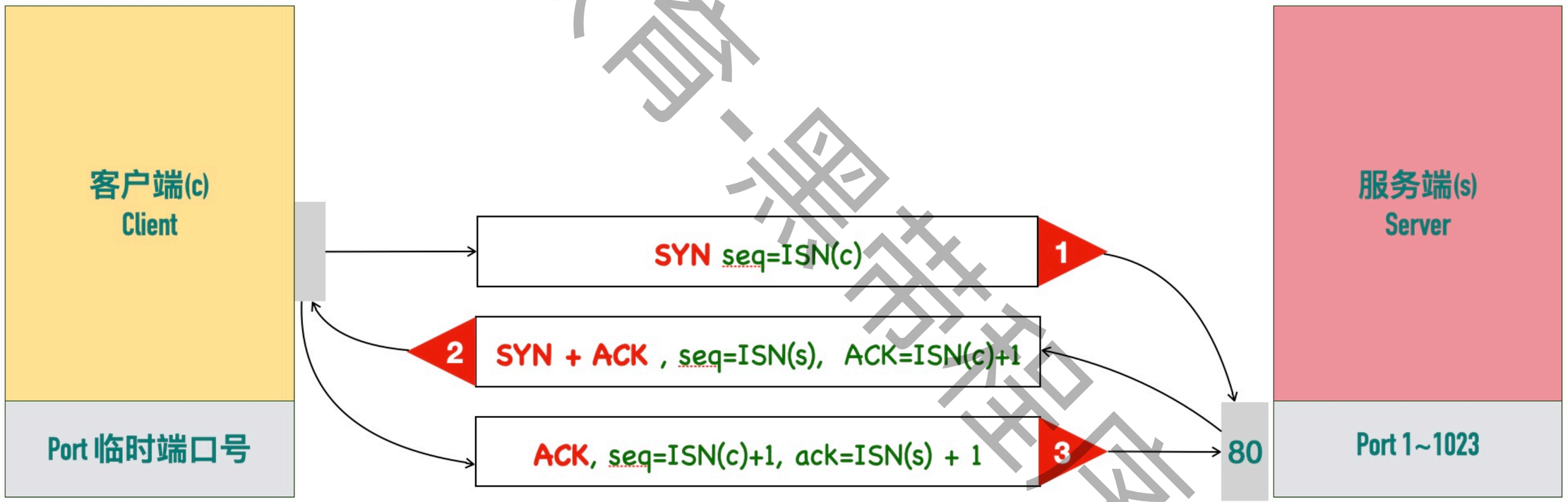
详解三次握手

一次经典的三次握手的过程



三次握手第一步：客户端发送 SYN

注意：**SYN 报文不携带数据，但是它占用一个序号，下次发送数据序列号要加一。**

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th></tr>
<tr><td colspan="2">源端口（Source port）</td><td colspan="2">目标端口（Destination port）</td></tr>
<tr><td colspan="4">序列号（Sequence number）</td></tr>
<tr><td colspan="4">确认号（Acknowledgment number）</td></tr>
<tr><td>头部长度</td><td>保留</td><td>NS</td><td>CWR</td><td>ECE</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td colspan="2">窗口大小（Window Size）</td></tr>
<tr><td colspan="4">校验和（Checksum）</td><td colspan="2">紧急指针（Urgent pointer）</td></tr>
<tr><td colspan="6">选项（Options）、填充（Padding）</td></tr>
</table>

为什么 SYN 段不携带数据却要消耗一个序列号呢?

记住:

- 不占用序列号的段是不需要确认的，比如纯 ACK 包
- SYN 段需要对方的确认，需要占用一个序列号
- 凡是消耗序列号的 TCP 报文段，一定需要对端确认。如果这个段没有收到确认，会一直重传直到达到指定的次数为止。

