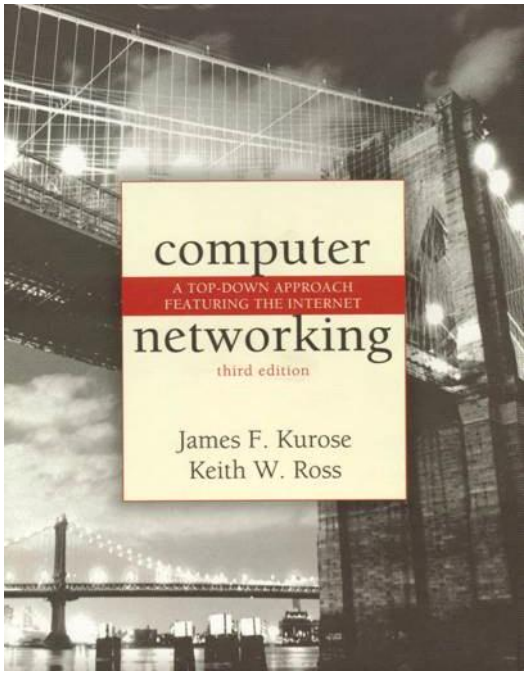
第三章 传输层


computer
A TOP-DOWN APPROACH
FEATURING THE INTERNET
networking
third edition
James F. Kurose
Keith W. Ross

第三章: 传输层

目标:

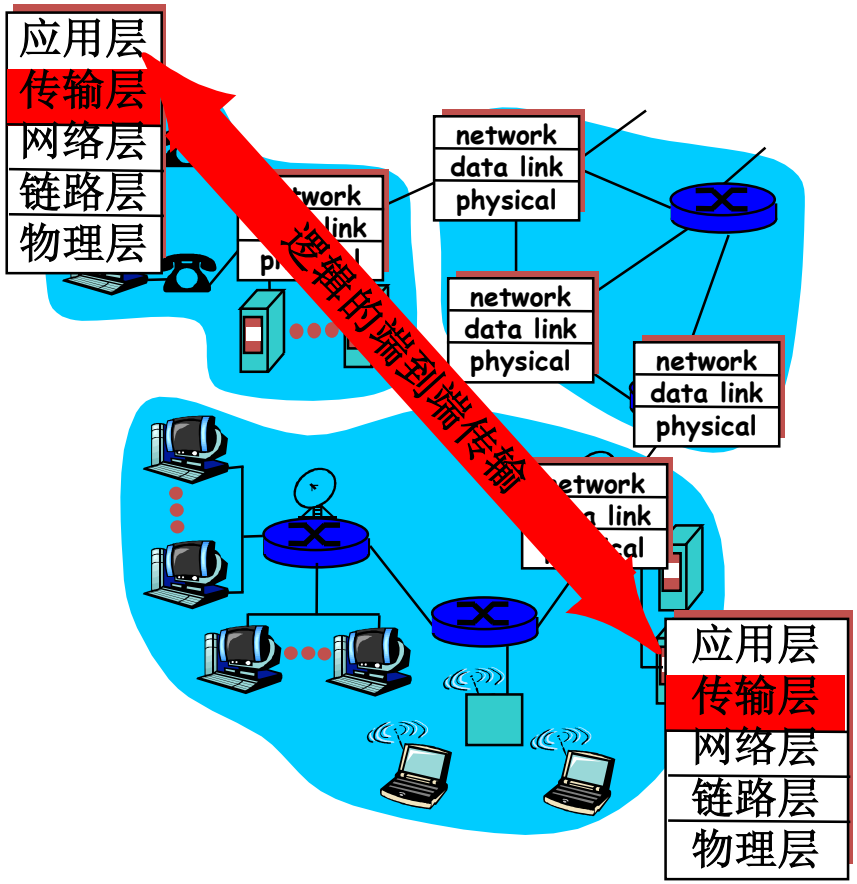
- 理解传输层服务以后的原则:
  - 复用/分解复用
  - 可靠数据传输
  - 流量控制
  - 拥塞控制
- 学习因特网的传输层协议:
  - UDP: 无连接传输
  - TCP: 面向连接传输
  - TCP 拥塞控制

第三章 提纲

- 3.1 传输层服务
- 3.2多路复用和多路分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
- 3.5 面向连接传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

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: 将应用程序报文封装成若干报文段传递给网络层,
  - 接收方: 将报文段解封装组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

- *网络层:*两个主机之间的逻辑通信
- *传输层:*两个进程之间的逻辑通信
  - 可靠, 增强的网络层服务

传输层和网络层类比

- 东西海岸两个家庭分别有12个孩子，每人每星期互相写一封信，东海岸收发员Ann，西海岸收发员Bill
  - 应用层报文=信件
  - 进程=孩子
  - 主机=家庭
  - 传输层协议=Ann和Bill （只在家工作，不参与邮政）
  - 网络层协议=邮政服务 （只识别家庭地址，不知内容）
  - 进程地址=信封上的孩子姓名
  - 主机地址=信封上的家庭地址

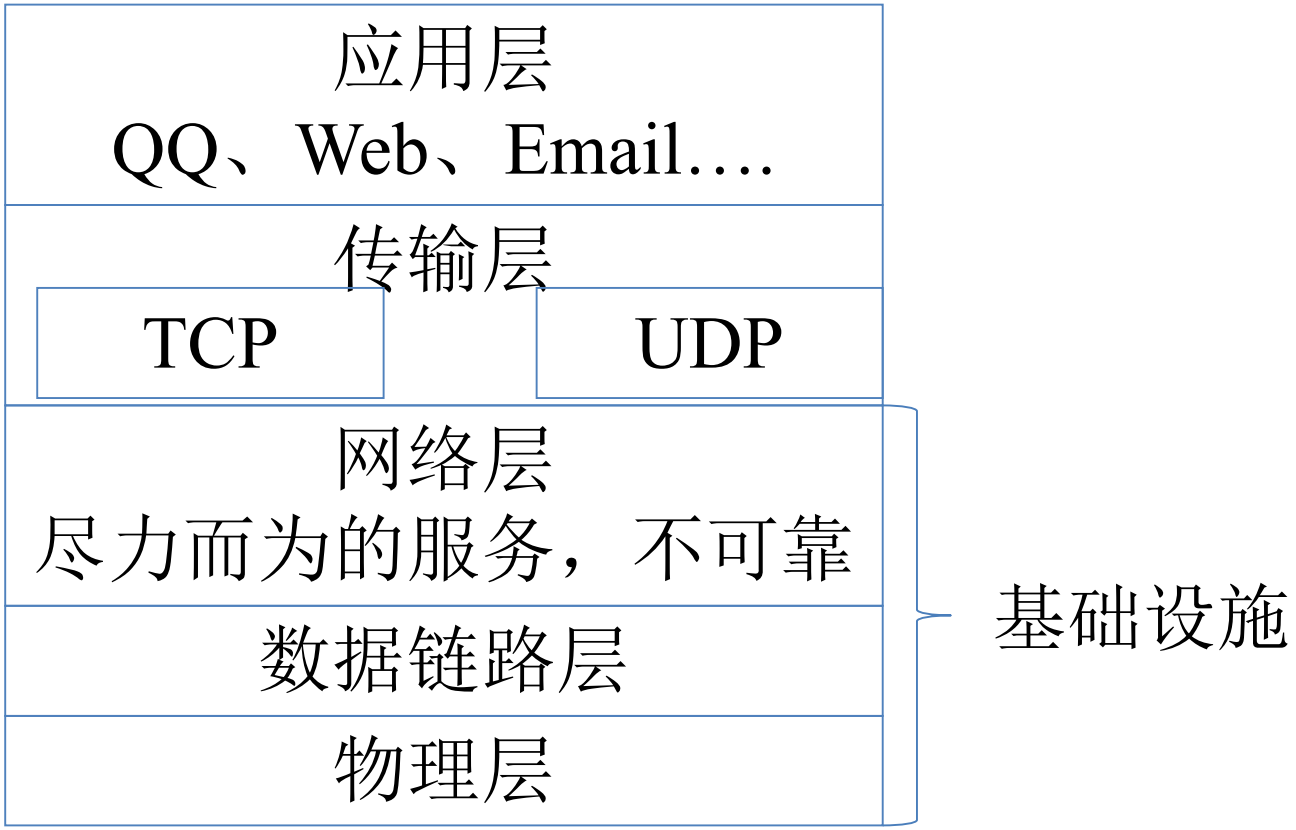
12个孩子完成12封信（应用层报文）	12个孩子分别读取各自的信件
Ann收集12封信，信上有收件人姓名（端口号）	Bill根据包裹内信件的收件人姓名分发到各个小孩
12封信打包为一个大包裹，写上对方家庭地址（IP地址），邮政投递服务	根据家庭地址 邮政投递服务将包裹投递到该家庭

Ann和Bill提供的服务受制于邮政服务

Internet 传输层协议

- 网络层的服务
  - 尽力而为交付（Best effort delivery service）
  - 不可靠的服务
    - 不确保交付；不保证按序交付；不保证数据完整性
- 可靠按序递交 (TCP服务)
  - 拥塞控制
  - 流量控制
  - 连接建立
- 不可靠的无序传递: UDP服务
  - “尽力传递” IP的直接扩展
- 不提供的服务:
  - 延迟保证
  - 带宽保证

Internet 传输层协议



第三章 提纲

- 3.1 传输层服务
- 3.2 多路复用和多路分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
- 3.5 面向连接传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

