09 | 长肥管道：为何文件传输速度这么慢？

2022-02-09 杨胜辉

《网络排查案例课》 课程介绍 >

讲述：杨胜辉
时长 26:57 大小 24.68M

你好，我是胜辉。

在上节课里，我们通过一个 MTU 引发问题的案例，学习了 MTU 相关的知识点，了解了 MTU、MSS、TCP 分段、IP 分片这些概念之间的区别和联系。我们也学习了用 iptables 修改 MSS 来达到规避 MTU 问题的目的，是不是觉得网络虽然学起来不简单，但学会了好像也挺好玩的？

那这个感觉就对了，学习本来就是一件有意思的事情，也是很有收获感的事情。学得更多，收益更多，反过来就能推动自己继续学得更多，形成良性循环。

当然，上节课我们主要说的是如果传输失败会怎么样，而从这节课开始，我们会讨论讨论传输起来以后的效率问题，比如传输速度。

传输速度是网络性能中非常重要的一部分内容，因为它直接影响了我们享受到的网络服务的品质。比如现在移动端网络越来越快，所以无论传输大文件还是实时的视频通话，都得到了很大的发展。

数据中心更是如此。服务器的网卡早就从多年前的千兆网卡（1Gbps）升级到万兆网卡（10Gbps）。而负载均衡、防火墙等网络设备，更是从 10Gbps 升级到 40Gbps，乃至 100Gbps。

在这个大背景下，照理说在数据中心之间传个文件早已不在话下。但是我们偏偏遇到了这样一个奇葩的问题：**数据中心间传输文件的速度居然只有 400 多 KB/s**，是不是很奇怪？你有没有遇到过类似的传输速度方面的问题，你又是如何通过网络分析，找到根因的呢？

所以这节课，我们就来研究分析下这个文件传输速度的案例，一起来深入探讨下如何提升 TCP 传输速度的话题。

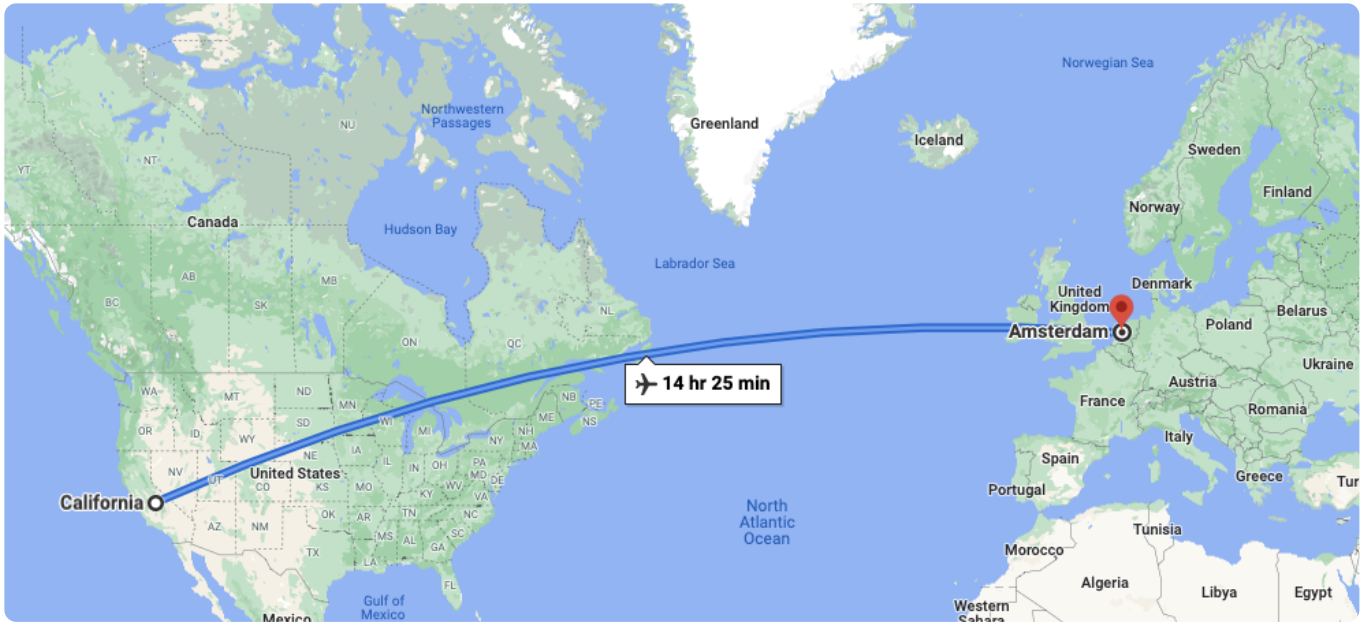
这样，当你日后也遇到传输速度方面的问题时，就可以运用这节课学到的知识，去真正解决问题。甚至，即使看起来没有速度问题，你也能主动发现提高网络性能的机会，带来额外的惊喜，同时也把你的网络维护的能力提升到一个更高的层次上来。

案例背景

这是 2017 年我处理的 eBay 内部的案例。当时我们有一个需求是把一个 Firmware 安装文件从美国数据中心拷贝到欧洲的一个 POP 点，这个点在荷兰的阿姆斯特丹。文件不算大，95MB。其实这是一项常规任务，因为我们每年都会做一到两次的 Firmware 升级，经常在美国几个数据中心之间拷贝安装文件，一般这样一个 100MB 左右的文件约二十多秒就能完成。

这一次，我们需要把全球 POP 节点的设备也进行升级。对我们来说，确实也是第一次做从北美到欧洲这么长距离的文件传输，结果发现传输速度慢了一个数量级。我们的 POP 点有很多个，按顺序做升级的话，每次文件传输都增加 3 分钟，那么 20 个 POP 就增加 60 分钟，显著拖累了整个的升级速度。

当然，地理上看，从美国加州到荷兰阿姆斯特丹确实极为遥远，距离数千公里，坐飞机也要 14 个多小时：



然后我们也来看一下美国不同的数据中心之间拷贝的速度：

bigfile.gz	100%	95MB	4.0MB/s	00:24

极客时间

再看一下从美国拷贝到荷兰的速度：

bigfile.gz	100%	95MB	450KB/s	03:31

极客时间

当你看到两种场景下传输同一个文件的速度有这么大的差别，第一感觉是什么呢？你觉得会是什么因素导致了速度变这么慢？会不会想到带宽？

确实，当时我们也才开始维护欧洲 POP 点，对美国和欧洲之间的底层传输网络情况并不熟悉，所以也怀疑：**会不会是带宽本身就不大呢？**我们找了负责骨干网的同事，了解到这个带宽有 10Gbps，远超这个传输速度。

然后，你会想到什么呢？对了，我补充一下：这个传输是 scp 拷贝，所以是基于 TCP 的。你自然明白，TCP 的传输速度也受丢包的影响。如果丢包严重，传输速度肯定跑不起来。

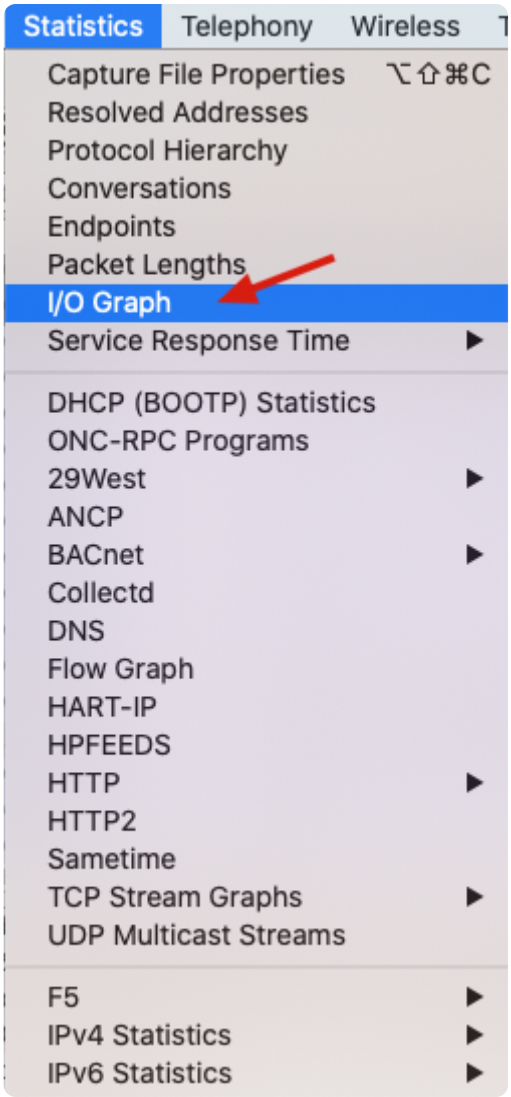
我们觉得这个也是有可能的，就去抓包检查。结果发现丢包率并不高，不高的丢包率却导致如此低的速度，这个可能性很小。

更为重要的是，**既然我们要排查传输速度的问题，那么首先要看的，是不是网络 I/O 的状况呢？**

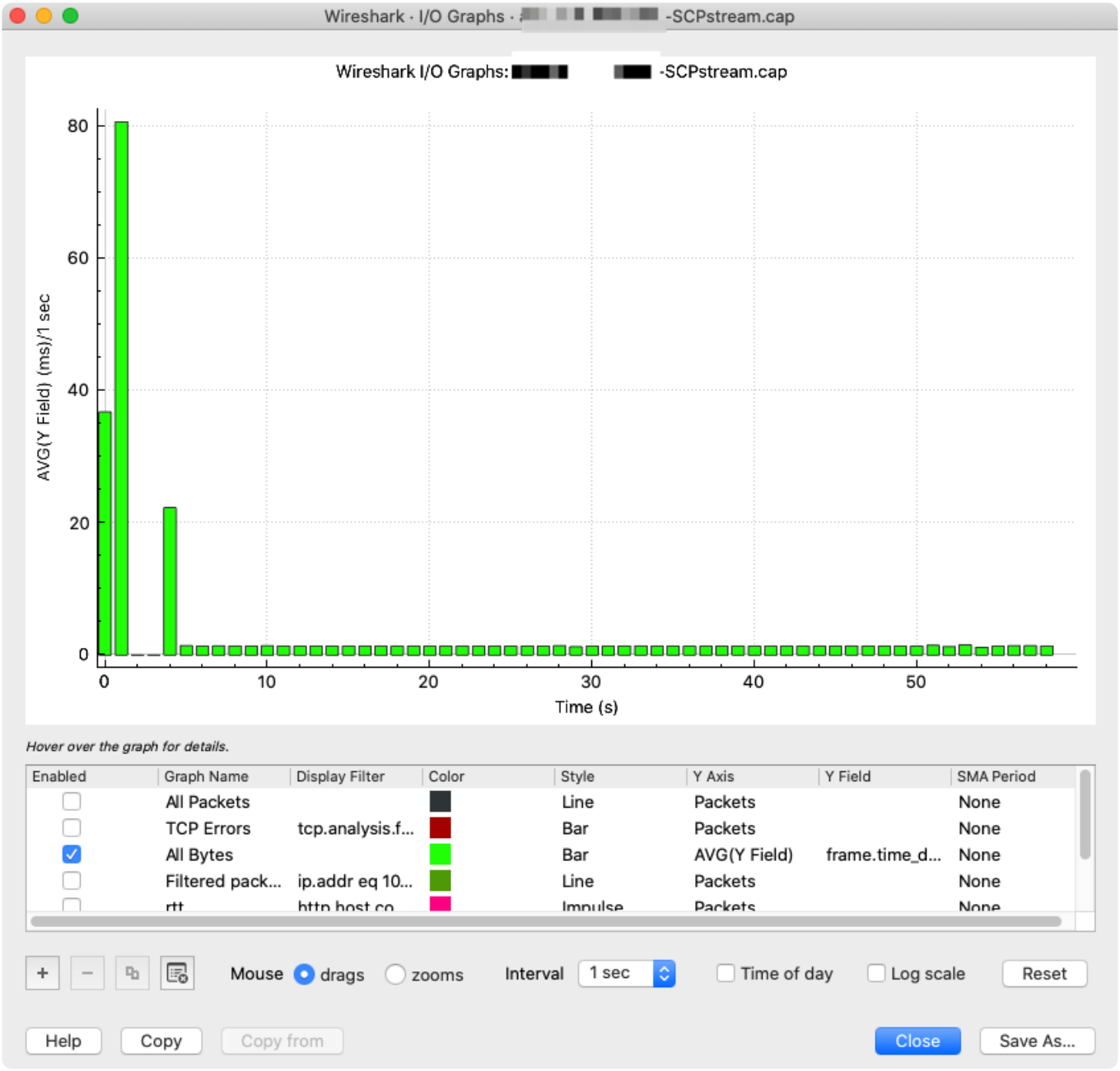
在 Wireshark 的众多统计工具中，有两个工具就是 I/O 分析而生的，一个是 **I/O Graph**，另一个是 **TCP Stream Graph**。我们先看 I/O Graph。

