计网题目 2

jask

09/10/2024

流量控制

发送方不能无脑的发数据给接收方，要考虑接收方处理能力。

如果一直无脑的发数据给对方，但对方处理不过来，那么就会导致触发重发机制，从而导致网络流量的无端的浪费。

为了解决这种现象发生，TCP 提供一种机制可以让「发送方」根据「接收方」的实际接收能力控制发送的数据量，这就是所谓的流量控制。

操作系统缓冲区与滑动窗口的关系

前面的流量控制例子，我们假定了发送窗口和接收窗口是不变的，但是实际上，发送窗口和接收窗口中所存放的字节数，都是放在操作系统内存缓冲区中的，而操作系统的缓冲区，会被操作系统调整。

当应用进程没办法及时读取缓冲区的内容时，也会对我们的缓冲区造成影响。

那操心系统的缓冲区，是如何影响发送窗口和接收窗口的呢？

当服务端系统资源非常紧张的时候，操心系统可能会直接减少了接收缓冲区大小，这时应用程序又无法及时读取缓存数据，那么这时候就有严重的事情发生了，会出现数据包丢失的现象。

为了防止这种情况发生，TCP 规定是不允许同时减少缓存又收缩窗口的，而是采用先收缩窗口，过段时间再减少缓存，这样就可以避免了丢包情况。

窗口管理

TCP 通过让接收方指明希望从发送方接收的数据大小（窗口大小）来进行流量控制。

如果窗口大小为 0 时，就会阻止发送方给接收方传递数据，直到窗口变为非 0 为止，这就是窗口关闭。

窗口关闭潜在的危险

接收方向发送方通告窗口大小时，是通过 ACK 报文来通告的。

那么，当发生窗口关闭时，接收方处理完数据后，会向发送方通告一个窗口非 0 的 ACK 报文，如果这个通告窗口的 ACK 报文在网络中丢失了，那麻烦就大了。

这会导致发送方一直等待接收方的非 0 窗口通知，接收方也一直等待发送方的数据，如不采取措施，这种相互等待的过程，会造成了死锁的现象。

TCP 是如何解决窗口关闭时，潜在的死锁现象呢？

为了解决这个问题，TCP 为每个连接设有一个持续定时器，只要 TCP 连接一方收到对方的零窗口通知，就启动持续计时器。

如果持续计时器超时，就会发送窗口探测 (Window probe) 报文，而对方在确认这个探测报文时，给出自己现在的接收窗口大小。

如果接收窗口仍然为 0，那么收到这个报文的一方就会重新启动持续计时器；

如果接收窗口不是 0，那么死锁的局面就可以被打破了。

窗口探测的次数一般为 3 次，每次大约 30-60 秒（不同的实现可能会不一样）。如果 3 次过后接收窗口还是 0 的话，有的 TCP 实现就会发 RST 报文来中断连接。

糊涂窗口综合症

如果接收方太忙了，来不及取走接收窗口里的数据，那么就会导致发送方的发送窗口越来越小。

到最后，如果接收方腾出几个字节并告诉发送方现在有几个字节的窗口，而发送方会义无反顾地发送这几个字节，这就是糊涂窗口综合症。

要知道，我们的 TCP + IP 头有 40 个字节，为了传输那几个字节的数据，要达上这么大的开销，这太不经济了。

糊涂窗口综合症的现象是可以发生在发送方和接收方：

接收方可以通告一个小的窗口

而发送方可以发送小数据

怎么让接收方不通告小窗口呢？

接收方通常的策略如下：

当「窗口大小」小于 min(MSS，缓存空间/2) ，也就是小于 MSS 与 1/2 缓存大小中的最小值时，就会向发送方通告窗口为 0 ，也就阻止了发送方再发数据过来。

怎么让发送方避免发送小数据呢？

发送方通常的策略：

使用 Nagle 算法，该算法的思路是延时处理，它满足以下两个条件中的一条才可以发送数据：

要等到窗口大小 >= MSS 或是数据大小 >= MSS 收到之前发送数据的 ack 回包只要没满足上面条件中的一条，发送方一直在囤积数据，直到满足上面的发送条件。

另外,Nagle 算法默认是打开的，如果对于一些需要小数据包交互的场景的程序，比如,telnet 或 ssh 这样的交互性比较强的程序，则需要关闭 Nagle 算法。

可以在 Socket 设置 TCP_NODELAY 选项来关闭这个算法 (关闭 Nagle 算法没有全局参数，需要根据每个应用自己的特点来关闭)。

拥塞控制

为什么要有拥塞控制呀，不是有流量控制了吗？

前面的流量控制是避免「发送方」的数据填满「接收方」的缓存，但是并不知道网络的中发生了什么。

一般来说，计算机网络都处在一个共享的环境。因此也有可能会因为其他主机之间的通信使得网络拥堵。

在网络出现拥堵时，如果继续发送大量数据包，可能会导致数据包时延、丢失等，这时 TCP 就会重传数据，但是一重传就会导致网络的负担更重，于是会导致更大的延迟以及更多的丢包，这个情况就会进入恶性循环被不断地放大….

所以,TCP 不能忽略网络上发生的事，它被设计成一个无私的协议，当网络发送拥塞时,TCP 会自我牺牲，降低发送的数据量。

于是，就有了拥塞控制，控制的目的就是避免「发送方」的数据填满整个网络。

为了在「发送方」调节所要发送数据的量，定义了一个叫做「拥塞窗口」的概念。

什么是拥塞窗口？和发送窗口有什么关系呢？

拥塞窗口 cwnd 是发送方维护的一个的状态变量，它会根据网络的拥塞程度动态变化的。

前面提到过发送窗口 swnd 和接收窗口 rwnd 是约等于的关系，那么由于加入了拥塞窗口的概念后，此时发送窗口的值是 swnd = min(cwnd, rwnd)，也就是拥塞窗口和接收窗口中的最小值。

