C1000k 实践报告
草稿 V0.1

Max

1. 前言

2014 转眼已经过去大半，收获寥寥，所以还是写点东西作为总结。 这是一篇实践性的文档，主要是记录实践的过程。文中绝大部分内容都来自文档和前人总结过的经验，我只是按照自己的理解进行了实践。如文中有任何问题都是我的原因造成，请直接发信给我 ppmsn2005 # gmail.com。谢谢！

本文及相关源码会持续发布在 github 上，你可以在 https://github.com/xiaojiaqi/C1000kPracticeGuide 找到最新的版本。

综述

本文主要描述的是如果利用一台被淘汰的服务器（二手价格在 1000 元以下）上实践 C1000K 的过程。需要申明的一点是本文是在一个实验室环境内中实践的过程，和真正在线环境 还存在不小的差异，此外因为业务类型和复杂程度不一样，业务对系统的要求会有各种偏重，所以不能简单的认为这个方案对所有的业务系统都适用，并且本文描述的只是一个最低的业务要求。因为个人精力和硬件性能条件限制，我没有能够对此继续深入。如果希望在线支持百万用户在线，我认为还需要在这个基础上继续优化，请读者注意。

对读者的背景要求，读者需要基本的 Linux 知识，对 TCP/IP 有一定的了解，编写过基础的服务程序，懂一些基本的网络 API。

首先此文的诞生必须感谢一个人：余锋。余老师 在 2010 年一篇非常精彩的《C1000K 高性能服务器构建技术》， 给我们揭开了一个新的世界。本文基本以此文作为主线进行了实践。

C1000K

从 C1000K 说起 ，C1000K 是什么意思？为了搞清楚这个问题，必须先提起”The C10K problem” (http://www.kegel.com/c10k.html)， 顾名思义 10K 就是 1 万，这篇文章可以说是高性能服务器开发的一个标志性文档，它讨论的就是单机为 1 万个连接提供服务这个问题，当时因为硬件和软件的限制，单机 1 万还是一个非常值得挑战的目标。但是时光荏苒，随着硬件和软件的飞速发展，单机 1 万的目标已经变成了最简单不过的事情。现在用任何一种主流语言都能提供单机 1 万的并发处理的能力。所以现在目标早已提高了 100 倍，变成 C1000k，也就是一台服务器为 100 万连接提供服务 。 国外公司 whatapps 的曾经分享过相关内容，他们在 2011 年 9 月宣称完成了单机 100 万用户支持（他们最后做到了 C2000k，而且

是实打实的在线用户）。所以 C10M，C100M 才是今后继续努力的方向，让我们继续努力吧。

目标

在实践以前，我们需要设定一个目标：如何算实现了 C1000k 的实践。

因为条件的限制，我把目标设定为，在一台普通的物理服务器上提供单机 100 万的连接，同时这台服务器每 10 秒就向所有的客户端发送 500 字节长度的消息。要求能稳定支持连接，发送数据稳定，客户端接收正常。

对于一个普通的程序的目标又是什么呢？作为一个程序员，如果能从零开始编写一个框架（程序），并用它达成上面的目标，我想对自己也是一种提高。因为你可以在系统优化，服务器架构设计等方面提高自己。而不仅仅是使用开源的 Nginx 这样的服务器来做到这一切。本文也提供了一个最简单的网络架构，来实现这个目标。

系统准备

硬件方面

因为条件限制，我使用的 一台非常老的 DELL 2950 的服务器。所有的测试都在这台服务器上进行。我相信现在的主流硬件都应该比它强，所以大家在实践的条件上应该不成问题。

服务器

首先查看服务器的信息

```
root@lotus:/# dmidecode | grep "Product Name"
        Product Name: PowerEdge 2950
        Product Name: 0N192H
```

显示主机为 2950 主机， 类型编号 0N192H ，这应该是一台 2007 年的产品。属于被淘汰的产品。

CPU

然后检查 CPU 的规格

```
root@lotus:/# cat /proc/cpuinfo | grep name | awk -F: '{print $(NF)}' | uniq -c
      4  Intel(R) Xeon(R) CPU           E5420  @ 2.50GHz
```

```
root@lotus:/# cat /proc/cpuinfo | grep physical |grep -v address| uniq -c
      4 physical id     : 0
```

以上信息显示了服务器使用了一块 4 核的 E5420，而且是一个物理硬核的 CPU(这款 CPU 目前淘宝的二手价格在 300 元左右)。

内存

检查内存大小

```
root@lotus:~# cat /proc/meminfo
MemTotal:       16428352 kB
MemFree:         2693392 kB
Buffers:         5510456 kB
Cached:          4586724 kB
SwapCached:         1296 kB
Active:          5833424 kB
```

说明机器内配备了 16G 内存。

网卡

```
root@lotus:/# dmesg | grep -i eth
[    0.137710] ACPI Error: Method parse/execution failed [\_SB_._OSC] (Node ffff880429462488), AE_NOT_FOUND (2013051
7/psparse-536)
[    1.124198] bnx2: Broadcom NetXtreme II Gigabit Ethernet Driver bnx2 v2.2.3 (June 27, 2012)
[    1.124911] bnx2 0000:03:00.0 eth0: Broadcom NetXtreme II BCM5708 1000Base-T (B2) PCI-X 64-bit 133MHz found at me
m da000000, IRQ 16, node addr 00:1e:c9:f0:04:cd
[    1.133670] bnx2 0000:07:00.0 eth1: Broadcom NetXtreme II BCM5708 1000Base-T (B2) PCI-X 64-bit 133MHz found at me
m d6000000, IRQ 16, node addr 00:1e:c9:f0:04:cf
[    1.296447] e1000e 0000:08:00.0 eth2: (PCI Express:2.5GT/s:Width x4) 00:15:17:86:e8:06
[    1.296454] e1000e 0000:08:00.0 eth2: Intel(R) PRO/1000 Network Connection
[    1.296534] e1000e 0000:08:00.0 eth2: MAC: 0, PHY: 4, PBA No: C57721-005
[    1.472423] e1000e 0000:08:00.1 eth3: (PCI Express:2.5GT/s:Width x4) 00:15:17:86:e8:07
[    1.472429] e1000e 0000:08:00.1 eth3: Intel(R) PRO/1000 Network Connection
[    1.472510] e1000e 0000:08:00.1 eth3: MAC: 0, PHY: 4, PBA No: C57721-005
```

