1、修改用户进程可打开文件数限制  
在Linux平台上，无论编写客户端程序还是服务端程序，在进行高并发TCP连接处理时，最高的并发数量都要受到系统对用户单一进程同时可打开文件数量的限制(这是因为系统为每个TCP连接都要创建一个socket句柄，每个socket句柄同时也是一个文件句柄)。可使用ulimit命令查看系统允许当前用户进程打开的文件数限制：  
[speng@as4 ~]$ ulimit -n  
1024  
这表示当前用户的每个进程最多允许同时打开1024个文件，这1024个文件中还得除去每个进程必然打开的标准输入，标准输出，标准错误，服务器监听 socket，进程间通讯的unix域socket等文件，那么剩下的可用于客户端socket连接的文件数就只有大概1024-10=1014个左右。也就是说缺省情况下，基于Linux的通讯程序最多允许同时1014个TCP并发连接。  
对于想支持更高数量的TCP并发连接的通讯处理程序，就必须修改Linux对当前用户的进程同时打开的文件数量的软限制(soft limit)和硬限制(hardlimit)。其中软限制是指Linux在当前系统能够承受的范围内进一步限制用户同时打开的文件数；硬限制则是根据系统硬件资源状况(主要是系统内存)计算出来的系统最多可同时打开的文件数量。通常软限制小于或等于硬限制。  
修改上述限制的最简单的办法就是使用ulimit命令：  
[speng@as4 ~]$ ulimit -n  
上述命令中，在中指定要设置的单一进程允许打开的最大文件数。如果系统回显类似于“Operation notpermitted”之类的话，说明上述限制修改失败，实际上是因为在中指定的数值超过了Linux系统对该用户打开文件数的软限制或硬限制。因此，就需要修改Linux系统对用户的关于打开文件数的软限制和硬限制。  
第一步，修改/etc/security/limits.conf文件，在文件中添加如下行：  
speng soft nofile 10240  
speng hard nofile 10240  
其中speng指定了要修改哪个用户的打开文件数限制，可用’\*'号表示修改所有用户的限制；soft或hard指定要修改软限制还是硬限制；10240则指定了想要修改的新的限制值，即最大打开文件数(请注意软限制值要小于或等于硬限制)。修改完后保存文件。  
第二步，修改/etc/pam.d/login文件，在文件中添加如下行：  
session required /lib/security/pam\_limits.so  
这是告诉Linux在用户完成系统登录后，应该调用pam\_limits.so模块来设置系统对该用户可使用的各种资源数量的最大限制(包括用户可打开的最大文件数限制)，而pam\_limits.so模块就会从/etc/security/limits.conf文件中读取配置来设置这些限制值。修改完后保存此文件。  
第三步，查看Linux系统级的最大打开文件数限制，使用如下命令：  
[speng@as4 ~]$ cat /proc/sys/fs/file-max  
12158  
这表明这台Linux系统最多允许同时打开(即包含所有用户打开文件数总和)12158个文件，是Linux系统级硬限制，所有用户级的打开文件数限制都不应超过这个数值。通常这个系统级硬限制是Linux系统在启动时根据系统硬件资源状况计算出来的最佳的最大同时打开文件数限制，如果没有特殊需要，不应该修改此限制，除非想为用户级打开文件数限制设置超过此限制的值。修改此硬限制的方法是修改/etc/rc.local脚本，在脚本中添加如下行：  
echo 22158 > /proc/sys/fs/file-max  
这是让Linux在启动完成后强行将系统级打开文件数硬限制设置为22158。修改完后保存此文件。  
完成上述步骤后重启系统，一般情况下就可以将Linux系统对指定用户的单一进程允许同时打开的最大文件数限制设为指定的数值。如果重启后用 ulimit-n命令查看用户可打开文件数限制仍然低于上述步骤中设置的最大值，这可能是因为在用户登录脚本/etc/profile中使用ulimit -n命令已经将用户可同时打开的文件数做了限制。由于通过ulimit-n修改系统对用户可同时打开文件的最大数限制时，新修改的值只能小于或等于上次 ulimit-n设置的值，因此想用此命令增大这个限制值是不可能的。所以，如果有上述问题存在，就只能去打开/etc/profile脚本文件，在文件中查找是否使用了ulimit-n限制了用户可同时打开的最大文件数量，如果找到，则删除这行命令，或者将其设置的值改为合适的值，然后保存文件，用户退出并重新登录系统即可。  
通过上述步骤，就为支持高并发TCP连接处理的通讯处理程序解除关于打开文件数量方面的系统限制。  
2、修改网络内核对TCP连接的有关限制（参考对比下篇文章“优化内核参数”）  
在Linux上编写支持高并发TCP连接的客户端通讯处理程序时，有时会发现尽管已经解除了系统对用户同时打开文件数的限制，但仍会出现并发TCP连接数增加到一定数量时，再也无法成功建立新的TCP连接的现象。出现这种现在的原因有多种。  