拥塞窗口 cwnd 变化的规则：

只要网络中没有出现拥塞，cwnd 就会增大；

但网络中出现了拥塞，cwnd 就减少；

那么怎么知道当前网络是否出现了拥塞呢？

其实只要「发送方」没有在规定时间内接收到 ACK 应答报文，也就是发生了超时重传，就会认为网络出现了用拥塞。

拥塞控制有哪些控制算法？

拥塞控制主要是四个算法：

慢启动

拥塞避免

拥塞发生

快速恢复

慢启动

TCP 在刚建立连接完成后，首先是有个慢启动的过程，这个慢启动的意思就是一点一点的提高发送数据包的数量，如果一上来就发大量的数据，这不是给网络添堵吗？

慢启动的算法记住一个规则就行：当发送方每收到一个 ACK，拥塞窗口 cwnd 的大小就会加 1。

这里假定拥塞窗口 cwnd 和发送窗口 swnd 相等，下面举个栗子：

连接建立完成后，一开始初始化 cwnd = 1 ，表示可以传一个 MSS 大小的数据。

当收到一个 ACK 确认应答后,cwnd 增加 1，于是一次能够发送 2 个

当收到 2 个的 ACK 确认应答后，cwnd 增加 2，于是就可以比之前多发 2 个，所以这一次能够发送 4 个

当这 4 个的 ACK 确认到来的时候，每个确认 cwnd 增加 1，4 个确认 cwnd 增加 4，于是就可以比之前多发 4 个，所以这一次能够发送 8 个。

可以看出慢启动算法，发包的个数是指数性的增长。

那慢启动涨到什么时候是个头呢？

有一个叫慢启动门限 ssthresh (slow start threshold) 状态变量。

当 cwnd < ssthresh 时，使用慢启动算法。

当 cwnd >= ssthresh 时，就会使用「拥塞避免算法」。

拥塞避免算法

前面说道，当拥塞窗口 cwnd 「超过」慢启动门限 ssthresh 就会进入拥塞避免算法。

一般来说 ssthresh 的大小是 65535 字节。

那么进入拥塞避免算法后，它的规则是：每当收到一个 ACK 时,cwnd 增加 1/cwnd。

接上前面的慢启动的栗子，现假定 ssthresh 为 8 ：

当 8 个 ACK 应答确认到来时，每个确认增加 1/8,8 个 ACK 确认 cwnd 一共增加 1，于是这一次能够发送 9 个 MSS 大小的数据，变成了线性增长。

我们可以发现，拥塞避免算法就是将原本慢启动算法的指数增长变成了线性增长，还是增长阶段，但是增长速度缓慢了一些。

就这么一直增长着后，网络就会慢慢进入了拥塞的状况了，于是就会出现丢包现象，这时就需要对丢失的数据包进行重传。

当触发了重传机制，也就进入了「拥塞发生算法」。

拥塞发生

当网络出现拥塞，也就是会发生数据包重传，重传机制主要有两种：

超时重传

快速重传

这两种使用的拥塞发送算法是不同的，接下来分别来说说。

发生超时重传的拥塞发生算法　当发生了「超时重传」，则就会使用拥塞发生算法。

这个时候,ssthresh 和 cwnd 的值会发生变化：

ssthresh 设为 cwnd/2 ，

cwnd 重置为 1

接着，就重新开始慢启动，慢启动是会突然减少数据流的。这真是一旦「超时重传」，马上回到解放前。但是这种方式太激进了，反应也很强烈，会造成网络卡顿。

发生快速重传的拥塞发生算法 还有更好的方式，前面我们讲过「快速重传算法」。当接收方发现丢了一个中间包的时候，发送三次前一个包的 ACK，于是发送端就会快速地重传，不必等待超时再重传。

TCP 认为这种情况不严重，因为大部分没丢，只丢了一小部分，则 ssthresh 和 cwnd 变化如下：

cwnd = cwnd/2 ，也就是设置为原来的一半；

ssthresh = cwnd ；

进入快速恢复算法

快速恢复 快速重传和快速恢复算法一般同时使用，快速恢复算法是认为，你还能收到 3 个重复 ACK 说明网络也不那么糟糕，所以没有必要像 RTO 超时那么强烈。

正如前面所说，进入快速恢复之前，cwnd 和 ssthresh 已被更新了：

cwnd = cwnd/2 ，也就是设置为原来的一半；

ssthresh = cwnd ；

然后，进入快速恢复算法如下：

拥塞窗口 cwnd = ssthresh + 3 (3 的意思是确认有 3 个数据包被收到了)；

重传丢失的数据包；

如果再收到重复的 ACK，那么 cwnd 增加 1；

如果收到新数据的 ACK 后，把 cwnd 设置为第一步中的 ssthresh 的值，原因是该 ACK 确认了新的数据，说明从 duplicated ACK 时的数据都已收到，该恢复过程已经结束，可以回到恢复之前的状态了，也即再次进入拥塞避免状态；

也就是没有像「超时重传」一夜回到解放前，而是还在比较高的值，后续呈线性增长。

实际情况

第一次握手 SYN 丢包

可以发现，每次超时时间 RTO 是指数（翻倍）上涨的，当超过最大重传次数后，客户端不再发送 SYN 包。

在 Linux 中，第一次握手的 SYN 超时重传次数，是如下内核参数指定的：

```
$ cat /proc/sys/net/ipv4/tcp_syn_retries5
```

tcp_syn_retries 默认值为 5，也就是 SYN 最大重传次数是 5 次。

当客户端发起的 TCP 第一次握手 SYN 包，在超时时间内没收到服务端的 ACK，就会在超时重传 SYN 数据包，每次超时重传的 RTO 是翻倍上涨的，直到 SYN 包的重传次数到达 tcp_syn_retries 值后，客户端不再发送 SYN 包。

TCP 第二次握手 SYN、ACK 丢包

当第二次握手的 SYN、ACK 丢包时，客户端会超时重发 SYN 包，服务端也会超时重传 SYN、ACK 包。

cp_syn_retries 是限制 SYN 重传次数，那第二次握手 SYN、ACK 限制最大重传次数是多少？

TCP 第二次握手 SYN、ACK 包的最大重传次数是通过 tcp_synack_retries 内核参数限制的，其默认值如下：

```
$ cat /proc/sys/net/ipv4/tcp_synack_retries 5
```

是的,TCP 第二次握手 SYN、ACK 包的最大重传次数默认值是 5 次。

当 TCP 第二次握手 SYN、ACK 包丢了后，客户端 SYN 包会发生超时重传，服务端 SYN、ACK 也会发生超时重传。

客户端 SYN 包超时重传的最大次数，是由 tcp_syn_retries 决定的，默认值是 5 次；服务端 SYN、ACK 包时重传的最大次数，是由 tcp_synack_retries 决定的，默认值是 5 次。

