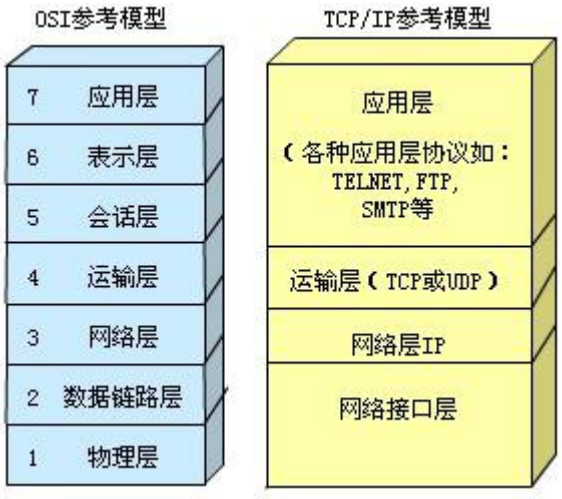
TCP 协议详解

TCP/IP 协议参考模型



TCP/IP 协议族按照层次由上到下

应用层：

向用户提供一组常用的应用程序，比如电子邮件、文件传输访问、远程登录等。远程登录 TELNET 使用 TELNET 协议提供在网络其它主机上注册的接口。TELNET 会话提供了基于字符的虚拟终端。文件传输访问 FTP 使用 FTP 协议来提供网络内机器间的文件拷贝功能。

传输层：

提供应用程序间的通信。其功能包括：一、格式化信息流；二、提供可靠传输。为实现后者，传输层协议规定接收端必须发回确认，并且假如分组丢失，必须重新发送。

网络层：

负责相邻计算机之间的通信。其功能包括三方面。

一、处理来自传输层的分组发送请求，收到请求后，将分组装入 IP 数据报，填充报头，选择去往信宿机的路径，然后将数据报发往适当的网络接口。

二、处理输入数据报：首先检查其合法性，然后进行寻径--假如该数据报已到达信宿机，则去掉报头，将剩下部分交给适当的传输协议；假如该数据报尚未到达信宿，则转发该数据报。

