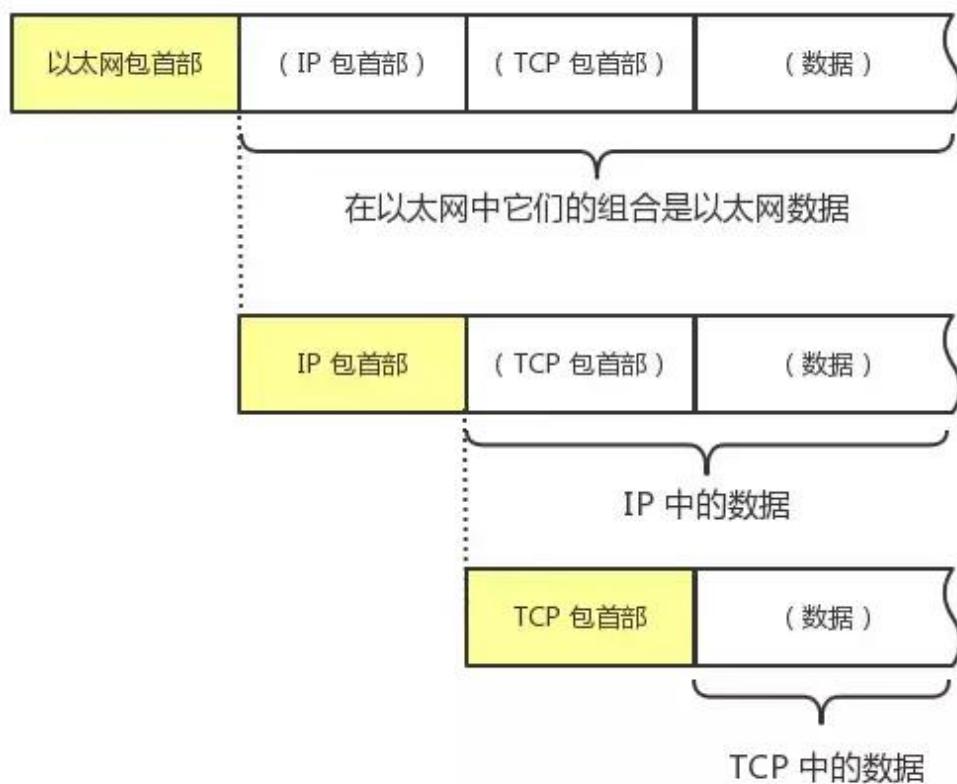


网络模型分层

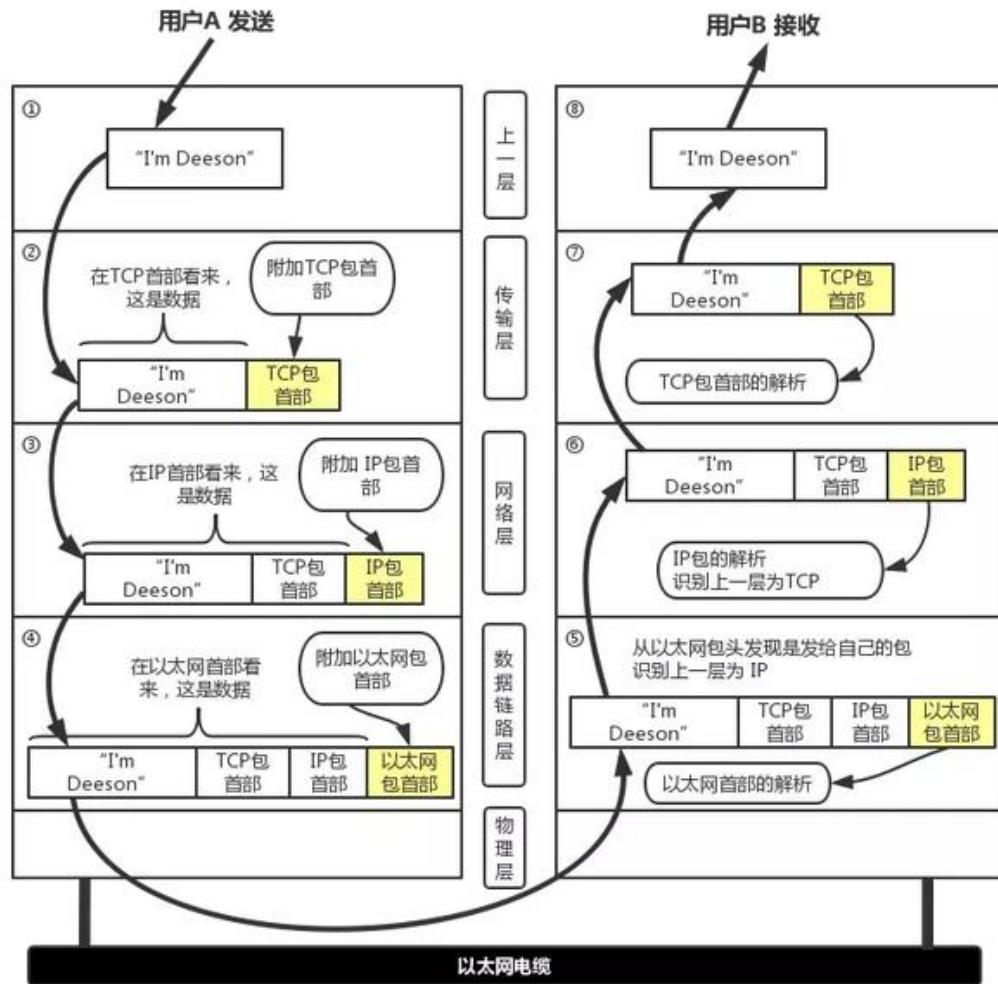
OSI七层模型	TCP/IP概念层模型	功能	TCP/IP协议族
应用层	应用层	文件传输,电子邮件,文件服务,虚拟终端	TFTP, HTTP, SNMP, FTP, SMTP, DNS, Telnet
表示层		数据格式化,代码转换,数据加密	没有协议
会话层		解除或建立与别的接点的联系	没有协议
传输层	传输层	提供端对端的接口	TCP, UDP
网络层	网络层	为数据包选择路由	IP, ICMP, RIP, OSPF, BGP, ICMP
数据链路层	链路层	传输有地址的帧以及错误检测功能	SLIP, CSLIP, PPP, ARP, RARP, MTU
物理层		以二进制数据形式在物理媒体上传输数据	ISO2110, IEEE802, IEEE802.2

计算机网络体系结构分层



数据包首部

下图以用户 a 向用户 b 发送邮件为例子：



数据处理流程

Ethernet -> Frame

Ip -> packet

Tcp -> segment

Mss : max segment size

Msl : max segment lifETIME

Tcpdump + wireshark 进行抓包+分析

基本用法:

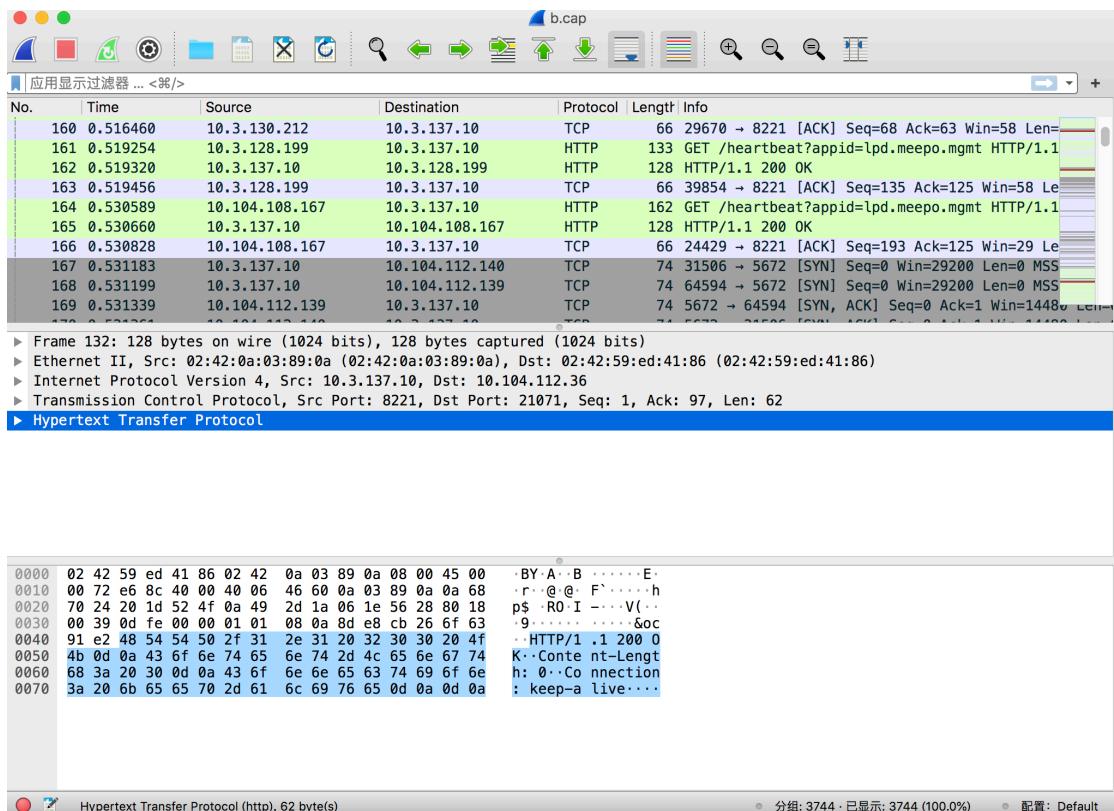
tcpdump -i eth0

```
tcpdump -A -i eth0 -e -s 0 -c 200 dst host alta1-web-httppizza-7.vm.elenet.me or src host alta1-web-httppizza-7.vm.elenet.me and port not 8221
```

