13 | 重传的再认识：没有任何丢包却也一直重传？

2022-02-18 杨胜辉

《网络排查案例课》 课程介绍 >

讲述：杨胜辉

时长 20:46 大小 19.02M

你好，我是胜辉。

在上节课，我带你深入探讨了 TCP 重传的知识点，包括超时重传和快速重传。想必你对于重传的现象和背后的原理，也已经有了不少的了解。那么现在，你可以来思考这样一种情况：用 Wireshark 打开一个抓包文件，你看到了满屏的 TCP Retransmission，第一感觉会是什么？

你应该会认为是掉包了，所以客户端重传了对吧？可能是网络路径上出了状况。

但实际上，网络状况是重传的一个重要因素，却不是唯一。另外一个因素也同样重要：**操作系统对 TCP 协议栈的实现**。

这是因为，TCP 等传输协议不是无根之木，它们必须依托于操作系统而存在，包括各种客户端、服务端、网络设备等等。就以重传为例，表面上看是由于网络状况而引发的，但其实真正操控重传行为自身的，还是操作系统，确切地说，是 TCP 通信两端的操作系统。

所以，在这节课里，我会给你再介绍一个十分特殊的案例，带你用一种全新的视角来审视 TCP 重传。通过这节课的学习，你将会对 TCP 的基本设计，特别是其中最复杂的知识点之一的重传部分，有更加深刻的理解。这样即使以后你在工作中遇到各种奇怪的 TCP 问题的时候，也不会再轻易被它们的表面所迷惑，而是能有更加准确的判断了。

eBay 的 HTTP 请求慢的问题是怎么解决的？

开头我们假设的那个场景，是一上来就直接分析 TCP 的重传问题，好像这个问题刚冒头，就是以“TCP 重传”的形式出现的一样。但在真实的生产环境当中，问题出现的时候就不会那么直接了，而是以应用层的某种形式，比如以“事务处理慢”这种形式出现的。

下面，我们看一个实际的案例。

应用层分析：应用为什么变慢了？

eBay 的应用大部分都是基于微服务进行设计和开发的。有一天，一个业务开发团队向我们基础架构团队报告了一个情况：从他们客户端集群向服务端（LB 上的 VIP）发送的请求，遇到了大量的 Response Timeout（返回超时）的报错。这里的 Timeout 是一个应用层的超时设置，如果客户端无法在 2 秒钟（2000ms）之内收到返回信息，就会抛出超时报错。

在我们内网，同数据中心的时延基本在 1ms 以内，跨数据中心的时延在 10ms 上下，都很快。这里设置的 2000ms 的超时，事实上大部分都是预留给了应用程序。这个应用的超时机制，跟 TCP 超时重传机制类似，应用也不想“干等”。

所以，我们首先采用了挨个测试的方法。因为整个路径是：

> **客户端 -> LB -> 服务器（就是 LB 后面的机器）**

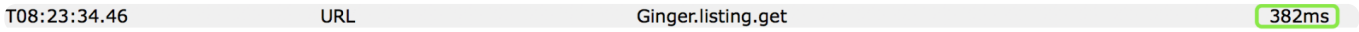
而报错是在客户端观察到的，那么我们可以对比两种路径：

> **客户端 -> LB -> 服务器**
>
> **客户端 -> 服务器（绕过 LB）**

看看在这两种情况下传输的请求都有什么不同。

客户端直接访问服务器的情况

我们发现，如果客户端绕过 LB VIP 去直接访问服务器，是正常的，没有超时的报错。服务器上的日志显示，很快收到了客户端发出的请求，花了三百多毫秒就处理完并回复了。比如这个服务器日志：

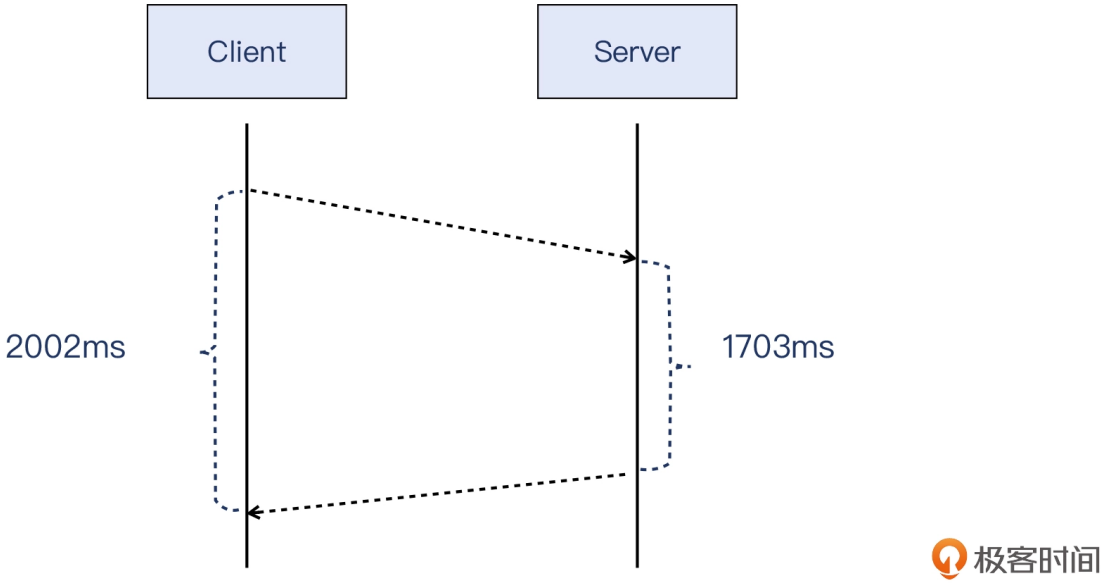


客户端访问 LB VIP 的情况

这种情况下，客户端会等待很长的时间才能拿到 HTTP 响应。日志也印证了这一点：服务器上的日志显示，这个请求的处理耗费了 1703 毫秒。如下图：

并且，客户端上的日志显示，同样是这个请求，在它看来，从发出请求给 LB 的 VIP，到收到 LB 的 VIP 返回的响应，一共消耗了 2002 毫秒。如下图：

我画了一张示意图，帮助你理解得更加清楚一些：



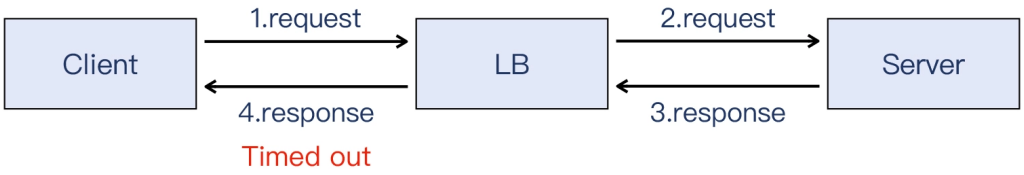
你也许会问：看起来不是服务器那头本身处理的耗时很长吗？为什么不查查服务器上应用代码的 Bug？

