

## 第3章 传输层

### 本章学习目标

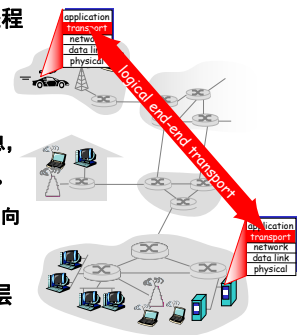
- 掌握传输层服务的原理:
  - 多路复用/多路分解
  - 可靠的数据传输
  - 流量控制
  - 拥塞控制
- 学习传输层协议:
  - UDP: 无连接传输
  - TCP: 面向连接的传输
  - TCP 拥塞控制

### 第三章 传输层

- 3.1 传输层服务
- 3.2 多路复用和多路分解
- 3.3 无连接传输:UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

### 传输层服务和协议

- 为运行于不同主机上的应用进程之间提供**逻辑连接**
- 传输层协议只运行于端系统
  - 发送端: 将来自应用进程的消息, 分解成**报文段**, 传递给网络层。
  - 接收端: 将报文段重组成消息, 向应用层传递
- 应用程序可以使用多种传输层协议
  - 因特网: TCP和 UDP



### 传输层 vs 网络层

- 网络层:** 提供主机间的**逻辑通信**
- 传输层:** 为运行在不同主机上的**应用进程之间**提供**逻辑通信**
  - 将来自应用进程的报文, 移动到网络边缘 (即网络层)
  - 并不关心报文在网络核心是如何移动的

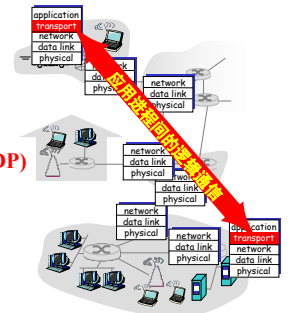
#### 一个示例:

一个家庭的12 个小孩向另一个家庭的12个孩子写信, 两个家庭负责收信的孩子分别是Ann和Bill

- 应用层消息 = 信件
- 应用进程 = 堂兄弟姐妹
- 主机 = 家庭
- 邮政服务=网络层协议
- Ann和Bill=传输层协议

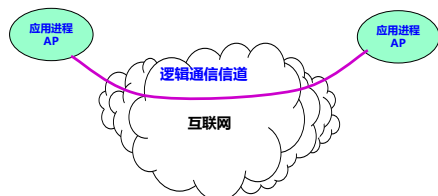
### Internet传输层协议

- 可靠的, 按序的数据交付 (TCP)**
  - 拥塞控制
  - 流量控制
  - 连接建立
- 不可靠的, 无序的数据交付(UDP)**
- 因特网传输层不提供:
  - 时延保证
  - 带宽保证



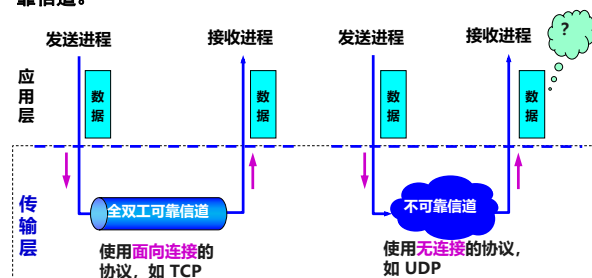
## Internet传输层协议

- 传输层向高层用户屏蔽了下面网络核心的细节，它使应用进程看见的就是好像在两个传输层实体之间有一条端到端的逻辑通信信道。



## Internet传输层协议

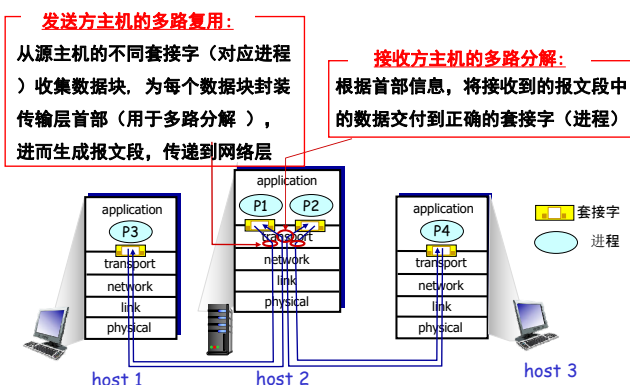
- 当传输层采用面向连接的TCP协议时，尽管IP网络是不可靠的（只提供尽最大努力服务），但这种逻辑通信信道就是一条全双工的可靠信道。
- 当传输层采用无连接的UDP协议时，这种逻辑通信信道是一条不可靠信道。



## 第三章 传输层

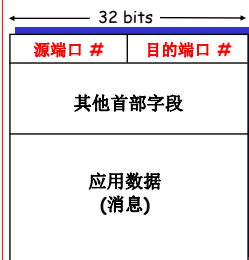
- 3.1 传输层服务
- 3.2 多路复用和多路分解
- 3.3 无连接运输:UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的运输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

## 多路复用/多路分解的工作过程



## 多路复用/分解的要求

- 套接字必须要有唯一标识符
- 每个报文段需要有特殊字段指明所要交付的接收方套接字。
  - 每个IP数据报都有源IP地址和目的IP地址
  - 每个IP数据报都携带一个传输层报文段
  - 每个传输层报文段都有一个源端口和目的端口
- 使用IP地址+端口号将报文段定向到适当的套接字（进程），为什么？



TCP/UDP 报文段格式

## 应用进程的标识：IP地址+端口号

- 运行在计算机中的进程是用进程标识符来标志的。但运行在应用层的各种应用进程却不应当让计算机操作系统指派它的进程标识符。因为不同的操作系统使用不同格式的进程标识符。为了使运行不同操作系统的应用进程能够互相通信，就必须用统一的方法对TCP/IP体系的应用进程进行标志。
- 由于进程的创建和撤销都是动态的，而且接收方有时会改换接收报文的进程，但并不需要也无法通知所有发送方，因此发送方也需要特定信息来识别接收方进程。
- 解决方案：使用端口号

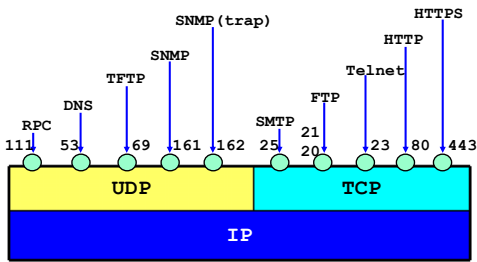
应用进程的标识：IP地址+端口号

- 端口用一个 16 位端口号进行标志，允许有65,535个不同的端口号。
- 端口号只具有本地意义，即端口号只是为了标识本计算机应用层中的各进程。
- 所以，两个计算机中的进程要互相通信，不仅必须知道对方的端口号（为了找到对方计算机中的应用进程），而且还要知道对方的 IP 地址（为了找到对方的计算机）。

端口号的分配

- 服务器端使用的端口号
  - 熟知端口，数值一般为 0 ~ 1023。
  - 登记端口号，数值为 1024 ~ 49151，为没有熟知端口号的应用程序使用的。使用这个范围的端口号必须在 IANA 登记，以防止重复。
- 客户端使用的端口号
  - 又称短暂端口号，数值为 49152 ~ 65535，留给客户进程选择暂时使用。通信结束后，这个端口号可供本机其他客户进程使用。
  - 当服务器进程收到客户进程的报文时，就知道了客户进程所使用的动态端口号。

常用的熟知端口

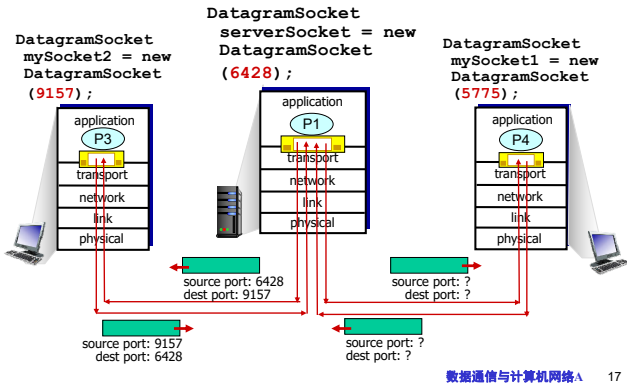


无连接多路分解

- 当主机接收UDP报文段：
  - 使用端口号生成套接字：

```
DatagramSocket mySocket1 = new DatagramSocket(12534);
```
  - 检查报文段中的目的端口号
  - 定向UDP报文段到使用该端口的套接字
- UDP 套接字采用二元组(目的IP地址、目的端口号)表示
  - 如果两个UDP报文段具有不同的源IP地址和/或源端口号，但具有相同的IP地址和目的端口号，则两个报文段将被定向到同一目的的主机的相同套接字

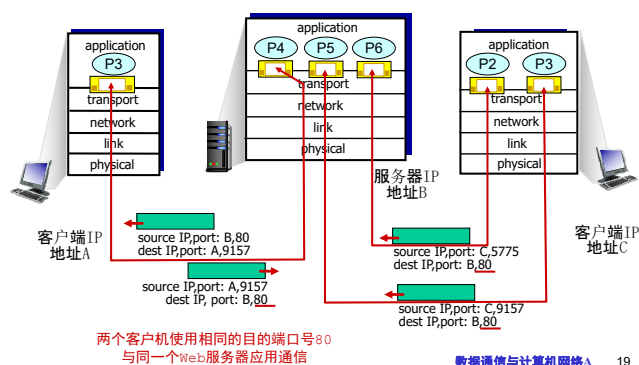
无连接多路复用/分解 (续)



