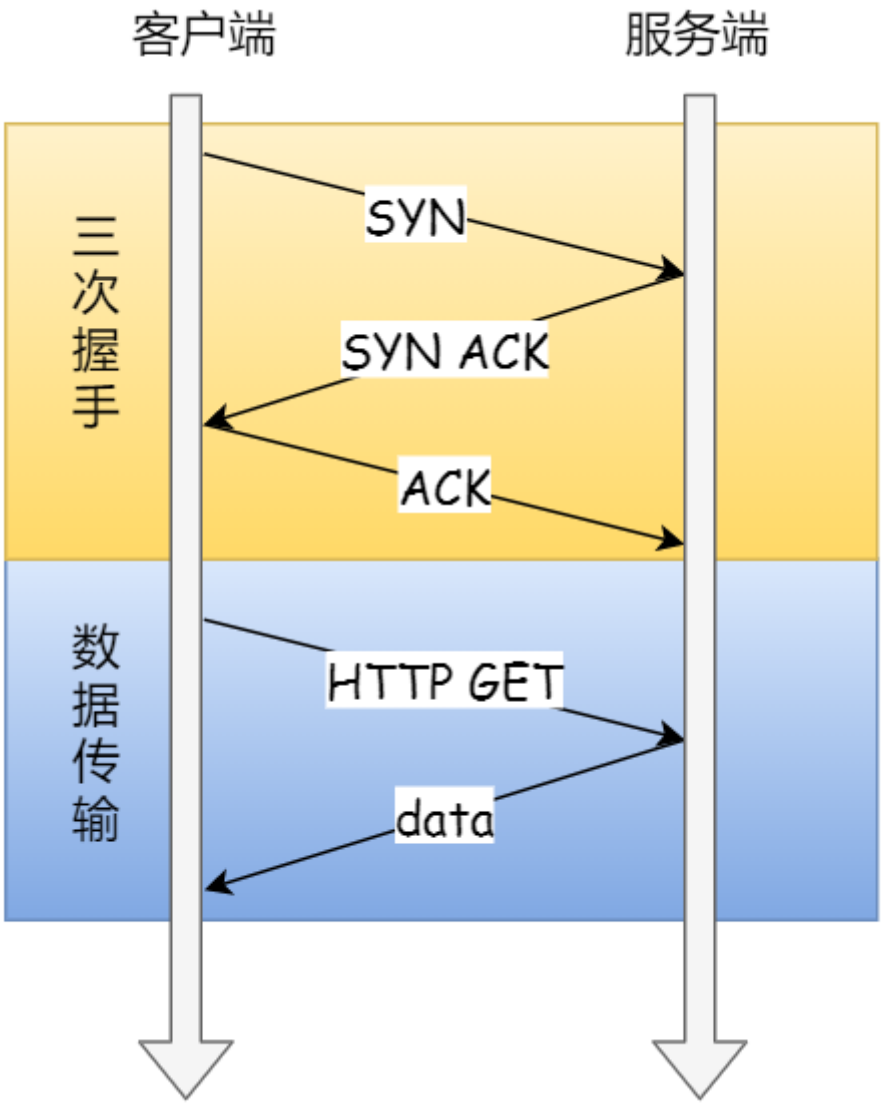
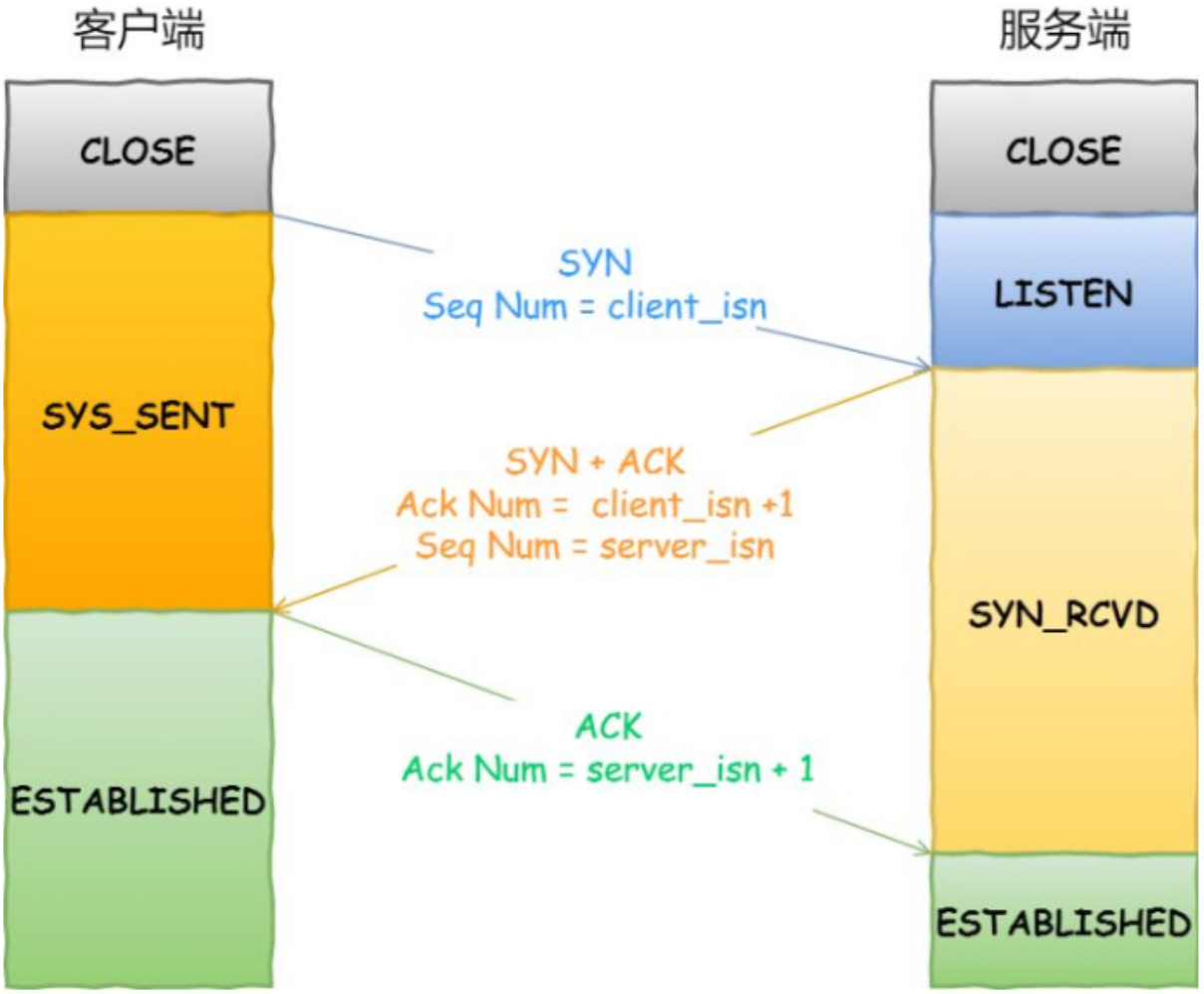
TCP三次握手的性能优化

