### TCP

TCP 详解（TCP机制）

https://blog.csdn.net/sinat\_36629696/article/details/80740678

TCP协议详解

<https://www.cnblogs.com/buxiangxin/p/8336022.html>

五分钟读懂TCP 协议——TCP协议简介

<https://blog.csdn.net/ningdaxing1994/article/details/73076795>

### 保证可靠性的机制

校验和

序列号(按序到达)

确认应答

[超时重传](#_TCP/IP协议--TCP的超时和重传)

连接管理

流量控制

[拥塞控制](#_TCP拥塞控制)

### 提高性能的机制

[滑动窗口](#_滑动窗口协议)

[快速重传](#_快重传和快恢复)

[延迟应答](#_网络学习-传输层TCP（延迟应答，捎带应答，粘包问题，TCP异常情况）)

[捎带应答](#_延迟应答)

### 定时器

什么是计时器呢？我们可以理解成一块闹钟，隔一段时间响一次，提醒TCP做特定的事情。TCP要正常工作，必须要有特定的计时器。那么TCP中有哪些计时器呢？

　　TCP中有四种计时器（Timer），分别为：

　　　　1.重传计时器：Retransmission Timer

　　　　2.坚持计时器：Persistent Timer

　　　　3.保活计时器：Keeplive Timer

　　　　4.时间等待计时器：Timer\_Wait Timer

　　（1）重传计时器

　　　　大家都知道TCP是保证数据可靠传输的。怎么保证呢？带确认的重传机制。在滑动窗口协议中，接受窗口会在连续收到的包序列中的最后一个包向接收端发送一个ACK，当网络拥堵的时候，发送端的数据包和接收端的ACK包都有可能丢失。TCP为了保证数据可靠传输，就规定在重传的“时间片”到了以后，如果还没有收到对方的ACK，就重发此包，以避免陷入无限等待中。

　　当TCP发送报文段时，就创建该特定报文的重传计时器。可能发生两种情况：

　　1.若在计时器截止时间到之前收到了对此特定报文段的确认，则撤销此计时器。

　　2.若在收到了对此特定报文段的确认之前计时器截止时间到，则重传此报文段，并将计时器复位。

　　（2）持久计时器

　　先来考虑一下情景：发送端向接收端发送数据包知道接受窗口填满了，然后接受窗口告诉发送方接受窗口填满了停止发送数据。此时的状态称为“零窗口”状态，发送端和接收端窗口大小均为0.直到接受TCP发送确认并宣布一个非零的窗口大小。但这个确认会丢失。我们知道TCP中，对确认是不需要发送确认的。若确认丢失了，接受TCP并不知道，而是会认为他已经完成了任务，并等待着发送TCP接着会发送更多的报文段。但发送TCP由于没有收到确认，就等待对方发送确认来通知窗口大小。双方的TCP都在永远的等待着对方。

　　要打开这种死锁，TCP为每一个链接使用一个持久计时器。当发送TCP收到窗口大小为0的确认时，就坚持启动计时器。当坚持计时器期限到时，发送TCP就发送一个特殊的报文段，叫做探测报文。这个报文段只有一个字节的数据。他有一个序号，但他的序号永远不需要确认；甚至在计算机对其他部分的数据的确认时该序号也被忽略。探测报文段提醒接受TCP：确认已丢失，必须重传。

　　坚持计时器的值设置为重传时间的数值。但是，若没有收到从接收端来的响应，则需发送另一个探测报文段，并将坚持计时器的值加倍和复位。发送端继续发送探测报文段，将坚持计时器设定的值加倍和复位，直到这个值增大到门限值（通常是60秒）为止。在这以后，发送端每个60秒就发送一个探测报文，直到窗口重新打开。

　　（3）保活计时器

　　　　保活计时器使用在某些实现中，用来防止在两个TCP之间的连接出现长时间的空闲。假定客户打开了到服务器的连接，传送了一些数据，然后就保持静默了。也许这个客户出故障了。在这种情况下，这个连接将永远的处理打开状态。

　　要解决这种问题，在大多数的实现中都是使服务器设置保活计时器。每当服务器收到客户的信息，就将计时器复位。通常设置为两小时。若服务器过了两小时还没有收到客户的信息，他就发送探测报文段。若发送了10个探测报文段（每一个像个75秒）还没有响应，就假定客户除了故障，因而就终止了该连接。