第三次握手丢包

服务端在进入 SYN_RECV 状态之后没有进入 ESTABLISHED 状态，服务端超时重传了 SYN、ACK 包，重传了 5 次后，也就是超过 tcp_synack_retries 的值（默认值是 5），然后就没有继续重传了，此时服务端的 TCP 连接主动中止了，所以刚才处于 SYN_RECV 状态的 TCP 连接断开了，而客户端依然处于 ESTABLISHED 状态；

虽然服务端 TCP 断开了，但过了一段时间，发现客户端依然处于 ESTABLISHED 状态，于是就在客户端的 telnet 会话输入了 123456 字符；

此时由于服务端已经断开连接，客户端发送的数据报文，一直在超时重传，每一次重传,RTO 的值是指数增长的，所以持续了好长一段时间，客户端的 telnet 才报错退出了，此时共重传了 15 次。

TCP 第一次握手的 SYN 包超时重传最大次数是由 tcp_syn_retries 指定,TCP 第二次握手的 SYN、ACK 包超时重传最大次数是由 tcp_synack_retries 指定，那 TCP 建立连接后的数据包最大超时重传次数是由什么参数指定呢？ TCP 建立连接后的数据包传输，最大超时重传次数是由 tcp_retries2 指定，默认值是 15 次，如下：

```
$ cat /proc/sys/net/ipv4/tcp_retries2
15
```

那如果客户端不发送数据，什么时候才会断开处于 ESTABLISHED 状态的连接？

这里就需要提到 TCP 的保活机制。这个机制的原理是这样的：

定义一个时间段，在这个时间段内，如果没有任何连接相关的活动,TCP 保活机制会开始作用，每隔一个时间间隔，发送一个「探测报文」，该探测报文包含的数据非常少，如果连续几个探测报文都没有得到响应，则认为当前的 TCP 连接已经死亡，系统内核将错误信息通知给上层应用程序。

在 Linux 内核可以有对应的参数可以设置保活时间、保活探测的次数、保活探测的时间间隔，以下都为默认值：

```
net.ipv4.tcp_keepalive_time=7200 net.ipv4.tcp_keepalive_intvl=75 net.ipv4.tcp_keepalive_probes=9
```

tcp_keepalive_time=7200：表示保活时间是 7200 秒（2 小时），也就 2 小时内如果没有任何连接相关的活动，则会启动保活机制

tcp_keepalive_intvl=75：表示每次检测间隔 75 秒；

tcp_keepalive_probes=9：表示检测 9 次无响应，认为对方是不可达的，从而中断本次的连接。也就是说在 Linux 系统中，最少需要经过 2 小时 11 分 15 秒才可以发现一个「死亡」连接。

在建立 TCP 连接时，如果第三次握手的 ACK，服务端无法收到，则服务端就会短暂处于 SYN_RECV 状态，而客户端会处于 ESTABLISHED 状态。

由于服务端一直收不到 TCP 第三次握手的 ACK，则会一直重传 SYN、ACK 包，直到重传次数超过 tcp_synack_retries 值（默认值 5 次）后，服务端就会断开 TCP 连接。

而客户端则会有两种情况：

如果客户端没发送数据包，一直处于 ESTABLISHED 状态，然后经过 2 小时 11 分 15 秒才可以发现一个「死亡」连接，于是客户端连接就会断开连接。

如果客户端发送了数据包，一直没有收到服务端对该数据包的确认报文，则会一直重传该数据包，直到重传次数超过 tcp_retries2 值（默认值 15 次）后，客户端就会断开 TCP 连接。

TCP 快速建立连接

客户端在向服务端发起 HTTP GET 请求时，一个完整的交互过程，需要 2.5 个 RTT 的时延。

由于第三次握手是可以携带数据的，这时如果在第三次握手发起 HTTP GET 请求，需要 2 个 RTT 的时延。

但是在下一次（不是同个 TCP 连接的下一次）发起 HTTP GET 请求时，经历的 RTT 也是一样，如下图：

在 Linux 3.7 内核版本中，提供了 TCP Fast Open 功能，这个功能可以减少 TCP 连接建立的时延。

在第一次建立连接的时候，服务端在第二次握手产生一个 Cookie（已加密）并通过 SYN、ACK 包一起发给客户端，于是客户端就会缓存这个 Cookie ，所以第一次发起 HTTP Get 请求的时候，还是需要 2 个 RTT 的时延；

在下次请求的时候，客户端在 SYN 包带上 Cookie 发给服务端，就提前可以跳过三次握手的过程，因为 Cookie 中维护了一些信息，服务端可以从 Cookie 获取 TCP 相关的信息，这时发起的 HTTP GET 请求就只需要 1 个 RTT 的时延；

