Linux 丢包那些事 _

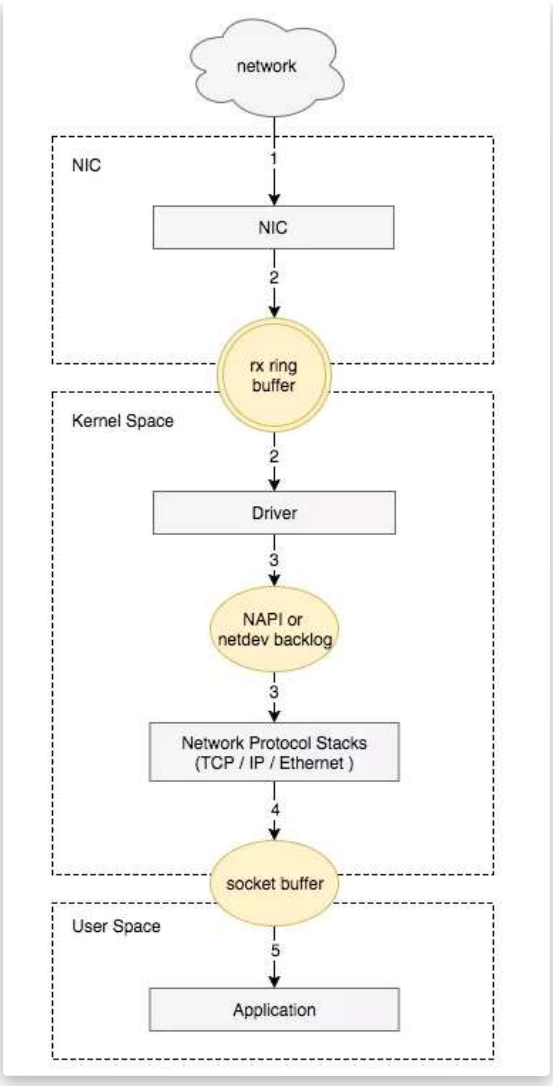
October 28, 2019 pm

3.6k 字 45 分钟

背景

最近一直在排查一些网络的问题，比如 connect timeout 、read timeout 以及一些丢包的问题，刚好想整理一些东西，方便和团队内及开发分享。

我们先看下 Linux 系统接收数据包的过程：



1. 网卡收到数据包。
2. 将数据包从网卡硬件缓存转移到服务器内存中。
3. 通知内核处理。
4. 经过 TCP/IP 协议逐层处理。
5. 应用程序通过 read() 从 socket buffer 读取数据。

