HTTP Server Pressure Test Detail

索引

1.TCP 是什么
2.运行方式
3.浅析 HTTP

TCP 是什么

传输控制协议（英语：Transmission Control Protocol，缩写为 TCP）是一种面向连接的、可靠的、基于字节流的传输层通信协议。在简化的计算机网络 OSI 模型中，位于网络层（IP 层）之上，应用层之下。

使用场景：不同主机的应用层之间经常需要可靠的、像管道一样的连接。

运行方式

Demo 程序

TCP 协议的运行大致可分为三个阶段：链接创建、数据传输和链接断开，接下来通过一个简易的 Client/Server 来了解下这几个阶段。

- TCPClientDemo.py（阻塞）
- TCPServerDemo.py（IO 复用）

提问：ip 地址"0.0.0.0" 代表的是什么意思？"127.0.0.1" 呢？

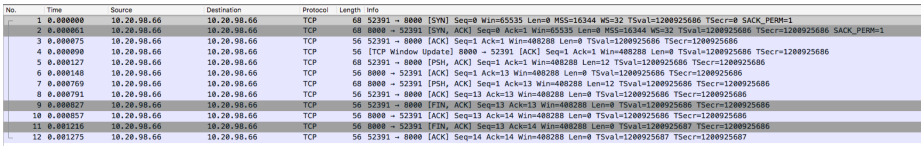
通过先执行 Server Demo，可以看出 TCP 的交互是先有一个以绑定好指定端口的 socket 在等待新链接的到来；接着运行的 ClientDemo，客户端 (client) 的 socket 连向刚被监听的端口，当完成链接创建的 “三次握手” 之后，client socket 的 connect 函数便完成并返回，下一步就可以进行数据的传输；client 先发送给 server 数据包，然后 server 返回该数据包后并将其内容打印出来，然后双方断开链接。

```
Greys-MacBook-Pro:HTTP_Server_Pressure_Test_Detail Grey$ python ./001_TCPServerDemo.py &
[1] 76171
Greys-MacBook-Pro:HTTP_Server_Pressure_Test_Detail Grey$
create server on port 8000
------------------------------------------------

Greys-MacBook-Pro:HTTP_Server_Pressure_Test_Detail Grey$ python ./001_TCPClientDemo.py
Incoming connection from ('10.20.98.66', 52391)

Server: recv: Hello, world
------------------------------------------------
Client: Received 'Hello, world'
Greys-MacBook-Pro:HTTP_Server_Pressure_Test_Detail Grey$
```

Figure 1: all_console_output

数据包

从上面打印出的 log 看，不足以让我们详细的了解 C / S 之间的交互过程，可以通过借助网络包分析工具来了解通信过程中 TCP 协议发送了哪些数据包。



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.20.98.66	10.20.98.66	TCP	68	52391 → 8000 [SYN] Seq=0 Win=65535 Len=0 MSS=16344 WS=32 TSval=1200925686 TSecr=0 SACK_PERM=1
2	0.000061	10.20.98.66	10.20.98.66	TCP	68	8000 → 52391 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=16344 WS=32 TSval=1200925686 TSecr=1200925686 SACK_PERM=1
3	0.000075	10.20.98.66	10.20.98.66	TCP	56	52391 → 8000 [ACK] Seq=1 Ack=1 Win=408288 Len=0 TSval=1200925686 TSecr=1200925686
4	0.000090	10.20.98.66	10.20.98.66	TCP	56	[TCP Window Update] 8000 → 52391 [ACK] Seq=1 Ack=1 Win=408288 Len=0 TSval=1200925686 TSecr=1200925686
5	0.000127	10.20.98.66	10.20.98.66	TCP	68	52391 → 8000 [PSH, ACK] Seq=1 Ack=1 Win=408288 Len=12 TSval=1200925686 TSecr=1200925686
6	0.000148	10.20.98.66	10.20.98.66	TCP	56	8000 → 52391 [ACK] Seq=1 Ack=13 Win=408288 Len=0 TSval=1200925686 TSecr=1200925686
7	0.000769	10.20.98.66	10.20.98.66	TCP	68	8000 → 52391 [PSH, ACK] Seq=1 Ack=13 Win=408288 Len=12 TSval=1200925686 TSecr=1200925686
8	0.000791	10.20.98.66	10.20.98.66	TCP	56	52391 → 8000 [ACK] Seq=13 Ack=13 Win=408288 Len=0 TSval=1200925686 TSecr=1200925686
9	0.000827	10.20.98.66	10.20.98.66	TCP	56	52391 → 8000 [FIN, ACK] Seq=13 Ack=13 Win=408288 Len=0 TSval=1200925686 TSecr=1200925686
10	0.000857	10.20.98.66	10.20.98.66	TCP	56	8000 → 52391 [ACK] Seq=13 Ack=14 Win=408288 Len=0 TSval=1200925686 TSecr=1200925686
11	0.001216	10.20.98.66	10.20.98.66	TCP	56	8000 → 52391 [FIN, ACK] Seq=13 Ack=14 Win=408288 Len=0 TSval=1200925687 TSecr=1200925686
12	0.001275	10.20.98.66	10.20.98.66	TCP	56	52391 → 8000 [ACK] Seq=14 Ack=14 Win=408288 Len=0 TSval=1200925687 TSecr=1200925687

