

这一部分只有两篇文章，但是篇幅都比较长。第一篇介绍了我主导开发的一个网络性能分析网站，也许它的功能不是你需要的，但是开发过程可以参考。比如说，你也可以利用 `tshark` 命令开发一个监控上网记录的工具，用 `"tshark -r(file_name) -Y "http.request.full_uri" -T fields -e http.request.full_uri"` 一句命令就可以生成原始数据，然后再编程做二次分析。第二篇介绍了网络加速器，现在才创业做这个显然太晚了，不过 `Wireshark` 很适合用来分析加速器的很多知识点，在实际中也大有用武之地。

Wireshark 好不好？当然好，几乎称得上业界最好，否则我也不会为它写了两本书。不过话说回来，再好的工具也有改进的空间，比如我能看到的不足之处就有两点。

- 对于特定职业的人群来说，Wireshark 的很多功能是完全用不到的。比如同一个公司的开发团队和运维团队，说起来都在用 Wireshark，但实际上使用的是完全不同的功能。初学者上手时根本不知道哪些功能适合自己的工作，不得不在探索上浪费很多时间。
- 每个人常用的功能就那么几个，却分布在不同的菜单里，有些还藏得很深。比如要查看 NFS 的读写响应时间，需要点五次鼠标才能找到，初学者根本记不住。

有没有办法“定制”一个分析工具，只提供我感兴趣的功能，而且简单到一键就能完成分析呢？也许在工业 4.0 时代会有这个服务，不过在此之前，我们只能自己开发了。今年我就和同事做了一个，本文会详细地加以介绍，希望对你有些参考价值。

我们的项目需求是这样的。

- 我司有很多团队需要和网络打交道，比如虚拟化、云计算、网络存储、镜像和备份等。大多数网络问题都很好解决，但性能问题却是公认的难点。
- 我司的这些团队成员都具备网络基础知识，比如熟读《TCP/IP 详解 卷 1：协议》，但是缺乏网络包分析技能，也没有时间学习 Wireshark。

假如有一个专门的工具来分析网络性能，生成的分析报告也简单易懂，肯定

会大受欢迎的。我期望这个工具能好用到什么程度？无需任何培训，只要丢个网络包进去，一份人人可以读懂的分析报告就出来了。考虑到这些团队在地理上非常分散（住在不同国家），行政上也属于不同部门，我决定把这个工具做成 Web 的形式，以便推广和维护。接下来就通过一个真实的案例，演示一下它究竟有多好用。

案例症状

用户抱怨某系统运行起来非常慢，这个系统的功能是处理一些网络存储上的数据。

排查过程

- 1. 把一些要处理的数据复制到该系统所在的本地硬盘，运行速度就上去了，说明该系统本身没有问题。
- 2. 网络工程师经过一系列检查，在网络上没有发现任何问题。
- 3. 存储工程师看到存储的响应非常快，所以也没有发现问题。

每一方都号称自己没有问题，那用户该怎么办？最后只好抓了个包，上传到我们的工具上分析。图 1 就是该工具的首页，它的全称为 Network Performance Analyzer，简称 NPA。用户唯一需要做的就是将网络包拖进方框，然后点一下 Upload 按钮。

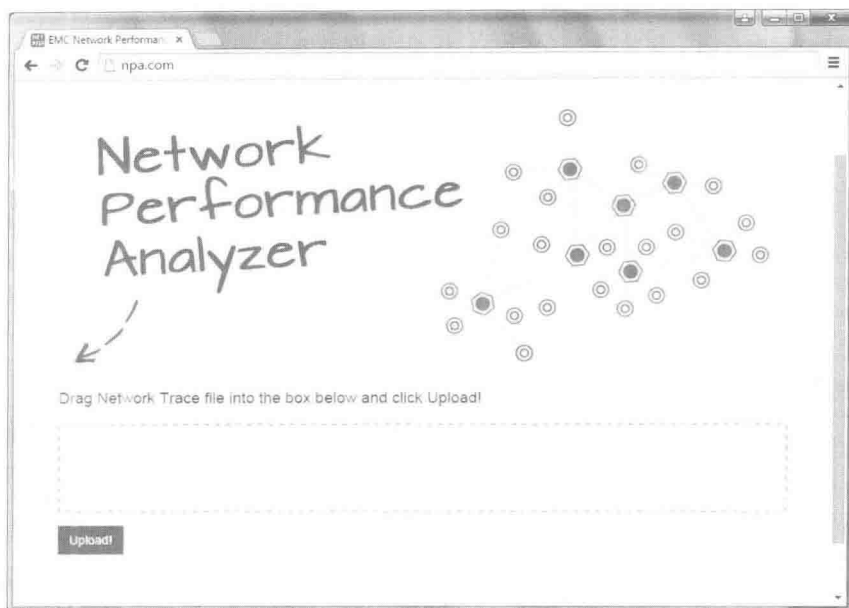


图 1

几秒钟后，分析报告就出来了。从上往下分别是“概况分析”、“应用层分析”、“传输层分析”等，下面我会逐项介绍。

图 2 显示的是“概况分析”，目的是给用户呈现一个直观的性能状况。比如“Data bytes rate: 22 kBps”和“Capture duration: 900 seconds”，表明在抓包的 900 秒里，平均性能才 22 KB/s，实在是很差。流量图的柱体高度起伏不大，说明这段时间内传输均匀，没有爆发性的流量或者暂停。

