面向连接的运输：TCP

TCP连接

什么是 TCP 连接?

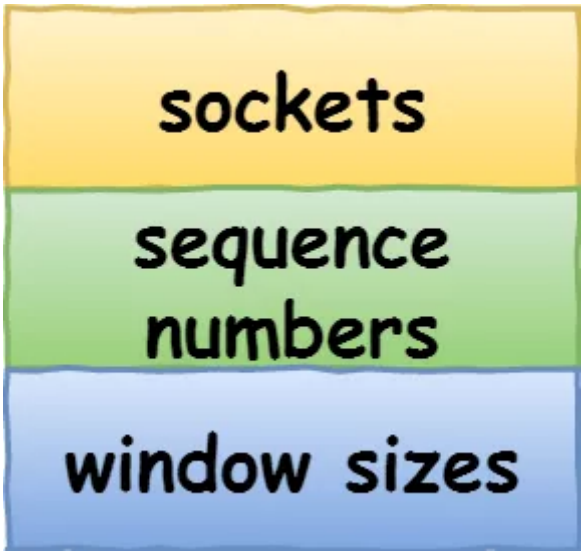
我们来看看 RFC 793 是如何定义「连接」的：

Connections:

The reliability and flow control mechanisms described above require that TCPs initialize and maintain certain status information for each data stream.

The combination of this information, including sockets, sequence numbers, and window sizes, is called a connection.

简单来说就是，**用于保证可靠性和流量控制维护的某些状态信息，这些信息的组合，包括Socket、序列号和窗口大小称为连接。**



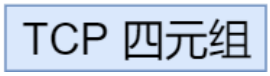
所以我们可以知道，建立一个 TCP 连接是需要客户端与服务器端达成上述三个信息的共识。

- **Socket**：由 IP 地址和端口号组成
- **序列号**：用来解决乱序问题等
- **窗口大小**：用来做流量控制

如何唯一确定一个 TCP 连接呢?

TCP 四元组可以唯一的确定一个连接，四元组包括如下：

- 源地址
- 源端口
- 目的地址
- 目的端口



TCP 四元组

源地址和目的地址的字段（32位）是在 IP 头部中，作用是通过 IP 协议发送报文给对方主机。

源端口和目的端口的字段（16位）是在 TCP 头部中，作用是告诉 TCP 协议应该把报文发给哪个进程。

TCP连接相关概念

TCP 被称为是**面向连接的**（ connection- oriented ），这是因为在一个应用进程可以开始向另一个应用进程发送数据之前，这两个进程必须先相互“握手”，即它们必须相互发送某些预备报文段，以建立确保数据传输的参数。 作为 TCP 连接建立的一部分，连接的双方都将初始化与 TCP 连接相关的许多 TCP 状态变量。

TCP 连接提供的是**全双工服务** (full-duplex serv ice ）： 如果一台主机上的进程 A 与另一台主机上的进程 B 存在一条 TCP 连接，那么应用层数据就可在从进程 B 流向进程 A 的同时，也从进程 A 流向进程 B 。 TCP 连接也总是点对点（ point-lo-point）的，即在单个发送方与单个接收方之间的连接。 所谓“多播”（参见 4. 7 节），即在一次发送操作中，从一个发送方将数据传送给多个接收方，对 TCP 来说这是不可能的 。

TCP通过**三次握手（three-way handshake）**建立连接。一旦建立起一条 TCP 连接，两个应用进程之间就可以相互发送数据了。客户进程通过套接字（该进程之门）传递数据流。 数据一旦通过该门，它就由客户中运行的 TCP 控制了。TCP 将这些数据引导到该连接的发送缓存（send buffer）里，发送缓存是在三次握手初期设置的缓存之一 。 接下来 TCP 就会不时从发送缓存里取出一块数据。TCP可从缓存中取出并放人报文段中的数据数量受限于最大报文段长度（ Maximum Segmenl Size, **MSS** ） 。 MSS 通常根据最初确定的由本地发送主机发送的最大链路层帧长度（即所谓的最大传输单元（ Maximum Transmission Unit, **MTU**）来设置 。

> **MTU**：即物理接口（**数据链路层**）提供给其上层（通常是IP层）最大一次传输数据的大小；以太网接口MTU=1500 Byte，这是以太网接口对IP层的约束，如果IP层有<=1500 byte 需要发送，只需要一个IP包就可以完成发送任务；如果IP层有> 1500 byte 数据需要发送，需要**分片**才能完成发送，这些分片有一个共同点，即**IP Header ID**相同。
>
> **MSS**：TCP提交给IP层最大分段大小，不包含TCP Header和 TCP Option，只包含TCP Payload(有效载荷)，MSS是TCP用来限制应用层最大的发送字节数。
>
> 如果底层物理接口MTU= 1500 byte，则 MSS = 1500- 20(IP Header) -20(TCP Header) = 1460 byte，如果应用层有2000 byte发送，需要两个segment才可以完成发送，第一个TCP segment = 1460 + 20，第二个TCP segment = 540 + 20

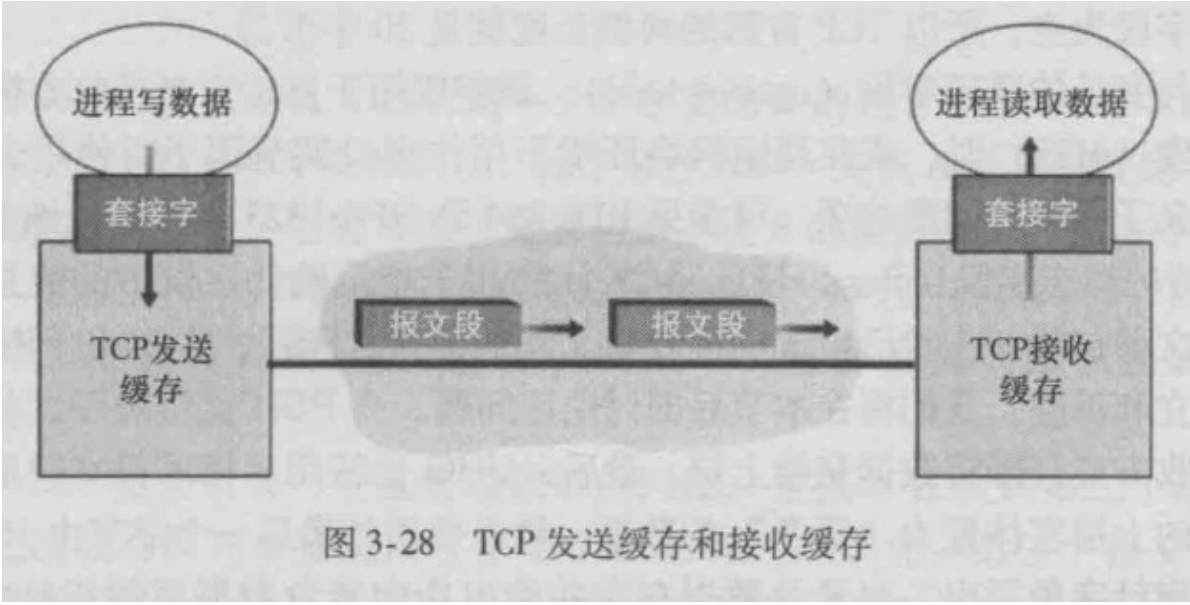


图 3-28　TCP 发送缓存和接收缓存

TCP 连接的组成包括：一台主机上的缓存、变量和与进程连接的套接字，以及另一台主机上的另一组缓存、变量和与进程连接的套接字。 如前面讲过的那样，在这两台主机之间的网络元素（路由器、交换机和中继器）中，没有为该连接分配任何缓存和变量。

TCP报文段结构

报文段结构详解

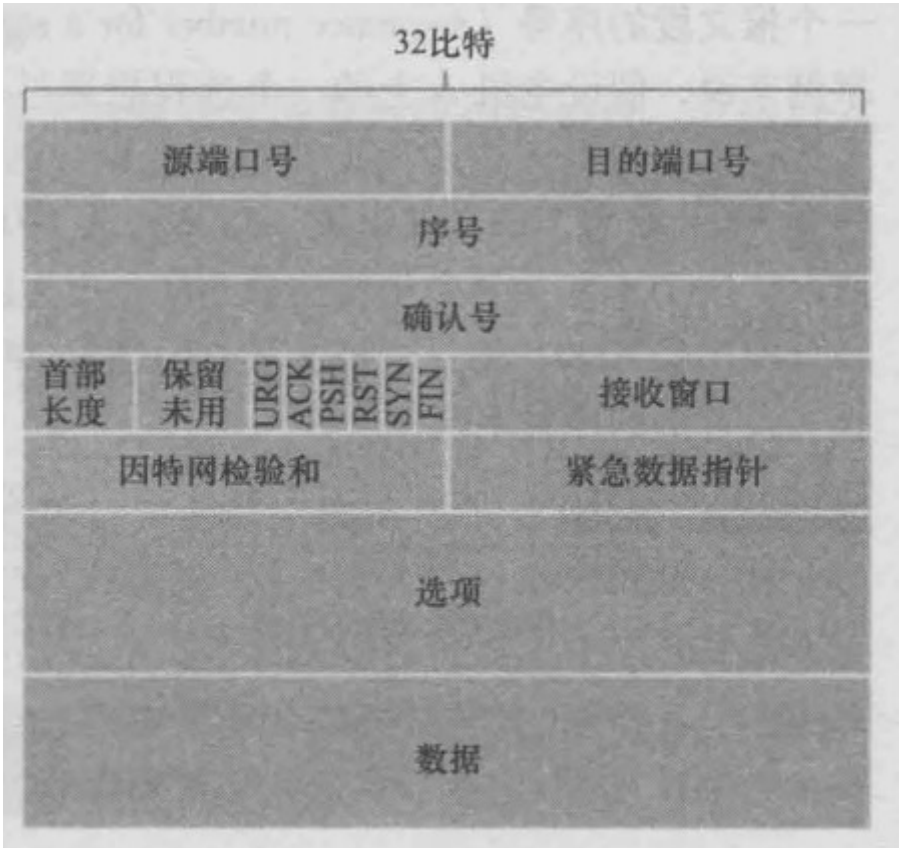


图 3-29　TCP 报文段结构

- **32 比特**的**序号字段**（ sequence number field ）
- **32 比特**的**确认号字段**（ acknowledgment number field ）。

> 序号字段和确认号字段被 TCP 发送方和接收方用来实现**可靠数据传输**。

- **4 比特**的**首部长度字段**（ header length field ），该字段指示了以 32 比特的字为单位的 TCP 首部长度 。 由于 TCP 选项字段的原因， TCP 首部的长度是可变的 。 （选项字段通常为空，所以 TCP 首部的典型长度就是 **20 字节。** ）
- **6 比特**的**标志字段**（flag field）。
  - ACK 比特用于指示确认字段中的值是有效的，即该报文段包括一个对已被成功接收报文段的确认。
  - RST、SYN 和 FIN 比特用于连接建立和拆除。
  - PSH 比特被设置的时候，就指示接收方应立即将数据交给上层。URG 比特用来指示报文段里存在着被发送端的上层实体置为“紧急”的数据。紧急数据的最后一个字节由 16 比特的紧急数据指针字段指出。 当紧急数据存在并给出指向紧急数据尾的指针的时候，TCP必须通知接收端的上层实体。（在实践中，PSH、URG 和紧急数据指针并没有使用。）
- **16 比特**的**接收窗口**字段（ receive window field ），该字段用于流量控制 。 该字段用于指示接收方愿意接受的字节数量。
- 可选与变长的**选项字段** （ options field ），该字段用于发送方与接收方协商最大报文段长度（ MSS ）或在高速网络环境下用作窗口调节因子时使用。

