4. 可靠数据传输（rdt）

4.1 简介

rdt在应用层、传输层和数据链路层都很重要

是网络Top 10问题之一

信道的不可靠特点决定了 可靠数据传输协议(rdt) 的复杂性

例：

应用层 sending process receiver process

------ DATA DATA rdt_send(data) deliver_data()

传输层 reliable channel | | reliable data transfer protocol(sending side) rdt(receiving side)

| |

udt_send() ----------packet----------packet---------- udt_rcv()

unrealiable channel

<--------provided server----------> <--------------------------service implementation--------------------------->

信道的不可靠特点决定了可靠数据传输协议(rdt)的复杂性

rdt在应用层-传输层， 网络层-传输层，传输层-数据链路层 都有

4.2 问题描述

rdt_send():将上层（如应用层）调用，以将数据交付给下方的发送实体

udt_send():被rdt调用，用以将分组放到不可靠的信道上传输到接收方

udt_rcv():当分组通过信道到达接收方时候调用

deliver_data():被rdt调用，将数据交付给上层

rdt_send，deliver_data 是本层与上层的接口

udt_send，udt_rcv 是本层协议实体跟下层的原语

<!-- 我们将：

渐进式地开发可靠数据传输协议（rdt）的发送方和接收方

只考虑单向数据传输

但控制信息是双向流动的

双向的数据传输问题实际上是2个单向数据传输问题的综合

使用有线状态机（FSM）来描述发送方和接收方　　引起状态变迁的事件

状态1---------->状态2　　FSM 有限状态机

状态：在该状态时，下一个状态只由下一个时间唯一确定 -->　　状态变迁是采取的动作

Rdt1.0:

在可靠信道上的可靠数据传输:

下层的信道是完全可靠的

没有比特出错

没有分组丢失

发送方和接收方的FSM

rdt_send(data)

发送方　---------->

packet=make_pkt(data)

udt_send(packet)

状态：等待来自上层的调用-->被调用--->等待调用

实际操作：等待----->数据来了--->封装packet---->打走packet----->恢复等待

即，接收 封装 发送

rdt_rcv(packet)

接收方　---------->

extract(packet,data)

deliver_data(data)