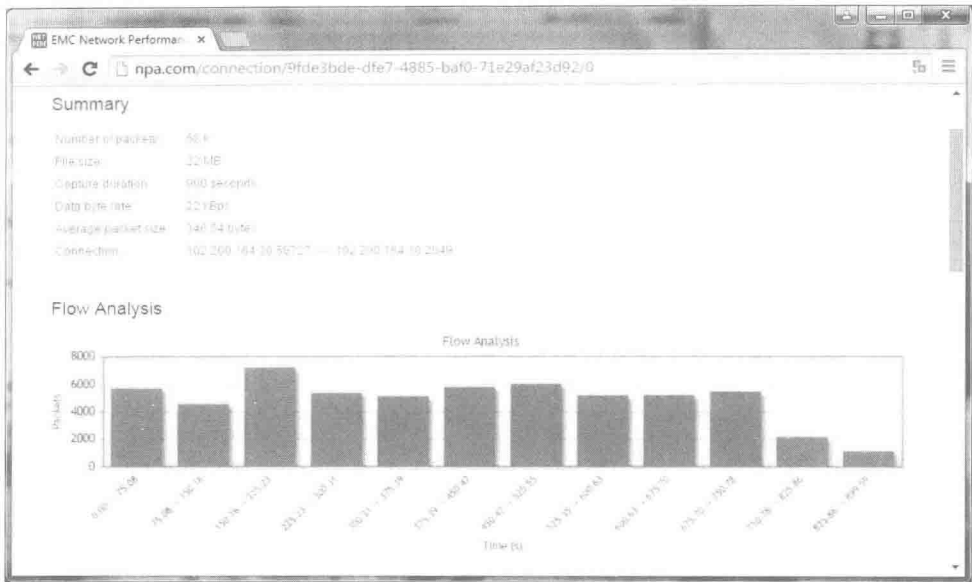


图 2

接下来是“应用层分析”，具体可见图 3。该工具自动判断出这个包的应用层协议是 NFSv3，因此把 NFS 响应时间（Service Response Time，SRT）和 IO Size 统计了出来。从图中的第一个方框可见 READ 的平均响应时间是 0.226 毫秒，算非常好了。可是从第二个方框却看到每次读的数据量只有 975 字节，还不到 1 KB，实在是太小了。这就像用货车从北京往上海运 1000 个包裹，假如每次能运 100 个，那 10 个来回时间就搞定了。而假如每次只能运 1 个，就得跑 1000 个来回，那浪费在路上的时间就非常可观了。因此，这个案例的解决方式就是调整软件的 IO Size，增大到每次读 64K 字节，性能立即得到大幅度提升。你可能会好奇，为什么同样的 IO Size，处理本地硬盘上的数据就没有性能问题呢？这就是网络的弱点了，TCP/IP 几层处理下来，总会增加一些延迟的。当来回次数特别多的时候，延迟的效应就被放大了。

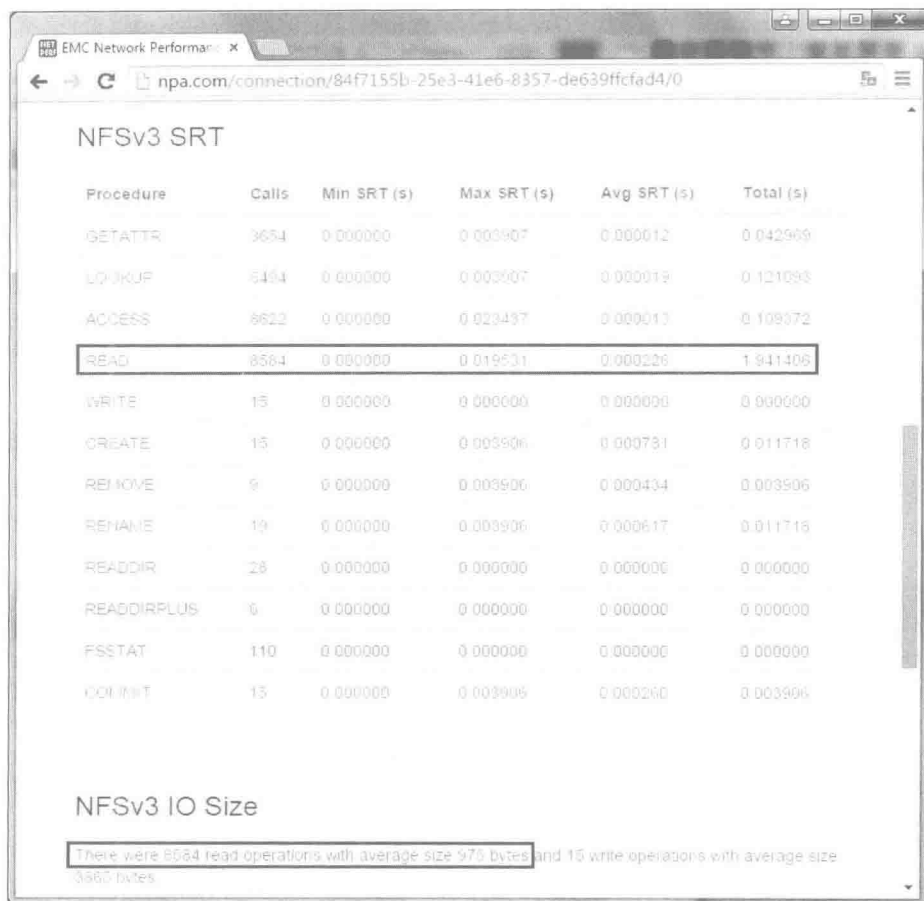


图 3

既然在应用层就已经找到症结，我们也没必要再去看传输层了。不过传输层可是性能问题的高发区，也是这个工具的特长之处，所以我忍不住再给大家看两个案例。

图 4 是 VMware 性能差的案例。抓包分析后，发现总共 250 秒的抓包时间里，有 190.8 秒被浪费在延迟确认上了，用上这工具之后简直就是秒杀。由于本书是黑白印刷的，所以看不出该工具已经把出问题的提示文本设置成红色背景，实际上是非常醒目的。

