第3章 运输层
Transport Layer

可靠数据传输原理

任课老师：周军海
Email:rj_zjh@hnu.edu.cn

第3章 要点

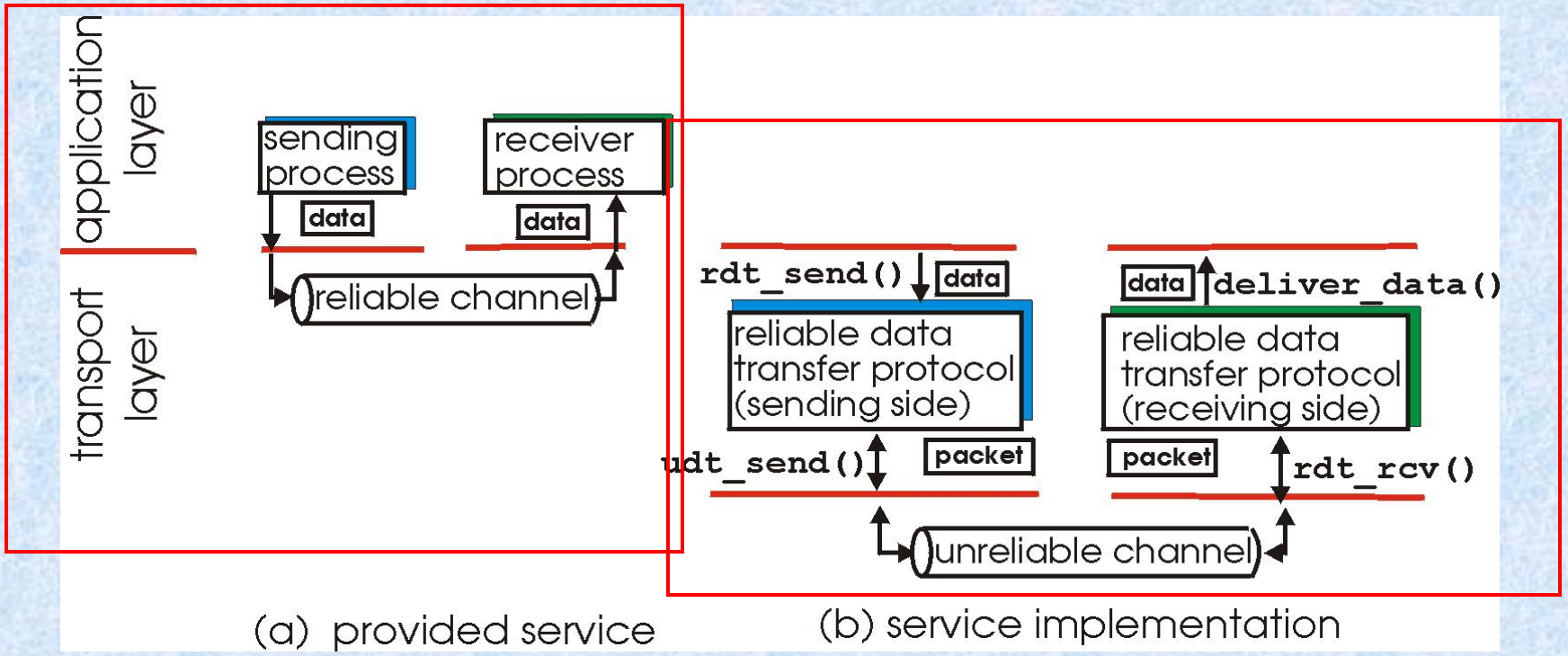
- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

教学重点&难点

- 教学重点
  - 理解可靠数据传输原理和概念
  - 分析三种可靠数据传输协议
  - 分析和应用流水线协议
- 教学难点
  - 分析三种可靠数据传输协议
  - 分析和应用流水线协议

可靠数据传输原理

- 在应用层、运输层、数据链路层的重要性
  - 网络中需解决的最重要问题之一!

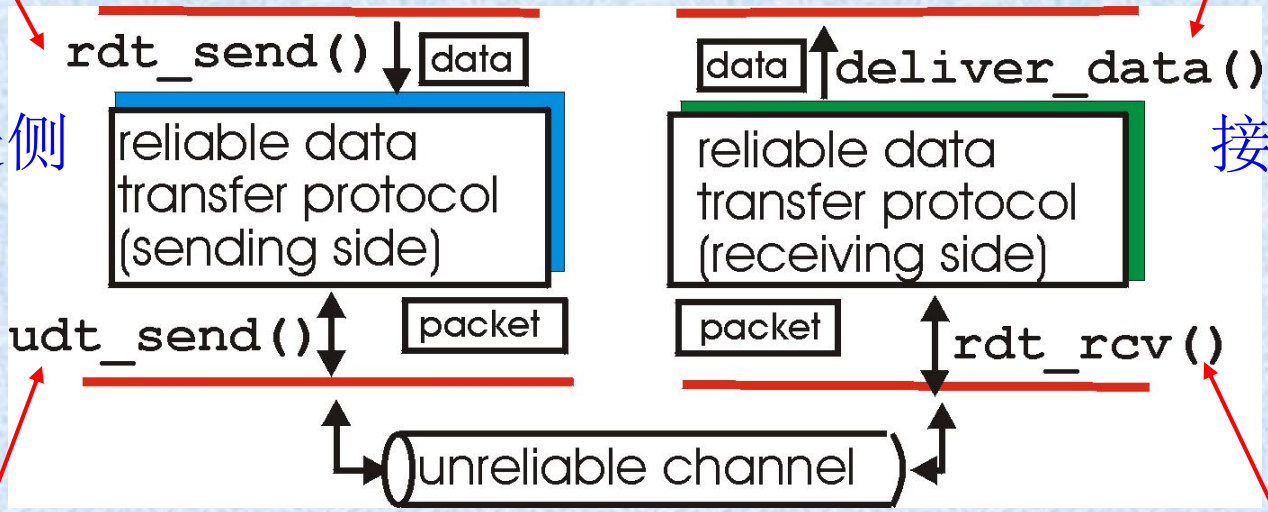


- 不可靠信道的特点决定了可靠数据传输协议 (rdt) 的复杂性

可靠数据传输: 基本概念

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



udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

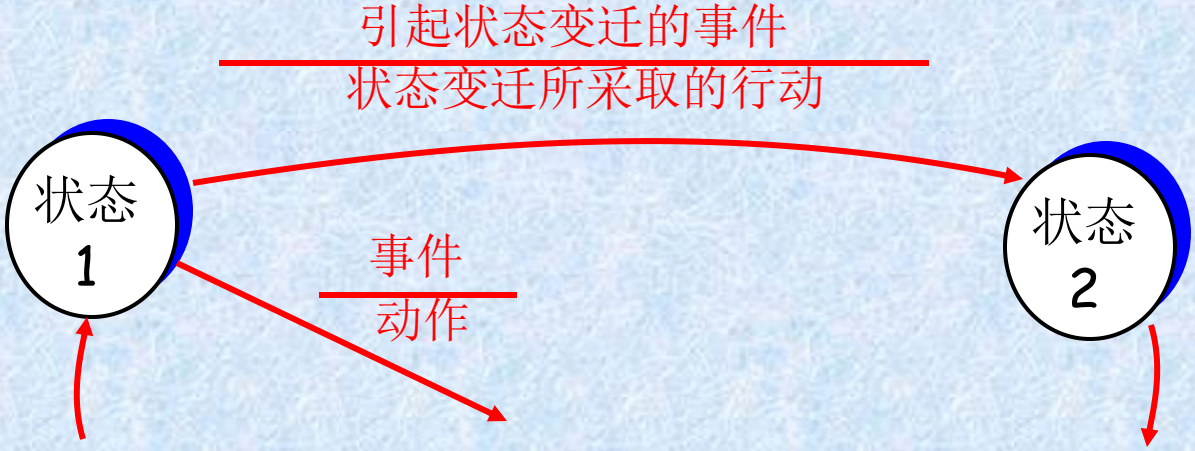
rdt_rcv(): called when packet arrives on rcv-side of channel

可靠数据传输: 基本概念

我们将:

- 逐渐递增地研究可靠数据传输协议 (rdt) 的发送方和接收方：
  - 仅考虑单向数据传输
  - 但控制信息将在两个方向流动！
- 使用有限状态机 (FSM)来定义发送方和接收方

状态: 当位于这个“状态时”，下个状态惟一地由下个事件决定



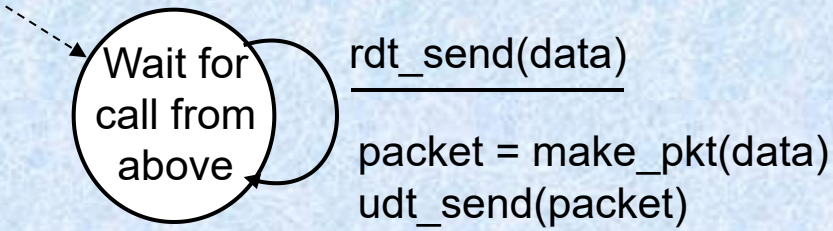
第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

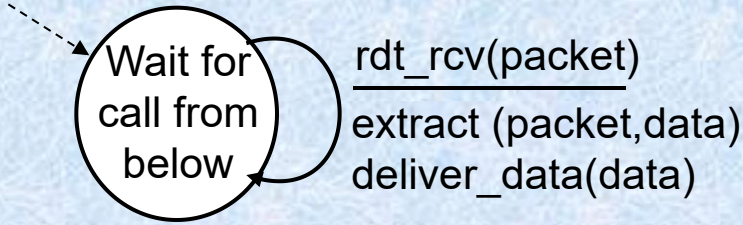
Rdt1.0: 经可靠信道的可靠传输

- 底层信道非常可靠
  - 无比特差错
  - 无分组丢失→分组按序到达
- 发送方、接收方的单独FSM:
  - 发送方将数据发向底层信道
  - 接收方从底层信道读取数据

