10 | 窗口：TCP Window Full会影响传输效率吗？

2022-02-11 杨胜辉

《网络排查案例课》 课程介绍 >

讲述：杨胜辉
时长 26:40 大小 24.43M

你好，我是胜辉。

有时候，不少知识点在过段时间重新回看的时候，又会有新的体会和发现。比如在第 8 讲里，我们回顾了一个 MTU 造成传输失败的案例，虽然整个排查过程的步骤不算很多，但也算是 TCP 传输问题的一个缩影了。尤其是其中那个失败的 TCP 流中的一些现象，比如客户端发出的重复确认（DupAck），还有服务端启动的超时重传，都值得我们继续深挖，所以我会在后续的课程里继续这个话题。

然后在上节课里，我们还探讨了传输速度的相关知识，也初步学习了窗口的概念。最后，我们终于推导出了 TCP 传输的核心公式：速度 = 窗口 / 往返时间。这个公式，对于我们理解传输本质和排查传输问题，都有很强的指导意义。

然而，如果你足够细心的话，其实可能会对上节课里的细节有一些疑问，比如：既然接收窗口满了，那为什么当时没有看到 TCP Window Full 这种提示呢？

其实，我这边也有不少内容按住没有展开，包括核心公式的理解，我们在这节课里将有一个新的认识。另外，我也将带你继续挖掘窗口这个细分领域，这样你以后遇到跟窗口相关的问题，就知道如何破解了。

案例：TCP Window Full 是导致异地拷贝速度低的原因吗？

也是在公有云服务的时候，有个客户有这么一个需求，就是要把文件从北京机房拷贝到上海机房。但是他们发现传输速度比较慢，就做了抓包。在查看抓包文件的时候，发现 Wireshark 有很多 **TCP Window Full** 这样的提示，不明白这些是否跟速度慢有关系，于是找我们来协助分析。

解读 Expert Information

我们先要了解一下抓包文件的整体状况。怎么看呢？当然是看 Expert Information 了：

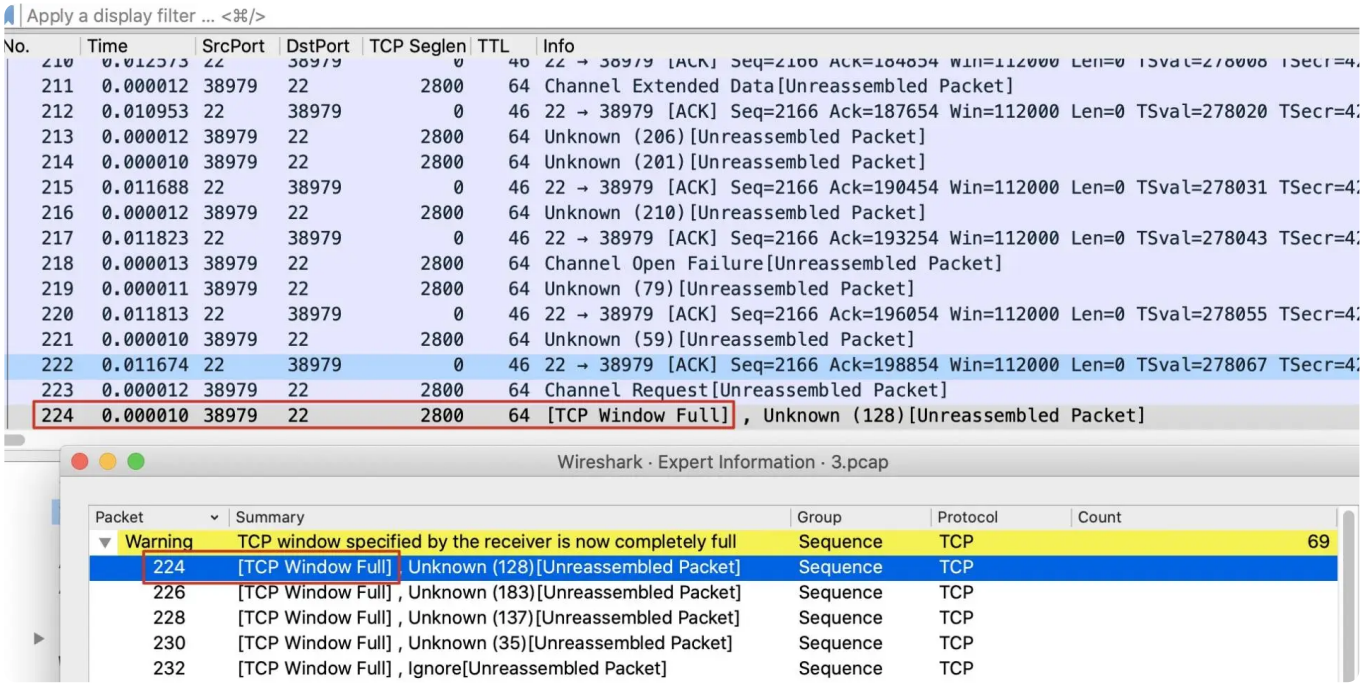
Severity	Summary	Group	Protocol	Count
Warning	TCP window specified by the receiver is now completely full	Sequence	TCP	69
Note	This frame is a (suspected) spurious retransmission	Sequence	TCP	2
Note	This frame is a (suspected) retransmission	Sequence	TCP	2
Chat	Connection establish acknowledge (SYN+ACK): server port 22	Sequence	TCP	1
Chat	Connection establish request (SYN): server port 22	Sequence	TCP	1

它确实提醒我们，有 69 个 Warning 级别报文，它们的问题是 TCP Window specified by the receiver is now completely Full。在展开进一步排查之前，我们先对这个信息做一下解读。

> TCP Window：在上节课里，我介绍了 TCP 的三种窗口，分别是发送窗口、接收窗口、拥塞窗口。那么这里说的是哪个窗口呢？一般说到 TCP Window，**如果没有特别指明，就是指接收窗口**。
>
> specified by the receiver：这也很明确，这个窗口是接收方的，其实就是佐证了这个窗口就是接收窗口。
>
> is now completely Full：窗口满了，这又怎么理解呢？你还记得在上节课里学过的在途数据，也就是 Bytes in flight 吗？**当在途数据的大小等于接收窗口的大小时，这个窗口就是“满了”**。

好了，这个信息解读完毕，一句话说就是：发生了 69 次在途数据等于接收窗口的情况。

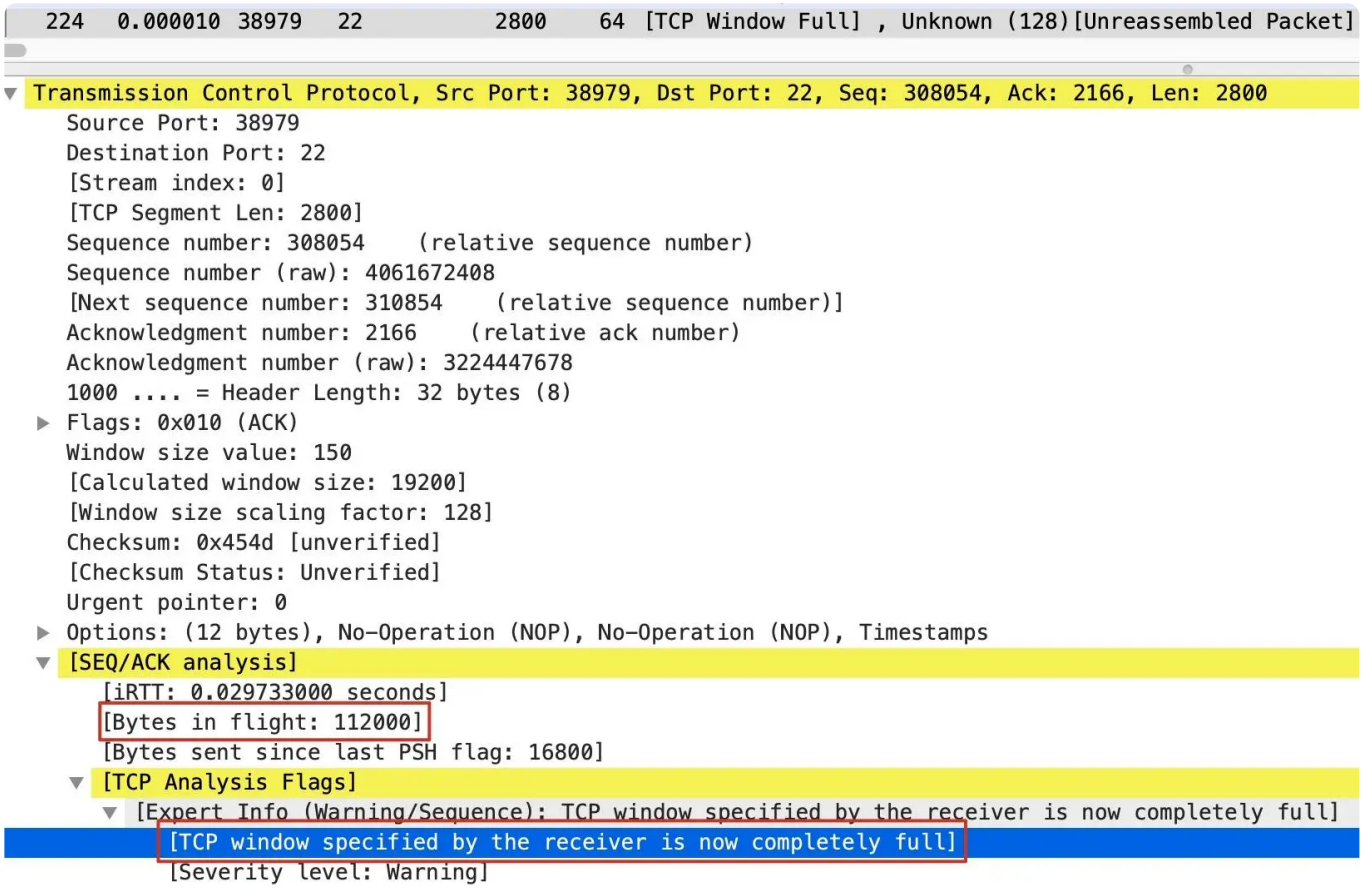
接下来我们看看 TCP Window Full 具体是个什么样子。比如我们选中 224 号报文，主界面也自动定位到了这个报文：



