22 | 为什么压力测试TPS总是上不去？

2022-03-11 杨胜辉

《网络排查案例课》 课程介绍 >

讲述：杨胜辉
时长 21:17 大小 19.51M

你好，我是胜辉。

在上一讲里，我们排查了一个跟操作系统紧密相关的性能问题。我们结合 top 和 strace 这两个工具，抓住了关键点，从而解决了问题。性能问题，确实也是我们日常技术工作中的一个重要话题。在出现性能问题以后，我们要有能力搞定它；而在出现性能问题之前，最好能提前预见到它。而要“预见”性能瓶颈，最好的方法就是做**压力测试**。

但是，我们在做压力测试的过程中也时常出现预料不到的情况。比如在离预期的瓶颈值还很远的时候，系统就出现了各种意外，影响到压力测试的继续进行。

那么在这节课里，我们会回顾几个典型的压力测试场景中的网络问题，一起来学习其中的关键要点。同时，我们还会学习一系列跟网络性能相关的压测工具和检测工具，这样以后

你遇到类似的问题时，就有所准备了。

压测要做什么？

压力测试的诉求实际上多种多样，不过大体上可以分为这几种。

应用的承受能力：这主要在第七层应用层，比如发起了压测，把服务端的 CPU 打到 95% 甚至 100%，观察这时候的请求的 TPS、请求耗时、并发量等等。而这些对于不同的业务场景，又会有不同的侧重点。比如：

对于时间敏感型业务来说，请求耗时（Latency）这个指标就是关键了。

对于经常做秒杀的电商来说，并发处理量 TPS（Transaction Per Second）就是一个核心关注点了。

LB 的连接处理能力：这主要在第四层 TCP，看 LB 能最大支持的 TCP 并发连接数。这时候，发起压测的客户端一般会指定比较大的并发数，这样就可以发起尽量多的 TCP 连接。

网络的承受能力：这可能主要在第三层 IP 层了，比如测试上行和下行带宽能否跑满、是否有丢包和额外的延迟，等等。特别是对于一些流量比较大的场景，很可能服务端计算能力都还在，但带宽已经不够用了，所以我们要提前发现这些隐患。

案例 1：压测 TPS 上不去

我们有个客户是传统企业转型做电商，有一次准备搞大促。为了确保大促顺利，他们要提前对网站进行压测。

客户用的压测工具比较简单，是 Apache ab。ab 是 Apache Benchmark 的缩写，它的用途就是对 HTTP 服务端发起测试，以获得性能指标（Benchmark）。ab 本身不是独立安装的，而是在 apache2-utils 工具包里，所以你可以这样来安装它：

复制代码

```
1 apt install apache2-utils
```

ab 是一个轻量级的工具，因为相对其他重量级的工具比如 LoadRunner 或者 JMeter 来说，ab 只要一行命令就可以发起压测了，是不是很省事？比如你可以这样：

复制代码

```
ab -c 100 -n 10000 目标URL
```

通过上面的命令，你就用 -c 100 这个参数，让 ab 发起了 100 个并发的请求，而 -n 10000 指定了总共发送的请求量。

> 补充：这里有一个小的注意点。如果目标 URL 只是站点名本身，还是需要在结尾处加上“>/”，要不然 ab 会报这个错误：ab: invalid URL

比如我用下面这条 ab 命令，对一个著名网站发起“压测”，当然我的参数选择的很小，只有 10 个并发，一共 100 次请求，尽量避免打扰到这个网站。我们可以看一下输出：

复制代码

```
$ ab -c 10 -n 100 https://www.baidu.com/abc
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking www.baidu.com (be patient).....done


Server Software:        Apache
Server Hostname:        www.baidu.com
Server Port:            443
SSL/TLS Protocol:       TLSv1.2,ECDHE-RSA-AES128-GCM-SHA256,2048,128
Server Temp Key:        ECDH P-256 256 bits
TLS Server Name:        www.baidu.com

Document Path:          /abc
Document Length:        201 bytes

Concurrency Level:      10
Time taken for tests:   9.091 seconds
Complete requests:      100
Failed requests:        0
Non-2xx responses:      100
Total transferred:      34600 bytes
HTML transferred:       20100 bytes
Requests per second:    11.00 [#/sec] (mean)
Time per request:       909.051 [ms] (mean)
```

```
Time per request:       90.905 [ms] (mean, across all concurrent requests)
Transfer rate:          3.72 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:      141  799 429.9    738    4645
Processing:    19   67 102.0     23     416
Waiting:       17   67 101.8     23     416
Total:        162  866 439.7    796    4666

Percentage of the requests served within a certain time (ms)
  50%    796
  66%    877
  75%    944
  80%   1035
  90%   1093
  95%   1339
  98%   1530
  99%   4666
 100%   4666 (longest request)
```

结尾部分的多个 Percentage，确切地说就是 Percentile，也就是百分位。比如 50% 796 这一行的意思是：第 50 个请求（因为总数是 100 个）的耗时小于等于 796 毫秒，另外 50 个请求大于 796 毫秒。那么我们也可以知道，耗时最长的那个就是排最后一名的 4666，它的耗时是 4666 毫秒。

这次，客户用 ab 时指定的具体参数是这样的：

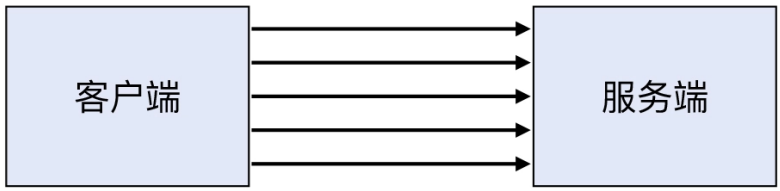
复制代码

```
ab -k -c 500 -n 100000 http://site.name.com/path
```

也就是并发数为 500，总次数为 10 万，而 -k 参数是启用长连接。然后就是查看以下两个主要指标：

ab 这个客户端的耗时分布，也就是前面刚介绍过的各个百分位的耗时数值。

服务端的性能指标，也就是在这个压力下面，服务端的 CPU、内存、网络丢包率等的统计数值。



极客时间

压测过程中，客户发现测试端的带宽用到 400Mbps 后，TPS 就上不去了，无论把并发量或者总量的数值进行怎样的调整，TPS 都会维持在一个稳定的数值。

```
1 Transfer rate:          52087.82 [Kbytes/sec] received
```

复制代码

那究竟是不是购买的服务端主机的性能不够的原因呢？要知道，客户端和服务端都是 1Gbps 的网卡，客户是想压满 100% 的带宽，现在只用到了 40% 的带宽，TPS 就上不去了。

为什么会这样呢？

我们就跟客户配合，一边做 ab 压测，一边做了 4 秒钟的 tcpdump 抓包，这对采样分析来说也够用了。然后打开抓包文件，查看专家信息（Expert Information），如下：

