<https://blog.csdn.net/fishmai/article/details/78344740>

Tcp协议本身是可靠的,并不等于应用程序用tcp发送数据就一定是可靠的.不管是否阻塞,send发送的大小,并不代表对端recv到多少的数据.  
  
在阻塞模式下, send函数的过程是将应用程序请求发送的数据拷贝到发送缓存中发送并得到确认后再返回.但由于发送缓存的存在,表现为:如果发送缓存大小比请求发送的大小要大,那么send函数立即返回,同时向网络中发送数据;否则,send向网络发送缓存中不能容纳的那部分数据,并等待对端确认后再返回(接收端只要将数据收到接收缓存中,就会确认,并不一定要等待应用程序调用recv);  
  
在非阻塞模式下,send函数的过程仅仅是将数据拷贝到协议栈的缓存区而已,如果缓存区可用空间不够,则尽能力的拷贝,返回成功拷贝的大小;如缓存区可用空间为0,则返回-1,同时设置errno为EAGAIN.  
  
  
linux下可用sysctl -a | grep net.ipv4.tcp\_wmem查看系统默认的发送缓存大小:  
net.ipv4.tcp\_wmem = 4096 16384 81920  
这有三个值,第一个值是socket的发送缓存区分配的最少字节数,第二个值是默认值(该值会被net.core.wmem\_default覆盖),缓存区在系统负载不重的情况下可以增长到这个值,第三个值是发送缓存区空间的最大字节数(该值会被net.core.wmem\_max覆盖).  
根据实际测试,如果手工更改了net.ipv4.tcp\_wmem的值,则会按更改的值来运行,否则在默认情况下,协议栈通常是按net.core.wmem\_default和net.core.wmem\_max的值来分配内存的.  
  
应用程序应该根据应用的特性在程序中更改发送缓存大小:  
  
socklen\_t sendbuflen = 0;  
socklen\_t len = sizeof(sendbuflen);  
getsockopt(clientSocket, SOL\_SOCKET, SO\_SNDBUF, (void\*)&sendbuflen, &len);  
printf("default,sendbuf:%d/n", sendbuflen);  
  
sendbuflen = 10240;  
setsockopt(clientSocket, SOL\_SOCKET, SO\_SNDBUF, (void\*)&sendbuflen, len);  
getsockopt(clientSocket, SOL\_SOCKET, SO\_SNDBUF, (void\*)&sendbuflen, &len);  
printf("now,sendbuf:%d/n", sendbuflen);  
  
  
需要注意的是,虽然将发送缓存设置成了10k,但实际上,协议栈会将其扩大1倍,设为20k.  
-------------------实例分析---------------  
  
在实际应用中,如果发送端是非阻塞发送,由于网络的阻塞或者接收端处理过慢,通常出现的情况是,发送应用程序看起来发送了10k的数据,但是只发送了2k到对端缓存中,还有8k在本机缓存中(未发送或者未得到接收端的确认).那么此时,接收应用程序能够收到的数据为2k.假如接收应用程序调用recv函数获取了1k的数据在处理,在这个瞬间,发生了以下情况之一,双方表现为:  
  
A. 发送应用程序认为send完了10k数据,关闭了socket:  
发送主机作为tcp的主动关闭者,连接将处于FIN\_WAIT1的半关闭状态(等待对方的ack),并且,发送缓存中的8k数据并不清除,依然会发送给对端.如果接收应用程序依然在recv,那么它会收到余下的8k数据(这个前题是,接收端会在发送端FIN\_WAIT1状态超时前收到余下的8k数据.), 然后得到一个对端socket被关闭的消息(recv返回0).这时,应该进行关闭.  
  
B. 发送应用程序再次调用send发送8k的数据:  
假如发送缓存的空间为20k,那么发送缓存可用空间为20-8=12k,大于请求发送的8k,所以send函数将数据做拷贝后,并立即返回8192;  
  
假如发送缓存的空间为12k,那么此时发送缓存可用空间还有12-8=4k,send()会返回4096,应用程序发现返回的值小于请求发送的大小值后,可以认为缓存区已满,这时必须阻塞(或通过select等待下一次socket可写的信号),如果应用程序不理会,立即再次调用send,那么会得到-1的值, 在linux下表现为errno=EAGAIN.  
  
