第六章 传输层

授课教师：张圣林

南开大学

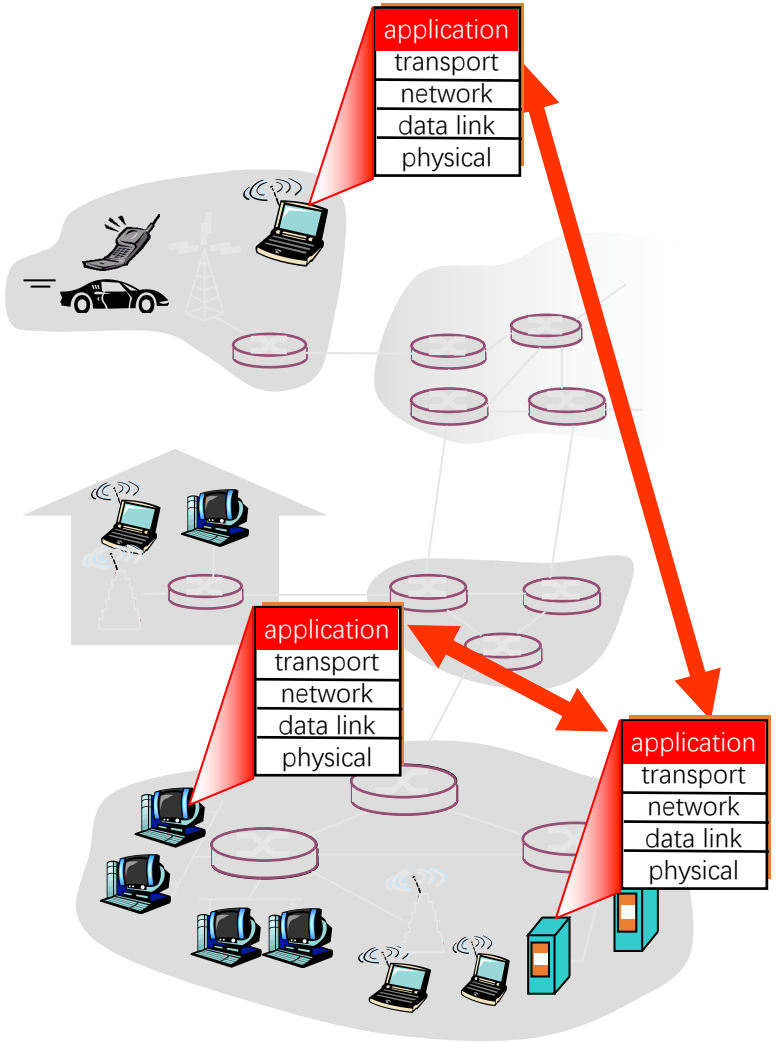
本章目标

- 掌握传输层服务原理
  - 传输层复用和分用
  - 可靠数据传输
  - 流量控制
  - 拥塞控制
- 掌握因特网传输层协议：
  - UDP协议
  - TCP协议

本章内容

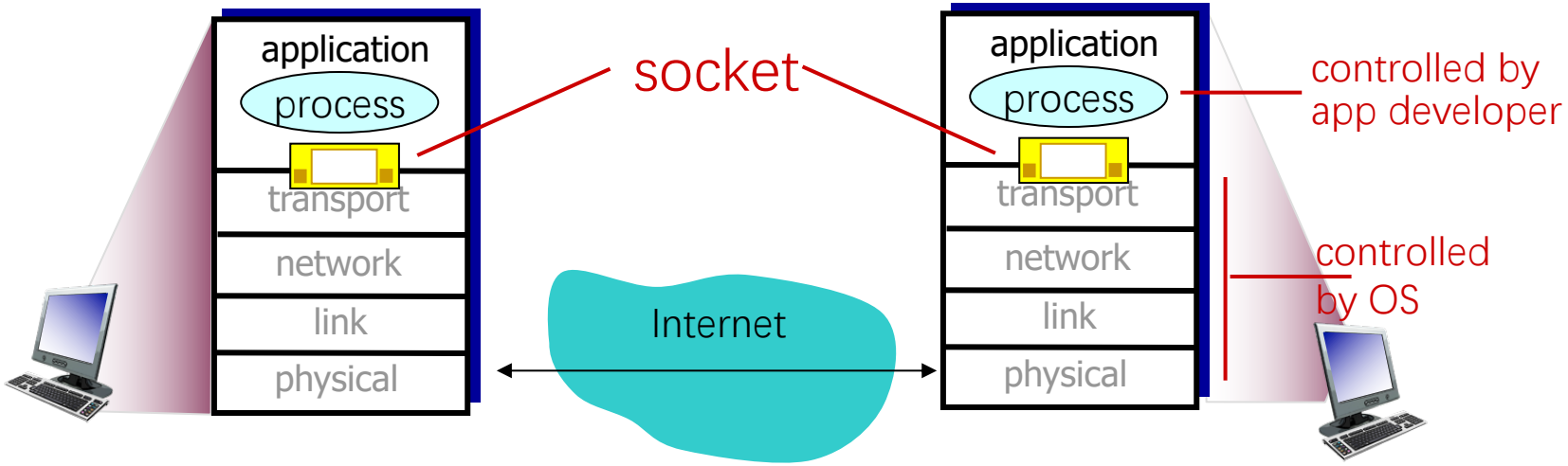
传输层的位置

- 传输层位于应用层和网络层之间：
  - 基于网络层提供的服务，向分布式应用程序提供通信服务
- 按照因特网的“端到端”设计原则：
  - 应用程序只运行在终端上，即不需要为网络设备编写程序
- 站在应用程序的角度：
  - 传输层应提供进程之间本地通信的抽象：即运行在不同终端上的应用进程仿佛是直接连在一起的



不同终端上的进程如何通信？

- 设想在应用程序和网络之间存在一扇“门”：
  - 需要发送报文时：发送进程将报文推到门外
  - 门外的运输设施（因特网）将报文送到接收进程的门口
  - 需要接收报文时：接收进程打开门，即可收到报文
- 在TCP/IP网络中，这扇“门”称为套接字（socket），是应用层和传输层的接口，也是应用程序和网络之间的API



传输层和网络层的关系：一个类比例子

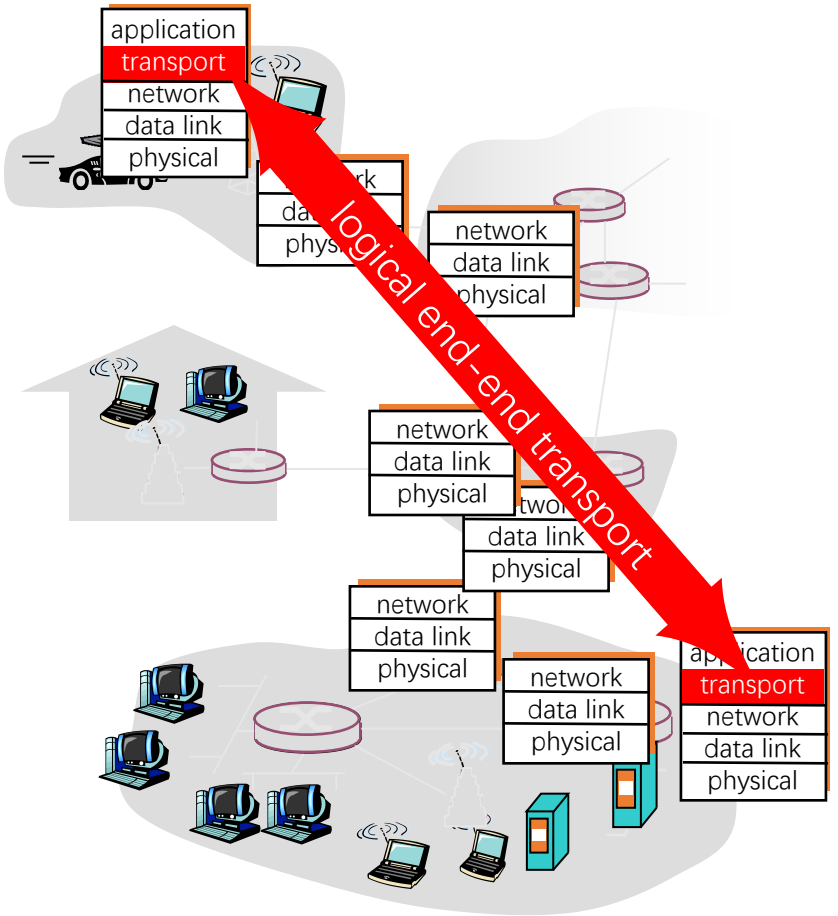
- 设想A家庭有12个孩子，B家庭也有12个孩子，他们每周通过手写信件进行联系：
  - A家庭推选Ann，B家庭推选Bill，分别负责各自家庭的信件收发
  - Ann和Bill定期收集家庭中其他孩子的信件，投递到门口的邮筒里
  - Ann和Bill定期打开家庭邮箱，取出里面的信件，分发给其他孩子
  - 邮政系统负责将邮筒里的信件投递到收信人的家庭邮箱里

- 类比：
  - 进程 = 孩子
  - 进程间通信 = 孩子之间通信
  - 应用报文 = 信件
  - 传输层 = Ann 和 Bill（提供人到人的服务）
  - 终端 = 家庭住宅
  - 网络层 = 邮政系统（提供门到门的服务）

传输层依赖网络层服务，并扩展网络层服务

传输层提供什么服务？

- 因特网的网络层提供**“尽力而为”**的服务：
  - 网络层尽最大努力在终端间交付分组，但不提供任何承诺
  - 具体来说，不保证交付，不保证按序交付，不保证数据完整，不保证延迟，不保证带宽等
- 传输层的**有所为**、**有所不为**:
  - 传输层可以通过差错恢复、重排序等手段提供可靠、按序的交付服务
  - 但传输层无法提供延迟保证、带宽保证等服务



因特网传输层提供的服务

- 最低限度的传输服务：
  - 将终端-终端的数据交付扩展到进程-进程的数据交付（6.3节）
  - 报文检错
- 增强服务：
  - 可靠数据传输
  - 流量控制
  - 拥塞控制

- 因特网传输层通过UDP协议和TCP协议，向应用层提供两种不同的传输服务：
  - UDP协议（6.4节）：仅提供最低限度的传输服务
  - TCP协议（6.5节，6.7节）：提供基础服务和增强服务

本章内容

6.1 概述和传输层服务

6.2 套接字编程

6.3 传输层复用和分用

6.4 无连接传输：UDP

6.5 面向连接的传输：TCP

6.6 理解网络拥塞

6.7 TCP拥塞控制

6.8 拥塞控制的发展

6.9 传输层协议的发展

传输层基本服务

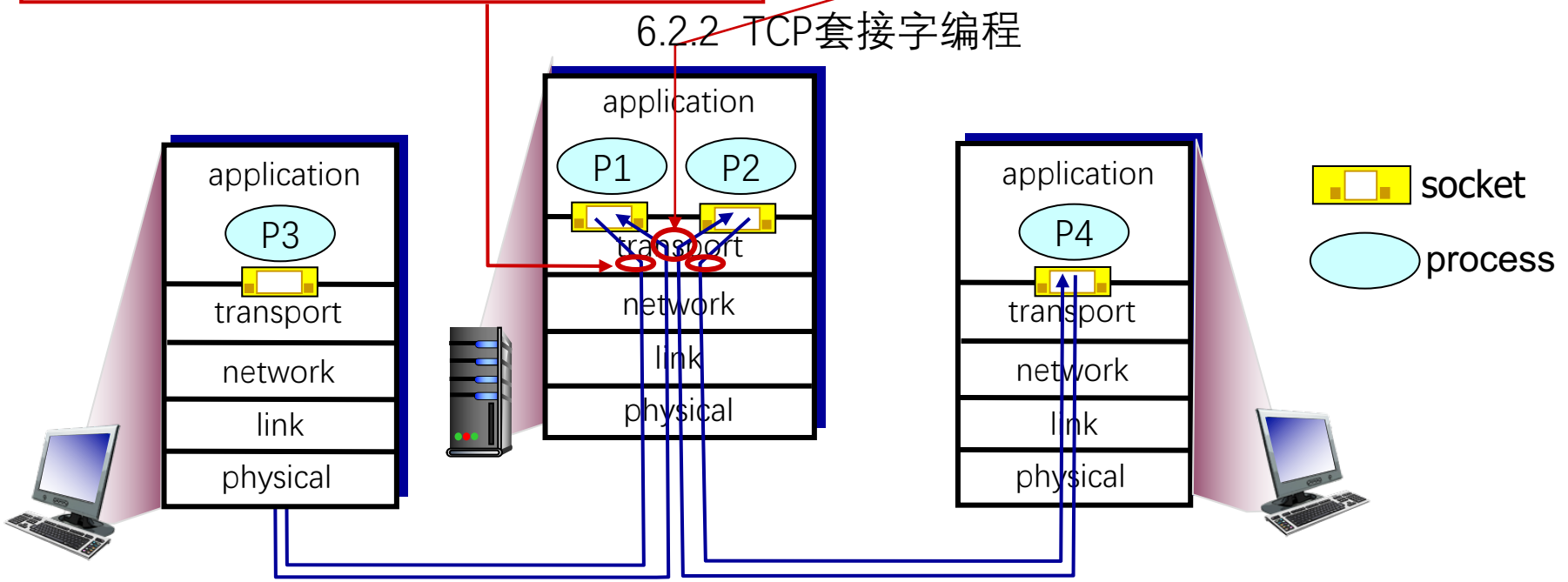
传输层基本服务：将主机间交付扩展到进程间交付，通过复用和分用实现

（发送端）复用:

传输层从多个套接字收集数据，交给网络层发送

（接收端）分用:

传输层将从网络层收到的数据，交付给正确的套接字



如何进行复用和分用：类比的例子

➢为将邮件交付给收信人：

- 每个收信人应有一个信箱，写有收信人地址和姓名（唯一标识）
- 信封上有收信人的地址和名字

➢为将报文段交付给套接字：

- 主机中每个套接字应分配一个唯一的标识
- 报文段中包含接收套接字的标识

➢复用：

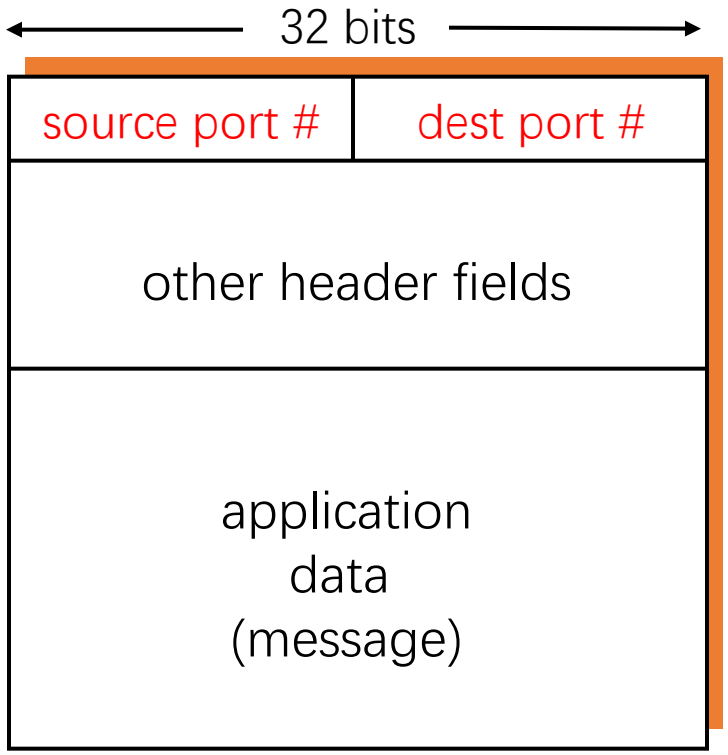
- 发送方传输层将套接字标识置于报文段中，交给网络层

➢分用：

- 接收方传输层根据报文段中的套接字标识，将报文段交付到正确的套接字

套接字标识与端口号

- 端口号是套接字标识的一部分：
  - 每个套接字在本地关联一个端口号
  - 端口号是一个16比特的数
- 端口号的分类：
  - 熟知端口：0 ~ 1023，由公共域协议使用
  - 注册端口：1024 ~ 49151，需要向IANA (Internet Assigned Numbers Authority)注册才能使用
  - 动态和/或私有端口：49152 ~ 65535，一般程序使用
- 报文段中有两个字段携带端口号
  - 源端口号：与发送进程关联的本地端口号
  - 目的端口号：与接收进程关联的本地端口号