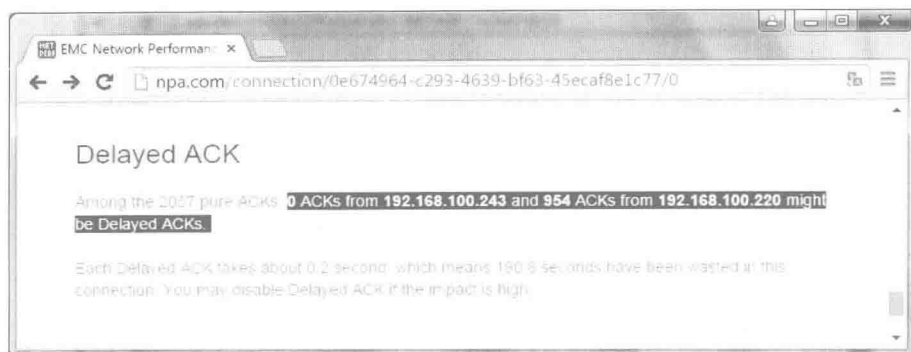


图 4

图 5 则是我上一本书的《深藏功与名》文章中提到过的某银行案例，根本原因是网络拥塞导致的丢包，而且 SACK 也没有启用，两个根源都被这工具分析出来了。当时要是用上这工具，也是很快就能解决的。

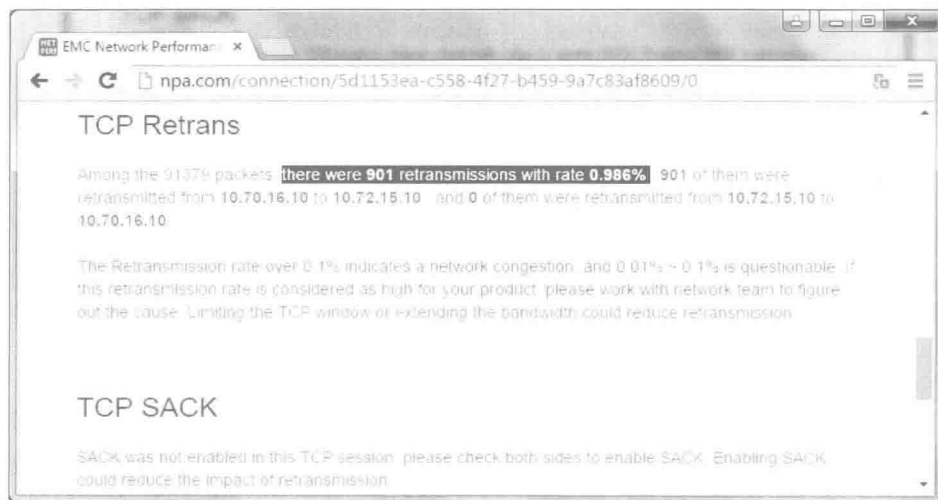


图 5

我手头的案例还有很多很多，篇幅所限就不一一列举了。可以说，我司的大多数网络性能问题都可以用这个工具找到症结。有很多团队已经从中受益，因为他们不用再请林沛满吃饭看包了，自助就能完成。我当然也很高兴，因为得以摆脱耗时的重复性劳动，有了更多的时间可以带娃。看到这里，不知道你是否也想打造一个适合自己职业的分析工具呢？有兴趣的话可以参考我的开发过程，大致可以分为三步。

第一步：收集旧问题。

我们不可能开发一套具有人工智能的程序来分析网络包。换句话说，自己打造的工具本质上不会比 Wireshark 更聪明。不过我们可以把自己的工作经验“传授”给这个程序，使它看上去比 Wireshark 智能很多。要如何做到呢？世界上绝大多数故障都不是第一次发生的，有经验的工程师可以把处理过的旧问题收集起来，归纳出每个问题在网络包中各有什么特征。以后抓到新的包，就可以用这些已知特征逐个去套，一旦发现匹配得上的就提示用户。比如我已知有 20 个原因会影响网络性能，每个原因在网络包中都会有一些特征，就可以在新抓的网络包里用这 20 个特征去逐个匹配。一旦发现有符合的就提醒用户，就像图 4 和图 5 那样。

Wireshark 需要用户点击多个按钮才会去分析，但我们的工具会主动分析并生成报告，这对用户来说就是智能化的体验。不只是网络性能问题，任何网络相关的技术领域都可以采用这个方法，比如从事 Windows Domain 相关工作的技术人员，可能保存着上百个常用的微软 KB，其中包括 DNS 解析出错、Authenticator 过大、UDP 包被切分丢弃，等等。这些问题都可以在网络包中以某个特征体现出来，因此也可以写成程序去匹配。网管员做监控也是如此，很多场景都是固定的。

把这些旧问题收集好了，就已经向成功迈出一大步。不过实际做起来可没那么轻松，你也许需要召集团队中最有经验的工程师，收集他们的需求和抓过的网络包，然后再筛选和测试。在这一步收集到的旧问题有多全面，就决定了你做出来的工具有多强大。

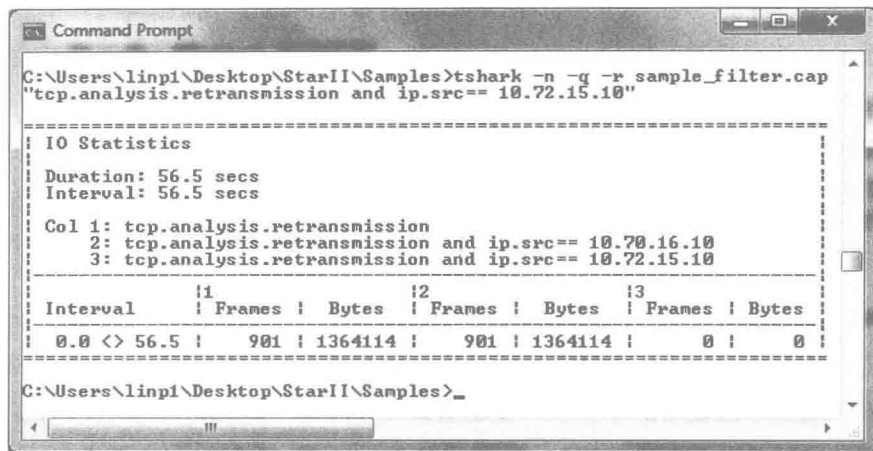
第二步：用 tshark 来做匹配。

tshark 是 Wireshark 的命令行形式，适合被其它程序调用来分析网络包；再加上其分析结果是文本输出的，所以作二次加工也很方便。基于这两点，选用 tshark 来匹配已知的特征是最合适的，如果你已经在上一步整理出了 20 个特征，那么再编辑 20 条 tshark 命令就基本可以搞定了。tshark 命令的使用方法在上一本书中已经介绍过，这里就不重复了。简单举个例子，已知性能问题的特征之一是 TCP 重传，那执行下面的命令就可以匹配了：


```
tshark -n -q -r <file_name> -z io,stat,0,tcp.analysis.retransmission,"tcp.analysis.  
retransmission and ip.src==<IP_A>","tcp.analysis.retransmission and ip.src==<IP_B>"
```

