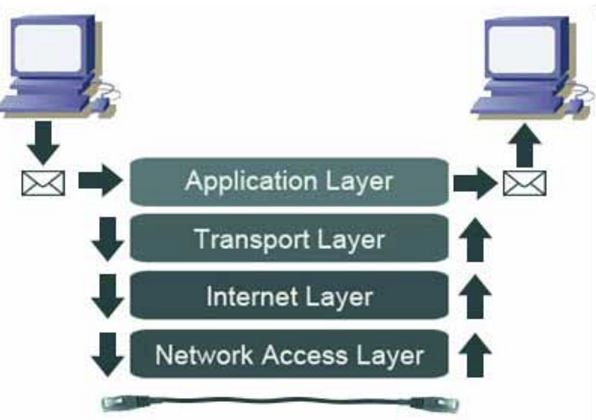
**漫谈TCP**

<http://www.cnblogs.com/yjf512/p/5909031.html#3598096>

不得不承认，tcp是一个非常复杂的协议。它包含了RFC793及之后的一些协议。能把tcp的所有方面面面具到地说清楚，本身就是个很复杂的事情。如果再讲得枯燥，那么就会更让人昏昏欲睡了。本文希望能尽量用稍显通俗的话把tcp描述清楚。

**关于分层**

请忘掉大学课本上学的七层模型，我们使用四层模型更为贴合我们的实际网络。应用层，传输层，网络层，网络接入层。



分层是为什么，其实和公司中职位是一样的，不同职位的人做不同的事情，然后不同职位的人合起来，一起完成了数据传输的事情。

**网络传输层**

网络传输层负责最底层的底层链路连接。两台主机之间进行互联，基于网线的物理硬件上的协议。在这个层面，主机与主机的交互只认得硬件mac编号，并不认识IP。这个层需要了解的一个概念是MTU，网络中每个路由都会设置一个MTU，代表这个路由中能通过的最大的包的大小。那么整个网络链路的MTU值就是由网络中所有路由的最小MTU决定。这个就好比水路管道，水流量是由管道链路中管子最小的那个链路来决定的。

**网络层**

IP的出现是很有必要的，就像给网络上每个机器一个门牌号，网络层，你可以把它理解为邮件运输工，它的职能就是负责把一包东西，从这个门牌运输到另外一个门牌。

**传输层**

传输层相比于网络层最大的不同就是引入了端口的概念。网络层只管发送地址和目的地址。但是发送主机上有可能有多个程序和同一个接收主机进行传输数据，怎么区分这多个程序呢？就引入了端口的概念。（发送IP地址，发送端口，接收IP地址，接收端口）四元组标示了一个主机的程序到另一个主机程序的唯一标示。传输层的职能，就是维护这个四元组。

其实传输层还有一个职能是定义发送方和接收方基本处理包的行为。上面说到网络层就相当于邮件运输工，它只负责把一包东西从一个地方放到另外一个地方，但是，这包东西是否送达了，送达之后接收方又有什么行为。这些都可以在传输层进行定义。注意，这里说的是可以，你也可以在传输层布不管这些，只做简单的基本封装四元组。你懂的，我说的就是UDP。

**应用层**

应用层，就更抽象一层了。我们这个端口和那个端口的连接是用来干什么的，传输文件？那么可以使用FTP。传输文本？那么可以使用HTTP。应用层就是实际上对具体的程序之间的交互功能进行定义的层。

