

计算机网络

端到端协议

华中科技大学电信学院 2016



学习目标

- 理解进程间通信的协议设计需求；
- 理解实现简单多路分解功能的传输协议UDP的原理；
- 掌握实现可靠字节流服务的传输协议TCP的设计原理，理解互联网架构设计的端到端设计原则；
- 掌握TCP建立连接的三次握手机制；
- 理解在互联网上实现传输层可靠传输的技术挑战，理解TCP滑动窗口协议和数据链路层滑动窗口协议设计的区别。

提纲

引言

- 核心问题：进程间如何通信
- 简单多路分解(UDP)
- 可靠字节流(TCP)
- 总结



回顾：需求分析

我们对网络的期望是什么？

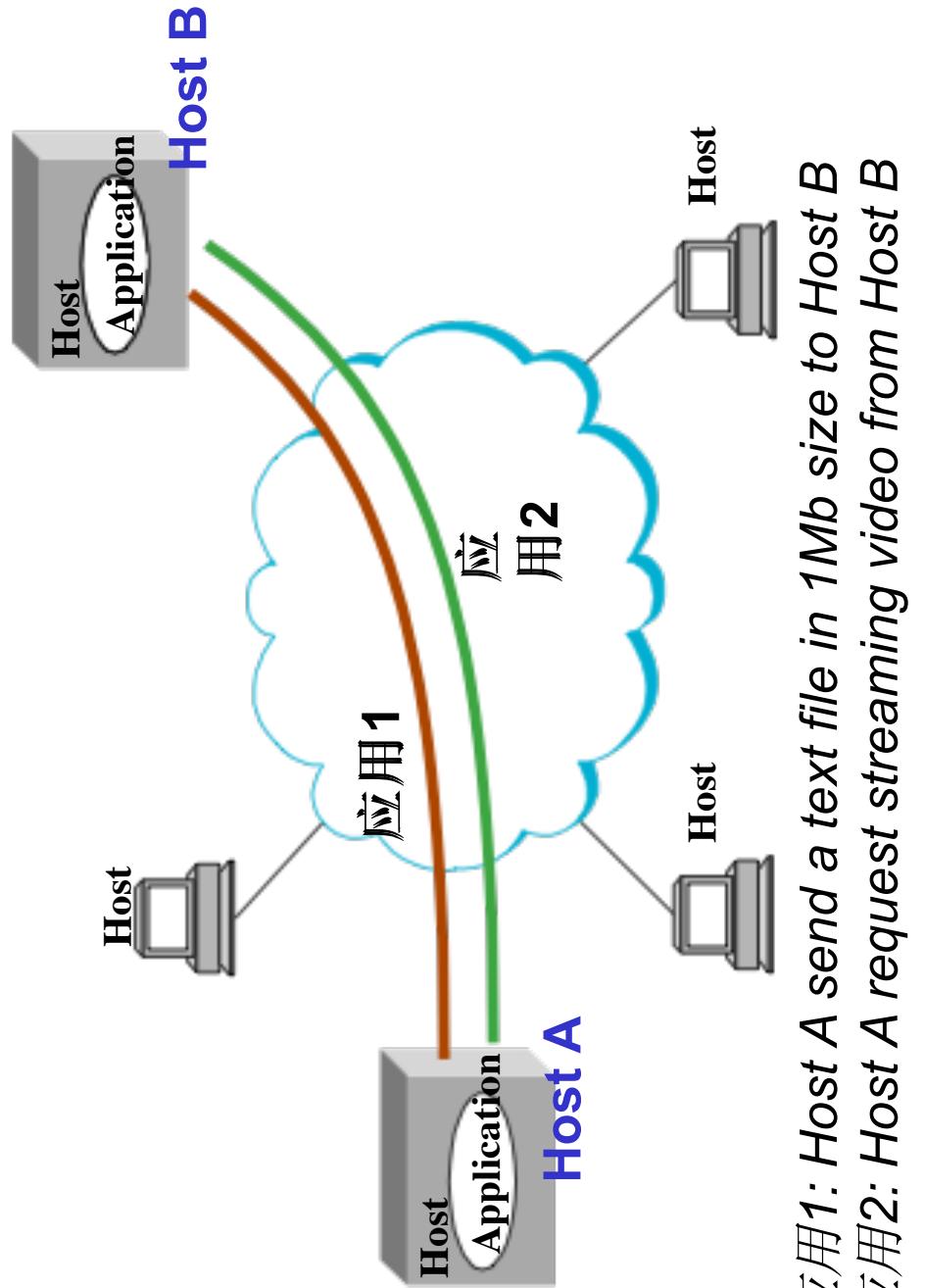
- 需求1：可扩展的连通性
- 节点和链路嵌套互联
- 需求2：高性价比的资源共享
- 采用统计复用方式共享硬件基础设施
- 基于突发流量特性，构建分组交换网络
- 需求3：支持通用服务
- 进程之间的通信服务



支持通用服务



- **目标 1:** 网络支持各种不同的应用
- **动机 1:** 简化目标, 对大多数的应用需求进行分类, 并提供相应的通用服务.

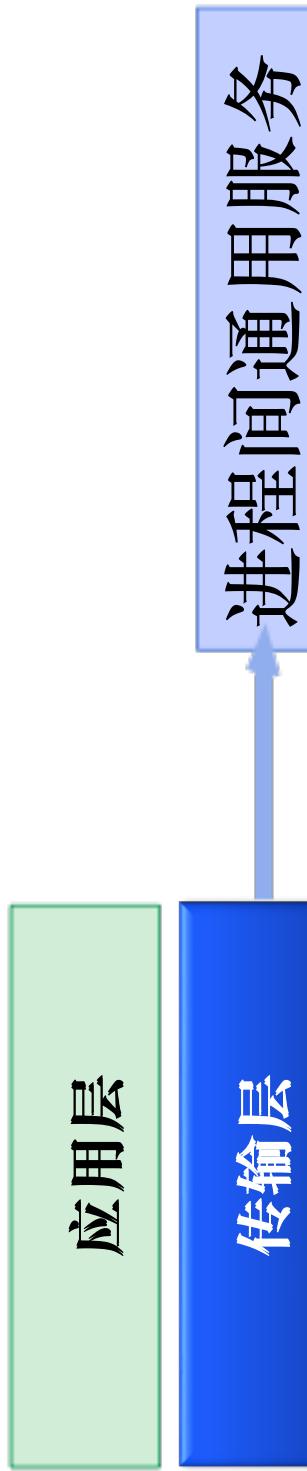


支持通用服务



- **目标2:** 网络在现实网络条件下支持各种不同的应用
- **动机2:** 提供可靠的消息传递,能够对故障进行分类处理
- 三类故障
 - 比特错误(比特级)
 - 外部电磁干扰等
 - 分组丢失(分组级)
 - 内存溢出,或分组出现了不可纠正的比特错误
 - 如何区分分组是丢失还是延迟到达?
 - 链路故障/节点当机(链路/节点级)
 - 如何区分主机故障还是运行速度慢?
 - 屏蔽部分故障
 - 使得网络对于应用程序而言具有更强的可靠性

问题：进程间通信



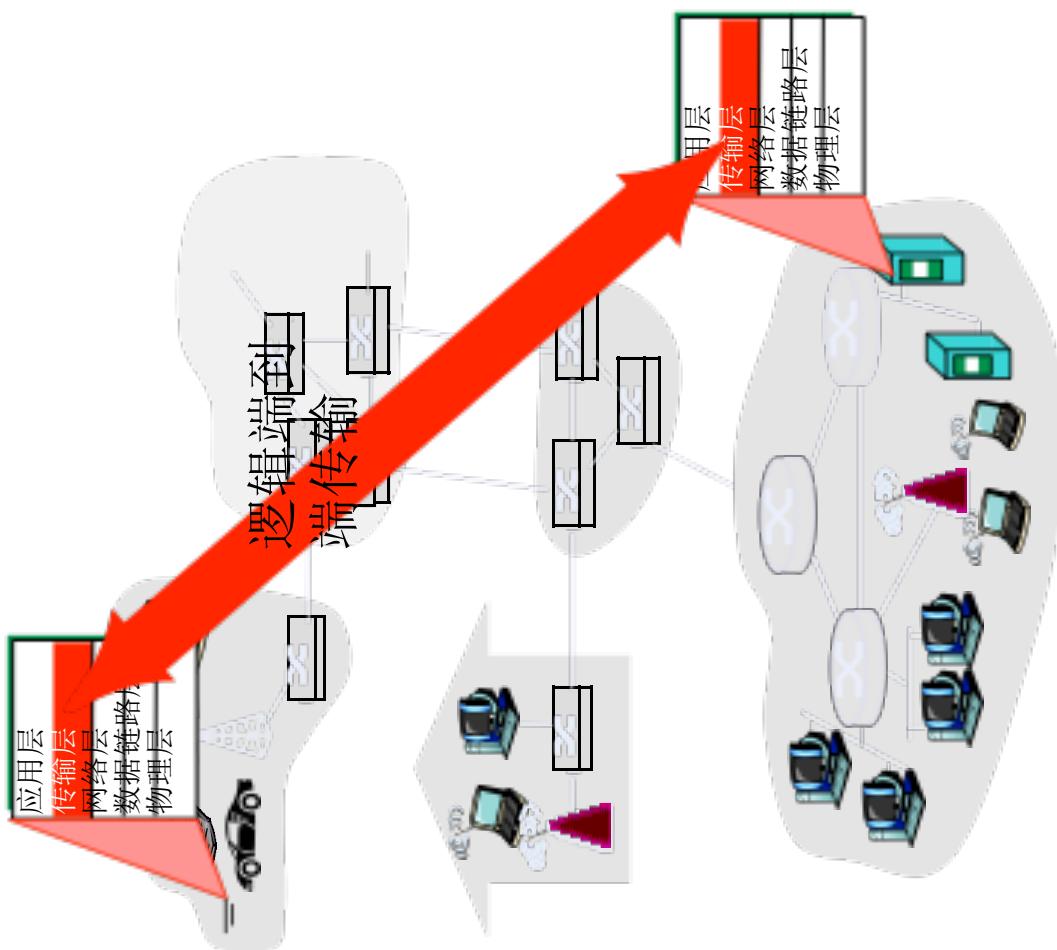
主机之间的连通性

- 链路层的互联
- 直连链路或分组交换网络

- 网络层的互联
- 主机到主机的通信协议
- 网络互联的异构性和扩展性问题
- IP服务模型及相关协议

传输服务和协议

- 为不同主机上运行的应用进程间提供**逻辑连接**在终端系统上运行的传输协议：
- 发送方：把应用层消息划分成**报文段**，并传送给网络层
- 接收方：将报文重组成消息，并传送给应用层
- 多个传输协议广泛应用：
 - Internet: TCP 和 UDP
 -





核心问题

进程间如何通信

提纲

- 引言
- 核心问题：进程间如何通信
 - 简单多路分解(UDP)
 - 可靠字节流(TCP)
- 总结

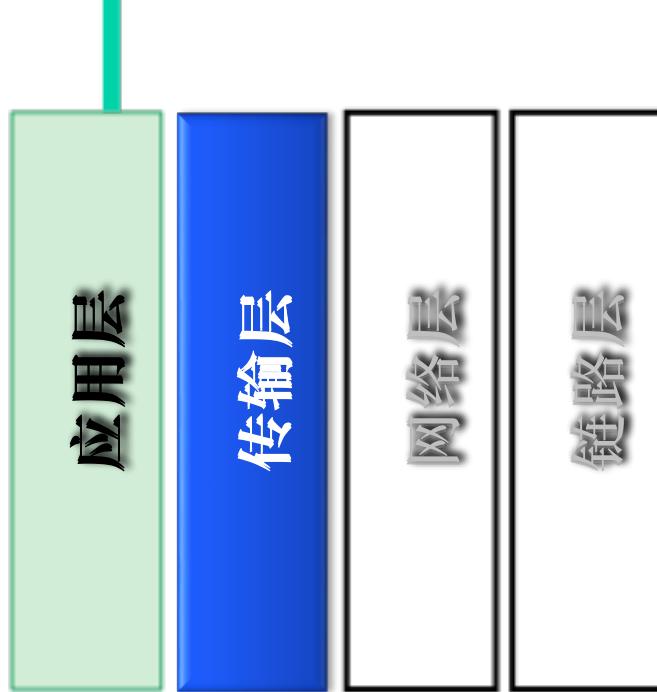


传输层：上层协议的观点



- 应用层进程期望传输层提供的特性：

- 保证消息的传输
- 传送过程中保证消息发送时的顺序
- 最多传送每个消息的一个副本
- 支持任意大的消息
- 支持发送方与接收方之间的同步
- 允许接收方对发送方进行流量控制
- 支持每台主机上的多个应用进程



传输层：下层网络的观点

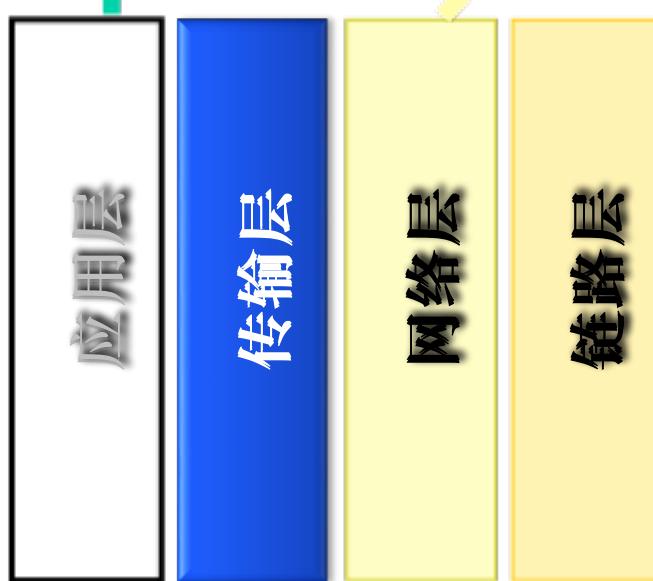


- 应用层进程期望传输层提供的特性：

- 保证消息的传输
- 传送过程中保证消息发送时的顺序
- 最多传送每个消息的一个副本
- 支持任意大的消息
- 支持发送方与接收方之间的同步
- 允许接收对方对发送方进行流量控制
- 支持每台主机上的多个应用进程……

- 底层网络可以提供的服务：

- 不可靠的分组传送
 - 丢弃
 - 乱序
 - 多个副本
- 不确定的时延
- 有限大小的分组



传输层：面临的风险



-0-

应用层进程期望传输层提供的特性：

- 支持多个应用进程
- 可靠的消息传递
- 流量控制
- 任意大小的分组长度

端到端协议

• 挑战

- 把底层网络低于要求的特性转变成为
- 应用程序所需的高级服务

应用层

传输层

网络层

链路层

• 底层网络可以提供的服务：

- 不可靠的分组传送(丢弃, 乱序, 多个副本)
- 不确定的时延
- 有限大小的分组

传输层协议



- 本章主要考虑四种有代表性的服务
 - 简单的异步解多路复用服务
 - UDP (用户数据报协议)
 - 可靠地字节流服务
 - TCP (传输控制协议)
 - ~~请求/响应服务~~
 - 远程过程调用(RPC)
 - SunRPC 和 DCE RPC
 - ~~实时传输服务~~
 - RTP

Multiplexing/demultiplexing

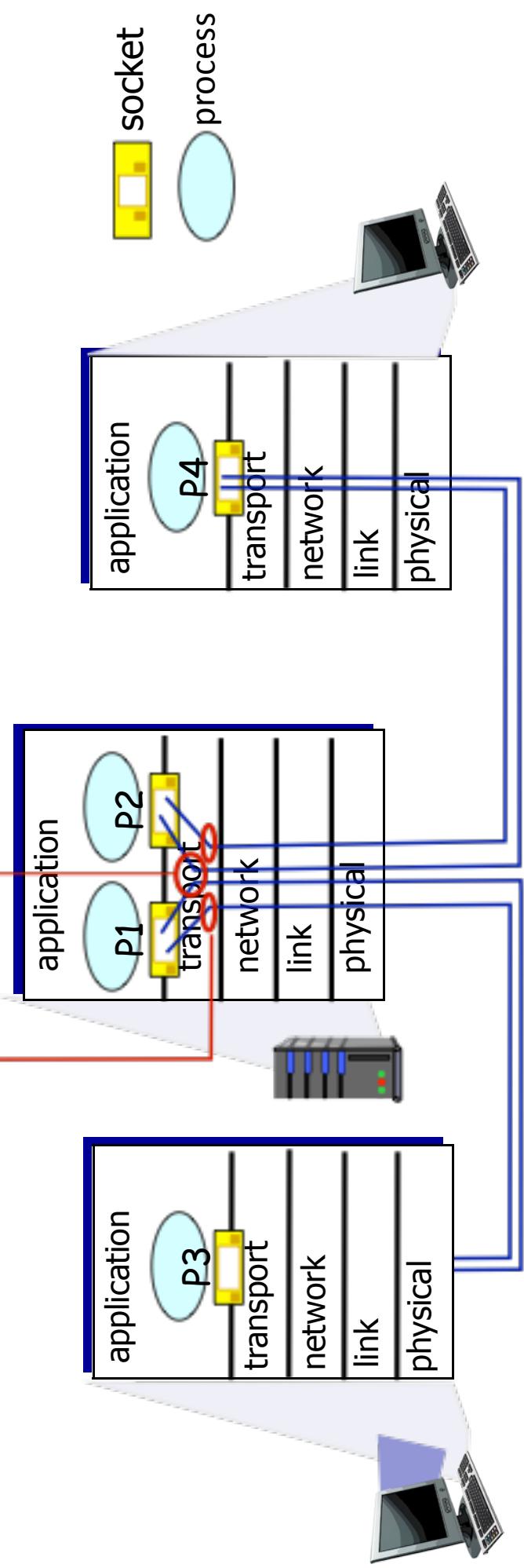


multiplexing at sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

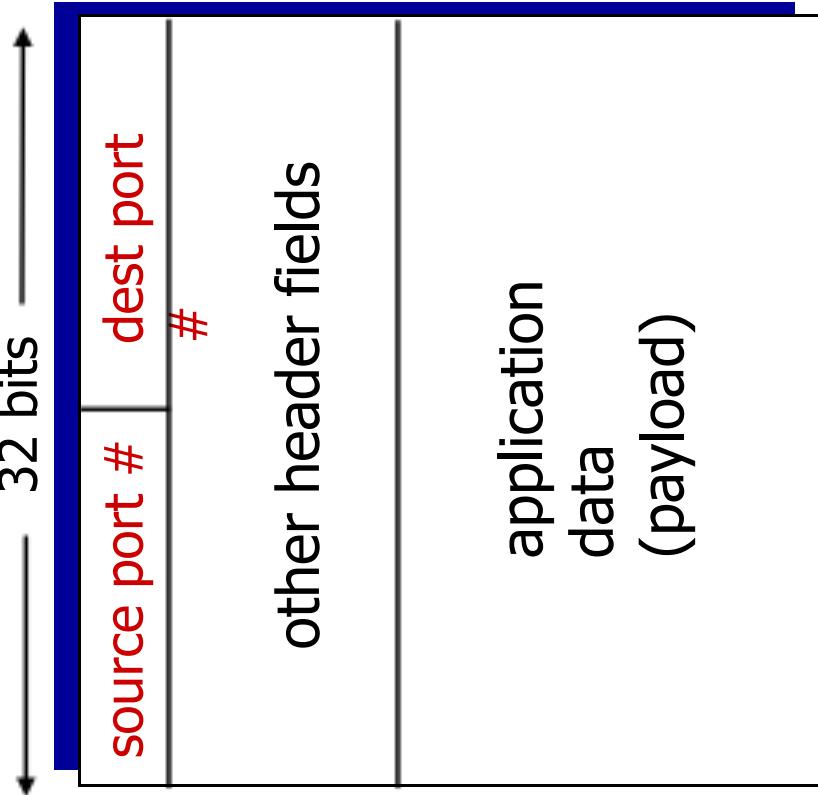
demultiplexing at receiver:

use header info to deliver received segments to correct socket



How demultiplexing works

- host receives IP datagrams
 - each datagram has source IP address, destination IP address
 - each datagram carries one transport-layer segment
 - each segment has source, destination port number
 - host uses ***IP addresses & port numbers*** to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing



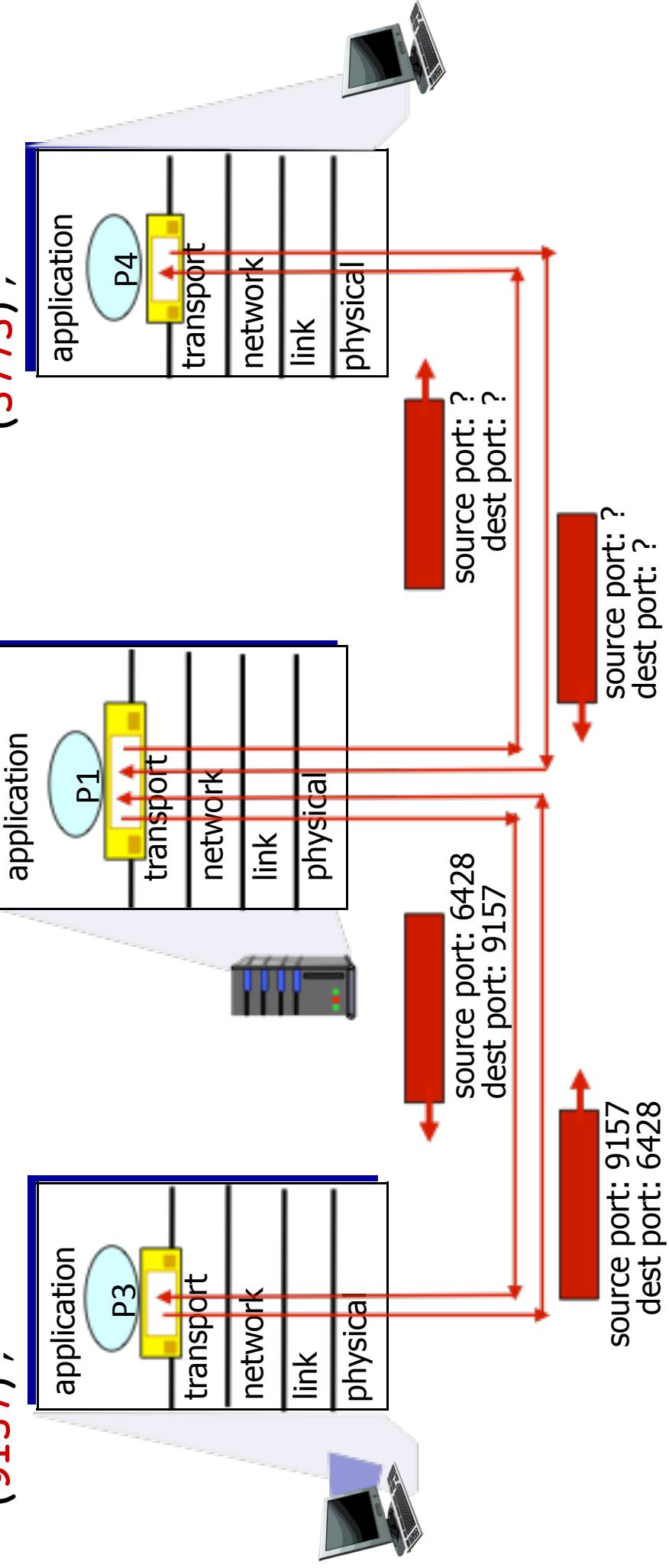
- recall: when creating datagram to send into UDP socket, must specify destination IP address destination port #
 - DatagramSocket mySocket1 = new DatagramSocket(12534);
- when host receives UDP segment:
 - checks destination port # in segment
 - directs UDP segment to socket with that port #
- IP datagrams with **same dest.port #**, but different source IP addresses
and/or source port numbers will be directed to **same socket** at dest

Connectionless demux: example



```
DatagramSocket  
serverSocket = new  
DatagramSocket  
(6428) ;  
  
DatagramSocket  
mySocket2 = new  
DatagramSocket  
(9157) ;
```