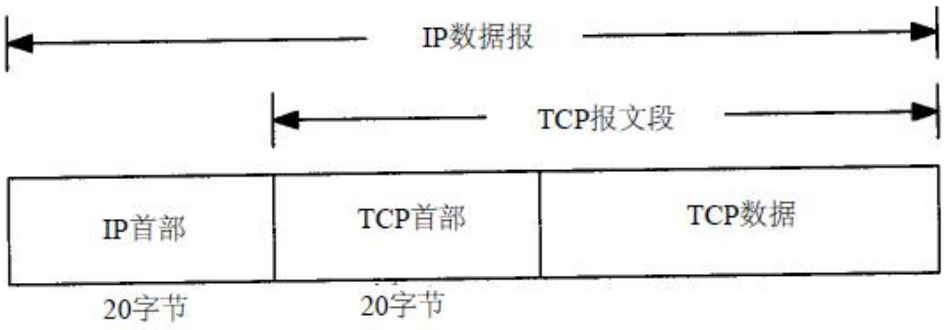
三、处理路径、流控、拥塞等问题。

网络接口层：

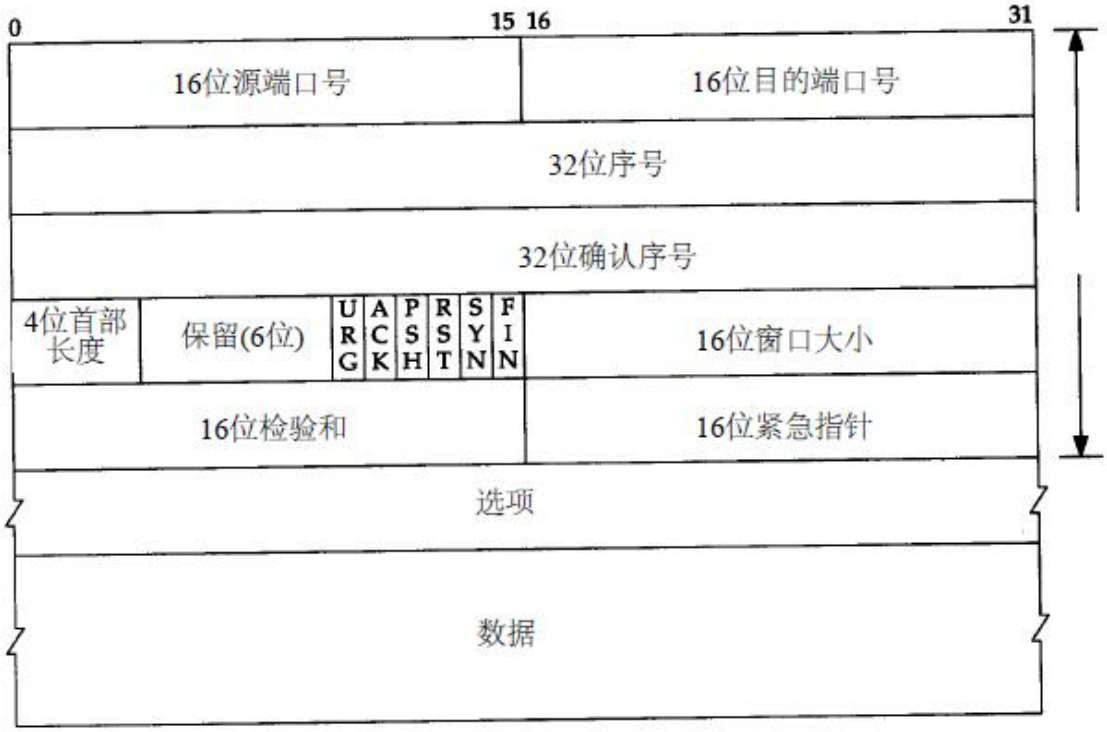
这是 TCP/IP 软件的最低层，负责接收 IP 数据报并通过网络发送之，或者从网络上接收物理帧，抽出 IP 数据报，交给 IP 层。