我们可以看到，除了 224 报文，也确实有很多其他报文也报了 TCP Window Full 的警告信息。

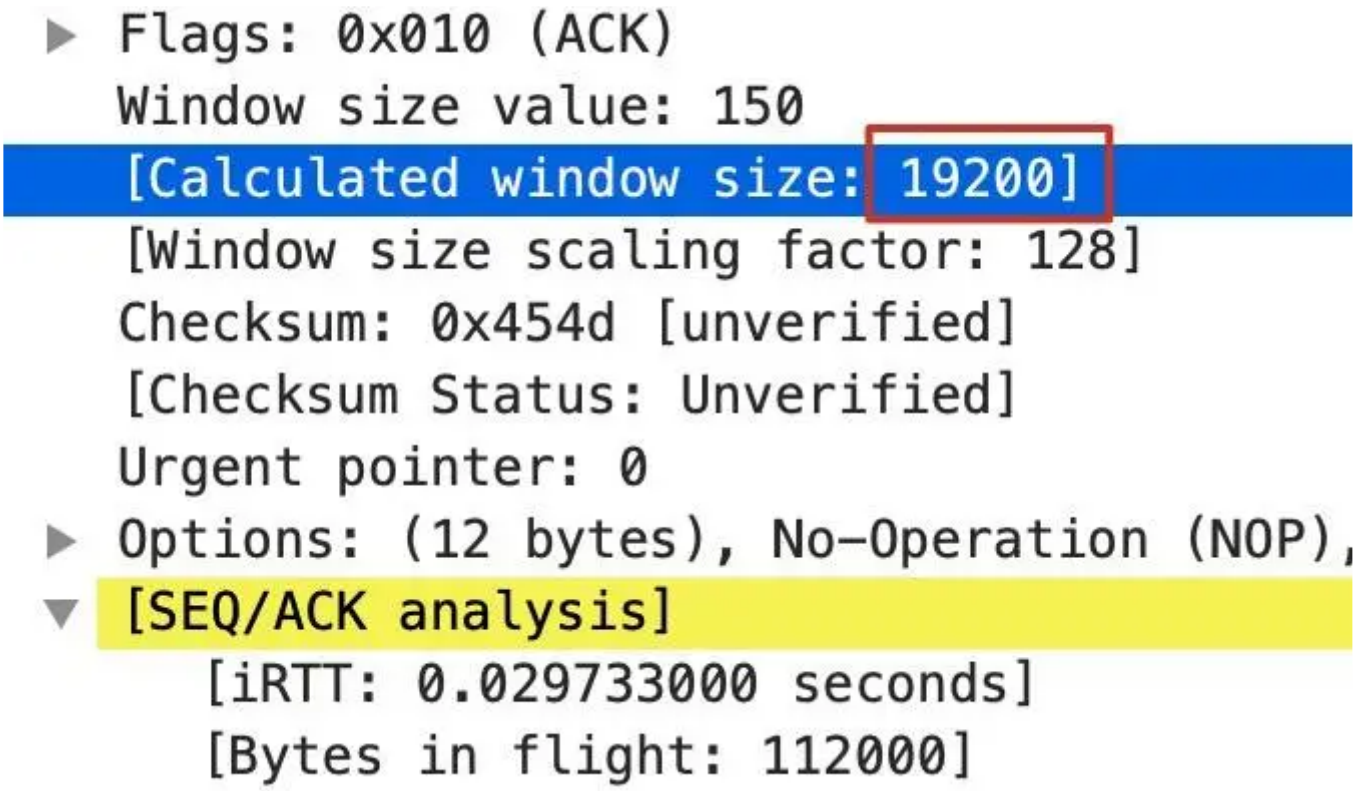
解读 TCP Window Full

TCP Window Full 这个信息非常直接明了，就是说“接收窗口满了”。不过，你可别以为这个信息是 TCP 报文里的某个字段。其实，它只是 Wireshark 通过分析得出的信息。你有没有注意到，TCP Window Full 前后是有方括号的。一般来说，**Wireshark 自己分析得到的信息，都会用方括号括起来**，而 TCP 报文本身的字段，是不会带这种方括号的。我们来看一个截图：



上面是 224 号报文的 TCP 详情，里面有不少信息也带上了方括号，比如其中的 [Bytes in flight: 112000]，这也是解读出来的，而且它跟 Window Full 关系很大。

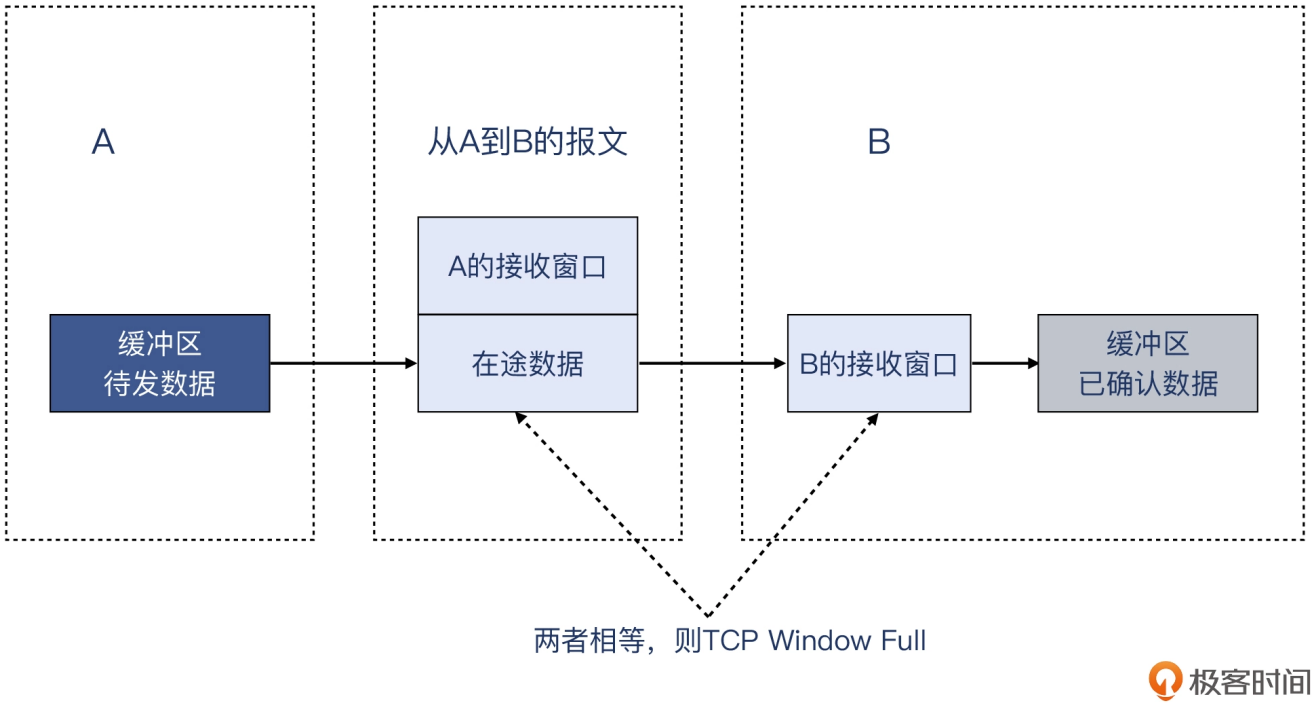
前面提到过，在途数据（或者叫在途字节数，Bytes in flight）等于接收窗口大小的时候，Wireshark 就会解读为 TCP Window Full 了。不过，如果你在上图中找一下 Window size，会发现它是 19200，而不是 Bytes in flight 的 112000，这又是为什么呢？



这是因为，我们把发送方和接收方的接收窗口搞混啦。这里你需要搞清楚：如果说在途数据的发送方是 A，接收方是 B，那么这里 Window Full 的窗口，**是 B 的接收窗口**，而不是 A 的接收窗口。上图是 A 的报文，自然没有我们要找的 B 的接收窗口信息了，那怎么找到 B 的接收窗口呢？

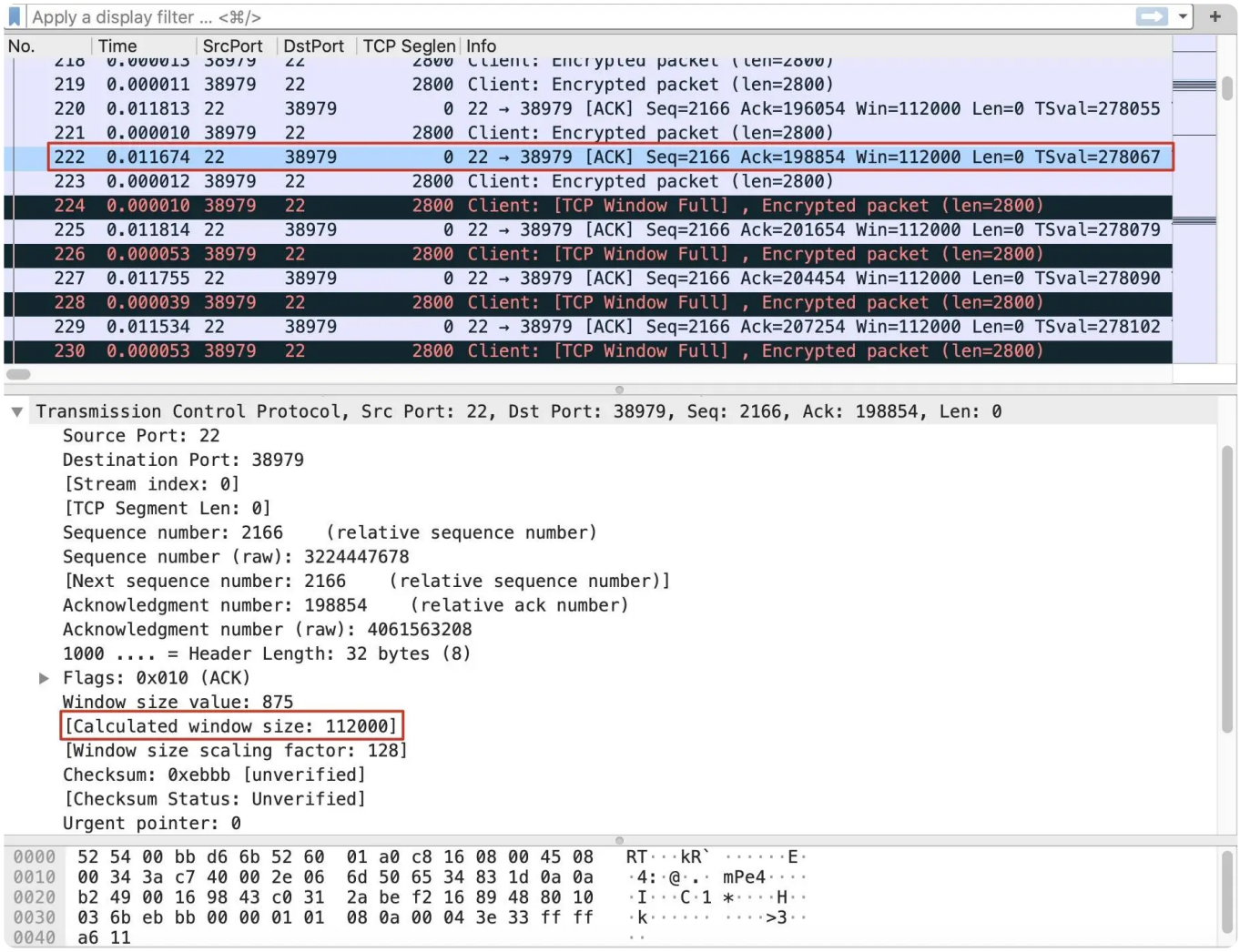
因为这次通信是 SCP 文件传输，那么 A 就是客户端，它的端口是 38979；B 就是服务端，它的端口是 22。我们的具体做法是：在抓包文件里，找到 B（也就是源端口为 22）的报文，而且应该是这个 TCP Window Full 报文之前的最近的一个，在这个报文里就有 B 的最近的接收窗口值。

