计算机网络

第3章 传输层协议

吴英

wuying@nankai.edu.cn

内容提纲

3.1 传输层需要解决的基本问题

3.2 TCP/IP体系结构中传输层协议与服务

3.3 用户数据报协议（UDP）

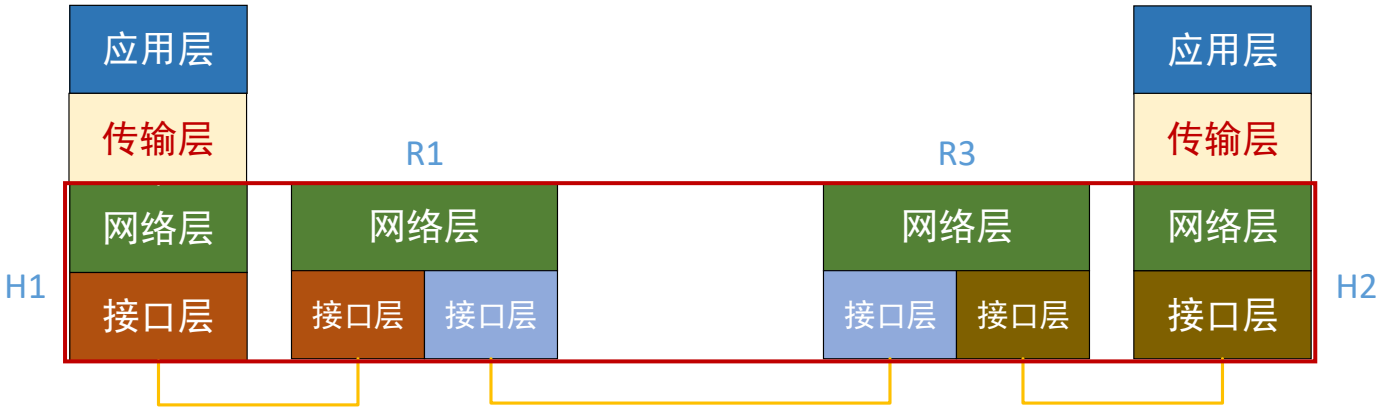
3.4 可靠数据传输

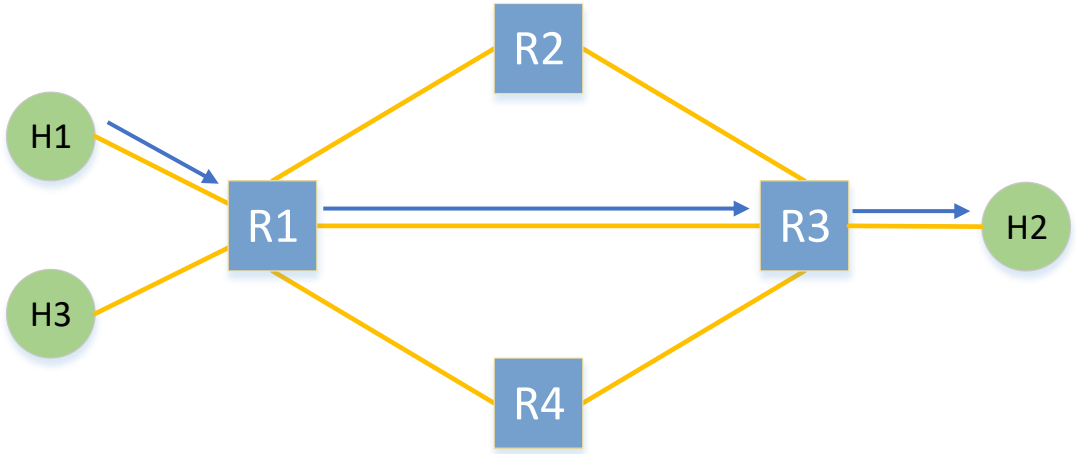
3.5 传输控制协议（TCP）

3.6 理解网络拥塞

3.7 TCP拥塞控制机制

3.1 传输层需要解决的基本问题

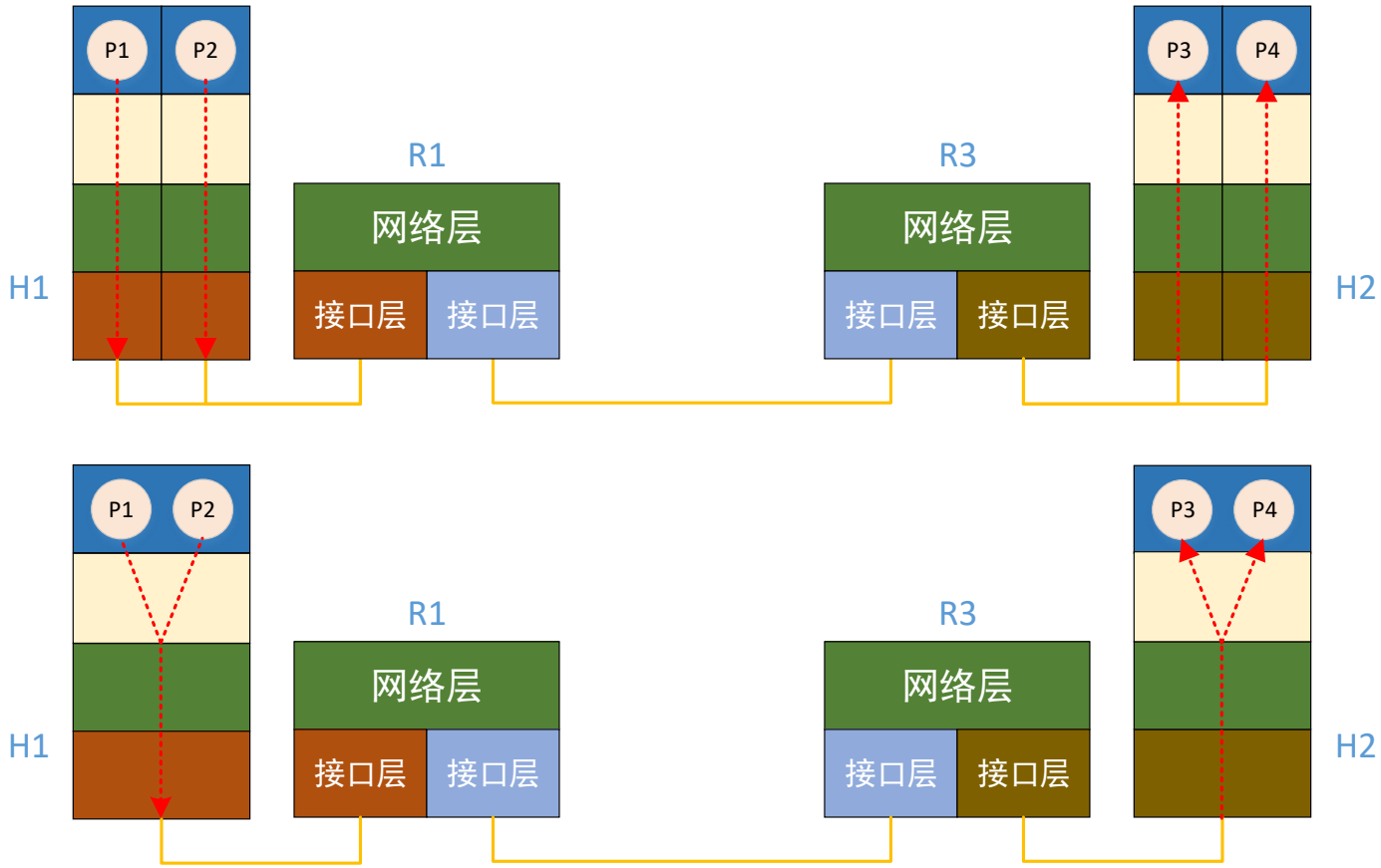




- 网络层：将IP数据包从源主机传送到目的主机，提供无连接不可靠服务
- **数据包传输存在的问题：** 延迟、乱序、出错、丢失等
- **传输层解决的问题之一：** 可靠性，向应用层提供可靠服务

3.1 传输层需要解决的基本问题

- 应用层运行多个应用进程



- 共享单一的网络层协议（IP）和网络接口
- **传输层解决的问题之二：**复用（Multiplexing）和分用（Demultiplexing）

3.2 TCP/IP体系结构中传输层协议与服务

- 传输层协议的基本功能
  - 复用和分用
  - 可靠性保证
- 传输层实体执行的动作
  - 发送端：将应用层的消息**封装**成传输层的数据单元，传递到网络层
  - 接收端：将从网络层接收的传输层数据单元，**处理**后交给应用层
- 传输控制协议TCP（Transport Control Protocol）
  - 为进程间通信提供面向连接的、可靠的传输服务
  - 实现复用分用、差错检测、确认重传、流量控制等传输层功能
- 用户数据报协议UDP（User Datagram Protocol）
  - 为进程间通信提供非连接的、不可靠的传输服务
  - 实现复用分用、差错检测等传输层功能

3.3 用户数据报协议UDP

RFC 768

UDP协议特点

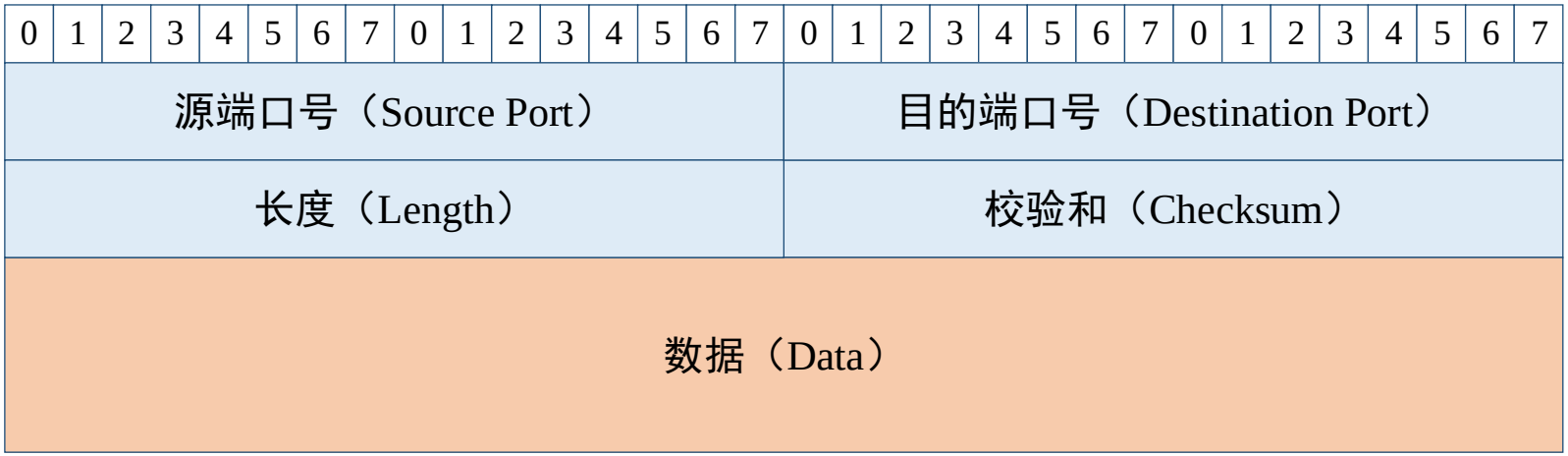
- 发送方和接收方不需要握手过程
- 每个UDP数据单元（**数据报**）独立传输
- 提供复用分用功能和**可选**的差错检测功能
- 支持组播通信（点到多点通信）
- 不提供可靠性保证：无确认重传、可能有出错、丢失、乱序等现象

3.3 用户数据报协议UDP

RFC 768

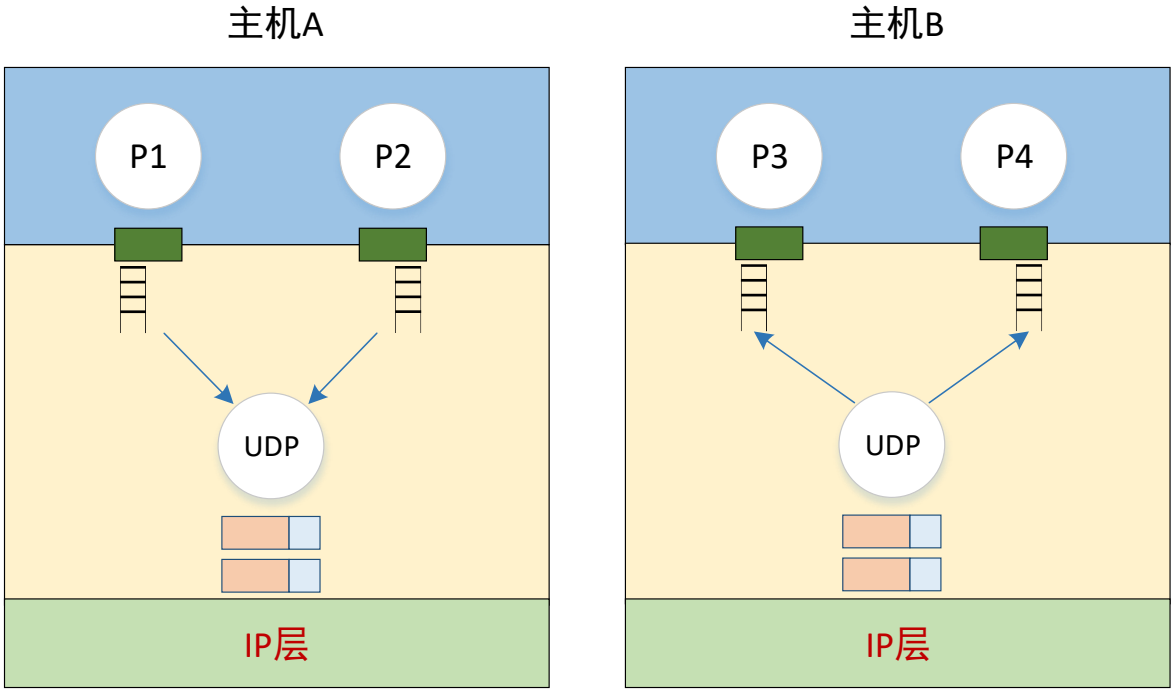
UDP数据报格式

- 长度：包含头部、以字节计数
- 校验和：为可选项，用于差错检测



3.3 用户数据报协议UDP

- UDP的复用和分用



- 进程标识：目的IP地址+目的端口号
- 例如：P1与P3通信，P2与P4通信，P1使用端口6000， P2使用端口7000，P3使用端口6000， P4使用端口8000

3.3 用户数据报协议UDP

UDP数据报的差错检测

- 可选项，利用数据报中携带冗余位（校验和域段）来检测数据报传输过程中出现的差错
- 发送端：利用自己产生的伪首部和发送的UDP数据报计算校验和
- 接收端：利用自己产生的伪首部和接收的UDP数据报计算校验和
- 伪首部：包含源IP地址、目的IP地址、协议类型等域段

<table>
<tr><td colspan="32">0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7</td></tr>
<tr><td colspan="32">源IP地址（Source IP address）</td></tr>
<tr><td colspan="32">目的IP地址（Destionation IP address）</td></tr>
<tr><td colspan="8">0</td><td colspan="8">协议（Protocol）</td><td colspan="16">长度（Length）</td></tr>
</table>

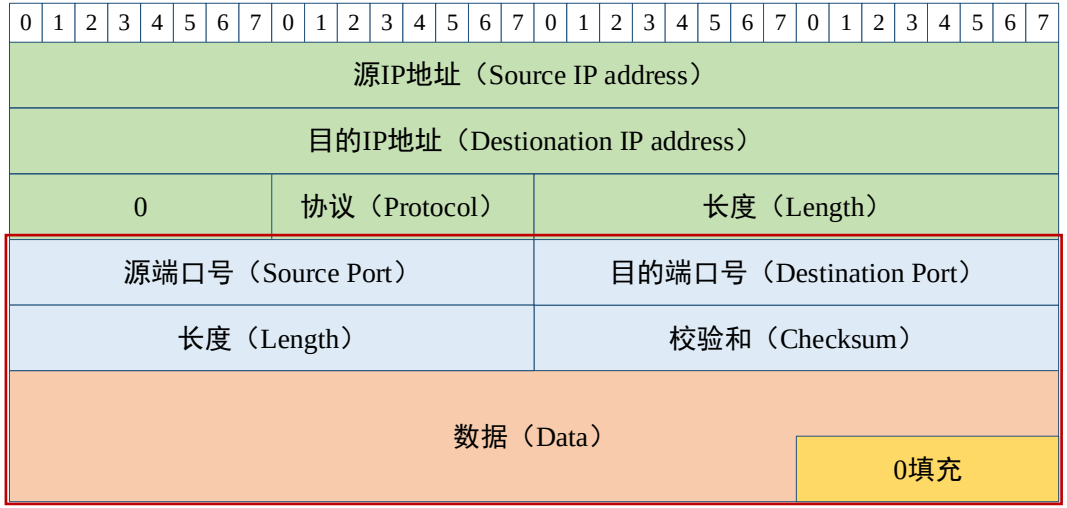
UDP校验和的计算方法

发送端：

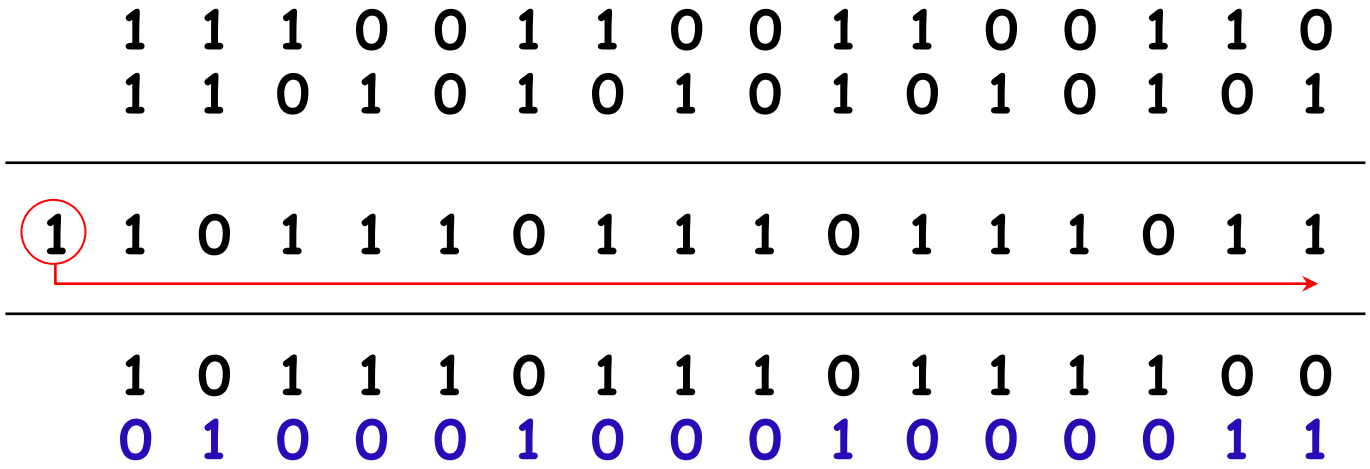
- 产生伪首部，校验和域段清0，将数据报用0补齐为16位整数倍
- 将伪首部和数据报一起看成16位整数序列
- 进行 16 位二进制反码求和运算，计算结果取反写入校验和域段

接收端：

- 产生伪首部，将数据报用0补齐为16为整数倍
- 按16位整数序列，采用 16 位二进制反码求和运算
- 如果计算结果位全1，没有检测到错误；否则，说明数据报存在差错



- TCP/IP校验和计算方法示例



计算 UDP校验和示例

伪首部	153.19.8.104			
	171.3.14.11			
	0	17	15	
UDP首部	1087		13	
	15		0	
数据	01010100	01000101	01010011	01010100
	01001001	01001110	01000111	0填充

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
将得出的结果求反码

10010110 11101101 → 求和得出的结果
01101001 00010010 → 校验和

3.3 用户数据报协议UDP

- UDP校验和计算伪码

```
u_short cksum(u_short *buf, int count)
  {
    register u_long sum = 0;
     while (count--)
     {
          sum += *buf++;
          if (sum & 0XFFFF0000)
          {
               sum &= 0XFFFF;
               sum++;
          }
     }
     Return ~(sum & 0XFFFF);
  }
```

单选题 5分

设置

假设要发送的数据D为11001010，采用校验和的计算方法，将D划分成两个4位二进制（1100和1010）进行运算，产生4位校验和，放在数据D后面一起传输，则传输的数据为：

A 110010100111

B 110010101000

C 110010100110

D 110010101001

提交

UDP校验和计算几点说明

- IPv4中UDP校验和是可选项，IPv6中变成强制项
  - 0无，非0有（如果计算结果为0，则以全1代替）
- UDP校验和覆盖的范围超出了UDP数据报本身，使用伪首部的目的是检验UDP数据报是否到达真正的目的地
  - 正确的目的地包括了特定的主机和该主机上特定的端口
- 伪首部不随用户数据报一起传输，接收方需自己形成伪首部进行校验
- 伪首部的使用破坏了层次划分的基本前提，即每一层的功能独立
  - 目的主机的IP地址UDP通常知道，源IP的使用需要通过路由选择决定

IP首部、ICMP、UDP、TCP都需要计算校验和，方法类似

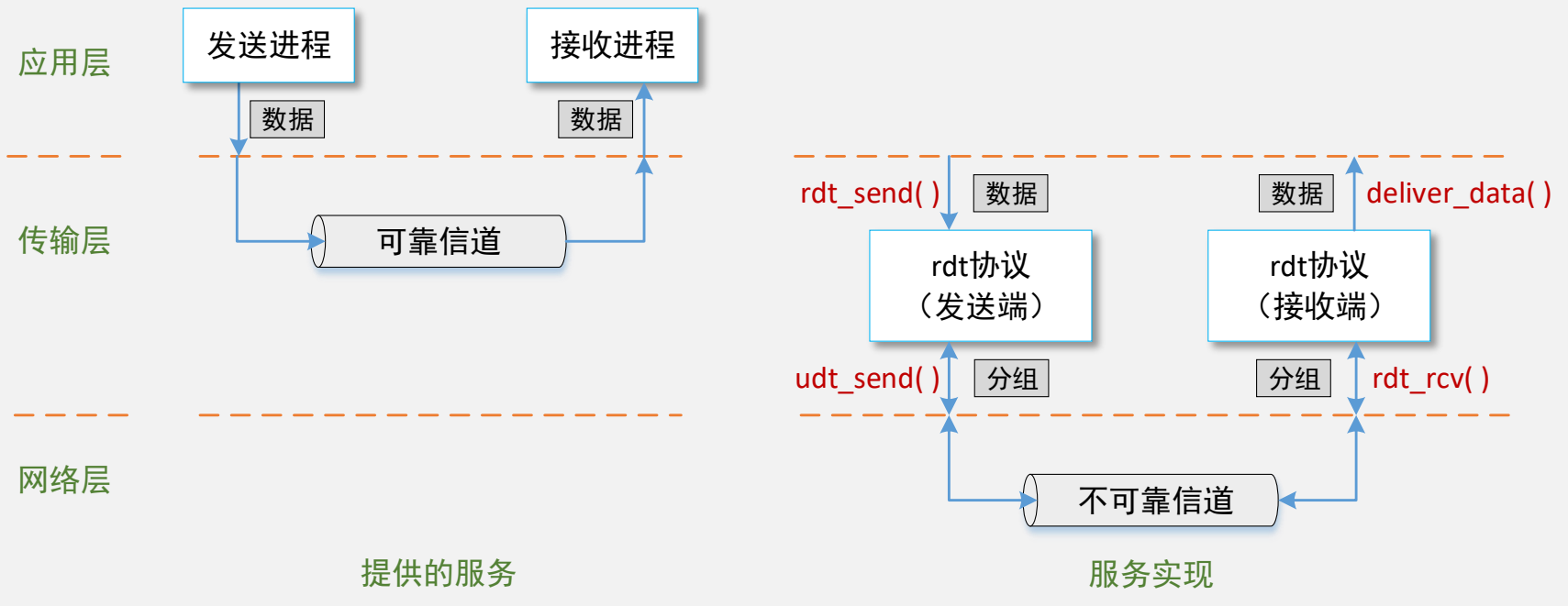
使用UDP服务的应用

- 流媒体应用（实时音频和视频）通常使用UDP服务
  - 能够容忍一定的丢失
  - 对时延敏感
- 其他使用UDP服务的应用，如：
  - DNS
  - SNMP
- 需要在UDP之上实现可靠传输，即在应用层增加可靠机制

为什么提供UDP服务？

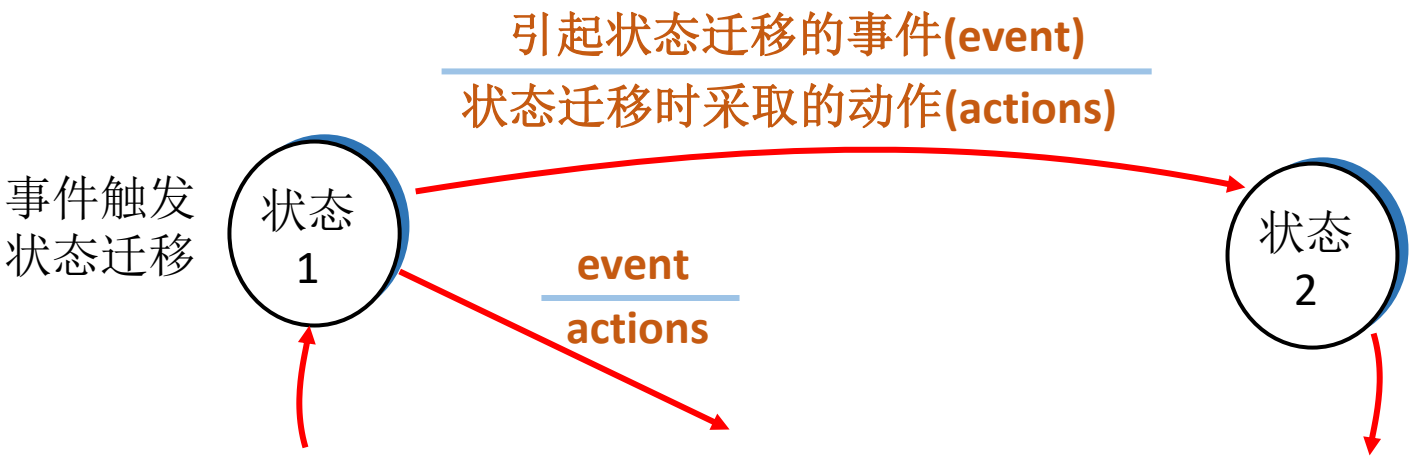
- 不需要建立连接，建立连接需要增加延时，特别对于简单的交互应用
- 协议简单：在发送端和接收端不需要维护连接状态
- 数据报头部短，额外开销小
- 无拥塞控制

- 可靠传输基本原理
  - 可靠传输在多个层次有所涉及，如应用层、传输层、链路层等
  - 计算机网络中的Top 10问题



rdt：reliable data transfer protocol（可靠数据传输协议）

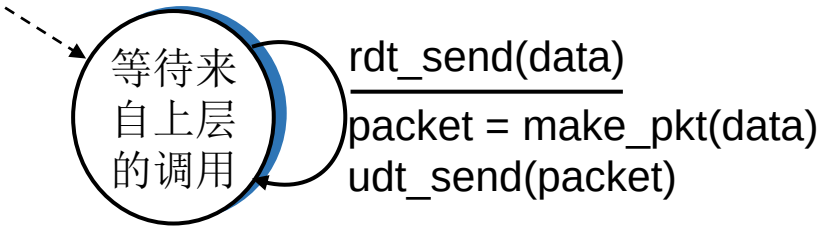
- 可靠传输基本原理
  - 考虑可靠协议的复杂性，采用渐进的方法
  - 考虑单向数据传输，控制信息可以双向传输
  - 使用有限状态机（FSM）描述发送端和接收端的状态和状态迁移



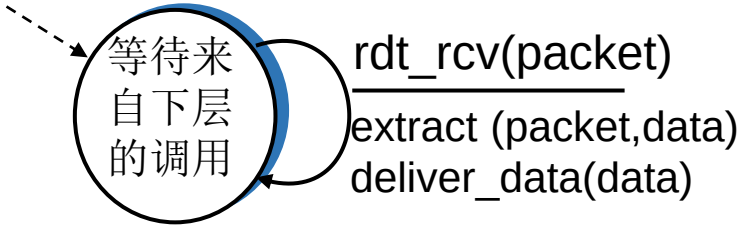
FSM：Finite State Machines

3.4 可靠数据传输

- 完全可靠通道上的可靠数据传输：rdt1.0
  - 下层通道是完全可靠的
    - 无位错误
    - 无分组丢失



发送端



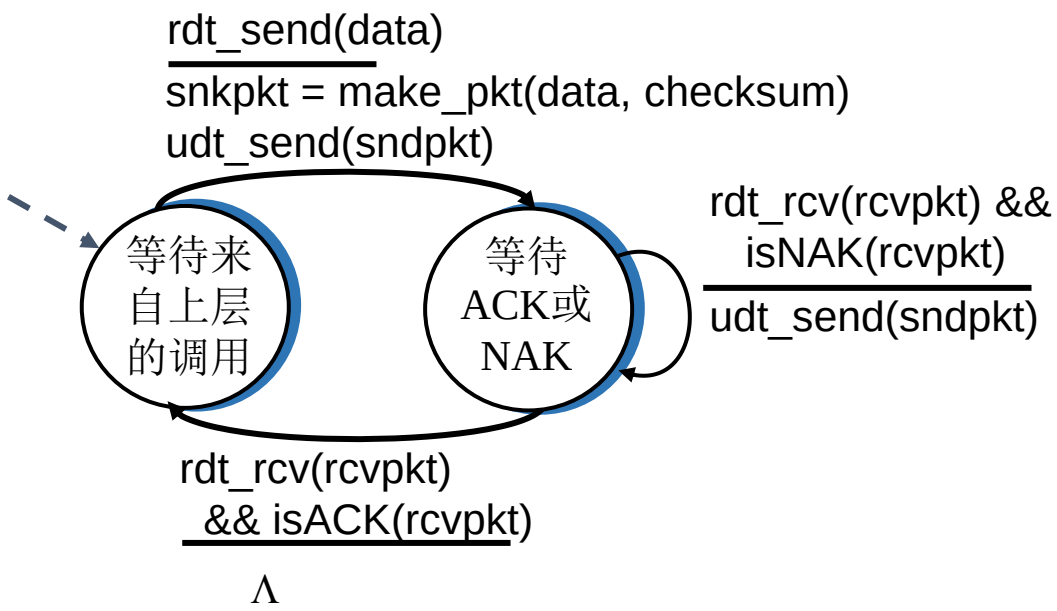
接收端

3.4 可靠数据传输

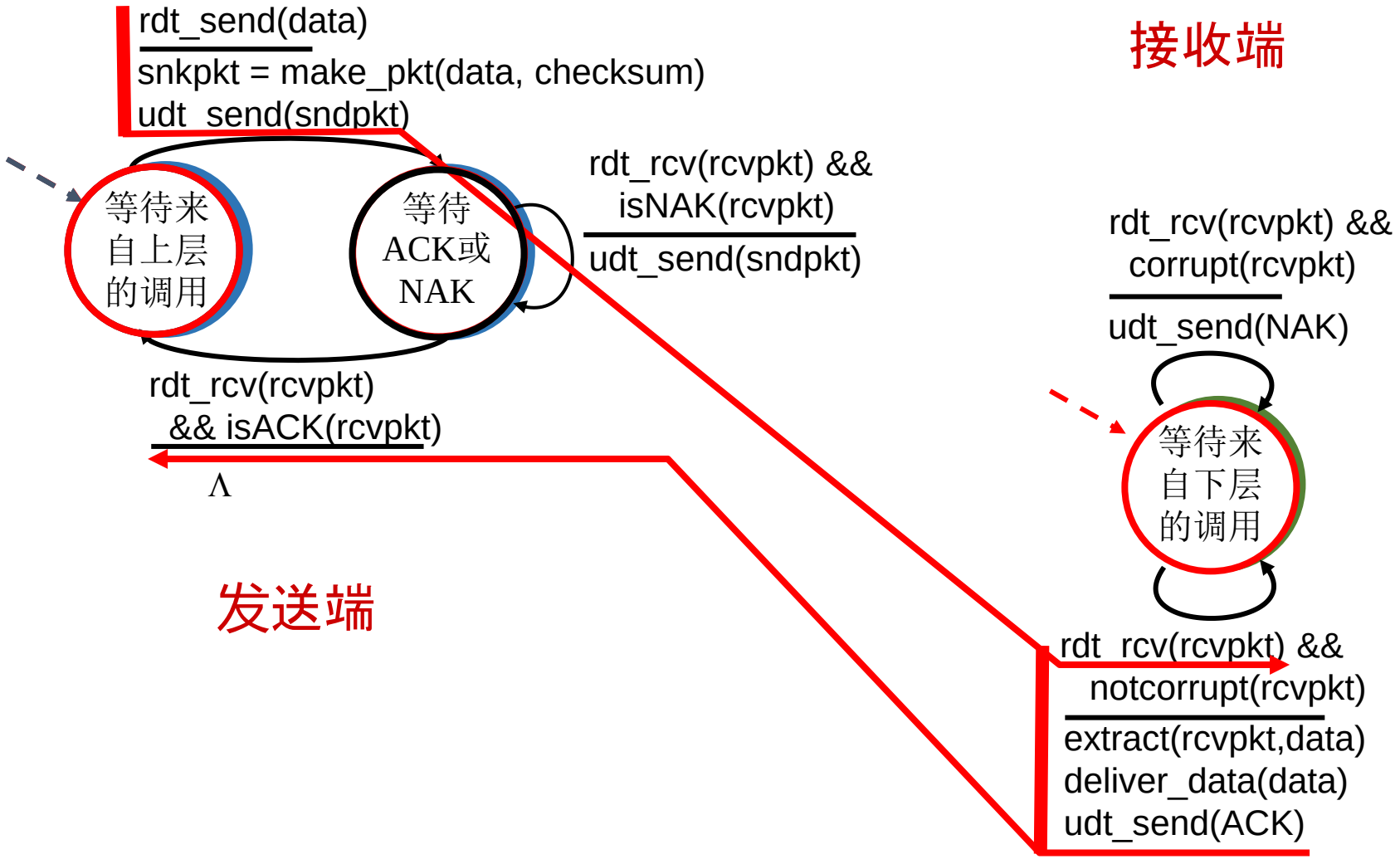
- 具有位错误通道上的可靠数据传输：rdt2.0
  - 下层通道可能造成某些位出现错误（如：1变0，0变1）
  - **需要解决的问题**：如何恢复差错（自动重传请求ARQ）
    - ACK：接收端通知发送端分组正确接收
    - NAK：接收端通知发送端接收的分组存在错误
    - 发送端收到NAK，则重传分组
  - **rdt2.0需要增加的功能**
    - 差错检测
    - 接收端反馈：ACK或NAK（控制分组）
    - 发送端重传分组