TCP 的首部格式

TCP 数据被封装在一个 IP 数据报中：



TCP 首部的数据格式。如果不计任何字段，它通常是 20 个字节：



一个 IP 地址和一个端口号也称为一个 socket，socket pair 可唯一确定互联网络中每个 TCP 连接的双方。

序号用来标识从 TCP 发端向 TCP 收端发送的数据字节流，它表示在这个报文段中的第一个数据字节。

当建立一个新的连接时，SYN 标志变为 1。

标志：

URG 紧急指针有效

ACK 确认序号有效

PSH 接收方应该尽快将这个报文段交给应用层

RST 重建连接

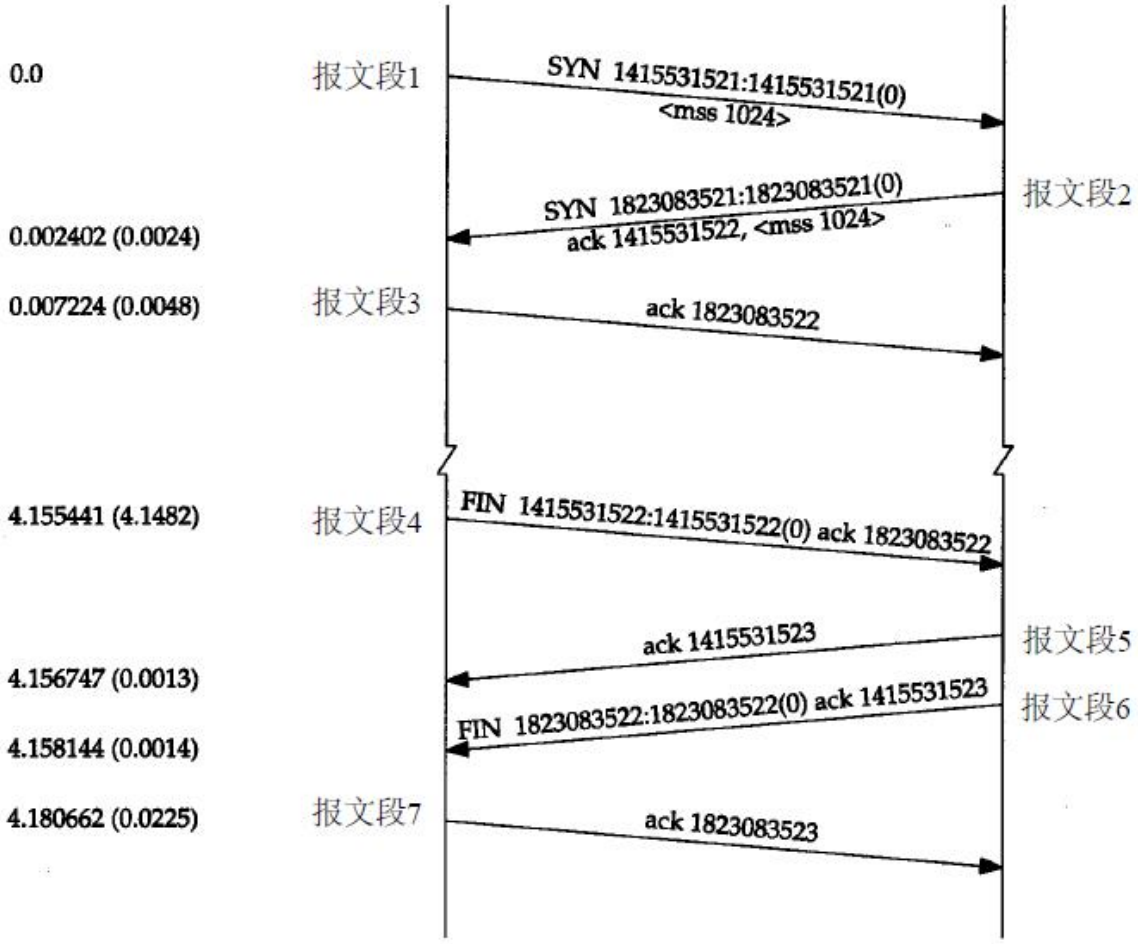
SYN 同步序号用来发起一个连接

FIN 发端完成发送任务

TCP 连接的建立和终止

三次握手

1 客户端发送一个 SYN 段指明客户打算连接的服务器端口，以及初始序号（ISN）
2 服务器发回包含服务器的初始序号的 SYN 报文段作为应答。同时，将确认序号设置为客户的 ISN+1 以对客户的 SYN 报文段进行确认。一个 SYN 将占用一个序号
3 客户端必须将确认序号设置为服务器的 ISN+1 以对服务器的 SYN 报文段进行确认



四次挥手

1 进行关闭的一方发送第一个 FIN
2 服务器收到 FIN，发回一个 ACK，确认序号为收到的序号+1。和 SYN 一样，一个 FIN 将占用一个序号。
3 同时 TCP 服务器还向应用程序传送一个文件结束符。接着这个服务器就关闭它的连接，导致它的 TCP 端发送一个 FIN。
4 客户必须发回一个 FIN，并将确认序号设置为收到序号+1。

