05 | 定位防火墙（一）：传输层的对比分析

2022-01-21 杨胜辉

《网络排查案例课》 课程介绍 >

讲述：杨胜辉

时长 23:50 大小 21.83M

你好，我是胜辉。今天我们来聊一个有趣的话题：防火墙。

在网络排查中，防火墙作为一个隐形神秘的存在，时常给排查工作带来一定的不确定性。有时候，你不知道为什么一些网络包能正常发出，但对端就是没收到。有时候，同样的两端之间，有些连接可以通信，有些就是不行。

这个时候，你很可能会怀疑是防火墙在从中作祟了，但是你有什么证据吗？

你不是防火墙工程师，就没有查看它的配置的权限。但是这样一个看不见摸不着的东西却可能正在影响着你的应用。相信你也一定想彻底转变被动的状态，把问题搞定。

其实，无论防火墙有多么神秘，**它本质上是一种网络设备**。既然是网络设备，那么它必然同样遵循我们知道的技术原理和网络规范。所以，防火墙的踪迹，虽然表面上给人一种虚无缥缈的感觉，但从理论上说，总是有迹可循的。所以这次，我就帮助你抓住防火墙的蛛丝马迹。

当然，防火墙的排查技巧确实并不简单，为了把这个主题讲透，我将用两节课的时间，来给你专门讲解这方面的排查技巧：一节是结合传输层和应用层的分析推理，一节是聚焦在网络层的精确打击。相信你通过这两讲的学习，一定能掌握不少独门秘技，从而为你的应用保驾护航。

今天这一讲，我们先学第一种方法：结合传输层和应用层的分析推理。

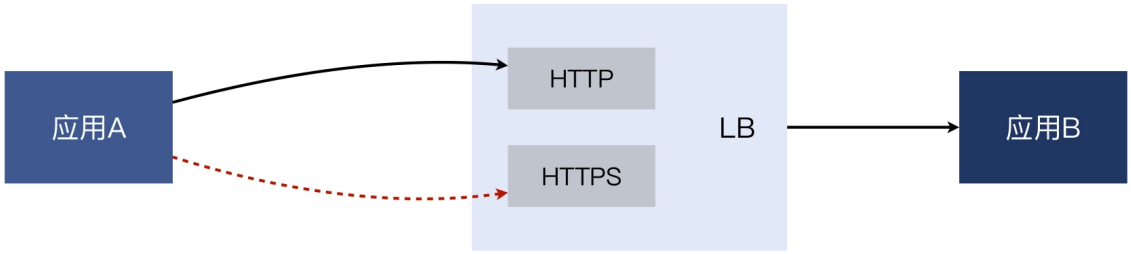
结合传输层和应用层的分析推理

这里传输层当然就是指 TCP/UDP，应用层就是问题表象，比如超时、报错之类。我们来看一个具体的例子。

这是我在 2017 年处理的一个案例。当时 eBay 内部的一个应用 A 访问应用 B 的时候，经常耗时过长，甚至有时候事务无法在限定时间内完成，就导致报错。而且我们发现，问题都是在访问 B 的 HTTPS 时发生的，访问 B 的 HTTP 就一切正常。

因为应用 A 对时间比较敏感，开发团队希望能既解决事务失败的问题，也改善事务处理慢的问题，于是我们运维团队开始调查。

那么，我们先来看一下这个案例中，应用请求路径的大体示意图：



极客时间

应用 A 和应用 B 各自都是一组独立的集群（多台服务器）。A 的众多机器，都访问 B 的位于负载均衡（LB）上的 VIP（虚拟 IP），然后 LB 再把请求转发给 B 的机器。这里的

HTTP 和 HTTPS 都位于同一个虚拟 IP，只是服务端口不同而已（一个是 80，一个是 443）。

既然问题是“A 觉得 B 很慢”，那么，我们除了**听听 A 的抱怨，是否也该问问 B 的解释，才显得比较公正呢？**这就是我们选择做“两侧抓包”的背后的考量了。

否则，假如我们只是在客户端（即 A 应用）上抓包，看到的报文显然也属片面。比如，客户端看到自己发出的报文迟迟未被服务端确认的话，那么这个报文究竟是丢失在网络路径途中，还是已经到达服务端但是被服务端丢弃了呢？显然，只在客户端抓包，是无法把这些事实弄得很清楚的。

所以，我们就在 A 中选择了一台机器（即客户端）做 tcpdump 抓包，同时在 LB 的 HTTPS VIP（即服务端）上也进行抓包。

这里我要给你友情提醒一下，做这种双向抓包，需要注意：

各端的抓包过滤条件一般以对端 IP 作为条件，比如 tcpdump dst host {对端 IP}，这样可以过滤掉无关的流量。

两端的抓包应该差不多在同时开始和结束，这样两端的报文就有尽量多的时间是重合的，便于对比分析。

在同时抓包的时间段内，要把问题重现，也就是**边重现，边抓包**。至于如何重现，又分两种情况：一种是我们知道触发条件，那么直接操作发起就好了；另一种是触发条件未知，那么只有在抓包的同时，耐心等待问题出现，然后再停止抓包。

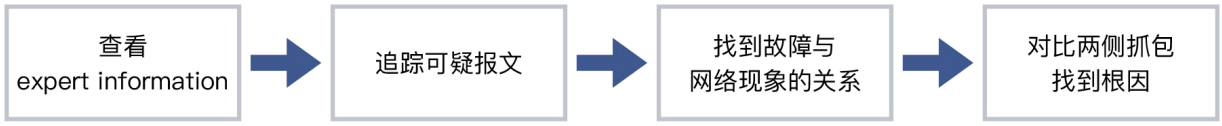
这次我们也是如此处理：一边抓包，一边跟开发团队配合观察应用日志。当观察到了日志中有意外事件（exception）出现后，停止了抓包。那么在这段抓包里，应该就含有跟这个意外事件相关的报文了。

