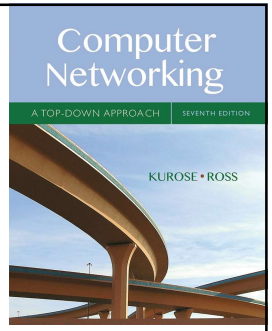
第 3 章
传 输 层



中国科学技术大学
自动化系 郑烇
改编自Jim kurose, Keith Ross

Computer Networking: A Top Down Approach
7th edition
Jim Kurose, Keith Ross
Addison-Wesley
April 2016

第三章：传输层

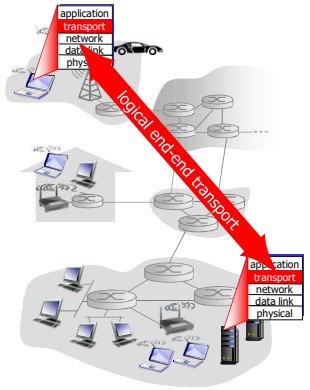
目标：

- 理解传输层的工作原理
  - 多路复用/解复用
  - 可靠数据传输
  - 流量控制
  - 拥塞控制
- 学习Internet的传输层协议
  - UDP：无连接传输
  - TCP：面向连接的可靠传输
  - TCP的拥塞控制

第三章：提纲

传输服务和协议

- 为运行在不同主机上的应用进程提供逻辑通信
- 传输协议运行在端系统
  - 发送方：将应用层的报文分成报文段，然后传递给网络层
  - 接收方：将报文段重组成报文，然后传递给应用层
- 有多个传输层协议可供应用选择
  - Internet：TCP和UDP



传输层 vs. 网络层

- 网络层服务：主机之间的逻辑通信
- 传输层服务：进程间的逻辑通信
  - 依赖于网络层的服务
    - 延时、带宽
  - 并对网络层的服务进行增强
    - 数据丢失、顺序混乱、加密

类比：东西2个家庭的通信

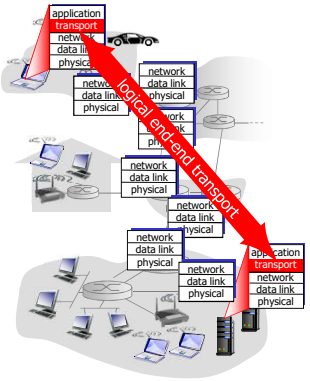
Ann家的12个小孩给另Bill家的12个小孩发信

- 主机 = 家庭
- 进程 = 小孩
- 应用层报文= 信封中的信件
- 传输协议= Ann 和 Bill
  - 为家庭小孩提供复用解复用服务
- 网络层协议 = 邮政服务
  - 家庭-家庭的邮包传输服务

有些服务是可以加强的：不可靠 -> 可靠；安全
但有些服务是不可以被加强的：带宽，延迟

Internet传输层协议

- 可靠的、保序的传输： TCP
  - 多路复用、解复用
  - 拥塞控制
  - 流量控制
  - 建立连接
- 不可靠、不保序的传输：UDP
  - 多路复用、解复用
  - 没有为尽力而为的IP服务添加更多的其它额外服务
- 都不提供的服务：
  - 延时保证
  - 带宽保证



第三章：提纲

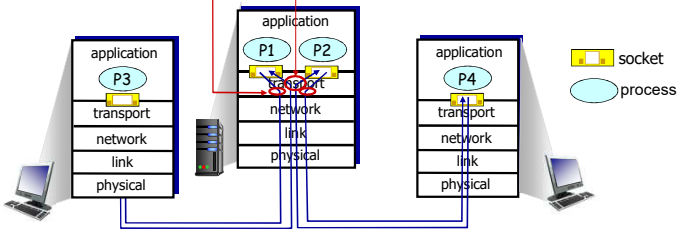
3.1 概述和传输层服务
3.2 多路复用与解复用
3.3 无连接传输：UDP
3.4 可靠数据传输的原理
3.5 面向连接的传输：TCP
- 段结构
- 可靠数据传输
- 流量控制
- 连接管理

3.6 拥塞控制原理
3.7 TCP 拥塞控制

多路复用/解复用

在发送方主机**多路复用**
从多个套接字接收来自多个进程的报文，根据套接字对应的IP地址和端口号等信息对报文段用头部加以封装 (该头部信息用于以后的解复用)

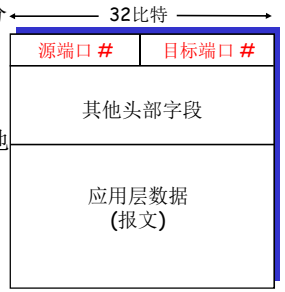
在接收方主机多路**解复用**
根据报文段的头部信息中的IP地址和端口号将接收到的报文段发给正确的套接字(和对应的应用进程)



多路解复用工作原理

UDP和TCP不同

- 解复用作用：TCP或者UDP实体采用哪些信息，将报文段的数据部分交给正确的socket，从而交给正确的进程
- 主机收到IP数据报
  - 每个数据报有源IP地址和目标地址
  - 每个数据报承载一个传输层报文段
  - 每个报文段有一个源端口号和目标端口号
    (特定应用有著名的端口号)
- 主机联合使用**IP地址**和**端口号**将报文段发送给合适的套接字



TCP/UDP报文段格式

无连接(UDP)多路解复用

- 创建套接字：

服务器端：

```
serverSocket=socket(PF_INET,
    SOCK_DGRAM,0);
bind(serverSocket, &sad, sizeof(sad));
```

serverSocket和Sad指定的端口号捆绑

客户端： UDP和TCP不同

```
ClientSocket=socket(PF_INET,
    SOCK_DGRAM,0);
```

没有Bind，ClientSocket和OS为之分配的某个端口号捆绑（客户端使用什么端口号无所谓，客户端主动找服务器）

- 在接收端，UDP套接字用二元组标识 (目标IP地址、目标端口号)
- 当主机收到UDP报文段：
  - 检查报文段的目标端口号
  - 用该端口号将报文段定位给套接字
- 如果两个不同源IP地址/源端口号的数据报，但是有相同的目标IP地址和端口号，则被定位到相同的套接字

PID	Socket	Des IP	Des Port
12345	77888	202.38.64.1	80
12346	77999	202.38.64.1	11010
.....			

无连接的多路解复用

- 回顾：创建拥有本地端口号的套接字

```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

- 回顾：当创建UDP段采用端口号，可以指定：
  - 目标IP地址
  - 目标端口号

- 当主机接收到UDP段时：
  - 检查UDP段中的目标端口号
  - 将UDP段交给具备那个端口号的套接字

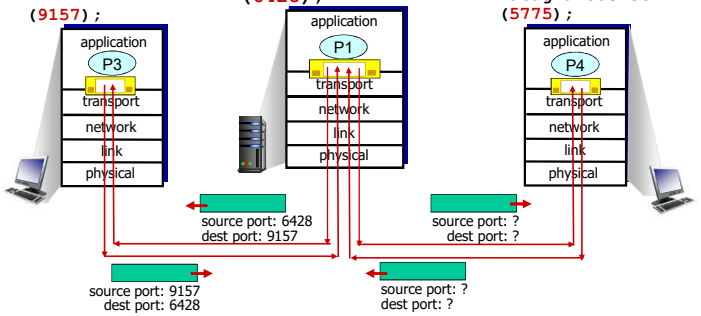
具备相同目标IP地址和目标端口号，即使是源IP地址 或/且源端口号的IP数据报，将会被传到相同的目标UDP套接字上

无连接多路复用：例子

```
DatagramSocket
mySocket2 = new
DatagramSocket
(9157);
```

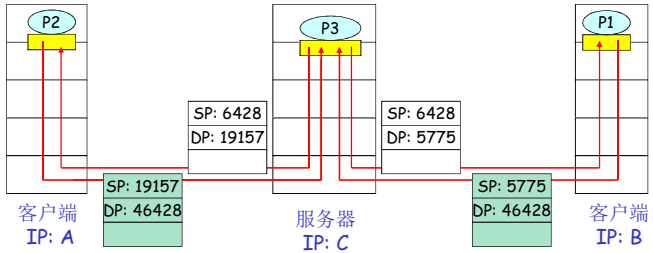
```
DatagramSocket
serverSocket = new
DatagramSocket
(6428);
```

