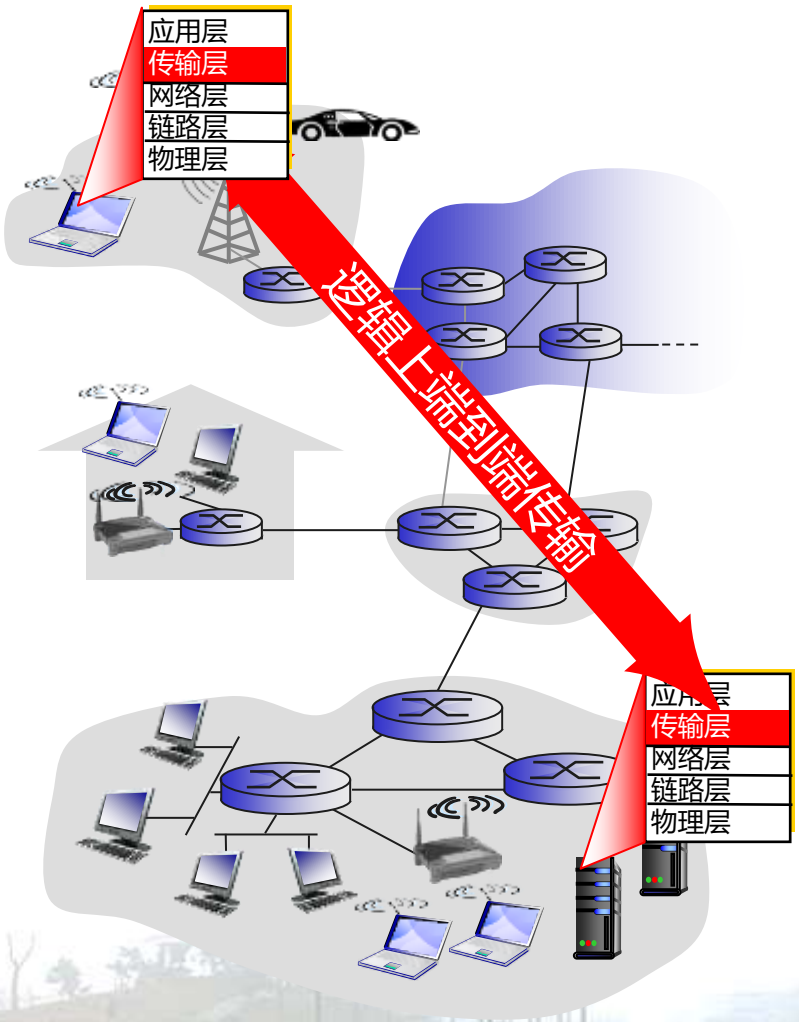
第三章 传输层

目录

传输层服务和协议

- 在不同主机上的应用程序进程之间提供逻辑的信道
- 传输层协议（软件实现）运行在终端系统上
  - 发送端：将应用程序的数据划分为分段（segment），交给网络层
  - 接收端：重组分段形成数据，交给应用层
- 多种传输层协议
  - 因特网传输层：TCP和UDP协议



传输层和网络层的关系

- 网络层： 主机之间的逻辑信道
- 传输层：进程 之间的逻辑信道
  - 基于网络层服务来实现

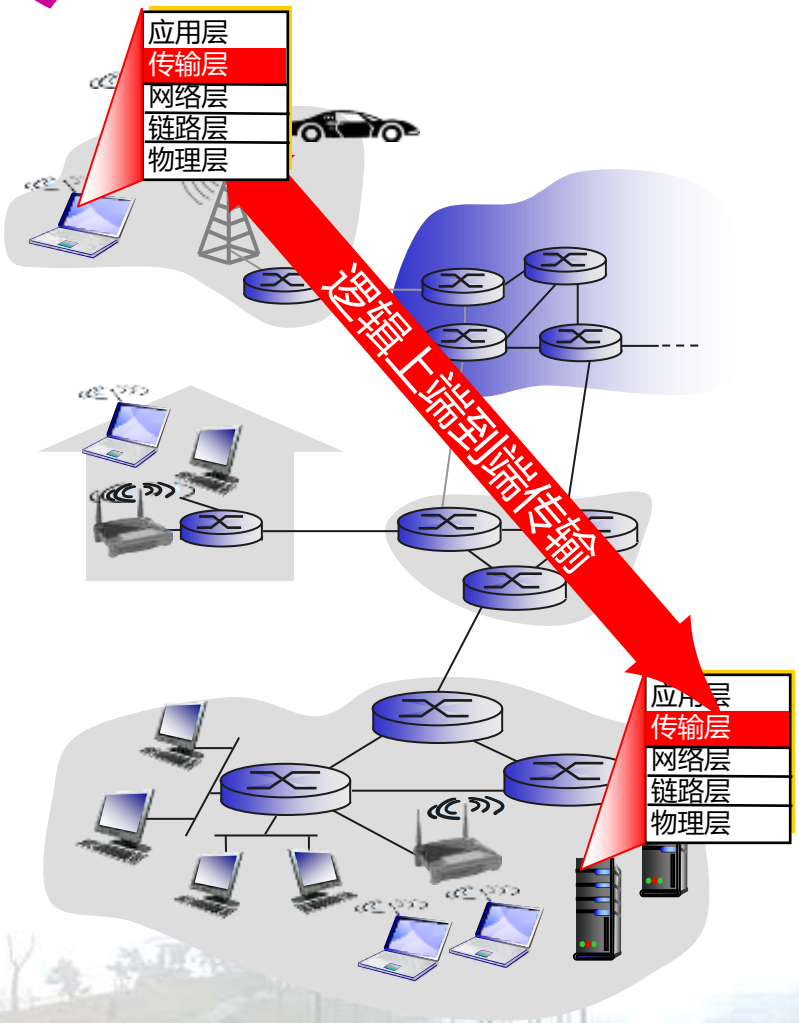
类比

1号宿舍楼里住的12名同学给2号宿舍楼里住的12名同学写信:

- 主机 = 宿舍楼
- 进程 = 同学
- 应用程序消息 = 装在信封里的信
- 传输层协议 = 1号楼和2号楼的楼管
- 网络层 = 邮政服务

因特网的传输层协议

- 可靠顺序传输 (TCP)
  - 拥塞控制
  - 流控制
  - 连接管理
- 不可靠的传输: UDP
  - IP协议的简单扩展
- 均不提供的服务
  - 时延保障
  - 带宽保障



目录

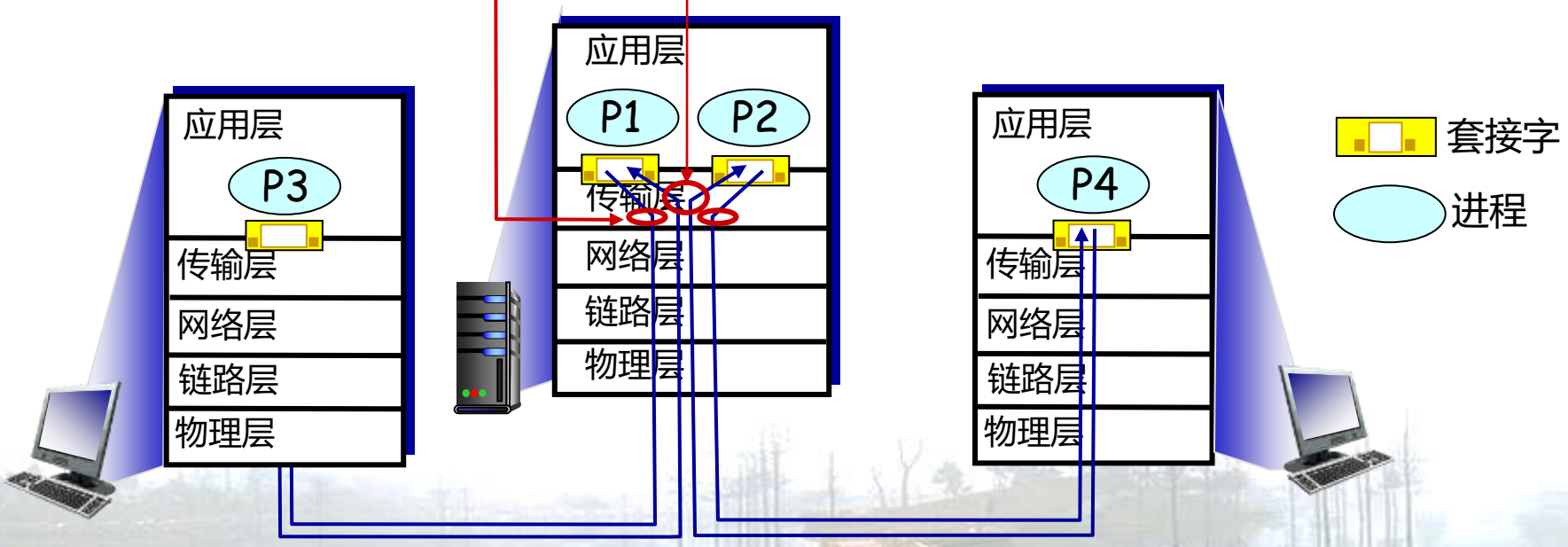
复用/解复用

在发送端复用：

处理多个套接字传来的数据，添加传输层协议头部（用于解复用）

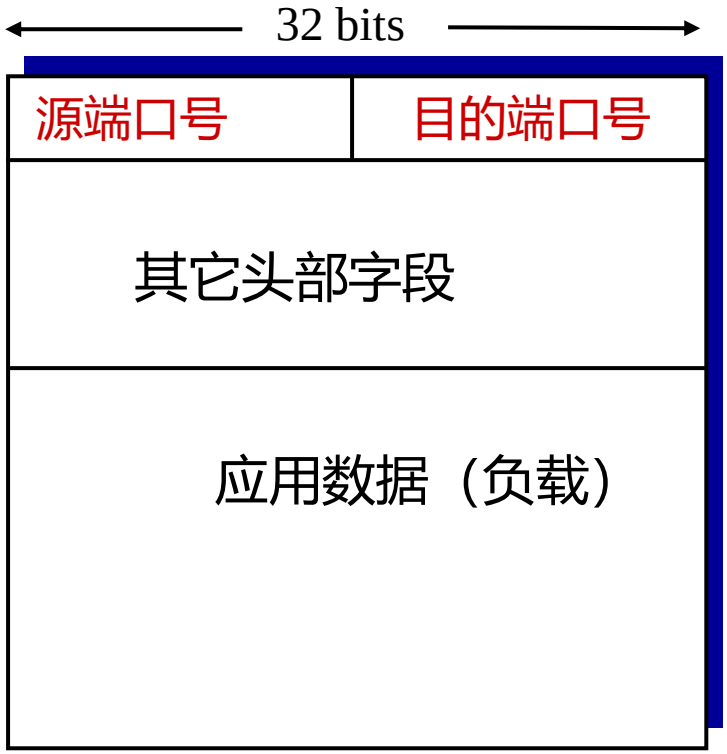
在接收端解复用：

基于传输层协议头部信息将分段交给相应的套接字



解复用工作原理

- 主机收到IP报文
  - 报文包含源IP地址和目的IP地址
  - 每个报文包含一个传输层协议的分段
  - 每个分段包含源端口号和目的端口焊
- 主机使用IP地址+端口号将分段送到相应的套接字
  - 目的地址+目的端口号二元组标识UDP套接字



TCP/UDP 分段格式

无连接解复用

- 上一章：创建套接字时绑定本地端口号：

```
DatagramSocket mySocket1 =
  new DatagramSocket(12534);
```

- 上一章：构造报文由UDP套接字发送时，必须指定
  - 目的IP地址
  - 目的端口号

- 收到UDP分段时：
  - 检查分段头部的目的端口号
  - 将UDP分段送到绑定该端口号的套接字

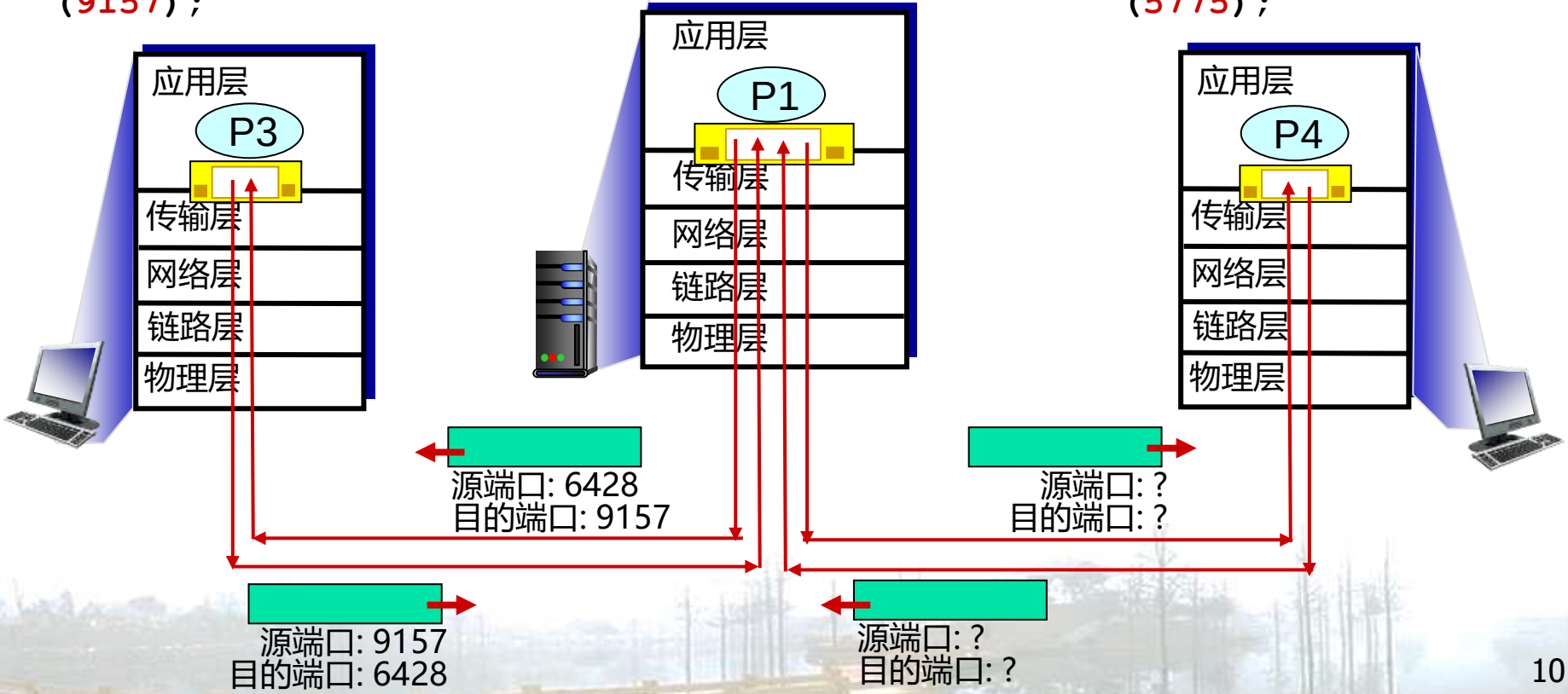
具有相同目的端口号，但来自不同源IP地址的报文被送到目的主机上相同的套接字

无连接解复用：举例

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

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

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



面向连接的解复用

- TCP套接字由四元组标识:
  - 源IP地址
  - 源端口号
  - 目的IP地址
  - 目的端口号
- 解复用：使用四元组将分段送到相应的套接字
- 服务器同时维持多个TCP套接字:
  - 每个套接字由其四元组标识
- 多个客户端连接web服务器时，为每个客户端创建一个套接字
  - 非持久HTTP为每个请求创建一个套接字

面向连接的解复用

- 服务器创建欢迎套接字，绑定12000端口

```
serverSocket.bind(('localhost',12000))
serverSocket.listen(1)
```

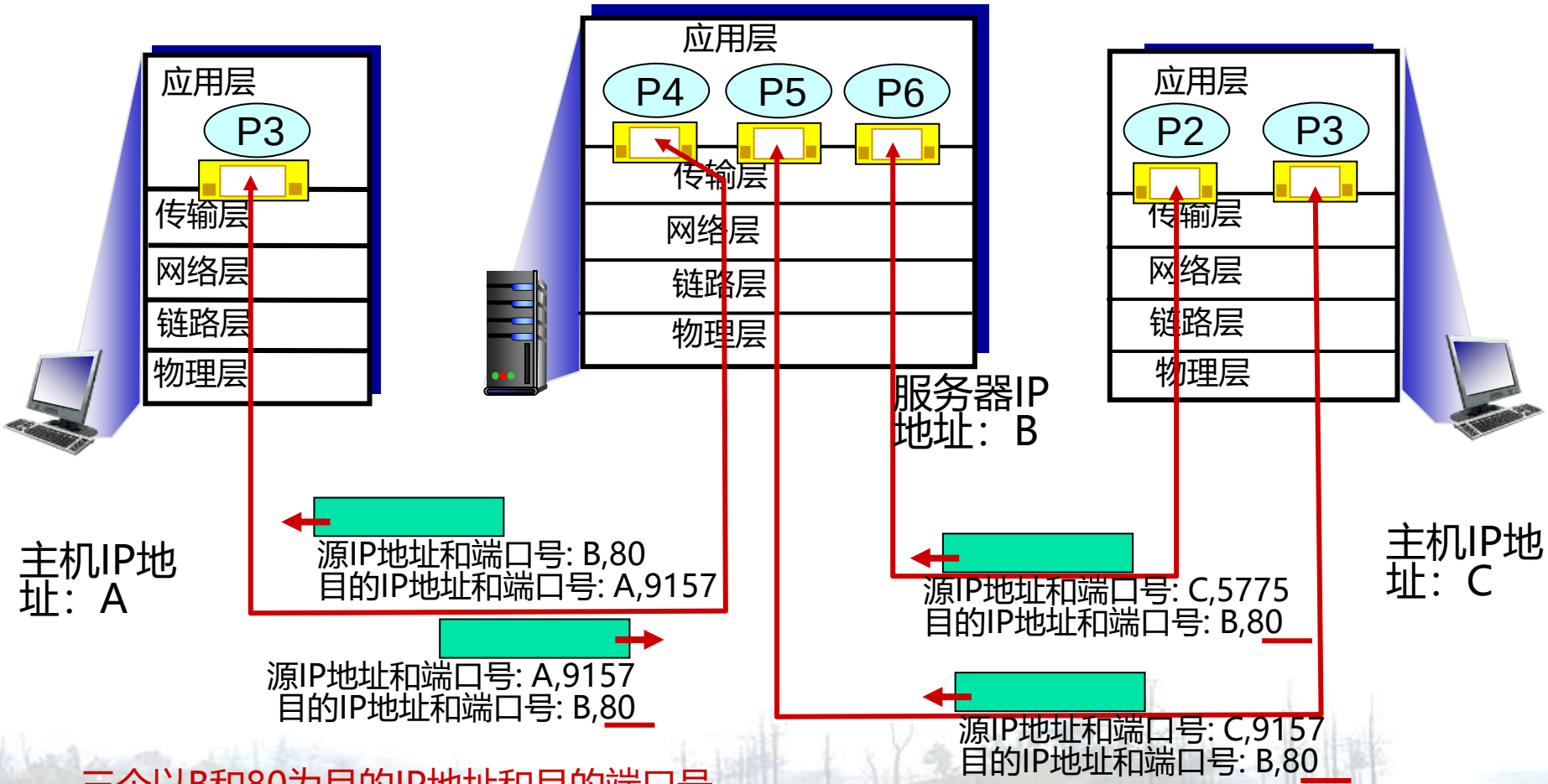
- 客户端创建套接字，连接服务器IP地址+端口

```
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,12000))
```

- 服务器接收连接连接请，创建四元组定义的连接套接字

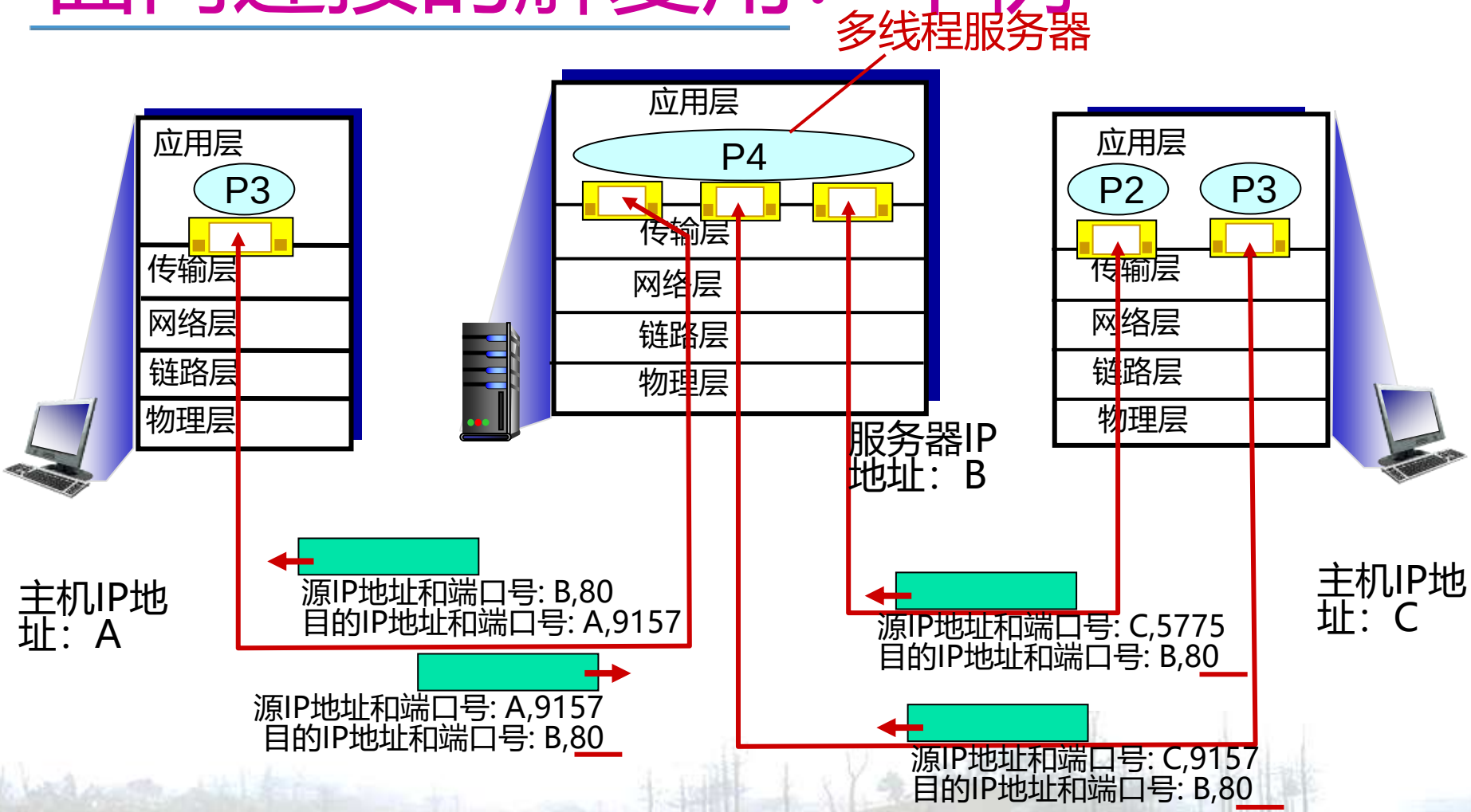
```
connectionSocket, addr = serverSocket.accept()
```

面向连接的解复用：举例



三个以B和80为目的IP地址和目的端口号的分段被送到不同的套接字

面向连接的解复用：举例

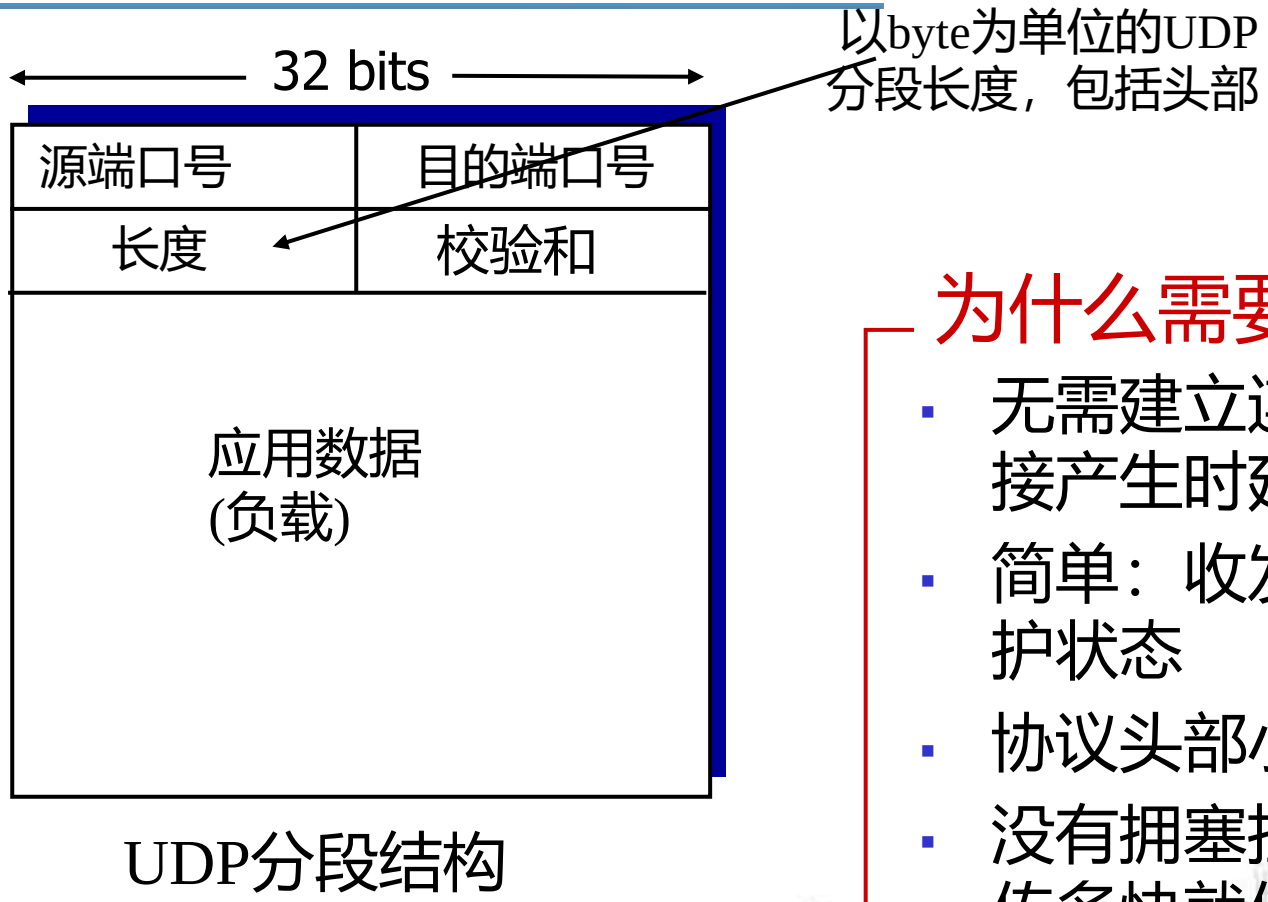

多线程服务器
应用层
P3
传输层
网络层
链路层
物理层
应用层
P4
传输层
网络层
链路层
物理层
服务器IP地址：B
应用层
P2
P3
传输层
网络层
链路层
物理层
主机IP地址：A
源IP地址和端口号: B,80
目的IP地址和端口号: A,9157
源IP地址和端口号: A,9157
目的IP地址和端口号: B,80
源IP地址和端口号: C,5775
目的IP地址和端口号: B,80
源IP地址和端口号: C,9157
目的IP地址和端口号: B,80
主机IP地址：C

目录

UDP：用户数据报协议[RFC 768]

- 朴素的因特网传输层协议
- 尽力而为的服务，UDP分段可能：
  - 丢失
  - 乱序到达目的地主机的应用层
- 无连接:
  - UDP发送端和接收端之间无需握手
  - 主机对每个UDP分段的处理是独立于其它分段的
- UDP的应用:
  - 流媒体(可容忍丢包、带宽敏感)
  - DNS
  - SNMP（简单网管协议）
- 在UDP协议上怎样实现可靠传输：
  - 由应用层提供可靠传输
  - 不同的应用可能有不同的差错恢复方法

UDP：分段头部



UDP分段结构

为什么需要UDP?