　　这种连接的断开当然不会使用四次握手，而是直接硬性的中断和客户端的TCP连接。

　　（4）时间等待计时器

时间等待计时器是在四次握手的时候使用的。四次握手的简单过程是这样的：假设客户端准备中断连接，首先向服务器端发送一个FIN的请求关闭包（FIN=final），然后由established过渡到FIN-WAIT1状态。服务器收到FIN包以后会发送一个ACK，然后自己有established进入CLOSE-WAIT.此时通信进入半双工状态，即留给服务器一个机会将剩余数据传递给客户端，传递完后服务器发送一个FIN+ACK的包，表示我已经发送完数据可以断开连接了，就这便进入LAST\_ACK阶段。客户端收到以后，发送一个ACK表示收到并同意请求，接着由FIN-WAIT2进入TIME-WAIT阶段。服务器收到ACK，结束连接。此时（即客户端发送完ACK包之后），客户端还要等待2MSL（MSL=maxinum segment lifetime最长报文生存时间，2MSL就是两倍的MSL）才能真正的关闭连接。

### [TCP/IP协议--TCP的超时和重传](https://www.cnblogs.com/firstForEver/p/5401755.html)

　　TCP是可靠传输。可靠之一体现在收到数据后，返回去一个确认。但是不能完全避免的是，数据和确认都可能丢失。解决这个办法就是，提供一个发送的重传定时器：如果定时器溢出时还没收到确认，它就重传这个报文段。

想法是完美的，关键之处在于超时和重传的策略，即怎么决定超时间隔和如何确定重传的频率。

书中举了一个简单的超时重传例子：

如图：

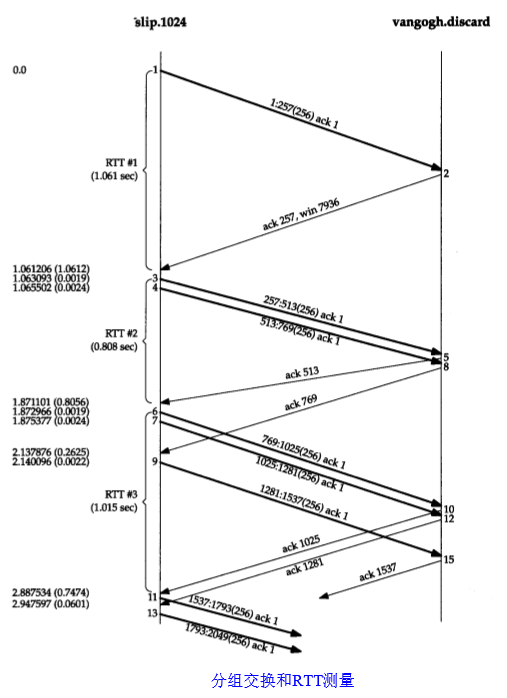


比如A往B传，传了一部分数据后，把B的网线拔了（前边讲过，如果不传数据的话，双方没法知道这个连接已经断了）。然后开始A再给B发数据，此时tcpdump出来发现，连续重传了一个报文段：时间间隔分别是，1.013s, 3, 6, 12, 24 和多个64s...最后发了个复位报文段表示我放弃了。。（从第一次开始发这个报文段，到最后发一个复位段的时间差大约是9分钟，这个9分钟一般在TCP实现中是不变的）。

往返时间测量：  
TCP的超时时间很大程度上是依赖报文段的往返时间。因此测量往返时间显得尤为重要。  
因为链路上的网络流量或者路由器等的存在，往返时间一般不会是一成不变的，可能会经常发生变化。  
最初的TCP规范这样协议：RTT(Round-Trip Time)表示往返时间，用M表示测量到的RTT。  
更新过的RTT = 0.9\*RTT + 0.1\*M 表示，我估计新的往返时间将是，0.9倍的之前的RTT + 0.1倍的新测量的RTT。（0.9叫平滑因子）  
得到了新的估计RTT，推荐的重传超时时间RTO（Retransmission TimeOut）的值应该设置为   
RTO = RTT\*b （这里的b是一个推荐值为2的时延离散因子）。超时时间就是大约2倍的往返时间。

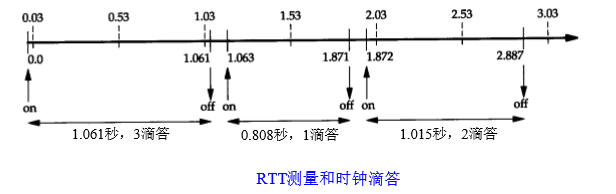
以上这个计算超时时间的方法存在缺陷，[Jacobson 1988] 作出了详细分析，当RTT变化范围比较大的时候，这个方法显得力不从心了，可能会引起不必要的重传。这样当网络负载比较高的时候，再重传会火上浇油...  
     这就又有了新的方法计算重传超时时间：  
前边说到如果RTT变化范围较大时，容易发生不必要重传。学过数学的都会知道，方差可以体现出波动大小。这个方法就是用到了方差来均衡下。  
这里有个公式用来计算RTO，懂的原理就好了，这个计算RTO的公式依赖于估计的RTT和均值偏差(逼近与标准差)，而最初的方法则使用了被平滑的RTT的一个倍数（b=2）。这块知道是这么个事就好~