ARQ：Automatic Repeat ReQuest

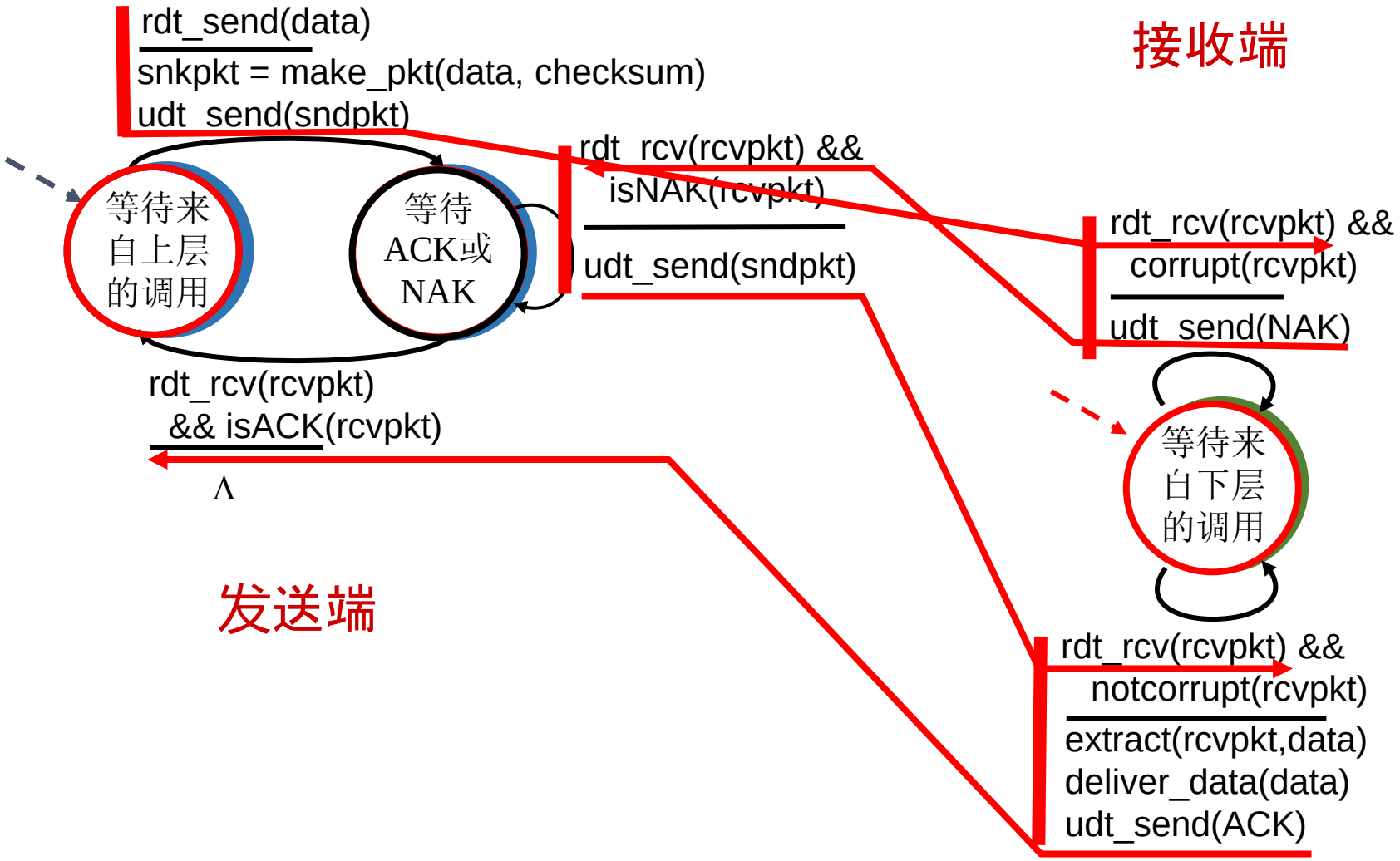
- rdt2.0：有限状态机


rdt_send(data)
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)
等待来自上层的调用
等待ACK或NAK
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ
发送端
接收端
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)
等待来自下层的调用
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

- rdt2.0：无差错情况


rdt_send(data)
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)
等待来自上层的调用
等待ACK或NAK
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ
发送端
接收端
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)
等待来自下层的调用
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

- rdt2.0：有差错情况


rdt_send(data)
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)
等待来自上层的调用
等待ACK或NAK
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ
发送端
接收端
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)
等待来自下层的调用
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt2.0：存在的问题

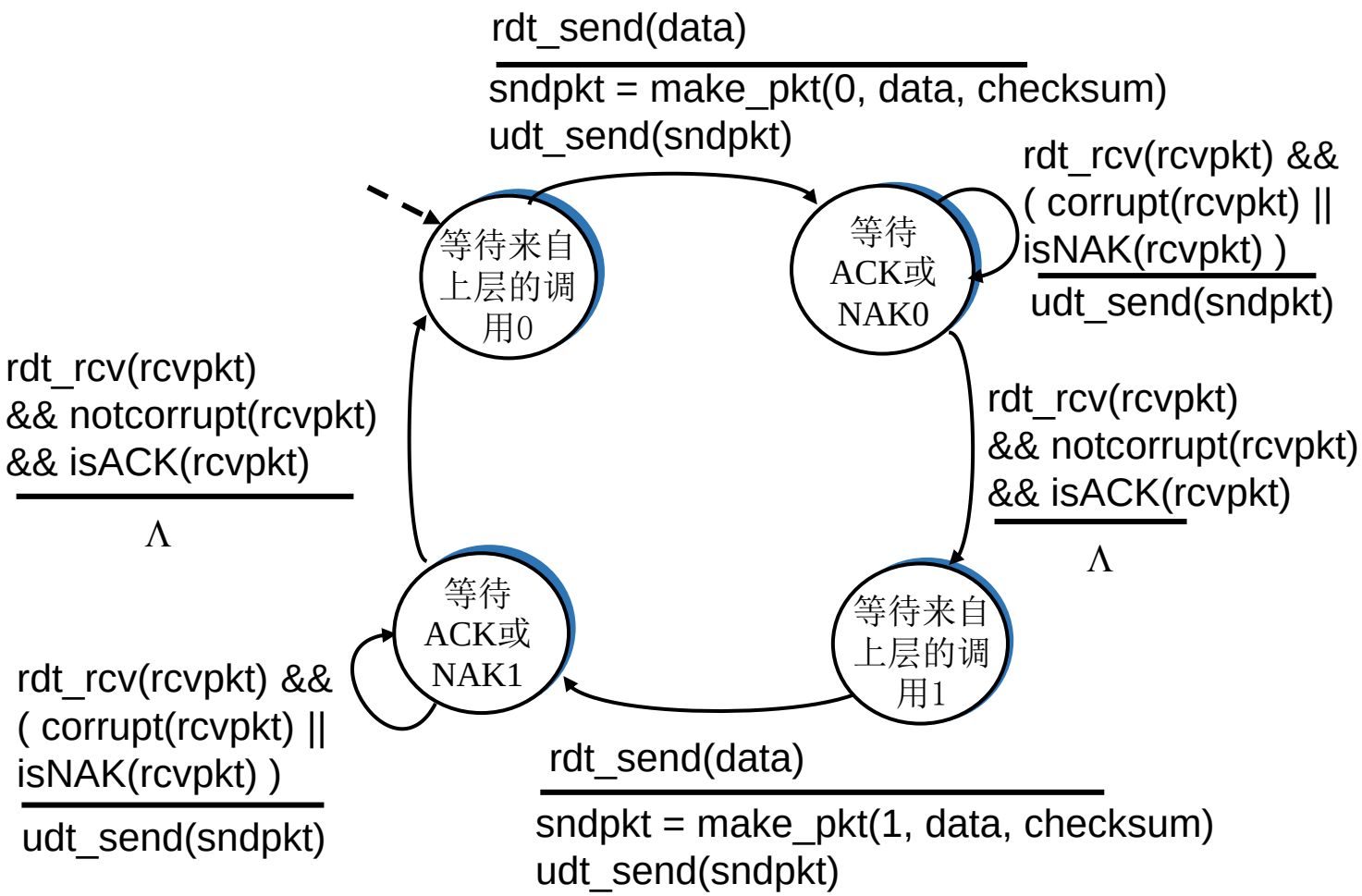
- 如果ACK/NAK受损会产生什么状况？
  - 发送端无法确认接收端的状况（ACK或NAK）
  - 不能简单进行重传：可能会造成**重复接收**
- **处理重复接收问题**
  - 发送端在每个分组中**增加序列号**
  - 如果无法判断是ACK或NAK，则重传当前的分组
  - 接收端丢弃重复的分组

rdt2.1：解决rdt2.0的问题

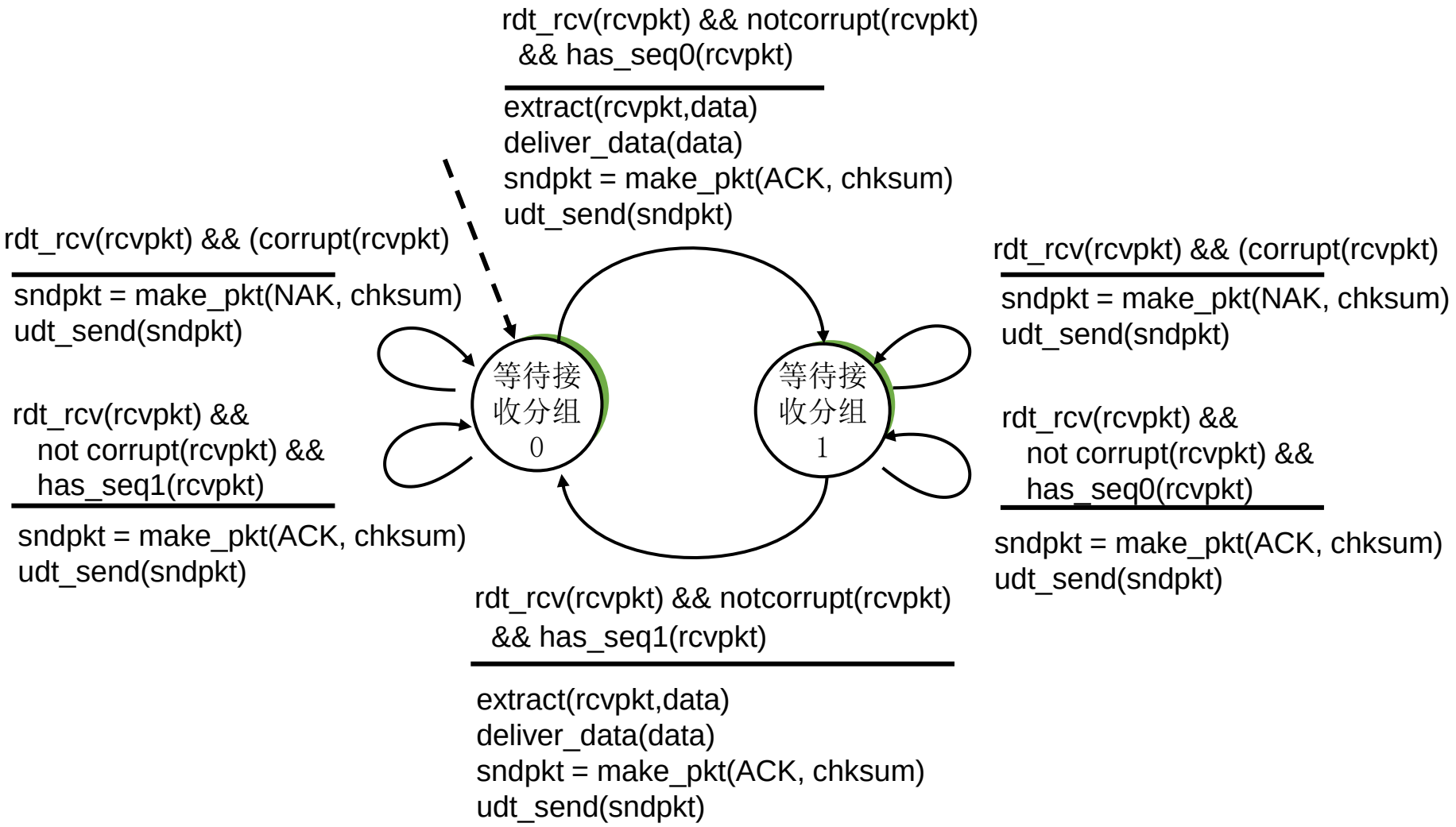
- 在rdt2.0基础上需要增加哪些功能？
  - 发送端在每个分组中**增加序列号** **使用几个序列号够用？**
  - 发送端通过校验字段验证ACK/NAK分组是否损坏
  - 如果ACK/NAK分组损坏，发送端重传当前的分组
  - 接收端根据序列号判断是否是重复的分组
  - 接收端在ACK/NAK分组中增加校验字段

停等机制：发送端发送一个分组，然后等待接收端响应

- rdt2.1：发送端状态机

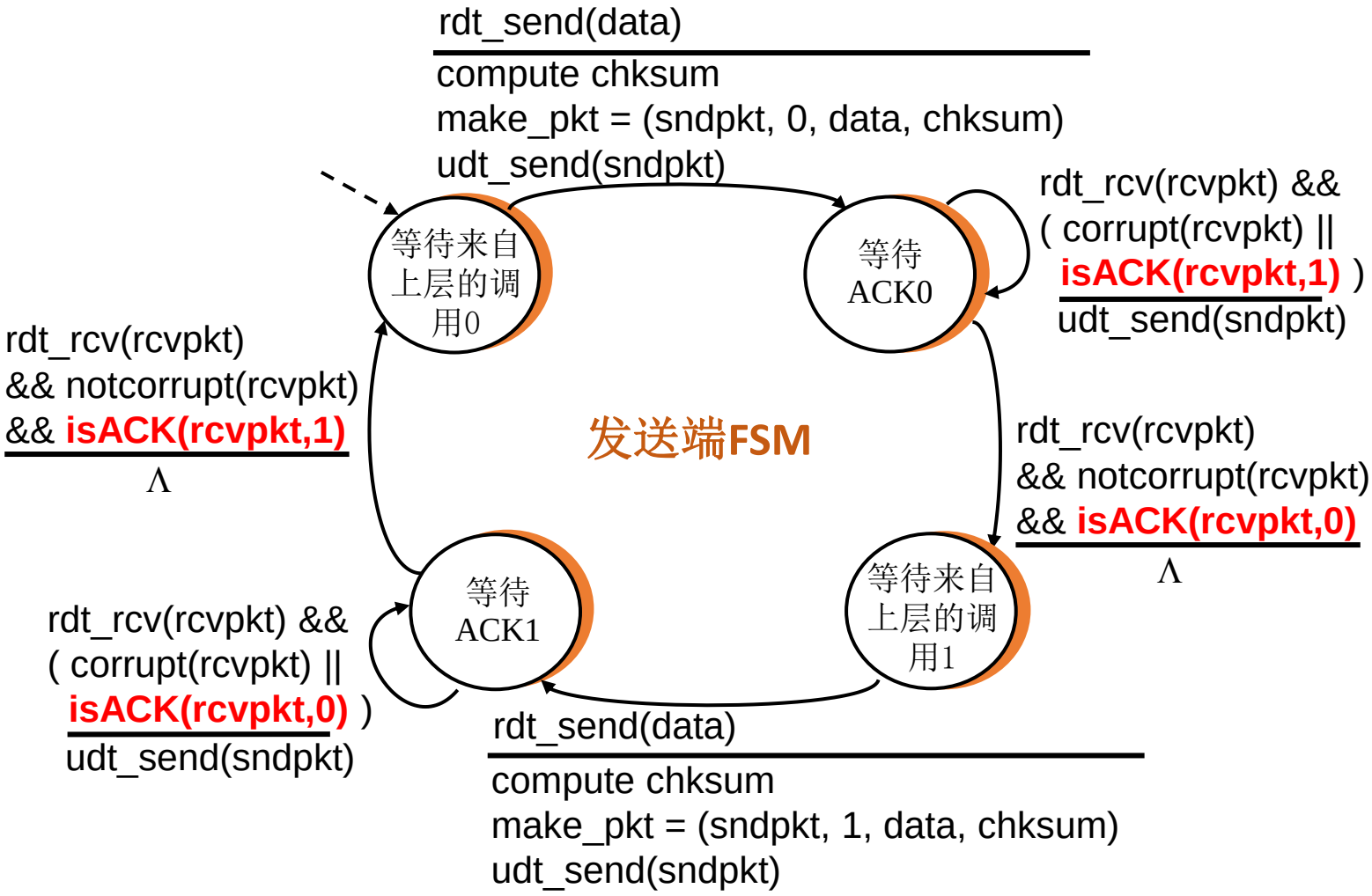


- rdt2.1：接收端状态机

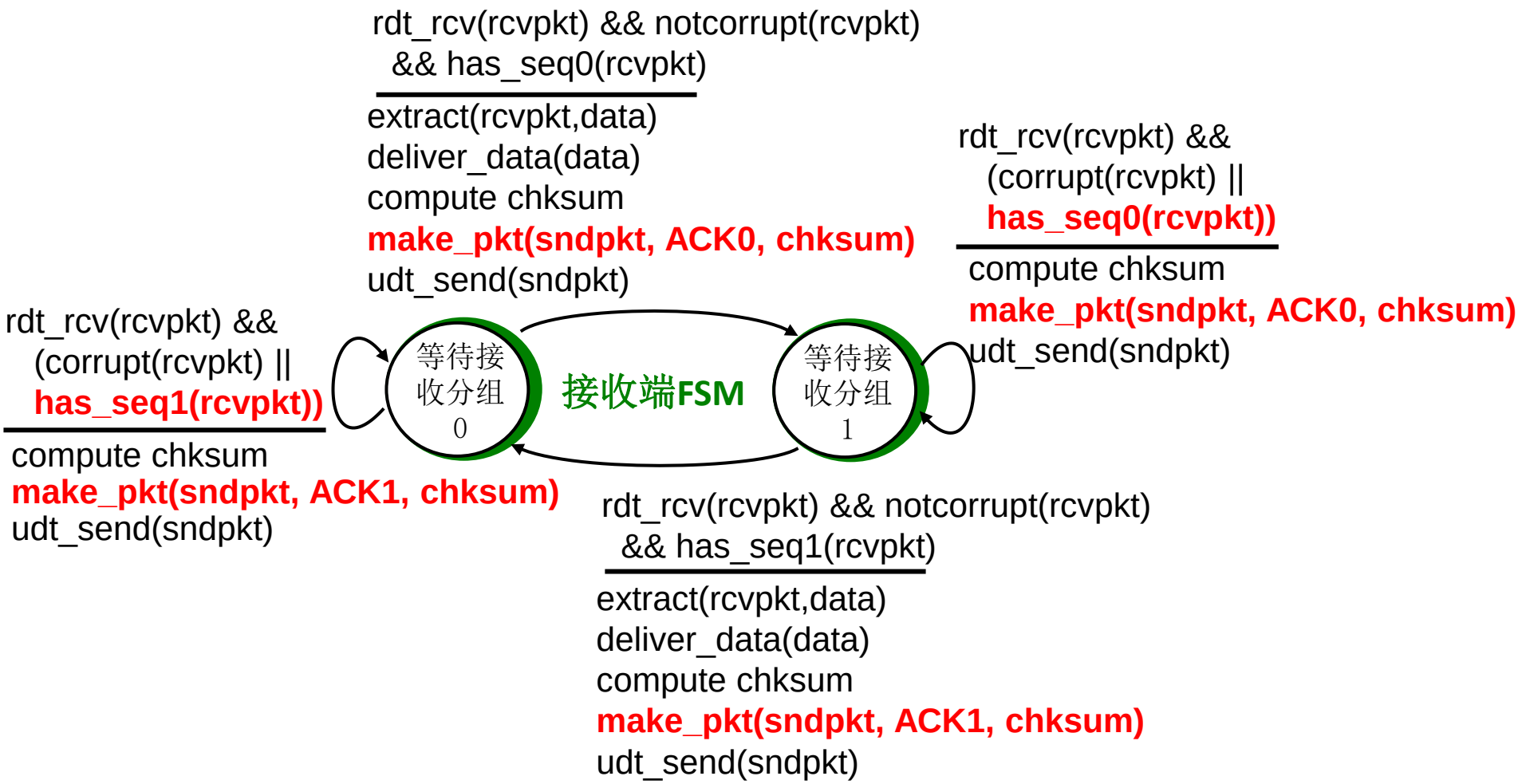


- rdt2.2：对rdt2.1的改进
  - 与rdt2.1功能相同，只使用ACK，不再使用NAK (NAK-free)
  - 接收端通过发送对最后正确收到的分组的ACK代替NAK
    - ACK中必须携带所确认分组的序列号
  - 发送端接收到重复的ACK，代表对当前分组的NAK，则
    - 重传当前的分组

- rdt2.2：发送端状态机



- rdt2.2：接收端状态机

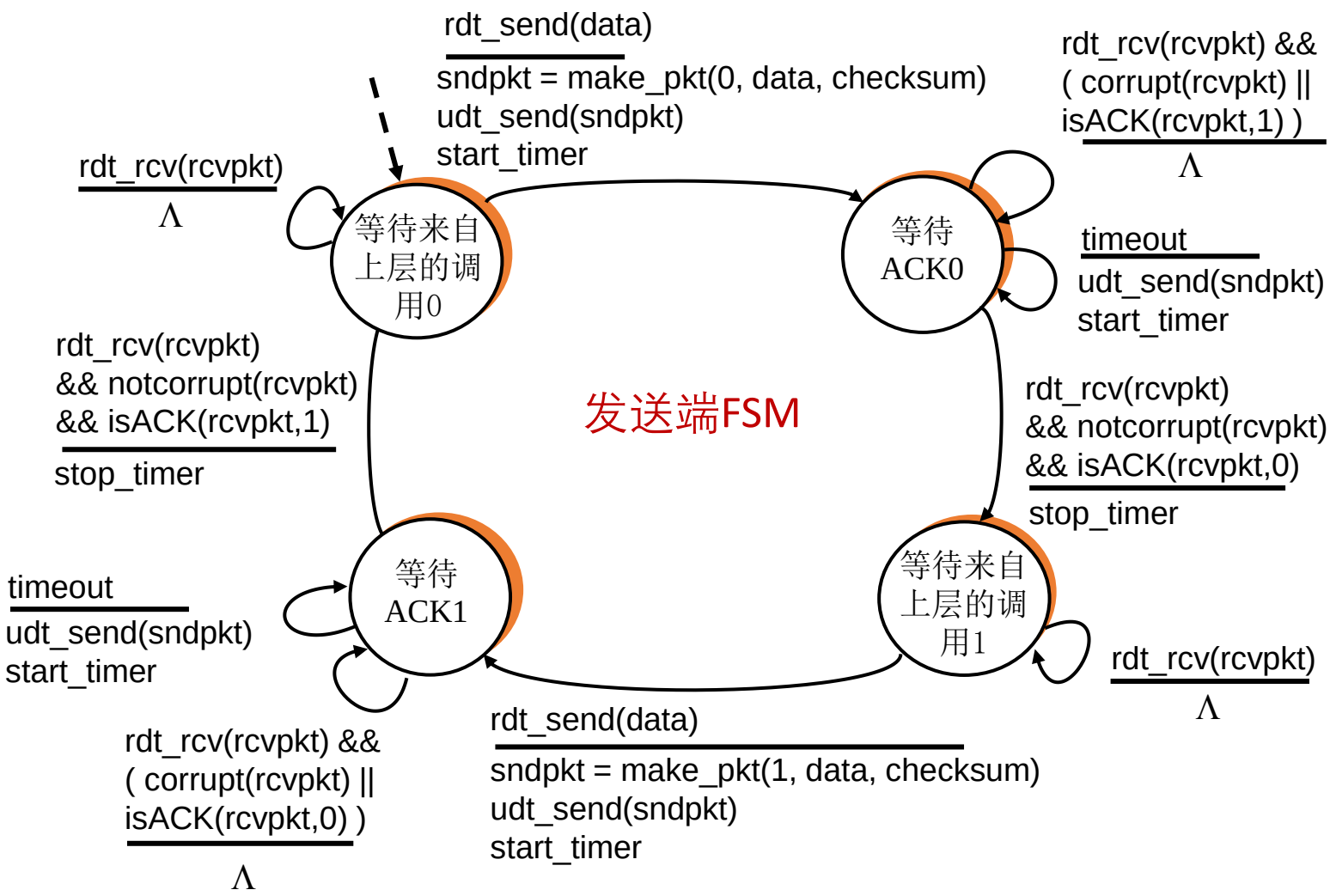


rdt3.0：通道既有差错又有丢失

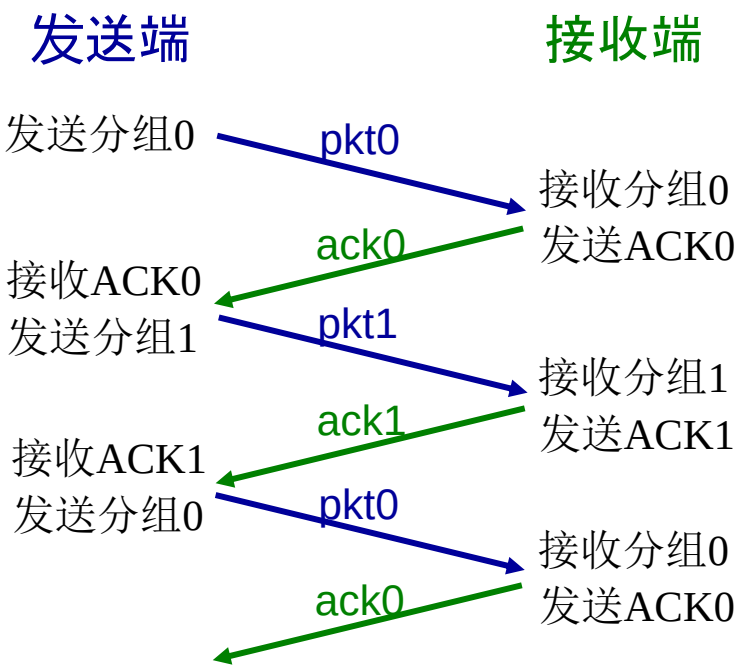
- **新的假设**：下层通道可能会有分组丢失（数据分组或ACK分组）
  - 如何检测丢失，当丢失发生时如何解决？
  - 前面的校验和、序列号、ACK、重传机制等不足以解决丢失检测问题
- **解决方法**：发送端等待一个**合理的时间**（**需要一个定时器**）
  - 如果未收到ACK，则重传当前的分组
  - 如果分组仅仅是被延迟，或是ACK丢失，会造成接收端重复接收
  - 接收端需要根据序列号判断重复的分组，并丢弃

合理时间：多长时间合理？过长和过短有什么问题？

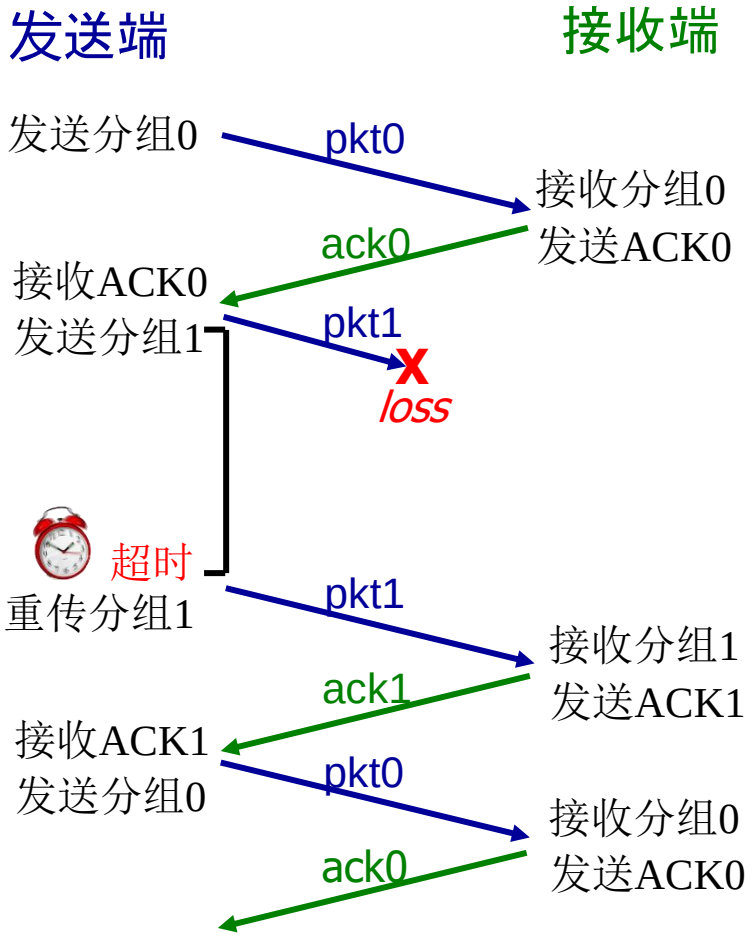
- rdt3.0：发送端状态机



rdt3.0：场景示例



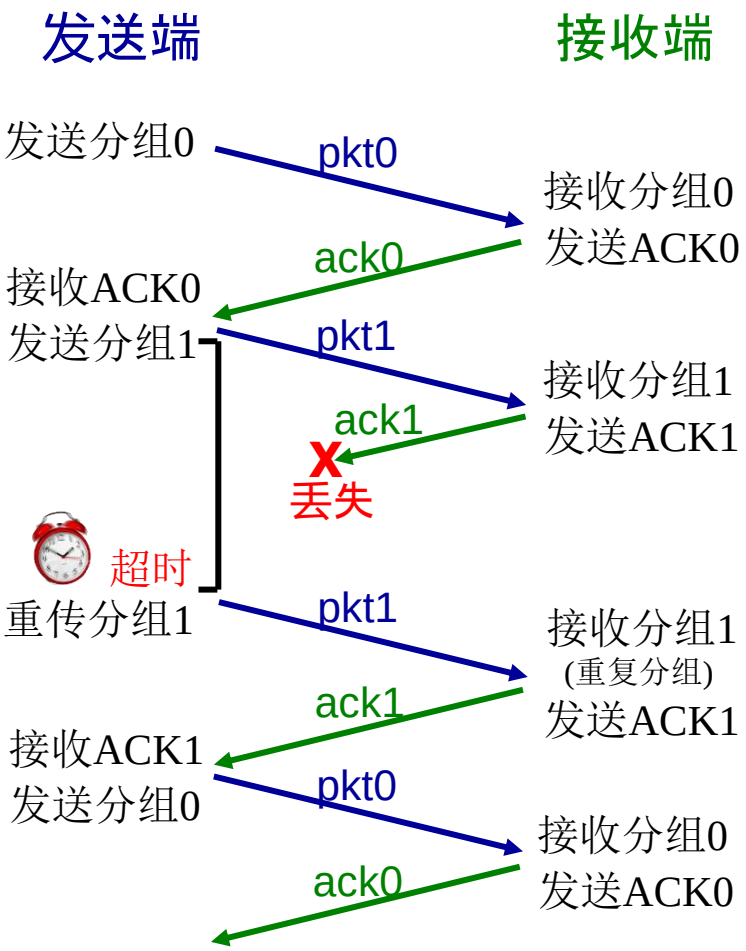
(a) 无丢失



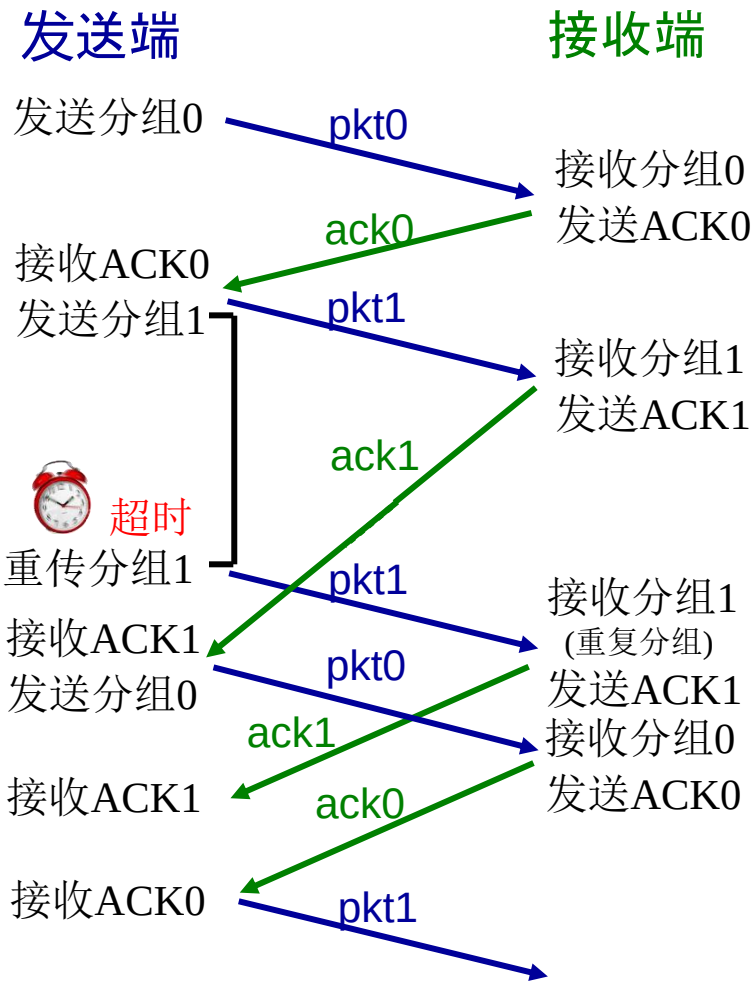
(b) 分组丢失

rdt3.0：场景示例

rdt 3.0可以解决失序问题吗？

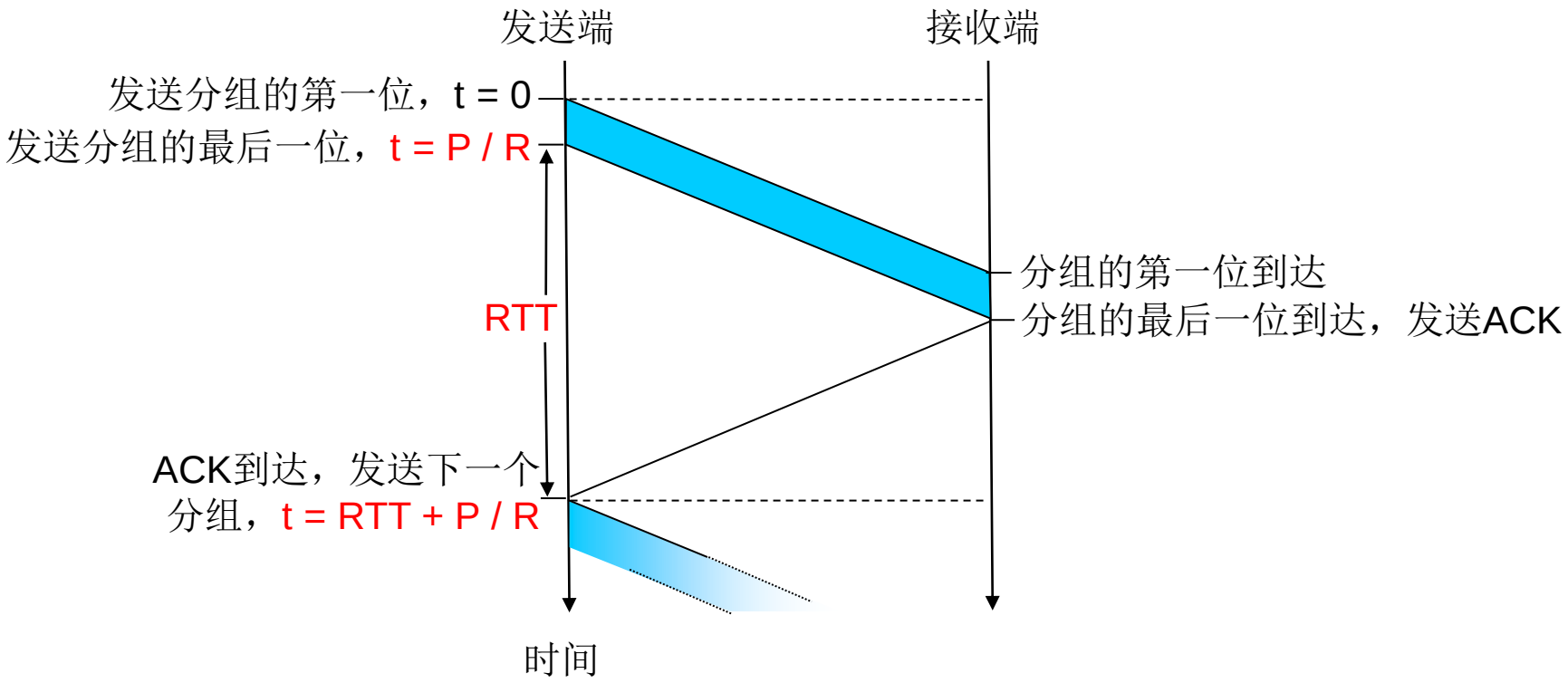


(c) ACK丢失



(d) 提前超时

- rdt3.0：性能问题（停等协议）
  - rdt3.0可以实现可靠机制，但存在性能问题



- rdt3.0：性能问题（停等协议）
  - 例如：R=1Gbps (10^9 bps) 的链路，往返延时RTT=30毫秒，分组长P=1000字节 (8000位)，发送一个分组所用的时间$TRANSP$为：