单纯用文字，可能未必容易理解，我给你画了一张示意图：

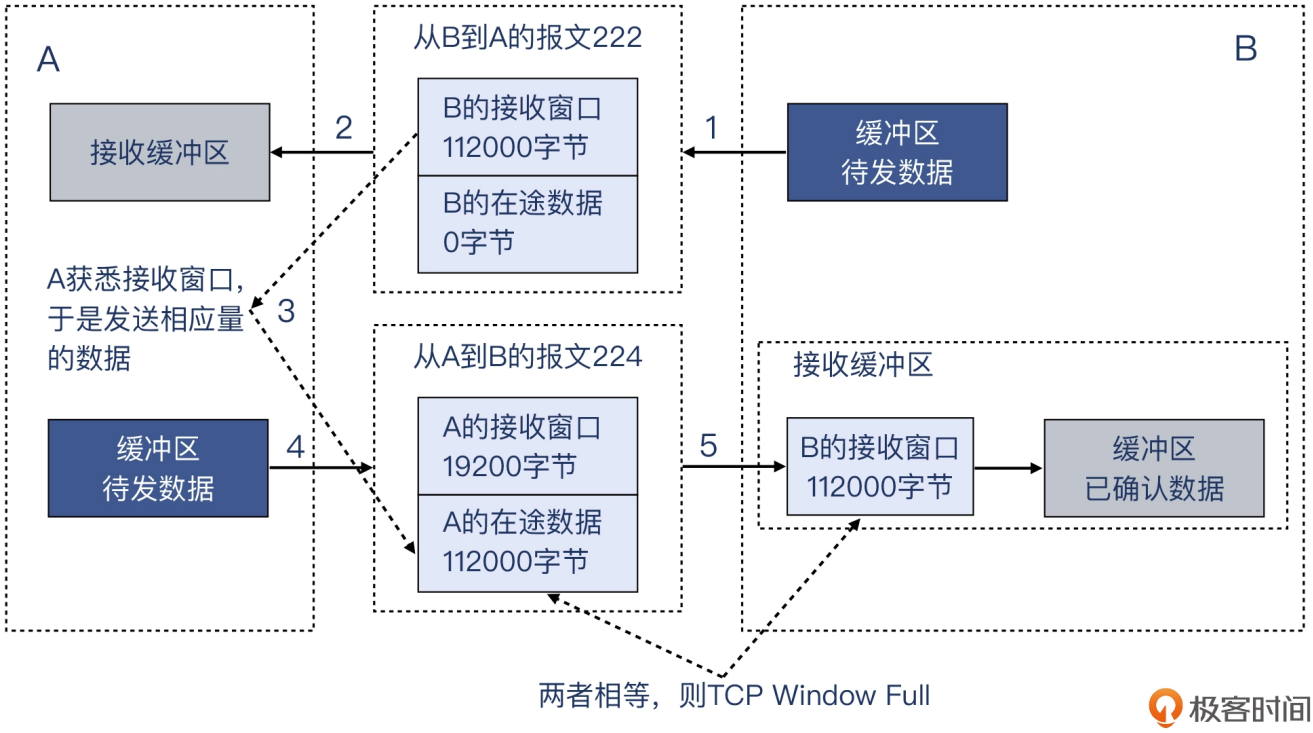


上图中，我还是用 A 指代发送端，用 B 指代接收端。当 A 的在途数据跟 B 的接收窗口大小相等时，Wireshark 就会判断出，这个接收窗口满了，这意味着：A 无法再从自己的发送缓冲区把数据发送出来了。只有当 B 回复 ACK，确认了 n 字节的数据后，A 才有可能发送最多 n 字节的数据（如果缓冲区有足够多的待发数据的话）。

让我们回到 Wireshark 窗口，找到离这个 TCP Window Full 最近的，从源端口 22 发送来的报文。我们发现，它就是下图这个 222 号报文：



可以看到，这个报文的接收窗口就是 112000，正好等于前面 224 号报文的 Bytes in flight 的 112000 字节。所以，我把前面的示意图改进一下供你参考。这里面的信息比较多，建议你耐心多花几分钟时间来充分理解其中的机制：



整个过程是这样的：

B 发送了报文 222 给 A，其中带有 B 自己的接收窗口 112000 字节。由于这是一个纯的确认报文，所以没有 TCP 载荷，也没有在途数据。

报文抵达 A 端，进入 A 的接收缓冲区。

A 从 222 号报文中得知，B 现在的接收窗口是 112000 字节，由于发送缓冲区有足够多的待发的数据，A 选择用满这个接收窗口，也就是连续发送 112000 字节。

A 把这 112000 字节的数据发送出来，成为报文 224，其中还带有 A 自己的接收窗口值 19200 字节，不过，由于这次主要是 A 向 B 传送数据，所以 B 发给 A 的基本都是纯确认报文，这些报文的载荷都是 0。极端情况下，即使 A 的接收窗口为 0，只要 B 回复的报文没有载荷，它们也是可以持续通信的。

224 报文抵达 B 端，正好填满 B 的接收窗口 112000 字节。Wireshark 分别从 222 报文中读取到 B 的接收窗口值，从 224 报文中读取到在途字节数，由于两者相等，所以 Wiresahrk 提示 TCP Window Full。而这个信息是被 Wiresahrk 展示在 224 报文中的。

自己验证 TCP Window Full

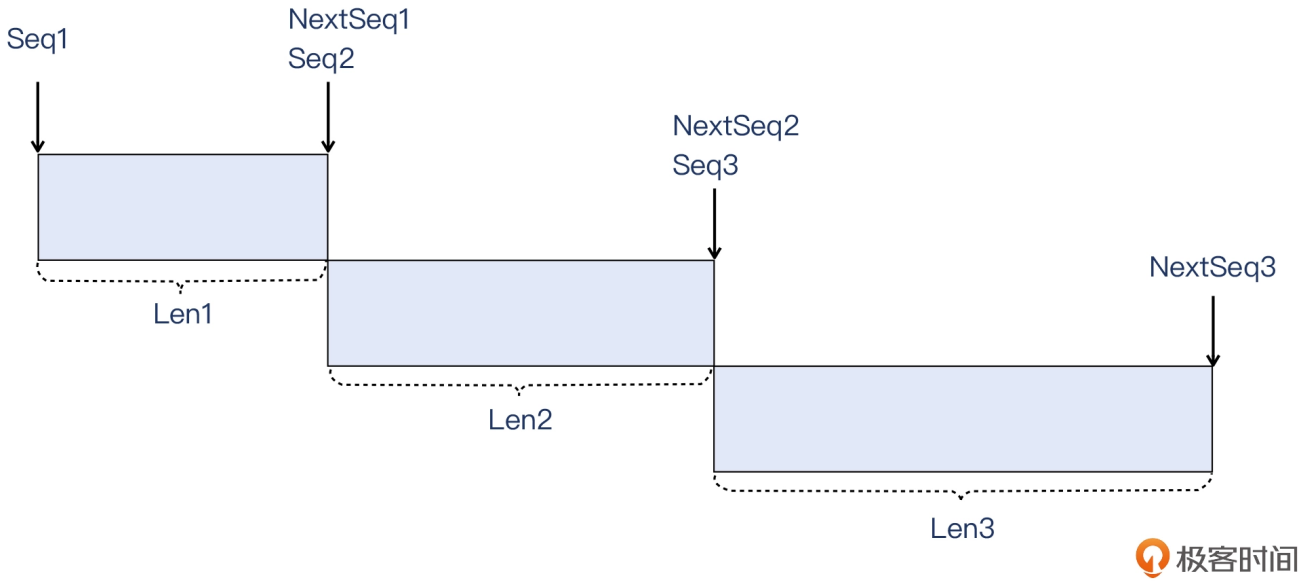
对于在途数据，既然 Wireshark 可以解读出来，那只要理解了 TCP 的原理，我们同样可以自己来计算，这不仅可以考查我们对 TCP 知识的掌握程度，同时对日常排查也有帮助。

有这么多好处，你是不是跃跃欲试了呢？不过在开始之前，我们要先学习一个新的概念。

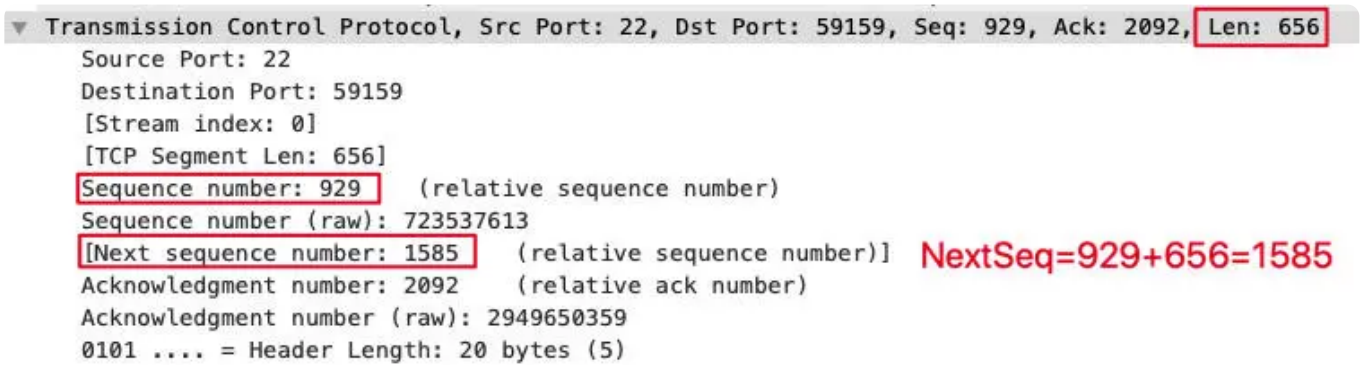
下个序列号

下个序列号，也就是 **Next Sequence Number**，缩写是 **NextSeq**。它是指当前 TCP 段的结尾字节的位置，但不包含这个结尾字节本身。很显然，下个序列号的值就是当前序列号加上当前 TCP 段的长度，也就是 **NextSeq = Seq + Len**。

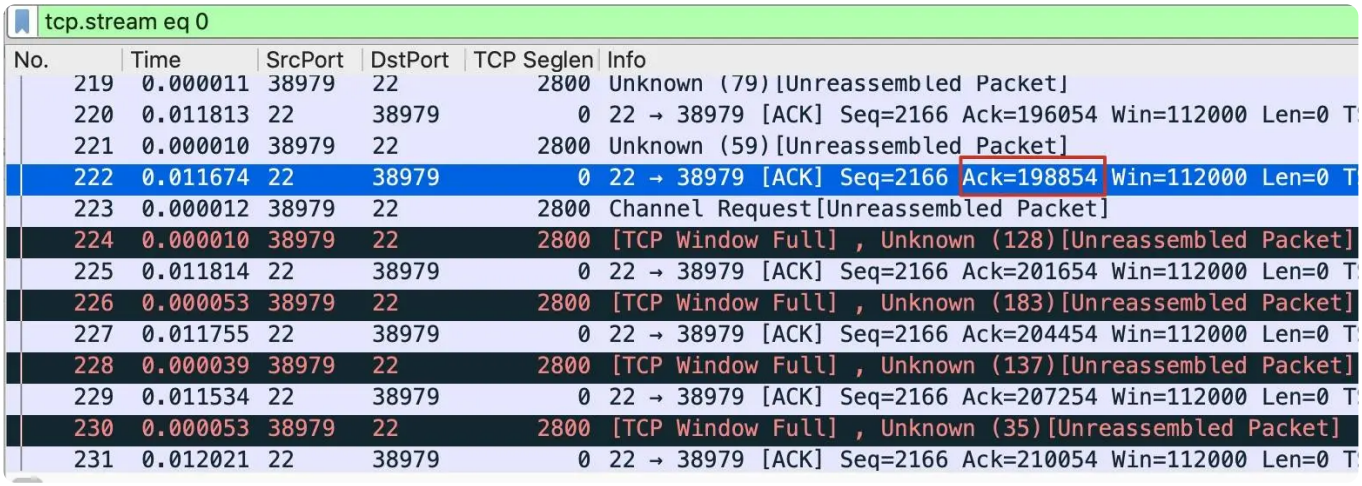
这也不难理解，因为 TCP 字节流是连续的，那么既然 Seq + Len 是这个报文的数据截止点，自然也是下一个报文的起始点，你可以参考这个示意图：