- 无需建立连接（建立连接产生时延）
- 简单：收发双方无需维护状态
- 协议头部小
- 没有拥塞控制：可以想传多快就传多快

UDP的校验和

目的：检测收到分段中的错误（例如：比特翻转）

发送端:

- 将整个分段内容，包括（伪）头部视为16-bit整数序列
- 校验和: 对分段内容求和取反
- 发送端将计算出的校验和写入UDP分段的校验和字段

接收端:

- 基于接收到的分段内容（包括校验和字段）求和
- 检查计算结果:
  - 包含‘0’ – 检测到错误
  - 全‘1’ – 未检测到错误，但是仍然可能有错误。

校验和计算：举例

例：两个16-bit数相加

	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	(1)	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
求和	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
取反，得到校验和	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

当产生进位时，进位的比特必须加到最末位

伪头部

- UDP计算校验和时，包含一部分IP头部的字段
  - 通过再UDP头部添加伪头部实现
  - 伪头部并不真正传输

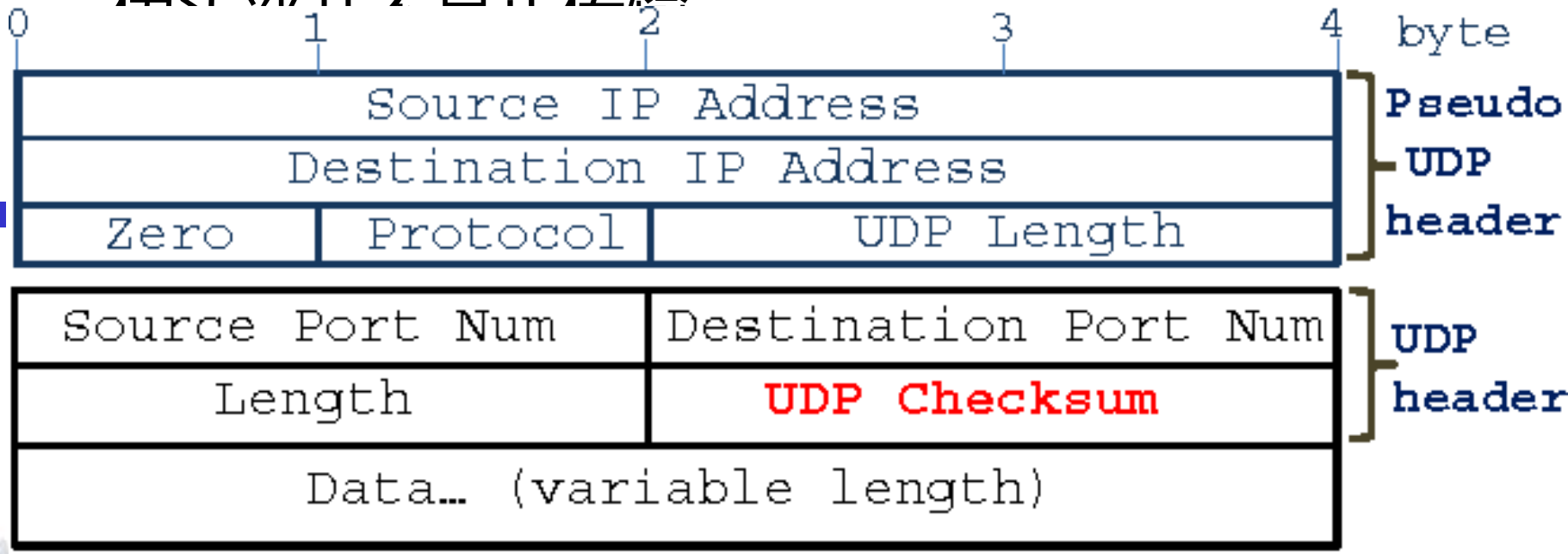


Fig. 1. UDP datagram and pseudo header.

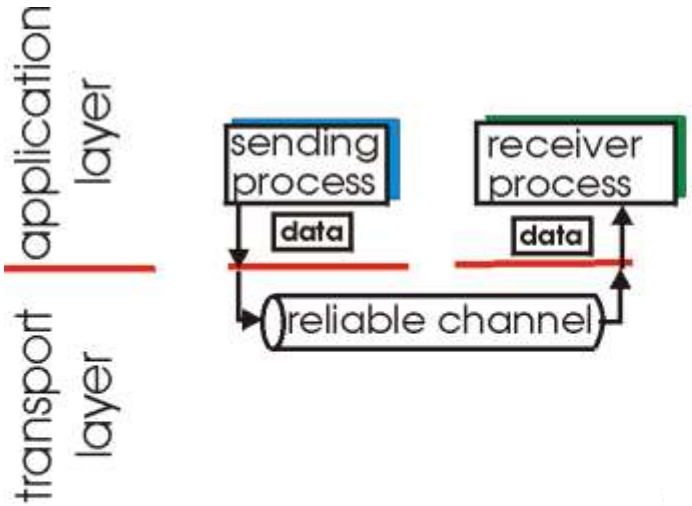
UDP校验和

- 很多链路层也提供了差错校验，为何在UDP计算校验和?
  - 不保证源到目的路径上的所有链路都实现差错校验
  - 在路由器内部也可能发生bit翻转
- 端到端原则

目录

可靠数据传输原理

- 在应用层、传输层、链路层都非常重要
  - 网络领域10大问题之一

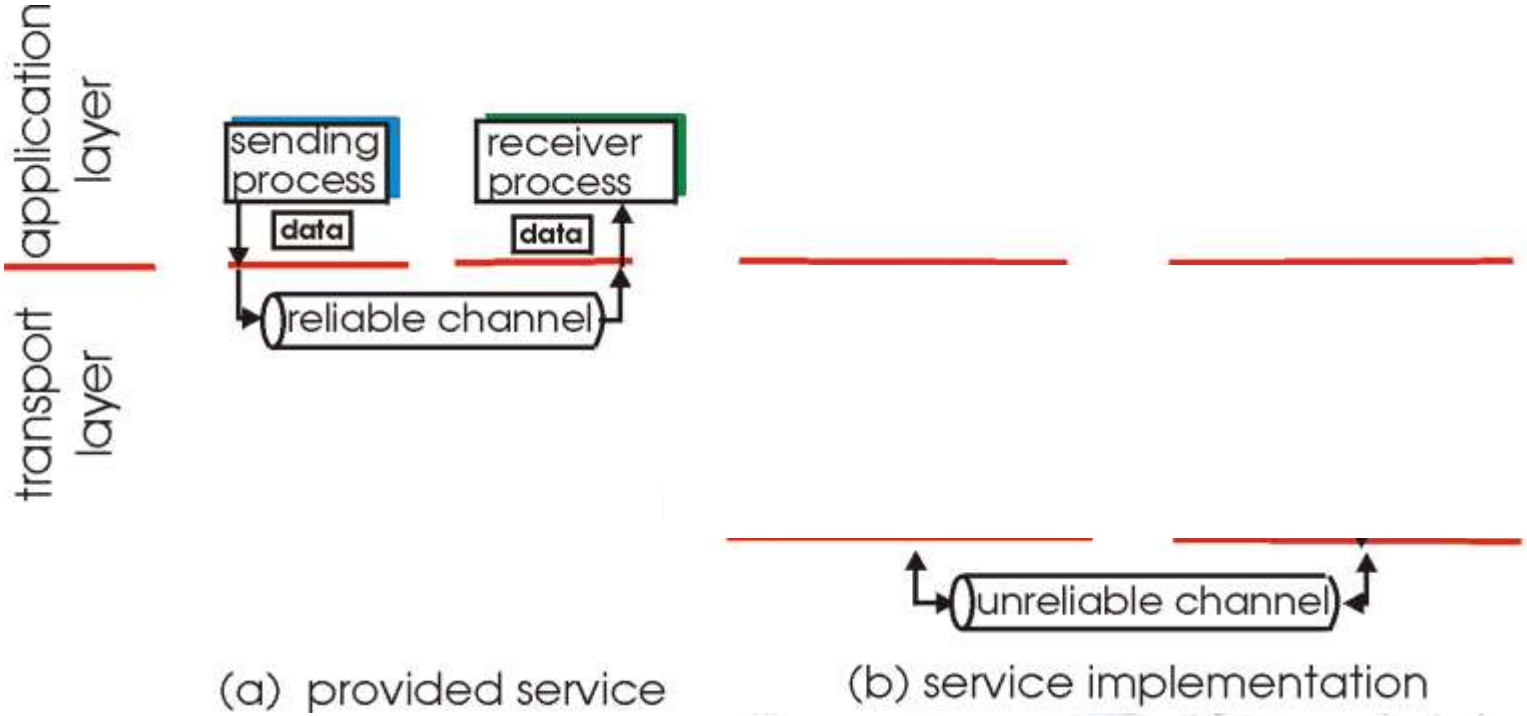


(a) provided service

- 可靠传输协议（rdt）的复杂程度取决于不可靠信道的特征（怎样不可靠）

可靠数据传输原理

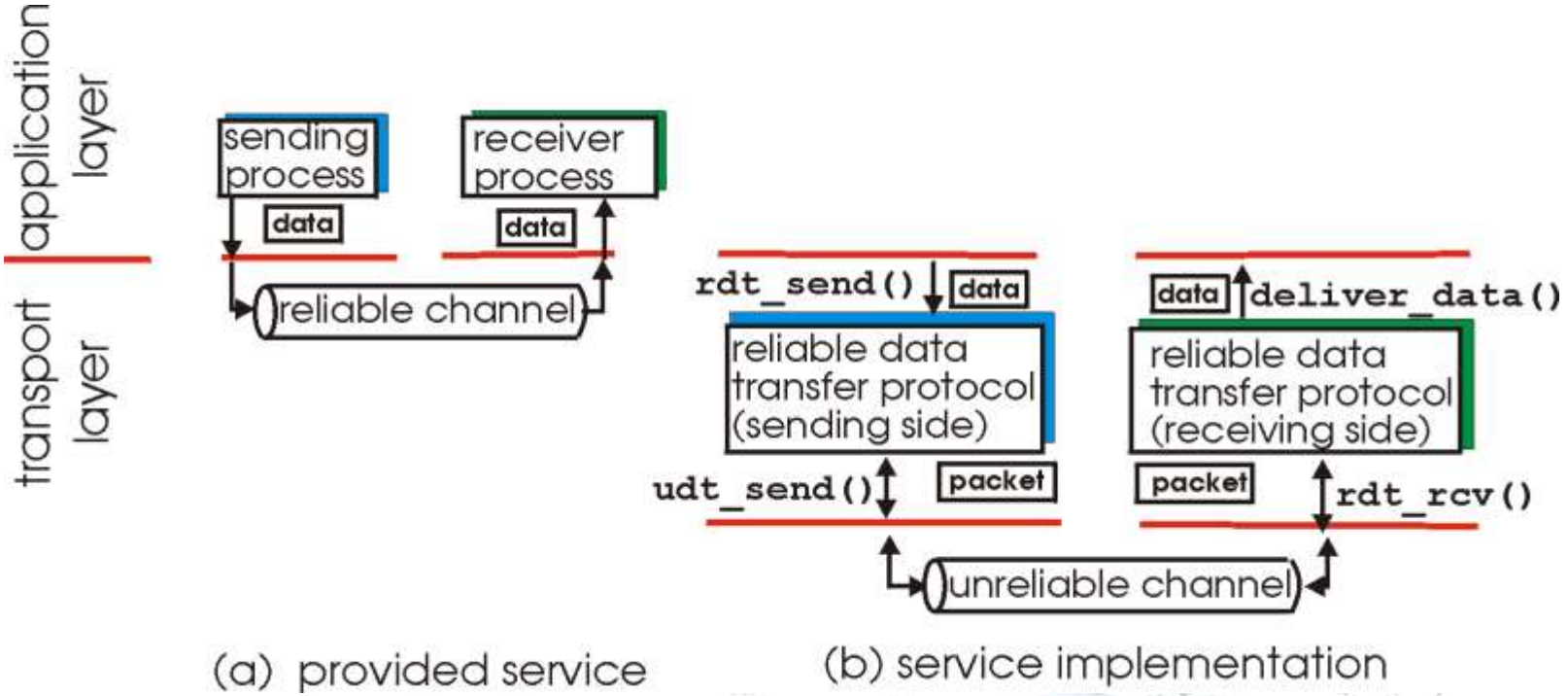
- 在应用层、传输层、链路层都非常重要
  - 网络领域10大问题之一



- 可靠传输协议（rdt）的复杂程度取决于不可靠信道的特征（怎样不可靠）

可靠数据传输原理

- 在应用层、传输层、链路层都非常重要
  - 网络领域10大问题之一

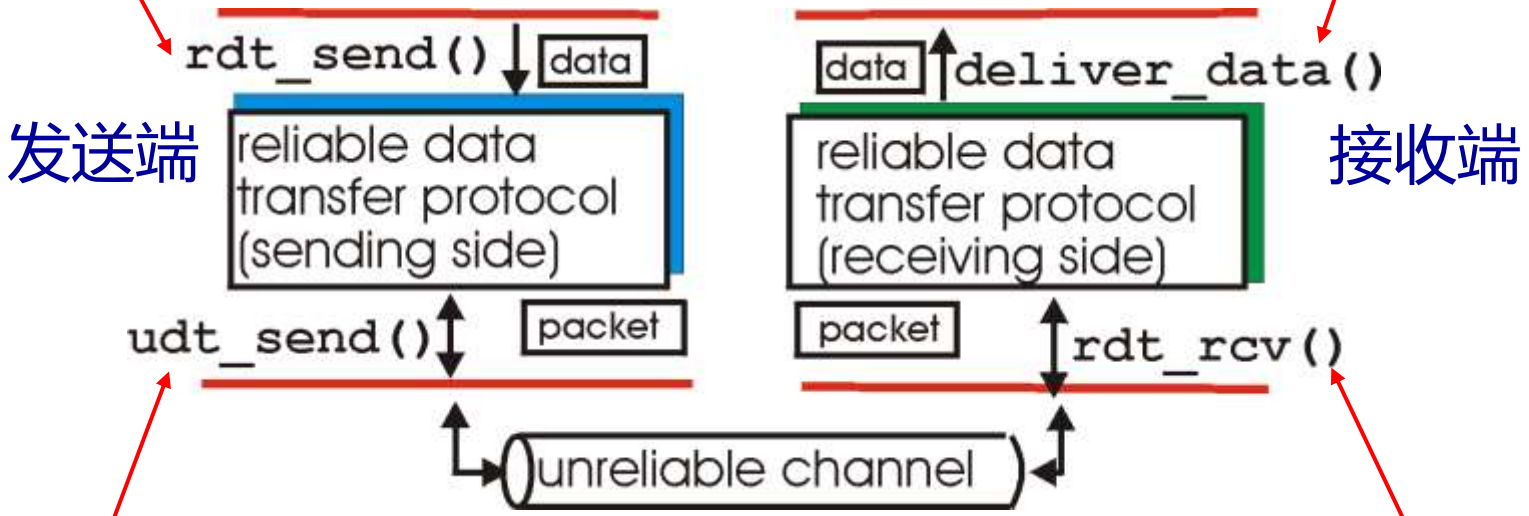


- 可靠传输协议（rdt）的复杂程度取决于不可靠信道的特征（怎样不可靠）

可靠数据传输：出发

rdt_send(): 由上层（应用）调用. 将数据送到接收端的上层

deliver_data(): 由**rdt**调用，将数据提交给上层



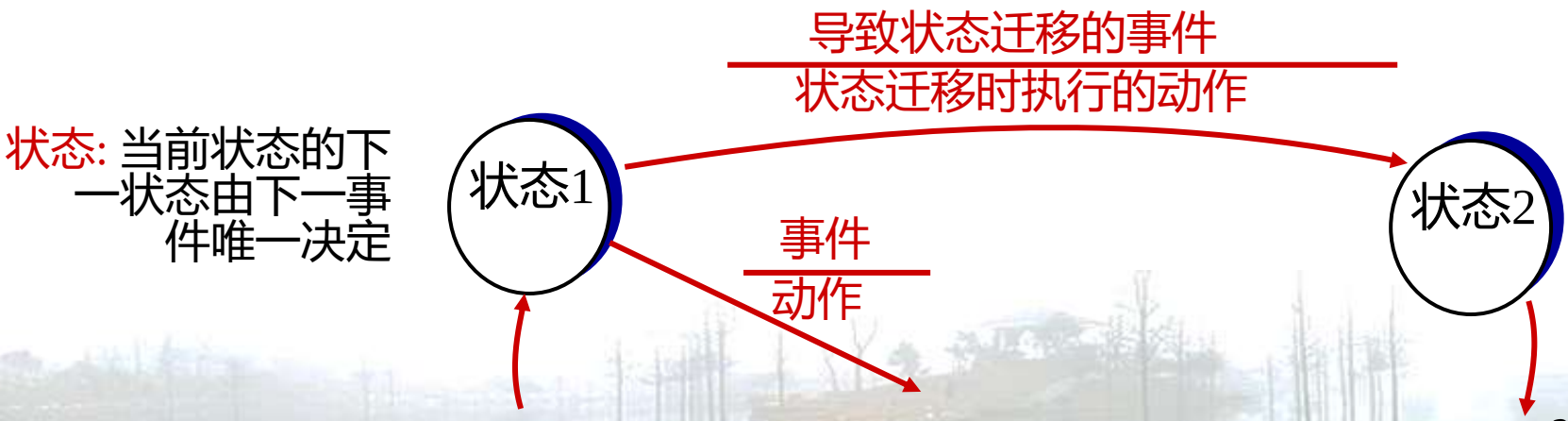
udt_send(): 由**rdt**调用，将数据包由不可靠信道向接收端传输

rdt_rcv(): 当数据包从不可靠信道到达接收端时被调用

可靠数据传输：出发

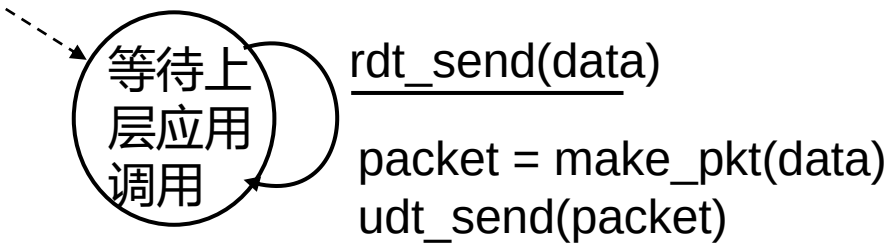
以下：

- 增量方式实现可靠传输协议rdt的发送端和接收端
- 仅考虑数据的单向传输
  - 控制信息可双向传输
- 使用有限状态机(FSM)描述发送端和接收端的状态行为

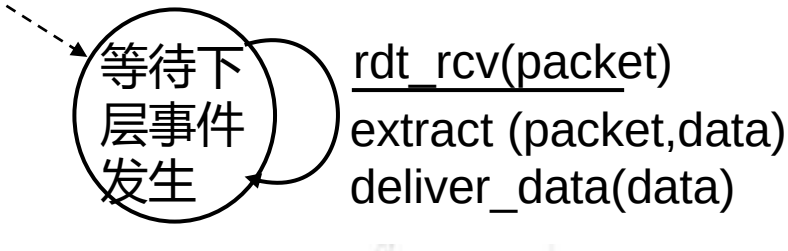


rdt1.0：可靠信道上的可靠传输

- 信道完全可靠
  - 无比特翻转
  - 无丢包
- 发送端和接收端各自用一个FSM描述:
  - 发送端向下层可靠信道发出数据
  - 接收端从下层可靠信道接收数据



发送端



接收端

rdt2.0：有bit翻转差错的信道

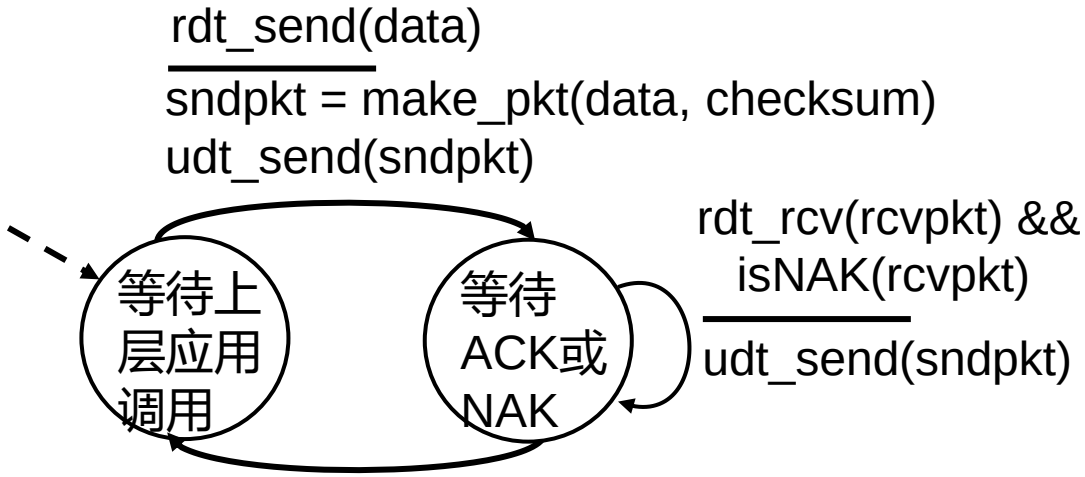
- 下层信道有bit翻转差错
  - 由校验和检测出来
- 问：如何从错误中恢复?

人类对话时，类似情况在如何处理?
•对讲机通话：收到
•当面交流时：刚才你说什么?

rdt2.0：有bit翻转的信道

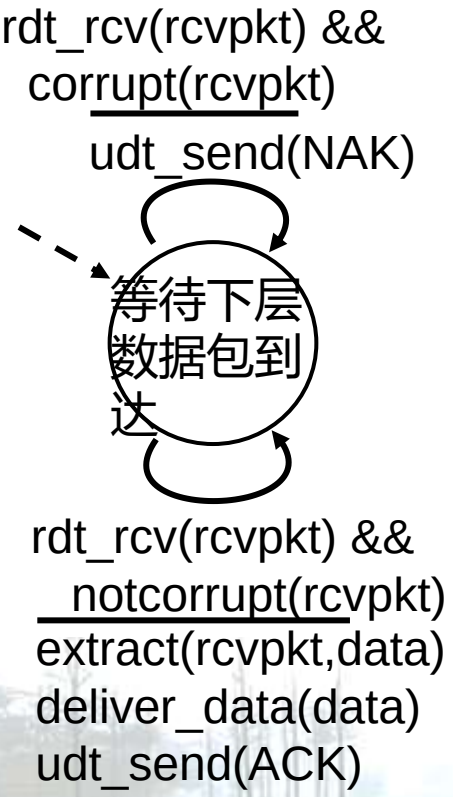
- 下层信道有bit翻转错误
  - 由校验和检测出来
- 问：如何从错误中恢复?
  - 确认(ACKs)：接收端明确告诉发送端某个数据包已正确收到
  - 负确认 (NAKs): 接收端明确告诉发送端某个数据包包含错误
  - 发送端重传NAK指示的数据包
- rdt2.0中的新机制:
  - 检错
  - 反馈：接收端向发送端反馈控制消息(ACK,NAK)