$$TRANSP = \frac{P}{R} = \frac{8000\ bit}{10^9\ bit/s} = 8\mu s$$

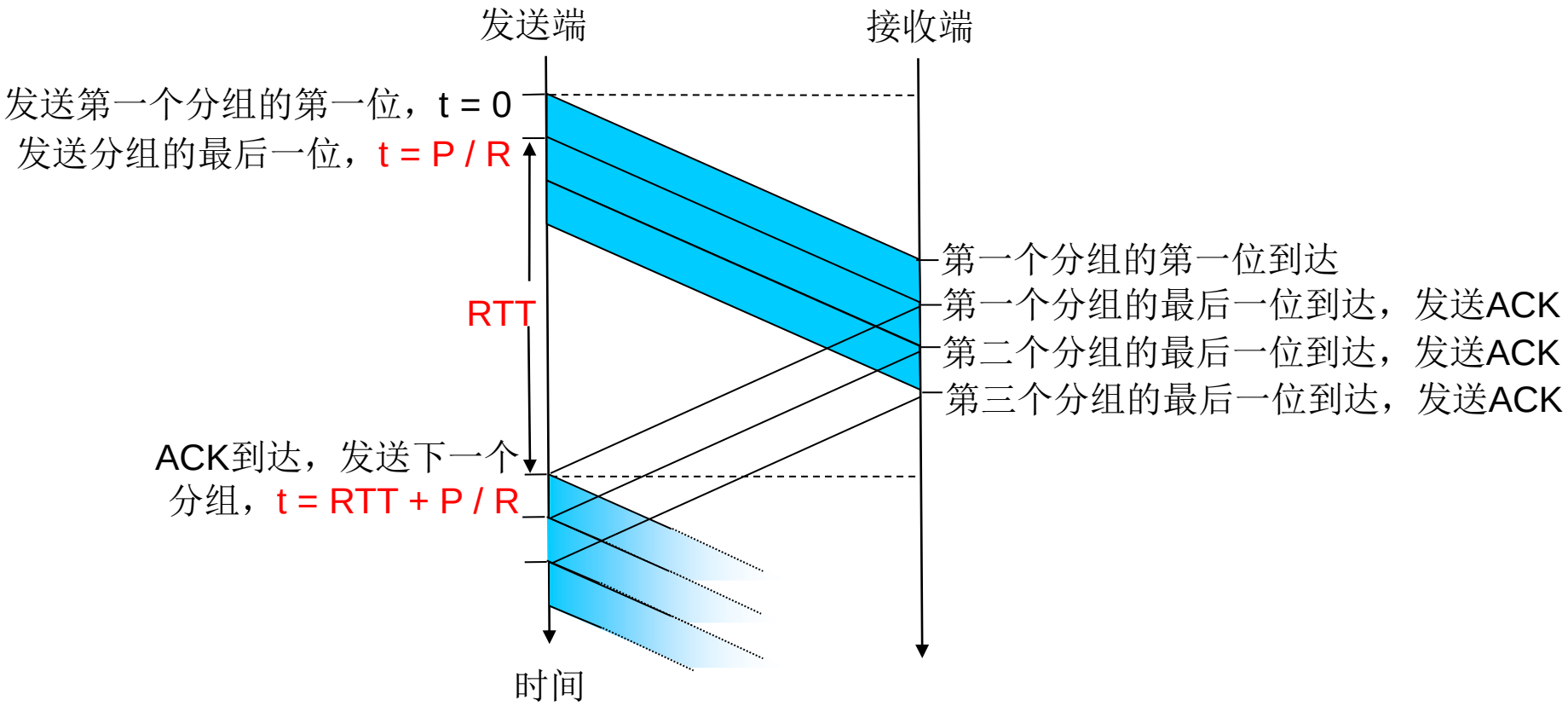
  - 发送端的链路利用率（忽略发送ACK的时间）

$$U_{sender} = \frac{P/R}{RTT + P/R} = \frac{0.008}{30 + 0.008} \approx 0.00027$$

  - 大约每30毫秒发送一个分组，1Gbps链路上的吞吐率约为264Kbps

如何提高链路利用率？Rdt 3.0如何改进？

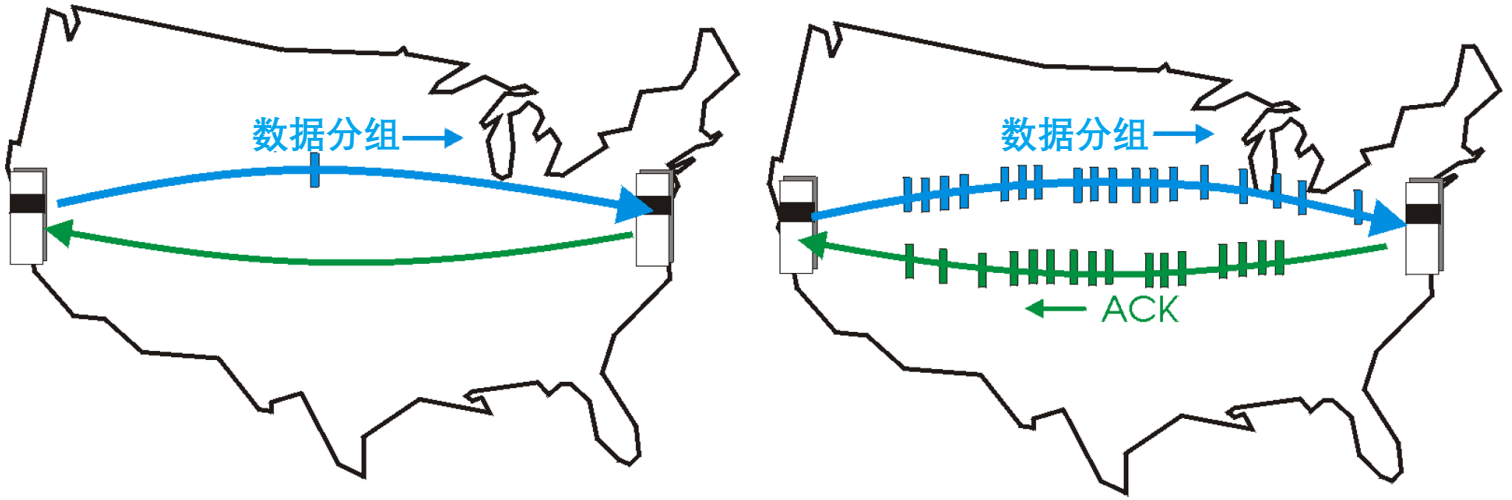
- rdt3.0：性能优化



$$U_{sender} = \frac{3P/R}{RTT + P/R} = \frac{0.008 \times 3}{30 + 0.008} \approx 0.00081$$

流水线协议

- 流水线机制：在确认未返回之前允许发送多个分组
  - 0和1两个序列号还够用吗？
  - 发送端和接收端缓冲区需要增加吗？

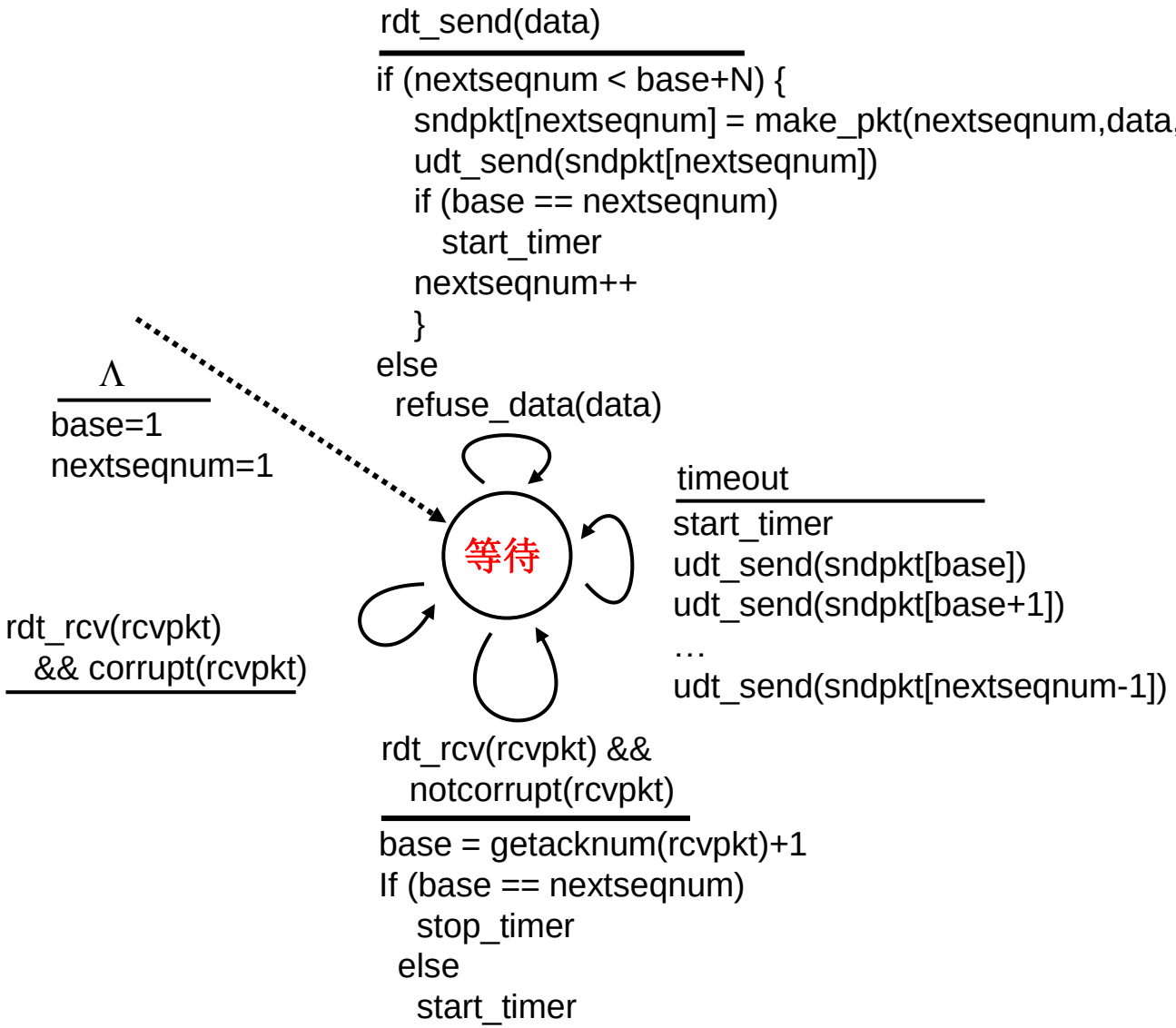


- 两种典型的流水线协议：
  - 回退N：*Go-Back-N (GBN)*
  - 选择重传：*Selective Repeat (SR)*

- 回退N：*Go-Back-N (GBN)*
  - 允许发送端发出N个未得到确认的分组
  - 需要增加序列号范围
    - 分组首部中增加k位的序列号，序列号空间为[0, 2^k-1]
  - 采用累积确认，只确认连续正确接收分组的最大序列号
    - 可能接收到重复的ACK
  - 发送端设置定时器，定时器超时时，重传所有未确认的分组

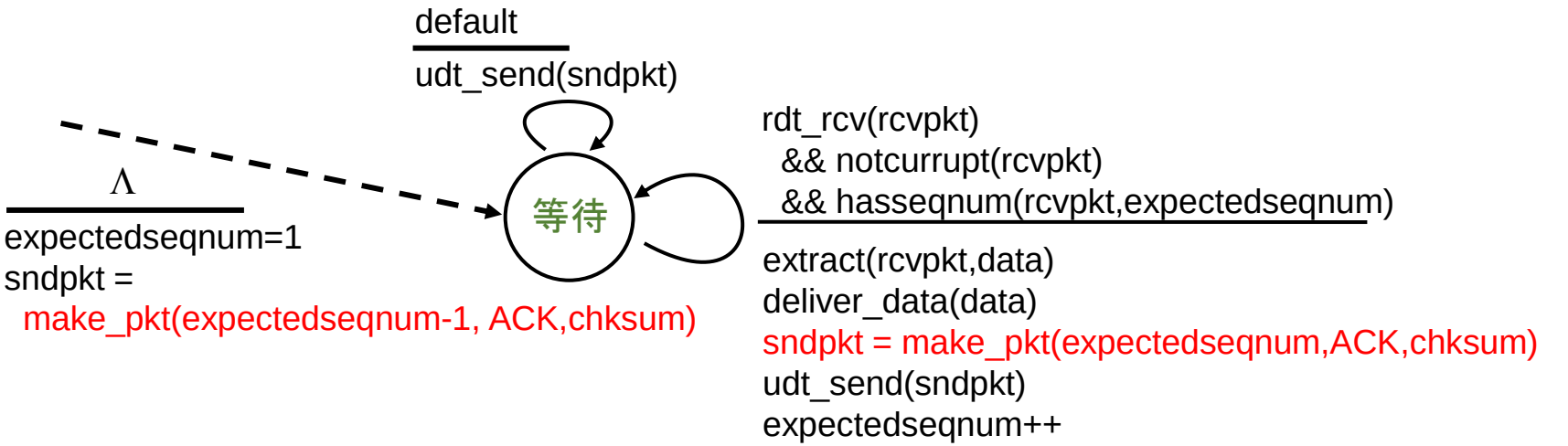
考虑设置窗口N的目的是什么？

- GBN发送端扩展FSM

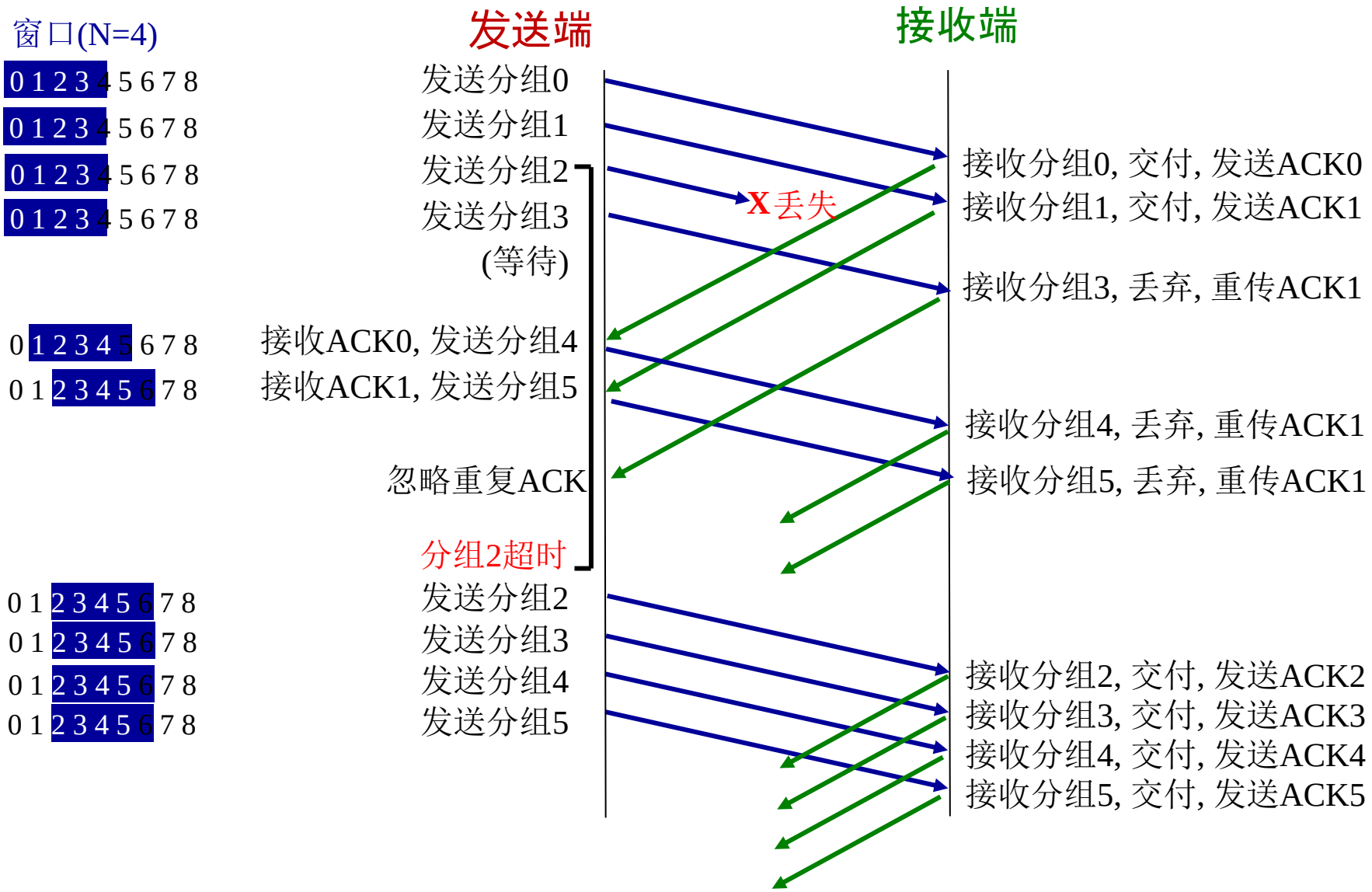


GBN接收端扩展FSM

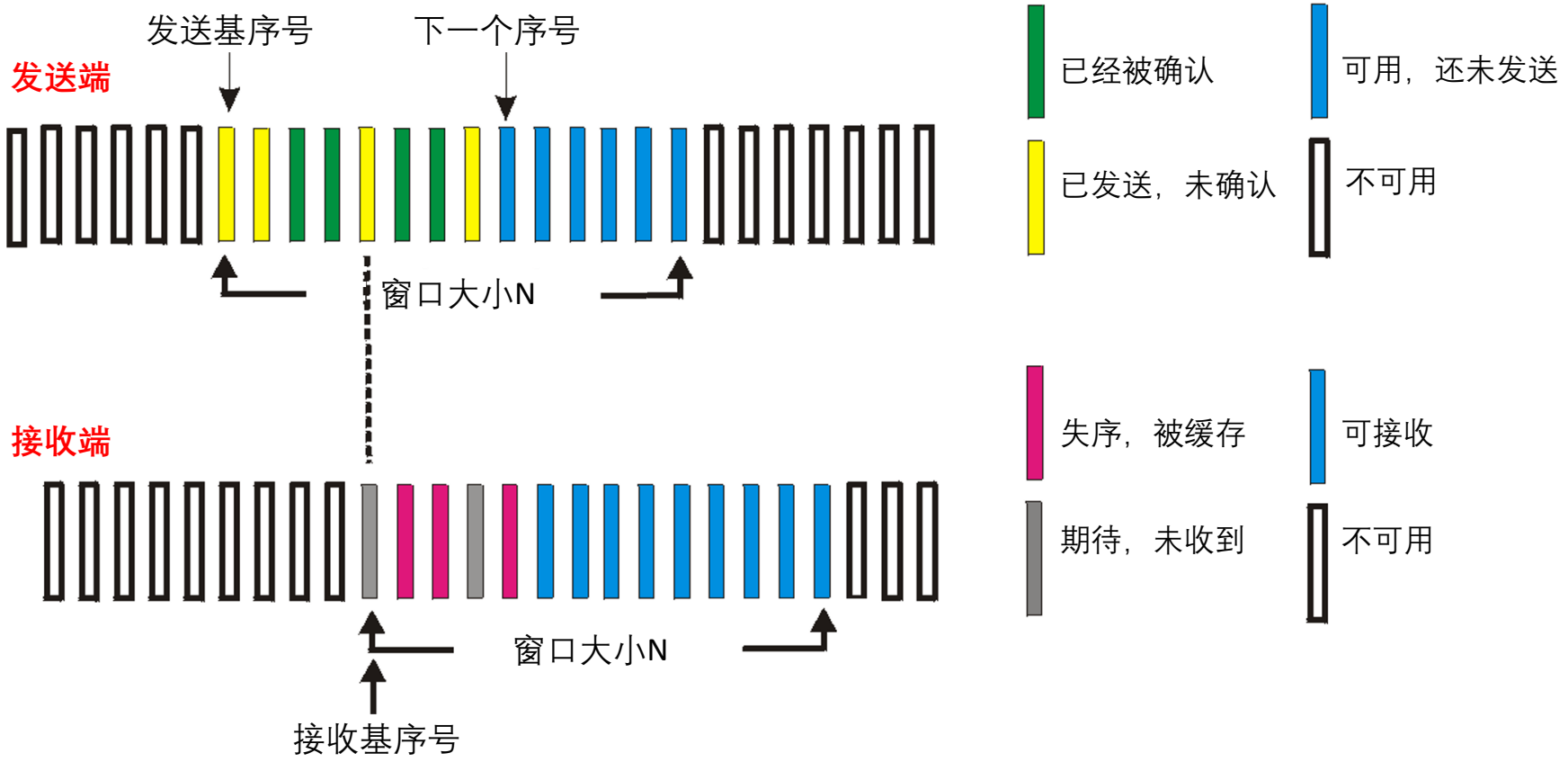
- 只使用ACK，确认按序正确接收的最高序号分组
  - 会产生重复的ACK，需要保存希望接收的分组序号 (expectedseqnum)
- 失序分组（未按序到达）处理
  - 不缓存、丢弃
  - 重发ACK，确认按序正确接收的最高序号分组



- GBN交互示例


窗口(N=4)
发送端
接收端
0 1 2 3 4 5 6 7 8
发送分组0
发送分组1
发送分组2
发送分组3
(等待)
X丢失
接收分组0, 交付, 发送ACK0
接收分组1, 交付, 发送ACK1
接收分组3, 丢弃, 重传ACK1
接收ACK0, 发送分组4
接收ACK1, 发送分组5
接收分组4, 丢弃, 重传ACK1
忽略重复ACK
接收分组5, 丢弃, 重传ACK1
分组2超时
发送分组2
发送分组3
发送分组4
发送分组5
接收分组2, 交付, 发送ACK2
接收分组3, 交付, 发送ACK3
接收分组4, 交付, 发送ACK4
接收分组5, 交付, 发送ACK5

- 选择重传：*Selective Repeat (SR)*
  - 接收端独立确认每个正确接收的分组，必要时缓存分组，对高层按序交付
  - 每个分组独立定时，发送端只重传未收到ACK的分组，



选择重传：*Selective Repeat (SR)*

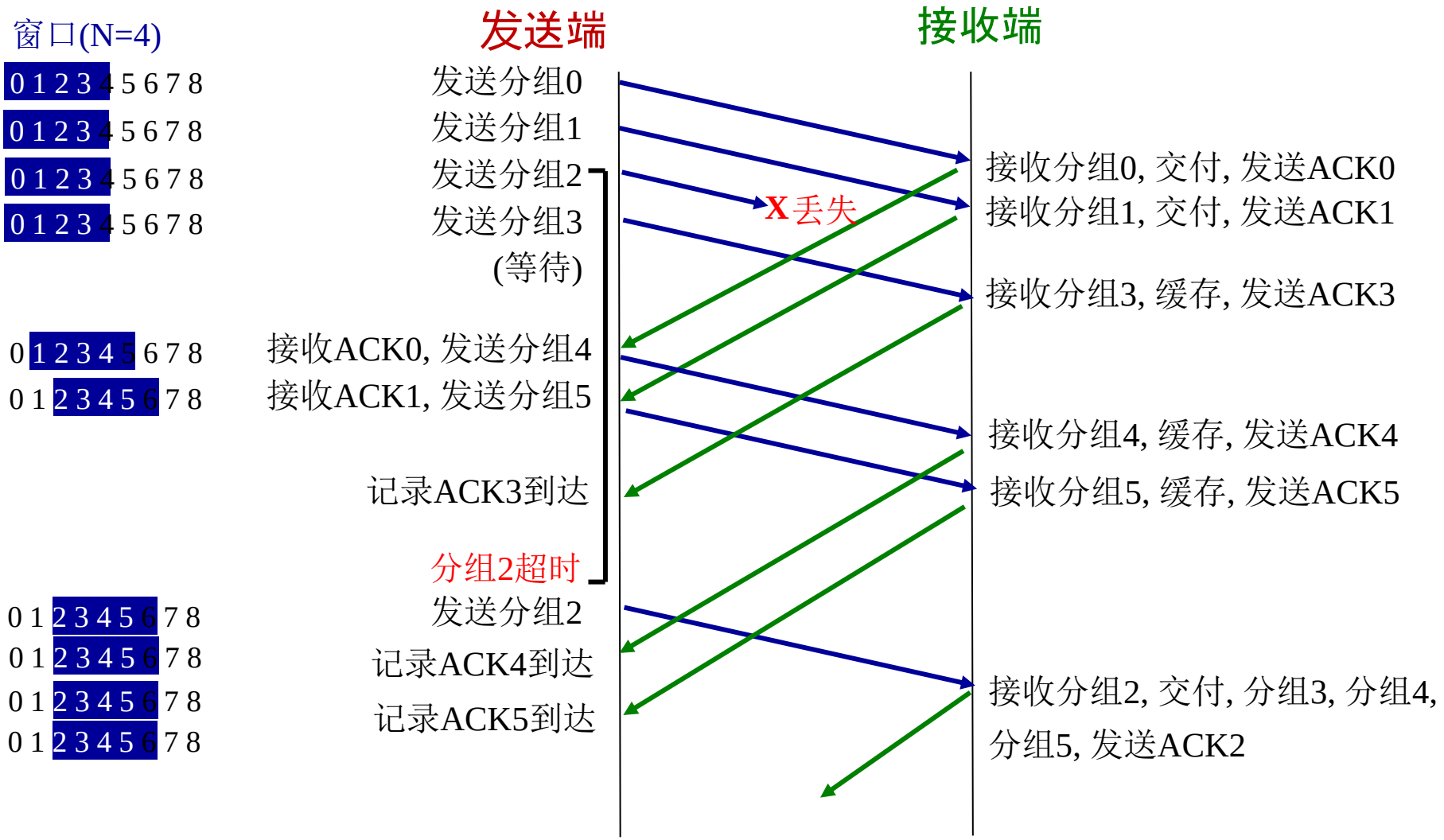
发送端

- **接收上层数据：**如果发送窗口中有可用的序号，则发送分组
- **超时(n)**：重传分组n，重启定时器
- **接收ACK(n)**：n在[send_base, send_base+N-1]区间，将分组n标记为已接收，如果是窗口中最小的未确认的分组，则窗口向前滑动，基序号为下一个未确认分组的序号

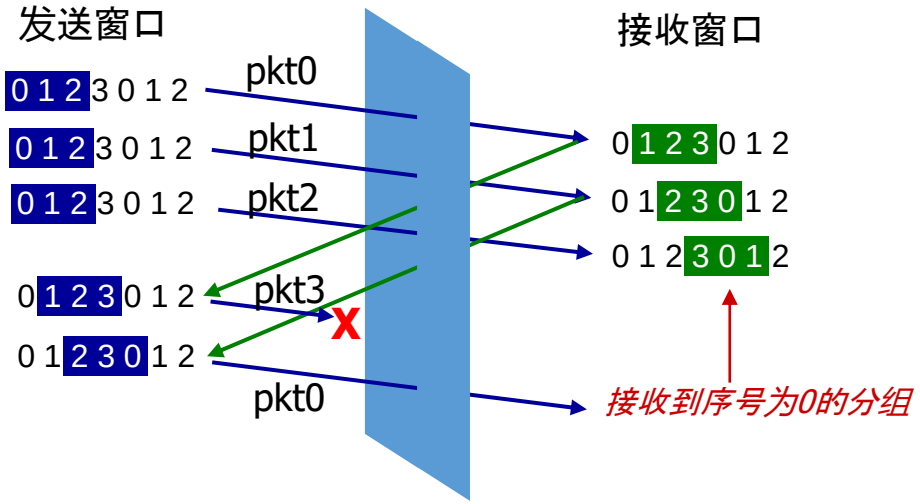
接收端：**接收分组n：**

- n在[rcv_base, rcv_base+N-1]区间，发送ACK(n)，缓存失序分组，按序到达的分组交付给上层，窗口向前滑动
- n在[rcv_base-N, rcv_base-1]区间，发送ACK(n)

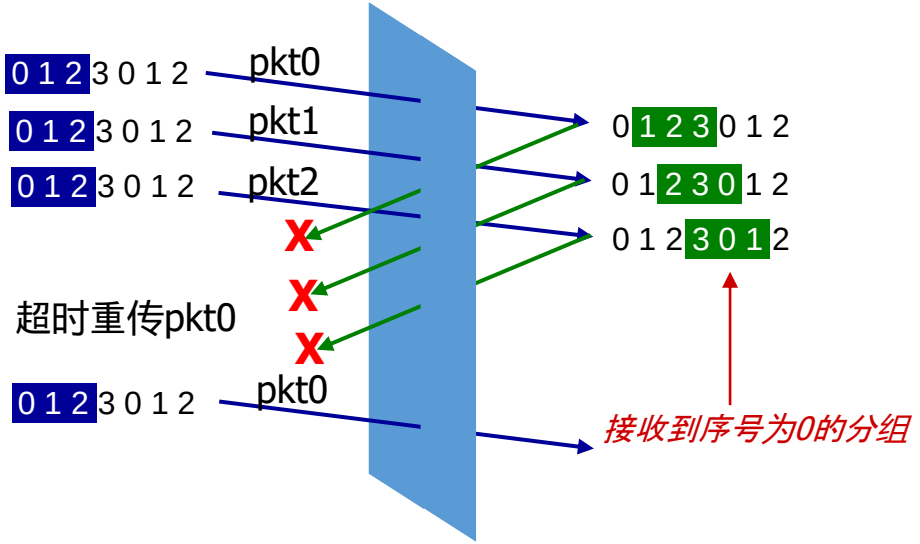
SR交互示例


窗口(N=4)
0 1 2 3 4 5 6 7 8
发送端
接收端
发送分组0
发送分组1
发送分组2
发送分组3
(等待)
X丢失
接收分组0, 交付, 发送ACK0
接收分组1, 交付, 发送ACK1
接收分组3, 缓存, 发送ACK3
接收ACK0, 发送分组4
接收ACK1, 发送分组5
接收分组4, 缓存, 发送ACK4
接收分组5, 缓存, 发送ACK5
记录ACK3到达
分组2超时
发送分组2
记录ACK4到达
记录ACK5到达
接收分组2, 交付, 分组3, 分组4,
分组5, 发送ACK2

- SR问题讨论
  - 例如：序号为0、1、2、3，窗口大小为3
  - 在两种情况下，接收端没有差别



问题1：窗口大小为2，情况会怎样？



单选题 5分

设置

问题：序号空间和窗口大小之间有怎样的关系？

A 序号空间应大于等于窗口大小的2倍

B 序号空间应至少比窗口大小多2

C 序号空间应为窗口大小的平方

提交

3.5 传输控制协议TCP-特点

RFCs: 793,1122,1323, 2018, 2581

- TCP协议特点
  - 提供可靠服务：按序、可靠交付
    - 提供字节流服务，不识别消息边界
  - 可靠传输机制
    - 提供差错检测（校验和）功能，正确接收返回确认
    - 使用序列号检测丢失和乱序
    - 超时重传机制，解决出错、丢失问题
    - 支持流水线机制，自适应窗口

