第3章 运输层（传输层）

第3章:运输层

目的:

- 理解运输层服务的原理:
  - 多路复用/多路分解
  - 可靠数据传输
  - 流量控制
  - 拥塞控制
- 运输层协议:
  - UDP 面向无连接的传输
  - TCP 面向可靠连接的传输
  - TCP 拥塞控制

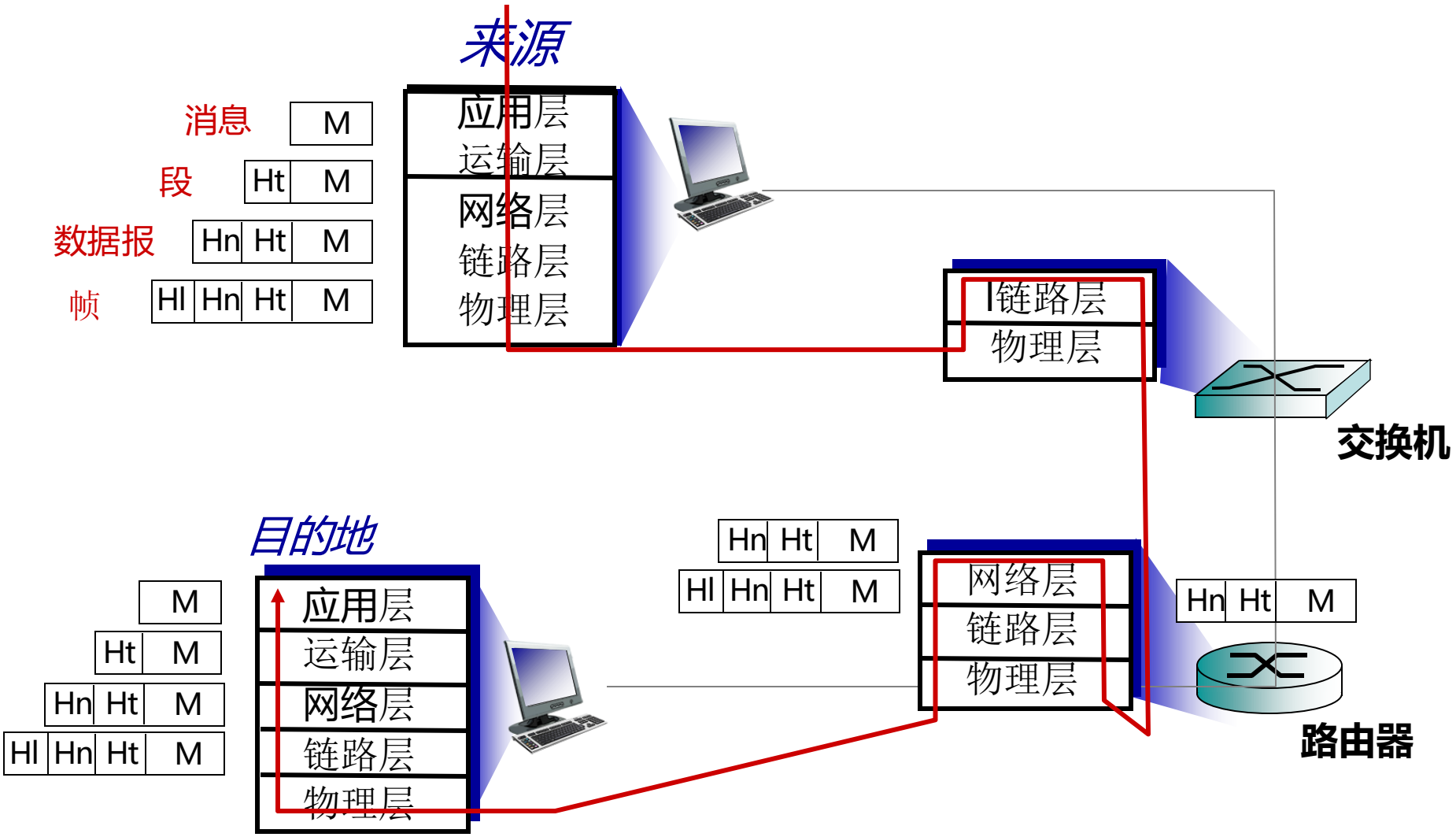
第3章:大纲

1. **3.1运输层服务**
2. 3.2多路复用和多路分解
3. 3.3无连接传输:UDP
4. 3.4可靠传输原理
5. 3.5面向连接的传输:TCP

1.分段结构　2.可靠的数据传输　3.流量控制　4.连接管理

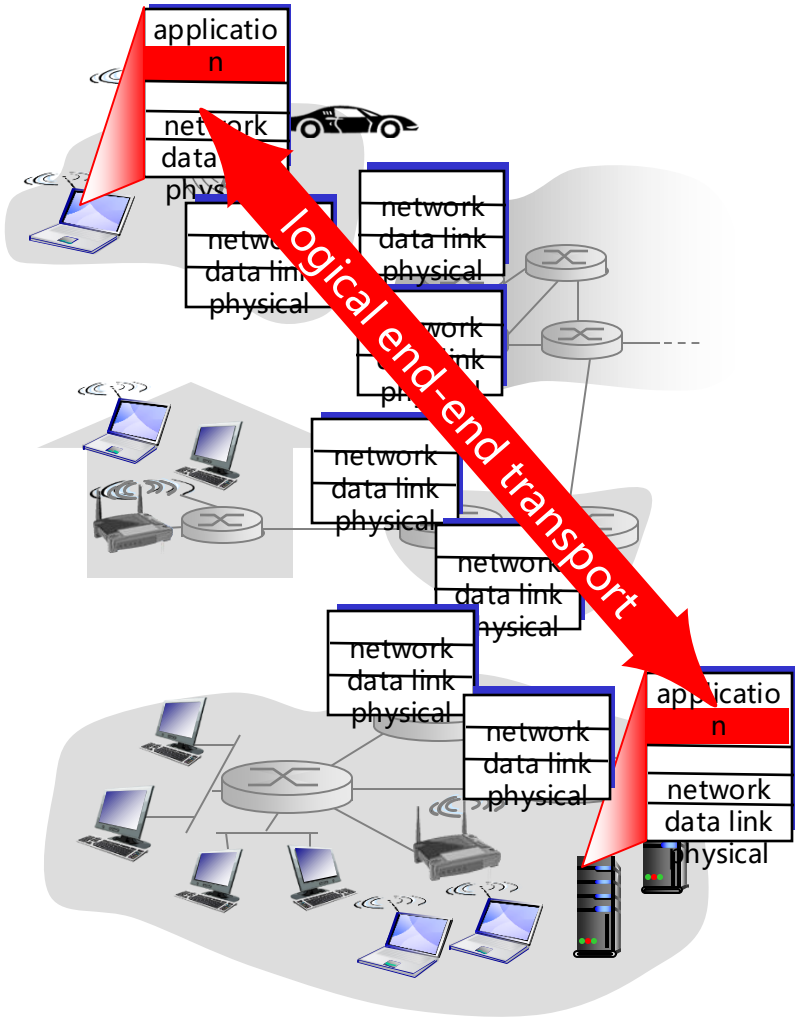
6. 3.6拥塞控制原理
7. 3.7 TCP的拥塞控制

回顾


来源
消息
M
段
Ht
M
数据报
Hn
Ht
M
帧
Hl
Hn
Ht
M
应用层
运输层
网络层
链路层
物理层
链路层
物理层
交换机
目的地
M
Ht
M
Hn
Ht
M
Hl
Hn
Ht
M
应用层
运输层
网络层
链路层
物理层
Hn
Ht
M
Hl
Hn
Ht
M
网络层
链路层
物理层
Hn
Ht
M
路由器

运输层服务

- 在不同主机上运行的应用程序进程之间提供逻辑通信
- 传输协议在终端系统中运行
  - 发送端将应用层报文分解成数据报传至网络层
  - 接收端重组数据报为应用层报文并传至应用层
- 多个传输协议可供应用程序使用
  - 互联网:TCP和UDP



运输层与网络层

- *网络层:*主机之间的逻辑通信
- *运输层:*进程间的逻辑通信
  - 依赖并增强网络层服务

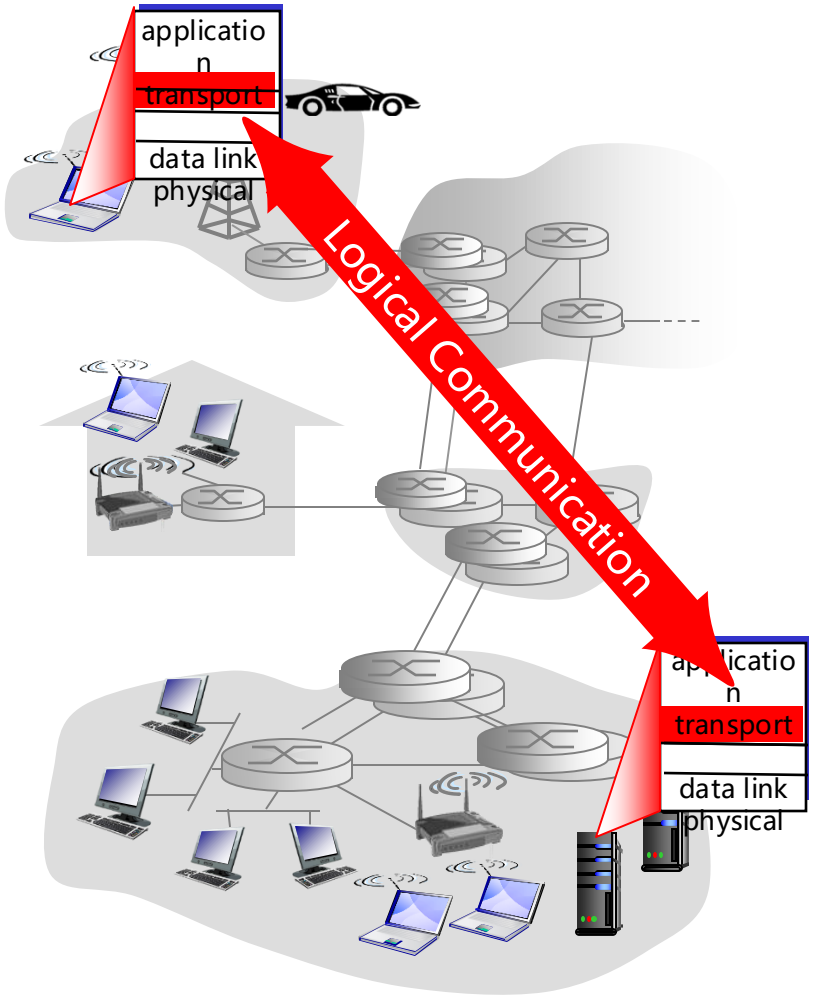
家庭类比:

Ann的12个孩子给Bill的12个孩子送礼物:

- 主机=家庭
- 进程=孩子
- 应用程序报文=包裹中邮件
- 运输层协议= Ann和Bill
- 网络层协议=邮政服务

互联网网络层协议

- 互联网协议(IP)
- “尽最大努力交付”
  - 不可靠的服务
  - 不保证分段交付
  - 不保证有序交付
  - 不保证数据的完整性
- *数据报:*网络层中数据包的名称



第3章:大纲

1. 3.1 运输层服务
2. **3.2 多路复用和多路分解**
3. 3.3 无连接传输:UDP
4. 3.4 可靠运输原理
5. 3.5 面向连接的传输:TCP

1.分段结构　2.可靠的数据传输　3.流量控制　4.连接管理

6. 3.6 拥塞控制原理
7. 3.7 TCP的拥塞控制

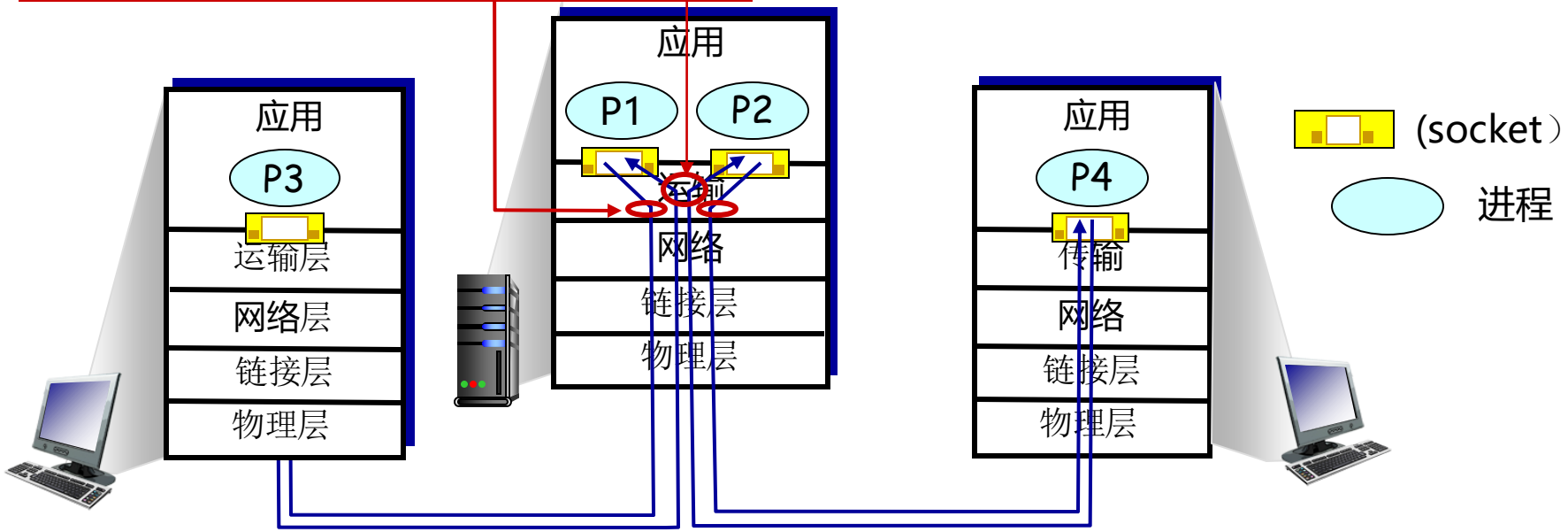
复用/解复用

复用发生在发送端

将来自多个socket的的数据，添加运输层头（用于分解）

分解发生在接收端:

利用运输层头部信息将接收到的报文段交付给正确的socket



多路分解的工作原理

- 主机使用IP地址和端口号将数据报定向到适当的套接字
- 主机接收IP数据报
  - 每个数据报都有源IP地址、目的IP地址
  - 每个数据报携带一个运输层报文段
  - 每个数据段都有源端口号、目的端口号

TCP/UDP数据段格式

无连接解复用

- 创建有主机本地端口的套接字#:

Datagram socket mySocket 1 = new DatagramSocket(12534)；

DatagramPacket mypacket1 =new DatagramPacket(data，data.length, …)

- 当创建要发送到UDP套接字的数据报时，必须指定
  - 目的地IP地址
  - 目的地端口

- 当主机收到UDP数据段时:
  - 检查段中的目的端口号
  - 将UDP段定向到具有该端口#的套接字

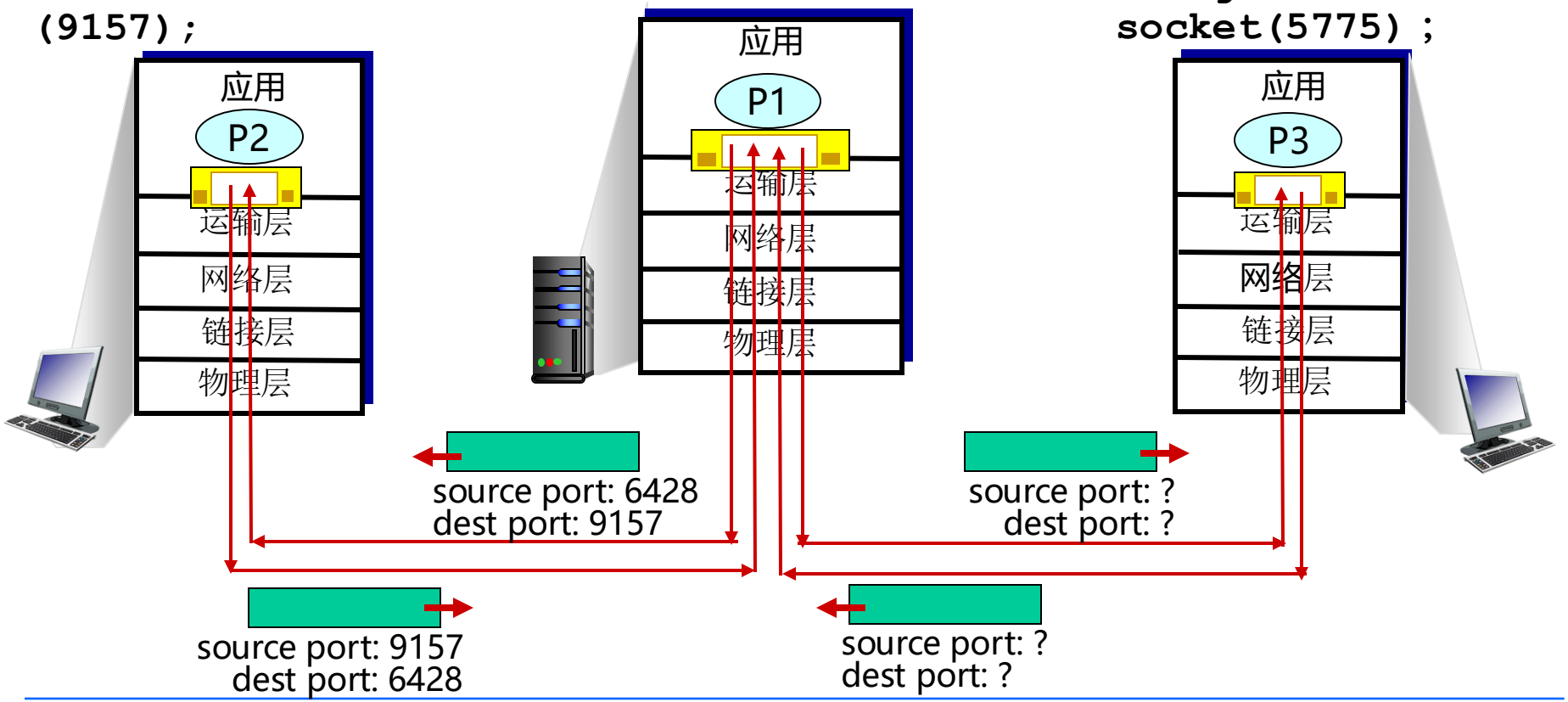
具有相同目的地的IP数据报。端口号，但不同的源IP地址和/或源端口号将被定向到目的地的同一套接字

无连接多路分解器:示例


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

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

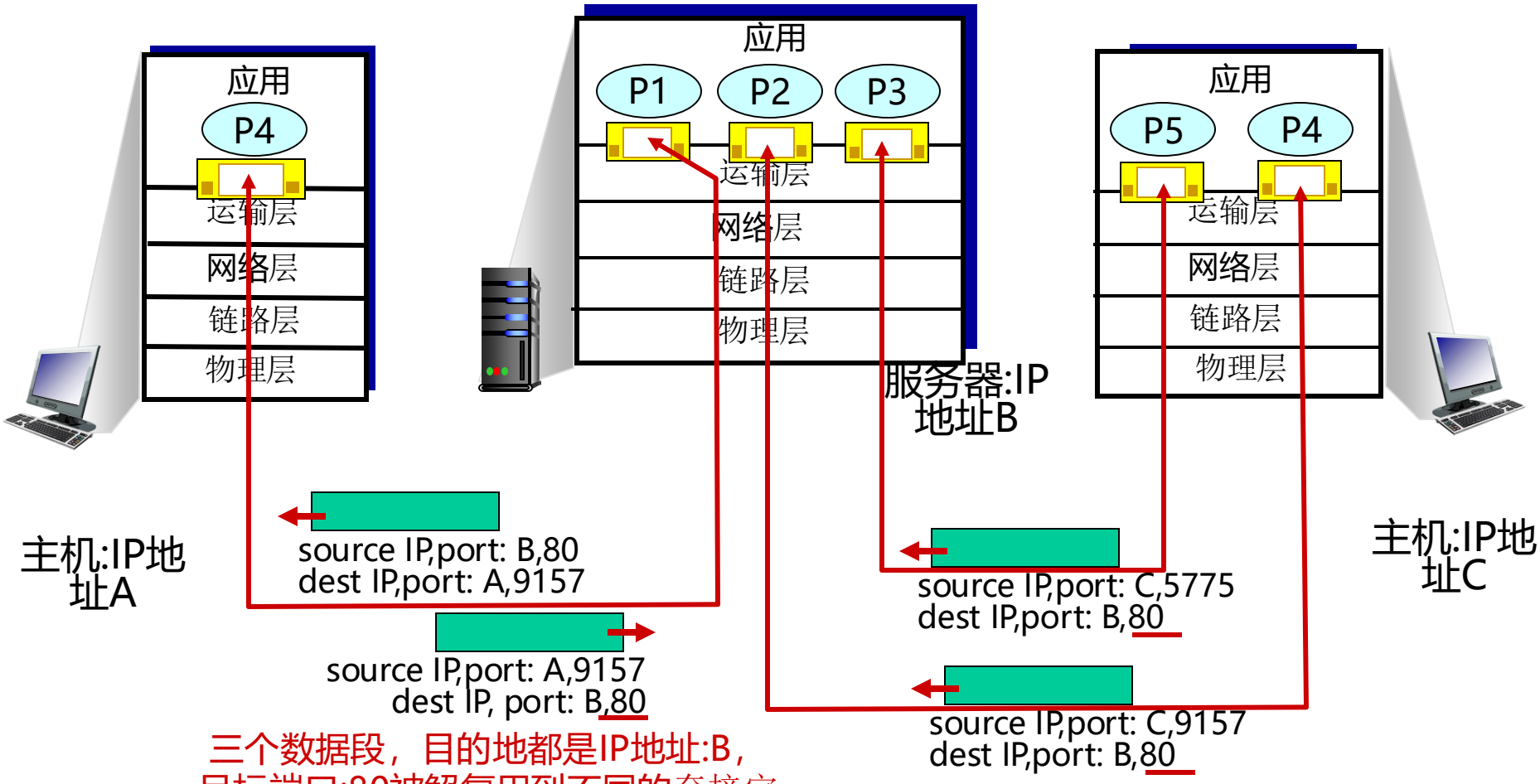
```
datagram socket my
 socket 1 = new
 datagram
 socket(5775) ;
```



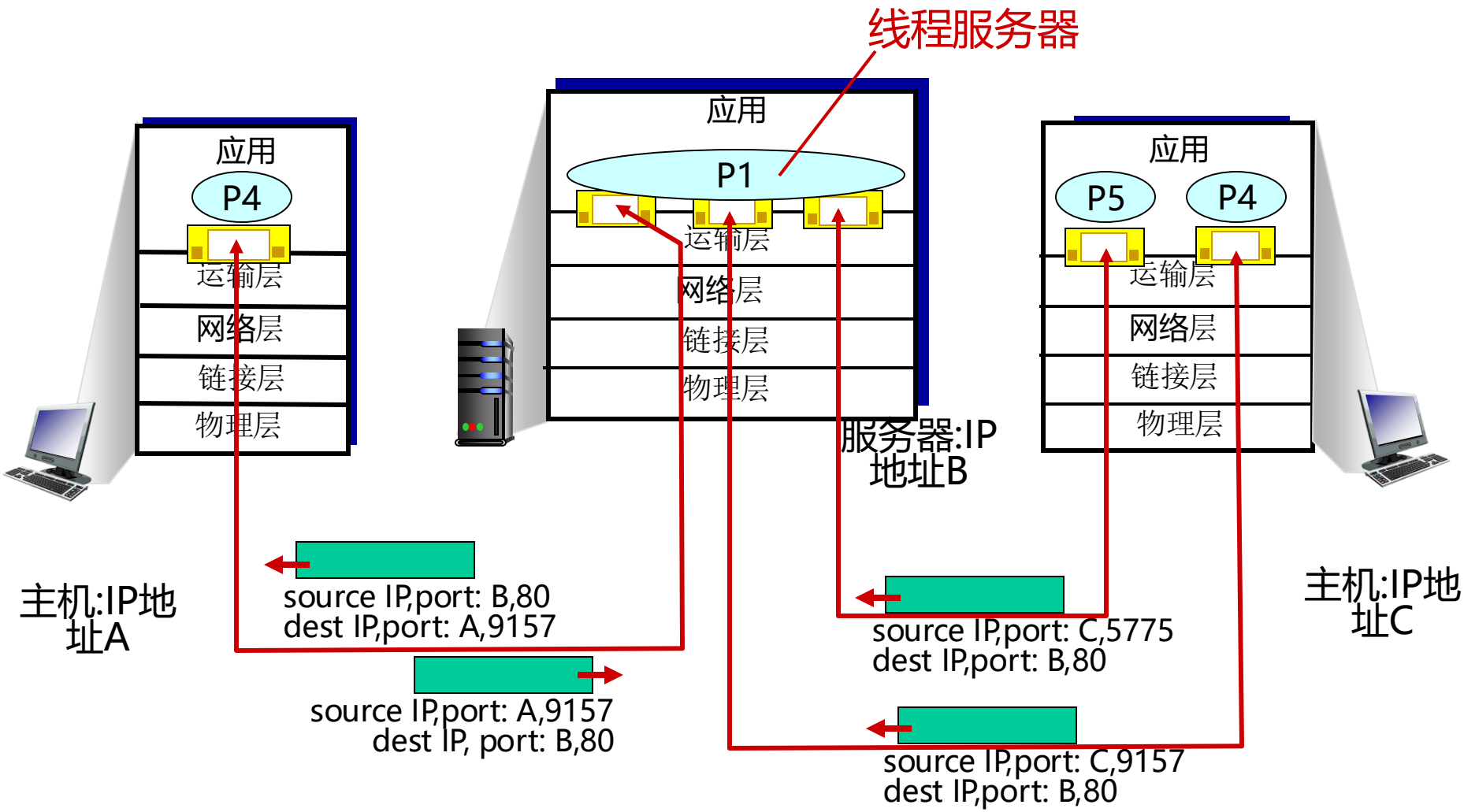
面向连接的多路分解器

- 由4元组标识的TCP套接字:
  - 源IP地址
  - 源端口号
  - 目标IP地址
  - 目标端口号
- 多路分解:接收器使用所有四个值将数据段定向到适当的套接字
- 服务器主机可能同时支持许多TCP套接字:
  - 每个套接字由其自己的4元组来标识
- Web服务器对于每个连接的客户端都有不同的套接字
  - 非持久性HTTP将为每个请求提供不同的套接字

面向连接的多路分解:示例



面向连接的多路分解:示例

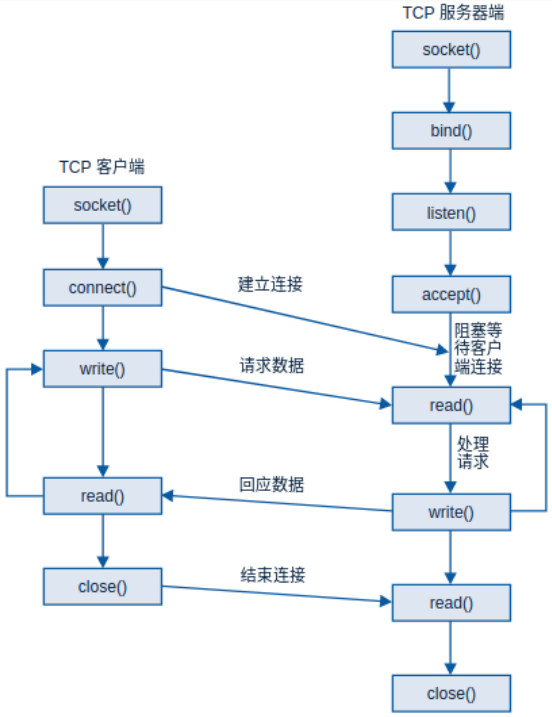


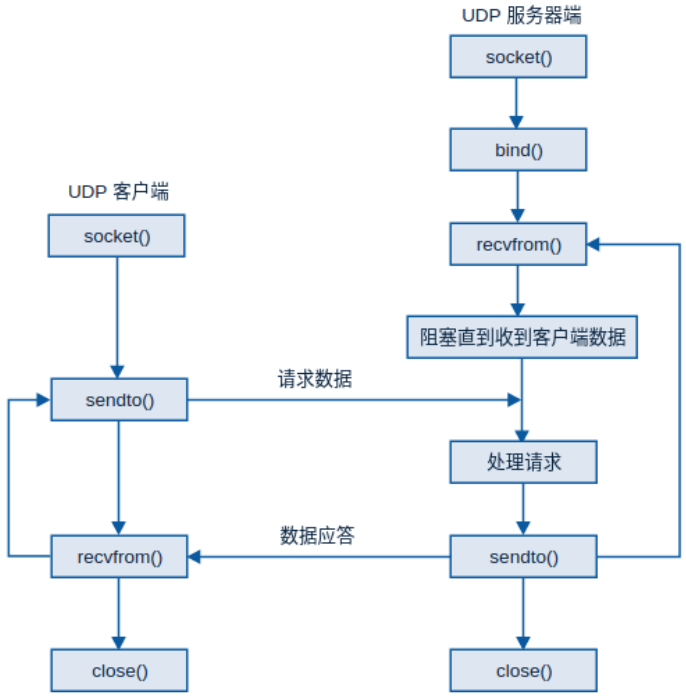
第3章:大纲

1. 3.1运输层服务
2. 3.2多路复用和多路分解
3. **3.3无连接传输:UDP**
4. 3.4可靠运输原理
5. 3.5面向连接的传输:TCP

1.分段结构　2.可靠的数据传输　3.流量控制　4.连接管理

6. 3.6拥塞控制原理
7. 3.7 TCP的拥塞控制





ssize_t **sendto**(**int** sockfd, **const void** ***buf**, **size_t** len, **int** flags, **const struct sockaddr** ***dest_addr**, socklen_t addrlen);

ssize_t **write**(int fd, const void*buf, size_t nbytes);

UDP：用户数据报协议 [RFC 768]

- 简单协议:
  - 基本功能
  - 错误检测
- “尽力而为”服务，UDP数据段可能是:
  - 丢失，错误
  - 无序交付
- *无连接:*
  - UDP发送方、接收方之间没有握手
  - 每个UDP数据段独立于其他数据段进行处理

- UDP用于:
  - 流媒体应用(允许丢失、但是传输速率敏感)
  - DNS，SNMP
- UDP上的可靠传输:
  - 在应用层增加可靠性
  - 尤其是出错重传机制！

UDP:报头

UDP数据段格式

UDP数据段的长度，以字节为单位，包括报头

为什么会有UDP?

