[再次深入理解TCP网络编程中的send和recv](http://blog.csdn.net/yusiguyuan/article/details/21439719)

标签： [网络编程](http://www.csdn.net/tag/%e7%bd%91%e7%bb%9c%e7%bc%96%e7%a8%8b)[c](http://www.csdn.net/tag/c)

2014-03-18 08:24 4084人阅读 [评论](http://blog.csdn.net/yusiguyuan/article/details/21439719#comments)(2) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/yusiguyuan/article/details/21439719#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

linux网络编程（76） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

目录[(?)[+]](http://blog.csdn.net/yusiguyuan/article/details/21439719)

本篇我们用一个测试机上的阻塞socket实例来说明主题。文章中所有图都是在测试系统上现截取的。

**需要理解的3个概念**

**1. TCP socket的buffer**

每个TCP socket在内核中都有一个发送缓冲区和一个接收缓冲区，TCP的全双工的工作模式以及TCP的流量(拥塞)控制便是依赖于这两个独立的buffer以及buffer的填充状态。接收缓冲区把数据缓存入内核，应用进程一直没有调用recv()进行读取的话，此数据会一直缓存在相应socket的接收缓冲区内。再啰嗦一点，不管进程是否调用recv()读取socket，对端发来的数据都会经由内核接收并且缓存到socket的内核接收缓冲区之中。recv()所做的工作，就是把内核缓冲区中的数据拷贝到应用层用户的buffer里面，并返回，仅此而已。进程调用send()发送的数据的时候，最简单情况（也是一般情况），将数据拷贝进入socket的内核发送缓冲区之中，然后send便会在上层返回。换句话说，send（）返回之时，数据不一定会发送到对端去（和write写文件有点类似），send()仅仅是把应用层buffer的数据拷贝进socket的内核发送buffer中，发送是TCP的事情，和send其实没有太大关系。接收缓冲区被TCP用来缓存网络上来的数据，一直保存到应用进程读走为止。对于TCP，如果应用进程一直没有读取，接收缓冲区满了之后，发生的动作是：收端通知发端，接收窗口关闭（win=0）。这个便是滑动窗口的实现。保证TCP套接口接收缓冲区不会溢出，从而保证了TCP是可靠传输。因为对方不允许发出超过所通告窗口大小的数据。 这就是TCP的流量控制，如果对方无视窗口大小而发出了超过窗口大小的数据，则接收方TCP将丢弃它。  
查看测试机的socket发送缓冲区大小，cat /proc/sys/net/ipv4/tcp\_wmem

第一个值是一个限制值，socket发送缓存区的最少字节数；  
第二个值是默认值；  
第三个值是一个限制值，socket发送缓存区的最大字节数；  
根据实际测试,发送缓冲区的尺寸在默认情况下的全局设置是16384字节，即16k。  
在测试系统上，发送缓存默认值是16k。  
proc文件系统下的值和sysctl中的值都是全局值，应用程序可根据需要在程序中使用setsockopt（）对某个socket的发送缓冲区尺寸进行单独修改，详见文章《深入浅出TCP中的send和recv》，不过这都是题外话。

**2. 接收窗口（滑动窗口）**

TCP连接建立之时的收端的初始接受窗口大小是14600，细节如图2所示（129是收端，130是发端）

http://blog.chinaunix.net/attachment/201309/25/29075379_1380094882G3g6.jpg

图2

接收窗口是TCP中的滑动窗口，TCP的收端用这个接受窗口----win=14600，通知发端，我目前的接收能力是14600字节。  
后续发送过程中，收端会不断的用ACK（ACK的全部作用请参照博文《TCP之ACK发送情景》）通知发端自己的接收窗口的大小状态，如图3，而发端发送数据的量，就根据这个接收窗口的大小来确定，发端不会发送超过收端接收能力的数据量。这样就起到了一个流量控制的的作用。

http://blog.chinaunix.net/attachment/201309/25/29075379_1380095372uU0O.jpg

图3

图3说明  
21,22两个包都是收端发给发端的ACK包  
第21个包，收端确认收到的前7240个字节数据，7241的意思是期望收到的包从7241号开始，序号加了1.同时，接收窗口从最初的14656（如图2）经过慢启动阶段增加到了现在的29120。用来表明现在收端可以接收29120个字节的数据，而发端看到这个窗口通告，在没有收到新的ACK的时候，发端可以向收端发送29120字节这么多数据。  
第22个包，收端确认收到的前8688个字节数据，并通告自己的接收窗口继续增长为32000这么大。

**3. 单个TCP的负载量和MSS的关系**

MSS在以太网上通常大小是1460字节，而我们在后续发送过程中的单个TCP包的最大数据承载量是1448字节，这二者的关系可以参考博文《TCP之1460MSS和1448负载》。

