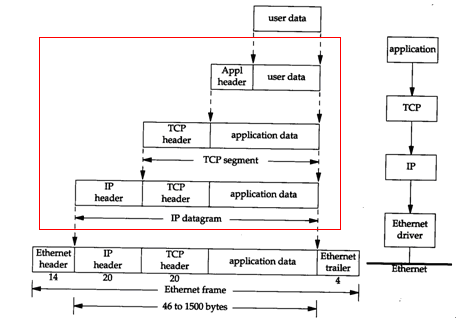
**用tcpdump学习TCP\_UDP协议**

**一 协议包结构**

**1 先来看看协议栈的结构**

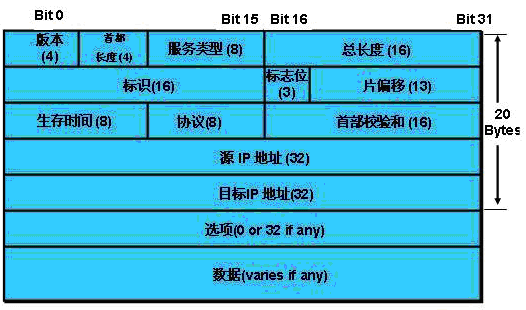
****

这里只讨论IP和TCP/UDP，以及应用层的关系。图中很清楚的表示出这几个层次之间的关系。从应用层往下来到TCP协议层，数据包加上了TCP包头，再往下到IP层，又加上了IP数据包头。

好，那接下来看看IP和TCP/UDP的协议包结构。

**2 IP和TCP/UDP的协议包结构**

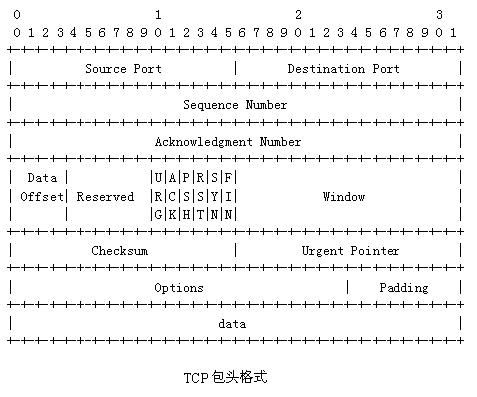
**(1) IP包**



首部长度是指IP包头长度（有可变选项，所以包头是变长的）。

总长度(16bit)字段是指整个IP包(IP header + data)的长度。后面tcpdump会发现，这个长度就是-X -s 0输出的完整包的长度。怎么就发现了呢？呵呵，在混沌的情况下，使用的办法总是比较土的。结合具体的应用层数据，往数据包外层推导，就发现了。

**(2)TCP包**



由于TCP协议比较复杂和讲究，所以有必要补充几句。

**Source port**(16bit)：源端口

**Destination port**(16bi)：目的端口

**Sequence Number**(32bit)：推送的数据的该数据包的开始序号。序号生成规则，有兴趣的话可以再查资料。

**Acknowledgment Number**(32bit)：确认接收数据的结束序号。就是说用来告诉推送方，已经接收到的数据情况。推送方会根据这个确认的结果，来决定是推送下一个数据包，还是重传。

**Data offset**(4bit)：头部数据偏移量，即tcp头部32bit的数量，要看tcp头部有多长（主要是后面的可变选项，所以TCP包头是变长的），需要找到这4bit的值。如果没有可变选项，TCP包头就是20字节。下面会结合具体的数据来加以说明，所以不清楚的不要着急，往下看就知道了。

**标志位**(6bit)：跳过Reserved字段，来看这6bit的标志位。这里说说其中几个标志。

**ACK**：确认数据标志，tcpdump输出里看到"ack"就是这个。

**PSH**：推送数据标志，tcpdump输出里看到的"P"就是这个。

**RST**：重置标志，这里说明连接关闭，告诉对方连接已经关闭。

**SYN**：连接标志，连接握手标志。

**FIN**：结束连接标志，告诉对方我要关闭连接了。

**Window**：发送数据窗口，即告诉对方，在不等确认的情况下，可以连续发来多大的数据。

**(3)UDP包**

|  |  |
| --- | --- |
| 源端口（16） | 目的端口（16） |
| 报文长（16） | 校验和（16） |

看完tcp的结构，再来看udp，简直太简单了。这里补充说一下16bit的报文长字段。因为UDP包头是定长的，就是图中看到的8个字节，所以这里的16bit的值是指整个UDP包（header+data）的字节数。