最大报文长度

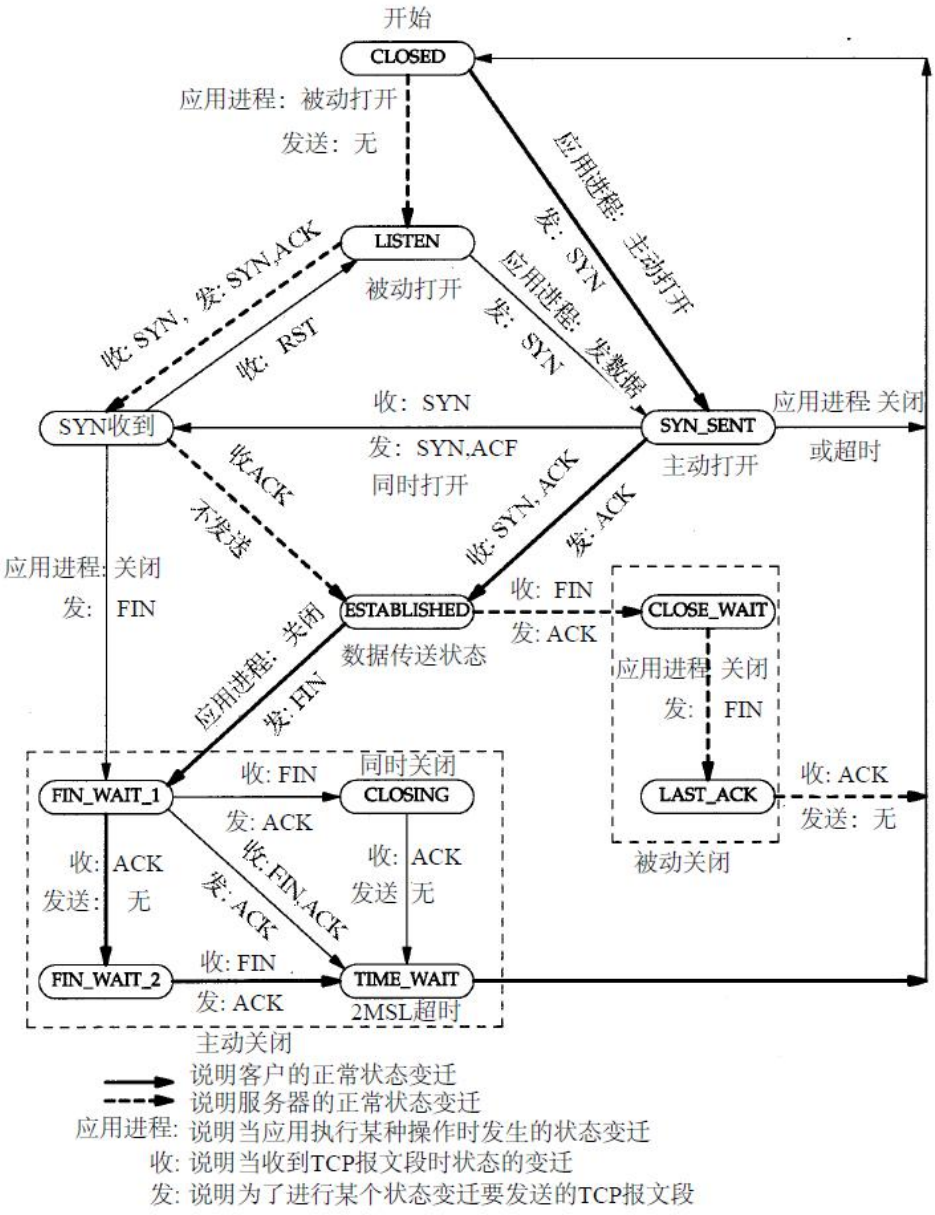
最大报文长度（MSS）表示 TCP 传往另一端的最大数据块的长度。
MSS 让主机限制另一端发送数据报的长度。加上主机也能控制它发送数据报的长度，这将使以较小 MTU 连接到一个网络上的主机避免分段。

TCP 的状态

TCP 的半关闭

TCP 提供了连接的一端在结束它的发送后还能接收来自另一端数据的能力，这就是所谓的半关闭。

TCP 的状态变迁图



2MSL 等待状态

TIME_WAIT 状态也称为 2MSL 等待状态。
对一个具体实现所给定的 MSL 值，处理的原则是：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 2 倍的 MSL。
这种 2MSL 等待的另一个结果是这个 TCP 连接在 2MSL 等待期间，定义这个连接的插口不能再被使用。这个连接只能在 2MSL 结束后才能再被使用。

平静时间的概念

TCP 在重启后的 MSL 秒内不能建立任何连接。

FIN_WAIT_2

只有当另一端的进程完成这个关闭，我们这端才会从 FIN_WAIT_2 状态进入 TIME_WAIT 状态。

复位报文段

TCP 首部中的 RST 是用于复位的。一般来说，无论何时一个报文段发往基准的连接出现错误，TCP 都会发出一个复位报文段。

到不存在的端口的连接请求

异常终止一个连接

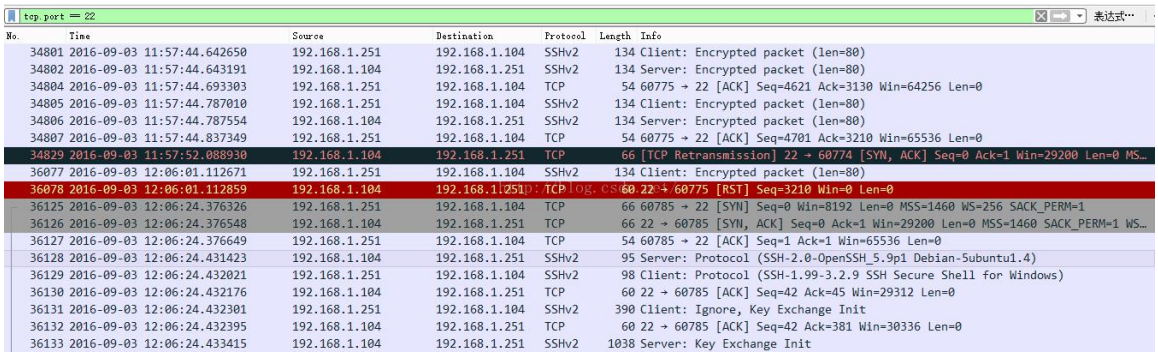
检测版打开连接

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！

以下是一个 Wireshark 抓包实例：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，两端的端口需要一致，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。