3.5 传输控制协议TCP-特点

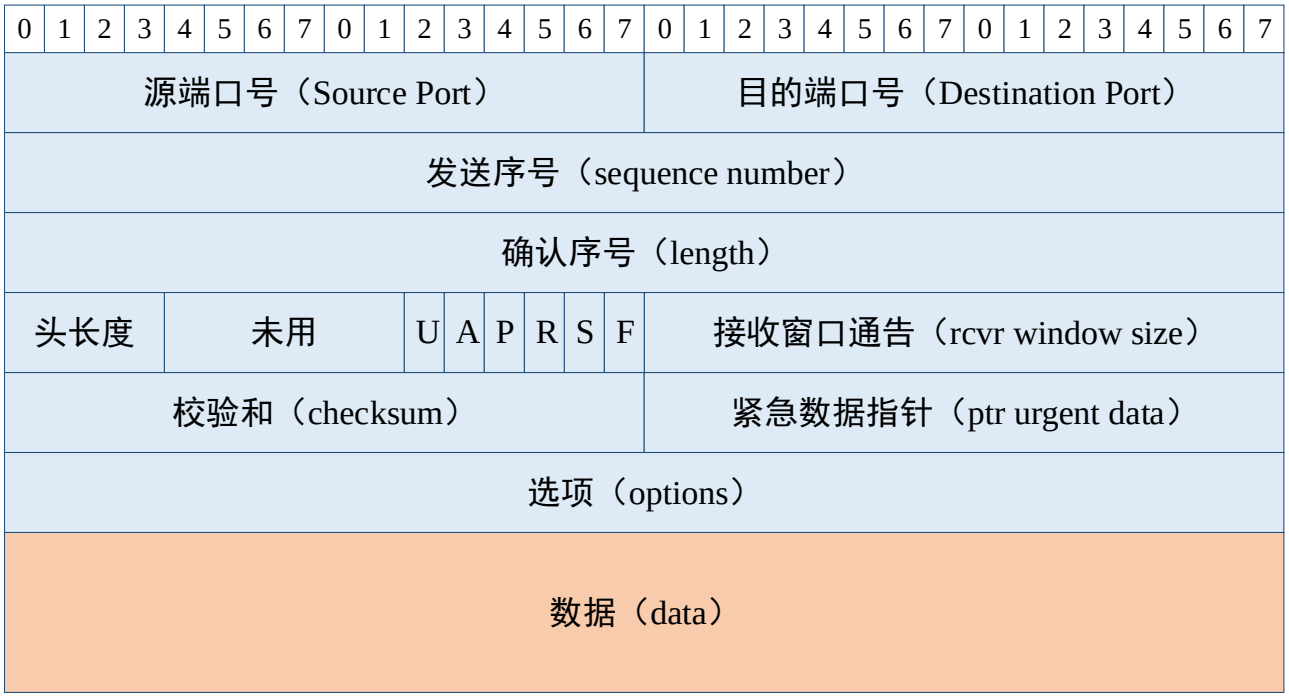
RFCs: 793,1122,1323, 2018, 2581

TCP协议特点（续）

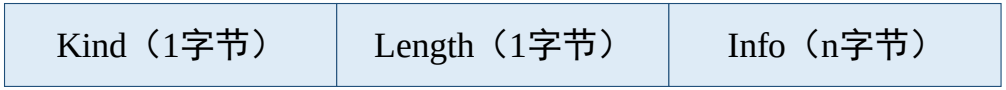
- 面向连接：发送数据之前发送方和接收方之间需要握手
  - 三次握手建立连接
  - 初始化所需的参数及分配缓冲区
- 提供复用分用功能
- 只提供点对点通信
- 具有流量控制和拥塞控制功能

3.5 传输控制协议TCP-段格式

TCP数据单元（段）格式

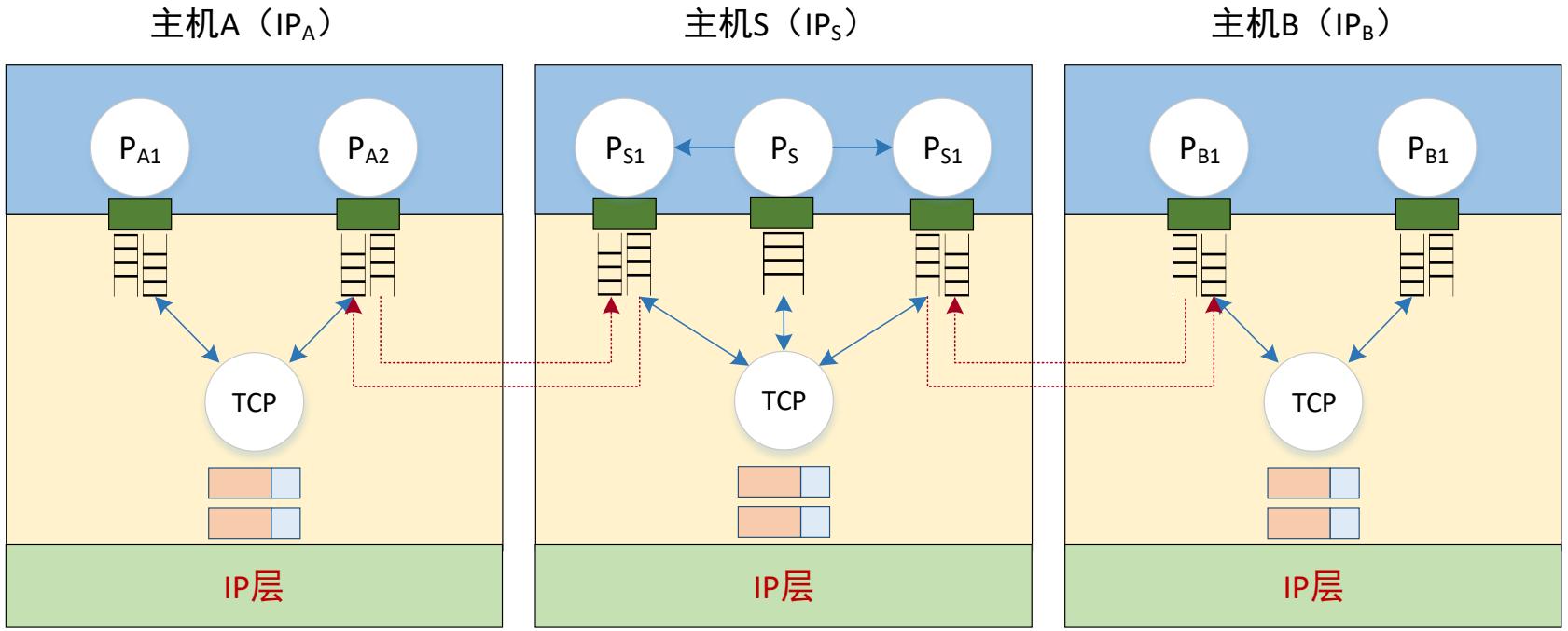


- 头长度：四个字节为计数单位，包含选项部分
- 接收窗口通告：指示接收缓冲区可接收的字节数
- 标志位：URG, ACK, PUSH, RESET, SYN, FIN
- 选项格式：



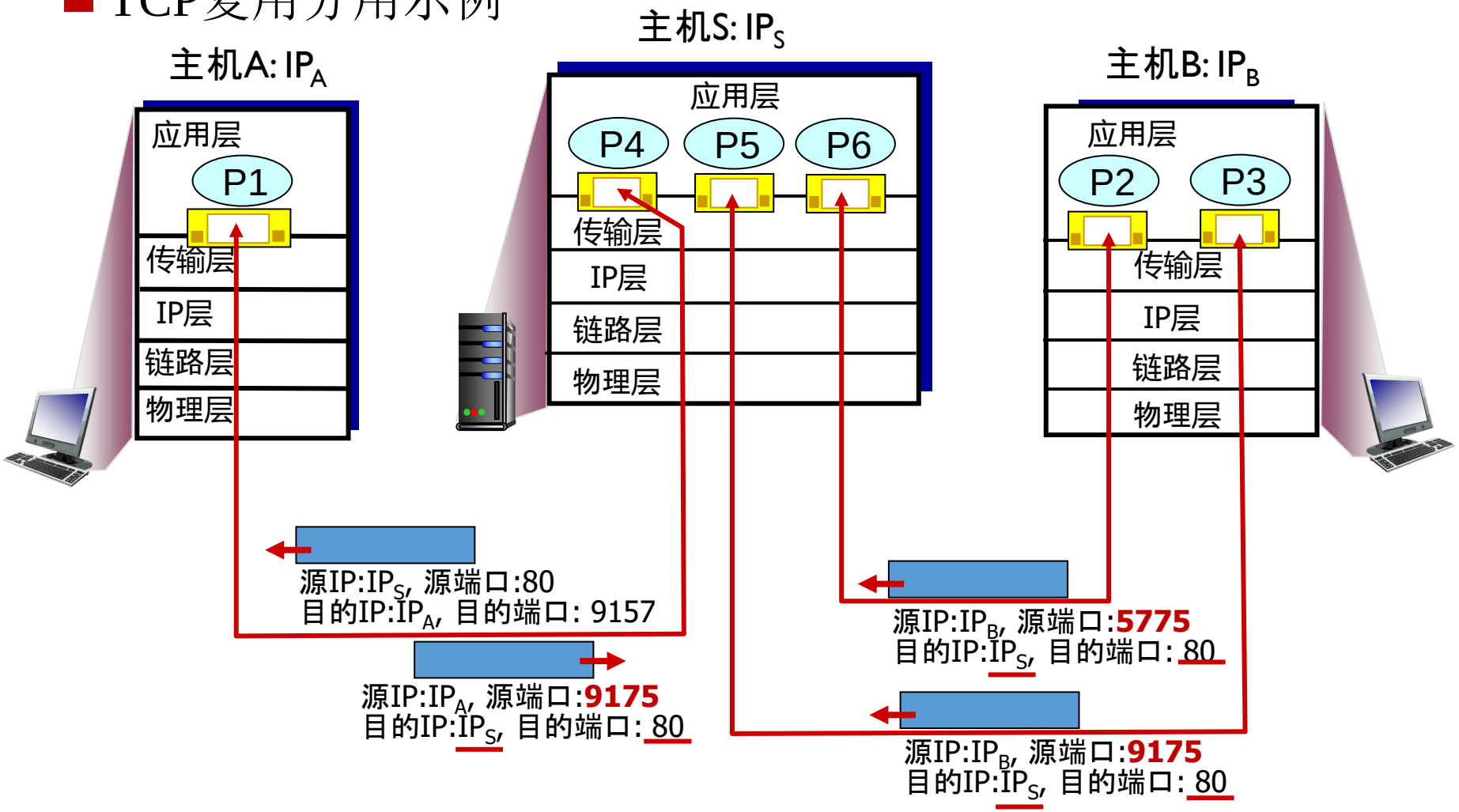
3.5 传输控制协议TCP-复用分用

TCP连接与复用、分用机制



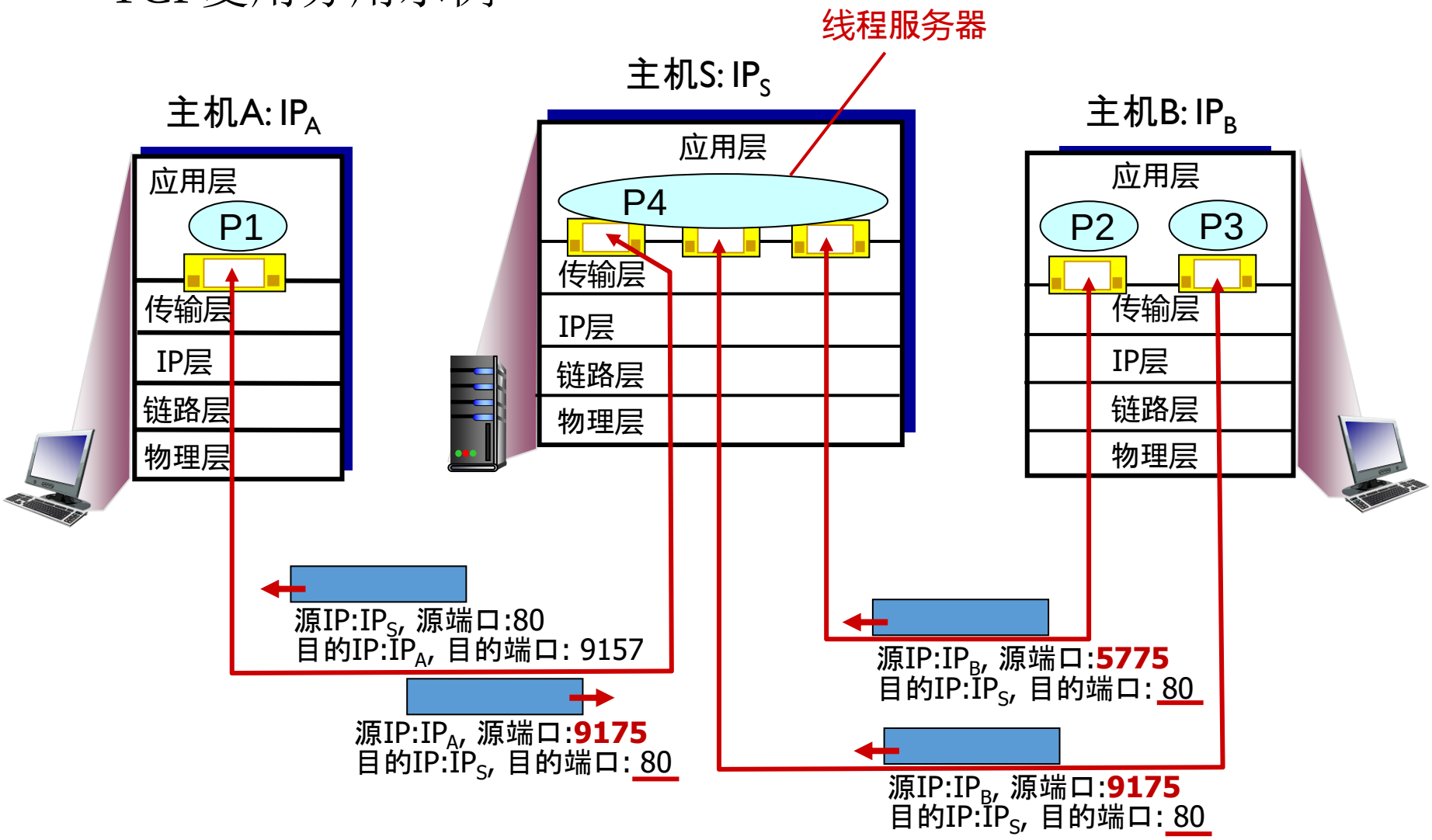
- **通信之前通过三次握手建立TCP连接**
  - 分配缓冲区、协商参数（初始序号、接收缓冲区大小、最大段尺寸等）
- 连接标识（**四元组**）：源IP地址、目的IP地址、源端口号、目的端口号
- 通过建立的TCP连接为应用进程提供**可靠的字节流服务**

- TCP复用分用示例

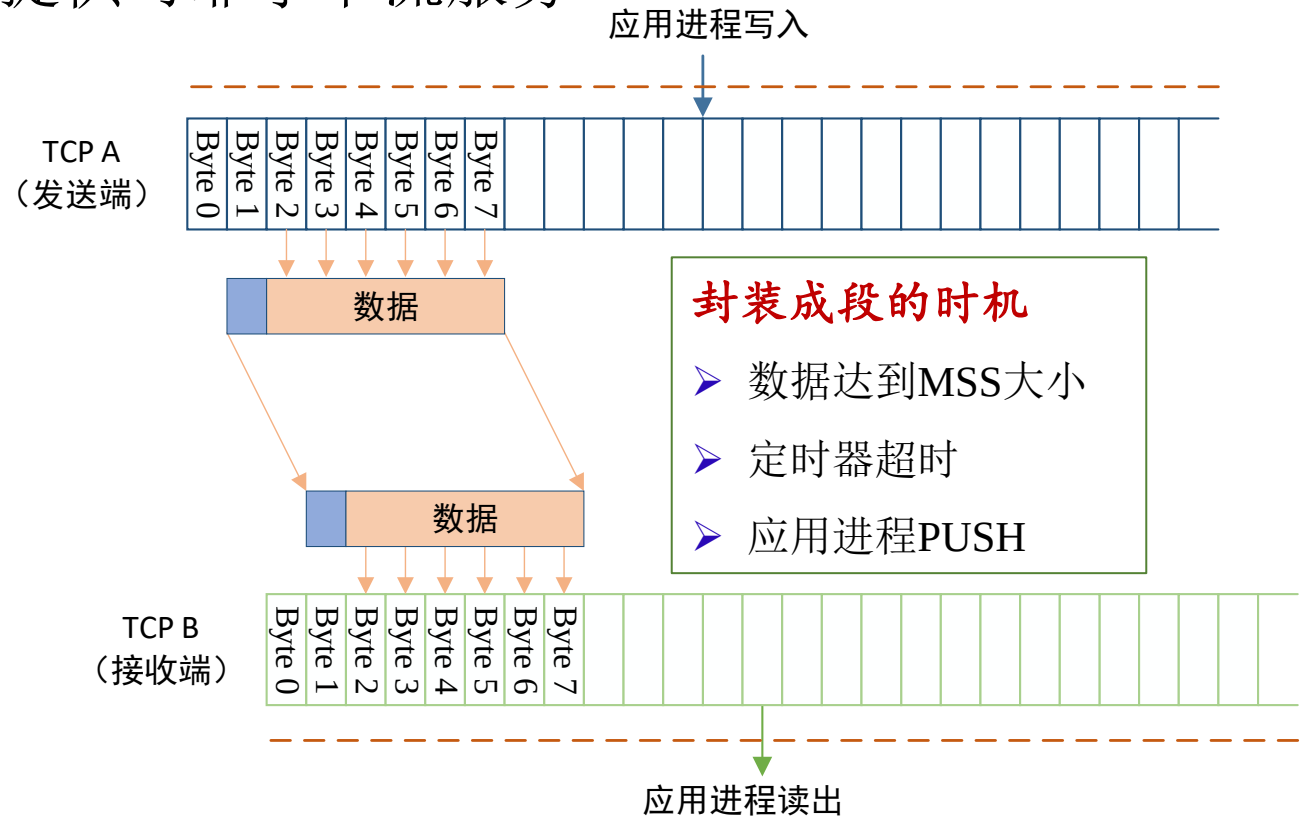


三个段的目的IP地址均为 IP_S，目的端口均为80，但是由于源IP地址或源端口不同，指向不同的套接口

- TCP复用分用示例


线程服务器
主机S: IP_S
主机A: IP_A
主机B: IP_B
应用层
P1
P2
P3
P4
传输层
IP层
链路层
物理层
源IP:IP_S, 源端口:80
目的IP:IP_A, 目的端口: 9157
源IP:IP_A, 源端口:9175
目的IP:IP_S, 目的端口: 80
源IP:IP_B, 源端口:5775
目的IP:IP_S, 目的端口: 80
源IP:IP_B, 源端口:9175
目的IP:IP_S, 目的端口: 80

TCP提供可靠字节流服务



- 最大段长度（MSS，Maximum Segment Size）
  - TCP连接建立时可以通告（选项部分），缺省为535字节
  - MSS尽可能大，但尽可能不使IP层分片

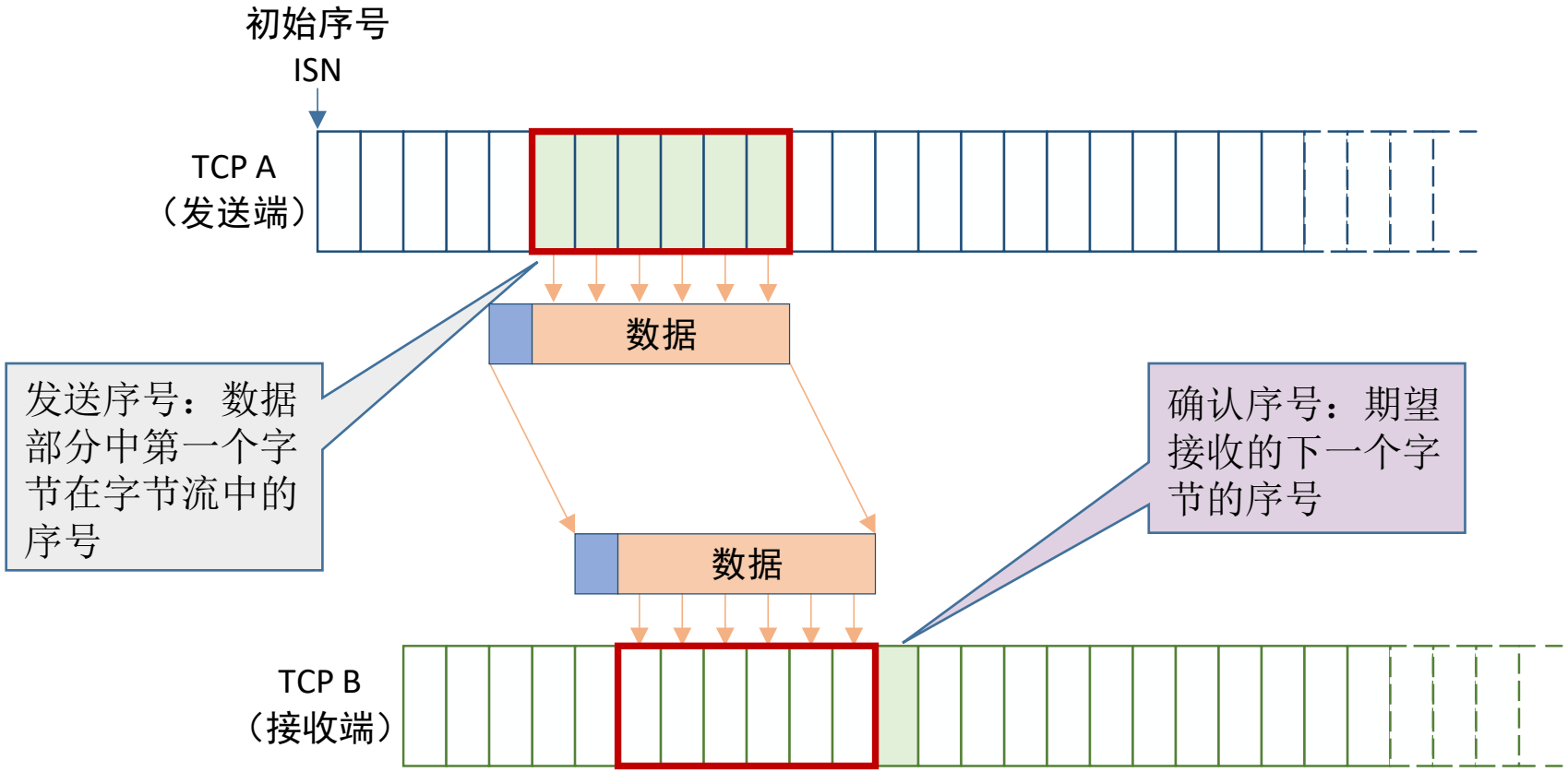
Kind=2	Length=4	最大报文段长度（2字节）

TCP可靠数据传输

- 基本机制
  - 发送端：发送数据、等待确认、超时重传
  - 接收端：进行差错检测，采用累积确认机制（确认按序正确接收到字节的下一个字节序列号）
- 支持流水线机制
  - 发送序号：32位
  - 确认序号：32位
  - 每个字节都有序列号，对段的边界没有要求
- 乱序段处理：协议没有明确规定
  - 接收端不缓存，可以正常工作，处理简单，效率低
  - 接收端缓存，效率高，处理复杂

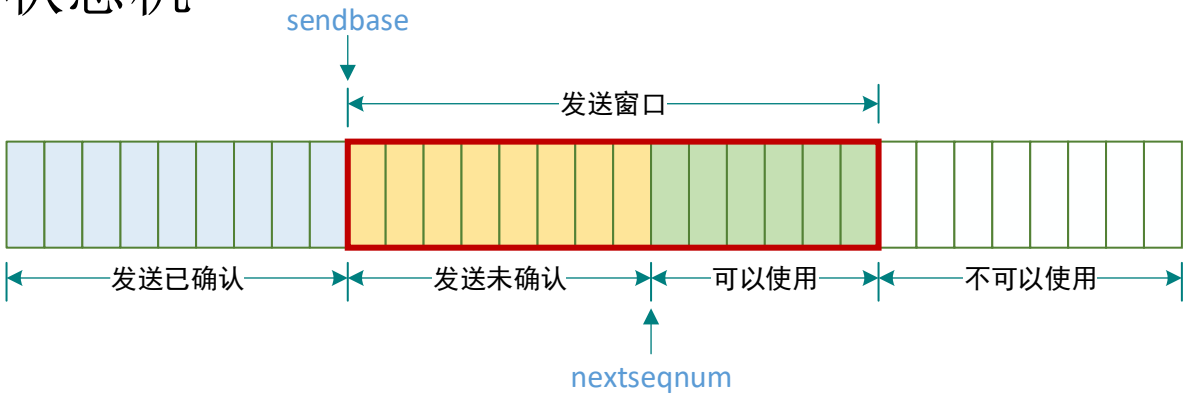
TCP SACK扩展：缓存乱序段，结合选择确认机制（SACK）

- TCP的发送序列号和确认序列号



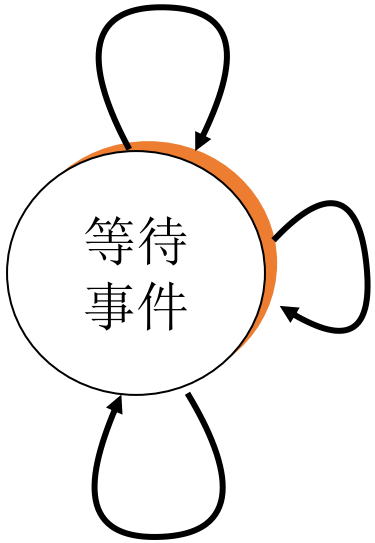
ISN选取: 每个TCP实体维护一个32位计数器，该计数器每4微秒增1，建立连接时从中读取计数器当前值（依赖具体实现）

- TCP可靠数据传输状态机



event: data received
from application above

create, send segment



event: timer timeout for
segment with seq # y

retransmit segment

event: ACK received,
with ACK # y

ACK processing

简化的发送端

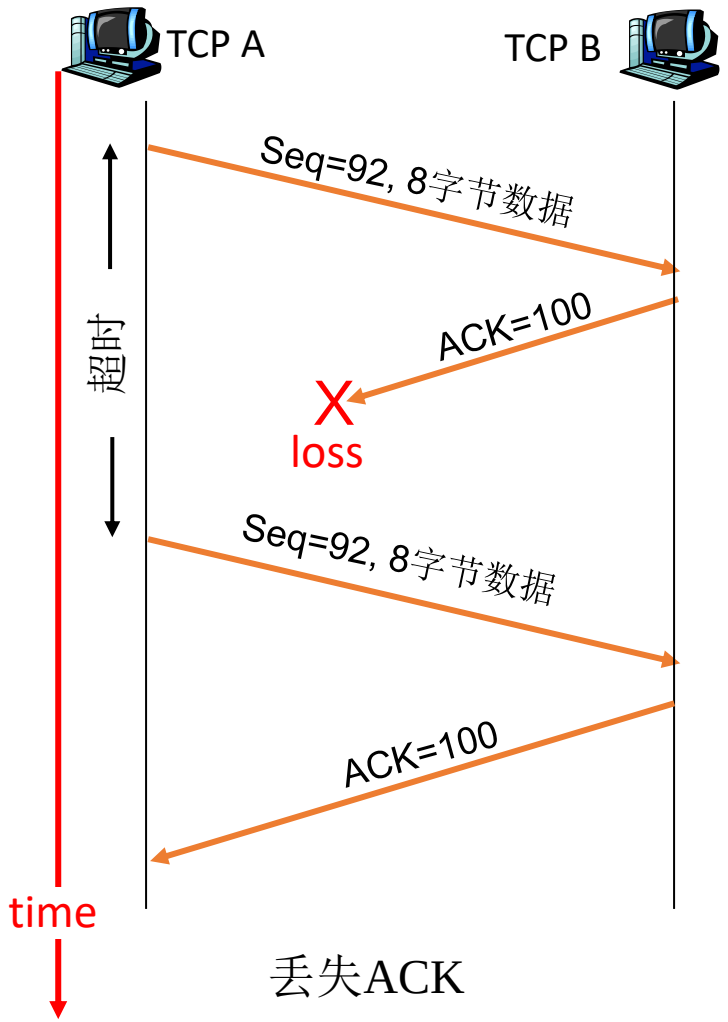
```
00  sendbase = initial_sequence number
01  nextseqnum = initial_sequence number
02
03  loop (forever) {
04     switch(event)
05     event: data received from application above
06          create TCP segment with sequence number nextseqnum
07          compute timeout interval for segment nextseqnum
08          start timer for segment nextseqnum
09          pass segment to IP
10          nextseqnum = nextseqnum + length(data)
11     event: timer timeout for segment with sequence number y
12          retransmit segment with sequence number y
13          compute new timeout interval for segment y
14          restart timer for sequence number y
15     event: ACK received, with ACK field value of y
16          if (y > sendbase) { /* cumulative ACK of all data up to y */
17              cancel all timers for segments with sequence numbers < y
18              sendbase = y
19              }
20          else { /* a duplicate ACK for already ACKed segment */
21              increment number of duplicate ACKs received for y
22              if (number of duplicate ACKS received for y == 3) {
23                   /* TCP fast retransmit */
24                  resend segment with sequence number y
25                  restart timer for segment y
26              }
27     }  /* end of loop forever */
```

- TCP接收端ACK产生机制（RFC 1122, RFC 2581, RFC 5681）

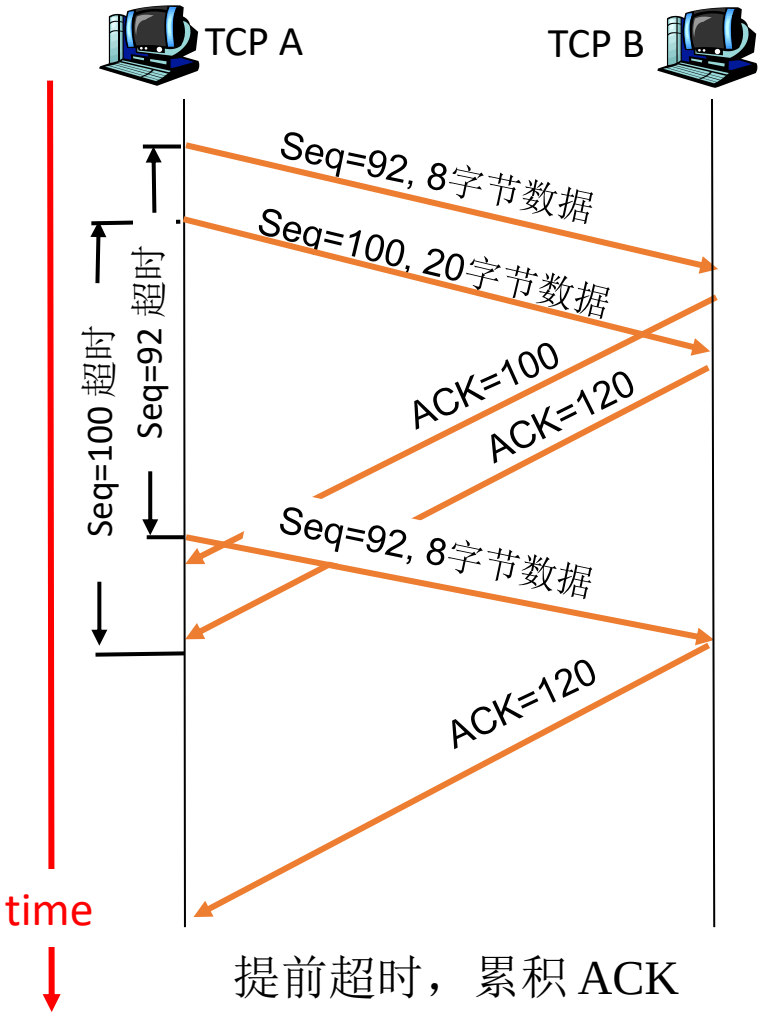
事件	TCP接收端动作
期望序号的报文段按序到达，之前的报文段均已被确认	延迟发送ACK，等待下一个报文段到达；等500毫秒，仍未收到下一个报文段，则发送ACK
期望序号的报文段按序到达，之前有一个延迟确认的报文段	发送ACK，确认两个按序到达的报文段
报文段未按序到达，到达的报文段序号高于期望的序号，数据流不连续	发送重复ACK，确认序号中包含期望接收的序号
到达的报文段填补了之前数据流不连续部分	发送ACK，确认序号中包含期望接收的序号