完美信道不存在！



发送方



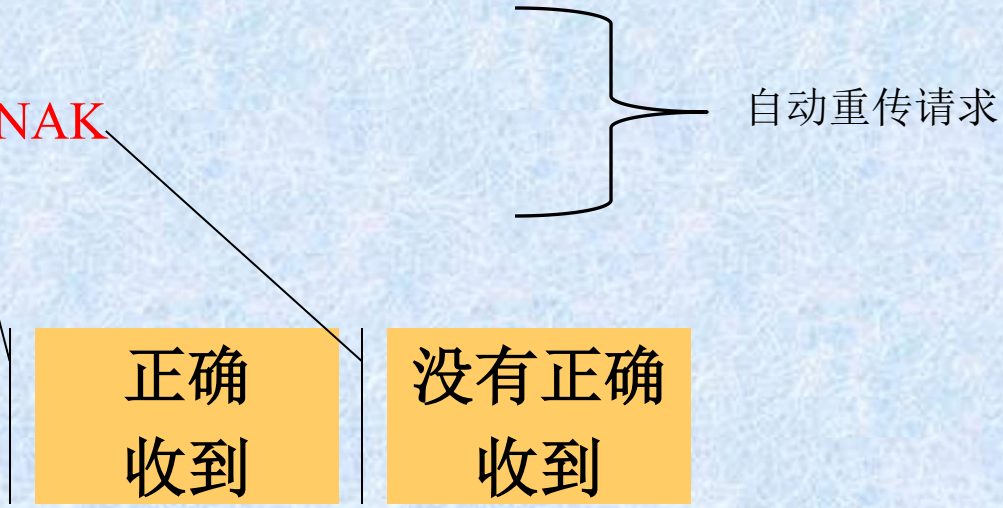
接收方

第3章 要点

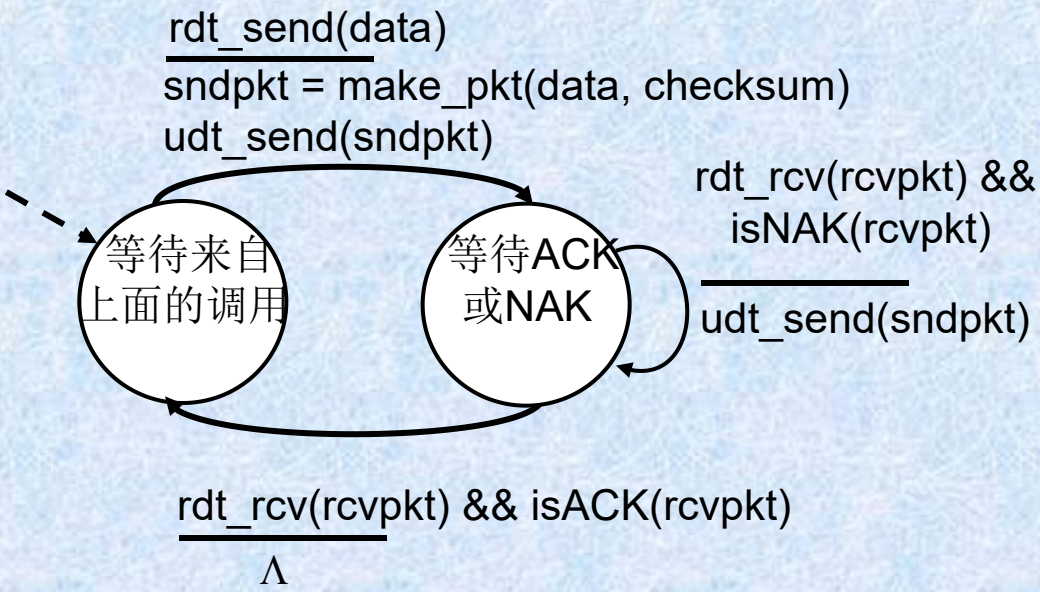
- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

Rdt2.0: 具有比特差错的信道

- 具有比特差错的底层信道
  - 有比特差错
  - 无分组丢失→分组按序到达
- 数据出错后处理方式
  - 检错重传
- rdt2.0新增加机制（与rdt1.0比较）
  - 检错
  - 反馈：ACK, NAK
  - 重传

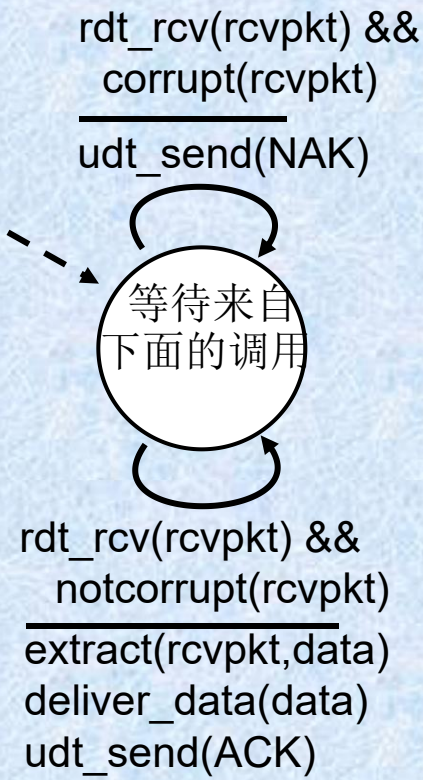


rdt2.0: FSM描述



发送方

接收方



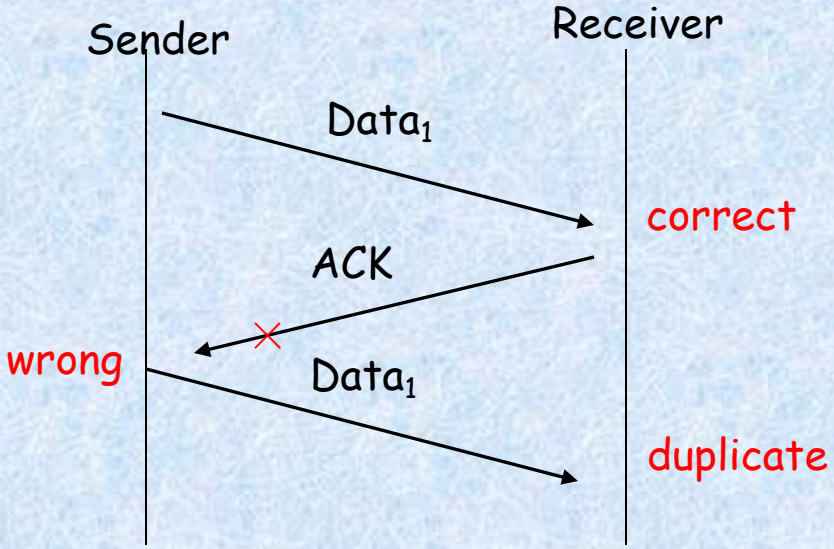
停等协议

发送方发出1个分组，等待接收方响应后再继续发送。（类似rdt2.0）

rdt2.0有重大的缺陷!

