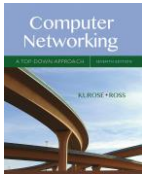
Chapter 3
传输层
Transport Layer



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

第三章：传输层

学习目标

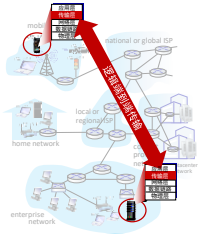
- 理解传输层服务原理:
  - 多路复用 multiplexing, 多路分解 demultiplexing
  - 可靠数据传输 reliable data transfer
  - 流量控制 flow control
  - 拥塞控制 congestion control
- 学习因特网中的传输层协议：
  - UDP: 无连接传输
  - TCP : 面向连接的可靠传输
  - TCP拥塞控制

第三章 要点

- **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: 运输 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

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 = 堂兄弟姐妹
- 应用层报文 = 信封上的字符
- 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

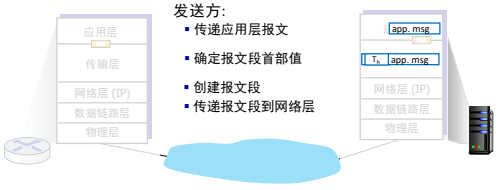
- 网络层: 主机（host）间的逻辑通信
- 传输层: 进程（process）间的逻辑通信
  - 依赖网络层服务

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 = 堂兄弟姐妹
- 应用层报文 = 信封上的字符
- 传输层协议 = Andy 和 Bill
- 网络层协议 = 邮政服务（包括邮车）

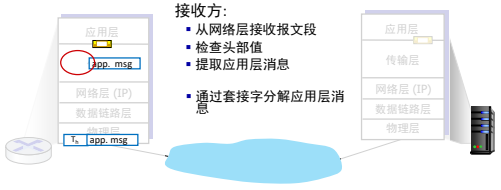
传输层行为



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

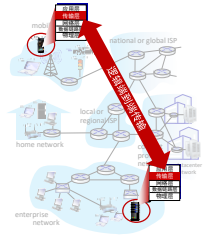


接收方:

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

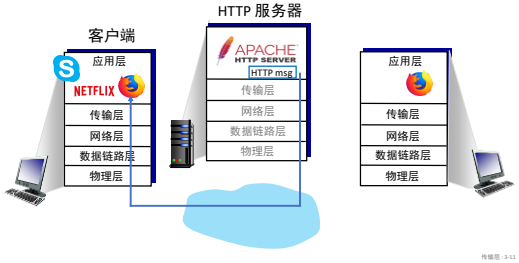
Internet 传输层协议

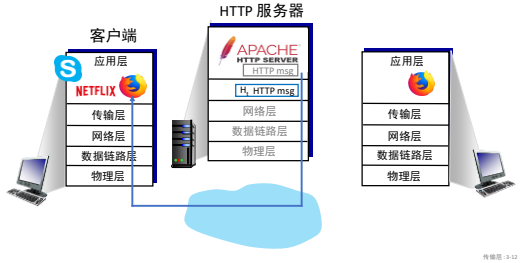
- TCP提供可靠数据交付、有序交付
  - 拥塞控制
  - 流量控制
  - 建立连接
- UDP提供不可靠交付、无序交付
  - 尽力而为的IP的简单扩展
- 不提供的服务:
  - 时延保证
  - 带宽保证

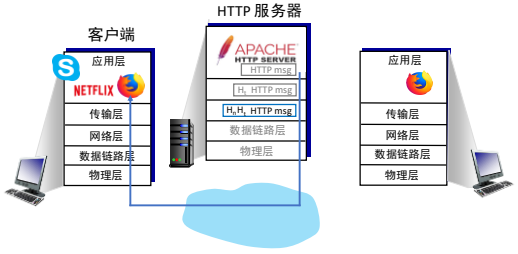


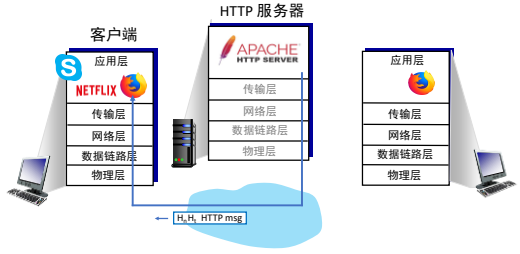
第三章：要点

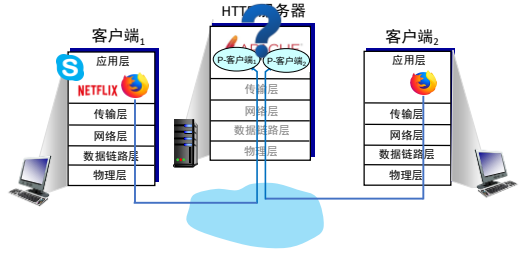
- 传输层服务
- **多路复用和多路分解**
- 无连接传输：UDP
- 可靠数据传输原理
- 面向连接的传输：TCP
- 拥塞控制原理
- TCP拥塞控制



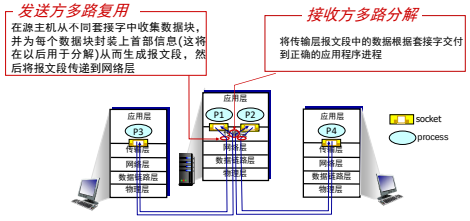






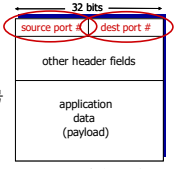


多路复用/多路分解
Multiplexing/demultiplexing



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，目的端口号
- 主机使用IP地址和端口号指引报文段找到特定的套接字