注：客户端在请求并存储了 Fast Open Cookie 之后，可以不断重复 TCP Fast Open 直至服务器认为 Cookie 无效（通常为过期）

TCP 重复确认和快速重传

当接收方收到乱序数据包时，会发送重复的 ACK，以便告知发送方要重发该数据包，当发送方收到 3 个重复 ACK 时，就会触发快速重传，立刻重发丢失数据包。

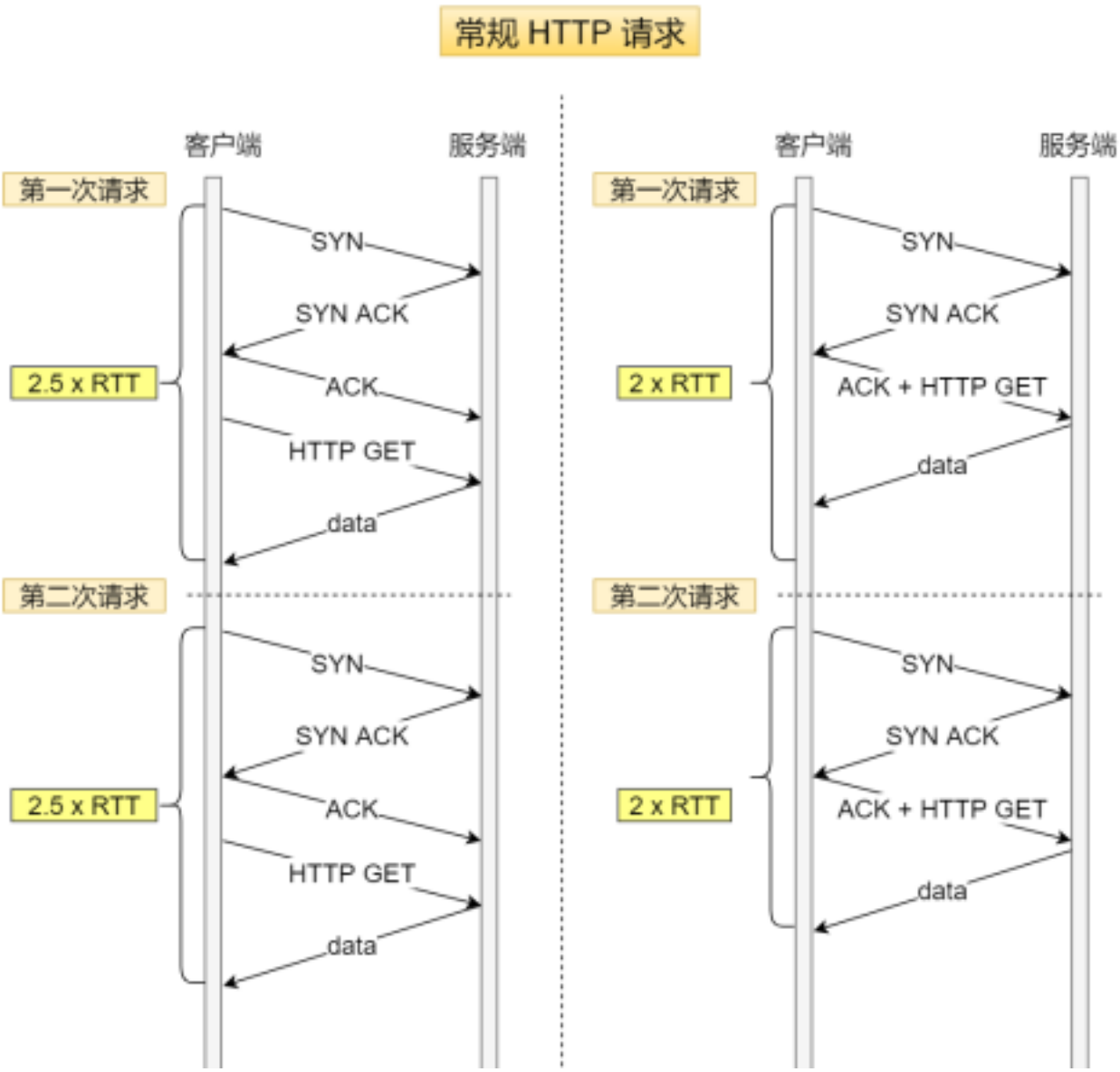


Figure 1: 常规请求

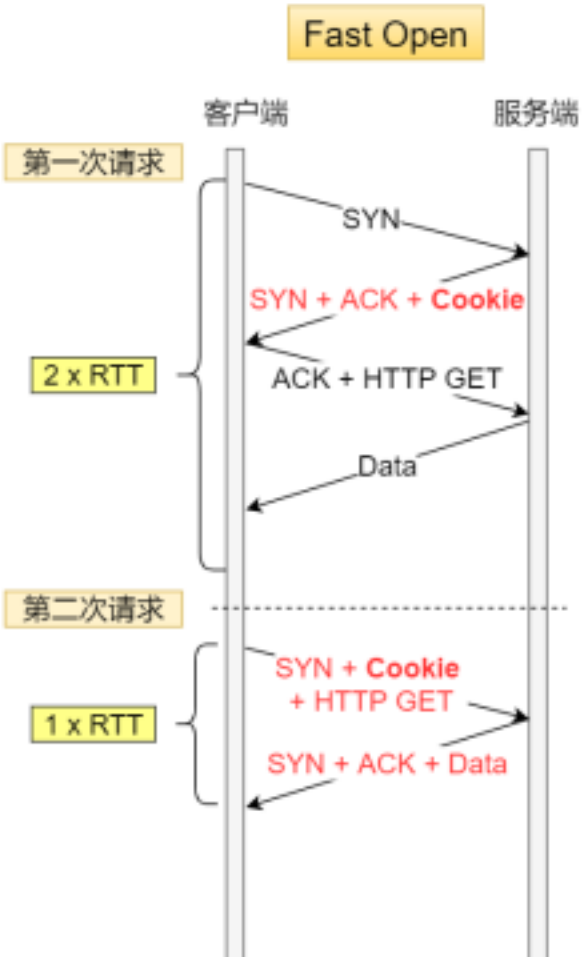


Figure 2: Fast Open

TCP 流量控制

TCP 为了防止发送方无脑的发送数据，导致接收方缓冲区被填满，所以就有了滑动窗口的机制，它可利用接收方的接收窗口来控制发送方要发送的数据量，也就是流量控制。

接收窗口是由接收方指定的值，存储在 TCP 头部中，它可以告诉发送方自己的 TCP 缓冲空间区大小，这个缓冲区是给应用程序读取数据的空间：

如果应用程序读取了缓冲区的数据，那么缓冲空间区就会把被读取的数据移除如果应用程序没有读取数据，则数据会一直滞留在缓冲区。

接收窗口的大小，是在 TCP 三次握手中协商好的，后续数据传输时，接收方发送确认应答 ACK 报文时，会携带当前的接收窗口的大小，以此来告知发送方。

零窗口通知与窗口探测

假设接收方处理数据的速度跟不上接收数据的速度，缓存就会被占满，从而导致接收窗口为 0，当发送方接收到零窗口通知时，就会停止发送数据。

接着，发送方会定时发送窗口大小探测报文，以便及时知道接收方窗口大小的变化。

发送窗口和 MSS 有什么关系？

发送窗口决定了一口气能发多少字节，而 MSS 决定了这些字节要分多少包才能发完。

举个例子，如果发送窗口为 16000 字节的情况下，如果 MSS 是 1000 字节，那就需要发送 1600/1000 = 16 个包。

发送方在一个窗口发出 n 个包，是不是需要 n 个 ACK 确认报文？

不一定，因为 TCP 有累计确认机制，所以当收到多个数据包时，只需要应答最后一个数据包的 ACK 报文就可以了。

连接队列

什么是 TCP 半连接队列和全连接队列？

在 TCP 三次握手的时候,Linux 内核会维护两个队列，分别是：

半连接队列，也称 SYN 队列；

全连接队列，也称 accepet 队列；

服务端收到客户端发起的 SYN 请求后，内核会把该连接存储到半连接队列，并向客户端响应 SYN+ACK，接着客户端会返回 ACK，服务端收到第三次握手的 ACK 后，内核会把连接从半连接队列移除，然后创建新的完全的连接，并将其添加到 accept 队列，等待进程调用 accept 函数时把连接取出来。

ss 命令查看全连接

当服务端并发处理大量请求时，如果 TCP 全连接队列过小，就容易溢出。发生 TCP 全连接队溢出的时候，后续的请求就会被丢弃，这样就会出现服务端请求数量上不去的现象。

