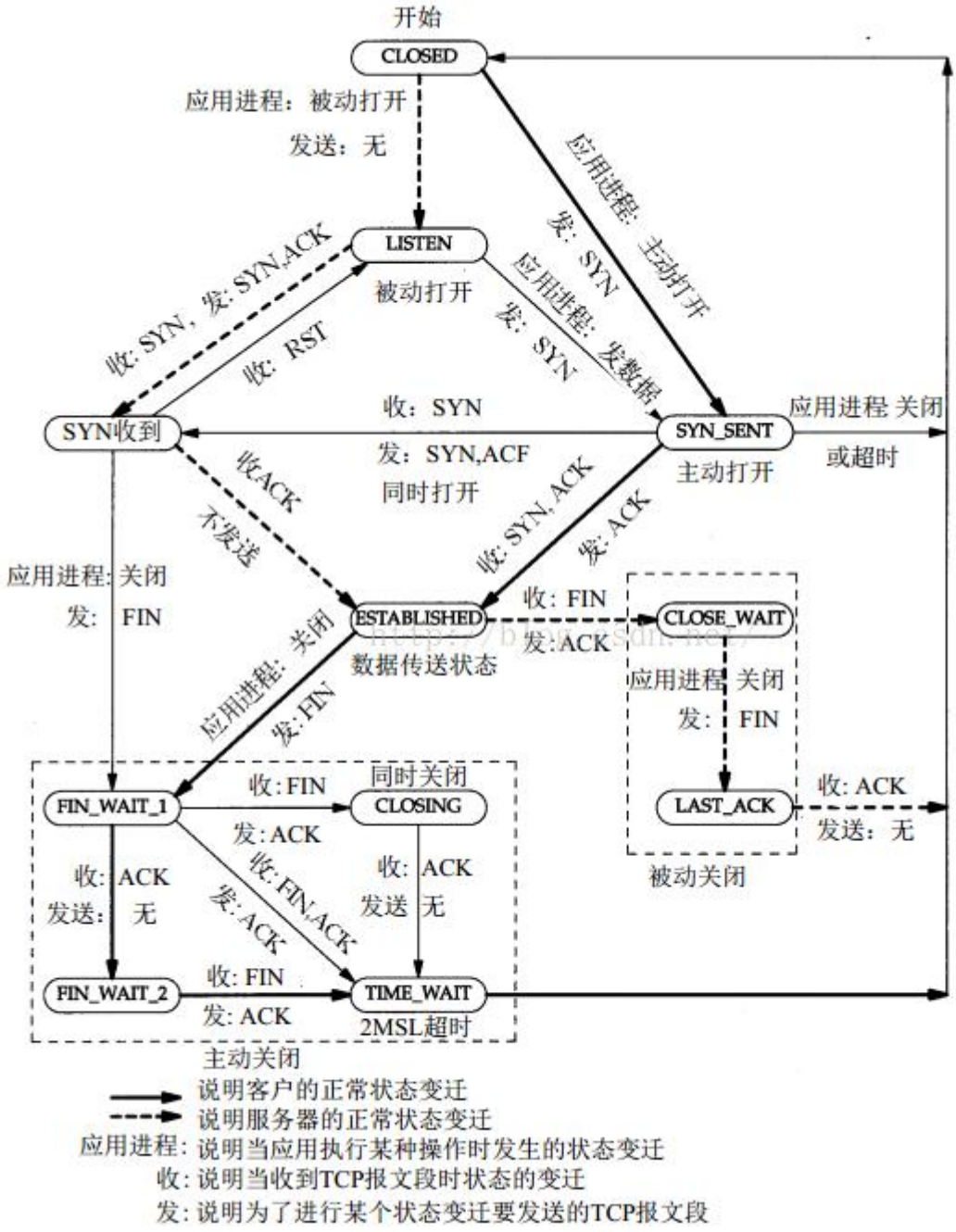
TCP 的状态变迁图

一个连接从开始建立到断开，经历了一连串的状态变化，这次主要分析下它的状态变迁图，首先上经典的状态变迁图



客户端的状态变迁：

CLOSED-->SYN_SENT-->ESTABLISHED-->FIN_WAIT_1-->FIN_WAIT_2-->TIME_WAIT-->CLOSED

服务器的状态变迁：

CLOSED-->LISTEN-->SYN_RCVD-->ESTABLISHED-->CLOSE_WAIT-->LAST_ACK-->CLOSED

CLOSED：这个状态不是一个真正的状态，是图中假想的一个起点或者是终点

LISTEN: 服务器等待连接过来的状态

SYN_SENT: 客户端发起连接（主动打开），变成此状态，如果 SYN 超时，或者服务器不存在直接 CLOSED

SYN_RCVD:服务器收到 SYN 包的时候，就变成此状态，

ESTABLISHED：完成三次握手，进入连接建立状态，说明此时可以进行数据传输了

FIN_WAIT_1:客户端执行主动关闭，发送完 FIN 包之后便进入 FIN_WAIT_1 状态

FIN_WAIT_2:客户端发送 FIN 包之后，收到 ACK，即进入此状态，其实就是半关闭的状态

TIME_WAIT：这个状态从图上看，有 3 中情况，从 FIN_WAIT_2 进入，客户端收到服务器发送过来的 FIN 包之后进入 TIME_WAIT 状态，有 CLOSING 状态进入，这是同时关闭的状态，同时发起 FIN 请求，同时接收并做了 ACK 的回复，从 FIN_WAIT_1 进入，收到对端的 FIN,ACK，并回复 ACK，这个地方感觉是，FIN 和 ACK 是一块来的.

CLOSE_WAIT:接收到 FIN 之后，被动的一方进入此状态，并回复 ACK

LAST_ACK：被动的一端发送 FIN 包之后 处于 LAST_ACK 状态

CLOSING:两边同时发出 FIN 请求

2MSL 等待状态

