

# IP（网际互连协议）

## UDP（用户数据报协议）

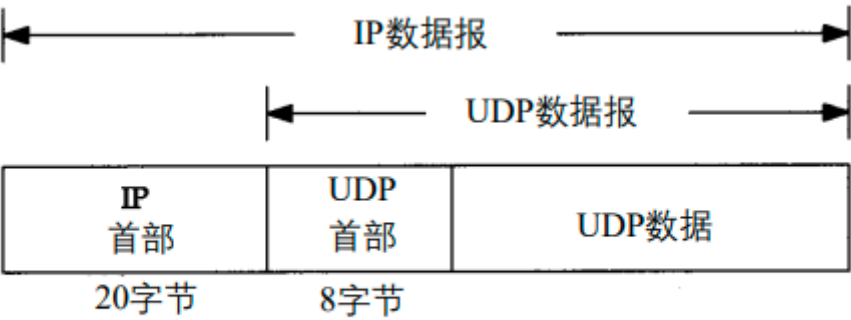


图11-1 UDP封装

## UDP首部

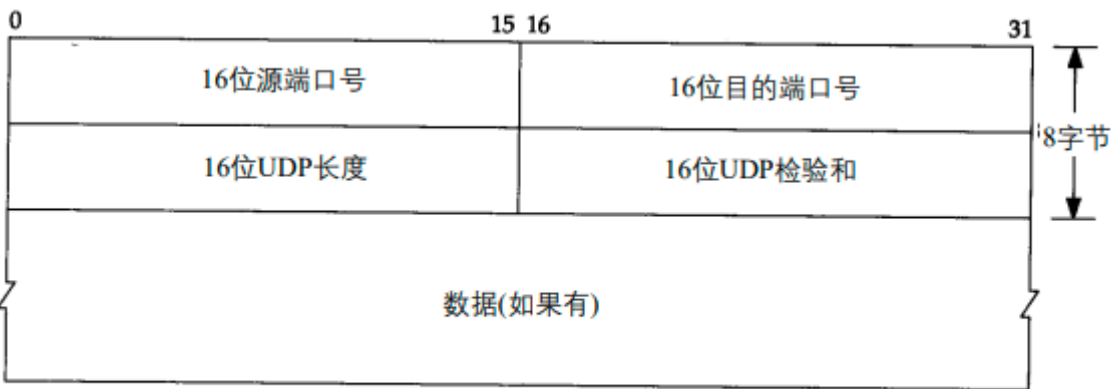


图11-2 UDP首部

## UDP校验和

UDP和TCP校验和不仅覆盖整个报文，而且还有12字节的IP伪首部

- 源IP地址(4字节)

- 目的IP地址(4字节)
- 协议(2字节, 第一字节补0)
- TCP/UDP包长(2字节)
- 另外UDP、TCP数据报的长度可以为奇数字节, 所以在计算校验和时需要在最后增加填充字节0 (注意, 填充字节只是为了计算校验和, 可以不被传送)。

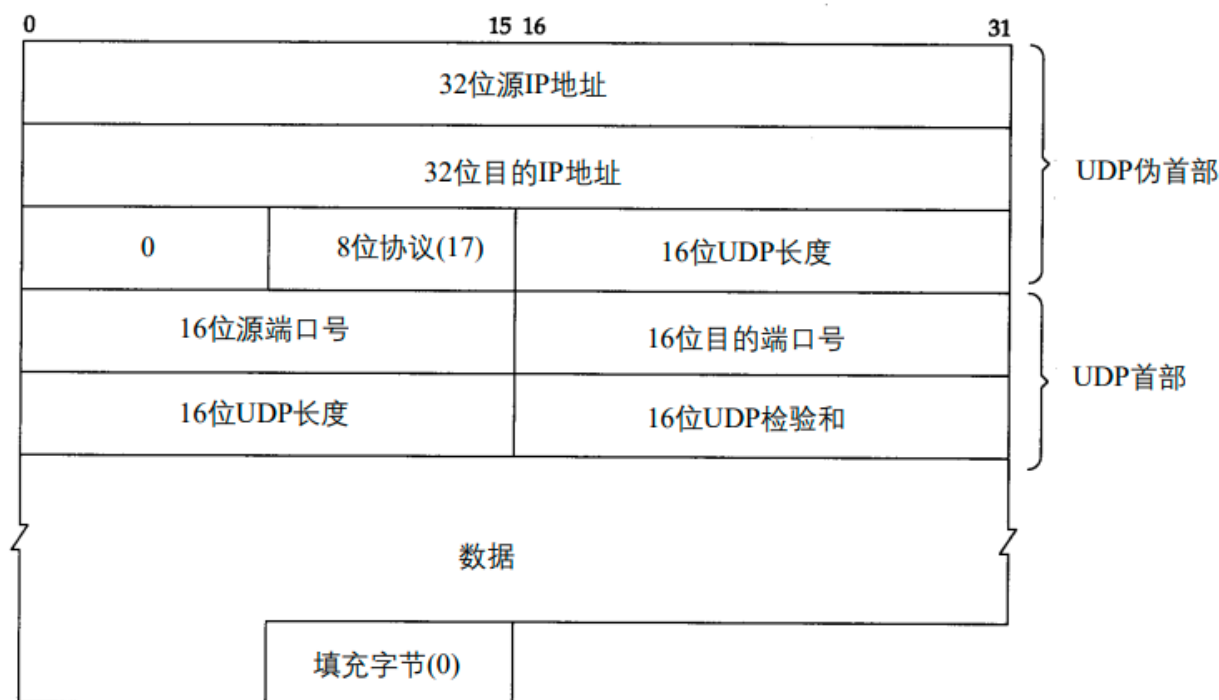


图11-3 UDP校验和计算过程中使用的各个字段

## UDP数据报分片

为什么分片?

