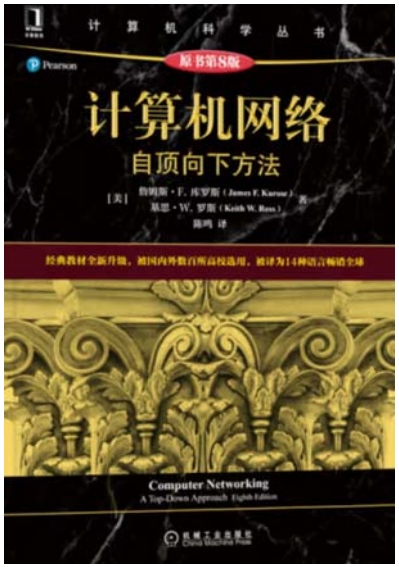
Chapter 3
运输层/传输层
Transport Layer



第三章：传输层

学习目标:

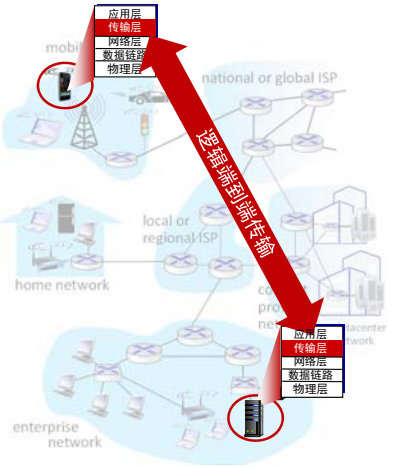
- 理解传输层服务原理:
  - 多路复用 multiplexing, 多路分解 demultiplexing
  - 可靠数据传输 reliable data transfer
  - 流量控制 flow control
  - 拥塞控制 congestion control
- 学习因特网中的传输层协议：
  - UDP: 无连接传输
  - TCP : 面向连接的可靠传输
  - TCP拥塞控制

第三章 传输层内容

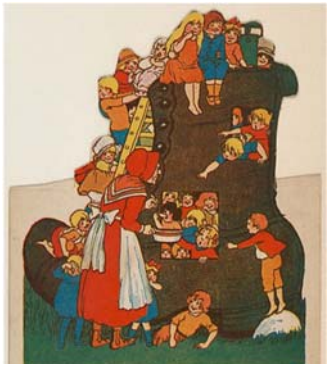
- **传输层服务**
- 多路复用和多路分解
- 无连接传输：UDP
- 可靠数据传输原则
- 面向连接的传输：TCP
- 拥塞控制原则
- TCP拥塞控制

传输层服务和协议

- 传输层协议为运行在不同主机上的应用**进程**之间提供了*逻辑通信 logical communication*
- 传输层协议是在端系统中而不是在路由器中实现的
  - 发送方: 传输层将从发送应用程序进程接收到的报文转换成传输层报文段segment
  - 接收方:网络层从数据报中提取传输层报文段，并将该报文段向上交给传输层。传输层则处理接收到的报文段，使该报文段中的数据为接收应用进程使用。
- 网络应用程序可以使用多种传输层协议
  - Internet: TCP，UDP



传输层和网络层的关系



邮政服务为两个家庭建提供逻辑通信
从一家送往另一家
而不是从一个人送往另一个人

家庭成员间寄信类比

安迪家的12个孩子给比尔家的12个孩子寄信：

- 主机（端系统） = 家庭
- 进程 = 兄弟姐妹
- 应用层报文 = 信（的字符）
- 传输层协议 = Andy 和 Bill
- 网络层协议 = 邮政服务（包括邮车）

每一个家庭有个孩子负责收发邮件，
两个家庭分别由Andy和Bill负责

传输层和网络层的关系

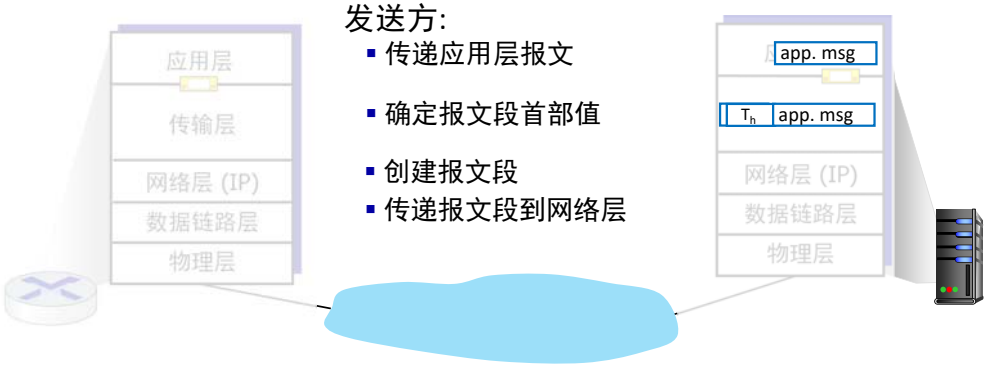
- 网络层: 主机（host）间的逻辑通信
- 传输层: 进程（process）间的逻辑通信
  - 依赖网络层服务

家庭成员间寄信类比

安迪家的12个孩子给比尔家的12个孩子寄信：

- 主机 = 家庭
- 进程 = 兄弟姐妹
- 应用层报文 = 信（的字符）
- 传输层协议 = Andy 和 Bill
- 网络层协议 = 邮政服务（包括邮车）

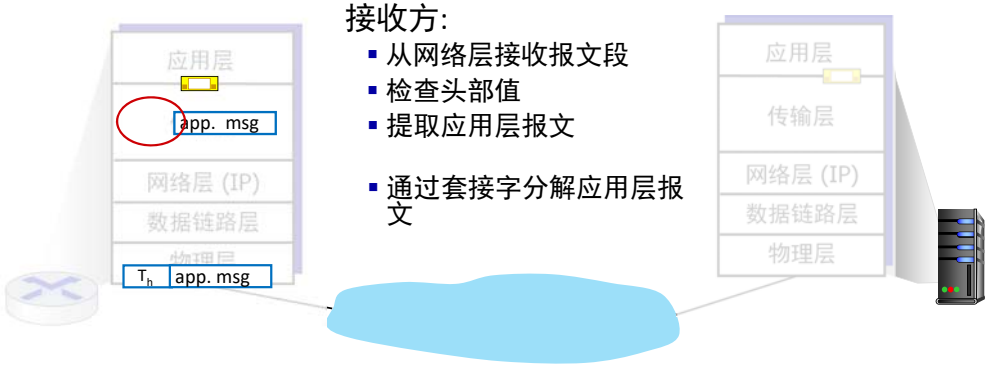
传输层行为



发送方:

- 传递应用层报文
- 确定报文段首部值
- 创建报文段
- 传递报文段到网络层

传输层行为

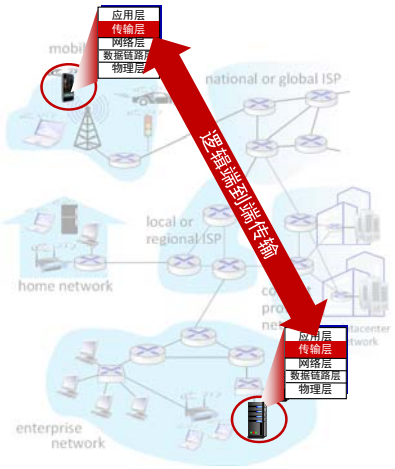


接收方:

- 从网络层接收报文段
- 检查头部值
- 提取应用层报文
- 通过套接字分解应用层报文

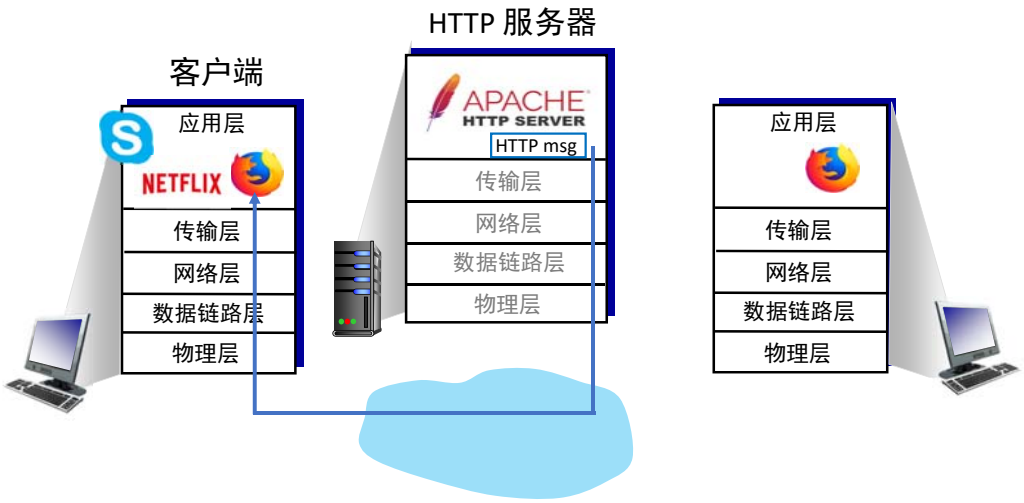
传输层协议

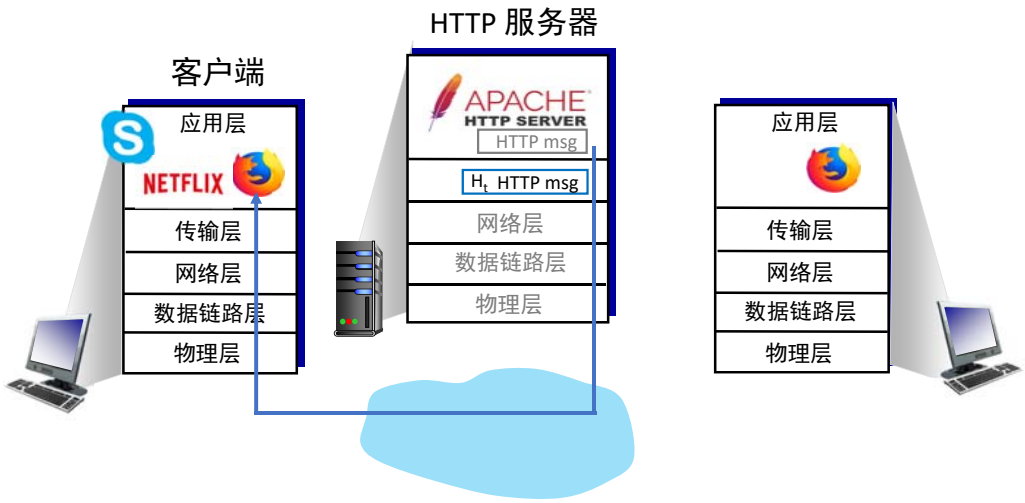
- TCP提供可靠数据交付、有序交付
  - 拥塞控制
  - 流量控制
  - 建立连接
- UDP提供不可靠交付、无序交付
  - 尽力而为的IP的简单扩展
- 不提供的服务:
  - 时延保证
  - 带宽保证

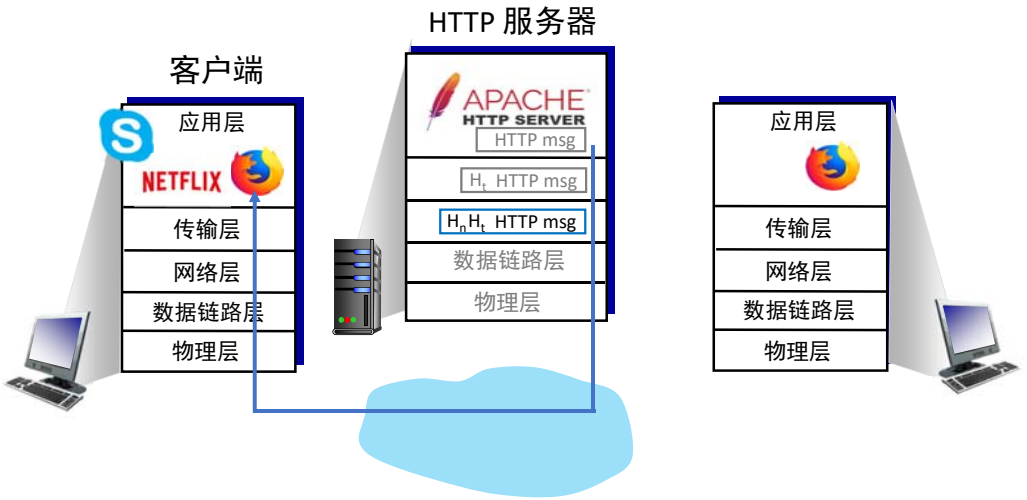


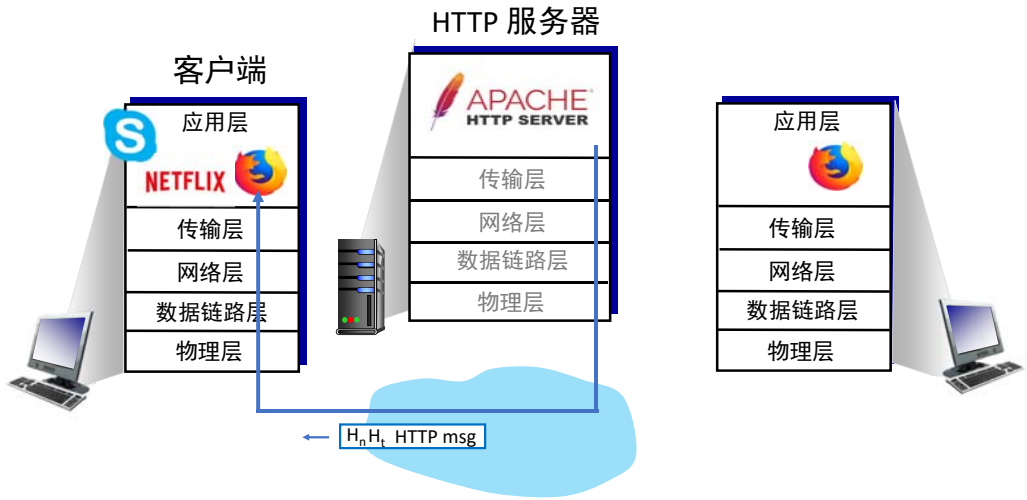
第三章：内容

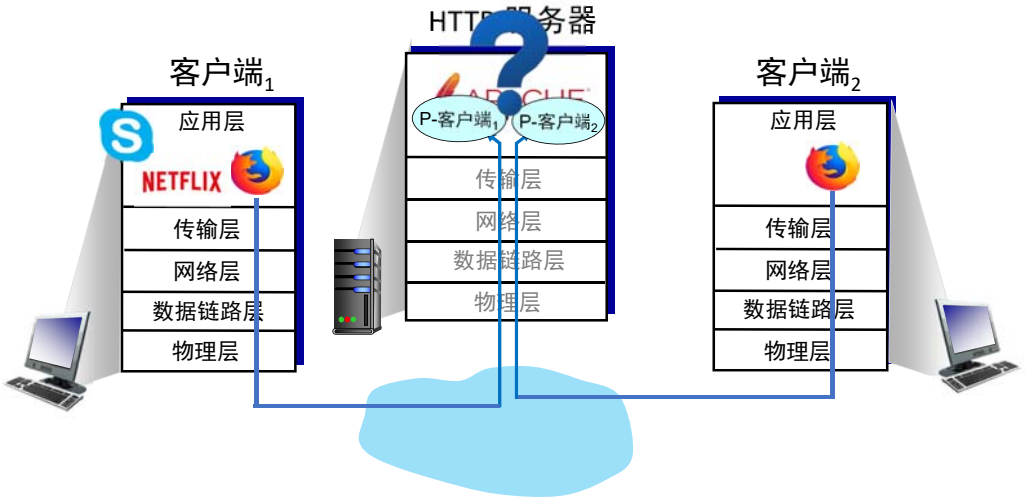
- 传输层服务
- **多路复用和多路分解**
- 无连接传输：UDP
- 可靠数据传输原理
- 面向连接的传输：TCP
- 拥塞控制原理
- TCP拥塞控制