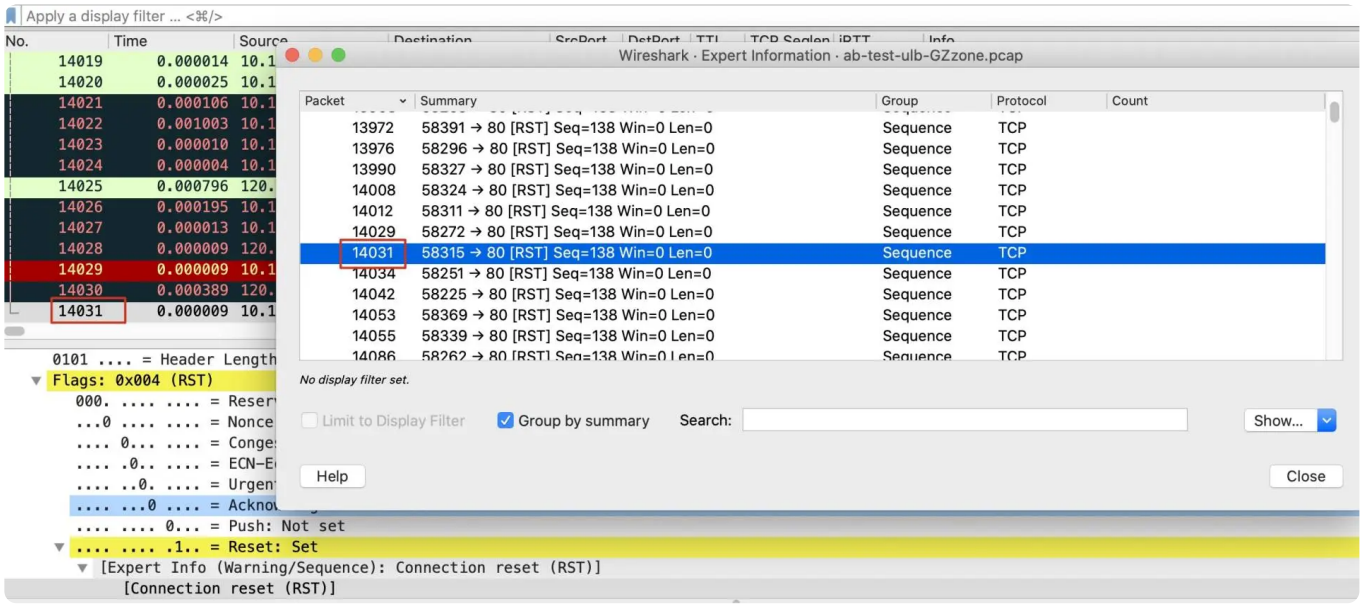
> 补充：抓包示例文件已经上传至 Gitee，建议结合抓包文件和文稿一起学习。

Severity	Summary	Group	Protocol	Count
Error	New fragment overlaps old data (retransmission?)	Malformed	TCP	170
Warning	Illegal characters found in header name	Protocol	HTTP	1
Warning	Connection reset (RST)	Sequence	TCP	1982
Warning	This frame is a (suspected) out-of-order segment	Sequence	TCP	19
Warning	Previous segment(s) not captured (common at capture start)	Sequence	TCP	222
Warning	ACKed segment that wasn't captured (common at capture start)	Sequence	TCP	210
Note	A new tcp session is started with the same ports as an earlier s...	Sequence	TCP	482
Note	Duplicate ACK (#1)	Sequence	TCP	1483
Note	This frame is a (suspected) spurious retransmission	Sequence	TCP	184
Note	This frame is a (suspected) retransmission	Sequence	TCP	2274
Chat	TCP window update	Sequence	TCP	9
Chat	Connection finish (FIN)	Sequence	TCP	2132
Chat	GET [illegible]	Sequence	HTTP	1603
Chat	Connection establish acknowledge (SYN+ACK): server port 80	Sequence	TCP	1975
Chat	Connection establish request (SYN): server port 80	Sequence	TCP	1435

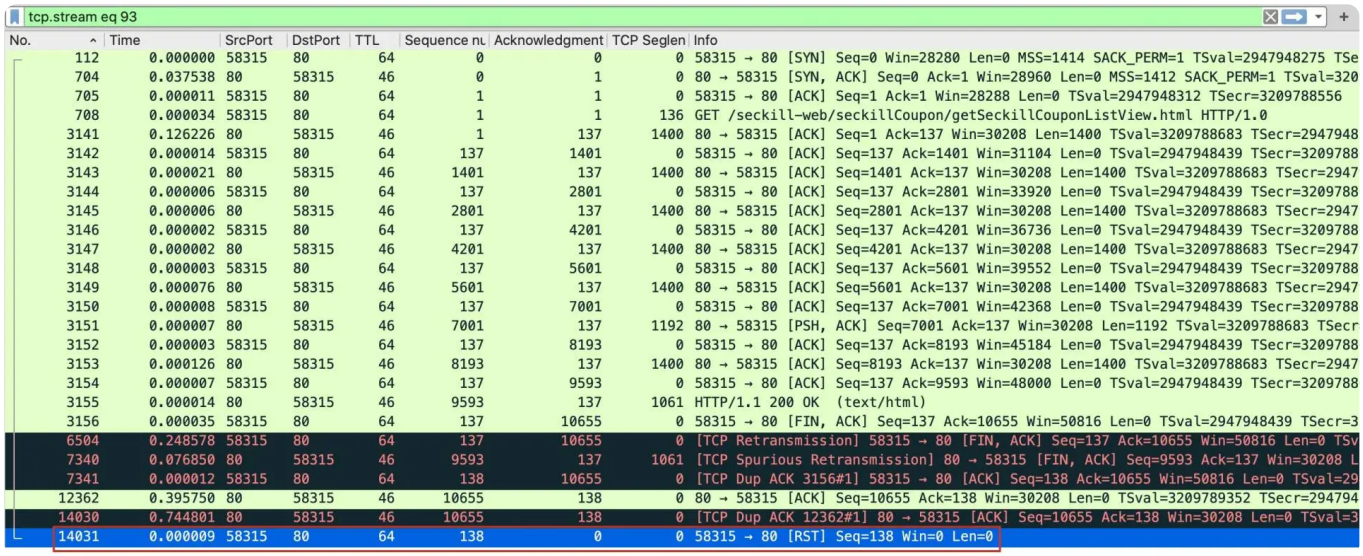
这次我就不做标记了，你自己先找找看，能否找到一些问题？

你有没有发现，GET 有 1603 次（倒数第三行），而 RST 有 1982 次（第三行），比 GET 这种 HTTP 请求的次数还更多。也就是说，平摊的话，每次 GET 请求对应了一次以上的 RST。这个现象会不会跟 TPS 上不去的问题有关系呢？