I/O Graph

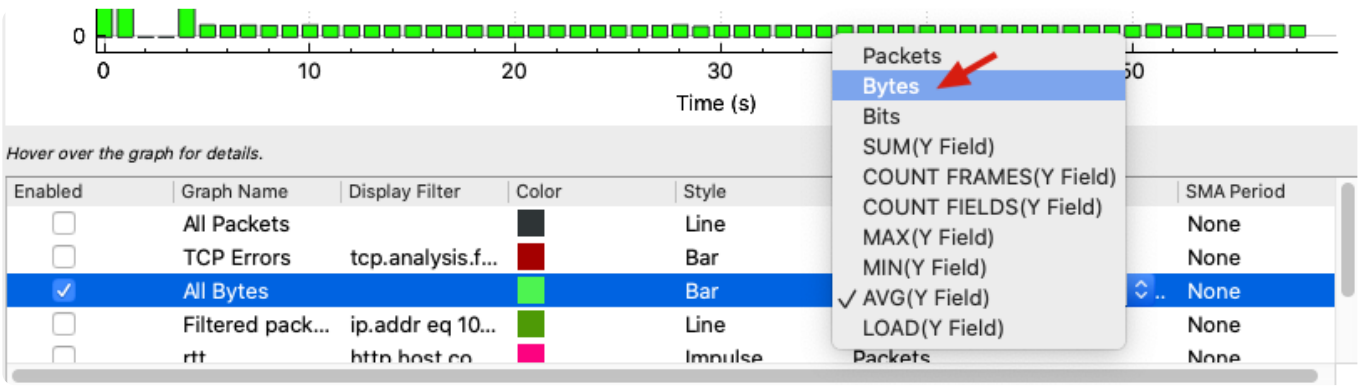
这个工具在 Statistics 下拉菜单中：



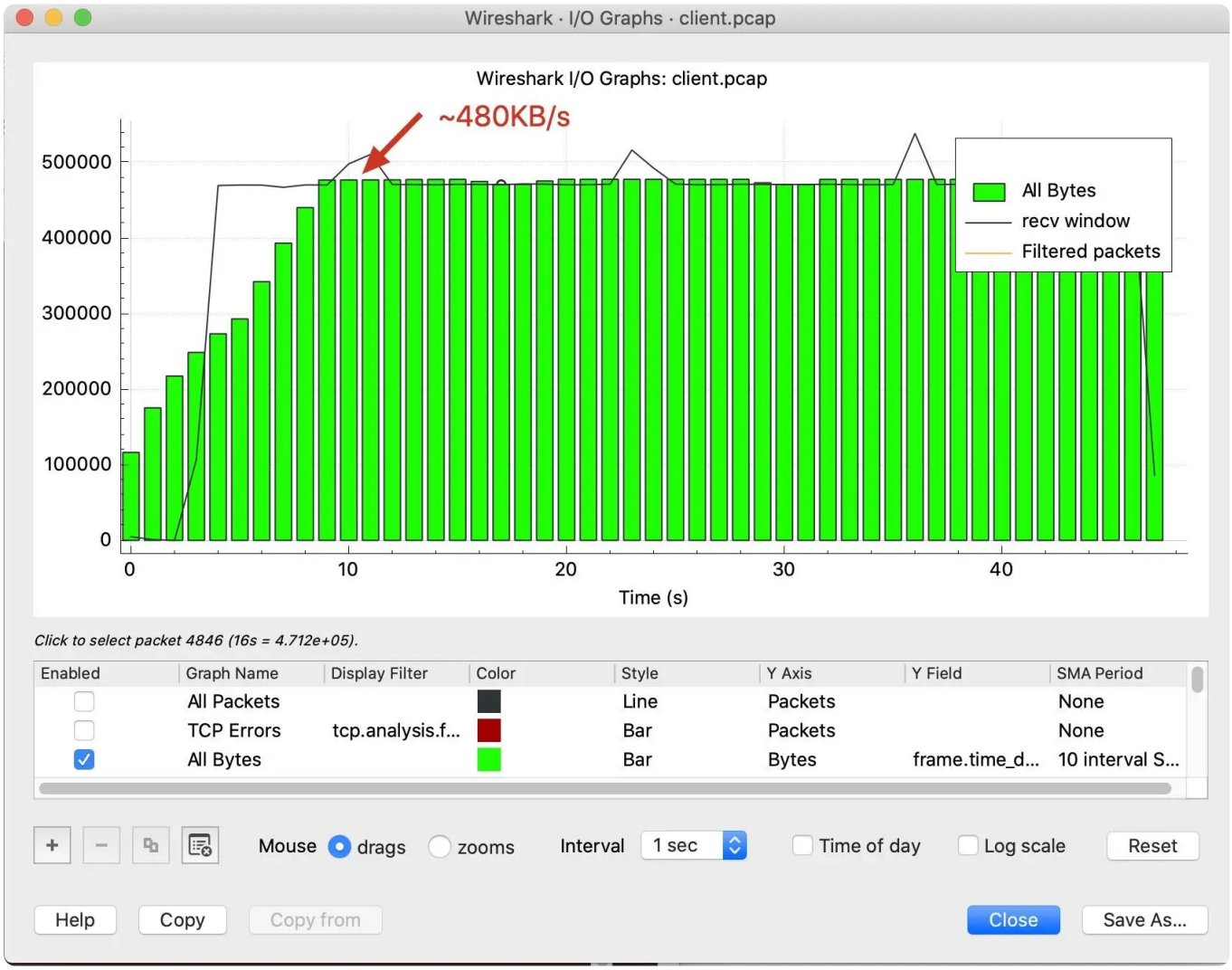
点击 I/O Graph 后，Wireshark 会弹出一个趋势图，其 X 轴是时间，Y 轴是性能指标：



注意，这时的图是不准确的，所以我们需要做一些调整。既然我们关注传输速度，所以就选择 All Bytes 这个指标项（也是默认选中的那项）作为 Y 轴，然后修改它的计量单位。很可能你的 Wireshark 默认选中的是 AVG(Y Field)，而这并不是我们要关注的**字节数**。我们可以双击 AVG(Y Field) 进入编辑模式，把它改为 **Bytes**：



很棒！这时我们就能清晰地看到，Wireshark 帮助我们计算出来的分时的速度趋势柱状图了，差不多速度在 480KB/s 上下：



你可能也注意到了，在 X 时间轴上看，一开始几秒速度比较低，第 7 秒才达到 400KB/s 以上。为什么一开始速度这么低呢？其实，这正是 TCP 慢启动的一个缩影：**初始阶段，速度是特别低的，但是会很快爬高。**

