计算机网络

第三章 传输层协议

徐敬东 张建忠

xujd@nankai.edu.cn

zhangjz@nankai.edu.cn

计算机网络与信息安全研究室

计算机学院&网络空间安全学院

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）

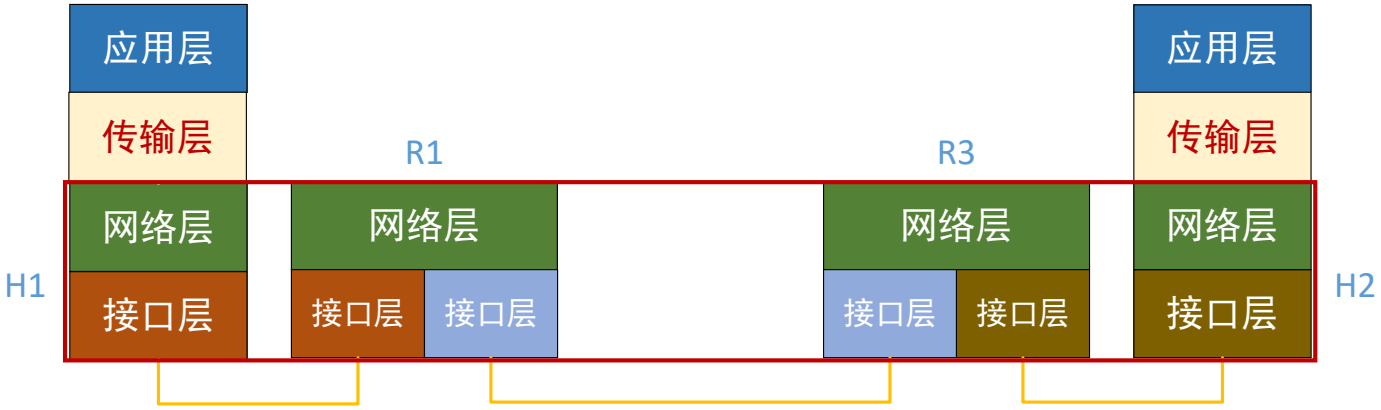
3.4 可靠数据传输

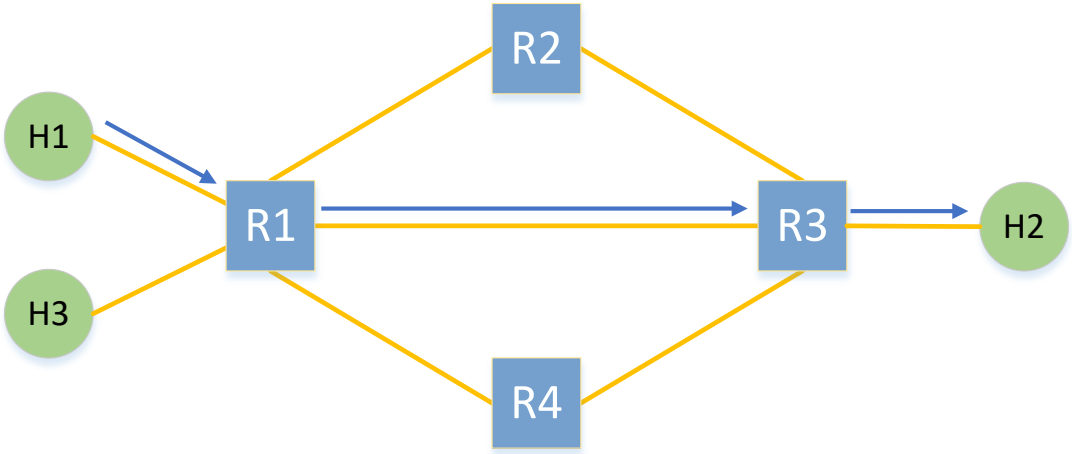