TCP/UDP 报文段格式

无连接的多路分解
Connectionless Demultiplexing

回顾:

- 创建包含端口号的套接字:

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

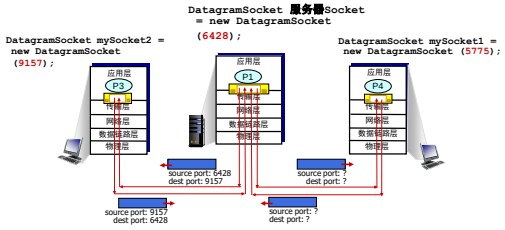
- 当创建UDP套接字时，必须明确如下二元组:
  - 目的IP地址
  - 目的端口号

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

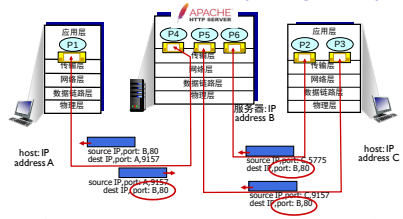
无连接多路分解的例子
Connectionless Demultiplexing: Example



面向连接的多路分解
Connection-oriented Demultiplexing

- TCP 套接字标识四元组 4-tuple:
  - 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
Connection-oriented demultiplexing: example



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：要点

- 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

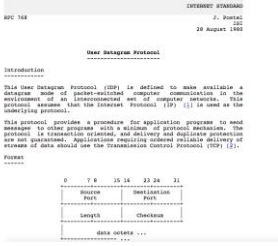
为什么要有UDP协议?

- 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]

- 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]



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

User Datagram Protocol

Introduction

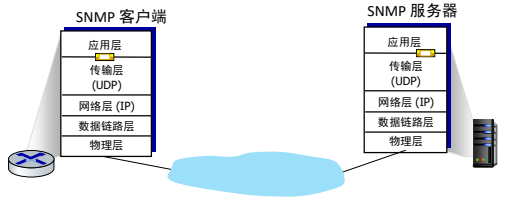
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

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].

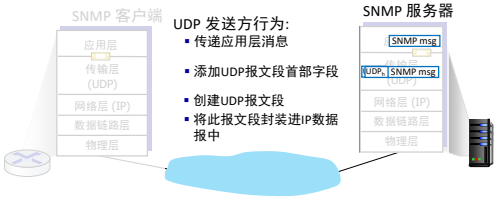
Format

0 7	8 15	16 23	24 31
Source Port		Destination Port	
Length		Checksum	
data octets ...			