我们也一度有这个怀疑，服务器上运行的是 Java 代码，会不会是 GC 造成的影响呢？所以我们也去查了当时 JVM 的运行情况，发现这段时间内并没有 GC 事件。因此，这个可能性也被排除。并且我们定位到，服务器的耗时主要是花费在了 read() 调用上，即读取网络 I/O 上面，所以还是需要回到网络排查的方向上来。

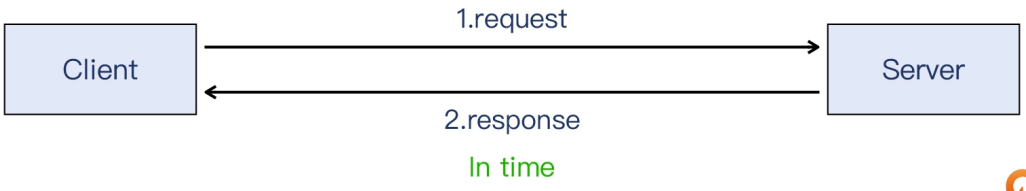
> 补充：Java 有相关的分析工具来定位耗时所在，或者用 strace 也可以定位系统调用的性能情况。

我们用一个示意图来概括这两种场景下的区别：

实际的问题场景



测试的正常场景



极客时间

由此，我们可以初步判定：问题出在 LB，或者 LB 前后的网络环节。

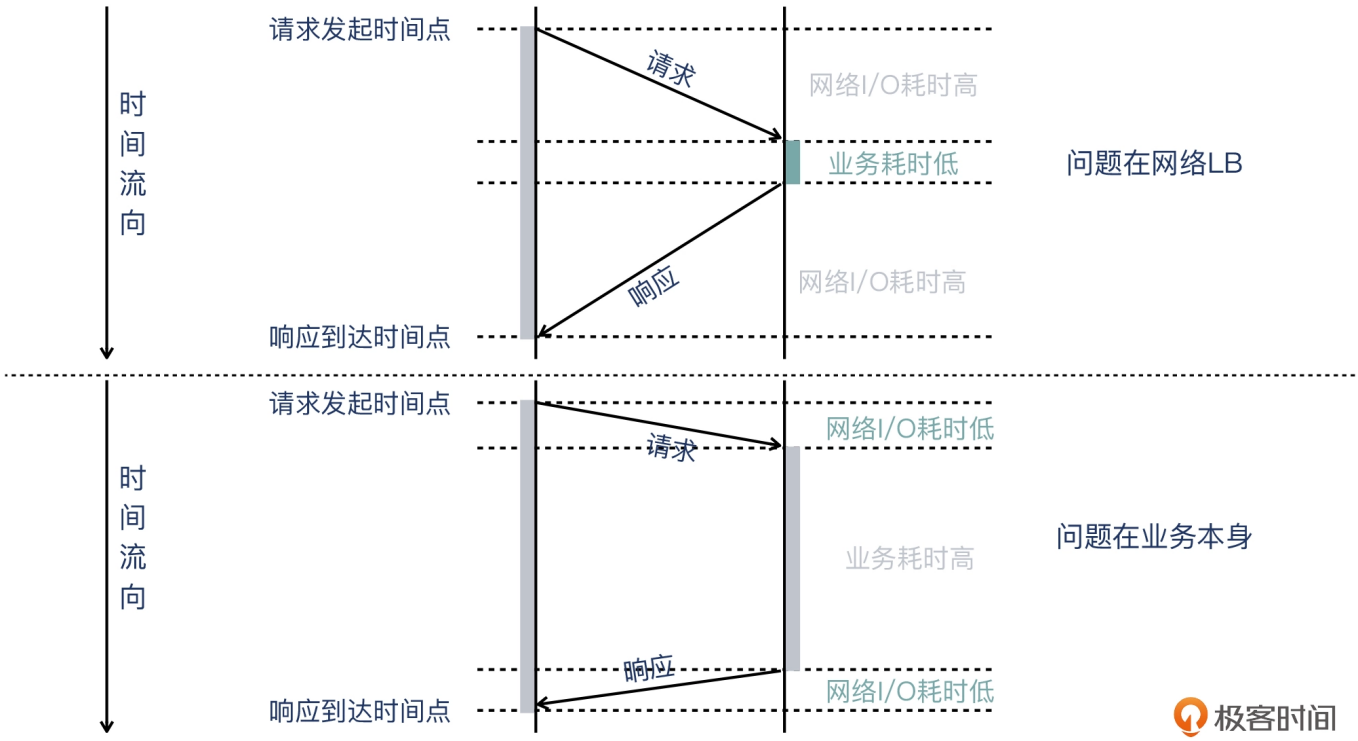
这里也是我想分享给你的一个小的排查原则：**针对客户端看到超时或者响应慢的这类问题，最好也检查下服务器本身花费的时间，两者对比，就能找到问题的方向了。**

我给你把整个思路用伪代码的形式组织如下：

复制代码

```
if（服务器耗时约等于客户端耗时）{
   检查服务器耗时分布
   if（服务器耗时在网络I/O）{
        检查中间网络或者LB
   } else {
        检查服务器应用程序或操作系统
   }
} else if（服务器耗时远小于客户端耗时）{
   检查中间网络或者LB
}
```

我也画了一个示意图，供你参考：



那么下面，我们就可以把排查重点，转到 **LB 和网络层面**上来。

抓住排查重点：LB 和网络层

首先，我们在 LB 进行了抓包，用 Wireshark 打开抓包文件，一开始看到的是一帆风顺，全绿：

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	client	LB-VIP	TCP	1514	59691 → 80 [ACK] Seq=1 Ack=1 Win=63712 Len=1460 [TCP segment of a reassembled PDU]
2	0.000122	client	LB-VIP	TCP	1002	59691 → 80 [PSH, ACK] Seq=1461 Ack=1 Win=63712 Len=948 [TCP segment of a reassembled PDU]
3	0.000171	client	LB-VIP	TCP	1514	59691 → 80 [ACK] Seq=2409 Ack=1 Win=63712 Len=1460 [TCP segment of a reassembled PDU]
4	0.000121	client	LB-VIP	TCP	1514	59691 → 80 [ACK] Seq=3869 Ack=1 Win=63712 Len=1460 [TCP segment of a reassembled PDU]
5	0.000005	client	LB-VIP	TCP	1514	59691 → 80 [ACK] Seq=5329 Ack=1 Win=63712 Len=1460 [TCP segment of a reassembled PDU]
6	0.000007	client	LB-VIP	TCP	1514	59691 → 80 [ACK] Seq=6789 Ack=1 Win=63712 Len=1460 [TCP segment of a reassembled PDU]
7	0.000007	client	LB-VIP	TCP	1514	59691 → 80 [ACK] Seq=8249 Ack=1 Win=63712 Len=1460 [TCP segment of a reassembled PDU]
8	0.000003	client	LB-VIP	TCP	1514	59691 → 80 [ACK] Seq=9709 Ack=1 Win=63712 Len=1460 [TCP segment of a reassembled PDU]
9	0.000002	client	LB-VIP	TCP	1514	59691 → 80 [ACK] Seq=11169 Ack=1 Win=63712 Len=1460 [TCP segment of a reassembled PDU]