序号和确认号详解

序号

TCP 把数据看成一个无结构的、有序的**字节流** 。 我们从 TCP 对序号的使用上可以看出这一点，因为序号是建立在传送的字节流之上，而不是建立在传送的报文段的序列之上 。一个报文段的序号（ sequence number for a segment）因此是该**报文段首字节的字节流编号。**

> UDP面向报文，不会出现粘包问题。

确认号

TCP 只确认该流中至第一 个丢失字节为止的字节，所以 TCP 被称为提供累积确认（ cumulative acknowledgment ） 。

可靠数据传输（差错控制）

TCP 在IP不可靠的尽力而为服务之上创建了 一种**可靠数据传输服务。**TCP 的可靠数据传输服务确保一个进程从其接收缓存中读出的数据流是无损坏、无间隔、非冗余和按序的数据流；即该字节流与连接的另一方端系统发送出的字节流是完全相同 。

往返时间的估计与超时间隔

估计往返时间

RTT : Roung Trip TIme 往返时间

TCP 维持一个 SampleRTT 均值（称为 EstimatedRTT) 。 一旦获得一个新SampleRTT 时，TCP 就会根据下列公式来更新 EstimaLedRTT:

$$EstimatedRTT = (1-\alpha)\cdot EstimatedRIT + \alpha \cdot SampleRTT$$

在［RFC 6298 J 中给出的α 参考值是α= $0.125(1/8)$，这时上面的公式变为 :

$$EstimatedRTT = 0.875\cdot EstimatedRIT + 0.125 \cdot SampleRTT$$

EstimatedRTT是一个 SampleRTT 值的加权平均值。这个加权平均对最近的样本赋予的权值要大于对老样本赋予的权值。 这是很自然的，因为越近的样本越能更好地反映网络的当前拥塞情况 。从统计学观点讲，这种平均被称为**指数加权移动平均** （ Exponential Weighted Moving Average, EWMA ） 。在EWMA中的“指数”一词看起来是指一个给定的 SampleRTT 的权值在更新的过程中呈指数型快速衰减。

除了估算 RTT 外，测量 RTT的变化也是有价值的 。 [RFC 6298] 定义了 RTT 偏差DevRπ ，用于估算SampleRTT一般会偏离 EstimatedRTT的程度：
$DevRTT = (1-\beta) \cdot DevRTT + \beta \cdot |SampleRTT - EstimatedRTT|$
注意到 DevRTT 是一个 SampleRTT 与 EstimatedRTT 之间差值的 EWMA。 如果SampleRTT值波动较小，那么 DevRTT 的值就会很小 ； 另一方面，如果波动很大，那么 DevRTT 的值就会很大。 β 的推荐值为 0.25。

设置和管理重传超时间隔

假设已经给出了 EstimatedRTT值和 DevRTT 值，那么 TCP 超时间隔应该用什么值呢?
很明显，超时间隔应该大于等于 EstimatedRTI，否则，将造成不必要的重传。 但是超时间隔也不应该比 EstimatedRIT 大太多，否则当报文段丢失时， TCP 不能很快地重传该报文段，导致数据传输时延大。 因此要求将超时间隔设为 EstimaledRTT 加上一定余量 。 当SampleRTT 值波动较大时，这个余量应该大些；当波动较小时，这个余量应该小些 。 因此， DevRTT 值应该在这里发挥作用了 。 在 TCP 的确定重传超时间隔的方法中，所有这些因素都考虑到了：

$$TimeoutInterval = EstimatedRTT + 4 \cdot DevRTT$$

推荐的初始 TimeoutInterval 值为 1 秒。同样，当出现超时后TimeoutInterval 值将加倍，以免即将被确认的后继报文段过早出现超时。不管怎样，一旦报文段收到并更新EstimatedR TT 后， TimeoutInterval 就又使用上述公式计算了。

三个主要事件

TCP 发送方有 3 个与发送和重传有关的主要事件：从上层应用程序接收数据；定时器届时和收到 ACK。

- 从上层应用程序接收数据
  - TCP 从应用程序接收数据，将数据封装在一个报文段中，并把该报文段交给 IP 。

    > 注意到每一个报文段都包含一个序号，这个序号就是该报文段第一个数据字节的字节流编号。

  - 还要注意到如果定时器还没有为某些其他报文段而运行，则当报文段被传给IP时， TCP 就启动该定时器 。

    > 定时器与最早的未被确认的报文段相关联。
    >
    > 该定时器的过期间隔是 Timeoutlnlerval ， 由 EstimatedRTI 和 DevRTT 计算得出的 。

- 超时
  - TCP 通过重传引起超时的报文段来响应超时事件 。 然后 TCP重启定时器。
- 来自接收方的包含了有效 ACK 字段值的报文段的到达
  - 当该事件发生时， TCP 将 ACK的值 y 与它的变量 SendBase 进行比较 。

    > TCP 状态变量 SendBase 是最早未被确认的字节的。
    >
    > SendBase -1 是指接收方已正确按序接收到的数据的最后一个字节的序号。

  - 如果 y >SendBase ， 则该 ACK 是在确认一个或多个先前未被确认的报文段。 因此发送方更新它的 SendBase 变量；

    > TCP 采用**累积确认**，所以 y 确认了字节编号在 y 前的所有字节都已经收到。

  - 如果当前有未被确认的报文段，TCP 还要重新启动定时器 。

超时间隔加倍

TCP 重传具有最小序号的还未被确认的报文段。只是每次 TCP 重传时都会将下一次的超时间隔设为先前值的两倍，而不是用从 EstimatedRTT和 DevRTT推算出的值 。

超时间隔在每次重传后会呈指数型增长。 然而，每当定时器在另两个事件（即收到上层应用的数据和收到 ACK ）中的任意一个启动时， TimeoutInterval由最近的 EstimatedRTT 值与 DevRTT 值推算得到 。

> 这种修改提供了一个形式受限的**拥塞控制** 。定时器过期很可能是由网络拥塞引起的，即太多的分组到达源与目的地之间路径上的一台（或多台）路由器的队列中，造成分组丢失或长时间的排队时延 在拥塞的时候，如果源持续重传分组，会使拥塞更严重 。 相反， TCP 使用更文雅的方式，每个发送方的重传都是经过越来越长的时间间隔后进行的 。

快速重传

超时触发重传存在的问题之一是***超时周期可能相对较长*** 。 当一个报文段丢失时，这种长超时周期迫使发送方延迟重传丢失的分组，因而增加了端到端时延。发送方通常可在超时事件发生之前通过注意所谓冗余 ACK 来较好地检测到丢包情况 。 **冗余 ACK**(duplicate ACK ）就是再次确认某个报文段的 ACK ，而发送方先前已经收到对读报文段的确认。

如果 TCP 发送方接收到对相同数据的 3 个冗余 ACK ， 它把这当作一种指示，说明跟在这个已被确认过 3 次的报文段之后的报文段已经丢失 。一旦收到 3 个冗余 ACK, TCP 就执行**快速重传** （ fast retransmit ） ，即在该报文段的定时器过期之前重传丢失的报文段 。

回退N步还是选择重传

TCP 确认是累积式的，正确接收但失序的报文段是不会被接收方逐个确认的 。TCP 发送方仅需维持已发送过但未被确认的字节的**最小序号(SendBase）**和**下一个要发送的字节的序号（NextSeqNum）** 。

TCP 采用**选择确认** （ selective acknowledgment)，它允许 TCP 接收方有选择地确认失序报文段，而不是累积地确认最后一个正确接收的有序报文段 。 当将该机制与**选择重传**机制结合起来使用时（即跳过重传那些已被接收方选择性地确认过的报文段） , TCP 看起来就很像我们通常的 SR 协议 。 因此，TCP 的差错恢复机制也许最好被分类为 GBN 协议与 SR 协议的混合体。

流量控制

接收窗口

TCP 为它的应用程序提供了**流量控制服务** （ flow- control service ）以消除发送方便接收方**缓存溢出**的可能性。流量控制因此是一个**速度匹配**服务，即发送方的发送速率与接收方应用程序的读取速率相匹配。

- 接收方缓存溢出 ----> 流量控制服务(flow- control)
- IP 网络的拥塞 -------> 拥塞控制(congestion control)

TCP 通过让发送方维护一个称为**接收窗口** （ receive window ）的变量来提供流量控制 。

> 接收窗口用于给发送方一个指示——该**接收方还有多少可用的缓存空间**。
>
> 因为TCP 是全双工通信，在连接两端的**发送方都各自维护一个接收窗口**。

假设主机 A 通过一条 TCP 连接向主机 B 发送一个大文件 主机 B 为该连接分配了一个接收缓存，并用 RcvBuffer 来表示其大小 。 主机 B 上的应用进程不时地从该缓存中读取数据。 定义以下变量：

- LastByteRead：主机 B 上的应用进程从缓存读出的数据流的最后一个字节的编号 。
- LastByteRcvd：从网络中到达的并且已放入主机 B 接收缓存巾的数据流的最后一个字节的编号。

由于 TCP 不允许已分配的缓存溢出，下式必须成立：

$$LaslByleRcvd - LastByteRead <= RevBuffer$$

接收窗口用 rwnd 表示，根据缓存可用空间的数量来设置：

$$rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]$$

由于该空间是随着时间变化的，所以rwnd 是动态的，如下图。

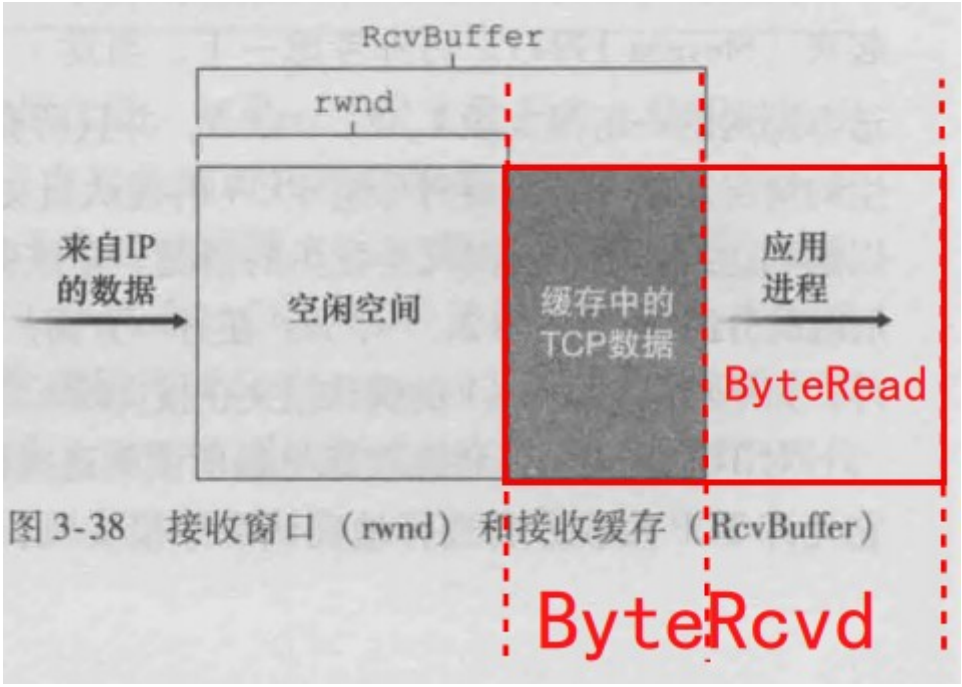


图 3-38　接收窗口（rwnd）和接收缓存（RcvBuffer）

连接是如何使用变量 rwnd 提供流量控制服务的呢？主机 B 通过把当前的rwnd值放人它发给主机 A 的报文段接收窗口字段中，通知主机 A 它在原连接的缓存中还有多少可用空间 。 开始时，主机 B设定 rwnd = RevBuffer。为了实现一点，主机 B 必须跟踪几个与连接有关的变量LastByteRcvd、LastByteRead等等。
主机 A 轮流跟踪两个变量， LastByteSent 和 LastByteAcked。
这两个变量之间的差 LastByteSent - LastByteAcked ，就是主机 A 发送到连接中但未被确认的数据量。通过将未确认的数据盖控制在值 rwnd 以内，就可以保证主机 A 不会使主机 B 的接收缓存溢出 。 因此，主机 A 在该连接的整个生命周期须保证：