从信息中可以看到，服务器上一共有 4 块网卡。其中 Eth0，Eth1 是一块 Broadcom 5708 网卡。因为条件限制，在实践过程中只使用了一块网卡。

软件方面：

操作系统

机器使用的是 64 位的 Ubuntu 12.04

```
root@lotus:~# lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 12.04.4 LTS
Release:        12.04
Codename:       precise
```

内核版本

```
root@lotus:~# uname -a
Linux lotus 3.11.0-15-generic #25~precise1-Ubuntu SMP Thu Jan 30 17:39:31 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
```

其他相关软件版本

G++

```
root@lotus:~# g++ --version
g++ (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
Copyright (C) 2011 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Golang

```
root@lotus:~# go version
go version go1.3 linux/amd64
```

系统优化

系统优化是非常关键的一步，因为默认的系统设置并不是以高并发高负载的服务器做为目标的，所有配置需要进行修改，这样可以尽可能的使用操作系统和硬件的能力。

对于系统优化，我认为可以分为 2 个层面的优化

1. 根据高并发高负载的业务要求，对 Linux 的系统设置进行优化，如最大文件数目，TCP/IP 的设置参数等等。这个层级的优化经验，基本适用于大多数的服务器设置。
2. 根据硬件的特性，对硬件的设置进行优化。这个级别的要求更高，也更有正针对性。

Linux 系统设置

1. 提高文件数目上限

在 Linux 中 socket 被表示为一个文件描述符，默认的文件数目上限是 1024，当然这是远远不够的。 你需要做的是自然是提高文件打开上限。

首先确认现有系统的文件打开上限，找一台没有经过优化的服务器。

```
root@mongo1:~# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 7782
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 7782
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

从上面的输出，你可以看到现有的 open files 的数值为 1024。这意味着这台服务器只能同时为 1024 个用户提供服务。当然这是远远不够。

提高打开文件数目上限，有 2 个方法。临时的办法存在诸多的问题，我并不推荐。这里介绍的另一种比较好的办法

a. 修改 /etc/security/limits.conf
   添加以下信息

```
* hard nofile 1025500
* soft nofile 1025500
```

b. 修改 /etc/sysctl.conf

```
fs.file-max=1025500
```

c. 运行 sysctl -p 使修改生效

d. 重新登录并确认修改生效

```
root@lotus:/proc/sys# ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 128196
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1025500
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 128196
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

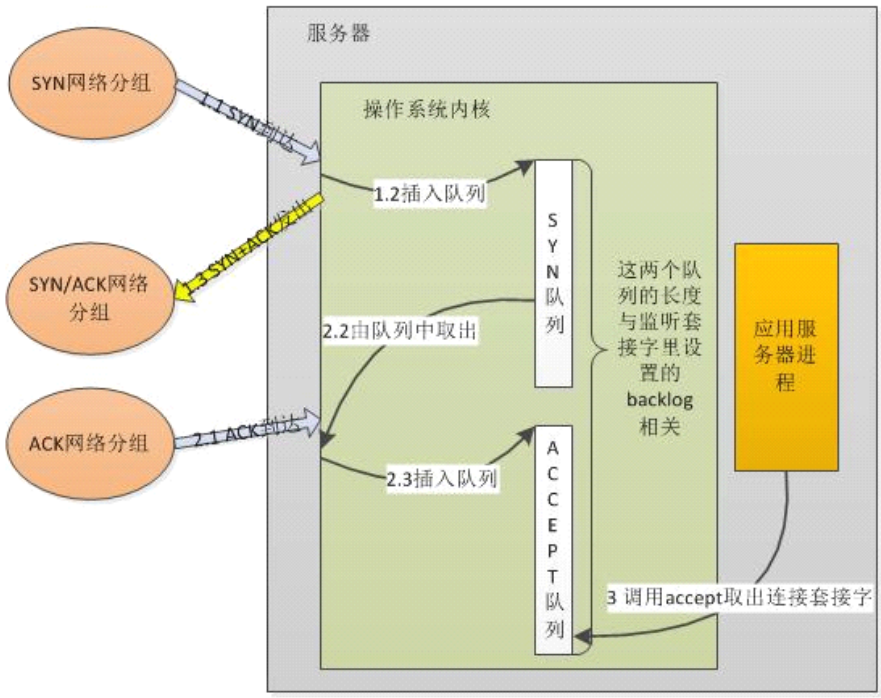
好了，现在文件打开上限已经被修改。下面我们进行 TCP/IP 部分的优化

2. TCP/IP 的优化

TCP/IP 的优化选项非常的多，而且 Linux 提供了非常友好的修改方法，使用起来非常方便。但是对于 TCP/IP 各个字段的意义，以及可能产生的后果，需要对 TCP/IP 本身有一个较为深入的理解。下面是我修改的一些字段，仅供参考。

- 修改 net.core.somaxconn 以及 net.core.tcp_max_syn_backlog

  要解释这个问题，需要对网络服务的 Accpet 有深入的了解，这里我借助陶辉老师关于高性能服务器系列博客里的一张图来解释这个选项。[http://taohui.org.cn/tcpperf1.html]



从这张图里可以看出， 如果有高并发的请求来进行连接，而我们的队列过小，客户端就直接连接失败。如果我们把队列扩大，那么我们的服务器就有机会把暂时处理不了的请求，暂存起来，慢慢处理。当然要处理大规模并发请求，还需要一些别的技巧。

如何确认修改已经生效了呢?

使用 ss -n -l 命令，检查 Send-Q 那一列，你就知道是否已经生效了。你可以看到默认值是 128 是不是非常的小。此外这个参数的修改 在 Nginx 等高性能服务器的配置里也有类似的参数。所以是一个很必要的参数。（这个参数还和 listen 的参数相关,所以代码部分也要注意）

