11 | 拥塞：TCP是如何探测到拥塞的？

2022-02-14 杨胜辉

《网络排查案例课》 课程介绍 >

讲述：杨胜辉
时长 24:25 大小 22.37M

你好，我是胜辉。

前面两节课，我们通过真实的案例，一起学习了 TCP 传输方面的知识，比如其中的核心概念：往返时间、接收窗口和发送窗口、在途字节数，还有推导出来的核心公式。

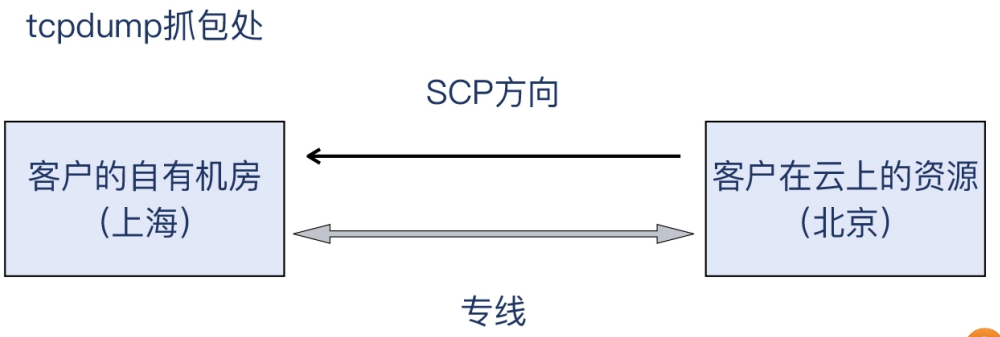
当然，在实际场景里，我们可以直接利用 Wireshark 的 I/O Graph 查看速度趋势图，这样最方便，并且有不同时段的速度，方便我们对整体状况做全面的评估。

不过，不知道你有没有发现，TCP 传输的起始阶段，速度都是从低到高升上来的，很少一上来就直接以最终速度运行的情形。其实，这个行为跟 **TCP 拥塞控制**有着密切的关系。所以这节课，我会带你了解什么是拥塞窗口、TCP 是如何检测和避开拥塞的。这样呢，以后你处理 TCP 拥塞相关的问题时候，就能有的放矢，做到有针对性的分析了。

好，让我们来看一个具体的案例吧。

案例

在公有云服务的时候，我们有个银行的客户，他们有一次需要跨机房拷贝一个大文件，也就是用 SCP 命令，把文件从公有云机房拷贝到他们的自建机房。但是客户发现速度比较慢，让我们看一下原因。



架构上说，客户自己的机房在上海，公有云上的资源则在北京。地理位置相差很远，而且机房所属的性质也完全不同，那两者如何通信呢？走公网的话不太安全，而且速度和质量没有保障。

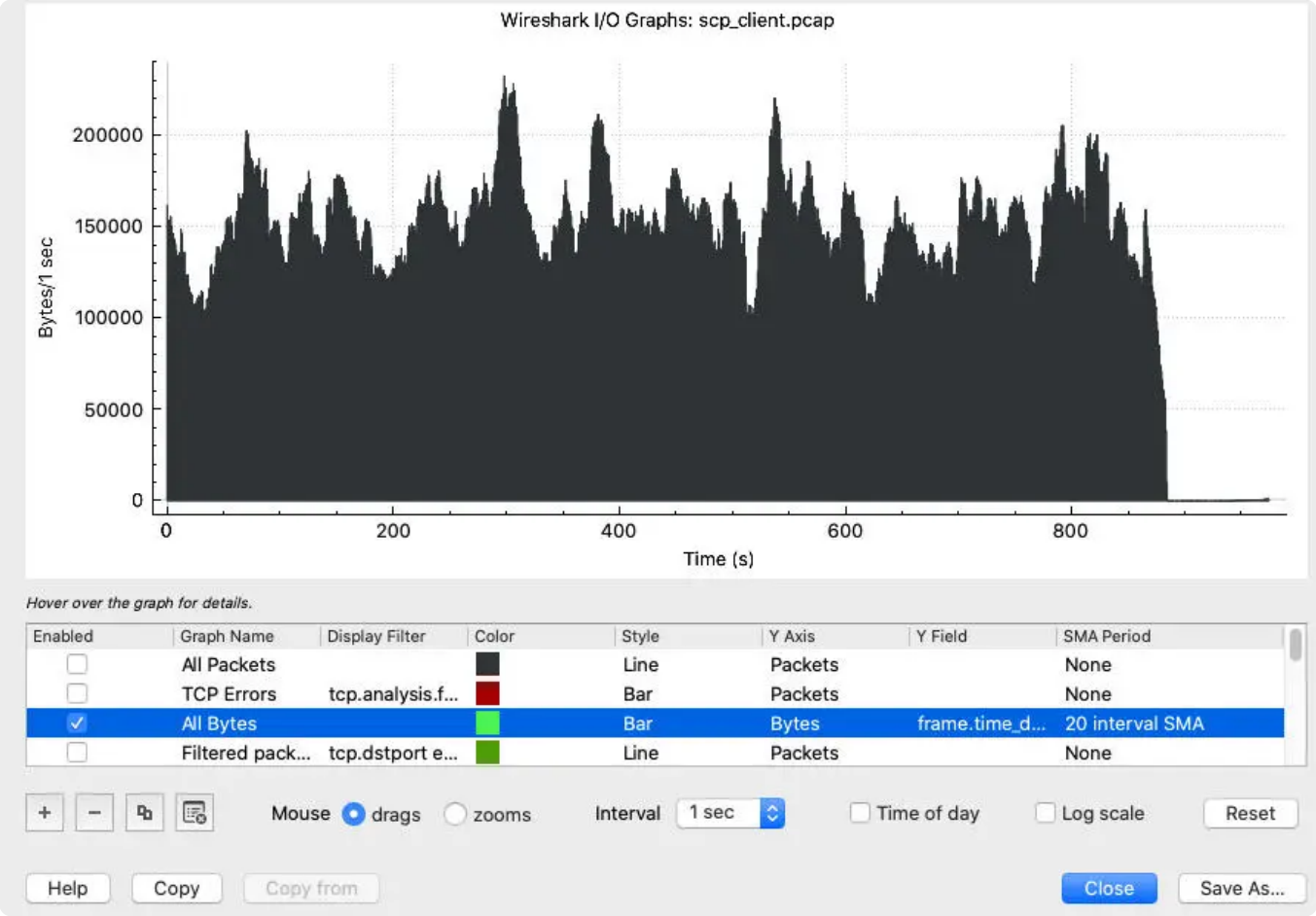
如果你熟悉网络的话，可能知道可以在两个机房之间搭建 VPN。而从可靠性角度考虑，客户选择了使用公有云的专线产品，也就是在上海自有机房和北京公有云之间，打通了专线。这样，客户在上海的自有机房，就可以直接以 10.x.x.x 这种内网地址，直达他们在北京公有云的资源，就好像真的在同一个内网一样。是不是挺酷的？

> 补充：上传的抓包文件我做了脱敏修改，所以 IP 也是随机值而不是 10.x.x.x。

可是，他们在传输文件的时候，发现速度只有 200KB/s 左右，达不到购买的专线带宽值。正好他们也在传输过程做了 tcpdump 抓包，我们来看看这个抓包文件的具体情况。

> 抓包示例文件已上传至 Gitee，你可以用 Wireshark 打开这个文件，跟随我的分析步骤来同步学习。

跟我们在第 9 讲和第 10 讲学过的一样，我们可以用 **I/O Graph** 这个小工具来直观地看一下传输速度：



从图上看，速度大体上在 100~200KB/s 之间浮动，有少数几个时段的速度在 230KB/s 左右。这是传输速度方面的整体概览。

另外就是要查看 TCP 传输过程中的一些行为了，这些行为没办法在 I/O Graph 上体现出来，但是它们很可能就是“因”，正是因为它们，才有了传输速度这个“果”。

现在课程快学到一半了，你应该对 Expert Information 很熟悉了吧？这次也不例外，我们来看一下 Expert Information：

Packet	Summary	Group	Protocol	Count
Warning	This frame is a (suspected) out-of-order segment	Sequence	TCP	266
Warning	Previous segment(s) not captured (common at capture start)	Sequence	TCP	4588
Note	This frame is a (suspected) spurious retransmission	Sequence	TCP	15
Note	This frame is a (suspected) fast retransmission	Sequence	TCP	1695
Note	This frame is a (suspected) retransmission	Sequence	TCP	4617
Note	Duplicate ACK (#1)	Sequence	TCP	18945
Chat	TCP window update	Sequence	TCP	1279