看完上面的3种包结构，你会发现，IP和UDP对数据包长度的界定都很清晰，交待得很清楚。为啥TCP只有一个开始序号呢？虽说TCP是一种流协议结构，但是也得知道流到哪里为止吧。

呵呵，这个问题其实跳出TCP这一层就很好理解了。TCP包是嵌在IP包里的，IP包头，IP数据包，还有TCP包头的长度都是有清楚交待的，那TCP包体的数据还能跑哪去？

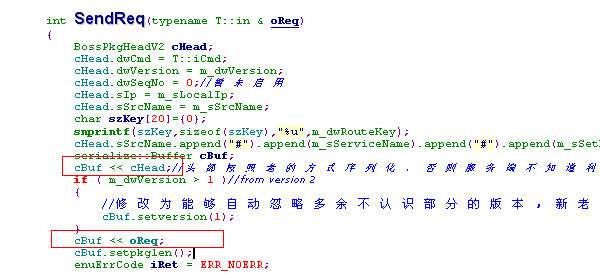
所以IP层和TCP层的关系是很密切的。

**二 tcpdump分析**

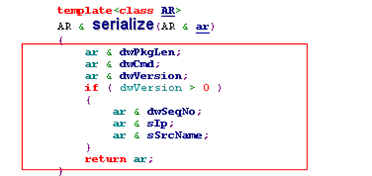
这里就拿小组内使用的appserver为例子来作分析(这里Appserver是一个tcp协议的服务框架)。

**1 首先来看客户端的数据结构**

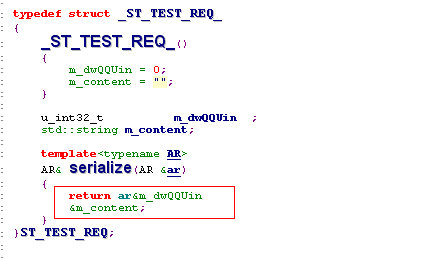
先来看SendReq这个函数，会发现序列化了两个结构体：cHead和oReq。



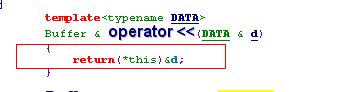
接着看这个cHead的序列化内容：

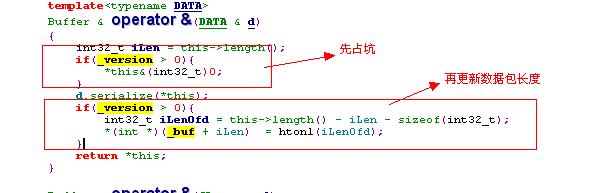


再来看我这里写的用于测试的oReq：



再加上客户端序列化的一些细节：





可见在buf的version大于0情况下，是先push一个int32\_t(0)，然后才是push结构体

由此，再回来看sendReq方法：

**先push cHead**（看SendReq的截图，此时buf的version还是初始值，等于0，所以不会在前面序列化一个int32\_t，下面push oReq之前，却先把buf的version置成1）

**然后push int32\_t** (push oReq时，buf的version等于1，即大于0，就先push这个length字段)

**然后push oReq**

由此再深入一层，看看客户端的应用层都序列化了哪些字段：

dwPkgLen(4) +

dwCmd(4) +

dwVersion(4) +

dwSeqNo(4) +

sIp (4+待定长度) +

sSrcName(4+待定长度)

(以上是cHead)

+ size(4) （此是version大于0是，序列oReq前先push的一个int32\_t）

+ m\_dwQQUin(4) 为方便调试，这里uin是36522855

+ m\_content ("1234567890" 4 +10) 这里m\_content的内容我且先设置成"1234567890"

