<http://bbs.chinaunix.net/forum.php?mod=viewthread&tid=969374>

这是根据自己的笔记整理的,如有错误,欢迎指出来.  
  
  
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:  
  
**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,而是继续调用send,那么第一次会像往常一样继续填充缓存区,然后返回,但如果再次调用send,进程会收到SIGPIPE信号,该信号的默认响应动作是退出进程.  
  
**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  
  
  
  
  
所以,要想编写优秀的socket程序也是很不容易的.特别是在为应用做优化时,很多工作都非常的烦琐.