Linux 有个参数可以指定当 TCP 全连接队列满了会使用什么策略来回应客户端。

tcp_abort_on_overflow 共有两个值分别是 0 和 1，其分别表示：

0 ：如果全连接队列满了，那么 server 扔掉 client 发过来的 ack ；

1 ：如果全连接队列满了,server 发送一个 reset 包给 client，表示废掉这个握手过程和这个连接；

如果要想知道客户端连接不上服务端，是不是服务端 TCP 全连接队列满的原因，那么可以把 tcp_abort_on_overflow 设置为 1，这时如果在客户端异常中可以看到很多 connection reset by peer 的错误，那么就可以证明是由于服务端 TCP 全连接队列溢出的问题。

tcp_abort_on_overflow 设为 0 可以提高连接建立的成功率，只有你非常肯定 TCP 全连接队列会长期溢出时，才能设置为 1 以尽快通知客户端。

如何增大 TCP 全连接队列呢？

TCP 全连接队列的最大值取决于 somaxconn 和 backlog 之间的最小值，也就是 min(somaxconn, backlog)

somaxconn 是 Linux 内核的参数，默认值是 128，可以通过 /proc/sys/net/core/somaxconn 来设置其值；

backlog 是 listen(int sockfd, int backlog) 函数中的 backlog 大小,Nginx 默认值是 511，可以通过修改配置文件设置其长度；

TCP 性能提高

TCP 三次握手的性能提升; TCP 四次挥手的性能提升; TCP 数据传输的性能提升;

三次握手

如何正确有效的使用这些参数, 来提高 TCP 三次握手的性能, 这就需要理解「三次握手的状态变迁」, 这样当出现问题时, 先用 netstat 命令查看是哪个握手阶段出现了问题, 再来对症下药, 而不是病急乱投医。

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,SYN 的全称就叫 Synchronize Sequence Numbers(同步序列号)。

SYN_SENT 状态的优化 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。

客户端在等待服务端回复的 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 5 次:

通常, 第一次超时重传是在 1 秒后, 第二次超时重传是在 2 秒, 第三次超时重传是在 4 秒后, 第四次超时重传是在 8 秒后, 第五次是在超时重传 16 秒后。没错, 每次超时的时间是上一次的 2 倍。

当第五次超时重传后, 会继续等待 32 秒, 如果服务端仍然没有回应 ACK, 客户端就会终止三次握手。

所以, 总耗时是 1+2+4+8+16+32=63 秒, 大约 1 分钟左右。

服务端优化

当服务端收到 SYN 包后, 服务端会立马回复 SYN+ACK 包, 表明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 内核就会建立一个「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, 服务端就无法再建立新的连接。

如何调整 SYN 半连接队列大小? 要想增大半连接队列, 不能只单纯增大 tcp_max_syn_backlog 的值, 还需一同增大 somaxconn 和 backlog, 也就是增大 accept 队列。否则, 只单纯增大 tcp_max_syn_backlog 是无效的。

增大 tcp_max_syn_backlog 和 somaxconn 的方法是修改 Linux 内核参数

如果 SYN 半连接队列已满, 只能丢弃连接吗? 并不是这样, 开启 syncookies 功能就可以在不使用 SYN 半连接队列的情况下成功建立连接。

syncookies 的工作原理: 服务器根据当前状态计算出一个值, 放在己方发出的 SYN+ACK 报文中发出, 当客户端返回 ACK 报文时, 取出该值验证, 如果合法, 就认为连接建立成功

如何查看服务端进程 accept 队列的长度? 可以通过 ss -ltn 命令查看:

如何绕过三次握手? 三次握手建立连接造成的后果就是,HTTP 请求必须在一个 RTT(从客户端到服务器一个往返的时间) 后才能发送。

开启 Tcp Fast Open TCP Fast Open 功能需要客户端和服务端同时支持, 才有效果。

如果客户端再次向服务器建立连接时的过程:

1. 客户端发送 SYN 报文, 该报文包含「数据」(对于非 TFO 的普通 TCP 握手过程,SYN 报文中不包含「数据」) 以及此前记录的 Cookie;
2. 支持 TCP Fast Open 的服务器会对收到 Cookie 进行校验: 如果 Cookie 有效, 服务器将在 SYN- ACK 报文中对 SYN 和「数据」进行确认, 服务器随后将「数据」递送至相应的应用程序; 如果 Cookie 无效, 服务器将丢弃 SYN 报文中包含的「数据」, 且其随后发出的 SYN-ACK 报文将只确认 SYN 的对应序列号;
3. 如果服务器接受了 SYN 报文中的「数据」, 服务器可在握手完成之前发送「数据」, 这就减少了握手带来的 1 个 RTT 的时间消耗;
4. 客户端将发送 ACK 确认服务器发回的 SYN 以及「数据」, 但如果客户端在初始的 SYN 报文中发送的「数据」没有被确认, 则客户端将重新发送「数据」;
5. 此后的 TCP 连接的数据传输过程和非 TFO 的正常情况一致。