* 数据包文件

一开始，服务端开始监听端口时，还未开始有数据包交互。接着，client socket 开始连接指定的服务端端口，上面我们说到，TCP 是像管道一样的连接，在 Demo 程序中，管道的一端是服务端的 8000 端口，另一端则是客户端在进行创建链接前，由操作系统随机生成的 52391 端口。

提问：`client socket` 的端口是否能跟服务端一样也指定某个特定端口?

链接创建：三次握手（three-way handshake）

链接创建时，先由客户端发送一个 SYN 数据包至准备连接的服务器端口，而后，服务端将返回一个 ACK 数据包表示对 SYN 数据包的确认同时也发送另外一个 SYN 数据包到客户端，然后，客户端再对这个返回的 SYN 数据包进行确认（发送 ACK 数据包），当这三个步骤完成时，也就表示链接创建的完成。(例图 No.1-No.3)

链接断开：四次挥手（four-way handshake）

TCP 链接是全双工的，即表示数据在两个方向上能同时传递数据，因此需要在两个方向上单独地进行关闭。当管道的其中一端收到 FIN 数据包时，意味着这个方向上已经完成了数据的传输，不再在这个方向上继续发送数据；当一端收到 FIN 时，仍可以继续发送数据，当发送完之后，也将发送另一个 FIN 完全关闭该链接。四次挥手指的是两次关闭连接的 FIN 数据包以及对应的 ACK 包用以做确认。(例图 No.9-No.12)

TCP 的状态机

尽管我们前面说道，TCP 是像管道一样的，可实际上，**网络上的传输时没有连接的，包括 TCP 也是一样**，它只是在通讯的双方维护一个 “连接状态”，让它看起来好像有连接一样。所以，认识 TCP 的状态变换是非常重要的。