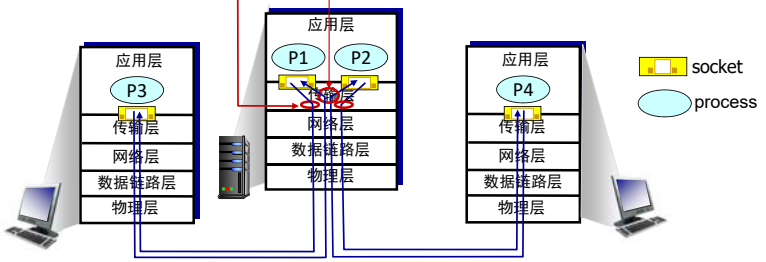
多路复用/多路分解
Multiplexing/demultiplexing

发送方多路复用

在源主机从不同套接字中收集数据块，并为每个数据块封装上首部信息(这将在以后用于分解)从而生成报文段，然后将报文段传递到网络层

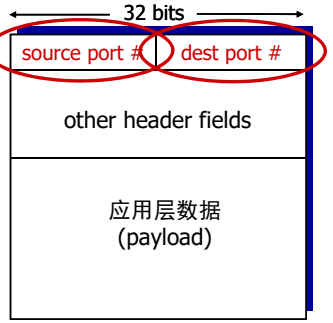
接收方多路分解

将传输层报文段中的数据根据套接字交付到正确的应用程序进程



多路分解工作模式

- 主机接收 IP 数据报(datagrams)
  - 每个数据报包含源IP地址和目的IP地址
  - 每个数据报携带一个传输层报文段(segment)
  - 每个报文段包含源端口号和目的端口号
  -
- 接收端的传输层检查IP地址和端口号字段，进而将报文段定向到特定的套接字



TCP/UDP 报文段格式

无连接的多路分解

回顾:

- 创建包含端口号的套接字:

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

- 一个UDP套接字由一个二元组标识:
  - 目的IP地址
  - 目的端口号

当主机接收UDP报文段时:

- 检查该报文段的目的端口号
- 将该UDP报文段交付给该端口号所标识的套接字

⬇

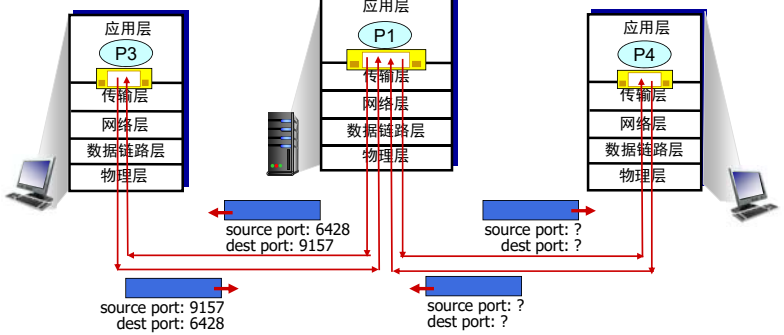
如果两个UDP报文段有不同的源IP地址和/或源端口号，但具有相同的目的IP地址和目的端口号，那么这两个报文段将通过相同的目的套接字被定向到相同的目的进程

无连接多路分解的例子

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

```
DatagramSocket 服务器Socket
 = new DatagramSocket
 (6428);
```

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

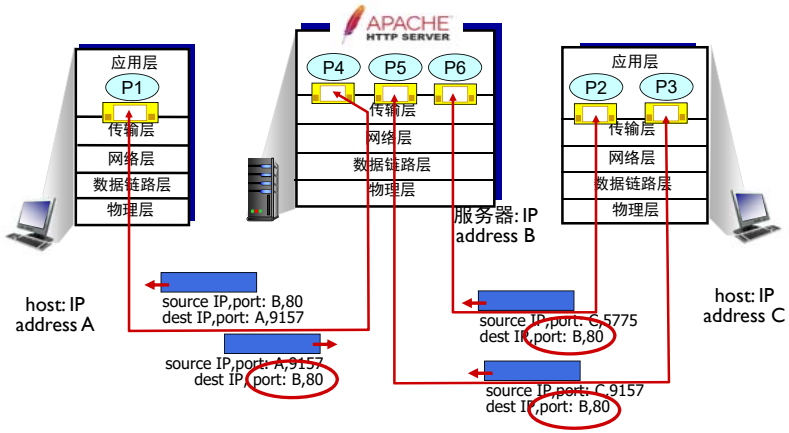


面向连接的多路分解

- 一个TCP 套接字由一个四元组标识:
  - 源IP地址source IP address
  - 源端口号source port number
  - 目的IP地址dest IP address
  - 目的端口号dest port number
- 分解:主机使用全部4个值来将报文段定向（分解） 到相应的套接字

- 服务器主机可以支持很多并行的TCP套接字：
  - 由其四元组来标识每个套接字
  - 每个套接字与一个进程相联系

面向连接多路分解的例子



使用不同的套接字对三个目的IP地址都是B,端口号都是80的报文段进行分解

总结

- 多路复用, 多路分解: 基于报文段，数据报头部内容
- **UDP:** 仅使用目的端口号进行多路分解
- **TCP:** 使用（源IP地址，源端口号，目的IP地址，目的端口号）进行多路分解
- 多路复用/多路分解在所有层都适用

第三章：内容

- 传输层服务
- 多路复用和多路分解
- **无连接传输：UDP**
- 可靠数据传输原理
- 面向连接的传输：TCP
- 拥塞控制原理
- TCP拥塞控制

用户数据报协议
UDP: User Datagram Protocol [RFC 768]

- 最简化的传输层协议
- 提供尽力而为的服务，UDP报文段可能
  - 丢包
  - 对应用程序交付失序
- *无连接 connectionless:*
  - 在UDP发送方和接收方之间无握手
  - 每个UDP报文段的处理独立于其他报文段

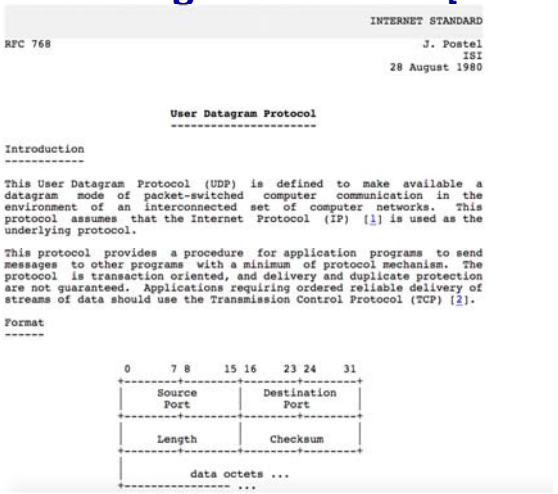
为什么要有UDP协议?

- 没有连接的建立（连接将增加时延）
- 简单：在发送方、接收方无连接状态
- 分组首部开销小
- 无拥塞控制：
  - UDP能够尽可能快地传输

用户数据报协议
UDP: User Datagram Protocol [RFC 768,1980]

- UDP 应用:
  - 流式多媒体应用（丢包容忍，速率敏感）
  - DNS
  - SNMP
  - HTTP/3
- 如果需要通过UDP进行可靠传输 (e.g., HTTP/3):
  - 在应用层添加必要的可靠机制
  - 在应用层添加拥塞控制

用户数据报协议
UDP: User Datagram Protocol [RFC 768，1980]



INTERNET STANDARD

RFC 768 — J. Postel, ISI, 28 August 1980

User Datagram Protocol

Introduction

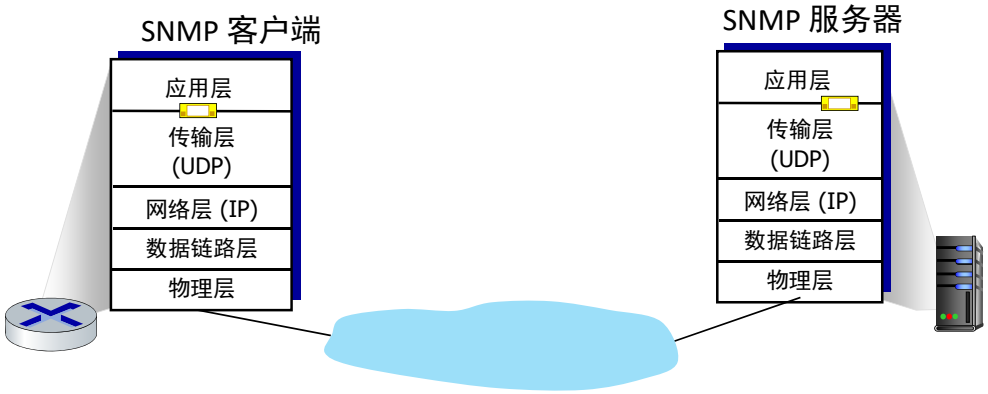
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

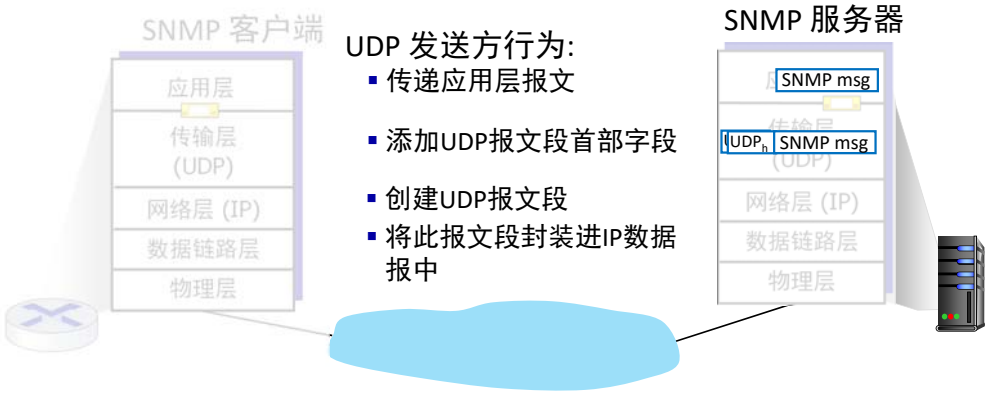
Format

只有2.5页，可以读下。

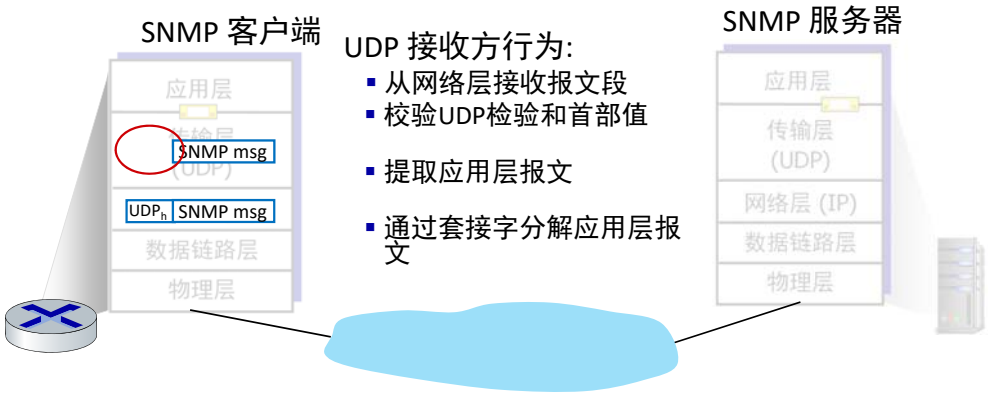
UDP: 传输层行为



UDP:传输层行为



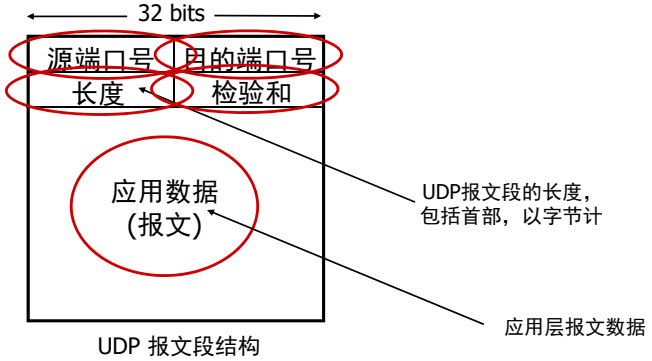
UDP: 传输层行为



UDP 接收方行为:

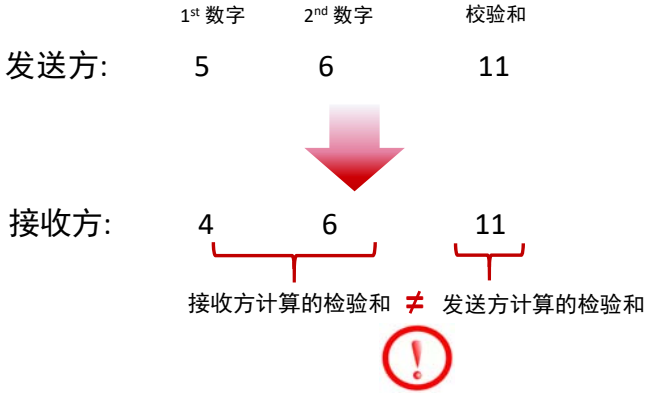
- 从网络层接收报文段
- 校验UDP检验和首部值
- 提取应用层报文
- 通过套接字分解应用层报文

UDP报文段的结构



UDP 检验和

目的: 在传输的报文段中检测“差错”（如比特翻转）



UDP 检验和

*目的:*在传输的报文段中检测“差错”（如比特翻转）

发送方:

- 将报文段内容处理为16比特整数序列
- 检验和 checksum: 报文段内容的加法（反码和）
- 发送方将检验和放入UDP检验和字段

接收方:

- 计算接收的报文段的检验和
- 核对计算的检验和是否等于检验和字段的值：
  - 不相等 - 检测到差错
  - 相等 – 未检测到差错。虽然如此，但是否可能会有差错？

校验和的例子

例子: 两个16bit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            ---------------------------------
wraparound    1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
              ---------------------------------
和sum           1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
校验和checksum  0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```

注意：当数字作加法时，最高位的进位需要加到结果中

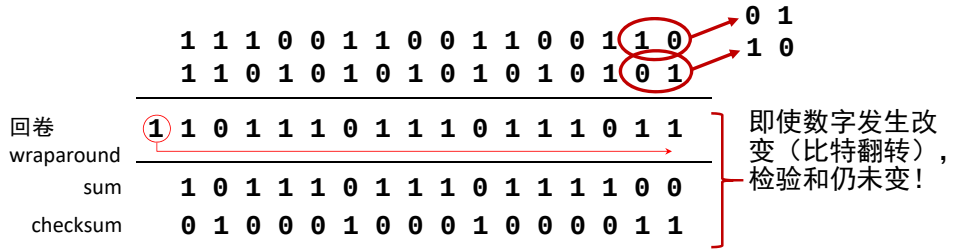
校验和 是 对和进行反码运算的结果

在接收方，将所有16bit字（包括检验和）加在一起，正确结果应该是全1。若有0，则出现了差错。

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

网络的检验和: 弱保护!

例子: 两个16bit整数相加



总结: UDP

- “不加修饰”的协议:
  - 报文段可能丢失，可能失序
  - 尽力而为的服务
- UDP的优点:
  - 不需要初始化/握手，没有握手的往返时间RTT（ Round-Trip Time ）
  - 当网络服务受损时可以正常工作
  - 有一定的可靠性(检验和)
- 在UDP之上的应用层负责实现附加功能 (e.g., HTTP/3)

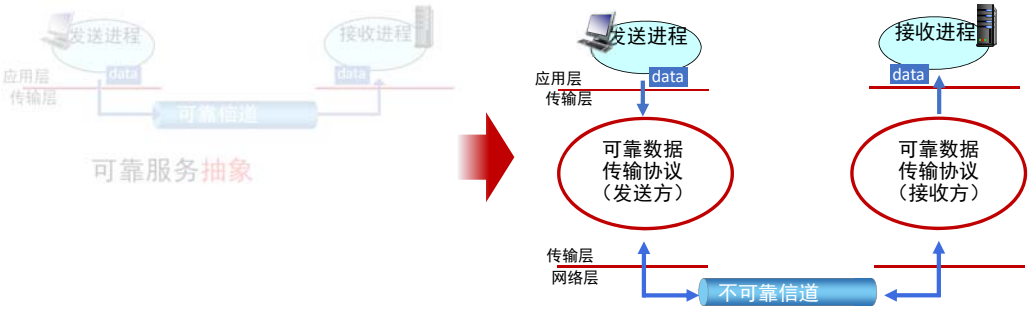
第三章：内容

- 传输层服务
- 多路复用和多路分解
- 无连接传输：UDP
- **可靠数据传输原理**
- 面向连接的传输：TCP
- 拥塞控制原理
- TCP拥塞控制

可靠数据传输

可靠服务抽象

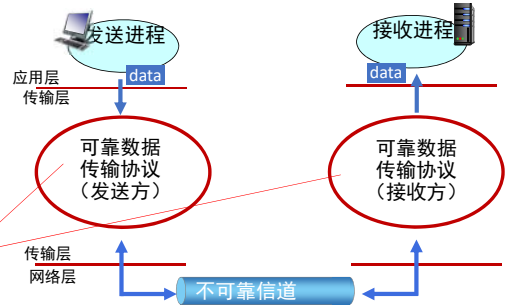
可靠数据传输



可靠服务的*实现*

可靠数据传输

可靠数据传输协议的复杂性（很大程度上）取决于不可靠信道的特性（丢失、损坏、重新排序数据？）

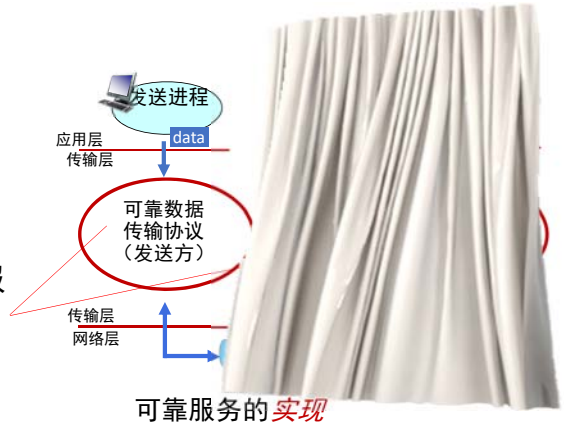


可靠服务的*实现*

可靠数据传输

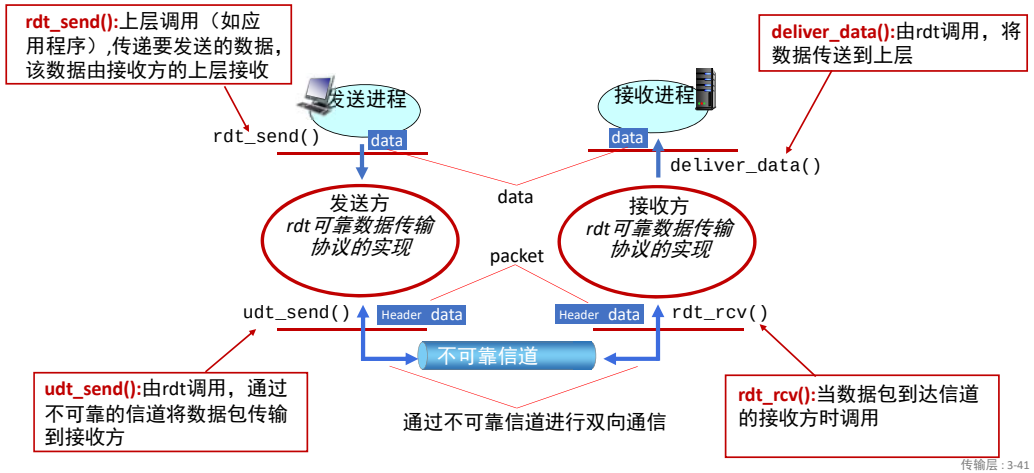
发送方，接收方不知道对方的“状态”，例如，是否收到报文？

- 除非通过信息传达



可靠服务的*实现*

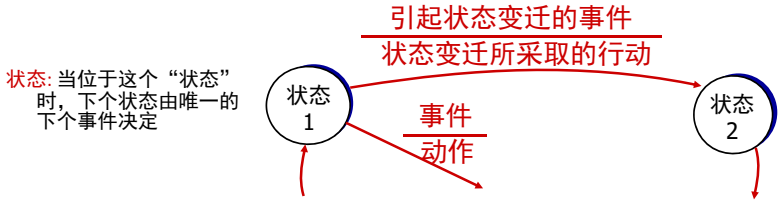
设计可靠数据传输协议(rdt): 接口



设计可靠数据传输协议: 基本概念

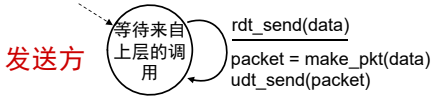
我们将:

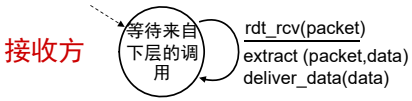
- 增量地设计和开发发送方、接收方的可靠数据传输协议
  - 仅考虑单向数据传输
  - 但控制信息将在两个方向流动!
- 使用有限状态机 finite state machines (FSM)来指定发送方和接收方



rdt1.0: 经完全可靠信道的可靠数据传输

- 底层信道非常可靠
  - 无比特差错
  - 无分组丢失
- 假定接收方和发送方的速率一样
- 分别为发送方和接收方定义FSM:
  - 发送方向底层信道发送数据
  - 接收方从底层信道接收数据





rdt2.0: 经具有比特差错信道的可靠数据传输

- 底层信道中分组中的比特可能受损
  - 使用检验和（例如因特网检验和）检测比特受损错误
- 问题:如何在错误中恢复过来?

人们在交流中如何恢复“错误”？

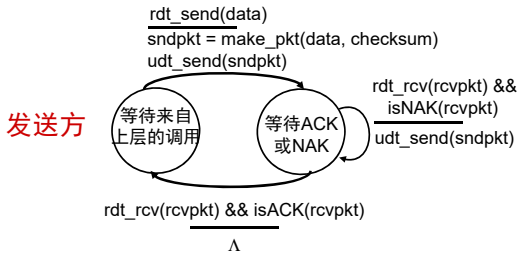
rdt2.0: 比特差错信道

- 底层信道可能会翻转分组中的比特
  - 使用检验和检测比特差错
- *问题：*如何从错误中恢复过来?
  - *肯定确认acknowledgements (ACKs):* 接收者告知发送方pkt正确接收
  - *否定确认negative acknowledgements (NAKs):* 接收者告知发送方pkt存在差错
  - 发送方在收到NAK反馈后*重新传输retransmits*

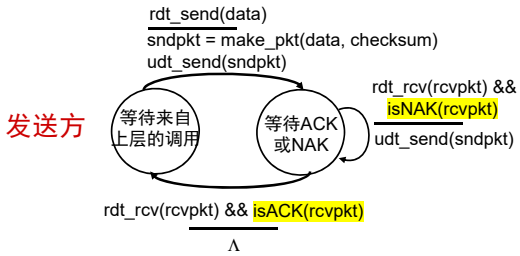
停等协议stop and wait

发送方发送一个分组，然后等待接收方响应

rdt2.0: FSM规范



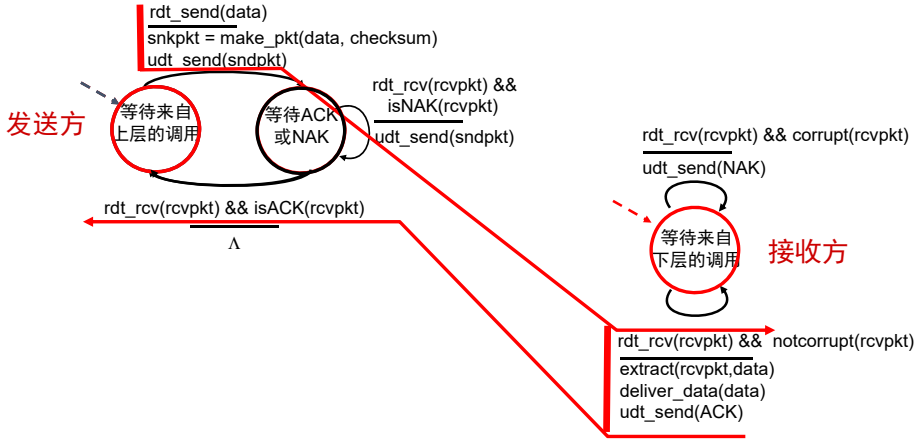
rdt2.0: FSM规范



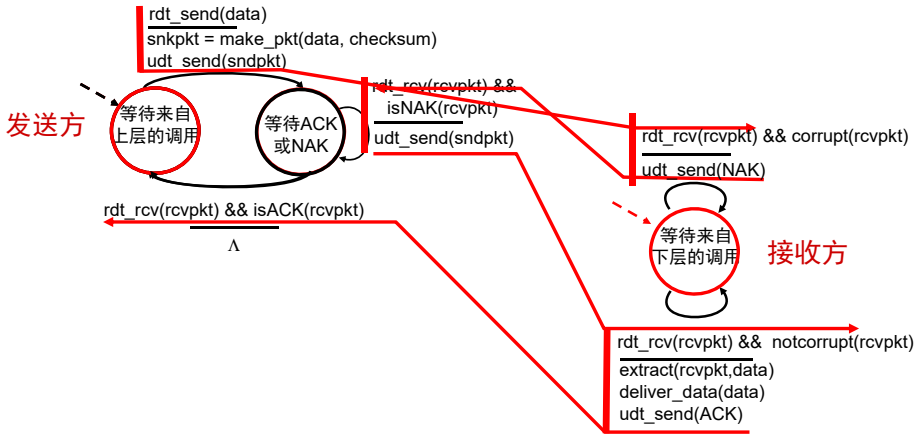
注意:接收方的“状态”（接收方是否正确地获得我的信息？）除非以某种方式由接收方通知发送方，否则发送方不知道

- 这就是我们需要一个协议的原因

rdt2.0: 接收到的数据没有错误情况下的操作



rdt2.0: 接收到的数据有比特差错情况下的操作



rdt2.0 有重大的缺陷!

如果ACK/NAK受损，将会出现何种情况?