所以, 之后发起 HTTP GET 请求的时候, 可以绕过三次握手, 这就减少了握手带来的 1 个 RTT 的时间消耗。

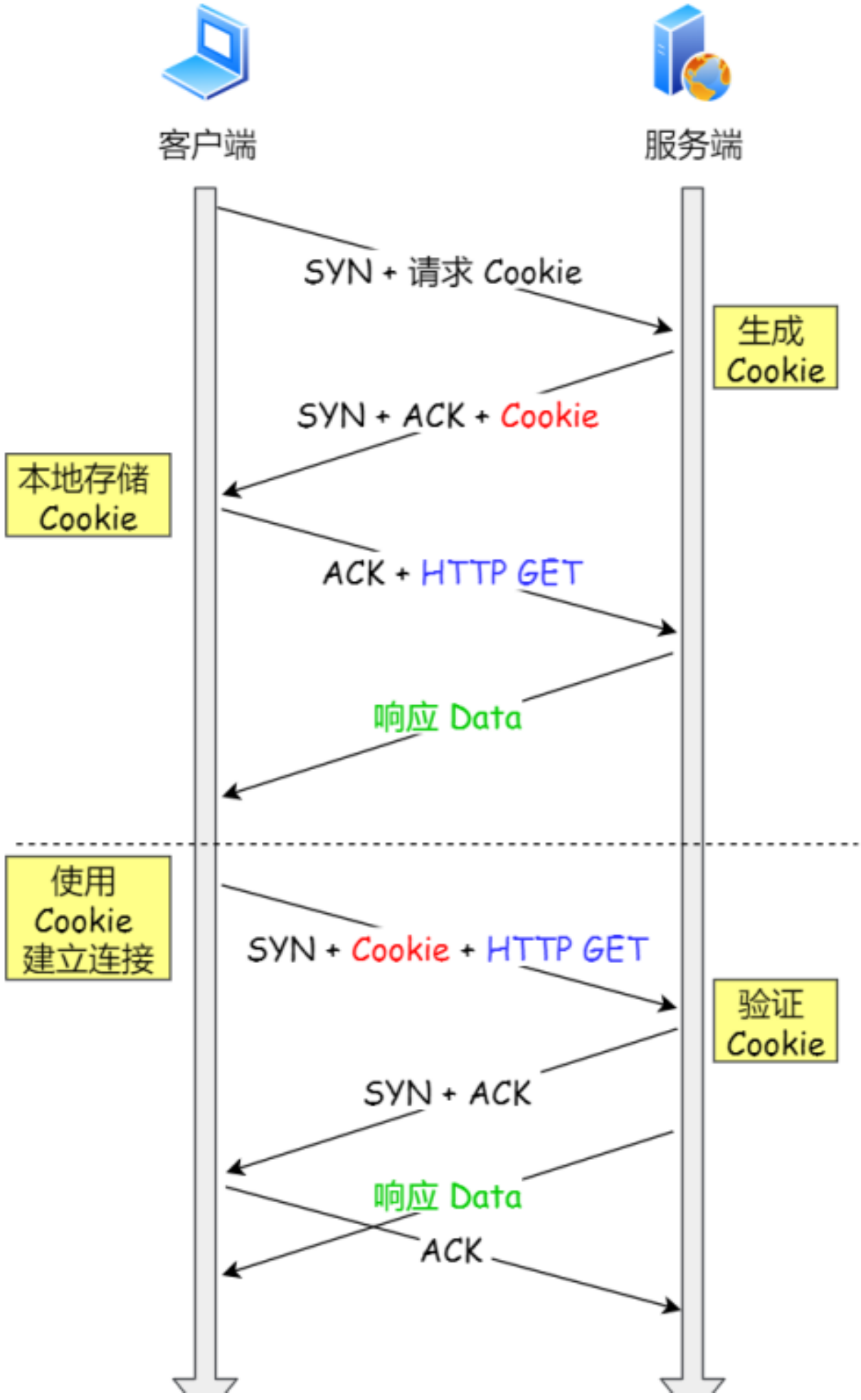


Figure 3: Cookie

TCP 四次挥手的性能提升

四次挥手过程只涉及了两种报文，分别是 FIN 和 ACK：

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 报文来关闭，它可以不走四次挥手流程，是一个暴力关闭连接的方式。

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（shutdown 参数须传入 SHUT_WR 或者 SHUT_RDWR 才会发送 FIN）。

调用 close 函数和 shutdown 函数有什么区别？ 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，调用了 close 函数的一方的连接叫做「孤儿连接」，如果你用 netstat -p 命令，会发现连接对应的进程名为空。

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。

FIN_WAIT1 状态的优化 主动方发送 FIN 报文后，连接就处于 FIN_WAIT1 状态，正常情况下，如果能及时收到被动方的 ACK，则会很快变为 FIN_WAIT2 状态。

但是当迟迟收不到对方返回的 ACK 时，连接就会一直处于 FIN_WAIT1 状态。此时，内核会定时重发 FIN 报文，其中重发次数由 tcp_orphan_retries 参数控制（注意,orphan 虽然是孤儿的意思，该参数却不只对孤儿连接有效，事实上，它对所有 FIN_WAIT1 状态下的连接都有效），默认值是 0。

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，连接就会直接关闭掉。

对于普遍正常情况时，调低 tcp_orphan_retries 就已经可以了。如果遇到恶意攻击,FIN 报文根本无法发送出去，这由 TCP 两个特性导致的：

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，当接收方接收窗口为 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，就可以通过接收窗口设为 0 ，这就会使得 FIN 报文都无法发送出去，那么连接会一直处于 FIN_WAIT1 状态。

解决这种问题的方法，是调整 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，此时连接就会是「孤儿连接」，因为它无法再发送和接收数据。

Linux 系统为了防止孤儿连接过多，导致系统资源长时间被占用，就提供了 tcp_max_orphans 参数。

如果孤儿连接数量大于它，新增的孤儿连接将不再走四次挥手，而是直接发送 RST 复位报文强制关闭。

FIN_WAIT2 状态的优化 当主动方收到 ACK 报文后，会处于 FIN_WAIT2 状态，就表示主动方的发送通道已经关闭，接下来将等待对方发送 FIN 报文，关闭对方的发送通道。这时，如果连接是用 shutdown 函数关闭的，连接可以一直处于 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，由于无法再发送和接收数据，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。

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，主动方会立刻回复 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 秒后才会进入关闭状态。TIME_WAIT 状态的连接，在主动方看来确实快已经关闭了。然后，被动方没有收到 ACK 报文前，还是处于 LAST_ACK 状态。如果这个 ACK 报文没有到达被动方，被动方就会重发 FIN 报文。重发次数仍然由前面介绍过的 tcp_orphan_retries 参数控制。TIME-WAIT 的状态尤其重要，主要是两个原因：