两个项目

输出示例如下：



```
C:\Users\linp1\Desktop\StarII\Samples>tshark -n -q -r sample_filter.cap  
"tcp.analysis.retransmission and ip.src== 10.72.15.10"  
  
-----  
| IO Statistics  
|-----  
| Duration: 56.5 secs  
| Interval: 56.5 secs  
|-----  
| Col 1: tcp.analysis.retransmission  
| 2: tcp.analysis.retransmission and ip.src== 10.70.16.10  
| 3: tcp.analysis.retransmission and ip.src== 10.72.15.10  
|-----  
| Interval | 1 | Frames | Bytes | 2 | Frames | Bytes | 3 | Frames | Bytes |  
|-----|-----|-----|-----|-----|-----|-----|  
| 0.0 <> 56.5 | 901 | 1364114 | 901 | 1364114 | 0 | 0 |  
|-----|-----|-----|-----|-----|-----|  
C:\Users\linp1\Desktop\StarII\Samples>_
```

图 6

在图 6 的输出中，列 1 的 Frames 表示所有重传包数，列 2 表示从 IP_A 到 IP_B 的重传包数，列 3 表示从 IP_B 到 IP_A 的重传包数。有了这些值就很容易统计重传率和重传方向。

当你不知道某个特征所对应的 tshark 命令是什么的时候，可以尝试从 Wireshark 中把它找出来，然后右键点击该特征，选择“Prepare a filter”→“Selected”，就可以在过滤栏生成表达式了，如图 7 所示。有了这个表达式就很容易应用到 tshark 命令中。

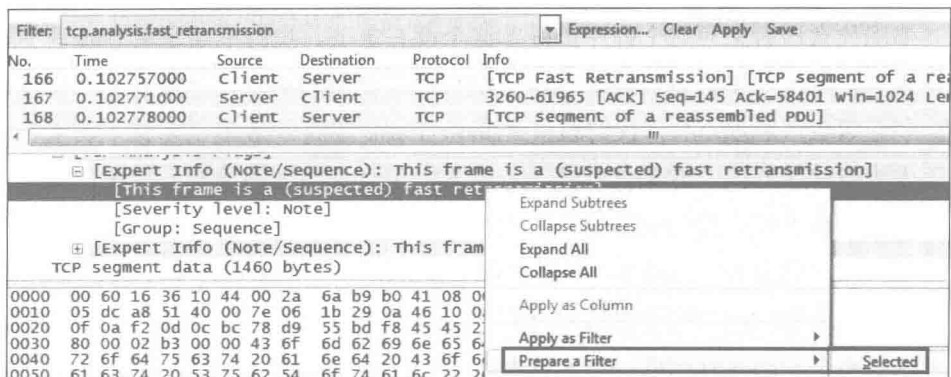


图 7

还有些命令是不能用这个方法找到的，只能自己查 tshark 的官方文档了，链接为 <http://www.wireshark.org/docs/man-pages/tshark.html>。tshark 命令真的非常强大，如果用得好，可以实现很多专业软件特有的功能。

这一步的 tshark 命令写得有多精确，就决定了你开发出来的工具有多可靠。

第三步：程序化。

到这一步，你已经整理了很多常见的问题，并知道如何用 tshark 命令来匹配它们，是时候写个程序来完成整项工作了。比如说，上一步从 tshark 输出中得到了重传的包数，那就可以用程序来计算重传率，并决定是否应该通知用户。这个程序可得好好设计，因为它关系到运行效率（当你抓到的网络包非常大时，就会发现运行效率是极其重要的，否则等半个小时都没有结果）。举个例子，应用层上有 HTTP、FTP、iSCSI、NFS、CIFS 等协议，每一个协议都有不同的问题，每个问题又对应着不同的 tshark 命令。我们总不能拿到一个网络包，就把所有 tshark 命令都运行一次吧？那样效率太低了。正确的方法是让程序先判断包里的应用层协议是什么，然后再调用其相关的命令。那怎样知道抓到的包是什么协议的呢？我们可以根据端口号来判断，比如端口号为 80 时，就调用 HTTP 相关的命令；端口号为 445 时，就调用 CIFS 相关命令……还有些实在无法用程序自动判断的，可以由用户来辅助完成。比如在页面上提供多个按钮，对应着不同的协议，让用户自己选择。总而言之，产品经理必须非常熟悉业务流程，才能把这个程序写得高效、科学、友好。

两个项目

打造自己的
分析工具

187

那用什么语言来写这个程序最好？这个没有定法。我们早期是用 Perl 写的命令行脚本，开发简单，运行速度也快。但它也有致命的缺点，就是界面不美观，推广和升级也很麻烦。后来我们改用 Python+Flask 做成了 Web 的形式，还请专业美工人士设计了界面，效果就好多了。作为一个有强迫症的伪产品经理，我还想强调细节的重要性，比如网络包分析过程中，一定要在页面上显示一个转动的菊花来延长用户的耐心，见图 8。不要小看这种小细节，如果分析时间超过三分钟，又没有菊花在转动，用户很可能以为程序已经死了，然后就点刷新，又得从头再来一次。对细节的重视程度，很大程度上决定了这个工具的用户体验。