网卡丢包

我们先看下ifconfig的输出：

```
# ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.5.224.27  netmask 255.255.255.0  broadcast 10.5.224.255
        inet6 fe80::5054:ff:fea4:44ae  prefixlen 64  scopeid 0x20<link>
        ether 52:54:00:a4:44:ae  txqueuelen 1000  (Ethernet)
        RX packets 9525661556  bytes 10963926751740 (9.9 TiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 8801210220  bytes 12331600148587 (11.2 TiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

RX（receive） 代表接收报文， TX(transmit) 表示发送报文。

- RX errors: 表示总的收包的错误数量，这包括 too-long-frames 错误，Ring Buffer 溢出错误，crc 校验错误，帧同步错误，fifo overruns 以及 missed pkg 等等。
- RX dropped: 表示数据包已经进入了 Ring Buffer，但是由于内存不够等系统原因，导致在拷贝到内存的过程中被丢弃。
- RX overruns: 表示了 fifo 的 overruns，这是由于 Ring Buffer(aka Driver Queue) 传输的 IO 大于 kernel 能够处理的 IO 导致的，而 Ring Buffer 则是指在发起 IRQ 请求之前的那块 buffer。很明显，

overruns 的增大意味着数据包没到 Ring Buffer 就被网卡物理层给丢弃了，而 CPU 无法及时的处理中断是造成 Ring Buffer 满的原因之一，上面那台有问题的机器就是因为 interruprs 分布的不均匀(都压在 core0)，没有做 affinity 而造成的丢包。

- RX frame: 表示 misaligned 的 frames。

对于 TX 的来说，出现上述 counter 增大的原因主要包括 aborted transmission, errors due to carrirer, fifo error, heartbeat erros 以及 windown error，而 collisions 则表示由于 CSMA/CD 造成的传输中断。

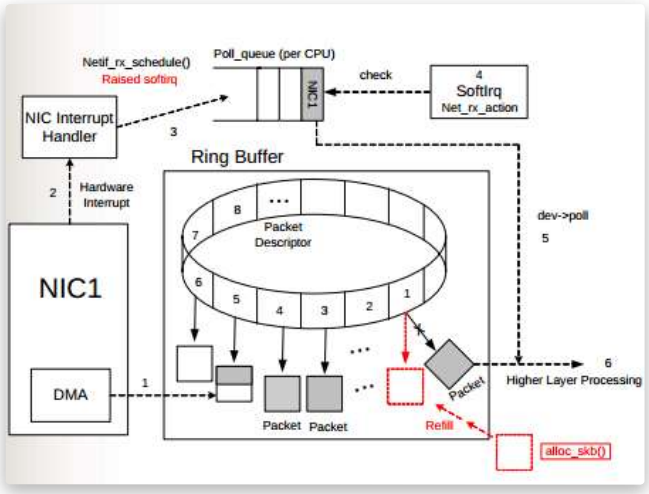
dropped 与 overruns 的区别： dropped，表示这个数据包已经进入到网卡的接收缓存 fifo 队列，并且开始被系统中断处理准备进行数据包拷贝（从网卡缓存 fifo 队列拷贝到系统内存），但由于此时的系统原因（比如内存不够等）导致这个数据包被丢掉，即这个数据包被 Linux 系统丢掉。 overruns，表示这个数据包还没有被进入到网卡的接收缓存 fifo 队列就被丢掉，因此此时网卡的 fifo 是满的。为什么 fifo 会是满的？因为系统繁忙，来不及响应网卡中断，导致网卡里的数据包没有及时的拷贝到系统内存， fifo 是满的就导致后面的数据包进不来，即这个数据包被网卡硬件丢掉。所以，个人觉得遇到 overruns 非0，需要检测cpu负载与cpu中断情况。

`netstat -i` 也会提供每个网卡的接发报文以及丢包的情况：

```
# netstat -i
Kernel Interface table
Iface      MTU    RX-OK RX-ERR RX-DRP RX-OVR    TX-OK TX-ERR TX-DRP TX-OVR Flg
eth0      1500 9528312730      0      0 0      8803615650      0      0      0 BMRU
```

Ring Buffer 溢出

如果硬件或者驱动没有问题，一般网卡丢包是因为设置的缓存区（ring buffer）太小。当网络数据包到达（生产）的速率快于内核处理（消费）的速率时， Ring Buffer 很快会被填满，新来的数据包将被丢弃。



通过 `ethtool` 或 `/proc/net/dev` 可以查看因Ring Buffer满而丢弃的包统计，在统计项中以fifo标识：

```
# ethtool -S eth0|grep rx_fifo
rx_fifo_errors: 0
# cat /proc/net/dev
Inter-|   Receive                                                |  Transmit
 face |bytes    packets errs drop fifo frame compressed multicast|bytes    packets errs drop fifo colls
  eth0: 10967216557060 9528860597    0    0    0     0          0         0 12336087749362 8804108661
```

如果发现服务器上某个网卡的 fifo 数持续增大，可以去确认 CPU 中断是否分配均匀，也可以尝试增加 Ring Buffer 的大小，通过 ethtool 可以查看网卡设备 Ring Buffer 最大值，修改 Ring Buffer 当前设置：

```
# 查看eth0网卡Ring Buffer最大值和当前设置
$ ethtool -g eth0
Ring parameters for eth0:

Pre-set maximums:
RX:     4096
RX Mini:    0
RX Jumbo:   0
TX:     4096
Current hardware settings:
RX:     1024
RX Mini:    0
RX Jumbo:   0
TX:     1024
# 修改网卡eth0接收与发送硬件缓存区大小
$ ethtool -G eth0 rx 4096 tx 4096
Pre-set maximums:
RX:     4096
RX Mini:    0
RX Jumbo:   0
TX:     4096
Current hardware settings:
RX:     4096
RX Mini:    0
RX Jumbo:   0
TX:     4096
```

netdev_max_backlog 溢出

netdev_max_backlog 是内核从 NIC 收到包后，交由协议栈（如 IP、TCP ）处理之前的缓冲队列。每个 CPU 核都有一个 backlog 队列，与 Ring Buffer 同理，当接收包的速率大于内核协议栈处理的速率时， CPU 的 backlog 队列不断增长，当达到设定的 netdev_max_backlog 值时，数据包将被丢弃。

```
# cat /proc/net/softnet_stat
2e8f1058 00000000 000000ef 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0db6297e 00000000 00000035 00000000 00000000 00000000 00000000 00000000 00000000 00000000
09d4a634 00000000 00000010 00000000 00000000 00000000 00000000 00000000 00000000 00000000
0773e4f1 00000000 00000005 00000000 00000000 00000000 00000000 00000000 00000000 00000000
```

其中： 每一行代表每个 CPU 核的状态统计，从 CPU0 依次往下； 每一列代表一个 CPU 核的各项统计：第一列代表中断处理程序收到的包总数；第二列即代表由于 netdev_max_backlog 队列溢出而被丢弃的包总数。从上面的输出可以看出，这台服务器统计中，确实有因为 netdev_max_backlog 导致的丢包。

netdev_max_backlog 的默认值是 1000，在高速链路上，可能会出现上述第二列统计不为 0 的情况，可以通过修改内核参数 net.core.netdev_max_backlog 来解决：