实际操作：等待----->数据来了--->解封装packet---->交给上层用户----->恢复等待

即，接收 解封装 交付

Rdt1.0只做了封装解封装

Rdt2.0：

具有比特差错的信道：

下层信道可能会出错：将分组中的比特反转（某一/几位， 1-0 反转）

用校验和来检测比特差错

问题：怎样从差错中恢复

确认ACK：　接收方显式地告诉发送方分组已被正确接收

否定确认NAK：接受方显式地告诉发送方分组已经发生了差错

发送方收到NAK后，发送方重传分组（这里需要发送方第一次发完后做一个副本）

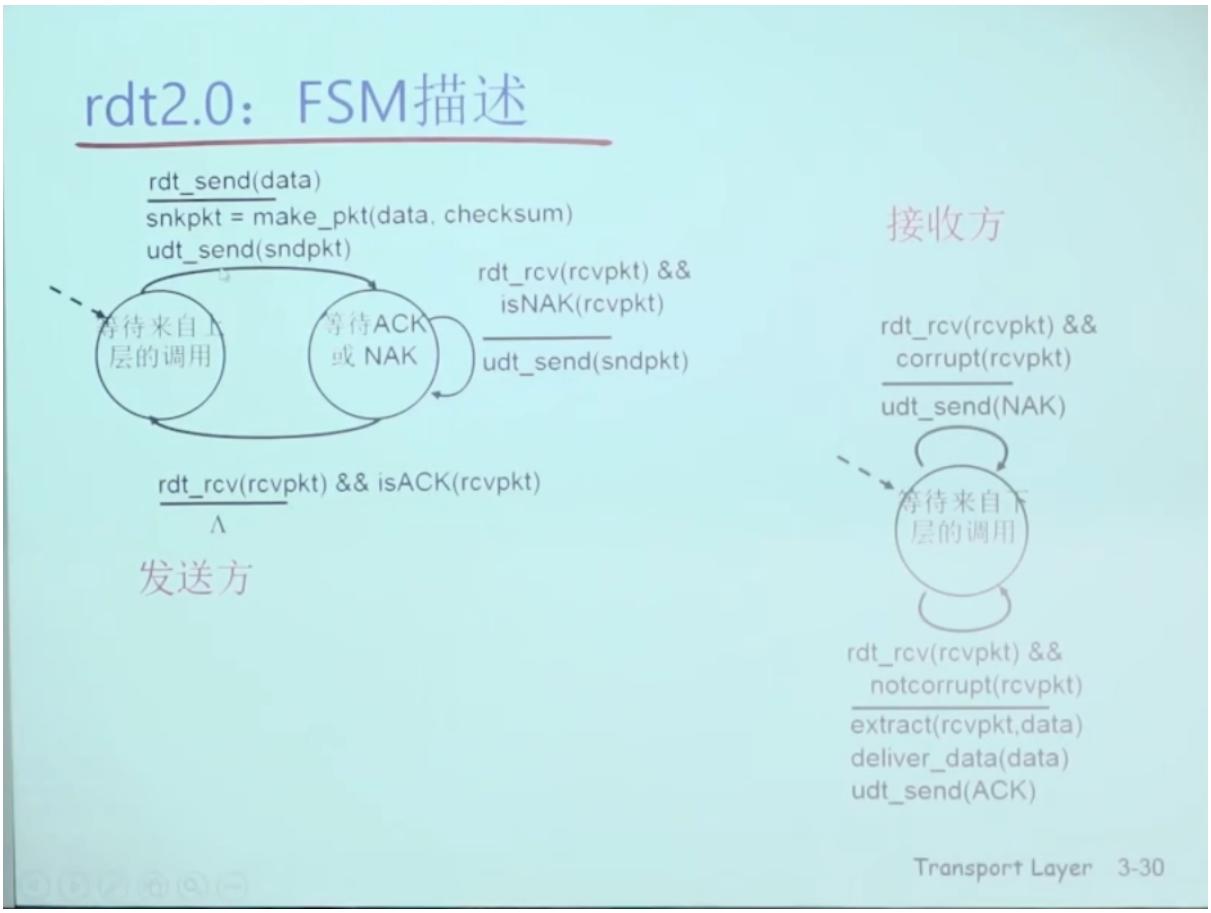
rdt2.0中地新机制：采用差错控制编码进行差错检测----------> D EDC 起作用了

发送方差错控制编码：缓存

接收方使用编码检错

接收方的反馈：控制报文（ACK,NAK）:接收方--->发送方

发送方收到反馈相应的动作（ACK发新的，NAK发老的）



rdt2.0的致命缺陷！->rdt2.1

如果ACK/NAK出错?

发送方不知道接收方发生了什么事

发送方该如何做?