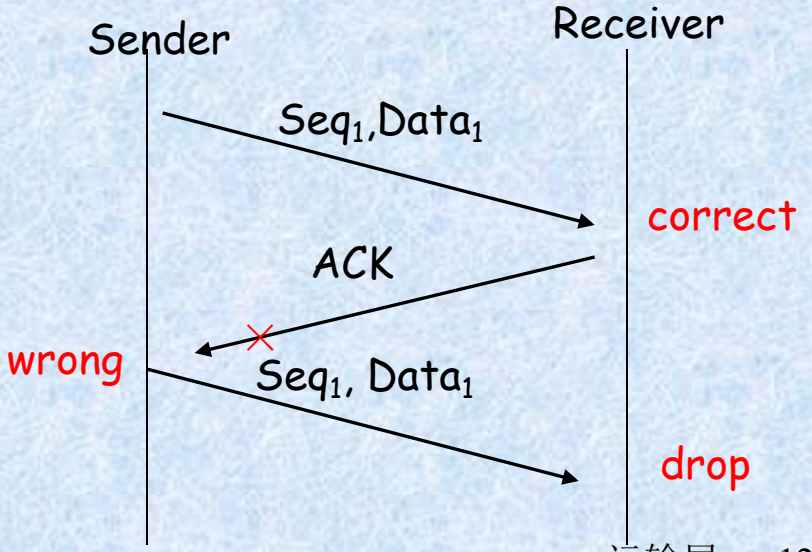
如果ACK/NAK受损，将会出现何种情况？

- 发送方不知道在接收方会发生什么情况！
- 不能只是重传：可能导致重复（ duplicate ）

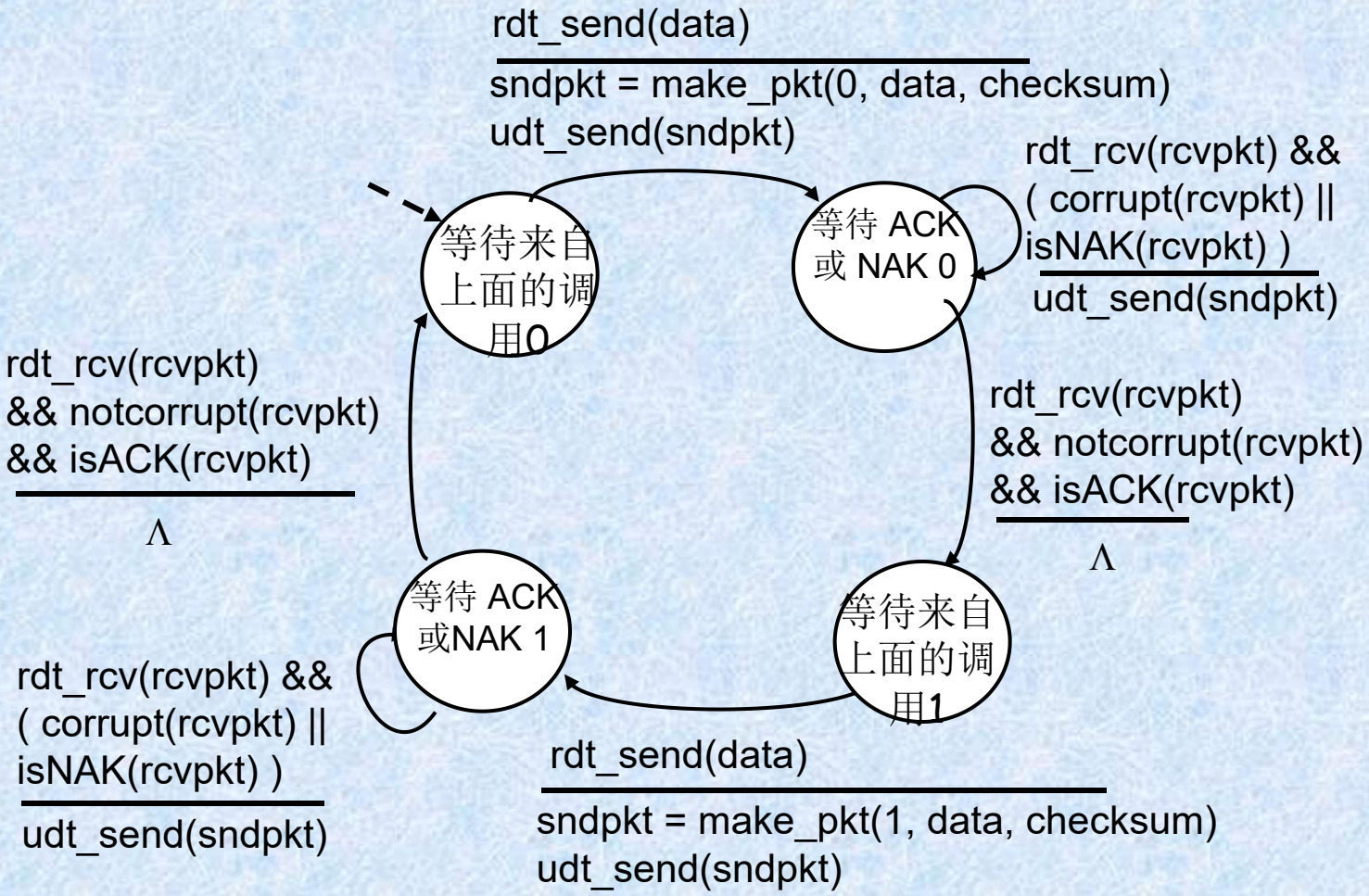


处理重复（序号机制）：

- 发送方对每个分组增加*序列号*
- 如果ACK/NAK受损，发送方重传当前的分组
- 接收方丢弃(不再向上交付)重复的分组



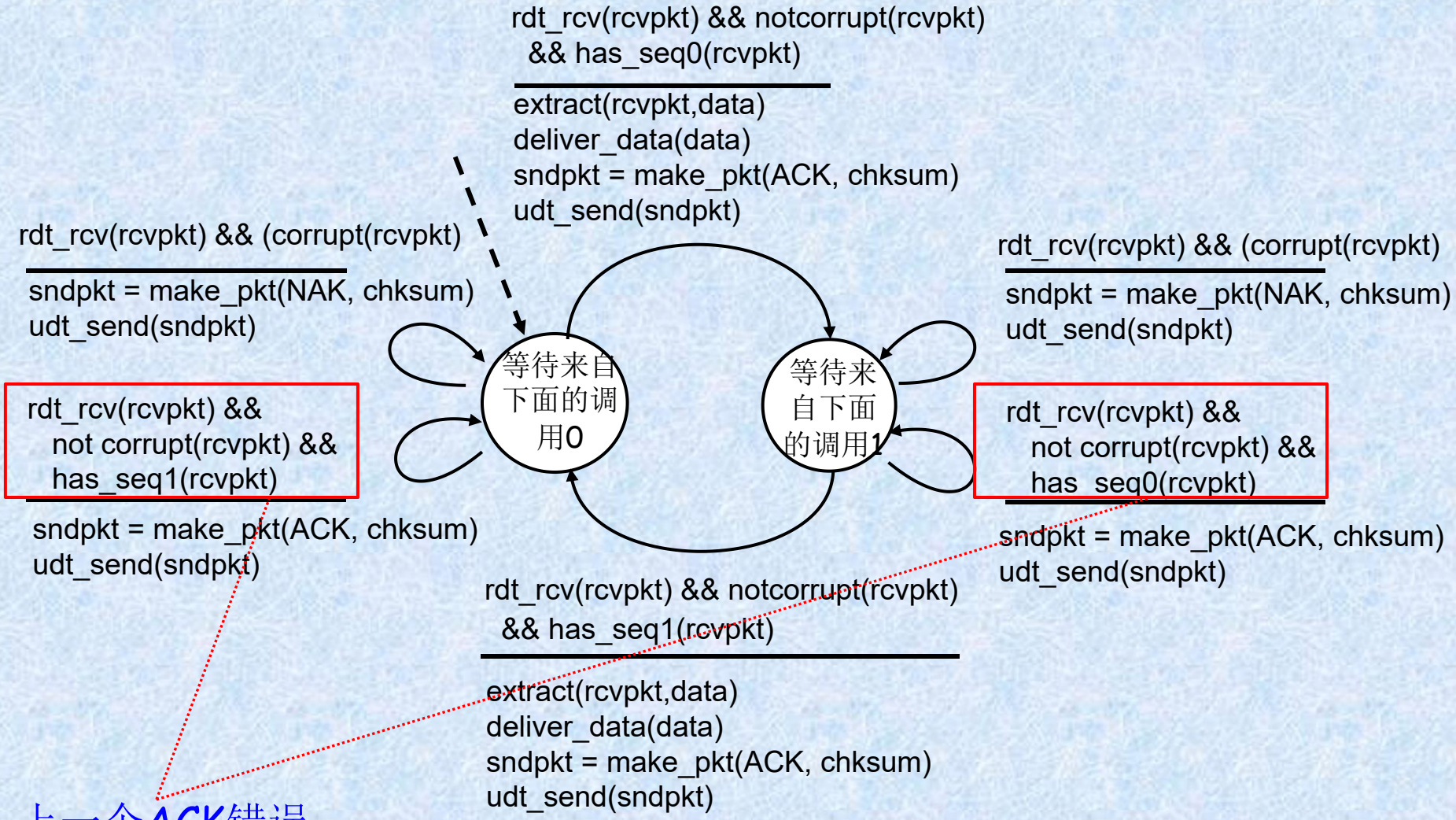
rdt2.1: 发送方, 处理受损的ACK/NAK



停等协议：采用1比特序号

分组不会丢失：ACK和NAK无需指定确认分组序号

rdt2.1: 接收方,处理受损的ACK/NAK



上一个ACK错误
发送方重发数据

rdt2.1: 讨论

发送方:

- 序号seq # 加入分组中
- 两个序号seq # (0,1) 将够用. (为什么?)
- 必须检查是否收到的ACK/NAK受损
- 状态增加一倍
  - 状态必须“记住”是否“当前的”分组具有0或1序号

思考：P6

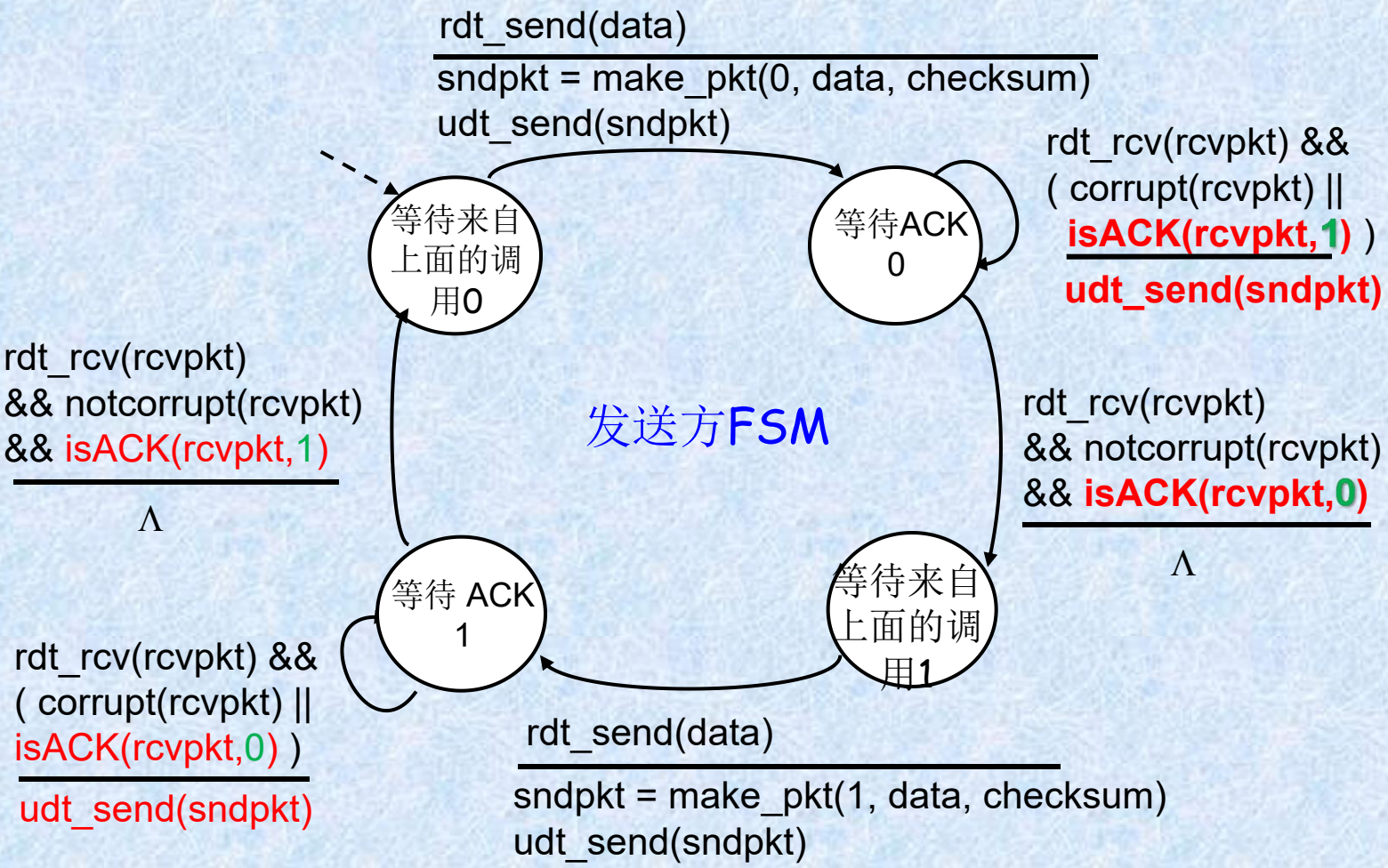
接收方:

- 必须检查是否接收到的分组是冗余的
  - 接收分组的序号是否是所期待的分组序号seq #
- 注意: 接收方无法知道是否它的最后的ACK/NAK在发送方已经正确接收

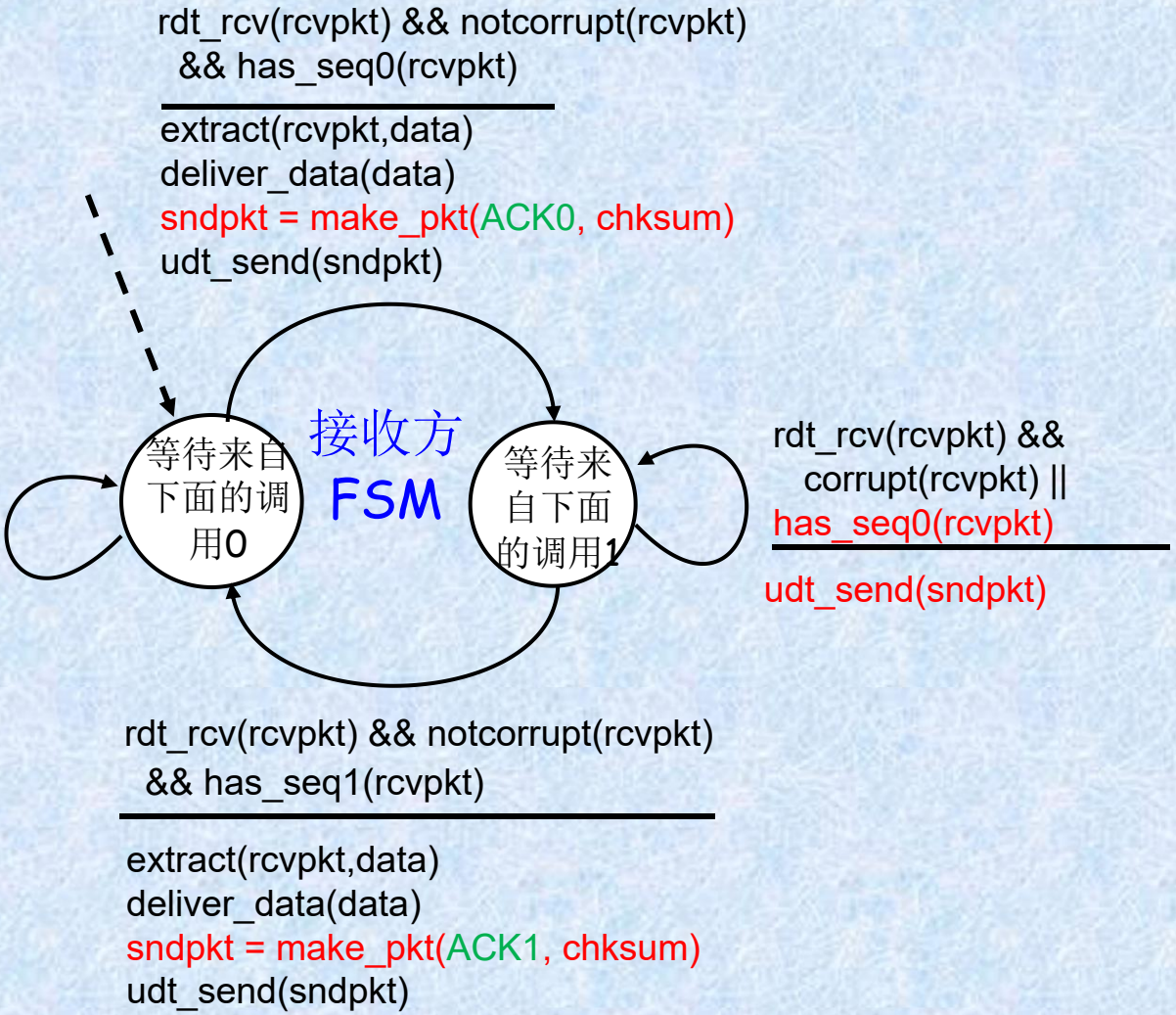
rdt2.2: 一种无NAK的协议

- 与rdt2.1一样的功能，仅使用ACK
- 代替NAK,接收方对最后正确接收的分组发送ACK
  - 接收方必须明确地包括被确认分组的序号
- 在发送方冗余的ACK导致如同NAK相同的动作：*重传当前分组*