- 发送方不知道在接收方会发生什么情况!
- 不能只是重传：可能导致冗余

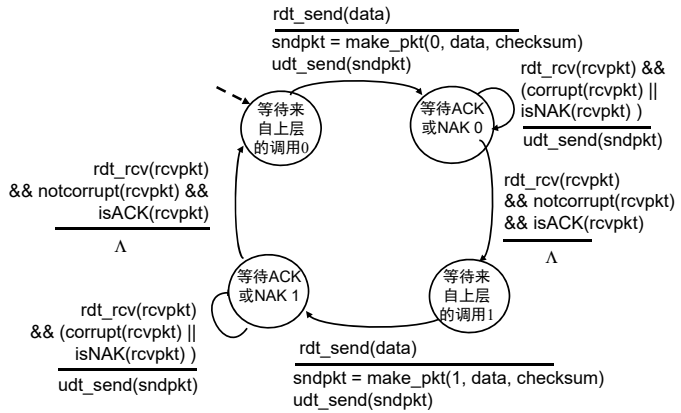
冗余分组如何处理:

- 如果ACK/NAK受损，发送方重传当前的分组，在通信信道中引入了冗余分组
- 发送方对每个分组增加*序列号 sequence number*
- 接收方丢弃冗余分组（不再向上交付）

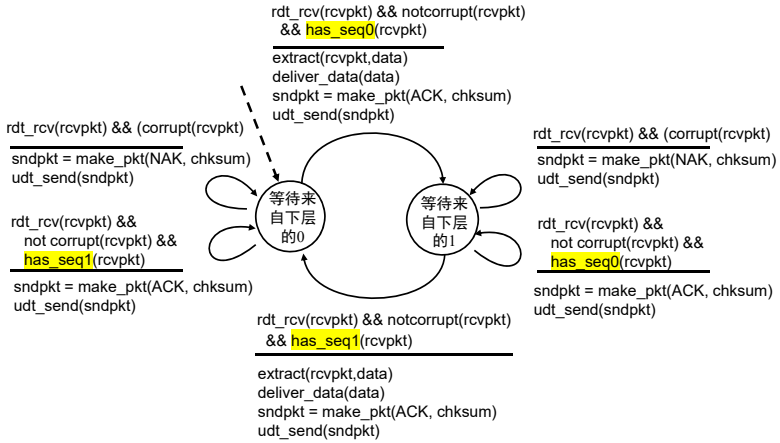
stop and wait

发送方发送一个分组，然后等待接收方响应

rdt2.1: 发送方, 包括处理受损的ACK/NAKs



rdt2.1: 接收方, 包括处理混乱的ACK/NAK



rdt2.1: 讨论

发送方:

- 序号seq # 加入分组中
- 两个序号seq #, (0,1)就够用，为什么?
- 必须检查是否收到的ACK/NAK受损
- 状态增加一倍
  - 状态必须“记住”是否“当前的”分组具有0或1序号

接收方:

- 必须检查是否接收到的分组是冗余的
  - 状态指示是否0或1是所期待的分组序号seq #
- 注意：接收方不能知道它发送的最后一个ACK/NAK是否发送方已经接收了

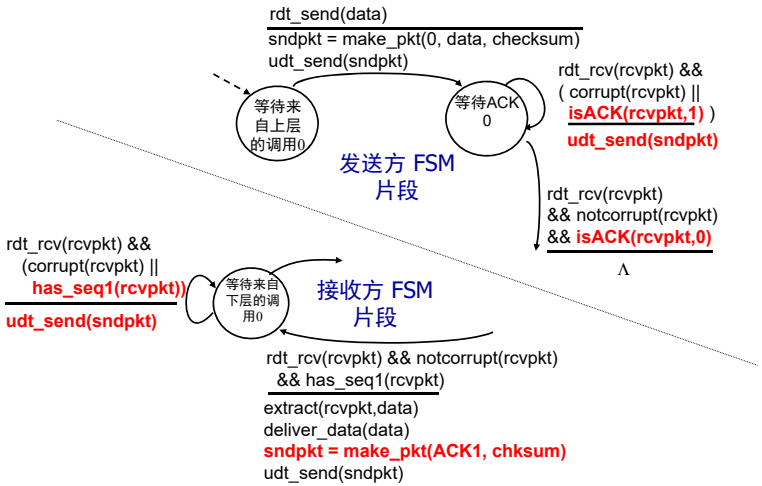
rdt2.2: 一种无NAK的协议

- 与rdt2.1一样的功能，仅使用ACK
- 代替NAK，接收方对最后正确接收的分组发送ACK
  - 接收方必须明确地包括被确认分组的序号
- 发送方接收冗余的ACK导致如同NAK相同的动作:知道接收方没有正确接收到跟在被确认两次的分组后面的分组，发送发需要*重传当前分组*

发送方收到冗余的ACK意味着收到NAK，重传当前的分组。

TCP也是利用这种仅使用ACK的方式来实现无NAK

rdt2.2: 发送方, 接收方片段



rdt3.0: 具有比特差错和丢包的信道

新的假设: *底层*信道也能丢失分组（数据或ACK）

- 检验和、序号、重传将是有帮助的，但还不够

问题: 人们如何处理会话中丢失的语句？

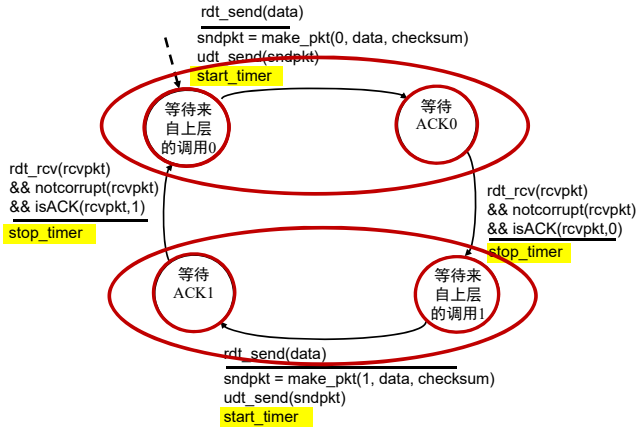
rdt3.0:具有比特差错和丢包的信道

方法: 发送方等待ACK一段“合理的”时间

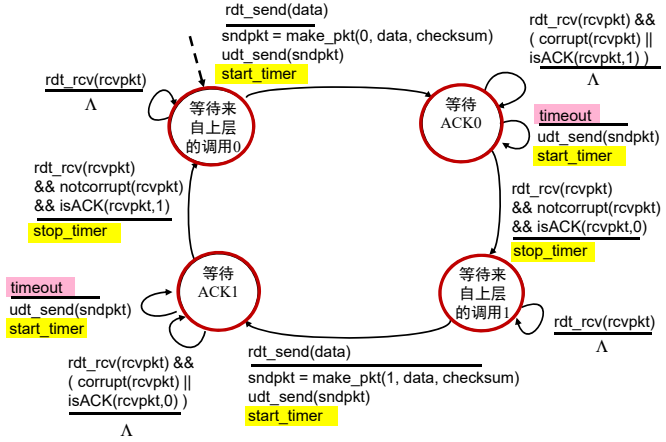
- 如在这段时间没有收到ACK则重传
- 如果分组（或ACK）只是延迟但没有丢失:
  - 重传将是冗余的，但序号的使用已经处理了该情况
  - 接收方必须指出被确认的分组序号
- 在“合理的”时间过后使用倒计数定时器来中断

超时

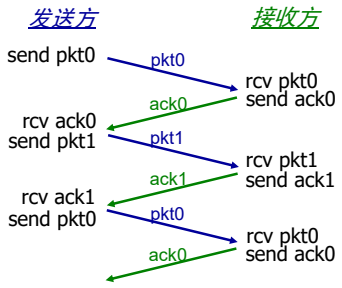
rdt3.0 发送方



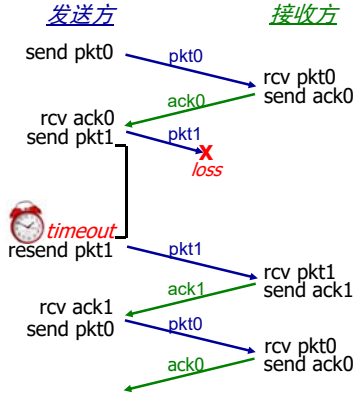
rdt3.0 发送方



rdt3.0 运行情况

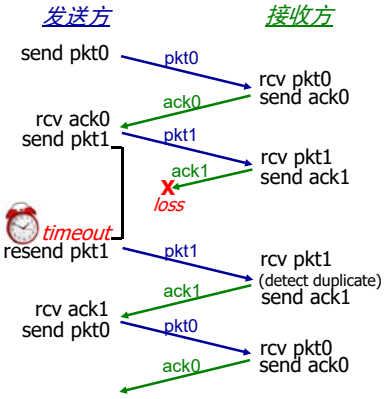


(a) 无分组丢失

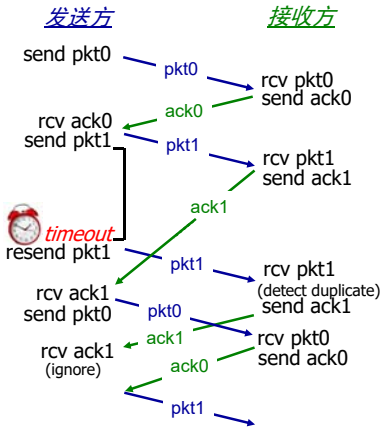


(b) 分组丢失

rdt3.0 运行情况



(c) ACK 丢失



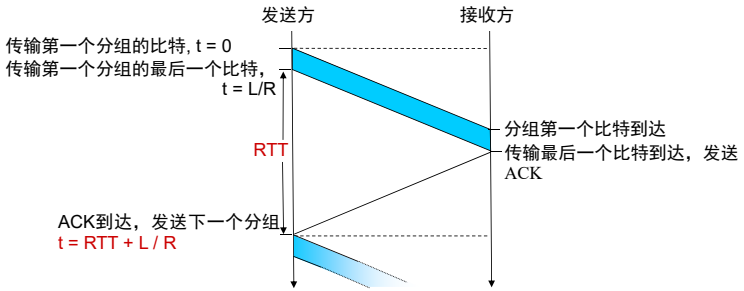
(d) 超时前置于ACK/延迟的 ACK

rdt3.0的性能 (停等协议)

- $U_{发送方}$: 发送方（或信道）的*利用率utilization* – 发送方实际忙于将发送比特送进信道的那部分时间与发送时间之比（ fraction of time sender busy sending ）
- 例子: 1 Gbps链路, 15ms端到端传播时延, 包括首部字段和数据的分组长度为8000 比特，即1000字节的分组
  - 将分组传递到信道的时间，即传输时延（推出分组所需要的时间）：

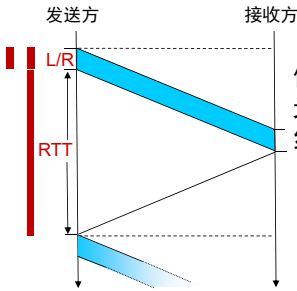
$$D_{trans} = \frac{L}{R} = \frac{8000\ bits}{10^9\ bits/sec} = 8\ microsecs$$

rdt3.0: 停等协议的运行



rdt3.0: 停等协议的运行

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

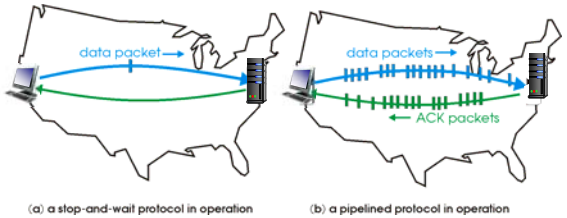


假设两个端系统之间的光速往返传播时延RTT大约为30ms

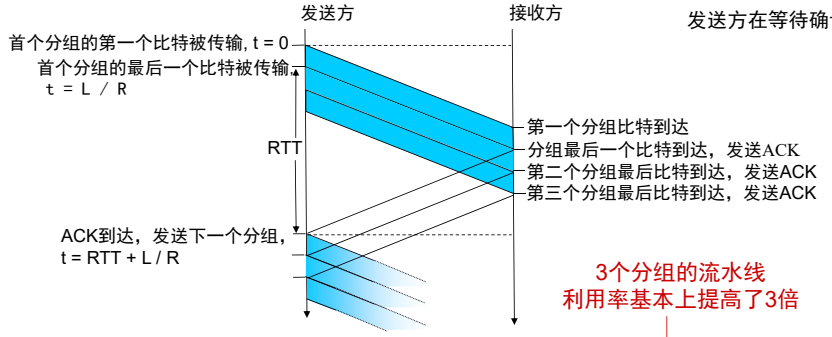
- 有效的吞吐量为267kbps，即使1Gbps的链路
- rdt 3.0 能够工作，但性能不好!
- 停等协议限制了底层基础设施（信道）的性能

rdt3.0: 流水线协议的运行

流水线: 发送方允许发送多个、“传输中的”、还没有应答的分组
- 序号的范围必须增加
- 发送方和/或接收方设有缓冲



流水线: 增加的利用率

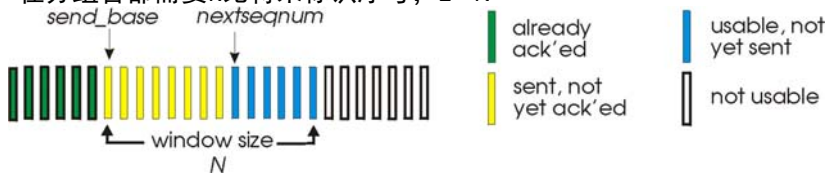


3个分组的流水线
利用率基本上提高了3倍

$$U_{sender} = \frac{3L/R}{RTT + L/R} = \frac{.0024}{30.008} = 0.00081$$

回退N步（ Go-Back-N ， GBN）协议：发送方

- 也叫作滑动窗口协议
- 发送方: “窗口”最大为N，允许发送N个连续的没有应答的分组
  - 在分组首部需要K比特来标识序号，2^k=N

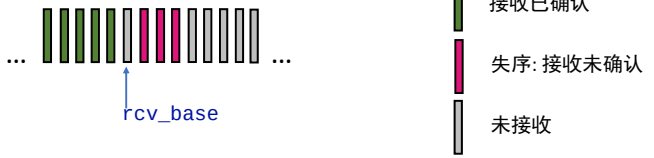


- *累积确认:* ACK(*n*)表明接收方已正确接收到序号为n的以前且包括n在内的所有分组
  - ACK(n)后，向前移动窗口到n+1位置
- 最早的已发送但未被确认的分组使用定时器
- *超时(n):* 若出现超时，重传窗口中的分组n及所有更高序号的分组

