24 | 丢包：如何确定丢包的存在及其程度？

2022-03-28 杨胜辉

《网络排查案例课》 课程介绍 >

讲述：杨胜辉

时长 21:43 大小 19.89M

你好，我是胜辉。

在上一讲里，我们回顾了一个网络路径排查的典型案例。我们是通过 **nc 工具**发起不同源端口的连接，从而定位了 ECMP 路径中的问题。这个排查方法的背后，其实是我们对三层网络的深入理解和灵活应对。在这一类的网络排查中，我们都未必要上抓包分析这样的“重型武器”，只要场景合适，我们就可以用小工具达到大效果。

现在我们也知道了，这个案例的根因是 ECMP 路径中某个节点存在**丢包**。而丢包，也是网络排查中特别普遍的现象，特别是下面这三个问题：

有没有丢包？

在哪里丢包？

丢包程度多严重？

这三个问题的组合，就使得很多故障场景变得复杂。特别是当丢包情况不太明显的时候，问题表象就变得更加“神出鬼没”了。

所以在这一讲里，我们将会对“丢包”这个十分典型的问题场景进行一次深入的探讨。这样，下次你遇到丢包等问题的时候，就有很多种“兵器”，也知道在什么场景下使用它们，从而真正突破“丢包”这个难点了。

那么首先，在讨论丢包之前，我们要先对网络排查工具做一下总体的审视。

路径排查工具概览

网络路径排查的工具有挺多，大体上可以分为两大类：探测类工具和统计类工具。

为什么要这么分呢？在我看来，网络信息的获取方式，大体上有动态和静态之分：

动态的信息在传输过程中体现，传输结束后就没有了。它获得的是通信两端在动态变化中的网络状况，所以我们需要做实时的探测，才能把这个动态的状态抓取下来。

静态的信息在传输结束后依然存在，继续保存在通信端的系统中。它是通信端的网络信息的历史统计。我们只要通过运行统计类工具，就能把这种累计的信息读取出来了。

所以下面，我们就先来看看这两类工具具体都包括了什么。

探测类工具

探测类工具主要包括 ping、traceroute、mtr、nc、telnet 等，它们都是从一端发起，对另外一端发送探测报文，然后观测报文的丢失、乱序、时延等情况。

其中，ping、traceroute、mtr 主要是利用 ICMP 或者 UDP 的特性，实现了**对网络路径状况的检测**。在接下来的课程中，我们会深入探讨 traceroute 和 mtr 的工作原理。

而 nc 和 telnet，则主要是测试**传输层连通性**的，比较典型的场景就是探测 TCP 握手能否成功。有了这两个工具，我们不需要做 tcpdump 抓包，就可以检测出 TCP 端口是否可以

连通了。

统计类工具

统计类工具包括 netstat、ss，这些工具都是在一端读取自己的历史统计值，而并不发送出探测报文。比如，我们可以通过 netstat -s 命令，看到很详细的传输层、网络层、数据链路层等的质量状况，包括报文丢失、重传、重置（RST）等情况的统计。比如下面这样：

复制代码

```
# netstat -s
Ip:
    Forwarding: 1
    41406 total packets received
    0 forwarded
    0 incoming packets discarded
    41406 incoming packets delivered
    30976 requests sent out
Icmp:
    16 ICMP messages received
    0 input ICMP message failed
    ICMP input histogram:
        echo replies: 16
    16 ICMP messages sent
    0 ICMP messages failed
    ICMP output histogram:
        echo requests: 16
IcmpMsg:
        InType0: 16
        OutType8: 16
Tcp:
    18 active connection openings
    0 passive connection openings
    0 failed connection attempts
    1 connection resets received
    2 connections established
    41353 segments received
    30924 segments sent out
    0 segments retransmitted
    0 bad segments received
    0 resets sent
Udp:
    37 packets received
    0 packets to unknown port received
    0 packet receive errors
    37 packets sent
    0 receive buffer errors
    0 send buffer errors
......
```

其中，跟丢包直接相关的是 **segments retransmitted** 这个指标，**如果它一直在增长，那一般意味着网络上存在丢包**，我们需要多加注意了。

当然，netstat 命令是相对传统的命令，属于早已停止维护的 net-tools 工具集。新的工具集是 iproute2，其中包括了 ip、ss、nstat 等命令。这些命令也可以读取到类似的网络统计信息。

不过，既然新老命令的功能类似，那为什么我们还要重复造这个轮子呢？

这其实是因为，netstat 主要是通过 /proc 文件系统收集信息的，而 ss 主要通过 netlink 内核接口获取数据（有些信息也依然要从 /proc 中读取），这个 netlink 接口的效率要比 /proc 接口更高，所以 ss 能更快地返回数据。

