



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

立足航天，服务国防，面向国民经济主战场

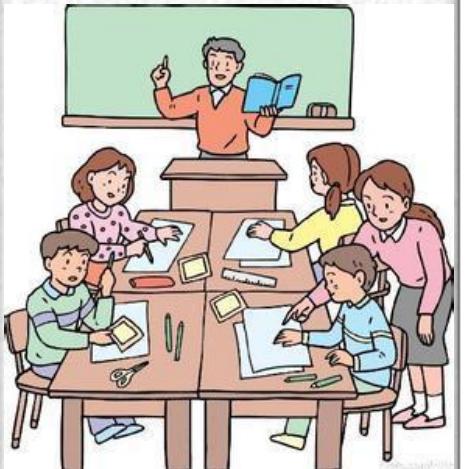
《计算机网络》

第5章 传输层



主要内容

本章学习目标



- ❖ 理解传输层服务
- ❖ 理解端到端原则
- ❖ 掌握传输层复用/分解方法
- ❖ 掌握UDP协议
- ❖ 掌握TCP协议
 - TCP协议特点
 - TCP段结构
 - TCP可靠数据传输
 - TCP流量控制
 - TCP连接控制
 - TCP拥塞控制
 - TCP公平性

主要内容

- ❖ 5.1 传输层服务
- ❖ 5.2 传输层多路复用/分用
- ❖ 5.3 UDP协议
- ❖ 5.4 TCP协议
 - 5.4.1 TCP段结构
 - 5.4.2 TCP可靠数据传输
 - 5.4.3 TCP流量控制
 - 5.4.4 TCP连接控制
 - 5.4.5 TCP拥塞控制
 - 5.4.6 TCP性能





哈爾濱工業大學

HARBIN INSTITUTE OF TECHNOLOGY

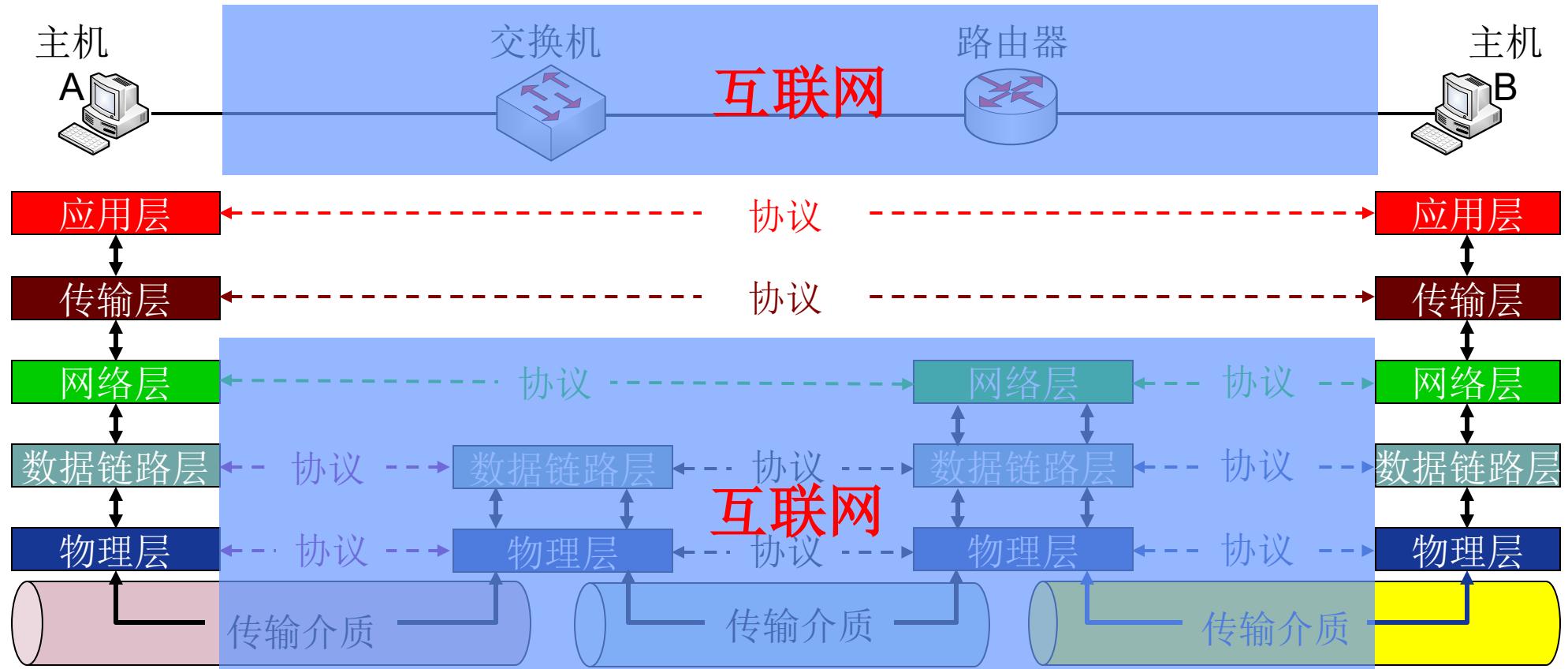
立足航天，服务国防，面向国民经济主战场

5.1 传输层服务



传输层？

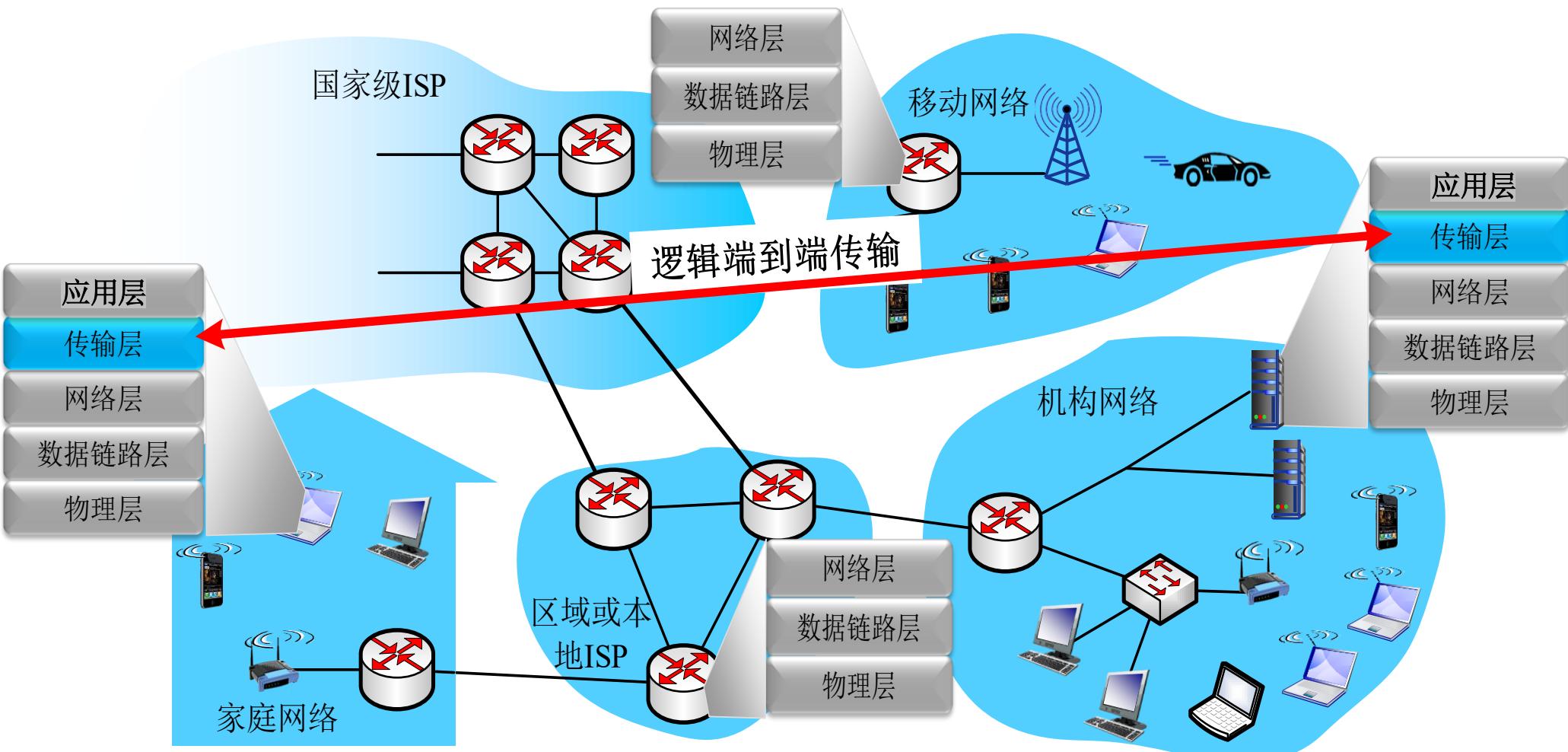
5.1 传输层服务





传输层？

5.1 传输层服务

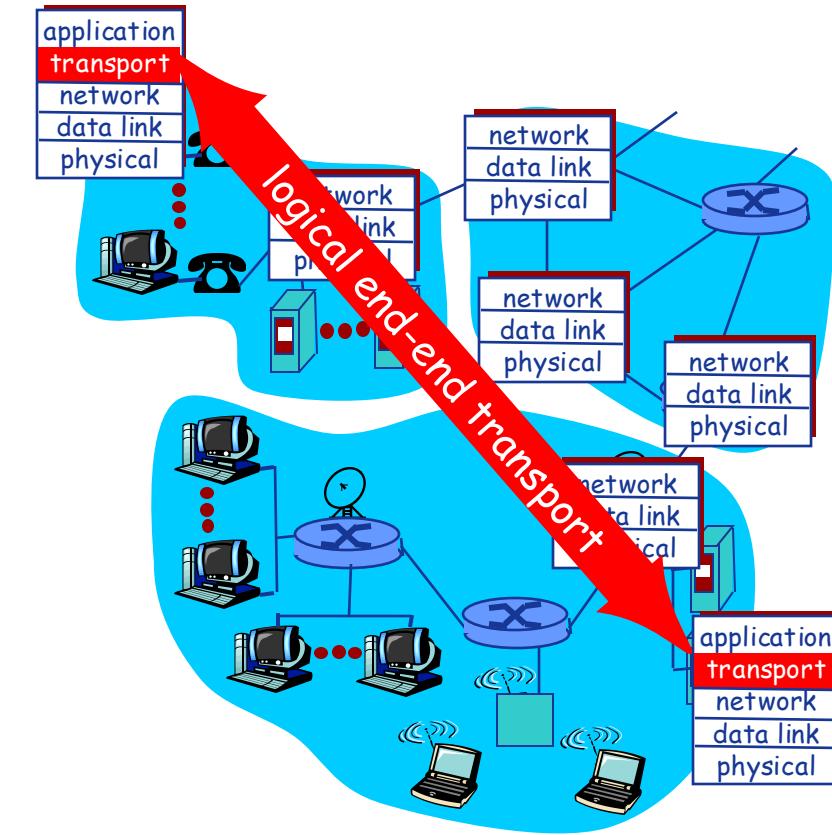




传输层服务和协议

5.1 传输层服务

- ❖ 传输层协议为运行在不同Host上的进程提供了一种逻辑通信机制
- ❖ 端系统运行传输层协议
 - **发送方**: 将应用递交的消息分成一个或多个的Segment，并向下传给网络层。
 - **接收方**: 将接收到的segment组装成消息，并向上交给应用层。
- ❖ 传输层可以为应用提供多种协议
 - Internet的TCP
 - Internet的UDP





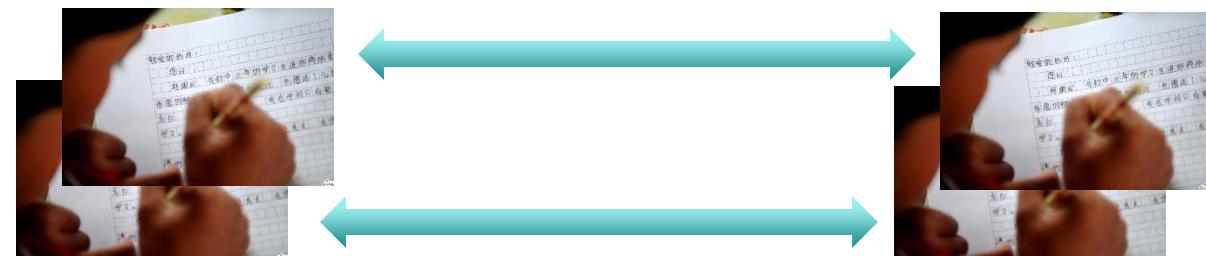
传输层 VS. 网络层

5.1 传输层服务

- ❖ 网络层：提供**主机**之间的逻辑通信机制
- ❖ 传输层：提供**应用进程**之间的逻辑通信机制
 - 位于网络层之上
 - 依赖于网络层服务
 - 对网络层服务进行（可能的）增强

家庭类比：

- 12个孩子给12个孩子发信
- ❖ 应用进程 = 孩子
 - ❖ 应用消息 = 信封里的信
 - ❖ 主机 = 房子
 - ❖ 传输层协议 = 李雷和韩梅梅，信件收发功能
 - ❖ 网络层协议 = 邮政服务，信件传输功能





Internet传输层协议

5.1 传输层服务

❖ 可靠、按序的交付服务(TCP)

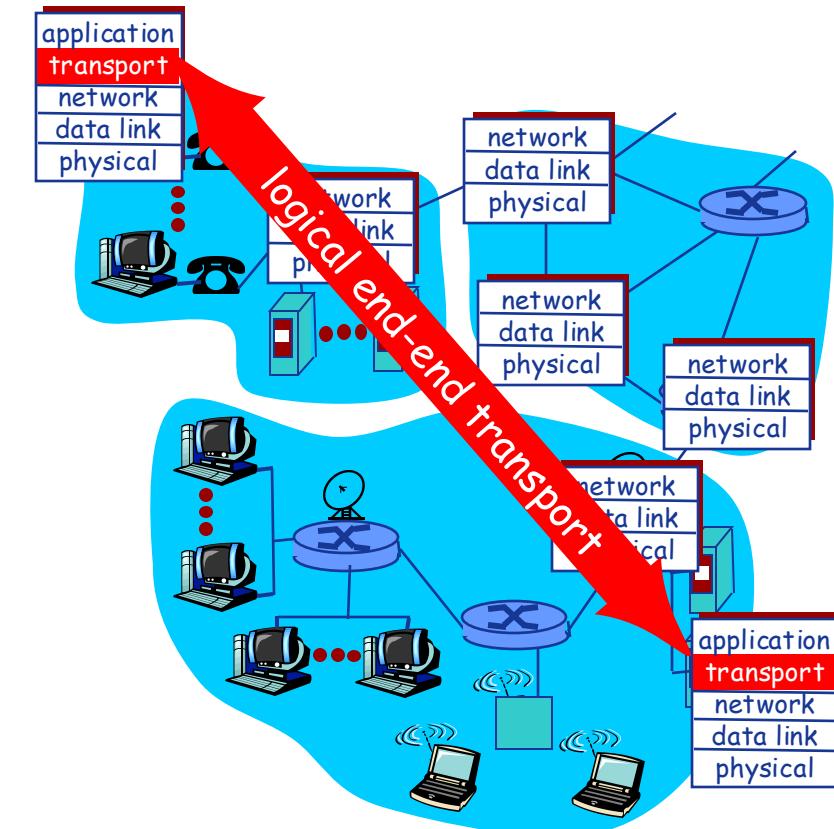
- 拥塞控制
- 流量控制
- 连接建立

❖ 不可靠的交付服务(UDP)

- 基于“尽力而为(Best-effort)”的网络层，没有做（可靠性方面的）扩展

❖ 两种服务均不保证

- 延迟
- 带宽





哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

立足航天，服务国防，面向国民经济主战场

5.2 传输层多路复用/分用



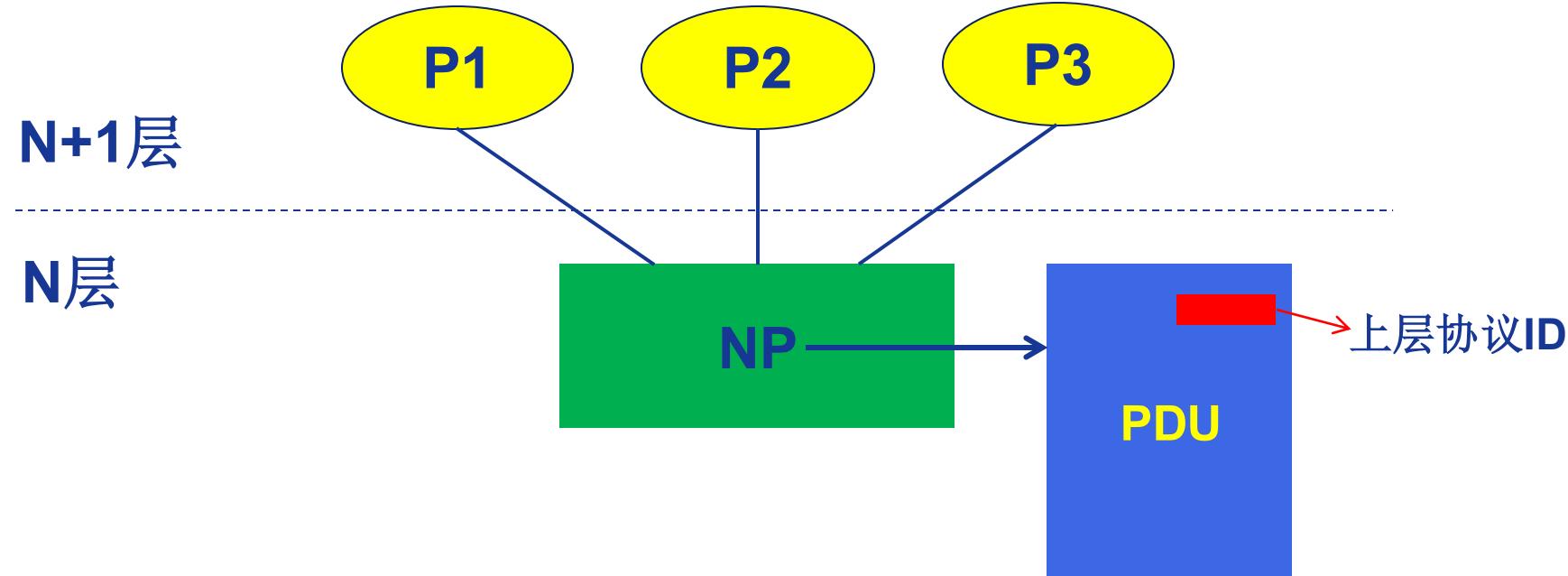
为什么需要多路复用 / 分用 ?

5.1 传输层服务

5.2 传输层多路复用/分解

Q: 为什么需要实现复用与分解? 如何实现复用与分解?

A: 如果某层的一个协议/实体直接为上层的多个协议/实体提供服务, 则需要分用/复用



➤ 复用与分解只在传输层进行吗?

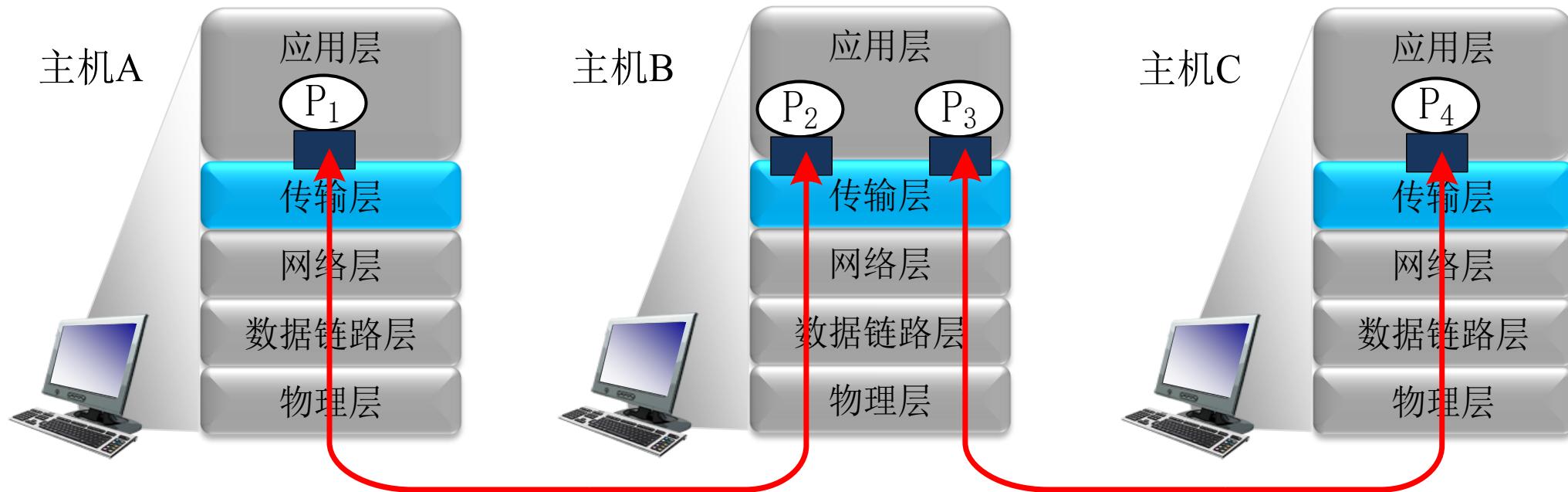




传输层多路复用/分用？

5.1 传输层服务

5.2 传输层多路复用/分解



图例: ○ 进程 ■ 套接字

- 传输层如何实现复用与分解功能?
- 可能通过其他方式实现复用与分解吗?





