说明：

1) 本文以TCP的发展历程解析容易引起混淆，误会的方方面面
2) 本文不会贴大量的源码，大多数是以文字形式描述，我相信文字看起来是要比代码更轻松的
3) 针对对象：对TCP已经有了全面了解的人。因为本文不会解析TCP头里面的每一个字段或者3次握手的细节，也不会解释慢启动和快速重传
义
4) 除了《TCP/IP详解》(卷一，卷二)以及《Unix网络编程》以及Linux源代码之外，学习网络更好的资源是RFC
5) 本文给出一个提纲，如果想了解细节，请直接查阅RFC
6) 翻来覆去，终于找到了这篇备忘，本文基于这篇备忘文档修改。

1.网络协议设计

ISO提出了OSI分层网络模型，这种分层模型是理论上的，TCP/IP最终实现了一个分层的协议模型，每一个层次对应一组网络协议完成一组特
功能，该组网络协议被其下的层次复用和解复用。这就是分层模型的本质，最终所有的逻辑被编码到线缆或者电磁波。

分层模型是很好理解的，然而对于每一层的协议设计却不是那么容易。TCP/IP的漂亮之处在于：协议越往上层越复杂。我们把网络定义为互
接在一起的设备，网络的本质作用还是“端到端”的通信，然而希望互相通信的设备并不一定要“直接”连接在一起，因此必然需要一些中
设备负责转发数据，因此就把连接这些中间设备的线缆上跑的协议定义为链路层协议，实际上所谓链路其实就是始发与一个设备，通过一根
终止于另一个设备。我们把一条链路称为“一跳”。因此一个端到端的网络包含了“很多跳”。

2.TCP和IP协议

终止于IP协议，我们已经可以完成一个端到端的通信，为何还需要TCP协议？这是一个问题，理解了这个问题，我们就能理解TCP协议为何成
在这个样子，为何如此“复杂”，为何又如此简单。

正如其名字所展示的那样，TCP的作用是传输控制，也就是控制端到端的传输，那为何这种控制不在IP协议中实现的。答案很简单，那就是这
增加IP协议的复杂性，而IP协议需要的就是简单。这是什么原因造成的呢？

首先我们认识一下为何IP协议是沙漏的细腰部分。它的下层是繁多的链路层协议，这些链路提供了相互截然不同且相差很远的语义，为了互联
些异构的网络，我们需要一个网络层协议起码要提供一些适配的功能，另外它必然不能提供太多的“保证性服务”，因为上层的保证性依赖
的约束性更强的保证性，你永远无法在一个100M吞吐量的链路之上实现的IP协议保证1000M的吞吐量...

IP协议设计为分组转发协议，每一跳都要经过一个中间节点，路由的设计是TCP/IP网络的另一大创举，这样，IP协议就无需方向性，路由信
协议本身不再强关联，它们仅仅通过IP地址来关联，因此，IP协议更加简单。路由器作为中间节点也不能太复杂，这涉及到成本问题，因此路
器只负责选路以及转发数据包。

因此传输控制协议必然需要在端点实现。在我们详谈TCP协议之前，首先要看一下它不能做什么，由于IP协议不提供保证，TCP也不能提供你
IP下层链路的这种保证，比如带宽，比如时延，这些都是链路层决定的，既然IP协议无法修补，TCP也不能，然而它却能修正始于IP层的一些
可保证性质”，这些性质包括IP层的不可靠，IP层的不按顺序，IP层的无方向/无连接。

将该小节总结一下，TCP/IP模型从下往上，功能增加，需要实现的设备减少，然而设备的复杂性却在增加，这样保证了成本的最小化，至于
或者因素，靠软件来调节吧，TCP协议就是这样的软件，实际上最开始的时候，TCP并不考虑性能，效率，公平性，正是考虑了这些，TCP协
复杂了起来。

3.TCP协议

这是一个纯软件协议，为何将其设计上两个端点，参见上一小节，本节详述TCP协议，中间也穿插一些简短的论述。

3.1.TCP协议

**确切的说，TCP协议有两重身份，作为网络协议，它弥补了IP协议尽力而为服务的不足，实现了有连接，可靠传输，报文按序到达。作为一
机软件，它和UDP以及左右的传输层协议隔离了主机服务和网络，它们可以被看做是一个多路复用/解复用器，将诸多的主机进程数据复用/
用到IP层。**可以看出，不管从哪个角度，TCP都作为一个接口存在，作为网络协议，它和对端的TCP接口，实现TCP的控制逻辑，作为多路复
解复用器，它和下层IP协议接口，实现协议栈的功能，而这正是分层网络协议模型的基本定义(两类接口，一类和下层接口，另一类和对等层
口)。

我们习惯于将TCP作为协议栈的最顶端，而不把应用层协议当成协议栈的一部分，这部分是因为应用层被TCP/UDP解复用了之后，呈现出了
太复杂的局面，应用层协议用一种不同截然不同的方式被解释，应用层协议习惯于用类似ASN.1标准来封装，这正体现了TCP协议作为多路复
解复用器的重要性，由于直接和应用接口，它可以很容易直接被应用控制，实现不同的传输控制策略，这也是TCP被设计到离应用不太远的地
的原因之一。

总之，**TCP要点有四，一曰有连接，二曰可靠传输，三曰数据按照到达，四曰端到端流量控制。**注意，TCP被设计时只保证这四点，此时它是也有些问题，然而很简单，然而更大的问题很快呈现出来，使之不得不考虑和IP网络相关的东西，比如公平性，效率，因此增加了拥塞控制样TCP就成了现在这个样子。

3.2.有连接，可靠传输，数据按序到达的TCP

IP协议是没有方向的，数据报传输能到达对端全靠路由，因此它是一跳一跳地到达对端的，只要有一跳没有到达对端的路由，那么数据传输失败，其实路由也是互联网的核心之一，实际上IP层提供的核心基本功能有两点，第一点是地址管理，第二点就是路由选路。TCP利用了IP路由简单的功能，因此TCP不必考虑选路，这又一个它被设计成端到端协议的原因。