如果你比较钻牛角尖的话，可能会提出的问题是：为什么要分为四层？为什么不是分为两层，五层？

这里分为几个问题解释：

**为什么要分层？**

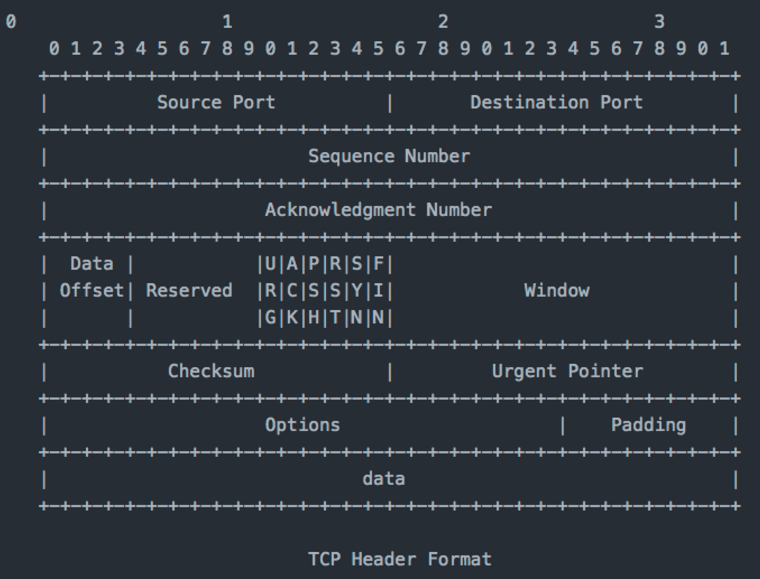
分层实际上是一个抽象的过程，和我们写代码的时候的封装是一样的。你要说，我只有一层，一个协议把所有从程序到物理网络的所有东西都描述一遍，这个可以不可以？可以。但是，这样你协议写多了以后，就会发现，物理网络部分几乎所有协议都一样，那么我倒不如另外写一个协议，然后其他协议使用include的方式来包含这个协议。好了，网络传输层就出来了。后面的原理也几乎一样了。所以分层是必要的。

**为什么分为四层？**

代码中有个过度设计的概念，分层中也照样有个过度设计的概念。基本上，分层层数越多，是越符合抽象的。但是到最后我们会发现，有一个协议整篇都是include，自己实际上并没有任何实质的东西。这个就是过度设计。基本上我们琢磨来琢磨去，按照上面说的，分为四层，是个很合适的设计。四层中的每一层都有自己负责的一块内容，内容大小适中，又没有相互耦合的地方。四层中的每一层都不可缺少。

**TCP的协议结构**

说到一个协议，最先应该展示的是它的结构。



整个结构正如车子的零部件一样，每个字段都有对应的作用：

**Source Port和Destination Port**

这两个字段表示的就是发送地址和目的地址的端口号。或许有人会问，那四元组中的其他两个发送主机IP和目的主机IP呢？ok，那是IP层的事情，请查看IP协议头。

**Sequence Number和Acknowledgment Number**

这两个字段就有的聊了。

首先第一个问题，序列号做什么用呢？

序列号是用来标记包的顺序的，假设有一段要传输的内容大小有9000字节，按照1460字节一个包的大小，假设初始序号为10000，那么我们就把这段内容分为10000-11460， 11460-12920，... 18760-19000 一共7个包。网络包由于网络问题，可能并不是顺序到达接收端的，那么接收端可以按照序列号来重新组装这段内容。

第二个问题，序列号有两个说明什么？

说明tcp是全双工的，就是说，tcp的任意一端可以发送数据，也可以接收数据。那么需要有个发送序列号seqence Number和接收序列号 Acknowledgement Number。

**Data offset 和 Reserved**

由于tcp头可能是不固定大小的（因为存在可选字段），所以需要有这个值来表示当前这个包的tcp头有多大。

Reserved就是保留字段

**Tag位**

就是上图中的URG，ACK，PSH，RST，SYN，FIN位，每个位置表示的意思是：

URG：紧急位，RFC已经建议废弃

ACK：说明这个包中带有回复信息

PSH：说明这个包中有传输数据

RST：重置位，说明这个包是用来要对方重置连接

SYN：建立连接，说明发送方向另一方发送建立连接的请求

FIN：结束位，说明发送一方告知另外一方，要请求中断连接

熟悉这些Tag位是非常必要的，我们一般讨论包请求的时候，使用的术语一般就是：

发送方请求一个SYN，接收方返回一个ACK。每每看到这种字眼，请不要傻眼。

还有一个误区，一个包是不是只能包含一个tag位？不是的。一个包可以包含一个或者多个tag位。比如一个包可以有ACK的功能，也能同时有SYN的请求功能。（在TCP三次握手的第二次握手的时候就是携带了ACK+SYN的标志位）。