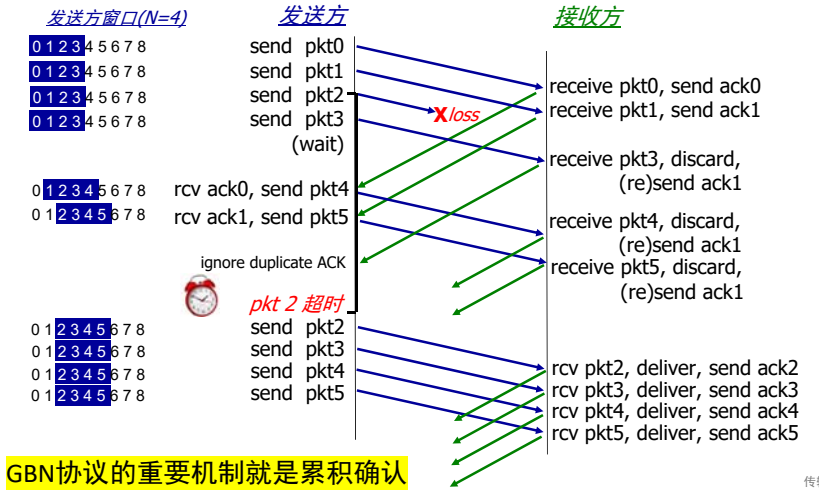
回退N步（Go-Back-N ， GBN）协议: 接收方

- 收到按序的正确分组，为分组发送确认
  - 如果一个序号为n的分组被正确接收到， 并且按序（即上次交付给上层的数据是序号为n-1的分组），则接收方为分组n发送一个ACK,并将该分组中的数据部分交付到上层
  - 可能会生成重复的ACK
  - 仅需要记住`rcv_base`
- 收到失序的分组:
  - 丢弃所有失序分组
  - 重新确认最高序号的分组

接收方看到的序号:



回退N步（Go-Back-N，GBN）协议示意



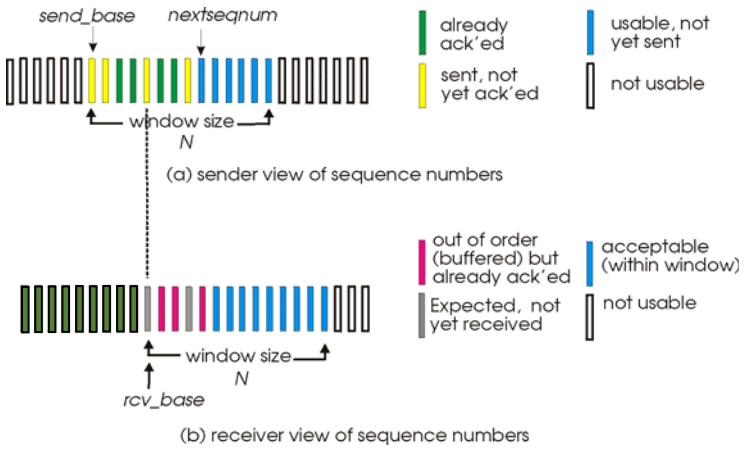
GBN协议的重要机制就是累积确认

选择重传（Selective Repeat，SR）协议

- 接收方单独确认每一个正确接收的分组
  - 需要缓存分组，以便最后按序交付给上层
- 发送方只需要重传没有收到ACK的分组
  - 发送方对每个没有收到确认的分组开启计时器
- 发送窗口
  - *N* 个连续的序号
  - 该窗口也限制了已发送但尚未确认的分组数量

SR协议的核心机制是接收方单独确认每一个接收的分组

选择重传: 发送方, 接收方窗口



选择重传

发送方

上层传来数据:

- 如果窗口中下一个序号可用，发送分组

timeout(*n*):

- 重传分组n，重启其计时器

ACK(*n*) n在 [sendbase,sendbase+N]:

- 标记分组n已经收到
- 如果n是最小未收到确认的分组，则向前滑动窗口的base指针到下一个未确认的序号

接收方

分组n在[rcvbase, rcvbase+N-1]

- 发送ACK(*n*)
- 失序：缓存
- 按序：交付（同时也交付所有缓存的按序分组），向前滑动窗口到下一个未收到的分组的序号

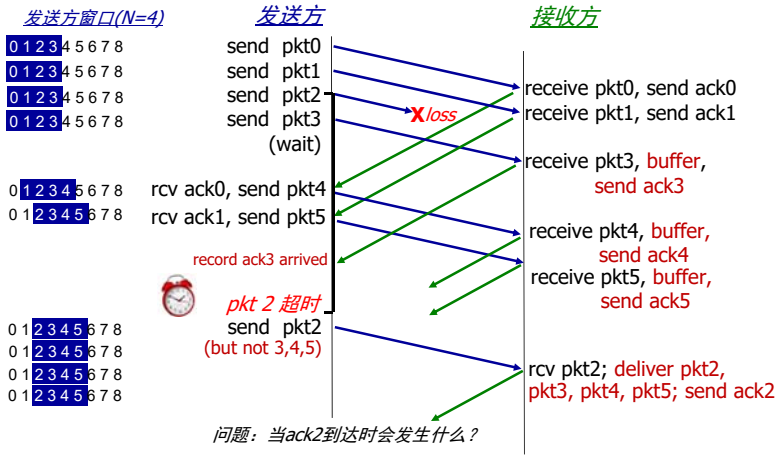
分组n在[rcvbase-N,rcvbase-1]

- ACK(*n*)

其他:

- 忽略

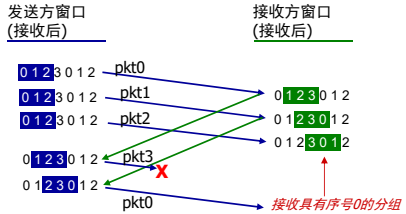
选择重传：示意



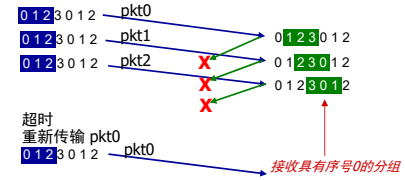
选择重传的窘境

例子:

- 序号: 0, 1, 2, 3
- 窗口大小 = 3



(a) 没有问题



(b) 糟糕!

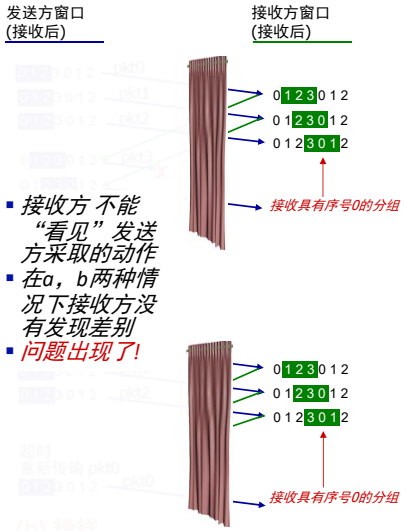
选择重传的窘境

例子:

- 序号: 0, 1, 2, 3
- 窗口大小 = 3

问题: 为了避免情况b的情况，序号长度和窗口大小应该有什么关系?

设：序号长度为k，即序列号为[0,2^k-1]
窗口长度为N



- *接收方 不能“看见”发送方采取的动作*
- *在a，b两种情况下接收方没有发现差别*
- *问题出现了!*

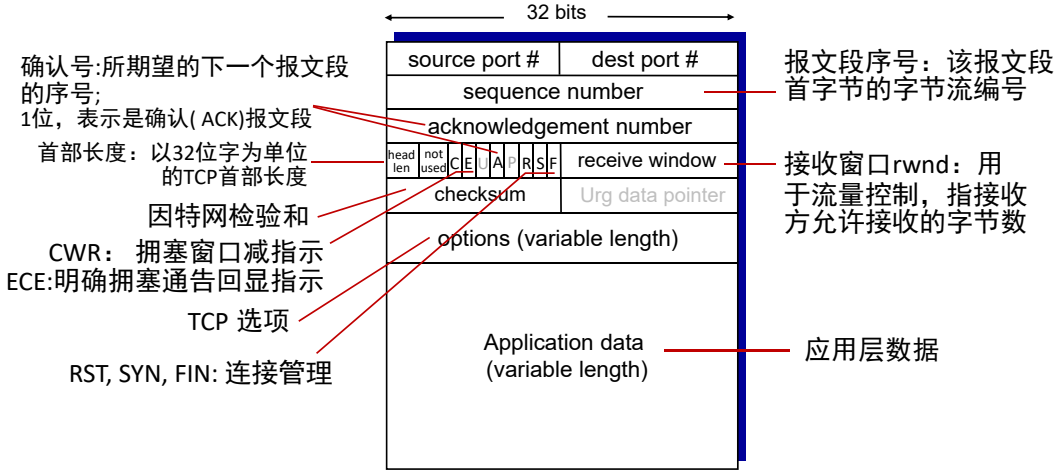
第三章：内容

- 传输层服务
- 多路复用和多路分解
- 无连接传输：UDP
- 可靠数据传输原理
- **面向连接的传输：TCP**
- 拥塞控制原理
- TCP拥塞控制

TCP: 概述 RFCs: 793,1122, 2018, 5681, 7323

- 点到点:
  - 单个发送方, 单个接收方
- 可靠，有序的字节流:
  - 逻辑连接，*“无缝对接”*
- 全双工服务:
  - 同一连接上的双向数据流
  - MSS: 最大报文段长度 maximum segment size
    MSS典型长度是1460字节
    MSS通常根据MTU来设置
    MSS（1460）+TCP/IP首部长度(通常是40字节）≤MTU（以太网链路1500字节）
- 累积确认
- 流水线:
  - TCP拥塞和流量控制设置窗口大小
- 面向连接的:
  - 在进行数据交换前，进行握手（交换控制信息）、初始化发送发与接收方的状态
- 流量控制:
  - 发送方不能淹没接收方

TCP 报文段（Segment）结构



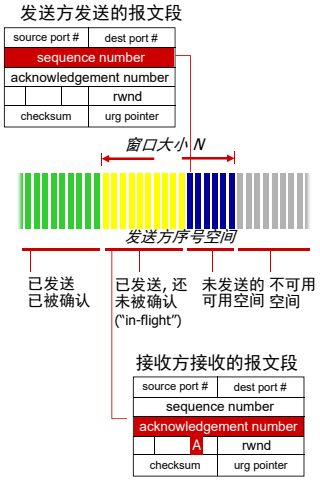
TCP 序号和确认号

序号:
- 报文段中第一个数据字节在字节流中的位置编号

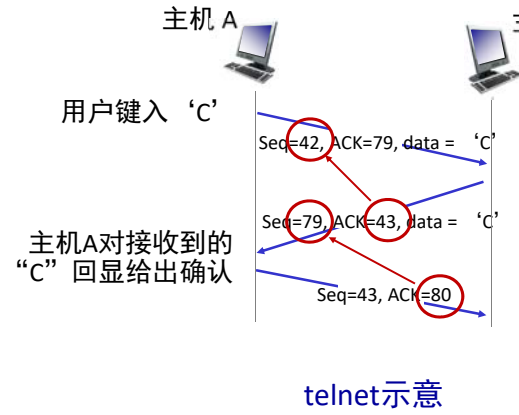
确认号:
- 期望从对方收到下一个字节的序号
- 累积确认

问题: 接收方如何处理失序报文

回答: TCP规范没有说明，由实现者自行选择实现：抛弃/缓存



TCP 序号和 确认号



telnet示意

TCP的序号是该报文段首字节的字节流编号。
- TCP将应用层数据看成是有序的字节流；
- 隐式地对数据流中的每一个字节编号

TCP的确认号：主机A填充在报文段的确认号是主机A期望从主机B收到的下一字节的序号-正在等待的数据的下一个字节序号。

TCP只确认流中至第一个丢失字节为止的字节---即累积确认

TCP往返时间（RTT）估计与超时

问题: 如何设置TCP超时值?

- 应大于RTT, 但是RTT是变化的!
- *太短:* 过早超时，导致不必要的重传
- *太长:* 对报文段的丢失响应太慢

问题: 如何估计RTT?