既然IP已经能尽力让单独的数据报到达对端，那么TCP就可以在这种尽力而为的网络上实现其它的更加严格的控制功能。TCP给无连接的IP网信增加了连接性，确认了已经发送出去的数据的状态，并且保证了数据的顺序。

3.2.1.有连接

这是TCP的基本，因为后续的传输的可靠性以及数据顺序性都依赖于一条连接，这是最简单的实现方式，因此TCP被设计成一种基于流的协议然TCP需要事先建立连接，之后传输多少数据就无所谓了，只要是同一连接的数据能识别出来即可。

疑难杂症1：3次握手和4次挥手

TCP使用3次握手建立一条连接，该握手初始化了传输可靠性以及数据顺序性必要的信息，这些信息包括两个方向的初始序列号，确认号由初列号生成，使用3次握手是因为3次握手已经准备好了传输可靠性以及数据顺序性所必要的信息，该握手的第3次实际上并不是需要单独传输的完全可以和数据一起传输。

TCP使用4次挥手拆除一条连接，为何需要4次呢？因为TCP是一个全双工协议，必须单独拆除每一条信道。注意，4次挥手和3次握手的意义同的，很多人都会问为何建立连接是3次握手，而拆除连接是4次挥手。**3次握手的目的很简单，就是分配资源，初始化序列号，这时还不涉及据传输，3次就足够做到这个了，而4次挥手的目的是终止数据传输，并回收资源，此时两个端点两个方向的序列号已经没有了任何关系，必待两方向都没有数据传输时才能拆除虚链路，不像初始化时那么简单，发现SYN标志就初始化一个序列号并确认SYN的序列号。因此必须单别在一个方向上终止该方向的数据传输。**

疑难杂症2：TIME_WAIT状态

为何要有这个状态，**原因很简单，那就是每次建立连接的时候序列号都是随机产生的，并且这个序列号是32位的，会回绕。**现在我来解释这TIME_WAIT有什么关系。

任何的TCP分段都要在尽力而为的IP网络上传输，中间的路由器可能会随意的缓存任何的IP数据报，它并不管这个IP数据报上被承载的是什么据，然而根据经验和互联网的大小，一个IP数据报最多存活MSL(这是根据地球表面积，电磁波在各种介质中的传输速率以及IP协议的TTL等推算出来的，如果在火星上，这个MSL会大得多...)。

现在我们考虑终止连接时的被动方发送了一个FIN，然后主动方回复了一个ACK，然而这个ACK可能会丢失，这会造成被动方重发FIN，这个可能会在互联网上存活MSL。

如果没有TIME_WAIT的话，假设连接1已经断开，然而其被动方最后重发的那个FIN(或者FIN之前发送的任何TCP分段)还在网络上，然而连接用了连接1的所有的5元素(源IP，目的IP，TCP，源端口，目的端口)，刚刚将建立好连接，连接1迟到的FIN到达了，这个FIN将以比较低但是可能的概率终止掉连接2.

为何说是概率比较低呢？这涉及到一个匹配问题，迟到的FIN分段的序列号必须落在连接2的一方的期望序列号范围之内。虽然这种巧合很少生，但确实会发生，毕竟初始序列号是随机产生了。因此终止连接的主动方必须在接受了被动方且回复了ACK之后等待2*MSL时间才能进入CLOSE状态，之所以乘以2是因为这是保守的算法，最坏情况下，针对被动方的ACK在以最长路线(经历一个MSL)经过互联网马上到达被动方失。

为了应对这个问题，RFC793对初始序列号的生成有个建议，那就是设定一个基准，在这个基准之上搞随机，这个基准就是时间，我们知道时单调递增的。然而这仍然有问题，那就是回绕问题，如果发生回绕，那么新的序列号将会落到一个很低的值。**因此最好的办法就是避开“重叠”，其含义就是基准之上的随机要设定一个范围。**

要知道，很多人很不喜欢看到服务器上出现大量的TIME_WAIT状态的连接，因此他们将TIME_WAIT的值设置的很低，这虽然在大多数情况下行，然而确实也是一种冒险行为。最好的方式就是，不要重用一个连接。

疑难杂症3：重用一个连接和重用一个套接字

这是根本不同的，单独重用一个套接字一般不会有任何问题，因为TCP是基于连接的。比如在服务器端出现了一个TIME_WAIT连接，那么该
标识了一个五元素，只要客户端不使用相同的源端口，连接服务器是没有问题的，因为迟到的FIN永远不会到达这个连接。记住，一个五元素
了一个连接，而不是一个套接字(当然，对于BSD套接字而言，服务端的accept套接字确实标识了一个连接)。

3.2.2.传输可靠性
基本上传输可靠性是靠确认号实现的，也就是说，每发送一个分段，接下来接收端必然要发送一个确认，发送端收到确认后才可以发送下一
节。这个原则最简单不过了，教科书上的“停止-等待”协议就是这个原则的字节版本，只是TCP使用了滑动窗口机制使得每次不一定发送一
节，但是这是后话，本节仅仅谈一下确认的超时机制。

怎么知道数据到达对端呢？那就是对端发送一个确认，但是如果一直收不到对端的确认，发送端等多久呢？如果一直等下去，那么将无法发
据的丢失，协议将不可用，如果等待时间过短，可能确认还在路上，因此等待时间是个问题，另外如何去管理这个超时时间也是一个问题。

疑难杂症4：超时时间的计算

绝对不能随意去揣测超时的时间，而应该给出一个精确的算法去计算。毫无疑问，一个TCP分段的回复到达的时间就是一个数据报往返的时
因此标准定义了一个新的名词RTT，代表一个TCP分段的往返时间。然而我们知道，IP网络是尽力而为的，并且路由是动态的，且路由器会
兆的缓存或者丢弃任何的数据报，因此这个RTT是需要动态测量的，也就是说起码每隔一段时间就要测量一次，如果每次都一样，万事大吉
而世界并非如你所愿，因此我们需要找到的恰恰的一个“平均值”，而不是一个准确值。