rdt2.0：FSM描述

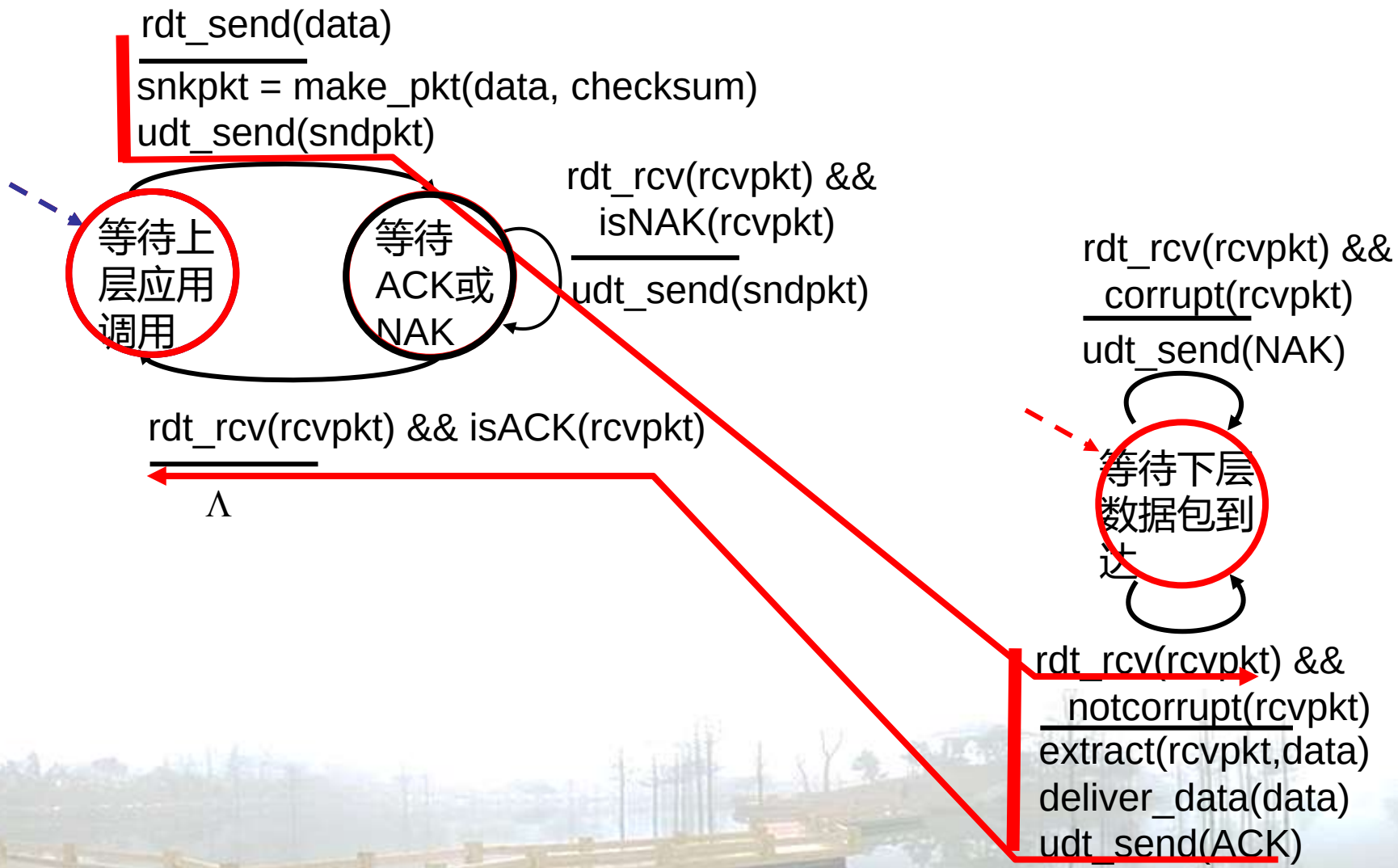


发送端

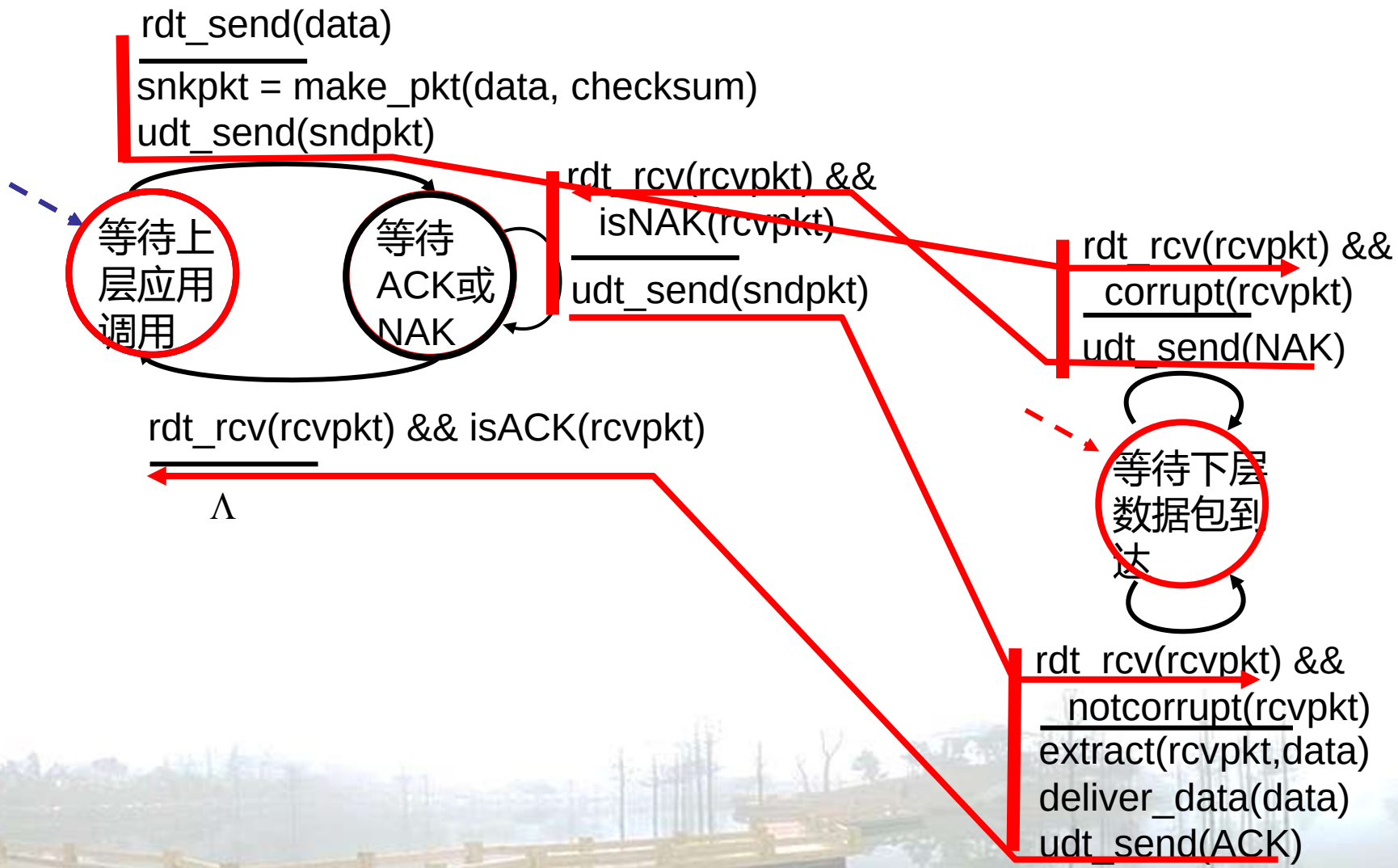
接收端



rdt2.0：当没有错误发生时



rdt2.0：当错误发生时



rdt2.0有致命缺陷

当ACK/NAK发生bit翻转

- 发送端不知道接收端是否正确收到数据包
- 不能简单地重传，可能导致重复发包

停-等协议

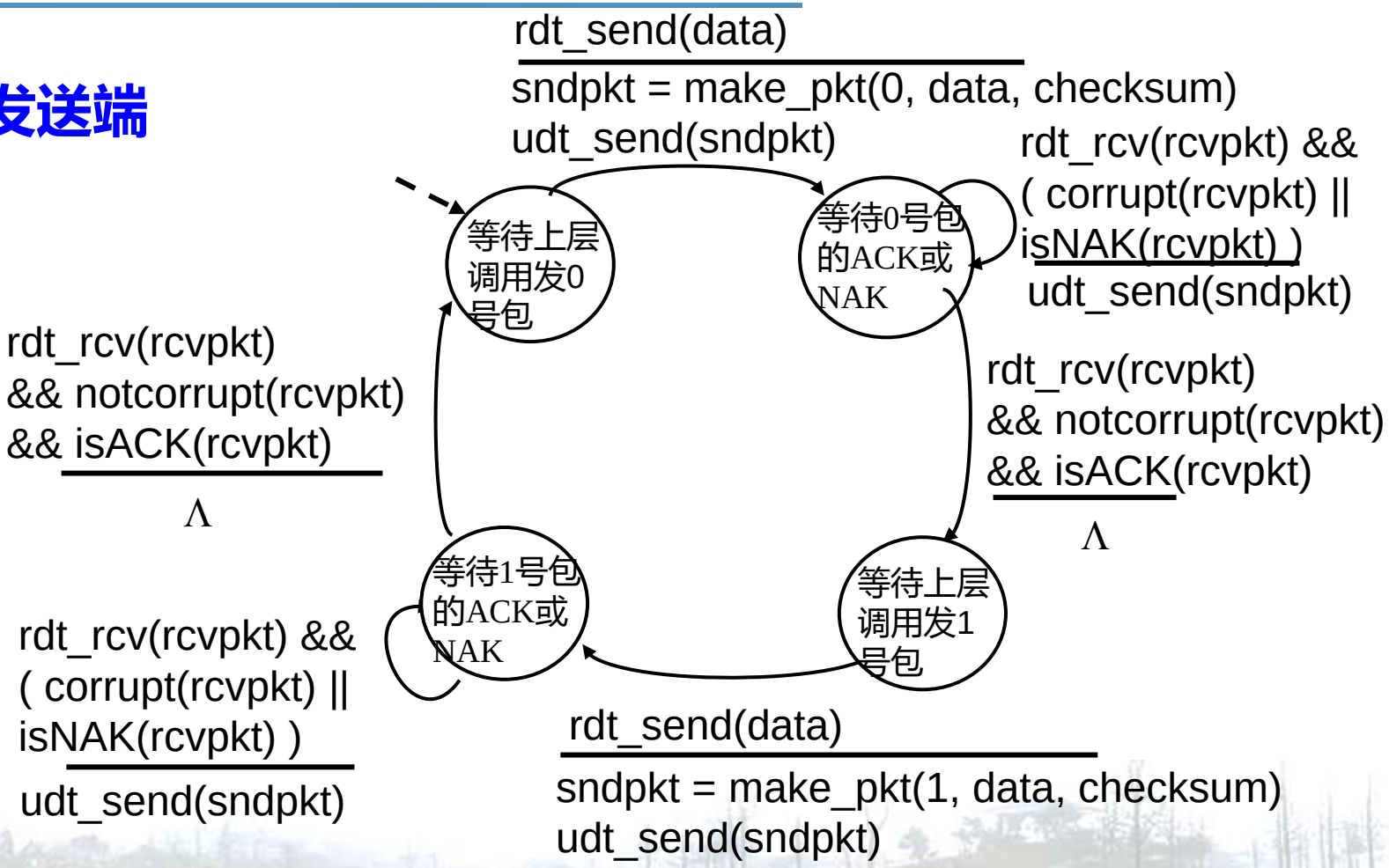
发送端发一个包,然后等待接收端的反馈，此时上层应用不能发新的数据包

处理重复发出的包:

- 当ACK/NAK错误时发送端重传数据包
- 发送端为每个数据包添加一个序列号
- 接收端忽略丢弃重复收到的数据包（相同序列号的包已收到过了）

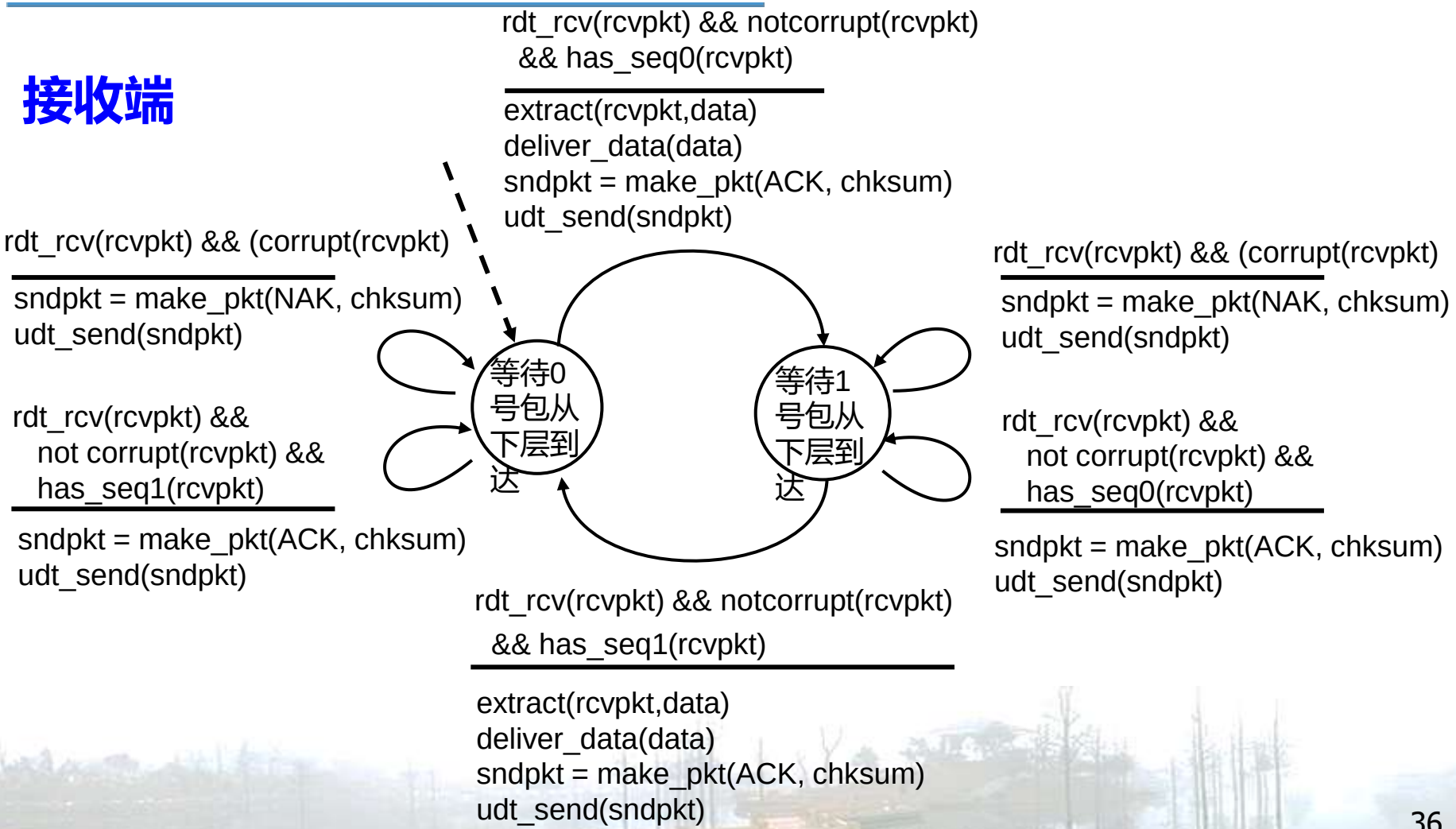
rdt2.1：处理差错的ACK/NAK

发送端



rdt2.1：处理差错的ACK/NAK

接收端



rdt2.1：讨论

发送端:

- 数据包添加序列号
- 两个序列号（0和1）足够，为什么?
- 检查ACK/NAK是否存在错误
- 两倍的状态数量
  - 对0和1号数据包，各有一套状态

接收端:

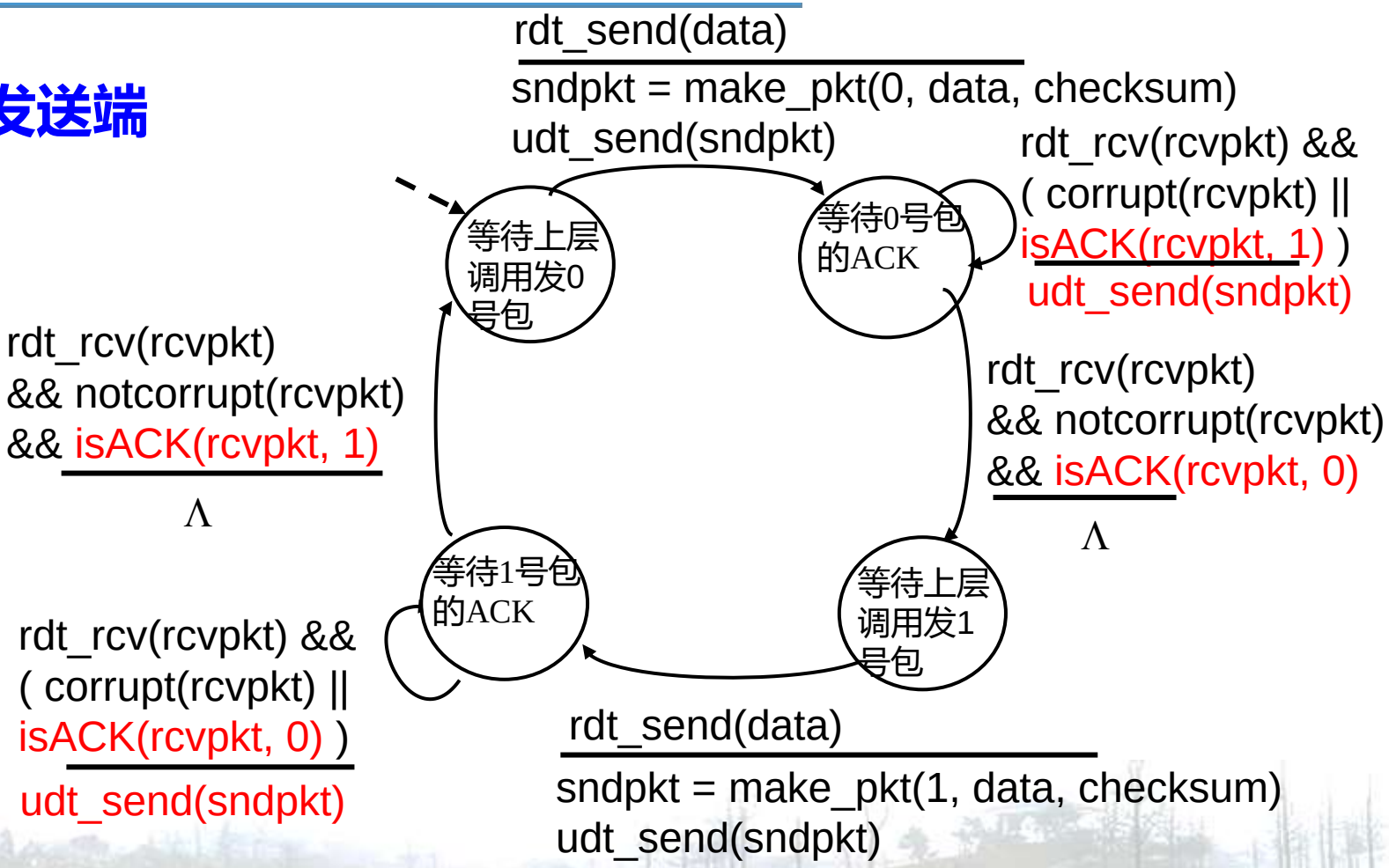
- 检查收到的包是否重复
  - 当前状态指示下一个期望收到的包是0号还是1号包
- 注意：接收端并不知道发出的ACK/NAK是否被正确接收

rdt2.2：不使用NAK的协议

- 实现和rdt2.1相同功能，但不使用NAK
- 当收到包含错误的数据包，接收端重复发送上个正确接收的数据包的ACK
  - ACK携带所确认数据包的序列号
- 发送端收到重复的ACK，采取和收到NAK时一样的动作：重传当前的包

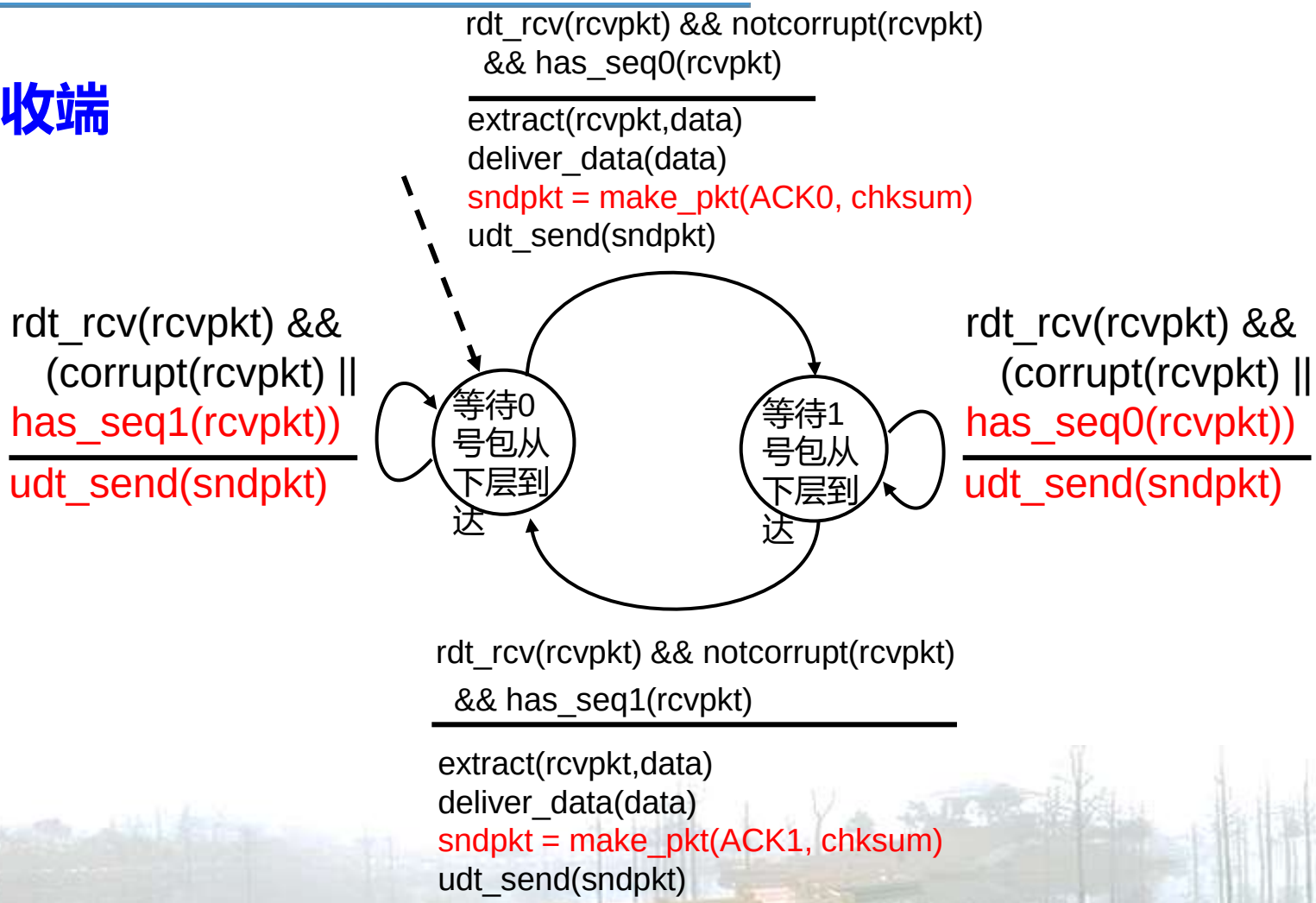
rdt2.2：发送端

发送端



rdt2.2：接收端

接收端



rdt3.0：信道包含差错和丢包

新的假设:

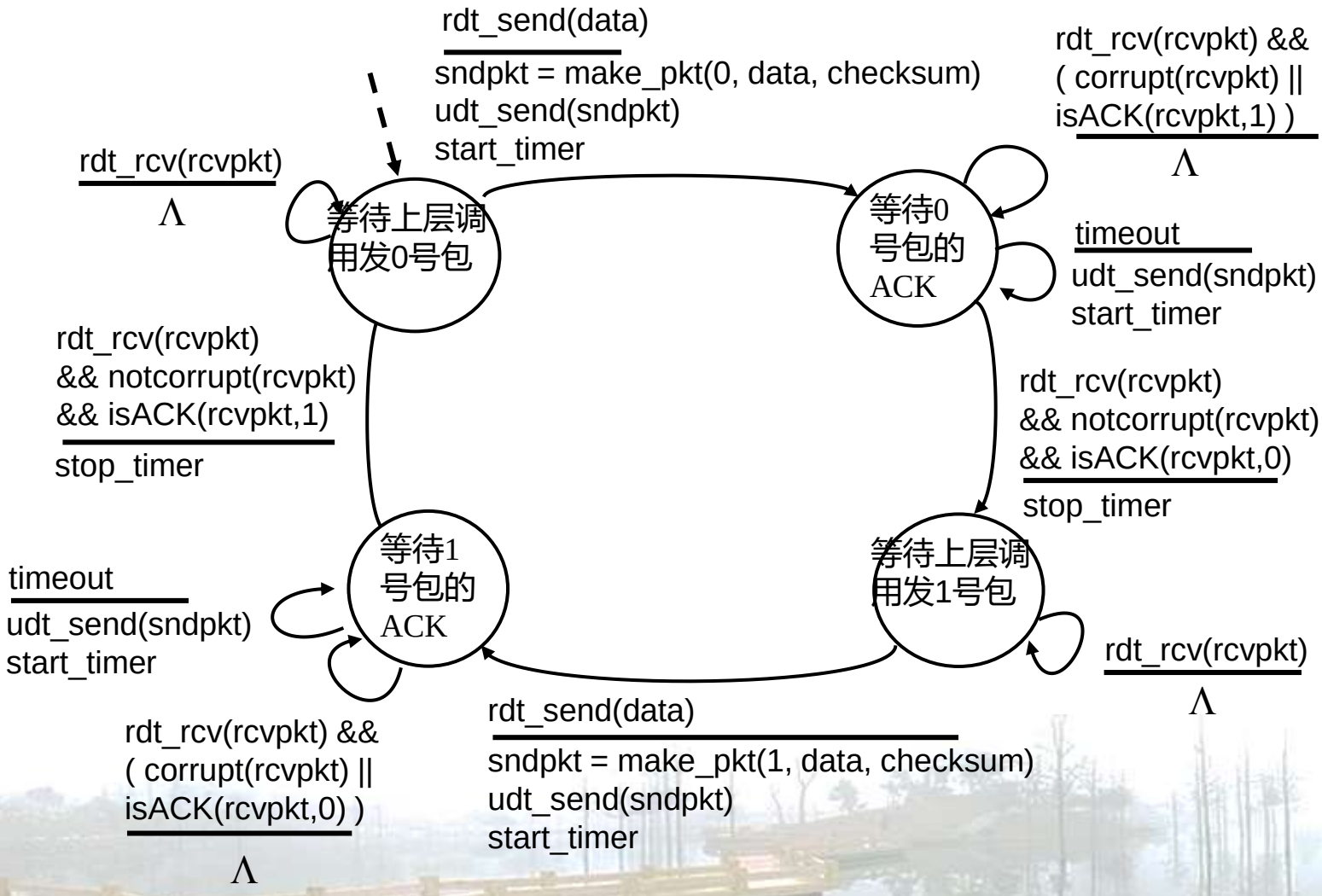
下层信道丢包 (数据包和ACK)

- 检验和、序列号、ACK机制有一些作用... 但不够

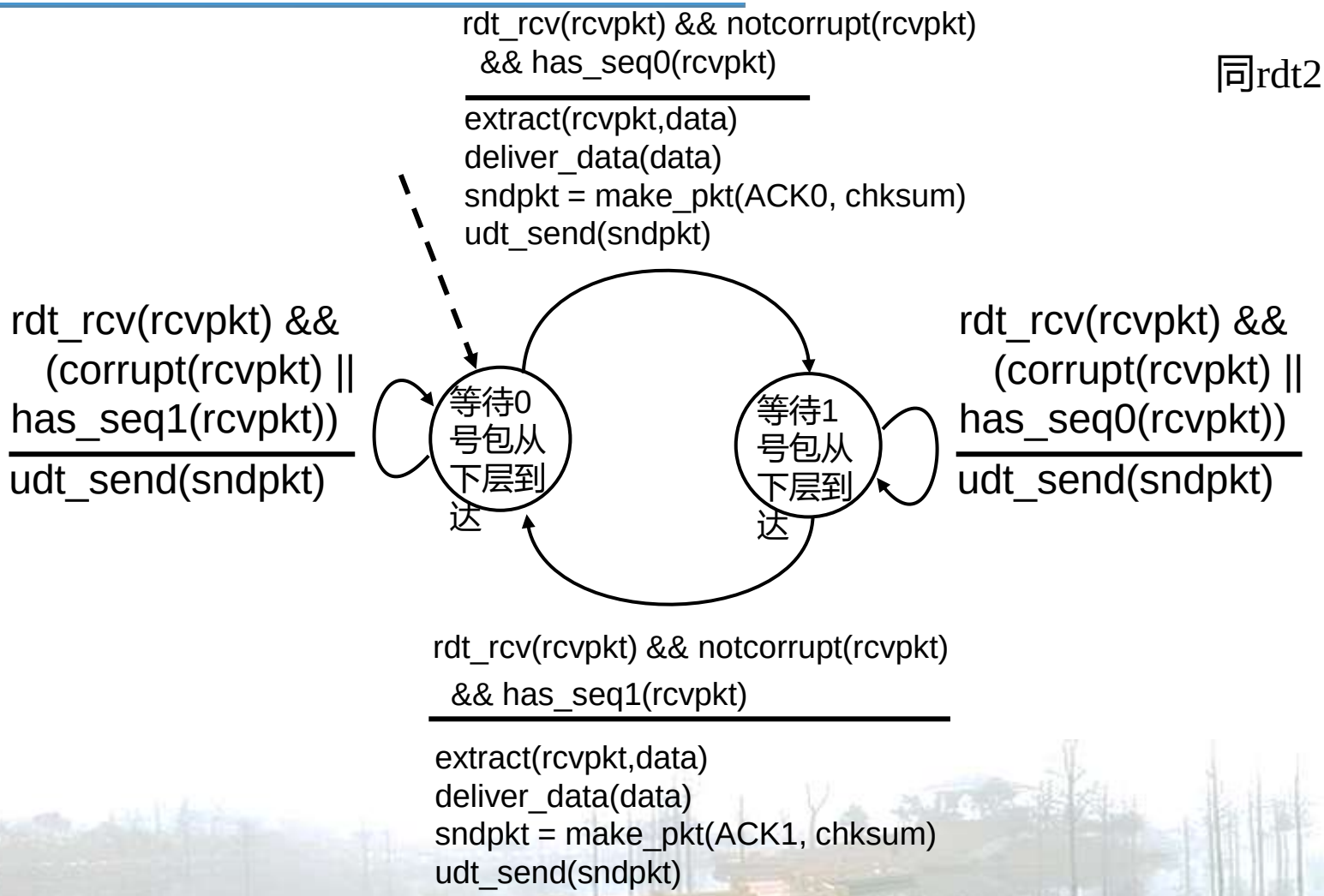
应对措施: 发送端等ACK一段“合理”的时间

- 如果超时未收到ACK，重传数据包
- 如果数据包只是延迟到达，并未丢失:
  - 接收端收到重复的数据包，可以用序列号区分
  - 接收端必须在ACK中明确确认的数据包的序列号
- 发送端需要倒计时器

rdt3.0：发送端

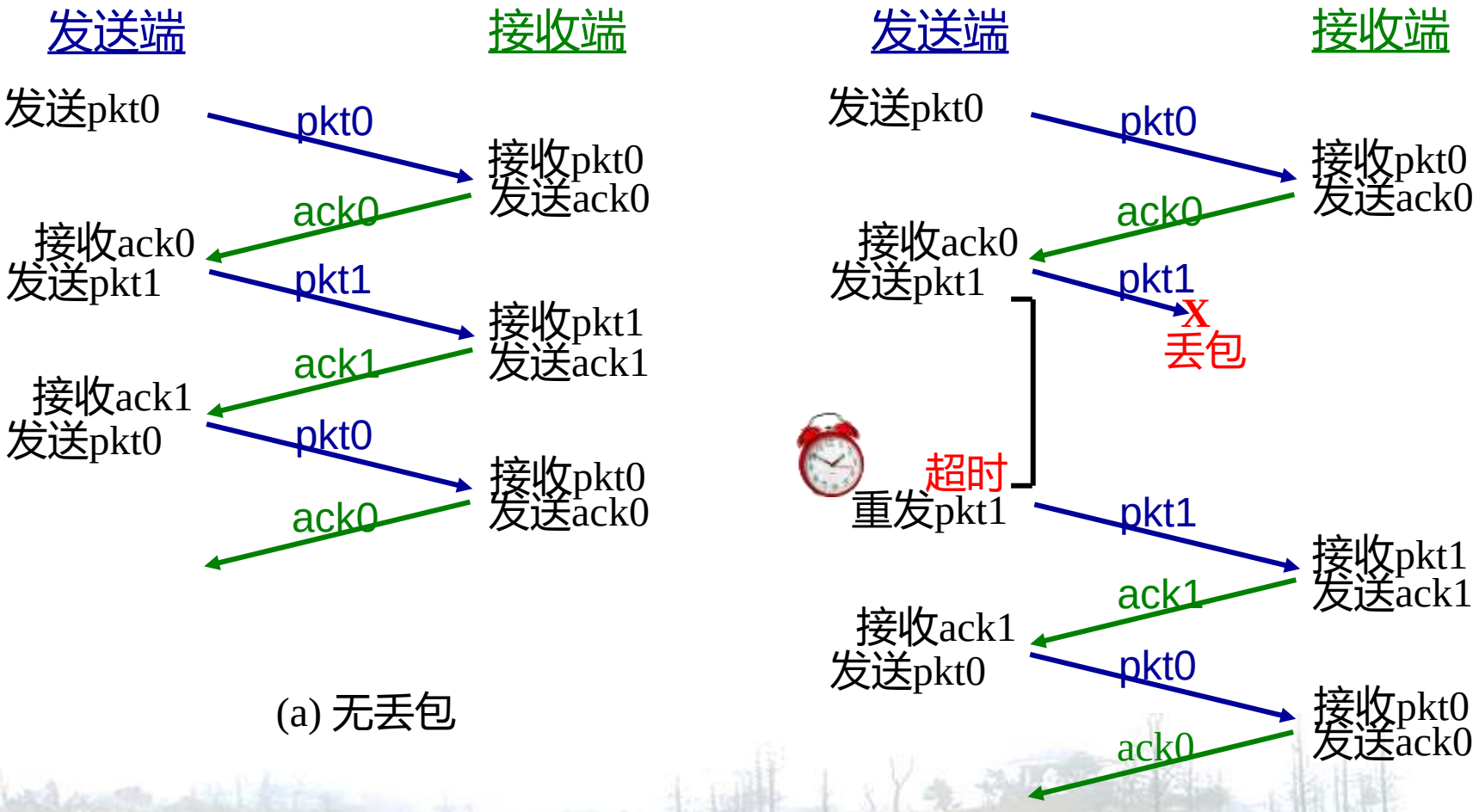

rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
等待上层调用发0号包
等待0号包的ACK
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
stop_timer
等待1号包的ACK
等待上层调用发1号包
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：接收端