$$LastByteSent - LastByteAcked <= rwnd$$

可能的问题

> 可能的问题

假设主机 B 的接收缓存已经存满，使得 rwnd =0 。 在将 rwnd =0 遇告给主机 A 之后，还要假设主机 B 没有任何数据要发给主机 A。 此时，考虑会发生什么情况 。 因为主机 B 上的应用进程将缓存清空，TCP 并不向主机 A 发送带有 rwnd 新值的新报文段；事实上， TCP 仅当在它有数据或有确认要发时才会发送报文段给主机 A 。 这样，主机 A 不可能知道主机 B 的接收缓存已经有新的空间了，即主机 A 被阻塞而不能再发送数据!

> 解决办法

为了解决这个问题， TCP 规范中要求：当主机 B 的接收窗口为 0 时，主机 A 继续发送只有一个字节数据的报文段 。 这些报文段将会被接收方确认 。 最终缓存将开始清空，并且确认报文里将包含一个非 0 的同时值。

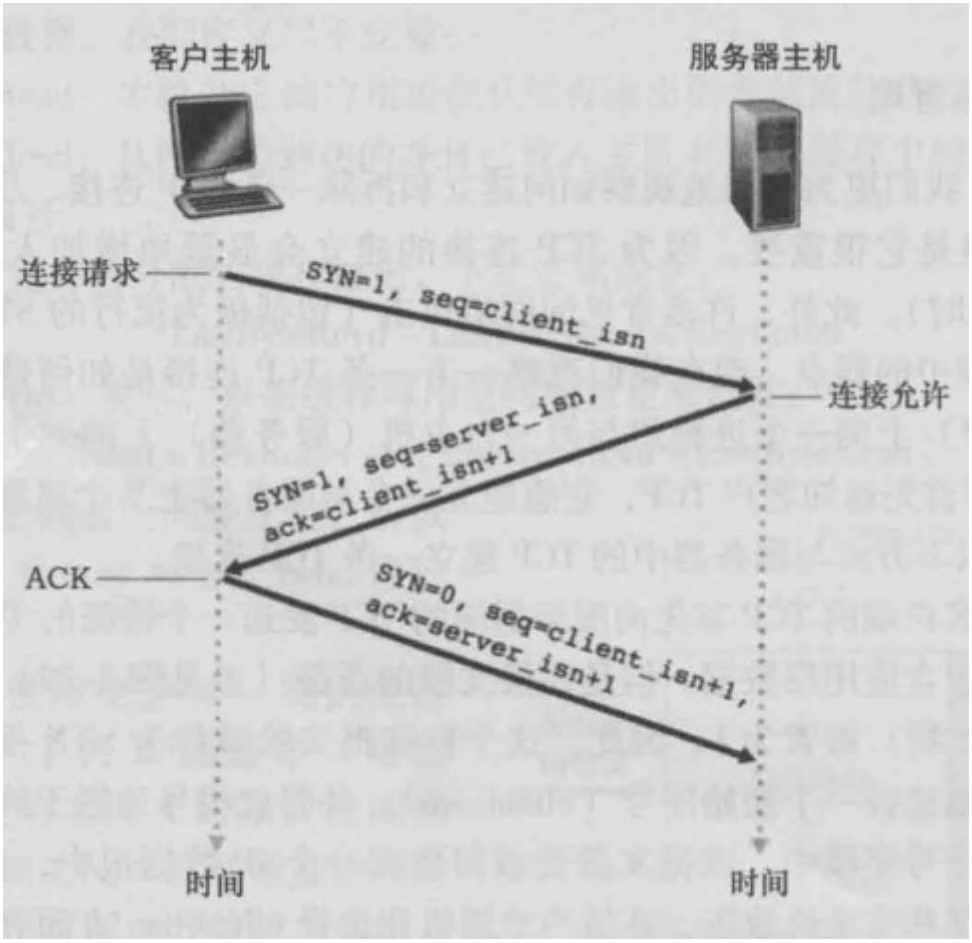
UDP无流量控制

描述了 TCP 的流量控制服务以后，我们在此要简要地提一下 UDP 并不提供流量控制 。
为了理解这个问题，考虑一下从主机 A 上的一个进程向主机 B 上的一个进程发送一系列UDP 报文段的情形 。 对于一个典型的 UDP 实现， UDP 将会把这些报文段添加到相应套接字（进程的门户）“前面”的一个有限大小的缓存中 。 进程每次从缓存巾读取一个完整的报文段。 如果进程从缓存中读取报文段的速度不够快，那么缓存将会溢出，并且将丢失报文段。

TCP连接管理

三次握手全过程

- 客户端的 TCP 首先向服务器端的 TCP 发送一个特殊的 TCP 报文段 。 该报文段中**不包含应用层数据** 。 在报文段的首部中的一个**标志位（SYN 比特）被置为 1** 。 因此，这个特殊报文段被称为 **SYN 报文段** 。 另外，客户会**随机地选择一个初始序号**（client_isn ），并将此编号放置于该起始的 TCP SYN报文段的**序号字段**中 。 该报文段会被封装在一个 IP 数据报中，并发送给服务器 。
- 第二步：一旦包含 TCP SYN 报文段的 IP 数据报到达服务器主机（假定它的确到达了！），服务器会从该数据报中提取出 TCP SYN 报文段，为该 TCP 连接**分配 TCP 缓存和变量**，并向该客户 TCP 发送允许连接的报文段。 （*在完成三次握手的第三步之前分配这些缓存和变量，使得 TCP 易于受到称为 SYN 洪泛的拒绝服务攻击*。 ）这个允许连接的报文段**也不包含应用层数据** 。 但是，在报文段的首部却包含 3 个重要的信息 。 首先， **SYN 比特被置为1**。其次，该 TCP 报文段首部的**确认号字段被置为 client_isn + 1** 。 最后，服务器选择自己的**初始序号(server_isn)** ，并将其放置到 TCP 报文段首部的**序号字段**中 。 这个允许连接的报文段实际上表明了：“我收到了你发起建立连接的 SYN 分组，该分组带有初始序号client_isn 。 我同意建立该连接 。 我自己的初始序号是 server_isn 。 ”该允许连接的报文段有时被称为 **SYNACK 报文段** （ SYN + ACK segment ） 。
- 第三步：在收到 SYNACK 报文段后，客户也要给该连接**分配缓存和变量** 。 客户主机则向服务器发送另外一个报文段：这最后一个报文段对服务器的允许连接的报文段进行了确认（该客户通过将值 **server_isn + 1** 放置到 TCP 报文段首部的**确认字段**中来完成此项工作）。因为连接已经建立了，所以该 **SYN比特被置为 0** 。 该三次握手的第三个阶段可以在报文段负载中**携带客户到服务器的数据。**
- 一旦完成这 3 个步骤，客户和服务器主机就可以相互发送包括数据的报文段了 。 在以后每一个报文段中， SYN 比特都将被置为 0。 注意到为了创建该连接，在两台主机之间发送了 3 个分组。 由于这个原因， 这种连接创建过程通常被称为 3 次握手(three- way handshake ） 。

- 

四次挥手全过程

客户应用进程发出一个关闭连接命令。 这会引起客户 TCP 向服务器进程发送一个特殊的TCP 报文段 。这个特殊的报文段让其首部中的 一 个标志位即 **FIN 比特被设置为 1** 。 当服务器接收到该报文段后，就向发送方回送 - 个**确认报文段。** 然后，服务器发送它自己的终止报文段，其 **FIN 比特被置为1。**最后，关闭该客户对服务器的终止报文段进行**确认** 。 此时，在两台主机上用于该连接的所有资源都被释放了。