- 没有连接建立(这会增加延迟)
- 简单:发送方、接收方没有连接状态
- 段首较短
- 没有拥塞控制:UDP可以按需要的速度快速传输

UDP校验和

*目标:*检测传输数据段中的“错误”(例如，翻转的bit位)

发送方:

- 将包括报头字段在内的数据段内容视为16位整数序列
- 校验和:段内容的相加(补码和)
- 发送方将校验和值放入UDP校验和字段

接收方:

- 计算接收数据段的校验和
- 检查计算的校验和是否等于校验和字段值:
  - 未检测到错误
  - 是-未检测到错误。但也许还是有错误?

互联网校验和:示例

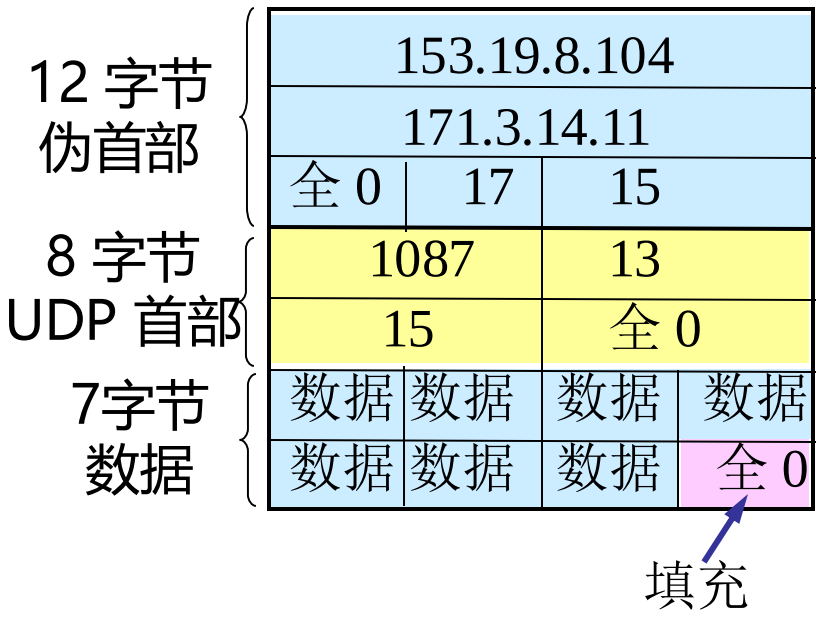
示例:将两个16位整数相加

sum + checksum = 1111111111111111111111111(如果有一位为0，则出错)

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
回卷操作	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
总和	1															
校验和	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	

注意:添加数字时，需要将最高有效位的进位加到结果中

互联网校验和:示例



10011001 00010011 → 153.19
00001000 01101000 → 8.104
10101011 00000011 → 171.3
00001110 00001011 → 14.11
00000000 00010001 → 0 和 17
00000000 00001111 → 15
00000100 00111111 → 1087
00000000 00001101 → 13
00000000 00001111 → 15
00000000 00000000 → 0(校验和)
01010100 01000101 →数据
01010011 01010100 →数据
01001001 01001110 →数据
01000111 00000000 →数据 和 0(填充)

按二进制反码运算求和 10010110 11101101 →求和得出的结果

将得出的结果求反码 01101001 00010010 →检验和

第3章:大纲

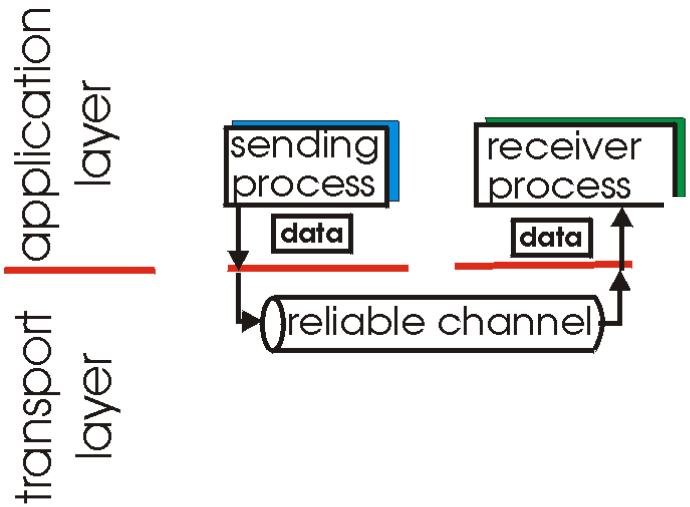
1. 3.1运输层服务
2. 3.2多路复用和多路分解
3. 3.3无连接传输:UDP
4. **3.4可靠运输原理**
5. 3.5面向连接的传输:TCP

1.分段结构　2.可靠的数据传输　3.流量控制　4.连接管理

6. 3.6拥塞控制原理
7. 3.7 TCP的拥塞控制

可靠数据传输的原则

- 在应用层、运输层、链路层非常重要
  - 10大重要网络课题之一!

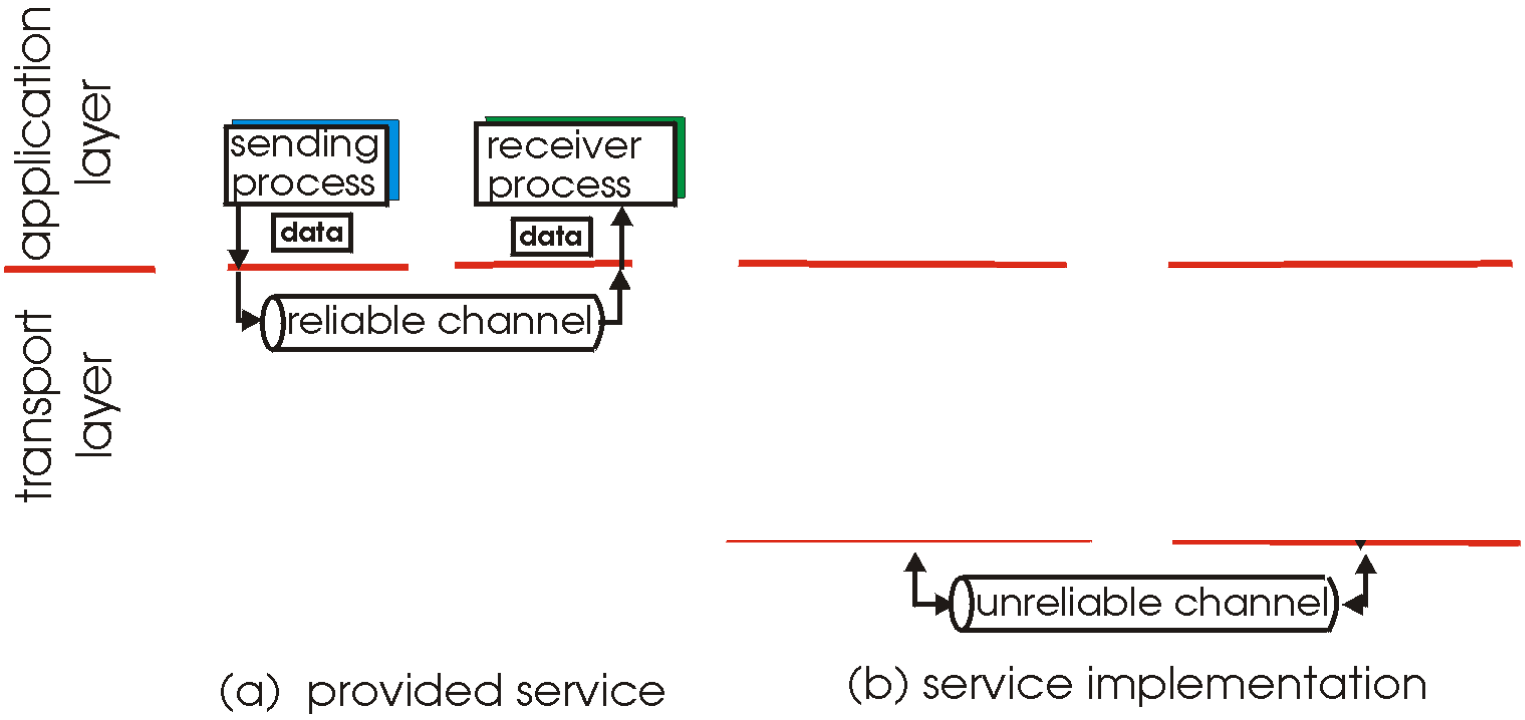


(a) provided service　　(b) service implementation

- 不可靠信道的特性将决定可靠数据传输协议的复杂性

可靠数据传输的原则

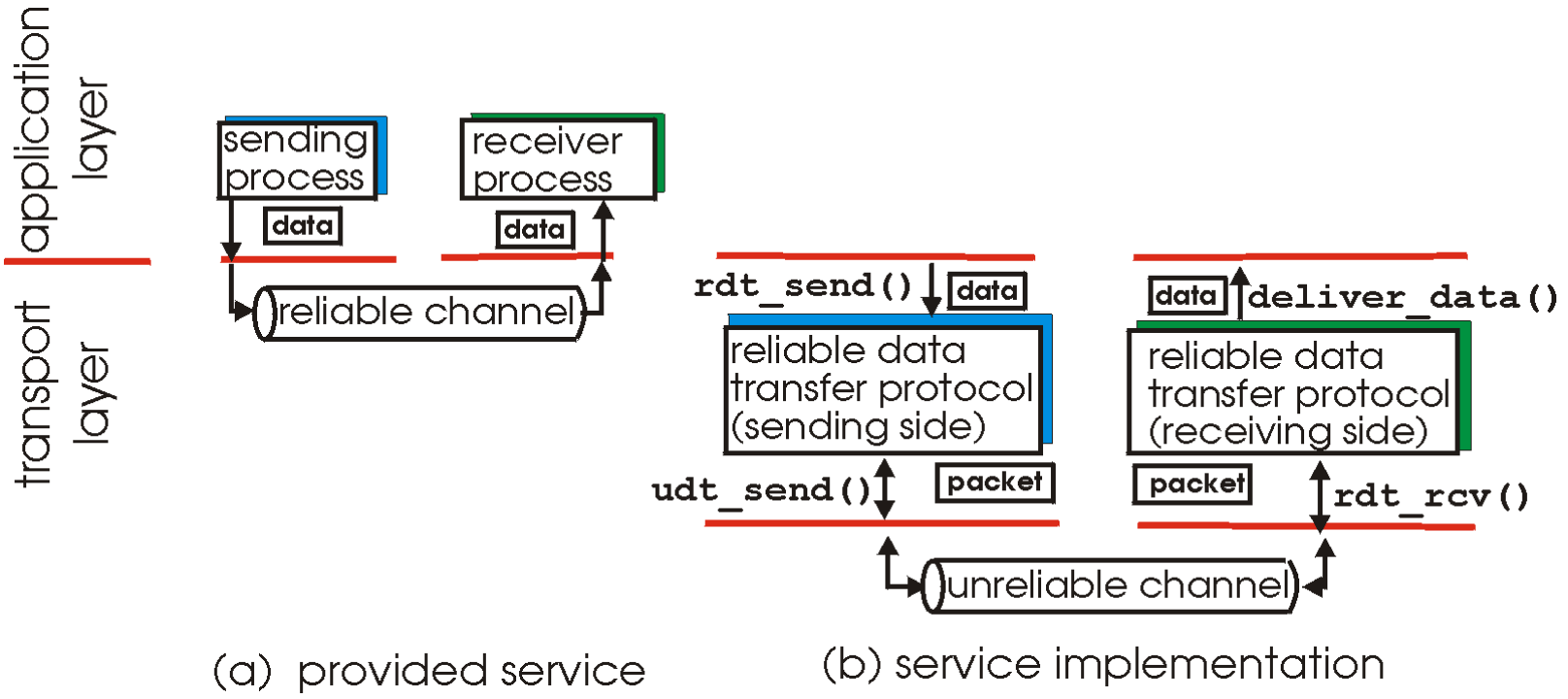
- 在应用层、运输层、链路层非常重要
  - 10大重要网络课题之一!



- 不可靠信道的特性将决定可靠数据传输协议的复杂性

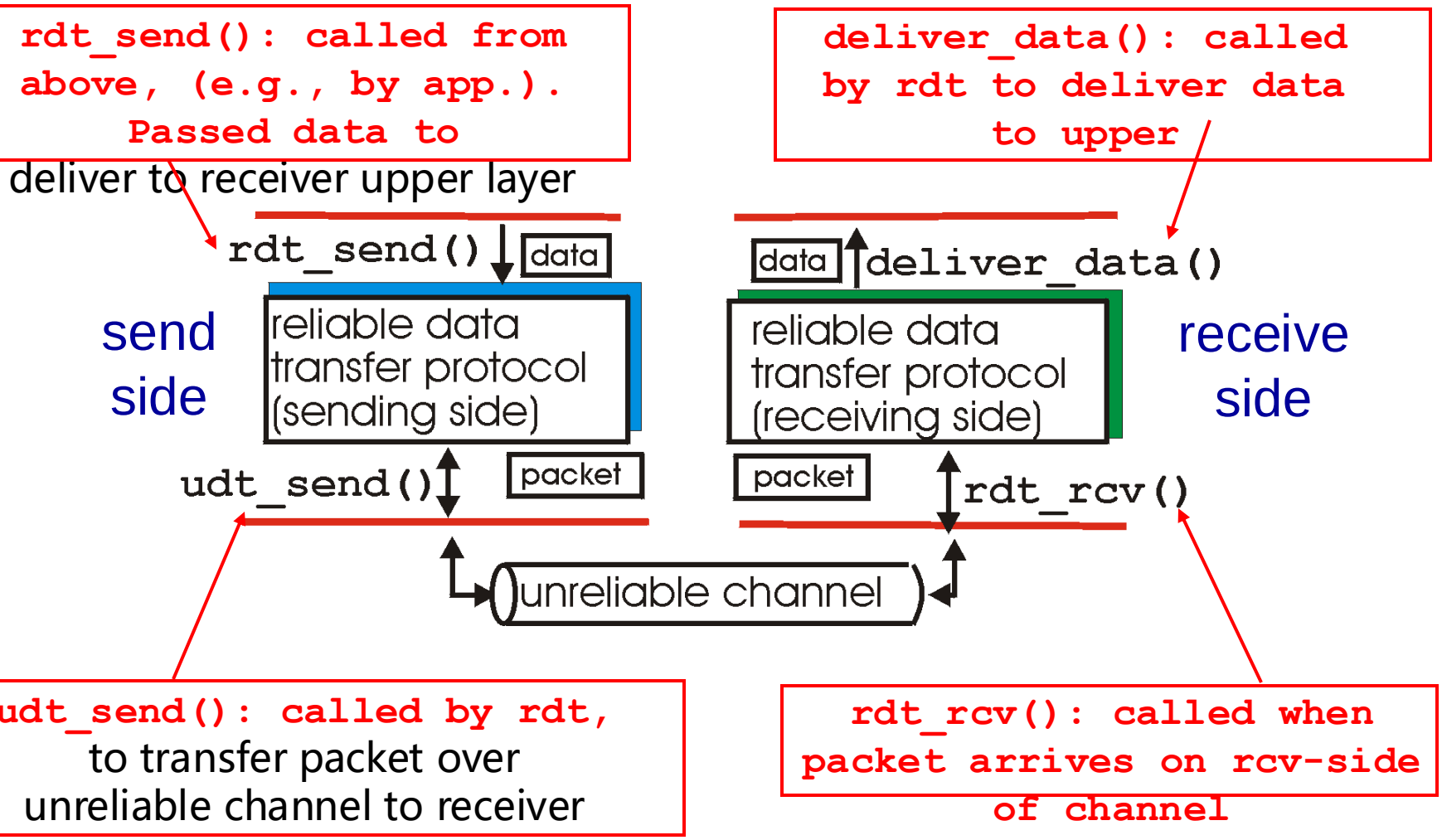
可靠数据传输的原则

- 在应用层、运输层、链路层非常重要
  - 10大重要网络课题之一!



- 不可靠信道的特性将决定可靠数据传输协议的复杂性

可靠的数据传输:入门

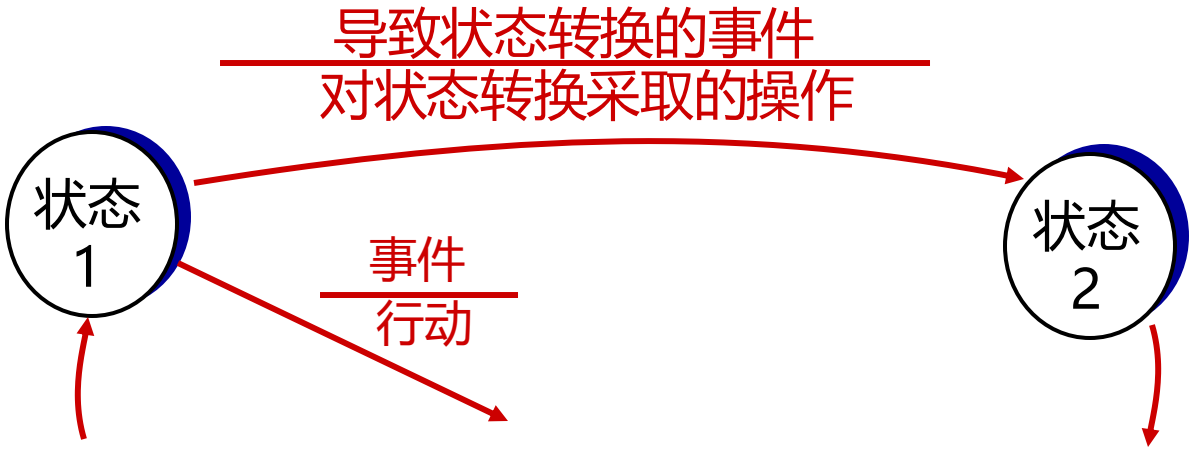

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
deliver_data(): called by rdt to deliver data to upper
rdt_send()
data
send side
reliable data transfer protocol (sending side)
udt_send()
packet
unreliable channel
packet
reliable data transfer protocol (receiving side)
data
deliver_data()
receive side
rdt_rcv()
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver
rdt_rcv(): called when packet arrives on rcv-side of channel

可靠的数据传输:入门

我们将依循软件开发范式:

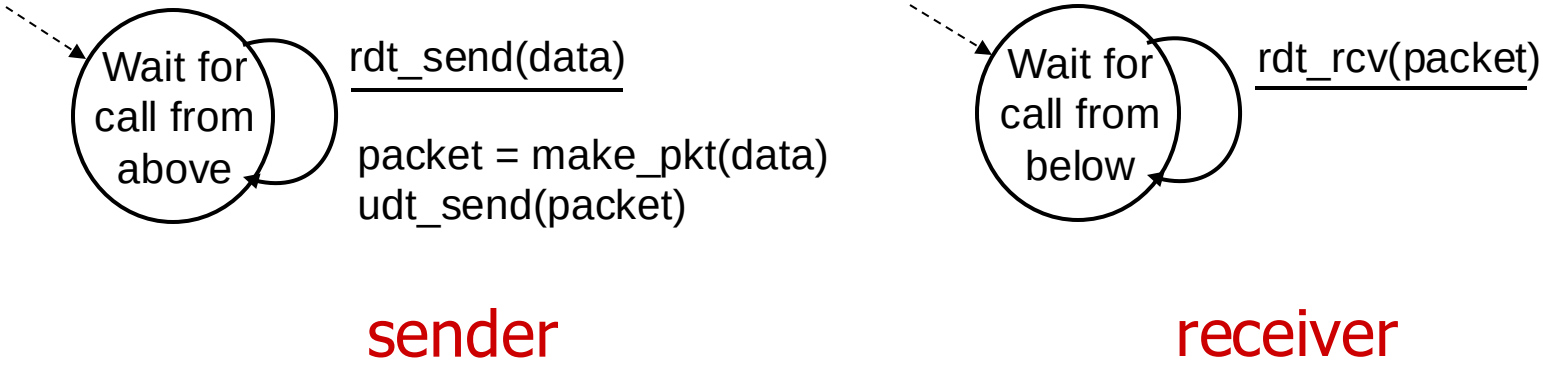
- 逐步开发可靠数据传输协议(rdt)的发送方和接收方
- 仅考虑单向数据传输
  - 但是控制信息将会双向流动!
- 使用有限状态机(FSM)来指定发送者、接收者

状态:当处于此“状态”时，下一个状态由下一个事件唯一确定



rdt1.0:通过可靠信道进行可靠传输

- 底层信道完全可靠
  - 没有bit错误
  - 没有数据包丢失
- 发送方、接收方的独立FSM:
  - 发送方将数据发送到可靠的信道
  - 接收器从可靠的信道里读取数据

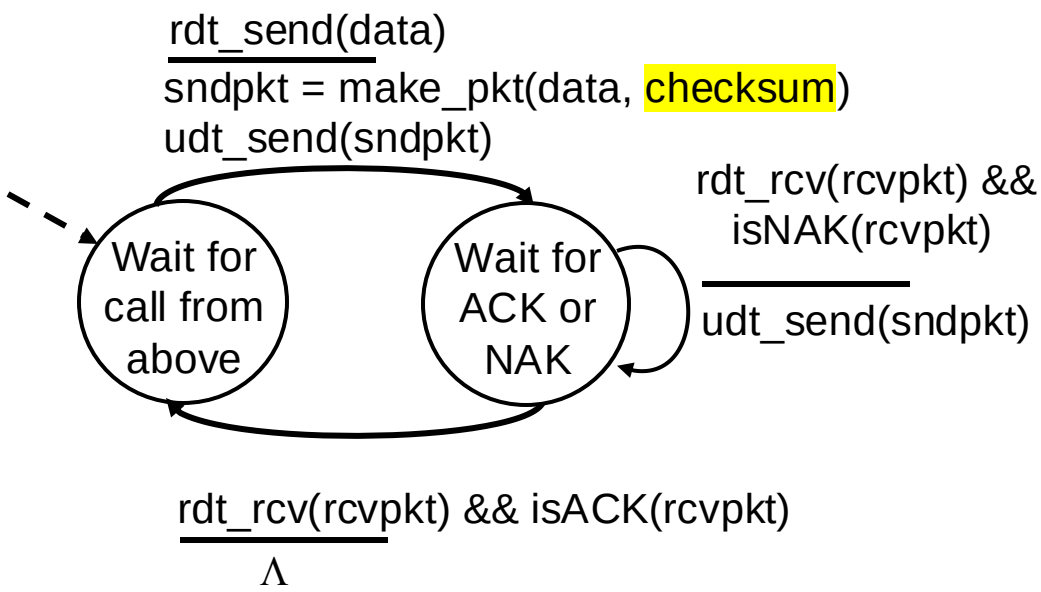


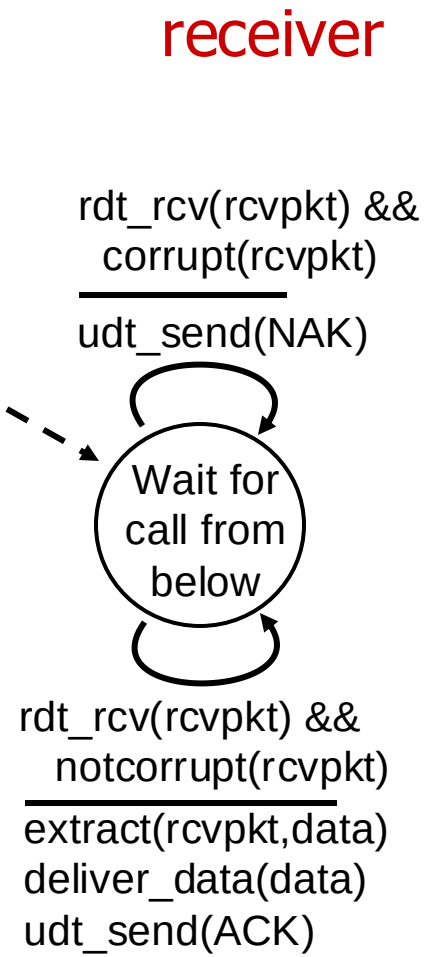
rdt2.0:信道存在bit差错

- 信道可能会翻转数据包中的bit位
  - 检测bit位错误的校验和
- *问题:如何从错误中恢复:*
  - *确认(ACK):*接收方明确告诉发送方pkt已收到
  - *否定确认(NAK):*接收方明确告诉发送方pkt有错误
  - 发送方在收到NAK时重新传输pkt
- rdt2.0中的新机制(超越rdt1.0):
  - 错误检测
  - 接收器反馈:控制消息(ACK，NAK) 接收端>发送段
  - 重传

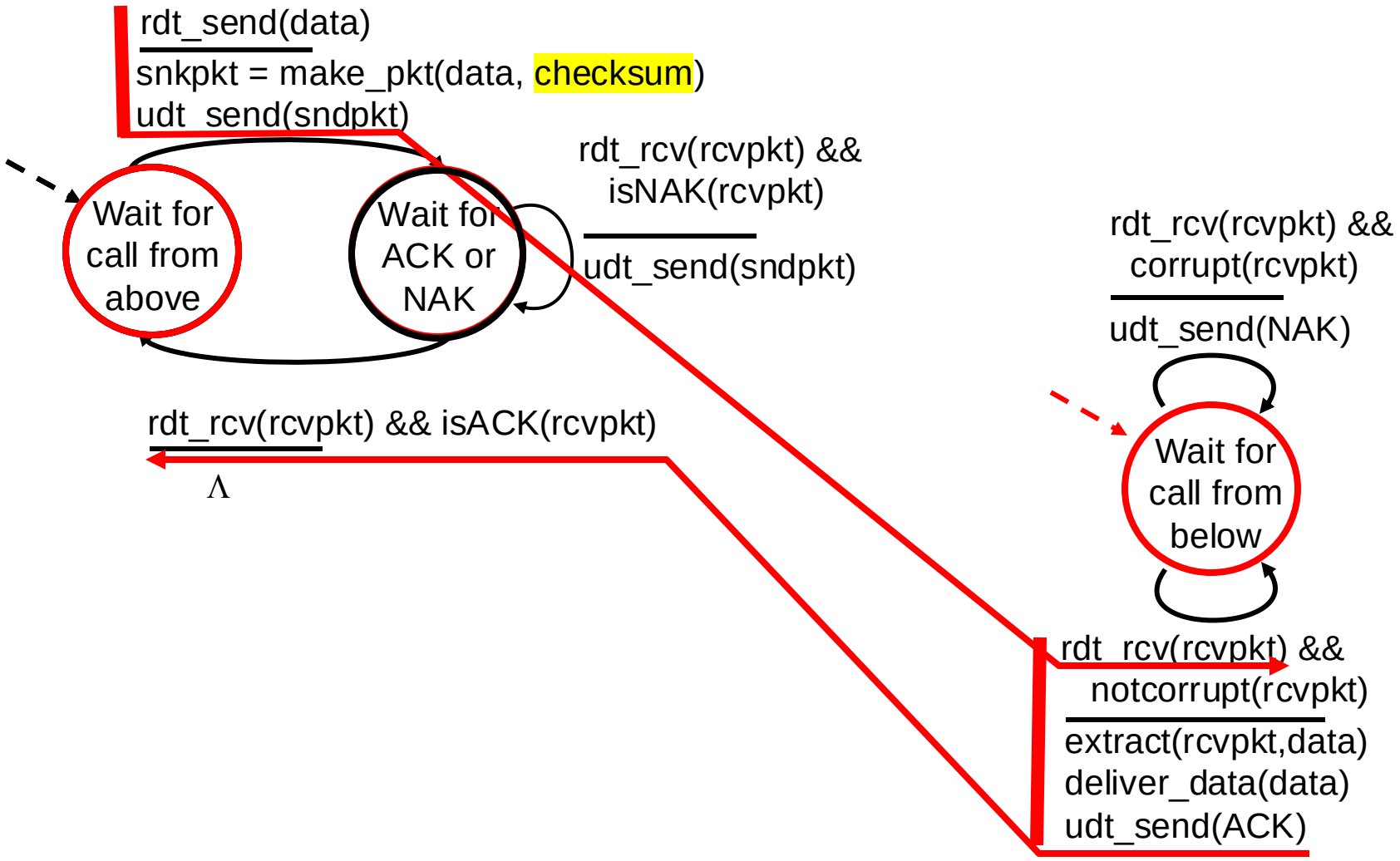
人类如何从“错误”中恢复
在谈话过程中?

rdt2.0:有限状态机描述

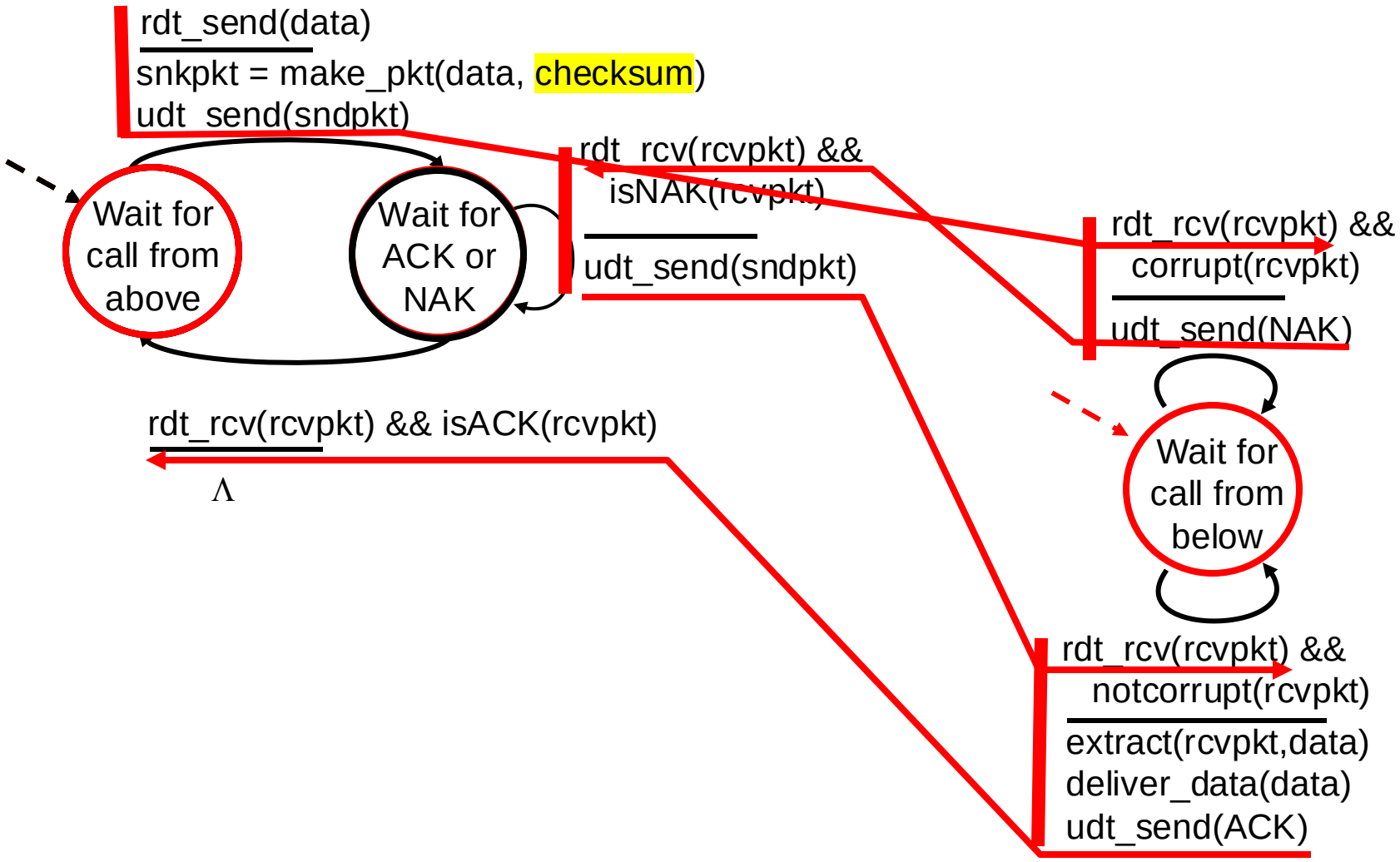




rdt2.0:无错


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)

rdt2.0:有错


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) && corrupt(rcvpkt)
udt_send(NAK)
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ
Wait for call from below
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0有一个致命缺陷！

如果ACK/NAK出错会发生什么?