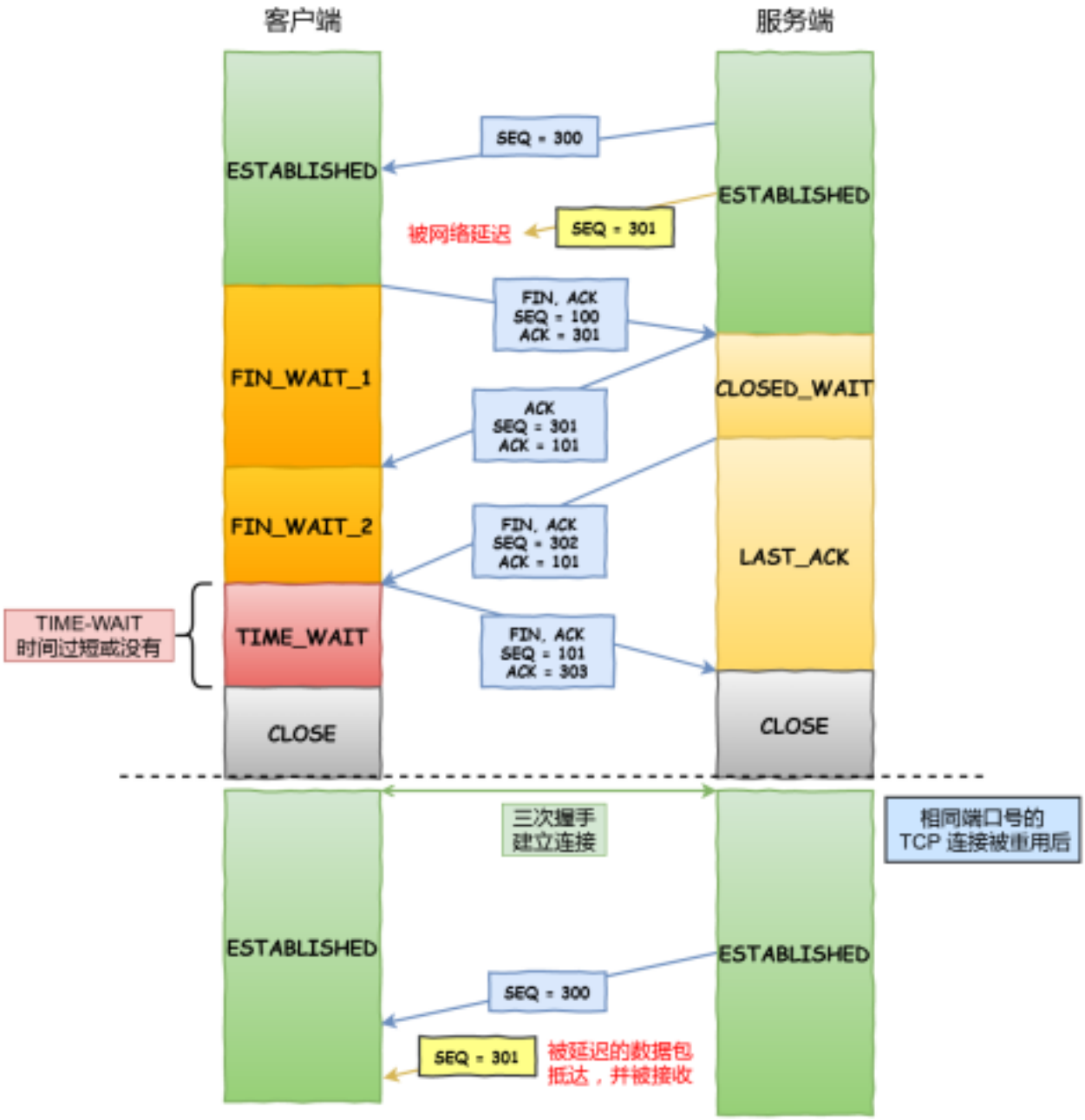
防止具有相同「四元组」的「旧」数据包被收到；

保证「被动关闭连接」的一方能被正确的关闭，即保证最后的 ACK 能让被动关闭方接收，从而帮助其正常关闭；

防止旧连接的数据包 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。

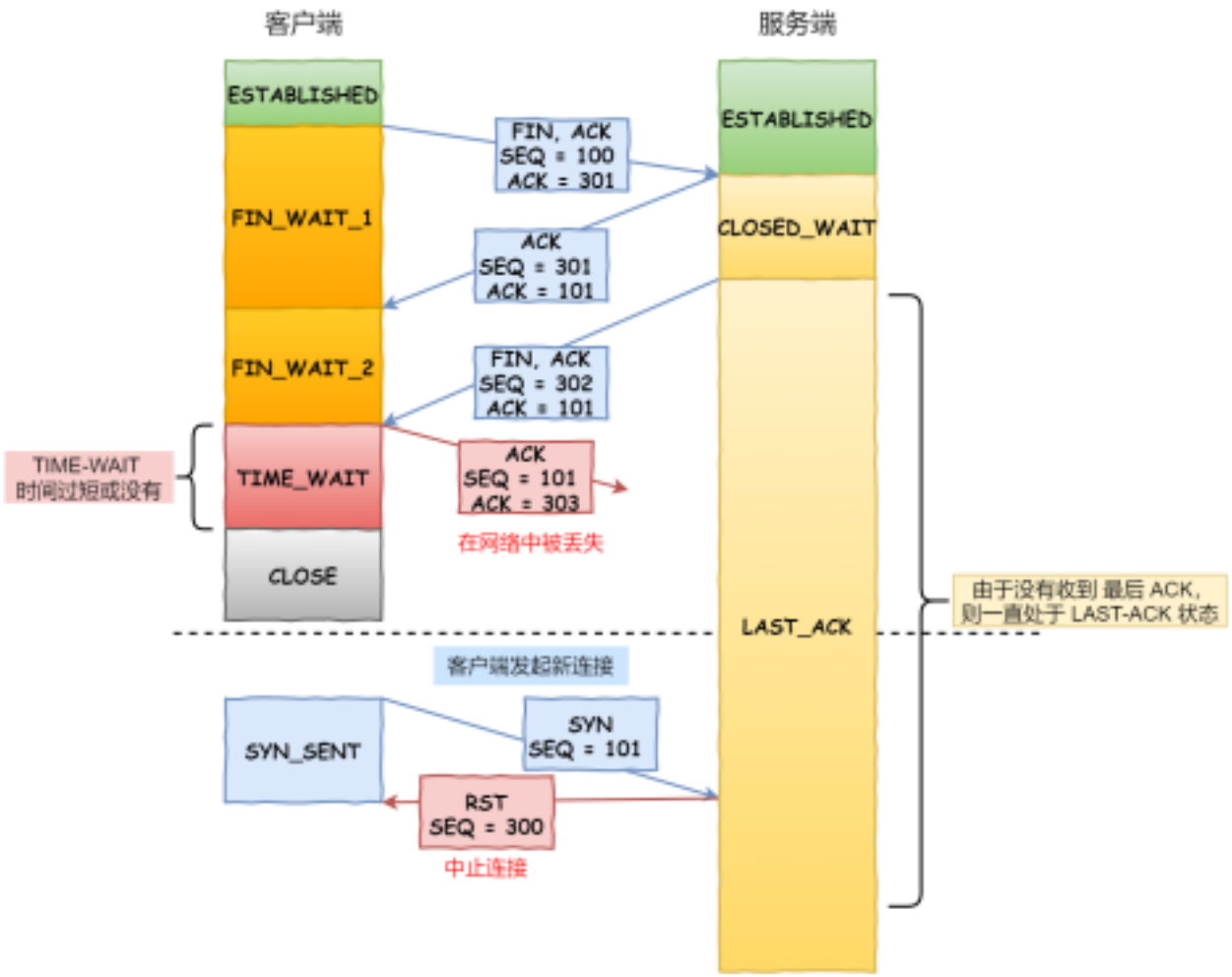
假设 TIME-WAIT 没有等待时间或时间过短，被延迟的数据包抵达后会发生什么呢？