重传-->可能重复

不重传--->可能死锁（或出错）

需要引入新的机制：序号

处理重复：

发送方在每个分组中加入序号

如果ACK/NAK出错，发送方重传当前分组

接收方丢弃（不发给上层）重复分组

停等协议：发送方发送一个分组，然后等待接收方的应答 stop and wish protocol

发送方：

在分组中加入序列号

两个序列号（0，1）就足够

一次只发送一个未经确认的分组

必须检测ACK/NAK是否出错（需要EDC）

状态数变成了两倍

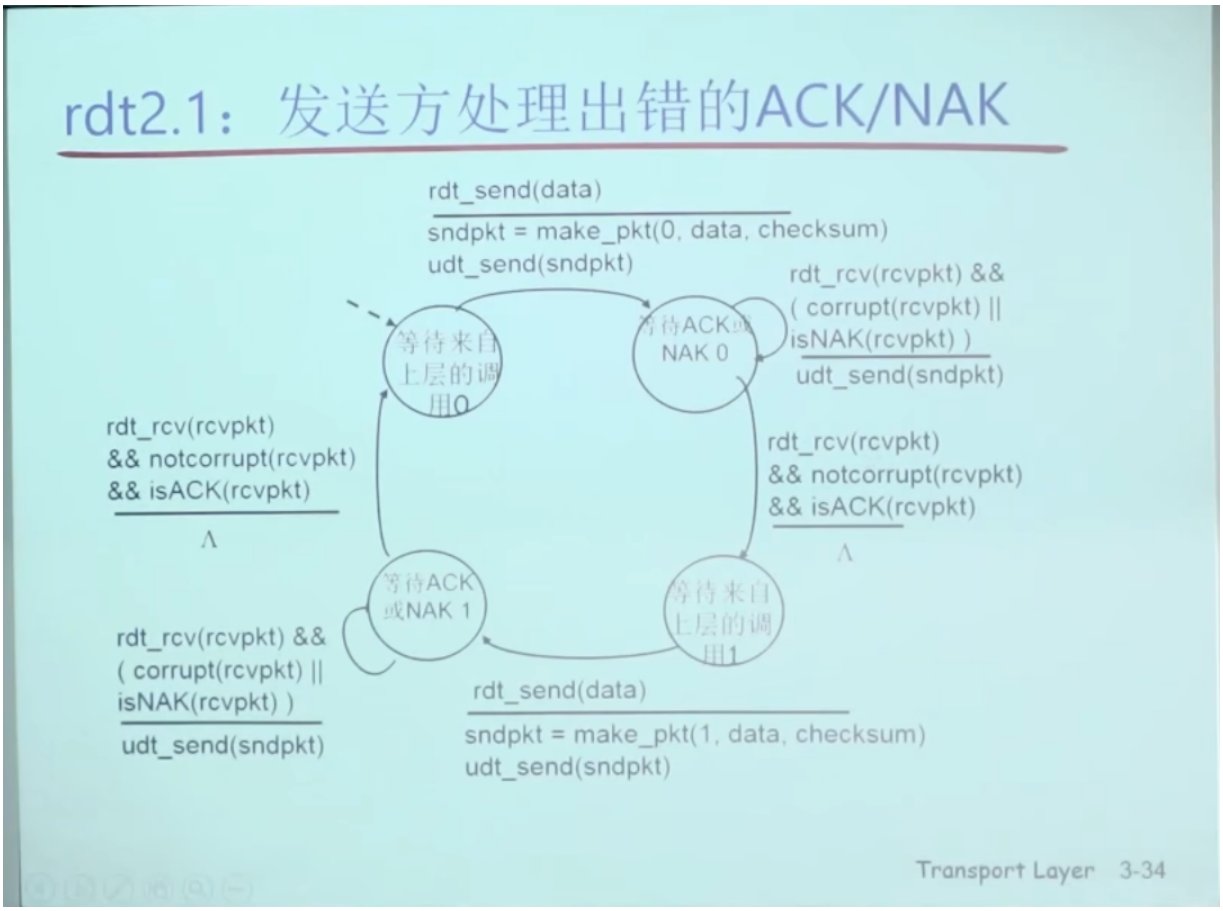
必须记住当前分组的序列号位0还是1

接收方：

必须检测是否接收到的分组是否重复

状态会指示希望收到的分组序号为0还是1

注意：接收方并不知道发送方是否收到了正确的ACK/NAK，没有安排确认的确认



Rdt2.2： NAK free 无NAK的协议

功能同rdt2.1,但是只使用ACK（ACK要编号）

接收方对最后正确接收的分组发ACK,以代替NAK

接收方必须显式地包含被正确接收分组地序号

当收到重复地ACK（如：再次收到ACK0）时，发送方与收到NAK采取相同地动作：重传当前分组

为后面的一次发送多个数据单位做一个准备

一层能够发送多个

每一个的应答都有：ACK,NAK:麻烦

使用对前一个数据单位的ACK,代替本数据单位的NAK ----> NAK1=ACK0

确认信息减少一半，协议处理简单

Rdt3.0:具有比特差错 和 分组丢失的信道

新的假设：下层信道可能会丢失分组（数据或ACK）

会死锁

机制目前只能处理下面的情况：

检验和

序列号

ACK

重传

方法：发送方等待ACK一段合理的时间 --------> 超时重传机制

