研讨课-1：数据可靠传输机制

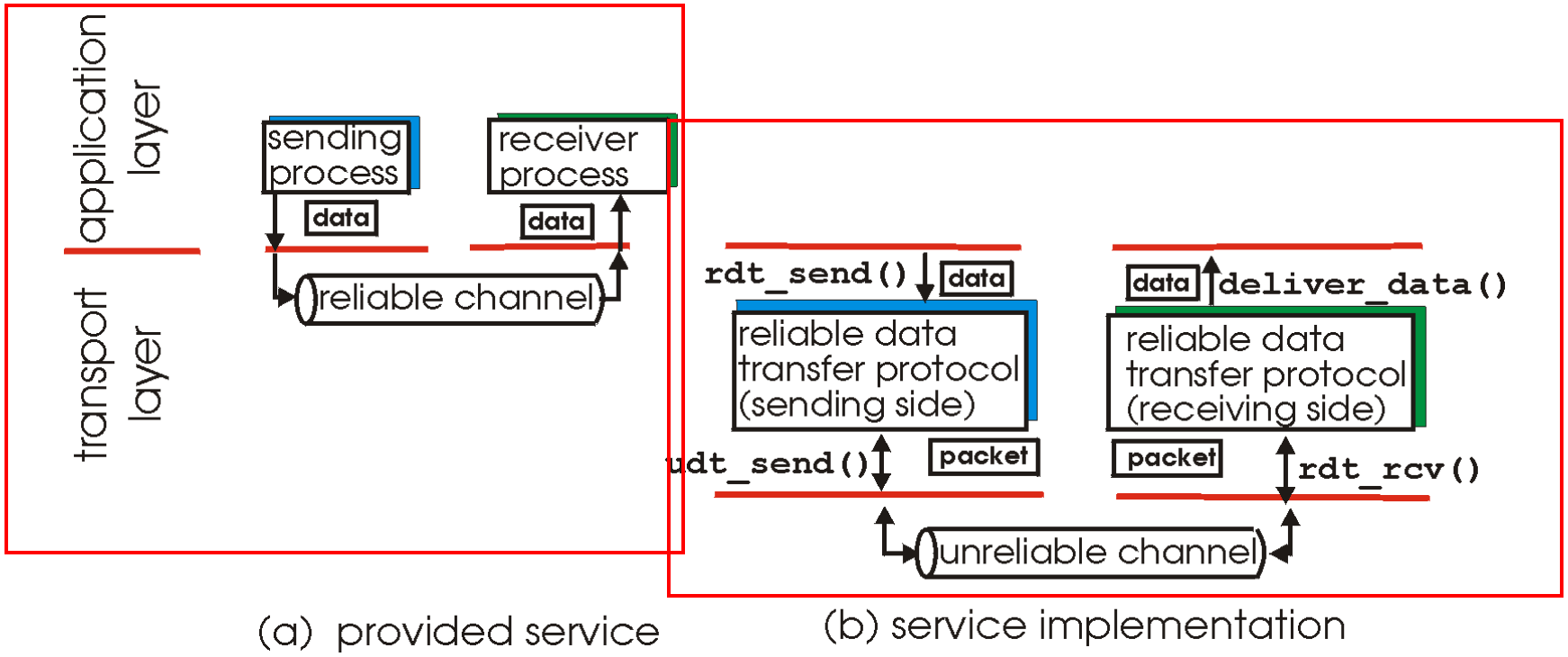
3.4 principles of reliable data transfer

3.5 TCP: reliable data transfer

- 目的
  - 通过研讨让学生理解在有比特差错和丢包的网络场景下，如何设计数据的可靠传输机制
  - 理解TCP协议中的可靠数据传输机制
- 采用的形式
  - 抛出问题+教师引导+学生讨论的形式

准备知识：可靠数据传输的原则

- 在应用层、运输层、数据链路层的重要性
  - **重要的网络主题中的最重要的10个之一!**

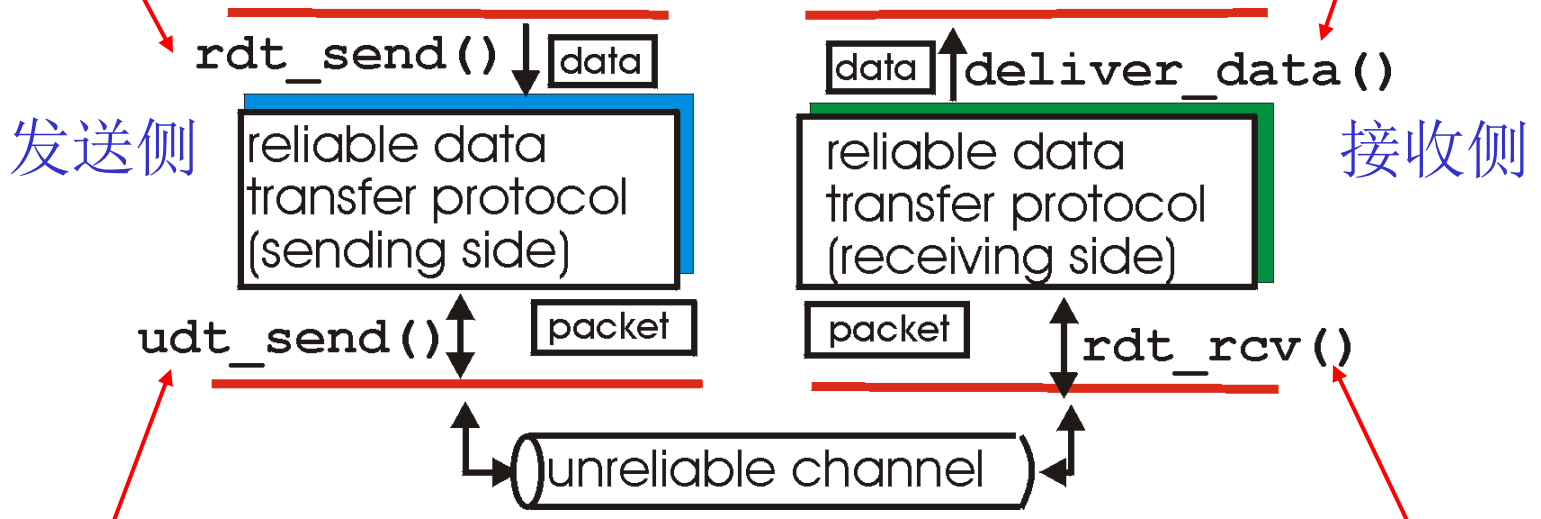


- 不可靠信道的特点决定了可靠数据传输 协议 (rdt) 的复杂性

准备知识：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

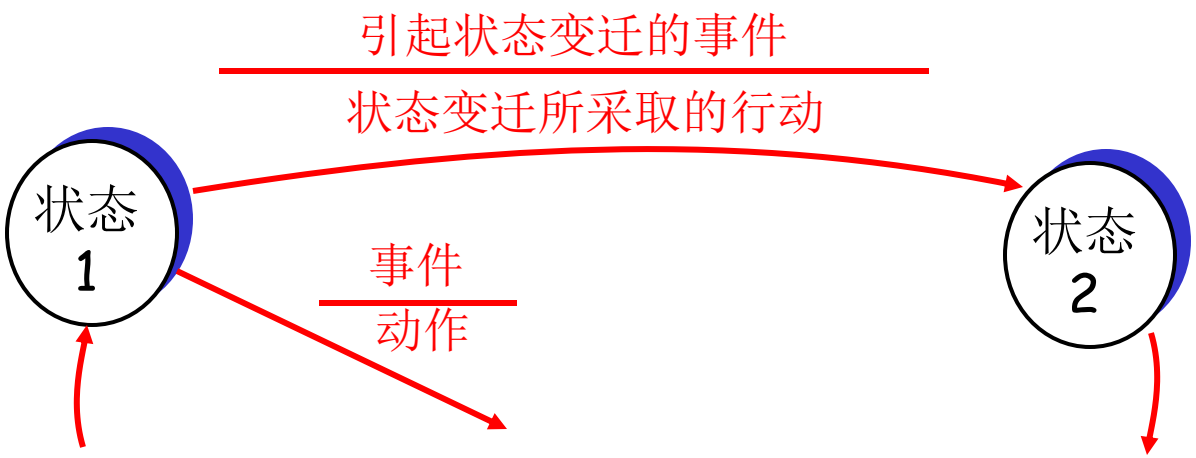
rdt_rcv(): called when packet arrives on rcv-side of channel

准备知识：可靠数据传输基本概念

我们将:

- 增强研发发送方，可靠数据传输协议 (rdt) 的接收方侧
  - 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

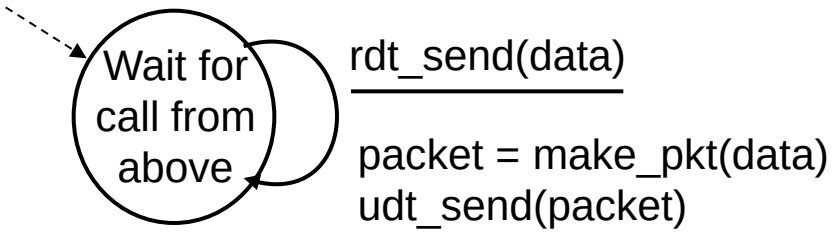


抛出问题

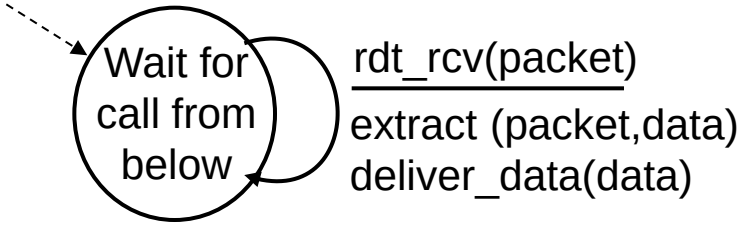
- 简单场景：收发双方**经可靠信道**传输数据分组
  - 无需提供额外的保证机制
- 较复杂场景：收发双方**经有比特差错的信道**传输数据分组
  - 需要提供的机制：
- 真实场景：收发双方**经有比特差错和丢包的信道**传输数据分组
  - 需要提供的机制：
- 因特网的TCP协议中采用的机制有哪些？
  - 分析采用的原因

简单场景：经可靠信道传输

- 场景解读：底层信道非常可靠
  - 无比特差错
  - 无分组丢失
- 机制设计：装发送方、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



发送方



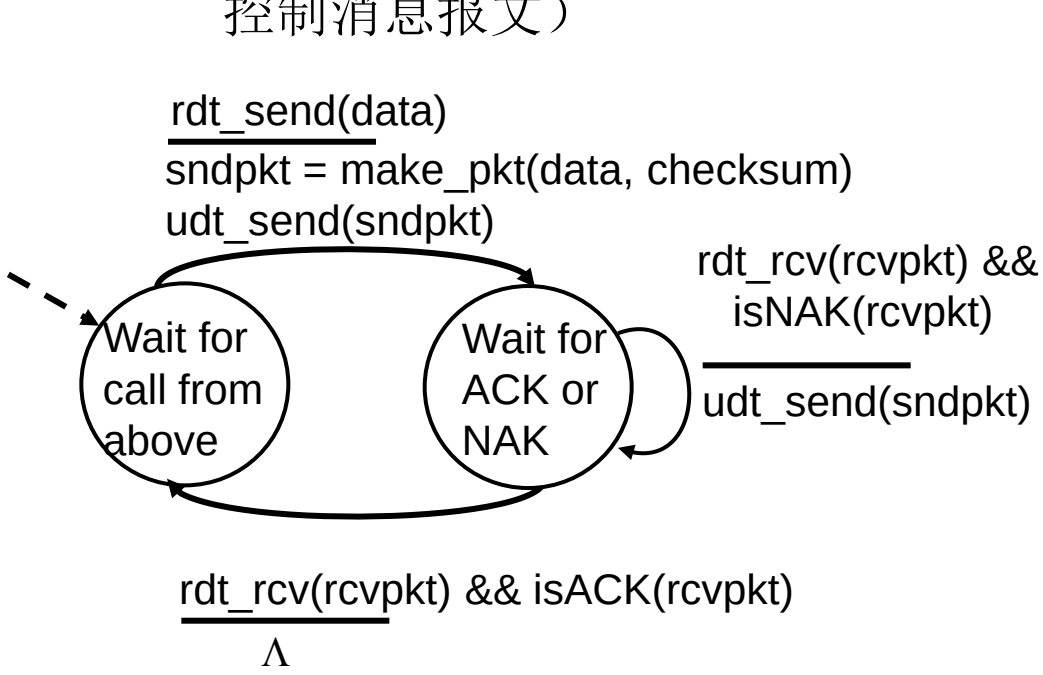
接收方

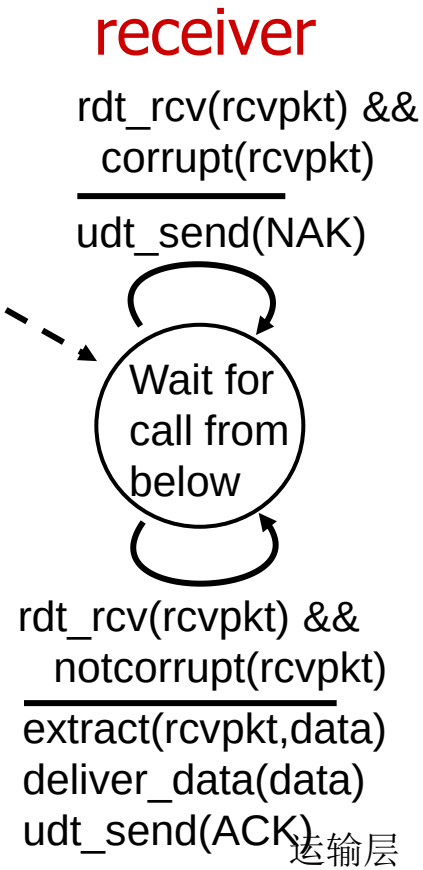
较复杂场景：经有比特差错的信道传输

- 场景解读：
  - 底层信道可能会将分组中的某些比特反转
  - 无分组丢失
- 机制设计：
  - 怎么检测到比特错误：检验和机制(checksum)
  - 检测到错误后的处理方法：由接收方通知发送方重传
    - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK（显式的告诉发送人成功收到了分组）
    - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors（接收人显式告诉发送人分组出错）
    - sender retransmits pkt on receipt of NAK（当收到NAK报文时，发送人重传出错的分组）