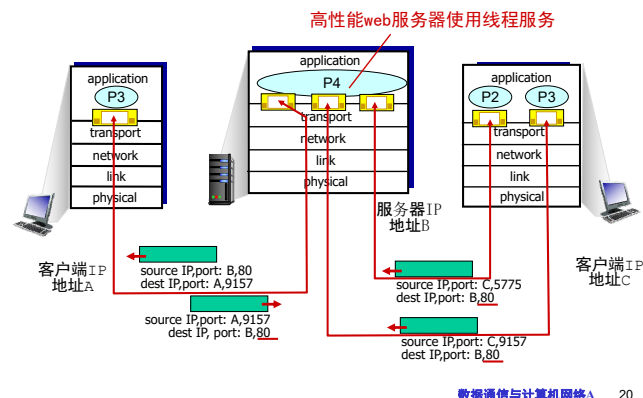
面向连接的多路分解/复用

- TCP 套接字由四元组标识：
  - 源IP地址
  - 源端口号
  - 目的IP地址
  - 目的端口号
- 分解：接收主机使用上述四个值将报文段多路分解到相应的套接字
- 服务器主机可同时支持多个并发的TCP套接字：
  - 每个套接字与一个进程关联，并由四元组来标识
- 对于每个连接的客户端，Web服务器使用不同的套接字：
  - 非持久 HTTP 将为每个请求使用不同的套接字

## 面向连接的多路复用/分解 (续)



## 面向连接的多路复用/分解 (续)



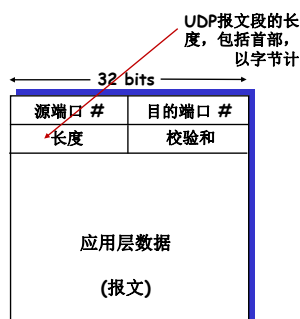
## 第三章 传输层

- 3.1 传输层服务
- 3.2 多路复用和多路分解
- 3.3 无连接传输:UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的传输: TCP
- 报文段结构
- 可靠数据传输
- 流量控制
- 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

## UDP: 用户数据报协议 [RFC 768]

- 提供“没有不必要的,”“基本要素的”Internet传输服务
- UDP 报文段可能:
  - 丢包
  - 对应用程序无序交付数据
- UDP的无连接:
  - UDP发送方和接收方之间无握手
  - UDP一次交付一个完整的应用层报文段,不分拆,也无需合并
- 常用于流式多媒体应用(丢包容忍、速率敏感)
  - DNS
  - SNMP
  - UDP的不可靠传输
    - 可在应用层增加可靠性
    - 可在应用层进行差错恢复!

## UDP: 用户数据报协议 [RFC 768]



### 为什么创建UDP协议?

- 无连接: 有连接的服务会增加时延
- 简单: 发送方和接收方之间无需维护连接状态
- 报文段首部减小: 提高路由器处理效率
- 无拥塞控制: UDP 能尽可能快的传输报文

## UDP 校验和

**目的:** 在传输的报文段中检查“差错”(例如: 比特翻转)

校验和的计算需要用到:

- 伪首部
- UDP首部
- UDP数据部分



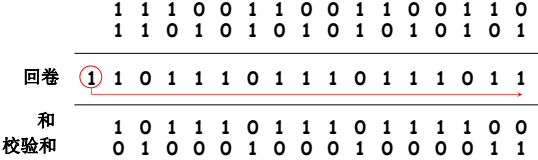
UDP 校验和

发送方: 计算校验和

- 校验和字段先置0
- 将伪首部、UDP首部、UDP数据部分以16bit为单位划分为“字”，数据部分不足16位补0
- 对16位“字”按位求和，最高位的进位必须加到最低位上，称为“回卷”
- 最后对16位和求反码，放入校验和字段。

回卷的例子

- 当进行按位求和运算时, 最高位的进位, 必须加到结果中



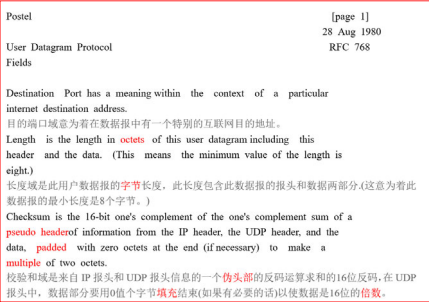
UDP 校验和

接收方: 检验校验和

- 对接收到的报文段的伪首部、UDP首部、UDP数据部分（包括校验和），按照16bit为单位划分，一起求和
- 如果分组无差错，接收方计算出的和将是16个1，如果一个比特是0，则可以知道分组出现了差错：
  - 可能还有差错吗？

UDP校验和补充

- rfc-768只规定UDP校验和如何计算，没规定校验和如何验算。具体实现方法属于操作系统协议栈内部的实现。所以自顶向下教材与谢希仁版教材对于校验和的演算，都正确。

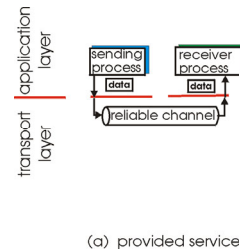


第三章 传输层

- 3.1 传输层服务
- 3.2 多路复用和多路分解
- 3.3 无连接运输:UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的运输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

可靠数据传输的原则

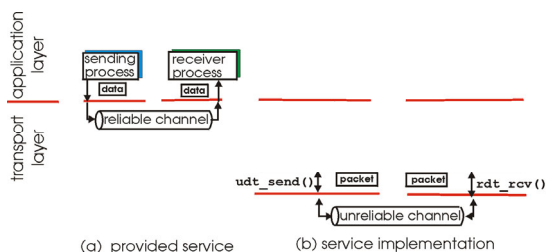
- 应用层、传输层、数据链路层中数据传输的重要性



- 不可靠信道决定了可靠数据传输协议的复杂性 (rdt)

## 可靠数据传输的原则

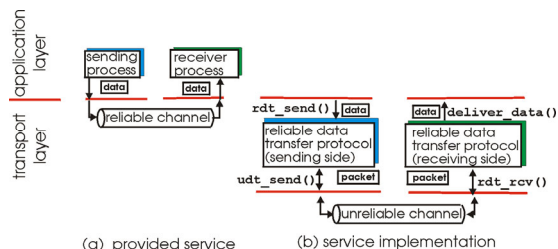
- 应用层、传输层、数据链路层中数据传输的重要性



- 不可靠信道决定了可靠数据传输协议的复杂性 (rdt)

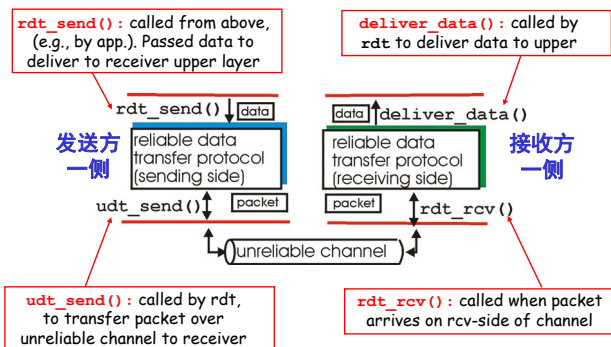
## 可靠数据传输的原则

- 应用层、传输层、数据链路层中数据传输的重要性



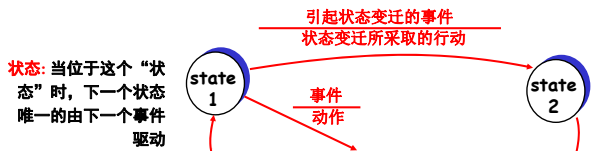
- 不可靠信道决定了可靠数据传输协议的复杂性 (rdt)

## 可靠数据传输：基本概念



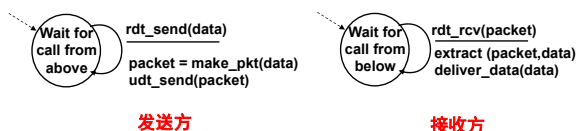
## 可靠数据传输：基本概念

- 进一步研究发送方、接收方侧的可靠数据传输协议
- 仅考虑单向数据传输，但控制信息在两个方向流动！
- 使用有限状态机 (FSM) 来定义发送方和接收方



## Rdt1.0: 经可靠信道的可靠传输

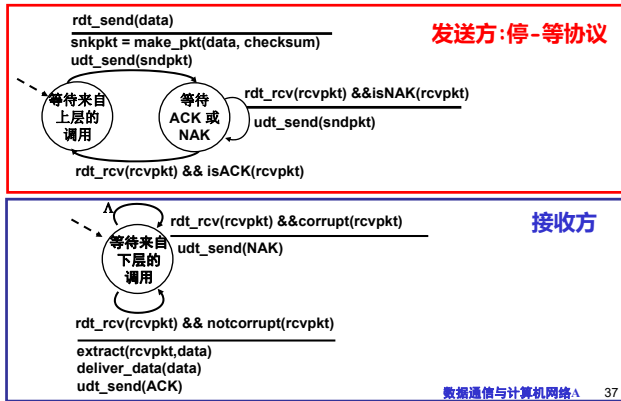
- 假设底层信道非常可靠
  - 无比差错、无分组丢失
- 发送方和接收方各自独立的状态机:
  - 发送方将数据发向底层的信道
  - 接收方从底层信道读取数据