我们先来看一下客户端的报文情况。打开抓包文件后，我一般会按部就班地做以下几件事：

查看 expert information；

重点关注可疑报文（比如 Warining 级别），追踪其整个 TCP 流；

深入分析这个可疑 TCP 流的第二到四层的报文，再结合应用层表象，综合分析其根因；结合两侧的抓包文件，找到属于同一个 TCP 流的数据包，对比分析，得出结论。



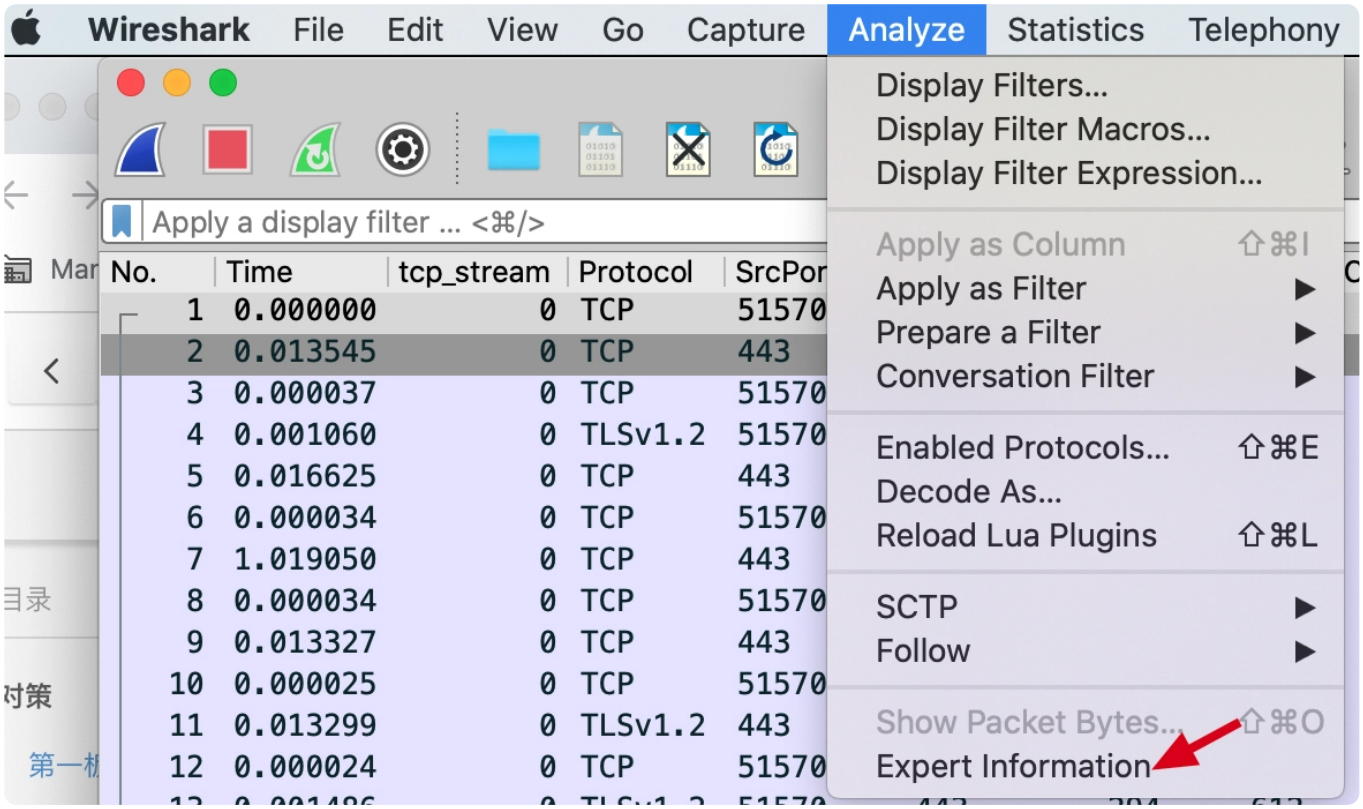
极客时间

那么接下来，我们就根据以上的步骤，来具体分析分析当前这个抓包文件。

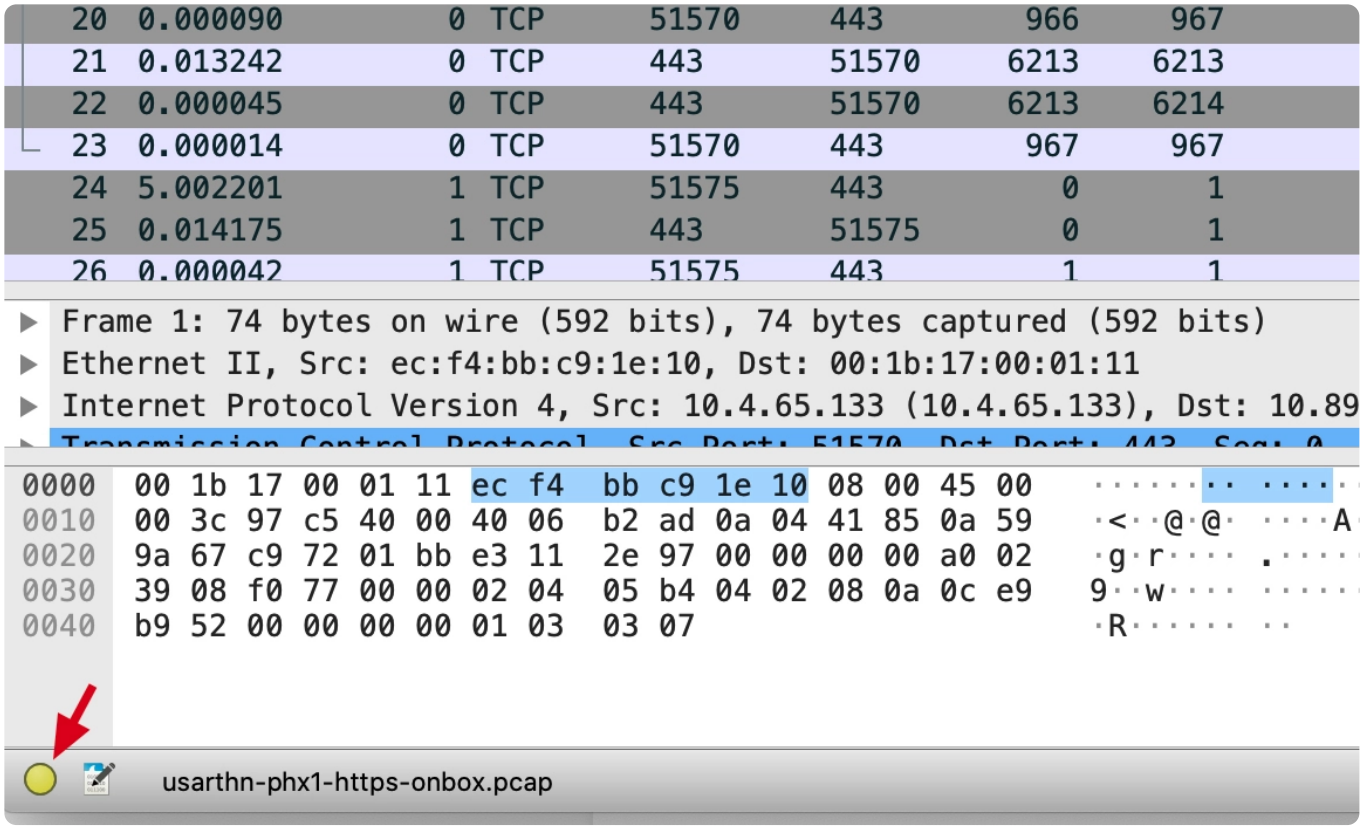
查看 expert information

这一步主要是为了获取整体的网络传送情况，这对于我们大体判断问题方向很有帮助。

那么在查看的时候，一种方式是打开 Analyze 菜单，选择最底部的 expert information，如下图所示：



另一种方式是直接在窗口左下角，点击那个黄色的小圆圈，如下图所示：

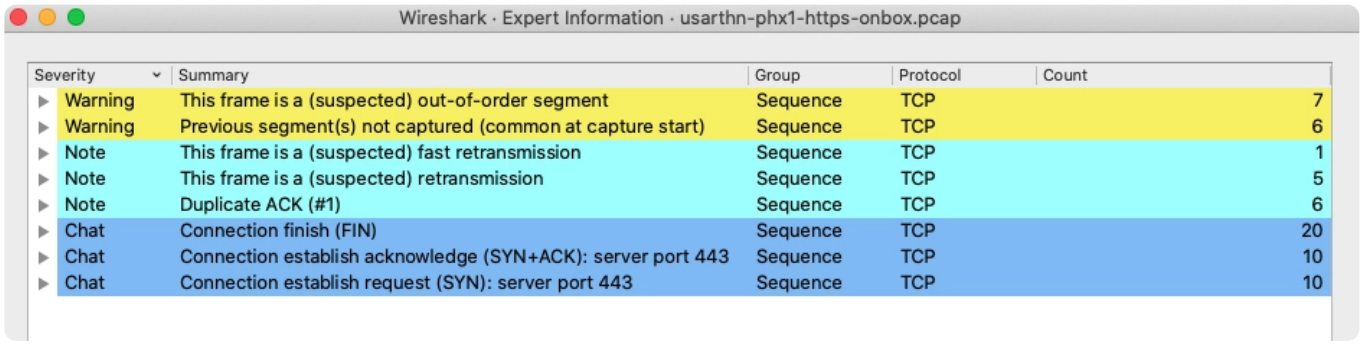


不过，我们要怎么理解 expert information 里面的各种信息呢？我来给你挨个介绍一下：