rdt2.2: 发送方片段


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
等待来自上面的调用0
等待ACK 0
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
Λ
发送方FSM
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
Λ
等待 ACK 1
等待来自上面的调用1
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
udt_send(sndpkt)
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

rdt2.2: 接收方片段


rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK0, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) &&
corrupt(rcvpkt) ||
has_seq1(rcvpkt)
udt_send(sndpkt)
等待来自下面的调用0
接收方
FSM
等待来自下面的调用1
rdt_rcv(rcvpkt) &&
corrupt(rcvpkt) ||
has_seq0(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)

第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

rdt3.0: 具有差错和丢包的信道

新假设: 下面的信道也能丢失分组(数据或ACK)

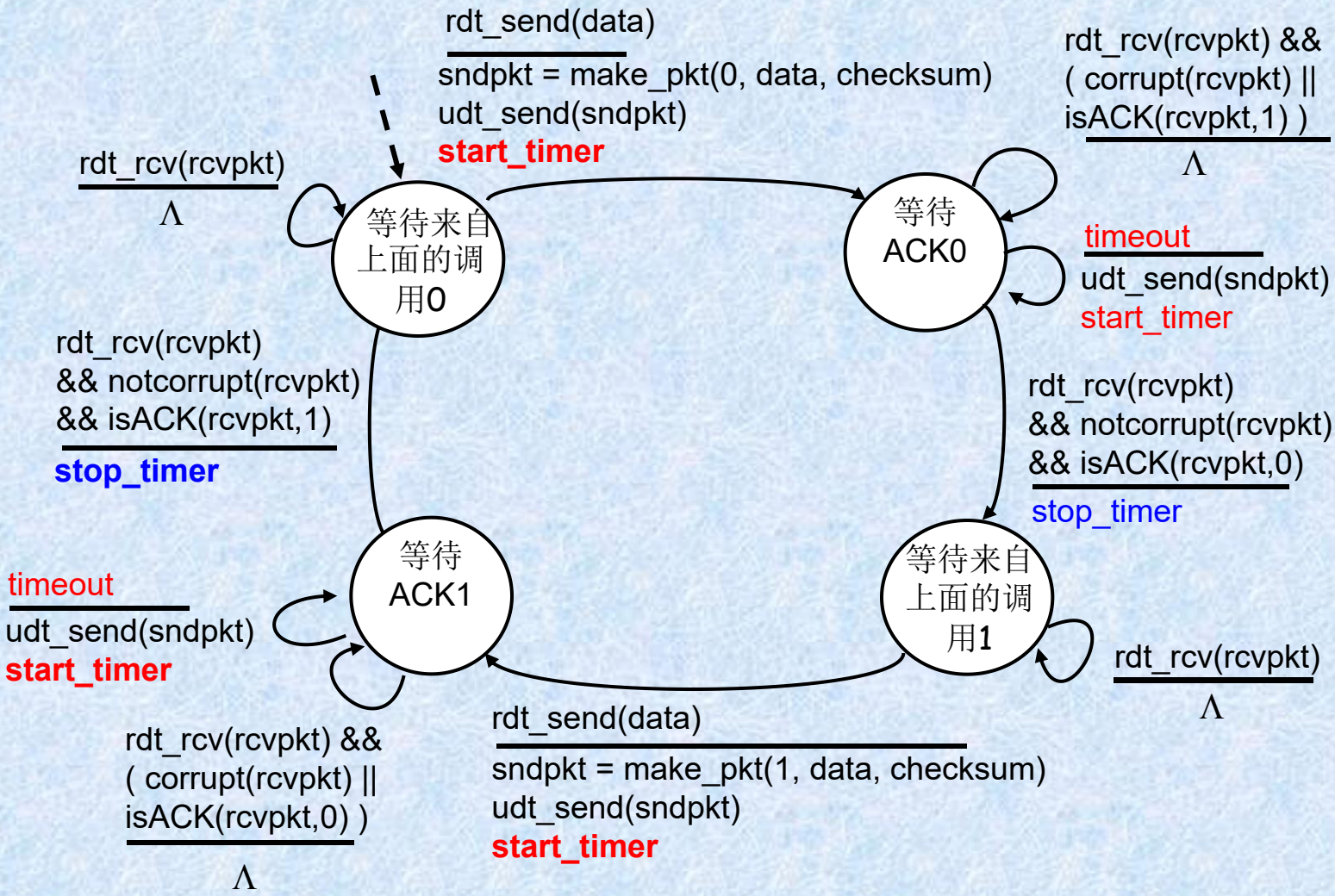
- 检验和、序号、重传将是有帮助的，但不充分

方法: 发送方等待ACK一段“合理的”时间

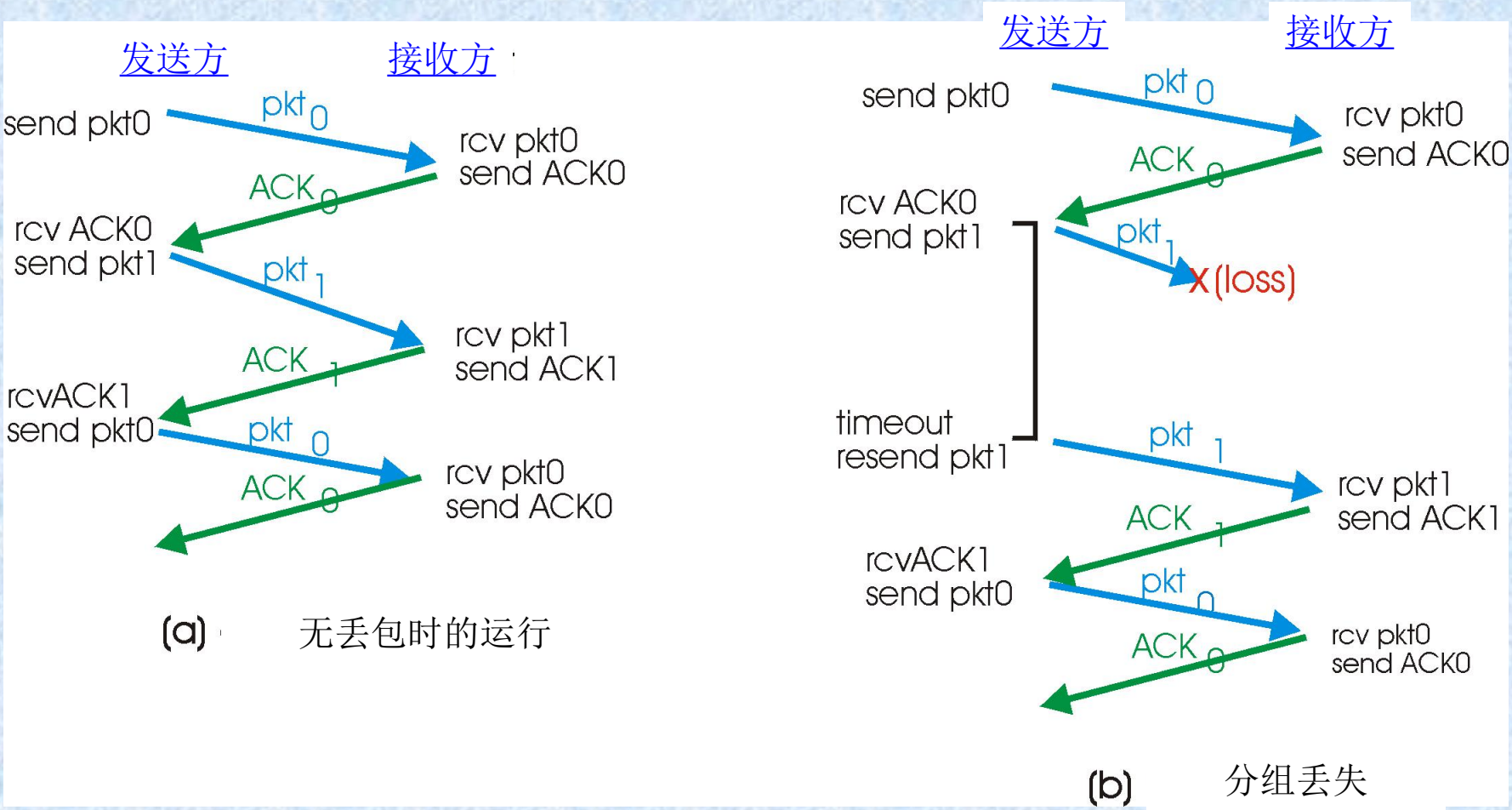
- 如在这段时间没有收到ACK则重传
- 如果分组(或ACK)只是延迟(没有丢失):
  - 重传将是冗余的，但序号的使用已经处理了该情况
  - 接收方必须定义被确认的分组序号
- 需要倒计时定时器