可见信息量还挺大的，包括了多种行为。

Warning 级别有两种，分别是乱序（Out-of-Order）和前面报文未抓取的情形。这两者本质上都是乱序引起的现象。

Note 级别，一共有四种，分别是：

- **Spurious 重传**：这是已经被确认过的数据再一次被重传。
- **快速重传**：收到 3 次及以上次数的重复确认后，不等超时就做出的重传。
- **重传**：超时计时器到点而触发的重传，这就是有名的超时重传。
- **重复确认**：确认号重复的多个报文，重复确认是引发快速重传的原因。

Chat 级别的 **TCP Window update**，这里主要是客户端（上海）向 SSH 服务端（北京）通告自己的接收窗口的变化，是比较正常的行为。

既然还是跟传输速度相关的话题，你应该还记得上节课刚深入讨论过的 TCP Window Full 了吧？但是这里为什么一条这样的信息都没有呢？

对于 TCP 传输来说，其速度大体上是窗口 / 往返时间。而这里的窗口，在不同情境下就有着不同的含义。我们用 CW（Congestion Window）来指代自身的拥塞窗口，而用 RW（Receive Window）指代对端的接收窗，那么：

当 RW<CW 时，这里的“窗口”就是 RW；

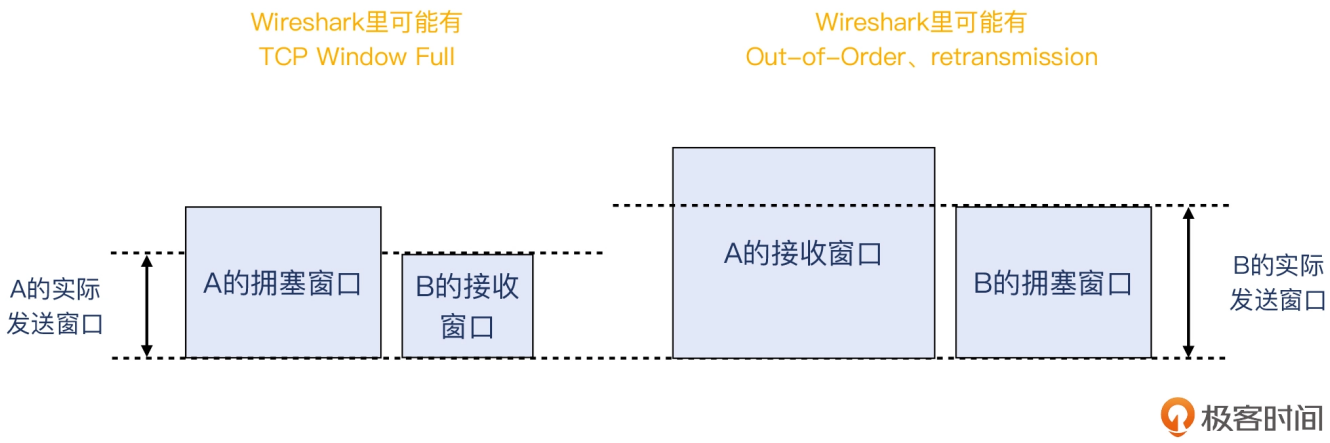
当 RW>CW 时，这里的“窗口”就是 CW。

对于情况 1，也就是对端的接收窗口小于自身拥塞窗口的情况，一般意味着传输过程中没有或者很少有“拥塞”发生，因而拥塞窗口能增长到较高的值。所以，传输速度的上限就是对端接收窗口值决定的。这样呢，也就容易在 Wireshark 里观察到 TCP Window Full 这样的现象。当然，也不是每次都一定有 TCP Window Full，像第 9 讲的案例就没有。

对于情况 2，也就是对端的接收窗口大于自身拥塞窗口的情况，这一般意味着传输过程中遇到了“拥塞”，因而拥塞窗口进行了适配，也就是往下调整，这往往会使得拥塞窗口变得比较小。

事实上，我们在第 9 讲也已经初步介绍了上面这些知识。我做一下搬运工，同时也做一下美工，对第 9 讲的图补充了 Wireshark 可能解读出来的信息，供你参考。下次你在

Wireshark 看到 TCP Window Full、Out-of-Order、或者 retransmission 时，就可以跟这里的图对应起来，协助你排查传输速度方面的问题了。



在当前这个案例里，因为乱序、重传等都有大几千个，非常多，所以我们初步判断，应该就是这些事件导致传输遇到了拥塞，也进而限制了传输速度。那么到这里，我们也就正式进入拥塞机制的学习了。我会给你概括其关键部分，也会结合案例里的抓包文件，来带你获得一个更加感性的认识。

TCP 拥塞控制

为了应对错综复杂的互联网网络环境，TCP 使用了**拥塞控制机制**来确保传输速度和稳定性。这里你也要注意，总的来说，拥塞控制主要是通信两端自己需要实现的功能，而途中的网络设备，比如交换机、路由器等等，除了可能会发出拥塞通知报文以外，其他时候它们只管转发报文，都是不会担负更多的拥塞控制的责任的。

TCP 拥塞控制主要有四个重要阶段：

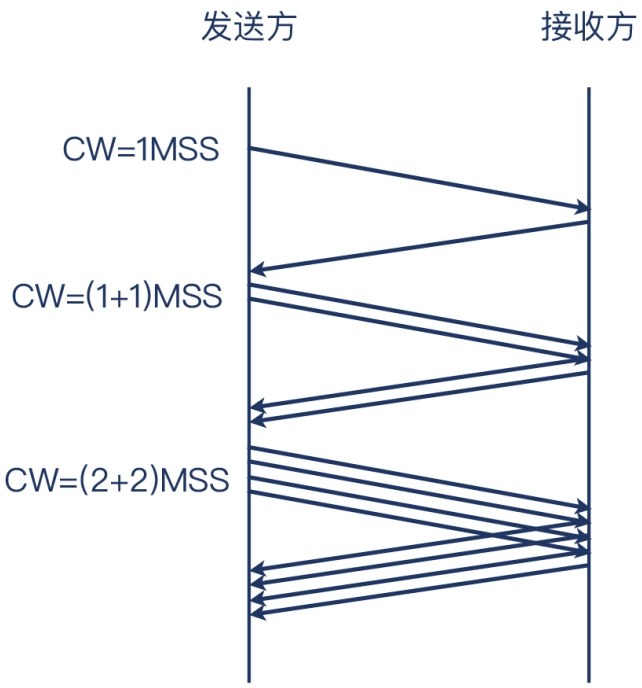
- 慢启动；
- 拥塞避免；
- 快速重传；
- 快速恢复。

其中还包括拥塞窗口这个概念，接下来我给你逐一介绍一下。

慢启动

Slow Start，是指 TCP 传输的开始阶段是从一个相对低的速度（“慢”一词的由来）开始的。事实上，在这个阶段，拥塞窗口会以翻倍的方式增长，所以从增长过程来看，叫“快启动”也未尝不可。

具体来说，在这个阶段，每次 TCP 收到一个确认了数据的 ACK，拥塞窗口就增加一个 MSS。比如下面这样：



不过这里的“确认了数据的 ACK”怎么理解呢？

它说的是有确认数据的 ACK 报文，而不是重复的 ACK 报文。比如收到 2 个 ACK 但确认号一样，那第二个 ACK 就不是“确认了数据的 ACK”了，拥塞窗口不会增加 2 个 MSS，而是只增加 1 个 MSS。