这个平均值如果仅仅直接通过计算多次测量值取算术平均，那是不恰当的，因为对于数据传输延时，我们必须考虑的路径延迟的瞬间抖动，
如果两次测量值分别为2和98，那么超时值将是50，这个值对于2而言，太大了，结果造成了数据的延迟过大(本该重传的等待了好久才重传)
而对于98而言，太小了，结果造成了过度重传(路途遥远，本该很慢，结果大量重传已经正确确认但是迟到的TCP分段)。

因此，**除了考虑每两次测量值的偏差之外，其变化率也应该考虑在内，如果变化率过大，则通过以变化率为自变量的函数为主计算RTT(如果
增大，则取值为比较大的正数，如果陡然减小，则取值为比较小的负数，然后和平均值加权求和)，反之如果变化率很小，则取测量平均值。**
不言而喻的，这个算法至今仍然工作的很好。

疑难杂症5：超时计时器的管理-每连接单一计时器

很显然，对每一个TCP分段都生成一个计时器是最直接的方式，每个计时器在RTT时间后到期，如果没有收到确认，则重传。然而这只是理论
合理，对于大多数操作系统而言，这将带来巨大的内存开销和调度开销，因此采取每一个TCP连接单一计时器的设计则成了一个默认的选择。
是单一的计时器怎么管理如此多的发出去的TCP分段呢？又该如何来设计单一的计时器呢。

设计单一计时器有两个原则：1.每一个报文在长期收不到确认都必须可以超时；2.这个长期收不到中长期不能和测量的RTT相隔太远。因此
RFC2988定义一套很简单的原则：

> **a.发送TCP分段时，如果还没有重传定时器开启，那么开启它。**
> **b.发送TCP分段时，如果已经有重传定时器开启，不再开启它。**
> **c.收到一个非冗余ACK时，如果有数据在传输中，重新开启重传定时器。**
> **d.收到一个非冗余ACK时，如果没有数据在传输中，则关闭重传定时器。**

我们看看这4条规则是如何做到以上两点的，根据a和c(在c中，注意到ACK是非冗余的)，任何TCP分段只要不被确认，超时定时器总会超时的
而为何需要c呢？只有规则a存在的话，也可以做到原则1。实际上确实是这样的，但是为了不会出现过早重传，才添加了规则c，如果没有规
那么万一在重传定时器到期前，发送了一些数据，这样在定时器到期后，除了很早发送的数据能收到ACK外，其它稍晚些发送的数据的ACK
不会到来，因此这些数据都将被重传。有了规则c之后，只要有分段ACK到来，则重置重传定时器，这很合理，因此大多数正常情况下，从数
发出到ACK的到来这段时间以及计算得到的RTT以及重传定时器超时的时间这三者相差并不大，一个ACK到来后重置定时器可以保护后发的数
不被过早重传。

这里面还有一些细节需要说明。一个ACK到来了，说明后续的ACK很可能会依次到来，也就是说丢失的可能性并不大，另外，即使真的有后
TCP分段丢失现象发生，也会在最多2倍定时器超时时间的范围内被重传(假设该报文是第一个报文发出启动定时器之后马上发出的，丢失了
一个报文的ACK到来后又重启了定时器，又经过了一个超时时间才会被重传)。虽然这里还没有涉及拥塞控制，但是可见网络拥塞会引起丢包
包会引起重传，过度重传反过来加重网络拥塞，设置规则c的结果可以缓解过多的重传，毕竟将启动定时器之后发送的数据的重传超时时间拉
最多一倍左右。最多一倍左右的超时偏差做到了原则2，即“这个长期收不到中长期不能和测量的RTT相隔太远”。

还有一点，如果是一个发送序列的最后一个分段丢失了，后面就不会收到冗余ACK，这样就只能等到超时了，并且超时时间几乎是肯定会比
器超时时间更长。如果这个分段是在发送序列的靠后的时间发送的且和前面的发送时间相隔时间较远，则其超时时间不会很大，反之就会比
大。

疑难杂症6：何时测量RTT

目前很多TCP实现了时间戳，这样就方便多了，发送端再也不需要保存发送分段的时间了，只需要将其放入协议头的时间戳字段，然后接收端只需将其回显在ACK即可，然后发送端收到ACK后，取出时间戳，和当前时间做算术差，即可完成一次RTT的测量。

3.2.3.数据顺序性
基本上传输可靠性是靠序列号实现的。

疑难杂症7：确认号和超时重传

确认号是一个很诡异的东西，因为TCP的发送端对于发送出去的一个数据序列，它只要收到一个确认号就认为确认号前面的数据都被收到了，即使前面的某个确认号丢失了，也就是说，发送端只认最后一个确认号。这是合理的，因为确认号是接收端发出的，接收端只确认按序到达的一个TCP分段。

另外，发送端重发了一个TCP报文并且接收到该TCP分段的确认号，并不能说明这个重发的报文被接收了，也可能是数据早就被接收了，只是其ACK丢失或者其ACK延迟到达导致了超时。值得说明的是，接收端会丢弃任何重复的数据，即使丢弃了重复的数据，其ACK还是会照发不误的。

标准的早期TCP实现为，只要一个TCP分段丢失，即使后面的TCP分段都被完整收到，发送端还是会重传从丢失分段开始的所有报文，这就会产生一个问题，那就是重传风暴，一个分段丢失，引起大量的重传。这种风暴实则不必要的，因为大多数的TCP实现中，接收端已经缓存了乱序的分段，这些被重传的丢失分段之后的分段到达接收端之后，很大的可能性是被丢弃。关于这一点在拥塞控制被引入之后还会提及(问题先述为快)，后来报文丢失导致超时就说明网络很可能已然拥塞，重传风暴只能加重其拥塞程度)。