TCP/UDP报文段格式

套接字端口号的分配

- 自动分配：
  - 创建套接字时不指定端口号
  - 由操作系统从49152～65535中分配
  - 客户端通常使用这种方法
- 使用指定端口号创建套接字：
  - 创建套接字时指定端口号
  - 实现公共域协议的服务器应分配众所周知的端口号（0～1023）
  - 服务器通常采用这种方法

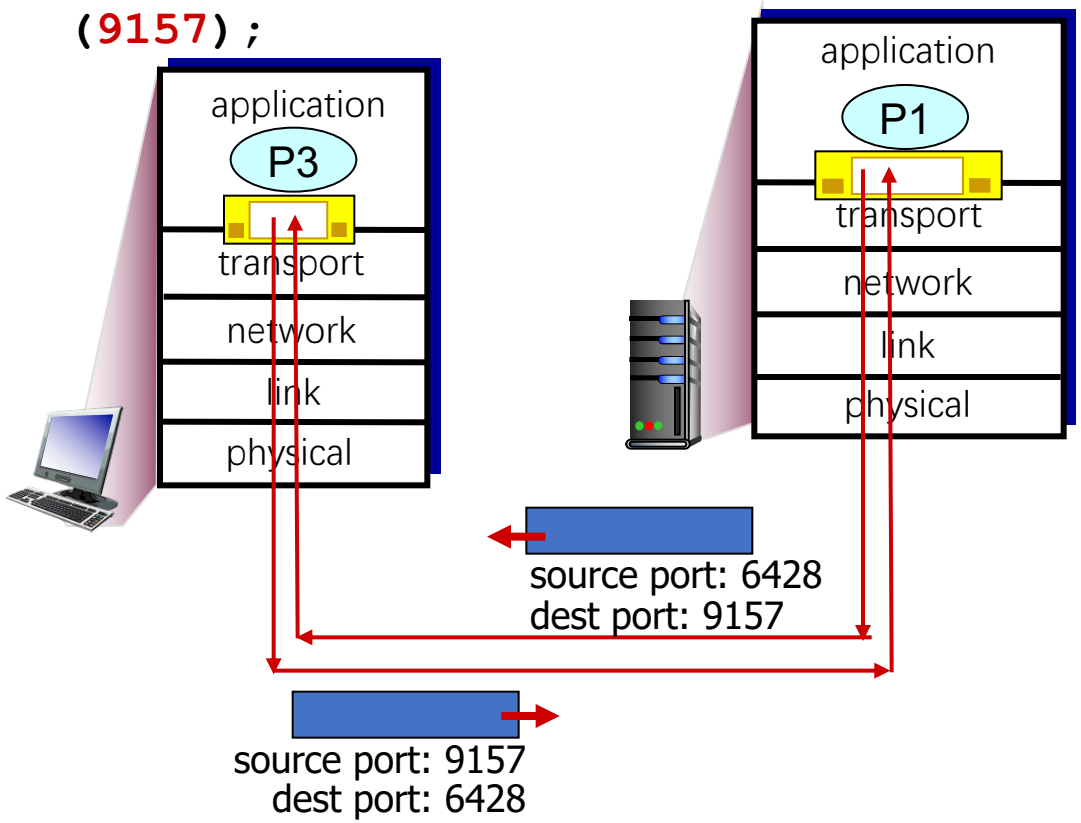
UDP分用的方法

- UDP套接字使用<IP地址, 端口号>二元组进行标识
- 接收方传输层收到一个UDP报文段后：
  - 检查报文段中的目的端口号，将UDP报文段交付到具有该端口号的套接字
  - <目的IP地址，目的端口号> 相同的UDP报文段被交付给同一个套接字，与 <源IP地址，源端口号> 无关
  - 报文段中的 <源IP地址，源端口号> 被接收进程用来发送响应报文

UDP分用: 举例

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

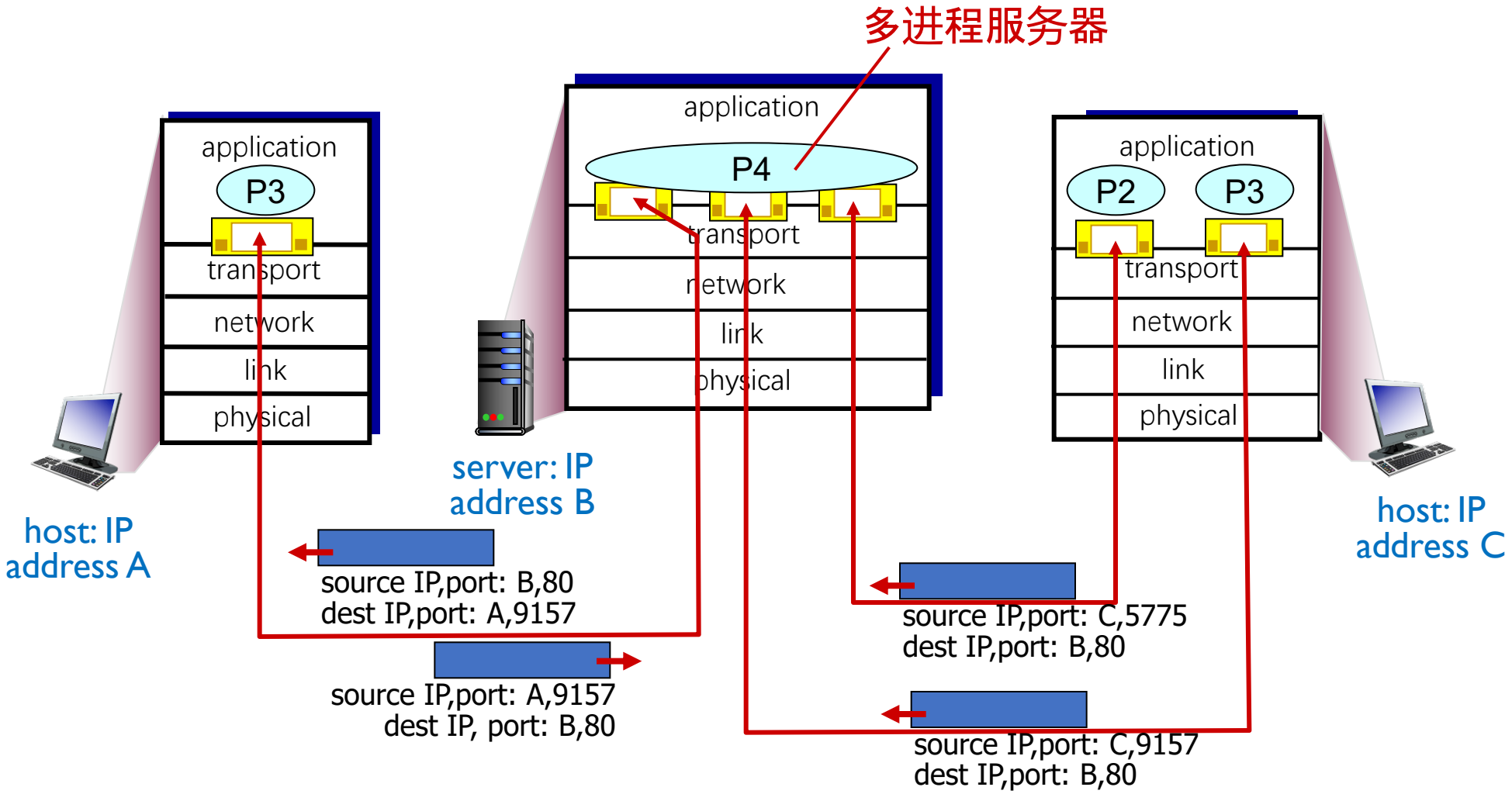
```
DatagramSocket
 serverSocket = new
 DatagramSocket
 (6428);
```



TCP服务器使用的套接字

- 一个TCP服务器为了同时服务很多个客户，使用两种套接字
- 监听套接字：
  - 服务器平时在监听套接字上等待客户的连接请求，该套接字具有众所周知的端口号
- 连接套接字：
  - 服务器在收到客户的连接请求后，创建一个连接套接字，使用临时分配的端口号
  - 服务器同时创建一个新的进程，在该连接套接字上服务该客户
  - **每个连接套接字只与一个客户通信**，即只接收具有以下四元组的报文段：
    - 源IP地址 = 客户IP地址，源端口号 = 客户套接字端口号
    - 目的IP地址 = 服务器IP地址，目的端口号 = 服务器监听套接字的端口号
- **连接套接字需要使用<源IP地址，目的IP地址，源端口号，目的端口号>四元组进行标识**，服务器使用该四元组将TCP报文段交付到正确的连接套接字

TCP分用: 举例


多进程服务器
application
P3
transport
network
link
physical
application
P4
transport
network
link
physical
application
P2
P3
transport
network
link
physical
host: IP address A
server: IP address B
host: IP address C
source IP,port: B,80
dest IP,port: A,9157
source IP,port: A,9157
dest IP, port: B,80
source IP,port: C,5775
dest IP,port: B,80
source IP,port: C,9157
dest IP,port: B,80

小结

- UDP套接字
  - 使用<IP地址，端口号>二元组标识UDP套接字
  - 服务器使用一个套接字服务所有客户
- TCP套接字
  - 使用<源IP地址，目的IP地址，源端口号，目的端口号> 四元组标识连接套接字
  - 服务器使用一个监听套接字和多个连接套接字服务多个客户，每个连接套接字服务一个客户

本章内容

6.1 概述和传输层服务

6.2 套接字编程

6.3 传输层复用和分用

6.4 无连接传输：UDP

6.5 面向连接的传输：TCP

6.6 理解网络拥塞

6.7 TCP拥塞控制

6.8 拥塞控制的发展

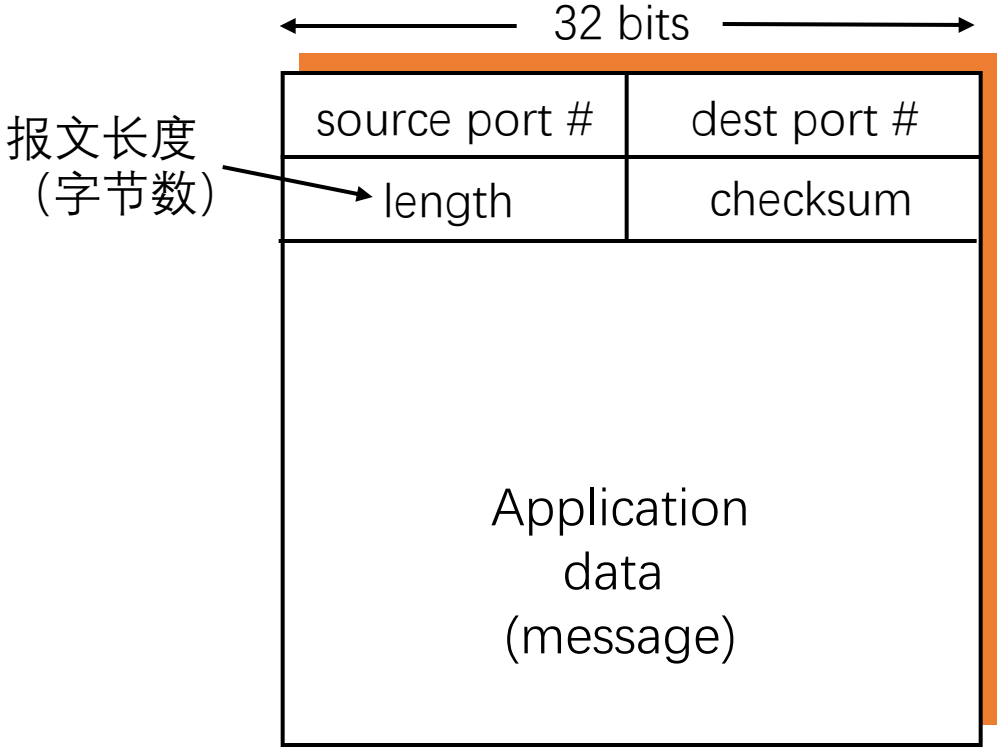
6.9 传输层协议的发展

UDP提供的服务

- 网络层提供的服务（best-effort service）：
  - 尽最大努力将数据包交付到目的主机
  - 不保证投递的可靠性和顺序
  - 不保证带宽及延迟要求
- UDP提供的服务：
  - 进程到进程之间的报文交付
  - 报文完整性检查（可选）：检测并丢弃出错的报文
- UDP需要实现的功能：
  - 复用和分用
  - 报文检错

UDP报文段结构

- UDP报文：
  - 报头：携带协议处理需要的信息
  - 载荷（payload）：携带上层数据
- 用于复用和分用的字段：
  - 源端口号
  - 目的端口号
- 用于检测报文错误的字段：
  - 报文总长度
  - 校验和（checksum）