因为每个类型的网络的MTU不同, 当一个IP数据报的大小 (IP报头+UDP报头+数据) 超过了该网络的MTU的大小,就需要对数据报分片, 然后进行传输。(数据报分片自身也可被再次分片)

MTU (Maximum Transmission Unit)

是指特定网络协议下的最大传输单元, 它是一种用于控制网络中数据包的大小。MTU 的大小决定了网络中传输的最大数据包的大小, 它是网络性能的重要参数。MTU 通常是以字节为单位来衡量的, 常见的 MTU 值有 1500 字节、576 字节等。MTU 的大小可以通过更改网络设备的配置来调整, 但具体的配置方法取决于使用的网络协议。

特点

1. 只有第一片具有运输层首部
2. 多个片到达目的端可能会失序, 因为每个片都有自己的IP首部, 在选择路由的时候都是独立的

- 3. 只有到达目的端才会开始重组
- 4. 当一个数据报的任何一个分片首先到达，IP层启动一个定时器，且收到新的分片也不会被重置，定时给出了同一数据报分片之间可被分隔的最大间隔时间
- 5. 如果任何一个分片丢失了（通过超时判断），认为整个数据报就丢失了。需要重发整个数据报



- 6. 分片后每个片的标识字段【Identification】的值是一样的
- 7. DF: “不分片”标志位；MF: “更多的片”标志位
- 8. 除最后一块外，每一片中的数据部分（除IP首部外其余部分）必须是8字节的整数倍（因为13位的片偏移值【Fragment offset】表示的偏移字节数是该值乘以8）

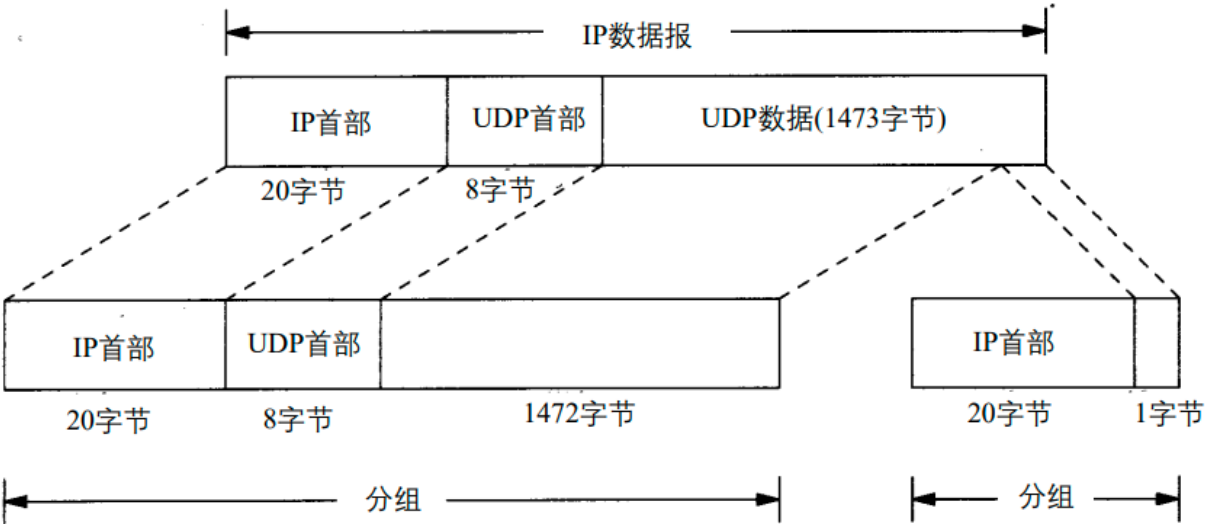


图11-8 UDP分片举例

ICMP不可达差错（需要分片）

当路由器收到一份需要分片的数据报，而在IP首部又设置了不分片（DF）的标志比特，就会产生ICMP不可达差错

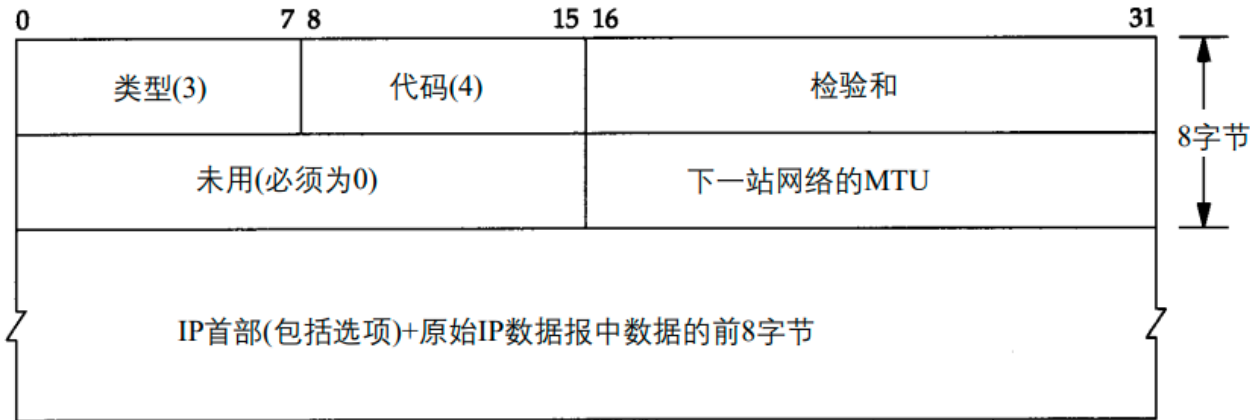


图11-9 需要分片但又设置不分片标志比特时的ICMP不可达差错报文格式


## 最大UDP数据报长度

IP数据报的最大长度是65535字节，这是由IP首部16比特总长度字段所限制的。去除20字节的IP首部和8个字节的UDP首部，UDP数据报中用户数据的最长长度为65507字节。但是，大多数实现所提供的长度比这个最大值小。

1. 应用程序可能会受到其程序接口的限制
2. 第二个限制来自于TCP/IP的内核实现。可能存在一些实现特性（或差错），使IP数据报长度小于65535字节。

## ICMP源站抑制差错

当一个系统（路由器或主机）接收数据报的速度比其处理速度快时，可能产生这个差错。注意限定词“可能”。即使一个系统已经没有缓存并丢弃数据报，也不要求它一定要发送源站抑制报文。

 使用UDP时很容易产生这样的ICMP差错

dajfkljsajfd;klasdfhuanghhaugnfulijlkjaskldfzenmezenhaungfulingfhaugndkkaldkfjkdjfaldfkfpingpignzed  
kjdsafkjfkwhaugnfu

## 补充

1. 每次IP数据报的标识位不同（不同分片具有相同的IP首部）。第二次发送的第3片和第4片不和第一次发送的第1片和第2片重组成一份IP数据报。

## TCP（传输控制协议）

 参考

TCP/IP详解 卷一（第一版）

[一文解析TCP协议所有知识点 - 知乎 \(zhihu.com\)](#)