3.5 传输控制协议（TCP）

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

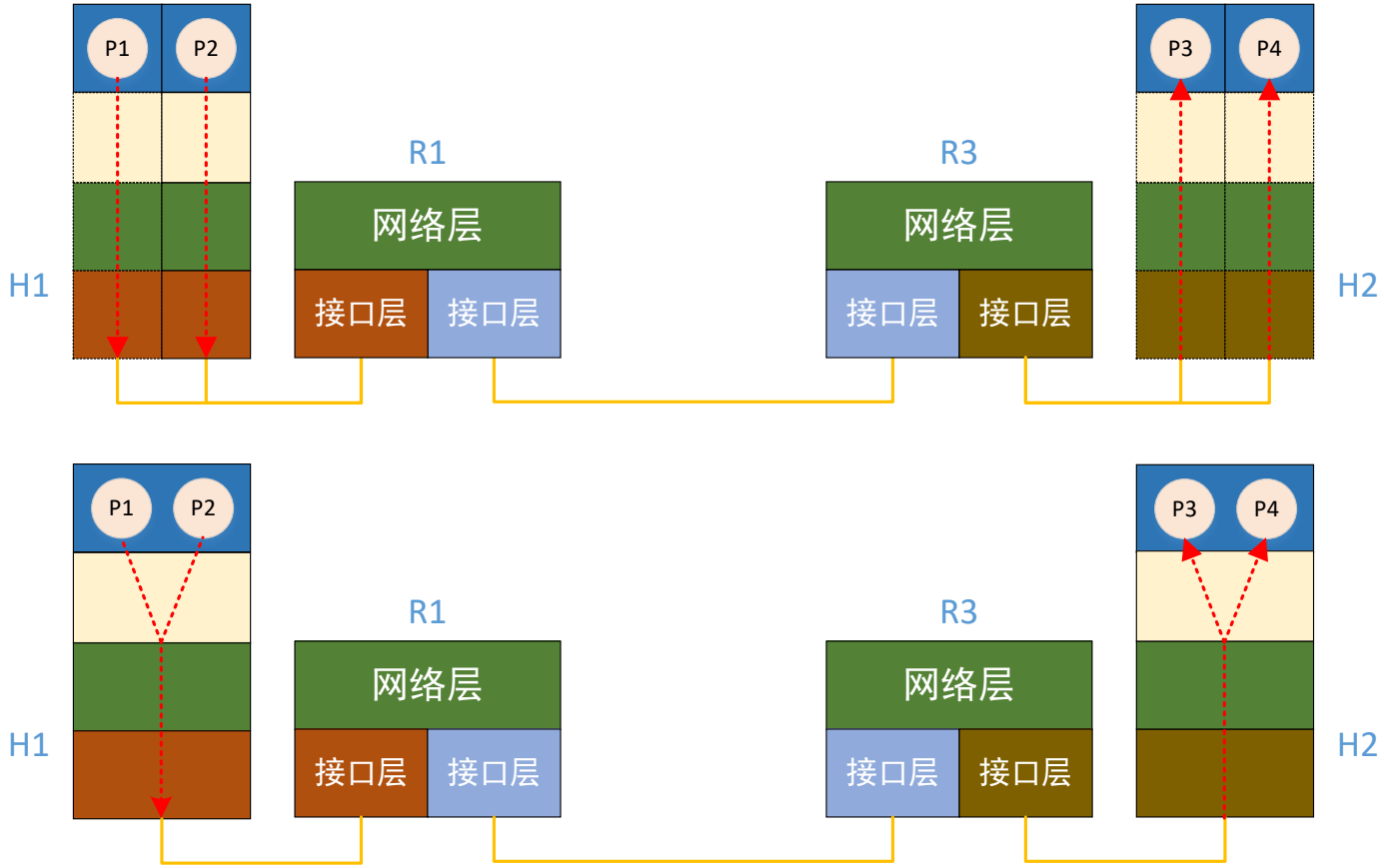




- 网络层：将IP数据包从源主机传送到目的主机，提供无连接不可靠服务
- **数据包传输存在的问题：** 延迟、乱序、出错、丢失等
- **传输层解决的问题之一：** 可靠机制，向应用层提供可靠服务