上图便是三次握手、四次挥手的各个状态转换，下面再放一个完整的 TCP 状态机，方便读者做一个全局的参考。

浅析 HTTP

通过抓取访问本地启动的 SimpleHTTPServer，我们来了解下位于应用层的 HTTP 协议是如何通过 TCP 协议来进行通信的。

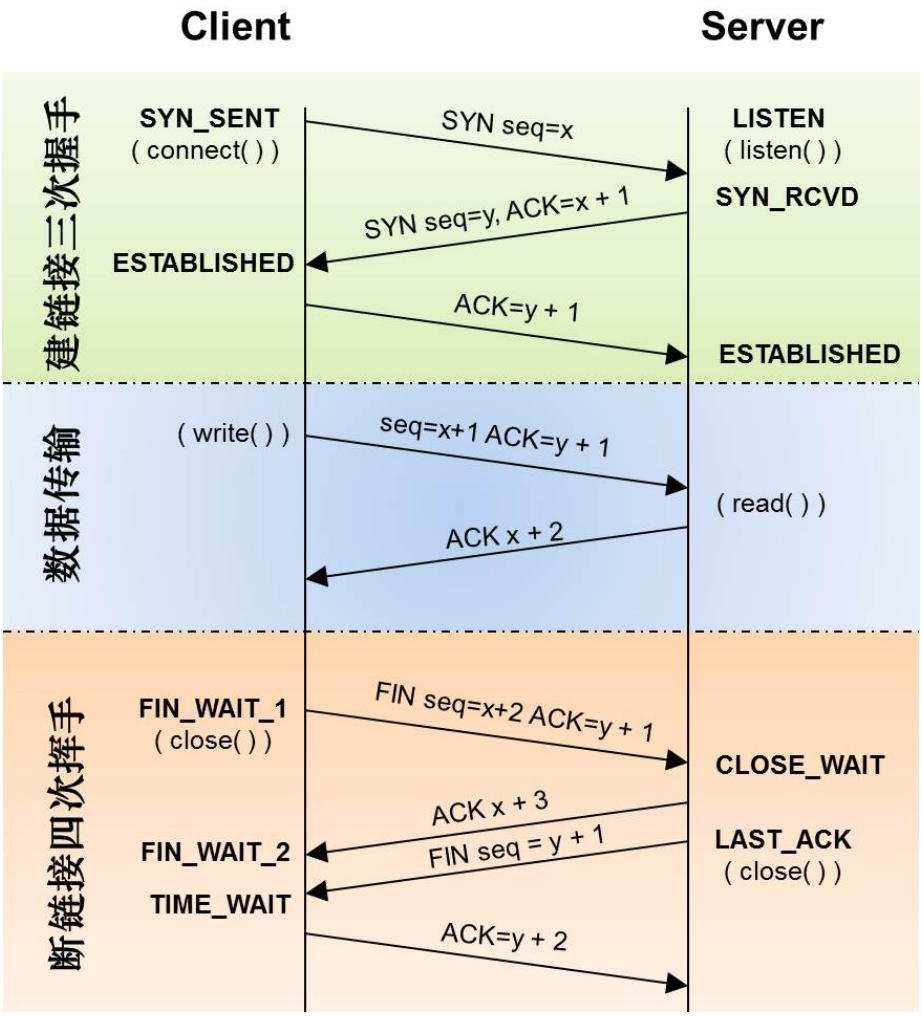


Figure 2: tcp_open_close

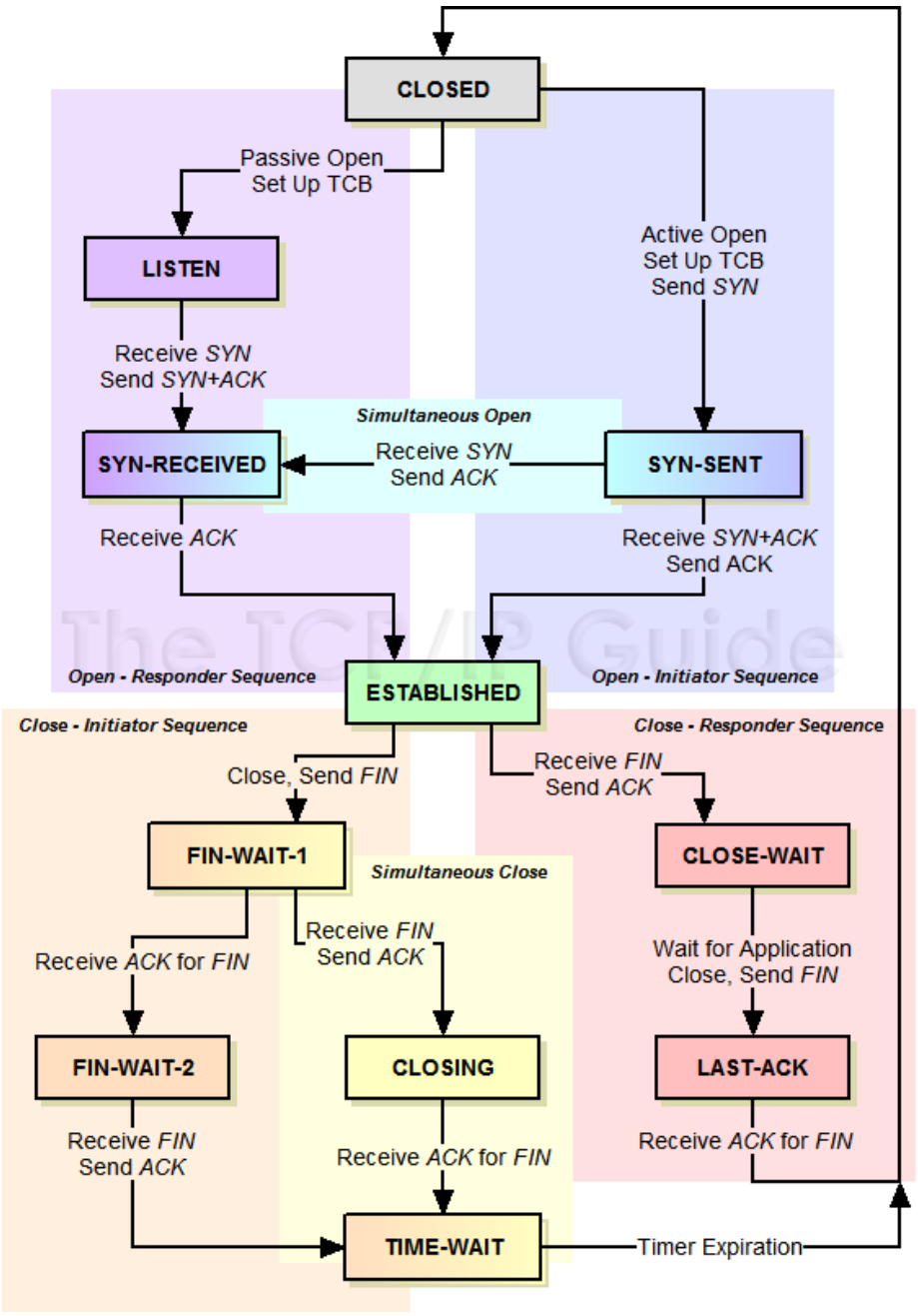
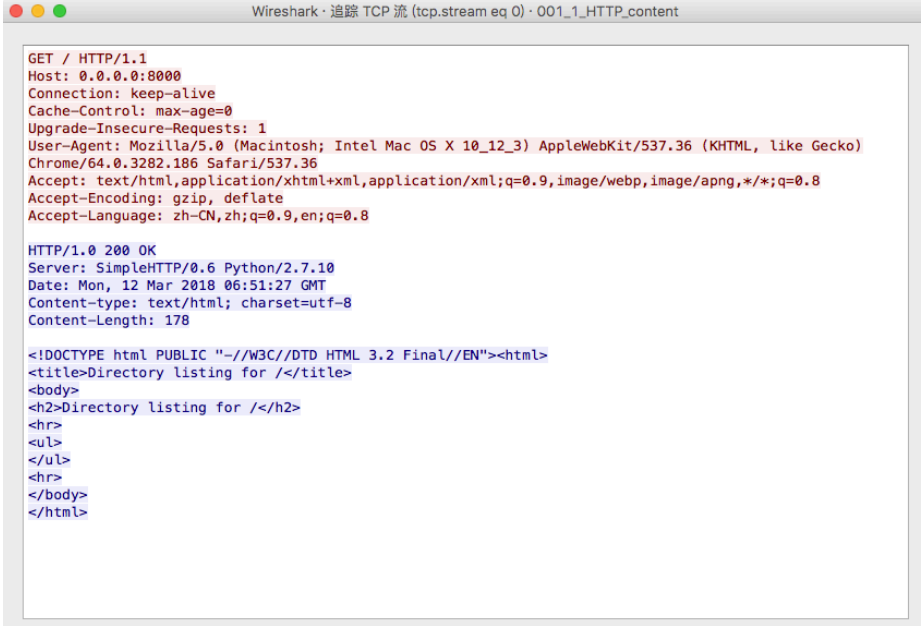


Figure 3: tcp_finite_state_machine

Tip: SimpleHTTPServer 可通过命令“python -m SimpleHTTPServer”启动，项目目录下已有一份已抓取好的网络包（001_1_HTTP_content.pcapng）



* HTTPClientDemo.py

查看 TCP 数据包的内容，我们可以看出，HTTP 协议是在 TCP 协议的基础上，使用带有特定格式的字符串内容。对于简单的 HTTP 接口调用，其实只是把访问的地址包装进特定位置，其他都采用默认值即可发起 HTTP 请求。

对 HTTP 服务器进行压力测试易出现的问题

为了方便演示，笔者找了额外的机器搭建了一个 Nginx 服务器用于演示。

在此次测试中，采用官方 TCP 网络库 asyncore 配合协程库 gevent 做一个简约的并发操作，为方便理解，请求头用一个固定简短的字符串：~~~python send_data_alive = ” ’
GET / HTTP/1.1 Host: 10.20.79.147:8000 Connection: keep-alive