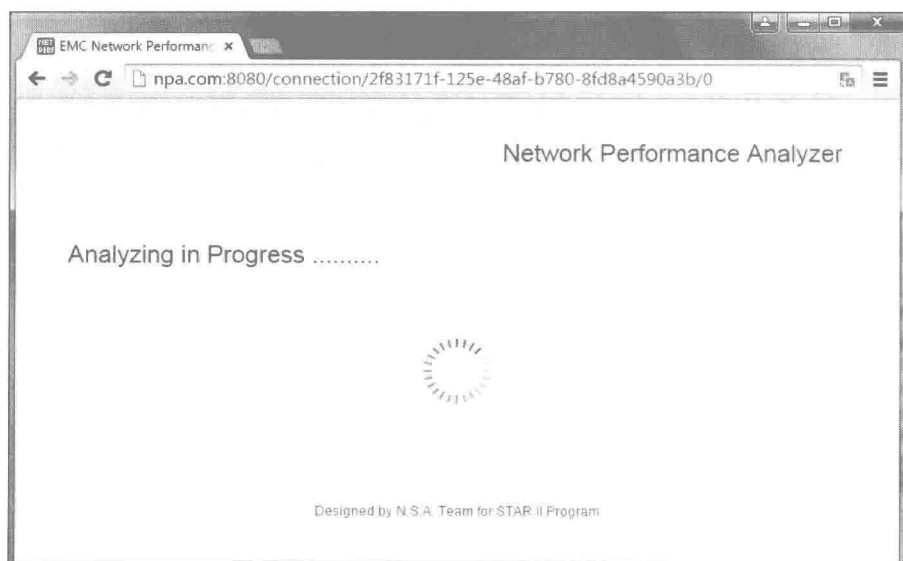


图 8

这个工具就介绍这么多，希望对你有参考价值。如果你在开发过程中遇到什么问题，也欢迎进一步交流。困难肯定会有，但只要肯动手去做，你就成功了一半。

现在是 2015 年初秋，一个收获的季节，这本书也写到了尾声。此刻我正在小区附近的咖啡馆里，斟酌最后一篇应该写些什么。此书的很多章节就是在这家店里完成的，但接下来应该有很长时间不会再来了，因为最近兴起的创业大军实在太吵。邻桌正在激情澎湃地讨论“盈利模式”、“估值”、“收购”、“A 轮 B 轮 C 轮”……上周还听到一位从阿里离职的工程师在怂恿同伴出来一起开发手机 APP，展望前景的语气让我想起了安利的培训。

其实我六七年前也产生过一个稍纵即逝的创业念头。与现在流行的 P2P、O2O 等概念不同，那时的 IT 创业主要集中在传统的技术领域，比如我老板做的数据迁移设备就卖了一个很好的价钱。有趣的是那产品两年后就被淘汰了，命运跟现在的初创公司很像。而我当时想做的是一个网络加速器，它究竟是个什么东西呢？细想起来，它跟我之前讲过的很多技术都有关联，不如最后一篇就写写它吧，就当作知识总结。

那几年我接触了世界上很多知名公司的数据中心，发现他们都有同样的痛点，即跨站点（site）的网络存在性能瓶颈。比如图 1 这样的环境中，纽约的用户访问伦敦的文件服务器，或者两边的数据库做同步，都会慢得出奇。

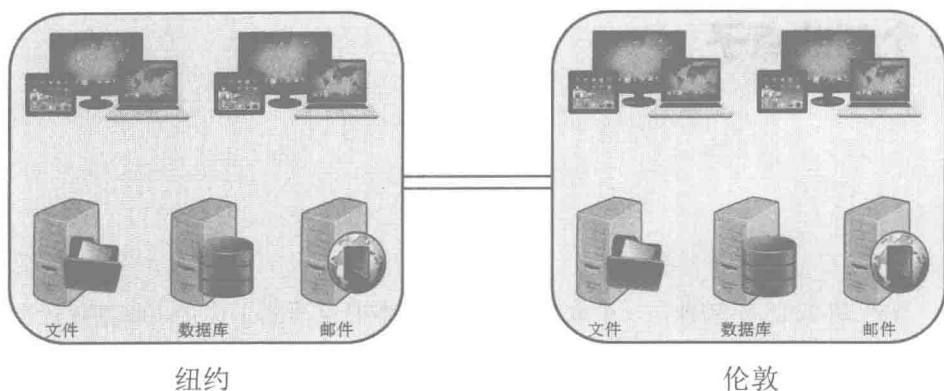


图 1

这类问题排查下去，一般会归根于带宽不足，解决方式就是花钱购买更大的带宽。然而很少人知道还有一个不花钱的办法，就是通过传输层（基本都是 TCP）的调优来提高带宽利用率，从而提升性能体验。那时候我已经学会了分析网络包，知道传统的 TCP 协议栈不能很好地应对跨站点的场景，所以带宽利用率偏低。有些严重的甚至在 50% 以下，因此存在很大的提升空间。比如客户从我司购买的文件服务器在跨站点时访问不快，但经过专业调优之后，性能可以提升两倍以上。这就是商机所在：既然可以通过调优的方式来达到和购买带宽一样的效果，我们就有了盈利的空间。接下来的问题就是怎样做成一个产品了。

人工调优能做到的事情，理论上程序也可以做到。因此我最早想做的产品是改进型的 TCP 协议栈，装在服务器上，使它在跨站点场景中能够更智能地工作，达到人工调优后的效果。不过很快就发现这个路子走不通，原因有三。

- 有很多操作系统不允许修改原有的 TCP 协议栈。比如我司的服务器就是完全封闭的，第三方厂商根本不知道怎么修改。有些服务器虽然就是普通的 Linux 或者 Windows，技术上能够修改，但是厂商声明一旦动了协议栈就不再提供技术支持。
- 即使服务器都用上了改进过的协议栈，也会受到客户端配置的约束，难以充分发挥。比如在客户端关闭了 TCP Timestamps (RFC 1323)，那在服务器上计算 RTT (往返时间) 时就会受到影响；或者客户端关闭了 SACK，那在服务器上启用 SACK 也没有意义。