疑难杂症8：乱序数据缓存以及选择确认

TCP是保证数据顺序的，但是并不意味着它总是会丢弃乱序的TCP分段，具体会不会丢弃是和具体实现相关的，RFC建议如果内存允许，还是缓存这些乱序到来的分段，然后实现一种机制等到可以拼接成一个按序序列的时候将缓存的分段拼接，这就类似于IP协议中的分片一样，但是IP数据报是不确认的，因此IP协议的实现必须缓存收到的任何分片而不能将其丢弃，因为丢弃了一个IP分片，它就再也不会到来了。

现在，TCP实现了一种称为**选择确认**的方式，接收端会显式告诉发送端需要重传哪些分段而不需要重传哪些分段。这无疑避免了重传风暴。

疑难杂症9：TCP序列号的回绕的问题

TCP的序列号回绕会引起很多的问题，比如序列号为s的分段发出之后，m秒后，序列号比s小的序列号为j的分段发出，只不过此时的j比上一次绕了一圈，这就是回绕问题，那么如果这后一个分段到达接收端，这就会引发彻底乱序-本来j该在s后面，结果反而到达前面了，这种乱序是TCP协议检查不出来的。我们仔细想一下，这种情况确实会发生，数据分段并不是一个字节一个字节发送出去的，如果存在一个速率为1Gbps的网络，TCP发送端1秒会发送125MB的数据，32位的序列号空间能传输2的32次方个字节，也就是说32秒左右就会发生回绕，我们知道这个值远小于MSL值，因此会发生的。

有个细节可能会引起误会，那就是TCP的窗口大小空间是序列号空间的一半，这样恰好在满载情况下，数据能填满发送窗口和接收窗口，序列号空间正好够用。然而事实上，TCP的初始序列号并不是从0开始的，而是随机产生的(当然要辅助一些更精妙的算法)，因此如果初始序列号比较接近2的32次方，那么很快就会回绕。

当然，如今可以用时间戳选项来辅助作为序列号的一个识别的部分，接收端遇到回绕的情况，需要比较时间戳，我们知道，时间戳是单调递增的，虽然也会回绕，然而回绕时间却要长很多。这只是一种策略，在此不详谈。还有一个很现实的问题，理论上序列号会回绕，但是实际上多少TCP的端点主机直接架设在1G的网络线缆两端并且接收方和发送方的窗口还能恰好被同时填满。另外，就算发生了回绕，也不是一件特别事情，回绕在计算机里面太常见了，只需要能识别出来即可解决，对于TCP的序列号而言，在高速网络(点对点网络或者以太网)的两端，数据乱序的可能性很小，因此当收到一个序列号突然变为0或者终止序列号小于起始序列号的情况后，很容易辨别出来，只需要和前一个确认的分段比较即可，如果在一个经过路由器的网络两端，会引发IP数据报的顺序重排，对于TCP而言，虽然还会发生回绕，也会慢得多，且考虑到拥塞窗口(目前还没有引入)一般不会太大，窗口也很难被填满到65536。

3.2.4.端到端的流量控制
端到端的流量控制使用滑动窗口来实现。滑动窗口的原理非常简单，基本就是一个生产者/消费者模型

疑难杂症10：流量控制的真实意义

很多人以为流量控制会很有效的协调两端的流量匹配，确实是这样，但是如果你考虑到网络的利用率问题，TCP的流量控制机制就不那么完美

了，造成这种局面的原因在于，滑动窗口只是限制了最大发送的数据，却没有限制最小发送的数据，结果导致一些很小的数据被封装成TCP分段，报文协议头所占的比例过于大，造成网络利用率下降，这就引出了接下来的内容，那就是端到端意义的TCP协议效率。

~~~~~~~~~~~~~~~~~~~~
承上启下
终于到了阐述问题的时候了，以上的TCP协议实现的非常简单，这也是TCP的标准实现，然而很快我们就会发现各种各样的问题。这些问题导致了标准化协会对TCP协议进行了大量的修补，这些修补杂糅在一起让人们有些云里雾里，不知所措。本文档就旨在分离这些杂乱的情况，实际上，根据RFC，这些杂乱的情况都是可以找到其单独的发展轨迹的。
~~~~~~~~~~~~~~~~~~~~

4.端到端意义上的TCP协议效率

4.1.三个问题以及解决

问题1描述：接收端处理慢，导致接收窗口被填满

这明显是速率不匹配引发的问题，然而即使速率不匹配，只要滑动窗口能协调好它们的速率就好，要快都快，要慢都慢，事实上滑动窗口在这点上做的很好。但是如果我们不得不从效率上来考虑问题的话，事实就不那么乐观了。考虑此时接收窗口已然被填满，慢速的应用程序慢腾腾读取了一个字节，空出一个位置，然后通告给TCP的发送端，发送端得知空出一个位置，马上发出一个字节，又将接收端填满，然后接收应用程序又一次慢腾腾...这就是糊涂窗口综合症，一个大多数人都很熟悉的词。这个问题极大的浪费了网络带宽，降低了网络利用率。好比从大同拉一吨煤到北京需要一辆车，拉1Kg煤到北京也需要一辆车(超级夸张的一个例子，请不要相信)，但是一辆车开到北京的开销是一定的...

