03 | 握手：TCP连接都是用TCP协议沟通的吗？

2022-01-17 杨胜辉

《网络排查案例课》 课程介绍 >

讲述：杨胜辉
时长 23:43 大小 21.73M

你好，我是胜辉。

在前面预习篇的两节课里，我们一起回顾和学习了网络分层模型与排查工具，也初步学习了一下抓包分析技术。相信现在的你，已经比刚开始的时候多了不少底气了。那么从今天开始，我们就要正式进入 TCP 这本大部头，而首先要攻破的，就是握手和挥手。

TCP 的三次握手非常有名，我们工作中也时常能用到，所以这块知识的实用性是很强的。更不用说，技术面试里面，无论是什么岗位，似乎只要是技术岗，都可能会问到 TCP 握手。可见，它跟操作系统基础、编程基础等类似，同属于计算机技术的底座之一。

握手，说简单也简单，不就是三次握手嘛。说复杂也复杂，别看只是三次握手，中间还是有不少学问的，有些看似复杂的问题，也能用握手的技术来解决。不信你就跟我看这几个

案例。

TCP 连接都是用 TCP 协议沟通的吗？

看到这个小标题，可能你都觉得奇怪了：TCP 连接不用 TCP 协议沟通还用什么呢？

确实，一般来说 TCP 连接是标准的 TCP 三次握手完成的：

1. 客户端发送 SYN；
2. 服务端收到 SYN 后，回复 SYN+ACK；
3. 客户端收到 SYN+ACK 后，回复 ACK。

这里面 SYN 会在两端各发送一次，表示“我准备好了，可以开始连接了”。ACK 也是两端各发送了一次，表示“我知道你准备好了，我们开始通信吧”。

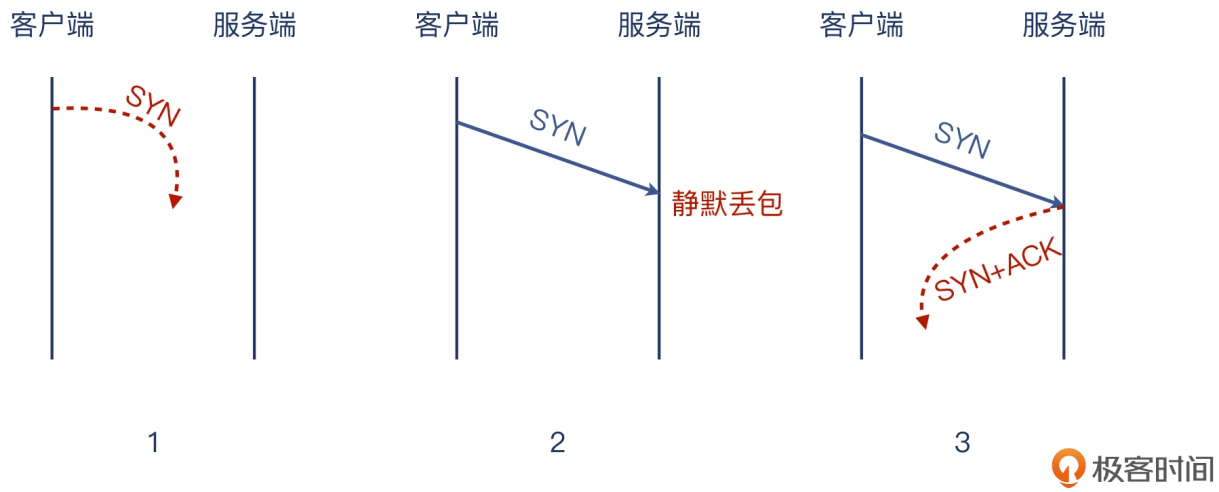
那既然是 4 个报文，为什么是三次发送呢？显然，服务端的 SYN 和 ACK 是合并在一起发送的，就节省了一次发送。这个在英文里叫 Piggybacking，就是背着走，搭顺风车的意思。

如果服务端不想接受这次握手，它会怎么做呢？可能会出现这么几种情况：

1. 不搭理这次连接，就当什么都没收到，什么都没发生。这种行为，也可以说是“装聋作哑”。
2. 给予回复，明确拒绝。相当于有人伸手过来想握手，你一巴掌拍掉，真的是非常刚了。

第一种情况，因为服务端做了“**静默丢包**”，也就是虽然收到了 SYN，但是它直接丢弃了，也不给客户端回复任何消息。这也导致了一个问题，就是客户端无法分清楚这个 SYN 到底是下面哪种情况：