所以,TCP 就设计出了这么一个机制，经过 2MSL 这个时间，足以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都是新建立连接所产生的。



- 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，这就会产生数据错乱等严重的问题。

Figure 4: 详情



- 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 `TIME-WAIT` 过短或没有，则就直接进入了 `CLOSE` 状态了，那么服务端则会一直处在 `LAST-ACK` 状态。
- 当客户端发起建立连接的 `SYN` 请求报文后，服务端会发送 `RST` 报文给客户端，连接建立的过程就会被终止。

Figure 5: 详情

保证连接正确关闭 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。

我们再回过头来看看，为什么 TIME_WAIT 状态要保持 60 秒呢？这与孤儿连接 FIN_WAIT2 状态默认保留 60 秒的原理是一样的，因为这两个状态都需要保持 2MSL 时长。MSL 全称是 Maximum Segment Lifetime，它定义了一个报文在网络中的最长生存时间（报文每经过一次路由器的转发，IP 头部的 TTL 字段就会减 1，减到 0 时报文就被丢弃，这就限制了报文的最长存活时间）。

为什么是 2 MSL 的时长呢？这其实是相当于至少允许报文丢失一次。比如，若 ACK 在一个 MSL 内丢失，这样被动方重发的 FIN 会在第 2 个 MSL 内到达，TIME_WAIT 状态的连接可以应对。

为什么不是 4 或者 8 MSL 的时长呢？你可以想象一个丢包率达到百分之一的糟糕网络，连续两次丢包的概率只有万分之一，这个概率实在是太小了，忽略它比解决它更具性价比。

因此，TIME_WAIT 和 FIN_WAIT2 状态的最大时长都是 2 MSL，由于在 Linux 系统中，MSL 的值固定为 30 秒，所以它们都是 60 秒。

虽然 TIME_WAIT 状态有存在的必要，但它毕竟会消耗系统资源。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 过多，就会导致端口资源被占用，因为端口就 65536 个，被占满就会导致无法创建新的连接；

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，所以理论上监听的端口可以继续监听。但是线程池处理不了那么多一直不断的连接了。所以当服务端出现大量 TIME_WAIT 时，系统资源被占满时，会导致处理不过来新的连接；

总结

优化四次挥手的策略	
策略	TCP 内核参数
调整 FIN 报文重传次数	tcp_orphan_retries
调整 FIN_WAIT2 状态的时间 (只适用 close 函数关闭的连接)	tcp_fin_timeout
调整孤儿连接的上限个数 (只适用 close 函数关闭的连接)	tcp_max_orphans
调整 time_wait 状态的上限个数	tcp_max_tw_buckets
复用 time_wait 状态的连接 (只适用客户端)	tcp_tw_reuse 、tcp_timestamps

Figure 6: 四次挥手优化

滑动窗口是如何影响传输速度的？

TCP 会保证每一个报文都能够抵达对方，它的机制是这样：报文发出去后，必须接收到对方返回的确认报文 ACK，如果迟迟未收到，就会超时重发该报文，直到收到对方的 ACK 为止。

所以，TCP 报文发出去后，并不会立马从内存中删除，因为重传时还需要用到它。

由于 TCP 是内核维护的，所以报文存放在内核缓冲区。如果连接非常多，我们可以通过 free 命令观察到 buff/cache 内存是会增大。

这样的传输方式有一个缺点：数据包的往返时间越长，通信的效率就越低。

要解决这一问题不难，并行批量发送报文，再批量确认报文即可。

然而，这引出了另一个问题，发送方可以随心所欲的发送报文吗？当然这不现实，我们还得考虑接收方的处理能力。

> 主动方的优化

主动发起 FIN 报文断开连接的一方，如果迟迟没收到对方的 ACK 回复，则会重传 FIN 报文，重传的次数由 `tcp_orphan_retries` 参数决定。

当主动方收到 ACK 报文后，连接就进入 FIN_WAIT2 状态，根据关闭的方式不同，优化的方式也不同：

- 如果这是 close 函数关闭的连接，那么它就是孤儿连接。如果 `tcp_fin_timeout` 秒内没有收到对方的 FIN 报文，连接就直接关闭。同时，为了应对孤儿连接占用太多的资源， `tcp_max_orphans` 定义了最大孤儿连接的数量，超过时连接就会直接释放。
- 反之是 shutdown 函数关闭的连接，则不受此参数限制；

当主动方接收到 FIN 报文，并返回 ACK 后，主动方的连接进入 TIME_WAIT 状态。这一状态会持续 1 分钟，为了防止 TIME_WAIT 状态占用太多的资源， `tcp_max_tw_buckets` 定义了最大数量，超过时连接也会直接释放。

当 TIME_WAIT 状态过多时，还可以通过设置 `tcp_tw_reuse` 和 `tcp_timestamps` 为 1 ，将 TIME_WAIT 状态的端口复用于作为客户端的新连接，注意该参数只适用于客户端。

Figure 7: 主动方优化

当接收方硬件不如发送方，或者系统繁忙、资源紧张时，是无法瞬间处理这么多报文的。于是，这些报文只能被丢掉，使得网络效率非常低。

为了解决这种现象发生，TCP 提供一种机制可以让「发送方」根据「接收方」的实际接收能力控制发送的数据量，这就是滑动窗口的由来。