不为人知的网络编程(五)：UDP的连接性和负载均衡-网络编程/专项技术区 - 即时通讯开发者社区!

原作者：黄日成，手Q游戏中心后台开发，腾讯高级工程师。从事C++服务后台开发4年多，主要负责手Q游戏中心后台基础系统、复杂业务系统开发，主导过手Q游戏公会、企鹅电竞App-对战系统等项目的后台系统设计，有丰富的后台架构经验。

1、前言

很早就计划写篇关于UDP的文章，尽管UDP协议远没TCP协议那么庞大、复杂，但是要想将UDP描述清楚，用好UDP却要比TCP难不少，于是文章从下笔写到最终写成，断断续续拖了好几个月。

说起网络 socket，大家自然会想到 TCP ，用的最多也是

TCP，UDP 在大家的印象中是作为 TCP 的补充而存在：是无连接、不可靠、无序、无流量控制的传输层协议。UDP的无连接性已经深入人心，协议上的无连接性指的是一个 UDP 的 Endpoint1(IP,PORT)，可以向多个 UDP 的 Endpointi (IP ， PORT)发送数据包，也可以接收来自多个 UDP 的 Endpointi(IP,PORT) 的数据包。实现上，考虑这样一个特殊情况：UDP Client 在 Endpoint_C1只往 UDP Server 的 Endpoint_S1 发送数据包，并且只接收来自 Endpoint_S1 的数据包，把 UDP 通信双方都固定下来，这样不就形成一条单向的虚”连接”了么?

本文将从实践出发，讨论UDP在实际应用中的连接性和负载均衡问题。

注意：本文中涉及到的具体代码、函数都是以Linux C++为例来讲解，如果您对Linux C++不太了解的话也没有关系，把它们当伪码看即可，必竟具体的语言实现并不妨碍问题解决思路的表达。

2、系列文章

本文是系列文章中的第5篇，本系列文章的大纲如下：

- 《不为人知的网络编程(一)：浅析TCP协议中的疑难杂症(上篇)》
- 《不为人知的网络编程(二)：浅析TCP协议中的疑难杂症(下篇)》
- 《不为人知的网络编程(三)：关闭TCP连接时为什么

会TIME_WAIT、CLOSE_WAIT》

- 《不为人知的网络编程(四)：深入研究分析TCP的异常关闭》
- 《不为人知的网络编程(五)：UDP的连接性和负载均衡》（本文）
- 《不为人知的网络编程(六)：深入地理解UDP协议并用好它》
- 《不为人知的网络编程(七)：如何让不可靠的UDP变的可靠？》
- 《不为人知的网络编程(八)：从数据传输层深度解密HTTP》
- 《不为人知的网络编程(九)：理论联系实际，全方位深入理解DNS》

如果您觉得本系列文章过于专业，您可先阅读《网络编程懒人入门》系列文章，该系列目录如下：

- 《网络编程懒人入门(一)：快速理解网络通信协议（上篇）》
- 《网络编程懒人入门(二)：快速理解网络通信协议（下篇）》
- 《网络编程懒人入门(三)：快速理解TCP协议一篇就够》
- 《网络编程懒人入门(四)：快速理解TCP和UDP的差异》
- 《网络编程懒人入门(五)：快速理解为什么说UDP有时比TCP更有优势》

本站的《脑残式网络编程入门》也适合入门学习，本系列大纲如下：

- 《脑残式网络编程入门(一)：跟着动画来学TCP三次握手和四次挥手》
- 《脑残式网络编程入门(二)：我们在读写Socket时，究竟在读写什么？》
- 《脑残式网络编程入门(三)：HTTP协议必知必会的一些知识》
- 《脑残式网络编程入门(四)：快速理解HTTP/2的服务器推送(Server Push)》

关于移动端网络特性及优化手段的总结性文章请见：

- 《现代移动端网络短连接的优化手段总结：请求速度、弱网适应、安全保障》
- 《移动端IM开发者必读(一)：通俗易懂，理解移动网络的“弱”和“慢”》
- 《移动端IM开发者必读(二)：史上最全移动弱网络优化方法总结》

3、参考资料

《TCP/IP详解 - 第11章·UDP：用户数据报协议》

《为什么QQ用的是UDP协议而不是TCP协议？》
《移动端IM/推送系统的协议选型：UDP还是TCP？》
《简述传输层协议TCP和UDP的区别》
《UDP中一个包的大小最大能多大》
《为什么说基于TCP的移动端IM仍然需要心跳保活？》