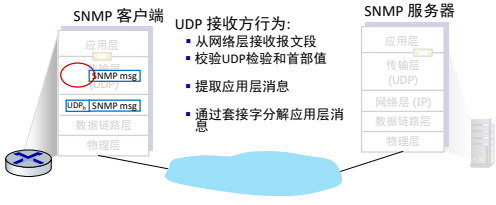
UDP: 传输层行为



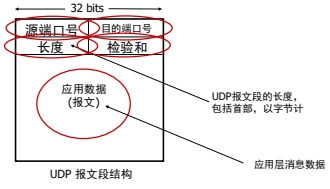
UDP:传输层行为



UDP: 传输层行为

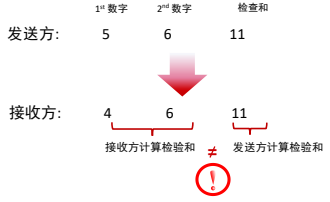


UDP数据段的头部
UDP segment header



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，还可能有差错吗？

校验和的例子
Internet Checksum: example

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，最高位进比特位的进位需要加到结果中

* 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

```
                1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0   0 1
                1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1   1 0
回卷 wraparound ① 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
sum             1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum        0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```

即使数字发生改变（比特翻转），检验和仍未变！

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，没有往返时间RTT（ Round-Trip Time ）产生
  - 当网络服务受损时可以正常工作
  - 有助于提高可靠性(检验和)
- 在网络层中的UDP之上构建附加功能 (e.g., HTTP/3)

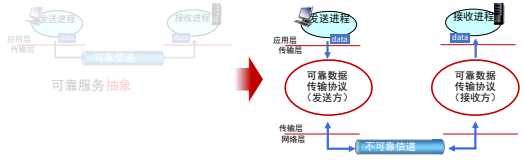
第三章：要点

- 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

可靠数据传输原理reliable data transfer protocol

可靠服务抽象

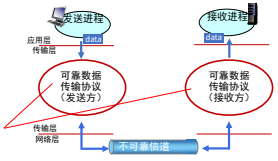
可靠数据传输原理reliable data transfer protocol



可靠服务的*实现*

可靠数据传输原理reliable data transfer protocol

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？）

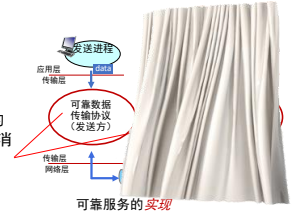


可靠服务的*实现*

可靠数据传输原理reliable data transfer protocol

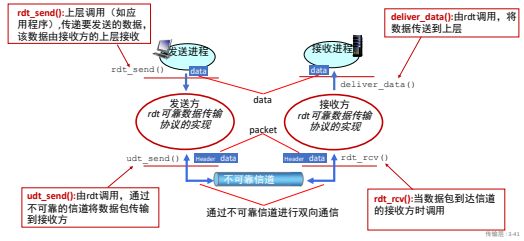
发送方，接收方不知道对方的“状态”，例如，是否收到消息？

- 除非通过信息传达



可靠服务的*实现*

可靠数据传输协议(rdt): 接口



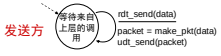
可靠数据传输: 基本概念

我们将:

- 增量开发发送方、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
- 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?

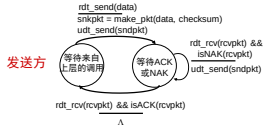
人们在交流中如何恢复“错误”？

rdt2.0: 比特差错信道

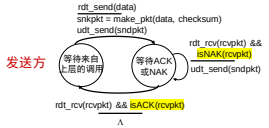
- 底层信道可能会翻转分组中的比特
  - 使用检验和检测比特差错
- *问题：*如何从错误中恢复过来?
  - *肯定确认acknowledgements (ACKs):* 接收者告知发送方pkt正确接收
  - *否定确认negative acknowledgements (NAKs):* 接收者告知发送方pkt存在差错
  - 发送方在收到NAK反馈后*重新传输retransmits*

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 specifications

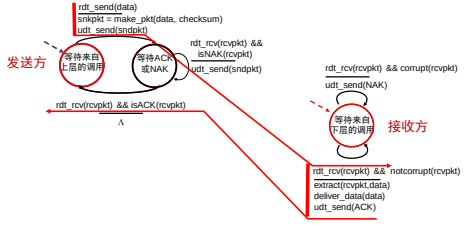


rdt2.0: FSM specification

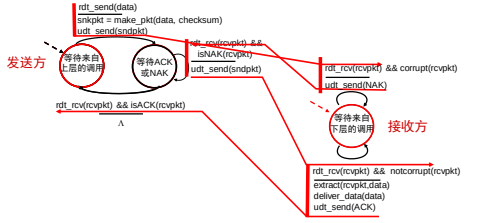


注意:接收方的“状态”（接收方是否正确地获得我的信息？）除非以某种方式由接收方通知发送方，否则发送方不知道
- 这就是我们需要一个协议的原因

rdt2.0: operation with no errors



rdt2.0: corrupted packet scenario



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!
- 不能知识重传：可能导致冗余

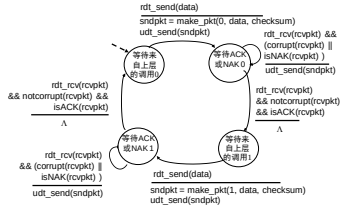
处理冗余:

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

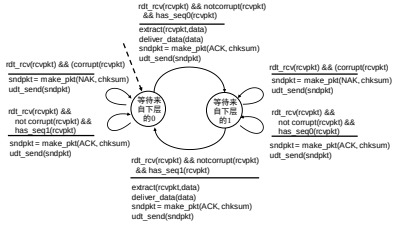
stop and wait

发送方发送一个分组，然后等待接收方响应

rdt2.1: 发送方, 处理受损的ACK/NAKs