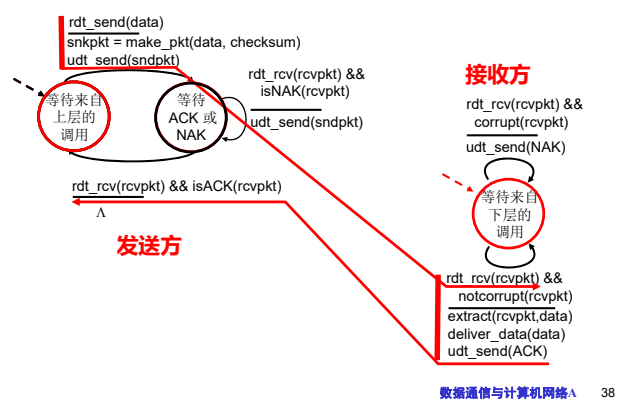
## Rdt2.0: 具有比特差错的信道

- 底层信道可能导致报文段中比特反转
  - 校验和检查比特差错
- 如何从差错中恢复?
  - 确认 (ACKs): 接收方明确告知发送方分组被正确接收
  - 否定确认 (NAKs): 接收方明确告知发送方分组接收有误
  - 发送方收到NAK消息后，重传有误的分组
- rdt2.0 采用与rdt1.0不同的机制:
  - 差错检测: 分组中使用校验和字段
  - 接收方反馈: 反馈消息(ACK, NAK) 由接收方->发送方

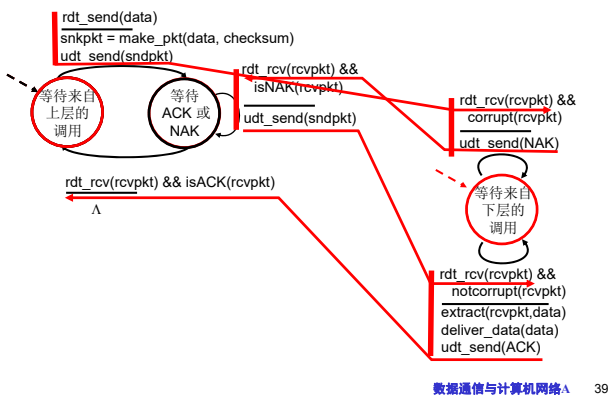
## rdt2.0: FSM 详解



## rdt2.0: 无差错时的操作



## rdt2.0: 有差错时的操作



## rdt2.0 存在一个致命的缺陷!

## 如果ACK/NAK 分组受损怎么办?

- ❑ 发送方如果收到受损的ACK/NAK，它无法知道接收方是否正确接收了上一块发送的数据！
- ❑ 解决思路1：在ACK/NAK中增加足够的检验和比特，使发送方不仅可以检测比特差错，还可以**纠正比特差错**
  - ❑ 要求传输信道仅产生差错但不会丢失分组
- ❑ 解决思路2：发送方重发这块数据分组？
  - ❑ 接收方不知道ACK/NAK是否被发送方正确收到，因此它**无法知道接收到的分组是新的分组，还是一个重发的分组**

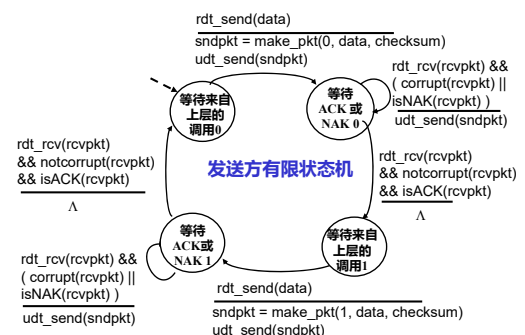
## 对rdt2.0的改进

### 对于解决思路2，如何处理冗余分组？

- 发送方在每个分组中增加一个“序号”字段
  - 发送方收到受损的 ACK/NAK 消息，或者正确的 NAK 消息时，重发这个受损消息或正确 NAK 消息对应的数据分组
  - 接收方检查收到的分组序号，即可确认分组是否是一次重传，接收方将直接丢弃冗余的分组
- 对于 rdt2.0 这种简单的停-等协议**
- 停等协议

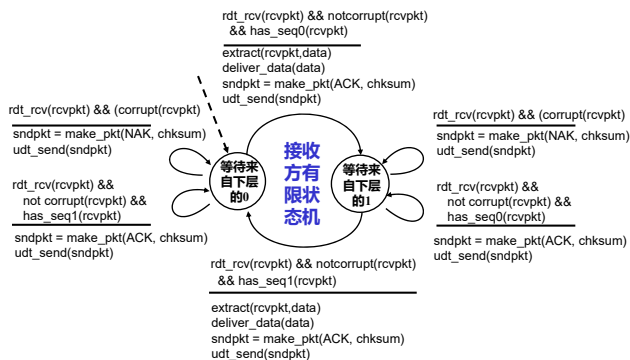
发送方发送一个分组后，然后等待接收方的应答
- 因为假设信道不丢失分组
  - 所以，只要使用 1bit 的分组序号即可

### rdt2.1: 发送方, 处理受损的ACK/NAK





## rdt2.1: 接收方, 处理受损的ACK/NAK



## rdt2.1: 总结

### 发送方:

- 分组新增了序列号
- 使用0, 1两个序列号即可
- 必须检查是否接收到受损的ACK/NAK
- 状态增加一倍
  - 状态必须能反映出当前正在发送的分组的序列号是0还是1

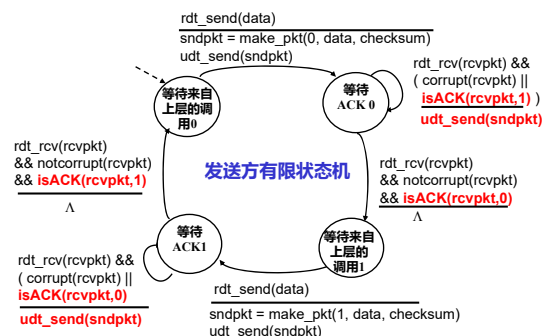
### 接收方:

- 必须检查接收到的分组是否冗余
  - 状态反映出希望接收的分组的序号是0还是1
- 注意: 接收方不可能知道最新发送的ACK/NAK是否已被发送方正确接收

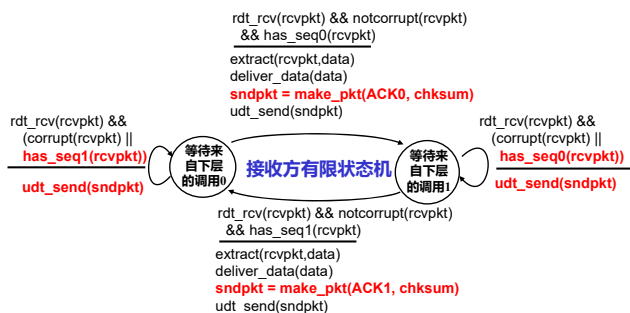
## 对rdt2.1的改进: 没有NAK确认的rdt2.2协议

- 改进策略:
  - 接收方对最后一次接收到的正确分组回复ACK
  - 接收方必须在ACK中加上被确认分组的序列号
- 结论: 对同一个分组的冗余ACK将实现与NAK一样的功能
  - 因为发送方认为, 跟在被确认两次的分组之后的分组, 接收方没有正确接收, 发送方将重发该分组
  - 而对于接收方, 可以由序号判断是否为冗余分组

## rdt2.2: 发送方



## rdt2.2: 接收方



## rdt3.0: 具有比特差错和丢包的信道

- 假设下层信道也会丢失分组 (数据或 ACK)
- 如何检测丢包? 丢包后如何处理?
  - rdt2.2协议中的校验和、序列号、ACK确认、重发等, 已经可以对丢包进行处理
  - 但要检测是否丢包, 需要增加新的协议机制

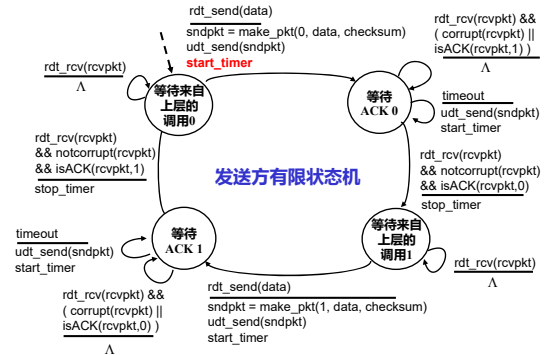


## rdt3.0: 具有比特差错和丢包的信道

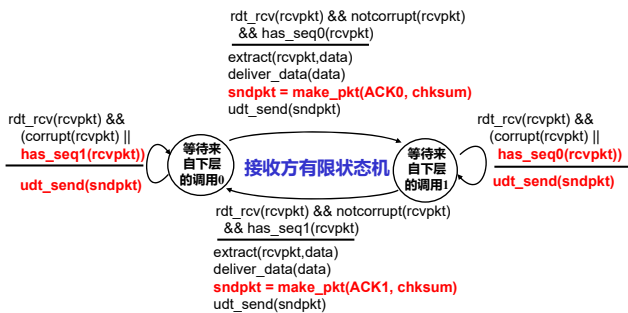
**解决方法:** 让发送方负责检测丢包和丢包后的恢复

- 发送方等待ACK一段“合理的”时间
- 因此需要**倒计时定时器**
- 如果在这段时间内发送方没有收到ACK, 则重发分组
- 如果数据分组(或ACK)只是延迟(没有丢失):
  - 重发将导致冗余, 因此使用分组序号可以解决分组冗余
  - 接收方必须在ACK中指定已被确认接收的分组的序号