**Window**

这个值就是著名的滑动窗口值。滑动窗口是接收端告诉发送端下次可以发送多少包。好吧，这里也需要面对几个问题：

避免误区：发送方和接受方的请求，响应并不是一一对应的。

网络上并不是只有发送方发一个请求，接收方回复一个ACK这种模式的。他们交互的模式更可能是：发送方一次发送多个请求包，接收方回复一个ACK，把这些请求包都回复了。这个使用前面的Acknowledgment Number是可以做到的。

但是基本上，在接收方的角度，ACK包一定是收到一个包之后，才返回一个ACK，就是说，没有无缘无故的发送重复ACK，没有一个请求，多个ACK这种情况。但是有多个请求，多个重复ACK的情况，这个时候，往往说明某个请求的包丢失了。

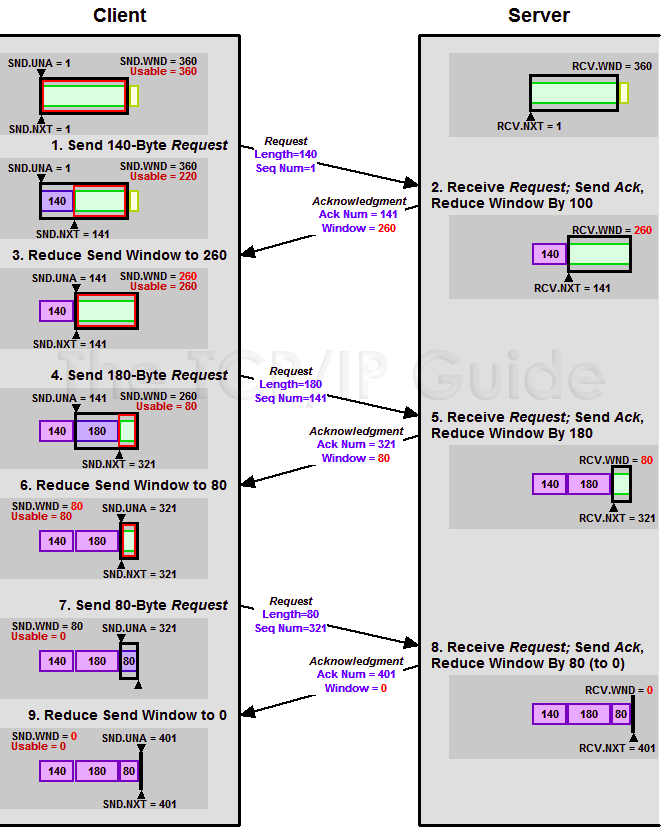
为什么需要有滑动窗口存在？

滑动窗口的存在是为了控制网络上包的数量。如果没有滑动窗口，那么就是一个很理想很理想的情况，发送方一有数据，加上包头达到MTU大小，直接发送，就和冲锋枪一样，突突突突。但是呢？这样子，实际上，没有考虑到接收方是否能接收完。接收端就像一个一直在吃饭的胖子，他的吃饭速度是固定的，它一次性最多能吃10碗饭，某个时刻可能已经吃了两碗饭，但是还没消化。所以这个时候，它只能再吃8碗饭了，如果这个时候你一下子给它80碗，必然导致它堵死了，吃不下不吃下。这个滑动窗口就是接收端告诉发送端我还能吃几碗饭的通话器。

总结下，这里已经有两个条件限制发送方的效率了，一个是MTU，全链路MTU大小，限制每次最大发送的包的大小。另一个是滑动窗口，限制发送方一次发送的包数量。

为什么叫滑动窗口？

滑动窗口我更愿意理解为发送方和接收方共同维护的。分别有发送窗口和接收窗口区别。



发送方数据有几个状态：数据已发送未收到ACK，数据已发送收到ACK

接收方数据有几个状态：数据已收到未被应用层消费，数据已收到已被应用层消费

把发送数据横拍做长列状，发送方一但有数据收到ACK，那么滑动窗口左侧边就进行左移。同样，一旦接收方有数据被应用层消费，那么，滑动窗口的右侧边就进行右移。整个过程，就好比努力爬行的蚯蚓，尾巴向前挪一寸，头部再向前走一寸，直到把整个数据都从头到尾移动完毕。