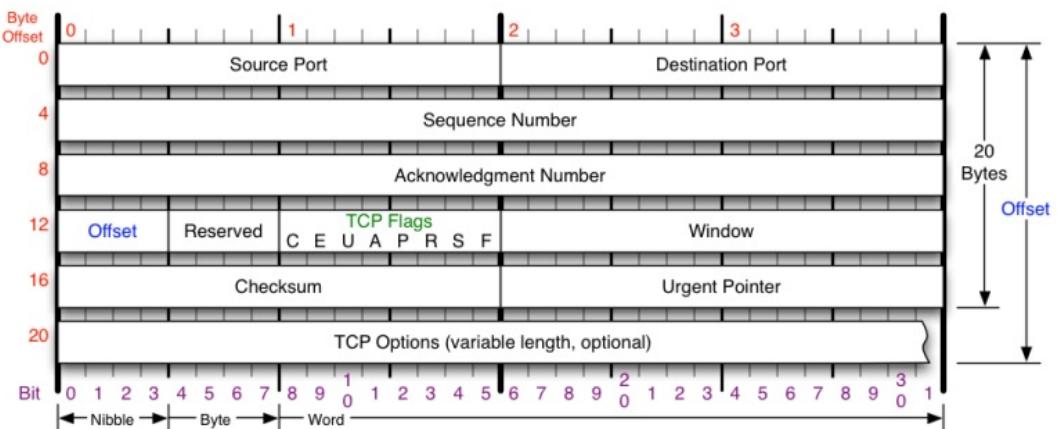
```
13:30:59.518231 IP altb-lpd-meepo-mgmt-659-1919223-5411945.c.elenet.me.25172 > 10.3.129.11.zabbix-trapper: Flags (.), ack 63, win 58, options [nop, nop, TS val 600345692 ecr 332735282], length 0
E..4..@.0...
...
...bT'C.yL....2...:B....
#...#.#2
13:30:59.521823 IP 10.3.134.213.49166 > altb-lpd-meepo-mgmt-659-1919223-5411945.c.elenet.me.8221: Flags (P.), seq 192:288, ack 125, win 58, options [nop, nop, TS val 3850078986 ecr 3876909098], length 96
E.....@->.i_
...
...
...i-...x:.....
.{.
...*GET /heartbeat?appid=lpd.meepo.mgmt HTTP/1.1
Host: 10.3.137.10:8221
Connection: Keep-Alive

13:30:59.521914 IP altb-lpd-meepo-mgmt-659-1919223-5411945.c.elenet.me.8221 > 10.3.134.213.49166: Flags (P.), seq 125:187, ack 288, win 57, options [nop, nop, TS val 3876909598 ecr 3850078986], length 62
E..r..@.0...].
...
... ..~..x.....9$J.....
....{
HTTP/1.1 200 OK
Content-Length: 0
Connection: keep-alive

13:30:59.522054 IP 10.3.134.213.49166 > altb-lpd-meepo-mgmt-659-1919223-5411945.c.elenet.me.8221: Flags (.), ack 187, win 58, options [nop, nop, TS val 3850078987 ecr 3876909598], length 0
E..4..@.->.i.
...
...
... ..-.-:j.....
. .....
13:30:59.522612 IP altbl-isochrone-ops-2.vm.elenet.me.56317 > altb-lpd-meepo-mgmt-659-1919223-5411945.c.elenet.me.8221: Flags (P.), seq 192:288, ack 125, win 29, options [nop, nop, TS val 1814460291 ecr 3507392504], length 96
```



TCP 头格式



TCP 头格式

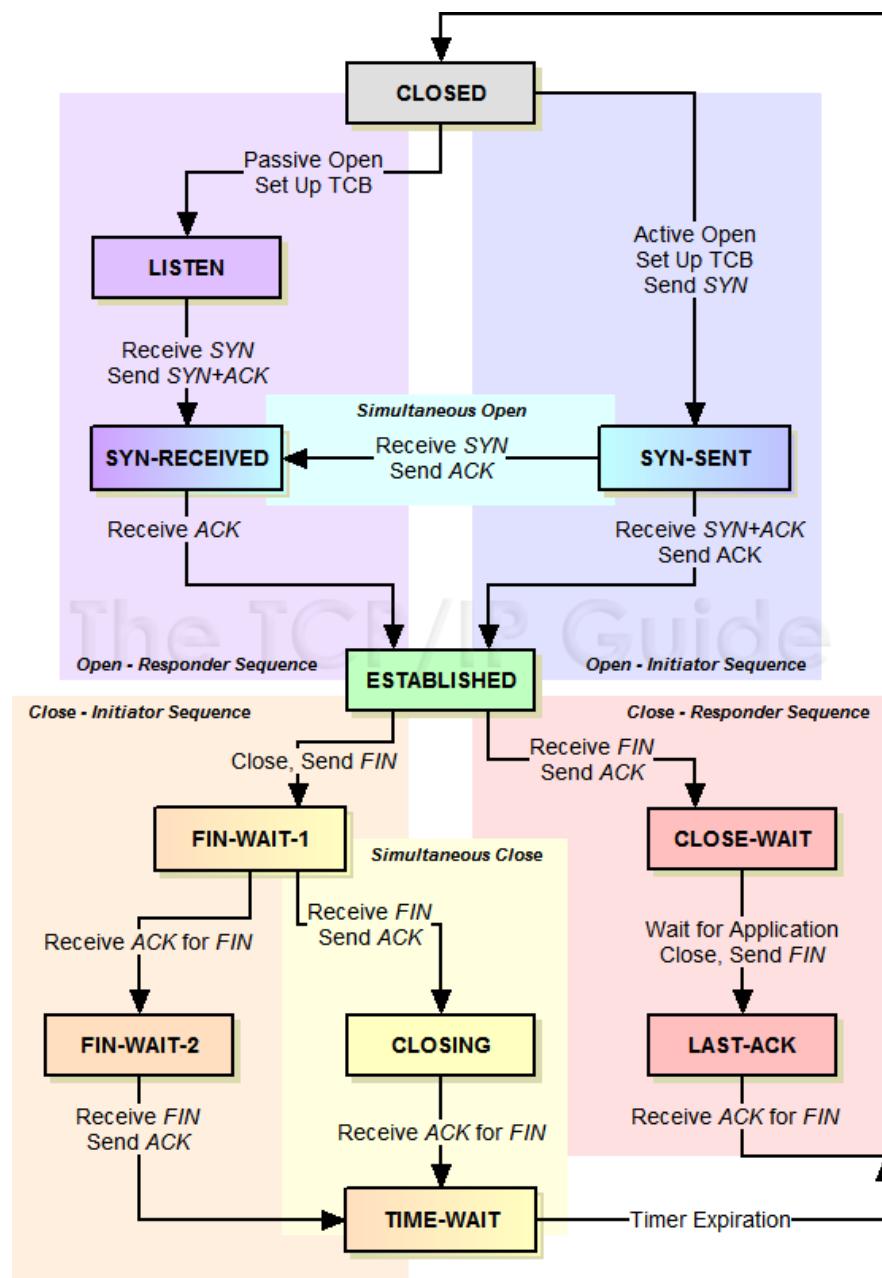
- TCP 的包是没有 IP 地址的，那是 IP 层上的事。但是有源端口和目标端口。
- 一个 TCP 连接需要四个元组来表示是同一个连接 (`src_ip`, `src_port`, `dst_ip`, `dst_port`) 准确说是五元组，还有一个是协议。
 - **Sequence Number** 是包的序号，用来解决网络包乱序 (reordering) 问题。
 - **Acknowledgement Number** 就是 ACK——用于确认收到，用来解决不丢包的问题。
 - **Window** 又叫 **Advertised-Window**，也就是著名的滑动窗口 (Sliding Window)，用于解决流控的。