Warning 条目的底色是黄色，意味着可能有问题，应重点关注。

Note 条目的底色是浅蓝色，是在允许范围内的小问题，也要关注。什么叫“允许范围内的小问题”呢？举个例子，TCP 本身就是容许一定程度的重传的，那么这些重传报文，就属于“允许范围内”。

Chat 条目的底色是正常蓝色，属于 TCP/UDP 的正常行为，可以作为参考。比如你可以了解到，这次通信里 TCP 握手和挥手分别有多少次，等等。



上图展示的就是客户端抓包文件的情况，我们逐个来解读这三种不同级别的 Severity（严重级别）。

Warning：有 7 个乱序（Out-of-Order）的 TCP 报文，6 个未抓到的报文（如果是在抓包开始阶段，这种未抓到报文的情况也属正常）。

Note：有 1 个怀疑是快速重传，5 个是重传（一般是超时重传），6 个重复确认。

Chat：有 TCP 挥手阶段的 20 个 FIN 包，握手阶段的 10 个 SYN 包和 10 个 SYN+ACK 包。

一般来说，**乱序**是应该被重点关注的。因为正常情况下，发送端的报文是按照先后顺序发送的，如果到了接收端发生了乱序，那么很可能是中间设备出现了问题，比如中间的交换机路由器（以及这节课的主角防火墙）做了一些处理，引发了报文乱序。