- 没有用户愿意为了改善跨数据中心的访问，而大动干戈地对服务器的 TCP 层作出改动。万一改动之后影响了本地访问性能怎么办？

注意：这三点只说明该产品不适合本文所针对的场景，而不是说它没有价值。事实上它在有些场景下可以工作得很好，现在也已经有商业化的产品了，比如硅谷有家叫 AppEx Networks 的公司推出的单边加速器 ZetaTCP 就不错。我后来才发现其 CEO 是位华人，在北京也有分公司。市面上还有一些很滑稽的加速器，比如通过每个包发两次来避免丢包的，在我看来就是浪费流量的七伤拳，不建议采用。

既然这个路子完全走不通，我们只能设计一个不同的产品了，它至少要满足以下需求才行。

- 它不需要对服务器或客户端的 TCP 协议栈作任何改动，所以实施的障碍会小很多。
- 它完全独立工作，所以不受客户端和服务端上的 TCP 设置所影响。比如客户端上没有启用 SACK 时，它也能处理好连续丢包的问题。
- 它只用于改善跨数据中心的网络性能，对本地访问毫无影响。

需求一旦明确，解决方案便呼之欲出了。如图 2 所示，只要在两个站点的出口各自架设一台加速器，代理两个站点之间的所有 TCP 连接，就可以满足以上所有需求。由于每台加速器与同站点设备之间的网络状况良好，所以瓶颈只会落在两台加速器之间的网络上，我们只需花心思提升这段网络的性能即可。也许有些读者看到这里会觉得好笑，现在这种加速器在国内外至少有十个牌子，连开源项目都有了，你还创什么业啊？现在的确是成熟的市场了，但是当年可完全不是这样，尤其没有听说过国内的公司。我也只是因为分析了足够多的网络包，便自然而然地萌生了引入加速器的念头。技术之外的话题就不多说了。

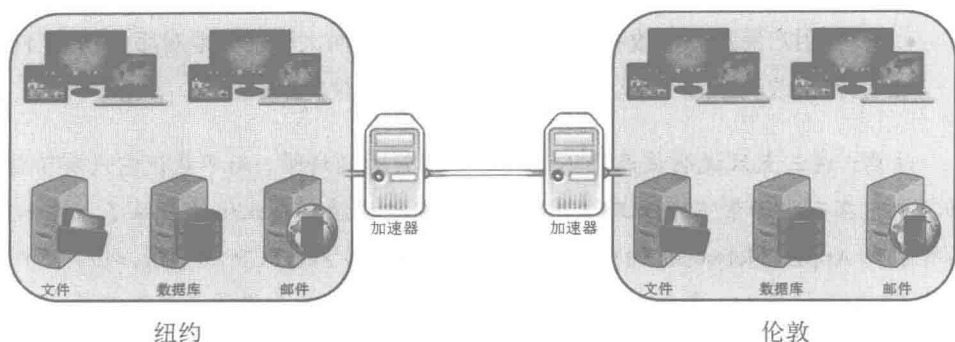


图 2

我们先来分析一下这段网络存在什么问题，才能对症下药，总结下来主要有两个大问题。

问题一：延迟高。

位于同一站点的两台设备之间往返时间一般也就几毫秒，而跨城网络的往返时间可能达到几十毫秒，跨国网络甚至可达上百毫秒。高延迟为什么会影响性能呢？因为它会造成长时间的空等：发完一个窗口的数据量后，发送方就不得不停下来等待接收方的确认。延迟越高，发送方需要等待的时间就越长。一图胜千言，图 3 演示了发送窗口都是 2 个 MSS，延迟时间分别为 10 毫秒和 20 毫秒时的传输过程，可见后者效率只有前者的 1/2。这好比用同一辆货车运货，从上海运到江苏肯定比从上海运到北京快得多。

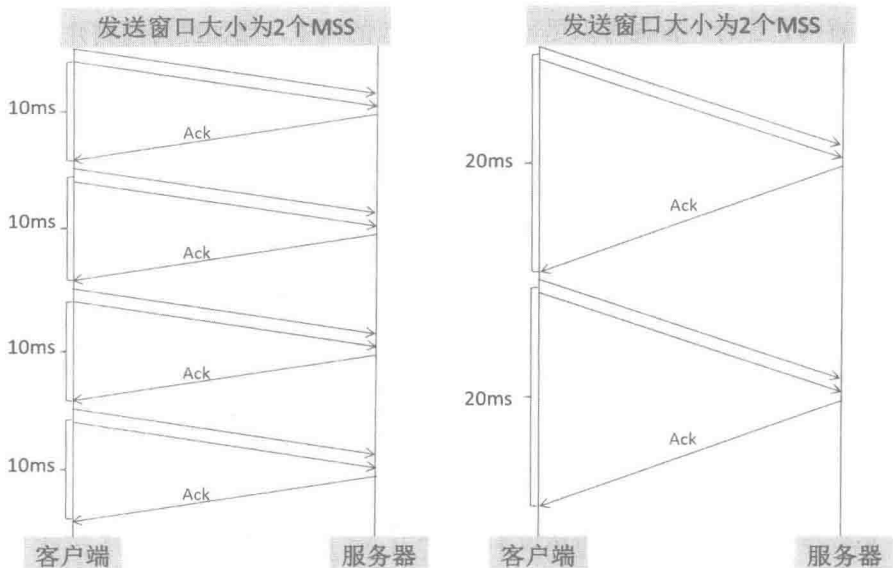


图 3

有时候我们会在 Wireshark 中看到 [TCP window Full] 的提示，就表明发送方进入了等待状态。这种症状在跨站点通信时是很常见的，具体可见本书《Wireshark 的提示》一文中的图 9。那么这个问题要怎么解决呢？在延迟时间无法减少的情况下，发送窗口越大，性能就越好，所以要尽可能增大窗口。