1. TCP三次握手

TCP是面向连接、可靠的、双向传输的传输层协议，传输数据前需要经过三次握手建立连接。



三次握手的过程在一个http请求的平均时间占比10%以上，在网络不佳、高并发或者遭遇SYN攻击等场景中，如果不能有效正确的调节三次握手中的参数，就会对性能产生很多影响。 出现问题时，先用netstat命令查看是哪个握手环节出现了问题，再来对症下药。

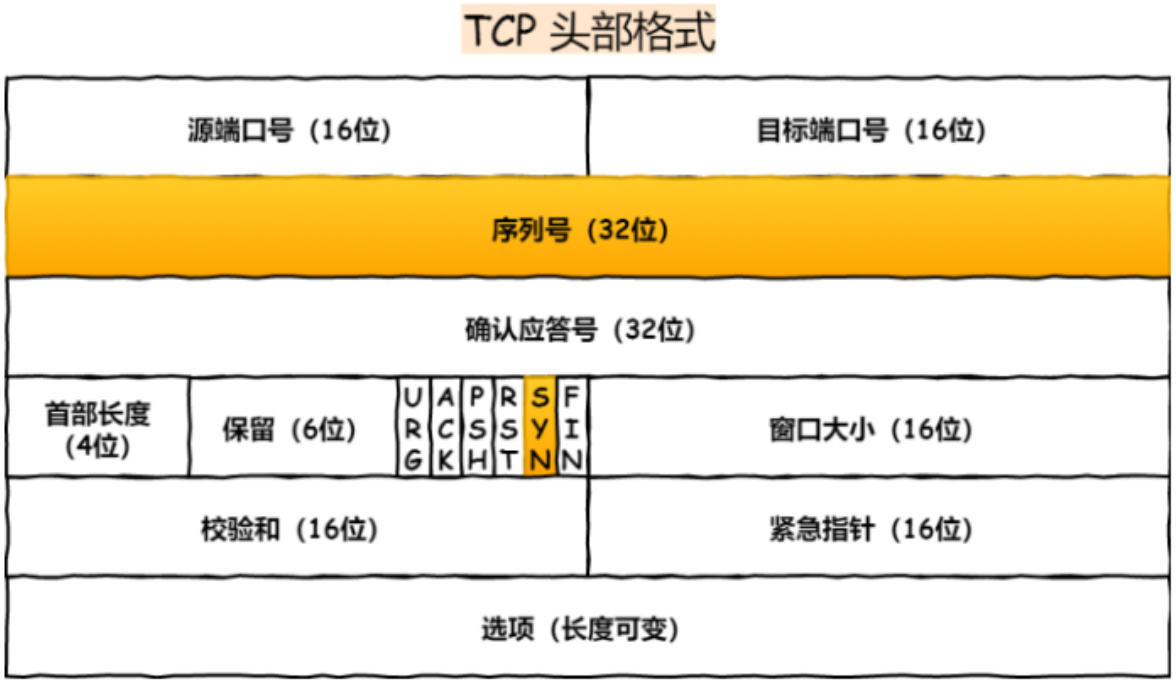


- SYN：Synchronize Sequence Numbers（同步序列号）
- ACK：用来确认收到

2.客户端优化

三次握手建立连接的首要目的是**同步序列号。**只有同步了序列号才有可靠传输，TCP许多特性都依赖于序列号实现，比如流量控制、丢包重传等，这也是三次握手中的报文称为SYN的原因。如下图所

