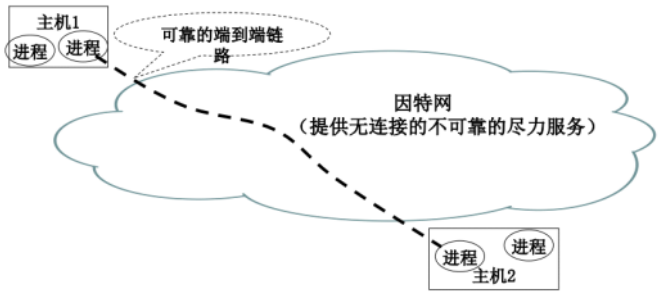
传输层

2019年5月29日 2:17

概述



- 传输层协议称为**端到端或进程到进程的协议**。因特网的传输层可以为两个进程在不可靠的网络层上建立一条**可靠的逻辑链路**，可以提供**字节流**传输服务，并且可以进行**流控制**和**拥塞控制**。

流控制：有滑动窗口协议

拥塞控制congest control：指很多数据发到整个网络上就会发生拥塞。TCP可以控制大家发送数据的量

TCP/UDP报文

- **因特网的传输层**有两个协议：**UDP**和**TCP**。UDP协议提供不可靠的尽力服务，TCP协议提供可靠的字节流服务。

IP头部	UDP报文/TCP报文

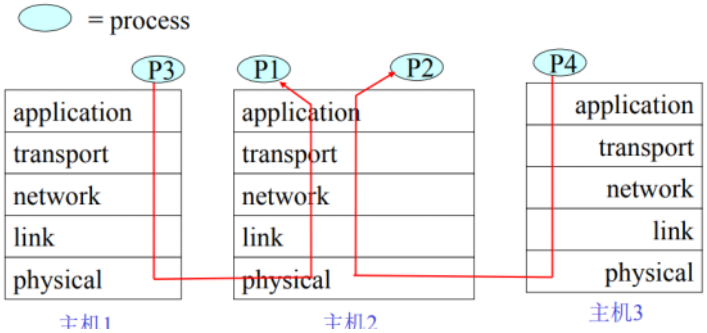
协议号：TCP=6　　ICMP=1
UDP=17　　IGMP=2

- 我们把传输层的数据单元（报文）称为数据段(segment)。

传输层的多路复用和解多路复用

接收方解多路复用：
把收到的数据段交给正确的上层进程

发送方多路复用：
收集来自上层进程的数据形成数据段并发送出去



端口号

UDP协议和TCP协议如何知道把收到的数据段交给哪个上层进程呢？利用数据段(segment)中的目的端口号 (2个字节)

- 知名端口

  0~1023。为提供知名网络服务的系统进程所用。
  例如: 80-HTTP，21-ftp Control，20-ftp Data，23-telnet，25-SMTP，110-POP3，53-DNS

- 注册端口

  1024~49151。在IANA注册的专用端口号，为企业软件所用。

- 动态端口

  49152~65535。没有规定用途的端口号，一般用户可以随意使用。也称为私用或暂用端口号。

TCP和UDP的区别

1. TCP是面向连接的，UDP是无连接的；
   UDP发送数据之前不需要建立连接
2. TCP是可靠的，UDP不可靠；
   UDP接收方收到报文后，不需要给出任何确认
3. TCP只支持点对点通信，UDP支持一对一、一对多、多对一、多对多；
4. TCP是面向字节流的，UDP是面向报文的；
   面向字节流是指发送数据时以字节为单位，一个数据包可以拆分成若干组进行发送，而UDP一个报文只能一次发完。
5. TCP有拥塞控制机制，UDP没有。网络出现的拥塞不会使源主机的发送速率降低，这对某些实时应用是很重要的，比如媒体通信，游戏；
6. TCP首部开销（20字节）比UDP首部开销（8字节）要大

什么时候选择UDP什么时候选择TCP

对某些实时性要求比较高的情况，选择UDP，比如游戏，媒体通信，实时视频流（直播），即使出现传输错误也可以容忍；其它大部分情况下，HTTP都是用TCP，因为要求传输的内容可靠，不出现丢失

HTTP不可以使用UDP，HTTP需要基于可靠的传输协议，而UDP不可靠

面向连接和无连接的区别

无连接的网络服务（数据报服务）-- 面向连接的网络服务（虚电路服务）

虚电路服务：首先建立连接，所有的数据包经过相同的路径，服务质量有较好的保证；

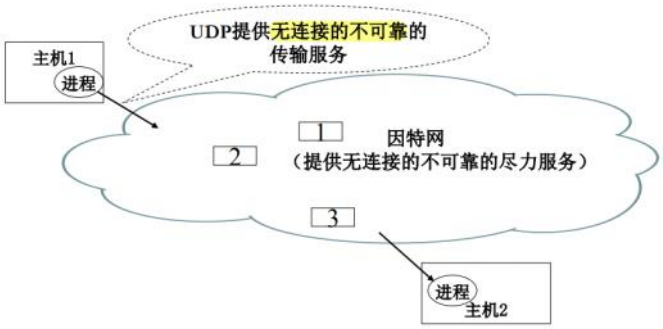
数据报服务：每个数据包含目的地址，数据路由相互独立（路径可能变化）；网络尽最大努力交付数据，但不保证不丢失、不保证先后顺序、不保证在时限内交付；网络发生拥塞时，可能会将一些分组丢弃；

TCP是进程间面向连接,即要先建立连接(如上网打开浏览器某个链接,服务器响应就是建立了连接),然后交给下层网络层链路层进行无连接的数据报传送(即服务器给你网页页面)。网络层不用虚电路,用面向无连接的数据报方式。