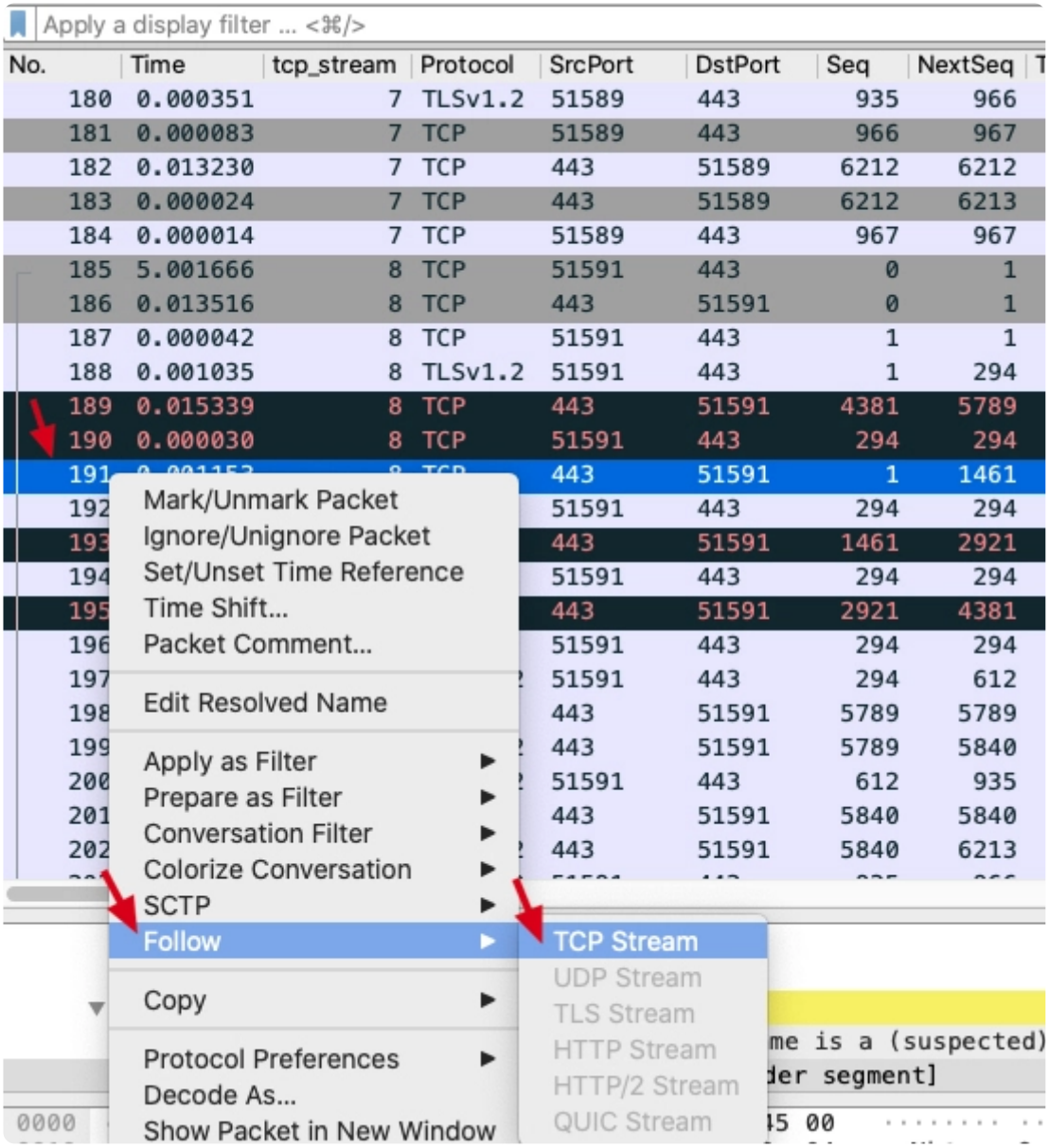
所以自然，这个乱序也容易引发应用层异常。

重点关注问题报文

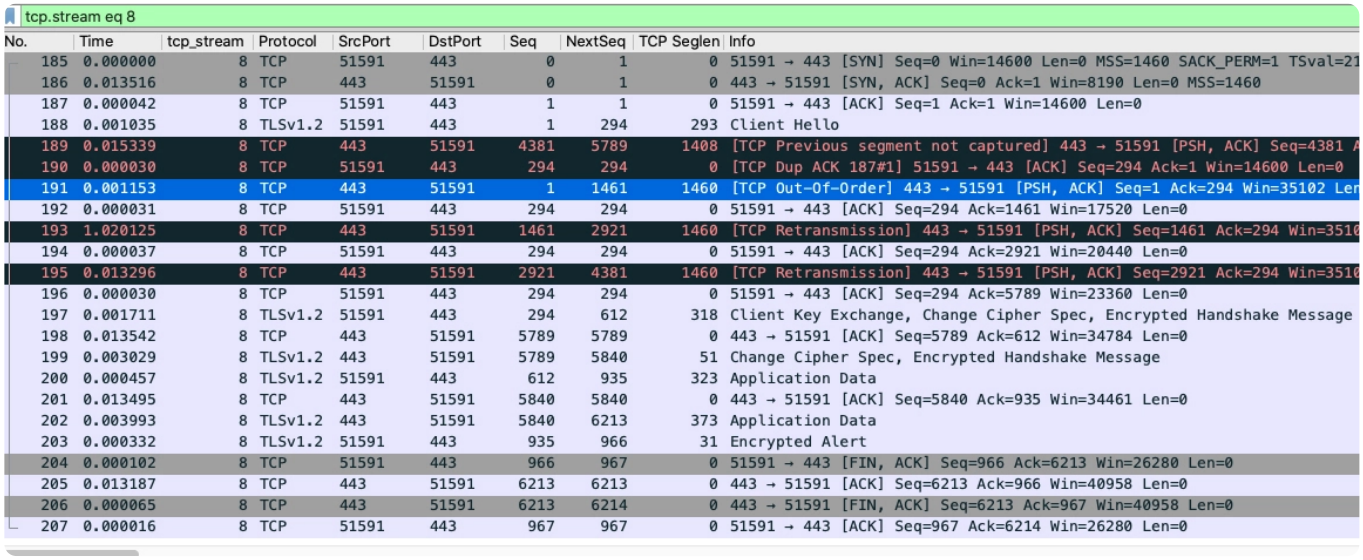
理解了这几个严重级别所代表的含义之后，我们还是回到 expert information 的进一步解读上来。点击 Warning 左边的小箭头，展开乱序的报文集合，我们就能看到这些报文的概览信息，如下所示：

Packet	Summary	Group	Protocol	Count
Warning	This frame is a (suspected) out-of-order segment	Sequence	TCP	7
122	[TCP Out-Of-Order] 443 → 51584 [PSH, ACK] Seq=1 Ack=294...	Sequence	TCP	
168	[TCP Out-Of-Order] 443 → 51589 [PSH, ACK] Seq=1 Ack=294...	Sequence	TCP	
170	[TCP Out-Of-Order] 443 → 51589 [PSH, ACK] Seq=1461 Ack=...	Sequence	TCP	
172	[TCP Out-Of-Order] 443 → 51589 [PSH, ACK] Seq=2921 Ack=...	Sequence	TCP	
191	[TCP Out-Of-Order] 443 → 51591 [PSH, ACK] Seq=1 Ack=294 ...	Sequence	TCP	
214	[TCP Out-Of-Order] 443 → 51593 [PSH, ACK] Seq=1 Ack=294...	Sequence	TCP	
216	[TCP Out-Of-Order] 443 → 51593 [PSH, ACK] Seq=1461 Ack=...	Sequence	TCP	

我习惯上会选择靠后一点的报文（因为相对靠后的报文所属的 TCP 流相对更完整），然后跟踪这些报文（Follow -> TCP Stream），找到所属的 TCP 流来进一步分析。比如选中 191 号报文，这时主窗口自动定位到了这条报文，我们在主窗口中选中该报文后右单击，选择 Follow，在次级菜单中点击 TCP Stream：



然后，我们就能看到过滤出来的这个 TCP 流的全部报文了！



细心的你可能会发现，界面里很多字段不是默认有的，比如 Seq、NextSeq、TCP Seglen 等等，这些其实都是我自定义添加的，目的就是**便于分析**（添加的办法我会在下一讲里介绍）。

这时，Wireshark 的显示过滤器栏出现了`tcp.stream eq 8`这个过滤器，这是我们刚刚点击 Follow -> TCP Stream 后自动生成的。

一眼看去，整串数据流确实有点问题，因为有好几个被 Wireshark 标注红色的报文。我们重点关注下 189、190、191、193、195 这几个报文。

189：服务端（HTTPS）回复给客户端的报文，TCP previous segment not captured 意思是，它之前的报文没有在应该出现的位置上被抓到（并不排除这些报文在之后被抓到）。