多路复用/多路分解

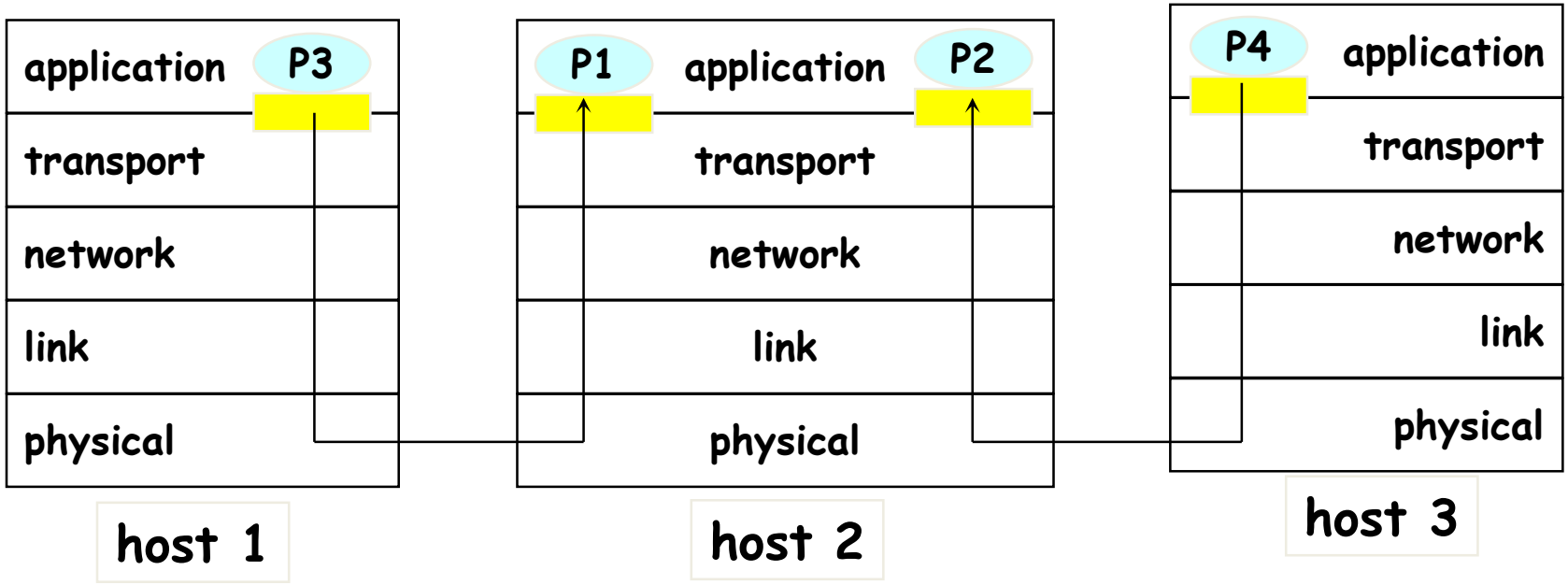
在接收主机多路分解：

将接收到的报文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
- 周知端口号:0-1023
  - HTTP:80;FTP:21;SMTP:25
- 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);
```

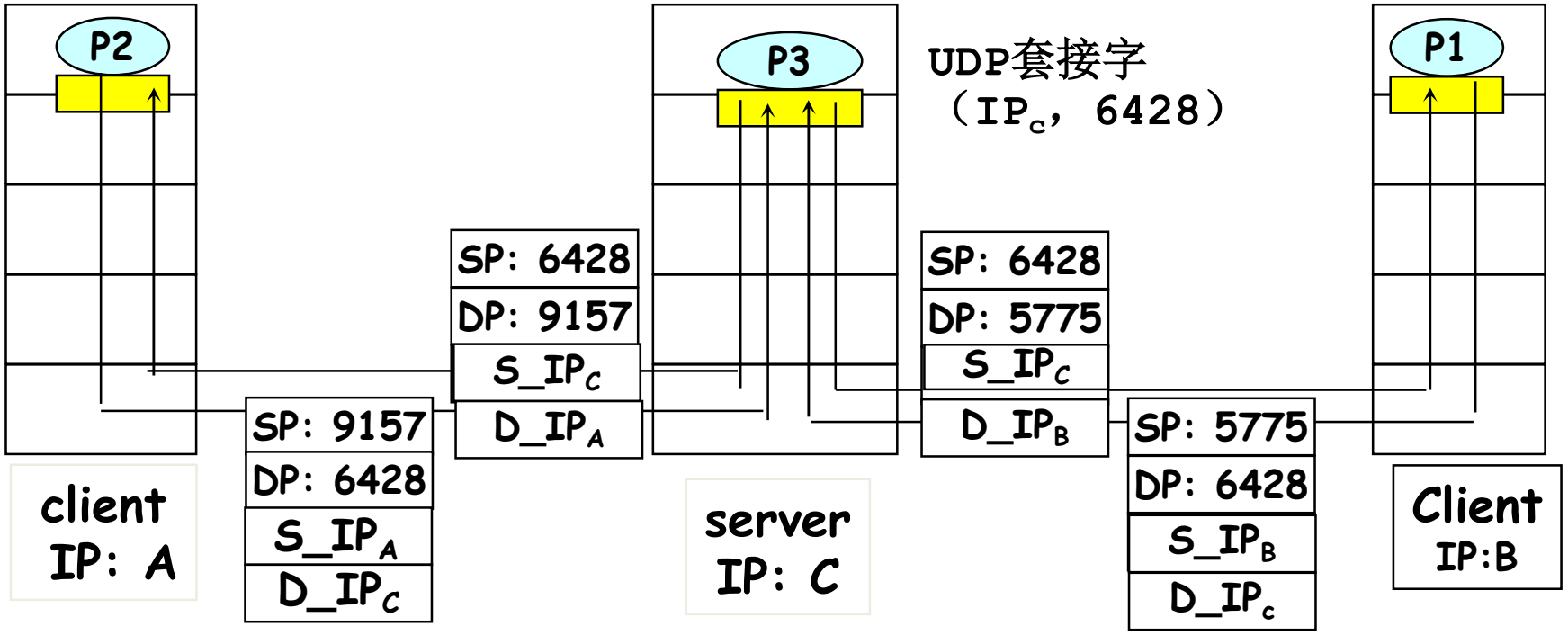
- UDP 套接字由两个因素指定:

(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 (续)

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

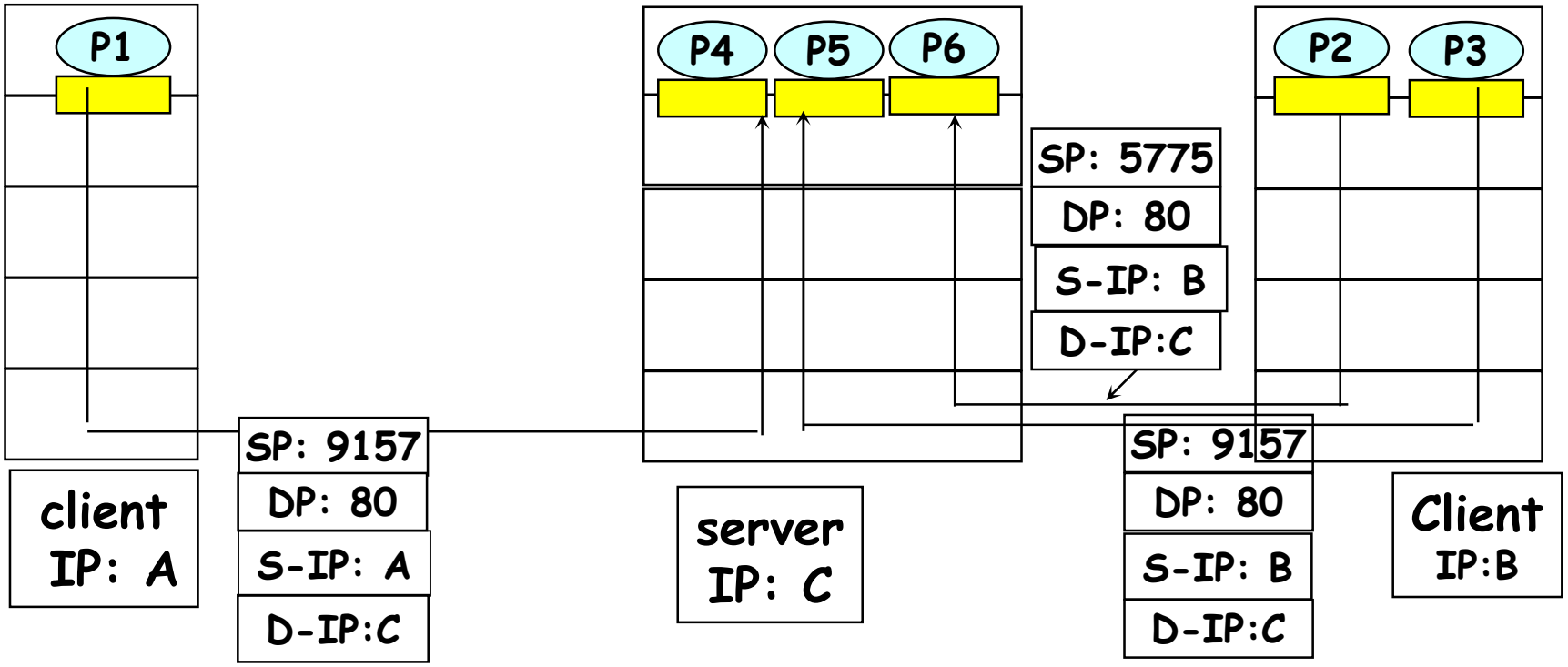


请求报文段中提供返回地址（包括IP地址和端口号）

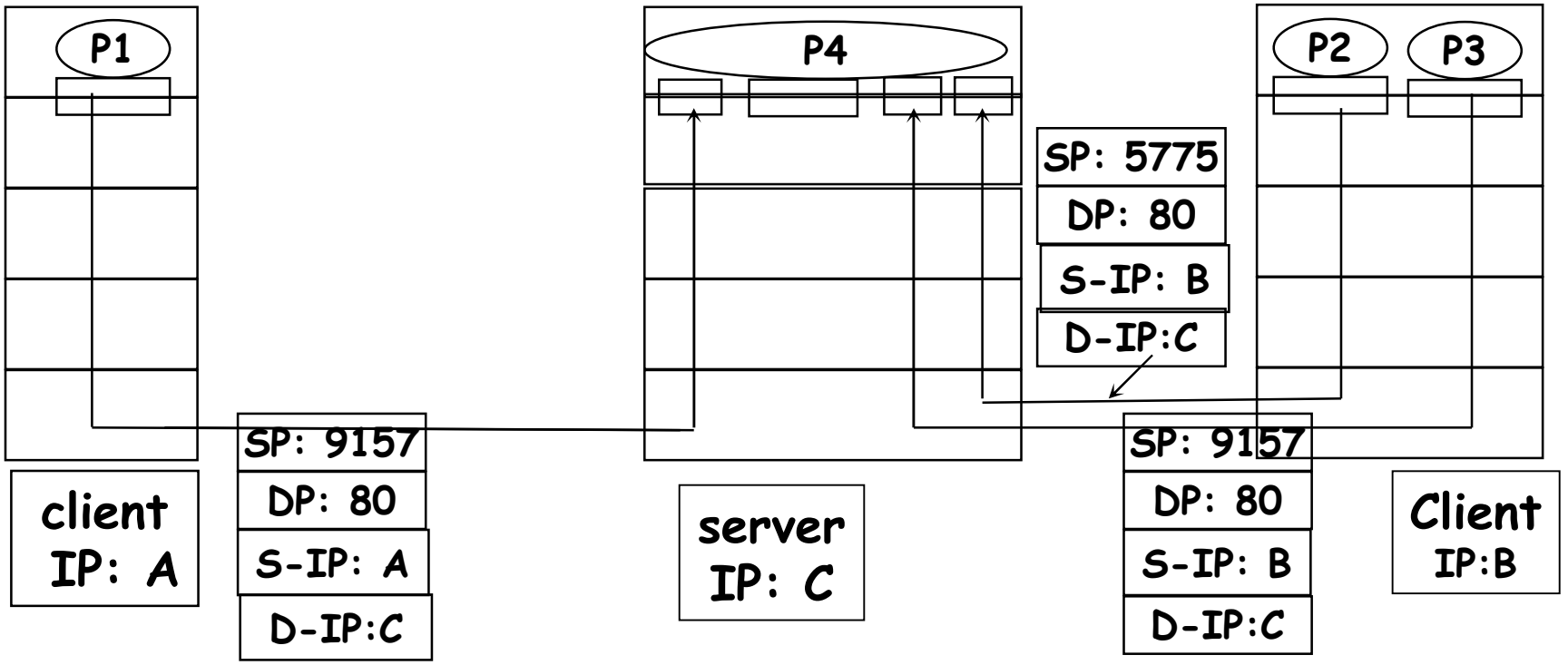
面向连接的多路分解

- 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
- 服务器主机支持很多同时的 TCP 套接字:
  - 每个套接字用4部分来表示
- Web服务器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 (续)


P1
P4
P5
P6
P2
P3
SP: 5775
DP: 80
S-IP: B
D-IP:C
SP: 9157
DP: 80
S-IP: A
D-IP:C
SP: 9157
DP: 80
S-IP: B
D-IP:C
client
IP: A
server
IP: C
Client
IP:B

面向连接的多路分解: web服务器如何工作?



第三章 提纲

- 3.1 传输层服务
- 3.2多路复用和多路复用
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
- 3.5 面向连接传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

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, UDP 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?

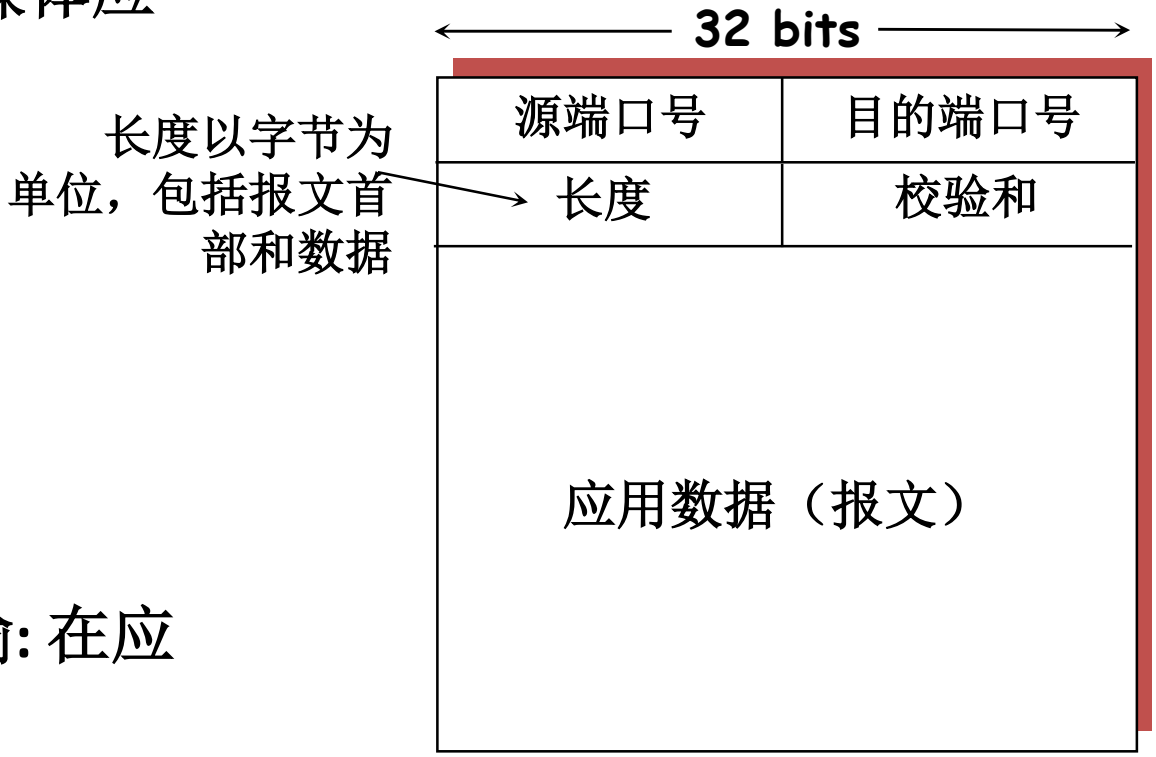
- 不需要建立连接 (减少延迟)
- 简单: 在发送者接收者之间不需要连接状态
- 很小的数据段首部,8字节
- 没有拥塞控制: UDP 能够用想象的快的速度传递

因特网应用：应用层协议，传输协议

应用	应用层协议	下面的传输协议
电子邮件	SMTP [RFC 2821]	TCP
远程终端访问	Telnet [RFC 854]	TCP
Web	HTTP [RFC 2616]	TCP
文件传输	FTP [RFC 959]	TCP
流媒体	通常专用 (e.g. RealNetworks)	TCP or UDP
因特网电话	通常专用 (e.g., Skype)	典型用 UDP
网络管理	SNMP	UDP
路由选择	RIP	UDP

UDP:用户数据报协议(RFC768)

- 经常用于流式多媒体应用
  - 容忍丢失
  - 速率敏感
- 其它 UDP 应用
  - DNS
  - SNMP
- UDP上的可靠传输: 在应用层增加可靠性
  - 应用特有的错误恢复!



UDP 数据报格式

UDP 校验和

目标: 检测传输的数据段的 “错误” 如bit丢失

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:

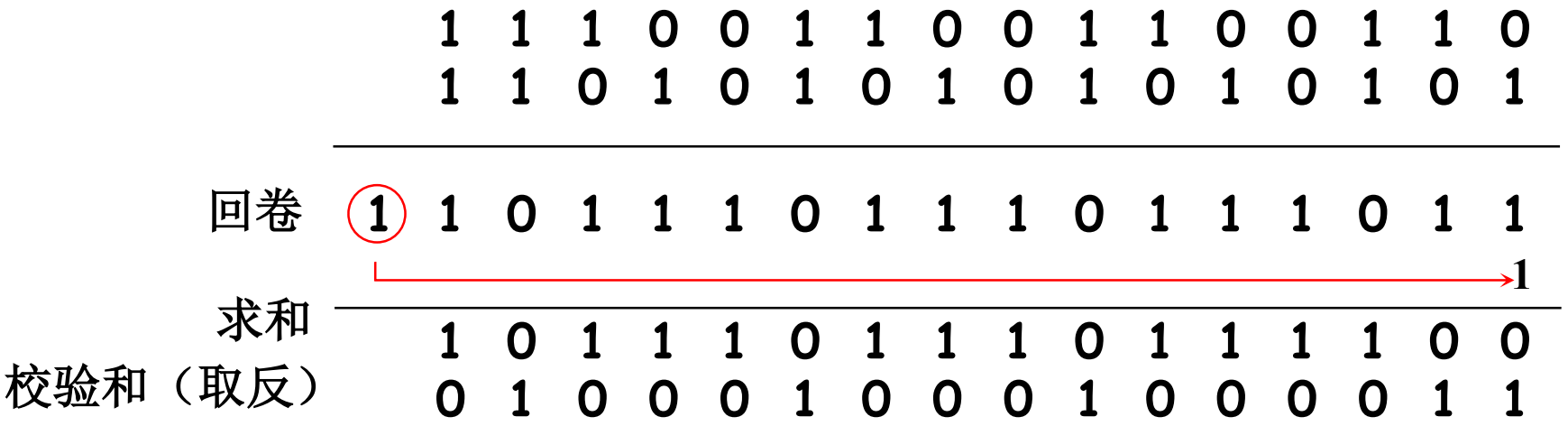
- 计算接收到数据段的校验和
- 检查 计算的校验和是否等于1111111111111111:
  - NO – 检测到错误
  - YES –没有检测到错误

= =没有错误？？

. 实际上可能是错误的

Internet 校验和例子

- **Note**
  - 在加数字的时候，从最高位溢出的bit必须要加到结果上
- 例:加两个16位整数

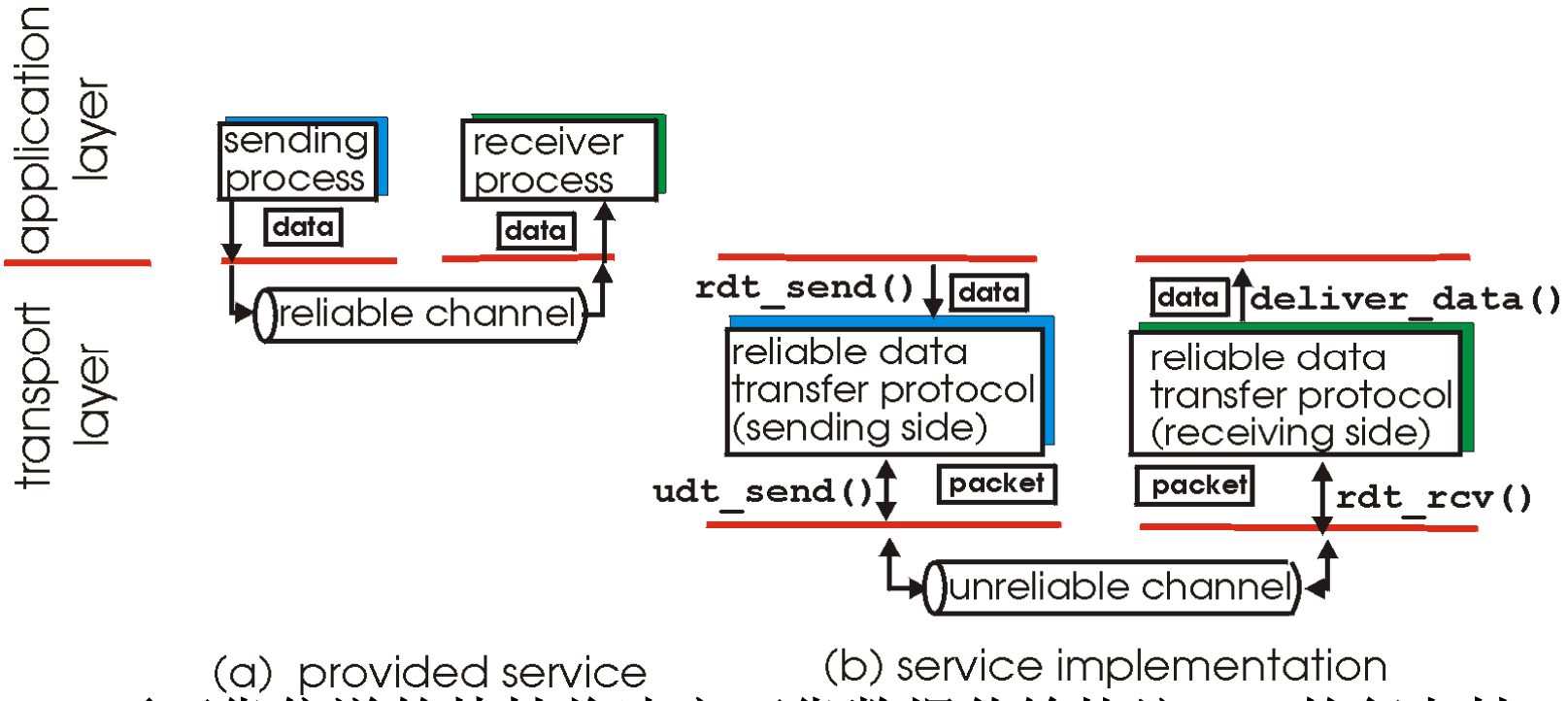


第三章 提纲

- 3.1 传输层服务
- 3.2多路复用和多路分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
- 3.5 面向连接传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

可靠数据传输原理

- 在应用层、传输层、链路层都有所应用

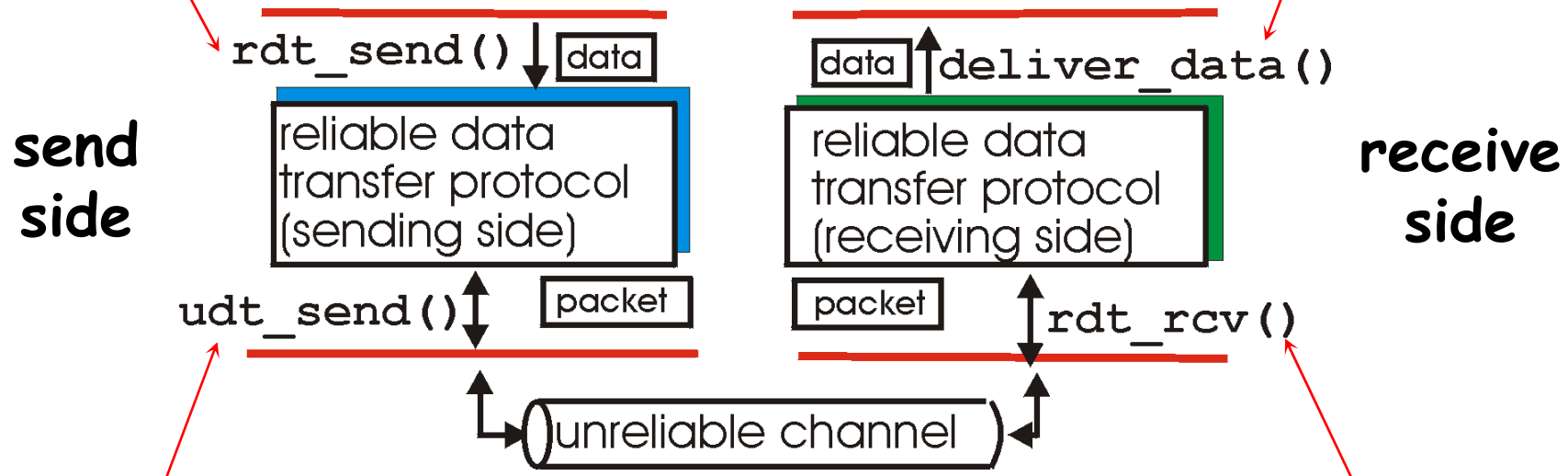


- 不可靠信道的特性将决定可靠数据传输协议**(rdt)**的复杂性

可靠数据传输:

rdt_send()：上层应用调用该函数要求可靠传输数据到接收者

deliver_data()：可靠传输调用该函数将数据传给上层应用



udt_send()：可靠传输调用该函数，在不可靠的信道上传输数据给接收者

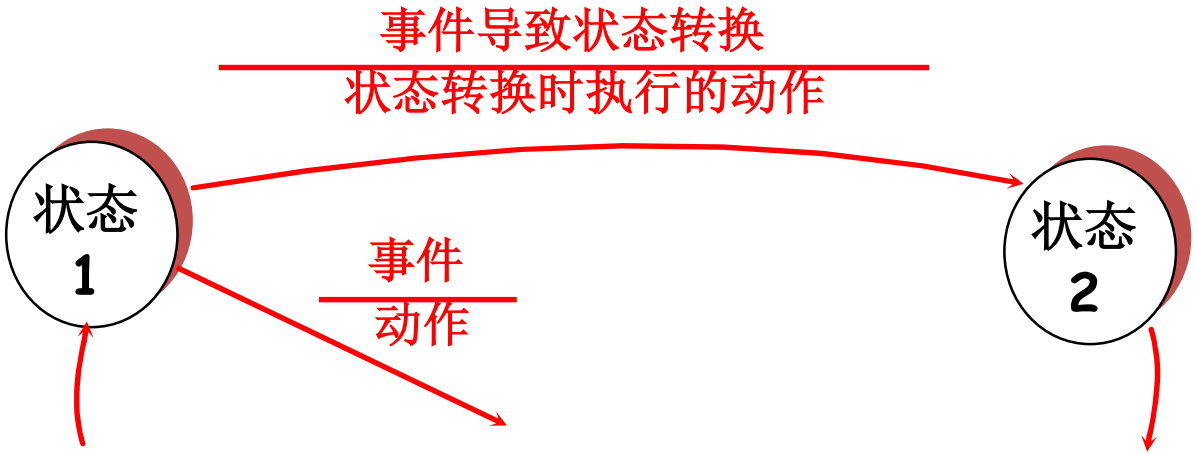
rdt_rcv()：接收方接收到数据后调用该函数完成可靠传输

可靠数据传输:

我们将

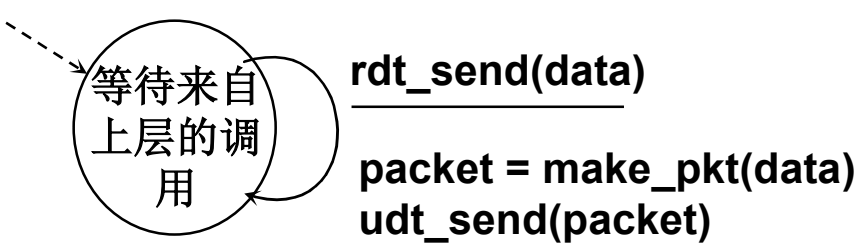
- 逐步开发发送方和接收方的可靠数据传输协议 (rdt)
- 仅考虑单向数据传输
  - 但控制信息将双向流动!
- 用有限状态机 (FSM) 来标示发送方和接收方

状态: 在一个状态时，由事件唯一的确定状态的转换

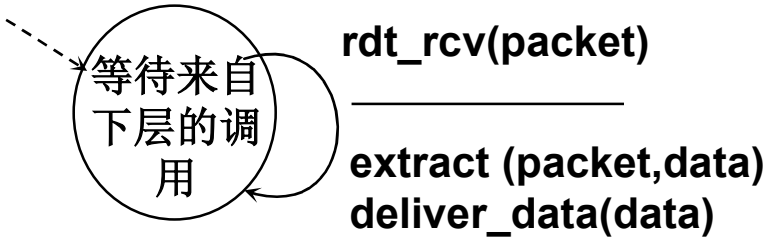


Rdt1.0: 完全可靠信道上的可靠数据传输

- 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



sender



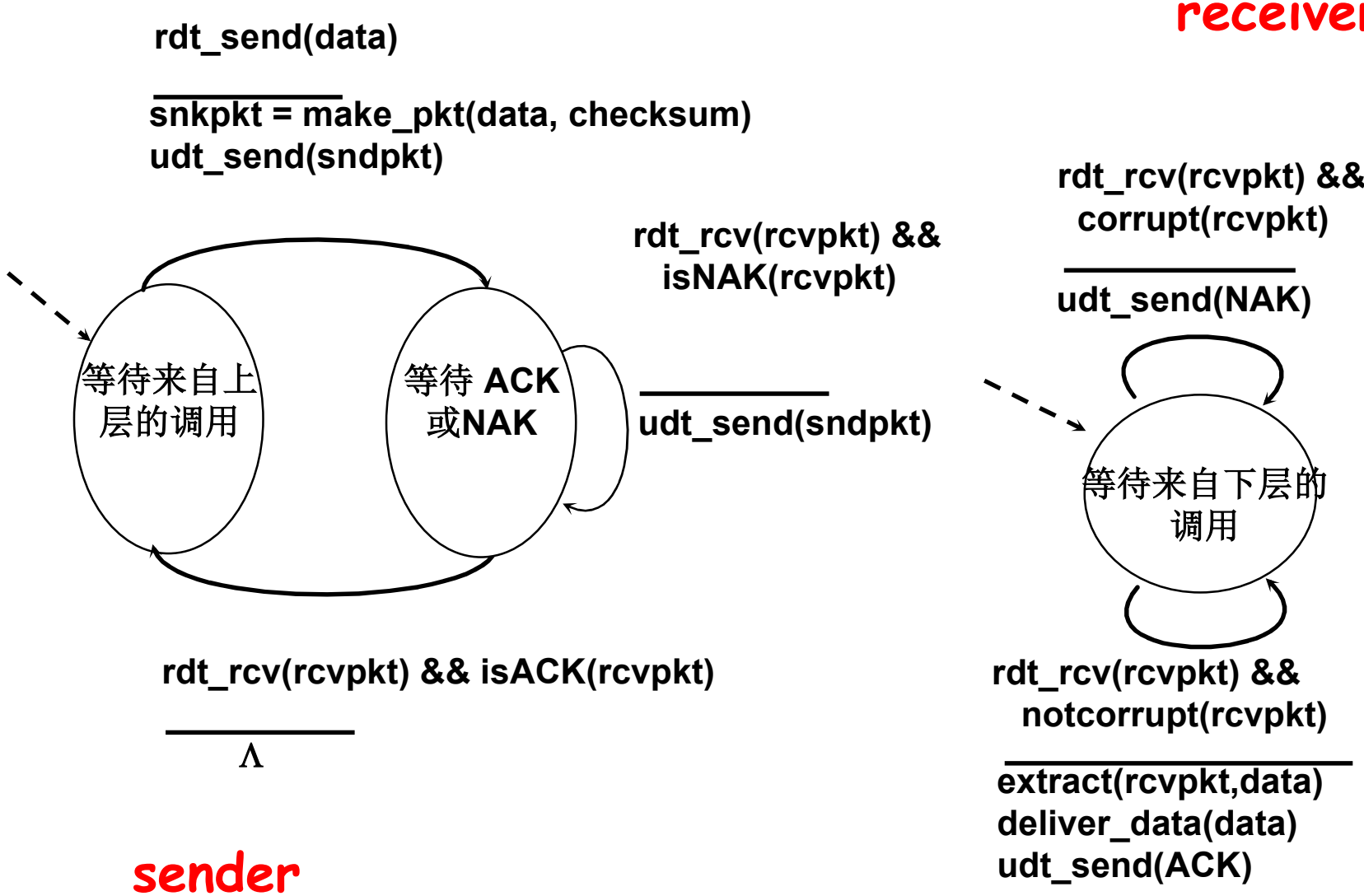
receiver

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`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
  - 重传
- ARQ协议

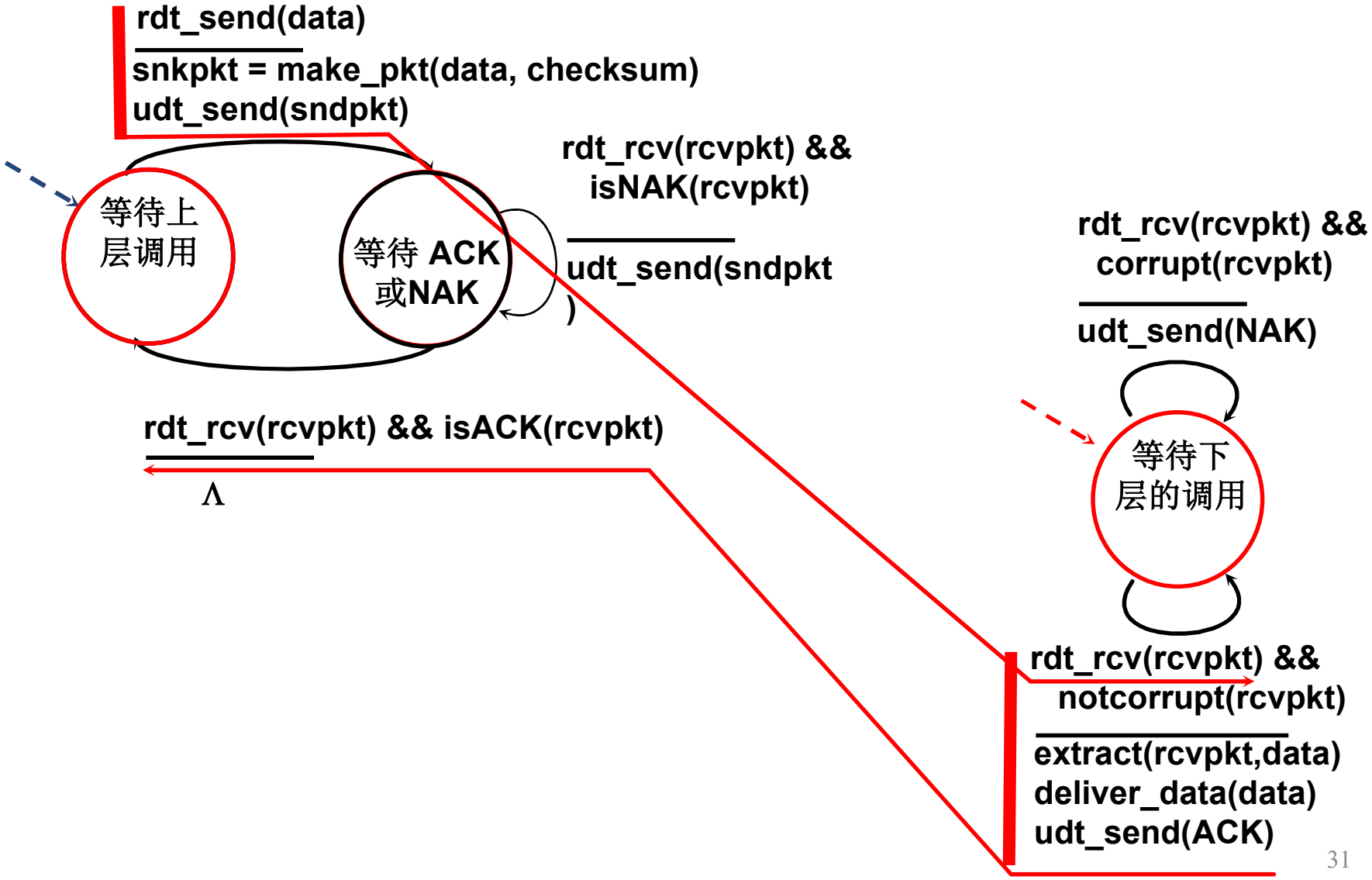