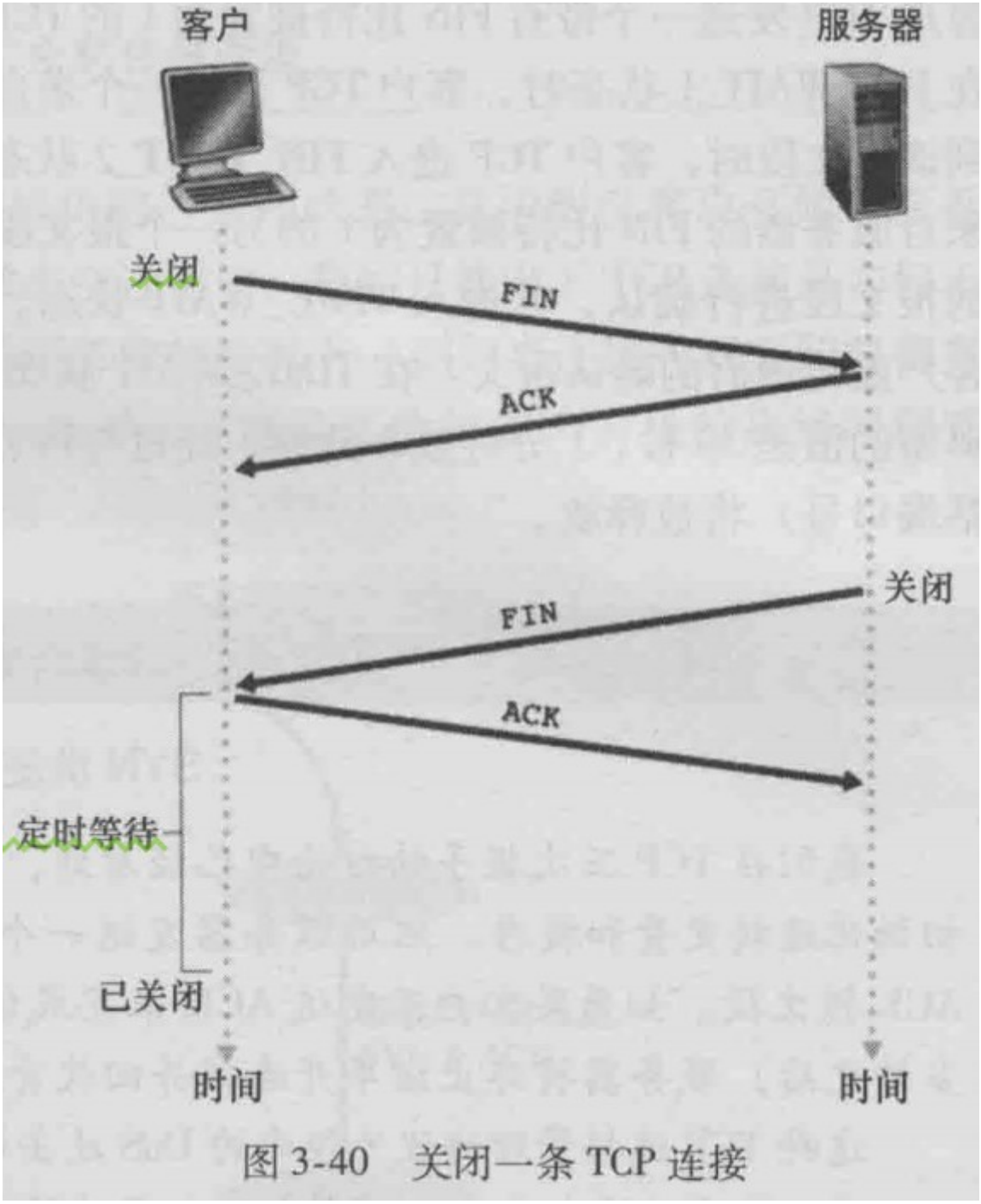


图 3-40　关闭一条 TCP 连接

客户端与服务端状态转移

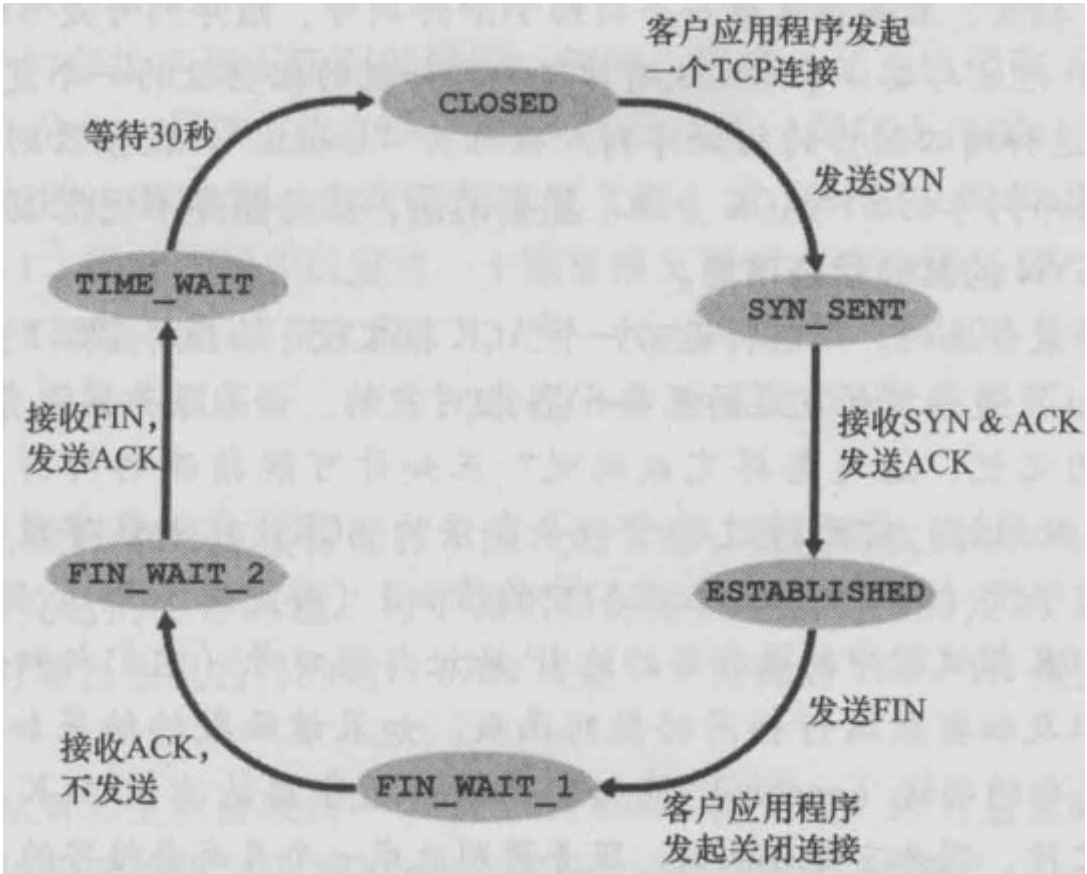


图 3-41　客户 TCP 经历的典型的 TCP 状态序列

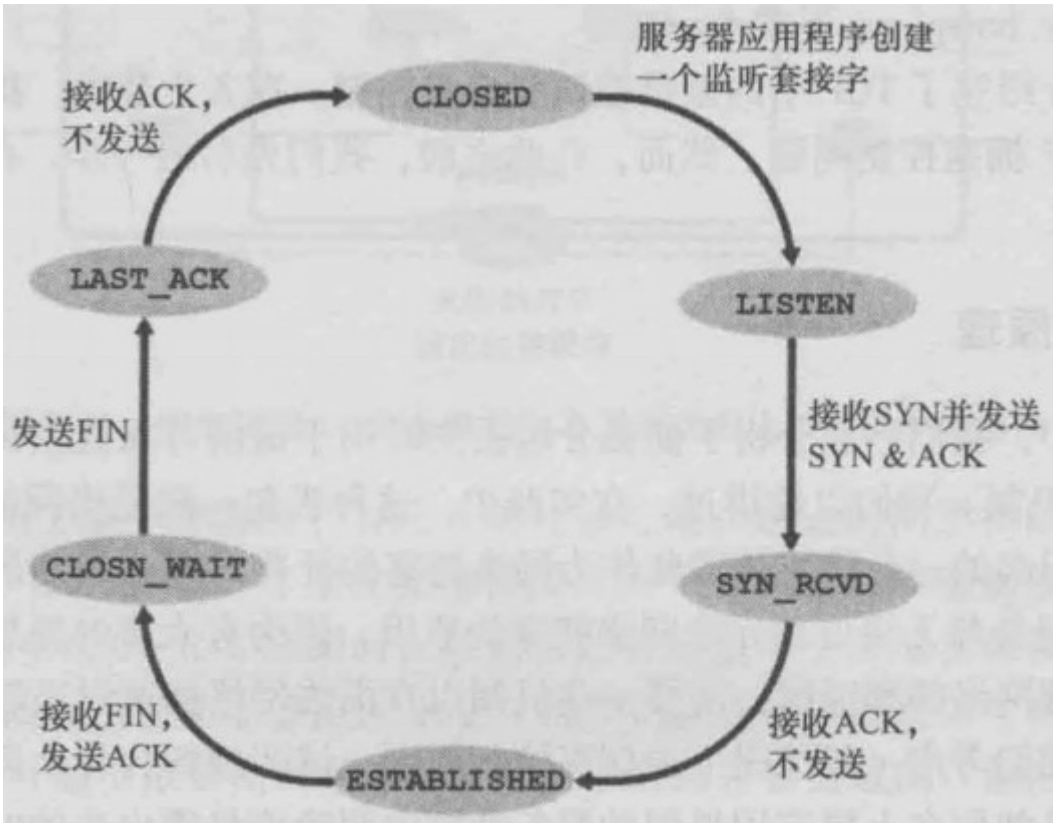


图 3-42　服务器端 TCP 经历的典型的 TCP 状态序列

SYN洪泛攻击

SYN 洪泛攻击

我们在 TCP 三次握手的讨论中已经看到，服务器为了响应一个收到的 SYN，分配并初始化连接变量和缓存。然后服务器发送一个 SYNACK 进行响应，并等待来自客户的 ACK 报文段。如果某客户不发送 ACK 来完成该三次握手的第三步，最终（通常在一分多钟之后）服务器将终止该半开连接并回收资源。

这种 TCP 连接管理协议为经典的 DoS 攻击即 **SYN 洪泛攻击**（SYN flood attack）提供了环境。在这种攻击中，攻击者发送大量的 TCP SYN 报文段，而不完成第三次握手的步骤。随着这种 SYN 报文段纷至沓来，服务器不断为这些半开连接分配资源（但从未使用），导致服务器的连接资源被消耗殆尽。这种 SYN 洪泛攻击是被记载的众多 DoS 攻击中的第一种［CERT SYN 1996］。幸运的是，现在有一种有效的防御系统，称为 **SYN cookie**［RFC 4987］，它们被部署在大多数主流操作系统中。SYN cookie 以下列方式工作：

- 当服务器接收到一个 SYN 报文段时，它并不知道该报文段是来自一个合法的用户，还是一个 SYN 洪泛攻击的一部分。因此服务器不会为该报文段生成一个半开连接。相反，服务器生成一个初始 TCP 序列号，该序列号是 SYN 报文段的源和目的 IP 地址与端口号以及仅有该服务器知道的秘密数的一个复杂函数（散列函数）。这种精心制作的初始序列号被称为“cookie”。服务器则发送具有这种特殊初始序列号的 SYNACK 分组。**重要的是，服务器并不记忆该 cookie 或任何对应于 SYN 的其他状态信息。**
- 如果客户是合法的，则它将返回一个 ACK 报文段。当服务器收到该 ACK，需要验证该 ACK 是与前面发送的某些 SYN 相对应的。如果服务器没有维护有关 SYN 报文段的记忆，这是怎样完成的呢？正如你可能猜测的那样，它是借助于 cookie 来做到的。前面讲过对于一个合法的 ACK，在确认字段中的值等于在 SYNACK 字段（此时为 cookie 值）中的值加 1（参见图 3-39）。服务器则将使用在 SYNACK 报文段中的源和目的地 IP 地址与端口号（它们与初始的 SYN 中的相同）以及秘密数运行相同的散列函数。如果该函数的结果加 1 与在客户的 SYNACK 中的确认（cookie）值相同的话，服务器认为该 ACK 对应于较早的 SYN 报文段，因此它是合法的。服务器则生成一个具有套接字的全开的连接。
- 在另一方面，如果客户没有返回一个 ACK 报文段，则初始的 SYN 并没有对服务器产生危害，因为服务器没有为它分配任何资源。

特殊情况

> 我们来考虑**当一台主机接收到一个 TCP 报文段，其端口号或源 IP 地址与该主机上进行中的套接字都不匹配**的情况 。 例如，假如一台主机接收了具有目的端口 80的一个 TCP SYN 分组，但该主机在端口 80 不接受连接（即它不在端口 80 上运行 Web 服务器） 。 则该主机将向源发送一个特殊重置报文段 。 该 TCP 报文段将 RST 标志位置为 1 。 因此，当主机发送一个重置报文段时，它告诉该源“我没有那个报文段的套接字 。 请不要再发送该报文段了” 。 当一台主机接收一个 UDP 分组，它的目的端口与进行中的 UDP 套接字不匹配，该主机发送一个特殊的 ICMP 数据报。

TCP拥塞控制

现在是我们考虑广受赞誉的 TCP 拥塞控制算法（ TCP congestion control algorithm ）细节的时候了，该算法包括 3 个主要部分： ①慢启动；②拥塞避免；③快速恢复。慢启动和拥塞避免是 TCP 的强制部分，两者的差异在于对收到的 ACK 做出反应时增加cwnd 长度的方式 。 我们很快将会看到慢启动比拥塞避免能更快地增加 cwnd 的长度。快速恢复是推荐部分，对 TCP 发送方并非是必需的 。