TIME_WAIT 状态也成为 2MSL 状态，设立这个状态的是因为来保证可靠的实现 TCP 全双工的关闭，MSL 指的是报文最大的生存时间。对于一个具体实现所给定的 MSL 值，处理原则是：当 TCP 执行一个主动关闭，并发回最后一个 ACK，该连接必须在 TIME_WAIT 状态停留的时间为 2MSL。这样可以让 TCP 再次发送最后的 ACK，以防这个 ACK 丢失！

处于 TIME_WAIT 状态的连接，这个 socket 的四要素（Source IP,Source Port，Dest IP,Dest Port）不能被使用，只能在 2MSL 结束后才能再次被使用，其实这个地方，**严格意义上来说是 Port 不能使用**。

某些实现中，可以使用 SO_REUSEADDR 选项，可以让调用者对处于 2MSL 等待的本地端口进行赋值，但是我们将看到 TCP 原则上扔将避免使用仍处于 2MSL 连接中的端口。

对于客户端来讲，主动关闭是处于 2MSL 状态下，但是如果不 bind 对应的 PORT 其实看起来没什么影响的，因为下一次启动客户端，就应该分配到别的 PORT 了，但是对于 bind Port 的客户端或者服务器来讲，肯定是会报告 Address already in use 的错误。

有一个怪异的问题，别以为 SO_REUSEADDR 就万事大吉了，四要素处于 2MSL，将不能被使用，尽管许多具体实现中允许一个进程重新使用仍处于 2MSL 等待的端口，**但是 TCP 不能允许一个新的连接建立在相同的四元素上【有遇到过这样的问题】。**

1. PC1 : Server 启动，监听端口 6666
2. PC2：Client connect to Server port 1098
3. 停止 PC1 服务器进程，这个时候 PC1 的 6666 和 PC2 的 1098 处于 2MSL 状态
4. 在 PC1 上尝试使用 port6666 启动客户端进程，同时与 PC2 端口 1098 进行连接
5. 会有 Address already in use 的错误
6. 使用 SO_REUSEADDR，继续做步骤 5 的动作
7. 仍然有 Address already in use 的错误