```
State       Recv-Q Send-Q                 Local Address:Port                 Peer Address:Port
LISTEN      0      50                           *:139                             *:*
LISTEN      0      6000                         *:80                              *:*
LISTEN      0      128                         :::22                             :::*
LISTEN      0      128                          *:22                              *:*
LISTEN      0      128                  127.0.0.1:631                             *:*
LISTEN      0      128                        ::1:631                             :::*
LISTEN      0      10000                        *:8888                            *:*
LISTEN      0      128                  127.0.0.1:6010                            *:*
LISTEN      0      128                        ::1:6010                            :::*
LISTEN      0      128                  127.0.0.1:6011                            *:*
LISTEN      0      128                        ::1:6011                            :::*
LISTEN      0      128                  127.0.0.1:6012                            *:*
LISTEN      0      128                        ::1:6012                            :::*
LISTEN      0      50                           *:445                             *:*
```

- tcp_syncookies

开启 tcp_syncookie 可以防止 syn floor 攻击,同时在 syn_backlog 已满的情况下,不会抛弃 syn 包。推荐打开

- tcp_max_tw_buckets

修改系统中处于 timewait 状态的连接的数目。关于 timewait 状态,Steven 老师说过他是我们的朋友。但有时候我们需要关闭它。主要为了防>止对系统资源的占有。我把它设置为 10000

- tcp_tw_recycle

用于快速回收处于 timewait 的连接。但是它和 timestamp 一起作用时可能会导致同一个 NAT 过来的连接失败。关闭它。

- timestamps

为了避免它和 tcp_tw_recycle 一起导致问题, 我推荐关闭它。

- tcp_tw_reuse

允许将 TIME-WAIT sockets 重新用于新的 TCP 连接,使用下来效果不是很好。但是还是建议开启。

- tcp_fin_timeout

如果本方关闭连接，则它在 FIN_WAIT_2 状态的时间。建议改为 10。

- tcp_synack_retries

对于远端的连接请求 SYN,服务器对应的 ack 响应的数目。我把它设置为 10。

- tcp_keepalive_time tcp_keepalive_intvl tcp_keepalive_probes

这主要是为了解决 TCP 的 CLOSE_WAIT 问题,有人说它是 TCP 的癌症。一般来说 TCP 处于 CLOSE_WAIT 的状态,说明你的连接处于半连接状态,你已经无法收到对方的信息了，绝大多数的情况下,你需要离开关闭连接。如果你的代码出现问题,忘记关闭了这个连接(TCP 是双工的),那么资源就一直被泄漏着。还有一种情况则是,对方故意不收取你的数据,导致你最后的 fin 包

无法发送给对方。（如果希望重现这个攻击的过程,你可以写一个简单的客户端,连接上一台 Nginx 获取一个比较大的文件。在发送完请求以后，并不读取数据，那么服务端的这个连接,将在 tcp_keepalive_time 时间内无法被关闭。）修改这 3 个参数可以减少被攻击的几率。

tcp_keepalive_time
防止空连接攻击,可以缩小该值，建议改为 180
tcp_keepalive_intvl
当探测没有确认时，重新发送探测的频度。缺省是 75 秒。建议改为 30 秒
tcp_keepalive_probes
进行多少次探测,因为探测的间隔是按照指数级别增长,默认为 9 次。建议改为 5 次。

客户端相关

- 提高 cwnd

提高拥塞窗口大小。 拥塞窗口主要是为了解决网络拥塞的问题，但是默认的拥塞窗口会导致网络传输数据的启动速度比较低。打个比方就象普通汽车起步的时候，都是慢慢换档，速度提升自然比较慢。但是那些 F1 跑车都是一起步就向 200 公里的时速冲，所以跑车的速度快多了。提高了这个数值就是让你的 Linux 在起步阶段也快起来。国内公司如 TX, 在曾它的一次手机分享中也提到了这个修改。按照 google 的推荐,我建议将其改为 10.
修改方法如下（火丁笔记 http://huoding.com/2013/11/21/299）

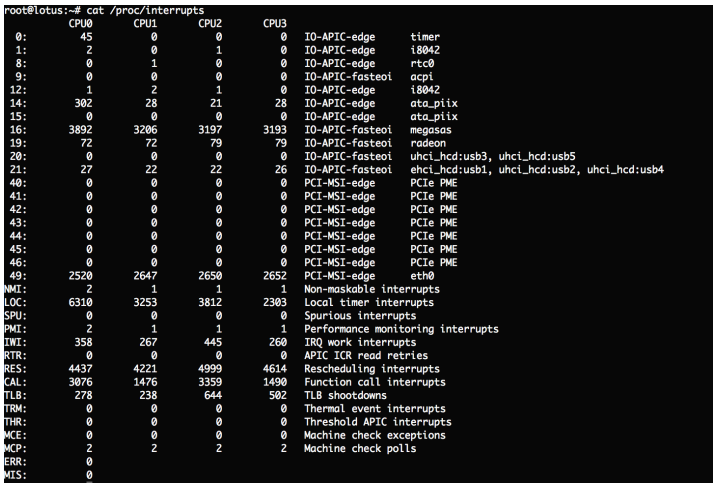
```
shell> ip route | while read p; do
           ip route change $p initcwnd 10;
       done
```

- ip_local_port_range

可用端口范围。很简单改到 1024 到 65535

针对硬件进行进行优化

首先我们需要确定网卡的 IRQ