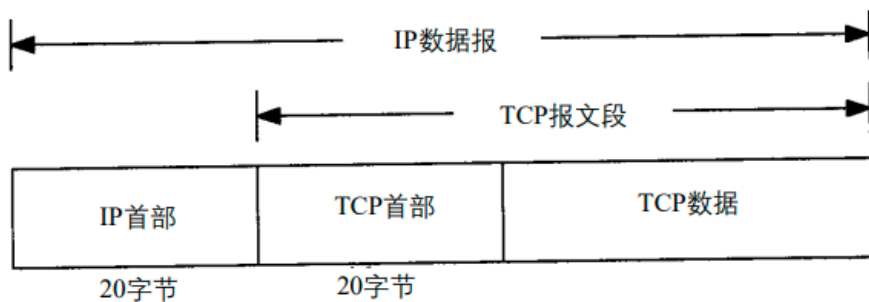


图17-1 TCP数据在IP数据报中的封装

## TCP首部

不计任选字段，它通常是**20**个字节

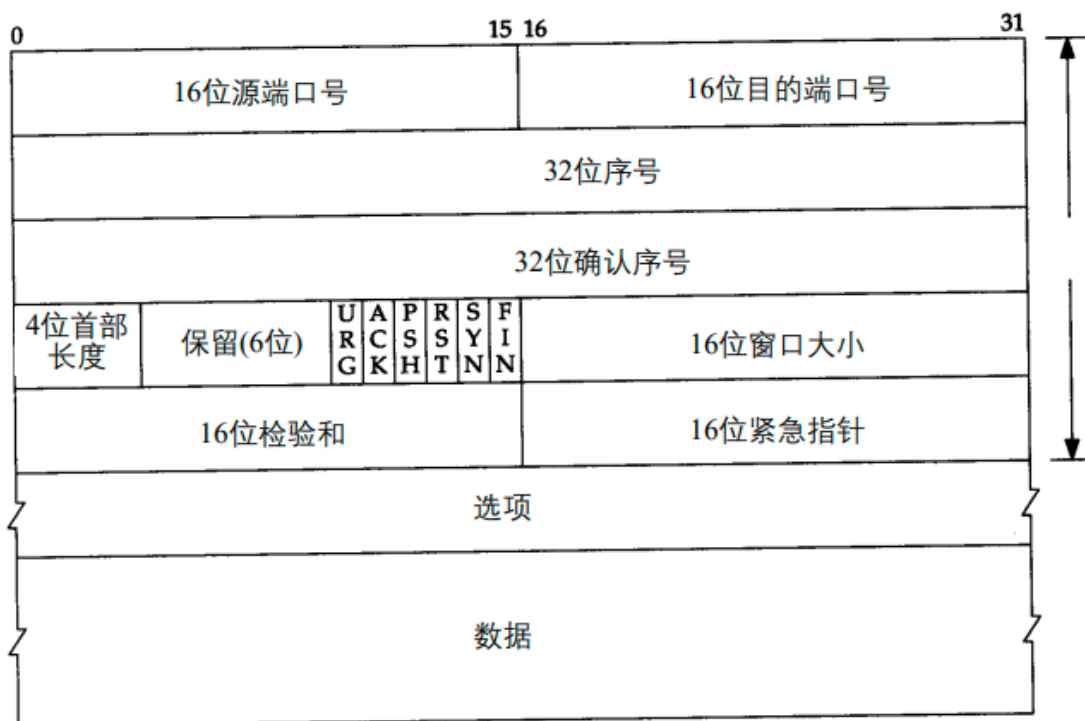


图17-2 TCP包首部

32位序号：

对传输的每个字节进行计数

当SYN标志为1，这个序号是建立连接的初始序号（ISN）

一个SYN或者是FIN标志都会消耗一个序号

32位确认序号：

只有ACK标志为1时有效

是上次成功收到数据的字节序号（seq）加1

4位首部长度：

指示TCP首部的长度。

它的值是首部中32位字的数目。

0101:  $5 \times 32\text{bit} = 160\text{bit} = 20\text{字节}$  (最小)

1111:  $15 \times 4\text{byte} = 60\text{字节}$  (最大)

6个标志比特:

- URG 紧急指针 (urgent pointer) 有效
- ACK 确认序号有效
- PSH 接收方应该尽快将这个报文段交给应用层
- RST 重新连接
- SYN 同步序号用来发起一个连接 (三次握手)
- FIN 发端完成发送任务 (四次挥手)

窗口大小:

TCP的流量控制是由连接的双方通过设定窗口大小来控制的。

这个值是接收端期望收到的字节数。

16位校验和:

校验和覆盖了整个TCP报文段: TCP首部和TCP数据。 (强制性)

16位紧急指针:

只有当URG标志置1时紧急指针才有效。

紧急指针是一个正的偏移量, 和序号字段中的值相加就能确定紧急数据最后一个字节的序号。

TCP紧急方式是发送端向另一端发送紧急数据的一种方式。

## 三次握手

SYN 1415531521:

客户端建立连接的初始序号ISN

请求段发送一个SYN段指明客户打算连接的服务器的端口, 以及初始序号ISN。

SYN 1823083521:

服务端建立连接的初始序号ISN (seq)

服务器发回包含服务器的初始序号的SYN报文段作为应答

ack 1415531522:

确认序号 = 客户的ISN加1

服务器对客户的SYN报文段进行确认。

ack 1823080522:

确认序号 = 服务的ISN加1

客户对服务器的SYN报文进行确认

<mss 1024>:

表示发端指明的最大报文段长度选项。

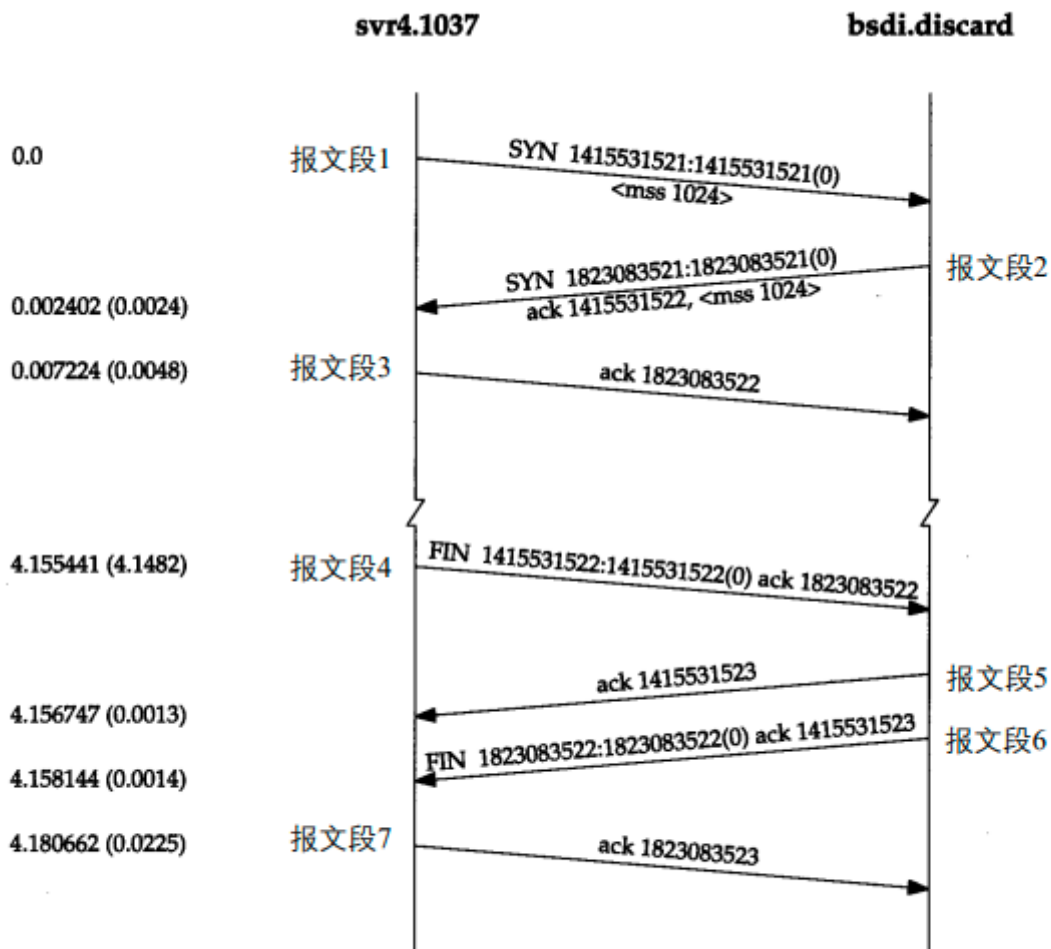


图18-3 连接建立与终止的时间系列

## 四次挥手

为什么是四次挥手？

答：这是由于TCP的半关闭（half-close）造成的。TCP连接是全双工，因此每个方向都需要单独进行关闭。

- 收到一个FIN只意味着在这一方向上没有数据流动。
- 和SYN一样，一个FIN将占用一个序号。

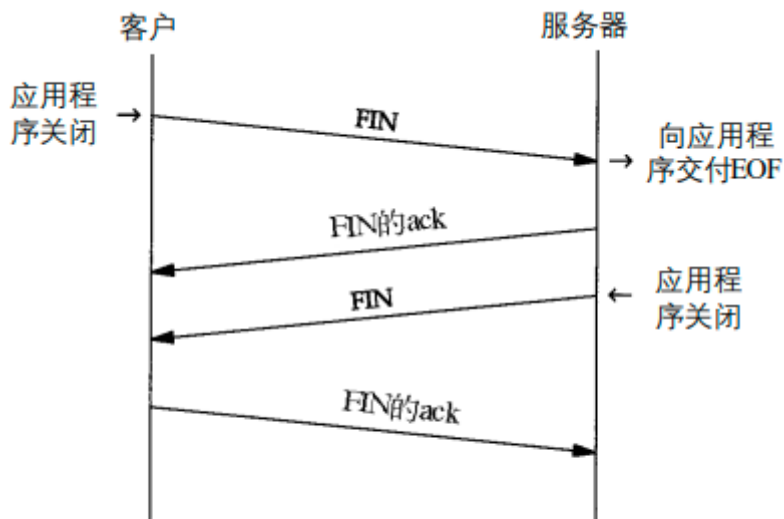


图18-4 连接终止期间报文段的正常交换

## 连接建立的超时

大多数伯克利系统将建立一个新连接的最长时间限制为75秒。在这个75秒的过程中，将会多次发送SYN，试图建立连接；

第2个SYN与第1个SYN间隔5.5s——6.0s

第3个SYN与第2个SYN间隔24.00s（精确）

具体原因下图

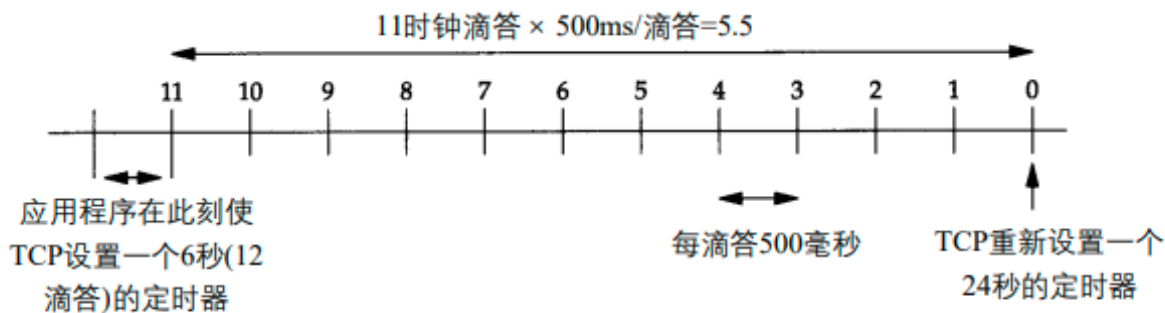


图18-7 TCP的500 ms定时器

## 最大报文段长度（MSS）

TCP在建立连接时，收发双方根据 MTU 计算出各自的 MSS，通过三次握手把自己的MSS告知对方，互相确认彼此的MSS大小，取较小的MSS值作为双方在TCP层分段的最大Payload

如何根据MTU计算MSS

MTU数值 = 1500 = IP头 (20) + TCP头 (20) + Data (假设没有VLAN Tag)

TCP的有效数据Data的最大值就是1500-20-20=1460，这就是MSS的值



- 当建立一个连接时，连接的双方都要告知各自的MSS。
- MSS选项只能出现在SYN报文段。
- MSS不包含TCP及IP的报头
- 发送方与接受方的MSS不一定相等。
- 如果一方没有接收到另一方的MSS值，则就默认536字节
- 注意，没有携带option的TCP报文，最大可以支持1460字节的Payload长度。一旦携带option，由于option需要占用空间，留给payload的空间将会相应减少，具体减少的空间等于option占用的空间。

在三次握手环节，双方确定最小MSS

Length	Info
66 65397 → 443	[SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
66 443 → 65397	[SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1360 WS=32 SACK_PERM=1