(以上是oReq)

好了，应用层就这些数据了。下面使用tcpdump抓包分析，来把这些字段的待定内容补上。

这里顺便附上客户端的一些代码：



**2 tcpdump抓包**

好了，磨了这么久刀，要开始砍柴了。

(1) 命令：sudo /usr/sbin/tcpdump -i eth1 -nn -X -s 0 'port 19019'

-i 指定网卡，机器配置不同，这里的值也可能不同，比如可能eth0等，试一下就知道了。

-X 16进制和ASC码输出

-s 0 输出完整的包，不截断。

(2)输出，这里先说推送数据部分，后面再补充tcp连接，应答和关闭。

19:37:33.697526 IP 172.25.34.80.57905 > 10.6.223.63.19019: P 1:93(92) ack 1 win 1460 <nop,nop,timestamp 266125572 458835790>

0x0000: 4500 0090 63de 4000 4006 1edb ac19 2250 E...c.@.@....."P

0x0010: 0a06 df3f e231 4a4b 8b4d a2d9 9409 525d ...?.1JK.M....R]

0x0020: 8018 05b4 b831 0000 0101 080a 0fdc c104 .....1..........

0x0030: 1b59 474e 0000 005c 0007 0001 0000 0002 .YGN...\........

0x0040: 0000 0000 0000 000c 3137 322e 3235 2e33 ........172.25.3

0x0050: 342e 3830 0000 0022 2e2f 7373 5f74 6573 4.80..."./ss\_tes

0x0060: 745f 7374 7562 2373 735f 7465 7374 2323 t\_stub#ss\_test##

0x0070: 3133 3337 3835 3934 3533 0000 0012 022d 1337859453.....-

0x0080: 4b67 0000 000a 3132 3334 3536 3738 3930 Kg....1234567890

解析：

先看摘要行：

这一段有"P"标志，即推送数据(但有的推送请求没有P)；

1:93(92) 推送的序号从1到93，数据包长92=93-1。这个长度是应用层下发的，即tcp包体的内容的长度。 这里的序号都是相对序号，tcpdump已经做过处理。

ack 1 确认序号1，告诉对方可以发序号1往后的数据了。

Win 1460 窗口大小是1460

再看数据包内容。

可以看到左边是16进制输出，右边是asc码输出。

Asc码输出虽然只能看到一些普通文本，但也是很有用的。每个字符代表左边16进制的一个字节的输出，没法正常显示的，就用一个点"."表示，所以这里"."也代表一个char(8bit)，对应左边16进制一个字节的输出。因此，右边asc输出有时可以帮助更快的定位字段。

好了，对应上面的包内容，这里来层层解剖：

**IP包头：**

http://top.oa.com/pictures/201206/1339229225_46.bmp

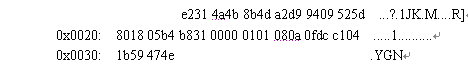
版本(4)

首部长度(5) 32bit的数量，这里是5\*32bit就是20 bytes

服务类型(00)

总长度(0090) 转成10进制，就是144。当然包含tcp包，这个长度就是tcpdump显示的总详细包内容的长度（-X选项才有的内容），从头4500到尾3930，算起来就是144字节。

好了，有了首部长度，我们把IP包头滑过，滑过20字节，来到TCP包：



**TCP包头：**

源端口（e231）即57905

目标端口(4a4b) 即19019

序列号(8b4d a2d9) 即2337120985，这个就是push的绝对序号了。

确认序号(9409 525d ) 即2483638877，这个是ack的绝对序号。

数据偏移量(8)，占4bit，即tcp头部32bit的数量，这里是8，就8\*32bit=32字节。所以要看tcp头部有多长（主要是后面的可变选项），需要找到这4bit的值

中间补0(reserved 6bit)

标志(18 6bit)

窗口(05b4)即1460这个值，和tcpdump看到的” win 1460”

校验和(b831)