回到tcp的windows字段，这个字段是接收端回复给发送端，告诉发送端接收端的窗口大小的。我们其实默认也把这个窗口大小叫做滑动窗口大小。

关于滑动窗口的概念的理解，我的感触是网上各种各样对这个滑动窗口的描述，不要陷入到咬文嚼字中，头脑中形象有这个滑动窗口的滑动过程，就可以了，很多文章很多描述可能是前后矛盾的。比如，下面两个关于发送窗口的描述：

* 发送窗口是由滑动窗口和拥塞窗口共同决定的。
* 发送窗口是由接收窗口决定的。

**CheckSum**

校验和。就是对TCP的头部和数据部分进行检验，是否在中途被篡改过。它和IP头中的校验和的算法是一样的，只是IP校验内容中不包括数据，但是TCP是包括头部和数据两个部分的。

**Urgent Pointer**

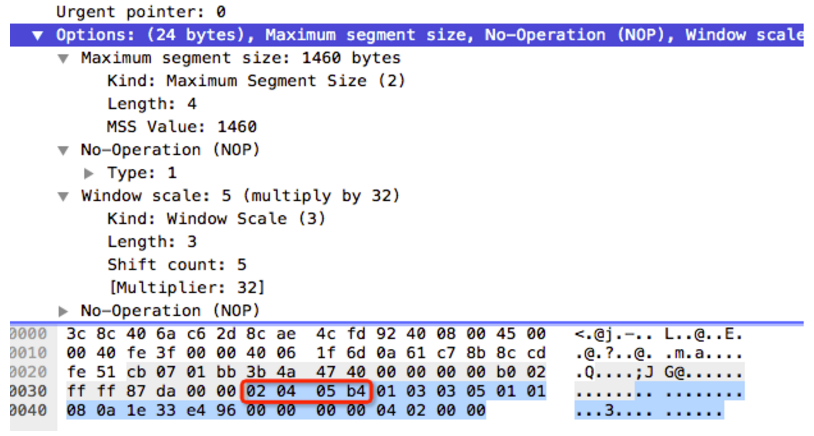
紧急数据指针。紧急数据指的是发送端告诉接收端，这个数据是非常紧急的，请优先读取，设计初期可能是由于考虑到中断或者异常等情况，但是在RFC6093中已经明确，紧急数据已经是废弃功能了。不建议使用。只为旧程序兼容而使用。

所以，对于Urgent Porinter和tag中的URG标示就不要使用了。

**Options 和 padding**

options字段相当于扩展使用的，RFC有哪些信息要传输，而头部没有安排的字段，就可以放在options中进行传输。padding是为了对齐字节位。

它根据kind＋length+ value的形式来定义存储哪些属性。具体看下图的例子：



比如kind =2 代表存储的是MSS值（最大内容大小，MTU-IP头-TCP头），它有四个字节的长度，具体值为1460（05 b4）

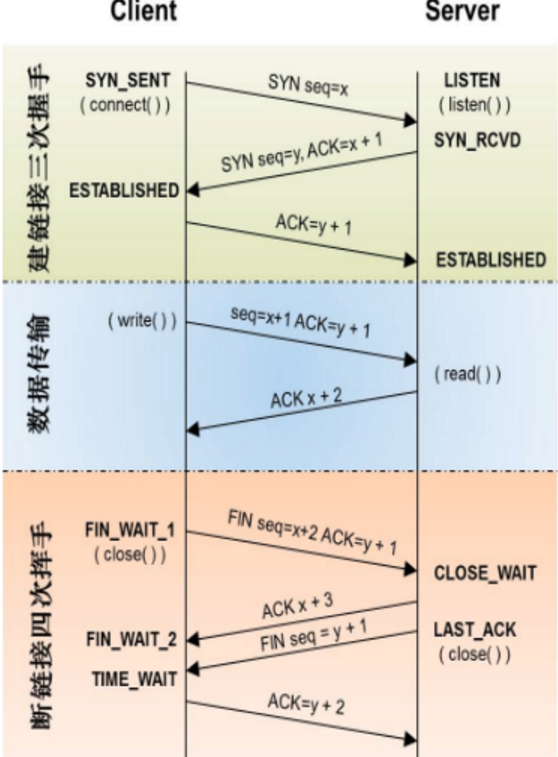
具体kind和对应的值的映射可以参考 http://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml