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



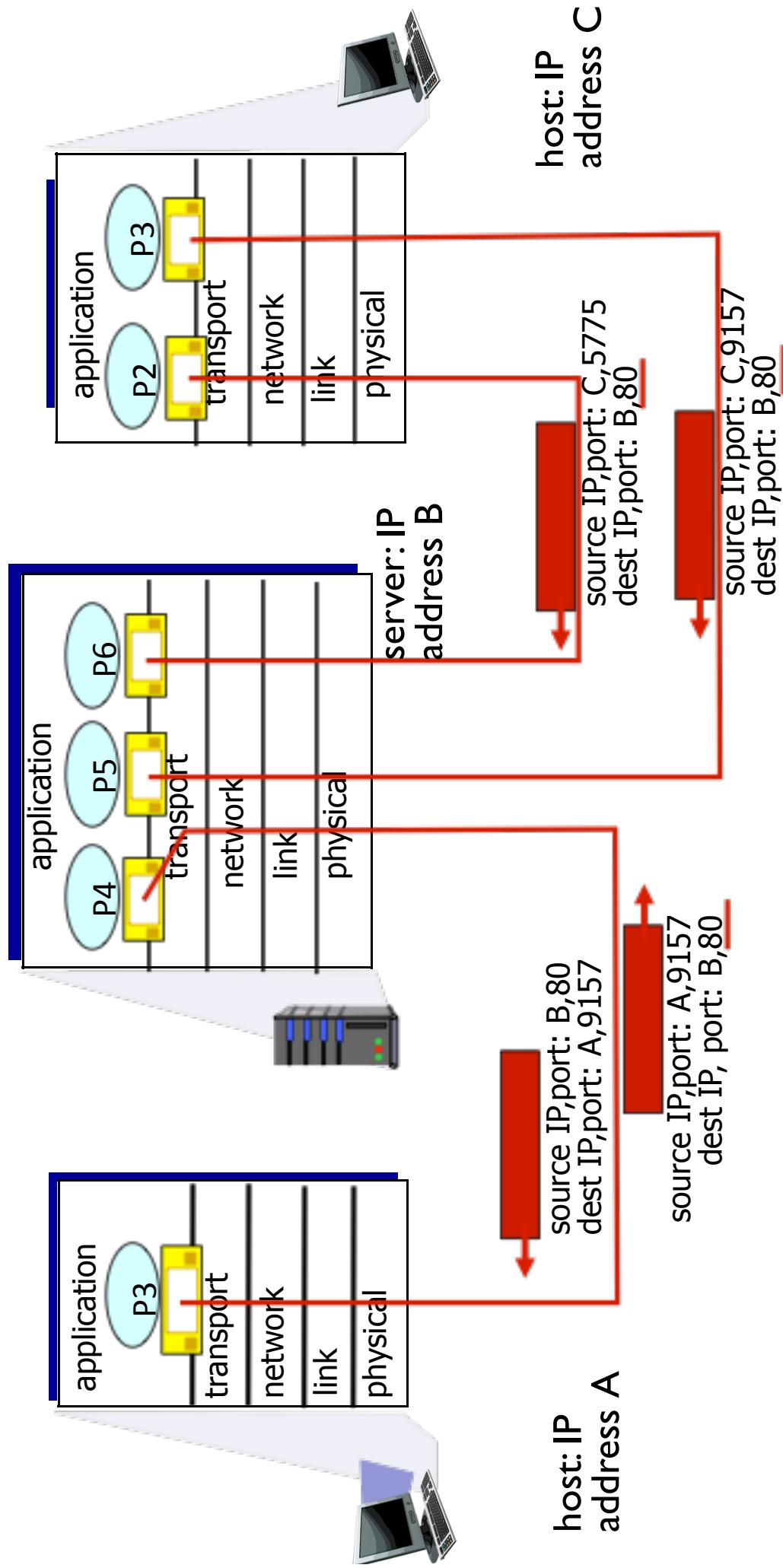
Connection-oriented demux



- TCP socket identified
 - by 4-tuple:
 - source IP address
 - source port number
 - dest IP address
 - dest port number
 - demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
 - each socket identified by its own 4-tuple
 - web servers have different sockets for each connecting client
 - non-persistent HTTP will have different socket for each request



Connection-oriented demux: example



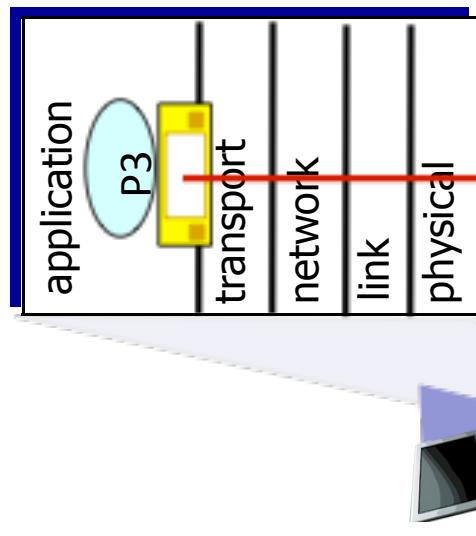
three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different sockets*

Connection-oriented demux: example



threaded

server



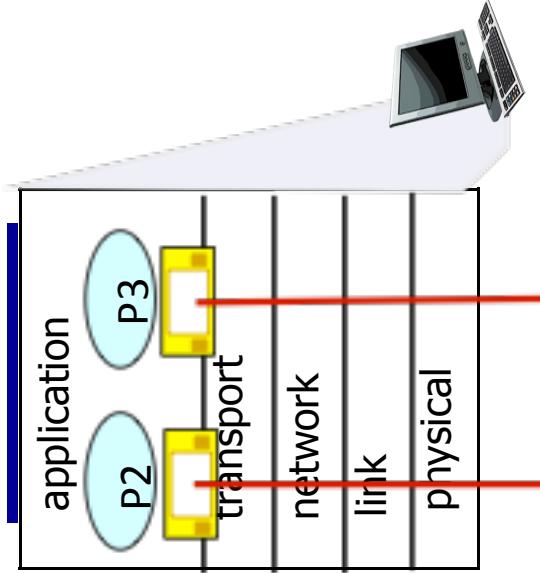
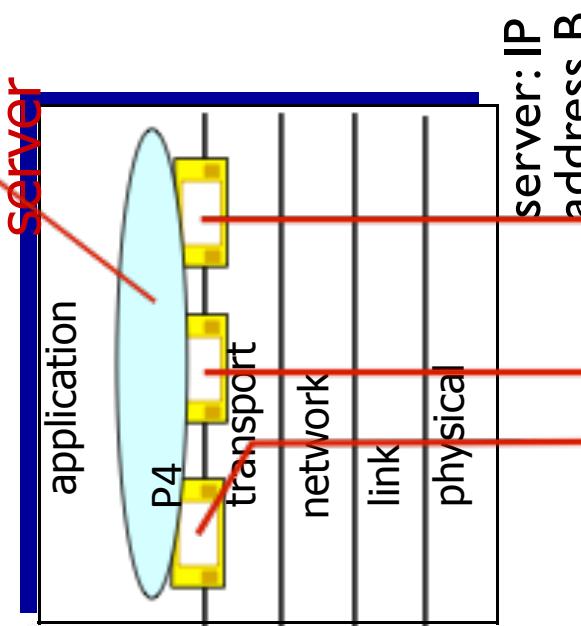
host: IP
address A

host: IP
address C

source IP, port: C, 5775
dest IP, port: B, 80

source IP, port: B, 80
dest IP, port: A, 9157

source IP, port: C, 9157
dest IP, port: B, 80



提纲

- 引言
- 核心问题：进程间如何通信
- 简单多路分解(UDP)
- 可靠字节流(TCP)
- 总结



用户数据报协议(UDP)



- TCP/IP协议栈中两个传输层协议之一

- UDP: 1980年实现

- TCP最早于1974年实现

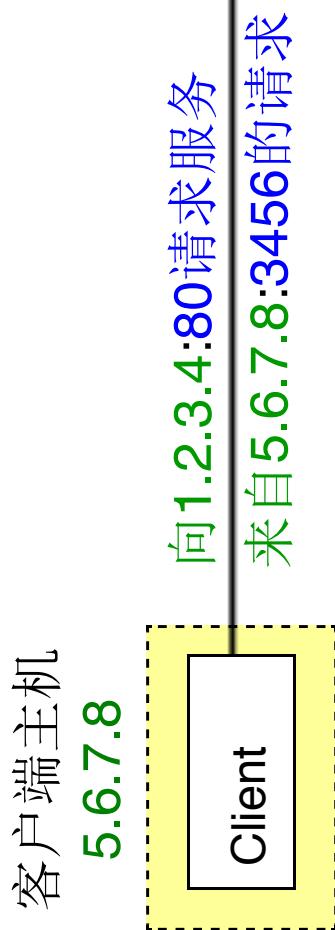
- 1981年IPv4实现

特点

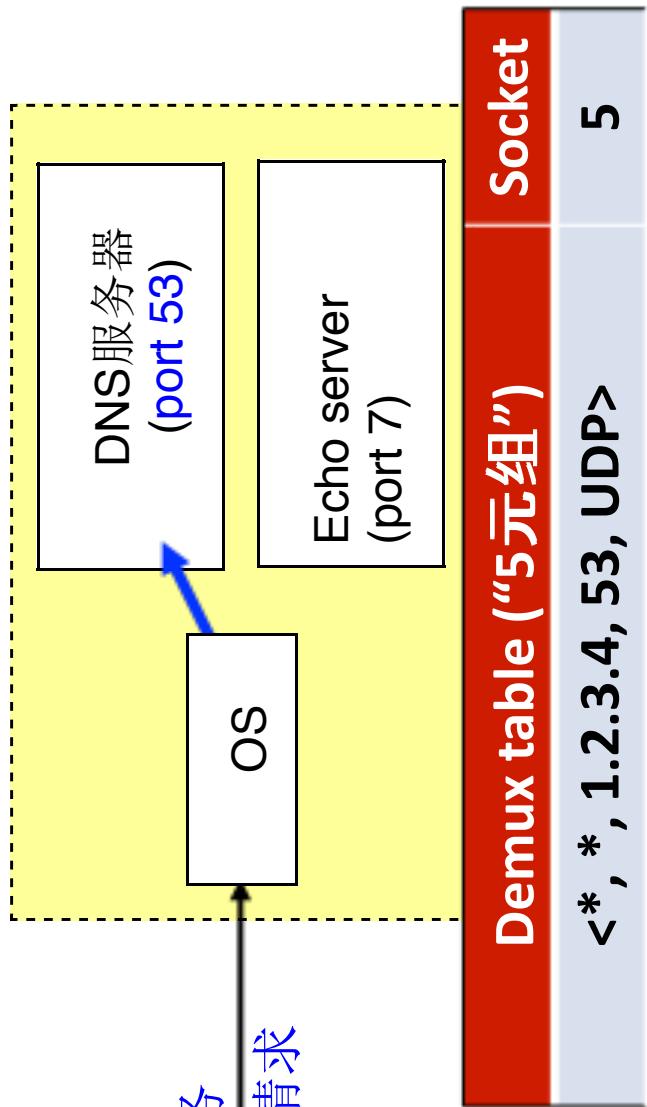
- 仅在IP基础上增加了一级解多路复用功能
- 面向报文
- 无连接
- 不保证消息的可靠传送
- 无流量控制

两大基本传送特点

- 解多路复用：端口号



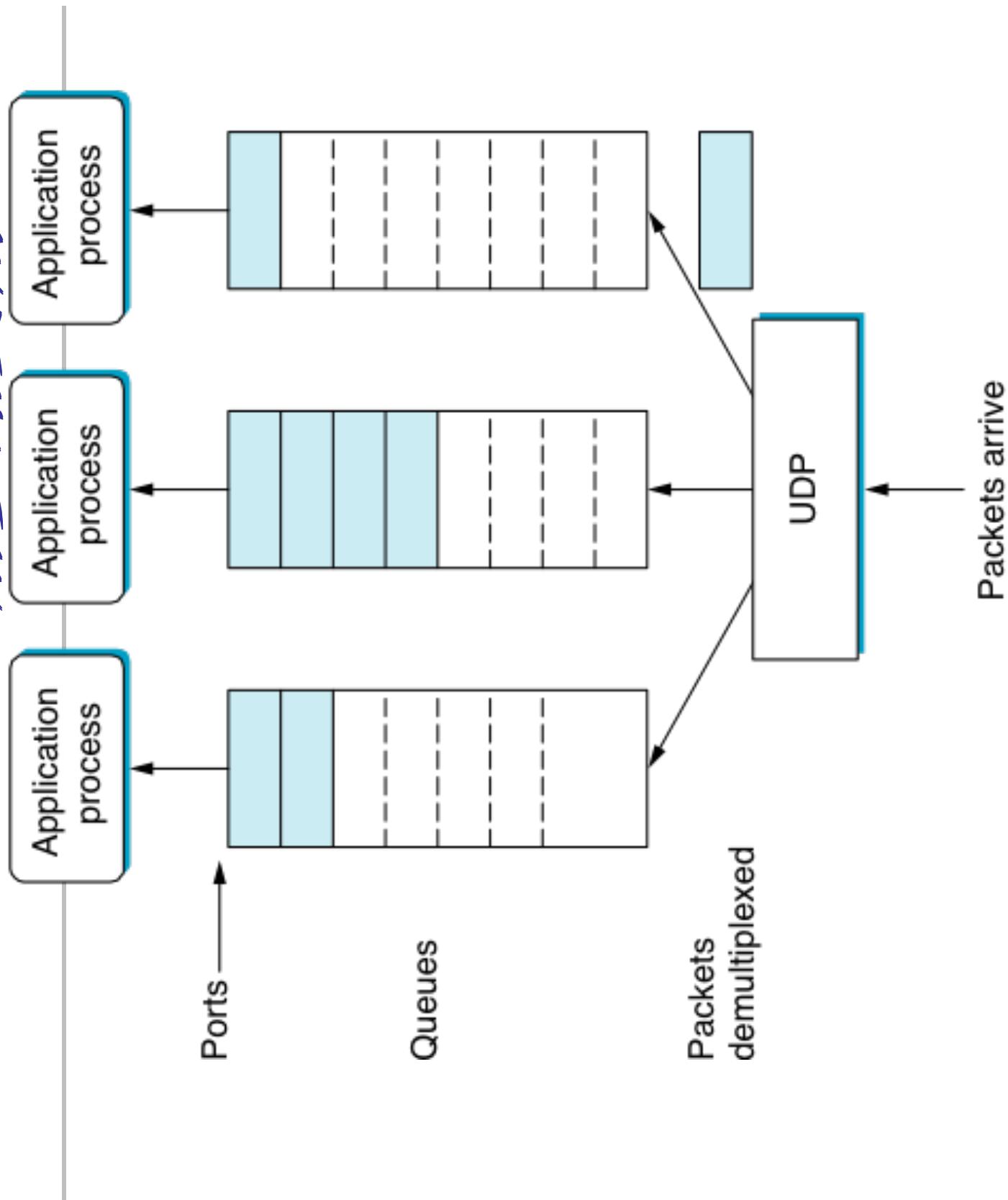
服务器主机 1.2.3.4



- 检错：校验和

detect corruption

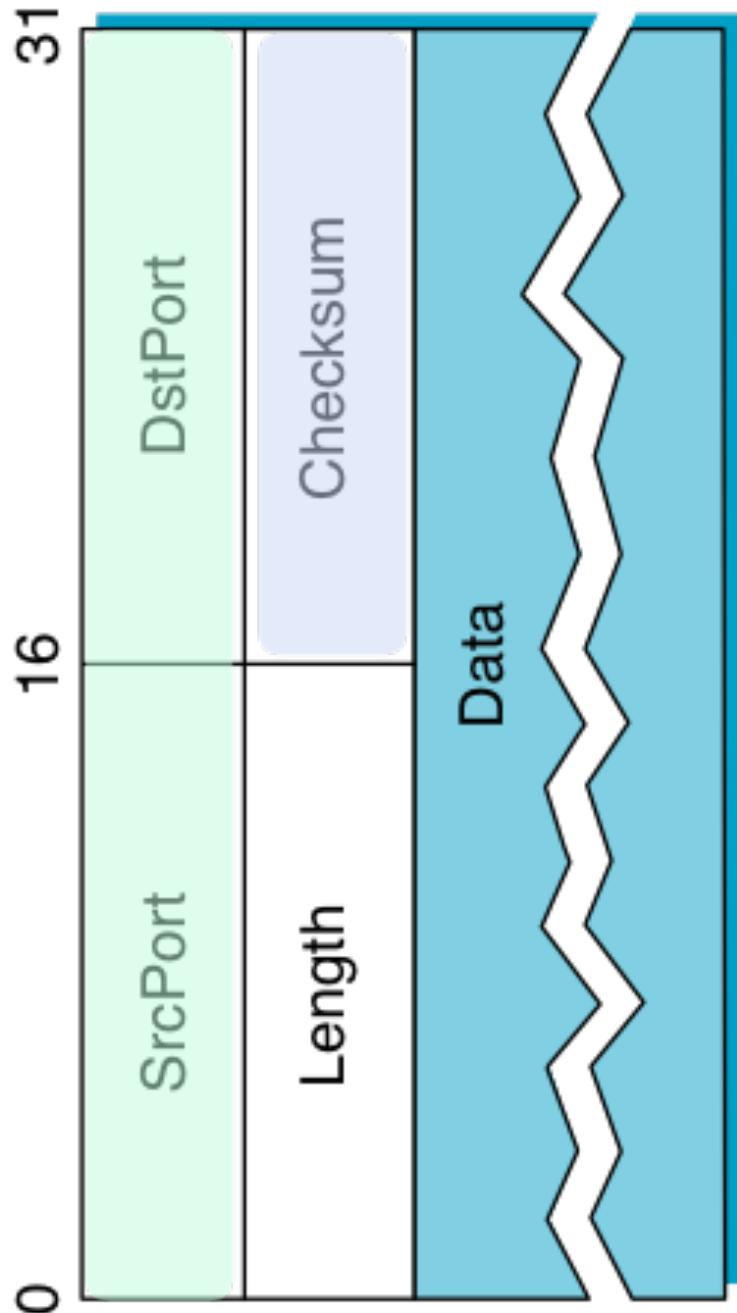
UDP 解多路复用



UDP 首部格式

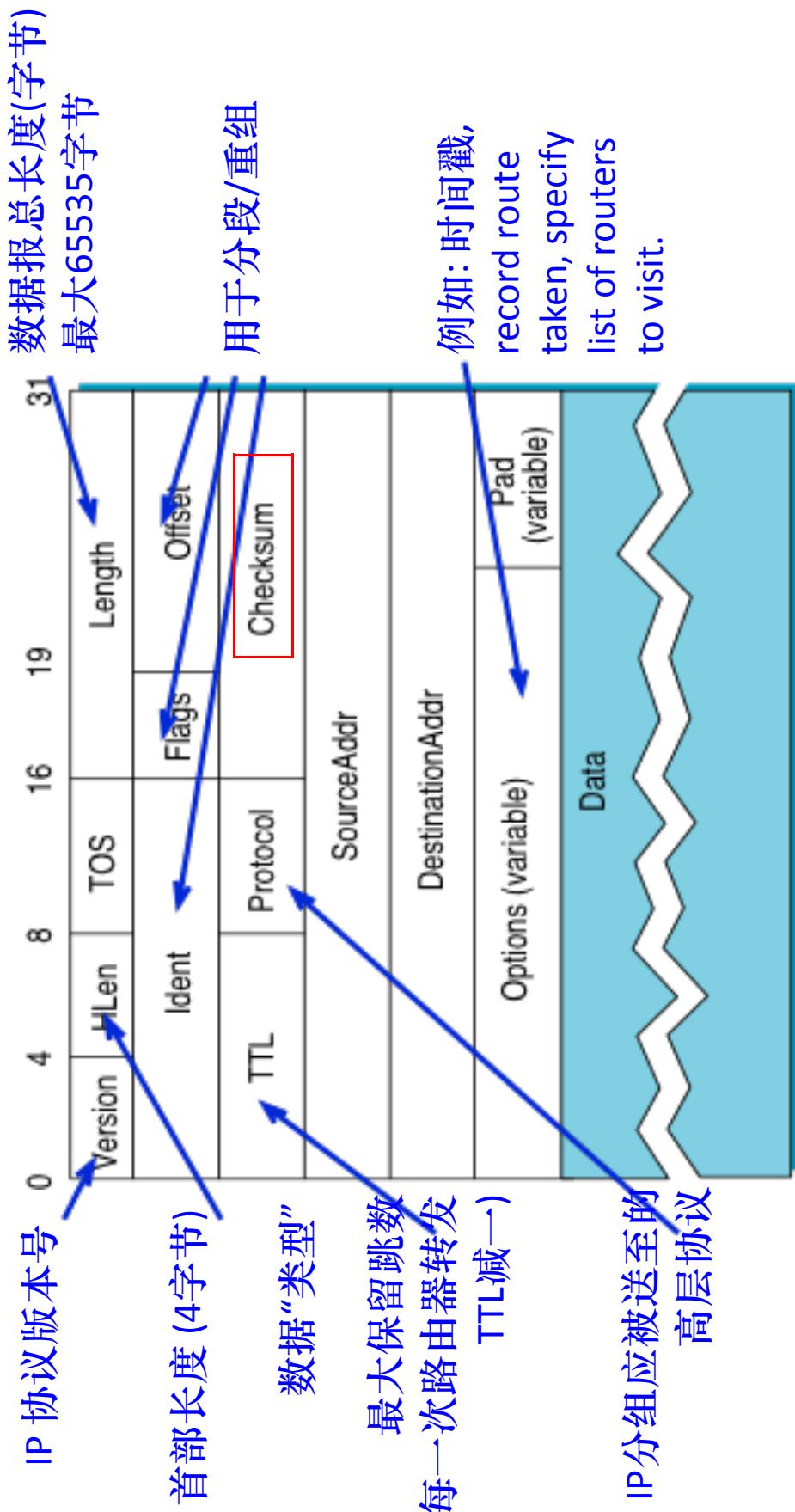
字段

- 源端口和目的端口：识别应用进程
- 校验和：IPv4可选项，计算整个UDP报文和伪首部





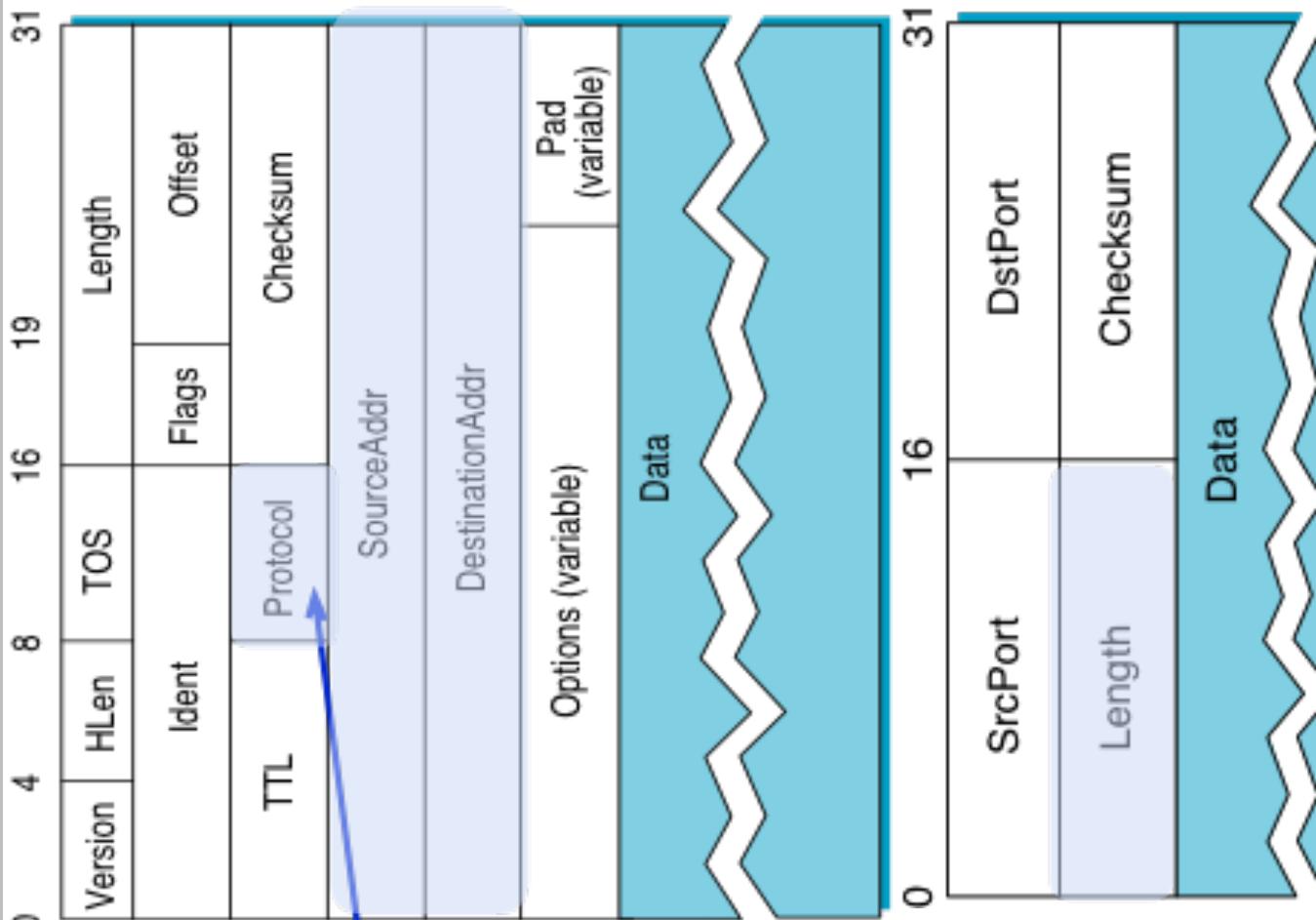
回顾IP服务模型：分组格式



UDP: 伪首部



IP分组应被送至的
高层协议



端口：公共服务



- 进程如何知道接收进程的端口号？
 - 通常的做法是让服务器进程在一个众所周知的端口接收消息
 - DNS, 端口53
 - SNMP, 端口25
 - 端口映射
 - 客户端和服务器通过众所周知的端口协商后续数据通信过程采用的端口，然后让出众众所周知的端口为其他客户进程服务