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

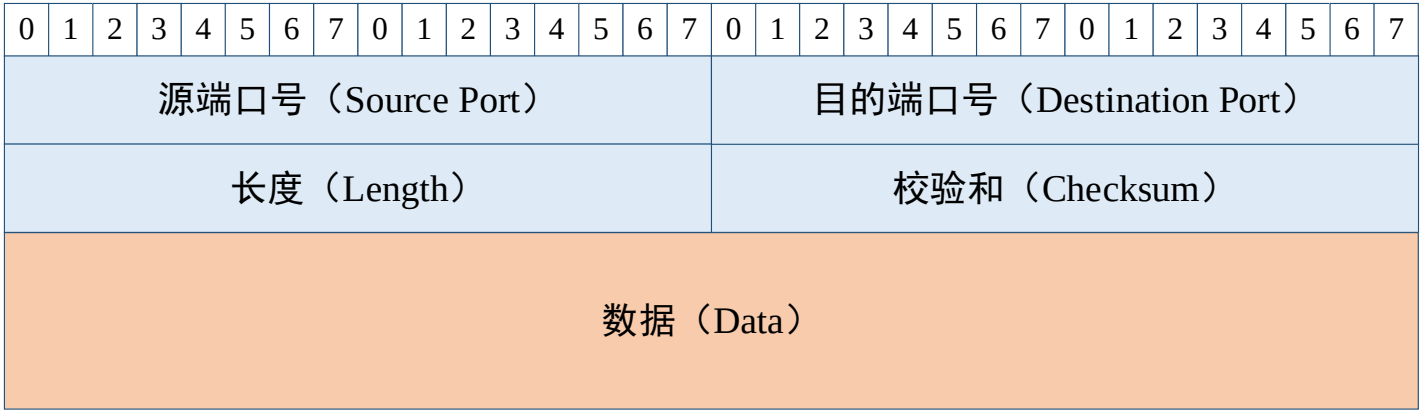
RFC 768

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

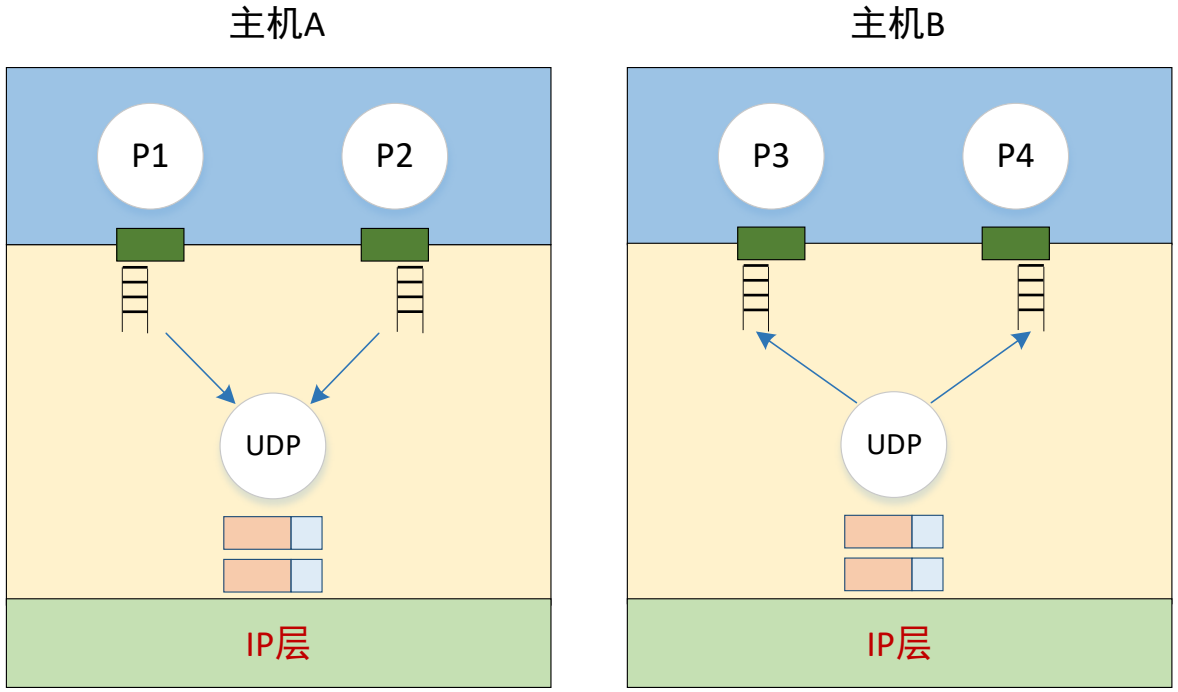
UDP数据报格式

- 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