```
DatagramSocket
mySocket1 = new
DatagramSocket
(5775);
```



无连接多路解复用(续)

serverSocket = serverSocket=socket(PF_INET,SOCK_DGRAM,0);
bind(serverSocket, sad, sizeof(sad));



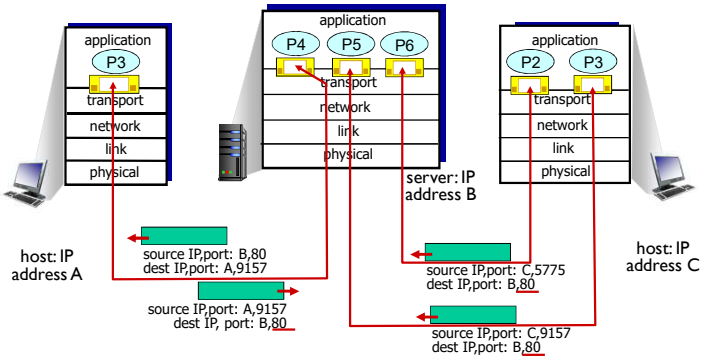
在每个接收到的报文段中都提供了"返回地址"

Socket	PID	Des IP	Des Port
77888	P3	C	6428
.....			

面向连接(TCP)的多路复用

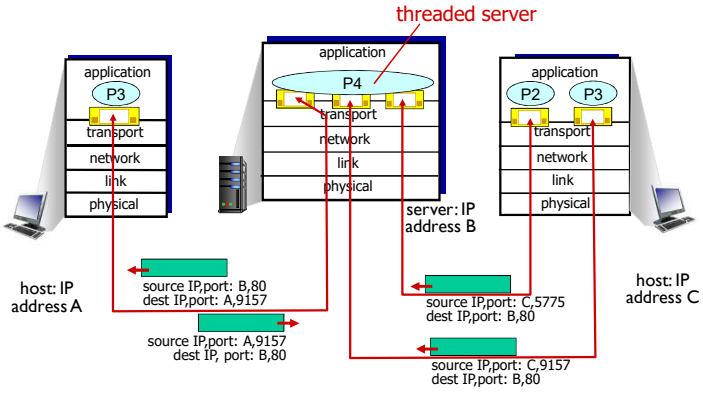
- TCP套接字:四元组本地标识:
  - 源IP地址
  - 源端口号
  - 目的IP地址
  - 目的端口号
- 解复用：接收主机用这四个值来将数据报定位到合适的套接字
- 服务器能够在一个TCP端口上同时支持多个TCP套接字：
  - 每个套接字由其四元组标识（有不同的源IP和源PORT）
- Web服务器对每个连接客户端有不同的套接字
  - 非持久对每个请求有不同的套接字

面向连接的解复用：例子



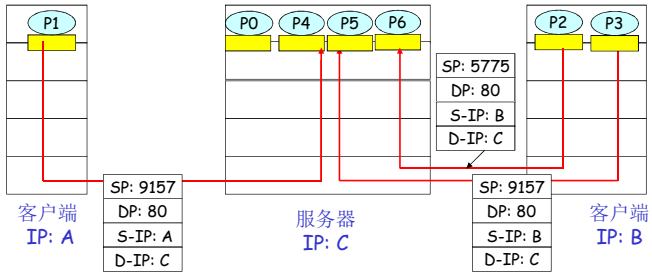
three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

面向连接的解复用：例子



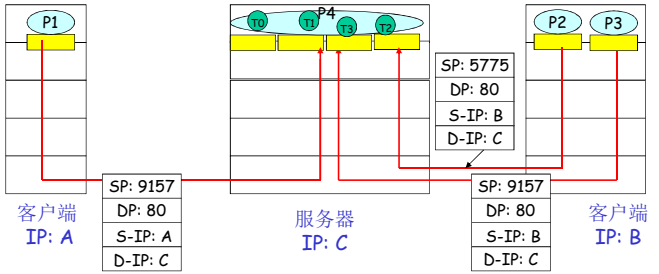
面向

Socket	PID	SIP	SPort	DIP	DPort
90000	P0			C	80
90004	P4	A	9157	C	80
90005	P5	B	9157	C	80
90006	P6	B	5775	C	80



Socke	PID	SIP	SPort	DIP	DPort
80011	P1	C	80	A	9157

面向连接的多路复用：多线程Web Server



- 一个进程下面可能有多个线程：由多个线程分别为客户提供服务
- 在这个场景下，还是根据4元组决定将报文段内容同一个进程下的不同线程
- 解复用到不同线程

第三章：提纲

3.1 概述和传输层服务
3.2 多路复用与解复用
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
- 段结构
- 可靠数据传输
- 流量控制
- 连接管理

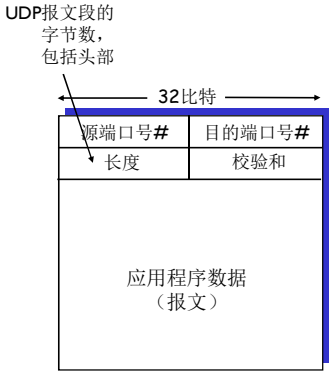
3.6 拥塞控制原理
3.7 TCP 拥塞控制

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones"Internet传输协议
- "尽力而为"的服务，报文段可能
  - 丢失
  - 送到应用进程的报文段乱序
- 无连接：
  - UDP发送端和接收端之间没有握手
  - 每个UDP报文段都被独立地处理

- UDP 被用于：
  - 流媒体（丢失不敏感，速率敏感、应用可控制传输速率）
  - DNS
  - SNMP
- 在UDP上可行可靠传输：
  - 在应用层增加可靠性
  - 应用特定的差错恢复

UDP：用户数据报协议



UDP报文段格式

为什么要有UDP?

- 不建立连接 （会增加延时）
- 简单：在发送端和接收端没有连接状态
- 报文段的头部很小(开销小)
- 无拥塞控制和流量控制：UDP可以尽可能快的发送报文段
  - 应用->传输的速率= 主机->网络的速率

UDP校验和

目标： 检测在被传输报文段中的差错 (如比特反转)

发送方：

- 将报文段的内容视为16比特的整数
- 校验和：报文段的加法和 （1的补运算）
- 发送方将校验和放在UDP的校验和字段

接收方：

- 计算接收到的报文段的校验和
- 检查计算出的校验和与校验和字段的内容是否相等：
  - 不相等---检测到差错
  - 相等---没有检测到差错，但也许还是有差错
    - 残存错误

Internet校验和的例子

- 注意：当数字相加时，在最高位的进位要回卷，再加到结果上

例子：两个16比特的整数相加

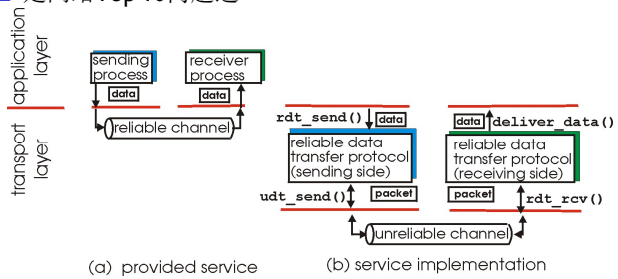
		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
回卷	(1)	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
和		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
校验和		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

- 目标端：校验范围+校验和=1111111111111111 通过校验
  - 否则没有通过校验
- 注：求和时，必须将进位回卷到结果上

第三章：提纲

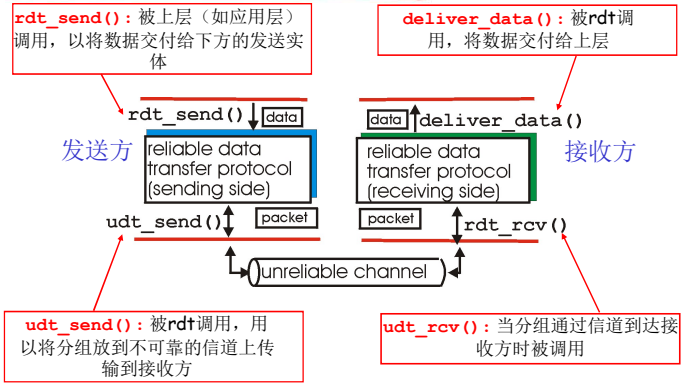
可靠数据传输（rdt）的原理

- rdt在应用层、传输层和数据链路层都很重要
- 是网络Top 10问题之一



- 信道的不可靠特点决定了可靠数据传输协议（ rdt ）的复杂性

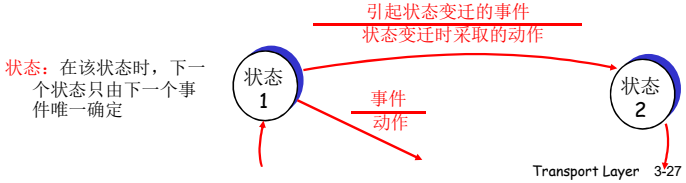
可靠数据传输：问题描述



可靠数据传输：问题描述（续）

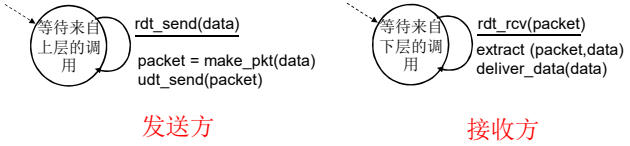
我们将：

- 渐增式地开发可靠数据传输协议（ rdt ）的发送方和接收方
- 只考虑单向数据传输
  - 但控制信息是双向流动的！
- 双向的数据传输问题实际上是2个单向数据传输问题的综合
- 使用有限状态机 (FSM) 来描述发送方和接收方



Rdt1.0：在可靠信道上的可靠数据传输

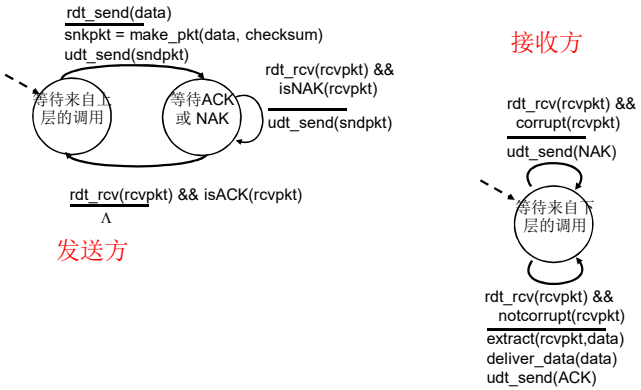
- 下层的信道是完全可靠的
  - 没有比特出错
  - 没有分组丢失
- 发送方和接收方的FSM
  - 发送方将数据发送到下层信道
  - 接收方从下层信道接收数据



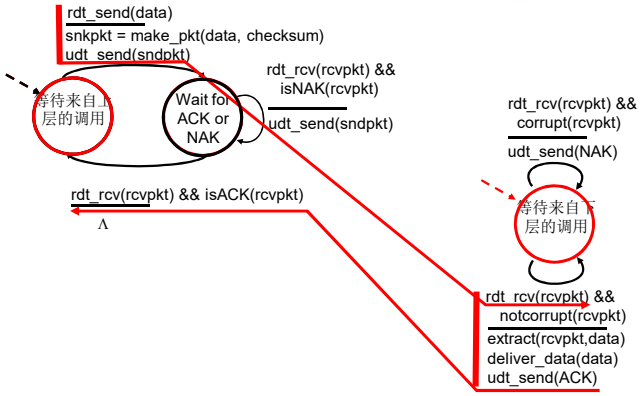
Rdt2.0：具有比特差错的信道

- 下层信道可能会出错：将分组中的比特翻转
  - 用校验和来检测比特差错
- 问题：怎样从差错中恢复：
  - 确认(ACK)：接收方显式地告诉发送方分组已被正确接收
  - 否定确认(NAK): 接收方显式地告诉发送方分组发生了差错
    - 发送方收到NAK后，发送方重传分组
- rdt2.0中的新机制：采用差错控制编码进行差错检测
  - 发送方差错控制编码、缓存
  - 接收方使用编码检错
  - 接收方的反馈：控制报文（ACK，NAK）：接收方→发送方
  - 发送方收到反馈相应的动作

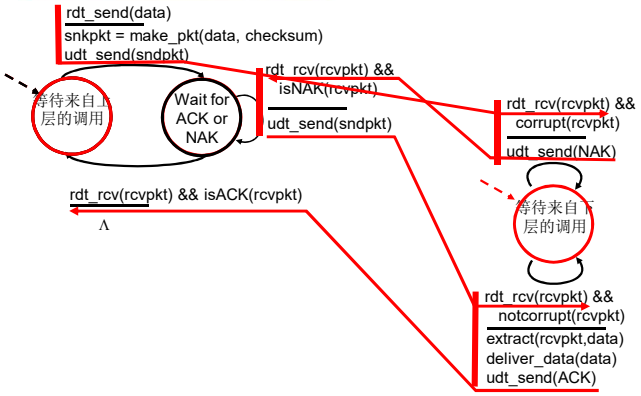
rdt2.0：FSM描述



rdt2.0：没有差错时的操作



rdt2.0：有差错时



rdt2.0的致命缺陷！ -> rdt2.1

如果ACK/NAK出错？

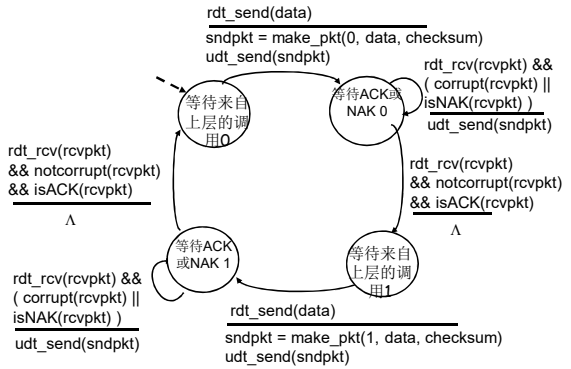
- 发送方不知道接收方发生了什么事情！
- 发送方如何做？
  - 重传？可能重复
  - 不重传？可能死锁(或出错)
- 需要引入新的机制
  - 序号

处理重复：

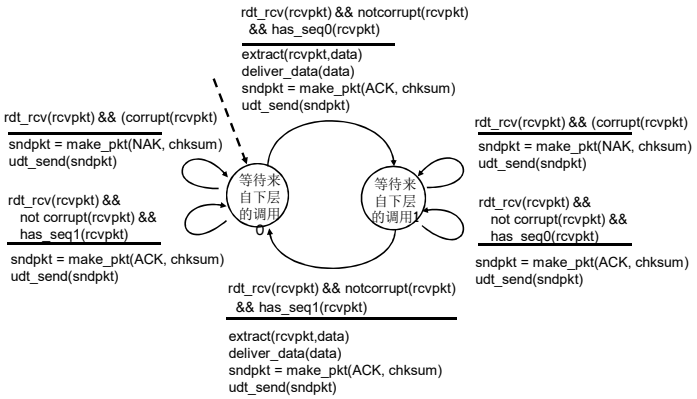
- 发送方在每个分组中加入序号
- 如果ACK/NAK出错，发送方重传当前分组
- 接收方丢弃（不发给上层）重复分组

停等协议
发送方发送一个分组，
然后等待接收方的应答

rdt2.1：发送方处理出错的ACK/NAK



rdt2.1：接收方处理出错的ACK/NAK



rdt2.1：讨论

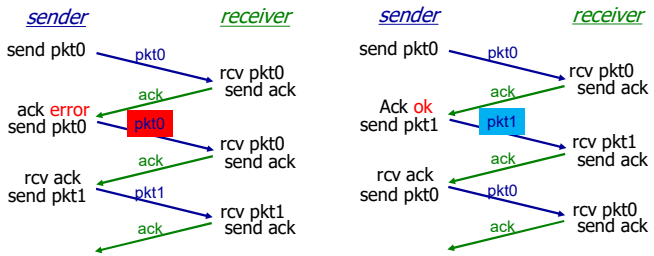
发送方：

- 在分组中加入序列号
- 两个序列号（0，1）就足够了
  - 一次只发送一个未经确认的分组
- 必须检测ACK/NAK是否出错（需要EDC ）
- 状态数变成了两倍
  - 必须记住当前分组的序列号为0还是1

接收方：

- 必须检测接收到的分组是否是重复的
  - 状态会指示希望接收到的分组的序号为0还是1
- 注意：接收方并不知道发送方是否正确收到了其ACK/NAK
  - 没有安排确认的确认
  - 具体解释见下页

rdt2.1的运行



(a) ack error　　(b) ack right

接收方不知道它最后发送的ACK/NAK是否被正确地收到

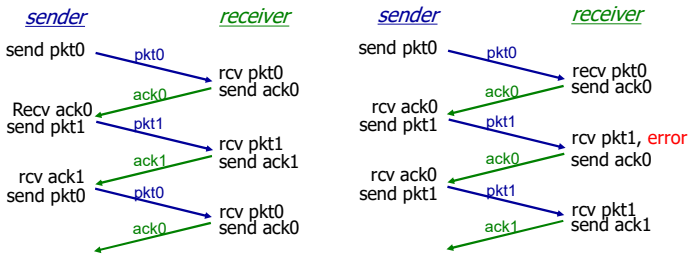
- 发送方不对收到的ack/nak给确认，没有所谓的确认的确认；
- 接收方发送ack，如果后面接收方收到的是：
  - 老分组p0？则ack 错误
  - 下一个分组？P1，ack正确

rdt2.2：无NAK的协议

- 功能同rdt2.1，但只使用ACK(ack 要编号）
- 接收方对最后正确接收的分组发ACK，以替代NAK