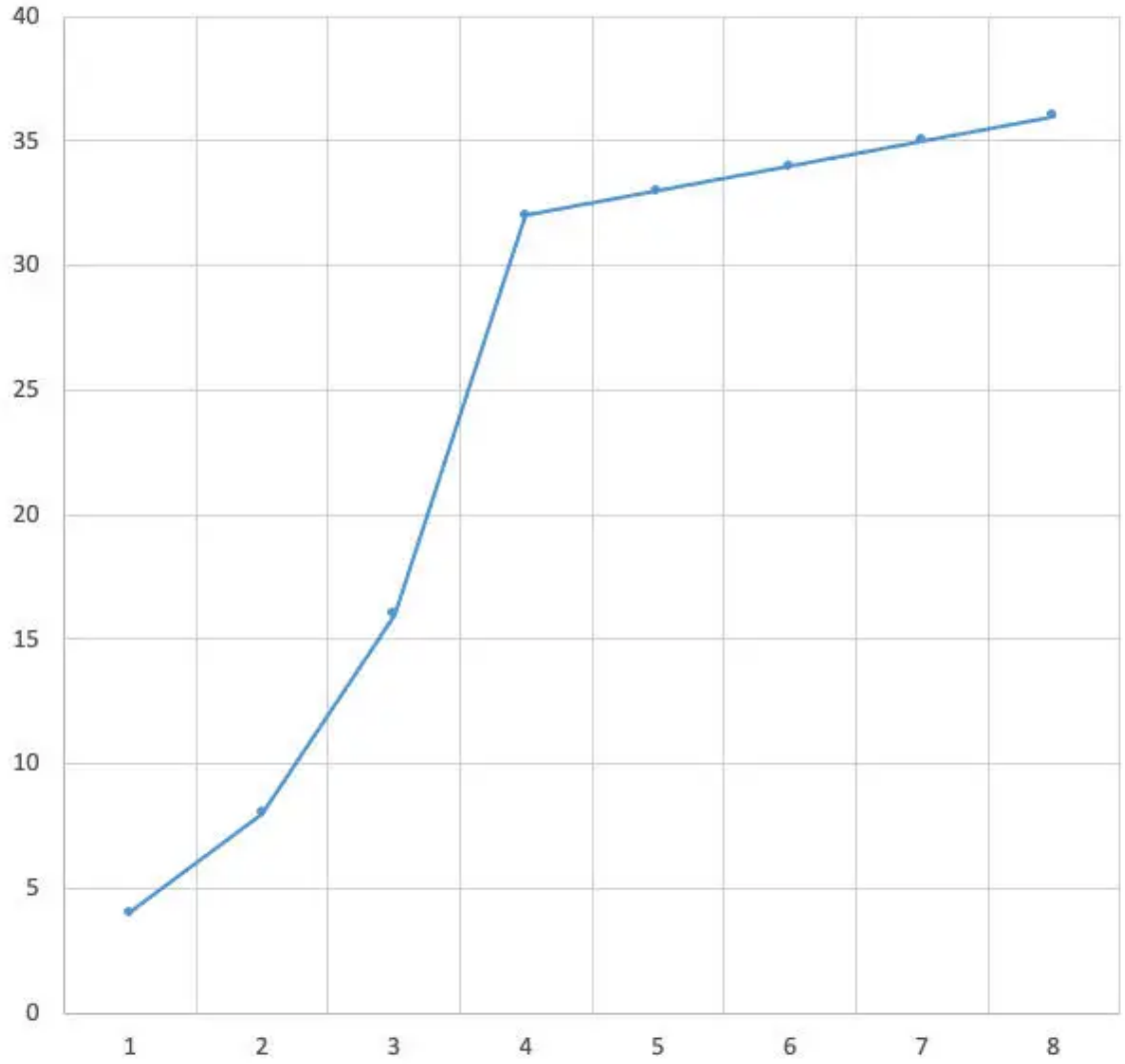
那么，这个过程什么时候终止呢？是下面两件事中有一件发生时：

遇到了拥塞；

拥塞窗口增长到慢启动阈值。

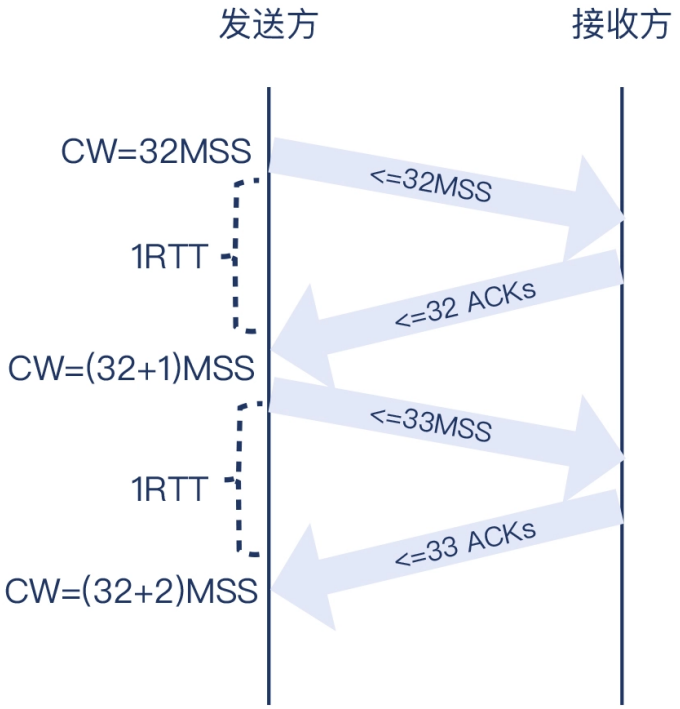
慢启动阈值

慢启动阈值（也有人称之为慢启动门限），英文简称 ssthresh。过了这个阈值，拥塞窗口的增长速度立刻就放缓了，变成了每过一个 RTT，拥塞窗口就只增长一个 MSS（此前是每个确认数据的 ACK，增长一个 MSS）。



比如上图的例子中，假设 ICW 是 4 个 MSS，ssthresh 是 32 个 MSS。在慢启动阶段，经过一个 RTT 后，CW 扩大为 8 个 MSS，然后是 16 个 MSS，32 个 MSS，以指数级上升。

那么到了这个阈值后，TCP 就进入了**拥塞避免阶段**，每过一个 RTT，拥塞窗口只增加一个 MSS，于是在图上看，就又变成了一条平直的斜率比较低的直线了。



极客时间

那么，如果拥塞窗口正好等于慢启动阈值，发送方应该选择继续慢启动过程（指数性增长），还是拥塞避免过程（线性增长）呢？RFC5681 的规定是“没有规定”，两种都可以。

间隔确认

这里有个情况必须要提一下。很多 TCP 实现里（比如 Windows 系统），确认报文是这样工作的：如果收到连续多个报文，确认报文是一个隔一个回复。也就是：

收到 1、2，对 2 进行确认；

收到 3、4，对 4 进行确认。

比如就在这个案例里，我们很容易就发现有这种隔一个报文再确认的现象：

No.	Time	SrcPort	DstPort	TCP Seglen	TTL	Sequence	Acknowledgment	nextSeq	Info
15	0.000520	22	56974	1388	59	5777	417	7165	Encrypted packet (len=1388)
16	0.000543	22	56974	1388	59	7165	417	8553	Encrypted packet (len=1388)
17	0.000028	56974	22	0	64	417	8553	417	56974 → 22 [ACK] Seq=417 Ac
18	0.000503	22	56974	1388	59	8553	417	9941	Encrypted packet (len=1388)
19	0.000545	22	56974	1388	59	9941	417	11329	Encrypted packet (len=1388)
20	0.000013	56974	22	0	64	417	11329	417	56974 → 22 [ACK] Seq=417 Ac
21	0.000510	22	56974	1388	59	11329	417	12717	Encrypted packet (len=1388)
22	0.000592	22	56974	1388	59	12717	417	14105	Encrypted packet (len=1388)
23	0.000025	56974	22	0	64	417	14105	417	56974 → 22 [ACK] Seq=417 Ac

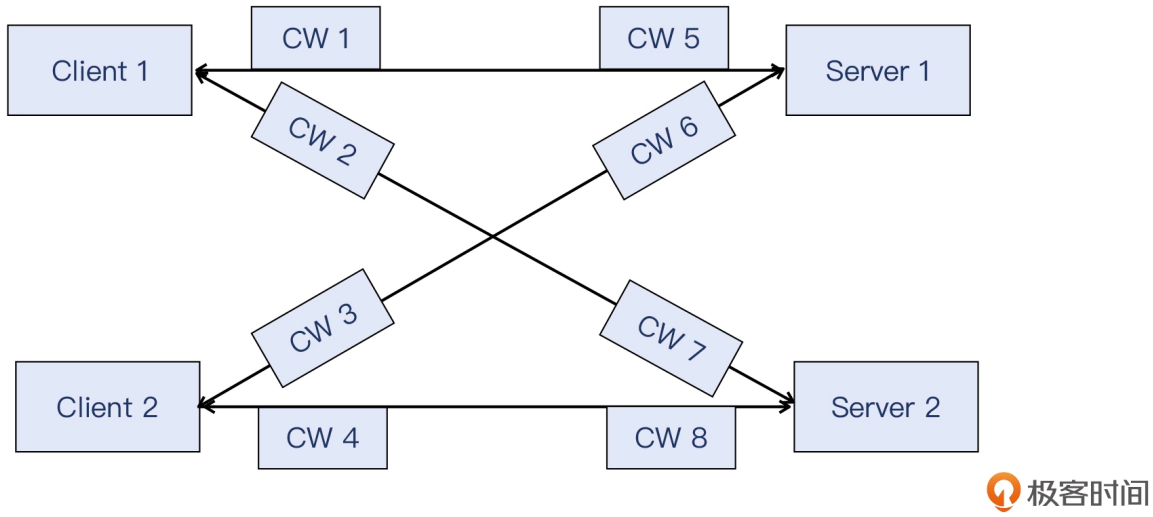
上图中，我们选中了 17 号报文，Wireshark 自动找到了被它确认的 16 号报文，也在它的左边打上了一个小小的勾。当然，我们也可以通过对比 NextSeq 和 ACK 来找到这种关系。我在图中就用红框和箭头找到了这里的 3 对 TCP 确认关系。