```
sysctl -w net.core.netdev_max_backlog=2000
```

Socket Buffer 溢出

Socket 可以屏蔽 linux 内核不同协议的差异，为应用程序提供统一的访问接口。每个 Socket 都有一个读写缓存区。

- 读缓冲区，缓存远端发来的数据。如果读缓存区已满，就不能再接收新的数据。
- 写缓冲区，缓存了要发出去的数据。如果写缓冲区已满，应用程序的写操作就会阻塞。

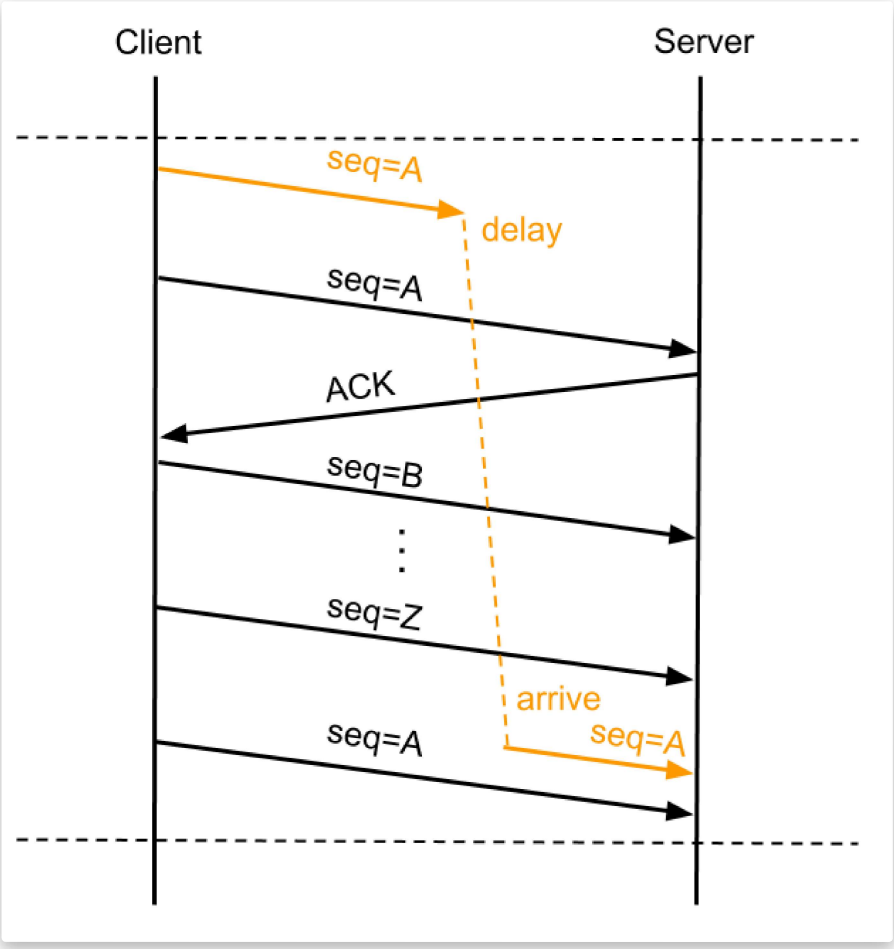
套接字内核选项列表		
套接字优化方法	**内核选项**	**参考设置**
增大每个套接字的缓冲区大小	net.core.optmem_max	81920
增大套接字接收缓冲区大小	net.core.rmem_max	513920
增大套接字发送缓冲区大小	net.core.wmem_max	513920
增大 TCP 接收缓冲区范围	net.ipv4.tcp_rmem	4096 87380 16777216
增大 TCP 发送缓冲区范围	net.ipv4.tcp_wmem	4096 65536 16777216
增大 UDP 缓冲区范围	net.ipv4.udp_mem	188562 251418 377124

半连接队列和全连接队列溢出

之前有个 connect timeout 的 case ，这篇博客里，我也详细介绍了如何去查看半连接队列和全连接队列，包括如何去优化，这里我不展开写。
但是补充一点，在半连接满的情况下，若启用syncookie机制，并不会直接丢弃 SYN 包，而是回复带有 syncookie 的 SYN+ACK 包，设计的目的是防范 SYN Flood 造成正常请求服务不可用。 syncookie 之前也有一篇博客分享，参考 谈谈 syn-cookie 的问题.

PAWS

PAWS 全名 Protect Againest Wrapped Sequence numbers ， 目的是解决在高带宽下， TCP 序列号在一次会话中可能被重复使用而带来的问题。



如上图所示，客户端发送的序列号为 A 的数据包 A1 因某些原因在网络中“迷路”，在一定时间没有到达服务端，客户端超时重传序列号为 A 的数据包 A2，接下来假设带宽足够，传输用尽序列号空间，重新使用 A，此时服务端等待的是序列号为 A 的数据包 A3，而恰巧此时前面“迷路”的 A1 到达服务端，如果服务端仅靠序列号A就判断数据包合法，就会将错误的数据传递到用户态程序，造成程序异常。

PAWS 要解决的就是上述问题，它依赖于 timestamp 机制，理论依据是：在一条正常的 TCP 流中，按序接收到的所有 TCP 数据包中的 timestamp 都应该是单调非递减的，这样就能判断那些 timestamp 小于当前 TCP 流已处理的最大 timestamp 值的报文是延迟到达的重复报文，可以予以丢弃。在上文的例子中，服务器已经处理数据包 Z，而后到来的 A1 包的 timestamp 必然小于 Z 包的 timestamp，因此服务端会丢弃迟到的 A1 包，等待正确的报文到来。