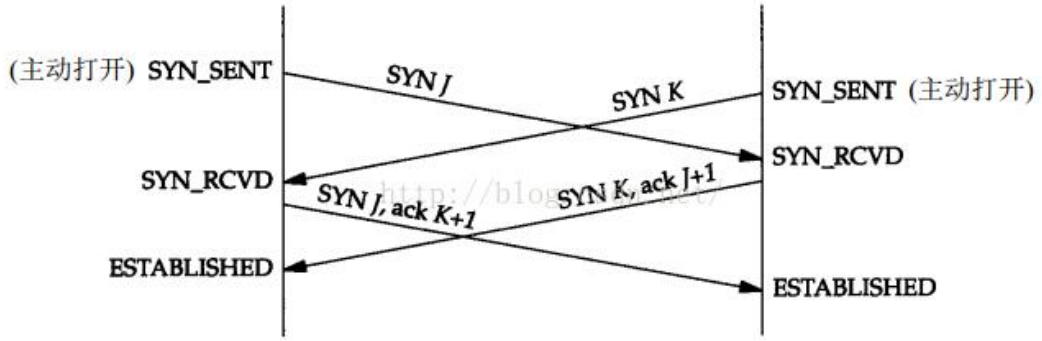
场景：

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

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

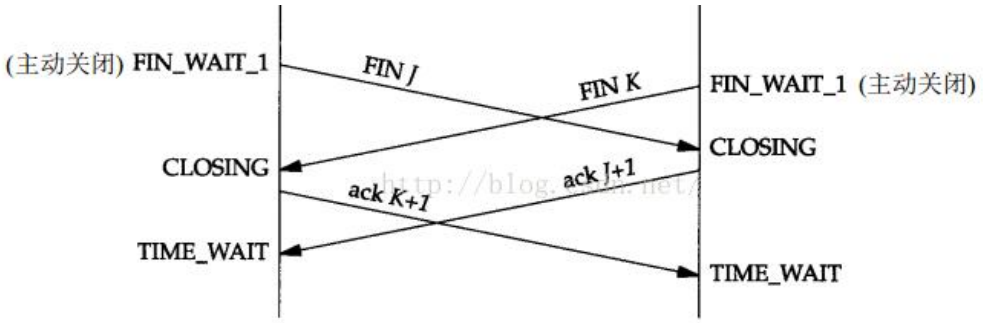
SYN 包同时打开对端，这种情况即为同时打开。

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！



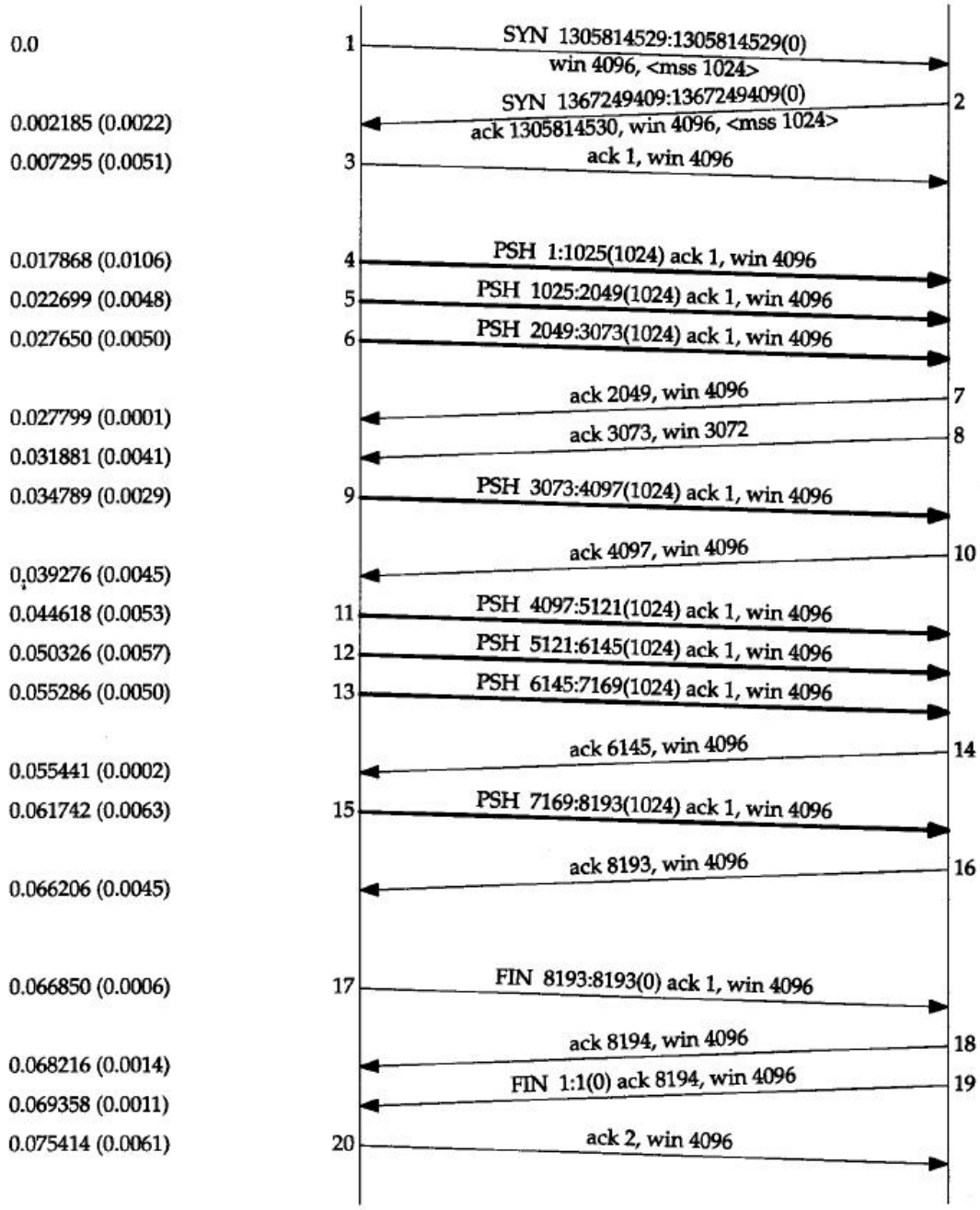
同时关闭

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 的成块数据流

TCP 使用滑动窗口协议来进行流量控制。该协议允许发送方在停止并等待确认前可以连续发送多个分组。由于发送方不必每发一个分组就停下来等待确认，因此该协议可以加速数据的传输。



在 TCP 协议当中窗口机制分为两种：

1. 固定窗口
2. 滑动窗口

固定窗口存在的问题

我们假设这个固定窗口的大小为 1，也就是每次只能发送一个数据，只有接收方对这个数据进行了确认后才能发送第二个数据。在图中我们可以看到，发送方每发送一个数据接收方就要给发送方一个 ACK 对这个数据进行确认。只有接收了这个确认数据以后发送方才能传输下个数据。

如果窗口过小，当传输比较大的数据的时候需要不停的对数据进行确认，这个时候就会造成很大的延迟。

如果窗口过大，我们假设发送方一次发送 100 个数据，但接收方只能处理 50 个数据，这样每次都只对这 50 个数据进行确认。发送方下一次还是发送 100 个数据，但接受方还是只能处理 50 个数据。这样就避免了不必要的数据来拥塞我们的链路。

因此，我们引入了滑动窗口。

滑动窗口（以字节为单位）

滑动窗口通俗来讲就是一种流量控制技术。

它本质上是描述接收方的 TCP 数据报缓冲区大小的数据，发送方根据这个数据来计算自己最多能发送多长的数据，如果发送方收到接收方的窗口大小为 0 的 TCP 数据报，那么发送方将停止发送数据，等到接收方发送窗口大小不为 0 的数据报的到来。