另外，ss 能获得的信息要比 netstat 更丰富。比如，ss 就可以获取到 socket option 的信息，但 netstat 就做不到。

我们可以看一个例子。我在自己的 Ubuntu 容器里运行了这条命令：

复制代码

```
1 $ ss -eit
```

就可以获取到很多 netstat 无法获取的信息：

```
root@08d984197cfb:/# ss -eit
State                Recv-Q                Send-Q                                Local Address:Port                                 Peer Address:Port
         Process
ESTAB                0                     0                                     172.17.0.3:43740                                  45.113.192.102:http
             ino:24085 sk:1 <->
         cubic wscale:2,7 rto:201 rtt:0.278/0.139 mss:1460 pmtu:1500 rcvmss:536 advmss:1460 cwnd:10 bytes_acked:1 segs_out:2 segs_in:1 send 420.1Mbps lastsnd:2717736 lastrcv:2717736
lastack:2717736 pacing_rate 833.4Mbps delivered:1 rcv_space:14600 rcv_ssthresh:64076 minrtt:0.278
ESTAB                0                     0                                     172.17.0.3:43762                                  45.113.192.102:http
             ino:35109 sk:3 <->
         cubic wscale:2,7 rto:201 rtt:0.603/0.38 ato:40 mss:1460 pmtu:1500 rcvmss:536 advmss:1460 cwnd:10 bytes_sent:120 bytes_acked:121 bytes_received:346 segs_out:12 segs_in:11 dat
a_segs_out:8 data_segs_in:2 send 193.7Mbps lastsnd:248080 lastrcv:504793 lastack:248079 pacing_rate 387.1Mbps delivery_rate 36.6Mbps delivered:9 app_limited busy:8ms rcv_space:14600
rcv_ssthresh:64076 minrtt:0.294
```

我们可以挑几个信息展开一下。

`cubic`：这条 TCP 连接用的拥塞控制算法是 cubic。

`wscale:2,7`：这条 TCP 连接的两端的 Window Scale 分别是 2 和 7。

`cwnd:10`：这条 TCP 连接的拥塞窗口为 10 个 MSS。

如果你感到好奇：“ss 是如何获取到这么多有用的信息的呢？”其实这个也不难。一个简单的办法就是使用第 21 讲我们介绍的 strace，然后就能观察到 ss -eit 拿到这些信息的具体过程了。

> 其实，类似这样的细微的探究过程，都能不经意间不断深化我们的技术能力。

nstat 工具

我们还可以了解一下 nstat 这个工具。它跟 ss 一样，也是 iproute2 工具包里面的。nstat 的一个特点是，如果不加参数，**它每次运行时输出的数值，是从上一次被执行以来这些计数器的变化值**。这就带来了一个挺明显的优势：我们不用像使用 netstat 那样，在两次输出值中找出变化量了，nstat 输出的直接就是变化量。

首次运行 nstat 时，输出的就是全部计数器的值。第二次运行时，就只是发生变化的数值了，比如下面的 nstat 输出的就是变化的值：

复制代码

```
root@08d984197cfb:/# nstat
#kernel
IpInReceives                    5                  0.0
IpInDelivers                    5                  0.0
IpOutRequests                   4                  0.0
TcpEstabResets                  1                  0.0
TcpInSegs                       5                  0.0
TcpOutSegs                      4                  0.0
TcpOutRsts                      3                  0.0
IpExtInOctets                   336                0.0
IpExtOutOctets                  160                0.0
IpExtInNoECTPkts                5                  0.0
```

当然，要查看全部值也是可以的，执行这条命令即可：

复制代码

```
nstat -a
```

nstat 还有个功能是按 JSON 格式做输出，这有助于我们做自动化运维。做法是 nstat 命令加上--json参数：

```
1 nstat --json
```

复制代码

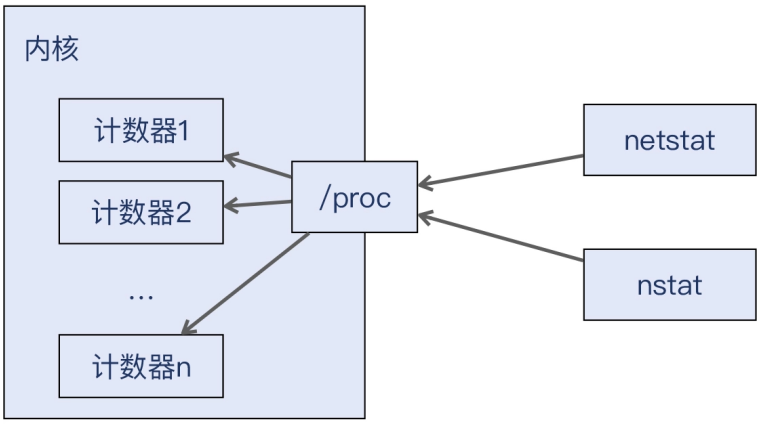
而既然 netstat 和 ss 都能读取到计数器的值，那么**这些计数器本身又是如何产生的呢？**

内核的 SNMP 计数器

这些计数器的统计和更新其实都是 Linux 内核的行为。内核根据RFC1213的标准，定义了 IP、ICMP、TCP、UDP 等协议相关的SNMP 计数器。在每次报文发送、接收、重传等行为发生时，都会往相应的计数器中更新数值。具体来说，这些计数器的定义在内核代码的 include/linux/snmp.h 文件中。

> 补充：SNMP 名义上是“简单网络管理协议”，其实名字叫“简单”的技术往往并不简单。事实上 SNMP 还是很庞大的，这里就不展开了。

所以说，这些数值都不是 netstat、nstat 这些命令去“生成”的，而是内核早就准备好这些数据了，工具去完成读取就好了。



极客时间

好了，讨论完各种工具，接下来我们就进入丢包这个核心话题。

如何确定丢包位置和丢包率？

在丢包这个问题上，我们最关心的可能就是**丢包在哪里、丢包率是多少**这两个问题。这里我们重点使用的工具，是 traceroute 和 mtr。

traceroute 和 mtr 的工作原理

我们先说 traceroute。其实在第 1 讲，我就提到过 traceroute 的两种模式：UDP 模式和 ICMP 模式。但是，**traceroute 为什么可以做到探测网络路径中的节点的呢？**要知道，ping 也是用 ICMP，它咋就看不到中间节点呢？