1. 在网络上丢失了，服务端收不到，自然不会有回复；
2. 对端收到了但没回，就是刚才说的“静默丢包”；
3. 对端收到了也回了，但这个回包在网络中丢了。

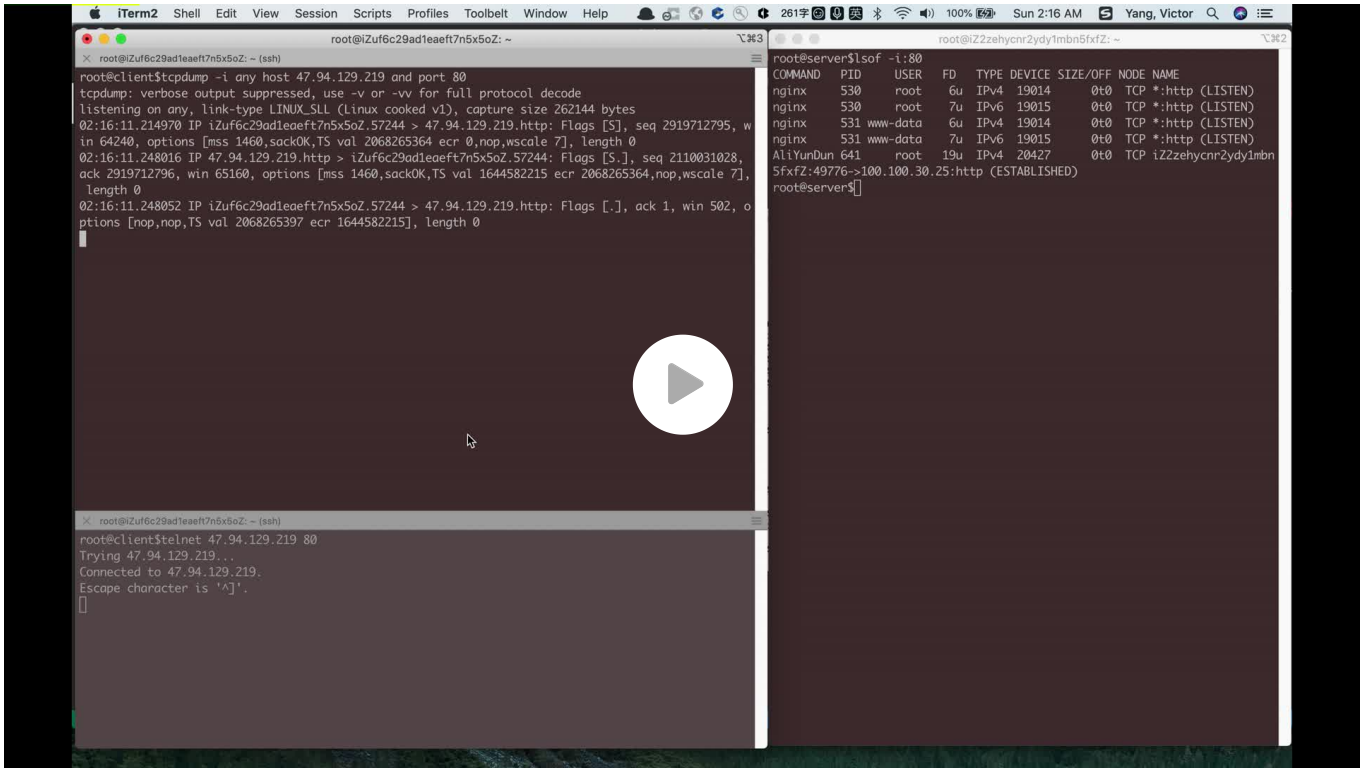


你看，就这么简单的一个 SYN，还能引申出三种状况出来。感觉什么东西一沾上网络，就要变成麻烦事啊。所以，跟我们在第 1 讲里学过的一样：设计网络协议真的不简单。

那么，从客户端的角度，对于 SYN 包发出去之后迟迟没有回应的情况，它的策略是做**重试**，而且不止一次。那会重试几次呢？重试多久呢？这个问题，一下子还不太好回答。不过，有 tcpdump 帮忙，我们可以搞清楚重试的问题，也可以搞清楚“TCP 连接是否都用 TCP 协议沟通”的问题。

动手实验

你可以借助 Iptables 和 tcpdump 做个实验，来验证这件事。你需要一台测试用的服务端，安装 Ubuntu 等 Linux 类系统，然后用你的笔记本作为客户端发起测试。这里我也放了一个视频，展示了这个实验过程，你可以结合着对照来看。



> 注意：在这个视频中，我是直接在 tcpdump 窗口里解读抓包结果的，而在下面我们是用 Wireshark 来解读，思路其实是一样的，只是操作方式略有不同，正好你可以都学习一下。

第一步，在服务端，执行下面的这条命令，让 Iptables 静默丢弃掉发往自己 80 端口的数据包：

```
1 Iptables -I INPUT -p tcp --dport 80 -j DROP
```

第二步，在客户端启动 tcpdump 抓包：

```
1 sudo tcpdump -i any -w telnet-80.pcap port 80
```

第三步，从客户端发起一次 telnet：

```
1 telnet 服务端IP 80
```

这个时候，这个 telnet 会挂起：

```
victor@victorebpf:~$ telnet 47.94.129.219 80
Trying 47.94.129.219...
```

大约一两分钟后才会失败退出，你随后就会明白背后发生了什么。

这时，你可以把客户端的 tcpdump 停掉了（按下 Ctrl+C）。然后用 Wireshark 打开这个抓包文件，看看里面是什么：

No.	Time	SrcPort	DstPort	TCP Seglen	Info
1	0.000000	32768	80	0	32768 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
2	1.010659	32768	80	0	[TCP Retransmission] 32768 → 80 [SYN] Seq=0 Win=64240 Len=0
3	2.015613	32768	80	0	[TCP Retransmission] 32768 → 80 [SYN] Seq=0 Win=64240 Len=0
4	4.255761	32768	80	0	[TCP Retransmission] 32768 → 80 [SYN] Seq=0 Win=64240 Len=0
5	8.192281	32768	80	0	[TCP Retransmission] 32768 → 80 [SYN] Seq=0 Win=64240 Len=0
6	16.127622	32768	80	0	[TCP Retransmission] 32768 → 80 [SYN] Seq=0 Win=64240 Len=0
7	33.024297	32768	80	0	[TCP Retransmission] 32768 → 80 [SYN] Seq=0 Win=64240 Len=0

telnet 挂起的原因就在这里：**握手请求一直没成功**。客户端一共有 7 个 SYN 包发出，或者说，除了第一次 SYN，后续还有 6 次重试。客户端当然也不是“傻子”，这么多次都失败，就放弃了连接尝试，把失败的消息传递给了用户空间程序，然后就是 telnet 退出。

这里有个信息很值得我们关注。第二列是数据包之间的时间间隔，也就是 1 秒，2 秒，4.2 秒，8.2 秒，16.1 秒，33 秒，每个间隔是上一个的**两倍**左右。到第 6 次重试失败后，客户端就彻底放弃了。

显然，这里的翻倍时间，就是“**指数退避**”（Exponential backoff）原则的体现。这里的时间不是精确的整秒，因为指数退避原则本身就不建议在精确的整秒做重试，最好是有所浮动，这样可以让重试成功的机会变得更大一些。

这里实际上也是一个知识点了：**TCP 握手没响应的话，操作系统会做重试**。在 Linux 中，这个设置是由内核参数 net.ipv4.tcp_syn_retries 控制的，默认值为 6，也就是我们前面刚观察到的现象。以下就是我的 Ubuntu 20.04 测试机的配置：

复制代码

```
1 $ sudo sysctl net.ipv4.tcp_syn_retries
```

```
2 net.ipv4.tcp_syn_retries = 6
```

还有另外好几个有关 TCP 重试的设置值，也都可以调整。更全面的内容呢，你可以直接 **man tcp**，查看 tcp 的内核手册的信息。比如下面就是对于 tcp_syn_retries 的解释：

复制代码

```
1 tcp_syn_retries (integer; default: 5; since Linux 2.2)
2        The  maximum  number of times initial SYNs for an active TCP connection
```

既然静默丢包会引起客户端空等待的问题，那我们直接拒绝，应该就能解决这个问题了吧？

正好，Iptables 的规则动作有好几种，前面我们用 DROP，那这次我们用 REJECT，这应该能让客户端立刻退出了。执行下面的这条命令，让 Iptables 拒绝发到 80 端口的数据包：

复制代码

```
1 Iptables -I INPUT -p tcp --dport 80 -j REJECT
```

跟前面的实验一样，我们在客户端发起 telnet 服务端 IP 80。果然，telnet 立刻退出，显示：

复制代码

```
1 $ telnet 47.94.129.219 80
2 Trying 47.94.129.219...
3 telnet: connect to address 47.94.129.219: Connection refused
4 telnet: Unable to connect to remote host
```

可见，连接请求确实被拒绝了。我在 telnet 同时也抓了包，我们来看一下抓包文件：

Apply a display filter ... <⌘/>

No.	Time	SrcPort	DstPort	TCP Seglen	Info
1	0.000000	52656	80	0	52656 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460

奇怪，抓包文件里并没有期望的 TCP RST？是我们抓包命令没写对吗？下面是这条命令，你已经初步学过 tcpdump 抓包命令了，看看有没有什么问题？

```
1 sudo tcpdump -i any -w telnet-80-reject.pcap host 47.94.129.219 and port 80
```

复制代码

命令语法没问题，要不然命令都无法执行。那过滤条件呢？指定了远端 IP 和端口，这是很常见的用法，应该也没什么问题。

但是，**这里隐藏了一个假设的前提**，也就是我们认为，这次握手的所有过程都是通过这个 80 端口进行的。但事实上呢？我们稍微改一下抓包条件，只保留远端 IP，去掉端口的限制：