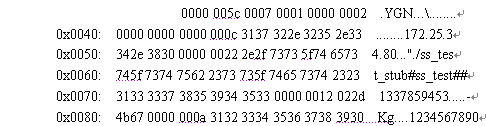
紧急数据指针(0000)

选项(0101 080a 0fdc c104 1b59 474e)

好了，这个时候我们可以来算一下接下来Tcp包体的长度了。

Tcp包体的长度=92 byte =IP包总(144) - IP头(20) – TCP头(32)，这个92就是tcp摘要行显示的长度。

好，接下来滑过tcp包头的32字节，来到tcp包体，这个就是我们应用层的内容！



先看右边，感性认知一下，"1234567890"想必就是m\_content字段吧，还有36522855这个QQ号，转成16进制，22D4B67，想必就是左边002d 4b67了吧。

然后结合确定长度的字段，推导不确定长度的字段，就可以把之前待定的内容不上了：

dwPkgLen(4) +

dwCmd(4)( 0007 0001) +

dwVersion(4)( 0000 0002) +

dwSeqNo(4)( 0000 0000) +

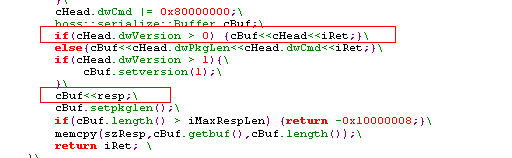
sIp("172.25.34.80")(4+12) (0000 000c )+

sSrcName(取自"/proc/%d/cmdline"，具体内容是"./ss\_test\_stub#ss\_test##1337859453" 4+34) = 70

+ size(4)

+ m\_dwQQUin(4) + m\_content("1234567890" 4 +10) = 92

上面说的都是客户端到服务端的请求包，这里补充一下服务端到客户端的回包：



发现一共序列化了这些字段：

cHead(…)+ iRet(4) + resp(4)=58

这里resp是int32。

Tcpdump输出：

19:37:33.698156 IP 10.6.223.63.19019 > 172.25.34.80.57905: P 1:79(78) ack 93 win 1448 <nop,nop,timestamp 458835790 266125572>

0x0000: 4500 0082 1514 4000 3a06 73b3 0a06 df3f E.....@.:.s....?

0x0010: ac19 2250 4a4b e231 9409 525d 8b4d a335 .."PJK.1..R].M.5

0x0020: 8018 05a8 22d6 0000 0101 080a 1b59 474e ...."........YGN

0x0030: 0fdc c104 0000 004e 8007 0001 0000 0002 .......N........

0x0040: 0000 0000 0000 000c 3137 322e 3235 2e33 ........172.25.3

0x0050: 342e 3830 0000 0022 2e2f 7373 5f74 6573 4.80..."./ss\_tes

0x0060: 745f 7374 7562 2373 735f 7465 7374 2323 t\_stub#ss\_test##

0x0070: 3133 3337 3835 3934 3533 0000 0000 0000 1337859453......

0x0080: 0001

用同样方法，剥掉IP包头，TCP包头，留下TCP包体，然后对应到我们的应用层的数据结构，就可以分析出每个字段的值。

cHead(…)+ iRet(4) + resp(4)

再展开：

dwPkgLen(4) +

dwCmd(4)( 8007 0001) +

dwVersion(4)( 0000 0002) +

dwSeqNo(4)( 0000 0000) +

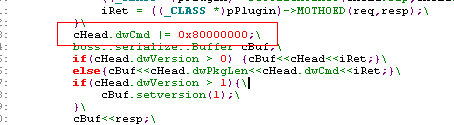
sIp("172.25.34.80")(4+12) (0000 000c )+

sSrcName("/proc/%d/cmdline" "./ss\_test\_stub#ss\_test##1337859453" 4+34)+

iRet(4) + resp(4)

有没有发现，刚才请求包的cmd是0007 0001，现在返回包的cmd怎么变成8007 0001了呢？

看看代码就知道了：



**三 TCP协议的补充**

**1 建立连接**

Gdb跟踪客户端进程，运行至：

nRet = connect(nSockfd, (struct sockaddr\*)&rServaddr, sizeof(rServaddr));