UDP报文格式

UDP校验和（checksum）

校验和字段的作用：对传输的报文段进行检错

发送方：

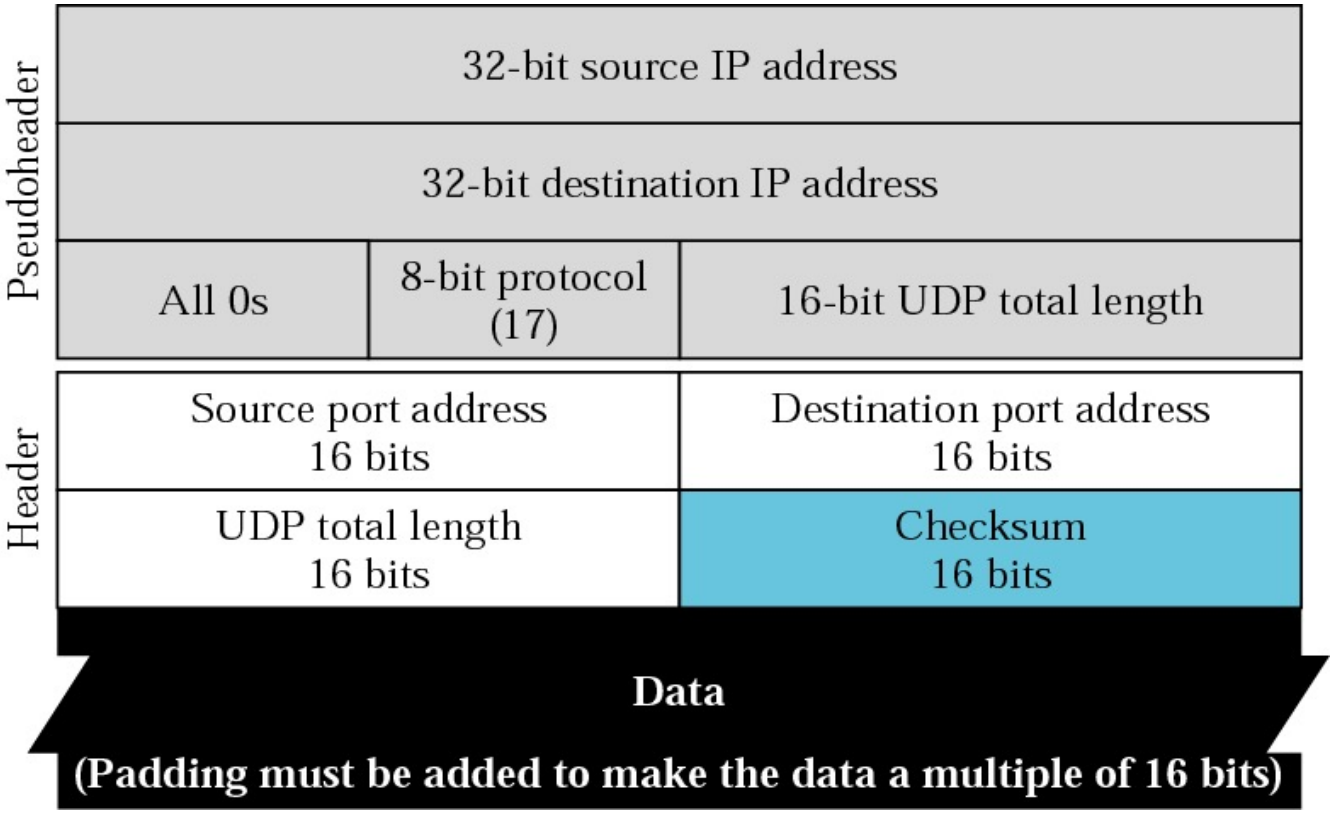
- 将报文段看成是由16比特整数组成的序列
- 对这些整数序列计算校验和
- 将校验和放到UDP报文段的checksum字段

接收方：

- 对收到的报文段进行相同的计算
- 与报文段中的checksum字段进行比较：
  - 不相等：说明报文段有错误
  - 相等：认为报文段没有错误

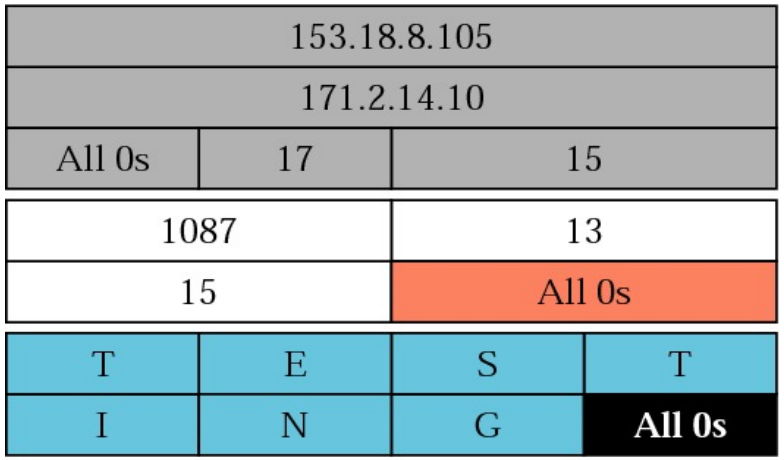
UDP校验和计算

- 计算UDP校验和时，要包括伪头、UDP头和数据三个部分
- UDP伪头是一个虚拟数据结构，取自IP报头，并非UDP报文的实际有效成分，包括：
  - 源IP地址，目的IP地址
  - UDP的协议号
  - UDP报文段总长度
- 计算校验和时包含伪头信息，是为了避免由于IP地址错误等造成的误投递
- UDP校验和的使用是可选的，若不计算校验和，该字段填入0



UDP校验和计算举例

- 计算校验和时，checksum字段填0
- 接收方对UDP报文（包括校验和）及伪头求和，若结果为0xFFFF，认为没有错误



10011001 00010010	153.18
00001000 01101001	8.105
10101011 00000010	171.2
00001110 00001010	14.10
00000000 00010001	0 and 17
00000000 00001111	15
00000100 00111111	1087
00000000 00001101	13
00000000 00001111	15
00000000 00000000	0 (checksum)
01010100 01000101	T and E
01010011 01010100	S and T
01001001 01001110	I and N
01000111 00000000	G and 0 (padding)
10010110 11101011	Sum
01101001 00010100	Checksum

为什么需要UDP？

为什么需要UDP？

- 应用可以尽可能快地发送报文：
  - 无建立连接的延迟
  - 不限制发送速率（不进行拥塞控制和流量控制）
- 报头开销小
- 协议处理简单

UDP适合哪些应用？

- 容忍丢包但对延迟敏感的应用：
  - 如流媒体
- 以单次请求/响应为主的应用：
  - 如DNS
- 若应用要求基于UDP进行可靠传输：
  - 由应用层实现可靠性

本章内容

6.1 概述和传输层服务
6.2 套接字编程
6.3 传输层复用和分用
6.4 无连接传输：UDP
6.5 面向连接的传输：TCP
6.6 理解网络拥塞
6.7 TCP拥塞控制
6.8 拥塞控制的发展
6.9 传输层协议的发展

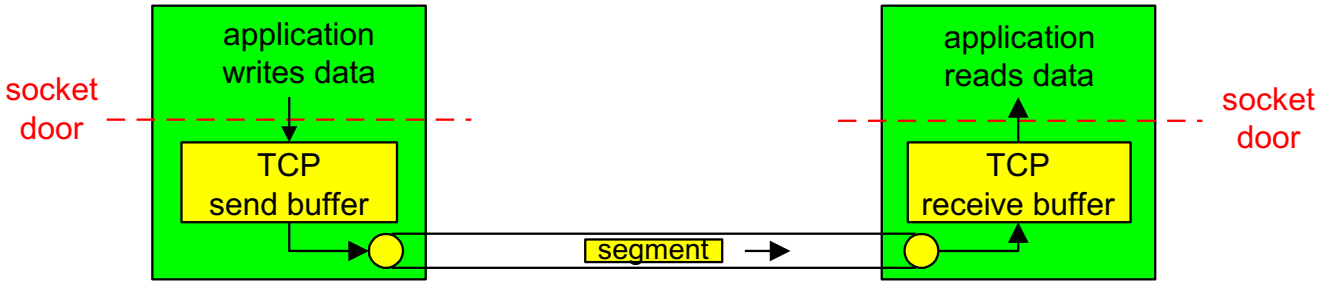
1. TCP概述
2. TCP报文段结构
3. TCP可靠数据传输
4. TCP流量控制
5. TCP连接管理

TCP 概述

- TCP服务模型：
  - 在一对通信的进程之间提供一条理想的字节流管道
- 点到点通信:
  - 仅涉及一对通信进程
- 全双工：
  - 可以同时双向传输数据
- 可靠、有序的字节流：
  - 不保留报文边界

需要的机制

- 建立连接:
  - 通信双方为本次通信建立数据传输所需的状态（套接字、缓存、变量等）
- 可靠数据传输：
  - 流水线式发送，报文段检错，丢失重传
- 流量控制:
  - 发送方不会令接收方缓存溢出

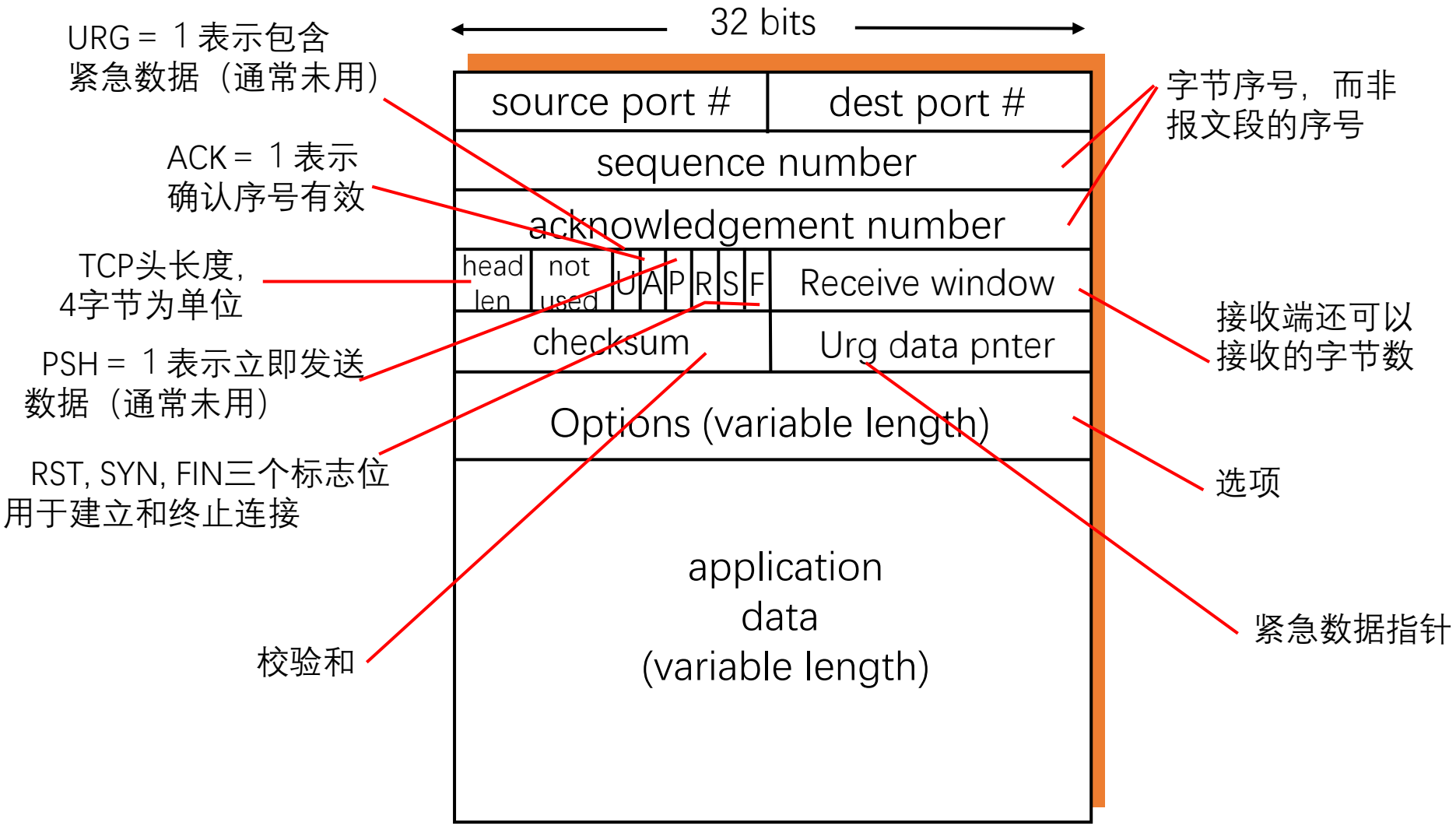


本章内容

6.1 概述和传输层服务
6.2 套接字编程
6.3 传输层复用和分用
6.4 无连接传输：UDP
6.5 面向连接的传输：TCP
6.6 理解网络拥塞
6.7 TCP拥塞控制
6.8 拥塞控制的发展
6.9 传输层协议的发展

1. TCP概述
2. TCP报文段结构
3. TCP可靠数据传输
4. TCP流量控制
5. TCP连接管理

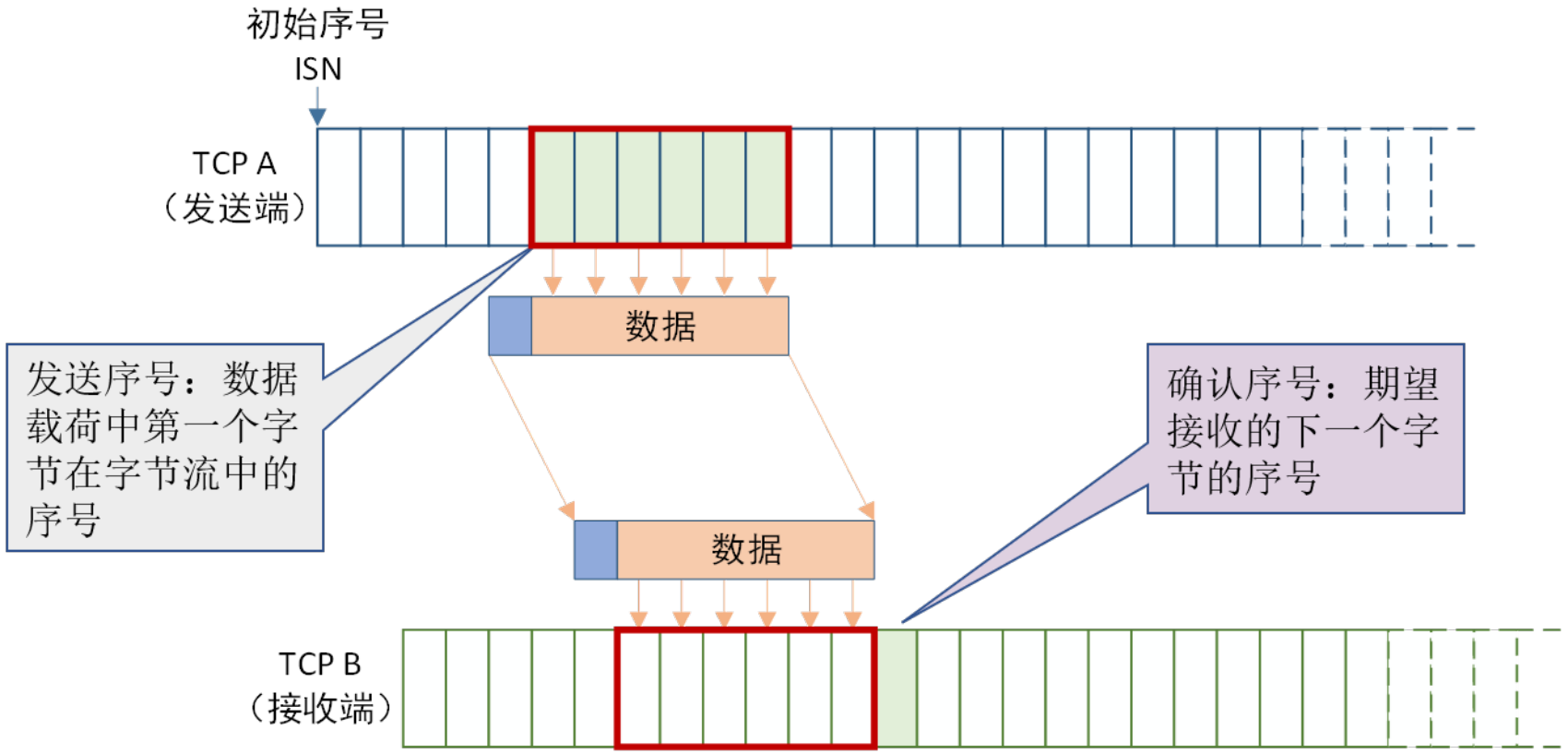
TCP报文段结构


32 bits
source port #
dest port #
sequence number
acknowledgement number
head len
not used
U
A
P
R
S
F
Receive window
checksum
Urg data pnter
Options (variable length)
application data (variable length)
URG = 1 表示包含紧急数据（通常未用）
ACK = 1 表示确认序号有效
TCP头长度，4字节为单位
PSH = 1 表示立即发送数据（通常未用）
RST, SYN, FIN三个标志位用于建立和终止连接
校验和
字节序号，而非报文段的序号
接收端还可以接收的字节数
选项
紧急数据指针