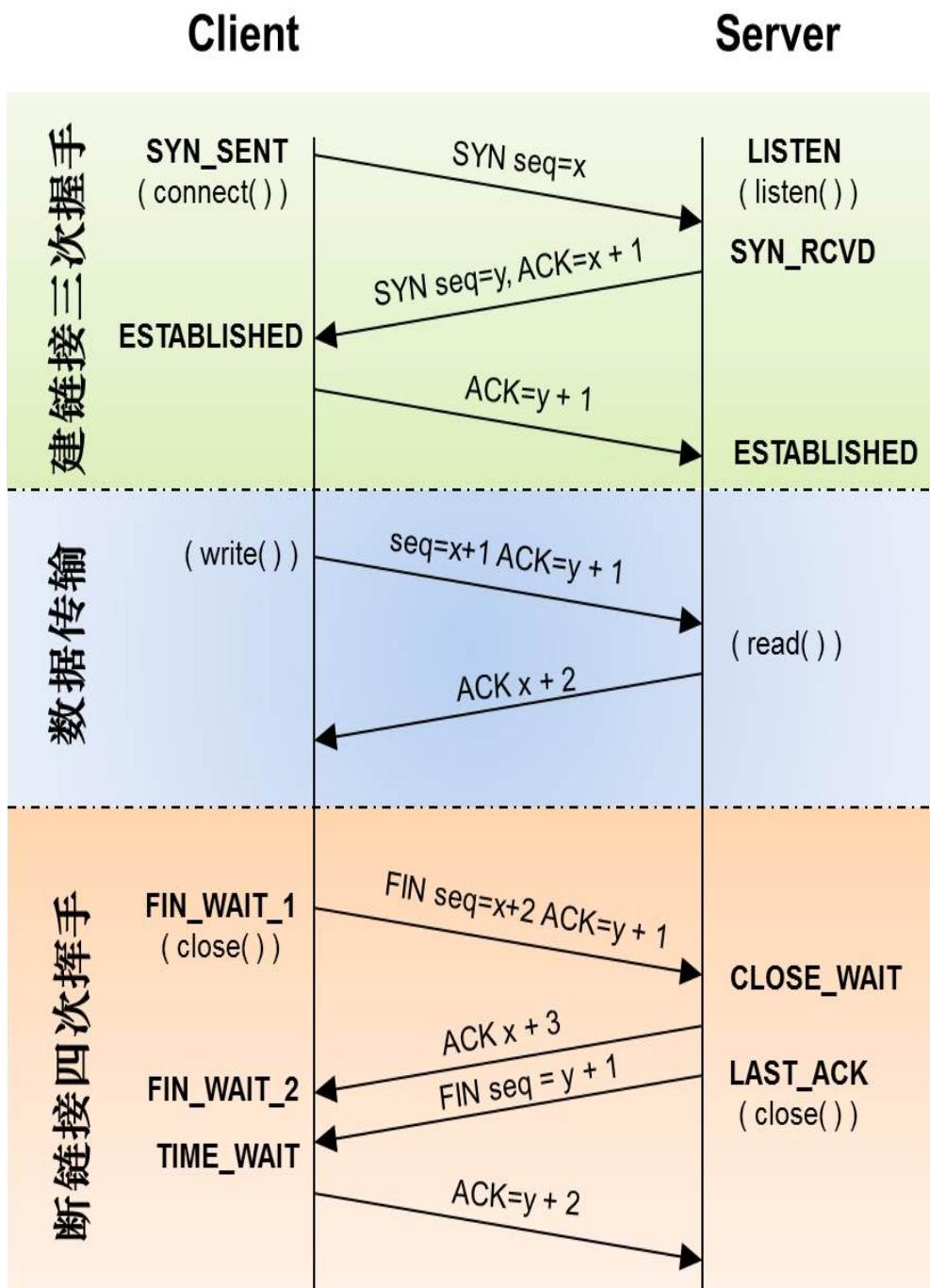
- **TCP Flag**，也就是包的类型，主要是用于操控 **TCP** 的状态机的。

TCP 的状态机

其实，网络上的传输是没有连接的，包括 **TCP** 也是一样的。而 **TCP** 所谓的“连接”，其实只不过是在通讯的双方维护一个“连接状态”，让它看上去好像有连接一样。所以，**TCP** 的状态变换是非常重要的。下面是：“**TCP 协议的状态机**”和“**TCP 建链接**”、“**TCP 断链接**”、“**传数据**”的对

照图。





- 对于建链接的 3 次握手，主要是要初始化 Sequence Number 的初始值。通信的双方要互相通知对方自己的初始化的 Sequence Number（缩写为 ISN: Initial Sequence Number）——所以叫 SYN，全称 Synchronize Sequence Numbers。也就

上图中的 x 和 y。这个号要作为以后的数据通信的序号，以保证应用层接收到的数据不会因为网络上的传输的问题而乱序（TCP 会用这个序号来拼接数据）。

The screenshot displays two TCP sessions in Wireshark:

- Session 1 (Top):** A connection from 10.3.137.10 (Source Port: 9879) to 10.104.105.139 (Destination Port: 44570). The first packet is a SYN with sequence number 0 and acknowledgement number 0. The second packet is an ACK with sequence number 0 and acknowledgement number 1.
- Session 2 (Bottom):** A connection from 10.104.105.139 (Source Port: 9879) to 10.3.137.10 (Destination Port: 44570). The first packet is a SYN-ACK with sequence number 0 and acknowledgement number 1. The second packet is an ACK with sequence number 1 and acknowledgement number 1.

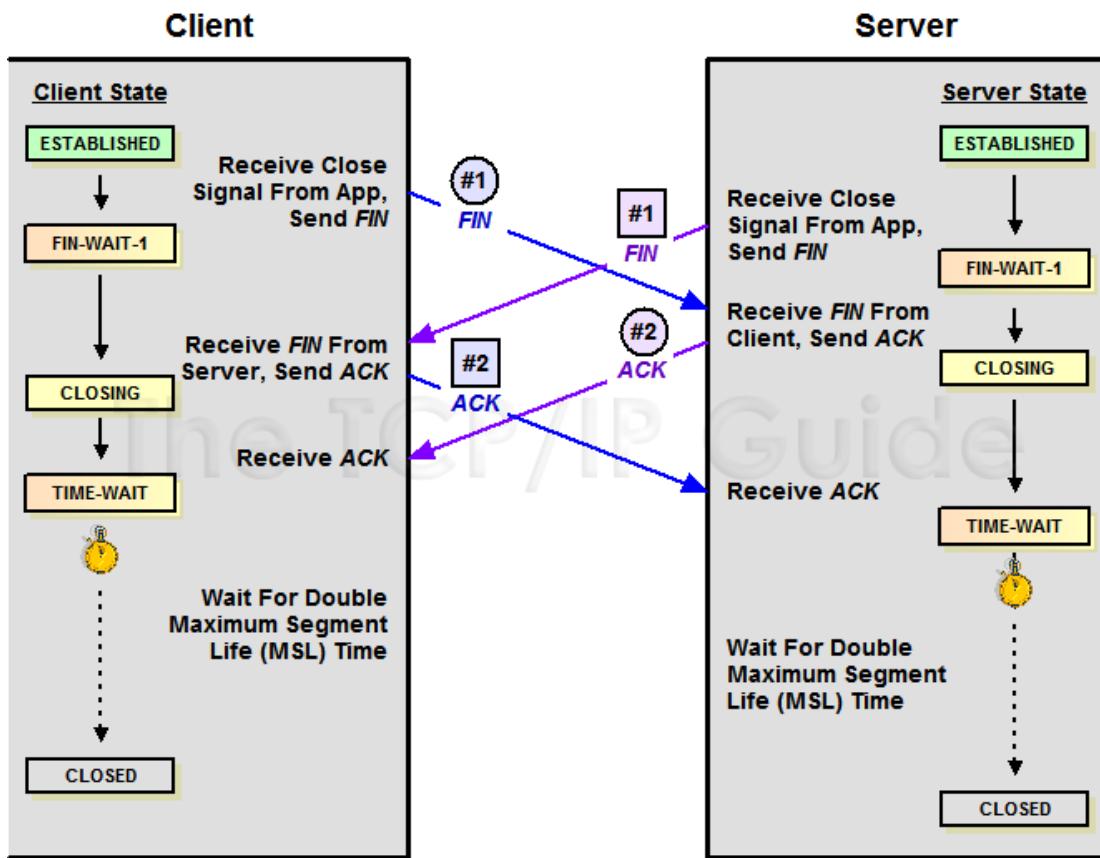
Both sessions show sequence numbers starting at 0 and acknowledgement numbers starting at 1, indicating the initial state of the connection.