会出现下面的报文：

#客户端80到服务端63，S标志请求建立连接

21:03:17.963509 IP 172.25.34.80.52682 > 10.6.223.63.19019: S 1386379368:1386379368(0) win 5840 <mss 1448,sackOK,timestamp 224213029 0,nop,wscale 2>

#服务端应答ack，同时也发送S标志，请求建立连接。

21:03:17.963528 IP 10.6.223.63.19019 > 172.25.34.80.52682: S 1546827191:1546827191(0) ack 1386379369 win 5792 <mss 1460,sackOK,timestamp 416922939 224213029,nop,wscale 2>

#客户端应答ack

21:03:17.963906 IP 172.25.34.80.52682 > 10.6.223.63.19019: . ack 1 win 1460 <nop,nop,timestamp 224213029 416922939>

这里"S"标志后跟的序号就是绝对序号。

可以看到建立一个TCP连接，需要3次网络请求，所谓三次握手。所以高效的tcp服务需要采用长连接的方式，省去建立连接的开销。

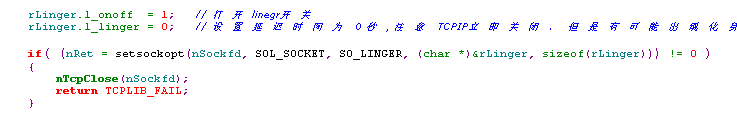
**2 关闭**

21:03:17.964903 IP 172.25.34.80.52682 > 10.6.223.63.19019: R 93:93(0) ack 79 win 1460 <nop,nop,timestamp 224213029 416922940>

这里客户端野蛮直接的给服务端发R标志，连接就这样关闭了。

为什么没有传说中的4次握手关闭连接？

来看代码：



注释掉这段代码，再看看：

12:22:42.574811 IP 172.25.34.80.45904 > 10.6.223.63.19019: F 10083:10083(0) ack 79 win 1460 <nop,nop,timestamp 324400975 517111620>

12:22:42.575200 IP 10.6.223.63.19019 > 172.25.34.80.45904: F 79:79(0) ack 10084 win 6516 <nop,nop,timestamp 517111620 324400975>

12:22:42.575211 IP 172.25.34.80.45904 > 10.6.223.63.19019: . ack 80 win 1460 <nop,nop,timestamp 324400975 517111620>