总儿言之，这个options字段增加了TCP的可扩展性。但是确实，到现在，它包含的内容也越来越复杂了。

**握手连接**

一个最常遇到的月经鸡汤面试题就是，UDP和TCP有什么不同。嗯，猴子和老虎就是不一样的。经常我们会提及的一点就是TCP是可靠的，UDP是不可靠的。TCP的可靠体现在哪里呢？握手连接的建立和消失就是其中一个体现。

TCP著名的三次握手和四次挥手



这个图里面的client和server应该理解为发送方和接收方。下面这一串描述请熟练练习到像串口相声一样：发送方发送一个SYN到接收方请求建立连接，接收方返回一个ACK确认收到请求，并携带一个SYN给发送方请求建立双向连接，发送方再返回一个ACK给接收方确认，这个时候连接就建立了。

顺势说下四次挥手吧。发送方发送一个FIN给接收方主动请求断开连接，接收方返回一个ACK确认，接着接收方再发送一个FIN请求断开另一方向的连接，发送方收到之后返回一个ACK确认。这个时候，连接就中断了。

在三次握手和四次挥手的时候，发送方和接收方的socket是有状态的，对，就是你使用netstat 能看到机器上socket的状态。

SYN\_SENT/SYN\_RCVD/ESTABLISH/FIN\_WAIT1/CLOSE\_WAIT/FIN\_WAIT2/LAST\_ACK/TIME\_WAIT

**backlog**

linux的TCP模块维护两个队列，半连接队列和链接队列，当三次握手的时候，收到第一个SYN，发送完ACK之后，就会把这个连接放入到半连接队列中，当第三步完成的时候，连接建立了，就把连接从半连接队列放到连接队列中。

半连接队列长度是由net.ipv4.tcp\_max\_syn\_backlog进行设置的。

连接队列的长度由我们创建socket的时候指定的backlog和net.core.somaxconn其中的较小值确定的。

如果backlog或者net.core.somaxconn设置过小，那么很多连接就无法建立，服务端会发送RST拒绝连接，这个也是很多服务器性能上不去或断开连接的原因。

**SYN\_FLOOD攻击**

三次握手，我们假设server端是按照正常的流程走的，但是client端是邪恶的，它发送了SYN之后，server端返回了ACK+SYN，但是client端一直hold住，或者直接掉线，不发送ack了，那么这个时候，server端就一直保持在SYN\_RCVD状态。

如果这种client端非常多，就会把前面说的半连接队列塞满，后面的连接就无法建立了，这个server的服务也就给中止了。

这种攻击就叫做SYN\_FLOOD攻击。

**TIME\_WAIT状态**

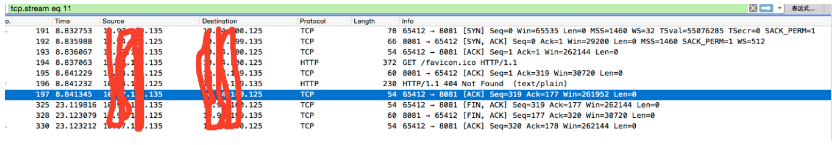
说到四次挥手，主动发起请求的一方会在TIME\_WAIT状态持续2MSL。MSL就是Maximum Segment Lifetime，一个包在网络上存活的最长时间，linux设置的MSL是30s。这个是为了防止最后一个ACK可能被丢失了，那么在2MSL中如果收到对方重复发送的FIN包，就需要重新发送ACK来关闭连接。TCP的这种行为，我们可以看作是一种负责任的行为，主动请求关闭的一方在很大程度上确保了对方收到断开确认请求之后才关闭这个连接的。当然，这也能保证了如果我这个端口被其他程序复用了，旧的请求不会发一个莫名其妙的FIN过来。

