**TCP丢包分析**

**----谢宝友 2019/08/04**



****

**本文简介：**

**本文介绍了TCP丢包问题在实际工作中的意义，并一步一步的编写内核模块代码，监控内核协议栈及网络设备丢包流程。**

# TCP丢包问题

在实际工程中，业务方常常会遇到网络套接字丢包，甚至完全不通的问题。这些问题产生的原因多种多样，常见原因有：

1. 网络设备问题。例如网络交换机丢包。
2. TCP/IP协议栈流量控制，导致报文未能从设备上发送出去。
3. 复杂的网络功能，在TCP/IP协议栈各层挂接了钩子。这些钩子可能错误的将报文转发到错误的目的地。

排查这类问题，常见的方法是使用tcpdump工具。但是tcpdump工具有一些缺点：

1. 只能在特定的TCP/IP层抓取报文
2. 性能损耗较大
3. 日志数据量大，分析麻烦
4. 对于集群中偶现的故障，不容易分析

为此，可以编写一个内核模块，在内核TCP/IP流程中注入钩子，一步一步的跟踪报文流程，直接找到丢包的原因。

# 一步一步实现丢包监控工具

## 第一步

在src/step1/main.c中，我们实现一个最简单的模块，这个模块仅仅打印模块加载/卸载的信息。主要代码如下：

16 static int drop\_packet\_init(void)

17 {

18 printk("drop-packet loaded.\n");

19

20 return 0;

21 }

22

23 static void drop\_packet\_exit(void)

24 {

25 printk("drop-packet unloaded.\n");

26 }

27

28 module\_init(drop\_packet\_init)

29 module\_exit(drop\_packet\_exit)

完整的代码位于src/step1中。

进入src/step1目录，运行make编译模块，然后执行如下命令加载模块：

insmod drop-packet.ko

通过dmesg命令可以看到加载模块产生的日志：

[79886.202735] drop-packet loaded.

运行如下命令卸载模块：

rmmod drop\_packet

同样的，通过dmesg命令可以看到卸载模块产生的日志：

[80102.988744] drop-packet unloaded.

## 第二步

第一步的模块功能过于简单。插入模块后，没有任何实质意义上的功能。

我们现在添加一个功能：动态的打开/关闭丢包检测功能。一旦我们在工程实施过程中，发现模块功能有问题，或者系统负载过高，就可以通过proc文件来关闭丢包检测功能。

首先，我们通过一个变量来表示当前是否打开丢包检测功能：

static unsigned long drop\_packet\_activated;

然后通过一个proc文件来控制这个变量。首先在初始化过程中注册proc文件，

80 static int drop\_packet\_init(void)

81 {

82 int ret;

83 struct proc\_dir\_entry \*pe;

84

85 proc\_mkdir("mooc", NULL);

86 proc\_mkdir("mooc/net", NULL);

87

88 ret = -ENOMEM;

89 pe = proc\_create("mooc/net/drop-packet",

90 S\_IFREG | 0644,

91 NULL,

92 &drop\_packet\_fops);

93 if (!pe)

94 goto err\_proc;

95

96 printk("drop-packet loaded.\n");

97

98 return 0;

99

100 err\_proc:

101 return ret;

102 }

这样，模块加载后，就存在一个名为/proc/mooc/net/drop-packet的文件。

在模块卸载的时候移除proc文件：

104 static void drop\_packet\_exit(void)

105 {

106 remove\_proc\_entry("mooc/net/drop-packet", NULL);

107 remove\_proc\_entry("mooc/net", NULL);

108 remove\_proc\_entry("mooc", NULL);

109

110 printk("drop-packet unloaded.\n");

111 }

这个proc文件的控制接口如下：

72 const struct file\_operations drop\_packet\_fops = {

73 .open = drop\_packet\_open,

74 .read = seq\_read,

75 .llseek = seq\_lseek,

76 .write = drop\_packet\_write,

77 .release = single\_release,

78 };

这是一个标准的proc文件控制接口。我们分别看看其接口实现：

写proc文件的接口实现：

43 static ssize\_t drop\_packet\_write(struct file \*file,

44 const char \_\_user \*buf, size\_t count, loff\_t \*offs)

45 {

46 int ret;

47 char cmd[255];

48 char chr[255];

49

50 if (count < 1 || \*offs)

51 return -EINVAL;

52

53 if (copy\_from\_user(chr, buf, 255))

54 return -EFAULT;

55

56 ret = sscanf(chr, "%255s", cmd);

57 if (ret <= 0)

58 return -EINVAL;

59

60 if (strcmp(cmd, "activate") == 0) {

61 if (!drop\_packet\_activated)

62 drop\_packet\_activated = activate\_drop\_packet();

63 } else if (strcmp(cmd, "deactivate") == 0) {

64 if (drop\_packet\_activated)

65 deactivate\_drop\_packet();

66 drop\_packet\_activated = 0;

67 }

68

69 return count;