## TCP半关闭

TCP提供了连接的一端在结束它的发送后还能接收来自另一端数据的能力。

如果应用程序不调用close而调用shutdown，且第2个参数值为1，则插口的API支持半关闭。然而，大多数的应用程序通过调用close终止两个方向的连接

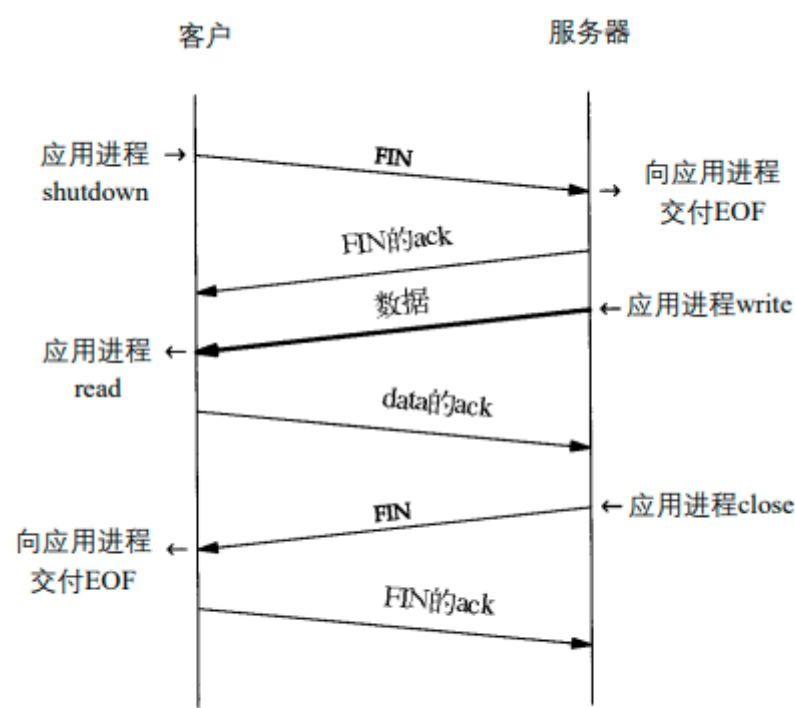


图18-10 TCP半关闭的例子

## TCP的状态变迁图

使用图18-12的状态图来跟踪图18-13的状态变化过程，以便明白每个状态的变化。

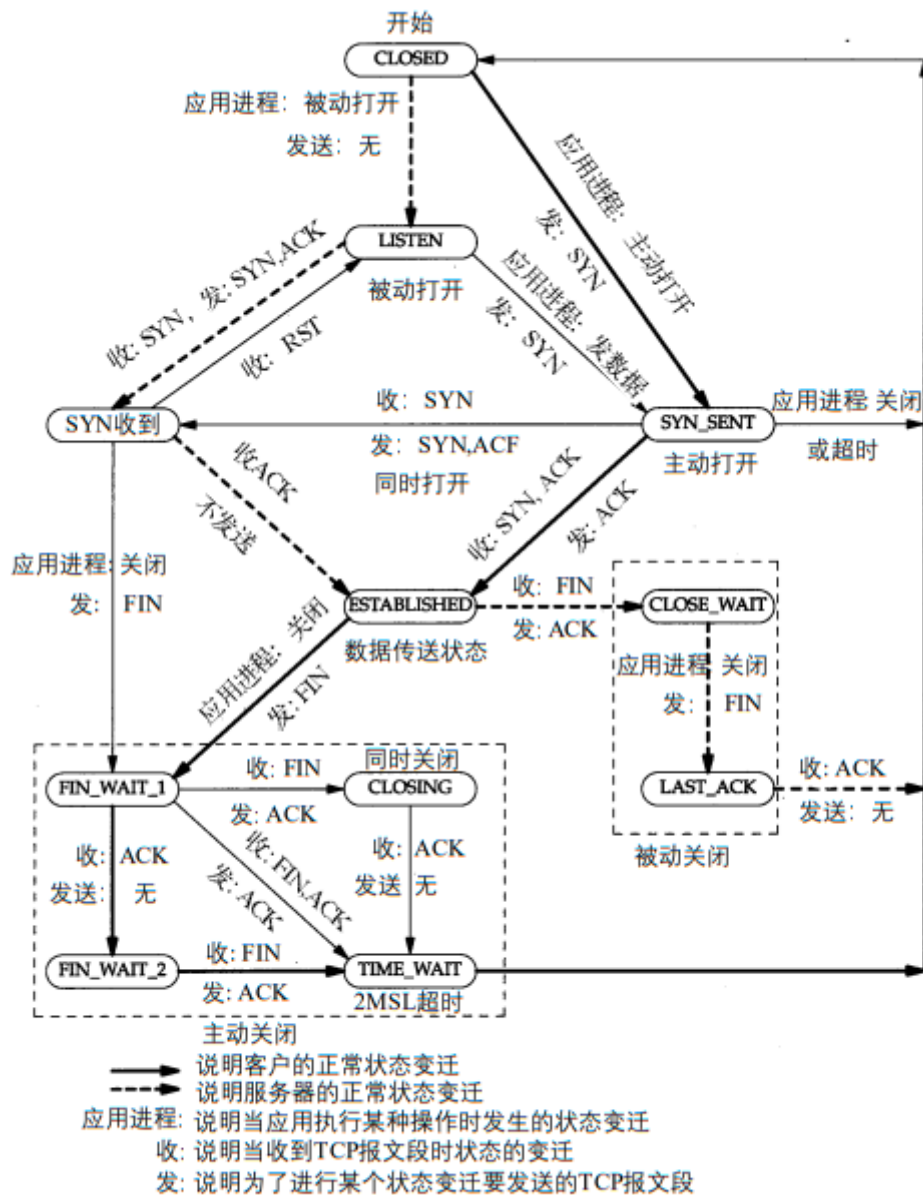


图18-12 TCP的状态变迁图

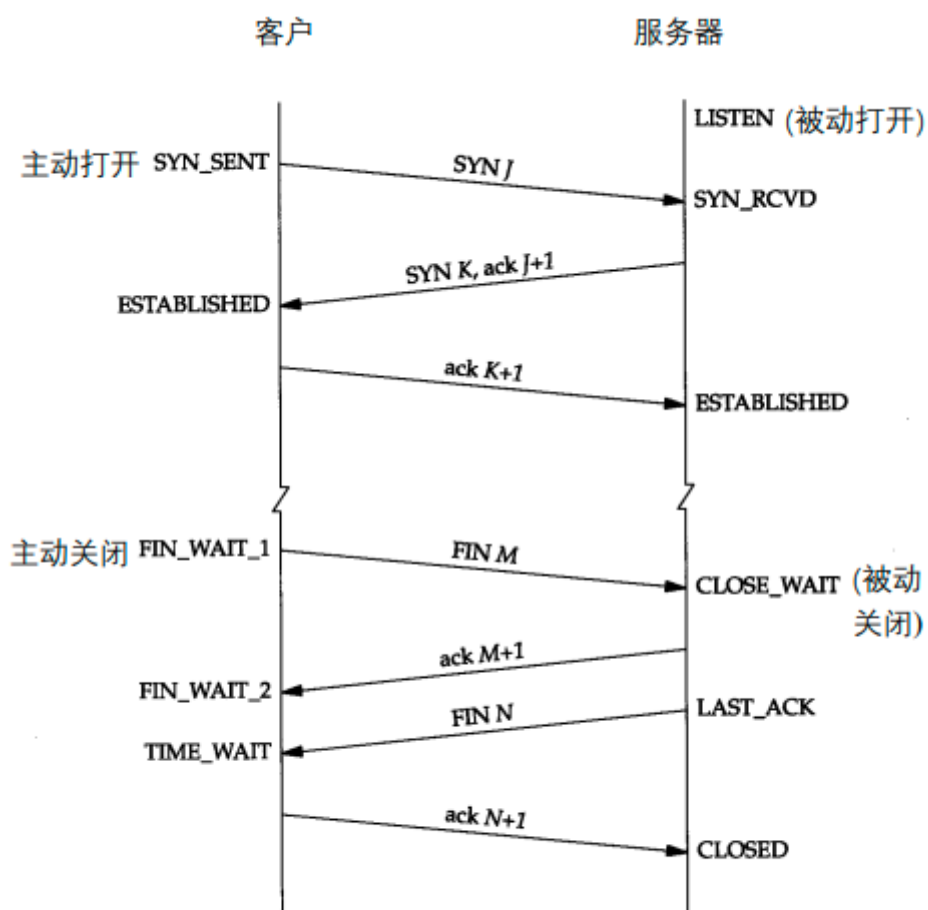


图18-13 TCP正常连接建立和终止所对应的状态

## 经受时延的确认

当有ACK需要发出的时候，ACK的两种发出场景：

- 一、当有数据需要发送的时候，会携带着ACK一块发出。
- 二、当系统的200ms的定时器溢出的时候，单独发出，这种经过定时器溢出而单独发出的ACK就叫做**经受时延确认的ACK**。