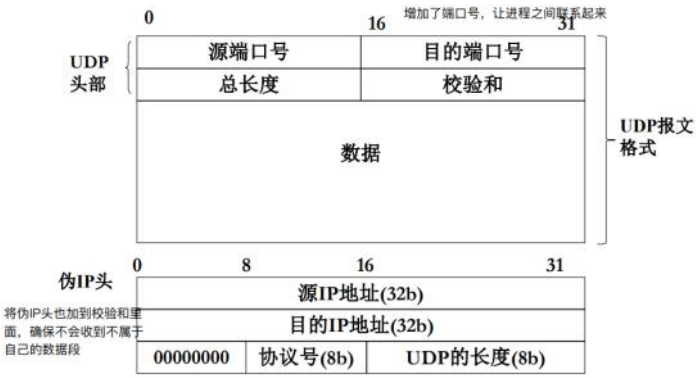
TCP如何保证传输的可靠性

1. 数据包校验
2. 对失序数据包重新排序（TCP报文具有序列号）
3. 丢弃重复数据
4. 应答机制：接收方收到数据之后，会发送一个确认（通常延迟几分之一秒）；
5. 超时重发：发送方发出数据之后，启动一个定时器，超时未收到接收方的确认，则重新发送这个数据；
6. 流量控制：确保接收端能够接收发送方的数据而不会缓冲区溢出

UDP协议



- 用户数据报协议(User Datagram Protocol)只提供无连接的不可靠的尽力服务。发送给接收进程的数据有可能丢失，也有可能错序。
- 接收进程每次接收一个完整的数据报，如果进程设置的接收缓冲区不够大，收到的数据报将被截断。



总长度： 整个UDP报文的长度。
源端口号和目的端口号： 用于关联发送进程和接收进程。
校验和： 由伪IP头、UDP头(校验和为0)和UDP数据形成。其中，伪IP头的协议号为17。如果发送方把校验和设置为0，接收方会忽略校验和 。UDP长度就是UDP头部的总长度。

pseudo header的作用：

增加检验误差范围：TCP伪真报头

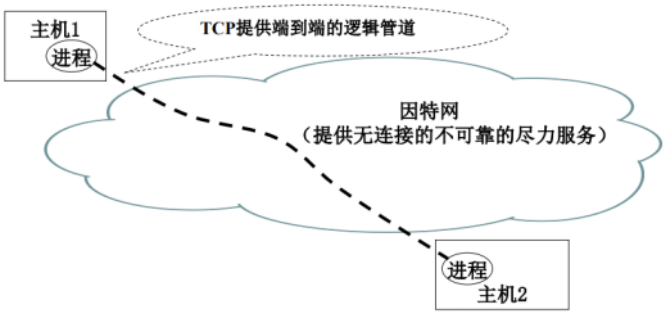
为此，TCP检验和计算做了一个改变。这个特殊的计算方法最终也被UDP所采用。除了对TCP段所有真实数据做校验和计算以外，一个12bit的TCP伪真报头在校验和计算之前被创建。这个报头包含来自TCP报头和IP数据包中的重要信息将被封装成TCP字段。

要计算TCP段报头的Checksum域的值，TCP伪真包头首先被创建和放置，从逻辑上讲，放在TCP段之前。然后对所有TCP段以及TCP伪真报头进行校验和计算。最后TCP伪真报头被丢弃。

当TCP段到达目的地，接收方的软件将执行同样的计算，它将根据TCP段中的真实数据行程一个TCP伪真报头，然后执行校验和计算(计算之前设置Checksum域值为零)。假如计算出来的结果跟源设备(Source device)放置在checksum域的值不匹配的话，这表明发生了某种类型的错误，通常这个字段将会被丢弃。

当然了，还有一个有趣的TCP伪真报头的含义：它违反了TCP设计者试图尊重TCP,IP分层的层次架构原则。对于校验，TCP必须知道IP信息，而技术上它不应该知道。TCP校验和就散要求，例如需要从接受端获取的IP数据包含递给TCP层的IP报头中的协议号。TCP伪真报头是一个回避严格层次架构的实用性很好的例子。

传输控制协议(TCP)



- TCP(Transmission Control Protocol)为进程之间提供面向连接的**可靠的数据传送服务**。
- TCP为**全双工**协议。TCP提供**流控制**机制，即控制发送方的发送速度，使发送的数据不会淹没接收方。作为因特网的主要数据发源地，TCP还提供**拥塞控制**功能。

TCP流控制：

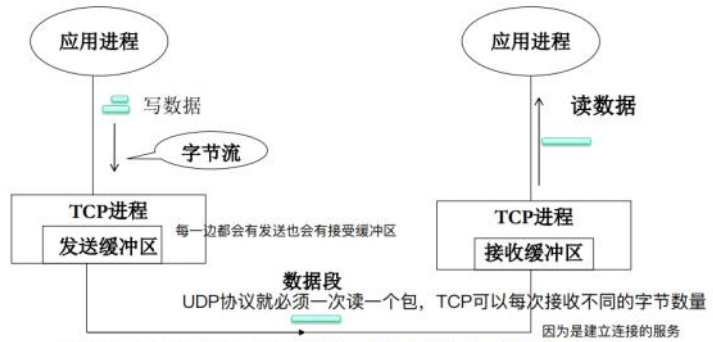
所谓的流量控制就是让发送方的发送速率不要太快，让接收方来得及接受。利用滑动窗口机制可以很方便的在TCP连接上实现对发送方的流量控制。TCP的窗口单位是字节，不是报文段，发送方的发送窗口不能超过接收方给出的接收窗口的数值。

