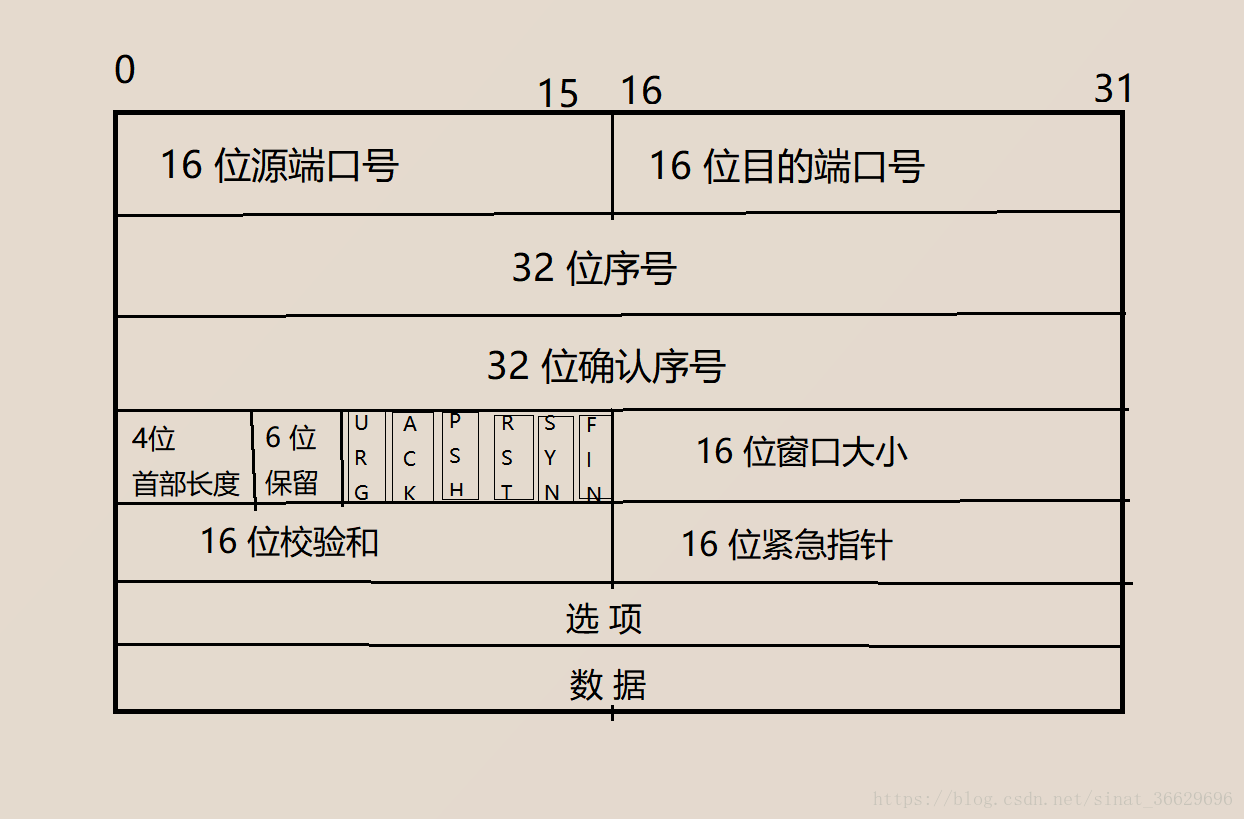
**TCP协议**

TCP协议全称: 传输控制协议, 顾名思义, 就是要对数据的传输进行一定的控制.  
先来看看它的报头  


我们来分析分析每部分的含义和作用

* 源端口号/目的端口号: 表示数据从哪个进程来, 到哪个进程去.
* 32位序号:
* 4位首部长度: 表示该tcp报头有多少个4字节(32个bit)
* 6位保留: 顾名思义, 先保留着, 以防万一
* 6位标志位

URG: 标识紧急指针是否有效  
ACK: 标识确认序号是否有效  
PSH: 用来提示接收端应用程序立刻将数据从tcp缓冲区读走  
RST: 要求重新建立连接. 我们把含有RST标识的报文称为**复位报文段**  
SYN: 请求建立连接. 我们把含有SYN标识的报文称为**同步报文段**  
FIN: 通知对端, 本端即将关闭. 我们把含有FIN标识的报文称为**结束报文段**

* 16位窗口大小:
* 16位检验和: 由发送端填充, 检验形式有CRC校验等. 如果接收端校验不通过, 则认为数据有问题. 此处的校验和不光包含TCP首部, 也包含TCP数据部分.
* 16位紧急指针: 用来标识哪部分数据是紧急数据.
* 选项和数据暂时忽略

**连接管理机制**

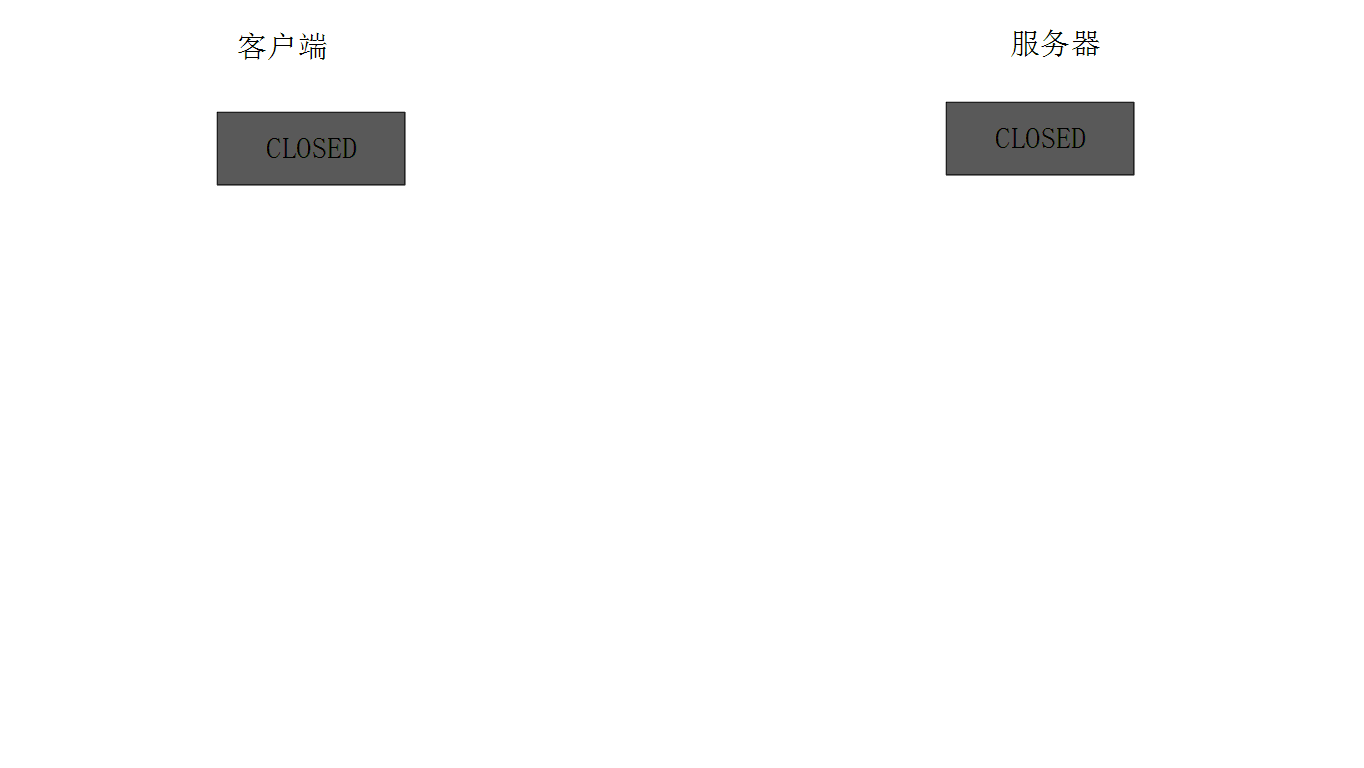
正常情况下, tcp需要经过三次握手建立连接, 四次挥手断开连接.

