

第3章 传输层

© 电子科技大学信息与软件工程学院
计算机网络课程组, 2022



理解传输层服务以后的原则:

复用/分解复用
可靠数据传输
流量控制
拥塞控制



学习因特网的传输层协议:

UDP: 无连接传输
TCP: 面向连接传输
TCP 拥塞控制

目录 CONTENT

01 传输层服务

02 多路复用和多路分解

03 无连接传输: UDP

04 可靠数据传输原理

05 面向连接传输: TCP

06 拥塞控制原理

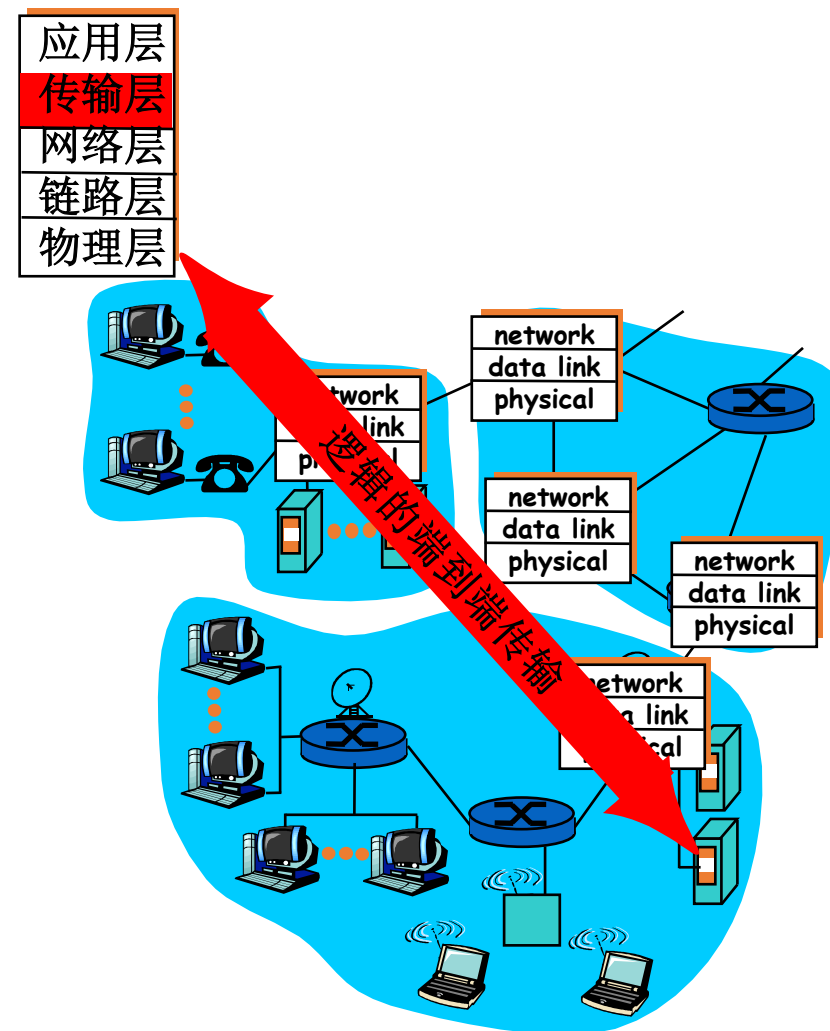
07 TCP 拥塞控制

01 传输层服务



传输层服务和协议

- 在两个不同的主机上运行的应用程序之间提供**逻辑通信**
- **传输层协议运行在端系统**
 - 发送方: 将应用程序报文分成数据段传递给网络层,
 - 接受方: 将数据段重新组装成报文传递到应用层
- **不只一个传输层协议可以用于应用程序**
 - 因特网: TCP 和 UDP



传输层和网络层

传输层: 两个进程之间的逻辑通信

- 可靠, 增强的网络层服务

网络层: 两个主机之间的逻辑通信

Internet 传输层协议

可靠按序递交 (TCP)

拥塞控制

流量控制

连接建立

不提供的服务:

延迟保证

带宽保证

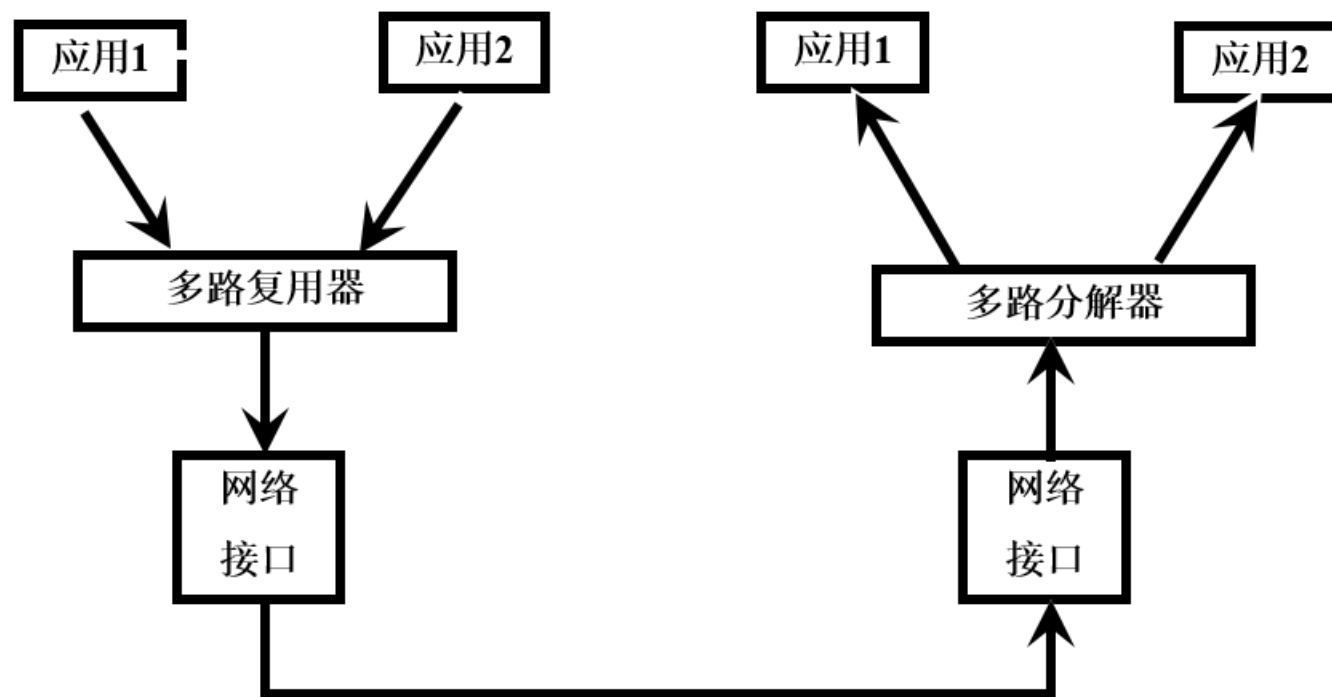
不可靠的无序传递: UDP

“尽力传递” IP的直接扩展

02 多路复用和多路分解





多路复用和多路分解

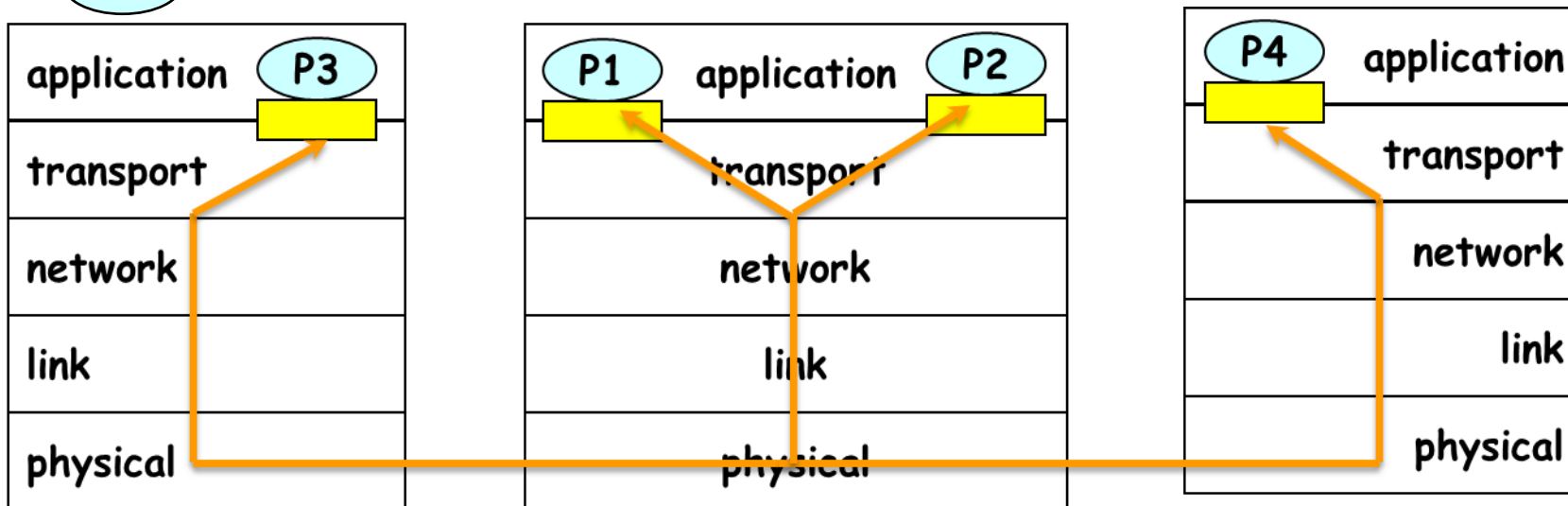


多路复用/多路分解

在接收主机多路分解:
将接收到的数据段传递到
正确的套接字 (多路分解)

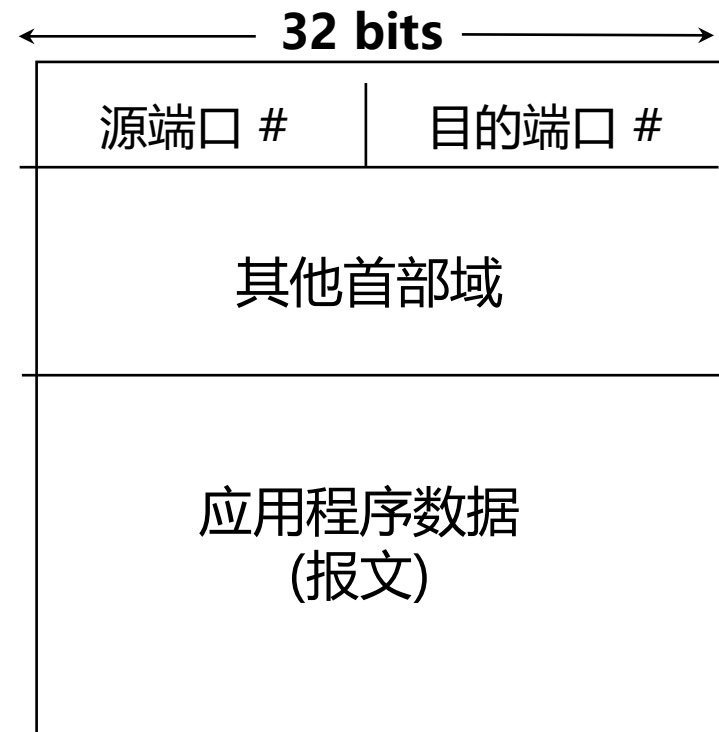
在发送主机多路复用:
从多个套接字收集数据, 用首
部封装数据, 然后将报文段
传递到网络层(多路复用)

 = 套接字  = 进程



多路分解如何工作？

- 主机收到IP数据报
 - 每个数据报有源IP地址，目的IP地址
 - 每个数据报搬运一个数据段
 - 每个数据段有源和目的端口号
(回忆: 对于特定应用程序具有周知端口号)
- 主机用IP地址和端口号指明数据段属于哪个合适的套接字



TCP/UDP 报文段格式

无连接多路分解

用端口号创建套接字:

```
DatagramSocket ServerSocket1 = new
```

```
DatagramSocket(9911);
```

```
DatagramSocket ServerSocket2 = new
```

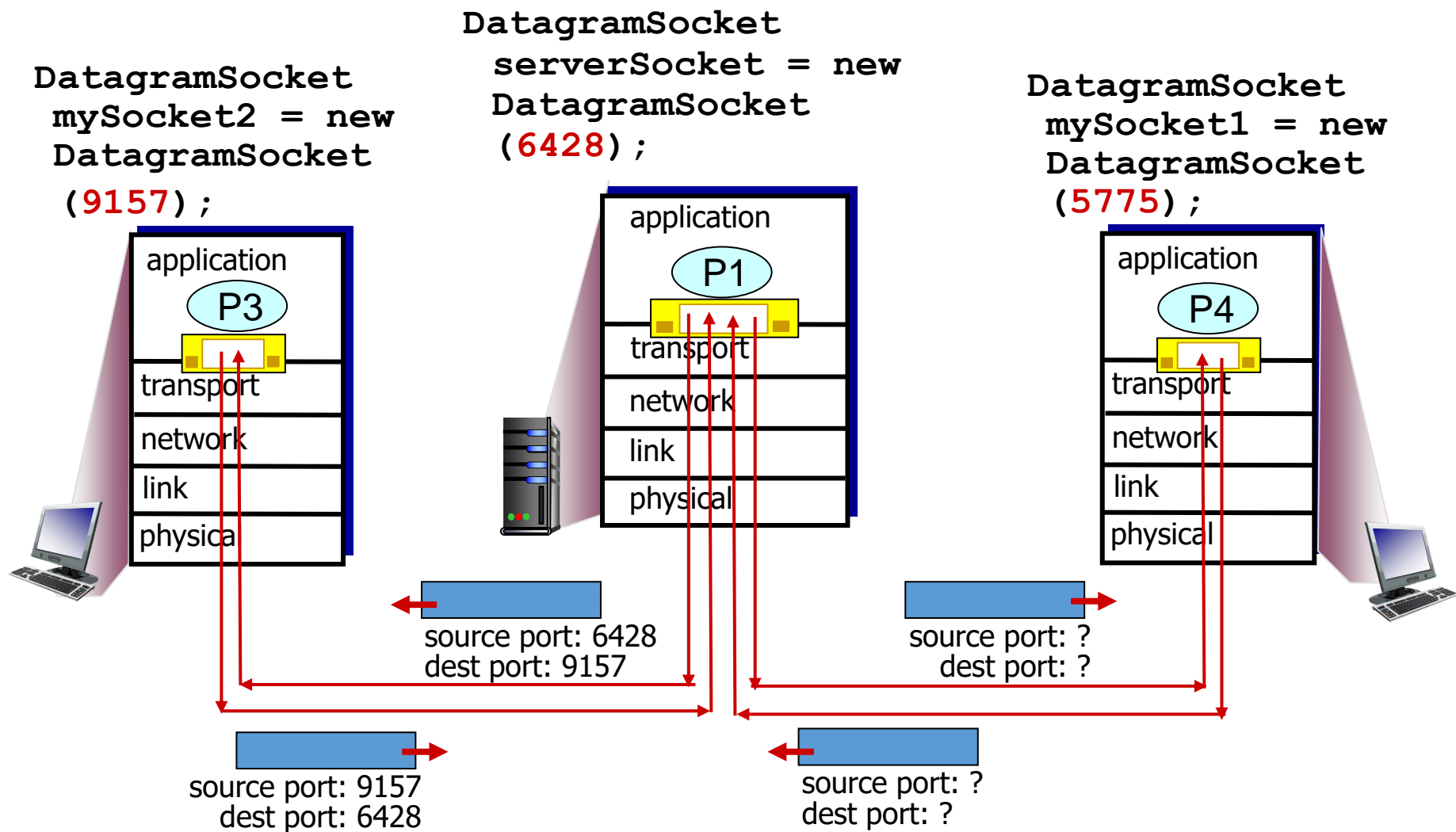
```
DatagramSocket(9922);
```

UDP 套接字由两个因素指定:

(目的IP地址, 目的端口号)

- 当主机收到UDP数据段:
 - 检查数据段中的目的端口号
 - 用端口号指示UDP数据段属于哪个套接字
- 具有不同的源IP地址且/或源端口号, 但具有相同的目的IP地址和目的端口号的IP数据报, 指向同样的套接字

无连接多路分解例子



请求报文段中提供返回地址（包括IP地址和端口号）

面向连接的多路分解

TCP 套接字由4部分指定:

源IP地址

源端口号

目的IP地址

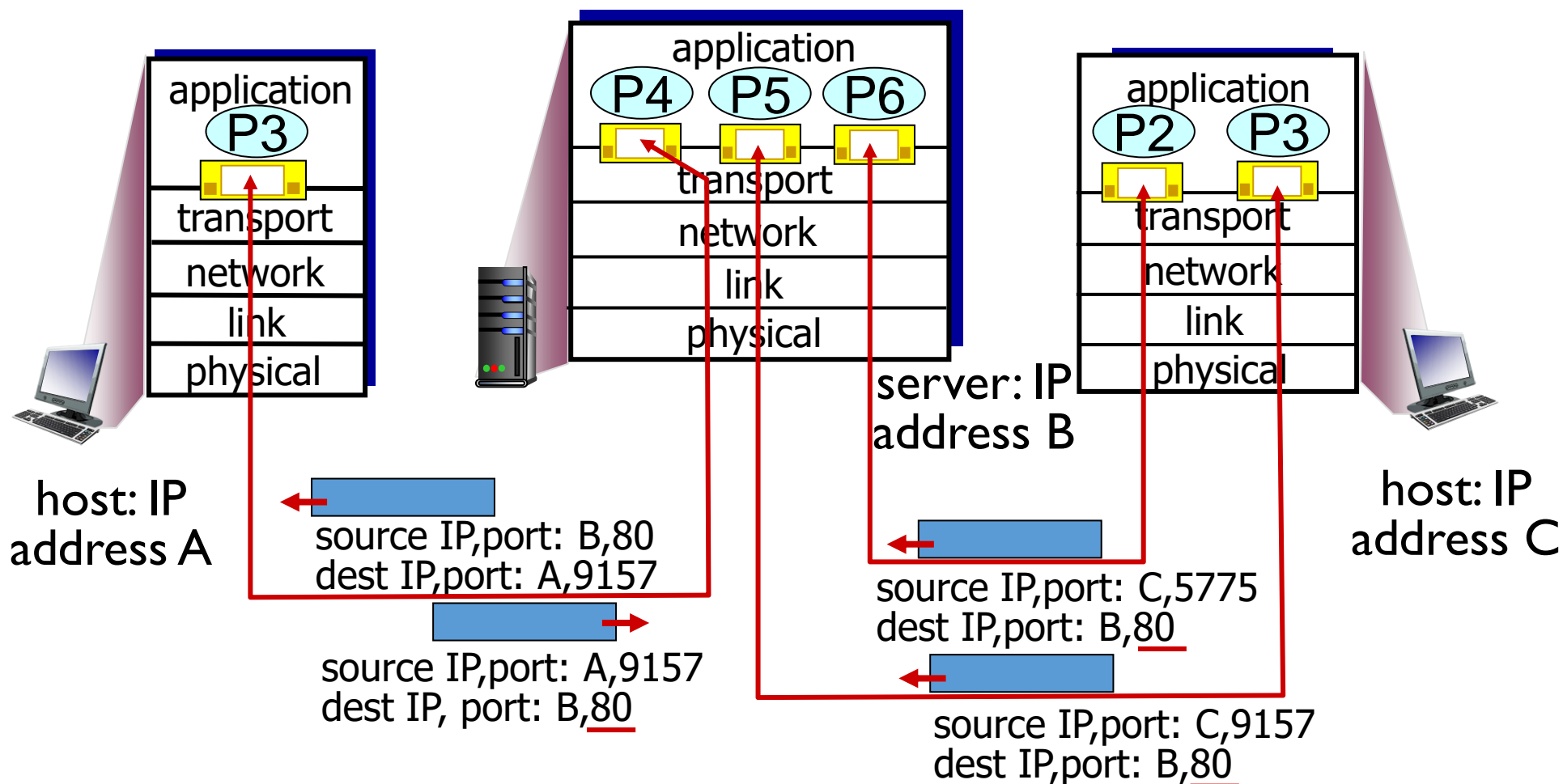
目的端口号

接收主机使用所有四个值将数据段定位到合适的套接字

服务器主机可同时支持很多个TCP 套接字:

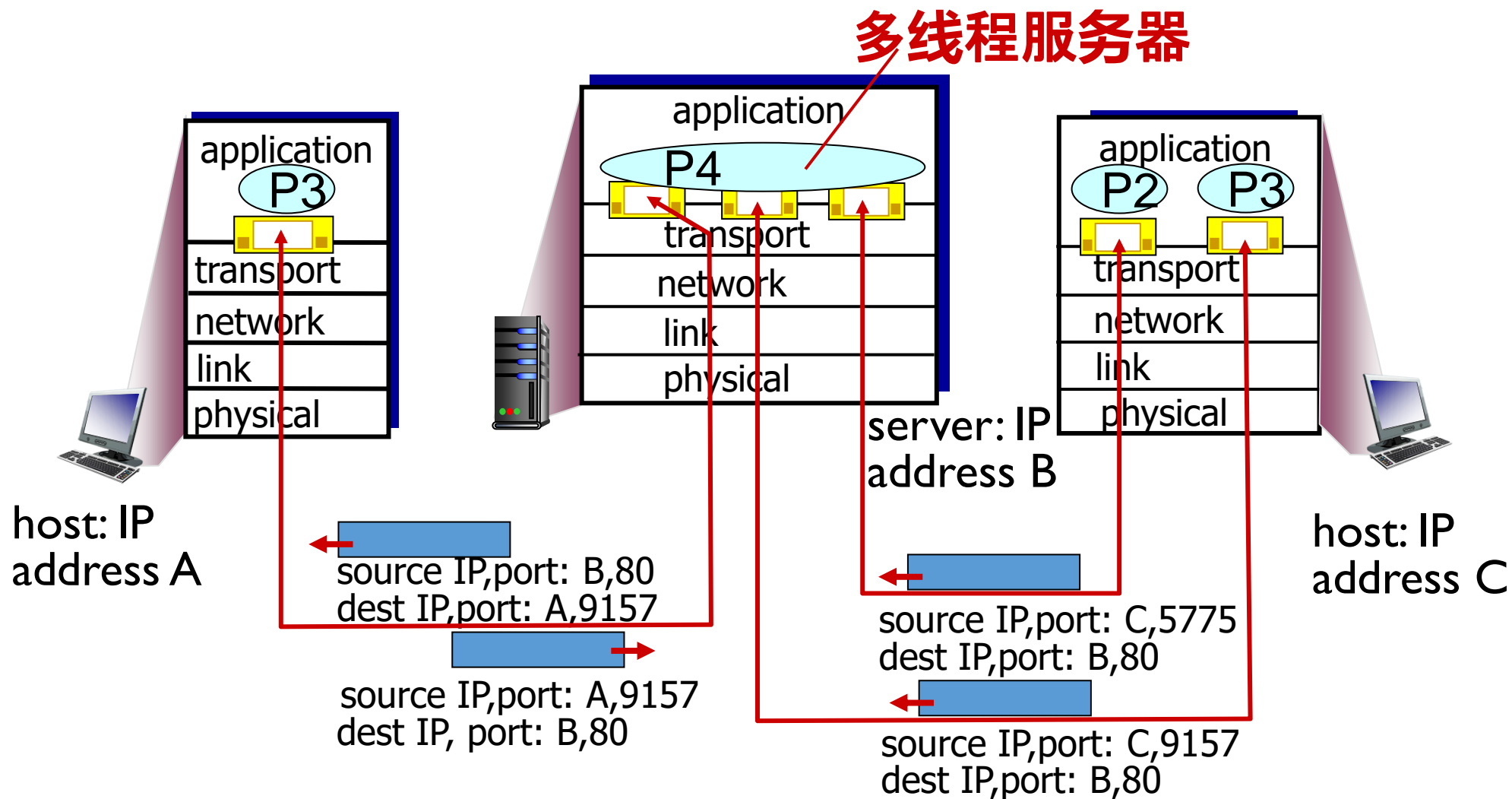
每个套接字用4部分来表示
以Web服务器为例: 对每个连接的客户都有不同的套接字
非持久 HTTP 将对每个请求有一个不同的套接字

面向连接的多路分解例子



三个被送至 IP address: B, dest port: 80 的报文, 被多路分解到三个不同的套接字

面向连接的多路分解例子



03 无连接传输: UDP



UDP: 用户数据报协议 [RFC 768]

- “无修饰” “不加渲染的” 因特网传输层协议
- “尽最大努力” 服务
- 数据段可能:
 - 丢失
 - 会传递失序的报文到应用程序
- 无连接:
 - 在UDP接收者发送者之间没有握手
 - 每个UDP 数据段的处理独立于其他数据段

为什么有 UDP?