- **每次发送1个分组(包括初始分组和重传分组)时，便启动一个定时器。**
- **响应定时器中断。**
- **终止定时器。**

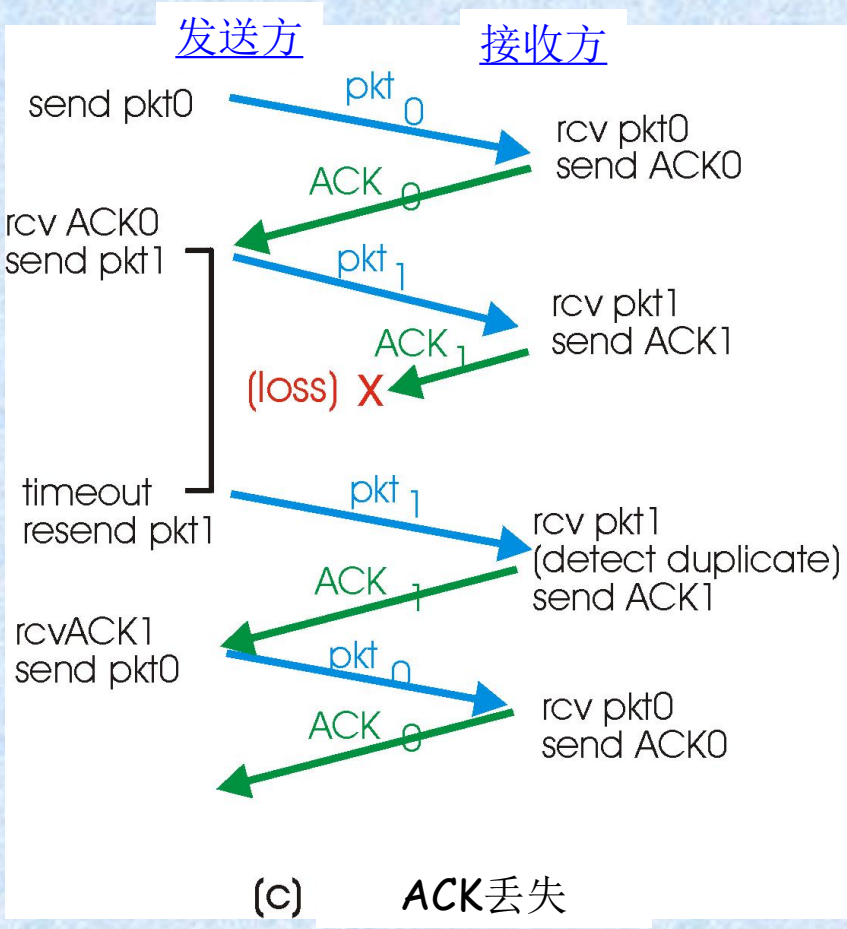
rdt3.0发送方


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
等待来自上面的调用0
等待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
等待ACK1
等待来自上面的调用1
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

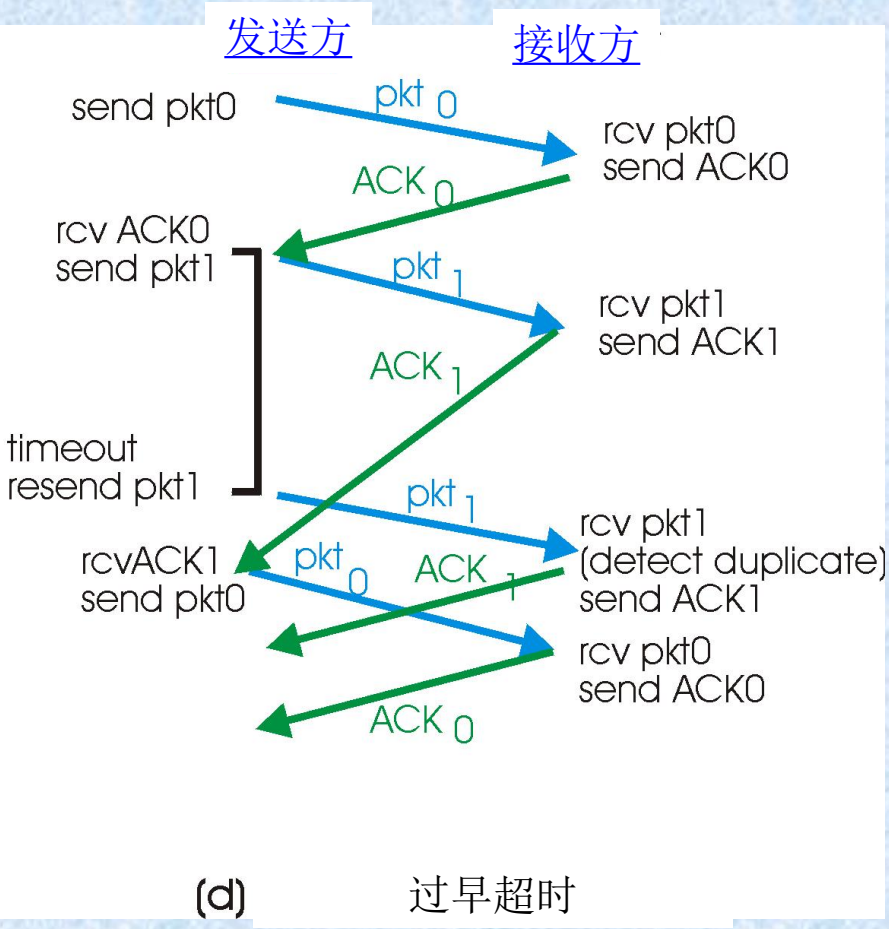
rdt3.0 运行情况


发送方
接收方
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) 无丢包时的运行
发送方
接收方
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) 分组丢失

rdt3.0运行情况



(c) ACK丢失



(d) 过早超时

rdt3.0的性能

- rdt3.0能够工作，但性能不太好
- 例子: 1 Gbps（10^9 bit/s）链路, 30ms往返传播时延。发送1KB分组所需时间:

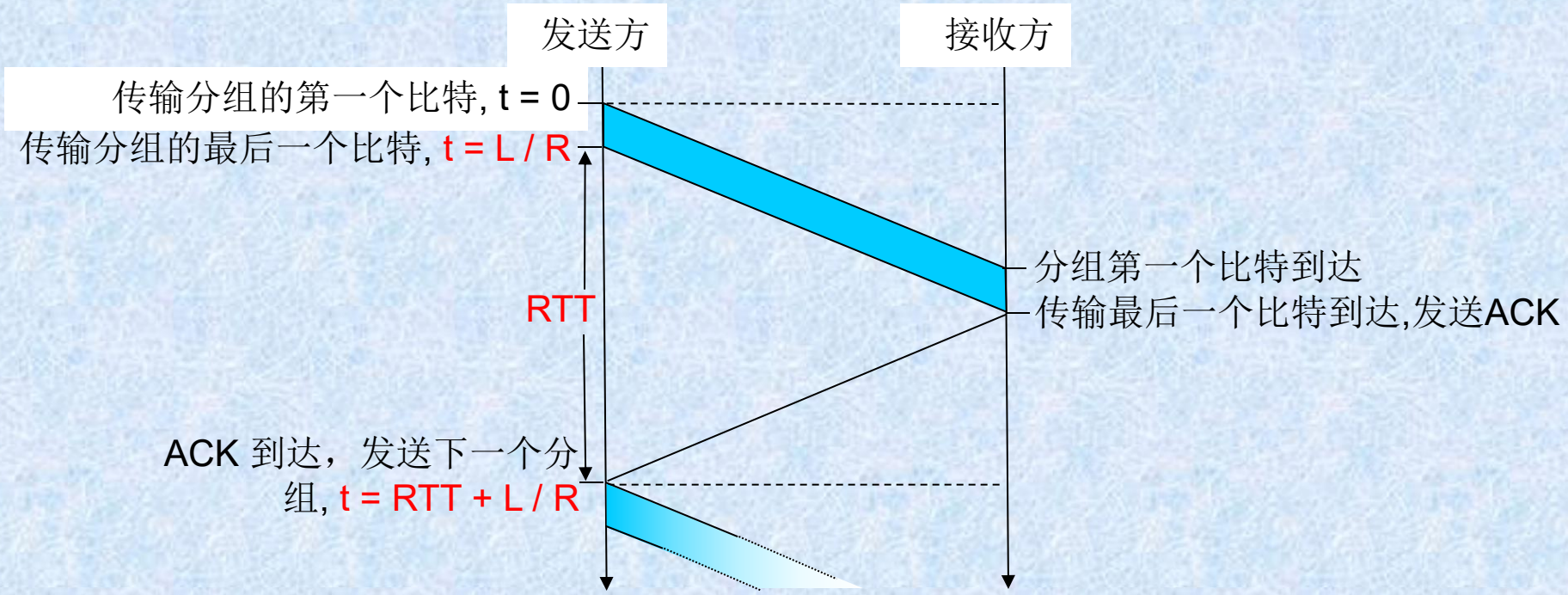
$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8kb/pkt}{10^9 \text{ b/sec}} = 8 \text{ microsec}$$

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

  - U_{sender}: 利用率， 发送方用于发送时间的比率
  - 每30 ms发送1KB 分组 -> 经1 Gbps 链路有33KB/sec 吞吐量
  - 网络协议限制了物理资源的使用!

$0.00027*10^9=270kb=33KB$

rdt3.0: 停等协议的运行

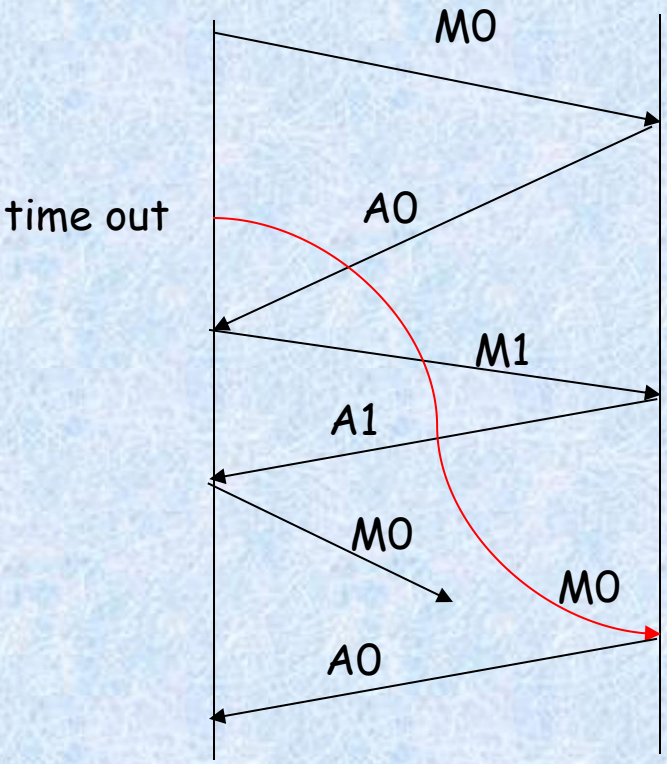


$$U_{sender} = \frac{L/R}{RTT+L/R} = \frac{0.008}{30.008} = 0.00027$$

作业：P8

rdt3.0: 停等协议的运行

- P13：如果发送方网络和接收方的网络能够对报文重排，那么比特交替协议将不能正确工作。



old version of M0 accepted!