你可以找一个工程师问这个问题：“traceroute 是如何显示每一个节点的？”很多人可能都会卡住，或者会给一个似是而非的答案，比如说：“可能是有什么协议吧，可以展示每一个节点的”。

> 所以我们不要小看一些表面上不起眼的小技术点，也许里面藏着的，还正是你的盲区，也是我们可以提高的地方。

实际上，traceroute 的工作原理，是巧妙地利用了 IP 报文的 **TTL 属性**。具体过程是这样的：

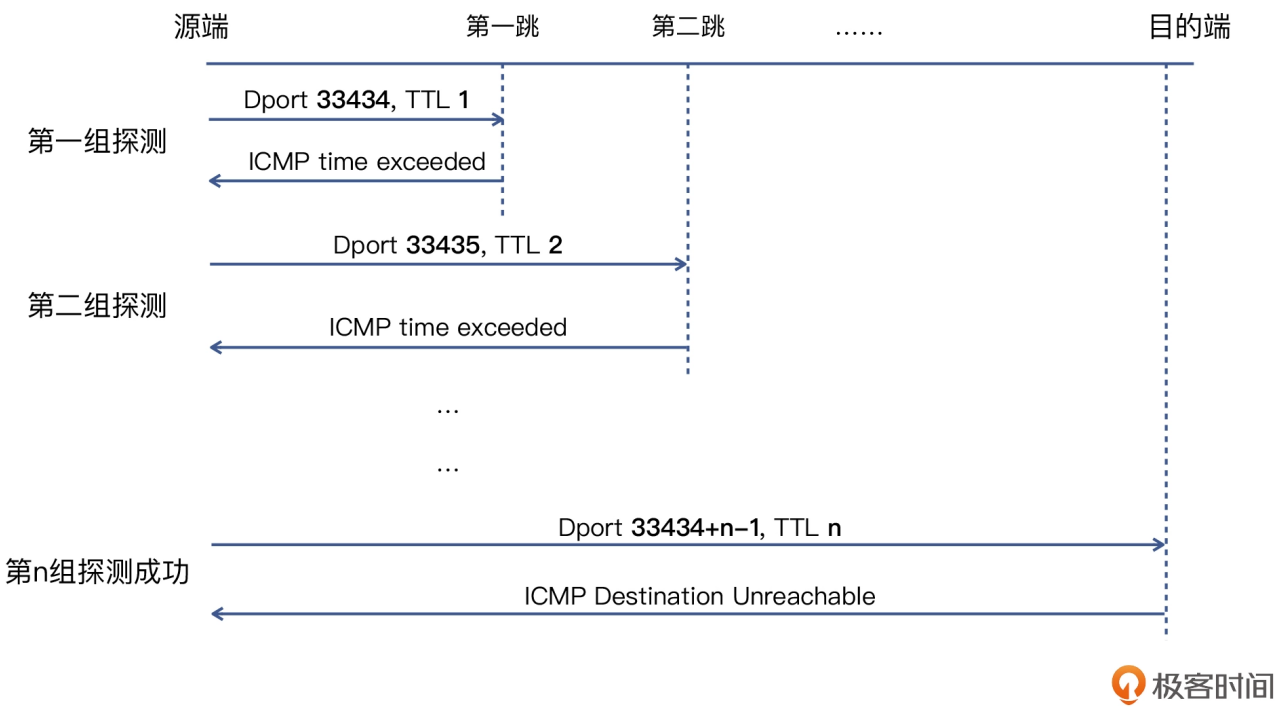
> 发送端首次探测时把 UDP 源端口为 33434，TTL 设置为 1。这个报文经过第一个路由器时 TTL 会被减 1，也就是变为 0，因此包被丢弃，这个路由器向源地址发回一个 ICMP 超时通知（ICMP Time Exceeded Message），于是第一跳就被探测出来了。
>
> 发送端把下一次发送的包的 UDP 源端口也加一，变为 33435，TTL 也在原来的基础加一，变为 2，这样就可以多前进一步。跟上面的步骤类似，第二跳也被探测出来了。
>
> 发送端一直重复 TTL 加一和 UDP 源端口加一的操作，直到报文最终到达目的地，后者回复一个端口不可达的 ICMP 错误信息（ICMP Port Unreachable）。当源地址收到这个消息时，就停止 traceroute。

> 补充：实际工作中，traceroute 会批量发送多个不同 TTL 值的报文然后等待回应，这样效率更高。

当然，在探测过程中，任何一个探测报文如果在限定时间内没有收到，traceroute 就会认为超时，在输出中这一跳就显示为一个星号。

另外，目的地也可能不回复 ICMP Port unreachable，那么 traceroute 的输出就会出现从某一个位置开始的后续跳数全部是星号，直到你按 Ctrl + C 终止探测，或者是直到默认的 64 跳检测全部完成。对于这种情况，你可以**加上 -I 参数**，让 traceroute 用 ICMP 协议的 echo request 消息进行探测，一般来说，**目的地总能对 UDP 和 ICMP 中的一种进行响应**。

我们看一下原理示意图：



Linux 的 traceroute 命令默认用的就是 UDP 模式，而 Windows 的 tracert 默认用了 ICMP 模式。有时候我们会看到每一跳有 2 个或者 3 个节点 IP，其实就表明这几个路径是启用了 ECMP 的，所以有多个节点会被选到。比如下面这个例子：