首先确认，TIME\_WAIT不是邪恶的，我们可能在服务器上很经常会看到TIME\_WAIT的连接，不必惊慌，除非这种连接数已经超过了系统的fd数，如果没有超过的话，我的建议，不要在压TIME\_WAIT数量上太下功夫，找出什么导致出现大量TIME\_WAIT的原因比较靠谱（大多数是网络问题）。

不过，服务端确实应当尽量避免TIME\_WAIT留在服务端，不管怎样，这个会消耗一些资源的。所以，把TIME\_WAIT留在客户端，服务端不主动断开连接是一个很好的方法，比如HTTP协议提供的KEEP\_ALIVE就是在这个方面做的很好的，客户端连接的时候告诉服务器不要主动断开连接。

**四次挥手的第二次请求和第三次请求能不能一次发送？**

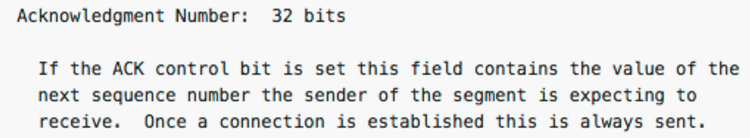
很容易会出现这个问题，第二次请求和第三次请求能不能一起发送呢？这样是不是能节约性能？答案是可以的。而linux也确实是这么实现的。你具体抓一个包，就会发现第二次请求和第三次请求是一起发送的。



**关于ACK**

在实际网络中抓包，你会发现，除了SYN之外，所有的请求包都带有ACK标示。即使是上图中的#197条已经对seq为177的消息发送了ACK，# 325也还会发送一个ACK seq=177。

这个是RFC的建议



建立连接之后，每个请求都要带上ACK标志。我们可以这么理解，由于在TCP的机制中，ACK是无法确认中途有没有丢失的，那么本着不发白不发的原则，每个请求都顺带带上当前已经ACK的信息。

**拥塞阻塞**

TCP不是一个自私的协议，它的设计充分考虑了互联网的大环境。试想，如果所有的网络发送方都不管网络的情况，明明网络已经堵塞了，还一个劲地发送大量包，甚至重发，那么这个时候，大家都没得玩了。于是，渐渐的，TCP引入了拥塞窗口（cwnd）的概念。拥塞窗口的存在单纯是为了避免网络上有超过当前网络能力而造成堵塞。拥塞窗口的单位是报文段个数。比如我们平时会说，现在的拥塞窗口为3，代表发送端可以一次性发送3个报文段。当然，实际上，发送端的最大发送窗口数取决于拥塞窗口（cwnd）和滑动窗口（win）的最小值。

控制网络拥塞的算法为拥塞算法，这个算法在不断演变，在不同操作系统中也有不同实现。

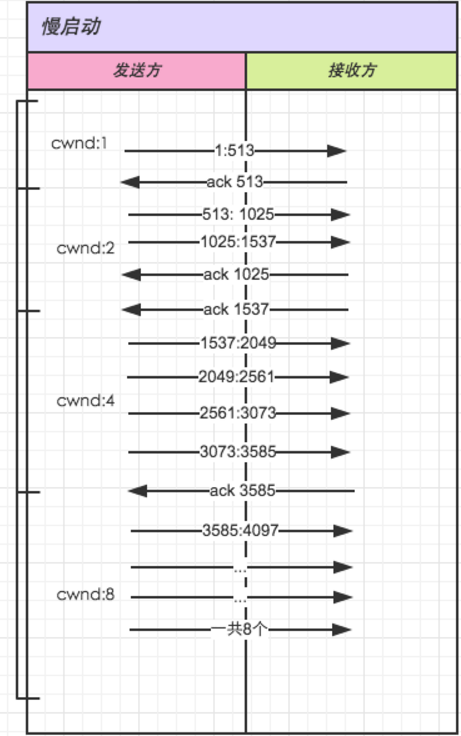
**慢启动**

控制拥塞我们首先会想到在刚刚连接网络的时候，是不是最好先慢慢检测网络情况，再确定发送包的数量。这就是我们说的慢启动算法。发送方从1个包开始，收到ACK，下次就发送2个包，收到这两个包的ACK（请注意，这里有可能只有一个ACK），下次就发送4个包。