问题1解决：窗口通告

对于问题1，很显然问题出在接收端，我们没有办法限制发送端不发送小分段，但是却可以限制接收端通告小窗口，这是合理的，这并不影响程序，此时经典的延迟/吞吐量反比律将不再适用，因为接收窗口是满的，其空出一半空间表示还有一半空间有数据没有被应用读取，和其空出一个字节的空间的效果是一样的，因此可以限制接收端当窗口为0时，直接通告给发送端以阻止其继续发送数据，只有当其接收窗口再次达到MSS一半大小的时候才通告一个不为0的窗口，此前对于所有的发送端的窗口probe分段(用于探测接收端窗口大小的probe分段，由TCP标准规定)全部通告窗口为0，这样发送端在收到窗口不为0的通告，那么肯定是一个比较大的窗口，因此发送端可以一次性发出一个很大的TCP分段，大量数据，也即拉了好几十吨的煤到北京，而不是只拉了几公斤。即，限制窗口通告时机，解决糊涂窗口综合症

问题2描述：发送端持续发送小包，导致窗口闲置

这明显是发送端引起的问题，此时接收端的窗口开得很大，然而发送端却不积累数据，还是一味的发送小块数据分段。只要发送了任和的分段，接收端都要无条件接收并且确认，这完全符合TCP规范，因此必然要限制发送端不发送这样的小分段。

问题2解决：Nagle算法

Nagel算法很简单，标准的Nagle算法为：

```
IF 数据的大小和窗口的大小都超过了MSS
   Then 发送数据分段
ELSE
   IF 还有发出的TCP分段的确认没有到来
      Then 积累数据到发送队列的末尾的TCP分段
   ELSE
      发送数据分段
   EndIF
EndIF
```

可是后来，这个算法变了，变得更加灵活了，其中的：

IF 还有发出的TCP分段的确认没有到来

变成了

IF 还有发出的不足MSS大小的TCP分段的确认没有到来

这样如果发出了一个MSS大小的分段还没有被确认，后面也是可以随时发送一个小分段的，这个改进降低了算法对延迟时间的影响。这个算法体现了一种自适应的策略，越是确认的快，越是发送的快，虽然Nagle算法看起来在积累数据增加吞吐量的同时也加大的时延，可事实上，如果对于类似交互式的应用，时延并不会增加，因为这类应用回复数据也是很快的，比如Telnet之类的服务必然需要回显字符，因此能和对端进行自适应协调。

注意，Nagle算法是默认开启的，但是却可以关闭。如果在开启的情况下，那么它就严格按照上述的算法来执行。

问题3.确认号(ACK)本身就是不含数据的分段，因此大量的确认号消耗了大量的带宽

这是TCP为了确保可靠性传输的规范，然而大多数情况下，ACK还是可以和数据一起捎带传输的。如果没有捎带传输，那么就只能单独回来一

ACK，如果这样的分段太多，网络的利用率就会下降。从大同用火车拉到北京100吨煤，为了确认煤已收到，北京需要派一辆同样的火车空载到大同去复命，因为没有别的交通工具，只有火车。如果这位复命者刚开着一列火车走，又从大同来了一车煤，这拉煤的哥们儿又要开一列火车去复命了。

问题3的解决：

RFC建议了一种延迟的ACK，也就是说，ACK在收到数据后并不马上回复，而是延迟一段可以接受的时间，延迟一段时间的目的是看能不能和接收方要发给发送方的数据一起回去，因为TCP协议头中总是包含确认号的，如果能的话，就将ACK一起捎带回去，这样网络利用率就提高了。大同复命的确认者不必开一辆空载火车回大同了，此时北京正好有一批货物要送往大同，这位复命者搭着这批货的火车返回大同。

如果等了一段可以接受的时间，还是没有数据要发往发送端，此时就需要单独发送一个ACK了，然而即使如此，这个延迟的ACK虽然没有可以被捎带的数据分段，也可能等到了后续到来的TCP分段，这样它们就可以取最大者一起返回了，要知道，TCP的确认号是收到的按序报文后一个字节的后一个字节。最后，RFC建议，延迟的ACK最多等待两个分段的积累确认。

4.2.分析三个问题之间的关联

三个问题导致的结果是相同的，但是要知道它们的原因本质上是不同的，问题1几乎总是出现在接收端窗口满的情况下，而问题2几乎总是发生在窗口闲置的情况下，问题3看起来是最无聊的，然而由于TCP的要求，必须要有确认号，而且一个确认号就需要一个TCP分段，这个分段不含数据，无疑是很小的。

三个问题都导致了网络利用率的降低。虽然两个问题导致了同样的结果，但是必须认识到它们是不同的问题，很自然的将这些问题的解决方案总在一起，形成一个全局的解决方案，这就是如今的操作系统中的解决方案。

4.3.问题的杂糅情况

疑难杂症11：糊涂窗口解决方案和Nagle算法

糊涂窗口综合症患者希望发送端积累TCP分段，而Nagle算法确实保证了一定的TCP分段在发送端的积累，另外在延迟ACK的延迟的那一会时间发送端会利用这段时间积累数据。然而这却是三个不同的问题。Nagle算法可以缓解糊涂窗口综合症，却不是治本的良药。

疑难杂症12：Nagle算法和延迟ACK

延迟ACK会延长ACK到达发送端的时间，由于标准Nagle算法只允许一个未被确认的TCP分段，那无疑在接收端，这个延迟的ACK是毫无希望后续数据到来最终进行积累确认的，如果没有数据可以捎带这个ACK，那么这个ACK只有在延迟确认定时器超时的时候才会发出，这样在等这个ACK的过程中，发送端又积累了一些数据，因此延迟ACK实际上是在增加延迟的代价下加强了Nagle算法。在延迟ACK加Nagle算法的情况下接收端只有不断有数据要发回，才能同时既保证了发送端的分段积累，又保证了延迟不增加，同时还没有或者很少有空载的ACK。

要知道，延迟ACK和Nagle是两个问题的解决方案。

疑难杂症13：到底何时可以发送数据

到底何时才能发送数据呢？如果单从Nagle算法上看，很简单，然而事实证明，情况还要更复杂些。如果发送端已经排列了3个TCP分段，分别1，分段2，分段3依次被排入，三个分段都是小分段(不符合Nagle算法中立即发送的标准)，此时已经有一个分段被发出了，且其确认还没有来，请问此时能发送分段1和2吗？如果按照Nagle算法，是不能发送的，但实际上它们是可以发送的，因为这两个分段已经没有任何机会再有新的数据了，新的数据肯定都积累在分段3上了。问题在于，分段还没有积累到一定大小时，怎么还可以产生新的分段？这是可能的，但这是另个问题，在此不谈。