重要的TCP选项

- 最大段长度（MSS）：
  - TCP段中可以携带的最大数据字节数
  - 建立连接时，每个主机可声明自己能够接受的MSS，缺省为536字节
- 选择确认（SACK）：
  - 最初的TCP协议只使用累积确认
  - 改进的TCP协议引入选择确认，允许接收端指出缺失的数据字节

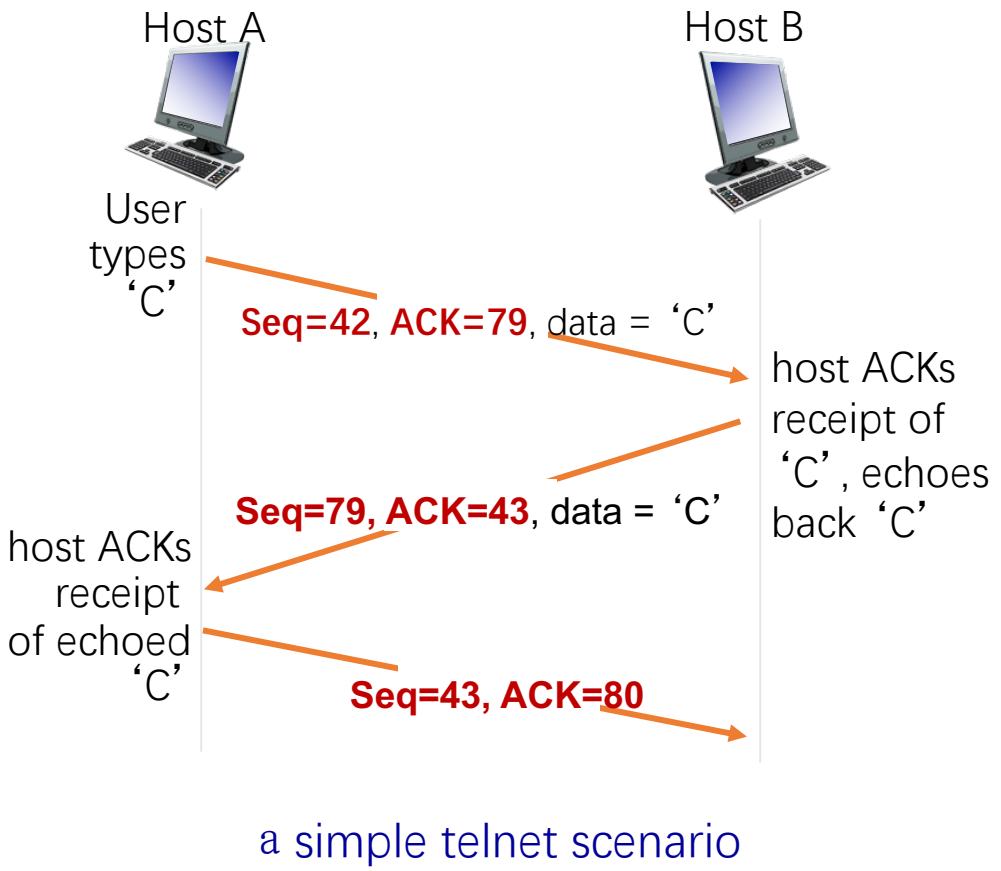
发送序号和确认序号的含义



初始序号的选取：每个TCP实体维护一个32位计数器，该计数器每4微秒增1，建立连接时从中读取计数器当前值（依赖具体实现，见连接管理）

发送序号和确认序号：使用举例

- 主机A向主机B发送仅包含一个字符‘C’的报文段：
  - 发送序号为42
  - 确认序号为79（对前一次数据的确认）
- 主机B将字符‘C’回送给主机A：
  - 发送序号为79
  - 确认序号为43（对收到‘C’的确认）
- 主机A向主机B发送确认报文段（不包含数据）：
  - 确认序号为80（对收到‘C’的确认）



a simple telnet scenario

本章内容

6.1 概述和传输层服务
6.2 套接字编程
6.3 传输层复用和分用
6.4 无连接传输：UDP
6.5 面向连接的传输：TCP
6.6 理解网络拥塞
6.7 TCP拥塞控制
6.8 拥塞控制的发展
6.9 传输层协议的发展

1. TCP概述
2. TCP报文段结构
3. TCP可靠数据传输
4. TCP流量控制
5. TCP连接管理

TCP可靠数据传输概述

- TCP 在不可靠的IP服务上建立可靠的数据传输
- 基本机制
  - 发送端：流水线式发送数据、等待确认、超时重传
  - 接收端：进行差错检测，采用累积确认机制
- 乱序段处理：协议没有明确规定
  - 接收端不缓存：可以正常工作，处理简单，但效率低
  - 接收端缓存：效率高，但处理复杂

一个高度简化的TCP协议

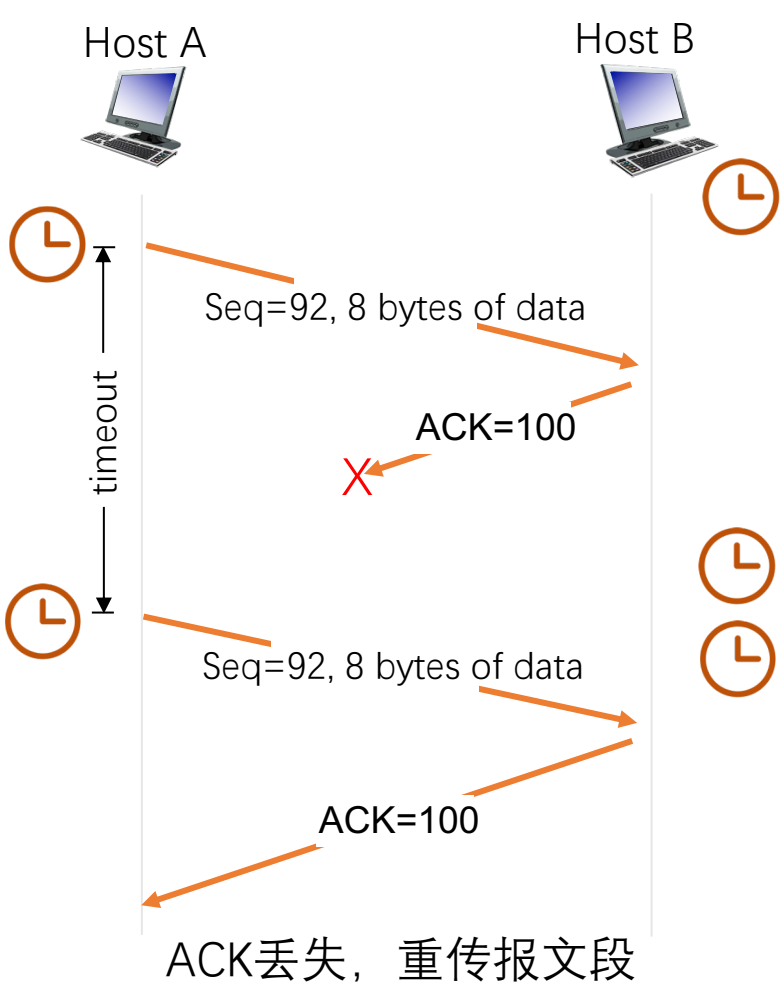
高度简化的TCP协议：仅考虑可靠传输机制，且数据仅在一个方向上传输

- 接收方：
  - 确认方式：采用累积确认，仅在正确、按序收到报文段后，更新确认序号；其余情况，重复前一次的确认序号（与回退N协议类似）
  - 失序报文段处理：缓存失序的报文段（与选择重传协议类似）
- 发送方：
  - 发送策略：流水线式发送报文段
  - 定时器的使用：仅对最早未确认的报文段使用一个重传定时器（与回退N协议类似）
  - 重发策略：仅在超时后重发最早未确认的报文段（与选择重传协议类似，因为接收端缓存了失序的报文段）

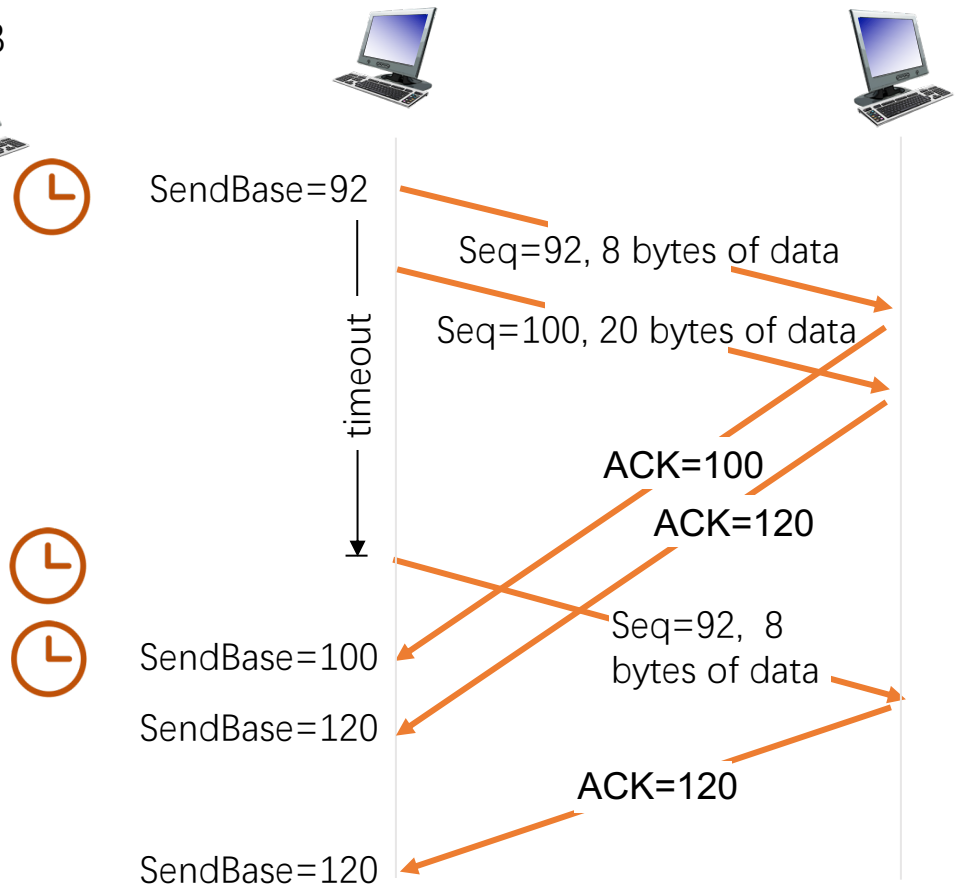
TCP发送方要处理的事件

- 收到应用数据：
  - 创建并发送TCP报文段
  - 若当前没有定时器在运行（没有已发送、未确认的报文段），启动定时器
- 超时:
  - 重传包含最小序号的、未确认的报文段
  - 重启定时器
- 收到ACK：
  - 如果确认序号大于基序号（已发送未确认的最小序号）：
    - 推进发送窗口（更新基序号）
    - 如果发送窗口中还有未确认的报文段，启动定时器，否则终止定时器

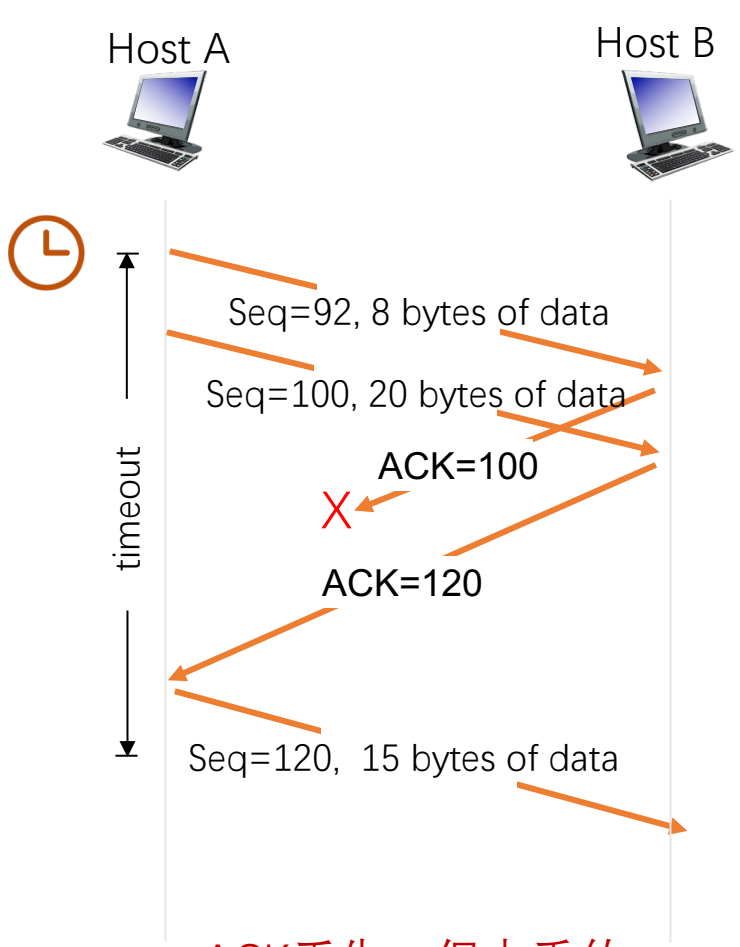
TCP可能的重传场景



ACK丢失，重传报文段



过早超时，仅第一个报文段重传



ACK丢失，但未重传

TCP可靠数据传输概述

- 思考以下问题：
  - 第二种情形，如果TCP像选择重传协议一样，每个报文段使用一个定时器，会怎么样？
  - 第三种情形，采用流水式发送和累积确认，可以避免重发哪些报文段？
- 可见，TCP通过采用以下机制减少了不必要的重传：
  - 只使用一个定时器，避免了超时设置过小时重发大量报文段
  - 利用流水式发送和累积确认，可以避免重发某些丢失了ACK的报文段

TCP的接收端

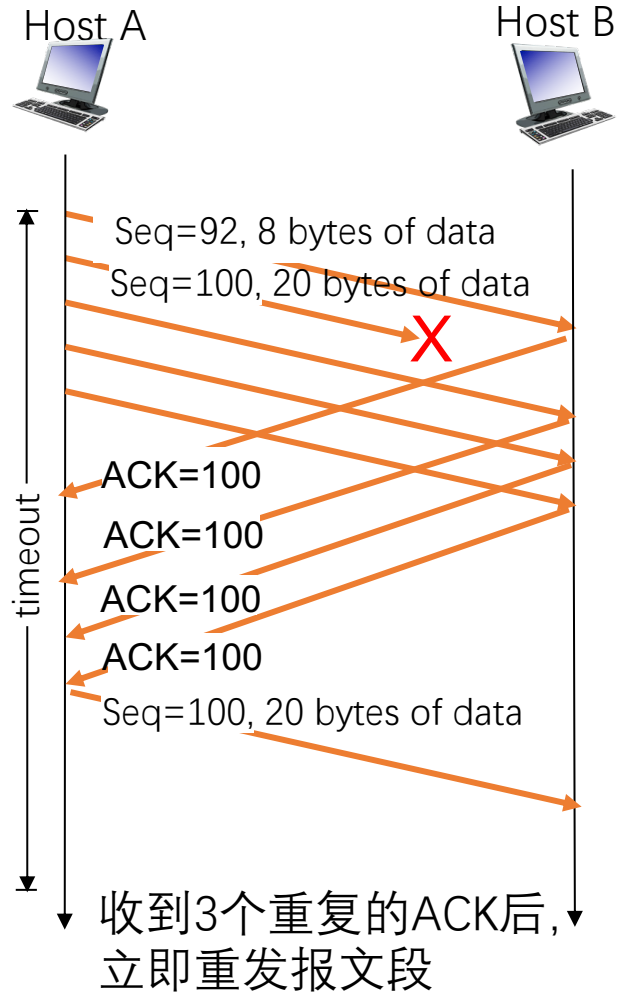
- 理论上，接收端只需区分两种情况：
  - 收到期待的报文段：发送更新的确认序号
  - 其它情况：重复当前的确认序号
- 为减小通信量，TCP允许接收端推迟确认：
  - 接收端可以在收到若干个报文段后，发送一个累积确认的报文段
- 推迟确认带来的问题：
  - 若延迟太大，会导致不必要的重传
  - 推迟确认造成RTT估计不准确
- TCP协议规定：
  - 推迟确认的时间最多为500ms
  - 接收方至少每隔一个报文段使用正常方式进行确认