- 这样设计，主要是为了提高性能，也就是最好能等到TCP要发送数据，携带着ACK一块发出。
- 如果在200ms溢出时刻等不到数据，就单独发送ACK。
- 这个200ms定时器是一直开着的，也就是说不一定这个定时器什么时候溢出，假设：在你有数据要发出的前一ms溢出了，且刚好有ACK需要发出，就比较尴尬，TCP会单独发出一个ACK，然后1ms后在单独发出数据。

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。

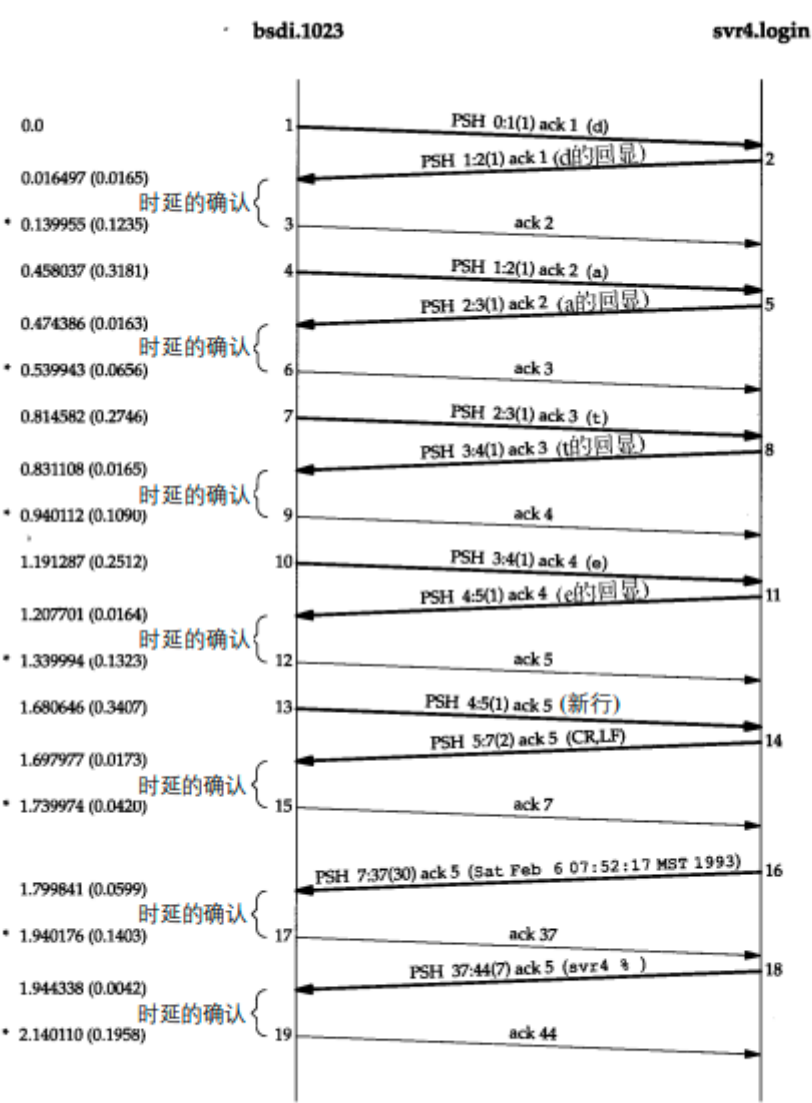


图19-3 在rlogin连接上键入date命令时的数据流时间系列

### Nagle算法

在较慢的广域网环境中，通常使用Nagle算法来减少这些小报文段的数目。

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（或者远程桌面的鼠标）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。