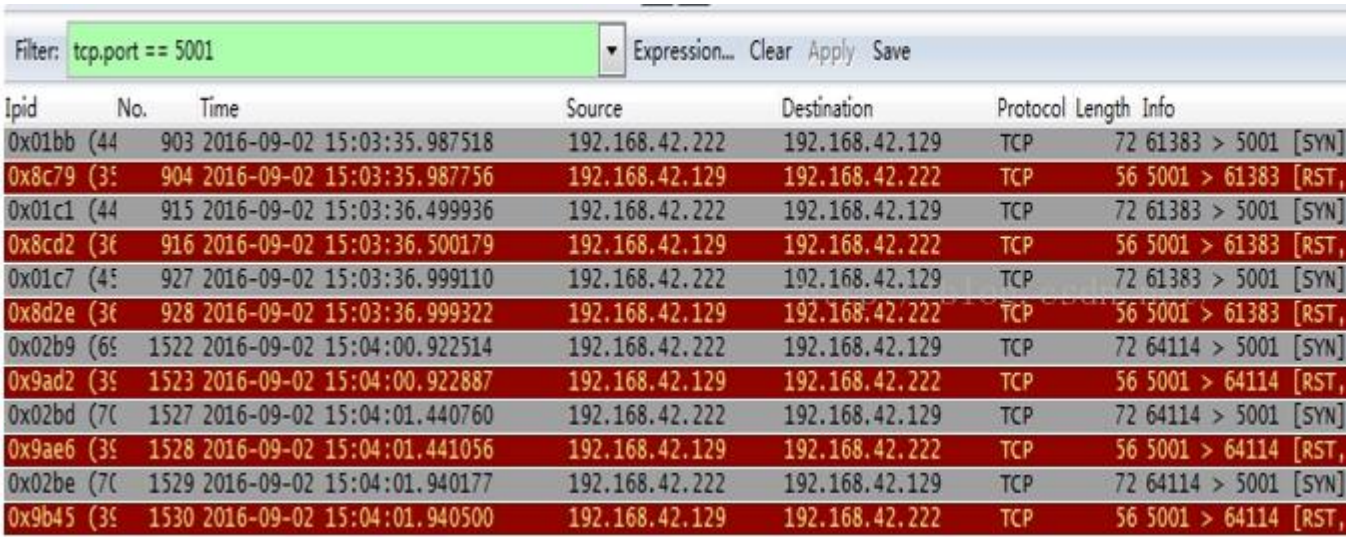
也就是说即使它能将它的本地端口设置为 6666，但它仍然不能和 PC2 的 1098 进行连接，因为顶一个这个连接的四要素处于 2MSL 状态

复位报文段

复位报文其实主要说的就是 RST 标志被置位，无论何时一个报文段发往基准的连接出现错误，TCP 都会发一个复位报文段，基准连接是指由目的 IP 地址和目的端口号以及源 IP 地址和源端口号指明的连接

a.到不存在的端口连接请求

这种状况是比较常见的，这个 port 没有被监听，但是有 client 去 connect，这个时候 TCP 就会使用复位



b.异常终止一个连接

终止连接的正常的方式，就是调用 close 发送 FIN 包，这种的方式可以称为有序释放。但是有一个可能是发送一个复位报文而不是 FIN 来中途释放连接，这种状况就称为异常释放

优点：1.丢弃任何待发送的数据并立即发送复位报文。 2.RST 的接收方会区分另一端执行的是异常关闭还是正常关闭

如何产生异常关闭？ linger no close SO_LINGER 提供了异常关闭的能力。

如果说对端被 RST 了，就会产生一个 Connection reset by peer 的错误！

```
1  0.0                  bsdi.1099 > svr4.8888: S 671112193:671112193(0)
                                                 <mss 1024>
2  0.004975 (0.0050)    svr4.8888 > bsdi.1099: S 3224959489:3224959489(0)
                                                 ack 671112194 <mss 1024>
3  0.006656 (0.0017)    bsdi.1099 > svr4.8888: . ack 1
4  4.833073 (4.8264)    bsdi.1099 > svr4.8888: P 1:14(13) ack 1
5  5.026224 (0.1932)    svr4.8888 > bsdi.1099: . ack 14
6  9.527634 (4.5014)    bsdi.1099 > svr4.8888: R 14:14(0) ack 1
```

c.检测半打开连接，对于半打开的连接，再次进行数据传输的时候，连接已经断开的一端也会通过 RST 包来终止连接

主要介绍：半打开连接，同时打开，同时关闭

TCP 半打开连接

如果一方已经关闭或者异常终止连接而另外一方却还不知道，这样的连接就称为半打开连接（Half open connection）。处于半打开的连接，如果双方不进行数据通信，是发现不了问题的，只有在通信是才真正的察觉到这个连接已经处于半打开状态，如果双方不传输数据的话，仍处于连接状态的一方就不会检测另外一方已经出现异常

半打开连接的一个常见的原因是客户端或者服务器突然掉电而不是正常的结束应用程序后再关机，这样即使重新启动后，原来的连接信息已经消失了，对端仍然保持半打开状态，如果需要发数据的话，这边收到之后 其实发现这个连接并不存在了，就会回复 RST 包告知，这个时候就需要重新建立连接了！