Linux的TCP实现在这个问题上表现的更加灵活，它是这么判断能否发送的(在开启了Nagle的情况下)：

```
IF (没有超过拥塞窗口大小的数据分段未确认 || 数据分段中包含FIN ) &&
   数据分段没有超越窗口边界
   Then
   IF 分段在中间(上述例子中的分段1和2) ||
         分段是紧急模式          ||
      通过上述的Nagle算法(改进后的Nagle算法)
      Then 发送分段
   EndIF
EndIF
```

曾经我也改过Nagle算法，确切的说不是修改Nagle算法，而是修改了“到底何时能发送数据”的策略，以往都是发送端判断能否发送数据的，是如果此时有延迟ACK在等待被捎带，而待发送的数据又由于积累不够或者其它原因不能发送，因此两边都在等，这其实在某些情况下不是好。我所做的改进中对待何时能发送数据又增加了一种情况，这就是“ACK拉”的情况，一旦有延迟ACK等待发送，判断一下有没有数据也待发送，如果有的话，看看数据是否大到了一定程度，在此，我选择的是MSS的一半：

IF (没有超过拥塞窗口大小的数据分段未确认 || 数据分段中包含FIN) &&
  数据分段没有超越窗口边界
  Then
  IF 分段在中间(上述例子中的分段1和2) ||
      分段是紧急模式 ||
    通过上述的Nagle算法(改进后的Nagle算法)
    Then 发送分段
  EndIF
ELSE IF 有延迟ACK等待传输   &&
  发送队列中有待发送的TCP分段   &&
  发送队列的头分段大小大于MSS的一半
    Then 发送队列头分段且捎带延迟ACK
EndIF

另外，发送队列头分段的大小是可以在统计意义上动态计算的，也不一定非要是MSS大小的一半。我们发现，这种算法对于交互式网路应用适应的，你打字越快，特定时间内积累的分段就越长，对端回复的越快(可以捎带ACK)，本端发送的也就越快(以Echo举例会更好理解)。

疑难杂症14：《TCP/IP详解(卷一)》中Nagle算法的例子解读

这个问题在网上搜了很多的答案，有的说RFC的建议，有的说别的。可是实际上这就是一个典型的“竞态问题”：
首先服务器发了两个分段：
数据段12：ack 14
数据段13：ack 14，54:56
然后客户端发了两个分段：
数据段14：ack 54，14:17
数据段15：ack 56，17:18

可以看到数据段14本来应该确认56的，但是确认的却是54。也就是说，数据段已经移出队列将要发送但还未发送的时候，数据段13才到来断处理程序抢占了数据段14的发送进程，要知道此时只是把数据段14移出了队列，还没有更新任何的状态信息，比如“发出但未被确认的分量”，此时软中断处理程序顺利接收了分段13，然后更新窗口信息，并且检查看有没有数据要发送，由于分段14已经移出队列，下一个接受检查的就是分段15了，由于状态信息还没有更新，因此分段15顺利通过发送检测，发送完成。

可以看Linux的源代码了解相关信息，tcp_write_xmit这个函数在两个地方会被调用，一个是TCP的发送进程中，另一个就是软中断的接收处中，两者在调用中的竞态就会引起《详解》中的那种情况。注意，这种不加锁的发送方式是合理的，也是最高效的，因此TCP的处理语义会判断，丢弃一切不该接收或者重复接收的分段的

~~~~~~~~~~~~~~~~~~
承上启下
又到了该承上启下，到此为止，我们叙述的TCP还都是简单的TCP，就算是简单的TCP，也存在上述的诸多问题，就更别提继续增加TCP的复杂性了。到此为止，我们的TCP都是端到端意义上的，然而实际上TCP要跑在IP网络之上的，而IP网络的问题是很多的，是一个很拥堵网络。不幸的是，TCP的有些关于确认和可靠性的机制还会加重IP网络的拥堵。
~~~~~~~~~~~~~~~~~~~~

5.IP网络之上的TCP

5.1.端到端的TCP协议和IP协议之间的矛盾

端到端的TCP只能看到两个节点，那就是自己和对方，它们是看不到任何中间的路径的。可是IP网络却是一跳一跳的，它们的矛盾之处在于端到端流量控制必然会导致网络拥堵。因为每条TCP连接的一端只知道它对端还有多少空间用于接收数据，它们并不管到达对端的路径上是有这么大的容量，事实上所有连接的这些空间加在一起将瞬间超过IP网络的容量，因此TCP也不可能按照滑动窗口流量控制机制很理想的运

势必需要一种拥塞控制机制，反应路径的拥塞情况。

疑难杂症15：拥塞控制的本质

由于TCP是端到端协议，因此两端之间的控制范畴属于流量控制，IP网络的拥塞会导致TCP分段的丢失，由于TCP看不到中间的路由器，因此丢失只会发生中间路由器，当然两个端点的网卡或者IP层丢掉数据分段也是TCP看不到的。因此拥塞控制必然作用于IP链路。事实上我们可以知，只有在以下情况下拥塞控制才会起作用：

a.两个或两个以上的连接(其中一个一定要是TCP，另一个可以是任意连接)经过同一个路由器或者同一个链路时；

b.只有一个TCP连接，然而它经过了一个路由器时。

其它情况下是不会拥塞的。因为一个TCP总是希望独享整条网络通路，而这对于多个连接而言是不可能的，必须保证TCP的公平性，这样这种控制机制才合理。本质上，拥塞的原因就是大家都想独享全部带宽资源，结果导致拥塞，这也是合理的，毕竟TCP看不到网络的状态，同时决定了TCP的拥塞控制必须采用试探性的方式，最终到达一个足以引起其“反应”的“刺激点”。

拥塞控制需要完成以下两个任务：1.公平性；2.拥塞之后退出拥塞状态。

疑难杂症16：影响拥塞的因素

我们必须认识到拥塞控制是一个整体的机制，它不偏向于任何TCP连接，因此这个机制内在的就包含了公平性。那么影响拥塞的因素都有什么呢？具有讽刺意味的是，起初TCP并没有拥塞控制机制，正是TCP的超时重传风暴(一个分段丢失造成后续的已经发送的分段均被重传，而这些重传大多数是不必要的)加重了网络的拥塞。因此重传必然不能过频，必须把重传定时器的超时时间设置的稍微长一些，而这一点在单一重传定时器的设计中得到了加强。除此TCP自身的因素之外，其它所有的拥塞都可以靠拥塞控制机制来自动完成。