示，TCP的头部格式。



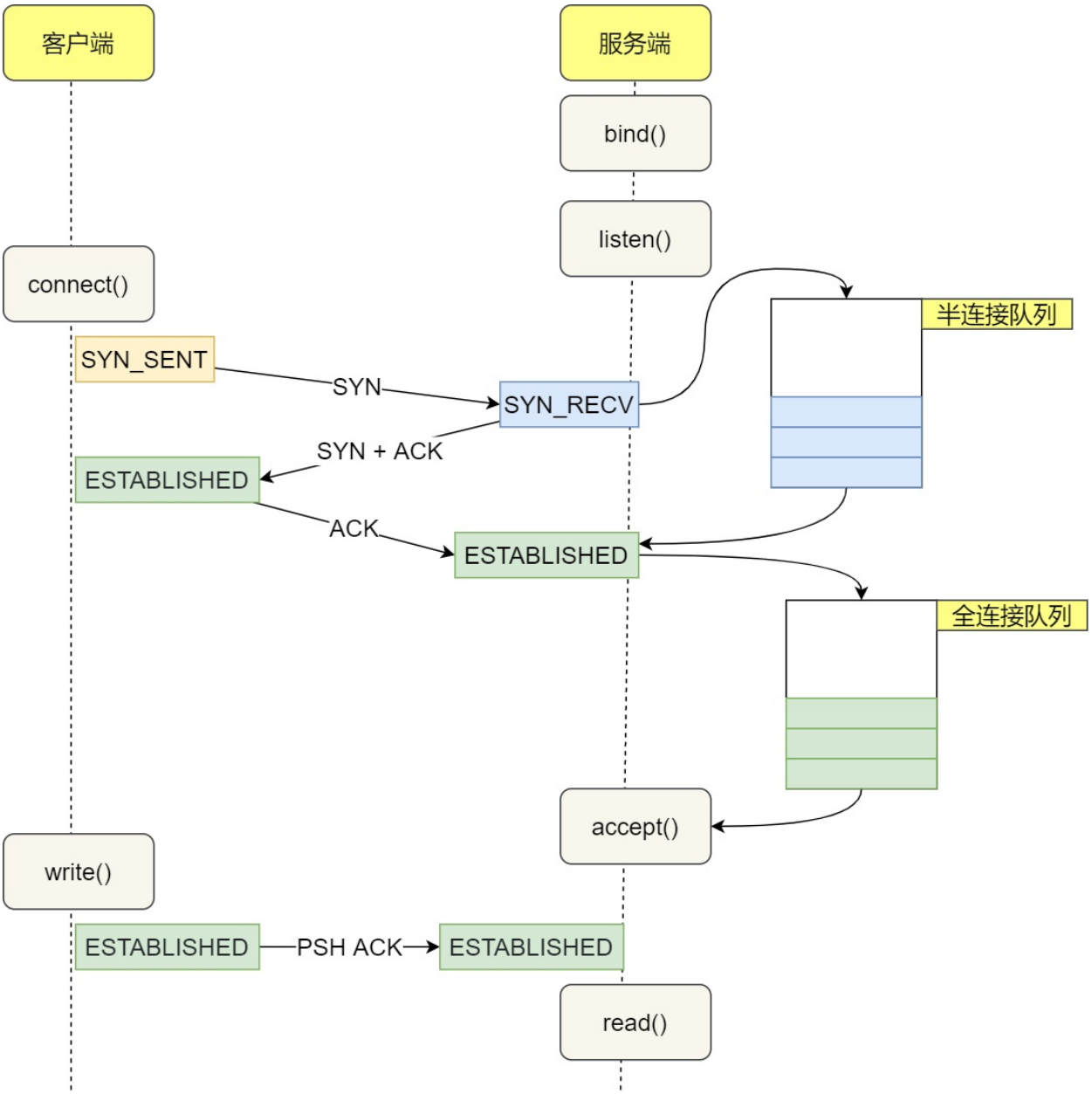
客户端作为主动发起连接方，首先它将发送SYN包，于是客户端的连接就会处于SYN_SENT状态。 此时，客户端等待服务端回复的ACK报文，正常情况下，服务器会在几毫秒内返回SYN+ACK，但如果客户端长时间没有收到SYN+ACK报文，则会重发SYN包，重发的次数由**tcp_syn_retries**参数控制，默认是6次。 通常，第一次超时重传是1秒后，第二次重传是2秒后，第三次重传是4秒后，每次重传时间间隔为上次的2倍。所以总共的耗时为1+2+4+8+16+32=63秒，大概一分钟后服务端仍然没有ACK回应，那么客户端会终止三次握手。

所以优化的思路就是根据网络的稳定性和目标服务器的繁忙程度修改 SYN 的重传次数，调整客户端的三次握手时间上限。比如内网中通讯时，就可以适当调低重试次数，尽快把错误暴露给应用程序。

3. 服务端优化

3.1 半连接队列优化

当服务端收到SYN包后，服务端会立马回复SYN+ACK包，表示确认收到了客户端的序列号，同时也把自己的序列号发给对方。 此时，服务端出现了新连接，状态是SYN_RCV。在这个状态下，linux内核会建立一个半连接队列来维护未完成的握手信息。当半连接队列溢出后，服务端就无法建立新的连接。



SYN攻击，攻击的就是这个半连接队列。可以通过如下指令查看：

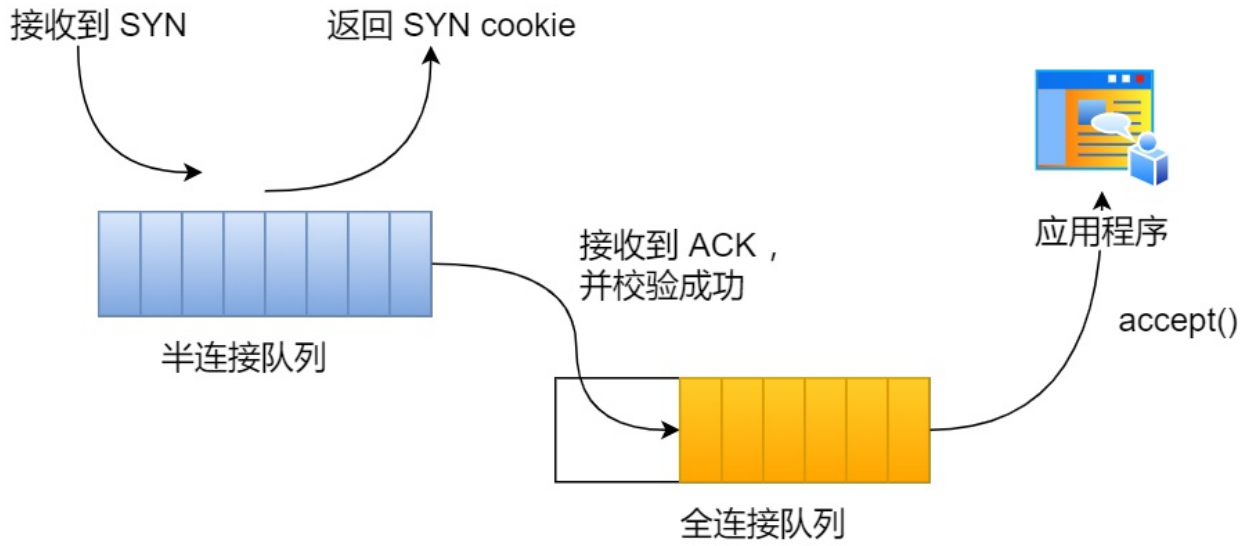
```
netstat -s | grep "SYNs to LISTEN"
2099 SYNs to LISTEN sockets dropped
```