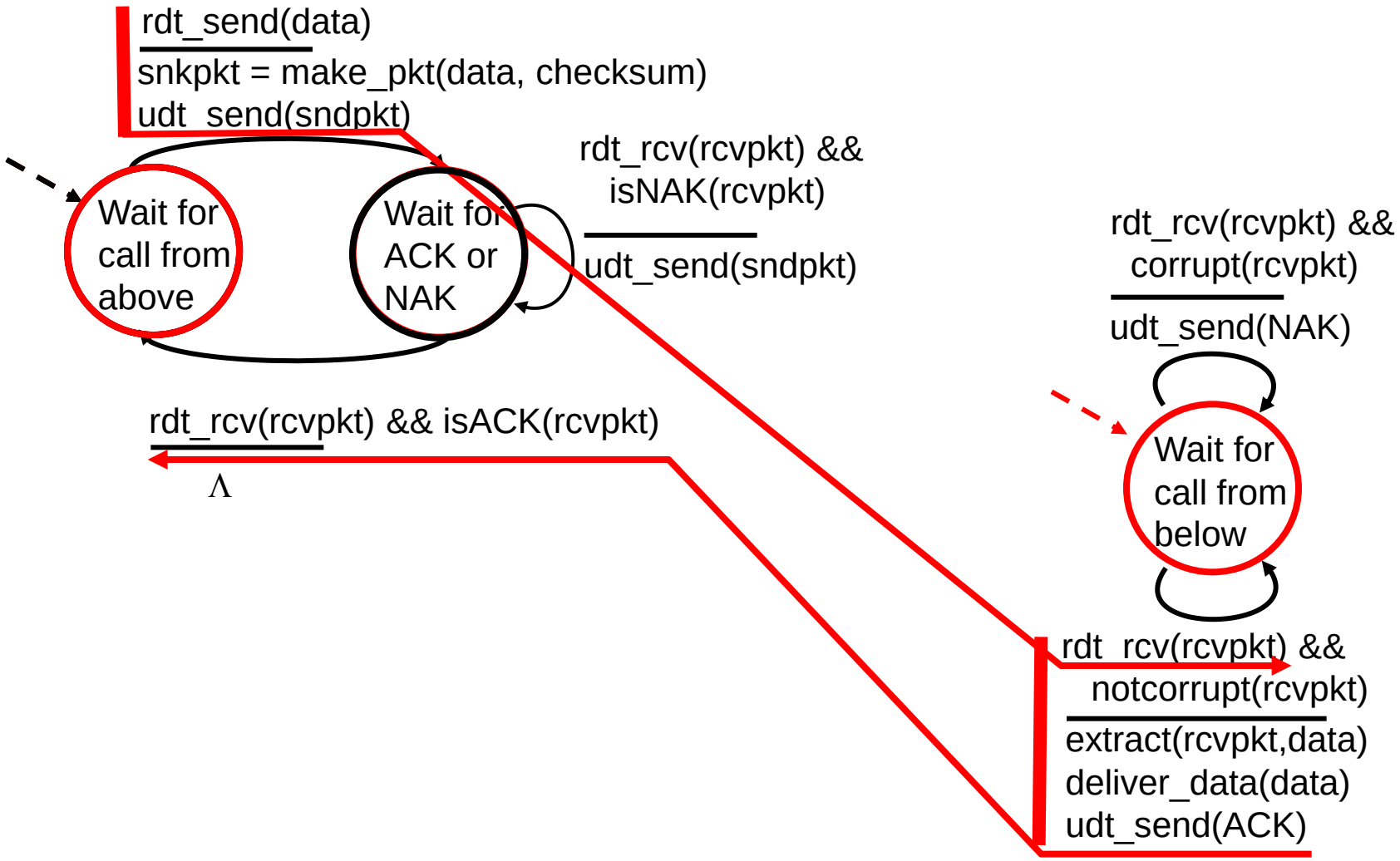
较复杂场景：经有比特差错的信道传输

- rdt2.0机制
  - error detection（出错检测）
  - receiver feedback: control msgs (ACK, NAK) rcvr->sender（发送控制消息报文）

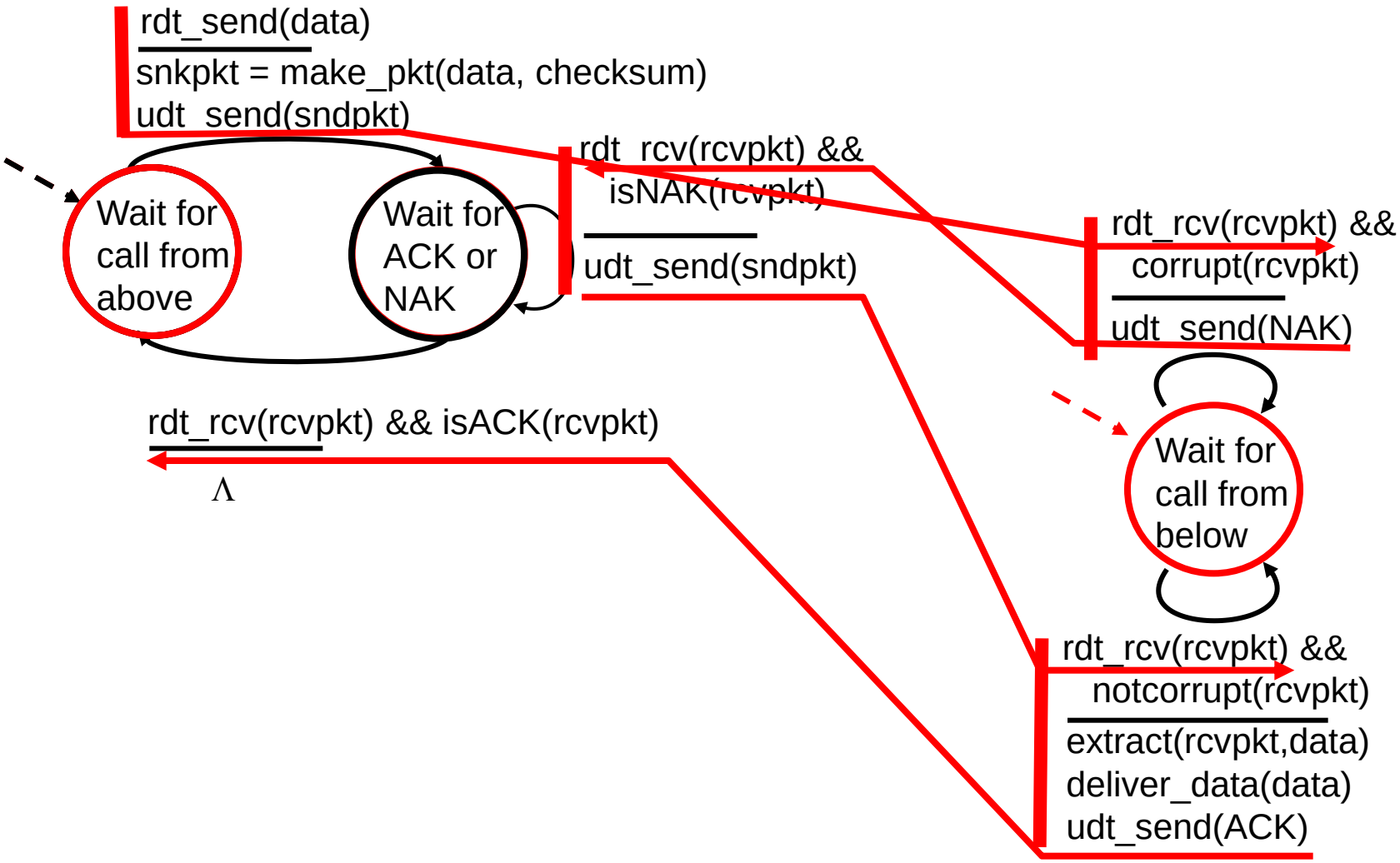




rdt2.0: operation with no errors



rdt2.0: error scenario



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：可能导致冗余

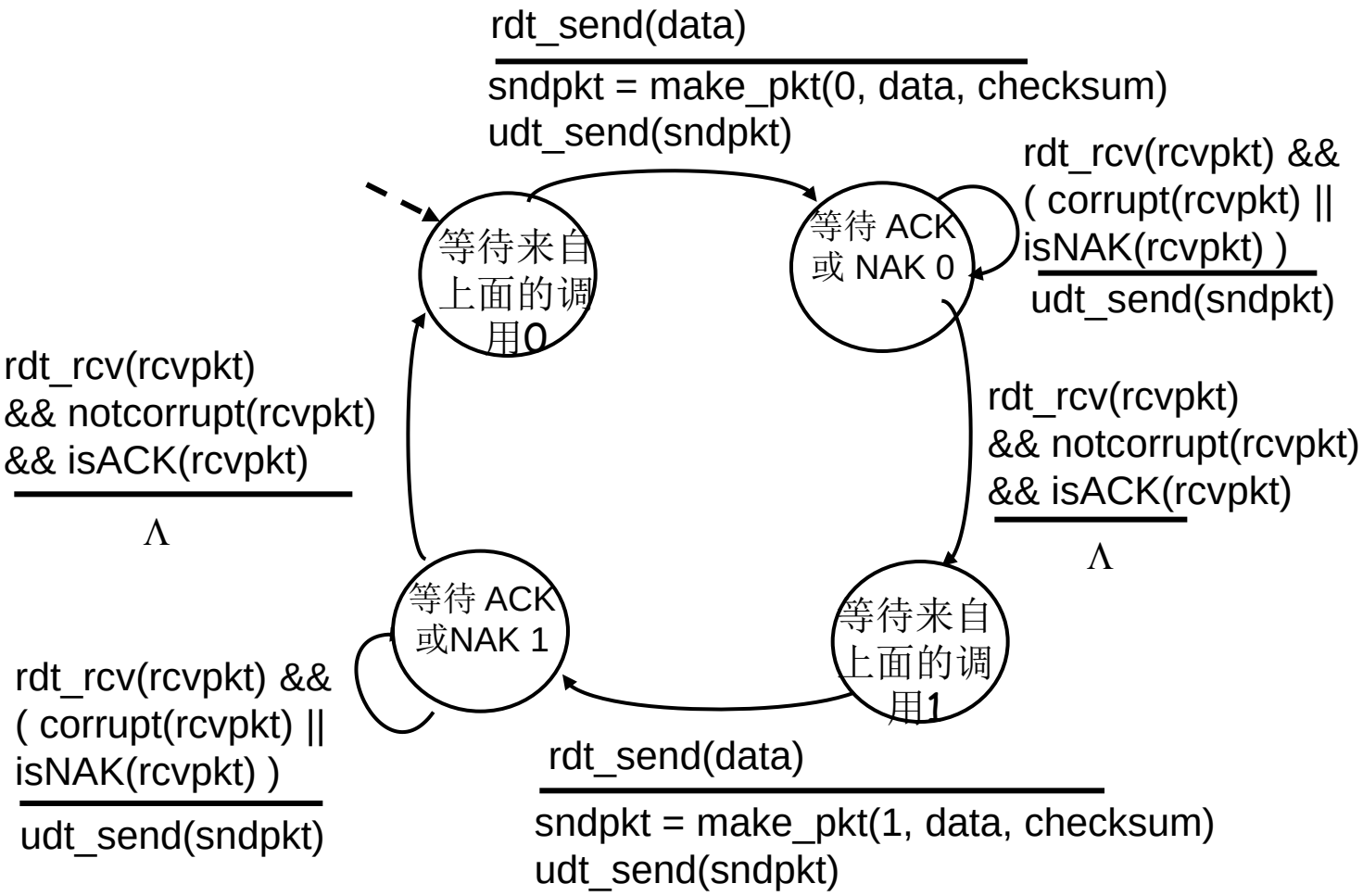
处理冗余:

- 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)冗余分组

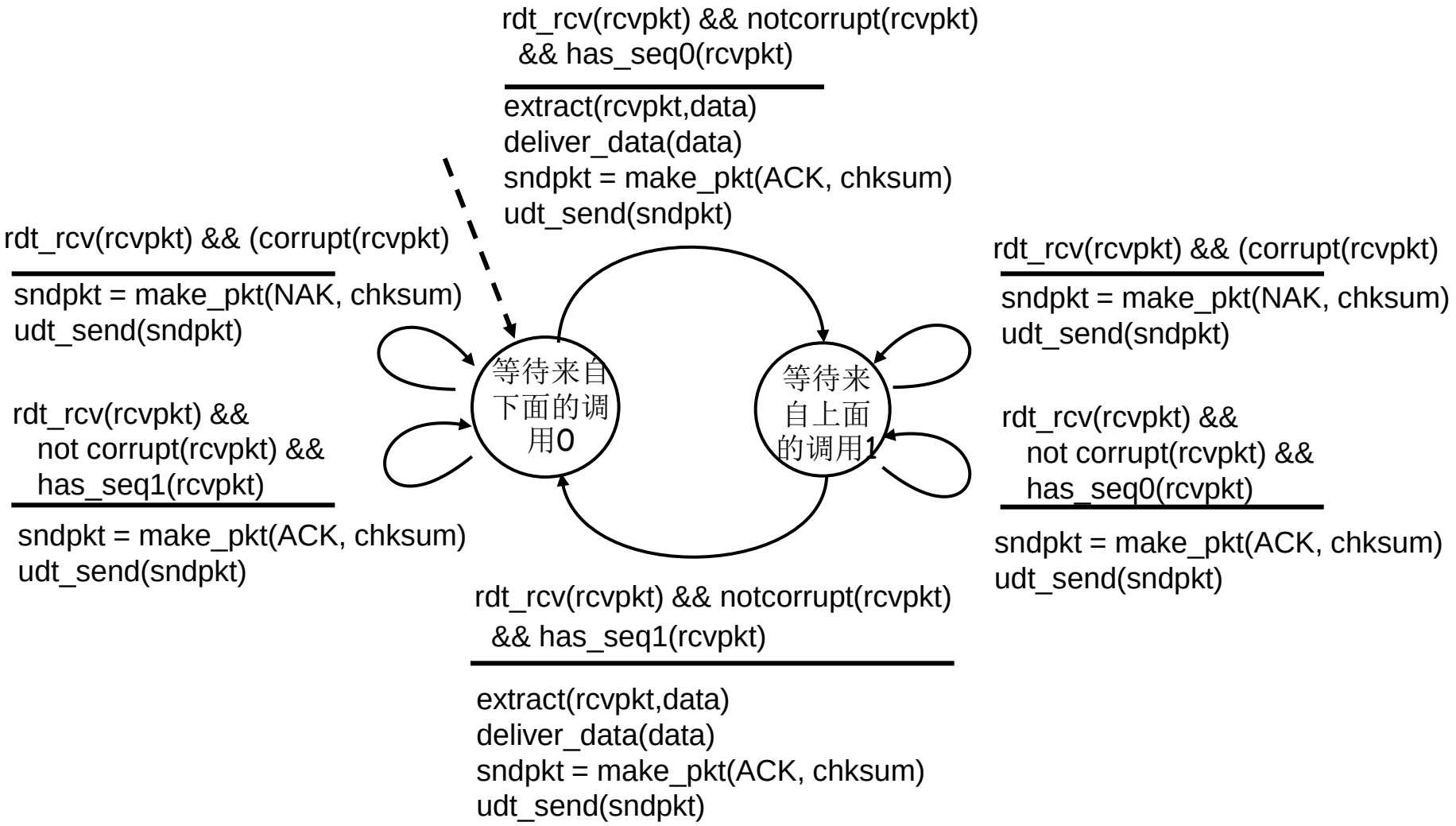
停止等待

发送方发送一个分组，然后等待接收方响应

rdt2.1: 发送方, 处理受损的ACK/NAK



rdt2.1: 接收方,处理受损的ACK/NAK



rdt2.1: 讨论

发送方:

- 序号seq # 加入分组中
- 两个序号seq. #'s (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”“当前的”分组具有0或1序号

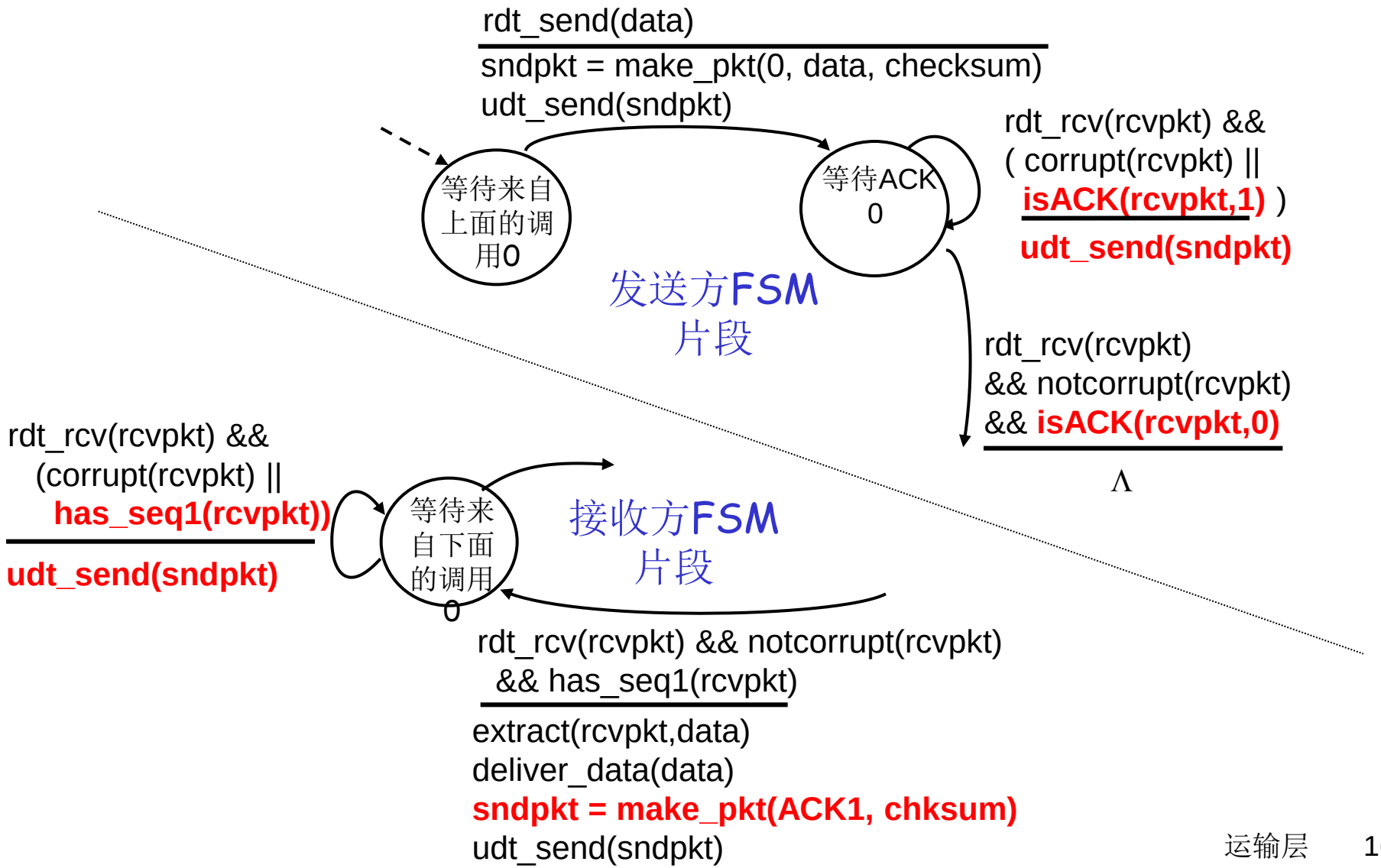
接收方:

- 必须检查是否接收到的分组是冗余的
  - 状态指示（0或1）表明所期待的分组序号seq #
- 注意: 接收方不能知道是否它最后的ACK/NAK在发送方已经成功接收

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, 接收方片段



rdt3.0: 具有比特差错和丢包的信道

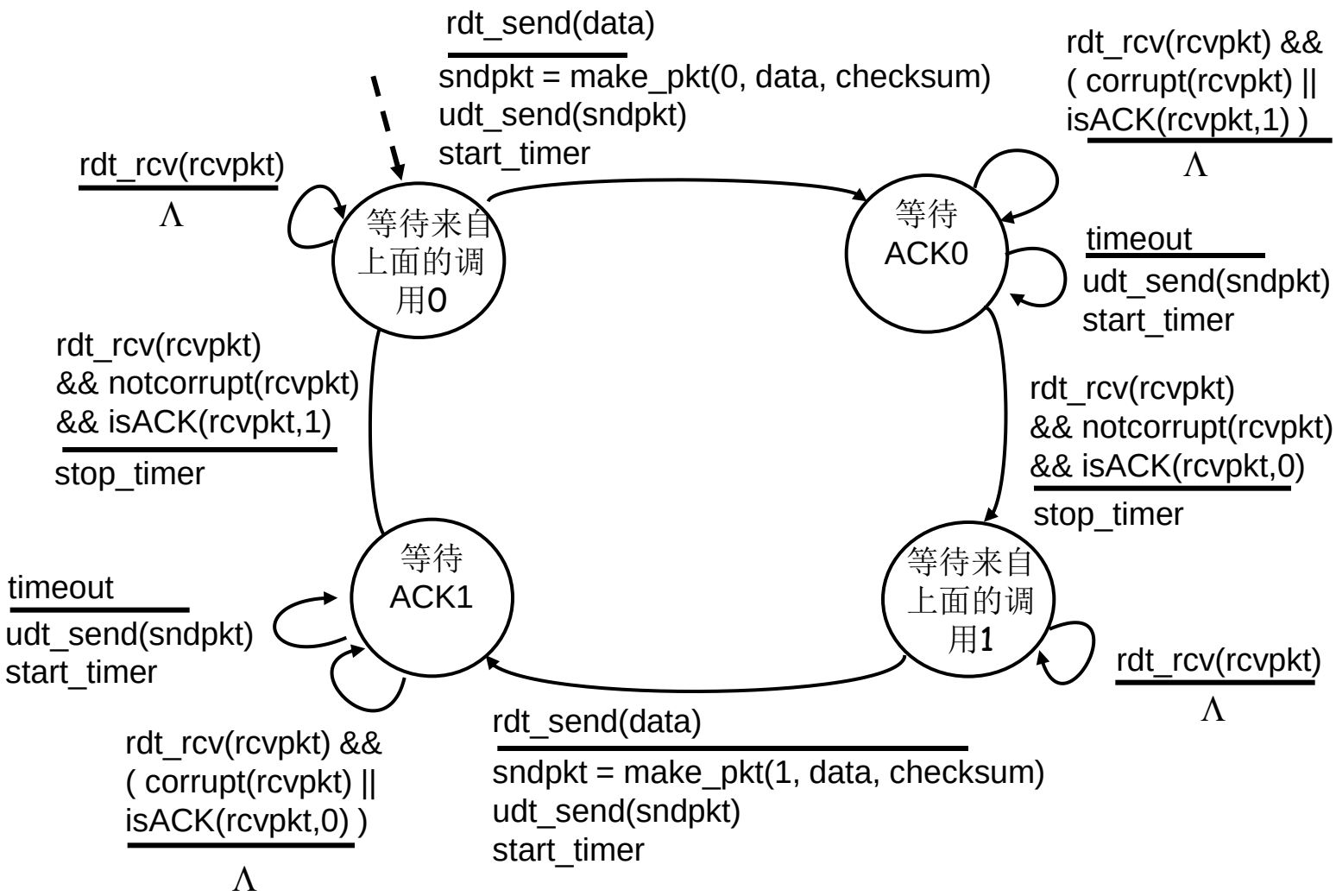
新假设: 下面的信道也能丢失分组(数据或ACK)