应用显示过滤器 ... <%>/>

No.	Time	Source	Destination	Protocol	Length	Info
277	0.768721	10.3.137.10	10.104.105.139	TCP	66	39448 → 9879 [SYN] Seq=0 Ack=1 Win=29696 Len=60
278	0.768738	10.3.137.10	10.104.105.139	TCP	74	9879 → 50042 [SYN, ACK] Seq=1 Ack=0 Win=2896
279	0.768752	10.104.105.134	10.3.137.10	TCP	74	9879 → 50042 [SYN, ACK] Seq=1 Ack=1 Win=29696
280	0.768759	10.3.137.10	10.104.105.135	TCP	122	23266 → 9879 [PSH, ACK] Seq=1 Ack=1 Win=29696
281	0.768761	10.3.137.10	10.104.105.134	TCP	66	50042 → 9879 [ACK] Seq=1 Ack=1 Win=29696
282	0.768771	10.3.137.10	10.104.105.138	TCP	122	47360 → 9879 [PSH, ACK] Seq=1 Ack=1 Win=29696
283	0.768803	10.104.113.41	10.3.137.10	TCP	74	9879 → 44570 [SYN, ACK] Seq=0 Ack=1 Win=2896
284	0.768812	10.104.105.131	10.3.137.10	TCP	74	9879 → 32050 [SYN, ACK] Seq=0 Ack=1 Win=2896
285	0.768817	10.104.105.130	10.3.137.10	TCP	74	9879 → 39104 [SYN, ACK] Seq=0 Ack=1 Win=2896
286	0.768831	10.3.137.10	10.104.113.41	TCP	66	44570 → 9879 [ACK] Seq=1 Ack=1 Win=29696
287	0.768839	10.3.137.10	10.104.105.131	TCP	66	23050 → 9879 [ACK] Seq=1 Ack=1 Win=29696
288	0.768842	10.104.105.137	10.3.137.10	TCP	66	9879 → 38162 [ACK] Seq=1 Ack=57 Win=29184
289	0.768845	10.3.137.10	10.104.105.130	TCP	66	39104 → 9879 [ACK] Seq=1 Ack=1 Win=29696
290	0.768852	10.104.105.132	10.3.137.10	TCP	74	9879 → 36924 [SYN, ACK] Seq=0 Ack=1 Win=2896
291	0.768855	10.104.105.136	10.3.137.10	TCP	74	9879 → 43926 [SYN, ACK] Seq=0 Ack=1 Win=2896
292	0.768860	10.3.137.10	10.104.105.132	TCP	66	36924 → 9879 [ACK] Seq=1 Ack=1 Win=29696
293	0.768866	10.3.137.10	10.104.105.136	TCP	66	43926 → 9879 [ACK] Seq=1 Ack=1 Win=29696

Sequence number (raw): 1948448367
[Next sequence number: 1 (relative sequence number)]
Acknowledgment number: 1 (relative ack number)
Acknowledgment number (raw): 3153784249
1000 = Header Length: 32 bytes (8)
Flags: 0x010 (ACK)
Window size value: 50
[Calculated window size: 29696]
[Window size scaling factor: 512]
Checksum: 0x0ec5 [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0

- 对于 4 次挥手，因为 TCP 是全双工的，所以，发送方和接收方都需要 Fin 和 Ack。只不过，有一方是被动的，所以看上去就成了所谓的 4 次挥手。如果两边同时断连接，那就会进入到 CLOSING 状态，然后到达 TIME_WAIT 状态。下图是双方同时断连接的示意图：



两端同时断连接

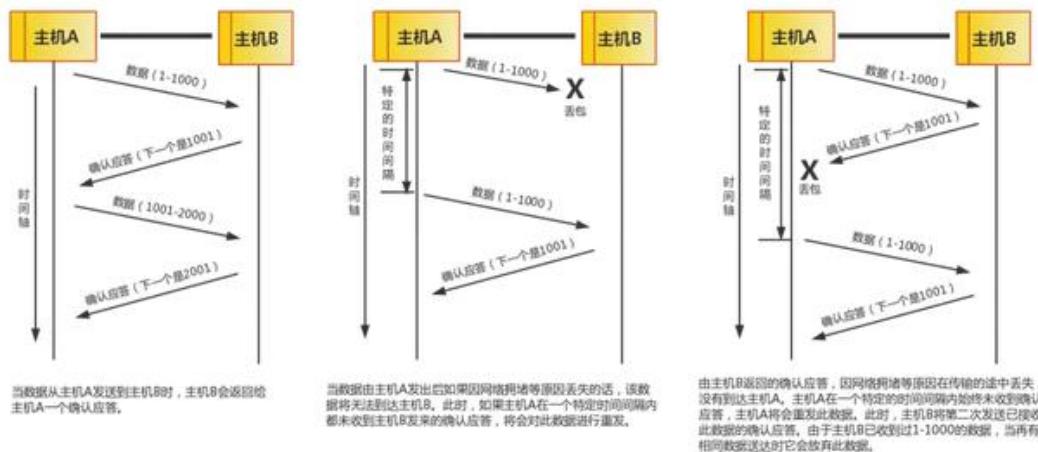
- 对于数据传输,
 - 1、在 TCP 中，当发送端的数据到达接收主机时，接收端主机会返回一个已收到消息的通知。这个消息叫做确认应答（ACK）。当发送端将数据发出之后会等待对端的确认应答。如果有确认应答，说明数据已经成功到达对端。**反之，则数据丢失的可能性很大。**
 - 2、在一定时间内没有等待到确认应答，发送端就可以认为数据已经丢失，并进行重发。由此，即使产生了丢包，仍然能够保证数据能够到达对端，实现可靠传输。

3、未收到确认应答并不意味着数据一定丢失。也有可能是数据对方已经收到，只是返回的确认应答在途中丢失。这种情况也会导致发送端误以为数据没有到达目的地而重发数据。

4、此外，也有可能因为一些其他原因导致确认应答延迟到达，在源主机重发数据以后才到达的情况也屡见不鲜。此时，源主机只要按照机制重发数据即可。

5、对于目标主机来说，反复收到相同的数据是不可取的。为了对上层应用提供可靠的传输，目标主机必须放弃重复的数据包。为此我们引入了序列号。

6、**序列号是按照顺序给发送数据的每一个字节（8位字节）都标上号码的编号。接收端查询接收数据 TCP 首部中的序列号和数据的长度，将自己下一步应该接收的序列号作为确认应答返送回去。通过序列号和确认应答号，TCP 能够识别是否已经接收数据，又能够判断是否需要接收，从而实现可靠传输。**



TCP 重传机制

TCP 要保证所有的数据包都可以到达，所以，必需要有重传机制。

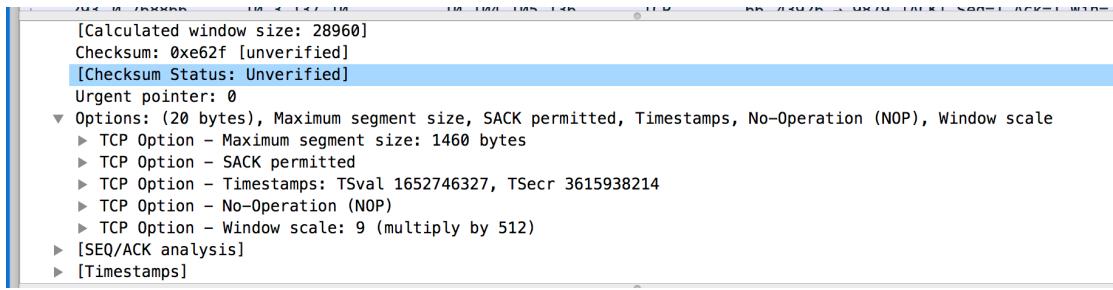
- **重发超时是指在重发数据之前，等待确认应答到来的那个特定时间间隔。**

如果超过这个时间仍未收到确认应答，发送端将进行数据重发。最理想的是，找到一个最小时间，它能保证“确认应答一定能在这个时间内返回”。
- TCP 要求不论处在何种网络环境下都要提供高性能通信，并且无论网络拥堵情况发生何种变化，都必须保持这一特性。为此，它在每次发包时都会计算往返时间及其偏差。将这个往返时间和偏差时间相加，重发超时的时间就是比这个总和要稍大一点的值。
- 在 BSD 的 Unix 以及 Windows 系统中，超时都以 0.5 秒为单位进行控制，因此重发超时都是 0.5 秒的整数倍。不过，最初其重发超时的默认值一般设置为 6 秒左右。
- 数据被重发之后若还是收不到确认应答，则进行再次发送。此时，等待确认应答的时间将会以 2 倍、4 倍的指数函数延长。
- 此外，**数据也不会被反复地重发。达到一定重发次数之后，如果仍没有任何确认应答返回，就会判断为网络或对端主机发生了异常，强制关闭连接。并且通知应用通信异常强行终止。**

以段为单位发送数据

- 在建立 TCP 连接的同时，也可以确定发送数据包的单位，我们也可以称其为“最大消息长度”（MSS）。最理想的情况是，最大消息长度正好是 IP 中不会被分片处理的最大数据长度。
- TCP 在传送大量数据时，是以 MSS 的大小将数据进行分割发送。进行重发时也是以 MSS 为单位。