4、UDP的”连接性”

估计很多同学认为UDP的连接性只是将UDP通信双方都固定下来了，一对一只是多对多的一个特例而已，这样UDP连接不连接到无所谓了。果真如此吗？其实不然，UDP的连接性可以带来以下2个好处。

4.1高效率、低消耗

我们知道Linux系统有用户空间(用户态)和内核空间(内核态)之分，对于x86处理器以及大多数其它处理器，用户空间和内核空间之前的切换是比较耗时(涉及到上下文的保存和恢复，一般3种情况下会发生用户态到内核态的切换：发生系统调用时、产生异常时、中断时)。那么对于一个高性能的服务应该减少频繁不必要的上下文切换，如果切换无法避免，那么尽量减少用户空间和内核空间的数据交换，减少数据拷贝。熟悉socket编程的同学对下面几个系统调用应该比较熟悉了，由于UDP是基于用户数据报的，只要数据包准备好就应该调用一次send或sendto进行发包，当然包的大小完全由应用层逻辑决定的。

细看两个系统调用的参数便知道，sendto比send的参数多2个，这就意味着每次系统调用都要多拷贝一些数据到内核空间，同时，参数到内核空间后，内核还需要初始化一些临时的数据结构来存储这些参数值(主要是对端Endpoint_S的地址信息)，在数据包发出去后，内核还需要在合适的时候释放这些临时的数据结构。进行UDP通信的时候，如果首先调用connect绑定对端Endpoint_S的后，那么就可以直接调用send来给对端Endpoint_S发送UDP数据包了。用户在connect之后，内核会永久维护一个存储对端Endpoint_S的地址信息的数据结构，内核不再需要分配/删除这些数据结构，只需要查找就可以了，从而减少了数据的拷贝。这样对于connect方而言，该UDP通信在内核已经维护这一个“连接”了，那么在通信的整个过程中，内核都能随时追踪到这个“连接”。

```
int connect(int socket, const struct sockaddr *address, socklen_t address_len);

ssize_t send(int socket, const void *buffer, size_t length,  int flags);

ssize_t sendto(int socket, const void *message, size_t length,  int flags, const struct sockaddr *dest_addr,  socklen_t dest_len);

ssize_t recv(int socket, void *buffer, size_t length, int flags);

ssize_t recvfrom(int socket, void *restrict buffer, size_t length,  int flags, struct sockaddr *restrict address, socklen_t *restrict address_len);
```

4.2错误提示

相信大家写 UDP Socket 程序的时候，有时候在第一次调用 sendto 给一个 unconnected UDP socket 发送 UDP 数据包时，接下来调用 recvfrom() 或继续调sendto的时候会返回一个 ECONNREFUSED 错误。对于一个无连接的 UDP 是不会返回这个错误的，之所以会返回这个错误，是因为你明确调用了 connect 去连接远端的 Endpoint_S 了。那么这个错误是怎么产生的呢？没有调用 connect 的 UDP Socket 为什么无法返回这个错误呢?

当一个 UDP socket 去 connect 一个远端 Endpoint_S 时，并没有发送任何的数据包，其效果仅仅是在本地建立了一个五元组映射，对应到一个对端，该映射的作用正是为了和 UDP 带外的 ICMP 控制通道捆绑在一起，使得 UDP socket 的接口含义更加丰满。这样内核协议栈就维护了一个从源到目的地的单向连接，当下层有ICMP(对于非IP协议，可以是其它机制)错误信息返回时，内核协议栈就能够准确知道该错误是由哪个用户socket产生的，这样就能准确将错误转发给上层应用了。对于下层是IP协议的时候，ICMP 错误信息返回时，ICMP 的包内容就是出错的那个原始数据包，根据这个原始数据包可以找出一个五元组，根据该五元组就可以对应到一个本地的connect过的 UDP socket，进而把错误消息传输给该 socket，应用程序在调用socket接口函数的时候，就可以得到该错误消息了。

对于一个无“连接”的UDP，sendto系统调用后，内核在将数据包发送出去后，就释放了存储对端Endpoint_S的地址等信息的数据结构了，这样在下层的协议有错误返回的时

候，内核已经无法追踪到源socket了。

这里有个注意点要说明一下，由于UDP和下层协议都是不可靠的协议，所以，不能总是指望能够收到远端回复的ICMP包，例如：中间的一个节点或本机禁掉了ICMP，socket api调用就无法捕获这些错误了。