翻了几页，突然画风一变，全红：

No.	Time	Source	Destination	Protocol	Length	Info
68	0.018573	LB-VIP	client	TCP	60	[TCP Dup ACK 53#1] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0
69	0.000001	LB-VIP	client	TCP	60	[TCP Dup ACK 53#2] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0
70	0.000000	LB-VIP	client	TCP	60	[TCP Dup ACK 53#3] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0
71	0.000000	LB-VIP	client	TCP	60	[TCP Dup ACK 53#4] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0
72	0.000000	LB-VIP	client	TCP	60	[TCP Dup ACK 53#5] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0
73	0.000001	LB-VIP	client	TCP	60	[TCP Dup ACK 53#6] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0
74	0.000000	LB-VIP	client	TCP	60	[TCP Dup ACK 53#7] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0
75	0.000000	LB-VIP	client	TCP	60	[TCP Dup ACK 53#8] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0
76	0.000000	LB-VIP	client	TCP	60	[TCP Dup ACK 53#9] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0
77	0.000000	LB-VIP	client	TCP	60	[TCP Dup ACK 53#10] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0
78	0.000001	LB-VIP	client	TCP	60	[TCP Dup ACK 53#11] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0
79	0.000000	LB-VIP	client	TCP	60	[TCP Dup ACK 53#12] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0
80	0.000000	LB-VIP	client	TCP	60	[TCP Dup ACK 53#13] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0

> 补充：抓包示例文件已经上传至 Gitee，建议你结合专栏内容和抓包示例文件一起学习，效果更好。

看到这个景象，你是否也会这样判断：“这肯定是有丢包了，所以接收方一直在回复 DupAck！赶快去查网络设备的问题。”

DupAck 多半是丢包引起的，但这次不是。为什么呢？我们先来看一下抓包文件展现出来的全貌。

第一阶段：连接建立，正常。

第二阶段：客户端开始发送数据包给 LB，正常。

第三阶段：客户端连续发送了约 70KB 数据（其中大部分数据还未被确认）。

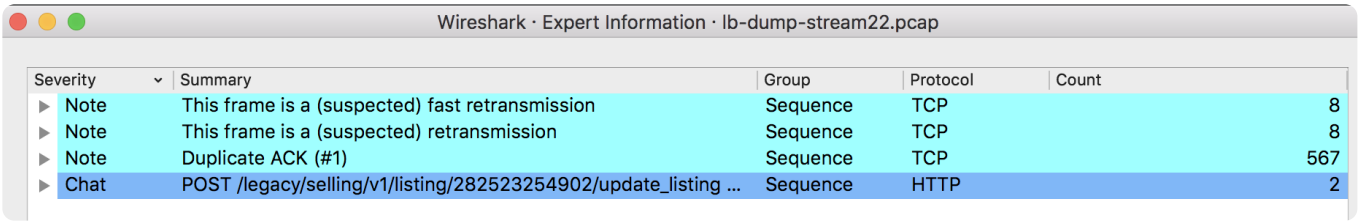
第四阶段：LB 发送 ACK 包确认收到了前面约 27KB 的数据。

第五阶段：客户端继续发送 70~91KB 的数据，目前为止也是正常的。

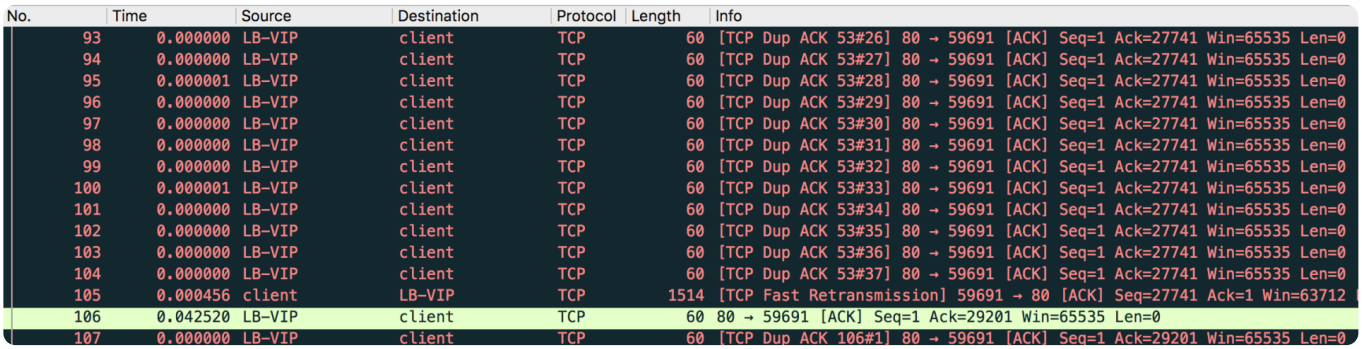
第六阶段：在 27KB 之后，LB 连续发送数十个 DupAck，然后客户端发送一次 TCP fast retransmission；这样的情况持续到客户端把所有应该发的数据包都发完。

那么，我们可以在 Wireshark 里打开 Expert Information 看一下汇总信息。

> 补充：关于 Expert Information 的解读，我在第 5 讲有介绍过，如果你感觉现在印象有点模糊的话，可以回头去温习一下。



从图上看，快速重传有 8 个，DupAck 有 567 个，平均每个快速重传对应了约 70 个 DupAck。这里，我们来看看其中一段连续的 DupAck 和随后的一个快速重传：



我给你解读一下：在 LB 给客户端发送了数十个 DupAck 之后（比如截图里包号 93 到 104），客户端发送了一次快速重传（包号 105），然后 LB 回复 ACK（包号 106，针对包号 105），接着 LB 继续新一轮的数十个 DupAck（从包号 107 开始），循环往复。

其实这里已经说明了这个案例的特殊性，也就是有“**规律性现象**”。如果是网络设备问题导致丢包，那么丢包会是随机现象，不太可能像这样有规律（70 个 DupAck 加一个快速重传，不断循环）。而**往往规律的背后，一般潜藏着某种未知的机制**。

当然，大量的 DupAck 和重传，确实跟应用层我们看到的严重的延迟现象对上了。也至少能回答“**应用为什么变慢**”这个问题了。当然，探究根因的话，接下来就是要回答“**为什么有重传**”这个问题。

网络排查推进：为什么客户端出现了重传？

在上节课，我们学习过两种重传方式，分别是快速重传和超时重传，而这次的重传呢，Wireshark 已经提醒我们，属于快速重传。那为什么会有这种快速重传呢？

这里我们先来看一下相关数据包的具体细节。

首先，我们看一下在 LB 回复大量 DupAck 之前，客户端的发送情况。

65	0.000003	client	LB-VIP	TCP	1514 59691 → 80 [ACK] Seq=88385 Ack=1 Win=63712 Len=1460 [TCP segment of a reassembled PDU]
66	0.000002	client	LB-VIP	TCP	1514 59691 → 80 [ACK] Seq=89845 Ack=1 Win=63712 Len=1460 [TCP segment of a reassembled PDU]
67	0.000001	client	LB-VIP	TCP	1514 59691 → 80 [ACK] Seq=91305 Ack=1 Win=63712 Len=1460 [TCP segment of a reassembled PDU]
68	0.018573	LB-VIP	client	TCP	60 [TCP Dup ACK 53#1] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0
69	0.000001	LB-VIP	client	TCP	60 [TCP Dup ACK 53#2] 80 → 59691 [ACK] Seq=1 Ack=27741 Win=65535 Len=0