那么什么是三次握手? 什么是四次挥手呢?

**三次握手**

第一次:  
客户端 - - > 服务器 此时服务器知道了客户端要建立连接了  
第二次:  
客户端 < - - 服务器 此时客户端知道服务器收到连接请求了  
第三次:  
客户端 - - > 服务器 此时服务器知道客户端收到了自己的回应

到这里, 就可以认为客户端与服务器已经建立了连接.

再来看个图.  


刚开始, 客户端和服务器都处于 CLOSE 状态.  
此时, 客户端向服务器主动发出连接请求, 服务器被动接受连接请求.

1, TCP服务器进程先创建传输控制块TCB, 时刻准备接受客户端进程的连接请求, 此时服务器就进入了 LISTEN（监听）状态  
2, TCP客户端进程也是先创建传输控制块TCB, 然后向服务器发出连接请求报文，此时报文首部中的同步标志位SYN=1, 同时选择一个初始序列号 seq = x, 此时，TCP客户端进程进入了 SYN-SENT（同步已发送状态）状态。TCP规定, SYN报文段（SYN=1的报文段）不能携带数据，但需要消耗掉一个序号。  
3, TCP服务器收到请求报文后, 如果同意连接, 则发出确认报文。确认报文中的 ACK=1, SYN=1, 确认序号是 x+1, 同时也要为自己初始化一个序列号 seq = y, 此时, TCP服务器进程进入了SYN-RCVD（同步收到）状态。这个报文也不能携带数据, 但是同样要消耗一个序号。  
4, TCP客户端进程收到确认后还, 要向服务器给出确认。确认报文的ACK=1，确认序号是 y+1，自己的序列号是 x+1.  
5, 此时，TCP连接建立，客户端进入ESTABLISHED（已建立连接）状态。当服务器收到客户端的确认后也进入ESTABLISHED状态，此后双方就可以开始通信了。

为什么不用两次?

* 主要是为了防止已经失效的连接请求报文突然又传送到了服务器，从而产生错误。如果使用的是两次握手建立连接，假设有这样一种场景，客户端发送的第一个请求连接并且没有丢失，只是因为在网络中滞留的时间太长了，由于TCP的客户端迟迟没有收到确认报文，以为服务器没有收到，此时重新向服务器发送这条报文，此后客户端和服务器经过两次握手完成连接，传输数据，然后关闭连接。此时之前滞留的那一次请求连接，因为网络通畅了, 到达了服务器，这个报文本该是失效的，但是，两次握手的机制将会让客户端和服务器再次建立连接，这将导致不必要的错误和资源的费。  
  如果采用的是三次握手，就算是那一次失效的报文传送过来了，服务端接受到了那条失效报文并且回复了确认报文，但是客户端不会再次发出确认。由于服务器收不到确认，就知道客户端并没有请求连接。

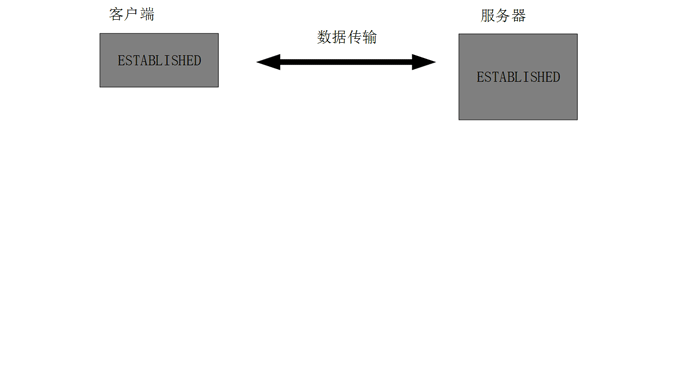
为什么不用四次?

* 因为三次已经可以满足需要了, 四次就多余了.

**再来看看何为四次挥手.**

数据传输完毕后，双方都可以释放连接.  
此时客户端和服务器都是处于ESTABLISHED状态，然后客户端主动断开连接，服务器被动断开连接.

1, 客户端进程发出连接释放报文，并且停止发送数据。  
释放数据报文首部，FIN=1，其序列号为seq=u（等于前面已经传送过来的数据的最后一个字节的序号加1），此时客户端进入FIN-WAIT-1（终止等待1）状态。 TCP规定，FIN报文段即使不携带数据，也要消耗一个序号。  
2, 服务器收到连接释放报文，发出确认报文，ACK=1，确认序号为 u+1，并且带上自己的序列号seq=v，此时服务端就进入了CLOSE-WAIT（关闭等待）状态。  
TCP服务器通知高层的应用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个CLOSE-WAIT状态持续的时间。  
3, 客户端收到服务器的确认请求后，此时客户端就进入FIN-WAIT-2（终止等待2）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最终数据）  
4, 服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，FIN=1，确认序号为v+1，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为seq=w，此时，服务器就进入了LAST-ACK（最后确认）状态，等待客户端的确认。  
5, 客户端收到服务器的连接释放报文后，必须发出确认，ACK=1，确认序号为w+1，而自己的序列号是u+1，此时，客户端就进入了TIME-WAIT（时间等待）状态。注意此时TCP连接还没有释放，必须经过2∗MSL（最长报文段寿命）的时间后，当客户端撤销相应的TCB后，才进入CLOSED状态。  
6, 服务器只要收到了客户端发出的确认，立即进入CLOSED状态。同样，撤销TCB后，就结束了这次的TCP连接。可以看到，服务器结束TCP连接的时间要比客户端早一些。

再来看一张图.  