5、UDP的负载均衡

在多核(多CPU)的服务器中，为了充分利用机器CPU资源，TCP服务器大多采用accept/fork模式，TCP服务的MPM机制(multi processing module)，不管是预先建立进程池，还是每到一个连接创建新线程/进程，总体都是源于accept/fork的变体。然而对于UDP却无法很好的采用PMP机制，由于UDP的无连接性、无序性，它没有通信对端的信息，不知道一个数据包的前置和后续，它没有很好的办法知道，还有没后续的数据包以及如果有的话，过多久才会来，会来多久，因此UDP无法为其预先分配资源。

5.1端口重用：SO_REUSEADDR、SO_REUSEPORT

要进行多处理，就免不了要在相同的地址端口上处理数据，SO_REUSEADDR允许端口的重用，只要确保四元组的唯一性即可。对于TCP，在bind的时候所有可能产生四元组不唯一的bind都会被禁止(于是，ip相同的情况下，TCP套接字处于TIME_WAIT状态下的socket，才可以重复

绑定使用)；对于connect，由于通信两端中的本端已经明确了，那么只允许connect从来没connect过的对端(在明确不会破坏四元组唯一性的connect才允许发送SYN包)；对于监听listen端，四元组的唯一性油connect端保证就OK了。

TCP通过连接来保证四元组的唯一性，一个connect请求过来，accept进程accept完这个请求后(当然不一定要单独accept进程)，就可以分配socket资源来标识这个连接，接着就可以分发给相应的worker进程去处理该连接后续的事情了。这样就可以在多核服务器中，同时有多个worker进程来同时处理多个并发请求，从而达到负载均衡，CPU资源能够被充分利用。

UDP的无连接状态(没有已有对端的信息)，使得UDP没有一个有效的办法来判断四元组是否冲突，于是对于新来的请求，UDP无法进行资源的预分配，于是多处理模式难以进行，最终只能“守株待兔“，UDP按照固定的算法查找目标UDP socket，这样每次查到的都是UDP socket列表固定位置的socket。UDP只是简单基于目的IP和目的端口来进行查找，这样在一个服务器上多个进程内创建多个绑定相同IP地址(SO_REUSEADDR)，相同端口的UDP socket，那么你会发现，只有最后一个创建的socket会接收到数据，其它的都是默默地等待，孤独地等待永远也收不到UDP数据。UDP这种只能单进程、单处理的方式将要破灭UDP高效的神话，你在一个多核的服务器上运行这样的UDP程序，会发现只有一个核在忙，其他CPU核心处于空闲的状态。创建多个绑定相同IP地址，相同端口的UDP程序，只会起到容灾备份的作用，不会起到负载均衡的作

用。

要实现多处理，那么就要改变UDP Socket查找的考虑因素，对于调用了connect的UDP Client而言，由于其具有了“连接”性，通信双方都固定下来了，那么内核就可以根据4元组完全匹配的原则来匹配。于是对于不同的通信对端，可以查找到不同的UDP Socket从而实现多处理。而对于server端，在使用SO_REUSEPORT选项(linux 3.9以上内核)，这样在进行UDP socket查找的时候，源IP地址和源端口也参与进来了，内核查找算法可以保证：

- [1] 固定的四元组的UDP数据包总是查找到同一个UDP Socket；
- [2] 不同的四元组的UDP数据包可能会查找到不同的UDP Socket。

这样对于不同client发来的数据包就能查找到不同的UDP socket从而实现多处理。这样看来，似乎采用SO_REUSEADDR、SO_REUSEPORT这两个socket选项并利用内核的socket查找算法，我们在多核CPU服务器上多个进程内创建多个绑定相同端口，相同IP地址的UDP socket就能做到负载均衡充分利用多核CPU资源了。然而事情远没这么顺利、简单。