70 }

这样，用户向接口文件写入activate字符，就会激活丢包检测功能。写入deactivate字符，就会关闭丢包检测功能。

激活/关闭丢包检测功能的实现函数是activate\_drop\_packet/deactivate\_drop\_packet，目前只实现了桩函数。

Proc文件读接口也比较简单：

30 static int drop\_packet\_show(struct seq\_file \*m, void \*v)

31 {

32 seq\_printf(m, "settings:\n");

33 seq\_printf(m, " activated: %s\n", drop\_packet\_activated ? "y" : "N");

34

35 return 0;

36 }

37

38 static int drop\_packet\_open(struct inode \*inode, struct file \*filp)

39 {

40 return single\_open(filp, drop\_packet\_show, NULL);

41 }

在加载模块后，用如下命令可以查看proc内容：

cat /proc/mooc/net/drop-packet

结果如下：

settings:

activated: N

执行如下命令，可以激活丢包检测功能：

echo activate > /proc/mooc/net/drop-packet

再次查看proc文件内容，结果如下：

settings:

activated: y

思考题：proc看起来是一个完美的方案，可是它在工程实践中，触发了宕机故障，这是为什么？

实际上，proc的实现有严重的缺陷：

1. 在3.10及更老的版本中，proc内核代码实现有BUG。如果存在/proc/mooc目录，并且你尝试再次创建这个目录的话，竟然会成功，并且在某种情况下会触发内核BUG，引起oops。
2. 在2.6.32版本中，不能指定proc默认分配的内存大小，如果输出的内容超过4K，那么会多次调用drop\_packet\_show，这会给代码编写带来一些困扰。

思考题：针对第一个问题，有什么好的方法找到这个缺陷并避开它？

思考题：针对第二个问题，有办法解决吗？毕竟，在生产系统中，还存在不少2.6.32版本的系统。

## 第三步

在第三步中，我们在内核TCP/IP各层挂接代码钩子。关键代码如下：

61 static int activate\_drop\_packet(void)

62 {

63 hook\_tracepoint("net\_dev\_xmit", trace\_net\_dev\_xmit\_hit, NULL);

64 hook\_kprobe(&kprobe\_dev\_queue\_xmit, "dev\_queue\_xmit",

65 kprobe\_dev\_queue\_xmit\_pre, NULL);

66 hook\_kprobe(&kprobe\_eth\_type\_trans, "eth\_type\_trans",

67 kprobe\_eth\_type\_trans\_pre, NULL);

68 hook\_kprobe(&kprobe\_napi\_gro\_receive, "napi\_gro\_receive",

69 kprobe\_napi\_gro\_receive\_pre, NULL);

70 hook\_kprobe(&kprobe\_\_\_netif\_receive\_skb\_core, "\_\_netif\_receive\_skb\_core",

71 kprobe\_\_\_netif\_receive\_skb\_core\_pre, NULL);

72 hook\_kprobe(&kprobe\_tcp\_v4\_rcv, "tcp\_v4\_rcv",

73 kprobe\_tcp\_v4\_rcv\_pre, NULL);

74

75 return 1;

76 }

这段代码在net\_dev\_xmit中挂接一个tracepoint钩子，在dev\_queue\_xmit/eth\_type\_trans/napi\_gro\_receive/\_\_netif\_receive\_skb\_core/tcp\_v4\_rcv等函数的入口处挂接一个kprobe钩子。

思考题：tracepoint和kprobe的背景知识，请大家自行百度。但是，如果我们想在ip\_local\_deliver函数的中间加钩子怎么办？ip\_local\_deliver的代码如下：

/\*

\* Deliver IP Packets to the higher protocol layers.

\*/

int ip\_local\_deliver(struct sk\_buff \*skb)

{

/\*

\* Reassemble IP fragments.

\*/

if (ip\_is\_fragment(ip\_hdr(skb))) {

if (ip\_defrag(skb, IP\_DEFRAG\_LOCAL\_DELIVER))

return 0;

}

return NF\_HOOK(NFPROTO\_IPV4, NF\_INET\_LOCAL\_IN, NULL, skb,

skb->dev, NULL,

ip\_local\_deliver\_finish);

}

hook\_tracepoint/hook\_kprobe的实现代码位于misc.c中。在这些实现中，有如下的宏，看起来代码是不是显得有点丑陋？

#if LINUX\_VERSION\_CODE < KERNEL\_VERSION(2, 6, 33)

#elif LINUX\_VERSION\_CODE < KERNEL\_VERSION(3, 15, 0)

#else

#endif

接下来，我们实现这些钩子。

## 第四步

在kprobe钩子中，要注意两点：

1. 传入参数均为pt\_regs，在这些寄存器现场中，按照x86-64的ABI规范保存了函数参数：第1到第6个参数分别保存在寄存器rdi、rsi、rdx、rcx、r8、r9中。
2. 函数一定要返回0。如果忘记返回值，或者返回其他值，均可能造成宕机。

典型的钩子实现函数如下：

116 static int kprobe\_tcp\_v4\_rcv\_pre(struct kprobe \*p, struct pt\_regs \*regs)

117 {

118 struct sk\_buff \*skb = (void \*)regs->di;