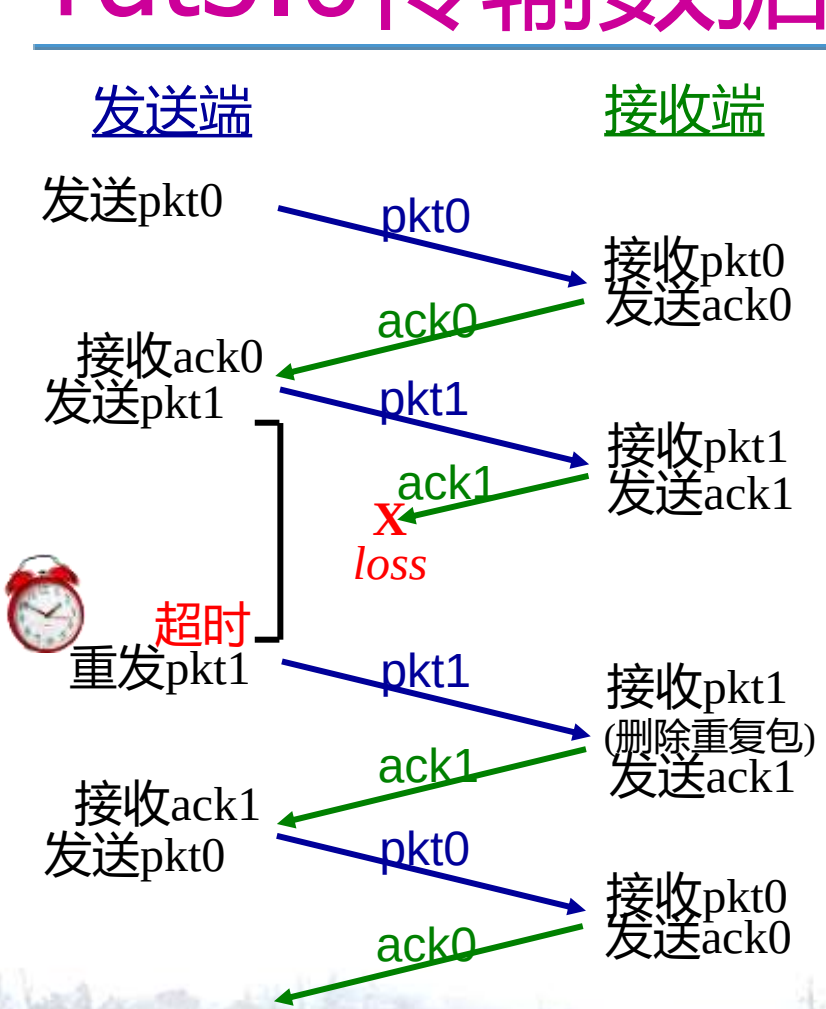


同rdt2.2一样

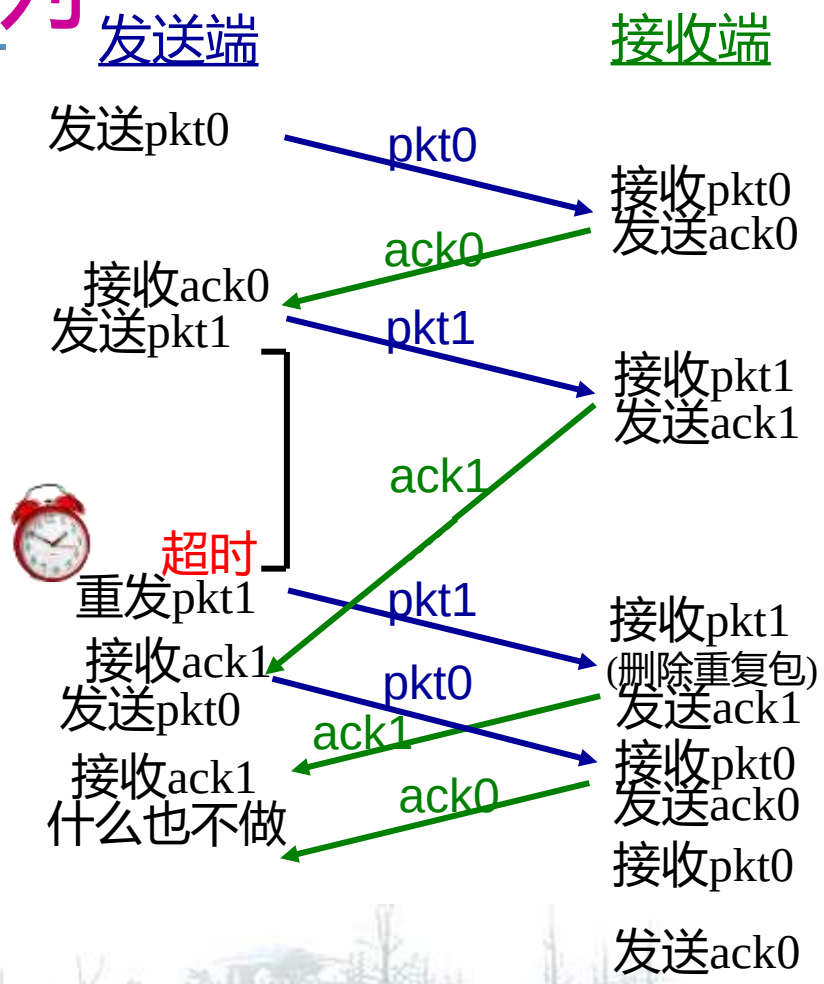
rdt3.0传输数据举例



rdt3.0传输数据举例



(c)丢ACK



(d) ACK到达前超时

rdt3.0性能

- rdt3.0功能正确，但是停-等协议性能很差
- 例如：1 Gbps的链路，15毫秒的传播时延，每个包8000bit:

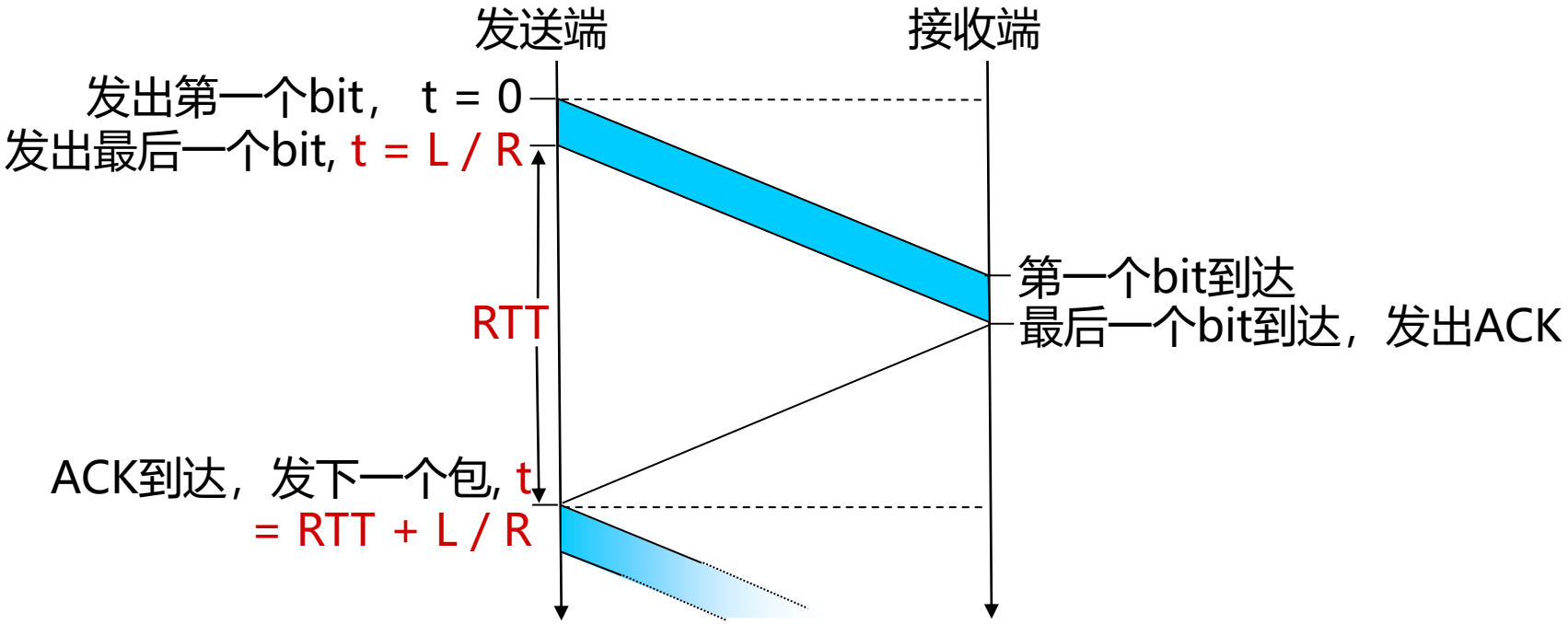
$$\text{传输时延} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ 微秒}$$

  - U_{sender}: 利用率–发包时间在完成整个数据包传输的时间占比

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

    - 若RTT=30毫秒，每30毫秒可完成传输1个包: 在1Gbps的链路上能够实现33kB/秒的速率
- 网络协议限制了物理资源的利用率!

rdt3.0：停-等操作

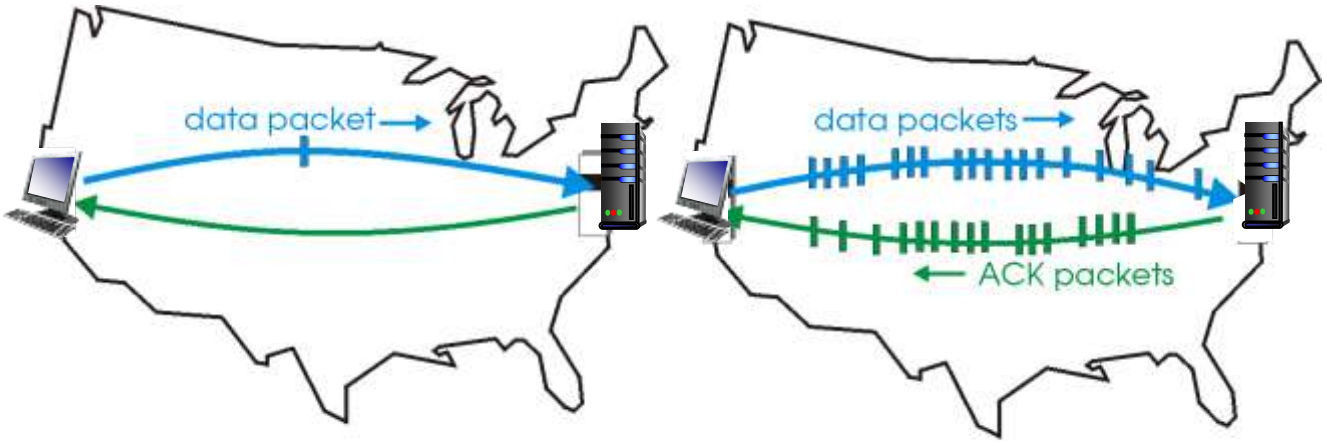


$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

流水线协议

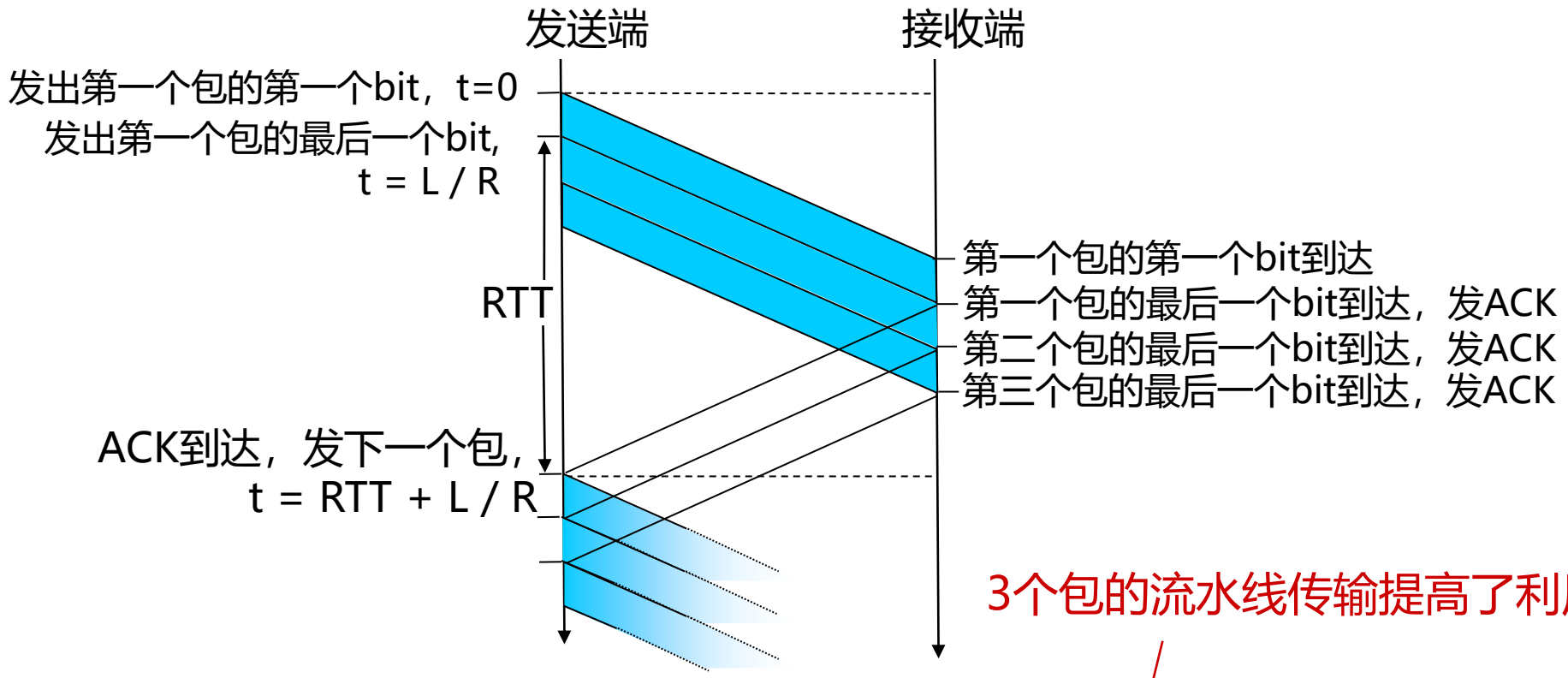
流水线传输: 发送端允许多个数据包 “在路上” “未确认”

- 增加序列号的空间
- 在发送端/接收端暂存数据包



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

流水线协议：提高利用率



3个包的流水线传输提高了利用率

$$U_{sender} = \frac{3L/R}{RTT + L/R} = \frac{.0024}{30.008} = 0.00081$$

流水线协议概览

回退N:

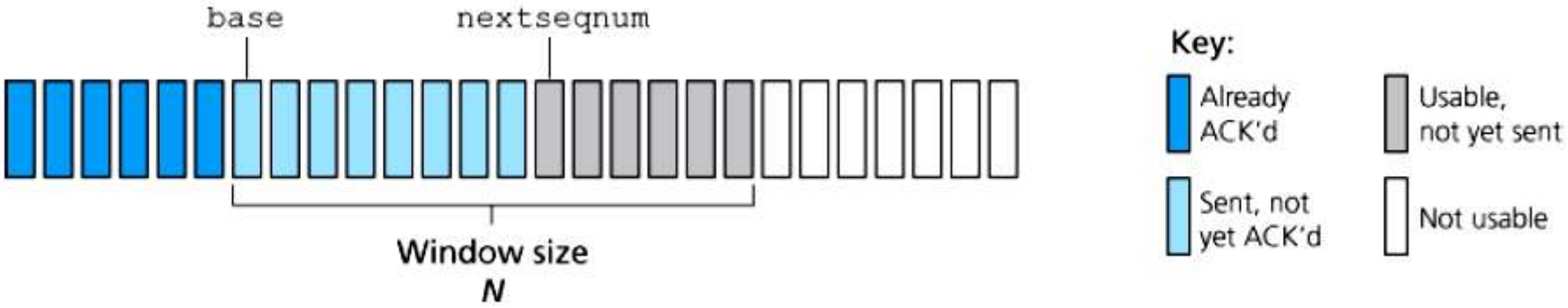
- 发送端可以容忍N个包在路上未确认
- 接收端发送累积ACK
  - 确认ACK序列号之前所有的包
  - 如果收到的数据包序列号不连续（有gap），不确认这部分数据包
- 对最早发出（序列号最小）的未确认数据包，发送端设置一个定时器
  - 当定时器超时，重传所有未确认的数据包

选择性重传:

- 发送端可以容忍N个包在路上未确认
- 接收端对每个收到的包单独发出ACK
  - 仅确认一个包
- 发送端为每一个未确认的包设置一个定时器
  - 当定时器超时，重传该数据包

回退N：发送端

- 包头部携带k-bit的序列号
- 大小为N的窗口，允许N个已发出未确认的数据包



- ACK(n): 确认所有序列号不大于n的数据包 - “累积ACK”
  - 可能收到重复的ACK
- 对最早发出（序列号最小）的未确认数据包，设置定时器
- timeout(n)：重传窗口中所有未确认的序列号不小于n的数据包

回退N：发送端FSM描述

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

等待

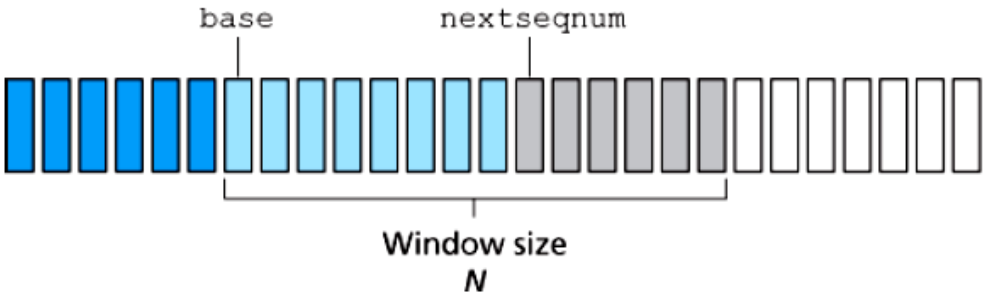
rdt_rcv(rcvpkt)
&& corrupt(rcvpkt)

timeout

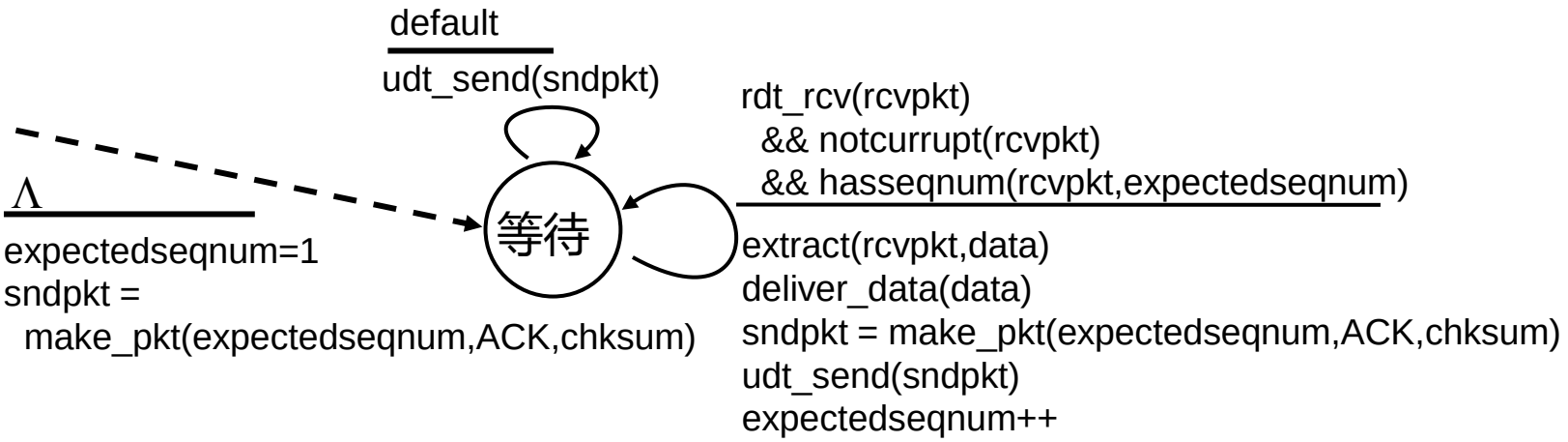
```
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

