我们是如何优化HAProxy以让其支持2,000,000个并发SSL连接的?

×

亲爱的读者：我们最近添加了一些个人消息定制功能，您只需选择感兴趣的技术主题，即可获取重要资讯的邮件和网页通知。

仔细上上图，我们可以发现两个信息：

1. 该机器建立了238万个TCP连接；
2. 此时内存使用量大约在48G左右。

看上去很赞吧？如果有人能够提供配置，并且在单台部署HAProxy的服务器上完成这样规模的调优，是不是更赞？本文将详细描述这个过程;)

本文是一系列关于HAProxy压力测试文章的最后一篇。如果有时间，建议读者能够先阅读本系列的其余两部分。这样能够更好的帮助我们了解本文所提及的内核级别调优需要的一些背景知识。

- HAProxy压力测试（第一部分）
- HAProxy压力测试（第二部分）

为了达到前面提到的效果，整个设置过程依赖许多小型组件。在贴出HAProxy的最终配置之前（如果读者实在没有耐心，可以滚动到底部直接查看），本文会按照笔者思路一步步实现最终目标。

我们要测试哪些内容

本文测试使用的组件软件是Haproxy 1.6版本。我们一直在生产环境的4核30GB内存服务器上使用该版本，不过目前所有的连接都不是基于SSL的。

本次实验我们将测试2个指标：

- 首先是CPU使用率，既我们将所有负载从非SSL改成SSL连接时的CPU使

用率上升情况。在这种情况下，由于建立SSL连接需要的5次握手和数据包加密，CPU使用率肯定会上升。
- 其次，我们希望能够测试出当前生产环境配置下，请求数和最大并发连接数的拐点。

我们需要获取第一个指标是因为我们即将全面铺开的特性依赖基于SSL的通信。而第二个指标的目的是能够合理安排生产环境中的HAProxy专用机器的数量。

测试使用的组件

- 多个施压机
- 几个不同配置的HAProxy 1.6机器：
  - 4核30GB内存
  - 16核30GB内存
  - 16核64GB内存
- 用于支撑这些并发连接的后端机器

HTTP和MQTT

如果读者阅读过本系列的第一篇文章，应该知道我们整个基础设施支持两种协议：

- HTTP
- MQTT

在我们的技术栈中，没有使用HTTP 2.0，因此没有持久化HTTP连接的功能性需求。因此在生产环境中单台HAProxy机器的连接数（输入+输出）大约在2*150k。虽然并发连接数不高，但是每秒请求数还是相当高的。

和HTTP协议不同，MQTT协议是另一种通信协议。它提供了服务质量（QoS）相关参数，同时也支持连接持久化。因此在MQTT信道上可以进行双向的持续通信。由于HAProxy可以支持MQTT（基于TCP）连接，我们统计到单台服务器高峰期有大约600-700k个TCP连接。

我们希望压力测试能够给出HTTP和MQTT两种协议连接数的具体数据。

针对HTTP应用服务器，有很多现成工具可以帮助我们完成压力测试并且能

够提供该诸如结果汇总、图表绘制等；但是对于MQTT协议，却鲜有类似的工具。我们自己开发了一个工具，但是在这种高压力场景下不够稳定。

因此我们决定使用HTTP压力测试客户端来模拟MQTT协议配置。很有意思吧？让我们继续。

初始设置

后面将会详细描述整个调优过程，希望能够对类似场景压力测试和性能测试有所帮助。

- 我们首先使用了一台16核30GB的机器来安装HAProxy。我们没有直接使用当前生产环境服务器的配置，因为预期HAProxy终端的SSL连接会消耗大量CPU。
- 在服务端，我们使用Node.js服务针对`ping`请求响应`pong`。
- 对于客户端，我们刚开始使用了Apache Bench。究其原因，主要是因为`ab`是HTTP协议压力测试工具中较为有名且比较稳定的工具，同时它提供了不错的汇总报告，能够帮助分析结果。

`ab` 提供了许多有用的参数，在后面的压力测试中也使用到了，例如：

- `-c` 并发量定义并发请求服务的数量；
- `-n` 请求数定义当前压力测试的总请求数；
- `-p 发送文件` 通过POST请求发送文件；

如果仔细观察这些参数，我们可以发现通过这三个参数可以组合出很多不同的用例。一个简单的`ab`命令例如：

```
ab -S -p post_smaller.txt -T application/json -q -n 100000 -c 3000 http://t
```