为什么最后客户端还要等待 2\*MSL的时间呢?

* MSL（Maximum Segment Lifetime），TCP允许不同的实现可以设置不同的MSL值。
* 第一，保证客户端发送的最后一个ACK报文能够到达服务器，因为这个ACK报文可能丢失，站在服务器的角度看来，我已经发送了FIN+ACK报文请求断开了，客户端还没有给我回应，应该是我发送的请求断开报文它没有收到，于是服务器又会重新发送一次，而客户端就能在这个2MSL时间段内收到这个重传的报文，接着给出回应报文，并且会重启2MSL计时器。
* 第二，防止类似与“三次握手”中提到了的“已经失效的连接请求报文段”出现在本连接中。客户端发送完最后一个确认报文后，在这个2MSL时间中，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失。这样新的连接中不会出现旧连接的请求报文。

为什么建立连接是三次握手，关闭连接确是四次挥手呢？

* 建立连接的时候， 服务器在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。  
  而关闭连接时，服务器收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，而自己也未必全部数据都发送给对方了，所以己方可以立即关闭，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送，从而导致多了一次。

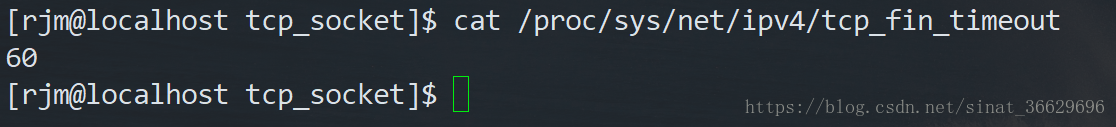
如果已经建立了连接, 但是客户端突发故障了怎么办?

* TCP设有一个保活计时器，显然，客户端如果出现故障，服务器不能一直等下去，白白浪费资源。服务器每收到一次客户端的请求后都会重新复位这个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75分钟发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

**理解TIME\_WAIT状态**

可以做一个实验, 先运行server, 再运行client连接server, 然后断开server, 再立马运行server.  
我们会发现:  


绑定的时候出了问题.  
这是因为,虽然server应用程序终止了,但TCP协议层的连接并没有完全断开,因此不能再次监听绑定同样的server端口.

TCP协议规定,主动关闭连接的一方要处于TIME\_ WAIT状态,等待2\*MSL(maximum segment lifetime)的时间后才能回到CLOSED状态.  
我们使用Ctrl-C终止了server, 所以server是主动关闭连接的一方, 在TIME\_WAIT期间仍然不能再次监听同样的server端口  
MSL在RFC1122中规定为两分钟,但是各操作系统的实现不同, 在Centos7上默认配置的值是60s;  
可以通过 cat /proc/sys/net/ipv4/tcp\_fin\_timeout 查看MSL的值  
  
**解决TIME\_WAIT引起的bind失败问题**  
在server的TCP连接没有完全断开之前不允许重新监听, 某些情况下可能是不合理的.

比如:

服务器需要处理非常大量的客户端的连接(每个连接的生存时间可能很短, 但是每秒都有大量的客户端来请求).  
这个时候如果由服务器端主动关闭连接(比如某些客户端不活跃, 就需要被服务器端主动清理掉), 就会产生大量TIME\_WAIT连接.  
由于我们的请求量很大, 就可能导致TIME\_WAIT的连接数很多, 导致服务器的端口不够用, 无法处理新的连接.

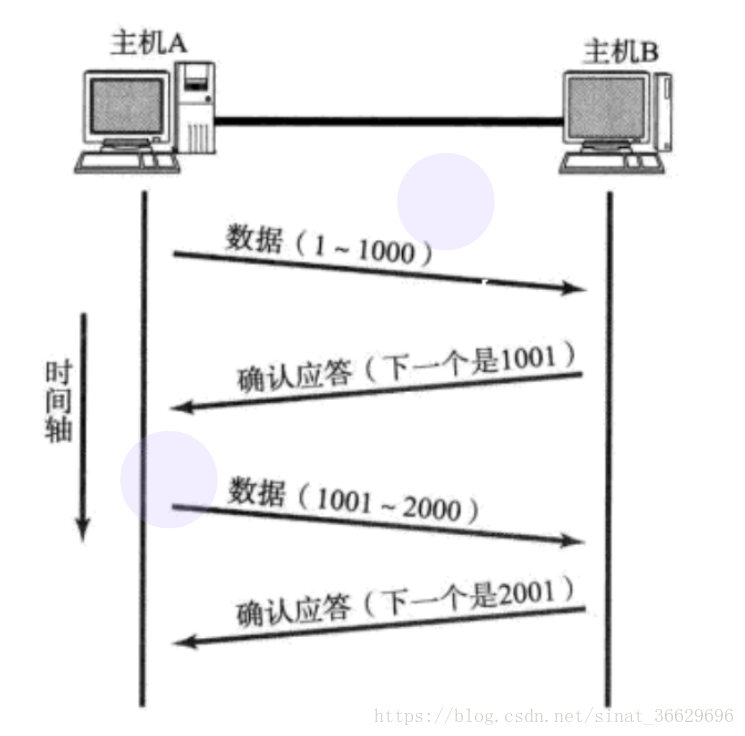
解决方法:  
- 使用setsockopt()设置socket描述符的选项SO\_REUSEADDR为1, 表示允许创建端口号相同但IP地址不同的多个socket描述符.