```
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
   stop_timer
  else
   start_timer
```



回退N：接收端FSM描述

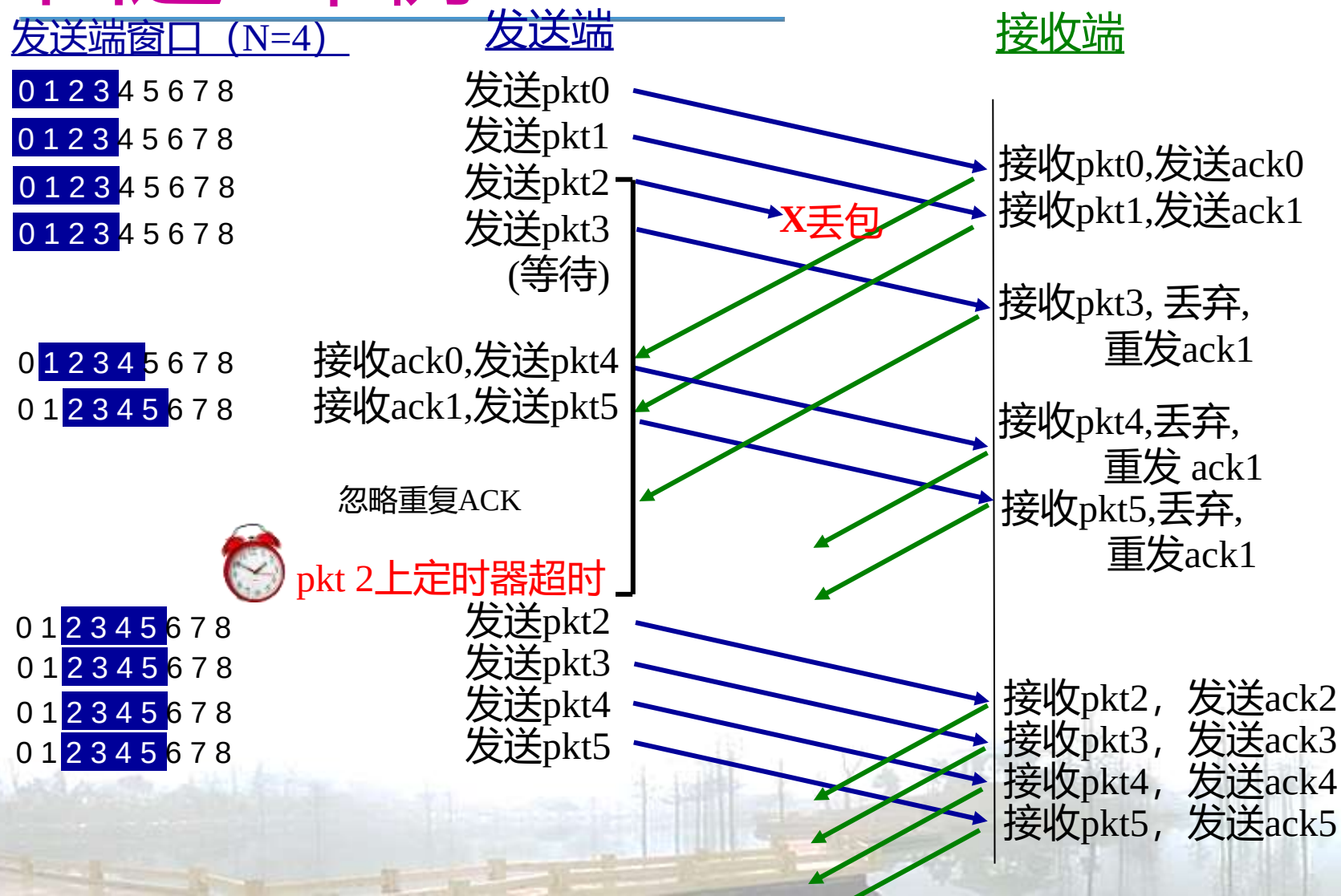


总是确认正确接收的顺序到达的最大序列号的数据包

- 可能产生重复的确认ACK
- 需要记住**expectedseqnum**（下一个希望收到的包的序列号）

- 乱序到达的数据包:
  - 丢弃(不暂存)
  - 重复确认当前收到的顺序到达的最大序列号的数据包

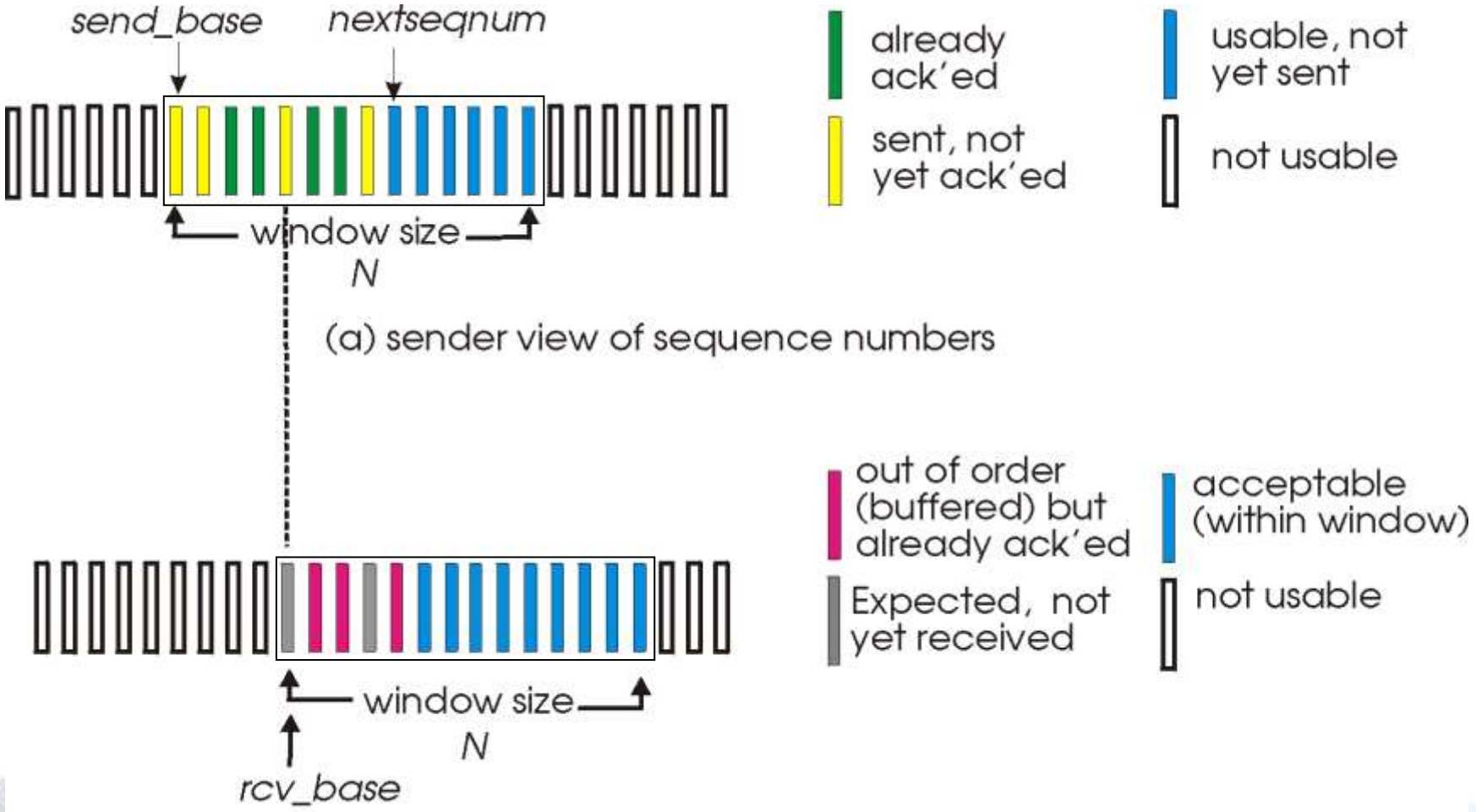
回退N举例


发送端窗口（N=4）
发送端
接收端
0 1 2 3 4 5 6 7 8
发送pkt0
发送pkt1
发送pkt2
发送pkt3
(等待)
X丢包
接收pkt0,发送ack0
接收pkt1,发送ack1
接收pkt3, 丢弃,
重发ack1
0 1 2 3 4 5 6 7 8
接收ack0,发送pkt4
0 1 2 3 4 5 6 7 8
接收ack1,发送pkt5
接收pkt4,丢弃,
重发 ack1
忽略重复ACK
接收pkt5,丢弃,
重发ack1
pkt 2上定时器超时
0 1 2 3 4 5 6 7 8
发送pkt2
发送pkt3
发送pkt4
发送pkt5
接收pkt2，发送ack2
接收pkt3，发送ack3
接收pkt4，发送ack4
接收pkt5，发送ack5

选择性重传

- 接收端单独确认每个正确接收的数据包
  - 暂存收到的数据包，排序提交给上层应用
- 发送端仅重传没有收到确认的超时数据包
  - 为每个尚未收到确认的数据包设置定时器
- 发送端窗口
  - N个连续的序列号
  - 用于限制已发出但尚未确认的数据包的个数

选择性重传的发送端接收端窗口



选择性重传

发送端

上层应用有数据待发:

- 发送窗口未满，发送数据包

timeout(n):

- 重传序列号为n的数据包，重启定时器

在[sendbase,sendbase+N]范围内收到ACK(n):

- 标识序列号为n的数据包为已确认
- 如果n是最小的未确认序列号，前移窗口至下一个未确认序列号

接收端

收到[rcvbase, rcvbase+N-1]范围内序列号为n的数据包

- 发送ACK(n)
- 数据包乱序：暂存
- 数据包顺序: 将该数据包和暂存的、和这个数据包形成连续数据的其它数据包提交上层应用，前移窗口至下一个未收到确认的数据包

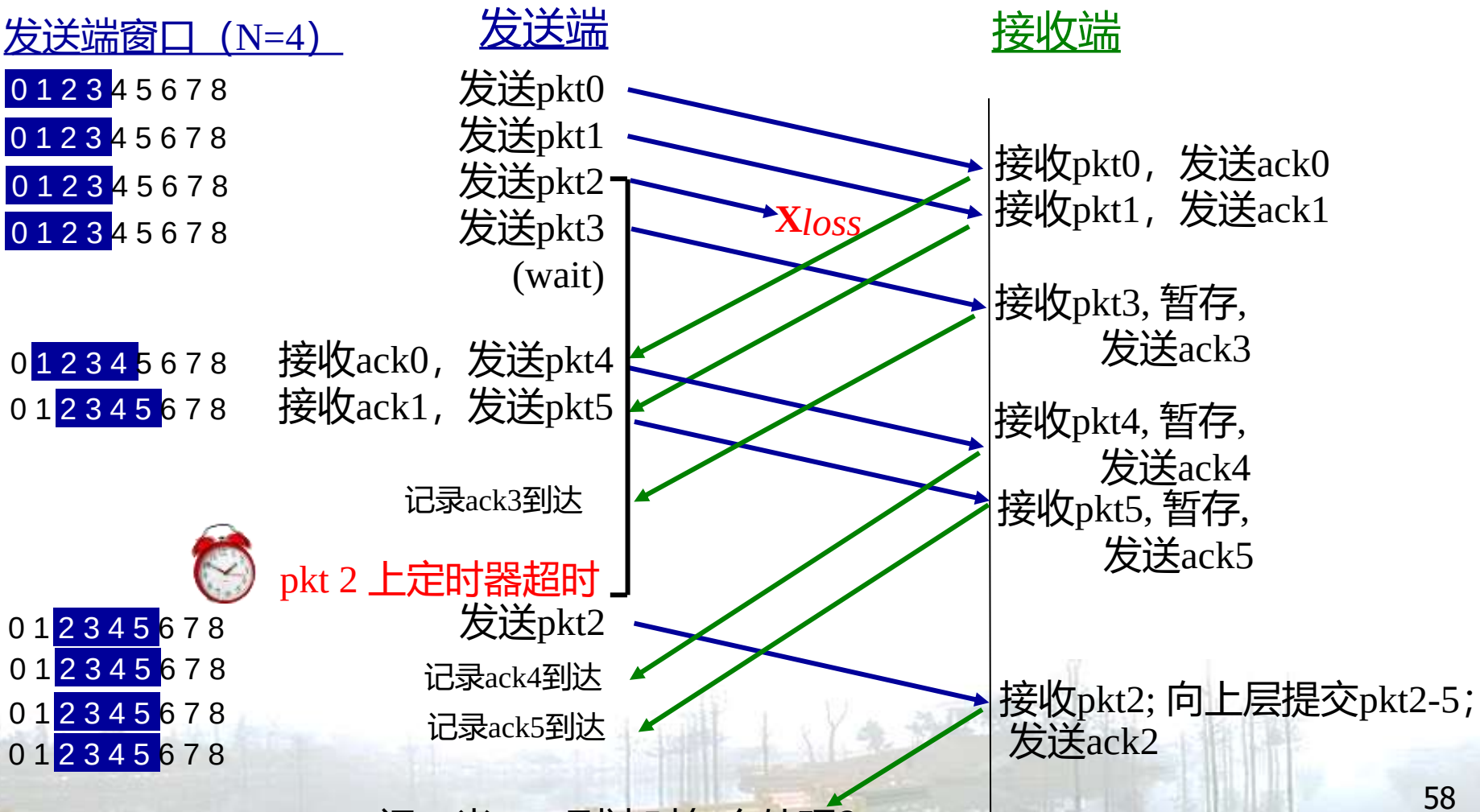
在[rcvbase-N,rcvbase-1]范围内收序列号为的数据包n

- ACK(n)，使发送端窗口前移

其它:

- 忽略

选择性重传举例

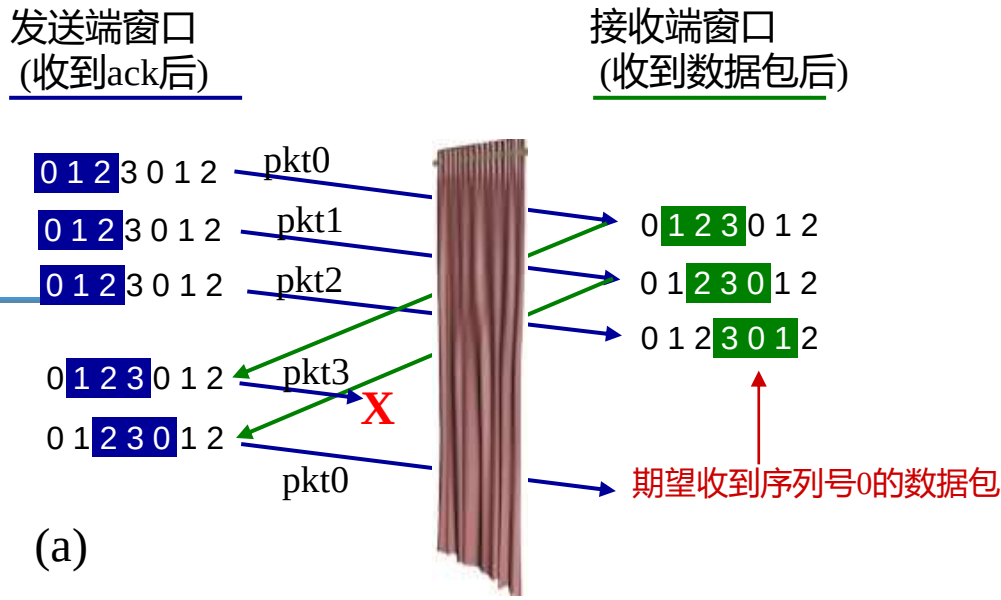

发送端窗口（N=4）
发送端
接收端
0 1 2 3 4 5 6 7 8
发送pkt0
发送pkt1
发送pkt2
发送pkt3
(wait)
接收pkt0，发送ack0
接收pkt1，发送ack1
X loss
接收pkt3, 暂存,
发送ack3
接收ack0，发送pkt4
接收ack1，发送pkt5
接收pkt4, 暂存,
发送ack4
接收pkt5, 暂存,
发送ack5
记录ack3到达
pkt 2 上定时器超时
发送pkt2
记录ack4到达
记录ack5到达
接收pkt2; 向上层提交pkt2-5;
发送ack2
问：当ack2到达时怎么处理?

选择性重传的两难困境

举例:

- 序列号: 0、1、2、3
- 窗口大小=3
- 接收端无法分辨两种情况!
- 无法区分重传数据包和新数据包

Q: 怎样设置序列号空间和窗口大小，避免这类情况?



接收端无法查看发送端的窗口!
两个例子里，接收端收到的数据包序列号一样
存在重大问题!

窗口大小不应超过序列号空间的一半

讨论

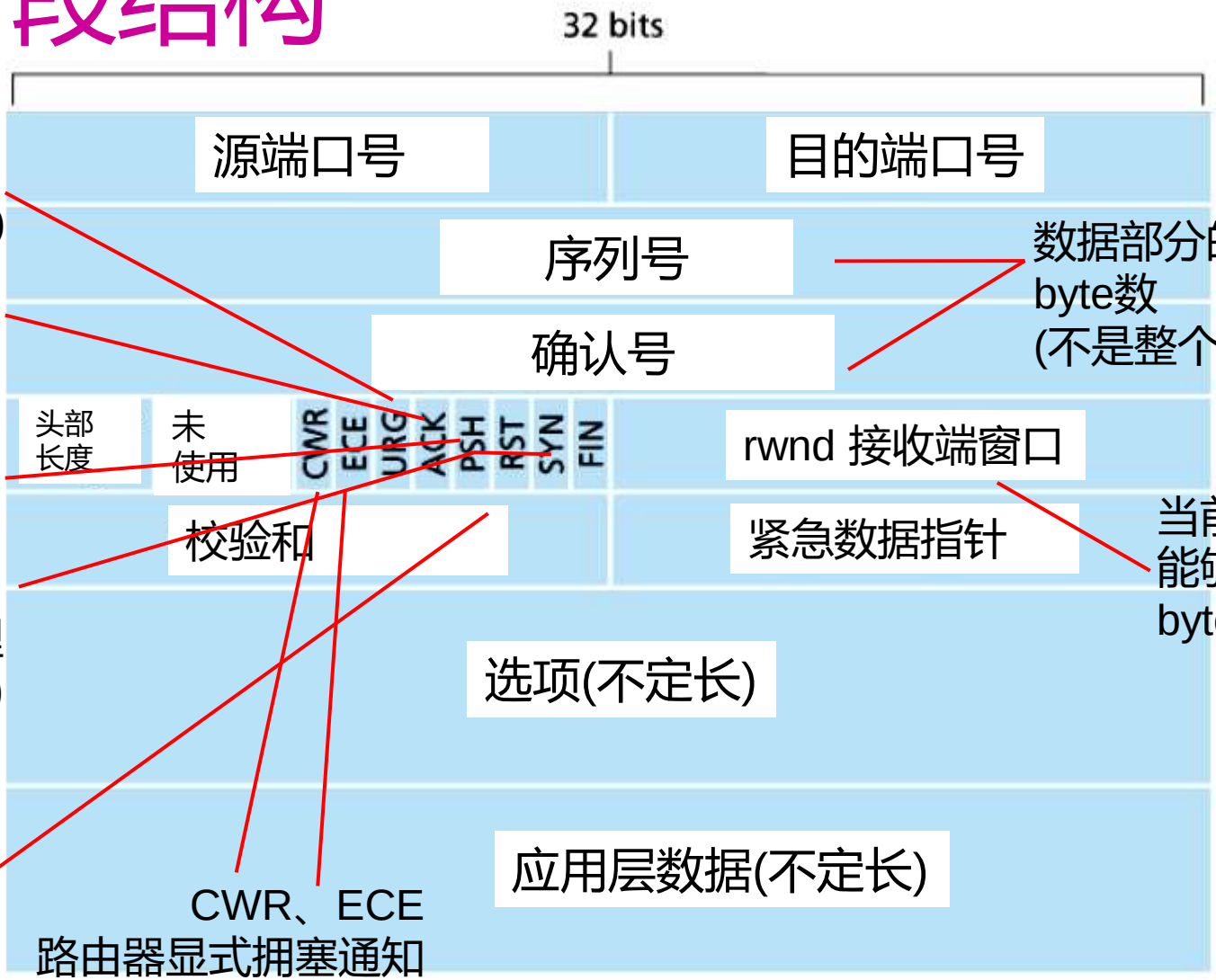
- 回退N
  - 优点：发送端仅维持一个计时器，接收端不用缓存乱序到达的数据包
  - 缺点：浪费带宽
- 选择性重传
  - 优点：不浪费带宽
  - 缺点：发送端维持多个计时器，接收端需缓存乱序到达的数据包

目录

TCP：概览RFCs: 793,1122,1323, 2018, 2581

- 端到端:
  - 一个发送端、一个接收端
- 可靠、顺序的比特流:
  - 比特流没有边界
- 流水线传输:
  - 通过拥塞控制和流控制调整窗口大小
- 全双工:
  - 一个连接上双向数据传输
  - MSS: maximum segment size，最大帧的大小
  - 1500-20-20=1460
- 面向连接:
  - 通过握手在数据传输前初始化发送端和接收端的协议状态
- 流控制:
  - 发送端速率不会压倒接收端速率

TCP分段结构


32 bits
源端口号
目的端口号
序列号
确认号
头部长度
未使用
CWR
ECE
URG
ACK
PSH
RST
SYN
FIN
rwnd 接收端窗口
校验和
紧急数据指针
选项(不定长)
应用层数据(不定长)
URG: 紧急数据(未启用)
ACK: 标识确认号是否有效
PSH: 推送数据(未启用)
RST, SYN, FIN: 连接管理(建立、撤销连接)
校验和(与UDP一样)
CWR、ECE 路由器显式拥塞通知
数据部分的byte数(不是整个分段的)
当前接收端能够接收的byte数

TCP序列号和确认

序列号:

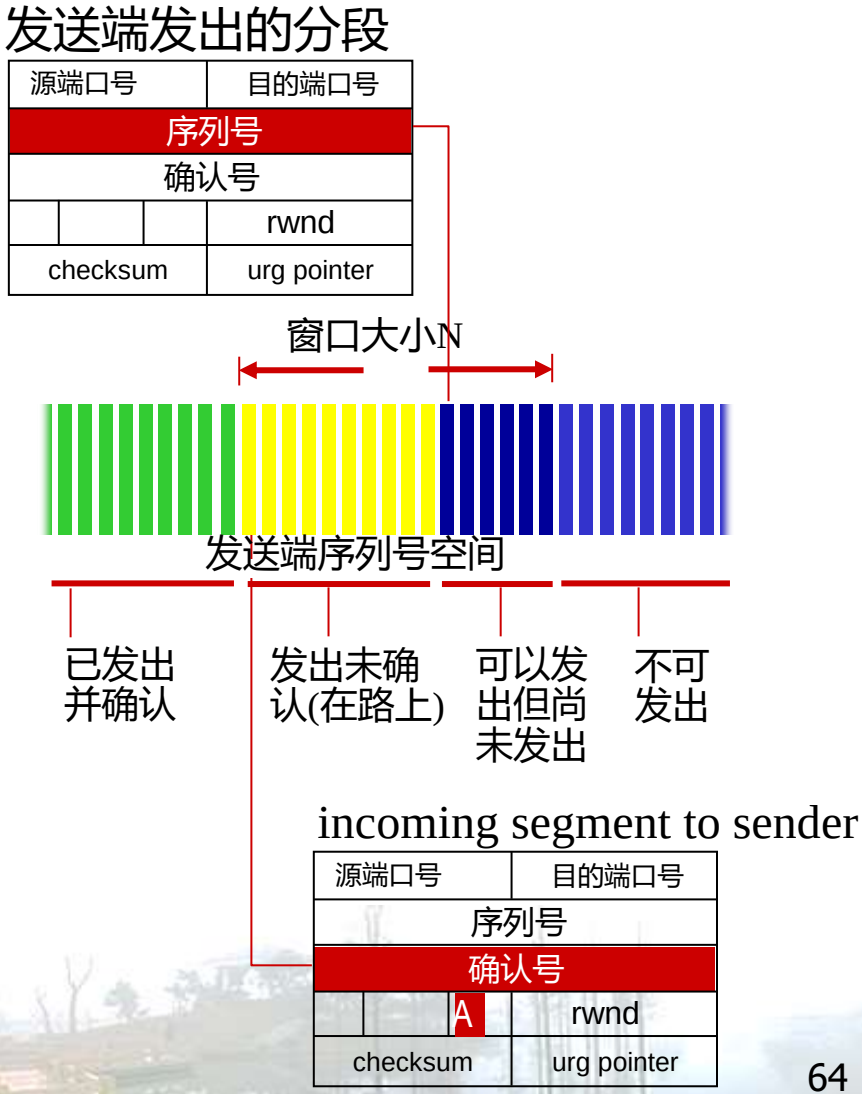
- 分段中第一个byte在整个byte流的次序编号

确认号:

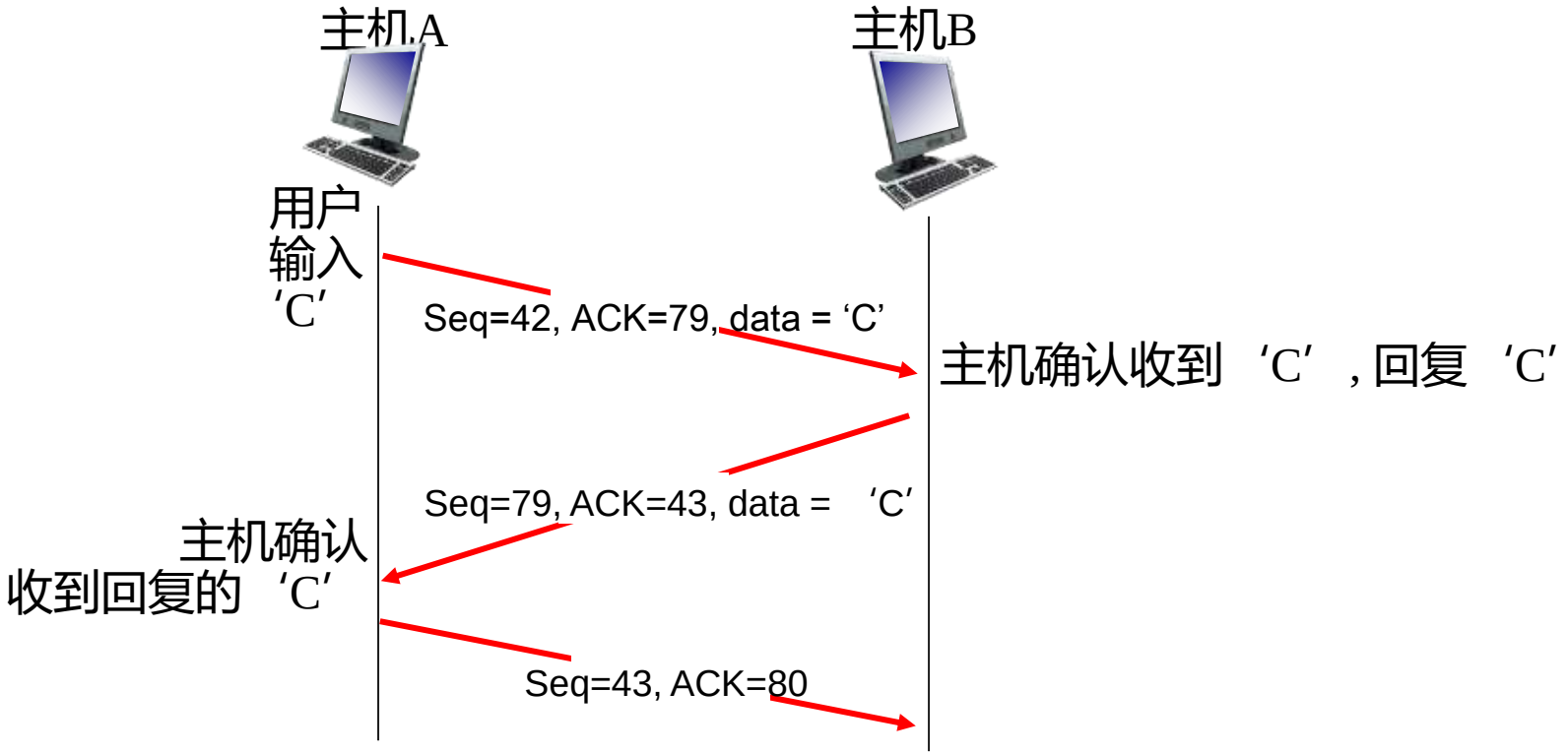
- 期望收到的下一个byte的序列号
- 累积确认

问: 如何处理乱序到达的分段?

- 答: 标准没有规定，取决于具体实现
- 多数时候，缓存并等待缺失的数据



TCP序列号和确认号



telnet场景

TCP往返时延和超时

问: 怎样设置TCP的超时?

- 比RTT长
- 但是RTT不断变化

- 设置过短: 分段到达前超时，导致不必要的重传
- 设置过长: 分段丢失时反应迟缓

问: 怎样测算RTT?

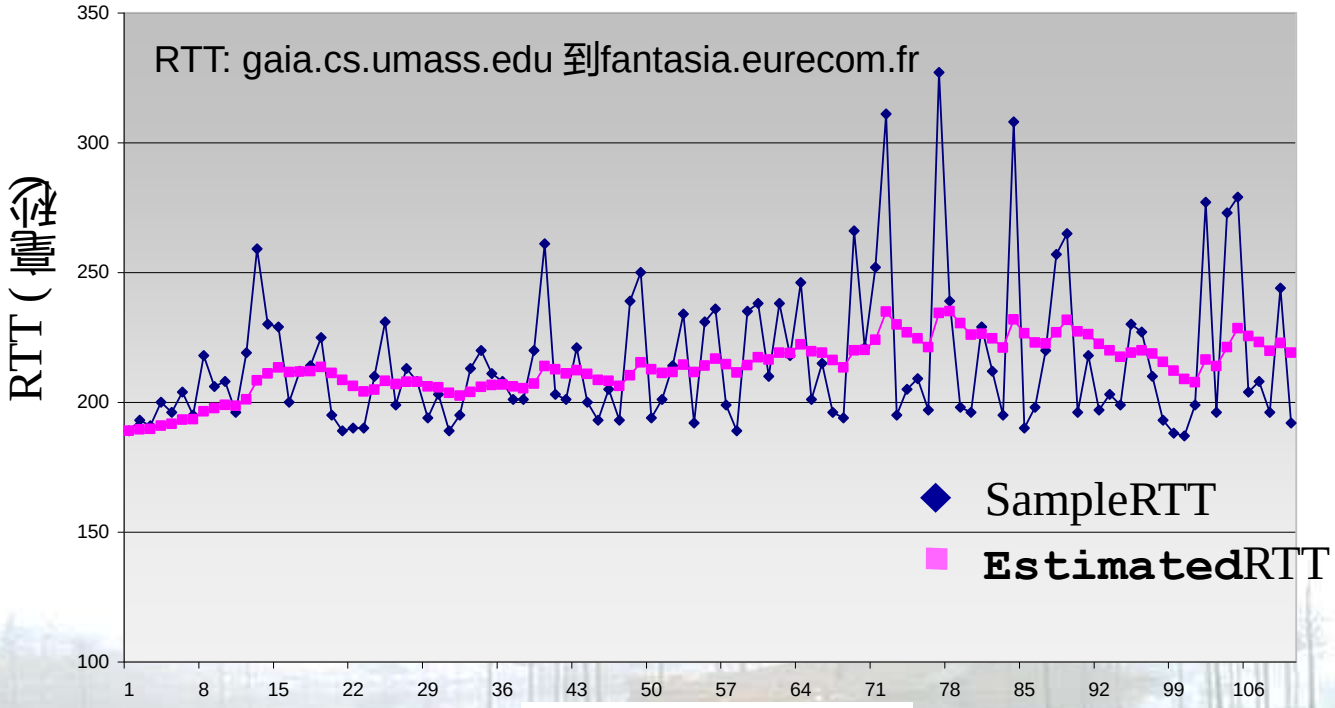
- SampleRTT: 每个分段发出到收到确认的这段时间
  - 忽略重传
- **SampleRTT**不断变化
  - 计算最近若干次测量获取的**SampleRTT**的均值

TCP往返时延和超时

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

- 指数加权滑动平均（EWMA）
- 单个采样的权重随时间指数递减
- 一般而言， α = 0.125

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



TCP往返时延和超时

- 超时：**EstimatedRTT** 加上“安全边界”
  - **EstimatedRTT** 具有大的方差-> 需要大的安全边界
- 从EstimatedRTT计算SampleRTT方差：

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

(一般而言，$\beta = 0.25$)

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

estimated RTT

“安全边界”

目录

TCP可靠数据传输

- TCP在IP不可靠传输的基础上创建可靠数据传输服务
  - 流水线传输
  - 累积确认
  - 单个重传定时器
- 以下事件导致重传:
  - 超时
  - 收到重复的确认

考虑一个简化的TCP发送端:

- 忽略重复确认
- 忽略流控制、拥塞控制

TCP发送端事件

从上层应用获得数据:

- 创建分段，分配序列号
- 序列号是分段携带的一个byte在byte流的序号
- 如果定时器未启动，启动定时器
  - 为最早的未确认分段设置定时器
  - 定时器超时:设置**TimeOutInterval**

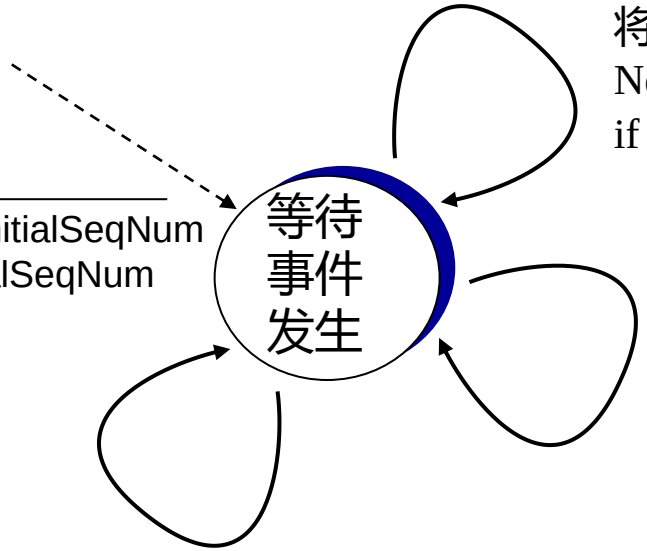
超时:

- 重传引发超时的分段
- 重启定时器

收到ack:

- 如果ack确认之前未确认的分段
  - 将分段标记为已确认
  - 如果仍有未确认分段，启动定时器

TCP发送端FSM（简化）



Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

从应用层获得数据

创建分段，序列号设置为 NextSeqNum
将分段交给IP协议栈(即，发送)
NextSeqNum = NextSeqNum + length(data)
if (当前没有定时器运行)
　启动定时器

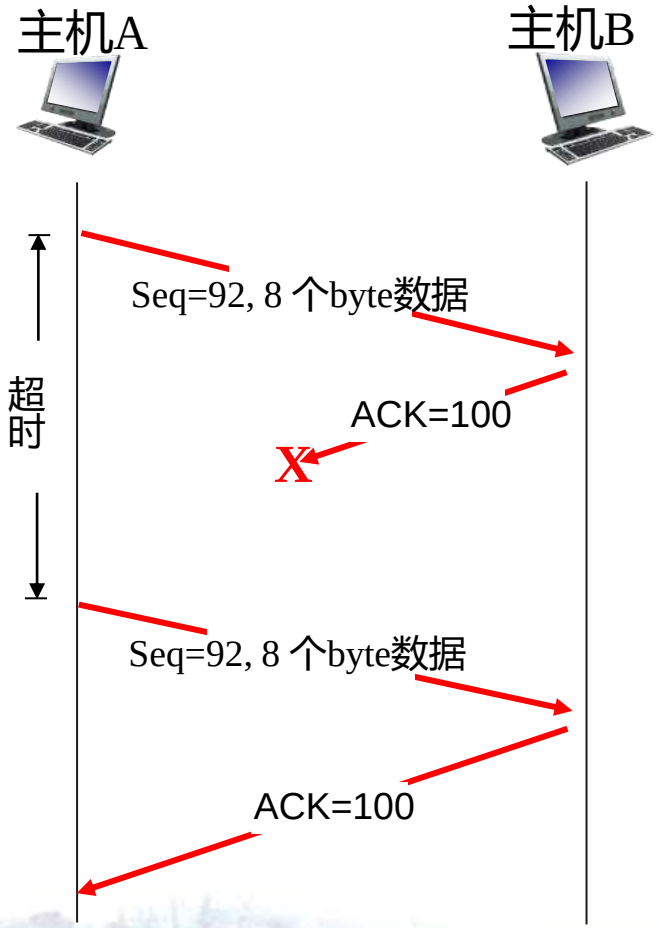
超时

重传未确认的具有最小序列号的分段
启动定时器

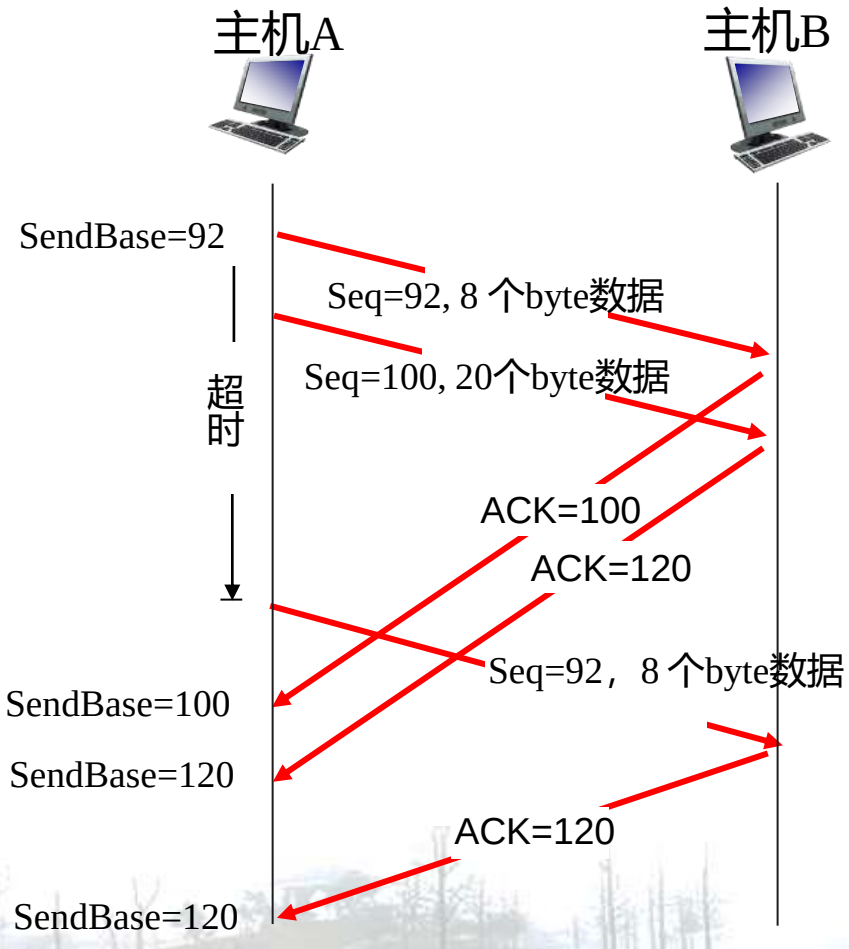
收到ACK，确认号=y