传输层多路复用/分用

5.1 传输层服务

5.2 传输层多路复用/分解

发送端进行多路复用:

从多个**Socket**接收数据，为每块数据封装上头部信息，生成**Segment**，交给网络层

接收端进行多路分用:

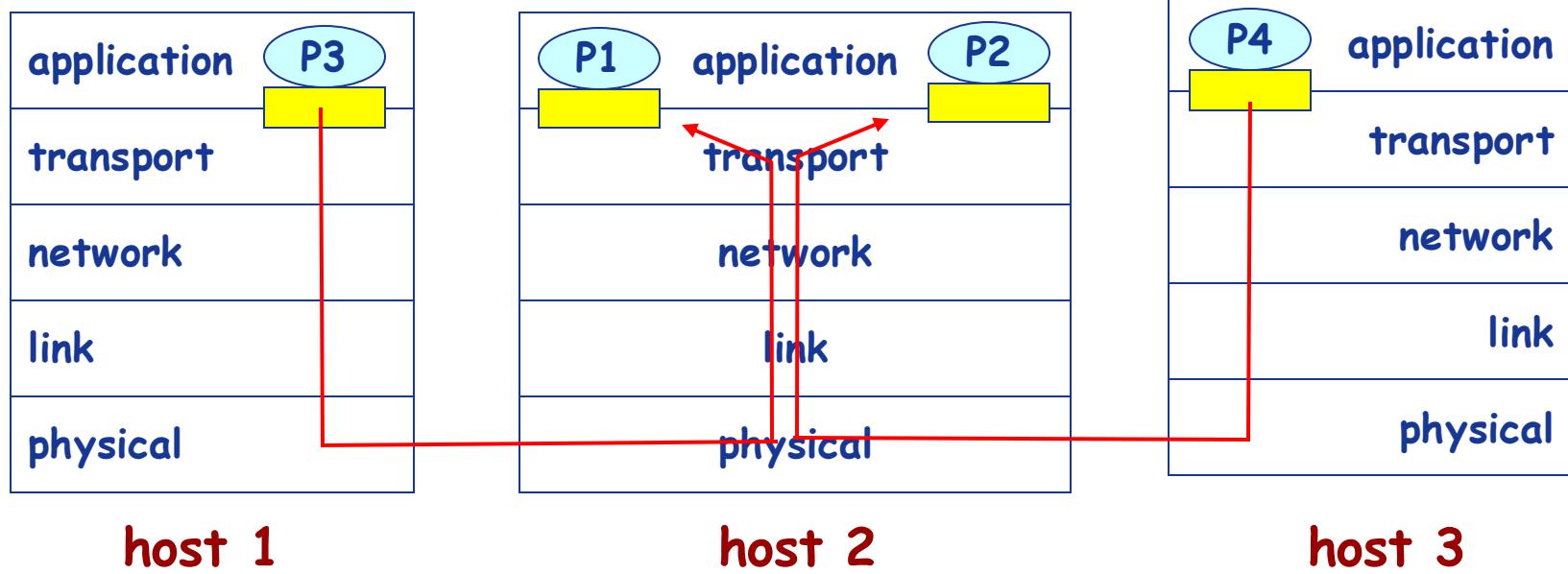
传输层依据头部信息将收到的**Segment**交给正确的**Socket**，即不同的进程



= socket



= process





传输层分用如何工作？

5.1 传输层服务

5.2 传输层多路复用/分解



❖ 主机接收到IP数据报(datagram)

- 每个数据报携带源IP地址、目的IP地址。
- 每个数据报携带一个传输层的段(Segment)。
- 每个段携带源端口号和目的端口号

❖ 主机收到Segment之后，传输层协议提取IP地址和端口号信息，将Segment导向相应的Socket

- TCP做更多处理



TCP/UDP 段格式



5.1 传输层服务

5.2 传输层多路复用/分解



传输层无连接分用

❖ 创建Socket并绑定端口号

```
DatagramSocket mySocket1 = new  
DatagramSocket(9911);
```

```
DatagramSocket mySocket2 = new  
DatagramSocket(9922);
```

❖ UDP的Socket用二元组标识

- (目的IP地址, 目的端口号)

❖ 主机收到UDP段后

- 检查段中的目的端口号
- 将UDP段导向绑定在该端口号的Socket

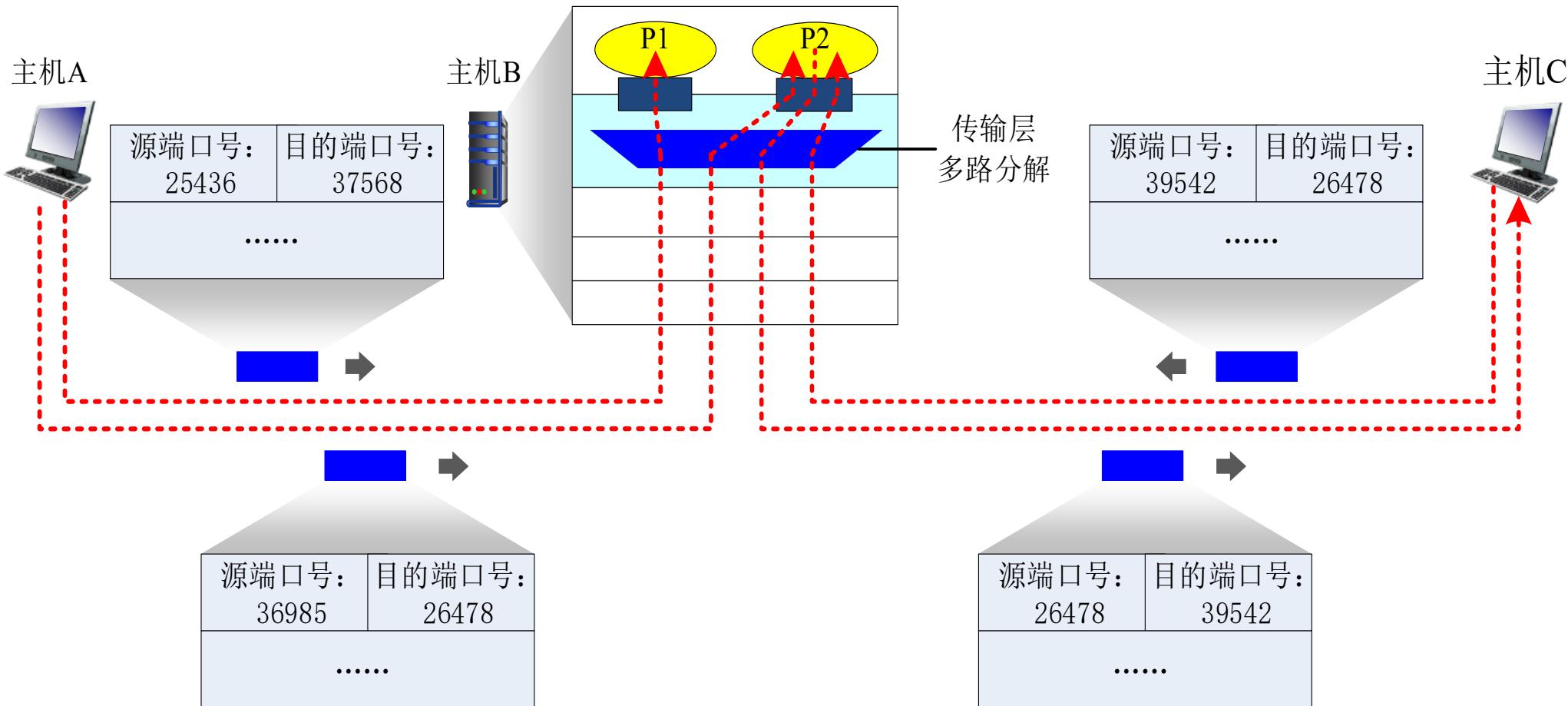
❖ 来自不同源IP地址和/或源端口号的IP数据包被导向同一个Socket



传输层无连接分用

5.1 传输层服务

5.2 传输层多路复用/分解





传输层面向连接的分用

5.1 传输层服务

5.2 传输层多路复用/分解

- ❖ TCP的Socket用四元组标识
 - 源IP地址
 - 源端口号
 - 目的IP地址
 - 目的端口号
- ❖ 接收端利用所有的四个值将 Segment导向正确的Socket
- ❖ 服务器可能同时支持多个 TCP Socket
 - 每个Socket用自己的四元组标识
- ❖ Web服务器为每个客户端创建不同的Socket

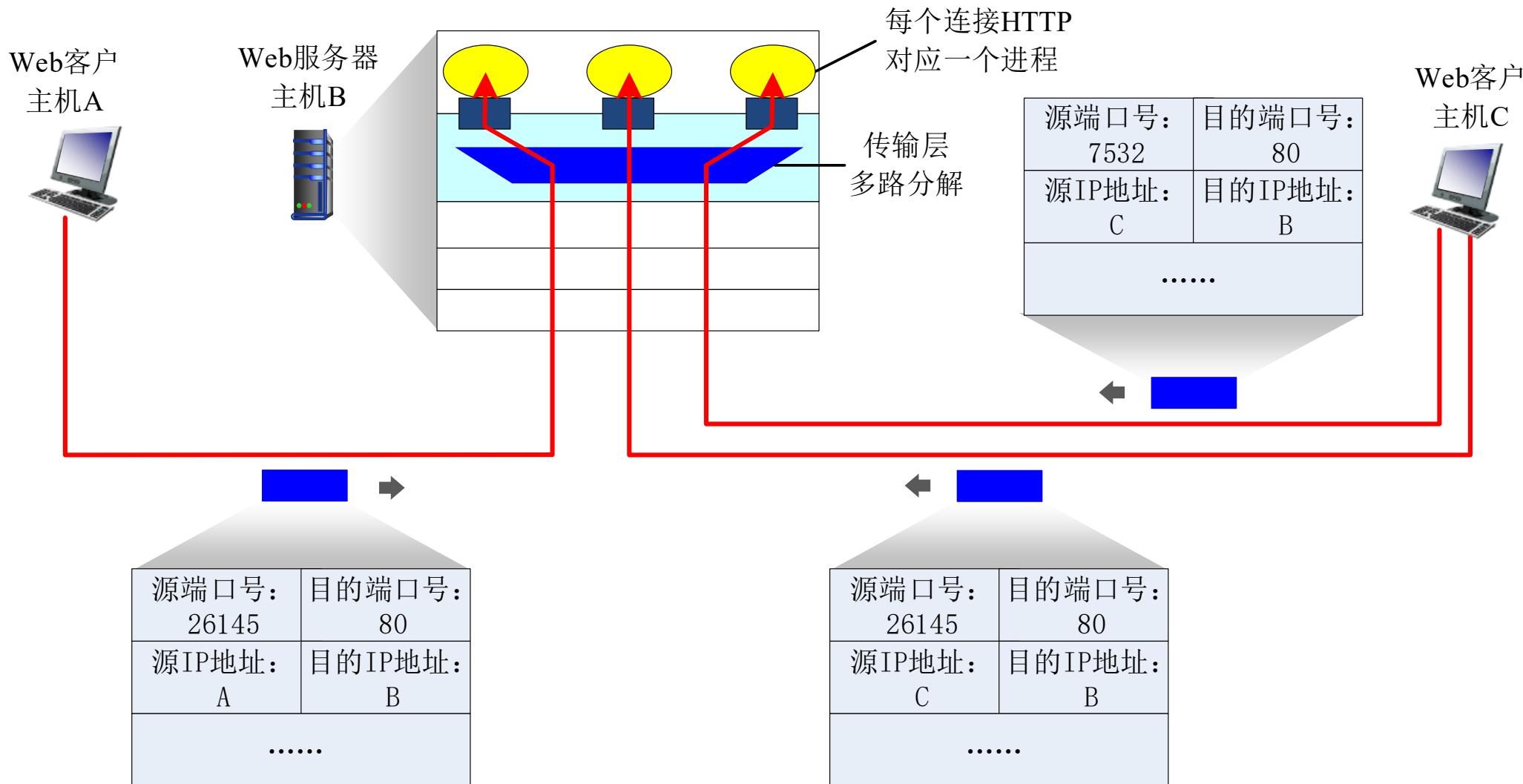




传输层面向连接的分用

5.1 传输层服务

5.2 传输层多路复用/分解





哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

立足航天，服务国防，面向国民经济主战场

5.3 UDP协议



UDP: 用户数据报协议 [RFC 768]

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议



- ❖ 基于Internet IP协议
 - 复用/分用
 - 简单的错误校验
- ❖ “Best effort” 服务, UDP段可能
 - 丢失
 - 非按序到达
- ❖ 无连接
 - UDP发送方和接收方之间不需要握手
 - 每个UDP段的处理独立于其他段

为什么需要UDP?

- ❖ 无需建立连接
(减少延迟)
- ❖ 实现简单: 无需维护连接状态
- ❖ 头部开销少
- ❖ 没有拥塞控制: 应用可更好地控制发送时间和速率



UDP: 用户数据报协议 [RFC 768]

5.1 传输层服务

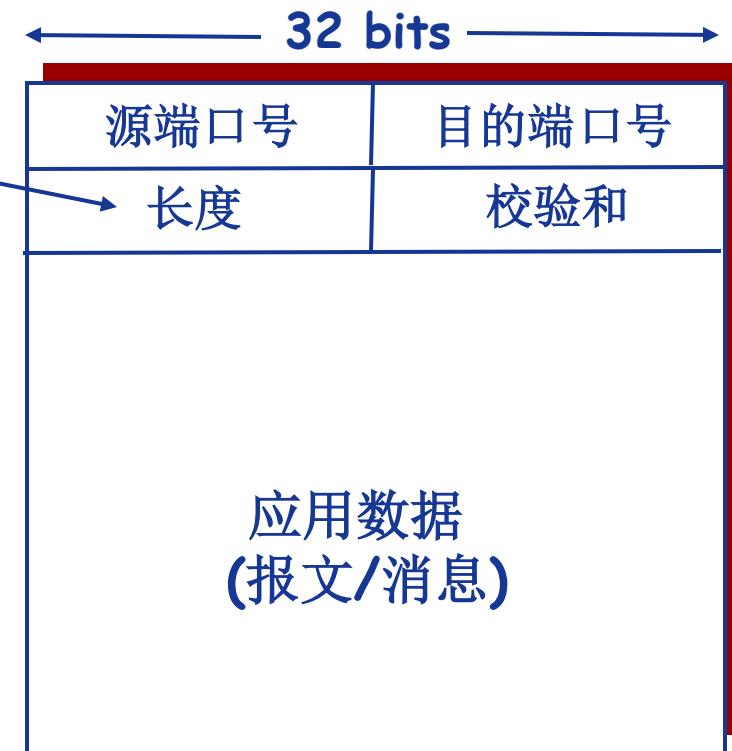
5.2 传输层多路复用/分解

5.3 UDP协议



- ❖ 常用于流媒体应用
 - 容忍丢失
 - 速率敏感
- ❖ UDP还用于
 - DNS
 - SNMP
- ❖ 在UDP上实现可靠数据传输?
 - 在应用层增加可靠性机制
 - 应用特定的错误恢复机制
 - 例如：停等协议、滑动窗口协议

UDP段的长度
(包含头部)



UDP报文段格式



UDP校验和(checksum)

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议



目的：检测**UDP**段在传输中是否发生错误（如位翻转）

❖ 发送方

- 将参与校验和计算的所有内容视为**16-bit整数序列**
- 校验和计算：
 - 计算整数序列的和（**sum**）
 - 进位也要加在和的后面
 - 将和按位求反（即反码），得到校验和（**checksum**）
- 发送方将校验和放入校验和字段

❖ 接收方

- 针对收到的**UDP**报文段，按发送方同样的方法构建**16位整数序列**
- 按相同算法计算整数序列的和（**sum**）
- 若**sum=1111111111111111**，则无错；否则，有错



UDP校验和(checksum)

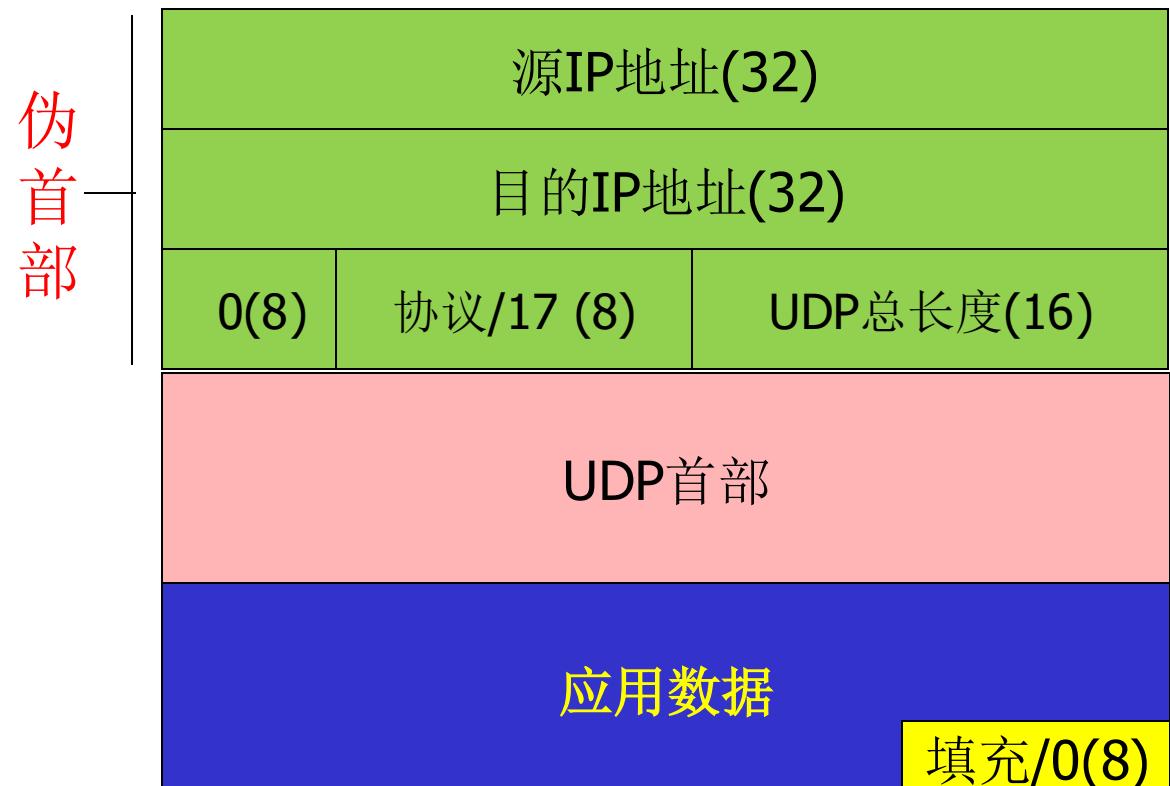
5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

❖ 3 部分:

- 伪首部
(Pseudo head)
- UDP首部
- 应用数据





校验和计算示例

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

❖ 示例：

	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
<hr/>																
回卷	1	1	0	1	1	1	0	1	1	1	0	1	1	0	1	1
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	1

❖ 注意：

- 最高位进位必须被加进去





立足航天，服务国防，面向国民经济主战场

5.4 TCP协议



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

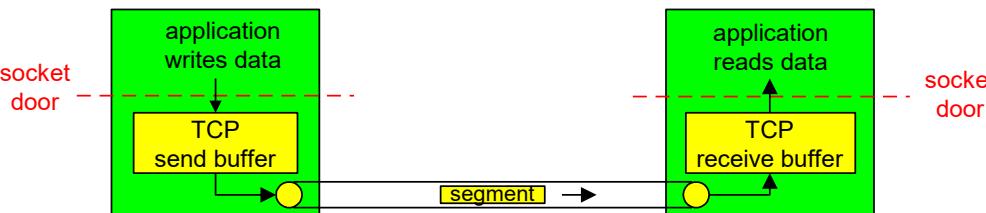
5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