我们选一个 RST 来看一下情况。比如 14031 号报文：



然后 Follow -> TCP Stream，就来到了这条 TCP 流：

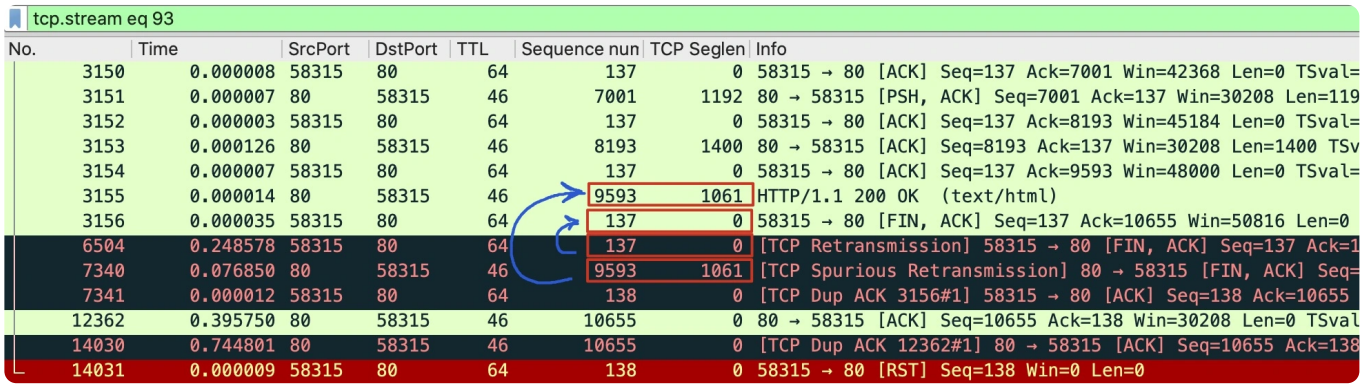


在这里，你有没有发现两个重传报文？一个是 6504 号报文，一个是 7340 号报文。我们再进一步看一下这两个的重传的原因是什么。一般来说，判断一个报文是否是重传，最方便的就是借助 Wireshark 本身提供的提示，Wireshark 说是 Retransmission，那就是了。

那么假设世界上还没有 Wireshark，你又**如何判断一个报文是否是重传呢？**其实可以根据两个关键信息：**序列号、载荷长度**。

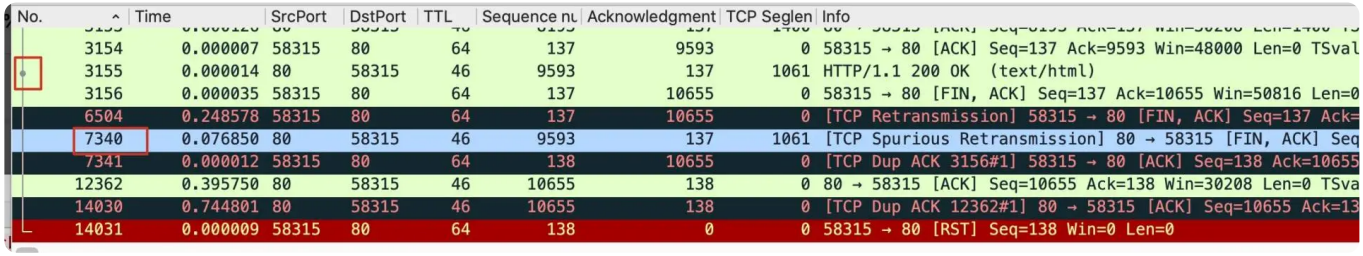
我在之前的课程里提到过，序列号本身反映的就是字节位置，载荷的长度就表示这段报文的实际字节长度，这两个信息就确定了信息的**起点和终点**，也就是决定了一个报文是否是之前报文的重传。

在上面的截图里，我们根据序列号（Sequence Number 列）和载荷长度（TCP Seglen 列），判断出下面两次重传：

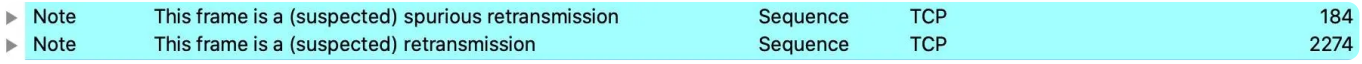


可见，7340 是 3155 报文的重传，因为它们的序列号都是 9593，载荷长度都是 1061。同样的道理，6504 和 3156 也是一对重传关系。

当然，我们也可以直接在 Wireshark 里选中 7340 号报文，它的基础报文也就是 3155 的**前面会出现一个小圆点**，就像下图这样：



总之，我们确认 TCP 传输中是有丢包和重传现象的，我们可以回头从专家信息那里得到证实，这里显示有 184 个 Suprious 重传和 2274 个（超时）重传。



> 补充：而出现 RST 的原因，是客户端在已经没有了这个 TCP 连接的情况下，收到了服务端的 ACK 报文。从现象来看，客户端应该是做了 TIME_WAIT 优化的设置，我们后面的实验里也会带到。

考虑到这是在压测场景中出现的丢包现象，我们根据经验，想到了**网卡包量**的问题。

一般说到网络性能，我们会讨论的就是带宽、时延、网速等等这些指标。实际上，另外一个性能指标常常在达到带宽极限之前就已经触顶了，它就是包量。

包量是对 PPS（Packet Per Second）的简称，一般用来衡量一台主机的网络处理能力。
那么，这次是不是触及了服务端主机的包量上限了呢？我们又如何检查包量指标呢？