复制代码

```
$ traceroute www.ebay.com
traceroute to e9428.a.akamaiedge.net (23.45.61.92), 64 hops max
  1   10.0.2.2  0.308ms  0.192ms  0.353ms
  2   192.168.1.1  4.284ms  1.379ms  1.360ms
  3   100.65.0.1  5.916ms  5.464ms  5.520ms
  4   61.152.53.149  148.051ms  146.005ms  4.126ms
  5   61.152.25.2  16.216ms  61.152.25.101  14.618ms  144.713ms
  6   *  202.97.83.13  8.385ms  *
  7   202.97.12.186  5.112ms  13.341ms  14.592ms
  8   202.97.41.142  49.747ms  54.797ms  202.97.41.130  58.370ms
  9   *  *  *
 10   *  *  *
```

```
13  11   *  *  *
14  12   *  *  *
15  13   *  *  *
```

当你改成用 ICMP 模式后，每一跳就变成只有一个 IP 了。原因我们在上一讲讨论过，就是因为五元组是一致的，所以路径也不变。比如我们还是对 www.ebay.com 进行探测，这次用 ICMP 模式，每跳就只有一个 IP 了，而且目的地也返回了有效的响应。

复制代码

```
1  $ traceroute -I www.ebay.com
2  traceroute to e9428.a.akamaiedge.net (23.45.61.92), 64 hops max
3    1   10.0.2.2  0.003ms  0.002ms  0.003ms
4    2   192.168.1.1  7.616ms  1.315ms  6.206ms
5    3   *  100.65.0.1  11.057ms  7.746ms
6    4   *  61.152.53.149  18.436ms  9.598ms
7    5   *  61.152.25.158  12.918ms  *
8    6   *  *  *
9    7   *  *  *
10   8   202.97.41.142  50.746ms  58.052ms  59.546ms
11   9   203.215.237.22  62.385ms  66.607ms  54.561ms
12  10   23.45.61.92  49.609ms  46.586ms  45.567ms
```

用 mtr 定位丢包点

在上一讲的案例中，我们是用 nc 加上 -p 参数指定多个源端口来测试，定位到了问题路径的存在。这对于问题节点丢包率比较高时，作用比较明显。

但是，当问题节点的丢包率不高的时候，我们用 nc 做多次测试，未必正巧能抓到现场。我们最好还要有**持续监控**的手段，运行一段时间或者发送一定数量（比如 1000 次以上）的报文，通过足够的探测数，来抓到偶发的故障现象。

这里呢，我们就要用到预习篇提到过的工具：**mtr**。mtr 相当于 traceroute 的加强版，特别是它可以指定任意次数的探测并生成详细的探测报告，而 traceroute 只能对每个节点发起 3 次探测，数据量就不够了。

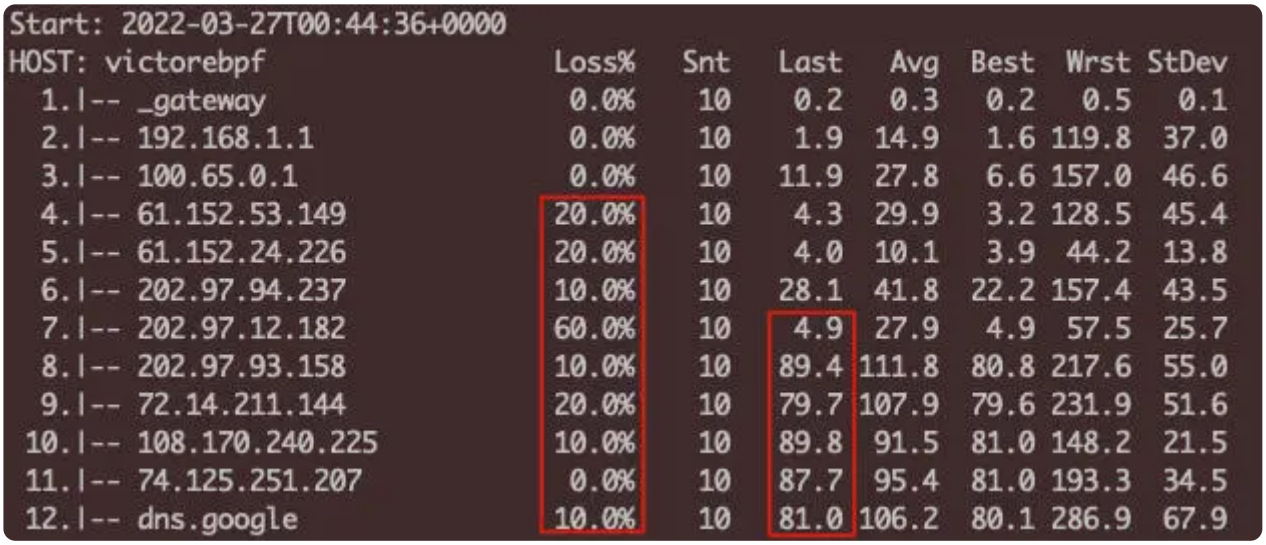
> 你也可以理解为 mtr = traceroute + ping，因为 ping 也可以发送很多次，也有最终的报告。