- ❖ 点对点
 - 一个发送方, 一个接收方
- ❖ 可靠的、按序字节流
- ❖ 流水线机制
 - TCP拥塞控制和流量控制机制设置窗口尺寸
- ❖ 发送方/接收方缓存
- ❖ 全双工(full-duplex)
 - 同一连接中能够传输双向数据流
- ❖ 面向连接
 - 通信双方在发送数据之前必须建立连接。
 - 连接状态只在连接的两端中维护, 在沿途节点中并不维护状态。
 - TCP连接包括: 两台主机上的缓存、连接状态变量、socket等
- ❖ 流量控制机制
- ❖ 拥塞控制



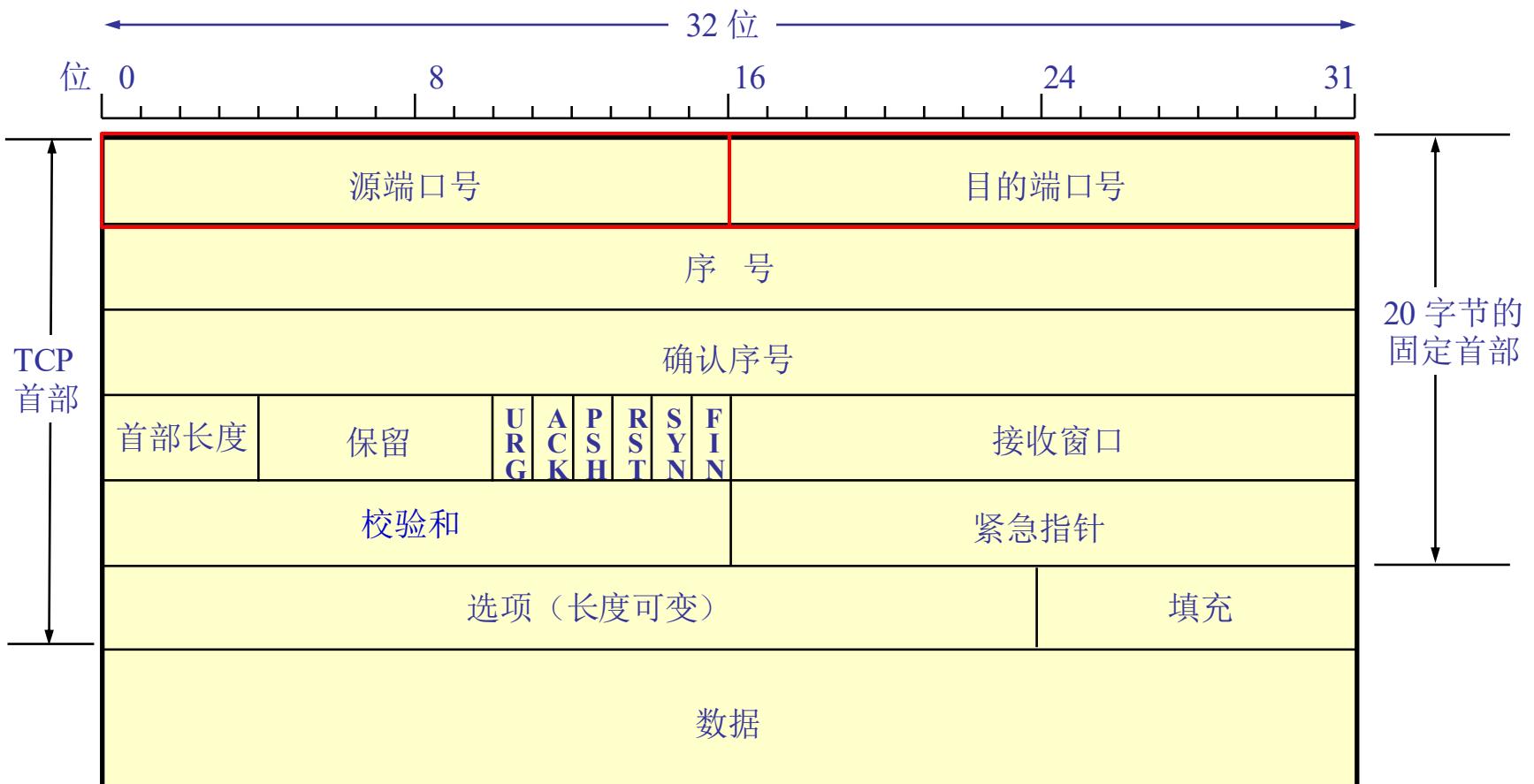


TCP段结构

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议**TCP段结构**

- ❖ 源端口号与目的端口号字段分别占16位
 - 多路复用/分解



TCP段结构

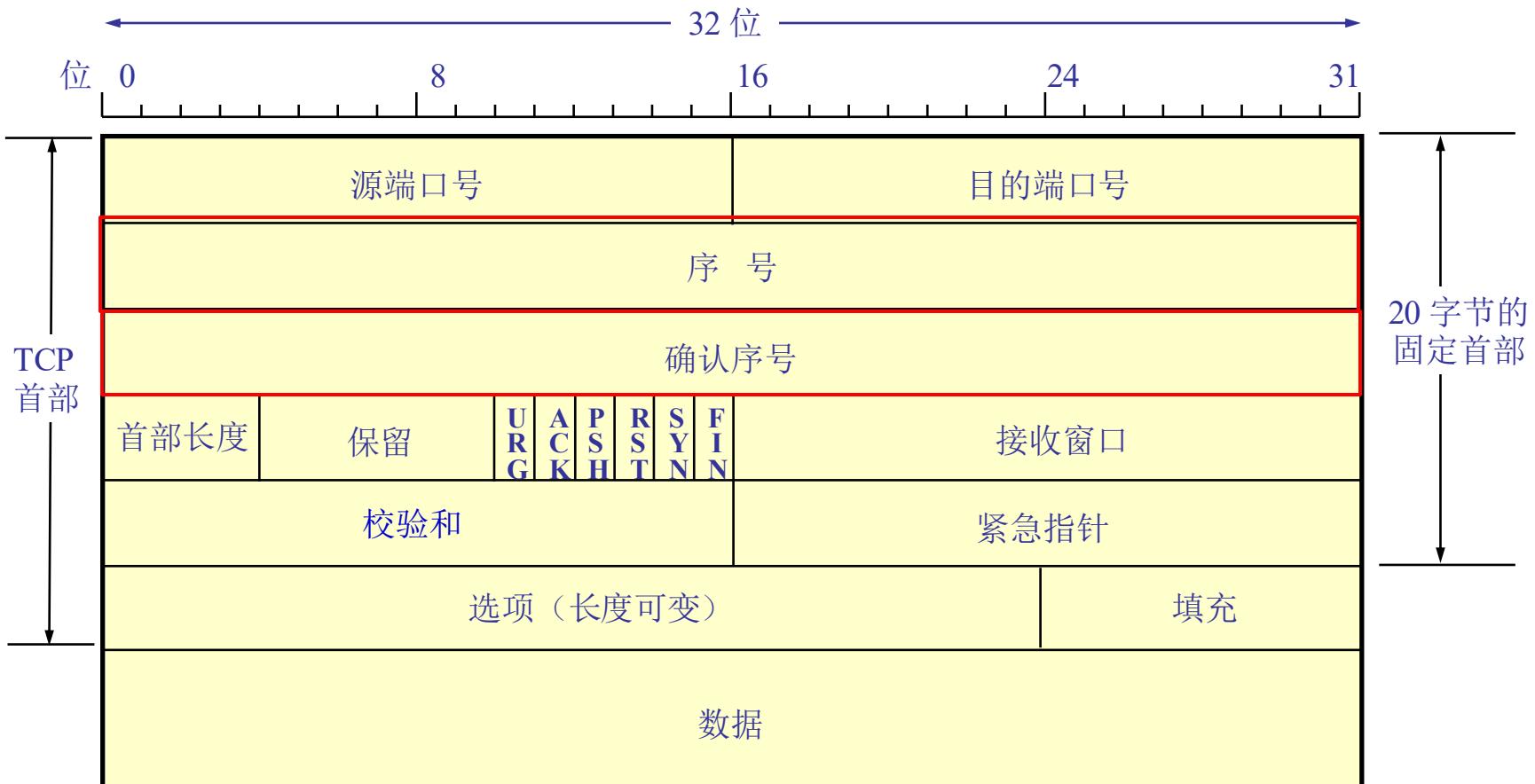
5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP段结构



❖ 序号字段与确认序号字段分别占32位

- 对每个应用层数据的每个字节进行编号
- 确认序号是期望从对方接收数据的字节序号，累计确认



TCP段结构

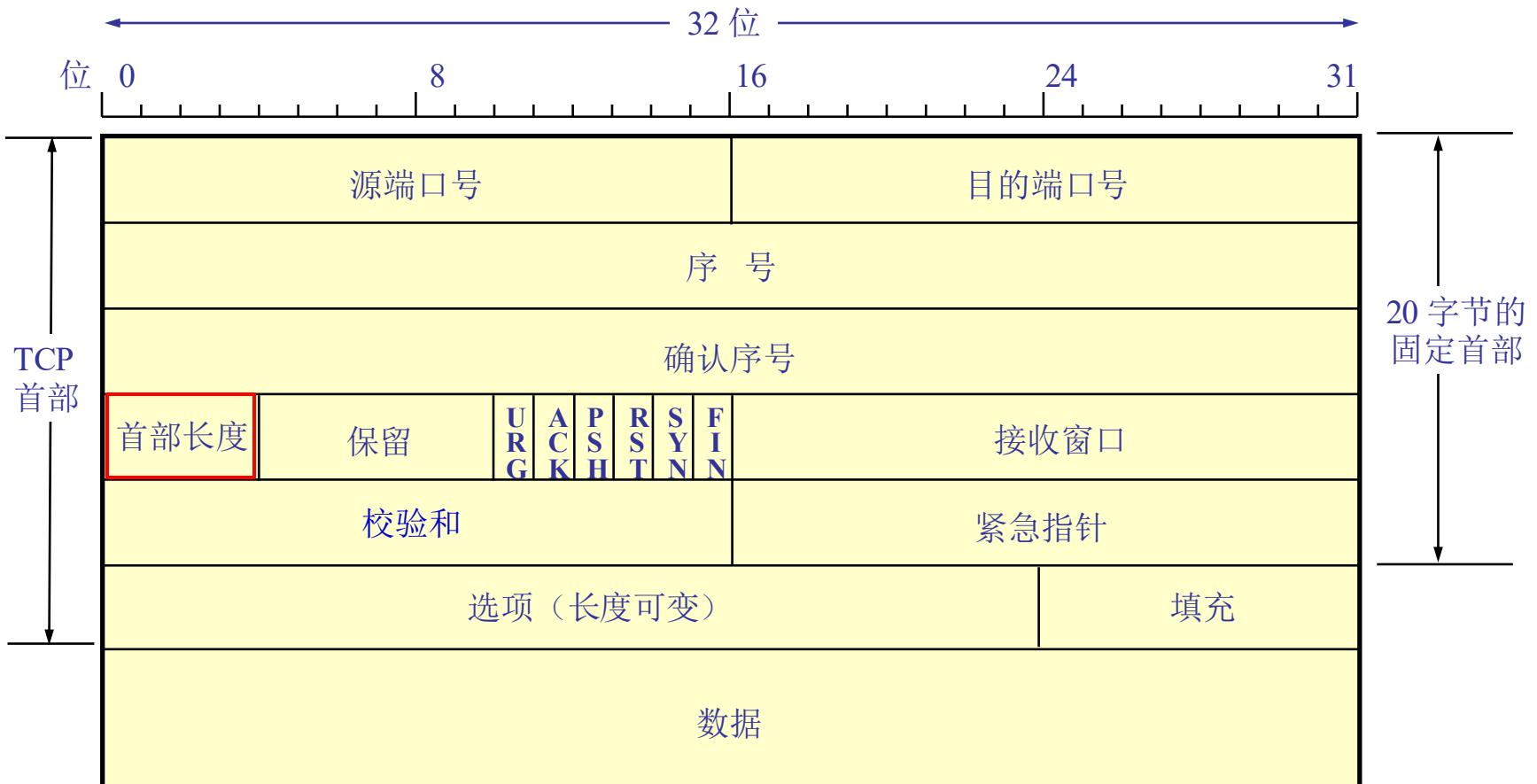
5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP段结构



- ❖ 首部长度字段占4位
 - 4字节为计算单位

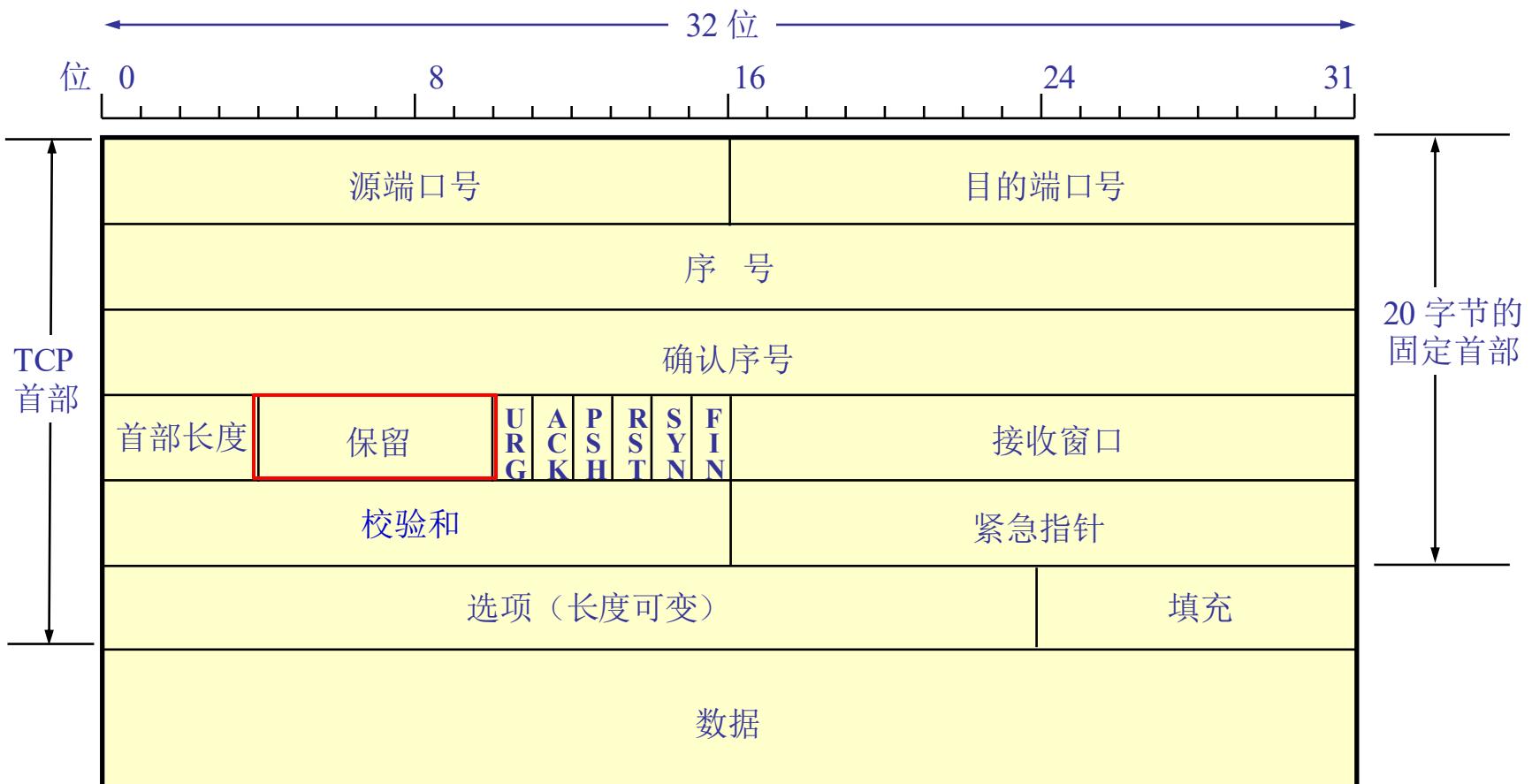


TCP段结构

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议**TCP段结构**

- ❖ 保留字段占6位
 - 目前值为0



TCP段结构

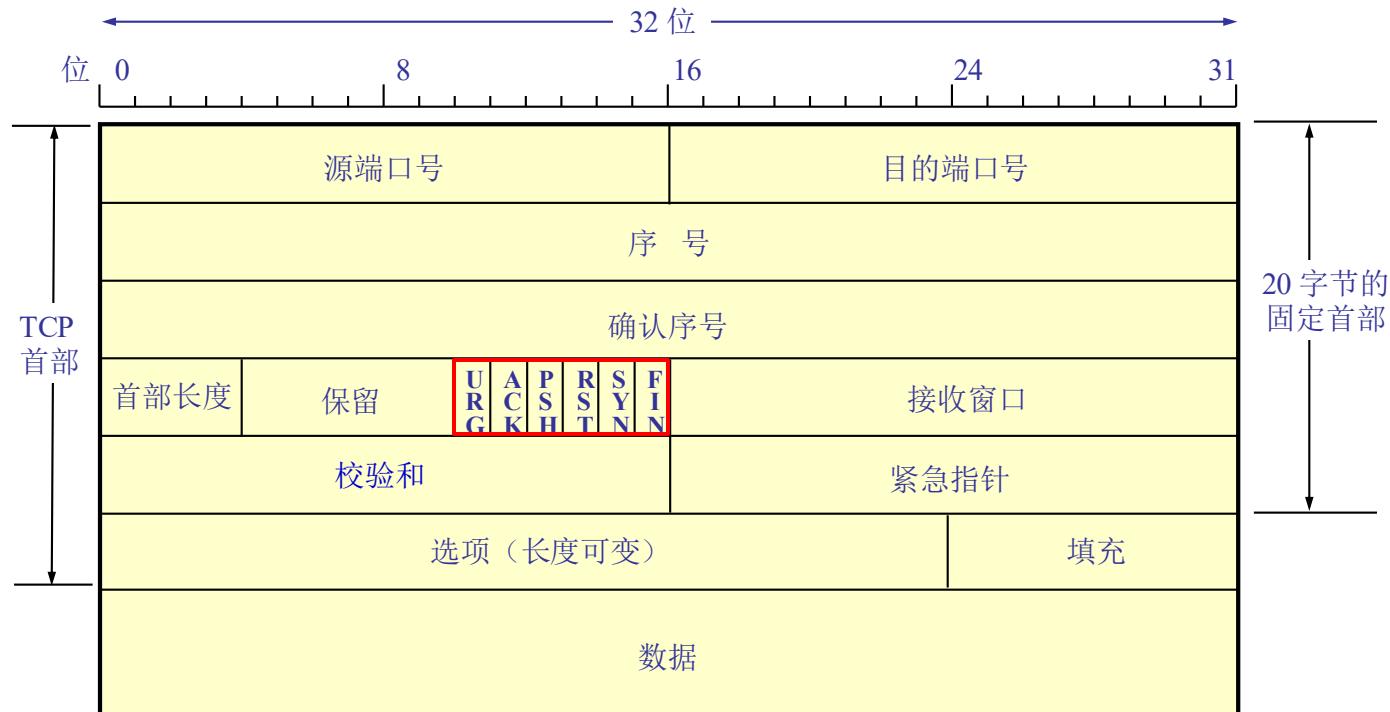
5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP段结构



❖ 6位标志位（字段）

- URG=1时，表明紧急指针字段有效
- ACK=1时，标识确认序号字段有效
- PSH=1时，尽快将段中数据交付接收应用进程
- RST=1时，重新建立TCP连接
- SYN=1时，表示该TCP段是一个建立新连接请求控制段
- FIN=1时，表明请求释放TCP连接