问题二：丢包率高。

丢包一般分两种情况：一种是网络质量差导致的零星丢包；另一种是拥塞导致的大量丢包。跨站点通信时这两种丢包概率都会增大，尤其是后者。这是因为链路上的情况比较复杂，而且不同的 TCP 连接会“恶意”地争夺本来就有限的带宽。比如图 2 中的文件服务器、数据库和邮件服务器等建立的 TCP 连接会各自为政，互相争夺带宽，直至发生丢包才停下来。这种情况很像上海马路上的车辆，为了加速而变道的车多了，就容易诱发交通事故。

丢包对性能的影响极大，可以说是网络传输的第一大忌，具体原因我都在上一本书中阐述了，这里再简单解释一下：传统 TCP 的流控机制是一旦丢包就认为发生了拥塞，所以发送方会急剧地减小发送窗口，甚至进入短暂的等待状态（即超时重传）。1%的丢包率不只是降低 1%的性能，而可能是 50%以上。这个问题有

办法缓解吗？也有。首先可以尽可能降低丢包的概率，比如提前预测并采取措施避免拥塞的发生；其次是更精细地处理丢包后的流控，避免过度限流。

一番分析下来，发现这两个问题还是很棘手的，但是不用担心，我们还手握王牌呢——在加速器上可以大做文章，大幅度缓解这两个问题所带来的影响。作为一个创业奸商，其实我们应该希望影响尽可能严重，带宽利用率最好在 50% 以下。因为这意味着留给加速器的提升空间就大了，客户购买之后能看到明显的效果，才会觉得物有所值。接下来要介绍的就是缓解这两个问题的措施，也是我们这个加速器的技术含量所在。

措施 1：启用 TCP window scale。

这样可以使最大接收窗口从 65,535 字节（老的 Windows 操作系统甚至只有 17520 字节）增加到 1,073,725,440 字节。发送窗口是受接收窗口和拥塞窗口共同限制的，启用 TCP window scale 之后，接收窗口就几乎限制不到了，当然内存也要跟得上才行。关于 TCP window scale 的更多信息，可参考本书的另一片文章《技术与工龄》。

措施 2：监测延迟来避免拥塞。

网络包是以队列的方式通过网络设备的。当拥塞即将发生时，队列变长，延迟就会显著提高。我做了一个从台湾机房往上海机房传数据的实验，一般情况下的往返时间为 74 毫秒（见图 4 方框中的 RTT），而拥塞丢包发生前会逐渐增加到 1.69 秒以上。根据这一特点，我们可以让加速器在延迟明显增加时，自动放慢发送速度，从而避免拥塞的发生。

一般情况:

No.	Time	Source	Destination	Protocol	Info
821	3.828125	Shanghai	Taiwan	TCP	8888-60479 [ACK] Seq=553 Ack=435308 Win=4096 Len=0
824	3.832031	Shanghai	Taiwan	TCP	8888-60479 [ACK] Seq=553 Ack=437491 Win=4096 Len=0
827	3.832031	Shanghai	Taiwan	TCP	8888-60479 [ACK] Seq=553 Ack=439674 Win=4096 Len=0
830	3.835937	Shanghai	Taiwan	TCP	8888-60479 [ACK] Seq=553 Ack=441654 Win=4096 Len=0
833	3.839843	Shanghai	Taiwan	TCP	8888-60479 [ACK] Seq=553 Ack=443836 Win=4096 Len=0
836	3.839843	Shanghai	Taiwan	TCP	8888-60479 [ACK] Seq=553 Ack=445616 Win=4096 Len=0

```

window size value: 4096
[calculated window size: 4096]
[window size scaling factor: -1 (unknown)]
[Checksum: 0x5bb8 (validation disabled)]
urgent pointer: 0
[options: (12 bytes), No-operation (NOP), No-operation (NOP), Timestamps]
[SEQ/ACK analysis]
[This is an ACK to the segment in frame: 724]
[The RTT to ACK the segment was: 0.074219000 seconds]

```

拥塞发生前:

No.	Time	Source	Destination	Protocol	Info
27323	122.937500	Shanghai	Taiwan	TCP	8888-50637 [ACK] Seq=6349 Ack=6085797 Win=4096 Len=0
27326	122.937500	Shanghai	Taiwan	TCP	8888-50637 [ACK] Seq=6349 Ack=6087929 Win=4096 Len=0
27329	122.941406	Shanghai	Taiwan	TCP	8888-50637 [ACK] Seq=6349 Ack=6090705 Win=4096 Len=0
27332	122.941406	Shanghai	Taiwan	TCP	8888-50637 [ACK] Seq=6349 Ack=6090725 Win=4096 Len=0
27334	122.941406	Shanghai	Taiwan	TCP	8888-50637 [ACK] Seq=6349 Ack=6093309 Win=4096 Len=0
27337	122.941406	Shanghai	Taiwan	TCP	8888-50637 [ACK] Seq=6349 Ack=6096075 Win=4096 Len=0

```

window size value: 4096
[calculated window size: 4096]
[window size scaling factor: -1 (unknown)]
[Checksum: 0x1f37 (validation disabled)]
urgent pointer: 0
[options: (12 bytes), No-operation (NOP), No-operation (NOP), Timestamps]
[SEQ/ACK analysis]
[This is an ACK to the segment in frame: 27015]
[The RTT to ACK the segment was: 1.695313000 seconds]

```

图 4

这其实就是 TCP Vegas 的理念。它用在服务器上时不见得很好，甚至有负作用。想象一台启用了传统 TCP 协议栈的服务器和一台启用了 Vegas 的服务器抢带宽，当拥塞即将出现时，用 Vegas 的那台监测到了延迟并主动放慢速度，从而缓解了拥塞，但传统的那台却得寸进尺，一直激进地抢带宽。最终结果可能是传统的那台反而赢了——劣币淘汰良币。而在加速器上引入 Vegas 理念就不一样了，由于每个 TCP 连接都是一样的算法，所以预测到拥塞时大家可以集体放缓，从而保证了公平性。这就像马路上每位司机都礼貌谦让，就不会发生事故，整条马路的通行效率也提高了。

除了能预测拥塞，监测延迟时间还有助于区分零星丢包和拥塞丢包，因为发生零星丢包时的延迟一般不变。区分它们有什么意义呢？传统 TCP 协议栈遇到丢包都一律当作拥塞处理，立即放慢速度甚至暂停。这样一刀切并不科学，零星丢包时重传一下就行了，没必要放慢速度。