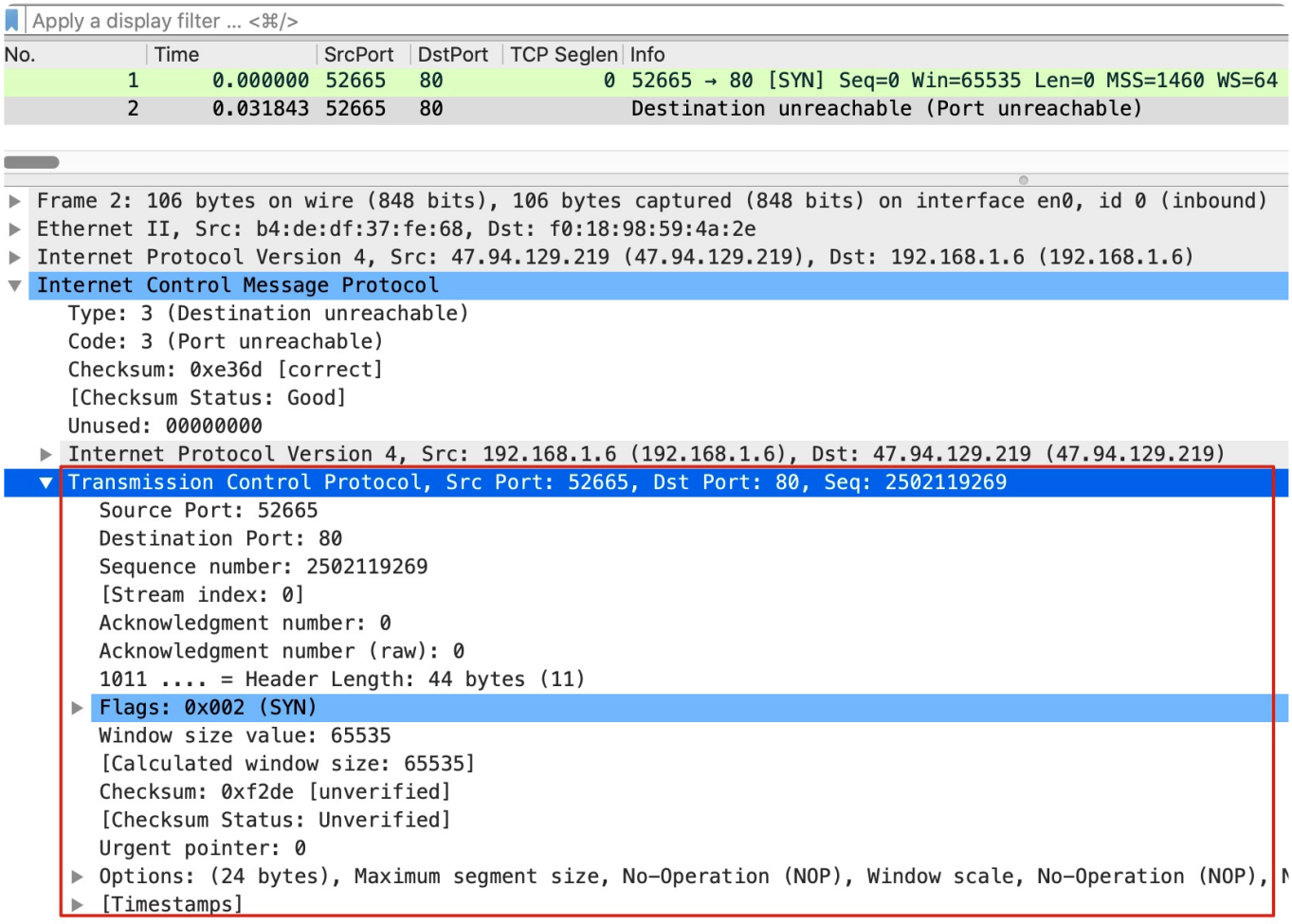
```
1 sudo tcpdump -i any -w telnet-80-reject.pcap host 47.94.129.219
```

复制代码

然后再来看看，我们抓到的报文是怎样的：

No.	Time	SrcPort	DstPort	TCP Seglen	Info
1	0.000000	52665	80	0	52665 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460
2	0.031843	52665	80		Destination unreachable (Port unreachable)

很意外，居然对端回复了一个 **ICMP 消息：Destination unreachable (Port unreachable)**。这还不是最意外的，我们选中这个报文，进一步看它的详情，可能会更惊讶：



原来，这个 ICMP 消息不仅通过 type=3 表示，这是一个“端口不可达”的错误消息，而且在它的 payload 里面，还携带了完整的 TCP 握手包的信息，而这个握手包，可是客户端发过来的。

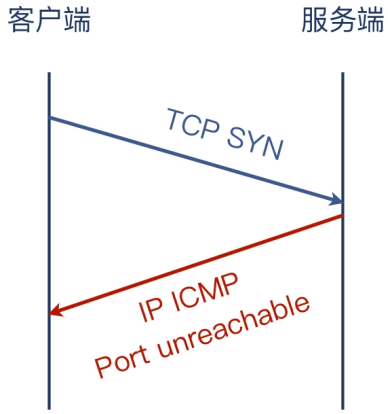
> 补充一下：如果我们回头再检查一下前面生成的 Iptables 规则，它是这样的：

复制代码

```
1 -A INPUT -p tcp -m tcp --dport 80 -j REJECT --reject-with icmp-port-unreachabl
```

> 原来，它自动补上了–reject-with icmp-port-unreachable，也就是说确实用 ICMP 消息做了回复。当然，你还可以把这个动作定义为–reject-with tcp-reset，那样的话就符合我们一开始的期望了。
>
> 事实上，无论是收到 TCP RST 还是 ICMP port unreachable 消息，客户端的 connect() 调用都是返回 ECONNREFUSED，这就是 telnet 都报“connection refused”的深层次原因。

所以，这个握手失败的情况终于搞清楚了，它是这么发生的：



极客时间

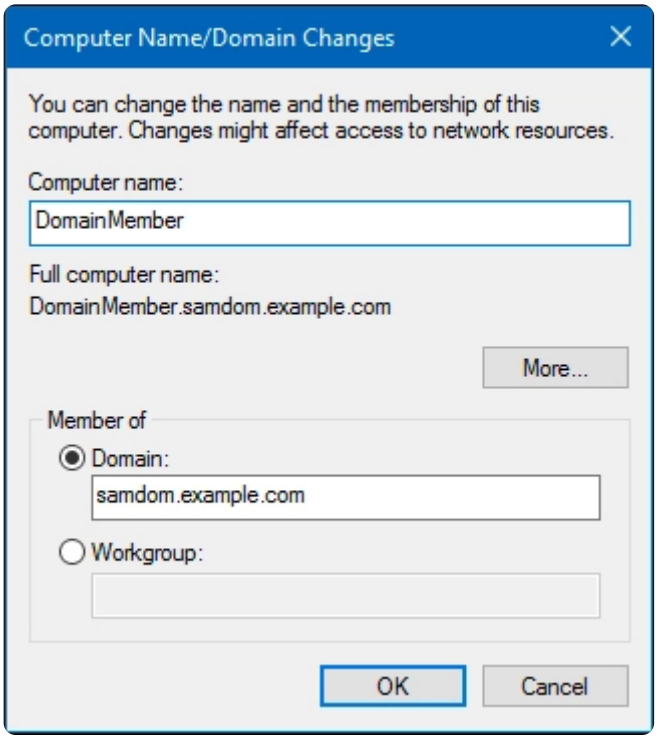
TCP 握手拒绝这个事，竟然可以是 **ICMP 报文**来达成的。“握手过程用 TCP 协议做沟通”，看起来这么理所当然的事情居然也会反转，你是不是也有点自我怀疑了：是不是其他网络知识，也未必是我自己认为的那样呢？

这个知识点，其实是几年前我在处理一个客户的 TCP 连接问题时遇到的。剧情么，前面已经给你“演”过一遍了。当时我也深感 TCP 的水太深，快没过脖子了，甚至有点喘不过气来……从此以后，我再也不敢小看任何知识点，同时也领教了 tcpdump 和 Wireshark 在网络分析方面的威力。有了这两个大杀器的帮助，我的网络水平提高很快。这个经验我也分享给你，相信你也一定能从中受益。

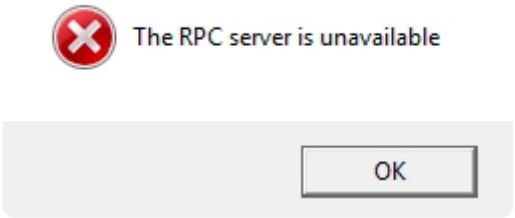
Windows 服务器加域报 RPC service unavailable?