rdt2.0: FSM 规范

receiver

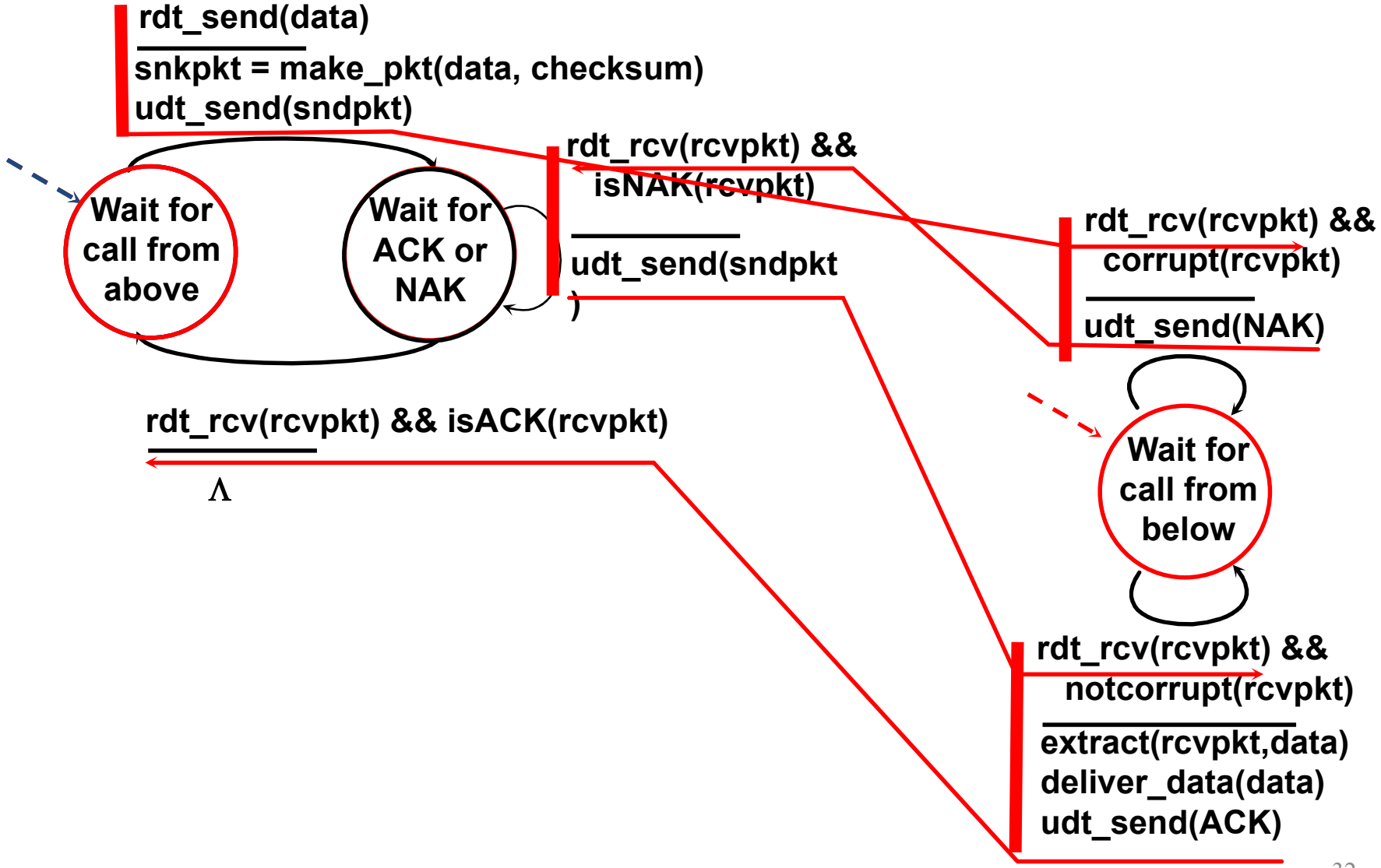


sender

rdt2.0: 没有错误时的操作



rdt2.0: 错误场景


rdt_send(data)
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)
Wait for call from above
Wait for ACK or NAK
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)
Wait for call from below
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

停一等协议

发送方发送一个报文，然后等待接收方的响应

rdt2.0 有一个致命缺陷!

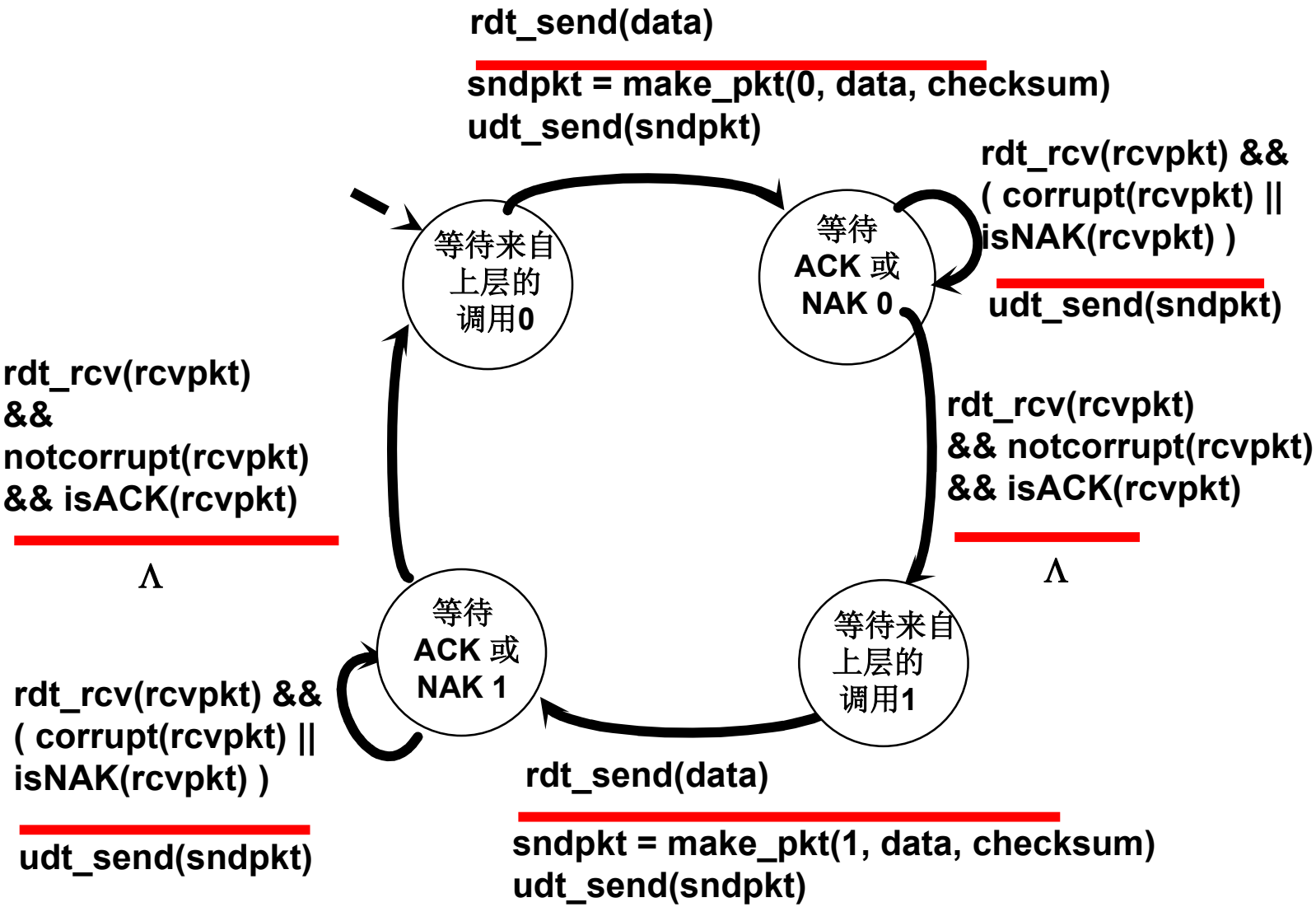
如果ACK/NAK混淆了会发生什么？

- 1.ACK或NAK传输错误。发送方并不知道接收方发生了什么!
- 2.增加足够的检验和比特，检查错误，纠错
- 3.重传，不能正确重发:可能重复

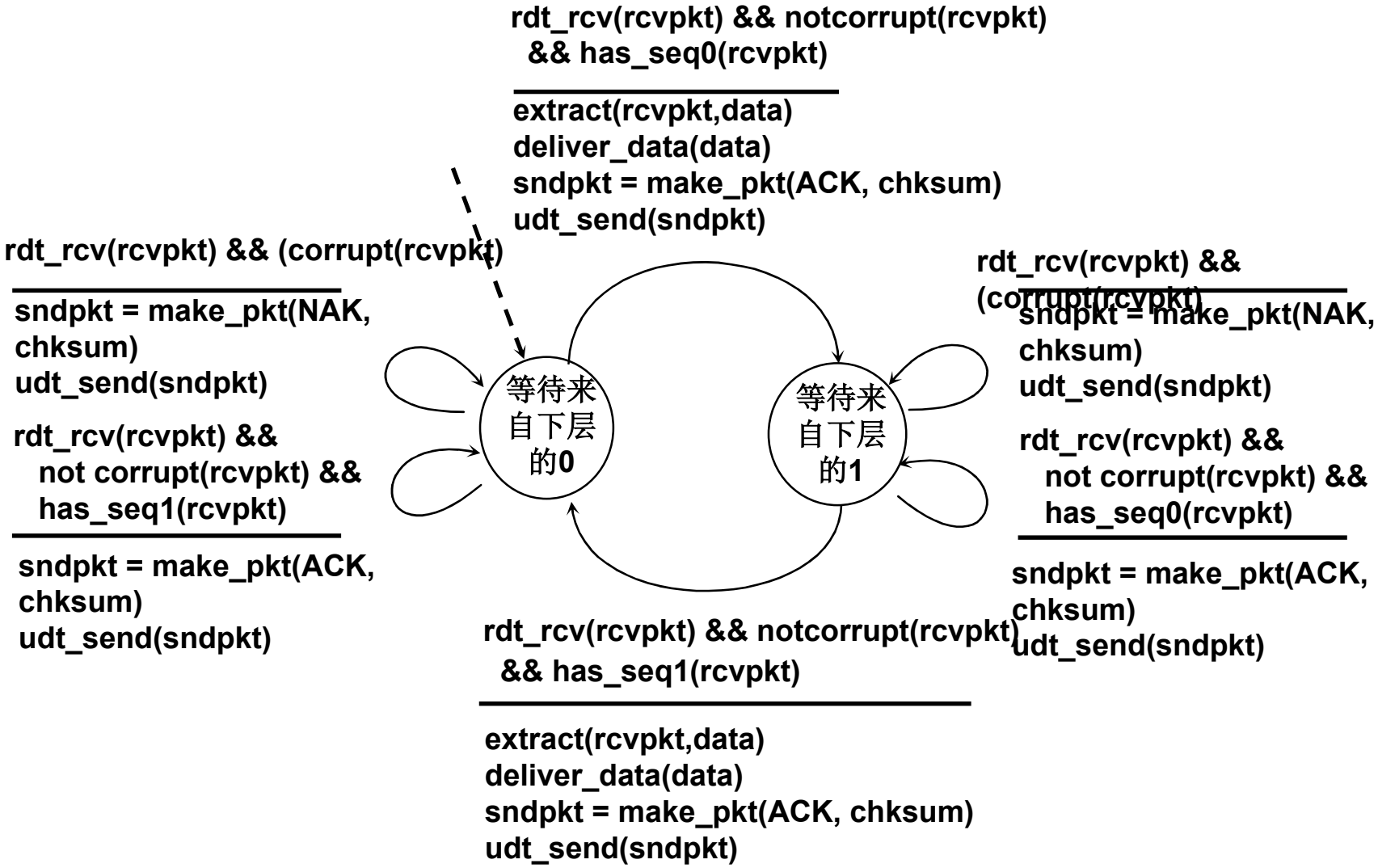
处理重复:

- 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）

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. 为什么?
- 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报文是1号还是0号

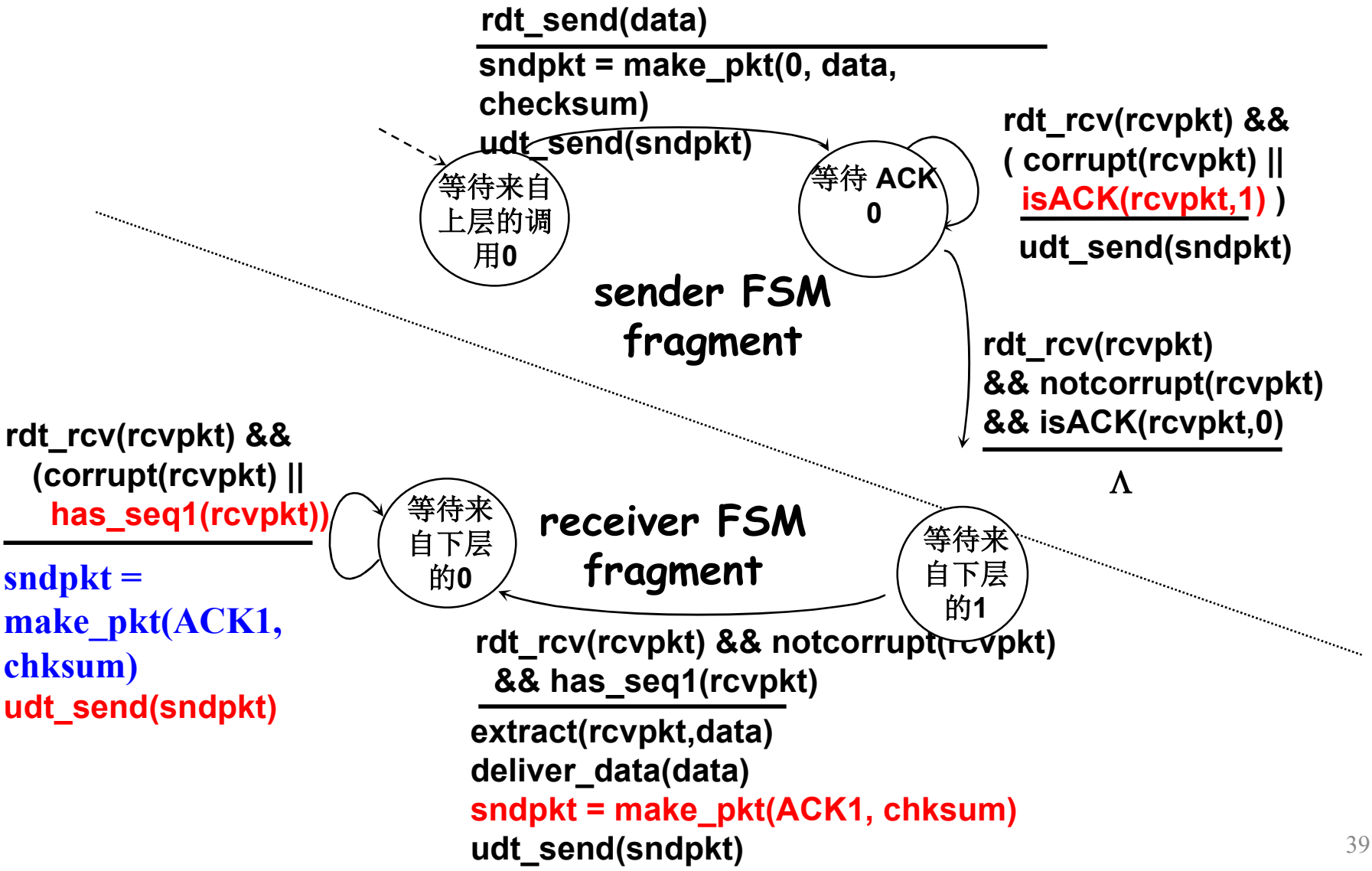
接收方:

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

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

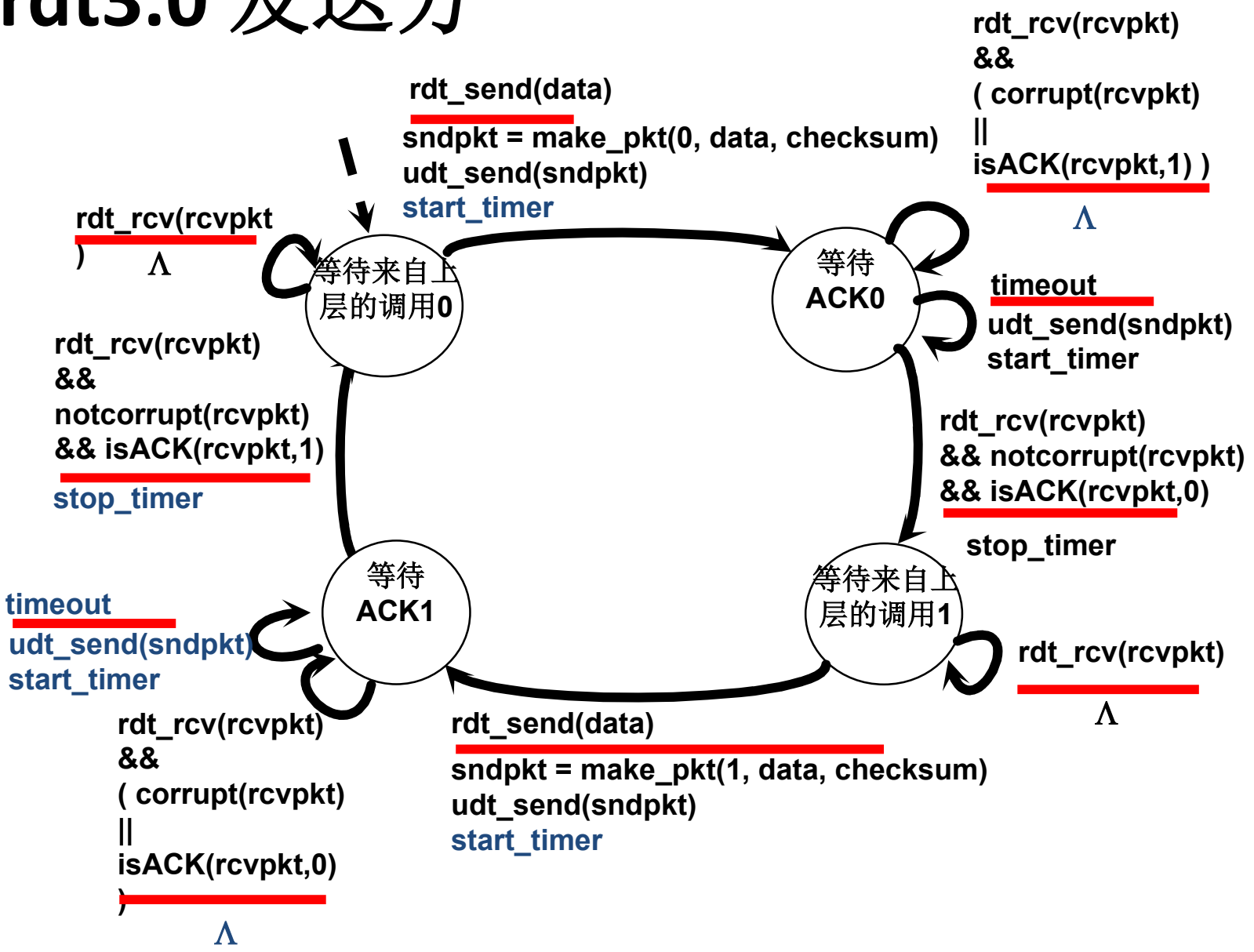
新假设: 下层信道还要丢失报文 (数据或者 ACKs)

- 校验和, 序号, 确认, 重发将会有帮助，但是不够

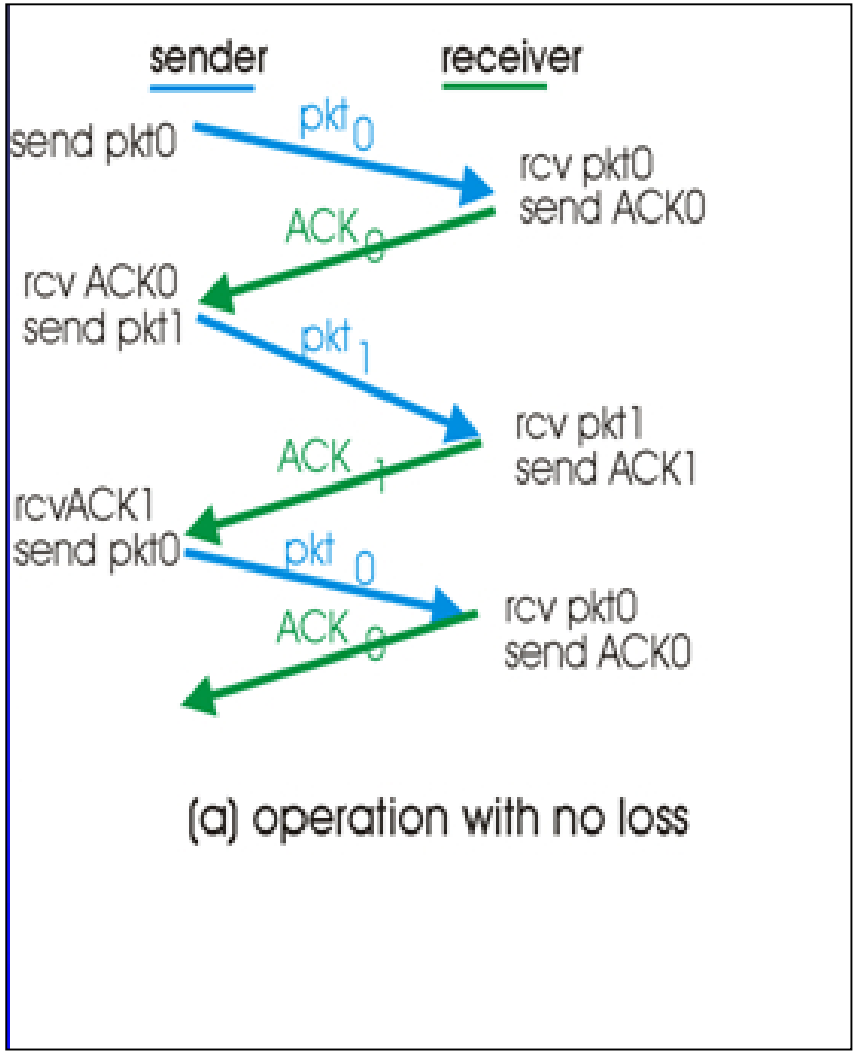
方法: 发送者等待“合理的”确认时间（超时重传）

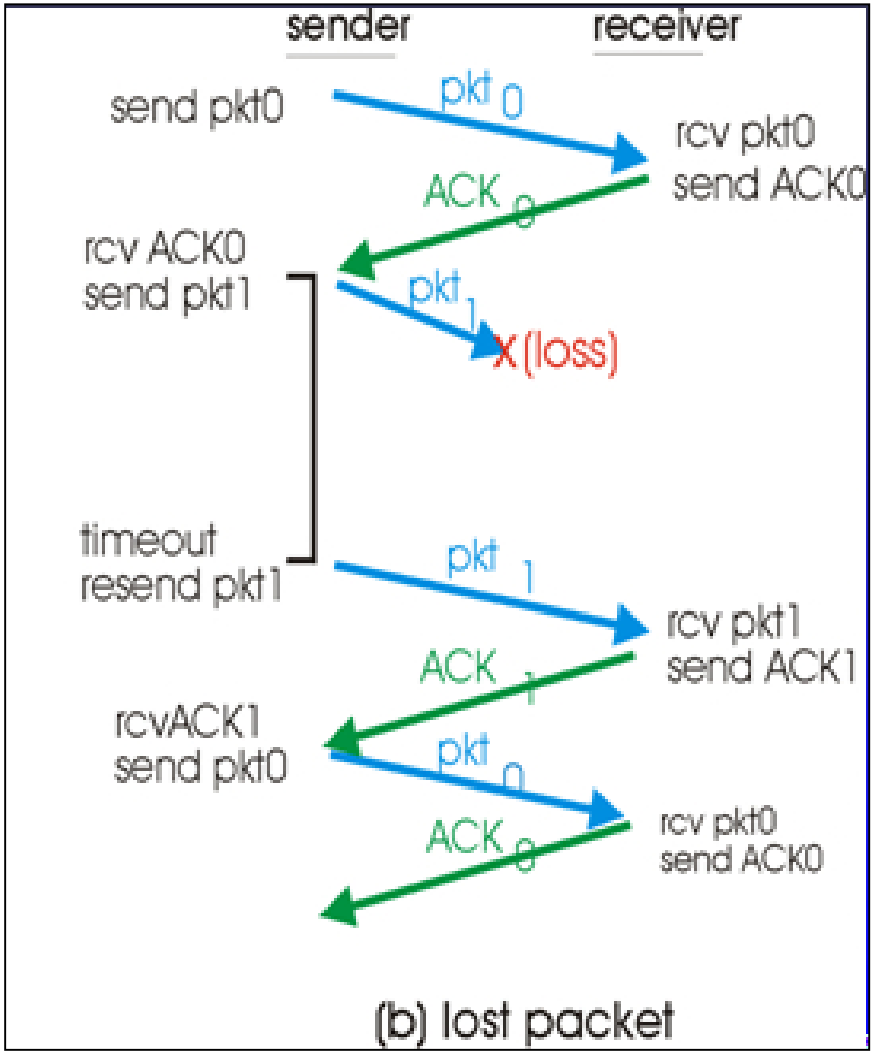
- 如果在这个时间内没有收到确认就重发
- 如果报文（或者确认）只是延迟 (没有丢失):
  - 重发将导致重复，但是使用序号已经处理了这个问题
  - 接收方必须指定被确认的报文序号
- 要求倒计时定时器

rdt3.0 发送方

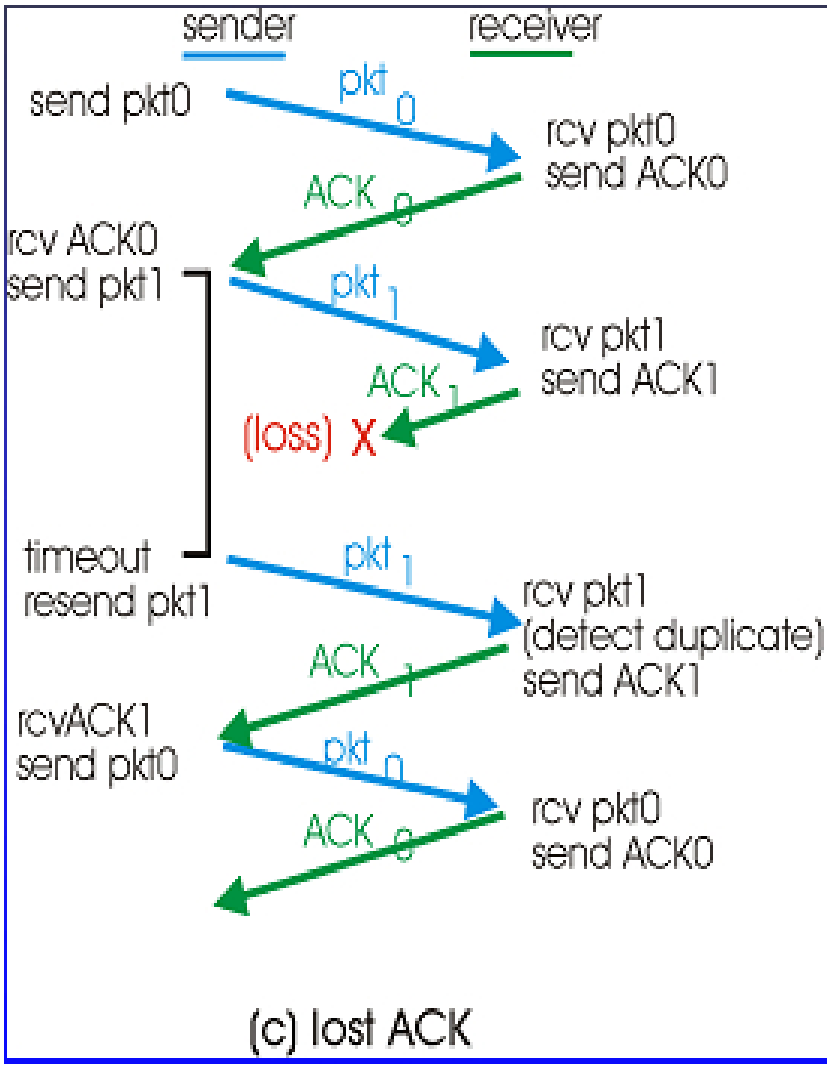

rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt
)
Λ
等待来自上层的调用0
rdt_rcv(rcvpkt)
&&
notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
stop_timer
等待
ACK1
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
&&
(corrupt(rcvpkt)
||
isACK(rcvpkt,1))
Λ
等待
ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
stop_timer
等待来自上层的调用1
rdt_rcv(rcvpkt)
Λ
rdt_rcv(rcvpkt)
&&
(corrupt(rcvpkt)
||
isACK(rcvpkt,0)
)
Λ
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

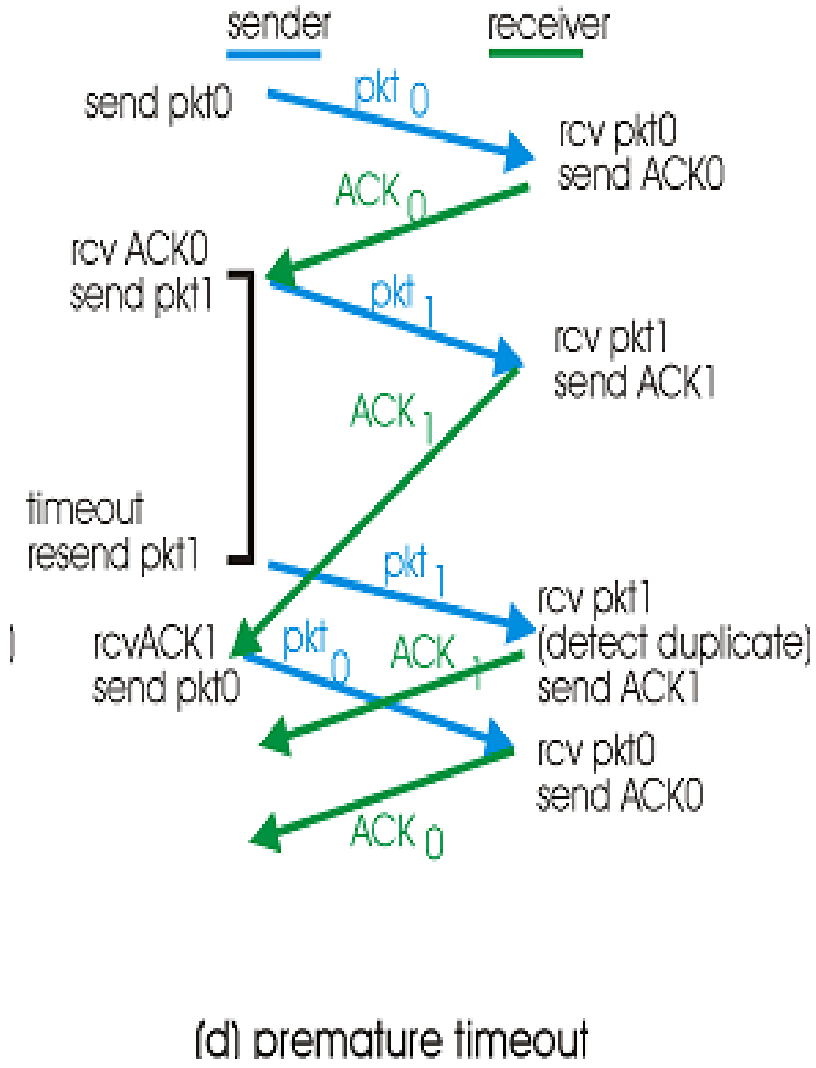
rdt3.0 操作


sender
receiver
send pkt0
pkt 0
rcv pkt0
send ACK0
ACK 0
rcv ACK0
send pkt1
pkt 1
rcv pkt1
send ACK1
ACK 1
rcvACK1
send pkt0
pkt 0
rcv pkt0
send ACK0
ACK 0
(a) operation with no loss


sender
receiver
send pkt0
pkt 0
rcv pkt0
send ACK0
ACK 0
rcv ACK0
send pkt1
pkt 1
X (loss)
timeout
resend pkt1
pkt 1
rcv pkt1
send ACK1
ACK 1
rcvACK1
send pkt0
pkt 0
rcv pkt0
send ACK0
ACK 0
(b) lost packet

rdt3.0 操作


sender
receiver
send pkt0
pkt 0
rcv pkt0
send ACK0
ACK 0
rcv ACK0
send pkt1
pkt 1
rcv pkt1
send ACK1
ACK 1
(loss) X
timeout
resend pkt1
pkt 1
rcv pkt1
(detect duplicate)
send ACK1
ACK 1
rcvACK1
send pkt0
pkt 0
rcv pkt0
send ACK0
ACK 0
(c) lost ACK


sender
receiver
send pkt0
pkt 0
rcv pkt0
send ACK0
ACK 0
rcv ACK0
send pkt1
pkt 1
rcv pkt1
send ACK1
ACK 1
timeout
resend pkt1
pkt 1
rcv pkt1
(detect duplicate)
send ACK1
rcvACK1
send pkt0
pkt 0
ACK 1
rcv pkt0
send ACK0
ACK 0
(d) premature timeout

rdt3.0的性能

- rdt3.0 工作但是性能很差
- 例如: 1 Gbps链路, 15 ms 端到端传输延迟, 1KB 报文:

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

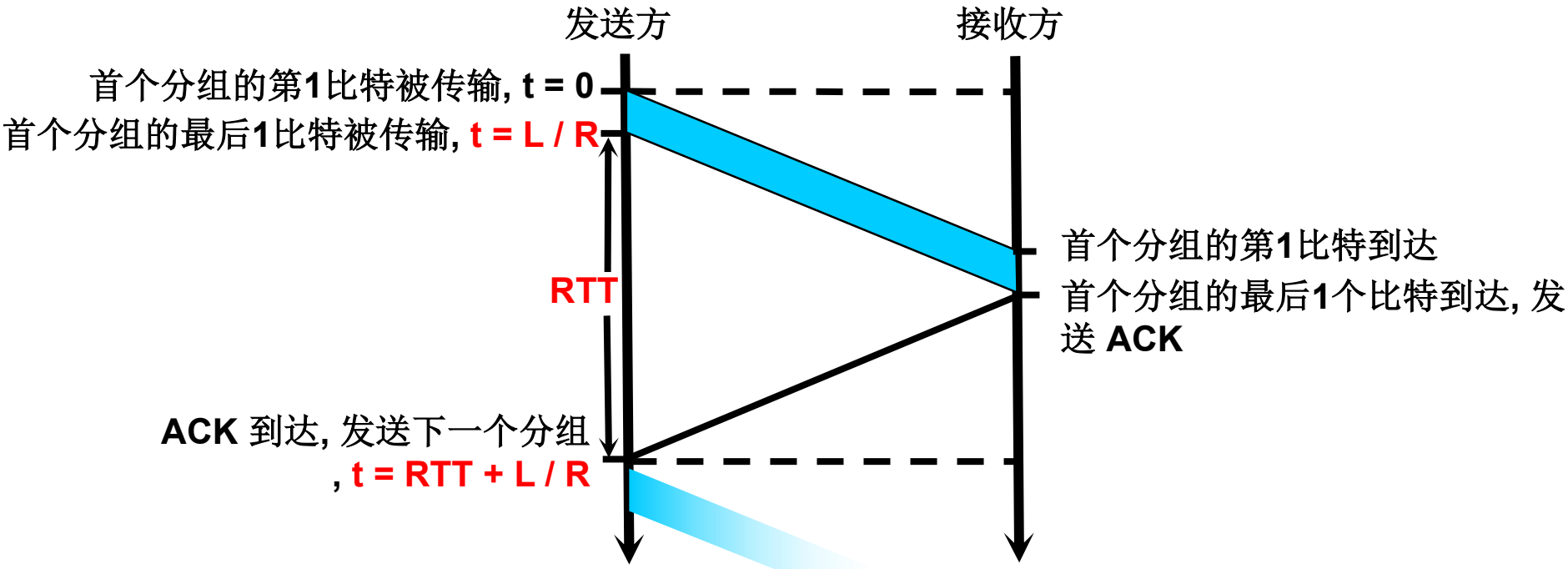
U sender: 利用率 – 发送方忙于发送的时间部分