另外，不要把路由器想成一种线速转发设备，再好的路由器只要接入网络，总是会拉低网络的总带宽，因此即使只有一个TCP连接，由于TCP发送方总是以发送链路的带宽发送分段，这些分段在经过路由器的时候排队和处理总是会有时延，因此最终肯定会丢包的。

最后，丢包的延后性也会加重拥塞。假设一个TCP连接经过了N个路由器，前N-1个路由器都能顺利转发TCP分段，但是最后一个路由器丢失一个分段，这就导致了这些丢失的分段浪费了前面路由器的大量带宽。

5.2.拥塞控制的策略

在介绍拥塞控制之前，首先介绍一下拥塞窗口，它实际上表示的也是“可以发送多少数据”，然而这个和接收端通告的接收窗口意义是不一样的，后者是流量控制用的窗口，而前者是拥塞控制用的窗口，体现了网络拥塞程度。

拥塞控制整体上分为两类，一类是试探性的拥塞探测，另一类则是拥塞避免(注意，不是常规意义上的拥塞避免)。

5.2.1.试探性的拥塞探测分为两类，之一是慢启动，之二是拥塞窗口加性扩大(也就是熟知的拥塞避免，然而这种方式是避免不了拥塞的)。
5.2.2.拥塞避免方式拥塞控制旨在还没有发生拥塞的时候就先提醒发送端，网络拥塞了，这样发送端就要么可以进入快速重传/快速恢复或者乘的减小拥塞窗口，这样就避免网络拥塞的一沓糊涂之后出现超时，从而进入慢启动阶段。
5.2.3.快速重传和快速恢复。所谓快速重传/快速恢复是针对慢启动的，我们知道慢启动要从1个MSS开始增加拥塞窗口，而快速重传/快速恢复是一旦收到3个冗余ACK，不必进入慢启动，而是将拥塞窗口缩小为当前阀值的一半加上3，然后如果继续收到冗余ACK，则将拥塞窗口加1个MSS，直到收到一个新的数据ACK，将窗口设置成正常的阀值，开始加性增加的阶段。
«
当进入快速重传时，为何要将拥塞窗口缩小为当前阀值的一半加上3呢？加上3是基于数据包守恒来说的，既然已经收到了3个冗余ACK，说明三个数据分段已经到达了接收端，既然三个分段已经离开了网络，那么就是说可以在发送3个分段了，只要再收到一个冗余ACK，这也说明1段已经离开了网络，因此就将拥塞窗口加1个MSS。直到收到新的ACK，说明直到收到第三个冗余ACK时期发送的TCP分段都已经到达对端了，此时进入正常阶段开始加性增加拥塞窗口。

疑难杂症17：超时重传和收到3个冗余ACK后重传

这两种重传的意义是不同的，超时重传一般是因为网络出现了严重拥塞(没有一个分段到达，如果有的话，肯定会有ACK的，若是正常ACK，置重传定时器，若是冗余ACK，则可能是个别报文丢失或者被重排序，若连续3个冗余ACK，则很有可能是个别分段丢失)，此时需要更加严缩小拥塞窗口，因此此时进入慢启动阶段。而收到3个冗余ACK后说明确实有中间的分段丢失，然而后面的分段确实到达了接收端，这因为这会发送冗余ACK，这一般是路由器故障或者轻度拥塞或者其它不太严重的原因引起的，因此此时拥塞窗口缩小的幅度就不能太大，此时进入重传/快速恢复阶段。

疑难杂症18：为何收到3个冗余ACK后才重传

这是一种权衡的结构，收到两个或者一个冗余ACK也可以重传，但是这样的话可能或造成不必要的重传，因为两个数据分段发生乱序的可能大，超过三个分段发生乱序的可能性才大，换句话说，如果仅仅收到一个乱序的分段，那很可能被中间路由器重排了，那么另一个分段很可上就到，然而如果连续收到了3个分段都没能弥补那个缺漏，那很可能是它丢失了，需要重传。**因此3个冗余ACK是一种权衡，在减少不必要和确实能检测出单个分段丢失之间所作的权衡。**

注意，冗余ACK是不能捎带的。

疑难杂症19：乘性减和加性增的深层含义

为什么是乘性减而加性增呢？拥塞窗口的增加受惠的只是自己，而拥塞窗口减少受益的大家，可是自己却受到了伤害。哪一点更重要呢？我
道TCP的拥塞控制中内置了公平性，恰恰就是这种乘性减实现了公平性。拥塞窗口的1个MSS的改变影响一个TCP发送者，为了使得自己拥塞
的减少影响更多的TCP发送者-让更多的发送者受益，那么采取了乘性减的策略。

当然，BIC算法提高了加性增的效率，不再一个一个MSS的加，而是一次加比较多的MSS，采取二分查找的方式逐步找到不丢包的点，然后
增。

疑难杂症20：TCP连接的传输稳定状态是什么

首先，先说一下发送端的发送窗口怎么确定，它取的是拥塞窗口和接收端通告窗口的最小值。然后，我们提出三种发送窗口的稳定状态：

a.IP互联网络上接收端拥有大窗口的经典锯齿状

b.IP互联网络上接收端拥有小窗口的直线状态

c.直连网络端点间的满载状态下的直线状态

其中a是大多数的状态，因为一般而言，TCP连接都是建立在互联网上的，而且是大量的，比如Web浏览，电子邮件，网络游戏，Ftp下载等
TCP发送端用慢启动或者拥塞避免方式不断增加其拥塞窗口，直到丢包的发生，然后进入慢启动或者拥塞避免阶段(要看是由于超时丢包还是
冗余ACK丢包)，此时发送窗口将下降到1或者下降一半，这种情况下，一般接收端的接收窗口是比较大的，毕竟IP网络并不是什么很快速的
络，一般的机器处理速度都很快。

但是如果接收端特别破，处理速度很慢，就会导致其通告一个很小的窗口，这样的话，即使拥塞窗口再大，发送端也还是以通告的接收窗口
送窗口，这样就不会发生拥塞。最后，如果唯一的TCP连接运行在一个直连的两台主机上，那么它将独享网络带宽，这样该TCP的数据流在
情况下将填满网络管道(我们把网络管道定义为带宽和延时的乘积)，其实在这种情况下是不存在拥塞的，就像你一个人独自徘徊在飘雨黄昏的
一样...