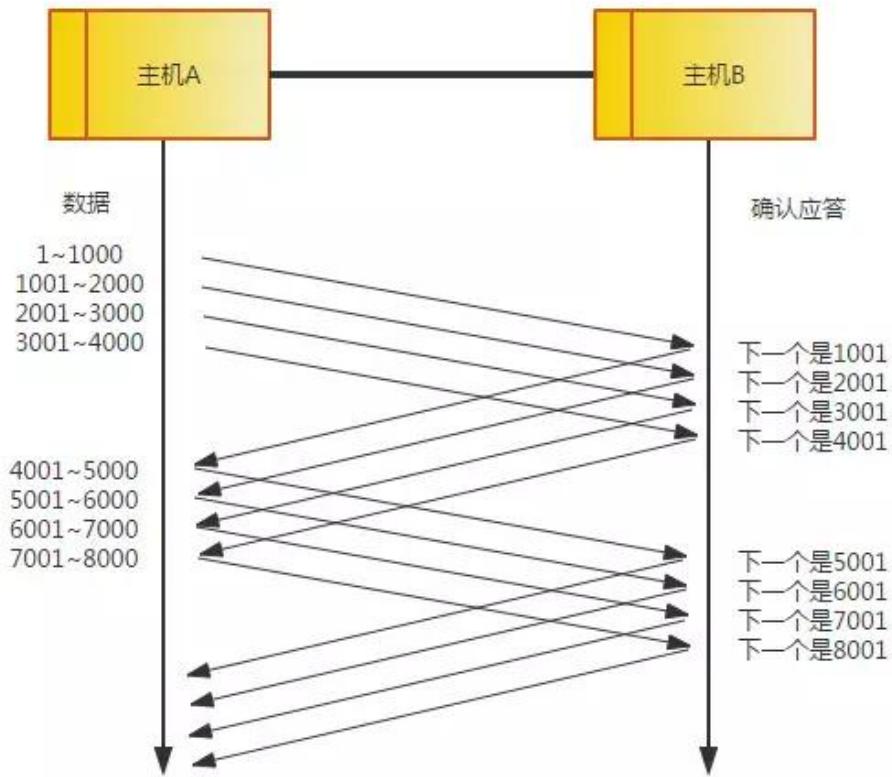
- MSS 在三次握手的时候，在两端主机之间被计算得出。两端的主机在发出建立连接的请求时，会在 TCP 首部中写入 MSS 选项，告诉对方自己的接口能够适应的 MSS 的大小。然后会在两者之间选择一个较小的值投入使用。



```
[Calculated window size: 28960]
Checksum: 0xe62f [unverified]
[Checksum Status: Unverified]
Urgent pointer: 0
▼ Options: (20 bytes), Maximum segment size, SACK permitted, Timestamps, No-Operation (NOP), Window scale
▶ TCP Option - Maximum segment size: 1460 bytes
▶ TCP Option - SACK permitted
▶ TCP Option - Timestamps: TSval 1652746327, TSecr 3615938214
▶ TCP Option - No-Operation (NOP)
▶ TCP Option - Window scale: 9 (multiply by 512)
▶ [SEQ/ACK analysis]
▶ [Timestamps]
```

利用窗口控制提高速度

- TCP 以 1 个段为单位，每发送一个段进行一次确认应答的处理。这样的传输方式有一个缺点，就是包的往返时间越长通信性能就越低。
- 为解决这个问题，TCP 引入了窗口这个概念。确认应答不再是以每个分段，而是以更大的单位进行确认，转发时间将会上被大幅地缩短。也就是说，发送端主机，在发送了一个段以后不必一直等待确认应答，而是继续发送。如下图所示：



根据窗口为4000字节时返回的确认应答，下一步就发送比这个值还要大4000个序列号为止的数据。这跟前面每个段接收确认应答以后再发送另一个新段的情况相比，即使往返时间变长也不会影响网络的吞吐量。

- 窗口大小就是指无需等待确认应答而可以继续发送数据的值。上图中窗口大小为4个段。这个机制实现了使用大量的缓冲区，通过对多个段同时进行确认应答的功能。

接收端给发送端的 **Ack** 确认只会确认最后一个连续的包，比如，发送端发了 1,2,3,4,5 一共五份数据，接收端收到了 1, 2，于是回 **ack 3**，然后收到了 4（注意此时 3 没收到），此时的 TCP 会怎么办？我们要知道，因为正如前面所说的，**SeqNum** 和 **Ack** 是以字节数为单位，所以 **ack** 的时候，不能跳着确认，只能确认最大的连续收到的包，不然，发送端就以为之前的都收到了。

超时重传机制

一种是不回 **ack**, 死等 3, 当发送方发现收不到 3 的 **ack** 超时后, 会重传 3。一旦接收方收到 3 后, 会 **ack** 回 4——意味着 3 和 4 都收到了。

但是, 这种方式会有比较严重的问题, 那就是因为要死等 3, 所以会导致 4 和 5 即便已经收到了, 而发送方也完全不知道发生了什么事, 因为没有收到 **Ack**, 所以, 发送方可能会悲观地认为也丢了, 所以有可能也会导致 4 和 5 的重传。

对此有两种选择:

- 一种是仅重传 **timeout** 的包。也就是第 3 份数据。
- 另一种是重传 **timeout** 后所有的数据, 也就是第 3, 4, 5 这三份数据。

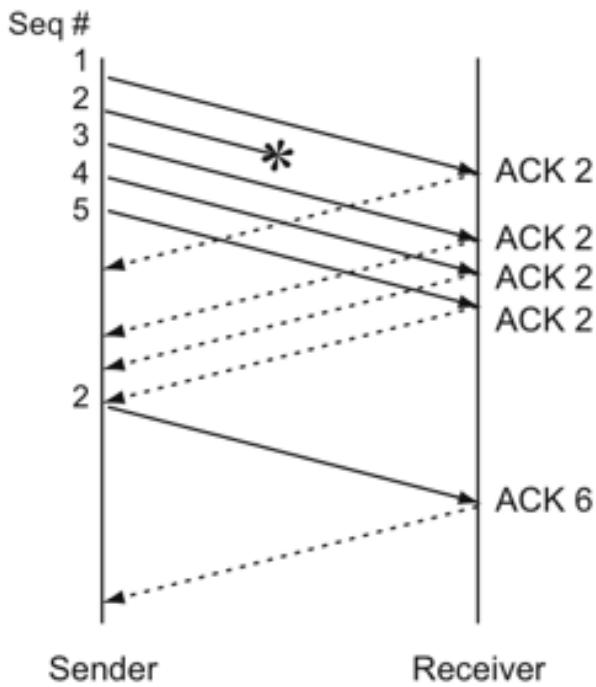
这两种方式有好也有不好。第一种会节省带宽, 但是慢, 第二种会快一点, 但是会浪费带宽, 也可能会有无用功。但总体来说都不好。因为在等 **timeout**, **timeout** 可能会很长

快速重传机制

于是, TCP 引入了一种叫 **Fast Retransmit** 的算法, 不以时间驱动, 而以数据驱动重传。也就是说, 如果, 包没有连续到达, 就 **ack** 最后那个

可能被丢了的包，如果发送方连续收到 3 次相同的 ack，就重传。Fast Retransmit 的好处是不用等 timeout 了再重传。

比如：如果发送方发出了 1, 2, 3, 4, 5 份数据，第一份先到送了，于是就 ack 回 2，结果 2 因为某些原因没收到，3 到达了，于是还是 ack 回 2，后面的 4 和 5 都到了，但是还是 ack 回 2，因为 2 还是没有收到，于是发送端收到了三个 ack=2 的确认，知道了 2 还没有到，于是就马上重传 2。然后，接收端收到了 2，此时因为 3, 4, 5 都收到了，于是 ack 回 6。示意图如下：

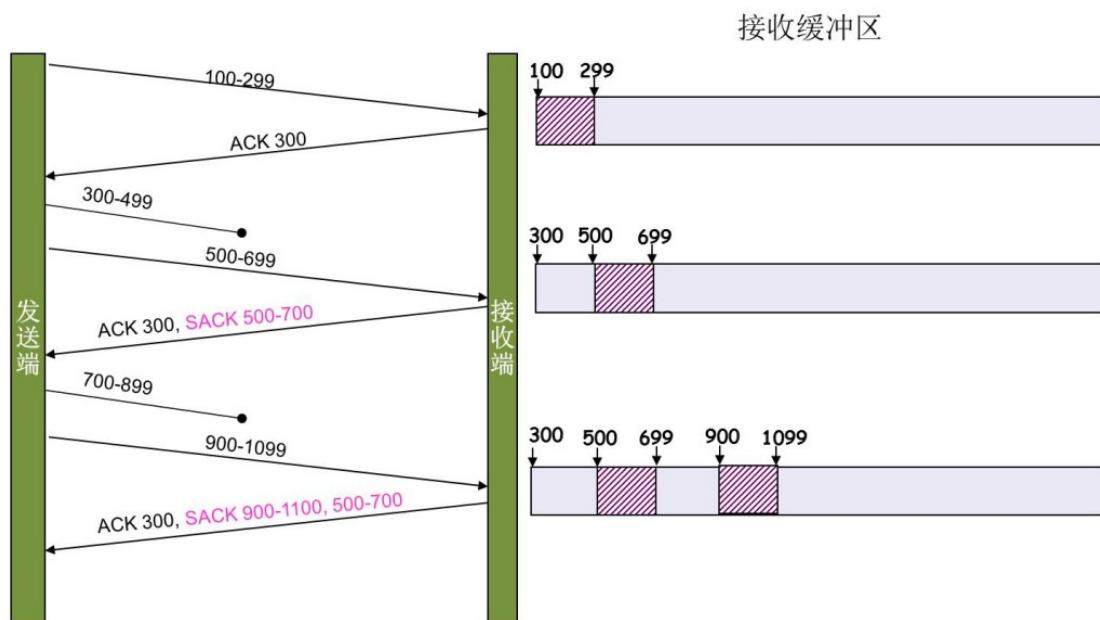


Fast Retransmit 只解决了一个问题，就是 timeout 的问题，它依然面临一个艰难的选择，就是，是重传之前的一个还是重传所有的问题。对于上面的示例来说，是重传#2 呢还是重传#2, #3, #4, #5 呢？因为发送