TCP接收端的事件和处理

接收端事件	接收端动作
收到一个期待的报文段，且之前的报文段均已发送过确认	推迟发送确认，在500ms时间内若无下一个报文段到来，发送确认
收到一个期待的报文段，且前一个报文段被推迟确认	立即发送确认（估计RTT的需要）
收到一个失序的报文段（序号大于期待的序号），检测到序号间隙	立即发送确认（快速重传的需要），重复当前的确认序号
收到部分或全部填充间隙的报文段	若报文段始于间隙的低端，立即发送确认（推进发送窗口），更新确认序号

快速重传

- 仅靠超时重发丢失的报文段，恢复太慢！
- 发送方可利用重复ACK检测报文段丢失：
  - 发送方通常连续发送许多报文段
  - 若仅有个别报文段丢失，发送方将收到多个重复序号的ACK
  - 多数情况下IP按序交付分组，重复ACK极有可能因丢包产生
- TCP协议规定：
  - 当发送方收到对同一序号的3次重复确认时，立即重发包含该序号的报文段
- 快速重传:
  - 所谓快速重传，就是在定时器到期前重发丢失的报文段



小结

- TCP可靠传输的设计要点：
  - 流水式发送报文段
  - 缓存失序的报文段
  - 采用累积确认
  - 只对最早未确认的报文段使用一个重传定时器
  - 超时后只重传包含最小序号的、未确认的报文段
- 以上措施可大量减少因ACK丢失、定时器过早超时引起的重传
- 超时值的确定：
  - 基于RTT估计超时值+ 定时器补偿策略
- 测量RTT：
  - 不对重传的报文段测量RTT
  - 不连续使用推迟确认
- 快速重传：
  - 收到3次重复确认，重发报文段

本章内容

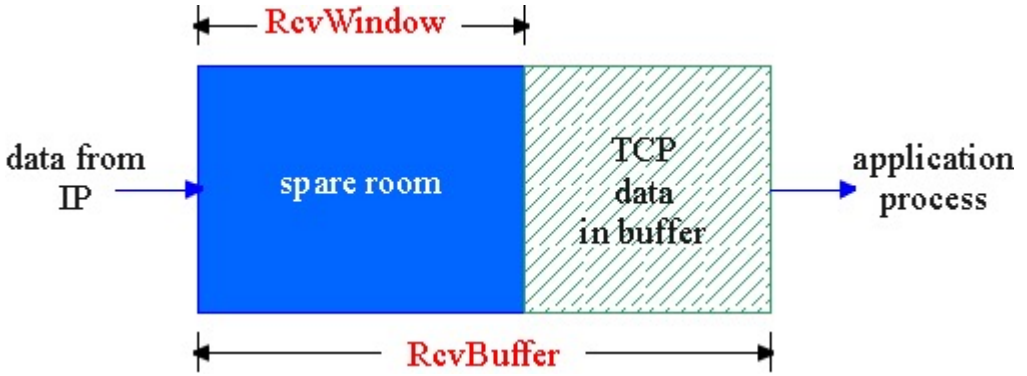
6.1 概述和传输层服务
6.2 套接字编程
6.3 传输层复用和分用
6.4 无连接传输：UDP
6.5 面向连接的传输：TCP
6.6 理解网络拥塞
6.7 TCP拥塞控制
6.8 拥塞控制的发展
6.9 传输层协议的发展

1. TCP概述
2. TCP报文段结构
3. TCP可靠数据传输
4. TCP流量控制
5. TCP连接管理

TCP的接收端：接收缓存

➢TCP接收端有一个接收缓存：

- 接收端TCP将收到的数据放入接收缓存
- 应用进程从接收缓存中读数据
- 进入接收缓存的数据不一定被立即取走、取完
- 如果接收缓存中的数据未及时取走，后续到达的数据可能会因缓存溢出而丢失



➢流量控制：

- 发送端TCP通过调节发送速率，不使接收端缓存溢出

为什么回退N、选择重传和UDP不需要流量控制

- 回退N协议和选择重传协议均假设：
  - 正确、按序到达的分组被立即交付给上层
  - 其占用的缓冲区被立即释放
- 发送方根据确认序号即可知道：
  - 哪些分组已被移出接收窗口
  - 接收窗口还可以接受多少分组
- 因此，回退N和选择重传不需要流量控制

- UDP不保证交付：
  - 接收端UDP将收到的报文载荷放入接收缓存
  - 应用进程每次从接收缓存中读取一个完整的报文载荷
  - 当应用进程消费数据不够快时，接收缓存溢出，报文数据丢失，UDP不负责任
- 因此，UDP不需要流量控制

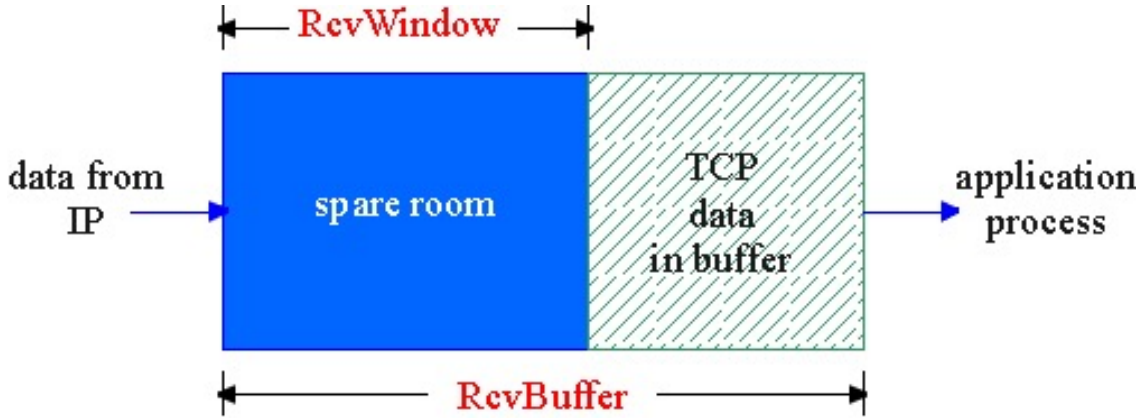
TCP如何进行流量控制

- 接收缓存中的可用空间称为接收窗口：

RcvWindow = RcvBuffer-[LastByteRcvd - LastByteRead]

- 接收方将RcvWindow放在报头中，向发送方通告接收缓存的可用空间
- 发送方限制已发送、未确认的字节数不超过接收窗口的大小，即：

LastByteSent-LastByteAcked ≦ RcvWindow



- 特别是，当接收方通告接收窗口为0时，发送方必须停止发送

非零窗口通告

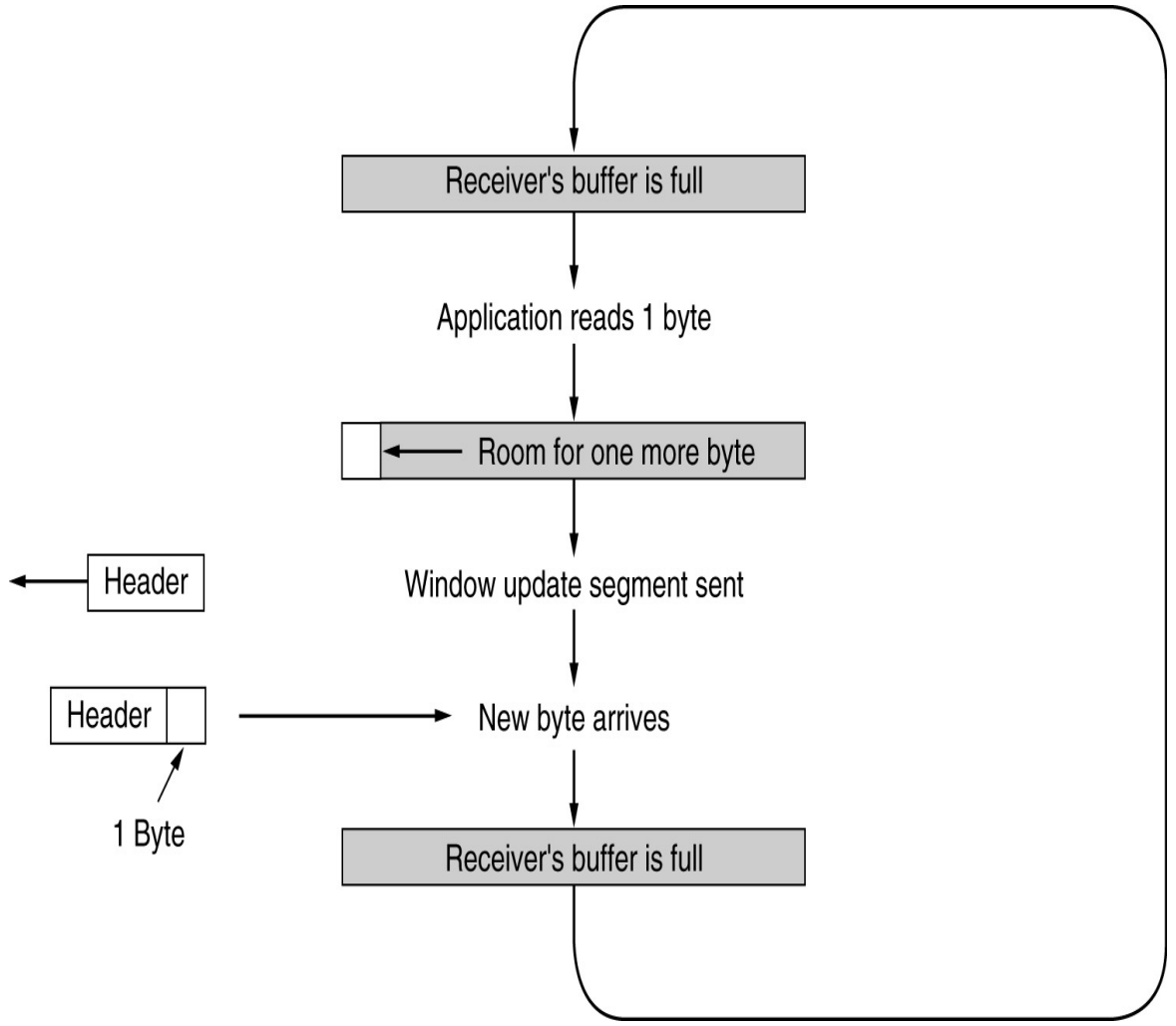
- 发送方/接收方对零窗口的处理：
  - 发送方：当接收窗口为0时，发送方必须停止发送
  - 接收方：当接收窗口变为非0时，接收方应通告增大的接收窗口
- 在TCP协议中，触发一次TCP传输需要满足以下三个条件之一：
  - 应用程序调用
  - 超时
  - 收到数据或确认
- 对于单向传输中的接收方，只有第三个条件能触发传输
- 当发送方停止发送后，接收方不再收到数据，如何触发接收端发送“非零窗口通告”呢？
- TCP协议规定：
  - 发送方收到“零窗口通告”后，可以发送“零窗口探测”报文段
  - 从而接收方可以发送包含接收窗口的响应报文段

零窗口探测的实现

- 发送端收到零窗口通告时，启动一个坚持定时器
- 定时器超时后，发送端发送一个零窗口探测报文段（序号为上一个段中最后一个字节的序号）
- 接收端在响应的报文段中通告当前接收窗口的大小
- 若发送端仍收到零窗口通告，重新启动坚持定时器

糊涂窗口综合症

- 当数据的发送速度很快、而消费速度很慢时，零窗口探测的简单实现带来以下问题：
  - 接收方不断发送微小窗口通告
  - 发送方不断发送很小的数据分组
  - 大量带宽被浪费
- 解决方案：
  - 接收方启发式策略
  - 发送方启发式策略



接收方启发式策略

- 接收端避免糊涂窗口综合症的策略：
  - 通告零窗口之后，仅当窗口大小显著增加之后才发送更新的窗口通告
  - 什么是显著增加：窗口大小达到缓存空间的一半或者一个MSS，取两者的较小值
- TCP执行该策略的做法：
  - 当窗口大小不满足以上策略时，推迟发送确认（但最多推迟500ms，且至少每隔一个报文段使用正常方式进行确认），寄希望于推迟间隔内有更多数据被消费
  - 仅当窗口大小满足以上策略时，才通告新的窗口大小

发送方启发式策略

- 发送方避免糊涂窗口综合症的策略：
  - 发送方应积聚足够多的数据再发送，以防止发送太短的报文段
- 问题：发送方应等待多长时间？
  - 若等待时间不够，报文段会太短
  - 若等待时间过久，应用程序的时延会太长
  - 更重要的是，TCP不知道应用程序会不会在最近的将来生成更多的数据
- Nagle算法的解决方法：
  - 在新建连接上，当应用数据到来时，组成一个TCP段发送（那怕只有一个字节）
  - 在收到确认之前，后续到来的数据放在发送缓存中
  - 当数据量达到一个MSS或上一次传输的确认到来（取两者的较小时间），用一个TCP段将缓存的字节全部发走
- Nagle算法的优点：
  - 适应网络延时、MSS长度及应用速度的各种组合
  - 常规情况下不会降低网络的吞吐量

TCP流量控制小结

- TCP接收端
  - 使用显式的窗口通告，告知发送方可用的缓存空间大小
  - 在接收窗口较小时，推迟发送确认
  - 仅当接收窗口显著增加时，通告新的窗口大小
- TCP发送端
  - 使用Nagle算法确定发送时机
  - 使用接收窗口限制发送的数据量，已发送未确认的字节数不超过接收窗口的大小