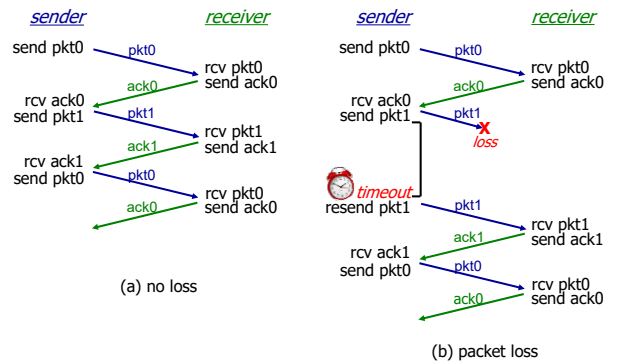
## rdt3.0 发送方



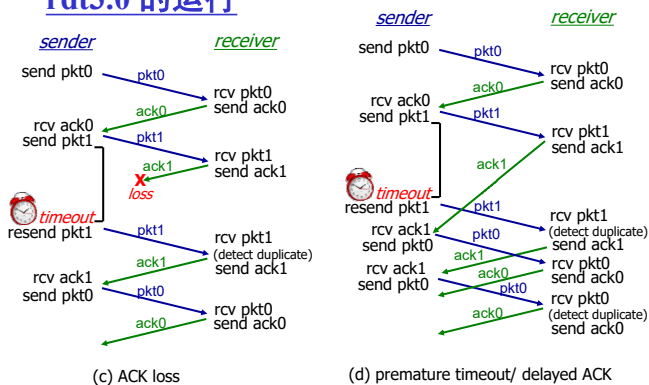
## rdt3.0: 接收方



## rdt3.0 的运行



## rdt3.0 的运行

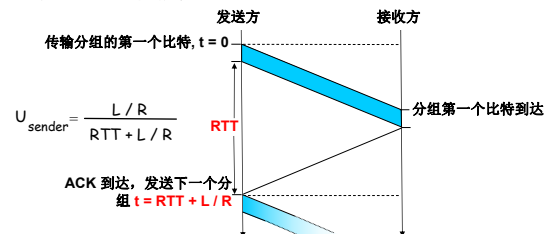


## 关于rdt3.0 的几点说明

- 发送方发送完一个分组后, 必须**暂时保留**已发送的分组的副本, 以备重发。
- **分组和确认都必须进行编号。**
- 超时计时器的超时时间设定, 应当比数据在分组传输的平均往返时间**更长一些**。
- 发送方最终总是可以收到对所有发出的分组的确认。如果发送方不断重发分组但总是收不到确认, 就说明通信线路太差, 不能进行通信。
- 这种可靠传输协议常称为**自动重传请求 ARQ (Automatic Repeat reQuest)**。意思是重发的请求是自动进行的, 接收方不需要请求发送方重发某个出错的分组。

## rdt3.0的性能

- rdt3.0 是一个功能正常的协议，但它是一个停-等协议，这使得rdt3.0性能欠佳



- $U_{\text{sender}}$ : 利用率 - 发送方用于发送的时间比值

## rdt3.0的性能

- 例如：1 Gbps链路, 15 ms 端-端传播时延, 1KB分组

$$T_{\text{transmit}} = \frac{L_{\text{bits}} (\text{分组长度})}{R_{\text{bps}} (\text{传送速率})} = \frac{8\text{kb/pkt}}{10^9 \text{ b/sec}} = 8 \text{ microsec}$$

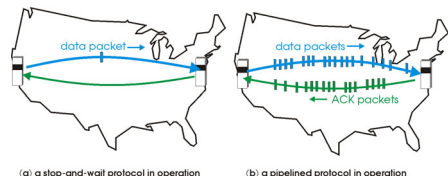
$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

- 如果往返时延RTT为30msec，对于这种每30msec一个1KB分组，意味着在1Gbps链路上仅有33kB/sec 吞吐量
- 网络协议限制了物理资源的使用！

## 流水线协议

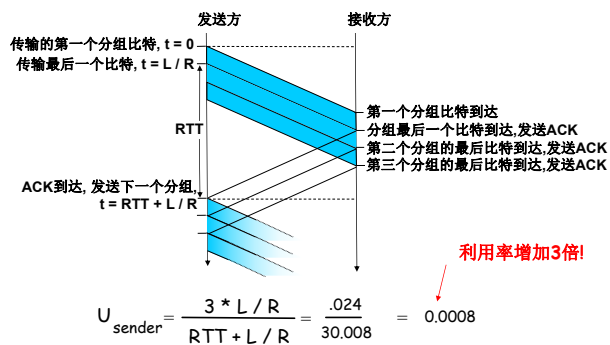
**流水线:** 发送方允许发送多个分组而无需等待确认

- 必须增加序号范围
- 协议的发送方和接收方必须缓存多个分组



- 流水线协议的两种形式: 回退N帧(go-Back-N), 选择性重传(selective repeat)

## 流水线协议: 增加利用率



## 流水线技术要求:

- 必须增加序号范围
  - 因为允许一次发送多个分组
- 发送方和接收方可能需要缓存多个分组
  - 发送方: 保存已发送但未被确认的分组
  - 接收方: 保存乱序到达的分组, 保证向上层按序提交
- 解决丢包、损坏、延时到达, 有两种基本处理方法:
  - 回退N帧(Go-Back-N)
  - 选择性重发(Selective Repeat, SR)

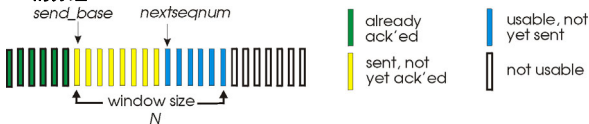
## 流水线协议的两种基本差错恢复方法

- 回退N帧(Go-Back-N)
  - 发送方在管道中最多可以有N个未经确认的分组
  - 接收方仅发送“累积”确认
  - 发送方对已发送未确认的第一个分组维护一个定时器
    - 定时器超时后, 重发所有未被确认的分组
- 选择性重发
  - 发送方在管道中最多可以有N个未经确认的分组
  - 接收方对每一个分组发送单独的确认
  - 发送方对每一个已发送未确认的分组维护一个定时器
    - 定时器超时后, 仅重发对应的那个未被确认的分组

## 回退N帧(Go-Back-N)

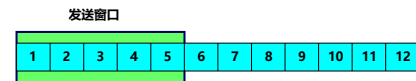
**发送方:**

- 分组需要 $K$ 比特序号, 因此序号范围是 $[0, 2^K-1]$
- “发送窗口”值为 $N$ , 即发送方最多允许缓存 $N$ 个连续的没有得到确认的分组

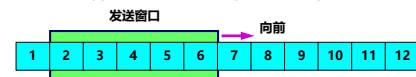


- ACK(n): 表示对所有序号 $\leq n$ 分组的确认，即“累计确认”
  - 可能收到冗余的多个确认 (见接收方)
- 对每个传输中的分组使用同一个计时器，该定时器对应最早被发送出去但未被确认的分组，该分组序号记为 `sendbase`
- `timeout(n)`: 超时，需要重发 `sendbase` 分组，及之后  $n-1$  个序号比 `sendbase` 大，已发送未被得到确认的分组

## 回退N帧(Go-Back-N)

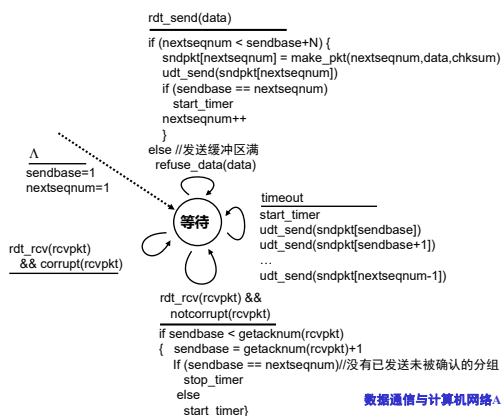


(a) 发送方维持发送窗口 (发送窗口N=5)

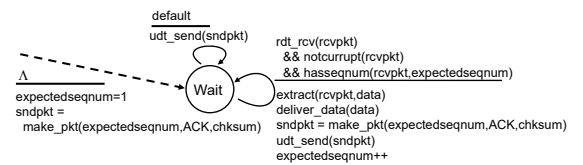


(b) 收到对编号1的分组的确认后, 发送窗口向前滑动

## GBN: 发送方扩展FSM



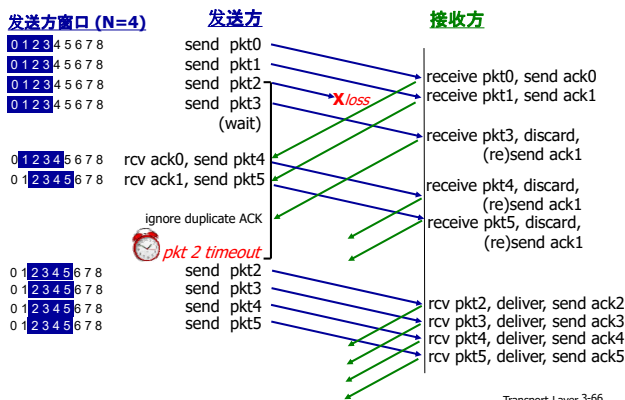
## GBN: 接收方扩展 FSM



- ❑ 对校验正确, 序号=期望序号(expectedseqnum)的分组:
  - ❑ 发送序号为expectedseqnum的确认, 告诉发送方到这个分组为止的之前分组都已正确收到。
  - ❑ 缺点: 可能产生冗余ACK
- ❑ 其余情况: 例如失序分组(提前到达或冗余)、校验和错误分组
  - ❑ 丢弃, 使用expectedseqnum序号重发确认