C. 接收应用程序在处理完1k数据后,关闭了socket:  
接收主机作为主动关闭者,连接将处于FIN\_WAIT1的半关闭状态(等待对方的ack).然后,发送应用程序会收到socket可读的信号(通常是 select调用返回socket可读),但在读取时会发现recv函数返回0,这时应该调用close函数来关闭socket(发送给对方ack);  
  
如果发送应用程序没有处理这个可读的信号,而是在send,那么这要分两种情况来考虑,假如是在发送端收到RST标志之后调用send,send将返回-1,同时errno设为ECONNRESET表示对端网络已断开,但是,也有说法是进程会收到SIGPIPE信号,该信号的默认响应动作是退出进程,如果忽略该信号,那么send是返回-1,errno为EPIPE(未证实);如果是在发送端收到RST标志之前,则send像往常一样工作;  
  
以上说的是非阻塞的send情况,假如send是阻塞调用,并且正好处于阻塞时(例如一次性发送一个巨大的buf,超出了发送缓存),对端socket关闭,那么send将返回成功发送的字节数,如果再次调用send,那么会同上一样.  
  
D. 交换机或路由器的网络断开:  
接收应用程序在处理完已收到的1k数据后,会继续从缓存区读取余下的1k数据,然后就表现为无数据可读的现象,这种情况需要应用程序来处理超时.一般做法是设定一个select等待的最大时间,如果超出这个时间依然没有数据可读,则认为socket已不可用.  
  
发送应用程序会不断的将余下的数据发送到网络上,但始终得不到确认,所以缓存区的可用空间持续为0,这种情况也需要应用程序来处理.  
  
如果不由应用程序来处理这种情况超时的情况,也可以通过tcp协议本身来处理,具体可以查看sysctl项中的:  
net.ipv4.tcp\_keepalive\_intvl  
net.ipv4.tcp\_keepalive\_probes  
net.ipv4.tcp\_keepalive\_time

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

http://www.ixpub.net/thread-1446913-1-1.html

发送成功只是表示发到了内核socket缓冲区  
此时如果close，正常情况会进入TIME\_WAIT状态，在此状态，对端可以继续接收数据  
但是如果发送方的接收缓冲区还有未读数据，就会走异常close的途径，置RST，立刻结束连接，没有TIME\_WAIT状态。这时对端就收不全数据，报错： Connection reset by peer。

///////////////////////////////////////////////////////////////////////////////////////////

值得参考的 TCP send和recv函数解析

http://blog.csdn.net/wjtxt/article/details/6603456

**一、 滑动窗口的概念**

        TCP数据包的TCP头部有一个window字段，它主要是用来告诉对方自己能接收多大的数据（注意只有TCP包中的数据部分占用这个空间），这个字段在通信双方建立连接时协商确定，并且在通信过程中不断更新，故取名为滑动窗口。有了这个字段，数据发送方就知道自己该不该发送数据，以及该发多少数据了。TCP协议的流量控制正是通过滑动窗口实现，从而保证通信双方的接收缓冲区不会溢出，数据不会丢失。

由于窗口大小在TCP头部只有16位来表示，所以它的最大值是65536，但是对于一些情况来说需要使用更大的滑动窗口，这时候就要使用扩展的滑动窗口，如光纤高速通信网络，或者是卫星长连接网络，需要窗口尽可能的大。这时会使用扩展的32位的滑动窗口大小。

**二、 滑动窗口移动规则**

        1、窗口合拢：在收到对端数据后，自己确认了数据的正确性，这些数据会被存储到接收缓冲区，等待应用程序获取。但这时候因为已经确认了数据的正确性，需要向对方发送确认响应ACK，又因为这些数据还没有被应用进程取走，这时候便需要进行窗口合拢，缓冲区的窗口左边缘向右滑动。注意响应的ACK序号是对方发送数据包的序号，一个对方发送的序号，可能因为窗口张开会被响应（ACK）多次。

        2、窗口张开：窗口收缩后，应用进程一旦从缓冲区(滑动窗口区或接收缓冲区)中取出数据，TCP的滑动窗口需要进行扩张，这时候窗口的右边缘向右扩张，实际上窗口这是一个环形缓冲区，窗口的右边缘扩张会使用原来被应用进程取走内容的缓冲区。在窗口进行扩张后，需要使用ACK通知对端，这时候ACK的序号依然是上次确认收到包的序号。

        3、窗口收缩，窗口的右边缘向左滑动，称为窗口收缩，HostRequirement RFC强烈建议不要这样做，但TCP必须能够在某一端产生这种情况时进行处理。

**三、send行为**

        默认情况下，send的功能是拷贝指定长度的数据到发送缓冲区，只有当数据被全部拷贝完成后函数才会正确返回，否则进入阻塞状态或等待超时。如果你想修改这种默认行为，将数据直接发送到目标机器，可以将发送缓冲区大小设为0，这样当send返回时，就表示数据已经正确的、完整的到达了目标机器。注意，这里只表示数据到达目标机器网络缓冲区，并不表示数据已经被对方应用层接收了。