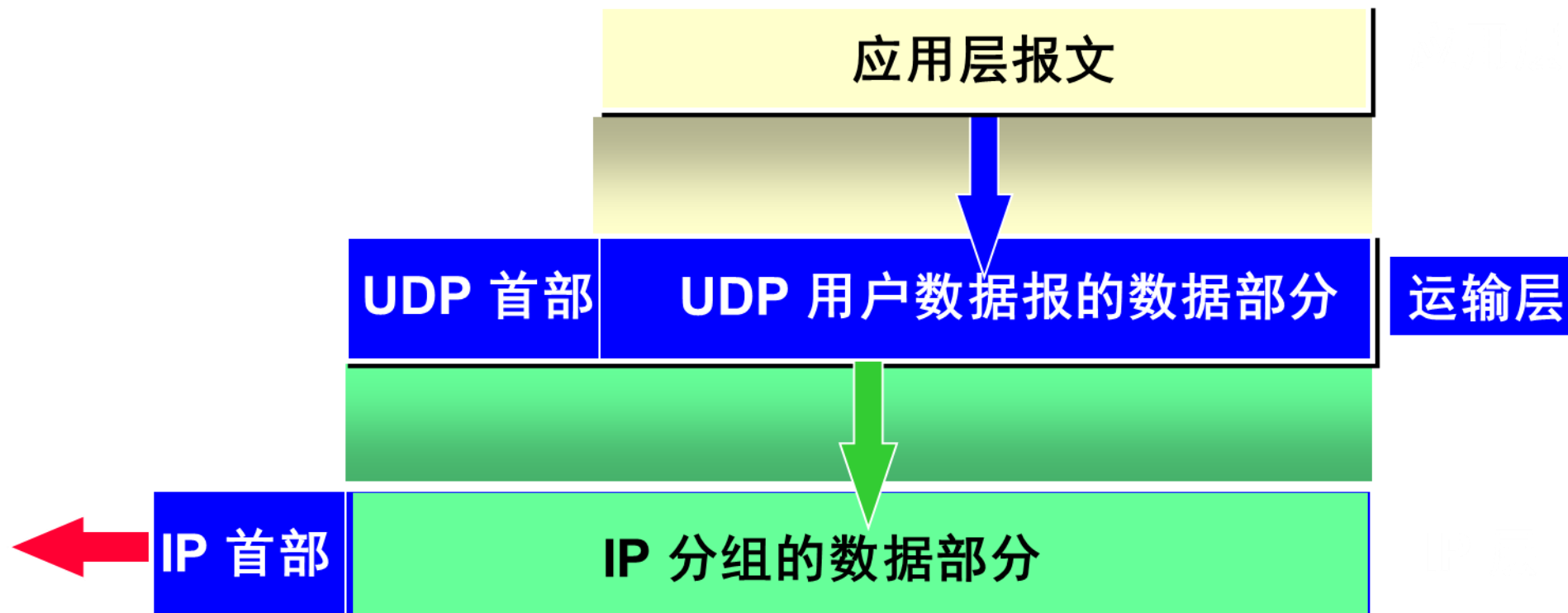
- 不需要建立连接 (减少延迟)
- 简单: 在发送者接受者之间不需要连接状态
- 很小的数据段首部
- 没有拥塞控制: UDP 能够用尽可能快的速度传递

UDP: 用户数据报协议 [RFC 768]

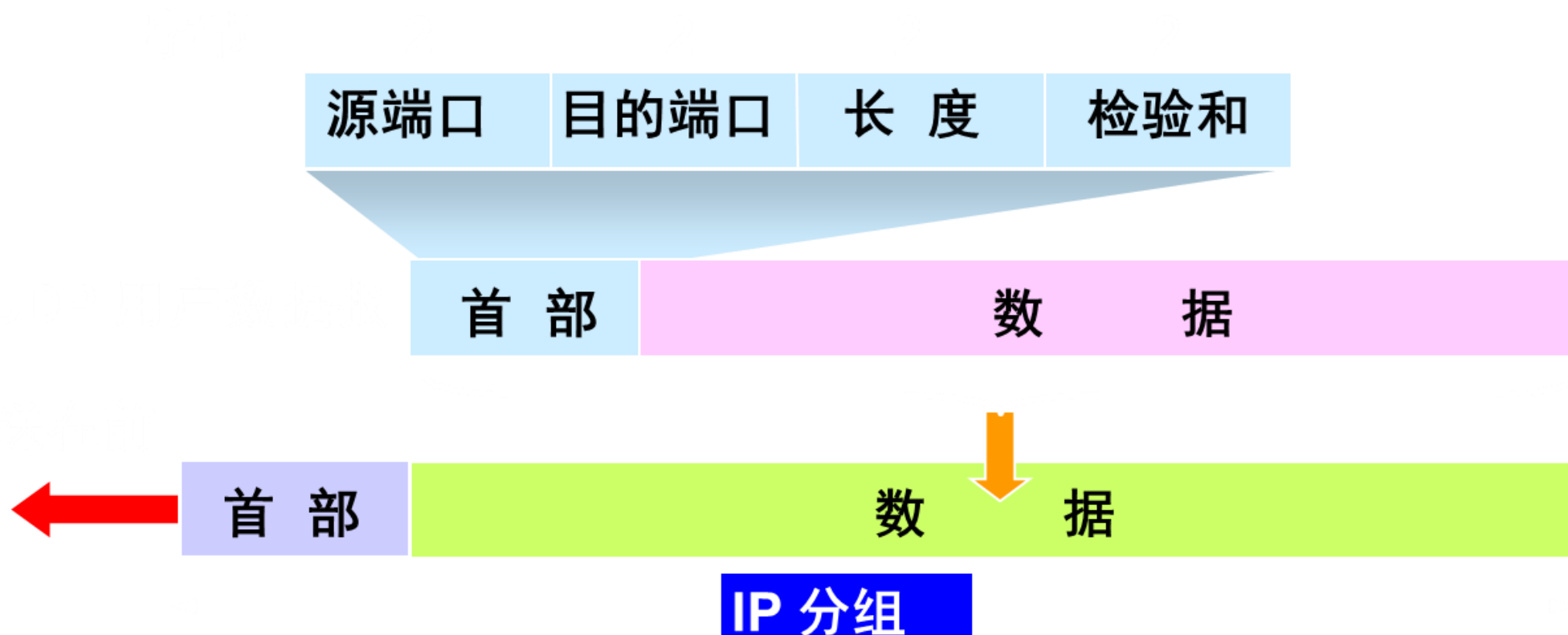
- UDP 只在 IP 的数据报服务之上增加了很少一点的功能，即端口的功能和差错检测的功能。
- 虽然 UDP 用户数据报只能提供不可靠的交付，但 UDP 在某些方面有其特殊的优点。
- UDP 是无连接的，即发送数据之前不需要建立连接。
- UDP 使用尽最大努力交付，即不保证可靠交付，同时也不使用拥塞控制。
- UDP 没有拥塞控制，很适合多媒体通信的要求。
- UDP 支持一对一、一对多、多对一和多对多的交互通信。

- UDP 的首部开销小，只有 8 个字节。
- UDP 是面向报文的。发送方 UDP 对应用程序交下来的报文，在添加首部后就向下交付 IP 层。UDP 对应用层交下来的报文，既不合并，也不拆分，而是保留这些报文的边界。
- 应用层交给 UDP 多长的报文，UDP 就照样发送，即一次发送一个报文。
- 接收方 UDP 对 IP 层交上来的 UDP 用户数据报，在去除首部后就原封不动地交付上层的应用进程，一次交付一个完整的报文。
- 应用程序必须选择合适大小的报文。

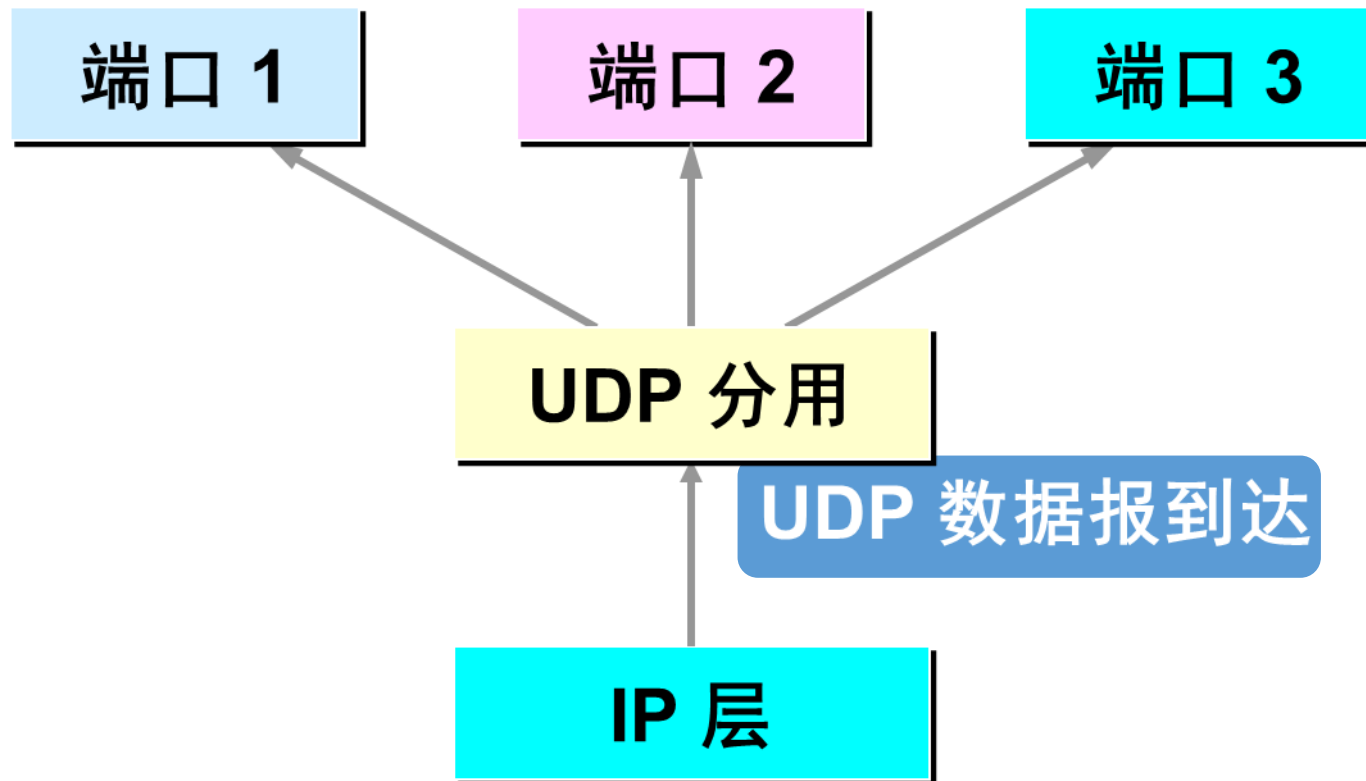
UDP 是面向报文的



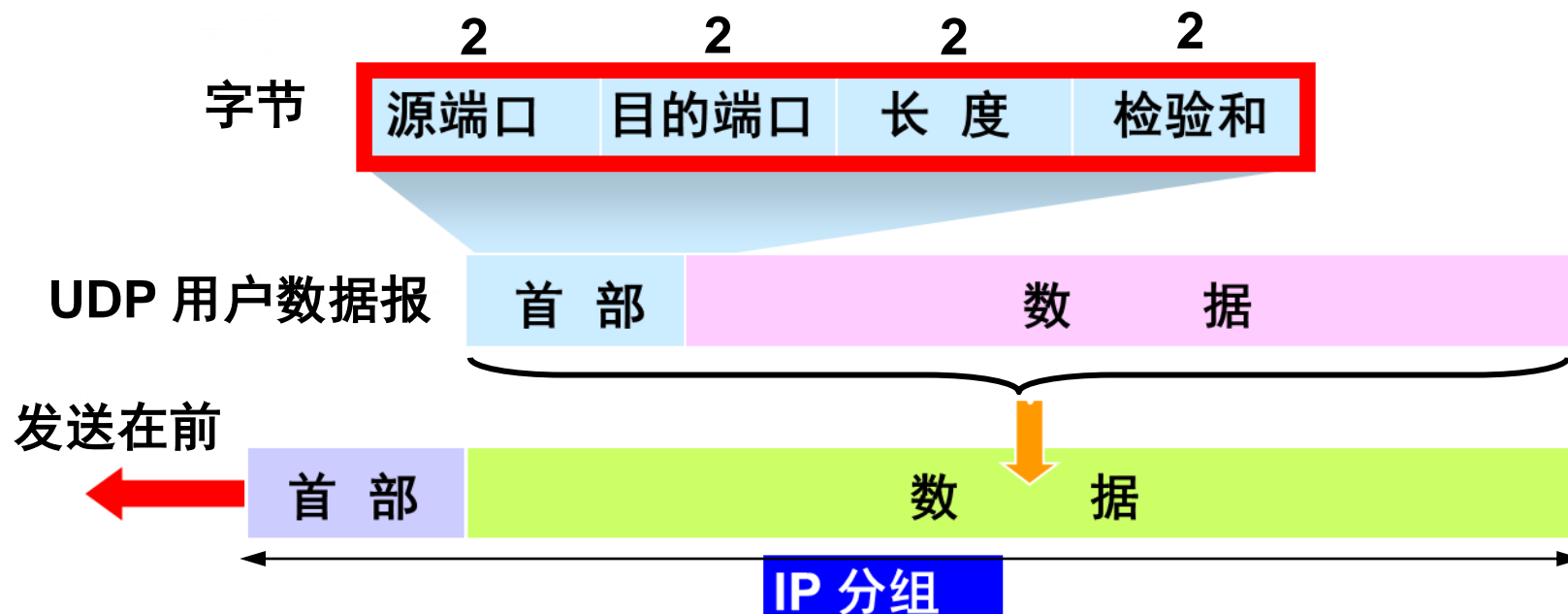
UDP 的首部格式



UDP 基于端口的多路分解

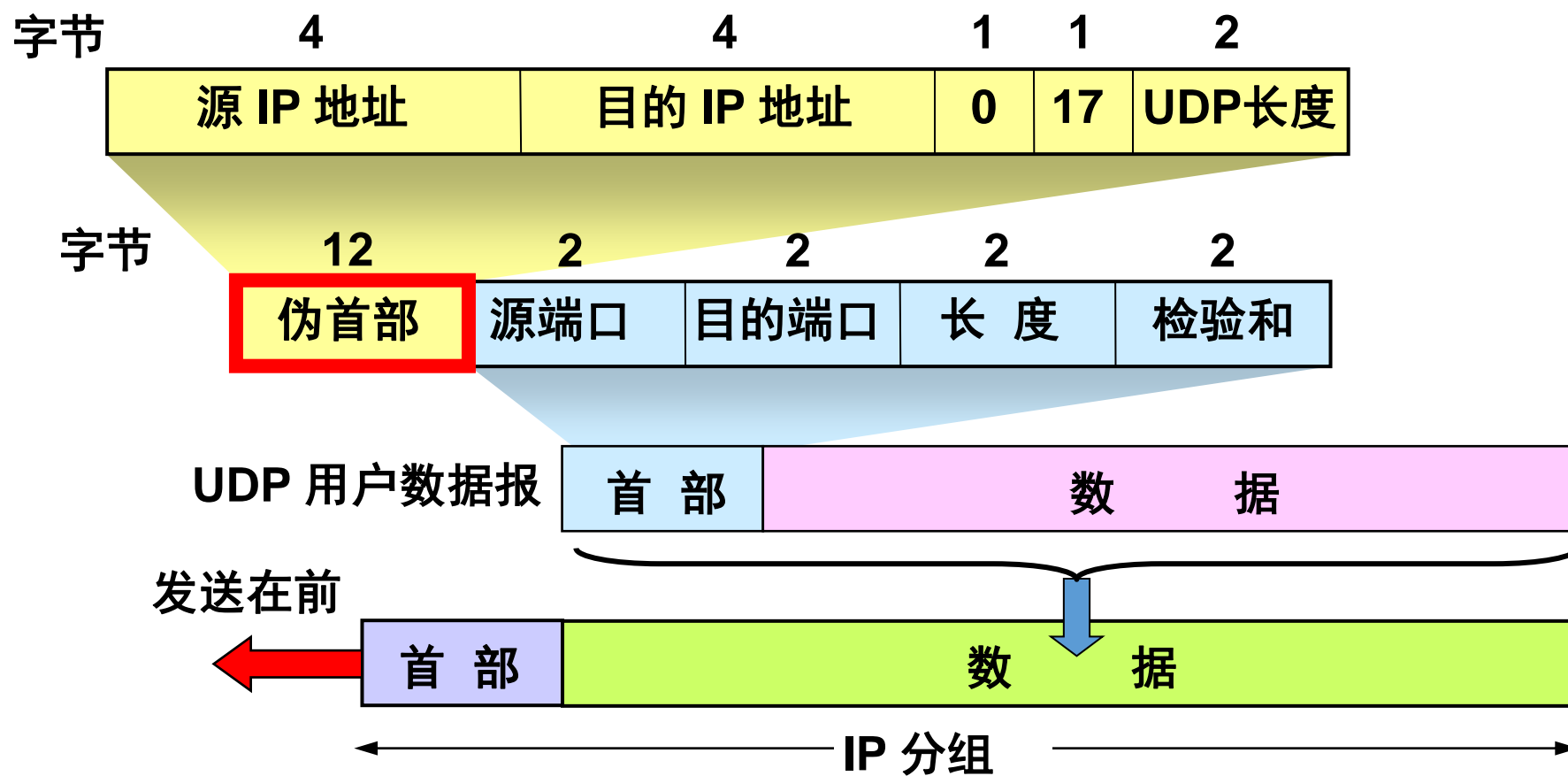


用户数据报 UDP 有两个字段：数据字段和首部字段。首部字段有 8 个字节，由 4 个字段组成，每个字段都是两个字节。长度是首部和数据的总长度



注：长度包括UDP首部，故最小值为8

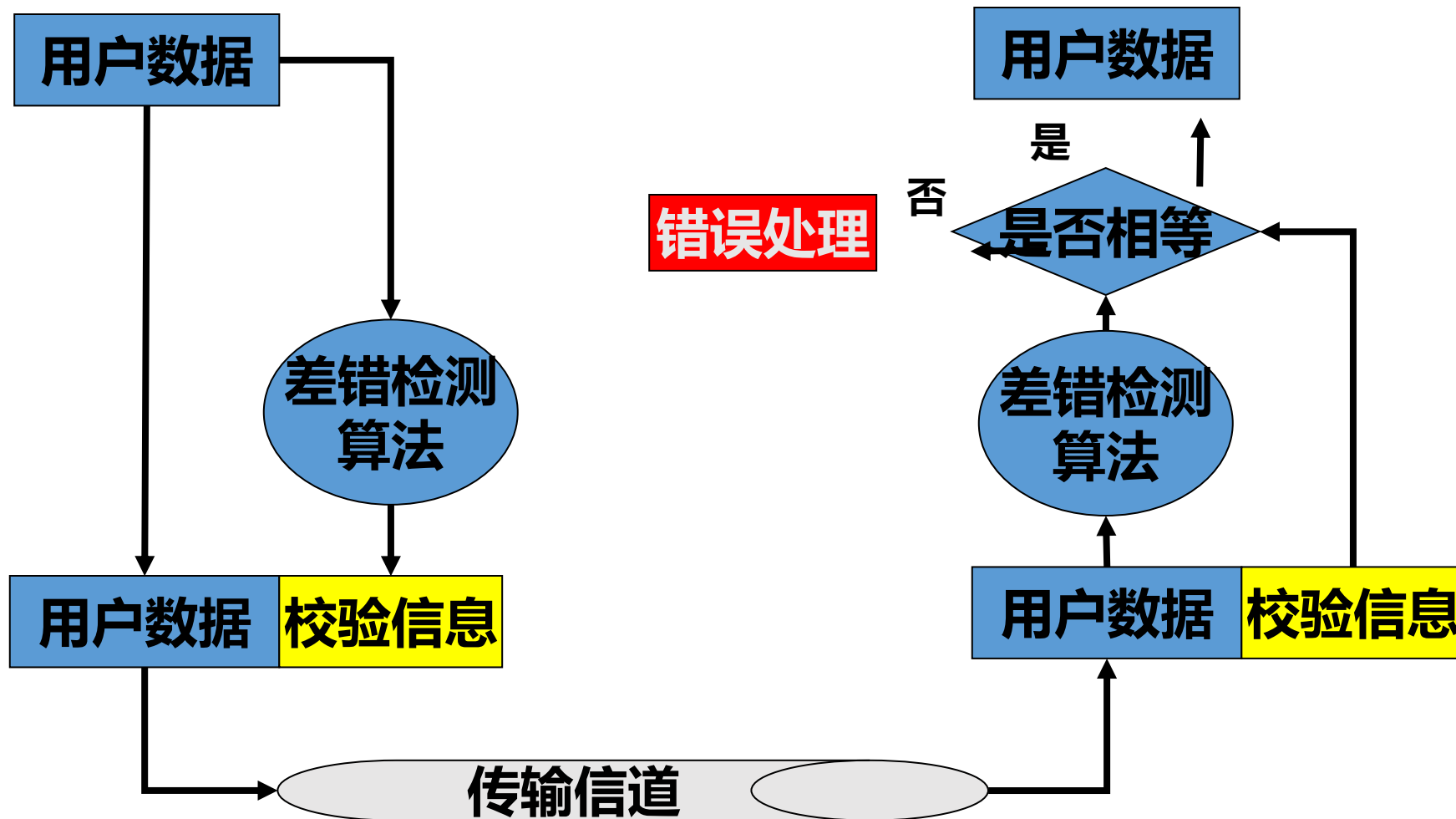
在计算检验和时，临时把“伪首部”和 UDP 用户数据报连接在一起。伪首部仅仅是为了计算检验和。



注：伪首部中的UDP长度，和首部中的长度一致

UDP 校验和与差错检测

错误检测不是
100%可靠!
协议有可能漏掉
一些错误, 但很
少;
大的校验信息
域能提供更好的
检错能力。



UDP 校验和

目标: 对传输的数据进行差错检测

发送方:

将数据段看成16bit的整数序列

校验和: 数据段内容相加 (1的补码和)

发送者将校验和值放入UDP的校验和域

接收方:

计算接收到数据段的校验和

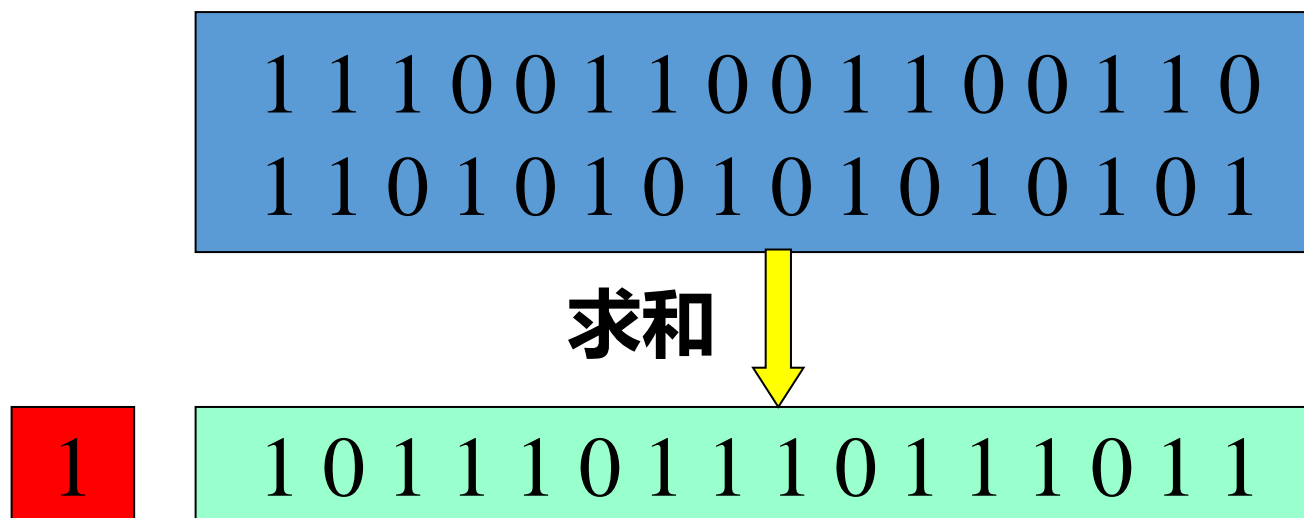
检查 计算的校验和是否等于校验和域中的值:

NO – 检测到错误

YES – 没有检测到错误

但是可能是错误的

Internet 校验和例子



求和时产生的进位必须回卷加到结果上

累加和

1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0

变反

校验和

0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1

最后的累加和必须按位变反才是校验和

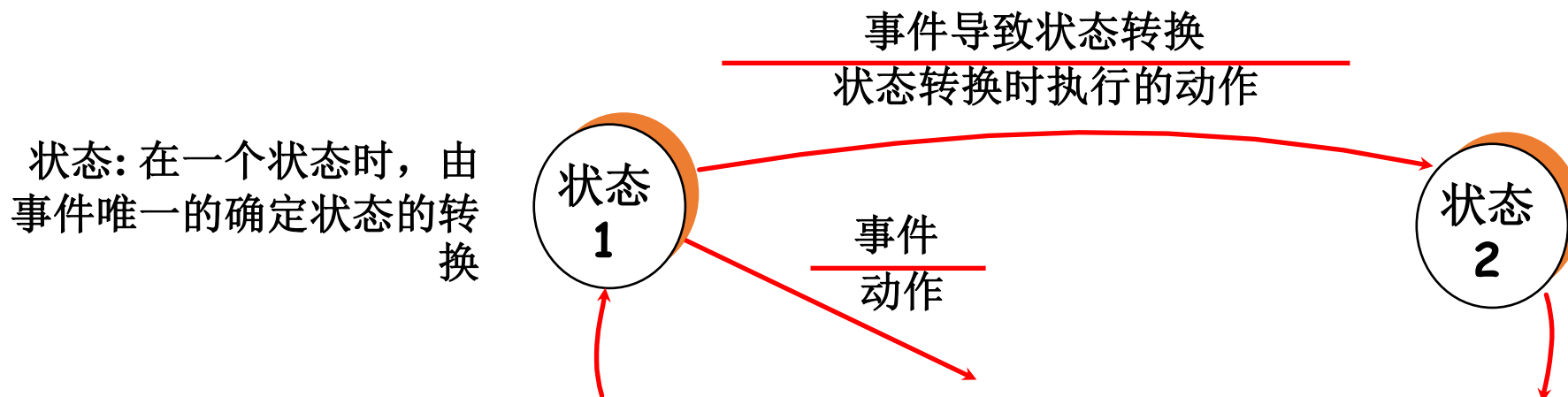
04 可靠数据传输原理



可靠数据传输

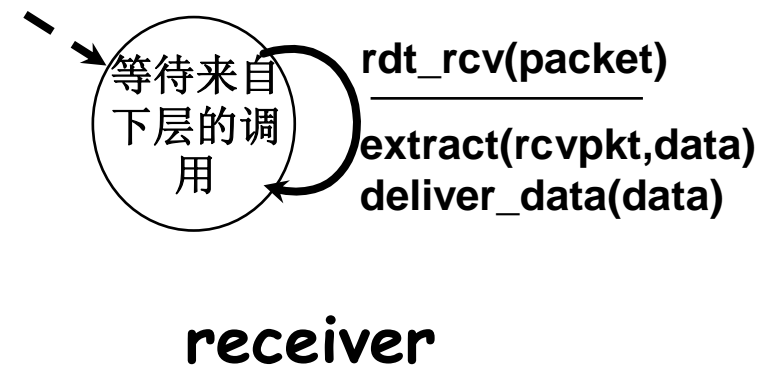
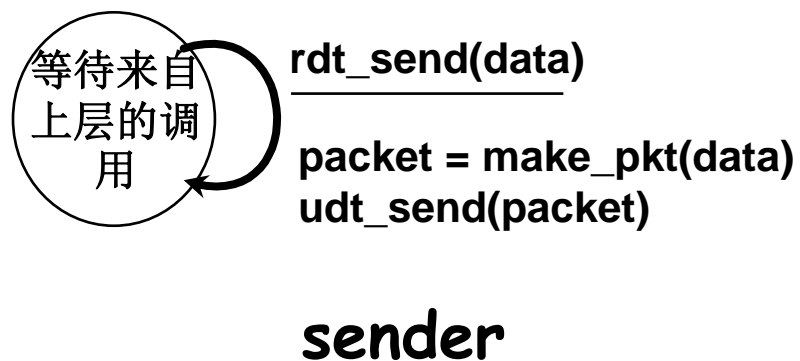
我们将

- 逐步开发发送方和接收方的可靠数据传输协议 (rdt)
- 仅考虑单向数据传输，但控制信息将双向流动!
- 用有限状态机 (FSM) 来标示发送方和接收方



Rdt1.0: 完全可靠信道上的可靠数据传输

- 在完美可靠的信道上
 - 没有bit错误
 - 没有分组丢失
- 发送方，接收方分离的 FSMs :
 - 发送方发送数据到下层信道
 - 接收方从下层信道接收数据



Rdt2.0: 具有bit错误的信道

下层信道可能让传输分组中的bit受损

- 校验和将检测到bit错误

问题: 如何从错误中恢复

- 确认(ACKs): 接收方明确告诉发送方 分组接收正确
- 否认 (NAKs):接收方明确告诉发送方 分组接收出错
- 发送方收到NAK后重发这个分组

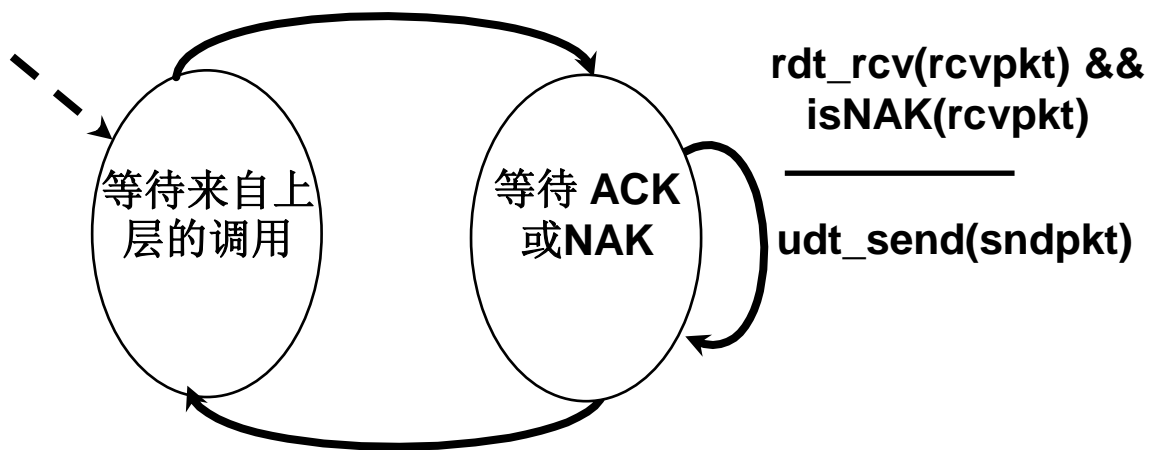
在 rdt2.0中的新机制 (在 rdt1.0中没有的):

- 差错检测
- 接收方反馈: 控制信息 (ACK,NAK) rcvr->sender

rdt2.0: FSM 规范

rdt_send(data)

snpkt = make_pkt(data, checksum)
udt_send(sndpkt)