```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1是累积确认的最后一个byte*/
   if (仍然有未确认的分段)
      启动定时器
     else
      关闭定时器
   }
```

TCP重传举例

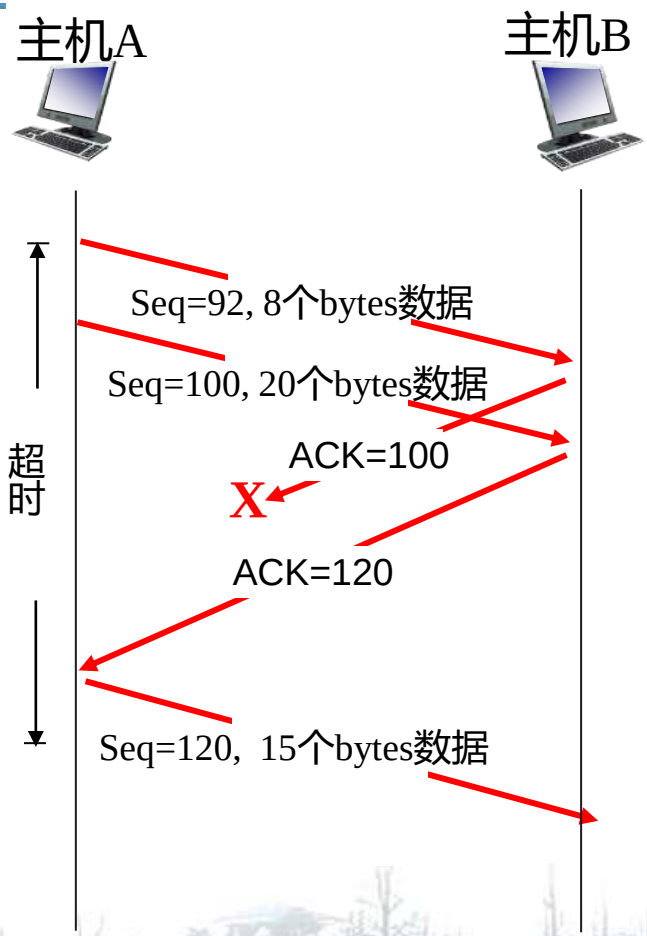


ACK丢失



ACK延误超时

TCP重传举例



累积确认ACK

TCP ACK产生

TCP接收端事件	TCP接收端动作
分段顺序到达，所有分段序列号之前的数据已经被确认	延迟发送ACK. 等待500ms，如果没有新的顺序到达的分段，确认该分段
分段顺序到达，并且有其它顺序到达的分段未被确认。	立刻发出一个累积确认ACK，确认两个顺序到达的分段
乱序到达一个分段，其序列号大于当前期望收到的序列号，形成一个序列号缺口（gap）	立刻发出一个重复的ACK，包含期望收到下一个序列号
到达的分段部分或全部补上了序列号缺口	如果该分段补上了缺口的低端，立刻发送ACK，包含期望收到下一个序列号

TCP快速重传

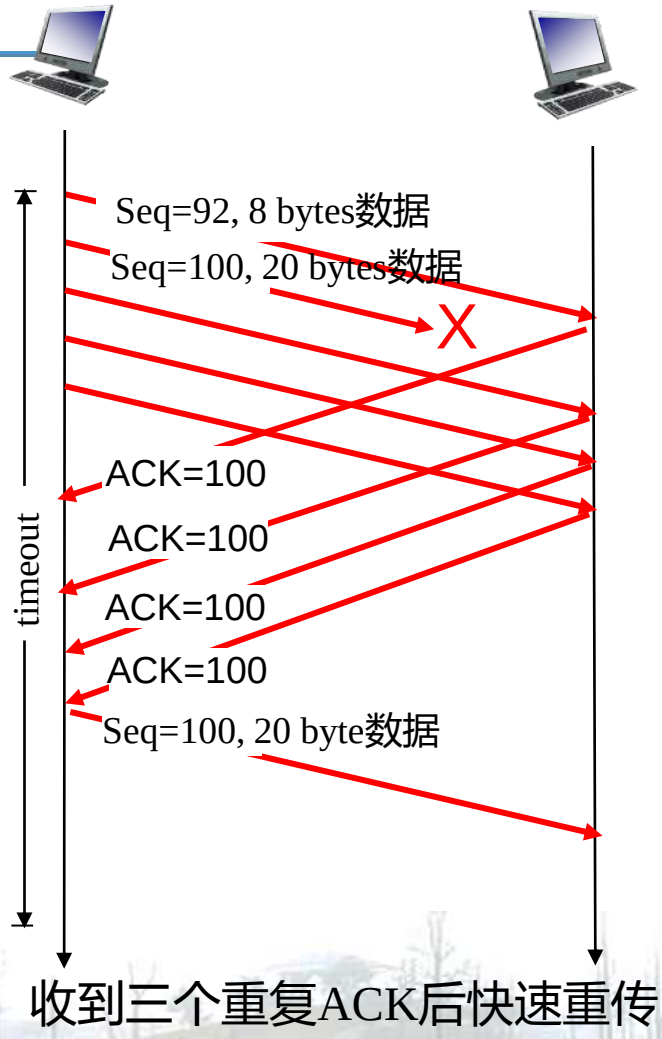
- 超时需要等较长时间:
  - 重发数据包前的时延很长
- 通过收到重复的ACK检测丢包.
  - 发送端通常背靠背发送数据包
  - 如果某个数据包丢失，后面的数据包会引发多个重复的ACK.

TCP快速重传

如果发送端收到3个相同确认号的ACK（“三个重复ACKs”），立刻重发尚未确认的最小序列号的分段

- 该分段大概率丢失，无需等到定时器超时

TCP快速重传



收到三个重复ACK后快速重传

讨论

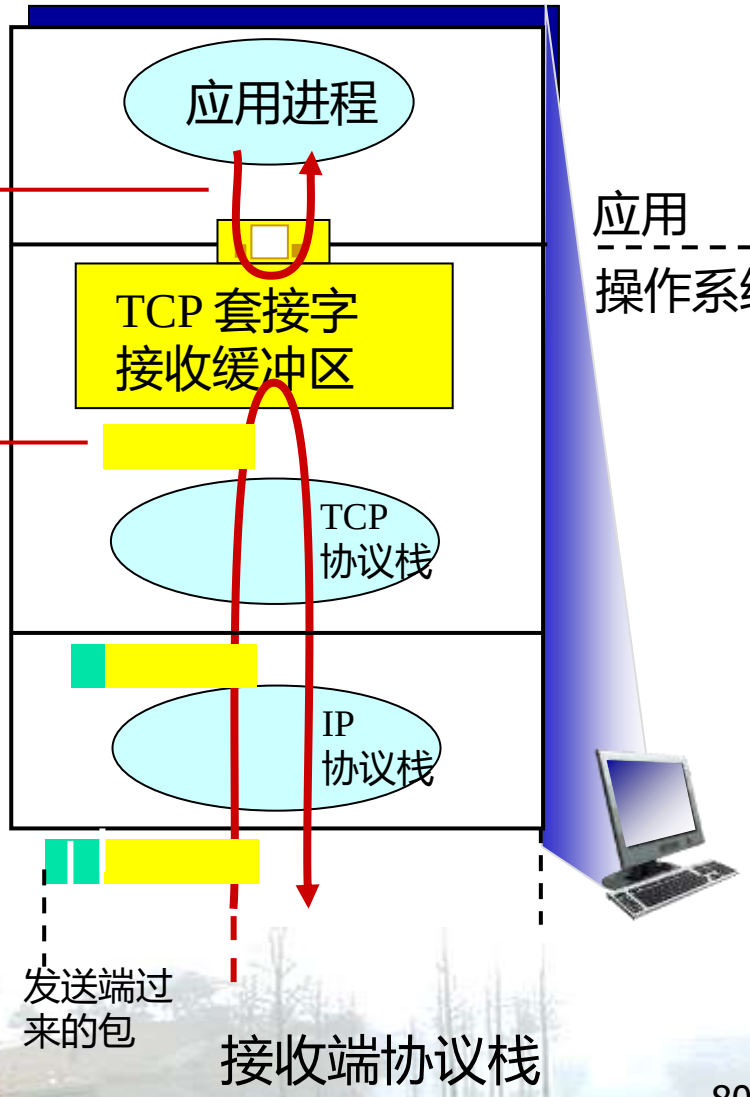
- 与回退N相似的地方：维护已发出未应答的最小序列号(SendBase)和将发出的下一个byte的序列号（NextSeqNum）；ack累积确认
- 与选择性重传相似的地方：（大多数实现）缓存乱序到达的数据，等待缺失的分段

目录

TCP流控制

应用程序从套接字缓冲区中取走数据….

…发送端发送速率必须慢于TCP向应用层交付数据的速率

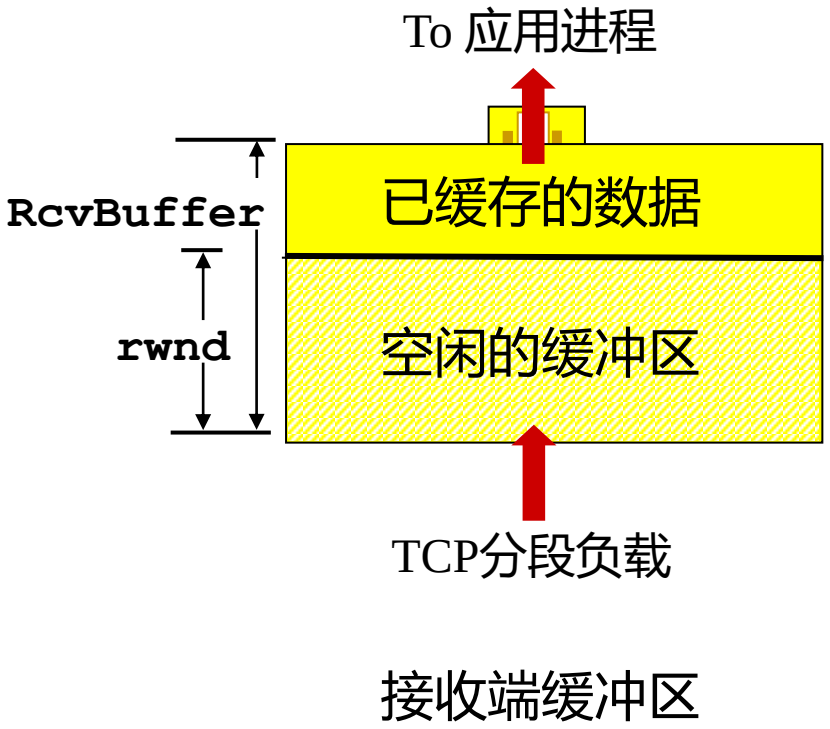


流控制

接收端控制发送端，使其发送速率不会太快，不会造成接收端缓冲区溢出

TCP流控制

- 接收端在从接收端到发送端的TCP分段头部**rwnd**字段里，告知发送端其当前空闲的缓冲区空间大小
  - 整个缓冲区大小**RcvBuffer**可由套接字选项设置 (一般缺省4096 bytes)
  - 多数操作系统自动调整**RcvBuffer**
- 发送端控制窗口，使得已发出w未确认的分段（即“在路上”的分段）不超过**rwnd**值
- 确保接收端缓冲区不至于溢出



TCP流控制

- 接收端维持两个变量
  - LastByteRead: 最后一个提交给应用进程的byte序列号
  - LastByteRcvd: 最后一个到达的byte序列号

$$rwnd = RcvBuffer - [LastByteRead - LastByteRcvd]$$

- 发送端维持两个变量
  - LastByteSent：最后一个发出去的byte序列号
  - LastByteAcked：最新确认的byte的序列号
  - 确保

$$LastByteSent - LastByteAcked \leq rwnd$$

TCP流控制

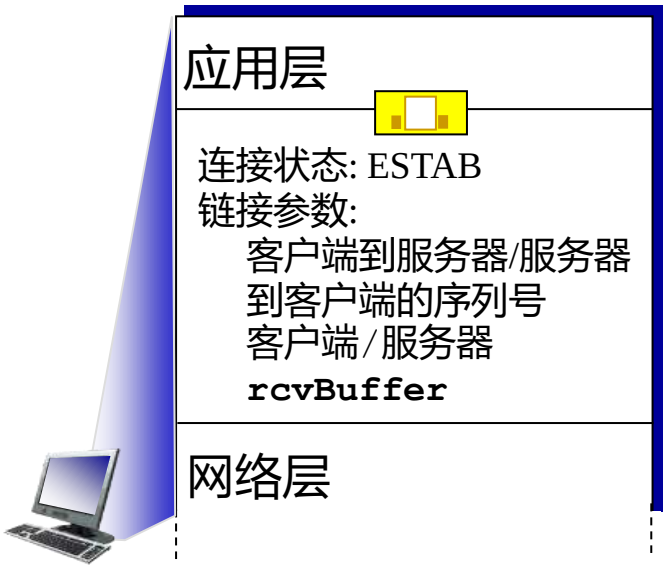
- 当接收端套接字缓冲区满
  - 告知发送端rwnd=0
  - 发送端停止发数据
  - 接收端缓冲区空了以后，没有数据需要确认，无法通知发送端继续发数据
- 当接收端rwnd=0，发送端继续发送1 byte大小的分段

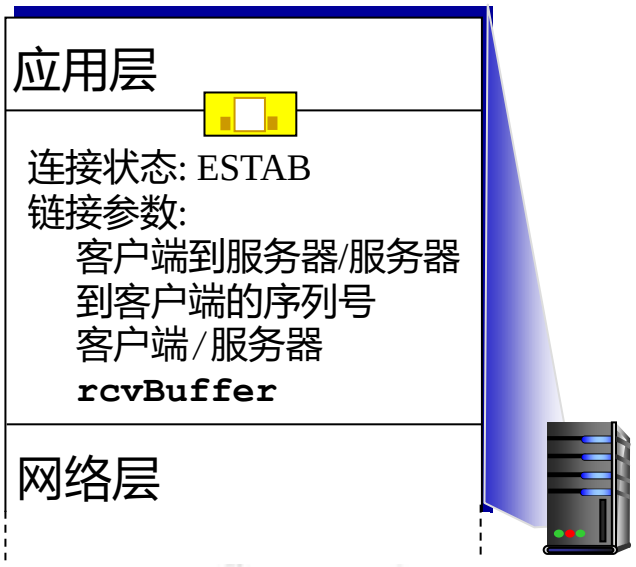
目录

连接管理

交换数据前，发送端和接收端需要“握手”

- 双方同意建立连接(每一方都要表达同意)
- 双方同意连接的相关参数（例如，初始序列号）