该命令的输出结果类似：

其中需要特别关注的数据有：

- 99%延时
- 每个请求耗时

- 失败请求数
- 每秒请求数

ab 最大的问题在于无法通过参数控制每秒请求数。我们只能通过调整并发度来获得期望的每秒请求数，这增加了尝试次数和错误几率。

万能的图表

我们不能通过多次随机压力的结果来得出结论，因为这样的数据没有意义。为了能够获取有意义的结果，必须设计一系列测试场景。因此我们参照了这个图：

该图表明，直到一个特定的点，随着请求总数的增长，延迟基本上没有变化。然而，当这个拐点之后，响应延迟几乎以指数级增长。这就是我们期望度量的机器或者配置的拐点。

Ganglia

在提供一些测试结果之前，首先介绍下Ganglia工具。

> Ganglia是一个为高性能计算系统（例如集群和网格计算）设计的可扩展分布式监控系统

下面的一些图表是我们一台服务器的监控数据截图，通过它们我们可以直观的了解Ganglia及其能够提供的图表信息。

看起来不错吧?

我们使用Ganglia来监控HAProxy服务器，以提供一些核心指标，包括：

1. TCP连接数：能够让我们了解当前系统上创建创建的TCP连接总数。注意，该数据是输入输出连接的总和。
2. 数据包收发量：HAProxy实际发送和接收到的TCP包数量。
3. 数据收发量：实际发送和接收的数据总量。
4. 内存：压力测试过程中服务器的内存变化情况。
5. 网络：了解压力测试过程中的网络带宽情况。

以下是基于之前的测试所得到的一些限制，也是我们希望在这次压力测试中能够达到的成绩：

> - TCP连接数700k
> - 发送数据包50k个，接收数据包60k个
> - 收发数据量均在10 – 15MB字节左右
> - 内存消耗约14 – 15G
> - 网络带宽7MB

以上数据均为每秒数据

HAProxy Nbproc配置

刚开始对HAProxy进行压力测试的时候，我们发现增加了SSL之后CPU使用率一下子就飙升，但此时的每秒请求数却很低。通过top命令排查之后发现，HAProxy只使用了1个CPU内核，而我们的机器上还有15个空闲的内核。

通过Google发现HAProxy有一个设置可以使其充分利用多核。这个配置称为`nbproc`，详情及其具体配置可以参照这篇文章：

http://blog.onefellow.com/post/82478335338/haproxy-mapping-process-to-cpu-core-for-maximum

这项配置调优使得后面的压力测试能够继续，因为让HAProxy能够充分利用多核才能继续后续压力测试集中的各种混合场景。

使用AB进行压力测试

从这里开始正式进入压力测试环节，以后将不再赘述我们的度量数据和期望达到的指标。

刚开始我们的唯一目标是通过分析前面描述的ab命令参数变化，找到性能拐点。

上表是我们进行了多次压力测试之后的结果数据。为了获得这样的结果，我们进行了超过500次测试，从数据上可以明显看出，每次测试结果都有多项数据有变动。

单客户端问题

随着压力的逐渐增加，我们发现施压客户端成了瓶颈。从Apache bench文档来看，它在发起请求时只使用单核，并且没有设置可以利用多核提升其性能。

为了能够提升客户端性能，我们使用了Linux平台上的一个工具，叫做Parallel。如果其名称一样，该工具能够并行命令，以充分利用CPU核心。这也是我们所期望的。以下是使用Parallel运行多个ab客户端的示例命令：

```
cat hosts.txt |  parallel  'ab  -S -p post_smaller.txt -T
 application/json -n 100000 -c 3000 {}'
sachinm@ip-192-168-0-124:~$ cat hosts.txt
http://test.haproxy.in:80/ping
http://test.haproxy.in:80/ping
http://test.haproxy.in:80/ping
```

上述命令可以并行运行3个ab客户端，同时访问相同的URL。这样能够帮我我们解决客户端性能瓶颈。

服务端的Sleep和Times参数

前面我们提到了一些通过Ganglia收集的数据，这里先来讨论下如何模拟这些数据的产生。

1. 发送和接收数据包数量。该数据可以通过POST请求中发送一些数据来模拟。该方式也能帮助提升带宽和收发字节数两项Ganglia数据项。
2. TCP连接建立数。为了模拟该数据，着实让我们花了好多时间。试想如果一个请求响应时间是1秒，需要大约每秒700k个请求才能达到我们预订的场景。这个数据在生产环境中很容易达到，但是在我们的测试场景中却几乎不可能产生。