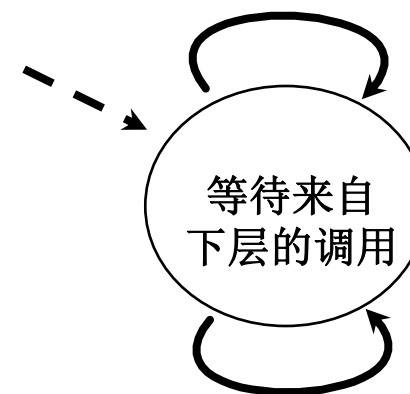
rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

sender

rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)

udt_send(NAK)

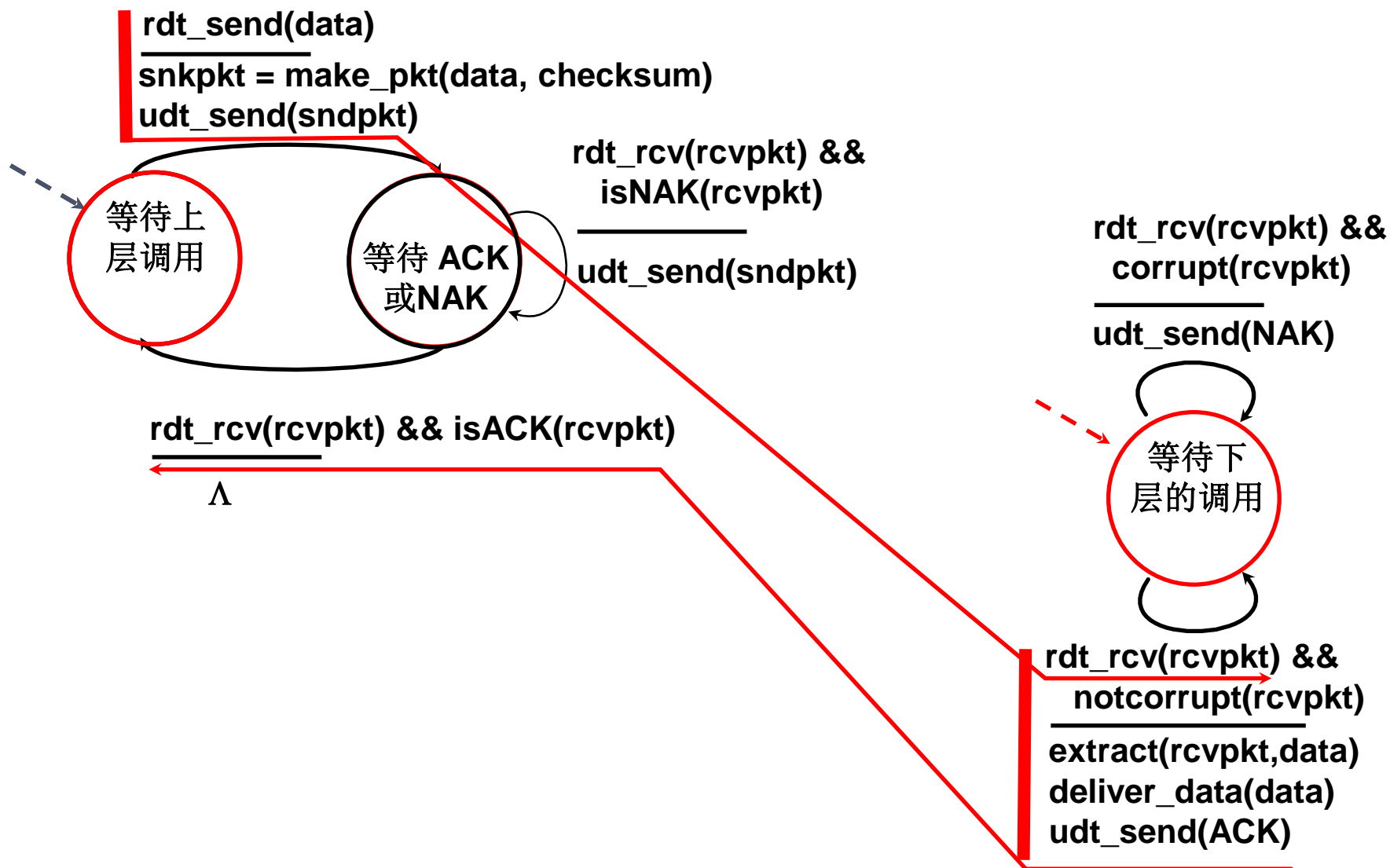


rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

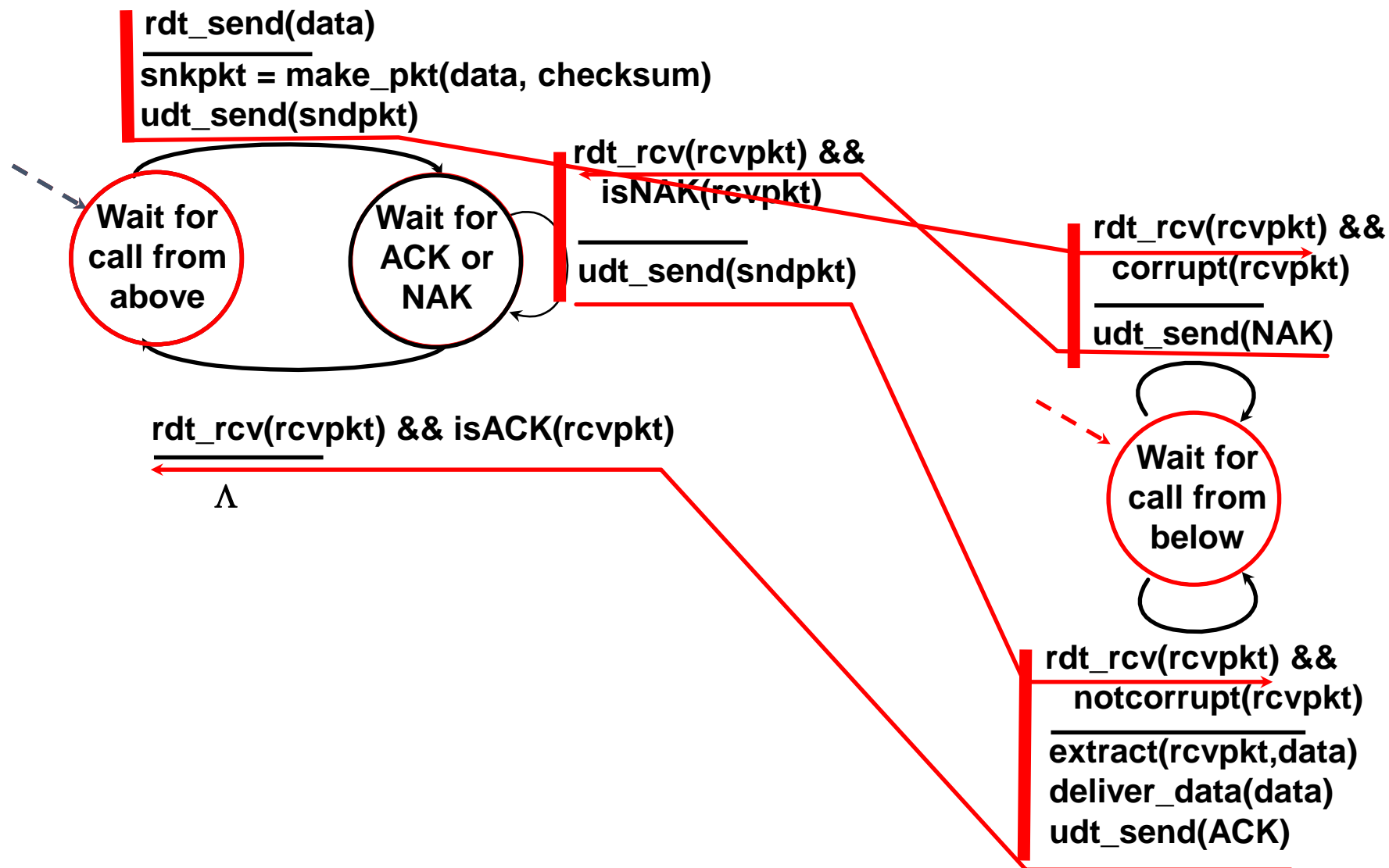
extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)

receiver

rdt2.0: 没有错误时的操作



rdt2.0: 错误场景



停 - 等协议

什么是停等协议?

发送方发送一个报文, 然后等待接受方的响应

stop and wait

rdt2.0 有一个致命缺陷!

如果ACK/NAK混淆了会发生什么?

发送方并不知道接收方发生了什么!

万能做法: **重发**

不能正确重发: 可能重复

处理重复:

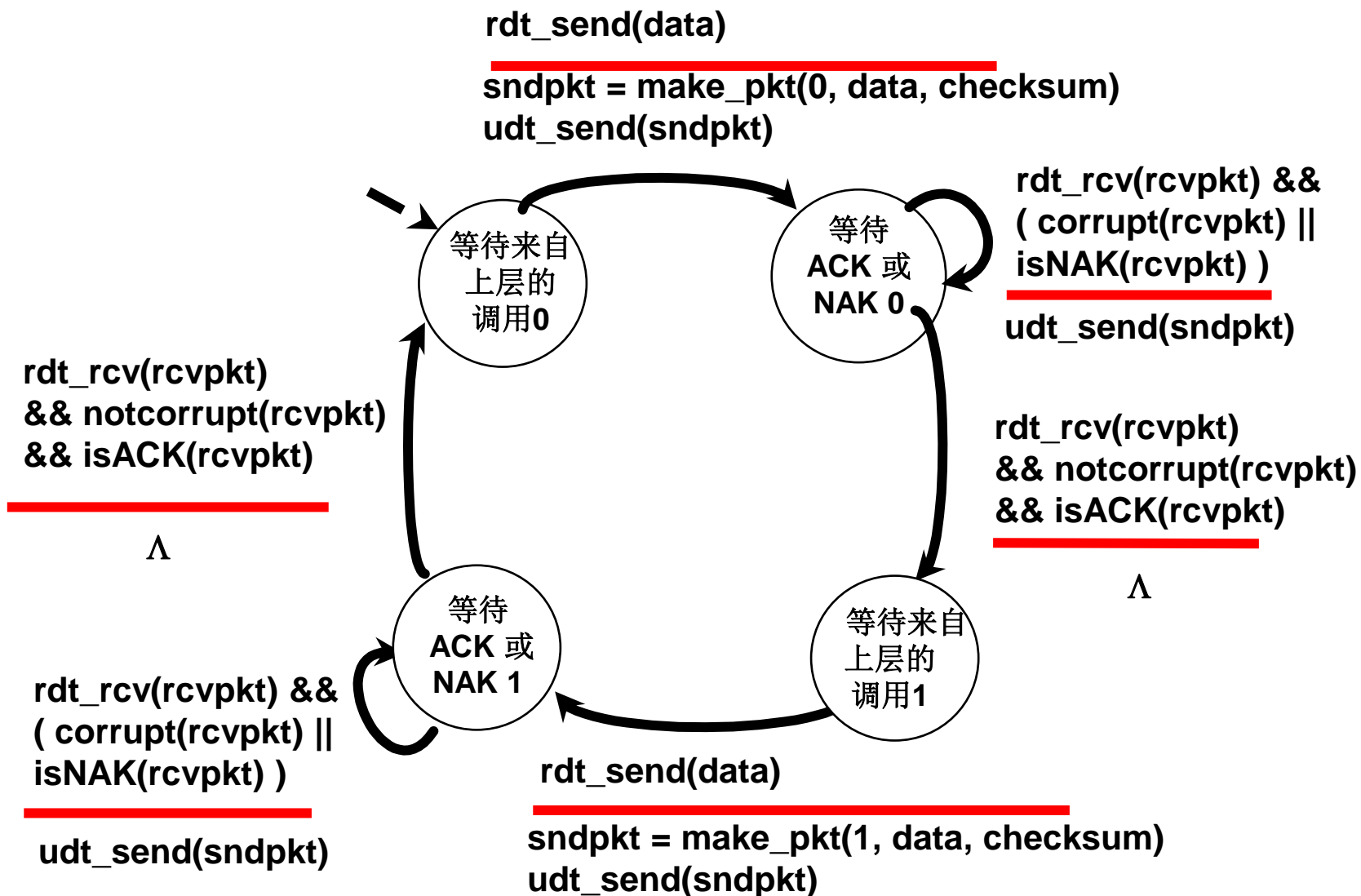
发送方给每个分组加一个序号

在 ACK/NAK 混淆时发送方重发当前分组

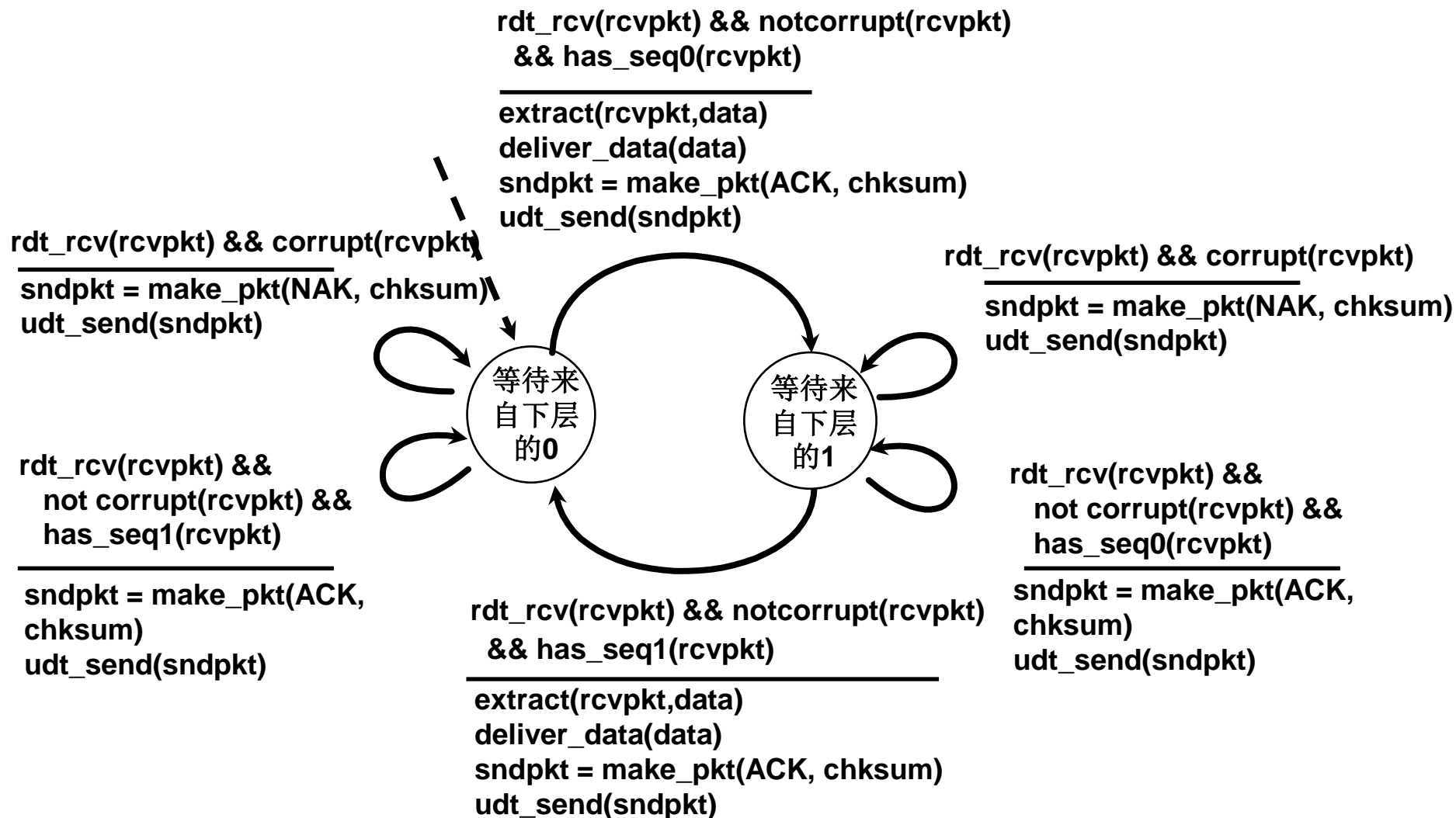
接收方丢弃重复的分组 (并不向上传递)

——停等协议数据包需要多少序号?

rdt2.1: 发送方处理混乱的 ACK/NAKs



rdt2.1: 接收方处理混乱的 ACK/NAKs



rdt2.1: 讨论

发送方:

- 序号 加到分组上
- 两个序号 (0,1) 就可以满足
- 必须检查是否收到混淆的 ACK/NAK
- 状态加倍

状态必须记住当前的分组是1号还是0号

接收方:

- 必须检查是否接收到重复的分组
状态指示0或者1: 是否希望的分组序号
- 注意:接收方并不知道它的上一个
ACK/NAK 是否被发送方正确收到

rdt2.2: 一个不要NAK的协议

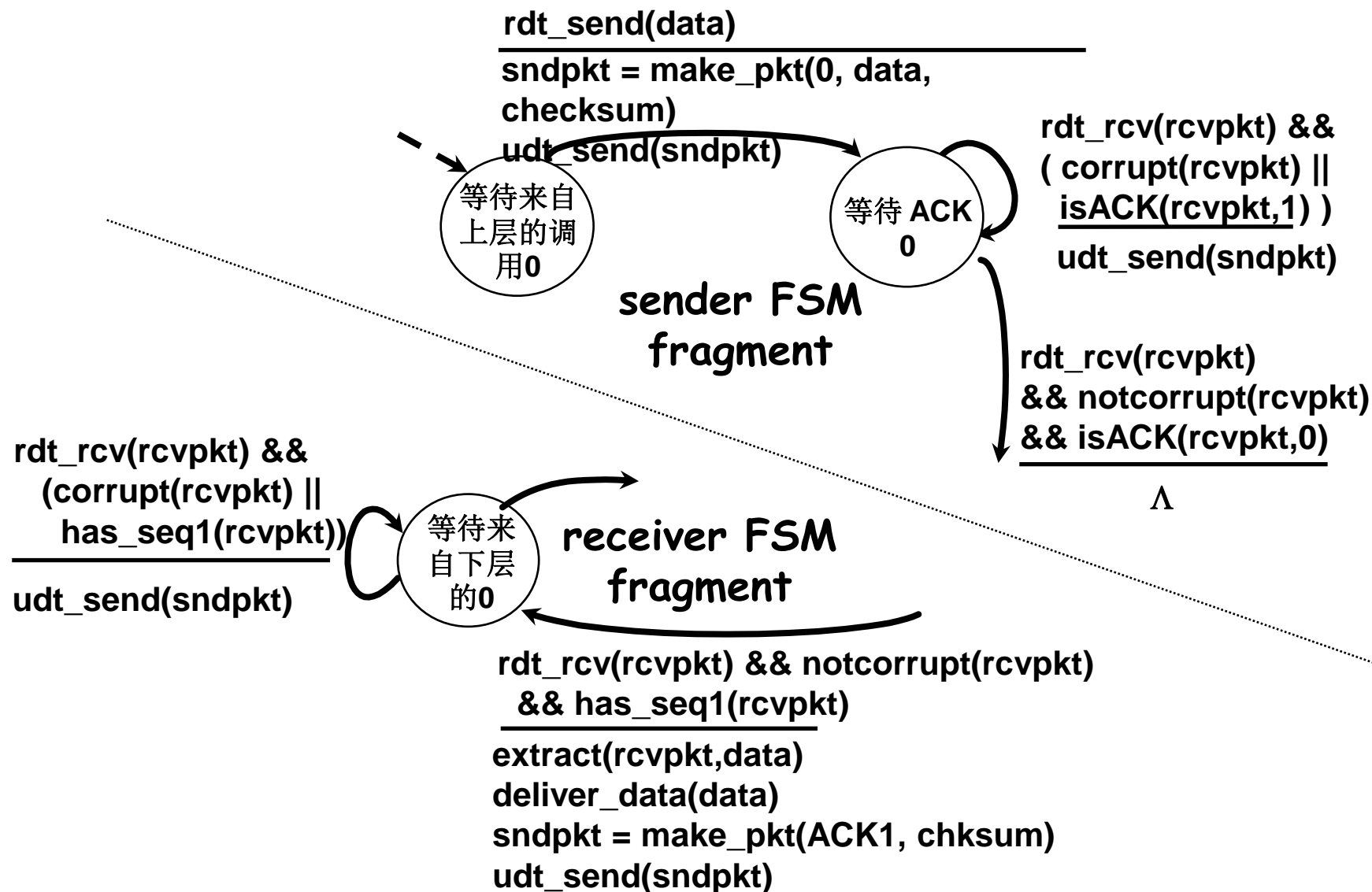
同 rdt2.1一样的功能, 只用 ACKs

不用 NAK, 如果上个报文接收正确接收方发送 ACK

接收方必须明确包含被确认的报文的序号

发送方收到重复 ACK 将导致和 NAK一样的处理: 重发当前报文

rdt2.2: 发送方,接收方片断



rdt3.0: 具有出错和丢失的信道

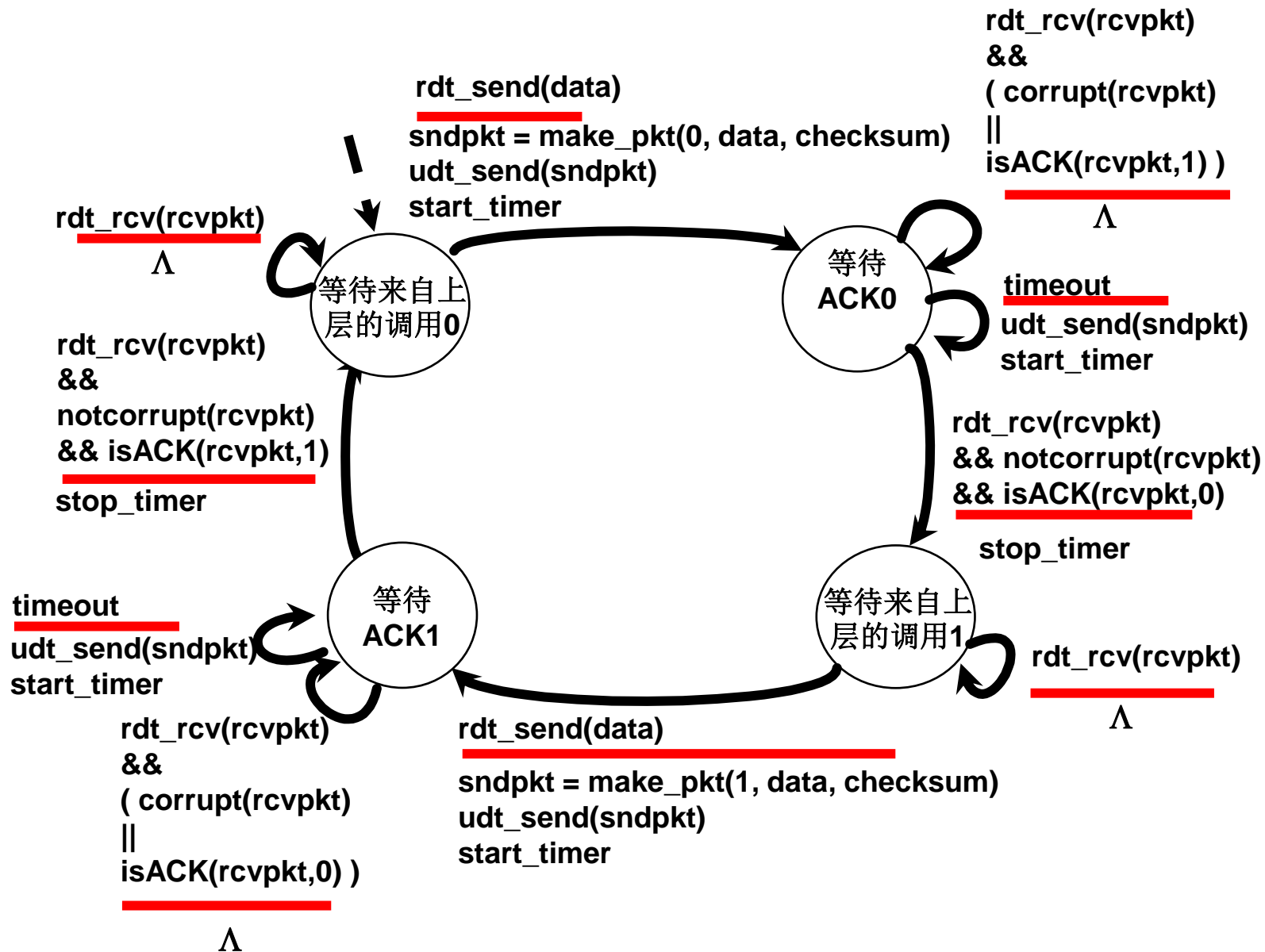
新假设: 下层信道还要丢失报文 (数据或者 ACKs)

校验和, 序号, 确认, 重发将会有帮助, 但是不够

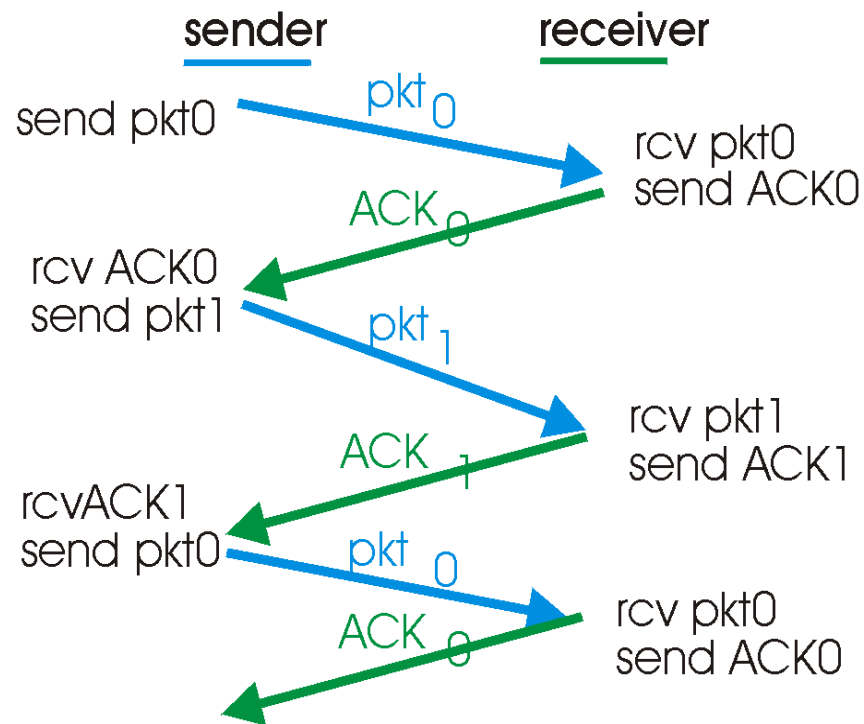
方法: 发送者等待“合理的”确认时间

- 如果在这个时间内没有收到确认就重发
- 如果报文 (或者确认) 只是延迟 (没有丢失):
重发将导致重复, 但是使用序号已经处理了这个问题
接受方必须指定被确认的报文序号
- 要求倒计时定时器
- 只有在定时器超时时才触发重发