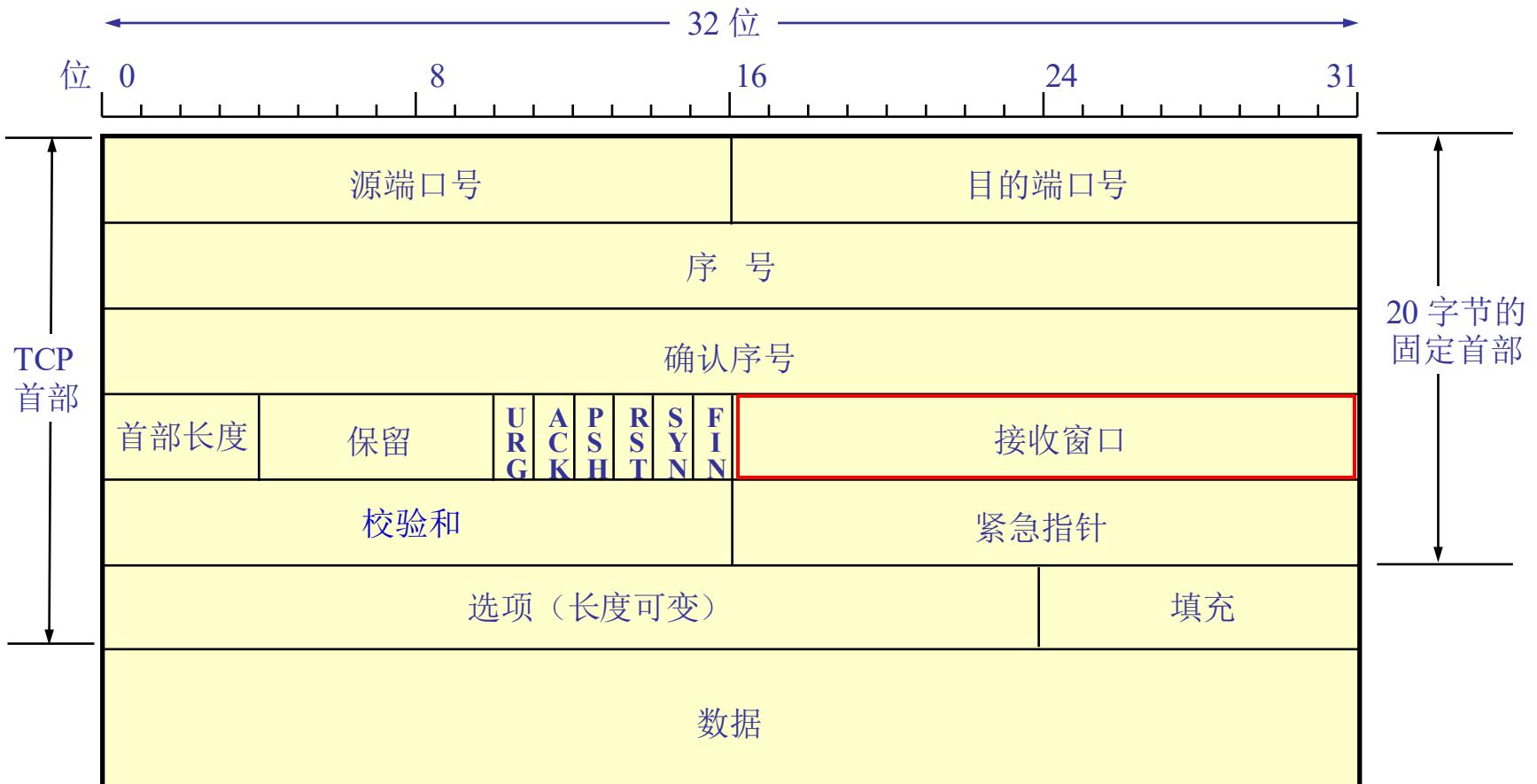


TCP段结构

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议**TCP段结构**

❖ 接收窗口字段占16位

- 流量控制

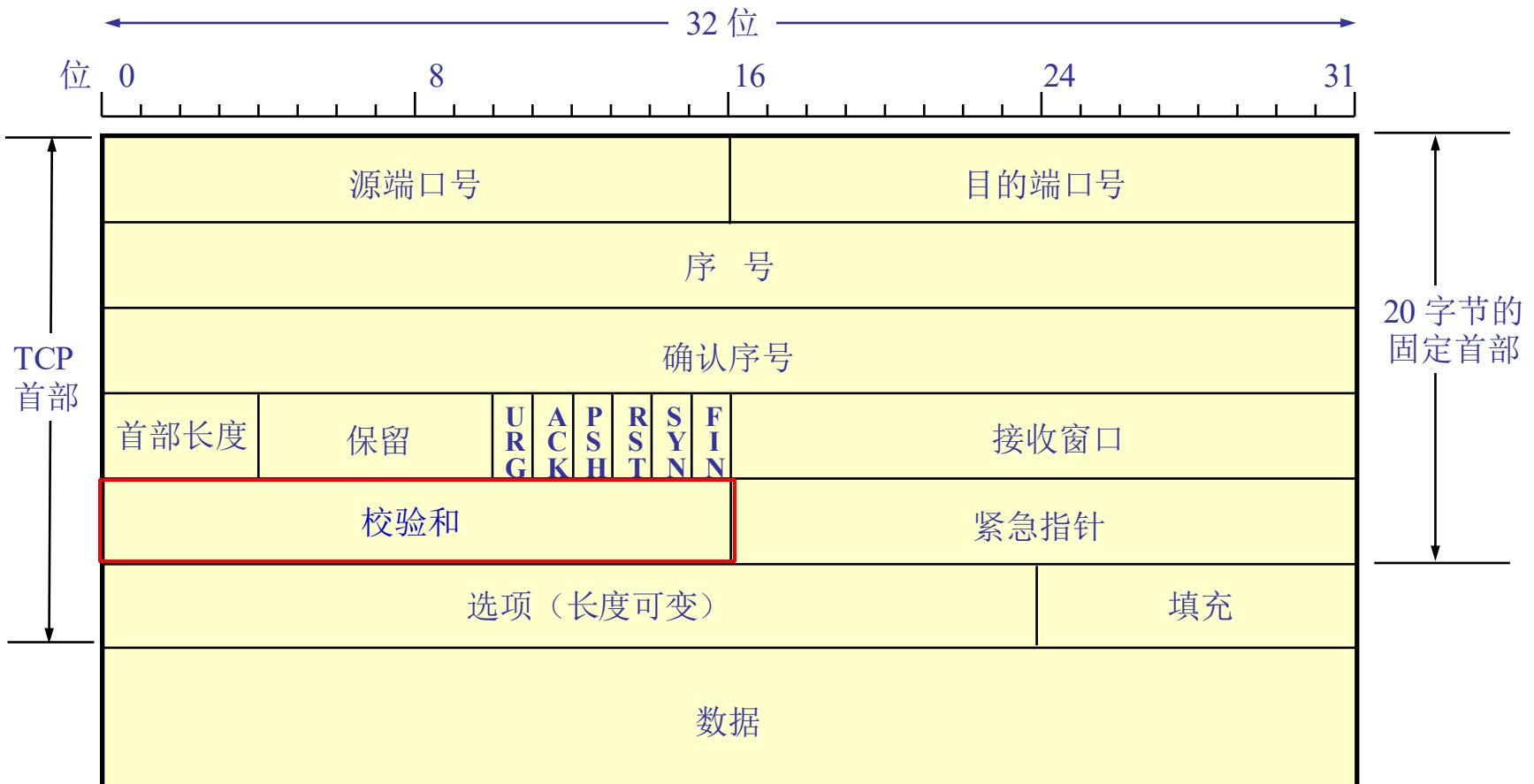


TCP段结构

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议**TCP段结构**

- ❖ 校验和字段占16位
 - 包括TCP伪首部、TCP首部和应用层数据三部分



TCP段结构

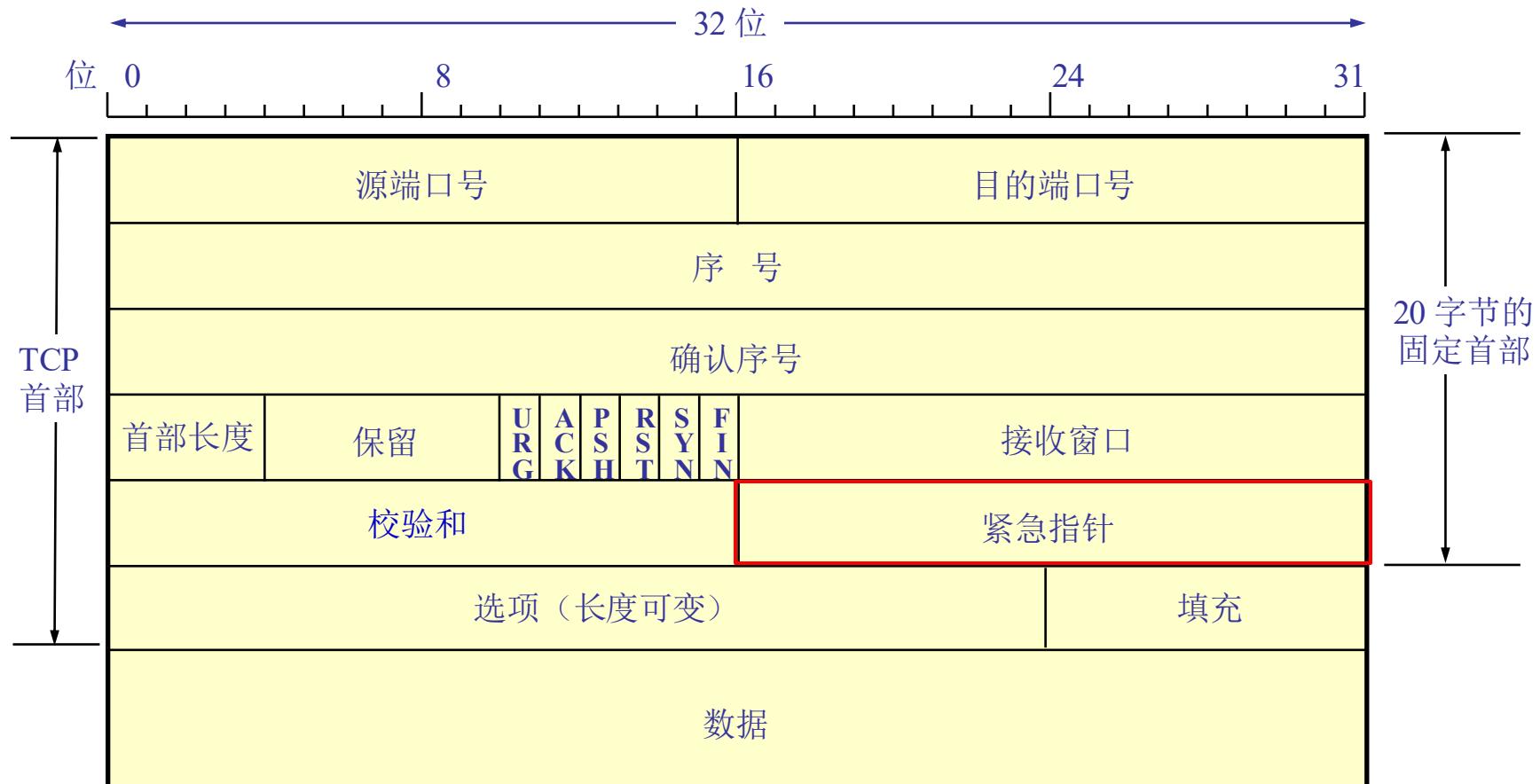
5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP段结构



- ❖ 紧急指针字段占16位
 - URG=1时才有效
 - 指出紧急数据最后一个字节在数据中的位置



TCP段结构

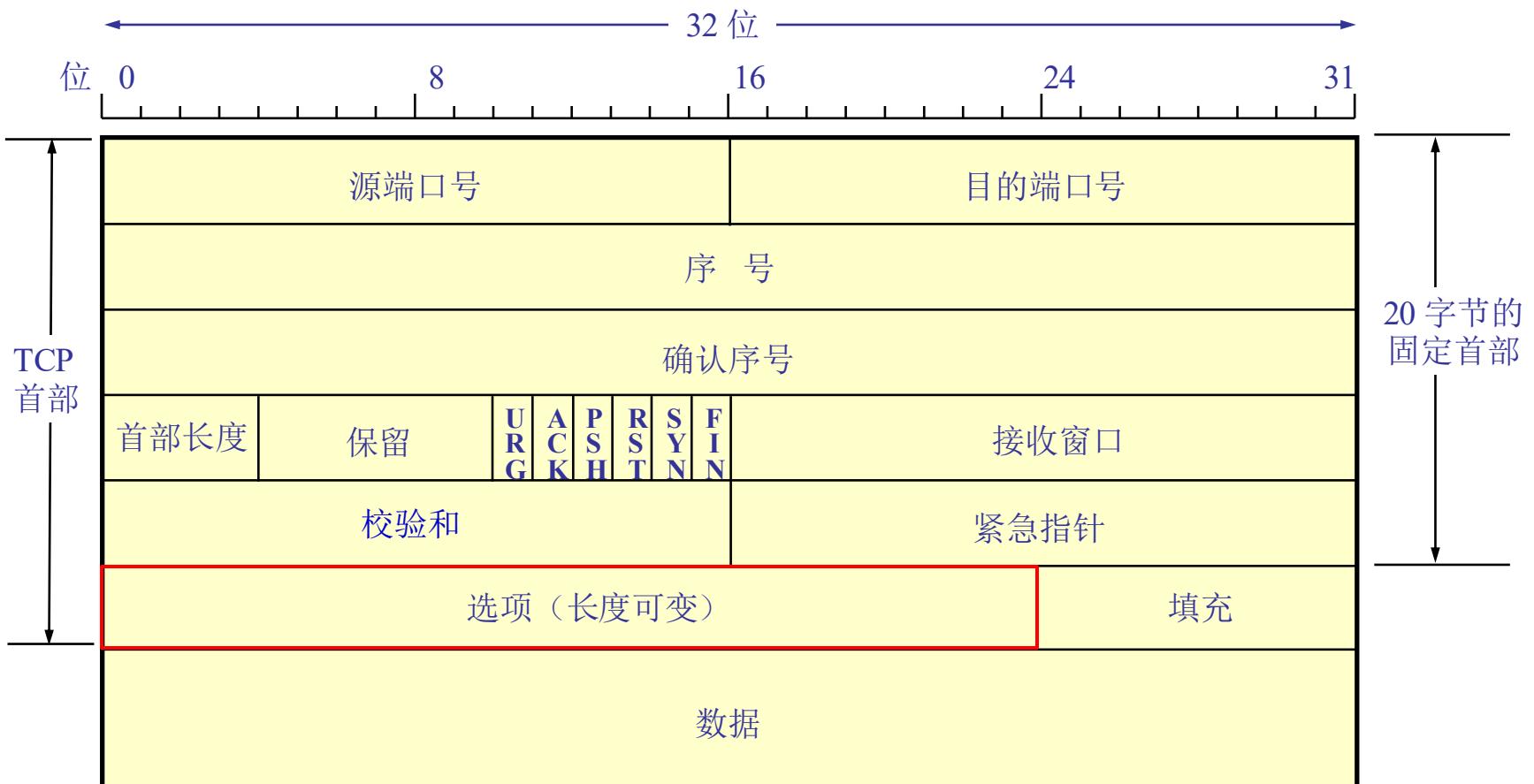
5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP段结构



❖ 选项字段的长度可变

- 最大段长度MSS
- 时间戳
- SACK

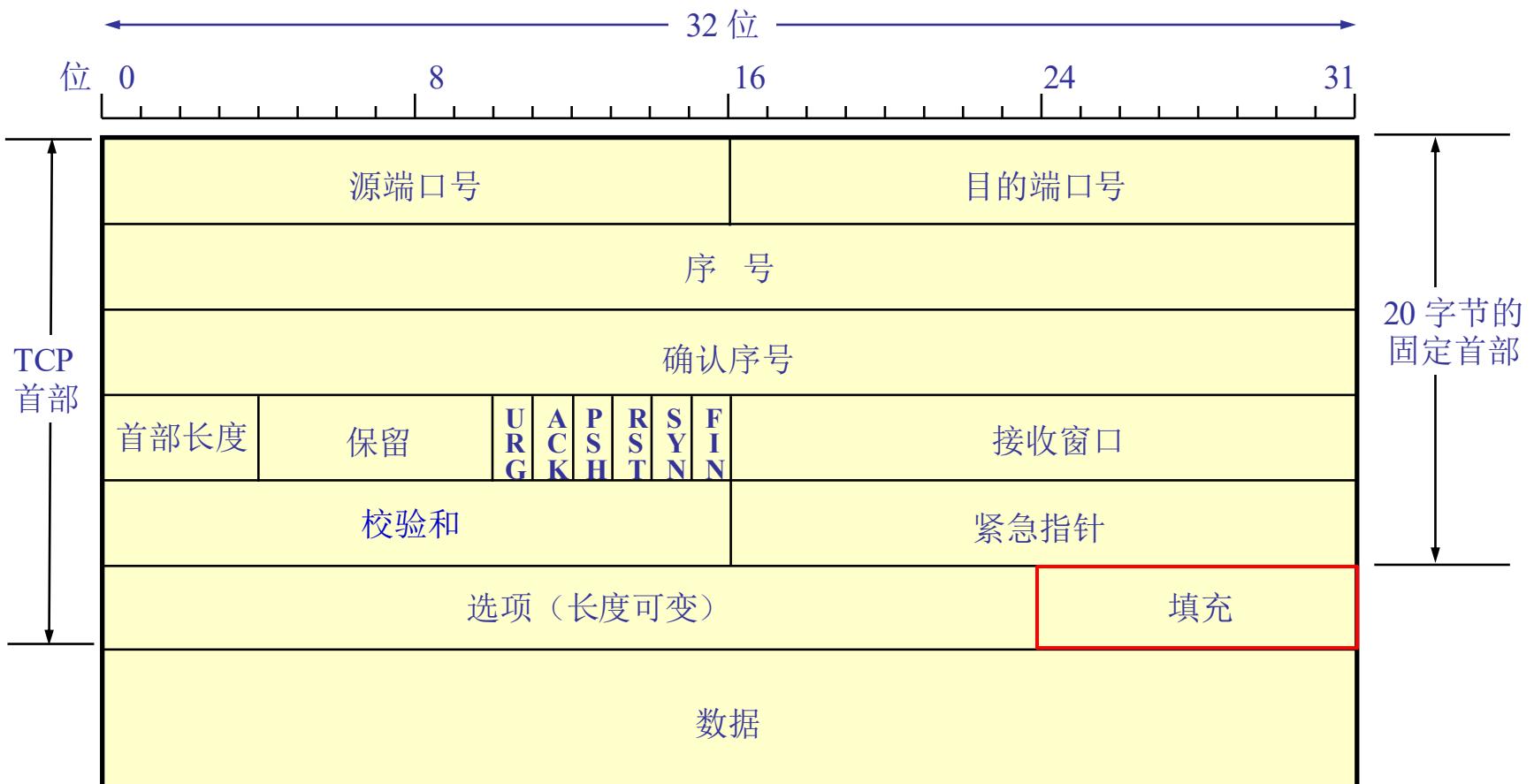


TCP段结构

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议**TCP段结构**

- ❖ 填充字段，长度为0~3个字节
 - 取值全0



5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP段结构



序列号:

- 序号是段（ **Segment** ）中第1个字节的编号，而不是段的“连续”编号
- 建立**TCP**连接时，双方随机选择序列号

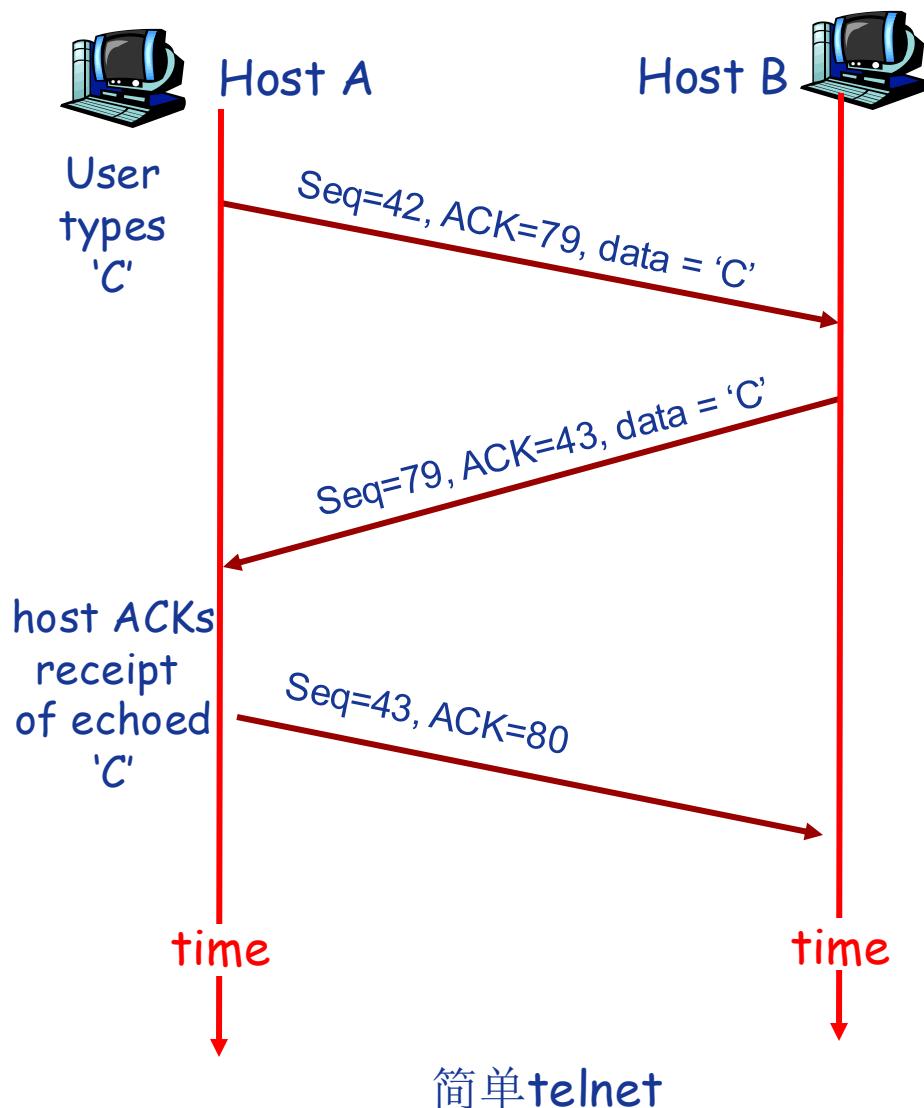
ACKs:

- 期望接收到的下一个字节的序列号
- 累计确认：该序列号之前的所有字节均已被正确接收到

Q: 接收方如何处理乱序到达的段？

- **A:** **TCP**规范中没有规定，由**TCP**的实现者做出决策

TCP: 序列号和ACK





TCP可靠数据传输概述

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP可靠数据传输



- ❖ TCP在IP层的不可靠服务基础上实现可靠数据传输服务
- ❖ 流水线机制
- ❖ 累积确认
- ❖ TCP使用单一重传定时器
- ❖ 触发重传的事件
 - 超时
 - 收到重复ACK
- ❖ 漸进式



TCP RTT和超时

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP可靠数据传输



- ❖ **问题:** 如何设置定时器的超时时间?
- ❖ 大于RTT
 - 但是RTT是变化的
- ❖ 过短:
 - 不必要的重传
- ❖ 过长:
 - 对段丢失时间反应慢

- ❖ **问题:** 如何估计RTT?
- ❖ **SampleRTT:** 测量从段发出去到收到ACK的时间
 - 忽略重传
- ❖ **SampleRTT变化**
 - 测量多个SampleRTT, 求平均值, 形成RTT的估计值 EstimatedRTT

$$\text{EstimatedRTT} = (1 - \alpha) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$

指数加权移动平均 **α 典型值: 0.125**



TCP RTT和超时

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP可靠数据传输



定时器超时时间的设置?