比如，我们用下面这条命令，就可以对目标 IP 完成指定次数的探测并生成详细的报告，报告里包含了每一跳的丢包率，很有帮助。

复制代码

```
$ mtr -c 10 -r 8.8.8.8
Start: 2022-03-27T00:44:36+0000
HOST: victorebpf                  Loss%   Snt   Last   Avg  Best  Wrst StDev
  1.|-- _gateway                   0.0%    10    0.2   0.3   0.2   0.5   0.1
  2.|-- 192.168.1.1                0.0%    10    1.9  14.9   1.6 119.8  37.0
  3.|-- 100.65.0.1                 0.0%    10   11.9  27.8   6.6 157.0  46.6
  4.|-- 61.152.53.149              20.0%    10    4.3  29.9   3.2 128.5  45.4
  5.|-- 61.152.24.226              20.0%    10    4.0  10.1   3.9  44.2  13.8
  6.|-- 202.97.94.237              10.0%    10   28.1  41.8  22.2 157.4  43.5
  7.|-- 202.97.12.182              60.0%    10    4.9  27.9   4.9  57.5  25.7
  8.|-- 202.97.93.158              10.0%    10   89.4 111.8  80.8 217.6  55.0
  9.|-- 72.14.211.144              20.0%    10   79.7 107.9  79.6 231.9  51.6
 10.|-- 108.170.240.225            10.0%    10   89.8  91.5  81.0 148.2  21.5
 11.|-- 74.125.251.207             0.0%    10   87.7  95.4  81.0 193.3  34.5
 12.|-- dns.google                 10.0%    10   81.0 106.2  80.1 286.9  67.9
```

可见，终点处的丢包率为 10.0%。不过，你也可能注意到了：为什么中间有好几个节点的丢包达到了 20.0% 甚至 60.0%，但终点丢包率反而更低？这是怎么回事？



为何中间节点丢包率更高？

首先，终点的丢包率（在这里是 10.0%）肯定是正确的，因为这就是 ICMP 报文到达终点后的情况。按常规的理解，中间节点的丢包率应该比终点的丢包率低或者持平，但第 4 跳的 20.0% 和第 7 跳的 60.0%，都比终点丢包率更高，这就令人费解了。另外，Last 等列的响应耗时指标，也出现了中间节点和终点倒挂的情况。

其实，这个问题需要我们结合两个知识点来理解：

mtr 探测路径节点的工作机制；

网络设备回复 ICMP 报文的工作机制。

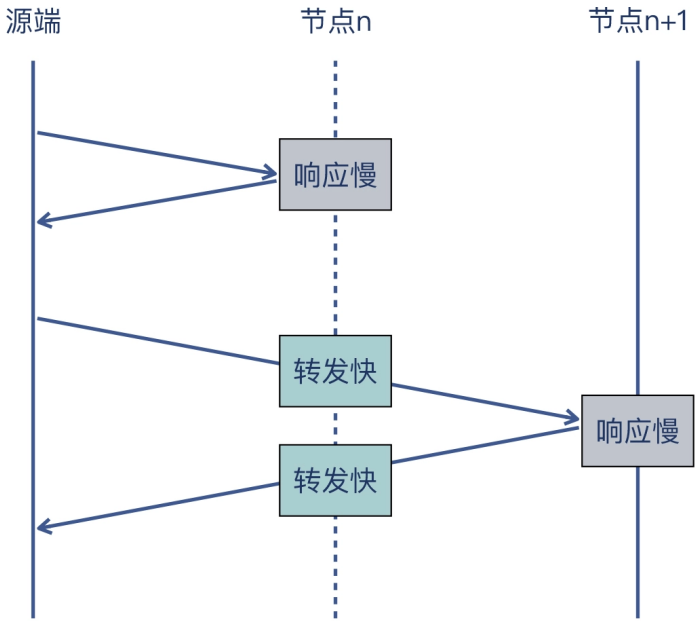
第一个知识点，mtr 跟 traceroute 一样，也是通过递增 TTL 的方式，使得 ICMP 或者 UDP 报文终止在中间节点，从而获取到这些节点回复的 ICMP time exceeded 消息。也正是通过这个消息，我们知道了这个节点的 IP、时延，以及我们关心的丢包率。

第二个知识点，是网络设备在回复 ICMP 报文时候的工作机制。其实，网络设备（交换机和路由器）在“转发”和 “回复”这两个任务上的工作机制是不同的：

对于“转发” ，大部分工作是卸载到数据面的硬件芯片完成的，这也是实现高速转发的底层基础。

对于“回复” ，因为需要自己生成一个 ICMP 响应报文，那就需要动用自己的 CPU 资源了，速度就会慢一些。

我们可以通过下面这个图来理解这个知识点：



极客时间

另外还有一些原因会导致这个现象，比如，**不少网络设备对自己的 ICMP 响应报文设置了限速**（rate limit），这也会加剧这种中间节点丢包的情况。

如何解读中间节点的丢包率？

那么，这个中间节点的丢包率是不是就毫无价值了呢？我们再看另外一个例子：

		Packets			Pings			
	Host	Loss%	Snt	Last	Avg	Best	Wrst	StDev
1.	_gateway	0.0%	292	0.2	0.2	0.1	4.3	0.3
2.	192.168.1.1	0.0%	292	9.5	7.1	1.4	133.2	13.4
3.	100.65.0.1	0.0%	292	8.9	11.6	4.0	136.0	10.2
4.	61.152.54.125	0.3%	292	13.5	10.3	3.3	172.5	16.2
5.	61.152.24.238	29.8%	292	5.1	8.5	3.3	89.1	7.6
6.	**202.97.83.13**	89.7%	291	17.0	11.8	5.4	19.6	4.4
7.	**202.97.39.5**	29.7%	291	11.4	9.9	4.9	60.2	5.7
8.	202.97.94.10	21.3%	291	87.6	89.1	79.5	295.3	25.8
9.	ae-3.r31.tokyjp05.jp.bb.gin.ntt.net	19.3%	291	82.5	89.3	78.1	382.0	30.4
10.	ae-3.r00.tokyjp08.jp.bb.gin.ntt.net	17.5%	291	83.8	89.5	79.1	297.4	27.6
11.	ae-1.fastly.tokyjp08.jp.bb.gin.ntt.net	19.9%	291	81.6	87.5	77.7	235.5	19.1
12.	151.101.109.67	22.3%	291	84.1	89.5	79.7	286.1	24.9