虽然 tcpdump + Wireshark 的组合威力强大，但用起来总是会稍微花点时间。**有没有不用抓包分析，也能做排查 TCP 连接问题的方法呢？**这样也好快一点啊。接下来这个例子，就是这样的。

我们 eBay 也有不少 Windows 服务器，这些机器都由 Active Directory（简称 AD）管理。有一次，我们有一台 Windows 服务器加入 AD 失败，相关同事已经排查了好久，一直没找到原因。操作过程就是最普通的加域动作：



然后，一开始显示加域成功，但是过一两分钟后，又会来个“回马枪”，冒出来一个 The RPC server is unavailable 的报错：



在 Windows 的体系里面，这个报错大体意思是连不上 RPC 服务器。同事检查过 RPC 服务端并没有问题，然后其他 Windows 客户端加域呢，也都正常，唯独这台就不行。

单独一台机器加不了域，本身也不是特别大的麻烦，但是同事还是想找一下根因，于是就让我帮忙。很幸运，当时我只用了大概十分钟就找到了原因（这里我有点不谦虚了，我对你扔过来的鸡蛋和番茄表示接受）。

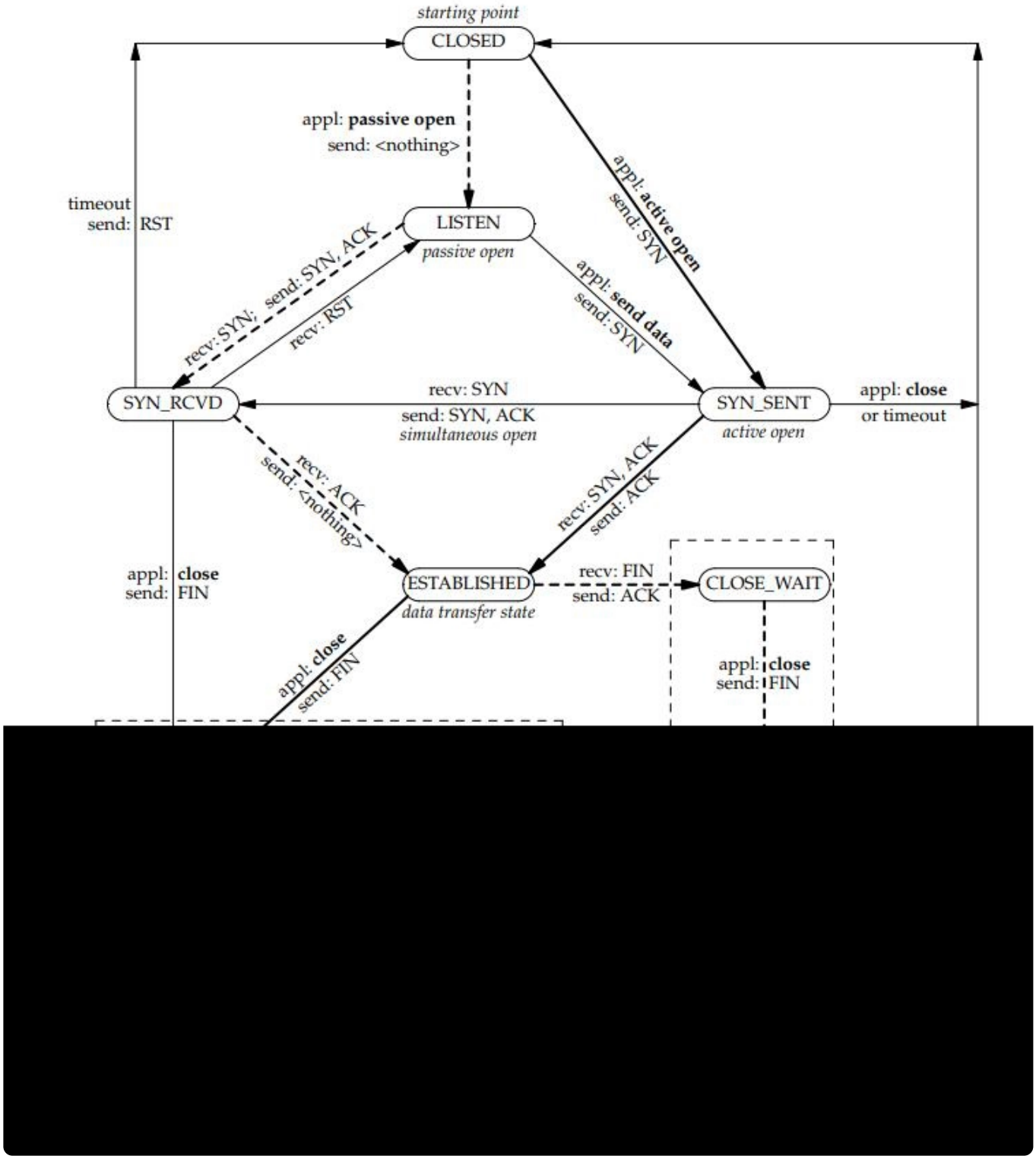
这倒不是我对 Windows 多么精通，**主要是正确的排查思路帮助了我**。给你分享一下我当时的思路：

1. 既然报错是 RPC unavailable，那可能意味着有一个 RPC 服务没有得到响应。
2. 没有得到服务端的响应，那多半是跟网络有关系，特别是跟端口的连通性有关系。

3. 要知道，RPC 使用的是动态端口，每次连接都可能连接到不同的服务端口。所以，我也**没办法预先知道是具体哪几个端口**，如果我知道的话，直接找防火墙团队去把那几个服务端口打开就好了，但这个做不到。这一点也是同事卡了许久的原因之一，他也不知道如何找到这些“动态会变的 RPC 端口”。
4. 要找到**实时在用**的动态 RPC 端口，最方便的方法就是**运行 netstat 命令**。无论连接是处在什么状态，比如是在传输数据的 ESTABLISHED 状态、新近关闭端口的 TIME_WAIT 状态，都可以用 netstat 命令看到。
5. 我运行了 netstat，在当时的命令输出中，我注意到有一个 **SYN_SENT 状态**的连接，它要连的就是服务端的一个高端口。

那么，这个 SYN_SENT 状态究竟说明了什么呢？

SYN_SENT 是 TCP 的 11 个状态之一。要理解 SYN_SENT 的含义，我们首先要把整个 TCP 状态机的机制搞清楚。关于 TCP 状态机，目前流传比较广的是下面这张图。我没有考证过这张图的出处，不过在 Stevens 的《UNIX 网络编程：套接字联网 API》里就有这张图，很有可能最早就是来自于 Stevens：



这张图浓缩了 TCP 状态转换的所有知识点，确实值得反复研读。不过，我鸡蛋里挑个骨头：这张图也有个小小的问题，就是对于初学者来说，它并不容易理解。

比如，多年前我自己在学习 TCP 的时候，就一直没有彻底看懂这张图。好笑的是，我经常假装自己看懂了，还拿这张图跟别人侃侃而谈，而对方还被我唬住了呢。所以你也要学会了：当大家都不是很懂的时候，你对自己的话越相信，你就越有说服力哦。