发送端超时重传：如果没有收到ACK---->重传

问题：如果分组（或ACK）只是被延迟了

重传会导致数据重复---->同2.1利用序列号处理

接收方必须指明被正确接收的序列号

需要一个倒计数定时器

链路层的timeout时间确定，传输层timeout时间式适应式的。 如果超时定时器没设置好，可能一直发两遍（设置时间<往返时间）

重传由立刻传变成等待timeout自动传

rdt3.0可以工作，但链路容量比较大的情况下，性能很差

链路容量（信道容量）比较大，一次发一个UDP的不能够充分利用链路的传输能力

信道容量：链路中可以容纳的比特量 ---> 俩收费站之间能有多少车

例：1Gbps的链路，15ms端-端传播延时，分组大小1kb

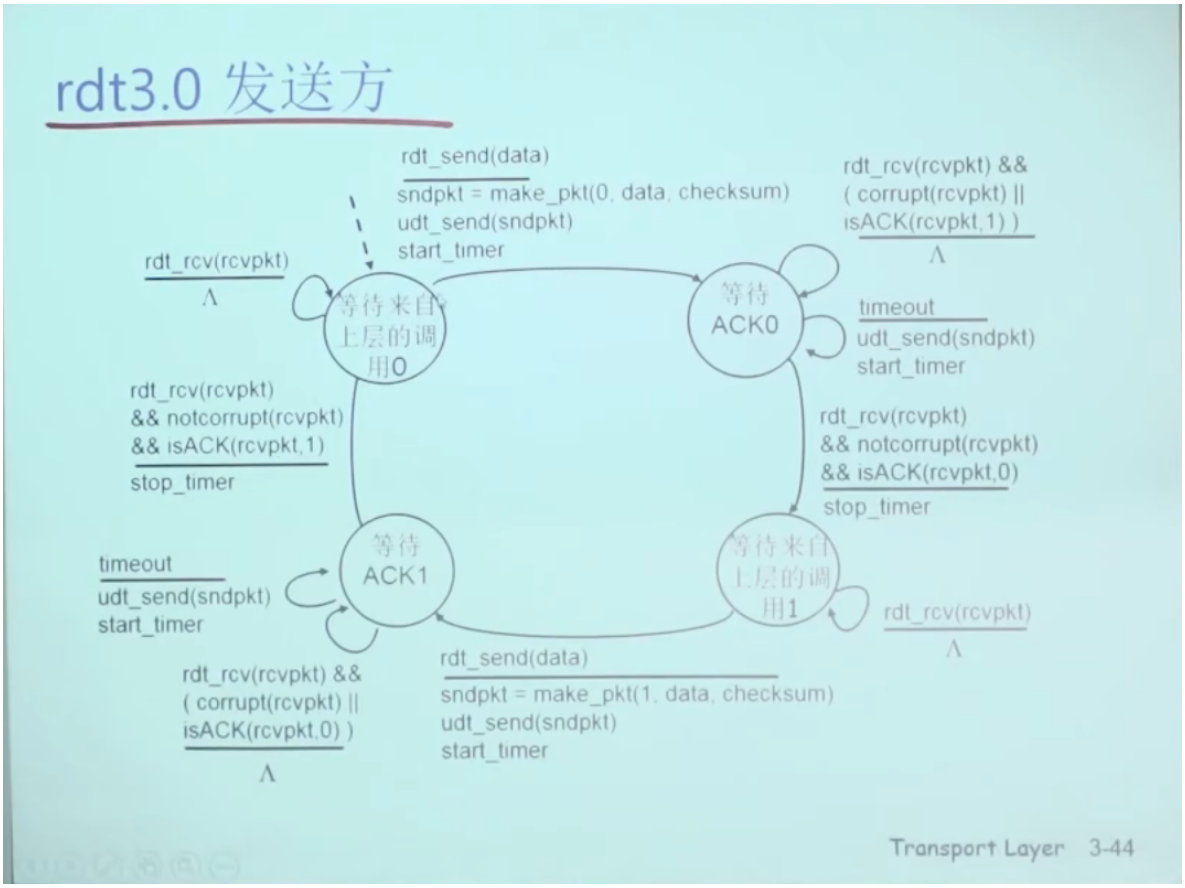
T_transmit=L(分组长度，bit)/R（传播速率，bps）=8kb/pkt / 10^9b/sec = 8微秒

U_sender= L/R / (RTT+L/R) = .008/30.008=0.00027

U_sender：利用率-忙于发送时间的比例

每30ms发送1kb的分组->270kbps=33.75kB/s的吞吐量（在1Gbps链路上）

瓶颈在于：网络协议限制了物理资源的利用



4.3 流水线协议

流水线：

允许发送方在未得到对方确认的情况下 一次发送多个分组

必须增加序号的范围：用多个bit表示分组的序号

在发送方/接收方要有缓冲区

发送方缓冲：未得到确认，可能需要重传

接收方缓冲：上层用户取得数据的速率!=接收到的数据速率 -----> 接收到的数据速率可能乱序，排序交付(可靠)