我们需要用的工具是 sar。

性能工具 sar

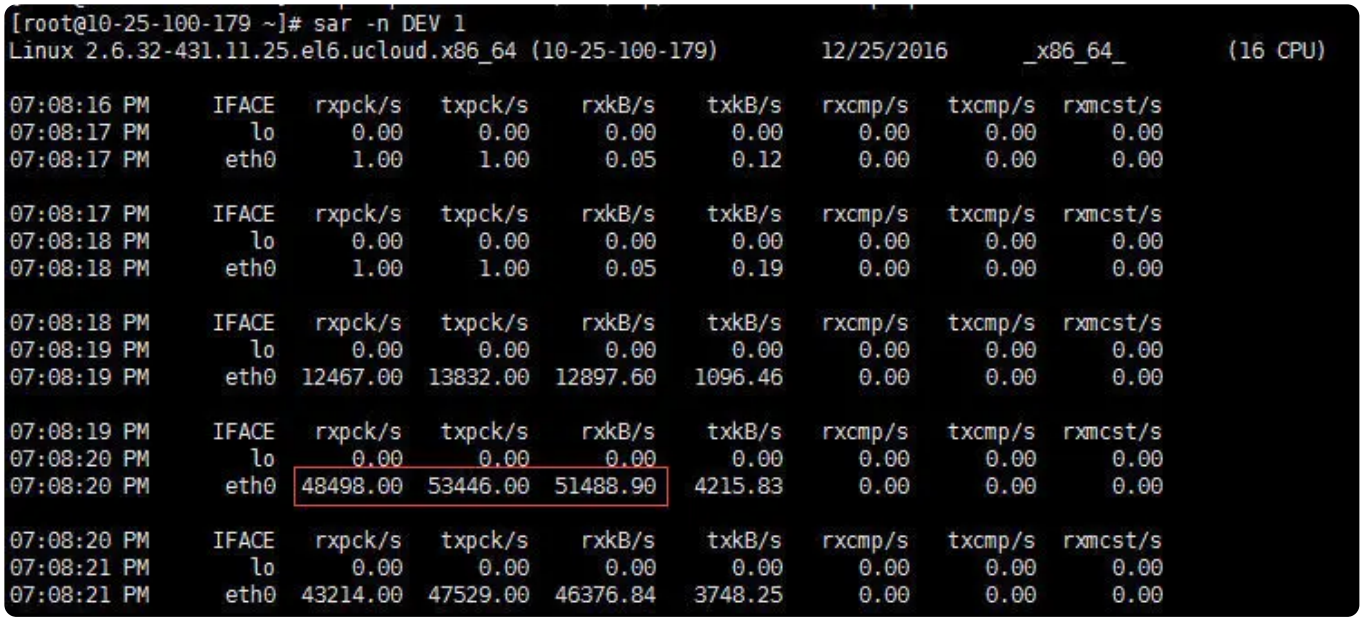
sar 是 sysstat 工具集的一部分。这个工具集包含了一些很有用的工具，除了 sar，还有 mpstat、iostat 等等。如果你平时经常做操作系统维护方面的工作，应该对它们比较熟悉了。这些工具还有一个共同的特点，运行的时候一般都是加上“间隔 次数”这样的参数的。也就是下面这样：

复制代码

```
sar -n DEV 1 10   #查看网卡性能
iostat 1 10       #查看IO性能
mpstat 2 5        #查看CPU性能
```

这些工具会按指定的时间间隔，运行指定的次数后再退出。这样我们就可以观察到这段时间内的多次输出了，很适合通过这种多次的观察，得到比较准确的性能数据趋势。

这次我们也在被压测的服务端云主机上运行了 `sar -n DEV`，得到了以下的性能数据：



```
[root@10-25-100-179 ~]# sar -n DEV 1
Linux 2.6.32-431.11.25.el6.ucloud.x86_64 (10-25-100-179)        12/25/2016      _x86_64_        (16 CPU)

07:08:16 PM     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s
07:08:17 PM        lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00
07:08:17 PM      eth0      1.00      1.00      0.05      0.12      0.00      0.00      0.00

07:08:17 PM     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s
07:08:18 PM        lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00
07:08:18 PM      eth0      1.00      1.00      0.05      0.19      0.00      0.00      0.00

07:08:18 PM     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s
07:08:19 PM        lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00
07:08:19 PM      eth0  12467.00  13832.00  12897.60   1096.46      0.00      0.00      0.00

07:08:19 PM     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s
07:08:20 PM        lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00
07:08:20 PM      eth0  48498.00  53446.00  51488.90   4215.83      0.00      0.00      0.00

07:08:20 PM     IFACE   rxpck/s   txpck/s    rxkB/s    txkB/s   rxcmp/s   txcmp/s  rxmcst/s
07:08:21 PM        lo      0.00      0.00      0.00      0.00      0.00      0.00      0.00
07:08:21 PM      eth0  43214.00  47529.00  46376.84   3748.25      0.00      0.00      0.00
```

前面说的包量，就是体现在两个 pck/s 指标中，一个是 **rxpck/s**，就是接收方向的包量；一个是 **txpck/s**，就是发送方向的包量。显然，图中的数值已经在 5 万左右，这个正是当时我们云主机性能的上限。难怪服务端已经无法提供更高的 TPS 了，因为网络包的处理都来不及做了。

那你可能会问了：“网络包也有大有小，这个包量指标说的是大包还是小包呢？”

其实，一般的包量测试，不是随便什么大小的报文都可以测试，而是普遍使用 64 字节长度的 IP 报文。另外，我们也要认识到，包的大小对包量性能的影响也不是很大。这是因为，对于网络处理来说，主要的开销在包的头部的处理上，而载荷本身的处理是很快的。

那么，对于客户遇到的这种包量达到上限的情况，我们可以选择的应对办法是这样的：

选择更高网络性能的主机，比如硬件的 RSS、软件的 RPS 等特性，都会大幅提升包量处理性能。

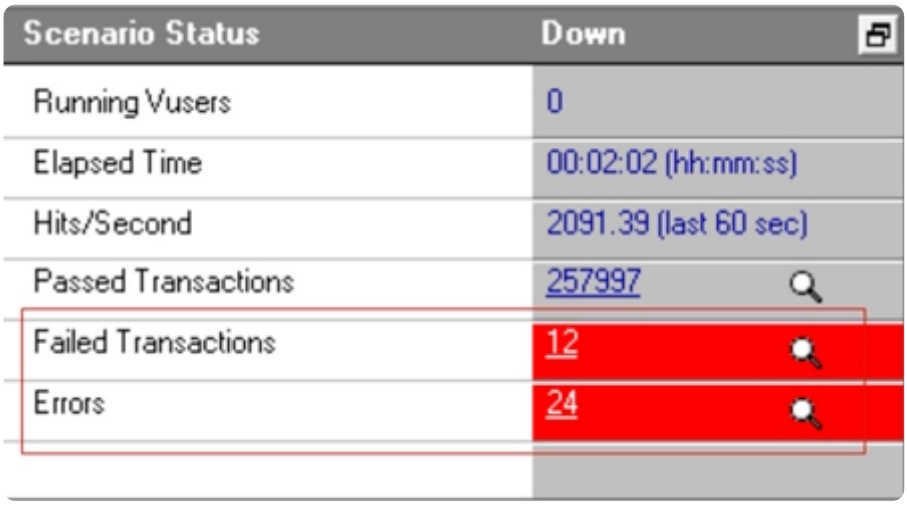
对服务集群进行水平扩展，也就是在 LB 后面增加服务器，这样 VIP 作为一个整体提供的包，处理能力也就提升了。

案例 2：LoadRunner 压测发现部分失败

这是另外一个客户，他们没有用 ab 这样的简单工具，而是用了 LoadRunner 这样一个企业级的测试软件。LoadRunner 的厉害之处在于，不仅可以发起巨大的请求量，而且可以