190：客户端回复给服务端的重复确认报文（DupAck），可能（DupAck 报文数量多的话）会引起重传。

191：服务端（HTTPS）给客户端的报文，是 TCP Out-of-Order，即乱序报文。

193：服务端（HTTPS）给客户端的 TCP Retransmission，即重传报文。

195：也是服务端（HTTPS）给客户端的重传报文。

以上都是根据 Wireshark 给我们提示的信息所做的一些解读，主要是针对 TCP**行为**方面的，这也是从 Wireshark 中读取出来的重要信息之一。另外一个重要的信息源是**耗时**（也就是时间列展示的时间间隔）。显然，在 192 和 193 号报文之间，有 1.020215 秒的时间间隔。

要知道，对于内网通信来说，时间是以毫秒计算的。**一般内网的微服务的处理时间，等于网络往返时间 + 应用处理时间**。比如，同机房环境内，往返时间（Round Trip Time）一般在 1ms 以内。比如一个应用本身的处理时间是 10ms，内网往返时间是 1ms，那么整体耗时就是 11ms。

然而，这里单单一个 193 号报文就引发了 1 秒的耗时，确实出乎意料。因此我们可以基本判定：**这个超长的耗时，很可能就是导致问题的直接原因。**

结合应用层做深入分析

那么，为什么会有这个 1 秒的耗时呢？

TCP 里面有**重传超时**的设计，也就是如果发送端发送了一个数据包之后，对方迟迟没有回应的话，可以在一定时间内重传。这个“一定时间”就是 TCP 重传超时（Retransmission Timeout）。显然，这里的 1 秒，很可能是这个重传超时的设计导致的。

为了确认这件事，我们就需要做这次分析里最为关键的部分了：**两端报文的对比分析**。我们最好有一个大一点的显示屏，打开这两个抓包文件，并且把两个 Wireshark 窗口靠近一些，更方便我们肉眼对两边报文进行比较。

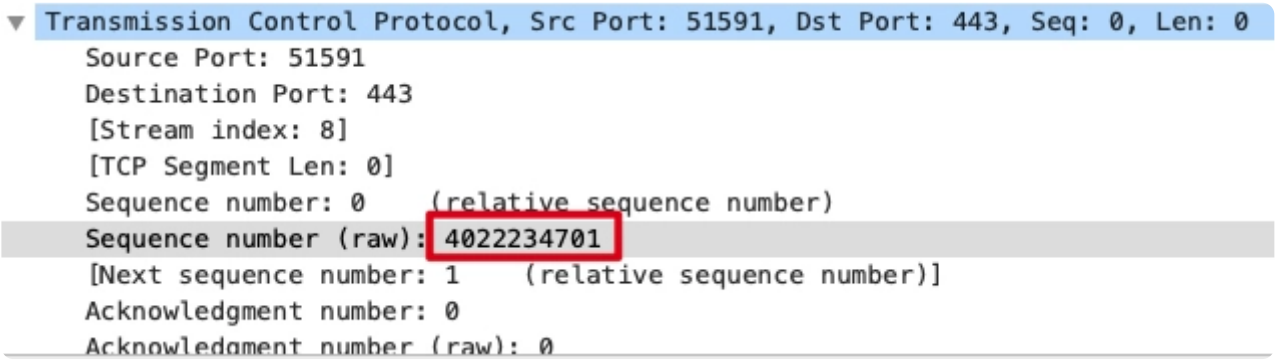
不过，当我们打开服务端（LB）的抓包文件，看到的却也是一大片报文。而要比较，必然要找到同样的报文才能做比较。这也是一个不小的难点：如何才能在服务端抓包文件里，定位到客户端的 TCP 流呢？我们接着往下看。

对比两侧文件

其实，找到另一端的对应 TCP 流的技巧是：**用 TCP 序列号**。

我们知道，TCP 序列号的长度是 4 个字节，其本质含义就是网络 IO 的字节位置（等价于文件 IO 的字节位置）。因为是 4 个字节，最大值可代表 4GB（即 2 的 32 次方）的数据，也就是如果一个流的数据超过 4GB，其序列号就要回绕复用了。不过一般来说，因为这个值的范围足够大，在短时间（比如几分钟）碰巧相同的概率几乎为零，因此我们可以以它作为线索，来精确定位这个 TCP 流在两端抓包文件中的位置。