- 发送者不知道接收者发生了什么！
- 不能只是重传:可能重复

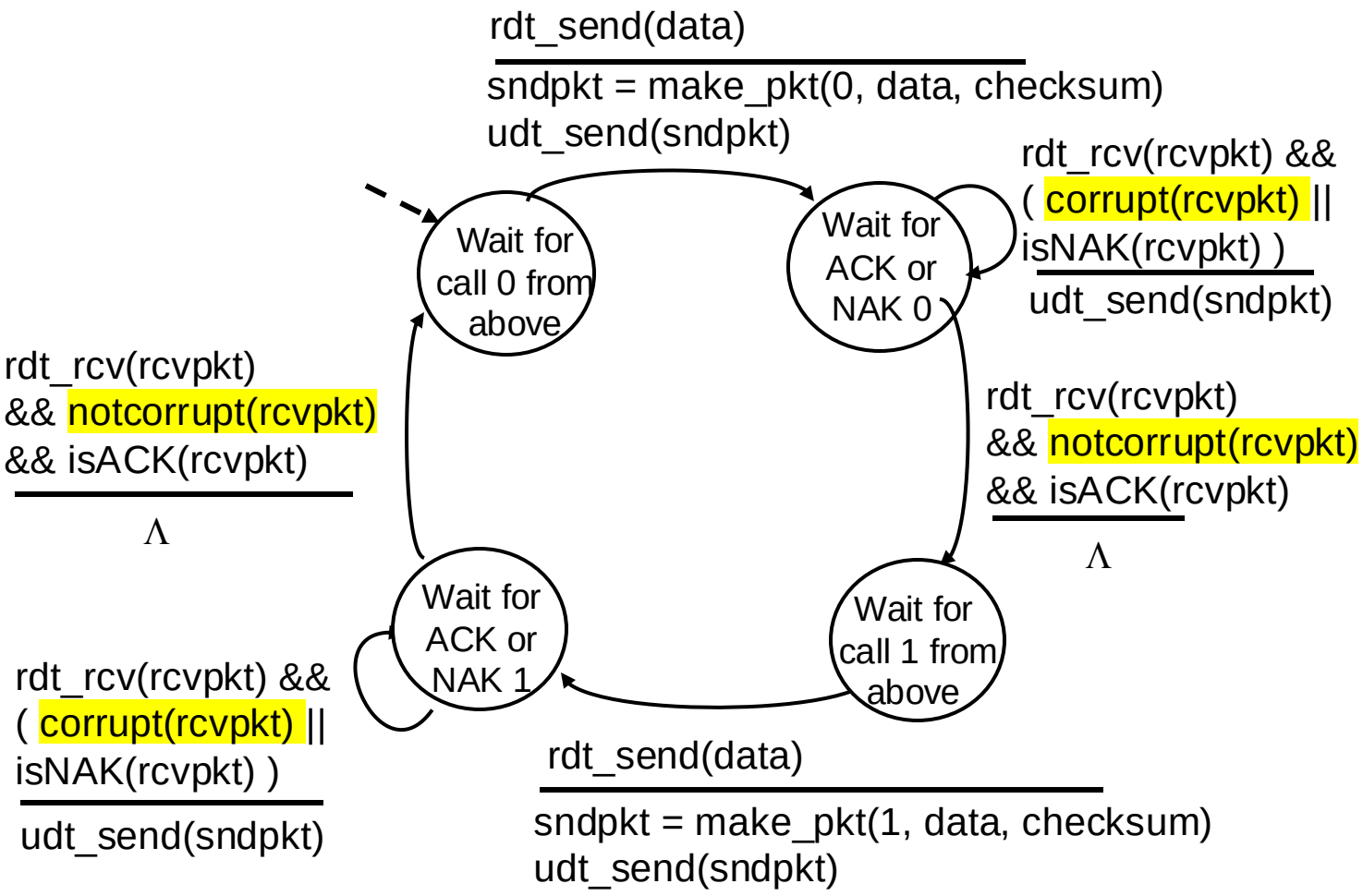
停止-等待（停等）协议

•发送端发一个分组，然后等待接收方的响应

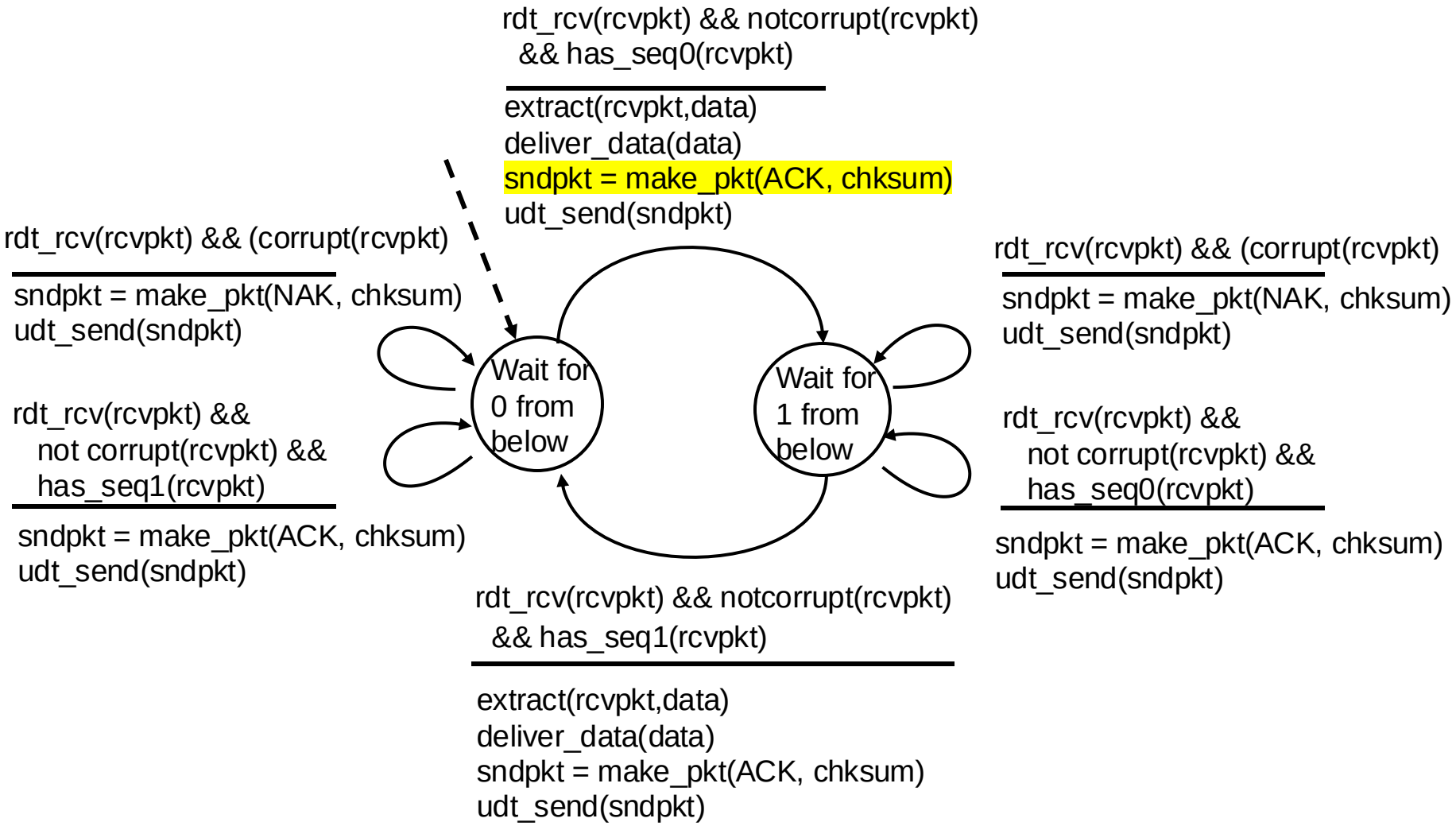
处理重复项:

- ACK/NAK 加入Checksum
- 发送方给每个分组加上sequence number （序号）
- 如果ACK/NAK出错，发送方则重传正确的分组
- 接收方丢弃重复的分组 (不向上交付)

rdt2.1:发送方


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK or NAK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
Wait for ACK or NAK 1
Wait for call 1 from above
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
udt_send(sndpkt)
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

rdt2.1:接收器


rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
Wait for
0 from
below
Wait for
1 from
below
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt2.1:讨论

发送方:

- 将序号添加到pkt
- 两个序列。#(0，1)就够了。为什么?
- 两倍状态
  - 状态必须“记住”是否“正确的”分组具有序号0或1。
- 必须检查收到的ACK/NAK是否损坏

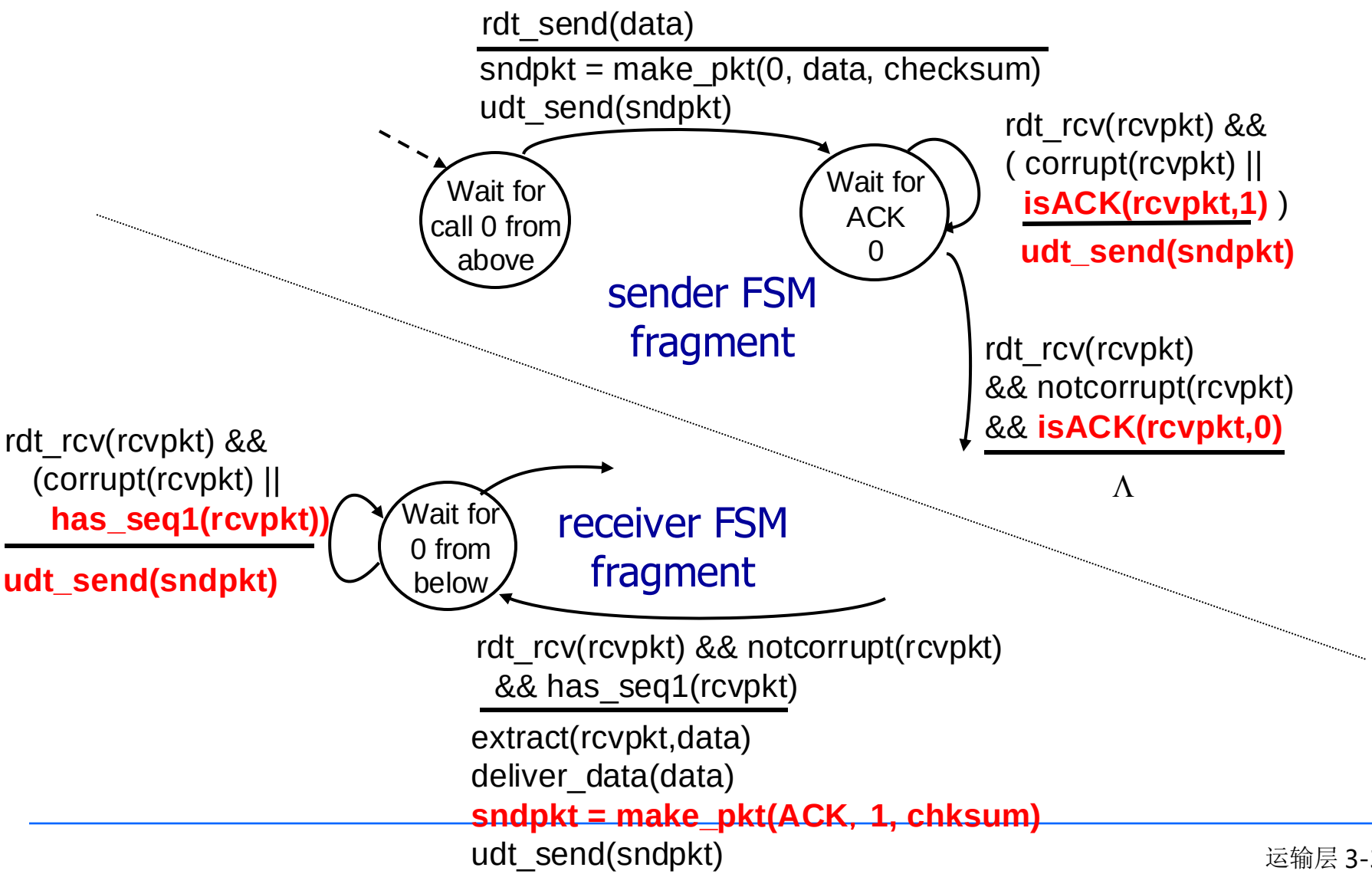
接收方:

- 注意:接收方不知道它的最后一个ACK/NAK在发送方是否收到
- 必须检查收到的数据包是否重复
  - 状态指示0或1是否是预期的序列号

rdt2.2:一种无NAK协议

- 功能与rdt2.1相同，仅使用ACK
- 接收器发送ACK而不是NAK，表示最后一个收到的pkt正常
  - 接收方必须明确包括被确认的数据包的序列号
- 发送方的重复ACK导致与NAK相同的操作:重传正确的分组

rdt2.2:发送与接收


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
udt_send(sndpkt)
sender FSM fragment
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
Λ
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))
udt_send(sndpkt)
Wait for 0 from below
receiver FSM fragment
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, 1, chksum)
udt_send(sndpkt)

rdt3.0:有错误和丢包

新的假设:

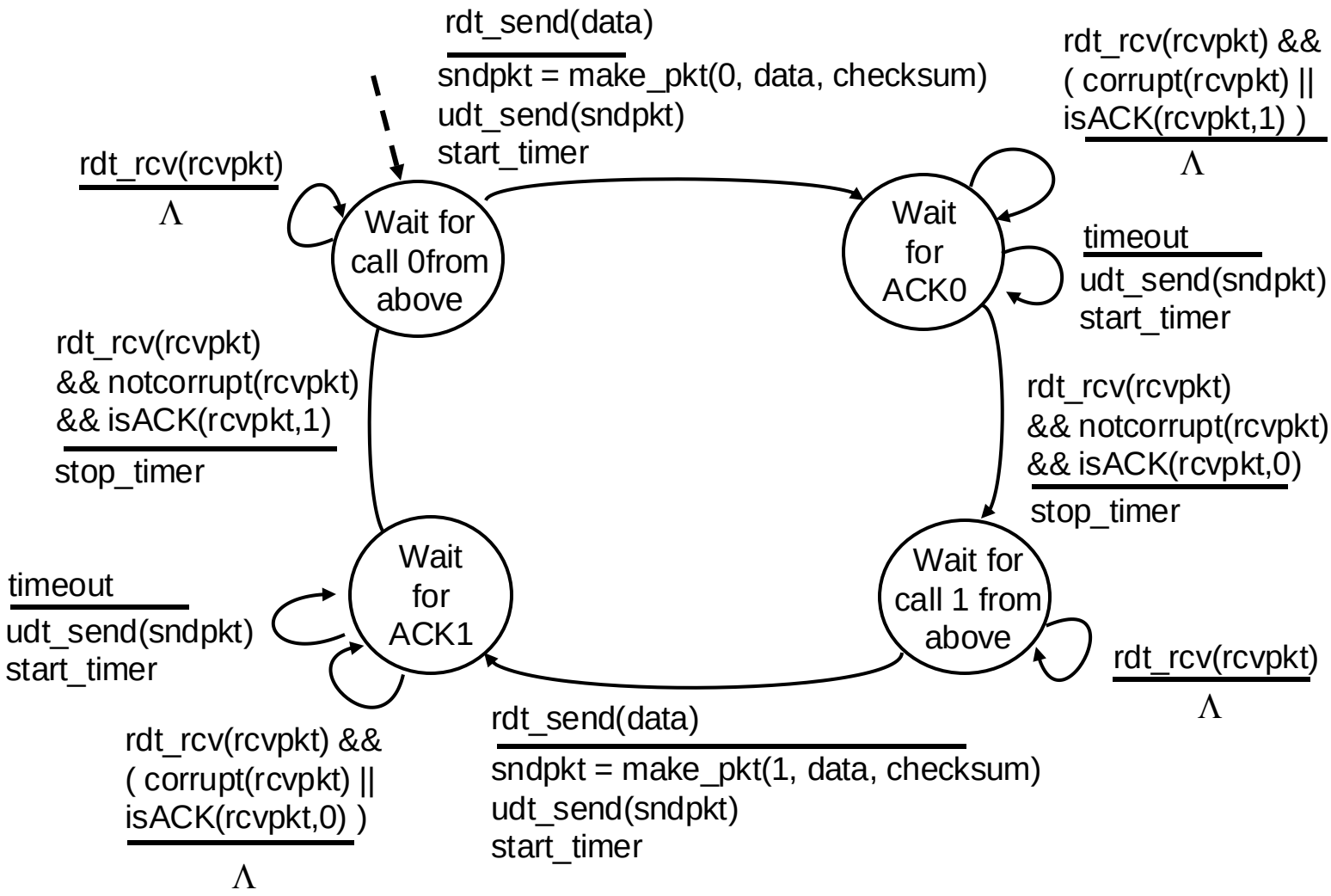
底层信道也可能丢包(数据或ACK)

- 校验和，序号、ACK、重新传输会有帮助...但还不够

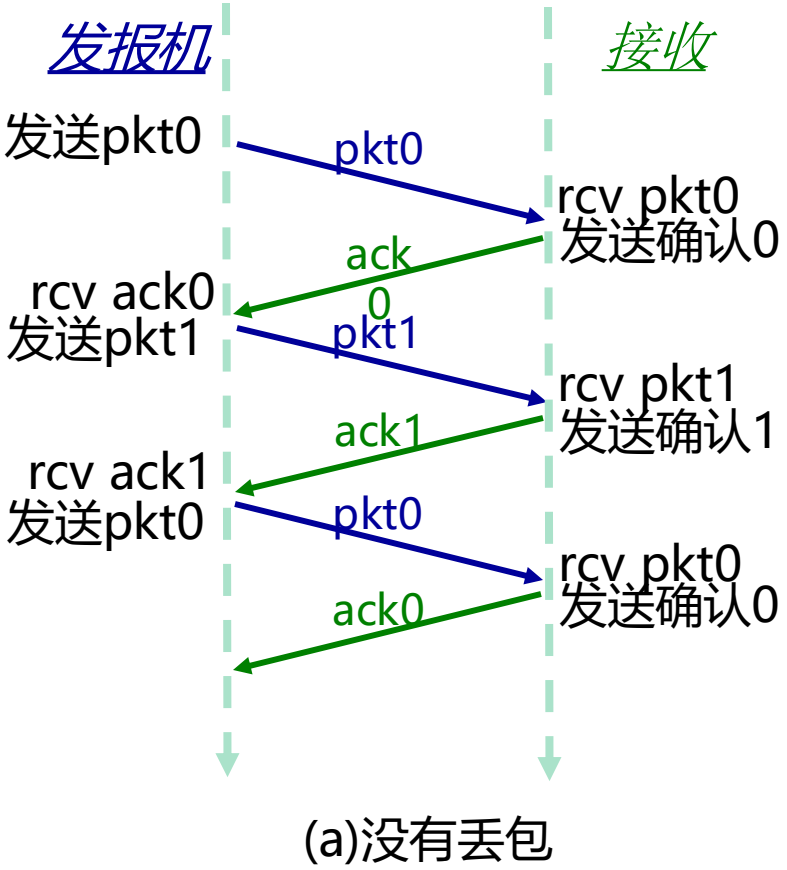
方法:发送方等待“合理的”时间来确认

- 需要计时器
- 如果此时没有收到ACK，则重新传输
- 如果pkt(或ACK)只是延迟了(没有丢失):
  - 重新传输将是重复的，但使用seq.#’已经处理了这个问题
  - 接收方必须dingyi 被确认的数据包的序列号

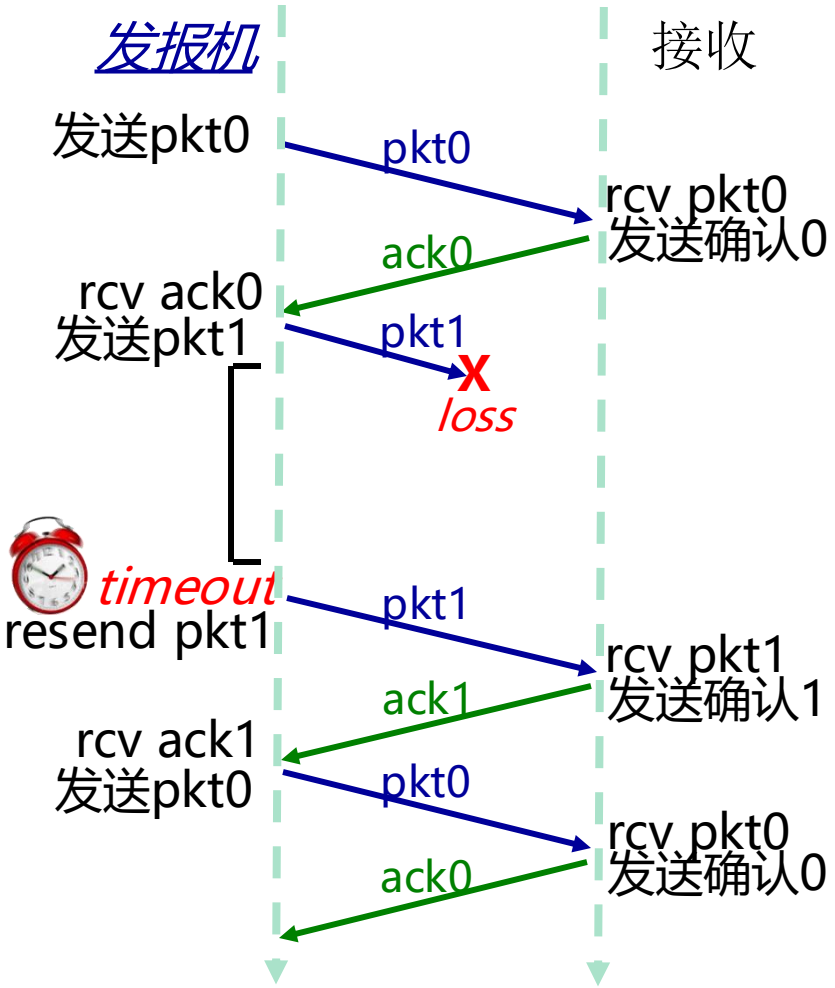
rdt3.0发送器


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
Wait for call 0from above
Wait for ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
stop_timer
Wait for ACK1
Wait for call 1 from above
timeout
udt_send(sndpkt)
start_timer
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示例

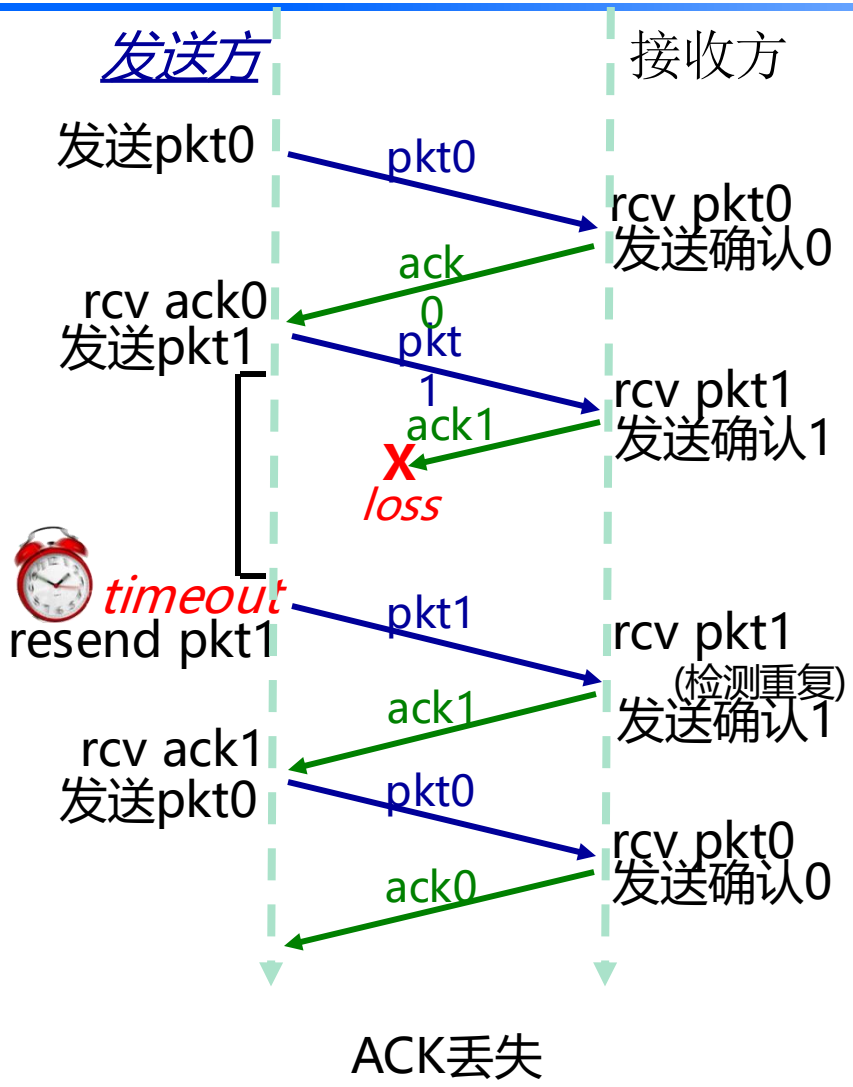


(a)没有丢包

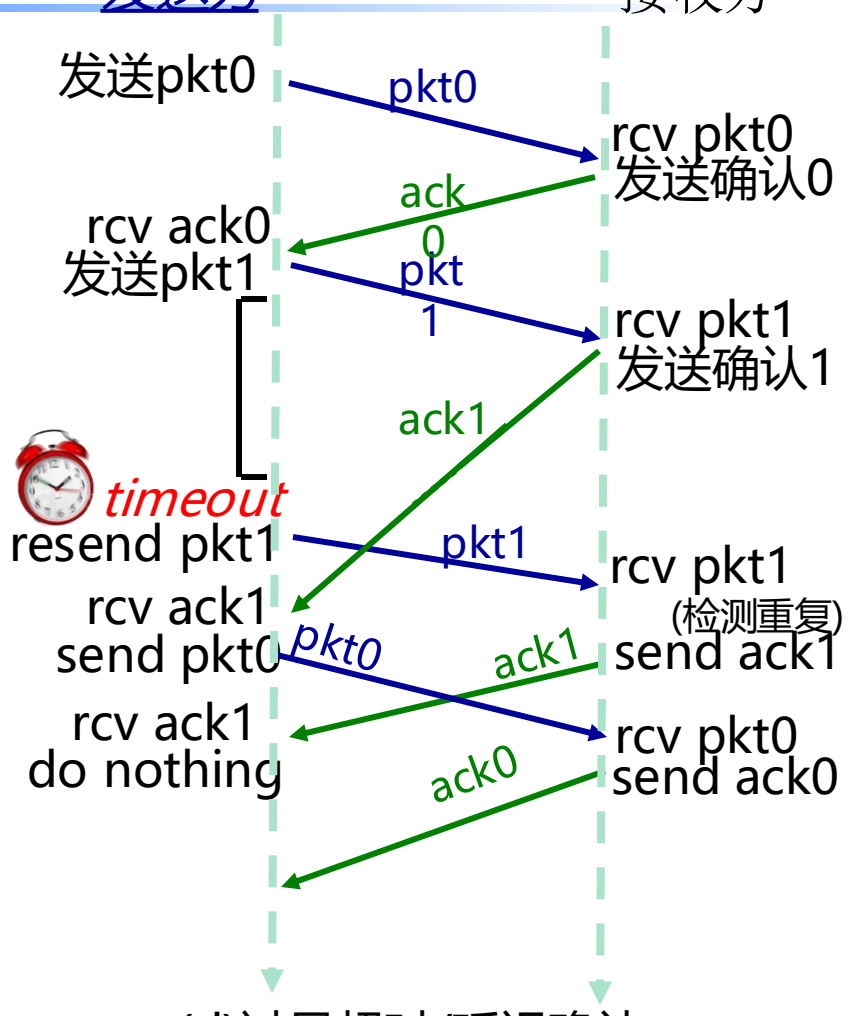


(b)丢包

rdt3.0示例



ACK丢失



(d)过早超时/延迟确认

rdt3.0的性能

- rdt3.0可以工作，但是性能很差
- 例如:1 Gbps链路、30毫秒RTT延迟、8000比特分组:

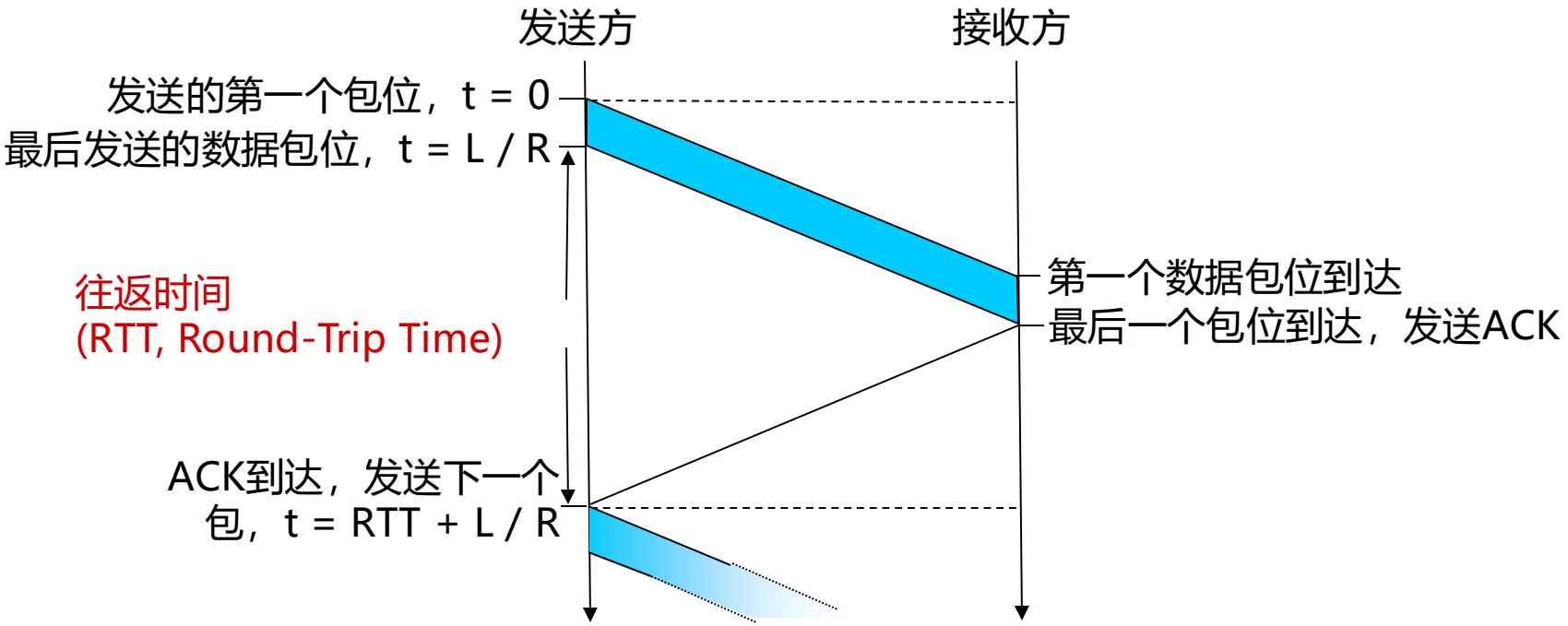
$$Dtrans = \frac{L}{R} = \frac{8000\ bits/pkt}{10^9\ bits/sec} = 8\ microsecs$$

- u发送方:利用率–发送方忙于发送的时间比例

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{0.008}{30.008} = 0.00027$$

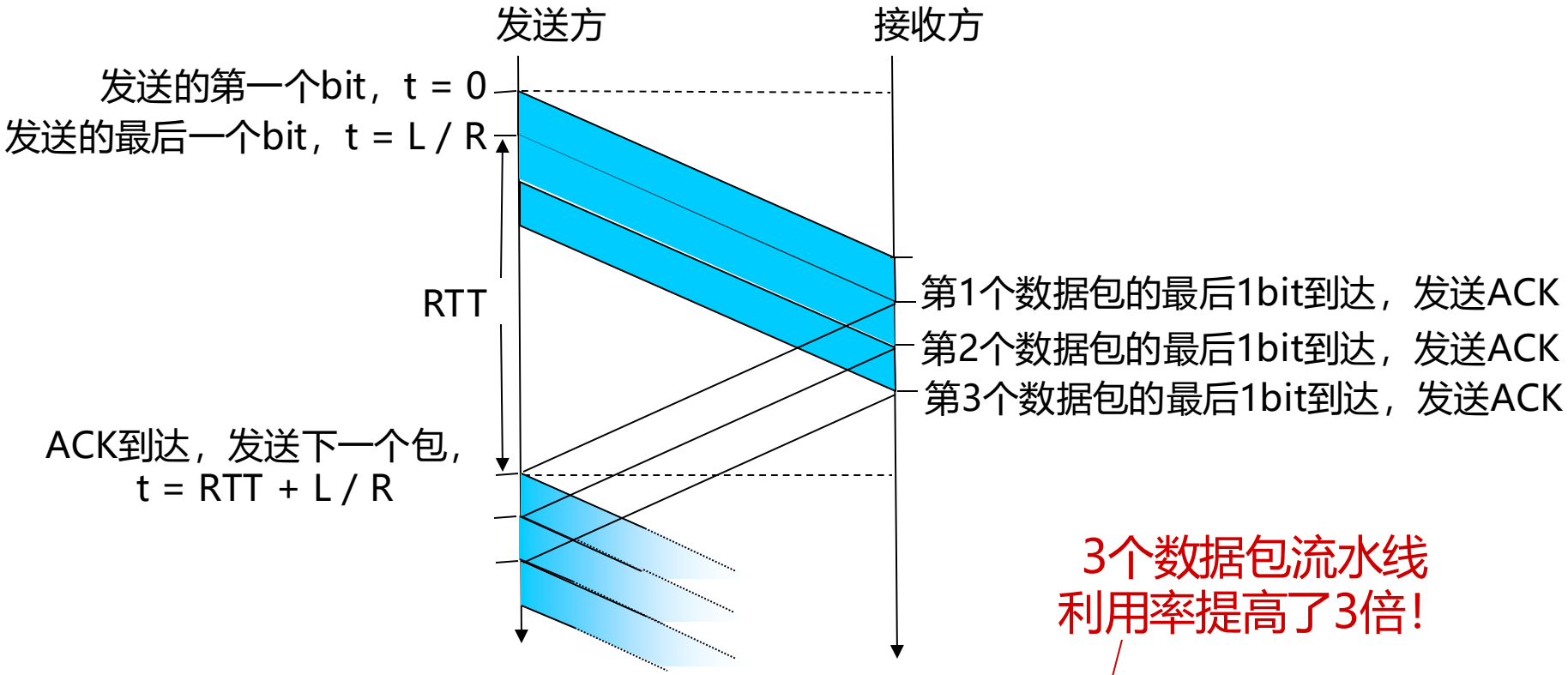
- 0.008毫秒传输。(8Kb数据包)，每30.008毫秒
- 1 Gbps链路上的267 kb/秒吞吐量
- 网络协议限制物理资源的使用!

rdt3.0:停止-等待操作



$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{0.008}{30.008} = 0.00027$$

流水线:提高利用率



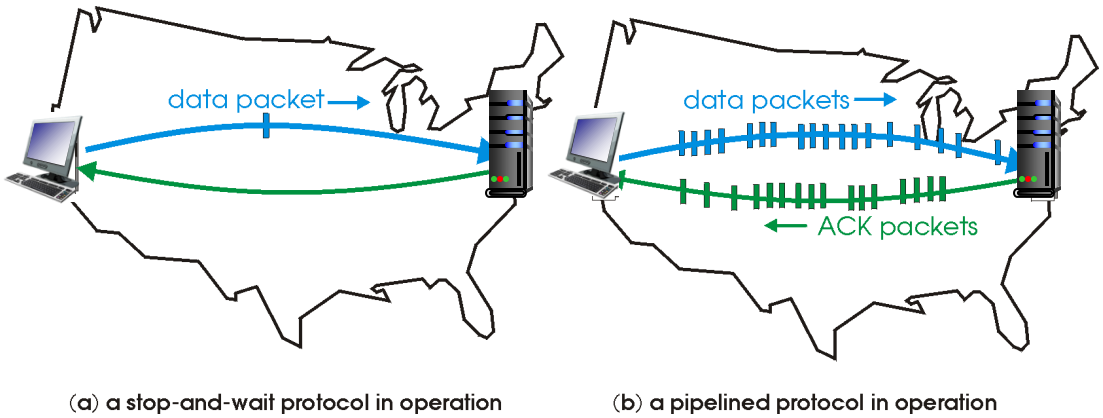
3个数据包流水线
利用率提高了3倍!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{0.0024}{30.008} = 0.00081$$

流水线可靠数据传输协议

流水线:发送者允许多个“进行中”的、尚未被确认的pkt（等待确认的分组）

- 必须增加序列号的范围
- 发送方和/或接收方的缓冲
- 方法:处理丢失、损坏、过度延迟



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

- 两种常用的流水线协议: 回退N步(GBN，go-back-N);
- 选择重传(SR，selective repeat)

流水线协议:概述

回退N步（Go Back N, GBN）：

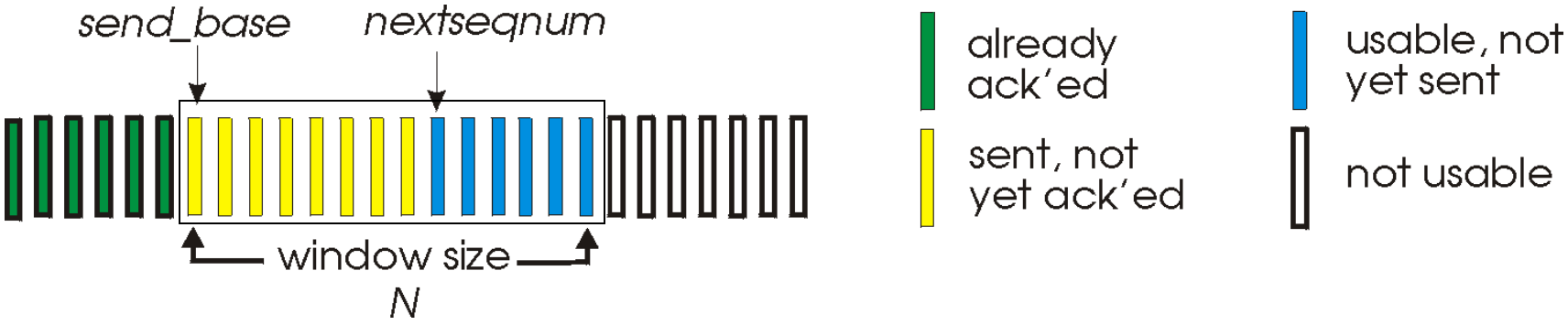
- 发送方管道中最多可以有N个未确认的数据包
- 接收器发送累积式ACK。
  - 如果乱序到达，不确认数据包
- 发送方对已发送、未确认的包启动计时器
  - 当超时，重新传输所有已发送未确认的数据包

选择重传:

- 发送方管道中最多可以有N个未确认的数据包
- 接收方为每个收到的分组分别发送ACK反馈。
- 发送方为每个未确认的数据包维护计时器
  - 当超时，仅重新传输该未确认的分组。

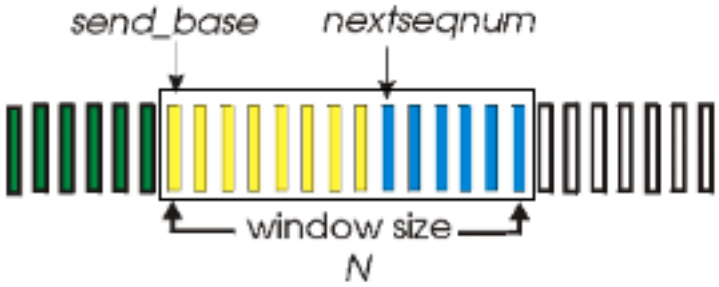
回退N步:发送方

- 在分组的首部设置中的k位序列号
- 允许最多N个连续未确认数据包的“滑动窗口”，允许连续的多个分组不被应答。



- ACK(n): ACK所有n号之前，包括n号在内的pkt-- “累积式ACK”
- 为最早的未应答（in-flight）的pkt设置计时器（timer）
- 当第n个pkt发生超时：重传pkt n和 pkt n以后的所有pkt

GBN:发送方FSM



rdt_send(data)

if (nextseqnum < base+N) {
  sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
  udt_send(sndpkt[nextseqnum])
  if (nextseqnum == base)
    start_timer
  nextseqnum++
  }
else
  refuse_data(data)

Λ

base=1
nextseqnum=1

Wait

timeout

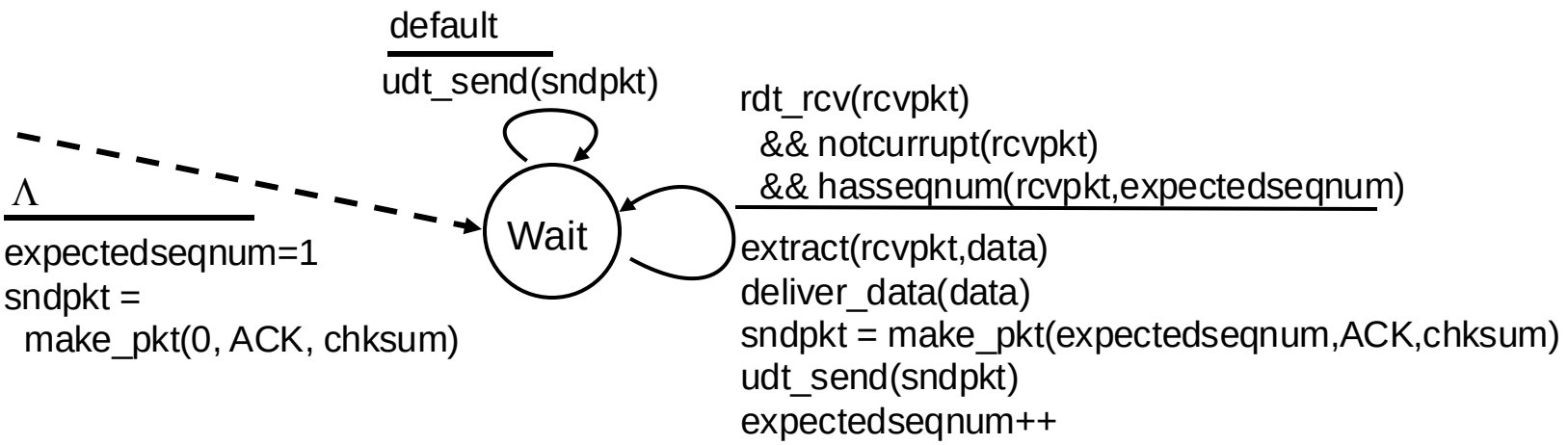
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])

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

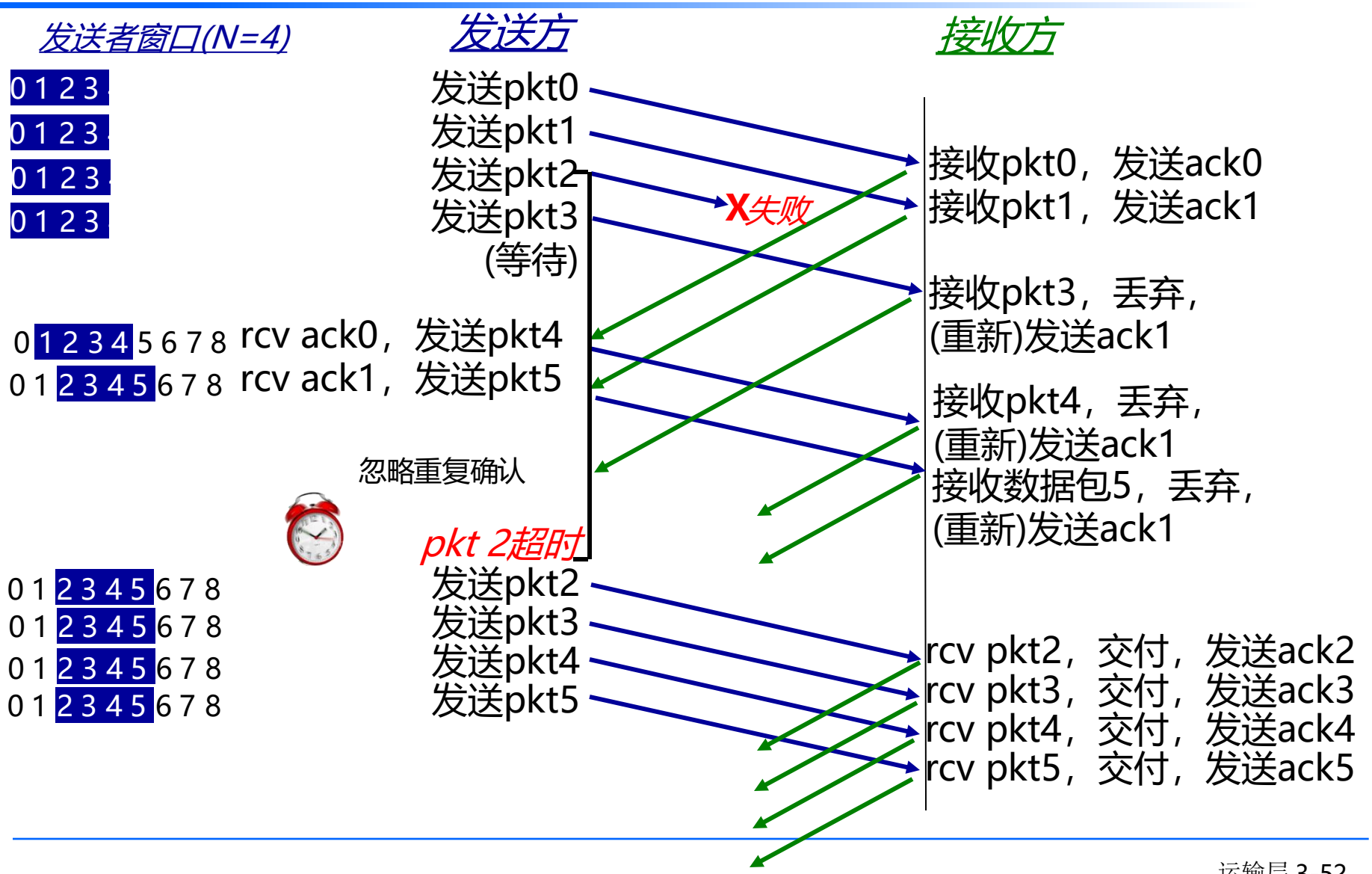


仅确认:总是为正确接收的具有最高有序序列编号的数据包发送确认

- 可能会生成重复的ack
- 只需要记住预期的序号

- 乱序分组:
  - 丢弃(不缓存):没有接收器缓存!
  - 接收到的分组中按序对最高seq#进行ACK

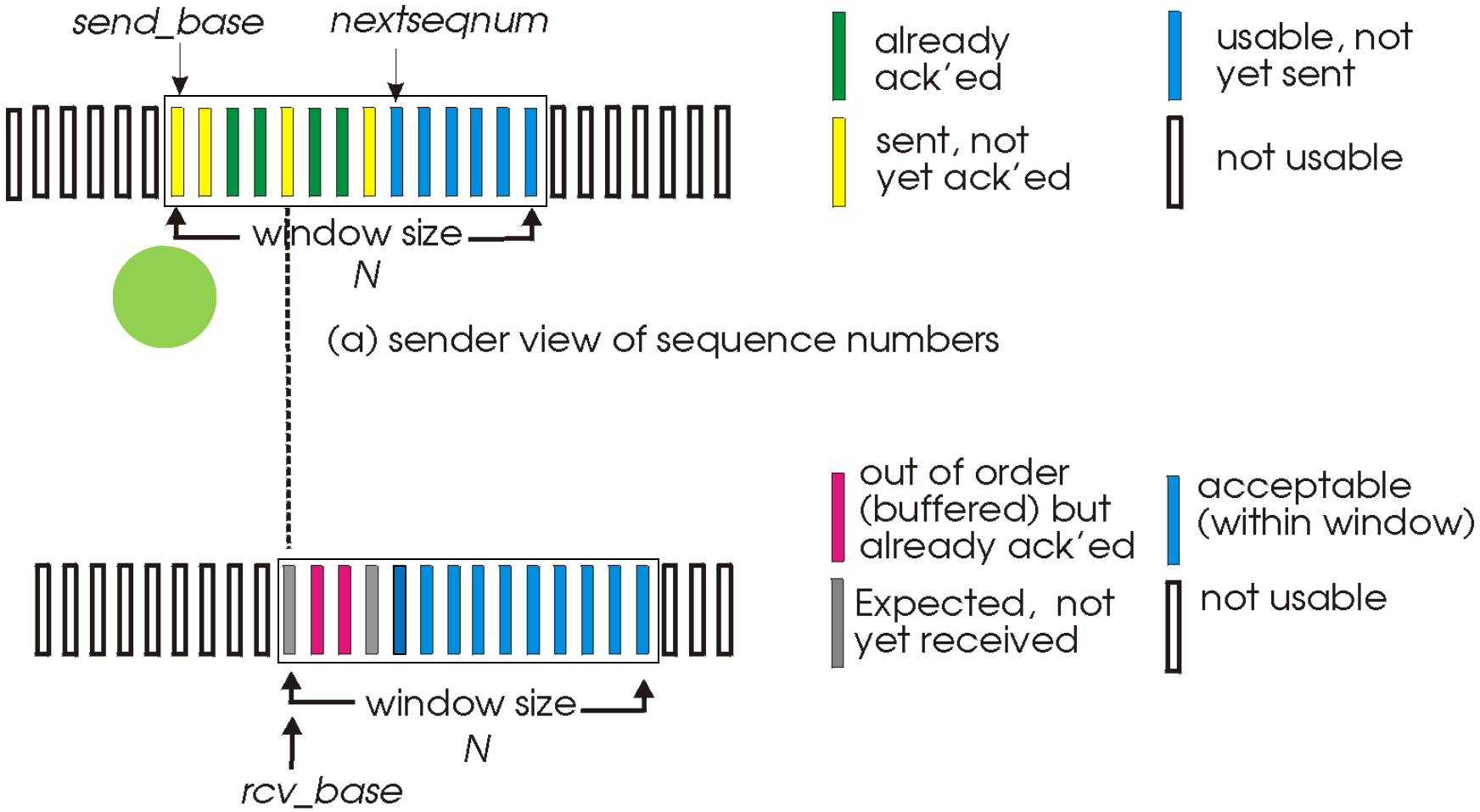
GBN运行


发送者窗口(N=4)
发送方
接收方
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
发送pkt0
发送pkt1
发送pkt2
发送pkt3
(等待)
X失败
接收pkt0，发送ack0
接收pkt1，发送ack1
接收pkt3，丢弃，
(重新)发送ack1
0 1 2 3 4 5 6 7 8 rcv ack0，发送pkt4
0 1 2 3 4 5 6 7 8 rcv ack1，发送pkt5
接收pkt4，丢弃，
(重新)发送ack1
忽略重复确认
接收数据包5，丢弃，
(重新)发送ack1
pkt 2超时
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
发送pkt2
发送pkt3
发送pkt4
发送pkt5
rcv pkt2，交付，发送ack2
rcv pkt3，交付，发送ack3
rcv pkt4，交付，发送ack4
rcv pkt5，交付，发送ack5

选择性重传

- 接收器单独确认所有正确接收的数据包
  - 根据需要缓存数据包，以便最终有序传送到上层
- 发送方只重发未收到应答的分组进行重发
  - 发送方对每个没有确认的的分组计时。
- 发送方窗口
  - *n个连续的序列号*
  - 编号有限

选择性重传:发送方、接收方窗口



选择性重复

发送方

来自上方的数据:

- 如果窗口中有下一个可用的序列号，发送数据包

超时(n):

- 重新发送数据包，重新启动计时器

ACK(n) in [sendbase, sendbase+N]:

- 将分组n标记为已接收
- 如果窗口中最小的pkt被确认，将窗口基数推进到下一个未确认的序列号

接收方

pkt n in [rcvbase，rcvbase+N-1]

- 发送确认(n)
- 无序:缓冲区
- 有序:交付(也交付缓存的有序数据包)，将窗口推进到下一个尚未接收的数据包

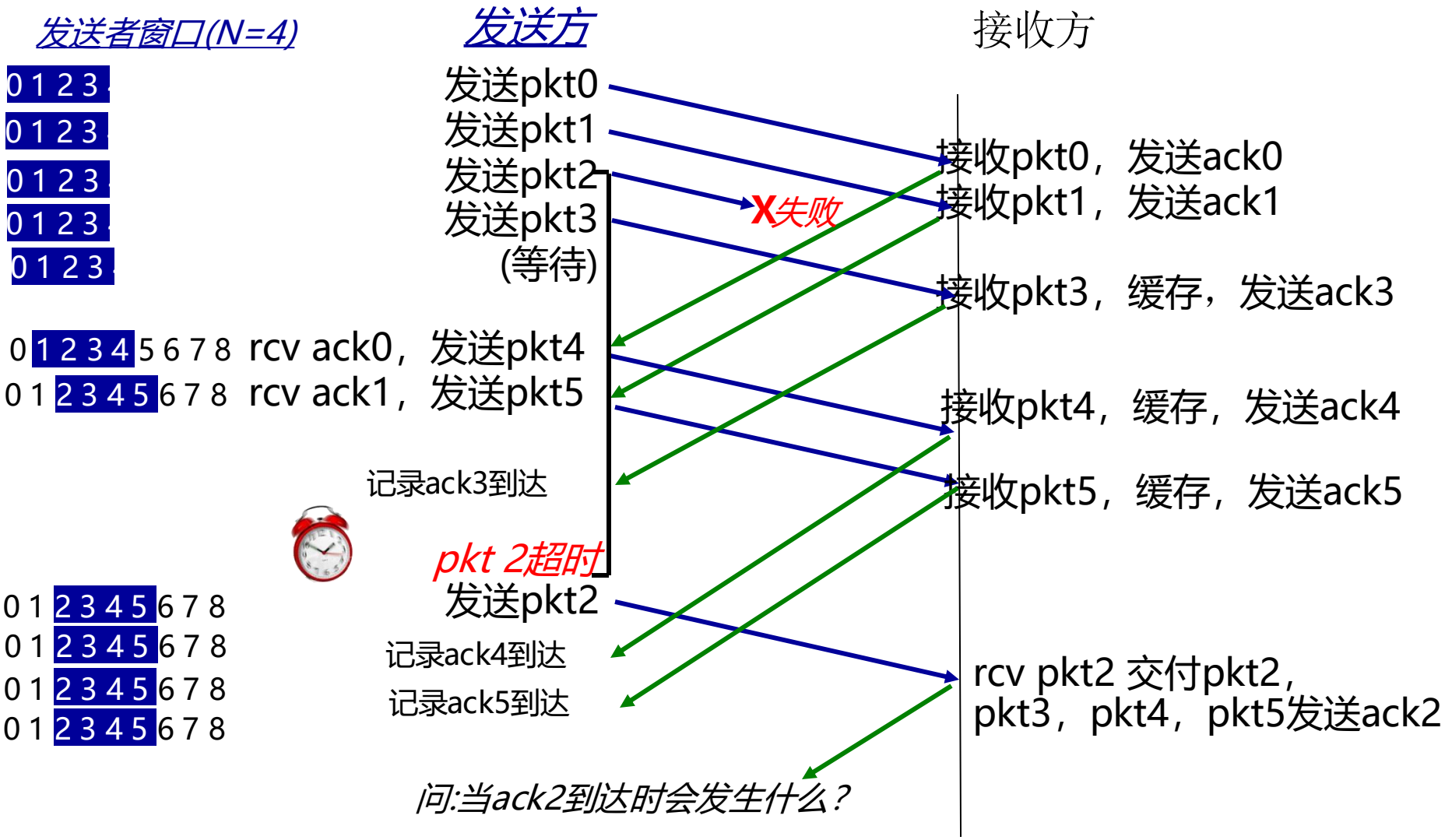
pkt n in [rcvbase-N，rcvbase-1]

- 确认(n)

否则:

- 忽略分组

选择重传：示例

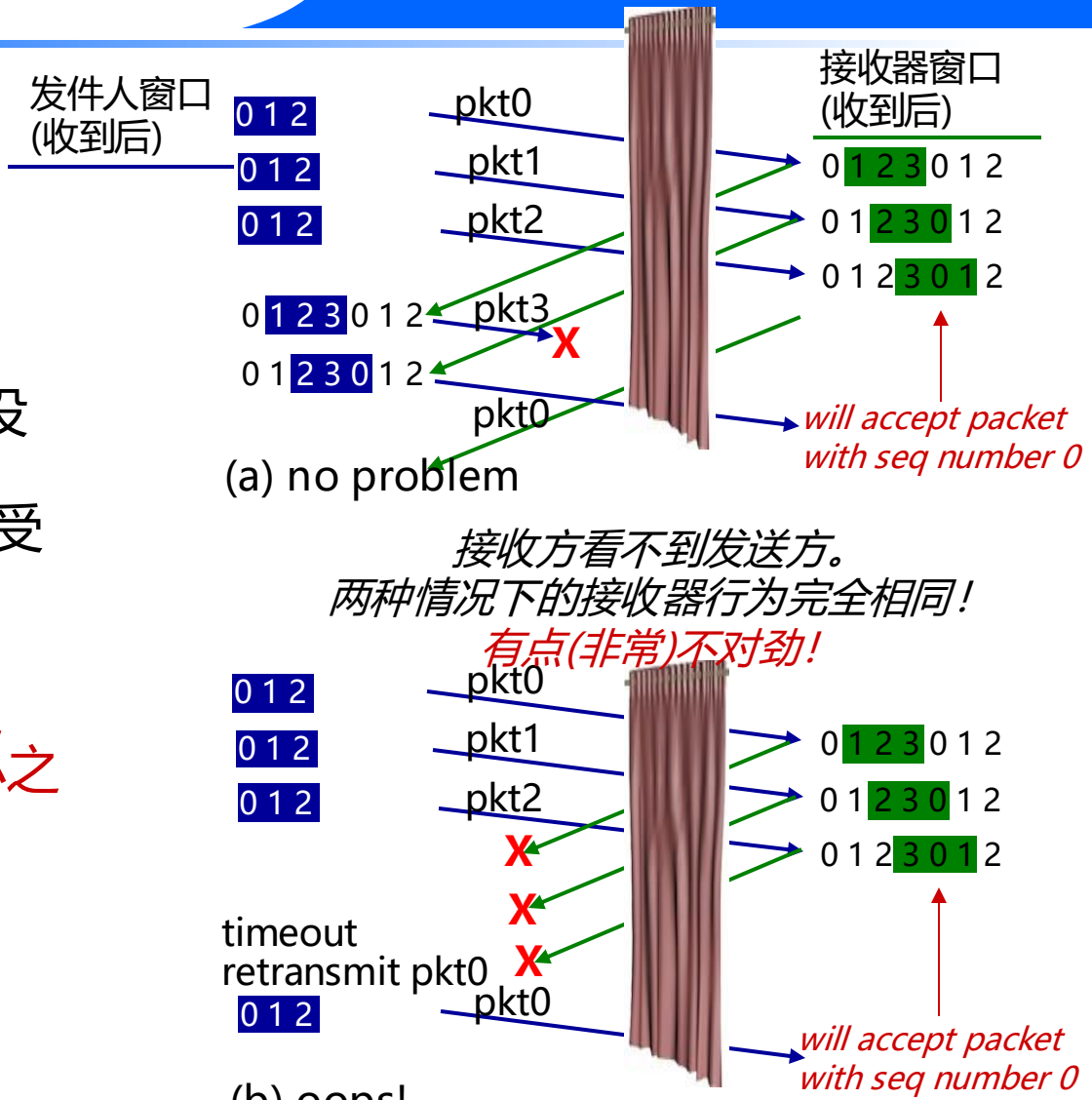

发送者窗口(N=4)
发送方
接收方
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
发送pkt0
发送pkt1
发送pkt2
发送pkt3
(等待)
X失败
接收pkt0，发送ack0
接收pkt1，发送ack1
接收pkt3，缓存，发送ack3
0 1 2 3 4 5 6 7 8 rcv ack0，发送pkt4
0 1 2 3 4 5 6 7 8 rcv ack1，发送pkt5
接收pkt4，缓存，发送ack4
接收pkt5，缓存，发送ack5
记录ack3到达
pkt 2超时
发送pkt2
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
记录ack4到达
记录ack5到达
rcv pkt2 交付pkt2，
pkt3，pkt4，pkt5发送ack2
问:当ack2到达时会发生什么？

选择重传:困境

示例:

- 序号:0，1，2，3
- 窗口大小=3
- 接收者认为两种情况没有区别!
- 重复数据在(b)中被接受为新数据

问:为了避免(b)中的问题，seq #大小和窗口大小之间的关系是什么?