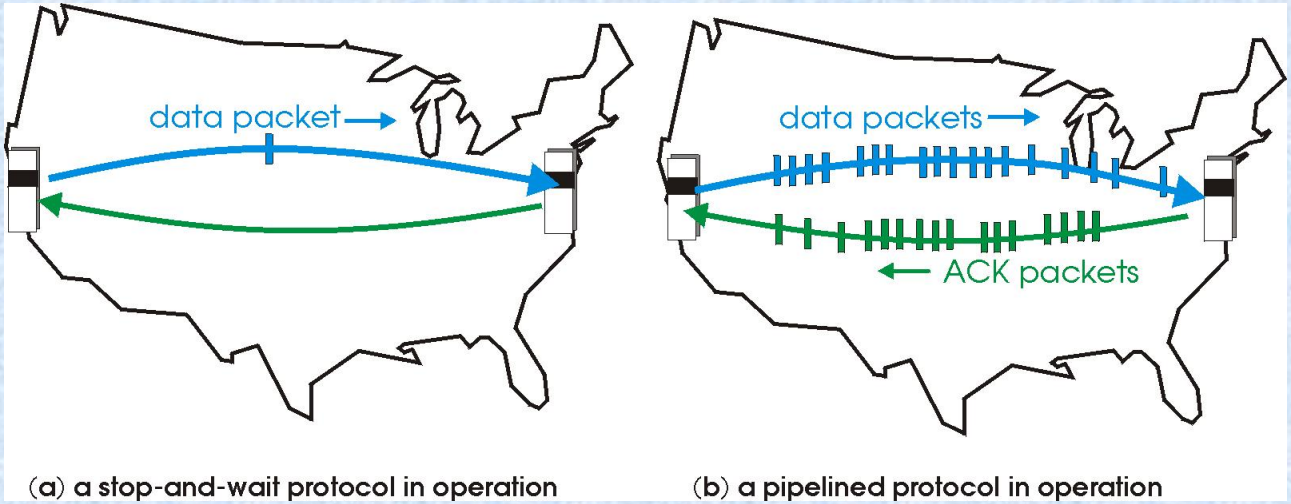
第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

流水线协议

流水线: 发送方允许发送多个、“传输中的”,还没有应答的报文段

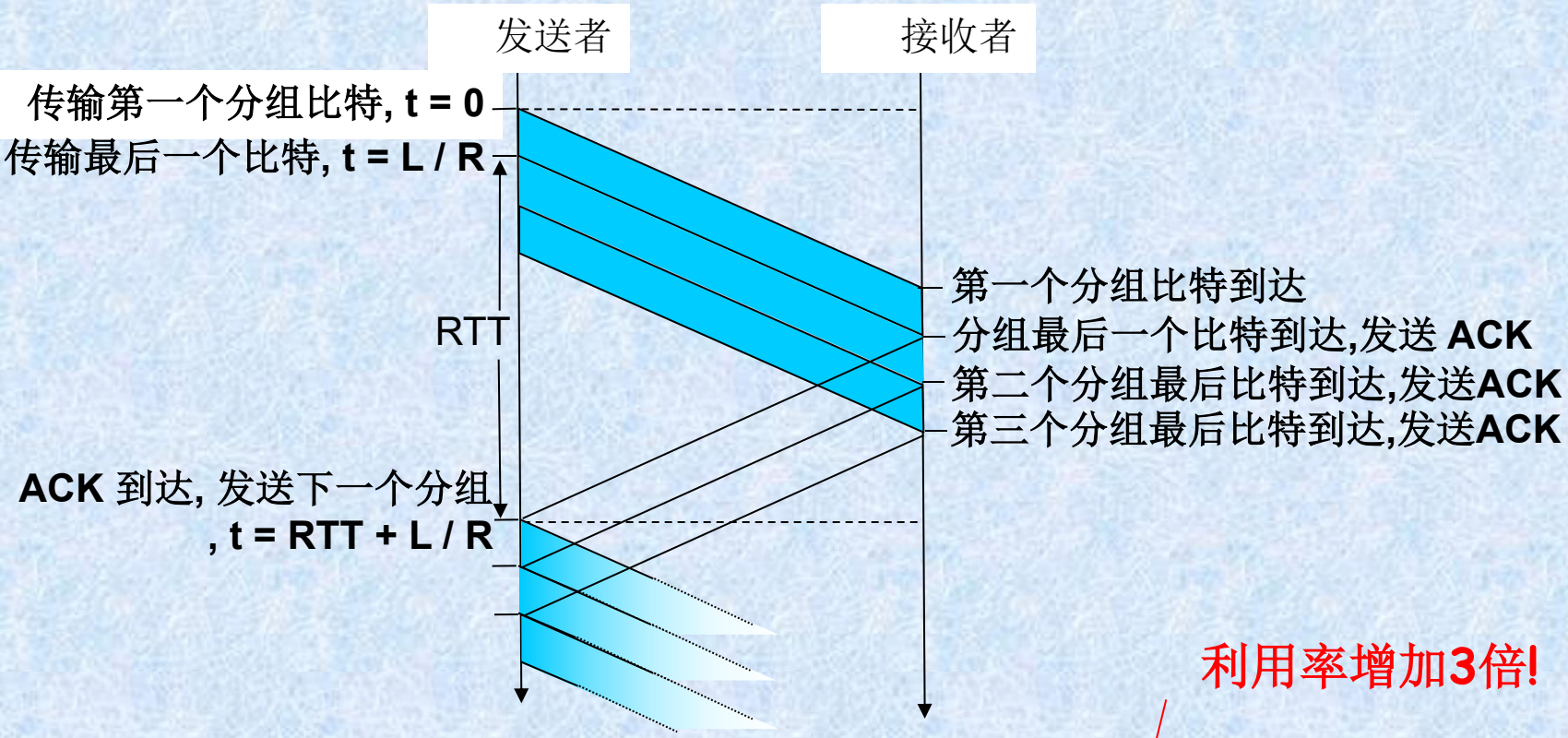
- 序号的范围必须增加
- 发送方和/或接收方设有缓冲



- 流水线协议的两种形式:

回退N步（Go-Back-N）, 选择性重传（Selective-Repeat）,

流水线协议: 增加利用率



利用率增加3倍!

$$U_{sender} = \frac{3*L/R}{RTT+L/R} = \frac{0.024}{30.008} = 0.0008$$

滑动窗口协议

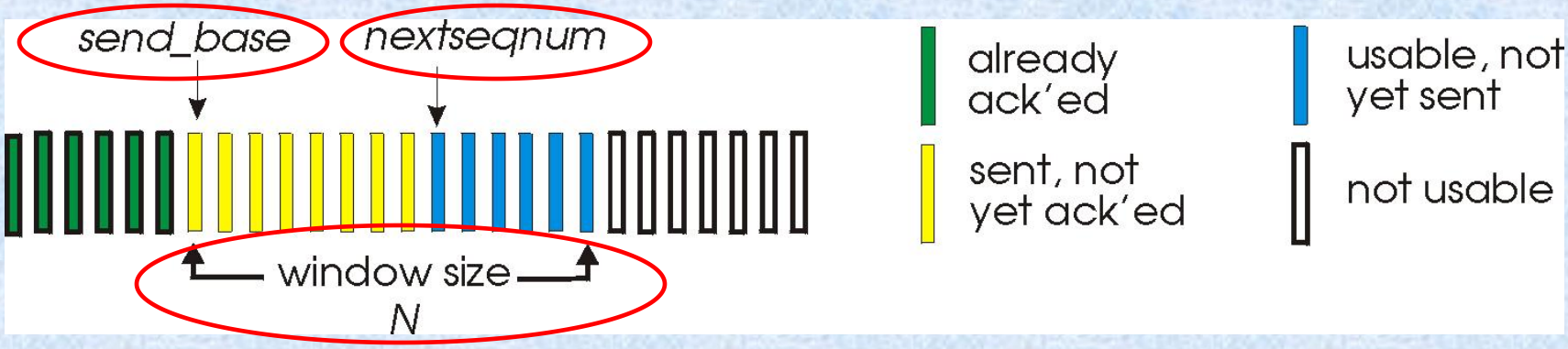
Go-Back-N和选择重传都是滑动窗口协议

- 发送方和接收方都具有一定容量的缓冲区（即窗口），允许发送端连续发送多个分组而不需要等待应答
- **发送窗口**就是发送端允许连续发送的分组的序号表，发送端可以不等待应答而连续发送的最大分组数称为发送窗口的尺寸
- **接收窗口**是接收方允许接收的分组的序号表，凡落在接收窗口内的分组，接收方都必须处理，落在接收窗口外的分组被丢弃。接收方每次允许接收的分组数称为接收窗口的尺寸

Go-Back-N

发送方：

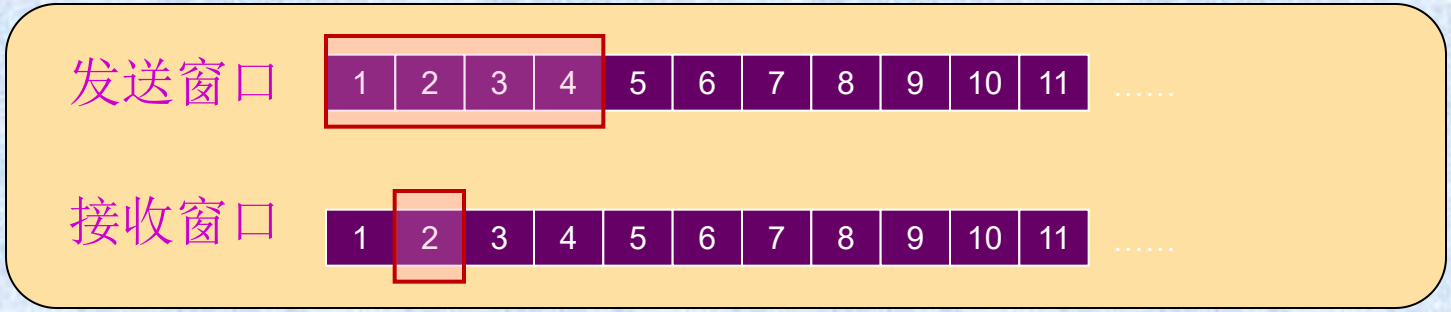
- “窗口”大小为N, 允许发送N个连续的没有应答分组
  - 基序号：最小未确认的分组序号
  - 下一个序号：最小未使用的分组序号