我们从这里可以解读出很多的信息，包括：

从第 5 跳开始到最后一跳都有丢包。其中的第 8 跳到最后一跳的丢包率接近（都在 20% 左右），那么这几跳需要重点关注。

从第 9 到第 11 跳的节点名称解析来看，这些是日本东京（tokyjp）的路由节点，运营商是 NTT。

第 11 跳的名称里带 fastly，这是一个 CDN 厂商的名称，所以这是 fastly CDN 的节点。

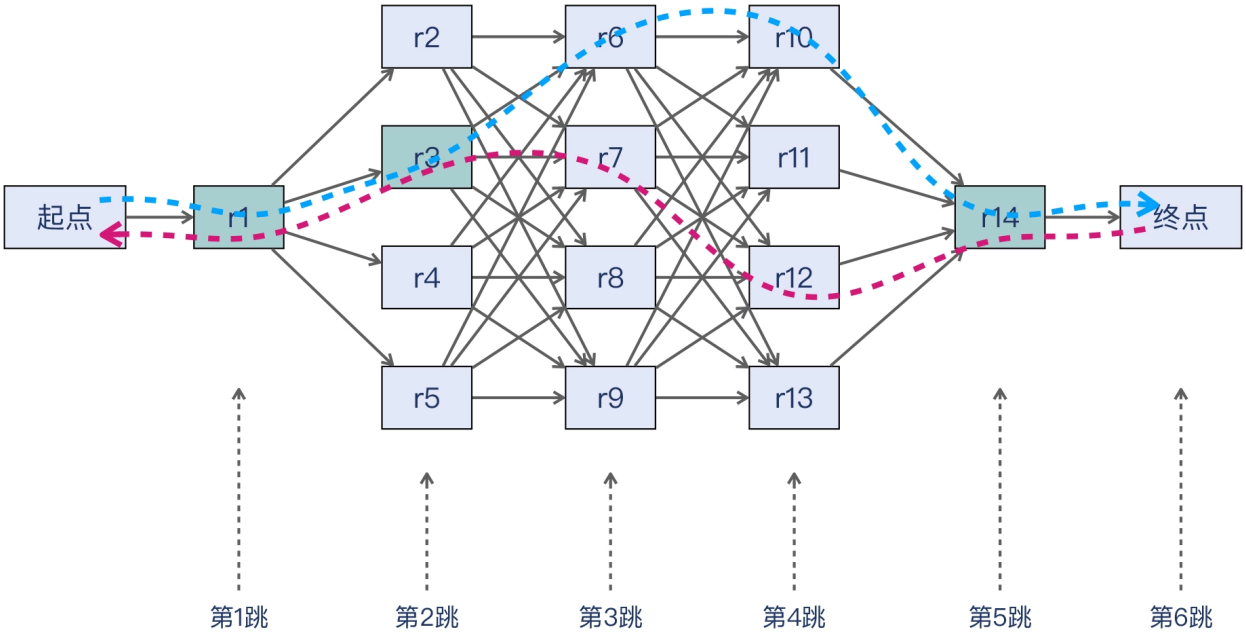
Last 这一列是最近一次响应耗时，第 7 和第 8 跳的耗时突增了 70ms。我们知道，时延跟地理距离成正比，而 70ms 是一个比较长的耗时了，所以我们可以判断，这两跳就是中日之间的海底光缆的两端。

根据这些信息，我们大致可以认为，丢包（本质是链路拥塞）主要发生在中日海底光缆这个位置。

当然，这是一个粗略的推断。**一般来说，mtr 最好在两端都做**，然后结合两边的 mtr 的报告来综合分析，结论会更加准确。

为什么要在两端都做呢？这是因为，网络路由一般是“不对称的”，也就是发送是一条路径，回来往往是另外一条路径了。那么从一端发起 mtr，只能看到这个方向的网络状况，另外一个方向的网络状况，就需要另外一端运行 mtr 来获取了。

比如我们借用一下上一讲的示意图。假设蓝色路径是起点到终点的路径，而紫色的是返回的路径，那么两条路径在 r1，r3，r14 这节点上是重合的，而在其他节点上“各走各的”。



你看，这跟我们抓包经常要在两端同时抓，背后的逻辑是不是差不多的？大道相通，我们积累得越多，越有可能打通这些知识之间的，达成更深的掌握。

丢包与 MTU

在第 8 讲里，我们学习了 MTU 相关的概念。MTU 是最大传输单元，而 PMTU 就是路径上的瓶颈 MTU（最小 MTU）。一旦报文设置了 DF=1 并超出 PMTU 的大小，就会被丢弃。特别是对于**相对较大尺寸的 TCP 报文（比如超过 1400 字节）总是传输失败的情况，可以优先排查 PMTU**。

那么，我们有什么方法可以很方便地就找到 PMTU 呢？

快速探测 PMTU 的方法

其实，我们可以用一个十分简单的命令：ping。给 **ping 命令加上“-s 尺寸”** 这个参数，就可以发送自定义载荷尺寸的报文。比如你可以试试这个命令：

```
ping www.baidu.com -s 1472
```

