有的包被成功 ACK 后，才能重新进入 Open 状态。

### 五、拥塞控制

拥塞的发生是因为路由器缓存溢出，拥塞会导致丢包，但丢包不一定触发拥塞。拥塞控制是快速传输的基础。一个拥塞控制算法一般包括慢启动算法、拥塞避免算法、快速重传算法、快速恢复算法四部分。



#### 5.1 慢启动算法

不同拥塞算法慢启动的逻辑有所不同，经典的 NewReno 慢启动的算法如下：

1. 连接建好的开始先初始化 cwnd = 10，表明可以传 10 个 MSS 大小的数据。
2. 每当收到一个 ACK，cwnd 加 1。这样每当过了一个 RTT，cwnd 翻倍，呈指数上升。
3. 还有一个 ssthresh（slow start threshold），是一个上限。当 cwnd >=ssthresh 时，就会进入“拥塞避免算法”。

Linux 3.0 后采用了 [**Google 的论文《An Argument for Increasing TCP’s Initial Congestion Window》**](https://static.googleusercontent.com/media/research.google.com/zh-CN/pubs/archive/36640.pdf)的建议——把 cwnd 初始化成了 10 个 MSS。而 Linux 3.0 以前，比如 2.6，Linux 采用了[RFC3390] 的建议，cwnd 跟 MSS 的值来变，如果 MSS < 1095，则 cwnd = 4；如果 MSS >2190，则 cwnd = 2；其它情况下，则是 3。

#### 5.2 拥塞避免算法

当 cwnd 增长到 sshthresh 时，就会进入“拥塞避免算法”。拥塞避免算法下 cwnd 成线性增长，即每经过一个往返时间 RTT 就把发送方的拥塞窗口 cwnd 加 1，而不是加倍。这样就可以避免拥塞窗口快速增长的问题。

每收到一个 ack 时 cwnd 的变化：  
cwnd = cwnd + 1 / cwnd

#### 5.3 快速重传算法

快速重传算法主要用于丢包检测，以便能更快重传数据包，更早的调整拥塞状态机状态，从而达到持续升窗的目的。具体重传策略见第三节 重传机制。

#### 5.4 快速恢复算法

当检测到丢包时，TCP 会触发快速重传并进入降窗状态。该状态下 cwnd 会通过快速恢复算法降至一个合理值。从历史发展来看，分为四个个阶段。

##### 5.4.1 BSD 初始版本

1. 收到 3 次重复 ACK，ssthresh 设为 cwnd/2，cwnd = cwnd / 2 + 3;
2. 每收到一个重复 ACK，窗口值加 1;
3. 收到非重复 ACK，窗口设为 ssthresh，退出

优点：在快速恢复期间，可以尽可能多的发送数据缺点：由于快速恢复未完成，尽可能多发送可能会加重拥塞。#### 5.4.2 [RFC3517]版本 1) 收到 3 次重复 ACK，ssthresh 设为 cwnd/2，cwnd = cwnd / 2 + 3; 2)**每收到一个重复 ACK，窗口值加 1/cwnd**; 3) 收到非重复 ACK，窗口设为 ssthresh，退出。

优点：在快速恢复期间，可以尽少量的发送数据（有利于拥塞恢复），且在快速恢复时间段的最后阶段，突发有利于抢带宽。

缺点：快速恢复末期的突发不利于公平性。

##### 5.4.2 Linux rate halving 算法

Linux 上并没有按照 [RFC3517] 标准实现，而是做了一些修改并运用到内核中。

1.收到 3 次重复 ACK，**sshthresh 设置为 cwnd/2，窗口维持不变**。2.每收到两个 ACK（不管是否重复），窗口值减 1：cwnd = cwnd - 1。3.新窗口值取 cwnd = MIN(cwnd, in\_flight+1)。4.直到退出快速恢复状态，cwnd = MIN(cwnd, ssthresh)。