接下来使用 SSH 协议复制一下这个场景：

1. 打开 SSH Secure Shell Client，然后登陆到远程的 **Linux** 服务器 192.168.1.104 DST PORT 22

2. 建立连接成功，可以在 client 进行 linux 操作

3. 需要模拟服务器出现异常，先关掉网卡，然后关机（关掉网卡，为了避免关机时服务器主动退出发送 FIN 包）

4. 这样服务器已经出现异常关闭，但是此时 Client 端并没有任何察觉，这个时候连接已经处于半打开状况

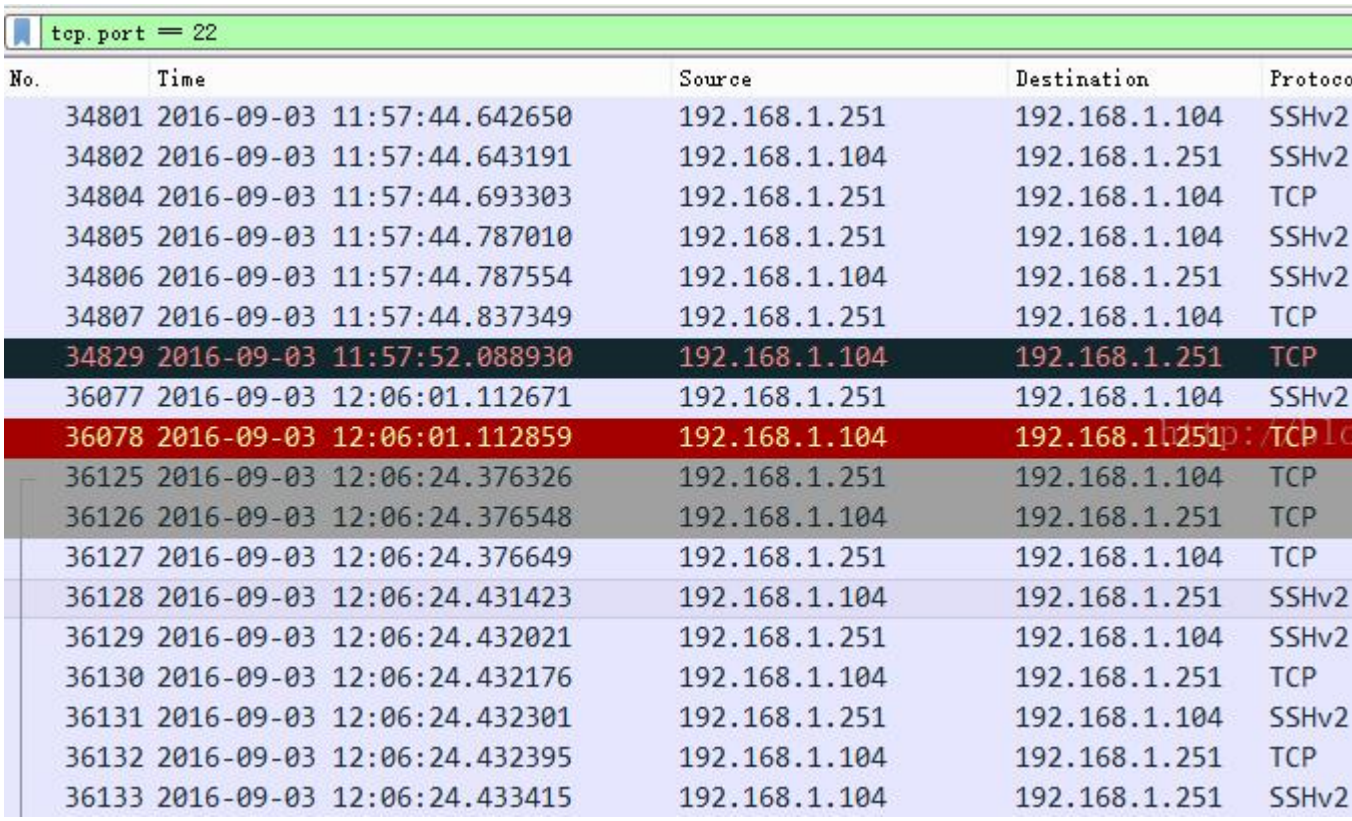
5. 重新打开服务器

6. 操作客户端，发现已经出现了异常，无法通信，并提示重新连接， 这个时候实际上客户端发送的包被服务器给 RST 了，因为之前的连接信息已经丢失了

7. 重新连接之后可以正常的进行通信

服务器重新启动，之前的连接信息都已经丢失，所以它将复位所以信息，因此它不知道数据报文段提到的连接，**处理原则就是接收方以复位做应答**。下面的图就是这个例子的 tcpdump 图示：