这样的间隔 ACK，可能会使得拥塞窗口的增长速度，比每次都 ACK 要更低一些。

拥塞窗口

Congestion Window，缩写是 CWND，或者 CW。拥塞窗口是不是操作系统全局统一的配置呢？其实这是比较常见的误解。拥塞窗口是每个连接分开维护的，比如同一个主机有两个 TCP 连接在传输数据的话，那么这两个连接就各自维护自己的拥塞窗口，比如一个很大而一个很小，都没有关系。

下图中，我用 CW 指代拥塞窗口，图中 CW1 到 CW8 都是各自不同、独立维护的拥塞窗口：



这里还有一个子概念很重要，叫**初始拥塞窗口**，英文是 Initial Congestion Window（或者 Initial Window），缩写为 ICW（或者 IW）。

在 Linux 内核 3.0 以前，初始拥塞窗口的大小比较小，在 2 到 4 个 MSS。2010 年，谷歌提出，为了充分利用现代互联网的传输能力，Linux 应该把 ICW 从 2~4 个 MSS 提升到 10 个 MSS。这也被应用到了 Linux 内核 3.0 版本及以后的版本中，比如在 include/net/tcp.h 中，就定义了 TCP_INIT_CWND 的值为 10。

```
/* TCP initial congestion window as per rfc6928 */
#define TCP_INIT_CWND       10
```

复制代码

前面刚介绍过，在慢启动阶段，每过一个 RTT，拥塞窗口就翻倍。那么不同的 ICW 就会造成不同的传输速度，比如：

ICW	CW after 1st RTT	CW after 2nd RTT	CW after 3rd RTT
2	4	8	16
10	20	40	80

极客时间

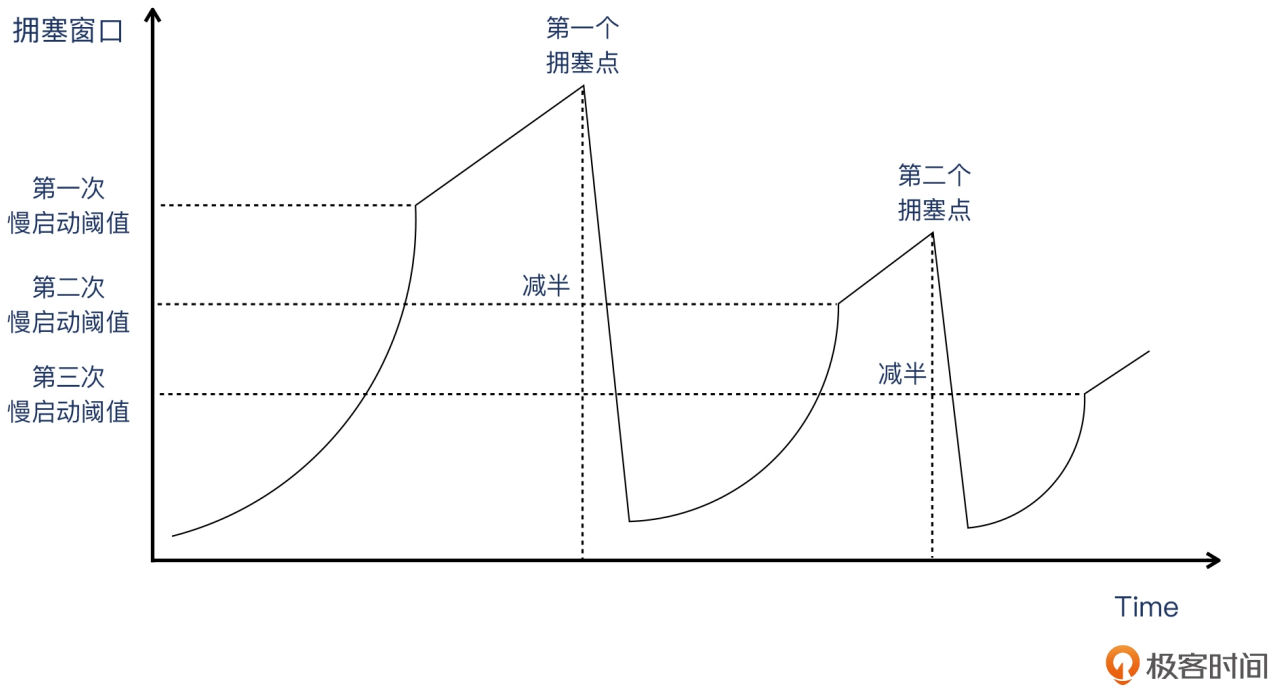
看着 ICW 的变迁，我真的就觉得很多知识是相通的。还记得我们在第 3 讲学习 TCP 握手的时候，介绍的 Window Scale 概念吗？为什么已经有 Window 字段，设计者们还要创造 Window Scale 呢？本质原因还是互联网发展很快，原先设计的 Window 不够用了。

那么现在 ICW 的增加也是如此：既然互联网条件好了那么多，咱就不要过于谨慎了吧？天地大得很，上来就迈大点的步子，后面跑起来就快上加快了！

拥塞避免

前面我说过，传输过了慢启动阈值（ssthresh）之后，就进入了**拥塞避免**阶段。这个阶段的特征是“**和性增长乘性降低**”，英文是 Addictive increase/mutiplicative decrease，缩写为 AIMD。它也翻自英文，怪不得这中文念起来略有不顺，特别是“和性增长”。其实说是“佛系增长”也许更容易理解吧，因为增长很慢，挺佛系的。

那么，因为 AIMD 的关系，每一个 RTT 里，拥塞窗口只增长一个 MSS，所以这个阶段的拥塞窗口的增长是线性的。直到探测到拥塞，然后拥塞窗口就要往下降。这个下降是直接减半的，所以叫**乘性降低**。我画了一个示意图，给你做参考：



当然，图中的第二个拥塞点比第一个低只是一种可能的情况，现实场景里什么情况都可能有，因为网络状况本身就是动态变化的。

窗口和 MSS 的关系

本来这也属于拥塞算法相关的知识点，但因为确实是比较常见的误区，所以我在这里单独拎出来介绍一下。首先，窗口一般比 MSS 大，而且大很多，可能会有个别同学以为“MSS 就是窗口最大值”。其实这是反过来的：窗口值一般比 MSS 大很多，相当于就是 MSS 的某个倍数，比如 2 倍、10 倍、50 倍等等。

MSS 是有确定上限的，我在前面课程里都多次提到过，MSS 一般为 1460，当然根据实际情况，也经常会有更低的值。比如，开启 TCP timestamp 等 Option 的话，肯定要相应地从 1460 字节里扣去这部分字节数，这样的话 MSS 就会低于 1460，比如可能是 1440 字节。