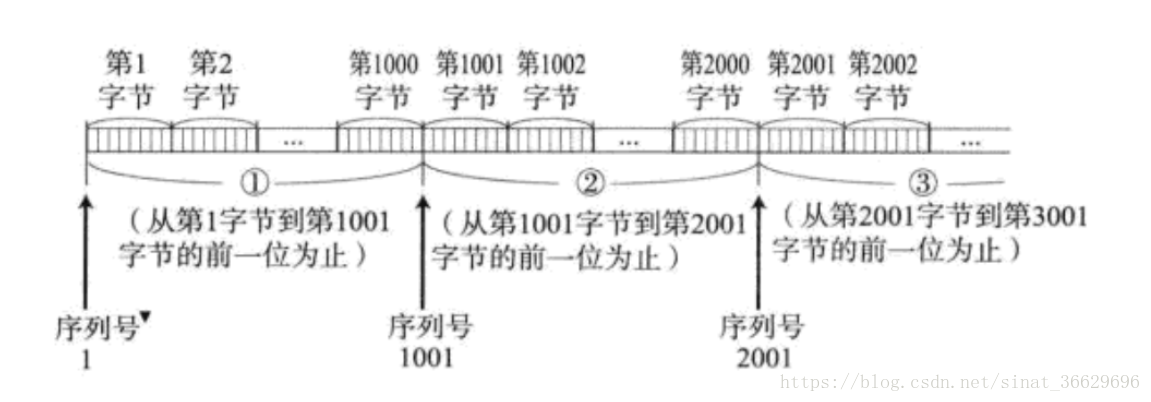
用法:

* 在server代码的socket()和bind()调用之间插入如下代码

int opt = 1;  
setsockopt(listen\_fd, SOL\_SOCKET, SO\_REUSEADDR, &opt, sizeof(opt));

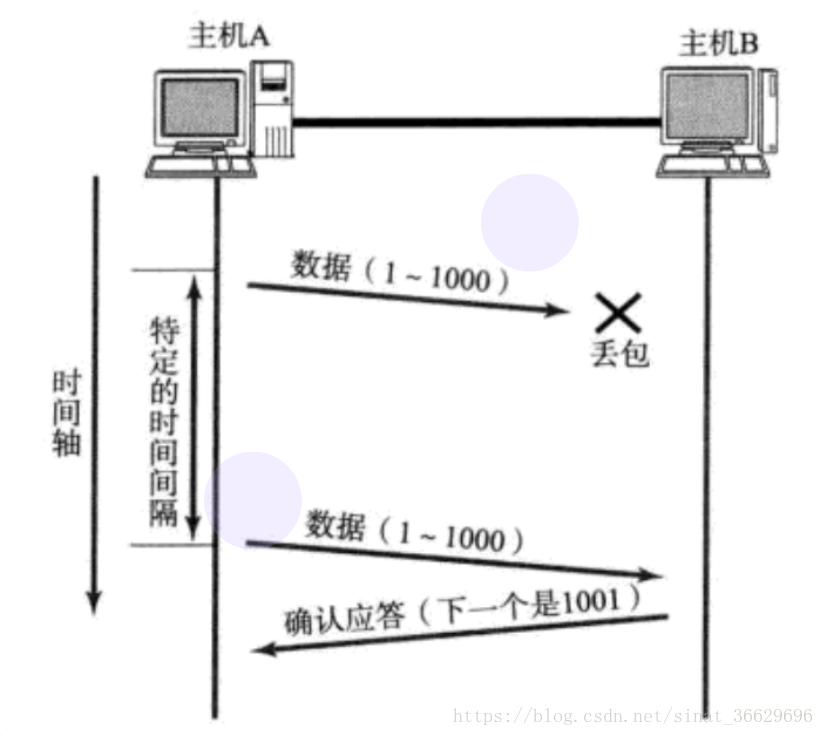
**确认应答机制(ACK机制)**

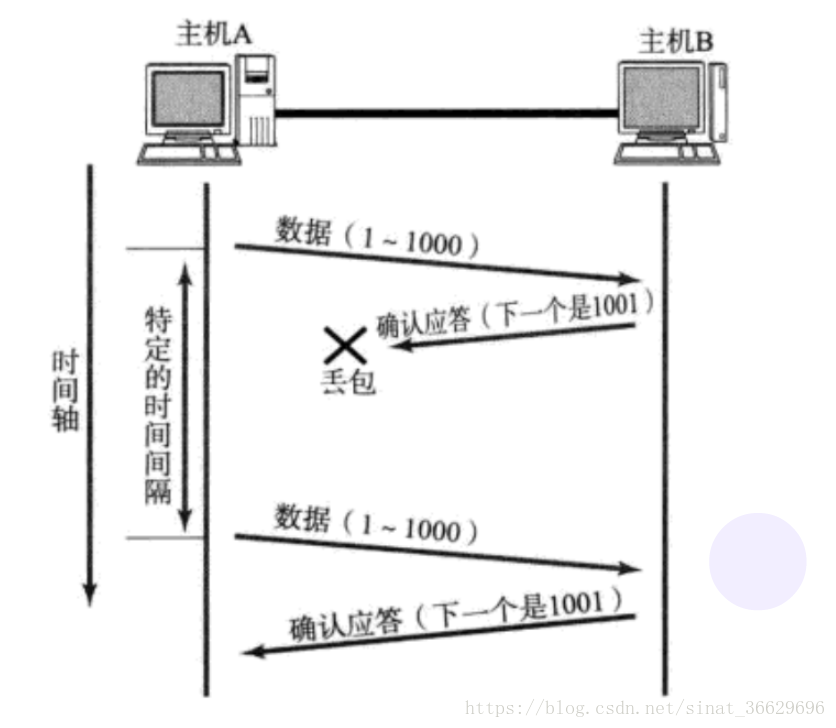


TCP将每个字节的数据都进行了编号, 即为序列号.  


每一个ACK都带有对应的确认序列号, 意思是告诉发送者, 我已经收到了哪些数据; 下一次你要从哪里开始发.  
比如, 客户端向服务器发送了1005字节的数据, 服务器返回给客户端的确认序号是1003, 那么说明服务器只收到了1-1002的数据.  
1003, 1004, 1005都没收到.  
此时客户端就会从1003开始重发.

**超时重传机制**



主机A发送数据给B之后, 可能因为网络拥堵等原因, 数据无法到达主机B  
如果主机A在一个特定时间间隔内没有收到B发来的确认应答, 就会进行重发  
但是主机A没收到确认应答也可能是ACK丢失了.  