5.2UDP Socket列表变化问题

通过上面我们知道，在采用SO_REUSEADDR、SO_REUSEPORT这两个socket选项后，内核会根据UDP数据包的4元组来查找本机上的所有相同目的IP地址，相同目的端口的socket中的一个socket的位置，然后以这个位置上的socket作为接收数据的socket。那么要确保来至同一个Client Endpoint的UDP数据包总是被同一个socket来处理，就需要保证整个socket链表的socket所处的位置不能改变，然而，如果socket链表中间的某个socket挂了的话，就会造成socket链表重新排序，这样会引发问题。于是基本的解决方案是在整个服务过程中不能关闭UDP socket(当然也可以全部UDP socket都close掉，从新创建一批新的)。要保证这一点，我们需要所有的UDP socket的创建和关闭都由一个master进行来管理，worker进程只是负责处理对于的网络IO任务，为此我们需要socket在创建的时候要带有CLOEXEC标志(SOCK_CLOEXEC)。

5.3UDP和Epoll结合：UDP的Accept模型

到此，为了充分利用多核CPU资源，进行UDP的多处理，我们会预先创建多个进程，每个进程都创建一个或多个绑定相同端口，相同IP地址(SO_REUSEADDR、SO_REUSEPORT)的UDP socket，这样利用内核的UDP socket查找算法来达到UDP的多进程负载均衡。然而，这完全依赖于Linux内核处理UDP socket查找时的一个算法，我们不能保证其它的系统或者未来的Linux内核不会改变算法的行为；同时，算法的查找能否做到比较好的均匀分布到不同的UDP socket，(每个处理进程只处理自己

初始化时候创建的那些UDP socket)负载是否均衡是个问题。于是，我们多么想给UPD建立一个accept模型，按需分配UDP socket来处理。

在高性能Server编程中，对于TCP Server而已有比较成熟的解决方案，TCP天然的连接性可以充分利用epoll等高性能event机制，采用多路复用、异步处理的方式，哪个worker进程空闲就去accept连接请求来处理，这样就可以达到比较高的并发，可以极限利用CPU资源。然而对于UDP server而言，由于整个Svr就一个UDP socket，接收并响应所有的client请求，于是也就不存在什么多路复用的问题了。UDP svr无法充分利用epoll的高性能event机制的主要原因是，UDP svr只有一个UDP socket来接收和响应所有client的请求。然而如果能够为每个client都创建一个socket并虚拟一个“连接”与之对应，这样不就可以充分利用内核UDP层的socket查找结果和epoll的通知机制了么。server端具体过程如下。

[1] UDP svr创建UDP socket fd,设置socket为REUSEADDR和REUSEPORT、同时bind本地地址local_addr：

```
1 listen_fd = socket(PF_INET, SOCK_DGRAM, 0)
2
3 setsockopt(listen_fd, SOL_SOCKET, SO_REUSEADDR, &opt,sizeof(opt))
4
5 setsockopt(listen_fd, SOL_SOCKET, SO_REUSEPORT, &opt, sizeof(opt))
6
7 bind(listen_fd, (struct sockaddr * ) &local_addr, sizeof(struct sockaddr))
```

[2] 创建epoll fd，并将listen_fd放到epoll中 并监听其可读

事件：

```
epoll_fd = epoll_create(1000);

ep_event.events = EPOLLIN|EPOLLET;

ep_event.data.fd = listen_fd;

epoll_ctl(epoll_fd , EPOLL_CTL_ADD, listen_fd, &ep_event)

in_fds = epoll_wait(epoll_fd, in_events, 1000, -1);
```

[3] epoll_wait返回时，如果epoll_wait返回的事件fd是listen_fd，调用recvfrom接收client第一个UDP包并根据recvfrom返回的client地址, 创建一个新的socket(new_fd)与之对应，设置new_fd为REUSEADDR和REUSEPORT、同时bind本地地址local_addr，然后connect上recvfrom返回的client地址：

```
recvfrom(listen_fd, buf, sizeof(buf), 0, (struct sockaddr )&client_addr, &client_len)

new_fd = socket(PF_INET, SOCK_DGRAM, 0)

setsockopt(new_fd , SOL_SOCKET, SO_REUSEADDR, &reuse,sizeof(reuse))

setsockopt(new_fd , SOL_SOCKET, SO_REUSEPORT, &reuse, sizeof(reuse))

bind(new_fd , (struct sockaddr ) &local_addr, sizeof(struct sockaddr));

connect(new_fd , (struct sockaddr * ) &client_addr, sizeof(struct sockaddr)
```

[4] 将新创建的new_fd加入到epoll中并监听其可读等事件：

```
client_ev.events = EPOLLIN;
```

```
client_ev.data.fd = new_fd ;

epoll_ctl(epoll_fd, EPOLL_CTL_ADD, new_fd , &client_ev)
```