        协议层在数据发送过程中，根据对方的滑动窗口，再结合MSS值共同确定TCP报文中数据段的长度，以确保对方接收缓冲区不会溢出。当本方发送缓冲区尚有数据没有发送，而对方滑动窗口已经为0时，协议层将启动探测机制，即每隔一段时间向对方发送一个字节的数据，时间间隔会从刚开始的30s调整为1分钟，最后稳定在2分钟。这个探测机制不仅可以检测到对方滑动窗口是否变化，同时也可以发现对方是否有异常退出的情况。

        push标志指示接收端应尽快将数据提交给应用层。如果send函数提交的待发送数据量较小，例如小于1460B（参照MSS值确定），那么协议层会将该报文中的TCP头部的push字段置为1；如果待发送的数据量较大，需要拆成多个数据段发送时，协议层只会将最后一个分段报文的TCP头部的push字段置1。

**四、recv行为**

        默认情况下，recv的功能是从接收缓冲区读取(其实就是拷贝)指定长度的数据。如果将接收缓冲区大小设为0，recv将直接从协议缓冲区(滑动窗口区)读取数据，避免了数据从协议缓冲区到接收缓冲区的拷贝。recv返回的条件有两种：

      1. recv函数传入的应用层接收缓冲区已经读满

      2. 协议层接收到push字段为1的TCP报文，此时recv返回值为实际接收的数据长度

        协议层收到TCP数据包后(保存在滑动窗口区)，本方的滑动窗口合拢（窗口值减小）；当协议层将数据拷贝到接收缓冲区(滑动窗口区—>接收缓冲区)，或者应用层调用recv接收数据(接收缓冲区—>应用层缓冲区，滑动窗口区—>应用层缓冲区)后，本方的滑动窗口张开(窗口值增大)。收到数据更新window后，协议层向对方发送ACK确认。

        协议层的数据接收动作完全由发送动作驱动，是一个被动行为。在应用层没有任何干涉行为的情况下（比如recv操作等），协议层能够接收并保存的最大数据大小是窗口大小与接收缓冲区大小之和。Windows系统的窗口大小默认是64K，接收缓冲区默认为8K，所以默认情况下协议层最多能够被动接收并保存72K的数据。

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

http://blog.csdn.net/wjtxt/article/details/6598925

TCP连接关闭的问题：

 从TCP协议角度来看，一个已建立的TCP连接有两种关闭方式，一种是正常关闭，即四次挥手关闭连接；还有一种则是异常关闭，我们通常称之为连接重置（RESET)。

        首先说一下正常关闭时四次挥手的状态变迁，关闭连接的主动方状态变迁是FIN\_WAIT\_1->FIN\_WAIT\_2->TIME\_WAIT，而关闭连接的被对方的状态变迁是CLOSE\_WAIT->LAST\_ACK->TIME\_WAIT。在四次挥手过程中ACK包都是协议栈自动完成的，而FIN包则必须由应用层通过closesocket或shutdown主动发送，通常连接正常关闭后，recv会得到返回值0，send会得到错误码10058。

        除此之外，在我们的日常应用中，连接异常关闭的情况也很多。比如应用程序被强行关闭、本地网络突然中断(禁用网卡、网线拔出)、程序处理不当等都会导致连接重置，连接重置时将会产生RST包，同时网络络缓冲区中未接收（发送）的数据都将丢失。连接重置后，本方send或recv会得到错误码10053(closesocket时是10038)，对方recv会得到错误码10054，send则得到错误码10053(closesocket时是10054)。

        操作系统为我们提供了两个函数来关闭一个TCP连接，分别是closesocket和shutdown。通常情况下，closesocket会向对方发送一个FIN包，但是也有例外。比如有一个工作线程正在调用recv接收数据，此时外部调用closesocket，会导致连接重置，同时向对方发送一个RST包，这个RST包是由本方主动产生的。

        shutdown可以用来关闭指定方向的连接，该函数接收两个参数，一个是套接字，另一个是关闭的方向，可用值为SD\_SEND,SD\_RECEIVE和SD\_BOTH。方向取值为SD\_SEND时，无论socket处于什么状态(recv阻塞，或空闲状态)，都会向对方发送一个FIN包，注意这点与closesocket的区别。此时本方进入FIN\_WAIT\_2状态，对方进入CLOSE\_WAIT状态，本方依然可以调用recv接收数据；方向取值为SD\_RECEIVE时，双发连接状态没有改变，依然处于ESTABLISHED状态，本方依然可以send数据，但是，如果对方再调用send方法，连接会被立即重置，同时向对方发送一个RST包，这个RST包是被动产生的，这点注意与closesocket的区别。