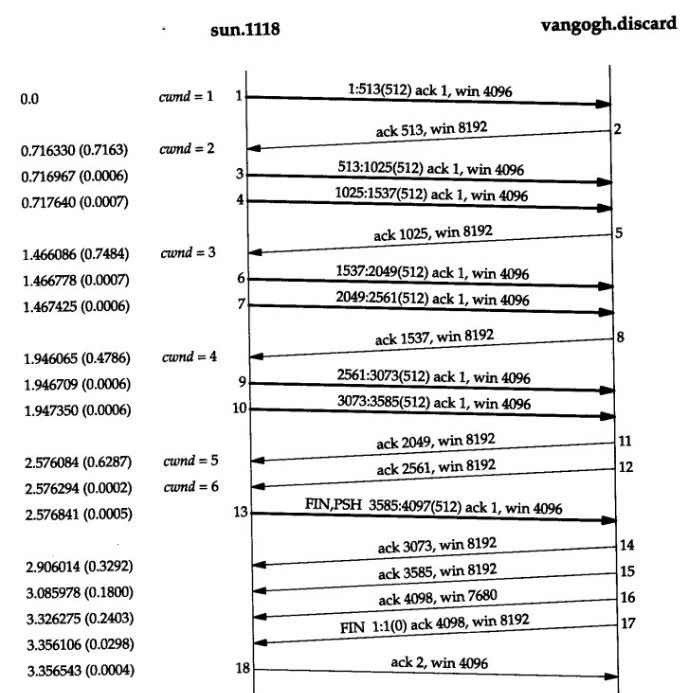
“每收到一个ACK，拥塞窗口就增加一个报文段”。

这句话我更愿意理解为“每确认一个包被ACK了，拥塞窗口就增加一个报文段”

这句话的理解就是，由于有“延迟ACK”算法，很有可能，当发送方发送两个请求包过来的时候，我只发送一个ACK。确认你发送的两个包，这个时候，cwnd实际上是加2，而不是加1。如下图中的cwnd为4的ACK。



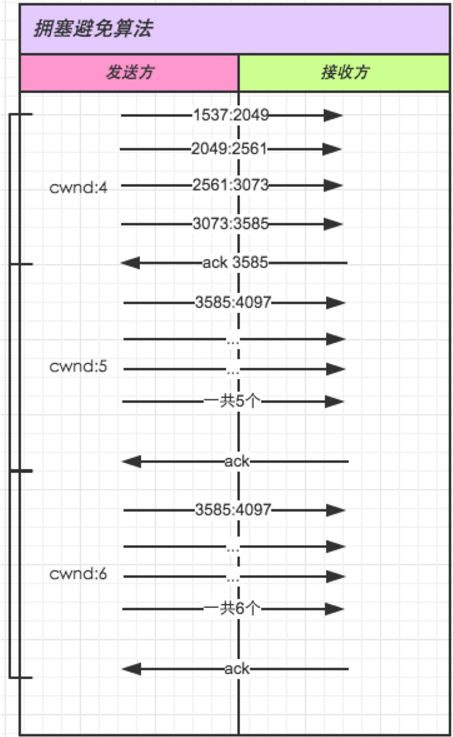
当然上图的情况太理想，实际的情况，坑cwnd为2的请求发出去两个报文包的时候，先返回了一个ACK，然后cwnd这个时候就为3，发送方就会继续发送请求包。。。更贴近实际的正如这个图：



**拥塞避免算法**

慢启动使得cwnd是呈指数增长。一定不可能是无限增长的，这里就有个阀值，超过这个阀值，就进入拥塞避免算法。

先说拥塞避免算法，拥塞避免算法说的是拥塞窗口的增加不再是“每收到一个ACK，拥塞窗口就增加一个报文段”。 而是“每收到一个ACK，cwnd = cwnd + 1/cwnd”。 这个就代表，