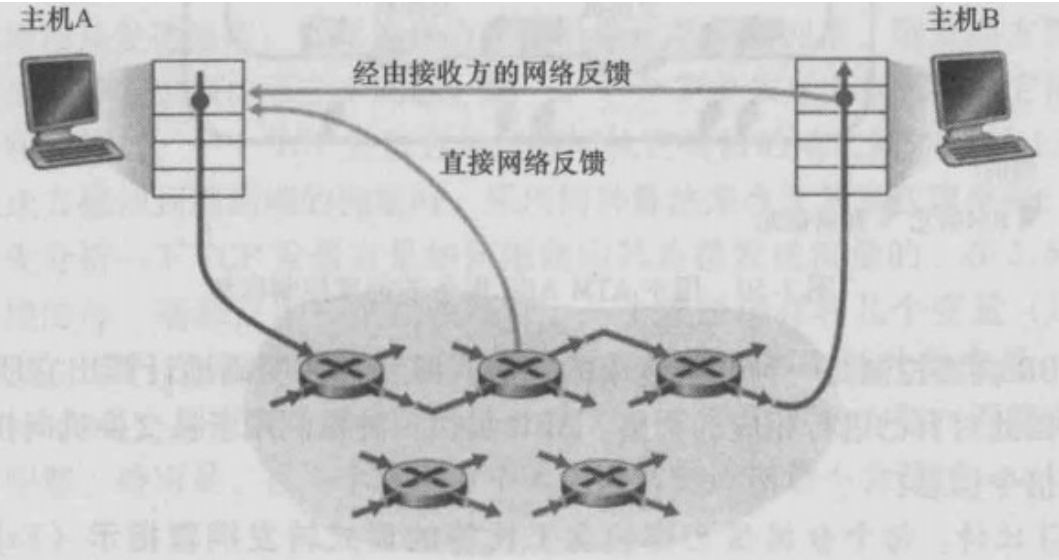
拥塞控制原理

拥塞原因与代价

- 当分组的到达速率接近链路容量时，分组经历巨大的排队时延。
- 发送方在遇到大时延时所进行的不必要重传会引起路由器利用其链路带宽来转发不必要的分组副本。
- 当一个分组沿一条路径被丢弃时，每个上游路由器用于转发该分组到丢弃该分组而使用的传输容量最终被浪费掉了。

拥塞控制方法

- 端到端拥塞控制 。 在端到端拥塞控制方法中，网络层没有为运输层拥塞控制提供显式支持。 即使网络中存在拥塞，端系统也必须通过对网络行为的观察（如分组丢失与时延）来推断之。TCP 必须通过端到端的方法解决拥塞控制，因为 IP 层不会向端系统提供有关网络拥塞的反馈信息。TCP 报文段的丢失（通过超时或 3 次冗余确认而得知）被认为是网络拥塞的一个迹象，TCP 会相应地减小其窗口长度 。 我们还将看到关于 TCP 拥塞控制的一些最新建议，即使用增加的往返时延值作为网络拥塞程度增加的指示 。
- 网络辅助的拥塞控制 。 在网络辅助的拥塞控制中，网络层构件（即路由器）向发送方提供关于网络中拥塞状态的显式反馈信息 。 这种反馈可以简单地用一个比特来指示链路中的拥塞情况。 该方法在早期的 IBM SNA和 DEC DECnel等体系结构中被采用，近来被建议用于TCP/IP 网络，而且还用在我们下面要讨论的 ATM可用比特率（ ABR ）拥塞控制中 。 更复杂的网络反馈也是可能的 。 例如，我们很快将学习的一种 ATM ABR 拥塞控制形式，它允许路由器显式地通知发送方，告知它（路由器）能在输出链路上支持的传输速率。 关于源端是增加还是降低其传输速率， XCP 协议对每个源提供了路由器计算的反馈，该反馈携带在分组首部中。

> 对于网络辅助的拥塞控制，拥塞信息从网络反馈到发送方通常有两种方式 。 直接反馈信息可以由网络路由器发给发送方 。 这种方式的通知通常采用了一种阻塞分组（ choke packet）的形式（主要是说：“我拥塞了！”）。第二种形式的通知是，路由器标记或更新从发送方流向接收方的分组中的某个字段来指示拥塞的产生 。 一旦收到一个标记的分组后，接收方就会向发送方通知该网络拥塞指示。注意到后一种形式的通知至少要经过一个完整的往返时间 。
>
> 
>
> 图 3-49　网络指示拥塞信息的两种反馈路径

拥塞控制的三个问题

TCP 必须使用端到端拥塞控制而不是使网络辅助的拥塞控制，因为 IP 层不向端系统提供显式的网络拥塞反馈 。TCP 所采用的方法是让每一个发送方根据所感知到的网络拥塞程度来限制其能向连接发送流量的速率。 如果一个 TCP 发送方感知从它到目的地之间的路径上没什么拥塞，则TCP 发送方增加其发送速率；如果发送方感知沿着该路径有拥塞，则发送方就会降低其发送速率。但是这种方法提出了三个问题。

- 一个 TCP 发送方如何限制它向其连接发送流量的速率呢?
  - 运行在发送方的 TCP 拥塞控制机制跟踪一个额外的变量，即**拥塞窗口**(congestion window ）。拥塞窗口表示为 cwnd，它对一个 TCP 发送方能向网络中发送流量的速率进行了限制 。特别是，在一个发送方中未被确认的数据量不会超过 cwnd 与 rwnd 中的最小值，即
  - $LastByteSent - LastByteAcked <= min\{cwnd, rwnd\}$
  - 因此粗略地讲，在每个往返时间（ RTT）的起始点，上面的限制条件允许发送方向该连接发送cwnd个字节的数据，在该 RTT 结束时发送方接收对数据的确认报文。 因此，**该发送方的发送速率大概是**$cwnd/RTT$ **字节 / 秒 。 通过调节 cwnd 的位，发送方因此能调整它向连接发送数据的这率 。**
- 一个 TCP 发送方如何感知从它到目的地之间的路径上存在拥塞或者没有拥塞呢?
  - 将一个 TCP 发送方的"**丢包事件**"定义为： 要么出现**超时**，要么收到来自接收方的 **3个冗余 ACK** 。当出现过度的拥塞时，在沿着这条路径上的一台（或多台）路由器的缓存会溢出，引起一个数据报（包含一个 TCP 报文段）被丢弃。 丢弃的数据报接着会引起发送方的丢包事件（要么超时或收到 3 个冗余 ACK），发送方就认为在发送方到接收方的路径上出现了拥塞的指示 。
  - TCP 的发送方将收到对于以前未确认报文段的确认作为一切正常的指示，即在网络上传输的报文段正被成功地交付给目的地，并使用确认来增加窗口的长度（及其传输速率） 。 注意到如果确认以相当慢的速率到达，则该拥塞窗口将以相当慢的速率增加 。 在另一方面，如果确认以高速率到达，则该拥塞窗口将会更为迅速地增大。 因为 TCP 使用**确认来触发（或计时）增大它的拥塞窗口长度**， TCP 被说成是 **自计时** （ self- clocking）的 。
- 当发送方感知到端到端的拥塞时，采用何种算法来改变其发送速率呢?
  - 拥塞控制算法的指导原则
  - **丢失的报文段表意味着拥塞**：因此当丢失报文段时应当降低 TCP 发送方的速率 。 对于给定报文段，一个超时事件或四个确认（一个初始 ACK 和其后的三个冗余 ACK ）被解释为跟随该四个 ACK 的报文段的"丢包事件"的一种隐含的指示 。 从拥塞控制的观点看，该问题是 TCP 发送方应当如何减小它的拥塞窗口长度，即减小其发送速率，以应对这种推测的丢包事件。
  - '**确认报文段指示一切顺利**：该网络正在向接收方交付发送方的报文段，因此，当对先前未确认报文段的确认到达时，能够增加发送方的速率 。 确认的到达被认为是一切顺利的隐含指示，即报文段正从发送方成功地交付给接收方，因此该网络不拥塞。拥塞窗口长度因此能够增加 。
  - **带宽探测**：给定 ACK 指示源到目的地路径无拥塞，而丢包事件指示路径拥塞，TCP 调节其传输速率的策略是增加其速率以响应到达的 ACK ， 除非出现丢包事件，此时才减小传输速率。因此，为探测拥塞开始出现的速率， TCP 发送方增加它的传输速率，从该速率后退，进而再次开始探测，看看拥塞开始速率是否发生了变化。

慢启动

如何进行

当一条 TCP 连接开始时， **cwnd 的值通常初始置为一个 MSS 的较小值。**

- 这就使得初始发送速率大约为 MSS/RTT。 由于对 TCP 发送方而言，可用带宽可能比 MSS/RTT 大得多， TCP 发送方希望迅速找到可用带宽的数量 。

- 因此，**在慢启动（slow- strat）状态， cwnd 的值以 1 个 MSS 开始并且每当传输的报文段首次被确认就增加1 个 MSS 。**

> 在图所示的例子中， TCP 向网络发送第一个报文段并等待一个确认。 当该确认到达时， TCP 发送方将拥塞窗口增加一个 MSS ，并发送出两个最大长度的报文段 。 这两个报文段被确认，则发送方对每个确认报文段将拥塞窗口增加一个 MSS ，使得拥塞窗口变为 4 个 MSS ，并这样下去 。 这一过程每过一个 RTT，发送速率就翻番 。

- 因此，TCP 发送速率**起始慢**，但在慢启动阶段以**指数增长。**

何时结束?

- 如果存在一个由超时指示的**丢包事件（即拥塞）**，TCP 发送方将第二个状态变量的值 ssthresh（慢启动阈值）设置为$cwnd/2$，即拥塞窗口值的一半，然后将 cwnd 设置为1并重新开始**慢启动**过程。。
- 因为当检测到拥塞时 ssthresh 设为 cwnd 的值一半，当到达或超过ssthresh 的值时，继续使 cwnd 翻番可能有些鲁莽 。 因此，当 cwnd 的值等于 ssthresh 时，结束慢启动并且 TCP 转移到**拥塞避免**模式。 我们将会看到，当进入拥塞避免模式时，TCP 更为谨慎地增加 cwnd 。
- 如果检测到 **3 个冗余 ACK**,这时 TCP 执行一种**快速重传**并进人**快速恢复**状态 。 慢启动中的 TCP 行为总结在图 3-52 中的 TCP 拥塞控制的 FSM 描述中 。