```
root@lotus:~# cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  0:         45          0          0          0   IO-APIC-edge      timer
  1:          2          0          1          0   IO-APIC-edge      i8042
  8:          0          1          0          0   IO-APIC-edge      rtc0
  9:          0          0          0          0   IO-APIC-fasteoi   acpi
 12:          1          2          1          0   IO-APIC-edge      i8042
 14:        302         28         21         28   IO-APIC-edge      ata_piix
 15:          0          0          0          0   IO-APIC-edge      ata_piix
 16:       3892       3206       3197       3193   IO-APIC-fasteoi   megasas
 19:         72         72         79         79   IO-APIC-fasteoi   radeon
 20:          0          0          0          0   IO-APIC-fasteoi   uhci_hcd:usb3, uhci_hcd:usb5
 21:         27         22         22         26   IO-APIC-fasteoi   ehci_hcd:usb1, uhci_hcd:usb2, uhci_hcd:usb4
 40:          0          0          0          0   PCI-MSI-edge      PCIe PME
 41:          0          0          0          0   PCI-MSI-edge      PCIe PME
 42:          0          0          0          0   PCI-MSI-edge      PCIe PME
 43:          0          0          0          0   PCI-MSI-edge      PCIe PME
 44:          0          0          0          0   PCI-MSI-edge      PCIe PME
 45:          0          0          0          0   PCI-MSI-edge      PCIe PME
 46:          0          0          0          0   PCI-MSI-edge      PCIe PME
 49:       2520       2647       2650       2652   PCI-MSI-edge      eth0
NMI:          2          1          1          1   Non-maskable interrupts
LOC:       6310       3253       3812       2303   Local timer interrupts
SPU:          0          0          0          0   Spurious interrupts
PMI:          2          1          1          1   Performance monitoring interrupts
IWI:        358        267        445        260   IRQ work interrupts
RTR:          0          0          0          0   APIC ICR read retries
RES:       4437       4221       4999       4614   Rescheduling interrupts
CAL:       3076       1476       3359       1490   Function call interrupts
TLB:        278        238        644        502   TLB shootdowns
TRM:          0          0          0          0   Thermal event interrupts
THR:          0          0          0          0   Threshold APIC interrupts
MCE:          0          0          0          0   Machine check exceptions
MCP:          2          2          2          2   Machine check polls
ERR:          0
MIS:          0
```

从结果可以得知，网卡 eth0 的 IRQ 为 49

从硬件信息可知我的 CPU 为 4 核，那么我们可以把主设备的中断处理分配到 4 个核上。配置的办法如下

```
root@lotus:/proc/irq/49# echo 0-3 >  /proc/irq/49/smp_affinity_list
root@lotus:/proc/irq/49# cat smp_affinity_list
0-3
root@lotus:/proc/irq/49# cat  /proc/irq/49/smp_affinity
0f
```

完成以后，就能提高效果了吗？从我个人的经验看，的确各个 CPU 处理的中断数目比较均衡了。但这块网卡是单队列的网卡，还需要启动 RPS 。（注意 RPS 对 linux 内核版本有要求，需要高于 2.6.32，而我们的内核是 3.x,所以肯定会支持这个特性）

如何确定我的网卡是一块单队列的网卡呢？
首先 lspci -vvv 找到网卡相应的信息，按照 [http://blog.chinaunix.net/uid-10915175-id-3367864.html] 的说法
“Ethernet controller 的条目内容，如果有 MSI-X && Enable+ && TabSize > 1，则该网卡是多队列网卡
”，这条信息显示了是否是多队列，很不幸这是块单队列的网卡。

```
03:00.0 Ethernet controller: Broadcom Corporation NetXtreme II BCM5708 Gigabit Ethernet (rev 12)
        Subsystem: Dell Device 01b2
        Control: I/O- Mem+ BusMaster+ SpecCycle+ MemWINV+ VGASnoop- ParErr+ Stepping- SERR+ FastB2B- DisINTx+
        Status: Cap+ 66MHz+ UDF- FastB2B+ ParErr- DEVSEL=medium >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
        Latency: 32 (16000ns min), Cache Line Size: 64 bytes
        Interrupt: pin A routed to IRQ 49
        Region 0: Memory at da000000 (64-bit, non-prefetchable) [size=32M]
        Capabilities: [40] PCI-X non-bridge device
                Command: DPERE- ERO- RBC=512 OST=8
                Status: Dev=03:00.0 64bit+ 133MHz+ SCD- USC- DC=simple DMMRBC=512 DMOST=8 DMCRS=32 RSCEM- 266MHz- 533MHz-
        Capabilities: [48] Power Management version 2
                Flags: PMEClk- DSI- D1- D2- AuxCurrent=0mA PME(D0-,D1-,D2-,D3hot+,D3cold+)
                Status: D0 NoSoftRst- PME-Enable- DSel=0 DScale=1 PME-
        Capabilities: [50] Vital Product Data
                Product Name: Broadcom NetXtreme II Ethernet Controller
                Read-only fields:
                        [PN] Part number: BCM5708B2
                        [EC] Engineering changes: lon_mlk
                        [SN] Serial number: 0123456789
                        [MN] Manufacture ID: 31 34 65 34
                        [RV] Reserved: checksum good, 32 byte(s) reserved
                End
        Capabilities: [58] MSI: Enable+ Count=1/1 Maskable- 64bit+
                Address: 00000000fee0f00c  Data: 4172
        Kernel driver in use: bnx2
        Kernel modules: bnx2
```

还有个办法 去/sys 里面再次确认一下

```
root@lotus:~# ls /sys/class/net/eth0/queues/
rx-0  tx-0
```

发现的确只有一个 RX 一个 TX。这说明这是一个单队列的网卡。虽然是穷人的硬件，但是也要努力把它做好。
优化过程如下：