第3章:大纲

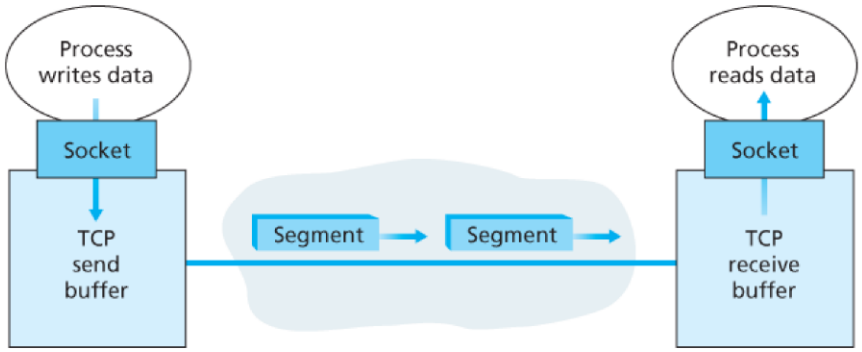
1. 3.1运输层服务
2. 3.2多路复用和多路分解
3. 3.3无连接传输:UDP
4. 3.4可靠运输原理
5. **3.5面向连接的传输:TCP**

1.分段结构　2.可靠的数据传输　3.流量控制　4.连接管理

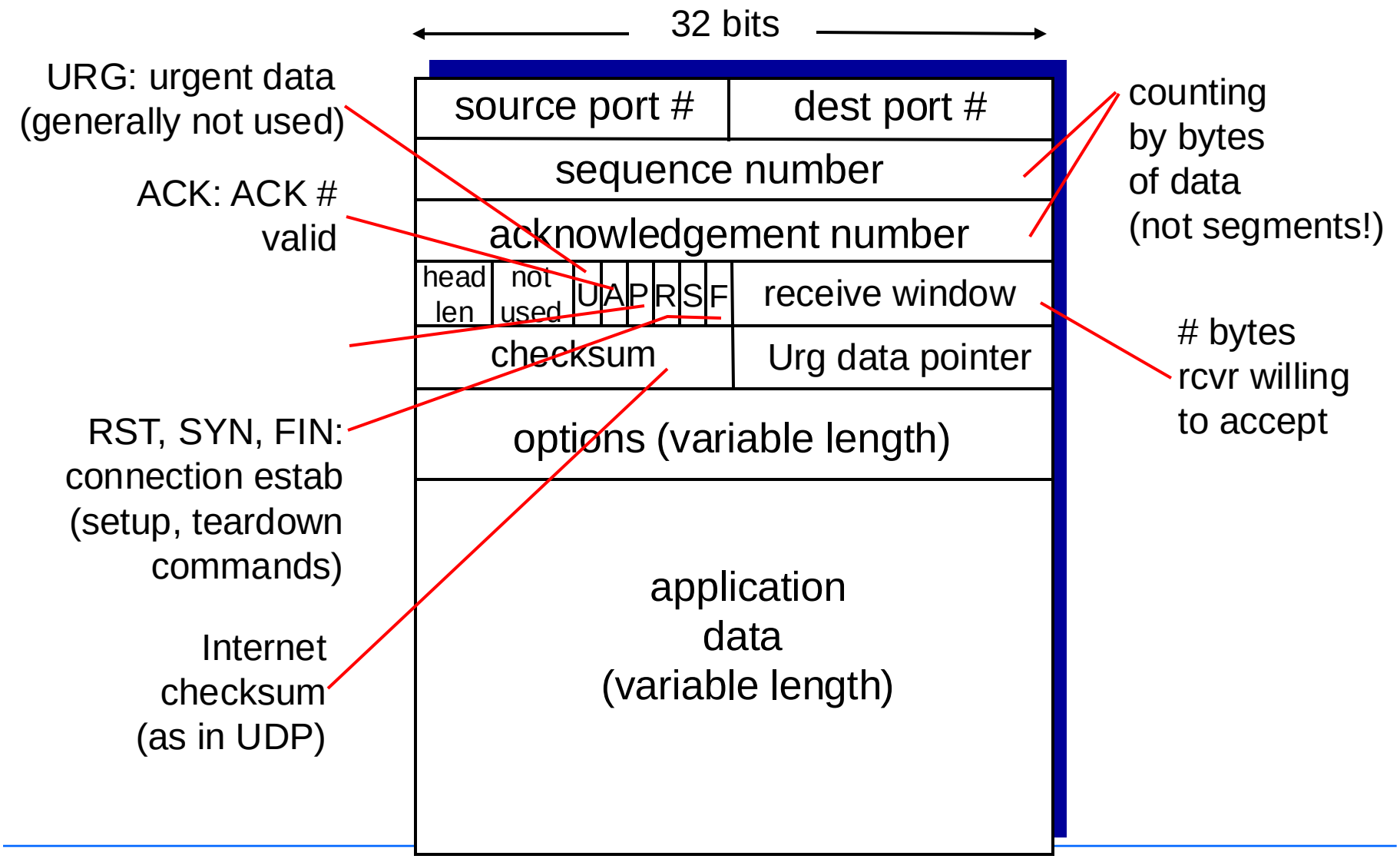
6. 3.6拥塞控制原理
7. 3.7 TCP的拥塞控制

TCP:概述RFCs: 793，1122，1323，2018，2581

- 面向连接:
  - 握手
  - 缓冲区、变量、套接字
  - 点对点(一个发送者，一个接收者)
- 可靠的数据传输:
  - 有序字节流
  - 流水线(与拥塞和流量控制相关的窗口大小)
- 流量和拥塞控制:
  - 发送方的发送速度不得超过接收方的处理速度
- 全双工数据:
  - 同一时间同一连接中的双向数据流传输
  - MSS: maximum segment size（最大数据段字节数1460，对应MTU1500，以太网和PPP链路)



TCP数据段格式


32 bits
source port #
dest port #
sequence number
acknowledgement number
head len
not used
U
A
P
R
S
F
receive window
checksum
Urg data pointer
options (variable length)
application data (variable length)
URG: urgent data (generally not used)
ACK: ACK # valid
RST, SYN, FIN: connection estab (setup, teardown commands)
Internet checksum (as in UDP)
counting by bytes of data (not segments!)
bytes rcvr willing to accept

TCP序列 数字，确认

序列号:

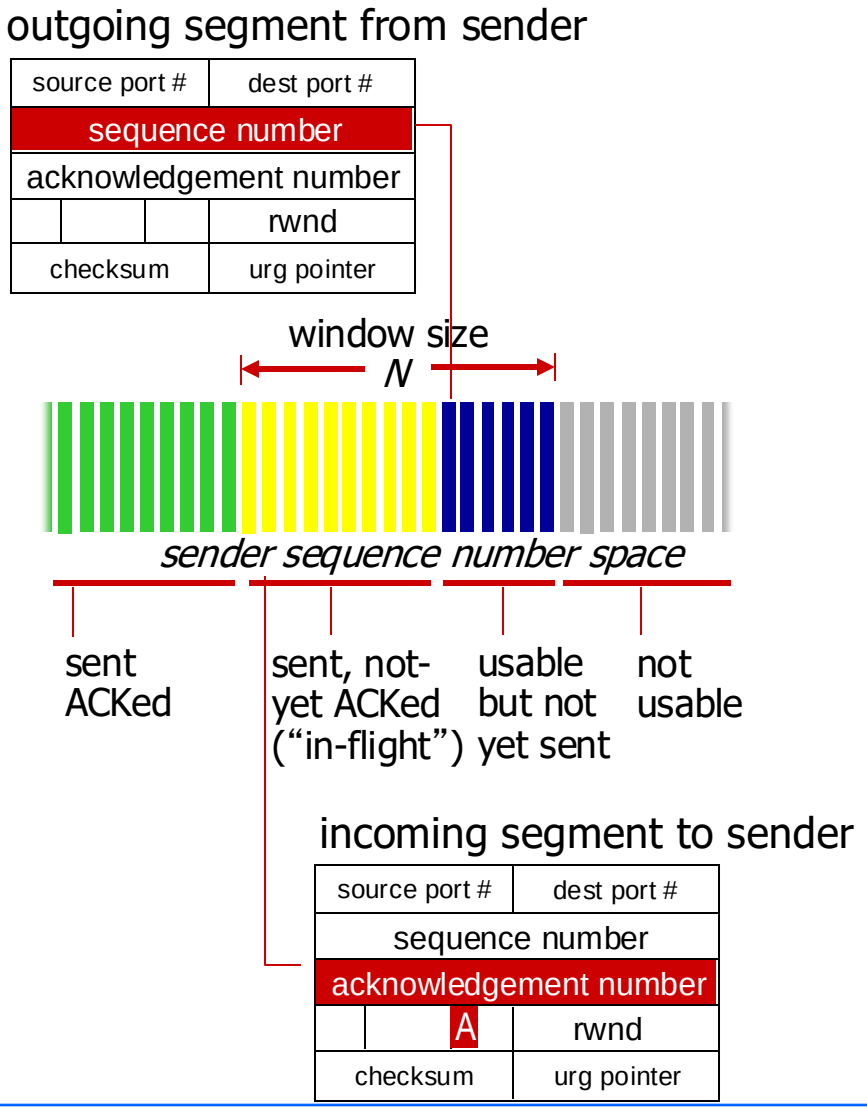
- 段数据中第一个字节的字节流“编号”
- 随机启动

确认号:

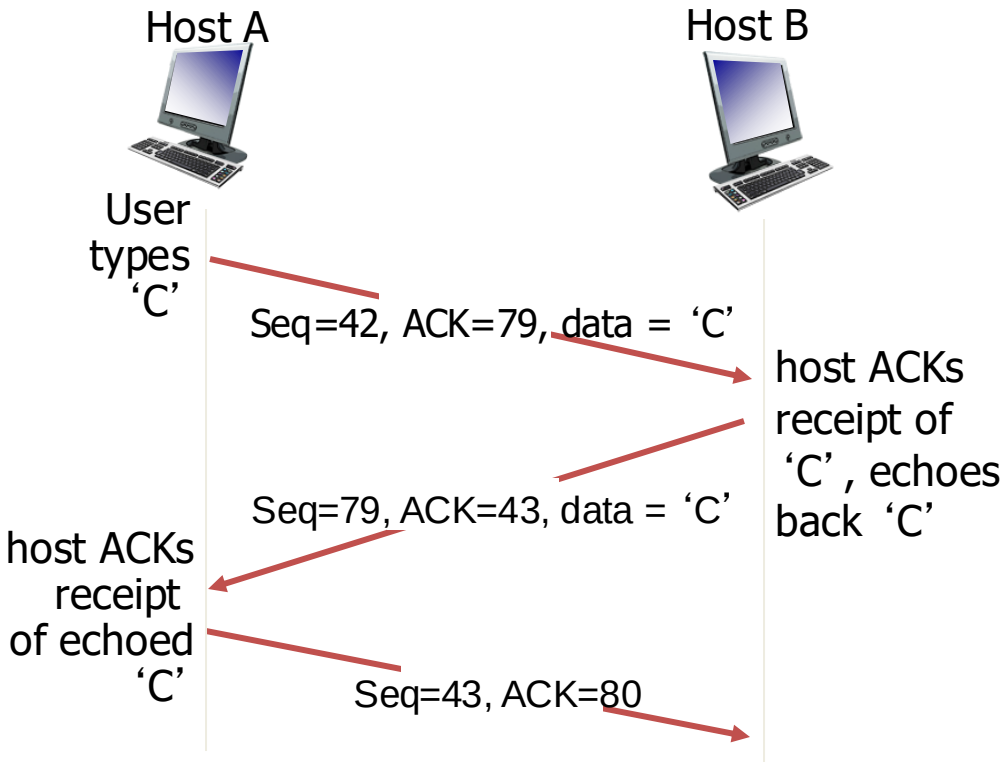
- seq为希望从对方发来的“下一个”字节的编号
- 累积ACK

问:接收器如何处理无序段

- 答:TCP规范没有说-取决于程序设计者



TCP序列 数字，确认



简单的telnet场景

TCP往返时间，超时

问:如何设置TCP超时Timer?

- 比RTT还长
  - 但是RTT各不相同
- *太短:过早超时，不必要的重新传输*
- *过长:对片段丢失反应迟钝*

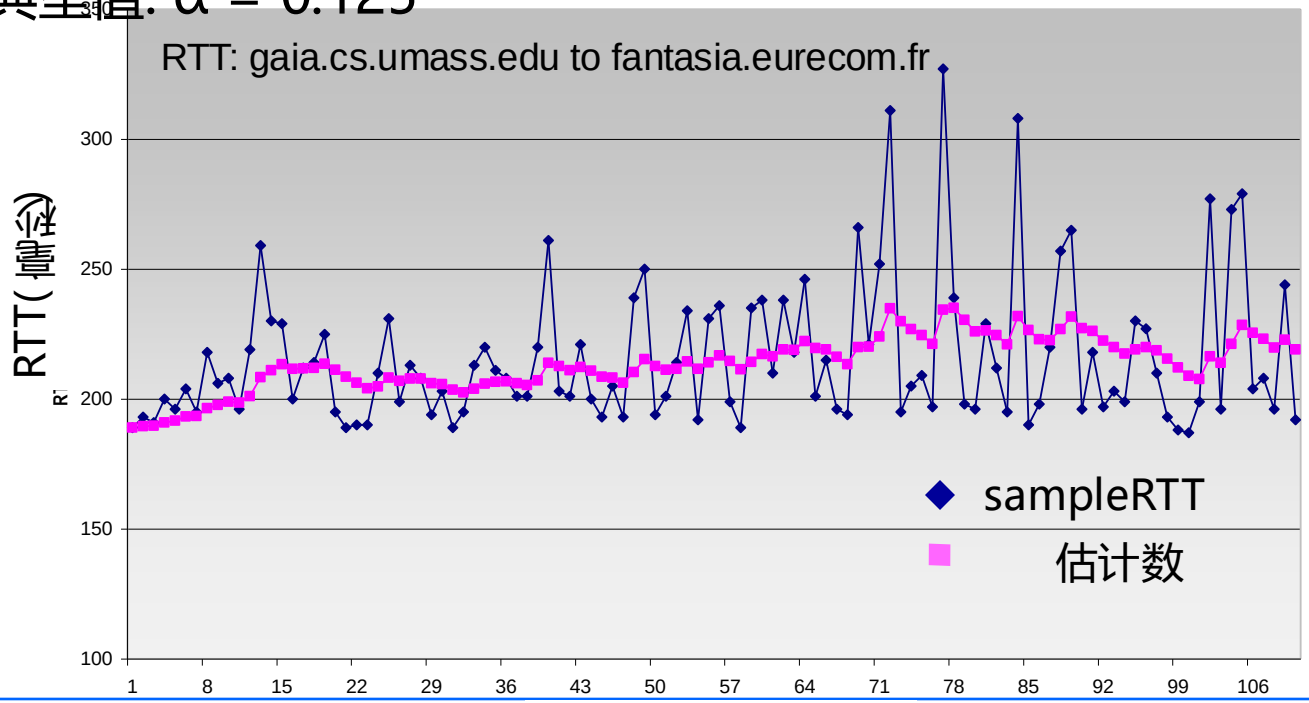
问:如何计算RTT?

- SampleRTT:从段传输到ACK接收的实测时间
  - 忽略重新传输
- SampleRTT会有所不同，希望估计RTT“更平滑”
  - 平均最近的几次测量，而不仅仅是当前的SampleRTT

TCP往返时间，超时

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

- 指数加权移动平均
- 过去样本影响以指数方式快速下降
- 典型值: α = 0.125



TCP往返时间，超时

- 估计SampleRTT与估计值的偏差:

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

  (通常， β = 0.25)

- 超时间隔:EstimatedRTT加上“安全余量”
  - 估计值的较大变化率->加大安全余量

TimeoutInterval = EstimatedRTT + 4*DevRTT

估计RTT

“安全余量”

第3章:大纲

1. 3.1运输层服务
2. 3.2多路复用和多路分解
3. 3.3无连接传输:UDP
4. 3.4可靠运输原理
5. **3.5面向连接的传输:TCP**

1.分段结构　2.可靠的数据传输　3.流量控制　4.连接管理

6. 3.6拥塞控制原理
7. 3.7 TCP的拥塞控制

可靠的数据传输

- TCP在IP的不可靠服务之上创建可靠的数据传输
  - 流水线段
  - 累积确认
  - 单次重发定时器
- 触发重新传输的原因:
  - 超时事件
  - 冗余确认事件

让我们首先考虑简化的TCP发送方:

- 忽略重复的ack
- 忽略流量控制
- 忽略拥塞控制

TCP发送方事件:

来自应用程序的数据:

- 使用序号创建段
  - seq #是段中第一个数据字节的字节流编号
- 如果计时器尚未运行，则启动计时器
  - 最早的未确认段
  - 超时间隔:TimeOutInterval

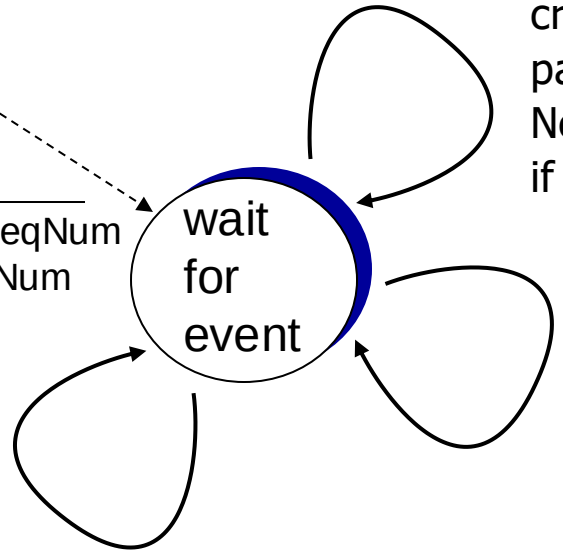
超时:

- 导致超时的重新传输字段
- 重启定时器

ack rcvd:

- 如果ack确认了先前未确认的数据段
  - 更新已知被确认的内容
  - 如果仍有未确认的数据段，则启动计时器

TCP发送方



Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

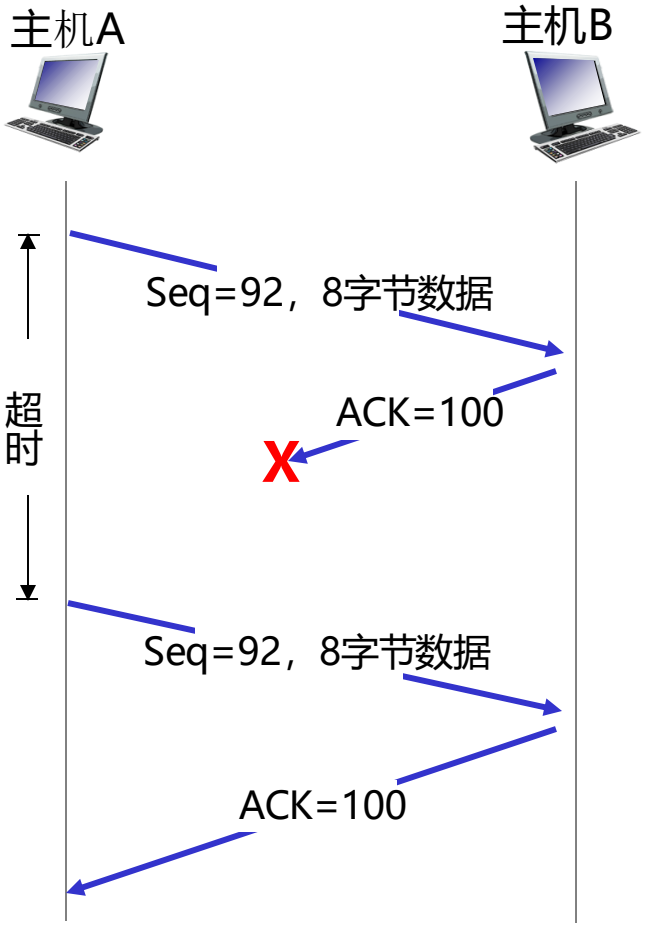
timeout

retransmit not-yet-acked segment
        with smallest seq. #
start timer

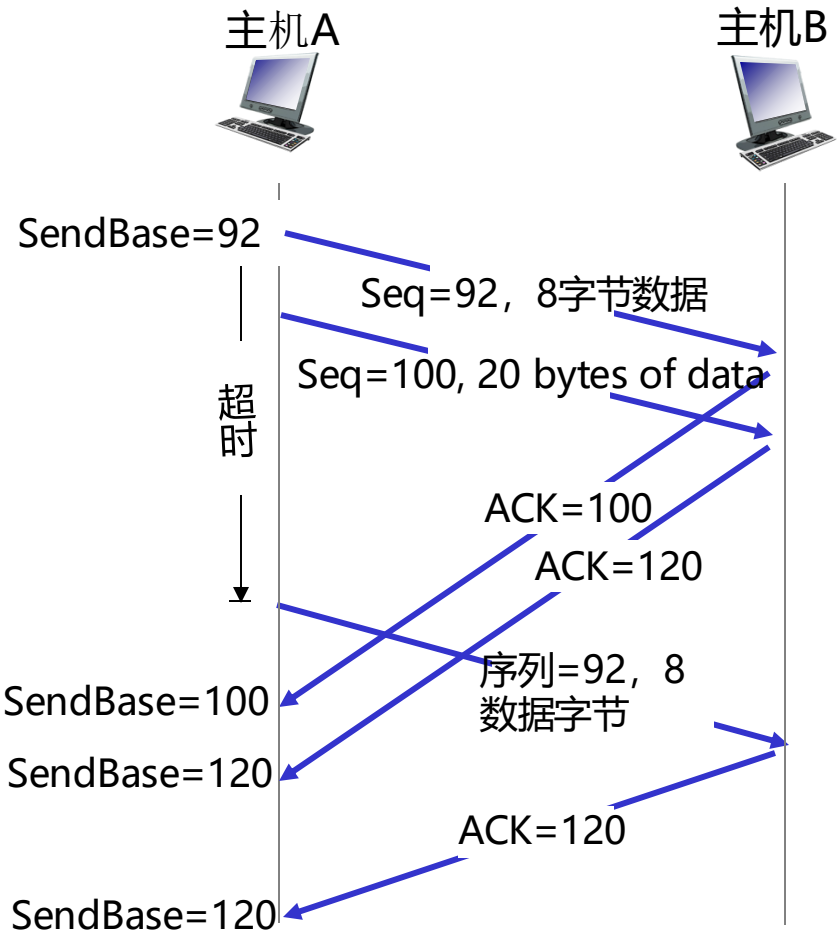
ACK received, with ACK field value y

```
if (y > SendBase) {
    SendBase = y
    /* SendBase–1: last cumulatively ACKed byte */
    if (there are currently not-yet-acked segments)
        start timer
      else stop timer
    }
```

TCP:重传场景

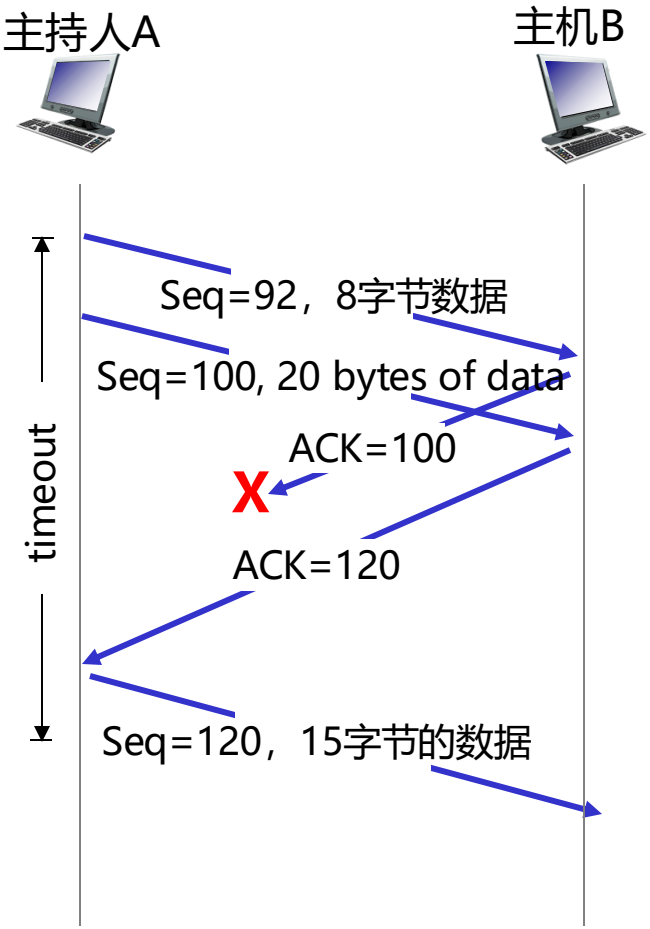


丢失ACK



过早超时

TCP:重传场景



累积确认+超时间隔加倍

TCP ACK生成[RFC 1122，RFC 2581]

TCP ACK生成策略

接收端事件	接收端的动作
有序数据段到达， 没有缺失的段， 所有其他数据段已经 ACKed	延迟ACK，等待500ms 看是否还有数据段到达，如果没有 发送ACK
有序数据段到达， 没有缺失的段， 有一个延迟ACK等待	立即发送一个累积ACK，以确认两个 按序报文段
失序数据段到达 Seq# 高于预测值 测到间隔	发送重复的ACK，说明seq#为下一个 期望的字节
到达的数据段部分或全部 填满了缺失的段	立即ACK，如果数据段处于缺 失的段的较低端