  - 接收方必须显式地包含被正确接收分组的序号
- 当收到重复的ACK（如：再次收到ack0）时，发送方与收到NAK采取相同的动作：重传当前分组
- 为后面的一次发送多个数据单位做一个准备
  - 一次能够发送多个
  - 每一个的应答都有：ACK，NACK；麻烦
  - 使用对前一个数据单位的ACK，代替本数据单位的nak
  - 确认信息减少一半，协议处理简单

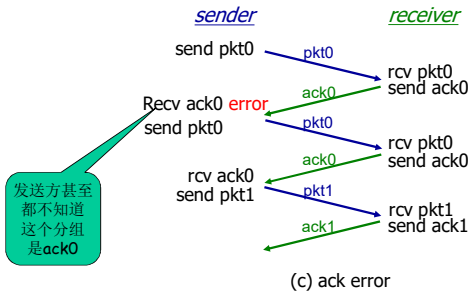
NAK free

rdt2.2的运行



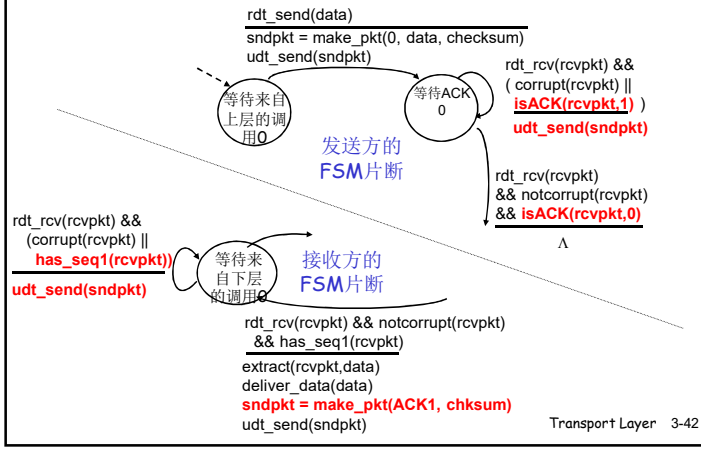
(a) No error（packet or ack error）　　(b) packet error

rdt2.2的运行



(c) ack error

rdt2.2：发送方和接收方片断



rdt3.0：具有比特差错和分组丢失的信道

新的假设：下层信道可能会丢失分组（数据或ACK）

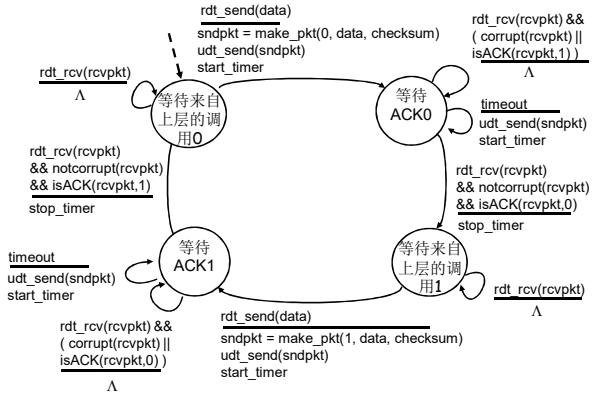
- 会死锁
- 机制还不够处理这种状况：
  - 检验和
  - 序列号
  - ACK
  - 重传

方法：发送方等待ACK一段合理的时间

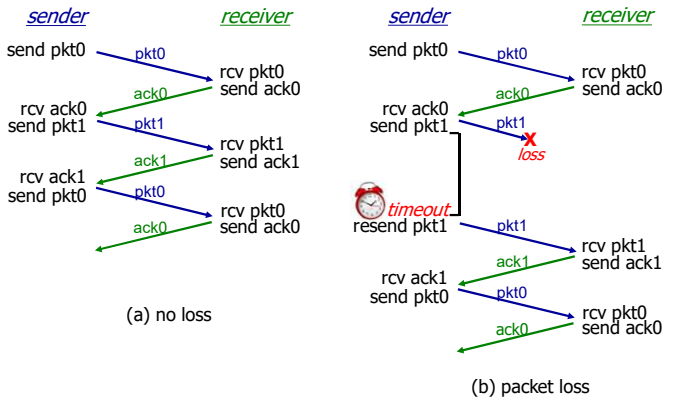
- 发送端超时重传：如果到时没有收到ACK->重传
- 问题：如果分组（或ACK ）只是被延迟了：
  - 重传将会导致数据重复，但利用序列号已经可以处理这个问题
  - 接收方必须指明被正确接收的序列号
- 需要一个倒计数定时器

链路层的timeout时间确定的
传输层timeout时间是适应式的

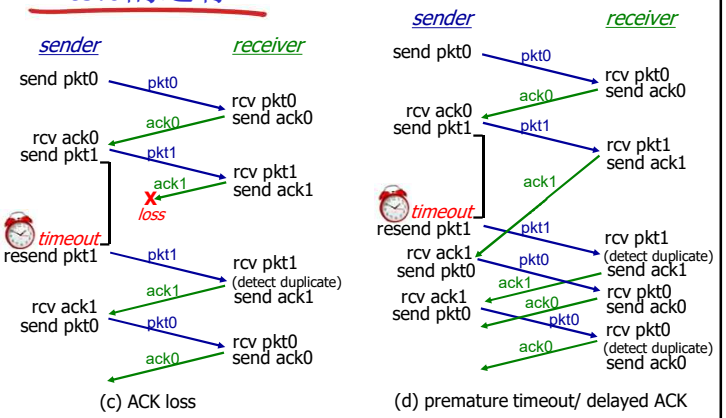
rdt3.0 发送方



rdt3.0的运行



rdt3.0的运行



- 过早超时（延迟的ACK）也能够正常工作；但是效率较低，一半的分组和确认是重复的；
- 设置一个合理的超时时间也是比较重要的；

rdt3.0的性能

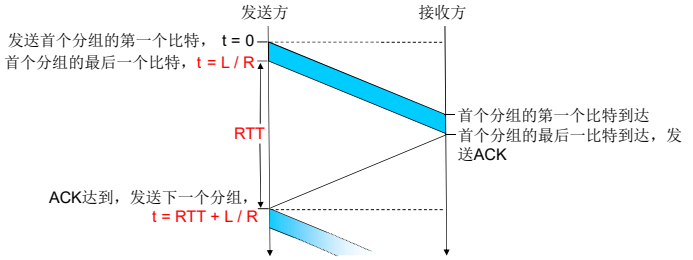
- rdt3.0可以工作，但链路容量比较大的情况下，性能很差
  - 链路容量比较大，一次发一个PDU 的不能够充分利用链路的传输能力
- 例：1 Gbps的链路，15 ms端-端传播延时，分组大小为1kB：

$$T_{transmit} = \frac{L\ (分组长度，比特)}{R\ (传输速率，bps)} = \frac{8kb/pkt}{10^9\ b/sec} = 8\mu s$$

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

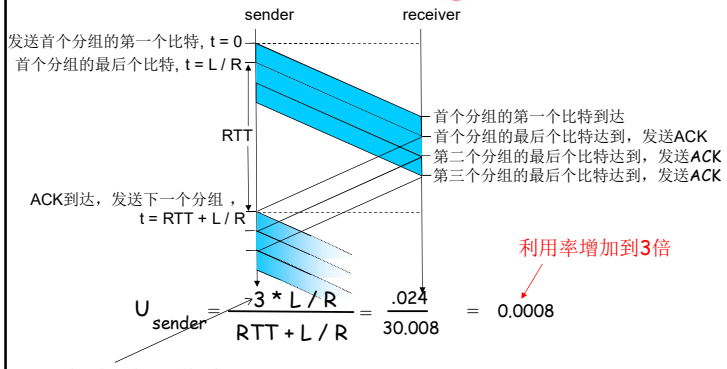
  - U_{sender}：利用率 - 忙于发送的时间比例
  - 每30ms发送1KB的分组→ 270kbps=33.75kB/s 的吞吐量（在1 Gbps 链路上）
  - 瓶颈在于：网络协议限制了物理资源的利用！

rdt3.0：停-等操作



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

流水线：提高链路利用率

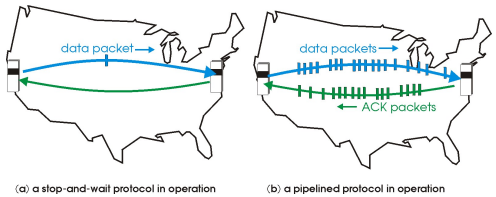


- 增加n,能提高链路利用率
- 但当达到某个n,其u=100%时,无法再通过增加n，提高利用率
- 瓶颈转移了->链路带宽

流水线协议

流水线：允许发送方在未得到对方确认的情况下一次发送多个分组

- 必须增加序号的范围:用多个bit表示分组的序号
- 在发送方/接收方要有缓冲区
  - 发送方缓冲：未得到确认，可能需要重传；
  - 接收方缓存：上层用户取用数据的速率≠接收到的数据速率；接收到的数据可能乱序，排序交付（可靠）



(a) a stop-and-wait protocol in operation　(b) a pipelined protocol in operation

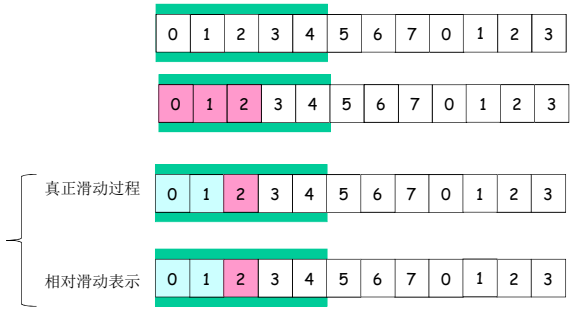
- 两种通用的流水线协议：回退N步(GBN)和选择重传(SR)

通用：滑动窗口(slide window)协议

- 发送缓冲区
  - 形式：内存中的一个区域，落入缓冲区的分组可以发送
  - 功能：用于存放已发送，但是没有得到确认的分组
  - 必要性：需要重发时可用
- 发送缓冲区的大小：一次最多可以发送多少个未经确认的分组
  - 停止等待协议=1
  - 流水线协议>1，合理的值，不能很大，链路利用率不能够超100%
- 发送缓冲区中的分组
  - 未发送的：落入发送缓冲区的分组，可以连续发送出去；
  - 已经发送出去的、等待对方确认的分组：发送缓冲区的分组只有得到确认才能删除

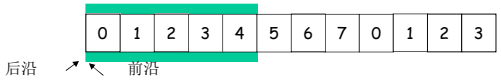
发送窗口滑动过程-相对表示方法

- 采用相对移动方式表示，分组不动
- 可缓冲范围移动，代表一段可以发送的权力



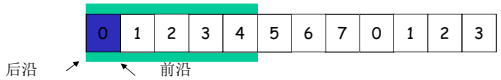
滑动窗口(slide window)协议

- 发送窗口：发送缓冲区内容的一个范围
  - 那些已发送但是未经确认分组的序号构成的空间
- 发送窗口的最大值<=发送缓冲区的值
- 一开始：没有发送任何一个分组
  - 后沿=前沿
  - 之间为发送窗口的尺寸=0
- 每发送一个分组，前沿前移一个单位



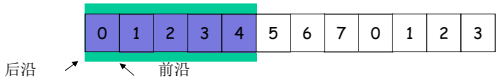
发送窗口的移动->前沿移动

- 发送窗口前沿移动的极限：不能够超过发送缓冲区



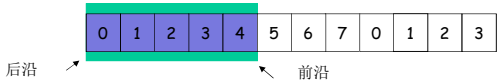
发送窗口的移动->前沿移动

- 发送窗口前沿移动的极限：不能够超过发送缓冲区



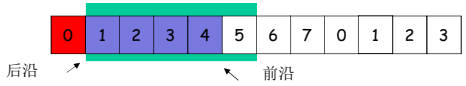
发送窗口的移动->后沿移动

- 发送窗口后沿移动
  - 条件：收到老分组的确认
  - 结果：发送缓冲区罩住新的分组，来了分组可以发送
  - 移动的极限：不能够超过前沿



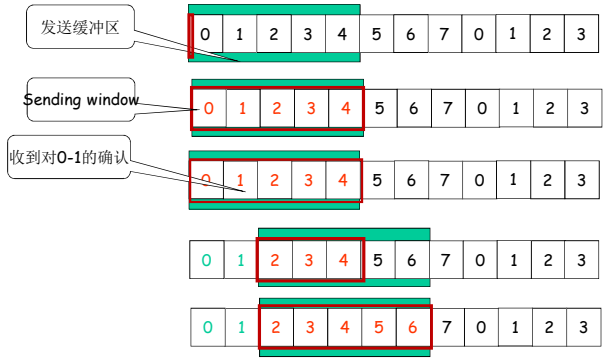
发送窗口的移动->后沿移动

- 发送窗口后沿移动
  - 条件：收到老分组(后沿)的确认
  - 结果：发送缓冲区罩住新的分组，来了分组可以发送
  - 移动的极限：不能够超过前沿



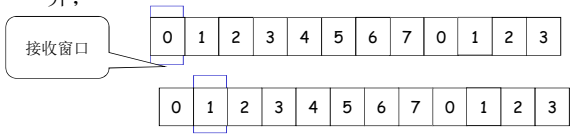
3.5 滑动窗口(slide window)协议

- 滑动窗口技术
  - 发送窗口 (sending window)



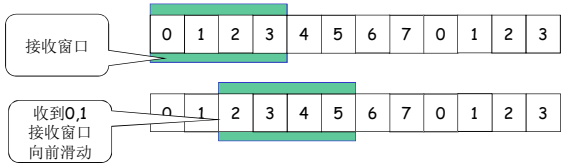
滑动窗口(slide window)协议-接收窗口

- 接收窗口 (receiving window)=接收缓冲区
  - 接收窗口用于控制哪些分组可以接收；
    - 只有收到的分组序号落入接收窗口内才允许接收
    - 若序号在接收窗口之外，则丢弃；
  - 接收窗口尺寸Wr=1，则只能顺序接收；
  - 接收窗口尺寸Wr>1 ，则可以乱序接收
    - 但提交给上层的分组，要按序
  - 例子：Wr = 1，在0的位置；只有0号分组可以接收；向前滑动一个，罩在1的位置，如果来了第2号分组，则丢弃；

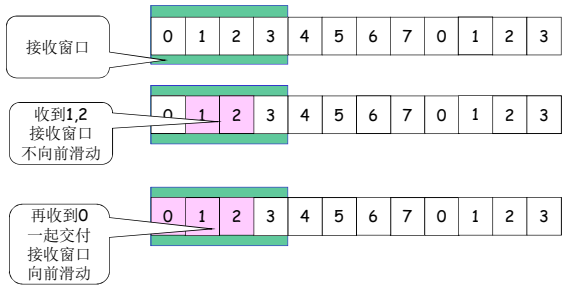


滑动窗口(slide window)协议-接收窗口

- 接收窗口的滑动和发送确认
  - 滑动：
    - 低序号的分组到来，接收窗口移动；
    - 高序号分组乱序到，缓存但不交付（因为要实现rdt，不允许失序），不滑动
  - 发送确认：
    - 接收窗口尺寸=1； 发送连续收到的最大的分组确认（累计确认）
    - 接收窗口尺寸>1； 收到分组，发送那个分组的确认（非累计确认）



滑动窗口(slide window)协议-接收窗口



正常情况下的2个窗口互动

- 发送窗口
  - 有新的分组落入发送缓冲区范围，发送->前沿滑动
  - 来了老的低序号分组的确认->后沿向前滑动->新的分组可以落入发送缓冲区的范围
- 接收窗口
  - 收到分组，落入到接收窗口范围内，接收
  - 是低序号，发送确认给对方
- 发送端上面来了分组->发送窗口滑动->接收窗口滑动->发确认

异常情况下GBN的2窗口互动

- 发送窗口
  - 新分组落入发送缓冲区范围，发送->前沿滑动
  - 超时重发机制让发送端将发送窗口中的所有分组发送出去
  - 来了老分组的重复确认->后沿不向前滑动->新的分组无法落入发送缓冲区的范围（此时如果发送缓冲区有新的分组可以发送）
- 接收窗口
  - 收到乱序分组，没有落入到接收窗口范围内，抛弃
  - （重复）发送老分组的确认，累计确认；

异常情况下SR的2窗口互动

- 发送窗口
  - 新分组落入发送缓冲区范围，发送->前沿滑动
  - 超时重发机制让发送端将超时的分组重新发送出去
  - 来了乱序分组的确认->后沿不向前滑动->新的分组无法落入发送缓冲区的范围（此时如果发送缓冲区有新的分组可以发送）