- 检查和、序号、重传将是有帮助的，但不充分

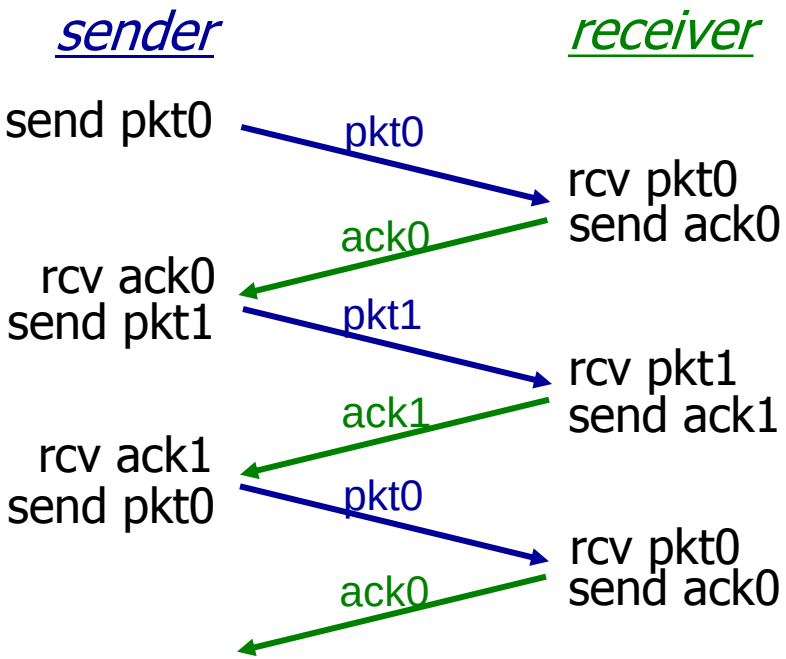
方法: 发送方等待ACK一段“合理的”时间

- 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

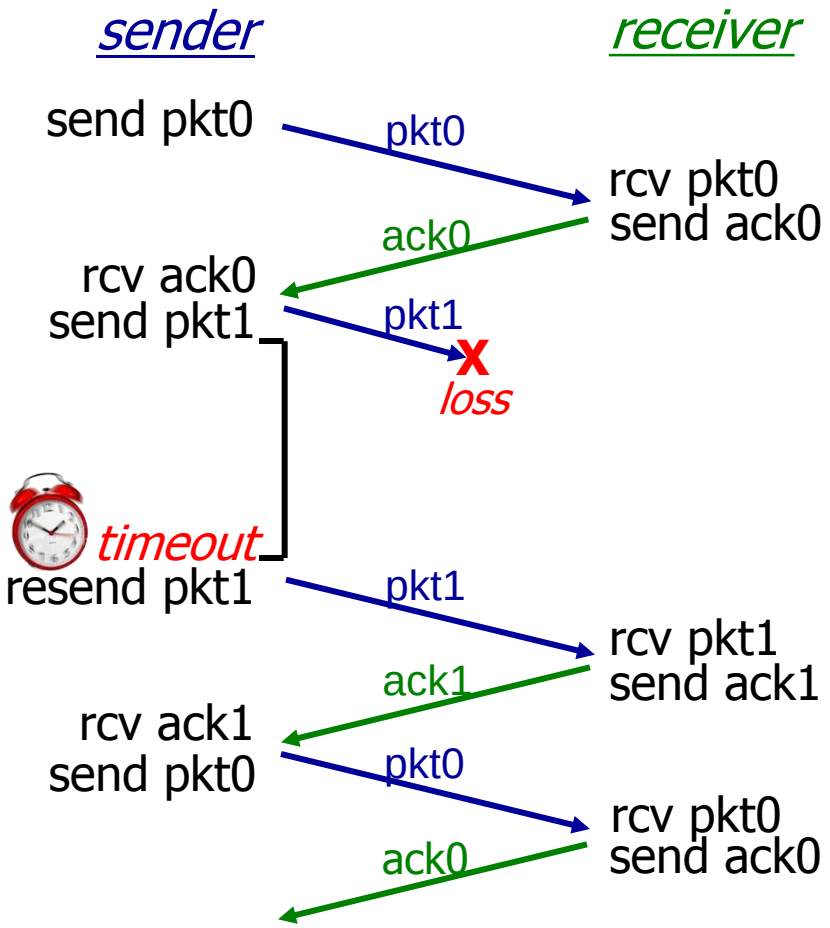
rdt3.0发送方


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 in action

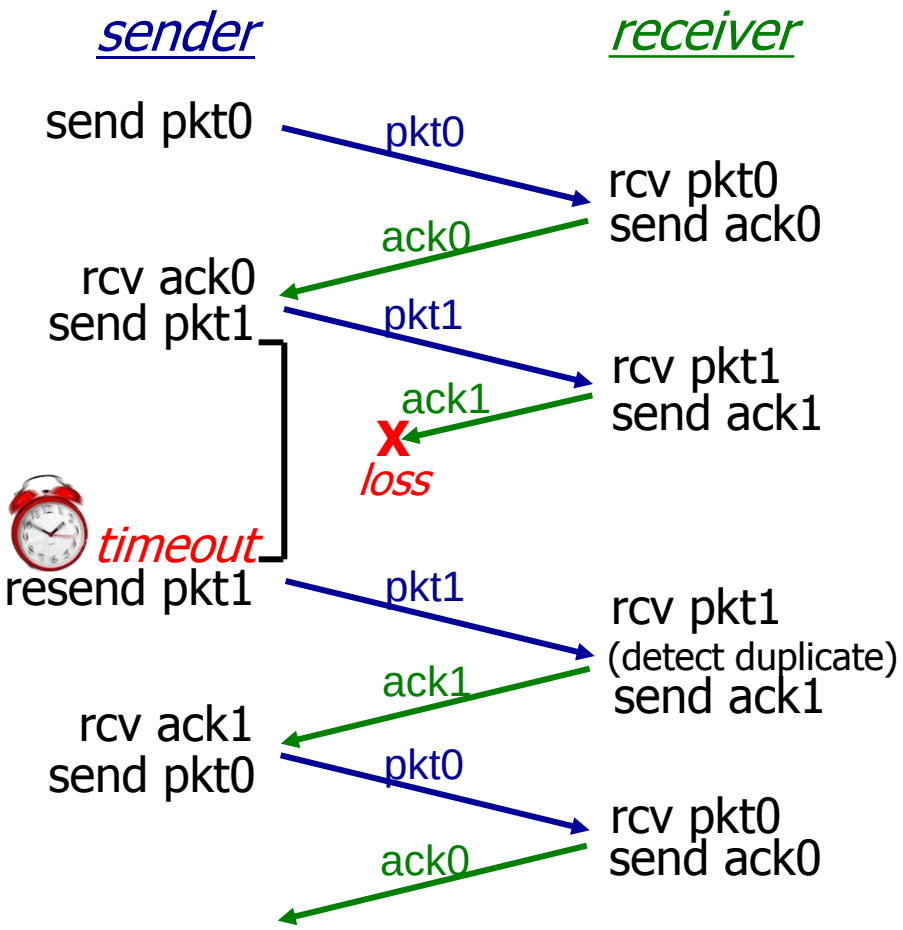


(a) no loss

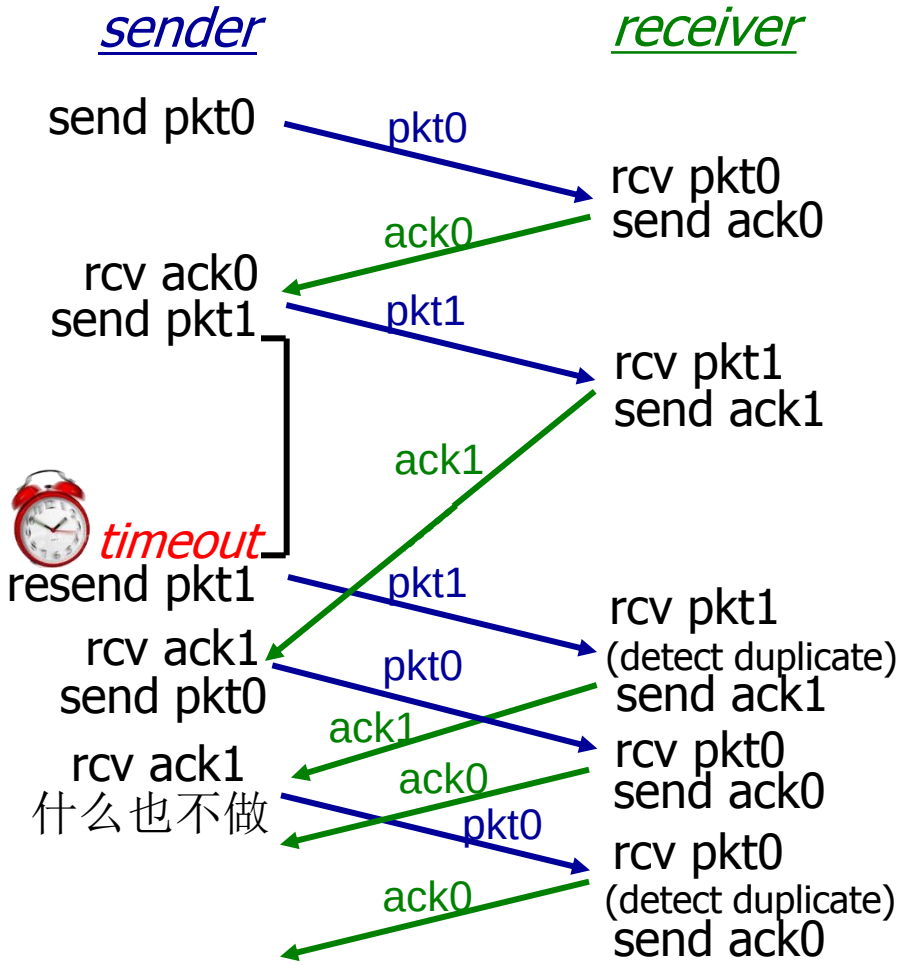


(b) packet loss

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

Class Experiments

- 请同学们自己画出rdt3.0接收方的FSM

上次研讨课内容回顾

- 通过探索的方式，逐层深入的讨论了真实场景下的可靠传输机制，最终得到了rdt3.0(停-等方式)协议；
- 探索了以下可靠传输机制：
  - 差错检查
  - 正确确认和否定确认
  - 重传机制
  - 定时器机制
- 要求大家能够掌握怎么通过有限状态机来描述协议

（简单协议描述，看得懂）

分析rdt3.0协议的性能

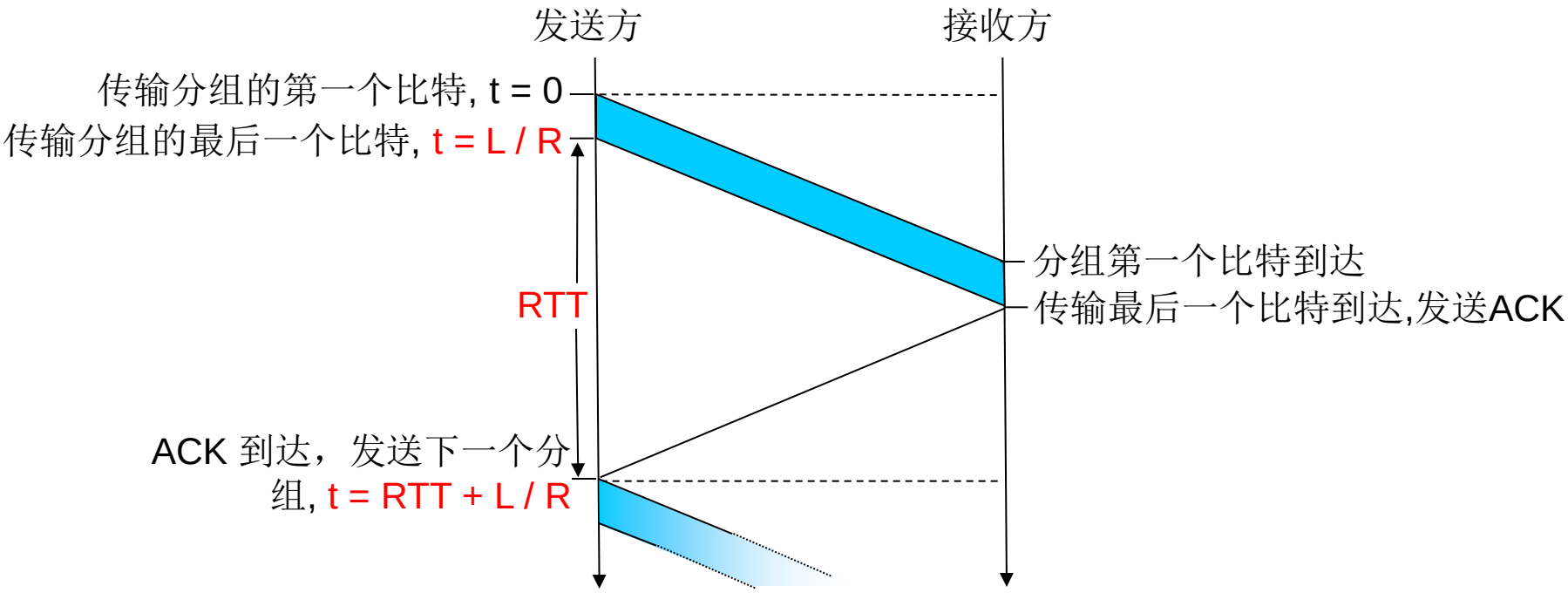
- rdt3.0能够工作，但性能不太好
- 例子: 1 Gbps链路, 15 ms端到端传播时延, 1KB分组:

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

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

  - U_{sender}: 利用率 – 发送方用于发送时间的比率
  - 每30 msec 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!

rdt3.0: 停等协议的运行

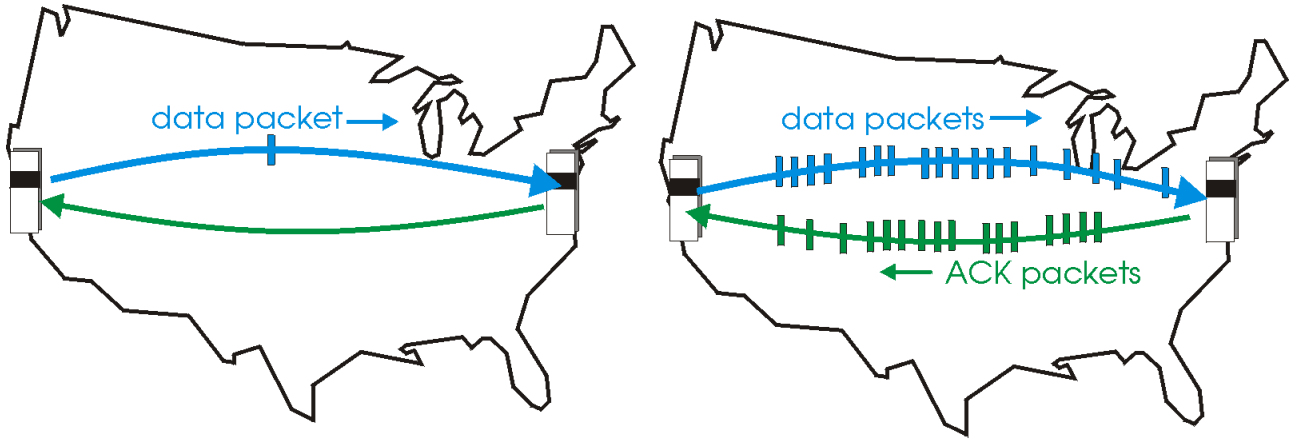


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

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

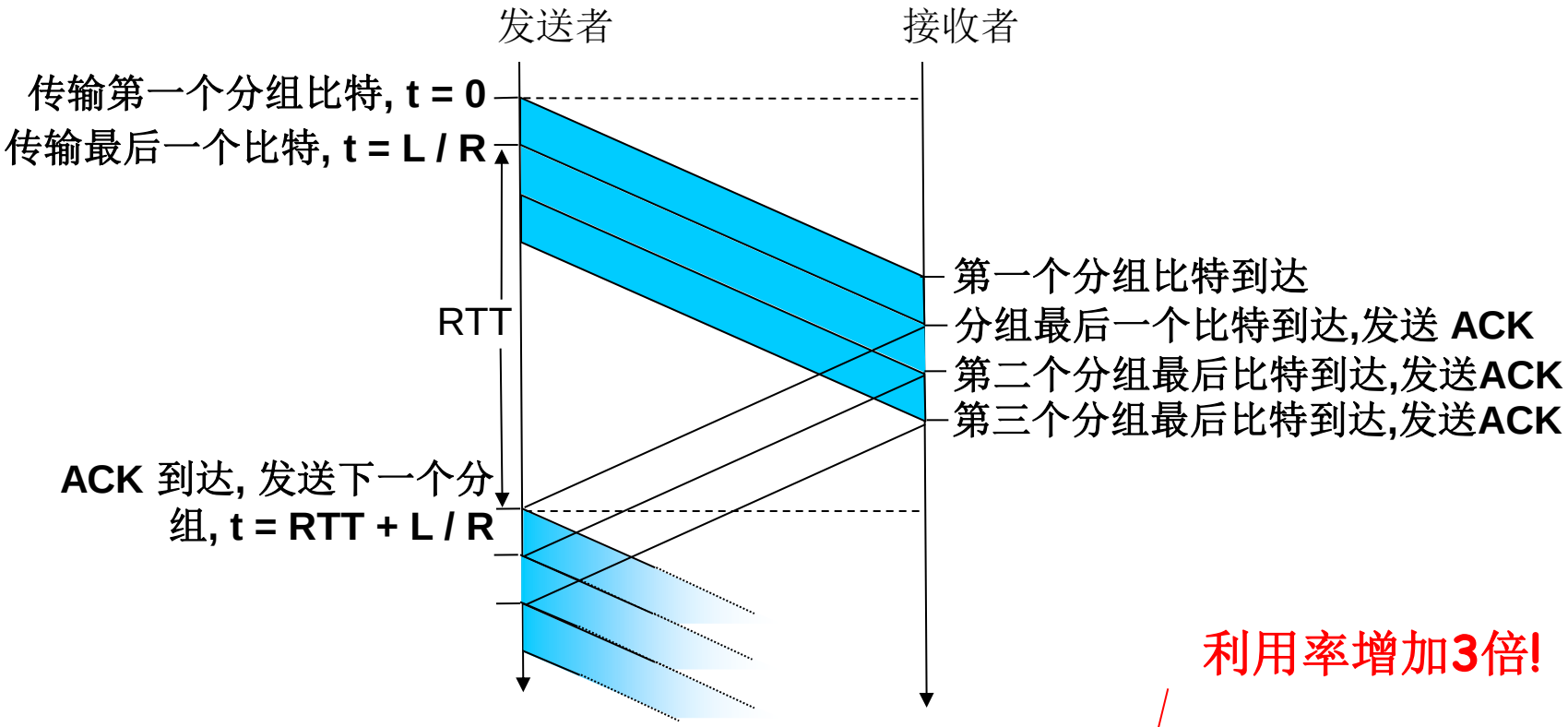


(a) a stop-and-wait protocol in operation　　(b) a pipelined protocol in operation

- 流水线协议的两种形式:

回退N帧法（go-Back-N），选择性重传（S-R），

流水线协议: 增加利用率



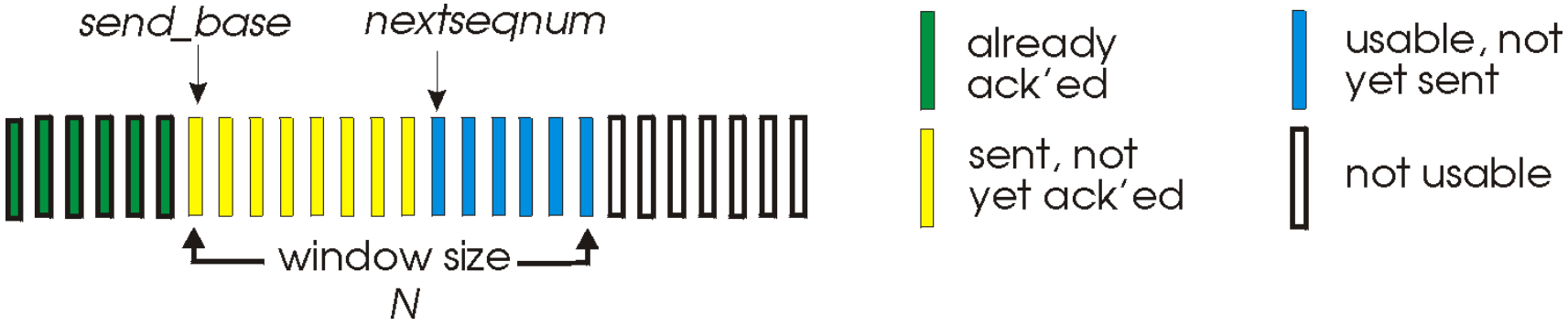
利用率增加3倍!