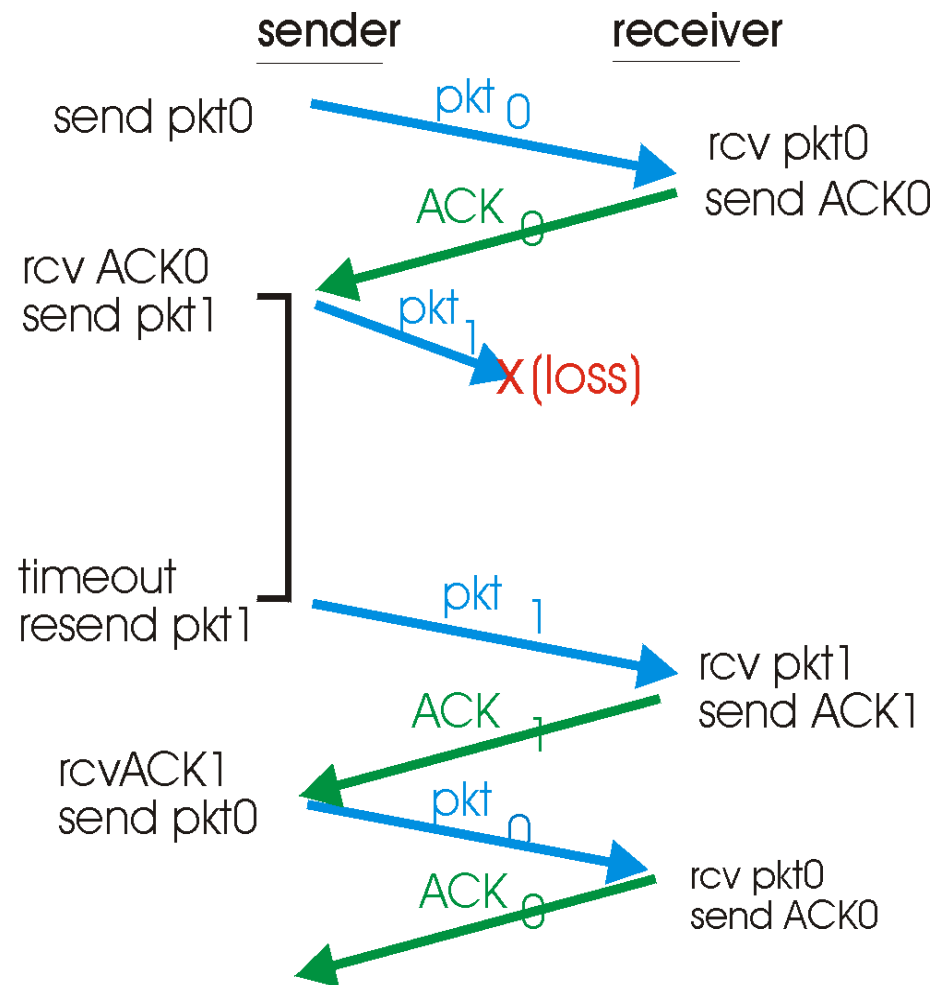
rdt3.0 发送方



rdt3.0 操作

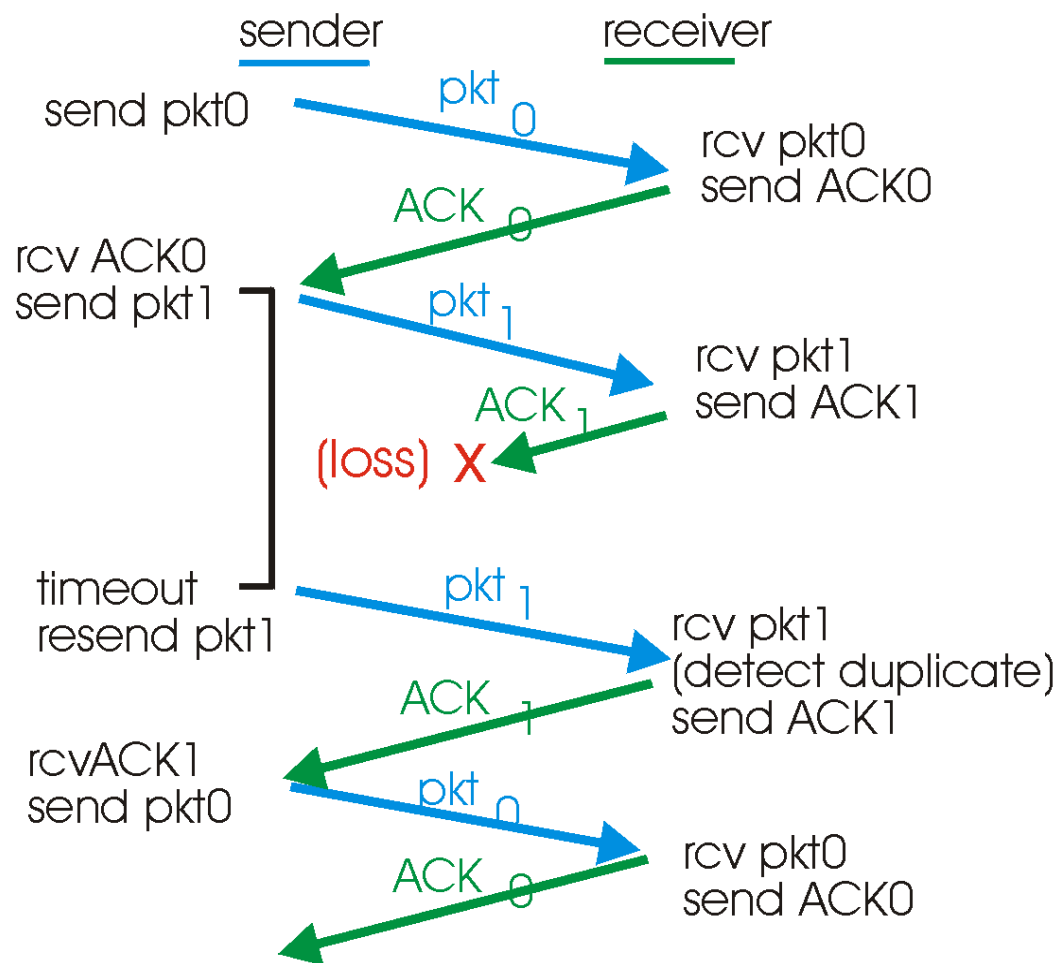


(a) operation with no loss

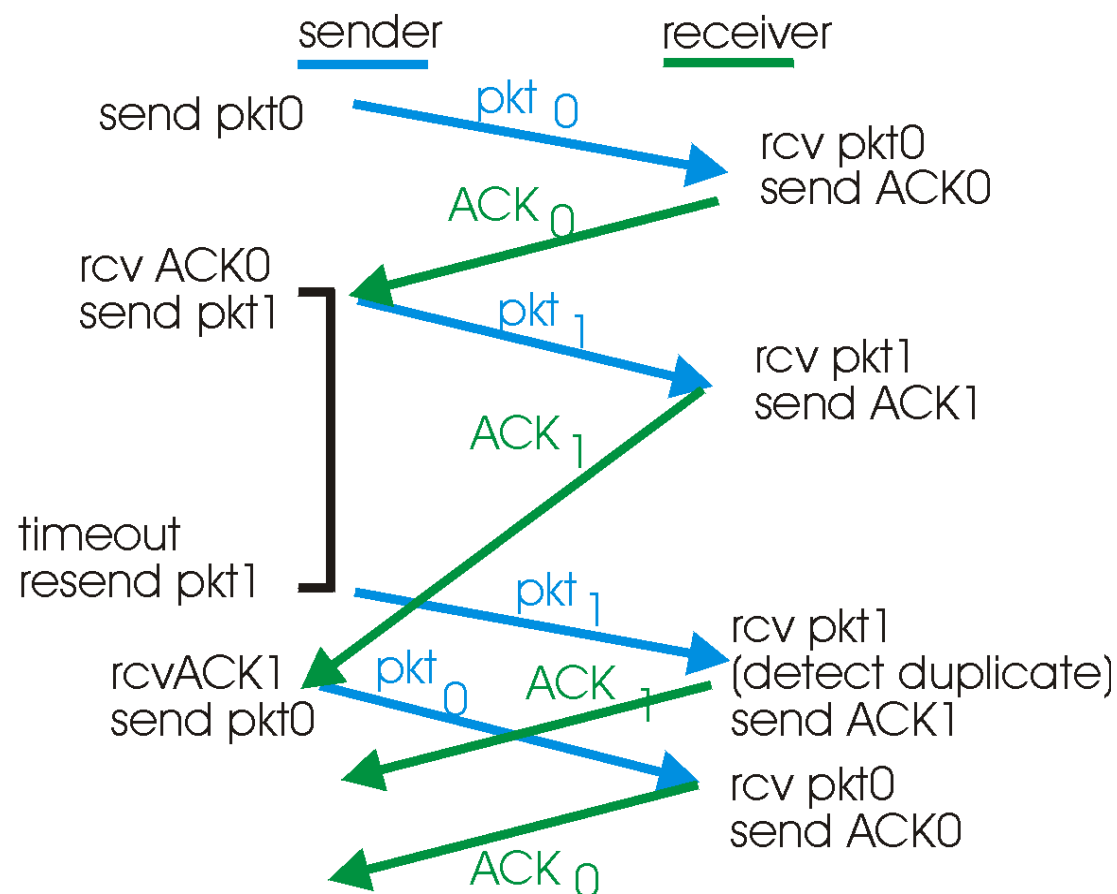


(b) lost packet

rdt3.0 操作



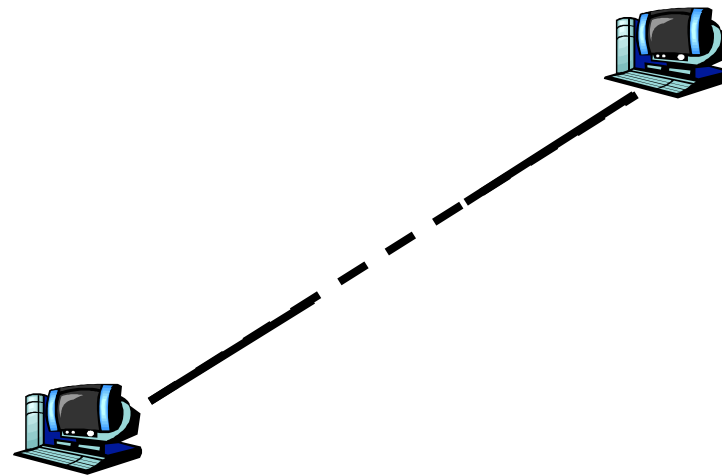
(c) lost ACK



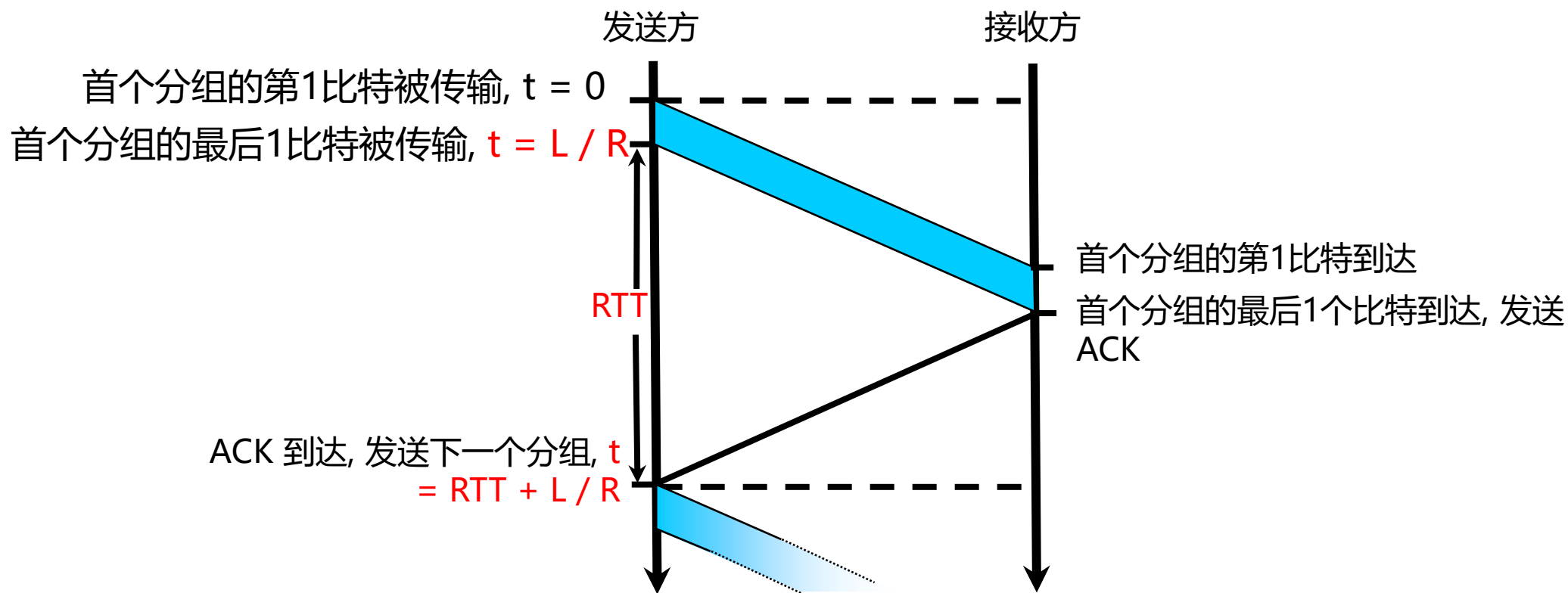
(d) premature timeout

rdt3.0的性能

例如: 1 Gbps链路, 15 ms 端到端传输
延迟, 8000bit 报文, 计算网络利用率?



rdt3.0: 停等操作



rdt3.0的性能

$$T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^{**}9 \text{ b/sec}} = 8 \text{ microsec}$$

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

U_{sender} : 利用率 – 发送方忙于发送的时间部分

1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link

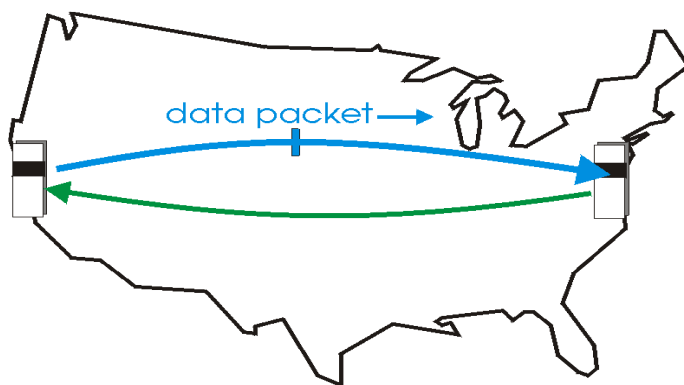
rdt3.0 能工作但是性能很差

网络协议限制了物理资源的使用!

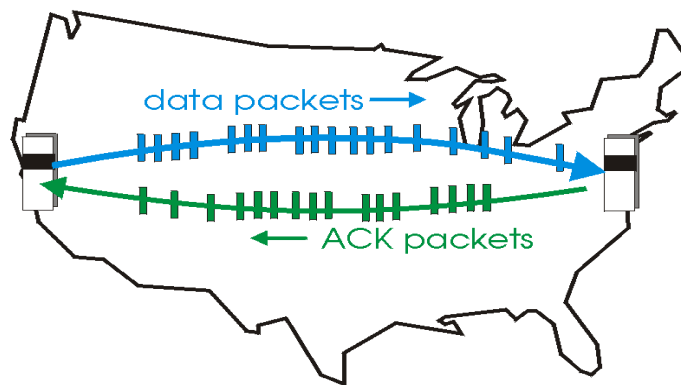
流水线技术

流水线: 发送方允许发送多个 “在路上的”, 还没有确认的报文

- 序号数目的范围必须增加
- 在发送方/接收方必须有缓冲区



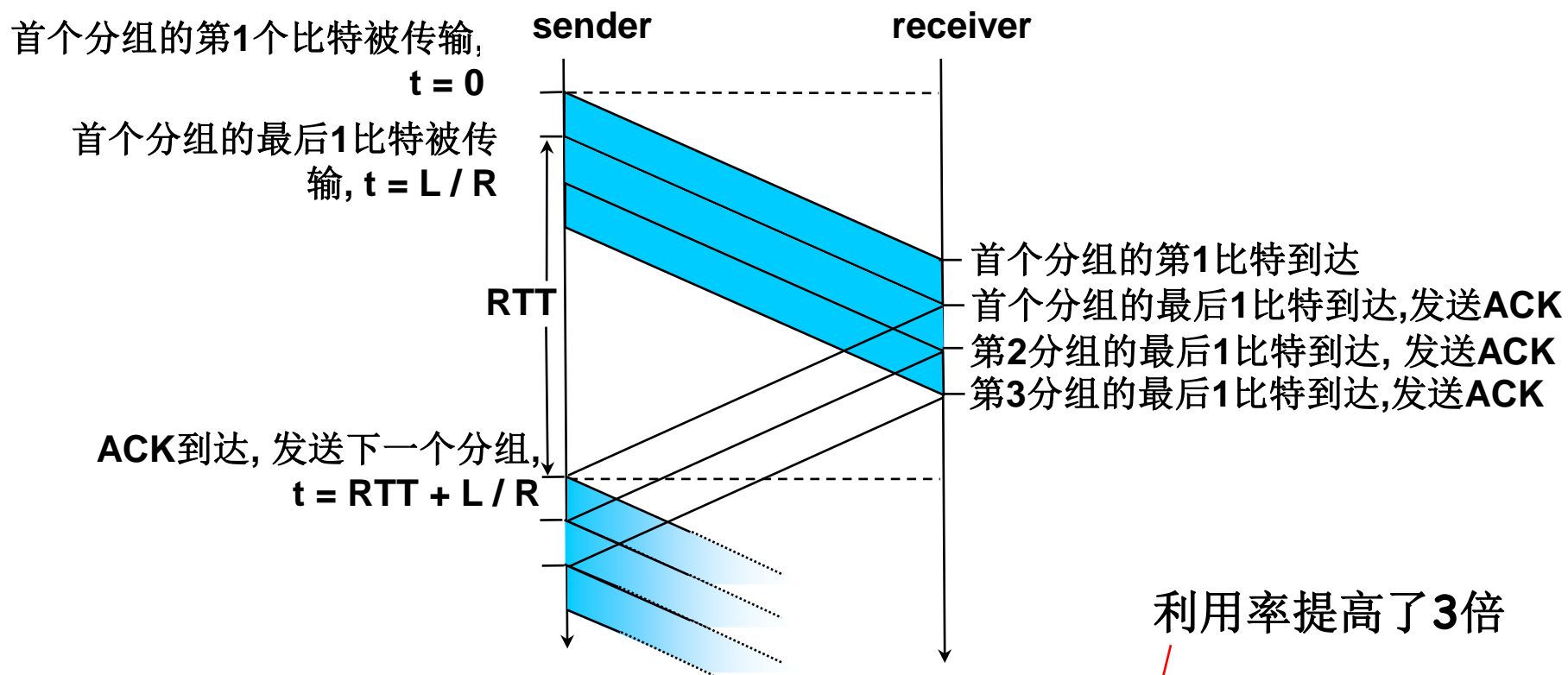
(a) a stop-and-wait protocol in operation



(b) a pipelined protocol in operation

流水线技术的两个通用形式: *go-Back-N*, 选择重传

流水线: 增加利用率



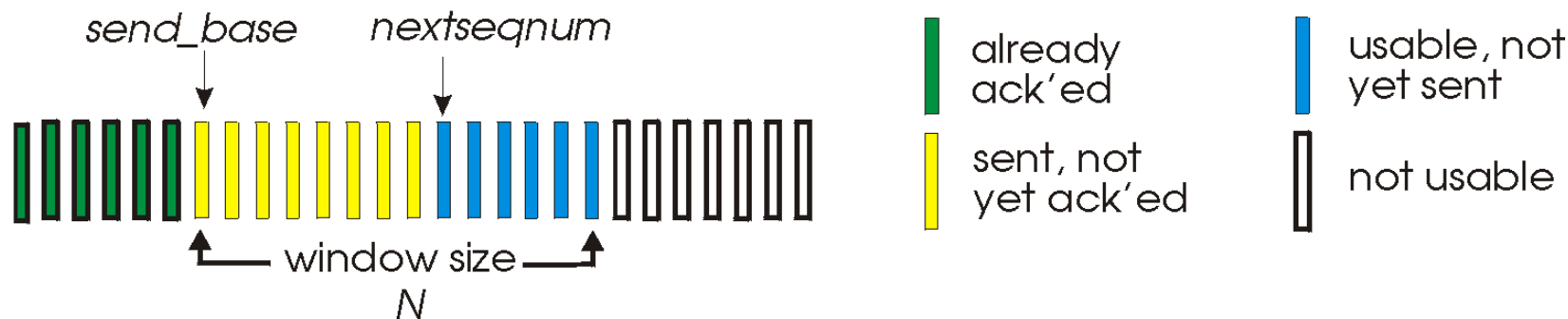
利用率提高了3倍

$$U_{\text{sender}} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Go-Back-N

发送方:

- 在分组头中规定一个k位的序号
- “窗口”, 允许的连续未确认的报文



ACK(n): 确认所有的报文直到 (包含) 序号n - “累积ACK”
对第一个发送未被确认的报文定时

超时(n): 重发窗口中的报文n及以上更高序号的报文(只有一个定时器记录最早的未被确认报文的发送时间)

GBN: 接收方

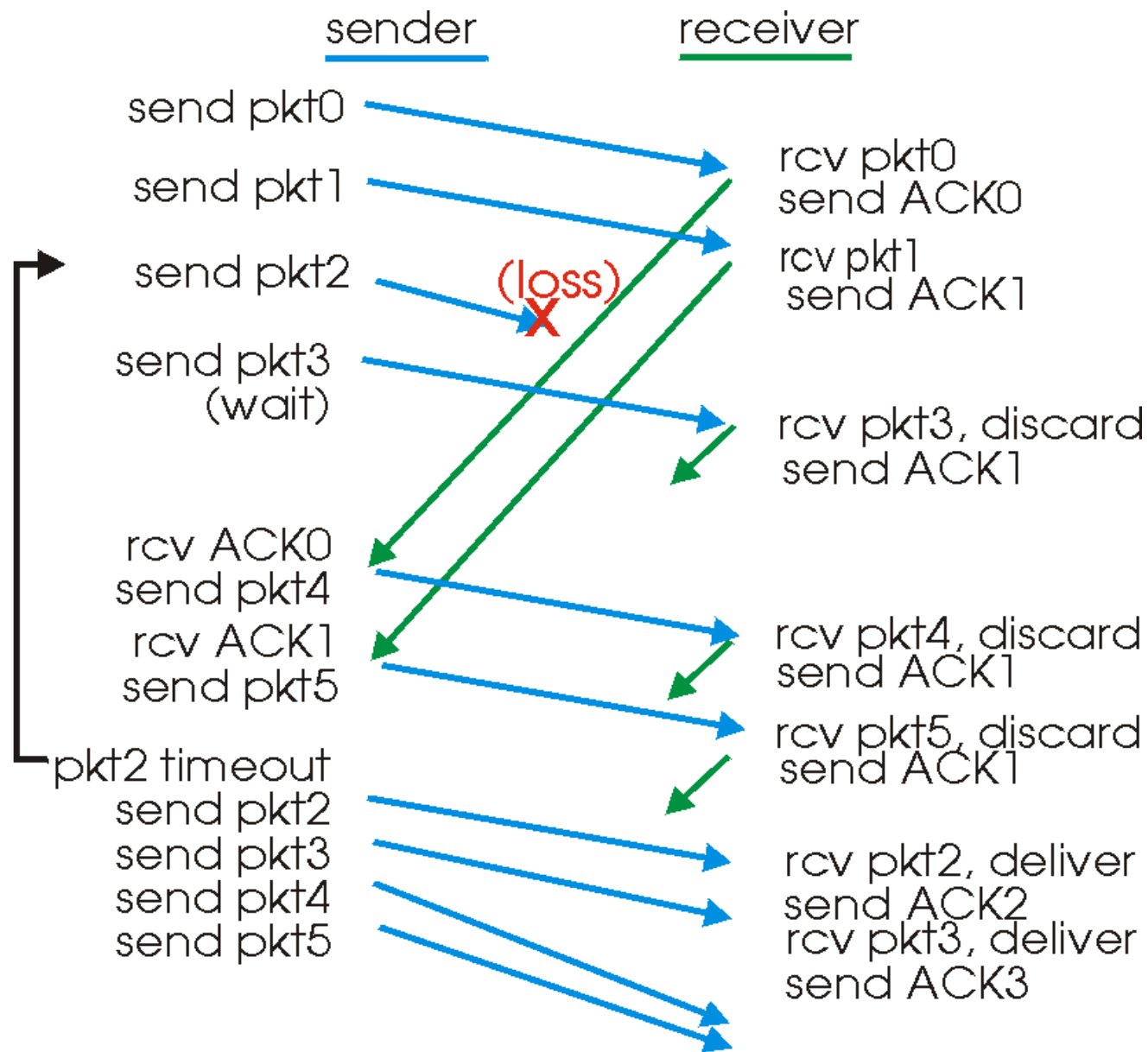
ACK-only: 总是为正确接收的最高序号的分组发送ACK。

- 可能生成重复的ACKs
- 只需要记住被期待接收的序号
expectedseqnum

接收到失序分组:

丢弃(不缓冲) -> 没有接收缓冲区!