PAWS 机制的实现关键是内核保存了 Per-Connection 的最近接收时间戳，如果加以改进，就可以用来优化服务器TIME_WAIT状态的快速回收。

TIME_WAIT 状态是TCP四次挥手中主动关闭连接的一方需要进入的最后一个状态，并且通常需要在该状态保持 2*MSL （报文最大生存时间），它存在的意义有两个：

1. 可靠地实现 TCP 全双工连接的关闭：关闭连接的四次挥手过程中，最终的 ACK 由主动关闭连接的一方（称为 A ）发出，如果这个 ACK 丢失，对端（称为 B ）将重发 FIN，如果 A 不维持连接的 TIME_WAIT 状态，而是直接进入 CLOSED，则无法重传 ACK， B 端的连接因此不能及时可靠释放。
2. 等待“迷路”的重复数据包在网络中因生存时间到期消失：通信双方 A 与 B， A 的数据包因“迷路”没有及时到达 B， A 会重发数据包，当 A 与 B 完成传输并断开连接后，如果 A 不维持 TIME_WAIT 状态 2MSL 时间，便有可能与 B 再次建立相同源端口和目的端口的“新连接”，而前一次连接中“迷路”的报文有可能在这时到达，并被 B 接收处理，造成异常，维持 2MSL 的目的就是等待前一次连接的数据包在网络中消失。

TIME_WAIT 状态的连接需要占用服务器内存资源维持， Linux 内核提供了一个参数来控制 TIME_WAIT 状态的快速回收：tcp_tw_recycle，它的理论依据是：

在 PAWS 的理论基础上，如果内核保存 Per-Host 的最近接收时间戳，接收数据包时进行时间戳比对，就能避免 TIME_WAIT 意图解决的第二个问题：前一个连接的数据包在新连接中被当做有效数据包处理的情况。这样就没有必要维持 TIME_WAIT 状态 2*MSL 的时间来等待数据包消失，仅需要等待足够的 RTO （超时重传），解决 ACK 丢失需要重传的情况，来达到快速回收 TIME_WAIT 状态连接的目的。

但上述理论在多个客户端使用 NAT 访问服务器时会产生新的问题：同一个 NAT 背后的多个客户端时间戳是很难保持一致的（ timestamp 机制使用的是系统启动相对时间），对于服务器来说，两台客户端主机各自建立的 TCP 连接表现为同一个对端IP的两个连接，按照 Per-Host 记录的最近接收时间戳会更新为两台客户端主机中时间戳较大的那个，而时间戳相对较小的客户端发出的所有数据包对服务器来说都是这台主机已过期的重复数据，因此会直接丢弃。

通过netstat可以得到因PAWS机制timestamp验证被丢弃的数据包统计：