TCP快速重传

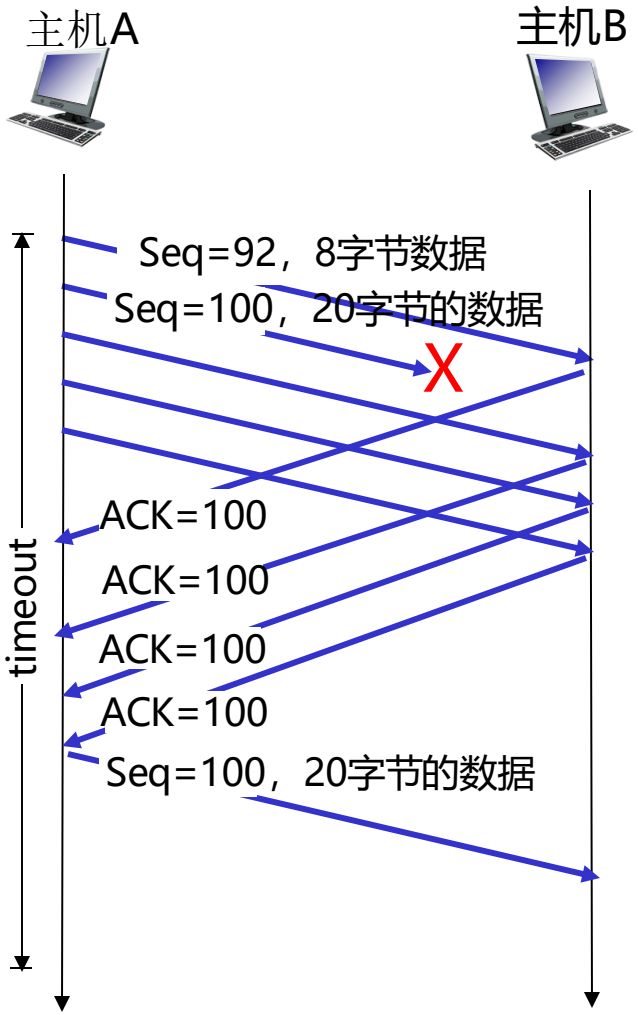
- 超时时间通常相对较长:
  - 重发丢失数据包前的长延迟
- 通过重复ack检测丢失的数据段。
  - 发送方经常连续发送许多数据段
  - 如果数据段丢失，可能会有许多重复的ack。

TCP快速重传

如果发送方收到相同数据的3个ack（“三次重复确认”），重新发送具有最小序列号的未确认数据段

- 可能是未确认数据段丢失，所以不要等待超时。

TCP快速重传



发送后快速重新传输收到三份重复确认

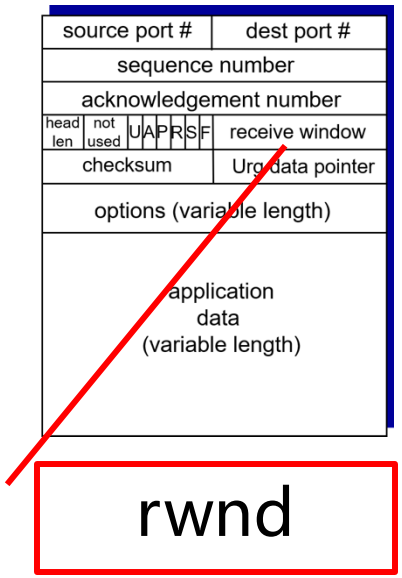
第3章:大纲

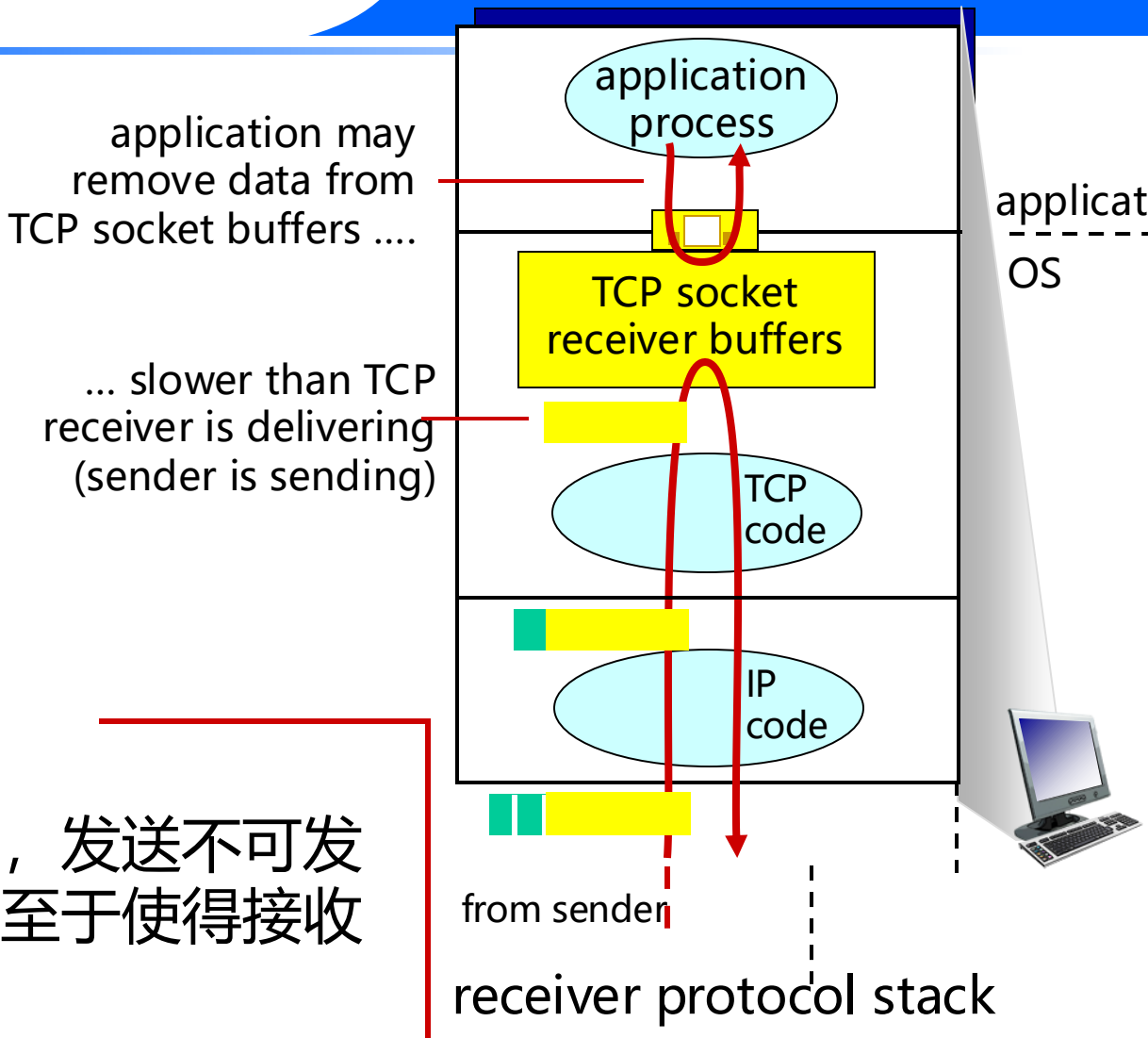
1. 3.1运输层服务
2. 3.2多路复用和多路分解
3. 3.3无连接传输:UDP
4. 3.4可靠运输原理
5. **3.5面向连接的传输:TCP**

1.分段结构　2.可靠的数据传输　3.流量控制　4.连接管理

6. 3.6拥塞控制原理
7. 3.7 TCP的拥塞控制

TCP流量控制



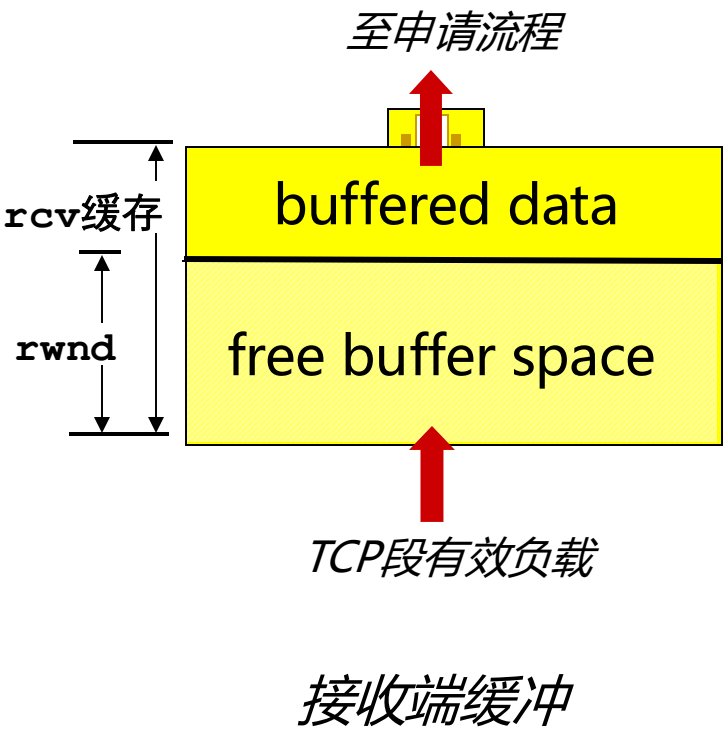


流量控制

接收方控制发送方，发送不可发送的太多、太快以至于使得接收端的缓存溢出

TCP流量控制

- 接收方通过在接收方到发送方数据段的TCP报头中包含rwnd值来“通告”空闲缓冲区空间
  - RcvBuffer大小通过socket选项设置(典型的默认值是4096字节)
  - 许多操作系统自动调整接收缓存
- 发送方将未确认(“传输中”)数据量限制为接收方的rwnd值
- 保证接收缓冲区不会溢出



第3章:大纲

1. 3.1运输层服务
2. 3.2多路复用和多路分解
3. 3.3无连接传输:UDP
4. 3.4可靠运输原理
5. **3.5面向连接的传输:TCP**

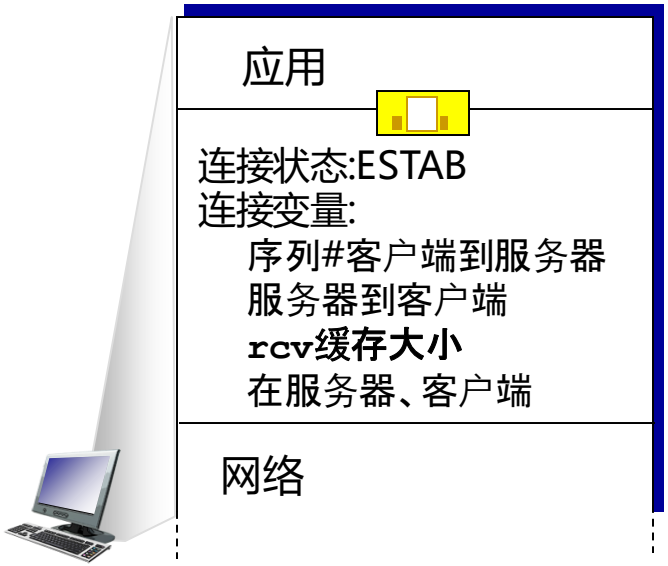
1.分段结构　2.可靠的数据传输　3.流量控制　4.连接管理

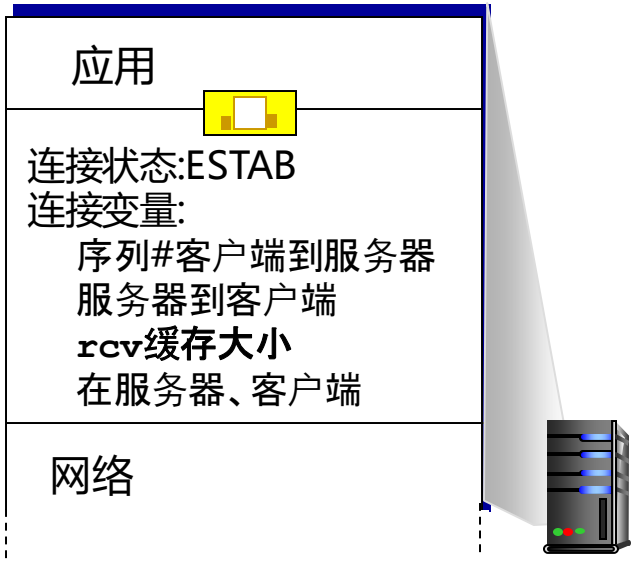
6. 3.6拥塞控制原理
7. 3.7 TCP的拥塞控制

连接管理

交换数据之前，发送方/接收方“握手”：

- 同意建立联系(双方都知道对方愿意建立联系)
- 就连接参数达成一致



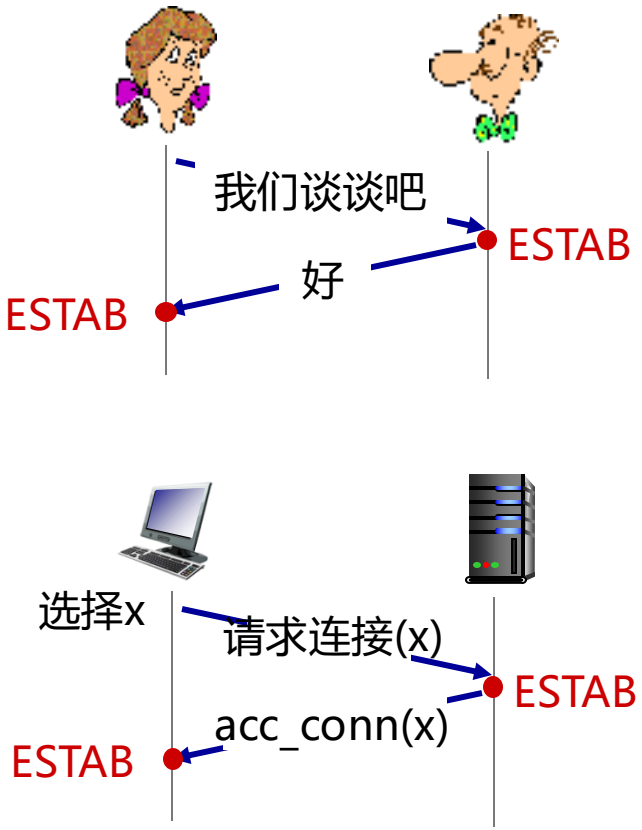



```
套接字客户端套接字=
newSocket("主机名"、"端口号");
```

```
套接字连接socket = welcome
   socket . accept() ;
```

同意建立联系

两次握手:

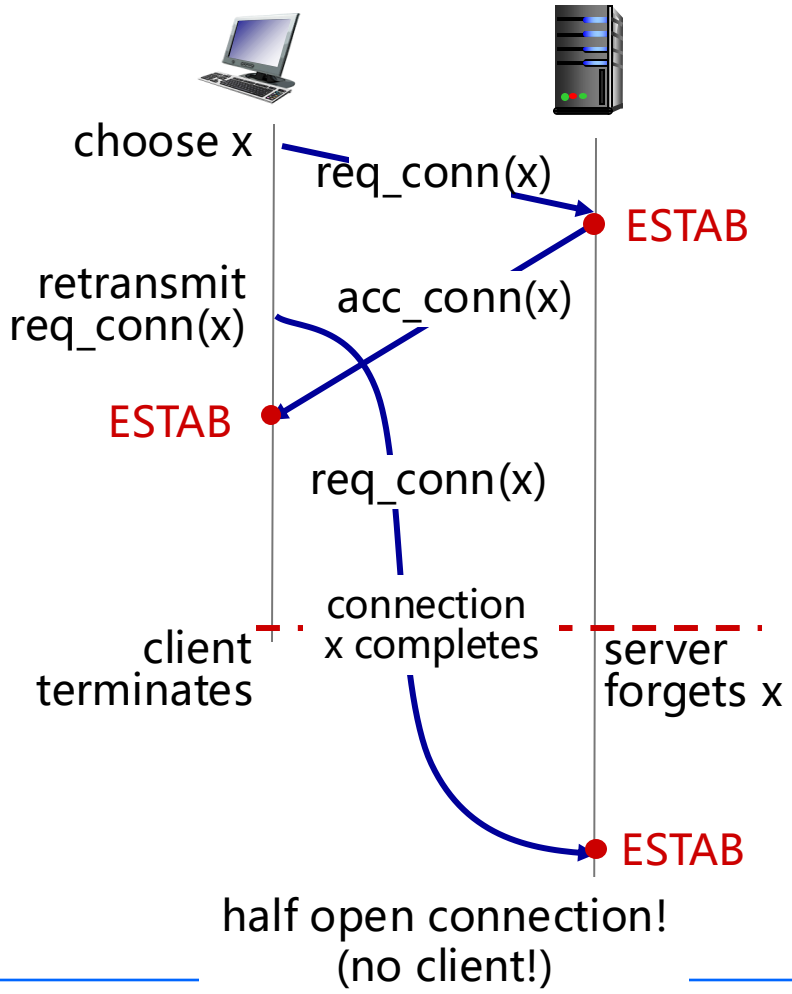


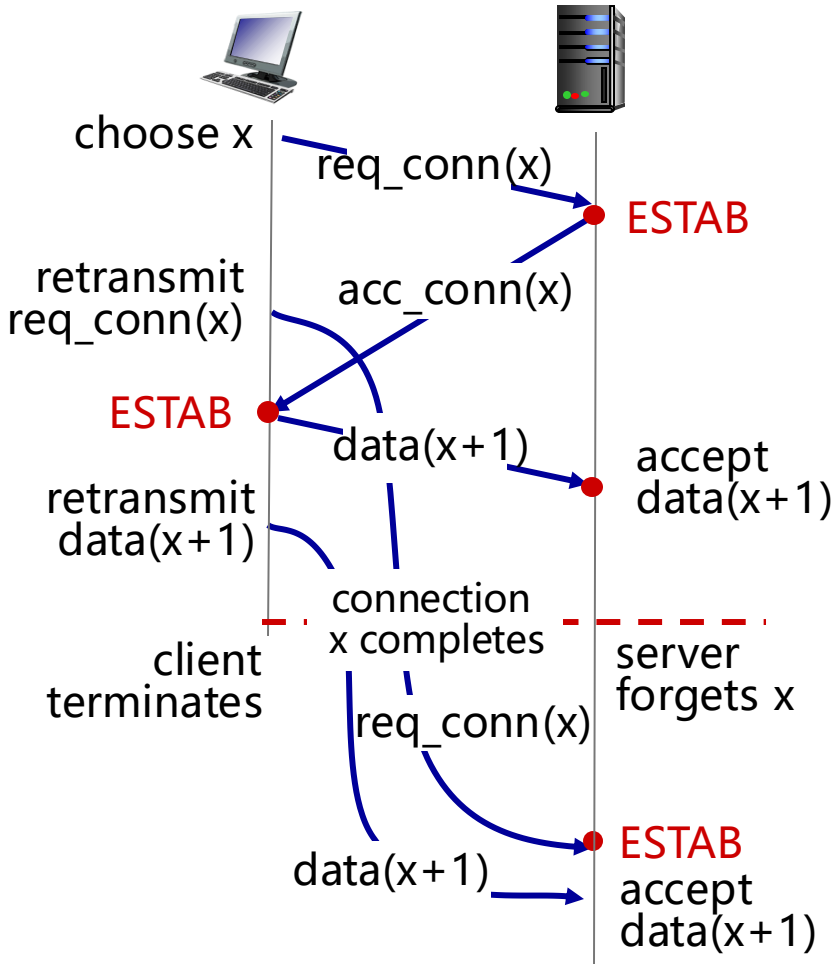
问:双向握手在网络中总是有效吗?

- 时延变化
- 重发报文由于报文丢失
- 报文重排
- 无法判断对方在不在

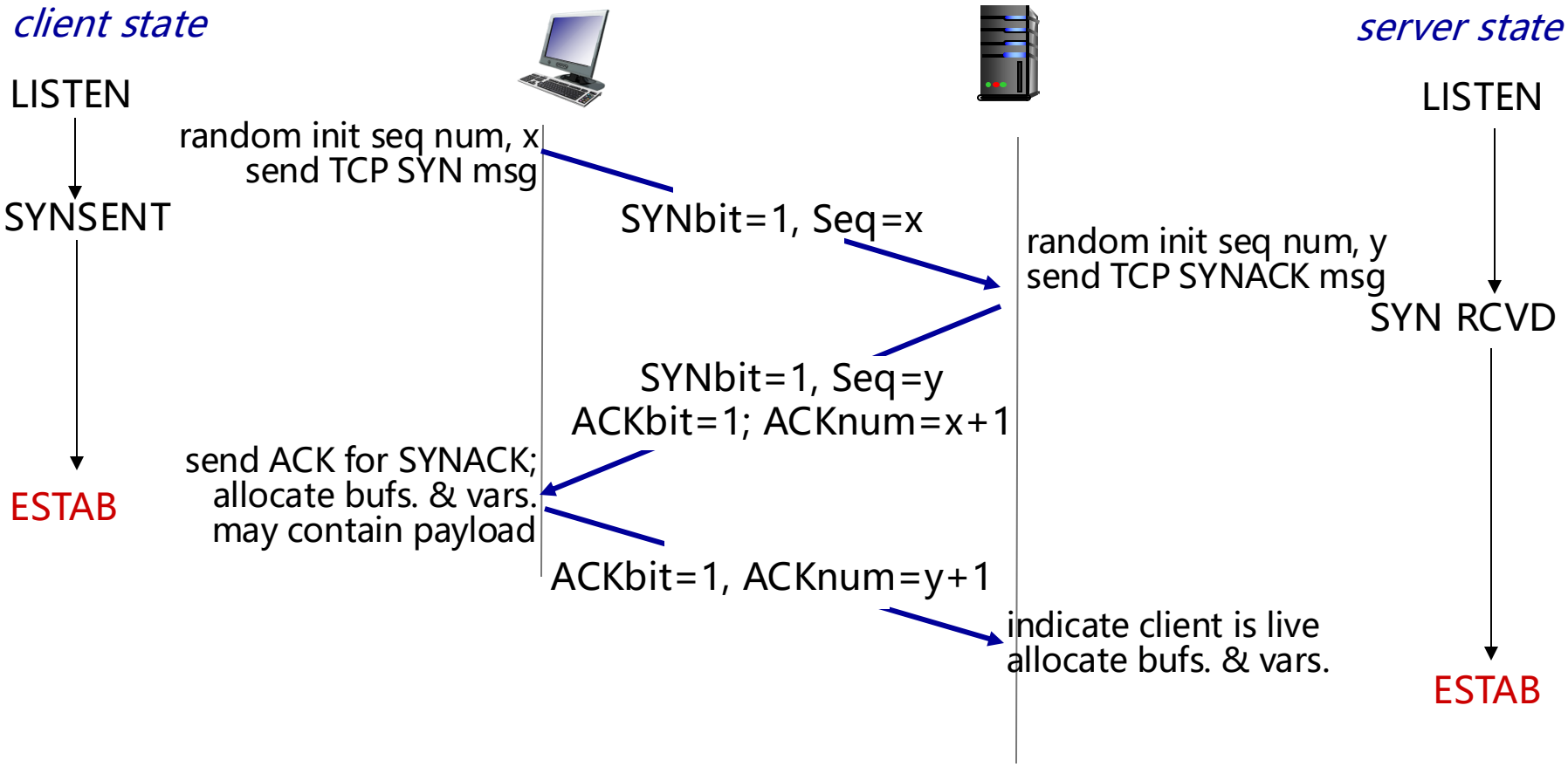
同意建立联系

双向握手失败场景:连接失效，数据重复





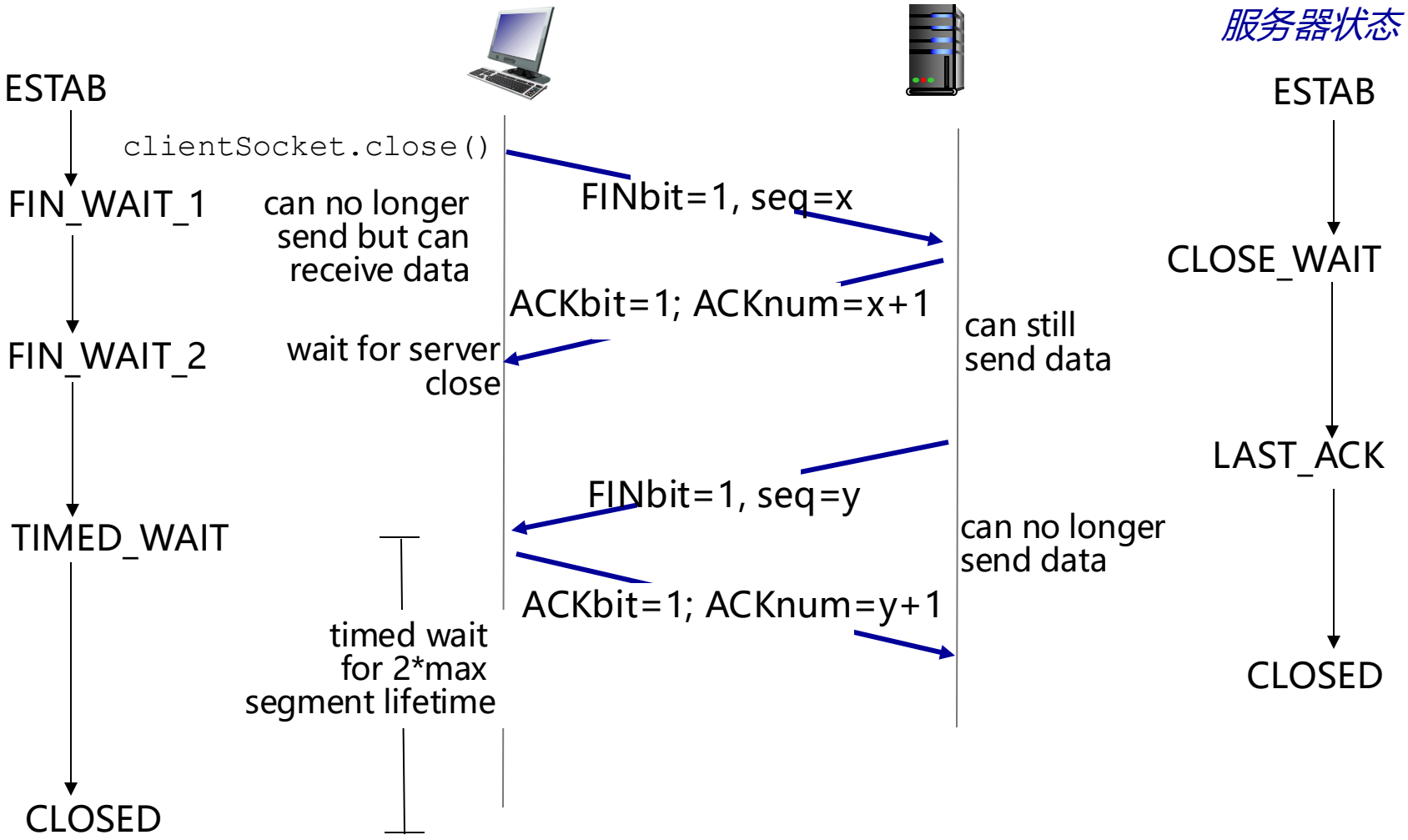
TCP三次握手


client state
LISTEN
SYNSENT
ESTAB
server state
LISTEN
SYN RCVD
ESTAB
random init seq num, x
send TCP SYN msg
SYNbit=1, Seq=x
random init seq num, y
send TCP SYNACK msg
SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1
send ACK for SYNACK;
allocate bufs. & vars.
may contain payload
ACKbit=1, ACKnum=y+1
indicate client is live
allocate bufs. & vars.