端并不清楚这连续的 3 个 ack(2)是谁传回来的？也许发送端发了 20 份数据，是#6, #10, #20 传来的呢。这样，发送端很有可能要重传从 2 到 20 的这堆数据（这就是某些 TCP 的实际的实现）。可见，这是一把双刃剑。

SACK 方法

另外一种更好的方式叫：**Selective Acknowledgment (SACK)**（参看 [RFC 2018](#)），这种方式需要在 TCP 头里加一个 SACK 的东西，ACK 还是 Fast Retransmit 的 ACK，SACK 则是汇报收到的数据碎版。参看下图：



这样，在发送端就可以根据回传的 SACK 来知道哪些数据到了，哪些没有到。于是就优化了 Fast Retransmit 的算法。当然，这个协议需要两

边都支持。在 Linux 下，可以通过 **tcp_sack** 参数打开这个功能（Linux 2.4 后默认打开）。

这里还需要注意一个问题——接收方 **Reneging**，所谓 **Reneging** 的意思就是接收方有权把已经报给发送端 **SACK** 里的数据给丢了。这样干是不被鼓励的，因为这个事会把问题复杂化了，但是，接收方这么做可能会有些极端情况，比如要把内存给别的更重要的东西。所以，发送方也不能完全依赖 **SACK**，还是要依赖 **ACK**，并维护 **Time-Out**，如果后续的 **ACK** 没有增长，那么还是要把 **SACK** 的东西重传，另外，接收端这边永远不能把 **SACK** 的包标记为 **Ack**。

注意：**SACK** 会消费发送方的资源，试想，如果一个攻击者给数据发送方发一堆 **SACK** 的选项，这会导致发送方开始要重传甚至遍历已经发出的数据，这会消耗很多发送端的资源。

TCP 的 RTT 算法

从前面的 TCP 重传机制我们知道 **Timeout** 的设置对于重传非常重要。

- 设长了，重发就慢，没有效率，性能差；
- 设短了，会导致可能并没有丢就重发。会增加网络拥塞，导致更多的超时，更多的超时导致更多的重发。

而且，这个超时时间在不同的网络的情况下，没有办法设置一个死的值。只能动态地设置。为了动态地设置，TCP 引入了 RTT——Round Trip Time，也就是一个数据包从发出去到回来的时间。这样发送端就大约知道需要多少的时间，从而可以方便地设置 Timeout——RTO（Retransmission TimeOut），以让我们的重传机制更高效。

经典算法

RFC793 中定义的经典算法是这样的：

- 1) 首先，先采样 RTT，记下最近好几次的 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)

$$\text{SRTT} = (\alpha * \text{SRTT}) + ((1 - \alpha) * \text{RTT})$$

- 3) 开始计算 RTO。公式如下：

$$\text{RTO} = \min [\text{UBOUND}, \max [\text{LBOUND}, (\beta * \text{SRTT})]]$$

其中：

- UBOUND 是最大的 timeout 时间，上限值
- LBOUND 是最小的 timeout 时间，下限值

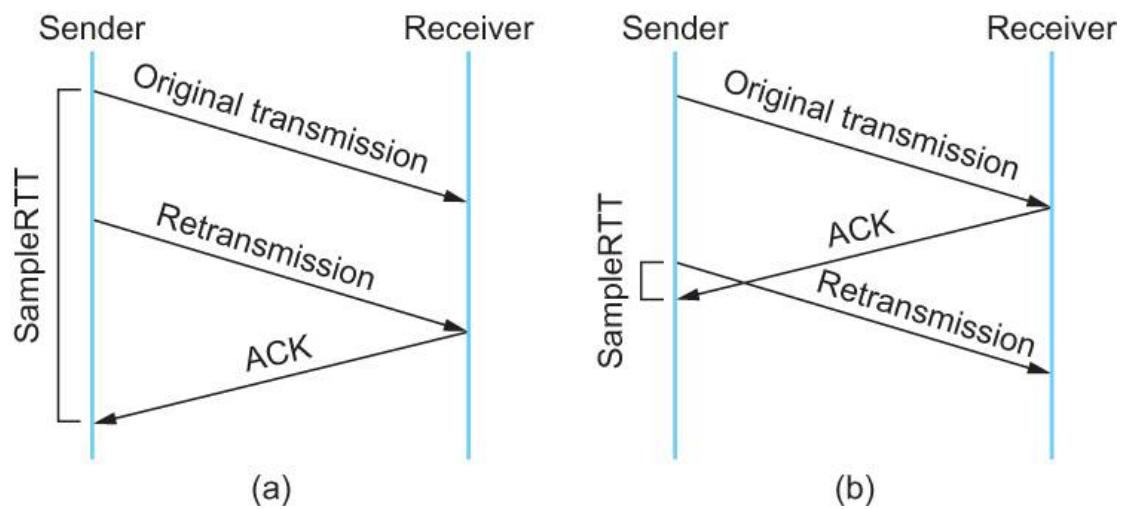
- β 值一般在 1.3 到 2.0 之间。

Karn / Partridge 算法

但是上面的这个算法在重传的时候会出有一个终极问题——你是用第一次发数据的时间和 **ack** 回来的时间做 RTT 样本值，还是用重传的时间和 **ACK** 回来的时间做 RTT 样本值？

这个问题无论你选那头都是按下葫芦起了瓢。 如下图所示：

- 情况 (a) 是 **ack** 没回来，所以重传。如果你计算第一次发送和 **ACK** 的时间，那么，明显算大了。
- 情况(b)是 **ack** 回来慢了，但是导致了重传，但刚重传不一会儿，之前 **ACK** 就回来了。如果你是算重传的时间和 **ACK** 回来的时间的差，就会算短了。



所以 1987 年的时候，搞了一个叫 [Karn / Partridge Algorithm](#)，这个算法的最大特点是——忽略重传，不把重传的 RTT 做采样（你看，你不需要去解决不存在的问题）。

但是，这样一来，又会引发一个大 BUG——如果在某一时间，网络闪动，突然变慢了，产生了比较大的延时，这个延时导致要重传所有的包（因为之前的 RTO 很小），于是，因为重传的不算，所以，RTO 就不会被更新，这是一个灾难。于是 Karn 算法用了一个取巧的方式——只要一发生重传，就对现有的 RTO 值翻倍（这就是所谓的 Exponential backoff），很明显，这种死规矩对于一个需要估计比较准确的 RTT 也不靠谱。

Jacobson / Karels 算法

前面两种算法用的都是“加权移动平均”，这种方法最大的毛病就是如果 RTT 有一个大的波动的话，很难被发现，因为被平滑掉了。所以，1988 年，又有人推出来了一个新的算法，这个算法叫 Jacobson / Karels Algorithm（参看 [RFC6289](#)）。这个算法引入了最新的 RTT 的采样和平滑过的 SRTT 的差距做因子来计算。公式如下：（其中的 DevRTT 是 Deviation RTT 的意思）

$$\text{SRTT} = \text{SRTT} + \alpha (\text{RTT} - \text{SRTT}) \quad \text{—— 计算平滑 RTT}$$

DevRTT = (1- β)*DevRTT + β *(|RTT-SRTT|) ——计算平滑 RTT 和真实的差距（加权移动平均）

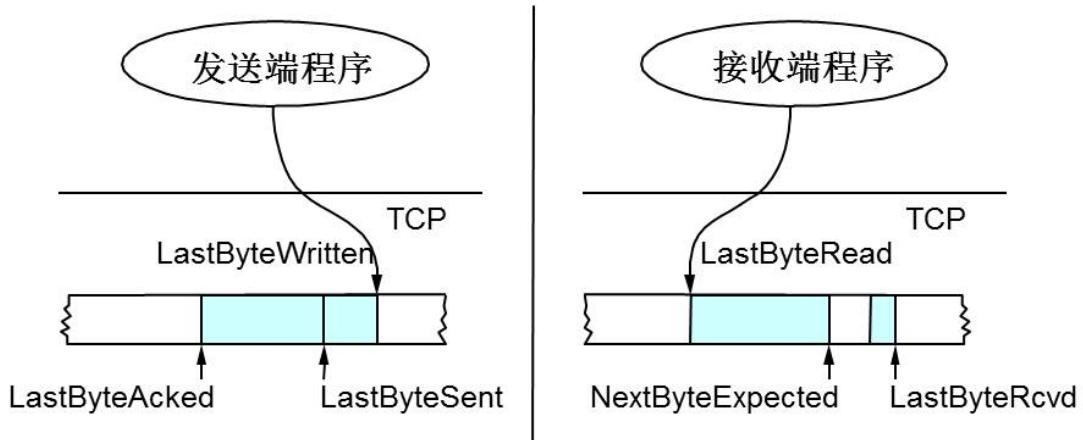
RTO= μ * SRTT + ∂ *DevRTT —— 神一样的公式

(其中：在 Linux 下， $\alpha = 0.125$, $\beta = 0.25$, $\mu = 1$, $\partial = 4$ ——这就是算法中的“调得一手好参数”，nobody knows why, it just works...) 最后的这个算法在被用在今天的 TCP 协议中（Linux 的源代码在：[tcp_rtt_estimator](#)）。