```
# netstat -s |grep -e "passive connections rejected because of time stamp" -e "packets rejects in establ
387158 passive connections rejected because of time stamp
825313 packets rejects in established connections because of timestamp
```

通过sysctl查看是否启用了 tcp_tw_recycle 及 tcp_timestamp :

```
$ sysctl net.ipv4.tcp_tw_recycle
net.ipv4.tcp_tw_recycle = 1
$ sysctl net.ipv4.tcp_timestamps
net.ipv4.tcp_timestamps = 1
```

如果服务器作为服务端提供服务，且明确客户端会通过 NAT 网络访问，或服务器之前有7层转发设备会替换客户端源IP时，是不应该开启 tcp_tw_recycle 的，而 timestamps 除了支持 tcp_tw_recycle 外还被其他机制依赖，推荐继续开启：

```
sysctl -w net.ipv4.tcp_tw_recycle=0
sysctl -w net.ipv4.tcp_timestamps=1
```

包丢在哪里了

第一个是 dropwatch， 之前 霸爷博客 也做过分享。

```
# dropwatch -l kas
Initalizing kallsyms db
dropwatch> start
Enabling monitoring...
Kernel monitoring activated.
Issue Ctrl-C to stop monitoring
1 drops at sk_stream_kill_queues+50 (0xffffffff81687860)
1 drops at tcp_v4_rcv+147 (0xffffffff8170b737)
1 drops at __brk_limit+1de1308c (0xffffffffa052308c)
1 drops at ip_rcv_finish+1b8 (0xffffffff816e3348)
1 drops at skb_queue_purge+17 (0xffffffff816809e7)
3 drops at sk_stream_kill_queues+50 (0xffffffff81687860)
2 drops at unix_stream_connect+2bc (0xffffffff8175a05c)
2 drops at sk_stream_kill_queues+50 (0xffffffff81687860)
1 drops at tcp_v4_rcv+147 (0xffffffff8170b737)
2 drops at sk_stream_kill_queues+50 (0xffffffff81687860)
```

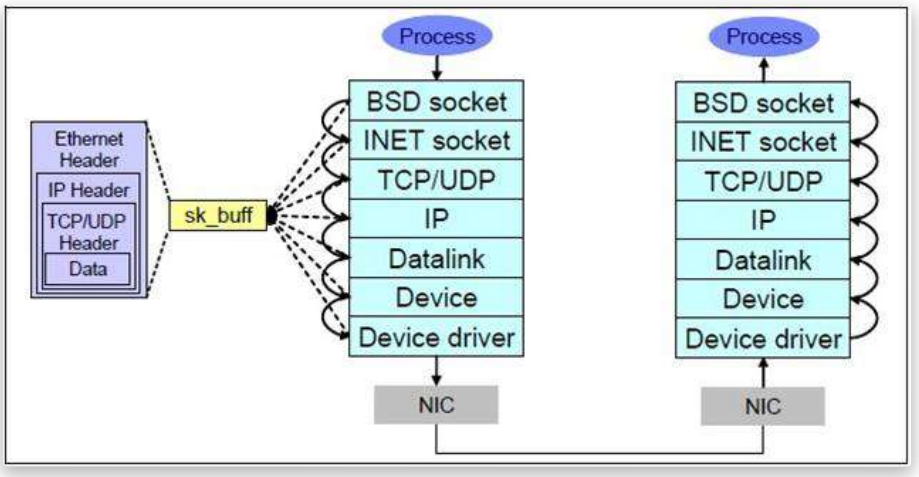
第二个是 perf 监视 kfree_skb 事件。

```
# perf record -g -a -e skb:kfree_skb
^C[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 1.212 MB perf.data (388 samples) ]

# perf script
containerd 93829 [031] 951470.340275: skb:kfree_skb: skbaddr=0xffff8827bfced700 protocol=0 location=0xff
            7fff8168279b kfree_skb ([kernel.kallsyms])
            7fff8175c05c unix_stream_connect ([kernel.kallsyms])
            7fff8167650f SYSC_connect ([kernel.kallsyms])
            7fff8167818e sys_connect ([kernel.kallsyms])
            7fff81005959 do_syscall_64 ([kernel.kallsyms])
            7fff81802081 entry_SYSCALL_64_after_hwframe ([kernel.kallsyms])
                   f908d __GI___libc_connect (/usr/lib64/libc-2.17.so)
                  13077d __nscd_get_mapping (/usr/lib64/libc-2.17.so)
                  130c7c __nscd_get_map_ref (/usr/lib64/libc-2.17.so)
                       0 [unknown] ([unknown])

containerd 93829 [031] 951470.340306: skb:kfree_skb: skbaddr=0xffff8827bfcec500 protocol=0 location=0xff
            7fff8168279b kfree_skb ([kernel.kallsyms])
            7fff8175c05c unix_stream_connect ([kernel.kallsyms])
            7fff8167650f SYSC_connect ([kernel.kallsyms])
            7fff8167818e sys_connect ([kernel.kallsyms])
            7fff81005959 do_syscall_64 ([kernel.kallsyms])
            7fff81802081 entry_SYSCALL_64_after_hwframe ([kernel.kallsyms])
                   f908d __GI___libc_connect (/usr/lib64/libc-2.17.so)
                  130ebe __nscd_open_socket (/usr/lib64/libc-2.17.so)
```

第三个是tcpdrop，之前我也有一篇 博客介绍, 它显示了源包和目标包的详细信息，以及 TCP 会话状态(来自内核)、TCP 标志(来自包 TCP 报头)和导致这次丢包的内核堆栈跟踪。