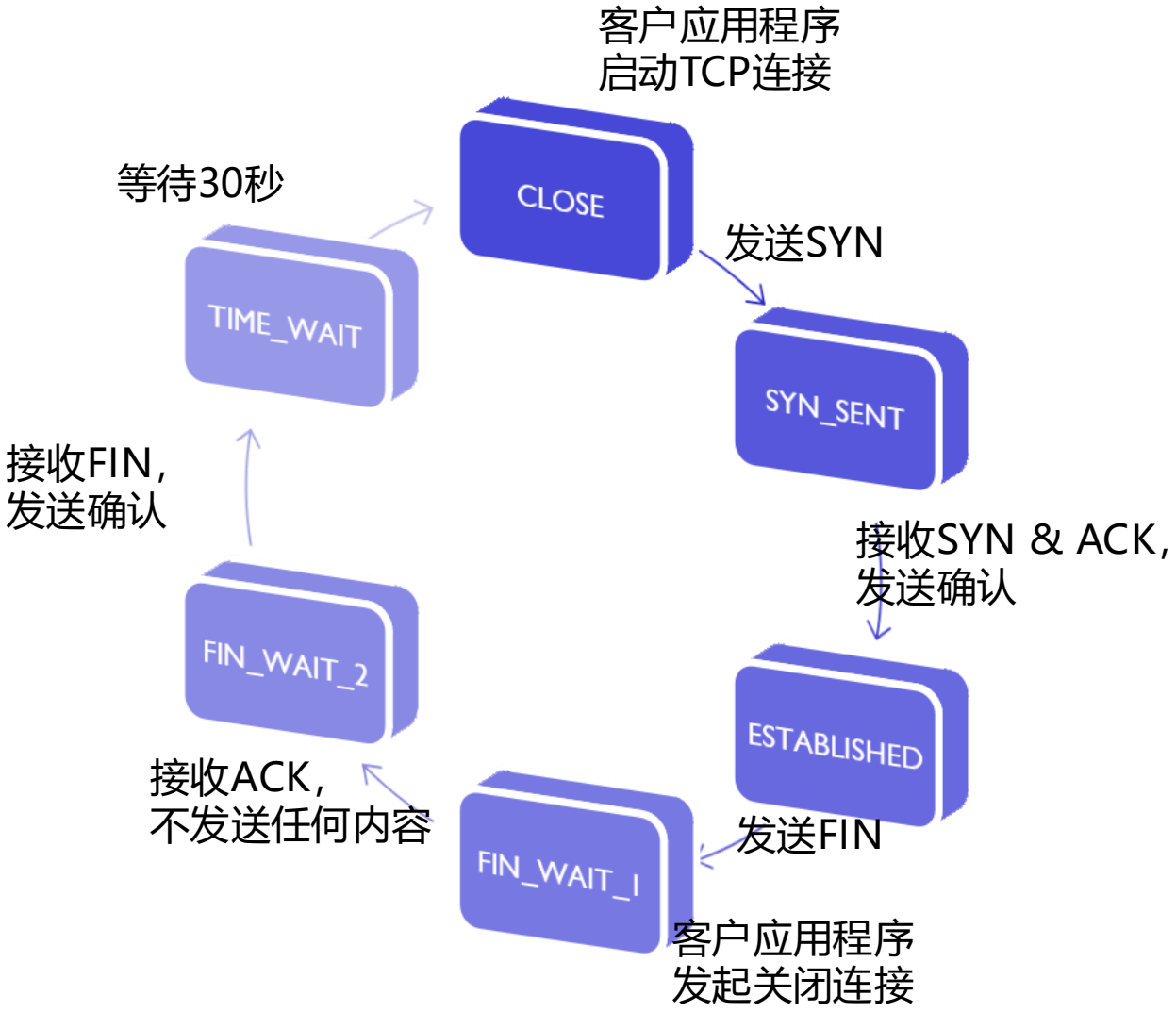
TCP:关闭连接

- 关闭连接
  - 客户端或服务器都可以终止
  - 每一侧关闭它们的连接侧
  - 发送FIN位= 1的TCP数据段
- 用ACK响应收到的FIN
  - 在接收FIN时，ACK可以与自己的FIN结合
- 同时进行的FIN交换可以处理

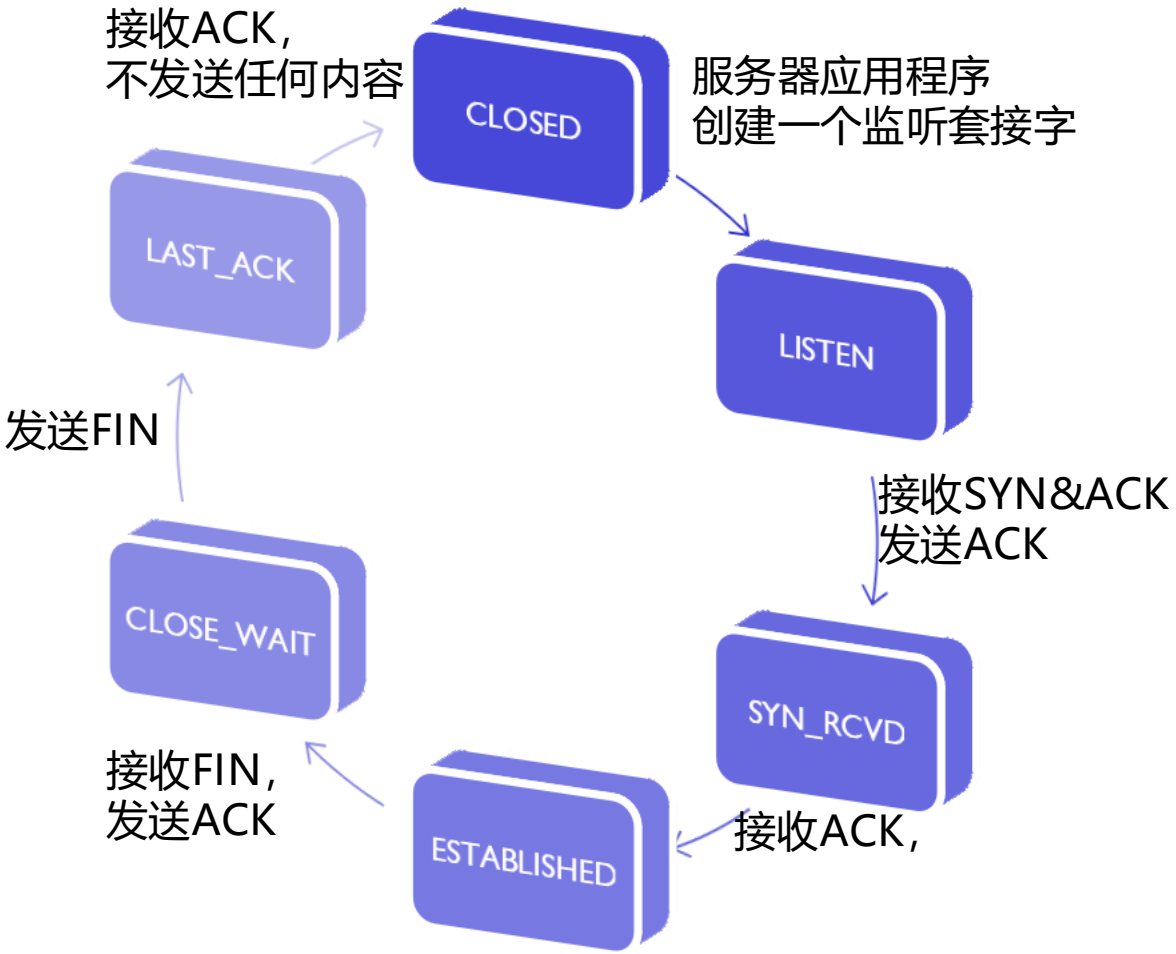
TCP:关闭连接


服务器状态
ESTAB
clientSocket.close()
FIN_WAIT_1
can no longer send but can receive data
FINbit=1, seq=x
ACKbit=1; ACKnum=x+1
CLOSE_WAIT
FIN_WAIT_2
wait for server close
can still send data
LAST_ACK
FINbit=1, seq=y
TIMED_WAIT
can no longer send data
ACKbit=1; ACKnum=y+1
timed wait for 2*max segment lifetime
CLOSED
CLOSED

客户端TCP访问的典型TCP状态序列



服务器端TCP访问的典型TCP状态序列



第3章:大纲

1. 3.1运输层服务
2. 3.2多路复用和多路分解
3. 3.3无连接传输:UDP
4. 3.4可靠运输原理
5. 3.5面向连接的传输:TCP

1.分段结构　2.可靠的数据传输　3.流量控制　4.连接管理

6. **3.6拥塞控制原理**
7. 3.7 TCP的拥塞控制

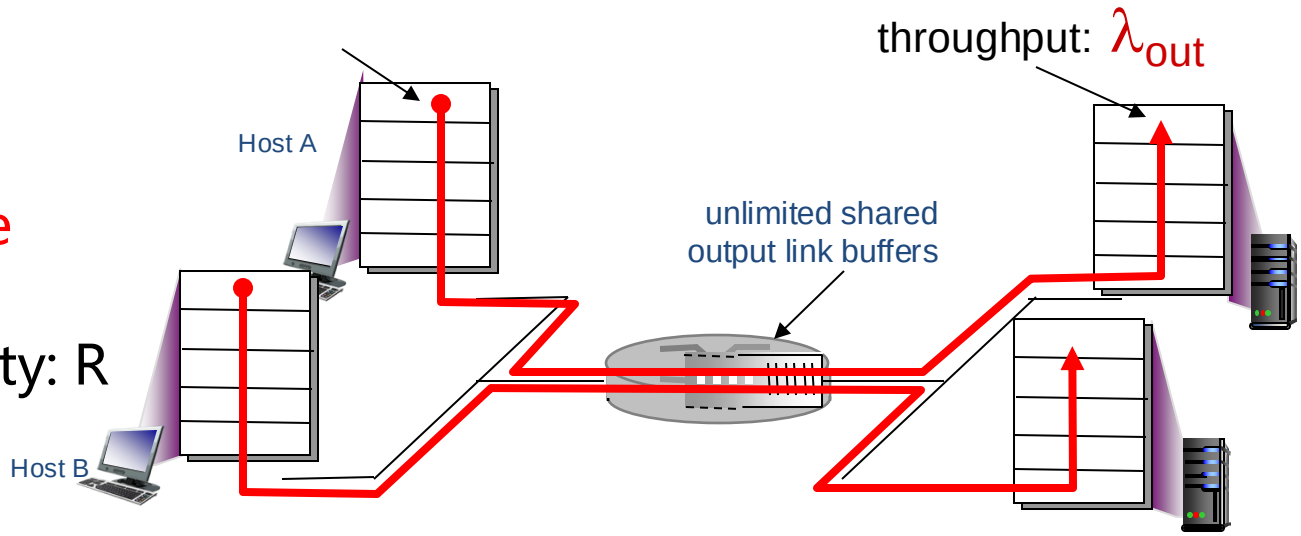
拥塞控制原理

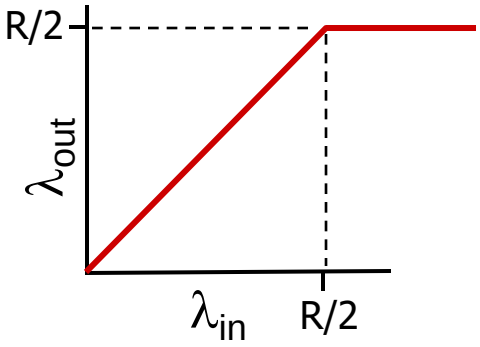
拥塞:

- 十大问题之一!
- 不同于流量控制!
- 拥塞控制:“太多来源发送太多数据，速度太快，网络无法处理”
- 表现形式:
  - 丢失的数据包(路由器缓冲区溢出)
  - 非常长的时延(在路由器缓冲区排队)

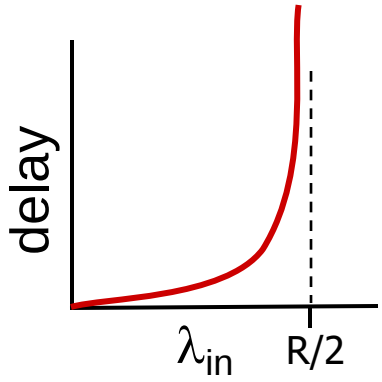
拥塞的原因/代价:场景1

- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no retransmission





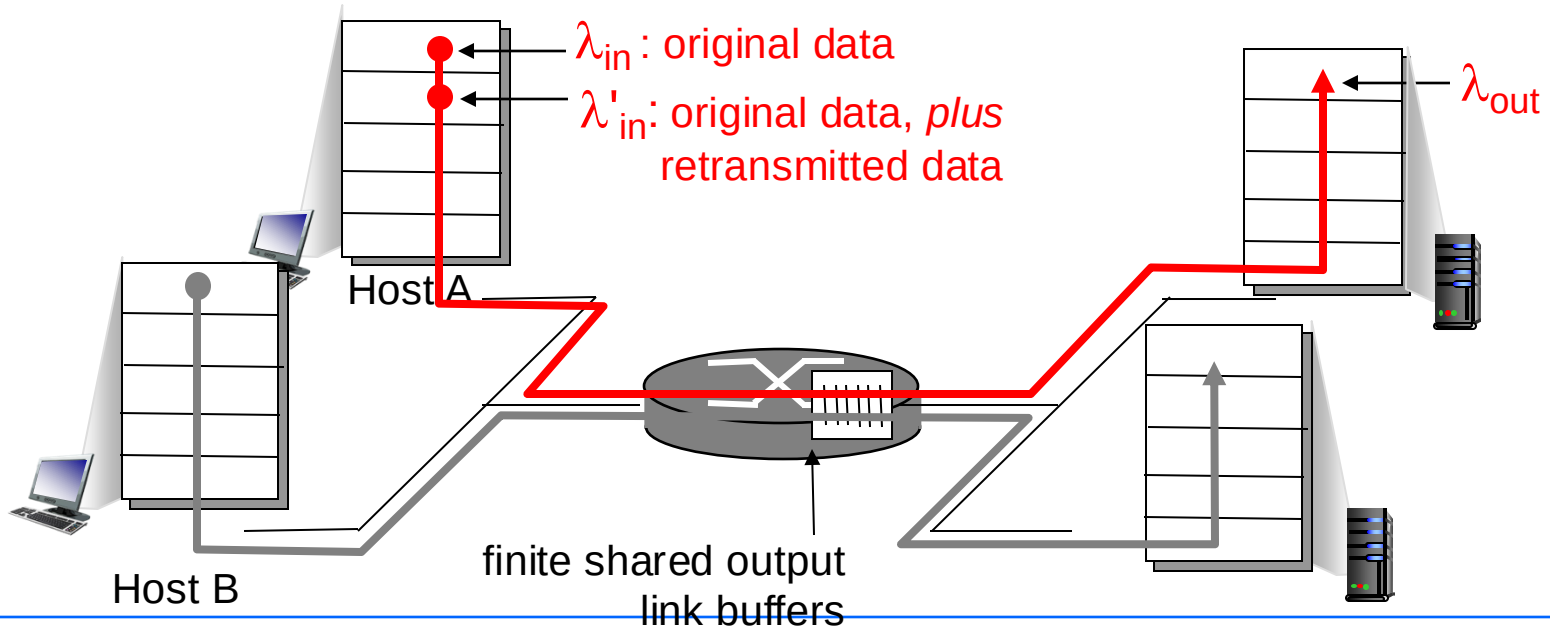
- maximum per-connection throughput: R/2



- large delays as arrival rate, λ_{in}, approaches capacity

拥塞的原因/代价:情景2

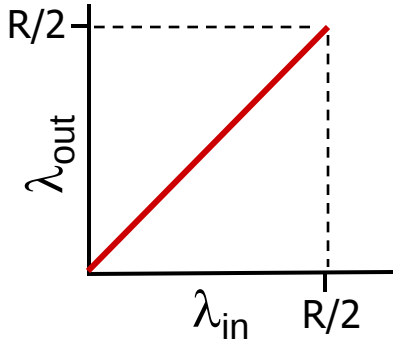
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

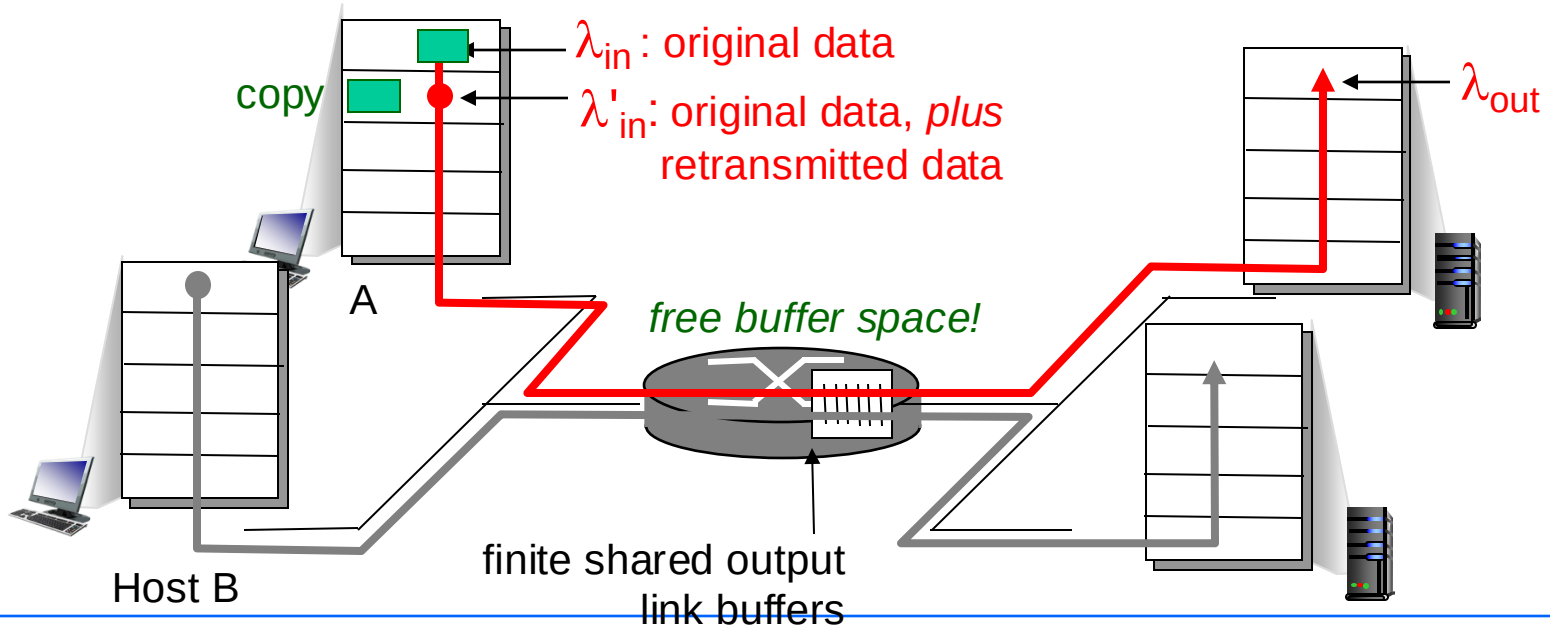


拥堵的原因/代价:情景2

idealization: perfect knowledge

- sender sends only when router buffers available



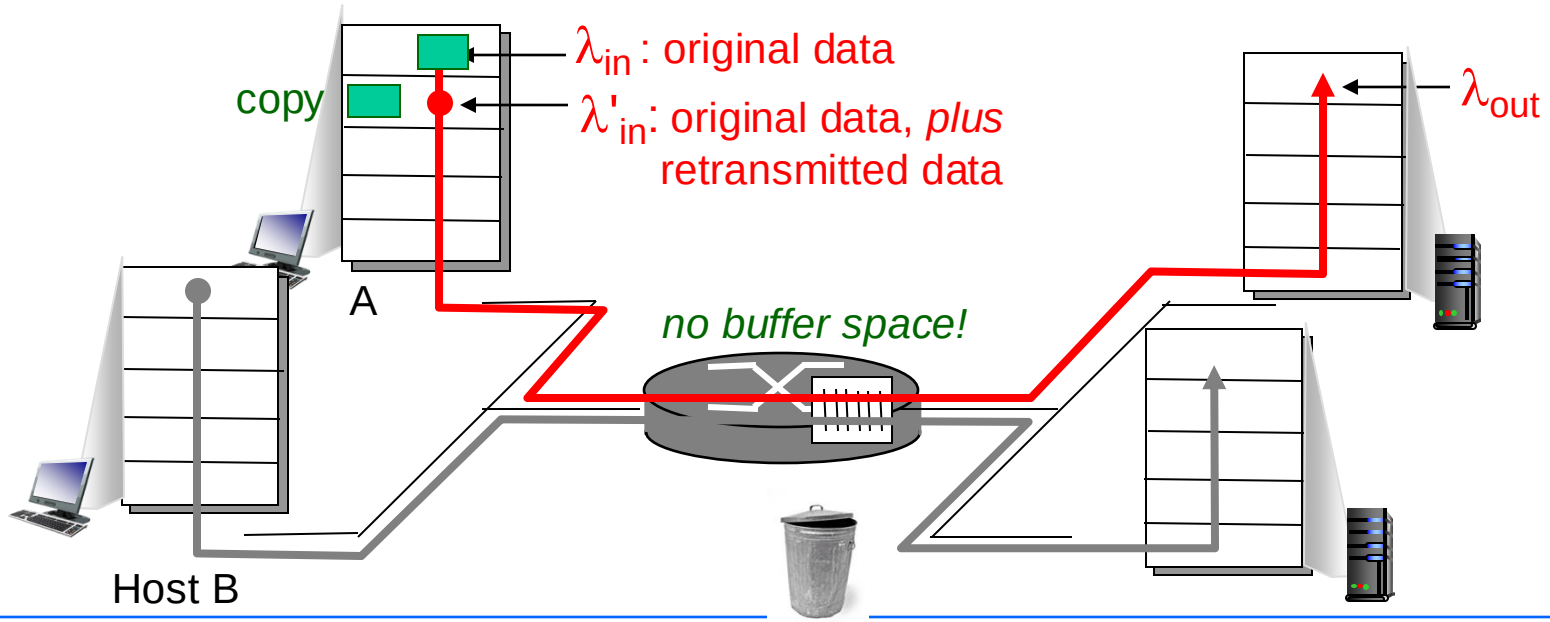


拥堵的原因/代价:情景2

Idealization: known loss

packets can be lost, dropped at router due to full buffers

- sender only resends if packet *known* to be lost

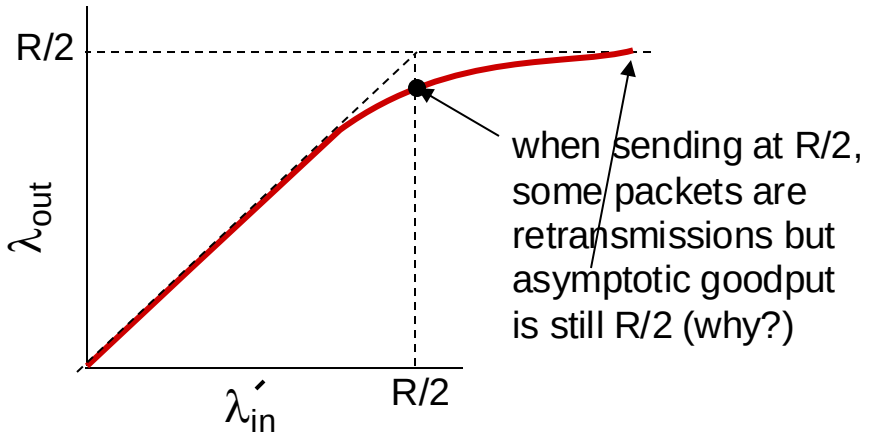


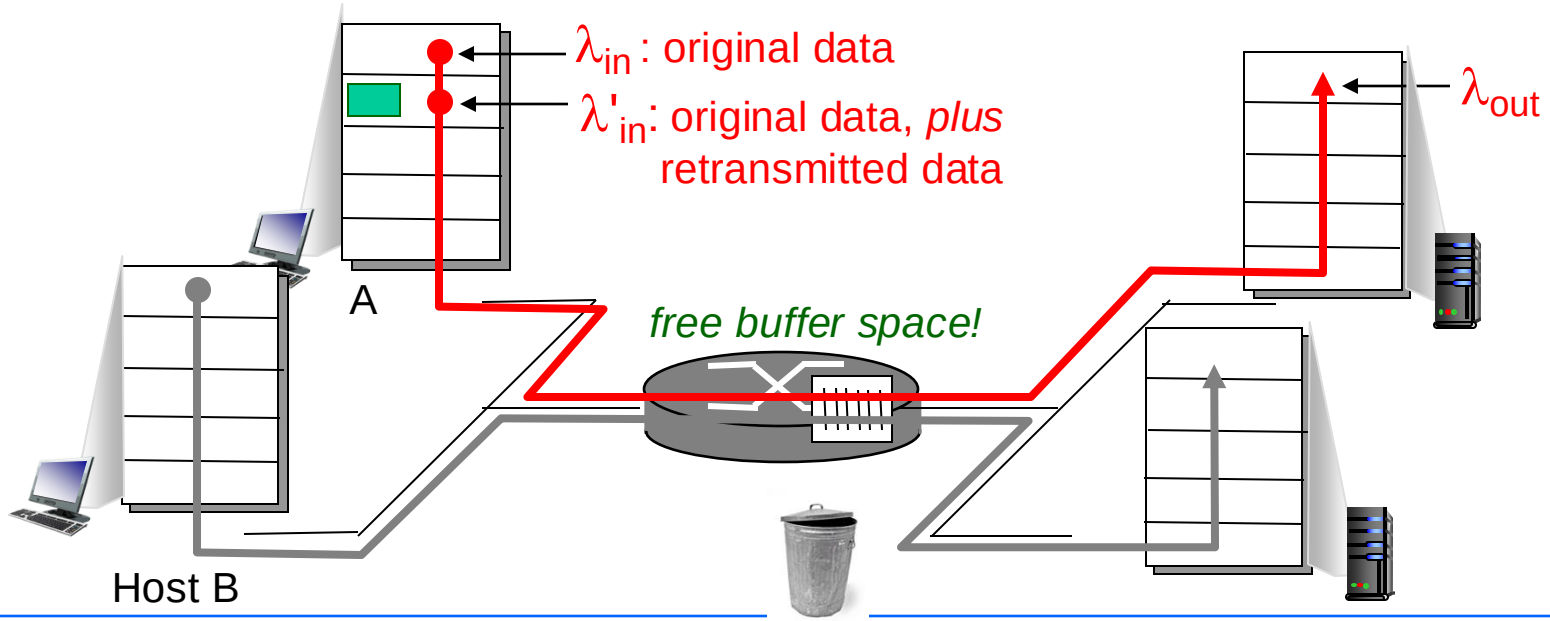
拥堵的原因/代价:情景2

Idealization: known loss

packets can be lost, dropped at router due to full buffers

- sender only resends if packet *known* to be lost

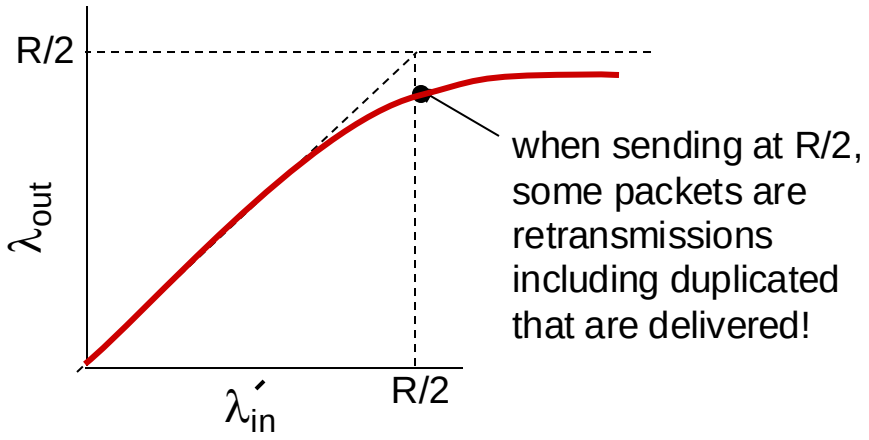


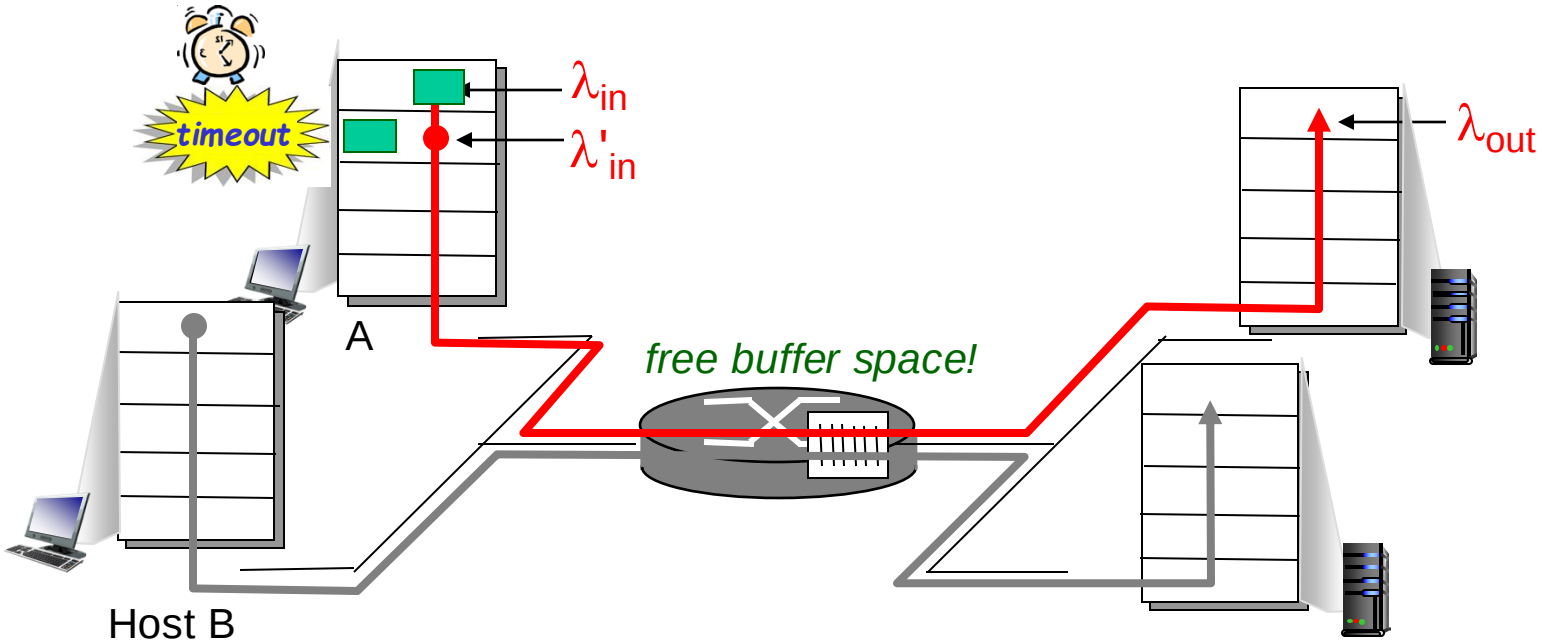


拥堵的原因/代价:情景2

Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered

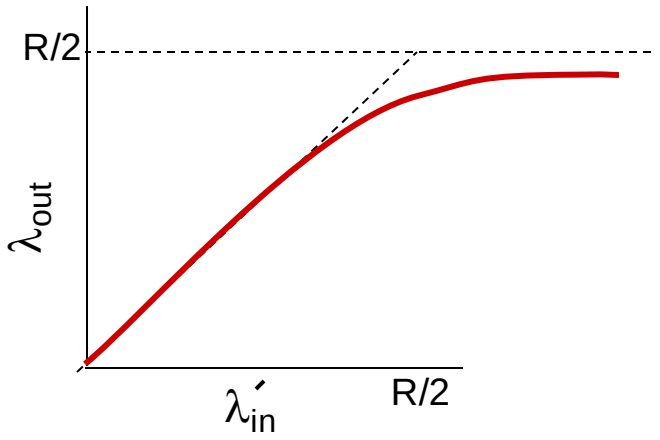




拥堵的原因/代价:情景2

Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered



"costs" of congestion:

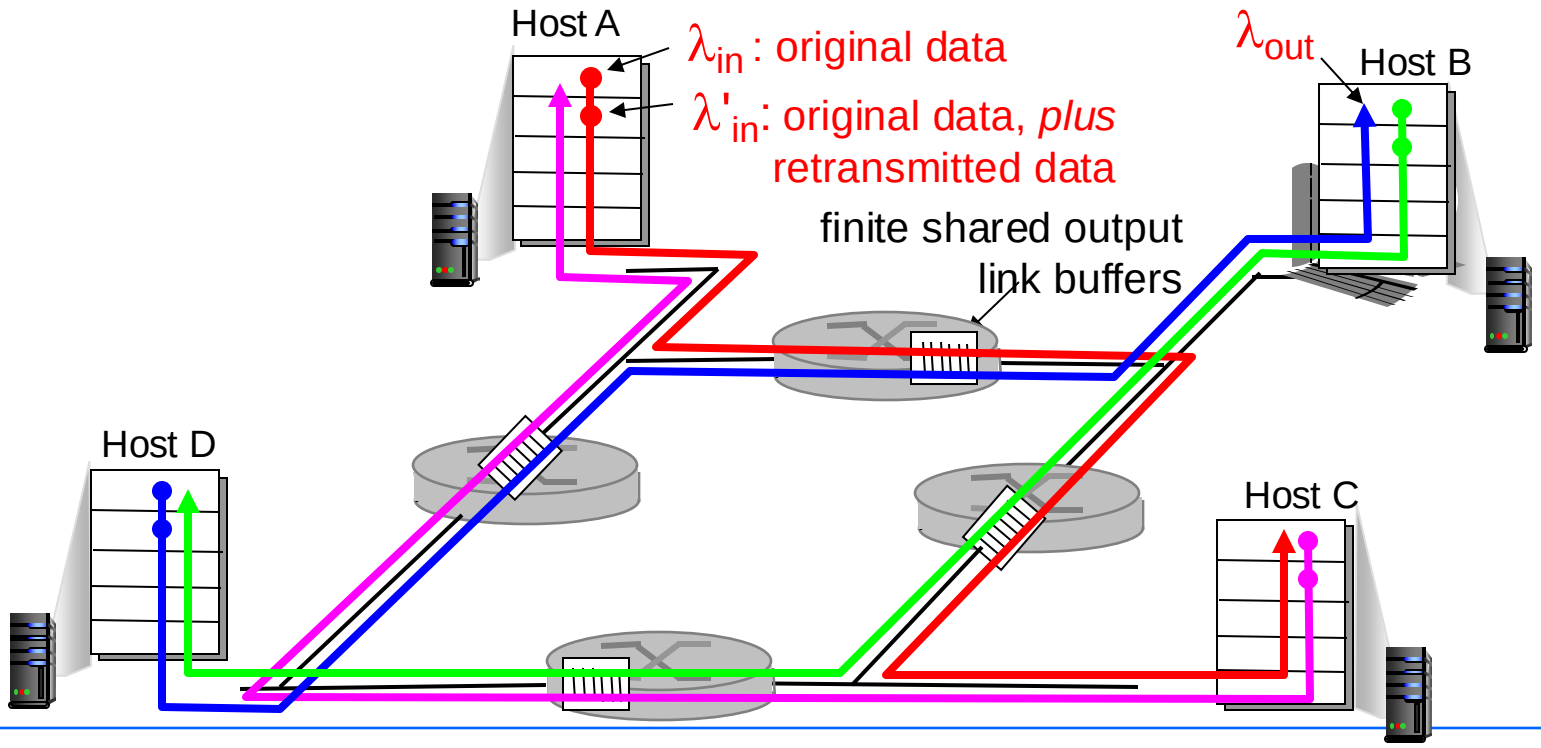
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

拥塞的原因/代价:情景3

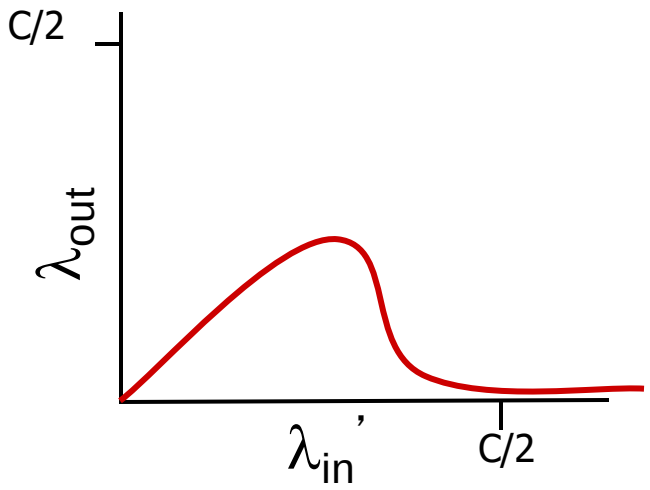
- four senders
- multihop paths
- timeout/retransmit

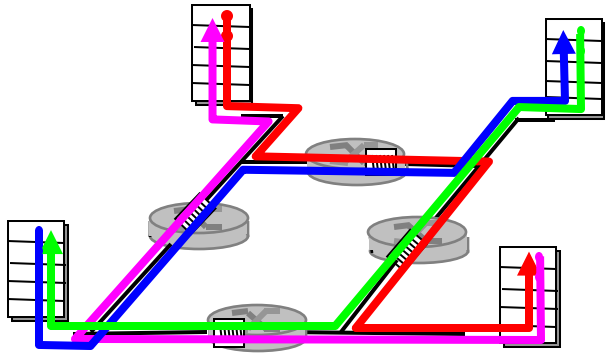
Q: what happens as λ_{in} and λ_{in}' increase ?