UDP的优点



- 控制数据发送的内容和时间
 - 一旦应用进程开始向socket写入数据
 - ... UDP封装数据并发送报文
- 不存在连接建立时延
- UDP不需要与目的进程之间进行联系而直接发送报文
- 无状态连接
 - 不需要分配缓存, 参数, 序号#, 等等.
 - ... 可以很容易的一次性处理多个客户端请求(肥服务器)
- 报文首部开销较小
- UDP首部只有8个字节长

UDP的缺点

- “尽最大努力交付”
- 不保证报文的传送，可能出现乱序到达
- 无拥塞控制
- 对网络的拥塞现象无自适应处理机制
- 抑制TCP流
- 一旦网络出现拥塞，TCP流会退避但UDP仍保留原速率发送
- 可以被用于网络攻击(UDP洪泛攻击)



采用UDP协议的应用



- 简单查询协议(例如域名解析系统, DNS)
 - 连接建立时延较大
 - 查询消息很小, UDP在消息内容上加载很小的协议开销(UDP首部)
 - 可以非常容易地进行重传
 - 适应报文乱序到达的危险
 - 多媒体应用
 - 不值得对丢弃或损坏报文进行重传
 - 对于语音、图像等数据传送过程,一定程度的报文丢失是可以接受的,
 - 应用案例：电话,视频会议,游戏

提纲

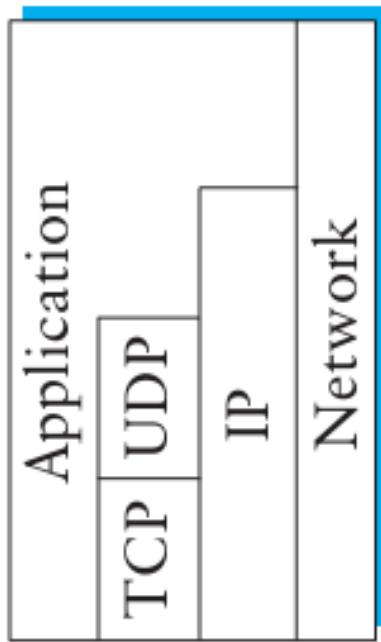
- 引言
- 核心问题: 进程间如何通信
 - 简单多路分解(UDP)
 - 可靠字节流(TCP)
- 总结



传输控制协议 (TCP)



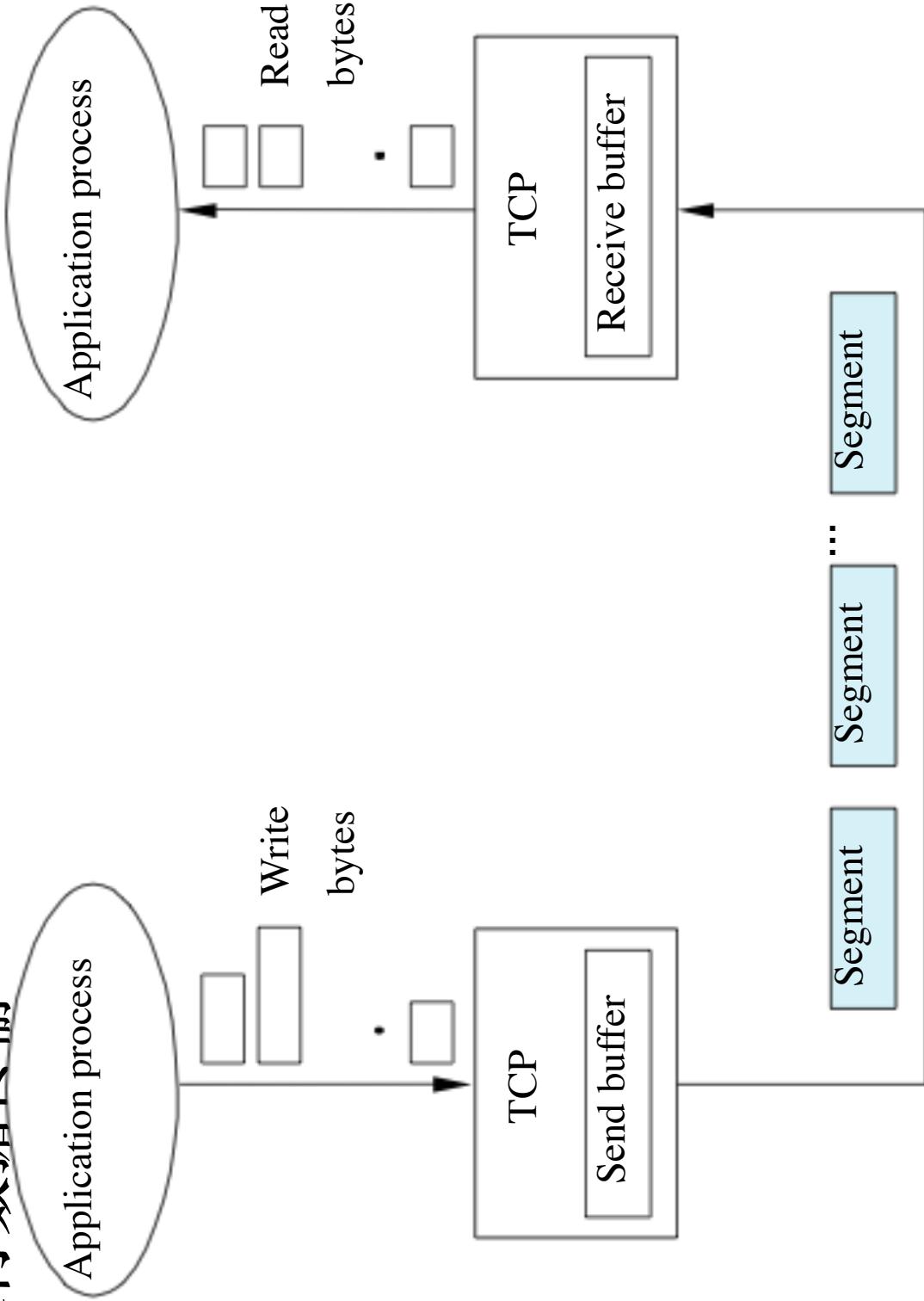
- TCP/IP协议栈中两个传输层协议之一
- UDP: 1980年实现
- TCP最早于1974年实现
- 1981年IPv4实现
-
-
-



- 向应用进程提供可靠的字节流服务
-

基于TCP的字节流传送

- TCP不区分应用进程写入报文的边界，采用数据段的方式进行数据传输



TCP传输的数据段



TCP 服务模型

面向连接

- 终端主机在数据交换之前需要连接连接
- 全双工：数据可双向传输
- 可靠性
 - 保证数据的传送
 - 数据按序到达
- 流量控制
 - 控制发送方的速率避免接收方过载
 - 在链路层也存在流量控制
- 拥塞控制
 - 控制发送方速率避免网络过载
 - 拥塞控制由网终且和传输层合作解决



流量控制 vs. 拥塞控制



Congestion control



Sender

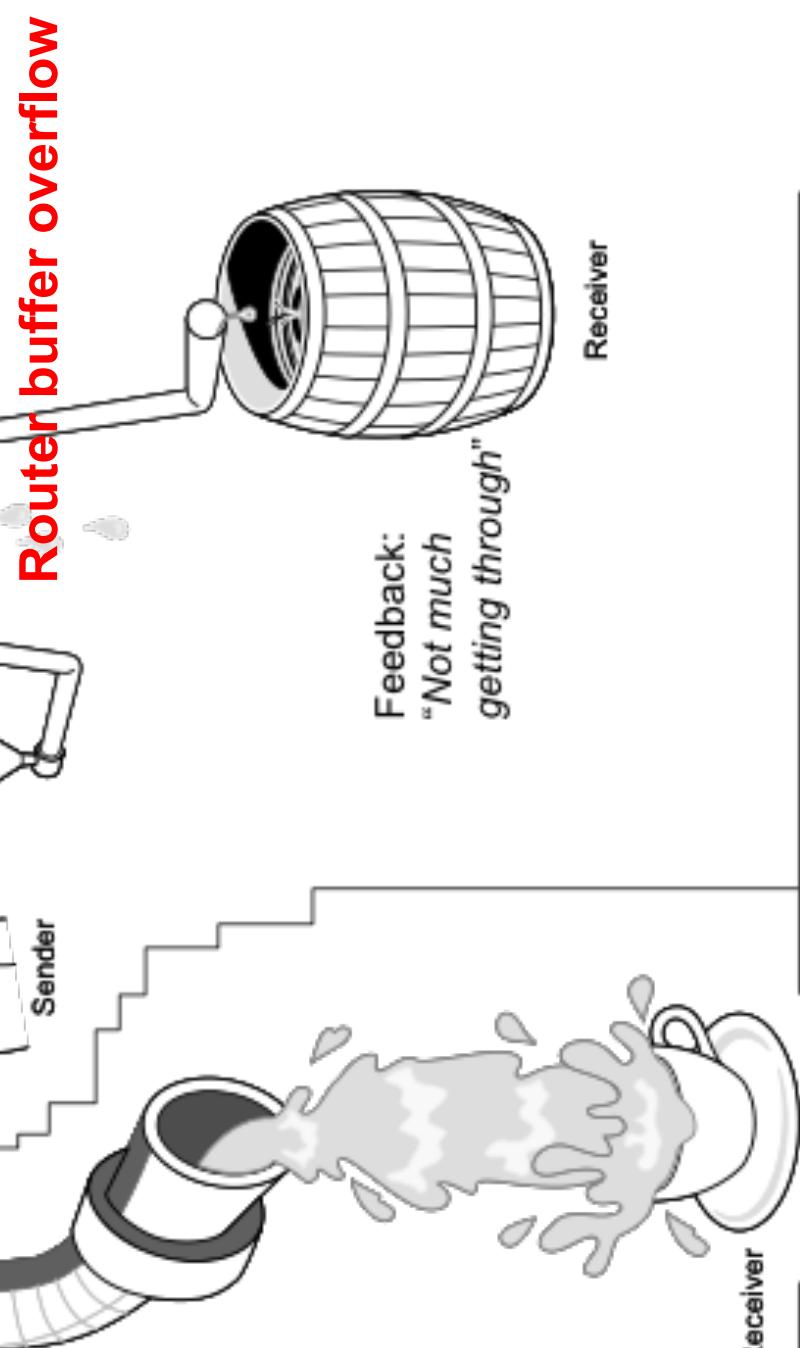
Sender

Feedback:
"Not much
getting through"

Receiver

Receiver

Router buffer overflow



Feedback:
"Receiver
overflowing"

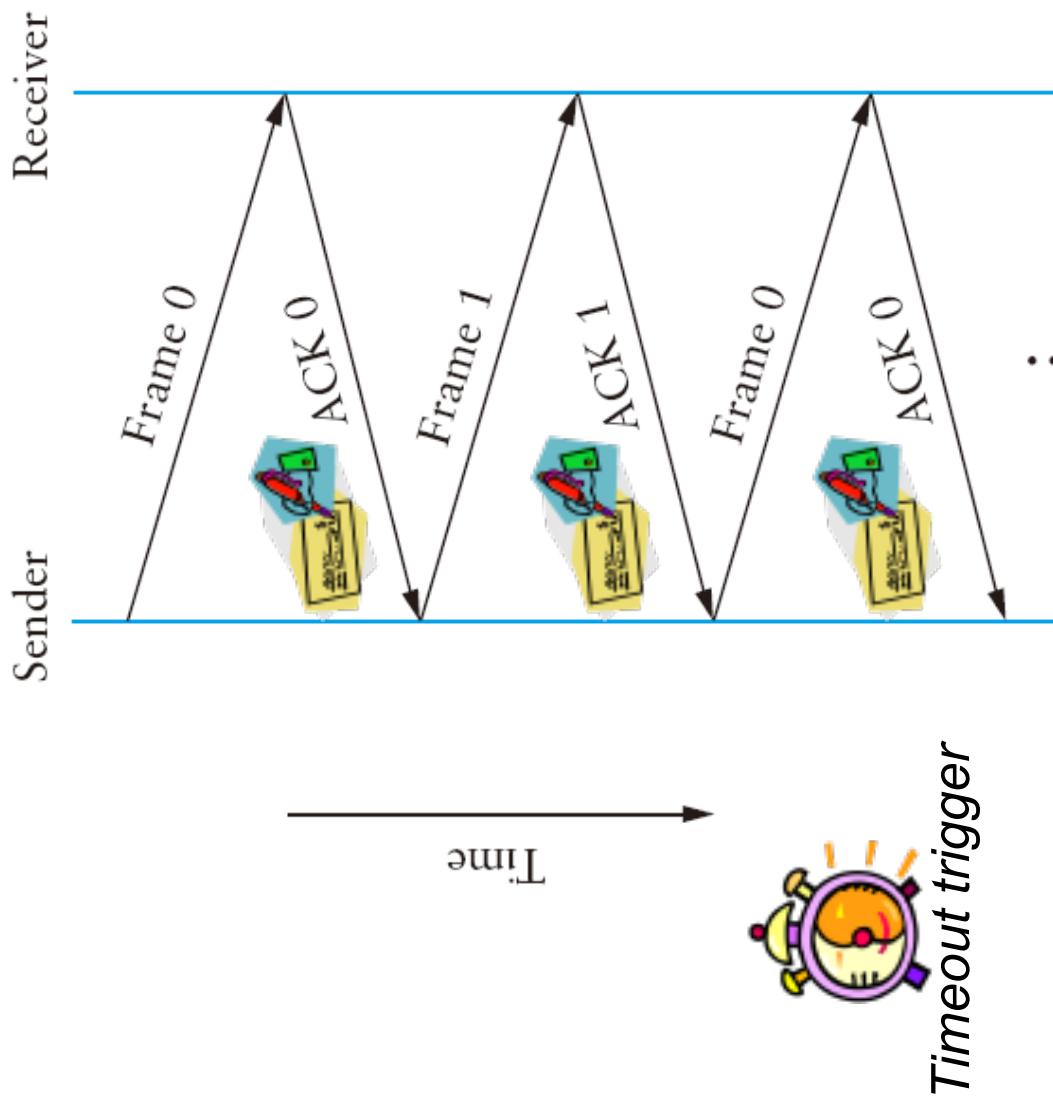
Receiver

提纲

- 引言
- 核心问题: 进程间如何通信
- 多路分解(UDP)
- 简单可靠字节流(TCP)
- 可靠到端的问题
- 报文段格式
- 连接的建立和终止
- 滑动窗口算法再讨论
- 触发传输
- 自适应重传
- 记录边界
- TCP扩展
- 其他设计选择
- 总结



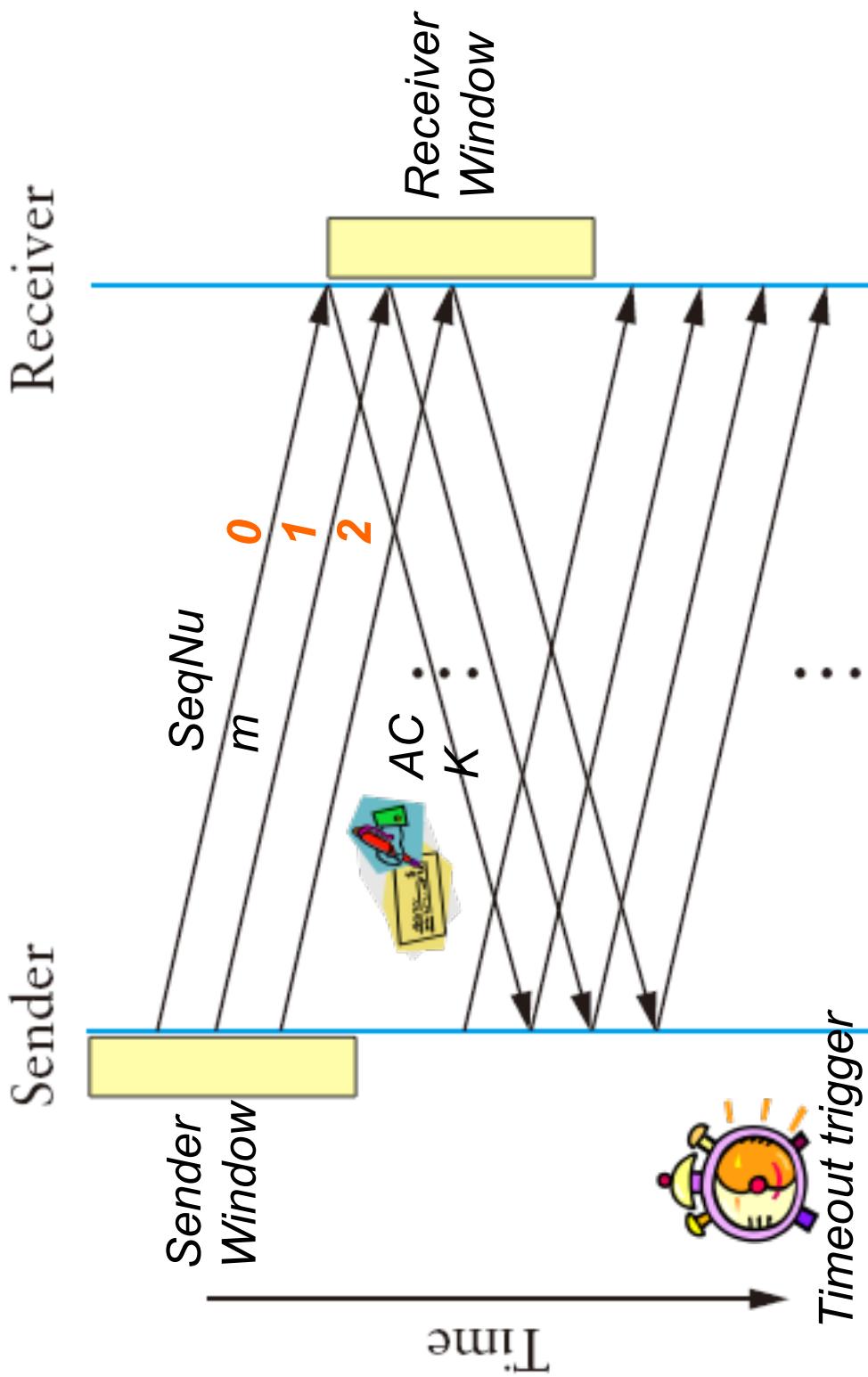
可靠通信回顾：最早的停止等待协议



提供可靠传输的两个基本机制：

- (1) ACK (确认)
- (2) 超时定时器

可靠通信回顾：滑动窗口算法



滑动窗口算法需要更多的组件：
(3) 发送窗口/接收窗口
(4) 数据帧序号SeqNum

可靠传输服务

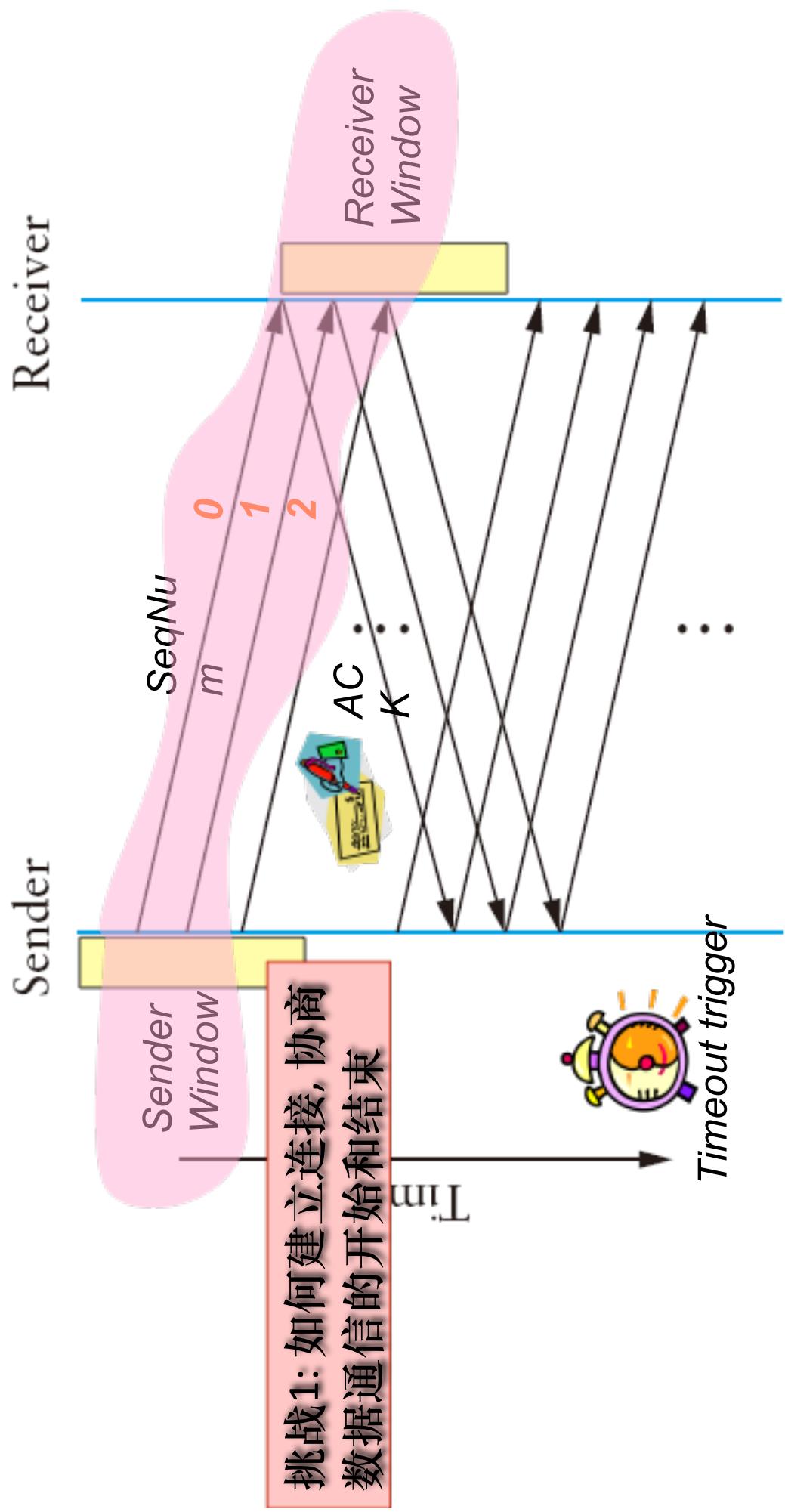


- 可靠传输服务的基本解决方法
 - ACK (确认)
 - 超时定时器
- 链路层
 - ARQ, 滑动窗口算法
- 传输层
 - 与链路层的可靠传输有什么区别?
 - TCP是否与底层提供了重复的服务?