[5] 当epoll_wait返回时，如果epoll_wait返回的事件fd是new_fd 那么就可以调用recvfrom来接收特定client的UDP包了：

```
recvfrom(new_fd , recvbuf, sizeof(recvbuf), 0, (struct sockaddr * )&client_addr, &client_len)
```

通过上面的步骤，这样 UDP svr 就能充分利用 epoll 的事件通知机制了。第一次收到一个新的 client 的 UDP 数据包，就创建一个新的UDP socket和这个client对应，这样接下来的数据交互和事件通知都能准确投递到这个新的UDP socket fd了。

这里的UPD和Epoll结合方案，有以下几个注意点：

- **[1] client要使用固定的ip和端口和server端通信，也就是client需要bind本地local address：**
  如果client没有bind本地local address，那么在发送UDP数据包的时候，可能是不同的Port了，这样如果server 端的new_fd connect的是client的Port_CA端口，那么当Client的Port_CB端口的UDP数据包来到server时，内核不会投递到new_fd，相反是投递到listen_fd。由于需要bind和listen fd一样的IP地址和端口，因此SO_REUSEADDR和SO_REUSEPORT是必须的；
- **[2] 要小心处理上面步骤3中connect返回前，Client**

已经有多个UDP包到达Server端的情况：
如果server没处理好这个情况，在connect返回前，有2个UDP包到达server端了，这样server会new出两个new_fd1和new_fd2分别connect到client，那么后续的client的UDP到达server的时候，内核会投递UDP包给new_fd1和new_fd2中的一个。

上面的UDP和Epoll结合的accept模型有个不好处理的小尾巴(也就是上面的注意点[2])，这个小尾巴的存在其本质是UDP和4元组没有必然的对应关系，也就是UDP的无连接性。

5.4UDP Fork 模型：UDP accept模型之按需建立UDP处理进程

为了充分利用多核 CPU (为简化讨论，不妨假设为8核)，理想情况下，同时有8个工作进程在同时工作处理请求。于是我们会初始化8个绑定相同端口，相同IP地址(SO_REUSEADDR、SO_REUSEPORT)的 UDP socket，接下来就靠内核的查找算法来达到client请求的负载均衡了。由于内核查找算法是固定的，于是，无形中所有的client被划分为8类，类型1的所有client请求全部被路由到工作进程1的UDP socket由工作进程1来处理，同样类型2的client的请求也全部被工作进程2来处理。这样的缺陷是明显的，比较容易造成短时间的负载极端不均衡。

一般情况下，如果一个 UDP 包能够标识一个请求，那么

简单的解决方案是每个 UDP socket n 的工作进程 n，自行 fork 出多个子进程来处理类型n的 client 的请求。这样每个子进程都直接 recvfrom 就 OK 了，拿到 UDP 请求包就处理，拿不到就阻塞。

然而，如果一个请求需要多个 UDP 包来标识的情况下，事情就没那么简单了，我们需要将同一个 client 的所有 UDP 包都路由到同一个工作子进程。为了简化讨论，我们将注意力集中在都是类型n的多个client请求UDP数据包到来的时候，我们怎么处理的问题，不同类型client的数据包路由问题交给内核了。这样，我们需要一个master进程来监听UDP socket的可读事件，master进程监听到可读事件，就采用MSG_PEEK选项来recvfrom数据包，如果发现是新的Endpoit(ip、port)Client的UDP包，那么就fork一个新的进行来处理该Endpoit的请求。

具体如下：

- [1] master进程监听udp_socket_fd的可读事件：
  pfd.fd = udp_socket_fd;pfd.events = POLLIN;
  poll(pfd, 1, -1);
  当可读事件到来，pfd.revents & POLLIN 为true。探测一下到来的UDP包是否是新的client的UDP包:recvfrom(pfd.fd, buf, MAXSIZE, MSG_PEEK, (struct sockaddr *)pclientaddr, &addrlen);查找一下worker_list是否为该client创建过worker进程了。
- [2] 如果没有查找到，就fork()处理进程来处理该请求，并将该client信息记录到worker_list中。查找到，

那么continue，回到步骤[1]。

- [3] 每个worker子进程，保存自己需要处理的client信息pclientaddr。worker进程同样也监听udp_socket_fd的可读事件。poll(pfd, 1, -1);当可读事件到来，pfd.revents & POLLIN 为true。探测一下到来的UDP包是否是本进程需要处理的client的UDP包:recvfrom(pfd.fd, buf, MAXSIZE, MSG_PEEK, (struct sockaddr *)pclientaddr_2, &addrlen); 比较一下pclientaddr和pclientaddr_2是否一致。