## 滑动窗口

发送端根据接收端通告的窗口大小来控制自己发送的数据量。

- 接收端的接收窗口取决于应用程序读取缓存数据的速度。
- 如果接收端通告的窗口大小为0，则发送的滑动窗口（发送窗口）为零窗口，此时发送端不能发送任何数据。

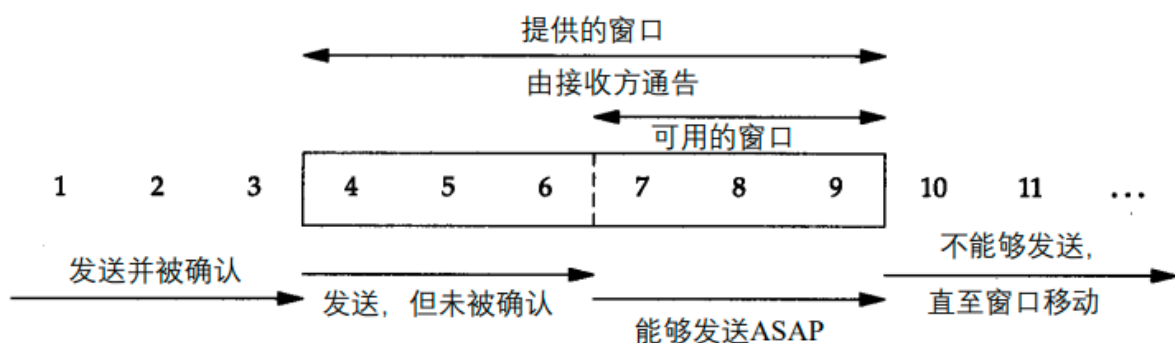


图20-4 TCP滑动窗口的可视化表示

## TCP坚持定时器（持续定时器）

当发送窗口为0后，这时就是零窗口状态，不能发送数据，只能等待接收端通告大于0的窗口，如果接收端通告大于0的窗口报文中途丢失，这时双发就处于假死状态。

因此，当窗口为0时，客户端会启动持续定时器，在定时器触发时主动发送零窗口探测报文段，服务端收到后，会同步发送响应报文更新窗口大小。

但是零窗口探测报文段也有可能会丢失，所以该报文也有一个重传计时器。

## 慢启动和拥塞避免

拥塞避免算法和慢启动算法需要对每个连接维持两个变量：

拥塞窗口 (*cwnd*)

慢启动门限 (*ssthresh*)

首先说慢启动：

- 当建立一个TCP连接后，拥塞窗口初始化为1个报文段。
- 每收到一个ACK，*cwnd*增加一个报文段（*cwnd*以字节为单位，但是慢启动以报文段大小为单位进行增加）

举例：

1. 发送方开始发送一个报文段，然后等待ACK。
2. 当收到该ACK时，拥塞窗口从1增加为2，即可发送两个报文段。
3. 当收到这两个报文段的ACK时，
4. 拥塞窗口就增加为4。

总结：这是一种指数增加关系。所以*cwnd*增长的很快，迟早会把整个网络带宽塞满，一般会有两种情况出现就会阻止*cwnd*的指数级增长。

- 超时：*ssthresh*重新设置为1，通过将*ssthresh*设为*cwnd*的一半，随后继续进行慢启动。
- 当*cwnd* ≥ *ssthresh*：就要停止慢启动而改用拥塞避免算法来控制*cwnd*增大的速度。

图中以报文段为单位显示*cwnd*和*ssthresh*，但他们实际上都是以字节为单位进行维护的。

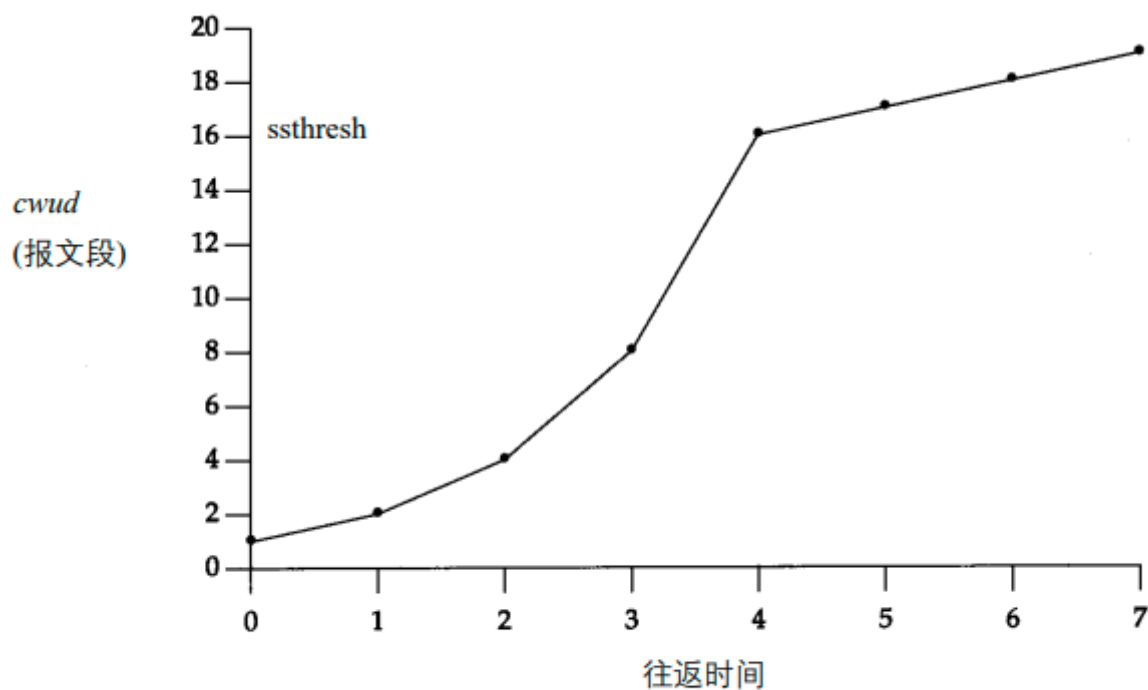


图21-8 慢启动和拥塞避免的可视化描述

### 拥塞避免：

当启用拥塞避免算法时，每收到一个ACK将cwnd增加 $1/cwnd$ 。

## 快速重传和快恢复

在使用慢开始和拥塞避免算法时，如果出现数据报超时，就认为网络拥塞了。

但是，当网络很流畅时，接收端收到了0-99, 100-199, 300-399, 400-499, 500-599；这并不代表网络拥塞了，而是200-299数据报可能因为某种原因丢失了，这时如果发送方认为网络拥塞造成数据包超时未确认，误用拥塞算法而将cwnd减小为1，这无疑降低了传输效率。于是使用快速重传算法来解决。