## GBN 操作

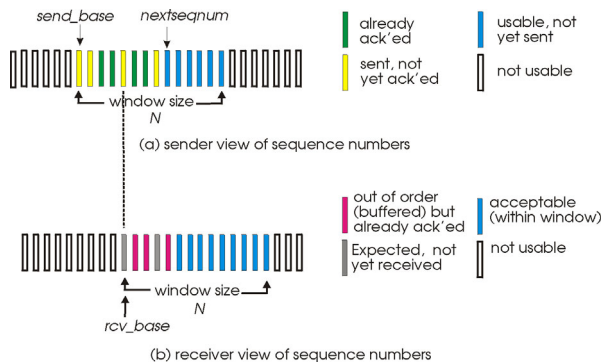
**发送方窗口 (N=4)**



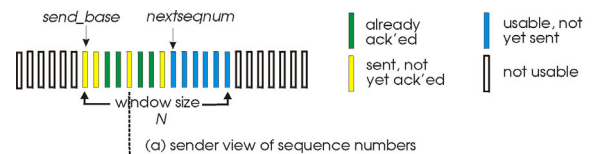
## 回退N帧(Go-Back-N)的优缺点

- 优点：接收方丢弃所有失序分组，简单
- 缺点：发送方必须重发那些失序分组(哪怕这些分组正确到达接收方)，但重发可能丢失或出错，而需要更多重发，当通信线路质量不好时，GBN 协议会带来负面的影响。

## 选择性重传: 发送方,接收方窗口



## 选择性重传: 发送方,接收方窗口



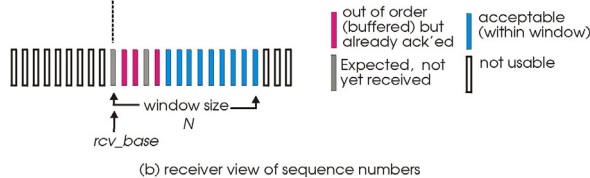
对于发送方

- 上层有数据需要发送: 如果发送窗口未满, 发送分组, 缓存未收已发送未收到确认的分组
- timeout(n): 重发分组 n, 重启定时器
- ACK(n) 落在  $[send\_base, send\_base + N]$ : 表明序号为 n 的分组已经收到, 如果  $n = send\_base$ , 向前滑动窗口 sendbase 指针到是当前最小的未被确认的分组序号

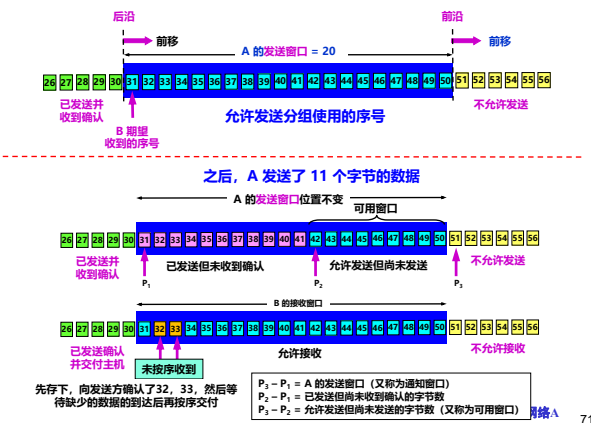
## 选择性重传: 发送方,接收方窗口

对于接收方

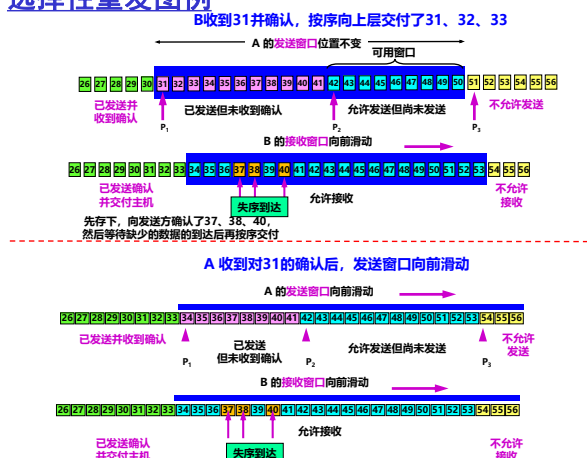
- 分组 n 落在  $[rcv\_base, rcv\_base + N - 1]$  且被正确接收: 发送 ACK(n)
  - 新分组: 缓存
  - 分组序号 = rcv\_base: 该分组与 rcv\_base 序号之后的, 之前缓存的连续分组, 交付到上层, 向前滑动窗口到下一个未收到分组的序号
- 分组 n 落在  $[rcv\_base - N, rcv\_base - 1]$  且被正确接收: ACK(n), 表明这是之前已经确认过的分组
- 其他: 忽略



## 选择性重发图例



## 选择性重发图例



## 选择性重发的操作

发送方窗口 (N=4)

发送方

send pkt0  
send pkt1  
send pkt2  
send pkt3 (wait)

rcv ack0, send pkt4  
rcv ack1, send pkt5

record ack3 arrived

pkt 2 timeout  
send pkt2

record ack4 arrived  
record ack5 arrived

接收方

receive pkt0, send ack0  
receive pkt1, send ack1

receive pkt3, buffer, send ack3

receive pkt4, buffer, send ack4  
receive pkt5, buffer, send ack5

rcv pkt2; deliver pkt2, pkt3, pkt4, pkt5; send ack2

Q: what happens when ack2 arrives?

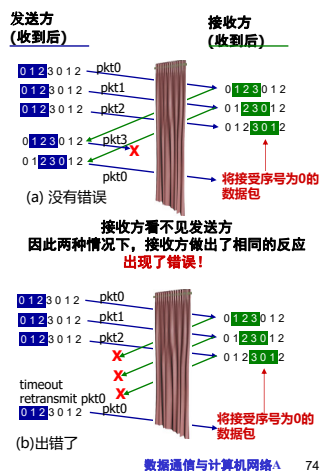
## 选择性重发：难点

例如：

- 序号：0, 1, 2, 3
- 窗口尺寸=3
- 在 (b) 情况下，将发送方发送的冗余分组0当成新分组收下来；

**Q:** 为了避免出现b的错误，序号长度与窗口尺寸有什么关系？

**A:** 窗口长度 ≤ 序号空间的一半



## 第三章 传输层

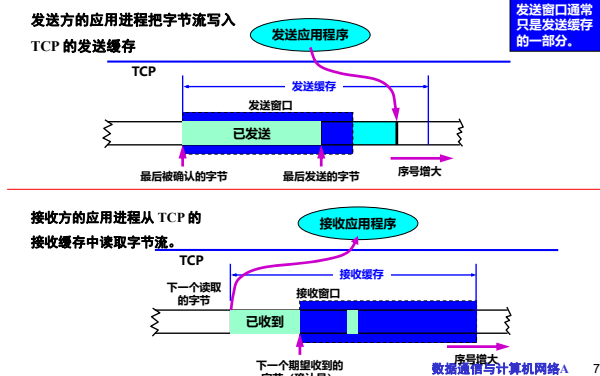
- 3.1 传输层服务
- 3.2 多路复用和多路分解
- 3.3 无连接运输:UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的运输: TCP
- 报文段结构
- 可靠数据传输
- 流量控制
- 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

## TCP: 概述 RFCs: 793, 1122, 1323, 2018, 2581

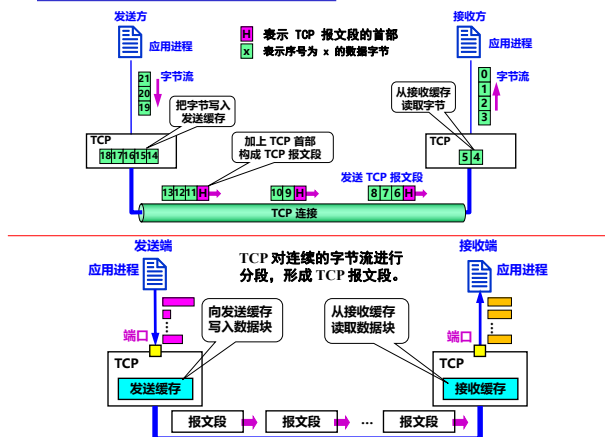
- 点到点全双工通信：
  - 一个发送方，一个接收方，同一连接上的双向数据流，MSS:最大报文段长度，MTU: 最大传输单元
- 面向连接：
  - 在进行数据交换前，初始化发送方、接收方状态，进行握手(交换控制信息)
- 流量控制：
  - 发送方不能“淹没”接收方
- 拥塞控制：
  - 抑制发送方速率来防止过分占用网络资源

## TCP: 概述 RFCs: 793, 1122, 1323, 2018, 2581

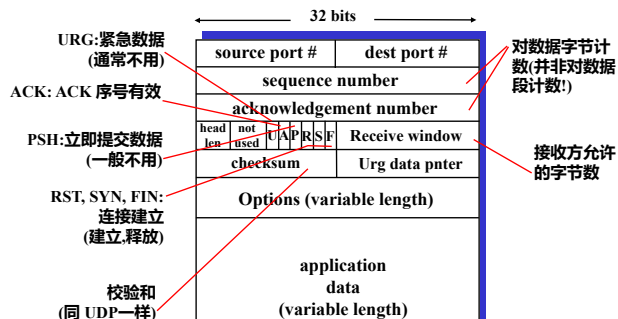
- 可靠、有序的字节流(stream)



## TCP 面向流的概念



## TCP 报文段结构



## TCP 序列号, 确认

### 序列号:

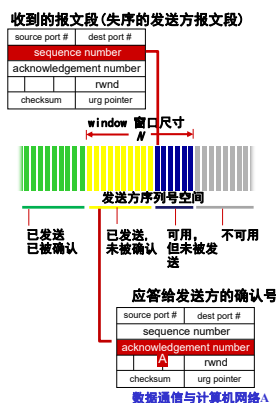
- 报文段中第1个数据字节在字节流中的位置编号

### 确认:

- 期望从对方收到的下一个字节的序号
- 累积确认

Q: 接收方如何处理失序报文段?