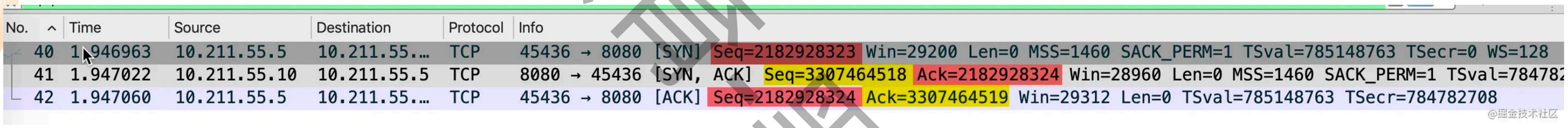
三次握手第二步：服务端回复 SYN + ACK

- 服务端收到客户端的 SYN 段以后，将 SYN 和 ACK 标记都置位
- 「序列号」存放服务端自己的序列号
- 「确认号」字段指定了对端（客户端）下次发送段的序号，这里等于客户端 ISN 加一

0										1	2	3
源端口（Source port）											目标端口（Destination port）	
序列号（Sequence number）												
确认号（Acknowledgment number）												
头部长度	保留	NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	窗口大小（Window Size）	
校验和（Checksum）											紧急指针（Urgent pointer）	
选项（Options）、填充（Padding）												

三次握手第三步：客户端回复 ACK

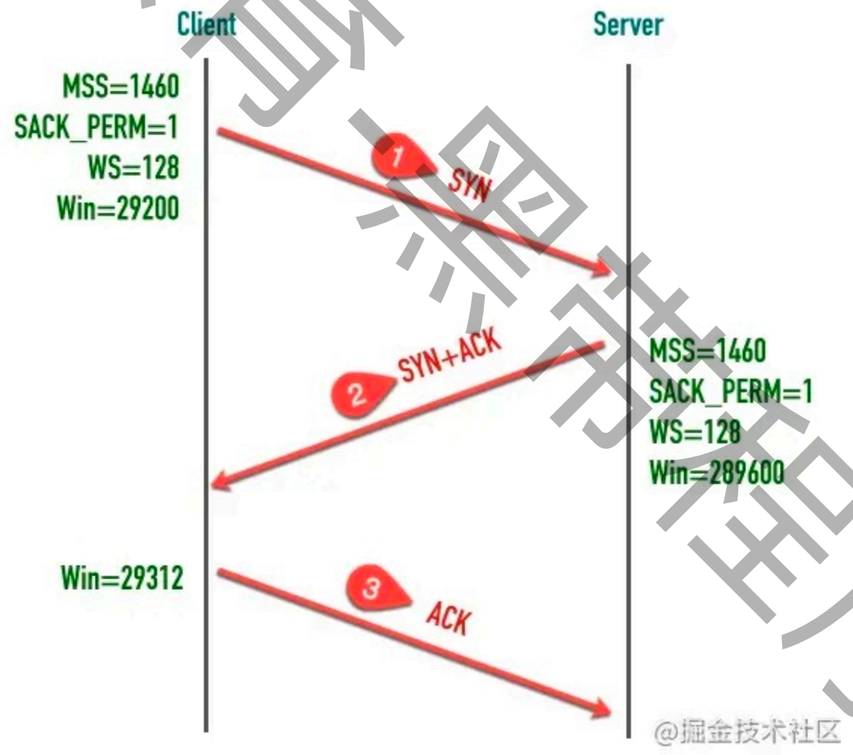
这个 ACK 段用来确认收到了服务端发送的 SYN



No.	Time	Source	Destination	Protocol	Info
40	1.946963	10.211.55.5	10.211.55.…	TCP	45436 → 8080 [SYN] Seq=2182928323 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=785148763 TSecr=0 WS=128
41	1.947022	10.211.55.10	10.211.55.5	TCP	8080 → 45436 [SYN, ACK] Seq=3307464518 Ack=2182928324 Win=28960 Len=0 MSS=1460 SACK_PERM=1 TSval=784782
42	1.947060	10.211.55.5	10.211.55.…	TCP	45436 → 8080 [ACK] Seq=2182928324 Ack=3307464519 Win=29312 Len=0 TSval=785148763 TSecr=784782708