- 接收窗口
  - 收到乱序分组，落入到接收窗口范围内，接收
  - 发送该分组的确认，单独确认；

GBN协议和SR协议的异同

- 相同之处
  - 发送窗口>1
  - 一次能够可发送多个未经确认的分组
- 不同之处
  - GBN :接收窗口尺寸=1
    - 接收端：只能顺序接收
    - 发送端：从表现来看，一旦一个分组没有发成功，如：0,1,2,3,4；假如1未成功，234都发送出去了，要返回1再发送；*GB1*
  - SR: 接收窗口尺寸>1
    - 接收端：可以乱序接收
    - 发送端：发送0,1,2,3,4，一旦1未成功，2,3,4,已发送，无需重发，选择性发送1

流水线协议：总结

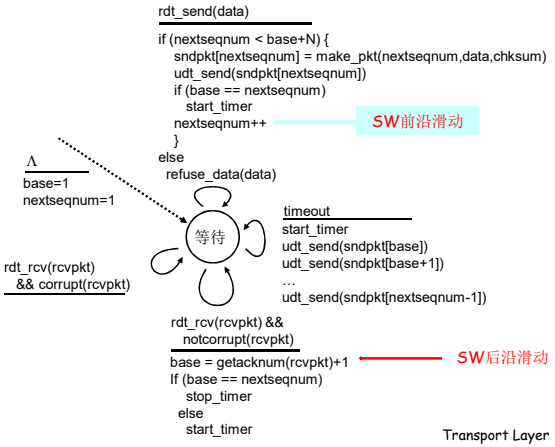
Go-back-N:

- 发送端最多在流水线中有N个未确认的分组
- 接收端只是发送累计型确认*cumulative ack*
  - 接收端如果发现gap，不确认新到来的分组
- 发送端拥有对最老的未确认分组的定时器
  - 只需设置一个定时器
  - 当定时器到时时，重传所有未确认分组

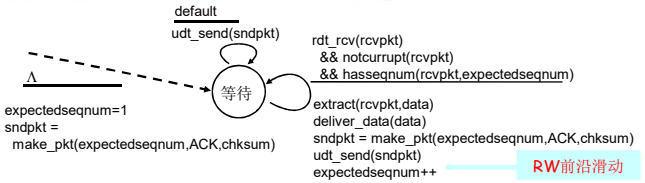
Selective Repeat:

- 发送端最多在流水线中有N个未确认的分组
- 接收方对每个到来的分组单独确认*individual ack*（非累计确认）
- 发送方为每个未确认的分组保持一个定时器
  - 当超时定时器到时，只是重发到时的未确认分组

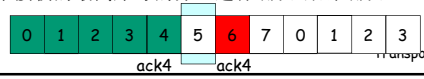
GBN：发送方扩展的FSM



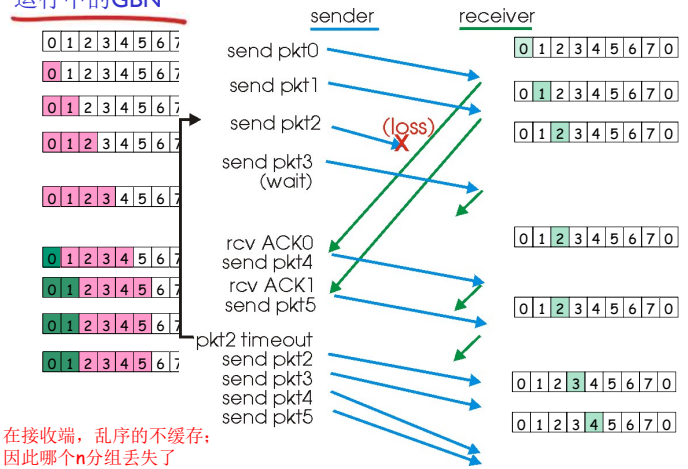
GBN：接收方扩展的FSM



- 只发送ACK：对顺序接收的最高序号的分组
  - 可能会产生重复的ACK
  - 只需记住**expectedseqnum**；接收窗口=1
    - **只一个变量就可表示接收窗口**
- 对乱序的分组：
  - 丢弃（不缓存）→ 在接收方不被缓存！
  - 对顺序接收的最高序号的分组进行确认-累计确认



运行中的GBN



在接收端，乱序的不缓存：
因此哪个n分组丢失了
GB到那个分组n：
即使n以后的分组传送都是正确的

选择重传SR

- 接收方对每个正确接收的分组，分别发送ACKn（非累积确认）
  - 接收窗口>1
    - 可以缓存乱序的分组
  - 最终将分组按顺序交付给上层
- 发送方只对那些没有收到ACK的分组进行重发-选择性重发
  - 发送方为每个未确认的分组设定一个定时器
- 发送窗口的最大值（发送缓冲区）限制发送未确认分组的个数

选择重传

发送方

从上层接收数据：

- 如果下一个可用于该分组的序号可在发送窗口中，则发送

timeout(n):

- 重新发送分组n，重新设定定时器

ACK(n) in [sendbase,sendbase+N]:

- 将分组n标记为已接收
- 如n为最小未确认的分组序号，将base移到下一个未确认序号

接收方

分组n [rcvbase, rcvbase+N-1]

- 发送ACK(n)
- 乱序：缓存
- 有序：该分组及以前缓存的序号连续的分组交付给上层，然后将窗口移到下一个仍未被接收的分组

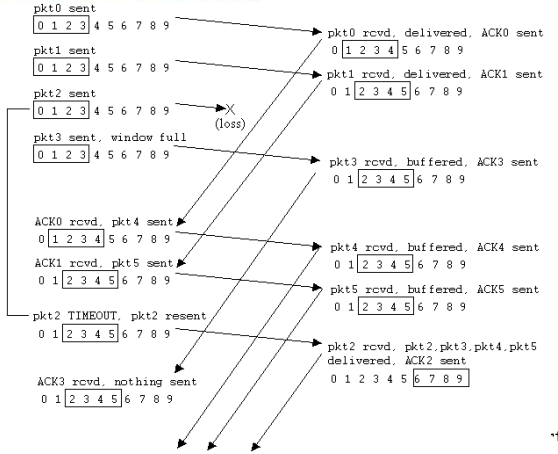
分组n [rcvbase-N, rcvbase-1]

- ACK(n)

其它：

- 忽略该分组

选择重传SR的运行



对比GBN和SR

	GBN	SR
优 点	简单，所需资源少（接收方一个缓存单元）	出错时，重传一个代价小
缺 点	一旦出错，回退N步代价大	复杂，所需要资源多（接收方多个缓存单元）

- 适用范围
  - 出错率低：比较适合GBN，出错非常罕见，没有必要用复杂的SR，为罕见的事件做日常的准备和复杂处理
  - 链路容量大（延迟大、带宽大）：比较适合SR而不是GBN，一点出错代价太大

窗口的最大尺寸

- GBN: 2^n-1
- SR:2^{n-1}

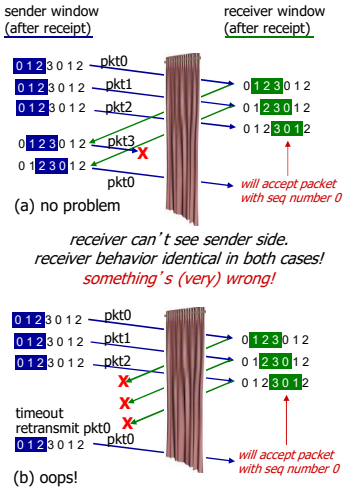
例如：n=2; 序列号：0, 1, 2, 3

- GBN =3
- SR=2

SR的例子：

- 接收方看不到二者的区别！
- 将重复数据误认为新数据 (a)

Q: 序号大小与窗口大小之间的关系？



第3章：提纲

3.1 概述和传输层服务
3.2 多路复用与解复用
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
- 段结构
- 可靠数据传输
- 流量控制
- 连接管理

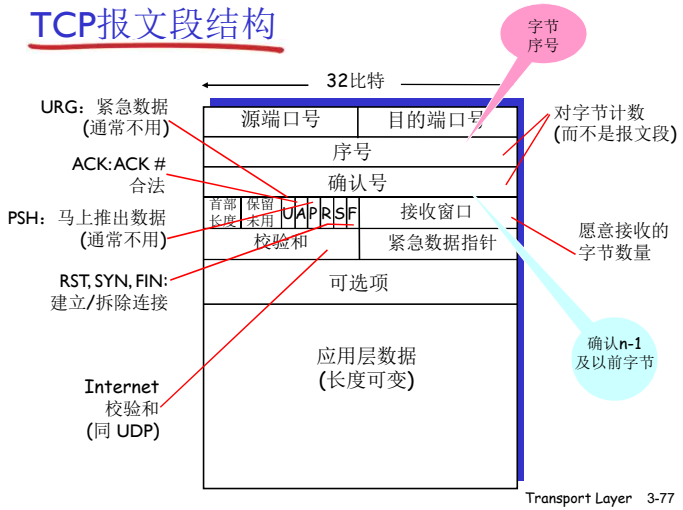
3.6 拥塞控制原理
3.7 TCP 拥塞控制

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

- 点对点：
  - 一个发送方，一个接收方
- 可靠的、按顺序的字节流：
  - 没有报文边界
- 管道化（流水线）：
  - TCP拥塞控制和流量控制设置窗口大小
- 发送和接收缓存

- 全双工数据：
  - 在同一连接中数据流双向流动
  - MSS：最大报文段大小
- 面向连接：
  - 在数据交换之前，通过握手（交换控制报文） 初始化发送方、接收方的状态变量
- 有流量控制：
  - 发送方不会淹没接收方

TCP报文段结构



TCP 序号，确认号

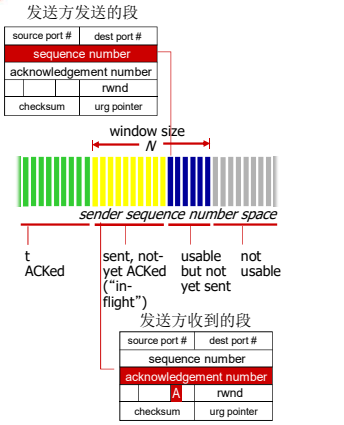
序号：

- 报文段首字节的在字节流的编号

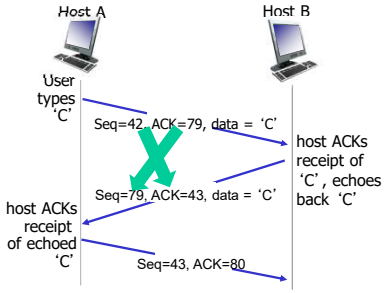
确认号：

- 期望从另一方收到的下一个字节的序号
- 累积确认

Q:接收方如何处理乱序的报文段-没有规定



TCP序号和确认号



简单telne场景

TCP往返延时（RTT）和超时

Q：怎样设置TCP超时？

- 比RTT要长
  - 但RTT是变化的
- 太短：太早超时
  - 不必要的重传
- 太长：对报文段丢失反应太慢，消极

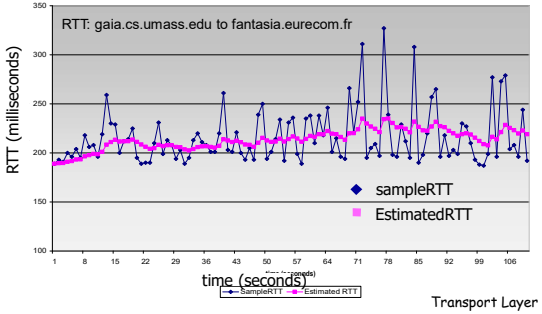
Q：怎样估计RTT？

- **SampleRTT**：测量从报文段发出到收到确认的时间
  - 如果有重传，忽略此次测量
- **SampleRTT**会变化，因此估计的RTT应该比较平滑
  - 对几个最近的测量值求平均，而不是仅用当前的**SampleRTT**

TCP往返延时（RTT）和超时

EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT

- 指数加权移动平均
- 过去样本的影响呈指数衰减
- 推荐值：α = 0.125



TCP往返延时（RTT）和超时

设置超时

- **EstimtedRTT** + 安全边界时间
  - **EstimatedRTT变化**大 (方差大)→ 较大的安全边界时间
- SampleRTT会偏离EstimatedRTT多远：

DevRTT = (1-β)*DevRTT +
β*|SampleRTT-EstimatedRTT|

(推荐值：β = 0.25)

超时时间间隔设置为：

TimeoutInterval = EstimatedRTT + 4*DevRTT

estimated RTT　　“safety margin”

第3章：提纲

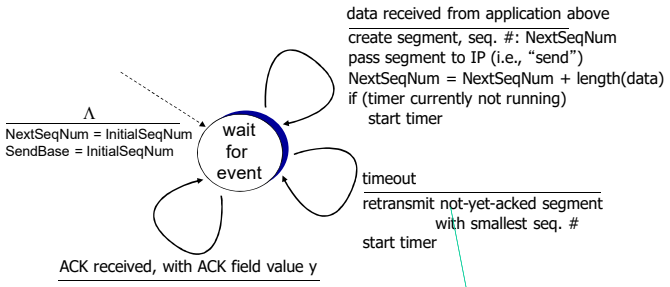
3.1 概述和传输层服务
3.2 多路复用与解复用
3.3 无连接传输：UDP
3.4 可靠数据传输的原理
3.5 面向连接的传输：TCP
- ■段结构
- ■可靠数据传输
- ■流量控制
- ■连接管理

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不可靠服务的基础上建立了rdt
  - 管道化的报文段
    - GBN or SR
  - 累积确认（像GBN）
  - 单个重传定时器（像GBN）
  - 是否可以接受乱序的，没有规范
- 通过以下事件触发重传
  - 超时（只重发那个最早的未确认段：SR）
  - 重复的确认
    - 例子：收到了ACK50,之后又收到3个ACK50
- 首先考虑简化的TCP发送方：
  - 忽略重复的确认
  - 忽略流量控制和拥塞控制

TCP 发送方(简化版)



只传送具有最小序号的未确段像SR，不是将所有未确认的都传一遍

TCP发送方事件：

从应用层接收数据：

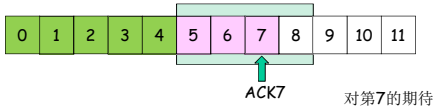
- 用nextseq创建报文段
- 序号nextseq为报文段首字节的字节流编号
- 如果还没有运行，启动定时器
  - 定时器与最早未确认的报文段关联
  - 过期间隔：TimeOutInterval

超时：

- 重传后沿最老的报文段
- 重新启动定时器

收到确认：

- 如果是对尚未确认的报文段确认
  - 更新已被确认的报文序号
  - 如果当前还有未被确认的报文段，重新启动定时器