rdt2.1: 接收方, handling garbled ACK/NAKs



rdt2.1: 讨论

发送方:

- 序号seq # 加入分组中
- 两个序号seq. #s (0,1)将够用，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:

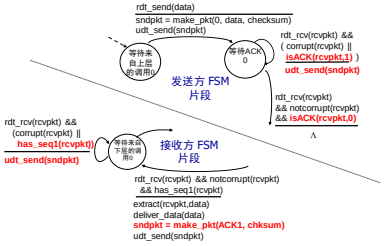
- 必须检查是否接收到的分组是冗余的
  - 状态指示是否0或1是所期待的分组序号seq #
- 注意：接收方不能知道它发送的最后一个ACK/NAK是否发送方已经接收了

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:*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就是使用这种方法。

rdt2.2: 发送方, 接收方片段



rdt3.0: 具有差错和丢包的信道

新的假设: *底层*信道也能丢失分组（数据或ACK）

- 检查和、序号、重传将是有帮助的，但还不够

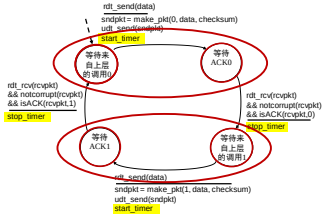
问题: 人们如何处理会话中丢失的语句？

rdt3.0:具有差错和丢包的信道

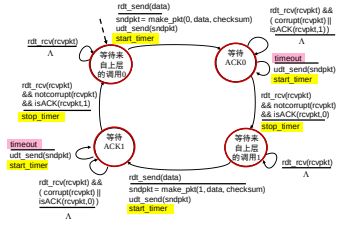
方法: 发送方等待ACK一段“合理的”时间

- 如在这段时间没有收到ACK则重传
- 如果分组（或ACK）指示延迟（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
  - 接收方必须定义被确认的分组序号
- 在“合理的”时间过后需要倒计时定时器来进行中断

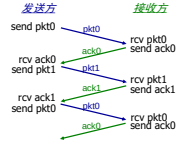
rdt3.0 发送方

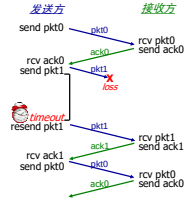


rdt3.0 发送方

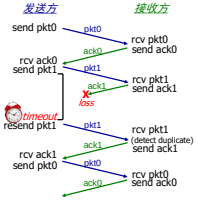


rdt3.0 运行情况

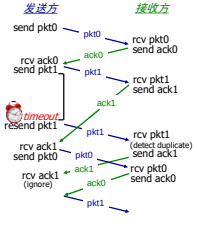




rdt3.0 运行情况



(c) ACK loss



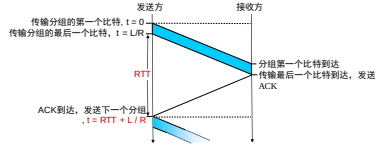
(d) premature timeout/ delayed ACK

rdt3.0的性能 (stop-and-wait)

- $U_{\text{发送方}}$: *利用率utilization* – 发送方用于发送的时间的比率
- 例子: 1 Gbps链路, 15ms端到端传播时延, 8000 bit分组
  - 将分组传递到信道的时间:

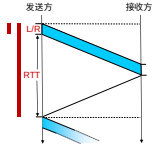
$$D_{trans} = \frac{L}{R} = \frac{8000\ bits}{10^9\ bits/sec} = 8\ microsecs$$

rdt3.0: 停等协议的运行



rdt3.0: 停等协议的运行

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



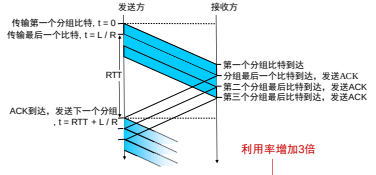
- rdt 3.0 能够工作，但性能不好!
- 网络协议限制了底层网络硬件所提供的能力

rdt3.0: 流水线协议

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

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

流水线协议: 增加利用率



利用率增加3倍

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

回退N步（Go-Back-N）：发送方

- 发送方：“窗口”最大为N，允许N个连续的没有应答分组
  - 在分组首部需要k比特序号，2^k=N

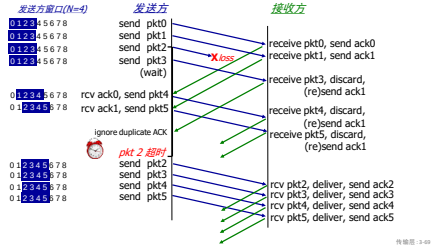
- *累积cumulative ACK:* ACK(*n*):表明接收方已正确接收到序号为n的以前且包括n在内的所有分组
  - 正在接收receiving ACK(*n*):在n+1开始向前移动窗口
- 对每个传输中的分组使用定时器timer for oldest in-flight packet
- *超时 (n):* 若超时，重传窗口中的分组n及所有更高序号的分组

回退N步（Go-Back-N）：接收方

- ACK-only: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:

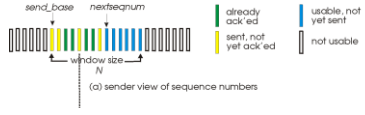
回退N步（Go-Back-N）示意



选择重传（Selective Repeat）

- 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应答分组数量

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，发送报文段

timeout(*n*):
- 重传分组n，重启其计时器

ACK(*n*) 在 [sendbase,sendbase+N]:
- 标记分组n已经收到
- 如果n是最小未收到应答的分组，向前滑动窗口base指针到下一个未确认序号

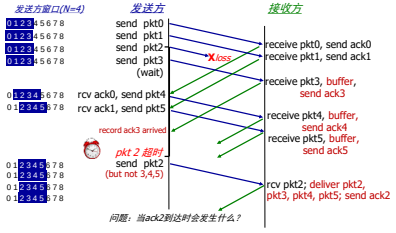
接收方

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

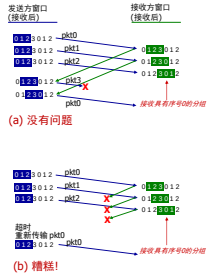
选择重传：示意



选择重传的窘境

例子:

- 序号: 0, 1, 2, 3
- 窗口大小 = 3