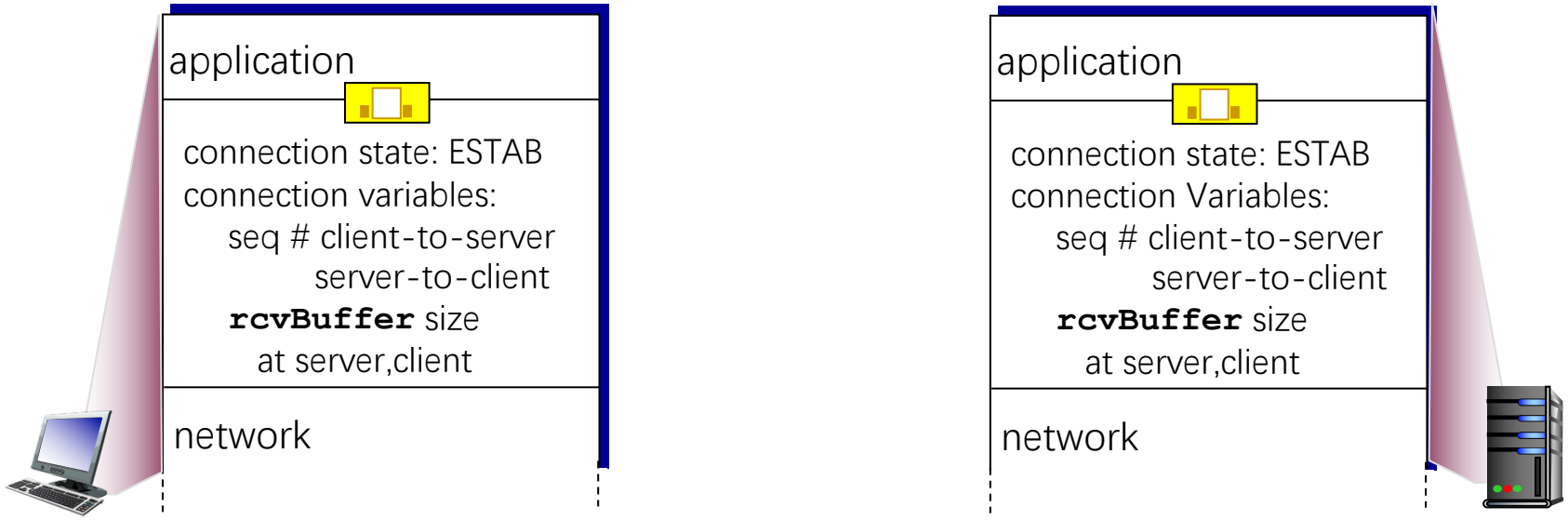
本章内容

6.1 概述和传输层服务
6.2 套接字编程
6.3 传输层复用和分用
6.4 无连接传输：UDP
6.5 面向连接的传输：TCP
6.6 理解网络拥塞
6.7 TCP拥塞控制
6.8 拥塞控制的发展
6.9 传输层协议的发展

1. TCP概述
2. TCP报文段结构
3. TCP可靠数据传输
4. TCP流量控制
5. TCP连接管理

建立TCP连接

- 建立一条TCP连接需要确定两件事：
  - 双方都同意建立连接（知晓另一方想建立连接）
  - 初始化连接参数（序号，MSS等）

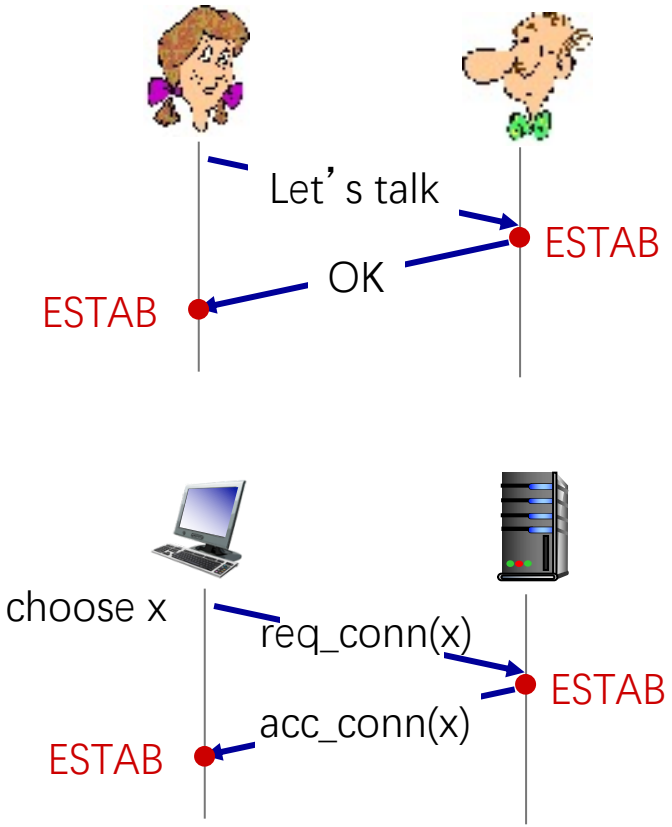



```
Socket clientSocket =
  newSocket("hostname","port
  number");
```

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

两次握手建立连接

两次握手建立连接的类比



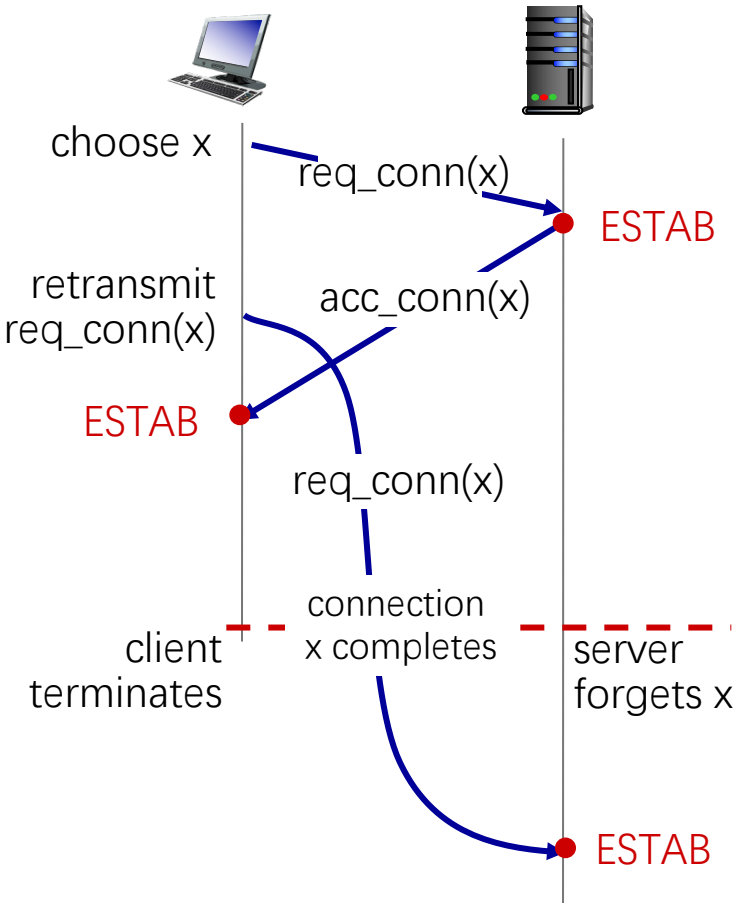
➢问题:

- 在网络中，2次握手总是可行的吗？

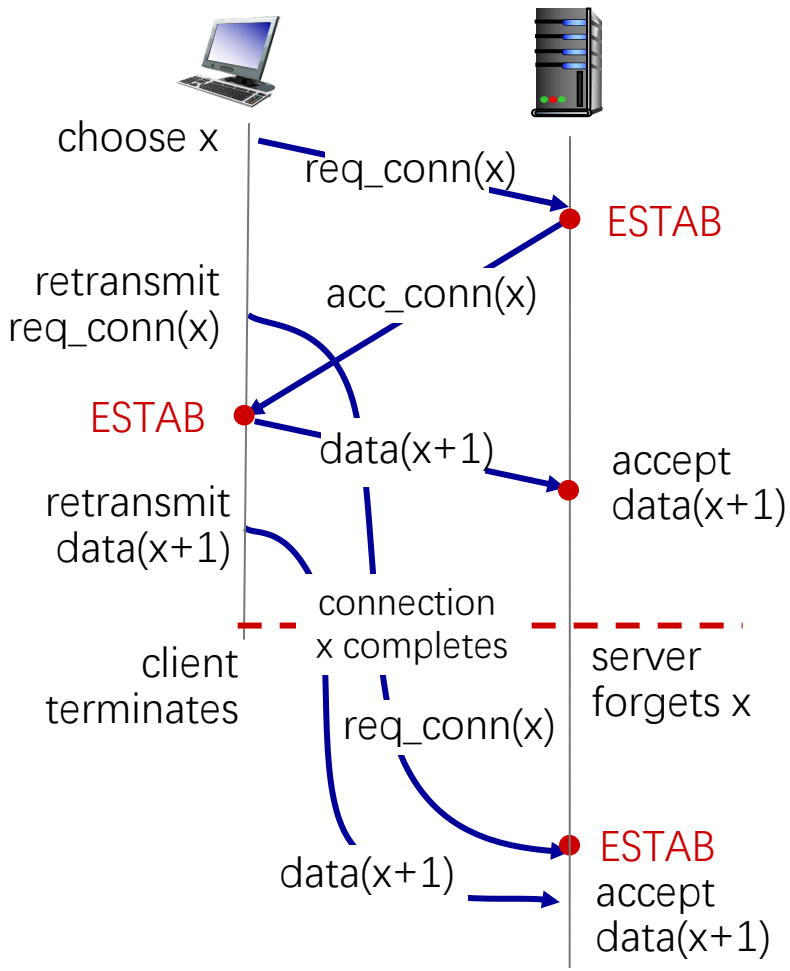
➢在一个不可靠的网络中，总会有一些意外发生：

- 报文传输延迟变化很大
- 存在重传的报文段
- 存在报文重排序

两次握手失败的情形



形成半开连接（没有client）



形成半开连接（没有client）

TCP三次握手建立连接

客户状态

clientSocket = socket(AF_INET, SOCK_STREAM)

LISTEN

clientSocket.connect((serverName,serverPort))

SYNSENT

ESTAB

服务器状态

serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('' ,serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()

LISTEN

SYN RCVD

ESTAB

choose init seq num, x
send TCP SYN msg

SYNbit=1, Seq=x

choose init seq num, y
send TCP SYNACK
msg, acking SYN

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

ACKbit=1, ACKnum=y+1

received ACK(y)
indicates client is live

TCP三次握手建立连接

1. 客户TCP发送SYN 报文段（SYN=1, ACK=0）
   - 给出客户选择的起始序号
   - 不包含数据
2. 服务器TCP发送SYNACK报文段（SYN=ACK=1）（服务器端分配缓存和变量）
   - 给出服务器选择的起始序号
   - 确认客户的起始序号
   - 不包含数据
3. 客户发送ACK报文段（SYN=0，ACK=1）（客户端分配缓存和变量）
   - 确认服务器的起始序号
   - 可能包含数据

如何选择起始序号

➢为什么起始序号不从0开始？

- 若在不同的时间、在同一对套接字之间建立了连接，则新、旧连接上的序号有重叠，旧连接上重传的报文段会被误以为是新连接上的报文段

➢可以随机选取起始序号吗？

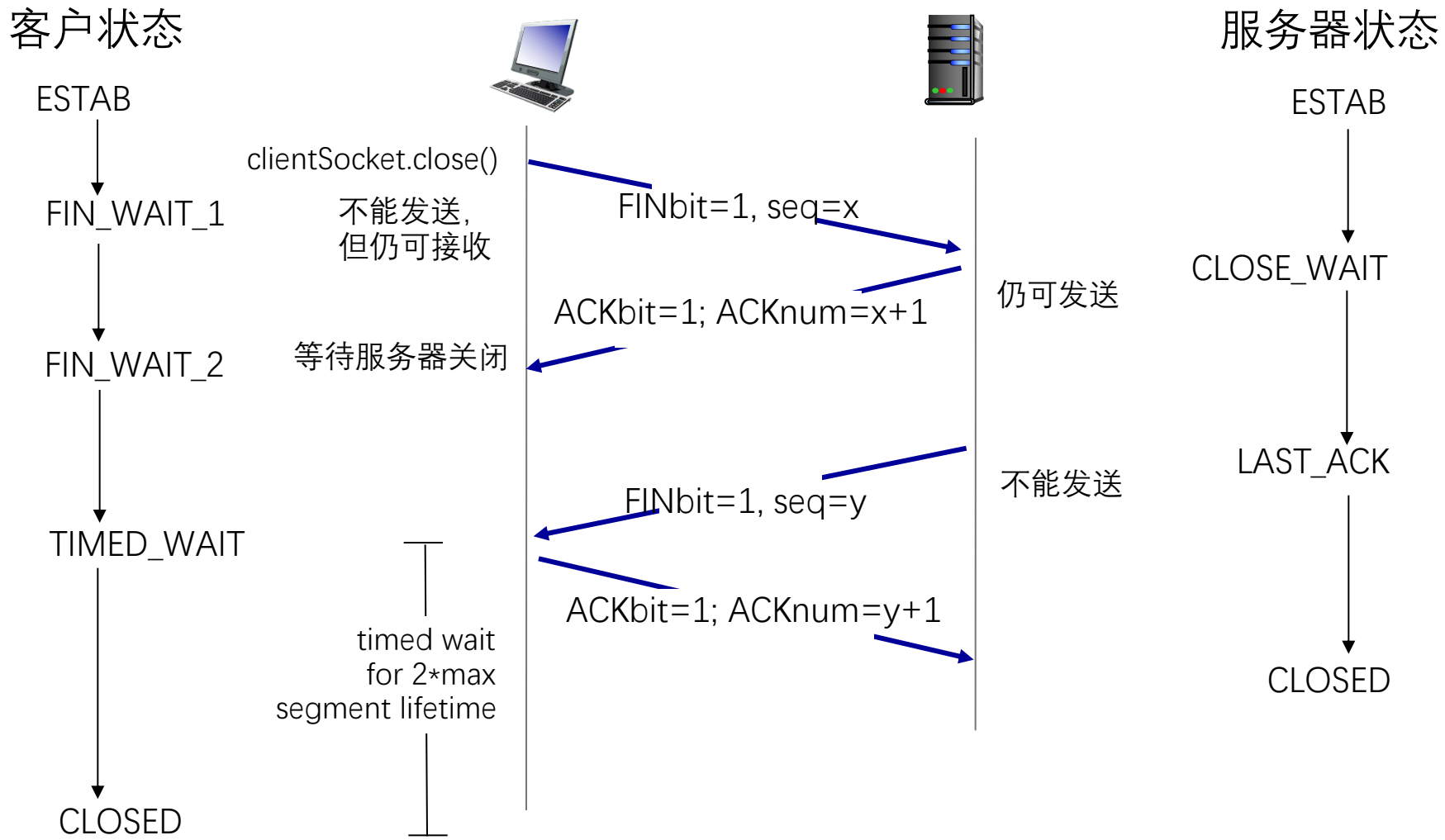
- 若在不同的时间、在同一对套接字之间建立了连接，且新、旧连接上选择的起始序号x和y相差不大，那么新、旧连接上传输的序号仍然可能重叠