  - 分组的序号字段k比特，序号范围[0,2^k-1]
  - TCP有一个32bit的序号字段，按字节计数

Go-Back-N

- 发送窗口尺寸为N；接收窗口尺寸为1。

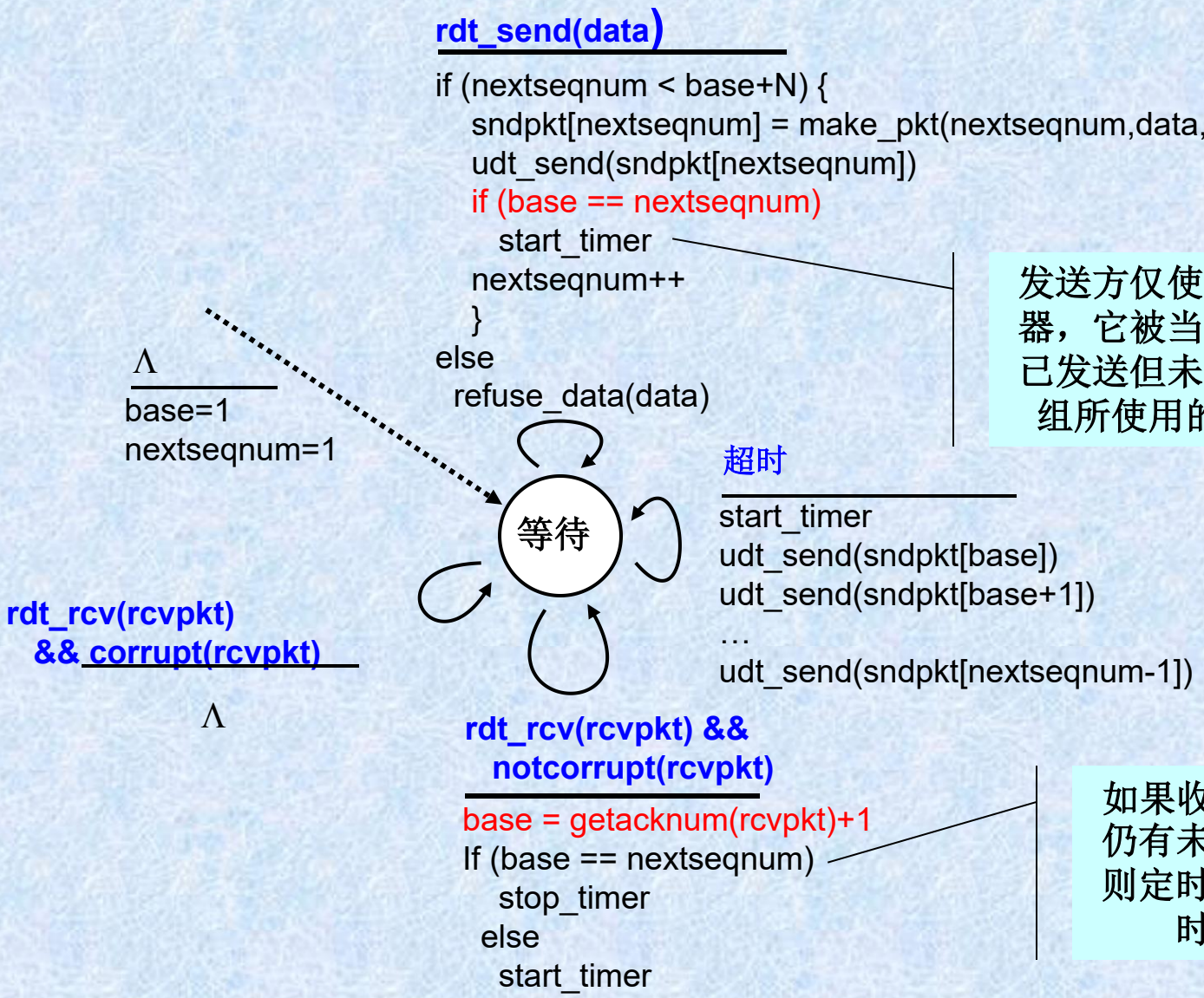


- 简单来说：位于发送窗口内的分组才允许被发送，位于接收窗口内的分组才能被接收，关键是窗口如何滑动。

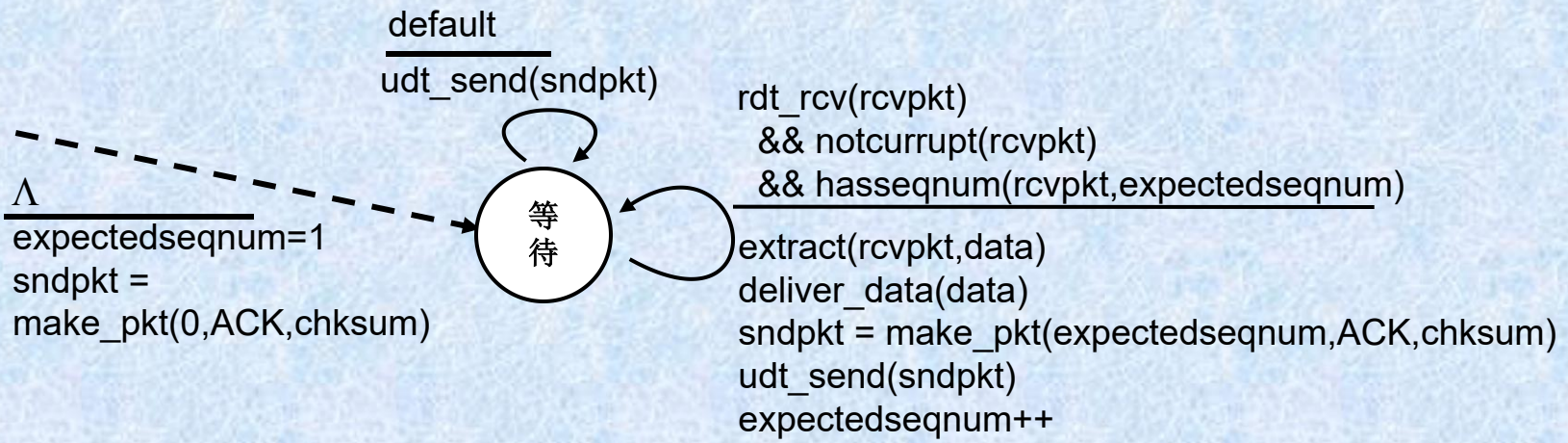
特征：累计ACK，全部重传

- ACK(n): 确认所有的（包括序号n）的分组 - “累计ACK"
- 若超时，重传所有已发送但未被确认的分组

GBN: 发送方扩展的 FSM

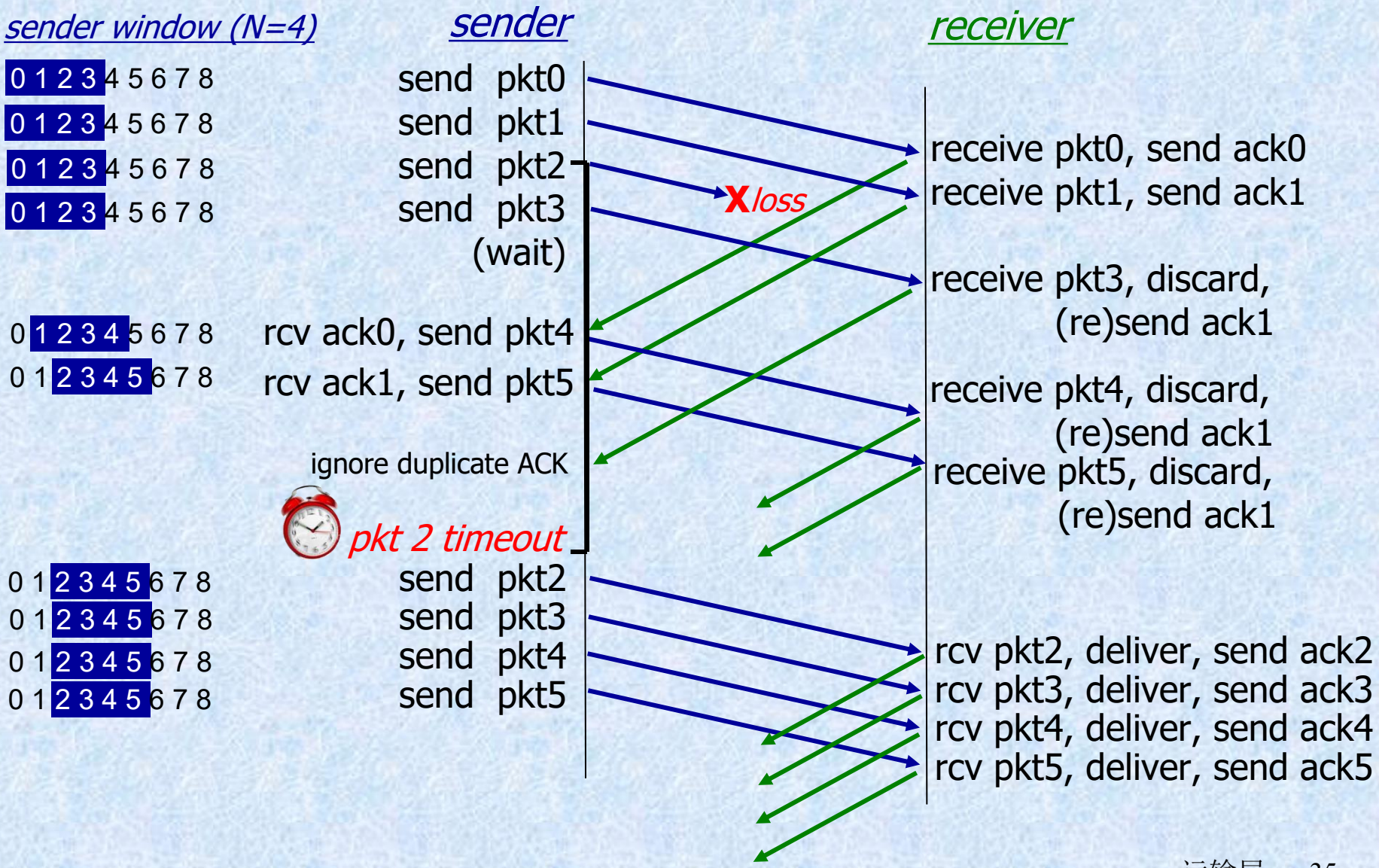


GBN: 接收方扩展 FSM



- 累积确认： 对正确接收的分组总是发送具有最高按序序号的ACK
  - 可能产生冗余的ACKs
  - 仅仅需要记住期望的序号值（**expectedseqnum**）
- 对失序的分组:
  - 丢弃 (不缓存) -> 没有接收缓冲区!
  - 重新确认具有按序的分组

Go-Back-N分组传输（示意）

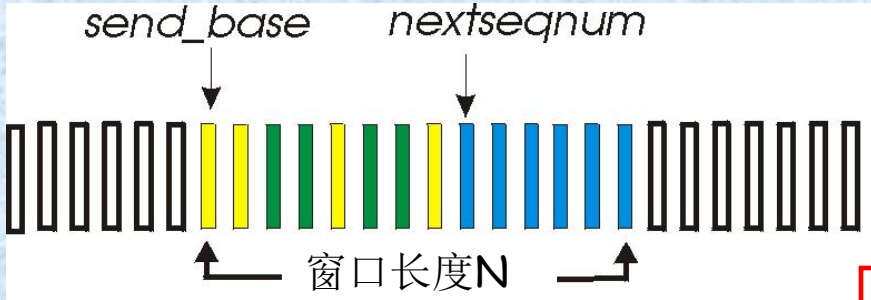


Go-Back-N

- 理解累计ACK和回退N个重传
- 发送方
  - 发送窗口滑动的条件：收到发送窗口内分组序号的确认分组
  - 超时重传时，重传所有未被确认的分组
- 接收方
  - 接收窗口滑动的条件：收到期望序号的分组
  - 累计ACK(n)：表明已经正确收到序号n（包括n）以前的分组
  - 对失序分组的处理：丢弃，重发（已按序接收分组的）ACK
- Go-Back-N不足：（效率明显高于停等协议）但仍有不必要重传的问题