复制代码

然后再运行：

```
ping www.baidu.com -s 1473
```

复制代码

看看两者的返回有没有区别？

ping 报文属于网络层的 ICMP 控制报文，所以整体的 IP 报文大小是载荷（1472 或者 1473）加上 IP 头部的 20 字节和 ICMP 头部的 8 字节。显然，指定 IP 报文载荷为 1473 后，整个 IP 报文就达到 1501 字节，正好超过了 1500 字节，所以 `ping www.baidu.com -s 1473` 就在发送的途中丢失，也就无法收到 ICMP 响应了。

不过，也许你的网络环境跟我不同，特别是如果有隧道的存在，那 -s 1472 也无法通过。那么你可以不断调整这个值，直到试出一个刚刚能通过的尺寸，然后在它基础上加上 28 字节，就是你的网络环境的 PMTU 了。

> 补充：这里还隐含了另外一个知识点，就是 IP 分片。关于它的细节，你可以回顾第 8 讲的内容。

如何统计丢包率？

最后，我们来学习一下丢包率的统计方法。要统计丢包率，众所周知的方法应该就是用 ping 了，或者说“长 ping”。比如下面的例子里，丢包率为 1%：

```
--- turner-tls.map.fastly.net ping statistics ---
100 packets transmitted, 99 received, 1% packet loss, time 99204ms
rtt min/avg/max/mdev = 122.012/134.749/301.225/27.648 ms
```

复制代码

不过 ping 还是有可能无法探测出网络的真实状况。其实在上一讲里，我们就发现，如果路径中有 ECMP，那么因为 ECMP 哈希转发策略的存在，ping 的网络路径可能就跟 TCP 的网络路径不同。这样可能就会造成这样的状况：

长 ping 没丢包的话，不等于应用也不丢包；

长 ping 丢包的话，应用丢包的可能性也很大。

所以，既然 ping 也未必准，那**不如直接一边跑应用一边抓包，然后对抓包文件进行分析**，从而得出丢包率。而这里，又分了两种途径：图形界面和命令行。

图形界面方法

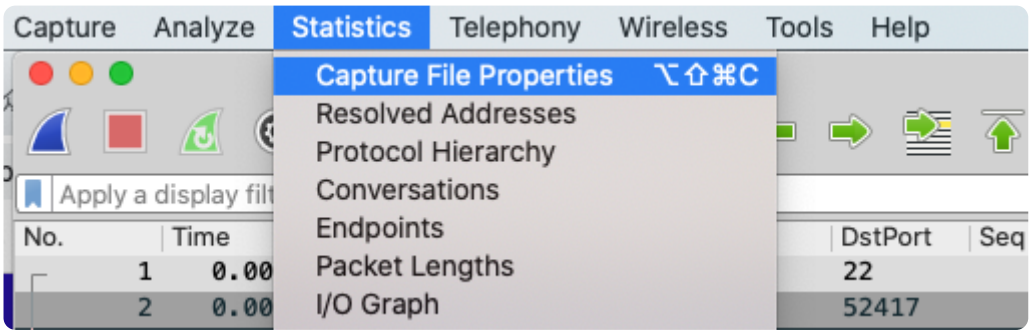
在 Wireshark 中我们能看到 TCP retransmission、Out-of-Order 等信息提示，那么我们就可以借助这些信息，计算出丢包率来。

我们看一个抓包文件的 Expert Information：

Severity	Summary	Group	Protocol	Count
Warning	Connection reset (RST)	Sequence	TCP	50
Warning	Illegal characters found in header name	Protocol	HTTP	50
Note	This frame is a (suspected) retransmission	Sequence	TCP	100
Note	Duplicate ACK (#1)	Sequence	TCP	100
Chat	Connection finish (FIN)	Sequence	TCP	100
Chat	POST /abc/abcdefg/	Sequence	HTTP	100
Chat	Connection establish acknowledge (SYN+ACK): server port 80	Sequence	TCP	50
Chat	Connection establish request (SYN): server port 80	Sequence	TCP	50

在第 4 讲中，我详细介绍过解读 Expert Information 的方法，这里就不赘述了，相信你也已经比较熟悉了。这里我们最关心的指标是 retransmission，它有 100 个。

这些数量算多吗？没有基数就不好说了，所以让我们看一下整体包量。打开 Wireshark 的 Statistics 下拉菜单，选中第一个选项即 Capture File Properties：



在弹出的界面中我们就能看到具体的数据了，比如这个抓包的整体包量是 750 个。

Statistics

Measurement	Captured	Displayed	Marked
Packets	750	750 (100.0%)	—
Time span, s	2.077	2.077	—
Average pps	361.1	361.1	—
Average packet size, B	421	421	—
Bytes	315500	315500 (100.0%)	0
Average bytes/s	151 k	151 k	—
Average bits/s	1215 k	1215 k	—