TCP可靠传输面临的新挑战



背景：IP 为异构网络不同的主机间通信提供了不可靠的传输服务



TCP面临的新挑战 - 1

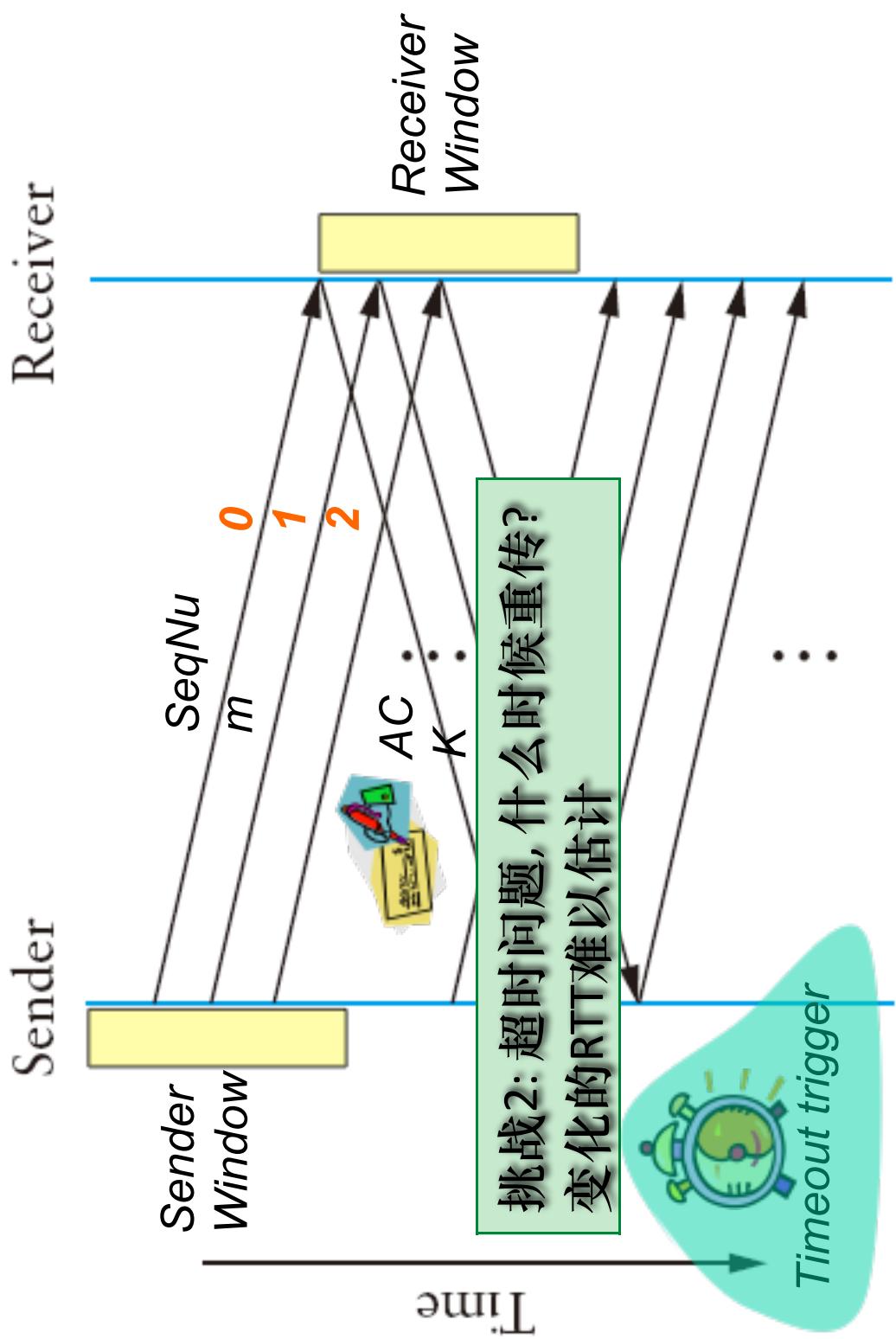


- 问题：连接
- 一个物理链路永远连接相同的两台主机，不需要建立连接
- TCP需要能够为运行在Internet上的任意两台主机上的进程提供逻辑连接
- 动机
 - TCP 需要明确的建立连接阶段
 - TCP 也有明确的断开连接阶段

TCP可靠传输面临的新挑战



背景：IP 为异构网络不同的主机间通信提供了不可靠的传输服务



TCP面临的新挑战 - 2

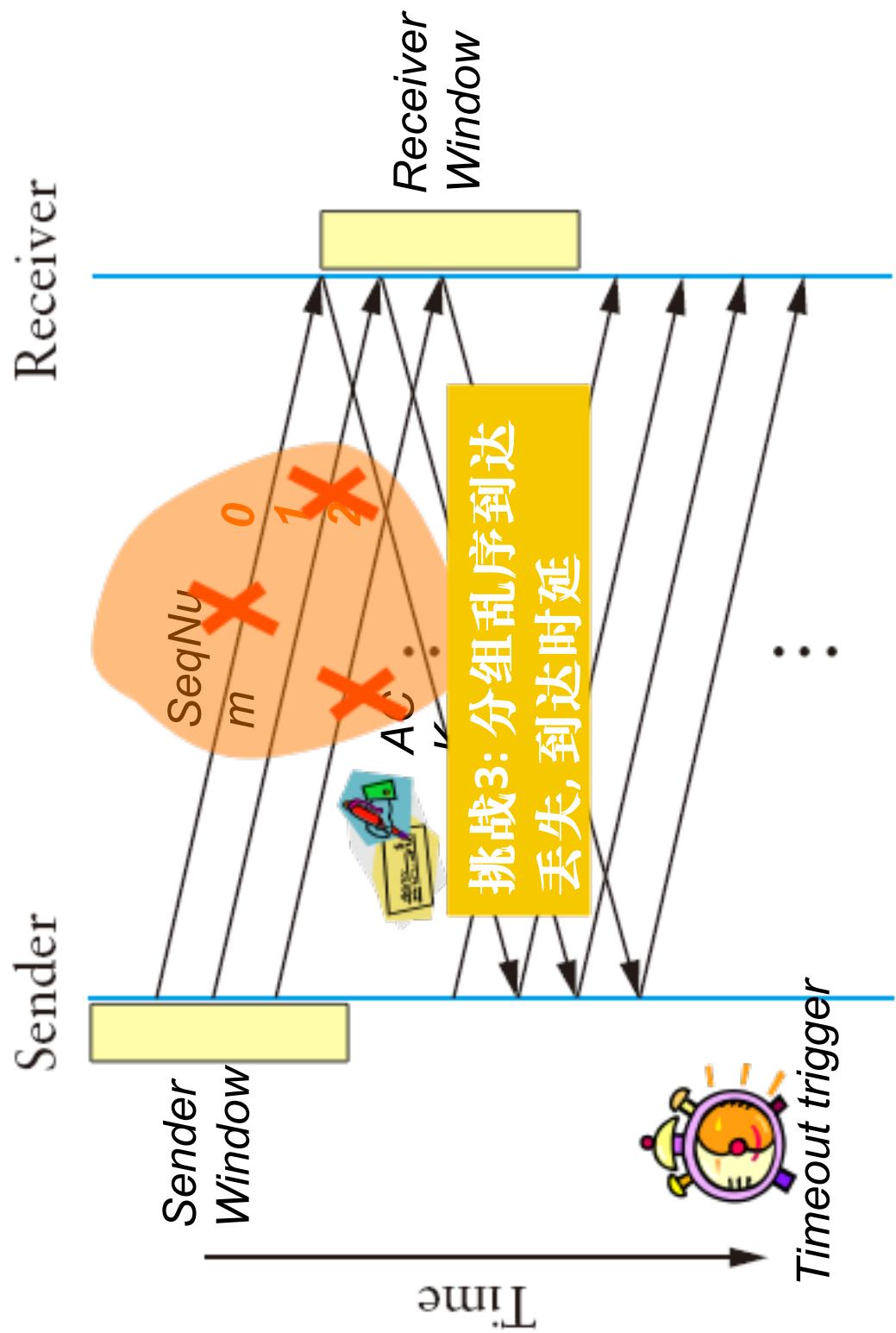


- 问题：超时重传
- 连接两台相同主机的物理链路通常具有固定的RTT
- 连接很可能具有差异很大的往返实验
- 不同的距离：San Francisco 到Boston, RTT 100 ms, 同一个房间内的两台主机, RTT 1 ms
- RTT的变化：San Francisco 到Boston, RTT 100 ms at 3 a.m., RTT 500 ms at 3 p.m.
- 动机
- 滑动窗口算法中的超时重传机制必须具有**适应性**.

TCP可靠传输面临的新挑战



背景：IP 为异构网络不同的主机间通信提供了不可靠的传输服务



TCP面临的挑战 - 3



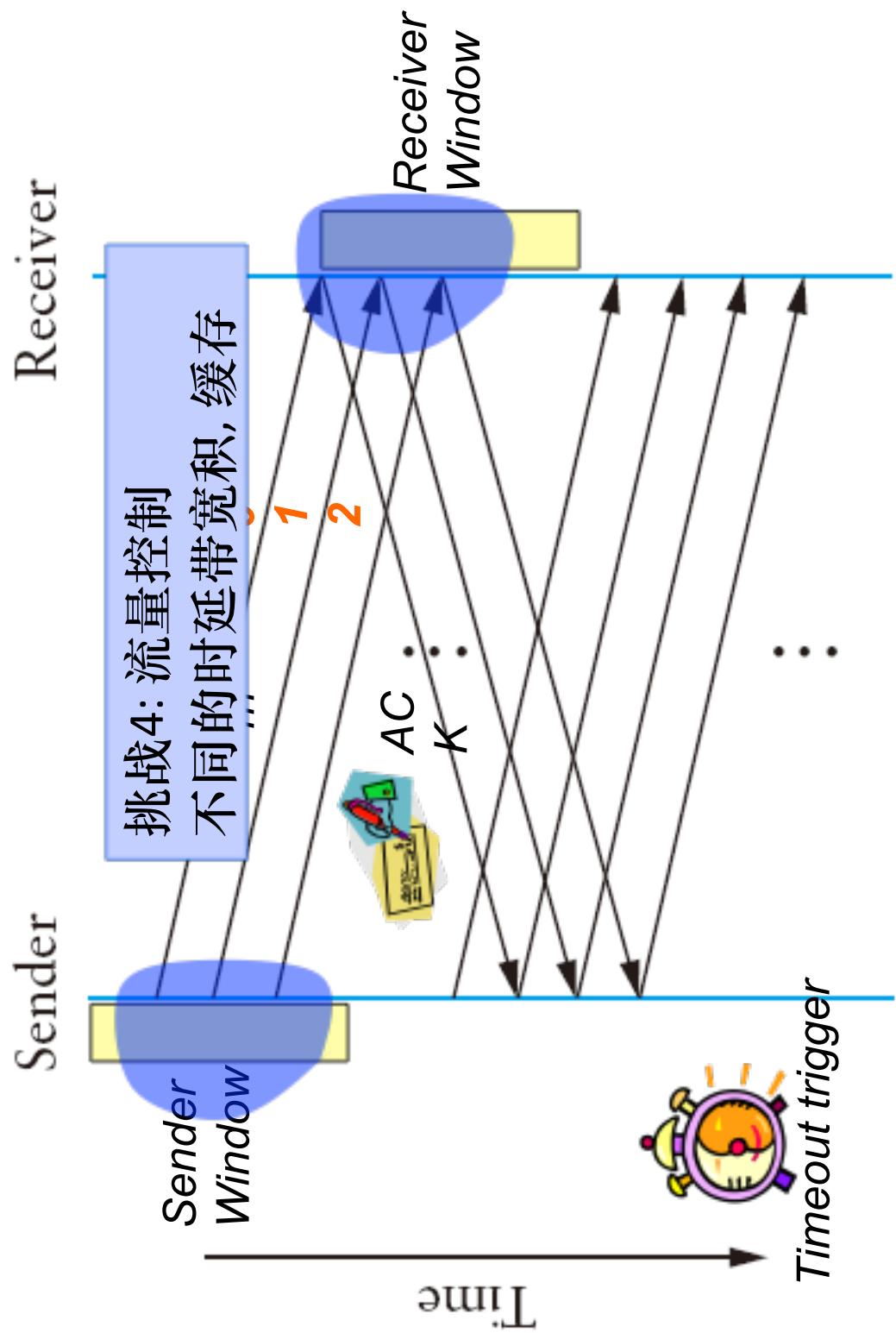
“研究表明，汉字的序顺并不一定能影响阅读，比如当你看完这句话后，才发现这里的字全是乱的。”

- 问题：乱序到达
- 点到点链路上不可能乱序到达
- 在多跳的Internet环境中可能出现分组乱序达到IP分组在TTL过期后被丢弃
- 动机
 - TCP 假设每一个分组有一个最大的生存周期，**最大段生存期 (MSL)**，当前协议的推荐值为120秒
 - TCP不得不为很早以前就被发出的分组突然出现在接收方做准备，这种分组可能会扰乱滑动窗口算法

TCP可靠传输面临的新挑战

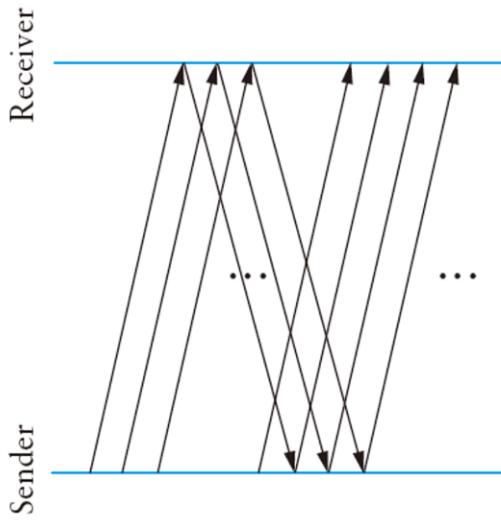


背景：IP 为异构网络不同的主机间通信提供了不可靠的传输服务

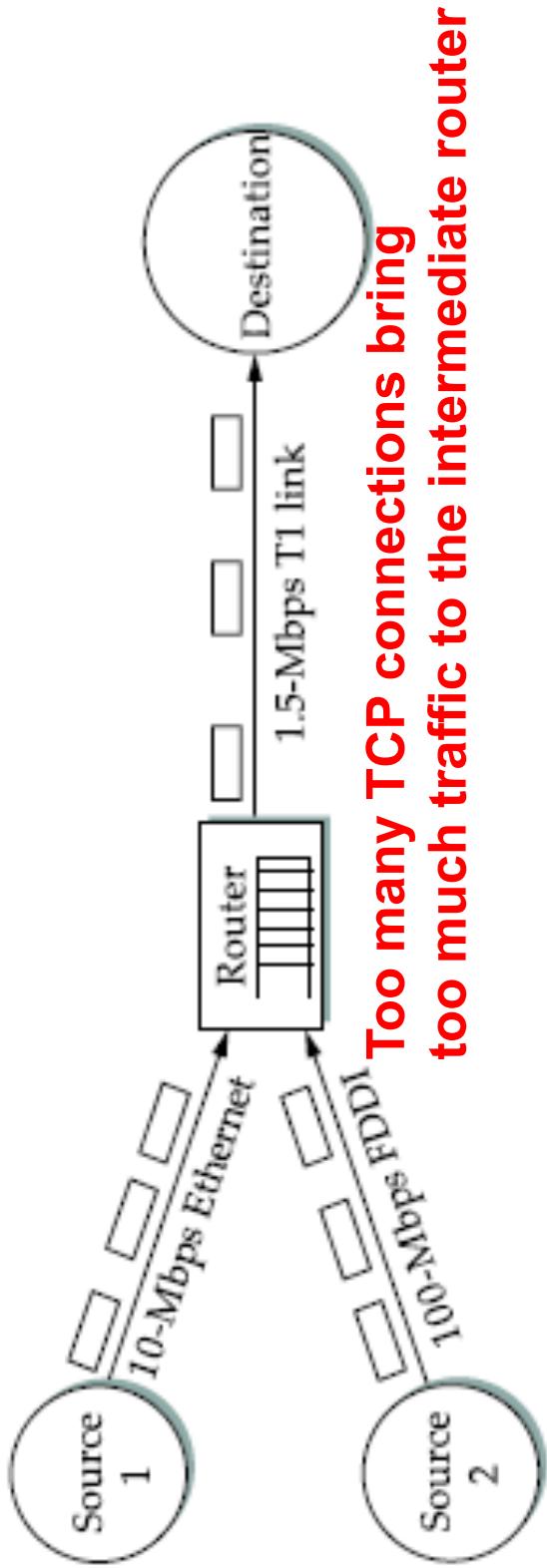


TCP面临的新挑战 - 4

- 问题：流量控制
 - 点到点链路的滑动窗口确定缓存参考链路的时延带宽积
 - Internet传输的缓存大小设计非常困难
 - 时延带宽积未知
 - 可能有成百个TCP连接共享缓存
- 云机
 - TCP必须包含一种机制使得连接的每一端能够“了解”另一端由什么资源（例如，多少缓冲区空间）用于连接=> 流量控制问题。



TCP可靠传输面临的新挑战



拥塞问题可以本地解决

TCP面临的新挑战 - 5



- 问题：拥塞控制
 - 点到点链路中不存在拥塞问题
 - Internet可能出现拥塞现象
 - TCP连接的发送端并不知道经过什么链路传送到目的地
 - 很多不同源产生的数据可能从同一低速链路通过
- 动机
 - TCP必须提供拥塞控制算法

小结: TCP面临的战略

- 与链路层提供的可靠传输相比, TCP面临的战略

- 建立连接

- 超时重传

- 乱序到达

- 流量控制

- 拥塞控制

- 是否存在逐跳解决方案?

- 由主机解决?还是由中间的路由器解决?



逐跳 vs. 端到端



- X.25 网络
 - 虚电路分组交换网络
 - 底层网络具有一定的可靠性
 - 报文在沿源主机到目的主机路径上的每对节点之间可以被可靠有序的传输
 - 逐跳采用滑动窗口算法
- TCP
 - 运行于数据报分组交换网络之上
 - 底层网络被认为不可靠的，且报文可能乱序到达
 - TCP在端到端的基础上采用滑动窗口算法提供可靠有序的传送

逐跳 vs. 端到端



- 一系列逐跳的保证不一定能够提供端到端的保证

端到端理论

- 一种功能不应该在系统的较低层提供，除非能在低层能够完全正确的被实现
- 有必要提供真正的端到端检测以保证可靠有序的服务，即使系统的低层已经实现了这种功能。

END-TO-END ARGUMENTS IN SYSTEM DESIGN

J.H. Saltzer, D.P. Reed and D.D. Clark*

M.I.T. Laboratory for Computer Science

J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design", ACM Trans. Comput. Syst., vol. 2, no. 4, pp. 277-288, 1984.

提纲

- 引言
- 核心问题: 进程间如何通信
- 简单多路分解(UDP)
- 可靠字节流(TCP)
- 端到端的问题
- 报文段格式
- 连接的建立和终止
- 滑动窗口算法再讨论
- 触发传输
- 自适应重传
- 记录边界
- TCP扩展
- 其他设计选择
- 总结

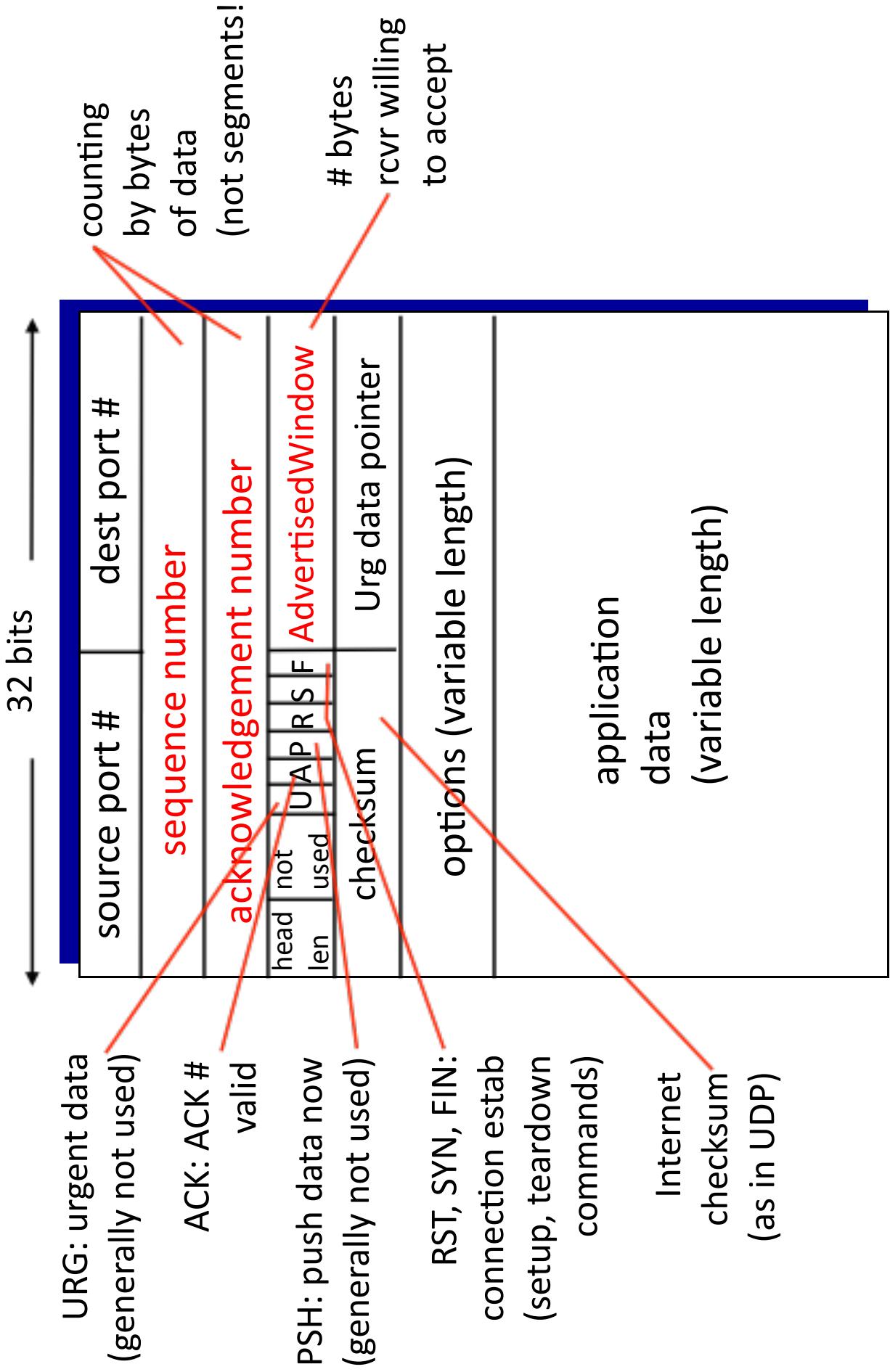


TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581



- **point-to-point:**
 - one sender, one receiver
 - **reliable, in-order byte stream:**
 - no “message boundaries”
- **full duplex data:**
 - bi-directional data flow in same connection
 - MSS: maximum segment size
- **connection-oriented:**
 - handshaking (exchange of control msgs) init sender, receiver state before data exchange
- **flow controlled:**
 - TCP congestion and flow control set window size
 - sender will not overwhelm receiver

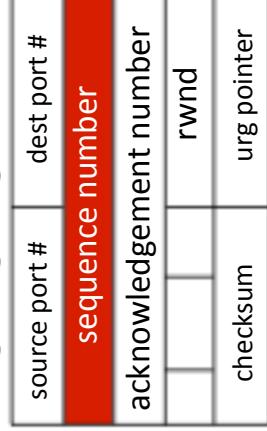
TCP segment structure



TCP seq. numbers, ACKs



outgoing segment from sender



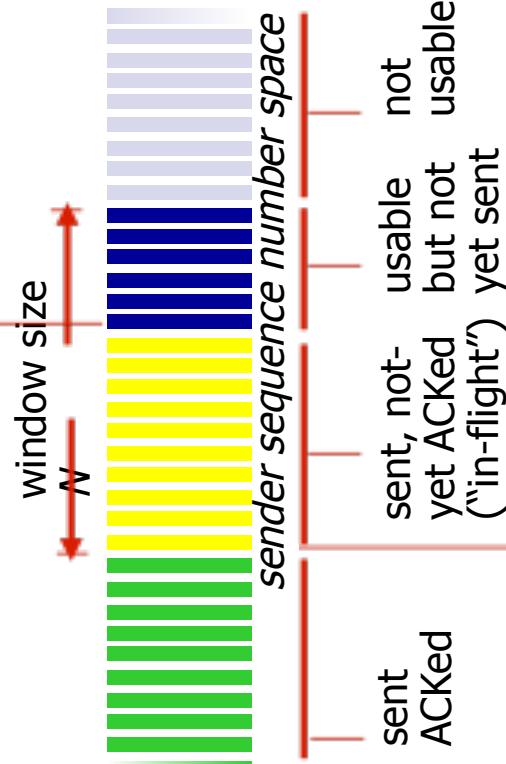
sequence numbers:

- byte stream “number” of first byte in segment’s data

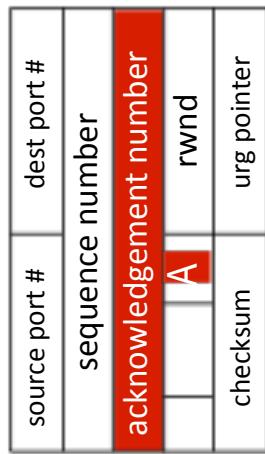
acknowledgements:

- seq # of next byte expected from other side
- cumulative ACK

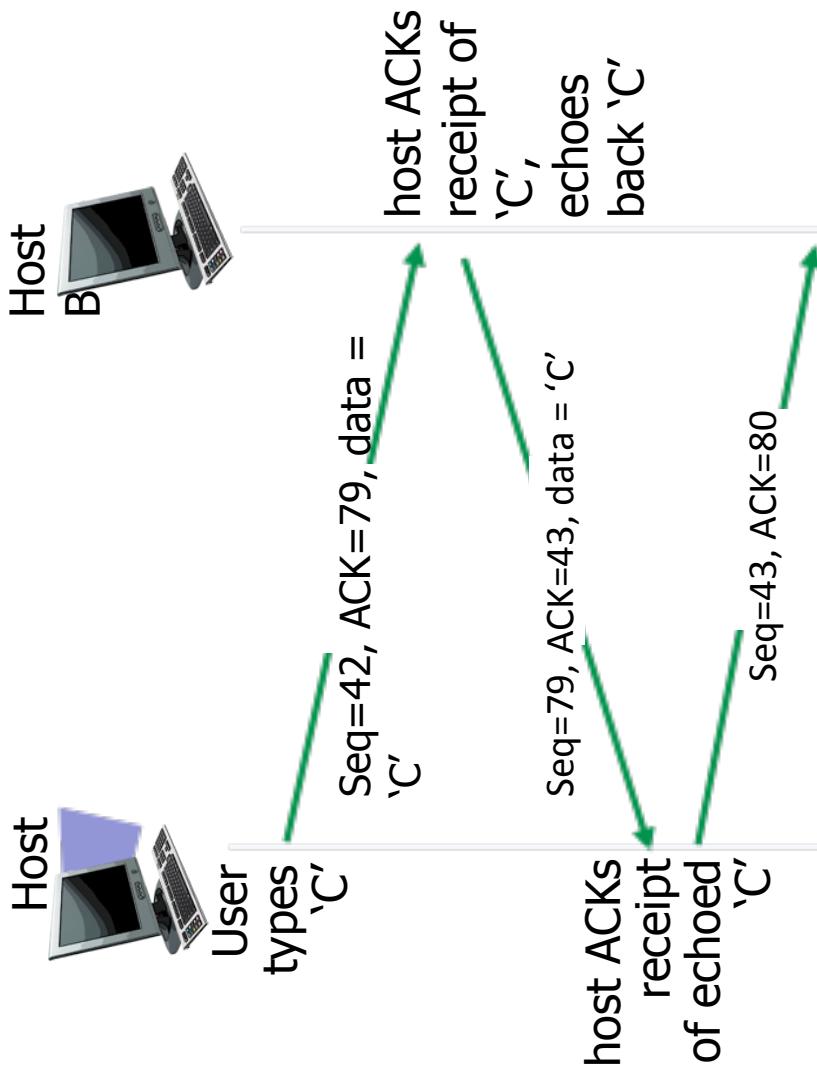
- Q:** how receiver handles out-of-order segments
- A: TCP spec doesn’t say, - up to implementor



incoming segment to sender



TCP seq. numbers, ACKs

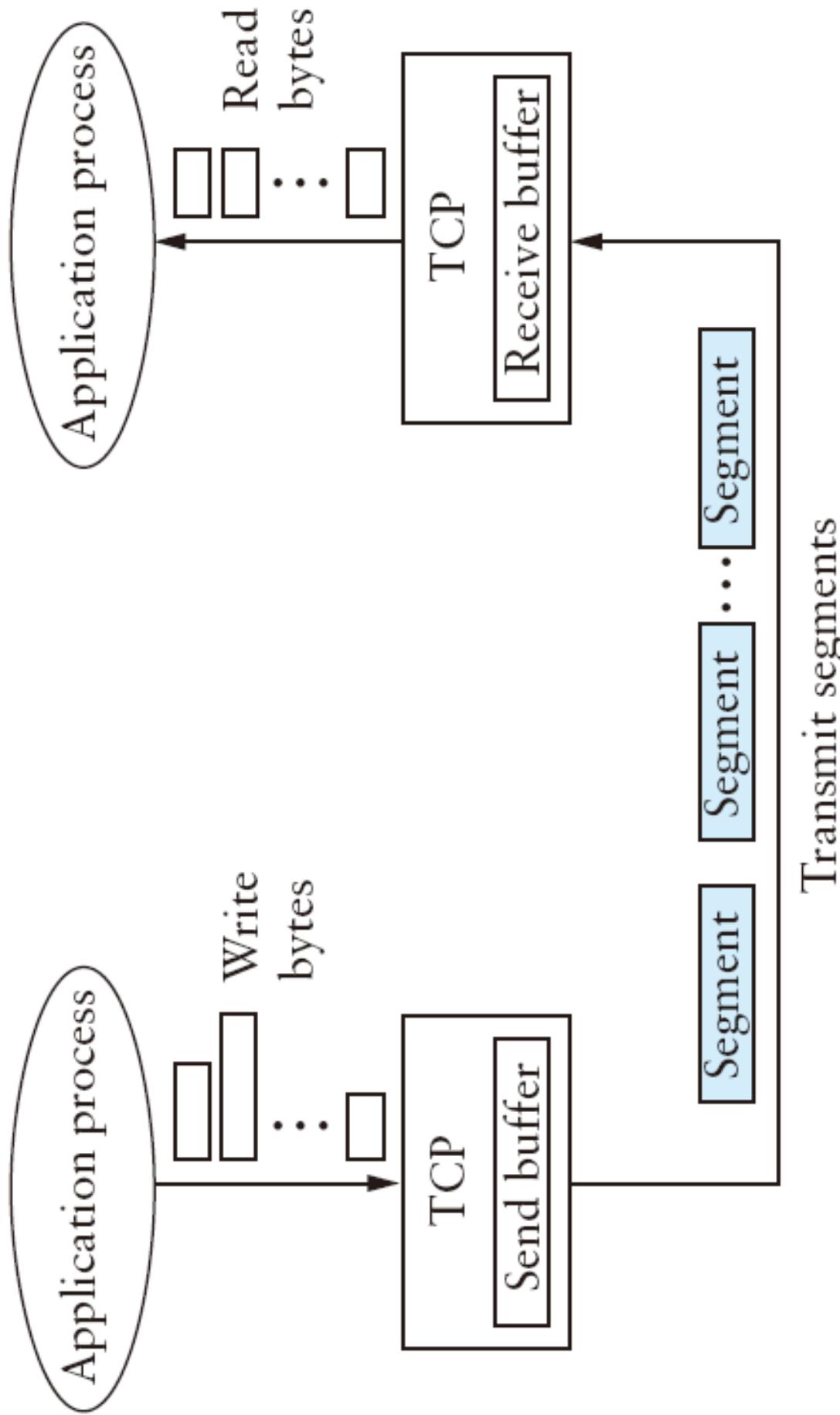


simple telnet scenario

TCP 数据段

- TCP 发送数据段
- 数据的数量达到了最大数据段的大小 (MSS)
 - 由应用进程触发, 例如 push 操作
 - 周期性定时器超时
- MSS 的选择
 - 尽可能避免 IP 分组分片
 - $MSS = MTU - IP \text{ 首部大小} - TCP \text{ 首部大小}$





提纲

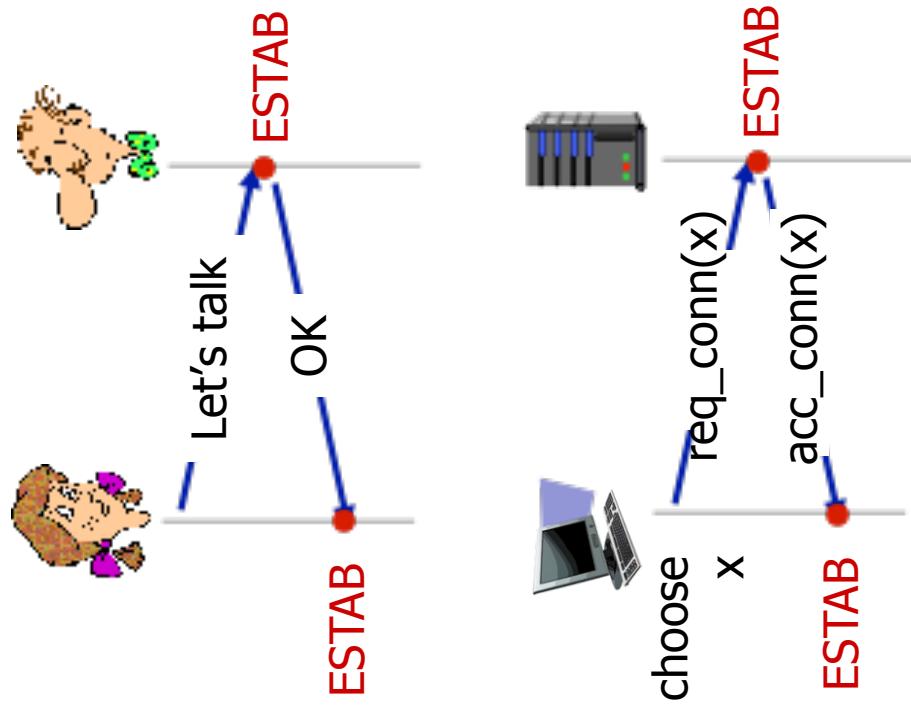
- 引言
- 核心问题: 进程间如何通信
- 简单多路分解(UDP)
- 可靠字节流(TCP)
- 端到端的问题
- 报文段格式
- 连接的建立和终止
- 滑动窗口算法再讨论
- 触发重传
- 自适应传输
- 记录边界
- TCP扩展
- 其他设计选择
- 总结



Agreeing to establish a connection



2-way handshake:

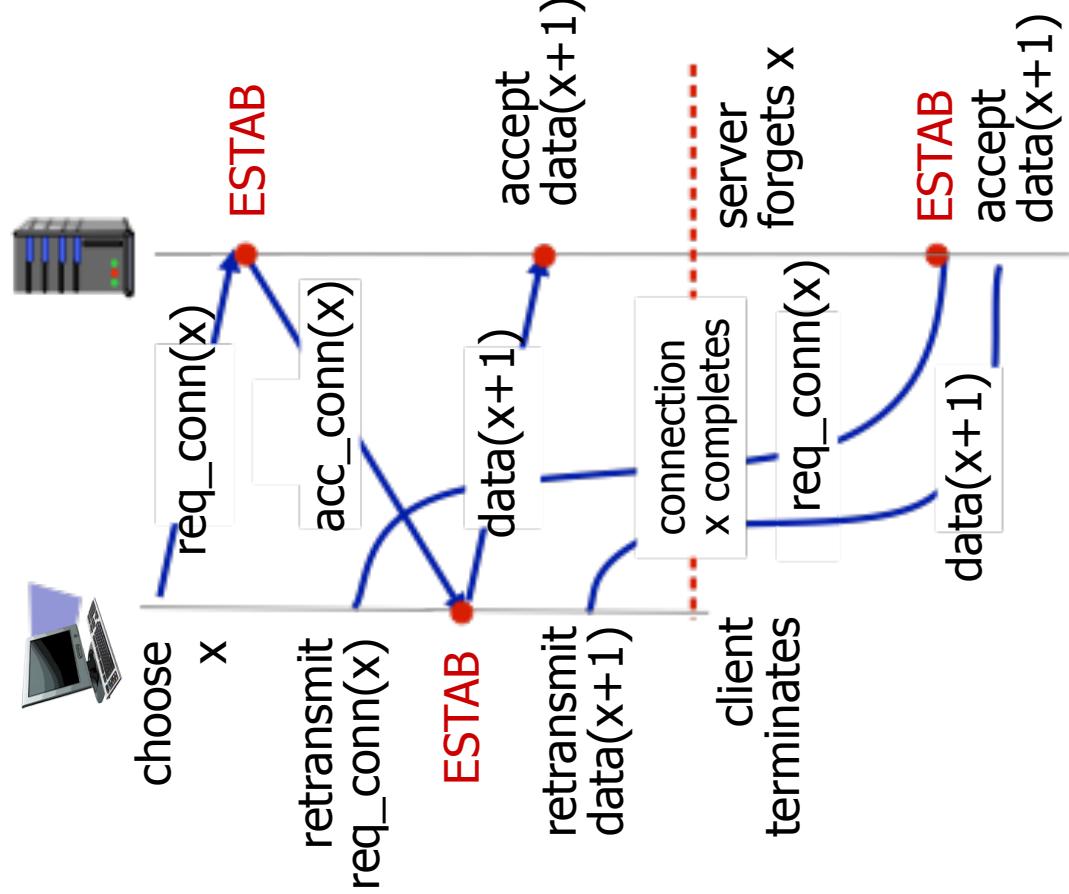
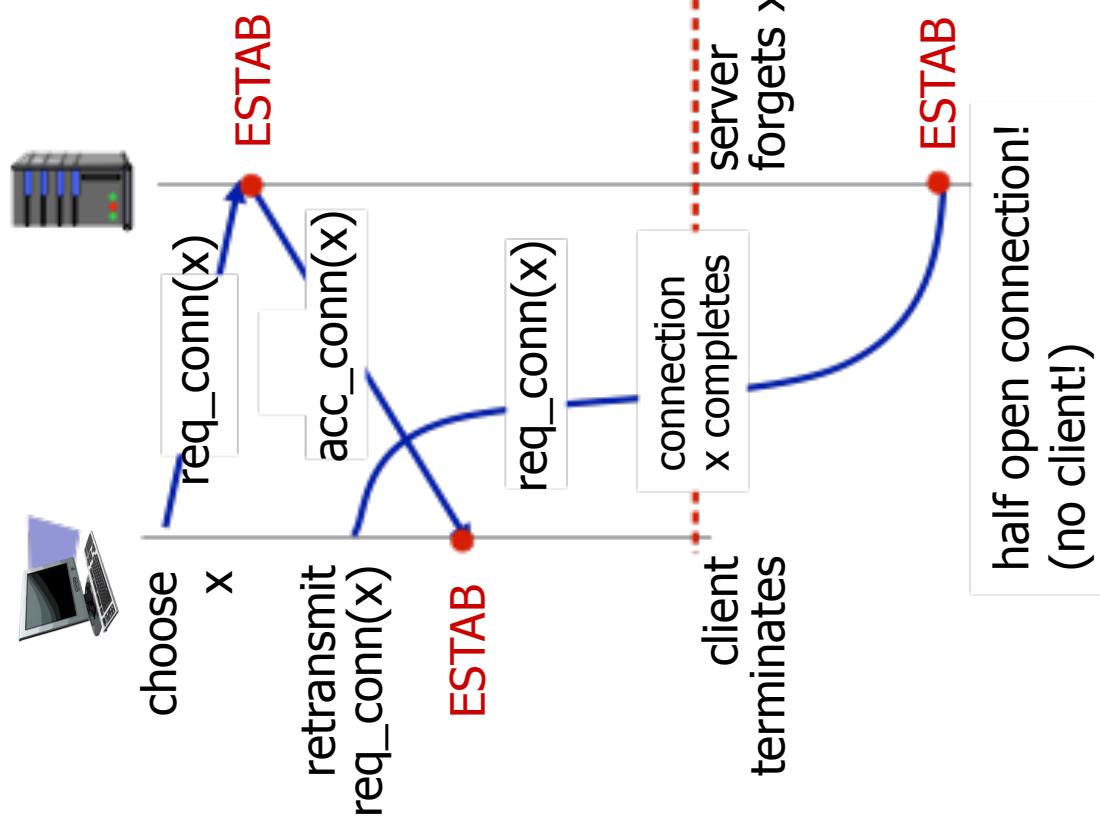


Q: will 2-way handshake always work in network?

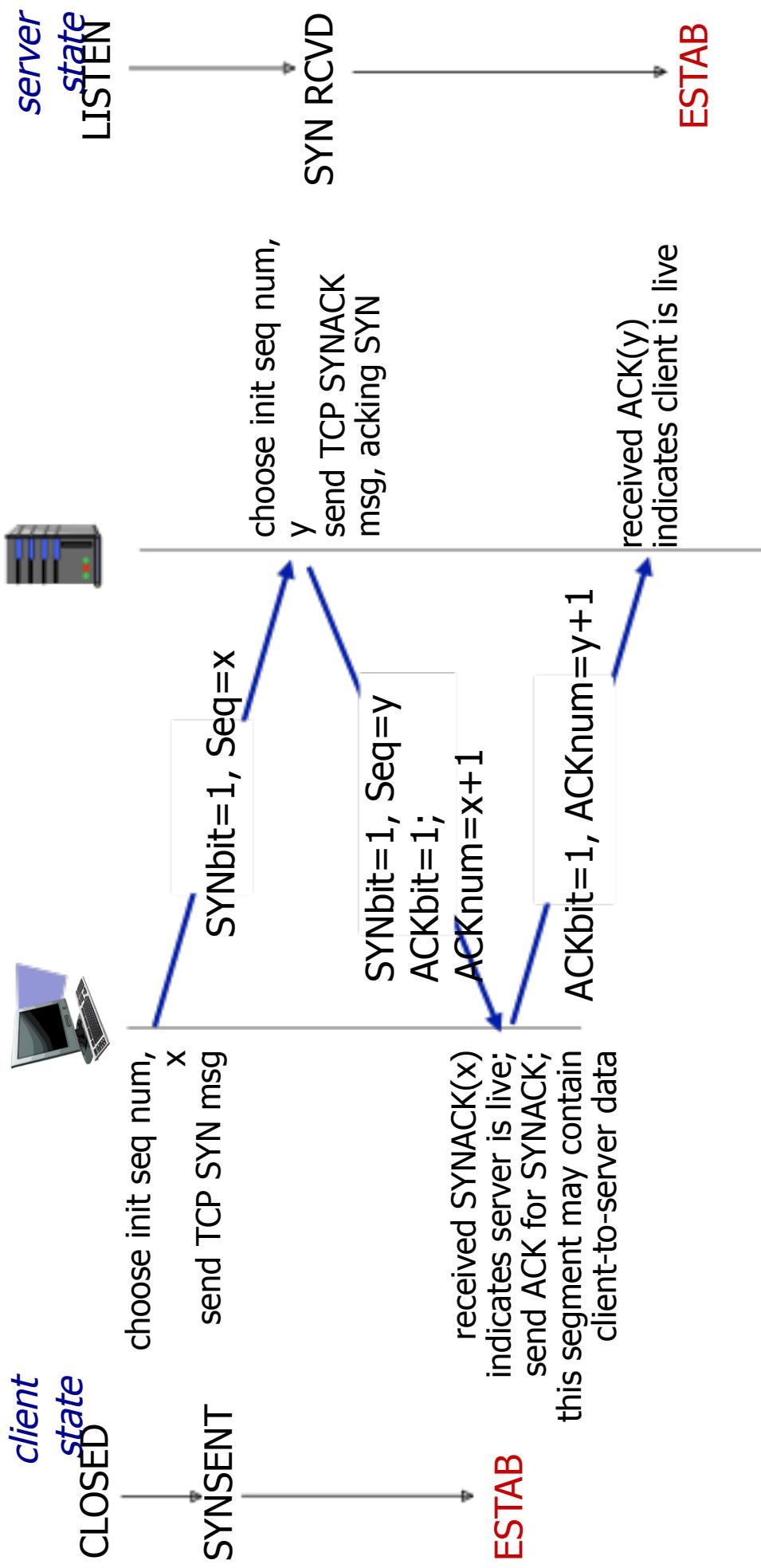
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- **Can't "see" other side**

Agreeing to establish a connection

2-way handshake failure scenarios:



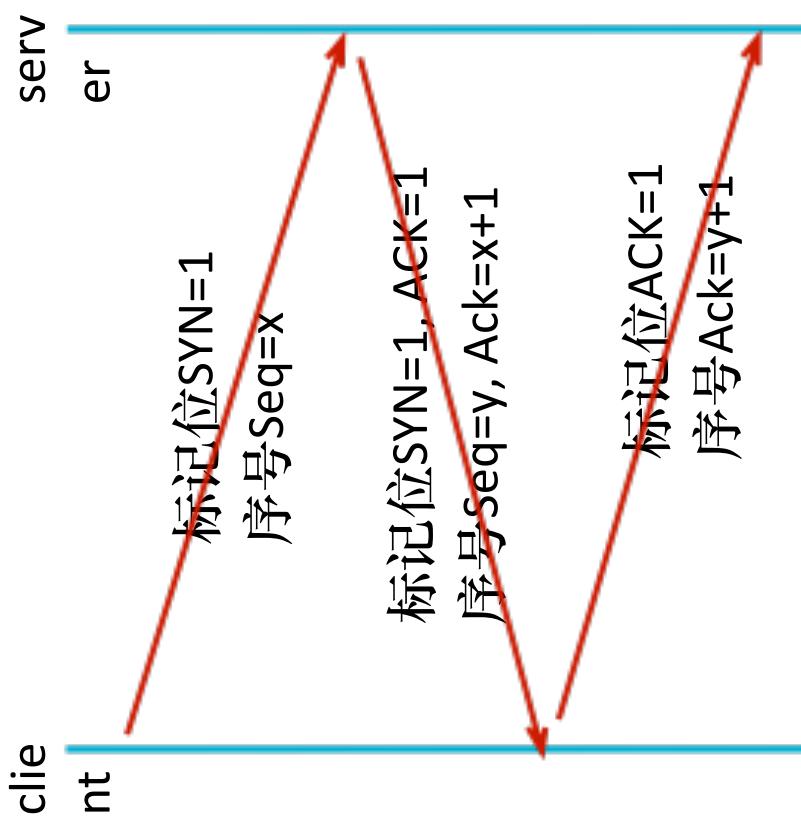
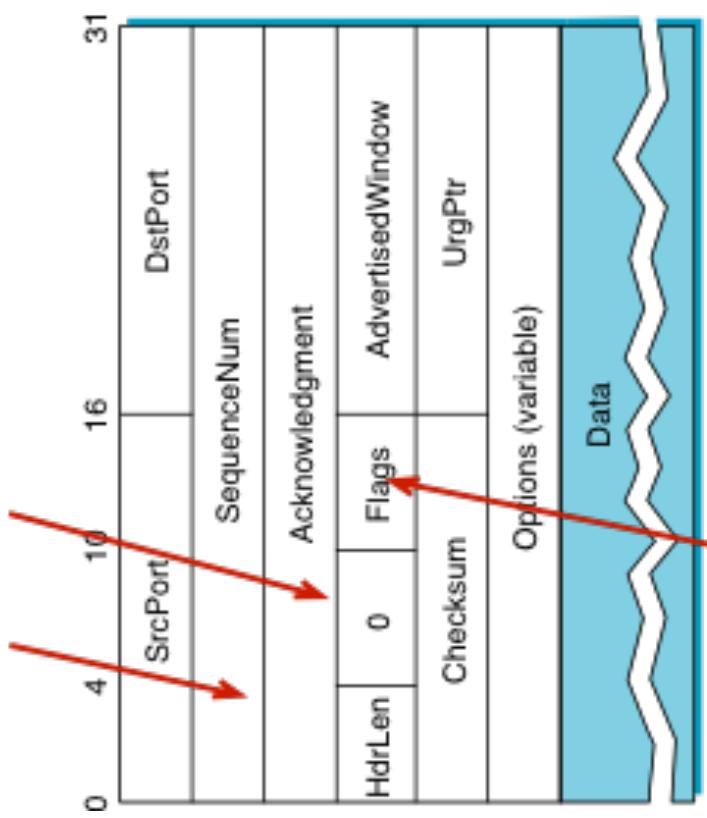
TCP三次握手



TCP三次握手



Seq, Ack字段可以存放序号数据



标志位字段通过多个bit标记是否为
SYN/FIN/ACK等建立连接

client发送连接建立请求(SYN=1,Seq=x)至server
Server回复确认前序请求(ACK=1,Ack=x+1), 同时请求(SYN=1,Seq=y)
Client回复确认前序请求(ACK=1,ACK=y+1)

TCP: 建立连接

- 在数据传输之前完成

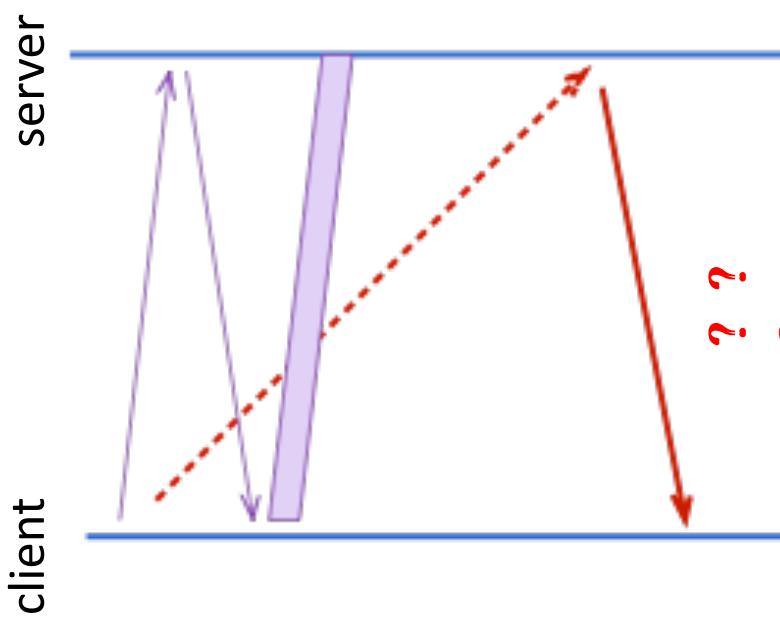
- 基于三次握手
 - 在两个TCP对端之间交换三次消息
 - 协商初始序号
- 初始的序号随机生成
- Acknowledgment字段值一般比对端发送的序号大1
- 为什么使用随机序号？