上面输出的是累加值，表示共有多少个TCP连接因为半连接队列溢出而被丢弃。隔几秒执行几次，如果有上升的趋势，说明当前存在半连接队列溢出的现象。

要想增大半连接队列需要增大**tcp_max_syn_backlog、somaxconn和backlog**，前两个参数通过修改内核参数，增大backlog的方式，每个web服务器都不同。

3.2 开启syncookies

增大半连接队列的方式始终是指标不治本，开启syncookies功能可以在不使用SYN半连接队列的情况下成功建立连接。syncookies的工作原理是服务器根据当前状态计算出一个值，放在己方发出的SYN+ACK报文中，当客户端返回ACK报文时，取出该值验证，如果合法则认为连接成功建立。



syncookies主要有以下三个值：

- 0：表示关闭该功能
- 1：表示当SYN半连接队列放不下时启用
- 2：表示无条件开启

应对SYN攻击，只需要设置为1即可。

3.3 SYN_RCV状态的优化

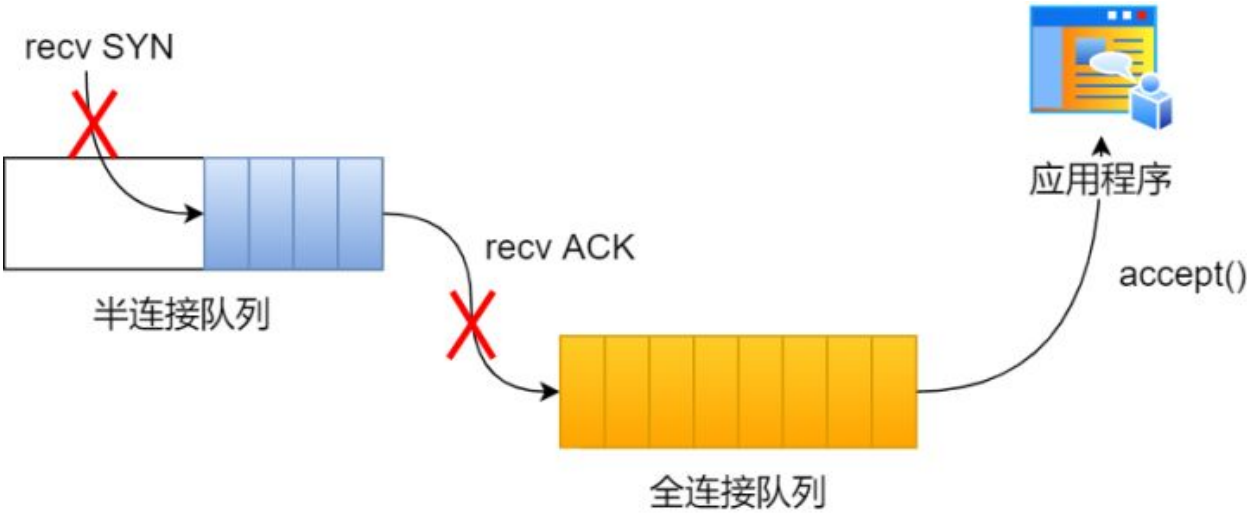
当客户端收到服务器发来的SYN+ACK报文后，就会回复ACK给服务器，同时客户端连接状态从SYN_SENT转换为ESTABLISHED，表示连接建立成功。 服务端收到客户端的ACK报文后，服务端的状态会从SYN_RCV状态转换为ESTABLISHED，表示连接成功建立。

如果服务器没有收到ACK，就会重发SYN+ACK报文。当网络繁忙、不稳定时，报文丢失就会严重。此时应该调大重发次数。反之则需要调小重发次数。修改重发次数的方法时修改内核参数**tcp_synack_retries。**

3.5 accept队列已满的优化

服务器收到ACK后连接成功建立，此时，linux内核会把连接从半连接队列移除，然后创建新的完全连接，将其添加到accept队列，等待进程调用accept函数把连接取出来。

如果进程不能及时的调用accept函数，就会造成accept队列（也称全连接队列）溢出，最终导致建立好的连接被丢弃。



丢弃连接是linux的默认行为，可以选择向客户端发送RST复位报文，告诉客户端连接建立失败。将linux内核参数**tcp_abort_on_overflow**设置为1即可打开此功能。

tcp_abort_on_overflow共有两个值：

- 0：如果accept队列满了，server扔掉client发过来的ack
- 1：如果accept队列满了，server发送一个RST包给client，表示废掉这个握手过程和连接。

如果在客户端看到很多connection reset by peer就可以证明是由于服务端TCP全连接队列溢出。

3.6 调整accept队列大小

accept队列的长度取决于 somaxconn 和 backlog 之间的最小值，其中：

- somaxconn是linux的内核参数，默认值是128.
- backlog是listen(int sockfd, int backlog) 函数中的 backlog 大小

tomcat、nginx、apache常见的web服务器backlog是511. 通过如下命令可以看accept队列的长度

```
ss -ltn
```

如下图所示

```
[huan@VM_0_5_centos ~]$ ss -lnt
State      Recv-Q Send-Q                 Local Address:Port                                Peer Address:Port
LISTEN     0      128                        127.0.0.1:3306                                       *:*
LISTEN     0      128                              *:22                                           *:*
LISTEN     0      128                           [::]:443                                        [::]:*
LISTEN     0      70                            [::]:33060                                      [::]:*
LISTEN     0      128                           [::]:80                                         [::]:*
```