往返时间RTT的测量：  
如图：

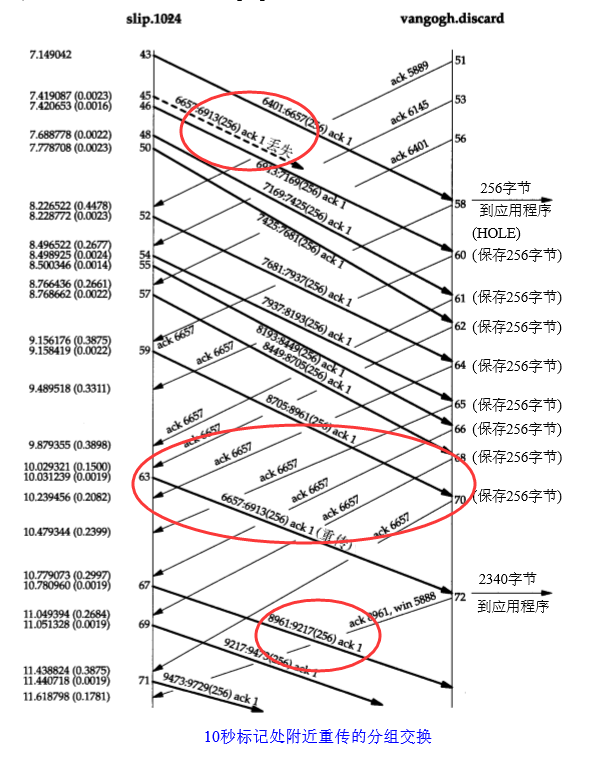


左边的时间轴上有三个括号，它们表明为进行RTT计算对哪些个报文段进行了计时，并不是所有的报文段都被计时。在发送一个报文段时，如果给定连接的重传定时器已经被使用，则该报文段不被计时。如图报文段4或者报文段7都没有参与计时。  
对每个连接而言，除了这个滴答计数器，报文段中数据的起始序号也被记录下来。当收到一个包含这个序号的确认后，该定时器就被关闭。如果ACK到达时数据没有被重传，则被平滑的RTT和被平滑的均值偏差将基于这个新测量进行更新。

在每次调用500 ms的TCP的定时器例程时，就增加一个计数器来完成计时。这意味着，如果一个报文段的确认在它发送550 ms后到达，则该报文段的往返时间RTT将是1个滴答（即500 ms）或是2个滴答(即1000 ms)。  
如图RTT测量和时钟滴答：



  -拥塞举例：  
主机slip总是通告窗口大小为4096，而主机vangogh则通告窗口为8192。  
如图：



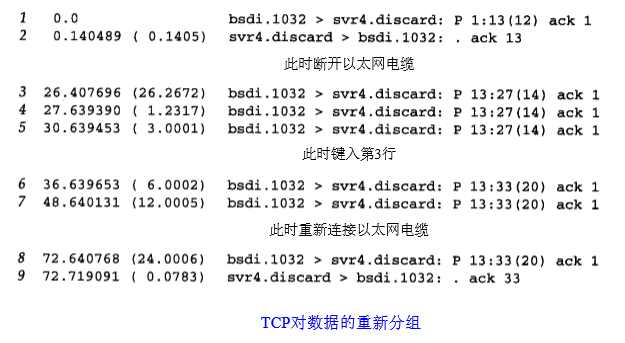
报文段45丢失了，报文段58是正常接收43的报文段给出的确认，然后接着接收主机连续发了8个ack 6657。可以看出是重发第三次（除了正常确认的中第3个）时，发送主机重传发送了63报文段。  
这收到第三个ack才重传也是算法中要求的，当收到第3个时，就假定一个报文段已经丢失并重传自那个序号起的一个报文段。这就是Jacobson的快速重传算法。  
值得注意的是，在重传后（报文段63），发送方继续正常的数据传输（报文段67、69和71）。TCP不需要等待对方确认重传。