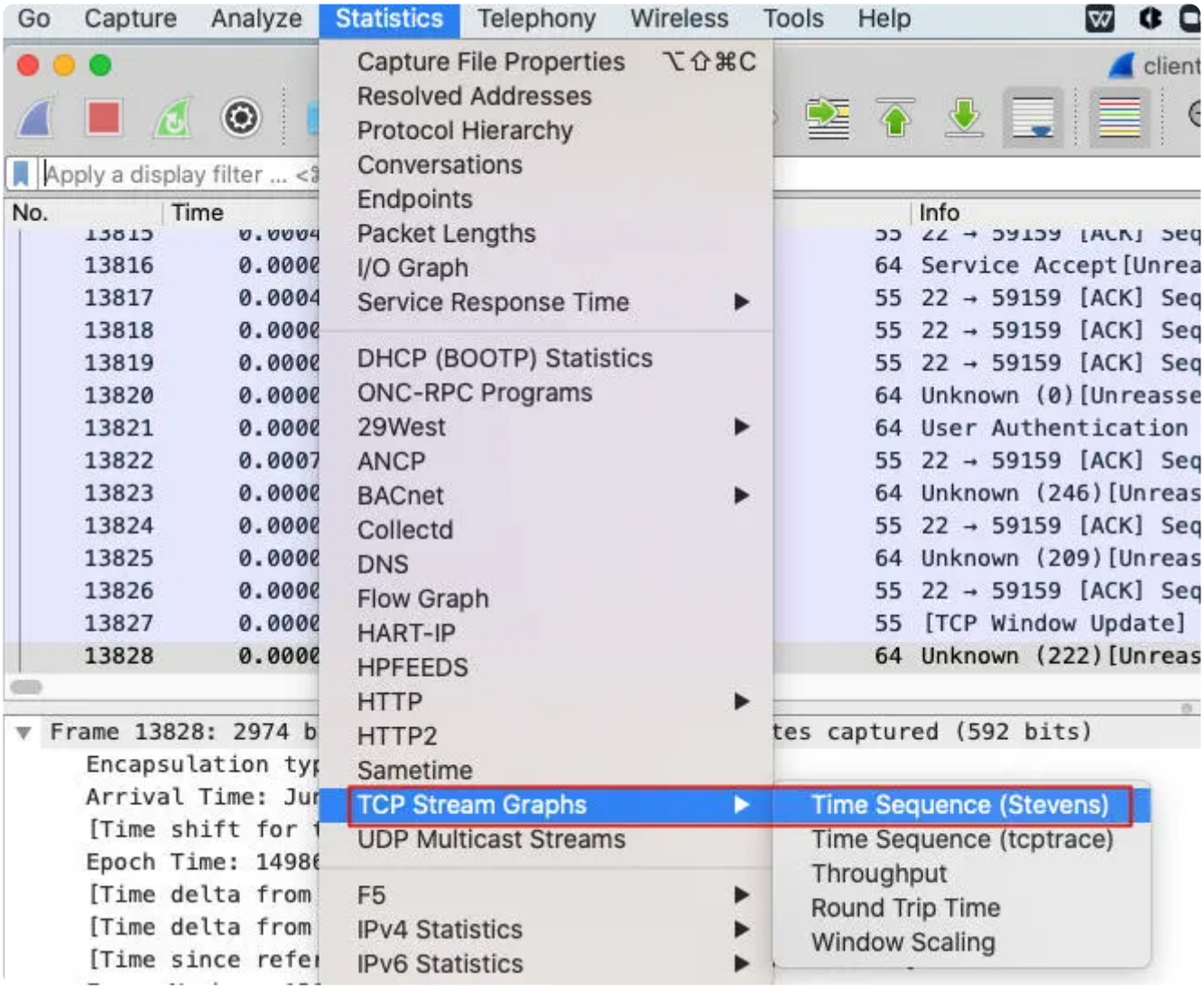
整体来看，这个传输速度还是很“稳定”的。对，一直很“稳定”得低。

TCP Stream Graphs

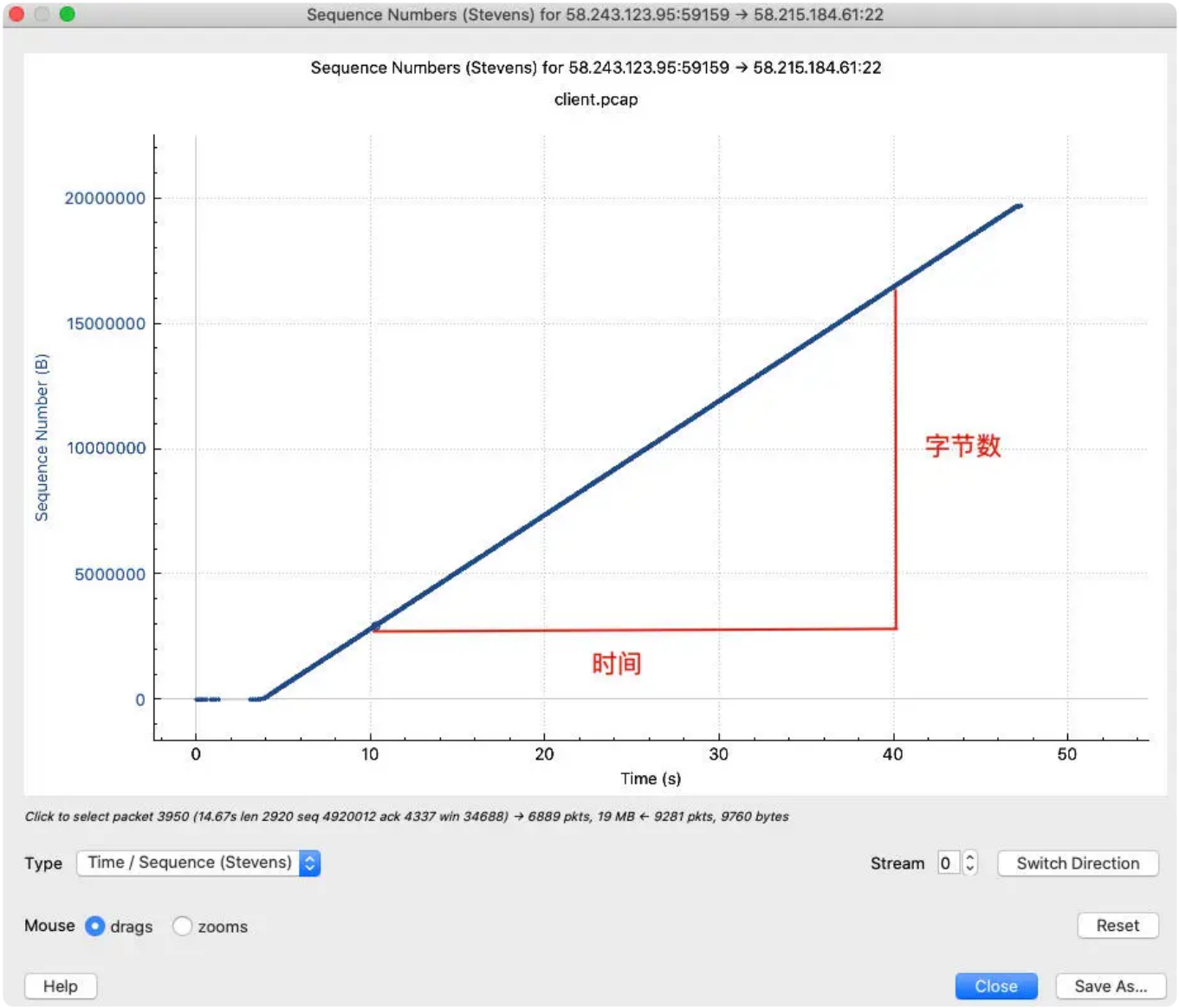
如果说 I/O Graph 展示的速度值很容易理解，那么 TCP Stream Graphs 展示的信息就需要一点 TCP 的知识来辅助理解了。

还是到 Statics 下拉菜单，选择 TCP Stream Graphs，在子菜单中选择 Time sequence (Stevens)。

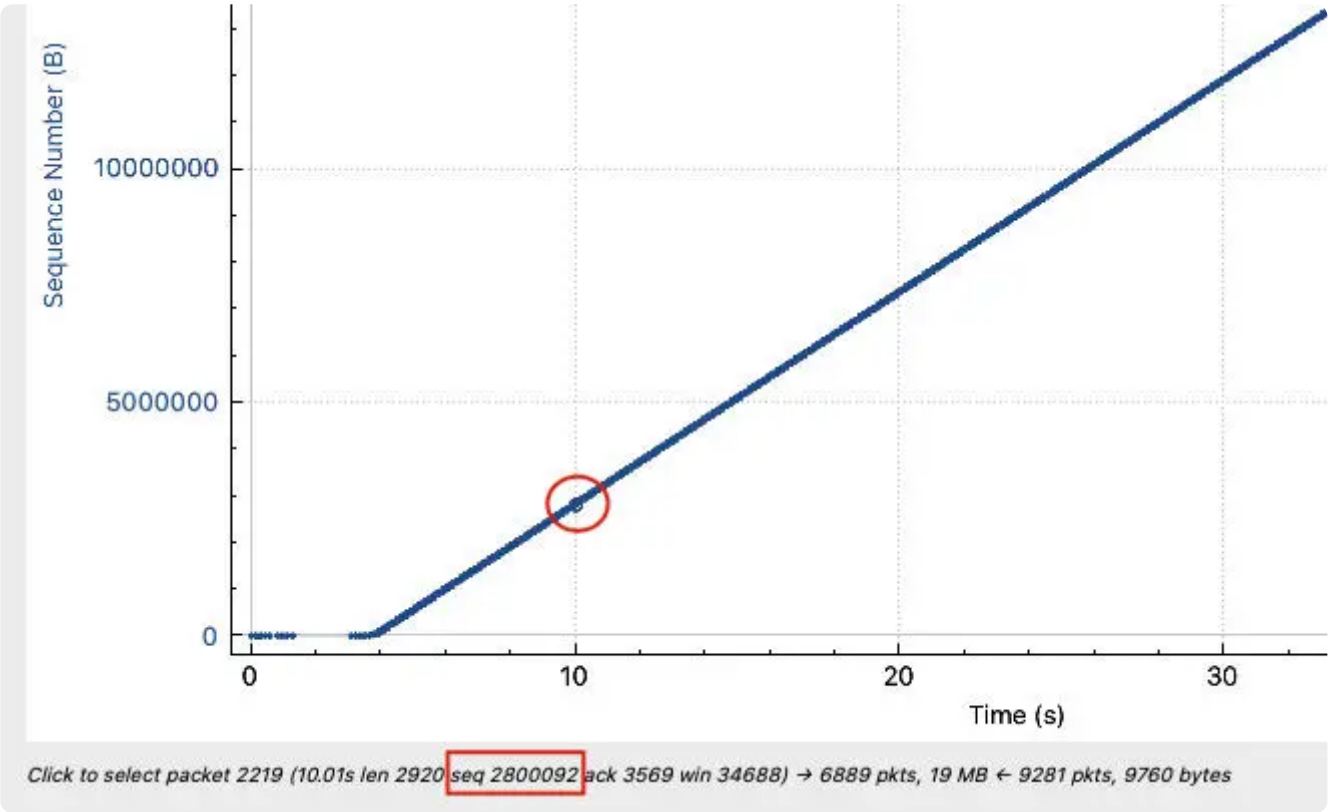
> 补充：Stevens 这个名字你应该是熟悉的，他就是鼎鼎大名的 TCP/IP 三卷和 Unix 网络编程两卷等名著的作者，Richard Stevens。他把网络协议的圣火带到人间，却又早早离开了人间。这个工具以他的名字命名，也代表了大家对他的深深的怀念。

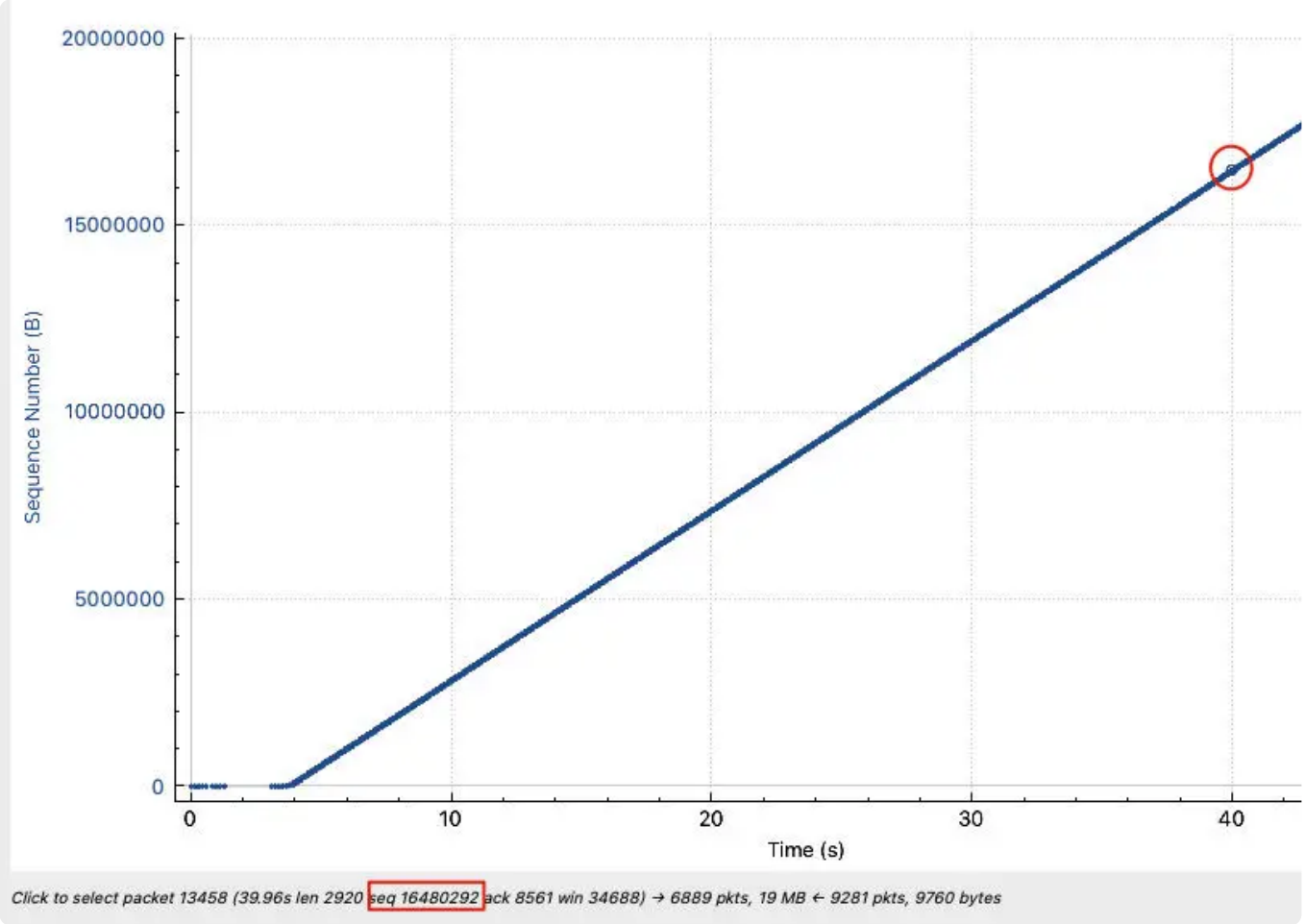


然后就能看到时间为 X 轴、TCP 序列号为 Y 轴的图了。你应该知道，序列号其实就等于字节数，那么显然，这里这条线的斜率也就是传输速率了。



我们可以自己计算这个斜率（速率）是多少。比如可以计算 10 秒和 40 秒两处的序列号的差距，再除以 (40-10) 秒，就是速率了。10 秒处的序列号是 2800092，40 秒处是 16480292，那么速率就是 (16480292-2800092)/(40-10)=456KB/s。