```
Socket clientSocket =

   newSocket("hostname","p
   ort number");
```

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

TCP3次交互握手

客户端状态

LISTEN

选择初始序列号，x
发送TCP SYN 消息

SYNSENT

SYNbit=1, Seq=x

选择初始序列号，y
发送TCP SYNACK
消息，确认 SYN

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

收到SYNACK(x)
表示服务器正常;
发送ACK 确认 SYNACK;
该消息可以携带客户端到服务器的数据

ESTAB

ACKbit=1, ACKnum=y+1

收到ACK(y)
表示客户端正常

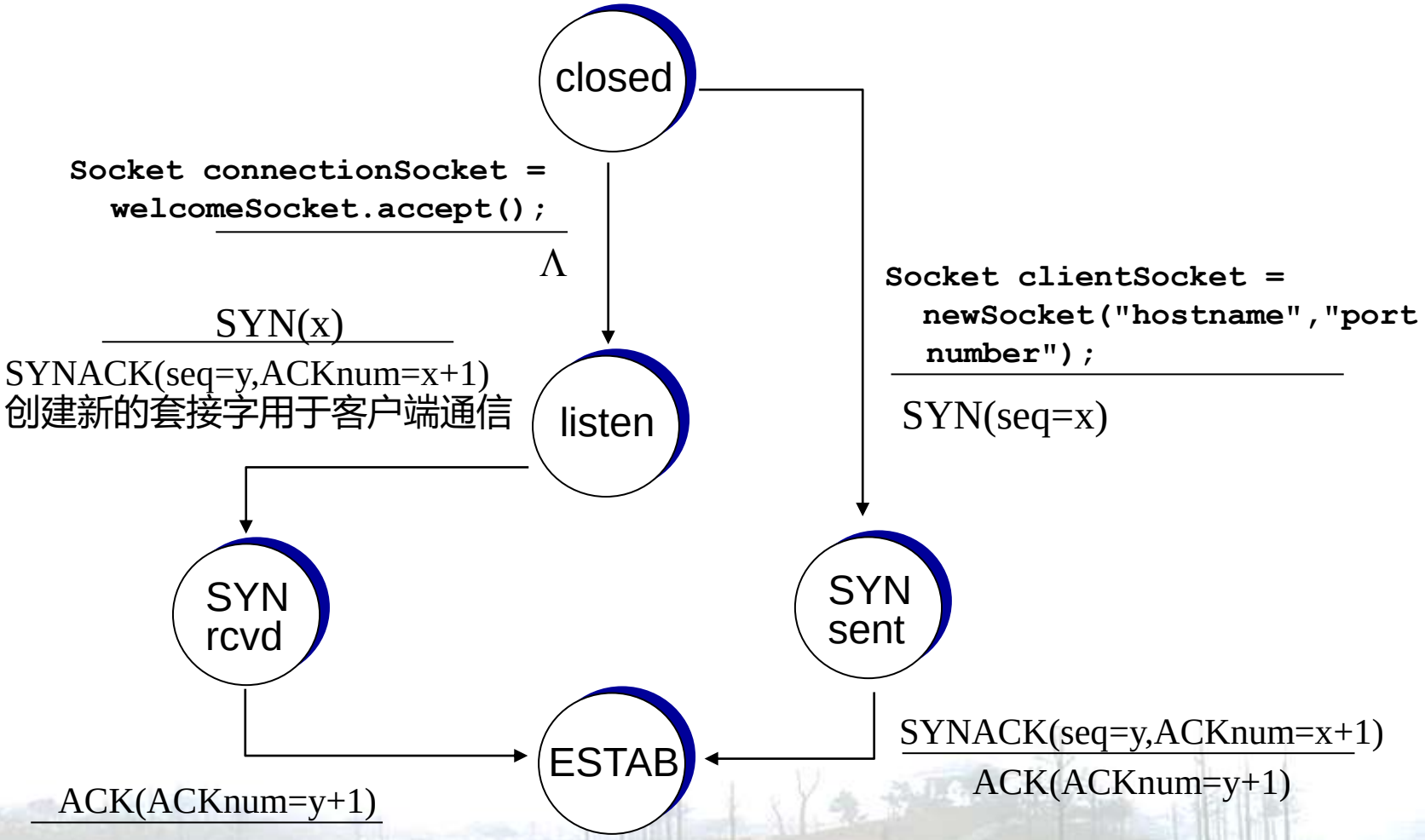
服务器状态

LISTEN

SYN RCVD

ESTAB

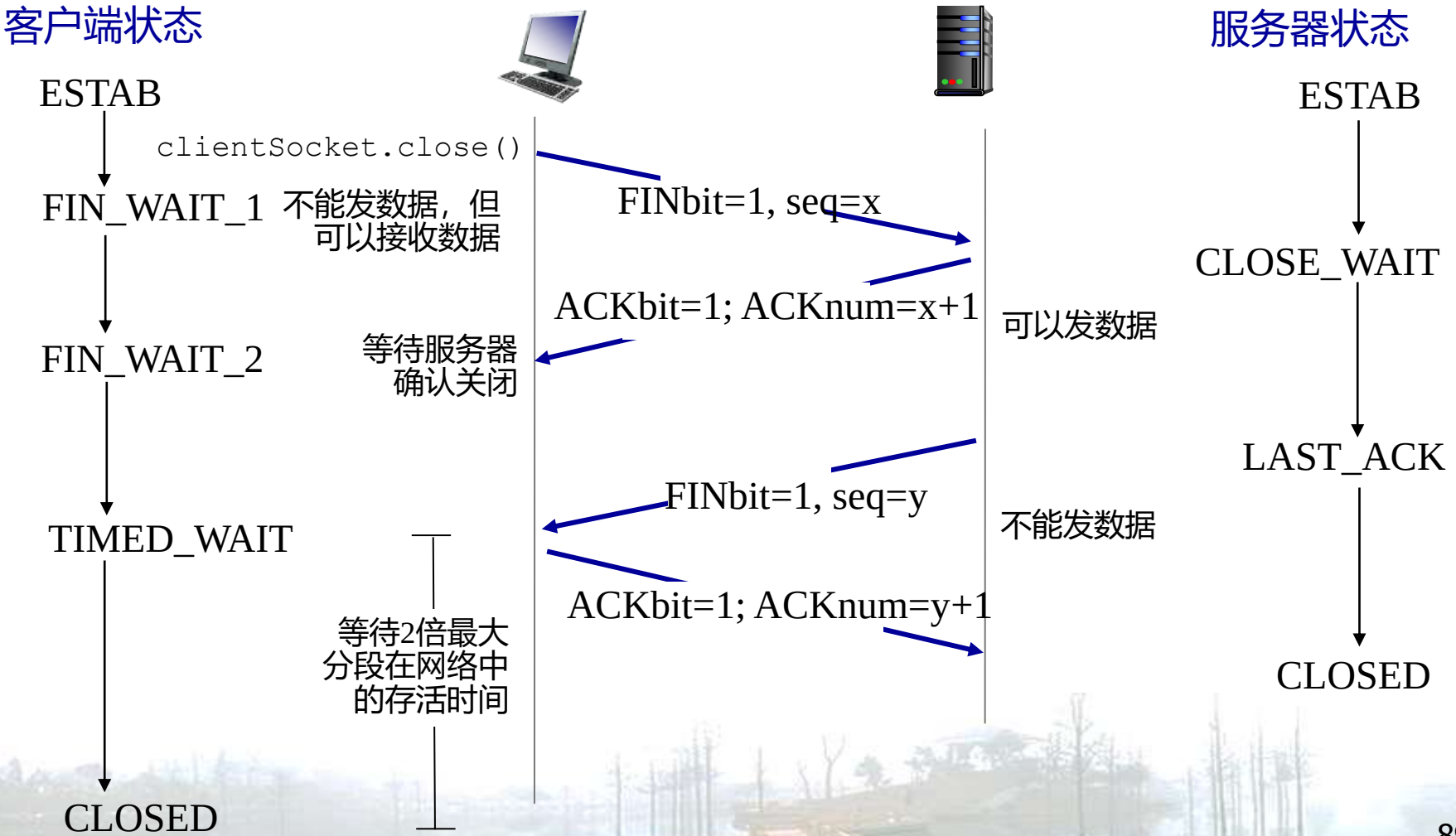
TCP3次交互握手FSM



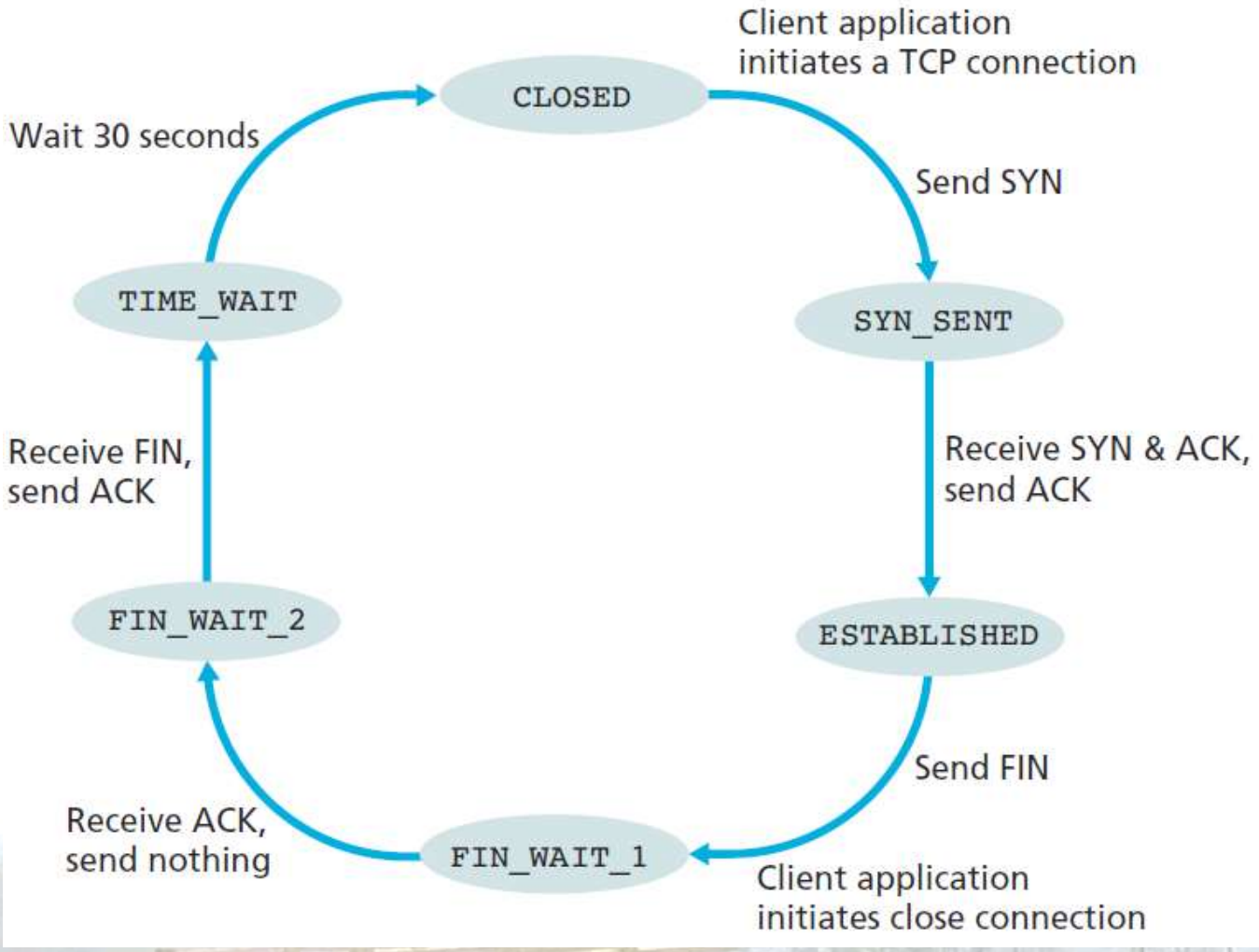
TCP：关闭连接

- 客户端和服务器都可以关闭连接
  - 在发送的TCP分段中设置FIN比特位 = 1
- 收到后回应设置ACK和FIN比特位=1的分段
  - 可以在同一个分段上设置ACK和FIN比特位
- 双方可以同时发出FIN

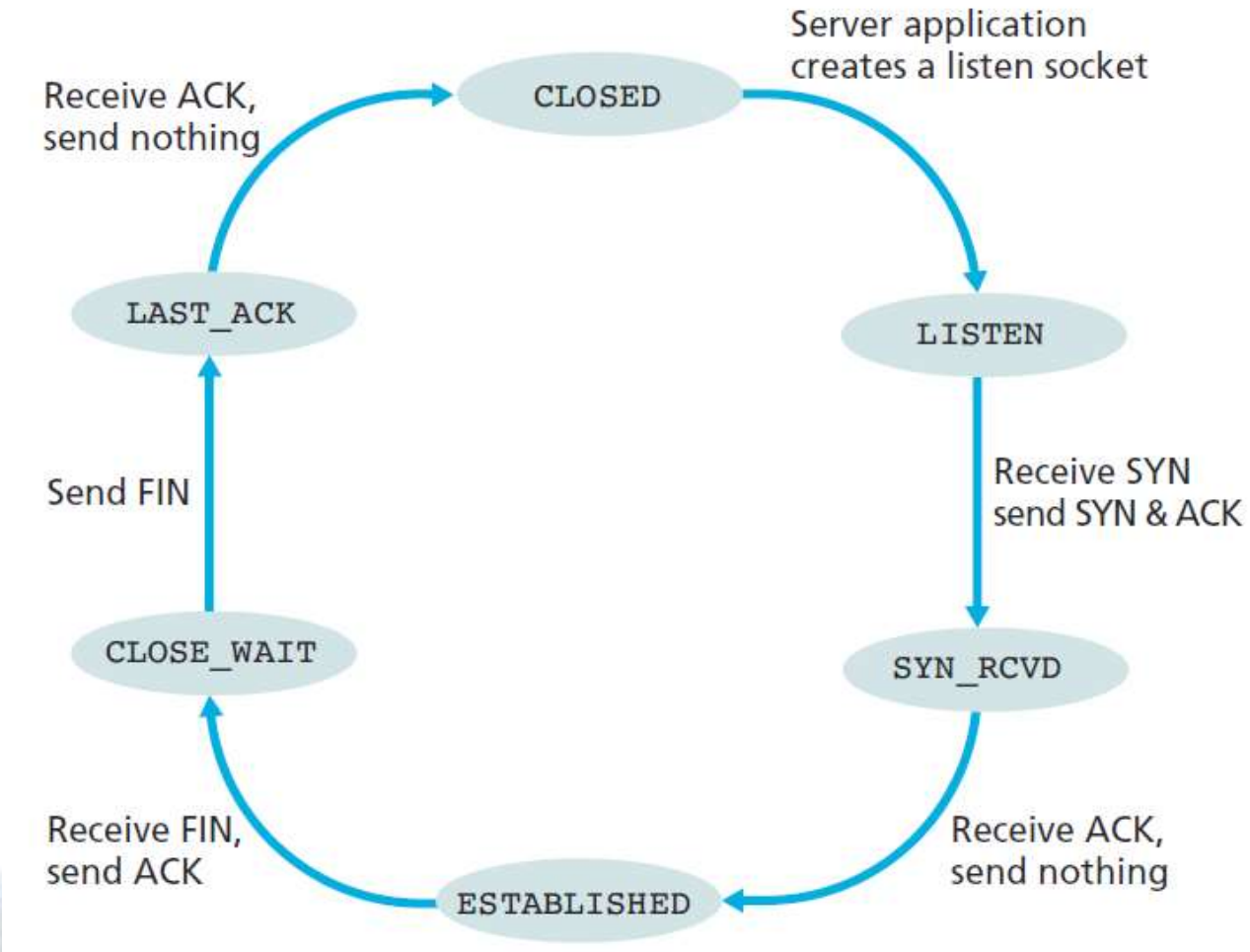
TCP：关闭连接


客户端状态
服务器状态
ESTAB
clientSocket.close()
FIN_WAIT_1
不能发数据，但
可以接收数据
FINbit=1, seq=x
ACKbit=1; ACKnum=x+1
可以发数据
FIN_WAIT_2
等待服务器
确认关闭
FINbit=1, seq=y
不能发数据
TIMED_WAIT
ACKbit=1; ACKnum=y+1
等待2倍最大
分段在网络中
的存活时间
CLOSED
ESTAB
CLOSE_WAIT
LAST_ACK
CLOSED

客户端TCP FSM


Client application
initiates a TCP connection
CLOSED
Send SYN
SYN_SENT
Receive SYN & ACK,
send ACK
ESTABLISHED
Send FIN
FIN_WAIT_1
Client application
initiates close connection
Receive ACK,
send nothing
FIN_WAIT_2
Receive FIN,
send ACK
TIME_WAIT
Wait 30 seconds

服务器TCP FSM


Server application
creates a listen socket
CLOSED
LISTEN
Receive SYN
send SYN & ACK
SYN_RCVD
Receive ACK,
send nothing
ESTABLISHED
Receive FIN,
send ACK
CLOSE_WAIT
Send FIN
LAST_ACK
Receive ACK,
send nothing

拒绝连接

- 服务器在端口x上收到TCP SYN连接请求分段
- 服务器没有在端口x上运行监听的套接字
- 服务器向链接发起端发送一个拒绝连接分段，设置RST比特位=1

目录

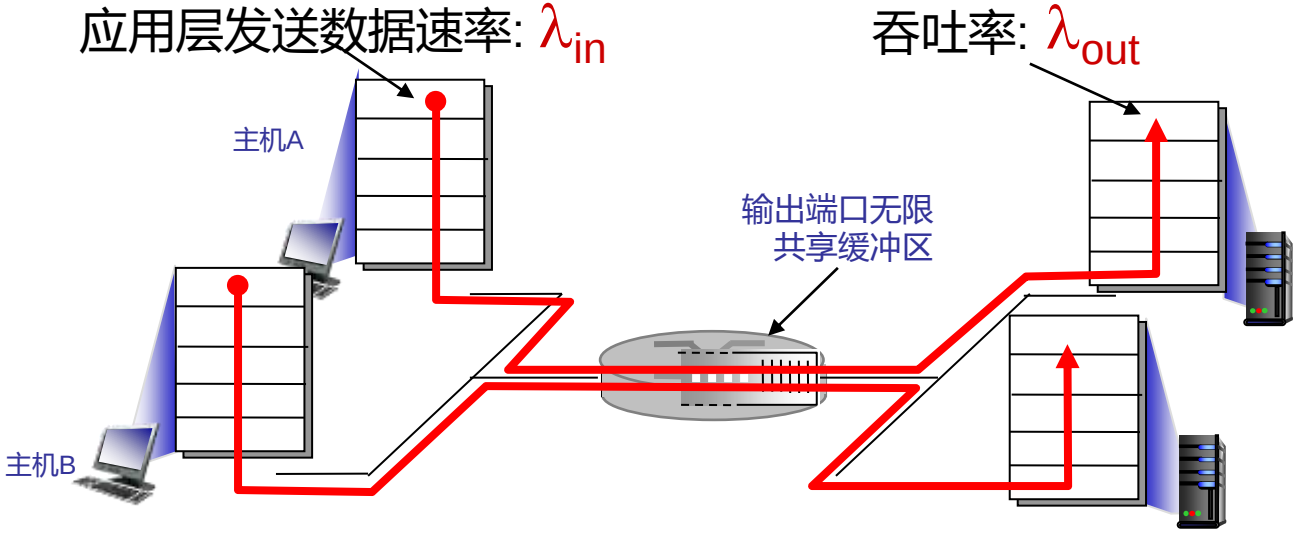
拥塞控制原理

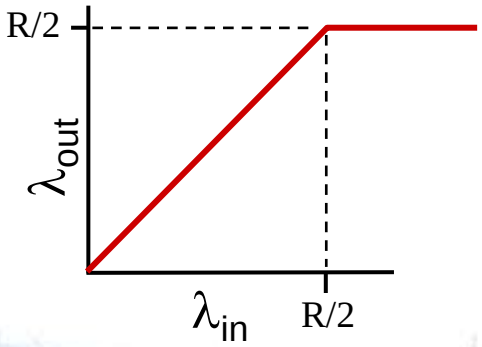
拥塞:

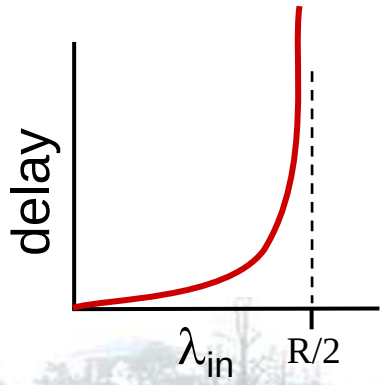
- 太多数据源太快发送太多数据，网络来不及处理
- 和流控制不同
- 拥塞产生的影响:
  - 丢包(路由器缓冲区溢出)
  - 长时延(数据包在路由器排队)
- 网络中前十重要的问题

拥塞的原因和代价：场景1

- 两个发送端、两个接收端
- 一个路由器，无限缓冲区
- 输出端口带宽：R
- 无重传



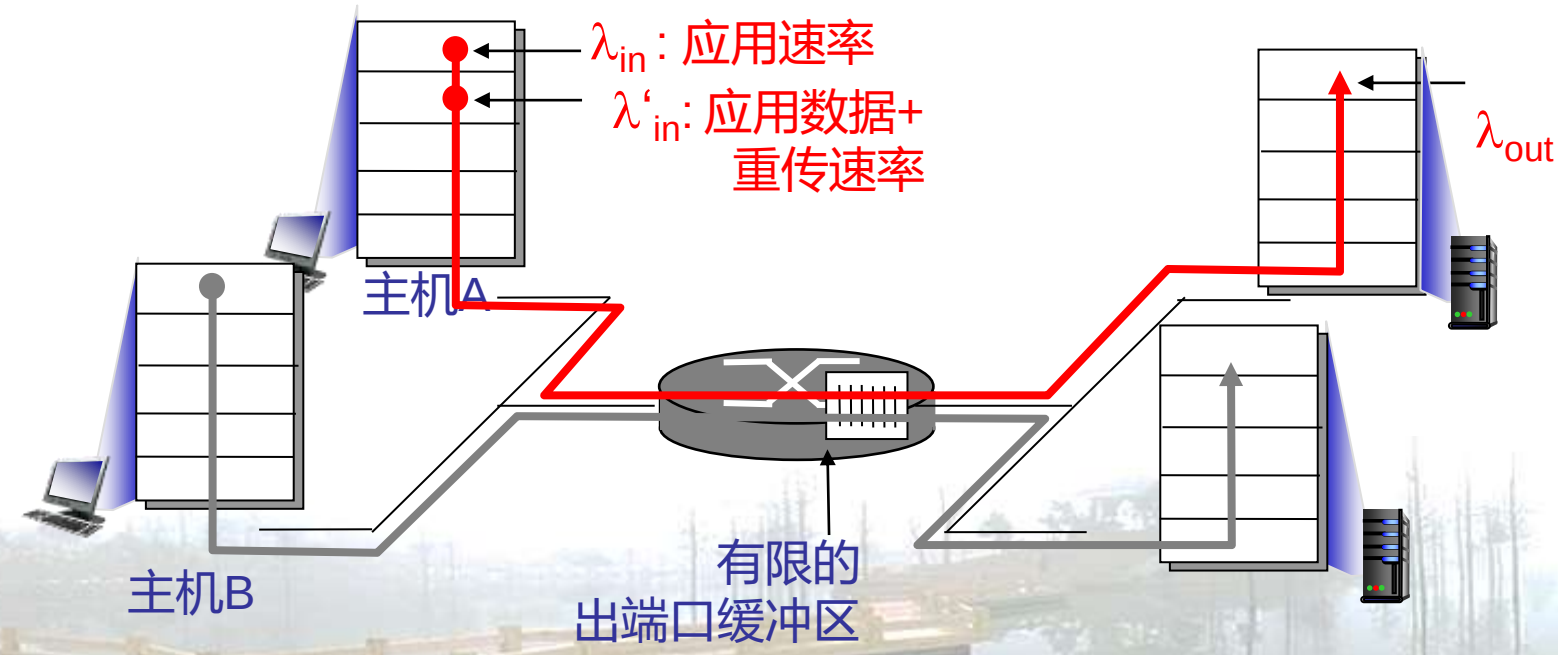




- 单个连接最大吞吐率 R/2
  - 当λ_{in}逼近R/2，时延趋近无限长

拥塞的原因和代价：场景2

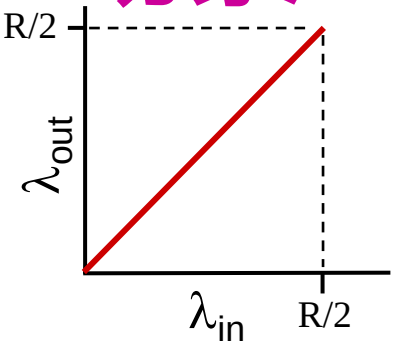
- 一个路由器，有限的缓冲区
- 发送端重传确认超时的数据包
  - 应用层输入=应用层输出: $\lambda_{in} = \lambda_{out}$
  - 传输层的输入: $\lambda'_{in} \geq \lambda_{in}$

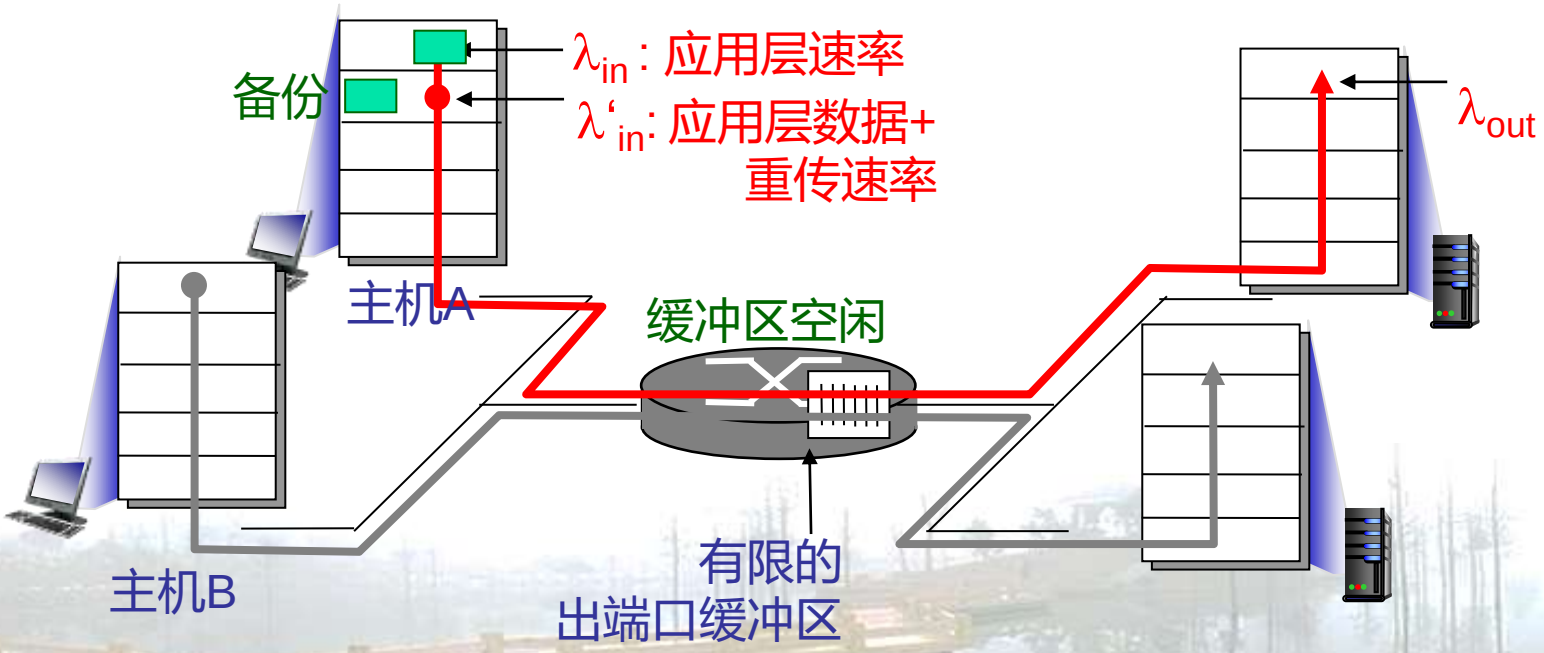


拥塞的原因和代价：场景2

理想情况

- 发送端知道路由器缓冲区何时有空位

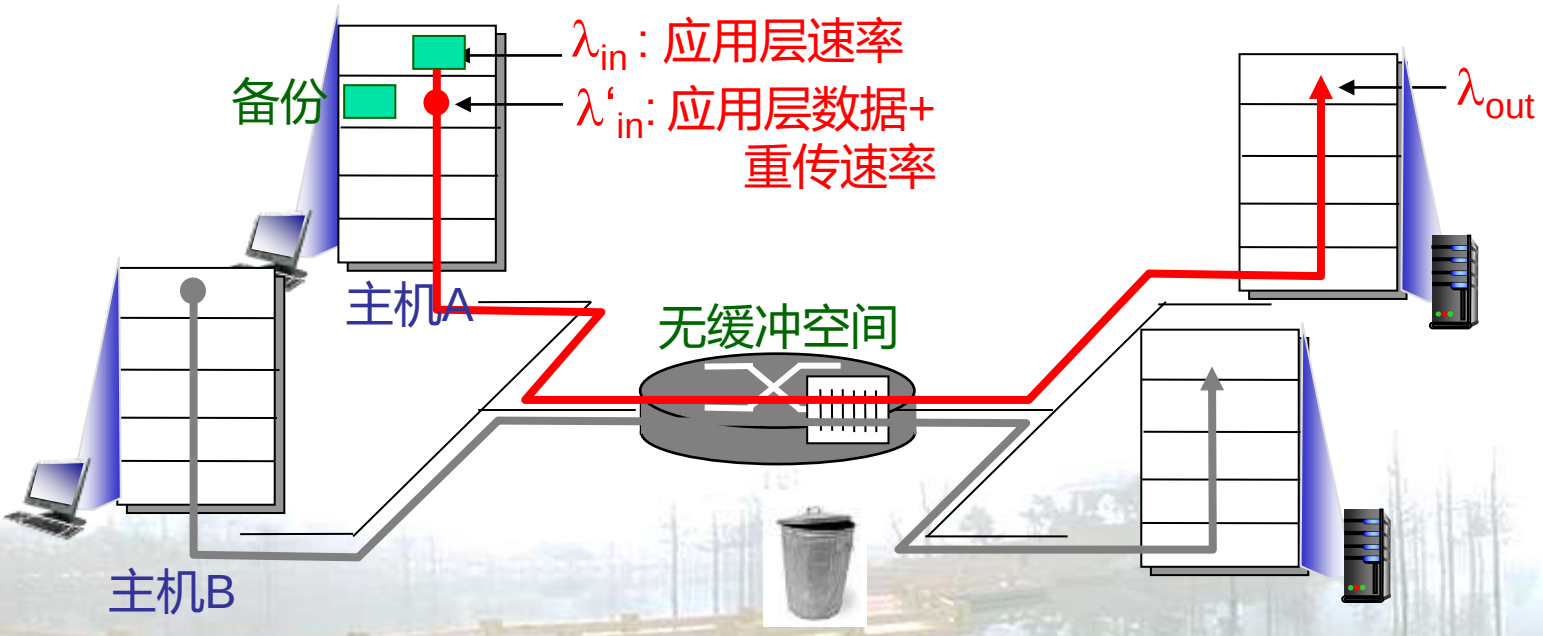




拥塞的原因和代价：场景2

理想情况：确切知道哪些数据包因为缓冲区满被丢弃了

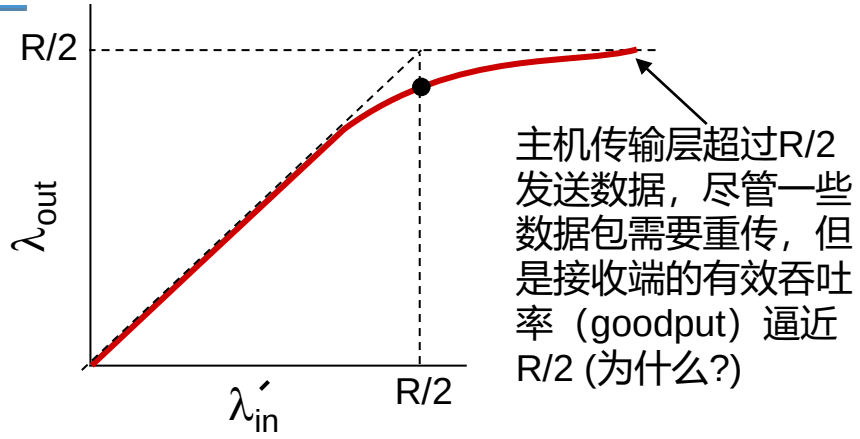
- 只重传被路由器丢掉的数据包

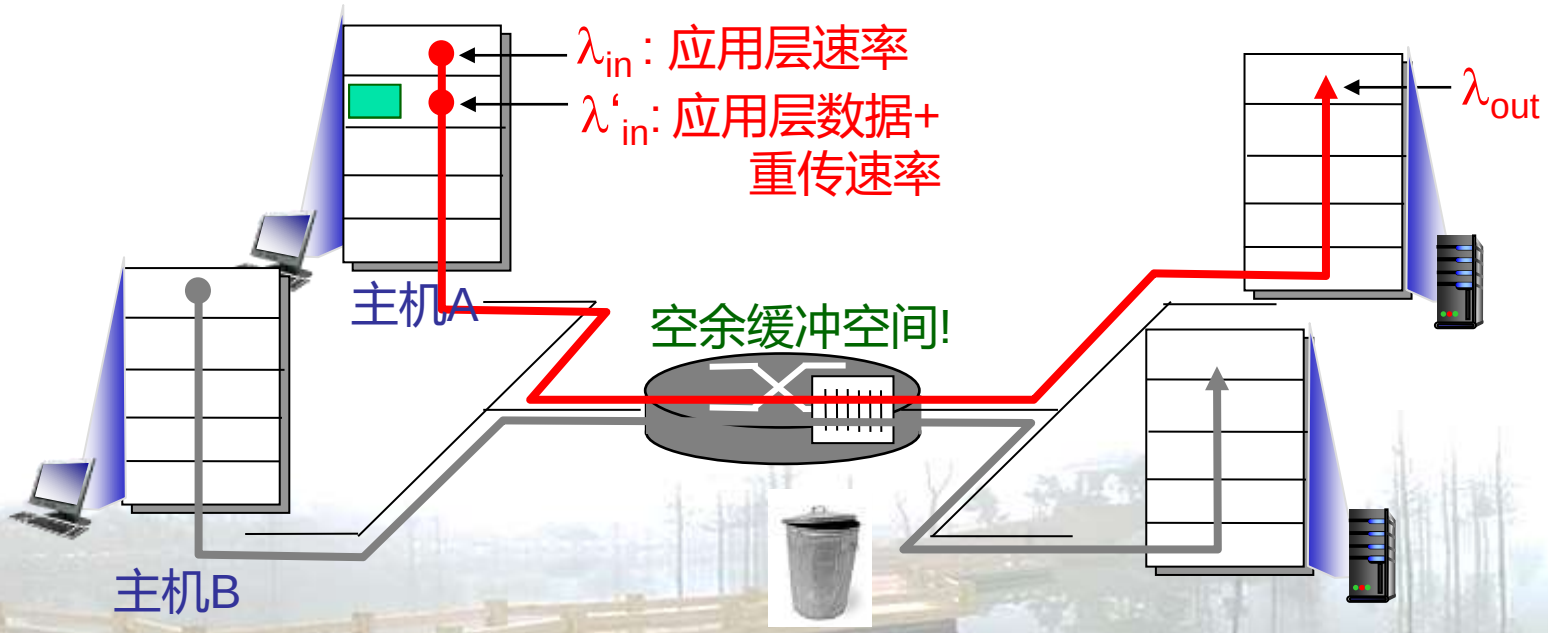


拥塞的原因和代价：场景2

理想情况：确切知道哪些数据包因为缓冲区满被丢弃了

- 只重传被路由器丢掉的数据包

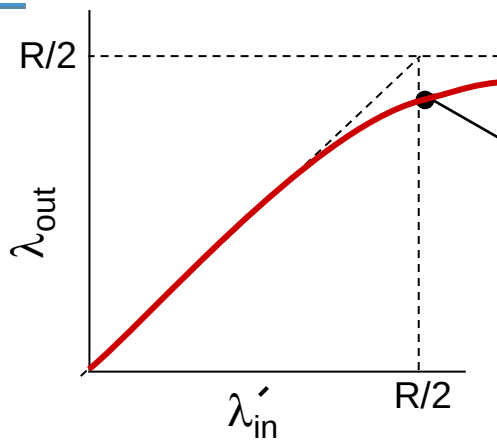




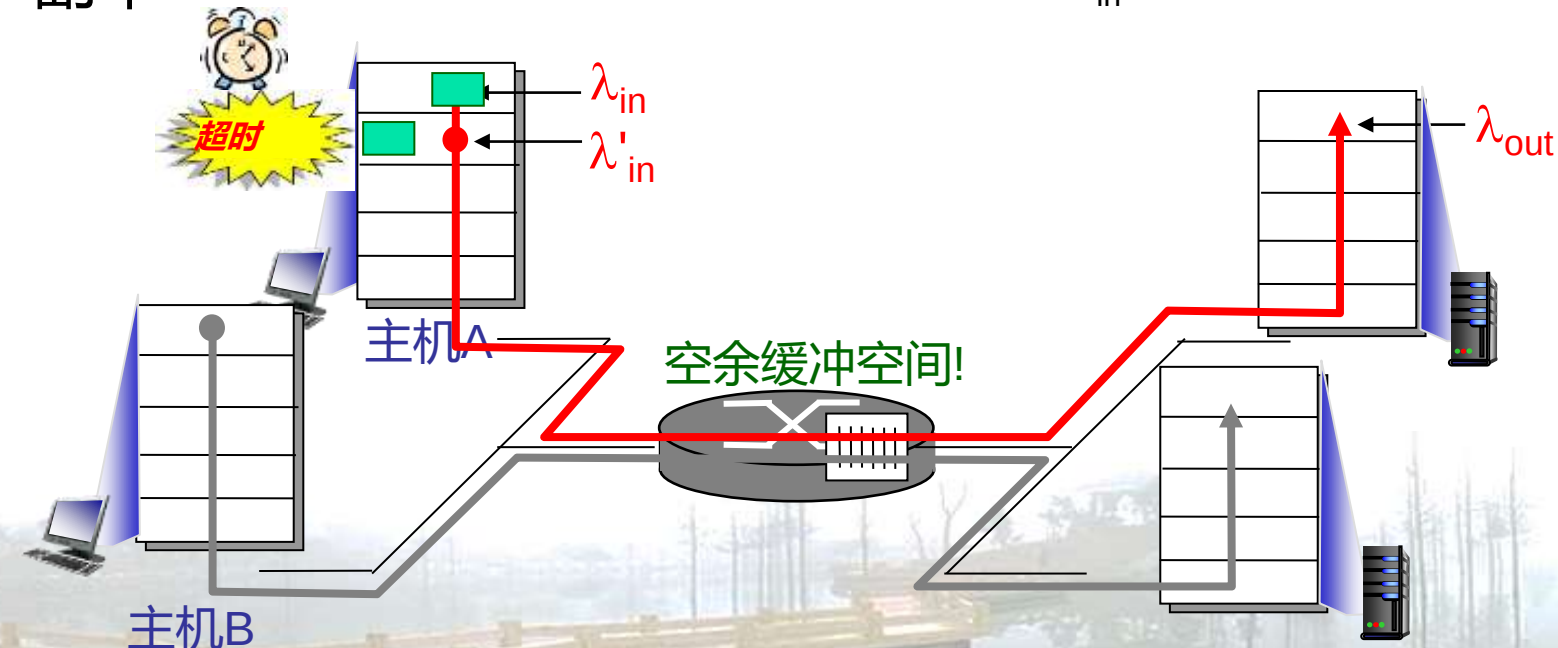
拥塞的原因和代价：场景2

真实情况: 重复传输

- 数据包可能因为缓冲区满被路由器丢弃
- 发送端可能因为过早超时重传，向接收端先后发出两个副本



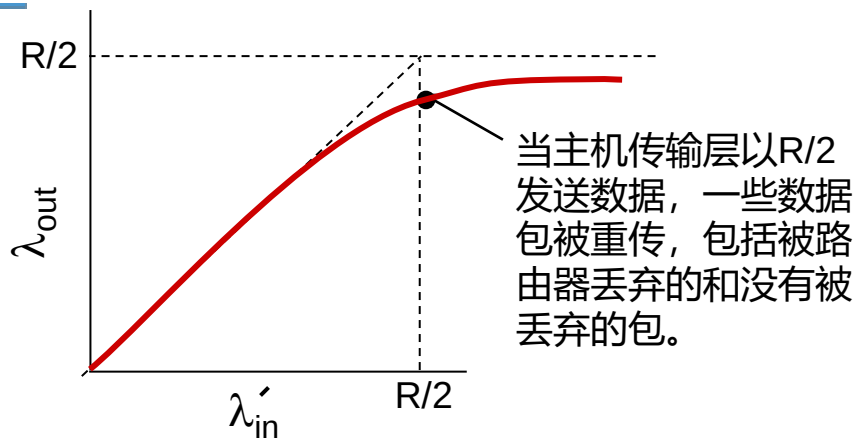
当主机传输层以R/2发送数据，一些数据包被重传，包括被路由器丢弃的和没有被丢弃的包。



拥塞的原因和代价：场景2

真实情况: 重复传输

- 数据包可能因为缓冲区满被路由器丢弃
- 发送端可能因为过早超时重传，向接收端先后发出两个副本



拥塞的代价：

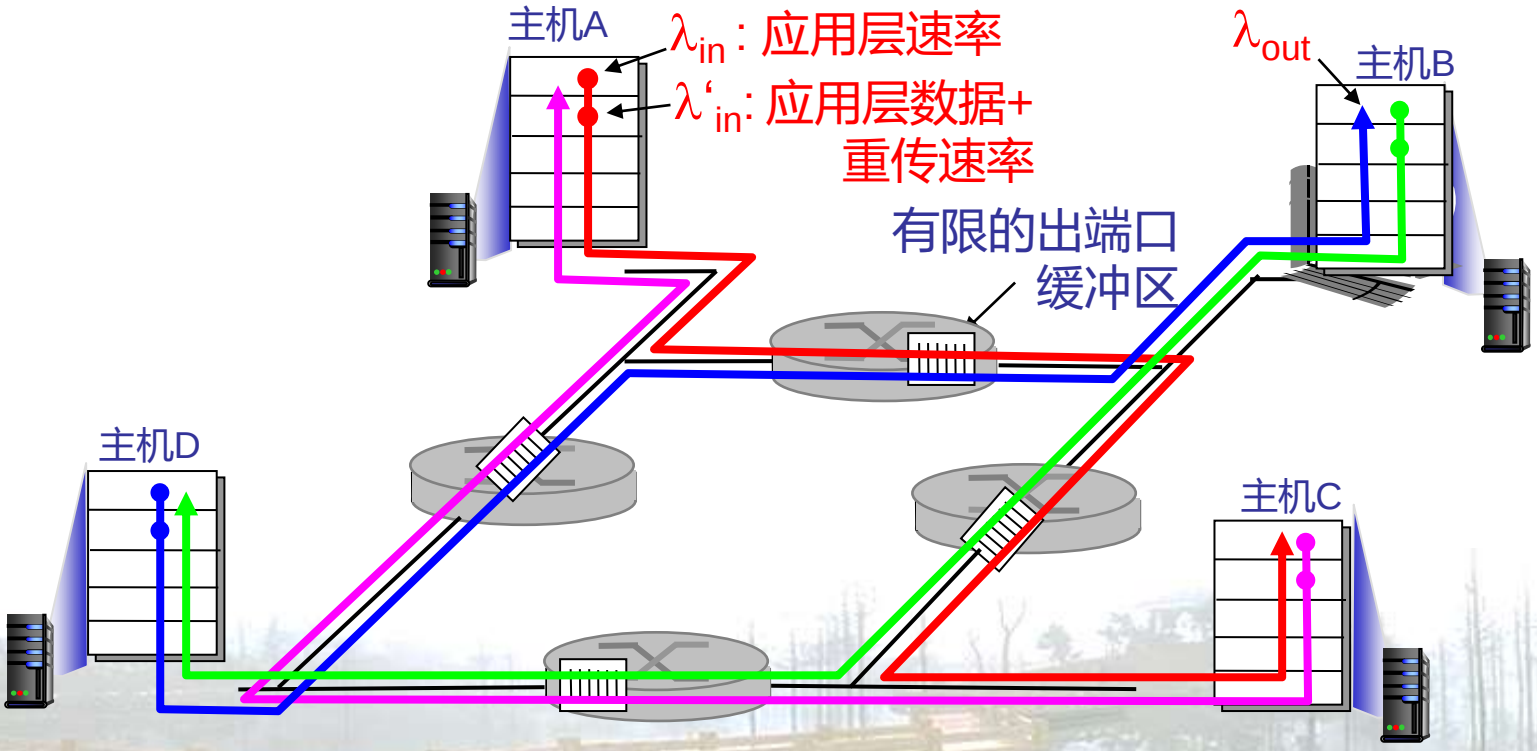
- 为实现给定的有效吞吐率（goodput），需要做额外的重传
- 不必要的重传，导致一个数据包的多个副本被送到接收端
  - 降低goodput

拥塞的原因和代价：场景3

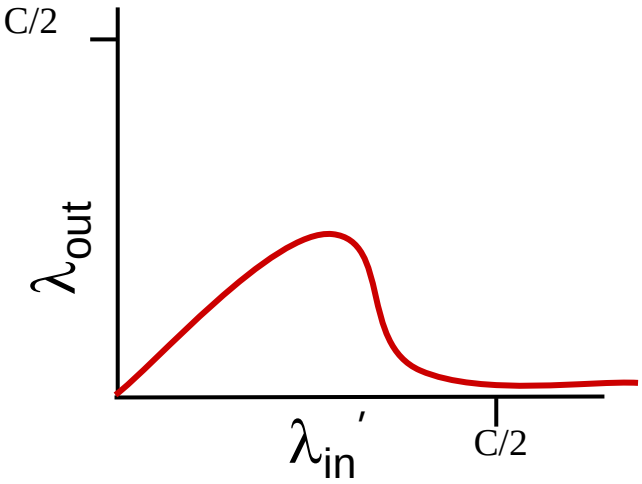
- 4个发送端
- 多跳路径
- 超时-重传

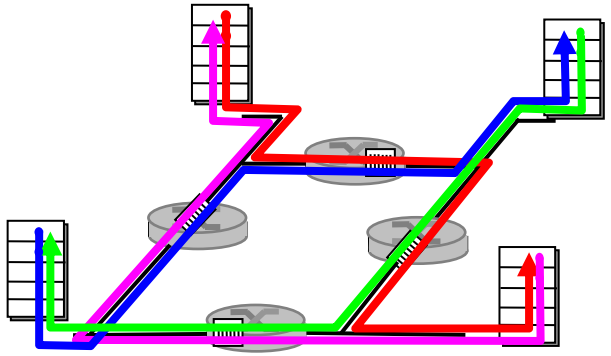
问: λ_{in} 和 λ_{in}' 增大会产生什么后果?

答: 当红色 λ_{in}' 增大, 所有蓝色数据包会被丢弃，导致蓝色吞吐率趋近于0



拥塞的原因和代价：场景3





拥塞的另一代价:

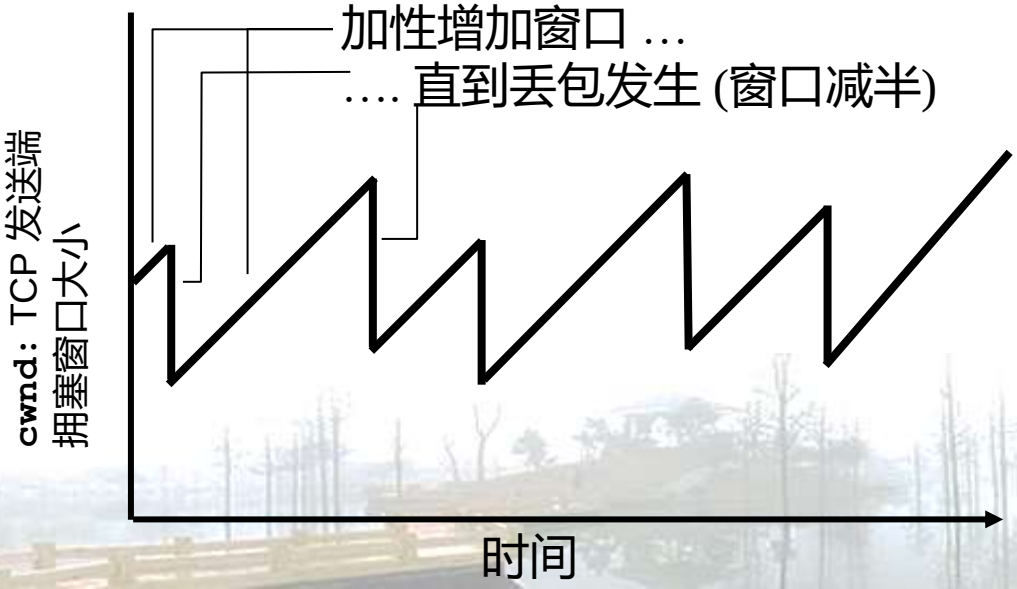
- 当拥塞导致丢包，数据包传输消耗的上游带宽被浪费了

目录

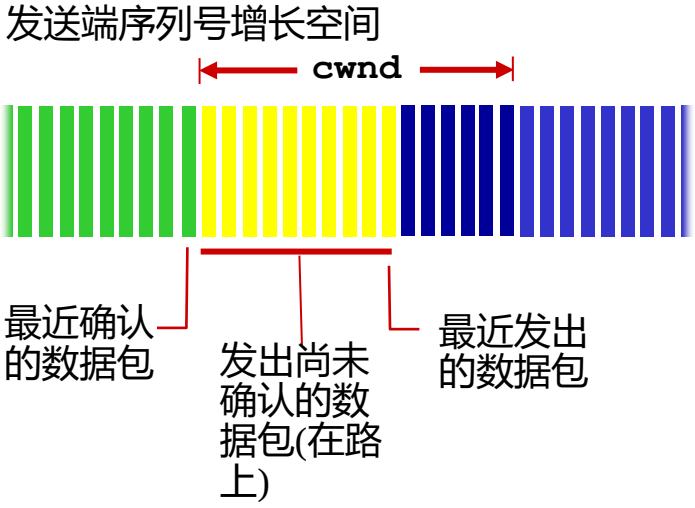
TCP拥塞控制：加性增、乘性减

- 发送端增加发送速率（窗口大小），试探使用更多的未占用带宽，直至丢包发生
  - 加性增：每个RTT增加拥塞窗口**cwnd**一个MSS，直到丢包
  - 乘性减：丢包时将 **cwnd** 减半

AIMD导致拥塞窗口锯齿状变化：试探更多带宽



TCP拥塞控制：细节



- 发送端控制:

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