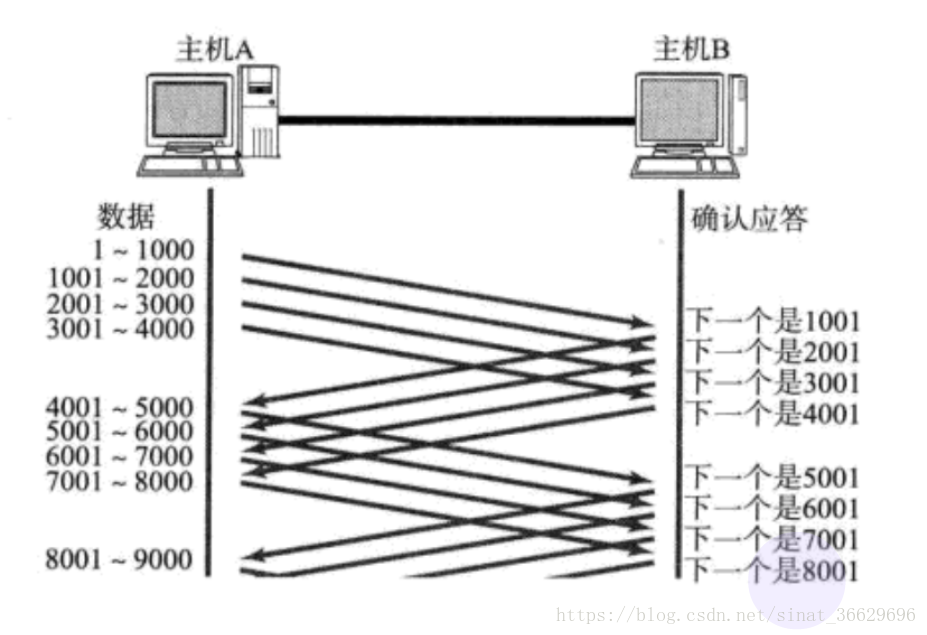
这种情况下, 主机B会收到很多重复数据.  
那么TCP协议需要识别出哪些包是重复的, 并且把重复的丢弃.  
这时候利用前面提到的序列号, 就可以很容易做到去重.

**超时时间如何确定?**  
最理想的情况下, 找到一个最小的时间, 保证 “确认应答一定能在这个时间内返回”.  
但是这个时间的长短, 随着网络环境的不同, 是有差异的.  
如果超时时间设的太长, 会影响整体的重传效率; 如果超时时间设的太短, 有可能会频繁发送重复的包.

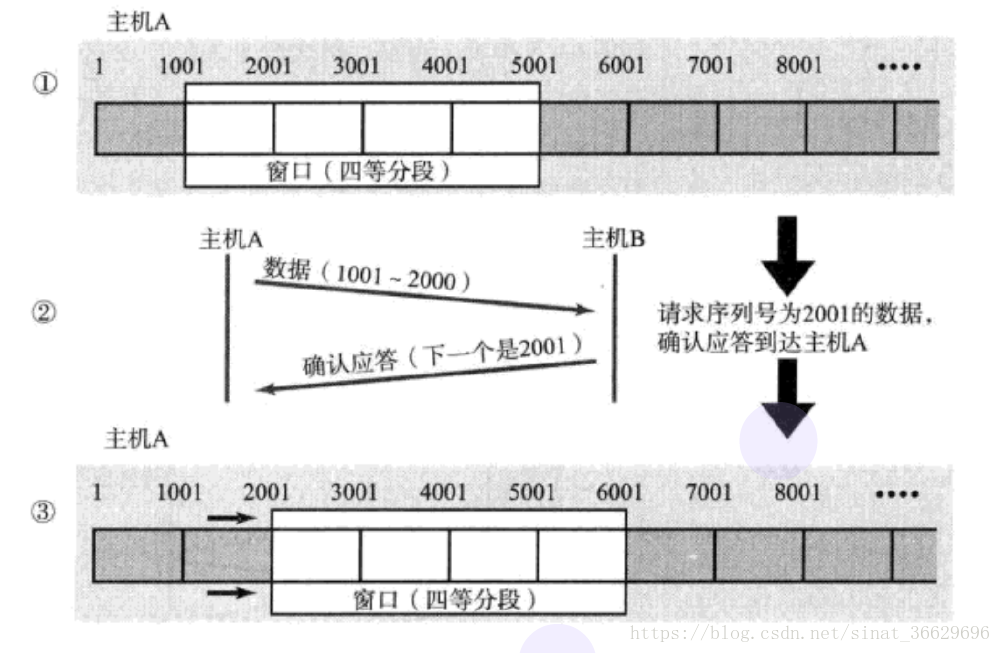
TCP为了保证任何环境下都能保持较高性能的通信, 因此会**动态计算**这个最大超时时间.

* Linux中(BSD Unix和Windows也是如此), 超时以500ms为一个单位进行控制, 每次判定超时重发的超时时间都是500ms的整数倍.  
  如果重发一次之后, 仍然得不到应答, 等待 2\*500ms 后再进行重传. 如果仍然得不到应答, 等待 4\*500ms 进行重传.  
  依次类推, 以指数形式递增. 累计到一定的重传次数, TCP认为网络异常或者对端主机出现异常, 强制关闭连接.

**滑动窗口**

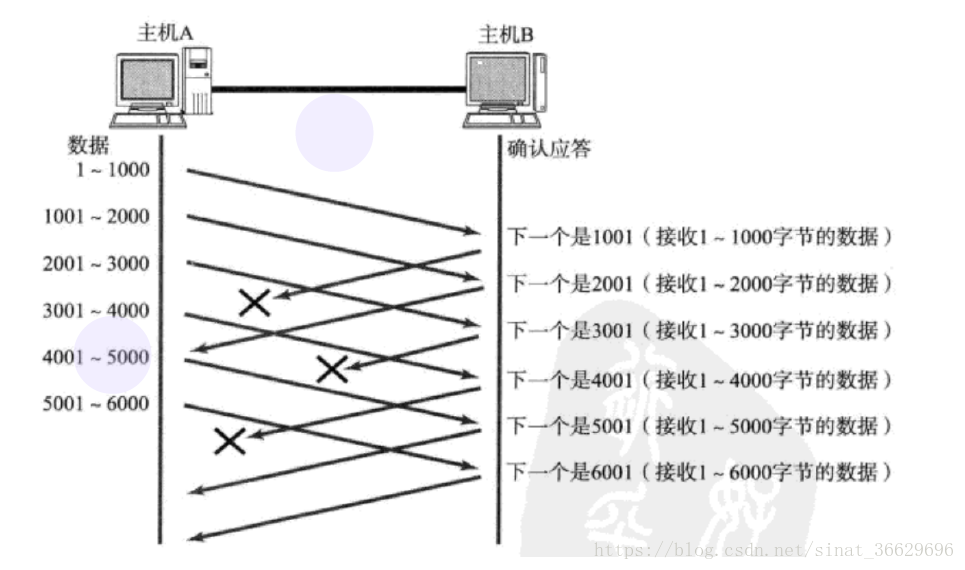
刚才我们讨论了确认应答机制, 对每一个发送的数据段, 都要给一个ACK确认应答. 收到ACK后再发送下一个数据段.  
这样做有一个比较大的缺点, 就是性能较差. 尤其是数据往返时间较长的时候.  
那么我们可不可以一次发送多个数据段呢?  
例如这样:  
  
一个概念: **窗口**  
窗口大小指的是无需等待确认应答就可以继续发送数据的最大值.  
上图的窗口大小就是4000个字节 (四个段).