- Recv-Q:当前accept队列的大小，也就是已经完成三次握手并等待服务端accept()的TCP连接
- Send-Q：accept队列的最大长度

可以使用如下命令查看丢掉的tcp连接个数。

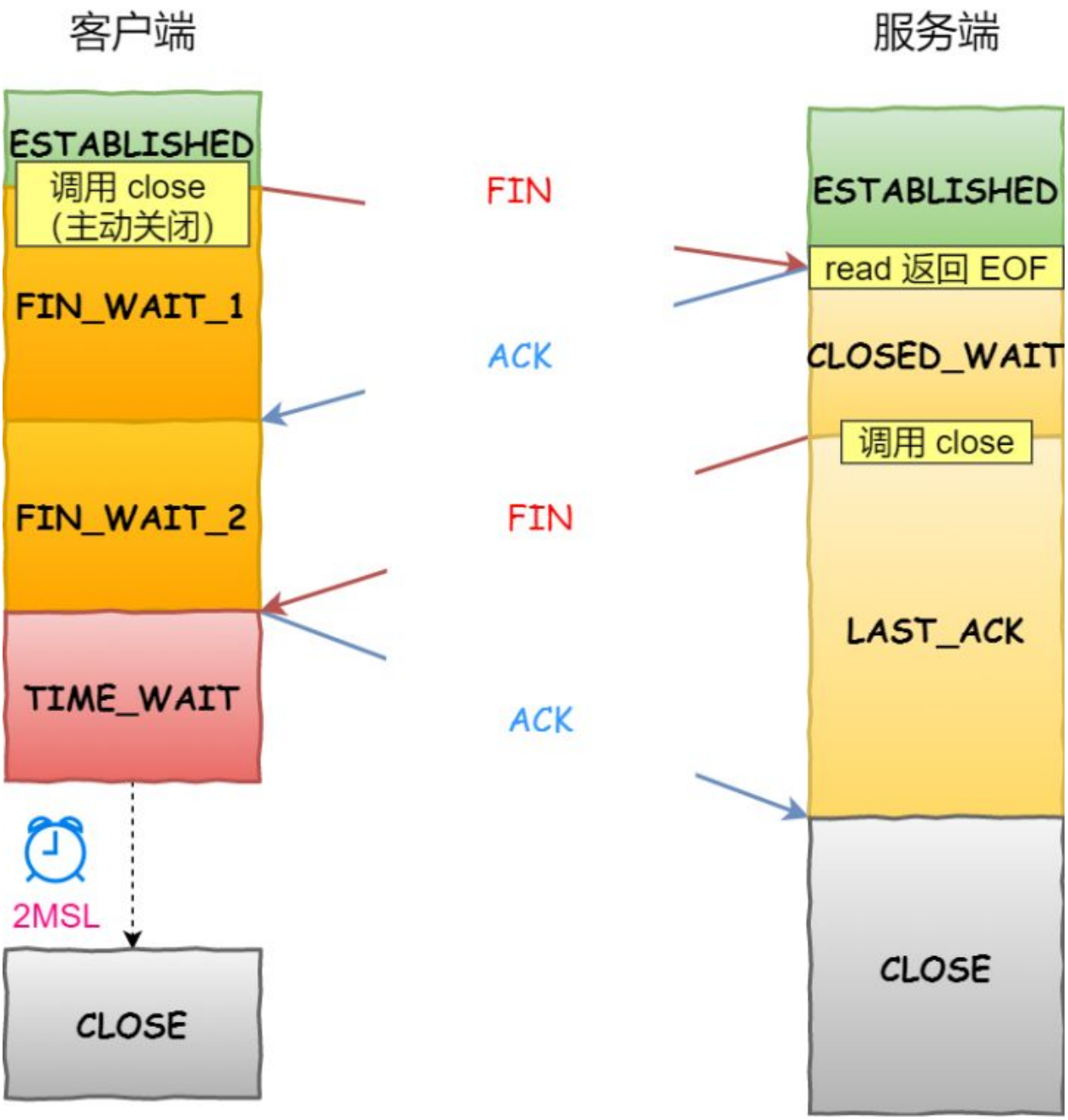
```
netstat -s | grep overflowed
```

这个值和半连接队列的查看一样是一个累加值。可以隔几秒钟观察这个数字，如果一直再增加，就应该调大accept队列。

TCP四次挥手性能优化

1. TCP四次挥手

客户端和服务端双方都可以主动断开连接，通常先关闭连接的一方称为主动方，后关闭连接的一方称为被动方。



可以看到，四次挥手只涉及到两种报文，FIN和ACK：

- FIN就是结束连接的意思，谁发出FIN报文，就表示它将不会再发送任何数据，关闭这一方向上的传输通道。
- ACK就是确认的意思，用来通知对方：你方的发送通道已关闭。

四次挥手的过程：

- 当主动方关闭连接时，会发送FIN报文，此时发送方的TCP连接状态将从ESTABLISHED变成FIN_WAIT1。
- 当被动方收到FIN报文时，内核会自动回复ACK报文，连接状态从ESTABLISHED变成CLOSE_WAIT，表示被动方等进程调用close函数关闭连接。
- 当主动方收到ACK后，连接状态由FIN_WAIT1变成FIN_WAIT2，也就是表示主动方的发送通道关闭了。
- 当被动方进入CLOSE_WAIT时，被动方还会继续处理数据，等到进程read函数返回0后，应用程序就会调用close函数，进而触发内核发送FIN报文，此时被动方的连接状态从CLOSE_WAIT转为LAST_ACK。
- 当主动方接收到这个FIN报文后，内核会回复ACK报文给被动方，同时主动方的连接状态由FIN_WAIT_2变为TIME_WAIT，在linux系统下大约一分钟后，TIME_WAIT状态的连接才会彻底关闭。
- 当被动方收到最后的ACK报文后，被动方的连接就会关闭。

你可以看到，每个方向都需要一个FIN和一个ACK报文，因此TCP关闭连接被称为4次挥手。主动关闭连接的，才会有TIME_WAIT状态。

2.主动方的优化

关闭连接的方式通常有两种，分别是RST报文关闭和FIN报文关闭。 如果进程异常退出了，内核就会发送RST报文暴力关闭连接，不走四次挥手流程。 安全关闭连接的方式必须通过四次挥手，它由进程调用close和shutdown函数发起FIN报文。 调用了close函数意味着完全断开连接，完全断开不仅指无法传输数据，而且也不能发送数据。此时，如果用**netstat -p**命令会发现对应的进程名为空。 使用close函数关闭连接是不优雅的。于是，就出现了一种优雅关闭连接的shutdown函数，它可以控制只关闭一个方向的连接。

```
int shutdown(int sock, int howto);
```