- **EstimatedRTT + “安全边界”**
- **EstimatedRTT变化大 → 较大的边界**

测量RTT的变化值: SampleRTT与EstimatedRTT的差值

$$\text{DevRTT} = (1 - \beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(typically, $\beta = 0.25$)

定时器超时时间的设置:

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$



TCP发送方事件

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP可靠数据传输



❖ 从应用层收到数据

- 创建Segment
- 序列号是Segment第一个字节的编号
- 开启定时器
- 设置超时时间：
TimeOutInterval

❖ 超时

- 重传引起超时的段
- 重启定时器

❖ 收到ACK

- 如果确认此前未确认的段
 - 更新SendBase
 - 如果窗口中还有未被确认的分组，重新启动定时器



TCP发送端程序

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP可靠数据传输



```
NextSeqNum = InitialSeqNum  
SendBase = InitialSeqNum  
loop (forever) {  
    switch(event)  
        event: data received from application above  
            create TCP segment with sequence number NextSeqNum  
            if (timer currently not running)  
                start timer  
            pass segment to IP  
            NextSeqNum = NextSeqNum + length(data)  
  
        event: timer timeout  
            retransmit not-yet-acknowledged segment with  
                smallest sequence number  
            start timer  
  
        event: ACK received, with ACK field value of y  
        if (y > SendBase) {  
            SendBase = y  
            if (there are currently not-yet-acknowledged segments)  
                start timer  
        }  
    } /* end of loop forever */
```



TCP重传示例

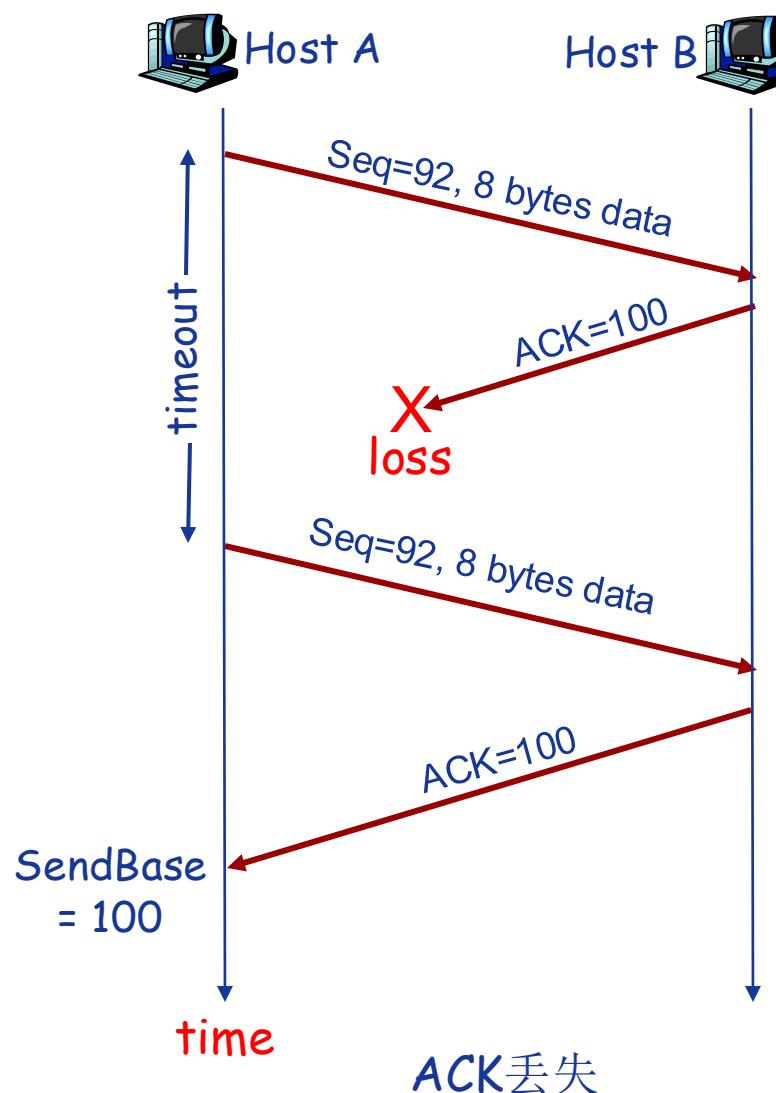
5.1 传输层服务

5.2 传输层多路复用/分解

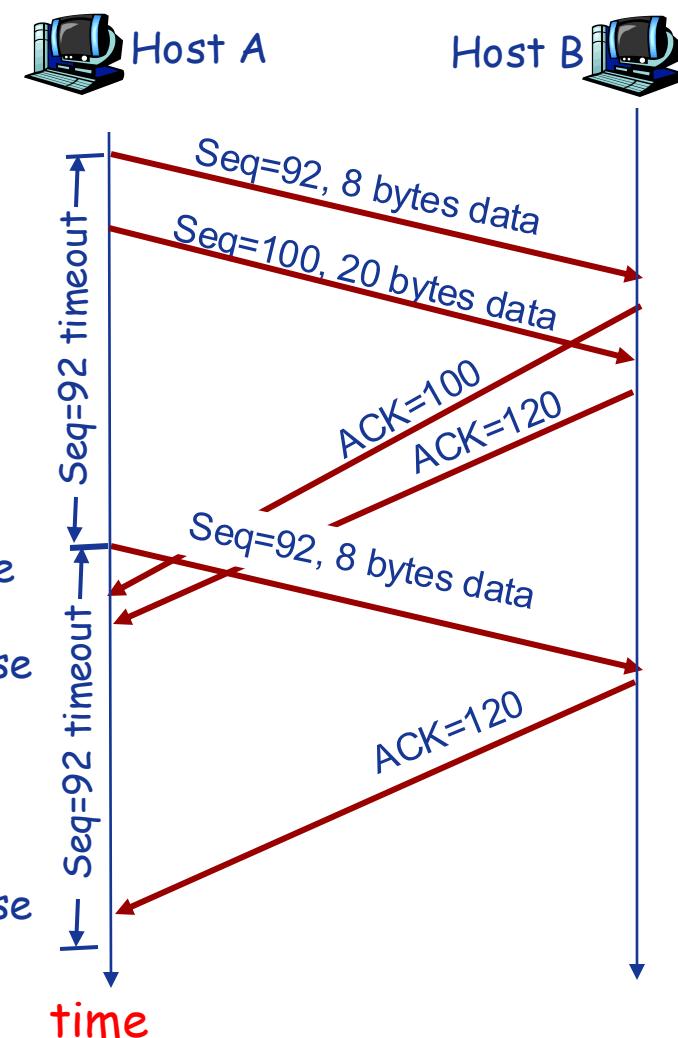
5.3 UDP协议

5.4 TCP协议

TCP可靠数据传输



ACK丢失

 $\text{Sendbase} = 100$
 $\text{SendBase} = 120$ 

过早超时



TCP重传示例

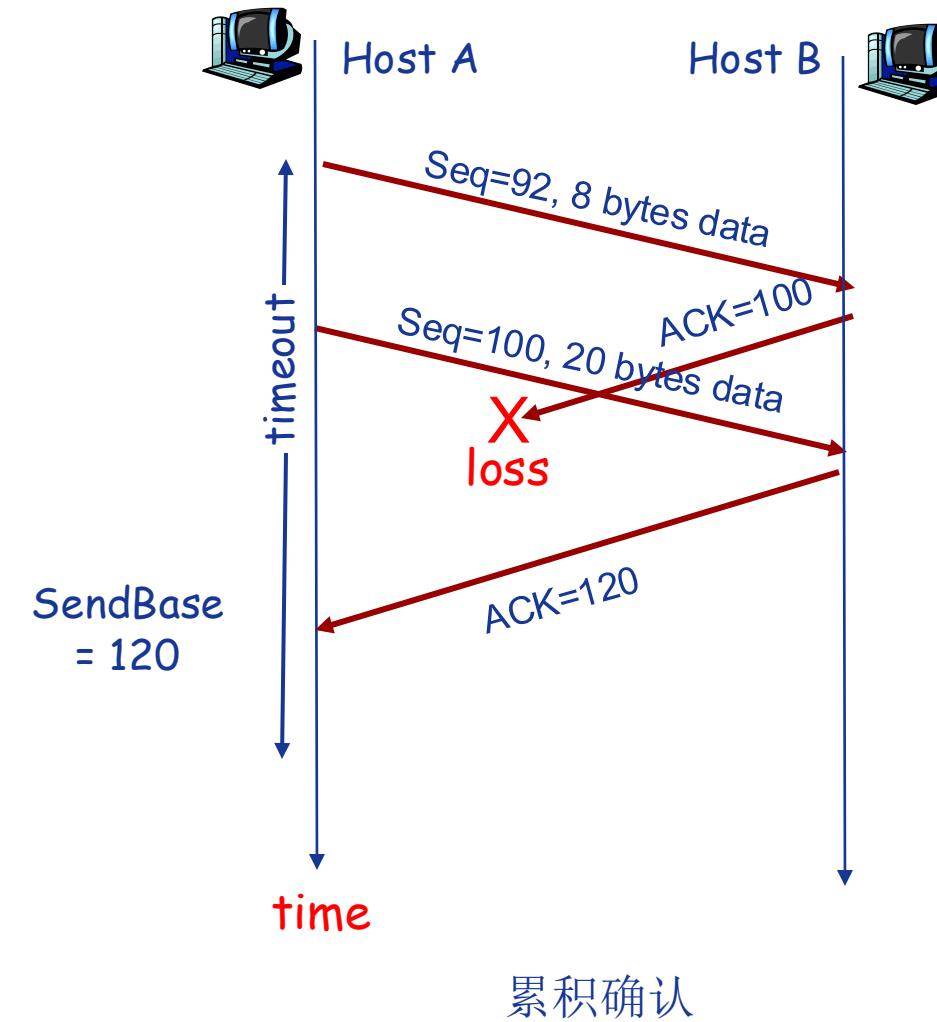
5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP可靠数据传输





TCP ACK生成: RFC 1122, RFC 2581

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP可靠数据传输



Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send <i>duplicate ACK</i> , indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment startsat lower end of gap



5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP可靠数据传输



快速重传机制

- ❖ TCP的实现中，如果发生超时，超时间隔将重新设置，即将超时间隔加倍，导致其很大
 - 重发丢失的分组之前要等待很长时间
- ❖ 通过重复ACK检测分组丢失
 - Sender会背靠背地发送多个分组
 - 如果某个分组丢失，可能会引发多个重复的ACK
- ❖ 如果sender收到对同一数据的3个ACK，则假定该数据之后的段已经丢失
 - 快速重传：在定时器超时之前即进行重传



快速重传算法

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP可靠数据传输



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

a duplicate ACK for
already ACKed segment

fast retransmit

单选题 1分

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP可靠数据传输



主机甲与主机乙间已建立一个TCP连接，主机甲向主机乙发送了两个连续的TCP段，分别包含300字节和500字节的有效载荷，第一个段的序列号为200，主机乙正确接收到两个段后，发送给主机甲的确认序列号是

- A 500
- B 700
- C 800
- D 1000

提交

单选题 1分

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

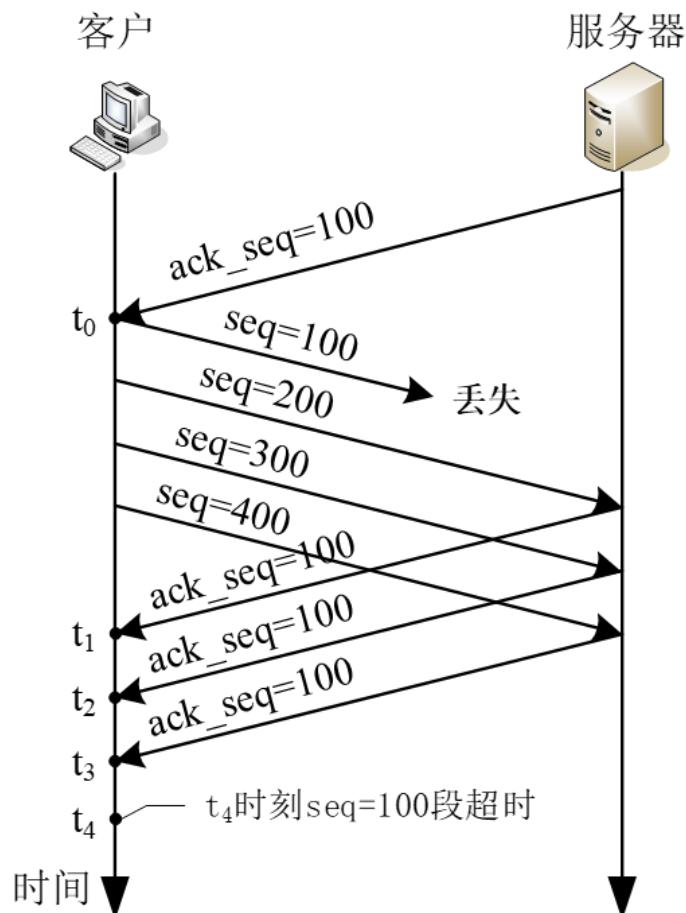
5.4 TCP协议

TCP可靠数据传输



- A t_1
- B t_2
- C t_3
- D t_4

某客户通过一个TCP连接向服务器发送数据的部分过程如下图所示。客户在 t_0 时刻第一次收到确认序列号 $ack_seq=100$ 的段，并发送序列号 $seq=100$ 的段，但发生丢失。若TCP支持快速重传，则客户重新发送 $seq=100$ 段的时刻是



提交



TCP流量控制

5.1 传输层服务

5.2 传输层多路复用/分解

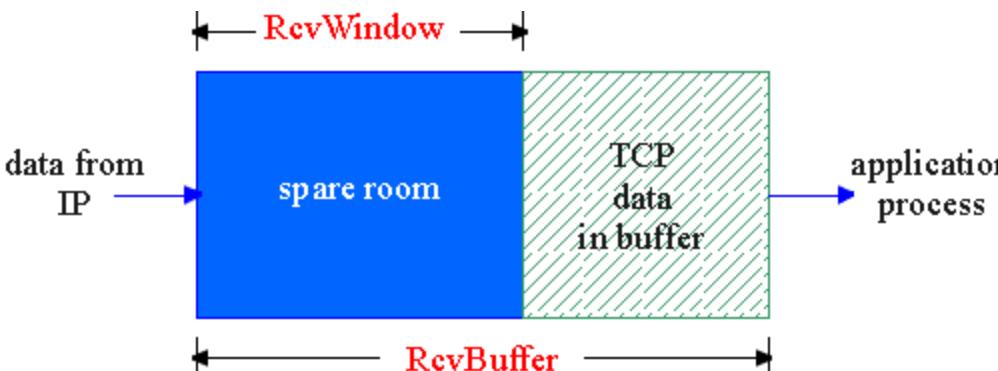
5.3 UDP协议

5.4 TCP协议

TCP流量控制



- ❖ 接收方为TCP连接分配 buffer



- 上层应用可能处理 buffer 中数据的速度较慢

flow control

发送方不会传输的太多、
太快以至于淹没接收方
(buffer溢出)

- ❖ 速度匹配机制



TCP流量控制

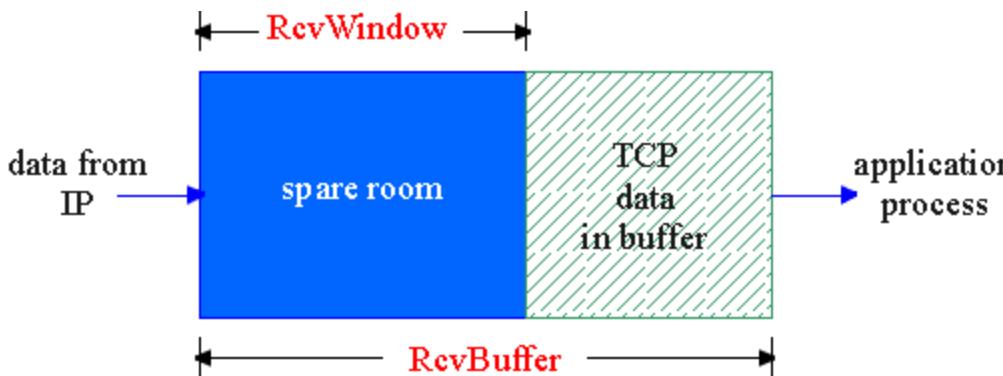
5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP流量控制



(假定TCP receiver丢弃乱序的
segments)

- ❖ Buffer中的可用空间(spare room)
 - = RcvWindow
 - = RcvBuffer - [LastByteRcvd - LastByteRead]

- ❖ Receiver通过在Segment的头部字段将**RcvWindow**告诉Sender
- ❖ Sender限制自己已经发送的但还未收到ACK的数据不超过接收方的空闲**RcvWindow**尺寸
- ❖ Receiver告知Sender **RcvWindow=0**,会出现什么情况?