重发最高序号分组的ACK



选择性重传(Selective Repeat, SR)

- 接收方分别确认已经收到的分组

必要时, 缓冲报文, 最后按序提交给上层

- 发送者只重发没有收到确认的分组

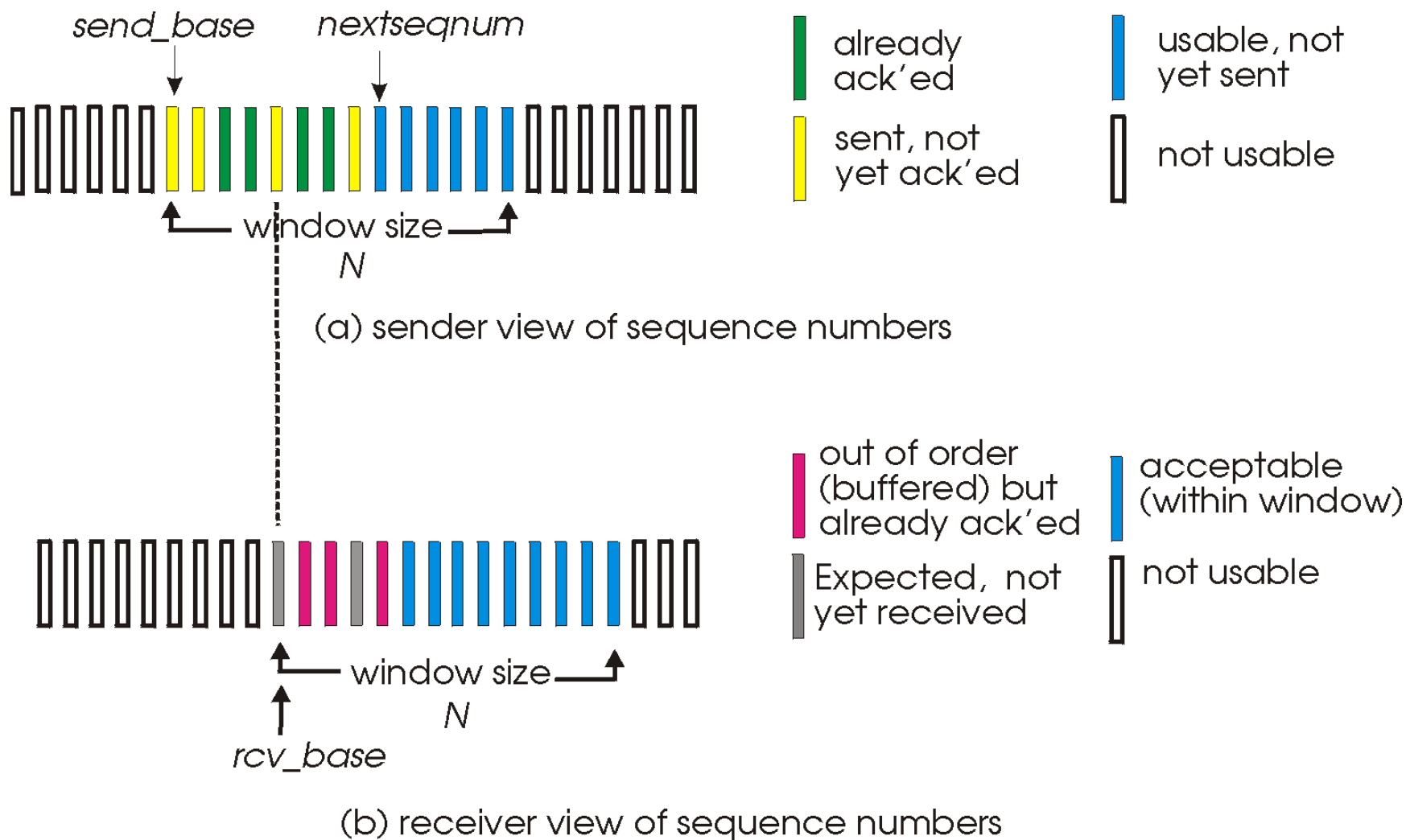
对每个没有确认的报文发送者都要启动一个定时器(每个未被确认的报文都有一个定时器)

- 发送窗口

N 个连续序号

限制被发送的未确认的分组数量

选择性重传: 发送者, 接收者窗口



选择性重传

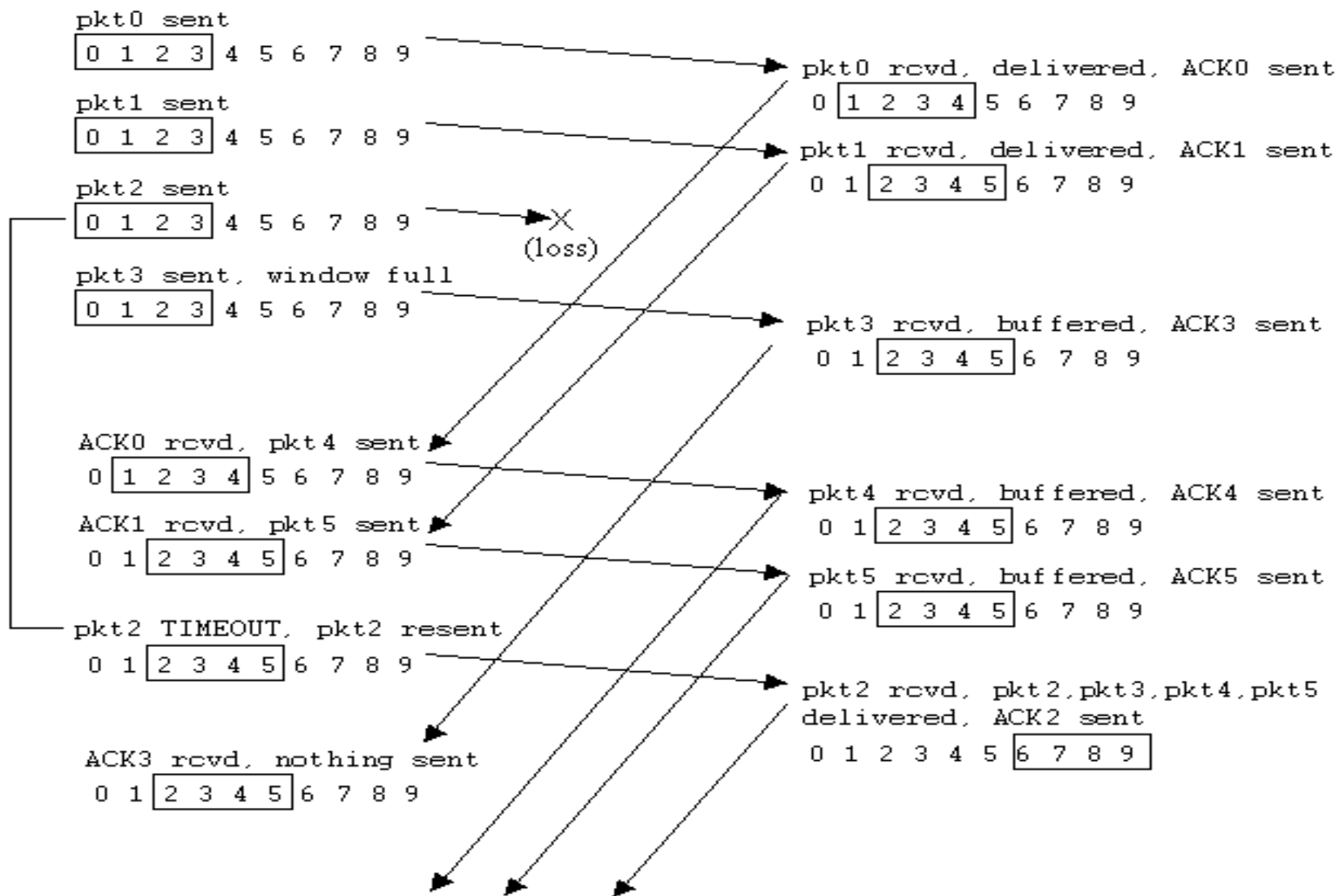
发送方

- 从上层收到数据：
如果下一个可用的序号在发送方窗口内，
则将数据打包并发送，启动定时器
- 超时(n):
重发分组n, 重启定时器
- 收到ACK(n)在[sendbase, sendbase+N-1]内:
标记分组n被接收
如果n是最小的未确认分组，则增加窗口基
序号到下一个未被确认的序号

接收方

- 分组n的序号在[rcvbase, rcvbase+N-1]内
发送ACK(n)
失序分组: 缓冲
有序分组: 交付上层 (包括已经缓冲的有序
分组), 提高窗口到下一个没有接收的分组
- 分组n在[rcvbase-N, rcvbase-1]内
发送ACK(n)
- 其他:
忽略

选择性重传的操作



选择性重传: 两难选择

例子:

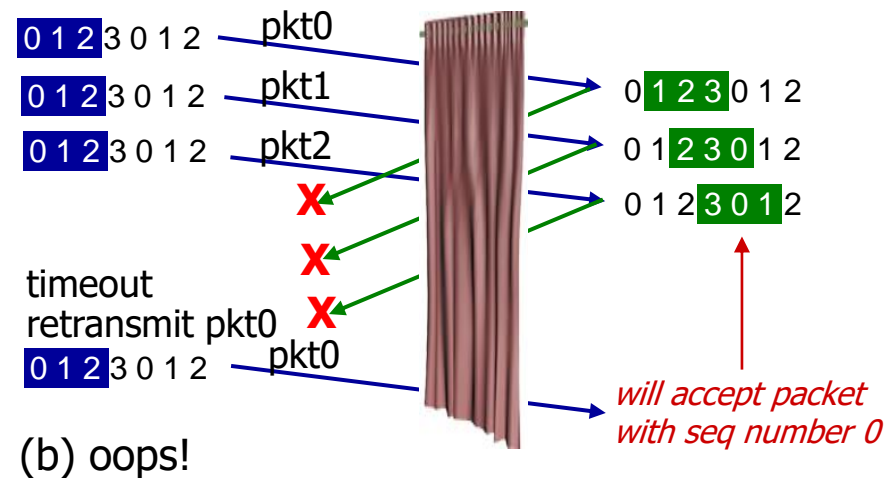
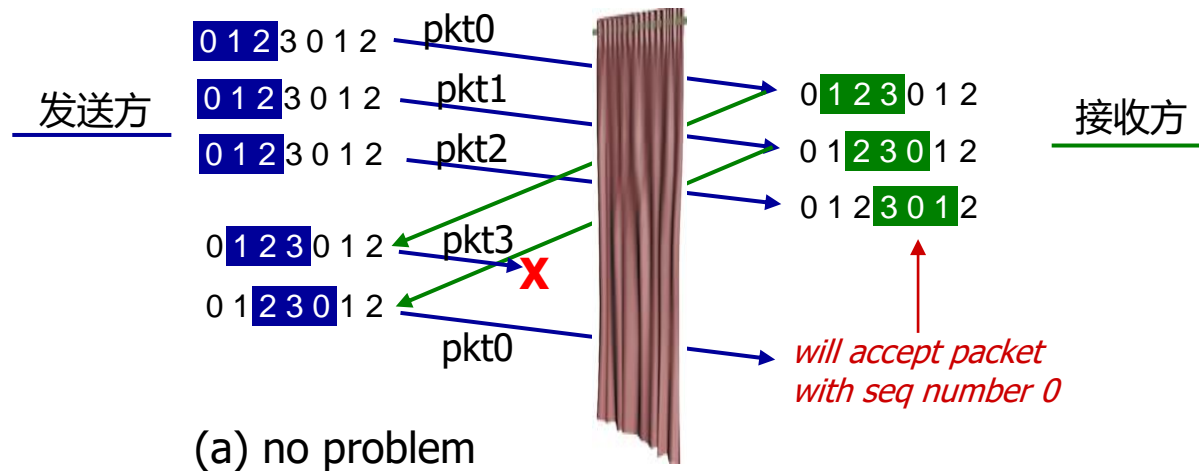
序号: 0, 1, 2, 3

window size=3

在两种情况下接收方没有感觉到差别!

Q: 窗口大小和序号大小有什么关系?

A: 窗口小于或等于序号空间大小的一半



接收方无法看到发送方，因此两种情况
对它来说是等同的，结果出了问题!

05 面向连接传输: TCP

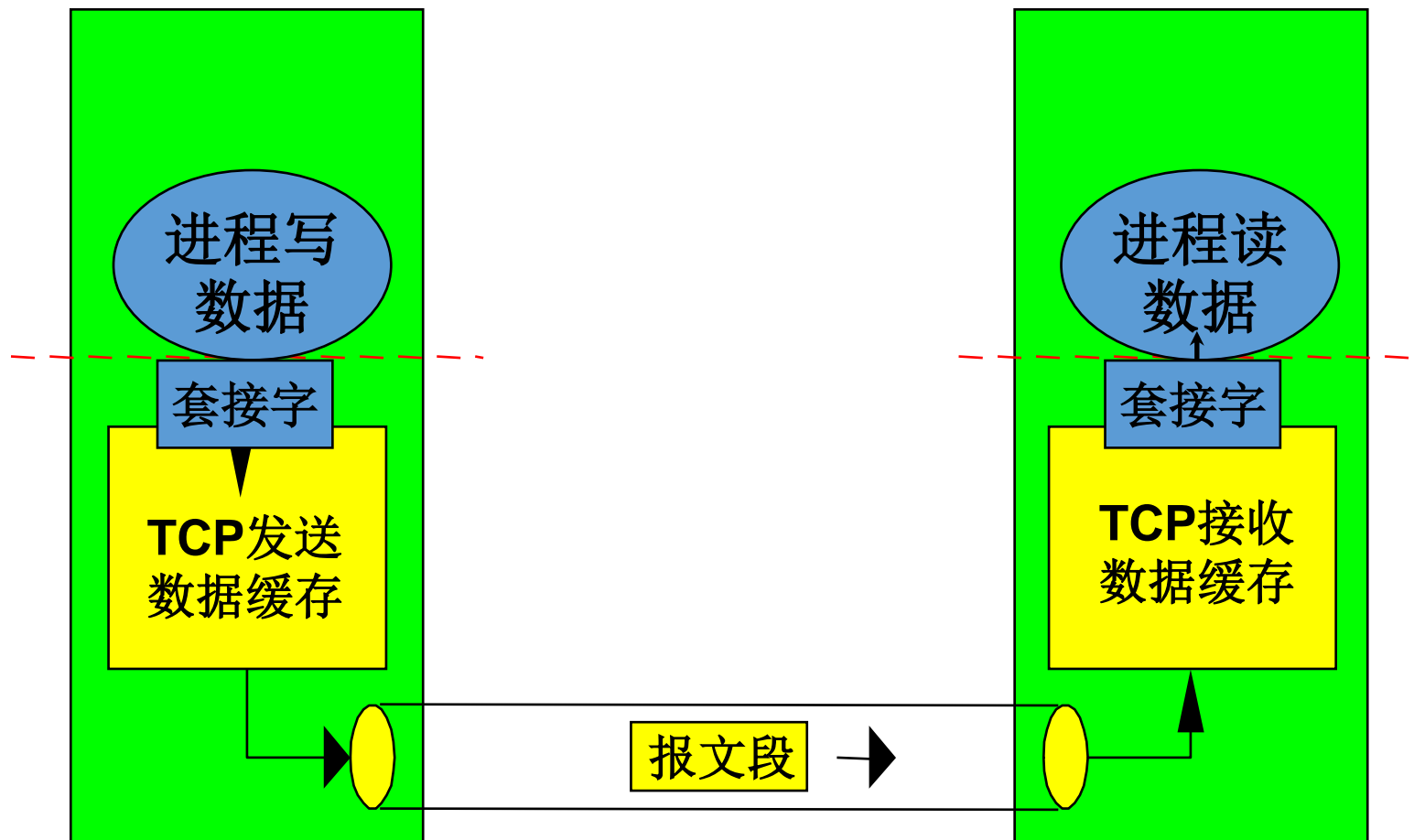


TCP: 概述 RFCs: 793, 1122, 1323, 2018, 2581

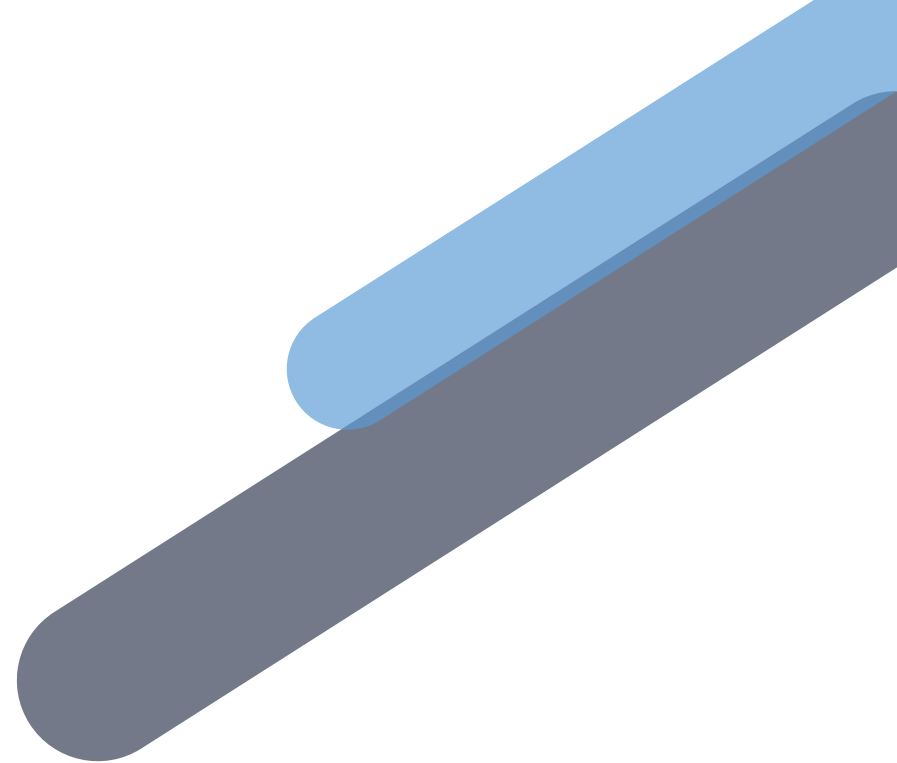
- 点到点:
一个发送者,一个接收者
- 可靠按序的字节流:
没有“信息边界”
- 流水线:
TCP 拥塞和流量控制设置窗口大小
- 收发缓冲区

- 全双工数据:
同一个连接上的双向数据流
MSS: 最大报文段长
- 面向连接:
在数据交换前握手(交换控制信息)
初始化发送方和接收方的状态
- 流量控制:
发送方不会淹没接收方

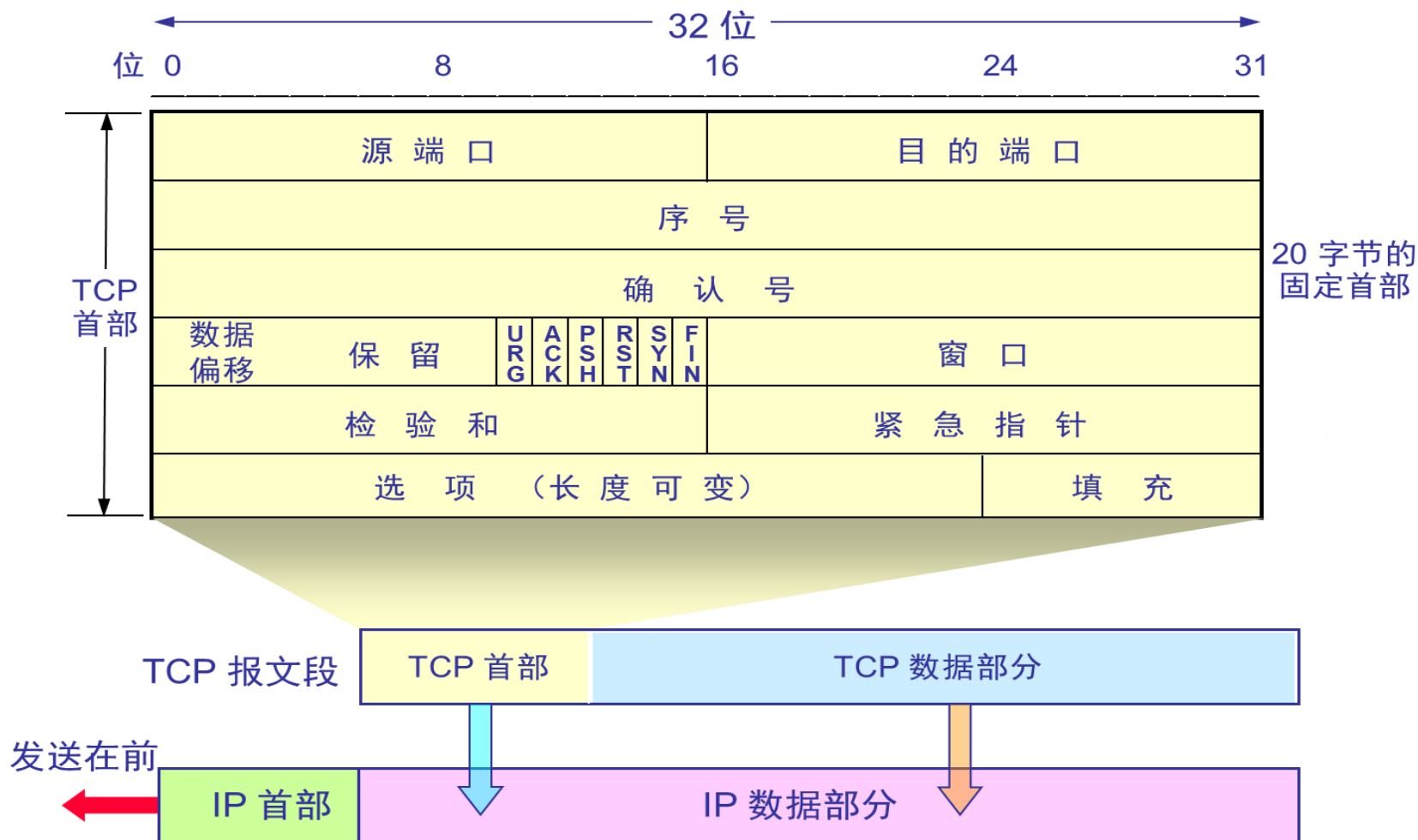
TCP: 概述 RFCs: 793, 1122, 1323, 2018, 2581



● 报文段结构



TCP 报文段的首部格式



注：这是早期首部格式，后来在标志位处加入了CWR, ECE（教材P154图）



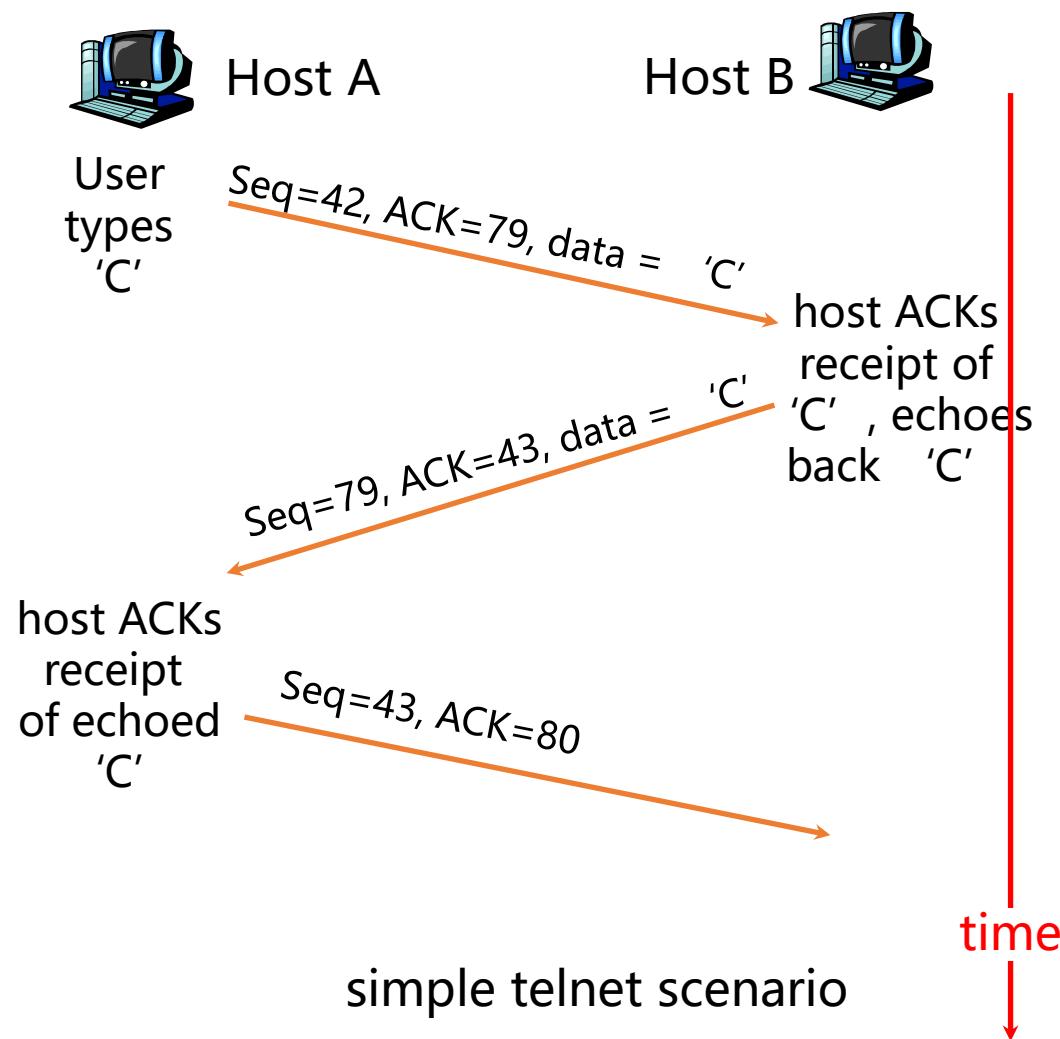
在图中，源端口和目的端口各占16位，序号和确认号各占32位。数据偏移字段——占4字节，用来让对方设置发送数据的标志，单位为字节（即每字节数据前加一个字节）。窗口和紧急指针各占16位，用来让对方设置接收数据的窗口大小。检验和占16位，用来校验数据的完整性。选项和填充占16位，用来扩展TCP头部的功能。

TCP 序号和确认

- 序号:
数据段中第一个字节在数据流中的位置编号
- 确认:
期望从另外一边收到的下一个字节的序号
- 累计ACK

问: 接收方如何处理失序的数据段

答: TCP规范没有明确规定, 由编程人员处理



TCP 往返时延的估计和超时

- 问: 如何设置 TCP 超时值?

答: 比 RTT 长; 但 RTT 变化

太短: 不成熟的超时

不必要的重传

太长: 对数据段丢失响应慢

TCP 往返时延的估计和超时

问: 如何估计 RTT?

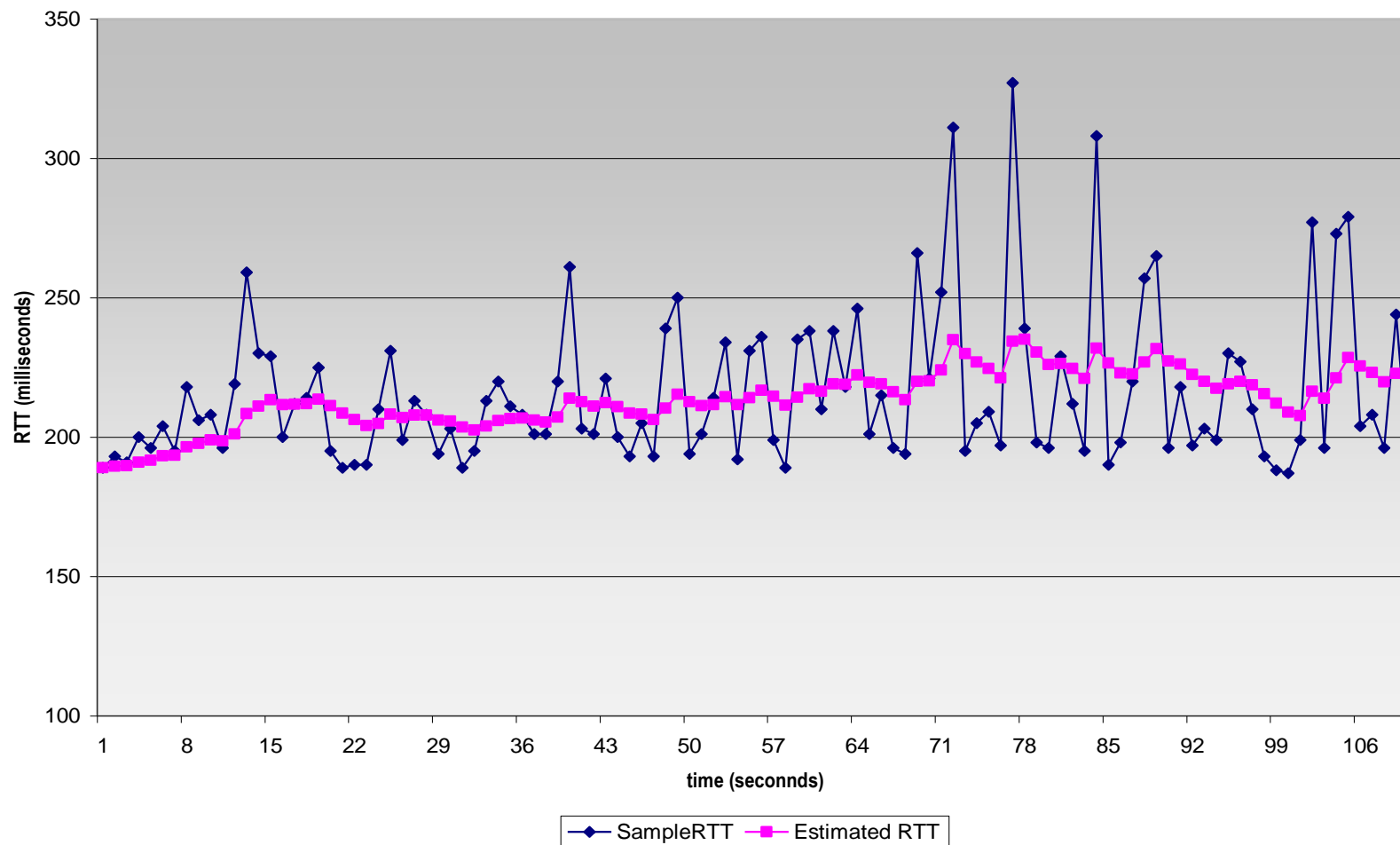
- 样本RTT (SampleRTT) : 测量从报文段发送到收到确认的时间
 - 忽略重传
- 样本RTT会变化,因此需要一个样本RTT均值 (Estimated RTT)
 - 对收到的样本RTT要根据以下公式进行均值处理

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

上述均值计算被称为: 指数加权移动平均, 典型的: $\alpha = 0.125$

RTT 估计例子

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr



TCP往返时延的估计和超时

- 设置超时

- EstimatedRTT 加上 “安全余量”

EstimatedRTT变化大 -> 更大的安全余量

- SampleRTT 偏离 EstimatedRTT多少的估计

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

(典型地, $\beta = 0.25$)

然后设置超时时间间隔:

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

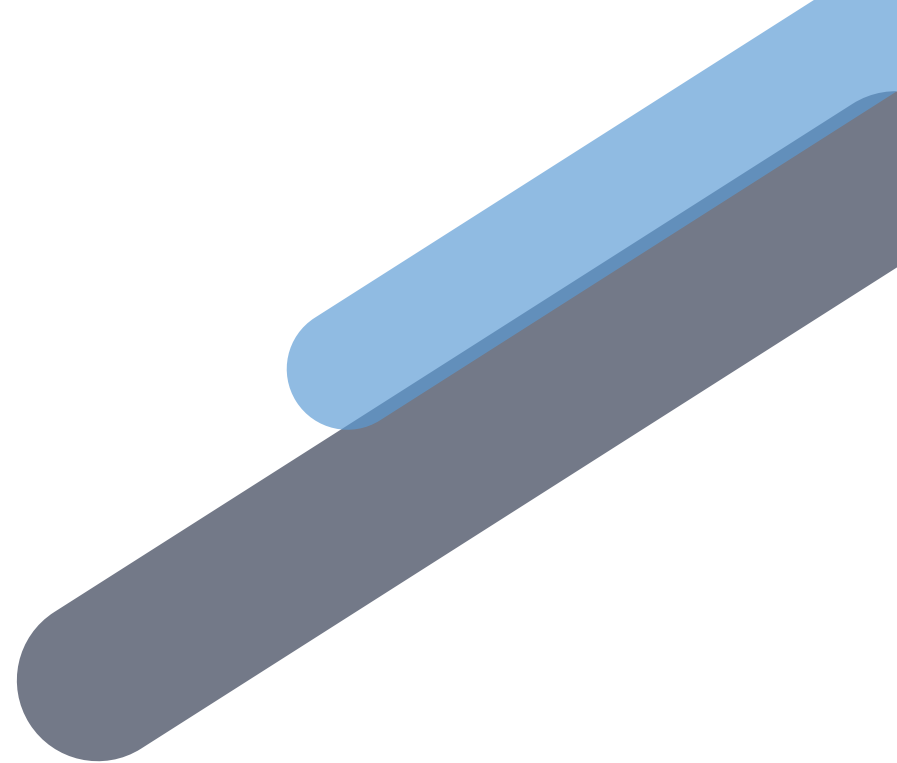


TCP往返时延的估计和超时初始化

- 设置超时

- 初始时TimeoutInterval设置为1秒
- 第一个样本RTT获得后, $\text{EstimatedRTT} = \text{SampleRTT}$,
 $\text{DevRTT} = \text{SampleRTT} / 2$,
 $\text{TimeoutInterval} = \text{EstimatedRTT} + \max(G, K * \text{DevRTT})$
($K=4$, G 是用户设置的时间粒度)

- **可靠数据传输机制**



可靠数据传输

- TCP在IP不可靠服务之上创建rdt服务
- 流水线技术处理报文段
- 累积确认
- TCP 使用单个重发定时器
- 触发重发:
 - 超时事件
 - 重复确认

TCP 发送方 (简化的)

```
NextSeqNum = InitialSeqNum  
SendBase = InitialSeqNum
```

```
loop (forever) {  
    switch(event)
```

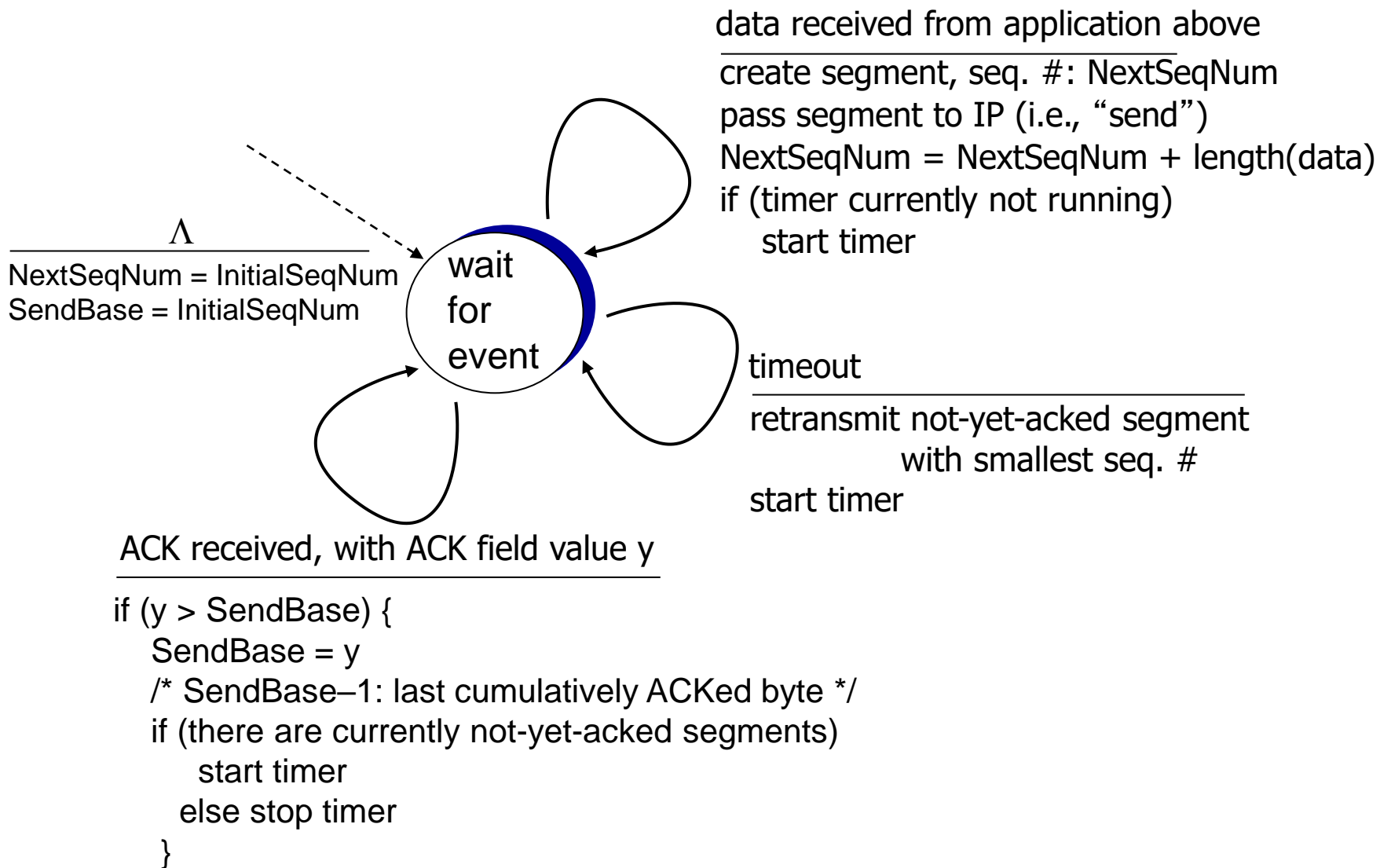
```
    event: data received from application above  
        create TCP segment with sequence number NextSeqNum  
        if (timer currently not running)  
            start timer  
        pass segment to IP  
        NextSeqNum = NextSeqNum + length(data)
```

```
    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 发送方 (简化的)



TCP 发送方事件:

从应用程序接收数据:

- 用序号创建一个报文
- 序号是报文中第一个数据字节在字节流中的位置编号
- 如果没有启动定时器, 则启动定时器
 - 定时器是最早没有被确认的报文发送时启动的
 - 设置超时间隔: TimeoutInterval

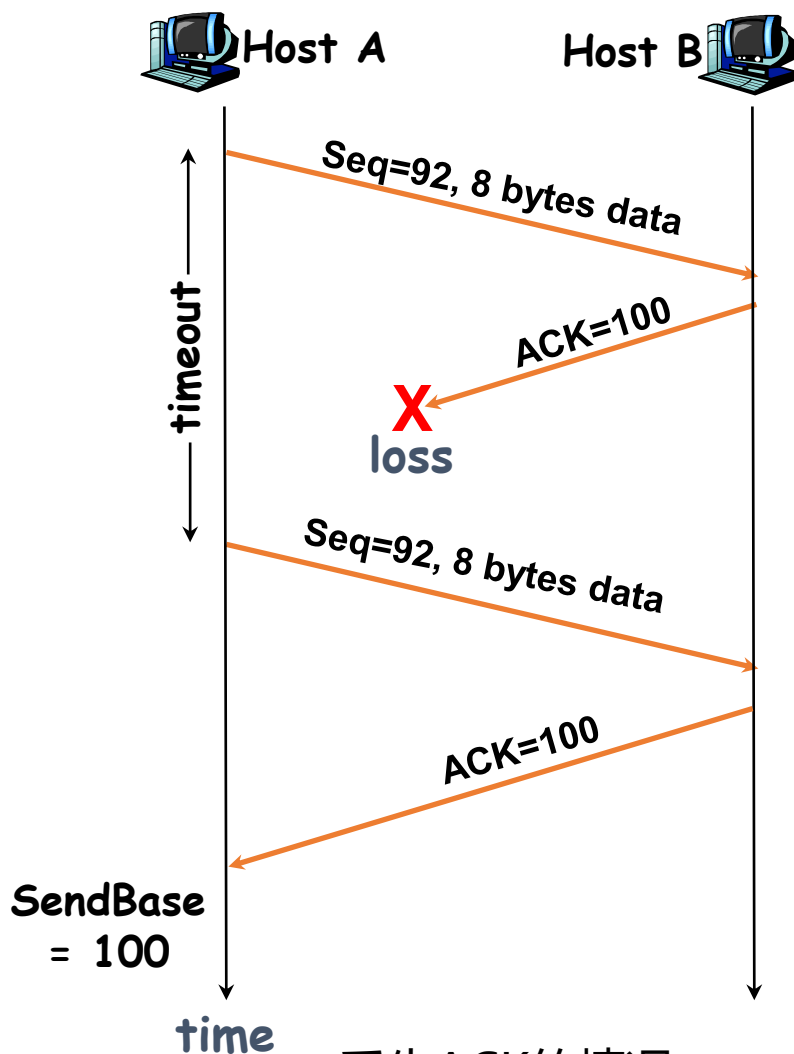
超时:

- 重发导致超时的报文
- 重新开始定时器

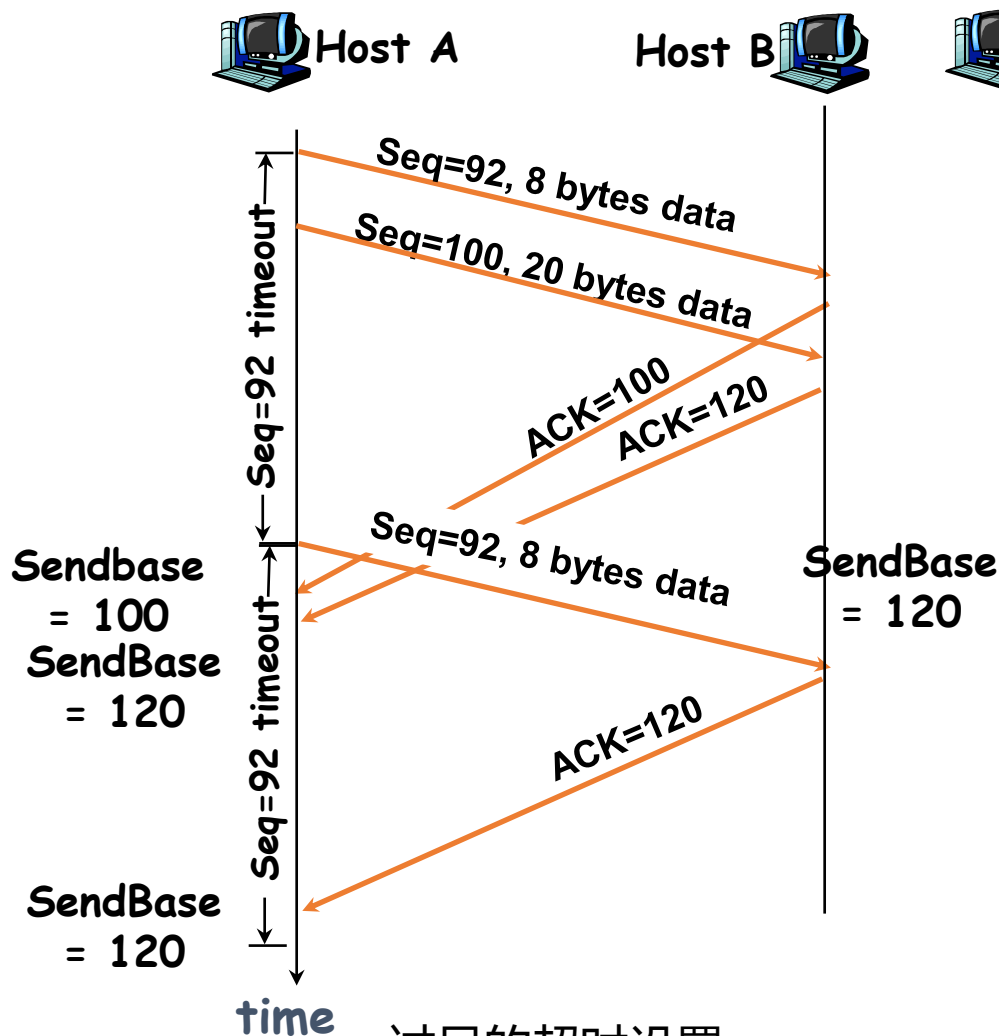
收到确认:

- 如果ACK落在窗口之内, 则确认对应的报文, 并且滑动窗口
- 若还有未确认的报文, 重新开始定时器

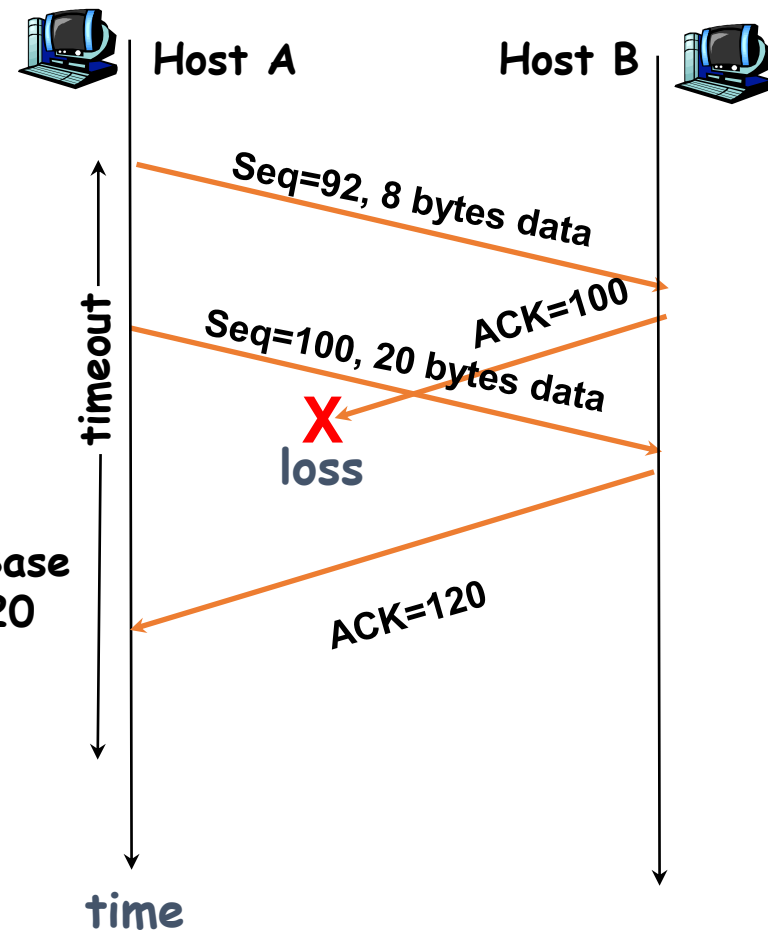
TCP: 重发场景



丢失ACK的情况



过早的超时设置



累积ACK的情况

快速重传

- 超时触发重传存在问题:
超时周期往往太长——
 - 重传丢失报文之前要等待很长时间,因此增加了网络的时延

- 发送方可以在超时之前通过重复的ACK检测丢失报文段
 - 发送方常常一个接一个地发送很多报文段
 - 如果报文段丢失,则发送方将可能接收到很多重复的ACKs
- 如果发送方收到一个确认后再收到3个对同样报文段的确认, 则发送方认为该报文段之后的数据已经丢失。
 - 启动**快速重传**: 在定时器超时之前重发丢失的报文段

快速重传算法