这里再分析一下接收端是怎么处理的： 当按序收到正常数据（报文段43）后，接收TCP将255个字节的数据交给用户进程。但下一个收到的报文段（报文段46）是失序的（数据的开始序号 6913 并不是下一个期望的序号 6657）。TCP保存256字节的数据，并返回一个已成功接收数据的最大序号加1（6657）的ACK。被vangogh接收到的后面7个报文段（48, 50, 52, 54, 55, 57和59）也是失序的，接收方TCP保存这些数据并产生重复ACK（TCP实现没法告诉对方，我就缺某某个报文段，它只能告诉发送方我的确认序号一直是这个）。  
当缺少的报文段（报文段 63）到达时，接收方TCP在其接收缓存中组合好第6657~8960字节的数据，并将这2304字节的数据交给用户进程。所有这些数据在报文段72中进行确认。  
值得注意的是，此时该ACK通告窗口大小为5888（8192-2304，原来的通告窗口大小是8192），这是因为用户进程此时还没有读取出这些缓存中的字节。

  -拥塞避免：  
该算法假定由于分组受到损坏引起的丢失是非常少的（远小于1%），因此分组丢失就意味着在源主机和目的主机之间的某处网络上发生了拥塞。  
有两种分组丢失的指示：发生超时和接收到重复的确认（如果使用超时作为拥塞指示，则需要使用一个好的RTT算法）。  
前边讲过慢启动，拥塞避免算法和慢启动算法是两个目的不同、独立的算法。但是当拥塞发生时，我们希望降低分组进入网络的传输速率，于是可以调用慢启动来作到这一点。在实际中这两个算法通常在一起实现。

  -快速重传和快速恢复算法：  
在前边拥塞举例时，观察到第三个ack过来，发送端才进行重传。这是因为：由于我们不知道一个重复的ACK是由一个丢失的报文段引起的，还是由于仅仅出现了几个报文段的重新排序，因此我们等待少量重复的ACK到来。假如这只是一些报文段的重新排序，则在重新排序的报文段被处理并产生一个新的ACK之前，只可能产生1 ~ 2个重复的ACK。 如果一连串收到3个或3个以上的重复ACK，就非常可能是一个报文段丢失了。于是我们就重传丢失的数据报文段，而无需等待超时定时器溢出。这就是快速重传算法。  
接下来收到重传的ACK以前，发送了3个新的数据的报文段（报文段67,69和71）。执行的不是慢启动算法而是拥塞避免算法。这就是快速恢复算法。在这种情况下没有执行慢启动的原因是由于收到重复的ACK不仅仅告诉我们一个分组丢失了，而是在收发两端之间仍然有流动的数据（由于接收方只有在收到另一个报文段时才会产生重复的ACK，而该报文段已经离开了网络并进入了接收方的缓存），因此我们不想执行慢启动来突然减少数据流。

重新分组：  
当TCP超时并重传时，它不一定要重传同样的报文段。而是TCP允许进行重新分组而发送一个较大的报文段，这将有助于提高性能（当然，这个较大的报文段不能够超过接收方声明的MSS）。  
如图：



第3个发送前，断开网线。开始发送3，此时发生了重传，在放弃连接前，又键入了几个字节。然后插上网线，发现第8行，是把前边两次的分组组装成了一个分组发过去的。

### TCP拥塞控制

　　拥塞控制是一个全局性的过程； 流量控制是点对点通信量的控制

　　TCP拥塞控制4个核心算法：慢开始（slow start）、拥塞避免（Congestion Avoidance）、快速重传（fast retransmit）、快速回复（fast recovery）

　　拥塞窗口（cwnd，congestion window），其大小取决于网络的拥塞程度，并且动态地在变化

　　慢开始算法的思路就是，不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。

为了防止cwnd增长过大引起网络拥塞，还需设置一个慢开始门限ssthresh状态变量。ssthresh的用法如下：

当cwnd < ssthresh时，使用慢开始算法。

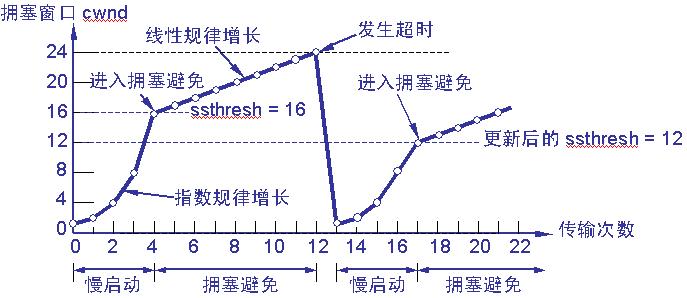
当cwnd > ssthresh时，改用拥塞避免算法。

当cwnd = ssthresh时，慢开始与拥塞避免算法任意。

拥塞避免算法让拥塞窗口缓慢增长，即每经过一个往返时间RTT就把发送发的拥塞窗口cwnd加1，而不是加倍。

无论是在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞，就把慢开始门限设置为出现拥塞时的发送窗口大小的一半。然后把拥塞窗口设置为1，执行慢开始算法。如下图：