第二个参数决定断开连接的方式，主要有以下三种方式：

- SHUT_RD(0):关闭连接的读这条通道，如果接收缓冲区有已接收的数据，则将会被丢弃，并且后续再收到新的数据，会对数据进行ACK，然后悄悄丢弃。也就是说另一方还是可以收到ACK，但这种情况根本不知道数据已被丢弃。
- SHUT_WR(1):关闭连接写这条通道，这就是常被称为**半关闭**的连接。如果发送缓冲区有未发送的数据将被立即发送出去，并发送一个FIN报文给另一方。
- SHUT_RDWR(2):相当于SHUT_RD 和 SHUT_WR 操作各一次，关闭套接字的读和写两个方向。

3.FIN_WAIT1状态的优化

主动方发送FIN报文后，连接就一直处于FIN_WAIT1状态。正常情况下，如果主动方能及时的收到被动方的ACK，状态很快就能变为FIN_WAIT2。但是如果迟迟没收到被动方的ACK报文，内核会定时重发FIN报文，其中重发次数由**tcp_orphan_retries**参数控制，默认是0.

```
[ptopenmng@iZbp19ftqv2b83s9th37tuZ ~]$ cat /proc/sys/net/ipv4/tcp_orphan_retries
0
```

但是这个0，特指8次。从linux内核代码可得知。

```
/* Calculate maximal number or retries on an orphaned socket. */
static int tcp_orphan_retries(struct sock *sk, int alive)
{
        int retries = sysctl_tcp_orphan_retries; /* May be zero. */

        /* We know from an ICMP that something is wrong. */
        if (sk->sk_err_soft && !alive)
                retries = 0;

        /* However, if socket sent something recently, select some safe
         * number of retries. 8 corresponds to >100 seconds with minimal
         * RTO of 200msec. */
        if (retries == 0 && alive)
                retries = 8;
        return retries;
}
```

如果FIN_WAIT1状态连接很多，我们就需要考虑降低tcp_orphan_retries的值，当重传次数超过tcp_orphan_retries时，连接就会直接关闭。 对于普遍正常情况，调低tcp_orphan_retries就已经可以了。如果遭遇到恶意攻击,FIN报文根本无法发出去，这是由于TCP的两个特性导致：

- 首先，TCP必须保证报文是有序发送的，FIN报文也不例外，当发送缓冲区还有数据没有发送时，FIN报文也不能提前发送。
- 其次，TCP有流量控制功能，当接收方窗口为0时，发送方就不能再发送数据。所以，当攻击者下载大文件时，就可以将接收窗口设为0，这就使得FIN报文都无法发送出去，那么连接就会一直处于FIN_WAIT1状态。

这种情况，可以通过调整**tcp_max_orphans**参数，它定义了孤儿连接的最大数量。 当进程调用了close函数关闭连接，此时连接就是孤儿连接，linux系统为了防止孤儿连接过多，占用系统资源，就设置了最大数量，当孤儿连接超过最大数量时，新增的孤儿连接就不走四次挥手，而是直接发送RST报文强制关闭。

4. FIN_WAIT2状态的优化

当主动方收到ACK报文后，就会进入FIN_WAIT2状态，就表示主动方的发送通道已经关闭，接下来等被动方发送FIN报文，关闭接收通道。 这时，如果连接时用shutdown函数关闭的，连接一直处于FIN_WAIT2状态，因为它可能还可以发送或接收数据。但对于close函数关闭的孤儿连接，由于无法发送和接收数据，所以这个状态不可以持续太久。而**tcp_fin_timeout**控制了这个状态下连接的持续时长，默认值是60秒。 这意味着对于孤儿连接，如果60秒后还没有收到FIN报文，连接就会直接关闭。

5. TIME_WAIT状态的优化

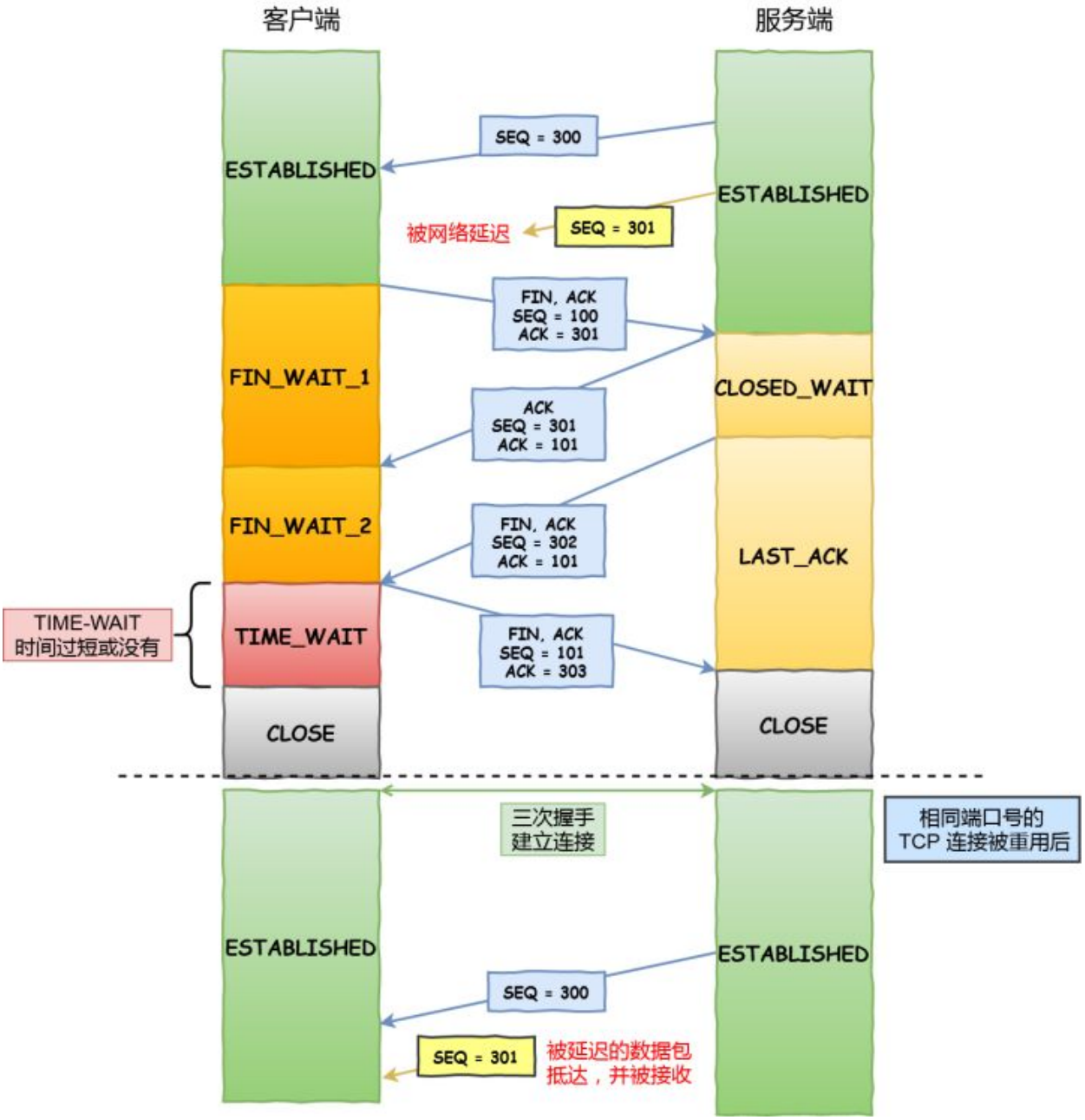
TIME_WAIT是主动方四次挥手的最后一个状态，也是最常遇见的状态。 当收到被动方发来的FIN报文后，主动方会回复ACK，表示确认对方的发送通道已经关闭。接着就处于TIME_WAIT状态。在linux系统，TIME_WAIT状态会持续60秒后才进入关闭状态。 TIME_WAIT状态的连接，在主动方看