你可能发现了，**两个 Graph 算出来的速度怎么有点差异？**一个是 480KB/s 左右，一个是 456KB/s，相差有 5% 左右。

其实，这是正常的。因为 I/O Graph 统计的 Bytes 是二层帧的大小，而 TCP Stream Graphs 关注的是四层 TCP 段的大小。后者比前者少了二层到四层的头部。严格来说，TCP Stream Graphs 的斜率，只是 TCP payload 的速率；而 I/O Graph 展示的，才是我们一般谈论的传输速度。当然，在定性的讨论中，这点差异是可以忽略的。

理解与传输相关的知识点

其实，上面用了两种方式查证速度，也无非是印证了我们在 scp 命令输出里，看到的速度值本身是准确的。但目前为止，我们还没有找到造成这个速度的原因。这时候，我们稍微调整一下问题描述：**假如带宽足够，网络也稳定，那又是什么决定了 TCP 传输的速度？**

回想一个小学时就学过的公式：**速度 = 距离 / 时间**。

看上去极为简单的公式，似乎在这里没什么帮助。不过奥妙的地方在于：什么是距离，什么又是时间？搞清楚了这两个问题的答案，就搞清楚了这次传输分析的精髓。不过我这里先卖个关子，先继续把接下来的知识点讲清楚，到后面你也许自己就能悟到答案了。

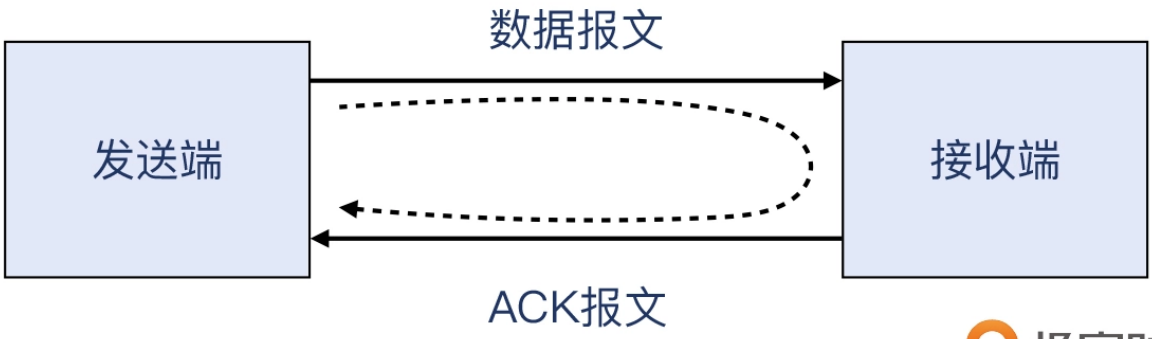
假设你在开车，开出去 100 公里，耗时 1 小时。这是因为你眼观四路、耳听八方，在确保安全的前提下用比较快的速度行驶。如果你蒙上眼睛，那是 1 米都不敢开的，是不是？

对于网络传输来说，报文发送出去后，发送端本身就“失去视力”了，也就是看不到报文在错综复杂的网络中行走时，到底遇到了什么样的“路况”。那它是继续发送更多数据好呢，还是先等对方确认这部分数据，然后再发送下一份数据好呢？如果确认了，那下一份数据又该发多少呢？一系列问题，需要我们回答。

不过还好，睿智的 TCP 的设计者们早就考虑到了这些问题。这里呢，有几个相关知识点你需要了解，且听我细细道来。

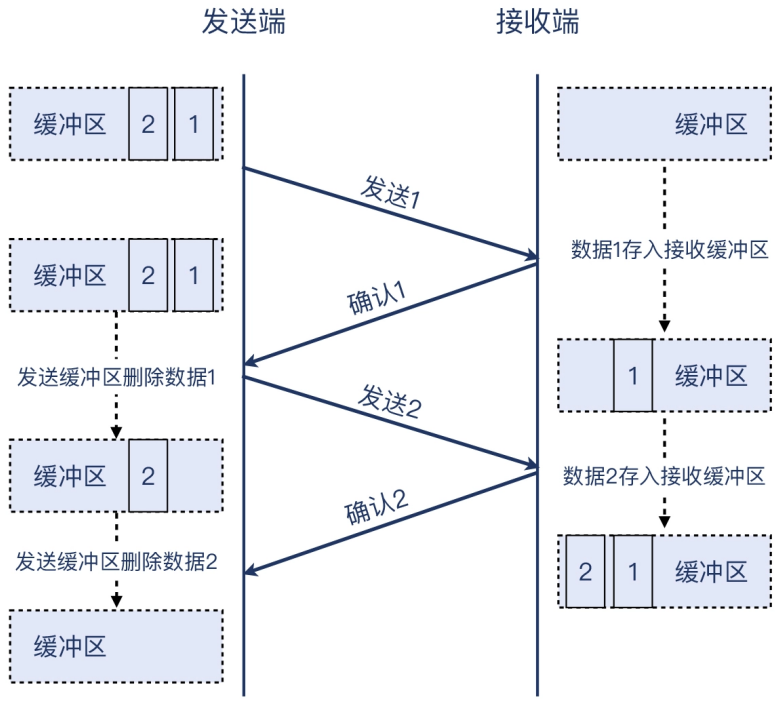
首先，一个报文被视为成功发送，基于以下的路径：

（开始）发送端 >> 数据报文 >> 接收端 >> ACK 报文 >> 发送端（结束）



这样一来一回的时间，就是报文被成功传送的耗时。你可能会有疑问：数据报文到了接收端不就是已经完成传输了吗？其实，这个时候发送端还不知道接收端是否已经完成接收了，因为还没收到确认。

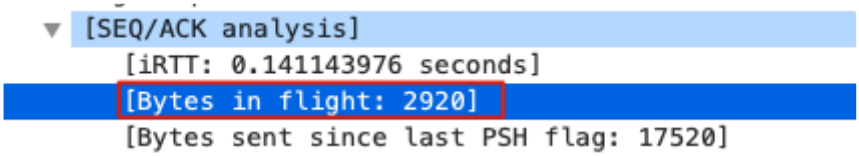
从**操作系统的角度**来看：“完成”的标志，是这部分数据可以从发送缓存中删除；从缓存中删除这部分数据的前提，就是收到 ACK 报文。即“确认多少，我就清除多少”。像下图这样：



来回的时间，在英文里叫 **Round Trip Time**（RTT），即往返时间，也叫**时延**。这是你需要知道的第一个名词。

你有没有发现，报文大部分时候就像飞机在空中航行，跟两端的哪一端都碰不着，除了起飞和降落。在“空中”的时间，就取决于 RTT。RTT 越长，报文在“空中”的时间就越长。这个时候，这些报文就有了一个新的身份，叫做“在途数据”，它的大小，叫做**在途字节数**。英文是 **Bytes in flight**。这是你需要知道的第二个名词。

事实上，Wireshark 也很周到地帮我们想到了这一点，在 TCP 详情页的 SEQ/ACK analysis 部分就有 **Bytes in flight** 信息。比如下面这个：

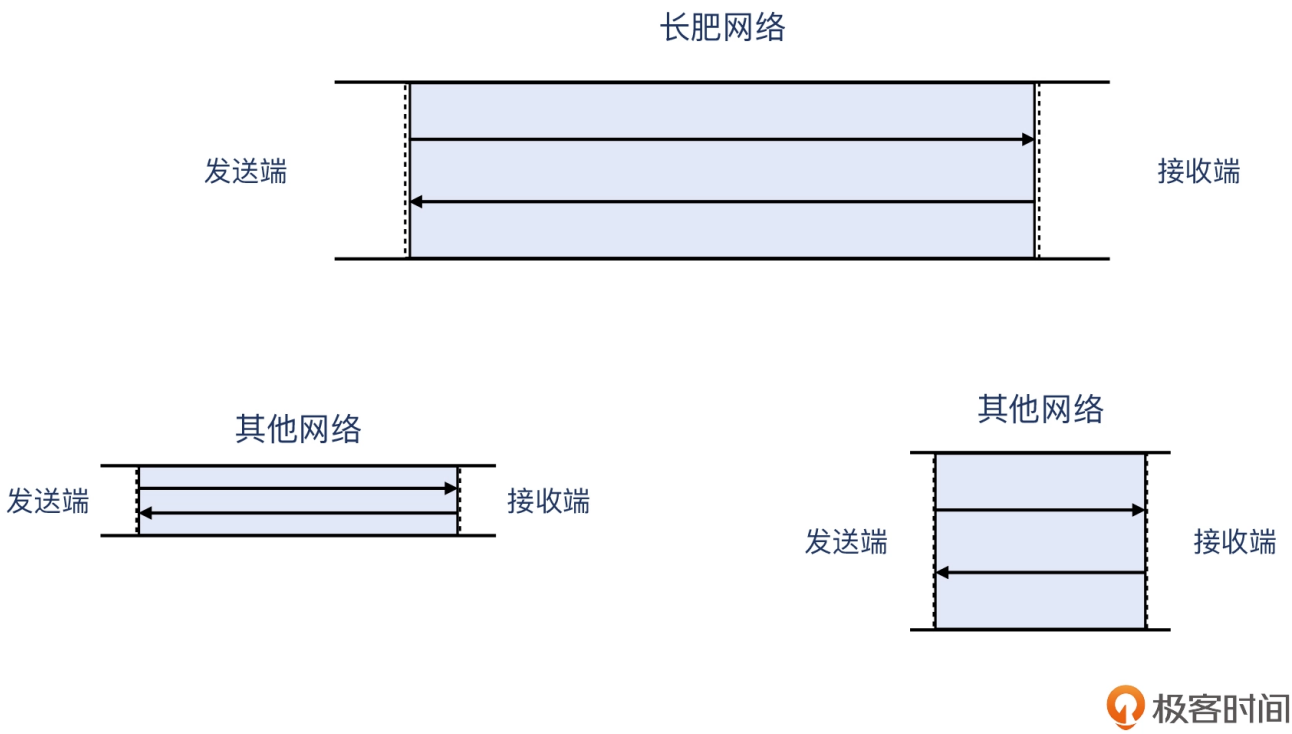


带宽，类似道路的车道数，你应该见过节假日里高速公路上停满私家车的图片。可见，车道越多，高速路越长，能容纳的车就越多。在网络世界里，带宽很大、RTT 很长的网络，被冠以一个特定的名词，叫做**长肥网络**，英文是 **Long Fat Network**。在长肥网络中的 TCP 连接，叫做**长肥管道**，英文是 **Long Fat Pipeline**。这是你需要知道的第三个名词。