**实例详解send()**

实例功能说明：接收端129作为客户端去连接发送端130，连接上之后并不调用recv（）接收，而是sleep（1000），把进程暂停下来，不让进程接收数据。内核会缓存数据至接收缓冲区。发送端作为服务器接收TCP请求之后，立即用ret = send(sock,buf,70k,0);这个C语句，向接收端发送70k数据。  
我们现在来观察这个过程。看看究竟发生了些什么事。wireshark抓包截图如下图4



图4

图4说明，包序号等同于时序  
1. 客户端sleep在recv（）之前，目的是为了把数据压入接收缓冲区。服务端调用"ret = send(sock,buf,70k,0);"这个C语句，向接收端发送70k数据。由于发送缓冲区大小16k，send（）无法将70k数据全部拷贝进发送缓冲区，故先拷贝16k进入发送缓冲区，下层发送缓冲区中有数据要发送，内核开始发送。上层send（）在应用层处于阻塞状态；  
2. 11号TCP包，发端从这儿开始向收端发送1448个字节的数据；  
3. 12号TCP包，发端没有收到之前发送的1448个数据的ACK包，仍然继续向收端发送1448个字节的数据；  
4. 13号TCP包，收端向发端发送1448字节的确认包，表明收端成功接收总共1448个字节。此时收端并未调用recv（）读取，目前发送缓冲区中被压入1448字节。由于处于慢启动状态，win接收窗口持续增大，表明接受能力在增加，吞吐量持续上升；  
5. 14号TCP包，收端向发端发送2896字节的确认包，表明收端成功接收总共2896个字节。此时收端并未调用recv（）读取，目前发送缓冲区中被压入2896字节。由于处于慢启动状态，win接收窗口持续增大，表明接受能力在增加，吞吐量持续上升；  
6. 15号TCP包，发端继续向收端发送1448个字节的数据；  
7. 16号TCP包，收端向发端发送4344字节的确认包，表明收端成功接收总共4344个字节。此时收端并未调用recv（）读取，目前发送缓冲区中被压入4344字节。由于处于慢启动状态，win接收窗口持续增大，表明接受能力在增加，吞吐量持续上升；  
8. 从这儿开始，我略去很多包，过程类似上面过程。同时，由于不断的发送出去的数据被收端用ACK确认，发送缓冲区的空间被逐渐腾出空地，send（）内部不断的把应用层buf中的数据向发送缓冲区拷贝，从而不断的发送，过程重复。70k数据并没有被完全送入内核,send()不管是否发送出去，send不管发送出去的是否被确认，send（）只关心buf中的数据有没有被全部送往发送缓冲区。如果buf中的数据没有被全部送往发送缓冲区，send（）在应用层阻塞，负责等待发送缓冲区中有空余空间的时候，逐步拷贝buf中的数据；如果buf中的数据被全部拷入发送缓冲区，send（）立即返回。  
9. 经过慢启动阶段接收窗口增大到稳定阶段，TCP吞吐量升高到稳定阶段，收端一直处于sleep状态，没有调用recv（）把内核中接收缓冲区中的数据拷贝到应用层去，此时收端的接收缓冲区中被压入大量数据；  
10. 66号、67号TCP数据包，发端继续向收端发送数据；  
11. 68号TCP数据包，收端发送ACK包确认接收到的数据，ACK=62265表明收端已经收到62265字节的数据，这些数据目前被压在收端的接收缓冲区中。win=3456，比较之前的16号TCP包的win=23296,表明收端的窗口已经处于收缩状态，收端的接收缓冲区中的数据迟迟未被应用层读走，导致接收缓冲区空间吃紧，故收缩窗口，控制发送端的发送量，进行流量控制；  
12. 69号、70号TCP数据包，发端在接收窗口允许的数据量的范围内，继续向收端发送2段1448字节长度的数据；  
13. 71号TCP数据包，至此，收端已经成功接收65160字节的数据，全部被压在接收缓冲区之中，接收窗口继续收缩，尺寸为1600字节；  
14. 72号TCP数据包，发端在接收窗口允许的数据量的范围内，继续向收端发送1448字节长度的数据；  
15. 73号TCP数据包，至此，收端已经成功接收66609字节的数据，全部被压在接收缓冲区之中，接收窗口继续收缩，尺寸为192字节。  
16. 74号TCP数据包，和我们这个例子没有关系，是别的应用发送的包；  
17. 75号TCP数据包，发端在接收窗口允许的数据量的范围内，向收端发送192字节长度的数据；  
18. 76号TCP数据包，至此，收端已经成功接收66609字节的数据，全部被压在接收缓冲区之中，win=0接收窗口关闭，接收缓冲区满，无法再接收任何数据；  
19. 77号、78号、79号TCP数据包，由keepalive触发的数据包，响应的ACK持有接收窗口的状态win=0，另外，ACK=66801表明接收端的接收缓冲区中积压了66800字节的数据。  
20. 从以上过程，我们应该熟悉了滑动窗口通告字段win所说明的问题，以及ACK确认数据等等。现在可得出一个结论，接收端的接收缓存尺寸应该是66800字节（此结论并非本篇主题）。  
send（）要发送的数据是70k，现在发出去了66800字节，发送缓存中还有16k，应用层剩余要拷贝进内核的数据量是N=70k-66800-16k。接收端仍处于sleep状态，无法recv（）数据，这将导致接收缓冲区一直处于积压满的状态，窗口会一直通告0(win=0)。发送端在这样的状态下彻底无法发送数据了，send（）的剩余数据无法继续拷贝进内核的发送缓冲区，最终导致send（）被阻塞在应用层；  
21. send（）一直阻塞中。。。  
  