好了，当然是跟你开个玩笑，做学问还是要严谨。那么，这张图的难点在哪呢？我觉得主要是视角不固定，一会是发送方，一会是接收方，对初学者来说很容易混淆。实际上，在 Stevens 的这本书中，还有另外一张图，我认为更加清晰明了，也是我想推荐给你的：

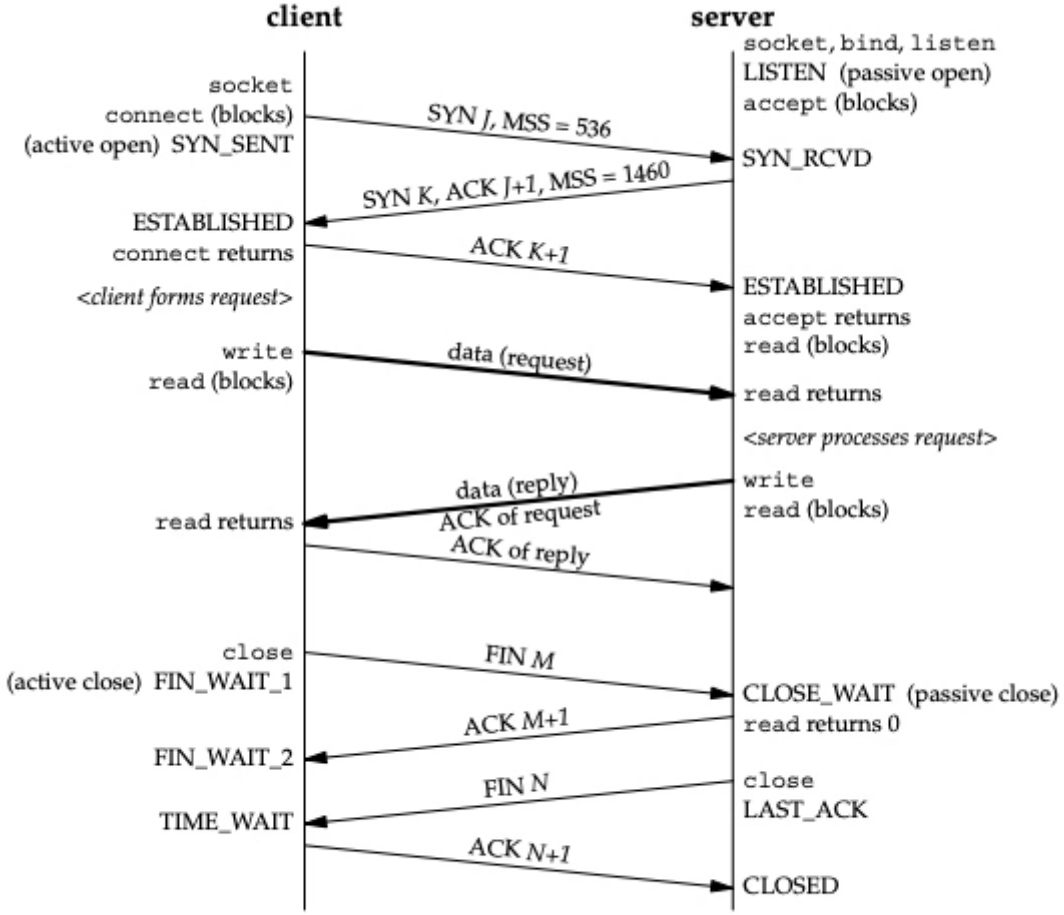


Figure 2.5 Packet exchange for TCP connection.

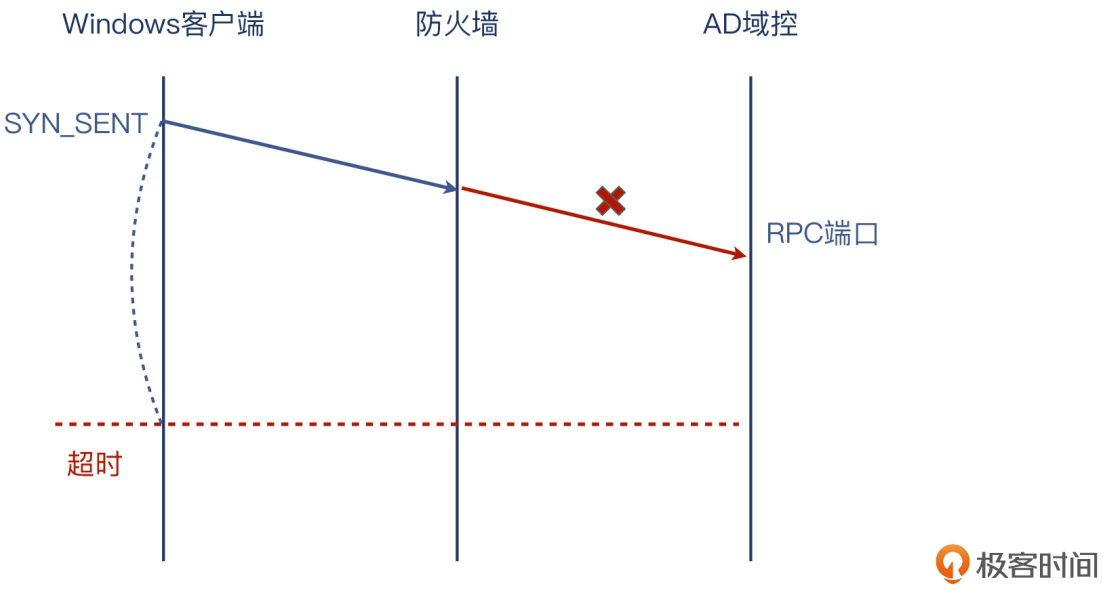
在上面这张图里，无论是客户端或者服务端，我们从上往下看，它要经历的各个 TCP 状态，都展示得十分清楚。我把这个过程解读如下：

客户端	服务端
1. 发送完SYN，客户端就进入了**SYN_SENT**状态。	
	2. 服务端收到了这个SYN之后，就进入了**SYN_RCVD**状

后续的过程，不用我继续解读，你也会看得很清楚了：**分别沿着左边和右边的垂直线从上往下看，就经历了客户端和服务端的 TCP 生命周期里的各种状态**，这个过程中，视角保持一致。你觉得是否比前面那张转换图，更加容易理解呢？

看懂了这张图，你应该就明白了：SYN_SENT 这个状态，意味着当时这个连接请求（SYN 包），已经从这台 Windows 服务器发出，试图跟远端的 AD 域控制器进行连接。但由于对端迟迟没有回应 SYN+ACK 报文，那么客户端这个连接的状态，就只能“停留”在 SYN_SENT 状态，无法转化为 ESTABLISHED 状态。

等到达了 SYN timeout 时间后，Windows 操作系统会放弃这次连接，而这个 SYN_SENT 状态的连接也会消失不见。所以，前面提到的“**实时**”两字，也是很关键的。如果不是在问题发生时运行 netstat，哪怕是过了几分钟再去运行 netstat，错过了这个 SYN_SENT，我也不能发现这个失败的 TCP 连接企图，也就无法定位到真正的原因了。



然后我们拿着这个端口去找防火墙团队，对方检查了配置，发现这个端口确实是禁止的。在开通后，问题就解决了。

所以说，**真的不要小看任何知识点和小工具，你掌握以后，完全可以起到关键性的作用**（对了，排查防火墙也时常是我们工作的痛点，我在第 5 和第 6 讲会专门讲解这方面的排查技巧，敬请期待）。