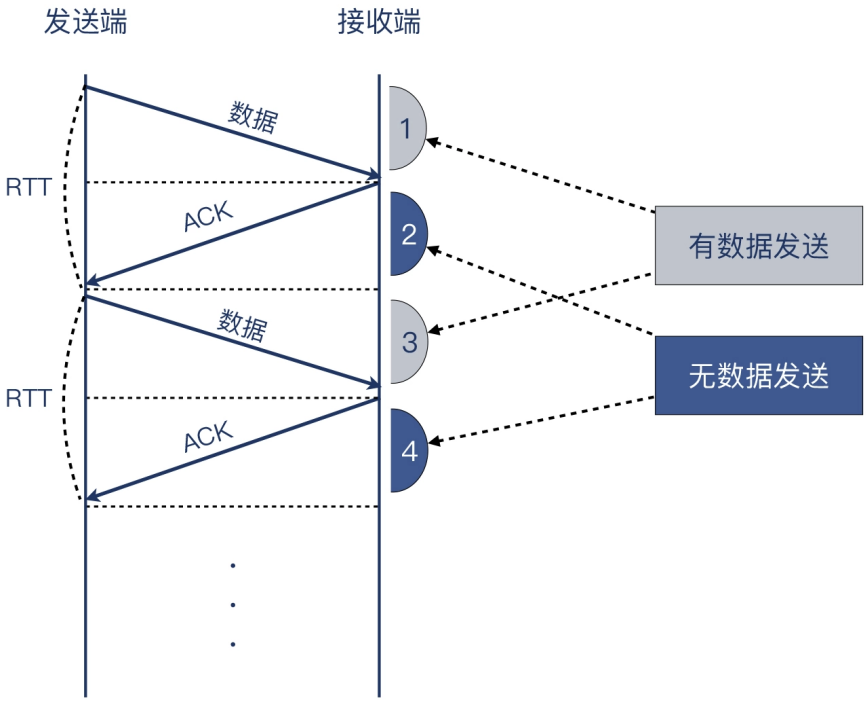
道路上最多可以容纳多少辆车呢？显然，是车道数×道路长度。那么类似地，带宽跟往返时间（RTT）相乘，就是在空中飞行的报文的最大数量，即**带宽时延积**。在英文里叫 **Bandwidth Delay Product**，缩写是 BDP（Delay 就是 RTT）。不用我说，你也明白带宽时延积是你需要知道的第四个名词。

其实，以上这些名词都是从英文翻译过来的，所以读起来感觉不太像本土词汇，特别是“长肥管道”。你别把“长肥”记成“肥肠”就行了。

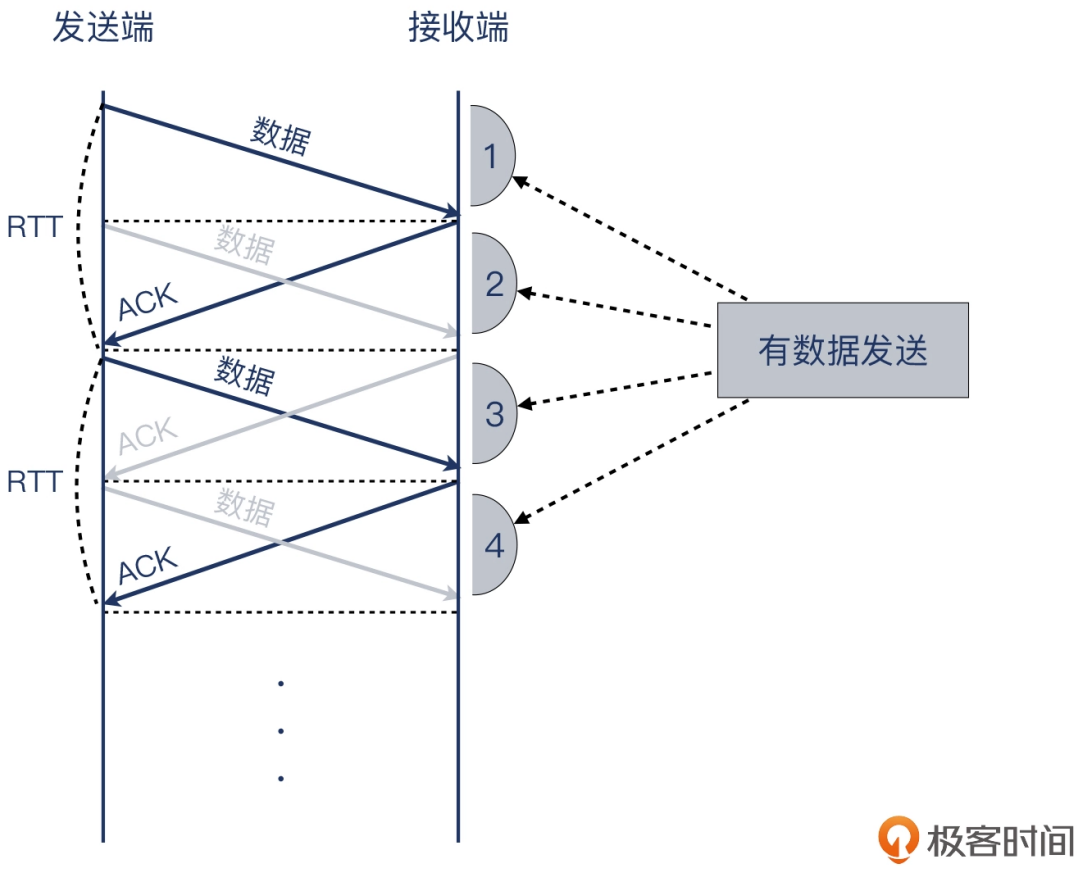
我们用图对比一下，就能明显看出，所谓的长肥管道，就是带宽时延积更大的网络，蓝色部分表示的就是带宽时延积：



你可能还会有疑问，在途数据不是应该 RTT 的一半（即单程时间）再乘以带宽吗？从飞机的比喻来看，也许是的。但是你要考虑下图这种情况。如果回程时间只是用来传输 ACK，没有被用来传输实际的数据，效率就打了对折了。就像图的右侧表示的，只有一半的时间在真正传输数据，另外一半时间没有在发送数据：



事实上，发送端并不知道什么时候报文到了接收端，它唯一知道的是什么时候自己收到了 ACK 报文，所以它会把这些时间全部用来发送报文。我把上图中原先不传数据的时间段 2 和 4 也改为发送数据，于是形成了下面这张图：



当然，**实际情况是远比示意图复杂的**。比如，在途字节数一定等于带宽×RTT吗？要知道，接收端也不傻，也是可以源源不断地对数据进行确认的，这样也就形成了更加复杂的关系：发送端在不断发出在途数据，接收端也在不断回复 ACK。这时候的在途数据，可以用下面的公式来获得：

> **inflight_data = (latest_sequence_sent + latest_len_sent) - max_ack_received**

最后还有个重要的概念是“窗口”。有人开玩笑说：讨论 TCP 握手的时候，大家侃侃而谈；一旦说到窗口，就都沉默了，气氛尴尬。继续往下学习这一部分，你会成为那个打破沉默的人。：）

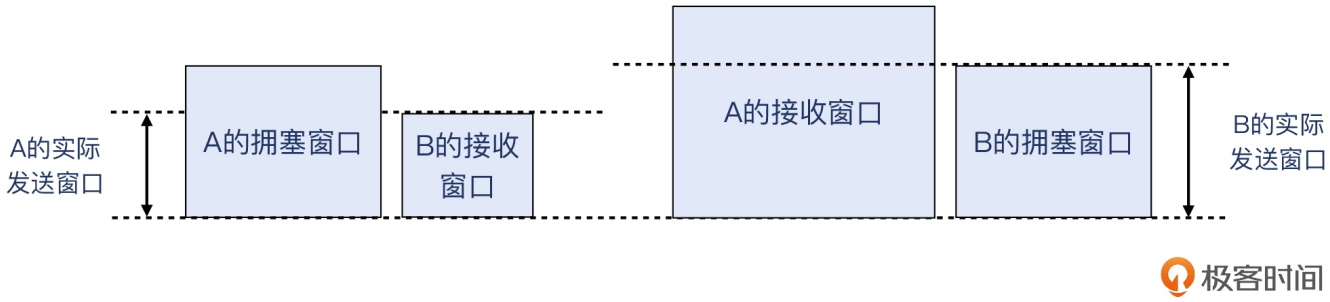
其实，下次别人跟你切磋“窗口技术”的时候，可以先下手为强：“你说的是到底哪个窗口？”因为事实上，TCP 有 3 个窗口：接收窗口、拥塞窗口，还有发送窗口。

接收窗口：它代表的是接收端当前最多能接收的字节数。通过 TCP 报文头部的 Window 字段，通信双方能互相了解到对方的接收窗口。

拥塞窗口：发送端根据实际传输的拥塞情况计算出来的可发送字节数，但不公开在报文中。各自暗地里各维护各的，互相不知道，也不需要知道。

发送窗口：对方的接收窗口和自身的拥塞窗口两者中，值较小者。实际发送的在途字节数不会大于这个值。

接收窗口是明的，在抓包文件里就能看到；拥塞窗口和发送窗口是暗的，抓包文件里没有。我们看这张图就明白了：



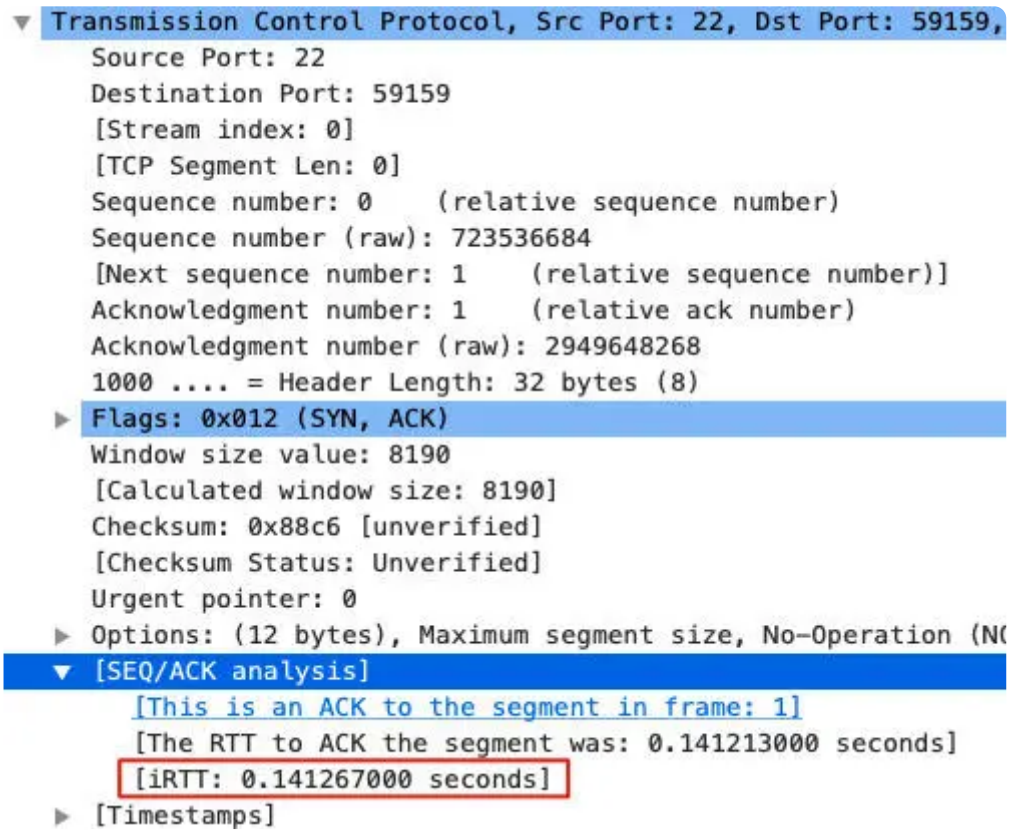
解题

好了，终于把传输相关的关键知识点介绍完了，是不是感觉信息量有点大，甚至还没完全理解？别急，下面我们把这些知识点应用到这个案例中，这些看似干涩的知识点就会逐步丰润起来。

时延

我们首先收集时延信息。用美国数据中心的发送端去 Ping 阿姆斯特丹接收端的 IP，我们发现**时延在 134ms** 左右。用 Ping 的原因是，它的 ICMP 报文比较轻量，不会引起双方很多的额外处理时间，所以适合用来获取相对纯粹的网络往返时间。