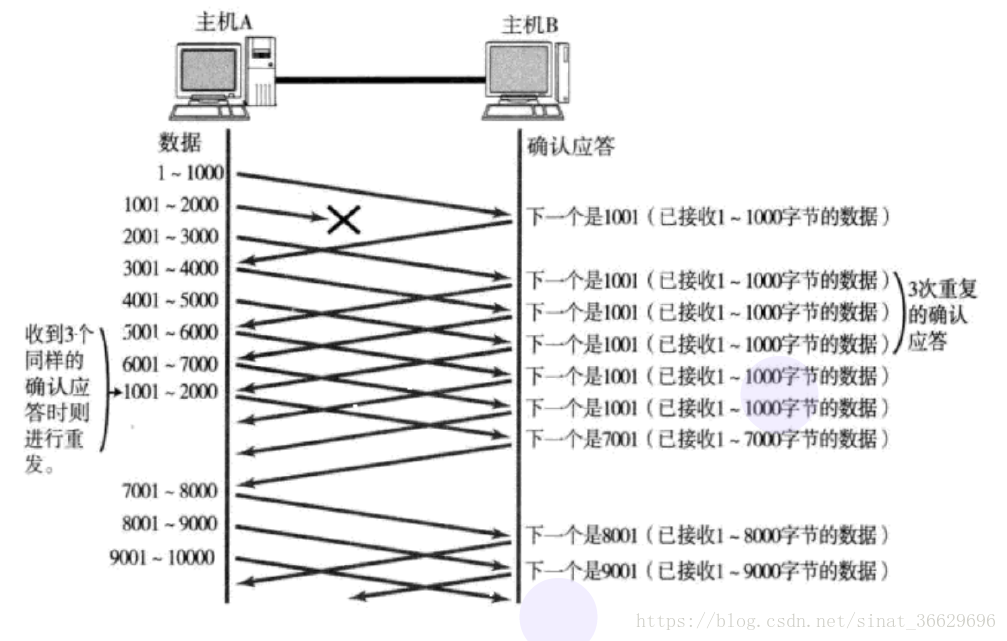
发送前四个段的时候, 不需要等待任何ACK, 直接发送  
收到第一个ACK确认应答后, 窗口向后移动, 继续发送第五六七八段的数据…

因为这个窗口不断向后滑动, 所以叫做**滑动窗口.**  
操作系统内核为了维护这个滑动窗口, 需要开辟发送缓冲区来记录当前还有哪些数据没有应答  
只有ACK确认应答过的数据, 才能从缓冲区删掉.  


如果出现了丢包, 那么该如何进行重传呢?

此时分两种情况讨论:

1, 数据包已经收到, 但确认应答ACK丢了.  
  
这种情况下, 部分ACK丢失并无大碍, 因为还可以通过后续的ACK来确认对方已经收到了哪些数据包.

2, 数据包丢失  


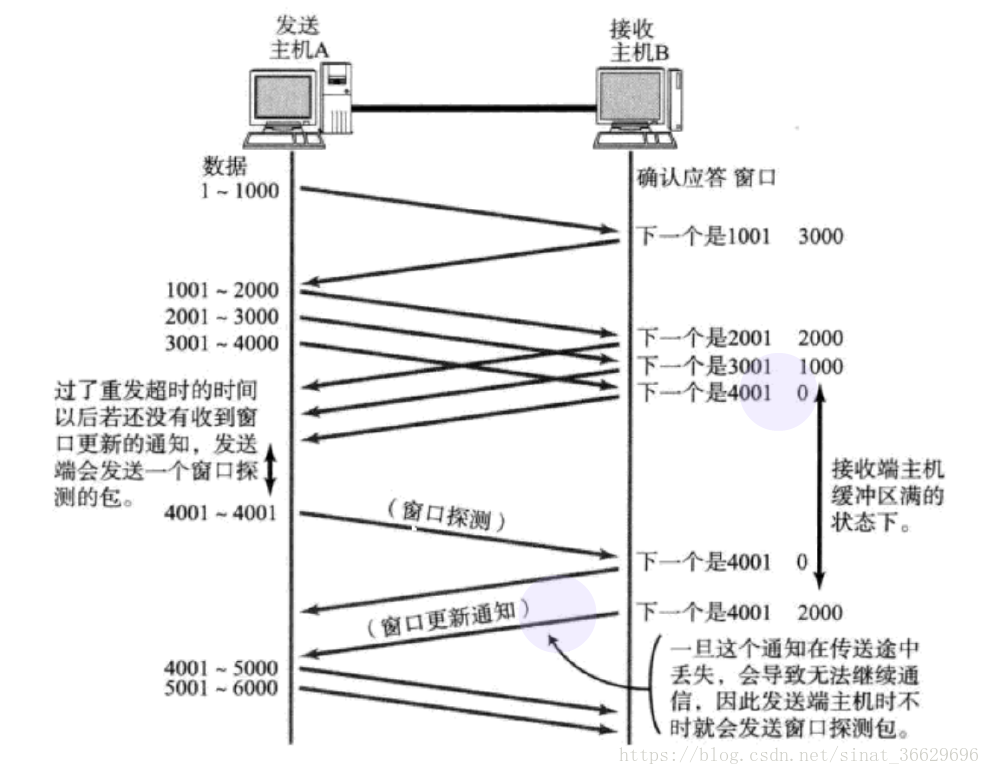
当某一段报文丢失之后, 发送端会一直收到 1001 这样的ACK, 就像是在提醒发送端 “我想要的是 1001”  
如果发送端主机连续三次收到了同样一个 “1001” 这样的应答, 就会将对应的数据 1001 - 2000 重新发送  
这个时候接收端收到了 1001 之后, 再次返回的ACK就是7001了  
因为2001 - 7000接收端其实之前就已经收到了, 被放到了接收端操作系统内核的接收缓冲区中.