这时候读者可能会问，那么你们是怎么实现的呢？我们在POST请求参数中引入了sleep参数，可以通过该参数让服务端休眠特定毫秒之后再返回响应数据。这样能够模拟生产环境中的耗时请求。所以我们设置了休眠20分钟，这样每秒请求数达到583个左右的时候，就能够维持700k个连接数的水位。

除此之外，我们还在向HAProxy发起POST请求的时候引入了另一个参数：

times。该参数可以指定服务端在关闭TCP连接之前响应数据的重复次数。这样能够帮助我们模拟产生更多的数据量。

Apache Bench遇到的问题

虽然通过Apache Bench我们获取了很多结果数据，但同时也遇到了很多问题。这里不会说明所有遇到的问题，因为这不是本文的终点，而且后面将会介绍新的压测客户端。

我们对从Apache Bench获取到了不错的结果数据，但是每次获取一个点数据时，TCP连接数这个要求总是难以达到。不知道为何Apache Bench无法正确处理前面提到的sleep参数，同时也还是没法满足我们对容量的需求。

前面提到了我们在单台施压机上通过Parallel工具并行执行多个ab客户端，但这种方式无法跨多台施压机。当时还没有发现pdsh这个工具，也算是一个遗憾。

同时，之前的数据我们还缺少超时数据。在HAProxy上，我们会有一些默认的超时设置，但是ab客户端和后端服务完全忽略了这些配置。对此我们花了大量时间进行这方面的研究，试图修正压力测试的方案。

前面提到过拐点图，但是随着各种问题的出现，目标有所偏移。但是，为了获取有意义的结论，必须重新聚焦于此。

通过Apache Bench，我们获得了拐点，但是TCP连接数一直没有上升。这些数据基于5到6台施压机上运行的大约40到45个ab客户端获得，但是TCP连接数一直没能够达到期望的量级。理论上，随着sleep参数值的上升，TCP连接数应该会上升，但是事实上却没有效果。

引入Vegeta

基于使用Apache Bench所遇到的问题，我继续搜索其他功能更为强大、更易扩容的压力测试工具，最终找到了Vegeta。

从我个人经验来看，Vegeta相比于Apache Bench，非常易扩容，并且提供了更多的功能。在我们的压力测试场景中，一个Vegeta客户端可以产生相当于15个Apache Bench客户端的吞吐量。

下面会介绍使用Vegeta获取到的压力测试结果。

使用Vegeta进行压力测试

首先来看下单个Vegeta客户端的使用命令。有趣的是，其中用来向后端服务器施压的命令参数叫做`attack`:p

```
echo "POST https://test.haproxy.in:443/ping" | vegeta -cpus=
32 attack -duration=10m  -header="sleep:30000"  -body=
post_smaller.txt -rate=2000 -workers=500  | tee reports.bin | vegeta report
```

这里我们简单了解下Vegeta提供的一些参数细节:

1. `-cpus=32`，定义客户端使用的内核数量。为了能够达到需要的压力值，我们将施压机配置调整撑了32核64GB内存。仔细观察结果数据会发现，实际压力并不大，配置调整的主要目的是为了能够支撑大量状态为后端服务器休眠的连接。
2. `-duration=10m`，该参数顾名思义，如果没有指定执行时间，测试将永久运行。
3. `-rate=2000`，每秒钟请求数。

从上图可以看出，我们仅仅使用一台4核机器，就达到了每秒32k个请求。这个结果比之前得出的拐点图有更高的性能，这里针对非SSL请求的拐点在31.5k。

下面是更多压力测试的数据:

对于SSL连接来说，每秒请求数16k也不算差了。由于更换了新的客户端，整个压力测试过程都需要从头开始，但是从整体来看，结果都优于ab客户端的结果。

随着CPU内核数的增加，在相同压力下响应延迟都有所降低，直到压力达到CPU性能极限。

但是，我们发现当CPU内核数从8增加到16的时候，每秒请求数没有太多的增长。不过如果我们最终决定在生产环境使用8核机器，也不可能将所有核

心都分配给HAProxy而不被其他任何进程占用。因此我们决定使用6核心进行一些测试，验证是否可以接受性能结果。

也不差吧。

引入sleep参数

目前为止，我们对于压力测试的结果非常满意。然而，现在场景没有模拟真实生产环境场景，直到引入了sleep参数。

```
echo "POST https://test.haproxy.in:443/ping" | vegeta
-cpus=32 attack -duration=10m  -header="sleep:1000"
 -body=post_smaller.txt-rate=2000 -workers=500  | tee reports.bin | vegeta
```