为什么要三次握手？



- TCP连接建立的三次握手 Three-way handshake
- 主要是为了防止已经失效的连接请求报文段又被接收方收到而导致的错误



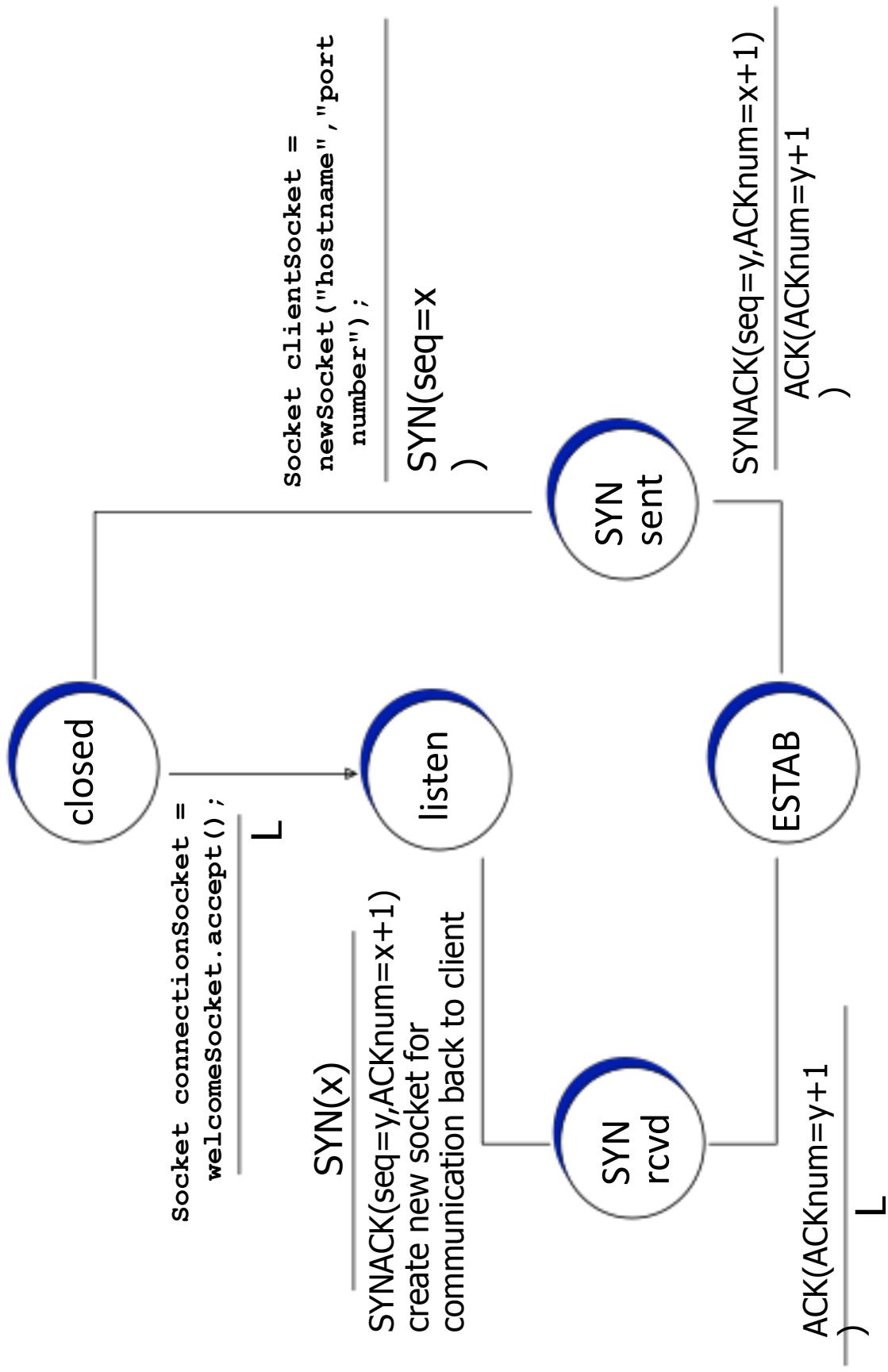
如果client没有收到server的确认，可能会导致下面的情况：

client向server发起多次连接建立请求，其中一次被server正确接收，完成了数据传输

之后，某个client发送的已经失效的连接建立请求又抵达了server，server误以为是新的请求，对其进行回复，并进入等待数据传输的状态，导致连接资源被占用和浪费了

采用三次握手，可以避免“已经失效的连接建立请求”的干扰

TCP 3-way handshake: FSM

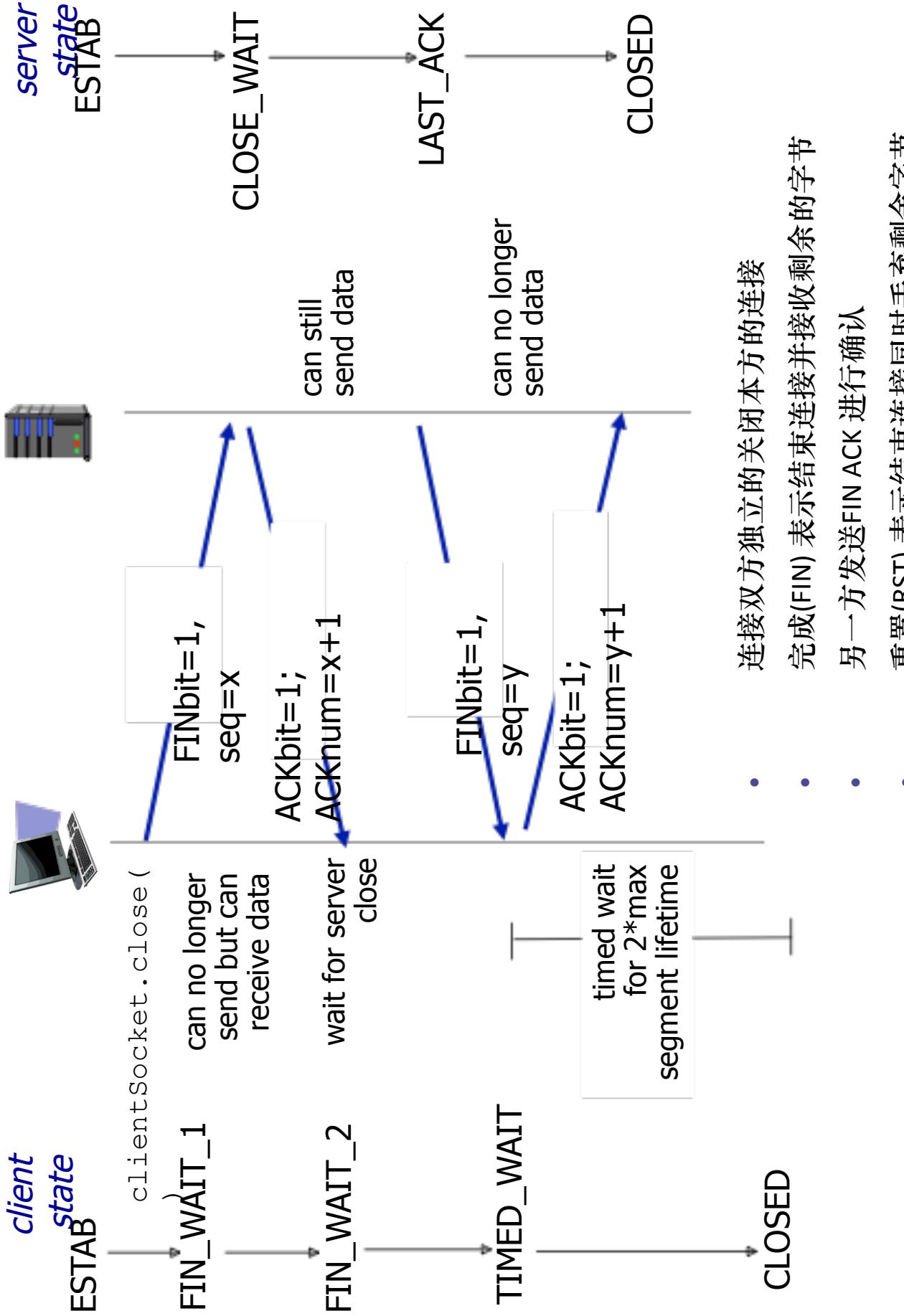


TCP: 終止連接

- client, server each close their side of connection
- send TCP segment with FIN bit = 1
- respond to received FIN with ACK
 - on receiving FIN, ACK can be combined with own FIN
 - simultaneous FIN exchanges can be handled



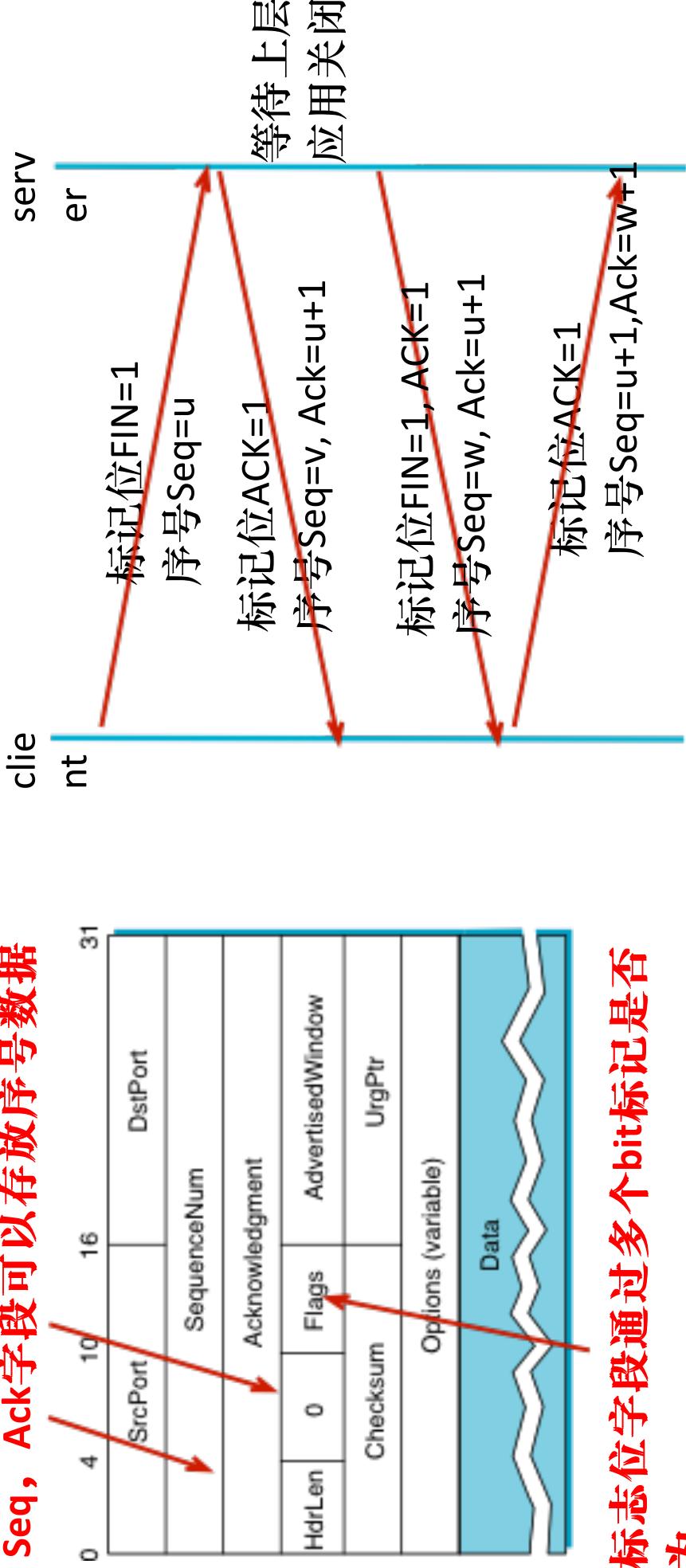
TCP: 終止连接





TCP连接断开：四次握手

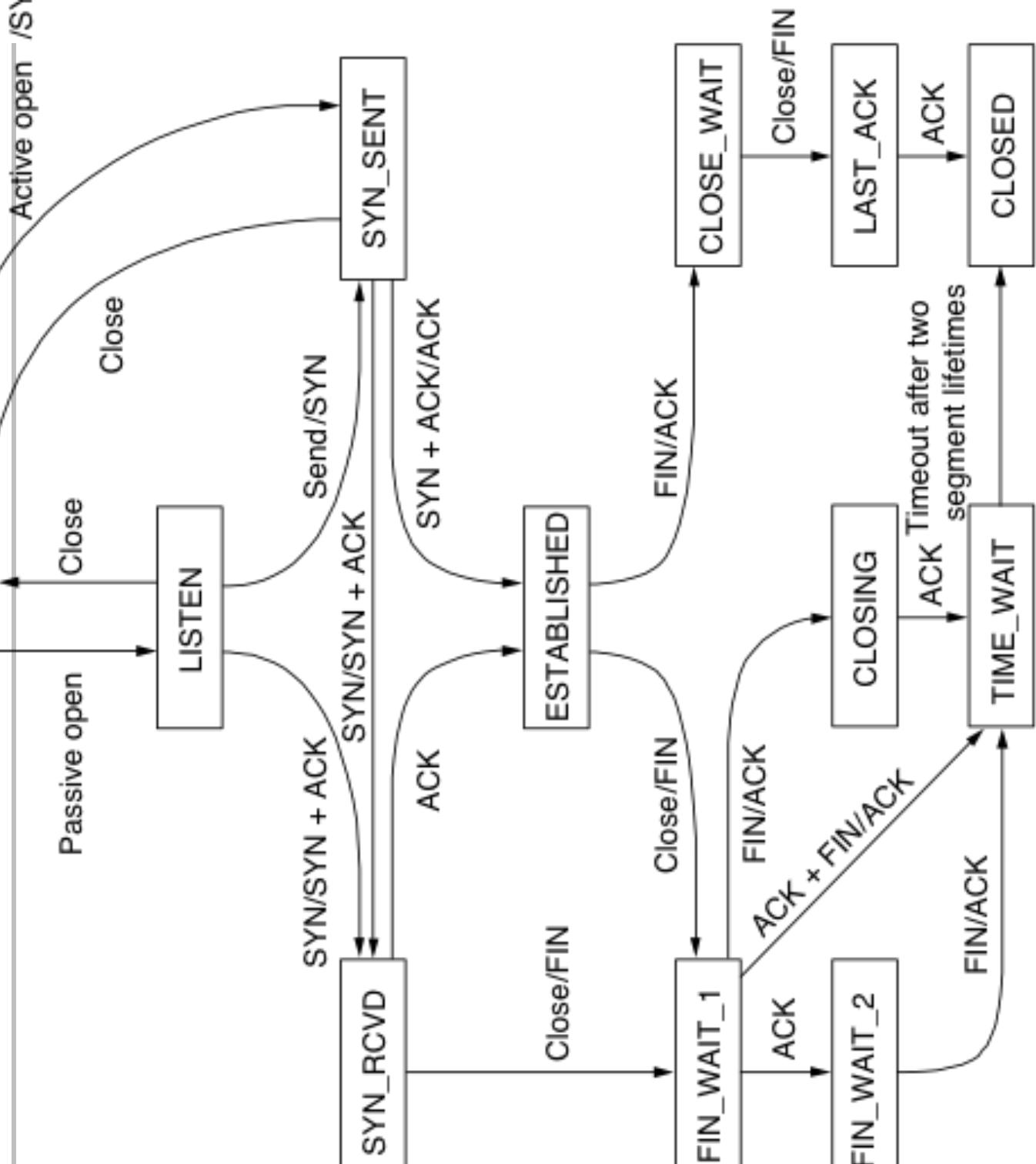
Seq, Ack字段可以存放序号数据



标志位字段通过多个bit标记是否为
SYN/FIN/ACK等

四次握手断开连接
client发送连接断开请求(SYN=1,Seq=u)至server
Server回复确认前序请求(ACK=1,Ack=u+1), 同时继续发送待传输数据(Seq=v)
Server等待上层应用数据传输完毕之后, Server重复确认前序请求(ACK=1,Ack=u+1),
同时发送连接断开请求(SYN=1,Seq=w)
Client回复确认前序请求(ACK=1,ACK=w+1,Seq=u+1), 继续等待2MSL时间后关闭

TCP 状态转换图(状态, 事件 / 动作)



提纲

- 引言
- 核心问题: 进程间如何通信
- 简单多路分解(UDP)
- 可靠字节流(TCP)
- 端到端的问题
- 报文段格式
- 连接的建立和终止
- 滑动窗口算法再讨论
- 触发重传
- 自适应边界
- TCP扩展
- 其他设计选择
- 总结



TCP 滑动窗口算法

- TCP采用改进的滑动窗口算法 实现

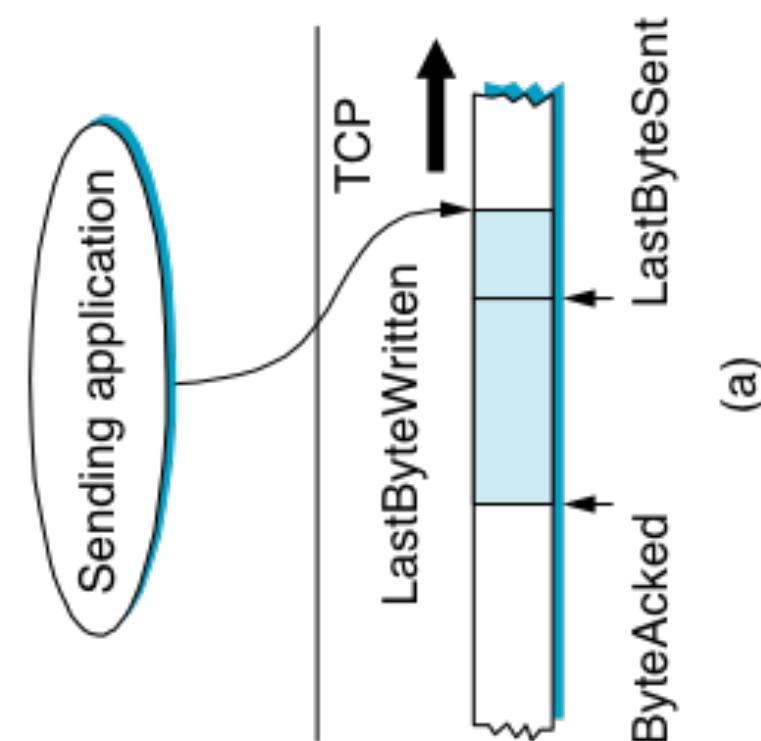
- 保证数据的可靠传送
- 确保数据的有序传送
- 流量控制(基于变化的 AdvertisedWindow 字段)

- 滑动窗口算法采用变化的接收窗口大小
- 接收方通知发送方其窗口大小, 该值可能随时间变化
- 接收窗口大小通过TCP首部的AdvertisedWindow 字段描述

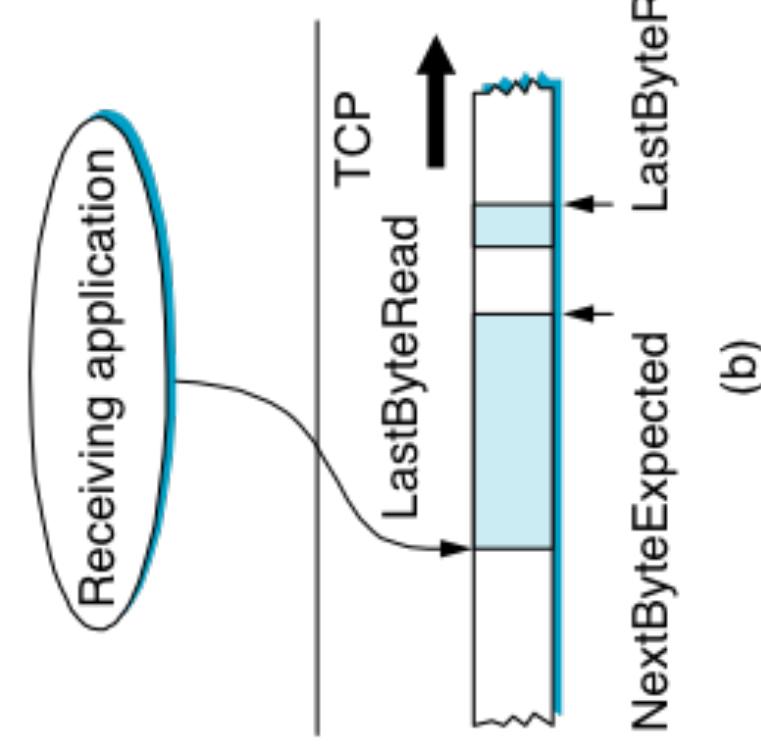


TCP 滑动窗口算法

1. 可靠有序的数据传送

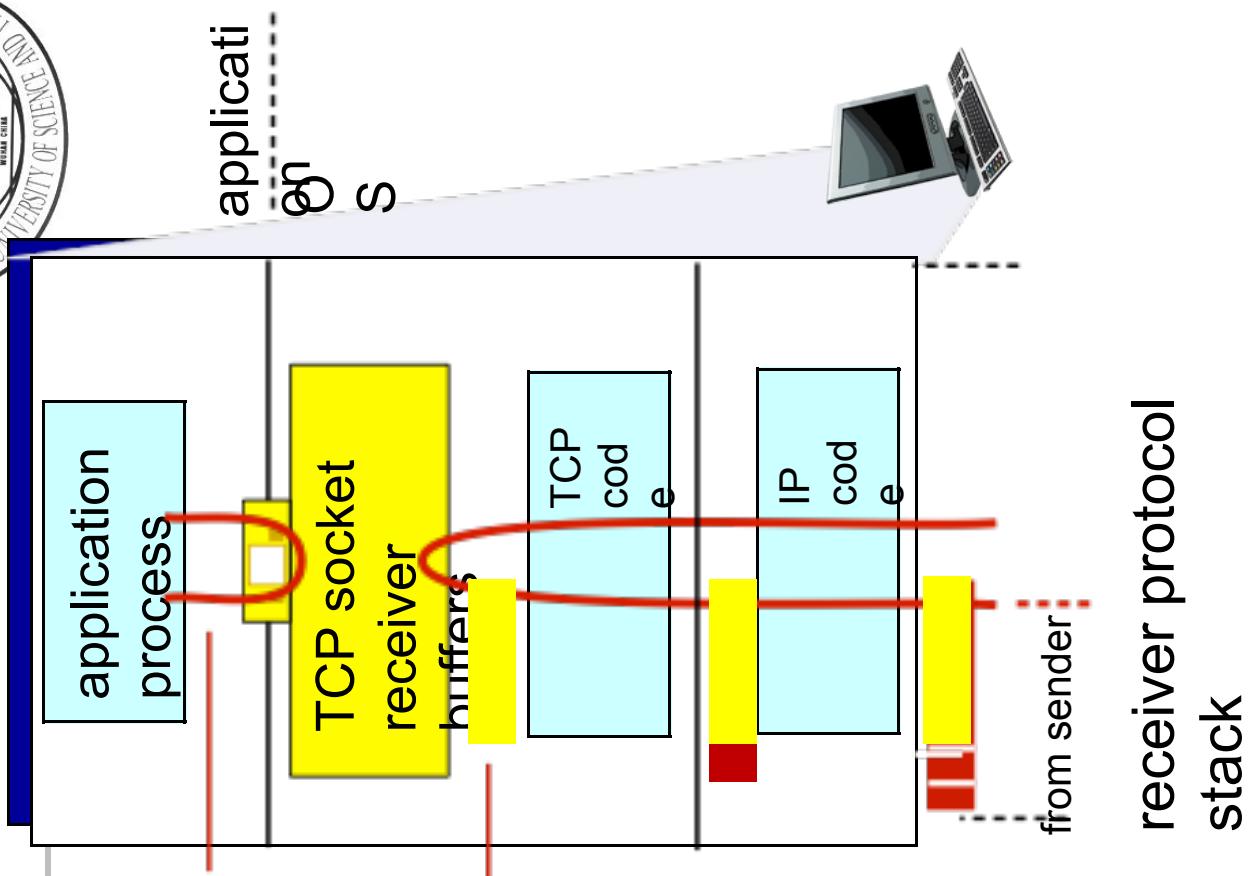


$\text{LastByteAcked} \leq \text{LastByteSent}$
 $\text{LastByteSent} \leq \text{LastByteWritten}$



$\text{LastByteRead} \leq \text{NextByteExpected}$
 $\text{NextByteExpected} \leq \text{LastByteRcvd} + 1$