你可以理解为：**窗口就是 n 个 MSS**。

> 补充：确切来说，窗口的单位是字节数，所以也经常不是 MSS 的整数倍，这也都是正常的。

快速重传

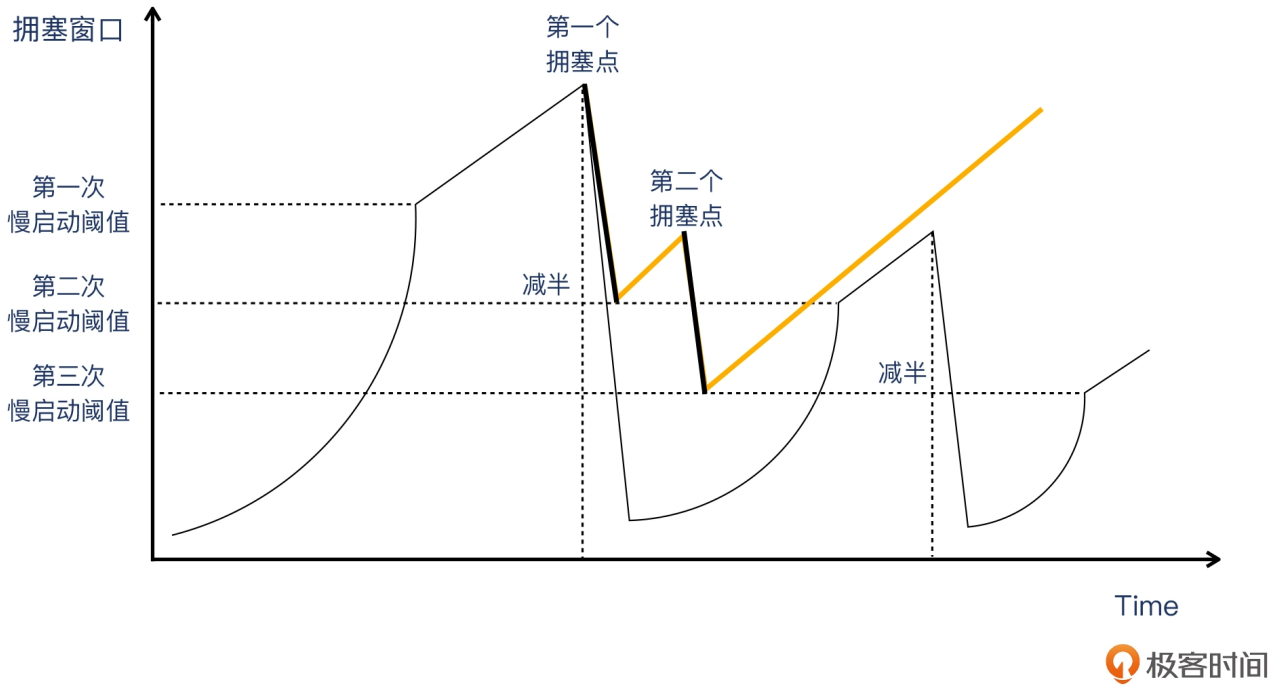
TCP 每发送一个报文，就启动一个超时计时器。如果在限定时间内没收到这个报文的确认，那么发送方就会认为，这个报文已经在网络上丢失了，于是需要重传这个报文，这种形式叫做**超时重传**。

一般来说，TCP 的最小超时重传时间为 200ms。这样的超时重传的机制虽然解决了丢包的问题，但也带来了一个新的问题：如果每次丢包都要等 200ms 或者更长时间，那应用不是就不能及时处理了吗？特别是对于有些时间敏感型的应用来说，影响更为严重。

所以，TCP 会用另外一种方式来解决超时重传带来的时间空耗的问题，就是用**快速重传**。在这个机制里，一旦发送方收到 3 次重复确认（加上第一次确认就一共是 4 次），就不用等超时计时器了，直接重传这个报文。

快速恢复

这是 TCP Reno 算法引入的一个阶段，它是跟随快速重传一起工作的。跟之前的“慢启动 -> 拥塞避免 -> 慢启动 -> 拥塞避免”这种做法不同的是，在遇到拥塞点之后，通过快速重传，就不再进入慢启动，**而是从这个减半的拥塞窗口开始**，保持跟拥塞避免一样的线性增长，直到遇到下一个拥塞点。你可以参考下面的图片来理解，橙色线就是快速恢复阶段：