措施 3：利用发送窗口实现优先级。

两个站点之间存在很多连接，且优先级各有不同，比如数据归档的优先级就可能低于其它应用，可以传慢一点。我们的加速器代理了两个站点之间的所有连接，因此很容易通过调节各个连接的发送窗口来实现优先级控制。优先级低的连接变慢了，就可以把带宽让给优先级高的连接，用户体验就会更好。

措施 4：启用 SACK。

SACK 即 Selective Acknowledgment，它是处理拥塞丢包时的法宝，尤其是在高延迟的跨站点环境中，详情可参考本书的另一片文章《来点有深度的》。SACK 必须在发送方和接收方都启用，这就是我们在两边各架设一台加速器的优势。单边 TCP 加速器的效果很可能因为另一端没有启用 SACK 而大打折扣。

措施 5：改进慢启动算法。

传统的 TCP 协议栈采用了非常保守的慢启动算法，即把拥塞窗口的初始值定义得非常小，不能大于 4 个 MSS。而且一旦发生超时重传，又要从头进入慢启动阶段，如图 5 所示。

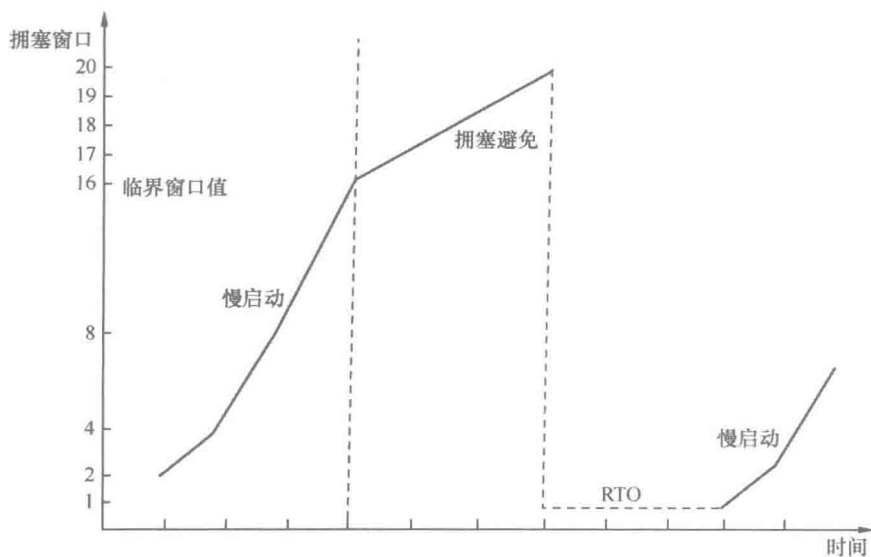


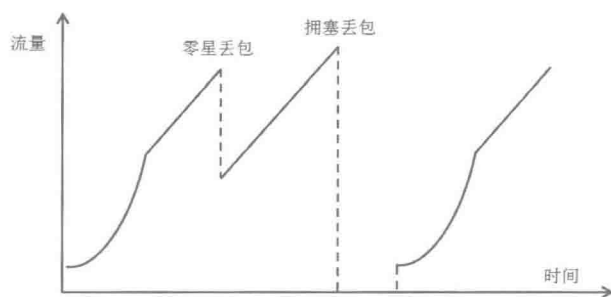
图 5

这就意味着传输过程中至少有一段时间的窗口极小，效率非常低。随着硬件的更新换代，现在的网络带宽已经今非昔比了，完全没必要如此保守。作为一个专业的 TCP 加速器，我们有必要在这一点作出改进。比如赋予发送方一定的“智能”，使用大一点但仍然安全的初始值。根据我的经验，在这一块是很有提升空间的，因为传统的 TCP 协议栈的初始值在现代网络中显得实在太小了。

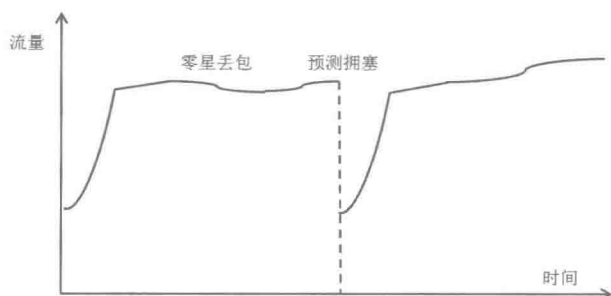
措施 6：启用 TCP Timestamps。

在本书的《一篇关于 VMware 的文章》一文中，已经介绍过延迟确认是如何影响性能的。不难理解，它也会严重影响 RTT 的统计。我们需要精确地监测延迟时间来预防拥塞，就必须在两边都启用 TCP Timestamps（见 RFC 1323 的 RTTM 一节）来排除延迟确认等因素的干扰。这也是双边加速的好处之一，在服务器上单边加速时很难排除客户端的干扰。

总结下来，这些措施合力实现了这样的效果：在起步的时候，它传输得更快；在抢夺带宽的时候，它更懂得谦让；在出现拥塞时，它恢复得更迅速。此外它还能在一定程度上避免拥塞，识别零星丢包等等，因此流量可以稳定在高位。加速前后的某个 TCP 连接，流量变化大致可以用图 6 来表示。



加速前流量图



加速后流量图

图 6

本文提到的这些措施我大多验证过，由于实验室中不存在高延迟，我还搭了一台专门制造延迟和丢包的路由设备来仿真。其中部分措施更是在用户环境中验证过多次。因此可以信心满满地说，这个加速器在技术上是完全可行的。那现在市面上的加速器采用的也是这些技术吗？从部分公司所公布的文档上，我的确看到了一些交集，当然它们还用到了压缩和消重等 TCP 之外的技术。Wireshark 在加速器领域也是大有可为，这就是为什么它的主要捐助者是加速器的领头羊 Riverbed。