从图中可以看到 36077 包是服务器异常之后发的包，但是服务器并没有识别这个连接，发送 36078 RST 包做应答。



同时打开

两个应用程序同时彼此执行主动打开的情况，2 端的端口需要一致，这就需要双方都熟知端口，这种情况发生的概率很小 ，这里简单的介绍一下

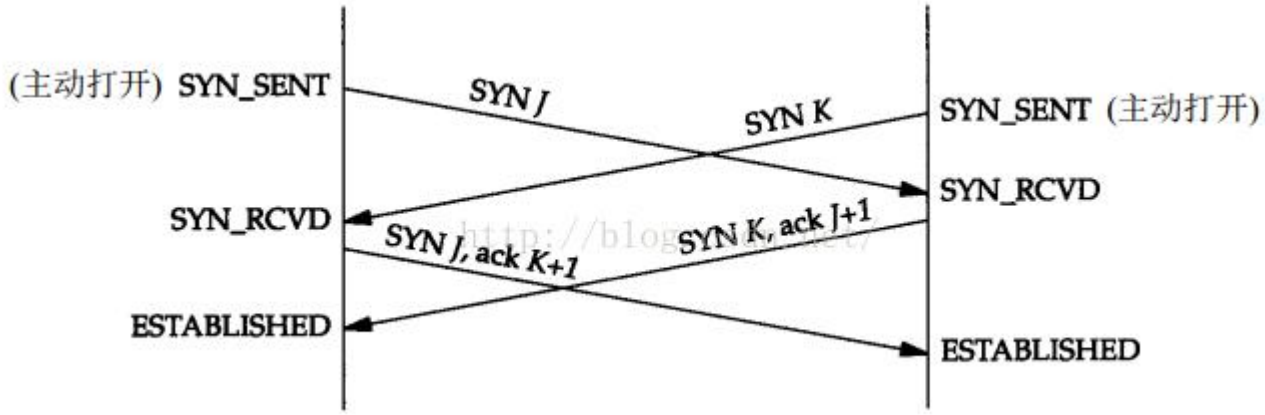
场景：

1. PC1 的应用程序使用端口 7777 与 PC2 的端口 8888 执行主动打开

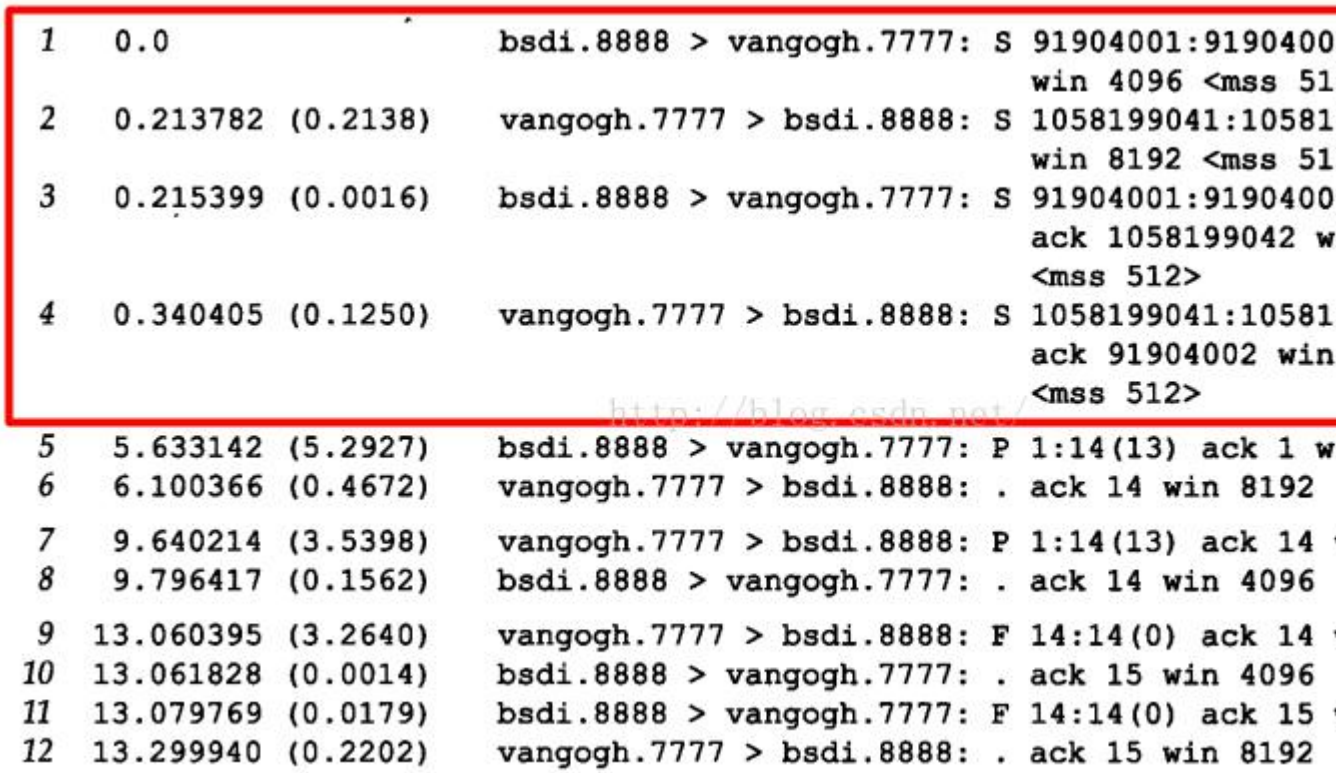
2. PC2 的应用程序使用端口 8888 与 PC1 的端口 7777 执行主动打开