可以看到，在第一个 LB DupAck 包之前，客户端发送的最后一个数据包是包号 67，它的信息是：

> 序列号为 91305，表明从握手开始有 91304（减去握手阶段的 1）字节的 TCP 载荷，从客户端发出了；
>
> 确认号为 1，因为 LB 还没回复 HTTP 响应，也就是没有握手以外的更多数据，所以客户端还是保持握手阶段的确认号 1。

然后我们再来看一下第一个 LB DupAck 包，其包号为 68：

序列号为 1，因为 LB 还没回复应用层的 HTTP 响应，所以还是保持握手阶段的序列号 1；

确认号为 27741，表示 LB 收到了 27740（减去握手阶段的 1）字节的数据，而第 27741 字节之后的数据并没有收到。

要知道，TCP 协议规定：**接收方回复的 ACK 包的确认号 = 发送方数据包的序列号 +TCP 载荷字节数**。如果接收方回复了 DupAck，假设这个 DupAck 的确认号为 n，那么其含义是：我只收到发送方给我的序列号为 n 之前的数据包，而序列号为 n（及其之后）的数据包，我都没有确认。

所以，LB 就通过 DupAck 包向客户端宣告：“我这边只确认收到序列号 27741 之前的数据包。”

而这里出现几十次 DupAck 的原因是，一旦 LB 认为某个数据包我没有收到（此处是序列号为 27741 的数据包），那么数据就“断档”了，之后客户端送过来的每个数据包，LB 都无法 ACK 这些数据包的序列号 +TCP 载荷字节数。所以虽然 ACK 包还是要发，但确认号却只能“停留”在丢包处的确认号，并且这样重复的 ACK 会有很多个。

为了便于理解，我把这个过程换一种方式给你展示一遍。除了 27741 这个号以外，其他序列号是为方便举例而编出来的：

Client	LB
seq 27741 (这个包掉了)	
	Ack 27741
seq 30000	
	DupAck 27741
seq 40000	
	DupAck 27741

上面是客户端来一个报文，LB 回复一次 DupAck。当然也可能像下面这样，连续来多个报文，LB 回复连续的多个 DupAck：

Client	LB
seq 27741 (这个包掉了)	
	Ack 27741
seq 30000	
seq 40000	
	DupAck 27741
	DupAck 27741

好了，现在情况就比较清楚了，虽然“丢包”的根因还没找到，但整个排查工作的脉络已经相对清晰了，即：**某处丢包 -> TCP 重传 -> TCP 传输速度下降 -> 应用层超时报错。**

看起来，我们离成功只剩一步了，也就是，在抓包文件中**找到那个丢失的序列号为 27741 的数据包**！

只要证明这个数据包确实是在网络上丢失的，那么我们就去修复网络。TCP 不丢包了不重传了，速度就上来了，应用就不超时了。逻辑圆满自洽，感觉胜利已经在向我们招手了，是不是？

可是峰回路转。我们去翻前面数据包的时候，发现根本就**没有那个“序列号为 27741”的数据包**！

```
20   0.000002 client     LB-VIP      TCP      1514 59691 → 80 [ACK] Seq=27229 Ack=1 Win=63712 Len=1460
21   0.000017 client     LB-VIP      TCP      1514 59691 → 80 [ACK] Seq=28689 Ack=1 Win=63712 Len=1460
```