在 TCP 通信中，因为协议栈本身也需要做拆解包、缓冲、Socket 处理等工作，所以 TCP 层面的 RTT 会比 IP 层面的 RTT 略长一点。好在，Wireshark 也提供了 RTT 信息。我们选取一个报文，在 TCP 详情的 SEQ/ACK analysis 部分，就有 iRTT 信息。这里是 141ms，比 Ping 探测到的 134ms 多了一点，这是因为 TCP 协议栈本身处理也有一些延迟：



iRTT 是 intial RTT 的缩写，Wireshark 就是从 TCP 握手阶段的报文里计算出这个值的。对于客户端来说，就是发出 SYN 和收到 SYN+ACK 的间隔。对于服务端，就是发出 SYN+ACK 和收到 ACK 的间隔。

带宽时延积

接下来我们算算带宽时延积是多少。时延是 134ms，带宽是 10Gbps，那么带宽时延积就是 0.134×10Gb。转换为 Byte 需要再除以 8，得到约 168MB。这个数值的意思是，假设这条链路完全被这次的文件传输连接所占满，那么最多可以有 168MB 的在途数据。

我们这个 Firmware 文件也才 95MB，如果按这个速度，那一个来回就传完了，也就是只需要 134ms！当然，TCP 有慢启动机制，不可能一开始就把一百多 MB 这么大的数字作为初始拥塞窗口，所以也不会真的一个来回就传完了。

到这一步，我们已经明白，**带宽时延积并不是限制速度的因素**。那么一定是别的东西，在暗地里约束着这次传输。这个东西是什么呢？我们快接近真相了。

发送窗口

在 Wireshark 里查看接收窗口的数值也是很直观的。任意一个 TCP 报文的头部都有 Window 字段，长度为 2 个字节，所以最大值为 2 的 16 次方，即 64KB。在第 3 讲我

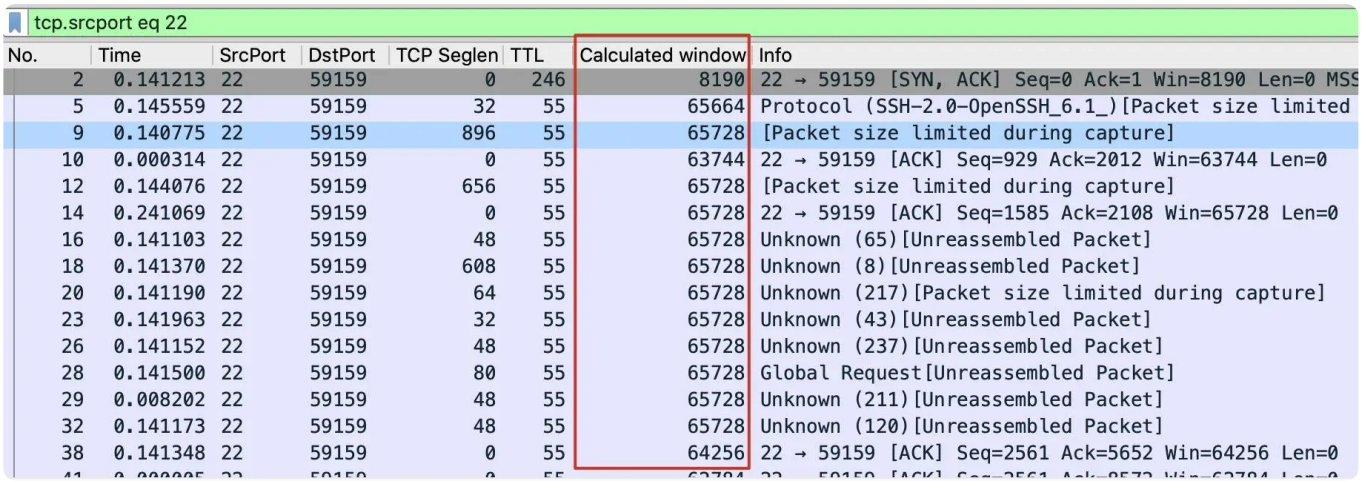
们讨论 TCP 握手相关案例时，提到过 **Window Scale**。它是在RFC1323中引进的，使得 Window 值最大能达到 2 的 30 次方，即 1GB。

我们看一下当时阿姆斯特丹 POP 点作为接收端，它宣告给美国数据中心的 TCP 接收窗口值是多少。为了方便查看，你可以在 Wireshark 里这样做：

添加 Calculated window size 为自定义列。操作方法是任选一个 TCP 报文，在 TCP 详情那里找到[Calculated window size:xxxx]，右单击，在弹出菜单中点击 Apply as Column。然后主窗口里就多了这样一列 Calculated window size。

在过滤器输入框输入 tcp.srcport eq 22，这样就把 POP 点（监听在 22 端口）发出的报文过滤出来了，更加方便我们查看。

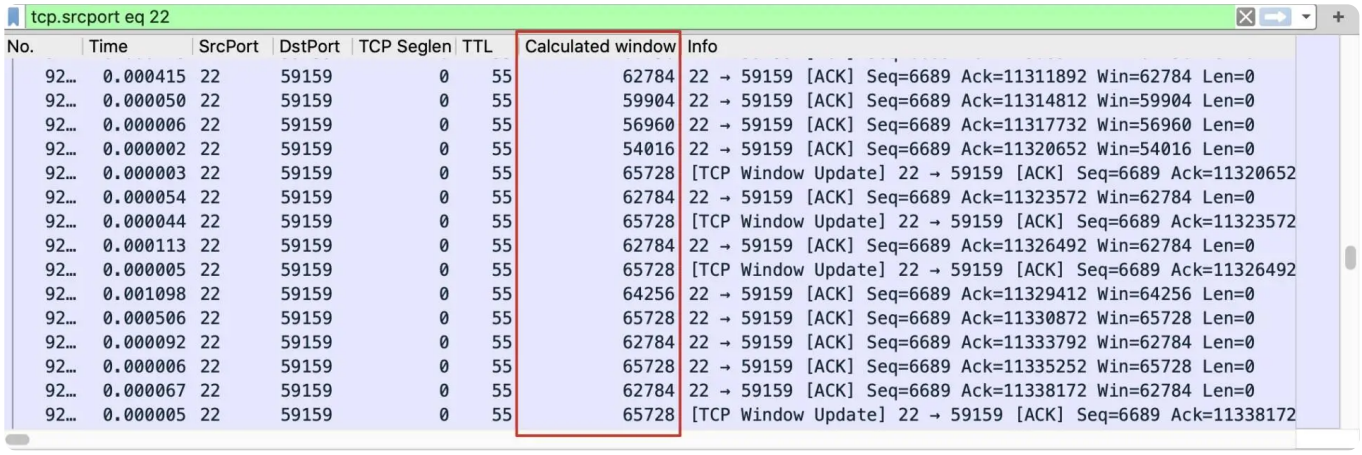
我得到的界面是这样的：



No.	Time	SrcPort	DstPort	TCP Seglen	TTL	Calculated window	Info
2	0.141213	22	59159	0	246	8190	22 → 59159 [SYN, ACK] Seq=0 Ack=1 Win=8190 Len=0 MSS
5	0.145559	22	59159	32	55	65664	Protocol (SSH-2.0-OpenSSH_6.1_)[Packet size limited
9	0.140775	22	59159	896	55	65728	[Packet size limited during capture]
10	0.000314	22	59159	0	55	63744	22 → 59159 [ACK] Seq=929 Ack=2012 Win=63744 Len=0
12	0.144076	22	59159	656	55	65728	[Packet size limited during capture]
14	0.241069	22	59159	0	55	65728	22 → 59159 [ACK] Seq=1585 Ack=2108 Win=65728 Len=0
16	0.141103	22	59159	48	55	65728	Unknown (65)[Unreassembled Packet]
18	0.141370	22	59159	608	55	65728	Unknown (8)[Unreassembled Packet]
20	0.141190	22	59159	64	55	65728	Unknown (217)[Packet size limited during capture]
23	0.141963	22	59159	32	55	65728	Unknown (43)[Unreassembled Packet]
26	0.141152	22	59159	48	55	65728	Unknown (237)[Unreassembled Packet]
28	0.141500	22	59159	80	55	65728	Global Request[Unreassembled Packet]
29	0.008202	22	59159	48	55	65728	Unknown (211)[Unreassembled Packet]
32	0.141173	22	59159	48	55	65728	Unknown (120)[Unreassembled Packet]
38	0.141348	22	59159	0	55	64256	22 → 59159 [ACK] Seq=2561 Ack=5652 Win=64256 Len=0

左上角是过滤器，右侧第二列就是刚刚新添加的 Calculated window size。显然，能从图中看出，传输起始阶段有不少重传。不过没关系，让我们集中注意力到接收窗口值上。

由图可见，这个阶段的接收窗口是 64KB 上下浮动，值偏小。照理来说，传输稳定阶段，接收窗口会大很多，也就应该会比 64KB 大不少吧？我们往下翻，滚动到整体报文的中间位置，看看此时接收窗口值是多少：