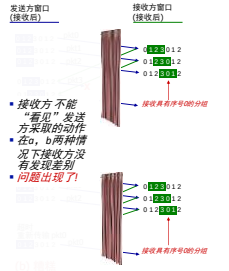


选择重传的窘境

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？



- *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!*

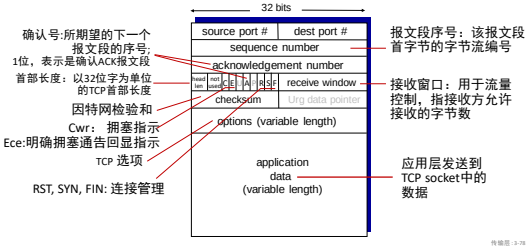
第三章：要点

- 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
- 累积 ACKs
- 流水线:
  - TCP拥塞和流量控制设置窗口大小
- 面向连接connection-oriented:
  - 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



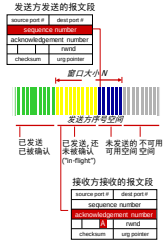
TCP 序号和确认号

序号:
- 报文段中第一个数据字节在字节流中的位置编号

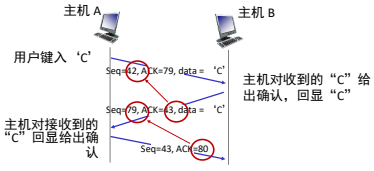
确认号:
- 期望从对方收到下一个字节的序号
- 累积确认

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

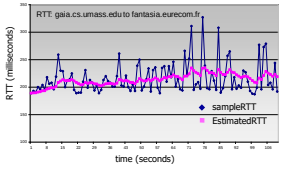
问题: 如何估计RTT?
- `SampleRTT`:从报文段发出到接收到确认的时间进行测量
  - 仅在某个时刻做一次SampleRTT测量
  - 绝不为已被重传的数据段估计SampleRTT
- `SampleRTT` 会变化，希望估计的RTT "较平滑"
  - 使用最近测量值的平均，并不是当前的`SampleRTT`

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/

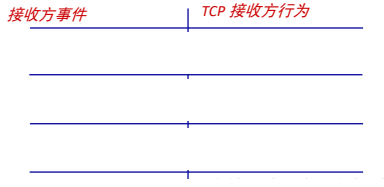
TCP 发送方 (simplified)

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`

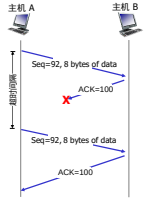
事件: 定时器超时
- 重传引起超时的报文段
- 计时器重启

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

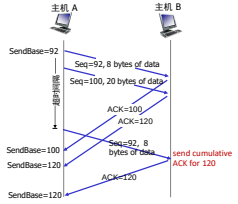
TCP 接收方: ACK 产生 [RFC 5681]



TCP: 重传情况

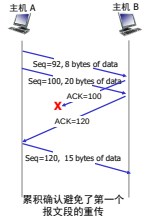


由于确认丢失而重传



报文段100没有重传

TCP: 重传情况



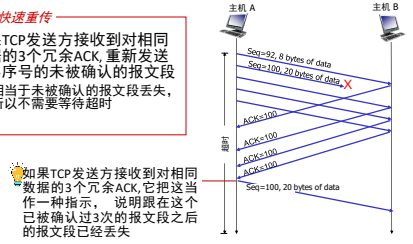
累积确认避免了第一个报文段的重传

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



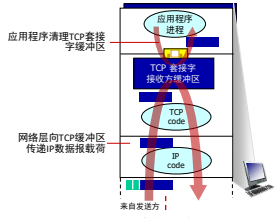
如果TCP发送方接收到对相同数据的3个冗余ACK, 它把这当作一种指示， 说明跟在这个已被确认过3次的报文段之后的报文段已经丢失

第三章：要点

- 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

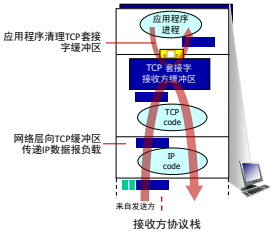
问题: 如果网络层传送数据的速率比应用层清理套接字缓冲速度更快会发生什么？



接收方协议栈

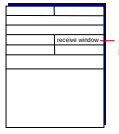
TCP 流量控制

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？

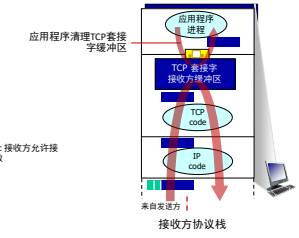


TCP 流量控制

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？



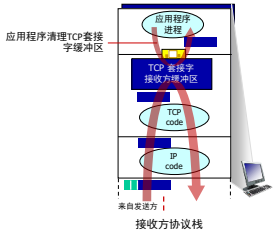
流量控制: 接收方允许接收的字节数



TCP 流量控制

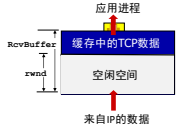
问题: 如果网络层传送数据的速率比应用层清理套接字缓冲速度更快会发生什么？

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。



TCP 流量控制

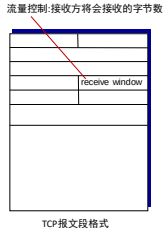
- 接收方在报文段接收窗口字段（rwnd, receive window）中“通告”其接收缓冲区的剩余空间
  - **`RcvBuffer` 根据套接字选项确定大小（通常的默认值为4096字节）**
  - 许多操作系统自动适应 **`RcvBuffer`**
- 发送方 要限制未确认的数据不超过接收窗口（**`rwnd`**）
- 保证接收缓冲区不溢出



接收窗口（rwnd）和接收缓存（RcvBuffer）

TCP 流量控制

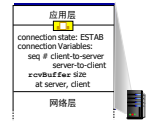
- 接收方在报文段接收窗口字段（rwnd）中“通告”其接收缓冲区的剩余空间
  - **`RcvBuffer` 根据套接字选项确定大小（通常的默认值为4096字节）**
  - 许多操作系统自动适应 **`RcvBuffer`**
- 发送方 要限制未确认的数据不超过接收窗口（**`rwnd`**）
- 保证接收缓冲区不溢出



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）

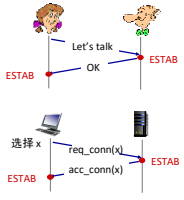