两种通用的流水线协议：回退N步（GBN）和 选择重传（SR）

通用：滑动窗口（slide window）协议

发送缓冲区：

形式：内存中的一个区域，落入缓冲区的分组可以发送

功能：用于存放已发送，但是没有得到确认的分组

必要性：需要重发时可用

发送缓冲区的大小：一次最多可以发送多少个未经确认的分组

停止等待协议=1

流水线协议>1，合理的值，不能很大，链路利用路不能超过100%

发送缓冲区中的分组：

未发送的：落入发送缓冲区的分组，可以连续发送出去

已经发送出去的、等待对方确认的分组：发送缓冲区的分组只有得到确认才能删除

SW（发送窗口） RW（接收窗口）

=1 =1 S-W stop and wish

>1 =1 GBN 回退N步

>1 >1 SR 选择重传

发送窗口：发送缓冲区内容的一个范围

发送窗口的最大值<=发送缓冲区的值

一开始：没有发送任何一个分组

后沿=前沿 理解为双指针，slow fast

之间为发送窗口的尺寸=0

每发送一个分组，前沿前移一个单位；收到确认，后沿前移一个单位

接收窗口=接收缓冲器

接收窗口用于控制哪些分组可以接收

只有收到的分组序号落入接收窗口内才允许接收

若序号在接收窗口之外，则丢弃

GBN:返回接收窗口前一位 n-1 ACKn-1 （假设当前窗口在n，ACKn-1，则会发送n）

接收窗口尺寸Wr=1，则只能顺序接收

Wr>1，则可以乱序接收

但是每个收到的确认是单独的

但提交给上层的分组要按序

接收窗口的滑动和发送确认：

滑动，低序号的分组到来，接收窗口移动

高序号分组乱序到，缓存但不交付（因为RDT不允许乱序），不滑动

发送确认：

接收窗口尺寸=1：（GBN） 发送连续收到的最大的分组确认（累计确认）-----> 意思是N包括N之前的分组都收到了

发送确认尺寸>1：（SR） 收到分组，发送那个分组的确认（非累计确认）-----> 意思就是第N个分组收到了，发N+1

正常情况下2个窗口互动：

发送窗口：

有新的分组落入发送缓冲区范围，发送----->前沿滑动

来了老的低序号分组的确认----->后沿向前滑动----->新的分组可以落入发送缓冲区的范围

接收窗口：

收到分组，落入到接收窗口的范围内，接收

是低序号，发送确认给对方

发送端上面来了分组----->发送窗口滑动----->接收窗口滑动----->发确认

异常情况下2个窗口滑动：

GBN:

发送窗口：

新分组落入发送缓冲区范围，发送----->前沿滑动

超时重发机制让发送端将发送窗口中的所有分组发送出去

来了老分组的重复确认----->后沿不向前滑动----->新的分组无法落入发送缓冲区的范围（此时如果发送缓冲区有新的分组可以发送）

接收窗口：

收到乱序分组，美欧落入到接收窗口范围内，抛弃。并发送顺序到达分组的最高序号n-1 ACKn-1----->意思是重发第N个分组

（重复）发送到老分组的确认，累计确认

SR:

发送窗口：

新分组落入发送缓冲区范围，发送----->前沿滑动

超时重发机制让发送端将超时的分组重新发送出去

来了乱序分组的确认----->后沿不向前滑动----->新的分组无法落入发送缓冲区的范围（此时如果发送缓冲区有新的分组可以发送）

接收窗口：

收到乱序分组，落入到接收窗口范围内，接收

发送该分组的确认，单独确认

SR的后延是可以移动的，但是遇到 未确认/未接收 就得停下

假设窗口大小4，发送数据0，1，2，3，接收到0，1，3 --->发送接收窗口移动到 2--->2发送超时重新发送，接收确认--->窗口滑动到4

即收到下沿确认时，下沿向前滑动

GBN与SR的异同：

相同：

发送窗口>1

一次能够发送多个未经确认的分组

不同：

GBN:接收窗口尺寸=1

接收端：只能按序接收

发送端：从表现看，一旦一个分组没有发送成功，如：0，1，2，3，4，加入1未成功，234都发送出去了，要返回1再发送； GB1

SR:接收窗口尺寸>1

接收端：可以乱序接收

发送端：发送0，1，2，3，4，5，一旦1未成功，234已发送，无需重发，选择发送1

总结：

GBN: Go-back-N

发送端最多在流水线有N个未确认的分组

接收端只是发送累计型确认

接收端如果发现gap，不确认新到来的分组

发送端拥有对最老的未确认分组的定时器

只需要设置一个定时器

当定时器到时，重传所有未确认分组------->这里的所有指n未确认，n之后都得重传

SR: Selective Repeat