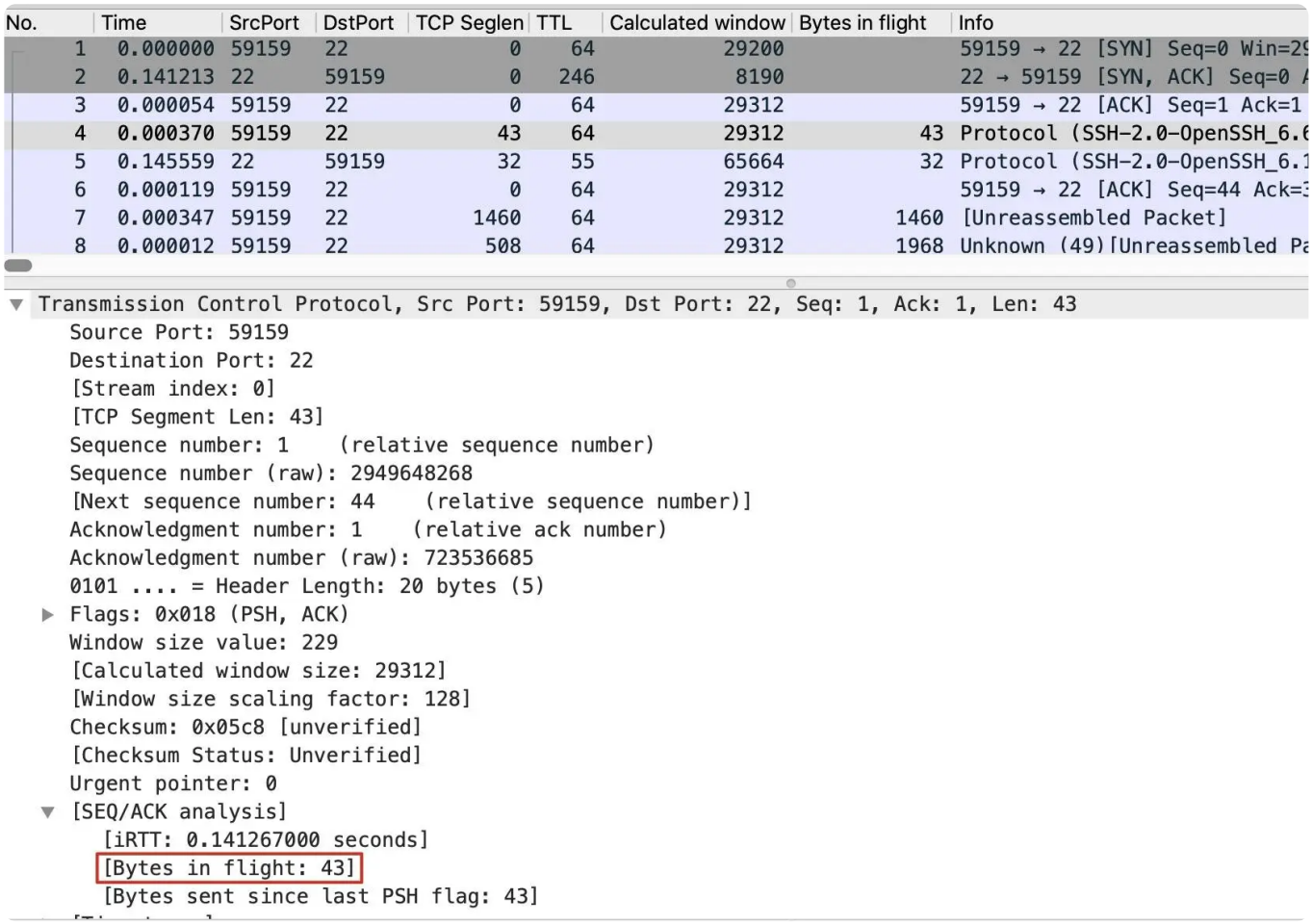
tcp.srcport eq 22

No.	Time	SrcPort	DstPort	TCP Seglen	TTL	Calculated window	Info
92…	0.000415	22	59159	0	55	62784	22 → 59159 [ACK] Seq=6689 Ack=11311892 Win=62784 Len=0
92…	0.000050	22	59159	0	55	59904	22 → 59159 [ACK] Seq=6689 Ack=11314812 Win=59904 Len=0
92…	0.000006	22	59159	0	55	56960	22 → 59159 [ACK] Seq=6689 Ack=11317732 Win=56960 Len=0
92…	0.000002	22	59159	0	55	54016	22 → 59159 [ACK] Seq=6689 Ack=11320652 Win=54016 Len=0
92…	0.000003	22	59159	0	55	65728	[TCP Window Update] 22 → 59159 [ACK] Seq=6689 Ack=11320652
92…	0.000054	22	59159	0	55	62784	22 → 59159 [ACK] Seq=6689 Ack=11323572 Win=62784 Len=0
92…	0.000044	22	59159	0	55	65728	[TCP Window Update] 22 → 59159 [ACK] Seq=6689 Ack=11323572
92…	0.000113	22	59159	0	55	62784	22 → 59159 [ACK] Seq=6689 Ack=11326492 Win=62784 Len=0
92…	0.000005	22	59159	0	55	65728	[TCP Window Update] 22 → 59159 [ACK] Seq=6689 Ack=11326492
92…	0.001098	22	59159	0	55	64256	22 → 59159 [ACK] Seq=6689 Ack=11329412 Win=64256 Len=0
92…	0.000506	22	59159	0	55	65728	22 → 59159 [ACK] Seq=6689 Ack=11330872 Win=65728 Len=0
92…	0.000092	22	59159	0	55	62784	22 → 59159 [ACK] Seq=6689 Ack=11333792 Win=62784 Len=0
92…	0.000006	22	59159	0	55	65728	22 → 59159 [ACK] Seq=6689 Ack=11335252 Win=65728 Len=0
92…	0.000067	22	59159	0	55	62784	22 → 59159 [ACK] Seq=6689 Ack=11338172 Win=62784 Len=0
92…	0.000005	22	59159	0	55	65728	[TCP Window Update] 22 → 59159 [ACK] Seq=6689 Ack=11338172

有点意外，不仅没有上升，反而偶尔还略微下降了，在 54~64KB 之间浮动。这是怎么回事？这跟速度慢是否有关系？

前面提到过，发送端实际的发送窗口是拥塞窗口和对方接收窗口这两者中的较小者。那究竟是多少呢？我们可以这么找到它：

> 选中传输中的一个报文，在 TCP 详情的[SEQ/ACK analysis]部分，找到[Bytes in flight]。



No.	Time	SrcPort	DstPort	TCP Seglen	TTL	Calculated window	Bytes in flight	Info
1	0.000000	59159	22	0	64	29200		59159 → 22 [SYN] Seq=0 Win=29
2	0.141213	22	59159	0	246	8190		22 → 59159 [SYN, ACK] Seq=0 A
3	0.000054	59159	22	0	64	29312		59159 → 22 [ACK] Seq=1 Ack=1
4	0.000370	59159	22	43	64	29312	43	Protocol (SSH-2.0-OpenSSH_6.6
5	0.145559	22	59159	32	55	65664	32	Protocol (SSH-2.0-OpenSSH_6.1
6	0.000119	59159	22	0	64	29312		59159 → 22 [ACK] Seq=44 Ack=3
7	0.000347	59159	22	1460	64	29312	1460	[Unreassembled Packet]
8	0.000012	59159	22	508	64	29312	1968	Unknown (49)[Unreassembled Pa

```
▼ Transmission Control Protocol, Src Port: 59159, Dst Port: 22, Seq: 1, Ack: 1, Len: 43
    Source Port: 59159
    Destination Port: 22
    [Stream index: 0]
    [TCP Segment Len: 43]
    Sequence number: 1    (relative sequence number)
    Sequence number (raw): 2949648268
    [Next sequence number: 44    (relative sequence number)]
    Acknowledgment number: 1    (relative ack number)
    Acknowledgment number (raw): 723536685
    0101 .... = Header Length: 20 bytes (5)
  ▶ Flags: 0x018 (PSH, ACK)
    Window size value: 229
    [Calculated window size: 29312]
    [Window size scaling factor: 128]
    Checksum: 0x05c8 [unverified]
    [Checksum Status: Unverified]
    Urgent pointer: 0
  ▼ [SEQ/ACK analysis]
      [iRTT: 0.141267000 seconds]
      [Bytes in flight: 43]
      [Bytes sent since last PSH flag: 43]
```

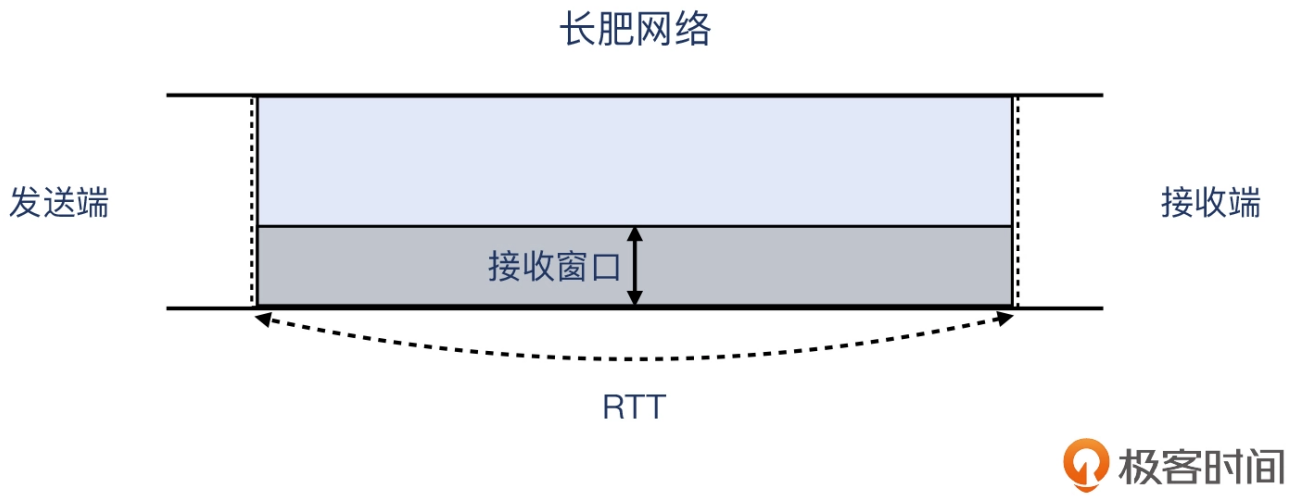
然后右单击，选中 Apply as column。

主界面里有 Bytes in flight 这一列后，你就可以排序，下图中显示，这次传输的客户端在途字节数最大是 65700，接近 64KB，这就是发送窗口。

Apply a display filter ... <⌘/>

No.	Time	SrcPort	DstPort	TCP Seglen	TTL	Calculated window	Bytes in flight	Info
16...	0.000012	59159	22	2920	64	34688	65700	Unknown (69)[Unreassembled Packet]
16...	0.000006	59159	22	2920	64	34688	65700	Unknown (209)[Unreassembled Packet]
16...	0.000015	59159	22	2920	64	34688	65700	Unknown (217)[Unreassembled Packet]
16...	0.000012	59159	22	2920	64	34688	65700	Unknown (252)[Unreassembled Packet]
16...	0.000010	59159	22	2920	64	34688	65700	Unknown (56)[Unreassembled Packet]
16...	0.000010	59159	22	2920	64	34688	65700	Unknown (101)[Unreassembled Packet]
15...	0.000003	59159	22	2920	64	34688	65700	Channel Open[Unreassembled Packet]
15...	0.000019	59159	22	2920	64	34688	65700	Unknown (246)[Unreassembled Packet]

64KB 很小啊，这就相当于，在一个长肥网络中，存在着一个“瘦”很多的管道，完全没有把带宽利用起来。我觉得可以叫做“长瘦管道”。



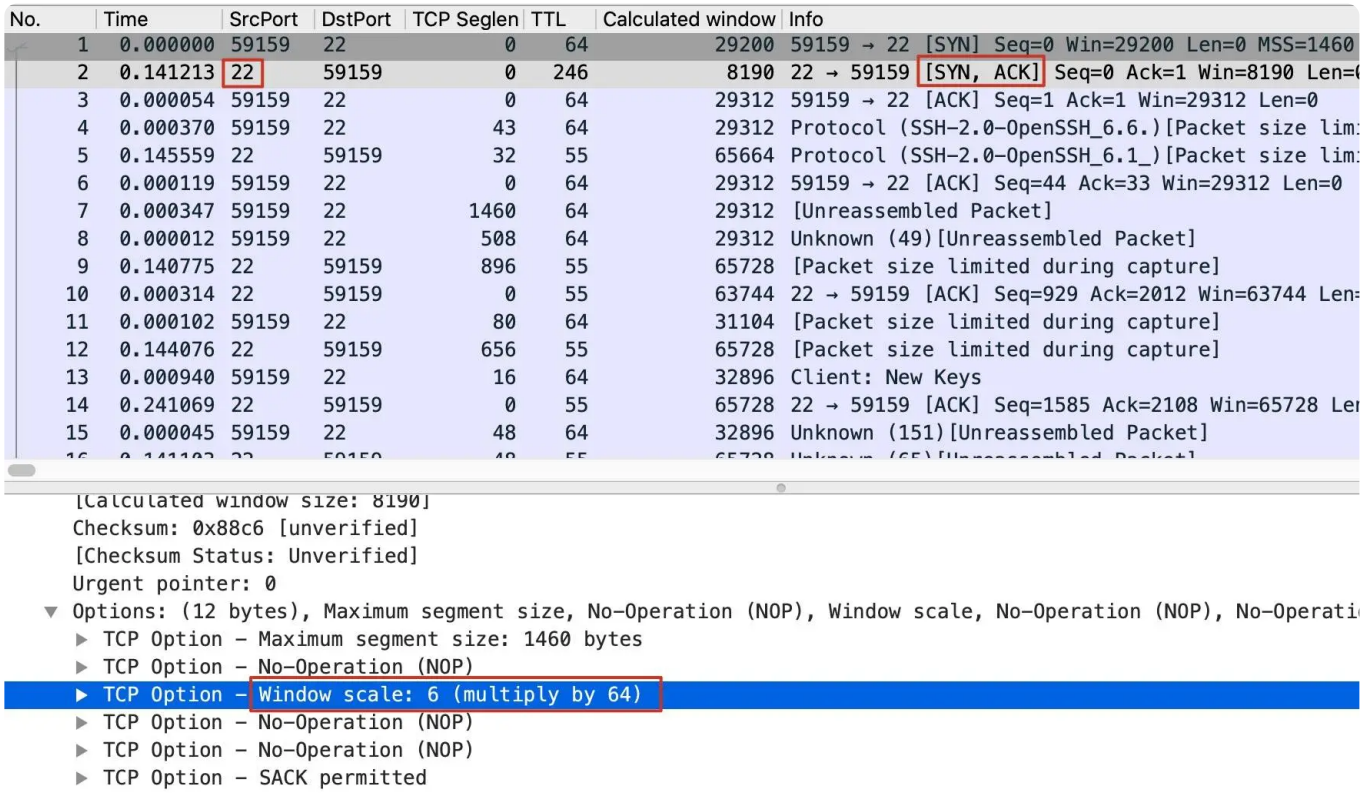
这个长瘦管道的传输速度是多少呢？在这个案例里，发送窗口 = 接收窗口，那么在网络上跑的数据量最多就是 64KB。那传输这 64KB 的耗时是多久呢？这不就是往返时间 RTT 吗？现在，你是否终于回想起前面我卖关子的那个小学公式呢？