```
Socket ClientSocket =
  newSocket("hostname","port number");
```

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

同意建立连接

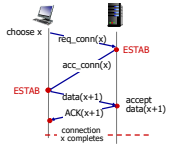
2次握手:



问题: 2次握手在网络层总是有效吗?

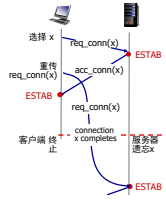
- 多变的延迟
- 由于报文丢失导致重传
- 消息重新排序
- 彼此“看不见”

2次握手情况



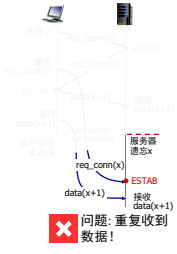
没问题!

2次握手情况



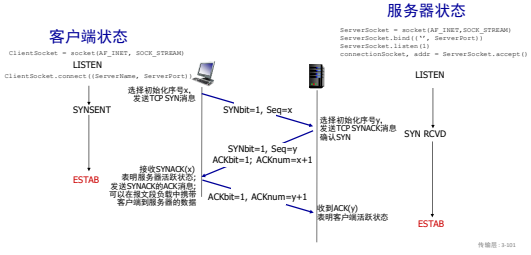
问题: 半打开连接! (没有客户端)

2次握手情况



问题: 重复收到数据!

TCP 3次握手

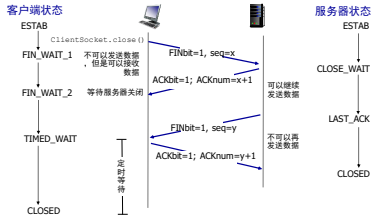


人类的3次握手协议

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位置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
- 双方的FIN可以同时进行交换

关闭TCP连接 - 4次挥手



第三章：要点

- 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
- 可靠面向连接的传输：TCP
- **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!
- 网络中top10问题之一!

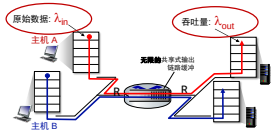
流量控制: 发送方给接收方发送的数据太快。

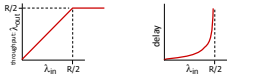
拥塞的原因与代价: 情况1

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
- 不需要重传

问题: 当发送速率 λ_{in} 接近R/2时会发生什么?



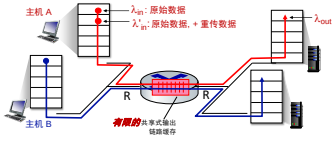


每连接的最大吞吐量（maximum per-connection throughput）: R/2

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

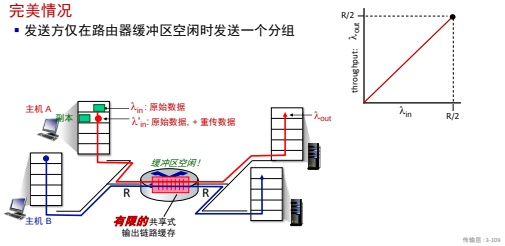
- 一个路由器，*有限*缓冲区
- 发送方重传丢失的超时的数据分组
  - 应用层输入 = 应用层输出: $\lambda_{in} = \lambda_{out}$
  - 传输层输入包括重传: $\lambda'_{in} \geq \lambda_{in}$



拥塞的原因与代价: 情况2

完美情况

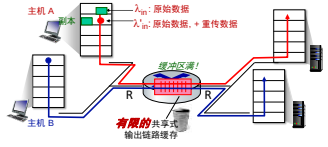
- 发送方仅在路由器缓冲区空闲时发送一个分组



拥塞的原因与代价: 情况2

部分完美情况

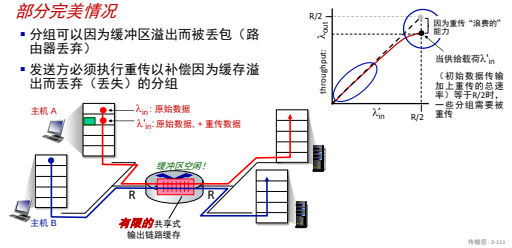
- 分组可能因为缓冲区溢出而被丢包（路由器丢弃）
- 发送方必须执行重传以补偿因为缓存溢出而丢弃（丢失）的分组



拥塞的原因与代价: 情况2

部分完美情况

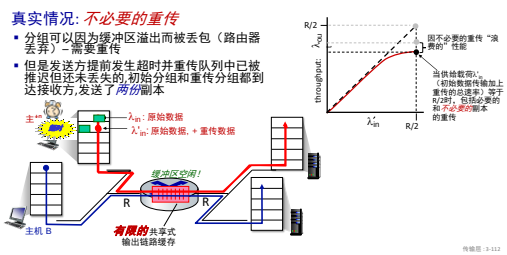
- 分组可以因为缓冲区溢出而被丢包（路由器丢弃）
- 发送方必须执行重传以补偿因为缓存溢出而丢弃（丢失）的分组



拥塞的原因与代价: 情况2

真实情况: *不必要的重传*

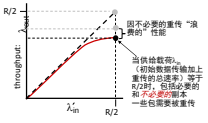
- 分组可以因为缓冲区溢出而被丢包（路由器丢弃）– 需要重传
- 但是发送方提前发生超时并重传队列中已被推迟但还未丢失的,初始分组和重传分组都到达接收方,发送了*两份*副本



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
- 但是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,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”：

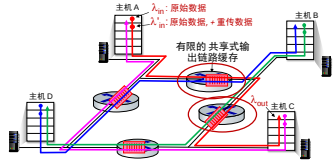
- 比额定的“吞吐量”做更多的工作
- 不必要的重传：链路承载分组的多个拷贝
  - 降低了最大可获得的吞吐量

拥塞的原因与代价: 情况3

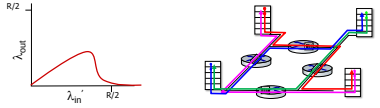
- 四个发送方
- 多跳路径
- 超时/重传

问题: 随着λ_{in}和λ'_{in}的增加将发生什么情况?

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



拥塞的原因与代价: 情况3

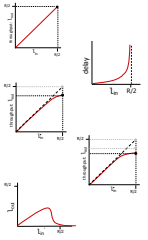