5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP连接管理



TCP连接管理

❖ TCP sender 和 receiver 在传输数据前需要建立连接

❖ 初始化 TCP 变量

- Seq. #
- Buffer 和流量控制信息

❖ Client: 连接发起者

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

❖ Server: 等待客户连接请求

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

Three way handshake:

Step 1: client host sends TCP SYN segment to server

- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment

- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data



TCP连接管理：建立

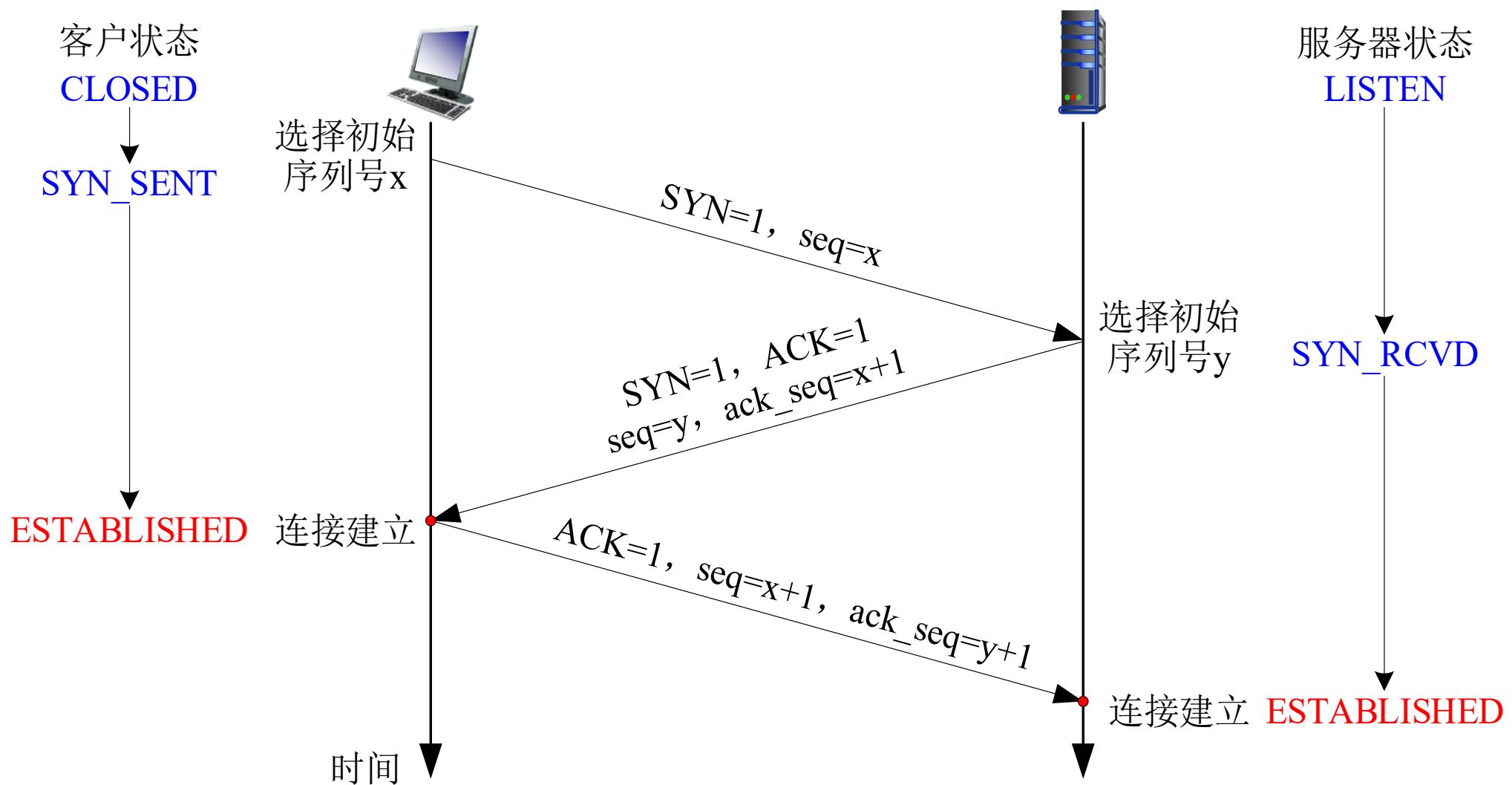
5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP连接管理





TCP连接管理：关闭

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP连接管理



Closing a connection:

client closes socket: `clientSocket.close();`

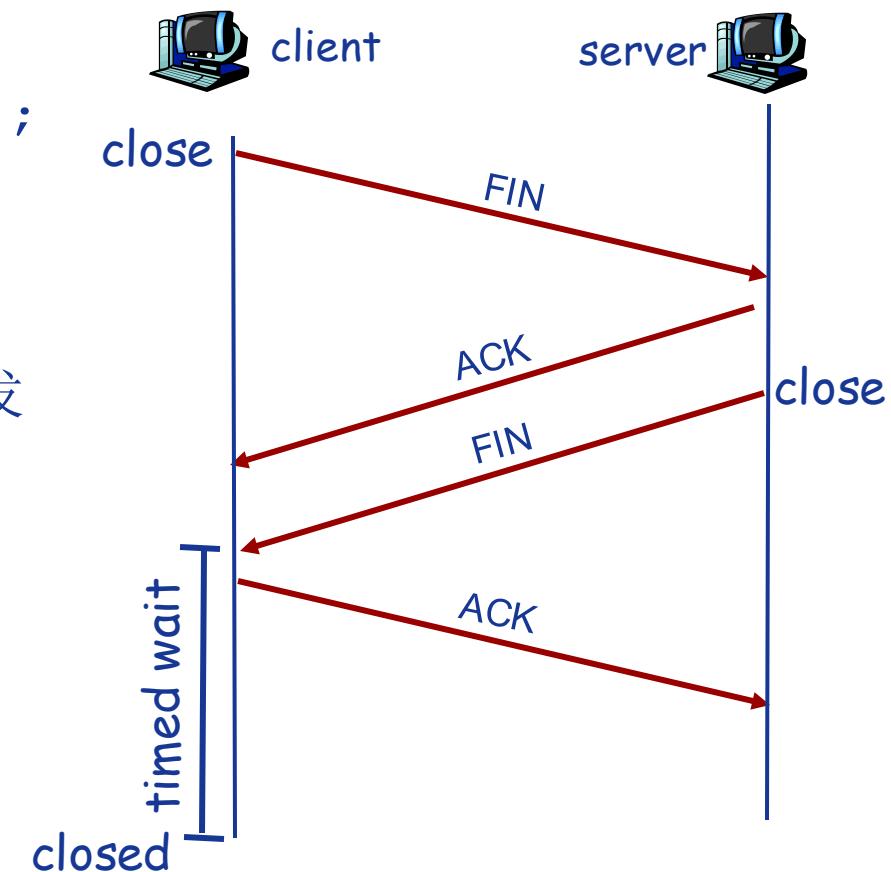
Step 1: client向server发送TCP FIN 控制
segment

Step 2: server 收到FIN, 回复ACK. 关闭连接, 发
送FIN.

Step 3: client 收到FIN, 回复ACK.

- 进入“等待” –如果收到FIN, 会重新发送
ACK

Step 4: server收到ACK. 连接关闭.





TCP连接管理：断连过程

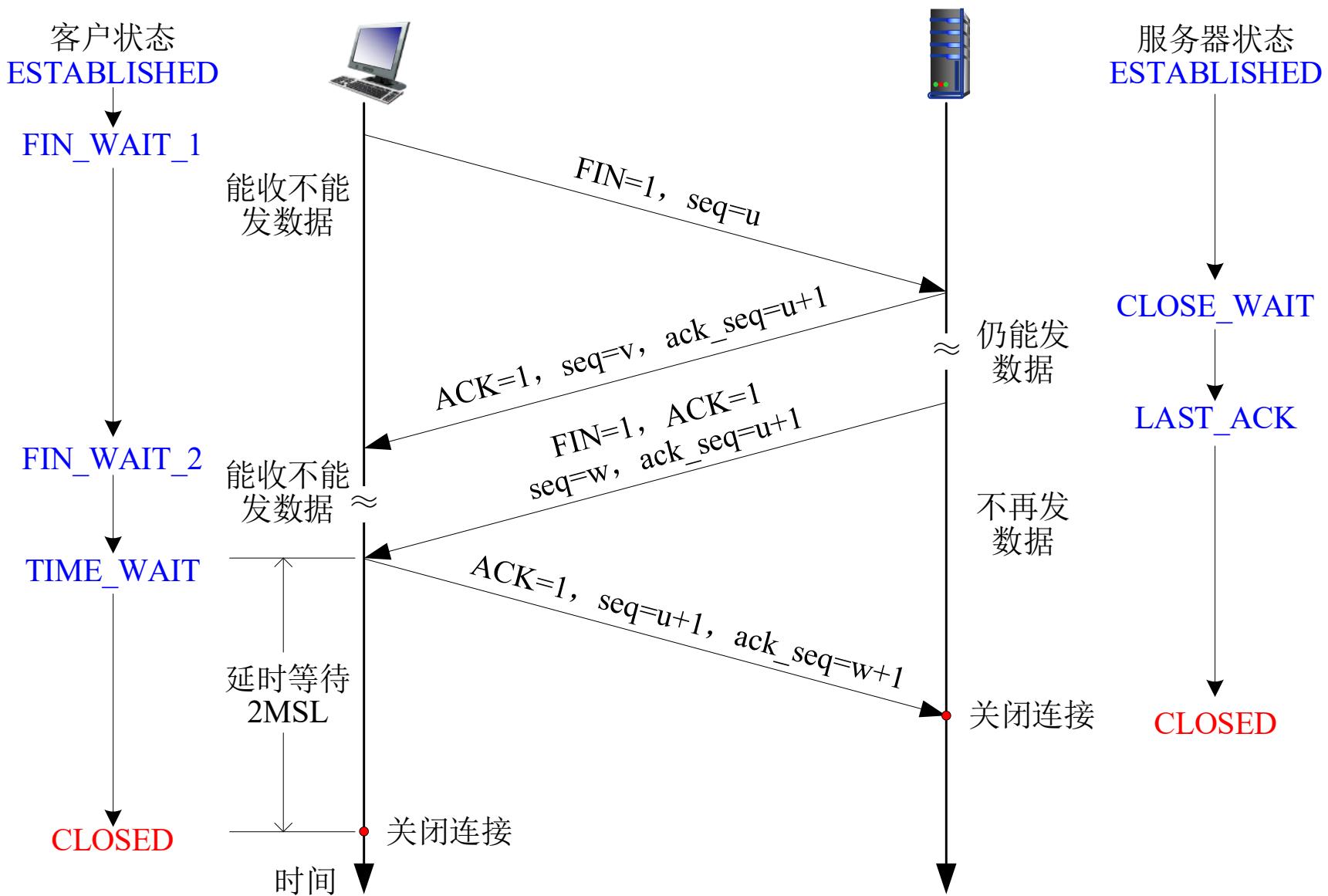
5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP连接管理





TCP连接管理

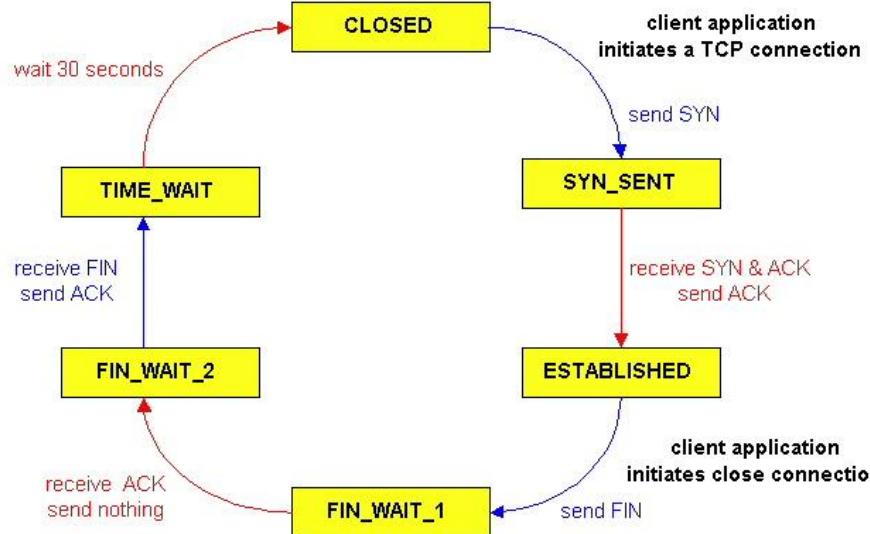
5.1 传输层服务

5.2 传输层多路复用/分解

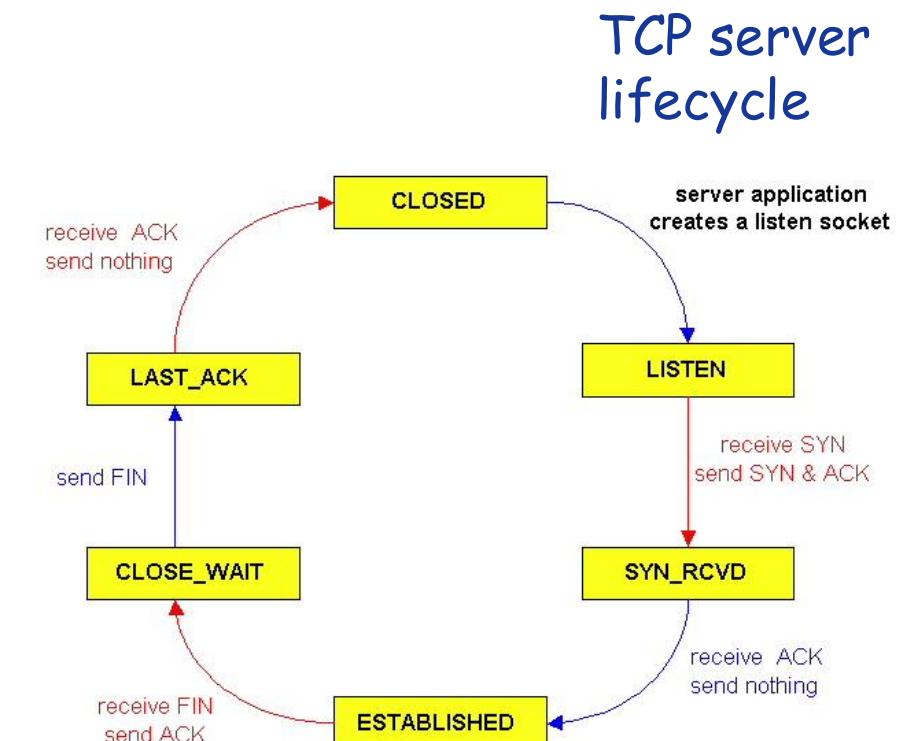
5.3 UDP协议

5.4 TCP协议

TCP连接管理



TCP client
lifecycle



单选题 1分

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP连接管理



若主机甲主动发起一个与主机乙的TCP连接，甲、乙选择的初始序列号分别为2018和2020，则第三次握手TCP段的确认序列号是

A 2019

B 2020

C 2021

D 2022

提交



TCP拥塞控制的基本原理

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP拥塞控制



❖ Sender限制发送速率

$$\frac{\text{LastByteSent} - \text{LastByteAcked}}{\text{CongWin}} \leq \text{CongWin}$$

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

❖ CongWin:

- 动态调整以改变发送速率
- 反映所感知到的网络拥塞

问题：如何感知网络拥塞？

❖ Loss事件=timeout或3个重复ACK

❖ 发生loss事件后，发送方降低速率

如何合理地调整发送速率？

❖ 加性增—乘性减: AIMD

❖ 慢启动: SS



加性增一乘性减: AIMD

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP拥塞控制

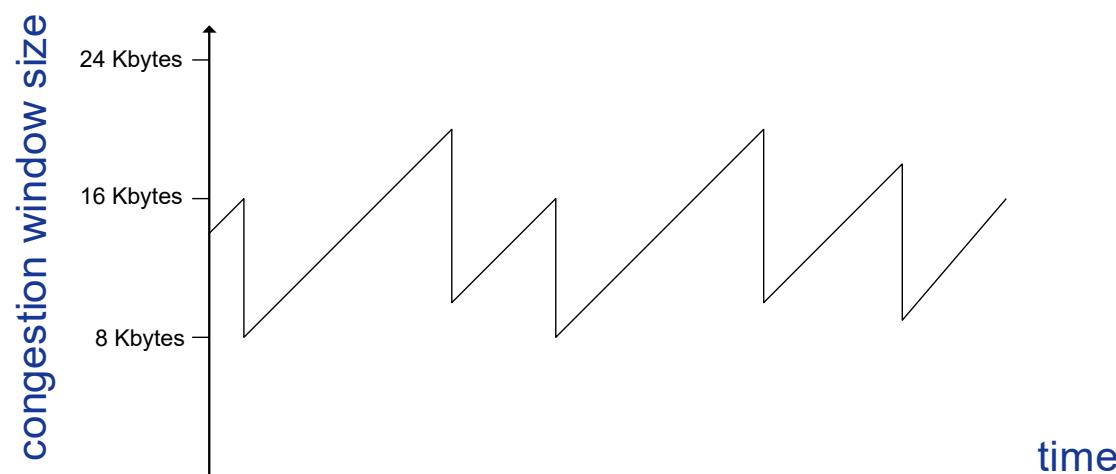


❖ **原理:** 逐渐增加发送速率, 谨慎探测可用带宽, 直到发生丢包

❖ **方法: AIMD**

- **Additive Increase:** 每个RTT将CongWin增大1个MSS-拥塞避免
- **Multiplicative Decrease:** 发生丢包后将CongWin减半

锯齿行为: 探测
可用带宽





TCP慢启动: SS

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP拥塞控制



❖ TCP连接建立时，

CongWin=1

- 例: MSS=500 byte,
RTT=200msec
- 初始速率=20k bps

❖ 可用带宽可能远远高于
初始速率:

- 希望快速增长

❖ 原理:

- 当连接开始时，指数性增长

Slowstart algorithm

```
initialize: Congwin = 1
for (each segment ACKed)
    Congwin++
until (loss event OR
      CongWin > threshold)
```



TCP慢启动: SS

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

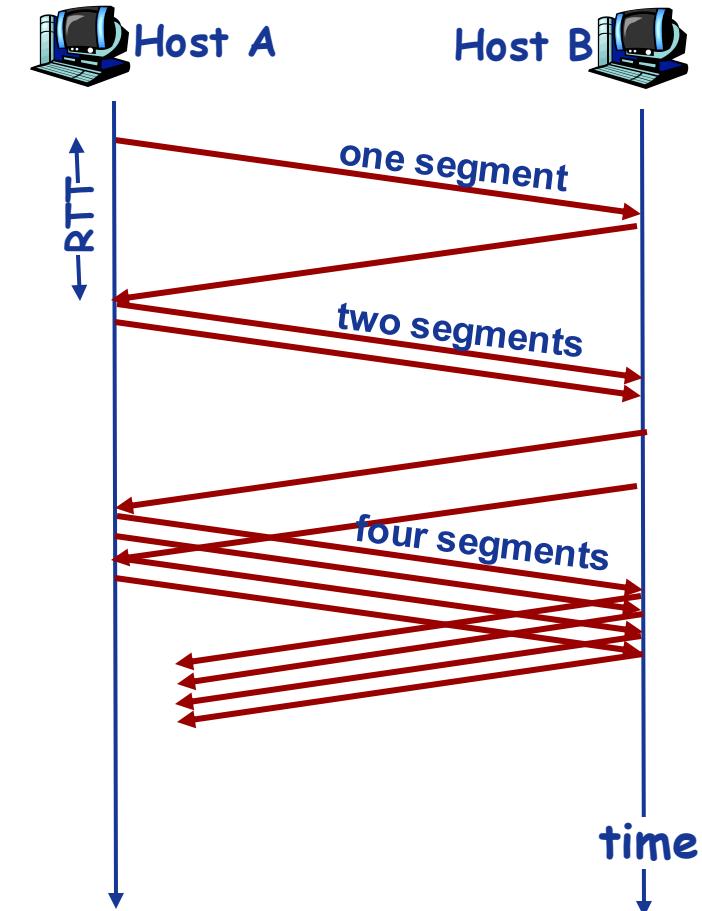
TCP拥塞控制



❖ 指数性增长

- 每个RTT将CongWin翻倍
- 收到每个ACK进行CongWin++操作

❖ 初始速率很慢，但是快速攀升





TCP慢启动: SS

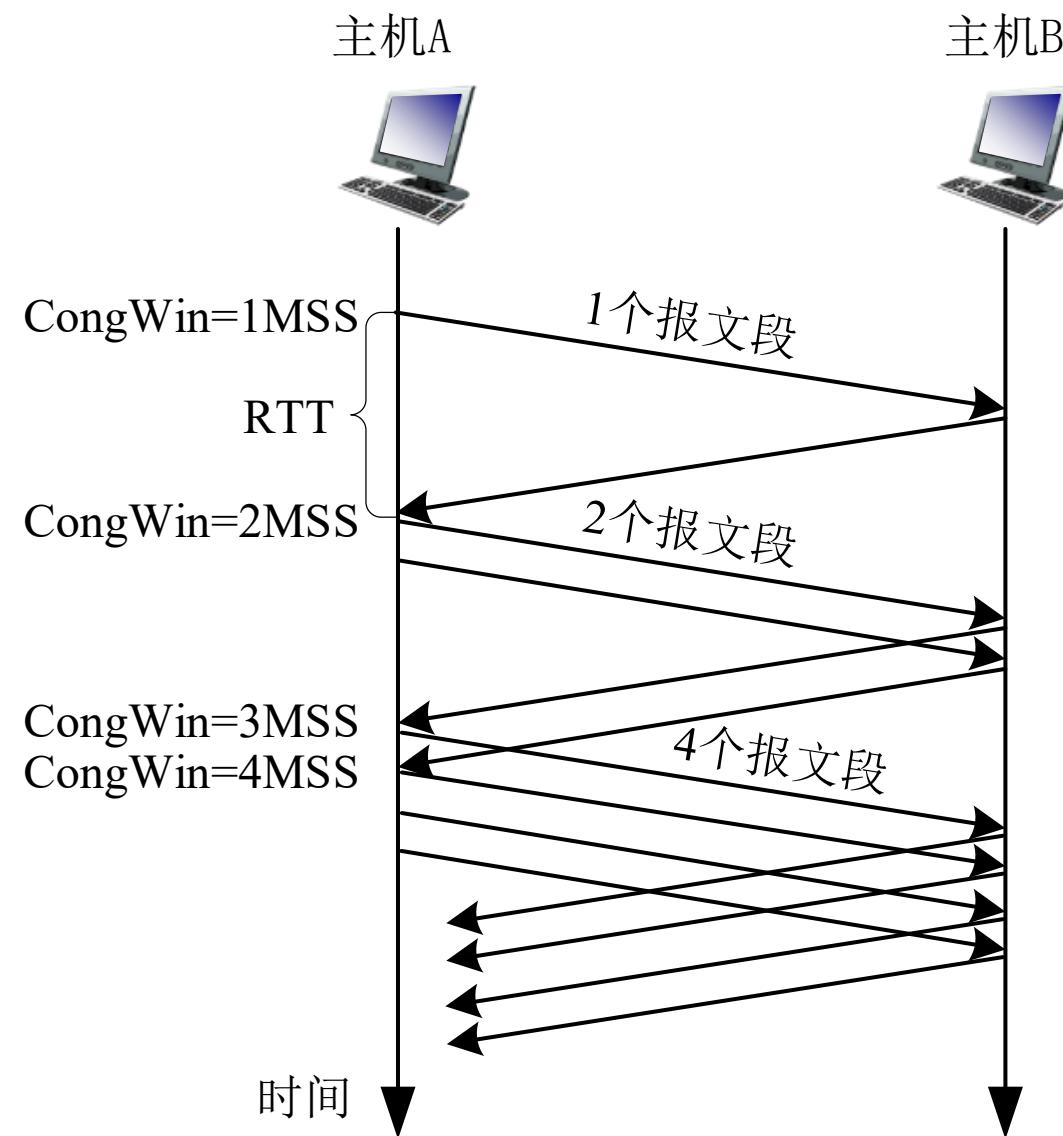
5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP拥塞控制





