作者：dog250-anonymous  
链接：https://zhuanlan.zhihu.com/p/363696360  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

上周放假时跟着小小学python，就写了一个所谓 ***“高并发UDP服务器”*** ，详见：

<https://zhuanlan.zhihu.com/p/362049294>

之所以这么写是因为我想让UDP服务看起来像TCP服务而已，但还是发现了新东西。

我所谓的 ***服务*** ，专门指的是那种长连接，高并发的服务。

在上面这个随笔里，我的观点是 ***让socket的数量上去，并发也就上去了，剩下的就交给CPU了*** ，因此最直接的方式就是预先创建海量的reuseport UDP socket，bind到同一个端口，有客户端接入的时候，从这个socket池子里分配一个来为该客户端提供服务。但有问题： - ***我们该预创建多少个socket？*** - ***如果预创建的socket都已经在服务了，又来了请求该怎么办？*** - ...

这里的问题在于， ***服务资源往往不该受socket数量限制，而是受CPU，内存的限制！*** 所以，本质上，我们需要一种动态创建socket的方式： - ***来一个请求，创建一个socket。***

这样就剩下最后一个问题了： - ***如何知道有客户端连接了？***

所以，需要有一个专门的socket在侦听！哈哈，是不是回到了TCP的老路子上了呢？这不就是TCP的[Listen/Accept](https://www.zhihu.com/search?q=Listen%2FAccept&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A%22363696360%22%7D)模型吗？

所以说呢，我对UDP的高并发模型的认识是正确的，至少我自己认为是正确的。我重新整理了代码，这次为了突破1024的限制，我用了poll(epoll还是有点麻烦，所以不管它)：

#!/usr/bin/python3

import select

import socket

sd = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

sd.bind(('192.168.56.101', 5001))

sd.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEPORT, 1)

READ\_ONLY = ( select.POLLIN | select.POLLPRI | select.POLLHUP | select.POLLERR)

poller = select.poll()

poller.register(sd, READ\_ONLY)

fd\_to\_socket = {sd.fileno():sd, }

while True:

events = poller.poll(1000)

for fd, flag in events:

s = fd\_to\_socket[fd]

if s == sd: # Accept

data, addr = s.recvfrom(1024)

csd = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)

csd.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEPORT, 1)

csd.bind(('192.168.56.101', 5001))

csd.connect(addr)

fd\_to\_socket[csd.fileno()] = csd

poller.register(csd, READ\_ONLY)

print('accept connection from %s:%s' % addr, 'sd[%d] created' % csd.fileno())

else: # Echo

data, addr = s.recvfrom(1024)

if data:

print('received "%s" from %s' % (data.splitlines()[0].decode('utf-8'), addr), 'sd[%d]' % s.fileno())

s.sendto(data, addr)

如果不看冰山下的东西，这就已经结束了。然而冰山下有什么呢？

还是先从问题说起，如果系统已经有了10000个 ***reuseport bind到同一个端口，connect到不同客户端的socket*** ，当这些客户端发来数据的时候，内核是如何根据四元组找到对应的socket的呢？

从最老的代码直到现在，内核都只是根据bind的源IP/源端口对组织hash表，bind到同一个地址和端口的socket会链接在同一个链表中，当报文到来的时候，内核协议栈会根据目标地址和目标端口做hash，在对应的[hash链表](https://www.zhihu.com/search?q=hash%E9%93%BE%E8%A1%A8&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A%22363696360%22%7D)中遍历查找： - ***在链表中遍历，冒泡排序，找到得分最高的。***

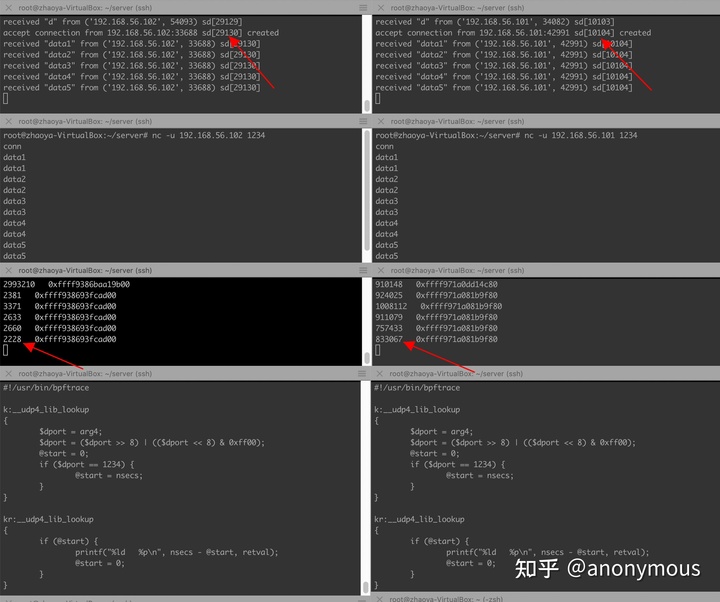
如果一个socket connect了remote，那么将会匹配四元组，匹配成功得高分，如果没有connect的socket，将只会匹配二元组，匹配成功得低分，如果连二元组都不匹配，不得分，一遍冒泡之后，就会得到一个最佳匹配结果。