上图是最接近 27741 序列号的附近的数据包，有序列号 27229 的包，也有序列号 28689 的包，但就是没有位于这两个数中间的 27741 的包。

这个时候，恍惚中有点感觉在看悬疑小说：一桩案件的元凶被查出是某某某，结果发现某某某这个人压根不存在。

你如果有跑步的爱好，应该会知道：跑步过程中会有一个极限区，此时我们的心肺会遇到很大的压力，这种难受的感觉很容易让人放弃。但是，如果继续坚持挺过这个极限区，身

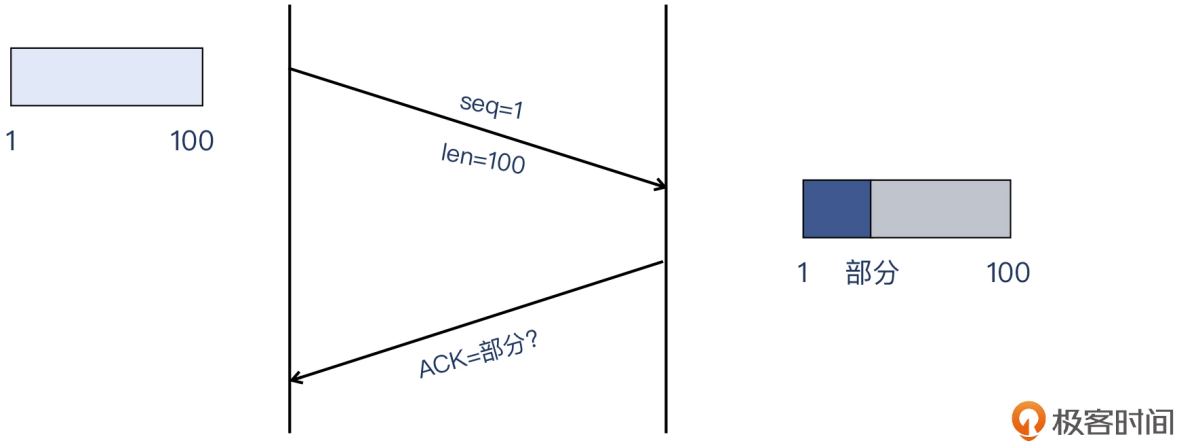
体就能提升到一个新的平台上继续平衡运转。进而，我们就可以继续快乐地跑下去了。

显然，我们的排查进入了“极限区”了。止步还是进步，就在一念之间。

TCP 的本质：再次思考什么是确认号？

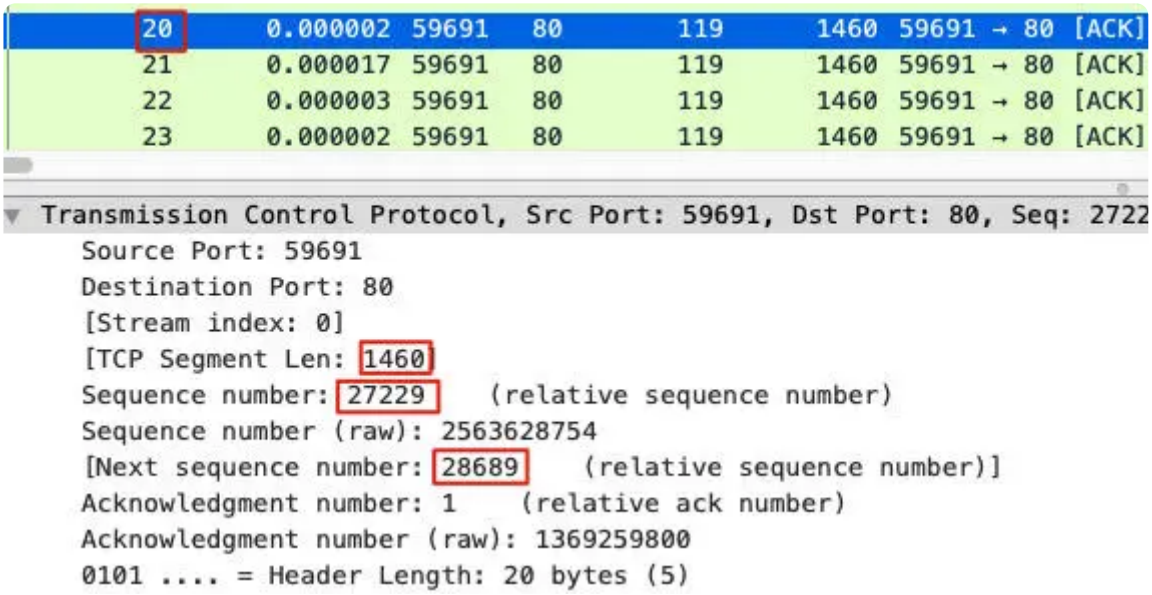
那么，这个序列号 27741 的包是消失了吗？还是说它就在那里，只是我们忽视了它的存在？

TCP 序列号、Payload（载荷）、TCP 确认号，一般情况下就是一个 A+B=C 的关系。但是，**确认号必须是序列号 + 全部载荷吗？它可以是序列号 + 部分载荷吗？**

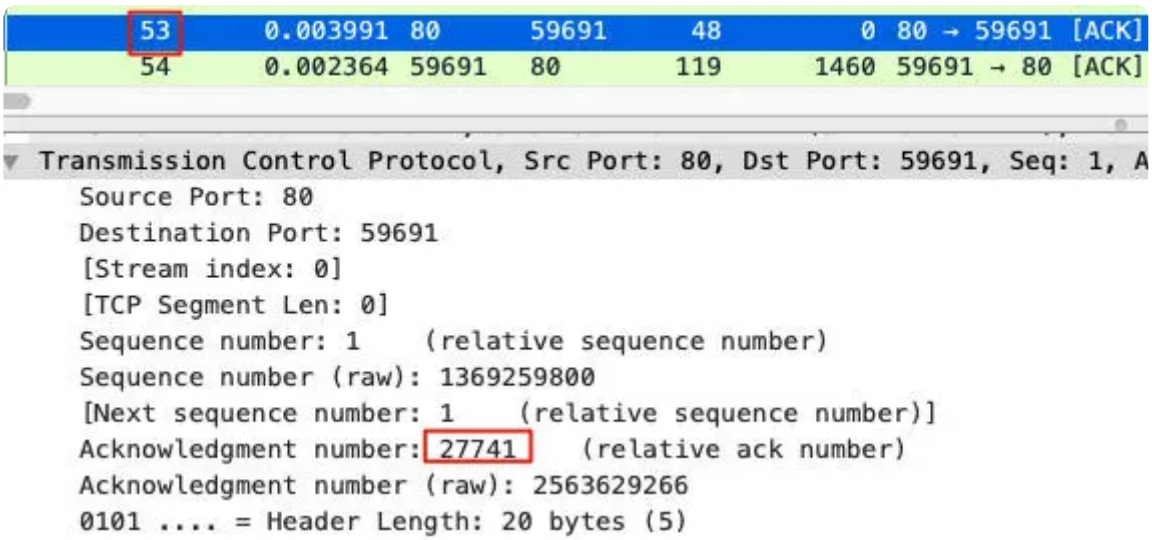


举个例子，如果我从网上购买了一套衣服（上衣 + 裤子），我也收到了全套。但我觉得裤子尺码不对，上衣还挺合身，我可以只确认我收到了上衣（当然裤子还是要退回的），让卖家重新发裤子给我吗？这是可以的。

如果 TCP 也可以这样呢？那么，**“寻找序列号为 27741 的包”也许就是个伪命题，这个“包”实际上并不是独立的一个包，它只是某一个 TCP 包的一部分（前半部分）。**我们来看一下包号 20（也就是前面找 27741 的时候，关注的两个报文之一）的详情：

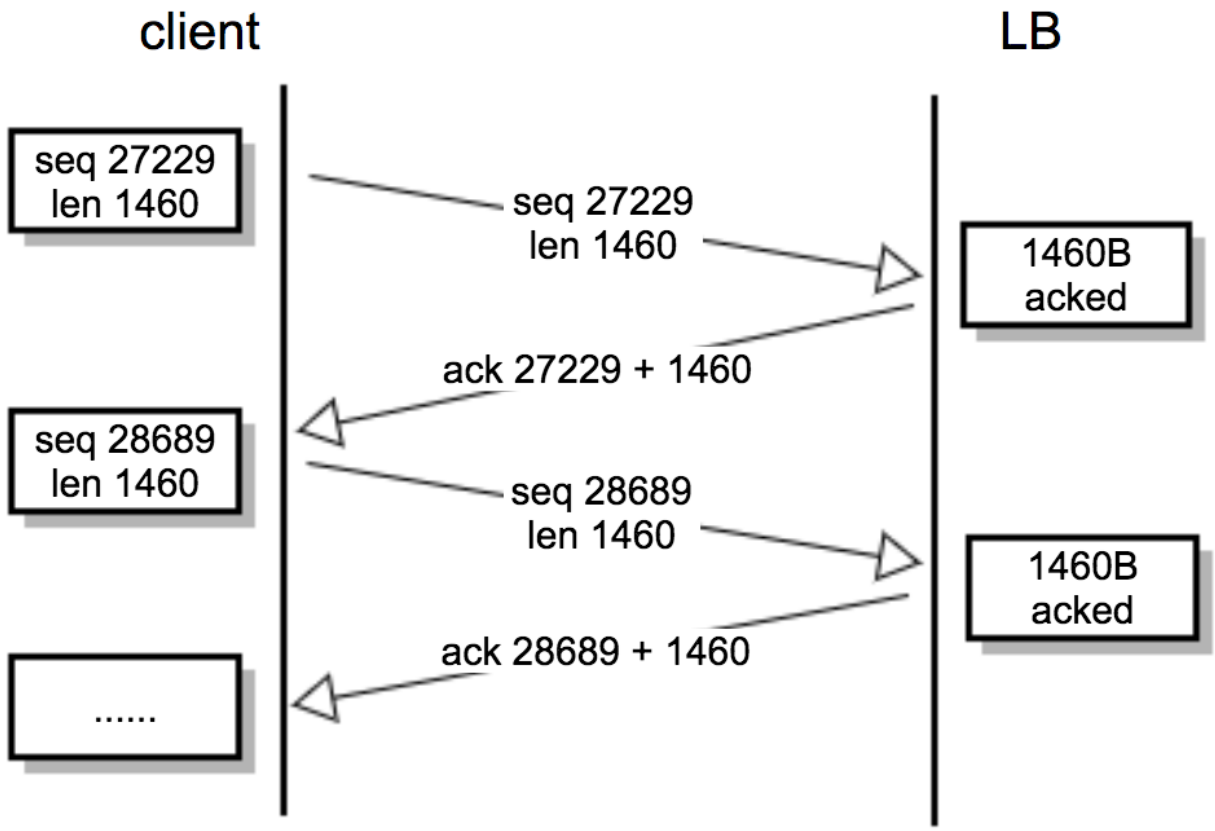


这个包是从客户端发给 LB 的，它的序列号为 27229，载荷为 1460 字节。Wireshark 也告诉我们，客户端将要发的下一个包的序列号会是 28689。然而，LB 回复的 ACK（后续同样的 ACK 就是 DupAck）却是这样：