- **cwnd** 根据网络拥塞情况动态变化

TCP 发送速率:

- 粗略可视为发出一个cwnd窗口的数据，等待ACK，再发出下一个窗口（更多）的数据

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

TCP拥塞控制：细节

- 实际上，发送端确保

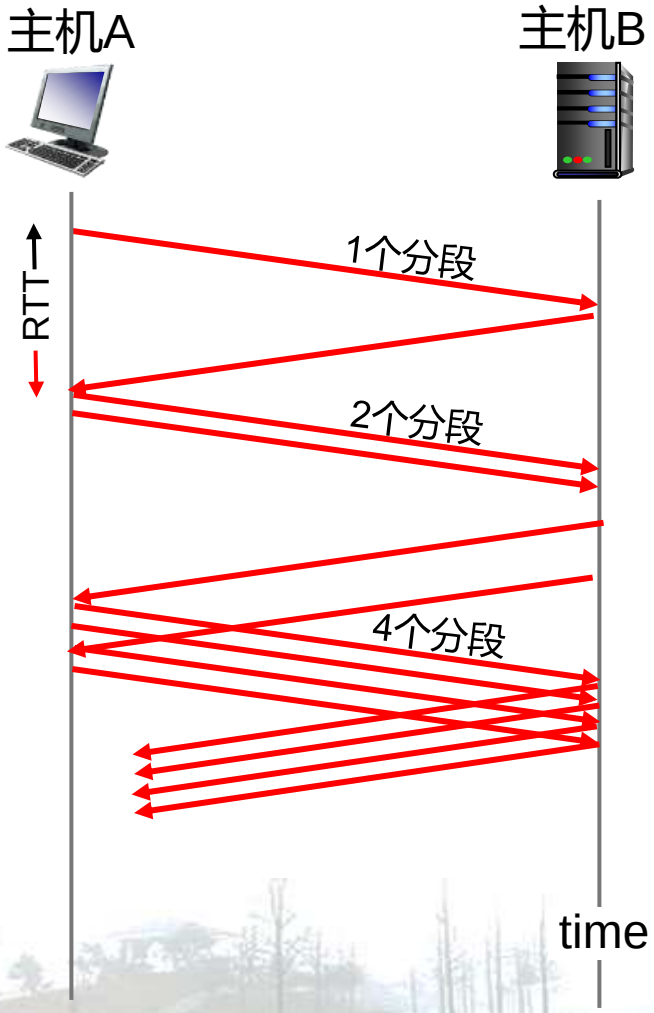
$$\text{LastByteSent} - \text{LastByteAcked} \leq \min\{\text{cwnd}, \text{rwnd}\}$$

TCP拥塞控制：细节

- 讨论两个拥塞控制算法
  - Tahoe和Reno
  - 实际上有很多
- 算法的状态
  - Tahoe：慢启动、拥塞避免
  - Reno：慢启动、拥塞避免、快速恢复
- 四种事件
  - 收到新的ACK、收到重复的ACK、超时、收到重复的ACK数量=3
- 算法改变两个变量
  - cwnd、 ssthresh

TCP慢启动

- 指数增大cwnd直到丢包发生:
  - 开始**cwnd** = 1 MSS
  - 每过一个RTT，**cwnd** 翻倍
  - 如何实现？每收到一个ACK，将cwd增加1个MSS
- 一开始速率很低，但是指数增长，慢启动并不慢



TCP：检测丢包和对丢包的反应

- 通过超时检测丢包：
  - **cwnd** 降为1个MSS;
  - 窗口指数增长（慢启动方式）直到到达一个门限，然后线性增长
- 通过收到3个重复ACK推测丢包： TCP Reno
  - 能收到重复ACK说明网络仍有一定带宽（后面的包都收到了）
  - 将**cwnd**减半，然后线性增长
- 无论超时还是收到3个重复ACK，TCP Tahoe都是将cwnd设置为1 MSS

TCP：由慢启动切换到拥塞避免

问: 何时从指数增长切换为线性增长?

答: 当**cwnd** 达到丢包之前窗口大小的一半

具体实现:

- 由变量 **ssthresh** 控制
- 当丢包发生时, **ssthresh** 设置为丢包前**cwnd**的1/2

设置初始ssthresh初始值（例如64KB）

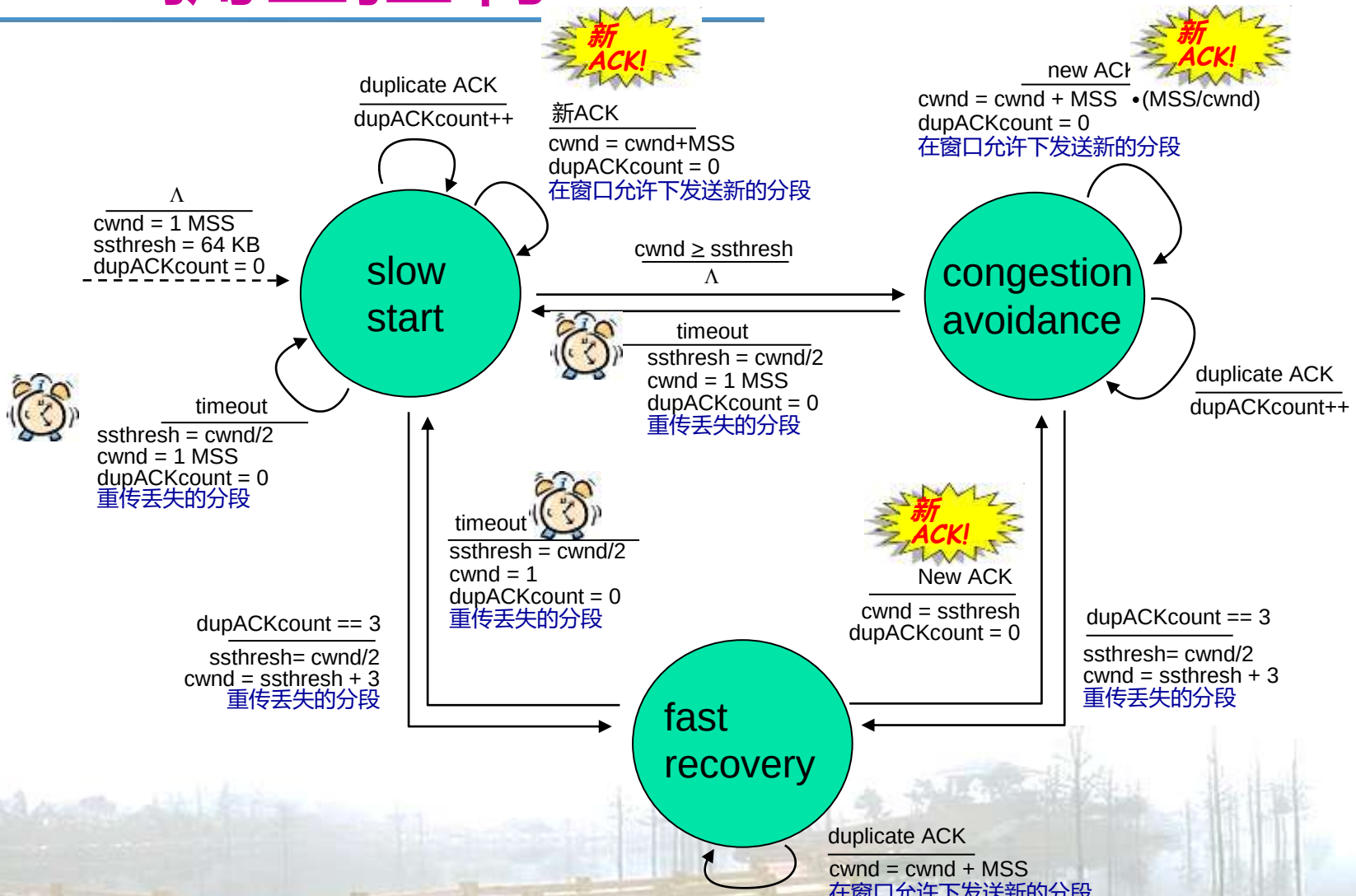
TCP：拥塞避免

- 每个RTT增加cwnd一个MSS
  - 每收到一个新的ACK，发送端cwnd增加MSS*(MSS/cwnd)
  - 等价于每个RTT增加cwnd一个MSS（线性增长）
- 何时结束?
  - ACK超时：
    - ssthresh = cwnd / 2, cwnd = 1 MSS，进入慢启动状态(Tahoe和Reno)
  - 收到3个重复的ACK:
    - ssthresh = cwnd/2, cwnd = ssthresh+3，进入快速恢复状态 (Reno)
    - 和ACK超时一样(Tahoe)

TCP：快速恢复

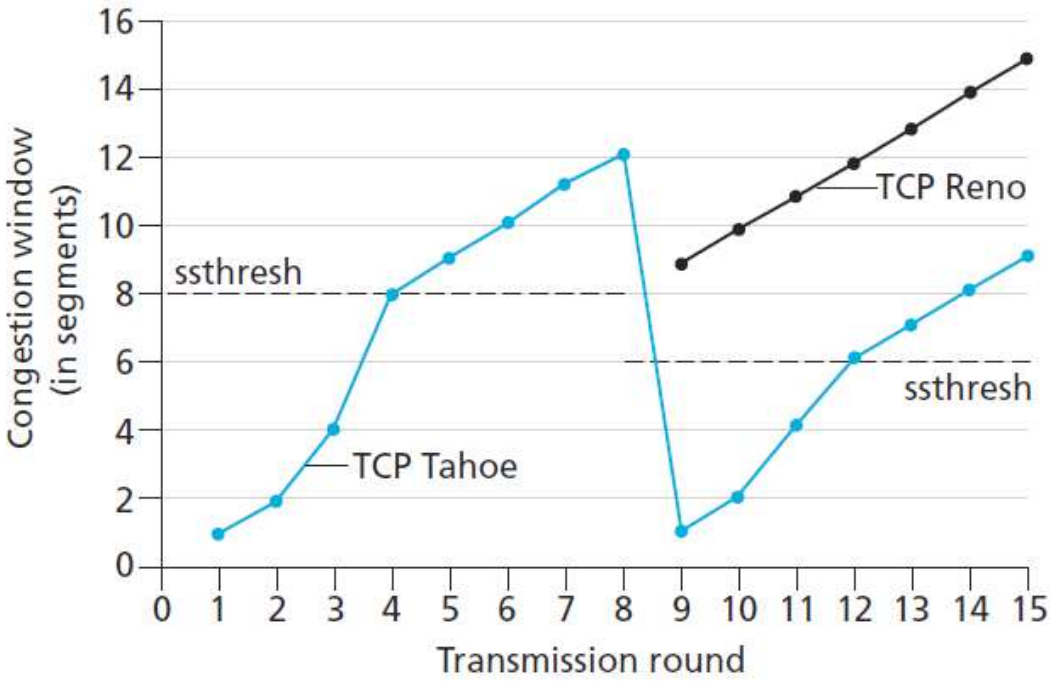
- 继续收到重复的ACK，每次增加cwnd一个MSS
- 收到新的ACK：
  - 设置cwnd = ssthresh，进入拥塞避免状态
- ACK超时：
  - ssthresh = cwnd / 2, cwnd = 1 MSS，进入慢启动状态

TCP拥塞控制FSM


新ACK!
duplicate ACK
dupACKcount++
新ACK
cwnd = cwnd+MSS
dupACKcount = 0
在窗口允许下发送新的分段
new ACK
cwnd = cwnd + MSS •(MSS/cwnd)
dupACKcount = 0
在窗口允许下发送新的分段
Λ
cwnd = 1 MSS
ssthresh = 64 KB
dupACKcount = 0
slow start
cwnd ≥ ssthresh
Λ
congestion avoidance
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
重传丢失的分段
duplicate ACK
dupACKcount++
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
重传丢失的分段
timeout
ssthresh = cwnd/2
cwnd = 1
dupACKcount = 0
重传丢失的分段
新ACK!
New ACK
cwnd = ssthresh
dupACKcount = 0
dupACKcount == 3
ssthresh= cwnd/2
cwnd = ssthresh + 3
重传丢失的分段
dupACKcount == 3
ssthresh= cwnd/2
cwnd = ssthresh + 3
重传丢失的分段
fast recovery
duplicate ACK
cwnd = cwnd + MSS
在窗口允许下发送新的分段

Reno和Tahoe举例

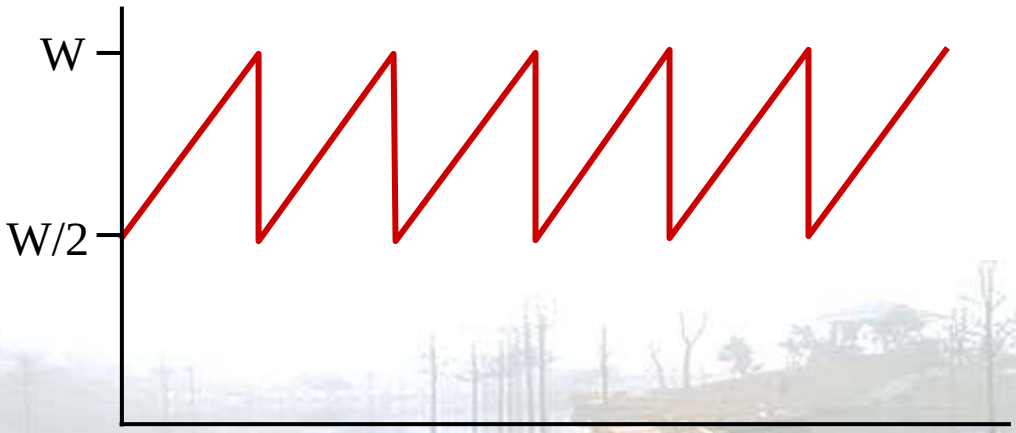
- 初始 ssthresh=8 MSS
- 开始8个RTT，Tahoe和Reno相同
  - cwnd涨到12 MSS
- 在第8个RTT检测到丢包(3个重复Ack)
  - 设置ssthresh=6 MSS (Tahoe和Reno)
  - Tahoe: cwnd降为1 MS
  - Reno: cwnd 降为6+3=9 MSS



TCP吞吐率

- TCP的平均吞吐率由窗口大小和RTT决定
  - 忽略慢启动，假设应用层一直有数据待传输
- W: 丢包发生时的窗口大小（单位bytes）
  - 平均窗口大小（也是平均“在路上”的byte数）¾ W
  - 平均吞吐率：每个RTT传输3/4W bytes

$$\text{平均TCP吞吐率} = \frac{3}{4}\frac{W}{RTT}\ \text{bytes/sec}$$



在"长、肥"管道上的TCP吞吐率

- 举例: 分段大小1500 byte, 100ms RTT, 希望实现10Gbps的吞吐率
- 根据上一页公式，需要W = 83,333个分段，显然不现实
- 吞吐率取决于传输路径的丢包率L [Mathis 1997]:

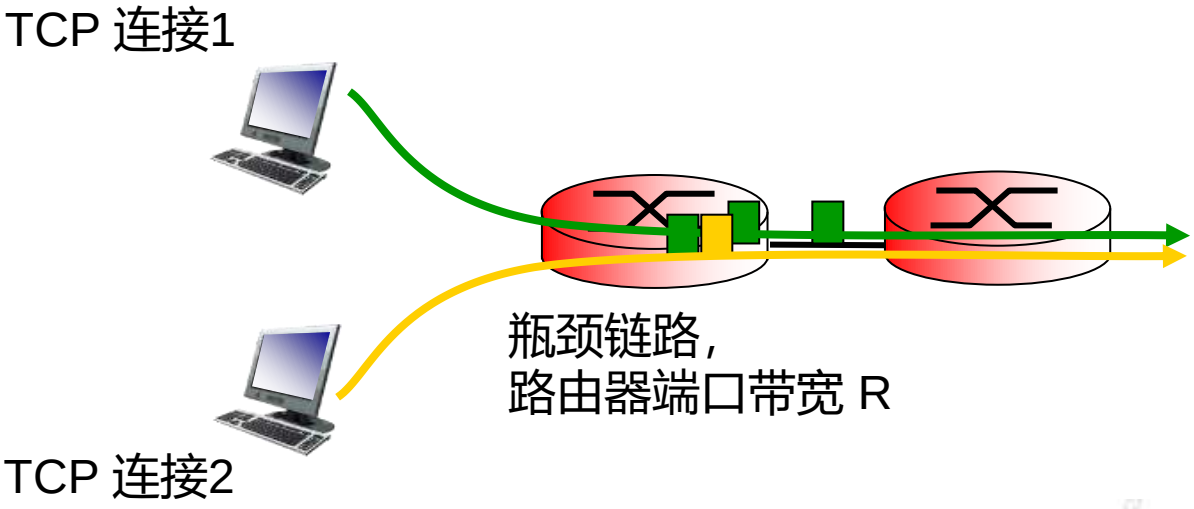
$$\text{TCP 吞吐率} = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

➜ 要实现10 Gbps 吞吐率, 丢包率L = $2 \cdot 10^{-10}$ –要求非常低的丢包率

- 实现高吞吐率的TCP拥塞控制是研究难点

TCP的公平性

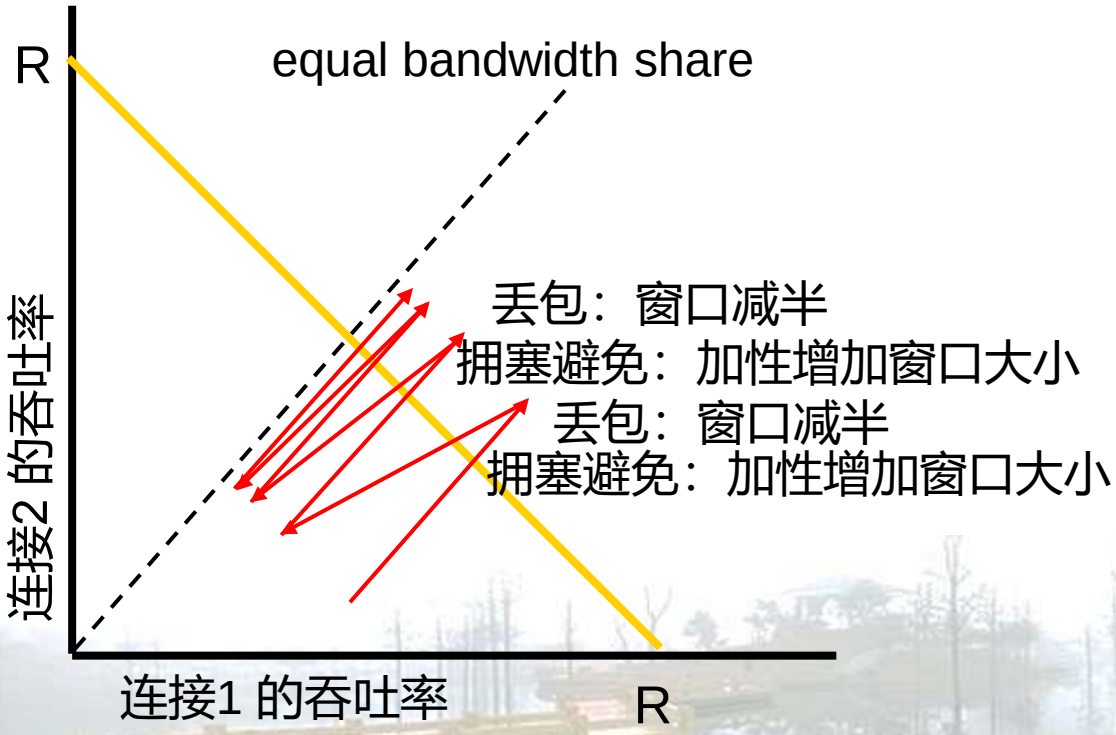
公平性目标: K个TCP会话共享一个瓶颈链路，带宽为R，则每个连接的平均吞吐率应为R/K



为什么TCP是公平的?

两个TCP会话竞争:

- 加性增导致当两个连接的吞吐率增加，沿斜率1上升
- 乘性减导致x和y坐标减半



公平性

UDP和公平性

- 多媒体应用通常不使用TCP
  - 不希望速率被拥塞控制限制
- 使用UDP:
  - 以恒定速率发送音频/视频数据，容忍丢包

多个并发TCP连接的公平性

- 两个主机上的应用程序可能建立多个TCP连接
- 浏览器
- 例如：带宽为R的瓶颈链路上已有9个TCP连接：
  - 新的应用建立1条TCP连接，获得R/10的带宽
  - 新的应用建立11条TCP连接，获得R/2的带宽

显式拥塞通知（Explicit Congestion Notification，ECN）

网络辅助拥塞控制:

- 发生拥塞的路由器设置IP头部两个bit (ToS字段)
- 接收端收到携带指示的IP报文
- 接收端在ACK分段中设置 ECE比特位，通知发送端路径上存在拥塞
- 发送端将cwnd减半，在下一个分段设置CWR比特位