思考：TCP基本的累积确认机制中，等待500毫秒确认的原因

- TCP重传场景

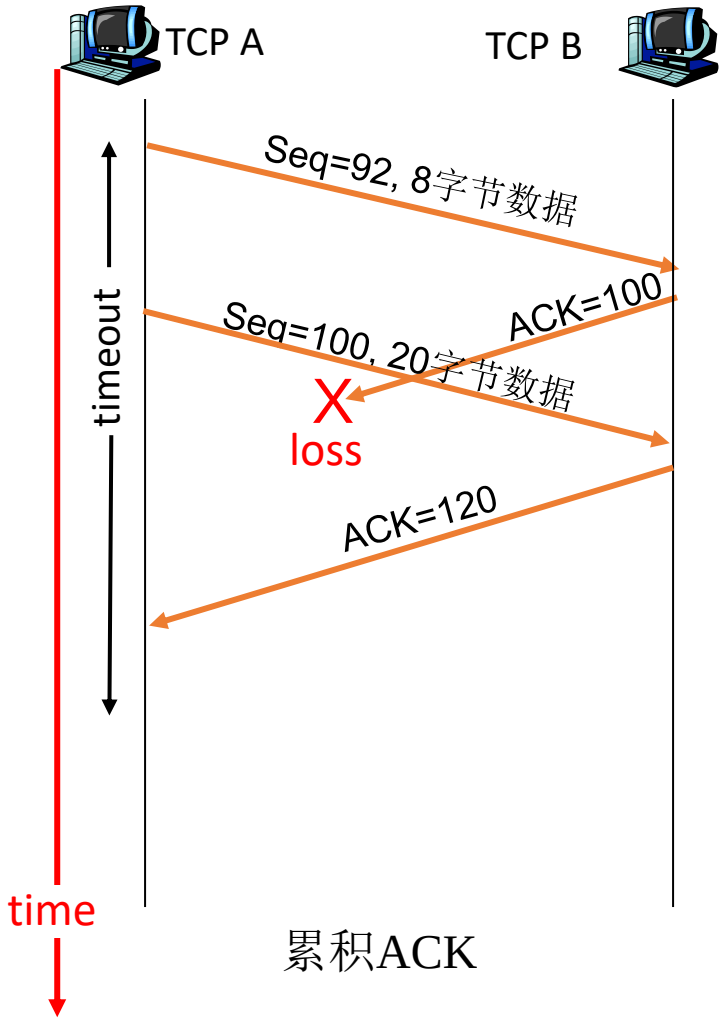


丢失ACK



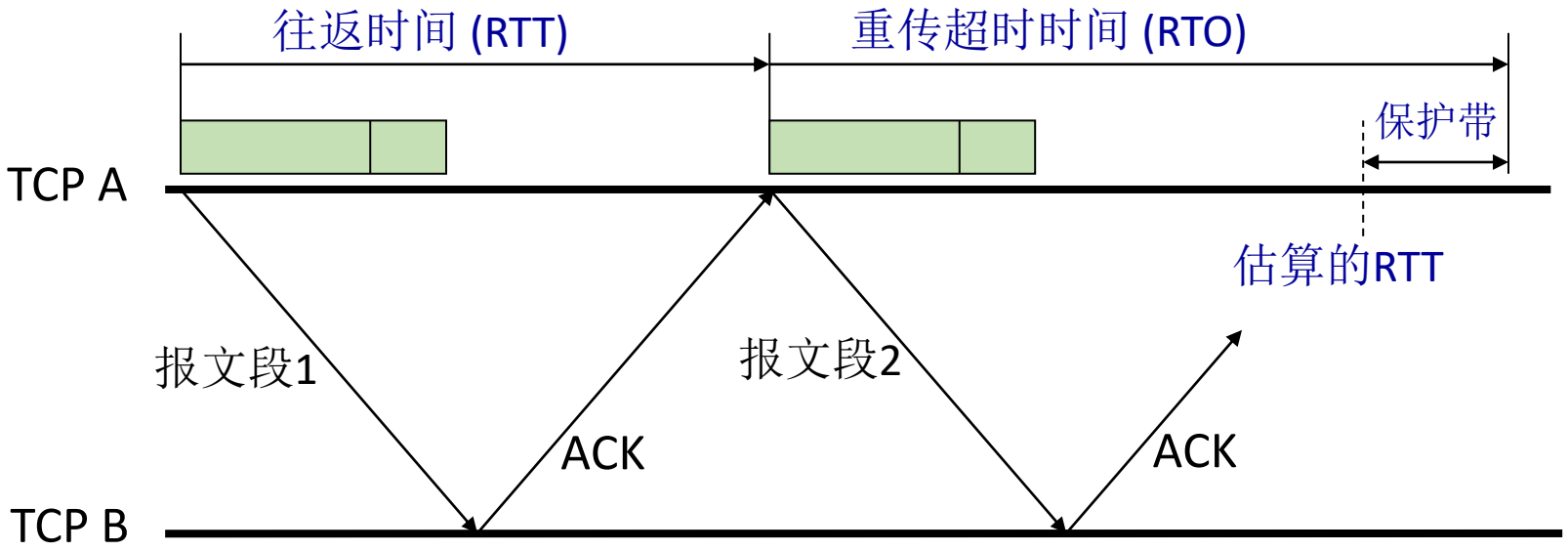
提前超时，累积 ACK

- TCP重传场景



- TCP重传策略结合了GBN和SR的某些特点
- TCP标准中没有明确说明接收端对失序报文段如何处理
- TCP SACK中使用了SR机制
- 问题：**超时时间 (timeout) 如何确定？**

- TCP重传超时时间考虑



- TCP采用**自适应方法**计算重传超时时间（原因？）
  - ✓ 基于往返时间（RTT）确定重传超时间（RTO）
- **问题**：如何准确估算
  - ✓ 上一次RTT可以测得，下一次RTT需要估算
  - ✓ 网络拥塞和路由变化，每次往返时间可能不同，有时会有较大变化

- TCP重传超时时间计算

RFC: 2988

理解适当RTO设置的重要性:

- RTO设置过大，对于丢失的报文段重传等待的时间过长，对于应用来说会引入较大的时延
- RTO设置过小，可能会提前超时，引入不必要的重传，浪费带宽资源

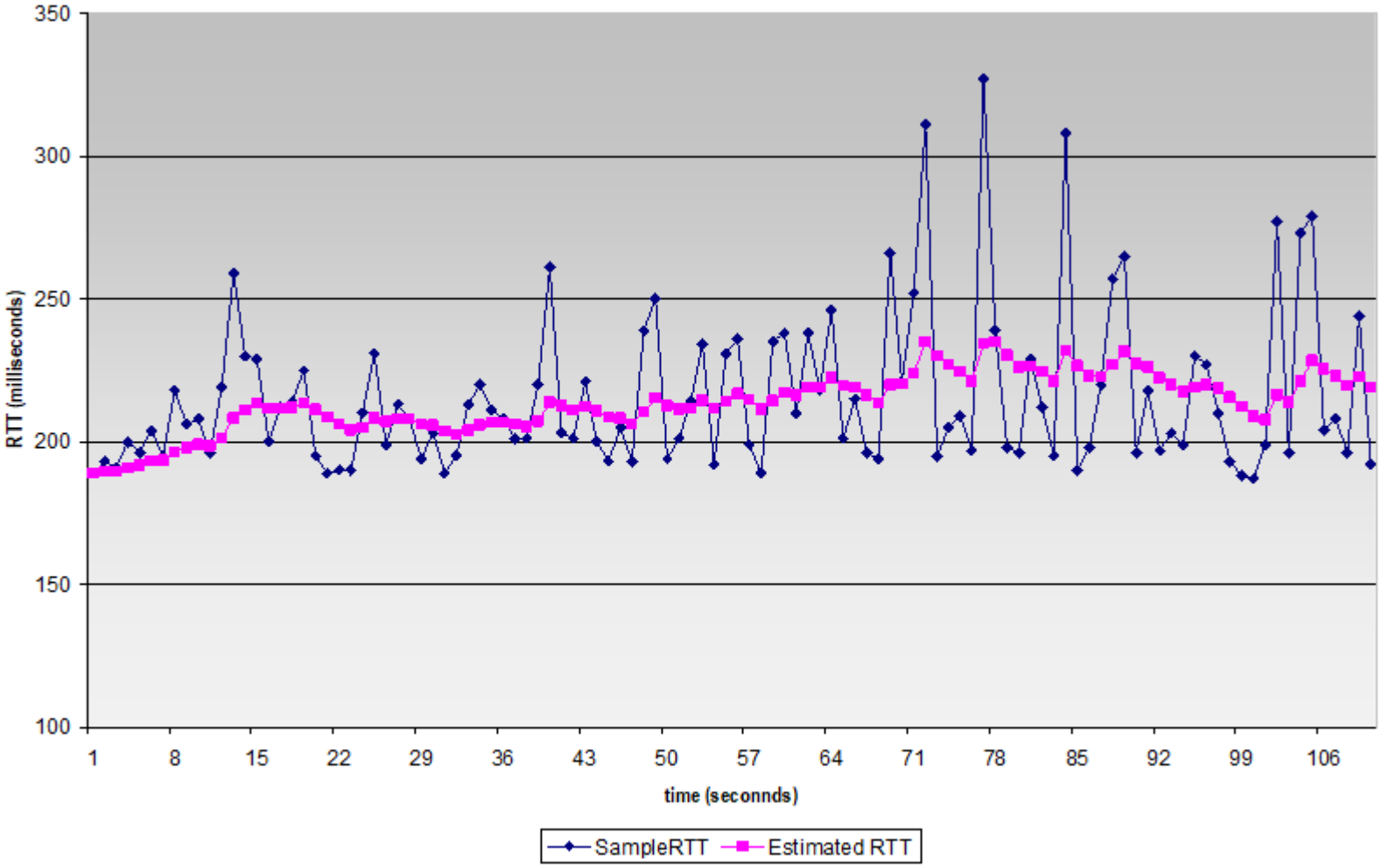
基本RTO估算算法:

$$EstimatedRTT=(1-\alpha)\,EstimatedRTT+\alpha\,SampleRTT \quad (推荐\ \alpha=1/8)$$

$$RTO=\beta * EstimatedRTT \quad (\beta>1,\ 推荐\ \beta=2)$$

- EstimatedRTT是SampleRTT的加权平均
- 最新样本赋予的权值大于老样本的权值（老化算法）
- 越新的样本越能更好地反映网络的当前状况

- RTT估算示例


350
300
250
200
150
100
RTT (milliseconds)
1
8
15
22
29
36
43
50
57
64
71
78
85
92
99
106
time (seconnds)
SampleRTT
Estimated RTT

- TCP重传超时时间计算（续）

基本RTO估算算法存在的问题：

- 没有考虑网络时延的变化情况
- 在实际情况中，网络拥塞情况会对网络时延有很大影响

Jacobson/Karels算法

- 对网络时延变化进行估算

$$EstimatedRTT=(1-\alpha)\ EstimatedRTT+\alpha\ SampleRTT$$

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

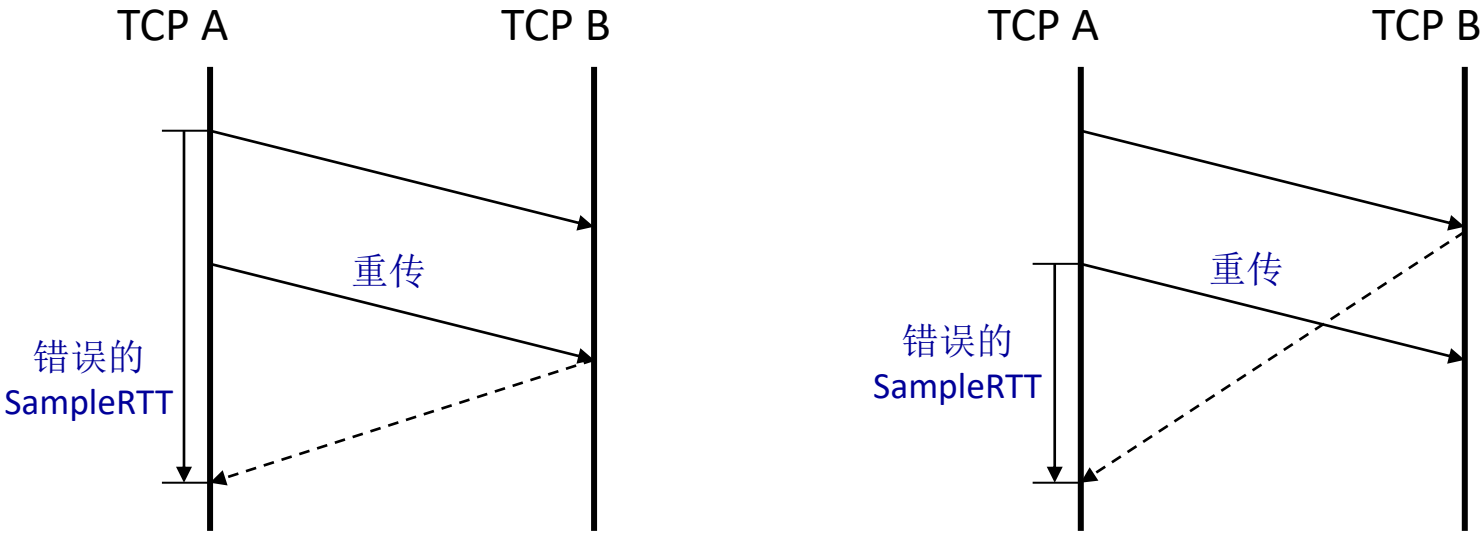
(δ 推荐值为1/4)

$$RTO=\mu*EstimatedRTT+\phi*DevRTT$$

($\mu\approx1$, $\phi\approx4$)

- TCP重传超时时间计算（续）

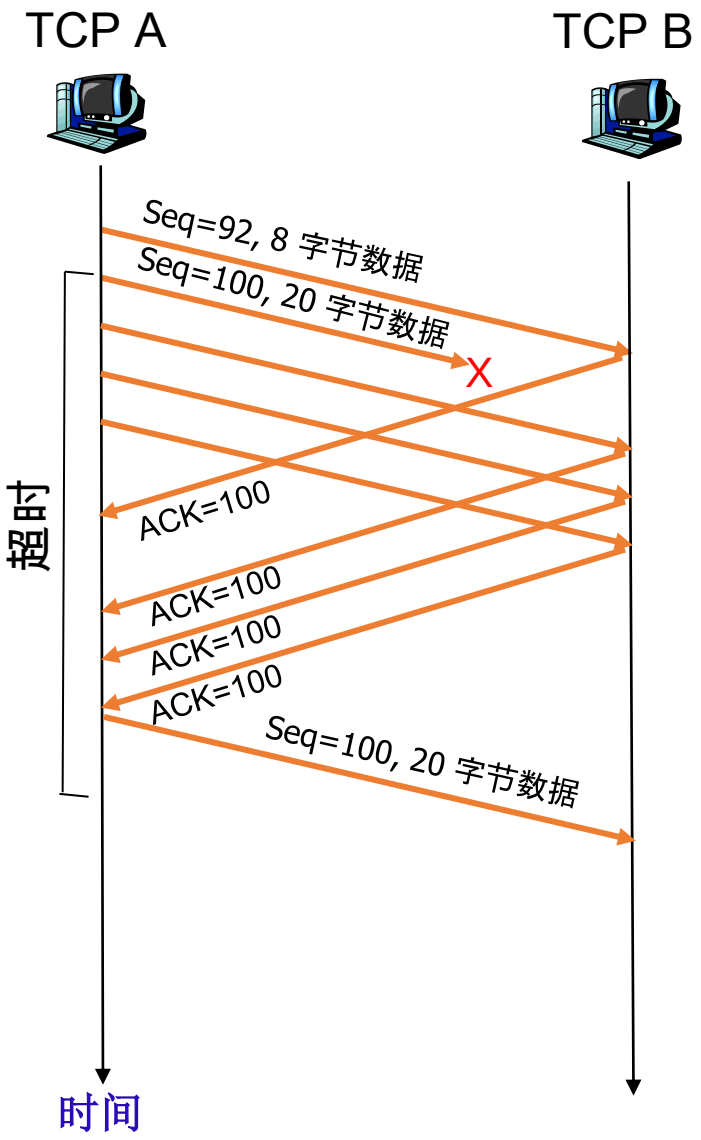
karn/Partridge算法



问题: 当报文段重传时如何估算RTT？

解决策略: 当重传时，不修改估算的RTT，RTO值翻倍

- TCP快速重传
  - RTO时间通常设置的相对较长
    - 重传丢失的TCP段之前引入较长的延时
  - 通过**三次重复ACK**检测TCP段的丢失
    - 有时发送端会连续发出多个TCP段（如大文件传输）
    - 如果一个TCP段丢失，发送端会接收到很多重复的ACK
    - 如果发送端对同一个序列号接收到**三次重复ACK**，则可以假设ACK序列号所指示的TCP段丢失
    - **快速重传**：在超时之前重传丢失的TCP段，以降低延迟



- TCP快速重传算法

```
event: ACK received, with ACK field value of y
        if (y > SendBase) {      /* cumulative ACK of all data up to y */
            SendBase = y
            }
        else {                   /* a duplicate ACK for already ACKed segment */
            increment count of dup ACKs received for y
            if (count of dup ACKs received for y == 3) {
                resend segment with sequence number y
                                          /* TCP fast retransmit */
            }
```

对已经确认的报文段的重复ACK

快速重传

- TCP选择确认（SACK）（RFC 2018）
  - 在建立连接时在选项部分通告**是否支持选择确认**（SACK）

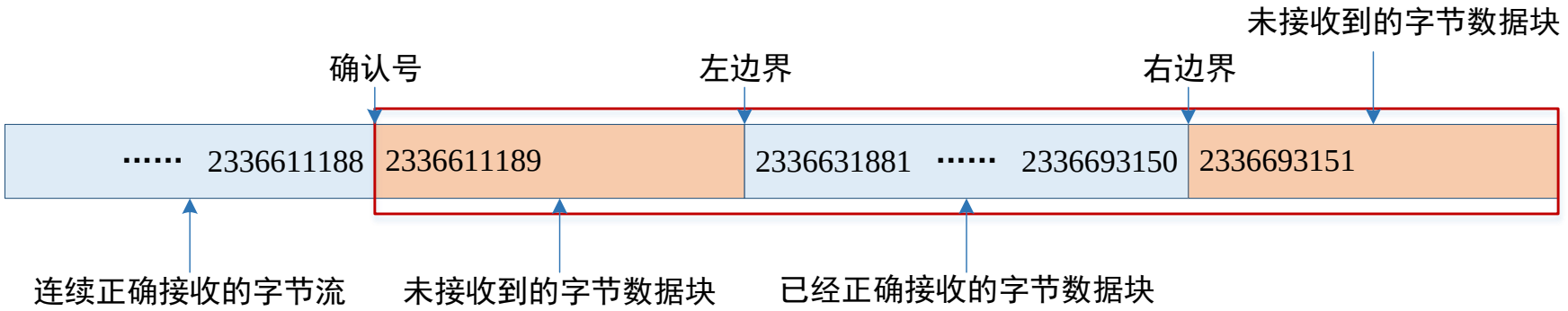
Kind=4	Length=2

  - 确认收到并缓存的**不连续的数据块**。每个块边界参数包含一个**4字节的序号**，其中块左边界表示不连续块的第一个字节的序号，块右边界表示不连续块的最后一个字节的**下一个序号**

Kind=5	Length=N*8+2	第一块左边界	第一块右边界	……	第N块左边界	第N块右边界

TCP选择确认示例

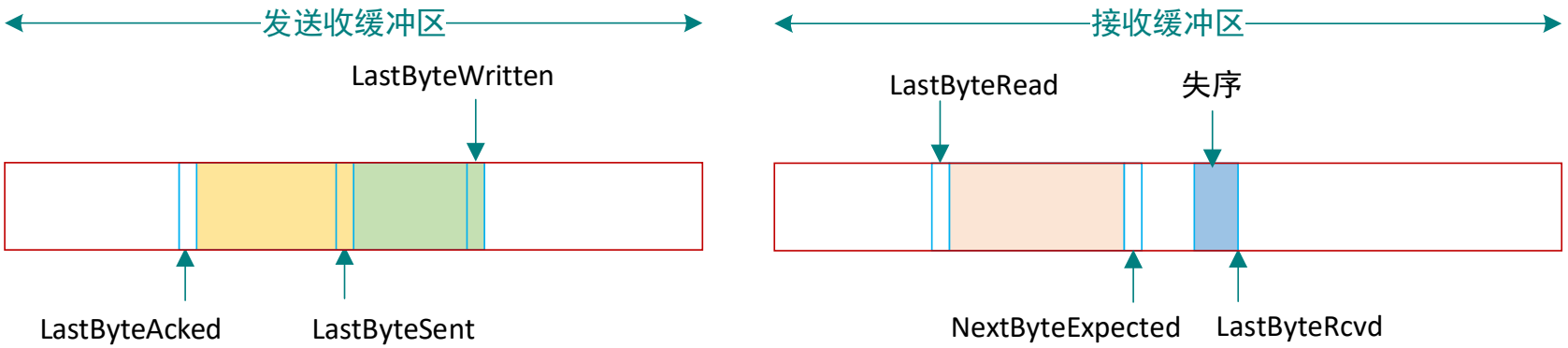
```
Transmission Control Protocol, Src Port: 64233, Dst Port: 443, Seq: 1860549586, Ack: 2336611189, Len: 0
    Source Port: 64233
    Destination Port: 443
    [Stream index: 10]
    [TCP Segment Len: 0]
    Sequence number: 1860549586
    Acknowledgment number: 2336611189                    确认号
    1000 .... = Header Length: 32 bytes (8)              首部长度32字节
  ▷ Flags: 0x010 (ACK)
    Window size value: 65520                             接收窗口
    [Calculated window size: 65520]
    [Window size scaling factor: -1 (unknown)]
    Checksum: 0x8beb [correct]
    [Checksum Status: Good]
    [Calculated Checksum: 0x8beb]
    Urgent pointer: 0
  ◢ Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), SACK    tcp选项
    ▷ TCP Option - No-Operation (NOP)
    ▷ TCP Option - No-Operation (NOP)
    ◢ TCP Option - SACK 2336631881-2336693151
        Kind: SACK (5)                                   这是一个选择确认SACK
        Length: 10                                       选项长度
        left edge = 2336631881                           左边界 —— 4字节
        right edge = 2336693151                          右边界 —— 4字节
        [TCP SACK Count: 1]
```



3.5 传输控制协议TCP —流量控制

- TCP流量控制
  - **回顾**：TCP流水线机制中发送端可以发出未确认的字节数受发送缓冲区大小的约束
    - 这部分数据必须由发送端缓存，以备重传
  - **流量控制的目的**：避免发送端发送数据过快，接收端不能及时处理，造成接收缓冲区溢出
    - 增加流控功能后，发送窗口还需要受接收能力的约束
  - **可变的滑动窗口**：接收端利用“**接收窗口通告**”域段告知发送端接收端缓冲区剩余的空间，发送端依据该通告调整**发送窗口**的大小

- TCP流量控制：滑动窗口



发送端

- LastByteAcked ≤ LastByteSent
- LastByteSent ≤ LastByteWritten
- 缓存的数据
  =LastByteWritten−LastByteAcked

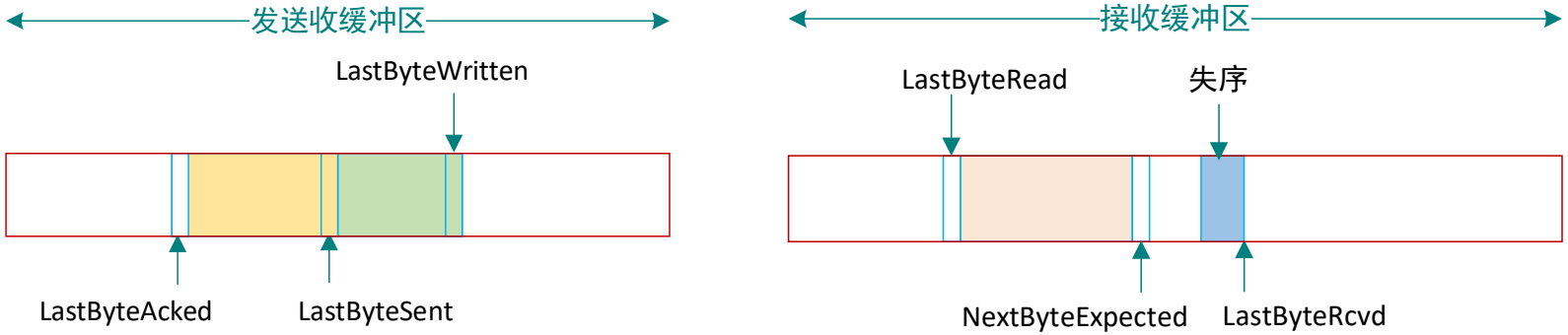
接收端

- LastByteRead < NextByteExpected
- NextByteExpected ≤ LastByteRcvd
- 缓存的数据
  =LastByteRcvd−LastByteRead

3.5 传输控制协议TCP —流量控制

发送缓冲区大小: MaxSendBuffer
接收缓冲区大小: MaxRcvBuffer
接收窗口通告: AdvertisedWindow

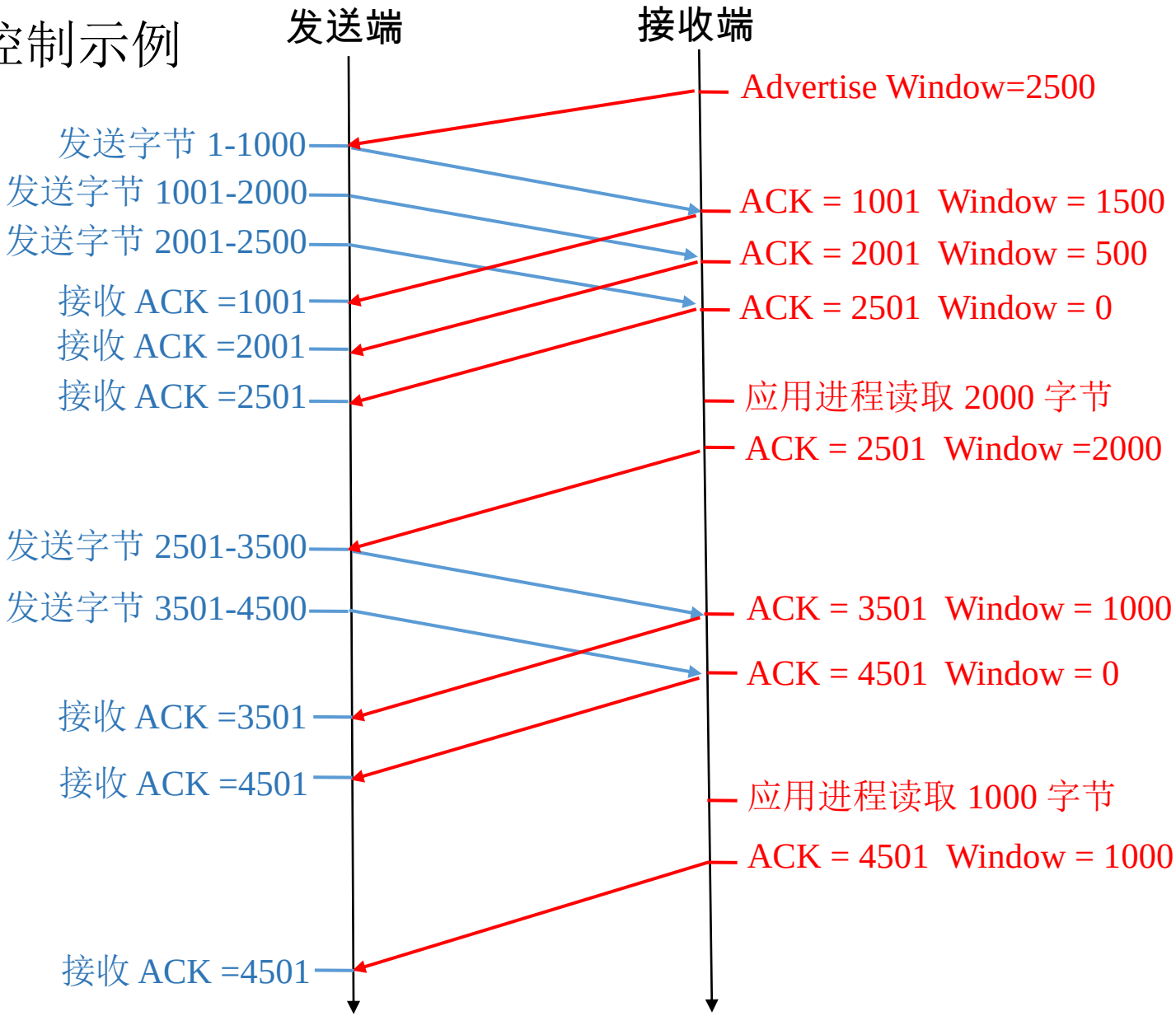
- TCP流量控制：流量控制约束



- 接收端
  - AdvertisedWindow = MaxRcvBuffer - (NextByteExpected – LastByteRead-1)
- 发送端
  - LastByteSent - LastByteAcked ≤ AdvertisedWindow
  - EffectiveWindow=AdvertisedWindow - (LastByteSent - LastByteAcked)
  - LastByteWritten - LastByteAcked ≤ MaxSendBuffer
  - 如果 (LastByteWritten - LastByteAcked) + y > MaxSenderBuffer，则阻塞发送

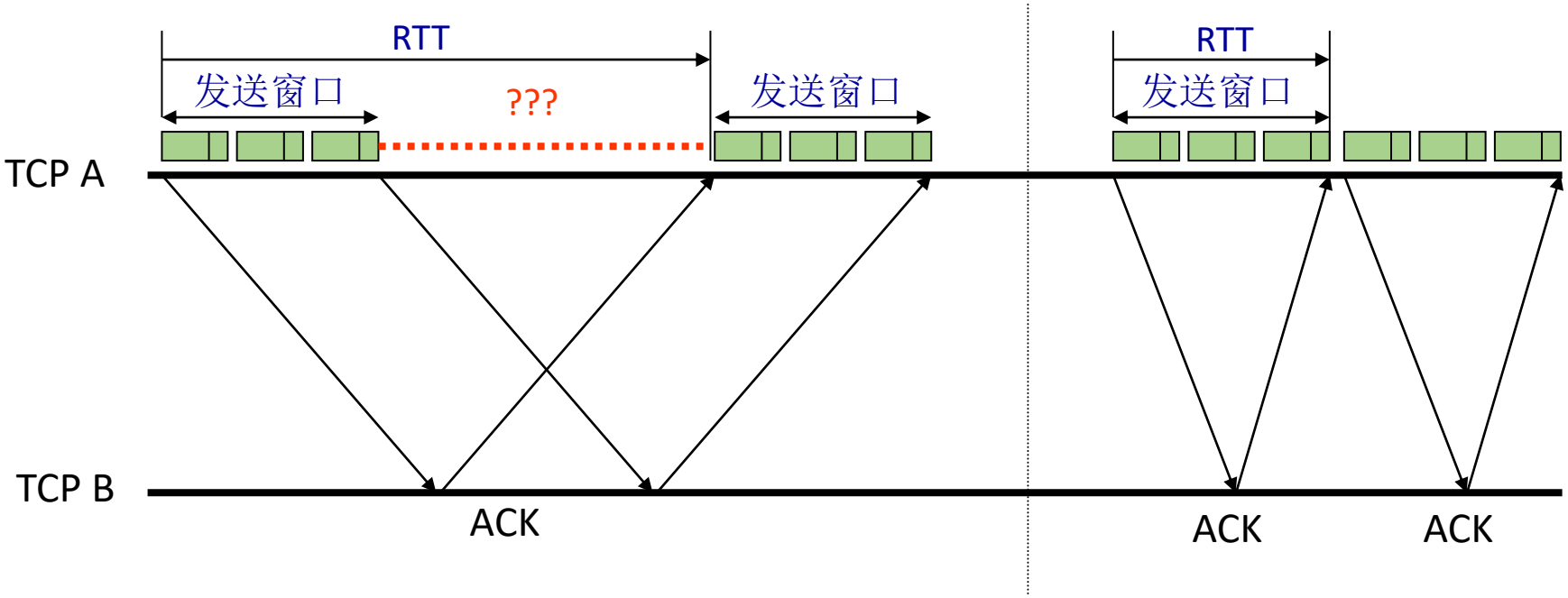
3.5 传输控制协议TCP—流量控制

- 流量控制示例



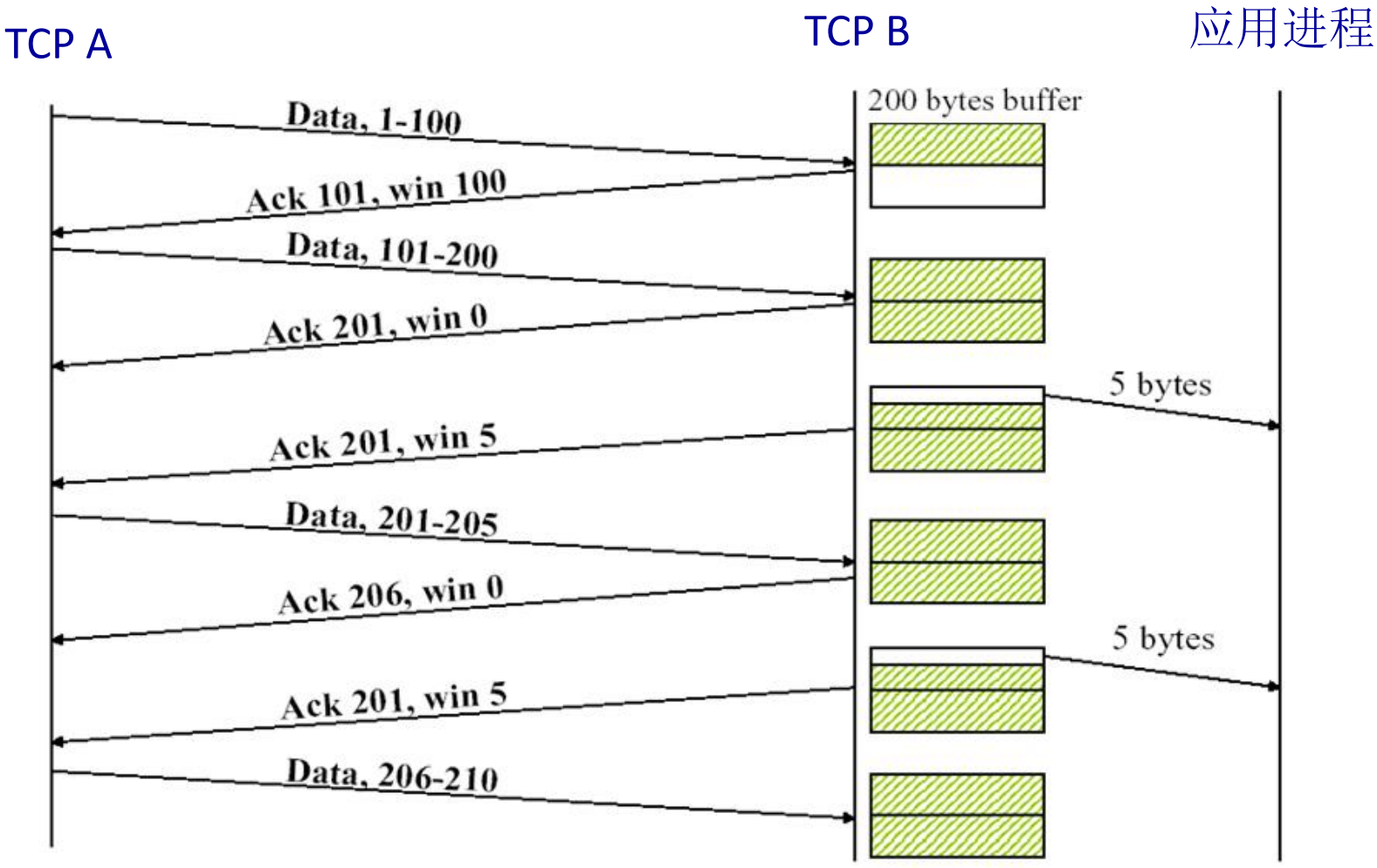
3.5 传输控制协议TCP—流量控制

- 接收窗口大小对性能的影响



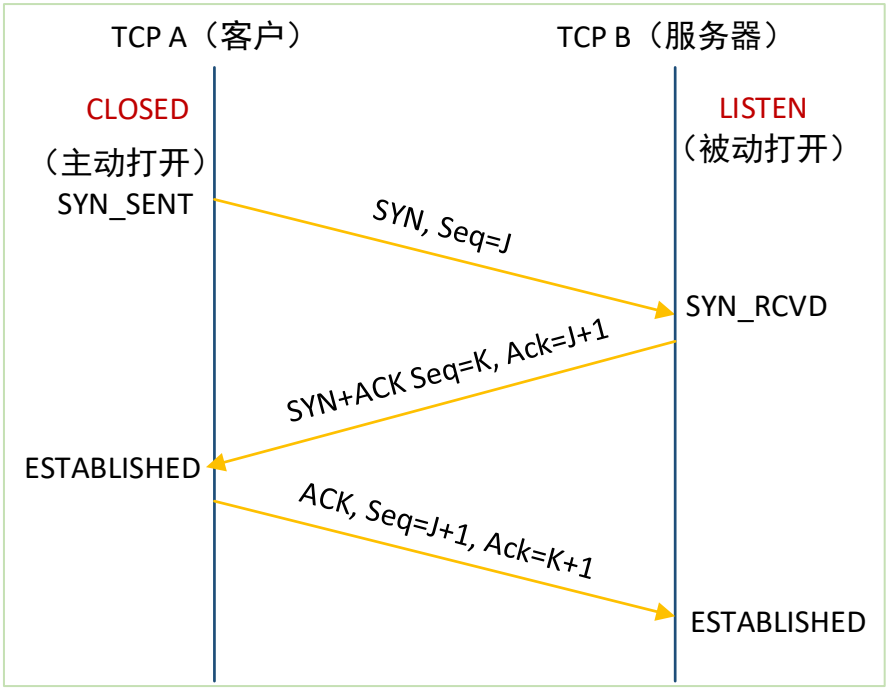
- 较小的缓冲区会影响网络的吞吐率
- 过大的缓冲区会浪费主机的存储资源

- 流量控制的其他性能问题



考虑发送端和接收端有哪些解决策略？

TCP连接建立



0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7
源端口号（Source Port）		目的端口号（Destination Port）	
发送序列号（sequence number）			
确认序列号（length）			
头长度	未用	U A P R **S** F	接收窗口通告（rcvr window size）
校验和（checksum）			紧急数据指针（ptr urgent data）
选项（options）			