我们再看那个最为可疑的数字 27741。显然，27741=27229+512。到这里看清楚了吗？这次 LB 确认的是一件“上衣”（512 字节），而余下的“裤子”（另外的 1460-512=948 字节），LB 并没有确认。没有被确认的数据，在客户端看来，是需要重新发送的。

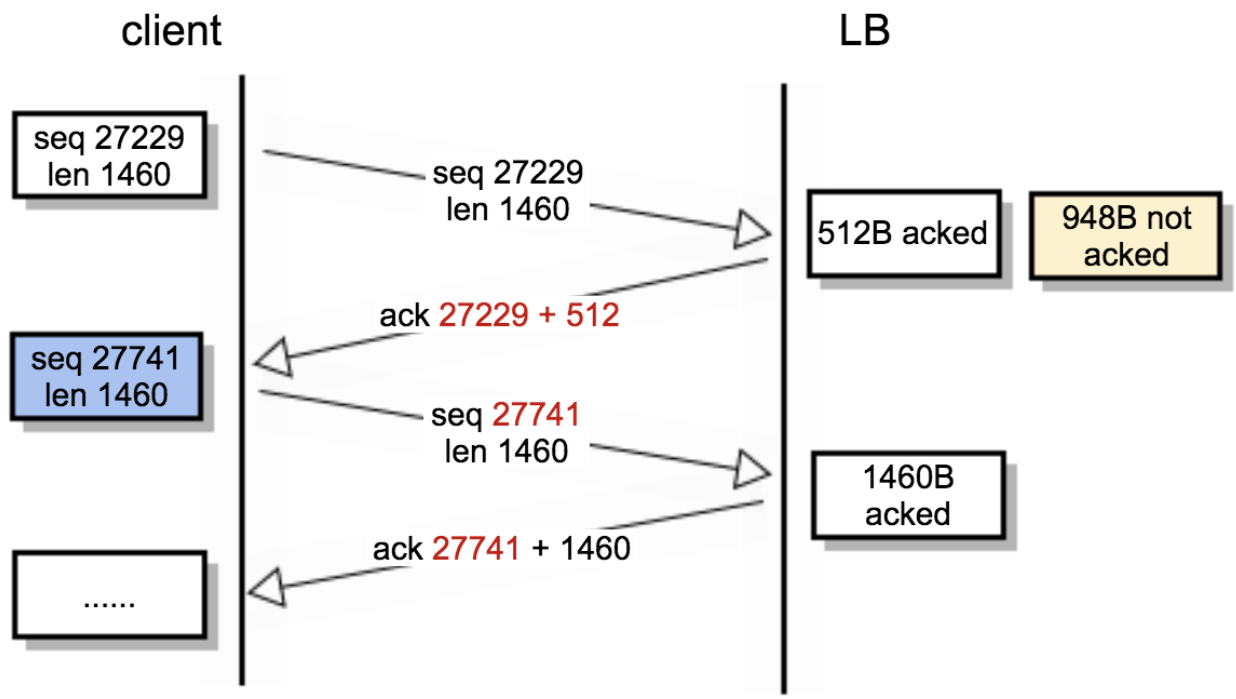
我们先看看正常情况（即每次确认 1460 字节 ）下的数据包交换过程：



然后再来看一下这次异常交换的过程。

> 客户端：我发给你从 27229 开始的 1460 字节，下一次你懂的，我将要发的是 28689 开始的数据。
>
> LB： 我也不知道最近怎么搞的，记性不好，你这次给我这些数据，我好像只认得前 512 字节，其他的我认不出来了，先确认这 512 字节吧。
>
> 客户端：怎么回事？只确认前 512 字节？麻烦了，我为了保证这次发送的 TCP 载荷依然能用足一个 MSS 即 1460 字节，必须**把前一个包的后 948 字节和下一个包的前 512 字节，组合在一起，变成一个新的 1460 字节的包**，再发送给你。不过还好，所有未被确认的数据还都在我的发送缓存（send buffer）里面，没有丢失。不过原先计算好的安排都要改掉了，我的 CPU 开销很大啊！