### 快速重传：

原则是发送方在发现个别报文丢失时，就应该快速重传，而不是等到超时了再重传。该算法具体实现：

假设场景：

接收端收到了0-99, 100-199, 300-399, 400-499, 500-599；  
其中200-299数据报丢失了。

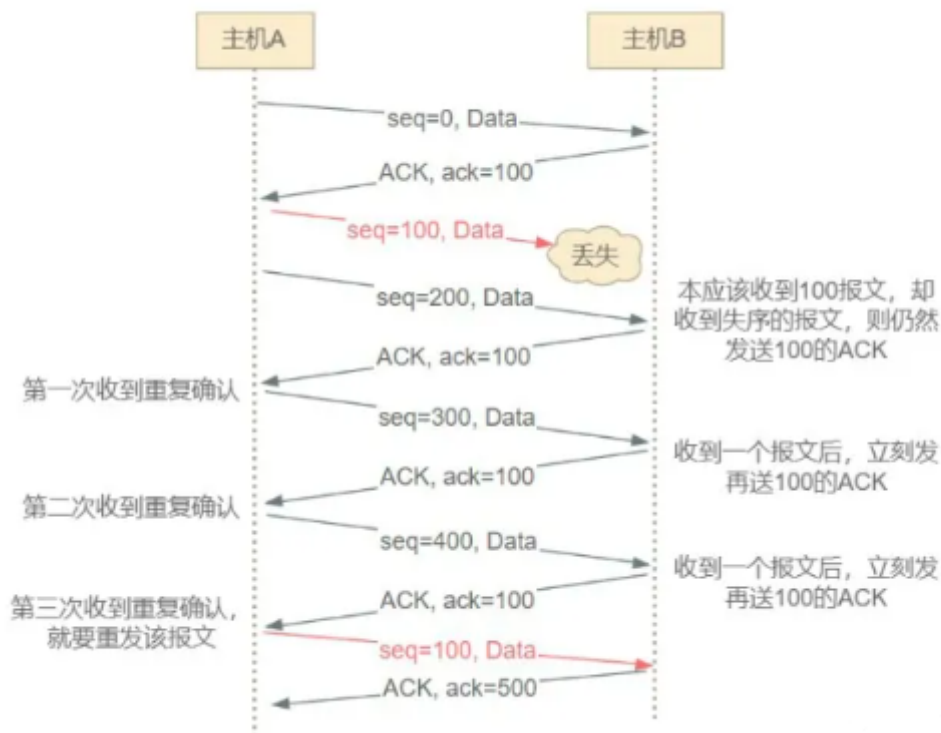
接收端操作：

1. 接收端收到300-399时，会发送重复确认ACK（确认序号200），300数据报存到缓存。
2. 收到400-499时，继续发送重复确认ACK（确认序号200），400数据报存到缓存。
3. 收到500-599时，同上操作。
4. 当接收到丢失的报文，就将已收到的失序报文一起累计确认。



发送端操作：

发送端如果收到3个或3个以上的重复确认ACK，就不会等超时再重发，而是立马将这个报文重新发出。



快恢复：

当发送方连续收到三个重复的ACK报文时，就知道只是个别报文端丢失了，于是不启用慢启动算法，而改用 快恢复算法。

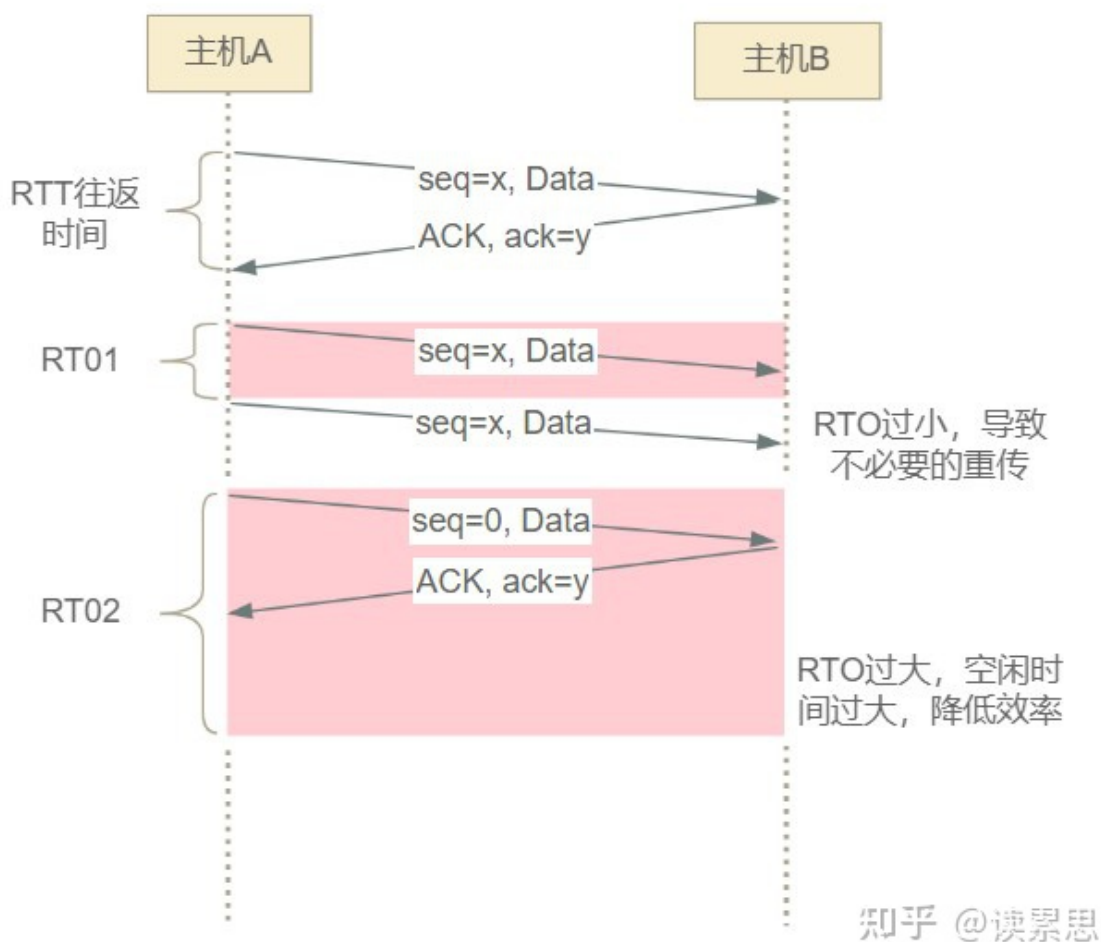
快恢复算法一般有两种实现方案：

- 发送方将满启动门限sssthresh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sssthresh+3。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。

## 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：

$RTTs_1 = RTT_1$  (第一个报文段的RTT作为RTTs)

$RTTs = (1 - \alpha) \times \text{旧的RTTs} + \alpha \times \text{新的RTT样本}$  (RFC6298标准推荐 $\alpha$ 值为0.125)

$RTO = RTTs + 4 \times RTTd$

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：

$RTTd_1 = RTT_1 \div 2$  (第一个报文段的RTTd值)

$\text{新的RTTd} = (1 - \beta) \times \text{旧的RTTd} + \beta \times (RTTs - \text{新的RTT样本})$  (RFC6298标准推荐 $\beta$ 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。**

## 补充

### 校验和

1. IP校验和只校验20字节的IP报头（强制）
2. ICMP校验和覆盖整个报文(ICMP报头+ICMP数据)（强制）
3. IGMP校验和包括IGMP报文（强制）
4. UDP和TCP校验和不仅覆盖整个报文，而且还有12字节的IP伪首部（强制）