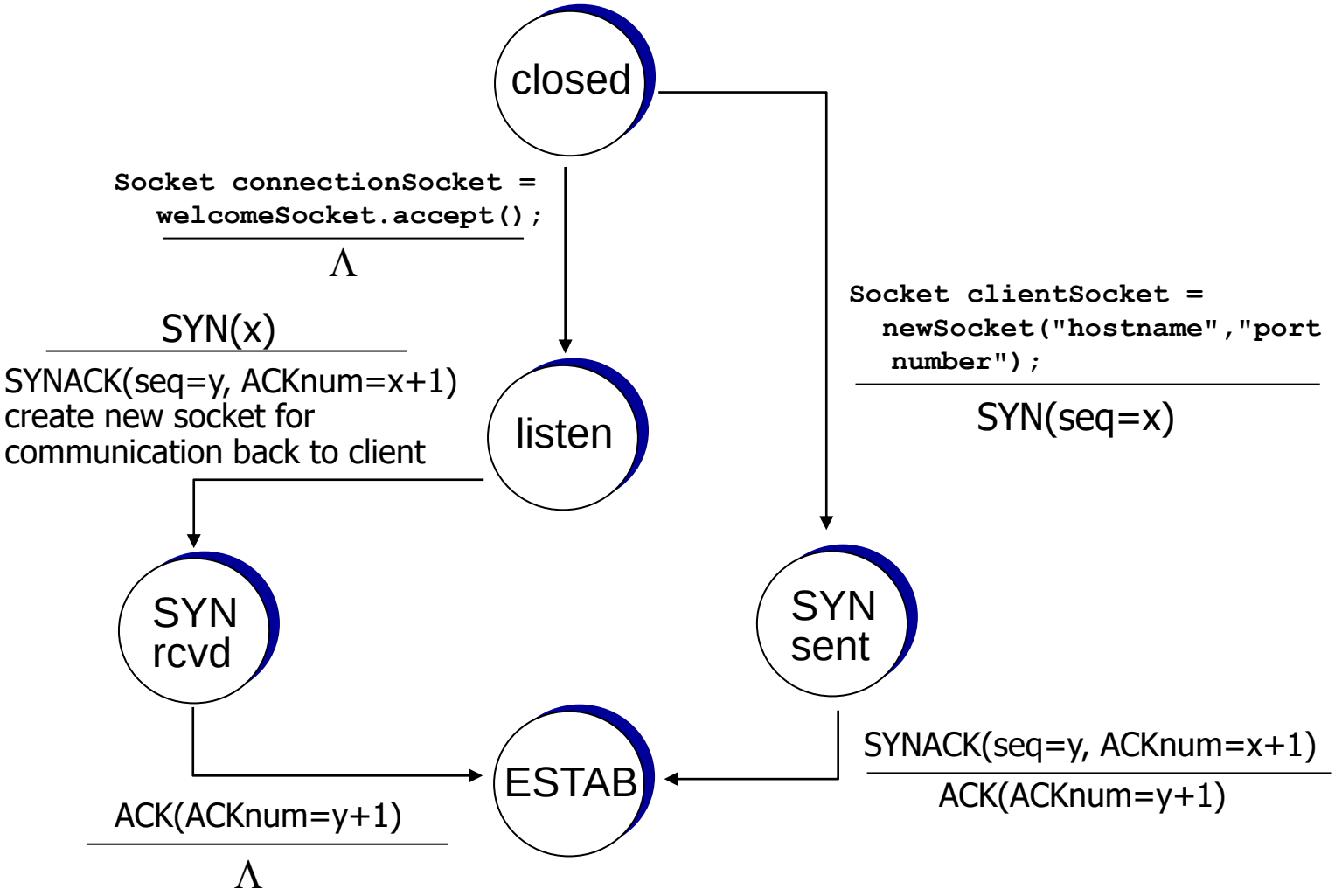
0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7
源端口号（Source Port）		目的端口号（Destination Port）	
发送序列号（sequence number）			
确认序列号（length）			
头长度	未用	U **A** P R **S** F	接收窗口通告（rcvr window size）
校验和（checksum）			紧急数据指针（ptr urgent data）
选项（options）			

- 步骤一：TCP A端发送SYN段
  - **分配缓冲区**，选择A→B初始序列号（ISN， Initial Sequence Number ）
- 步骤二：TCP B接收SYN段，回送SYN+ACK段，
  - **分配缓冲区**，选择B→A的初始序列号
- 步骤三：TCP A回送ACK

SYN洪泛攻击？

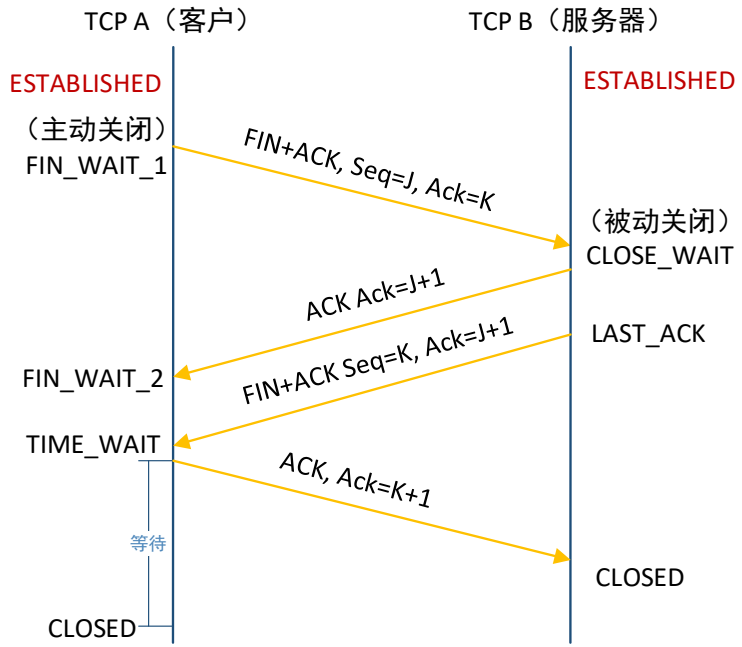
3.5 传输控制协议TCP-连接管理

- TCP连接建立状态机



3.5 传输控制协议TCP-连接管理

TCP连接关闭

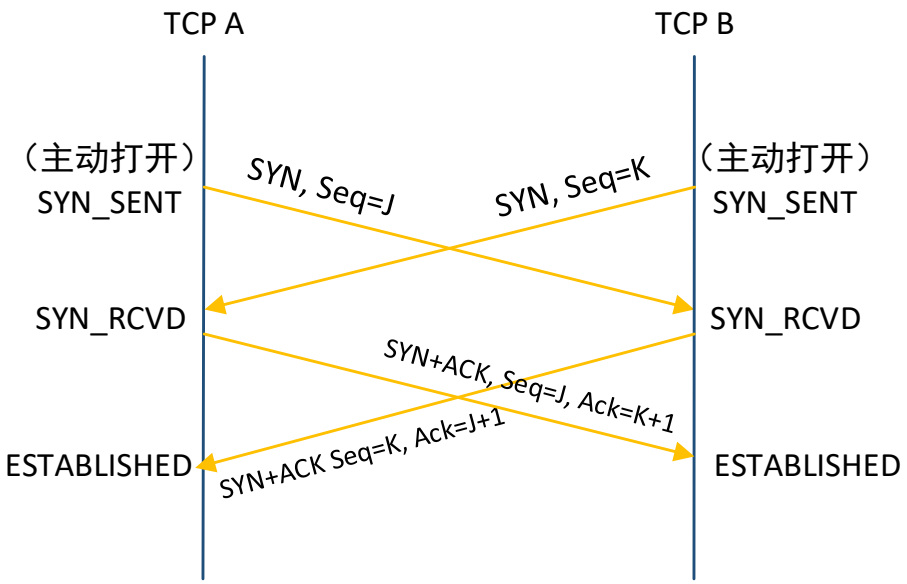


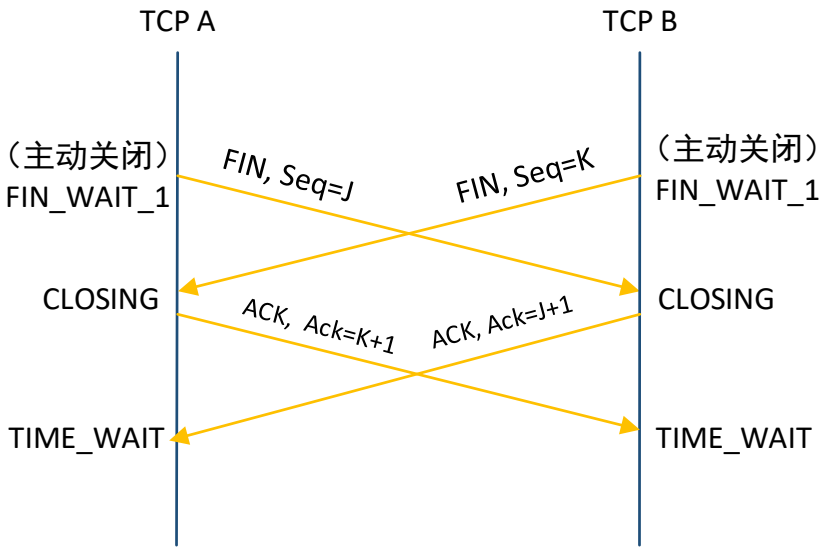
0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7
源端口号（Source Port）		目的端口号（Destination Port）	
发送序列号（sequence number）			
确认序列号（ACK number）			
头长度	未用 U **A** P R S **F**	接收窗口通告（rcvr window size）	
校验和（checksum）		紧急数据指针（ptr urgent data）	

0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7
源端口号（Source Port）		目的端口号（Destination Port）	
发送序列号（sequence number）			
确认序列号（ACK number）			
头长度	未用 U **A** P R S F	接收窗口通告（rcvr window size）	
校验和（checksum）		紧急数据指针（ptr urgent data）	

- 步骤一：TCP A端发送FIN段
  - A不再向B发送数据，但仍可以接收数据
- 步骤二：TCP B端接收FIN段，回送ACK段
  - B仍可以向A发送数据
- 步骤三：TCP B端发送FIN段，等待TCP A端返回ACK
- 步骤四：TCP A端接收FIN段，回送ACK段，等待两倍的段生存期关闭连接；TCP B端接收ACK，关闭连接

- 同时打开与同时关闭连接





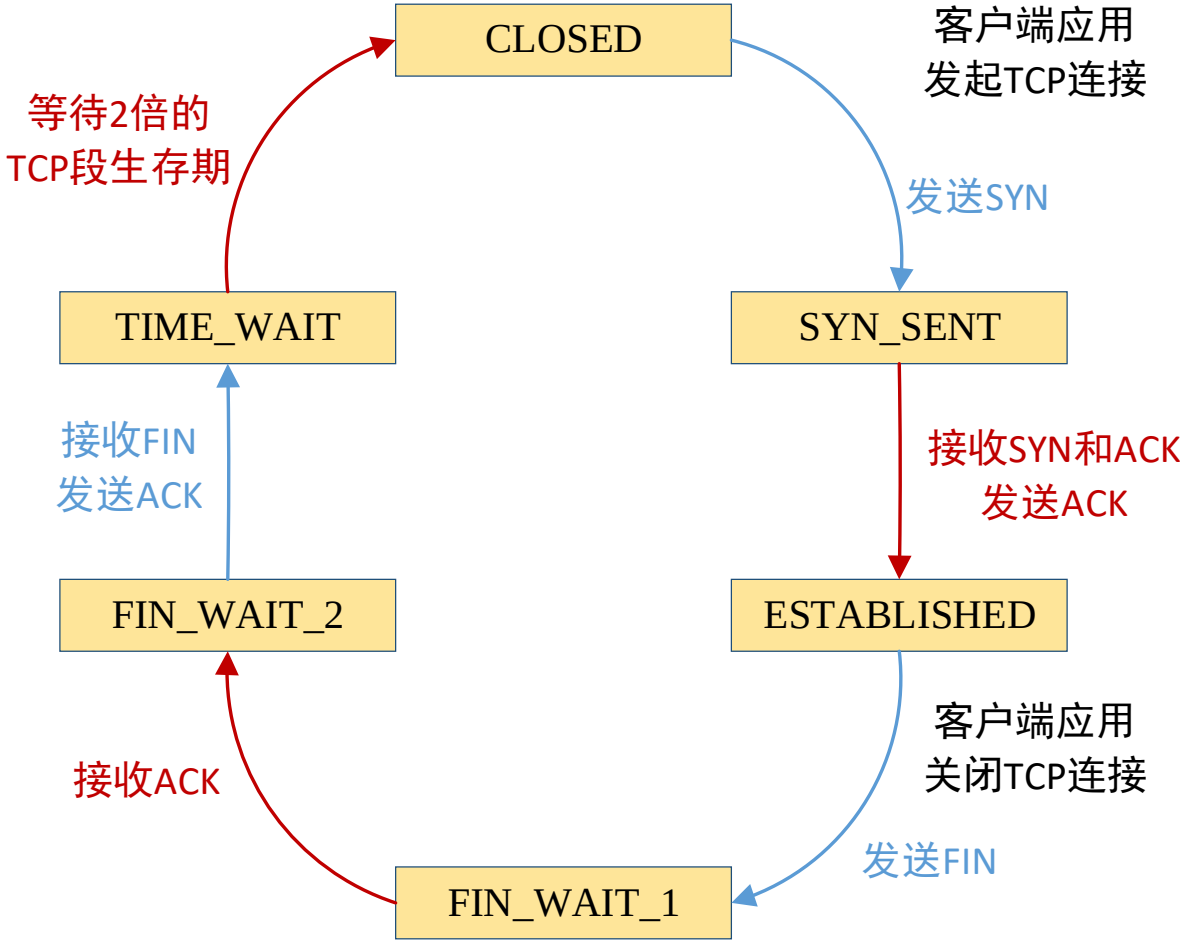
- 连接的半打开状态：连接的一端存在、而另一端不存在
  - ✓ 当一个进程终止连接未能通知到另一方时，例如：掉电、异常关闭等
  - ✓ 如何解决？

TCP定时器:

- 连接建立定时器（75秒）
- 重传定时器（RTO）
- 延迟ACK定时器（500毫秒）
- 持续定时器（避免0窗口死锁）
- 保活定时器（避免半打开）
- 静默定时器（避免端口号重用等）

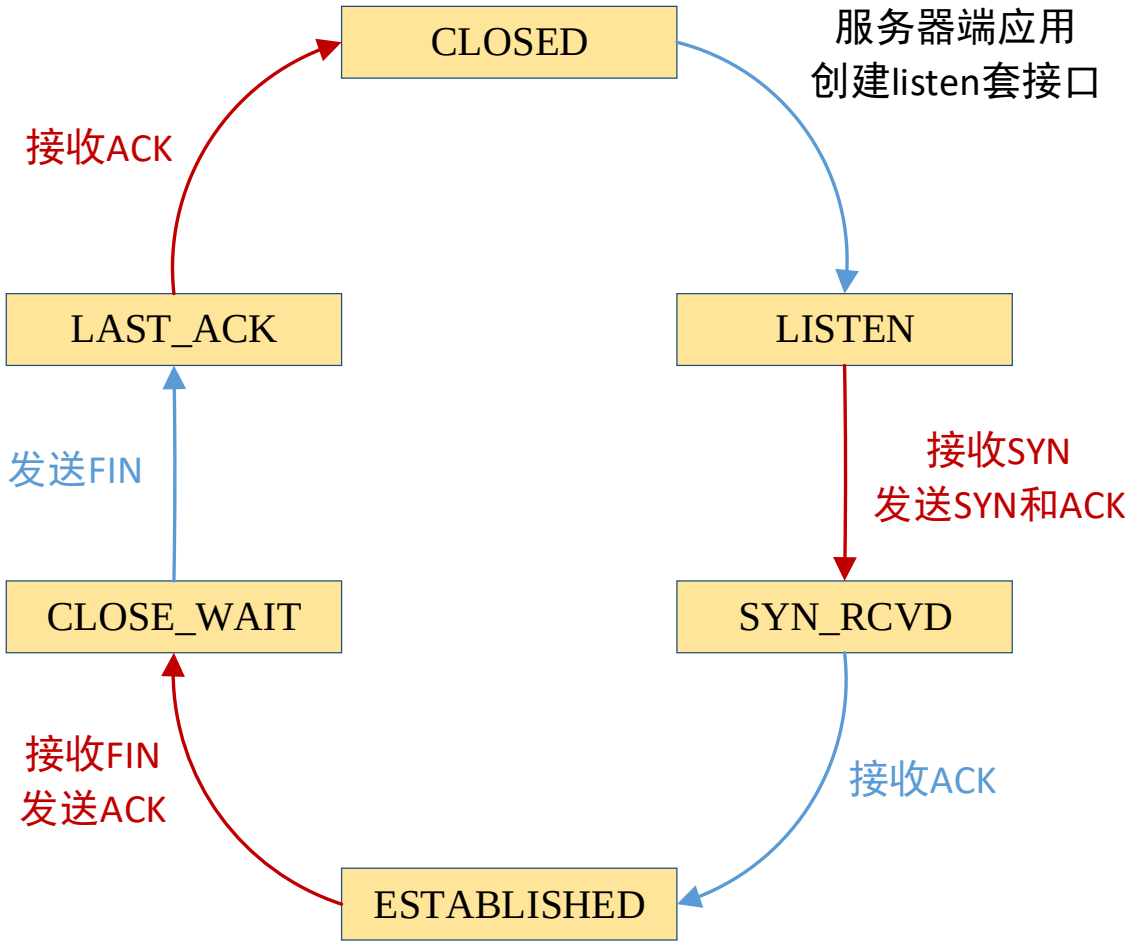
3.5 传输控制协议TCP-连接管理

- TCP客户端生命周期



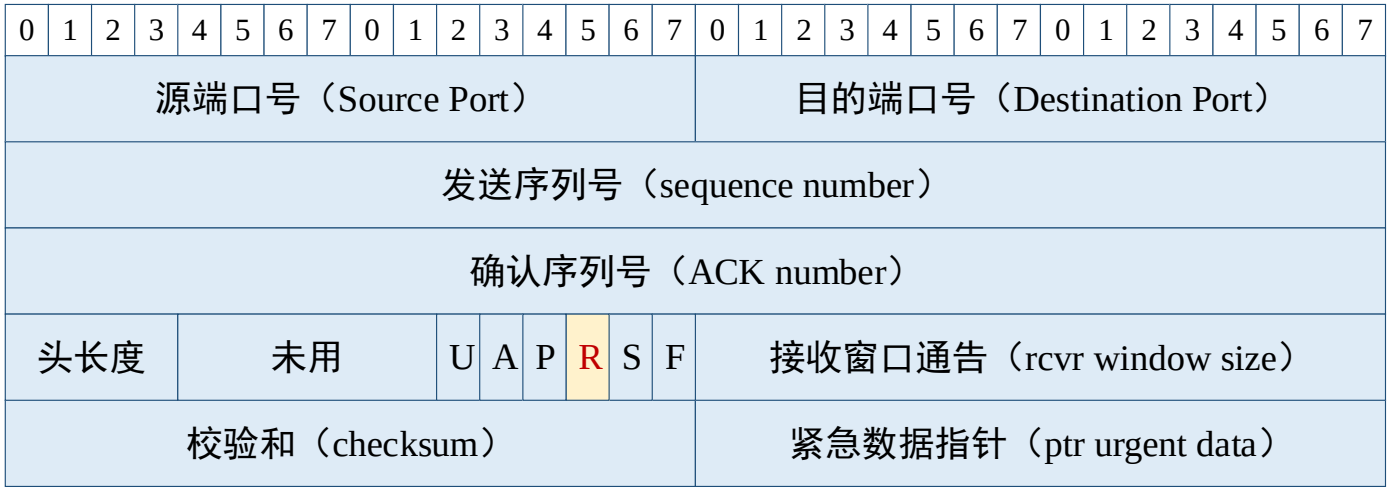
3.5 传输控制协议TCP-连接管理

- TCP服务器端生命周期



TCP Reset报文段的使用

- ✓ 发送连接请求到**没有进程监听**（处于LISTEN状态）的端口
- ✓ 客户端和服务器的某一方在交互的过程中**发生异常**，该方系统将向对端发送Reset报文段，告之对方释放相关的TCP连接。
- ✓ 接收端收到TCP段，但是发现该TCP段并不**在其已建立的TCP连接列表内**，则其直接向对端发送Reset报文段



TCP Reset报文段的使用（续）

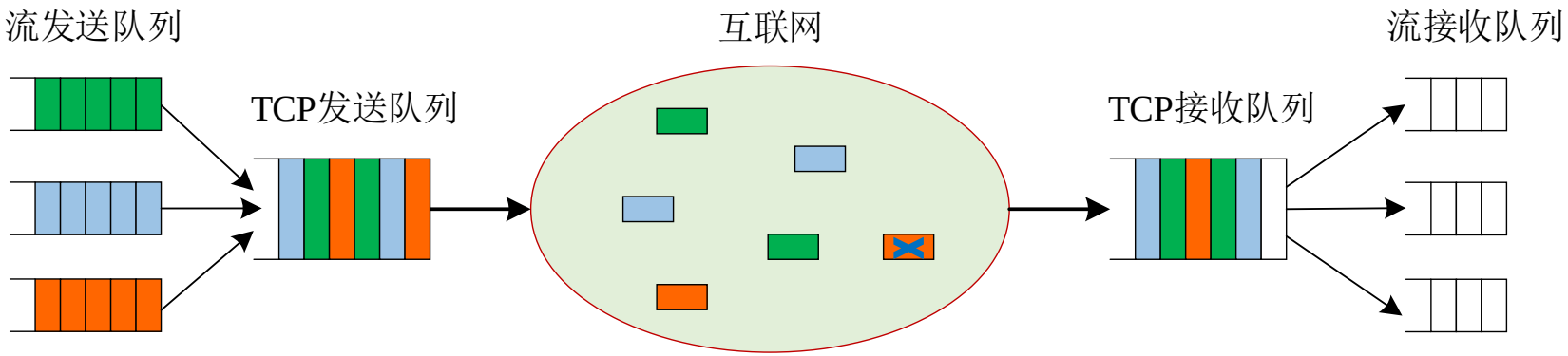
- ✓ 在交互的双方中的某一方**长期未收到来自对方的确认报文**，则其在**超出一定的重传次数或时间后**，会主动向对端发送Reset报文段释放该TCP连接
- ✓ 有些应用开发者在设计应用系统时，**会利用Reset报文段快速释放已经完成数据交互的TCP连接，以提高业务交互的效率**

<table>
<tr><td colspan="16">0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7</td><td colspan="16">0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7</td></tr>
<tr><td colspan="16">源端口号（Source Port）</td><td colspan="16">目的端口号（Destination Port）</td></tr>
<tr><td colspan="32">发送序列号（sequence number）</td></tr>
<tr><td colspan="32">确认序列号（ACK number）</td></tr>
<tr><td colspan="8">头长度</td><td colspan="8">未用</td><td>U</td><td>A</td><td>P</td><td>R</td><td>S</td><td>F</td><td colspan="16">接收窗口通告（rcvr window size）</td></tr>
<tr><td colspan="16">校验和（checksum）</td><td colspan="16">紧急数据指针（ptr urgent data）</td></tr>
</table>

TCP问题讨论

✓ TCP接收端队头阻塞问题

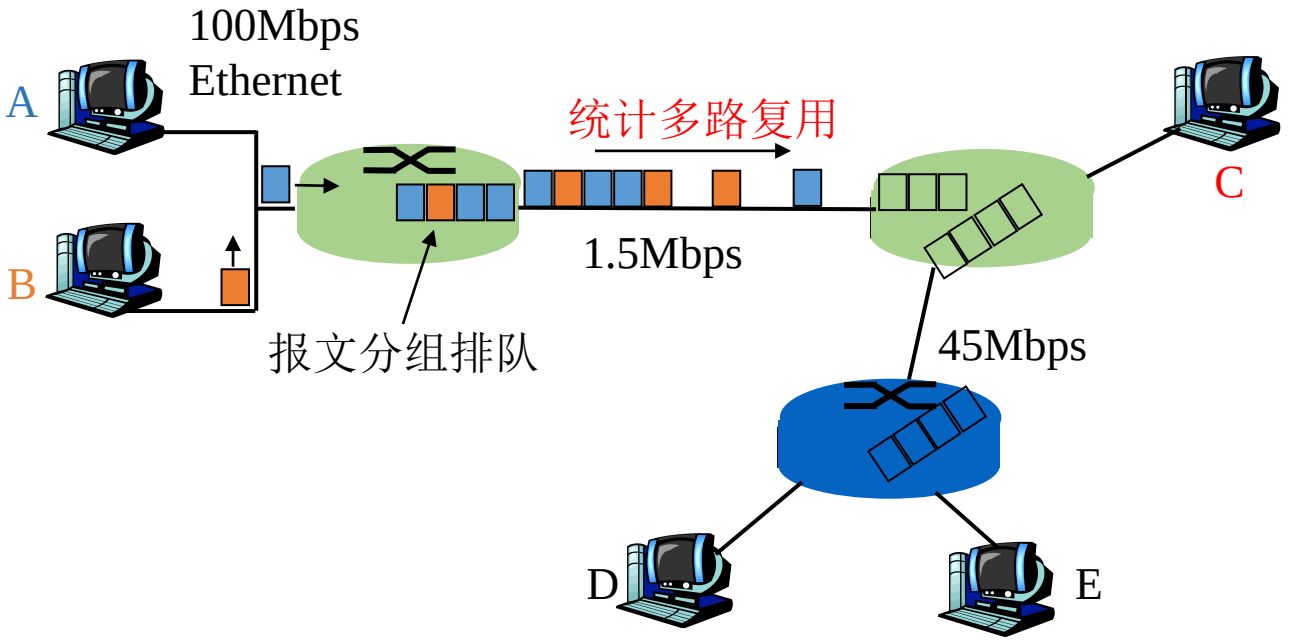
- TCP提供面向字节流的可靠服务，接收的数据要与发送端一致的顺序提供给应用进程
- 当有数据段丢失时，需要超时重传，对于接收队列存在队头阻塞问题，会影响应用协议性能



使用多TCP连接，或设计UDP之上的可靠数据传输，如QUIC协议

拥塞控制

✓ **回顾**：当报文分组到达时，如果出口链路忙，报文分组需要在路由器缓存中排队等待，**引入排队时延**；如果缓冲填满，报文分组会被**丢弃**；**过长的排队时延和丢弃会对网络性能会产生较大的影响**

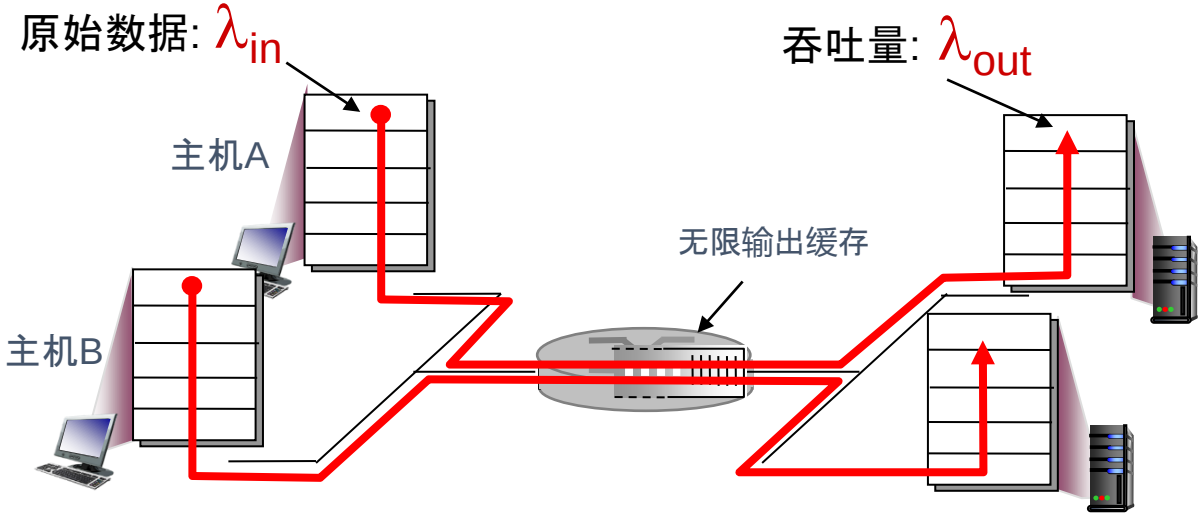


3.6 理解网络拥塞

拥塞控制

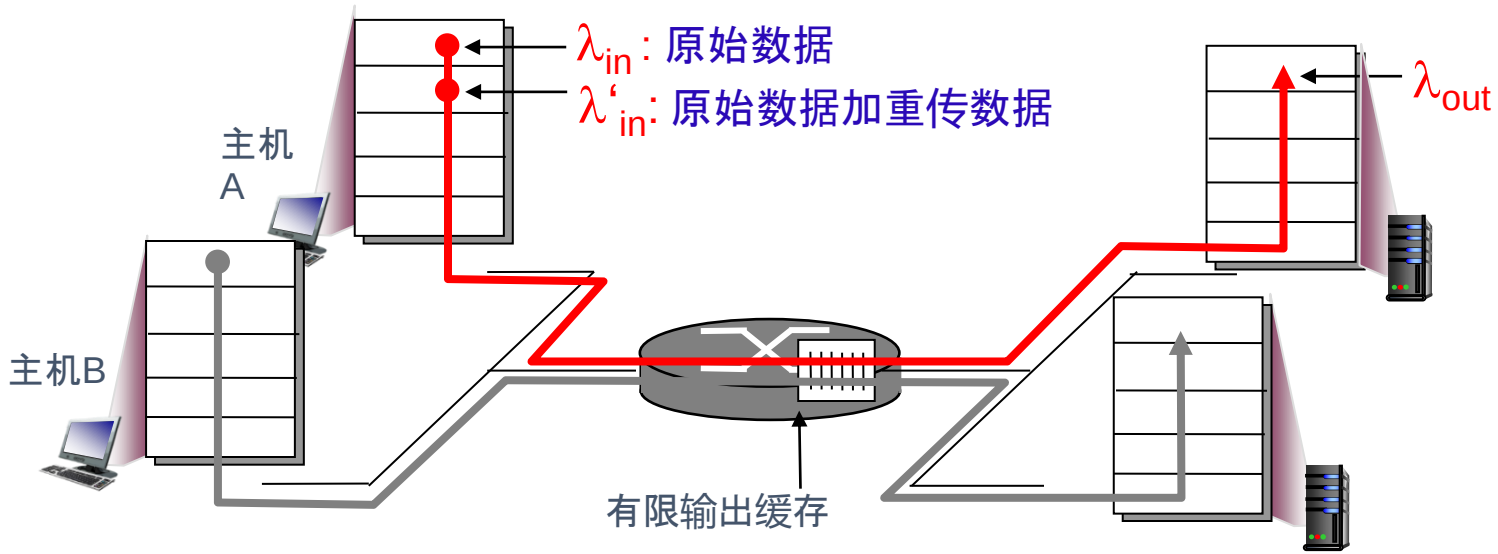
- ✓ **网络拥塞：**主机发送的数据**过多或过快**，造成网络中的路由器（或其他设备）无法及时处理，从而**引入时延或丢弃**
- ✓ 存储转发交换所采用的统计多路复用机制中，拥塞不可避免
  - 统计多路复用可以有效利用链路带宽资源
  - 轻度拥塞可以接受，中度或严重拥塞需要避免
- ✓ **拥塞控制也是网络中的Top 10问题**
- ✓ **注意：拥塞控制与流量控制的区别**