5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP拥塞控制



Threshold变量

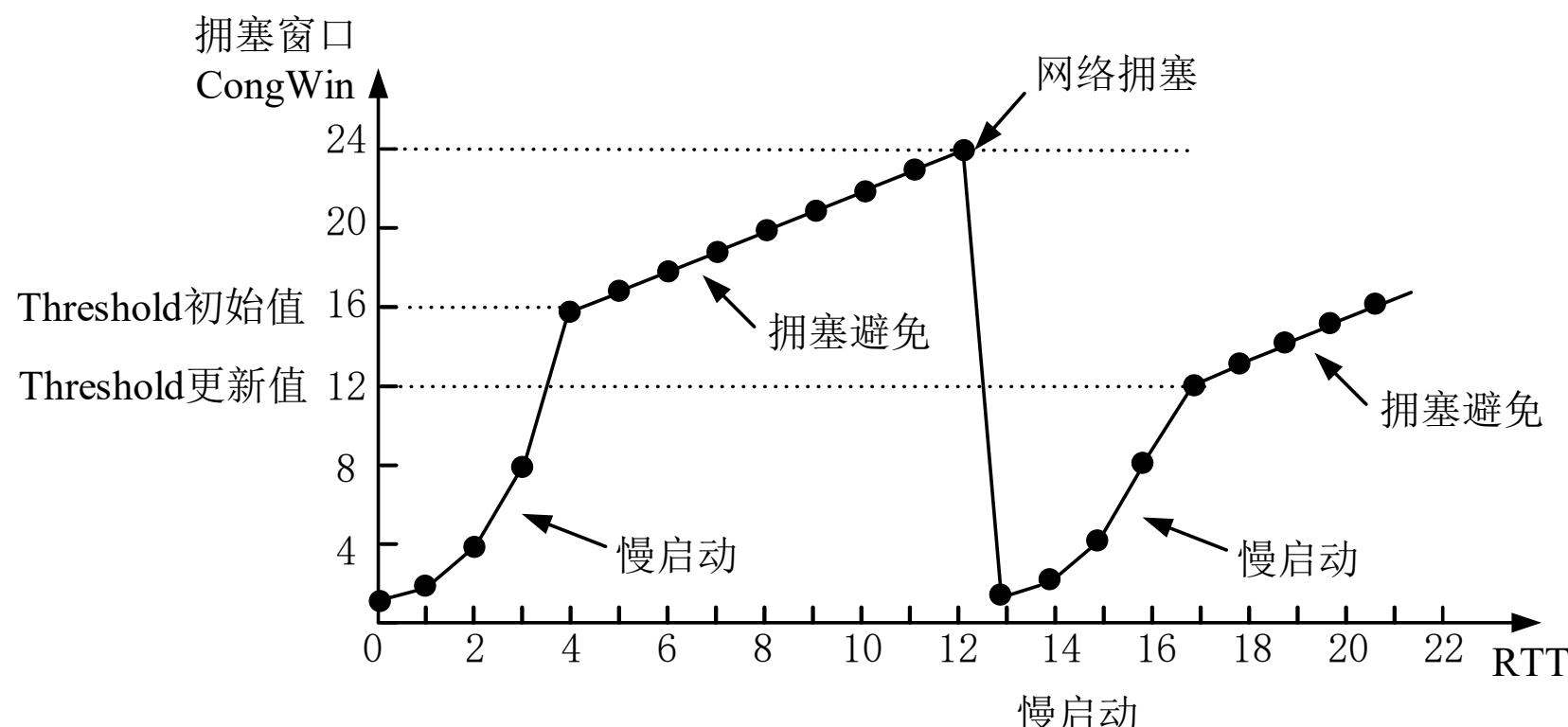
Q:何时应该指数性增长切换为线性增长(拥塞避免)?

A:当CongWin达到Loss事件前值的 $1/2$ 时.

实现方法:

❖ 变量 Threshold

❖ Loss事件发生时, Threshold 被设为Loss事件前CongWin 值的 $1/2$ 。





5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP拥塞控制

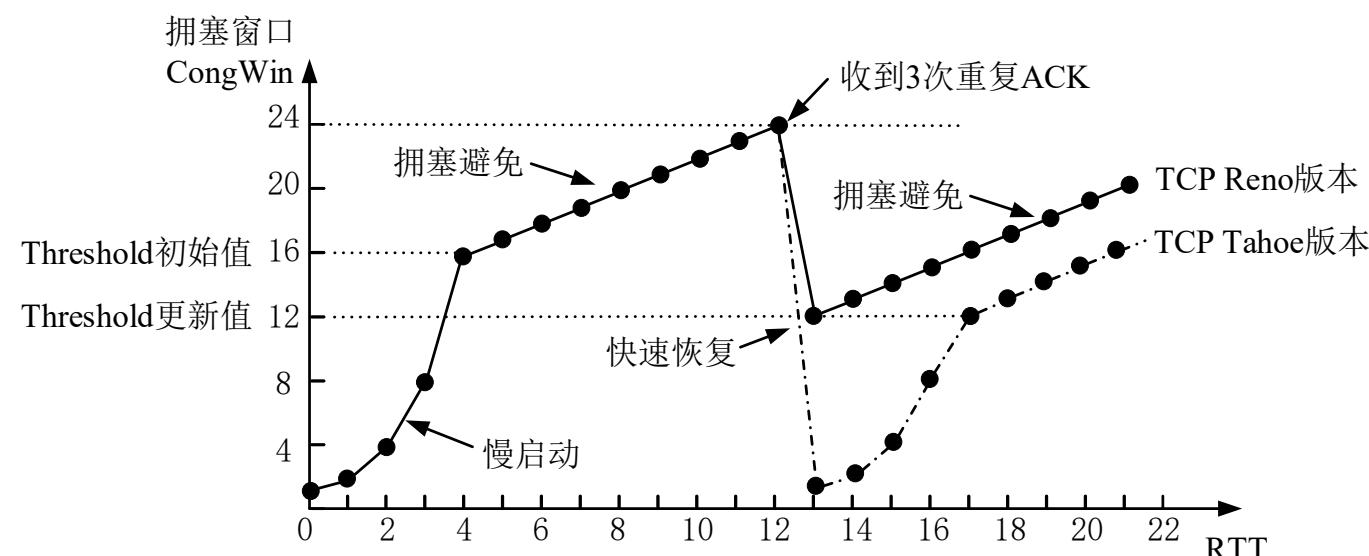


- ❖ 3个重复ACKs:
 - CongWin切换到一半
 - 然后线性增长
- ❖ Timeout事件:
 - CongWin直接设为1个MSS
 - 然后指数增长
 - 达到threshold后，再线性增长

LOSS事件的处理

Philosophy:

- 3个重复ACKs表示网络还能够传输一些 segments
- timeout事件表明拥塞更为严重





TCP拥塞控制：总结

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP拥塞控制



- ❖ When CongWin is below Threshold, sender in **slow-start phase**, window grows exponentially.
- ❖ When CongWin is above Threshold, sender is in **congestion-avoidance phase**, window grows linearly.
- ❖ When a **triple duplicate ACK** occurs, Threshold set to CongWin/2 and CongWin set to Threshold.
- ❖ When **timeout** occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS.



TCP拥塞控制

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP拥塞控制



State	Event	TCP Sender Action	Commentary
Slow Start (SS)	<i>ACK receipt for previously unacked data</i>	$CongWin = CongWin + MSS,$ <i>If (CongWin > Threshold) set state to "Congestion Avoidance"</i>	<i>Resulting in a doubling of CongWin every RTT</i>
Congestion Avoidance (CA)	<i>ACK receipt for previously unacked data</i>	$CongWin = CongWin + MSS * (MSS/CongWin)$	<i>Additive increase, resulting in increase of CongWin by 1 MSS every RTT</i>
SS or CA	<i>Loss event detected by triple duplicate ACK</i>	$Threshold = CongWin/2,$ $CongWin = Threshold,$ <i>Set state to "Congestion Avoidance"</i>	<i>Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.</i>
SS or CA	<i>Timeout</i>	$Threshold = CongWin/2,$ $CongWin = 1 MSS,$ <i>Set state to "Slow Start"</i>	<i>Enter slow start</i>
SS or CA	<i>Duplicate ACK</i>	<i>Increment duplicate ACK count for segment being acked</i>	<i>CongWin and Threshold not changed</i>



TCP拥塞控制算法

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP拥塞控制



```
Th = ?  
CongWin = 1 MSS  
/* slow start or exponential increase */  
While (No Packet Loss and CongWin < Th) {  
    send CongWin TCP segments  
    for each ACK increase CongWin by 1  
}  
/* congestion avoidance or linear increase */  
While (No Packet Loss) {  
    send CongWin TCP segments  
    for CongWin ACKs, increase CongWin by 1  
}  
Th = CongWin/2  
If (3 Dup ACKs) CongWin = Th;  
If (timeout) CongWin=1;
```



例题

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP拥塞控制



【例1】一个TCP连接总是以1 KB的最大段长发送TCP段，发送方有足够的多的数据要发送。当拥塞窗口为16 KB时发生了超时，如果接下来的4个RTT（往返时间）时间内的TCP段的传输都是成功的，那么当第4个RTT时间内发送的所有TCP段都得到肯定应答时，拥塞窗口大小是多少？

【解】 $\text{threshold} = 16/2 = 8 \text{ KB}$, $\text{CongWin} = 1 \text{ KB}$, 1个RTT后,
 $\text{CongWin} = 2 \text{ KB}$, 2个RTT后, $\text{CongWin} = 4 \text{ KB}$, 3个RTT后,
 $\text{CongWin} = 8 \text{ KB}$, Slowstart is over; 4个RTT后, $\text{CongWin} = 9 \text{ KB}$

单选题 1分

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP拥塞控制



主机甲和主机乙之间已建立了一个TCP连接，TCP最大段长度为1000字节。若主机甲的当前拥塞窗口为4 000字节，在主机甲向主机乙连续发送两个最大段后，成功收到主机乙发送的对第一个段的确认段，确认段中通告的接收窗口大小为2000字节，则此时主机甲还可以向主机乙发送的最大字节数是

- A 1000
- B 2000
- C 3000
- D 4000

提交



TCP的吞吐率

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP吞吐率分析



- ❖ 给定拥塞窗口大小和RTT， TCP的平均吞吐率是多少？
 - 忽略掉Slow start
- ❖ 假定发生超时时CongWin的大小为W， 吞吐率是W/RTT
- ❖ 超时后， CongWin=W/2， 吞吐率是W/2RTT
- ❖ 平均吞吐率为： $0.75W/RTT$



TCP的吞吐率

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP吞吐率分析



- ❖ 举例：每个Segment有1500个byte, RTT是100ms, 希望获得10Gbps的吞吐率
 - $\text{throughput} = W * \text{MSS} * 8 / \text{RTT}$, 则
 - $W = \text{throughput} * \text{RTT} / (\text{MSS} * 8)$
 - $\text{throughput} = 10\text{Gbps}$, 则 $W = 83,333$
- ❖ 窗口大小为83,333



5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP吞吐率分析



TCP的吞吐率

❖ 吞吐率与丢包率(loss rate, L)的关系

- CongWin从 $W/2$ 增加至 W 时出现第一个丢包，那么一共发送的分组数为

$$W/2 + (W/2+1) + (W/2+2) + \dots + W = 3W^2/8 + 3W/4$$

- W 很大时， $3W^2/8 \gg 3W/4$ ，因此 $L \approx 8/(3W^2)$

$$W = \sqrt{\frac{8}{3L}} \quad \text{Throughput} = \frac{0.75 \cdot MSS \cdot \sqrt{\frac{8}{3L}}}{RTT} \approx \frac{1.22 \cdot MSS}{RTT \sqrt{L}}$$

❖ $L = 2 \cdot 10^{-10}$ **Wow!!!**

❖ 高速网络下需要设计新的TCP



TCP拥塞控制的改进

5.1 传输层服务

5.2 传输层多路复用/分解

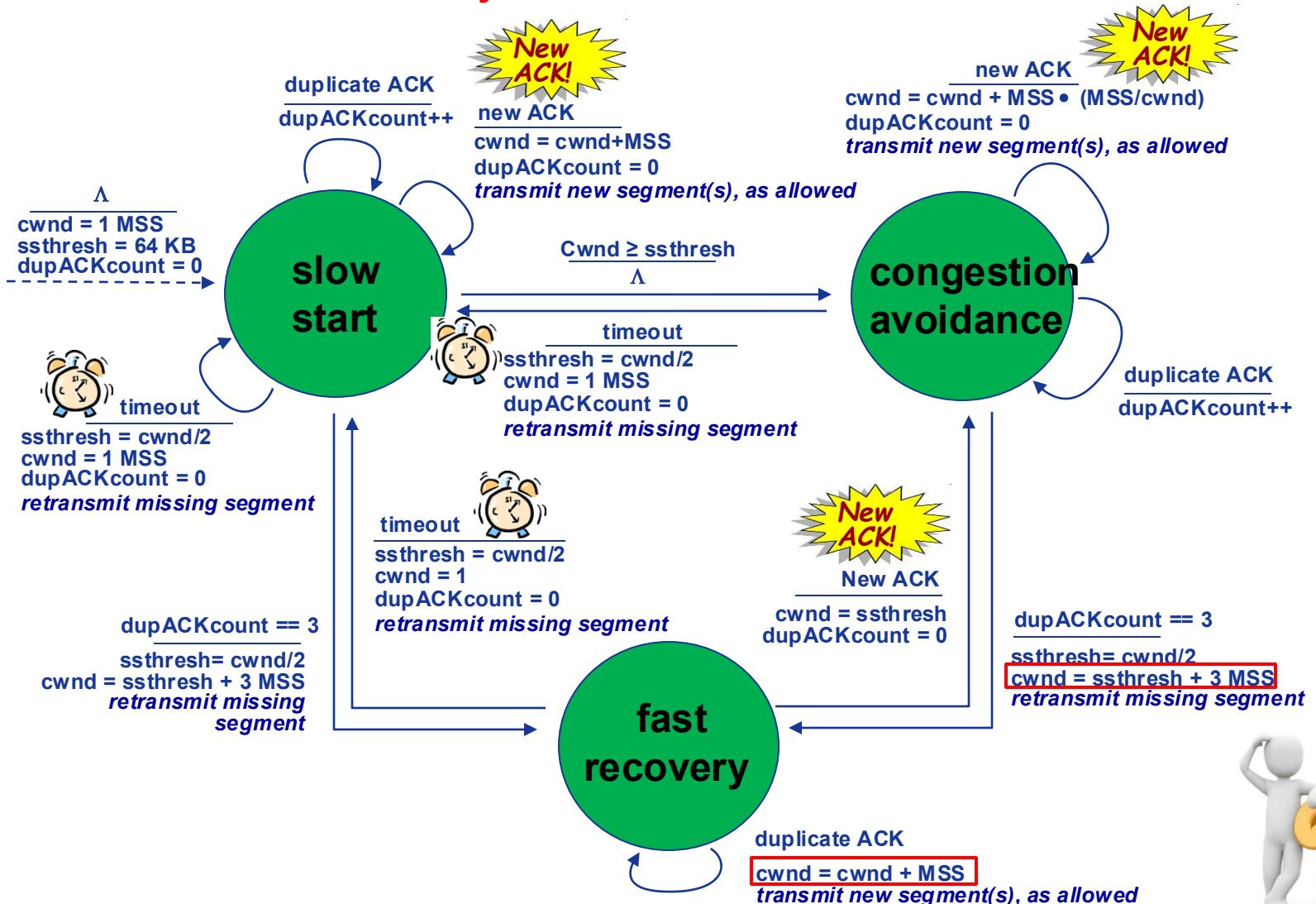
5.3 UDP协议

5.4 TCP协议

TCP拥塞控制改进



❖ 快速回复 (fast recovery)





TCP拥塞控制的改进

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

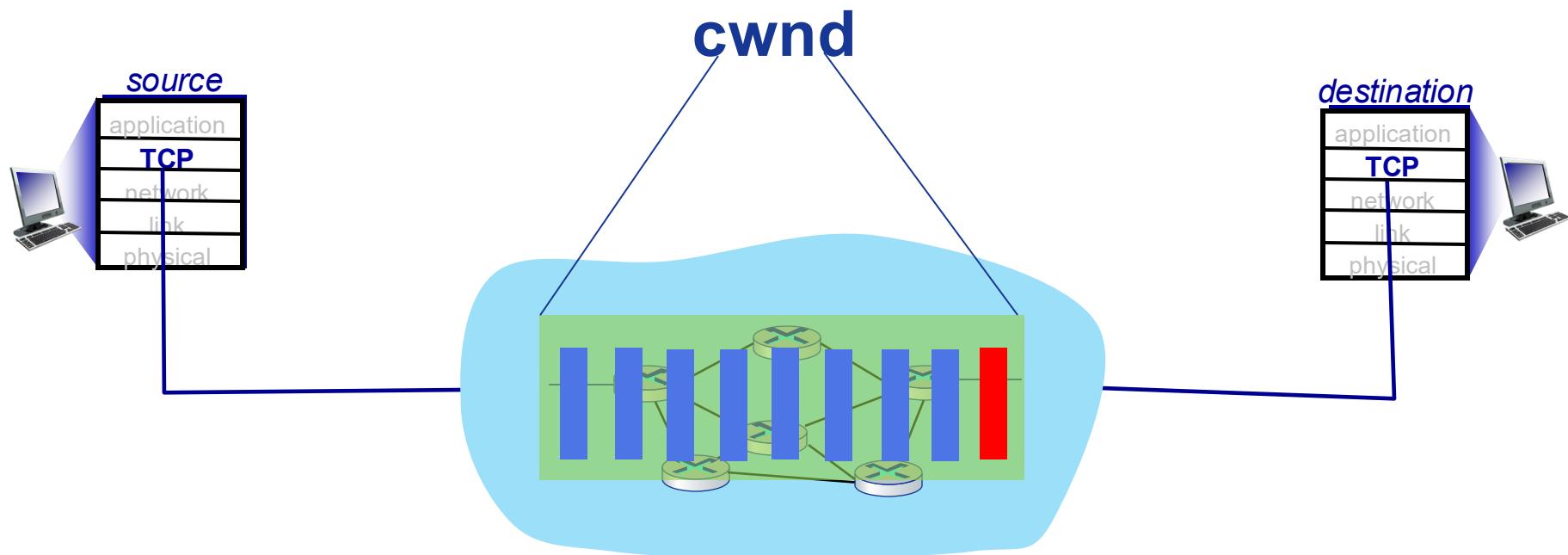
5.4 TCP协议

TCP拥塞控制改进



❖ 快速回复 (fast recovery)

- 为什么窗口要膨胀?
- 为什么会出现3次重复确认?