A: TCP规范没有给出明确说明, 由实现者自行选择实现方法 - 抛弃或者缓存



## TCP 序号和确认号

### 序列号:

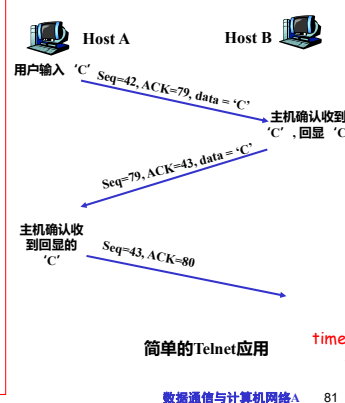
- 报文段中第1个数据字节在字节流中的位置编号

### 确认:

- 期望从对方收到的下一个字节的序号
- 累积确认

Q: 接收方如何处理失序报文段?

A: TCP规范没有给出明确说明, 由实现者自行选择实现方法 - 抛弃或者缓存



## TCP 往返时延(RTT)和超时

Q: 如何设置TCP超时值?

- $\geq RTT$

- 但 RTT是变化的

- 太小: 过早超时

- 导致不必要的重传

- 太大: 对报文段的丢失响应太慢

- 使用Karn算法, 引入平均往返时延  $EstimatedRTT$  来计算超时重传时间  $TimeoutInterval$

## TCP 往返时延(RTT)和超时

Q: Karn算法如何计算平均往返时延?

- SampleRTT: 实际测量的, 发送报文到接收到ACK的时间

- 重发的报文不参与测量

- SampleRTT 变化较大, 希望估计的 RTT 较为“平滑”

- 根据最近的SampleRTT, 计算平均往返时延如下:

$$EstimatedRTT(i) = (1-\alpha) * EstimatedRTT(i-1) + \alpha * SampleRTT(i)$$

其中  $EstimatedRTT(1) = SampleRTT(1)$

## TCP 往返时延和超时(续)

Q: 为什么重发报文不参与计算平均往返时延?



## TCP 往返时延和超时(续)

Q: Karn算法如何计算超时重发时间  $TimeoutInterval$ ?

- ①  $TimeoutInterval(i) = EstimatedRTT(i) + 4 * DevRTT$

- DevRTT用于估计SampleRTT偏离EstimatedRTT的程度, 也用作安全余量来计算超时时间隔

- ②  $EstimatedRTT(i) = (1-\alpha) * EstimatedRTT(i-1) + \alpha * SampleRTT(i)$

$EstimatedRTT(1) = SampleRTT(1)$ ,  $\alpha$ 一般取0.125

- ③  $DevRTT(i) = (1-\beta) * DevRTT(i-1) + \beta * |EstimatedRTT(i) - SampleRTT(i)|$

$DevRTT(1) = SampleRTT(1) / 2$ ,  $\beta$ 的推荐值为0.25



谢版教材:  $RTT_s = EstimatedRTT$   
 $RTT_d = DevRTT$   
 $RTO = TimeoutInterval$



## TCP 往返时延和超时(续)

**Q:** 当报文段的时延突然增大了很多, 导致在原来计算的重发时间内不会收到确认报文, 于是发送方重发报文段。而根据 Karn 算法, 不考虑重发报文段的 SampleRTT。超时重发时间无法更新.....

**A:** 引入修正的Karn算法。报文段每重传一次，利用下式计算：

$$\text{TimeoutInterval}(i) = \gamma * \text{TimeoutInterval}(i)$$

$\gamma$  的典型值是 2 。

- 当不再发生报文段的重传时，才根据报文段的往返时延更新平均往返时延和超时重发时间。

### 第三章 传输层

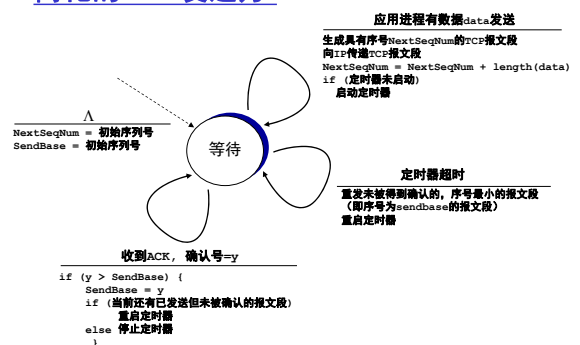
- 3.1 传输层服务
- 3.2 多路复用和多路分解
- 3.3 无连接传输:UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的运输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

## TCP可靠数据传输

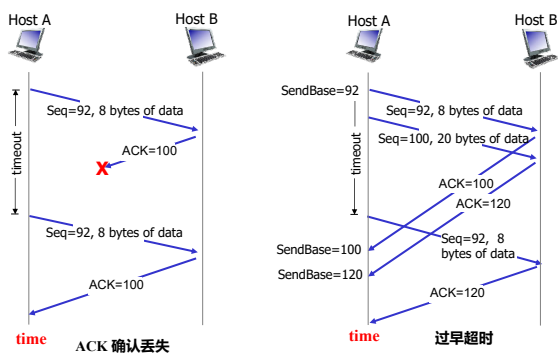
- TCP 在IP不可靠服务基础上创建可靠数据传输
  - 流水线发送
  - 采用以字节为单位的滑动窗口控制发送方和接收方缓存
  - 选择性重发，TCP 对多个报文使用单个重发定时器
  - 累积确认

- 下列事件触发重传:
  - 超时
  - 冗余的ACK确认
- 首先考虑一个简化的TCP发送方:
  - 忽略重复的ACK
  - 不考虑拥塞控制

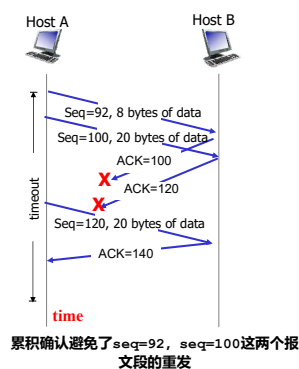
### 简化的TCP发送方:



## TCP: 重发的情况



## TCP 重发的情况 (续)





## 产生TCP ACK的建议 [RFC 5681]

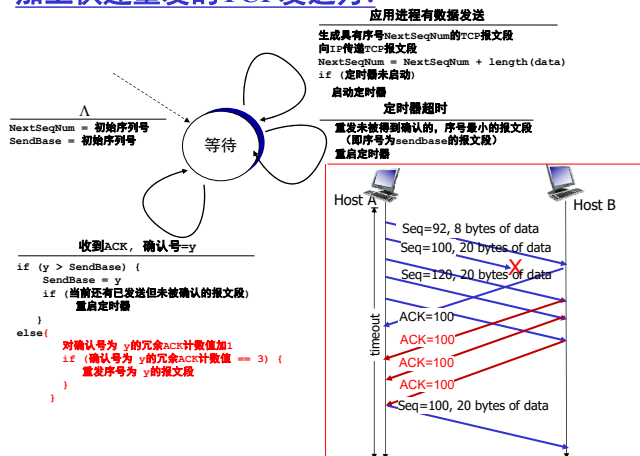
| 接收方事件                                 | TCP 接收方行为   |
|---------------------------------------|---|
| 所期望序号的报文段按序到达。<br>所有在期望序号及以前的数据都已经被确认 | 延迟的ACK。对另一个按序报文段的到达最多等待500 ms。如果下一个按序报文段在这个时间间隔内没有到达，则发送一个ACK |
| 有期望序号的报文段按序到达。<br>另一个按序报文段等待发送ACK     | 立即发送单个累积ACK，以确认两个按序报文段  |
| 比期望序号大的失序报文段到达，<br>检测出数据流中的间隔。        | 立即发送冗余ACK，指明下一个期待字节的序号（也就是间隔的低端字节序号）                          |
| 部分或者完全填充已接收到数据间隔的报文段到达                | 倘若该报文段起始于间隔的低端，则立即发送ACK                                       |

网络通信与计算机网路 97

## TCP快速重发

- ❑ 超时时间通常相对较长:
  - ❑ 重发丢失分组之前会等待较长时延
- ❑ 发送方可以在超时事件发生之前, 通过冗余ACK检测丢包情况:
  - ❑ 发送方经常一个接一个的发送报文段
  - ❑ 如果报文段丢失, 发送方将会收到很多重复ACK.
- ❑ 因此, 如果发送方收到对同一报文的3个冗余ACK, 就可以认为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, 不必等待定时器超时, **快速重发**

### 加上快速重发的TCP发送方:



### 课间练习:

P27. 主机 A 和 B 经一条 TCP 连接通信, 并且主机 B 已经收到了来自 A 的最长为 126 字节的所有字节。假定主机 A 随后向主机 B 发送两个紧接的报段的报文。第一个和第二个报文段分别包含了 80 字节和 40 字节的数据。在第一个报文段中, 序号是 127, 源端口号是 302, 目的端口号是 80。无论何时主机 B 接收到来自主机 A 的报文段, 它都会发送确认。

从主机 A 发往 B 的第二个报文段中, 序号、源端口号和目的端口号各是什么?

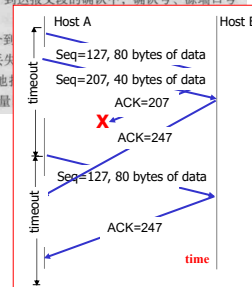
如果第一个报文段在第二个报文段之前到达, 在第一个到达报文段的确认中, 确认号、源端口号和目的端口号各是什么?

如果第二个报文段在第一个报文段之前到达, 在第一个到达报文段的确认中, 确认号、源端口号和目的端口号各是什么?