- 拥塞的代价：情景1
  - 两个发送端、两个接收端
  - 一台路由器，无限缓存
  - 输出链路的容量：R
  - 无丢失、无重传，可能会造成高时延



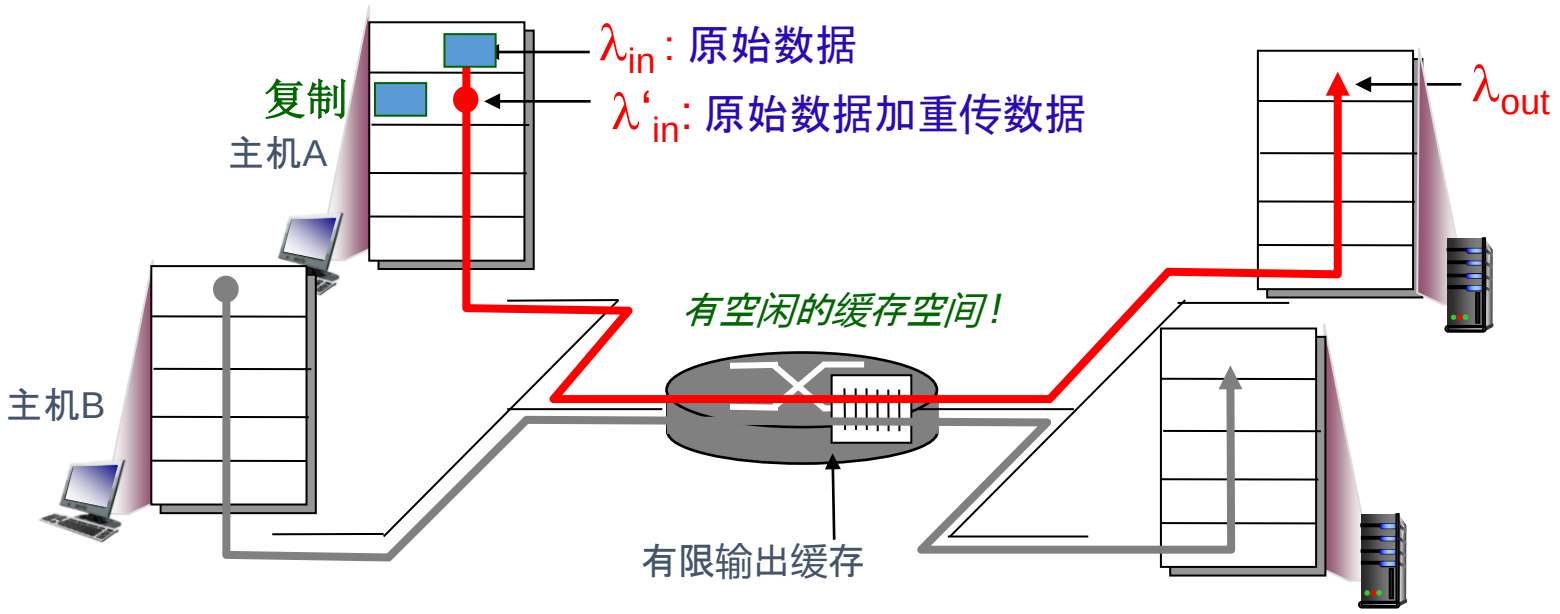
拥塞的代价：情景2

- 一台路由器，有限缓存
- 发送端重传超时的报文分组
  - ✓ 应用层输入=应用层输出：$\lambda_{in} = \lambda_{out}$
  - ✓ 传输层输入包括了重传报文分组：$\lambda'_{in} \geq \lambda_{in}$



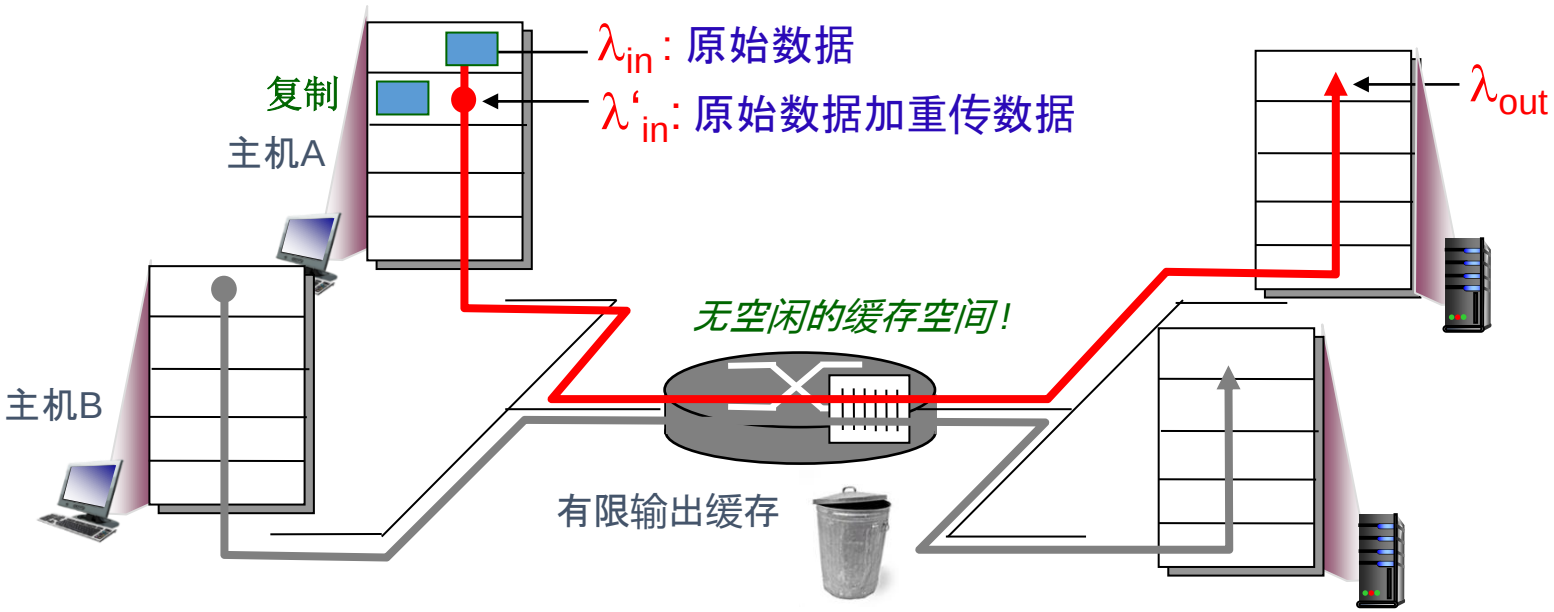
在这种情况下实现的性能在很大程度上取决于如何进行重传

- 拥塞的代价：情景2
  - 理想情况：发送端对路由器中的缓存是否空闲已知



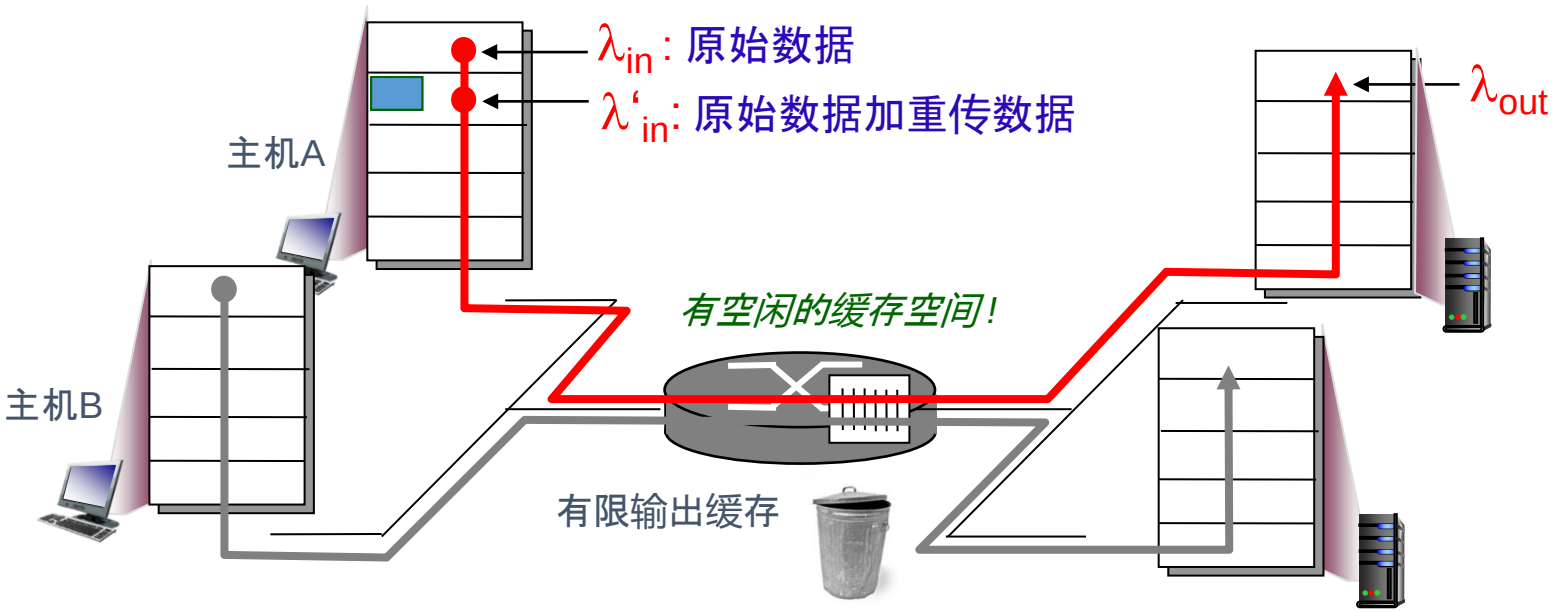
拥塞的代价：情景2

- 理想情况：发送端可以获知路由器中报文分组的丢失
  - ✓ 路由器缓存满时，报文分组会丢失
  - ✓ 发送端只重传已知丢失的报文分组



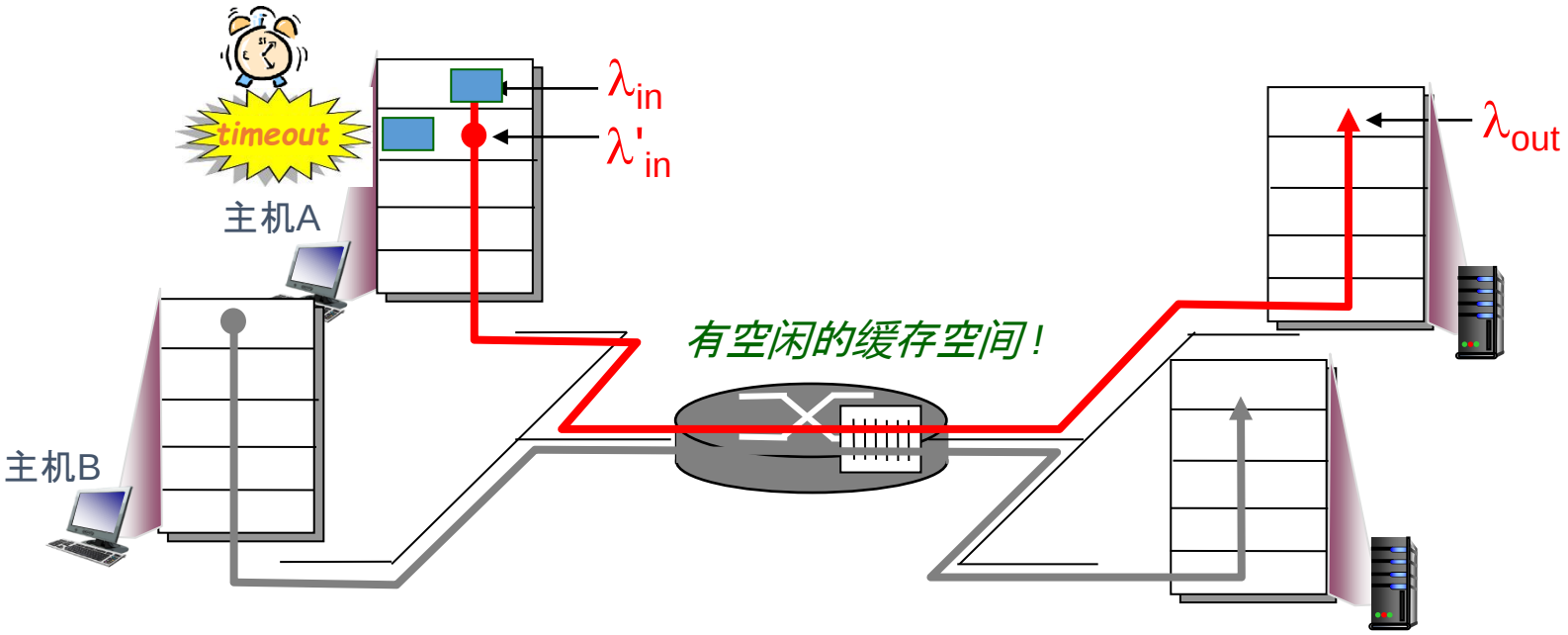
拥塞的代价：情景2

- 理想情况：发送端可以获知路由器中报文分组的丢失
  - ✓ 路由器缓存满时，报文分组会丢失
  - ✓ 发送端只重传已知丢失的报文分组

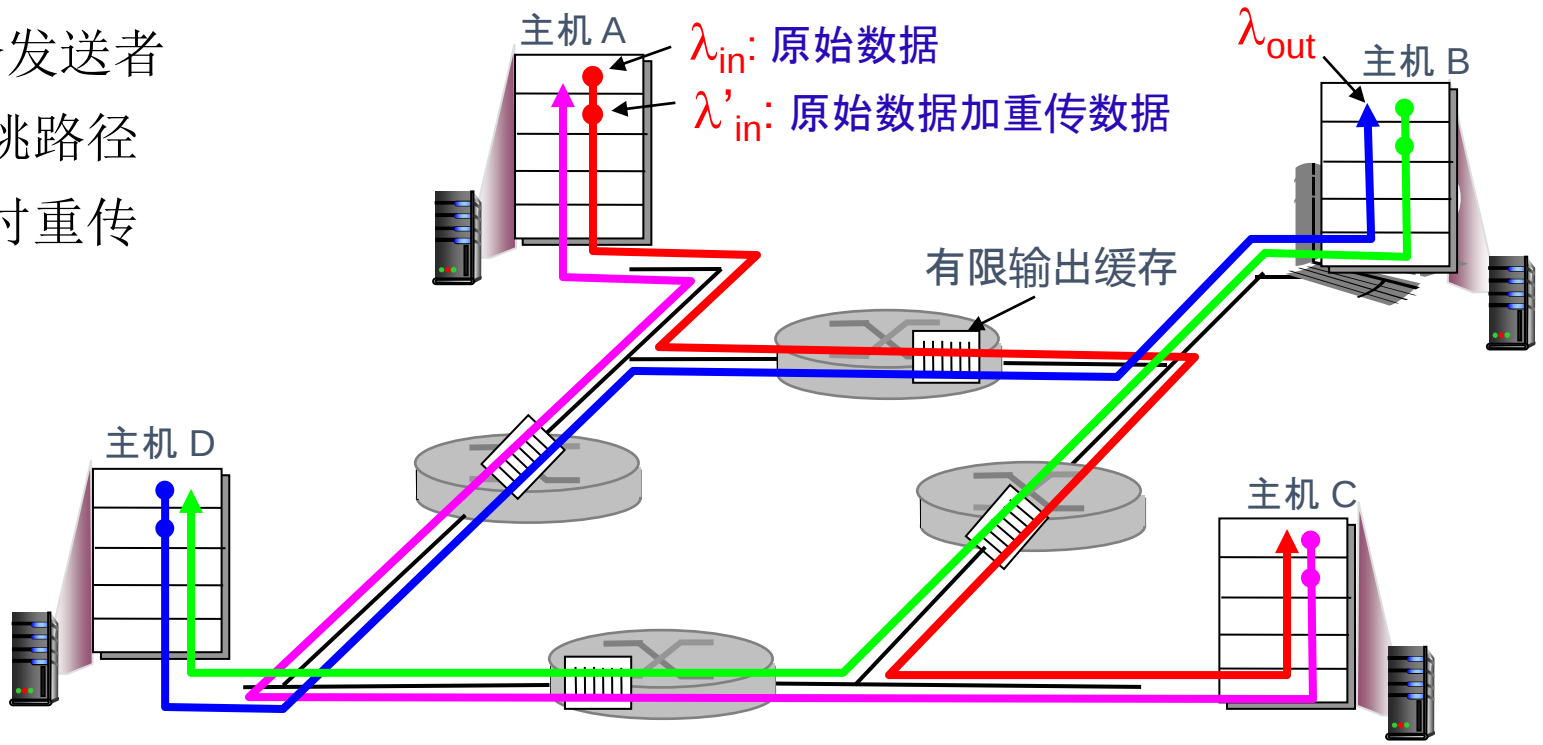


拥塞的代价：情景2

- 实际情况：会产生重复的报文分组
  - ✓ 路由器缓存满时，报文分组会丢失
  - ✓ 发送端提前超时，会产生两个相同的报文分组



- 拥塞的代价：情景3
  - 4个发送者
  - 多跳路径
  - 超时重传



问题： 随着λ_{in}和λ_{in}'的增加会发生什么？

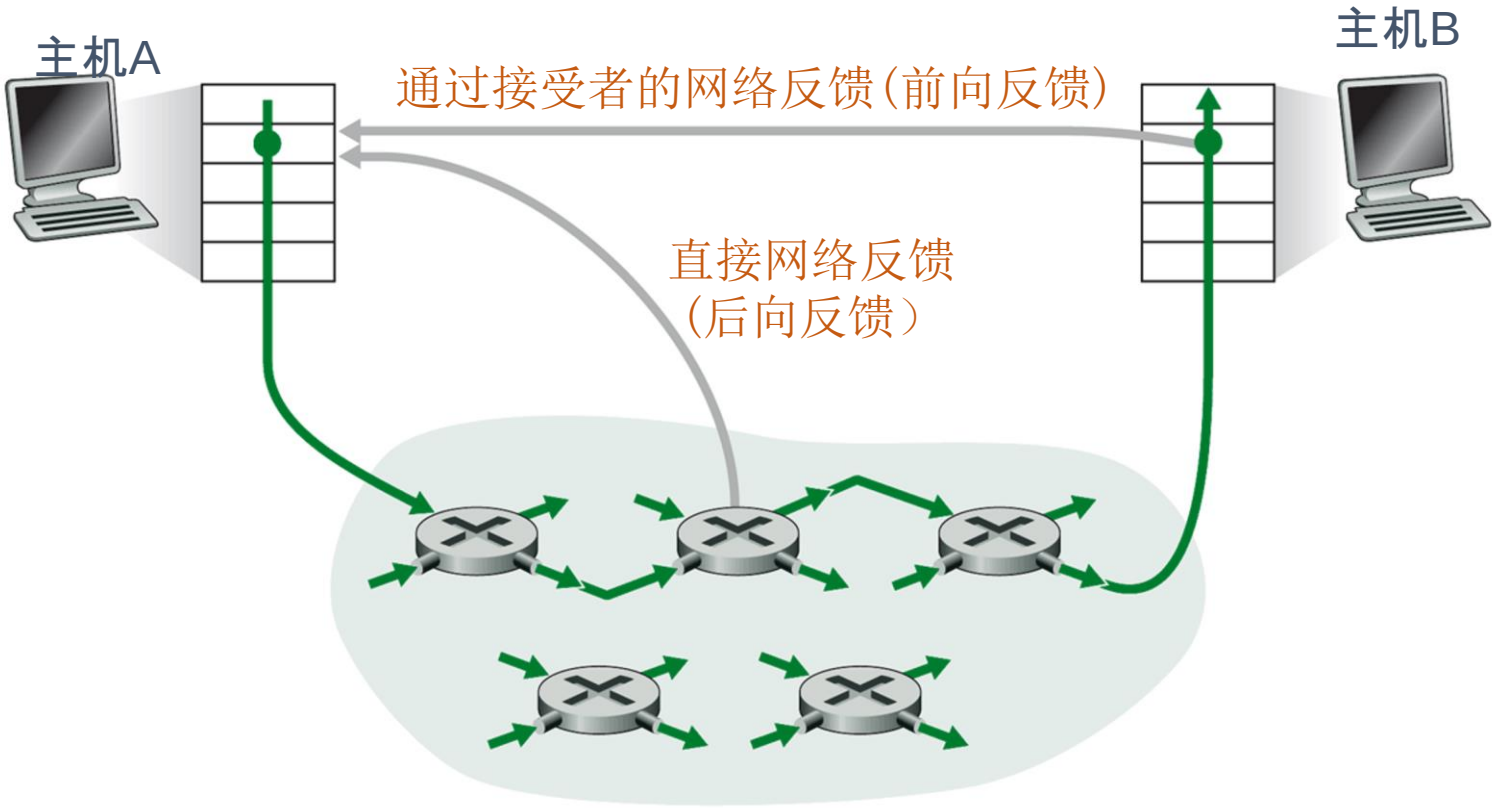
结果： 随着红色 λ_{in}' 增加，所有到达的蓝色报文分组可能会被丢弃，蓝色连接的吞吐率可能降为0

3.6 理解网络拥塞

- 拥塞控制方法：两种广泛使用的拥塞控制方法
  - 端到端拥塞控制
    - ✓ 网络中无明确的反馈
    - ✓ 端系统通过观察丢失、延迟推断是否发生拥塞
    - ✓ TCP采用的拥塞控制策略
  - 网络辅助的拥塞控制
    - ✓ 路由器提供到端系统的反馈
    - ✓ 例如：可以使用1位指示拥塞（如X.25, ATM）

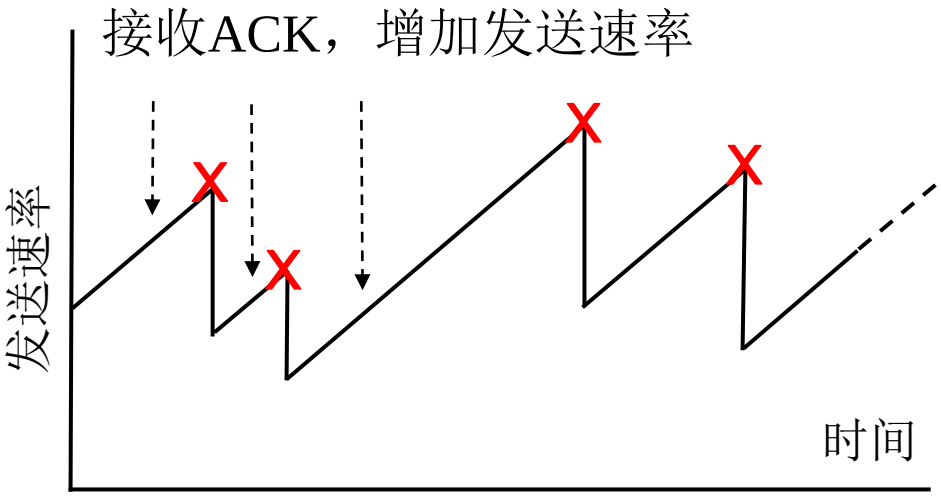
3.6 理解网络拥塞

- 拥塞控制方法：网络辅助的拥塞控制



3.7 TCP拥塞控制

- **目标：**既不造成网络严重拥塞，又能更快地传输数据
  - **关键问题：发送多快比较恰当？**
- **带宽探测：**接收到ACK提高传输速率，发生丢失事件降低传输速率
  - ACK返回：说明网络并未拥塞，可以继续提高发送速率
  - 丢失事件：**假设所有丢失是由于拥塞造成的**，降低发送速率

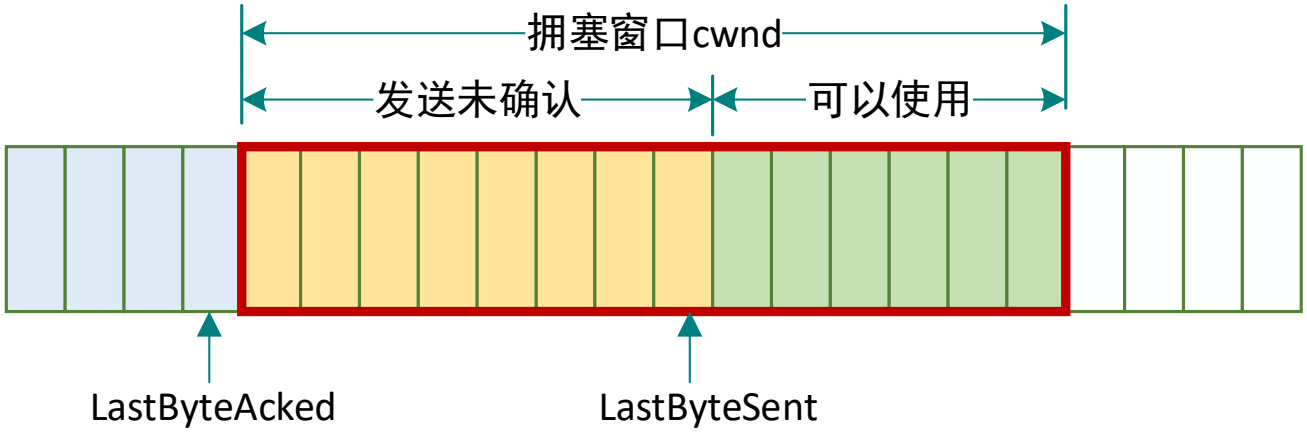


X 丢失，降低发送速率

TCP的拥塞控制的锯齿现象

问题：以什么速度增长和降低发送速率？

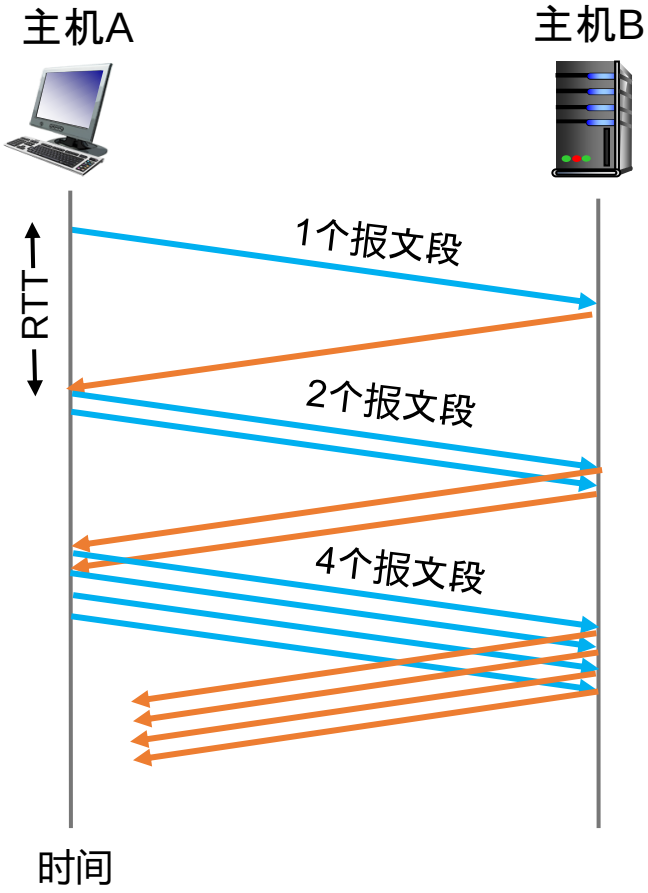
- TCP拥塞控制控制窗口
  - 采用**基于窗口的方法**，通过拥塞窗口的增大或减小控制发送速率



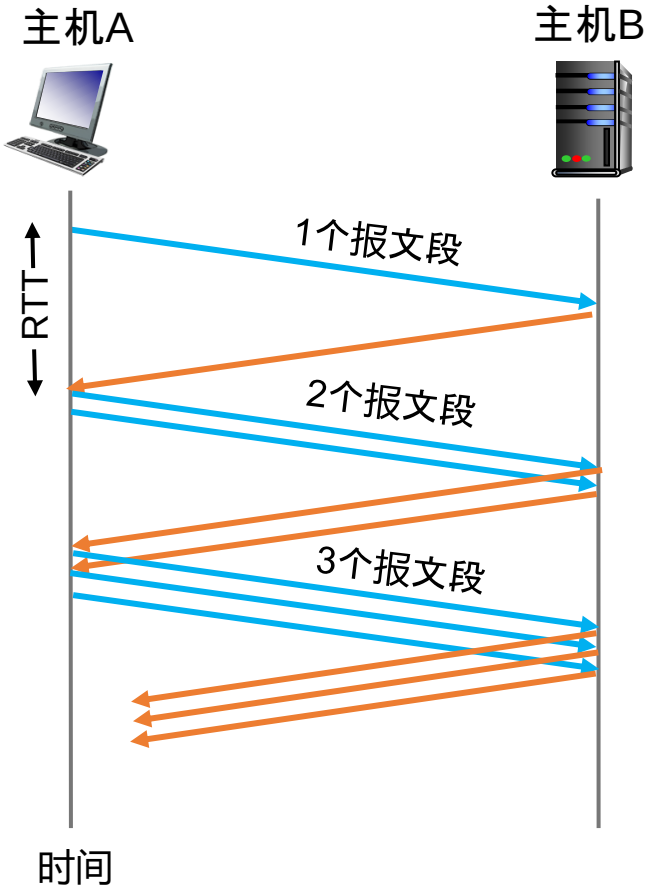
$$LastByteSent - LastByteAcked \leq CongestionWindow\ (cwnd)$$

实际发送窗口取决于接收通告窗口和拥塞控制窗口中较小值

- TCP拥塞控制：慢启动阶段
  - 初始拥塞窗口：cwnd=1(MSS)
  - 每个RTT，cwnd翻倍（指数增长）
  - 每接收到一个ACK，cwnd增1(MSS)
  - 当**连接初始建立**或**报文段超时未得到确认时**，TCP拥塞控制进入慢启动阶段
  - **特点**：初始值小，增长速度快



- TCP拥塞控制：拥塞避免阶段
  - **阈值ssthresh**：拥塞窗口达到该阈值时，慢启动阶段结束，进入拥塞避免阶段
  - 每个RTT，cwnd增1（线性增长）



注意：TCP使用字节计数，当收到ACK时，拥塞窗口计算如下：

$$cwnd = cwnd + MSS \cdot \frac{MSS}{cwnd}$$

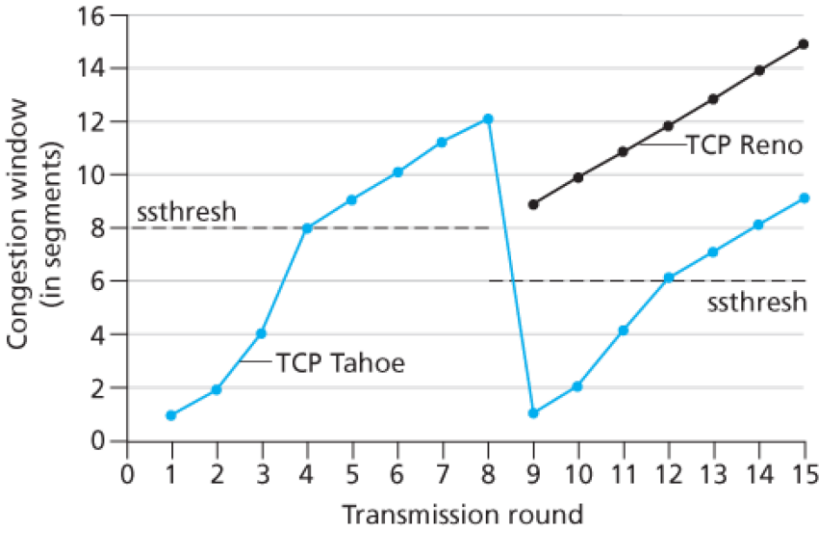
TCP拥塞控制：丢失检测

- 通过超时检测丢失：
  - 阈值ssthresh = cwnd/2
  - cwnd=1(MSS)，进入慢启动阶段
- 通过三次重复ACK检测丢失（**TCP RENO**算法）：
  - 阈值ssthresh = cwnd/2
  - cwnd= ssthresh+3（MSS），进入线性增长（拥塞避免阶段）
  - 注：重复ACK指明网络仍可以交付一些报文段（拥塞不严重）
- **TCP Tahoe**算法对于两种丢失情况均将cwnd设成1（MSS），并进入慢启动阶段

- TCP拥塞控制： **RENO**算法

慢启动算法

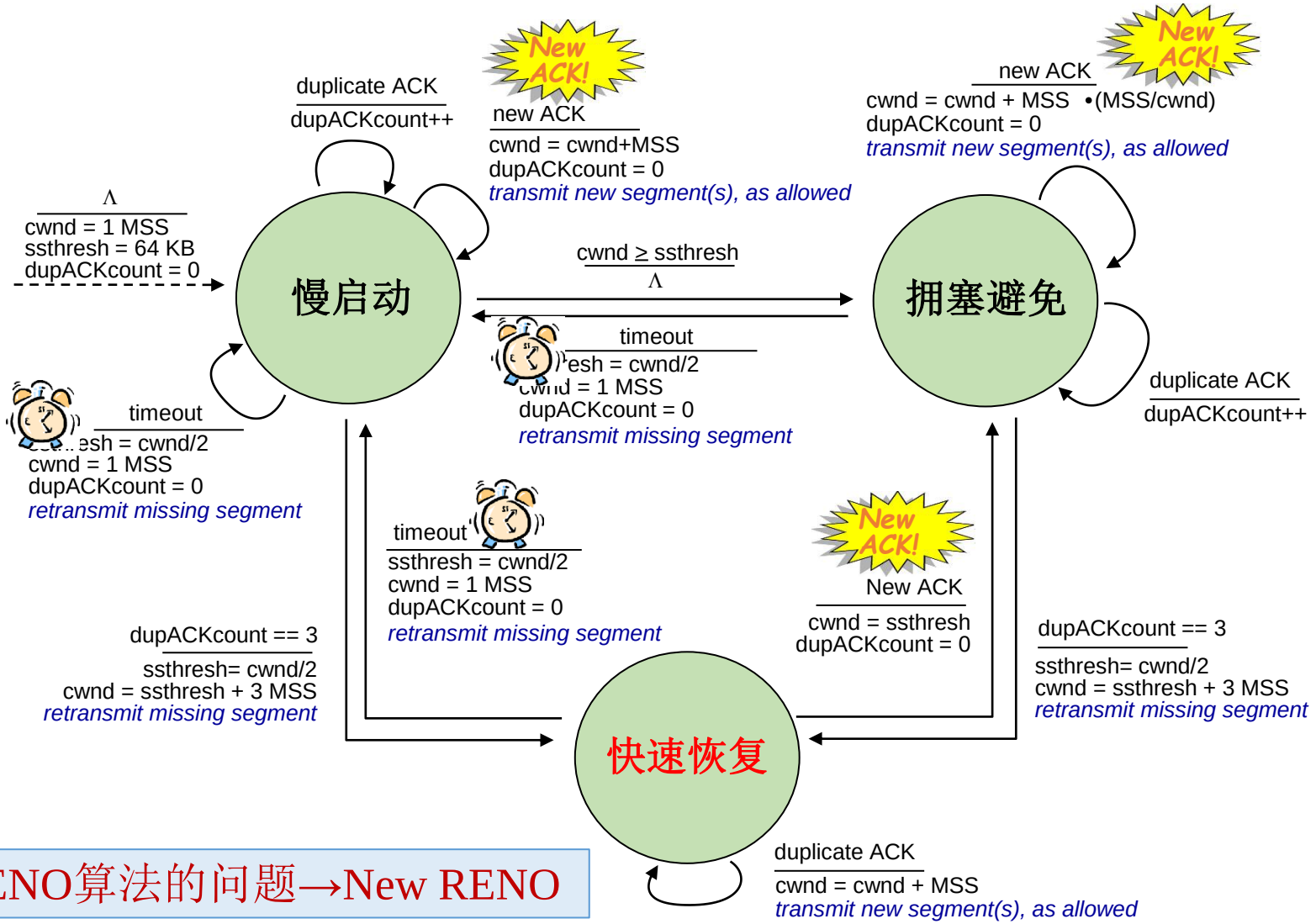
```
initialize: cwnd = 1
for (each segment ACKed)
      cwnd ++
until (loss event OR
      cwnd >= ssthresh)
```