来确实快关闭了。但被动方没有收到ACK前，一直处于LAST_ACK状态。如果这个ACK没有到达被动方，被动方就会重发FIN报文。 TIME_WAIT状态尤其重要，主要是两个原因：

- 防止具有相同的“四元组”的旧数据包被收到
- 保证被动关闭连接的一方能正确的关闭，即保证最后的ACK能让被动方接收，从而帮助其正常关闭。

5.1 防止旧连接的数据包被接收

TIME_WAIT的一个作用是防止收到历史数据，从而导致数据错乱的问题。

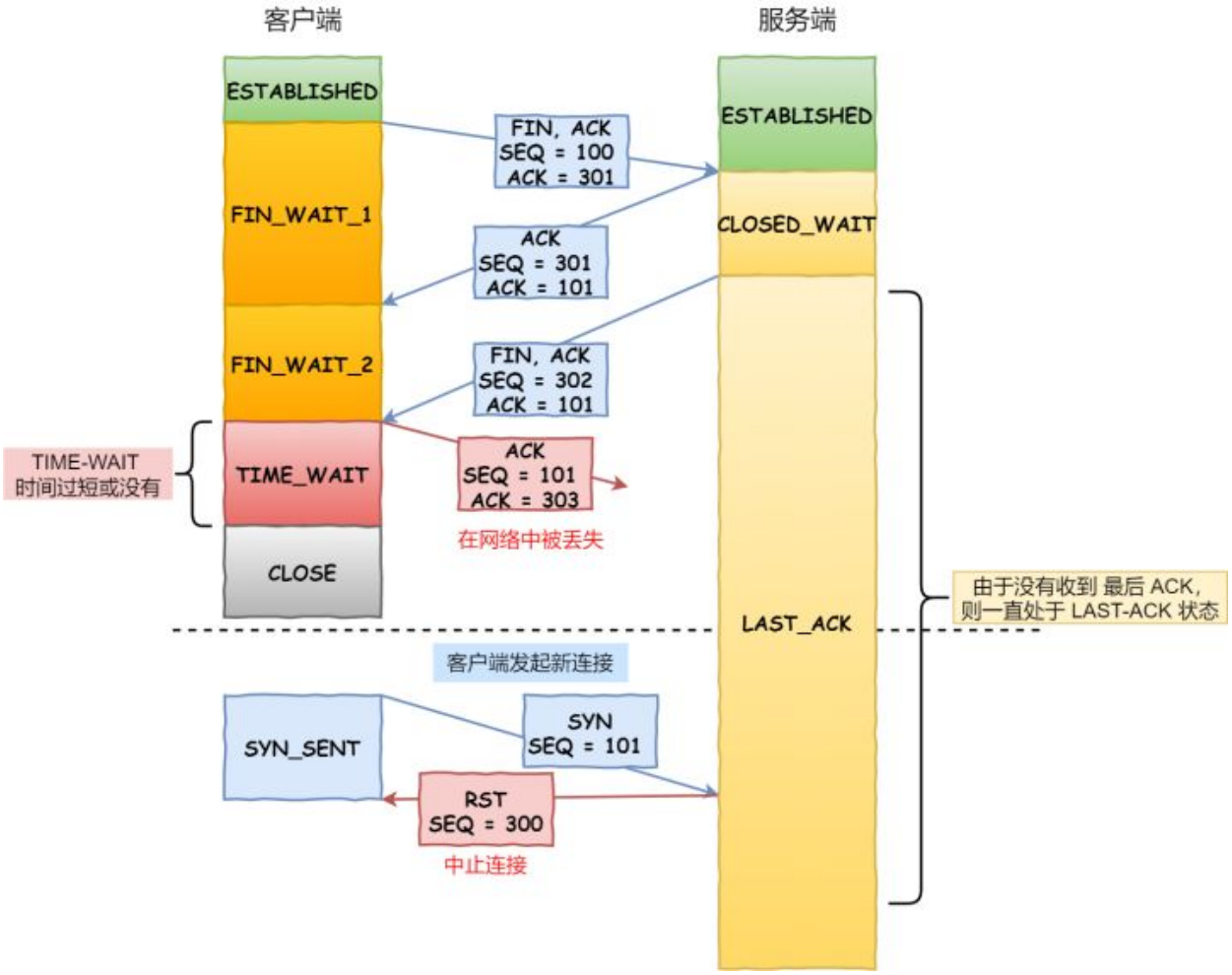


- 如上图黄色框框服务端在关闭连接之前发送的SEQ=301报文，被网络延迟了。
- 这时有相同端口的TCP连接被复用后，被延迟的SEQ=301抵达了客户端，那么客户端是有可能正常接收这个过期的报文，这时就会产生数据错乱的问题。