TCP拥塞控制的改进

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

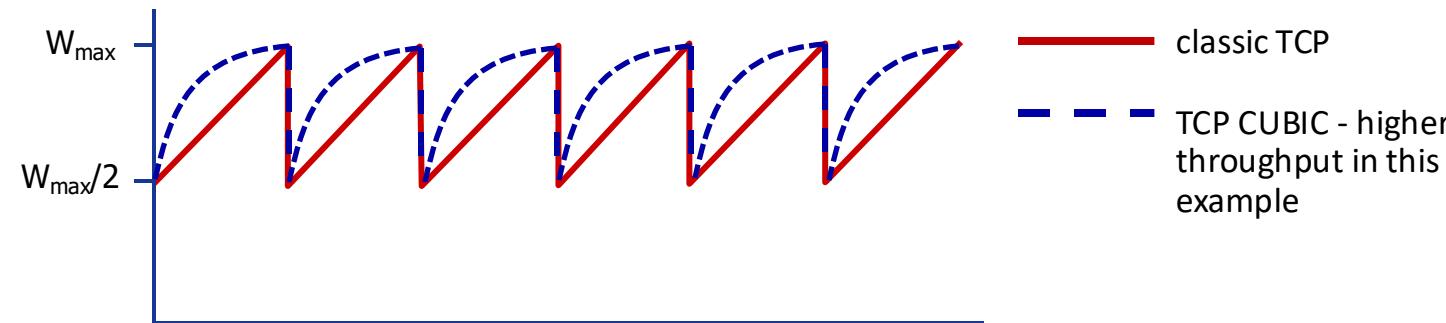
5.4 TCP协议

TCP拥塞控制改进



❖ TCP CUBIC

- Is there a better way than AIMD to “probe” for usable bandwidth?
- Insight/intuition:
 - W_{\max} : sending rate at which congestion loss was detected
 - congestion state of bottleneck link probably (?) hasn't changed much
- after cutting rate/window in half on loss, initially ramp to W_{\max} *faster*, but then approach W_{\max} more *slowly*





5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

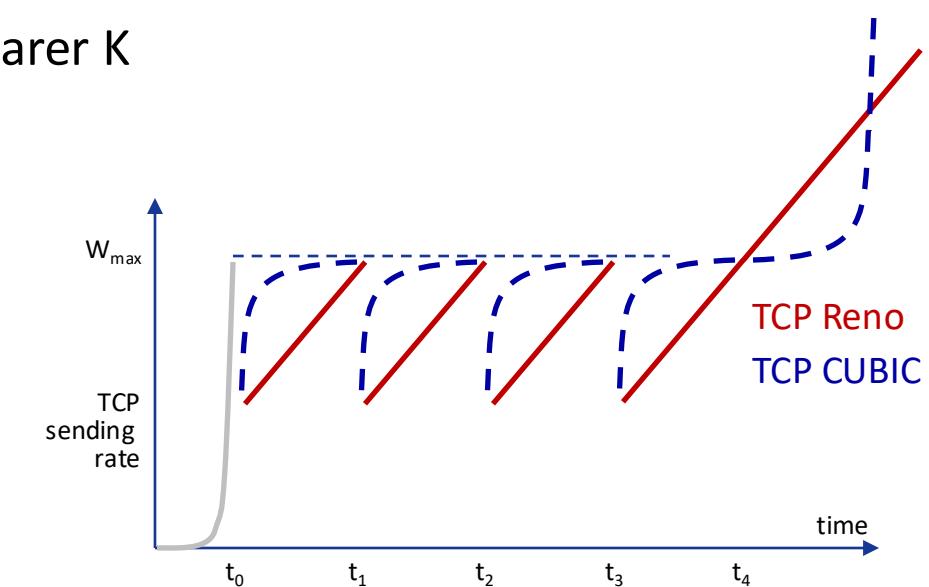
TCP拥塞控制改进



TCP拥塞控制的改进

❖ TCP CUBIC

- K: point in time when TCP window size will reach W_{\max}
 - K itself is tuneable
- increase W as a function of the *cube* of the distance between current time and K
 - larger increases when further away from K
 - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux,
most popular TCP for popular
Web servers





TCP拥塞控制的改进

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

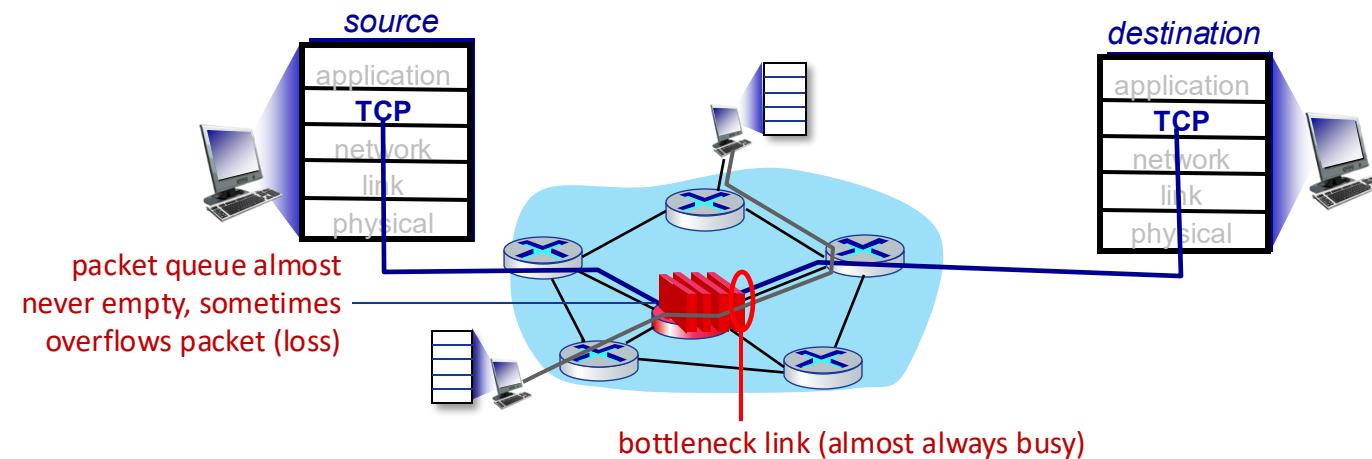
5.4 TCP协议

TCP拥塞控制改进



❖ Delay-based TCP congestion control

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*





TCP拥塞控制的改进

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

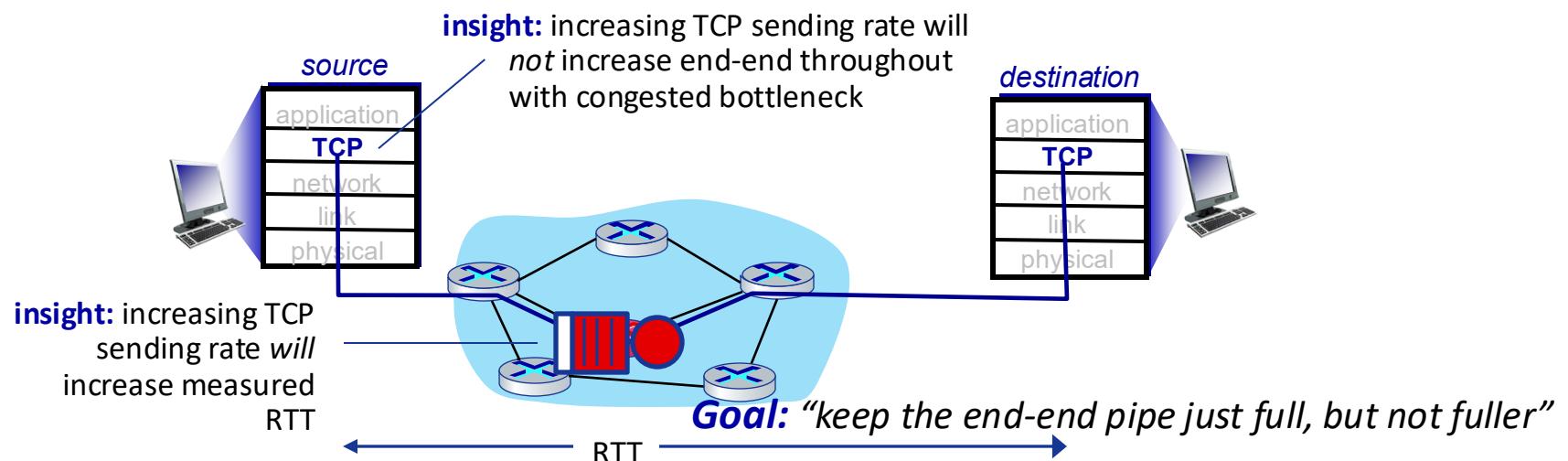
5.4 TCP协议

TCP拥塞控制改进



❖ Delay-based TCP congestion control

- TCP (classic, CUBIC) increase TCP's sending rate until packet loss occurs at some router's output: the *bottleneck link*
- understanding congestion: useful to focus on congested bottleneck link





TCP拥塞控制的改进

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

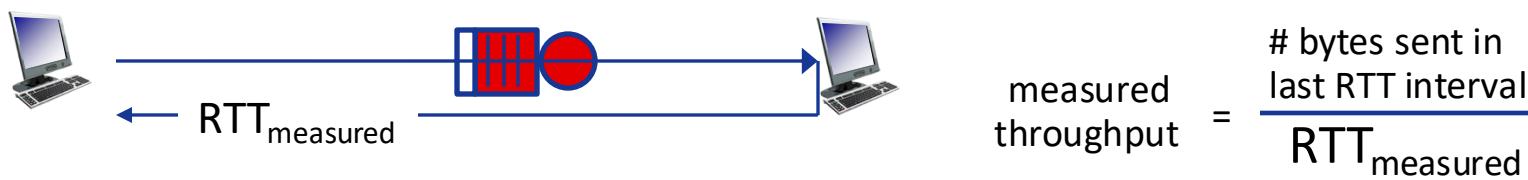
5.4 TCP协议

TCP拥塞控制改进



❖ Delay-based TCP congestion control

- Keeping sender-to-receiver pipe “just full enough, but no fuller”: keep bottleneck link busy transmitting, but avoid high delays/buffering



Delay-based approach:

- RTT_{\min} - minimum observed RTT (uncongested path)
- uncongested throughput with congestion window $cwnd$ is $cwnd/\text{RTT}_{\min}$

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



TCP拥塞控制的改进

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

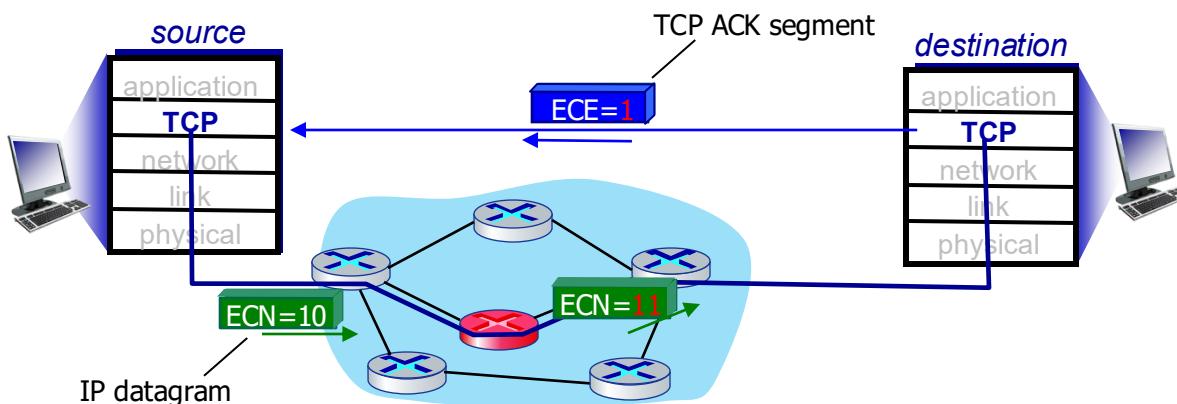
TCP拥塞控制改进



❖ Explicit congestion notification (ECN)

TCP deployments often implement *network-assisted* congestion control:

- two bits in IP header (ToS field) marked *by network router* to indicate congestion
 - policy to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)





TCP的公平性

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

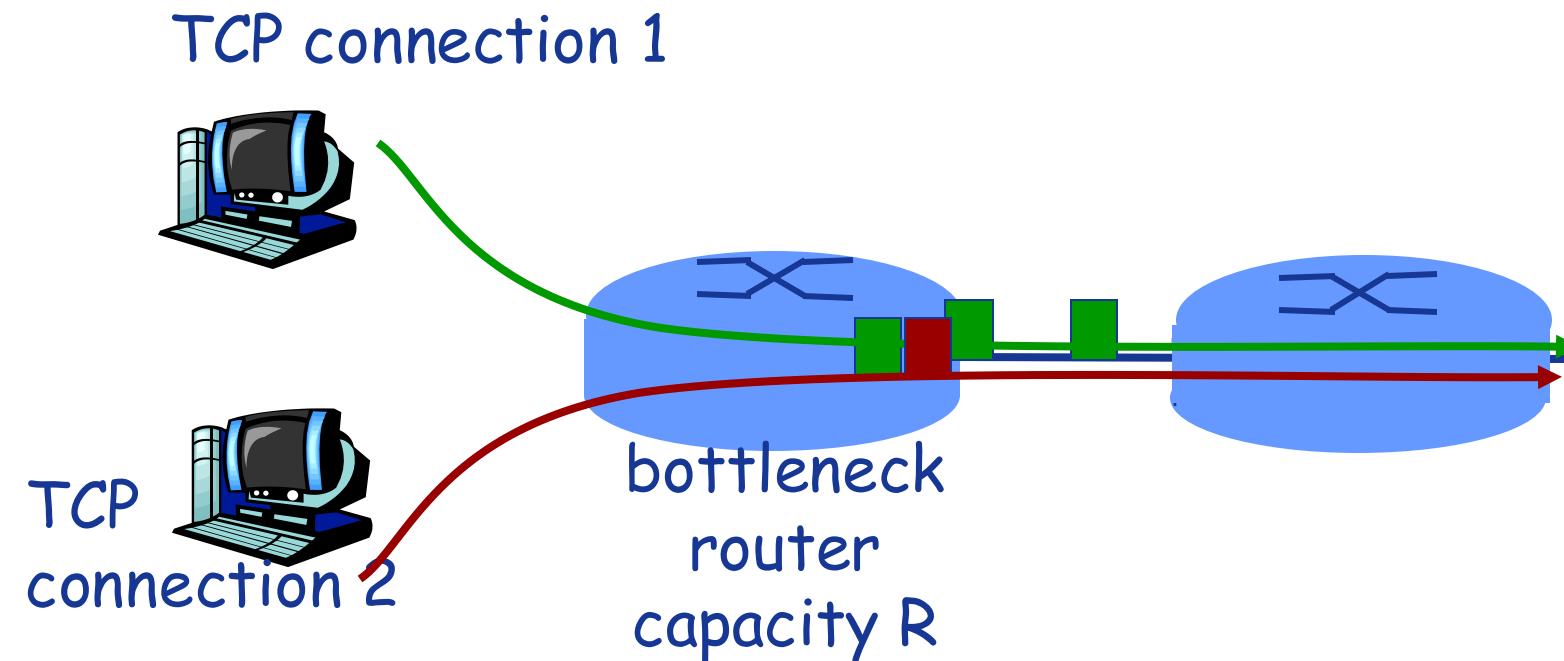
5.4 TCP协议

TCP性能分析



❖ 公平性?

- 如果K个TCP Session共享相同的瓶颈带宽R，那么每个Session的平均速率为 R/K





TCP具有公平性吗？

5.1 传输层服务

5.2 传输层多路复用/分解

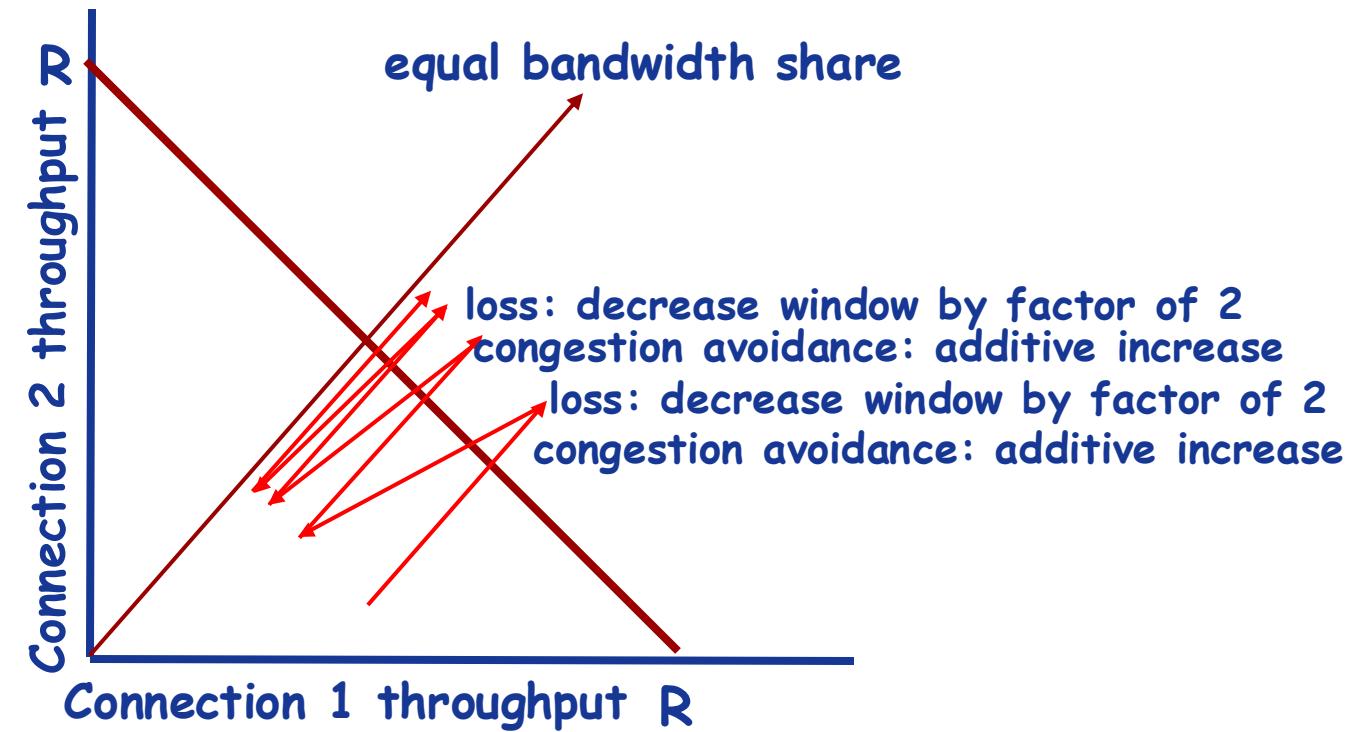
5.3 UDP协议

5.4 TCP协议

TCP性能分析



❖ 是的！





TCP的公平性

5.1 传输层服务

5.2 传输层多路复用/分解

5.3 UDP协议

5.4 TCP协议

TCP性能分析



❖ 公平性与UDP

- 多媒体应用通常不使用TCP，以免被拥塞控制机制限制速率
- 使用UDP：以恒定速率发送，能够容忍丢失
- 产生了不公平

❖ 研究：TCP friendly

❖ 公平性与并行TCP连接

- 某些应用会打开多个并行连接
 - Web浏览器
 - 产生公平性问题
- ❖ 例子：链路速率为R，已有9个连接
- 若新的应用请求建立1个TCP连接，则获得 $R/10$ 的速率
 - 若新的应用请求建立11个TCP连接，则获得 $R/2$ 的速率



哈爾濱工業大學
HARBIN INSTITUTE OF TECHNOLOGY

立足航天，服务国防，面向国民经济主战场



谢 谢！