拥塞控制的原理：

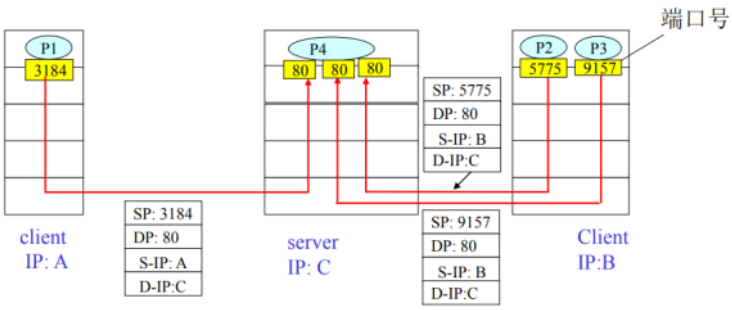
在某段时间，若对网络中的某一资源的需求超过了该资源所能提供的可用部分，网络的性能就要变化，这种情况叫做拥塞。网络拥塞往往是由许多因素引起的，简单的提高节点处理机的速度或者扩大结点缓存的存储空间并不能解决拥塞问题。例如当某个结点缓存容量扩展到非常大，于是凡到达该结点的分组均可在结点的缓存队列中排队，不受任何限制。由于输出链路的容量和处理机的速度并未提高，因此在这队列中的绝大多数的分组在排队等待时间会大大增加，结果上层软件只好把他们进行重传。
因此，问题的是指往往是整个系统的各个部分不匹配，只有各个部分平衡了，问题才会得到解决。

拥塞控制和流量控制的差别：

所谓拥塞控制就是防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。拥塞控制所要做的都有一个前提，就是网络能承受现有的网络负荷。
流量控制往往指的是点对点通信量的控制，是个端到端的问题。流量控制所要做的就是控制发送端发送数据的速率，以便使接收端来得及接受。

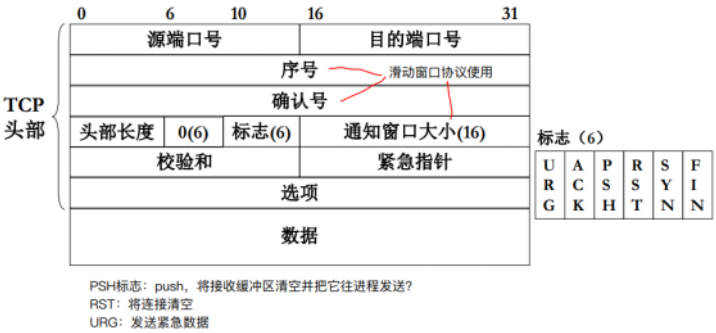


- 一个TCP连接提供可靠的字节流服务。字节流服务表示没有消息边界。例如，多次发送的数据可以放在一个数据段中传送且不标识边界。
- 每个数据段的数据部分的最大长度(字节)不能超过MSS (Maximum Segment Size) 。 MTU：数据链路层数据帧数据部分最大部分
- 每个TCP连接可以由四元组唯一标识：源IP地址，源端口号,目的IP地址,目的端口号。



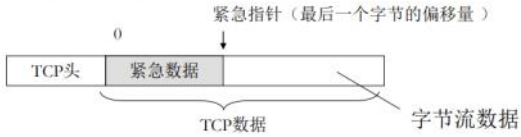
SP: source port#
DP: dest port#
S-IP: source IP addr
D-IP: dest IP addr

TCP报文格式



PSH标志：push，将接收缓冲区清空并把它往进程发送?
RST：将连接清空
URG：发送紧急数据

- 字节流中的每个字节均被编号。初始序号采用基于时间的方案， 一般采用随机数。数据部分的第一个字节的编号为初始序号加1。
- 每个数据段的**序号**采用其数据部分第一个字节的编号。**确认号**为期待接收的下一个数据段的序号。只有设置了确认标志，确认号才有效。
- **头部长度**以四个字节为单位。
- **校验和**由伪IP头、TCP头和TCP数据部分形成。其形成方法与UDP协议类似。
- **紧急指针**用于指出带外数据(out-of-band)的边界。标志URG为1时有效。



- 发送窗口为确认号之后发送方还可以发送的字节的序号范围。接收方用**通知窗口大小(advertised window)**告知发送方接收窗口的大小，发送方会据此修改发送窗口大小。
- 建立连接时的**选项**：MSS(Maximumu Segment Size)、窗口比例(Scale)：是否使用选择性确认(SACK-Permitted)。 Unix系统的默认值：MSS为536，SACK-Permitted为False。Windows 的默认值MSS为1460，SACK-Permitted为True。
- 数据传送时的选项：选择性确认的序号范围(Seletive ACK,SACK)，时间戳等。

URG: 表示本数据段包含紧急数据。

ACK: 表示确认号有效。

PSH: 告知接收方发送方执行了推送(Push)操作，接收方需要尽快将所有缓存的数据交给接收进程。

RST: 重置(Reset)连接。因为连接出现了错误，通知对方立即中止连接并释放相关资源。

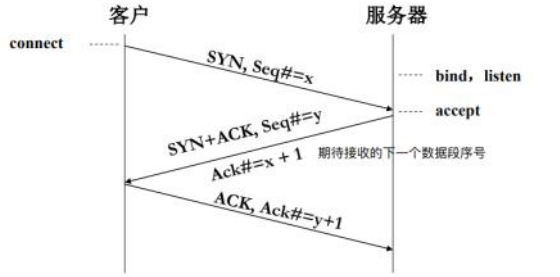
SYN: 同步(Synchronize)序号标志，用来发起一个TCP连接。

FIN: 结束(Finish)标志，向对方表示自己不再发送数据。

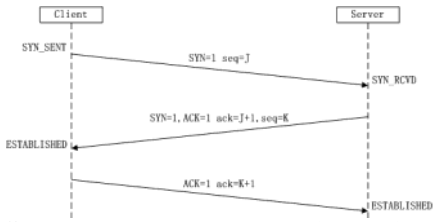
* 所有标志为1有效。

三次握手建立连接

开放三个标志位：SYN ACK FIN



- 客户端需要知道服务器的IP地址和端口号。服务器收到客户端发来的连接请求(SYN报文)后查看是否有进程监听该端口，若有，则将此连接请求传给该进程，否则，服务器发RST拒绝它。如果该进程接受连接请求，则发回SYN+ACK报文，。
- 每一步均采用超时重传，多次重发后将放弃。重发次数与间隔时间依系统不同而不同。
- x和y为初始序号，分别用于两个方向的数据传送，它们一般采用基于时间的方案，如：随机数。两个方向的下一个数据段的序号分别为x+1和y+1。
- 头两个数据段确定的选项： Scale，MSS ，SACK-Permited



