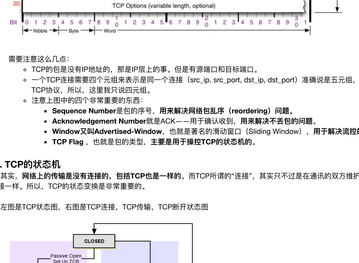
我们知道TCP在网络OSI的七层模型中的第四层——Transport层，IP在第三层——Network层，ARP在第二层——Data Link层，在第二层上的数据，我们叫Frame，在第三层上的数据叫Packet，第四层的数据叫Segment。

我们程序的数据首先会打到TCP的Segment中，然后TCP的Segment会打到IP的Packet中，然后再打到以太网Ethernet的Frame中，传到对端后，各个层解析自己的协议，然后把数据交给更高层的协议处理。

- Transport：传输层
- Network：网络层
- Data Link：数据链路层

- Segment：报文
- Packet：数据包
- Frame：以太帧

## 一. TCP头部格式



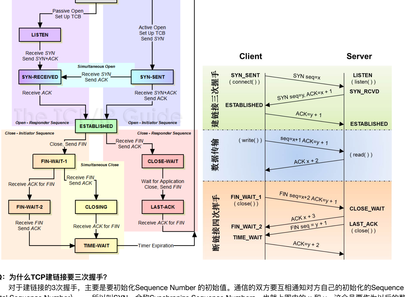

需要注意这么几点：

- TCP协包是没有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，但因为这里只是说TCP协议，所以，这里我只说四元组。
- 注意上图中的四个非常重要的东西：
  - **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。**

## 二. 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。

左图是TCP状态图，右图是TCP连接，TCP传输，TCP断开状态图




**Q：为什么TCP建链要三次握手？**

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：Inital 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会用这个序号来排拨数据）

**Q：为什么TCP断开链接要四次握手？**

对于4次挥手，其实你仔细看是2次，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，那就会就进入到CLOSING状态，然后到达TIME_WAIT状态

## 三. 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。

注意，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。

### 1. 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。因为都在等timeout，timeout可能会很长（在下篇会说TCP是怎么动态地计算出timeout的）

### 1. 快速重传机制

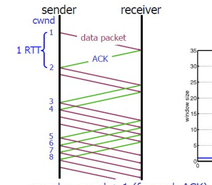
于是，TCP引入了一种叫**Fast Retransmit** 的算法，**不以时间驱动，而以数据驱动重传**。也就是说，如果，包没有连续到达，就ack最后那个可能被丢了的包，如果发送方连续收到3次相同的ack，就重传。Fast Retransmit的好处是不用等timeout了再重传。

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，于是就马上重转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：


## 四. 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被数据，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：




上图中，我们可以看到：

- 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，LastByteRcved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 – 1;
- 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）

- 已收到ack确认的数据。
- 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）

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）




## 五. TCP的拥塞控制

上面我们知道了，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。TCP的设计者觉得，一个伟大而牛逼的协议仅仅做到流控并不够，因为流控只是网络模型4层以上的事，TCP的还应该更聪明地知道整个网络上的事。

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。试想一下，如果一个网络内有成千上万的TCP连接都这么行事，那么马上就会形成“网络风暴”，TCP这个协议就会拖垮整个网络，这是一个灾难。

所以，TCP不能忽略网络上发生的事情，而无脑地一个劲地重发数据，对网络造成更大的伤害，对此TCP的设计理念是：**TCP不是一个自私的协议，当拥塞发生的时候，要做自我牺牲，就像交通阻塞一样。每个车都应该把路让出来，而不要再去抢路了。**

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**

### 1. 慢热启动算法 – Slow Start

首先，我们来看一下TCP的慢热启动。慢启动的意思是，刚刚加入网络的连接，一点一点地提速，不要一上来就像那些特权车一样霸道地把路占满。新同学上高速还是要慢一点，不要把已经在高速上的秩序给搞乱了。

慢启动的算法如下(cwnd全称Congestion Window)：

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 >= ssthresh时，就会进入“拥塞避免算法”（后面会说这个算法）

所以，我们可以看到，如果网速很快的话，ACK也会返回得快，RTT也会短，那么，这个慢启动就一点也不慢。下图说明了这个过程。




### 2. 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 >= 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