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

Go-Back-N

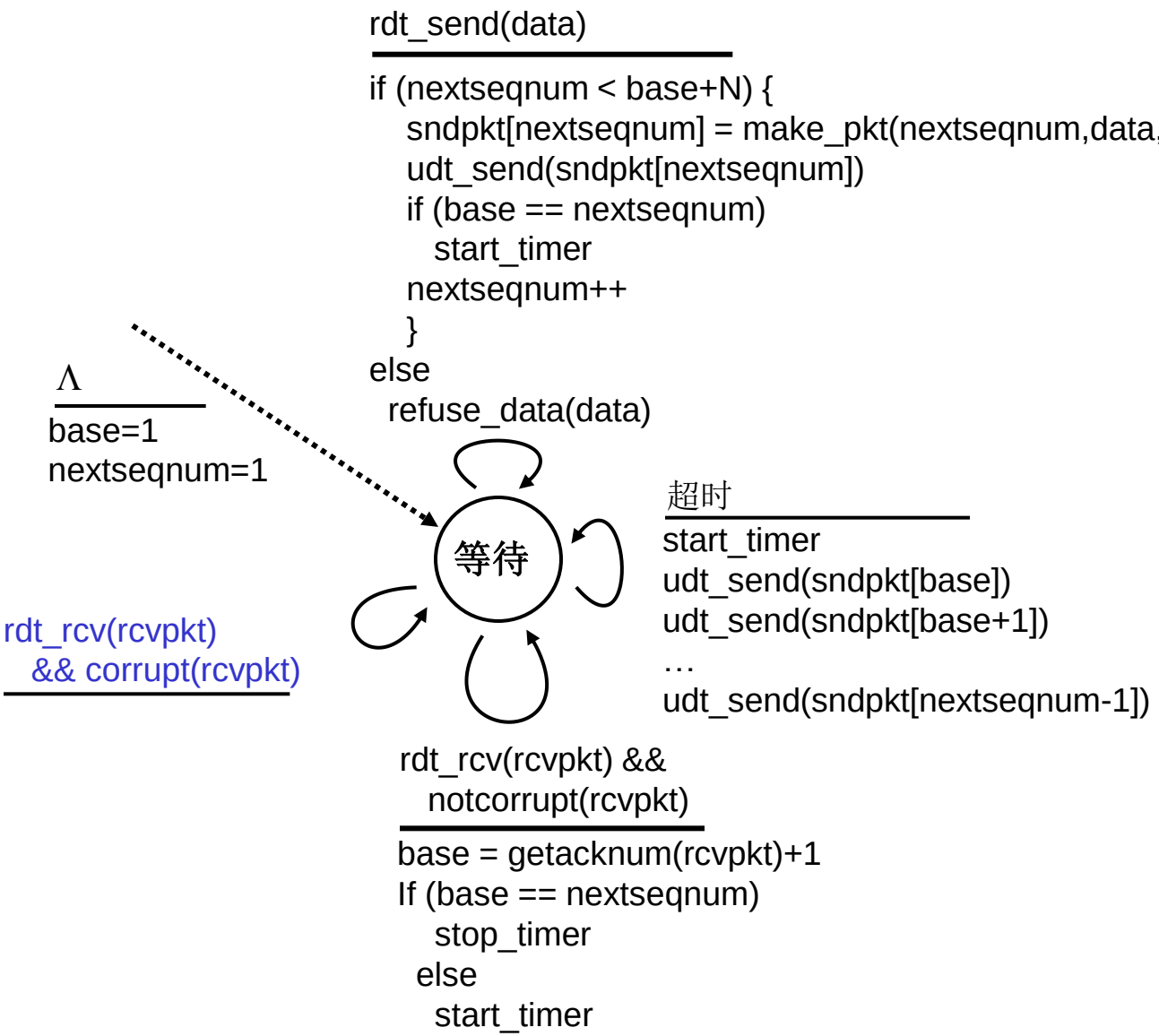
发送方:

- 在分组首部需要K比特序号，2^k=N
- “窗口”最大为N, 允许N个连续的没有应答分组

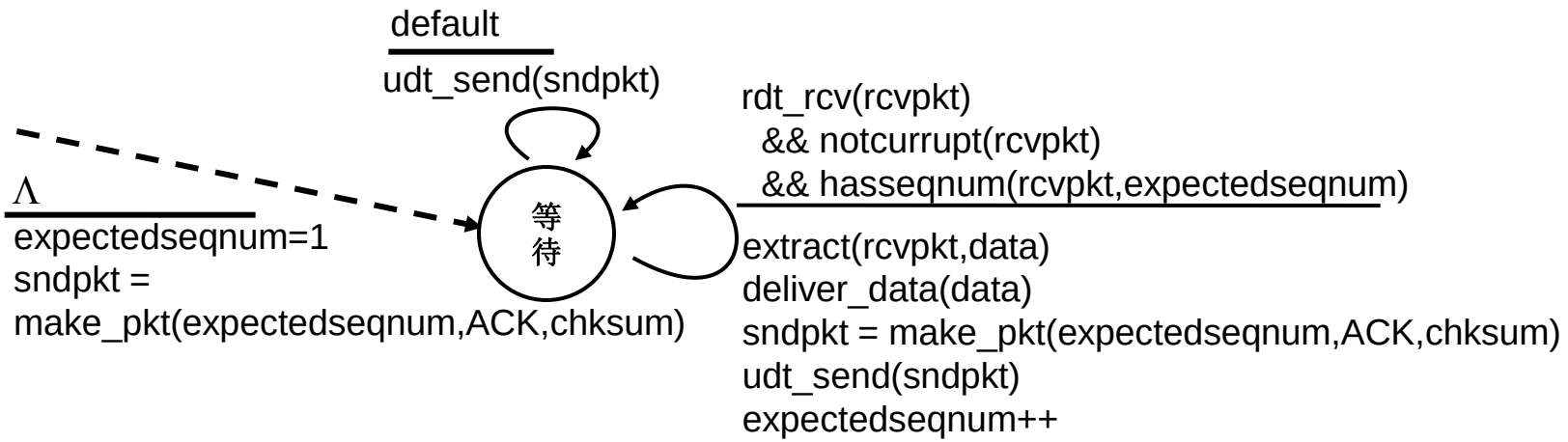


- ACK(n): ACKs all pkts up to, including seq # n - *“cumulative ACK”*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

GBN: 发送方扩展的 FSM

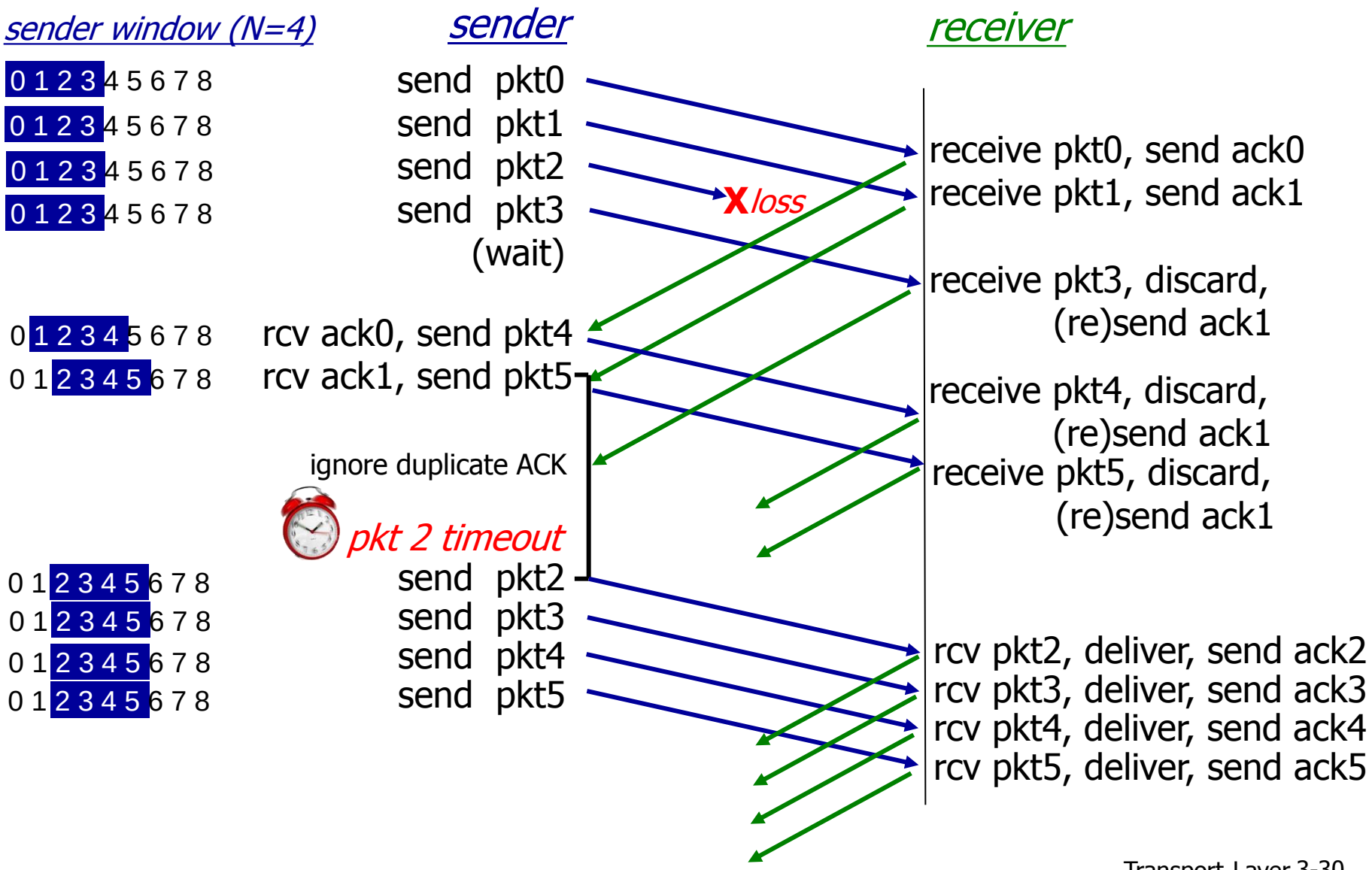


GBN: 接收方扩展 FSM



- 只有ACK: 对发送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

GBN in action

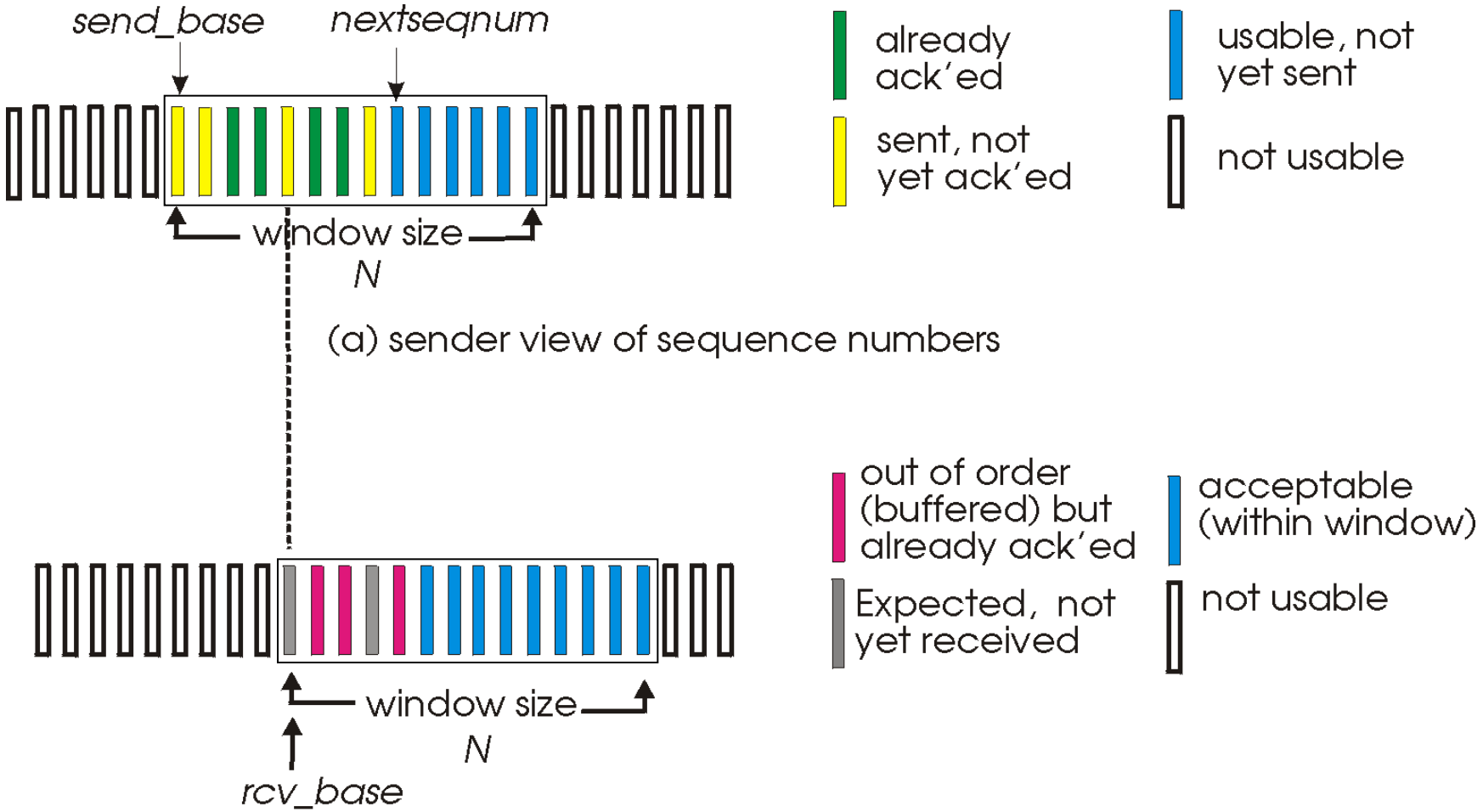


选择性重传（Selective Repeat）

GBN改善了信道效率，但仍然有不必要重传问题，为此进一步提出了SR机制

- SR接收方分别确认所有正确接收的报文段
  - 需要缓存分组，以便最后按序交付给给上层
- SR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
- SR发送窗口
  - N个连续的序号
  - 也需要限制已发送但尚未应答分组的序号