模拟用户的复杂行为，比如登录、浏览、加入购物车等等。这一系列事务有前后状态关系，这就不是简单的 ab 可以做到的了。

但是测试结果中的小几十个 Failed 和 Error 引起了客户的疑虑。他们怀疑：是不是我们公有云的机器或者网络质量不行，所以才导致了这些失败呢？



我们就尝试复现这个压测场景，同时也对测试机上的 TCP 资源情况做检查，其中最主要的就是源端口了。

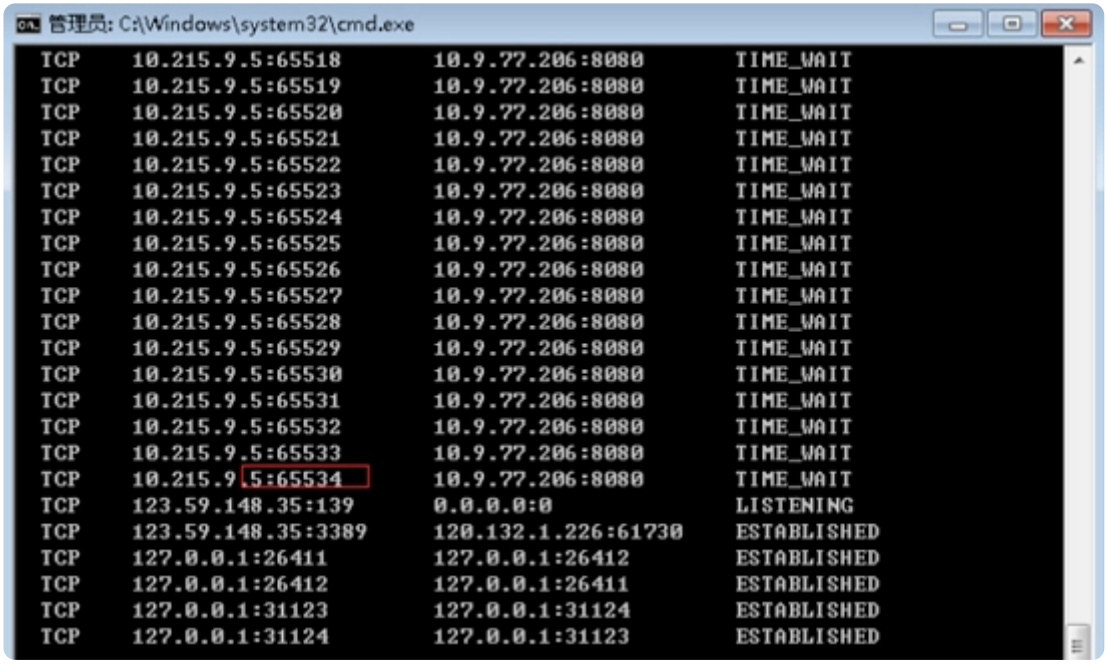
为什么会想到这个呢？因为压测发起的请求，都是依托于 TCP 连接的。我们在第 1 讲里就提到过：**TCP 连接是基于五元组的**。那么对于客户端来说，源 IP、目的 IP、目的端口、协议，这四个元素都不会变化，**唯一会变的就是自己的源端口**了。那么我们来看看，当时测试机的源端口情况。

这是一台 Window 机器，跟 Linux 类似，我们执行这条命令：

```
1 netstat -ant
```

复制代码

输出如下图：

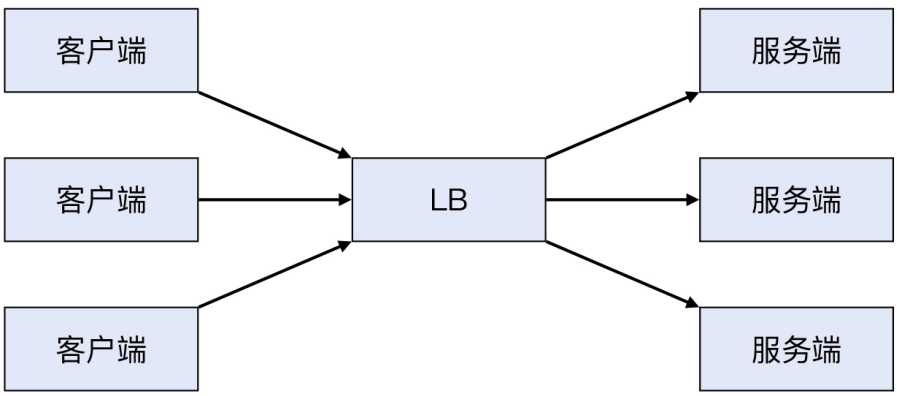


原来，确实有大量的 TCP 连接都在 TIME_WAIT 状态，尤其是源端口已经用到了 **65534**。所以可以肯定，这次压测中出现失败的原因，就在于源端口耗尽。

那要如何处理呢？显然我们也变不出更多的端口来。这个时候，我们可以**调整压测软件的设置，比如从短连接改成长连接**。这样就可以避免源端口耗尽的情况，因为所有的请求是在长连接里完成的，只要连接池本身设置合理，源端口就不会被用完。

案例 3：压测报 cannot assign requested address 错误

我们内部有一个团队在做业务压力测试，结果遇到了一个奇怪的报错：cannot assign requested address。这次的场景是这样的：这个团队从多台客户端机器向 LB 上的一个 VIP，发送大量的请求，而 LB 的后面就是很多的服务器。



但是，还没等到这些服务器的负载跑起来，客户端那边提前报错了。这是一个 Go 语言的程序，具体的报错信息如下：

```
1 https://test.vip/a": dial tcp 10.123.123.12:443: connect: cannot assign reques
2 http post Post \"https://test.vip/b": dial tcp 10.123.123.12:443: connect: can
```

于是他们就怀疑：既然报错是连接出错，那是不是 LB 有什么问题呢？比如，是否 LB 本身处理能力不够，不能服务这么大量的连接请求？否则为什么客户端会报 connect 的错误呢？要知道，用来发起 TCP 连接的系统调用（syscall）就是 connect。

确实，这个报错信息“dial tcp 10.123.123.12:443: connect: cannot assign requested address”说得不太清楚。表面上看，就是客户端往 10.123.123.12:443 这个 VIP 发起连接请求（用 connect）然后遇到了报错，也难怪测试团队会找到我们来查看 LB 的问题。

我们检查了 LB，发现一切正常。于是把排查方向换到客户端。在这次测试中，有一个参数是关于 HTTP Keep-alive 的。如果你对课程前面的内容还有印象的话，应该还记得我们在第 7 讲“保活机制”里，深入讨论过这个问题。

简单来说，这次的测试配置里，HTTP Keep-alive 没有打开，导致这些 TCP 连接被视作短连接来处理了，也就是一次 HTTP 请求和响应完成后，这条连接就关闭了。由于发起关闭的是客户端自己，于是这条连接也就进入了 TIME_WAIT 状态。

而要查看 TIME_WAIT 状态的连接数量，我们可以用 **netstat 命令**，配合管道和 awk 来完成统计。比如，这次我在一台客户端机器上执行了下面的命令：

```
1 $ netstat -ant |awk '{++a[$6]} END{for (i in a) print i, a[i]}'
2 TIME_WAIT 28231
```

或者用下面的 **ss 命令**会更快：

```
1 ss -ant | awk '{++s[$1]}END{for(k in s) print k,s[k]}'
```

```
2  TIME_WAIT 28231
```

可见，处于 TIME_WAIT 状态的连接数接近 3 万个，差不多就是 Linux 的本地动态端口的范围了。我们随后检查这台 Linux 机器的本地源端口范围，执行了下面的命令：