3. SYN 包同时打开对端，这种情况即为同时打开

TCP 中，对于同时打开它仅建立一条连接而不是两条连接，状态变迁图如下：同时发送 SYN 包，然后收到进行确认直接进入 ESTABLISHED 状态，可以看到同时打开需要连接建立需要 4 个报文段，比三次握手多一次！



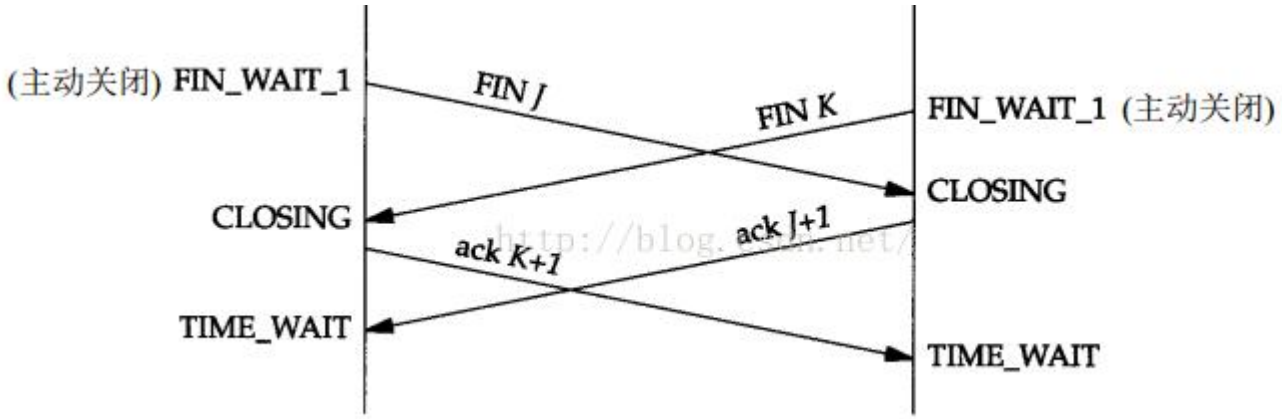
我有尝试制造一下这种场景，使用 tcpdump 来抓一下图示，但是没有复现出来，这个地方就借用教科书中的图来看下