该fork模型很别扭，过多的探测行为，一个数据包来了，会”惊群”唤醒所有worker子进程，大家都去PEEK一把，最后只有一个worker进程能够取出UDP包来处理。同时到来的数据包只能排队被取出。更为严重的是，由于recvfrom的排他唤醒，可能会造成死锁。

考虑下面一个场景：

假设有 worker1、worker2、worker3、和 master 共四个进程都阻塞在 poll 调用上，client1 的一个新的 UDP 包过来，这个时候，四个进程会被同时唤醒，worker1比较神速，赶在其他进程前将 UPD 包取走了(worker1可以处理client1的 UDP 包)，于是其他三个进程的 recvfrom 扑空，它们 worker2、worker3、和 master 按序全部阻塞在 recvfrom 上睡眠(worker2、worker3 排在 master 前面先睡眠的)。这个时候，一个新 client4 的 UDP 包packet4到来，(由于recvfrom的排他唤醒)这个时候只有worker2会从recvfrom的睡眠中醒来，然而worker而却不能处理该请求

UDP包。如果没有新UDP包到来，那么packet4一直留在内核中，死锁了。之所以recv是排他的，是为了避免“承诺给一个进程”的数据被其他进程取走了。

6、本文小结

通过上面的讨论，不管采用什么手段，UDP的accept模型总是那么别扭，总有一些无法自然处理的小尾巴。UDP的多路负载均衡方案不通用，不自然，其本因在于UPD的无连接性、无序性(无法标识数据的前续后继)。我们不知道client 还在不在，于是难于决策虚拟的”连接”何时终止，以及何时结束掉fork出来的worker子进程(我们不能无限fork 吧)。于是，在没有好的决策因素的时候，超时似乎是一个比较好选择，毕竟当所有的裁决手段都失效的时候，一切都要靠时间来冲淡。

（原文链接：点此进入）

7、更多资料

《TCP/IP详解 - 第11章·UDP：用户数据报协议》
《TCP/IP详解 - 第17章·TCP：传输控制协议》
《TCP/IP详解 - 第18章·TCP连接的建立与终止》
《TCP/IP详解 - 第21章·TCP的超时与重传》
《技术往事：改变世界的TCP/IP协议（珍贵多图、手机慎点）》
《通俗易懂-深入理解TCP协议（上）：理论基础》
《通俗易懂-深入理解TCP协议（下）：RTT、滑动窗口、

拥塞处理》
《理论经典：TCP协议的3次握手与4次挥手过程详解》
《理论联系实际：Wireshark抓包分析TCP 3次握手、4次挥手过程》
《计算机网络通讯协议关系图（中文珍藏版）》
《UDP中一个包的大小最大能多大？》
《Java新一代网络编程模型AIO原理及Linux系统AIO介绍》
《NIO框架入门(一)：服务端基于Netty4的UDP双向通信Demo演示》
《NIO框架入门(二)：服务端基于MINA2的UDP双向通信Demo演示》
《NIO框架入门(三)：iOS与MINA2、Netty4的跨平台UDP双向通信实战》
《NIO框架入门(四)：Android与MINA2、Netty4的跨平台UDP双向通信实战》
《P2P技术详解(一)：NAT详解——详细原理、P2P简介》
《P2P技术详解(二)：P2P中的NAT穿越(打洞)方案详解》
《P2P技术详解(三)：P2P技术之STUN、TURN、ICE详解》
《高性能网络编程(一)：单台服务器并发TCP连接数到底可以有多少》
《高性能网络编程(二)：上一个10年，著名的C10K并发连接问题》
《高性能网络编程(三)：下一个10年，是时候考虑C10M并发问题了》
《高性能网络编程(四)：从C10K到C10M高性能网络应用的理论探索》
《不为人知的网络编程(一)：浅析TCP协议中的疑难杂症

(上篇)》

《不为人知的网络编程(二)：浅析TCP协议中的疑难杂症(下篇)》

《不为人知的网络编程(三)：关闭TCP连接时为什么会TIME_WAIT、CLOSE_WAIT》

《不为人知的网络编程(四)：深入研究分析TCP的异常关闭》

>> 更多同类文章 ……