```
TIME     PID    IP SADDR:SPORT          > DADDR:DPORT          STATE (FLAGS)
05:46:07 82093  4  10.74.40.245:50010   > 10.74.40.245:58484   ESTABLISHED (ACK)
    tcp_drop+0x1
    tcp_rcv_established+0x1d5
    tcp_v4_do_rcv+0x141
    tcp_v4_rcv+0x9b8
    ip_local_deliver_finish+0x9b
    ip_local_deliver+0x6f
    ip_rcv_finish+0x124
    ip_rcv+0x291
    __netif_receive_skb_core+0x554
    __netif_receive_skb+0x18
    process_backlog+0xba
    net_rx_action+0x265
    __softirqentry_text_start+0xf2
    irq_exit+0xb6
    xen_evtchn_do_upcall+0x30
    xen_hvm_callback_vector+0x1af

05:46:07 85153  4  10.74.40.245:50010   > 10.74.40.245:58446   ESTABLISHED (ACK)
    tcp_drop+0x1
    tcp_rcv_established+0x1d5
    tcp_v4_do_rcv+0x141
    tcp_v4_rcv+0x9b8
    ip_local_deliver_finish+0x9b
    ip_local_deliver+0x6f
    ip_rcv_finish+0x124
    ip_rcv+0x291
    __netif_receive_skb_core+0x554
    __netif_receive_skb+0x18
    process_backlog+0xba
    net_rx_action+0x265
    __softirqentry_text_start+0xf2
    irq_exit+0xb6
    xen_evtchn_do_upcall+0x30
    xen_hvm_callback_vector+0x1af
```

总结



linux 网络协议栈太深，每一层都有可能出现各种各样的问题，我们需要了解这些原理，同时利用好工具去排查这些问题。同时我们在优化的时候，不要盲目的看别人的优化结果，更重要的是体系化的去了解 linux 协议栈的实现，只有知其所以然，才能结合实际业务特点，得出最合理的优化配置。

最后我按照倪鹏飞之前的优化，整理了一个表格，方便参考（数值仅供参考，具体配置还需要结合实际场景来调整）：

TCP 优化		
TCP 优化方法	**内核选项**	**参考设置**
增大处于 TIME_WAIT 状态的连接数量	net.ipv4.tcp_max_tw_buckets	1048576
增大连接跟踪表的大小	net.netfilter.nf_conntrack_max	1048576
缩短处于 TIME_WAIT 状态的超时时间	net.ipv4.tcp_fin_timeout	15
缩短连接跟踪表中处于 TIME_WAIT 状态连接的超时时间	net.netfilter.nf_conntrack_tcp_timeout_time_wait	30
允许 TIME_WAIT 状态占用的端口还可以用到新建的连接中	net.ipv4.tcp_tw_reuse	1
增大本地端口号的范围	net.ipv4.ip_local_port_range	10000 65000
增加系统和应用程序的最大文件描述符数	fs.nr_open（系统），systemd 配置文件中的 LimitNOFILE（应用程序）	1048576
增加半连接的最大数量	net.ipv4.tcp_max_syn_backlog	16384
开启 SYN Cookies	net.ipv4.tcp_syncookies	1
缩短发送 Keepalive 探测包的间隔时间	net.ipv4.tcp_keepalive_intvl	30
减少 Keepalive 探测失败后通知应用程序前的重试次数	net.ipv4.tcp_keepalive_probes	3
缩短最后一次数据包到 Keepalive 探测包的间隔时间	net.ipv4.tcp_keepalive_time	600

sre tcp linux

‹ [译] A deep dive into Kubernetes controllers

[译] Linux bcc/eBPF tcpdrop ›

NickName E-Mail

留下你的建议

Submit

1 Comments