第一种原因可能是因为Linux网络内核对本地端口号范围有限制。此时，进一步分析为什么无法建立TCP连接，会发现问题出在connect()调用返回失败，查看系统错误提示消息是“Can’t assign requestedaddress”。同时，如果在此时用tcpdump工具监视网络，会发现根本没有TCP连接时客户端发SYN包的网络流量。这些情况说明问题在于本地Linux系统内核中有限制。其实，问题的根本原因在于Linux内核的TCP/IP协议实现模块对系统中所有的客户端TCP连接对应的本地端口号的范围进行了限制(例如，内核限制本地端口号的范围为1024~32768之间)。当系统中某一时刻同时存在太多的TCP客户端连接时，由于每个TCP客户端连接都要占用一个唯一的本地端口号(此端口号在系统的本地端口号范围限制中)，如果现有的TCP客户端连接已将所有的本地端口号占满，则此时就无法为新的TCP客户端连接分配一个本地端口号了，因此系统会在这种情况下在connect()调用中返回失败，并将错误提示消息设为“Can’t assignrequested address”。有关这些控制逻辑可以查看Linux内核源代码，以linux2.6内核为例，可以查看tcp\_ipv4.c文件中如下函数：  
static int tcp\_v4\_hash\_connect(struct sock \*sk)  
请注意上述函数中对变量sysctl\_local\_port\_range的访问控制。变量sysctl\_local\_port\_range的初始化则是在tcp.c文件中的如下函数中设置：  
void \_\_init tcp\_init(void)  
内核编译时默认设置的本地端口号范围可能太小，因此需要修改此本地端口范围限制。  
第一步，修改/etc/sysctl.conf文件，在文件中添加如下行：  
net.ipv4.ip\_local\_port\_range = 1024 65000  
这表明将系统对本地端口范围限制设置为1024~65000之间。请注意，本地端口范围的最小值必须大于或等于1024；而端口范围的最大值则应小于或等于65535。修改完后保存此文件。  
第二步，执行sysctl命令：  
[speng@as4 ~]$ sysctl -p  
如果系统没有错误提示，就表明新的本地端口范围设置成功。如果按上述端口范围进行设置，则理论上单独一个进程最多可以同时建立60000多个TCP客户端连接。  
第二种无法建立TCP连接的原因可能是因为Linux网络内核的IP\_TABLE防火墙对最大跟踪的TCP连接数有限制。此时程序会表现为在 connect()调用中阻塞，如同死机，如果用tcpdump工具监视网络，也会发现根本没有TCP连接时客户端发SYN包的网络流量。由于 IP\_TABLE防火墙在内核中会对每个TCP连接的状态进行跟踪，跟踪信息将会放在位于内核内存中的conntrackdatabase中，这个数据库的大小有限，当系统中存在过多的TCP连接时，数据库容量不足，IP\_TABLE无法为新的TCP连接建立跟踪信息，于是表现为在connect()调用中阻塞。此时就必须修改内核对最大跟踪的TCP连接数的限制，方法同修改内核对本地端口号范围的限制是类似的：  
第一步，修改/etc/sysctl.conf文件，在文件中添加如下行：  
net.ipv4.ip\_conntrack\_max = 10240  
这表明将系统对最大跟踪的TCP连接数限制设置为10240。请注意，此限制值要尽量小，以节省对内核内存的占用。  
第二步，执行sysctl命令：  
[speng@as4 ~]$ sysctl -p  
如果系统没有错误提示，就表明系统对新的最大跟踪的TCP连接数限制修改成功。如果按上述参数进行设置，则理论上单独一个进程最多可以同时建立10000多个TCP客户端连接。  
3、使用支持高并发网络I/O的编程技术  
在Linux上编写高并发TCP连接应用程序时，必须使用合适的网络I/O技术和I/O事件分派机制。  
可用的I/O技术有同步I/O，非阻塞式同步I/O(也称反应式I/O)，以及异步I/O。在高TCP并发的情形下，如果使用同步I/O，这会严重阻塞程序的运转，除非为每个TCP连接的I/O创建一个线程。但是，过多的线程又会因系统对线程的调度造成巨大开销。因此，在高TCP并发的情形下使用同步 I/O是不可取的，这时可以考虑使用非阻塞式同步I/O或异步I/O。非阻塞式同步I/O的技术包括使用select()，poll()，epoll等机制。异步I/O的技术就是使用AIO。  
从I/O事件分派机制来看，使用select()是不合适的，因为它所支持的并发连接数有限(通常在1024个以内)。如果考虑性能，poll()也是不合适的，尽管它可以支持的较高的TCP并发数，但是由于其采用“轮询”机制，当并发数较高时，其运行效率相当低，并可能存在I/O事件分派不均，导致部分TCP连接上的I/O出现“饥饿”现象。而如果使用epoll或AIO，则没有上述问题(早期Linux内核的AIO技术实现是通过在内核中为每个 I/O请求创建一个线程来实现的，这种实现机制在高并发TCP连接的情形下使用其实也有严重的性能问题。但在最新的Linux内核中，AIO的实现已经得到改进)。  
综上所述，在开发支持高并发TCP连接的Linux应用程序时，应尽量使用epoll或AIO技术来实现并发的TCP连接上的I/O控制，这将为提升程序对高并发TCP连接的支持提供有效的I/O保证。