另一个拥塞的“代价”：

- 当分组丢失时，任何用于传输该分组的上有传输能力都被浪费！

拥塞的原因与代价: 感悟

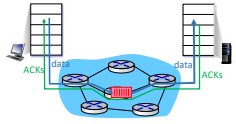
- 吞吐量永远不能超过传输能力
- 延迟随着传输速度接近传输能力而增加
- 丢失/重传降低有效吞吐量
- 不必要的重传进一步降低了有效吞吐量
- 上游传输能力/缓冲区被下游丢包所浪费



拥塞控制方法

端到端的拥塞控制:

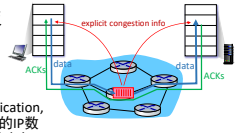
- 不能从网络得到明确的反馈
- 根据观察到的时延和丢包现象*推断*出拥塞
- 这是TCP所采用的方法



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
  - TCP 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,
  - ATM可用比特率拥塞控制, DECbit 协议



第三章：要点

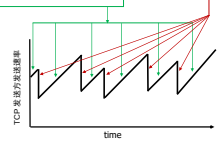
- 传输层服务
- 多路复用和多路分解
- 无连接传输：UDP
- 可靠数据传输原理
- 面向连接的传输：TCP
- 拥塞控制原理
- **TCP拥塞控制**

TCP 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



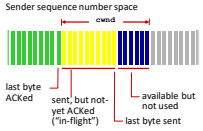
AIMD 锯齿状行为:
带宽*探测*

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!
  - 具有理想的稳定性

TCP 拥塞控制: 细节



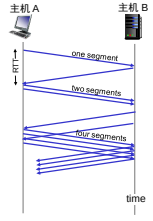
TCP 发送行为:
- 发送cwnd 字节,在RTT时间内等待ACK，然后发送更多的字节

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

- 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)

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
  - 每个RTT时间后翻倍 **cwnd**
  - 每收到ACK，增加拥塞窗口
- *总结:* 初始速率很低，但以指数速度增加

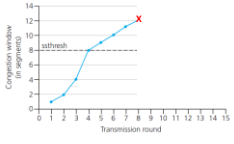


TCP: 从慢启动到拥塞避免(congestion avoidance)

问题: 什么时候从指数增长变为线性增长?

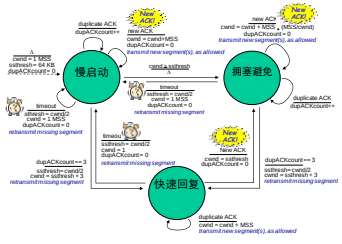
*回答:*当cwnd在超时之前达到其值的1/2时

实现方法:
- 阈值变量 **ssthresh**
- 在丢包事件发生时，阈值 **ssthresh** 设置为发生丢包以前的 **cwnd** 的一半



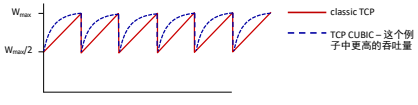
* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

总结: TCP 拥塞控制



TCP CUBIC

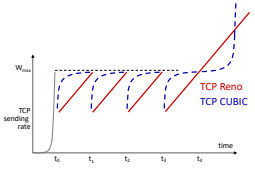
- 有没有比AIMD更好的方法来“探测”可用带宽?
- Insight/intuition:
  - W_{max}:检测到拥塞丢包时的发送速率
  - 瓶颈链路的拥塞状态可能 (?)还没有大幅度改善
  - 在减半速率/窗口后，最初以更快的速度爬升到W_{max}，但随后以更慢的速度接近W_{max}



TCP CUBIC

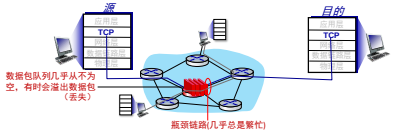
- K: TCP窗口大小将达到W_{max}的时间点
  - K本身是可调的
- 增加W作为当前时间和K之间距离的立方的函数
  - 离K越远，增幅越大
  - 当接近K时，小幅增加

- 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。



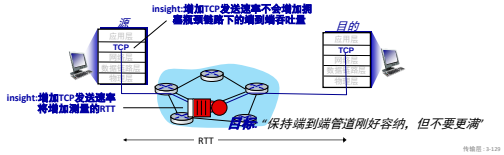
TCP 和拥塞的“瓶颈链路”

- 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*



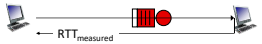
TCP 和拥塞的“瓶颈链路”

- 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/缓冲



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

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

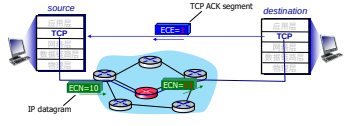
- 无诱导/强制丢失的拥塞控制
- 在保持低延迟的同时最大化整个过程

- 许多部署的tcp采用基于延迟的方法