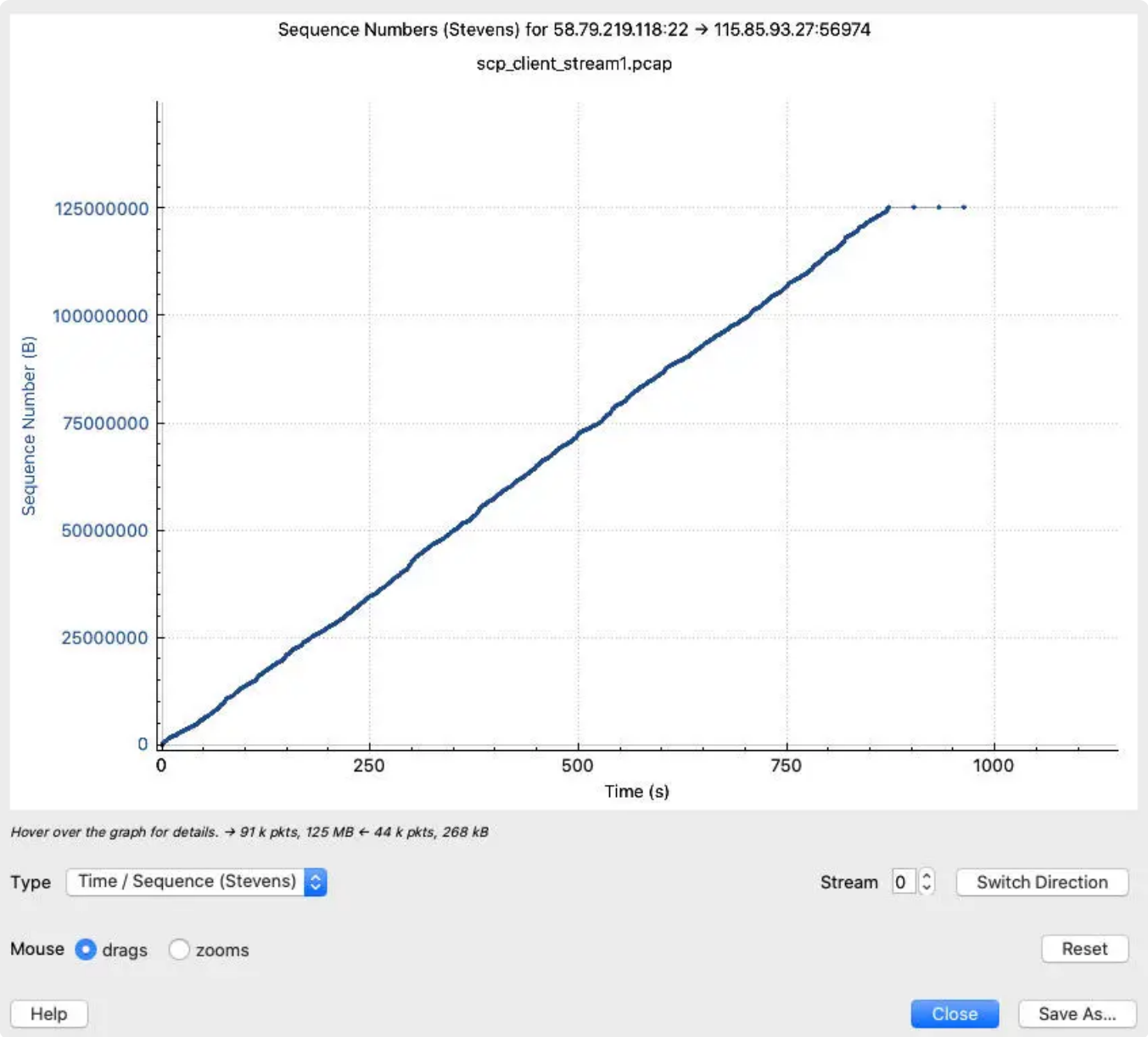


那么，是不是有了快速重传和快速恢复，传输过程中都不用反复进入慢启动了呢？其实并不是这样。如果遇到超时，也一样要回到慢启动阶段，重新开始。

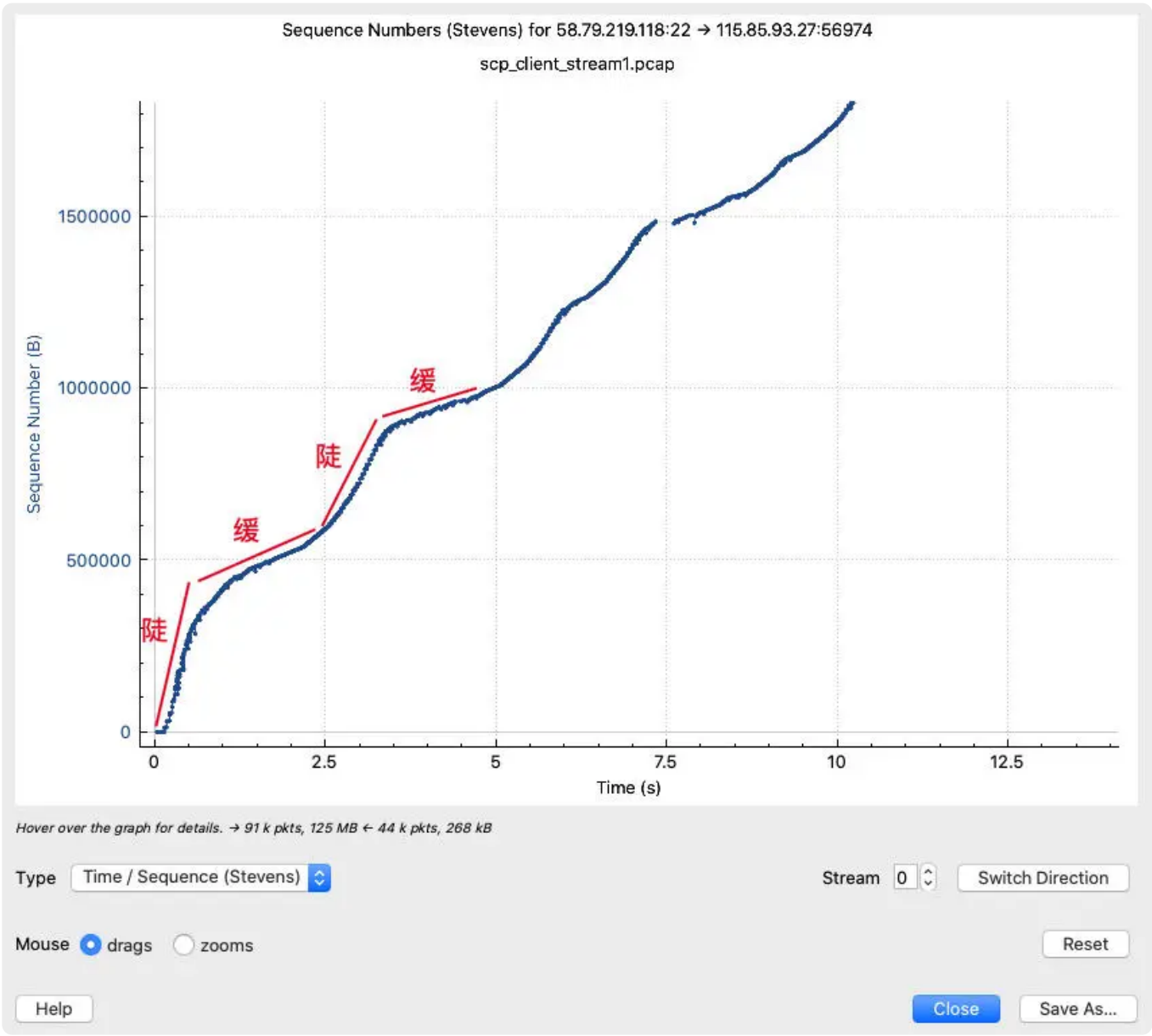
回到案例

好了，拥塞控制相关的知识点告一段落。这里，正好我们结合实际案例，来学习一下这部分知识。

在这次从北京传输文件到上海的过程中，有没有慢启动呢？要查看这个信息，用 Wireshark 的 TCP Stream Graphs 再合适不过了。我们打开 Statistics 菜单下的 TCP Stream Graphs -> Time Sequence (Stevens)，得到下图：



这里的斜率还是比较平稳的，说明虽然有很多乱序和重传，但整体速度还算稳定。当然，如果我们放大这条线，就能看到不同的景象了。比如放大到前 10 秒，明显曲线的波动幅度加大了不少：



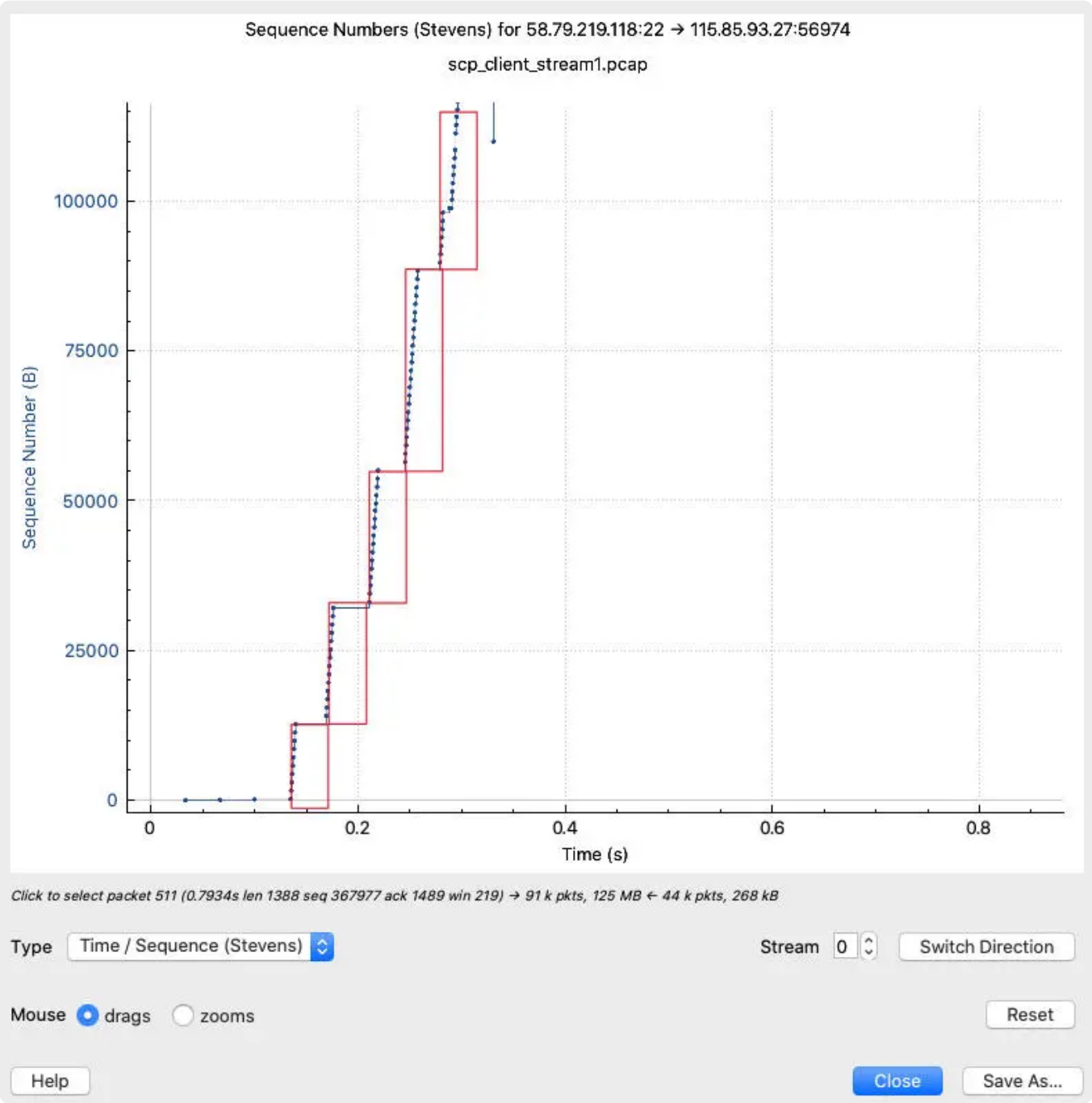
我们可以看到，在起点的时候，这条曲线的斜率是很高的，过了 0.5 秒以后开始平缓，然后到了 2.5 秒开始又陡峭了，然后大体上在循环着这个过程。这样看来，你可能会想：“陡的部分都是在慢启动吗？”

整体来说，确实可以这么理解。陡的部分是慢启动过程，所以斜率比较高；缓的部分是拥塞避免阶段，斜率比较低一些。

所以我们也发现，**慢启动不是只在 TCP 连接启动时候发生，而是可能在传输过程中发生多次。**前面的示意图里就显示，一旦拥塞避免阶段探测到了拥塞，TCP 还是会回到慢启动过