在 Wireshark 里，我们也可以找到 NextSeq 这个解读值，比如下图这样：

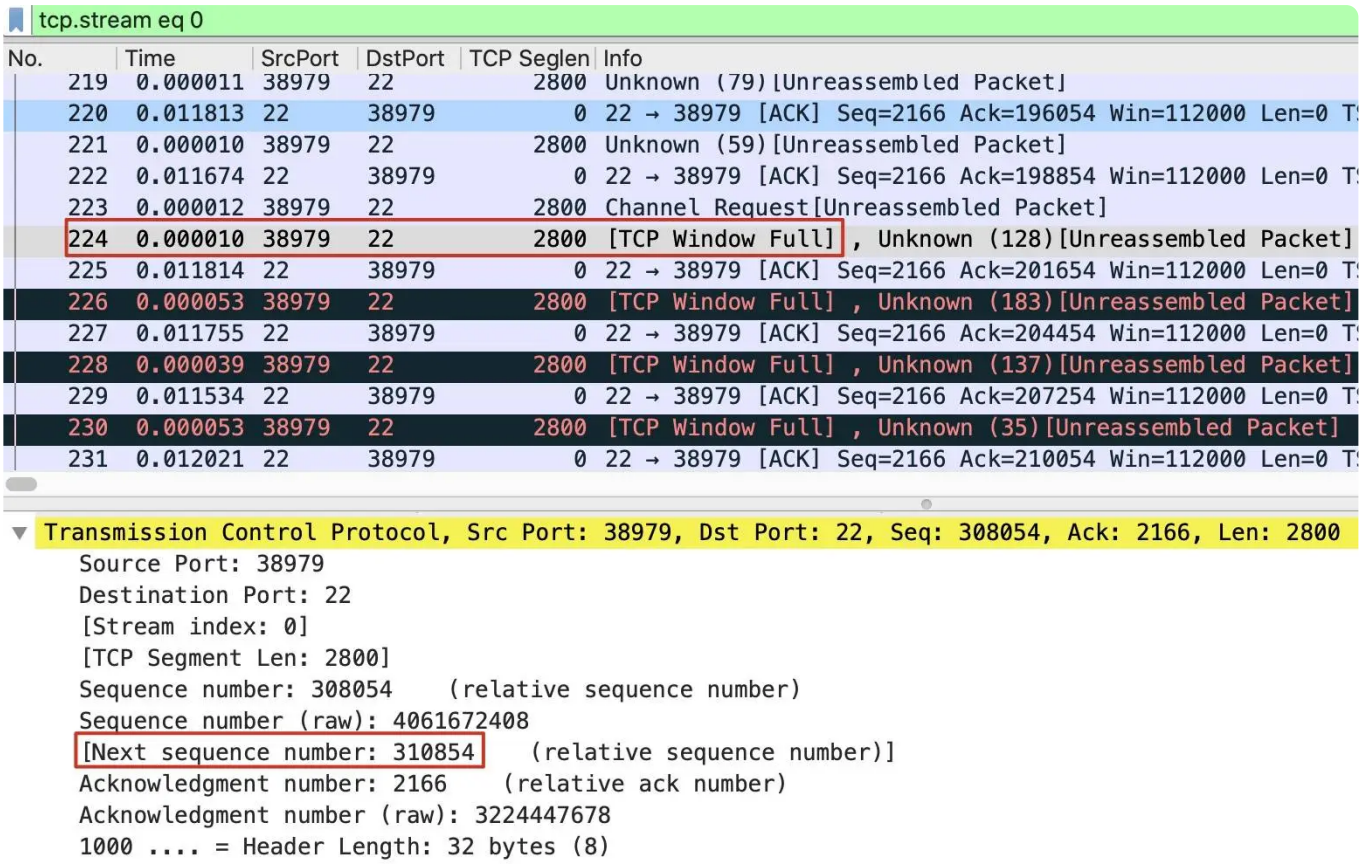


明白了 NextSeq，我们来看如何手工验证 TCP Window Full。比如，还是分析 224 号报文的这次 TCP Window Full，我们可以这么做，来验证一下在途数据是否真的是 112000 字节。



首先，跟上面的步骤类似，我们要找到 222 号报文。在这个报文里，服务端（源端口 22）告诉客户端：“我确认你发送来的 198854 字节的数据”。我们先把这个数字记为 X。

然后，我们查看 224 号报文里，客户端发送的数据到了哪个位置：



我们可以在 224 号报文的 TCP 详情页，看到 Next sequence number: 310854，而这个数字，就是客户端发送的数据的最新的位置。我们把这个数字记为 Y。当然，你也可以像前面说的那样，把 Seq 和 Len 加起来，也就是 308054 + 2800，得到的自然也是 310854。

最后，我们做一个最简单的减法：Y - X = 310854 - 198854 = 112000！这正是前面说的在途数据的大小。

恭喜你，你已经学会了如何手工计算在途数据的方法，这也意味着你对 TCP 的了解又更深入了一点。你可以这么来总结计算在途数据的方法：

> **Bytes_in_flight = latest_nextSeq - latest_ack_from_receiver**

不过，你会不会觉得，虽然这个计算方法对理解窗口有帮助，但是既然 Wireshark 会给我们提示，那这种计算也主要是自我练习而已，应该不会真的用得上吧？

这还真不好说。因为，Wireshark 在不少场景下并不会给你提示。比如，在接收窗口接近满但又不是完全满的时候，哪怕是离窗口满只差 1 个字节，Wireshark 也不会提示 TCP Window Full 了。但是，在途数据都已经逼近接收窗口的 99.9% 了，你还觉得这个肯定没有问题，或者一定没有隐患吗？

要知道，这种临界状况也很可能跟问题根因有关。那么你掌握了这个方法，就可以把排查做得更彻底了。或者，如果你想预防性能瓶颈，那么提前找到这种窗口临界满的状况，也是有益的。

到这里，我们可以回答开始时候的问题了：为什么上节课里，没有看到 TCP Window Full 这种提示呢？我们看一下当时的报文状况吧。

在 22:30:39.067477，接收端的确认号为 7105632：

No.	Time	SrcPort	DstPort	TCP Seglen	TTL	Acknowledgment	nextSeq	Calculated window	Bytes in flight
5812	22:30:39.068477	22	59159	0	55	7105632	5153	65728	
5813	22:30:39.068534	59159	22	2920	64	5153	7171332	34688	65700

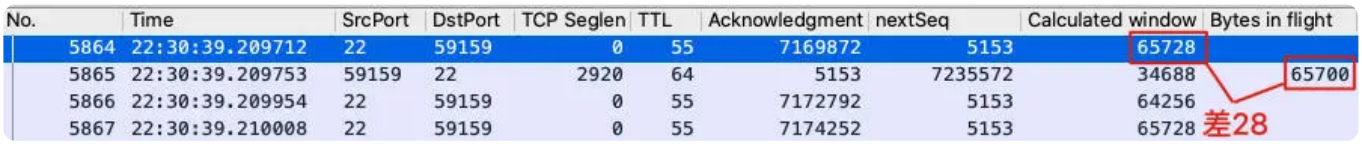
然后在 22:30:39.209712，接收端的确认号为 7169872：

No.	Time	SrcPort	DstPort	TCP Seglen	TTL	Acknowledgment	nextSeq	Calculated window	Bytes in flight
5864	22:30:39.209712	22	59159	0	55	7169872	5153	65728	
5865	22:30:39.209753	59159	22	2920	64	5153	7235572	34688	65700

这两个报文的时间跨度正好是 141ms 左右，也就是这次传输里面的**往返时间**。在这个往返时间里，接收端确认了多少数据呢？是 7169872-7105632=64240，也就是 64KB。这个

就是 99.9% 逼近 TCP Window Full 了，但是因为还差小几十个字节，所以 Wireshark 并没有提示 TCP Window Full！

你可能还想追问：那为什么不把这剩余的 0.1% 的窗口“榨干”，非要留一点呢？我们看一下当时的接收窗口和在途数据的具体情况，就以上面选择的 5864 报文附近为例：



接收端（源端口为 22）的接收窗口为 65728，发送端（源端口为 59159）的在途数据为 65700，两者相差只有 28 字节。对于发送端来说，没有必要为了这区区 28 字节再发送一个小报文了，等接收窗口空余出多一点的空间后再动身不迟。

> 如果你还没学习过上节课的内容，可能会对这些信息感到疑惑，建议先去把上节课学完，再来学习这一讲，效果更好。

TCP Window Full 对传输的影响

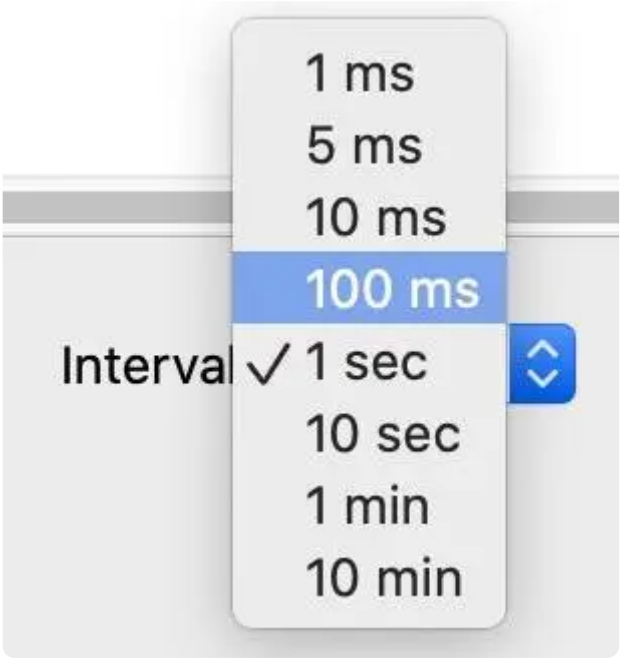
好了，现在我们已经对 TCP Window Full 做了充分的分析，而且也明白了：这就是**接收端的接收窗口小于发送端的发送能力而出现的状况**。我们也很容易得出推论：瓶颈在接收端，**TCP Window Full 也确实会影响传输速度**。

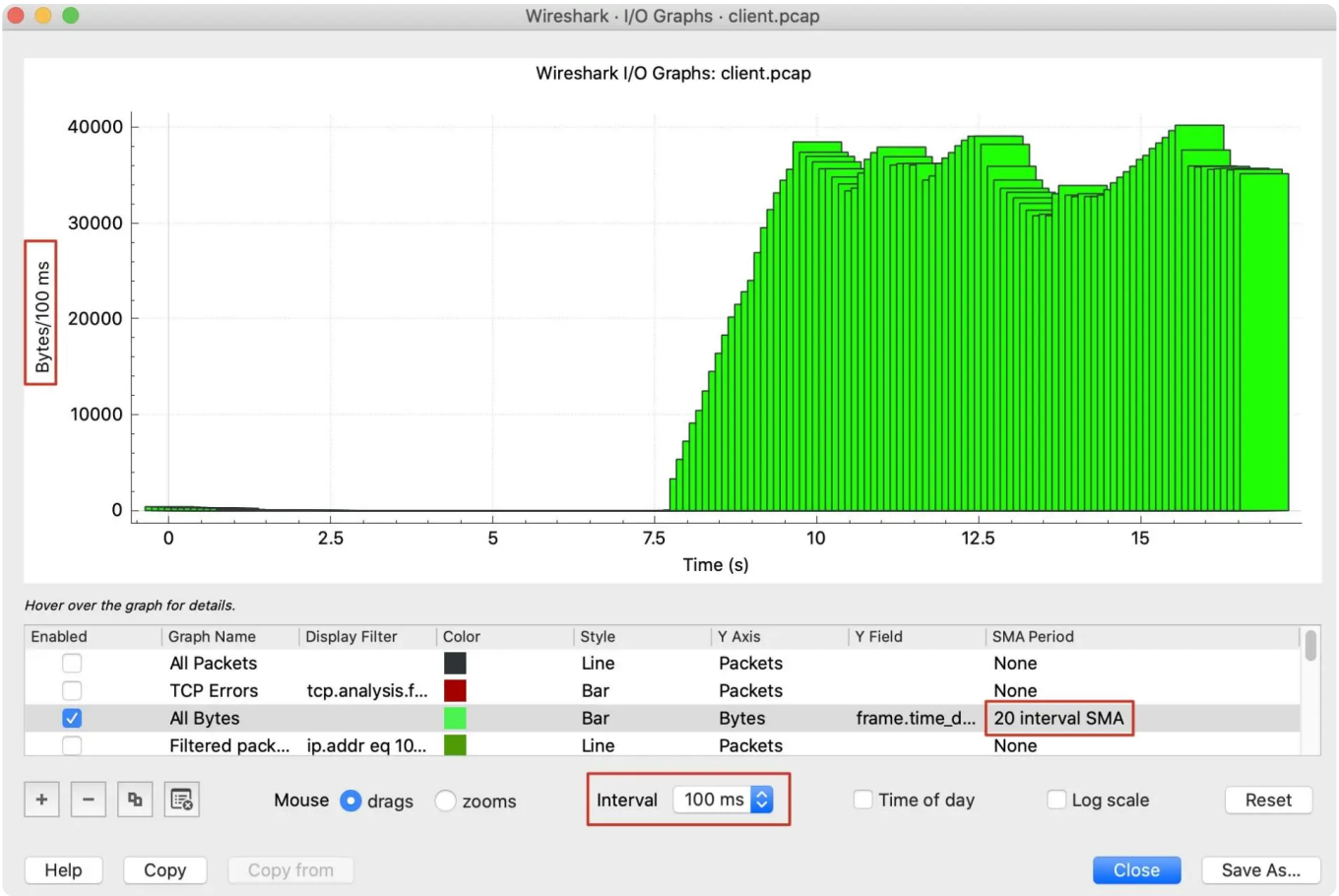
春节刚过，你可能对高速公路上的状况也感受深刻吧！很多路段出现了堵车，这就相当于 TCP Window Full，更多的车辆上不了高速了，只好堵在外面。如果高速公路的路更宽、车速更快，那么就相当于接收窗口变得更大，车辆就能进更多，也就相当于 Bytes in flight 更大了。这么说来，TCP 流量控制和高速车流控制这两个领域也有不少共通之处，说不定双方都互有借鉴呢。