5.2.4.主动的拥塞避免

前面我们描述的拥塞控制方式都是试探性的检测，然后拥塞窗口被动的进行乘性减，这样在接收端窗口很大的情况下(一般都是这样，网络拥
分段就不会轻易到达接收端，导致接收端的窗口大量空置)就可能出现锯齿形状的“时间-窗口”图，类似在一个拥堵的北京X环上开车，发送
动，车开动，停止，等待，发动机发动，车开动...听声音也能听出来。

虽然TCP看不到下面的IP网络，然而它还是可以通过检测RTT的变化以及拥塞窗口的变化推算出IP网络的拥堵情况的。就比方说北京东四环一
递公司要持续送快递到西四环，当发件人发现货到时间越来越慢的时候，他会意识到“下班高峰期快到了”...

可以通过持续观测RTT的方式来主动调整拥塞窗口的大小而不是一味的加性增。然而还有更猛的算法，那就是计算两个差值的乘积：

(当前拥塞窗口-上一次拥塞窗口)x(当前的RTT-上一次的RTT)

如果结果是正数，则拥塞窗口减少1/8，若结果是负数或者0，则窗口增加一个MSS。注意，这回不再是乘性减了，可以看出，减的幅度比乘
幅度小，这是因为这种拥塞控制是主动的，而不是之前的那种被动的试探方式。在试探方式中，乘性减以一种惩罚的方式实现了公平性，而
里的主动方式中，当意识到要拥塞的时候，TCP发送者主动的减少了拥塞窗口，为了对这种自首行为进行鼓励，采用了小幅减少拥塞窗口的
式。需要注意的是，在拥塞窗口减小的过程中，乘积的前一个差值是负数，如果后一个差值也是负数，那么结果就是继续缩减窗口，直到拥
解或者窗口减少到了一定程度，使得后一个差值成了正数或者0，这种情况下，其实后一个差值只能变为0。

疑难杂症21：路由器和TCP的互动

虽然有了5.2.4节介绍的主动的拥塞检测，那么路由器能不能做点什么帮助检测拥塞呢？这种对路由器的扩展是必要的，要知道，每天有无数
TCP要通过路由器，虽然路由器不管TCP协议的任何事(当然排除连接跟踪之类的，这里所说的是标准的IP路由器)，但是它却能以一种很简单
式告诉TCP的两端IP网络发生了拥堵，**这种方式就是当路由器检测到自己发生轻微拥堵的时候随机的丢包，随机丢包而不是连续丢包对于TC
是有重大意义的，随机丢包会使TCP发现丢弃了个别的分段而后续的分段仍然会到达接收端，这样TCP发送端就会接收到3个冗余ACK，然后
快速重传/快速恢复而不是慢启动。**

这就是路由器能帮TCP做的事。

6.其它

疑难杂症22：如何学习TCP

很多人发帖问TCP相关的内容，接下来稀里哗啦的就是让看《TCP/IP详解》和《Unix网络编程》里面的特定章节，我觉得这种回答很不负责
因为我并不认为这两本书有多大的帮助，写得确实很不错，然而可以看出Richard Stevens是一个实用主义者，他喜欢用实例来解释一切，

解》通篇都是用tcpdump的输出来讲述的，这种方式只是适合于已经对TCP很理解的人，然而大多数的人是看不明白的。

如果想从设计的角度来说，这两本书都很烂。我觉得应该先看点入门的，比如Wiki之类的，然后看RFC文档,793，896，1122等)，这样你就
TCP为何这么设计了，而这些你永远都不能在Richard Stevens的书中得到。最后，如果你想，那么就看一点Richard Stevens的书，最重要
是写点代码或者敲点命令，然后抓包自己去分析。

疑难杂症23：Linux，Windows和网络编程

我觉得在Linux上写点TCP的代码是很不错的，如果有BSD那就更好了。不推荐用Winsock学习TCP。虽然微软声称自己的API都是为了让事
简单，但实际上事情却更复杂了，如果你用Winsock学习，你就要花大量的时候去掌握一些和网络编程无关但是windows平台上却少不了的

6.1.总结

TCP协议是一个端到端的协议，虽然话说它是一个带流量控制，拥塞控制的协议，然而正是因为这些所谓的控制才导致了TCP变得复杂。同时
特性是互相杂糅的，流量控制带来了很多问题，解决这些问题的方案最终又带来了新的问题，这些问题在解决的时候都只考虑了端到端的意
但实际上TCP需要尽力而为的IP提供的网络，因此拥塞成了最终的结症，拥塞控制算法的改进也成了一个单独的领域。

在学习TCP的过程中，切忌一锅粥一盘棋的方式，一定要分清楚每一个算法到底是解决什么问题的，每一个问题和其他问题到底有什么关联
些问题的解决方案之间有什么关联，另外TCP的发展历史也最好了解一下，这些都搞明白了，TCP协议就彻底被你掌控了。接下来你就可以学
Socket API了，然后高效的TCP程序出自你手！

转载自：笨不敌剑

相关阅读：
- OpenSSL HeartBleed漏洞原理漫画图解
- WEB常见问题排查（PDF）
- 网络性能调优–阿里云鸣嵩
- 一些LVS实验配置、工具和方案
- 利用nc工具高效创建TCP/UDP协议的连接