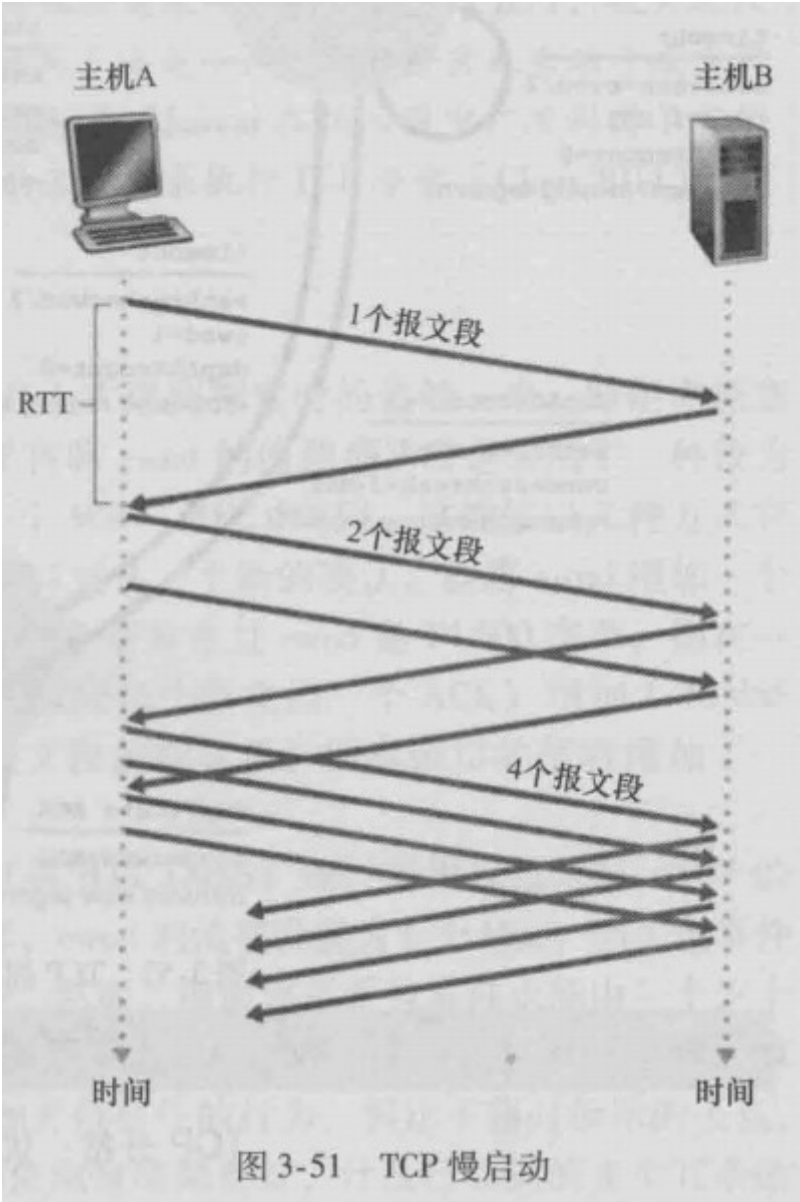


图 3-51　TCP 慢启动

拥塞避免

如何进行

一旦进入拥塞避免状态， cwnd 的值大约是上次遇到拥塞时的值的一半，距离拥塞可能并不遥远！因此， TCP 无法每过一个 RTT 再将 cwnd 的值翻番，而是采用了一种较为保守的方法，**每个 RTT 只将 cwnd 的值增加一个 MSS 。**通用的方法是对于 TCP 发送方无论何时到达**一个新的确认**，就**将 cwnd 增加一个MSS* (MSS/cwnd)** 字节 。

> 如果 MSS 是 1460 字节并且 cwnd 是 14 600 字节，则在一个 RTT 内发送 10 个报文段 。 每个到达 ACK （假定每个报文段一个 ACK）增加 $1/10$ MSS的拥塞窗口长度，因此在收到对所有 10 个报文段的确认后，拥塞窗口的值将增加了一个 MSS 。

何时结束

- 当出现**超时**时， TCP 的拥塞避免算法行为相同 。 与慢启动的情况一样， ssthresh 的值被更新为 cwnd 值的一半，cwnd 的值被设置为 1 个 MSS。
- **三个冗余 ACK 事件**： 在这种情况下，网络继续从发送方向接收方交付报文段（就像由收到冗余 ACK 所指示的那样）。 TCP 对这种丢包事件的行为，相比于超时指示的丢包，应当不那么剧烈 。 TCP 将 **ssthresh 的值记录为 cwnd 的值的一半，将 cwnd 的值减半**，接下来**快速重传**并进入**快速恢复**状态。

快速恢复

在快速恢复中，对于引起 TCP 进入快速恢复状态的缺失报文段，对收到的每个冗余的ACK, cwnd 的值增加一个 MSS。最终，当对丢失报文段的一个 ACK 到达时， TCP 在降低cwnd 后进入拥塞避免状态 。 如果出现超时事件，快速恢复在执行如同在慢启动和拥塞避免中相同的动作后，迁移到慢启动状态：当丢包事件出现时， cwnd 的值被设置为 1 个MSS ， 并且 sslhresh 的值设置为 cwnd 值的一半 。
快速恢复是 TCP 推荐的而非必需的构件。 有趣的是，一种称为 TCP Tahoe 的 TCP 早期版本，不管是发生超时指示的丢包事件，还是发生 3 个冗余 ACK 指示的丢包事件，都无条件地将其拥塞窗口减至 1 个 MSS ， 并进入慢启动阶段。 TCP 的较新版本TCP Reno ， 则综合了快速恢复。
下图展示了 Reno 版 TCP 与 Tahoe 版 TCP 的拥塞控制窗口的演化情况。在该图中，阈值初始等于 8 个 MSS 。 在前 8 个传输回合， Tahoe 和 Reno 采取了相同的动作 。 拥塞窗口在慢启动阶段以指数速度快速爬升，在第 4 轮传输时到达了阔值。 然后拥塞窗口以线性速度爬升，直到在第 8 轮传输后出现 3 个冗余 ACK。注意到当该丢包事件发生时，拥塞窗口值为 12 × MSS 。 于是 ssthresh被设置为 $0.5 * cwnd = 6 * MSS$。在TCP Reno下，拥塞窗口被设置为 $cwnd = 0.5 * cwnd + 3 * MESS = 9 * MESS$，然后线性地增长 。 在 TCP Tahoe 下，拥塞窗口被设置为 1 个MSS ， 然后呈指数增长，直至到达 ssthresh 值为止，在这个点它开始线性增长 。

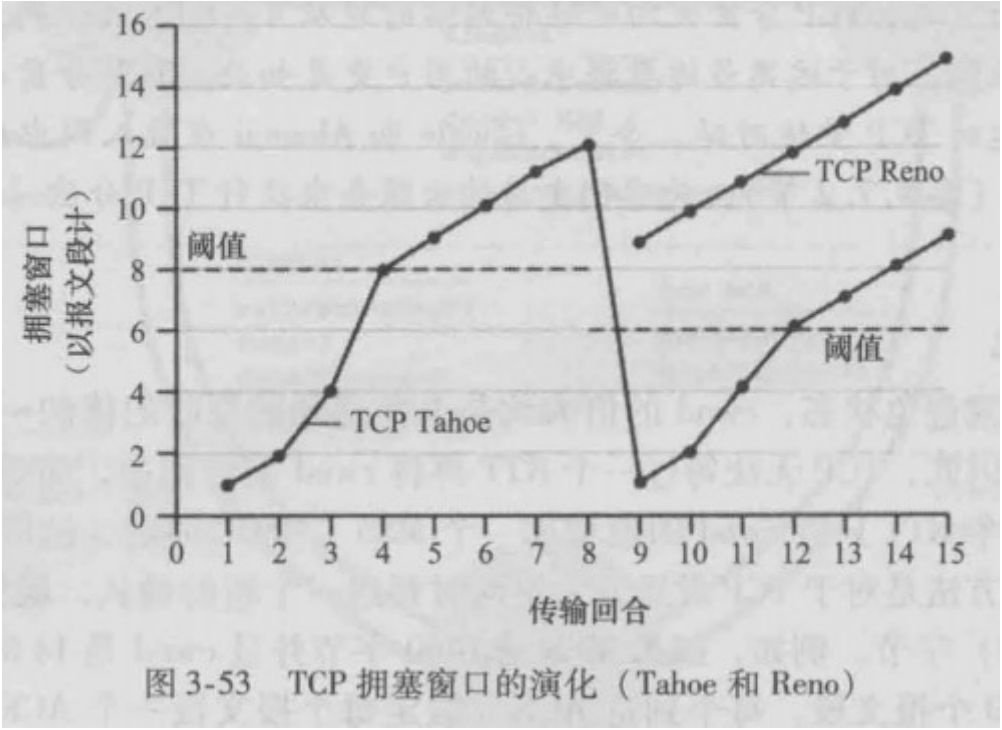
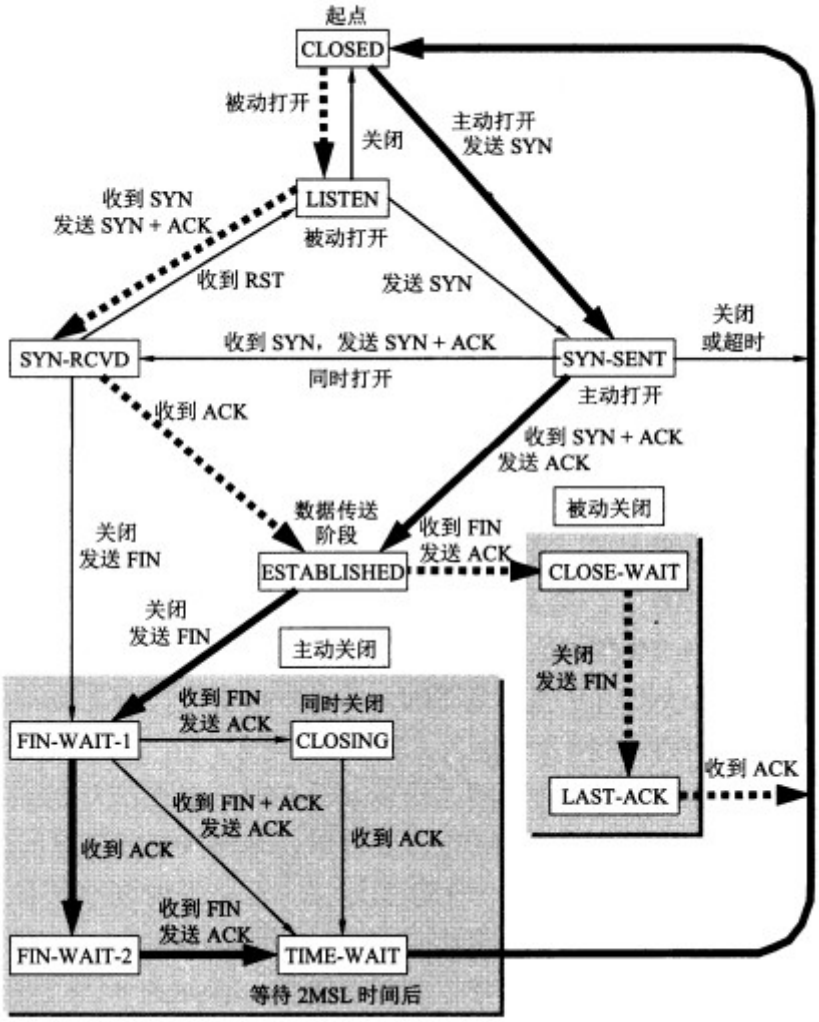


图 3-53　TCP 拥塞窗口的演化（Tahoe 和 Reno）

TCP有限状态机



常考问题

为什么 UDP 头部没有「首部长度」字段，而 TCP 头部有「首部长度」字段呢?

原因是 TCP 有**可变长**的「选项」字段，而 UDP 头部长度则是**不会变化**的，无需多一个字段去记录 UDP 的首部长度。

为什么 UDP 头部有「包长度」字段，而 TCP 头部则没有「包长度」字段呢?

先说说 TCP 是如何计算负载数据长度：

```
TCP 数据的长度 = IP 总长度 - IP 首部长度 - TCP 首部长度
```

其中 IP 总长度 和 IP 首部长度，在 IP 首部格式是已知的。TCP 首部长度，则是在 TCP 首部格式已知的，所以就可以求得 TCP 数据的长度。

大家这时就奇怪了问：“ UDP 也是基于 IP 层的呀，那 UDP 的数据长度也可以通过这个公式计算呀？为何还要有「包长度」呢？”

这么一问，确实感觉 UDP 「包长度」是冗余的。

因为为了网络设备硬件设计和处理方便，首部长度需要是 `4` 字节的整数倍。

如果去掉 UDP 「包长度」字段，那 UDP 首部长度就不是 `4` 字节的整数倍了，所以小林觉得这可能是为了补全 UDP 首部长度是 `4` 字节的整数倍，才补充了「包长度」字段。

Linux 系统中如何查看 TCP 状态?