内核参数sysctl.conf的优化

/etc/sysctl.conf 是用来控制linux网络的配置文件，对于依赖网络的程序（如web服务器和cache服务器）非常重要，RHEL默认提供的最好调整。

推荐配置（把原/etc/sysctl.conf内容清掉，把下面内容复制进去）：  
net.ipv4.ip\_local\_port\_range = 1024 65536  
net.core.rmem\_max=16777216  
net.core.wmem\_max=16777216  
net.ipv4.tcp\_rmem=4096 87380 16777216  
net.ipv4.tcp\_wmem=4096 65536 16777216  
net.ipv4.tcp\_fin\_timeout = 10  
net.ipv4.tcp\_tw\_recycle = 1  
net.ipv4.tcp\_timestamps = 0  
net.ipv4.tcp\_window\_scaling = 0  
net.ipv4.tcp\_sack = 0  
net.core.netdev\_max\_backlog = 30000  
net.ipv4.tcp\_no\_metrics\_save=1  
net.core.somaxconn = 262144  
net.ipv4.tcp\_syncookies = 0  
net.ipv4.tcp\_max\_orphans = 262144  
net.ipv4.tcp\_max\_syn\_backlog = 262144  
net.ipv4.tcp\_synack\_retries = 2  
net.ipv4.tcp\_syn\_retries = 2

这个配置参考于cache服务器varnish的推荐配置和SunOne 服务器系统优化的推荐配置。

varnish调优推荐配置的地址为：http://varnish.projects.linpro.no/wiki/Performance

不过varnish推荐的配置是有问题的，实际运行表明“net.ipv4.tcp\_fin\_timeout = 3”的配置会导致页面经常打不开；并且当网友使用的是IE6浏览器时，访问网站一段时间后，所有网页都会打不开，重启浏览器后正常。可能是国外的网速快吧，我们国情决定需要调整“net.ipv4.tcp\_fin\_timeout = 10”，在10s的情况下，一切正常（实际运行结论）。

修改完毕后，执行：  
/sbin/sysctl -p /etc/sysctl.conf  
/sbin/sysctl -w net.ipv4.route.flush=1

命令生效。为了保险起见，也可以reboot系统。

调整文件数：  
linux系统优化完网络必须调高系统允许打开的文件数才能支持大的并发，默认1024是远远不够的。

执行命令：  
Shell代码  
echo ulimit -HSn 65536 >> /etc/rc.local  
echo ulimit -HSn 65536 >>/root/.bash\_profile  
ulimit -HSn 65536