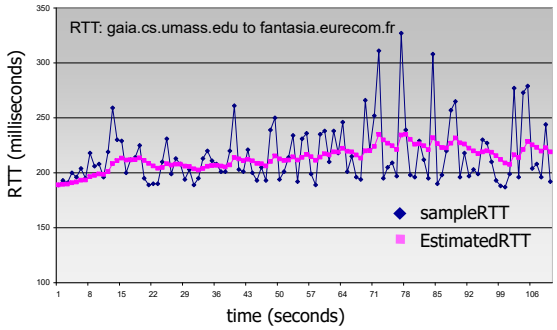
- SampleRTT: 从报文段发出到接收到确认的时间进行测量
  - 仅在某个时刻做一次SampleRTT测量
  - 绝不为已被重传的数据段估计SampleRTT
- SampleRTT 会变化，希望估计的RTT “较平滑”
  - 使用最近测量值的平均，并不是当前的SampleRTT

TCP往返时间（RTT）估计与超时

使用SampleRTT均值： EstimatedRTT

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

- 指数加权移动平均 exponential weighted moving average (EWMA)
- 过去的样本指数级衰减来产生影响
- 典型值: α = 0.125



TCP往返时间（RTT）估计与超时

- 超时间隔: **EstimatedRTT** 加 “安全余量”
  - **EstimatedRTT大变化： 更大的安全余量**

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

estimated RTT “安全余量”

- **DevRTT**: 计算**SampleRTT的EWMA和EstimatedRTT之间的差值**:

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

(typically, β = 0.25)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

TCP 发送方 (simplified)

事件: 从上面应用程序接收到数据

- 生成具有序列号的报文段
- 序号是报文段中第一个数据字节的数据流编号
- 如果定时器当前没有运行，启动定时器
  - 将定时器想象为与最早的未被确认的报文段相关联
  - 超时间隔: **TimeOutInterval**

事件: 定时器超时

- 重传发生超时的报文段
- 计时器重启

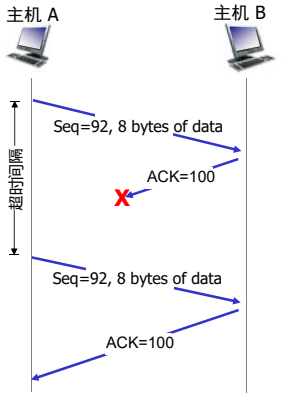
事件: 收到ACK

- 如果ACK是确认先前未被确认的报文段
  - 更新被确认的报文段序号
  - 如果当前没有收到任何确认报文段，重启定时器

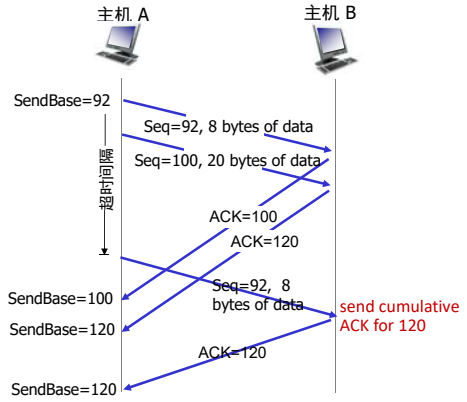
TCP 接收方: ACK 产生 [RFC 5681]

接收方事件	*TCP 接收方行为*
具有所期望序号的按序报文段到达。所有在期望序号以前的数据都已经被确认	延迟确认。等待最多500ms以接收下一个报文段。如果下一个报文段在这个时间间隔内没有到达， 则发送确认
具有所期望序号的按序报文段到达，且前一个已到达的报文段正在等待发送确认	立即发送单个累积确认,以确认收到两个按序报文段
比期望序号大的失序报文段到达。检测出收到的报文段序号间的间隔	立即发送冗余确认,指示下一个期待字节的序号（其为间隔的低端的序号）
能部分或完全填充接收所接收到的间隔的报文段到达	倘若该报文段起始于间隔的低端， 则立即发送确认

TCP: 重传情况



由于确认丢失而重传

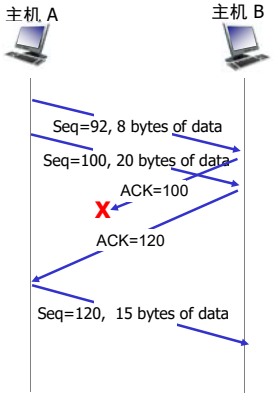


报文段100没有重传，为什么？

当接收到该重传段时接收方已经收到前两个段，因此，对于到目前为止收到的两个段重新发送累积ACK，而不是仅对第一个段重新发出ACK。

TCP: 重传情况

累积确认：避免了第一个报文段的重传



超时时间: 超时间隔加倍避免网络拥塞

- TCP重传时会将下一次的超时时间设为先前值的两倍
- 收到上层应用的数据或者收到ACK后，超时计时器由最近的EstimatedRTT和DevRTT值推算得到

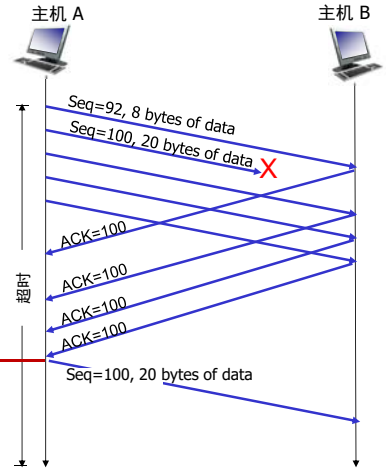
TCP 快速重传

TCP 快速重传

如果TCP发送方接收到对相同数据的3个冗余ACK, 重新发送最小序号的未被确认的报文段

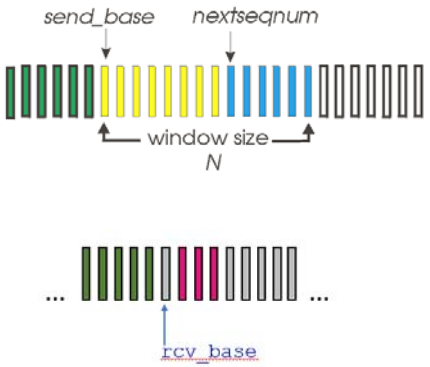
- 相当于未被确认的报文段丢失，所以不需要等待超时
  - TCP能够更快地从很可能发生的丢失事件中恢复

如果TCP发送方接收到对相同数据的3个冗余ACK,它把这当作一种指示，说明跟在这个已被确认过3次的报文段之后的报文段已经丢失

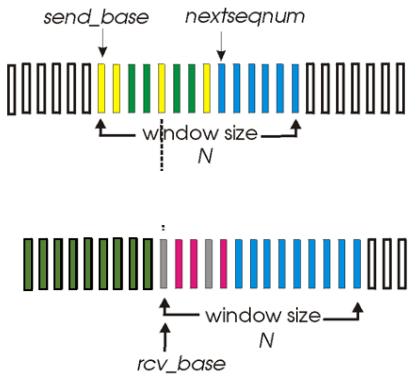


TCP采用GBN还是SR

- GBN



- SR

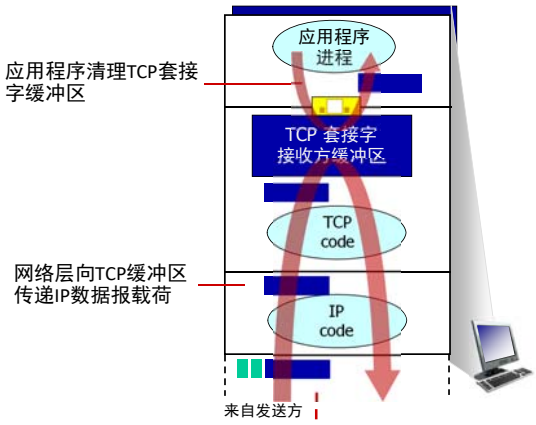


第三章：内容

- 传输层服务
- 多路复用和多路分解
- 无连接传输：UDP
- 可靠数据传输原理
- **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**
- 拥塞控制原理
- TCP拥塞控制

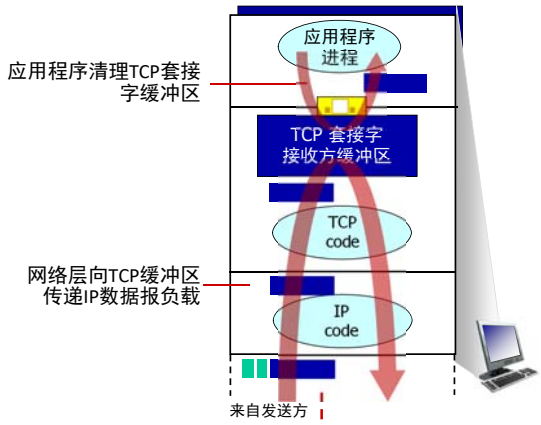
TCP 流量控制

问题: 如果网络层传送数据的速率比应用层清理套接字缓冲速度更快会发生什么？



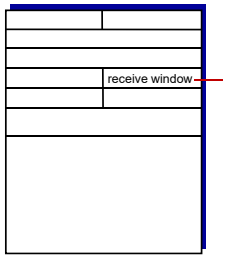
TCP 流量控制

问题: 如果网络层传送数据的速率比应用层清理套接字缓冲速度更快会发生什么？

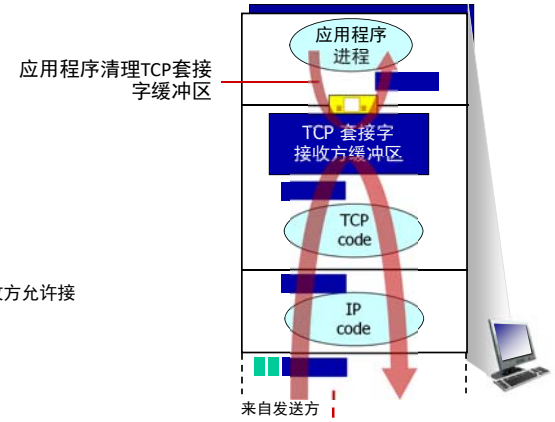


TCP 流量控制

问题: 如果网络层传送数据的速率比应用层清理套接字缓冲速度更快会发生什么？



流量控制: 接收方允许接收的字节数



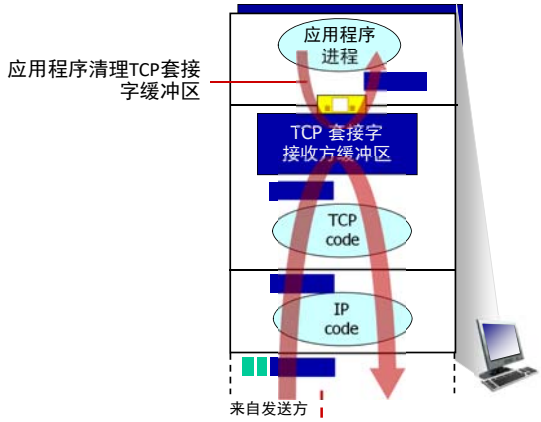
接收方协议栈

TCP 流量控制

问题: 如果网络层传送数据的速率比应用层清理套接字缓冲速度更快会发生什么？

流量控制

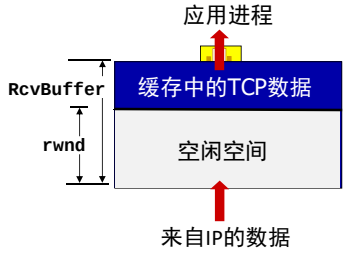
发送方不能发送太多、太快的数据让接收方缓冲区溢出。



接收方协议栈

TCP 流量控制

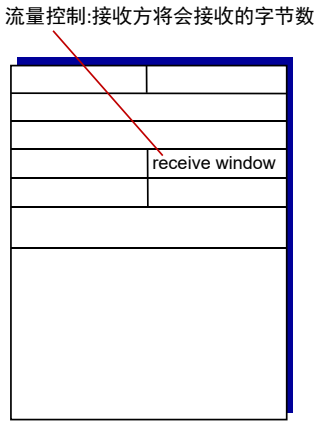
- 接收方在报文段接收窗口字段（rwnd, receive window）中“通知”其接收缓冲区的剩余空间
  - **RcvBuffer 根据套接字选项确定大小（通常的默认值为4096字节）**
  - 许多操作系统自动适应 **RcvBuffer**
- 发送方 要限制未确认的数据不超过接收窗口（**rwnd**）
- 保证接收缓冲区不溢出



接收窗口（rwnd）和接收缓存（RcvBuffer）

TCP 流量控制

- 接收方在报文段接收窗口字段（rwnd）中“通告”其接收缓冲区的剩余空间
  - **RcvBuffer 根据套接字选项确定大小（通常的默认值为4096字节）**
  - 许多操作系统自动适应 **RcvBuffer**
- 发送方 要限制未确认的数据不超过接收窗口（**rwnd**）
- 保证接收缓冲区不溢出

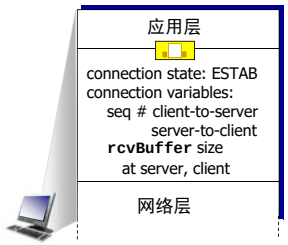


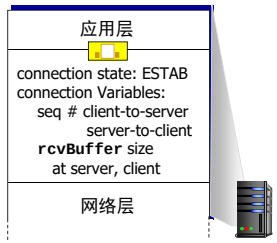
TCP报文段格式

TCP 连接管理

在交换数据前, 发送方/接收方的“握手”:

- 同意建立连接（彼此知道对方想要建立连接）
- 确认连接参数（比如开始序号）



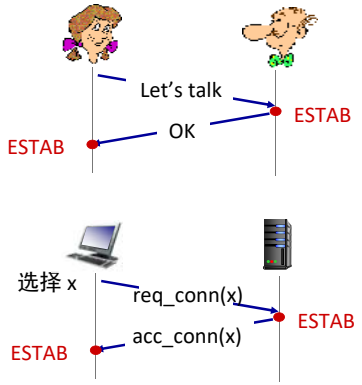



```
Socket ClientSocket =
  newSocket("hostname","port number");
```

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


同意建立连接

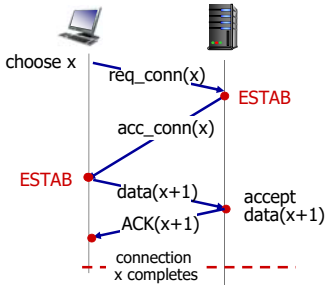
2次握手:



问题: 2次握手在网络层总是有效吗?

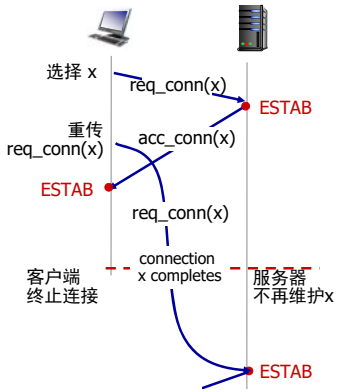
- 多变的延迟
- 由于报文丢失导致重传
- 报文重新排序
- 彼此“看不见”

2次握手情况



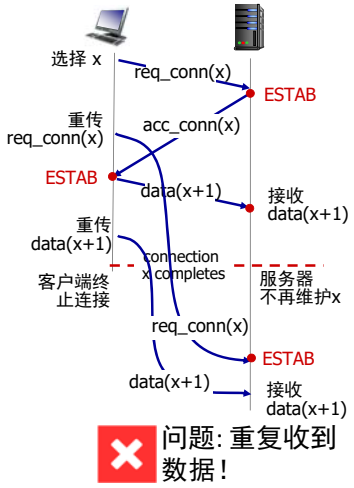
没问题!