```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
          smallest sequence number
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
              start timer
          }

} /* end of loop forever */
```

简化的TCP 发送方

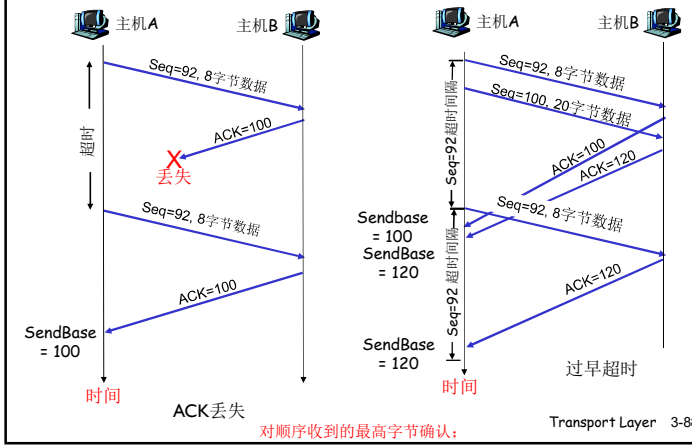
注释：

- SendBase-1: 最后一个累积确认的字节

例：

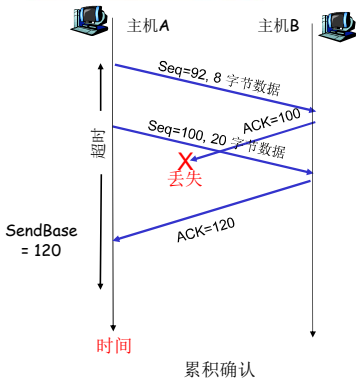
- SendBase-1 = 71; y= 73，因此接收方期望73+；y > SendBase，因此新的数据被确认

TCP：重传



对顺序收到的最高字节确认：

TCP：重传

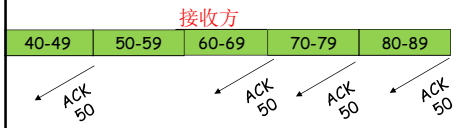


产生TCP ACK的建议 [RFC 1122, RFC 2581]

接收方的事件	TCP接收方动作
所期望序号的报文段按序到达。所有在期望序号之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500ms。如果下一个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到达。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。
比期望序号大的报文段乱序到达。检测出数据流中的间隔	立即发送重复的ACK，指明下一个期待字节的序号
能部分或完全填充接收数据间隔的报文段到达。	若该报文段起始于间隔（gap）的低端，则立即发送ACK。

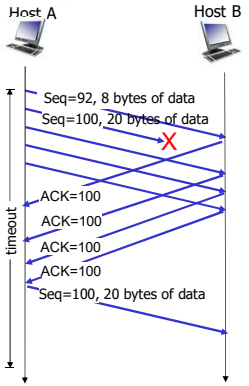
快速重传

- 超时周期往往太长：
  - 在重传丢失报文段之前的延时太长
- 通过重复的ACK来检测报文段丢失
  - 发送方通常连续发送大量报文段
  - 如果报文段丢失，通常会引起多个重复的ACK



- 如果发送方收到同一数据的3个冗余ACK，重传最小序号的段：
  - 快速重传：在定时器过时之前重发报文段
  - 它假设跟在被确认的数据后面的数据丢失了
    - 第一个ACK是正常的；
    - 收到第二个该段的ACK，表示接收方收到一个该段后的乱序段；
    - 收到第3，4个该段的ack，表示接收方收到该段之后的2个，3个乱序段，可能性非常大段丢失了

TCP快速重传



fast retransmit after sender receipt of triple duplicate ACK

快速重传算法：