TCP 滑动窗口

我们都知道，TCP 必需要解决的可靠传输以及包乱序（reordering）的问题，所以，TCP 必需要知道网络实际的数据处理带宽或是数据处理速度，这样才不会引起网络拥塞，导致丢包。

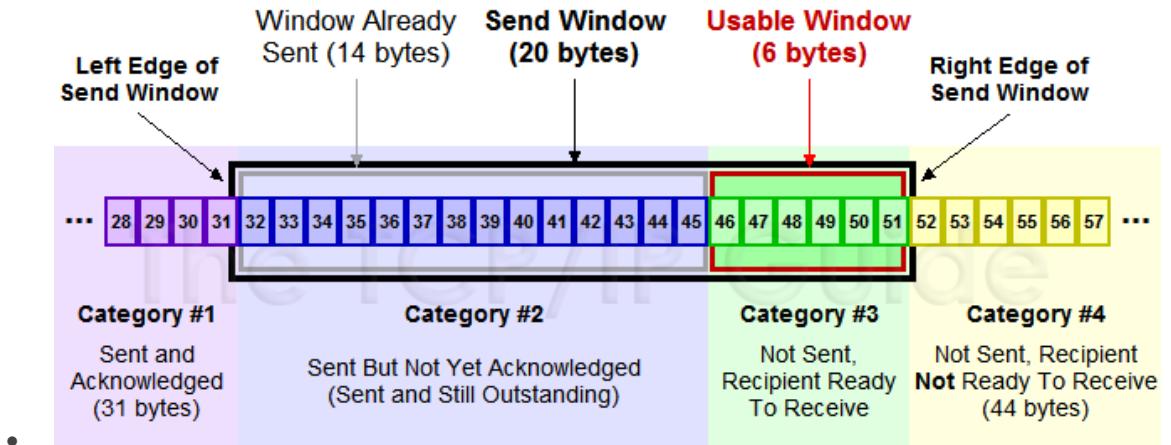
所以，TCP 引入了一些技术和设计来做网络流控，Sliding Window 是其中一个技术。前面我们说过，TCP 头里有一个字段叫 **Window**，又叫 **Advertised-Window**，这个字段是接收端告诉发送端自己还有多少缓冲区可以接收数据。于是发送端就可以根据这个接收端的处理能力来发送数据，而不会导致接收端处理不过来。为了说明滑动窗口，我们需要先看一下 TCP 缓冲区的一些数据结构：



- 接收端 `LastByteRead` 指向了 TCP 缓冲区中读到的位置，`NextByteExpected` 指向的地方是收到的连续包的最后一个位置，`LastByteRcvd` 指向的是收到的包的最后一个位置，我们可以看到中间有些数据还没有到达，所以有数据空白区。
- 发送端的 `LastByteAcked` 指向了被接收端 Ack 过的位置（表示成功发送确认），`LastByteSent` 表示发出去了，但还没有收到成功确认的 Ack，`LastByteWritten` 指向的是上层应用正在写的地方。

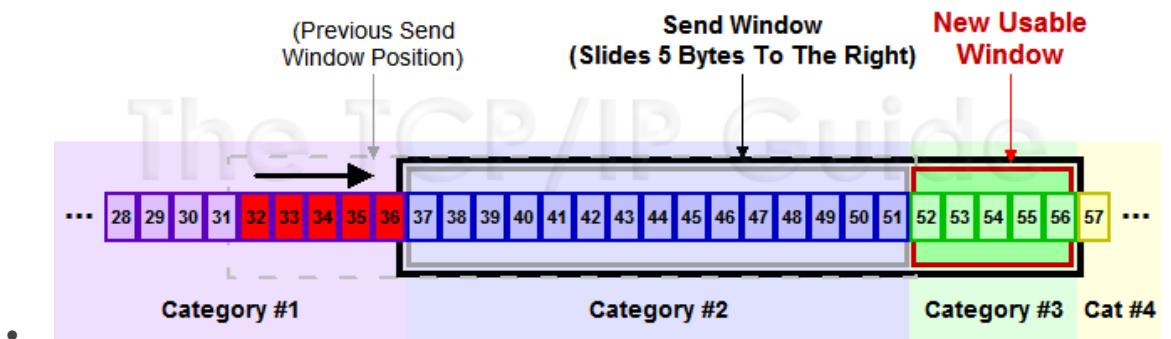
于是：

- 接收端在给发送端回 ACK 中会汇报自己的 `AdvertisedWindow = MaxRcvBuffer - (LastByteRcvd - LastByteRead)`;
- 而发送方会根据这个窗口来控制发送数据的大小，以保证接收方可以处理。
- 下面我们来看一下发送方的滑动窗口示意图：

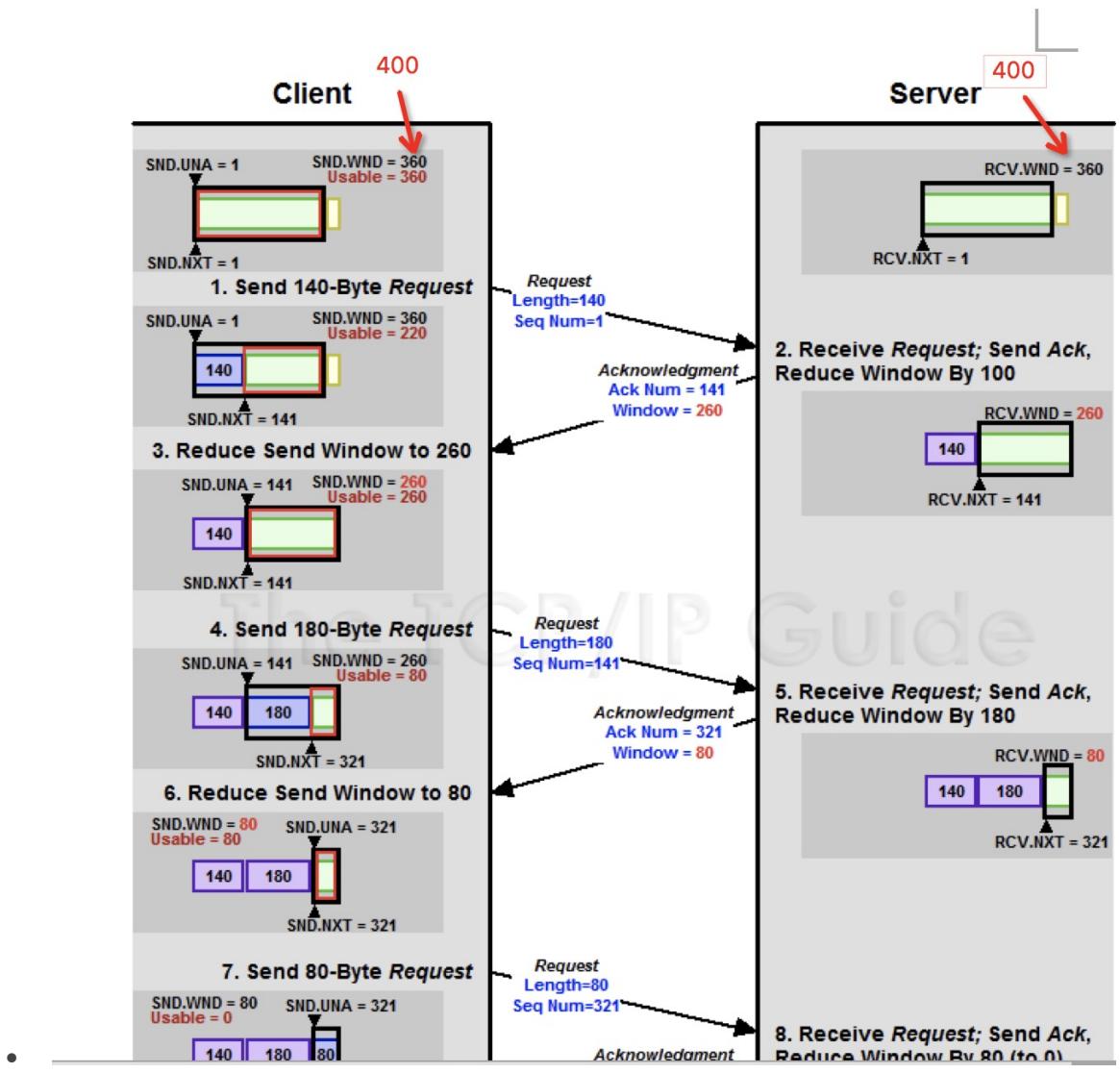


上图中分成了四个部分，分别是：（其中那个黑模型就是滑动窗口）

- #1 已收到 ack 确认的数据。
- #2 发还没收到 ack 的。
- #3 在窗口中还没有发出的（接收方还有空间）。
- #4 窗口以外的数据（接收方没空间）
- 下面是个滑动后的示意图（收到 36 的 ack，并发出了 46-51 的字节）：