三次握手还交换了什么？

最大段大小（MSS）、窗口大小（Win）、窗口缩放因子（WS)、是否支持选择确认（SACK_PERM）



初始 ISN 生成算法

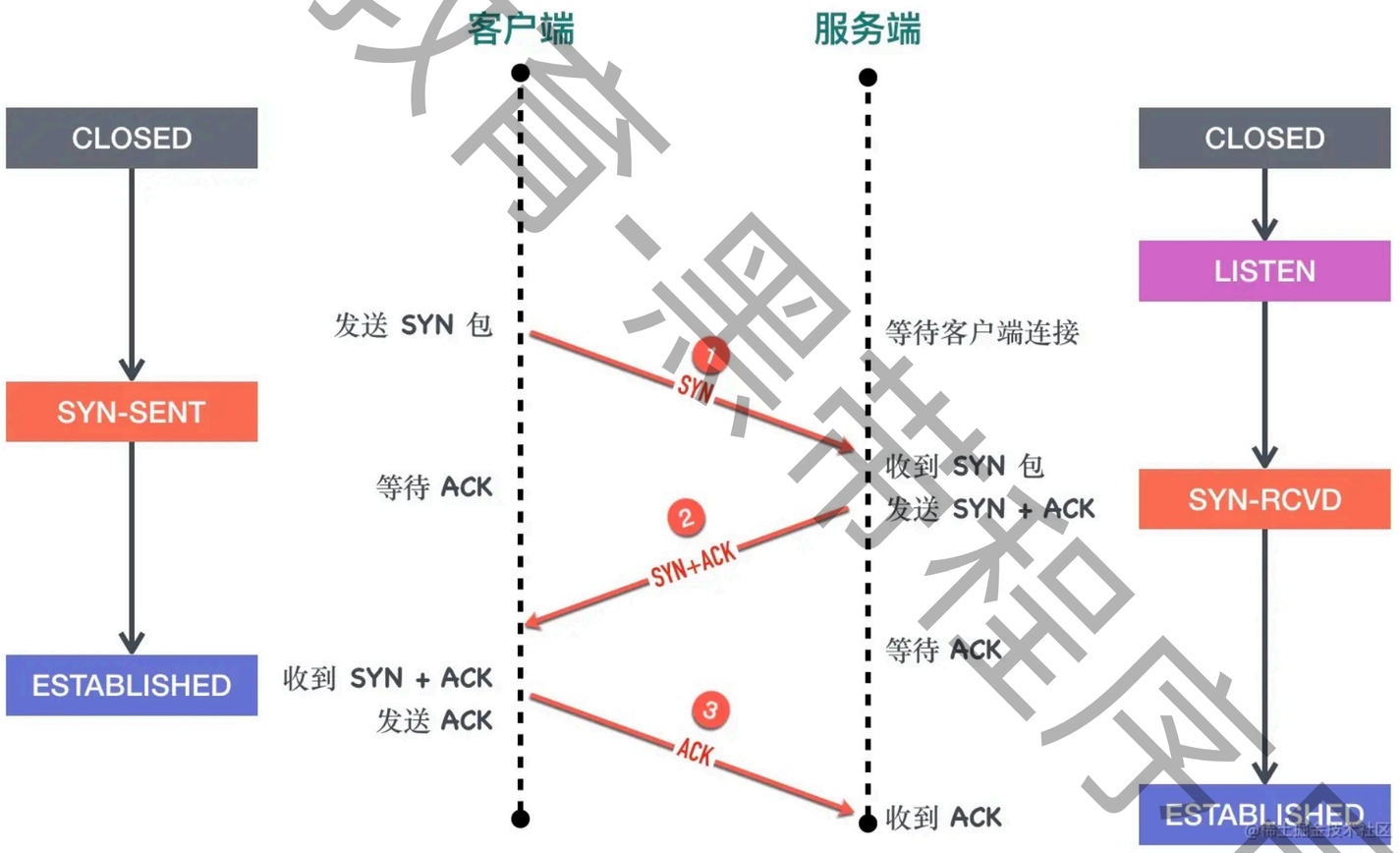
```
__u32 secure_tcp_sequence_number(__be32 saddr, __be32 daddr,
                                 __be16 sport, __be16 dport)
{
    u32 hash[MD5_DIGEST_WORDS];

    net_secret_init();
    hash[0] = (__force u32)saddr;
    hash[1] = (__force u32)daddr;
    hash[2] = ((__force u16)sport << 16) + (__force u16)dport;
    hash[3] = net_secret[15];

    md5_transform(hash, net_secret);

    return seq_scale(hash[0]);
}

static u32 seq_scale(u32 seq)
{
    return seq + (ktime_to_ns(ktime_get_real()) >> 6);
}
```

ISN 能设置成一个固定值呢?