首先，我们可以记录下客户端侧抓包文件中，那条 TCP 流的某个报文的 TCP 序列号。比如选择 SYN 包的序列号，是 4022234701：



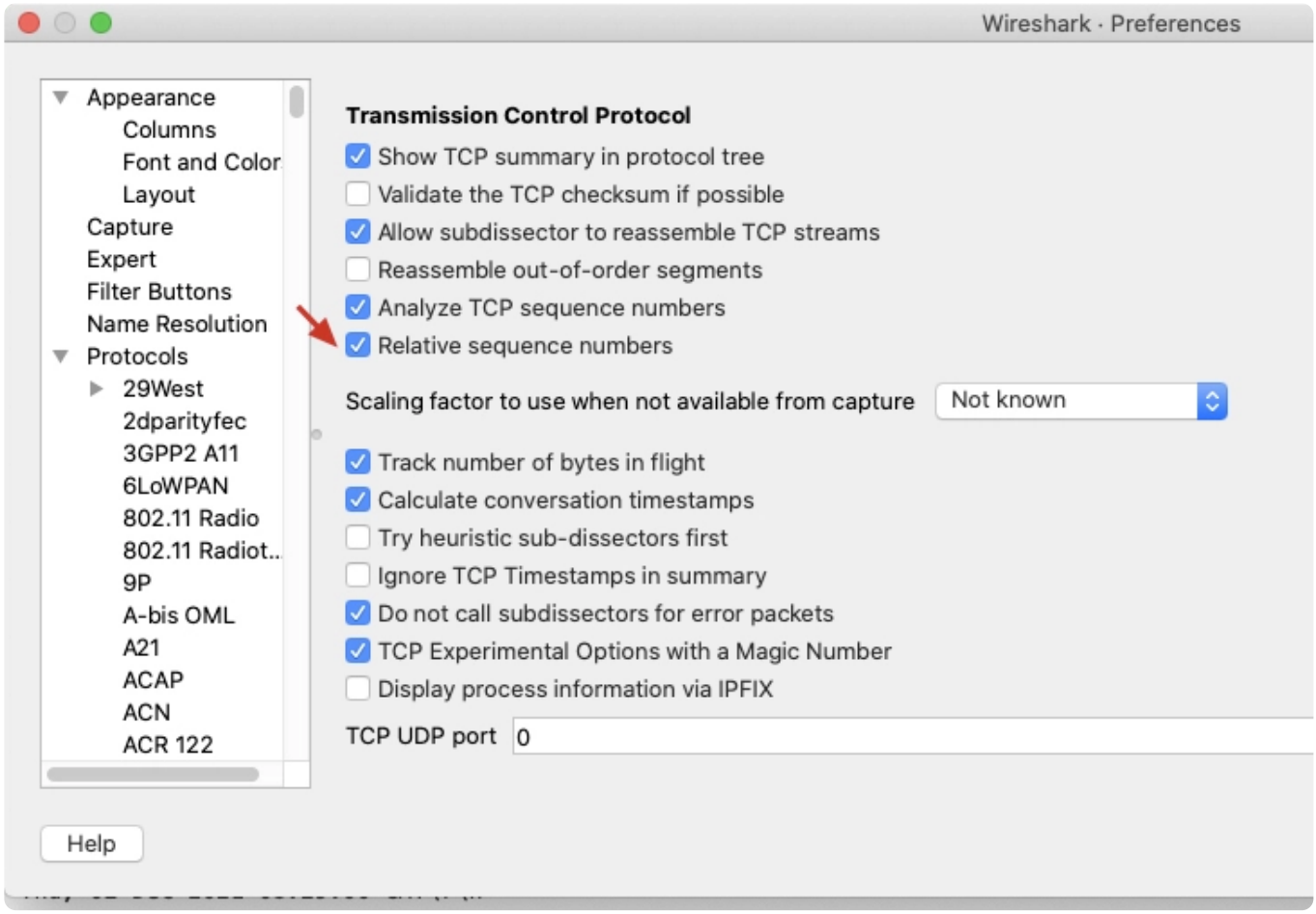
注意，这里必须选**裸序列号**（Raw Sequence Number）。Wireshark 主窗口里显示的序列号是处理过的“相对序列号”，也就是为了方便我们阅读，把握手阶段的初始序列号当 1 处理，后续序列号相应地也都减去初始序列号，从而变成了相对序列号。

但是显然，这样处理后，无论在哪个 TCP 流里面，Wireshark 展示的握手阶段序列号都是 1，后续序列号也都是 1+ 载荷字节数。相对序列号肯定是到处“撞车”的，所以不能作为选取的条件。

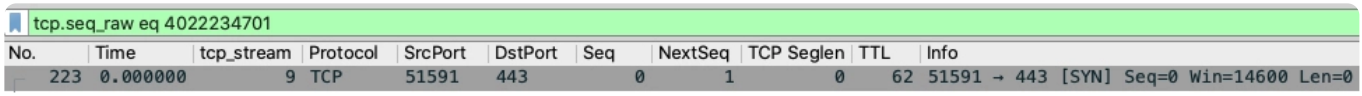
那么，查看裸序列号的方法是怎么样的呢？

打开 Wireshark 的 Preference（配置）菜单：

在弹出菜单的左侧选择 Protocols，选中其中的 TCP，然后在右侧的选项中，把“Relative sequence numbers”前面的勾去掉，就可以显示裸序列号了：



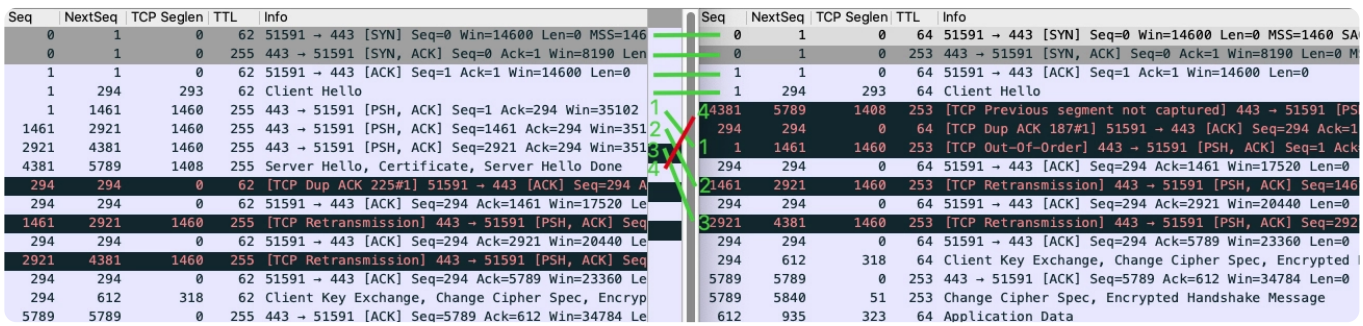
然后，我们再到服务端抓包文件里输入过滤器：`tcp.seq_raw eq 4022234701`，得到同样的这个 SYN 包：



正是我们的搜索条件是裸序列号，所以打开服务端抓包文件的那个 Wireshark 窗口里才可以搜到这个报文。这也是利用了裸序列号在网络上传输是不会发生变化的这一特性。