复制代码

```
1 $ cat /proc/sys/net/ipv4/ip_local_port_range
2 32768  60999
```

于是发现它的下限是 32768，上限是 60999，范围正好就是 28231，跟 TIME_WAIT 的数量一致，显然也是一次源端口耗尽导致的压测问题。

当然，用 sysctl 也一样可以查看这个范围：

复制代码

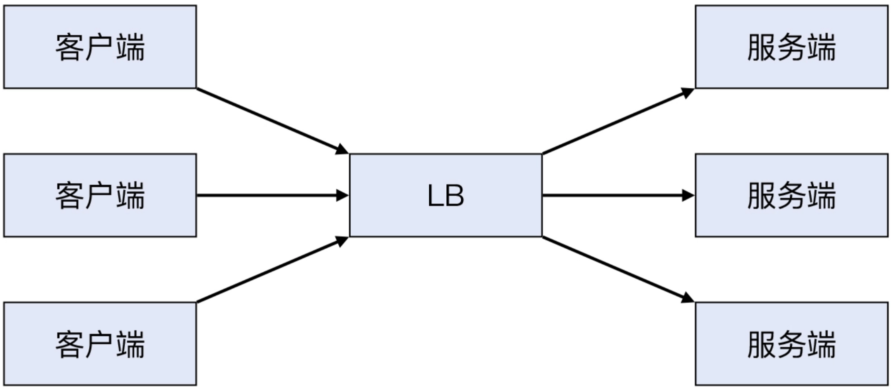
```
1 sysctl net.ipv4.ip_local_port_range
```

不过你可能会问：“难道压测中的网络排查，除了包量和源端口，就没有别的问题了吗？”

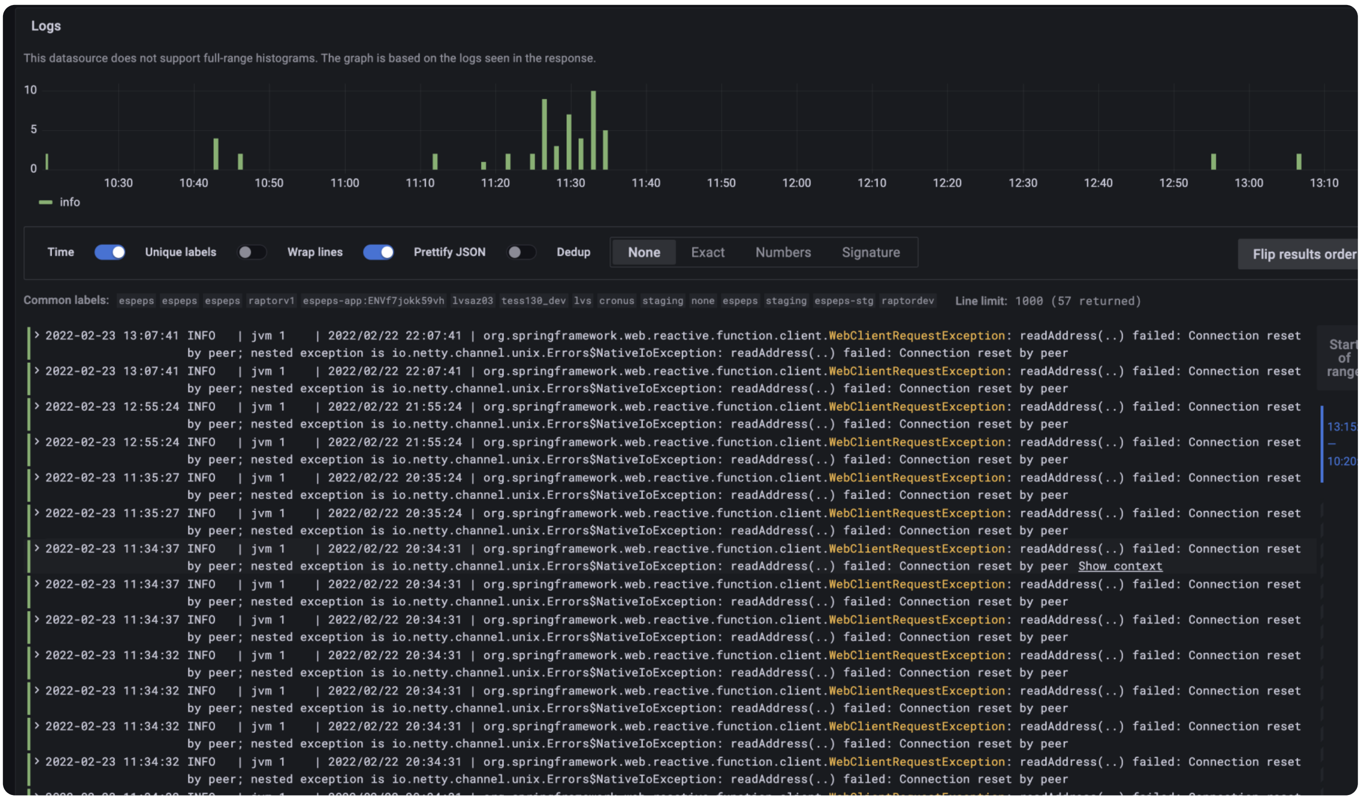
我们来看一个不一样的案例。

案例 4：压测遇到 connection reset by peer

这次的场景是一个应用团队用 netty http client 去调用一个 VIP 做压测。具体来说是 9 台客户端机器，向同一个 LB VIP 发起大量的请求。



结果遇到了 connection reset by peer。蹊跷的是，这个报错是零零星星出现的。而像之前的例子，如果真的源端口用尽了，那么接下来一段时间内，这些请求都会因为本地没有源端口可用而宣告失败，也就是报错会是大面积出现，而不会零星出现。



从图上看，报错数量不超过 50 个，主要集中在 11 点 20 分到 11 点 35 分这个时间段，这正是压测的时段。因为报错只有 50 来个，这相比于这次压测发起的成千上万的请求来说，是很小的比例了。

> 补充：Y 轴就是报错的个数，最高值是 10，我们把几个柱体的高度加起来，就是报错的总数量了。

我们了解到，压测期间每个客户端的请求频率是 700TPS，所以 9 台客户端一共会发起 6300TPS 的请求量。这个问题诡异的地方就是，大部分的请求都能得到正确及时的回复，但是隔了两三分钟，就会出现几次这种 connection reset by peer 的问题。

那我们需要理解一下，**为什么会 reset**。

我们在做网络排查的时候，如果在 Wireshark 里看到 TCP RST，往往会觉得它不是一个好的征兆。确实，有时候是 RST 引起了故障，有时候又是网络故障迫使 TCP 用 RST 来结束连接。无论 RST 是因还是果，它总是跟问题本身逃不脱关系。