```
root@lotus:/sys/class/net/eth0/queues/rx-0# echo 0f > rps_cpus
root@lotus:/sys/class/net/eth0/queues/rx-0# cat rps_cpus
00000000,00000000,00000000,00000000,00000000,00000000,00000000,0000000f
root@lotus:/sys/class/net/eth0/queues/rx-0# echo 32768 > rps_flow_cnt
root@lotus:/sys/class/net/eth0/queues/rx-0# cat rps_flow_cnt
32768
root@lotus:/sys/class/net/eth0/queues/rx-0#
```

目前系统的优化，就告一个段落。我的 sysctl.conf 是这样的

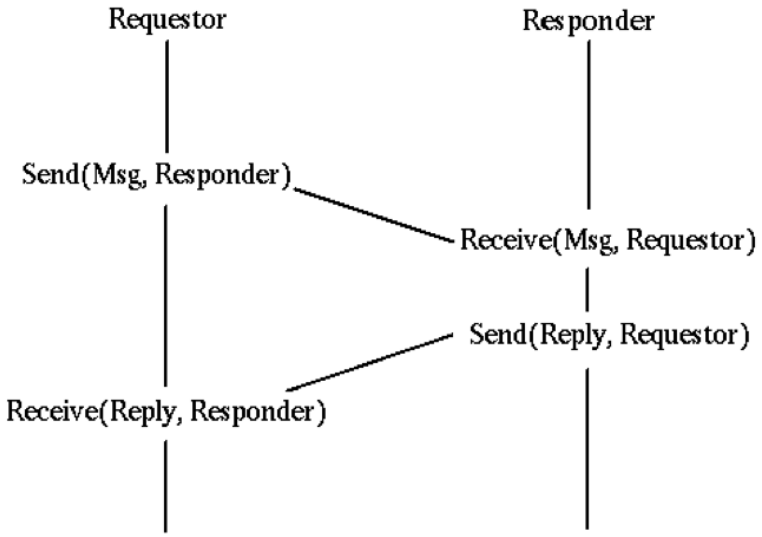
```
root@lotus:~# sysctl -p
fs.file-max = 1025500
net.core.netdev_max_backlog = 30000
net.core.somaxconn = 10000
net.core.rps_sock_flow_entries = 32768
net.ipv4.tcp_max_syn_backlog = 10000
net.ipv4.tcp_max_tw_buckets = 10000
net.ipv4.tcp_fin_timeout = 10
net.ipv4.tcp_timestamps = 0
net.ipv4.tcp_tw_recycle = 0
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_synack_retries = 10
net.ipv4.tcp_window_scaling = 1
net.ipv4.tcp_keepalive_time = 180
net.ipv4.tcp_keepalive_intvl = 30
net.ipv4.tcp_keepalive_probes = 5
net.ipv4.ip_local_port_range = 1024 65535
```

剩下开始写程序了。

编码

这是一篇实践性的文章，而讨论高性能服务器设计这样的话题，实在过于宽泛了。所以我只是从我的角度来描述看待网络程序开发。

在我看来一个最简单的网络通讯的过程如下图所示



如果用从代码的角度看，阻塞的网络 IO 模式无疑是最符合我们人类的思维模式。

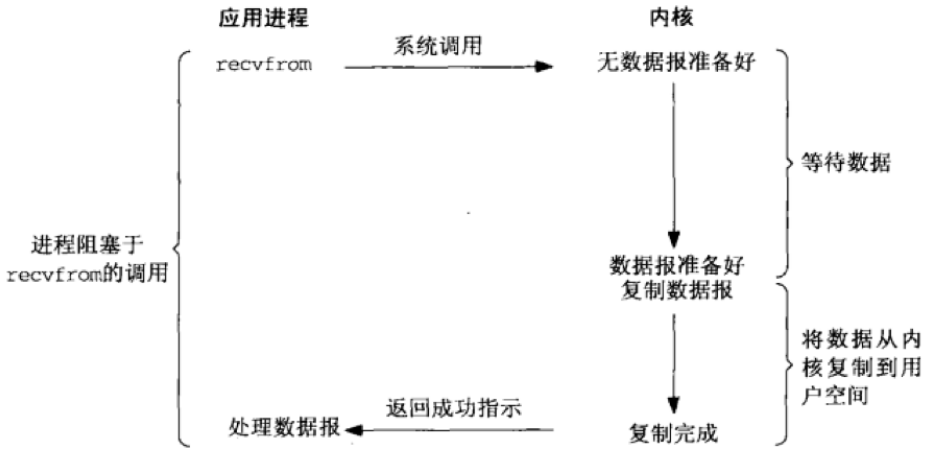


图6-1　阻塞式I/O模型

非常可惜，阻塞模式唯一的缺点，也就是最大的缺点。性能太过于低下了，宝贵的 CPU 资源被完全浪费了。所以大家又陆续的采用了多线程加阻塞 IO 的模式，非阻塞轮询的方式，多路复用的方法，异步 IO，还有诸如纤程这样的模型。各自方法我认为都有有缺点，多线程方法加阻塞 IO 根本无法大规模提高性能。而非阻塞轮询则适用面太小。 多路复用方式 如 epoll 则基本成为了一个比较主流的设计方法，但是它最大的问题是将逻辑块碎片化了，成为程序员逻辑设计的一个难题。纤程方式则试图在这几个方面获取更好的平衡。但是在 C/C++平台上，实现过程过于晦涩，还存在着调试困难这样的问题，可以说并不完美。直到我看到了 Go ，以及它的并发模型

Concurrency: goroutines, channels, and `select` [edit]

Go provides facilities for writing concurrent programs that share state by communicating.[23][24][25] Concurrency refers not only to multithreading and CPU parallelism, which Go supports, but also to asynchrony: letting slow operations like a database or network-read run while the program does other work, as is common in event-based servers.[26]

http://en.wikipedia.org/wiki/Go_(programming_language)#Concurrency:_goroutines.2C_channels.2C_and_select

我想这正是需要的东西。 但是在 IT 界，我们每每都能听得宣称上天入地，无所不能，而实际却是一团糟的产品。所以 Go 会是这样吗？我不知道对于 Go 我也完全是个新手，学习时间不超过 10 个小时，那么姑且一试吧 。

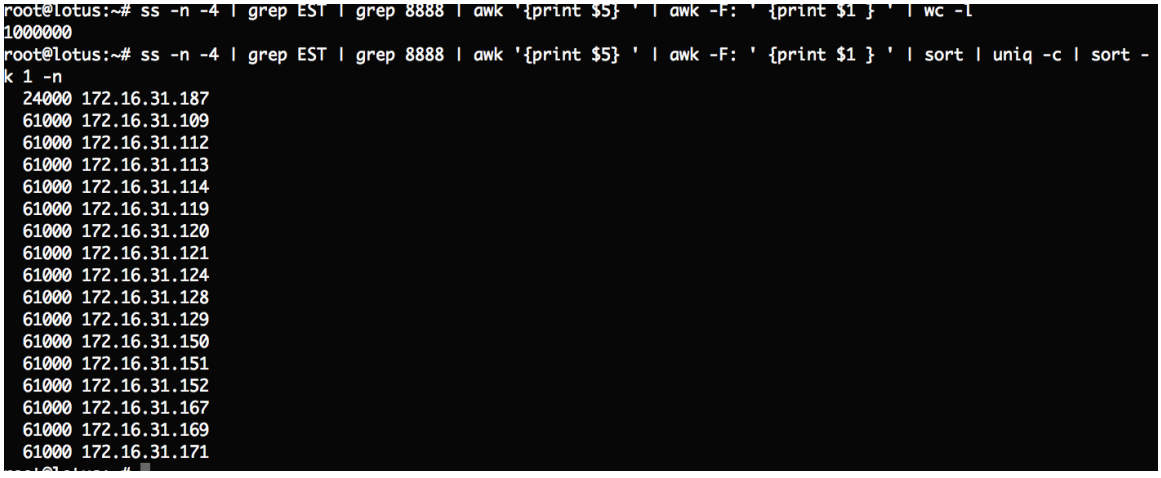
我一共开发了 2 个版本的 C1000K 服务器，一个使用了我比较熟悉的 C++，另一个是 Go。 C++ 是从零开始，所以代码应该在 1000 行以内，简单的设计了一个网络框架，使用了 epoll,pipe,cpu 亲缘性等基本功能。而 Go 只有 100 行不到，非常惊人。

测试

让我们开始测试吧，测试的客户机分布在 16 台不同的虚拟机上，每个客户端都启动了 60000 个以上的连接连向了服务器。

下面是

C++ 版本的系统截图

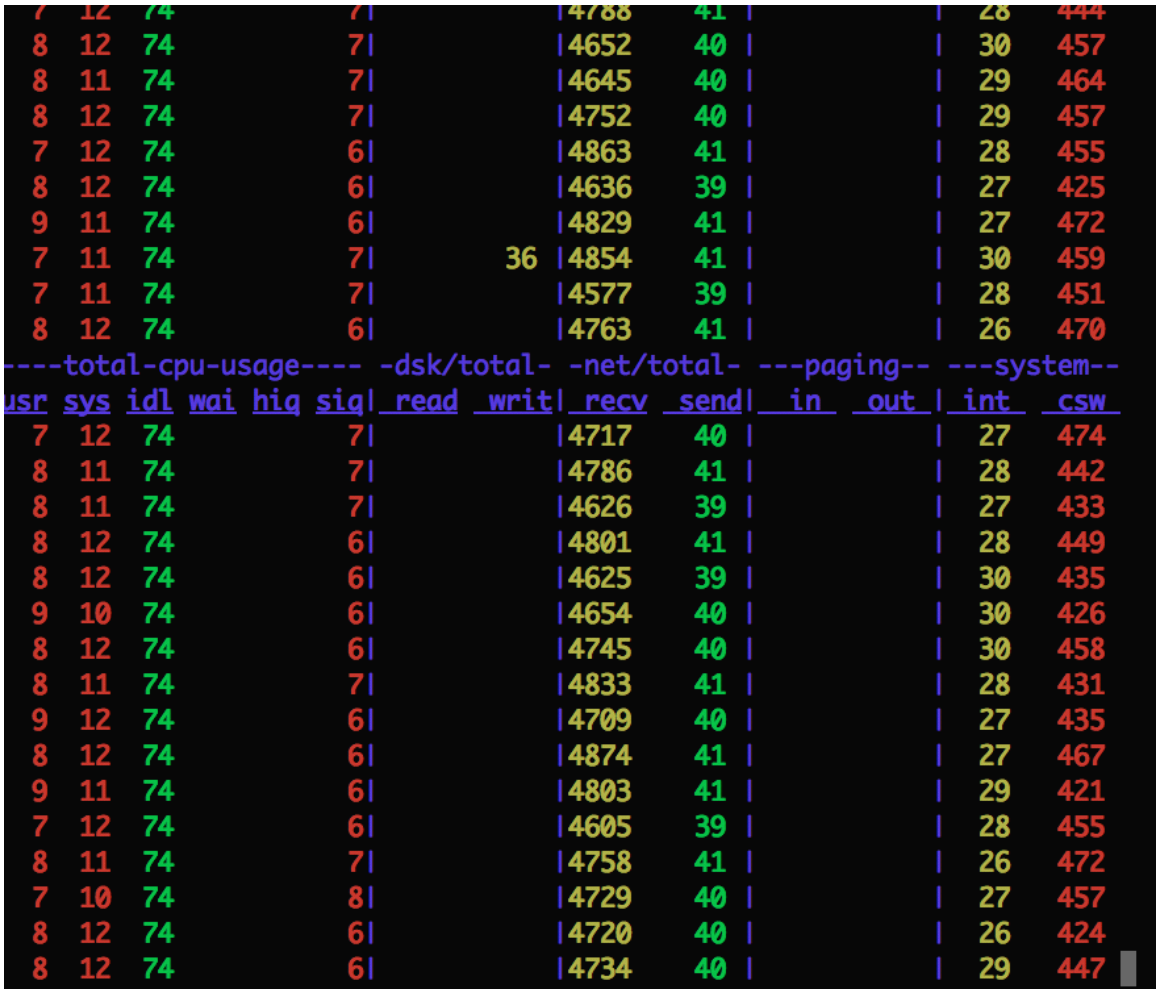