发送端最多在流水线中有N个未确认的分组（发送窗口的最大值限制未确认分组的个数）

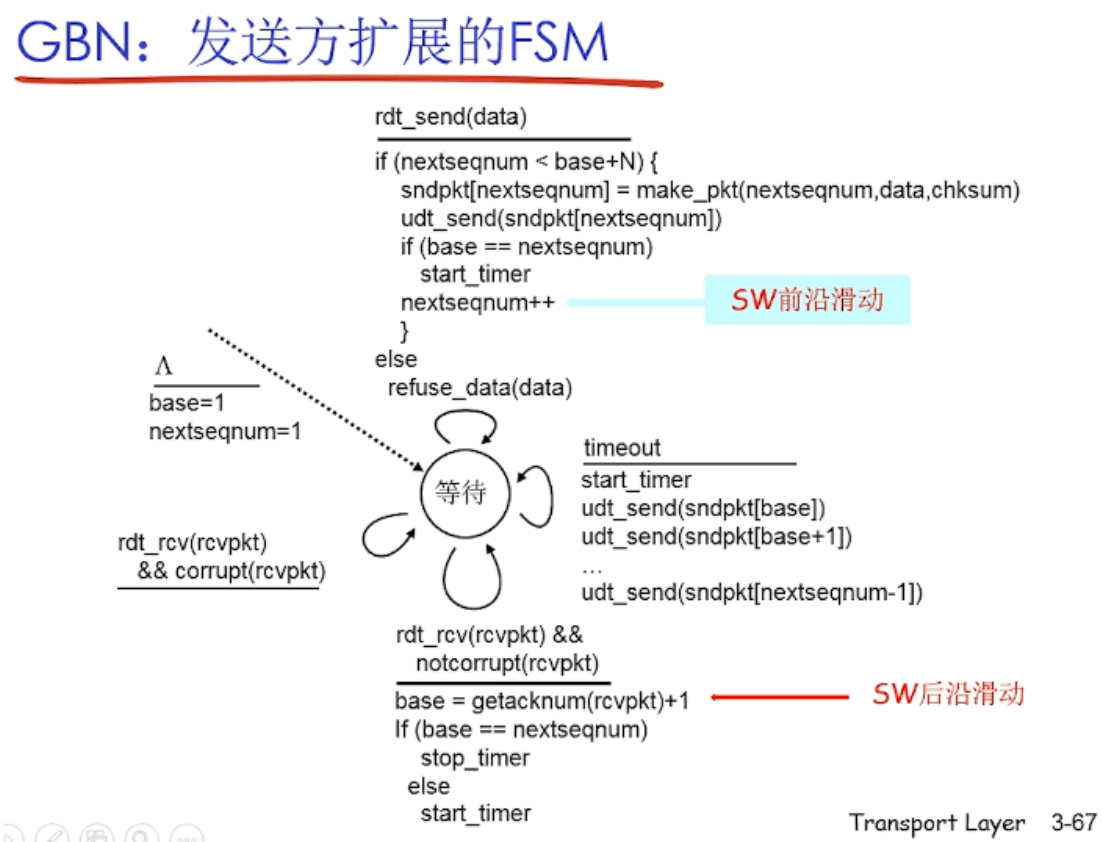
接收方对每个到来的分组单独确认（非累计确认）

接收窗口>1,可以缓存乱序的分组

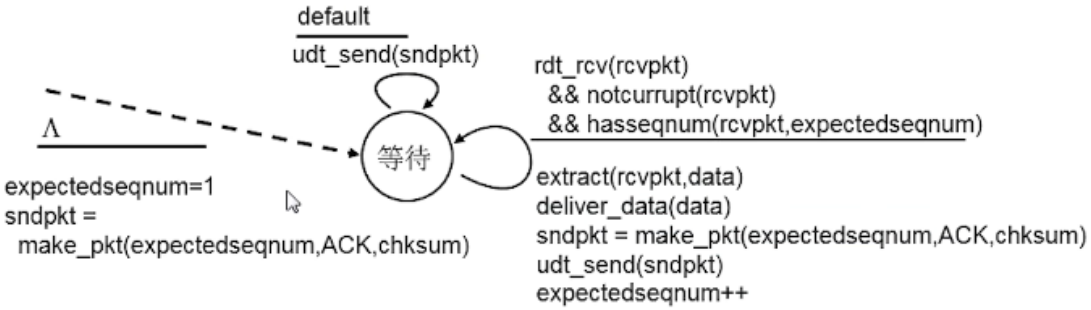
最终将分组按序交付到上层

发送方为每个未确认的分组保持一个定时器

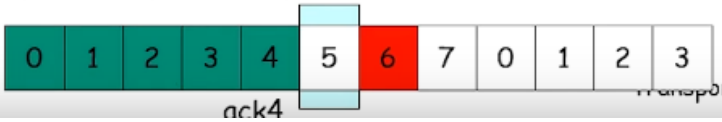
到定时器超时，重发超时的分组



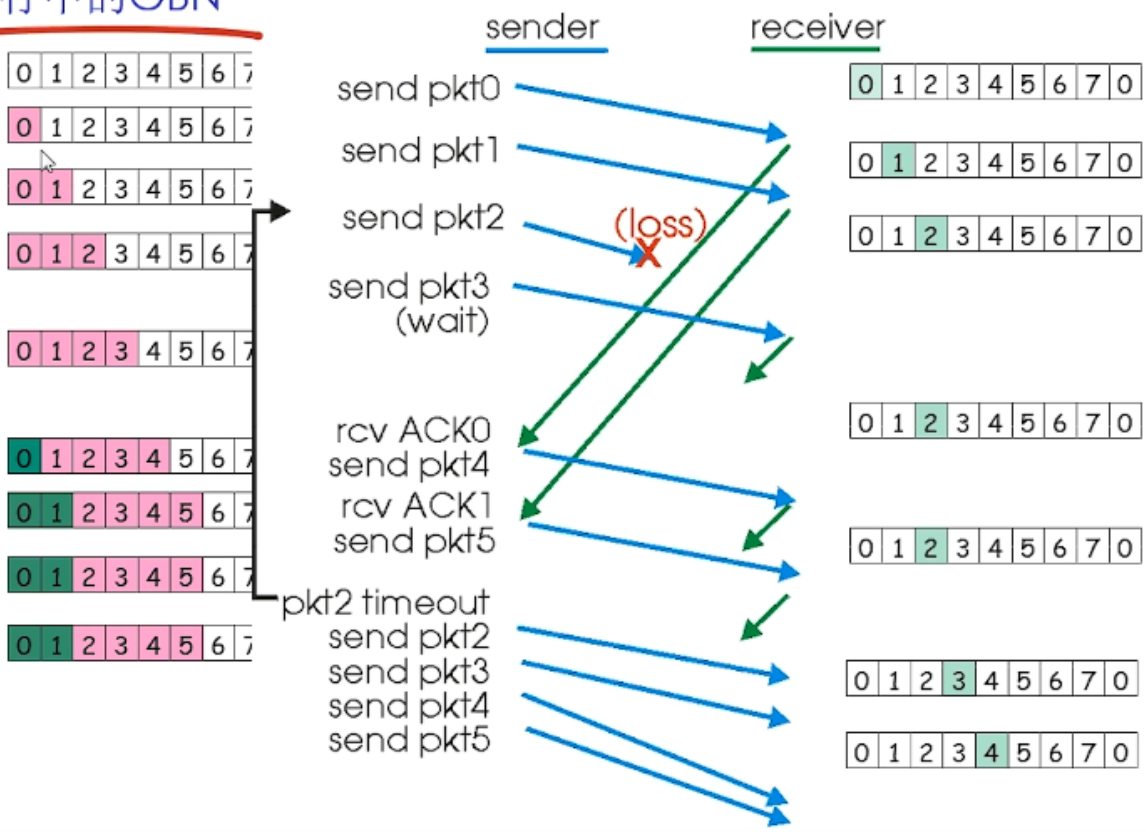
GBN：接收方扩展的FSM



- 只发送ACK：对顺序接收的最高序号的分组
  - 可能会产生重复的ACK
  - 只需记住**expectedseqnum**；接收窗口=1
    - 只一个变量就可表示接收窗口
- 对乱序的分组：
  - 丢弃（不缓存）→ 在接收方不被缓存！
  - 对顺序接收的最高序号的分组进行确认-累计确认



运行中的GBN



选择重传

发送方

从上层接收数据：

- 如果下一个可用于该分组的序号可在发送窗口中，则发送

timeout(n):

- 重新发送分组n，重新设定定时器

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

- 将分组n标记为已接收
- 如n为最小未确认的分组序号，将base移到下一个未确认序号

接收方

分组n [rcvbase, rcvbase+N-1]