拥塞控制的具体过程如下：



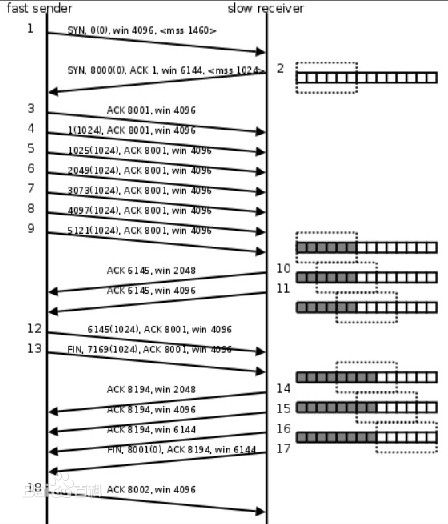
（1）TCP连接初始化，将拥塞窗口设置为1

（2）执行慢开始算法，cwnd按指数规律增长，直到cwnd=ssthresh时，开始执行拥塞避免算法，cwnd按线性规律增长

（3）当网络发生拥塞，把ssthresh值更新为拥塞前ssthresh值的一半，cwnd重新设置为1，按照步骤（2）执行

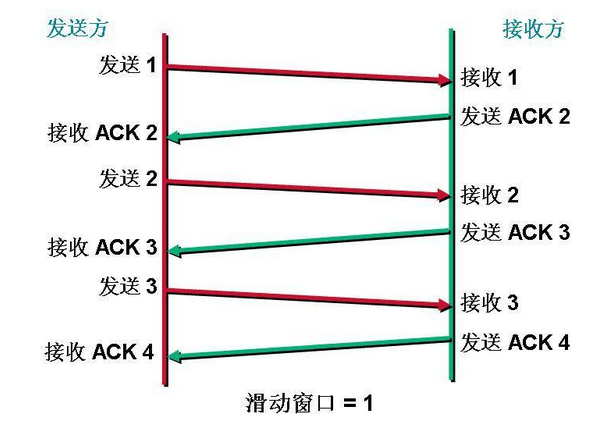
### 滑动窗口协议

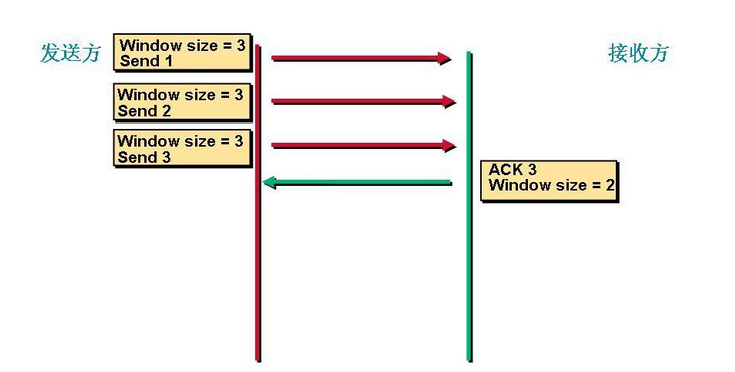
滑动窗口协议（Sliding Window Protocol），属于[TCP协议](https://baike.baidu.com/item/TCP%E5%8D%8F%E8%AE%AE)的一种应用，用于网络数据传输时的流量控制，以避免拥塞的发生。该协议允许发送方在停止并等待确认前发送多个数据分组。由于发送方不必每发一个分组就停下来等待确认，因此该协议可以加速数据的传输，提高网络吞吐量。

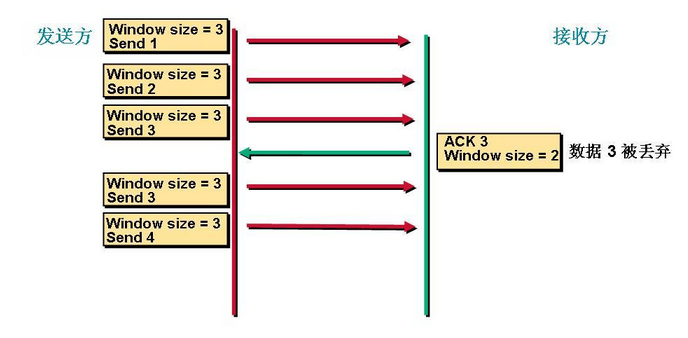


TCP的滑动窗口机制   
      TCP这个协议是网络中使用的比较广泛，他是一个面向连接的可靠的传输协议。既然是一个可靠的传输协议就需要对数据进行确认。TCP协议里窗口机制有2种：一种是固定的窗口大小；一种是滑动的窗口。这个窗口大小就是我们一次传输几个数据。对所有数据帧按顺序赋予编号，发送方在发送过程中始终保持着一个发送窗口，只有落在发送窗口内的帧才允许被发送；同时接收方也维持着一个接收窗口，只有落在接收窗口内的帧才允许接收。这样通过调整发送方窗口和接收方窗口的大小可以实现流量控制。