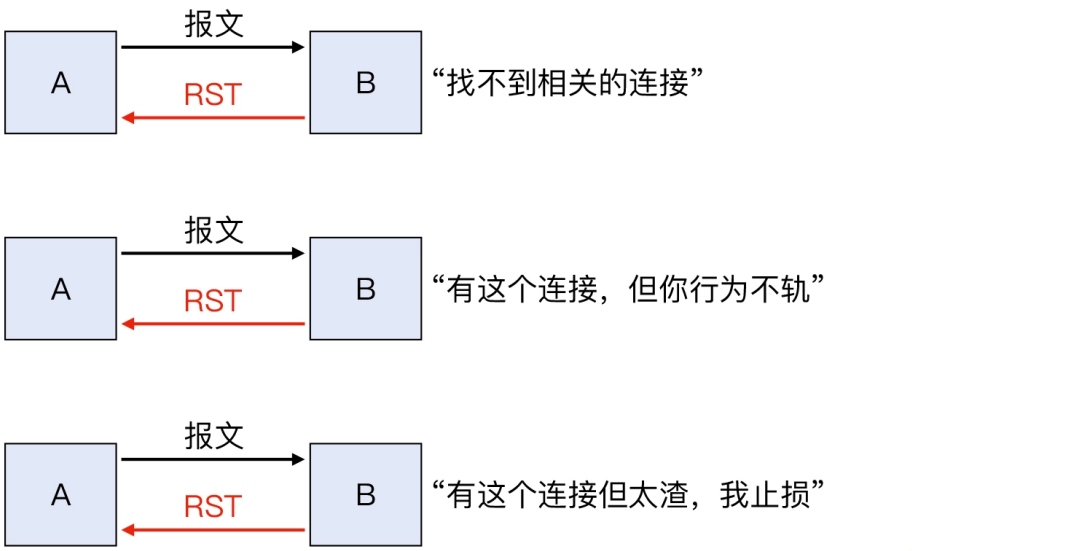
那干脆在 TCP 里取消 RST，是不是很多问题就会被解决呢？当然不是，RST 在 TCP 里面是一个非常必要的组成部分。没有 RST，其实就没有“坏”的结束，也就没有“好”的开始。

大体上，TCP RST 的原因可以分为这么几个大类：

找不到相关连接，那么接收端可以放心地直接发送 RST。

找到了相关连接，但收到的报文不符合 TCP 规范，那么接收端也可以发送 RST。

找到了相关连接，但传输状况恶劣，内核选择及时“止损”，发送 RST。

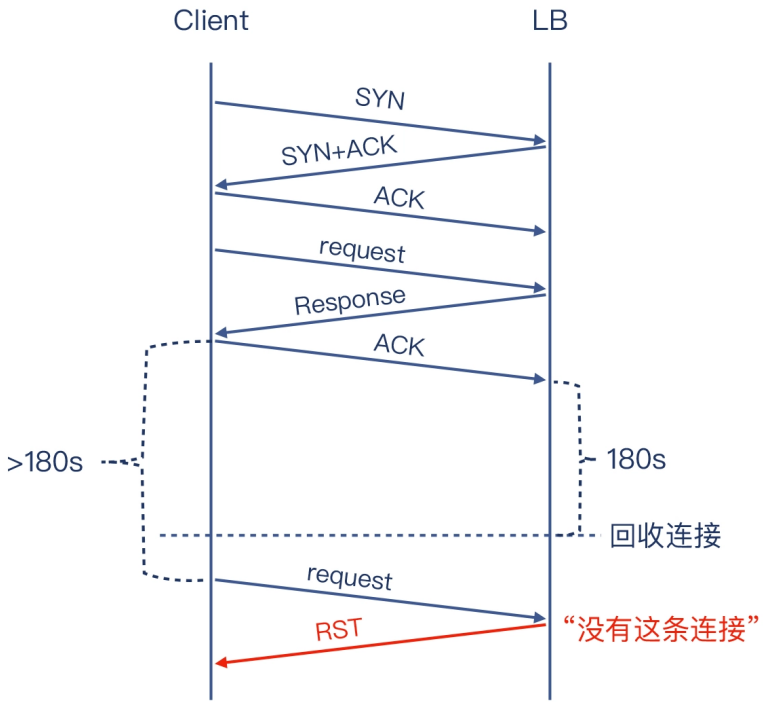


而这次案例里面的情况，就符合第一种。这是因为：

我们的 LB 上的 VIP 有一个 idle timeout 的设置，如果客户端在一定时限内不发任何报文，那么这条连接将被回收。这个时限是 180 秒，而且回收时不向客户端发送 FIN 或者 RST 报文。

这次的压测客户端框架里，有一个设置值也是关于 idle timeout 的。不过这个值设置的是 360 秒。

你有没有发现问题？这两个 idle timeout 值一边大一边小，配合不当，就会出现下面这样的问题：



某条 TCP 连接中完成一次 HTTP 请求和响应之后，连接没有被关闭。过了 180 秒，LB 这一侧的连接被回收了，但客户端那边还没到 360 秒，所以还认为这条连接是活着的，于是在 180 秒之后发起了一次请求。报文到达 LB，后者在连接表里没有查到这条连接，于是回复了 RST。

根因清楚了，解决起来异常简单：把客户端的 Idle timeout 参数，从原先的 360 秒，改成比 LB 的 180 秒更低的值就好了。这次改到了 120 秒，RST 就完全消失了。

你看，虽然在案例 2 里，我们知道了压测最好使用长连接，这样可以避免源端口耗尽的问题。但是不等于你设置了长连接就一劳永逸了，还需要考虑 idle timeout 的问题。这次压

测的关键在于：要确保长连接本身是真正有效的，你需要**确保客户端的 idle timeout 小于服务端的 idle timeout，这样才能避免连接失效导致的 RST**。

> 补充：这个库是 Ractor-Netty，idle timeout 对应的是 evictInBackground + maxLifeTime。

实验

回顾完这些压测案例，你可能会发现其实难度并不太大，是不是也跃跃欲试了呢？接下来，我们就用 ab 来做个小实验，这次还捎带一个小任务：**控制 TIME_WAIT 状态的连接的数量**。

在案例 2 和 3 里面，源端口耗尽的问题，本质是 TIME_WAIT 需要停留 2MSL 的时长。要解决 TIME_WAIT 连接过多的问题，方法有很多。这次我们只实验其中的一种方法，就是修改 `net.ipv4.tcp_max_tw_buckets` 这个内核参数。

它的默认值为 16384，改为更小的值后，超过这个数值的 TIME_WAIT 连接会被清除，这样，TIME_WAIT 连接数就被控制住了。

这个实验在本地就可以完成，比如在你的笔记本上安装 VirtualBox，然后在这个虚拟机里面完成实验。

步骤一：安装 Nginx 和 Apache ab。

复制代码

```
apt install nginx
apt install apache2-utils
```

步骤二：启动 ab 测试，执行下面的命令。

复制代码

```
ab -c 500 -n 10000 http://localhost/
```

步骤三：观察 TCP 连接的各种状态的统计数。

```
ss -ant | awk '{++s[$1]}END{for(k in s) print k,s[k]}'
```