图4和send（）的关系说明完毕。  
那什么时候send返回呢？有3种返回场景

**send（）返回场景**

**场景1，我们继续图4这个例子，不过这儿开始我们就跳出图4所示的过程了**

22. 接收端sleep（1000）到时间了，进程被唤醒，代码片段如图5

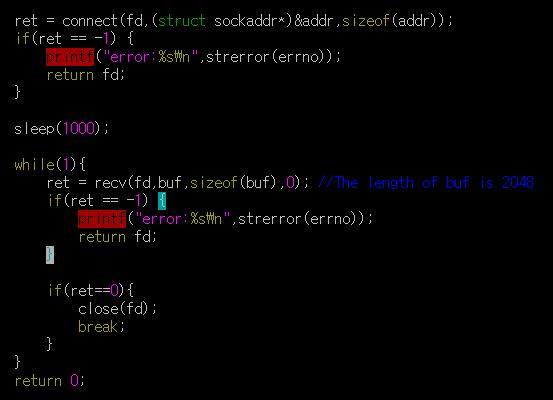


图5  
随着进程不断的用"recv（fd,buf,2048,0）;"将数据从内核的接收缓冲区拷贝至应用层的buf，在使用win=0关闭接收窗口之后，现在接收缓冲区又逐渐恢复了缓存的能力，这个条件下，收端会主动发送携带"win=n（n&gt;0）"这样的ACK包去通告发送端接收窗口已打开；

23. 发端收到携带"win=n(n>0)"这样的ACK包之后，开始继续在窗口运行的数据量范围内发送数据。发送缓冲区的数据被发出；  
24. 收端继续接收数据，并用ACK确认这些数据；  
25. 发端收到ACK，可以清理出一些发送缓冲区空间，应用层send（）的剩余数据又可以被不断的拷贝进内核的发送缓冲区；  
26. 不断重复以上发送过程；  
27. send（）的70k数据全部进入内核，send（）成功返回。  
  
**场景2，我们继续图4这个例子，不过这儿开始我们就跳出图4所示的过程了**  
22. 收端进程或者socket出现问题，给发端发送一个RST，请参考博文《》；  
23. 内核收到RST，send返回-1。  
  
**场景3，和以上例子没关系**  
连接上之后，马上send（1k），这样，发送的数据肯定可以一次拷贝进入发送缓冲区，send（）拷贝完数据立即成功返回。

**send()发送结论**

其实场景1和场景2说明一个问题  
send（）只是负责拷贝，拷贝完立即返回，不会等待发送和发送之后的ACK。如果socket出现问题，RST包被反馈回来。在RST包返回之时，如果send（）还没有把数据全部放入内核或者发送出去，那么send（）返回-1，errno被置错误值；如果RST包返回之时，send（）已经返回，那么RST导致的错误会在下一次send（）或者recv（）调用的时候被立即返回。  
场景3完全说明send（）只要完成拷贝就成功返回，如果发送数据的过程中出现各种错误，下一次send（）或者recv（）调用的时候被立即返回。

**概念上容易疑惑的地方**

1. TCP协议本身是为了保证可靠传输,并不等于应用程序用tcp发送数据就一定是可靠的，必须要容错；  
2. send（）和recv（）没有固定的对应关系，不定数目的send()可以触发不定数目的recv（），这话不专业，但是还是必须说一下，初学者容易疑惑；  
3. 关键点，send（）只负责拷贝，拷贝到内核就返回，我通篇在说拷贝完返回，很多文章中说send（）在成功发送数据后返回，成功发送是说发出去的东西被ACK确认过。send（）只拷贝，不会等ACK；  
4. 此次send（）调用所触发的程序错误，可能会在本次返回，也可能在下次调用网络IO函数的时候被返回。

实际上理解了阻塞式的，就能理解非阻塞的。