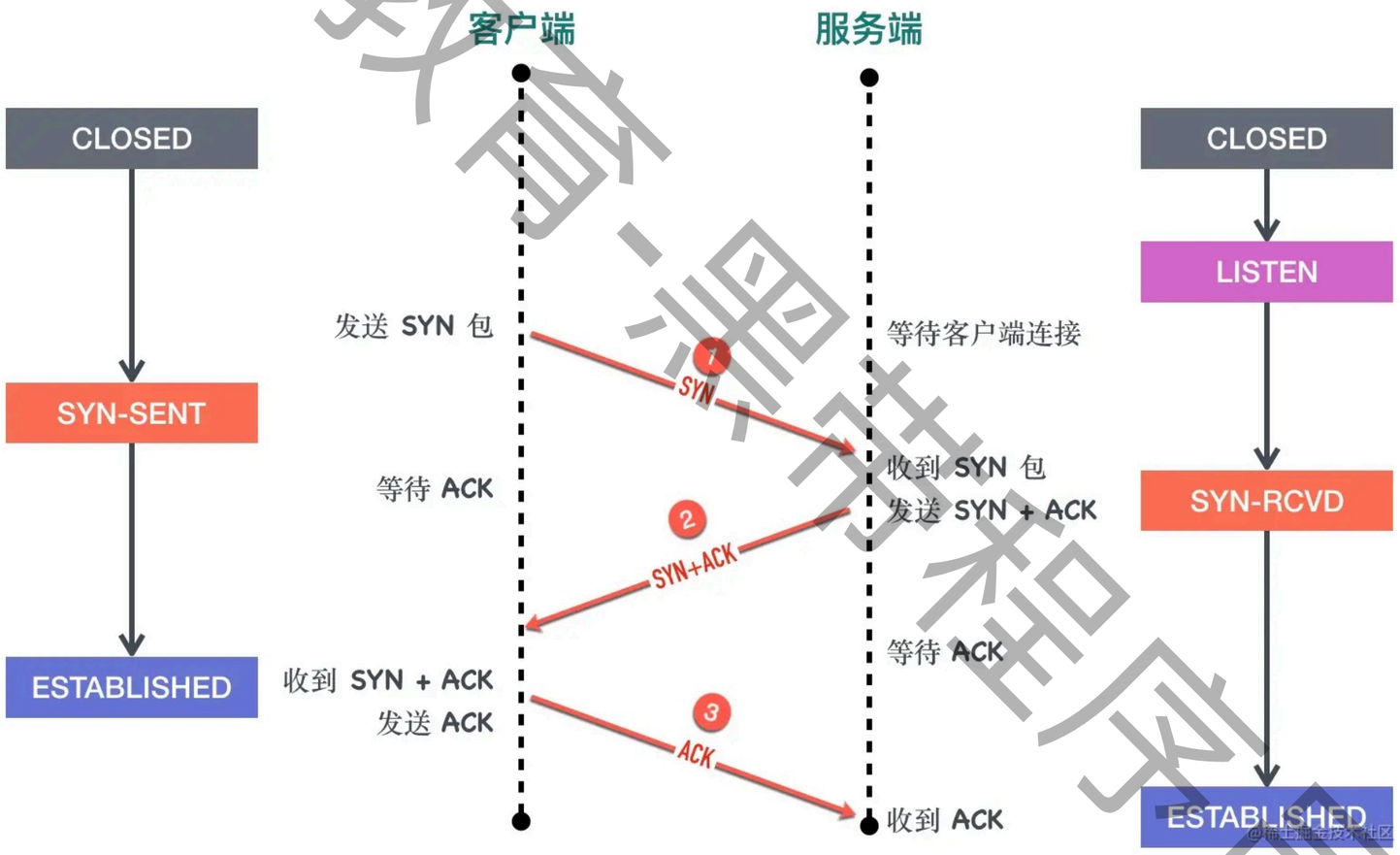
- 安全性
- 开启 SO_REUSEADDR 端口重用

三次握手中的状态变迁

三次握手中的状态变迁



如果客户端发出去的 SYN 包，服务端一直没有回 ACK 会发生什么?