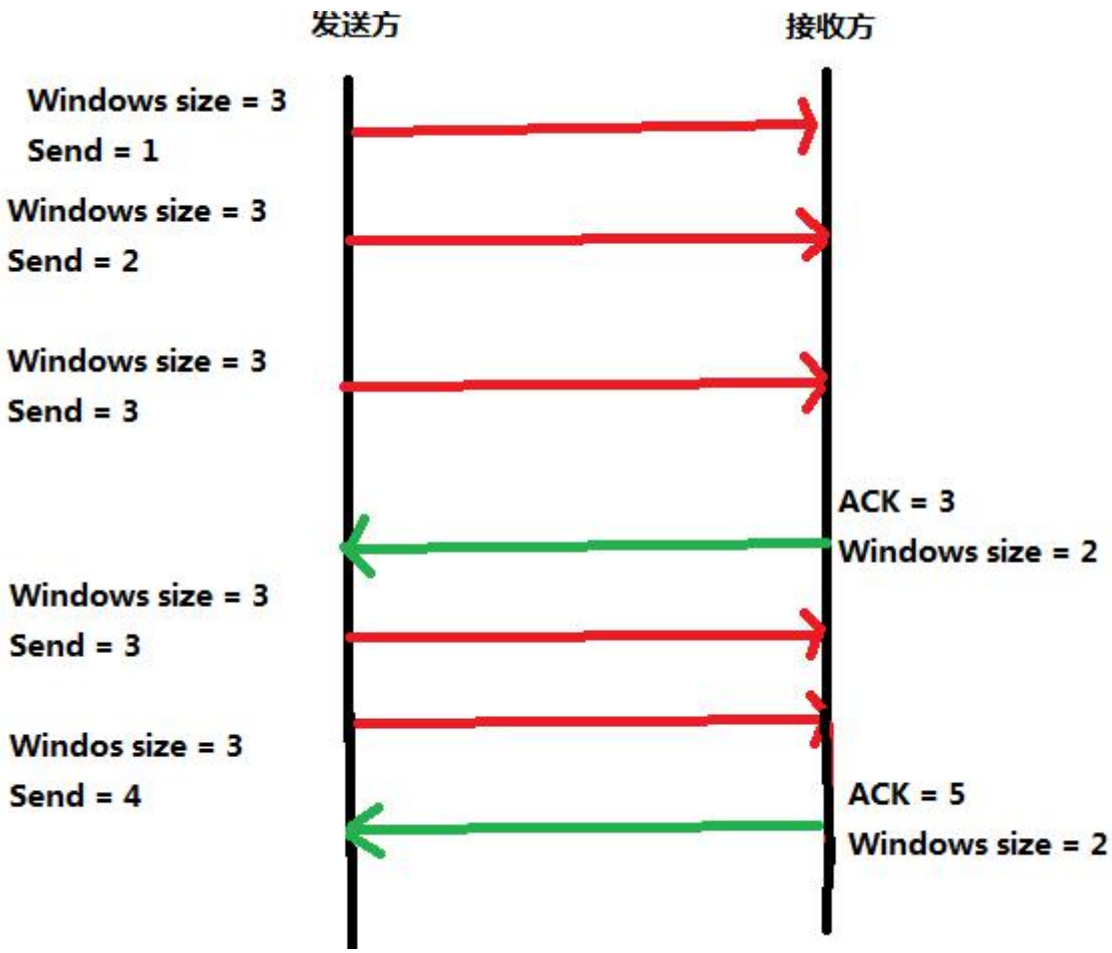
工作原理：

第一次发送数据这个时候的窗口大小是根据链路带宽的大小来决定的。

假设这时候的窗口是 3.这个时候接收方收到数据以后会对数据进行确认告诉哦发送方我下次希望收到的数据是多少。

在上图中：我们看到接收方发送的 ACK = 3（这是对发送方发送序列 2 的回答确认，下一次接收方期望接收到的是 3 序列信号），这个时候发送方收到这个数据以后就知道我第一次发送的 3 个数据对方只收到了两个，就知道第三个数据对方没有收到，下次返送的时候就从第 3 个数据开始发。这时候窗口大小就变为了 2。当链路变好或者变差，这个窗口还会发生变化。

如下图所示：



死锁状态：

当接收端向发送端发送零窗口报文段后不久，接收端的接收缓存又有了一些存储空间，于是接收端向发送端发送了 Windows size = 2 的报文段，然而这个报文段在传输过程中丢失了。发送端一直等待收到接收端发送的非零窗口的通知，而接收端一直等待发送端发送数据，这样就死锁了。