Selective repeat: sender, receiver windows



GBN只有发送窗口；SR既有发送窗口，也有接收窗口

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]:

- 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

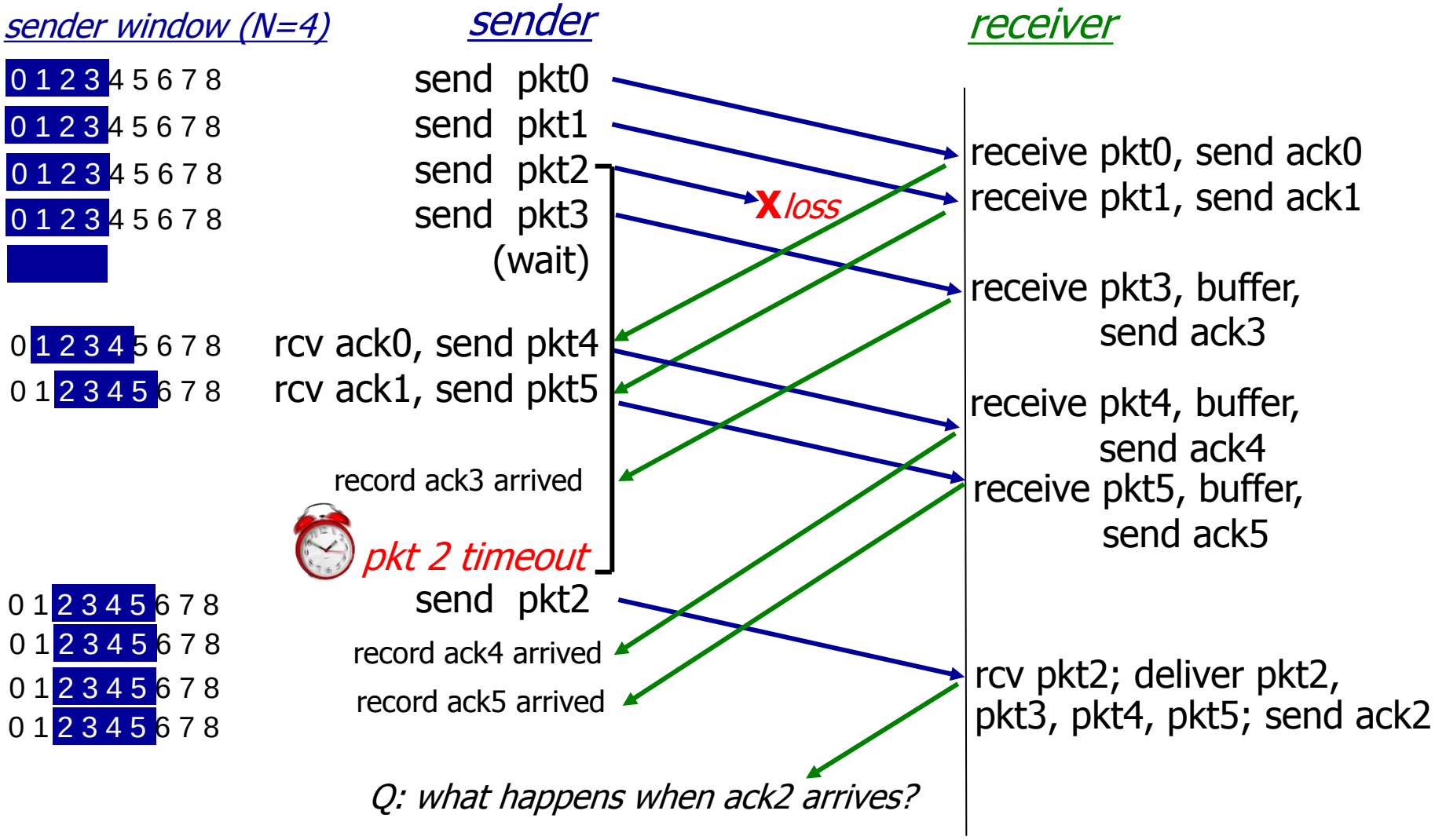
分组n在[rcvbase-N,rcvbase-1]

- ACK(n)

其他:

- 忽略

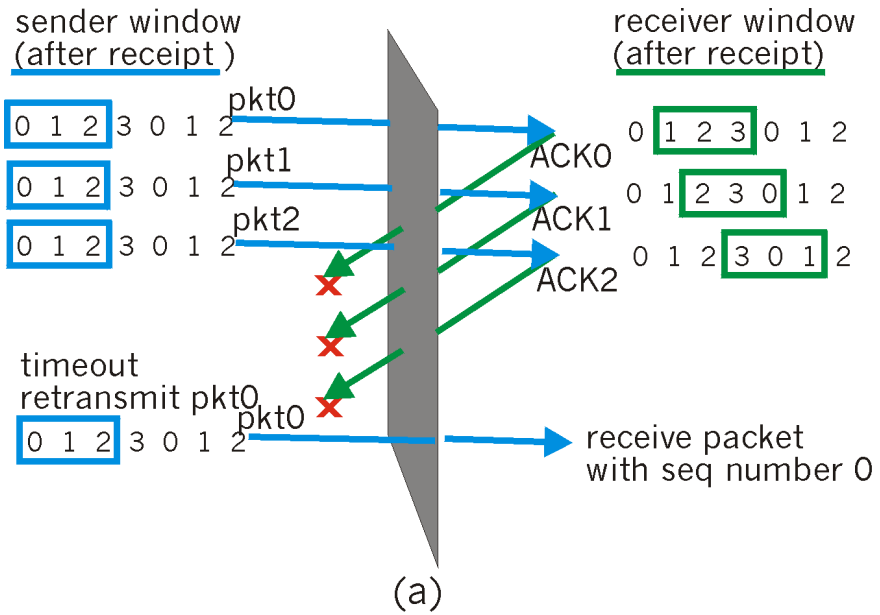
Selective repeat in action



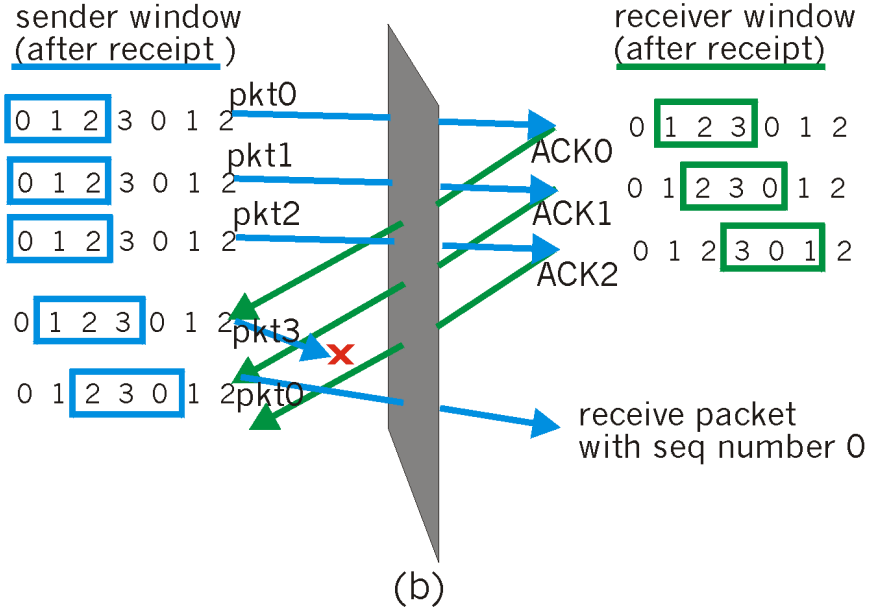
选择重传: 困难的问题

例子:

- 序号: 0, 1, 2, 3
- 窗口长度 = 3



- 接收方：在(a)和(b)两种情况下接收方没有发现差别!
- 在 (a)中不正确地将新的冗余的当为新的，而在(b)中不正确地将新的当作冗余的



问题: 序号长度与窗口长度有什么关系?

回答: 窗口长度小于等于序号空间的一半

请同学回答，这两种问题的场景是什么，带来了什么样的问题

基于第一次课的研讨题目

- 考虑一种仅使用否定确认的可靠数据传输协议。假定发送方只是偶尔发送数据。只用NAK的协议是否会比使用ACK的协议更好？为什么？现在我们假设发送方要发送大量的数据，并且该端到端连接很少丢包，在第二种情况下，只用NAK的协议是否会比使用ACK的协议更好，为什么？（胡蝶）

对题目的理解：

- 在有丢包的情况下，只使用NAK，发送方和接收方如何获知丢包？
- 偶尔发送数据理解为数据分组发送间隔时间较长，在检测丢包时会产生什么情况？

基于第一次课的研讨题目

- 为什么窗口长度必须小于等于序号空间的一半？

（黄锦发）

对题目的理解：

- 假设极端情况，发送窗口发出的包，接收方全部收到，但是返回的确认消息全部丢失，这时发送方的窗口和接收方的窗口包含的序号分别记录如下：

  [m-w, m-1] [m, m+w-1] 总共包括2w个包，这2w个包的序号是不应该有重叠的，因此2w<=N，这里N表示序号空间，如果有重叠会产生什么后果？

可靠数据传输机制及用途总结

机制	用途和说明
检验和	用于检测在一个传输分组中的比特错误。
定时器	用于检测超时/重传一个分组，可能因为该分组（或其ACK）在信道中丢失了。由于当一个分组被时延但未丢失（过早超时），或当一个分组已被接收方收到但从接收方到发送方的ACK丢失时，可能产生超时事件，所以接收方可能会收到一个分组的多个冗余拷贝。
序号	用于为从发送方流向接收方的数据分组按顺序编号。所接收分组的序号间的空隙可使该接收方检测出丢失的分组。具有相同序号的分组可使接收方检测出一个分组的冗余拷贝。
确认	接收方用于告诉发送方一个分组或一组分组已被正确地接收到了。确认报文通常携带着被确认的分组或多个分组的序号。确认可以是逐个的或累积的，这取决于协议。
否定确认	接收方用于告诉发送方某个分组未被正确地接收。否定确认报文通常携带着未被正确接收的分组的序号。
窗口、流水线	发送方也许被限制仅发送那些序号落在一个指定范围内的分组。通过允许一次发送多个分组但未被确认，发送方的利用率可在停等操作模式的基础上得到增加。我们很快将会看到，窗口长度可根据接收方接收和缓存报文的能力或网络中的拥塞程度，或两者情况来进行设置。

研讨课-2：TCP的数据可靠传输机制

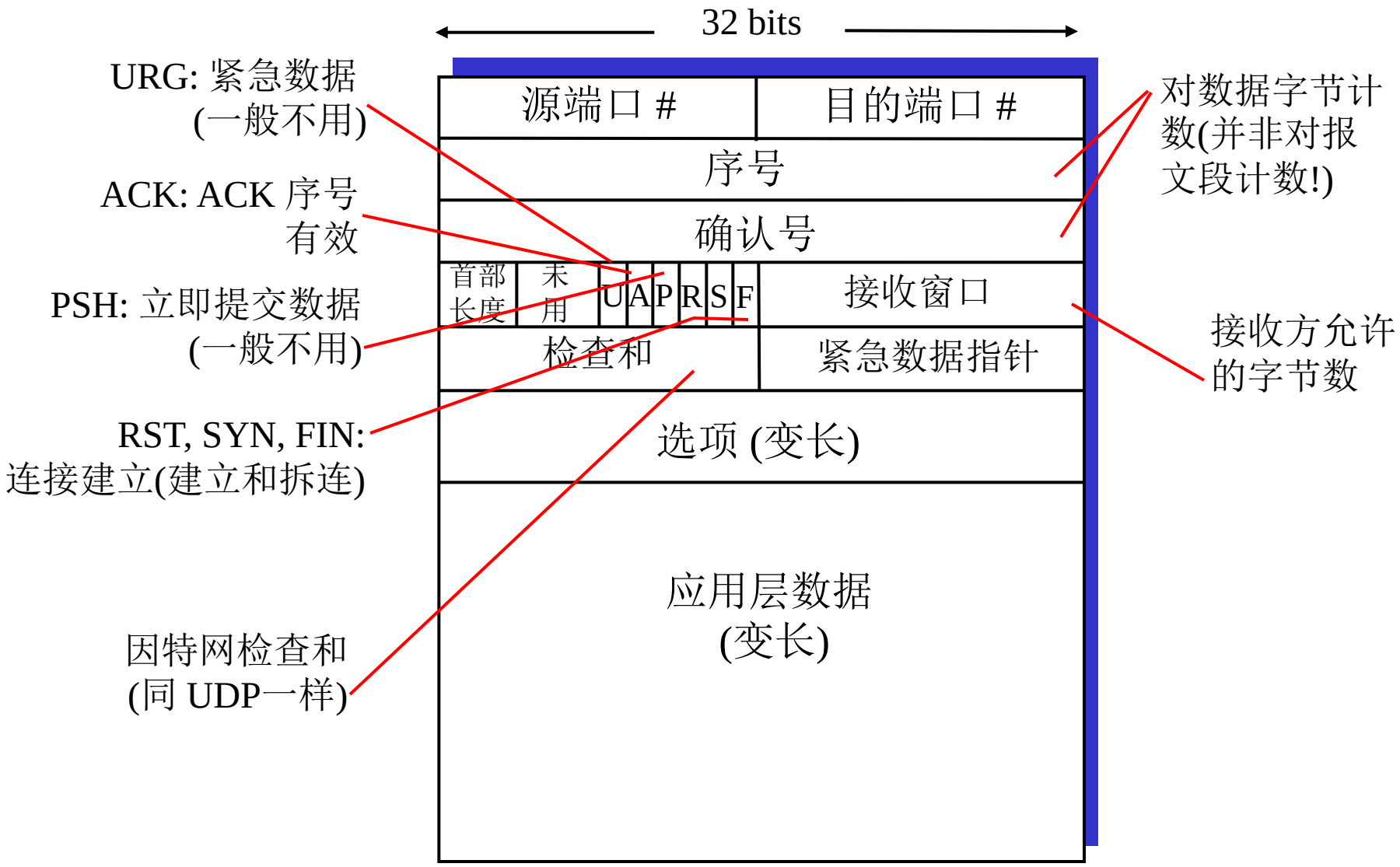
- TCP的可靠传输机制的设计
  - 首先请两名同学分别介绍你为TCP协议设计的可靠传输机制（刘鑫，可准备你自己的PPT）；
  - 包括如下内容：
    - 差错检查？
    - 序号？
    - 正确确认？
    - 否定确认？
    - 重传？
    - 定时器?
    - 流水线？GBN机制，还是SR机制，收发双方是否需要缓冲区？
    - 尝试画出发送方和接收方的有限状态机