```
top - 01:26:15 up 5 days,  7:37,  5 users,  load average: 1.89, 1.59, 1.07
Tasks:   1 total,   0 running,   1 sleeping,   0 stopped,   0 zombie
Cpu0  :  0.0%us,  0.0%sy,  0.0%ni, 99.3%id,  0.3%wa,  0.0%hi,  0.3%si,  0.0%st
Cpu1  :  3.4%us, 11.0%sy,  0.0%ni, 77.0%id,  0.0%wa,  0.0%hi,  8.6%si,  0.0%st
Cpu2  :  3.4%us, 13.8%sy,  0.0%ni, 76.6%id,  0.0%wa,  0.0%hi,  6.2%si,  0.0%st
Cpu3  :  3.1%us, 11.9%sy,  0.0%ni, 75.6%id,  0.0%wa,  0.0%hi,  9.5%si,  0.0%st
Mem:  16428352k total, 14761404k used,  1666948k free,  5227328k buffers
Swap: 16386296k total,     2884k used, 16383412k free,  4245444k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
14286 root      20   0  514m 312m 1112 S   63  1.9  13:00.10 Server
```

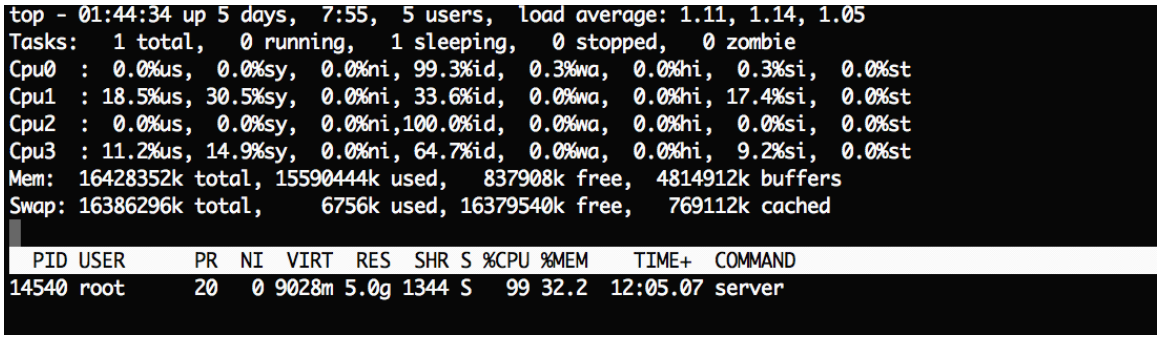
```
 7  26  47          20|          |5347    46 |          |  37  1951
 2   7  85           6|          |9773    83 |          |  17  2002
 3  14  75           9|          |3273    28 |          |  23  2005
 4  15  67          13|          |8903    76 |          |  28  1531
 2   6  88           5|          |4698    40 |          |  12  1084
 5  15  69          11|          |4119    35 |          |  26  1426
 7  27  48          19|          |9413    80 |          |  38  1733
 3  17  66          14|          |  10    88 |          |  29  1442
 2   5  89           4|          |4450    38 |          |  15  1504
 2   4  90           4|          |2217    19 |          |  14  1386
 7  29  47          17|          |5344    45 |          |  36  1860
 2   8  85           6|          |9777    83 |          |  17  1920
 3  13  75           9|          |3273    28 |          |  25  1934
 4  16  68          12|          |8902    76 |          |  28  1485
 2   7  89           2|          |4699    40 |          |  12  1056
 4  16  69          11|          |4119    35 |          |  26  1390
 7  27  47          19|          |9412    80 |          |  38  1853
 6  17  66          12|          |  10    88 |          |  28  1404
 1   5  91           3|          |4413    38 |          |  12  1446
 1   5  90           4|          |2221    19 |          |  11  1381
----total-cpu-usage---- -dsk/total- -net/total- ---paging-- ---system--
usr sys idl wai hiq sig| read  writ| recv  send|  in   out | int   csw
 8  27  48          17|          |5399    46 |          |  35  1850
 2   8  84           6|          |9727    83 |          |  17  1906
 2  12  75           9|          |3277    28 |          |  23  1999
 4  14  68          14|          |8937    76 |          |  27  1523
 1   6  89           3|          |4652    40 |          |  11  1037
```

Go 版本的系统截图