程，只是这次的慢启动阈值跟之前的不同。然后如果多次遇到拥塞，就会重复这个过程，直到传输结束。

而要查看最初的慢启动的过程（最初的慢启动也称为冷启动，Cold Start），还得继续放大。比如下图中，我们在 0~0.4 秒区间内，才终于明显找到它了。



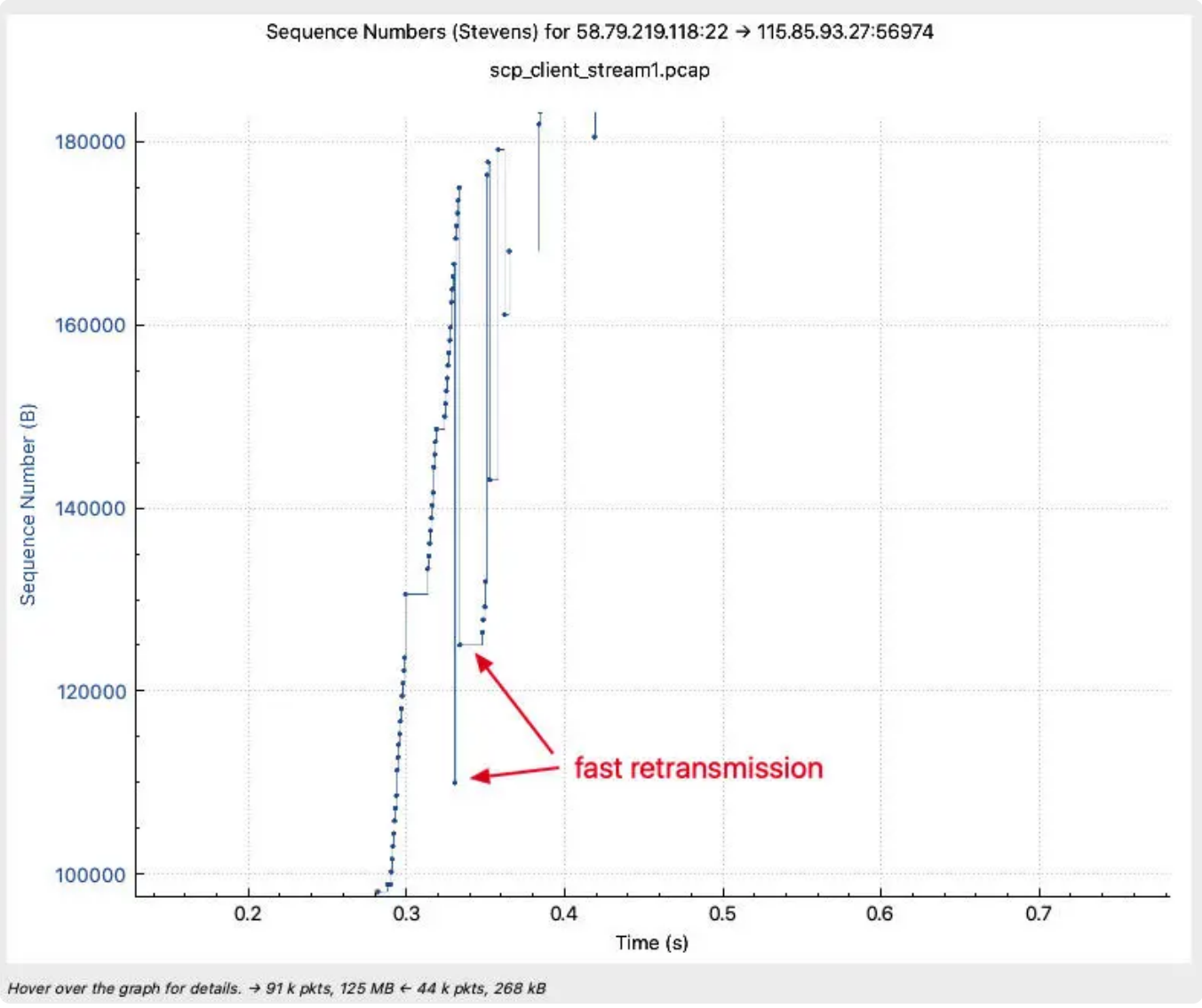
这里我用红色方框标注出了每一次往返时间（RTT），那么在每个 RTT 内，22 端口都连续发送了很多报文给到客户端，每次报文的序列号都是图上的一个点。显然，每个 RTT 后，发送的报文数量，都比前一轮 RTT 里发送的更多。

那你可能会有疑问：为什么这里的慢启动，并没有严格按照“翻倍”这样的原则来进行，比如说第二个红框的高度应该是第一个红框高度的两倍，但为什么图上并不是这样呢？

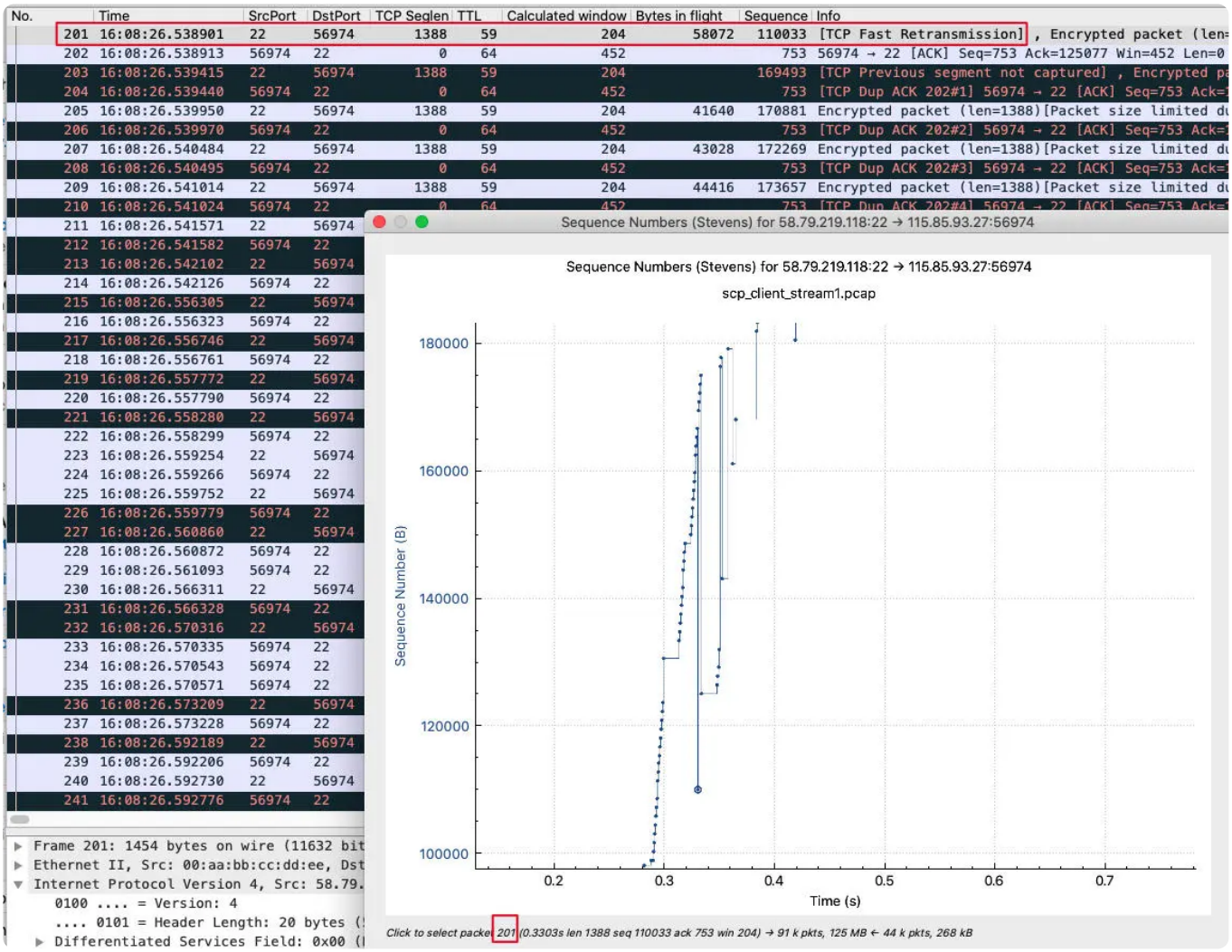
我个人的理解是这样的：在慢启动阶段，并不一定是“每次 RTT 就翻倍”，也可能会比翻倍低一些。这是因为，**在慢启动阶段，TCP 每收到一个 ACK，拥塞窗口就增加一个 MSS**。假设初始拥塞窗口是 2 MSS，发送 2 个数据报文后：

> 收到 1 个 ACK（也就是间隔确认），那么在这次 RTT 内，拥塞窗口就变成了 3 MSS，这是“不翻倍”的。
>
> 收到 2 个 ACK，那么在这次 RTT 内，拥塞窗口就变成了 4 MSS，这是“翻倍”的。

然后我们再看一下拥塞。在这个 Graph 中，也明显有几个点出现了异常。因为 Y 轴代表序列号，那么这些低处的点位，**它们的序列号自然就是比前一个更低（也就是更小）的值，说明这些是老的报文再次发了出去，所以可以判断为是重传。**



比如我们选中最下面这个点。这里的点都比较小，定位起来稍麻烦一些。你如果没有一点外科医生的手艺，那还是多一些耐心吧，慢慢移动鼠标才能定位到它。



定位到这个点之后，发现它是 201 号报文。在主窗口里，我们可以看到，201 号报文是一个 TCP Fast Retransmission，也就是快速重传报文。那么通过这个重传，TCP 拥塞控制机制就感知到了拥塞，然后进入了拥塞避免阶段。