> **速度 = 距离 / 时间**

所以，这次的传输速度的上限就是 window/RTT = 64KB/134ms = **478KB/s**！还记得前面用 TCP Stream Graphs(Stevens) 斜线图得到的速率吗？那个是 456KB/s。可见，实际值只比上限值略低，说明虽然窗口很小，但传输过程本身还是比较充分地利用了这个有限的窗口。

整个案件得以真相大白：限制速度的最大因素，既不是带宽，也不是丢包，而是**窗口**，确切地说就是**接收端（POP 点）的接收窗口**。因为这些接收端的设备比较特殊，沿用了老旧的配置，导致 TCP 接收窗口过小。

窗口小是因为 Window Scale 没被启用吗？我们检查发现，Window Scale 其实启用了，你可以回头看一下窗口值的截图，有几次的值是 65728，明确超过了 65535。另外，在握手包里也可以看到 Window Scale 被启用的信息：



> 红框中的 `TCP Option-Windows scale: 6` 就是表示窗口系数（Scale Factor）为 6，即 2 的 6 次方（得 64）。也就是说，真正的接收窗口值，是 Window 字段的值乘以 64。

那为什么启用了 Window Scale，还是被限制在 64KB 附近呢？我们发现，这还是受限于这台设备本身的特点。在某些情况下，这里的 Window Scale 虽然启用了，但无法充分工作，导致实际上这台设备的接收窗口一直被压制在 64KB 附近。

想象一下，发送端每次“喂到”网络上的数据只有 64KB，这 64KB 还需要经过 134ms 后才能到达接收端。然后就这样周而复始，直到全部的 95MB 发送完成。明明是很宽敞的网络，但可惜，对端的收发室太小了，你只好把大货车停车库里，改开电瓶车，一小车一小车地送货，你急也没用。这速度，哪里还起得来？

我们对它的配置做了修改，使得接收窗口大幅增加后，速度马上提上去很多倍，这才彻底解决了问题。

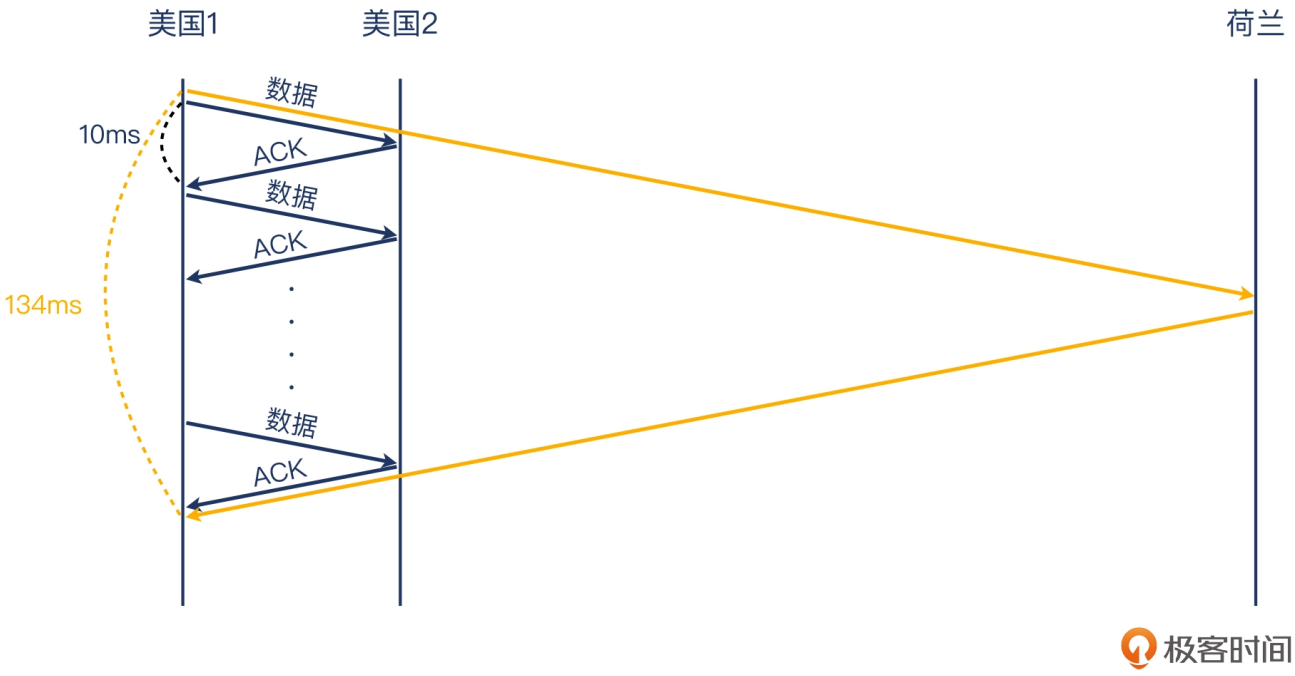
最后的疑问

不过你是否留意到了，难道当时美国本土数据中心里面的设备就没有这个问题吗？如果也是用了这个老旧的配置，难道就没有传输慢的问题吗？

我给你的答案是：其实也是老旧的配置，其实速度也是慢的。但是为什么这个问题就没有被暴露呢？这又是跟前面的知识点有关系了。你能想到是哪个因素掩盖了这个问题吗？

这次的答案不是窗口，而是**往返时间**。在美国本土的多个数据中心之间，RTT 大约在 10ms 多一点，也就是只有美国到荷兰的 134ms 的十分之一。联系公式“**速度 = 距离 / 时间**”，分子（接收窗口）不变，分母（时延）变为原来的十分之一，那么速度会变成原来的十倍。十倍的速度就没有显得那么慢了，所以并没有引起我们的注意。

我用一个图表示一下这种 RTT 的不同引起的传输上的区别：



你看，传输速度的问题，可以说就是窗口和往返时间这两个大玩家在起作用。你只要**抓住这两个主要矛盾，就能解决大部分传输速度的问题**了。其他因素也可能有它的作用，但一般不是核心矛盾。我们要学会抓重点，这对于工作，乃至对于人生，都是很有意义的。

小结

这节课，我用了一个典型的文件传输的案例，帮你回顾了跟 TCP 传输有关的方方面面的知识点，包括：

时延：也叫往返时间，是通信两端之间的一来一回的时间之和。

在途报文：发送端已经发出但还未被确认的报文，时延越长，发送窗口越大，在途报文可能越多，这两者是正比关系。

带宽时延积：带宽和时延的乘积，表示这个网络能承载的最多的在途数据量。

发送窗口：发送窗口是拥塞窗口和对方接收窗口两者中的较小值。

基于以上的知识，我们得以推导出最终的**核心公式：速度上限 = 发送窗口 / 往返时间**。用英文可以表示为：velocity = window/RTT。

以后你在处理 TCP 传输速度问题的时候，一样可以应用上面这些知识：先获取时延，再定位发送窗口，最后用这个公式去得到速度的上限值。

除了这些 TCP 的知识点，我也提到了使用 Wireshark 的一些技巧，包括：

查看 I/O Graph 的方法，这个可以直观地展示数据传输的速度。

查看 TCP Stream Graphs 的方法，这个能看到 TCP 序列号随着时间的变化趋势，同时也可以经过简单计算推导出 TCP 载荷的传输速度（因为没有计入各种报文头部，这个值比 #1 的速度值要略低一些）。

查看 TCP Window Scale 是否启用，以及启用的话，Scale Factor 的值的查找方法。

总而言之，你在自己处理类似的传输速度、延迟等问题的时候，一方面可以充分利用我分享的协议方面的知识，获得坚实的理论支撑；一方面也可以使用这节课提到的这些 Wireshark 分析技巧，获得数据上的支持。当你把理论和数据这两条“腿”都锻炼健壮后，你一定能在网络排查的世界里，更加坚定有力地前行。

思考题

你有没有在工作中遇到过 TCP 传输速度相关的问题呢？通过这节课的学习，你已经掌握了传输速度相关的不少知识，你准备怎么运用这些知识，来解决这个传输问题呢？

欢迎在留言区分享出你的答案和思考过程，也欢迎你把今天的内容分享给更多的朋友，我们一起成长。

上一篇 08 | 分段：MTU引发的血案

下一篇 10 | 窗口：TCP Window Full会影响传输效率吗？

精选留言 (7)

写留言

江山如画

2022-02-09

我们公司之前有一个场景是多个客户端连接同一个服务端，如果某个客户端下载文件或上传文件，占用大量的带宽，会导致新的客户端连不上服务端。后来在服务端使用 tc 工具做了流控，根据客户端个数均分带宽，还有限制单个IP的最大使用带宽。

学了今天的课程后，突然想到 tc tools 这类流控工具，底层原理是不是就是调整服务端...

展开

共 2 条评论 > 5

一把螺丝刀

2022-02-10

请问是做了什么修改来增大窗口呢？

作者回复: 您好，案例里的设备是特殊硬件有其自己的配置方式，不具备普适性就没展开了。linux有相关参数，比如mem_rmax、rmin这些，都会影响接受窗口，你可以调整看看效果

3

Realm
2022-02-09

设备上配置了限速64k引起的？老师，一般网络限速是基于窗口吗？

作者回复: 确切说并不是直接调整带宽限制，而是修改设备对每个tcp连接的最大接收窗口的限制。按照"窗口/往返时间"这个公式，同一个窗口配合不同的往返时间，起到的速度限制各不相同~

共 2 条评论 >

1

我来也
2022-02-11

刚才点错了，点成发布了。这个继续。

最近两天的研究，发现跟我的路由器有关系。
1. 我通过wifi连接家里的路由器，推送时会卡住，但通过手机热点推送又正常。检查了mtu，mss之类的没看到异常。...

展开

共 6 条评论 >

我来也
2022-02-11

最近的两篇文章比较有意思，可以借来排查一下一个比较困扰我的问题。
我电脑推送docker镜像时某些层经常卡住，然后重试。

中间的链路非常扰：
mac 的docker 是自己的一套网段，我需要通过一个http 代理来推送镜像。我电脑上用了...

展开

宝仔
2022-02-10

老师你好，咨询一个问题，在linux服务器上如何使用tcpdump抓取https协议的报文，并且以明文的形式展现。有些场景需要判断客户端是否有bug，发过来的http报文中是否带了某个header，客户端太多并且也没权限，所以需要在服务端抓包

作者回复: tcpdump工作在底层，无法获取，也不应该获取TLS的密钥，所以不能直接解密的，这是安全设计使然。

后面课程（实战二的TLS排查部分）我会介绍如何解密https流量的技术细节和相关原理的～

共 2 条评论 >

taochao_zs

2022-02-09

之前碰到过公有云智能dns解析异常：ip地址池解析取远不取近，进而影响RTT变大，最后引起文件传输性能下降。

作者回复: 不错，选取的服务端ip的远近造成了往返时间的区别，对速度影响明显～