**这种机制被称为 “高速重发控制” ( 也叫 “快重传” )**

**流量控制**

接收端处理数据的速度是有限的. 如果发送端发的太快, 导致接收端的缓冲区被填满, 这个时候如果发送端继续发送, 就会造成丢包, 进而引起丢包重传等一系列连锁反应.  
因此TCP支持根据接收端的处理能力, 来决定发送端的发送速度.  
这个机制就叫做 **流量控制(Flow Control)**

接收端将自己可以接收的缓冲区大小放入 TCP 首部中的 “窗口大小” 字段,  
通过ACK通知发送端;  
窗口大小越大, 说明网络的吞吐量越高;  
接收端一旦发现自己的缓冲区快满了, 就会将窗口大小设置成一个更小的值通知给发送端;  
发送端接受到这个窗口大小的通知之后, 就会减慢自己的发送速度;  
如果接收端缓冲区满了, 就会将窗口置为0;  
这时发送方不再发送数据, 但是需要定期发送一个窗口探测数据段, 让接收端把窗口大小再告诉发送端.

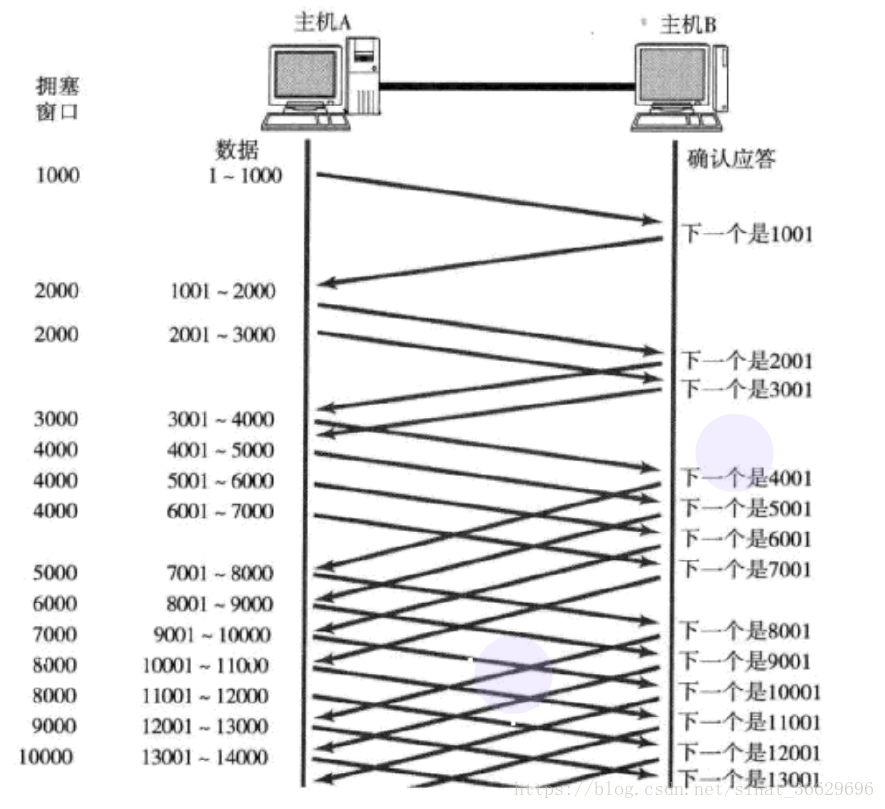


那么接收端如何把窗口大小告诉发送端呢?  
我们的TCP首部中, 有一个16位窗口大小字段, 就存放了窗口大小的信息;  
16位数字最大表示65536, 那么TCP窗口最大就是65536字节么?  
实际上, TCP首部40字节选项中还包含了一个窗口扩大因子M, 实际窗口大小是窗口字段的值左移 M 位(左移一位相当于乘以2).

**拥塞控制**

虽然TCP有了滑动窗口这个大杀器, 能够高效可靠地发送大量数据.  
但是如果在刚开始就发送大量的数据, 仍然可能引发一些问题.  
因为网络上有很多计算机, 可能当前的网络状态已经比较拥堵.  
在不清楚当前网络状态的情况下, 贸然发送大量数据, 很有可能雪上加霜.

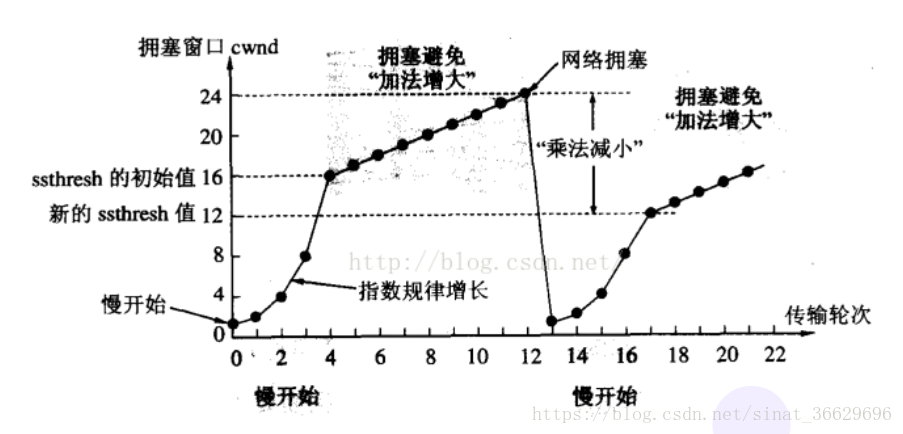
因此, TCP引入 慢启动 机制, 先发少量的数据, 探探路, 摸清当前的网络拥堵状态以后, 再决定按照多大的速度传输数据.



在此引入一个概念 拥塞窗口

* 发送开始的时候, 定义拥塞窗口大小为1;
* 每次收到一个ACK应答, 拥塞窗口加1;
* 每次发送数据包的时候, 将拥塞窗口和接收端主机反馈的窗口大小做比较, 取较小的值作为实际发送的窗口

像上面这样的拥塞窗口增长速度, 是指数级别的.  
“慢启动” 只是指初使时慢, 但是增长速度非常快.  
为了不增长得那么快, 此处引入一个名词叫做慢启动的阈值, 当拥塞窗口的大小超过这个阈值的时候, 不再按照指数方式增长, 而是按照线性方式增长.



* 当TCP开始启动的时候, 慢启动阈值等于窗口最大值
* 在每次超时重发的时候, 慢启动阈值会变成原来的一半, 同时拥塞窗口置回1

少量的丢包, 我们仅仅是触发超时重传;  
大量的丢包, 我们就认为是网络拥塞;  
当TCP通信开始后, 网络吞吐量会逐渐上升;  
随着网络发生拥堵, 吞吐量会立刻下降.