从图中我们可以看到前 4 个报文就是同时打开的 2 个 SYN 和 2 个 ACK。

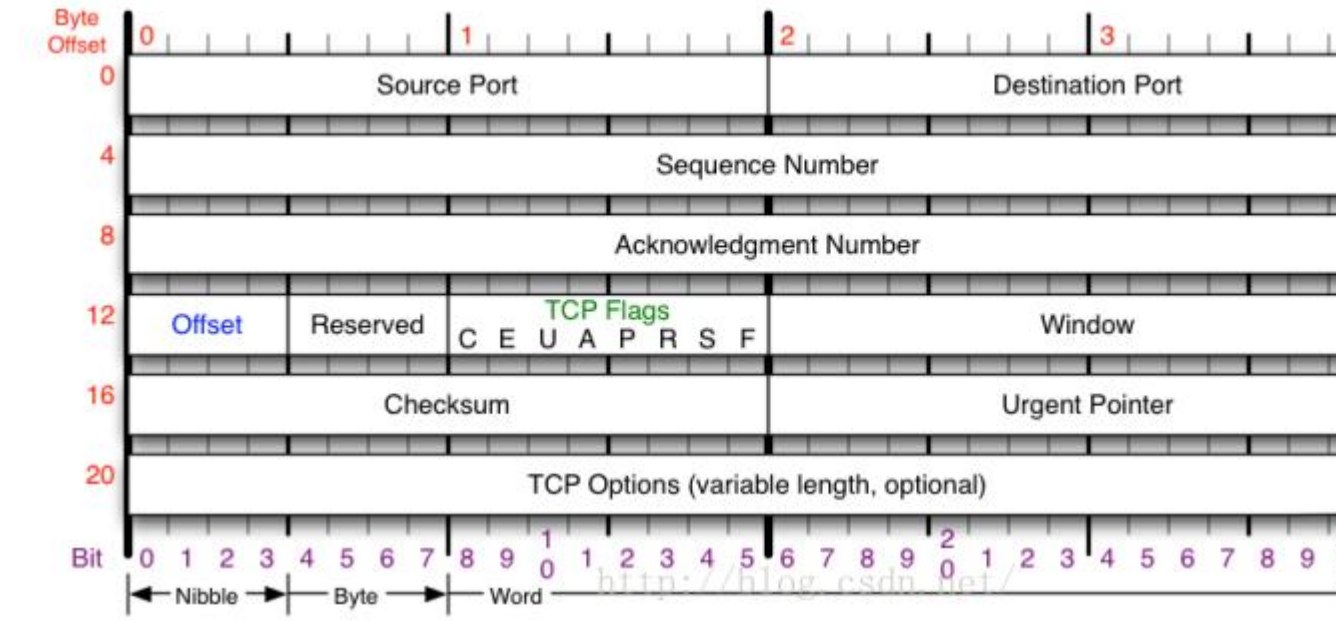
同时关闭

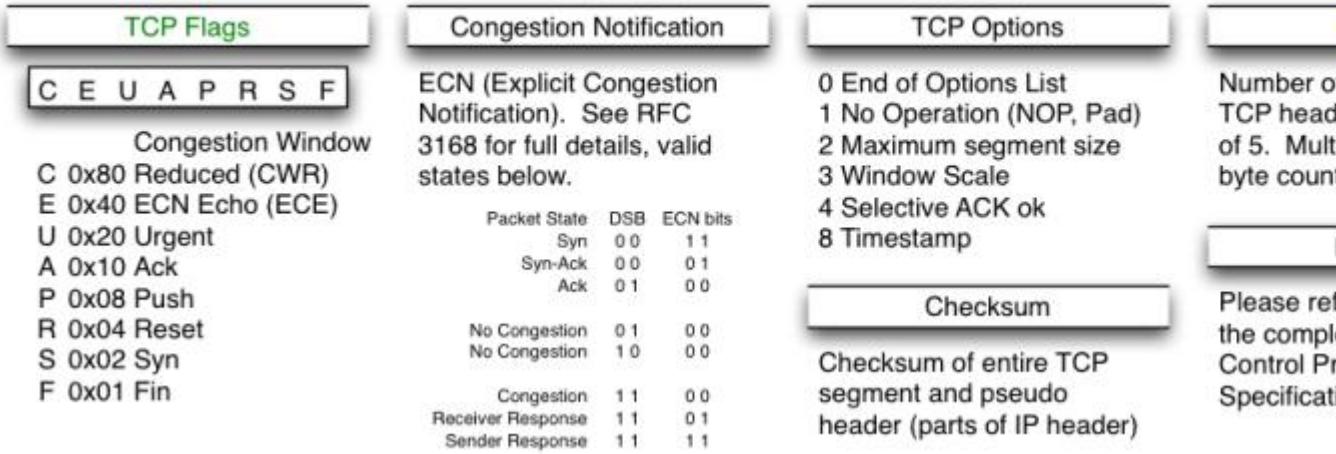
有同时打开，理所应当的也有同时关闭的场景，TCP 协议也允许同时关闭。状态变化可以看到下图了，同时发送 FIN 包，两端同时执行主动关闭，进入 FIN_WAIT_1 的状态，从 FIN_WAIT_1 状态收到 FIN 包的时候进入 CLOSING 状态，然后回复 ACK，进入 TIME_WAIT 状态。