上述命令设置了1000毫秒的休眠时间，会让服务端应用随机休眠0到1000毫秒。因此上述命令的平均延迟是⩾ 500ms。

最后一个单元格中的数字分别表示：

```
TCP连接建立数，包发送数，包接收数
```

从中我们可以清晰看出，在6核心机器上，最大每秒请求数从20k降低到了8k。显然，增加了休眠时间之后，由于大量的TCP连接数，对结果产生了较大影响。不过此时总的连接数已经接近我们期望的700k的水位。

里程碑 #1

我们如何增加TCP连接数？非常简单，只需要通过sleep参数增加休眠时间。我们持续增加该参数，最终停留在60秒，既最终平均延迟在30秒左右。

Vegeta提供了一个有趣的参数：请求成功率。我们发现在该休眠时间下，只有50%的请求成功率。参见下面数据：

通过设置60000毫秒休眠时间，我们达到了高达400k TCP连接数，同时每秒请求数8k的结果。图表中60000R中的R代表随机。

对此我们发现的第一个问题是Vegeta的默认请求超时时间是30秒，因此会有

50%的请求失败。因此我们将这个超时时间设置撑了70s，以避免在后续测试到再次遇到。

修改了客户端超时时间之后，我们很容易的达到了700k个连接。唯一的问题是连接数这个数据不稳定，只是峰值数据能够达到。因此系统峰值连接数能够达到600k到700k，但是并不能长时间维持。

我们希望能够达到这样的效果：

该图展示了连接数稳定维持在780k。上面表格数据中，每秒钟请求数非常高，但在实际生产环境中，单台HAProxy机器上的请求数要少很多（大约在300左右）。

但是我们如果大幅削减生产环境中的HAProxy机器（目前大约在30台，这意味着集群每秒请求数为30*300大约9k），首先遇到的瓶颈会是TCP连接数，而不是CPU。

> 因此我们决定尝试验证每秒请求数900、网络带宽30MB/s和2.1M TCP连接数的场景下验证。我们使用该场景是因为这是生产环境单台HAProxy机器压力的3倍。

另外，目前仅仅给HAProxy分配了6个内核。我们希望测试分配3个内核时的性能，因为这是我们模拟生产环境机器配置的最简单方式（前面提到过，我们的生产环境机器配置是4核30GB内存。）因此设置`nbproc = 3`是最方便的方式。

记住现在我们使用的机器是16核30GB内存，其中3个内核分配给HAProxy。

里程碑 #2

我们获得了不同机器配置下的每秒请求数上限，现在我们仅剩下前面提到的一个任务：达到生产环境的3倍负载

- 每秒请求数900
- TCP连接数2.1m
- 网络带宽30MB/s

我们再次在达到220k个TCP连接数上受阻。无论怎么设置休眠时间，TCP连接数就是无法再上升。

让我们来计算下，220k个TCP连接和每秒请求数900，110,000 / 900 ~= 120秒。这里使用110k因为220k个连接同时包含了输入和输出，既双向总数。

这让我们怀疑2分钟是系统某处的一个限制，通过查看HAProxy日志可以验证，日志中大部分连接的总耗时都在120000毫秒。

```
Mar 23 13:24:24 localhost haproxy[53750]: 172.168.0.232:48380
[23/Mar/2017:13:22:22.686] api~ api-backend/http31 39/0/2062/-1/122101
 -1 0 - - SD-- 1714/1714/1678/35/0 0/0 {0,"","",""}
"POST /ping HTTP/1.1"
```

其中122101是总处理时长。日志中所有字段详细值参见HAProxy文档。

经过进一步研究，我们发现Node.js有2分钟的默认超时时间。

具体信息参见下面一些资料：

- 如何修改Node.js请求默认超时时间
- Node.js Http server文档

解决了超时时间之后，事情并没有想象中的顺利。当连接数达到1.3m个时，HAProxy连接数突然下降到0,然后再次开始上升。通过dmesg命令查看内核日志之后发现，该现象是系统内存不足造成的。通过更换成16核64GB内存，并设置`nbproc = 3`之后，最终达到了2.4m个连接。