TCP 的连接状态查看，在 Linux 可以通过 `netstat -napt` 命令查看。

TCP 连接状态查看

有一个 IP 的服务器监听了一个端口，它的 TCP 的最大连接数是多少?

服务器通常固定在某个本地端口上监听，等待客户端的连接请求。

因此，客户端 IP 和 端口是可变的，其理论值计算公式如下:

```
最大 TCP 连接数 = 客户端的IP 数 X 客户端的端口数
```

对 IPv4，客户端的 IP 数最多为 `2` 的 `32` 次方，客户端的端口数最多为 `2` 的 `16` 次方，也就是服务端单机最大 TCP 连接数，约为 `2` 的 `48` 次方。

当然，服务端最大并发 TCP 连接数远不能达到理论上限。

- 首先主要是**文件描述符限制**，Socket 都是文件，所以首先要通过 `ulimit` 配置文件描述符的数目；
- 另一个是**内存限制**，每个 TCP 连接都要占用一定内存，操作系统是有限的。

TCP如何实现可靠传输

- 序号机制（序号、确认号）：确保了数据是按序、完整到达。tcp会按最大传输单元(MTU)合理分片，接收方会缓存未按序到达的数据，重新排序后交给应用层。而UDP：IP数据报大于1500字节，大于MTU。这个时候发送方的IP层就需要分片，把数据报分成若干片，是的每一片都小于MTU。而接收方IP层则需要进行数据报的重组。由于UDP的特性，某一片数据丢失时，接收方便无法重组数据报，导致丢弃整个UDP数据报。
- 建立连接（标志位）：通信前先建立全双工的连接，确认通信实体存在。
- 流量控制（接收窗口）：当接收方来不及处理发送方的数据，能通过滑动窗口，提示发送方降低发送的速率，防止包丢失。
- 拥塞控制（拥塞窗口）：当网络拥塞时，通过拥塞窗口，减少数据的发送，防止包丢失。
- 数据校验（校验和）：TCP报文头有CRC校验和，用于校验报文是否损坏。
- 超时重传（定时器）：保证因链路故障未能到达的数据能够被多次重发。

如何基于UDP实现可靠传输

我们首先参考下TCP是如何实现可靠传输的。见上。我们只要在应用层实现TCP的这些机制就可以保证可靠。校验和不需要实现，因为UDP是有校验和的。但我们的目标可不仅仅是可靠。我们要超时重传，那什么时候重传呢，重传什么呢，直接照抄TCP吗？而且TCP的流量控制与拥塞控制可是饱受诟病的。我们如果把TCP重新实现一遍，那干嘛不直接用TCP呢，使用全世界程序员贡献的TCP代码肯定比我们自己写的更高效。

金融领域有一个概念 **不可能三角**（Impossible trinity），意为在金融政策方面，资本自由流动、固定汇率和货币政策独立性三者不可能兼得。在我看来网络传输也有不可能三角，**实时性/吞吐量**，**可靠性**，**公平性**是不可能兼得的。UDP为了实时性，抛弃了可靠性和公平性。TCP为了可靠性和公平性牺牲了实时性。UDP和TCP全世界广泛应用的今天，大量网络带宽被UDP占用，而TCP由于内置的拥塞控制、流量控制机制而被迫为UDP让路。这形成了一种“劣币驱逐良币的局面”。公平的TCP协议应用得越来越少，不公平的UDP协议应用得越来越多。当然，大多数公司会自己基于UDP实现一套协议，不是纯正的UDP。但不可否认的是，网络越来越不公平了。

我们基于UDP实现可靠传输的时候也要考虑这个问题。实时性、可靠性、公平性，这三点可靠性是我们必须的，但是另外两点该如何抉择呢。我觉得这要根据我们具体的产品来决定。比如我们要做一套实时通信工具，那么实时性和可靠性就是必须的，公平性只能为我们的目标让路。在这种情况下，我们实现的协议大概会抛弃拥塞控制与流量控制机制，并且超时重传的间隔也会减少。比如我们要做网盘，在用户上传文件和下载文件的时候，对于普通用户就加上流量控制和拥塞控制，限制些网速，也为网络的公平贡献自己的一份力。对于VIP客户嘛，流量控制和拥塞控制还是要去除的，毕竟网盘上传下载的网速是非常关键的，直接决定了一个网盘的生死。

TCP 与 UDP 的区别

1. TCP 面向连接，UDP 是无连接的；
2. TCP 提供可靠的服务，也就是说，通过 TCP 连接传送的数据，无差错，不丢失，不重复，且按序到达；UDP 尽最大努力交付，即不保证可靠交付
3. TCP 的逻辑通信信道是全双工的可靠信道；UDP 则是不可靠信道
4. 每一条 TCP 连接只能是点到点的；UDP 支持一对一，一对多，多对一和多对多的交互通信
5. TCP 面向字节流（可能出现黏包问题），实际上是 TCP 把数据看成一连串无结构的字节流；UDP 是面向报文的（不会出现黏包问题）
6. UDP 没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如 IP 电话，实时视频会议等）
7. TCP 首部开销20字节；UDP 的首部开销小，只有 8 个字节
8. TCP经常用于：FTP 文件传输，HTTP/HTTPS
9. UDP常用于包总量较少的通信包括`DNS`、`SNMP`等，视频、音频等多媒体通信，广播通信

TCP三次握手的原因

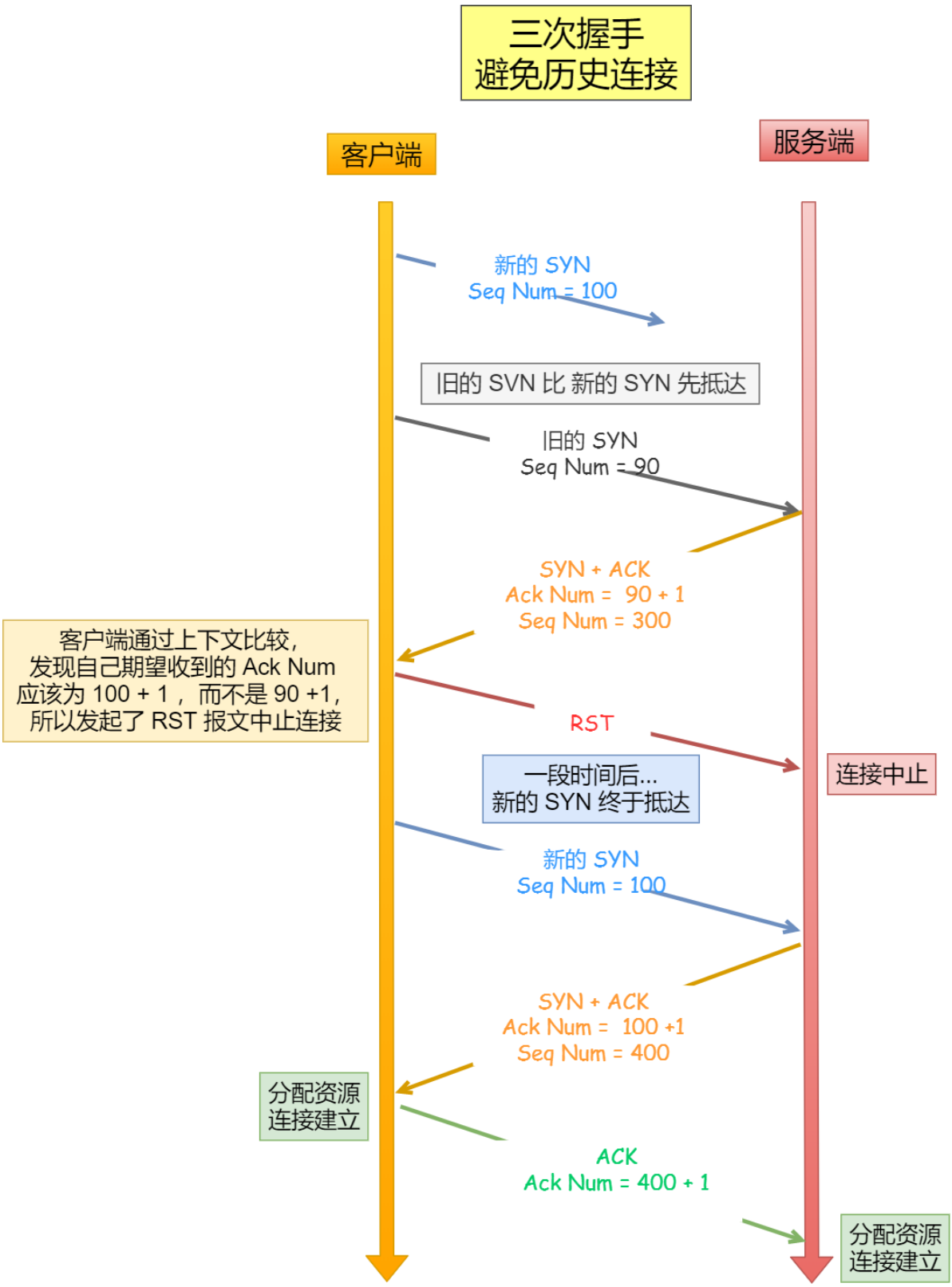
原因一：避免历史连接

我们来看看 RFC 793 指出的 TCP 连接使用三次握手的**首要原因**：

The principle reason for the three-way handshake is to prevent old duplicate connection initiations from causing confusion.

简单来说，三次握手的**首要原因是为了防止旧的重复连接初始化造成混乱。**

网络环境是错综复杂的，往往并不是如我们期望的一样，先发送的数据包，就先到达目标主机，反而它很骚，可能会由于网络拥堵等乱七八糟的原因，会使得旧的数据包，先到达目标主机，那么这种情况下 TCP 三次握手是如何避免的呢？



三次握手避免历史连接

客户端连续发送多次 SYN 建立连接的报文，在网络拥堵等情况下：

- 一个「旧 SYN 报文」比「最新的 SYN 」 报文早到达了服务端；
- 那么此时服务端就会回一个 `SYN + ACK` 报文给客户端；
- 客户端收到后可以根据自身的上下文，判断这是一个历史连接（序列号过期或超时），那么客户端就会发送 `RST` 报文给服务端，表示中止这一次连接。

如果是两次握手连接，就不能判断当前连接是否是历史连接，三次握手则可以在客户端（发送方）准备发送第三次报文时，客户端因有足够的上下文来判断当前连接是否是历史连接：

- 如果是历史连接（序列号过期或超时），则第三次握手发送的报文是 `RST` 报文，以此中止历史连接；
- 如果不是历史连接，则第三次发送的报文是 `ACK` 报文，通信双方就会成功建立连接；