然后**丢包率可以大致认为是重传率，也就是重传报文数 / 整体报文数**。在这里就是 100/750 = 13.3%。

> 补充：当然，为了方便讨论，这里略过了重复重传的报文被重复计数的情况。

命令行方法

另外一个办法是用 **capinfos 命令**获取总的报文数：

复制代码

```
$ capinfos viaLB.pcap
File name:           viaLB.pcap
File type:           Wireshark/tcpdump/... - pcap
File encapsulation:  Ethernet
File timestamp precision:  microseconds (6)
Packet size limit:   file hdr: 65535 bytes
Packet size limit:   inferred: 84 bytes
Number of packets:   750
File size:           68 kB
Data size:           315 kB
Capture duration:    2.076944 seconds
First packet time:   2016-12-09 20:06:00.629223
Last packet time:    2016-12-09 20:06:02.706167
Data byte rate:      151 kBps
Data bit rate:       1215 kbps
Average packet size: 420.67 bytes
Average packet rate: 361 packets/s
SHA256:              9c34dc15bcf69b419c0e3eb0c37fa851485adbe3953018b957b14de33
RIPEMD160:           c353fdd8d634dd38ac395980b4824751f667908f
SHA1:                fd86e4f6969cc8a27ff1a29cf2de88ab6d57db7d
Strict time order:   True
Number of interfaces in file: 1
Interface #0 info:
                     Encapsulation = Ethernet (1 - ether)
                     Capture length = 65535
                     Time precision = microseconds (6)
                     Time ticks per second = 1000000
                     Number of stat entries = 0
```

```
29                      Number of packets = 750
```

结尾处就是总的报文数量，即 Number of packets = 750。那么重传个数如何在命令行里获取呢？

你是否还记得之前我们学习过 tshark 这个命令？这里也是用 **tshark**，执行下面的命令：

复制代码

```
$ tshark -n -q -r viaLB.pcap -z "io,stat,0,tcp.analysis.retransmission"

===================================
| IO Statistics                   |
|                                 |
| Duration: 2.077 secs            |
| Interval: 2.077 secs            |
|                                 |
| Col 1: tcp.analysis.retransmission |
|---------------------------------|
|                |1               | |
| Interval       | Frames |  Bytes | |
|--------------------------------| |
| 0.000 <> 2.077 |    100 | 145400 | |
===================================
```

最后一行的 Frames 100 就是指重传报文个数 100。同样的，把两个数字相除就得出了丢包率。

丢包多少算严重？

回到重传数量占整体比例的讨论。前面的例子里丢包率是 13.3%。直觉上看，这个比例也高了。那如果降低一个数量级，比如 1.33%，这算高还是低呢？

这个问题可能也没有标准答案，毕竟每个应用对于丢包和重传的敏感度也有不同。而且由于公网情况复杂，本身就有一定的丢包率存在，比如像前面的海底光缆拥塞造成的丢包，也难以避免。

实际上，公网丢包率在 1% 左右是一个可以接受的范围。如果明显超过 1%，比如达到了 5% 以上，那对应用的影响就会比较明显了，此时应该通过节点修复或者链路调整，来解决丢包的问题，把丢包率控制在 1% 左右，最好是 1% 以下。

而内网网络会比公网稳定很多。一般来说，**一个正常的内网也有万分之一左右的丢包率**。如果明显超过了这个比率，比如达到了千分之一的话，尽管依然比公网丢包率低一个数量级，但也需要认真对待并解决。

小结

这节课，我们一起探讨了丢包这个关键话题的工具、原理，还有方法。我们学习了多种工具，包括：

探测类工具：ping、mtr、traceroute、nc、telnet 等。

统计类工具：netstat、ss、nstat。

其中，netstat 和 nstat 都会展示 **TCP retransmission 的数值，如果它随时间递增，那就说明这台主机的对外通信存在丢包的现象**。我们也了解了 Linux 内核本身实现了 SNMP 计数器，这些计数器记录了系统启动后的各个网络指标的数值，而 netstat 等工具正是读取内核中这些计数器，获得了相应指标的数值。

另外我们也要清楚一点，ss 能提供比 netstat 更多的网络信息，特别是 TCP socket 的各种属性。这些信息对 TCP 方面的排查工作很有帮助，比如运行：

复制代码

```
1 ss -iet
```

而对于路径排查的重要工具 **mtr** 的原理和使用方法，你也要好好掌握。在 mtr 的输出中，要注意中间节点的丢包率。如果终点的丢包率比前面节点的低，那前面节点的丢包率只能作为有限的参考。如果节点丢包率出现越往后越高的情况，这样的丢包率的参考价值就高很多了。

而且，最好**在通信两端都做 mtr，然后结合两边的探测结果做综合分析**。

另外，在丢包这个主题中，**MTU** 也是重要角色。因为路径 MTU 的不一致，某些情况下就会发生超出 MTU 而丢包的现象。由于 ICMP 消息不一定会回到发送方，就会导致发送方的 PMTU 机制不能正常工作，也就是无法感知到这个 MTU 超限的事实，导致连续发送失败。

一般我们会建议对自己网络情况了解清楚后，再对主机设置合适的 MTU，这样可以最大程度上确保不发生 MTU 超限引发问题。另外呢，我们也可以**用 ping 来探测 PMTU**，也就是运行：

```
1 ping -s 1472    #或者其他数值
```

复制代码

最后，我们还学习了如何根据抓包文件，计算出丢包率，也就是**丢包率 = 重传个数 / 总报文数**。另外公网丢包率 1% 和内网丢包率万分之一，可以认为是正常范围。

思考题

最后，再给你留两道思考题：

Linux 中运行 ifconfig -a 看到的网卡接口的 dropped 指标，是指我们这里讨论的丢包吗？为什么？

TCP 是如何在发生丢包的情况下保证传输可靠性的呢？

欢迎你在留言区分享你的答案，我们一同进步、成长。

上一篇 23 | 路径排查：没有网络设备权限要如何做排查？

下一篇 不定期加餐（一）| 八仙过海，各显神通：透传真实源IP的各种方法

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。