优点：在快速恢复期间，取消窗口陡降过程，可以更平滑的发送数据 缺点：降窗策略没有考虑 PIPE 的容量特征，考虑一下两点：

a.如果快速恢复没有完成，窗口将持续下降下去 b.如果一次性 ACK/SACK 了大量数据，in\_flight 会陡然减少，窗口还是会陡降，这不符合算法预期。

##### 5.4.3 prr 算法 [RFC6937]

PRR 算法是最新 Linux 默认推荐的快速恢复算法。prr 算法是一种按比例降窗的算法，其最终效果是：

1.在快速恢复过程中，拥塞窗口非常平滑地向 ssthresh 收敛；2.在快速恢复结束后，拥塞窗口处在 ssthresh 附近

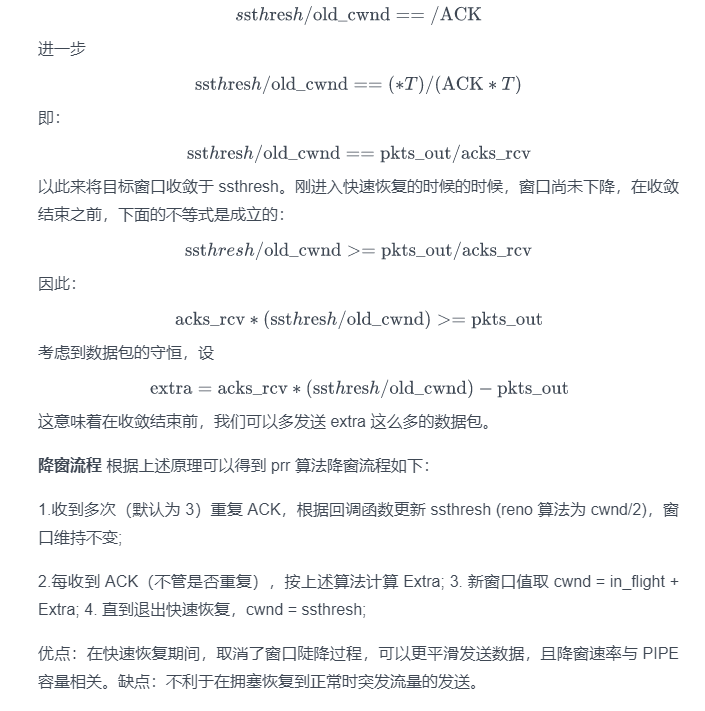
PRR 降窗算法实时监控以下的变量：in\_flight：它是窗口的一个度量，in\_flight 的值任何时候都不能大于拥塞窗口的大小。

prr\_delivered：本次收到 ACK 进入降窗函数的时候，一共被 ACK 或者 SACK 的数据段数量。它度量了本次从网络中清空了哪些数据段，从而影响 in\_flight。

prr\_out：进入快速恢复状态后已经被发送了多少数据包。在 transmit 例程和 retransmit 例程中递增。

to\_be\_out：当前还可以再发多少数据包。

**算法原理**根据**数据包守恒原则**，能够发送的数据包总量是本次接收到的 ACK 中确认的数据包的总量，然而处在拥塞状态要执行的并不是这个守恒发送的过程，而是降窗的过程，因此需要在被 ACK 的数据包数量和可以发送的数据包数量之间打一个折扣，PRR 希望达到的目标是：