好了，拥塞的知识点介绍得差不多了，案例也快结束了。那么案例的结论又是什么呢？

其实就是**专线上的限速设置失误**造成的。这个限速应该是通过网络硬件设备完成的，它单位时间内只允许一定字节数的报文通过，如果超过限制，就会丢弃这些报文。

现在我们学习了 TCP 拥塞控制机制，应该已经明白了：丢包对于发送端来说就是“拥塞”，然后它就根据拥塞控制机制，主动进入拥塞避免阶段，以确保传输速度，不至于大面积丢包。事实上就自动降低了速率，达到了我们想要的“限速”的效果。

我们的同事把这个限速设置值给搞错了，并不是客户购买的那个带宽值。修正之后就解决了问题。

虽然说这个排查很简单，不过通过这些抓包文件的拆解分析，还是可以对我们理解 TCP 拥塞控制的机制，有很大的帮助。

实验一下

拥塞控制机制对 TCP 传输十分重要，技术细节也很复杂，所以是由内核实现的。这也是内核实现 TCP 栈的巨大优势：应用程序可以集中于业务逻辑，而不需要操心传输和拥塞这种底层细节了。

那么，这是不是意味着，TCP 拥塞这个东西有点“高冷”，咱们也就看看，评论一下，好像啥都做不了？

并不是，接下来的实验，就是可以改变一些拥塞控制行为的。

实验 1：修改初始拥塞窗口

有些时候，我们也有修改拥塞行为的需求。比如，修改初始拥塞窗口（ICW）。这样，当你认为某条链路网络状况比较糟糕，用更低的 ICW 更合理时，就可以这么做。

在 Linux 操作系统上，修改初始拥塞窗口的方法是这样的：

运行`ip route`命令，找到当前的路由条目，把整行都进行复制，记为 item：

复制代码

```
$ ip route
default via 10.0.2.2 dev enp0s3 proto dhcp src 10.0.2.15 metric 100
```

运行`ip route change item initcwnd n`，把路由项 item 的初始拥塞窗口修改为 n，比如改为 2：

复制代码

```
$sudo ip route change default via 10.0.2.2 dev enp0s3 proto dhcp src 10.0.2.15
```

实验 2：改变 TCP 拥塞控制算法

有时候，我们还需要调整 TCP 拥塞控制算法。在 Linux 里，我们可以通过 **sysctl 命令**，查看或者修改这个算法。比如，Linux 默认是用 cubic 算法，那你可以运行下面这条命令，看看是不是这样？

复制代码

```
$ sysctl net.ipv4.tcp_congestion_control
net.ipv4.tcp_congestion_control = cubic
```

那如果你想用最新的 BBR 算法，该怎么做呢？如果内核大于 4.9，Linux 里就已经默认带有 BBR 了。执行下面的命令即可：

复制代码

```
$ sudo sysctl net.ipv4.tcp_congestion_control=bbr #配置拥塞算法为BBR
net.ipv4.tcp_congestion_control = bbr
$ sudo sysctl net.core.default_qdisc=fq           #调整缓存队列算法
net.core.default_qdisc = fq
```

> 补充：把这些配置写入到 /etc/sysctl.conf，即使机器重启，配置也会保持不变。

TCP BBR 拥塞控制算法是 Google 于 2016 年提出的新的算法。它的开发背景是，当今网络设备的缓存越来越大，导致丢包这个行为不像以前缓存小的时代那么频繁，但是报文延迟的问题比以前严重了。所以，要更加准确地探测拥塞，我们应该更多地关注延迟，并基于延迟的变化作出拥塞窗口的调整。

小结

这节课，我们学习了 TCP 传输中非常核心的一块内容：拥塞控制。拥塞控制的实现，主要依靠这几个环节。

慢启动：每收到一个 ACK，拥塞窗口（CW）增加一个 MSS。

拥塞避免：策略是“和性增长乘性降低”，每一个 RTT，CW 增加一个 MSS。

快速重传：接收到 3 次或者以上的重复确认后，直接重传这个丢失的报文。

快速恢复：结合快速重传，在遇到拥塞点后，跳过慢启动阶段，进入线性增长。

另外，我们也复习了拥塞窗口（CW）和接收窗口（RW）是如何决定了传输速度上限的，简单来说：

当 RW<CW 时，速度由 RW 决定；

当 RW>CW 时，速度由 CW 决定。

然后，我们也知道了拥塞窗口（CW）是每条连接分开各自维护的，以及初始拥塞窗口（ICW）的概念，并且知道，从 Linux 3.0 内核开始，ICW 已经提升到 10 个 MSS。

在 Wireshark 使用技巧上，你也要清楚如何用 **TCP Stream Graphs** 的 Time sequence (Stevens) 小工具，来观察慢启动和拥塞避免等现象，包括其中发生的快速重传等行为，都可以在图上看到，这非常有利于你的排查工作。

除此之外，我们也可以对拥塞控制做一些调整：

用`ip route change`命令，调整某个网卡接口的初始拥塞窗口。

用`sysctl net.ipv4.tcp_congestion_control`命令，查看和修改内核使用的拥塞控制算法。

思考题

你在工作中有没有遇到拥塞引起的问题，或者有没有在抓包分析过程中，观察到过拥塞现象呢？欢迎在留言区分享你的经验，我们一同成长、进步。

附件

抓包示例文件：https://gitee.com/steelvictor/network-analysis/tree/master/11

赞 1 提建议

上一篇 10 | 窗口：TCP Window Full会影响传输效率吗？

下一篇 12 | 重传的认识：重传到底是怎么回事？

精选留言 (5)

写留言

一子三木

2022-02-18

TCP 拥塞控制主要有四个重要阶段：
- 慢启动
慢启动是指 TCP 传输的开始阶段是从一个相对低的速度开始的。在这个阶段，每次 TCP 收到一个确认了数据的 ACK，拥塞窗口就增加一个 MSS。如果到达慢启动阈值塞窗口的增长速度立刻就放缓了，变成了每过一个 RTT，拥塞窗口就只增长一个 MS...

展开

作者回复: 很好的笔记和打卡，加油：)

webmin

2022-02-14

流量图上观察到"平顶山"就一个最明显的流量达限的情况，一但出口流量达到上限就容易引发相互踩踏的情况，传不出去，使用的人就更着急，就会点击更多次，或者多窗多设备重试，引发更多的拥塞。

作者回复: 嗯所以TCP就设计了拥塞避免这种机制，为了让网络使用尽可能的公平。生活中也是，不排队随便挤的话反而慢，排队来反而快。TCP拥塞控制也是类似，有了规矩，不随意争抢，大家的机会就会比较均等：)

江山如画

2022-02-14

老师，有个小问题，在慢启动的时候，每收到一个 ack 报文，拥塞窗口就加 1 个mss，不太清楚为什么拥塞窗口这个阶段是翻倍增长。 图上看慢启动阶段，第一次发送了1个报文，第二次发送了2个报文，第三次发送了4个报文，是因为慢启动阶段，每次都发出上一轮的2倍的报文数目，达到翻倍增长的目的么？

展开

作者回复: 这确实是一个应该关注的知识点。慢启动阶段的“拥塞窗口翻倍”是一种表现，但不是规定。规定是“每收到一个ack，拥塞窗口就增加一个MSS”。比如：
初始拥塞窗口为10MSS；假如每次都有足量的数据要发送，那么：
第一次往返：发出10个MSS, 收到10个Ack，此时拥塞窗口增长为10+10也就是20个MSS
第二次往返：发出20个MSS，收到20个Ack，此时拥塞窗口增长为20+20也就是40个MSS

上面这种就是“翻倍”，但注意，RFC不是说一定要在一个RTT里面翻倍，而是因为一个RTT里经常收到跟发送量同等数量的Ack，所以效果上是翻倍的。

共 9 条评论 >

Chao

2022-02-14

慢启动阈值 （ ssthresh ）的初始值是如何确定的，当慢启动过程中没有进入重传状态，如何进入拥堵避免？

共 1 条评论 >

1

那时刻

2022-02-14

请问老师两个问题
1. 拥塞避免，直到探测到拥塞，然后拥塞窗口就要往下降。请问如何探测拥塞呢？依据接收方的接收窗口与in flight数据大小吗？是不是还有其它方法？
2. 快速恢复，如果遇到超时，也一样要回到慢启动阶段，重新开始。请问这个超时时间是多少呢？可改配置吗？

展开

共 2 条评论 >