- 下面我们来看一个接受端控制发送端的图示：



Zero Window

上图，我们可以看到一个处理缓慢的 Server（接收端）是怎么把 Client（发送端）的 TCP Sliding Window 给降成 0 的。此时，你一定会问，如果 Window 变成 0 了，发送端就不发数据了，你可以想像成“Window Closed”，如果发送端不发数据了，接收方一会儿 Window size 可用了，怎么通知发送端呢？

解决这个问题，TCP 使用了 Zero Window Probe 技术，缩写为 ZWP，也就是说，发送端在窗口变成 0 后，会发 ZWP 的包给接收方，让接收方来 ack 他的 Window 尺寸，一般这个值会设置成 3 次每次大约 30-60 秒（不同的实现可能会不一样）。如果 3 次过后还是 0 的话，有的 TCP 实现就会发 RST 把链接断了。

注意：只要有等待的地方都可能出现 DDoS 攻击，Zero Window 也不例外，一些攻击者会在和 HTTP 建好链发完 GET 请求后，就把 Window 设置为 0，然后服务端就只能等待进行 ZWP，于是攻击者会并发大量的这样的请求，把服务器端的资源耗尽。

Silly Window Syndrome

Silly Window Syndrome 翻译成中文就是“糊涂窗口综合症”。正如你上面看到的一样，如果我们的接收方太忙了，来不及取走 Receive Windows 里的数据，那么，就会导致发送方越来越小。到最后，如果接收方腾出几个字节并告诉发送方现在有几个字节的 window，而我们的发送方会发送这几个字节。

要知道，我们的 TCP+IP 头有 40 个字节，为了几个字节，要达上这么大的开销，这太不经济了。

所以需要避免对小的 window size 做出响应，直到有足够的大的 window size 再响应，这个思路可以同时实现在 sender 和 receiver 两端。

- 如果这个问题是由 **Receiver** 端引起的，那么就会使用 David D Clark's 方案。在 **receiver** 端，如果收到的数据导致 **window size** 小于某个值，可以直接 **ack(0)** 回 **sender**，这样就把 **window** 给关闭了，也阻止了 **sender** 再发数据过来，等到 **receiver** 端处理了一些数据后 **windows size** 大于等于了 **MSS**，或者，**receiver buffer** 有一半为空，就可以把 **window** 打开让 **send** 发送数据过来。
- 如果这个问题是由 **Sender** 端引起的，那么就会使用著名的 [Nagle's algorithm](#)。这个算法的思路也是延时处理，他有两个主要的条件：1) 要等到 **Window Size>=MSS** 或是 **Data Size >=MSS**，2) 收到之前发送数据的 **ack** 回包，他才会发数据，否则就是在攒数据。

TCP 的拥塞处理 – Congestion Handling

上面我们知道，TCP 通过 **Sliding Window** 来做流控（Flow Control），但是 TCP 觉得这还不够，因为 **Sliding Window** 需要依赖于连接的发送端和接收端，其并不知道网络中间发生了什么。具体一点，我们知道 TCP 通过一个 timer 采样了 RTT 并计算 RTO，但是，如果网络上的延时突然增加，那么，TCP 对这个事做出的应对只有重传数据，但是，重传会导致网络的负担更重，于是会导致更大

的延迟以及更多的丢包，于是，这个情况就会进入恶性循环被不断地放大。

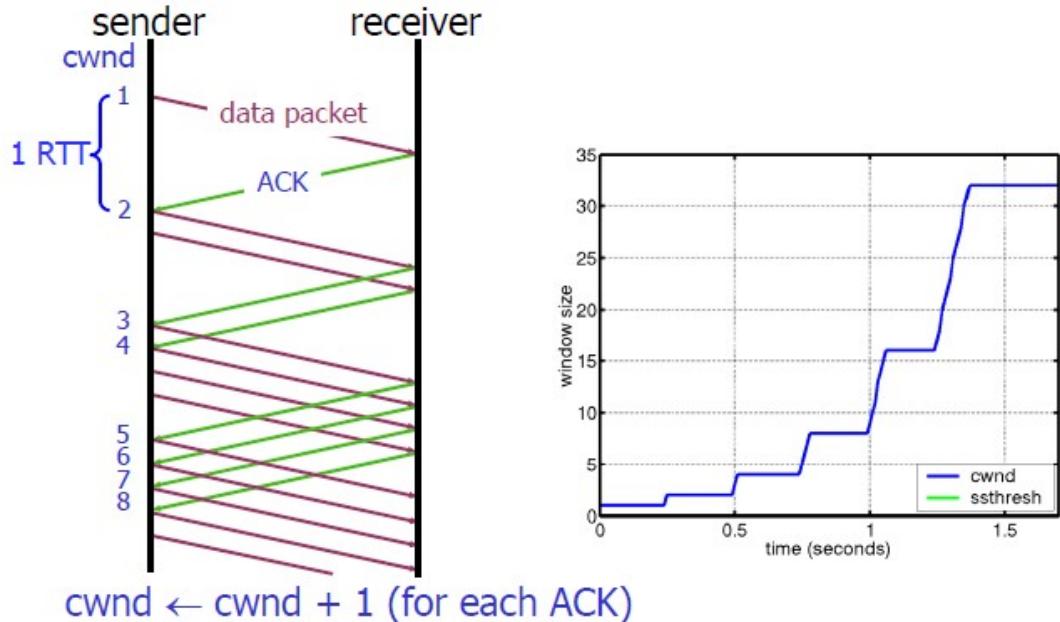
所以，需要根据网络状况 动态调整发送数据的速率。

拥塞控制主要是四个算法： **1) 慢启动， 2) 拥塞避免， 3) 拥塞发生， 4) 快速恢复。**

cwnd 全称 Congestion Window 拥塞窗口

慢启动的算法如下

- 1) 连接建好的开始先初始化 $cwnd = 1$ ，表明可以传一个 MSS 大小的数据。
- 2) 每当收到一个 ACK, $cwnd++$; 呈线性上升
- 3) 每当过了一个 RTT, $cwnd = cwnd * 2$; 呈指数让升
- 4) 还有一个 $ssthresh$ (slow start threshold)，是一个上限，当 $cwnd \geq ssthresh$ 时，就会进入“拥塞避免算法”



在 Google 的论文《[An Argument for Increasing TCP's Initial Congestion Window](#)》，Linux 3.0 后也采用了这篇论文的建议——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](#)，cwnd 是跟 MSS 的值来变的，如果 $MSS < 1095$ ，则 $cwnd = 4$ ；如果 $MSS > 2190$ ，则 $cwnd = 2$ ；其它情况下，则是 3。

拥塞避免算法 – Congestion Avoidance

前面说过，还有一个 ssthresh (slow start threshold)，是一个上限，当 $cwnd \geq ssthresh$ 时，就会进入“拥塞避免算法”。一般来说 ssthresh 的值是 65535，单位是字节，当 cwnd 达到这个值时后，算法如下：

- 1) 收到一个 ACK 时， $cwnd = cwnd + 1/cwnd$
- 2) 当每过一个 RTT 时， $cwnd = cwnd + 1$

这样就可以避免增长过快导致网络拥塞，慢慢的增加调整到网络的最佳值。很明显，是一个线性上升的算法。

拥塞状态时的算法

前面我们说过，当丢包的时候，会有两种情况：

1) 等到 RTO 超时，重传数据包。TCP 认为这种情况太糟糕，反应也很强烈。

- $\text{sshthresh} = \text{cwnd} / 2$
- cwnd 重置为 1
- 进入慢启动过程

2) Fast Retransmit 算法，也就是在收到 3 个 duplicate ACK 时就开启重传，而不用等到 RTO 超时。

- TCP Tahoe 的实现和 RTO 超时一样。
- TCP Reno 的实现是：
 - $\text{sshthresh} = \text{cwnd}$
 - $\text{cwnd} = \text{cwnd} / 2$
 - 进入快速恢复算法——Fast Recovery

- 上面我们可以看到 RTO 超时后，`sshthresh` 会变成 `cwnd` 的一半，这意味着，如果 `cwnd` \leq `sshthresh` 时出现的丢包，那么 TCP 的 `sshthresh` 就会减了一半，然后等 `cwnd` 又很快地以指数级增涨爬到这个地方时，就会成慢慢的线性增涨。我们可以看到，TCP 是怎么通过这种强烈地震荡快速而小心得找到网站流量的平衡点的。

然后，真正的 **Fast Recovery** 算法如下：

- $cwnd = sshthresh + 3 * MSS$ （3的意思是确认有3个数据包被收到了）
- 重传 **Duplicated ACKs** 指定的数据包
- 如果再收到 **duplicated Ack**s，那么 $cwnd = cwnd + 1$
- 如果收到了新的 **Ack**，那么， $cwnd = sshthresh$ ，然后就进入了拥塞避免的算法了。