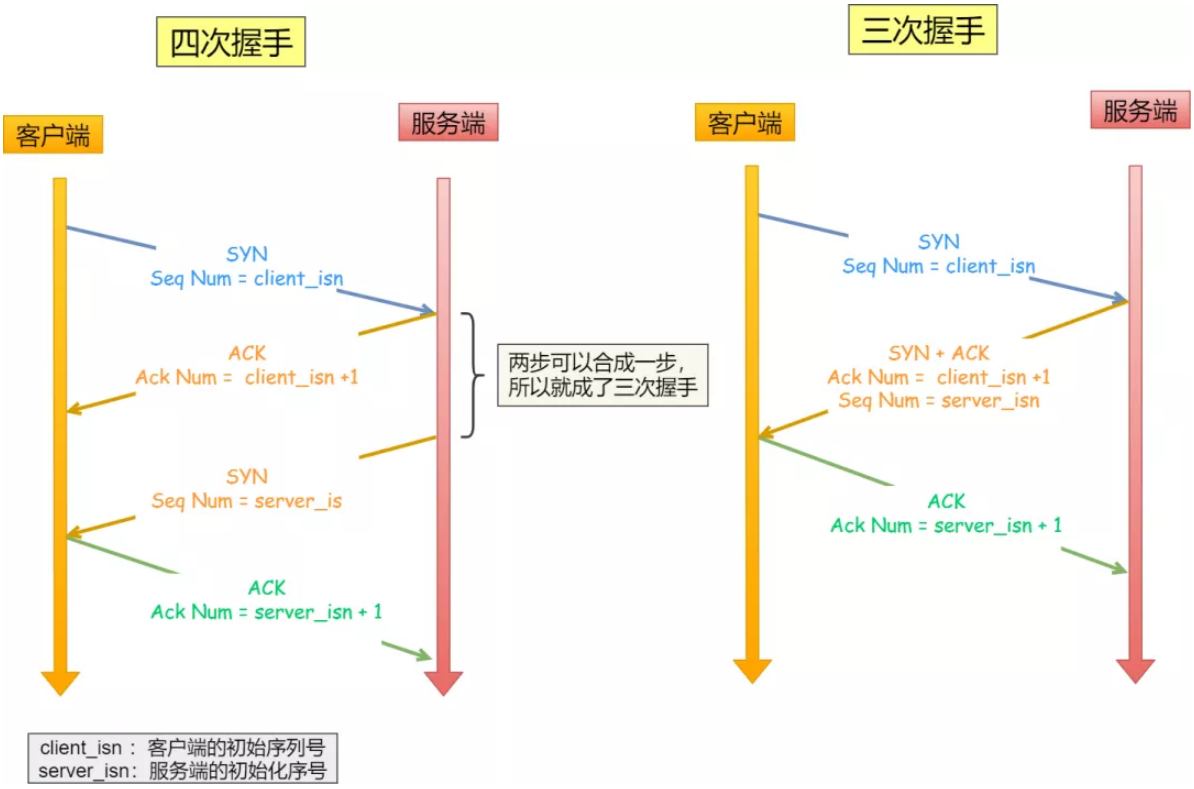
所以， TCP 使用三次握手建立连接的最主要原因是**防止历史连接初始化了连接。**

原因二：同步双方初始序列号

TCP 协议的通信双方， 都必须维护一个「序列号」， 序列号是可靠传输的一个关键因素，它的作用：

- 接收方可以去除重复的数据；
- 接收方可以根据数据包的序列号按序接收；
- 可以标识发送出去的数据包中， 哪些是已经被对方收到的；

可见，序列号在 TCP 连接中占据着非常重要的作用，所以当客户端发送携带「初始序列号」的 `SYN` 报文的时候，需要服务端回一个 `ACK` 应答报文，表示客户端的 SYN 报文已被服务端成功接收，那当服务端发送「初始序列号」给客户端的时候，依然也要得到客户端的应答回应，**这样一来一回，才能确保双方的初始序列号能被可靠的同步。**



四次握手与三次握手

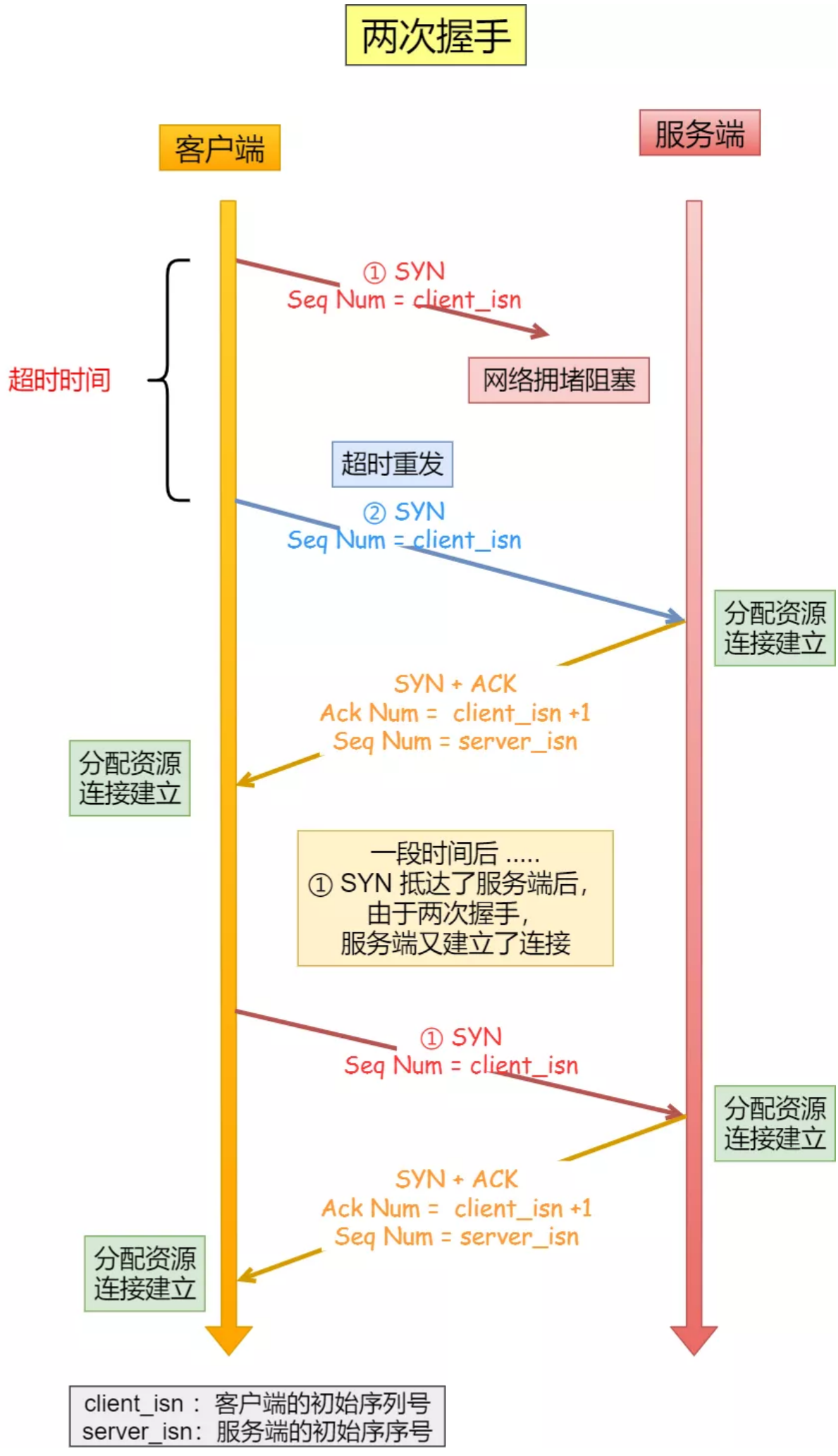
四次握手其实也能够可靠的同步双方的初始化序号，但由于**第二步和第三步可以优化成一步**，所以就成了「三次握手」。

而两次握手只保证了一方的初始序列号能被对方成功接收，没办法保证双方的初始序列号都能被确认接收。

原因三：避免资源浪费

如果只有「两次握手」，当客户端的 `SYN` 请求连接在网络中阻塞，客户端没有接收到 `ACK` 报文，就会重新发送 `SYN` ，由于没有第三次握手，服务器不清楚客户端是否收到了自己发送的建立连接的 `ACK` 确认信号，所以每收到一个 `SYN` 就只能先主动建立一个连接，这会造成什么情况呢?

如果客户端的 `SYN` 阻塞了，重复发送多次 `SYN` 报文，那么服务器在收到请求后就会**建立多个冗余的无效链接，造成不必要的资源浪费。**


两次握手
客户端
服务端
① SYN
Seq Num = client_isn
超时时间
网络拥堵阻塞
超时重发
② SYN
Seq Num = client_isn
分配资源
连接建立
SYN + ACK
Ack Num = client_isn +1
Seq Num = server_isn
分配资源
连接建立
一段时间后 .....
① SYN 抵达了服务端后，
由于两次握手，
服务端又建立了连接
① SYN
Seq Num = client_isn
分配资源
连接建立
SYN + ACK
Ack Num = client_isn +1
Seq Num = server_isn
分配资源
连接建立
client_isn： 客户端的初始序列号
server_isn： 服务端的初始序序号

两次握手会造成资源浪费

即两次握手会造成消息滞留情况下，服务器重复接受无用的连接请求 `SYN` 报文，而造成重复分配资源。

小结

TCP 建立连接时，通过三次握手**能防止历史连接的建立，能减少双方不必要的资源开销，能帮助双方同步初始化序列号。**序列号能够保证数据包不重复、不丢弃和按序传输。

不使用「两次握手」和「四次握手」的原因：

- 「两次握手」：无法防止历史连接的建立，会造成双方资源的浪费，也无法可靠的同步双方序列号；
- 「四次握手」：三次握手就已经理论上最少可靠连接建立，所以不需要使用更多的通信次数。

初始序列号 ISN 是如何随机产生的?

起始 `ISN` 是基于时钟的，每 4 毫秒 + 1，转一圈要 4.55 个小时。

RFC1948 中提出了一个较好的初始化序列号 ISN 随机生成算法。

ISN = M + F (localhost, localport, remotehost, remoteport)

- `M` 是一个计时器，这个计时器每隔 4 毫秒加 1。
- `F` 是一个 Hash 算法，根据源 IP、目的 IP、源端口、目的端口生成一个随机数值。要保证 Hash 算法不能被外部轻易推算得出，用 MD5 算法是一个比较好的选择。

TCP四次挥手的原因

客户端发送了 FIN 连接释放报文之后，服务器收到了这个报文，就进入了 CLOSE-WAIT 状态。这个状态是为了让服务器端发送还未传送完毕的数据，传送完毕之后，服务器会发送 FIN 连接释放报文。

TIME_WAIT

客户端接收到服务器端的 FIN 报文后进入此状态，此时并不是直接进入 CLOSED 状态，还需要等待一个时间计时器设置的时间 2MSL。这么做有两个理由：

- 确保最后一个确认报文能够到达。如果 B 没收到 A 发送来的确认报文，那么就会重新发送连接释放请求报文，A 等待一段时间就是为了处理这种情况的发生。
- 等待一段时间是为了让本连接持续时间内所产生的所有报文都从网络中消失，使得下一个新的连接不会出现旧的连接请求报文。

TCP 黏包问题

原因

TCP 是一个基于字节流的传输服务（UDP 基于报文的），“流” 意味着 TCP 所传输的数据是没有边界的。所以可能会出现两个数据包黏在一起的情况。

解决

- 发送定长包。如果每个消息的大小都是一样的，那么在接收对等方只要累计接收数据，直到数据等于一个定长的数值就将它作为一个消息。
- 包头加上包体长度。包头是定长的 4 个字节，说明了包体的长度。接收对等方先接收包头长度，依据包头长度来接收包体。
- 在数据包之间设置边界，如添加特殊符号 `\r\n` 标记。FTP 协议正是这么做的。但问题在于如果数据正文中也含有 `\r\n`，则会误判为消息的边界。
- 使用更加复杂的应用层协议。