回到客户这次的案例。我们看看，这次的传输速度是多少呢？在上节课里，我介绍了在 Wireshark 里查看 TCP 传输速度的两种方法。比如，我们现在用 I/O Graph 来看一下：

> 补充：如果你的 I/O Graph 显示的不是这种图，那需要像图中这样：
>
> 选中 All Bytes 指标；
>
> Y 轴的单位选为 Bytes。

这个图不能说不对，但柱子比较粗，看起来不是很精确。这是因为，它默认是以 1 秒为间隔而计算的速度。但是 TCP 传输中途，很可能每过几毫秒都有所变化，所以，如果我们要看更加精细的图，可以调整一下粒度，把 Interval 从 1sec 改为 100ms，看看会怎么样：





这样看起来精确了很多。我已经把这个抓包文件上传到Gitee，你可以下载后，依照这里的步骤，把我做过的排查工作也演练一遍，这对你掌握课程知识是很有帮助的。

> 补充：如果你用 Wireshark 查看到的 I/O Graph 跟我这里的不同，那可以对比一下 Interval 和 SMA period 这两个配置是否跟图中的一致。

这里有个小的注意点：因为我们选择的间隔是 100ms，所以 Y 轴的数字就是 Bytes/100ms，换算成 Bytes/s 的话，要把 Y 轴的数字再**乘以 10**。从图上看，在一开始的 8 秒几乎没有数据传输发生，从第 8 秒开始速度上到了 400KB/s（就是图上的 40K*10）左右，一直到结束都大致维持在 300~400KB/s 这个区间里。

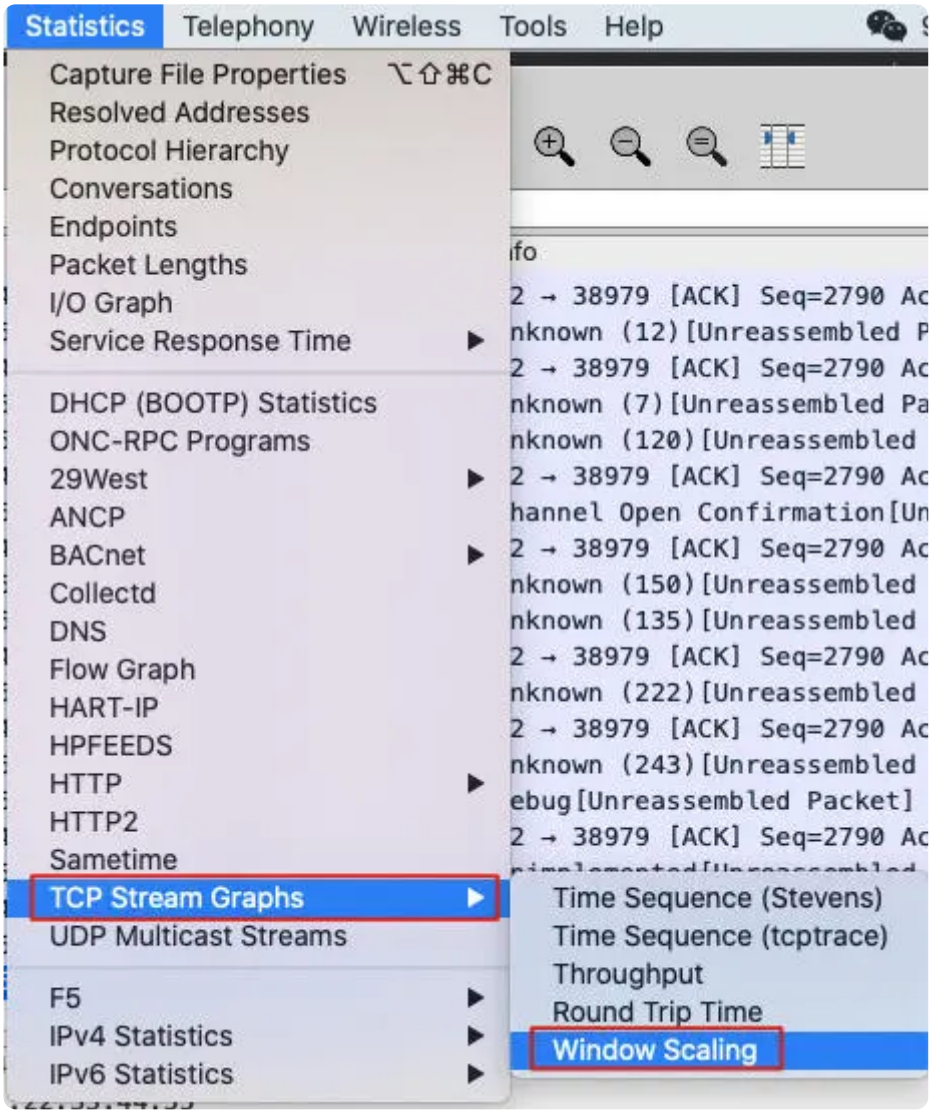
继续深挖窗口

一般来说，接收窗口、拥塞窗口、发送窗口，这些都不是一上来就是一个很大的值的，而是跟汽车起步阶段类似，逐步跑起来的。那么这就产生了一个很有意思的话题：这些窗口之间都是怎么协调的呢？直观上感觉，无论哪个更快了，另外两个就要受影响。

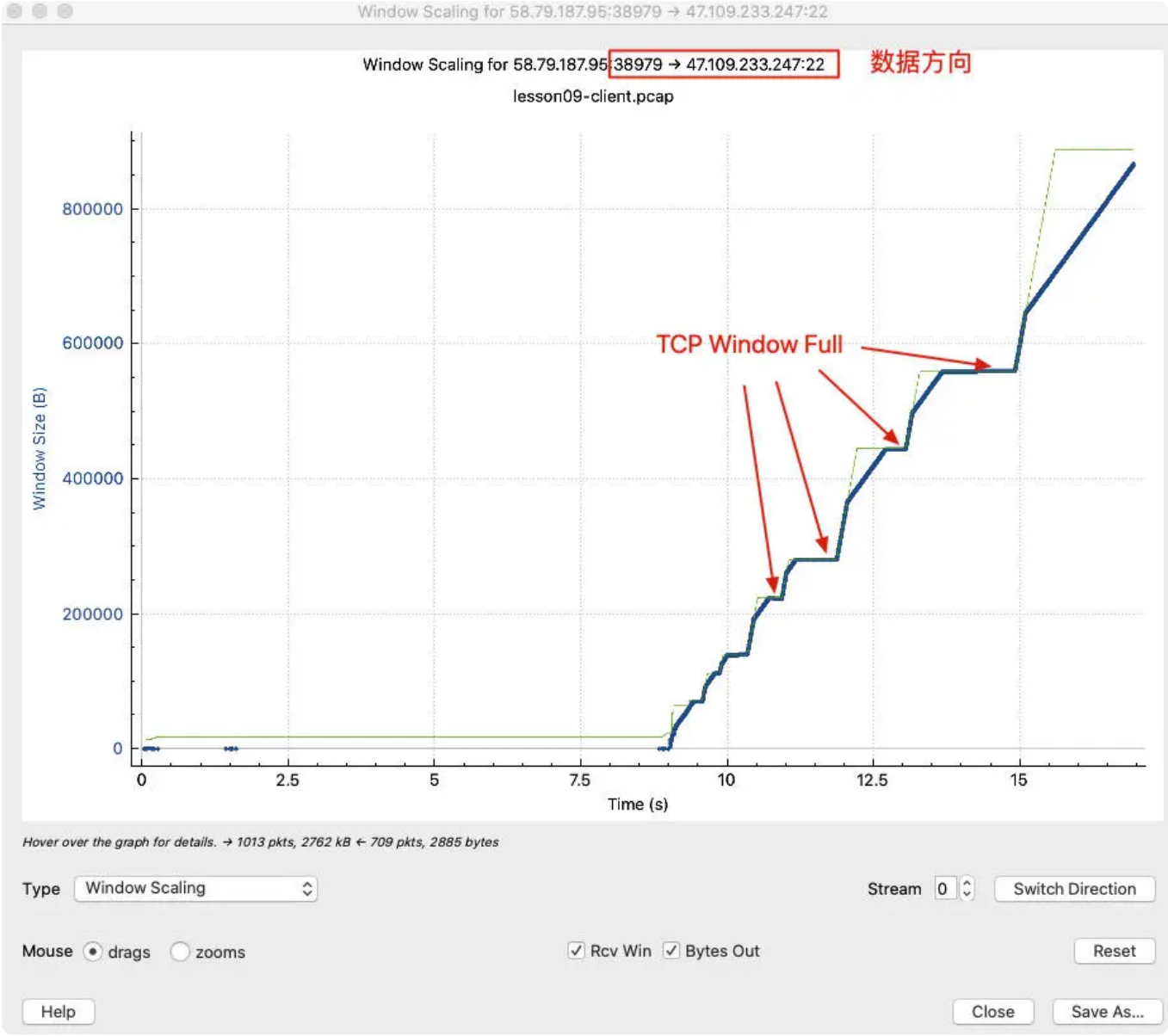
我们很容易理解，假设起始值相同，如果接收窗口增长的速度小于拥塞窗口的增长速度，那么接收窗口就成了瓶颈；反过来说，拥塞窗口增速更小，那么它就成了瓶颈。

当接收窗口成为瓶颈的时候，很容易就出现这里的 TCP Window Full 的现象。不过，我们这么多讨论都是基于文字，如果有更加直观的方式，让我们理解这个现象就更好了。这里，我们就可以再学一个 Wireshark 的小工具：TCP Stream Graphs 里面的 **Window Scaling**。

我们还是打开 Wireshark 的 Statistics 下拉菜单，找到 TCP Stream Graphs，在二级菜单中，选择 Window Scaling：