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

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

网络协议限制了物质资源的使用!

rdt3.0: 停等操作

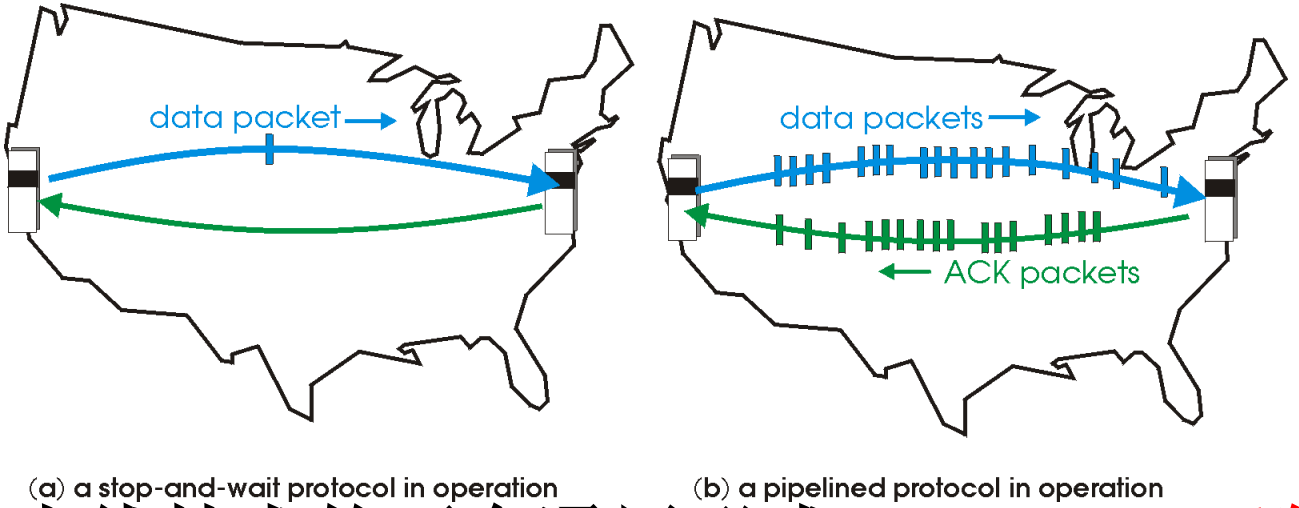


$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

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

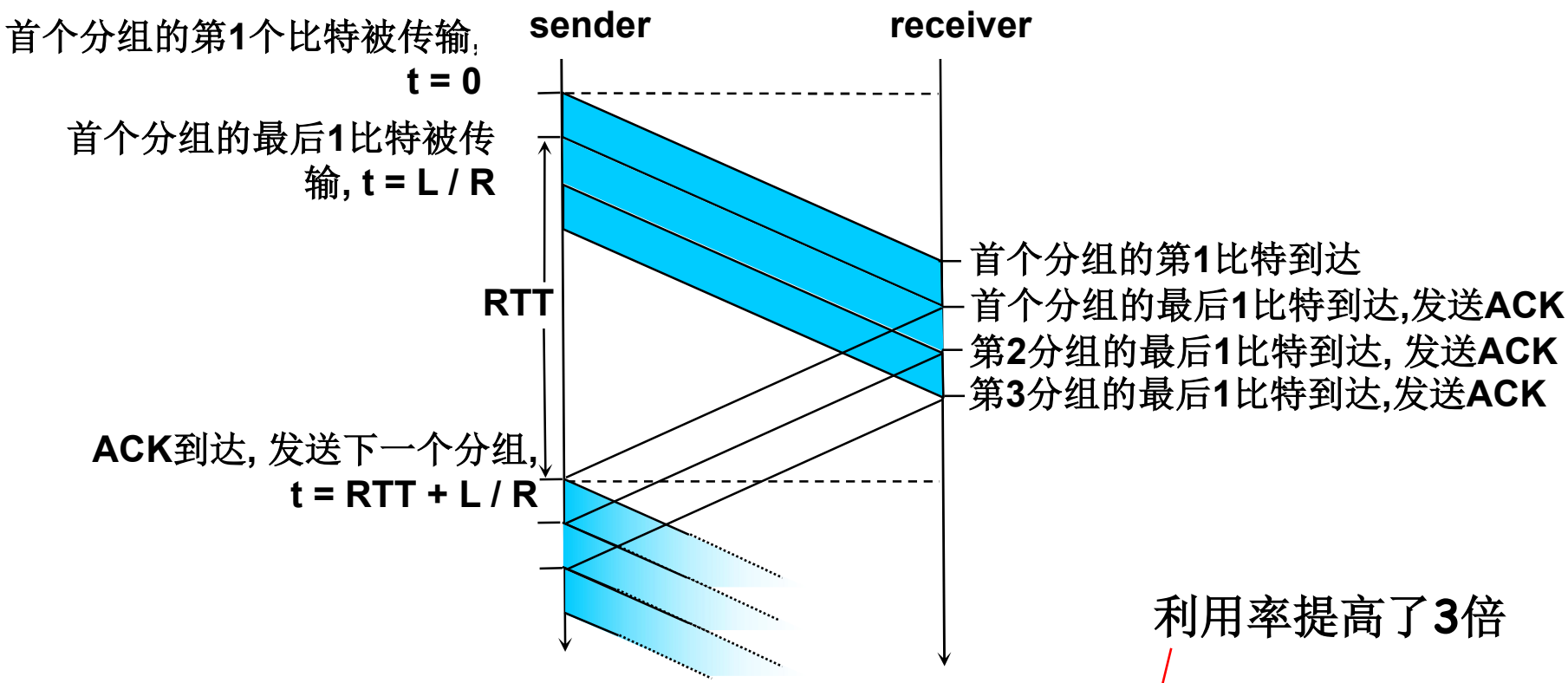
- 序号数目的范围必须增加
- 在发送方/接收方必须有缓冲区



(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



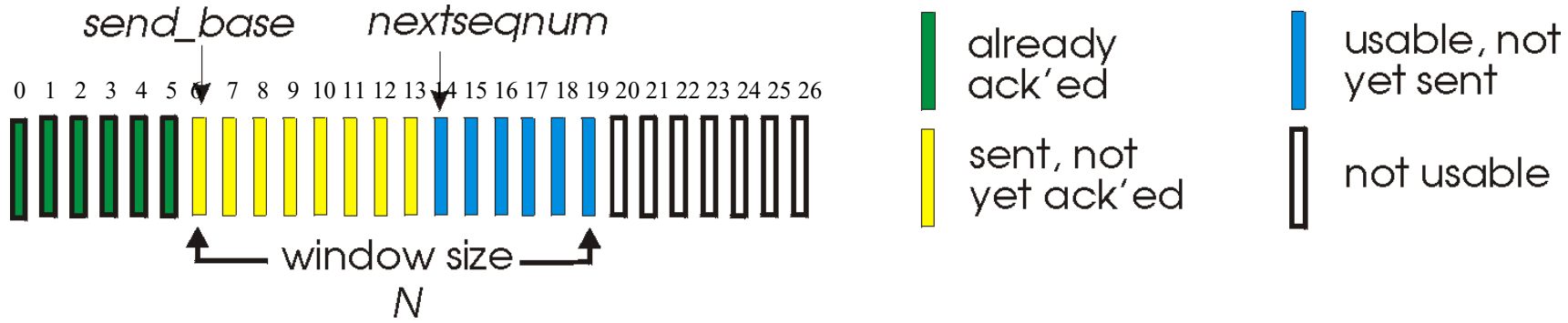
利用率提高了3倍

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

Go-Back-N

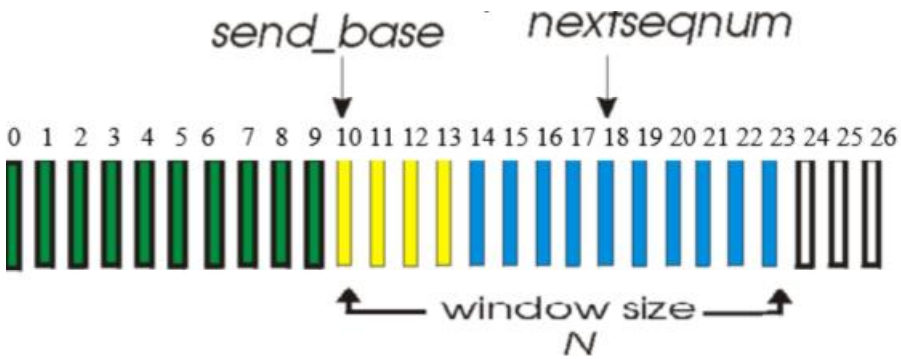
发送方:

- 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



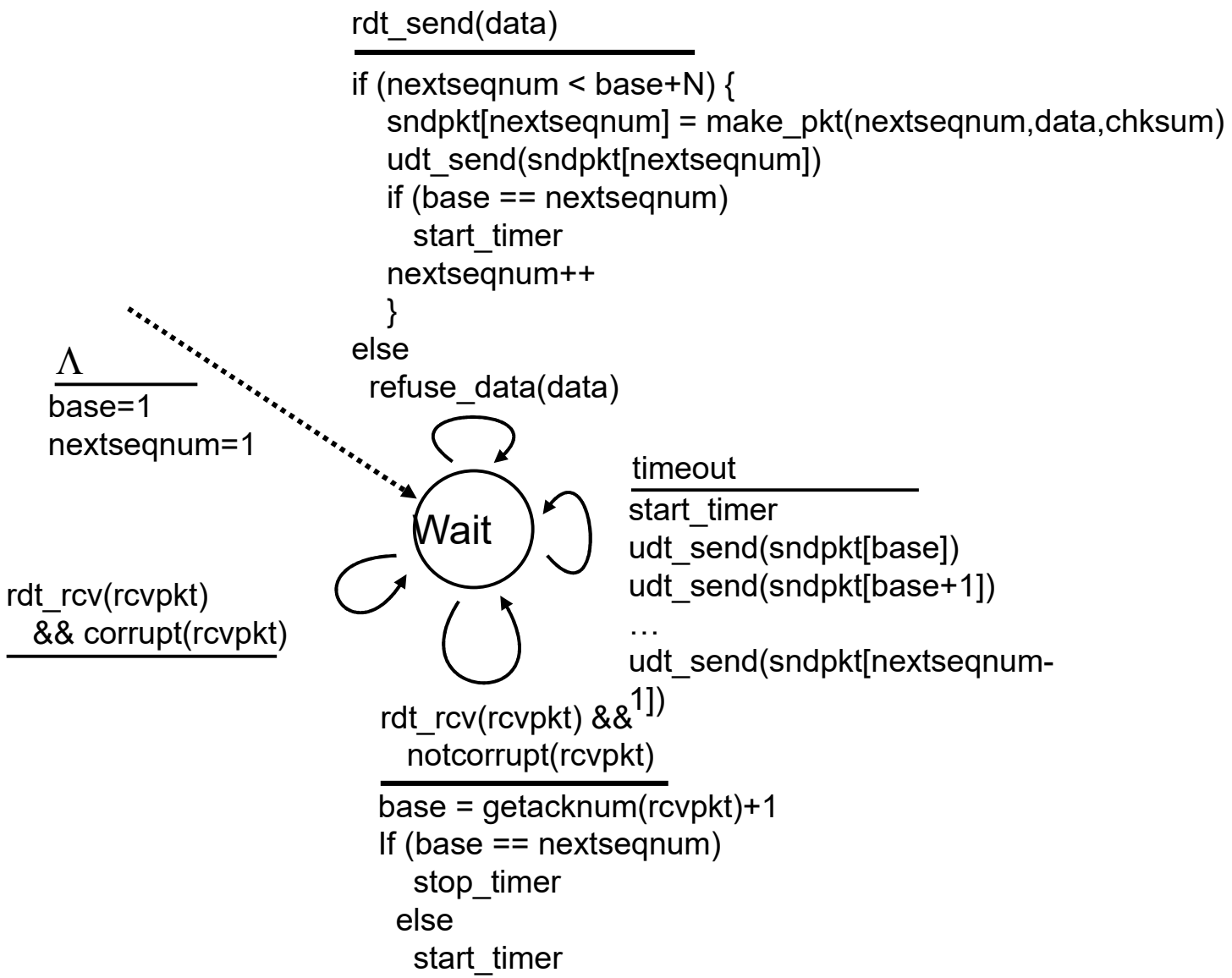
ACK(n): 确认所有的报文直到（包含）序号n - “累积ACK”

若收到*ACK*（9)

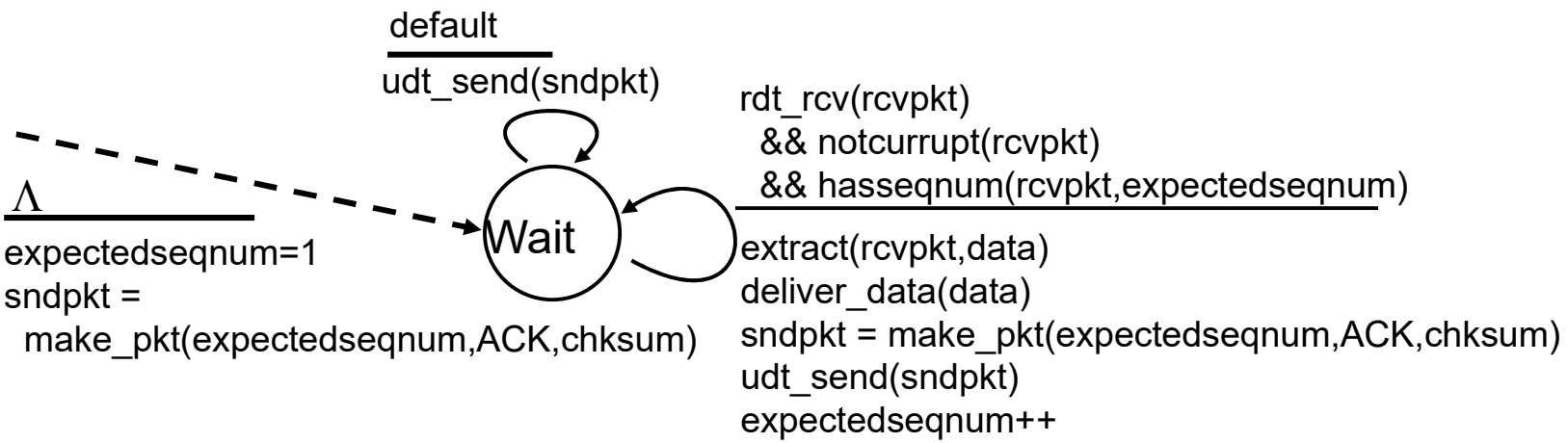


超时(n): 重发窗口中的报文n及以上更高序号的报文(只有一个定时器记录最早的未被确认报文的发送时间)

GBN: 发送方FSM


rdt_send(data)
if (nextseqnum < base+N) {
sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
udt_send(sndpkt[nextseqnum])
if (base == nextseqnum)
start_timer
nextseqnum++
}
else
refuse_data(data)
Λ
base=1
nextseqnum=1
Wait
timeout
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)
rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
stop_timer
else
start_timer

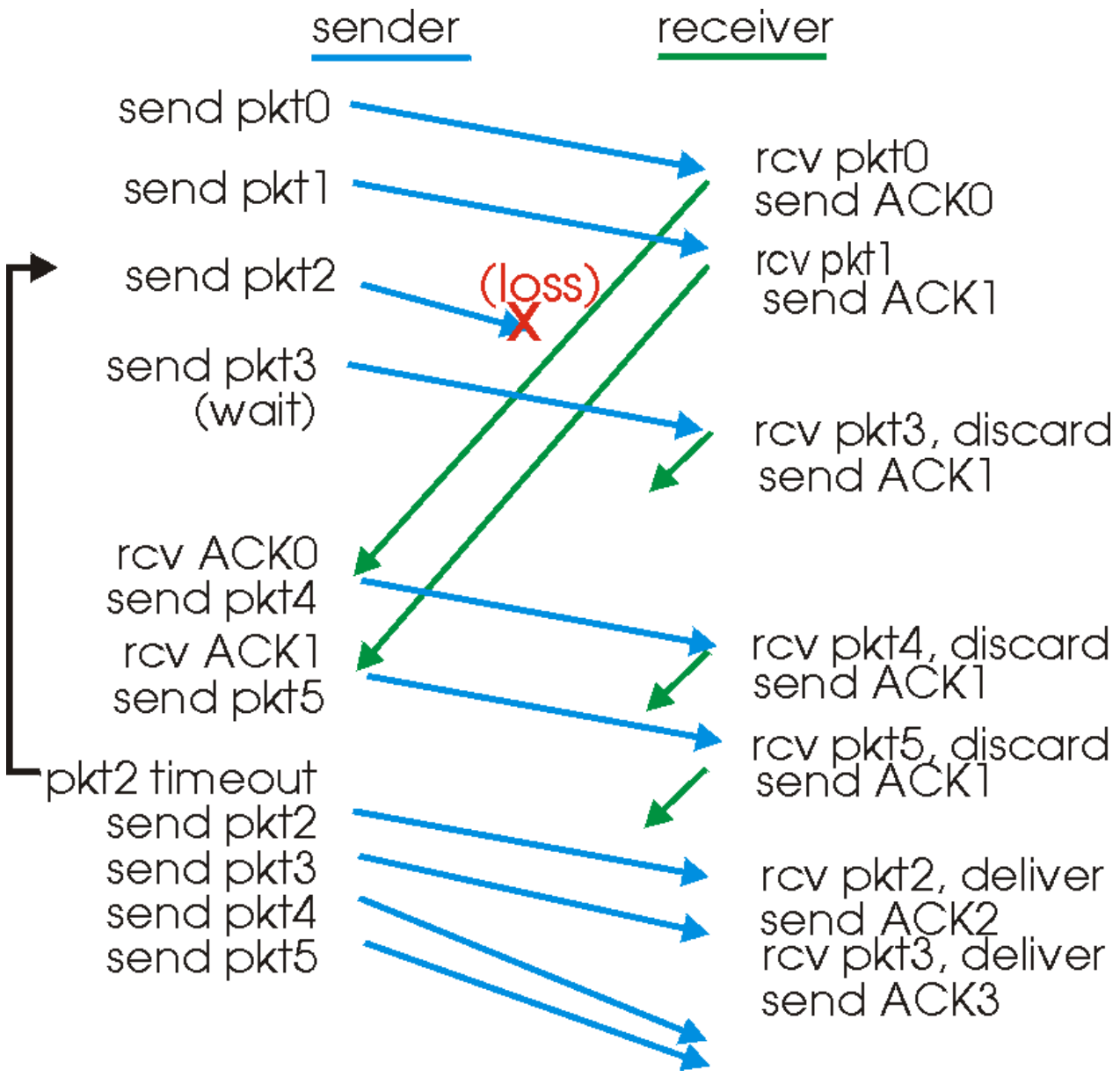
GBN: 接收方FSM



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`**

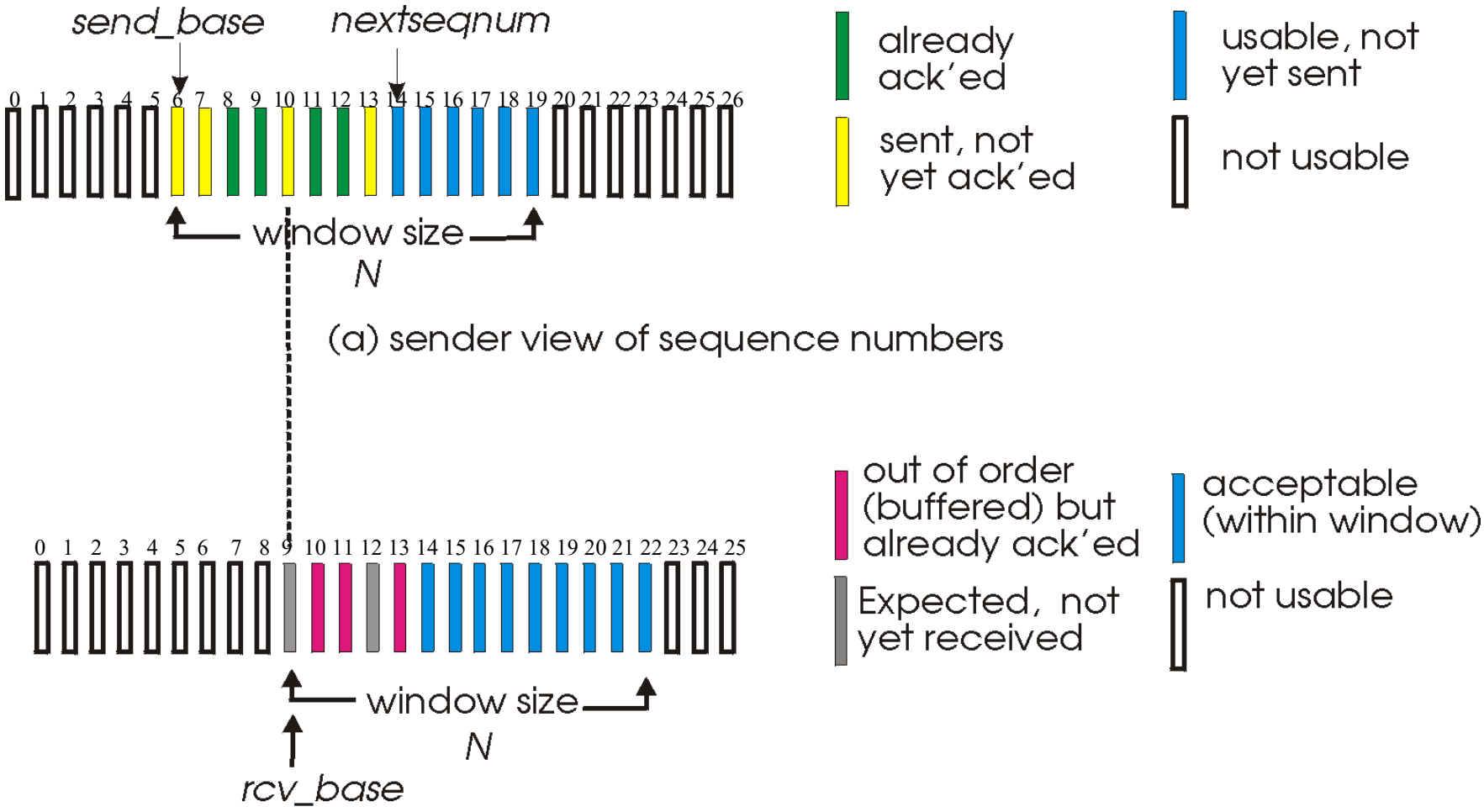
- 接收到失序分组:
  - 丢弃(不缓冲) -> 没有接收缓冲区!
  - 重发正确接收的最高序号分组的**ACK**


sender
receiver
send pkt0
send pkt1
send pkt2
(loss)
send pkt3
(wait)
rcv ACK0
send pkt4
rcv ACK1
send pkt5
pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt3, discard
send ACK1
rcv pkt4, discard
send ACK1
rcv pkt5, discard
send ACK1
rcv pkt2, deliver
send ACK2
rcv pkt3, deliver
send ACK3

选择性重传

- 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
- 接收窗口
  - N‘ 个连续序号
  - 可接收缓存的分组数量

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

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), 滑动窗口到下一个没有接收的分组 not-yet-received pkt

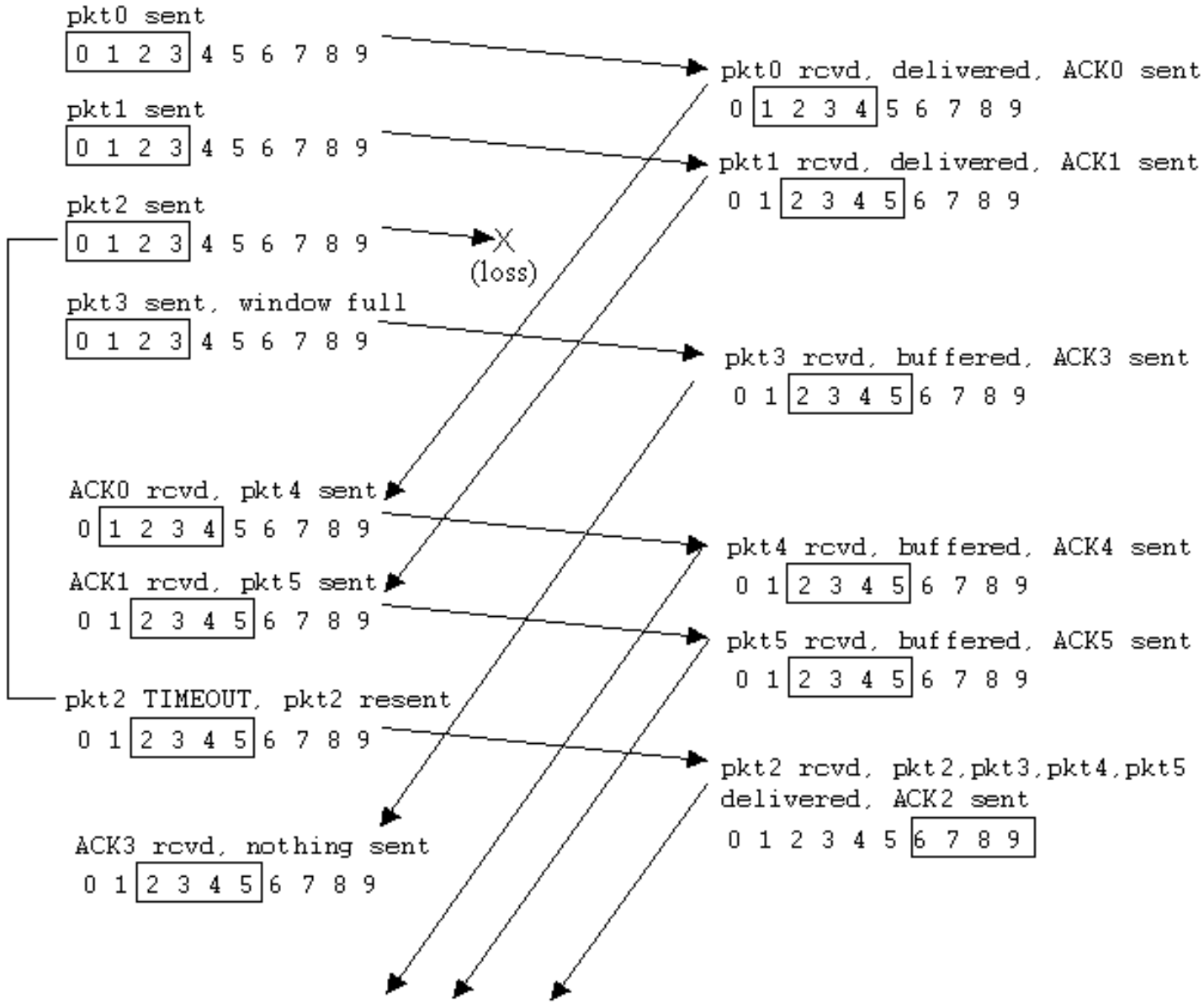
分组n在[rcvbase-N,rcvbase-1]内

发送ACK(n)

其他:

忽略

选择性重传的操作


pkt0 sent
0 1 2 3 4 5 6 7 8 9
pkt1 sent
0 1 2 3 4 5 6 7 8 9
pkt2 sent
0 1 2 3 4 5 6 7 8 9
(loss)
pkt3 sent, window full
0 1 2 3 4 5 6 7 8 9
ACK0 rcvd, pkt4 sent
0 1 2 3 4 5 6 7 8 9
ACK1 rcvd, pkt5 sent
0 1 2 3 4 5 6 7 8 9
pkt2 TIMEOUT, pkt2 resent
0 1 2 3 4 5 6 7 8 9
ACK3 rcvd, nothing sent
0 1 2 3 4 5 6 7 8 9
pkt0 rcvd, delivered, ACK0 sent
0 1 2 3 4 5 6 7 8 9
pkt1 rcvd, delivered, ACK1 sent
0 1 2 3 4 5 6 7 8 9
pkt3 rcvd, buffered, ACK3 sent
0 1 2 3 4 5 6 7 8 9
pkt4 rcvd, buffered, ACK4 sent
0 1 2 3 4 5 6 7 8 9
pkt5 rcvd, buffered, ACK5 sent
0 1 2 3 4 5 6 7 8 9
pkt2 rcvd, pkt2,pkt3,pkt4,pkt5
delivered, ACK2 sent
0 1 2 3 4 5 6 7 8 9

选择性重传: 两难选择

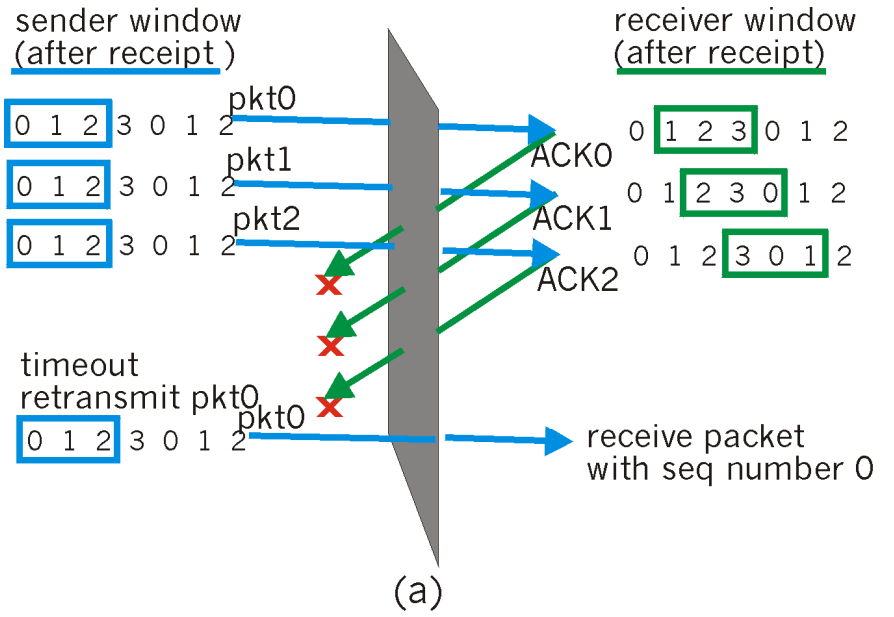
例子:

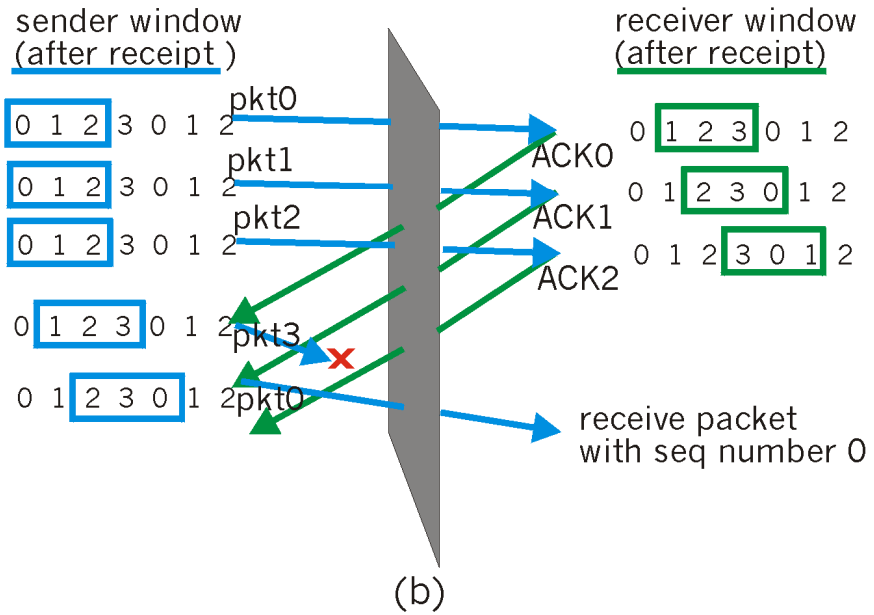
- 序号: 0, 1, 2, 3
- **window size=3**

- 在两种情况下接收方没有感觉到差别!

Q: 序号大小和窗口大小有什么关系? (小于或等于序号空间大小的一半)

$N \leq 2^{k-1}$





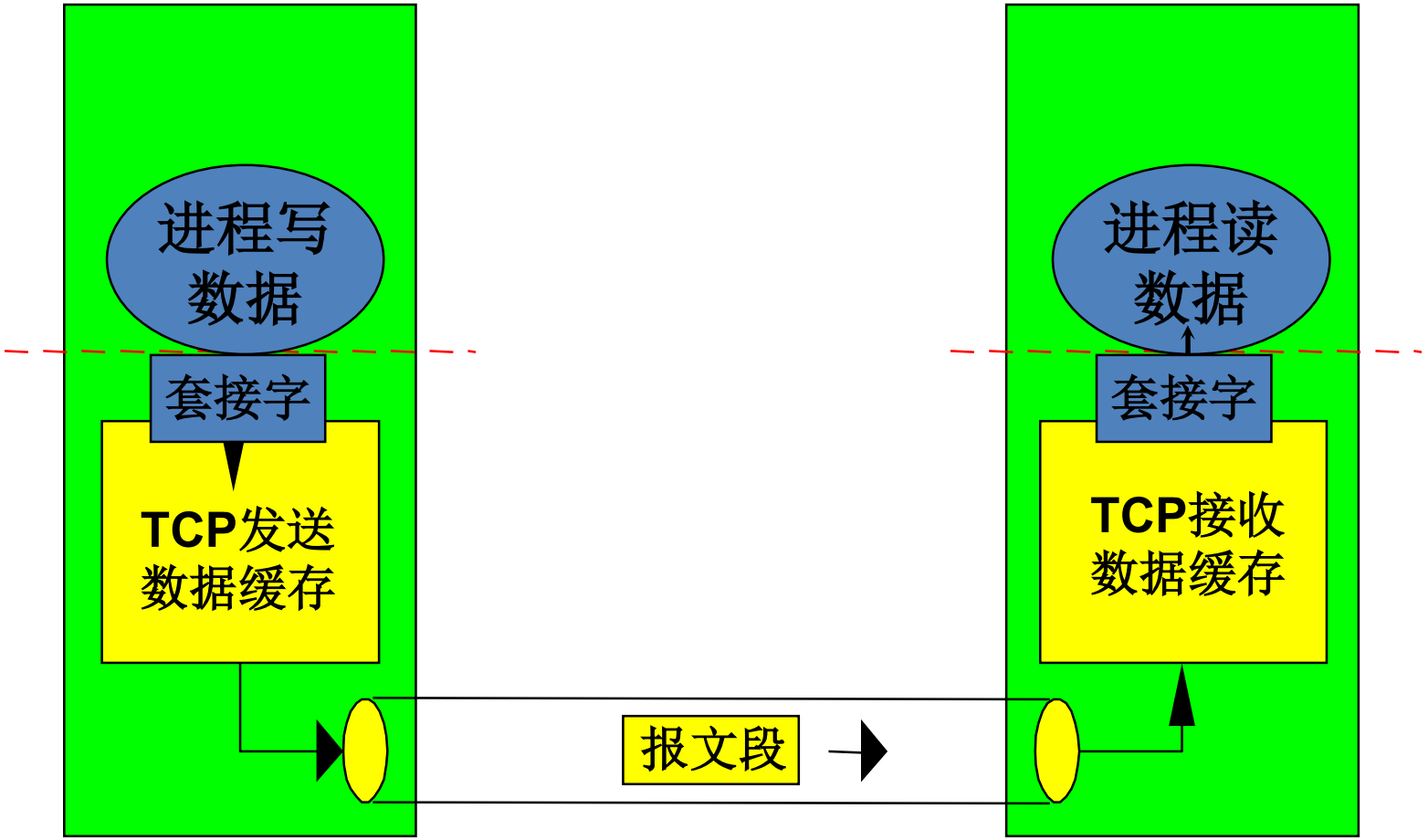