  - 部署在谷歌（内部）主干网络

显式拥塞通知
Explicit congestion notification (ECN)

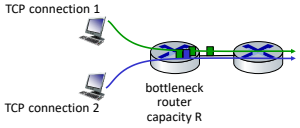
TCP部署通常实施网络辅助拥塞控制：

- 网络路由器标记的IP头（ToS字段）中的两位表示拥塞
  - 确定网络运营商选择的标记的策略
- 携带到目的地的拥挤指示
- 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C、E位标记）



TCP 公平性

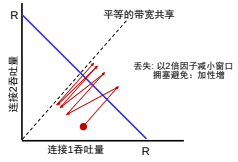
公平目标: 如果K个TCP会话共享带宽为R的瓶颈链路，每个会话应有R/K的平均链路速率



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，在理想情况下:

- 相同RTT
- 仅在拥塞避免中限定会话数目

公平性（续）

公平性和UDP

- 多媒体应用通常不用TCP
  - 不希望拥塞控制抑制速率
- 使用UDP:
  - 音频/视频以恒定速率发送，能容忍报文丢失

公平性和并行TCP连接

- 不能防止2台主机之间打开多个并行连接
- WEB浏览器以这种方式工作
- 例子:支持9个连接的速率R的链路：
  - 新应用请求一个TCP连接，则得到R/10的带宽
  - 新应用请求11个TCP连接，则得到R/2的带宽

第三章：总结

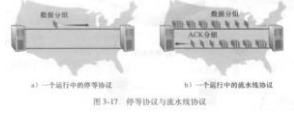
- 传输层服务原则:
  - 多路复用与多路分解
  - 可靠数据传输
  - 流量控制
  - 拥塞控制
- 因特网中的实例和实现
  - UDP
  - TCP

作业

- 在我们的rdt协议中，为什么需要引入序号?
- 在我们的rdt协议中，为什么需要引入定时器?

作业

- 考虑显示在图3-17中的网络跨越国家的例子（两个端系统之间的光速往返传播时延RTT大约为30毫秒。 假定彼此通过一条发送速率R为1Gbps （每秒10^9比特） 的信道相连 ）。窗口长度设置成多少时， 才能使该信道的利用率超过90%?假设分组的长度为1500字节（包括首部字段和数据） 。



作业

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

a. 在从主机A发往B的第二个报文段中，序号、源端口号和目的端口号各是什么？
b. 如果第一个报文段在第二个报文段之前到达，在第一个到达报文段的确认中，确认号、源端口号和目的端口号各是什么？
c. 如果第二个报文段在第一个报文段之前到达，在第一个到达报文段的确认中，确认号是什么？
d. 假定由A发送的两个报文段按序到达B。第一个确认丢失了而第二个确认在第一个超时间隔之后到达。画出时序图，显示这些报文段和发送的所有其他报文段和确认。（假设没有其他分组丢失。）对于图上每个报文段，标出序号和数据的字节数量；对于你增加的每个应答，标出确认号。

作业

- 比较GBN、SR和TCP（无延时的ACK）。假设对所有3个协议的超时值足够长，使得5个连续的数据报文段及其对应的ACK能够分别由接收主机（主机B）和发送主机（主机A）收到（如果在信道中无丢失）。假设主机A向主机B发送5个数据报文段，并且第二个报文段（从A发送）丢失。最后，所有5个数据报文段已经被主机B正确接收。
  a. 主机A总共发送了多少报文段和主机B总共发送了多少ACK?它们的序号是什么？对所有3个协议回答这个问题。
  b. 如果对所有3个协议超时值比5 RTT长得多，则哪个协议在最短的时间间隔中成功地交付所有5个数据报文段？

作业

- 考虑图3-58。假设TCP Reno是一个经历如上所示行为的协议，回答下列问题。在各种情况中，简要地论证你的回答。

a. 指出TCP慢启动运行时的时间间隔。
b. 指出TCP拥塞避免运行时的时间间隔。
c. 在第16个传输轮回之后，报文段的丢失是根据3个冗余ACK还是根据超时检测出来的？
d. 在第22个传输轮回之后，报文段的丢失是根据3个冗余ACK还是根据超时检测出来的？
e. 在第1个传输轮回里，ssthresh的初始值设置为多少？
f. 在第18个传输轮回里，ssthresh的值设置为多少？
g. 在第24个传输轮回里，ssthresh的值设置为多少？
h. 在哪个传输轮回内发送第70个报文段？
i. 假定在第26个传输轮回后，通过收到3个冗余ACK检测出有分组丢失，拥塞的窗口长度和ssthresh的值应当是多少？
j. 假定使用TCP Tahoe（而不是TCP Reno），并假定在第16个传输轮回收到3个冗余ACK。在第19个传输轮回，ssthresh和拥塞窗口长度是什么？
k. 再次假设使用TCP Tahoe,在第22个传输轮回有一个超时事件。从第17个传输轮回到第22个传输轮回（包括这两个传输轮回），一共发送了多少分组？

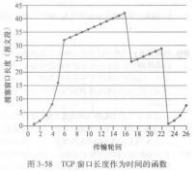
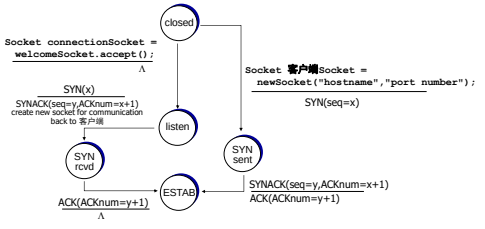


图 3-58 TCP 窗口长度作为时间的函数

Additional Chapter 3 slides

TCP 3-way handshake FSM



Closing a TCP Connection