这时候就能看到 Windows Scaling 的趋势图了：

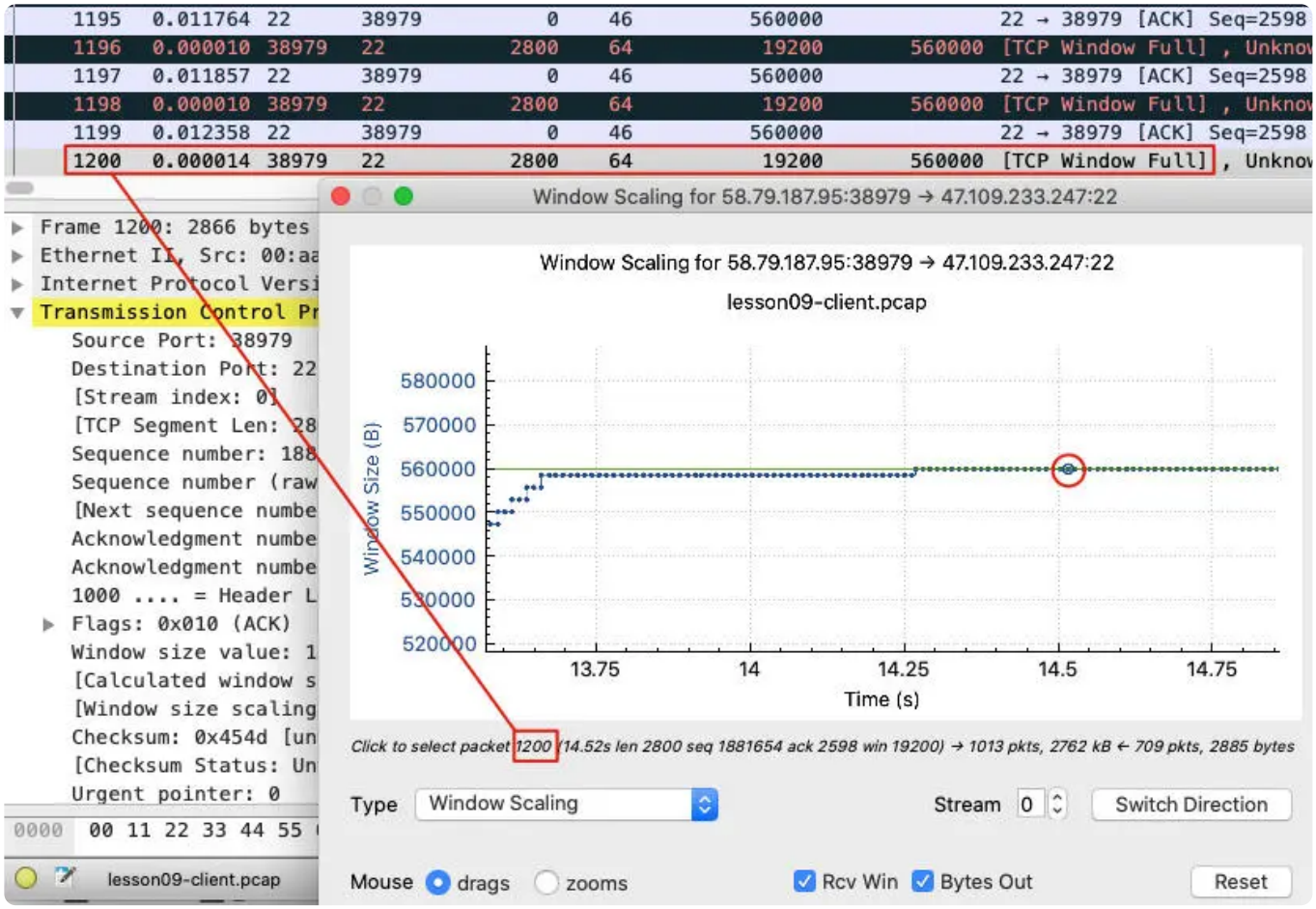


我就直接给你把关键信息标注出来了。这里主要是两个关键点。

数据流的方向要找对：比如这次传输是从客户端向 SSH 服务端发送数据，所以要确认这是从一个高端口向 22 端口发送数据的流向。如果搞反了，那图就变成了 SSH 服务端回复的 ACK 报文了，不是你要分析的传输速度了。

定位 TCP Window Full：在这里，Receive Window 是“阶梯”式的，每次变化后会保持在一个“平台”一小段时间，那么这时候 Bytes Out（发送的数据，也就是 Bytes in flight）就有可能触及这个“平台”，每次真的碰上的时候，就是一次 TCP Window Full。

我们可以看一个例子。图中的蓝线代表 Bytes Out，绿线代表 Receive Window。你可以像我这样，在这几个“平台”区域，找到蓝线和绿线的汇合点，然后在这些点上点击鼠标左键，就能定位到 TCP Window Full 事件了。



上图中，我用鼠标放大了一个“平台”，然后选中了一个 Receive Window 和 Bytes Out 重合的点，它是 1200 号报文，主窗口也自动定位到了这个报文，果然它也是一次 TCP Window Full。

验证传输公式

在上节课里，我们推导出了 TCP 传输的核心公式：**速度 = 窗口 / 往返时间**。既然当前案例里 TCP Window Full 的时候，Bytes in flight 跟接收窗口相等，那么在这个公式里的窗口，是否就是 Bytes in flight 呢？我们来验证一下。

还是在 Wireshark 窗口里，我们要添加这么几个自定义列，以便进行数据比对：

Acknowledgement Number：确认号

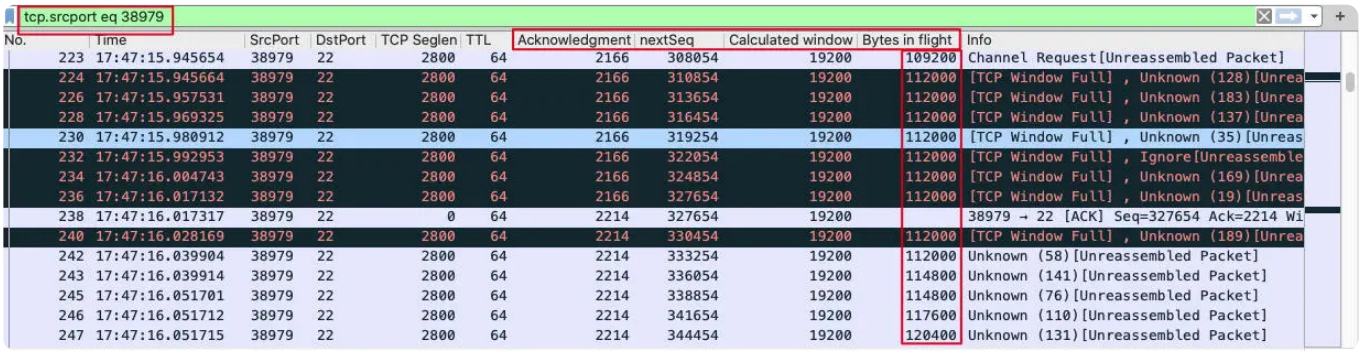
Next Sequence Number：下个序列号

Caculated Window Size：计算后的接收窗口

Bytes in flight：在途字节数

> 补充：如果对怎么添加自定义列还不清楚，可以复习第 5 讲中的添加 TTL 自定义列的部分。

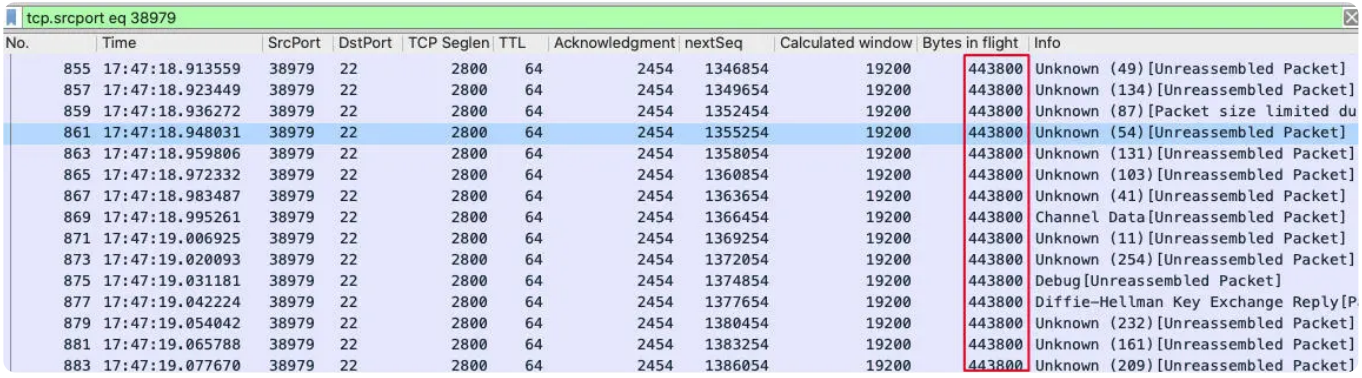
另外，因为我们要集中检查发送端的 Bytes in flight，就需要把源端口 38979 的报文过滤出来，这样就不会被另一个方向的报文给干扰了。



tcp.srcport eq 38979

No.	Time	SrcPort	DstPort	TCP Seglen	TTL	Acknowledgment	nextSeq	Calculated window	Bytes in flight	Info
223	17:47:15.945654	38979	22	2800	64	2166	308054	19200	109200	Channel Request[Unreassembled Packet]
224	17:47:15.945664	38979	22	2800	64	2166	310854	19200	112000	[TCP Window Full] , Unknown (128)[Unrea
226	17:47:15.957531	38979	22	2800	64	2166	313654	19200	112000	[TCP Window Full] , Unknown (183)[Unrea
228	17:47:15.969325	38979	22	2800	64	2166	316454	19200	112000	[TCP Window Full] , Unknown (137)[Unrea
230	17:47:15.980912	38979	22	2800	64	2166	319254	19200	112000	[TCP Window Full] , Unknown (35)[Unreas
232	17:47:15.992953	38979	22	2800	64	2166	322054	19200	112000	[TCP Window Full] , Ignore[Unreassemble
234	17:47:16.004743	38979	22	2800	64	2166	324854	19200	112000	[TCP Window Full] , Unknown (169)[Unrea
236	17:47:16.017132	38979	22	2800	64	2166	327654	19200	112000	[TCP Window Full] , Unknown (19)[Unreas
238	17:47:16.017317	38979	22	0	64	2214	327654	19200		38979 → 22 [ACK] Seq=327654 Ack=2214 Wi
240	17:47:16.028169	38979	22	2800	64	2214	330454	19200	112000	[TCP Window Full] , Unknown (189)[Unrea
242	17:47:16.039904	38979	22	2800	64	2214	333254	19200	112000	Unknown (58)[Unreassembled Packet]
243	17:47:16.039914	38979	22	2800	64	2214	336054	19200	114800	Unknown (141)[Unreassembled Packet]
245	17:47:16.051701	38979	22	2800	64	2214	338854	19200	114800	Unknown (76)[Unreassembled Packet]
246	17:47:16.051712	38979	22	2800	64	2214	341654	19200	117600	Unknown (110)[Unreassembled Packet]
247	17:47:16.051715	38979	22	2800	64	2214	344454	19200	120400	Unknown (131)[Unreassembled Packet]