2次握手情况



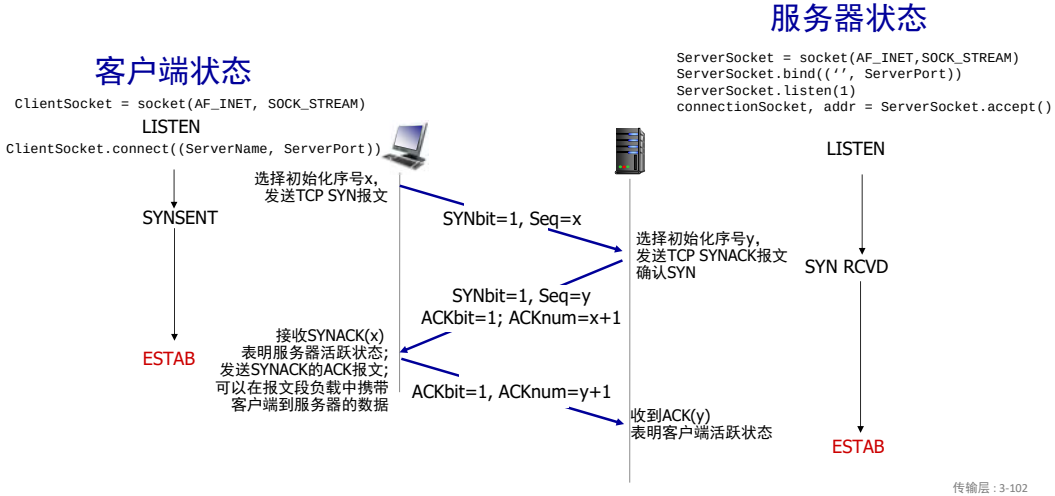

问题: 半打开连接! (没有客户端)

2次握手情况



问题: 重复收到数据！

TCP 3次握手

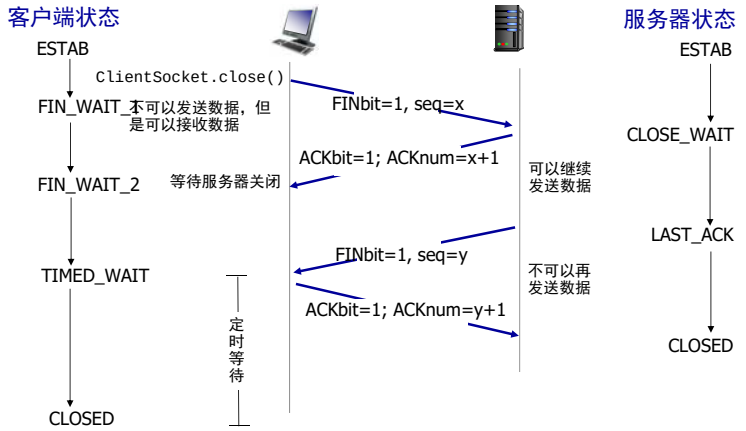


人类的3次握手协议

关闭TCP连接

- 参与一条TCP连接的两个进程中的任何一个都能终止该连接
  - 发送FIN位为1的TCP报文段
- 用ACK响应接收到的FIN
  - 收到FIN时，ACK可与FIN同时发送
- 对收到的FIN进行确认后，等待一段时间（30秒、1分或者2分），连接就正式关闭了，所有的资源将被释放

关闭TCP连接 - 4次挥手



第三章：内容

- 传输层服务
- 多路复用和多路分解
- 无连接传输：UDP
- 可靠数据传输原理
- 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
- 拥塞控制原理
- TCP拥塞控制

拥塞控制原理

拥塞:

- 非正式地：“太多的源发送太多的数据，使网络来不及处理”
- 表现:
  - 长时延（路由器缓冲区中排队）
  - 丢包（路由器缓冲区溢出）

拥塞控制:
太多的发送源；发送得太快

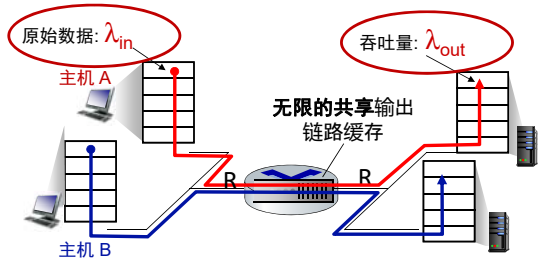
- 不同于流量控制!
- 是组网技术中前10个基础性重要问题之一!

流量控制: 发送方给接收方发送的数据太快。

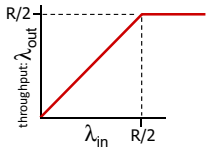
拥塞的原因与代价: 情况1

最简单的情况:

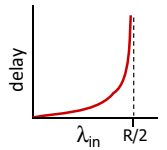
- 一个路由器，无限缓冲区
- 输入、输出链路能力: R
- 两个流，主机A与B分别向路由器提供流量的速率是 λ_{in}
- 不需要重传



问题: 当发送速率 λ_{in} 接近R/2时，会发生什么?



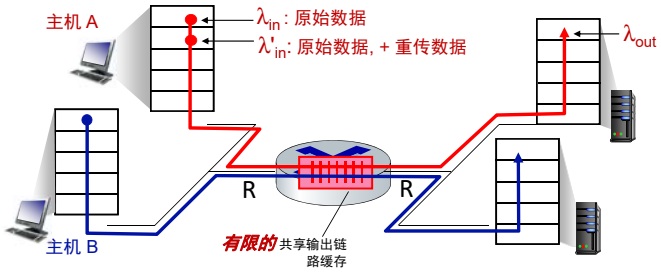
每连接的最大吞吐量（maximum per-connection throughput）: R/2



当发送速率λ_{in}接近R/2时，源和目的之间的平均时延就会越来越大

拥塞的原因与代价: 情况2

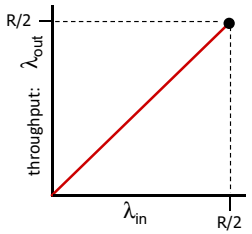
- 一个路由器，*有限*缓冲区
- 发送方重传丢失的超时的数据分组
  - 应用层输入 = 应用层输出: $\lambda_{in} = \lambda_{out}$
  - 传输层输入包括重传: $\lambda'_{in} \geq \lambda_{in}$

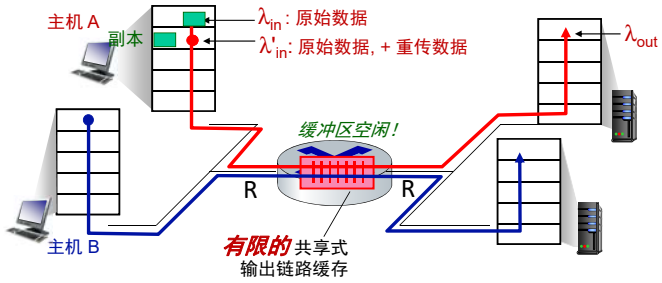


拥塞的原因与代价: 情况2

完美情况

- 发送方仅在路由器缓冲区空闲时发送一个分组

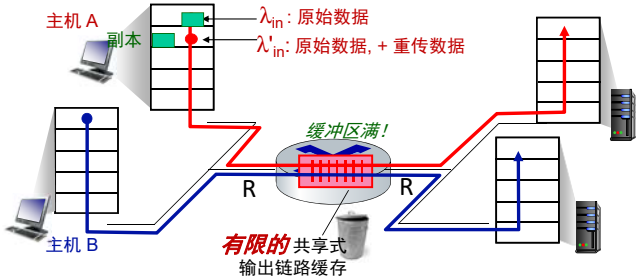




拥塞的原因与代价: 情况2

部分完美情况

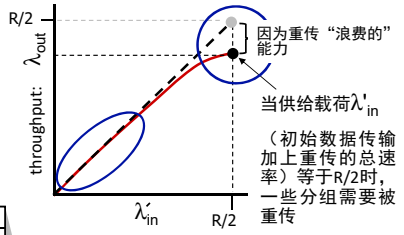
- 分组可能因为缓冲区溢出而被丢包（路由器丢弃）
- 发送方必须执行重传以补偿因为缓存溢出而丢弃（丢失）的分组

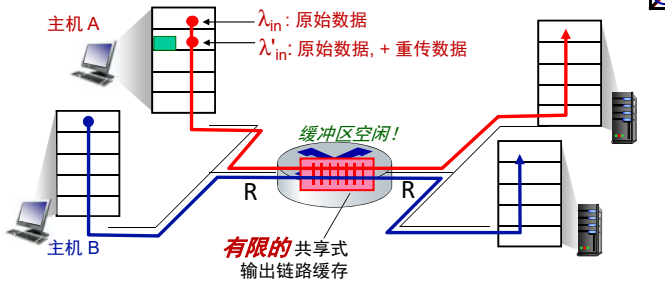


拥塞的原因与代价: 情况2

部分完美情况

- 分组可以因为缓冲区溢出而被丢包（路由器丢弃）
- 发送方必须执行重传以补偿因为缓存溢出而丢弃（丢失）的分组

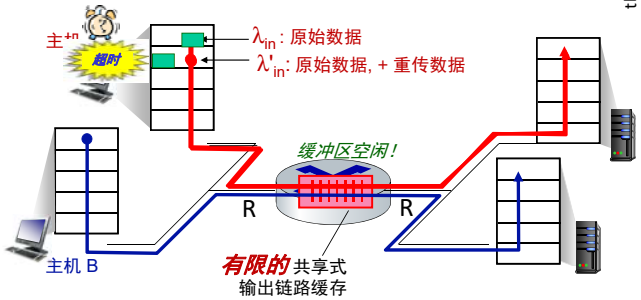


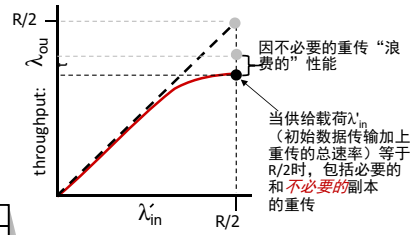


拥塞的原因与代价: 情况2

真实情况: *不必要的重传*

- 分组可以因为缓冲区溢出而被丢包（路由器丢弃）– 需要重传
- 发送方提前发生超时并重传队列中已被推迟但还未丢失的分组,则初始分组和重传分组都到达接收方,发送了*两份*副本

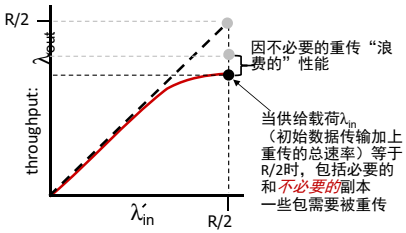




拥塞的原因与代价: 情况2

真实情况: *不必要的重传*

- 分组可以因为缓冲区溢出而被丢包（路由器丢弃）– 需要重传
- 发送方提前发生超时并重传队列中已被推迟但还未丢失的分组,初始分组和重传分组都到达接收方,发送了*两份*副本



拥塞的“代价”：

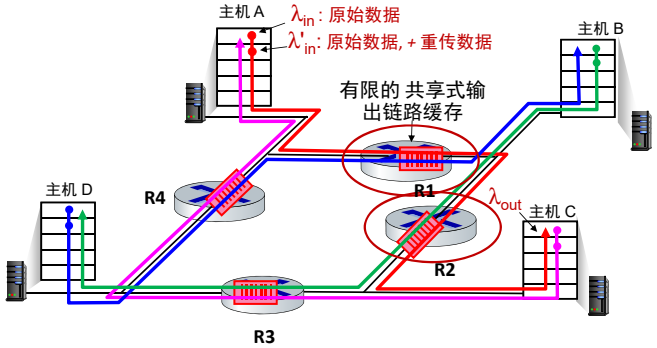
- 比额定的“吞吐量”做更多的工作
- 不必要的重传：链路承载分组的多个拷贝
  - 降低了最大可获得的吞吐量

拥塞的原因与代价: 情况3

- 四个发送方
- 多跳路径
- 超时/重传

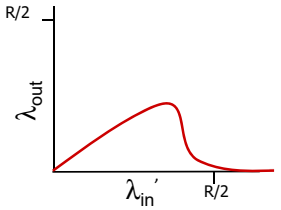
问题: 随着λ_{in}和λ_{in}'的增加将发生什么情况? 比如主机A到B的路径

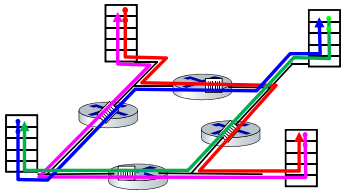
回答: 当红色 λ_{in}' 增加，所有正在到达的上层队列的蓝色分组将被丢弃，蓝色吞吐量→0



R2会发生变化？红色线路的分组到达C的情况？

拥塞的原因与代价: 情况3



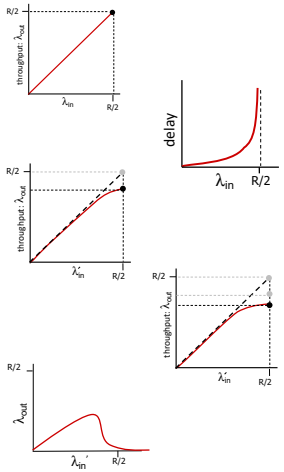


另一个拥塞的“代价”：

- 在大流量情况下，每当有一个分组在第二跳路由器上被丢弃时，第一跳路由器所做的将分组转发到第二跳路由器的工作就是徒劳无功
- 当一个分组沿一条路径被丢弃时，每个上游路由器用于转发该分组到丢弃该分组而使用的传输容量最终被浪费！

拥塞的原因与代价: 感悟

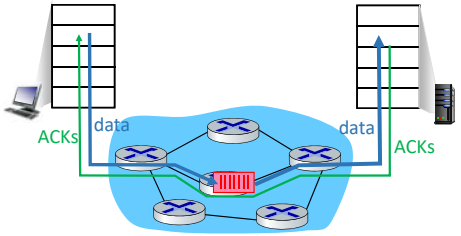
- 吞吐量永远不能超过传输能力
- 延迟随着传输速度接近传输能力而增加
- 丢失/重传降低有效吞吐量
- 不必要的重传进一步降低了有效吞吐量
- 上游传输能力/缓冲区被下游丢包所浪费



拥塞控制方法

端到端的拥塞控制:

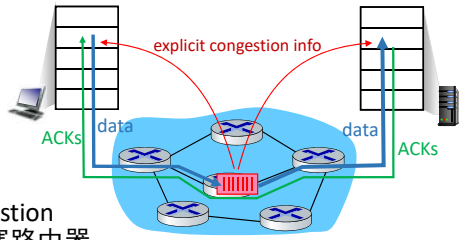
- 不能从网络得到明确的反馈
- 根据观察到的时延和丢包现象*推断*出拥塞
- 这是TCP所采用的方法



拥塞控制方法

网络辅助的拥塞控制:

- 路由器向发送/接收主机*直接*反馈
- 可以指示拥塞的程度或者显式地设置发送速率
  - IP首部TOS字段的 ECN（Explicit Congestion Notification,明确拥塞通告）， 在拥塞路由器的IP数据报首部设置ECN比特，送给目的主机，再由目的主机通知发送主机；TCP首部的ECE、CWR配合使用
  - ATM可用比特率拥塞控制, DECbit 协议



第三章：内容

- 传输层服务
- 多路复用和多路分解
- 无连接传输：UDP
- 可靠数据传输原理
- 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
- 拥塞控制原理
- TCP拥塞控制

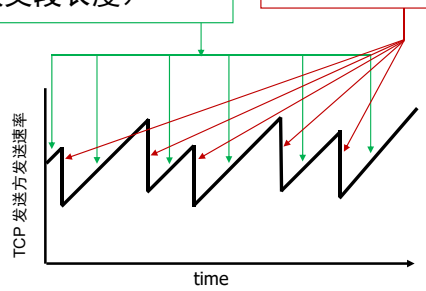
TCP 拥塞控制: AIMD, 加性增,乘性减
Additive- Increase, Multiplicative- Decrease

- *方法:* 发送方可以提高发送速率，直到发生数据包丢失（拥塞），然后在丢失事件时降低发送速率

Additive Increase
如没有检测到丢包事件，每个RTT时间拥塞窗口值增加一个MSS（最大报文段长度）

❖

Multiplicative Decrease
丢包事件后，拥塞窗口值减半



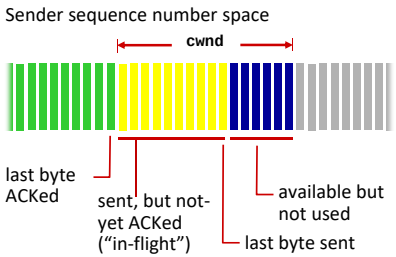
AIMD 锯齿状行为:
带宽*探测*

TCP AIMD: 扩展知识

乘性减 细节: 发送速率
- 出现3个冗余ACK事件时cwnd减半(TCP Reno)
- 当检测到超时事件时缩减到1个MSS (TCP Tahoe)

为什么是 AIMD?
- AIMD –一个分布式的异步算法-已经被证明:
  - 实现了全网范围内的拥塞流量优化!
  - 用户和网络性能等几个重要方面都被同时优化

TCP 拥塞控制: 细节



TCP 发送行为:
- 发送cwnd 字节,在RTT时间内等待ACK，然后发送更多的字节

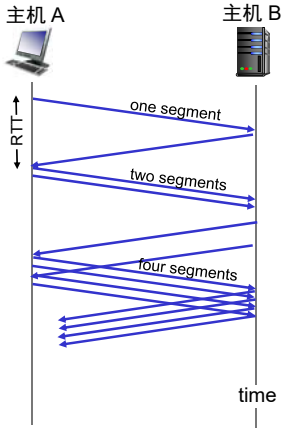
$$\text{TCP rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- TCP 发送方限制传输: `LastByteSent- LastByteAcked ≤ cwnd`
- cwnd（congestion window）拥塞窗口，是随拥塞状态动态变化的 (由TCP拥塞控制实现)

在这一节，假设TCP接收缓存足够大

TCP 慢启动（ slow start ）

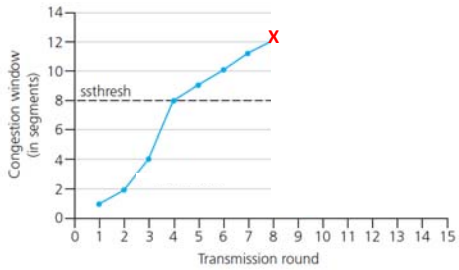
- 当连接开始时，以指数级增加速率，直到第一个丢包事件发生:
  - 初始化 **cwnd** = 1 MSS
  - 每收到1个ACK，增加1个拥塞窗口
  - 每个RTT时间后翻倍 **cwnd**
- *总结:* 初始速率很低，但以指数速度增加



TCP: 从慢启动到拥塞避免(congestion avoidance)

问题: 什么时候从指数增长变为线性增长?

回答: 超时指示的丢包事件发生时，将ssthresh设置为cwnd的1/2；当cwnd到达或超过ssthresh时，从慢启动进入拥塞避免

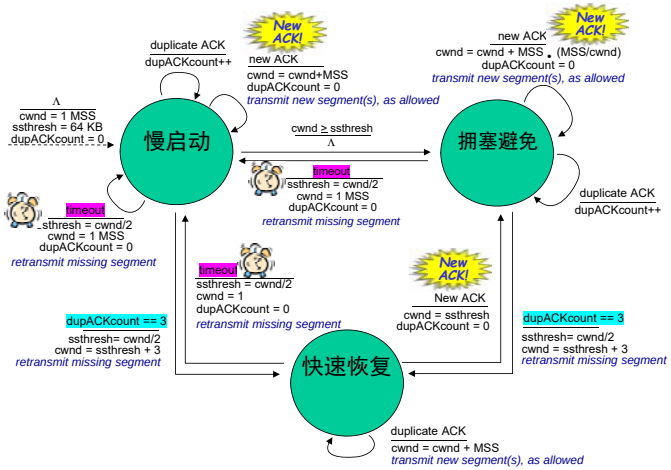


实现方法:

- 阈值变量 **ssthresh（慢启动阈值）**
- 在丢包事件发生时，阈值 **ssthresh** 设置为发生丢包时的 **cwnd** 的一半

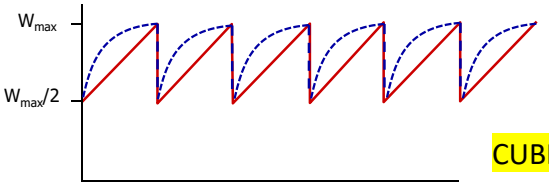
* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

总结: TCP 拥塞控制



TCP CUBIC

- 有没有比AIMD更好的方法来“探测”可用带宽?
- 主要思想:
  - W_{max}:检测到拥塞丢包时的发送速率
  - 瓶颈链路的拥塞状态可能 (?)还没有大幅度改善，丢包较多
  - 在减半速率/窗口后，最初以更快的速度爬升到W_{max}，但随后以更慢的速度接近W_{max}

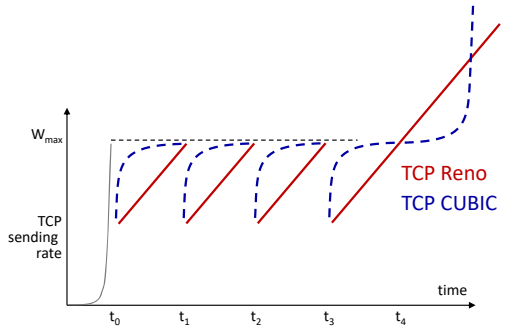


CUBIC是基于丢包做的拥塞控制

TCP CUBIC

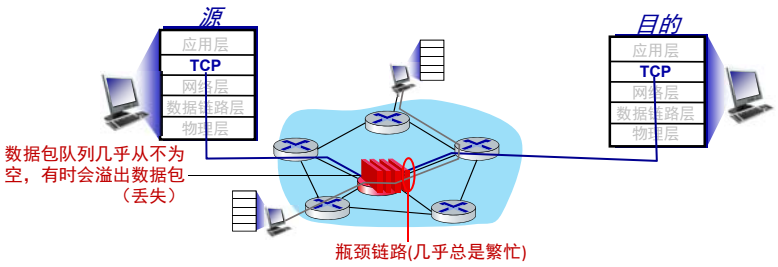
- K: TCP窗口大小将达到W_{max}的时间点
  - K本身是可调的
- 增加W作为当前时间和K之间距离的立方的函数
  - 离K越远，增幅越大
  - 当接近K时，小幅增加

- Linux中TCP CUBIC是默认的，是最流行的用于流行Web服务器的TCP服务。



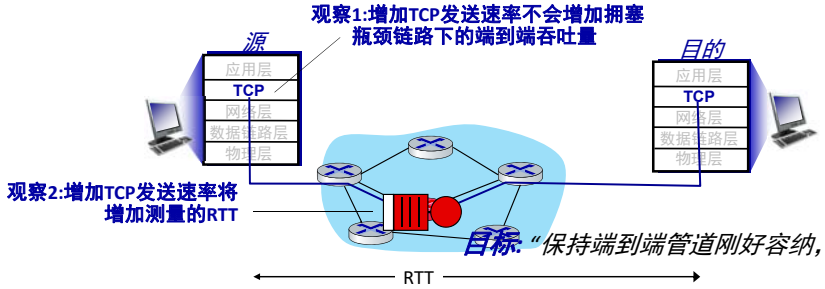
TCP 和拥塞的“瓶颈链路”

- TCP (经典, CUBIC)增加TCP的发送速率，直到某个路由器的输出发生数据包丢失: *瓶颈链路*



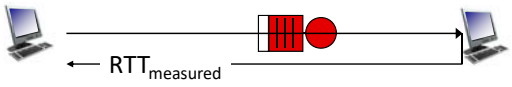
TCP 和拥塞的“瓶颈链路”

- TCP (经典, CUBIC)增加TCP的发送速率，直到某个路由器的输出发生数据包丢失: *瓶颈链路*
- 理解拥塞:有助于关注拥塞的瓶颈链路



基于延迟的TCP拥塞控制

保持发送方到接收方管道“刚好足够，但没有更满”：保持瓶颈链路忙于传输，但避免高延迟



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measured}}$$

基于延迟的方法：

- RTT_{min}-观测到的最小RTT（未阻塞路径）
- 拥塞窗口的未阻塞吞吐量cwnd为cwnd/RTT_{min}

```
if measured throughput “very close” to  uncongested throughput
      increase cwnd linearly            /* since path not congested */
else if measured throughput “far below” uncongested throughout
      decrease cwnd  linearly           /* since path is congested */
```


基于延迟的TCP拥塞控制

- 无丢包下的拥塞控制
- 最大化吞吐量（保持管道正好满…）时保持低延迟（“但不是更满”）

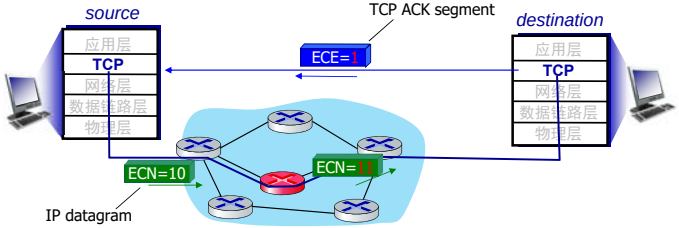
- 许多已部署的tcp采用基于延迟的拥塞控制
  - Google BBR - 谷歌公司提出的一个开源TCP拥塞控制的算法，Linux4.19内核中已经将拥塞控制算法从CUBIC（该算法从2.6.19内核开始引入到Linux）改为BBR

网络辅助拥塞控制

显式拥塞通知 Explicit congestion notification (ECN)

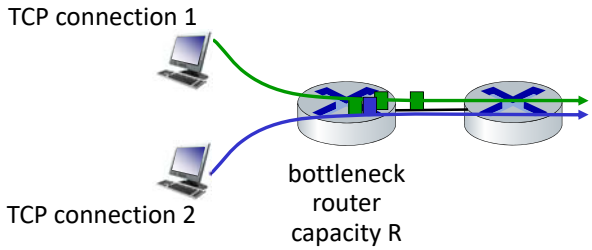
已部署的TCP通常都实现了网络辅助拥塞控制：

- 网络路由器标记的IP头（ToS字段）中的两位表示拥塞
- 目的地在ACK段上设置ECE位，以通知发送方拥塞
- 同时涉及IP（IP头ECN位标记）和TCP（TCP头CWR、ECE位标记）
  - IP的ECN位：IP首部的TOS字段中的第7和8位，00代表该报文并不支持ECN，01和10对路由器而言表明支持ECN功能，11标识发生拥塞；参见RFC3168
  - TCP的CWR和ECE位：TCP接收端收到IP头中的ECN=11标记，并在回复ACK时将ECE bit置1, TCP发送端收到ECE bit置1的ACK报文时，需要将自己的发送速率减半，并在发送下一个报文时，将CWR bit置1



TCP 公平性

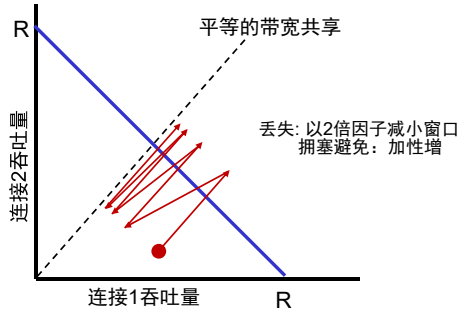
公平目标: 如果K个TCP会话共享带宽为R的瓶颈链路，每个会话应有R/K的平均链路速率



问题: 为什么TCP能保证公平性?

例子: 两个竞争会话:

- 随着吞吐量增加，按照斜率1加性增加
- 等比例地乘性降低吞吐量



TCP公平吗?

回答: 是的，在以下理想情况下是公平的

- 相同RTT
- 拥塞避免时有固定数量的会话个数

公平性（续）

公平性和UDP

- 多媒体应用通常不用TCP
  - 不希望拥塞控制遏制其传输速率
- 使用UDP:
  - UDP没有内置的拥塞控制
  - 音频/视频以恒定速率发送，能容忍报文丢失
  - 导致UDP流量压制TCP流量

公平性和并行TCP连接

- 并行TCP连接影响公平性
- WEB浏览器通常使用多个并行TCP连接来传送一个Web页中的多个对象
- 例子:支持9个连接的速率R的链路：
  - 某新应用若请求一个TCP连接，则得到R/10的带宽
  - 某新应用若请求11个TCP连接，则得到R/2的带宽

第3章：总结

- 传输层服务的原理:
  - 多路复用与多路分解
  - 可靠数据传输
    - ✓确认、定时器、重传、序号机制
  - 流量控制
  - 拥塞控制
- 因特网中的实例和实现
  - UDP: 无连接传输
  - TCP : 面向连接的可靠传输
  - TCP拥塞控制

第三章 作业

- 第8版
- 3、4、14、15、22、27、32、36、40、43

- 第7版
- 3、4、14、15、22、27、32、36、40、43