研讨课-2：TCP的数据可靠传输机制

- **TCP的可靠传输机制的设计**
  - 请两名同学针对前面2名同学的设计进行点评或提问（张西源、夏梅娟）；
    - 差错检查？
    - 序号？
    - 正确确认？
    - 否定确认？
    - 重传？
    - 定时器?
    - 流水线？GBN机制，还是SR机制，收发双方是否需要缓冲区？

VINTON CERF, ROBERT KAHN因为设计了TCP协议获得了2004年的图灵奖！！！

(1) TCP 报文段结构


32 bits
URG: 紧急数据
(一般不用)
ACK: ACK 序号
有效
PSH: 立即提交数据
(一般不用)
RST, SYN, FIN:
连接建立(建立和拆连)
因特网检查和
(同 UDP一样)
源端口 #
目的端口 #
序号
确认号
首部长度
未用
U
A
P
R
S
F
接收窗口
检查和
紧急数据指针
选项 (变长)
应用层数据
(变长)
对数据字节计数(并非对报文段计数!)
接收方允许的字节数

TCP序号和确认号

- 序号：一个报文段的序号是其首字节的字节流编号。

例子：假设数据流由一个500,000的字节组成，每个段的数据为1000字节，则需要将数据流分为500个报文段，第1个报文段的序号为0，第2个报文段的序号为1000，如图

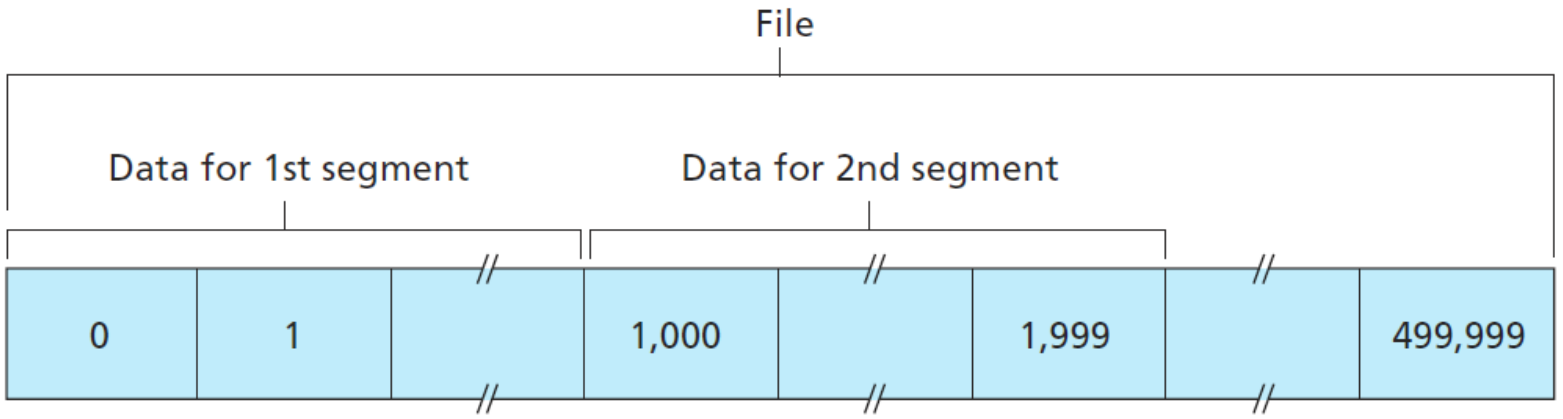


Figure 3.30 ♦ Dividing file data into TCP segments

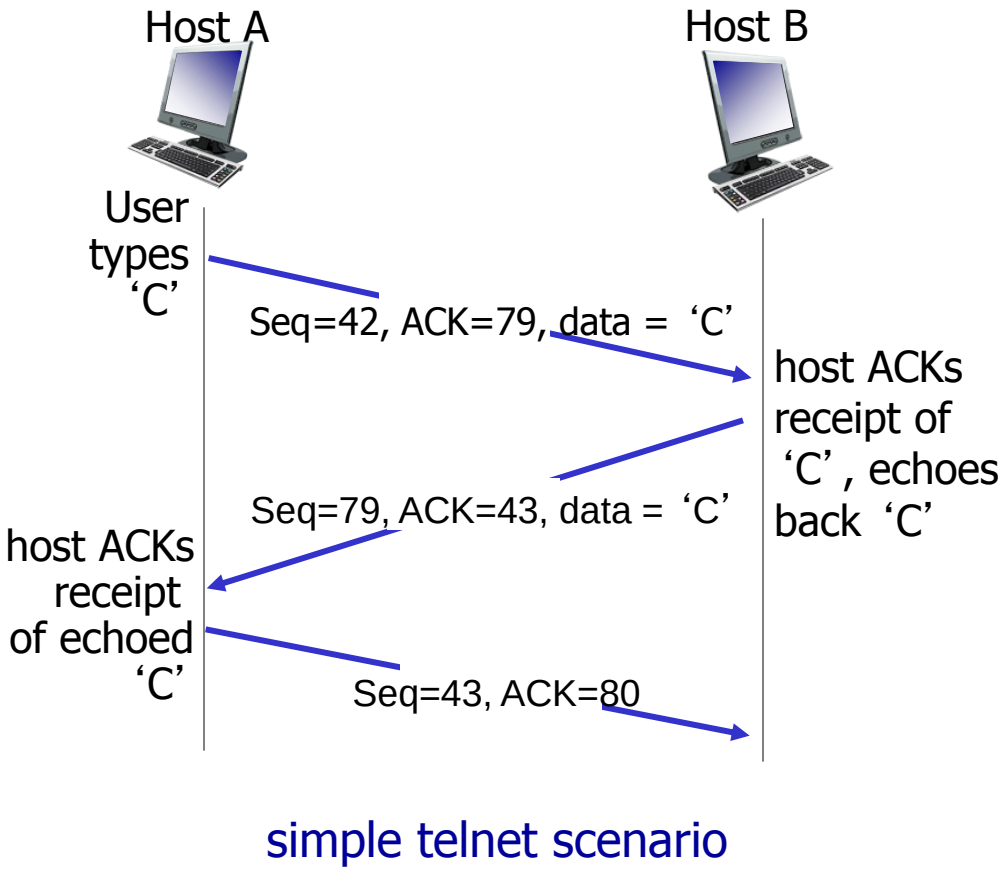
TCP序号和确认号

- 确认号：期望从对方收到下一个字节的序号
  - 累计确认机制：TCP只确认数据流中至第1个未收到字节为止的字节。

例子：假设主机A已经收到主机B的编号为0-535的所有字节，因此主机A发给主机B的确认包中的确认号为536.

什么是累计确认：假设主机A收到的字节为0-500,510-535，那么主机A发送给主机B的确认包中的确认号为501.

TCP seq. numbers, ACKs



simple telnet scenario

(2) TCP正确确认机制(ACK)

- **TCP的接收方的主要任务：**
  - **接收segment，判断是否损坏(校验和)；**
  - **对收到的segment进行ACK确认（累计确认）**
- **TCP中只有正确确认：**
  - **检测到segment损坏，不发生动作，不通知重传；**
  - **根据收到的segment的情况，通过发送不同的ACK(n)来通知接收方需要发送的segment；**

TCP ACK 产生 [RFC 1122, RFC 2581]

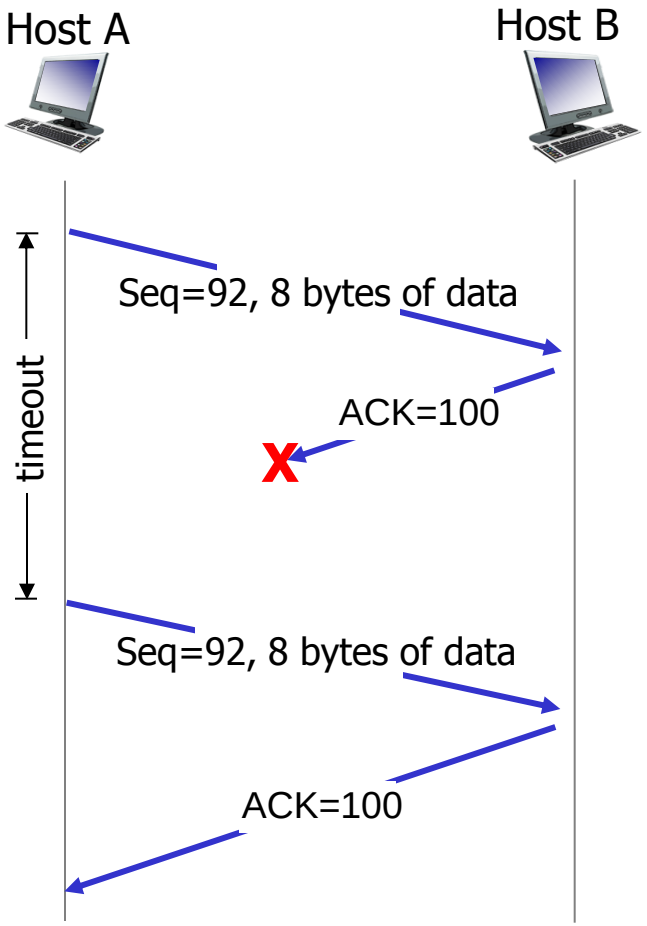
接收方事件	TCP 接收方行为
所期望序号的报文段按序到达。所有在期望序号及以前的数据都已经被确认	延迟的ACK。对另一个按序报文段的到达最多等待500 ms。如果下一个按序报文段在这个时间间隔内没有到达，则发送一个ACK
有期望序号的报文段按序到达。另一个报文段等待发送ACK	立即发送单个累积ACK，以确认两个按序报文段
比期望序号大的失序报文段到达，检测出数据流中的间隔。	立即发送冗余ACK，指明下一个期待字节的序号（也就是间隔的低端字节序号）
部分或者完全填充已接收到数据间隔的报文段到达	倘若该报文段起始于间隔的低端，则立即发送ACK

(3) 发送方的重传机制

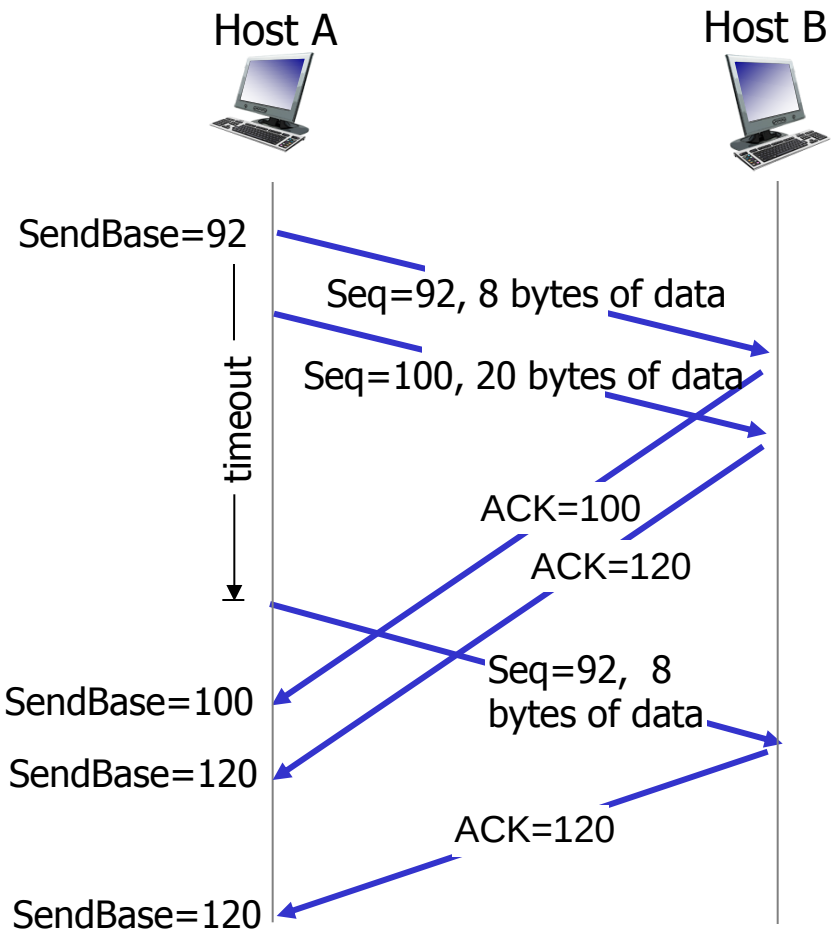
- **TCP的发送方的主要任务：**
  - **发送数据报；**
  - **根据收到的ACK确认，判断是否启动重传机制；**
- **在什么情况下，启动重传(判断数据报丢失)：**
  - **定时器超时，则认为对应的数据报丢失，立即重传；**
  - **收到3个相同的ACK(n)确认，则认为对应的数据报丢失，立即重传；**