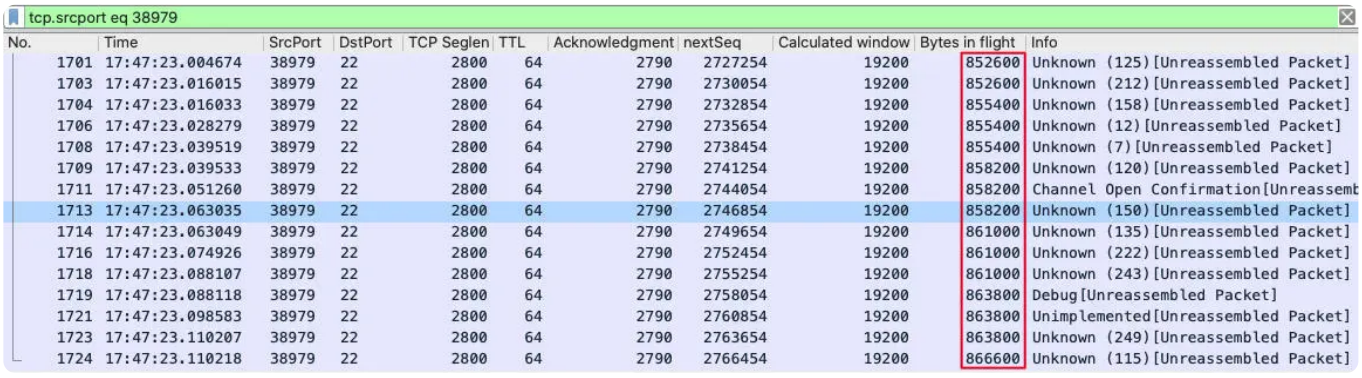
在这里，我们看到的 Bytes in flight 是 112000 字节左右。从右边滚动条的位置来看，这是在传输过程的初期。让我们滚动到中间和后期，看看这些在途字节数是多少：



tcp.srcport eq 38979

No.	Time	SrcPort	DstPort	TCP Seglen	TTL	Acknowledgment	nextSeq	Calculated window	Bytes in flight	Info
855	17:47:18.913559	38979	22	2800	64	2454	1346854	19200	443800	Unknown (49)[Unreassembled Packet]
857	17:47:18.923449	38979	22	2800	64	2454	1349654	19200	443800	Unknown (134)[Unreassembled Packet]
859	17:47:18.936272	38979	22	2800	64	2454	1352454	19200	443800	Unknown (87)[Packet size limited du
861	17:47:18.948031	38979	22	2800	64	2454	1355254	19200	443800	Unknown (54)[Unreassembled Packet]
863	17:47:18.959806	38979	22	2800	64	2454	1358054	19200	443800	Unknown (131)[Unreassembled Packet]
865	17:47:18.972332	38979	22	2800	64	2454	1360854	19200	443800	Unknown (103)[Unreassembled Packet]
867	17:47:18.983487	38979	22	2800	64	2454	1363654	19200	443800	Unknown (41)[Unreassembled Packet]
869	17:47:18.995261	38979	22	2800	64	2454	1366454	19200	443800	Channel Data[Unreassembled Packet]
871	17:47:19.006925	38979	22	2800	64	2454	1369254	19200	443800	Unknown (11)[Unreassembled Packet]
873	17:47:19.020093	38979	22	2800	64	2454	1372054	19200	443800	Unknown (254)[Unreassembled Packet]
875	17:47:19.031181	38979	22	2800	64	2454	1374854	19200	443800	Debug[Unreassembled Packet]
877	17:47:19.042224	38979	22	2800	64	2454	1377654	19200	443800	Diffie-Hellman Key Exchange Reply[P
879	17:47:19.054042	38979	22	2800	64	2454	1380454	19200	443800	Unknown (232)[Unreassembled Packet]
881	17:47:19.065788	38979	22	2800	64	2454	1383254	19200	443800	Unknown (161)[Unreassembled Packet]
883	17:47:19.077670	38979	22	2800	64	2454	1386054	19200	443800	Unknown (209)[Unreassembled Packet]

中期这里的 Calculated Window Size 明显增大了，到了 445312 字节。再看看后半程：



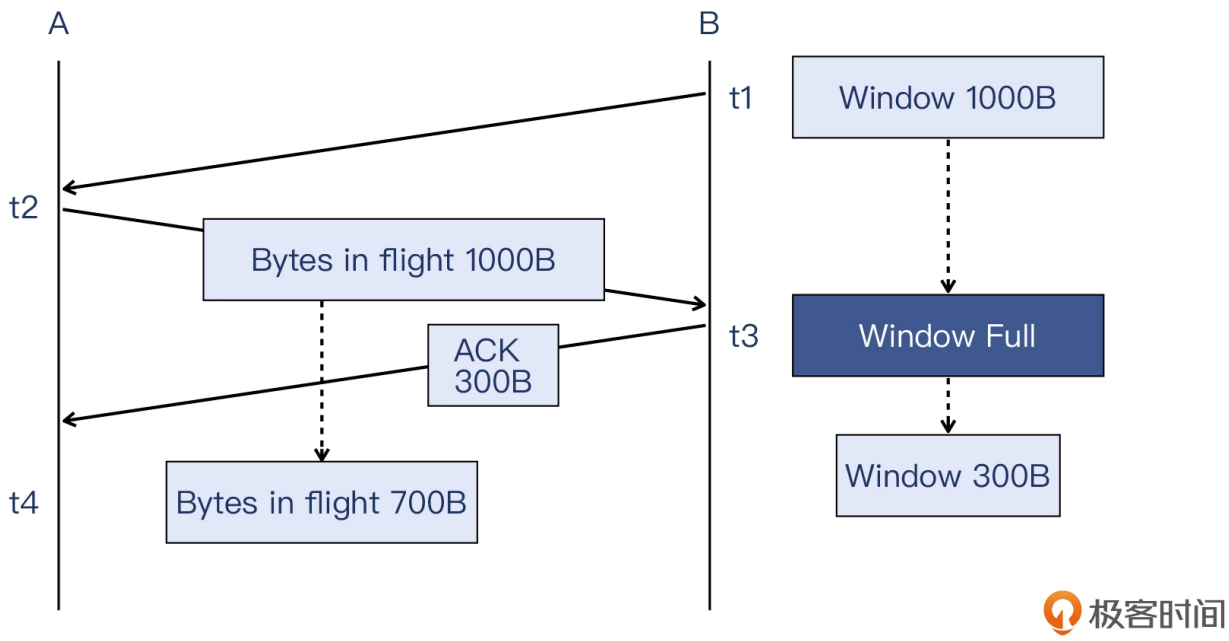
tcp.srcport eq 38979

No.	Time	SrcPort	DstPort	TCP Seglen	TTL	Acknowledgment	nextSeq	Calculated window	Bytes in flight	Info
1701	17:47:23.004674	38979	22	2800	64	2790	2727254	19200	852600	Unknown (125)[Unreassembled Packet]
1703	17:47:23.016015	38979	22	2800	64	2790	2730054	19200	852600	Unknown (212)[Unreassembled Packet]
1704	17:47:23.016033	38979	22	2800	64	2790	2732854	19200	855400	Unknown (158)[Unreassembled Packet]
1706	17:47:23.028279	38979	22	2800	64	2790	2735654	19200	855400	Unknown (12)[Unreassembled Packet]
1708	17:47:23.039519	38979	22	2800	64	2790	2738454	19200	855400	Unknown (7)[Unreassembled Packet]
1709	17:47:23.039533	38979	22	2800	64	2790	2741254	19200	858200	Unknown (120)[Unreassembled Packet]
1711	17:47:23.051260	38979	22	2800	64	2790	2744054	19200	858200	Channel Open Confirmation[Unreassemb
1713	17:47:23.063035	38979	22	2800	64	2790	2746854	19200	858200	Unknown (150)[Unreassembled Packet]
1714	17:47:23.063049	38979	22	2800	64	2790	2749654	19200	861000	Unknown (135)[Unreassembled Packet]
1716	17:47:23.074926	38979	22	2800	64	2790	2752454	19200	861000	Unknown (222)[Unreassembled Packet]
1718	17:47:23.088107	38979	22	2800	64	2790	2755254	19200	861000	Unknown (243)[Unreassembled Packet]
1719	17:47:23.088118	38979	22	2800	64	2790	2758054	19200	863800	Debug[Unreassembled Packet]
1721	17:47:23.098583	38979	22	2800	64	2790	2760854	19200	863800	Unimplemented[Unreassembled Packet]
1723	17:47:23.110207	38979	22	2800	64	2790	2763654	19200	863800	Unknown (249)[Unreassembled Packet]
1724	17:47:23.110218	38979	22	2800	64	2790	2766454	19200	866600	Unknown (115)[Unreassembled Packet]

最后阶段已经达到 863800 字节。综合这三个阶段来看，折中值差不多在 400KB 左右，我们把它除以 RTT 0.029 秒，得到的是 400KB/0.029s=13790KB/s。显然，这个数值远超

过前面 I/O Graph 里看到的 300~400KB/s。这是怎么回事呢？难道我们的核心公式是错的吗？

不知道你有没有考虑到这个问题：Bytes in flight 是指真的一直在网络上两头不着吗？一般来说，数据到了接收端，接收端就发送 ACK 确认这部分数据，然后 TCP Window 就往下降了。比如 ACK 300 字节，那么 TCP Window 就又空出来 300 字节，也就是发送端又可以新发送 300 字节了。



像图上这种情况：

B 通知 A：“我的接收窗口是 1000”；

A 向 B 发送了 1000 字节，此时 B 的接收窗口满；

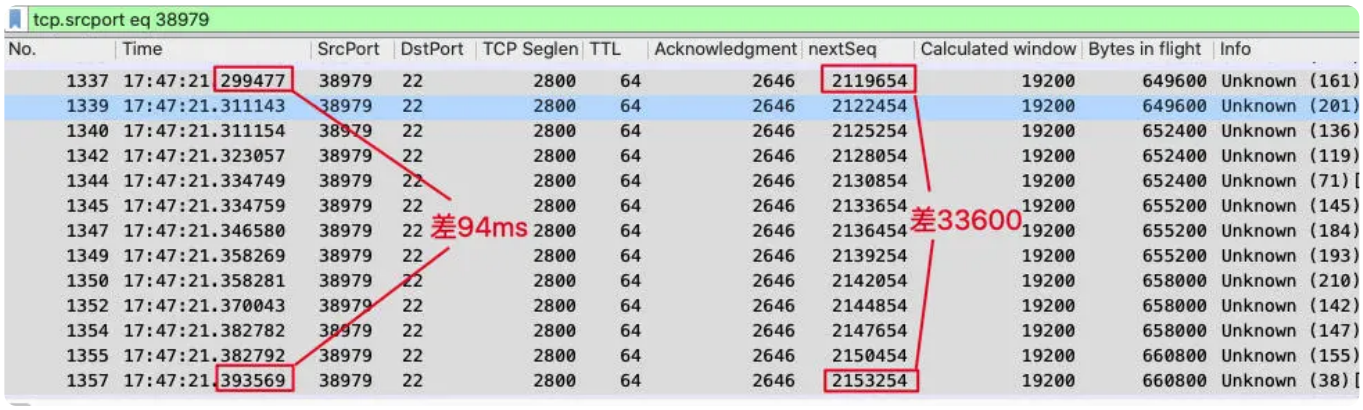
B 向 A 确认了 300 字节的数据，自身的接收窗口也扩大为 300 字节；

A 的在途字节数也从 1000 字节变成 700 字节，因为刚刚有 300 字节被 B 确认了。

图中的 t1 到 t4 表示时间点。t2 到 t4 就是一次往返的时间，在这个往返时间内，被传输的数据是 1000 字节吗？不是。因为被确认的只有 300 字节，所以传输完的也只有这 300 字节，速度也就不是 1000/RTT，而是 300/RTT！我们可以把核心公式做一下改进，变成下面这个：

> **velocity = acked_data/RTT**

我们再用改进后的公式来计算这次的速度。我们可以选择传输中间偏后面一点的报文来做分析。比如下图中，我们选择 1337 号和 1357 号报文为起始和截止点，计算 NextSeq 的差值，还有时间的差值，然后两个差值相除。