```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase = y
                if (there are currently not-yet-acknowledged segments)
                    start timer
            }
            else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                }
```

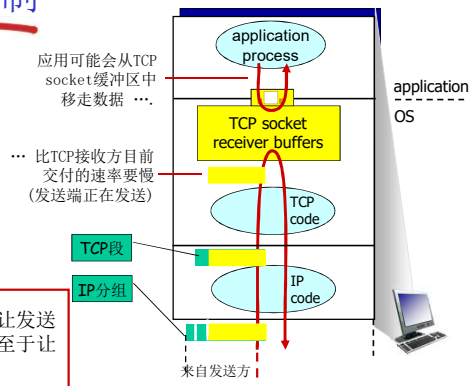
已确认报文段的一个重复确认

快速重传

第3章：提纲

TCP 流量控制

缓冲区TCP往里面写，app从当中读取



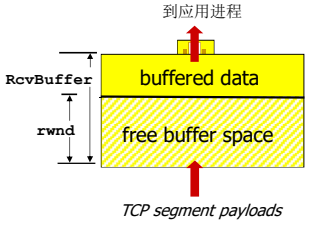
接收方协议栈

流量控制

接收方控制发送方，不让发送方发送的太多、太快以至于让接收方的缓冲区溢出

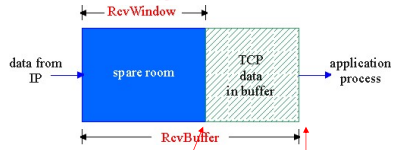
TCP流量控制

- 接收方在其向发送方的TCP段头部的rwnd字段“通告”其空闲buffer大小
  - RcvBuffer大小通过socket选项设置（典型默认大小为4096 字节）
  - 很多操作系统自动调整RcvBuffer
- 发送方限制未确认（“in-flight”）字节的个数≤接收方发送过来的 rwnd 值
- 保证接收方不会被淹没



接收方buffer大小

TCP流量控制



（假设TCP接收方丢弃乱序的报文段）

- 缓存中的可用的空间