```
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
        }
    }
```

对已确认的报文
段的重复确认

快速重传

产生TCP ACK 的建议 [rfc 5681]

接收方的事件

TCP 接收方行为

期望序号的报文段按序到达. 所有在期望序号以前的报文段都被确认

延迟**ACK**. 等到 **500ms**看是否有下一个报文段, 如果没有, 发送**ACK**

期望序号的报文段按序到达. 另一个按序报文段等待发送**ACK**

立即发送单个累积**ACK**, 确认两个有序的报文段

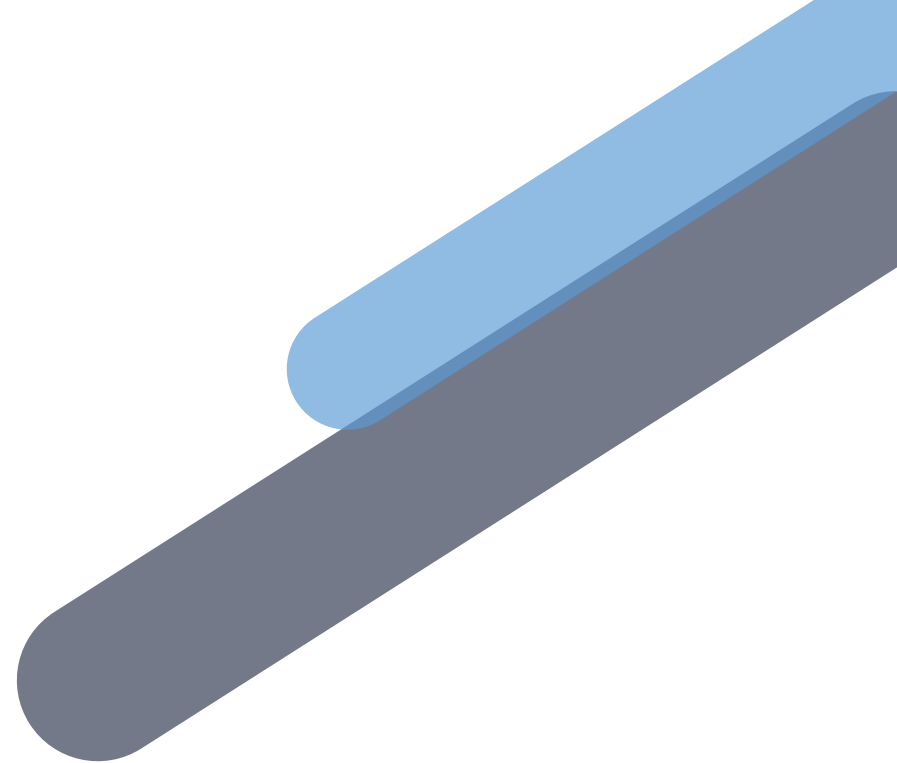
收到一个失序的报文段, 高于期望的序号, 检测到缝隙

立即发送重复 **ACK**, 指出期望的序号

到达的报文段部分地或者完全地填充接收数据间隔

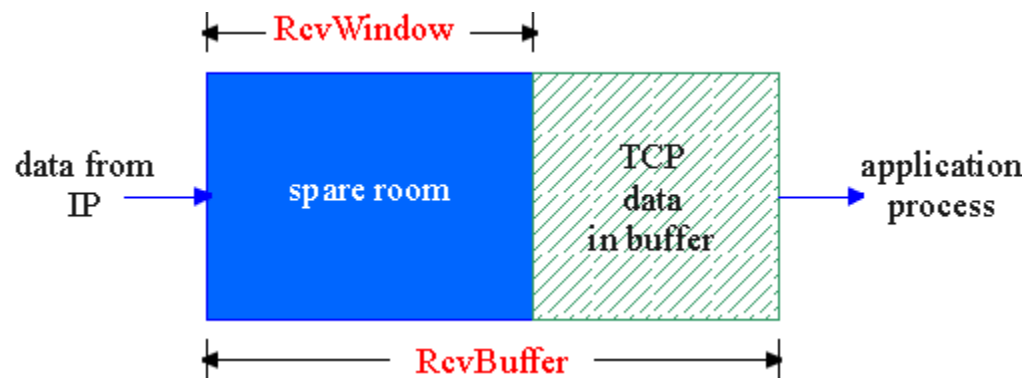
立即发送 **ACK**, 证实缝隙低端的报文段已经收到

- **流量控制**



TCP 流量控制

TCP连接的接收方有一个接收缓冲区:



应用程序可能从这个缓冲区读出数据很慢

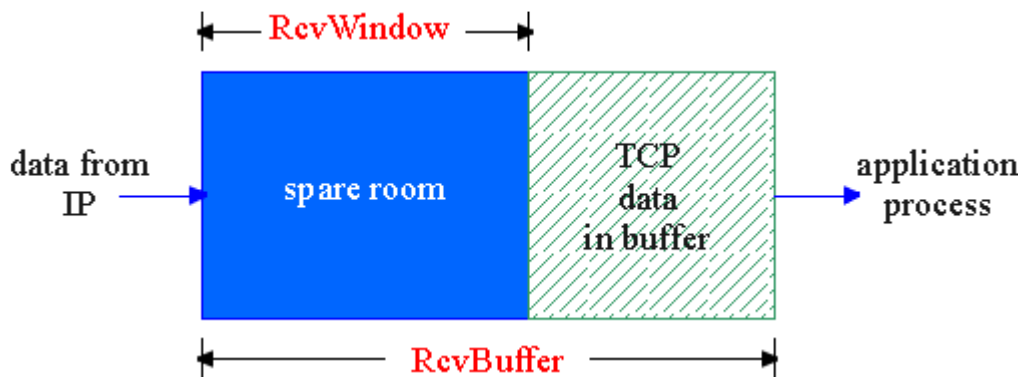
速度匹配服务

发送速率和接收应用程序的提取速率匹配

流量控制

发送方不能发送的太多太快, 让接收缓冲区溢出

TCP 流控: 如何工作

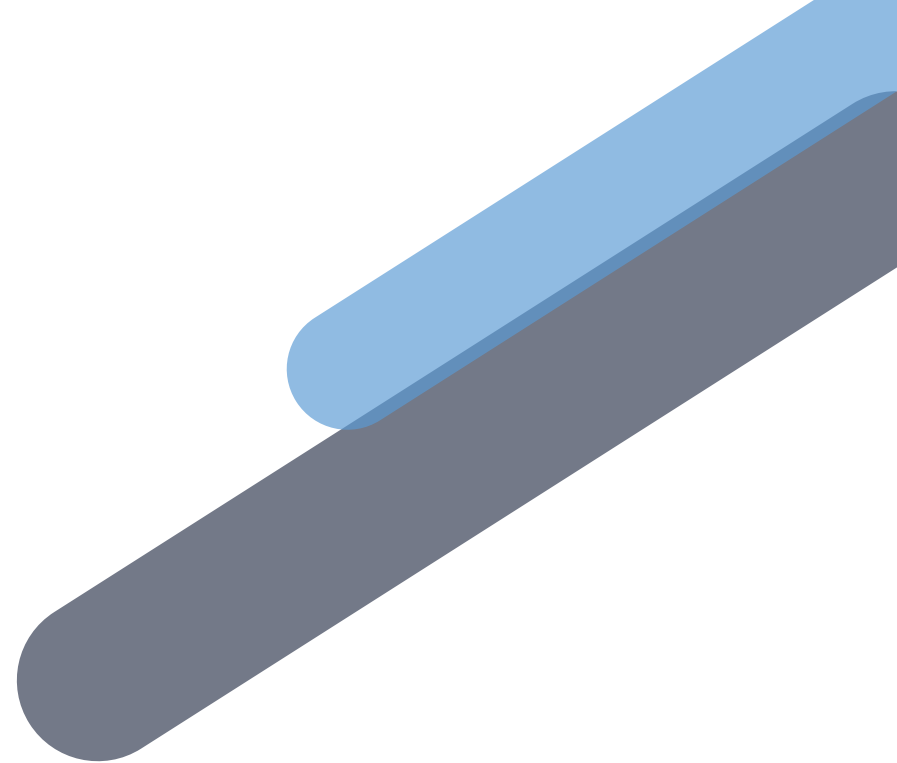


(假设 TCP 接收方丢弃失序的报文段)

- 流量控制使用接收窗口:接收缓冲区的剩余空间
- 接收方在报文段中宣告接收窗口的剩余空间
- 发送方限制没有确认的数据不超过接收窗口
 - 保证接收缓冲区不溢出

源 端 口										目 的 端 口									
序 号																			
确 认 号																			
数据 偏移		保 留		U R G	A C K	P S H	R S T	S Y N	F I N	窗 口									
检 验 和										紧 急 指 针									
选 项 （长 度 可 变）															填 充				

● 连接管理



TCP 连接管理

● 建立连接

回忆: TCP在交换数据报文段之前在发送方和接收方之间建立连接

- 初始化TCP 变量:

- 序号

- 缓冲区流控信息

(例, 接收窗口)

- 客户: 连接发起者

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

- 服务器: 被客户联系

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

三次握手:

Step 1: 客户发送TCP SYN报文段到服务器

- 指定初始的序号

- 没有数据

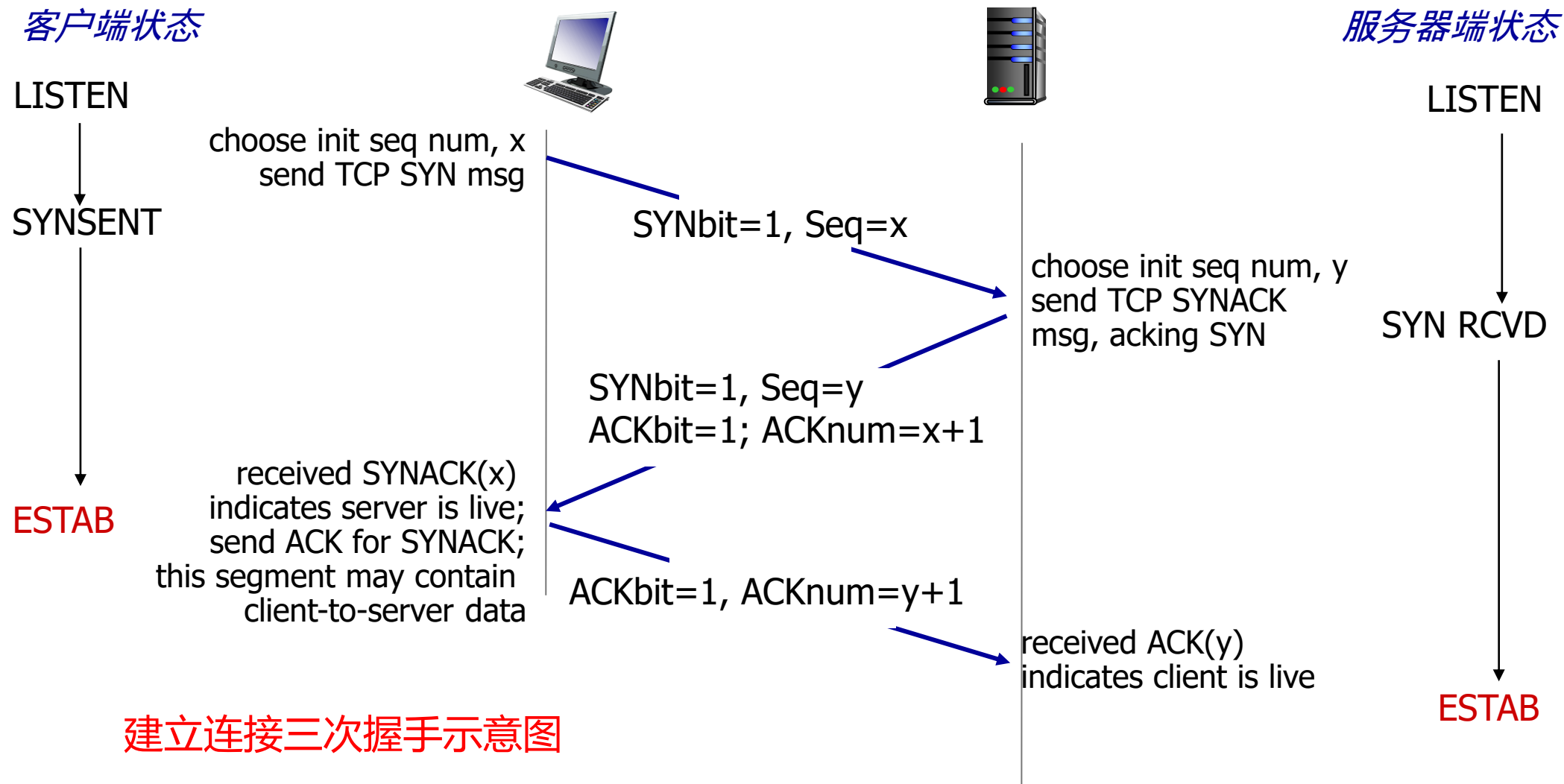
Step 2: 服务器接收SYN, 回复
SYN/ACK 报文段

- 服务器分配缓冲区

- 指定服务器的初始序号

Step 3: 客户接收 SYN/ACK, 回复 ACK
报文段, 可能包含数据

TCP 连接管理



TCP 连接管理(继续)

● 关闭连接

客户关闭套接字: `_clientSocket.close`

Step 1: 客户发送 TCP FIN 控制报文段到服务器

Step 2: 服务器接收 FIN, 回复 ACK. 进入半关闭连接状态;

Step 3: 服务器发送FIN到客户, 客户接收 FIN, 回复 ACK,
进入 “time wait” 状态

等待结束时释放连接资源

Step 4: 服务器接收 ACK. 连接关闭.

TCP 连接管理(继续)

客户端状态

ESTAB

`clientSocket.close()`

FIN_WAIT_1

can no longer
send but can
receive data

FIN_WAIT_2

wait for server
close

TIMED_WAIT

timed wait
for $2 * \text{max}$
segment lifetime

CLOSED



服务器端状态

ESTAB

CLOSE_WAIT

LAST_ACK

CLOSED

FINbit=1, seq=x

ACKbit=1; ACKnum=x+1

can still
send data

FINbit=1

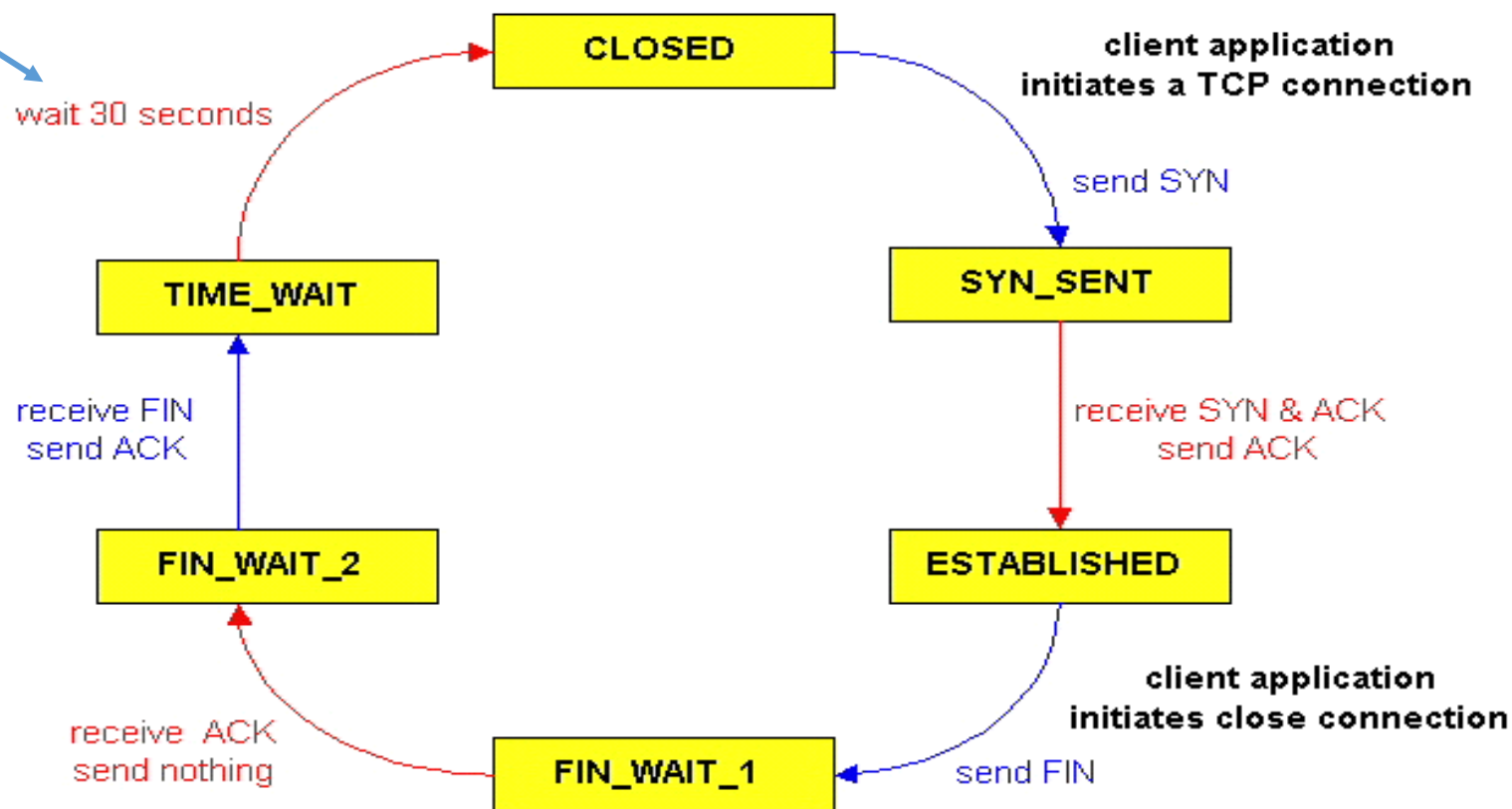
ACKbit=1; ACKnum=y+1

can no longer
send data

关闭连接四次挥手示意图

TCP 连接管理(继续)

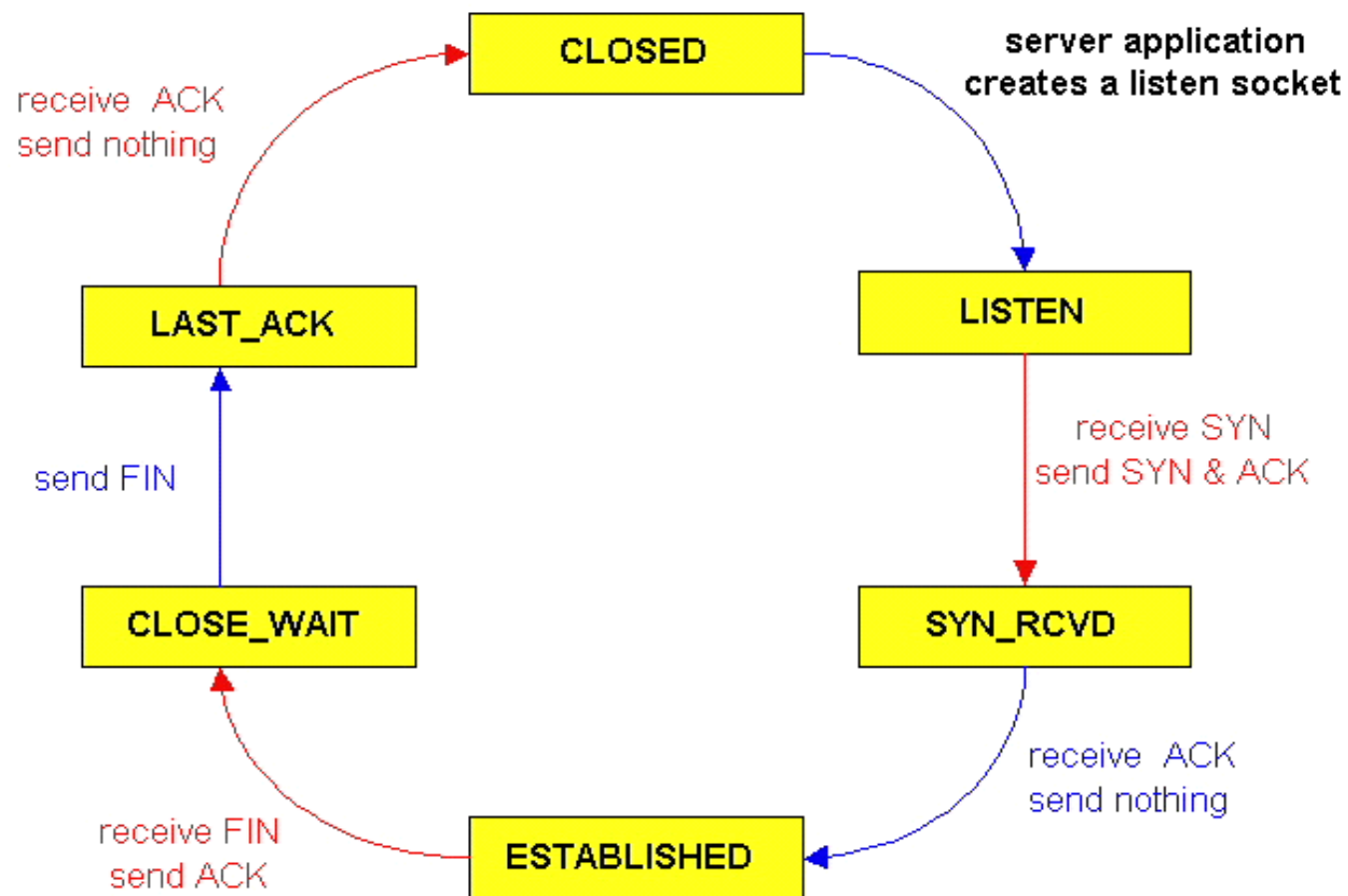
2MSL时间(Maximum Segment Lifetime)



TCP 客户端状态转换图

Ref: TCP释放连接时为什么time_wait状态必须等待2MSL时间: <https://www.cnblogs.com/huenchao/p/6266352.html>

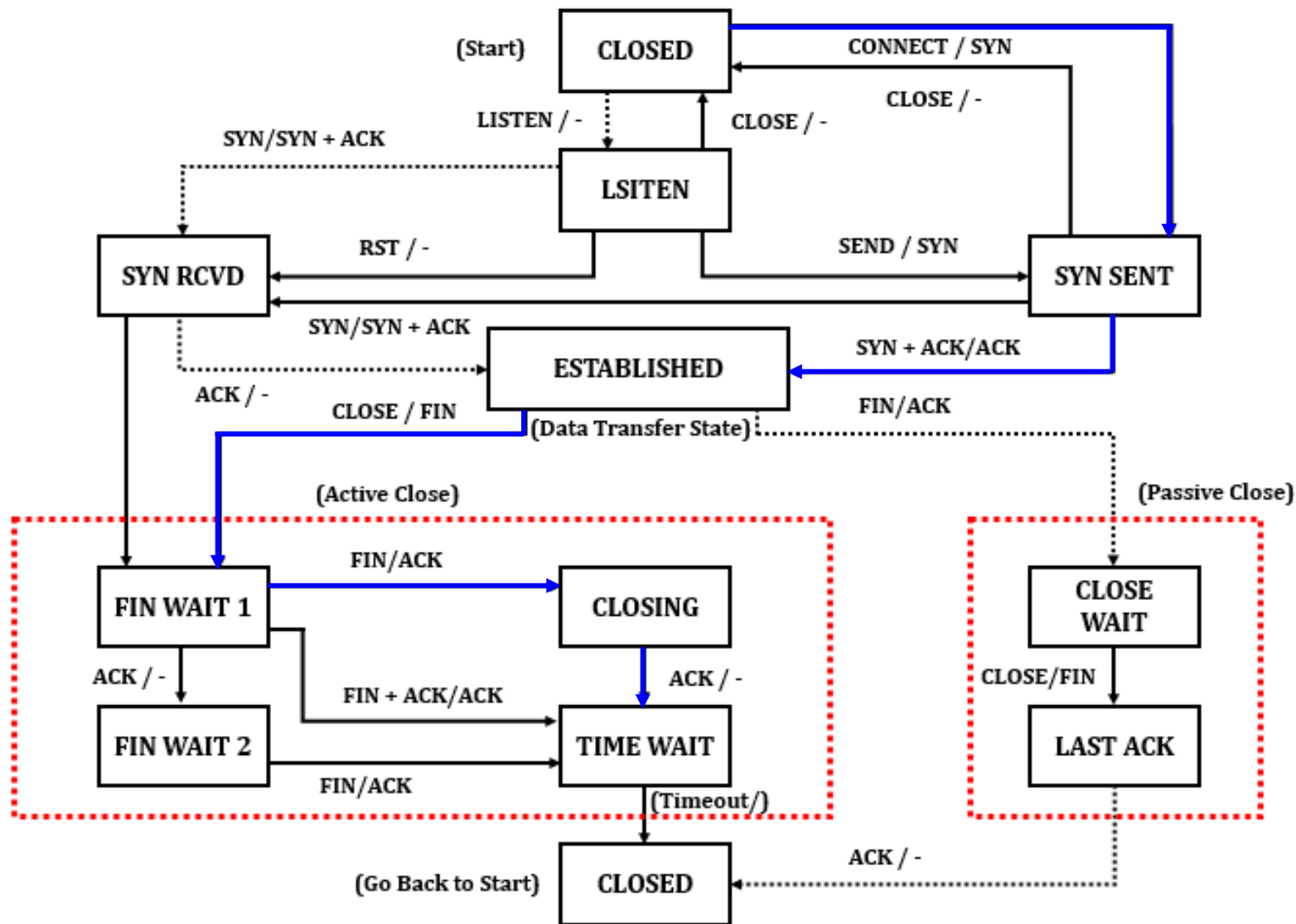
TCP 连接管理(继续)



TCP 服务器端状态转换图

TCP有限状态机*

- TCP 协议的操作可以使用一个具有 11 种状态的有限状态机来表示——如图，请课外学习。
- 延伸学习：TCP安全问题



06 拥塞控制原理



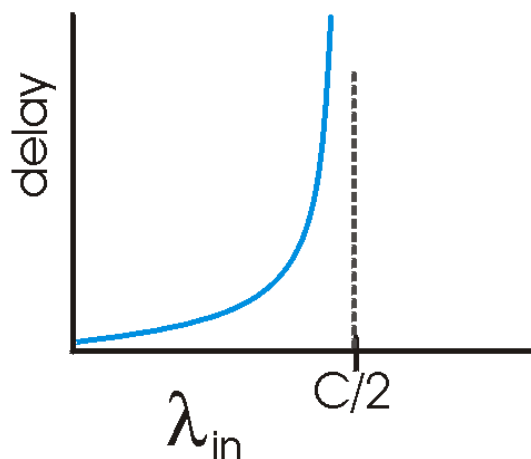
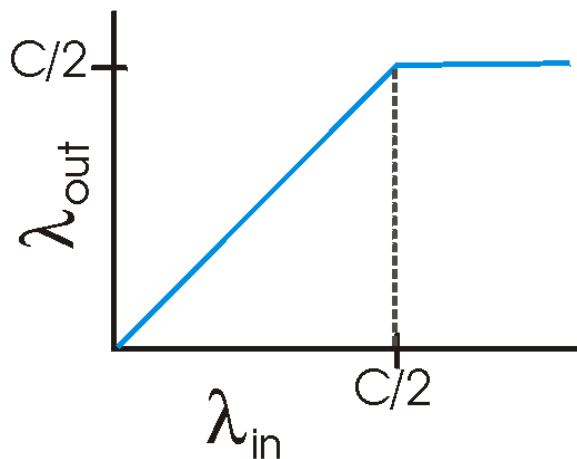
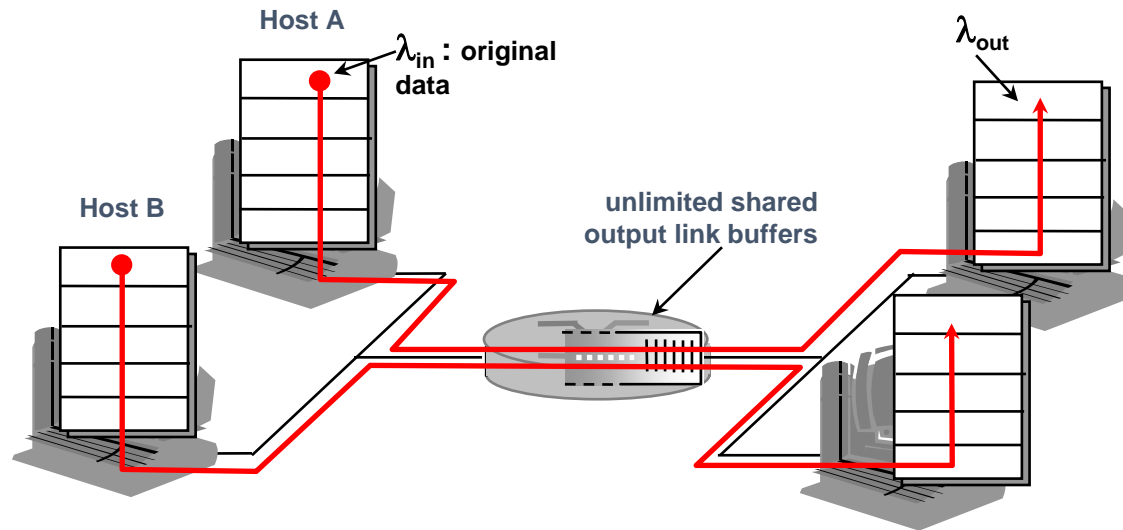
拥塞控制原理

- 拥塞:
 - 从信息角度看: “太多源主机发送太多的数据, 速度太快以至于网络来不及处理”
 - 和流量控制不同!
 - 表现:
 - 丢失分组 (路由器的缓冲区溢出)
 - 长延迟 (在路由器的缓冲区排队)

拥塞的原因和代价: 场景1

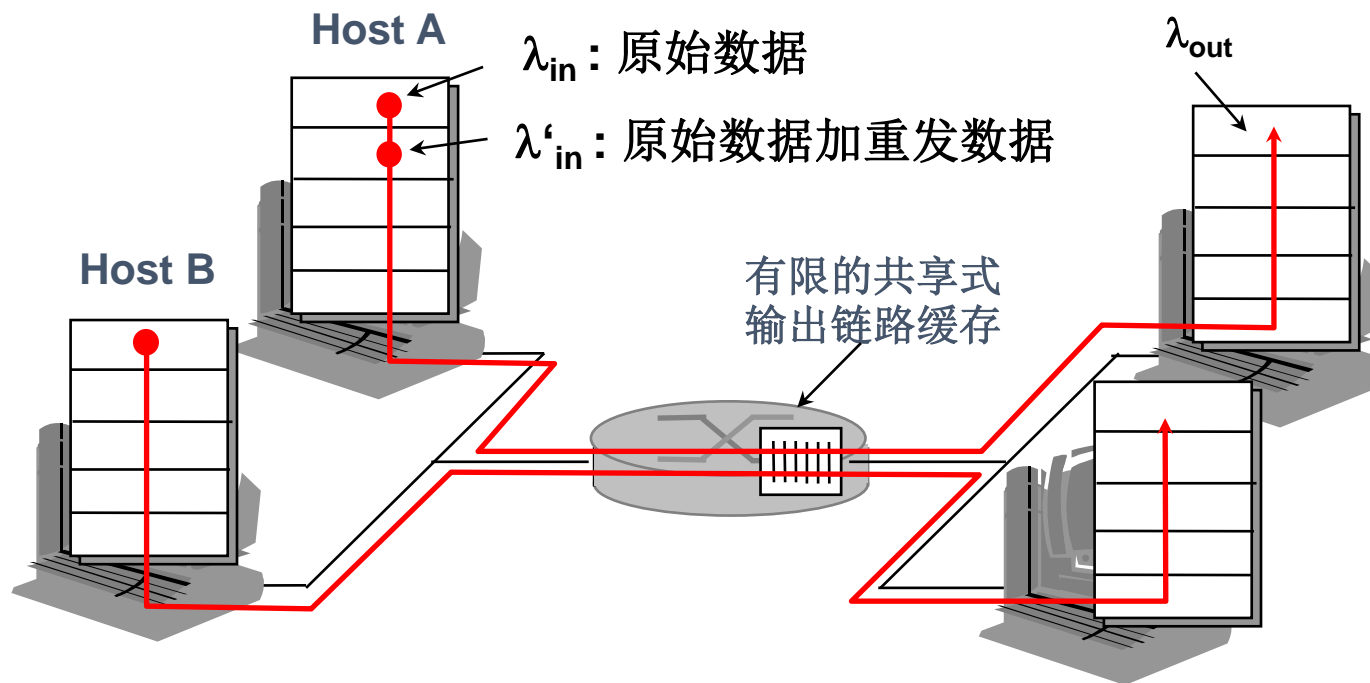
假设:

- 两个发送者, 两个接收者
- 一个路由器, 无限缓冲区
- 不执行重发
- 链路带宽为C
- 每个主机最大可达吞吐量 $C/2$, 总的吞吐量为C
- 但是, 拥塞时延在 $C/2$ 达到无限大



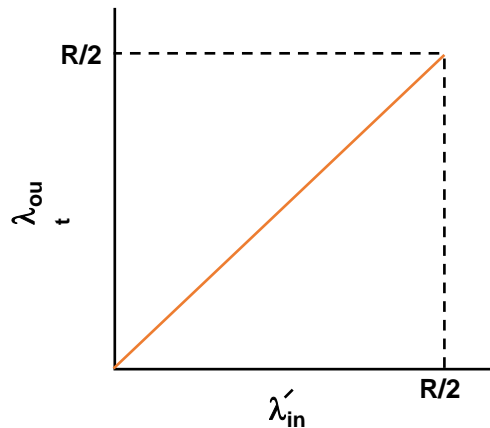
拥塞的原因和代价: 场景2

- 一个路由器，有限缓冲区
- 发送方重发丢失的报文

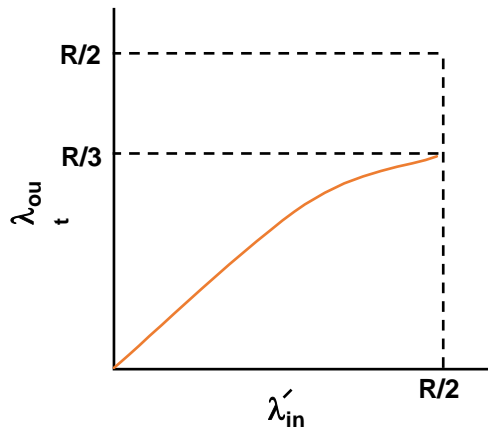


拥塞的原因和代价: 场景2

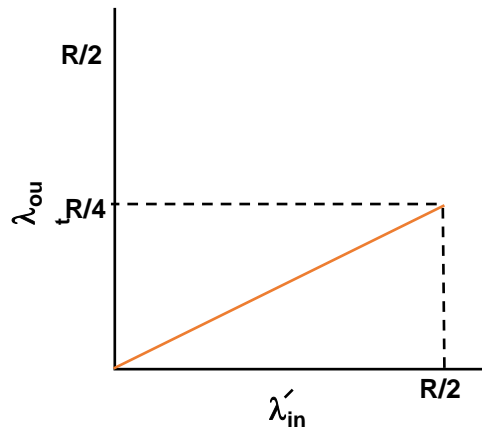
1. 总是: $\lambda_{in} = \lambda_{out}(\text{goodput})$
2. 仅当数据丢失时才重发:
3. 超时而没有丢失的报文重发: 导致同样的 λ_{out} 需要比完美情况更大的 λ'_{in}



a.



b.



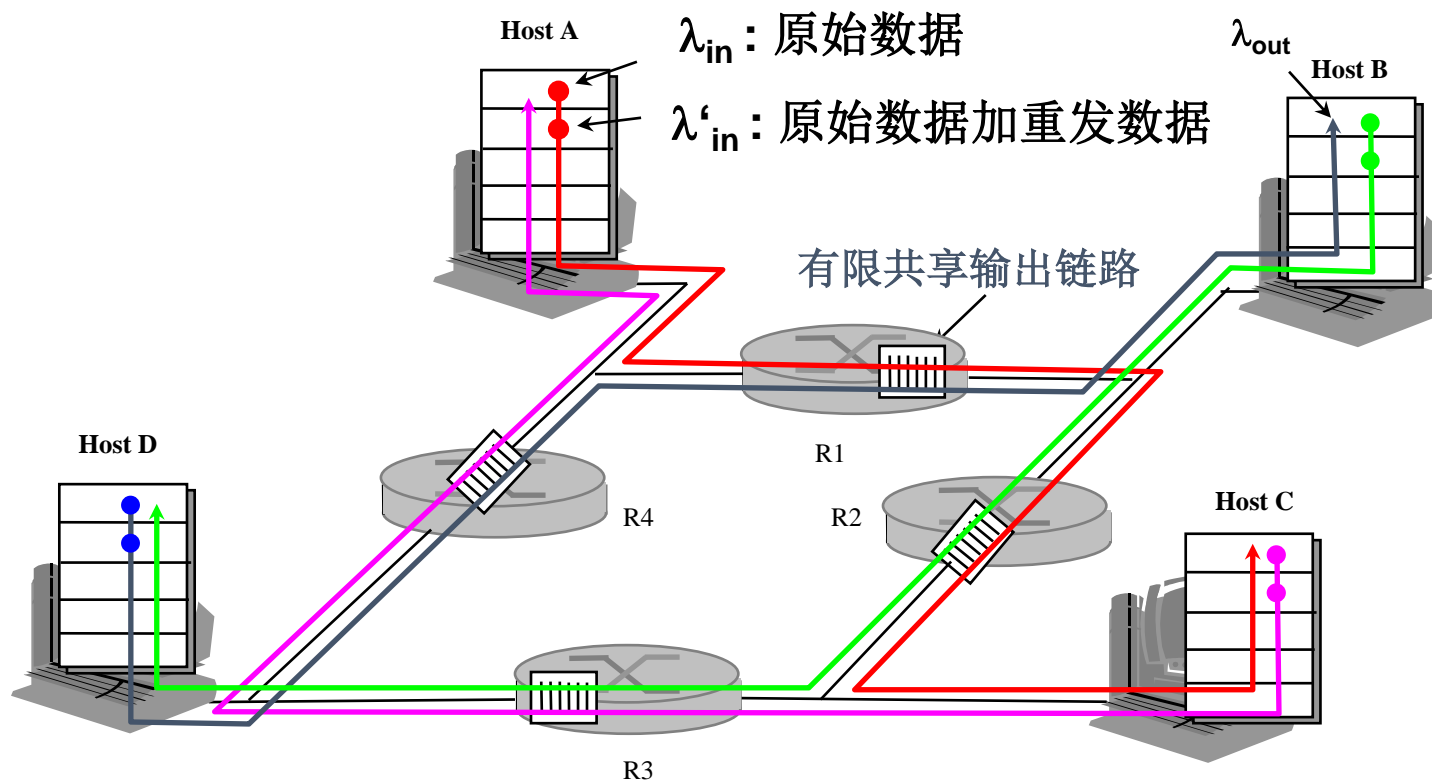
c.

拥塞的“代价”
更多的工作 (重发) 用来得到
“好的吞吐量”
不必要的重发: 链路需要运输
多个分组的拷贝

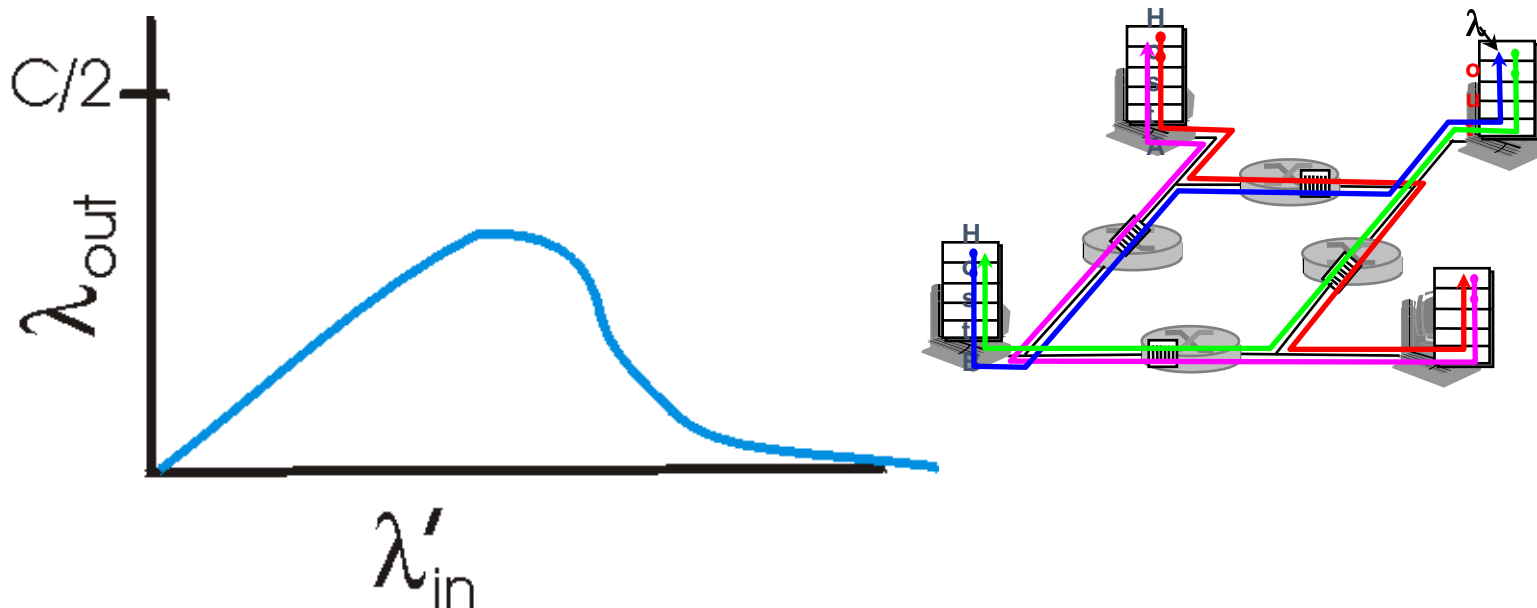
拥塞的原因和代价: 场景3

- 四个发送方
- 多跳路径
- 超时/重发

问: 在 λ_{in} 和 λ'_{in} 增加的时候什么会发生？



拥塞的原因和代价: 场景3