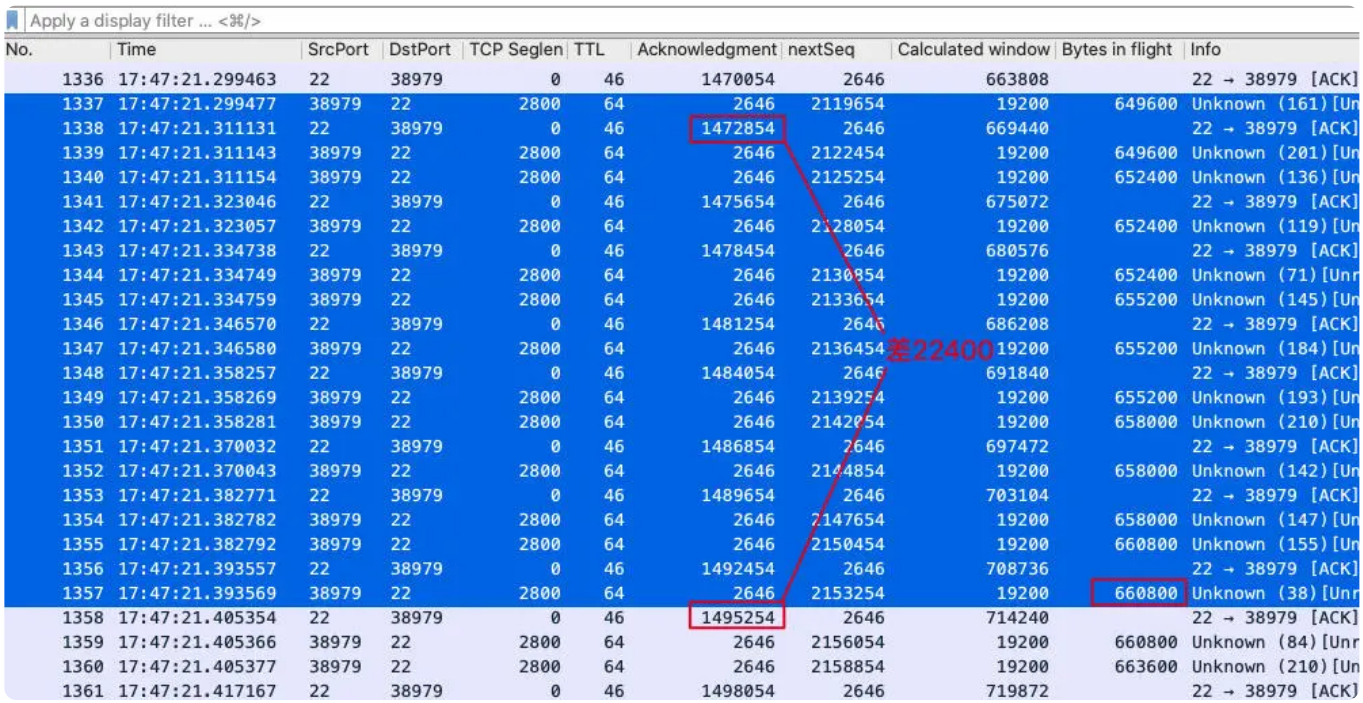


No.	Time	SrcPort	DstPort	TCP Seglen	TTL	Acknowledgment	nextSeq	Calculated window	Bytes in flight	Info
1337	17:47:21.299477	38979	22	2800	64	2646	2119654	19200	649600	Unknown (161)
1339	17:47:21.311143	38979	22	2800	64	2646	2122454	19200	649600	Unknown (201)
1340	17:47:21.311154	38979	22	2800	64	2646	2125254	19200	652400	Unknown (136)
1342	17:47:21.323057	38979	22	2800	64	2646	2128054	19200	652400	Unknown (119)
1344	17:47:21.334749	38979	22	2800	64	2646	2130854	19200	652400	Unknown (71) [
1345	17:47:21.334759	38979	22	2800	64	2646	2133654	19200	655200	Unknown (145)
1347	17:47:21.346580	38979	22	2800	64	2646	2136454	19200	655200	Unknown (184)
1349	17:47:21.358269	38979	22	2800	64	2646	2139254	19200	655200	Unknown (193)
1350	17:47:21.358281	38979	22	2800	64	2646	2142054	19200	658000	Unknown (210)
1352	17:47:21.370043	38979	22	2800	64	2646	2144854	19200	658000	Unknown (142)
1354	17:47:21.382782	38979	22	2800	64	2646	2147654	19200	658000	Unknown (147)
1355	17:47:21.382792	38979	22	2800	64	2646	2150454	19200	660800	Unknown (155)
1357	17:47:21.393569	38979	22	2800	64	2646	2153254	19200	660800	Unknown (38) [

33600/0.094 = **357KB/s**。是不是很接近 I/O Graph 的值了？看来这样计算才是正确的！

那为什么在上节课里我们就可以用 **窗口 / 往返时间** 来计算速度，而且数值也很准呢？而这种方法用到这里就完全不对呢？

这是因为上节课的案例，在途数据一旦到了接收端，都被及时确认了。而当前这个案例里面并没有这样。也就是说，这次的案例，出现了“滞留”现象。



No.	Time	SrcPort	DstPort	TCP Seglen	TTL	Acknowledgment	nextSeq	Calculated window	Bytes in flight	Info
1336	17:47:21.299463	22	38979	0	46	1470054	2646	663808		22 → 38979 [ACK]
1337	17:47:21.299477	38979	22	2800	64	2646	2119654	19200	649600	Unknown (161) [Un
1338	17:47:21.311131	22	38979	0	46	1472854	2646	669440		22 → 38979 [ACK]
1339	17:47:21.311143	38979	22	2800	64	2646	2122454	19200	649600	Unknown (201) [Un
1340	17:47:21.311154	38979	22	2800	64	2646	2125254	19200	652400	Unknown (136) [Un
1341	17:47:21.323046	22	38979	0	46	1475654	2646	675072		22 → 38979 [ACK]
1342	17:47:21.323057	38979	22	2800	64	2646	2128054	19200	652400	Unknown (119) [Un
1343	17:47:21.334738	22	38979	0	46	1478454	2646	680576		22 → 38979 [ACK]
1344	17:47:21.334749	38979	22	2800	64	2646	2130854	19200	652400	Unknown (71) [Unr
1345	17:47:21.334759	38979	22	2800	64	2646	2133654	19200	655200	Unknown (145) [Un
1346	17:47:21.346570	22	38979	0	46	1481254	2646	686208		22 → 38979 [ACK]
1347	17:47:21.346580	38979	22	2800	64	2646	2136454	19200	655200	Unknown (184) [Un
1348	17:47:21.358257	22	38979	0	46	1484054	2646	691840		22 → 38979 [ACK]
1349	17:47:21.358269	38979	22	2800	64	2646	2139254	19200	655200	Unknown (193) [Un
1350	17:47:21.358281	38979	22	2800	64	2646	2142054	19200	658000	Unknown (210) [Un
1351	17:47:21.370032	22	38979	0	46	1486854	2646	697472		22 → 38979 [ACK]
1352	17:47:21.370043	38979	22	2800	64	2646	2144854	19200	658000	Unknown (142) [Un
1353	17:47:21.382771	22	38979	0	46	1489654	2646	703104		22 → 38979 [ACK]
1354	17:47:21.382782	38979	22	2800	64	2646	2147654	19200	658000	Unknown (147) [Un
1355	17:47:21.382792	38979	22	2800	64	2646	2150454	19200	660800	Unknown (155) [Un
1356	17:47:21.393557	22	38979	0	46	1492454	2646	708736		22 → 38979 [ACK]
1357	17:47:21.393569	38979	22	2800	64	2646	2153254	19200	660800	Unknown (38) [Unr
1358	17:47:21.405354	22	38979	0	46	1495254	2646	714240		22 → 38979 [ACK]
1359	17:47:21.405366	38979	22	2800	64	2646	2156054	19200	660800	Unknown (84) [Unr
1360	17:47:21.405377	38979	22	2800	64	2646	2158854	19200	663600	Unknown (210) [Un
1361	17:47:21.417167	22	38979	0	46	1498054	2646	719872		22 → 38979 [ACK]

还是 1337 到 1357 号报文，我们去掉了过滤器 `tcp.srcport eq 38979`，这样就展示了双向报文。可以看到，服务端（B 端）在这段时间内，只确认了 22400 字节（1495254 - 1472854），而同样时刻的在途数据，却一直维持在一个比较高的数字，在 660KB 上下。所以，**真正完成了传输的数据量，是前者 22400B，而不是“虚浮”的 660KB**。

那你可能又要问了：既然已经确认了 22400 字节，为什么客户端的在途字节数还是没有变化呢？

这是因为，客户端被确认了 22400 字节的数据，马上又把这个尺寸的数据发送出去了，事实上就**维持了这个在途字节数的尺寸**。

我可以再做一个比喻帮助你理解这个现象。我们如果去银行的一个窗口（这可不是 TCP 窗口）排队办业务，现在排队人数为 10 人，相当于 Bytes in flight 为 10。每分钟都有一个人能完成业务办理，原以为队列会减小为 9 人，结果每当有一个人出来，保安就喊：“下一个！”于是就立刻又补进来一个人，所以队伍还是维持在 10 人这么长。

那么，窗口的业务员的办理速度是多少呢？显然不是 10 人 / 分钟，而是 1 人 / 分钟了。这样是不是理解起来容易多了？而上节课的情况，相当于这里的“每次就处理一个人”，所以处理速度就是 1 人 / 分钟，也就可以用“速度 = 窗口 / 往返时间”来计算了。

小结

这节课，我们集中讨论了 TCP 传输中的窗口相关的知识，特别是围绕 TCP Window Full 这个 Wiresahrk 中比较常见的信息，展开了深入的讨论。你需要重点掌握以下这些知识点：

Wireshark 报告 TCP Window Full 是因为，一端的在途数据跟另一端的接收窗口相等。

TCP 的下个序列号（Next Sequence Number）等于序列号和段长度之和，即 **NextSeq = Seq + Len**。

我们可以自己人工计算出在途字节数，公式是 **Bytes_in_flight = latest_nextSeq - latest_ack_from_receiver**。

人工计算在途字节数的方法是：

- 找到最近一次发送出去的报文的 NextSeq，记为 X；
- 找到在这次发送之前收到的最近的 ACK，记录它的 ACK，记为 Y；
- X - Y，得到在途字节数。

另外我们也知道了，TCP Window Full 确实会影响到传输速度。

除了技术知识点，我也带你学习了下面这些 Wireshark 工具使用技巧：

在 Statistics 下拉菜单下的 I/O Graph 工具，可以直观地展示传输速度图。

同是 Statistics 菜单下的 TCP Stream Graphs 的 Window Scaling 工具，可以直观的展示 TCP Window Full 历史曲线图。

最后，我们还发现这个案例里的接收端有“数据滞留”的现象，这就导致“速度 = 窗口 / 往返时间”的公式遇到了挑战，而我们可以进一步优化为新的公式：**速度 = 确认数据 / 往返时间**。这个改进后的公式，可以兼容这种有“数据滞留”现象的传输场景。

思考题

给你留两道思考题：

TCP 的序列号和确认号，最大可以到多少？

接收端只确认部分数据，导致了“数据滞留”现象，这个现象背后的原因可能是什么呢？

欢迎你把答案分享到留言区，我们一同交流和进步。

附录

抓包示例文件：https://gitee.com/steelvictor/network-analysis/tree/master/10

赞 0 提建议

上一篇 09 | 长肥管道：为何文件传输速度这么慢？

精选留言 (5)

写留言

江山如画
2022-02-11

问题1:
序列号和确认号都占用 32bit，空间范围从 [0, 2**32-1], 最大是 2**32-1

问题2:
感觉接收端确认的这部分数据，就是用户进程从内核接收缓冲区中读取的数据。出现数...
展开

2

那时刻
2022-02-11

第一题：tcp头的序列号是32位，所以最大值是2^32。 如果短时间内发送大量数据，会有序列号回绕的问题。
第二题，接收端只确认了部分数据，可能接收程序处理慢，没有及时响应网卡缓冲区的irq，接收了部分数据。抑或网卡多队列导致接收不及时。
展开

作者回复: 恩差不多。第一题确切说是2^32-1 ：)

1

Realm
2022-02-11

1. 默认最大2的32次方，好像也有一些机制，比如使用timestaps，用来防止序列号环绕，是不是一定程度上也突破了这个最大限制？
2 接收端，用户程序处理速度过慢，或者网卡出现故障。

展开

Chao

2022-02-11

2 ** 32 -1

作者回复: 是的~

我来也

2022-02-11

从老师的文章中学了不少工具的使用方法，以前只会傻傻的看每个报文。马上就把它给用起来，😁

作者回复: 哈哈 好的 多利用工具，效率继续提升