- 第一次握手：Client将SYN置1，随机产生一个初始序列号seq发送给Server，进入SYN_SENT状态；
- 第二次握手：Server收到Client的SYN=1之后，知道客户端请求建立连接，将自己的SYN置1，ACK置1，产生一个acknowledge number=sequence number+1，并随机产生一个自己的初始序列号，发送给客户端；进入SYN_RCVD状态；
- 第三次握手：客户端检查acknowledge number是否为序列号+1，ACK是否为1，检查正确之后将自己的ACK置为1，产生一个acknowledge number=服务器发的序列号+1，发送给服务器；进入ESTABLISHED状态；服务器检查ACK为1和acknowledge number为序列号+1之后，也进入ESTABLISHED状态；完成三次握手，连接建立。

TCP连接可以两次握手吗?

client 发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达 server。本来这是一个早已失效的报文段。但 server 收到此失效的连接请求报文段后，就误认为是 client 再次发出的一个新的连接请求。于是就向 client 发出确认报文段，同意建立连接。假设不采用 “三次握手”，那么只要 server 发出确认，新的连接就建立了。由于现在 client 并没有发出建立连接的请求，因此不会理睬 server 的确认，也不会向 server 发送数据。但 server 却以为新的运输连接已经建立，并一直等待 client 发来数据。这样，server 的很多资源就白白浪费掉了。采用 “三次握手” 的办法可以防止上述现象发生。例如刚才那种情况，client 不会向 server 的确认发出确认。server 由于收不到确认，就知道 client 并没有要求建立连接。

可以采用四次握手吗

可以。但是会降低传输的效率。

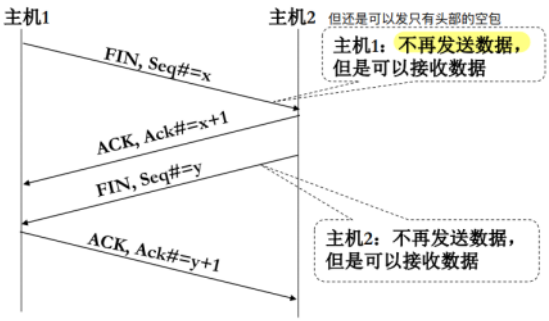
四次握手是指：第二次握手：Server只发送ACK和acknowledge number；而Server的SYN和初始序列号在第三次握手时发送；原来协议中的第三次握手变为第四次握手。出于优化目的，四次握手中的二、三可以合并。

第三次握手中，如果客户端的ACK未送达服务器，会怎样?

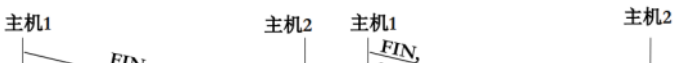
由于Server没有收到ACK确认，因此会重发之前的SYN+ACK（默认重发五次，之后自动关闭连接），Client收到后会重新传ACK给Server。

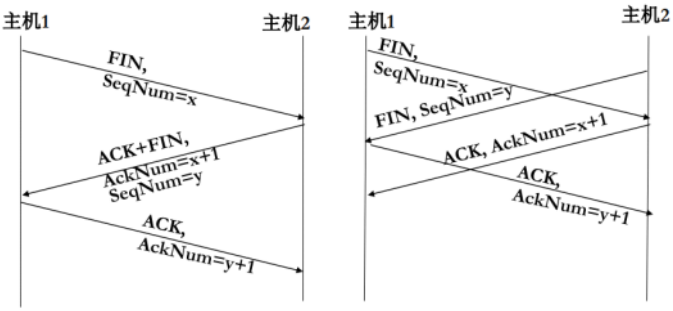
如果Client向服务器发送数据，服务器会以RST包响应。

四次握手关闭连接



- FIN报文采用超时自动重发方式。在若干次重发后依然没有收到确认，则发送RST报文给对方后强行关闭连接。不同的系统重发方法不同。x和y都是上一个收到的数据段的确认号。
- 先发送FIN报文的一方在ACK发送完毕后需要等待2MSL(Maximum Segment Lifetime)的时间才完全关闭连接。TCP标准中MSL采用60秒，Unix采用30秒。





合并中间两次握手(ACK和FIN)　　双方同时发出FIN

TCP滑动窗口

- TCP协议使用选择性确认，不使用NAK。
- 只有一个超时定时器。
- 采用字节流方式，每个数据段使用其第一个字节的编号作为**序号**。**确认号**为期待接收的下一个字节(下一个数据段)的序号。

初始序号是随机的，是建立连接的序号x，然后数据段开始的序号是x+1

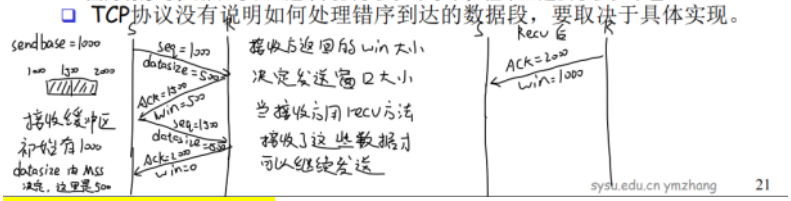
- TCP连接: MSS=1000 x(初始序号)=1999 发送的字节数=2500

TCP协议没有数据段总长度，但可以用IP协议头部中的总长度减去头部长度得到

Seq#	2000	3000	3500
	1000	500	1000

因为有序号，所以可以知道中间丢了多少个字节，但不知道丢了多少个包

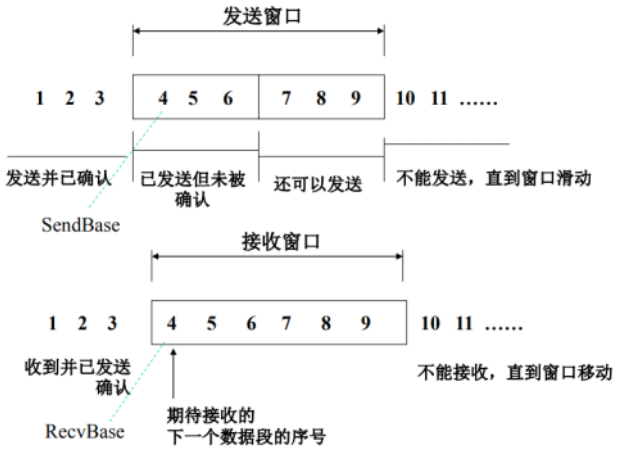
- TCP协议没有说明如何处理错序到达的数据段，要取决于具体实现。



为什么序号要用一个与时间相关的随机数?