这里还有一个技术点我想给你展开一下。我们在前面已经讨论过了 SYN 重试的问题，显然，这次 Windows 的 SYN_SENT 的背后，我们相信，应该也是有数次的 SYN 重试的情况。同时，因为我观察到，这个 SYN_SENT 停留了大约有十几二十秒，所以我判断应该也有指数退避的存在，所以这个状态才保留了那么长时间。

也就是说，无论是 Linux 还是 Windows，都实现了类似的 TCP 握手方面的容错手段。还是那句话：设计网络不容易。**理解了设计者的初心**，很多问题就不会那么模糊了，可能你一下子就能看清。

发送的数据还能超过接收窗口？

最后一个案例表面上并不直接跟握手相关，但背地里就……不剧透了，看剧情。

前段时间，有个朋友找到我咨询一个问题。他们最近处理了一个 Redis 相关的技术问题，让他们既开心又“闹心”。开心的是整体分析是正确的，问题也得以解决；“闹心”的是，唯独有个技术点好像无法自圆其说，所以想让我看看到底是怎么回事。

这个问题是：Redis 服务告诉客户端它的接收窗口是 190 字节，但是客户端居然会发送 308 字节，**大大超出了接收窗口**。下图是他们用 Wireshark 打开抓包文件后的界面：

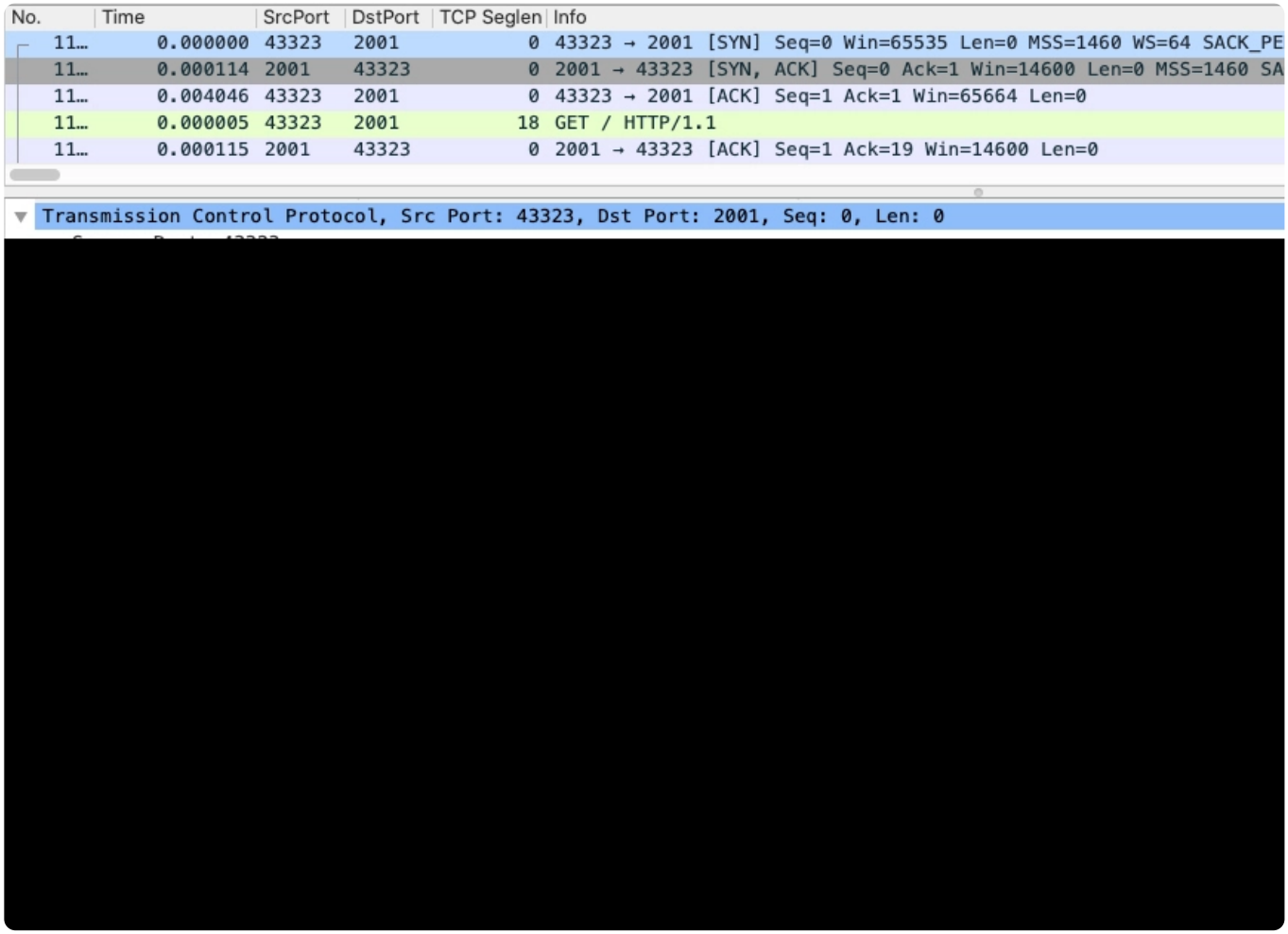
我一开始也懵了：难道 TCP 的深水又到我脖子这儿了？在我多年的抓包分析经历中，数据超过接收窗口的情况，好像还没有遇到过，这次算是 TCP 准备再次让我“开开眼”吗？

不过我很快又稳定了下来，因为我想到了一个朋友他们没有注意到的细节。在说到 TCP 窗口的时候，一般都会提到一个很重要的概念：**Window Scale**。这是因为，TCP 最初是七八十年代的产物，1981 年 9 月定稿的RFC793才第一次正式确定了 TCP 的标准。当时的网络带宽还处于“石器时代”，机器的带宽只有现在的百分之一，那么 TCP 接收窗口自然也没必要很大，2 个字节长度代表的 65535 字节的窗口足矣。

但是后来网络带宽越来越大，65535 字节的窗口慢慢就不够用了，于是设计者们又想出了一个巧妙的办法。原先的 Window 字段还是保持不变，在 TCP 扩展部分也就是 TCP Options 里面，增加一个 Window Scale 的字段，它表示原始 Window 值的右移位数，最高可以右移 14 位。

如果你还没有完全忘记计算机课的基本知识，那么应该明白这是一个非常大的提升了（扩大了 2 的 14 次方，即 16384 倍）。16384 乘以 65535，这个数字就是 1G 字节，也就是说，一个启用了 Window Scale 特性的 TCP 连接，最大的接收窗口可以达到 1GB。可以说，这个数字至今都是够用的。

说了这么多，我们用 Wireshark 来看看它究竟长啥样。选中一个 SYN 报文，在 Wireshark 窗口中部找到 TCP 的部分，展开 Options 就能看到了：