后端代码

下面是HAProxy后端服务的源码。我们在代码中使用了statsd库，以获取服务端每秒请求数。

```
var http = require('http');
var createStatsd = require('uber-statsd-client');
qs = require('querystring');
var sdc = createStatsd({
host: '172.168.0.134',
port: 8125
```

```
});
var argv = process.argv;
var port = argv[2];
function randomIntInc (low, high)
{
    return Math.floor(Math.random() * (high - low + 1) + low);
}
function sendResponse(res,times, old_sleep)
{
    res.write('pong');
    if(times==0)
    {
        res.end();
    }
    else
    {
        sleep = randomIntInc(0, old_sleep+1);
        setTimeout(sendResponse, sleep, res,times-1, old_sleep);
    }
}
var server = http.createServer(function(req, res)
{
   headers = req.headers;
   old_sleep = parseInt(headers["sleep"]);
   times = headers["times"] || 0;
   sleep = randomIntInc(0, old_sleep+1);
   console.log(sleep);
   sdc.increment("ssl.server.http");
   res.writeHead(200);
   setTimeout(sendResponse, sleep, res, times, old_sleep)
});
server.timeout = 3600000;
server.listen(port);
```

同时我们还有一个小脚本来运行多个后端服务。整个测试中，我们使用了8台服务器，每台服务器上运行了10个后端服务进程，以避免后端服务称为压力测试的瓶颈。

```
counter=0
while [ $counter -le 9 ]
do
   port=$((8282+$counter))
   nodejs /opt/local/share/test-tools/HikeCLI/nodeclient/httpserver.js $por
   echo "Server created on port "  $port
   ((counter++))
done
```

```
echo "Created all servers"
```

客户端代码

对于客户端，每个IP有63k个TCP连接的限制。如果对此不了解，参见本系列的前面一篇文章。

因此为了达到2.4m个连接（双向连接，对于客户端来说要发起1.2m个连接），我们需要大约20台机器。在所有20台机器上同时运行Vegeta命令非常痛苦，即使使用了类似csshx工具，仍然需要从所有Vegeta合并最终测试结果。

脚本如下：

```
result_file=$1
declare -a machines=("172.168.0.138" "172.168.0.141
" "172.168.0.142" "172.168.0.18" "
172.168.0.5" "172.168.0.122" "172.168.0.123" "
172.168.0.124" "172.168.0.232" " 172.168.0.24
4" "172.168.0.170" "172.168.0.179" "
172.168.0.59" "172.168.0.68" "172.168.0.137" "
;172.168.0.155" "172.168.0.154" "172.168.0.45" "
172.168.0.136" "172.168.0.143")
bins=""
commas=""
for i in "${machines[@]}"; do bins=$bins","$i".
bin"; commas=$commas","$i;  done;
bins=${bins:1}
commas=${commas:1}
pdsh -b -w "$commas" 'echo "POST
http://test.haproxy.in:80/ping" | /home/sachinm/.linuxbrew/bin/vegeta -cpus
attack -connections=1000000 -header="sleep:20" -header="
times:2" -body=post_smaller.txt -timeout=2h -rate=3000 -workers=
500 > ' $result_file for i in "${machines[@]}"; do  scp sachinm
@$i:/home/sachinm/$result_file $i.bin ; done;
vegeta report -inputs="$bins"
```

幸好这里使用了pdsh工具，使得我们能够在多台远程服务器上并行的执行命令。同时Vegeta也提供了结果合并功能，这也是我们急需的。

HAProxy配置

本节大概是读者最希望了解的内容，下面是我们在压力测试场景中使用的HAProxy配置。其中最重要的部分是nbproc和maxconn设置。其中maxconn设置允许HAProxy能够支持我们期望达到的TCP连接数。

maxconn 设置会影响HAProxy进程的ulimit，例如：

最大文件打开数设置到4m因为HAProxy的最大连接数设置成了2m。干净利落！

下文介绍了一些HAProxy优化方式，以达到我们期望的指标：

https://www.linangran.com/?p=547

这里从http30一直到http83 :p

以上就是本文所有内容，如果您能阅读到这里，我真心佩服 :)

这里要特别感谢Dheeraj Kumar Sidana，没有他的帮助，我无法或者如此多有意义的结果。:)

如果本文对您有所帮助请告诉我。同时，如果您认为本文有用，请推荐并帮助传播。

查看英文原文： how we fine tuned haproxy to achieve 2000000 concurrent ssl connections