如果连接关闭，但网络会遗留一些带序号的数据，但如果这两个进程之间又重新建立一次连接，就很可能出现被网上遗留数据段的干扰，所以如果使用了随机序号的数据，则就不会受到干扰。

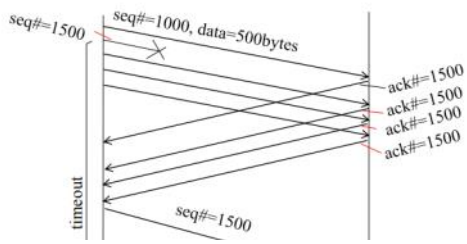
ACK是期待接收下一个包的序号



* advWin为接收窗口的大小，即空闲块的大小（包含错序到达的数据段）

快速重传

- 如果发送方收到一个数据段的3次重复的*ACK*，它就认为其后的数据段（由确认号指出）已经丢失，在超时之前会重传该数据段，这种方法称为快速重传(fast retransmit)。



接），Client收到后会重新传ACK给Server。

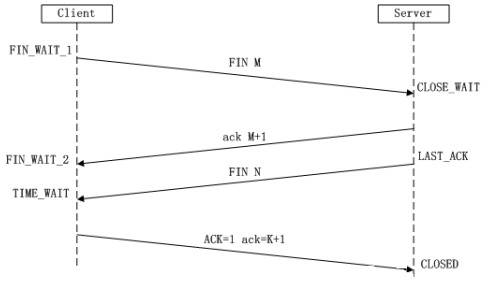
如果Client向服务器发送数据，服务器会以RST包响应。

如果已经建立了连接，但客户端出现了故障怎么办?

服务器每收到一次客户端的请求后都会重新复位一个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75秒钟发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接。

初始序列号是什么?

TCP连接的一方A，随机选择一个32位的序列号（Sequence Number）作为发送数据的初始序列号（Initial Sequence Number，ISN），比如为1000，以该序列号为原点，对要传送的数据进行编号：1001、1002...三次握手时，把这个初始序列号传送给另一方B，以便在传输数据时，B可以确认什么样的数据编号是合法的；同时在进行数据传输时，A还可以确认B收到的每一个字节，如果A收到了B的确认编号（acknowledge number）是2001，就说明编号为1001-2000的数据已经被B成功接受。



- 第一次挥手：Client将FIN置为1，发送一个序列号seq给Server；进入FIN_WAIT_1状态；
- 第二次挥手：Server收到FIN之后，发送一个ACK=1，acknowledge number=收到的序列号+1；进入CLOSE_WAIT状态。此时客户端已经没有要发送的数据了，但仍可以接受服务器发来的数据。
- 第三次挥手：Server将FIN置1，发送一个序列号给Client；进入LAST_ACK状态；
- 第四次挥手：Client收到服务器的FIN后，进入TIME_WAIT状态；接着将ACK置1，发送一个acknowledge number=序列号+1给服务器；服务器收到后，确认acknowledge number后，变为CLOSED状态，不再向客户端发送数据。客户端等待2*MSL（报文段最长寿命）时间后，也进入CLOSED状态。完成四次挥手。

为什么不能把服务器发送的ACK和FIN合并起来，变成三次挥手（CLOSE_WAIT状态意义是什么）?

因为服务器收到客户端断开连接的请求时，可能还有一些数据没有发完，这时先回复ACK，表示接收到了断开连接的请求。等到数据发完之后再发FIN，断开服务器到客户端的数据传送。

如果第二次挥手时服务器的ACK没有送达客户端，会怎样?

客户端没有收到ACK确认，会重新发送FIN请求。

客户端TIME_WAIT状态的意义是什么?

第四次挥手时，客户端发送给服务器的ACK有可能丢失，TIME_WAIT状态就是用来重发可能丢失的ACK报文。如果Server没有收到ACK，就会重发FIN，如果Client在2*MSL的时间内收到了FIN，就会重新发送ACK并再次等待2MSL，防止Server没有收到ACK而不断重发FIN。

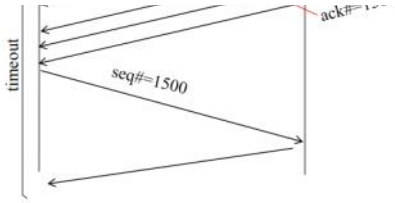
MSL(Maximum Segment Lifetime)，指一个片段在网络中最大的存活时间，2MSL就是一个发送和一个回复所需的最大时间。如果直到2MSL，Client都没有再次收到FIN，那么Client推断ACK已经被成功接收，则结束TCP连接。

TCP流量控制

窗口字段 —— 占 2 字节。窗口字段用来控制对方发送的数据量，单位为字节。TCP 连接的一端根据设置的缓存空间大小确定自己的接收窗口大小，然后通知对方以确定对方的发送窗口的上限。

使用滑动窗口协议实现流量控制。防止发送方发送速率太快，接收方缓存区不够导致溢出。接收方会维护一个接收窗口 receiver window（窗口大小单位是字节），接受窗口的大小是根据自己的资源情况动态调整的，在返回ACK时将接受窗口大小放在TCP报文中的窗口字段告知发送方。发送窗口的大小不能超过接受窗口的大小，只有当发送方发送并收到确认之后，才能将发送窗口右移。