拥塞避免算法

```
/* slowstart is over        */
/* cwnd >= ssthresh      */
Until (loss event) {
  every w segments ACKed:
      cwnd ++
  }
ssthresh = cwnd /2
If (loss detected by timeout) {
    cwnd = 1
    perform slowstart }
If (loss detected by triple
              duplicate ACK)
    cwnd = ssthresh + 3
```

- TCP拥塞控制： **RENO**算法状态机



RENO算法的问题→New RENO

单选题 5分

设置

主机A和主机B之间新建一条TCP连接，主机A的拥塞控制初始阈值为32KB，主机A向主机B发送的每个TCP段中包含1KB数据，并一直有数据发送；主机B为该连接分配16KB接收缓存，并对每个TCP段进行确认。若主机B收到的数据全部存入缓存，且应用进程未读取，发送端未出现超时事件，则主机A从连接建立成功时刻起，经过4个RTT后，其发送窗口为多少？

A 16

B 9

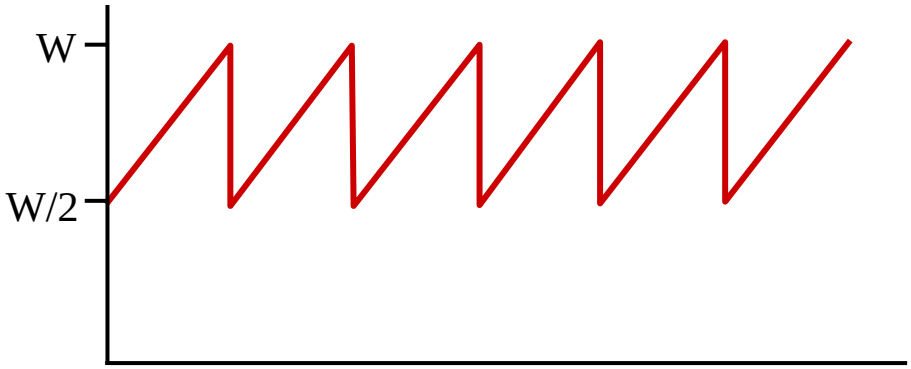
C 8

D 1

提交

- TCP拥塞控制：吞吐率问题讨论
  - TCP连接的吞吐率与拥塞窗口大小和RTT相关，我们假设：
    - 忽略慢启动阶段
    - 总有数据要发送
    - 设W为丢失事件发生时拥塞窗口的大小，W和RTT不变
    - 一个连接的平均吞吐率

$$\text{一个连接的平均吞吐量} = \frac{0.75 \cdot W}{RTT}$$

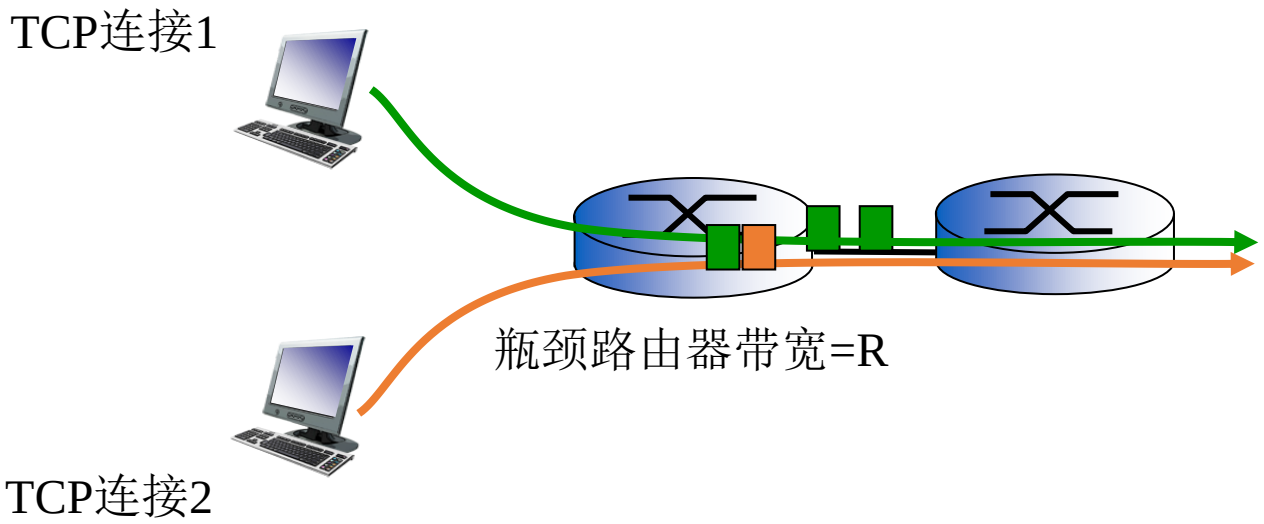


- TCP拥塞控制：吞吐率问题讨论
  - TCP**长肥管道**问题
    - 例如：报文段长度为1500字节，**RTT为100毫秒**，要获得**10Gbps**的吞吐率，要求**平均窗口**长度= **83333报文段**
    - 假设丢包率为L，则

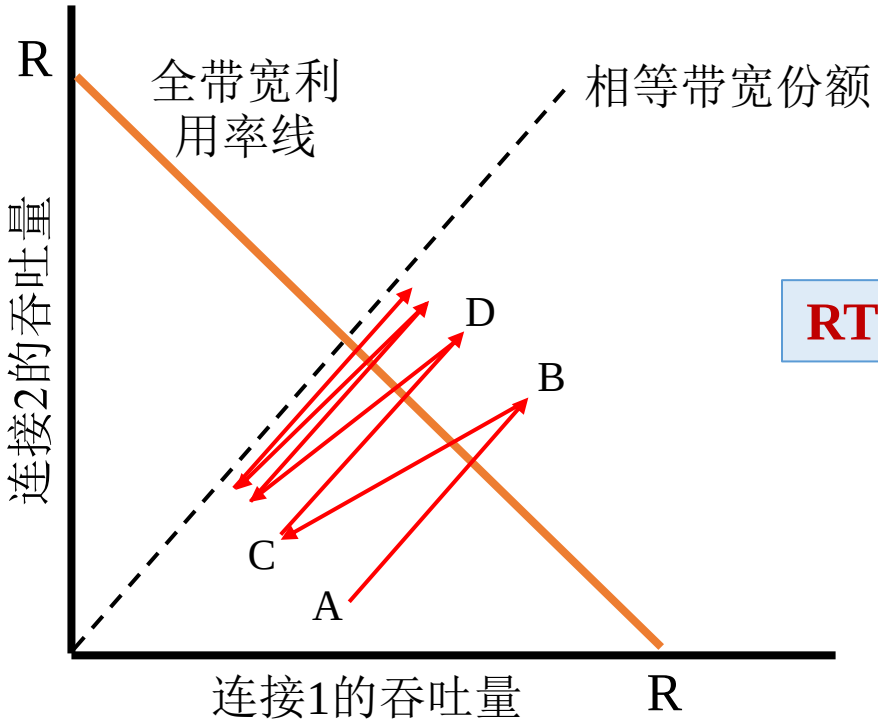
$$\text{一个连接的平均吞吐量} \approx \frac{1.22 \cdot MSS}{RTT \cdot \sqrt{L}}$$

    - 要获得10 Gbps的吞吐量，则 $L \approx 2 \times 10^{-10}$ （非常小的丢失率，很难达到）
    - 对高速网络需要对TCP进行优化（**RFC 3649**）

- TCP拥塞控制：公平性问题讨论
  - **公平性的目标**：如果有*K*个TCP连接共享带宽为*R*的瓶颈链路，每个连接应获得*R*/*K*的平均速率



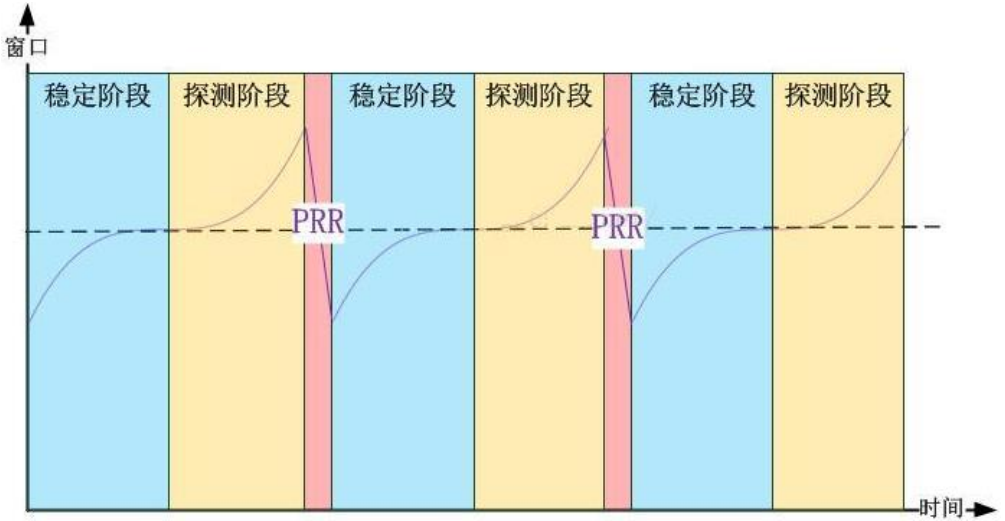
- TCP拥塞控制：公平性问题讨论
  - TCP是否公平？
  - 两个竞争的连接：
    - ✓ 假设两个连接**RTT相等**，从源主机到目的主机只有一条TCP连接

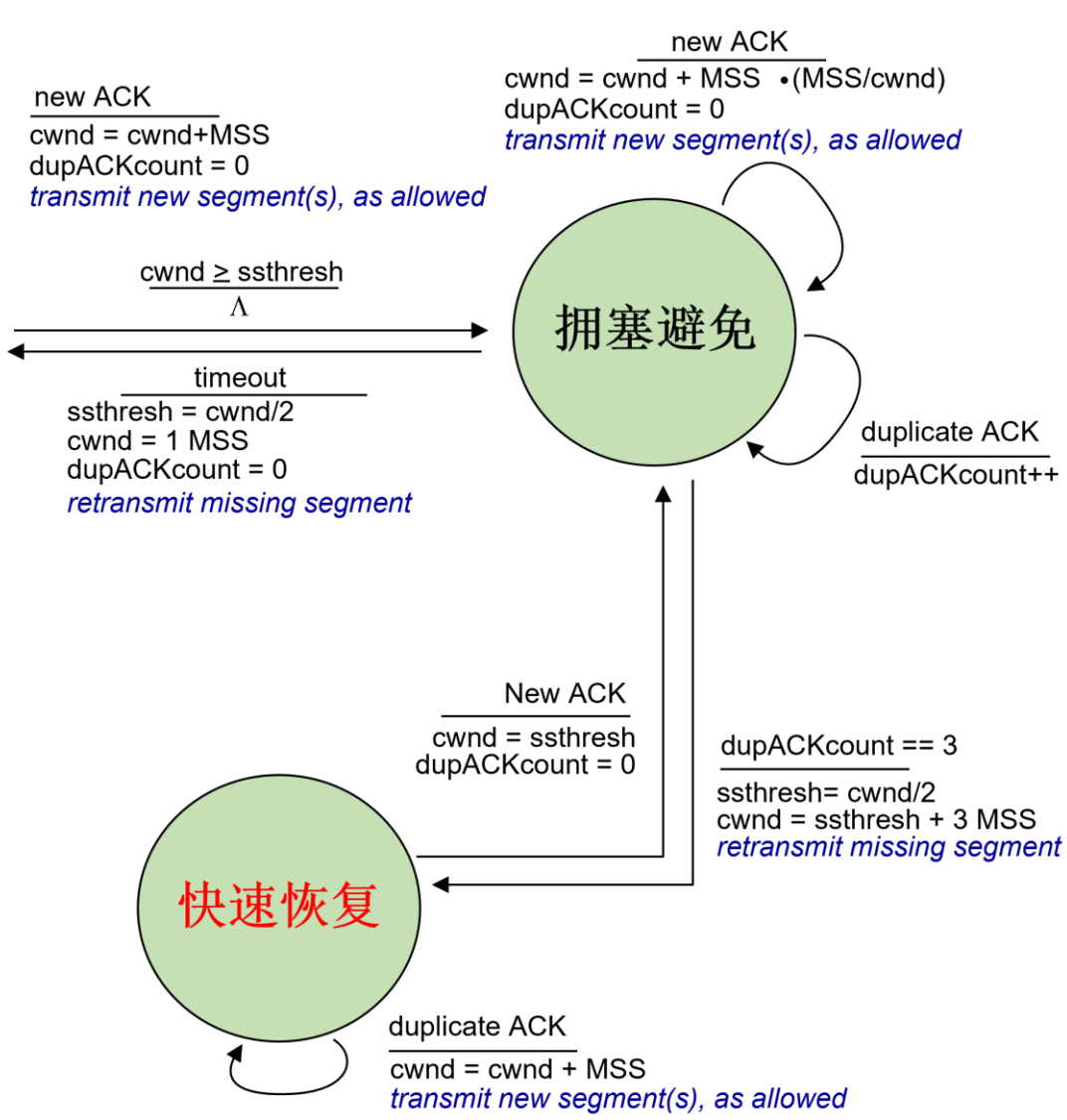


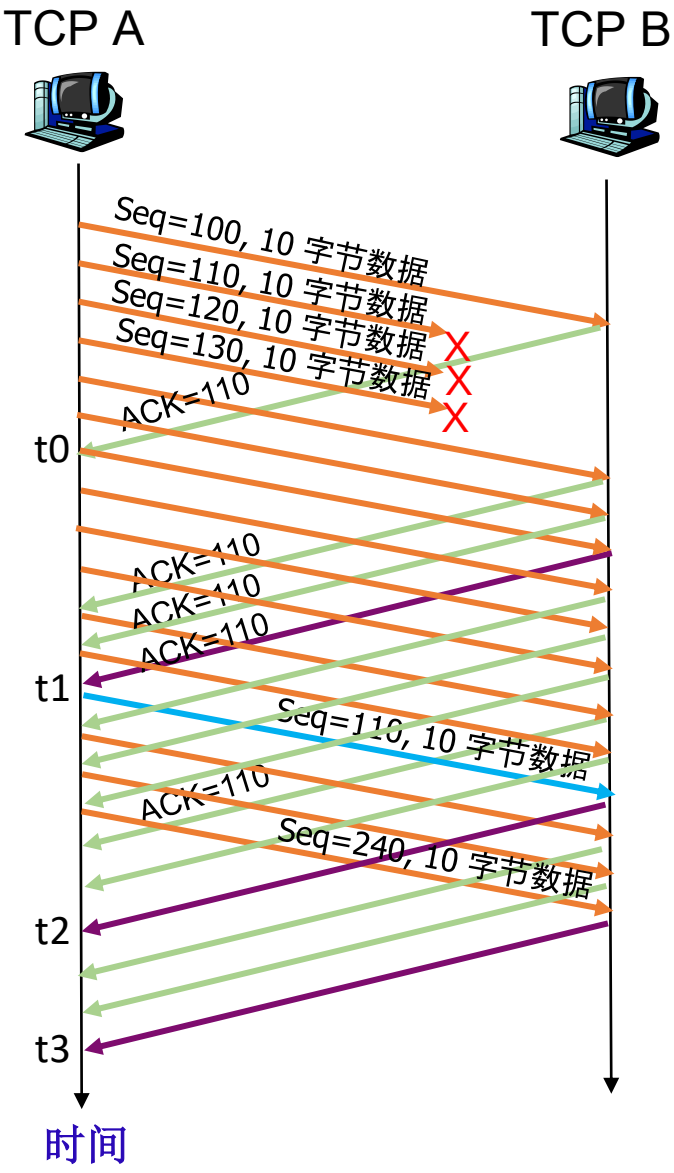
RTT越小拥塞窗口增速越快

- TCP拥塞控制：公平性问题讨论
  - TCP与UDP是否公平？
    - ✓ 有些多媒体应用通常使用UDP，以恒定的速率发送音视频，能容忍一些丢失，避免TCP拥塞控制带来的速率限制及抖动
    - ✓ **UDP流量会压制TCP流量！**（采用哪些方法解决？）
  - 多TCP连接的公平性问题
    - ✓ 一些应用可以建立多个并行的TCP连接，以提高传输速率及解决TCP队头阻塞问题（例如目前的Web应用）
    - ✓ 例如：一个带宽为R的链路由9个存在的应用共享，每个应用使用一个TCP连接；如果一个新的应用也使用1个TCP连接，则每个应用获得相同的R/10带宽；如果新的应用使用11个TCP连接，则会获得R/2的带宽
    - ✓ **多TCP连接存在公平性问题！**

- TCP拥塞控制：当前实现的有代表性的几个算法
  - New RENO算法
    - ✓ RENO仅考虑了每次拥塞发生时只丢失一个报文段的情形
    - ✓ New RENO主要解决**一次拥塞多个报文段丢失**，而造成的拥塞窗口和阈值多次折半问题（使TCP吞吐率降低）
  - Cubic算法
    - ✓ Linux内核2.6之后的默认TCP拥塞控制算法
    - ✓ 解决由于**拥塞窗口的变化对RTT的依赖**而造成的不公平性问题




new ACK
cwnd = cwnd+MSS
dupACKcount = 0
transmit new segment(s), as allowed
new ACK
cwnd = cwnd + MSS •(MSS/cwnd)
dupACKcount = 0
transmit new segment(s), as allowed
cwnd ≥ ssthresh
Λ
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
retransmit missing segment
拥塞避免
duplicate ACK
dupACKcount++
New ACK
cwnd = ssthresh
dupACKcount = 0
dupACKcount == 3
ssthresh= cwnd/2
cwnd = ssthresh + 3 MSS
retransmit missing segment
快速恢复
duplicate ACK
cwnd = cwnd + MSS
transmit new segment(s), as allowed


TCP A
TCP B
Seq=100, 10 字节数据
Seq=110, 10 字节数据
Seq=120, 10 字节数据
Seq=130, 10 字节数据
X
X
X
ACK=110
t0
ACK=110
ACK=110
ACK=110
t1
Seq=110, 10 字节数据
ACK=110
Seq=240, 10 字节数据
t2
t3
时间

单选题 5分

设置

此题未设置答案，请点击右侧设置按钮

设TCP连接的MSS为10字节，在t0时刻，发送端处于拥塞避免状态，阈值ssthresh为64MSS，拥塞窗口cwnd为80MSS，则在t3时刻发送端进入何种状态，cwnd和ssthresh的大小分别为（整个过程中没有超时事件发生）

A 拥塞避免，cwnd=48， ssthresh=40

B 快速回复，cwnd=23， ssthresh=20

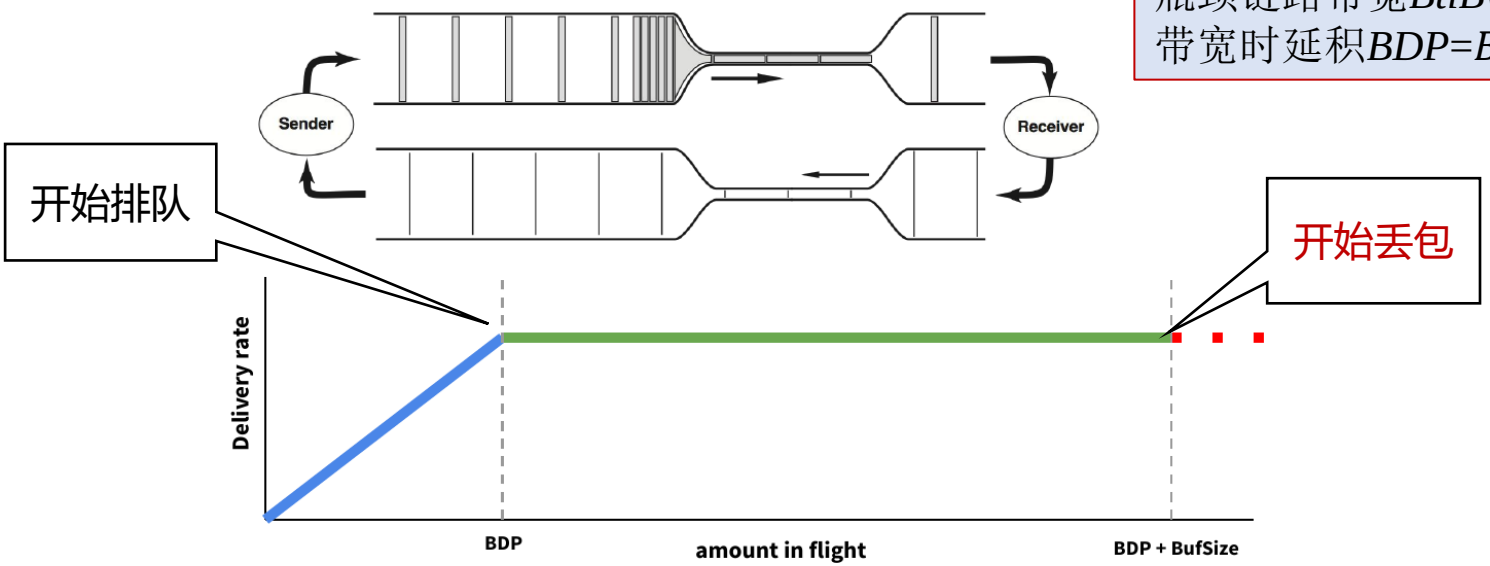
C 拥塞避免，cwnd=23， ssthresh=20

D 快速回复，cwnd=48， ssthresh=40

提交

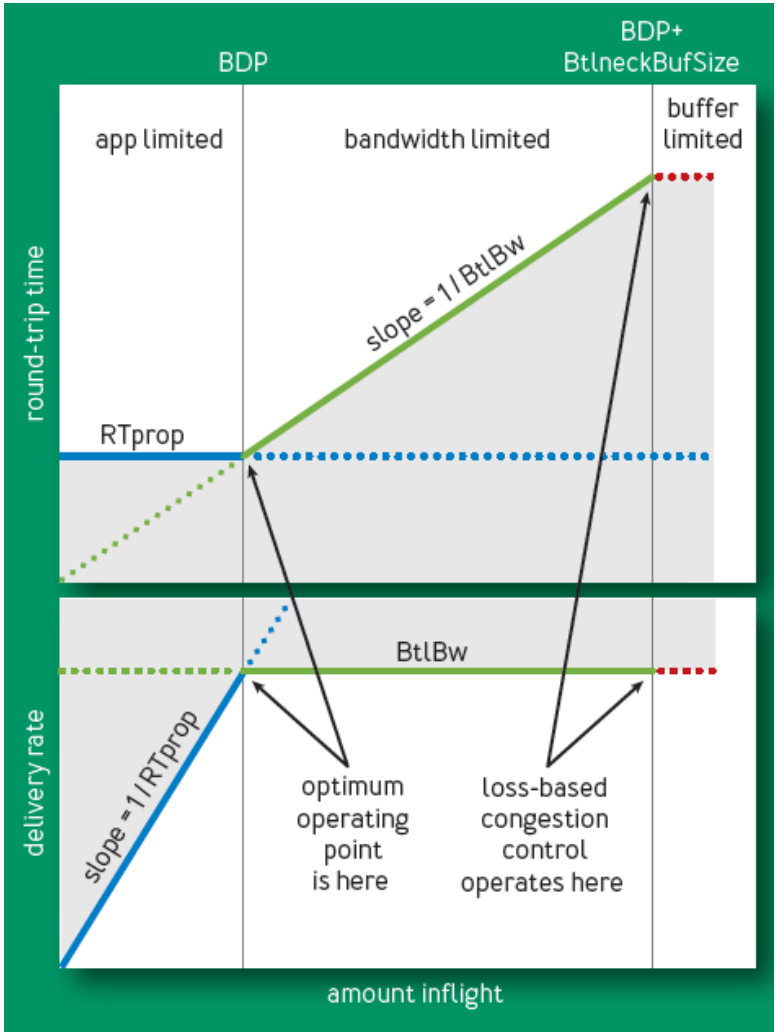
- BBR算法（**B**ottleneck **B**andwidth and **R**ound-trip propagation time ）
  - 2016年由谷歌设计提出，目前已集成到Linux 4.9内核中
  - BBR 算法不将出现丢包或时延增加作为拥塞信号，主动探测网络带宽

瓶颈链路带宽$BtlBw$
带宽时延积$BDP=BtlBw \times RTprop$

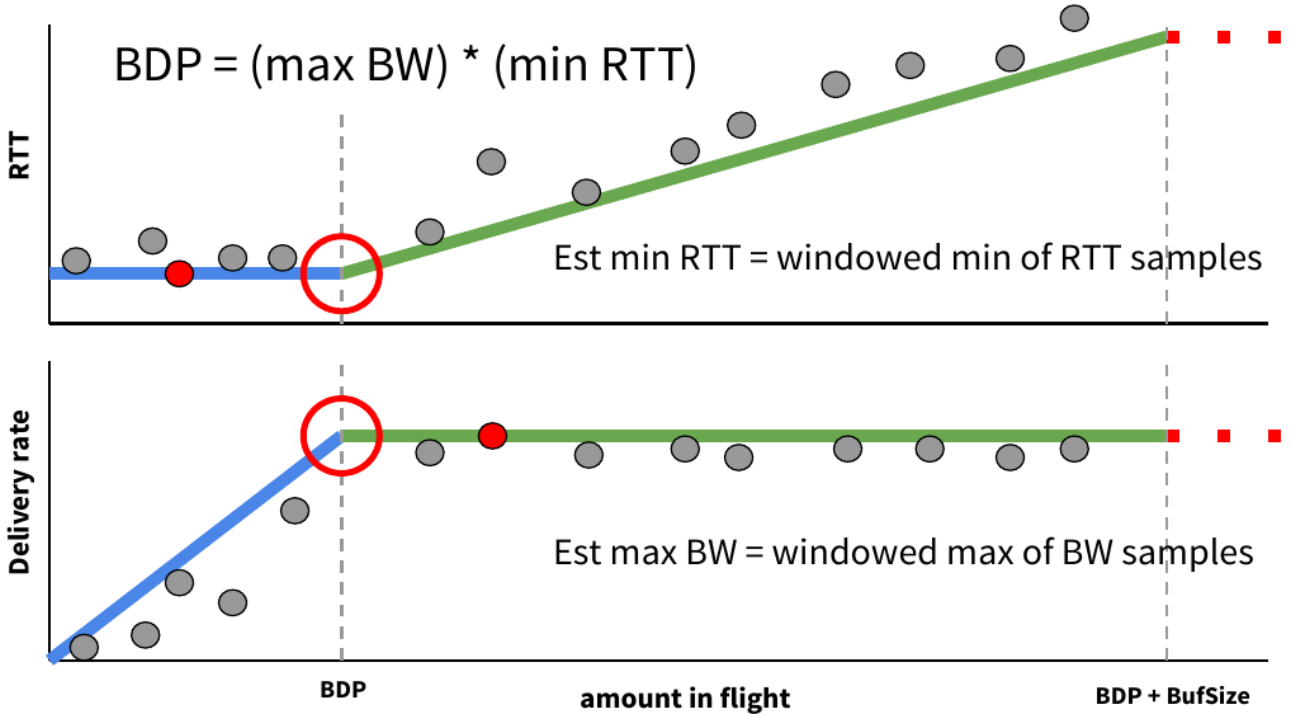


Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. BBR: Congestion-Based Congestion Control. Communications of the ACM. Vol. 60, No. 2, 2017

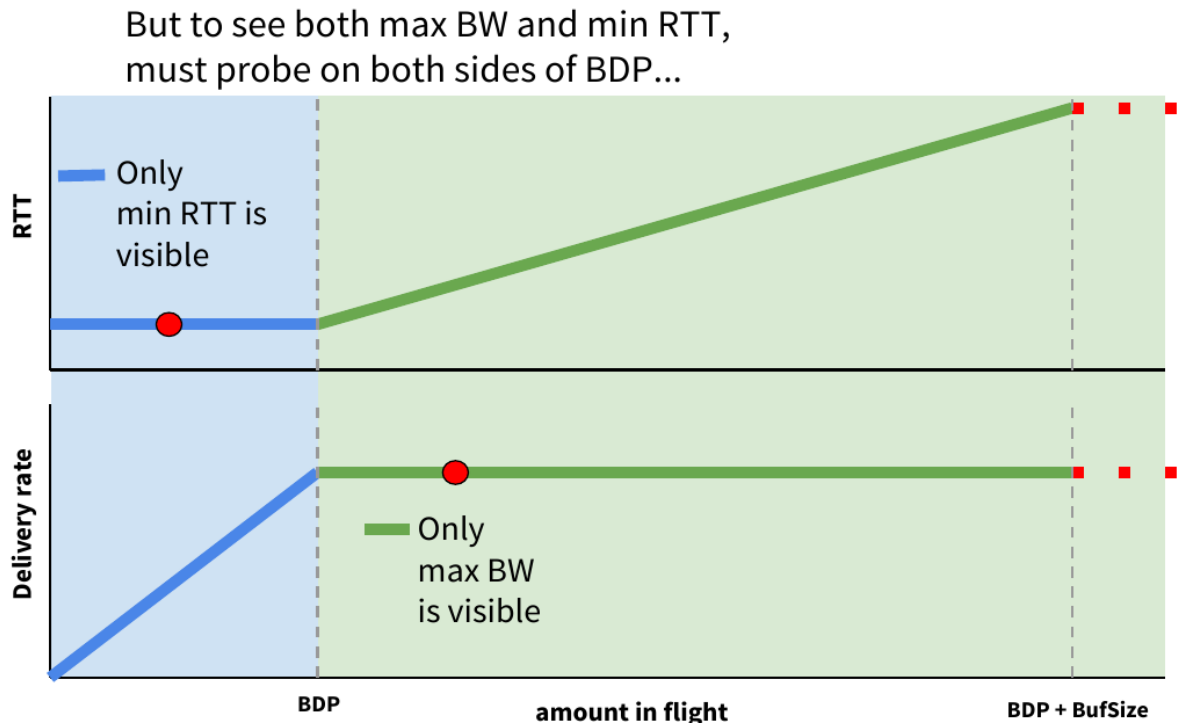
- BBR算法的核心思想
  - **寻找最高吞吐率、最低时延的点**
  - 试图测量图中左侧优化点的BtlBw，尽量将cwnd收敛到实际BDP，从而避免出现丢包
    - ✓ *RTprop*：往返时间
    - ✓ *BtlBw*：瓶颈链路带宽
    - ✓ *BDP*：带宽时延积=$BtlBw \times RTprop$



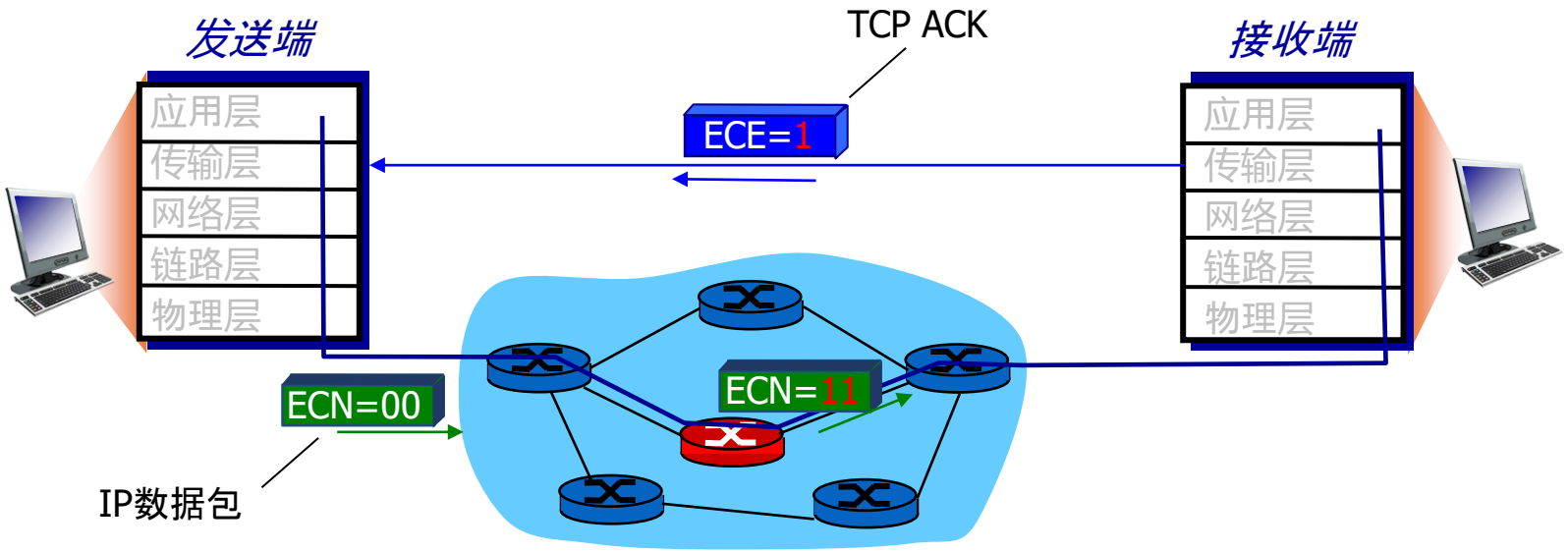
- 优化点的近似观测
  - 用过去10秒内的最小RTT (min RTT) 和最大投递率 (max BW)，分别近似 *Rtprop* 和 *BtlBw*，并依据这两个值估算当前*BDP*



- 优化点的近似观测
  - max BW和min RTT不能同时被测得
    - ✓ 要测最大带宽，需要把瓶颈链路填满，此时缓存中有排队分组，延迟较高
    - ✓ 要测最低延迟，需要保证链路队列为空，网络中分组越少越好，cwnd较小



- TCP拥塞控制：显示拥塞通知（ECN）
  - 一种网络辅助的拥塞控制机制，在RFC3168中定义
  - 路由器使用IP数据包首部TOS域段中的两位指示网络拥塞
  - 采用前向通知：网络拥塞通知携带到接收端
  - 接收端在返回的ACK中设置ECE位（TCP报文段的保留字段的低位），将网络拥塞通知给发送端



总 结

- 传输层基本概念和原理
  - 复用与分用
  - 可靠数据传输
  - 流量控制
  - 拥塞控制
- 两个典型传输层协议
  - UDP
  - TCP

Thank You!

吴英

wuying@nankai.edu.cn