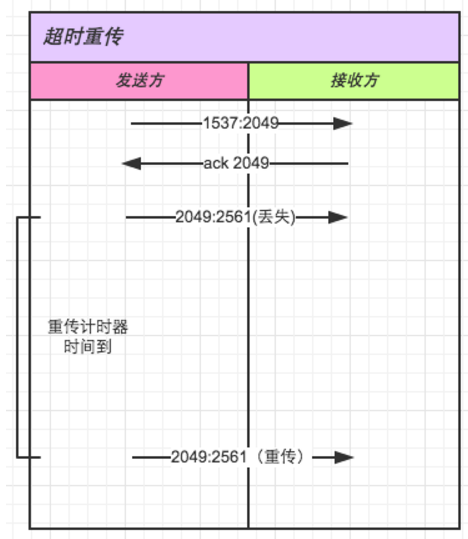


**判断拥塞**

我们怎么判断拥塞呢？有两种判断方法：

a 超时重传（发出去的包在指定时间内没有收到ACK）

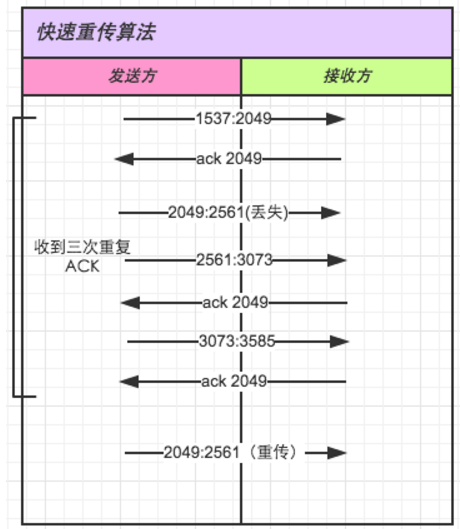
这个指定时间是通过超时定时器来计算的，发出去一个包，超时定时器就开始计时，当超时定时器到时间之后，没有收到ACK，那么这个时候就判断为拥堵了。需要进行重传。



当被这个情况触发，TCP认为网络情况非常糟糕，所以会直接把cwnd调整为1，sshthread 调整为cwnd/2 。 重新进入到慢启动流程。

b 快速重传（重复收到ACK）

这个是由于发送方一次性发送多个请求（比如5个请求，但是第二个请求丢失了，第一三四五请求到了接收端）三四五请求触发了三个ACK返回，但是由于接收端没有收到请求一，返回的三个ACK都是ACK一的，所以发送方就表现为收到重复ACK。当连续收到三条重复ACK的时候就进行重传，不需要等待重传计时器



这个时候TCP会觉得网络还是可以的，反应不会那么激烈，cwnd调整为cwnd/2, sshthresh调整为cwnd大小，进入快速恢复算法。

**快速恢复算法**

快速恢复算法是为了不要有一个重传就那么大响应。能尽快恢复到网络流畅时候稳定的状态。

* 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 Acks，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。

**定时器**

TCP中有四个定时器，有的定时器之前已经说过了。

重传定时器

是为了重传的时候使用的。

2MSL定时器

在上面说到TCP挥手的时候，四次挥手中最后一次挥手，主动发起的一方会进入TIME\_WAIT状态2MSL的时常，这个定时器就是用来计算这个的。

坚持定时器

当滑动窗口为0的时候，发送方不会再发送包给接收方了。但是不发送包怎么知道接收方现在的窗口是不是还为0呢。这个时候就需要不定时去接收方咨询是否滑动窗口还为0。这个不定时的算法就是使用坚持定时器来进行咨询的。

这个算法是使用TCP指数退避方法，第一次1.5秒，第二次1.5x2秒，第三次1.5x4... 以此规律来进行轮询的。

保活定时器

tcp有个keepalive机制，这个只有在一定时间内（tcp\_keepalive\_time，默认每2个小时），没有数据包传递了，发送方在发送心跳检测，如果发送成功，则连接继续，如果没有正常返回，则在指定次数内（tcp\_keepalive\_probes，默认是9次），指定间隔（tcp\_keepalive\_intvl，默认是17s）发送心跳包。

后记

tcp的东西真是比较复杂。好吧，这玩意根本没法讲通俗。我的理解或许有偏差，如果有错误，希望能帮忙留言提出。同时，强烈建议，看完这篇再看一遍耗子哥的《TCP的那些事儿》(http://coolshell.cn/articles/11564.html)。或许还能回来告诉下我哪些地方我理解有问题。