```
  7  12  74           7|          |4788    41 |          |  28   444
  8  12  74           7|          |4652    40 |          |  30   457
  8  11  74           7|          |4645    40 |          |  29   464
  8  12  74           7|          |4752    40 |          |  29   457
  7  12  74           6|          |4863    41 |          |  28   455
  8  12  74           6|          |4636    39 |          |  27   425
  9  11  74           6|          |4829    41 |          |  27   472
  7  11  74           7|         36|4854    41 |          |  30   459
  7  11  74           7|          |4577    39 |          |  28   451
  8  12  74           6|          |4763    41 |          |  26   470
----total-cpu-usage---- -dsk/total- -net/total- ---paging-- ---system--
usr sys idl wai hiq siq| read  writ| recv  send|  in   out | int   csw
  7  12  74           7|          |4717    40 |          |  27   474
  8  11  74           7|          |4786    41 |          |  28   442
  8  11  74           7|          |4626    39 |          |  27   433
  8  12  74           6|          |4801    41 |          |  28   449
  8  12  74           6|          |4625    39 |          |  30   435
  9  10  74           6|          |4654    40 |          |  30   426
  8  12  74           6|          |4745    40 |          |  30   458
  8  11  74           7|          |4833    41 |          |  28   431
  9  12  74           6|          |4709    40 |          |  27   435
  8  12  74           6|          |4874    41 |          |  27   467
  9  11  74           6|          |4803    41 |          |  29   421
  7  12  74           6|          |4605    39 |          |  28   455
  8  11  74           7|          |4758    41 |          |  26   472
  7  10  74           8|          |4729    40 |          |  27   457
  8  12  74           6|          |4720    40 |          |  26   424
  8  12  74           6|          |4734    40 |          |  29   447
```