发送窗口的上限为接受窗口和拥塞窗口中的较小值。接受窗口表明了接收方的接收能力，拥塞窗口表明了网络的传送能力。



超时重传定时器(ICSK_TIME_RETRANS)在以下几种情况下会被激活：

1. 发现对端把保存在接收缓冲区的SACK段丢弃时。

2. 发送一个数据段时，发现之前网络中不存在发送且未确认的段。

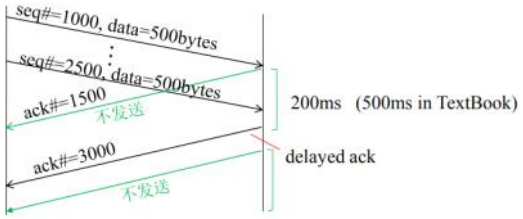
   之后每当收到确认了新数据段的ACK，则重置定时器。所以超时定时器一般是用来算未确认的最小序号

3. 发送SYN包后。

4. 一些特殊情况。

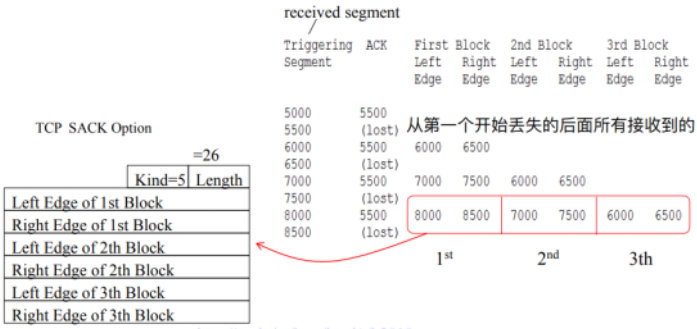
延迟确认

- 采用**延迟确认**(*delayed ACK*)时，接收方并不在收到数据段立即进行确认，而是延迟一段时间再确认。如果这个期间收到多个数据段，则只需要发送一个确认。如果在这个期间接收方有数据帧要发往发送方，还可以使用**捎带确认**(*piggybacking*)。
- 大部分的系统 (Windows, Unix)的延迟确认时间为200毫秒。TCP标准要求延迟确认不大于500毫秒。



选择性确认

选择性确认允许接收方把收到的数据块通过数据段的选项告知发送方，使发送方不会重传这些数据块。



http://tools.ietf.org/html/rfc2018
http://tools.ietf.org/html/rfc2883

TCP超时计算- 初始公式

$$EstimatedRTT = (1-\alpha) \times EstimatedRTT + \alpha \times SampleRTT$$

- $0<\alpha\leq 1$, α越小过去样本的影响越大。
- 一般取值: $\alpha = 0.9$。这会使过去影响指数减少。
- 这个公式也称为指数加权移动平均方法 (Exponentially Weighted Moving Average).
- RTO(retransmission timeout) = 2x EstimatedRTT

这个公式的另一种表示方法：

$$EstimatedRTT = EstimatedRTT + \alpha \times (SampleRTT - EstimatedRTT)$$

* 蓝色表示变量的旧值

为什么需要重传?

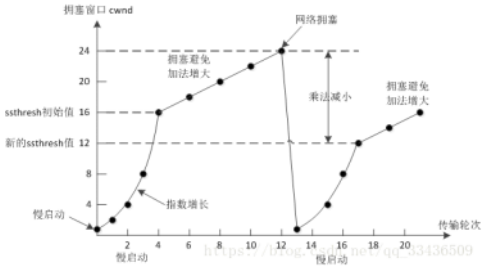
TCP本身需要提供可靠的服务，方式之一就是确认接收方真的收到了数据，如果过了一段时间，即超时了，还没有收到确认的报文，

发送窗口的上限为接受窗口和拥塞窗口中的较小值。接受窗口表明了接收方的接收能力，拥塞窗口表明了网络的传送能力。

什么是零窗口（接收窗口为0时会怎样）

如果接收方没有能力接收数据，就会将接收窗口设置为0，这时发送方必须暂停发送数据，但是会启动一个持续计时器(persistence timer)，到期后发送一个大小为1字节的探测数据包，以查看接收窗口状态。如果接收方能够接收数据，就会在返回的报文中更新接收窗口大小，恢复数据传送。

TCP拥塞控制



拥塞控制主要由四个算法组成：慢启动（Slow Start）、拥塞避免（Congestion voidance）、快重传（Fast Retransmit）、快恢复（Fast Recovery）

慢启动：刚开始发送数据时，先把拥塞窗口（congestion window）设置为一个最大报文段MSS的数值，每收到一个新的确认报文之后，就把拥塞窗口加1个MSS。这样每经过一个传输轮次（或者说是每经过一个往返时间RTT），拥塞窗口的大小就会加倍

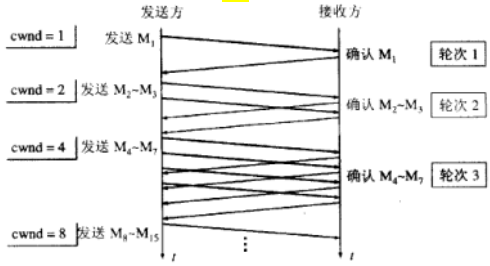


图 5-24 发送方每收到一个确认就把窗口 cwnd 加 1

拥塞避免：当拥塞窗口的大小达到慢开始门限(slow start threshold)时，开始执行拥塞避免算法，拥塞窗口大小不再指数增加，而是线性增加，即每经过一个传输轮次只增加1MSS.