这次异常通信的过程如下图所示：

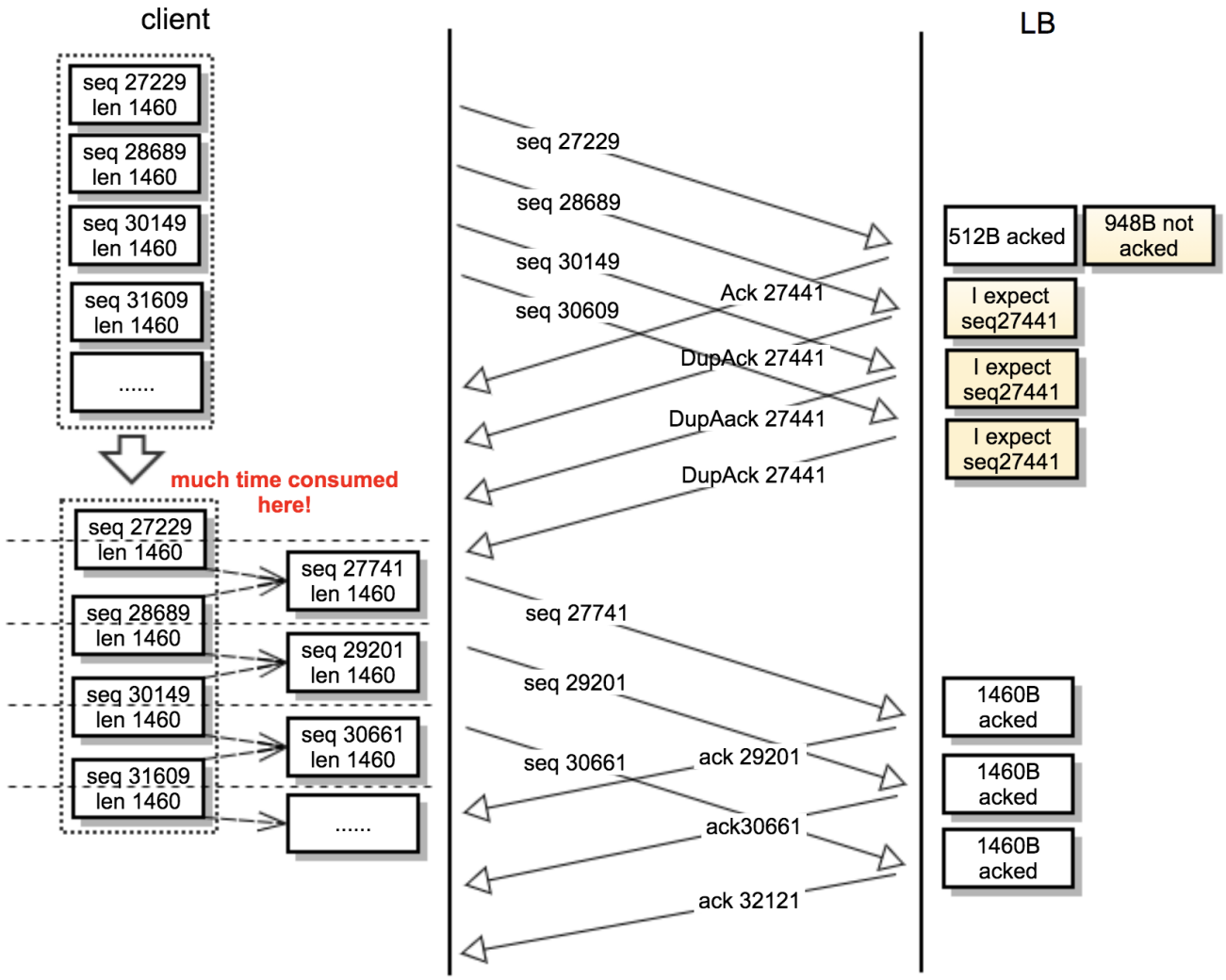


这里，我再提醒你一个关键点：**确认号本身代表字节数，所以它是字节级别的，而不是报文级别**。也就是说，确认号是精确到某个字节的，而不是某个报文。

说到这里，我们再回顾下前面提到过的现象：平均每个快速重传对应了约 70 个 DupAck，而每次重传都需要客户端把发送缓冲区里面打包好的数据包，挨个拆开，重组成 LB 想要的样子（512 字节的位移的关系）。

想必，现在你已经清楚为什么应用会超时（超过 2s）了：因为时间都花费在了各种包的分拆、重组上面了。光是客户端想成功发完一个 POST 请求，都花费了远远超出预期的时长。就如同一辆车频繁熄火，还怎么可能高速行驶呢？

我画了一张更详细的图，来帮助你理解这个复杂的过程：



客户端发送 HTTP POST 请求，在 TCP 层面体现为一系列数据包（30KB 以内）给 LB，LB 转发给服务器，在 30KB 之前一切正常。客户端应用程序的 Timeout 计时器，也从 POST 请求发送的那一刻开始计时。

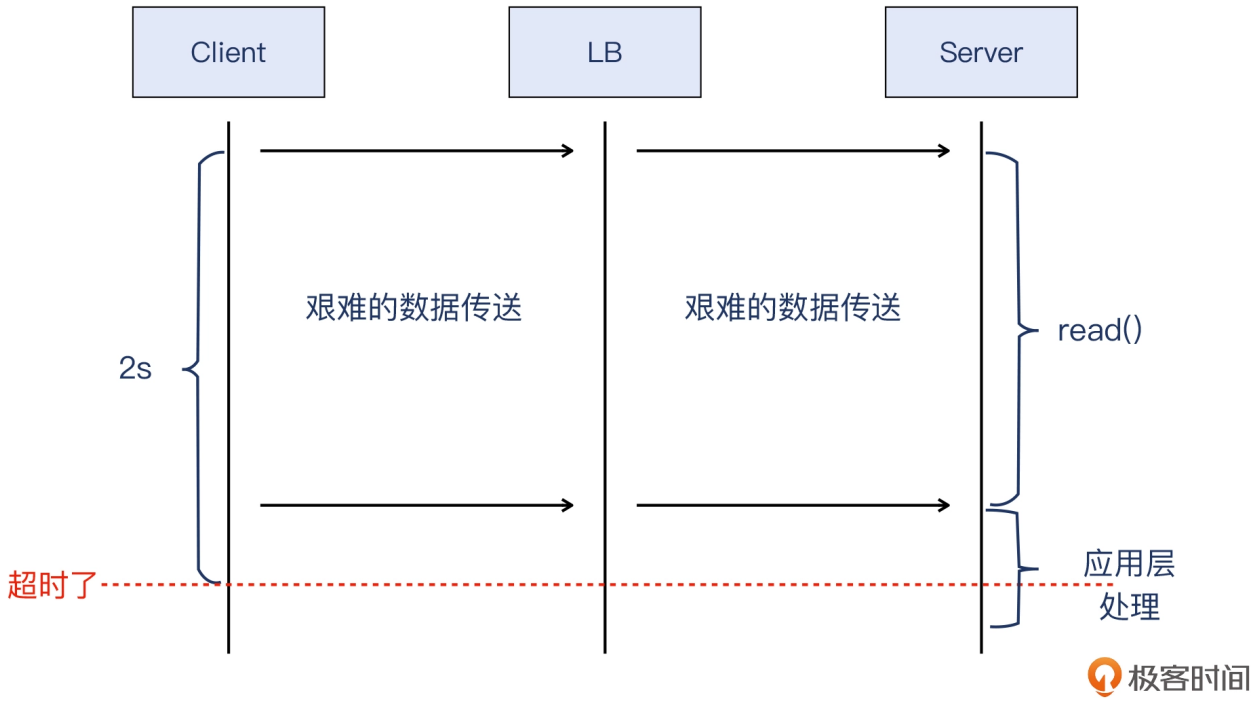
大约在客户端发送数据到 30KB 左右时，LB 回复的 ACK 包确认的数据，不再是数据包分界点的字节数，而是**位于中间的某个字节数**（这个行为比较罕见）。

客户端累积收到 3 个这样的 DupAck（在这个案例里达到 70 多个），认为该包丢失。加上这个包的特殊性（是之前某个完整包的一部分），客户端会从缓冲区找出对应的字节数，拼凑上后续包的数据，组装成一个新的 MSS 大小（1460 字节）的数据包，并且在此处消耗了可观的时间。

LB 继续发送类似的“中间确认”包，客户端继续进行“拆包、重组”的操作，此处**持续消耗客户端的时间**。

服务器的时间都花在 read() 调用上（因为数据还在重传和重组过程中），无法及时收取完整的 POST 请求并计算处理，此时已经无法在客户端预定的 Timeout 时限内完成任务，于是客户端报错。

为了更加方便你的理解，我把这个过程概括成了下面这张图：



真相大白了！凭借这些详细的分析和充足的证据，我们说服了 LB 厂商并确认这是一个 Bug，它容易在请求尺寸比较大的情况下被触发。比如在这个案例里，一个 POST 请求平均大小在 100KB，也就触发了这个 Bug。