拥塞的另一个代价:
当分组丢失后, 任何上游路由器的发送能力都浪费了!

拥塞控制的方法

● 端到端拥塞控制:

- 没有从网络中得到明确的反馈
- 从端系统观察到的丢失和延迟推断出拥塞
- TCP采用的方法

● 网络辅助的拥塞控制:

- 路由器给端系统提供反馈
- 单bit指示拥塞 (SNA, DECnet, TCP/IP ECN, ATM)
- 指明发送者应该发送的速率

*情况分析: ATM ABR 拥塞控制 (了解)

ABR: 可用比特率:

- “弹性服务”
- 如果发送方通道“低载”：
发送方应该利用有效带宽
- 如果发送方通道拥塞：
发送方应该调节到保证速率

RM (资源管理) 信元:

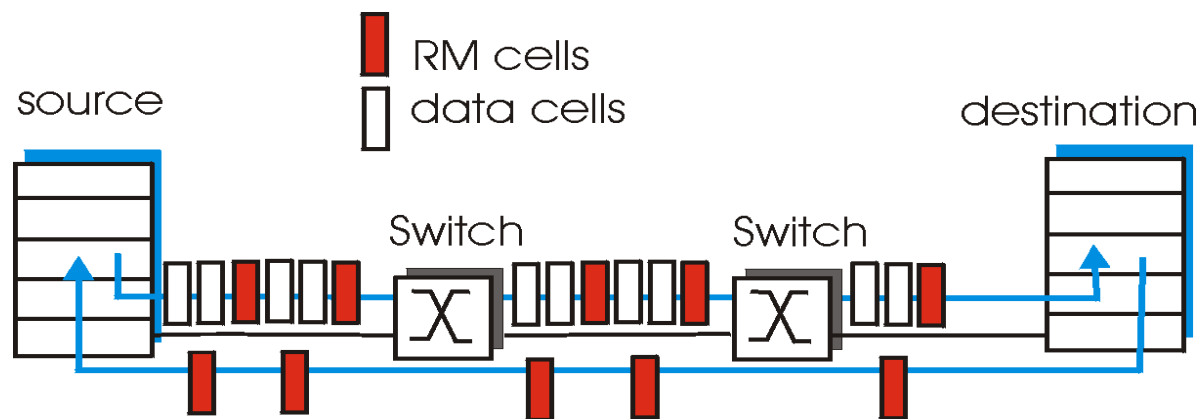
- 发送方发送,点缀在数据信元中
- RM信元中的bit是交换机设置 (网络辅助)

NI bit: 速率不要增加 (轻度拥塞)

CI bit: 拥塞指示

- 接收方不改变RM 信元的bit,将其返回给发送者

* 情况分析: ATM ABR 拥塞控制



- RM信元的两个字节的 ER (明确速率) 域
 - 拥塞的交换机可能降低信元中的 ER 值
 - 发送方的发送速率因此调整到通道支持的最低速率
 - 数据信元中的EFCI 位: 在拥塞的交换机中设置为1
- 如果数据信元有EFCI, 比RM先到, 发送方设置CI比特于返回的RM信元中

07 TCP 拥塞控制



TCP 拥塞控制

- 端到端控制 (没有网络辅助)
- 发送方限制发送:
LastByteSent-LastByteAcked
< min(CongWin , RcvWindow)
- 大体上,

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

CongWin 是动态的, 感知的网络拥塞的函数

- 发送方如何感知拥塞 ?
 - 丢失事件 = 超时或者 3 个重复的ACKs
 - TCP 发送方在丢失事件发生后降低发送速率 (CongWin)
- 三个机制:
 - 慢启动
 - AIMD
 - 对拥塞事件作出反应

TCP 慢启动 (slow start)

- 连接开始的时候, CongWin = 1 MSS

Example: MSS = 500 bytes & RTT = 200 msec

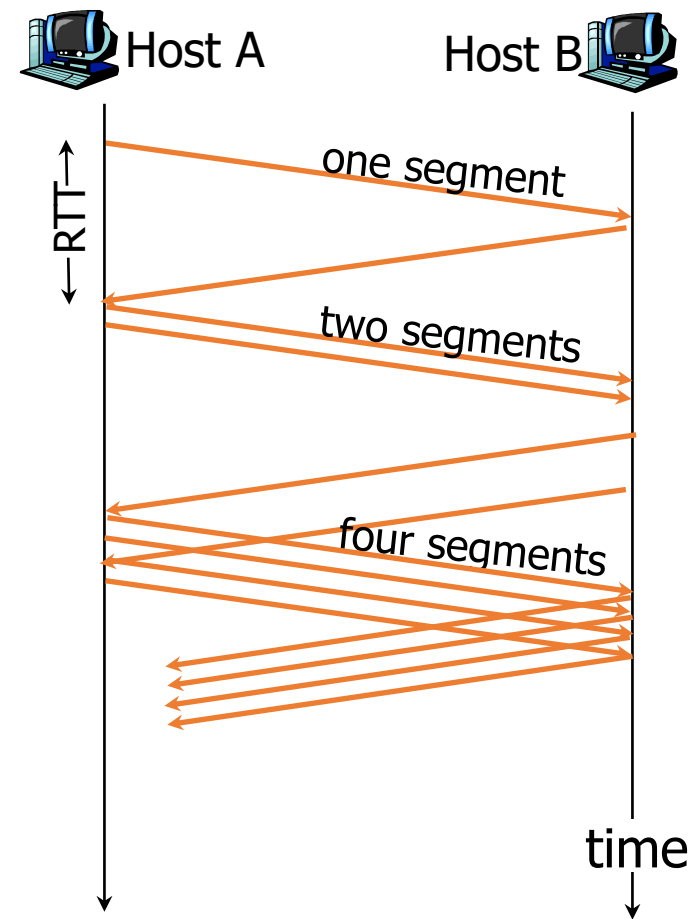
初始速率 = 20 kbps

- 有效带宽将 \gg MSS/RTT

希望尽快达到期待的速率, 故将以2的指数方式增加速率, 直到产生丢失事件, 或者达到某个阈值ssthresh

TCP 慢启动(更多)

- 当连接开始的时候以指数方式增加速率:
在每个 RTT 内倍增 CongWin——每
收到一个ACK, CongWin 加 1
- 总结: 初始速率慢但是呈指数快速增长



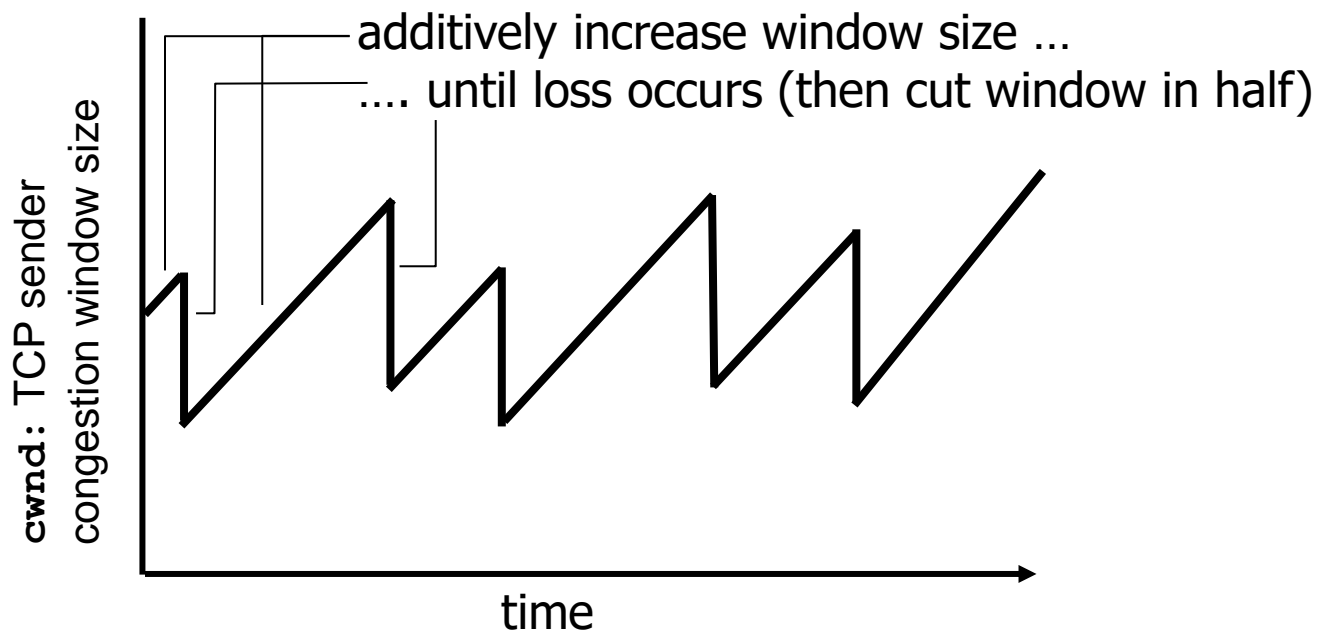
TCP AIMD(Additive-increase,multiplicative-decrease)

发送方增加传输速率（窗口大小），探测可用带宽，直到发生丢包事件

乘性递减：发生丢包事件后将拥塞窗口减半

加性递增：每个RTT内如果没有丢失事件发生，拥塞窗口增加一个MSS

AIMD 锯齿状
的行为特点：
探测带宽



Long-lived TCP connection

对拥塞事件的反应

- 当超时事件发生时:
 - CongWin 立即设置为 1个 MSS;
 - 窗口开始指数增长 (进入慢启动)
 - 到达一个阈值后再线性增长
- 收到三个重复的确认时:
CongWin 减半+3 (Reno版)
然后, 窗口线性增长

- 怎么理解不同的丢包事件?
 - 3 个重复的 ACKs 表明网络具有传输一些数据段的能力
 - 在三个重复的确认之前超时是“更加严重的警告”

注: 上述为TCP Reno版本的内容, 在TCP Tahoe版本里,
无论超时还是三个重复, 都直接将CongWin 置为 1个 MSS

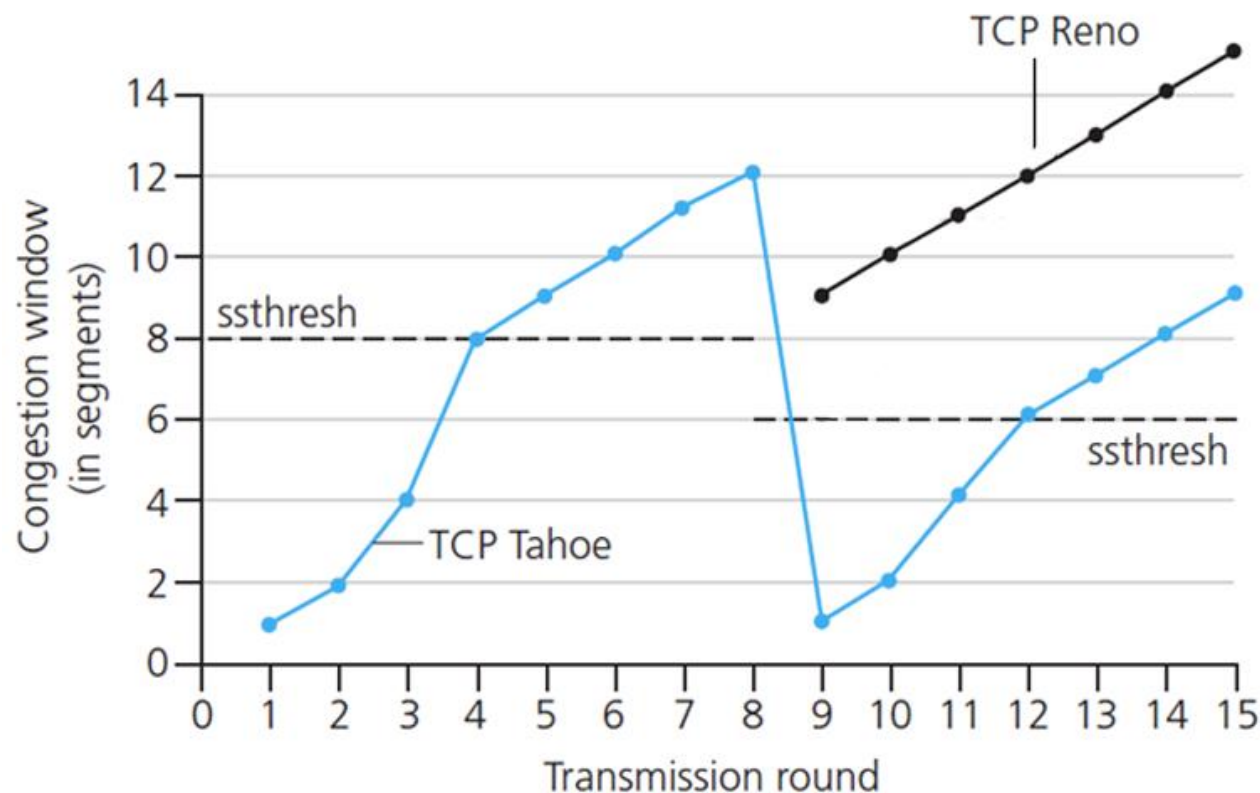
对拥塞事件的反应(更多)

什么时候从指数增加变为线性增加

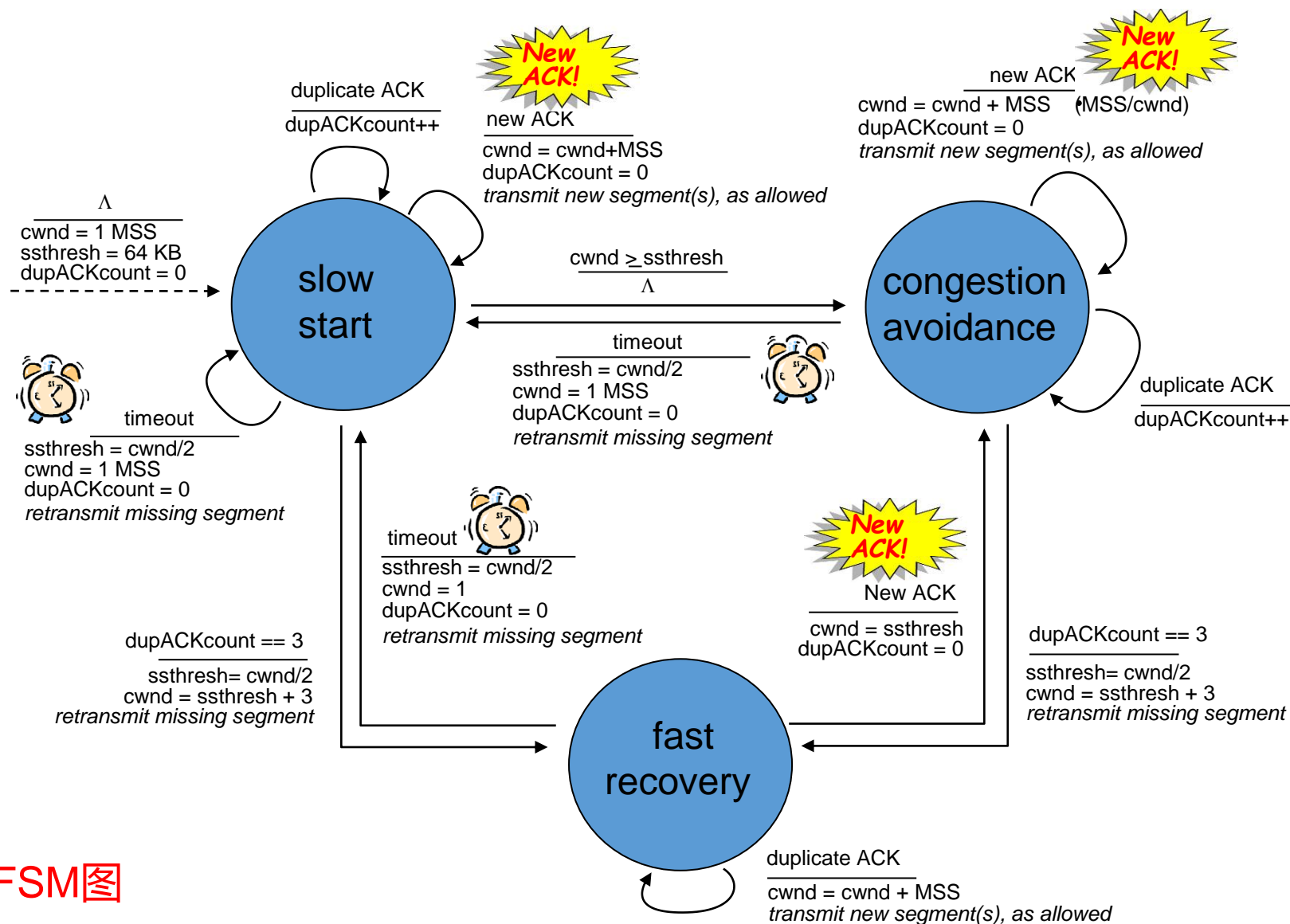
答: 当 CongWin 达到超时前的一半的时候

实现:

- 变化的阈值
- 发送丢失事件后, 阈值设置为丢失前的 CongWin 的一半, 最小为2



总结: TCP 拥塞控制



TCP拥塞控制的FSM图

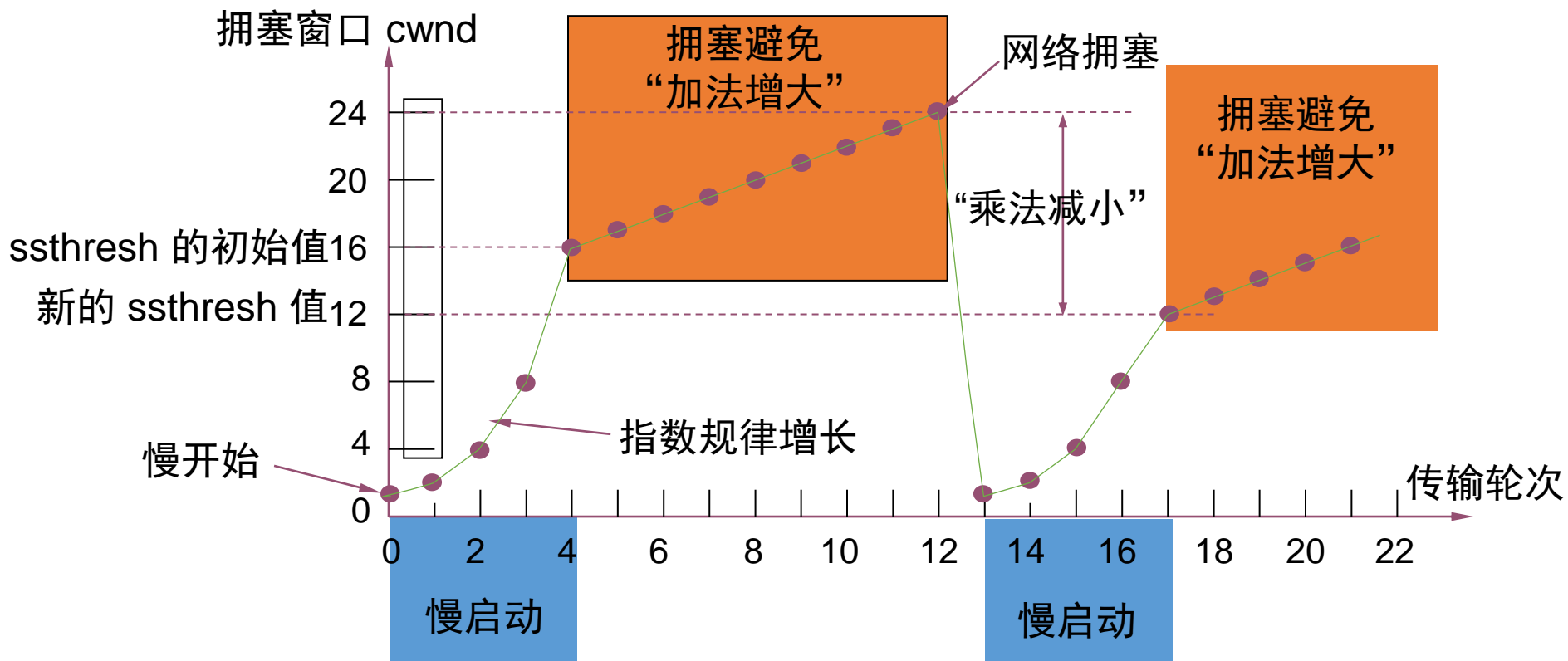
总结: TCP 拥塞控制

- 当 CongWin 低于阈值, 发送方处于慢启动阶段, 窗口指数增长.
- 当 CongWin 高于阈值, 发送方处于拥塞避免阶段, 窗口线性增长.
- 当三个重复的ACK 出现时, 阈值置为 CongWin/2 并且 CongWin 置为阈值加上3个MSS并进入快速恢复阶段, 此时每收到一个重复的ACK 拥塞窗口增加1MSS, 如果收到新的ACK则拥塞窗口置成阈值) .
- 当超时发生时, 阈值置为 CongWin/2 并且 CongWin 置为1 MSS.

TCP拥塞控制的RFC文档: <https://tools.ietf.org/html/rfc5681>

中文版见: <https://blog.csdn.net/fengfengdiandia/article/details/81354592>

慢开始和拥塞避免算法的实现举例(Tahoe)



当 TCP 连接进行初始化时，将拥塞窗口置为 1。图中的窗口单位不使用字节而使用报文段。慢开始门限的初始值设置为 16 个报文段，即 $ssthresh = 16$ 。

TCP 平均吞吐量

- 假设忽略慢启动
- 假设在丢失发生时, 设 W 是窗口大小
- 如果窗口为 W , 吞吐量是 W/RTT
- 丢失发生后, 窗口降为 $W/2$, 吞吐量为 $W/2RTT$.
- 平均吞吐量为 $0.75 W/RTT$

TCP 未来

举例: 1500 字节的数据段, 100ms RTT, 希望10 Gbps 吞吐量
要求窗口大小 $W = 83,333$ 个报文段

按照一个连接的平均吞吐量公式(L 为丢包率):

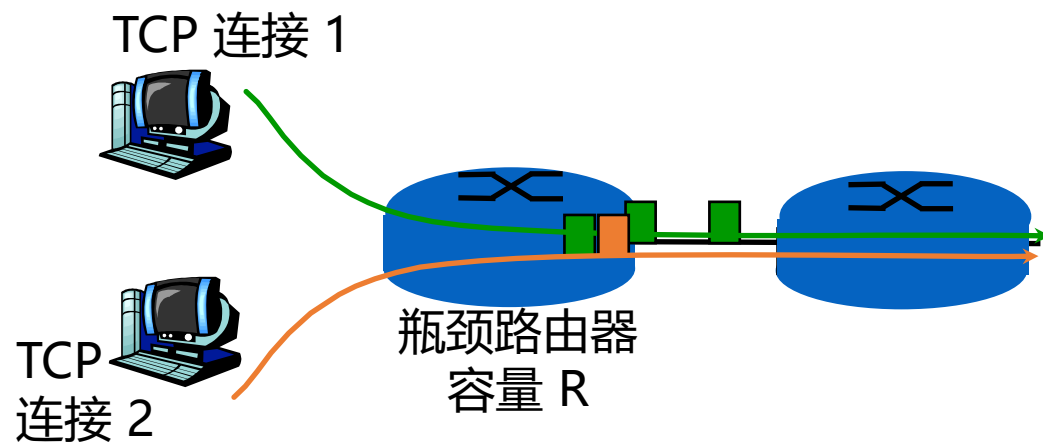
$$\frac{1.22 \cdot MSS}{RTT \sqrt{L}}$$

则现在的TCP为达到10 Gbps的吞吐量, 要求 $L = 2 \times 10^{-10}$, 即
每500万个报文段只允许丢失一个报文段。

用于高速的TCP的新版本是必要的!

TCP 公平*

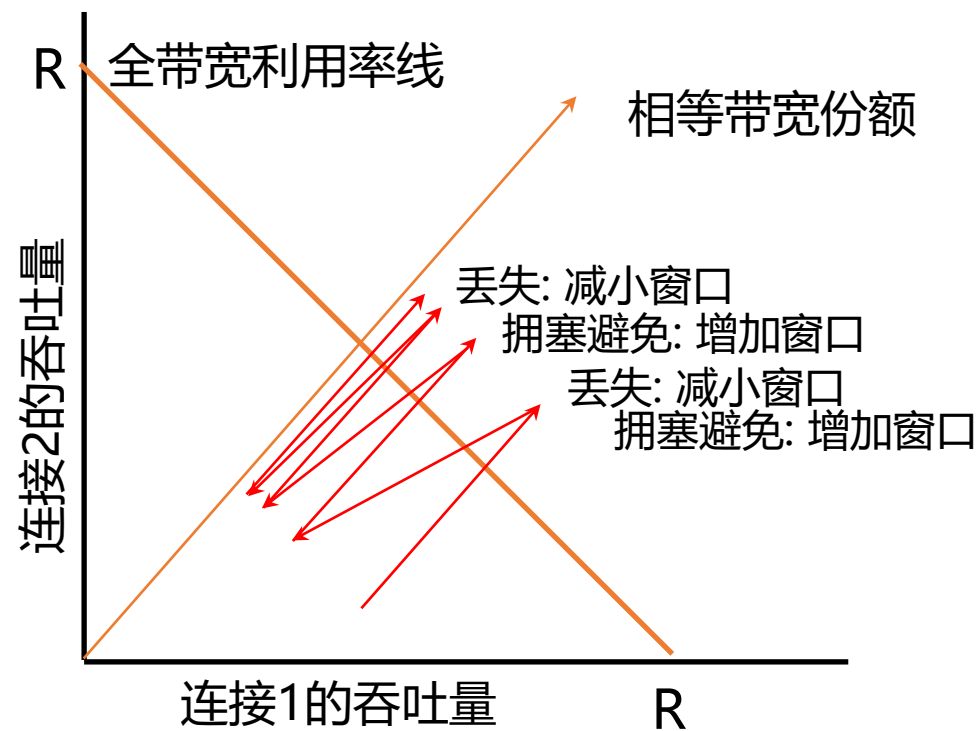
公平目标: 如果K个TCP 共享带宽为R的瓶颈链路每个应该有 R/K 的平均速率。



TCP 为什么是公平的?

两个竞争的会话:

按照斜率1加性增加, 同时吞吐量增加, 乘性降低等比例减少吞吐量



公平 (更多)

公平性和UDP

多媒体应用常常不用 TCP

不希望速率被拥塞扼杀

用UDP代替TCP:

用固定速率发送音/视频, 容忍报文丢失

公平性和并行 TCP 连接

不能阻止应用程序在两个主机之间打开平行的TCP连接.

Web 浏览器就这样做

例子: R 速率的链路支持9个连接;

新的应用要求一个 TCP, 得到速率 $R/10$

新的应用要求11个TCPs, 则得到超过 $R/2$!

第三章 总结

- 传输层服务背后的原理:
 - 多路复用, 多路分解
 - 可靠数据传输
 - 流控
 - 拥塞控制
- 因特网中的实例和实现
 - UDP
 - TCP
- 下面:
 - 离开网络的 “边界” (应用层, 传输层)
 - 进入网络 “核心”