119 struct iphdr \*iphdr;

120

121 if (!drop\_packet\_activated)

122 return 0;

123

124 if (skb->protocol != cpu\_to\_be16(ETH\_P\_IP))

125 return 0;

126

127 iphdr = ip\_hdr(skb);

128 trace\_packet(skb, iphdr, TCP\_V4\_RCV);

129

130 return 0;

131 }

关键是调用函数是trace\_packet，它实现了对报文的跟踪：

我们接下来看看如何实现这个函数，以达到跟踪报文的能力。

## 第五步

要实现对每个连接的报文进行跟踪，我们需要对每个连接进行描述：

58 struct conn\_desc

59 {

60 int protocol;

61 int saddr;

62 int sport;

63 int daddr;

64 int dport;

65

66 atomic64\_t packages[TRACK\_COUNT];

67

68 struct rb\_node rb\_node;

69 struct list\_head list;

70 };

在这个数据结构中，protocol表示连接的协议类型，目前支持UDP/TCP两种连接。

saddr/sport/daddr/dport分别表示源地址/源端口/目的地址/目的端口

Packages表示在各个步骤（发送钩子/驱动接收钩子/TCP接收钩子等等）中接收到的报文数量。

Rb\_node用于将连接描述符添加到全局红黑树中。

List用于将连接描述符添加到全局链表中。

思考题：全局链表用于遍历所有连接信息，而全局红黑对也可以实现遍历的功能。看起来全局链表是多余的数据结构？

31 static struct rb\_root drop\_packet\_tree = RB\_ROOT;

32 static LIST\_HEAD(drop\_packet\_list);

33 static DEFINE\_SPINLOCK(drop\_packet\_lock);

上面三个全局变量分别表示：

全局红黑树的根，保存所有连接信息。

全局链表，保存所有连接信息。

保护前面两个数据结构的锁。

思考题：为什么这里用红黑树而不是基树，看起来基树的效率更高？

trace\_packet函数的关键在于find\_alloc\_desc。这个函数在全局数据结构中查找连接描述符，如果没有找到，就分配一个连接描述符。

如果找到这样的描述符，就增加它的计数。

当然了，我们还应当修改drop\_packet\_show，输出所有连接的统计信息。

这里需要注意的是红黑树的用法。

思考题：如果drop\_packet\_show要输出的内容超过了4K，您会发现什么现象？有兴趣实践一下吗？

思考题：红黑树的示例用法，在内核文档中。知道怎么找到它吗？

现在，我们可以激活丢包监控功能，并使用如下命令从网络中下载一点信息：

wget <http://www.baidu.com> -o /dev/null

然后看看如下命令会有什么输出：

cat /proc/mooc/net/drop-packet

## 第六步

我们还需要做一些收尾工作。

首先，工程实践中，每秒可能有数十G的数据，要监控的报文太多。如果每个报文都监控，可能拉高系统负载。因此我们需要进行过滤。

修改drop\_packet\_write函数，允许用户设置要过滤的条件：

409 } else if (strcmp(cmd, "source-addr") == 0) {

410 char addr[255];

411

412 ret = sscanf(chr, "%255s %255s", cmd, addr);

413 if (ret == 2) {

414 drop\_patcket\_saddr = ipstr2int(addr);

415 } else if (ret == 1) {

416 drop\_patcket\_saddr = 0;

417 }

418 } else if (strcmp(cmd, "source-port") == 0) {

419 unsigned int port;

420

421 ret = sscanf(chr, "%255s %255d", cmd, &port);

422 if (ret == 2) {

423 drop\_patcket\_sport = port;

424 } else if (ret == 1) {

425 drop\_patcket\_sport = 0;

426 }

427 } else if (strcmp(cmd, "dest-addr") == 0) {

428 char addr[255];

429

430 ret = sscanf(chr, "%255s %255s", cmd, addr);

431 if (ret == 2) {

432 drop\_patcket\_daddr = ipstr2int(addr);

433 } else if (ret == 1) {

434 drop\_patcket\_daddr = 0;

435 }

436 } else if (strcmp(cmd, "dest-port") == 0) {

437 unsigned int port;

438

439 ret = sscanf(chr, "%255s %255d", cmd, &port);

440 if (ret == 2) {

441 drop\_patcket\_dport = port;

442 } else if (ret == 1) {

443 drop\_patcket\_dport = 0;

444 }

445 }

同时应当修改trace\_packet函数，对报文进行过滤。

思考题：这里故意略过了对trace\_packet函数的修改，有兴趣实现它吗？

最后，如果某个粗心的工程维护人员在没有关闭报文跟踪功能的时候，就卸载了模块，此时我们的钩子还在运行。但是模块代码已经不存在了，这必然会导致宕机故障。

因此，我们需要在卸载模块的时候，做一些扫尾工作：

488 if (drop\_packet\_activated)

489 deactivate\_drop\_packet();

490 drop\_packet\_activated = 0;

压轴题：第347行的synchronize\_sched是什么作用？如果去掉，会有什么问题？