实际上，在 Bug 修复之前，我们通过扩大 TCP receive buffer size，使得缓冲区足够大（你 HTTP POST 请求大，我缓冲区更大），也做到了对 Bug 的规避。

小结

今天介绍的确实是一个比较罕见的案例，也是我处理过的众多网络排查案例中，遇到的仅有的一次“确认号在中间位置”的情况。那么从协议规范来说，这样“确认中间位置”的做法，到底是否违规呢？

我们看一下 TCP 协议的第一版规范RFC793，看看它对确认号的要求是什么：

复制代码

```
1 if the ACK bit is on
2 ......
3         ESTABLISHED STATE
4
5           If SND.UNA < SEG.ACK =< SND.NXT then, set SND.UNA <- SEG.ACK.
```

其中几个缩写的含义如下：

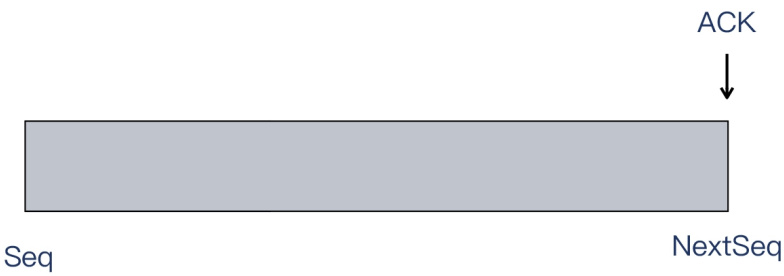
```
1 SND.UNA - send unacknowledged #这是指已发送的但未被确认的TCP段的位置
2 SND.NXT - send next           #这是已经发送的TCP段的下个序列号
```

复制代码

> “下个序列号”这个知识点在第 10 讲介绍过，你可以回头复习一下。

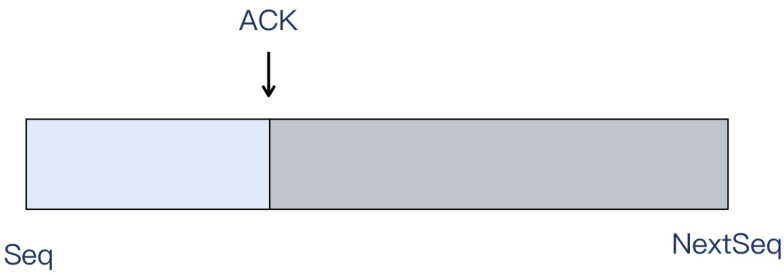
TCP 应该接受 **SND.UNA < SEG.ACK =< SND.NXT** 这样的情形，也就是收到的报文的确认号，应该大于已经被确认的数据的位置，并且小于等于（要发送的）下个序列号。

一般来说，我们看到的大部分是“确认号等于下个序列号”的情况，如下图：



极客时间

但是从这个案例的情况来看，就是**确认号是在中间位置**。这虽然很少见，但也不违规，也可以被操作系统接纳并处理。只不过，引起的开销有点过大了。



极客时间

除了上面的知识点以外，我也建议你务必关注整个排查过程带来的启发：

网络排查过程中要仔细核对各种事实和数据，避免仅根据表面现象轻易下结论。比如，在这个案例里，很多的 TCP 重传很容易让我们把关注点错引到网络状况上面去。所以只有仔细核对这些数据，发现其中的问题，才不会被自己的思维惯性所误导。

对于各种重传的现象和成因应该有充分的了解，这样对排查方向的确定有很大的帮助。特别是重传的两个大类，即快速重传和超时重传，它们的特征和应对策略，你最好熟记于心，这样等你遇到类似情况时，很快可以对症下药，提高解决问题的效率。

基于前面两点做细致踏实的分析，即使得出的结论比较意外，也应该保持实事求是的态度去看待和验证。在这次案例中，TCP 确认号没有像常规的那样 ACK=RCV.NXT，这也是出乎意料的事情。正是因为我们充分尊重这样的事实，并进行推理，才能突破既有的思维，找到了真正的原因。

对于“超时、处理慢”这类问题，建议你对比客户端和服务端的耗时，这有利于你找到正确的排查方向。在这个案例中，我们比较了两端的耗时，发现两者接近；然后，通过在服务器上做系统排查，发现时间主要花费在 read() 上，这就说明，问题很可能出在网络或者 LB 上。课程中我给你整理了一段伪代码，梳理了这种排查思路，你可以拿来参考。

最后，**对于排查期间发现的规律性的现象，可以重点关注。**规律性的背后藏着的东西，跟问题的根因，多半有着密切的联系。所以这种规律性问题，也许正是我们排查的突破口。

思考题

最后还是给你留两道思考题：

如果接收端收到一个确认包，其确认号为 200，而当前的未被确认的位置在 500，那么接收端会怎么处理这个看起来“迟到并且重复”的确认包呢？

你有没有遇到过这种“确认号在中间位置”的情况？当时有没有引起什么问题呢？

欢迎在留言区分享你的答案，也欢迎你把今天的内容分享给更多的朋友。

附录

抓包示例文件：https://gitee.com/steelvictor/network-analysis/tree/master/13

上一篇 12 | 重传的认识：重传到底是怎么回事？

下一篇 14 | 安全：用Wireshark把DDoS攻击照出原形

精选留言 (4)

写留言

taochao_zs
2022-02-18

1 根据滑动窗口定义：左边界不会左移只会右移（推进），收到左边已确认的重复确认包会直接丢弃。
2 老师说的问题还没碰到过，以后碰到再补上；）

作者回复: 正确，这个ack报文会被忽略
以后遇到类似问题心里有底了：）

2

MeowTvづ
2022-02-18

这一讲的例子也说明，两端没有开启SACK技术：）

作者回复: 哈哈，确实如此~

共 2 条评论 >

1

Realm
2022-02-18

1 直接丢掉，服务端说，别把我当傻子，我这里记着呢；2 没有遇到，感谢老师的“惊艳”分享，很奇葩，这种分组好的，重新拆开，又组装成新的分组，对性能影响巨大.

作者回复: 正确！：）

可以参考内核的这几行代码对老旧ack的处理，很直白：

```
/* If the ack is older than previous acks
 * then we can probably ignore it.
 */
if (before(ack, prior_snd_una))
    goto old_ack;
```

那时刻

2022-02-18

根据rfc793文档，for ESTABLISHED STATE，If the ACK is a duplicate (SEG.ACK < SND.UNA), it can be ignored. 在链接状态的时候，如果收到ack number小于未被ack的number，这个ack包会被忽略。我粗略看了下文档，其它状态下，貌似没有说明，不知是否有遗漏。

老师提到通过扩大 TCP receive buffer size，使得缓冲区足够大，做到了对 Bug 的规避。...

展开

作者回复: 是的，就是这段，你找的很对。另外调整接受缓冲区，对这个设备来说是配置一个profile，跟linux还是不同的，设备的tcp是自己维护的并不是内核，所以方式也不同。如果是linux，就是你说的配置了~