我们逐一理解下。

Kind：这个值是 3，每个 TCP Option 都有自己的编号，3 代表这是 Window Scale 类型。

Length：3 字节，含 Kind、Length（自己）、Shift count。

Shift count：6，也就是我们最为关心的窗口将要被右移的位数，2 的 6 次方就是 64。

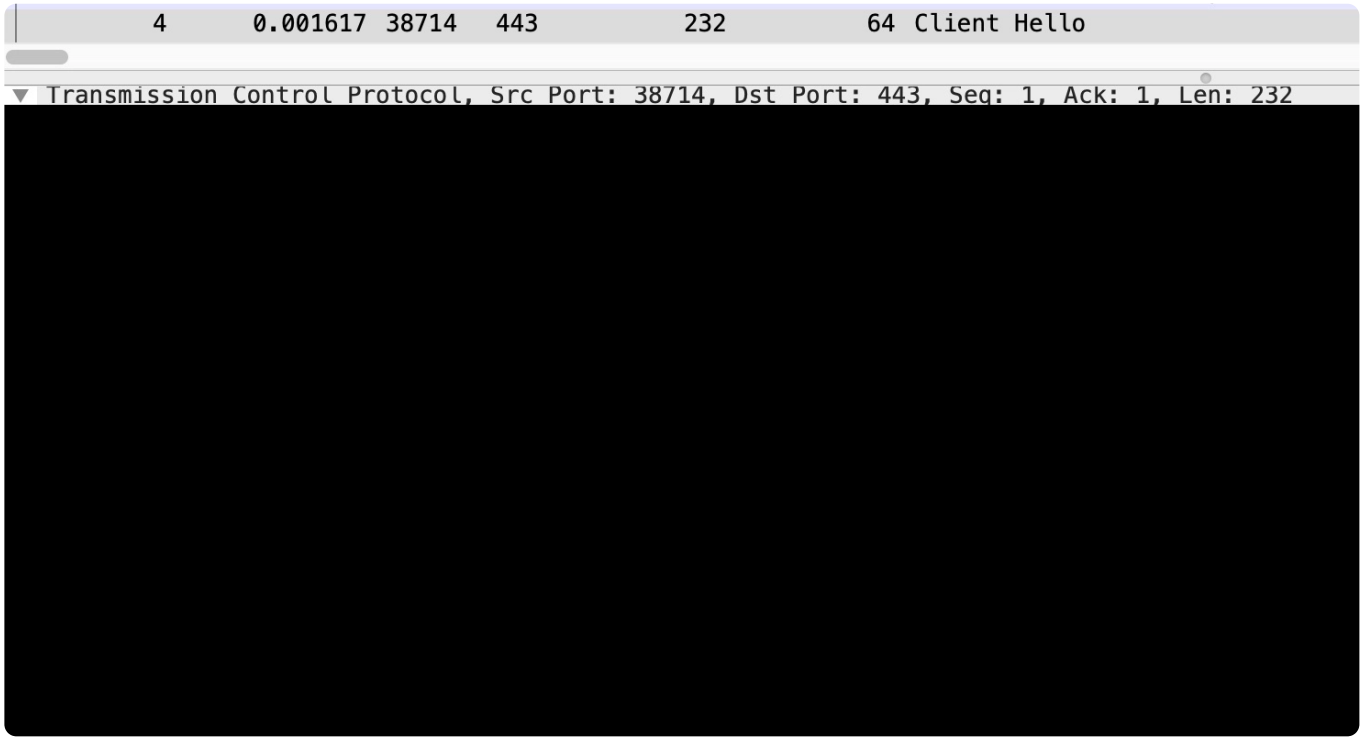
> 小小提醒：SYN 包里的 Window 是不会被 Scale 放大的，只有握手后的报文才会。

当然，TCP 的窗口也是 TCP 知识体系里一块挺大的分支领域，我会在当前这个“实战一”模块的传输效率部分，也就是第 9~11 讲里，详细讲解这方面的知识，帮你把这块的东西真正搞透。

回到握手。既然 Window Scale 这么有用，那每个 TCP 报文应该都是带上这个信息的吧，因为它在 TCP 头部里面嘛，而每个 TCP 报文都有头部的，不是吗？

你要这样想就错了。事实上，Window Scale 只出现在 TCP 握手里面。你再想想就明白了：这个是“控制面”的信息，说一次让双方明白就够了，每次都说，不光显得“话痨”，也很浪费带宽啊。一般传输过程中的报文，完全不需要再浪费这 3 个字节来传送一个已经同步过的信息。所以，握手之后的 TCP 报文里面，是不带 Window Scale 的。

比如，我们来看一个传输中的报文，比如客户端发送的 TLS Client Hello 报文：



可见，原始窗口 502 字节，放大 128 倍后就是 64256 字节了。

说到这里，想必你已经明白了：我朋友这次的疑惑，其实就是**缺少 TCP 握手包**造成的。要知道，Wireshark 也一样要依赖握手包，才能了解到这次连接用的 Window Scale 值，然后才好在原始 Window 值的基础上，对 Window 值进行右移（放大），得出真正的窗口值。于是，因为这次他们的抓包没有抓取到握手报文，所以 Wireshark 里看到的窗口，就是 190 字节，而不是 190 字节的某个倍数了！

当时通信的另一端当然知道这个信息，所以它发送 308 字节一点都不意外，因为这个值根本就没超出接收窗口。

那么，**是不是没有抓取到握手包的话，Wireshark 里读取到的 Window 就一定不对呢？**大部分时候是这样的。不过，还有一部分老系统的 TCP 栈并没有启用 Window Scale，那么抓包文件中有没有握手包都没关系，只要看基本 Window 就好了。

说到这里，你对 TCP 握手的印象，是不是又有改变呢？它简单，也丰富；它靠谱，也调皮。你只有真的读懂它，才不会被它牵着鼻子走。而读懂它的方法是什么呢？

就是多读些 TCP 理论，就是多做些抓包分析，就是多处理些案例，更是多走走，多看看。只要有心，你总有机会可以学会，可以成长。

小结

作为这个模块的第一课，这次我们围绕 TCP 握手展开了几个有趣的案例，并从中梳理了以下知识点：

1. 客户端发起的连接请求可能因为各种原因没有回复，这时客户端会做**重试**。一般在 Linux 里，重试次数默认是 6 次，内核参数是 net.ipv4.tcp_syn_retries。重试间隔遵循了**指数退避原则**。
2. 服务端拒绝 TCP 握手，除了用 TCP RST，另外一种方式是通过 **ICMP Destination unreachable（Port unreachable）消息**。从客户端应用程序看，这两种回复都属于“对端拒绝”，所以应用表面看不出区别，但我们在抓包的时候要注意，如果单纯抓取服务端口的报文，就会漏过这个 ICMP 消息，可能对排查不利。
3. 对于连通性相关的问题，除了用 tcpdump+Wireshark 这个黄金组合，我们还可以在理解 TCP 握手原理的基础上，使用小工具（比如 netstat）来排查。特别是对于 RPC 服务场景，在问题发生时及时**执行 netstat -ant，找到 SYN_SENT 状态的连接**，这个很可能是突破口。
4. 我们也学习了如何在 Wireshark 中查看 Window Scale。握手包中的 Window Scale 信息十分重要，这会帮助我们知道正确的接收窗口。在分析抓包文件时，**要注意是否连接的握手包被抓取到**，没有握手包，这个 Window 值一般就不准。

可以说，**应用都靠连接，连接都靠握手。**掌握好了握手，你的 TCP 就算入门了。学完这节课之后，你有没有觉得，今天的你比昨天的你，要强一些了呢？加油！后面更多的知识在等你来发现。

思考题

最后，还是按照惯例，还是给你留几道思考题：

1. 在 Linux 中，还有一个内核参数也是关于握手的，net.ipv4.tcp_synack_retries。你知道这个参数是用来做什么的吗？
2. 如果握手双方，一方支持 Window Scale，一方不支持，那么在这个连接里，Window Scale 最终会被启用吗？你可以参考 RFC1323，给出你的解答。