无论在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞（其根据就是没有收到确认），就要把慢开始门限ssthresh设置为出现拥塞时的发送方窗口值的一半（但不能小于2）。然后把拥塞窗口cwnd重新设置为1，执行慢开始算法。（这是不使用快重传的情况）

快重传：快重传要求接收方在收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时捎带确认。快重传算法规定，发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器时间到期。

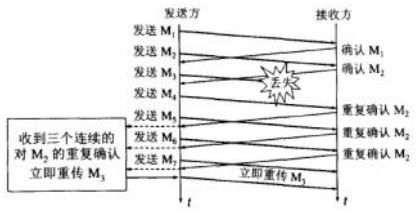
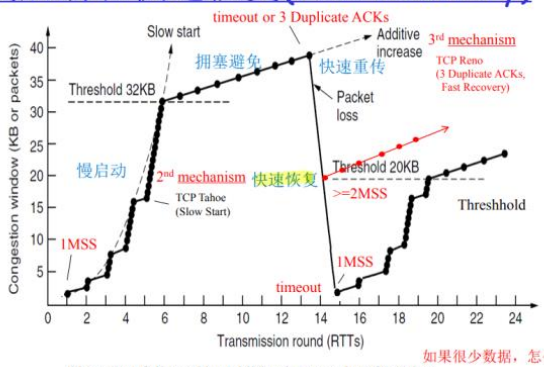


图 5-26 快重传的示意图

TCP拥塞控制:
慢启动和快速恢复(Fast Recovery)



Slow start followed by additive increase in TCP Tahoe

如果很少数据，怎么办？

快恢复：当发送方连续收到三个重复确认时，就把慢开始门限减半，然后从慢开始门限重新执行，执行拥

为什么需要重传?

TCP本身需要提供可靠的服务，方式之一就是确认接收方真的收到了数据，如果过了一段时间，即超时了，还没有收到确认的报文，认为报文可能被丢失，就重新传送报文，确保数据都能被收到

为什么要动态的计算超时时间?

网络流量和路由器在包的传输过程中可能改变，因此RTT（Round Trip Time）也会变化，如果超时时间保持不变，假如RTT变的大了，可能出现ACK还在再发送的路上，却直接重发了包，造成不必要的浪费

如何动态计算超时重传时间?

TCP经典算法RTT是：R <- αR + (1-α)M，重传时间为 RTO=Rβ

RTO表示重传超时时间（Retransmission Timeout）意思是如果超过这个时间还没有收到ack就重新发送
β 是RTT的变异系数，当传输时间可以忽略不计的时候，最大时延和平均时延的变化最大，可以看做所有的时延都是因为处理所造成的，这个时候最大值是平均值的两倍，推荐β取值为2。【假设往返时间最大值是R,如果传输时延忽略不计，那么这两次变化的平均传输时延就是0.5R，也就说最大值是平均值的两倍】

Jacobson / Karels 算法

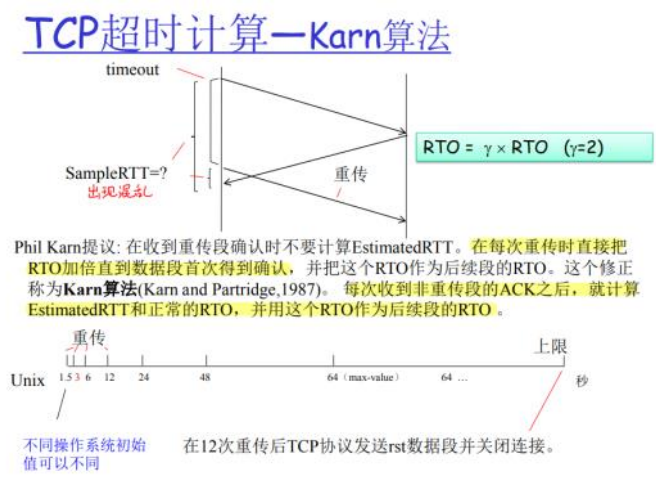
前面两种算法用的都是“加权移动平均”，这种方法最大的毛病就是如果RTT有一个大的波动的话，很难被发现，因为被平滑掉了。所以，1988年，又有人推出来了一个新的算法，这个算法叫Jacobson / Karels Algorithm（参看RFC6289）。这个算法引入了最新的RTT的采样和平滑过的SRTT的差距做因子来计算。 公式如下：（其中的DevRTT是Deviation RTT的意思）

SRTT = SRTT + α (RTT – SRTT) —— 计算平滑RTT

DevRTT = (1-β)*DevRTT + β*(|RTT-SRTT|) ——计算平滑RTT和真实的差距（加权移动平均）

RTO= μ * SRTT + ∂ *DevRTT —— 神一样的公式

（其中：在Linux下，α = 0.125，β = 0.25， μ = 1，∂ = 4 ——这就是算法中的“调得一手好参数”，nobody knows why, it just works...）最后的这个算法在被用在今天的TCP协议中（Linux的源代码在：tcp_rtt_estimator）。



如果很少数据，怎么办?

快恢复：当发送方连续收到三个重复确认时，就把慢开始门限减半，然后从慢开始门限重新执行，执行拥塞避免算法。不执行慢开始算法的原因：因为如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方认为现在网络可能没有出现拥塞。
也有的快重传是把开始时的拥塞窗口cwnd值再增大一点，即等于 ssthresh + 3*MSS 。这样做的理由是：既然发送方收到三个重复的确认，就表明有三个分组已经离开了网络。这三个分组不再消耗网络的资源而是停留在接收方的缓存中。可见现在网络中减少了三个分组。因此可以适当把拥塞窗口扩大些。

TCP粘包

TCP粘包就是指发送方发送的若干包数据到达接收方时粘成了一包，从接收端缓冲区来看，后一包数据的头紧接着前一包数据的尾。

如果发送方发送的多组数据本来就是同一块数据的不同部分，比如说一个文件被分成多个部分发送，这时当然不需要处理粘包现象；如果多个分组毫不相干，甚至是并列关系，那么这个时候就一定要处理粘包现象了

出现粘包的原因：

- 发送方：默认使用Nagle算法（主要作用：减少网络中报文段的数量），将多次间隔较小、数据量较小的数据，合并成一个数据量大的数据块，进行发送；
- 接收方：TCP将接收到的数据包保存在接收缓存里，然后应用程序主动从缓存读取收到的分组。如果TCP接收数据包到缓存的速度大于应用程序从缓存中读取数据包的速度，多个包就会被缓存，应用程序就有可能读取到多个首尾相接粘到一起的包。