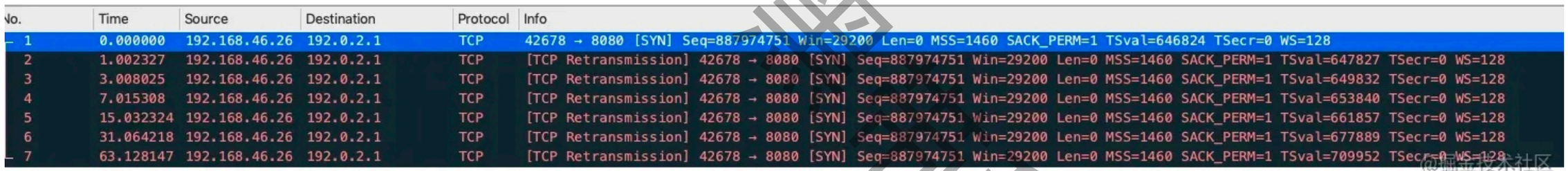
packetdrill 脚本

```
// 新建一个 server socket
+0    socket(..., SOCK_STREAM, IPPROTO_TCP) = 3

// 客户端 connect
+0 connect(3, ..., ...) = -1
```

客户端重传 SYN 的次数由什么决定?

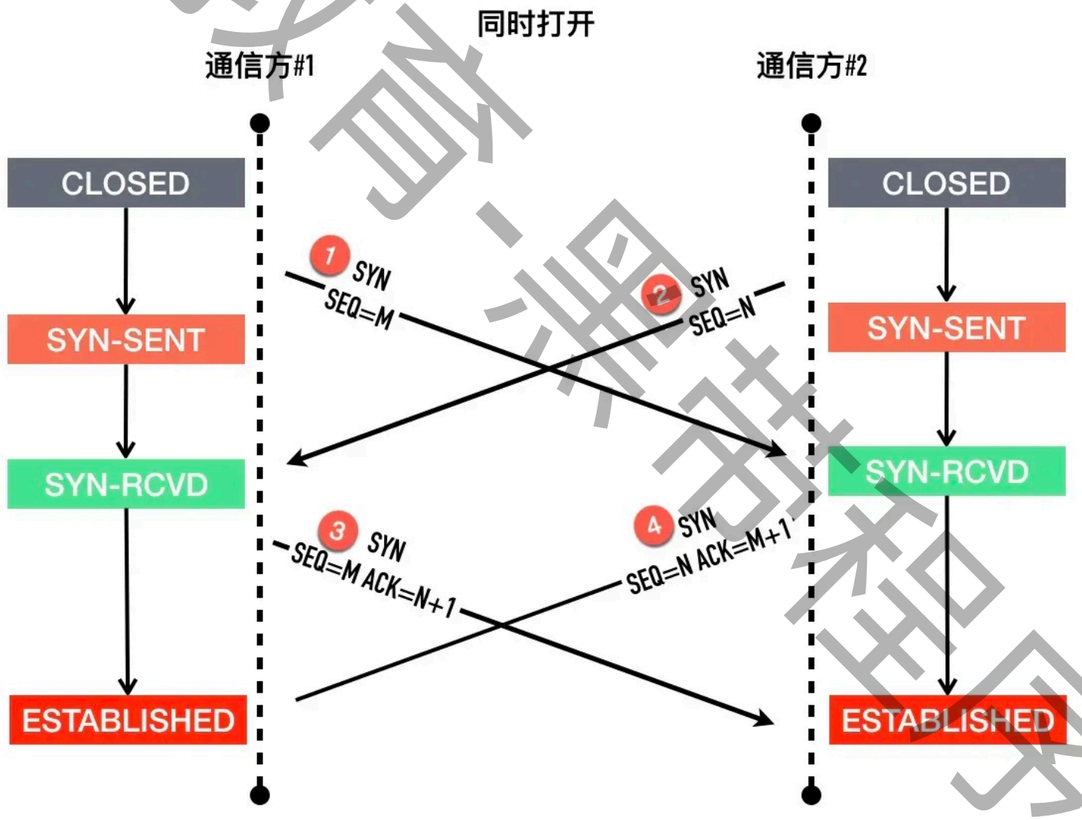
重传总时间：63s = 1s+2s+4s+8s+16s+32s

```
$ sysctl -a | grep tcp_syn_retries
net.ipv4.tcp_syn_retries = 6
```