TCP 的头部

TCP 属于协议层的第三次，封包被称为 segment，现在主要来看下 TCP 头部的格式，如下图





一般情况下 TCP Header 的长度为 20 个字节，没有 TCP Options

由上图，需要注意的地方：

1. TCP layer 没有 IP 地址的概念，那个是 IP 层的，所以前 4 个字节是**源端口和目的端口**

2. Sequence Number：传输数据过程中，为每一个封包分配一个序号，**保证网络传输数据的顺序性**

3. Acknowledgment Number：用来确认确实有收到相关封包，内容表示期望收到下一个报文的序列号，**用来解决丢包的问题**

4. TCP Flags：这部分主要标志数据包的属性，比如 SYN，RST，FIN 等，操控 TCP 的状态机

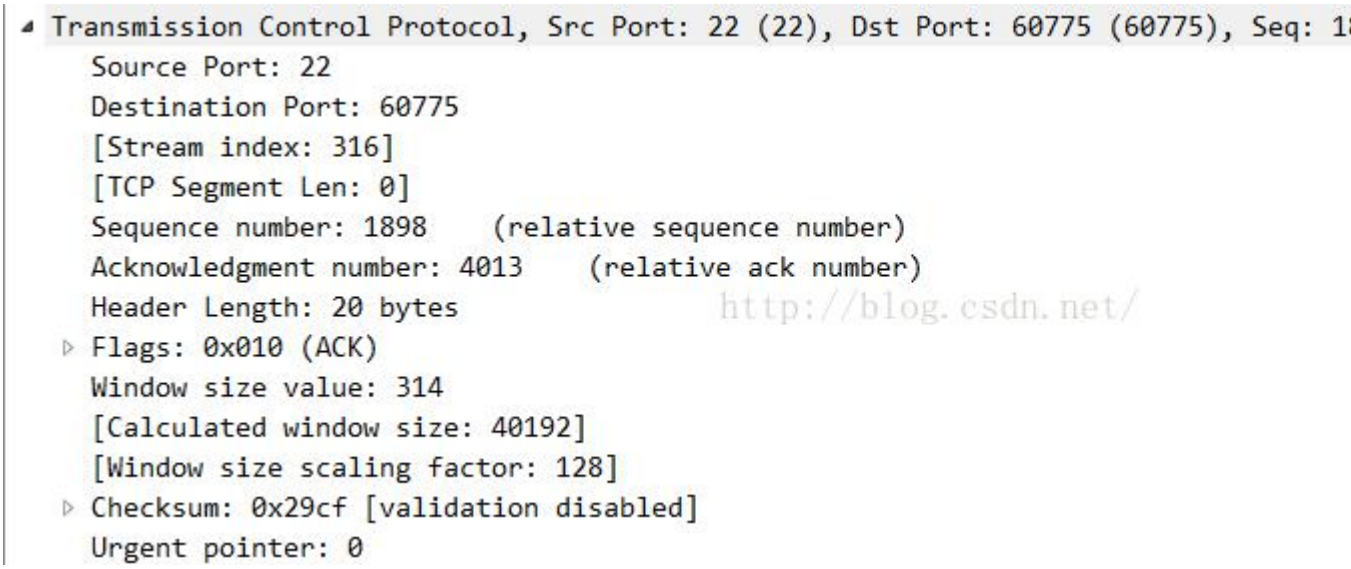
5. Window：滑动窗口，主要用于解决流控拥塞的问题

6. Checksum：校验值

7. Urgent Pointer：紧急指针，可以告知紧急的数据位置，需要和 Flag 的 U flag 配合使用

8. TCP Options：一般包含在三次握手中，有 Option 的选项！

在 tcpdump 抓出来的 TCP Header 如下图



TCP Option

每个选项的开始都是 1 个 Byte 的 kind 字段，说明选项的类型，Kind 为 0/1 的时候，选项只占 1 个 Byte，其他选项在 kind 字段后面还有 len 字节，说明总长度包括 kind 和 len 的字节。看下图

从图中我们可以看到

0. 代表选项表结束

1.代表无操作

2.代表 MSS

3.代表窗口扩大银子

8.代表时间戳

其他的 kind 值为 4/5/6/7 四个选项被称为选择 ACK 及回显选项，目前回显选项已经被时间戳给代替！

下面来看下在 SYN 上显示的 TCP 选项

<mss 512, nop, wscale 0, nop, nop, timestamp 146647 0>

从图中可以看到：

0~3 Byte ： MSS 512

4~7 Byte ： nop，wscale 0

8~11Byte：nop，nop，timestamp

12~15 Byte：146647

16~19 Byte：0 最少 4 个字节对齐

在 tcpdump 抓出的解析是这样的：

<mss 1460, nop, wscale 8, nop, nop, sack>

```
26562 2016-09-03 11:41:44.978270    192.168.1.251    192.168.1.104    TCP
26563 2016-09-03 11:41:44.978447    192.168.1.104    192.168.1.251    TCP
26564 2016-09-03 11:41:44.978557    192.168.1.251    192.168.1.104    TCP
```

```
Urgent pointer: 0
▲ Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scale, No
  ▲ Maximum segment size: 1460 bytes
      Kind: Maximum Segment Size (2)
      Length: 4
      MSS Value: 1460
  ▲ No-Operation (NOP)
    ▷ Type: 1
  ▲ Window scale: 8 (multiply by 256)
      Kind: Window Scale (3)
      Length: 3
      Shift count: 8
      [Multiplier: 256]
  ▲ No-Operation (NOP)
    ▷ Type: 1
  ▲ No-Operation (NOP)
    ▷ Type: 1
  ▲ TCP SACK Permitted Option: True
      Kind: SACK Permitted (4)
      Length: 2
```