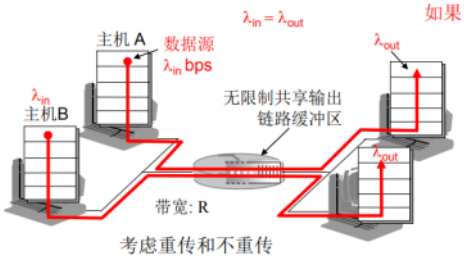
解决方法：

- 发送方：关闭Nagle算法；
- 接收方：在应用层进行处理。将所有数据全部读完之后，再进行分组。分组的方法可以通过规定开始符和结束符的方法；也可以在每组数据前加上数据长度。

拥塞的表现

拥塞(a top-10 problem!):

- 简单来说: "太多的主机发送太快太多的数据给网络处理"。 这不同于流控制!
- 表现:
  - 丢包 (路由器上缓冲区溢出)
  - 长延迟 (在路由器缓冲区中排队)



考虑重传和不重传

TCP拥塞控制

- 超时或收到3个重复ack就认为丢包了，看作拥塞发生了。
- TCP协议通过减少发送速率来控制拥塞。发送速率与发送窗口大小有关:

  发送速率 rate = SWS(Sending Window Size)/RTT

- 引入拥塞窗口变量CongWin来限制SWS。

  SWS = Min(CongWin, AdvWin) → 通知窗口大小

- TCP协议改变CongWin的三种机制:
  - 加性增乘性减(AIMD)
  - 慢启动(slow start)
  - 在超时时间之后的保守方法

CongWin-发送窗口的流量控制
AdvWin-接收窗口要求的流量控制

通过在接收窗口大小和用来限制的拥塞窗口大小来限制发送的数量

TCP拥塞控制:加性增乘性减

- *加性增(additive increase)*: 每个RTT**CongWin**增加1MSS，直到丢包
- *乘性减(multiplicative decrease)*: 在丢包后**CongWin减半**

1st mechanism

用锯齿方式寻找带宽

TCP拥塞控制: 慢启动(slow start)

(1) 初始时，CongWin设为1 MSS, 阈值(threshold)设为65535, 发送一个数据段。

(2) 在当前窗口所有数据段的确认都收到之后, CongWin加倍。
实际上，每收到一个确认，CongWin增加一个MSS。把它称为慢启动(slow start)
是因为这个方法比立即采用通知窗口更慢。

(3) 当拥塞发生时，把当前CongWin (或SWS)的一半保存为阈值(threshold)，然后
CongWin又从 1MSS开始慢启动。

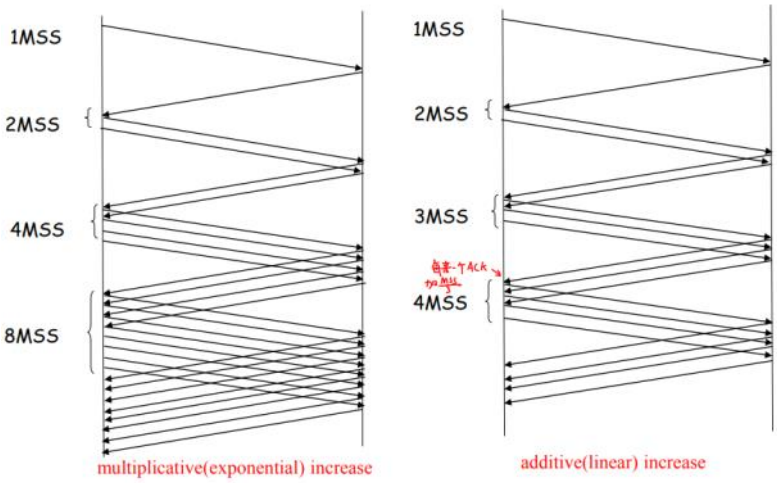
(4) 当 CongWin增长到等于或大于阈值 (threshold)时，在当前窗口所有数据段的确认
都收到之后, CongWin增加一个MSS。 (Congestion Avoidance)
实际上， 每收到一个ACK，CongWin增加SegSize*SegSize/CongWin 。如果发生
拥塞，转 (3)。

(5) 用系统参数TCP_MaxWin(一般为65535)限制 CongWin的大小。

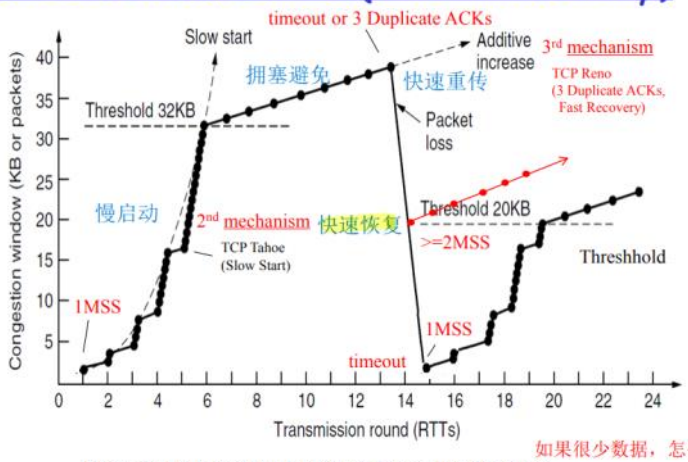
* SegSize 为被确认的数据段的大小

* 假设算法开始时通知窗口大小AdvWin=65535

因为一般CongWin都远小于AdvWin或者是因为接收端总是能很快的从缓冲区接收数据处理，所以一般AdvWin都是大于CongWin的所以只需要考虑CongWin的值来控制拥塞。



TCP拥塞控制:
慢启动和快速恢复(Fast Recovery)



如果很少数据，怎么办？

Slow start followed by additive increase in TCP Tahoe

如果是因为连续收到3个ACK确认的拥塞，则会使用快速恢复