No.	Time	Source	Destination	Protocol	Info
1	0.000000	192.168.46.26	192.0.2.1	TCP	42678 → 8080 [SYN] Seq=887974751 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=646824 TSecr=0 WS=128
2	1.002327	192.168.46.26	192.0.2.1	TCP	[TCP Retransmission] 42678 → 8080 [SYN] Seq=887974751 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=647827 TSecr=0 WS=128
3	3.008025	192.168.46.26	192.0.2.1	TCP	[TCP Retransmission] 42678 → 8080 [SYN] Seq=887974751 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=649832 TSecr=0 WS=128
4	7.015308	192.168.46.26	192.0.2.1	TCP	[TCP Retransmission] 42678 → 8080 [SYN] Seq=887974751 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=653840 TSecr=0 WS=128
5	15.032324	192.168.46.26	192.0.2.1	TCP	[TCP Retransmission] 42678 → 8080 [SYN] Seq=887974751 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=661857 TSecr=0 WS=128
6	31.064218	192.168.46.26	192.0.2.1	TCP	[TCP Retransmission] 42678 → 8080 [SYN] Seq=887974751 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=677889 TSecr=0 WS=128
7	63.128147	192.168.46.26	192.0.2.1	TCP	[TCP Retransmission] 42678 → 8080 [SYN] Seq=887974751 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=709952 TSecr=0 WS=128

冷知识：同时打开



TCP 自连接：一个看似 Bug 的现象

源端口号和目标端口号都是自己可能吗?

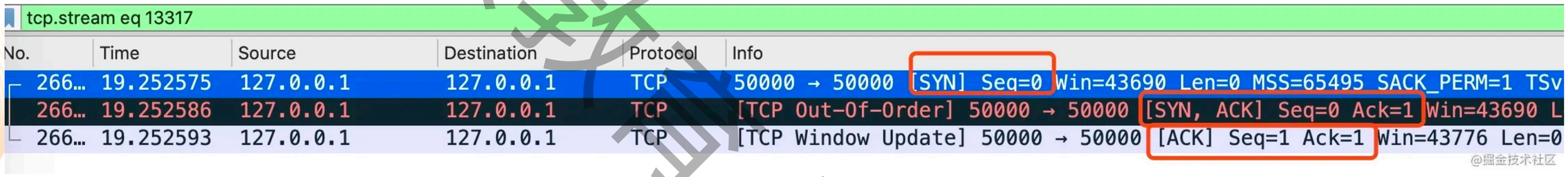
测试脚本 self_connect.sh

```
while true
do
    telnet 127.0.0.1 50000
done
```