**拥塞控制**, 归根结底是TCP协议想尽可能快的把数据传输给对方, 但是又要避免给网络造成太大压力的折中方案.

**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, 返回的窗口大小就是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儿再应答, 比如等待200ms再应答, 那么这个时候返回的窗口大小就是1M

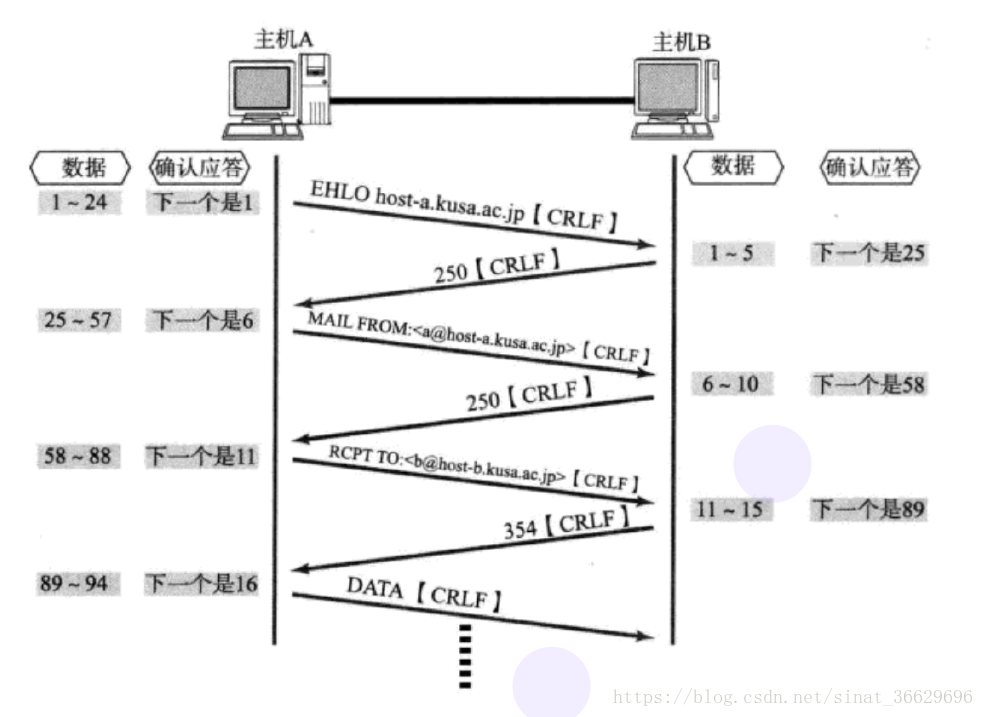
窗口越大, 网络吞吐量就越大, 传输效率就越高.  
TCP的目标是在保证网络不拥堵的情况下尽量提高传输效率;

那么所有的数据包都可以延迟应答么?  
肯定也不是  
有两个限制

* 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N和最大延迟时间, 依操作系统不同也有差异  
一般 N 取2, 最大延迟时间取200ms

**捎带应答**

在延迟应答的基础上, 我们发现, 很多情况下  
客户端和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, thank you” 一起发送给客户端  


**面向字节流**

创建一个TCP的socket, 同时在内核中创建一个 发送缓冲区 和一个 接收缓冲区;  
调用write时, 数据会先写入发送缓冲区中;  
如果发送的字节数太大, 会被拆分成多个TCP的数据包发出;  
如果发送的字节数太小, 就会先在缓冲区里等待, 等到缓冲区大小差不多了, 或者到了其他合适的时机再发送出去;  
接收数据的时候, 数据也是从网卡驱动程序到达内核的接收缓冲区;  
然后应用程序可以调用read从接收缓冲区拿数据;  
另一方面, TCP的一个连接, 既有发送缓冲区, 也有接收缓冲区,  
那么对于这一个连接, 既可以读数据, 也可以写数据, 这个概念叫做 全双工

由于缓冲区的存在, 所以TCP程序的读和写不需要一一匹配  
例如:

* 写100个字节的数据, 可以调用一次write写100个字节, 也可以调用100次write, 每次写一个字节;
* 读100个字节数据时, 也完全不需要考虑写的时候是怎么写的, 既可以一次read 100个字节, 也可以一次read一个字节, 重复100次;

**粘包问题**

首先要明确, 粘包问题中的 “包”, 是指应用层的数据包.  
在TCP的协议头中, 没有如同UDP一样的 “报文长度” 字段  
但是有一个序号字段.  
站在传输层的角度, TCP是一个一个报文传过来的. 按照序号排好序放在缓冲区中.  
站在应用层的角度, 看到的只是一串连续的字节数据.  
那么应用程序看到了这一连串的字节数据, 就不知道从哪个部分开始到哪个部分是一个完整的应用层数据包.  
此时数据之间就没有了边界, 就产生了粘包问题

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  
- 可以在数据包的头部, 约定一个数据包总长度的字段, 从而就知道了包的结束位置  
还可以在包和包之间使用明确的分隔符来作为边界(应用层协议, 是程序员自己来定的, 只要保证分隔符不和正文冲突即可)

**对于UDP协议来说, 是否也存在 “粘包问题” 呢?**

对于UDP, 如果还没有向上层交付数据, UDP的报文长度仍然存在.  
同时, UDP是一个一个把数据交付给应用层的, 就有很明确的数据边界.  
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.  
不会出现收到 “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.

**TCP 小结**

为什么TCP这么复杂?

因为既要保证可靠性, 同时又要尽可能提高性能.

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
* 连接管理
* 流量控制
* 拥塞控制

提高性能的机制

* 滑动窗口
* 快速重传
* 延迟应答
* 捎带应答

定时器

* 超时重传定时器
* 保活定时器
* TIME\_WAIT定时器

**基于 TCP 的应用层协议**

HTTP  
HTTPS  
SSH  
Telnet  
FTP  
SMTP  
…

当然, 也包括我们自己写TCP程序时自定义的应用层协议

**TCP 和 UDP 对比**

我们说了TCP是可靠连接, 那么是不是TCP一定就优于UDP呢?

TCP和UDP之间的优点和缺点, 不能简单绝对地进行比较  
TCP用于可靠传输的情况, 应用于文件传输, 重要状态更新等场景  
UDP用于对高速传输和实时性要求较高的通信领域  
例如, 早期的QQ, 视频传输等. 另外UDP可以用于广播

归根结底, TCP和UDP都是一种工具, 什么时机用, 具体怎么用, 还是要根据具体的需求场景去决定.