```
= RcvWindow
= RcvBuffer-[LastByteRcvd - LastByteRead]
```


第3章：提纲

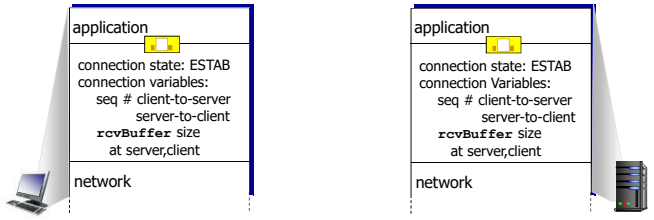
3.1 概述和传输层服务
3.2 多路复用与解复用
3.3 无连接传输：UDP
3.4 可靠数据传输的原理
3.5 面向连接的传输：TCP
- 段结构
- 可靠数据传输
- 流量控制
- 连接管理

3.6 拥塞控制原理
3.7 TCP 拥塞控制

TCP连接管理

在正式交换数据之前，发送方和接收方握手建立通信关系:

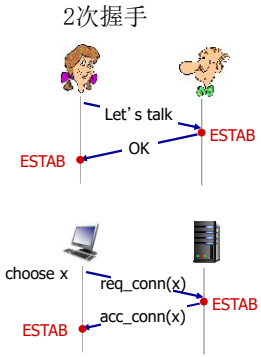
- 同意建立连接（每一方都知道对方愿意建立连接）
- 同意连接参数