#### 5.5 记分板算法

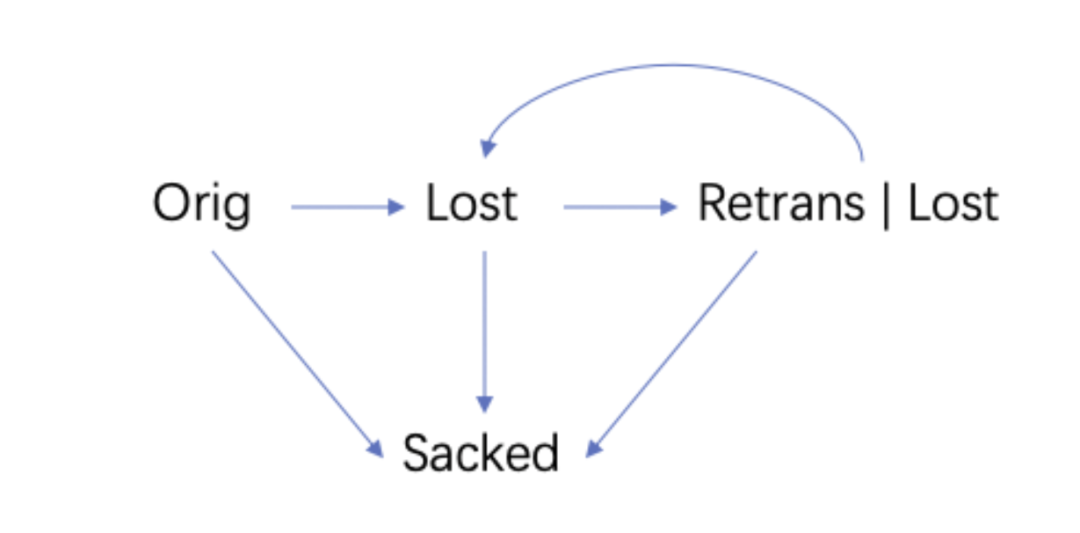
记分板算法是为了统计网络中正在传输的包数量，即tcp\_packets\_in\_flight。只有当 cwnd > tcp\_packets\_in\_flight 时，TCP 才允许发送重传包或者新数据包到网络中。tcp\_packets\_in\_flight和packets\_out, sacked\_out, retrans\_out,lost\_out有关。其中packets\_out表示发出去的包数量，sacked\_out为sack的包数量，retrans\_out为重传的包数量，lost\_out为loss的包数量，这里的loss包含rto,FR和RACK等机制判断出来的丢失包。



为了可以正确统计这些数据，内核给每个 tcp 包(tcp\_skb\_cb)添加了sacked字段标记该数据包当前的状态。

\_\_u8 sacked; /\* State flags for SACK. \*/  
#define TCPCB\_SACKED\_ACKED 0x01 /\* SKB ACK'd by a SACK block \*/  
#define TCPCB\_SACKED\_RETRANS 0x02 /\* SKB retransmitted \*/  
#define TCPCB\_LOST 0x04 /\* SKB is lost \*/  
#define TCPCB\_TAGBITS 0x07 /\* All tag bits \*/  
#define TCPCB\_REPAIRED 0x10 /\* SKB repaired (no skb\_mstamp\_ns) \*/  
#define TCPCB\_EVER\_RETRANS 0x80 /\* Ever retransmitted frame \*/  
#define TCPCB\_RETRANS (TCPCB\_SACKED\_RETRANS|TCPCB\_EVER\_RETRANS| \  
 TCPCB\_REPAIRED)

需要在意的有TCPCB\_SACKED\_ACKED（被 SACK 块 ACK'd）,TCPCB\_SACKED\_RETRANS(重传),TCPCB\_LOST（丢包），其状态转换图如下：



记分板状态转换逻辑如下：

1. 首先判定丢包，打L tag，lost\_out++，即 L
2. 如果需要重传，打Rtag，retrans\_out++，即 L|R
3. 如果再次丢包，取消Rtag，retrans\_out–，lost\_out 维持不变，go to step2，此时标记位为 L
4. 当 SACKED 时，取消L|R，retrans\_out–，lost\_out–，此时为 S
5. 当 snd\_una 向右更新时，packet\_out–

#### 5.6 拥塞窗口校验

在 [RFC2861] 中，区分了 TCP 连接数据传输的三种状态：

* network-limited：TCP 的数据传输受限于拥塞窗口而不能发送更多的数据。
* application-limited：TCP 的数据传输速率受限于应用层的数据写入速率，并没有到达拥塞窗口上限。
* idle：发送端没有额外的数据等待发送，当数据发送间隔超过一个 RTO 的时候就认为是 ilde 态。

cwnd 代表了对网络拥塞状态的一个评估，拥塞控制要根据 ACK 来更新 cwnd 的前提条件是，当前的数据发送速率真实的反映了 cwnd 的状况，也就是说当前传输状态是 network-limited。假如 tcp 隔了很长时间没有发送数据包，即进入 idle，那么当前真实的网络拥塞状态很可能就会与 cwnd 反映的网络状况有差距。另外在 application-limited 的场景下，受限数据的 ACK 报文还可能把 cwnd 增长到一个异常大的值，显然是不合理的。基于上面提到的两个问题，[RFC2861]引入了拥塞窗口校验(CWV，Congestion Window Validation)算法。