- 发送ACK(n)
- 乱序：缓存
- 有序：该分组及以前缓存的序号连续的分组交付给上层，然后将窗口移到下一个仍未被接收的分组

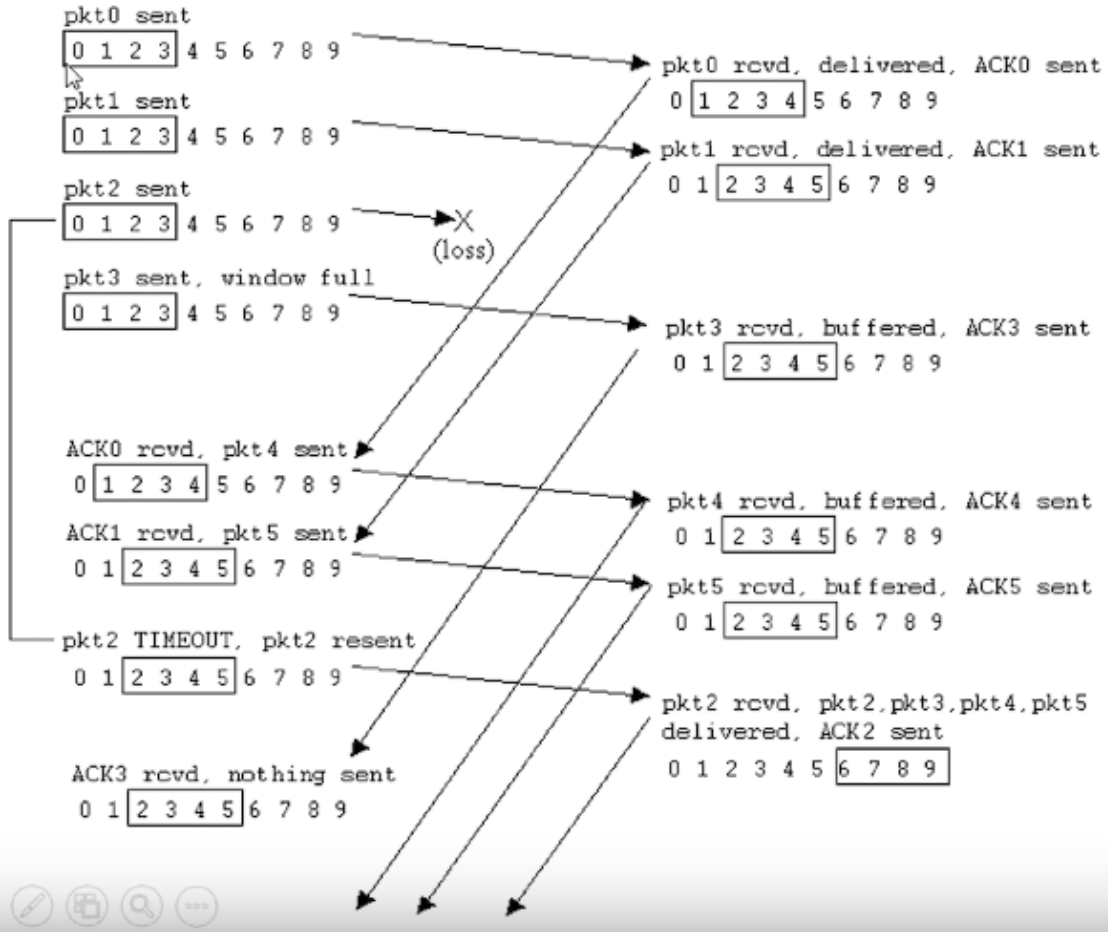
分组n [rcvbase-N, rcvbase-1]

- ACK(n)

其它：

- 忽略该分组

选择重传SR的运行



对比GBN与SR

GBN:

优点：

简单，所需资源少（接收方一个缓存单元）

缺点：

一旦出错，回退N步代价大

SR:

优点：出错时，重传1个代价小

缺点：复杂，所需资源多（接收方多个缓存单元）

使用范围：

出错率低：比较适合GBN，出错非常罕见，没有必要使用SR，为罕见的事情做日常的准备和复杂处理

链路容量大（延迟大，带宽大）：适合SR，因为GBN出错代价太大了

窗口最大尺寸：

GBN:2^N - 1

SR: 2^(N-1)

否则

SR:

接收方看不到两者的区别

把重复数据误认为新数据