```
Socket clientSocket =
  newSocket("hostname","port
  number");
```

```
Socket connectionSocket =
  welcomeSocket.accept();
```


同意建立连接

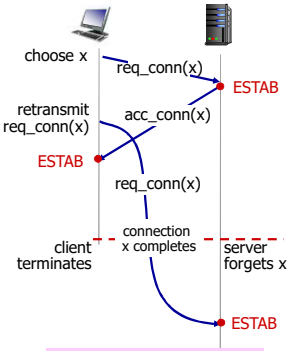


Q:在网络中，2次握手建立连接总是可行吗？

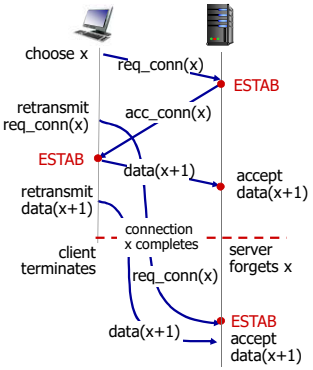
- 变化的延迟（连接请求的段没有丢，但可能超时）
- 由于丢失造成的重传 (e.g. req_conn(x))
- 报文乱序
- 相互看不到对方

同意建立连接

2次握手的失败场景：

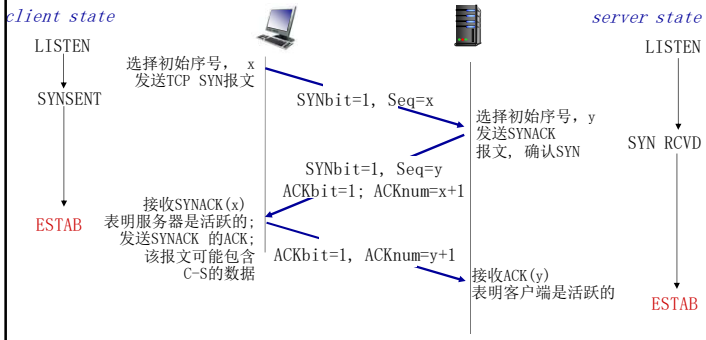


半连接
(只在服务器端维护了连接!)



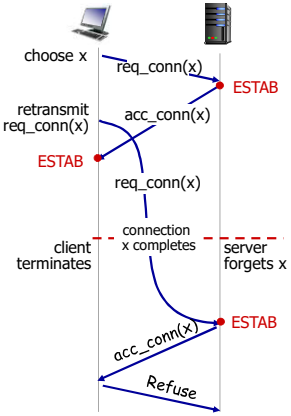
老的数据被当成新的数据接受了

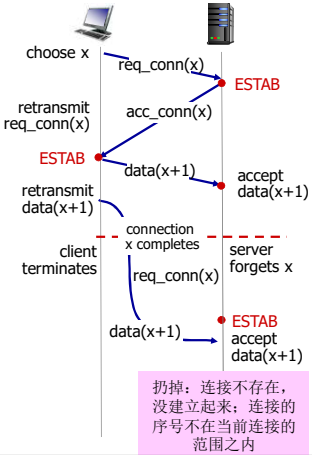
TCP 3次握手



解决方案：变化的初始序号+双方确认对方的序号（3次握手）

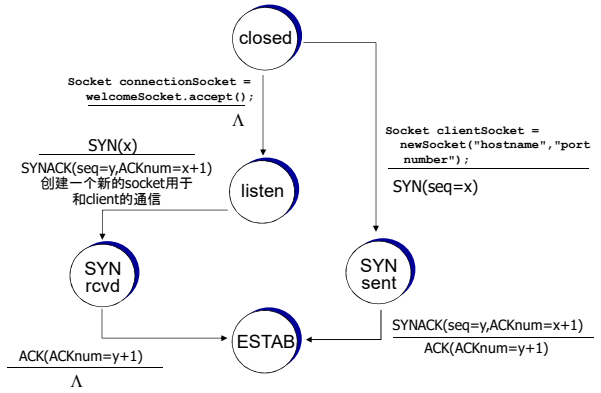
3次握手解决：半连接和接收老数据问题





扔掉：连接不存在，
没建立起来；连接的
序号不在当前连接的
范围之内

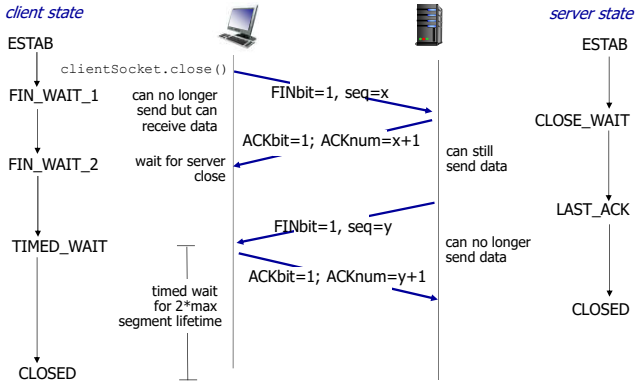
TCP 3次握手：FSM



TCP：关闭连接

- 客户端，服务器分别关闭它自己这一侧的连接
  - 发送FIN bit = 1的TCP段
- 一旦接收到FIN，用ACK回应
  - 接到FIN段，ACK可以和它自己发出的FIN段一起发送
- 可以处理同时的FIN交换

TCP：关闭连接



第3章：提纲

3.1 概述和传输层服务
3.2 多路复用与解复用
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
- 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

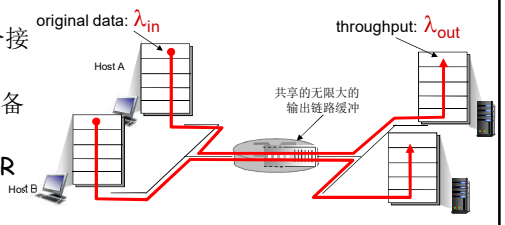
拥塞控制原理

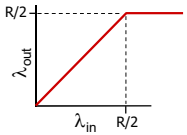
拥塞：

- 非正式的定义："太多的数据需要网络传输，超过了网络的处理能力"
- 与流量控制不同
- 拥塞的表现：
  - 分组丢失 (路由器缓冲区溢出)
  - 分组经历比较长的延迟(在路由器的队列中排队)
- 网络中前10位的问题!

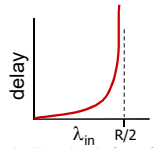
拥塞的原因/代价: 场景1

- 2个发送端，2个接收端
- 一个路由器，具备无限大的缓冲
- 输出链路带宽：R
- 没有重传





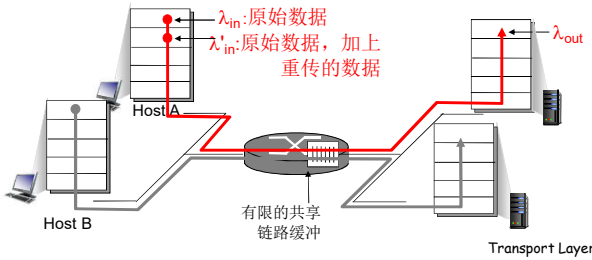
- 每个连接的最大吞吐量：R/2



- 当进入的速率λ_{in}接近链路链路带宽R时，延迟增大

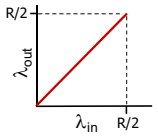
拥塞的原因/代价: 场景2

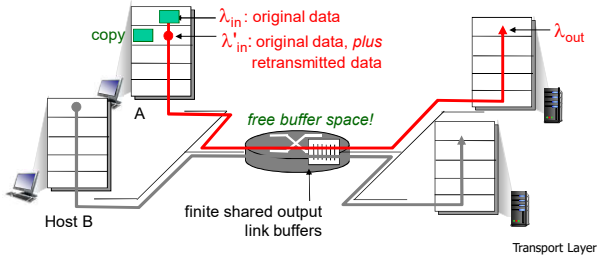
- 一个路由器，**有限的**缓冲
- 分组丢失时，发送端重传
  - 应用层的输入=应用层输出：$\lambda_{in} = \lambda_{out}$
  - 传输层的输入包括重传：$\lambda'_{in} \geq \lambda_{in}$



拥塞的原因/代价: 场景2

理想化：发送端有完美的信息
发送端知道什么时候路由器的缓冲是可用的
- ✓ 只在缓冲可用时发送
- ✓ 不会丢失：$\lambda'_{in} = \lambda_{in}$



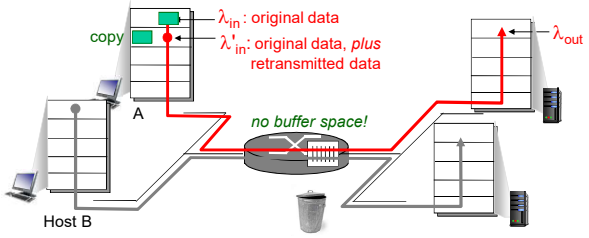


拥塞的原因/代价: 场景2

理想化：掌握丢失信息

分组可以丢失，在路由器由于缓冲器满而被丢弃

- 如果知道分组丢失了，发送方重传分组

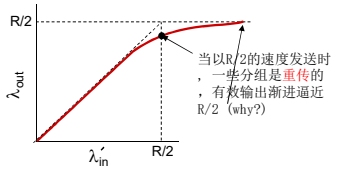


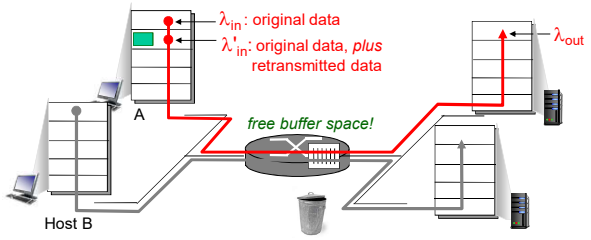
拥塞的原因/代价: 场景2

理想化：掌握丢失信息

分组可以丢失，在路由器由于缓冲器满而被丢弃

- 如果发送端知道分组丢失了，发送方重传分组

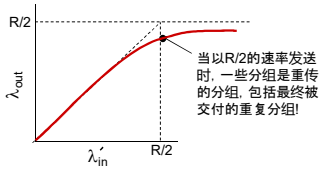


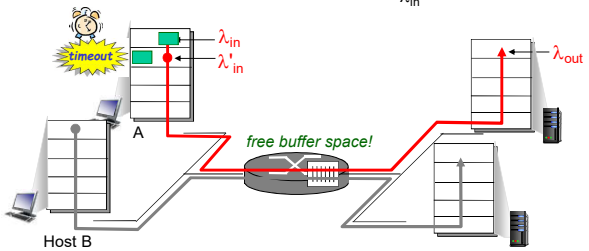


拥塞的原因/代价: 场景2

现实情况：重复

- 分组可能丢失，由于缓冲器满而被丢弃
- 发送端最终超时，发送第2个拷贝，2个分组都被传出

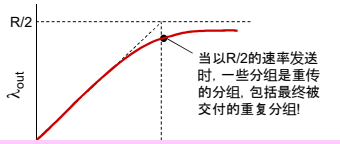




拥塞的原因/代价: 场景2

现实情况：重复

- 分组可能丢失，由于缓冲器满而被丢弃
- 发送端最终超时，发送第2个拷贝，2个分组都传到



输出比输入少原因：1）重传的丢失分组；2）没有必要重传的重复分组

拥塞的“代价”：

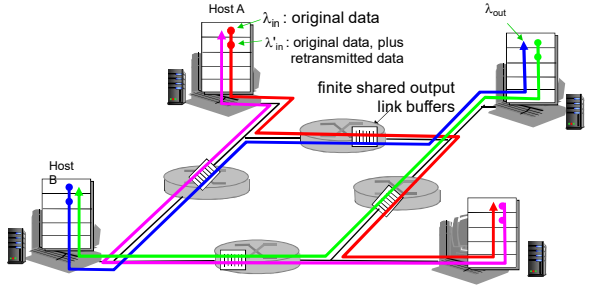
- 为了达到一个有效输出，网络需要做更多的工作（重传）
- 没有必要的重传，链路中包括了多个分组的拷贝
  - 是那些没有丢失，经历的时间比较长（拥塞状态）但是超时的分组
  - 降低了的“goodput”

拥塞的原因/代价: 场景3

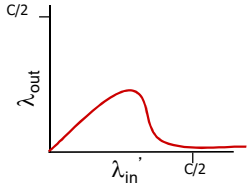
- 4个发送端
- 多重路径
- 超时 / 重传

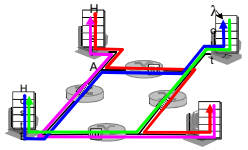
Q: 当 λ_{in} λ'_{in} 增加时，会发生什么?

A: 当红色的λ'_{in}增加时，所有到来的蓝色分组都在最上方的队列中丢弃了，蓝色吞吐->0



拥塞的原因/代价: 场景3





又一个拥塞的代价:

- 当分组丢失时，任何"关于这个分组的上游传输能力"都被浪费了

拥塞控制方法

2种常用的拥塞控制方法:

端到端拥塞控制:

- 没有来自网络的显式反馈
- 端系统根据延迟和丢失事件推断是否有拥塞
- TCP采用的方法

网络辅助的拥塞控制:

- 路由器提供给端系统以反馈信息
  - 单个bit置位，显示有拥塞 (SNA, DECbit, TCP/IP ECN, ATM)
  - 显式提供发送端可以采用的速率

案例学习: ATM ABR 拥塞控制

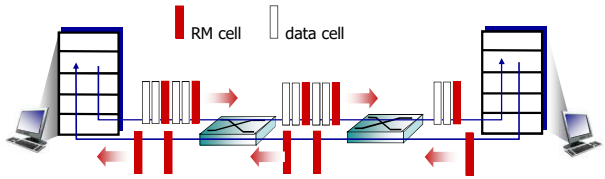
ABR: available bit rate:

- "弹性服务"
- 如果发送端的路径 "轻载"
  - 发送方使用可用带宽
- 如果发送方的路径拥塞了
  - 发送方限制其发送的速度到一个最小保障速率上

RM (资源管理) 信元:

- 由发送端发送,在数据信元中间隔插入
- RM信元中的比特被交换机设置 ("网络辅助")
  - NI bit: no increase in rate (轻微拥塞)速率不要增加了
  - CI bit: congestion indication 拥塞指示
- 发送端发送的RM 信元被接收端返回, 接收端不做任何改变

案例学习: ATM ABR 拥塞控制



- 在RM信元中的2个字节 ER (explicit rate)字段
  - 拥塞的交换机可能会降低信元中ER的值
  - 发送端发送速度因此是最低的可支持速率
- 数据信元中的EFCI bit: 被拥塞的交换机设置成1
  - 如果在管理信元RM前面的数据信元EFCI被设置成了1, 接收端在返回的RM信元中设置CI bit

第3章：提纲

3.1 概述和传输层服务
3.2 多路复用与解复用
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
- 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

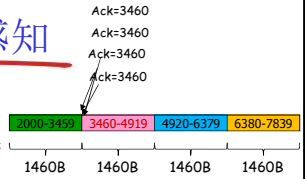
TCP 拥塞控制：机制

- 端到端的拥塞控制机制
  - 路由器不向主机有关拥塞的反馈信息
    - 路由器的负担较轻
    - 符合网络核心简单的TCP/IP架构原则
  - 端系统根据自身得到的信息，判断是否发生拥塞，从而采取动作

拥塞控制的几个问题
- 如何检测拥塞
  - 轻微拥塞
  - 拥塞
- 控制策略
  - 在拥塞发送时如何动作，降低速率
    - 轻微拥塞，如何降低
    - 拥塞时，如何降低
  - 在拥塞缓解时如何动作，增加速率

TCP 拥塞控制：拥塞感知

发送端如何探测到拥塞?

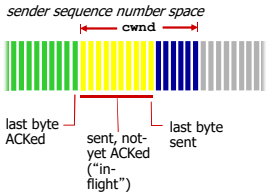


- 某个段超时了（丢失事件）：拥塞
  - 超时时间到，某个段的确认没有来
  - 原因1：网络拥塞（某个路由器缓冲区没空间了，被丢弃）概率大
  - 原因2：出错被丢弃了（各级错误，没有通过校验，被丢弃）概率小
  - 一旦超时，就认为拥塞了，有一定误判，但是总体控制方向是对的
- 有关某个段的3次重复ACK：轻微拥塞
  - 段的第1个ack，正常，确认绿段，期待红段
  - 段的第2个重复ack，意味着红段的后一段收到了，蓝段乱序到达
  - 段的第2、3、4个ack重复，意味着红段的后第2、3、4个段收到了，橙段乱序到达，同时红段丢失的可能性很大（后面3个段都到了，红段都没到）
  - 网络这时还能够进行一定程度的传输，拥塞但情况要比第一种好

TCP 拥塞控制：速率控制方法

如何控制发送端发送的速率
- 维持一个拥塞窗口的值：CongWin
- 发送端限制已发送但是未确认的数据量（的上限）：

  `LastByteSent-LastByteAcked ≤ CongWin`
- 从而粗略地控制发送方的往网络中注入的速率



$$\text{rate} \approx \frac{\text{CongWin}}{\text{RTT}} \text{bytes/sec}$$

TCP 拥塞控制：速率控制方法

如何控制发送端发送的速率

- CongWin是动态的，是感知到的网络拥塞程度的函数
  - 超时或者3个重复ack，CongWin↓
    - 超时时：**CongWin降为1MSS，进入SS阶段然后再倍增到CongWin/2（每个RTT），从而进入CA阶段**
    - **3个重复ack ：CongWin降为CongWin/2，CA阶段**
  - 否则（正常收到Ack，没有发送以上情况）：CongWin跃跃欲试↑
    - **SS阶段：加倍增加（每个RTT）**
    - **CA阶段：线性增加（每个RTT）**

TCP拥塞控制和流量控制的联合动作

联合控制的方法：

- 发送端控制*发送但是未确认*的量同时也不能够超过接收窗口，满足流量控制要求
  - SendWin=min{CongWin, RecvWin}
  - 同时满足 拥塞控制和流量控制要求

TCP 拥塞控制：策略概述

拥塞控制策略：

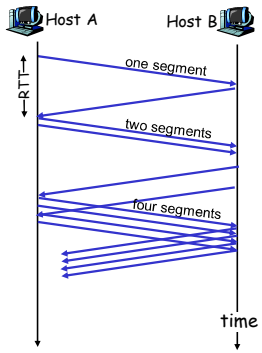
- 慢启动
- AIMD：线性增、乘性减少
- 超时事件后的保守策略

TCP 慢启动

- 连接刚建立，**CongWin = 1 MSS**
  - 如: MSS = 1460bytes & RTT = 200 msec
  - 初始速率 = 58.4kbps
- 可用带宽可能>> MSS/RTT
  - 应该尽快加速，到达希望的速率
- 当连接开始时，指数性增加发送速率，直到发生丢失的事件
  - 启动初值很低
  - 但是速度很快

TCP 慢启动（续）

- 当连接开始时，指数性增加（每个RTT）发送速率直到发生丢失事件
  - 每一个RTT， CongWin加倍
  - 每收到一个ACK时，CongWin**加1**（why）
  - 慢启动阶段：只要不超时或3个重复ack，一个RTT，CongWin加倍
- 总结: 初始速率很慢，但是加速却是指数性的
  - 指数增加，SS时间很短，长期来看可以忽略



TCP 拥塞控制：AIMD

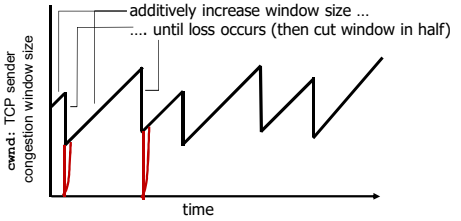
乘性减:

丢失事件后将CongWin降为1，将CongWin/2作为阈值，进入慢启动阶段（倍增直到CongWin/2）

加性增：

当CongWin>阈值时，一个RTT如没有发生丢失事件，将**CongWin**加1MSS: *探测*

AIMD saw tooth behavior: probing for bandwidth



TCP拥塞控制：AIMD

- 当收到3个重复的*ACKs*:
  - **CongWin** 减半
  - 窗口（缓冲区大小）之后线性增长
- 当超时事件发生时:
  - **CongWin被设置成 1 MSS**，进入SS阶段
  - 之后窗口指数增长
  - 增长到一个阈值（上次发生拥塞的窗口的一半）时，再线性增加

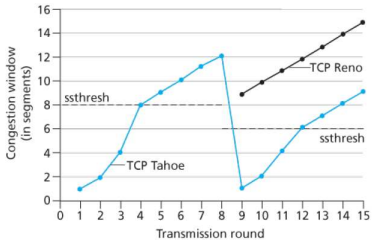
思路

- 3个重复的ACK表示网络还有一定的段传输能力
- 超时之前的3个重复的ACK表示“警报”

改进(续)

Q:什么时候应该将指数性增长变成线性？

A:在超时之前，当**CongWin变成上次发生超时的窗口的一半**



实现:

- 变量：Threshold
- 出现丢失，Threshold设置成 CongWin的1/2

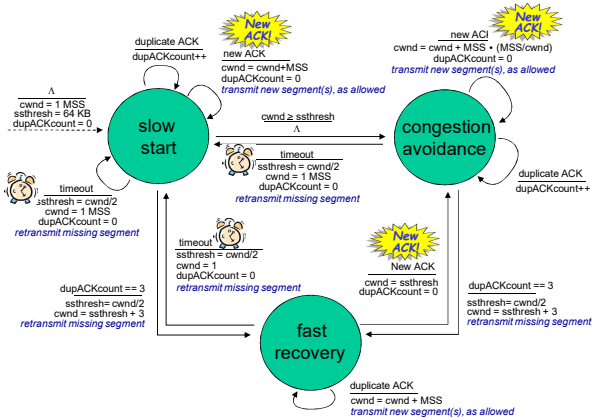
总结: TCP拥塞控制

- 当**CongWin<Threshold**, 发送端处于慢启动阶段（**slow-start**）, 窗口指数性增长.
- 当**CongWin> Threshold**, 发送端处于拥塞避免阶段（**congestion-avoidance**）, 窗口线性增长.
- 当收到三个重复的*ACKs* **(triple duplicate ACK), Threshold设置成 CongWin/2，CongWin=Threshold+3.**
- 当超时事件发生时**timeout**, **Threshold=CongWin/2 CongWin=1 MSS**，进入SS阶段

TCP 发送端拥塞控制

事件	状态	TCP 发送端行为	解释
以前没有收到ACK的data被ACKed	慢启动(SS)	CongWin = CongWin + MSS If (CongWin > Threshold) 状态变成 “CA”	每一个RTT CongWin 加倍
以前没有收到ACK的data被ACKed	拥塞避免 (CA)	CongWin = CongWin+MSS * (MSS/CongWin)	加性增加, 每一个RTT对CongWin 加一个 1 MSS
通过收到3个重复的ACK，发现丢失的事件	SS or CA	Threshold = CongWin/2, CongWin = Threshold+3, 状态变成 “CA”	快速重传, 实现乘性的减. CongWin 没有变成1 MSS.
超时	SS or CA	Threshold = CongWin/2, CongWin = 1 MSS, 状态变成 “SS”	进入slow start
重复的 ACK	SS or CA	对被ACKed 的segment， 增加重复ACK的计数	CongWin and Threshold 不变

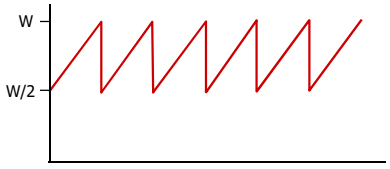
总结： TCP拥塞控制



TCP 吞吐量

- TCP的平均吞吐量是多少，使用窗口window尺寸W和RTT来描述?
  - 忽略慢启动阶段，假设发送端总有数据传输
- W：发生丢失事件时的窗口尺寸（单位：字节）
  - 平均窗口尺寸（#in-flight字节）：3/4W
  - 平均吞吐量：RTT时间吞吐3/4W

$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$



TCP 未来: TCP over “long, fat pipes”

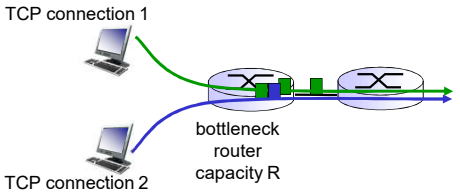
- 例如: 1500字节 / 段, 100ms RTT, 如果需要10 Gbps 吞吐量
- T=0.75W/R -> W=TR/0.75=12.5M字节=83333段
- 需要窗口大小 W = 83,333 in-flight 段
- 吞吐量用丢失率表示:

$$T = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- ➜ L = $2 \cdot 10^{-10}$ （为了达到10Gbps的吞吐，平均50亿段丢失一个） *非常非常小的丢失率！可能远远低于链路的物理丢失率，达不到的*
- 网络带宽增加，需要更新的TCP版本!

TCP 公平性

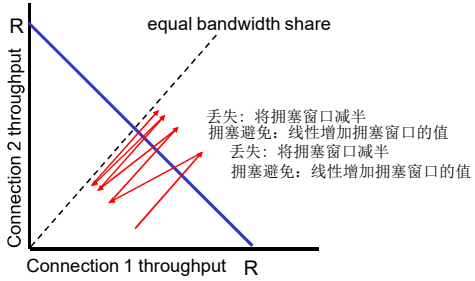
公平性目标: 如果 K个TCP会话分享一个链路带宽为R的瓶颈，每一个会话的有效带宽为 R/K



为什么TCP是公平的?

2个竞争的TCP会话:

- 加性增加，斜率为1, 吞吐量增加
- 乘性减，吞吐量比例减少



公平性 (续)

公平性和 UDP

- 多媒体应用通常不是用 TCP
  - 应用发送的数据速率希望不受拥塞控制的节制
- 使用UDP:
  - 音视频应用泵出数据的速率是恒定的, 忽略数据的丢失
- 研究领域: TCP 友好性

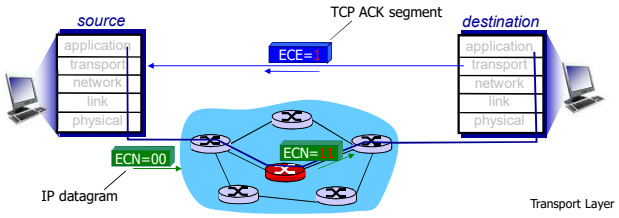
公平性和并行TCP连接

- 2个主机间可以打开多个并行的TCP连接
- Web浏览器
- 例如: 带宽为R的链路支持了9个连接;
  - 如果新的应用要求建1个TCP连接,获得带宽R/10
  - 如果新的应用要求建11个TCP连接,获得带宽R/2

Explicit Congestion Notification (ECN)

网络辅助拥塞控制:

- TOS字段中2个bit被网络路由器标记，用于指示是否发生拥塞
- 拥塞指示被传送到接收主机
- 在接收方-到发送方的ACK中，接收方(在IP数据报中看到了拥塞指示）设置ECE bit，指示发送方发生了拥塞



第三章 总结(1/2)

- 传输层提供的服务
  - 应用进程间的逻辑通信
    - Vs 网络层提供的是主机到主机的通信服务
  - 互联网上传输层协议：UDP TCP
    - 特性
- 多路复用和解复用
  - 端口：传输层的SAP
  - 无连接的多路复用和解复用
  - 面向连接的多路复用和解复用
- 实例1：无连接传输层协议 UDP
  - 多路复用解复用
  - UDP报文格式
  - 检错机制：校验和
- 可靠数据传输原理
  - 问题描述
  - 停止等待协议：
    - Rdt1.0 rdt2.0,2.1 ,2.2 Rdt 3.0
  - 流水线协议
    - GBN
    - SR

第三章 总结（2/2）

- 实例2：面向连接的传输层协议-TCP
  - 概述：TCP特性
  - 报文段格式
    - 序号，超时机制及时间
  - TCP可靠传输机制
  - 重传，快速重传
  - 流量控制
  - 连接管理
    - 三次握手
    - 对称连接释放
- 拥塞控制原理
  - 网络辅助的拥塞控制
  - 端到端的拥塞控制
- TCP的拥塞控制
  - AIMD
  - 慢启动
  - 超时之后的保守策略

第三章 展望

下二章:

- 离开网络"边缘" (应用层和传输层)
- 深入到网络的"核心"
- 2个关于网络层的章
  - 数据平面
  - 控制平面