” ’ ~~~

- HTTPClientDemoMultiAsync.py（并发 Demo）

在笔者的机器上，通过执行该并发程序，发起 500 个并发请求至服务器，响应正常，这样就可以确认我们的测试服务器是没有问题的，能负载住这样的并发压力（笔者并不是一开始就选中 Nginx 的，曾试用其他 HTTP 服务器，比如 python 自带的 SimpleHTTPServer，但效果不理想，经过多次挑选才找到的 Nginx，要验证问题，多个方案相互比较也是很重要的）。

但当反复执行这个并发程序多次的时候，出现了异常报错的信息。

```
Traceback (most recent call last):
  File "/usr/local/lib/python2.7/site-packages/gevent/greenlet.py", line 536, in run
    result = self._run(*self.args, **self.kwargs)
  File "001_HTTPClientDemoMultiAsync.py", line 16, in __init__
    self.connect( (host, PORT) )
  File "/usr/local/Cellar/python/2.7.14_2/Frameworks/Python.framework/Versions/2.7/lib/pytho
    raise socket.error(err, errorcode[err])
error: [Errno 49] EADDRNOTAVAIL
Tue Mar 13 15:04:35 2018 <Greenlet at 0x101f68870: HTTPClient('10.20.79.147')> failed with e
```

此时我们发现无法正常连接上服务器端口，但静止一段时间后再执行该并发程序又恢复正常，那么我们就可以推测出，程序代码应该可以正常使用的，可能是系统资源用完导致的错误，于是我们接着便开始对系统资源进行诊断。

首先，当程序抛出异常的时候，我们查看下当时系统的网络情况，可发现系统中出现了很多状态为 TIME_WAIT 的链接，数目与并发数一致：~~shell $ netstat -p tcp -n | awk '$6== "TIME_WAIT" {print $4}' | wc -l 500~~

接着,查看该系统的可用端口数,发现可用端口数只有 536 个:~~shell $ sysctl net.inet.ip.portrange.first net.inet.ip.portrange.last net.inet.ip.portrange.first: 65000 net.inet.ip.portrange.last: 65535~~

所以，当第一次并发程序执行结束退出的时候，系统的链接处于 TIME_WAIT 状态还没有完成释放，紧跟着执行多几次压测程序（成熟的压测程序是应该不间断持续进行的，这样耗光的速度会更快），便会把资源完全耗光，导致无法进行持续稳定的压力测试。

TIME_WAIT 缘由

在上面 TCP 的状态图中，从 TIME_WAIT 状态到 CLOSED 状态，有一个超时设置，这个超时设置是 2MSL（RFC793 定义了 MSL 为 2 分钟，Linux 设置成了 30s）为什么要这有 TIME_WAIT? 为什么不直接给转成 CLOSED 状态呢? 主要有两个原因： 1）TIME_WAIT 确保有足够的时间让对端收到了 ACK，如果被动关闭的那方没有收到 Ack，就会触发被动端重发 Fin，一来一去正好 2 个 MSL，* 2）有足够的时间让这个连接不会跟后面的连接混在一起（你要知道，有些自做主张的路由器会缓存 IP 数据包，如果连接被重用了，那么这些延迟收到的包就有可能会跟新连接混在一起）。所以，TIME_WAIT 很重要，而且，在大并发的短链接下，这个问题就变得更加严重。

解决思路

尽管这个状态非常耗时，且不可缺少，但是我们还是有办法避免这个问题。让我们回到 TCP 四次挥手的图表，我们可以发现，其实，TIME_WAIT 只出现在管道中的一端，所以这个状态要么发生在客户端要么发生在服务端。那么，发生在两端都是消耗资源，差别是什么呢? 答案是，如果发生在服务端，这个状态下的链接使用的端口是服务器监听的端口，所有在服务端的链接都使用同一个端口，这样将大大减少端口资源的占用(因为一个地址理论上最多只有 65535 个可用端口)；而去我们还可以看到，哪方先发起的关闭链接，哪方就将进入 TIME_WAIT 状态，所以，只要每次都由服务端先发起的关闭链接操作，那客户端就能避免这个问题。要由服务端关闭链接，我们只需要将请求头部的 Connection 属性设为 close 即可，读者可自行修改压测程序 demo，检验结果。

```
send_data_close = '''\
GET / HTTP/1.1
Host: 10.20.79.147:8000
Connection: close

'''
```

提问：服务端链接既然使用的是同一个端口，那是否可以创建无限多个链接？文件描述符又是什么？