第三章 提纲

- 3.1 传输层服务
- 3.2多路复用和多路复用
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
- 3.5 面向连接传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

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

- 面向连接:
  - 在数据交换前三次握手(交换控制信息) 初始化发送方和接收方的状态
  - 客户进程向服务器进程发起
  - 连接状态保存在端系统
- 全双工数据:
  - 同一个连接上的双向数据流
- 点到点:
  - 一个发送者,一个接收者
- 收发缓冲区
  - 建立连接时设置
  - 最大报文段长度MSS(Maximum Segment Size)， 以太网1460字节，受最大传输单元MTU限制

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

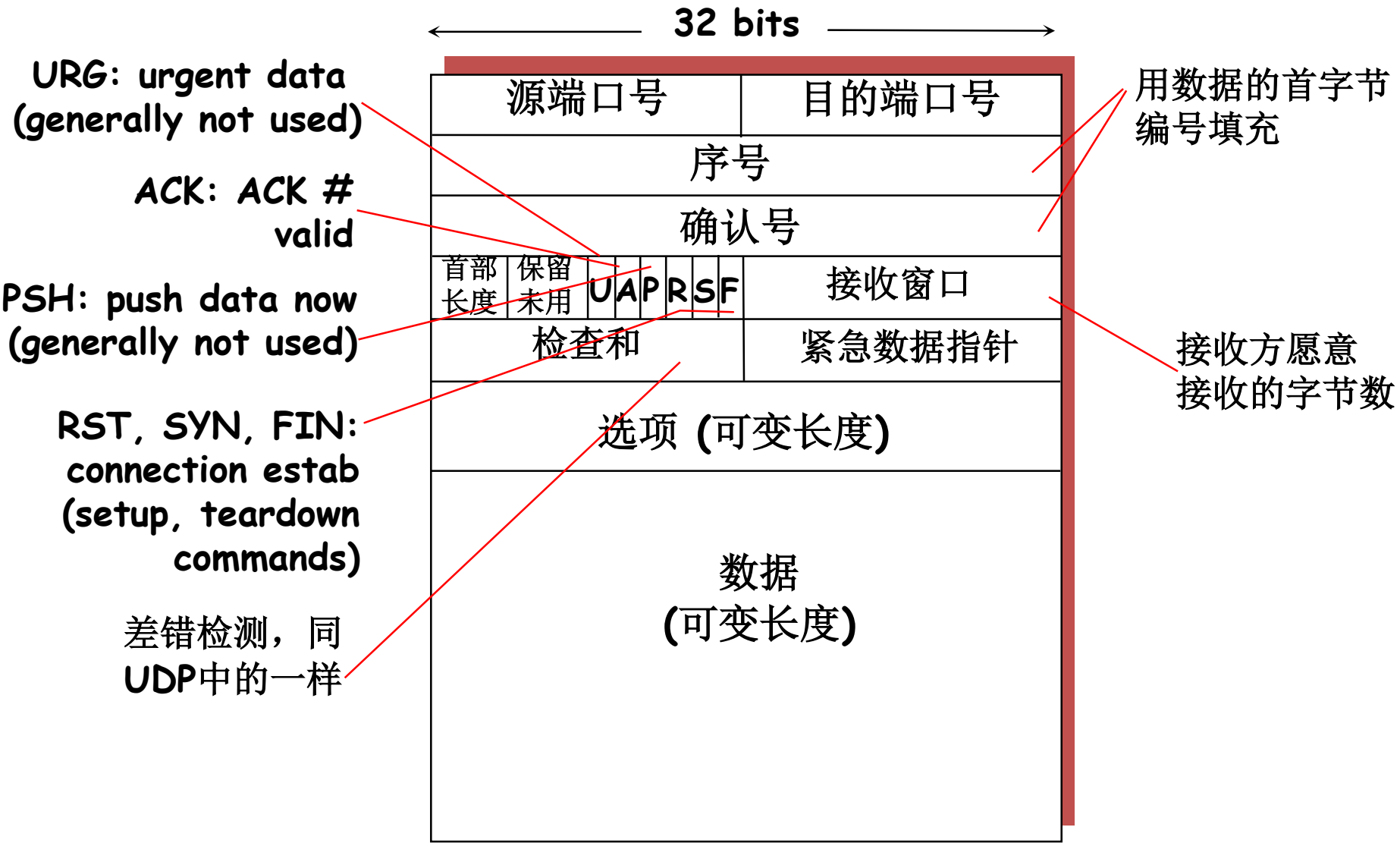


收发缓冲区

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

- 可靠按序的字节流:
  - “在方便的时候发送数据”
  - 没有“信息边界”
- 流水线:
  - TCP 拥塞控制设置窗口大小
- 流量控制:
  - 发送方不会淹没接收方

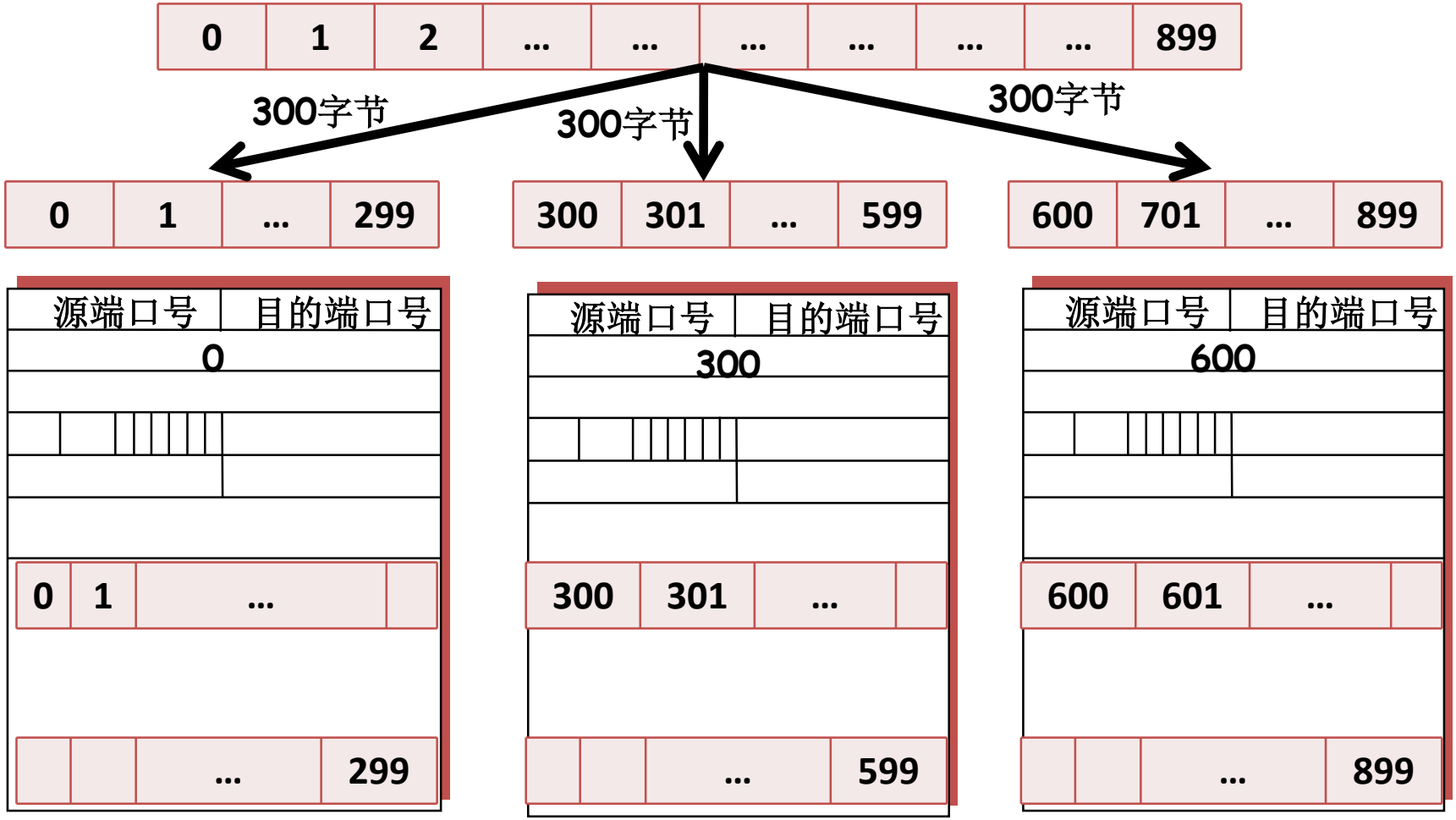
TCP 报文段结构


32 bits
URG: urgent data
(generally not used)
ACK: ACK #
valid
PSH: push data now
(generally not used)
RST, SYN, FIN:
connection estab
(setup, teardown
commands)
差错检测，同
UDP中的一样
源端口号
目的端口号
序号
确认号
首部长度
保留未用
U
A
P
R
S
F
接收窗口
检查和
紧急数据指针
选项 (可变长度)
数据
(可变长度)
用数据的首字节
编号填充
接收方愿意
接收的字节数

TCP 序号和确认

序号:

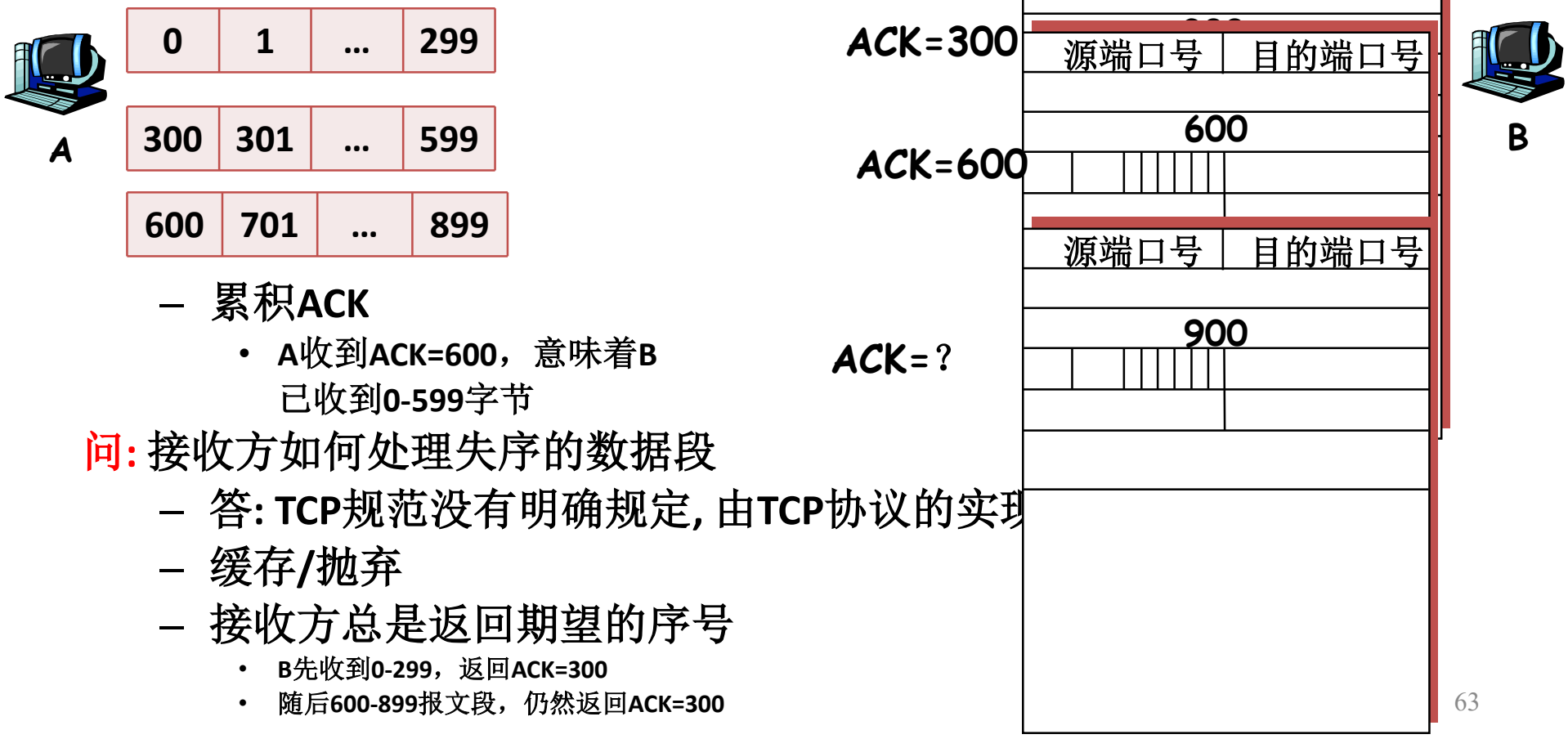
– 数据段中第一个字节在数据流中的位置编号



TCP 序号和确认

确认:

- 一方填充的确认号是它期望从另一方收到的下一个字节的序号

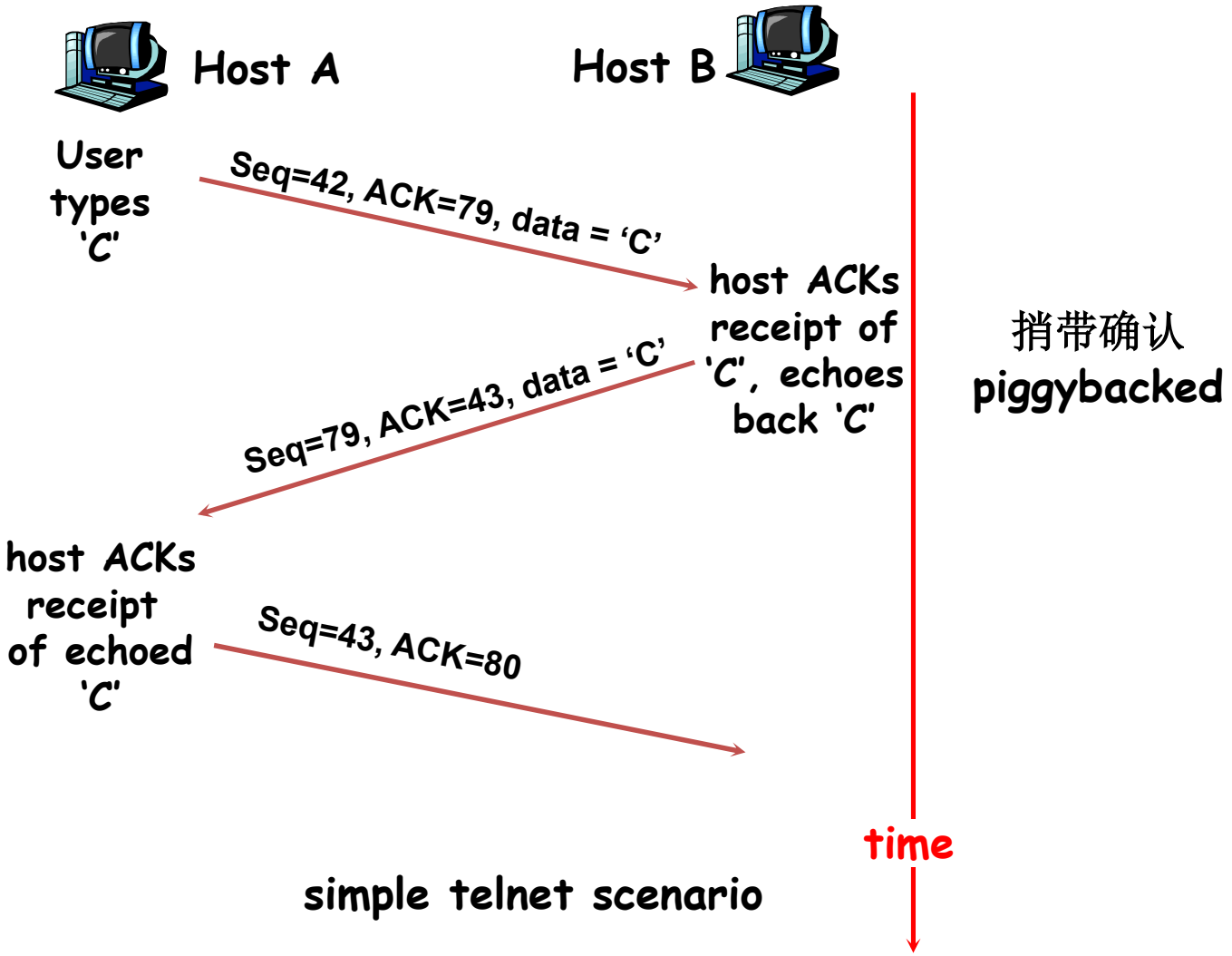


- 累积ACK
  - A收到ACK=600，意味着B已收到0-599字节

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TCP协议的实现
- 缓存/抛弃
- 接收方总是返回期望的序号
  - B先收到0-299，返回ACK=300
  - 随后600-899报文段，仍然返回ACK=300

TCP 序号和确认



simple telnet scenario

TCP 往返时延的估计和超时

问: 如何设置 TCP 超时值?

- 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

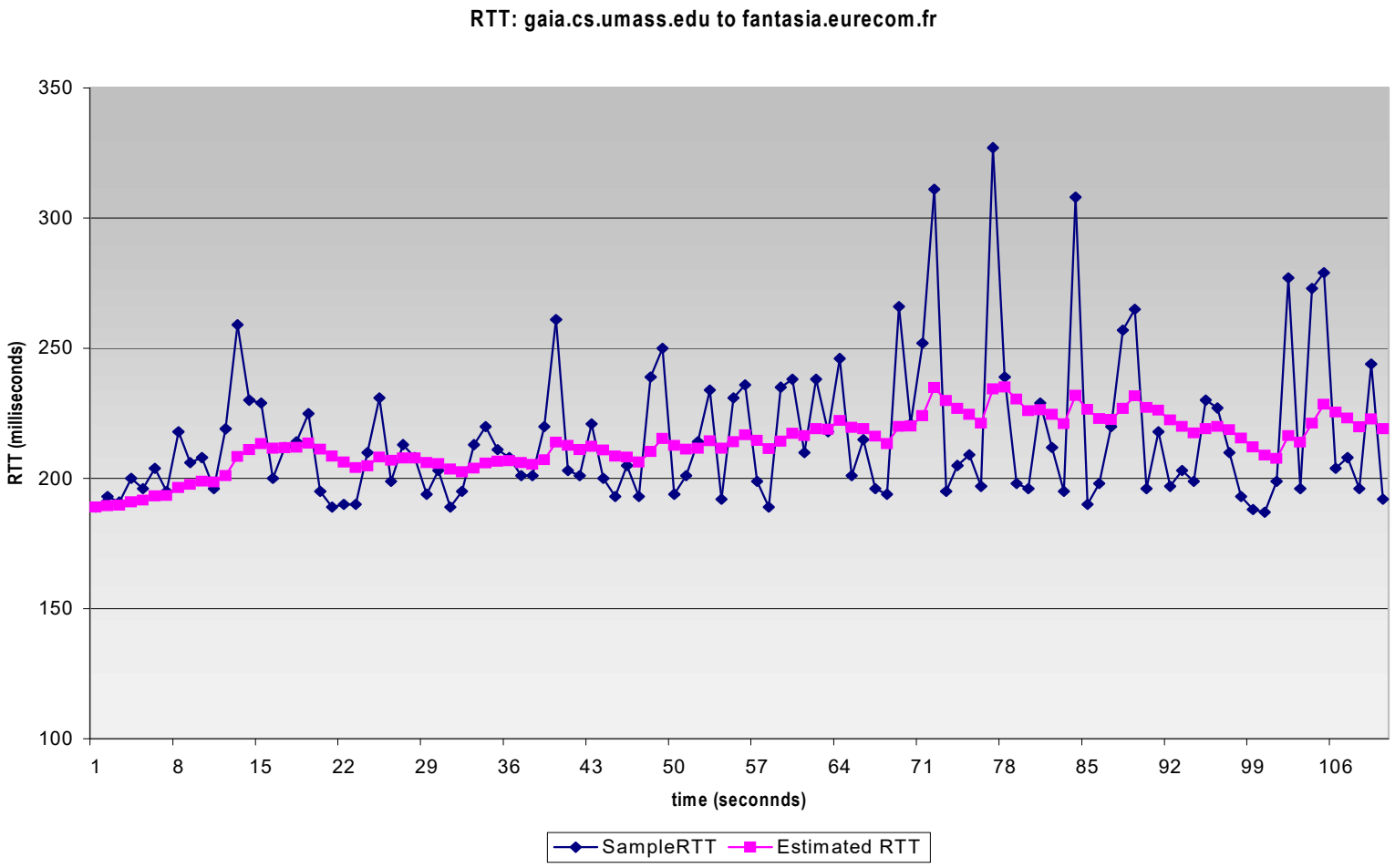
问: 如何估计 RTT?

- 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

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

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
RTT (milliseconds)
350
300
250
200
150
100
1
8
15
22
29
36
43
50
57
64
71
78
85
92
99
106
time (seconnds)
SampleRTT
Estimated RTT

TCP往返时延的估计和超时

设置超时

- **EstimtedRTT** 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

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

(典型地, $\beta = 0.25$)

然后设置超时时间间隔:

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

第三章 提纲

- 3.1 传输层服务
- 3.2多路复用和多路分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
- 3.5 面向连接传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

TCP 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
  - 冗余确认

TCP 发送方事件:

从应用程序接收数据:

- 用序号创造一个数据段
- 序号是数据段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数据段发送时启动的)
- 设置超时间隔: `TimeOutInterval`

超时:

- 重发导致超时的数据段
- 重新开始定时器

收到确认:

- 如果确认了还没有确认的数据段
  - 更新还没有确认的状态
  - 还有未完成的数据段，重新开始定时器

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: 上层产生数据data，创建序号为NextSeqNum的报文段
      if (定时器未启动)
          启动 timer
      将报文段下传到IP
      NextSeqNum = NextSeqNum + length(data)

  event: 定时器超时
      重传最小序号的未被确认的报文段；
      启动 timer

  event: 收到确认报文，确认字段值为y
      if (y > SendBase) {
          SendBase = y;//累积确认
          if (还有未被确认的数据段)
              启动 timer
          }

} /* end of loop forever */
```