这时可以看到常规的关闭流程了。这里有三个网络请求，由于TCP是全双工的，就是来和去（即read和write）可以独立工作，所以第二步是可以拆成两步的。

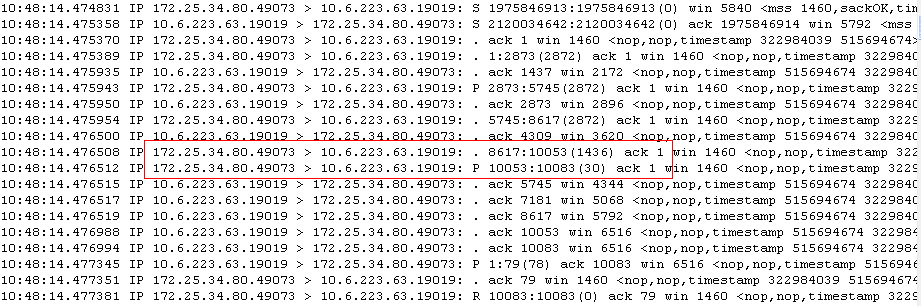
所以不用去纠结三次还是四次握手，本质上就是要有去有回。

因此，可以发现，我们的客户端在关闭这里是做了优化的。提高关闭的效率。

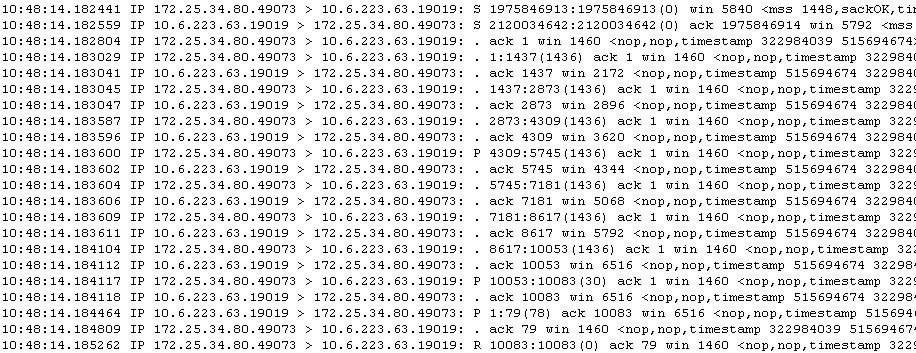
**3 发送数据和确认接收**

加大客户端的数据推送量(count置成1000)，看看发送数据这块情况。

客户端机器上的报文：



服务端机器上的报文：



发现推送数据有时有P标志，有时没有，查了下，说法是：数据推送完成或内核缓存满通知应用层接收数据。由于这里涉及到内核和用户层的交互，没法用gdb跟踪。

从上面的截图发现，客户端和服务端的tcpdump报文竟然是不一样的！服务端可以看到是非常工整的一去一回的方式，比如63对80 ack 2873（已经收到了2873序号之前的数据，告诉对方可以发2873了），然后接着才是80对63 推2873:4309(1436)。

而客户端则没有那么工整，会出现客户端连续往服务端推数据的情况（上面截图的红框部分）。

这么看的话，服务端是采用了Nagle算法的（简单讲就是一去一回的方式），而客户端是采用滑动窗口机制(简单讲就是有点批量发送然后一起确认的意思)。

比较郁闷的是，查了网上资料，不管我试图怎么关闭服务端Nagle算法，都不生效。Tcpdump看到都是那个样。

（尝试采用setsockopt(nSockfd,IPPROTO\_TCP,TCP\_NODELAY…);关闭Nagle算法）。

同样，不管我怎么想办法让客户端采用Nagle算法，也是不生效。

这个问题先抛出来，等待求解的那天。或许我这里的理解是有偏差的。

**四 UDP协议**

Udp协议会简单很多，这里简单说一下包结构吧。

命令：sudo /usr/sbin/tcpdump -i eth1 -nn -X -s 0 'port 19101'

输出：

11:35:18.320280 IP 10.130.136.162.48464 > 172.27.28.241.19101: UDP, length 94

0x0000: 4500 007a 0000 4000 3c11 e242 0a82 88a2 E..z..@.<..B....

0x0010: ac1b 1cf1 bd50 4a9d 0066 6220 0000 0001 .....PJ..fb.....

0x0020: 0000 000a 7573 6572 5f6c 6f67 696e 0000 ....user\_login..

0x0030: 0048 3835 3933 3837 3834 7c32 3230 2e32 .H85938784|220.2

0x0040: 3530 2e35 352e 3231 307c 3337 3037 3338 50.55.210|370738

0x0050: 3538 3130 7c31 3536 3934 3833 3430 3335 5810|15694834035

0x0060: 3431 3130 3532 7c32 3030 372d 3033 2d31 411052|2007-03-1

0x0070: 3120 3131 3a33 393a 3334 1.11:39:34

IP包头：

http://top.oa.com/pictures/201206/1339229517_90.bmp

版本(4)

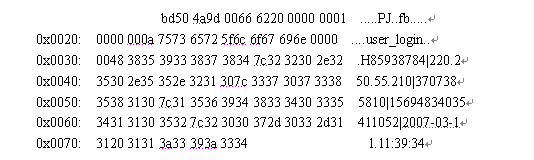
首部长度(5) 32bit的数量，这里是5\*32就是20 bytes

服务类型(00)

总长度(007a) 即122 当然包含udp包，这个长度就是tcpdump显示的总详细包内容的长度

标识(63de)

滑过IP包头，来到UDP包：



源端口：bd50 即48464

目地端口：4a9d 即19101

报文长：0066 即102

校验和：6220

UDP包内容长度= IP包总长(122) - IP包头(20) – UDP包头(8)=94，这个就是tcpdump汇总行显示的长度