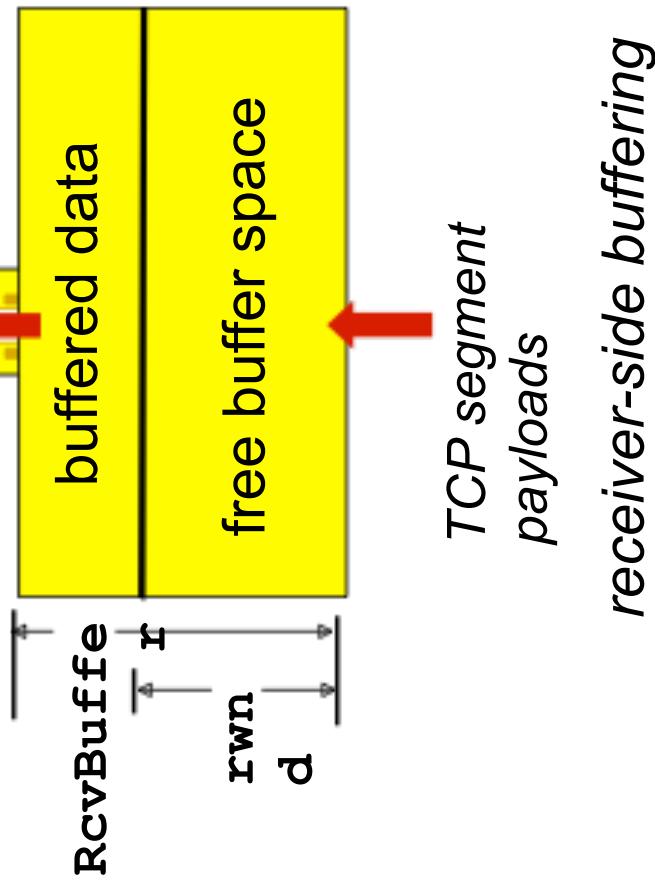
TCP滑动窗口算法的流量控制



flow control
receiver controls sender, so
sender won't overflow receiver's
buffer by transmitting too much,
too fast

TCP滑动窗口算法的流量控制

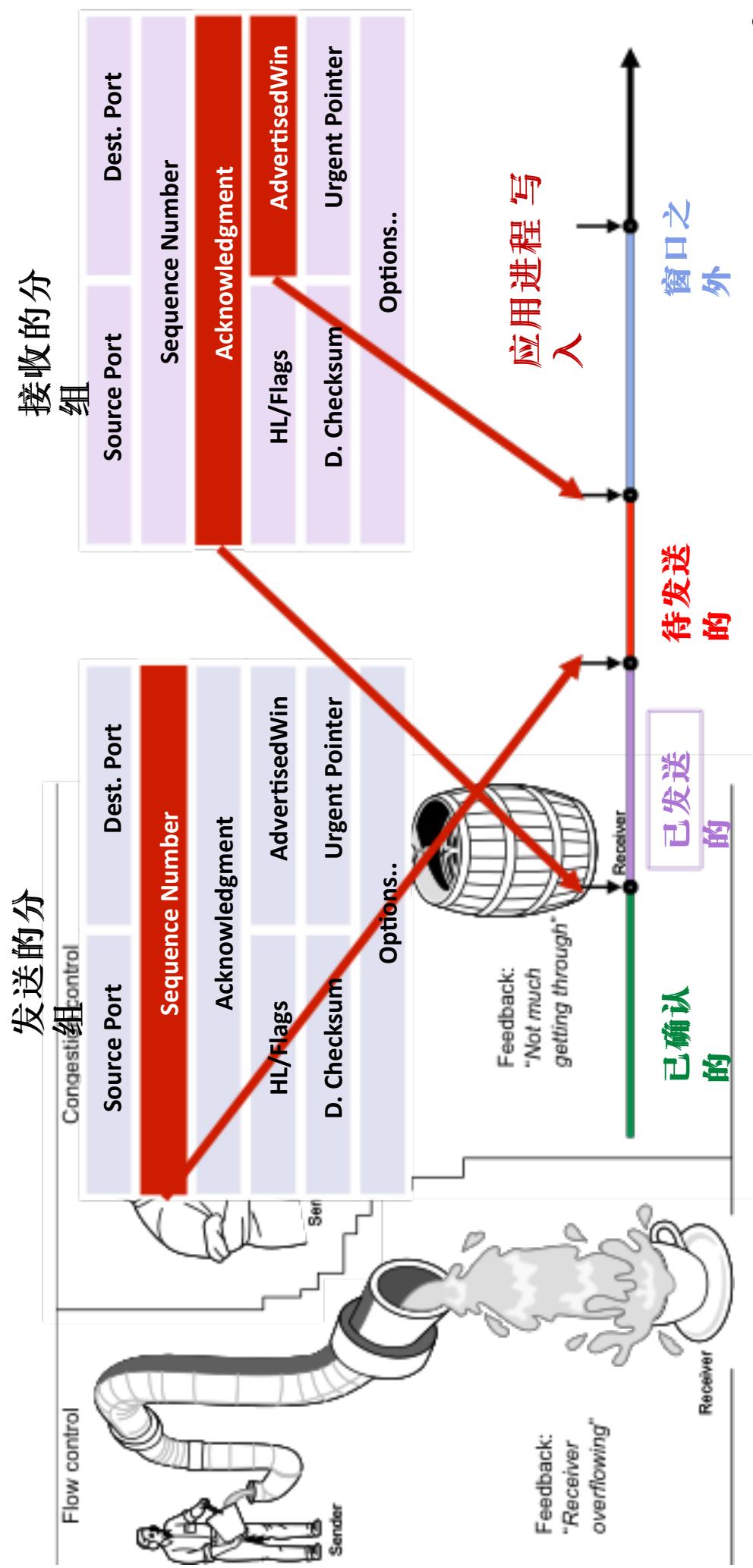
- receiver “advertises” free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
 - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
 - many operating systems autoadjust **RcvBuffer**
 - sender limits amount of unacked (“in-flight”) data to receiver’s **rwnd** value
 - guarantees receive buffer will not overflow



TCP滑动窗口算法

2. 流量控制

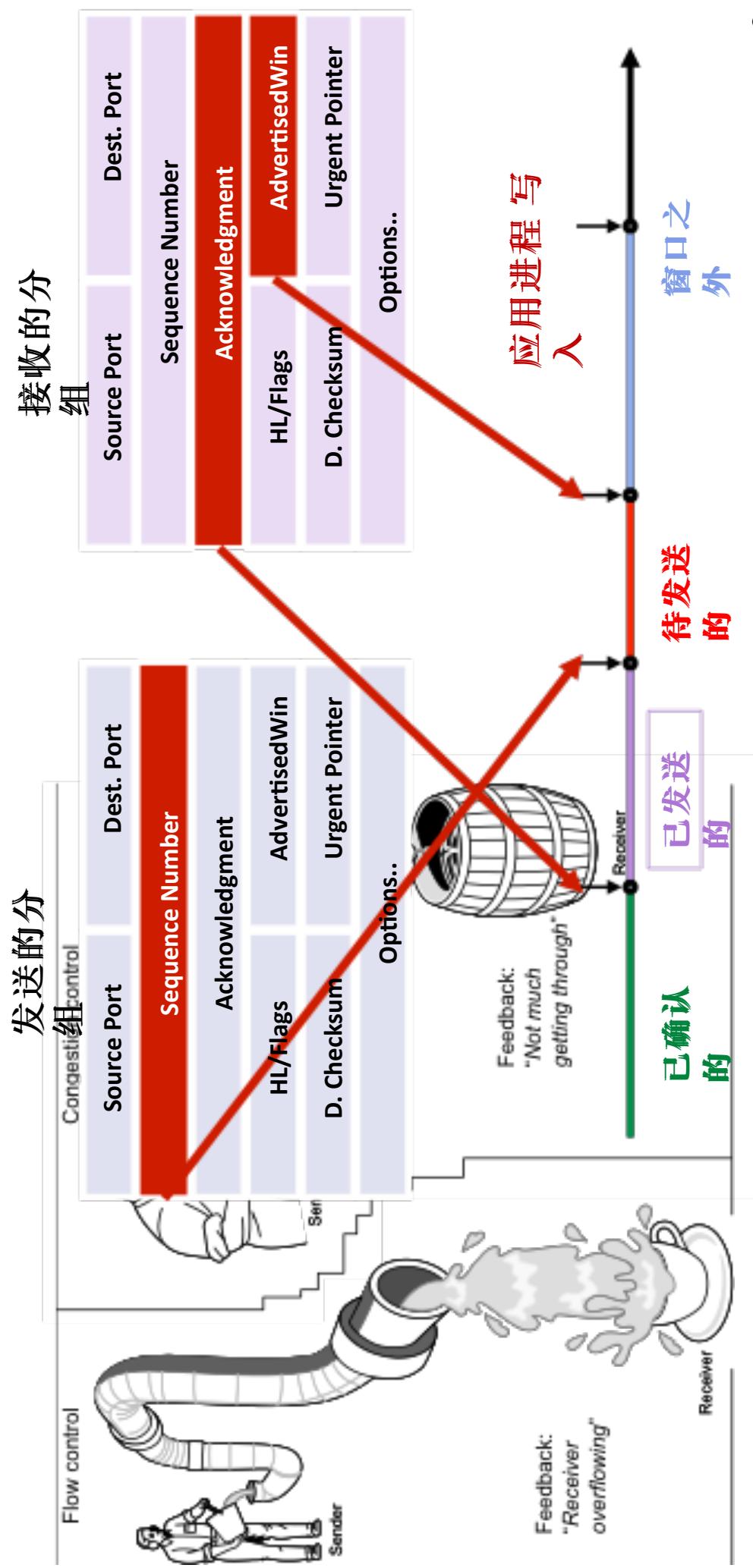
发送方在接收方来得及处理的条件下尽可能快地发送数据
发送方根据接收方的需求动态调整发送方的窗口大小、



TCP滑动窗口算法

2. 流量控制

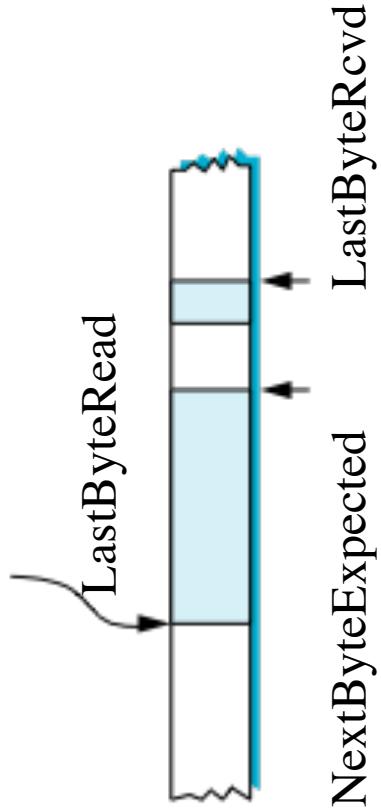
发送方在接收方来得及处理的条件下尽可能快地发送数据
发送方根据接收方的需求动态调整发送方的窗口大小、



TCP滑动窗口算法

2. 流量控制

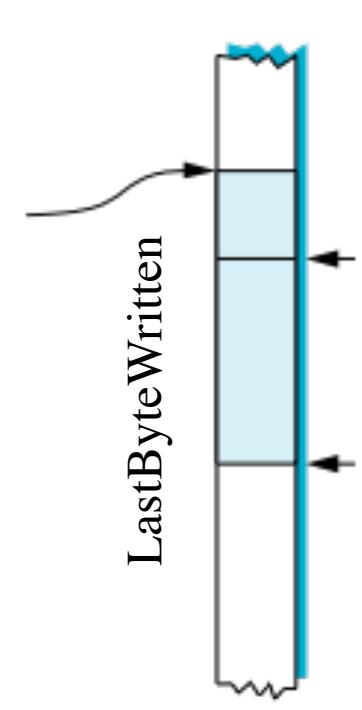
- AdvertisedWindow字段(接收方的角度):



$$\begin{aligned} & \text{LastByteRcvd} - \text{LastByteRead} \\ & \leq \text{MaxRcvBuffer} \end{aligned}$$

$$\begin{aligned} & \text{AdvertisedWindow} = \text{MaxRcvBuffer} \\ & - (\text{NextByteExpected} - 1) - \text{LastByteRead} \end{aligned}$$

- AdvertisedWindow字段(发送方的角度):



$$\begin{aligned} & \text{LastByteSent} - \text{LastByteAcked} \\ & \leq \text{AdvertisedWindow} \end{aligned}$$

$$\begin{aligned} & \text{EffectiveWindow} = \text{AdvertiseWindow} - \\ & (\text{LastByteSent} - \text{LastByteAcked}) \end{aligned}$$

LastByteWritten – LastByteAcked
≤ MaxSendBuffer
Sender's view



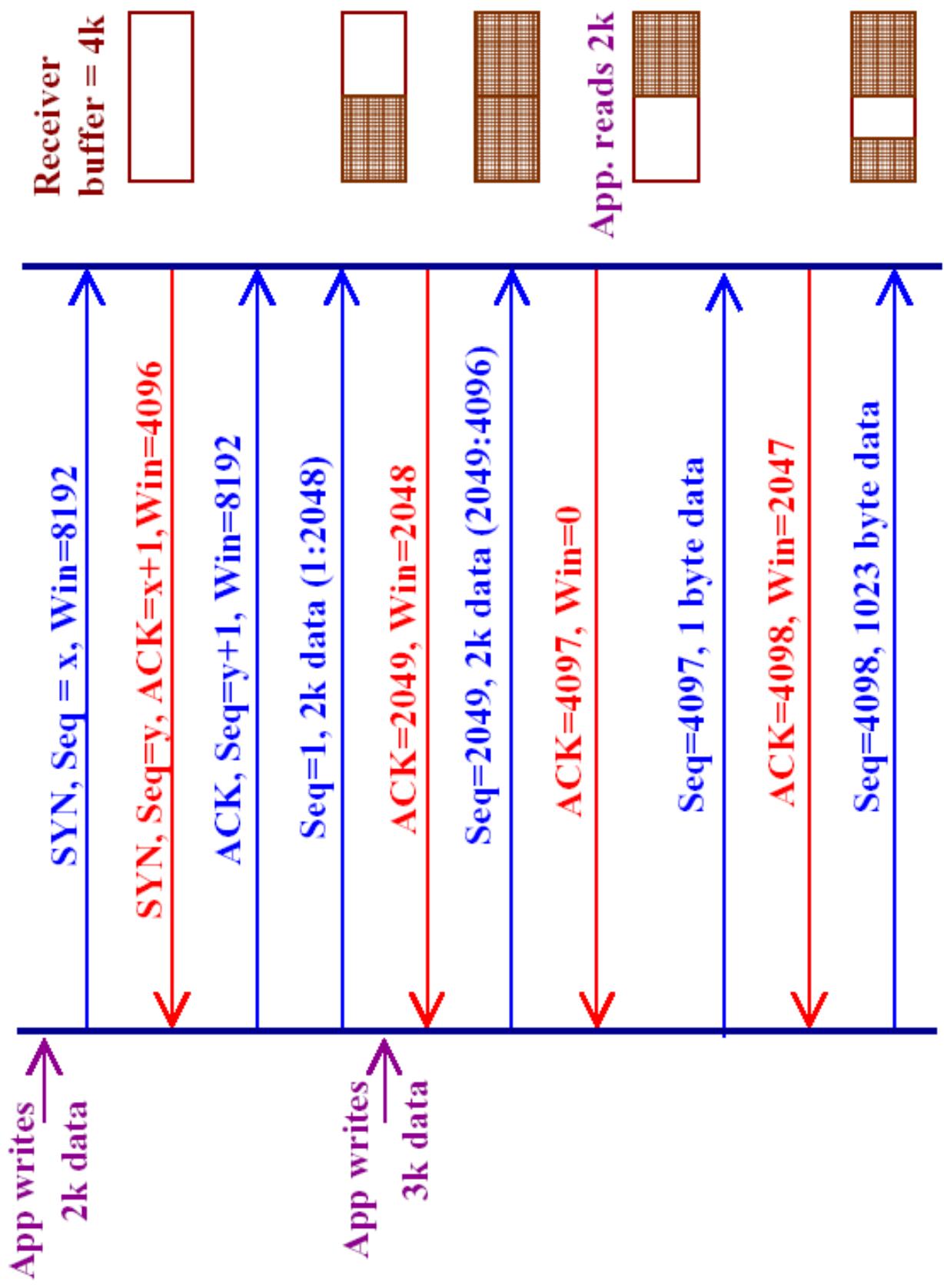
TCP滑动窗口算法

2. 流量控制(续)

- TCP 操作
- 连接建立: 接收方将其缓存大小写入 AdvertisedWindow 字段
 - 第一: 发送方可以发送不超过 $\min(\text{available data}, \text{AdvertisedWindow})$ 的数据
 - 其后: 接收方对收到的数据段进行确认, 并 AdvertisedWindow 字段利用通告发送方其当前可获得的缓存大小
 - 发送方可以发送不超过 $\min(\text{available data}, \text{AdvertisedWindow} - \text{Amount of unacknowledged data})$ 的数据
 - 如果 AdvertisedWindow == 0, 发送方继续发送 1 字节的数据段请求当前的 AdvertisedWindow

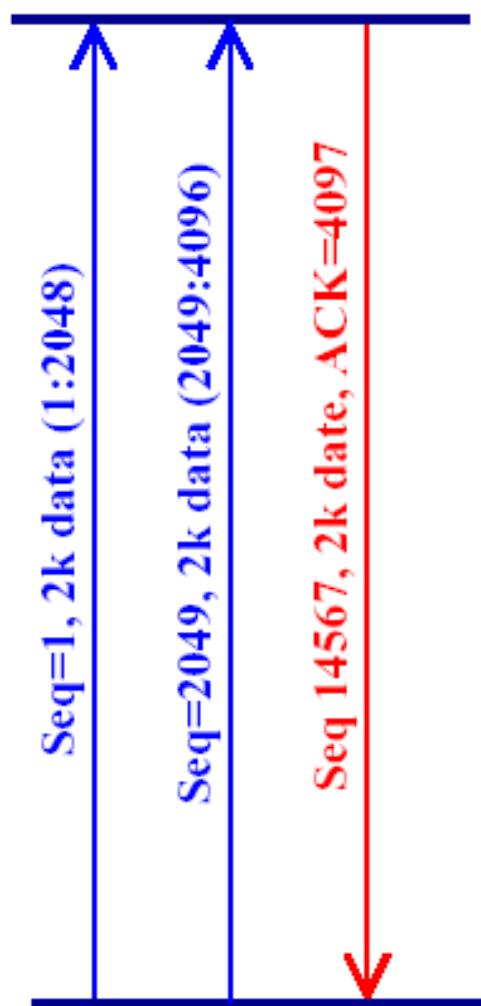


TCP 流量控制: 示例



TCP流量控制:示例

- 确认包括期望接收的下一个字节



- 确认携带在数据段中,通过首部的ACK标志位进行标识



TCP 流量控制的深入讨论

- 为什么发送方周期性的发送探测数据段?
- 接收方简单的对来自发送方的数据段进行确认，而其自身从不发起任何活动.
- TCP 设计原则
 - 聪明的发送方/笨拙的接收方
- TCP中接收方被设计成尽可能的简单





TCP滑动窗口算法

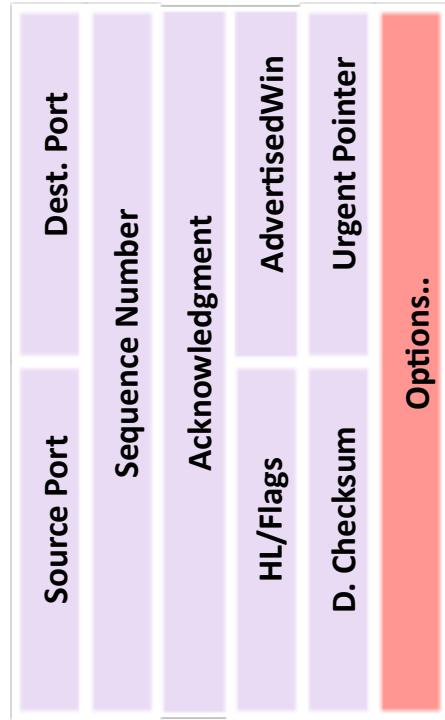
3. 防止回绕

- 通常的方法
- 序号空间的大小是窗口空间大小的2倍
然而，小组的生命周期可能会很长

Bandwidth **Time until Wraparound**

T1 (1.5 Mbps)	6.4 hours
Ethernet (10 Mbps)	57 minutes
T3 (45 Mbps)	13 minutes
FDDI (100 Mbps)	6 minutes
STS-3 (155 Mbps)	4 minutes
STS-12 (622 Mbps)	55 seconds
STS-24 (1.2 Gbps)	28 seconds

32位序号空间多久被用完(发送回绕)



对序号字段空间进行扩展

-0-



TCP滑动窗口算法

4. 保持管道满载

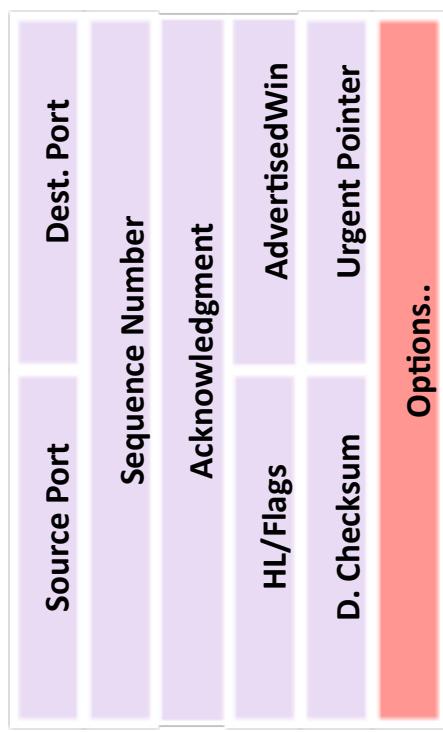
通常的方法

发送方的窗口大小大于时延带宽积

Bandwidth Delay × Bandwidth Product

T1 (1.5 Mbps)	18 KB
Ethernet (10 Mbps)	122 KB
T3 (45 Mbps)	549 KB
FDDI (100 Mbps)	1.2 MB
STS-3 (155 Mbps)	1.8 MB
STS-12 (622 Mbps)	7.4 MB
STS-24 (1.2 Gbps)	14.8 MB

100ms RTT条件下所需的窗口大小，最初TCP首部仅仅预留16比特的字段长度，仅允许64KB大小的 AdvertisedWindow.