所以TCP就设计出这么一个机制，经过2MSL这个时间，足够让两个方向上的数据包都被丢弃，使得原来连接的数据包在网络中都自然消失，再出现的数据包一定是新建立连接所产生的。

5.2 保证连接正确关闭

TIME_WAIT的另外一个作用是等待足够的时间以确保最后的ACK能让被动关闭方接收，从而帮助其正常关闭。



- 如上图红色框框客户端四次挥手的最后一个ACK报文如果在网络中被丢失了，此时如果客户端直接close，那么服务端会一直处在LAST_ACK状态。重发FIN报文。
- 当客户端发起建立连接的SYN请求报文后，服务端可能会发送RST报文给客户端，连接建立的过程就会被中止。

5.3 TIME_WAIT状态的优化

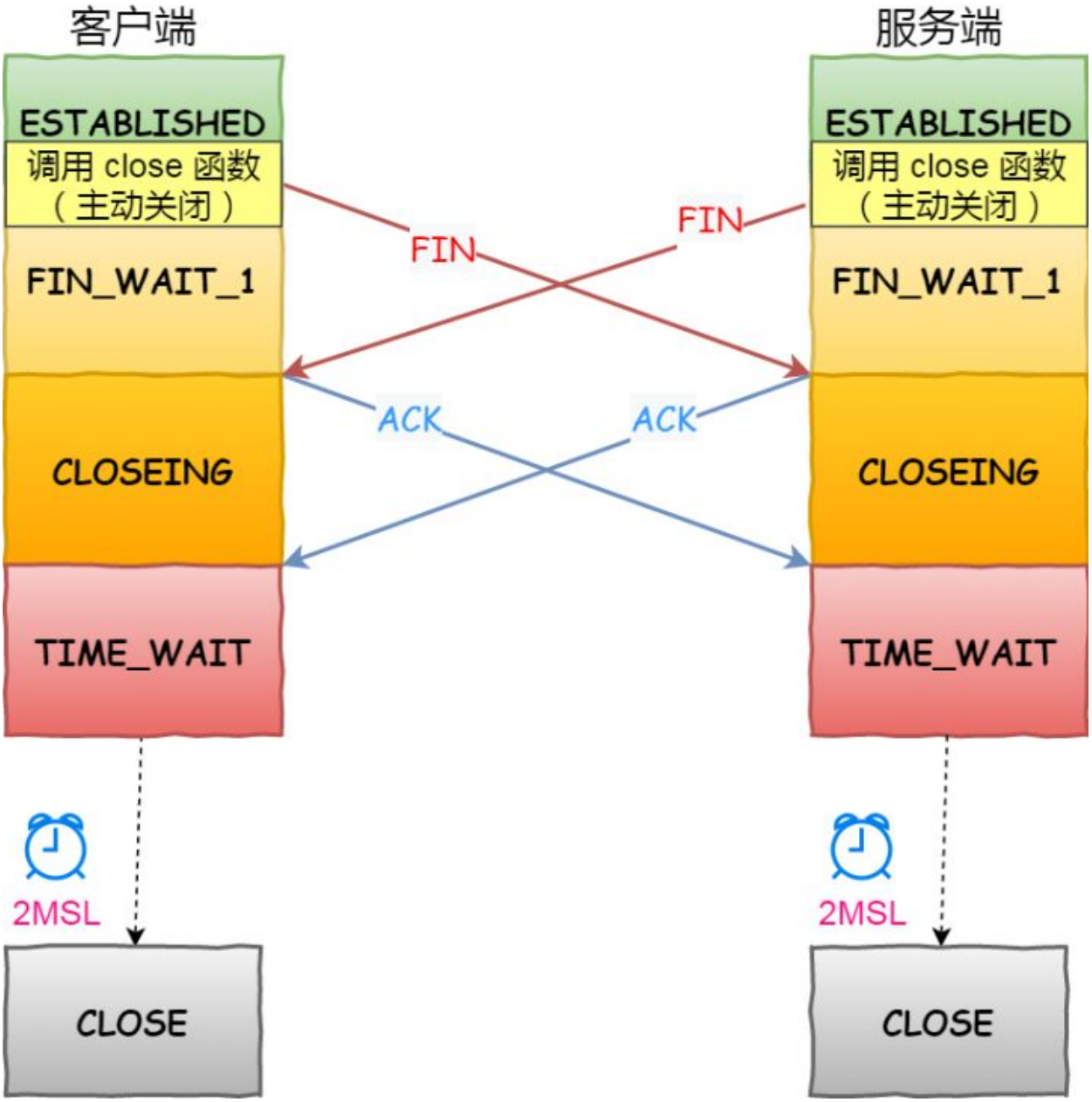
虽然TIME_WAIT状态有存在的必要，但它会消耗系统资源。如果发起连接一方的TIME_WAIT状态过多，占满了所有端口资源，则会导致无法创建新连接。

- 客户端受端口资源限制：如果客户端TIME_WAIT状态过多，就会导致端口资源被占用，端口就65536个。
- 服务端受系统资源限制：由于一个四元组表示TCP连接，理论上服务端可以建立很多连接，服务端确实只监听一个端口，但是会把连接扔给处理线程，但是TIME_WAIT状态的连接过多，会导致系统资源被占满，处理不了新的连接。

linux提供了**tcp_max_tw_buckets**参数控制TIME_WAIT状态的连接个数，当系统超过该数值时，新关闭的连接就不再经历TIME_WAIT而直接关闭。

6. 连接双方同时关闭连接

由于TCP是全双工的协议，所以会出现双方同时关闭连接的情况，也就是同时发送了FIN报文。两边同时发送FIN报文，都认为自己是主动方，同时进入FIN_WAIT1状态。



双方在等待ACK报文的过程中，都等来了FIN报文。这是一种新情况，连接会进入CLOSING的状态，它替代了FIN_WAIT2状态。接着双方内核回复ACK，同时进入TIME_WAIT状态。