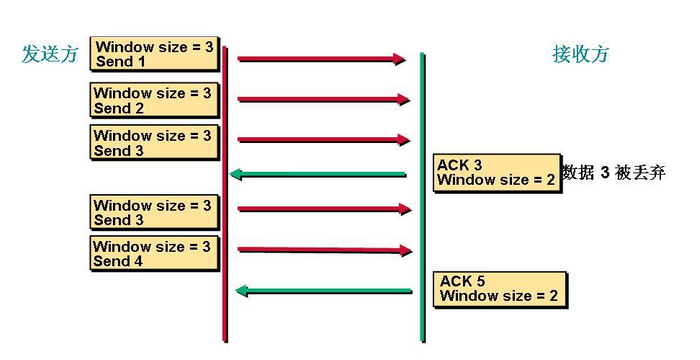
      TCP滑动窗口技术通过动态改变窗口大小来调节两台主机间数据传输。每个TCP/IP主机支持全双工数据传输，因此TCP有两个滑动窗口：一个用于接收数据，另一个用于发送数据。TCP使用肯定确认技术，其确认号指的是下一个所期待的字节。假定发送方设备以每一次三个数据包的方式发送数据，也就是说，窗口大小为3。发送方发送序列号为1、2、3的三个数据包，接收方设备成功接收数据包，用序列号4确认。发送方设备收到确认，继续以窗口大小3发送数据。当接收方设备要求降低或者增大网络流量时，可以对窗口大小进行减小或者增加，本例降低窗口大小为2，每一次发送两个数据包。当接收方设备要求窗口大小为0，表明接收方已经接收了全部数据，或者接收方应用程序没有时间读取数据，要求暂停发送。发送方接收到携带窗口号为0的确认，停止这一方向的数据传输。

我们可以看下面一张图来分析一下固定窗口大小有什么问题。   
   
这里我们可以看到假设窗口的大小是1，也是就每次只能发送一个数据只有接受方对这个数据进行确认了以后才能发送第2个数据。我们可以看到发送方每发送一个数据接受方就要给发送方一个ACK对这个数据进行确认。只有接受到了这个确认数据以后发送方才能传输下个数据。 这样我们考虑一下如果说窗口过小，那么当传输比较大的数据的时候需要不停的对数据进行确认，这个时候就会造成很大的延迟。如果说窗口的大小定义的过大。我们假设发送方一次发送100个数据。但是接收方只能处理50个数据。这样每次都会只对这50个数据进行确认。发送方下一次还是发送100个数据，但是接受方还是只能处理50个数据。这样就避免了不必要的数据来拥塞我们的链路。所以我们就引入了滑动窗口机制，窗口的大小并不是固定的而是根据我们之间的链路的带宽的大小，这个时候链路是否拥护塞。接受方是否能处理这么多数据了。    
我们看看滑动窗口是如何工作的。我们看下面几张图。



首先是**第一次发送数据这个时候的窗口大小是根据链路带宽的大小来决定的**。我们假设这个时候窗口的大小是3。这个时候接受方收到数据以后会对数据进行确认告诉发送方我下次希望手到的是数据是多少。这里我们看到**接收方发送的ACK=3(这是发送方发送序列2的回答确认，下一次接收方期望接收到的是3序列信号)**。这个时候发送方收到这个数据以后就知道我第一次发送的3个数据对方**只收到了2个**。就知道第3个数据对方没有收到。下次在发送的时候就从第3个数据开始发。这个时候**窗口大小就变成了2**。   
 

这个时候发送方发送2个数据。



看到接收方发送的ACK是5就表示他下一次希望收到的数据是5，发送方就知道我刚才发送的2个数据对方收了这个时候开始发送第5个数据。   
这就是滑动窗口的工作机制，当链路变好了或者变差了这个窗口还会发生变话，并不是第一次协商好了以后就永远不变了。                   
滑动窗口协议   
滑动窗口协议，是TCP使用的一种流量控制方法。该协议允许发送方在停止并等待确认前可以连续发送多个分组。由于发送方不必每发一个分组就停下来等待确认，因此该协议可以加速数据的传输。   
只有在接收窗口向前滑动时（与此同时也发送了确认），发送窗口才有可能向前滑动。      
收发两端的窗口按照以上规律不断地向前滑动，因此这种协议又称为滑动窗口协议。      
当发送窗口和接收窗口的大小都等于1时，就是停止等待协议。