GBN协议，发送方已经发送了编号为0~7的分组。当计时器超时时，若发送方只收到0、2、3号帧的确认，则发送方需要重发的帧数是（　）。

选择性重传SR: 发送方有选择性地重传



a. 发送方看到的序号

已经确认

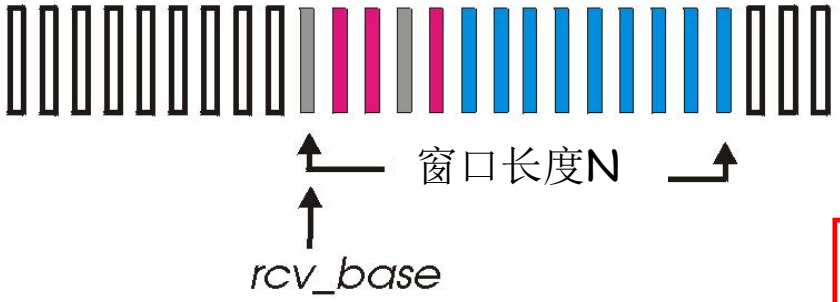
发送，还未确认

可用，还未发送

不可用

发送方只需要重传没有收到ACK的分组

发送方定时器对每个没有确认的分组计时



b. 接收方看到的序号

失序(已缓存)但未被交付（窗口内）

期待，还未收到

可接受（窗口内）

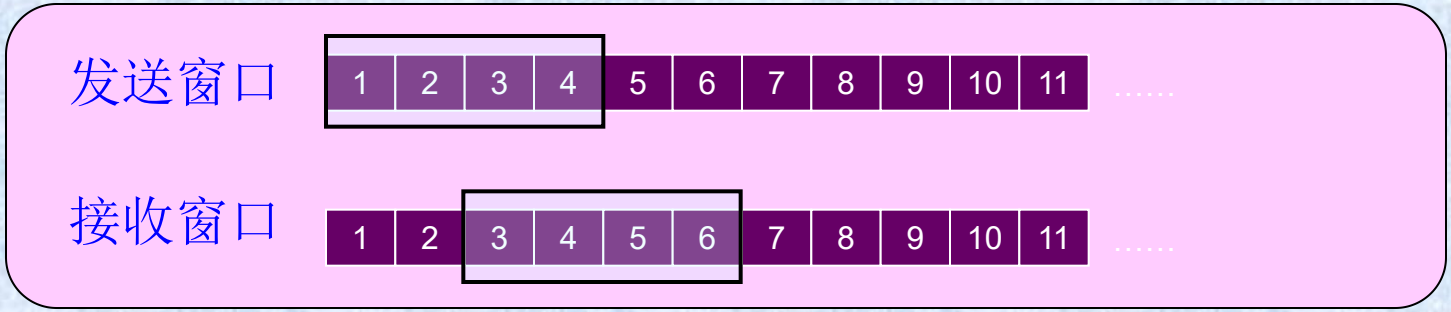
不可用

接收方分别确认正确收到的分组

需要缓存分组，以便最后按序交付给上层

选择重传SR

- 发送窗口尺寸为N；接收窗口尺寸为N。



特征：独立ACK，重传单个分组

- 独立ACK: 对每个分组使用单独的确认
- 需N个定时器，若某个分组超时，则重传该分组
- 接收窗口为N，对非按序到达的分组进行缓存

选择性重传

发送方

上层传来数据：

- 如果窗口中下一个序号可用，发送报文段

timeout(n):

- 重传分组n, 重启其计时器

ACK(n) 在 [sendbase,sendbase+N-1]:

- 标记分组 n 已经收到
- 如果n 是最小未收到应答的分组，向前滑动窗口base指针到下一个未确认序号

接收方

分组n在 [rcvbase, rcvbase+N-1]

- 发送 ACK(n)
- 失序: 缓存
- 按序: 交付 (也交付所有缓存的按序分组),向前滑动窗口到下一个未收到报文段的序号

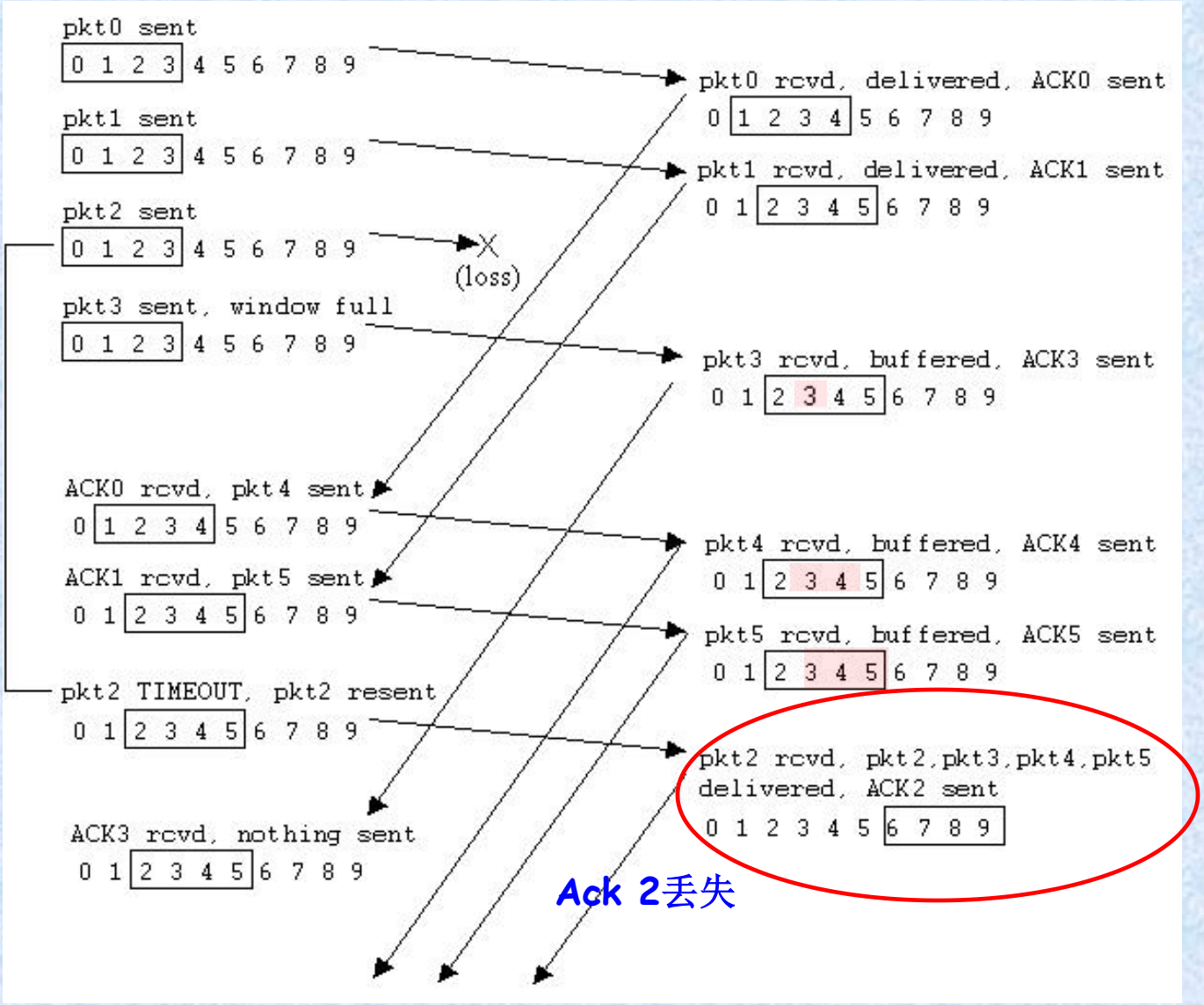
分组n在[rcvbase-N,rcvbase-1]

- ACK(n)

其他:

- 忽略

选择重传的操作


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
X
(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
Ack 2丢失

选择重传的理解

- 理解单独ACK和单个分组重传
- 发送方
  - 发送窗口滑动的条件：收到最小未收到应答的分组的确认
  - 超时重传时，仅重传超时的单个分组
- 接收方
  - 接收窗口滑动的条件：收到最低位置的分组
  - 单独ACK
  - 对失序分组的处理：接收窗口内缓存
    接收窗口外丢弃

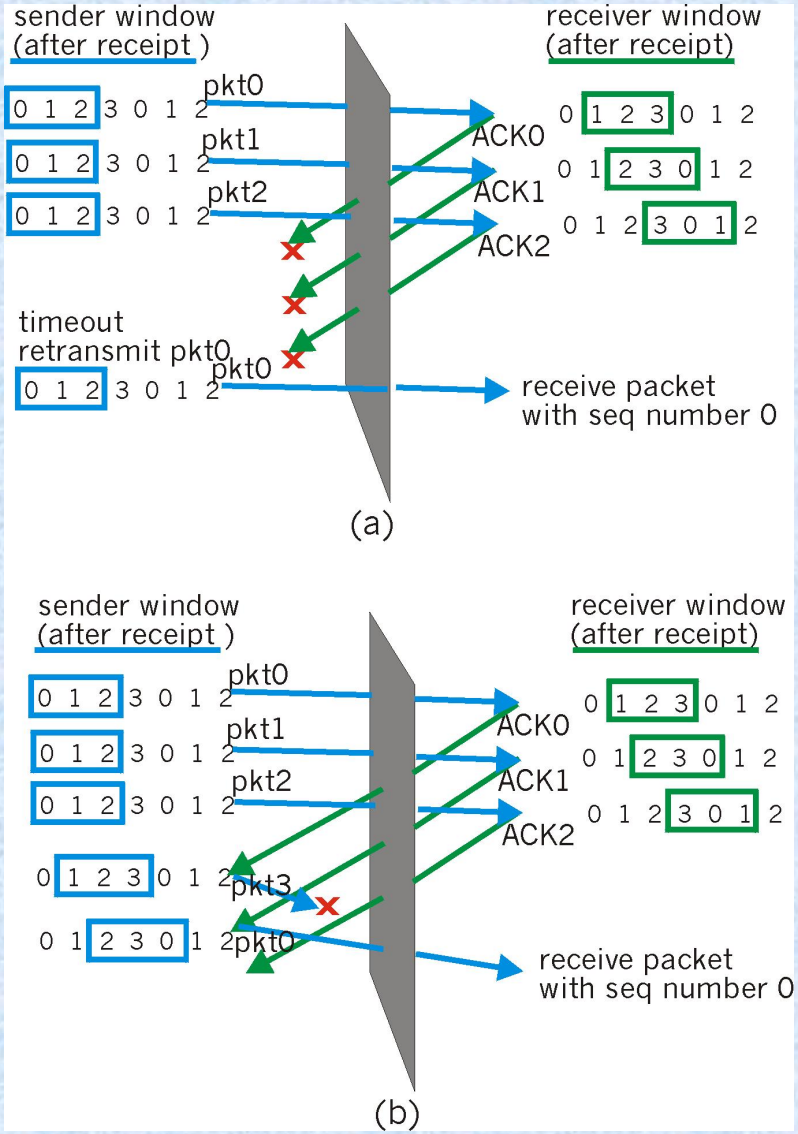
即使分组之前被确认过，仍然发送ACK

选择重传: 接受窗口太大的困境

难以区分新分组和重传分组

例子:

- 序号: 0, 1, 2, 3
- 窗口长度 = 3
- 接收方：在(a)和(b)两种情况下接收方无法发现差别!
- 在 (a)中不正确地将冗余的当为新的，而在(b)中没有办法区分新的和冗余的



问题: 序号长度与窗口长度有什么关 系?

回答：对于SR协议，发送窗口+接收窗口长度之和小于等于序号空间

思考题：P22, P23, P24