理解了这个机制之后，其实不难看出一个优化点： - ***如果已经匹配了一个***[***connected socket***](https://www.zhihu.com/search?q=connected+socket&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A%22363696360%22%7D)***的四元组，就没有必要继续遍历下去了，因为不可能再有更佳的匹配了。***

但这只是冰山一角。

如果在明知hash链表很长的时候，对connected socket进行四元组hash组织的话，查这个表的效率是相当高的，Linux内核的jhash是一个十分完美的[hash算法](https://www.zhihu.com/search?q=hash%E7%AE%97%E6%B3%95&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A%22363696360%22%7D)实现。

这就是我上周最后一天半的实际工作，测试效果非常好：



左边是优化过的，已经接入了3万的连接，socket实际查询时间只有2000ns(最下面一栏是测量脚本)，而右边原生版本仅仅接入了1万连接，socket查询时间就90万ns了。

注意左边最上面的那一行，接近300万ns的时延，那是client第一次接入时查询到UDP的 ***“Listen socket”*** 所需的时间，它恰恰是原生版本socket查询时间的3倍，毕竟它的连接数是原生版本的3倍嘛，这意味着客户端第一次接入时，是不能受惠于优化机制的： - ***此时还没有反向connect呢。***

这和TCP的处理是一致的， ***TCP先匹配ESTABLISHED socket，然后再匹配Listen socket*** 。然而TCP的ESTABLISHED socket是精确四元组组织的，所以匹配效率非常高。UDP无法全部精确四元组组织，只能针对connected socket来这么干，因此，只要有一个socket无法四元组组织，就无法避开那个链表冒泡遍历。

有人会质疑，我加了一个四元组匹配逻辑，这不是额外增加处理时延吗？并不。我仅仅在二元组hash链表长度达到一定程度的时候，才会进入四元组匹配，这个长度阈值如何确定呢： - ***假设阈值为L，意味着遍历冒泡L个socket的处理时间和四元组hash计算的处理和匹配时长一致。***

所谓的可扩展性不就这回事嘛？其实就是$O(1)$，hash才是王道。那么如何保证四元组hash不会冲突成一个超长链表呢？不是学数学的，还是相信jhash为好吧。

我一直都觉得UDP的实现效率差是历史原因，历史上并没有什么动力促使人们对UDP采用更优化的算法，可如今呢？事情起了变化，QUIC，HTTP 3+，各种乱七八糟的基于UDP的自研传输协议，我就不信精通这精通那的高人们没人做优化，我这点儿雕虫小技只是一个引子罢了。 ***对UDP实现上的不足持续优化是正道。一直到把UDP搞成像TCP那样乱七八糟，但效率高啊！***

当然，如果大家一窝蜂all in DPDK/XDP的话，内核里的这点儿破事，那就留给low人吧。

纵观一部事关[Linux内核](https://www.zhihu.com/search?q=Linux%E5%86%85%E6%A0%B8&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A%22363696360%22%7D)的技术史，就是一部把任何一个逻辑搬进再搬出再搬进Linux内核的历史，Linux内核就是这么被乱入乱出的： - 2.4内核里有一个Web服务器。 - 2.6内核里没有了这个Web服务器。 - 嫌内核协议栈处理太低效，就有了Netmap/PF\_RING/DPDK的bypass技术。 - 把协议处理逻辑搬进smartnic。 - 把SSL/TLS搬进Linux内核。 - 以在Linux内核实现一个加解密隧道而自豪(就是我)。 - 在Netfilter上实现NAT64。 - 把NAT64处理逻辑通过tun网卡搬到用户态。 - 用户态实现UDP可靠传输库。 - UDP可靠传输库搬进内核。

不过说实话，网络协议栈本就不适合在内核里实现，只是一开始恰好这么做了后面也就延续了而已。

[浙江温州](https://www.zhihu.com/search?q=%E6%B5%99%E6%B1%9F%E6%B8%A9%E5%B7%9E&search_source=Entity&hybrid_search_source=Entity&hybrid_search_extra=%7B%22sourceType%22%3A%22article%22%2C%22sourceId%22%3A%22363696360%22%7D)皮鞋湿，下雨进水不会胖。UDP实现高并发其实非常简单(续集)