### 快重传和快恢复

　　快速重传(Fast retransmit)要求接收方在收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方），而不要等到自己发送数据时捎带确认。

　　快重传算法规定，发送方只要一连收到3个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计数器时间到期。

　　快速恢复(Fast Recovery)

　　（1）当发送方连续收到三个重复确认，就执行“乘法减小”算法，把慢开始门限ssthresh减半。这是为了预防网络发生拥塞。请注意：接下去不执行慢开始算法。

（2）由于发送方现在认为网络很可能没有发生拥塞，因此与慢开始不同之处是现在不执行慢开始算法（即拥塞窗口cwnd现在不设置为1），而是把cwnd值设置为慢开始门限ssthresh减半后的数值，然后开始执行拥塞避免算法（“加法增大”），使拥塞窗口缓慢地线性增大。

### 网络学习-传输层TCP（延迟应答，捎带应答，粘包问题，TCP异常情况）

2018年05月24日 23:11:22 [zgege](https://me.csdn.net/zgege) 阅读数：411

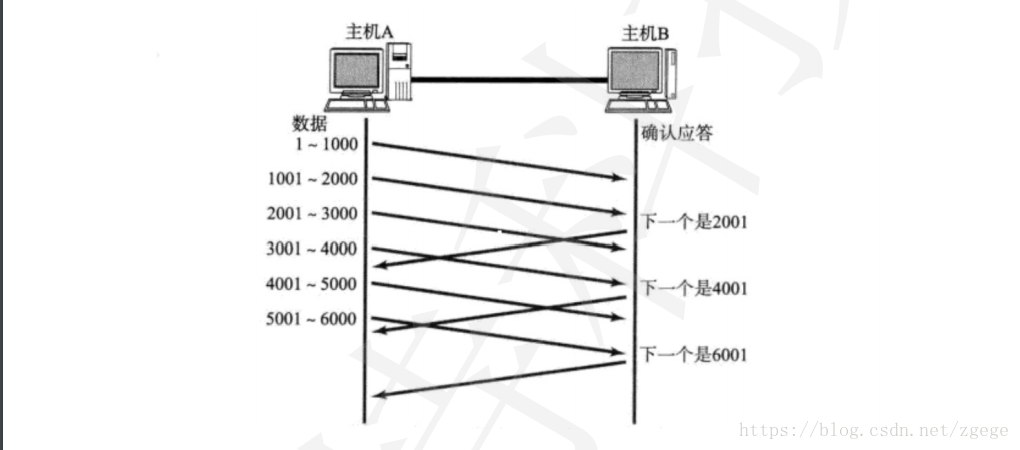
版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/zgege/article/details/80445372

| **内容** | **关键字** |
| --- | --- |
| 捎带应答 | 顺风车 |
| 延迟应答 | 窗口调整，超时时间，数据包限制N=2 |
| 粘包问题 | 面向字节流，边界，变长包，定长包，UDP粘包？ |
| TCP异常情况 | 进程终止，机器重启，机器断电/网线断开，保活定时器 |

#### ****延迟应答****

如果接收数据的主机立刻返回ACK应答, 这时候返回的窗口可能   
比较小.   
假设接收端缓冲区为1M. 一次收到了500K的数据; 如果立刻应答, 返回的窗口就是500K;但实际上可能处理端处理的速度很快, 10ms之内就把500K数据从缓冲区消费掉了;在这种情况下, 接收端处理还远没有达到自己的极限, 即使窗口再放大一些, 也能处理过来;如果接收端稍微等一会再应答, 比如等待200ms再应答, 那么这个时候返回的窗口大小就是1M;

一定要记得, 窗口越大, 网络吞吐量就越大, 传输效率就越高. 我的目标是在保证网络不拥塞的情况下尽量提高传输效率;那么所有的包都可以延迟应答么? 肯定也不是;数量限制: 每隔N个包就应答一次;时间限制: 超过最大延迟时间就应答一次;

具体的数量和超时时间, 依操作系统不同也有差异; 一般N取2,   
超时时间取200ms;   
  
**捎带应答**

在延迟应答的基础上, 我们发现, 很多情况下, 客户端服务器在应   
用层也是 “一发一收” 的. 意味着客户端给服务器说了 “How are   
you”, 服务器也会给客户端回一个 “Fine, thank you”;   
那么这个时候ACK就可以搭顺风车, 和服务器回应的 “Fine,