接下来还是 Follow -> TCP Stream，翻出来这个 SYN 包所属的服务端抓包里的 TCP 流。

好了，现在我们睁大眼睛，来仔细比对这些报文的对应情况：

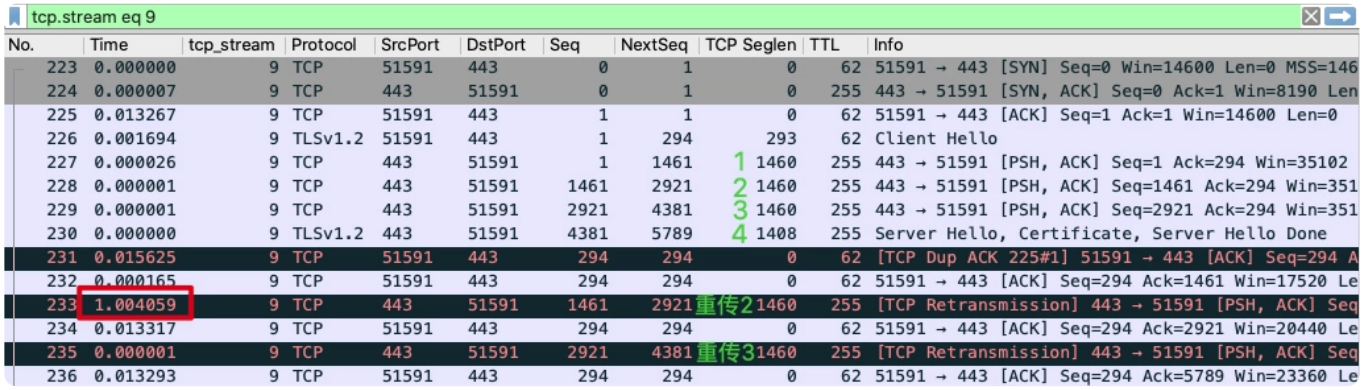


左侧是服务端抓包，右侧是客户端抓包。前 4 个报文的顺序没有任何变化，但服务端随后一口气发送的 4 个包（这里叫它们 1、2、3、4 吧），到了客户端却变成了 4、1、2、3！这也就是 Wireshark 提示我们的：

Out-of-Order：包 1、2、3。

TCP previous segment not captured：包 4。

下面，**我们再从服务端的角度**，来看一下报文顺序、重传、1 秒耗时这三者间的关系：



前面刚说到“服务端发送 4 个报文后，客户端收到的是 4、1、2、3”。因为后面 3 个报文的顺序还是正确的，真正乱序的其实只是 4，所以就导致了这样一个状况：乱序是乱序的，但是“不够乱”，也就是不能满足快速重传的条件“3 个重复确认”。

这样的话，服务端就不得不用另外一种方式做重传，即**超时重传**。当然，这里的 1 秒超时是硬件 LB 的设置值，而 Linux 的默认设置是 200 毫秒。

不过，撇开这些细节不谈，我们现在知道了一个重要的事实：**客户端和服务端之间，有报文乱序的情况**。

我们查看了其他 TCP 流，也有很多类似的乱序报文，而这种程度的乱序发生在内网是不应该的，因为内网比公网要稳定很多。以我个人的经验，内网环境常见的丢包率在万分之一上下，乱序的几率我没有严格考证过，因为跟各个环境的具体拓扑和配置的关系太大了。但从经验上看，乱序几率大概在百分之一以下到千分之一左右都属正常。

我们把这两个抓包文件以及分析过程和推论，发给了网络安全部门。他们对于实际的抓包信息也很重视，经过排查，发回了一个我们“期待已久”但一直无法证实的推测：问题出现在防火墙上！

具体来说，是这样两个事实：

1. 在客户端和服务端之间，各有一道防火墙，两者之间设立有隧道；
2. 因为软件 Bug 的问题，这个隧道在大包的封包拆包的过程中，很容易发生乱序。

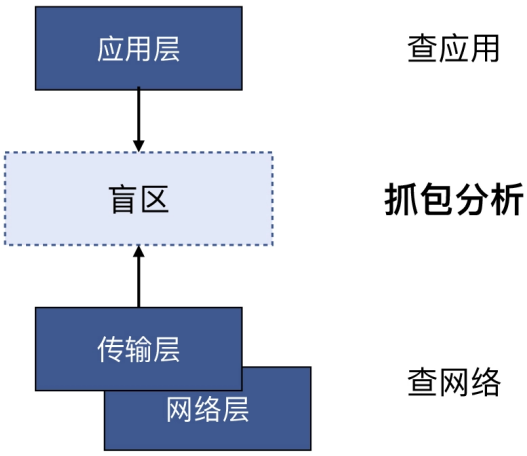
就像下图这样：

极客时间

两侧抓包，对比分析。就是这样一个方法，最终让我们发现了防火墙方面的问题。我们做了设备升级，效果是立竿见影，事务慢的问题完全消失了。开发团队也非常感谢我们的排查工作。

那么换位思考，你如果是开发团队，你会查哪些层面呢？对，一般是集中在应用程序上，可能也会做一点 ping、telnet、traceroute 之类的检查。但术业有专攻，难以把排查工作下探到传输层和网络层。

而如果你是网络安全团队，你又会怎么查呢？没错，也是集中在网络层或者传输层上面，不太会把排查上升到应用层。



极客时间

显然，两边团队没法“会合”。这种状况，在很多组织里并不少见。不过还好，我所在的团队正好既熟悉网络也熟悉应用，能把网络和应用之间的联动细节搞清楚，进而排查出根因。故事也算有了一个圆满的结局。你了解了整个排查过程，是否也有所启发呢？

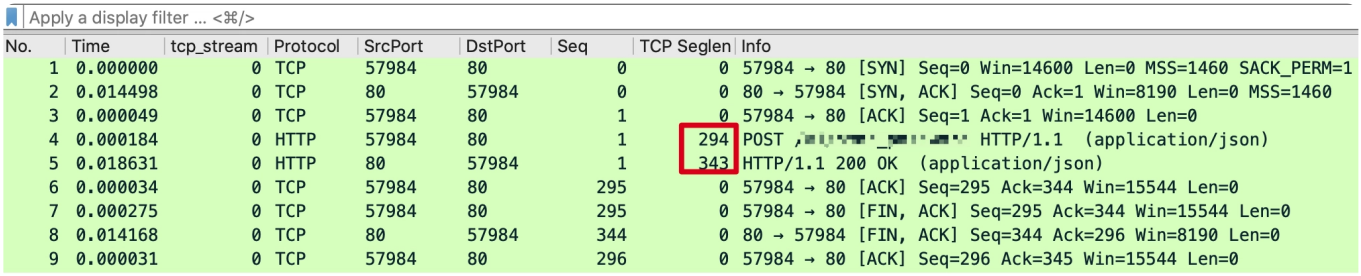
不过，这里我们还有两个小的疑问没有解决：

为什么隧道会引发乱序？

首先，隧道本身并不直接引起乱序。隧道是在原有的网络封装上再加上一层额外的封装，比如 IPIP 隧道，就是在 IP 头部外面再包上一层 IP 头部，于是形成了在原有 IP 层面里的又一个 IP 层，即“隧道”（各种隧道技术也是 SDN 技术的核心基础）。由于这个封装和拆封都会消耗系统资源，加上代码方面处理不好，那么出 Bug 的概率就大大增加了。这就是在这个案例里，隧道会引发乱序的原因。

为什么 HTTP 事务没有被影响，只有 HTTPS 被影响？

在这个案例里，HTTP 确实一直没有被影响到。因为从抓包来看，这个场景的 HTTP 的 TCP 载荷，其实远没有达到一个 MSS 的大小。我们来看一下当时的 HTTP 抓包：