解决方法：

TCP 为每个连接设有一个持续计时器。只要 TCP 连接的一方收到对方的零窗口通知，就启动持续计时器，若持续计时器设置的时间到期，就发送一个零窗口探测报文段（仅携带 1 字节的数据），而对方就在确认这个探测报文段时给出了现在的窗口值。

TCP 报文段的发送时机（传输效率问题）

可以用以下三种不同的机制控制 TCP 报文段的发送时机：

（1）TCP 维持一个变量 MSS，等于最大报文段的长度。只要缓冲区存放的数据达到 MSS 字节时，就组装成了一个 TCP 报文段发送出去

（2）由发送方的应用进程指明要发送的报文段，即：TCP 支持推送操作

（3）发送方的一个计时器期限到了，这时就把当前已有的缓存数据装入报文段（但长度不能超过 MSS）发送出去。

Nagle 算法（控制 TCP 报文段的发送时机）

主旨：避免大量发送小包。

初衷：避免发送大量的小包，防止小包泛滥于网络，在理想情况下，对于一个 TCP 连接而言，网络上每次只能有一个小包存在。它更多的是端到端意义上的优化。

发送方将第一个数据字节发送出去，把后面到达的数据字节缓存起来。当发送方接收对第一个数据字符的确认后，再把发送缓存中的所有数据组装成一个报文段再发送出去，同时继续对随后到达的数据进行缓存。只有在收到对前一个报文段的确认之后，才继续发送下一个报文段。规定一个 TCP 连接最多只能有一个未被确认的未完成的小分组，在该分组的确认到达之前不能发送其它的小分组。当数据到达较快而网络速率较慢时，用这样的方法可以明显的减少所用的网络带宽。

Nagle 算法还规定：当达到的数据已经达到发送窗口大小的一半或者已经达到报文段的最大长度时，就可以立即发送一个报文段。

慢启动

TCP 在连接过程的三次握手完成后，开始传数据，并不是一开始向网络通道中发送大量的数据包，这样很容易导致网络中路由器缓存空间耗尽，从而发生拥塞；而是根据初始的 cwnd 大小逐步增加发送的数据量，cwnd 初始化为 1 个最大报文段(MSS)大小（这个值可配置不一定是 1 个 MSS）；每当有一个报文段被确认，cwnd 大小指数增长。
开始 —> cwnd = 1
1 个 RTT 后 —> cwnd = 2*1 = 2
2 个 RTT 后 —> cwnd = 2*2= 4
3 个 RTT 后 —> cwnd = 4*2 = 8

拥塞避免

cwnd 不能一直这样无限增长下去，一定需要某个限制。TCP 使用了一个叫慢启动门限(ssthresh)的变量，一旦 cwnd>=ssthresh（大多数 TCP 的实现，通常大小都是 65536），慢启动过程结束，拥塞避免阶段开始；
拥塞避免：cwnd 的值不再指数级往上升，开始加法增加。此时当窗口中所有的报文段都被确认时，cwnd 的大小加 1，cwnd 的值就随着 RTT 开始线性增加，这样就可以避免增长过快导致网络拥塞，慢慢的增加调整到网络的最佳值。
非 ECN 环境下的拥塞判断，发送方 RTO 超时，重传了一个报文段；

1，把 ssthresh 降低为 cwnd 值的一半；

2，把 cwnd 重新设置为 1；

3，重新进入慢启动过程。

TCP 的超时与重传

TCP 通过在发送时设置一个定时器来解决数据和确认丢失的问题。如果当定时器溢出时还没有收到确认，它就重传该数据。
对每个连接，TCP 管理 4 个不同的定时器：
1 重传定时器使用于当希望收到另一端的确认。
2 坚持（persist）定时器使窗口大小信息保持不断流动，即使另一端关闭了其接收窗口。
3 保活（keepalive)定时器可以检测到一个空闲连接的另一端何时崩溃或重启。
4 2MSL 定时器测量一个连接处于 TIME_WAIT 状态的时间。

快速重传

快速重传，TCP 在收到重复的 3 次 ACK 时，会认为重传队列中的第一个报文段被网络丢弃，但由于收到的重复的 3 次 ACK，则认为该报文段之后的三个报文已经被接收端收到，则不等待重传定时器超时，直接重发重传队列中的第一个报文段。

1，把 ssthresh 设置为 cwnd 的一半

2，把 cwnd 再设置为 ssthresh 的值(具体实现有些为 ssthresh+3)

3，重新进入拥塞避免阶段。