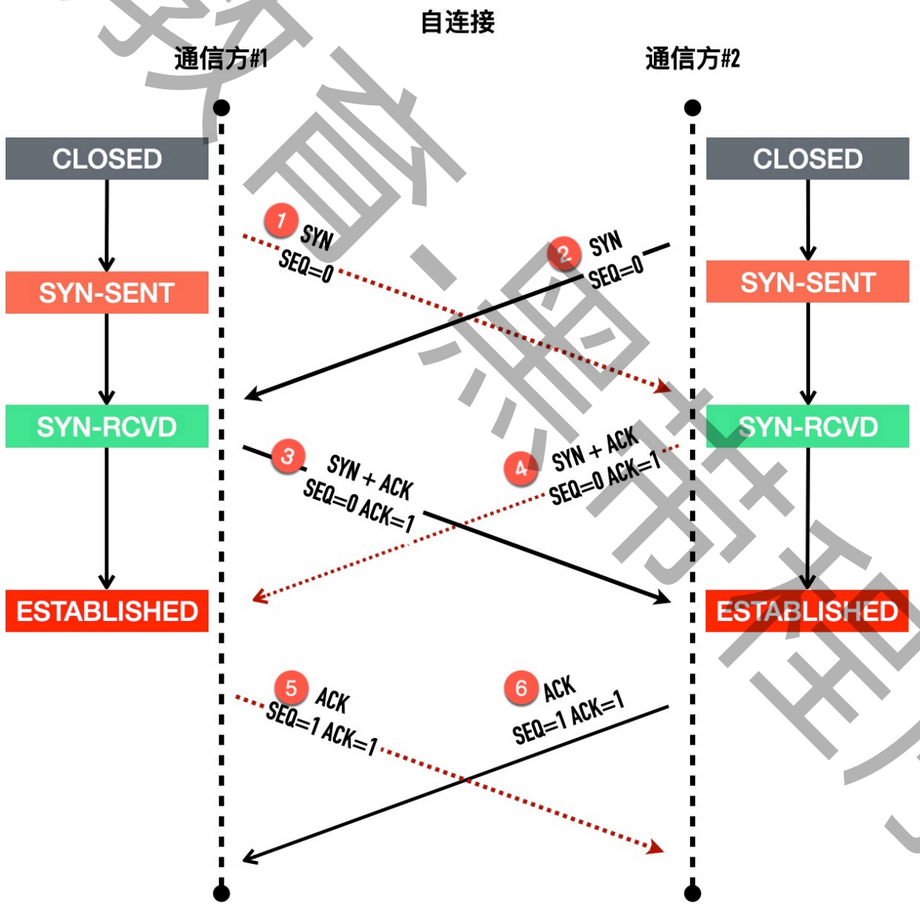
有图有真相

```
telnet: Unable to connect to remote host: Connection refused
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.

~ (ssh)
VM-8-4-ubuntu :: ~ » ss -tnpa | head -1; ss -tnpa | grep :50000
State  Recv-Q Send-Q                  Local Address:Port     Peer Address:Port  Process
ESTAB  0      0                        127.0.0.1:50000       127.0.0.1:50000  users:(("telnet",pi
VM-8-4-ubuntu :: ~ »
```

抓包现场



自连接包交互过程



自连接的危害

你写的业务系统 B 会访问本机服务 A，服务 A 监听了 50000 端口
1. 业务系统 B 的代码写的稍微比较健壮，增加了对服务 A 断开重连的逻辑
2. 如果有一天服务 A 挂掉比较长时间没有启动，业务系统 B 开始不断 connect 重连
3. 系统 B 经过一段时间的重试就会出现自连接的情况
4. 这时服务 A 想启动监听 50000 端口就会出现地址被占用的异常，无法正常启动

自连接的进程占用了端口，导致真正需要监听端口的服务进程无法监听成功
自连接的进程看起来 connect 成功，实际上服务是不正常的，无法正常进行数据通信

如何解决自连接问题

- 让服务监听的端口与客户端随机分配的端口不可能相同即可
- 当出现自连接的时候，主动关掉连接

让服务监听的端口与客户端临时端口号不可能相同即可

客户端临时端口号的范围由 /proc/sys/net/ipv4/ip*oca*lport_range 文件决定，只要服务端监控的端口号不在这个范围内就可以了。

当出现自连接的时候，主动关掉连接

```
func (sd *sysDialer) doDialTCP(ctx context.Context, laddr, raddr *TCPAddr) (*TCPConn, error) {
    fd, err := internetSocket(ctx, sd.network, laddr, raddr, syscall.SOCK_STREAM, 0, "dial", sd.Dialer.Control)
    for i := 0; i < 2 && (laddr == nil || laddr.Port == 0) && (selfConnect(fd, err) || spuriousENOTAVAIL(err)); i++ {
        if err == nil {
            fd.Close()
        }
        return newTCPConn(fd), nil
}
func selfConnect(fd *netFD, err error) bool {
    // If the connect failed, we clearly didn't connect to ourselves.
    if fd.laddr == nil || fd.raddr == nil {
        return true
    }
    l := fd.laddr.(*TCPAddr)
    r := fd.raddr.(*TCPAddr)
    return l.Port == r.Port && l.IP.Equal(r.IP)
}
```

我对 Ktor 库的自连接的修复经历

https://juejin.cn/post/6926375458846015495/