假定由 A 发送的两个报文段按序到达 B。第一个确认丢失。画出此时序图, 显示这些报文段和发送的所有其他丢失。对于图上每个报文段, 标出序号和数据的字节数量。

```
sequenceDiagram
    participant A as Host A
    participant B as Host B
    Note over A: Seq=127, 80 bytes of data
    A->>B: Seq=127, 80 bytes of data
    Note over B: Seq=127, 80 bytes of data
    B->>A: ACK=207
    Note over A: Seq=207, 40 bytes of data
    A->>B: Seq=207, 40 bytes of data
    Note over B: Seq=207, 40 bytes of data
```

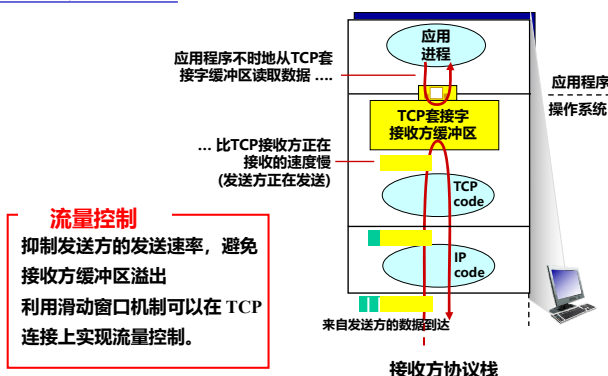
- a) 序列号207, 源端口号302, 目的端口号80.  
b) 确认号207, 源端口号80, 目的端口号302.  
c) 确认号127  
d) 如右图



### 第三章 传输层

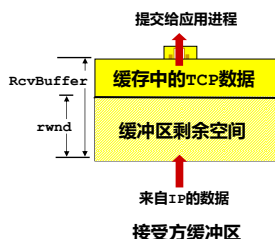
- 3.1 传输层服务
- 3.2 多路复用和多路分解
- 3.3 无连接运输:UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的运输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

## TCP 流量控制



## TCP 流量控制:工作原理

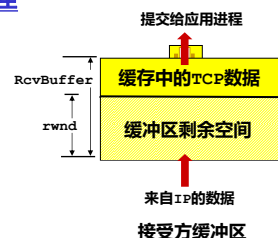
- 接收方把当前的 $rwnd$ 值放入TCP报文段首部的接收窗口字段中,通知发送方其接收缓冲区的RcvBuffer的剩余空间
  - RcvBuffer值通过套接字选项设置,通常默认为4096字节
  - 许多操作系统会自动调整RcvBuffer
  - 初始时 $rwnd=RcvBuffer$
  - 假设接收方并不缓存失序报文段



数据通信与计算机网络A 103

## TCP 流量控制:工作原理

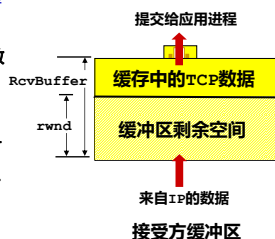
- LastByteRead: 接受方应用进程从缓存读取的数据流的最后一个字节的编号
- LastByteRecv: 从网络到达并已放入接收方缓存中的数据流的最后一个字节的编号
- 由于不允许缓存溢出,因此有: $LastByteRecv - LastByteRead \leq RcvBuffer$
- 接收窗口 $rwnd = RcvBuffer - [LastByteRecv - LastByteRead]$



数据通信与计算机网络A 104

## TCP 流量控制:工作原理

- 发送方轮流跟踪两个变量:
  - LastByteSent: 发送方发送的数据流的最后一个字节的编号
  - LastByteAcked: 发送方已发送且已被确认的最后一个字节的序号
- 显然,必须限制发送方的已发送但未确认的数据不能超过 $rwnd$ ,以保证接收方缓冲区不溢出,即: $LastByteSent - LastByteAcked \leq rwnd$



数据通信与计算机网络A 105

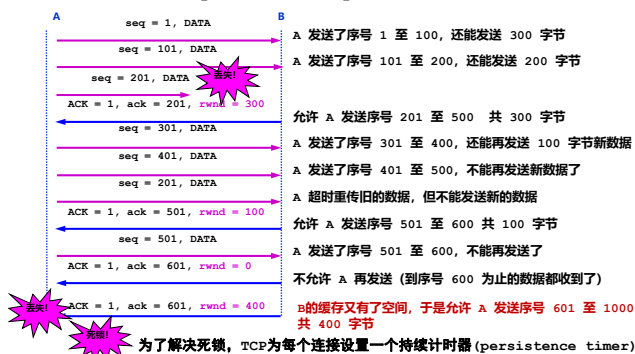
## TCP 流量控制:图例

A向B发送数据。建立TCP连接时B告诉A:“我的接收窗口 $rwnd=400B$ ”  
注意: A的 $LastByteSent - LastByteAcked \leq rwnd$



## TCP 流量控制:图例

A向B发送数据。建立TCP连接时B告诉A:“我的接收窗口 $rwnd=400B$ ”  
注意: A的 $LastByteSent - LastByteAcked \leq rwnd$

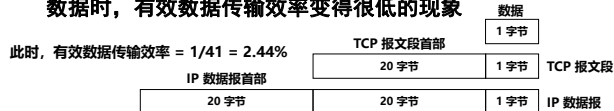


## TCP 流量控制:解决死锁

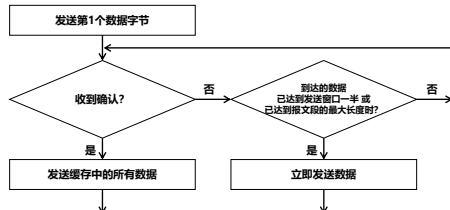
- 为了解决死锁问题, TCP 为每一个连接设有一个持续计时器(persistence timer)。
- 只要TCP连接的一方收到对方的零窗口通知, 就启动该持续计时器。
- 若持续计时器设置的时间到期, 就发送一个零窗口探测报文段(仅携带 1 字节的数据), 而对方就在确认这个探测报文段时给出了现在的窗口值。
- 若窗口仍然是零, 则收到这个报文段的一方就重新设置持续计时器。
- 若窗口不是零, 则死锁的僵局就可以打破了。

## TCP 流量控制:糊涂窗口综合症

- 糊涂窗口综合症: 每次仅发送一个字节或很少几个字节的数据时, 有效数据传输效率变得很低的现象

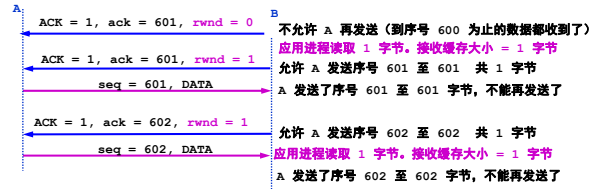


- 解决方法: 使用 Nagle 算法。



## TCP 流量控制:糊涂窗口综合症

- 例如: 接收方糊涂窗口综合症



- 解决方法: 让接收方等待一段时间, 使得: 或者接收缓存已有足够空间容纳一个最长的报文段; 或者等到接收缓存已有一半空闲的空间。
- 只要出现这两种情况之一, 接收方就发出确认报文, 并向发送方通知当前的窗口大小。

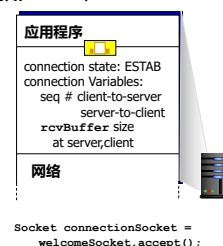
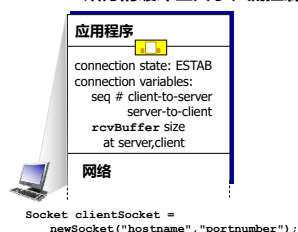
## 第三章 传输层

- 3.1 传输层服务
- 3.2 多路复用和多路分解
- 3.3 无连接运输:UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的运输: TCP
- 报文段结构
- 可靠数据传输
- 流量控制
- 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

## TCP 连接管理

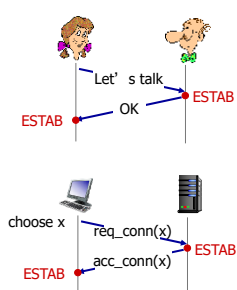
回顾: TCP发送方、接收方在交换报文段之前必须“握手”

- 同意建立连接
- 协商连接参数
  - 双方的初始序列号
  - 双方的缓冲区大小、流控制信息(例如: rwnd)



## TCP 连接管理——建立连接

### 2次握手:

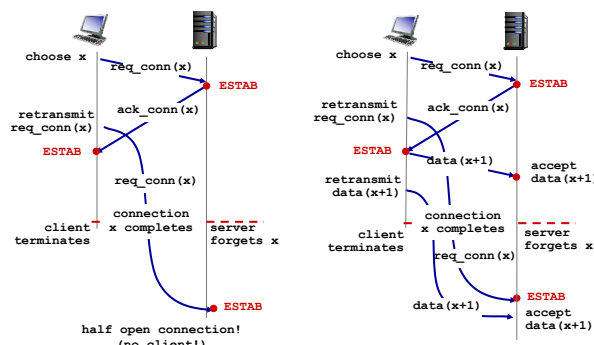


Q:2次握手总是能正常工作吗?