欢迎在留言区分享你的答案，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。

扩展知识：聊聊几个常见误区

很多时候，我们的成长不仅是由于学到了正确的知识，更是由于纠正了“错误的认知”。下面列几个常见误区，你看看自己有没有“中招”。

UDP 也有握手

有些同学会有这个误解，可能是跟 nc 这个命令有关。我们来看一个 TCP 端口 22 的测试：

复制代码

```
victor@victorebpf:~$ nc -v -w 2 47.94.129.219 22
Connection to 47.94.129.219 22 port [tcp/ssh] succeeded!
```

同一时间的 tcpdump 抓包，显示这个 TCP 经历了成功的握手和挥手：

复制代码

```
$ sudo tcpdump -i any host 47.94.129.219
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on any, link-type LINUX_SLL (Linux cooked v1), capture size 262144 b
11:58:10.749372 IP victorebpf.51072 > 47.94.129.219.ssh: Flags [S], seq 966857
11:58:10.781734 IP 47.94.129.219.ssh > victorebpf.51072: Flags [S.], seq 31701
11:58:10.781880 IP victorebpf.51072 > 47.94.129.219.ssh: Flags [.], ack 1, win
11:58:10.782344 IP victorebpf.51072 > 47.94.129.219.ssh: Flags [F.], seq 1, ac
11:58:10.782586 IP 47.94.129.219.ssh > victorebpf.51072: Flags [.], ack 2, win
11:58:10.821202 IP 47.94.129.219.ssh > victorebpf.51072: Flags [P.], seq 1:42,
11:58:10.821292 IP victorebpf.51072 > 47.94.129.219.ssh: Flags [R], seq 966857
```

如果我们用 nc 测试 **UDP** 22 端口，看看会发生什么。注意，UDP 22 是没有服务在监听的。但是 nc 一样告诉我们 succeeded！这似乎在告诉我们，这个 UDP 22 端口确实是在监听的：

复制代码

```
$ nc -v -w 2 47.94.129.219 22
Connection to 47.94.129.219 22 port [tcp/ssh] succeeded!
victor@victorebpf:~$ nc -v -w 2 47.94.129.219 -u 22
Connection to 47.94.129.219 22 port [udp/*] succeeded!
```

同一时间的抓包，显示客户端发送了 4 个 UDP 报文，但服务端没有任何回复：

复制代码

```
11:59:05.605556 IP victorebpf.54145 > 47.94.129.219.22: UDP, length 1
11:59:05.605995 IP victorebpf.54145 > 47.94.129.219.22: UDP, length 1
11:59:06.606436 IP victorebpf.54145 > 47.94.129.219.22: UDP, length 1
11:59:07.607134 IP victorebpf.54145 > 47.94.129.219.22: UDP, length 1
```

从表象上看，nc 告诉我们：这个跟 UDP 22 端口的“连接”是成功的，这是 nc 的 Bug 吗？可能并不算是。原因就在于，**UDP 本身不是面向连接的**，所以没有一个确定的 UDP 协议层面的“答复”。这种答复，需要由调用 UDP 的应用程序自己去实现。

那为什么在这里，nc 还是要告诉我们成功呢？可能只是因为对端没有回复 ICMP port unreachable。nc 的逻辑是：

> 对于 UDP 来说，除非明确拒绝，否则可视为“连通”；
>
> 对 TCP 来说，除非明确接受，否则视为“不连通”。

所以，当你下次用 nc 探测 UDP 端口，不通的结果是可信的，而能通（succeeded）的结果并不准确，只能作为参考。

一台机器最多 65535 个 TCP 连接

这也是很常见的误区了。我还是小白的时候，也曾经深信不疑。当时读到一篇讨论服务器可以承受多少 TCP 连接（就是 C10k 问题）的文章时，还觉得奇怪，不是端口范围只有

0~65535 吗？为什么还会有几十万上百万连接呢？

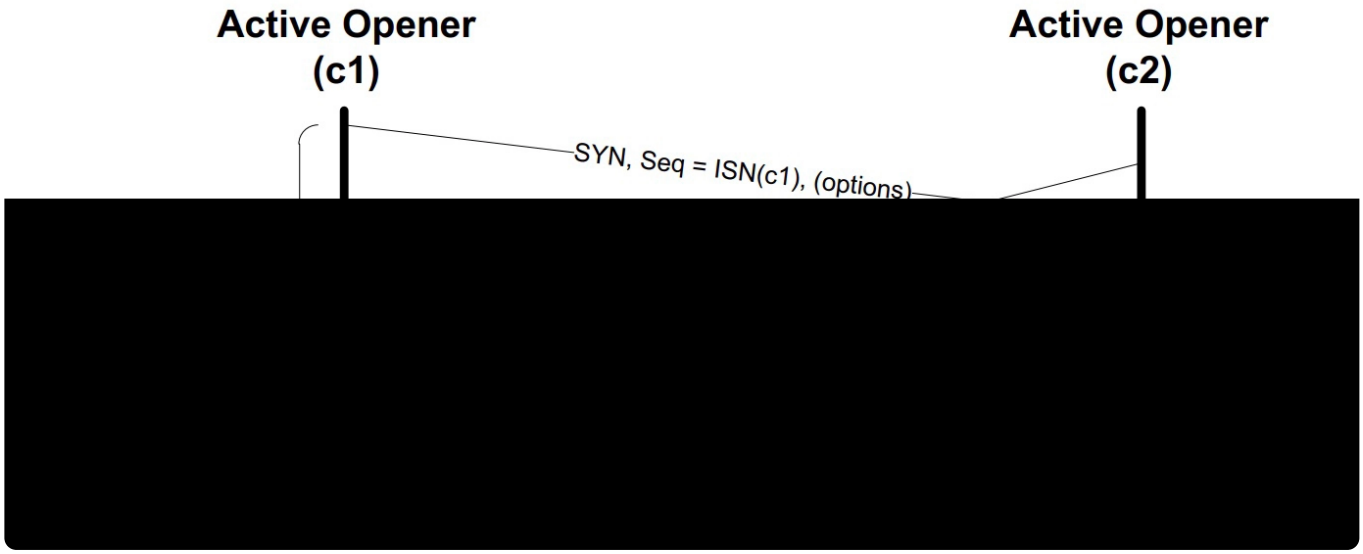
这就是没有意识到，连接是四元组（咱们在第一节课讲到过），并不是单纯的源端口或者目的端口。那么多个数相乘，这个乘积当然可以远远超过 65535 了。先不谈论海量级网站的场景，就算我们维护一台 Web 服务器，假如当前有 10 万台客户端连着你，平均每个客户端跟你有 6 个连接（这很常见），那么就是 60 万个连接了，是不是也早就超过 6 万了？

当然，在限定场景下，一个客户端（假设只有一个出口 IP）和一个服务端（假设也只有一个 IP 和一个服务端口），那么确实只能最多发起 6 万多个连接。但你自己也已经明白，这跟前面的误解，已经是两回事了。

不能同时发起握手

如果两端同时发送了 SYN 给对方，也就是双方都收到了一个 SYN，那么接下来，它们会进入什么状态呢？你可能觉得这应该不行。

其实，通信双方还真的可以同时向对方发送 SYN，也能建立起连接。你可以参考这节课里我提到的 **TCP 状态转换图**。在 Richard Stevens 的《TCP/IP 详解（第一卷）》里，也提到了这个知识点，参考下图：



当然，这种情况是很罕见的，你可以参考一下，也丰富一下你对 TCP 握手的理解。

赞 2 提建议

上一篇 02 | 抓包分析技术初探：你会用tcpdump和Wireshark吗？

下一篇 04 | 挥手：Nginx日志报connection reset by peer是怎么回事？

精选留言

写留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。