➢结论：必须避免新、旧连接上的序号产生重叠

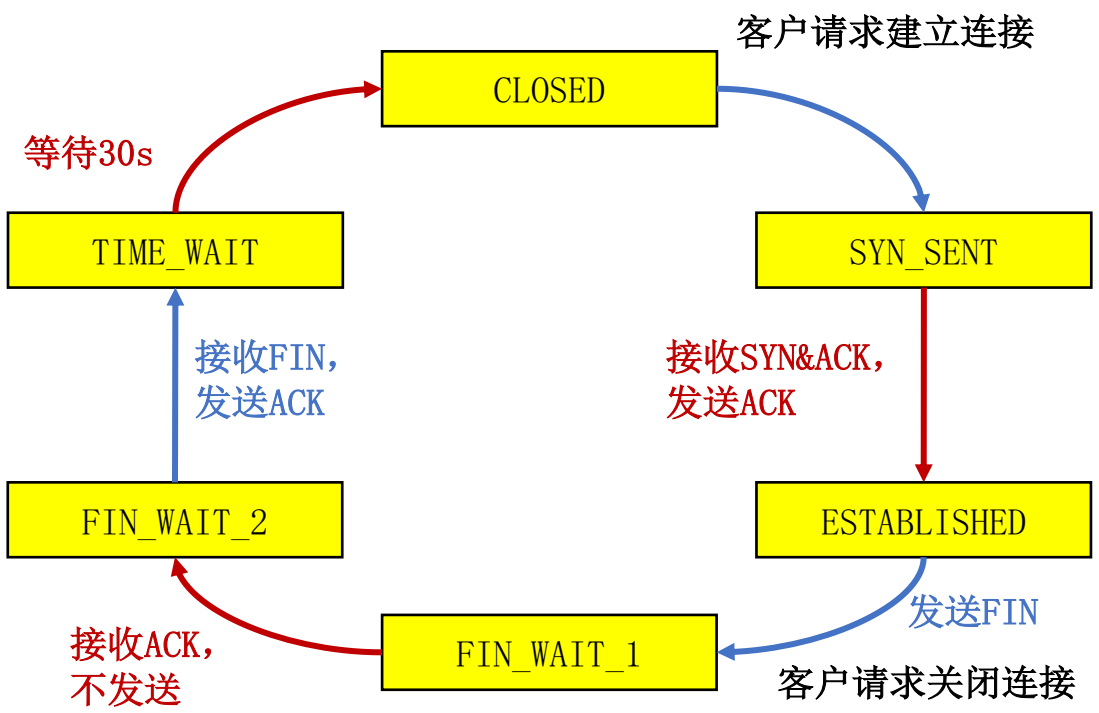
TCP起始序号的选择

- 基于时钟的起始序号选取算法：
  - 每个主机使用一个时钟，以二进制计数器的形式工作，每隔ΔT时间计数器加1
  - 新建一个连接时，以本地计数器值的最低32位作为起始序号
  - 该方法确保连接的起始序号随时间单调增长
- ΔT取较小的值（4微秒）：
  - 确保发送序号的增长速度，不会超过起始序号的增长速度
- 使用较长的字节序号（32位）：
  - 确保序号回绕的时间远大于分组在网络中的最长寿命

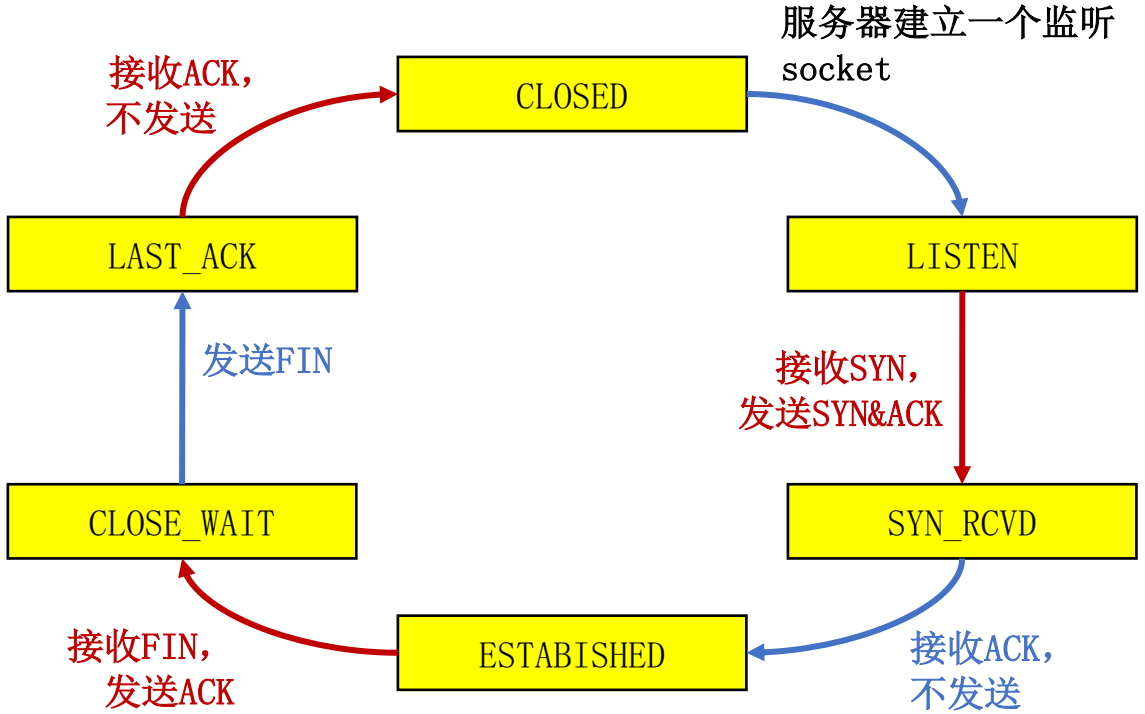
关闭TCP连接


客户状态
ESTAB
FIN_WAIT_1
FIN_WAIT_2
TIMED_WAIT
CLOSED
clientSocket.close()
不能发送，
但仍可接收
FINbit=1, seq=x
仍可发送
ACKbit=1; ACKnum=x+1
等待服务器关闭
FINbit=1, seq=y
不能发送
ACKbit=1; ACKnum=y+1
timed wait
for 2*max
segment lifetime
服务器状态
ESTAB
CLOSE_WAIT
LAST_ACK
CLOSED

客户/服务器经历的TCP状态序列



客户生命周期



服务器生命周期

SYN洪泛攻击

- TCP实现的问题：
  - 服务器在收到SYN段后，发送SYNACK段，分配资源
  - 若未收到ACK段，服务器超时后重发SYNACK段
  - 服务器等待一段时间（称SYN超时）后丢弃未完成的连接，SYN超时的典型值为30秒~120秒
- SYN洪泛攻击：
  - 攻击者采用伪造的源IP地址，向服务器发送大量的SYN段，却不发送ACK段
  - 服务器为维护一个巨大的半连接表耗尽资源，导致无法处理正常客户的连接请求，表现为服务器停止服务

本章内容

网络拥塞的类比例子：交通拥堵

交通拥堵:

- 起因：大量汽车短时间内进入路网，超出路网的承载能力
- 表现：道路通行能力下降，车速变慢，甚至完全停滞
- 措施：减少车辆进入路网（交通管制）

网络拥塞:

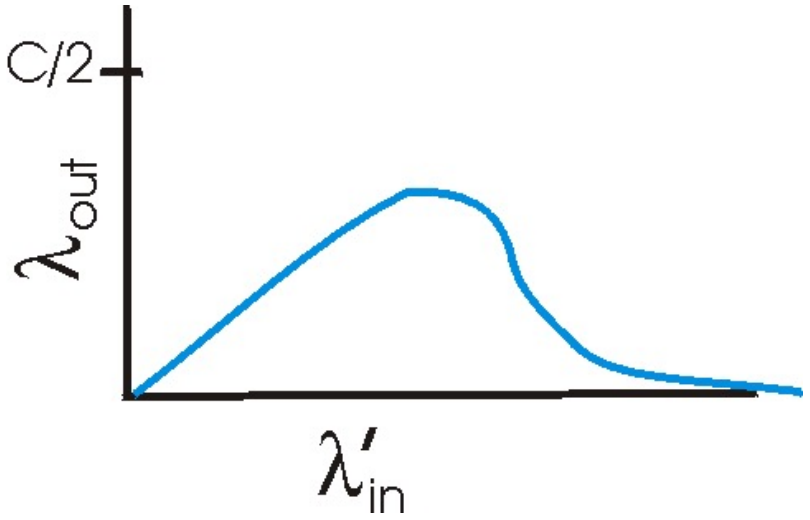
- 起因：大量分组短时间内进入网络，超出网络的处理能力
- 表现：分组延迟增大，网络吞吐量下降，甚至降为0
- 措施：减少分组进入网络（拥塞控制）

流量控制与拥塞控制的异同：

- 流量控制：限制发送速度，使不超过接收端的处理能力
- 拥塞控制：限制发送速度，使不超过网络的处理能力

网络拥塞的后果

- 网络拥塞造成：
  - 丢包：由路由器缓存溢出造成
  - 分组延迟增大：链路接近满载造成
- 大量网络资源用于：
  - 重传丢失的分组
  - （不必要地）重传延迟过大的分组
  - 转发最终被丢弃的分组
- 结果：
  - 进入网络的负载很重，网络吞吐量却很低



拥塞控制的常用方法

- 网络辅助的拥塞控制
  - 路由器向端系统提供显式的反馈，例如：
    - 设置拥塞指示比特
    - 给出发送速率指示
  - ATM、X.25采用此类方法
- 端到端拥塞控制
  - 网络层不向端系统提供反馈
  - 端系统通过观察丢包和延迟，自行推断拥塞的发生
  - TCP采用此类方法

本章内容

TCP拥塞控制要解决的问题

- TCP使用端到端拥塞控制机制：
  - 发送方根据自己感知的网络拥塞程度，限制其发送速率
- 需要回答三个问题：
  - 发送方如何感知网络拥塞？
  - 发送方采用什么机制来限制发送速率？
  - 发送方感知到网络拥塞后，采取什么策略调节发送速率？

拥塞检测和速率限制

发送方如何感知拥塞?

- 发送方利用丢包事件感知拥塞：
  - 拥塞造成丢包和分组延迟增大
  - 无论是丢包还是分组延迟过大，对于发送端来说都是丢包了
- 丢包事件包括：
  - 重传定时器超时
  - 发送端收到3个重复的ACK

发送方采用什么机制限制发送速率?

- 发送方使用拥塞窗口cwnd限制已发送未确认的数据量:

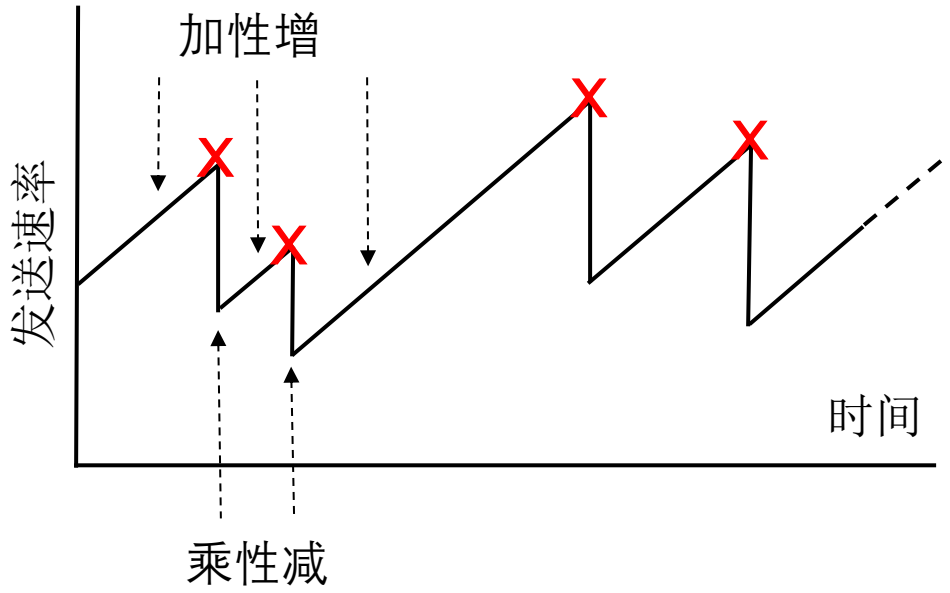
$$LastByteSent - LastByteAcked \le cwnd$$

$$rate = \frac{cwnd}{RTT} \quad Bytes/sec$$

- cwnd随发送方感知的网络拥塞程度而变化

拥塞窗口的调节策略：AIMD

- 乘性减（Multiplicative Decrease）
  - 发送方检测到丢包后，将cwnd的大小减半（但不能小于一个MSS）
  - 目的：迅速减小发送速率，缓解拥塞
- 加性增（Additive Increase）
  - 若无丢包，每经过一个RTT，将cwnd增大一个MSS，直到检测到丢包
  - 目的：缓慢增大发送速率，避免振荡



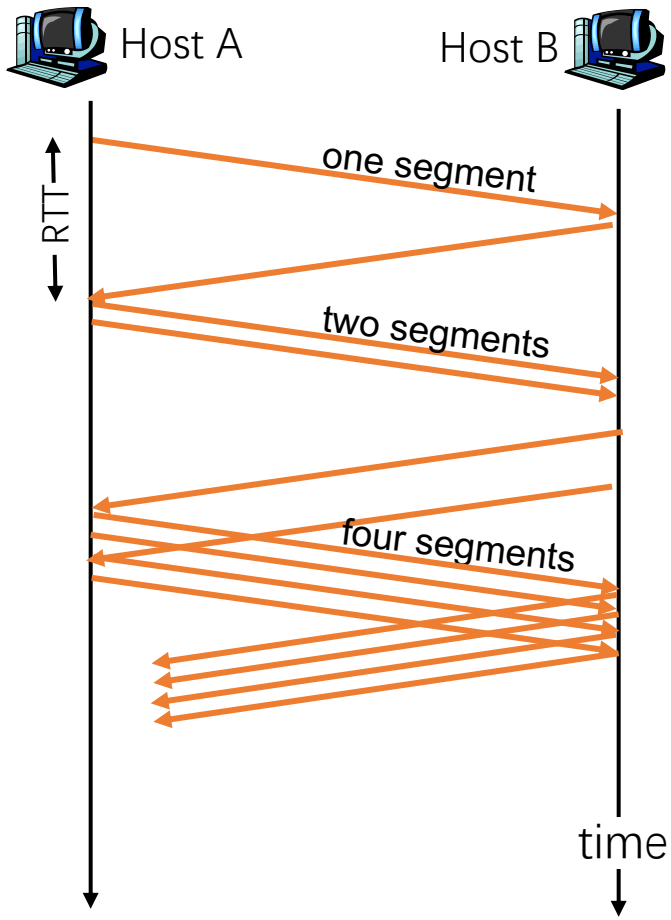
cwnd呈锯齿状变化

TCP慢启动

- 在新建的连接（或沉寂了一段时间的连接）上，以什么速率发送数据（此时接收窗口达最大值）？
- 早期的TCP协议：
  - 发送端仅以接收窗口大小限制发送速率，网络经常因为拥塞而崩溃！
- 采用“加性增”增大发送窗口，太慢！
  - 在新建连接上，令cwnd = 1 MSS，起始速度= MSS/RTT
  - 然而，网络中的可用带宽可能远大于MSS/RTT
- 慢启动的基本思想：
  - 在新建连接上指数增大cwnd，直至检测到丢包（此时终止慢启动）
  - 希望迅速增大cwnd至可用的发送速度

慢启动的实施

- 慢启动的策略：
  - 每经过一个RTT，将cwnd加倍
- 慢启动的具体实施：
  - 每收到一个ACK段，cwnd增加一个MSS
  - 只要发送窗口允许，发送端可以立即发送下一个报文段
- 特点：
  - 以一个很低的速率开始，按指数增大发送速率
- 慢启动比谁“慢”？
  - 与早期TCP按接收窗口发送数据的策略相比，采用慢启动后发送速率的增长较慢