复制代码

此时你应该会看到 1 万多个 TIME_WAIT 的连接，然后等待 2 分钟后继续步骤四。

步骤四：修改内核参数 tcp_max_tw_buckets 为 100。

```
sysctl net.ipv4.tcp_max_tw_buckets=100
```

复制代码

步骤五：再次 ab 测试。

```
ab -c 500 -n 10000 http://localhost/
```

复制代码

步骤六：再次观察 TCP 连接的各种状态的统计数。

```
ss -ant | awk '{++s[$1]}END{for(k in s) print k,s[k]}'
```

复制代码

此时 TIME_WAIT 应该只有 100 了。

当然，TIME_WAIT 本身被设计出来是有原因的，建议你在改动它之前，把自己的场景想清楚，然后再改不迟。

小结

这节课，我们回顾了几个典型的压力测试场景中的网络问题，你需要重点掌握以下这些知识点。

关于压测指标和工具。轻量级的工具是 ab，它十分方便，而且也可以发起大量的请求，在简单的压测场景下很有用。比如下面的命令：

```
ab -k -c 100 -n 10000 目标URL
```

复制代码

重量级工具是 LoadRunner 和 JMeter，它们可以实现复杂的交互行为，也提供更加详细的报告。

在网络性能领域，**包量是对 PPS 的简称，一般用来衡量一台主机的网络处理能力。**而评估包量的工具是 sar，我们可以运行下面的命令获取到实时的包量值：

```
sar -n DEV 1 10
```

复制代码

关于 TCP 的知识。这节课里我们也再次温习了 TCP 相关的知识，包括：

判断一个报文是否是之前报文的重传，可以根据两个关键信息：**序列号、载荷长度。**这两个值分别相同的多个报文，互相就是重传关系了。

压测数据起不来的一个常见原因是**源端口耗尽**，要验证这一点，可以执行以下命令，来查看 TIME_WAIT 或者 ESTABLISHED 状态的连接数是否到达了上限：

```
ss -ant | awk '{++s[$1]}END{for(k in s) print k,s[k]}'
```

复制代码

或者：

```
netstat -ant |awk '{++a[$6]} END{for (i in a) print i, a[i]}'
```

复制代码

而连接数到达上限值的问题，往往跟可用的**本地源端口范围**有关。要查看端口范围，我们可以执行这条命令：

```
sysctl net.ipv4.ip_local_port_range
```

复制代码

压测中遇到 RST 的原因，可能跟两端的 idle timeout 不合理有关，建议把客户端的 idle timeout 设置为比服务端 idle timeout 更低的值。

思考题

最后还是再给你留两道思考题：

测试 HTTP 性能经常用 ab，那测试带宽本身，一般用什么工具呢？

要把 TIME_WAIT 停留的时间 2MSL 改小，你知道用什么方法吗？

欢迎在留言区分享你的答案，也欢迎你把今天的内容分享给更多的朋友。

附录

抓包示例文件：https://gitee.com/steelvictor/network-analysis/tree/master/22

上一篇 21 | 为什么用了负载均衡更加不均衡？

下一篇 答疑（一）| 第1~5讲思考题答案

精选留言 (6)

写留言

JianXu

2022-03-12

当年蒋公领导Cms 项目，压测必须见到cpu，mem 或者IO 之一见顶，不然不算过。客户端压测必须有多机并发避免受限单机性能。需要做大小包测试，对应到cms 就是小的文档和大的文档压测。目标机也需要多机集群，因为牵涉到服务端同步。


Realm

2022-03-11

1 iperf 和 netperf 都是最常用的网络性能测试工具； 2 也可以通过修改内核参数，优化tw的问题 #启用 timewait 快速回收。 net.ipv4.tcp_tw_recycle = 1 #开启重用。允许将 TIME-WAIT sockets 重新用于新的 TCP 连接。 net.ipv4.tcp_tw_reuse = 1

展开

作者回复: 你的补充很好： ）

不过，tw_recycle这个配置从4.12内核开始被移除了。关于这个变更，这里有详细的说明：https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4396e46187ca5070219b81773c4e65088dac50cc


那时刻

2022-03-11

1.测试带宽，可以使用iPerf工具 2.Linux的TIME_WAIT貌似是hard code为60秒。而阿里云的机器可以通过 sysctl -w "net.ipv4.tcp_tw_timeout=[$TIME_VALUE] 来修改TIME_WAIT。 window下可以通过修改注册表 HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TCPIP\Parameters] "TcpTimedWaitDelay"=dword:0000001E

展开

作者回复: 1.是的

2.确实，Linux的这个值要修改只能重新编译内核了，阿里云的机器提供了方便的sysctl接口来修改这个值~


那时刻

2022-03-11

请问老师，文中提到：对于网络处理来说，主要的开销在包的头部的处理上，而载荷本身的处理是很快的。指的是内核需要处理头部信息所消耗的开销么？而载荷是由应用程序处理，不计入网络处理？

作者回复: 因为对操作系统来说，处理报文的开销主要就是对头部的拆解和封装，而对载荷做的结构化处理就很少，对载荷的操作主要是内存复制。比如一个64字节的报文和一个640字节的报文，对CPU的开销差别不大，但是带宽上的差异就大了，所以大报文测试可能先触及带宽瓶颈；小报文测试，可能先触及CPU瓶颈。


Chao
2022-03-11

Idle timeout 导致连接被rest。 遇到一个相同案例， 浏览器做法是遇到这种情况 进行了重试。 虽然有http headers 里可以申明 keepalive 超时时间 。 但好像没有客户端实现其读取并主动断开。

作者回复: 对，现代浏览器都有重试机制，可以对不少网络状况进行容错~
关于你提到的服务端返回的Keep-Alive头部，我们可以看下MDN的解释：
https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Keep-Alive

timeout: An integer that is the time in seconds that the host will allow an idle connection to remain open before it is closed. A connection is idle if no data is sent or received by a host. A host may keep an idle connection open for longer than timeout seconds, but the host should attempt to retain a connection for at least timeout seconds.
max: An integer that is the maximum number of requests that can be sent on this connection before closing it. Unless 0, this value is ignored for non-pipelined connections as another request will be sent in the next response. An HTTP pipeline can use it to limit the pipelining.

也就是说，timeout值是连接保持的最短时间，但可以超过这个时间。
max是指开启了http pipeline以后才有意义，但大部分http场景是不启用Pipeline的，所以这个值一般就是被忽略了：）

Liam
2022-03-11

带宽测试：iperf, pktgen 修改timewait的时间： 修改TCP_TIME_WAIT, 重新编译内核。默认2MSL是60s

作者回复: 是的，修改TW时间需要重新编译内核。国内有云厂商提供了定制版的linux，可以用sysctl来动态修改这个参数，就方便很多了~

共 2 条评论 >