TCP 发送方 (简化的)

注意：

- SendBase-1：已累积确认的字节序号

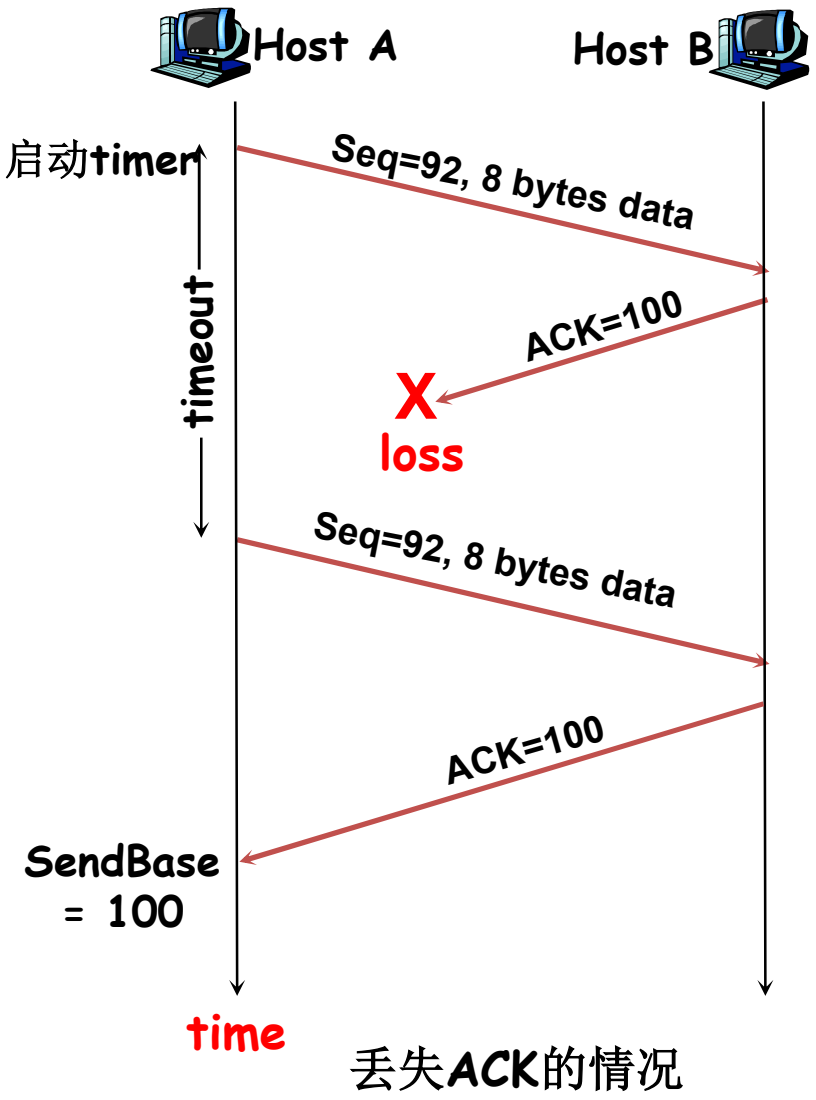
Example:

- SendBase-1 = 71；

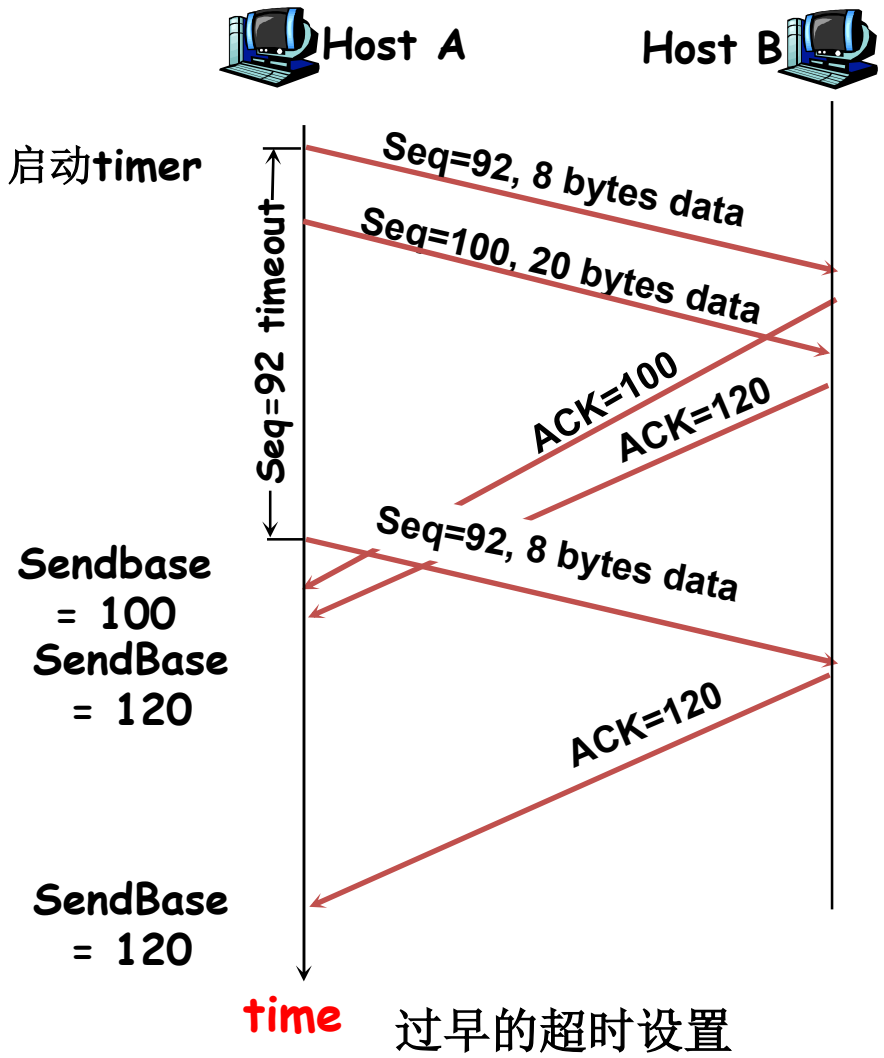
y= 73，接收方需要序号73以后的字节；

y > SendBase，新的数据被确认

TCP: 重发场景

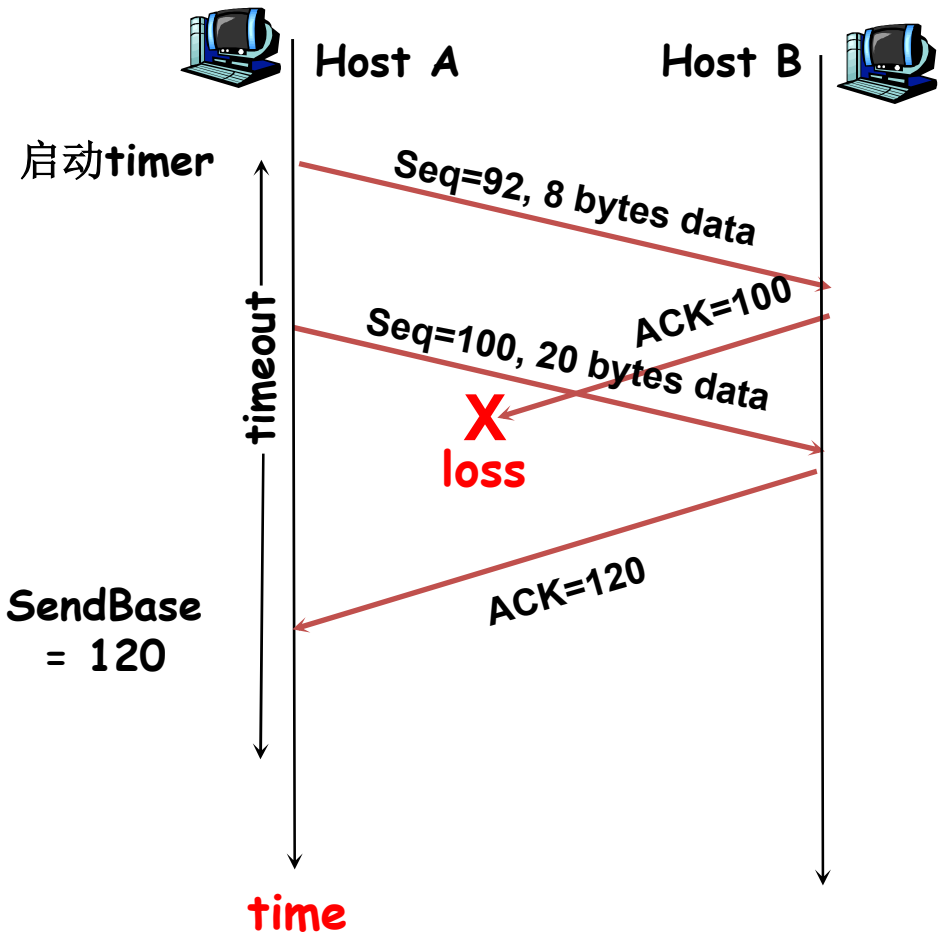


丢失ACK的情况



过早的超时设置

TCP 重发场景 (更多)



累积ACK的情况

超时间隔

- 上层发送报文段或收到ACK
  - TimeOutInterval根据EstimatedRTT和DevRTT设置
- 定时器超时
  - TimeOutInterval=2* TimeOutInterval

TCP ACK 的产生 [RFC 1122, RFC 2581]

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

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3个对同样报文段的冗余确认，则发送方认为该报文段已经丢失。
  - 启动快速重传:在定时器超时之前重发丢失的报文段

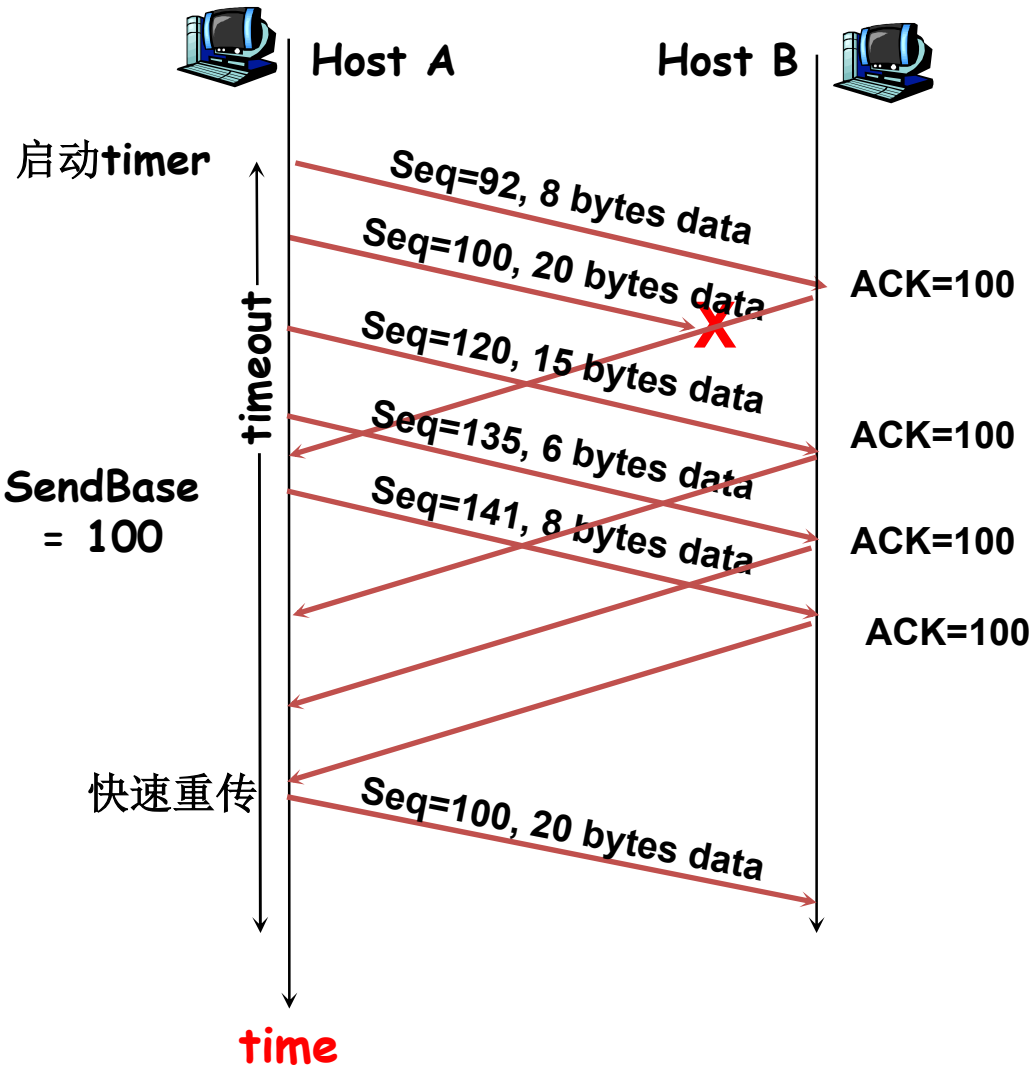
快速重传算法:

```
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

快速重传


Host A
Host B
启动timer
Seq=92, 8 bytes data
Seq=100, 20 bytes data
ACK=100
X
Seq=120, 15 bytes data
ACK=100
timeout
Seq=135, 6 bytes data
SendBase
= 100
Seq=141, 8 bytes data
ACK=100
ACK=100
快速重传
Seq=100, 20 bytes data
time

TCP是GBN还是SR

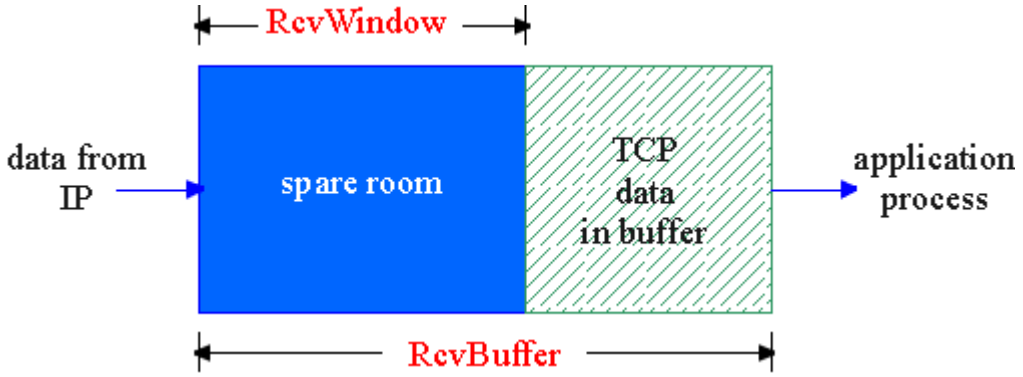
- TCP具有GBN的特点
  - 累积确认
  - 一个重传定时器
- TCP具有SR的特点
  - 只重传一个报文
  - 缓存失序报文

第三章 提纲

- 3.1 传输层服务
- 3.2多路复用和多路复用
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
- 3.5 面向连接传输: TCP
  - 数据段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

TCP 流量控制

- TCP连接的接收边有一个接收缓冲区:



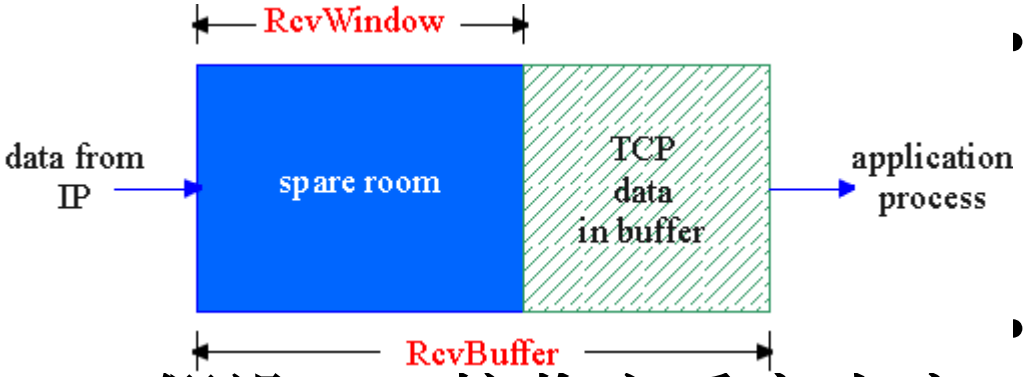
应用程序可能从这个缓冲区读出数据很慢

- 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

Rwnd=RcvBuffer-[LastByteRcvd-LastByteRead]

- 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

LastByteSent-LastByteAcked<=Rwnd

第三章 提纲

- 3.1 传输层服务
- 3.2多路复用和多路复用
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
- 3.5 面向连接传输: TCP
  - 数据段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

TCP 连接管理

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(继续)

关闭连接:

客户关闭套接字:

```
clientSocket.close();
```

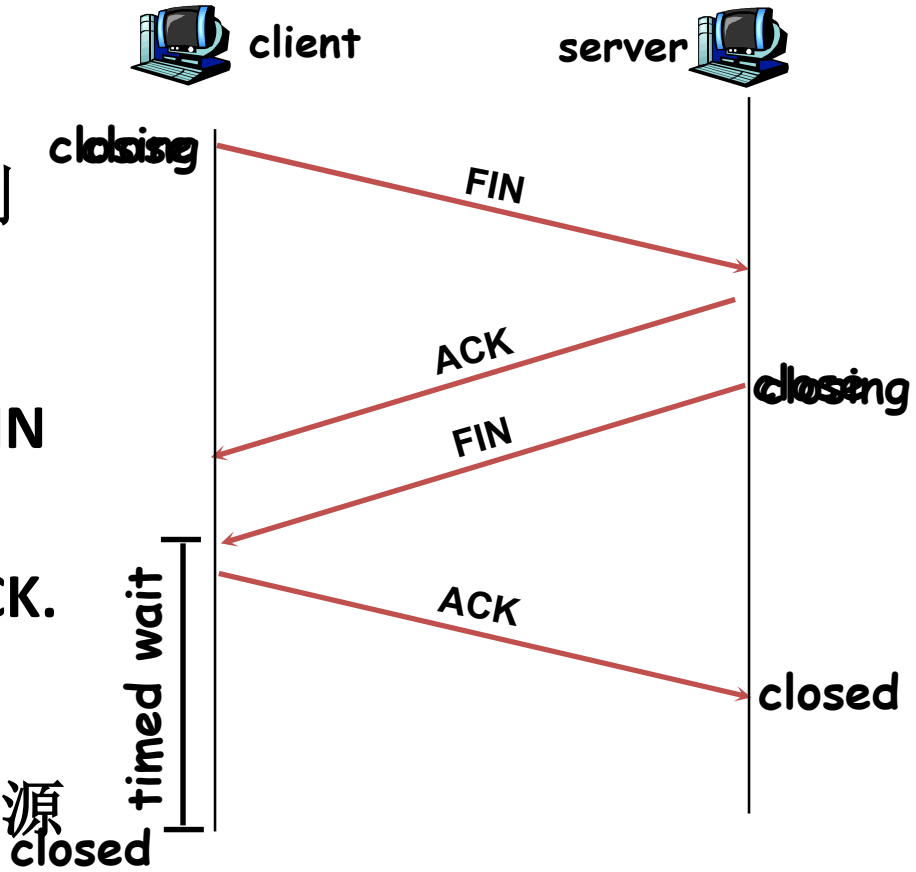
Step 1: 客户发送 TCP FIN 控制报文段到服务器

Step 2: 服务器接收 FIN, 回复 ACK. 半关闭连接, 并发送FIN 到客户

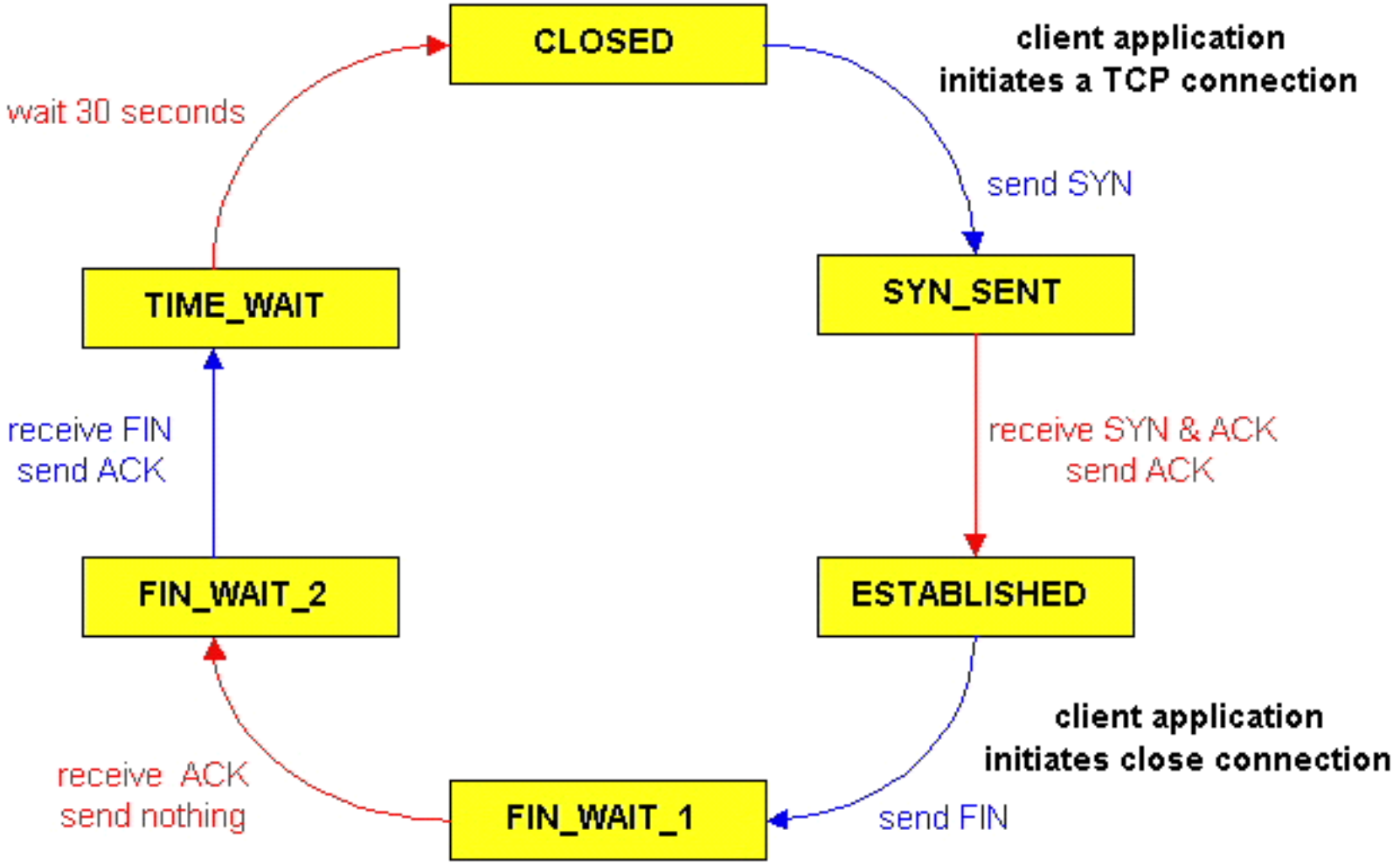
Step 3: 客户接收 FIN, 回复 ACK.

- 进入 “timed wait”
- 等待结束时释放连接资源

Step 4: 服务器接收 ACK. 连接关闭.

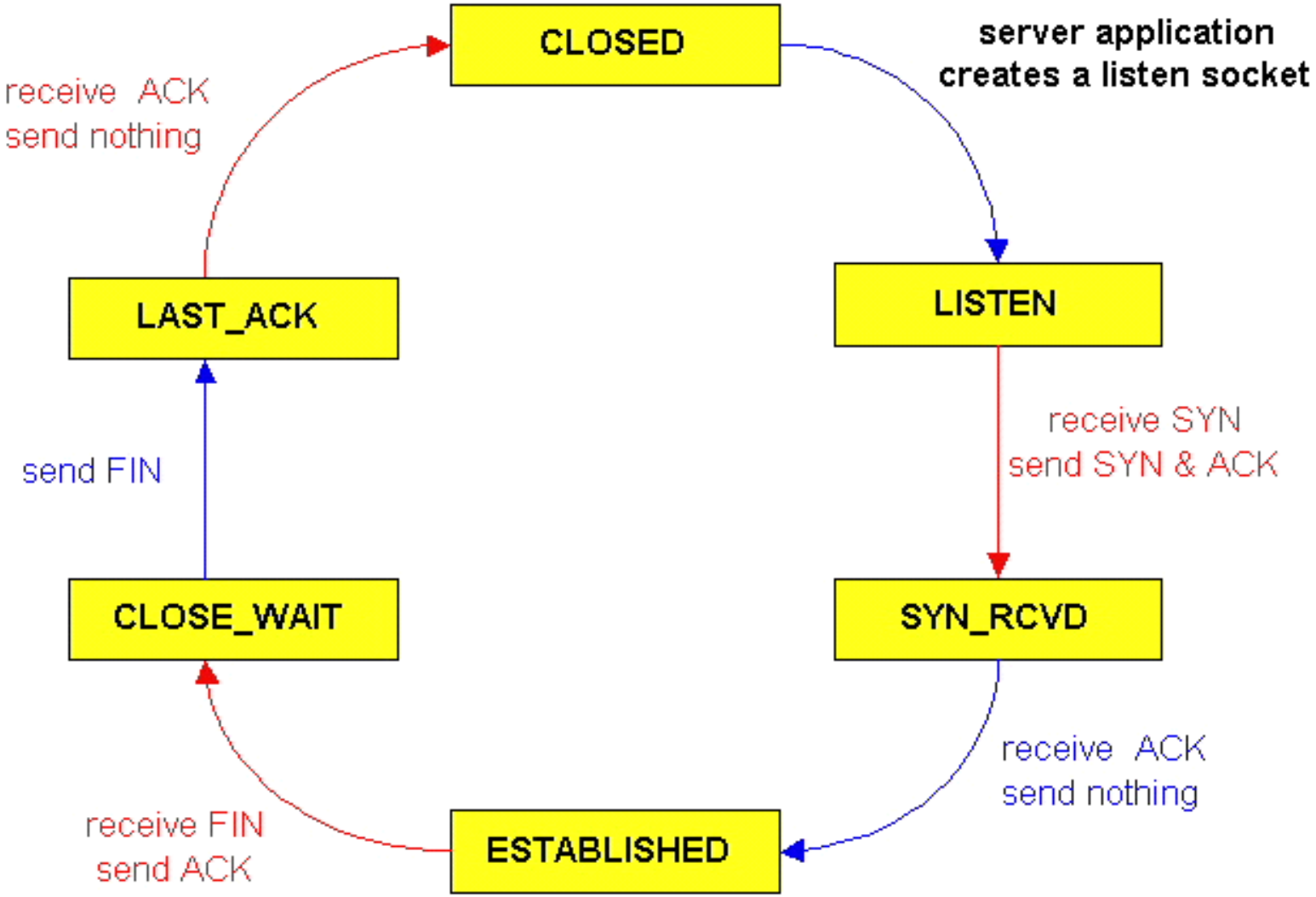


TCP 连接管理(继续)



TCP 客户端状态转换图

TCP 连接管理(继续)



TCP 服务器端状态转换图

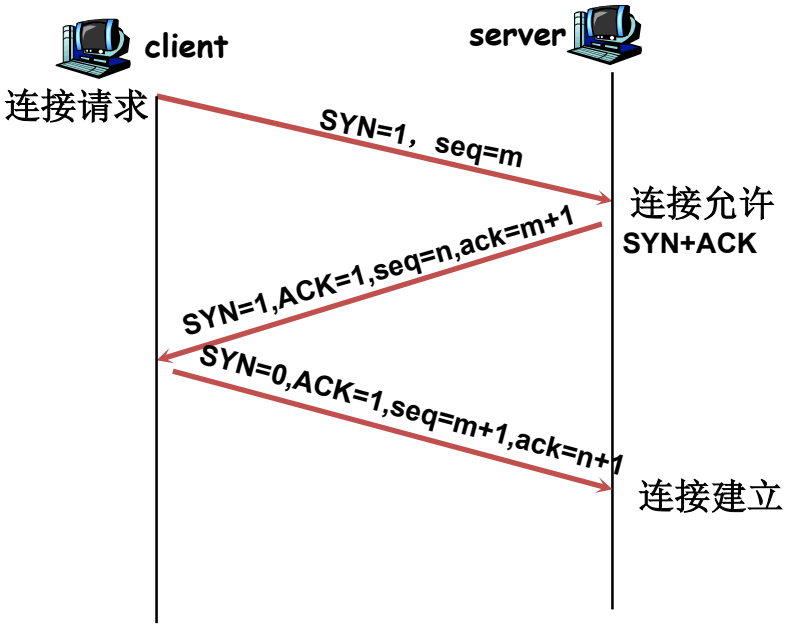
TCP的可靠数据传输机制

- 序号、确认号
- 校验和
- 累积确认
- 一个重传定时器，超时重传
- 快速重传，收到3个冗余的确认（4个重复的确认）

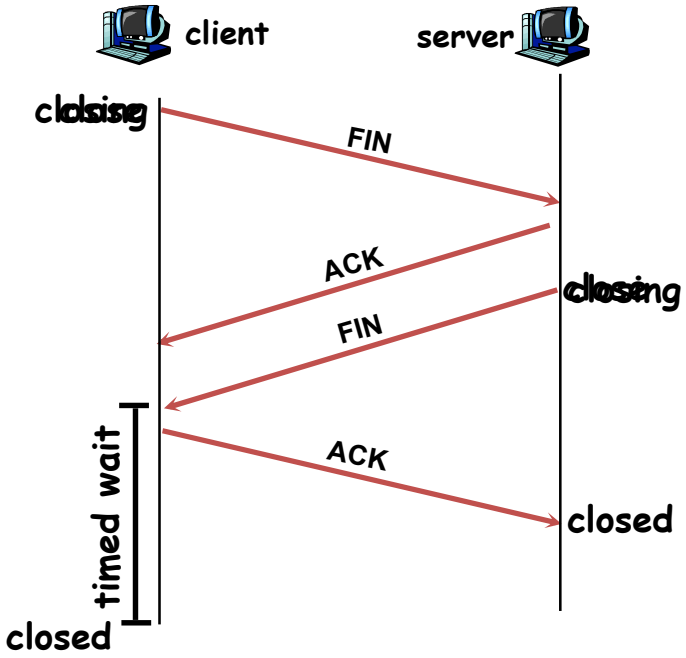
	TCP	GBN	SR
确认方式	累积确认	累积确认	单独确认
确认序号	期望序号的确认	正确接收分组最高序号的确认 (期望序号-1)	对接收分组序号的确认
重传定时器数量	1个，发出未被确认的最小序号报文段	1个，发出未被确认的最小序号分组	每个发出未被确认的分组分别有一个定时器
超时重传机制	只重发定时器超时的报文段	所有发出未被确认的分组均重发	只重发定时器超时的分组
快速重传机制	有	无	无
失序报文处理	缓存/丢弃	丢弃	缓存

TCP的流量控制和连接管理

- TCP流量控制
  - 速率匹配服务，发送方发送速度不会淹没接收方，针对特定的收发双方
  - 通过TCP首部中的接收窗口字段实现
- TCP连接建立，三次握手



- TCP连接断开，四次握手



第三章 提纲

- 3.1 传输层服务
- 3.2多路复用和多路复用
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
- 3.5 面向连接传输: TCP
  - 数据段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

拥塞控制原理

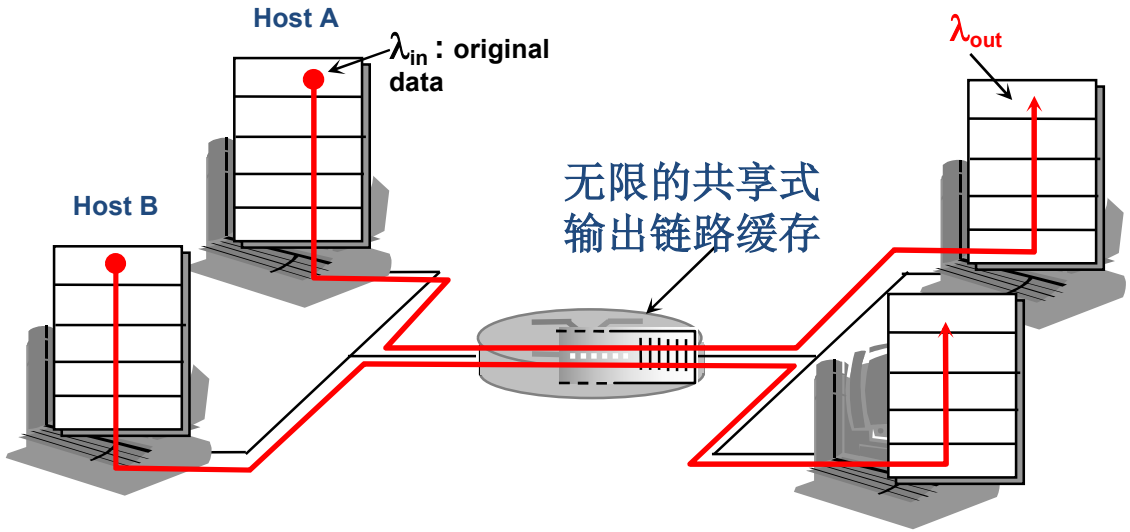
拥塞:

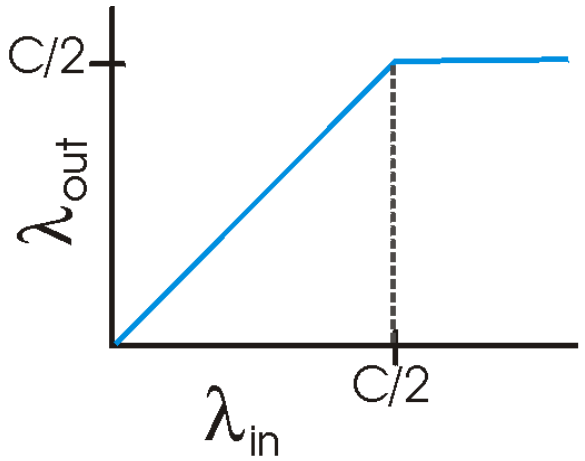
- 从信息角度看:“太多源主机发送太多的数据，速度太快以至于网络来不及处理”
- 和流量控制不同!
- 表现:
  - 丢失分组 (路由器的缓冲区溢出)
  - 长延迟 (在路由器的缓冲区排队)

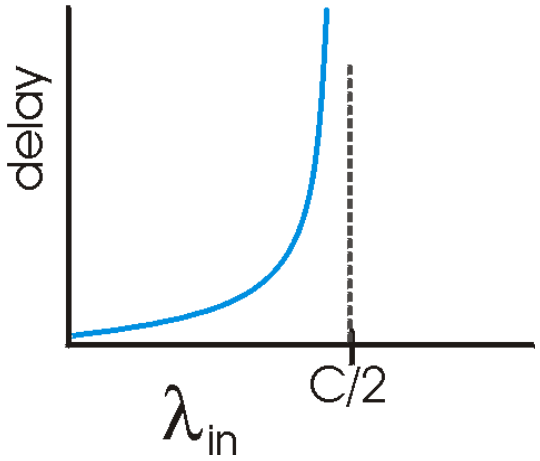
拥塞的原因和代价: 场景1

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



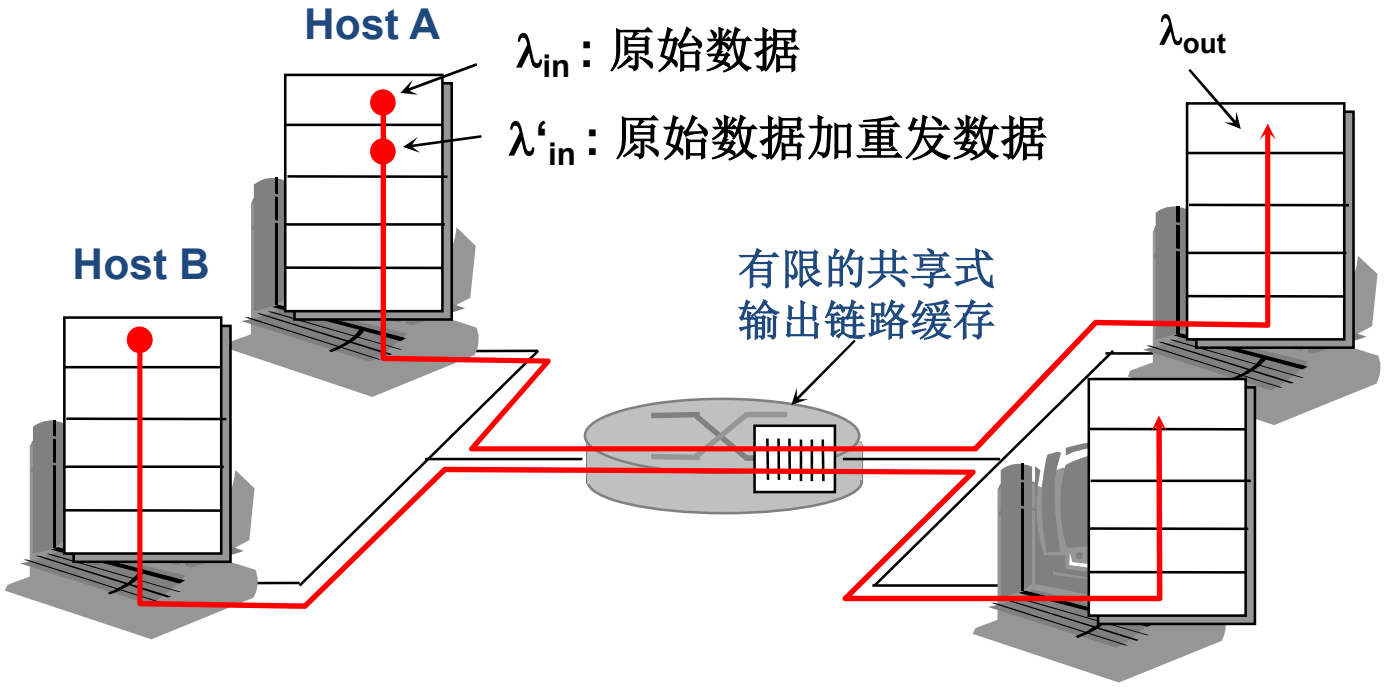




- 每个主机最大可达吞吐量C/2,总的吞吐量为C
- 但是,拥塞时延在接近链路容量时达到无限大

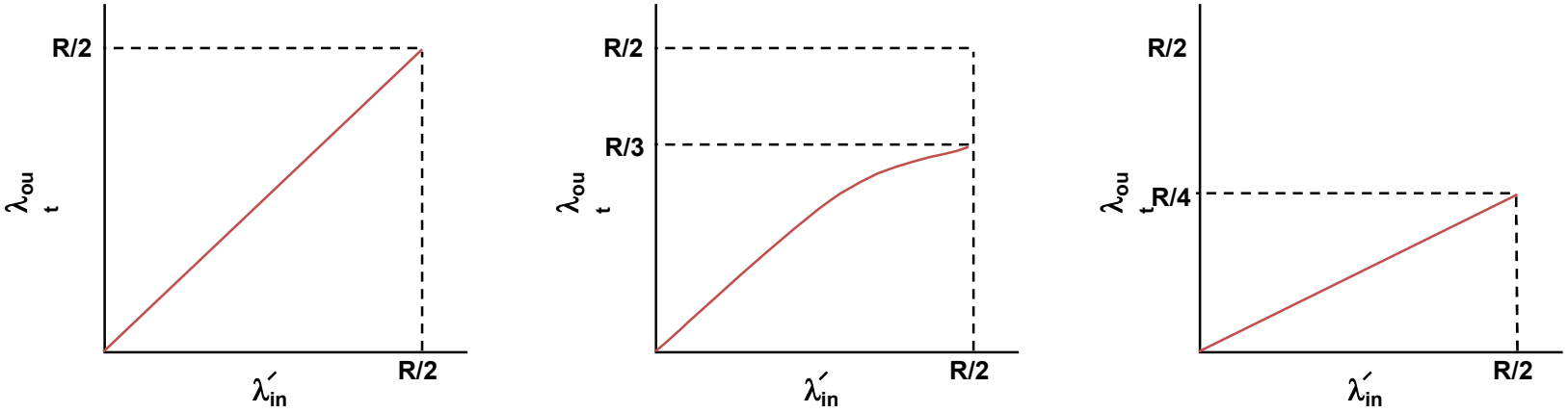
拥塞的原因和代价: 场景2

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



a. b. c.

拥塞的“代价”

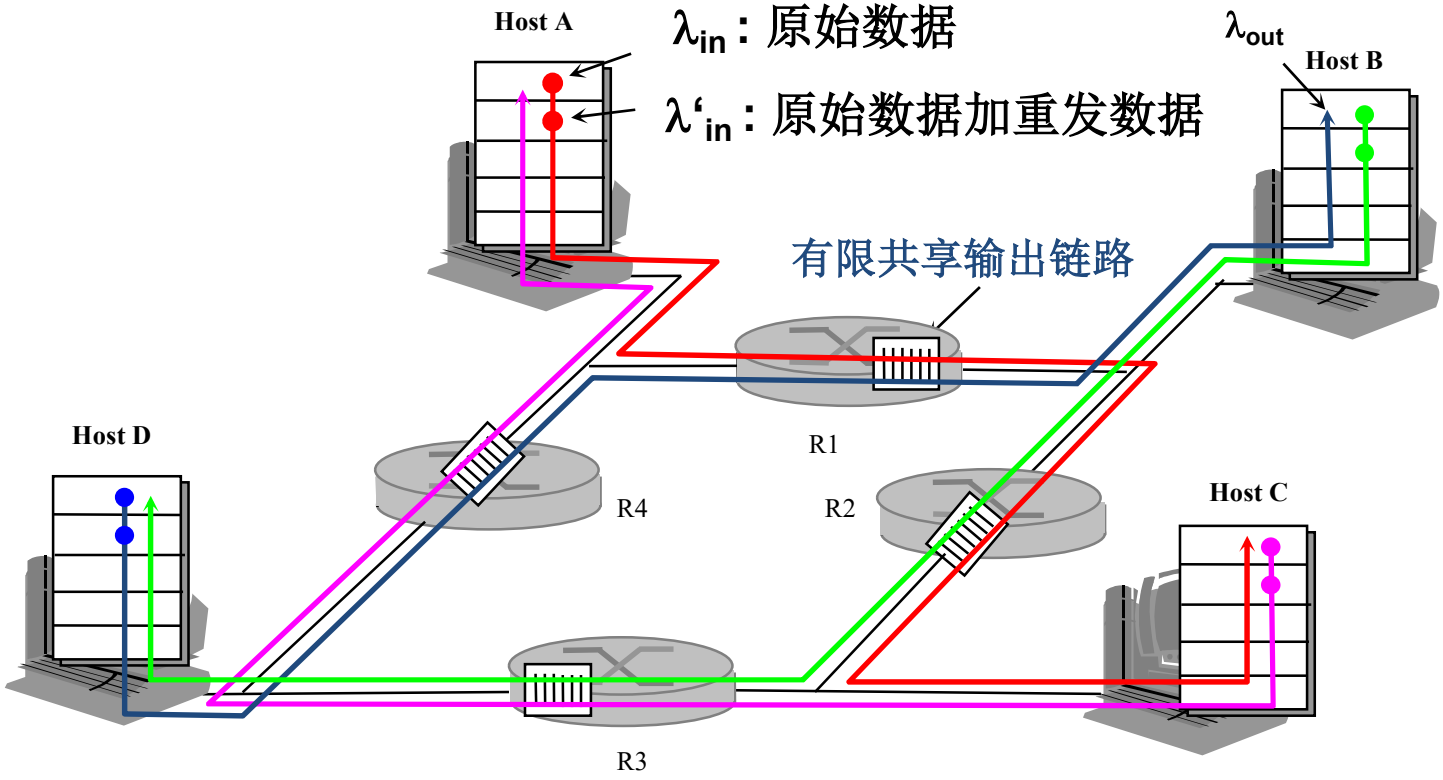
b:更多的工作 (重发) 用来得到“好的吞吐量”

c:不必要的重发: 链路需要运输多个分组的拷贝

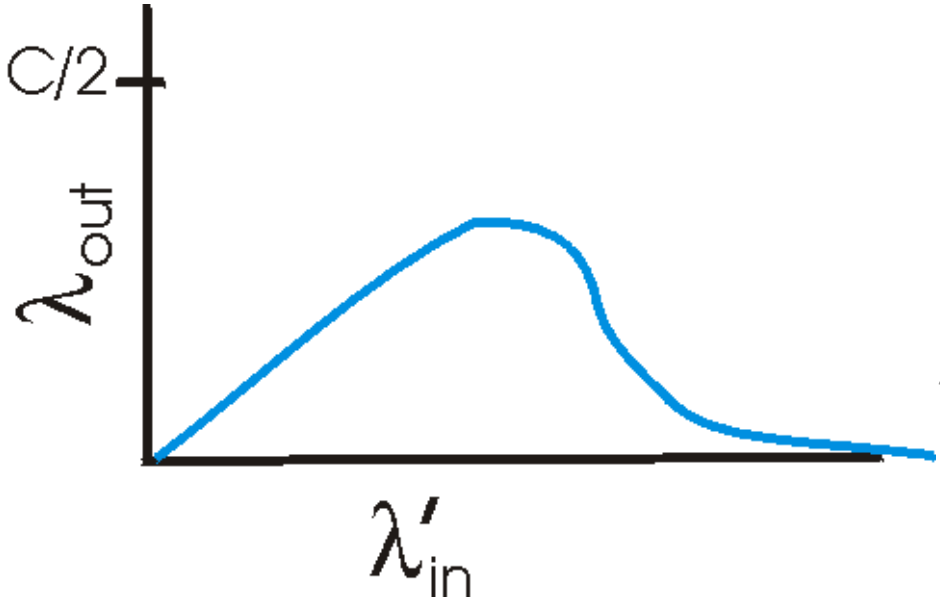
拥塞的原因和代价: 场景3

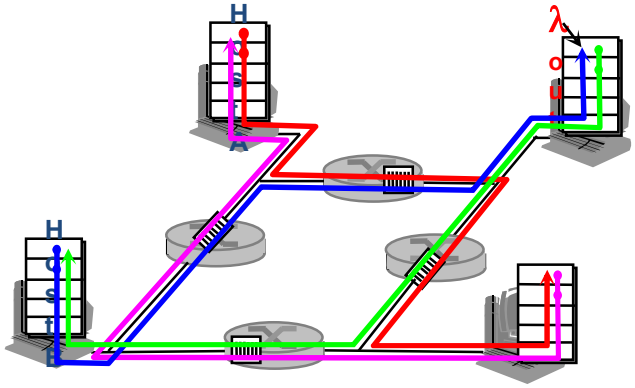
- 四个发送方
- 多跳路径
- 超时/重发

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

拥塞控制有两个主要的方法:

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, DECbit, 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（自学）

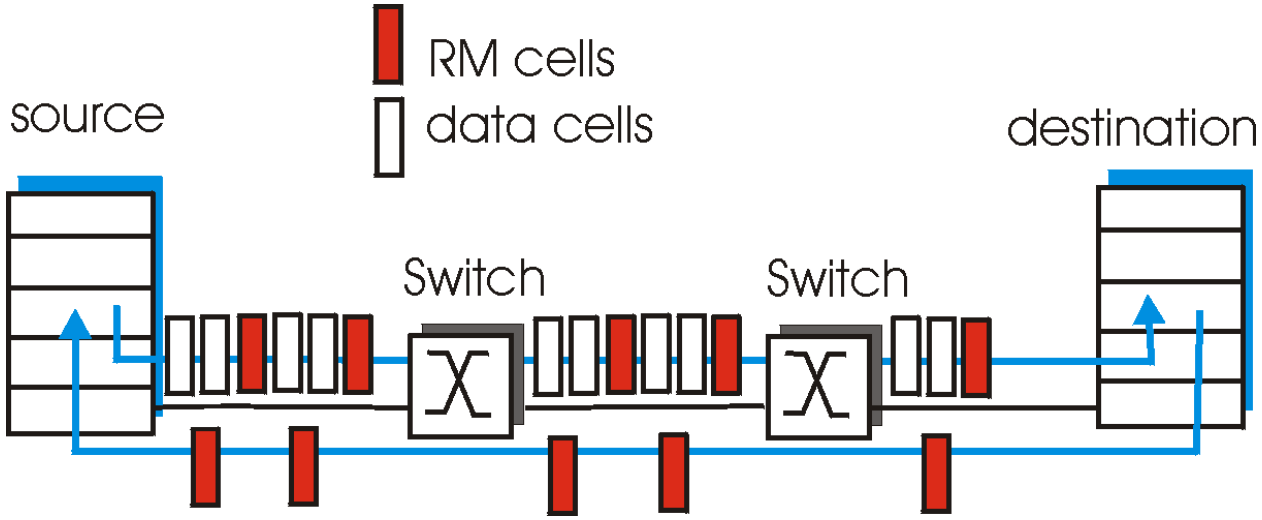
ABR: 可用比特率:

- “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

第三章 提纲

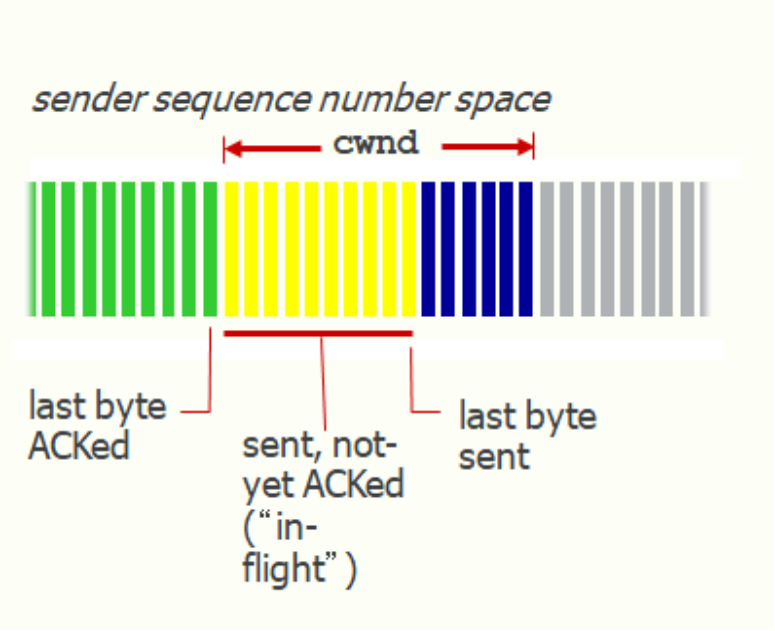
- 3.1 传输层服务
- 3.2多路复用和多路分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
- 3.5 面向连接传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP 拥塞控制

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( Cwnd, RcvWindow )`
- 大体上,

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

- Cwnd是动态的, 感知的网络拥塞的函数



TCP 拥塞控制

发送方如何感知拥塞?

- 丢失事件 = 超时或者 3 个冗余的ACKs
- TCP 发送方在丢失事件发生后降低发送速率 (Cwnd)

发送方如何判断不拥塞?

- 收到确认报文段，增加发送方发送速率

两个阶段:

- 慢启动阶段
- 拥塞避免阶段
  - AIMD机制
  - 对丢包事件作出反应

TCP 慢启动(Slow Start, SS)

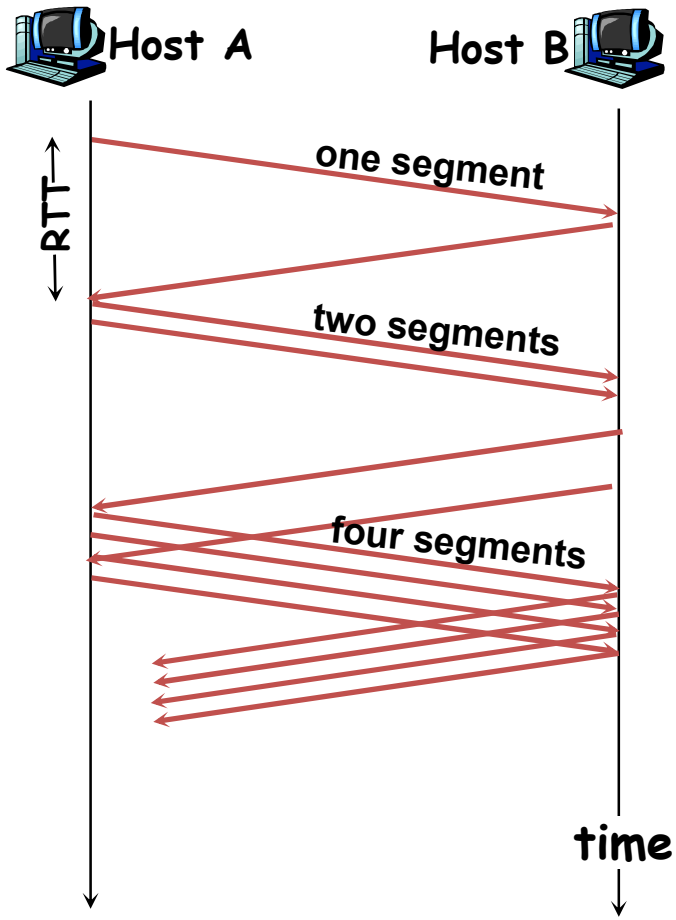
- 连接开始的时候,
  Cwnd = 1 MSS
  - **Example: MSS = 500 bytes & RTT = 200 msec**
  - 初始速率 = **20 kbps**
- 有效带宽将 >>
  MSS/RTT
  - 希望尽快达到期待的速率

当连接开始的时候以2的指数方式增加速率

- 1 MSS
- 2 MSS
- 4 MSS
- 8 MSS
- ……

TCP 慢启动(更多)

- 当连接开始的时候以指数方式增加速率
  - 倍增 Cwnd在每个 RTT 内
  - 每收到ACK完成增加 Cwnd
- 总结: 初始速率慢但是呈指数快速增长



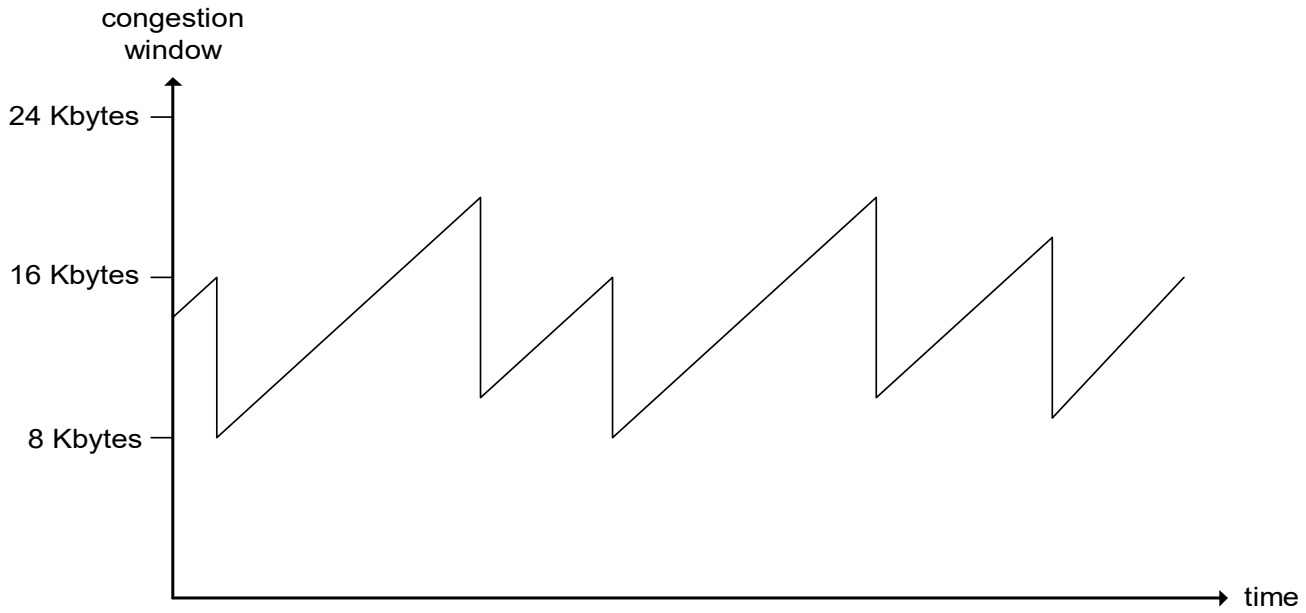
拥塞避免阶段

- 慢启动阶段Cwnd超过设定的阈值时进入拥塞避免（Collision Avoidance, CA）阶段
- Cwnd持续增长，直至发生丢包事件
  - 收到三个冗余的确认：Cwnd按AIMD机制变化
  - 但超时事件后，TCP进入慢启动过程:

TCP AIMD(Additive-increase,multiplicative-decrease)

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：*检测*

乘性递减: 发生丢失事件后将拥塞窗口减半



Long-lived TCP connection

对丢包事件的反应

- 收到三个冗余的确认后：
  - Cwnd 减半+3个MSS
  - 然后，窗口线性增长
- 但超时事件后，TCP进入慢启动过程:
  - Cwnd 立即设置为 1个MSS;
  - 窗口开始指数增长
  - 到一个阀值后再线性增长

原理

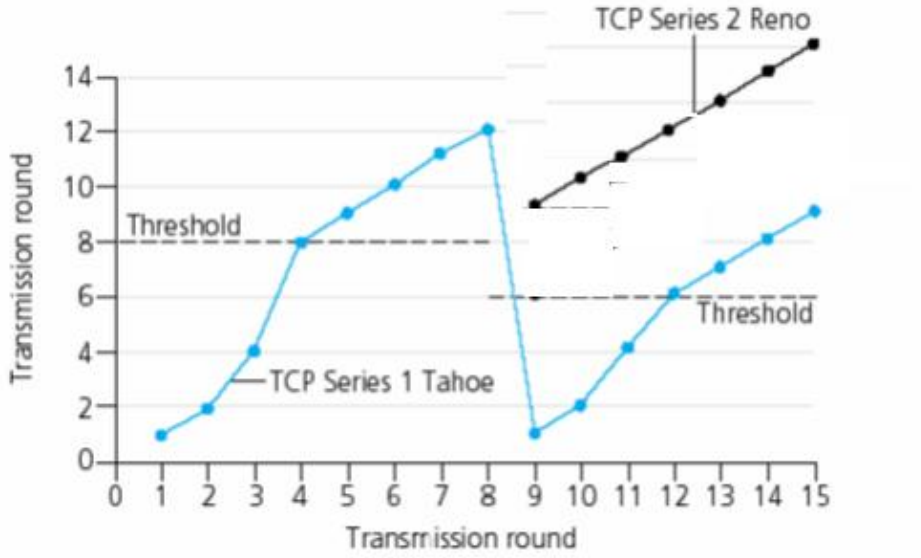
• 3 个冗余的 ACKs 表明网络具有传输一些数据段的能力

• 在三个冗余的确认之前超时是“更加严重的警告”

对超时事件的反应(更多)

问: 什么时候从指数增加变为线性增加

答: 当 Cwnd 达到超时前的一半的时候.



实现:

- 变化的阈值Threshold
- 发生丢包事件后,阈值设置为丢失前的 Cwnd 的一半

总结: TCP 拥塞控制

- 当 Cwnd 低于阈值, 发送方处于慢启动阶段, 窗口指数增长.