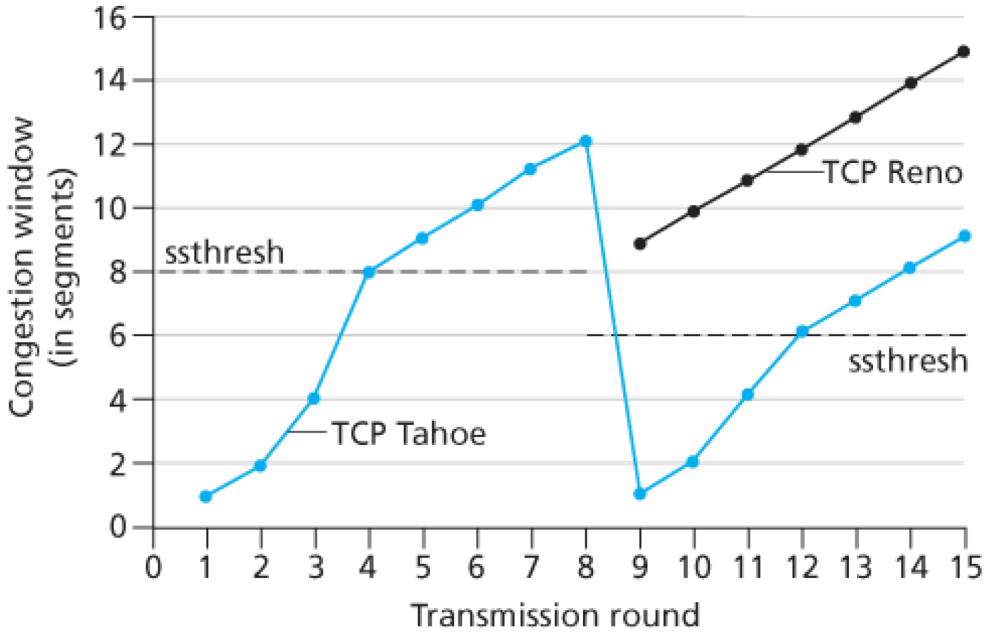
区分不同的丢包事件

- 超时和收到3个重复的ACK，它们反映出来的网络拥塞程度是一样的吗？ 当然不一样！
- 超时：说明网络交付能力很差
- 收到3个重复的ACK：说明网络仍有一定的交付能力
- 目前的TCP实现区分上述两种不同的丢包事件

- 收到3个重复的ACK：
  - 将cwnd降至一半
  - 使用AIMD调节cwnd
- 超时：
  - 设置门限 =cwnd/2
  - cwnd=1MSS
  - 使用慢启动增大cwnd至门限
  - 使用AIMD调节cwnd

TCP拥塞控制的实现

- 发送方维护变量ssthresh
- 发生丢包时，ssthresh=cwnd/2
- ssthresh是从慢启动转为拥塞避免的分水岭：
  - cwnd低于门限时，执行慢启动
  - cwnd高于门限：执行拥塞避免
- 拥塞避免阶段，拥塞窗口线性增长：
  - 每当收到ACK， cwnd=cwnd + MSS*(MSS/cwnd)

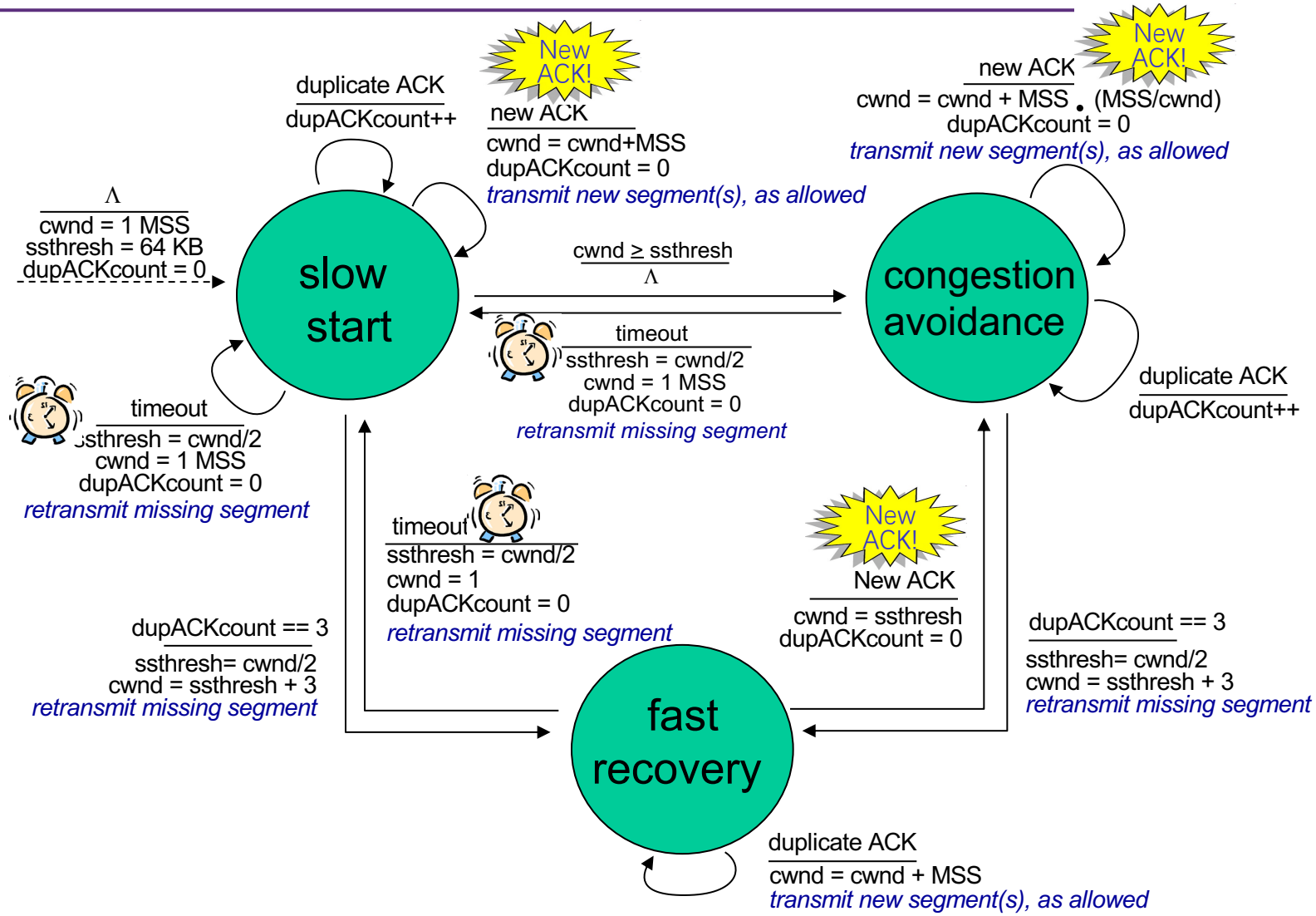


- 检测到3个重复的ACK后：
  - TCP Reno实现： cwnd= ssthresh+3，线性增长
  - TCP Tahoe实现：cwnd=1 MSS，慢启动

TCP发送端的事件与动作

State	Event	TCP Sender Action	Commentary
慢启动 (SS)	收到新的确认	cwnd = cwnd + MSS, If (cwnd > ssthresh) set state to “Congestion Avoidance”	每经过一个RTT，cwnd加倍
拥塞避免 (CA)	收到新的确认	cwnd = cwnd+MSS * (MSS/cwnd)	每经过一个RTT，cwnd增加一个MSS
SS or CA	收到3个重复的确认	ssthresh = cwnd/2, cwnd = Threshold+3, Set state to “Congestion Avoidance”	cwnd减半，然后线性增长
SS or CA	超时	ssthresh = cwnd/2, cwnd = 1 MSS, Set state to “Slow Start”	cwnd降为一个MSS，进入慢启动
SS or CA	收到一个重复的确认	统计收到的重复确认数	cwnd 和 ssthresh 都不变

TCP拥塞控制状态机


slow start
congestion avoidance
fast recovery
Λ
cwnd = 1 MSS
ssthresh = 64 KB
dupACKcount = 0
duplicate ACK
dupACKcount++
New ACK!
new ACK
cwnd = cwnd+MSS
dupACKcount = 0
transmit new segment(s), as allowed
cwnd ≥ ssthresh
Λ
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
retransmit missing segment
new ACK
cwnd = cwnd + MSS • (MSS/cwnd)
dupACKcount = 0
transmit new segment(s), as allowed
duplicate ACK
dupACKcount++
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
retransmit missing segment
timeout
ssthresh = cwnd/2
cwnd = 1
dupACKcount = 0
retransmit missing segment
New ACK
cwnd = ssthresh
dupACKcount = 0
dupACKcount == 3
ssthresh= cwnd/2
cwnd = ssthresh + 3
retransmit missing segment
dupACKcount == 3
ssthresh= cwnd/2
cwnd = ssthresh + 3
retransmit missing segment
duplicate ACK
cwnd = cwnd + MSS
transmit new segment(s), as allowed

TCP连接的吞吐量

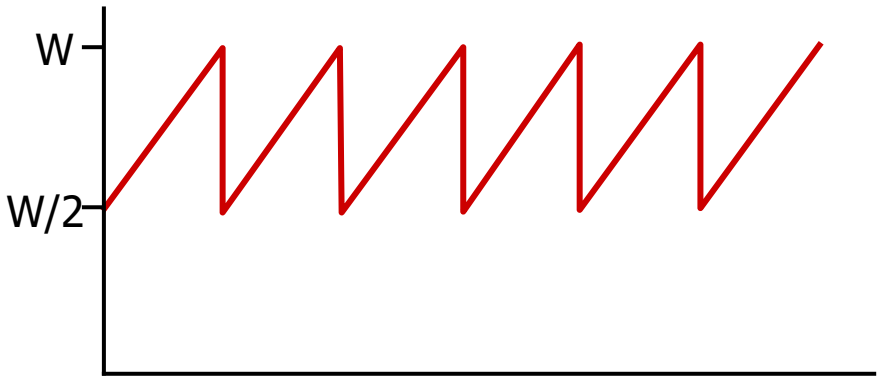
一个长期存活的TCP连接，平均吞吐量是多少？

- 忽略慢启动阶段（该阶段时间很短），只考虑拥塞避免阶段
- 令W=发生丢包时的拥塞窗口，此时有：

  throughput = W/RTT
- 发生丢包后调整 cwnd=W/2，此时有：

  throughput=W/2RTT
- 假设在TCP连接的生命期内，RTT 和 W几乎不变，有：

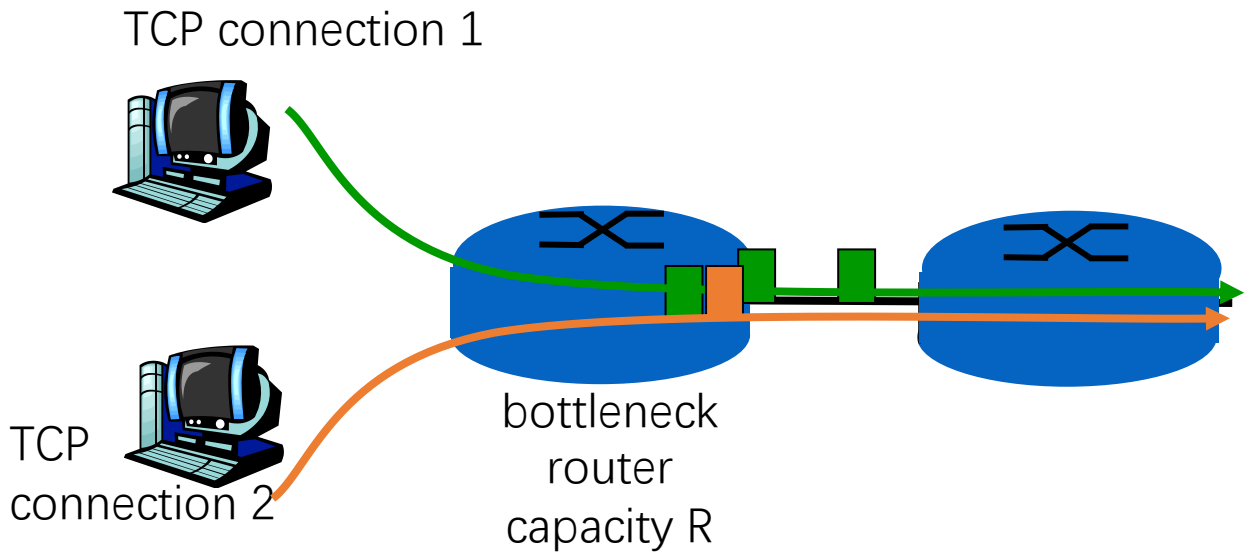
  Average throughout=0.75 W/RTT



TCP的公平性

➢公平性目标:

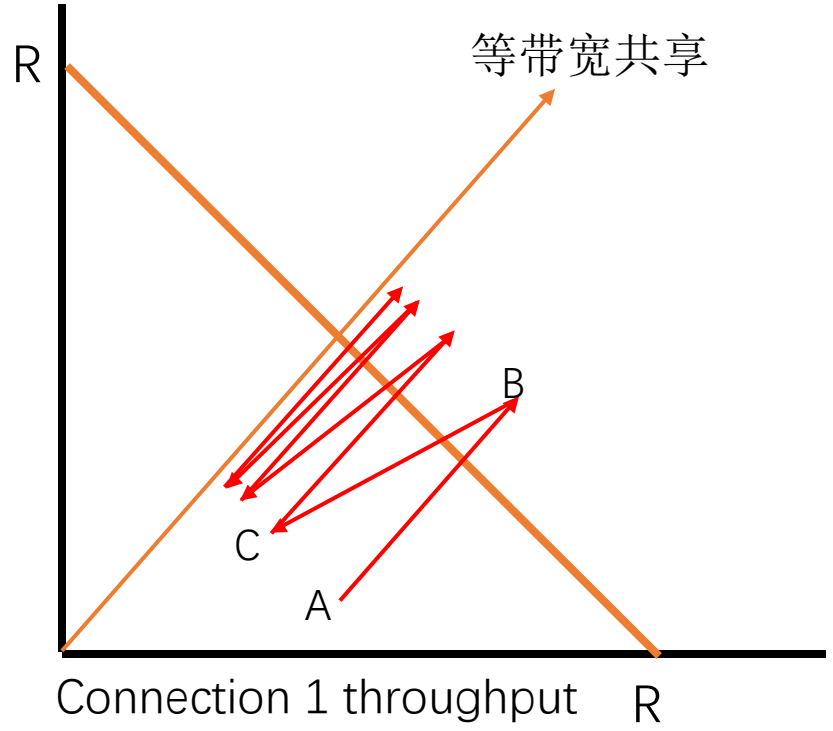
- 如果K条TCP连接共享某条带宽为R的瓶颈链路，每条连接应具有平均速度R/K



为什么TCP是公平的

➢考虑两条竞争的连接（各种参数相同）共享带宽为R的链路:

- 加性增：连接1和连接2按照相同的速率增大各自的拥塞窗口，得到斜率为1的直线
- 乘性减：连接1和连接2将各自的拥塞窗口减半



TCP公平性更复杂的情形

- 若相互竞争的TCP连接具有不同的参数（RTT、MSS等），不能保证公平性
- 若应用（如web）可以建立多条并行TCP连接，不能保证带宽在应用之间公平分配，比如：
  - 一条速率为R的链路上有9条连接
  - 若新应用建立一条TCP连接，获得速率 R/10
  - 若新应用建立11条TCP，可以获得速率 R/2！

关于TCP和UDP的一些思考

1. TCP提供可靠传输、流量控制、拥塞控制，而UDP都没有，能否说TCP服务优于UDP服务？
2. 多媒体应用希望的传输层服务是：带宽有保证、延迟有保证、顺序有保证、但能忍受一些丢包，TCP或UDP能够满足多媒体应用的需求吗？
3. 现实中的多媒体应用，有的使用TCP、有的使用UDP，它们分别出于什么考虑选择采用TCP或UDP？
4. 如何阻止UDP流量压制TCP流量？