A: as red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput → 0



拥堵的原理/代价:情景3





another "cost" of congestion:

- when packet dropped, any "upstream" transmission capacity used for that packet was wasted!

拥塞控制方法

拥塞控制的两种主要方法:

端到端拥塞控制:

- 没有来自网络的明确反馈
- 对拥塞问题的了解来自于对数据丢失和延迟的推断
- TCP来解决

网络辅助拥塞控制:

- 路由器向终端系统提供反馈
  - 一个比特位的说明 (IBM SNA, DECNet, TCP/IP ECN, ATM)
  - 显式告知发送方所应采用的数据速率

案例研究:ATM ABR拥塞控制

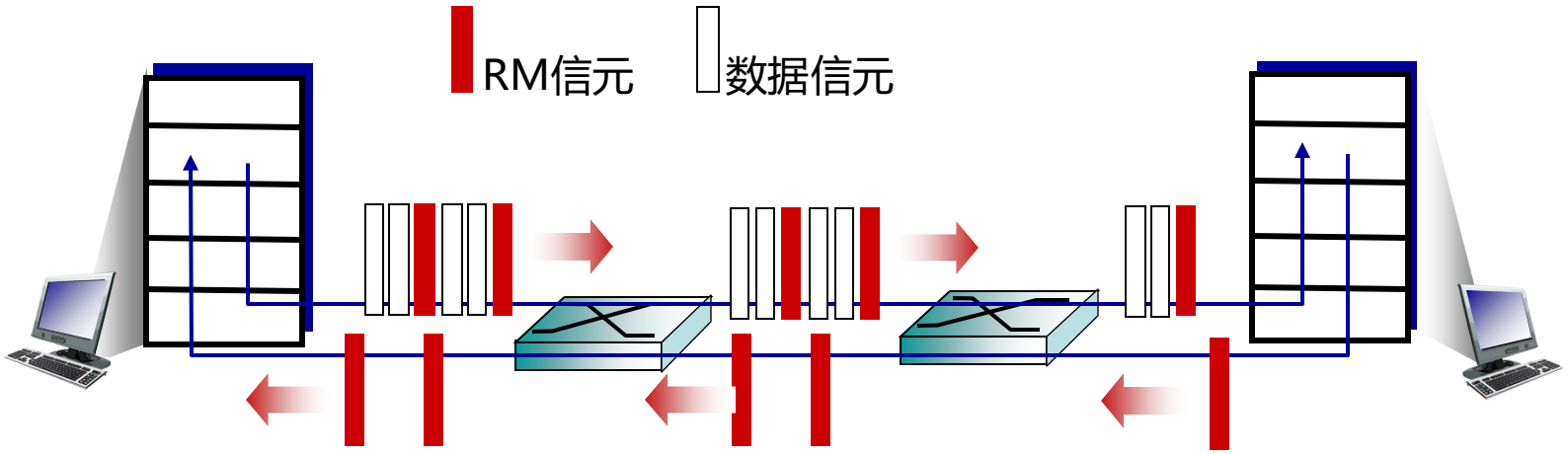
ABR:可用数据速率:

- “弹性服务”
- 如果发送方路径“负载不足”:
  - 发送方应使用可用带宽
- 如果发送方路径拥塞:
  - 发送方被限制到最低保证速率

RM(资源管理)单元:

- 由发送方发送，中间穿插数据单元
- RM单元中的位由开关设置(“网络辅助”)
  - *NI位:不增加速率(轻度拥塞)*
  - *CI位:拥塞指示*
- 接收方返回给发送方的RM信元，比特保持不变

案例研究:ATM ABR拥塞控制



- 数据信元中的EFCI(显式前向拥塞指示)位:在拥塞交换机中设置为1
  - 如果RM信元之前的数据信元设置了EFCI，则接收方在返回的RM信元中设置CI位
- RM信元中的两字节ER(显式速率)字段
  - 拥塞的交换机可能会降低单元中的er值
  - 发送方的发送速率，即路径上可支持的最大速率

第3章:大纲

1. 3.1运输层服务
2. 3.2多路复用和多路分解
3. 3.3无连接传输:UDP
4. 3.4可靠运输原理
5. 3.5面向连接的传输:TCP

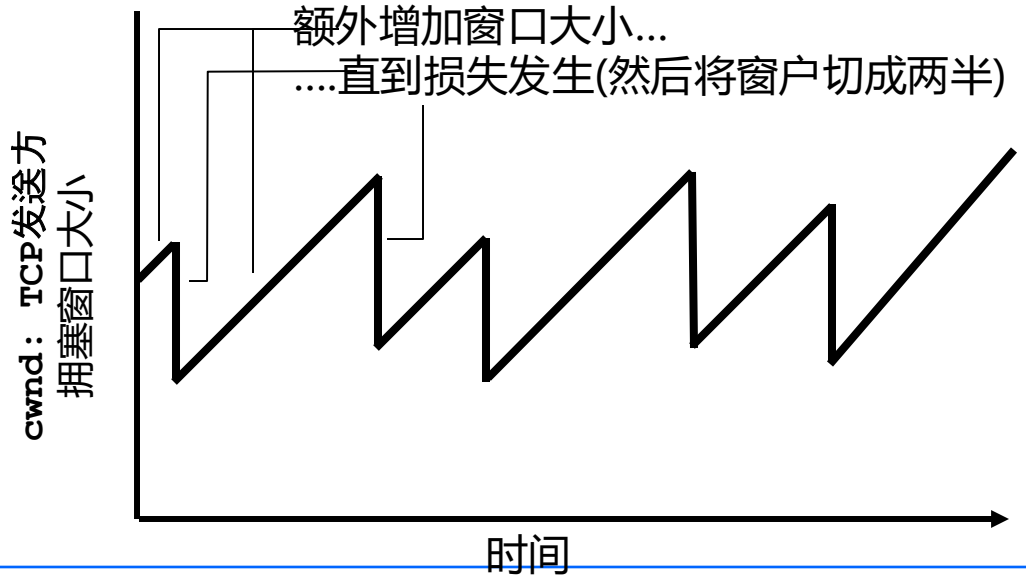
1.分段结构　2.可靠的数据传输　3.流量控制　4.连接管理

6. 3.6拥塞控制原理
7. **3.7 TCP的拥塞控制**

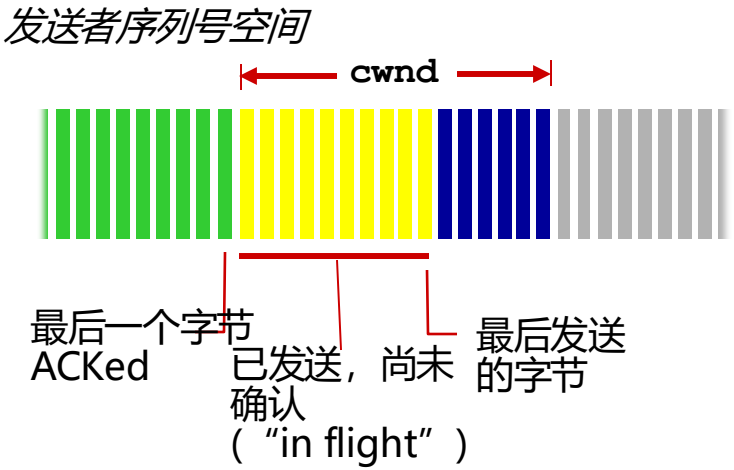
TCP拥塞控制:加法增加乘法减少

- *方法:发送方增加传输速率(窗口大小)，探测可用带宽，直到丢失发生*
  - *加法增加:每RTT增加1毫秒，直到检测到丢失*
  - *乘法递减：损失后将cwnd减半*

AIMD锯齿行为:探查对于带宽



TCP拥塞控制:详细信息



- **cwnd是感知网络拥塞的动态函数**
- 发送方限制传输:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$$

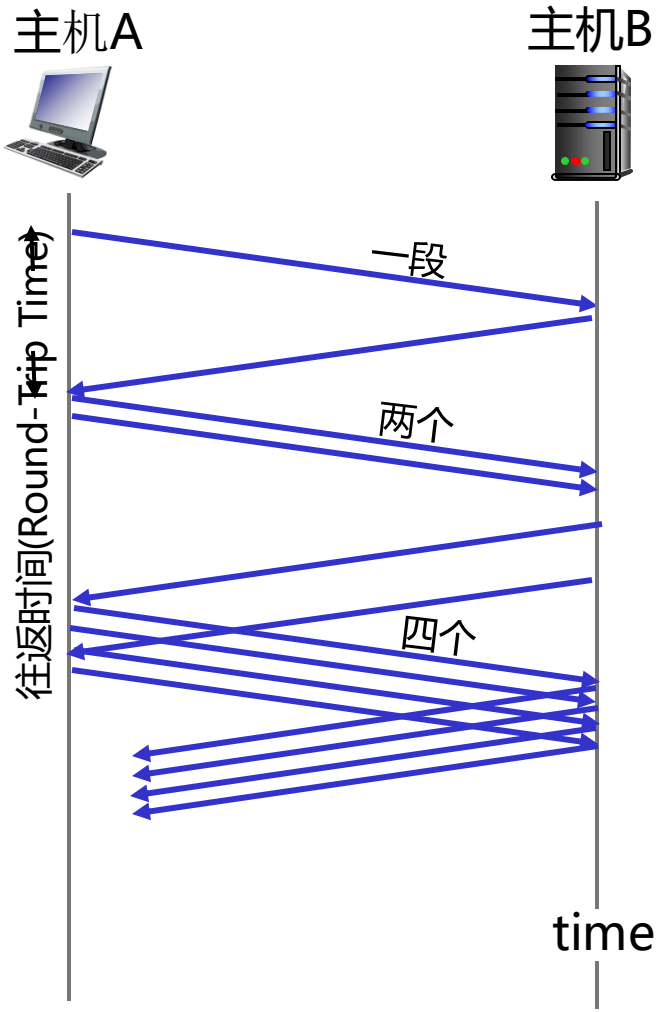
TCP发送速率:

- *大致是:发送cwnd字节，等待RTT的ack，然后发送更多的字节*

$$\text{速度} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ 字节/秒}$$

TCP慢启动

- 当连接开始时，以指数方式增加速率，直到第一次丢失事件:
  - 初始cwnd = 1 MSS
  - 每个RTT加倍cwnd
  - 每收到一个ACK就增加cwnd
- *总结:初始速率很慢，但以指数级快速上升*

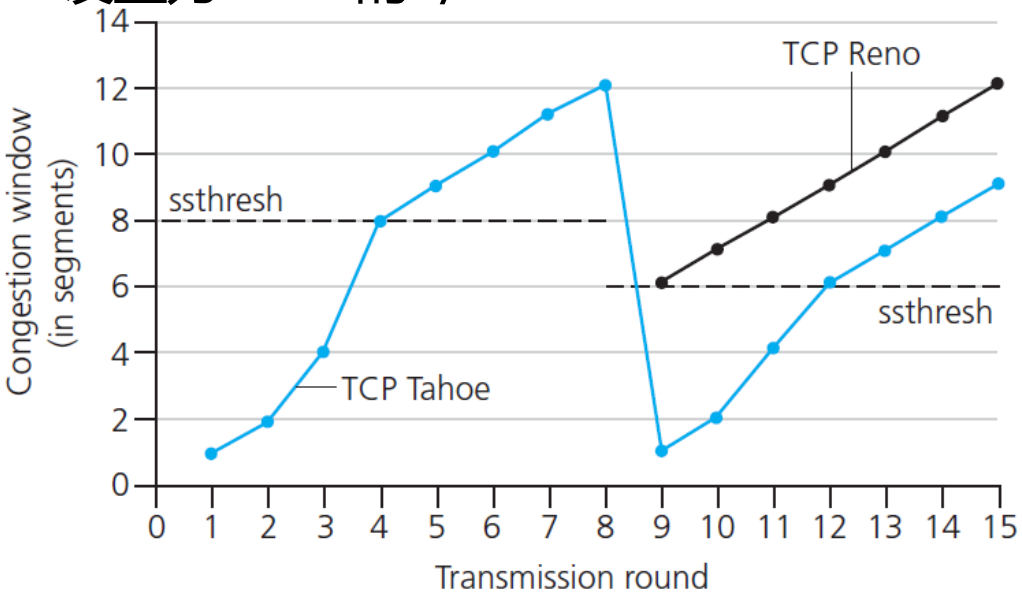


TCP:检测，对丢包做出反应

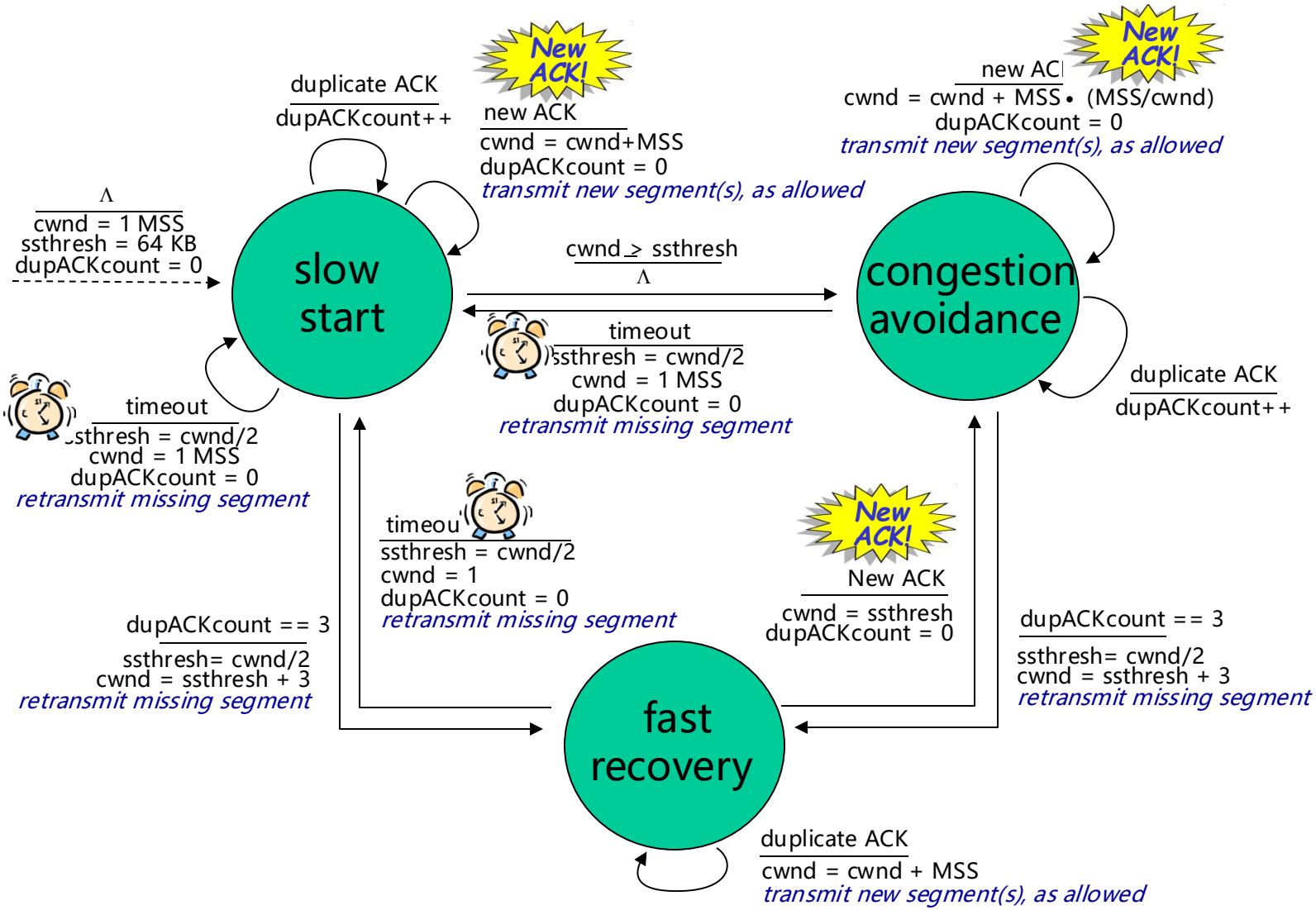
- 指数增长切换到线性增长
  - 变量ssthresh
- 丢失- TCP Tahoe(超时或3次重复确认)
  - cwnd设置为1 MSS
  - 发生丢包，ssthresh被设置为cwnd的1/2；
  - 窗口随后呈指数增长(如在慢启动中)到阈值，然后线性增长

TCP:检测，对丢包做出反应

- 由3个重复ack指示的丢包：TCP RENO
  - 重复确认表示网络能够传送某些数据段
  - cwnd减半（注意），然后线性增长
  - ssthresh设置为cwnd的1/2

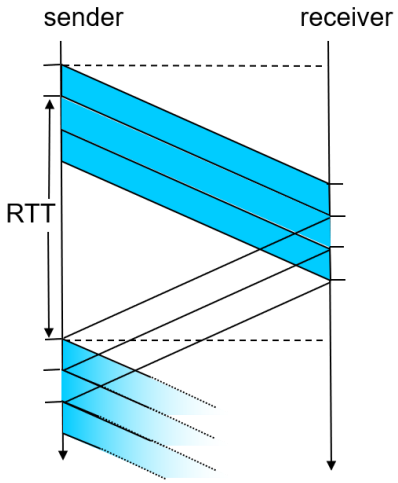


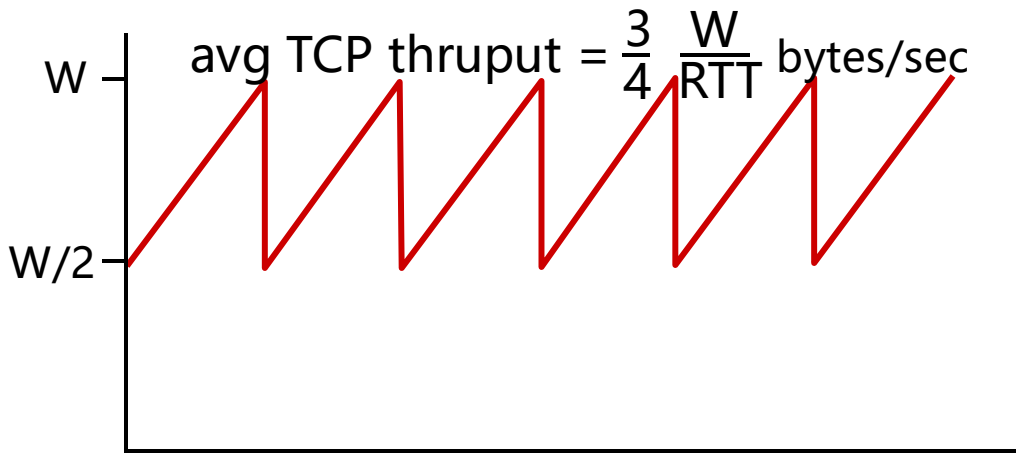
总结:TCP拥塞控制


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

TCP吞吐量

- 平均值。作为窗口大小函数的TCP吞吐量，RTT?
  - 忽略慢启动，假设总是有数据要发送
- w:发生丢失的窗口大小(以字节为单位)
  - 平均值。窗口大小(传输中的字节数)为3/4W
  - 平均值。TCP平均吞吐量 = 3/4* W/RTT Bps





TCP未来:通过“又长又粗的管道”的TCP

- 示例:1500字节数据段，100毫秒RTT，需要10 Gbps吞吐量
- 根据段丢失概率的吞吐量，L [Mathis 1997]:

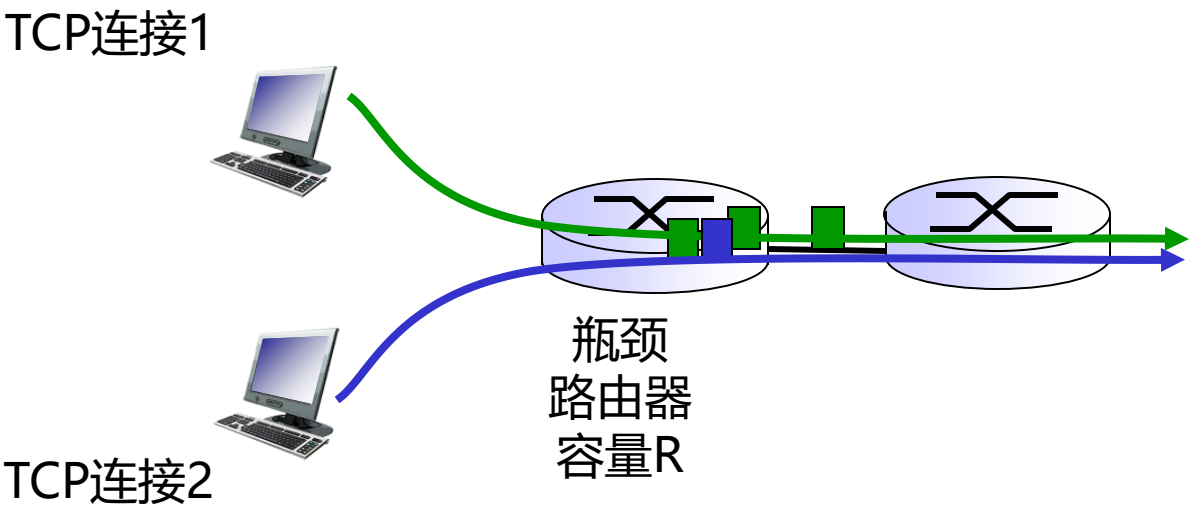
$$\text{TCP throughput} = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

→要实现10 Gbps的吞吐量，需要L = 2·10-10的丢失率—一个非常小的丢失率!

- 需要W = 83333个 “in flight”段
- 面向高速网的TCP新版本

TCP公平性

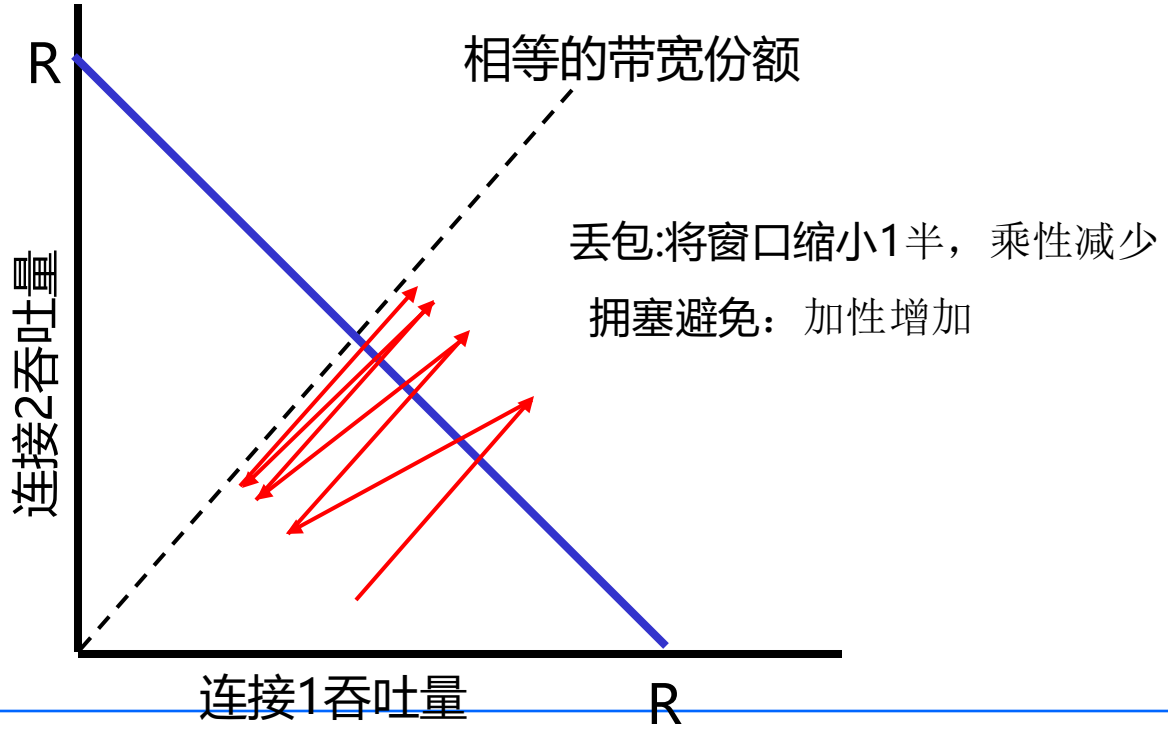
公平性目标:如果K个TCP会话共享带宽为R的同一瓶颈链路，则每个会话的平均速率应该为R/K



TCP为什么公平?

两场竞争性对话:

- 随着吞吐量的增加，累加增加的斜率为1
- 倍增减少成比例地减少吞吐量



公平(更多)

公平和UDP

- 多媒体应用程序通常不使用TCP
  - 不希望拥塞控制限制速率
- 请改用UDP:
  - 以恒定速率发送音频/视频，容许丢包

公平，并行TCP连接

- 应用程序可以在两台主机之间打开多个并行连接
- 网络浏览器就是这样做的
- 例如，具有9个现有连接的速率为R的链路:
  - 新应用程序要求1个TCP，获得速率R/10
  - 新应用要求11个TCP，得到R/2

第三章:总结

- 运输层服务背后的原则:
  - 多路复用、多路分解
  - 可靠的数据传输
  - 流量控制
  - 连接管理
  - 拥塞控制
- 在互联网上实现
  - 用户数据报协议(User Datagram Protocol)
  - 传输控制协议（Transmission Control Protocol)

接下来:

- 离开网络“边缘”(应用层、运输层)
- 进入网络“核心”

第三章:作业

- R6、R8、R12、R13、R15、R17
- P27、P40、P42
- 请从发现问题、分析问题、解决问题的角度,用思维导图或知识架构图等"图"的形式,小结可靠数据传输协议（从rdt1.0到传输控制协议（Transmission Control Protocol）可靠数传)的要点。