Apply a display filter ... <⌘/>

No.	Time	tcp_stream	Protocol	SrcPort	DstPort	Seq	TCP Seglen	Info
1	0.000000	0	TCP	57984	80	0	0	57984 → 80 [SYN] Seq=0 Win=14600 Len=0 MSS=1460 SACK_PERM=1
2	0.014498	0	TCP	80	57984	0	0	80 → 57984 [SYN, ACK] Seq=0 Ack=1 Win=8190 Len=0 MSS=1460
3	0.000049	0	TCP	57984	80	1	0	57984 → 80 [ACK] Seq=1 Ack=1 Win=14600 Len=0
4	0.000184	0	HTTP	57984	80	1	294	POST /[redacted] HTTP/1.1 (application/json)
5	0.018631	0	HTTP	80	57984	1	343	HTTP/1.1 200 OK (application/json)
6	0.000034	0	TCP	57984	80	295	0	57984 → 80 [ACK] Seq=295 Ack=344 Win=15544 Len=0
7	0.000275	0	TCP	57984	80	295	0	57984 → 80 [FIN, ACK] Seq=295 Ack=344 Win=15544 Len=0
8	0.014168	0	TCP	80	57984	344	0	80 → 57984 [FIN, ACK] Seq=344 Ack=296 Win=8190 Len=0
9	0.000031	0	TCP	57984	80	296	0	57984 → 80 [ACK] Seq=296 Ack=345 Win=15544 Len=0

TCP 载荷只有两三百字节，远小于 MSS 的 1460 字节。这个跟隧道的关系是很大的，因为**隧道会增加报文的大小**。

比如通常 MTU 为 1500 字节的 IP 报文，做了 IPIP 隧道封装后，就会达到 1520 字节，所以一般有隧道的场景下，主机的 MTU 都需要改小以适配隧道需求。如果网络没有启用 Jumbo Frame，那这个 1520 字节的报文，就会被路由器 / 防火墙拆分为 2 个报文。而到了接收端，又得把这两个报文合并起来。这一拆一合，出问题的概率就大大增加了。

> 补充：在 Linux 中，设置了 ipip 隧道后，这个隧道接口的 MTU 会自动降低 20 字节，也就是从默认 1500 降低到 1480 字节。

> 这个案例里是特殊的防火墙，它的 MTU 的逻辑跟 Linux 有所不同。

事实上，在大包情况下，这个隧道引发的是两种不同的开销：

IPIP 本身的隧道头的封包和拆包；

IP 层因为超过 MTU 而引发的报文分片和合片。

因为 HTTPS 是基于 TLS 加密的，TLS 握手阶段的多个 TCP 段（segment）就都撑满了 MSS（也就是前面分析的 1、2、3 的数据包），于是就触发了防火墙隧道的 Bug。

到这里，你可能又会问了：这个例子中的丢包和乱序问题，其实也不限于防火墙，在路由器交换机层面也是有可能发生的，有没有办法可以更加确定地定位到防火墙，而不是其他网络设备呢？

这就是我们在下节课要进行深入解析的内容了，即聚焦在网络层的精确打击，你可以期待一下。

小结

这次的“两侧抓包”，实际上就起到了决定性的作用。那么除了当前这个案例，总的来说，还有**哪些情况下适合做“两侧抓包”**呢？我个人的看法是这样的：

1. **有条件的话（比如对两侧设备都有权限），就尽量做两侧抓包。**这样可以收集到更多的信息。有更全面的信息，也就更容易作出更准确的判断。这好比我们做数学或者物理题，条件越充足，解题也相对越顺利。即使信息有所富余，也不会干扰到排查工作的正确性。当然这会损失一些效率，就看你怎么权衡了。
2. **有丢包或者重传的情况的话，更应该做两侧抓包。**因为只有通过比较两侧报文，才能确定具体的丢包位置等信息，而这些信息对于排查工作十分关键。我们经常会出现的情况是，完全不同的两种故障原因，在一侧（比如客户端）看起来很可能是相同的现象。这就好比，一个一半黑一半白的球体，当其中的一面正对着我们的时候，我们是完全不知道另外一面可能是完全不同的颜色。对于网络排查也是如此。
3. **有些信息在单侧抓包里就能明确下来的，一般就没必要做两侧抓包了。**比如下节课要讲的方法就是这样，这里先给你卖个关子，下节课我们再深入探讨。

另外，在课程中我还介绍了**在两个不同的抓包文件中如何定位到同个报文的方法，也就是使用裸序列号**。

在一侧的文件中找到某个报文的裸序列号，作为搜索条件，在另外一侧的报文中搜索得到同样这个报文。这正是利用了 TCP 裸序列号在网络中传输的一致性（不变性）。后面的课程中，我还将介绍更多这种“寻找同样报文”的方法 ，基本思想也都是基于某些信息在网络传输的一致性。

下次你遇到乱序、重传等情况时，也都可以运用我介绍的排查方法。当然，未必要照搬这里的每个步骤，但整体思想是类似的，希望通过我自己的经验总结，能给你一些启发。

思考题

这节课里，我介绍了使用裸序列号作为定位两侧同个报文的手段。那么要定位两侧的同个报文，除了这个方法，还有哪些方法呢？你可以从网络七层模型出发，给出自己的思考。

欢迎在留言区分享你的答案，我们一起讨论。如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友，我们下节课再见。

上一篇 04 | 挥手：Nginx日志报connection reset by peer是怎么回事？

精选留言 (3)

那时刻

2022-01-21

要定位两侧的同个报文，还可以依据消息内容，frame contains进行过滤


那时刻

2022-01-21

老师的思考题，我想到的是依据 源地址，源端口，目的地址，目的端口的角度进行过滤消息

作者回复: 恩这个可以定位到同个tcp流，也是很好的进展。不过要定位到报文级别，这个就不够了。你既然可以准确定义流，相信你也可以定义到报文级别，再想想？


includestdio.h

2022-01-21

老师留的思考题没啥头绪，我就大概按照我的理解总结下本篇的知识点吧 - -

排查原则：
“两端抓包”，听完“A”的抱怨，也该问问“B”的解释，这样才公平
...
展开

作者回复: 你的总结太好了，学习委员就是你了：）这个排查分析的思路虽然不包含具体的操作，但对我的排查工作的开展非常有指导意义，希望对你也有帮助