- 网络时延可变
- 消息重发 (例如: 丢包)
- 消息需要重新排序
- 无法看到另一侧

## TCP 连接管理——建立连接

导致2次握手协议失败的场景:

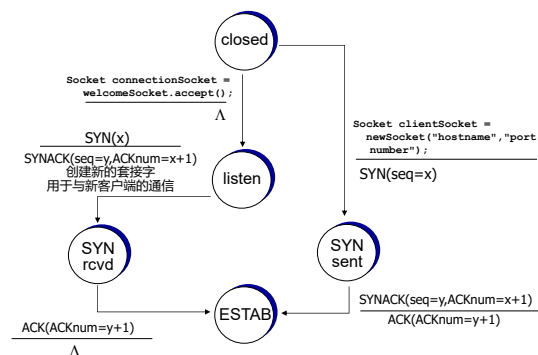


## TCP 连接管理——建立连接

### 三次握手:客户端对服务器应答做再次确认

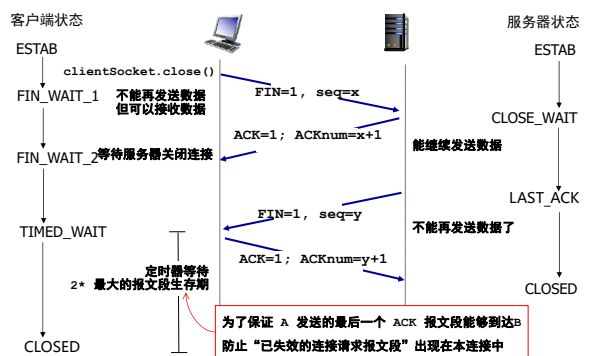


## TCP 连接管理



Transport Layer 3-116

## TCP 连接管理——关闭连接



Transport Layer 3-117

## 第三章 传输层

- 3.1 传输层服务
- 3.2 多路复用和多路分解
- 3.3 无连接运输:UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的运输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理(略)
- 3.7 TCP 拥塞控制

数据通信与计算机网络 119

## 第三章 传输层

- 3.1 传输层服务
- 3.2 多路复用和多路分解
- 3.3 无连接运输:UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的运输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理(略)
- 3.7 TCP 拥塞控制

数据通信与计算机网络 127

## 拥塞控制的两类方法

### 网络辅助的拥塞控制:

- 路由器为端系统提供反馈
  - 用一个比特位指出链路出现拥塞 (IBM SNA, DECnet, ATM)
- 指示发送方按照一定速率发送

### 端到端拥塞控制:

- 由于无法从网络得到明确的反馈, 因此端系统只能根据观察到的时延和丢包现象, 推断出现拥塞
- 这种拥塞控制策略被 TCP 协议所采用
- TCP让每一个发送方根据所感知的网络拥塞程度来限制其能发送流量的速率

数据通信与计算机网络 128

TCP 拥塞控制细节

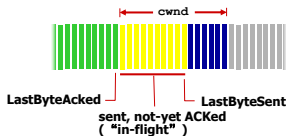
必须解决三个问题：

- TCP发送方如何限制发送速率？
- 发送方如何感知从它到目的地之间的路径上存在拥塞？
- 当发送方感知到拥塞发生时，采用何种算法解决拥塞？

ps：为了更加关注拥塞控制，我们假设TCP接受方缓存足够大，即接收窗口rwnd的值不受限制。

TCP拥塞控制: 细节

发送方如何限制发送速率？

- 发送方使用一个状态变量——LastByteAckedLastByteSent (sent, not-yet ACKed)
- 拥塞窗口cwnd来限制传输：
  - 发送方已发送但未被确认的报文数量 ≤ min{cwnd, rwnd}
  - 当rwnd不受限时，可描述为：
$$LastByteSent - LastByteAcked \leq cwnd$$
- 拥塞窗口cwnd是一个与网络拥塞情况相关的变量
- 因此动态调整cwnd的值，就能调整发送方的发送速率。
- 忽略不计丢包和发送时延，粗略地：

发送速率 =  $\frac{cwnd}{RTT}$  字节/秒

TCP拥塞控制: 细节

发送方如何感知到网络发生拥塞？

- 定义丢包事件为：超时 或 收到 3个重复的ACK
  - 出现过度拥塞时，发送方到接受方路径上的一台或多台路由器缓存会溢出，引起数据报被丢弃，发送方将“看到”冗余的ACK或者ACK超时未达
- 结论：当TCP发送方检测到丢包事件后，意识到网络发生拥塞。

TCP拥塞控制: 细节

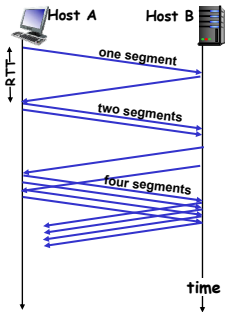
发送方采用何种算法解决拥塞？

- 丢包发生后，应当降低发送方的发送速率（减小拥塞窗口）
- 对先前未确认报文段(已发送但未被确认的)的确认到达时，应该增加发送方发送速率（增加发送窗口）
- 算法：增加发送速率以响应正确到达的ACK，探测可用带宽，直到检测到丢包发生，包括3个主要部分
  - 慢启动
  - 拥塞避免
  - 快速恢复

TCP 慢启动

当建立TCP连接时，以指数速度增加发送速率：

- 拥塞窗口cwnd初始值 = 1 MSS
- 每收到一个对先前未确认报文段的确认(新确认)，cwnd= cwnd+1MSS，这样每经过一个传输轮次(RTT)，cwnd加倍，呈指数增长
- 总结：初始发送速率很低，但能以指数速度快速增加
- PS：为了叙述方便，使用TCP报文段的个数作为窗口大小的单位，而TCP报文段通常按MSS字节来组装。关于cwnd真正初值设置，见谢版教材P232

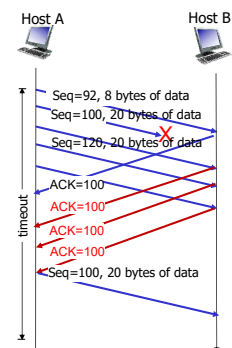


TCP 慢启动

何时结束慢启动？

- 引入阈值变量ssthresh，ssthresh在TCP初始化时被设置为一个很大的值(通常为65KB)
- 当cwnd≥ssthresh时
  - 发送方进入拥塞避免状态
- 超时事件发生：
  - ssthresh=当前cwnd值/2，cwnd=1 MSS；
  - 重新进入慢启动状态
- 收到3个冗余的ACK：
  - ssthresh=当前cwnd值/2，cwnd=ssthresh+3\*MSS
  - 发送方快速重发那些导致冗余ACK的丢失分组，进入快速恢复状态

## TCP快速重发:



数据通信与计算机网络A 135

## TCP 慢启动

- 收到3个冗余的ACK时，为什么不采用超时事件发生时遵循的“慢启动”的处理方式而进入“快速恢复”？
- 即使某个分组确实丢失了，但3个冗余ACK的到来表明某些报文段还是会被接收方收到，这表明网络还具备传送报文段的能力
- 因此发送方仍然可以具备一定的发送速率

数据通信与计算机网络A 136

## TCP 拥塞避免

### 拥塞避免

- 每经过一个传输轮次 (RTT)，cwnd只增加1MSS，cwnd呈线性速率缓慢增长

- 具体实现时常用的计算方法：收到一个对先前未确认报文段的确认， $cwnd = cwnd + MSS * (MSS / cwnd)$

### 如何结束拥塞避免？

- 超时事件发生：
  - $ssthresh = \text{当前} cwnd \text{ 值} / 2$ ， $cwnd = 1 \text{ MSS}$ ;
  - 重新进入慢启动状态
- 收到3个冗余的ACK：
  - $ssthresh = \text{当前} cwnd \text{ 值} / 2$ ， $cwnd = ssthresh + 3 * MSS$
  - 发送方快速重发那些导致冗余ACK的丢失分组，进入快速恢复状态

数据通信与计算机网络A 137

## TCP 快速恢复

### 快速恢复

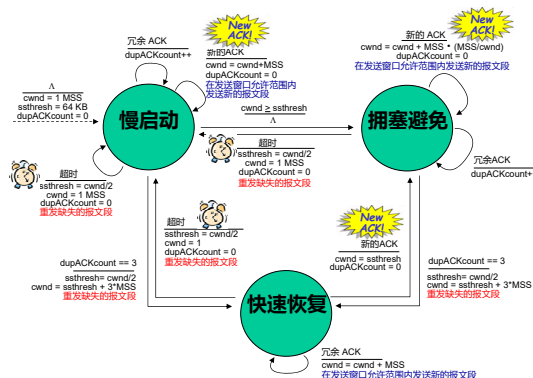
- 收到冗余ACK： $cwnd = cwnd + 1 \text{ MSS}$

### 如何结束快速恢复？

- 超时事件发生：
  - $ssthresh = \text{当前} cwnd \text{ 值} / 2$ ， $cwnd = 1 \text{ MSS}$ ;
  - 重新进入慢启动状态
- 收到一个对先前未确认报文段的确认：
  - 降低cwnd， $cwnd = ssthresh$
  - 进入拥塞避免状态

数据通信与计算机网络A 138

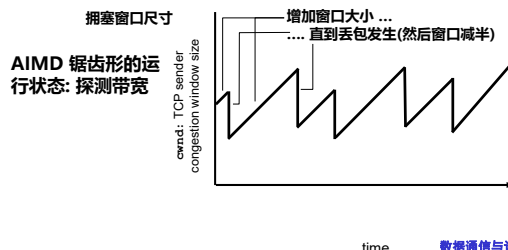
## TCP拥塞控制总结:



数据通信与计算机网络A 139

## TCP 拥塞控制算法: AIMD加性增, 乘性减

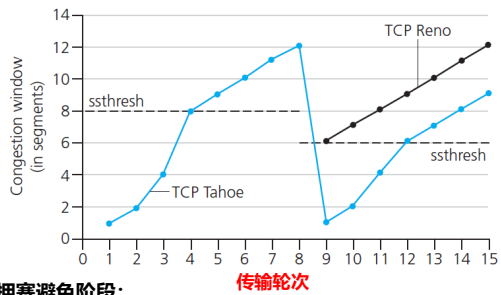
- 加性增: 在拥塞避免阶段，每一个RTT时间后，拥塞窗口 (cwnd) 值增加一个MSS(最大报文段长度)，直到检测到丢包
- 乘性减: 丢包后将拥塞窗口(cwnd)减少到原来的一半



time 数据通信与计算机网络A 140

## TCP拥塞算法(Tahoe版本和Reno版本)

拥塞窗口 (以MSS为单位)



在拥塞避免阶段:

- 对于超时和3个冗余Ack, TCP Tahoe将拥塞窗口cwnd置为1
- 而对于3个冗余ACK: TCP Reno则将新的cwnd设为当前cwnd/2+3MSS