**面向字节流和数据报的区别**

面向报文的传输方式是应用层交给UDP多长的报文，UDP就照样发送，即一次发送一个报文。因此，应用程序必须选择合适大小的报文。若报文太长，则IP层需要分片，降低效率。若太短，会是IP太小。UDP对应用层交下来的报文，既不合并，也不拆分，而是保留这些报文的边界。这也就是说，应用层交给UDP多长的报文，UDP就照样发送，即一次发送一个报文。

面向字节流的话，虽然应用程序和TCP的交互是一次一个数据块（大小不等），但TCP把应用程序看成是一连串的无结构的字节流。TCP有一个缓冲，当应用程序传送的数据块太长，TCP就可以把它划分短一些再传送。如果应用程序一次只发送一个字节，TCP也可以等待积累有足够多的字节后再构成报文段发送出去。

#### ****粘包问题****

**边界**

在前面讲解UDP时，提到了UDP保留了报文的边界，下面我们来谈谈TCP和UDP中报文的边界问题。在默认的阻塞模式下，TCP无边界，UDP有边界。

* 对于TCP协议，客户端连续发送数据，只要服务端的这个函数的缓冲区足够大，会一次性接收过来，即客户端是分好几次发过来，是有边界的，而服务端却一次性接收过来，所以证明是无边界的；
* 而对于UDP协议，客户端连续发送数据，即使服务端的这个函数的缓冲区足够大，也只会一次一次的接收，发送多少次接收多少次，即客户端分几次发送过来，服务端就必须按几次接收，从而证明，这种UDP的通讯模式是有边界的。

**粘包问题**

TCP无边界，造成对采用TCP协议发送的数据进行接收比较麻烦，在接收的时候易出现粘包，即发送方发送的若干包数据到接收方接收时粘成一包。由于TCP是流协议，对于一个socket的包，如发送 10AAAAABBBBB两次，由于网络原因第一次又分成两次发送， 10AAAAAB和BBBB，如果接包的时候先读取10(包长度)再读入后续数据，当接收得快，发送的慢时，就会出现先接收了 10AAAAAB,会解释错误 ,再接到BBBB10AAAAABBBBB，也解释错误的情况。这就是TCP的粘包。   
在网络传输应用中，通常需要在网络协议之上再自定义一个协议封装一下，简单做法就是在要发送的数据前面再加一个自定义的包头，包头中可以包含数据长度和其它一些信息，接收的时候先收包头，再根据包头中描述的数据长度来接收后面的数据。详细做法是：先接收包头，在包头里指定包体长度来接收。设置包头包尾的检查位（ 比如以0xAA开头，0xCC结束来检查一个包是否完整）。对于TCP来说：   
1）不存在丢包，错包，所以不会出现数据出错 ；   
2）如果包头检测错误，即为非法或者请求，直接重置即可。

**那么如何避免粘包问题呢?** 归根结底就是一句话, 明确两个包之   
间的边界.

* 对于定长的包, 保证每次都按固定大小读取即可; 例如上面的Request结构, 是固定大小的, 那么就从缓冲区从头开始按sizeof(Request)依次读取即可;
* 对于变长的包, 可以在包头的位置, 约定一个包总长度的字段, 从而就知道了包的结束位置;
* 对于变长的包, 还可以在包和包之间使用明确的分隔符(应用层协议, 是程序猿自己来定的, 只要保证分隔符不和正文冲突即可);

**思考: 对于UDP协议来说, 是否也存在 “粘包问题” 呢?**   
对于UDP, 如果还没有上层交付数据, UDP的报文长度仍然在. 同时, UDP是一个一个把数据交付给应用层. 就有很明确的数据边界.站在应用层的站在应用层的角度, 使用UDP的时候, 要么收到完整的UDP报文, 要么不收. 不会出现”半个”的情况.

#### ****TCP异常情况****

**进程终止**: 进程终止会释放文件描述符, 仍然可   
以发送FIN. 和正常关闭没有什么区别.

**机器重启**: 和进程终止的情况相同.

**机器掉电/网线断开:** 接收端认为连接还在, 一旦接收端有写入操作, 接收端发现连接已经不在了, 就会进行reset. 即使没有写入操作, TCP自己也内置了一个**保活定时器**, 会定期询问对方是否还在. 如果对方不在, 也会把连接释放.

另外, 应用层的某些协议, 也有一些这样的检测机制. 例如HTTP长连接中, 也会定期检测对方的状态. 例如QQ, 在QQ断线之后, 也会定期尝试重新连接