```
root@lotus:~# ss -n -6 | grep EST | grep 8888 | awk '{print $5} ' | awk -F: ' {print $4 } ' | wc -l
1000498
root@lotus:~# ss -n -6 | grep EST | grep 8888 | awk '{print $5} ' | awk -F: ' {print $4 } ' | sort | uniq -c | sort -
k 1 -n
  28150 172.16.31.187
  57375 172.16.31.167
  60984 172.16.31.119
  60989 172.16.31.120
  61000 172.16.31.109
  61000 172.16.31.112
  61000 172.16.31.113
  61000 172.16.31.114
  61000 172.16.31.121
  61000 172.16.31.124
  61000 172.16.31.128
  61000 172.16.31.129
  61000 172.16.31.150
  61000 172.16.31.151
  61000 172.16.31.152
  61000 172.16.31.169
  61000 172.16.31.171
```



小结

通过比较可以看出，在内存使用方面，应用程序层面 C++版本 以 500M 对 9G 完胜。CPU 占有方面 C++版本也要略低一些。网络流量吞吐方面，和客户端连接方面,C++的版本更稳定一些，Go 版本偶尔会出现客户端连接失败的情况。（截图里可以看出来，当然这也有一定偶然性），另一个原因就是我使用 Go 的技术经验不足，可以继续改进，也欢迎大家对 Go 版本提出意见。

我认为 Golang 的表现非常惊艳，虽然目前还有不完善的地方，但是它还很年轻，但还在快速的进化，我认为它必将统一后台服务器开发领域。虽然 C++也在高性能领域证明了自己能力，当但是从开发效率和对人员要求这一角度来看，Go 简直完胜了 C++。因为我这样一个初出茅庐的程序员，也可能开发出 100 万级别的应用服务器，对于一个同等水平的 C++工程师来说，这完全不可想象的事情。 Go 是个好东西。

最后，这篇文章的生成是在借鉴了大量的网上资料的前提下完成的。这些资料也是我近年来参考的一些积累，但是不幸的是，当时不少原始链接有失效，所以不是所有资料都是原创地址，请见谅。 如果你发现那些资料有遗漏，也欢迎来邮件告诉我。 谢谢！

Reference

UNP
Steven 老师的书
余老师的博客 （干货很多）
http://blog.yufeng.info/
火丁笔记（非常实用）
http://huoding.com/
再叙 TIME_WAIT
http://huoding.com/2013/12/31/316
浅谈 TCP 优化

http://huoding.com/2013/11/21/299

记录一个软中断问题

http://huoding.com/2013/10/30/296

香草的技术博客

http://blog.chunshengster.me/

提高 Linux 上 socket 性能（IBM 的文章还是需要一读的）

http://www.ibm.com/developerworks/cn/linux/l-hisock.html

陶辉笔记（有很多很好的内容）

http://taohui.org.cn/tcpperf1.html

酷 壳 TCP 的那些事（能全读懂,TCP 也算是初窥门径了，博客内容很好）

http://coolshell.cn/articles/11564.html

华仔-技术博客

http://blog.csdn.net/yunhua_lee/article/details/8146830

Linux 内核优化若干参数说明

http://www.tuicool.com/articles/63aiqe

javawebsoa 的博客

http://www.cnblogs.com/javawebsoa/archive/2013/05/18/3086034.html

lenky0401 个人博客 （很多高性能服务器方面分享）

http://blog.chinaunix.net/uid/7907749.html

Ideawu 的博客（另一个 C1000k 实现的博客）

http://www.ideawu.net/blog/tag/c1000k

TCP 协议疑难杂症全景解析

http://blog.csdn.net/dog250/article/details/6612496

多队列网卡相关

http://ju.outofmemory.cn/entry/138

http://blog.csdn.net/turkeyzhou/article/details/7528182

http://blog.chinaunix.net/xmlrpc.php?r=blog/article&uid=26642180&id=3131179

http://blog.csdn.net/h_flight/article/details/9121999

http://www.tuicool.com/articles/e6JFf2

http://blog.chinaunix.net/uid-10915175-id-3367864.html

pagefault 的博客

http://www.pagefault.info

相关资料

http://en.wikipedia.org/wiki/Go_(programming_language)

https://github.com/torvalds/linux/blob/master/Documentation/networking/scaling.txt

http://itlab.idcquan.com/cisco/TCP/525263.html

http://my.oschina.net/nyankosama/blog/271319

http://blog.chinaunix.net/uid-7377299-id-112984.html
http://blog.chinaunix.net/uid-21505614-id-2181210.html
http://blog.csdn.net/wireless_tech/article/details/6405755
http://blog.csdn.net/zhangskd/article/details/7608343
…..

Email: ppmsn2005#gmail.com