首先看定时器超时，引起重传的几种场景。。。
再分析3个相同ACK确认是什么情况。。。

TCP: retransmission scenarios

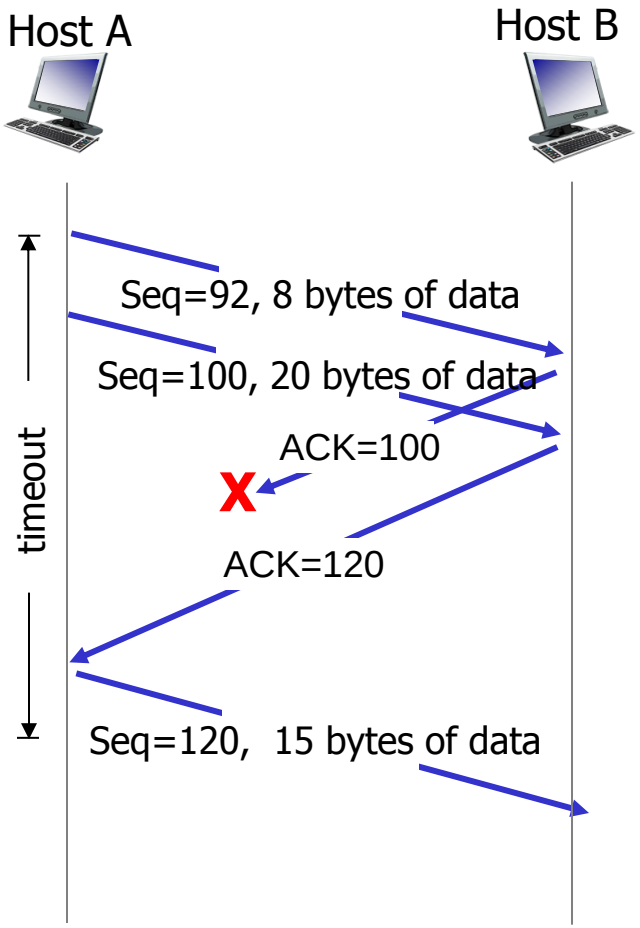


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK

TCP fast retransmit

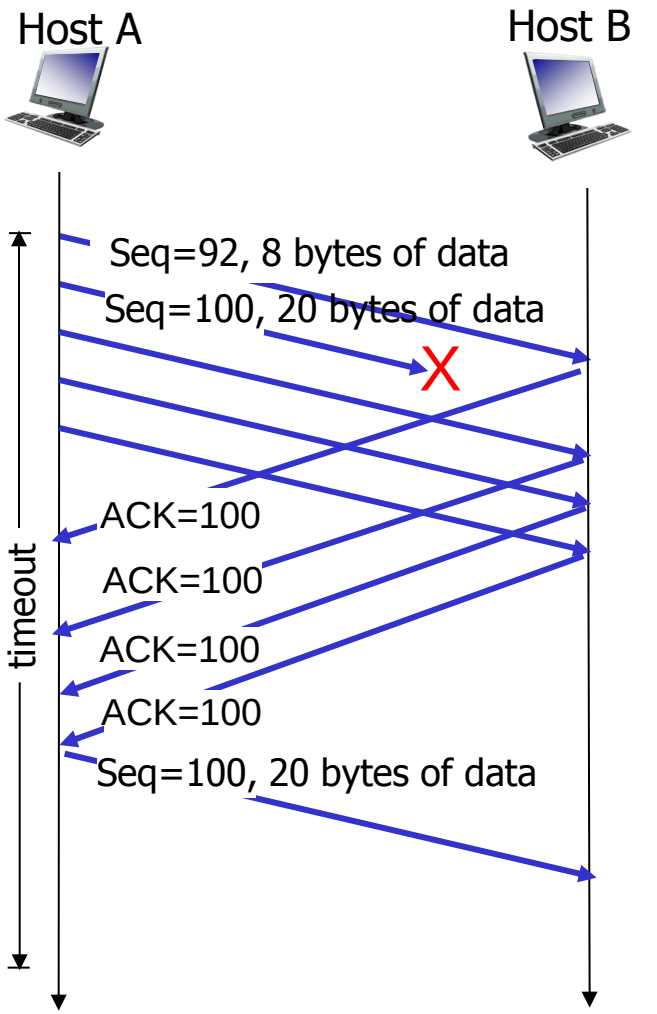
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #

- likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

(4) TCP的定时器机制

- **TCP的发送方只维护一个定时器(timer)；**
  - 启动：每当发送一个报文且这个报文是最早的发送且还未确认的，则启动定时器；
  - 终止/重启：
    - 当收到确认且这个确认是最早的发送且还未确认的报文段的确认，则终止定时器，且将发送窗口向前移动，并对新的最早发送且未确认的报文段启动定时器；
  - 超时后的动作：重传定时器对应的数据段，并对重传的数据段重启定时器；
  - 超时时间间隔的合理设置：
    - 设置什么值更为合理？
    - 如果再次超时，这个时间间隔是否需要调整？

    （随机请一位同学来回答）

TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- ***too short***: premature timeout, unnecessary retransmissions
- ***too long***: slow reaction to segment loss

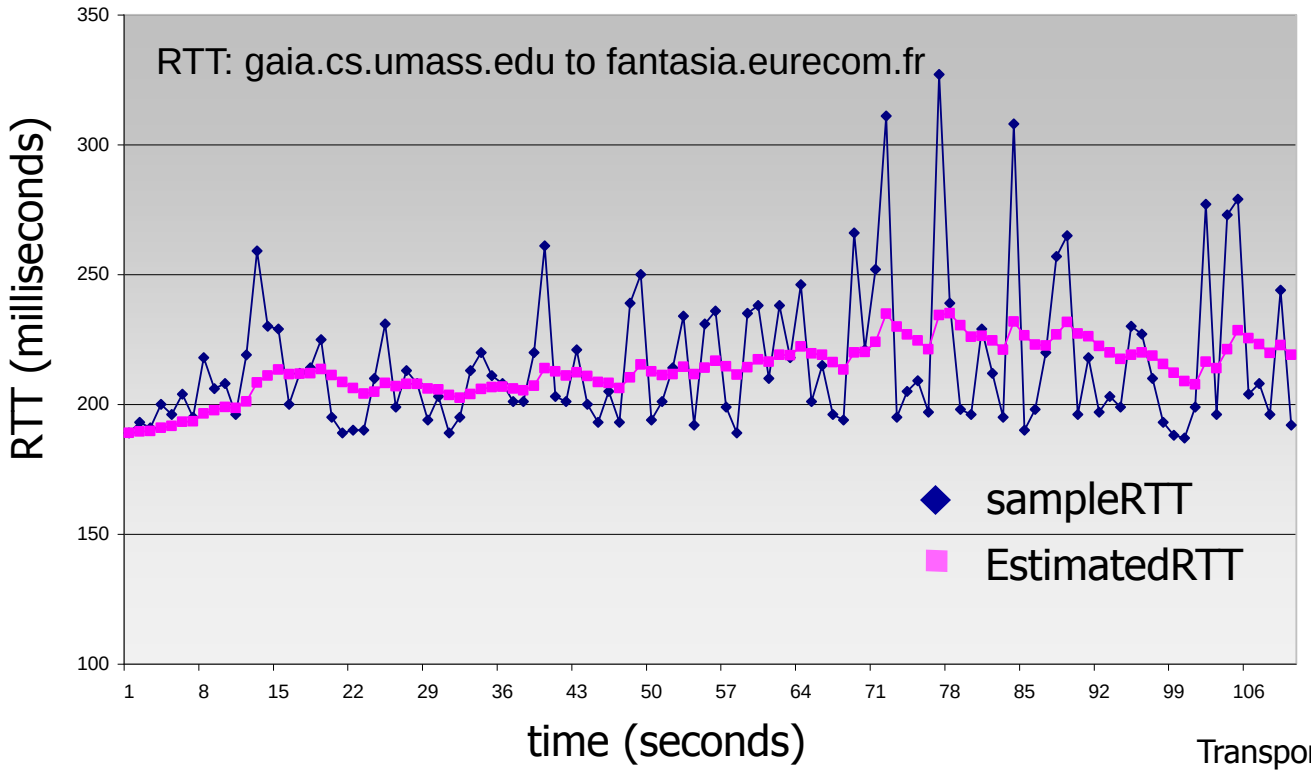
Q: how to estimate RTT?

- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several *recent* measurements, not just current **`SampleRTT`**

TCP round trip time, timeout

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

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time, timeout

- timeout interval: **`EstimatedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`** -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

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

(typically, $\beta = 0.25$)

$$\texttt{TimeoutInterval} = \texttt{EstimatedRTT} + 4*\texttt{DevRTT}$$

estimated RTT

"safety margin"

(5) TCP流水线机制

- **TCP采用了一种GBN和SR的混合机制**
  - **累计确认**
  - **单个定时器**
  - **将失序达到的报文段缓存**
  - **每次重传只发送一个报文段(无论是定时器超时，还是收到3个重复ACK)**

总结：TCP 发送方事件

1.从应用层接收数据:

- 根据序号创建报文段
- 序号是报文段中第一个数据字节的数据流编号
- 如果未启动，启动计时器(考虑计时器用于最早的没有确认的报文段)
- 超时间隔:

```
TimeOutInterval=
EstimatedRTT +
4*DevRTT
```

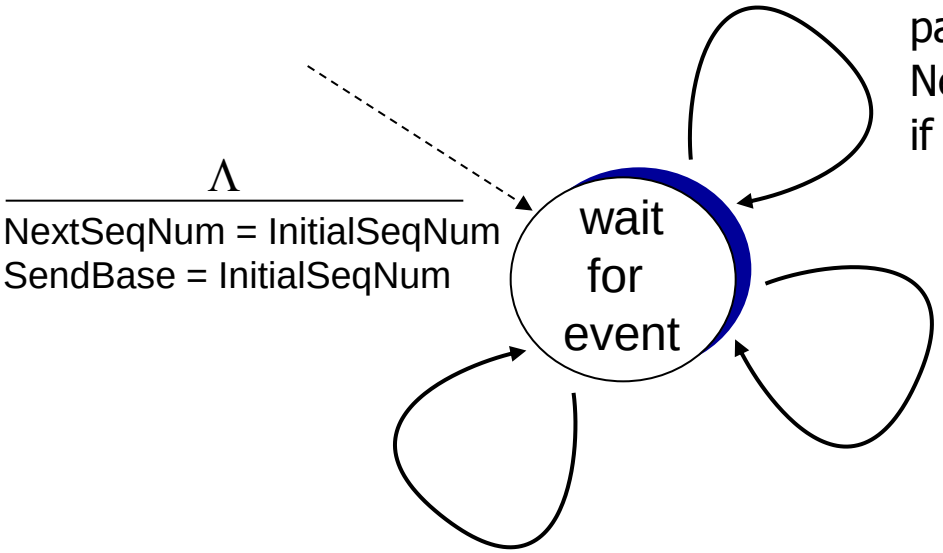
2.超时:

- 重传导致超时的报文段
- 重新启动计时器

3.收到确认：

- 如果确认了先前未被确认的报文段
  - 更新被确认的报文段序号
  - 如果还有未被确认的报文段，重新启动计时器
- 如果收到冗余的对已确认的报文段的确认

TCP sender (simplified)（无快速重传）



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
    }
```

开放题目

- 目前TCP和UDP都没有提供任何加密机制，使得用户名和密码等隐私数据使用明文传输，导致网络链路上可以嗅探到分组中的隐私数据，为此引入了一个新的技术，安全套接字层(SSL)，请同学简单介绍SSL机制的过程，可以结合Wireshark抓包来进行讲解。(**黎桐**、**林勇聪**)
- QUIC(Quick UDP Internet Connections)是由Google提出的一种基于UDP改进的低时延的互联网传输层协议。其目标是取代现有的TCP协议，请同学对QUIC协议做简单的介绍，以及运行一个简单的配置，介绍个人体验。

课堂练习

- R12、P27、P31

R12. Visit the Go-Back-N Java applet at the companion Web site.

a. Have the source send five packets, and then pause the animation before any of the five packets reach the destination. Then kill the first packet and resume the animation. Describe what happens.

b. Repeat the experiment, but now let the first packet reach the destination and kill the first acknowledgment. Describe again what happens.

c. Finally, try sending six packets. What happens?

课堂练习

P27. Host A and B are communicating over a TCP connection, and Host B has already received from A all bytes up through byte 126. Suppose Host A then sends two segments to Host B back-to-back. The first and second segments contain 80 and 40 bytes of data, respectively. In the first segment, the sequence number is 127, the source port number is 302, and the destination port number is 80. Host B sends an acknowledgment whenever it receives a segment from Host A.

a. In the second segment sent from Host A to B, what are the sequence number, source port number, and destination port number?

b. If the first segment arrives before the second segment, in the acknowledgment of the first arriving segment, what is the acknowledgment number, the source port number, and the destination port number?

c. If the second segment arrives before the first segment, in the acknowledgment of the first arriving segment, what is the acknowledgment number?

课堂练习

d. Suppose the two segments sent by A arrive in order at B. The first acknowledgment is lost and the second acknowledgment arrives after the first timeout interval. Draw a timing diagram, showing these segments and all other segments and acknowledgments sent. (Assume there is no additional packet loss.) For each segment in your figure, provide the sequence number and the number of bytes of data; for each acknowledgment that you add, provide the acknowledgment number.

P31. Suppose that the five measured `SampleRTT` values (see Section 3.5.3) are 106 ms, 120 ms, 140 ms, 90 ms, and 115 ms. Compute the `EstimatedRTT` after each of these SampleRTT values is obtained, using a value of $\alpha = 0.125$ and assuming that the value of `EstimatedRTT` was 100 ms just before the first of these five samples were obtained. Compute also the `DevRTT` after each sample is obtained, assuming a value of $\beta = 0.25$ and assuming the value of `DevRTT` was 5 ms just before the first of these five samples was obtained. Last, compute the TCP `TimeoutInterval` after each of these samples is obtained.