- 当 Cwnd 高于阀值, 发送方处于拥塞避免阶段, 窗口线性增长.
- 当三个冗余的ACK 事件出现时,阈值置为 Cwnd/2 并且Cwnd 置为Cwnd/2 +3.发送方处于拥塞避免阶段
- 当超时事件发生时， 阈值置为Cwnd/2 并且 Cwnd 置为1 MSS.发送方处于慢启动阶段

TCP 发送方拥塞控制

Event	State	TCP Sender Action	Commentary
收到未被确认数据的ACK	慢启动 (SS)	Cwnd = Cwnd +MSS, If (Cwnd > Threshold) 进入拥塞避免状态	每经1个RTT时间Cwnd加倍
收到未被确认数据的ACK	拥塞避免状态 (CA)	Cwnd = Cwnd+MSS * MSS / Cwnd	线性递增， 每经1个RTT时间Cwnd = Cwnd +1MSS
收到3个冗余的ACK	SS or CA	Threshold = Cwnd/2+3, Cwnd = Threshold, 进入拥塞避免阶段	快速重传， Cwnd减半（指数递减）
超时	SS or CA	Threshold = Cwnd/2, Cwnd = 1 MSS, 进入慢启动阶段	Enter slow start
收到冗余的ACK	SS or CA	对冗余的ACK计数	Cwnd 和 Threshold 没有变化

第三章 总结

- 传输层服务后面的原理:
  - 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”

第三章 复习大纲

- 传输层提供的服务
  - 进程通信
  - 面向连接和无连接（是否建立连接）
  - 可靠传输实现原理
- UDP协议特性
- 校验和的实现思想
- TCP协议特性及其实现
  - TCP报文，固定报头为20字节
  - 连接管理
  - 可靠传输
  - 流量控制
  - 拥塞控制

第3章 传输层

- 传输层：主机进程之间端到端的逻辑通信
- 网络层：主机之间的逻辑通信
- 传输层提供的服务*
  - 面向连接的TCP可靠传输服务
  - 无连接的UDP不可靠传输服务
  - 建立在网络层尽力而为服务的基础上，不保证服务质量
- 多路分解/复用
  - UDP套接字：目的IP，目的端口
  - TCP套接字：源IP，源端口，目的IP，目的端口
- UDP协议
  - UDP数据会丢失、失序到达
  - 优点：简单快速，延迟小，开销小

UDP协议

- 适应场景
  - 容忍数据丢失
  - 速率敏感
  - 链路传输可靠
- UDP报文格式
- 校验和计算*
  - 16bit整数对齐相加
  - 有溢出则回卷
  - 结果取反
  - 接收方校验：结果非全1表示出错；全1表示没有检测到错误（可能有错）

UDP 数据报格式

可靠数据传输原理

- 不可靠信道的特性将决定可靠数据传输协议(rdt)的复杂性

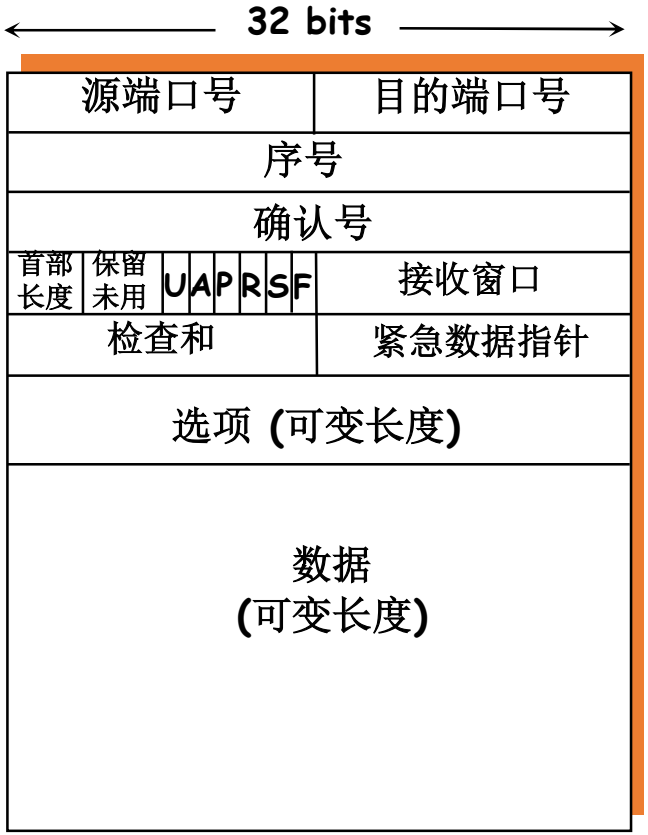
	信道假设		可靠传输机制
Rdt1.0	完美可靠		无
Rdt2.0	bit错误	数据bit错误	差错检测机制（校验和）*
			接收方反馈（ACK，NAK）*
			发送方重传*
Rdt2.1		数据和确认bit错误	序号（解决接收方收到重复分组问题）*
Rdt2.2			“ACK+序号”取代NAK
Rdt3.0	bit错误+丢失		定时器，超时重传*
Go-Back-N	bit错误+丢失		流水线技术，序号增大，发送方缓冲区（发送窗口）*
选择性重传	bit错误+丢失		流水线技术，发送窗口、接收窗口*

可靠传输协议*

	停等协议rdt3.0	GBN	SR
发送窗口大小（序号k比特）	1	$<=2^k-1$， >1	$<=2^{k-1}$，>1
接收窗口	1	1	>1
确认方式	单独确认	累积确认	单独确认
定时器数量	无	1个，发出未被确认的最小序号分组	每个发出未被确认的分组分别有一个定时器
超时重传机制	无	所有发出未被确认的分组均重发	只重发定时器超时的分组
失序报文处理	丢弃	丢弃	缓存

TCP协议

- 面向连接
- 全双工
- 点到点
- 可靠按序的字节流
- 拥塞控制
- 流量控制
- TCP报文段结构
  - 固定首部长20字节，最长60字节 *
  - 序号：报文段中第一个字节在应用报文数据流中的位置编号*
  - 确认号：期望从另一方收到的下一个字节的序号*
  - 首部长度：4bit，4字节为单位
  - 接收窗口：流量控制，接收方能够接收的字节数*



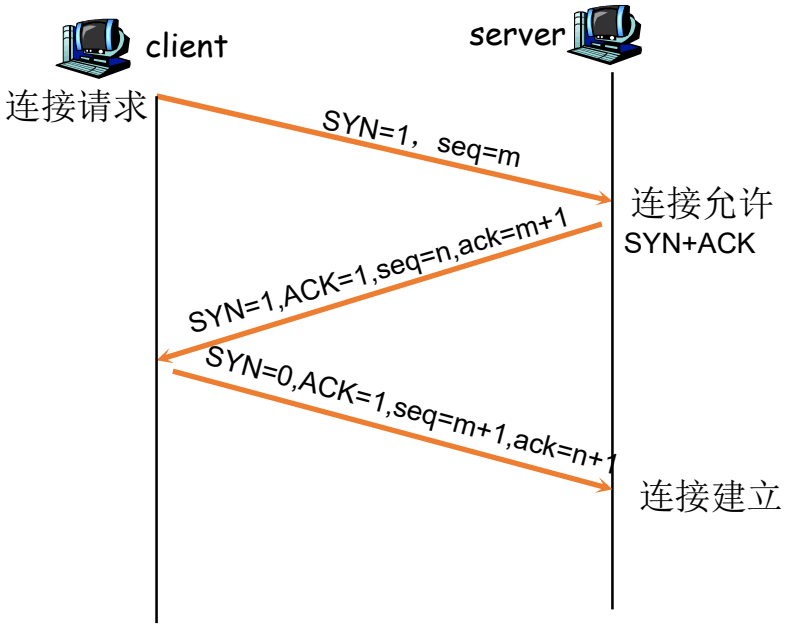
TCP的可靠数据传输机制*

- 序号、确认号
- 校验和
- 累积确认
- 一个重传定时器，超时重传
- 快速重传，收到3个冗余的确认（4个重复的确认）

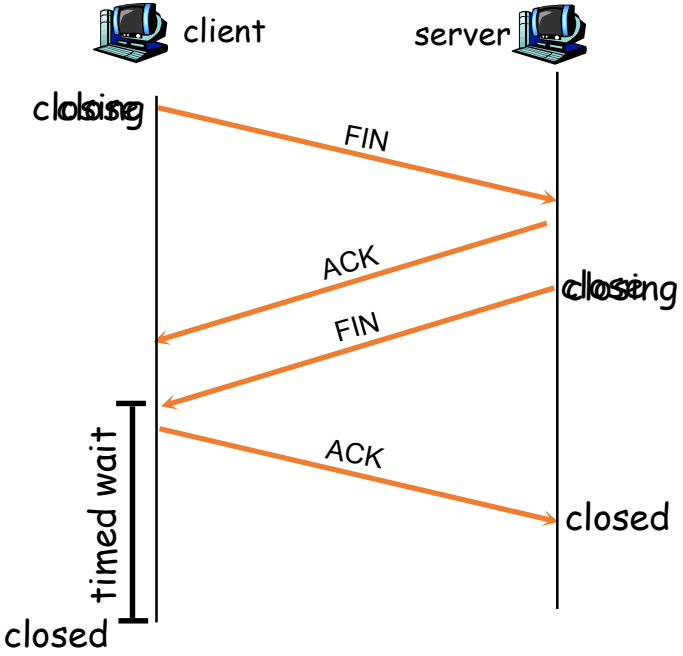
	TCP	GBN	SR
确认方式	累积确认	累积确认	单独确认
确认序号	期望序号的确认	正确接收分组最高序号的确认	对接收分组序号的确认
重传定时器数量	1个，发出未被确认的最小序号报文段	1个，发出未被确认的最小序号分组	每个发出未被确认的分组分别有一个定时器
超时重传机制	只重发定时器超时的报文段	所有发出未被确认的分组均重发	只重发定时器超时的分组
快速重传机制	有	无	无
失序报文处理	缓存/丢弃	丢弃	缓存

TCP的流量控制和连接管理*

- **TCP流量控制**
  - **速率匹配服务，发送方发送速度不会淹没接收方，针对特定的收发双方**
  - **通过TCP首部中的接收窗口字段实现**
- **TCP连接建立，三次握手**



- TCP连接断开，四次握手



流量控制和拥塞控制*

- 本质上都是控制发送方的发送速率
- 流量控制是一种速率匹配服务，发送方发送速度不会淹没接收方，针对特定发送方和接收方，局部概念
- 拥塞控制是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，需要源主机降低发送速率，控制网络流量，全局概念

TCP的拥塞控制机制*

- 端到端的拥塞控制和网络辅助的拥塞控制
- 当 Cwnd 低于阈值, 发送方处于慢启动阶段, 窗口指数增长.
- 当 Cwnd 高于阀值, 发送方处于拥塞避免阶段, 窗口线性增长.
- 当三个冗余的ACK 事件出现时,阈值置为Cwnd/2 并且Cwnd 置为 Cwnd/2 +3MSS.发送方处于拥塞避免阶段(TCP Reno)
- 当超时事件发生时 ， 阈值置为Cwnd/2 并且Cwnd 置为1 MSS.发送方进入慢启动阶段

TCP Tahoe协议：Cwnd直接置为1MSS，进入慢启动阶段

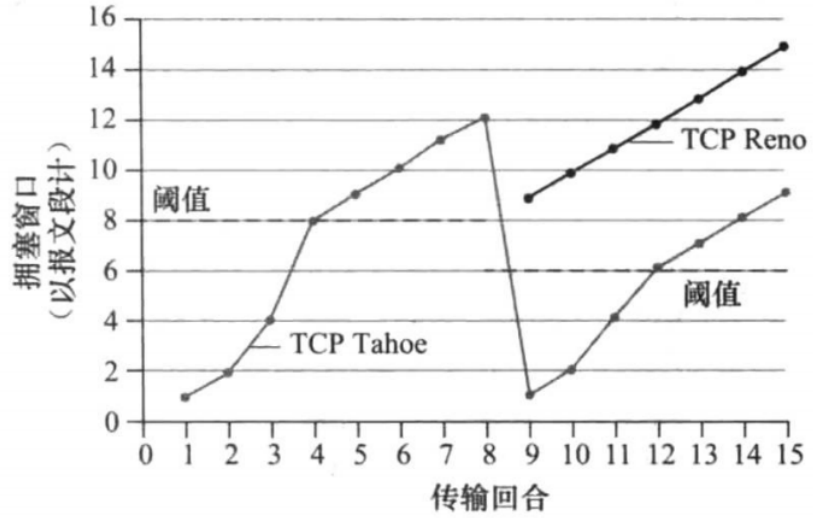


图 3-52　TCP 拥塞窗口的演化（Tahoe 和 Reno）

作业

- R9，R15，P3，P24,P27,P37,P40

涉及计算-校验和

- 给出二进制数据计算校验和
  - Tips：二进制序列按16bit对齐，末位不足16比特补0
    二进制加法，有进位则回卷
    最后的商求反则为校验和
- 给出校验后的数据如何进行校验
  - Tips：对数据的二进制序列同样计算校验和
    若不全为1则说明数据传输出错
    若全为1则说明未检测处错误，不能断定无错

涉及计算-TCP RTT

- **EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT**

DevRTT = (1-β)*DevRTT +
β*|SampleRTT-EstimatedRTT|

定时器**TimeoutInterval = EstimatedRTT + 4*DevRTT**

涉及计算-TCP 拥塞窗口

- 理解阈值threshold，当前拥塞窗口Cwnd在threshold下方，则处于慢启动阶段，Cwnd在每个RTT周期后指数增长；
- Cwnd大于等于threshold，则处于拥塞避免阶段，Cwnd在每个RTT周期线性增长
- 出现丢失事件，阈值一律设置为当前拥塞窗口的一半，threshold=Cwnd/2
- 若该事件为收到3个冗余ACK判断的丢失，Cwnd减半并且加3个MSS（对于TCP Reno），仍为拥塞避免阶段
- 若该事件是超时，Cwnd=1MSS，进入慢启动阶段

涉及计算-TCP 拥塞窗口

- 能从拥塞窗口变化图形指明慢启动和拥塞避免阶段的时间周期
- 通过特殊点判断出现何种事件（如Cwnd减半等）
- 能计算每个RTT周期发送的MSS数量