TCP的解决方案

- 16比特的AdvertisedWindow 大于时延带宽积
- 对 AdvertisedWindow 字段大小进行扩展

TCP 设计：问题及解决方案



No.	问题及挑战	解决方案	章节
1	连接建立	建立：三次握手 终止：四次握手	5.2.3
2	超时定时器问题		
3	分组乱序到达	基于窗口的缓存管理	5.2.4
4	流量控制	通过AdvertisedWindow实现 基于窗口的流量控制	5.2.4
5	拥塞控制		
6	协议扩展	TCP首部的Seq和 AdvertisedWindow字段扩展	5.2.4

提纲

- 引言
- 核心问题: 进程间如何通信
- 简单多路分解(UDP)
- 可靠字节流(TCP)
- 端到端的问题
- 报文段格式
- 连接的建立和终止
- 滑动窗口算法再讨论
- 触发重传
- 自适应传输
- 记录边界
- TCP扩展
- 其他设计选择
- 总结



TCP中的触发传输



缺省情况下TCP有三种机制触发一个报文段的传输：

(1) 当收集到的字节数达到MSS

- MSS(最大数据段长度)通常设置为MTU-IP_首部长度-TCP首部
- 比喻：公汽站发车的场景下，因为车坐满了乘客而发车

(2) 发送进程明确要求TCP发送

- Push操作：发送应用进程调用该操作使得TCP发出缓存中所有未发送的字节
- 比喻：公汽站发车的场景下，因为目的地要求而发车

(3) 定时器激活

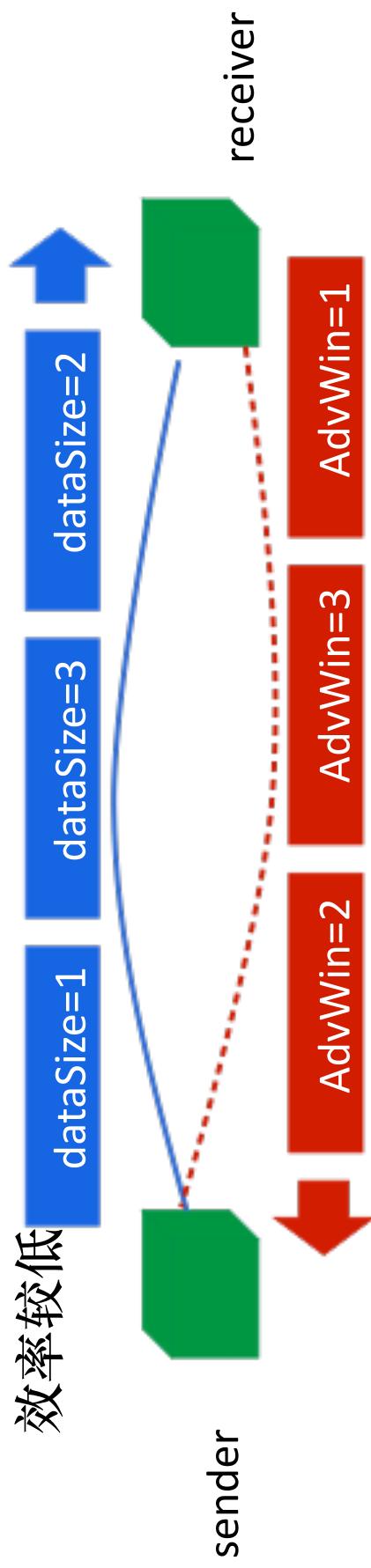
- 每到定时器到时发送方发送当前缓存中所有有待发送的字节
- 模拟：公汽站发车的场景下，根据时刻表发车

TCP触发送传输的问题-1



- 问题1：傻瓜窗口症状
 - 基本的触发送机制没有考虑流量控制的影响
 - 如果TCP发送的窗口大小严格受控于发送方的通知窗口，可能导致“傻瓜窗口症状”

例如，接收方缓存的可用容量被上层应用取走后，分别为2、3、1，那么相应的发送方发送的报文段为2、3、1个字节，效率较低



- 解决方案：
 - 接收方延迟回复通知窗口

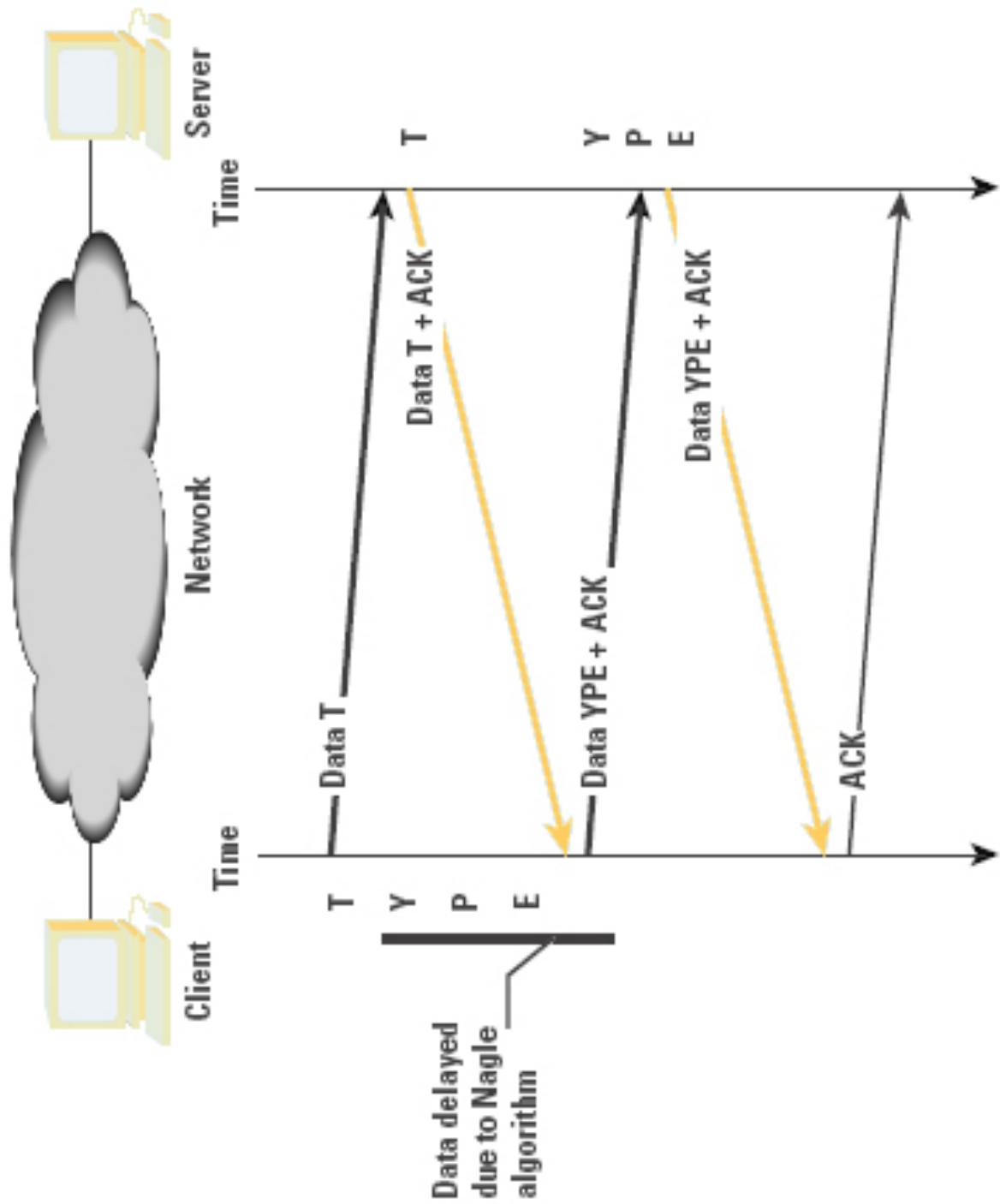
TCP触发光传输的问题-2



- 问题
 - 当应用层交付数据速度较快，而网络速率较慢时，较小的报文段浪费带宽
- 解决方案：Nagle算法，利用ACK实现自计时
 - 只要TCP发出了数据，发送方终究会收到一个ACK。
 - 可以把这个ACK视为激活的定时器，触发传输更多的数据

```
When the application produces data to send
if both the available data and the window ≥ MSS
    send a full segment
else
    if there is unACKed data in flight
        buffer the new data until an ACK arrives
    else
        send all the new data now
```

Nagle 算法



TCP 设计：问题及解决方案



No.	问题及挑战	解决方案	章节
1	连接建立	建立：三次握手 终止：四次握手	5.2.3
2	超时定时器问题		
3	分组乱序到达	基于窗口的缓存管理	5.2.4
4	流量控制	通过AdvertisedWindow通告实现基于窗口的流量控制	5.2.4
5	拥塞控制		
6	协议扩展	TCP首部的Seq和AdvertisedWindow字段扩展	5.2.4
7	傻瓜窗口症状	Nagle 算法：基于ACK自计时	5.2.5

提纲

- 引言
- 核心问题: 进程间如何通信
- 简单多路分解(UDP)
- 可靠字节流(TCP)
- 端到端的问题
- 报文段格式
- 连接的建立和终止
- 滑动窗口算法再讨论
- 触发重传
- 自适应边界
- TCP扩展
- 其他设计选择
- 总结



TCP超时重传



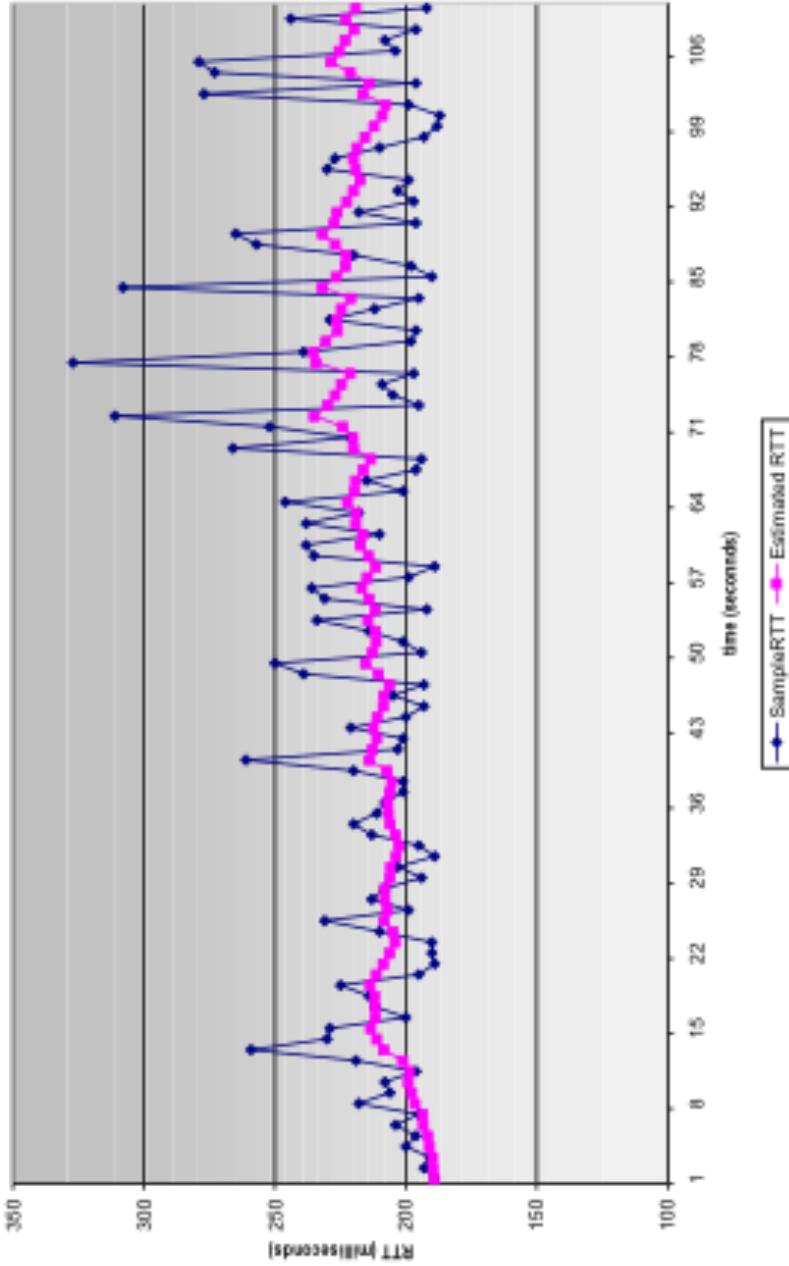
- TCP超时重传的问题
- 根据原始可靠传输ARQ机制，出现定时器超时时会触发重传
 - 但是，TCP连接的RTT时延难以确定
 - TCP连接经过的路径可能发生变化，导致RTT发生变化
 - 沿途路由器队列排队时延会根据负载的波动发生变化
- TCP的解决方案
 - 采用自适应机制估计当前连接的RTT的值，进而设置超时的设定值

自适应重传

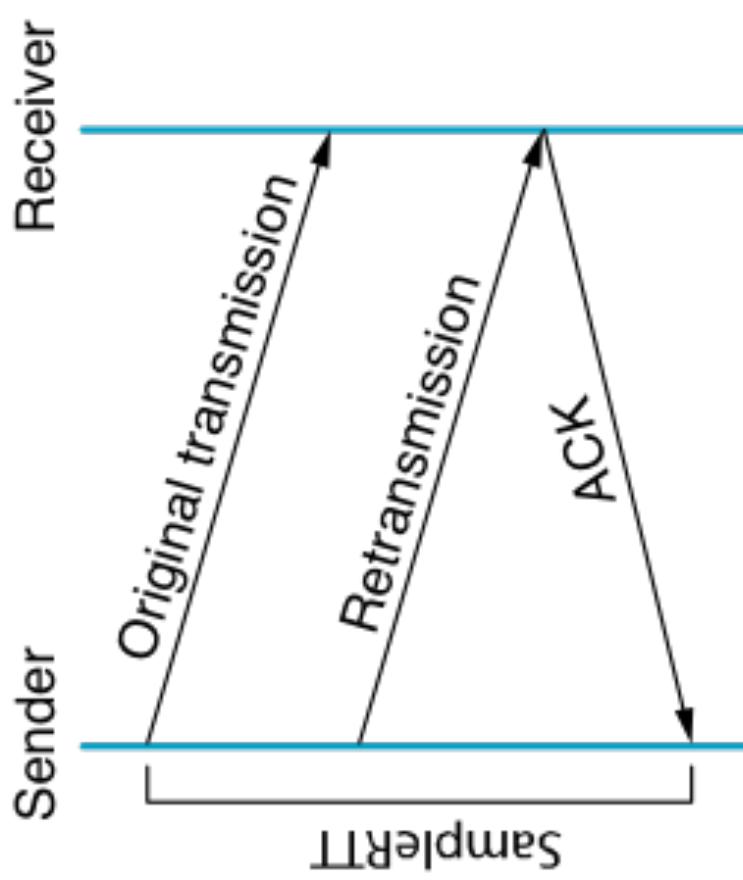
原始算法

$$\text{EstimatedRTT} = a \times \text{EstimatedRTT} + (1-a) \times \text{SampleRTT}$$
$$\text{Timeout} = 2 \times \text{EstimatedRTT}$$

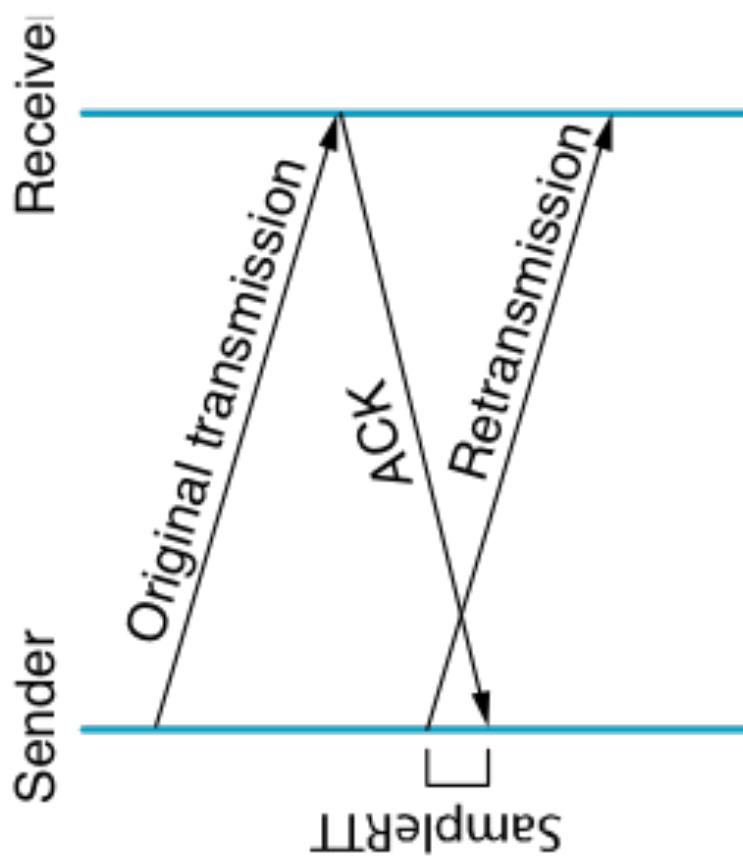
- 平滑因子 a : 原始TCP规范建议值为 $0.8 \sim 0.9$
- 样本 RTT: ACKed 到达时间 - sent 时间
-
-



对RTT估计的原始方法的问题



(a)



(b)

问题：无法区分当前分组的ACK与重传分组的ACK

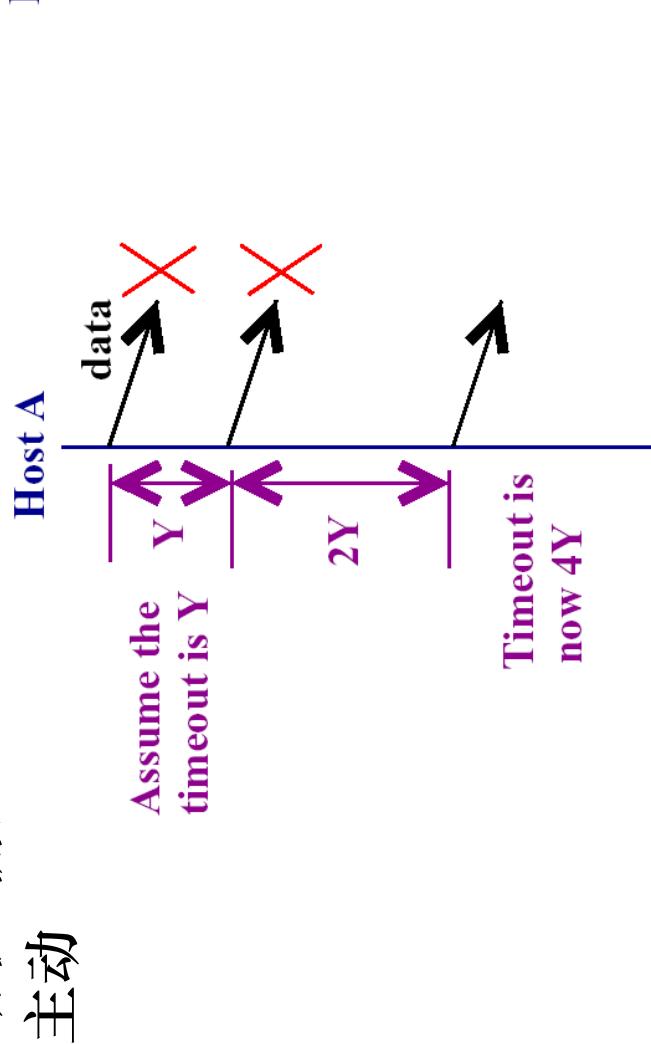
Karn/Partridge 算法



- 提出了两点改进
 - 当TCP重传数据段时停止计算RTT的样本值
 - 每次重传后设置下次超时的值为上次的两倍
- 指数退避
 - 动机：拥塞最多导致数据段丢失，TCP源端对超时的反应不应该太主动

指数退避

- 动机：拥塞最多导致数据段丢失，TCP源端对超时的反应不应该太主动



问题：没有必要对较小的estimated RTT乘以2；样本变化很大时超时值应该远不止是estimated RTT的2倍



Jacobson/Karels 算法

- 原始算法中未考虑RTT的波动

- 建议

$$\text{Difference} = \text{SampleRTT} - \text{EstimatedRTT}$$

$$\text{EstimatedRTT} = \text{EstimatedRTT} + (d \times \text{Difference})$$

$$\text{Deviation} = \text{Deviation} + d (|\text{Difference}| - \text{Deviation})$$

$$\text{Timeout} = m \times \text{Estimated RTT} + f \times \text{Deviation}$$

- 其中 d 是0和1之间的小数, m 缺省设置为1, f 缺省设置为4
- **Deviation**表示估计的RTT与实际的RTT之间的误差, 样本变化小意味着estimated RTT更可信, 反之依然注意重传基于选择性的ACK(SACK), TCP的另一项扩展

TCP round trip time, timeout



- timeout interval: EstimatedRTT plus “safety margin”
 - large variation in EstimatedRTT -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

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

(typically, $= 0.25$)

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



“safety margin”

TCP 设计：问题及解决方案



No.	问题及挑战	解决方案	章节
1	连接建立	建立：三次握手 终止：四次握手	5.2.3
2	超时定时器问题	采用Jacobson/ Karels算法估计RTT	5.2.6
3	分组乱序到达	基于窗口的缓存管理	5.2.4
4	流量控制	通过AdvertisedWindow通告实现基于窗口的流量控制	5.2.4
5	拥塞控制		
6	协议扩展	TCP首部的Seq和AdvertisedWindow字段扩展	5.2.4
7	傻瓜窗口症状	Nagle 算法：基于ACK自计时	5.2.5

提纲

- 引言
- 核心问题: 进程间如何通信
- 简单多路分解(UDP)
- 可靠字节流(TCP)
- 端到端的问题
- 报文段格式
- 连接的建立和终止
- 滑动窗口算法再讨论
- 触发传输
- 自适应重传
- 记录边界
- TCP扩展
- 其他设计选择
- 总结



其他设计选择

- 面向流的协议vs. 请求响应协议
- 面向字节的协议vs. 面向报文的协议
- 显示的建立/断开连接 vs. 隐式的建立/断开连接
- 基于窗口vs. 基于速率



提纲

- 引言
- 核心问题: 进程间如何通信
- 简单多路分解(UDP)
- 可靠字节流(TCP)
- 端到端的问题
- 报文段格式
- 连接的建立和终止
- 滑动窗口算法再讨论
- 触发传输
- 自适应重传
- 记录边界
- TCP扩展
- 其他设计选择
- 总结



总结

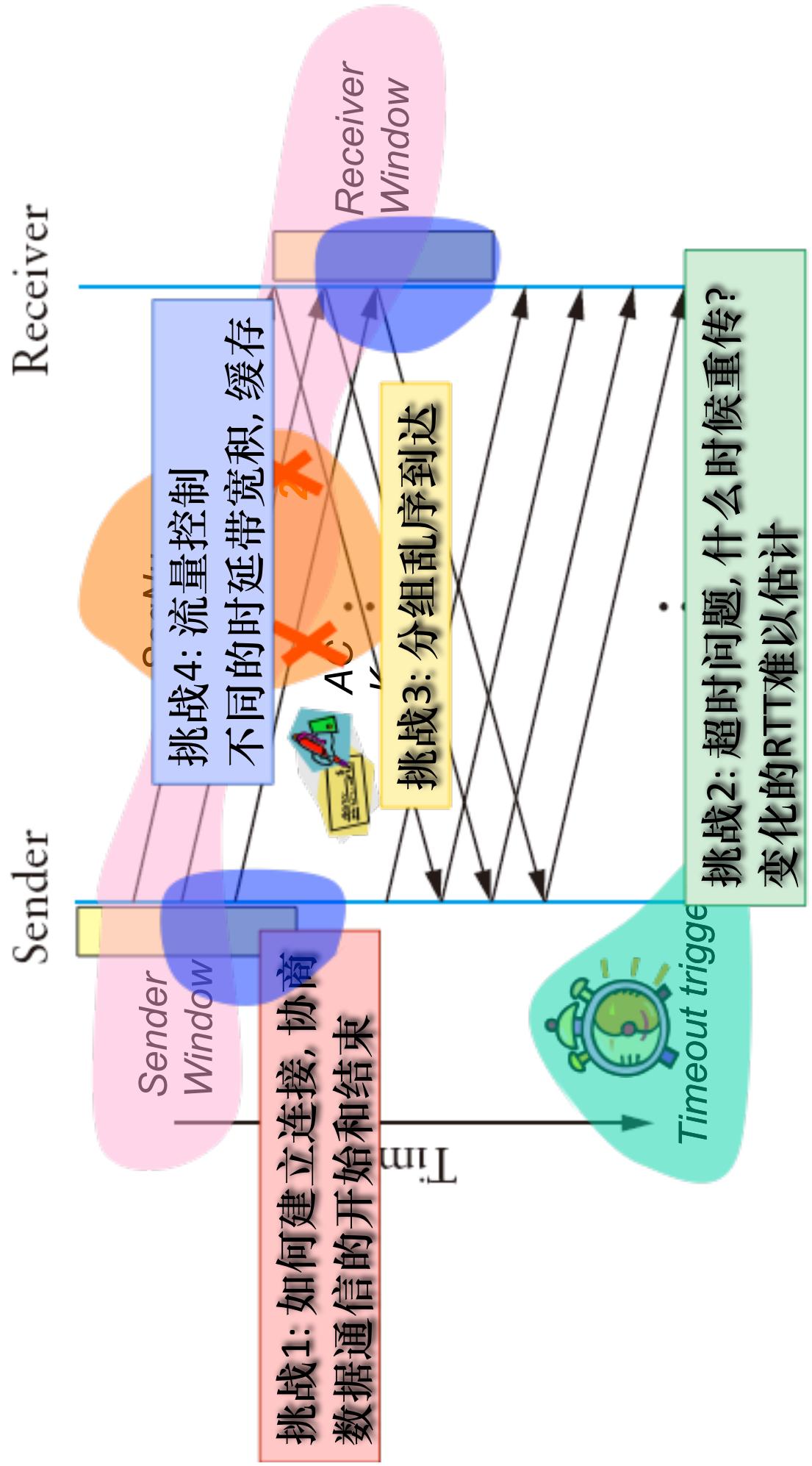


- TCP/IP协议栈中的两种传输层协议
 - UDP: 1980年实现
 - TCP最早于1974年实现
 - 1981年IPv4实现
- UDP
 - 仅在IP基础上增加了一级解多路复用功能
 - 面向报文, 无连接
- TCP
 - 在IP基础上提供了可靠的字节流服务
 - 在基本的滑动窗口协议上进行了很多改进

TCP可靠传输面临的新挑战



底层网络：IP 提供为异构网络不同的主机间通信提供了不可靠的无连接服务



TCP 设计:问题及解决方案



No.	问题及挑战	解决方案
1	连接建立	建立: 三次握手 终止: 四次握手
2	超时定时器问题	采用Jacobson/ Karel's算法估计RTT
3	分组乱序到达	基于窗口的缓存管理
4	流量控制	通过AdvertisedWindow通告实现基于窗口的流量控制
5	拥塞控制	
6	协议扩展	TCP首部的Seq和AdvertisedWindow字段扩展
7	傻瓜窗口症状	Nagle 算法: 基于ACK自计时

基于不可靠IP
的挑战

复杂协议设计
的挑战

谢谢！

华中科技大学
电子信息与通信学院
Email: itec@hust.edu.cn
网址: <http://itec.hust.edu.cn>





- Chapter 5 in L. L. Peterson and B. S. Davie, *Computer Networking: A System Approach (5th edition)*, Morgan Kaufmann, 2012
- Chapter 3 in James F. Kurose and Keith W. Ross, *Computer Networking: A Top-Down Approach (6th edition)*, Pearson Education Inc., 2012
- 吴功宜, 计算机网络 (第3版), 清华大学出版社社, 2011

附录