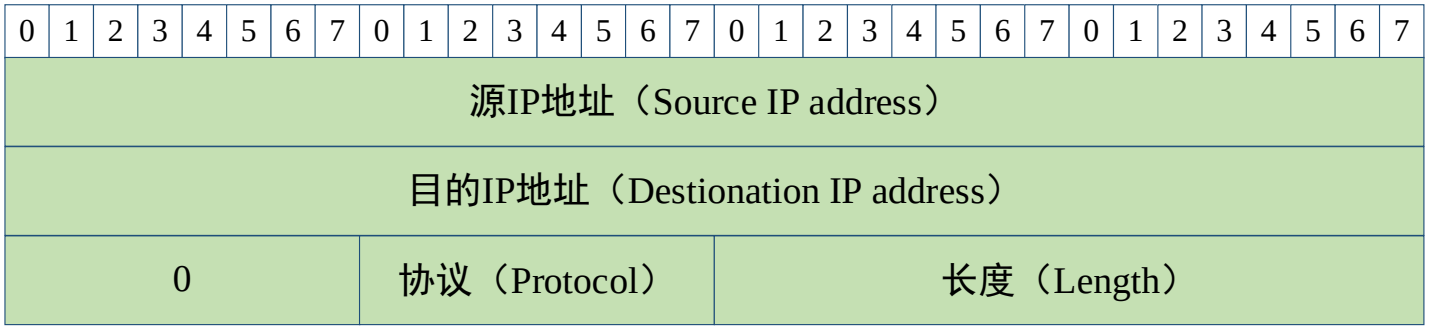


- 进程标识：目的IP地址+目的端口号
- 例如：P1与P3通信，P2与P4通信，P1使用端口6000， P2使用端口7000，P3使用端口6000， P4使用端口8000

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



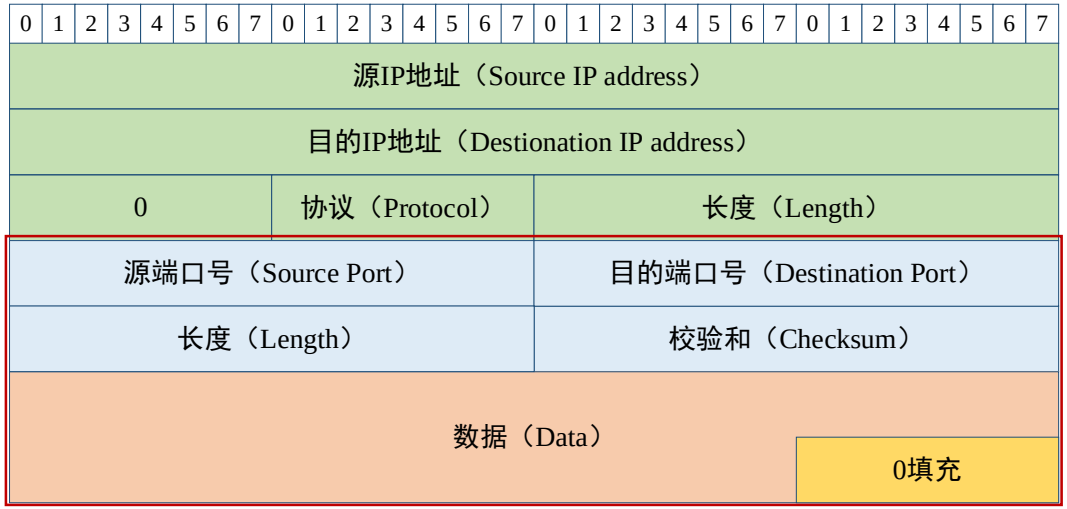
UDP校验和的计算方法

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

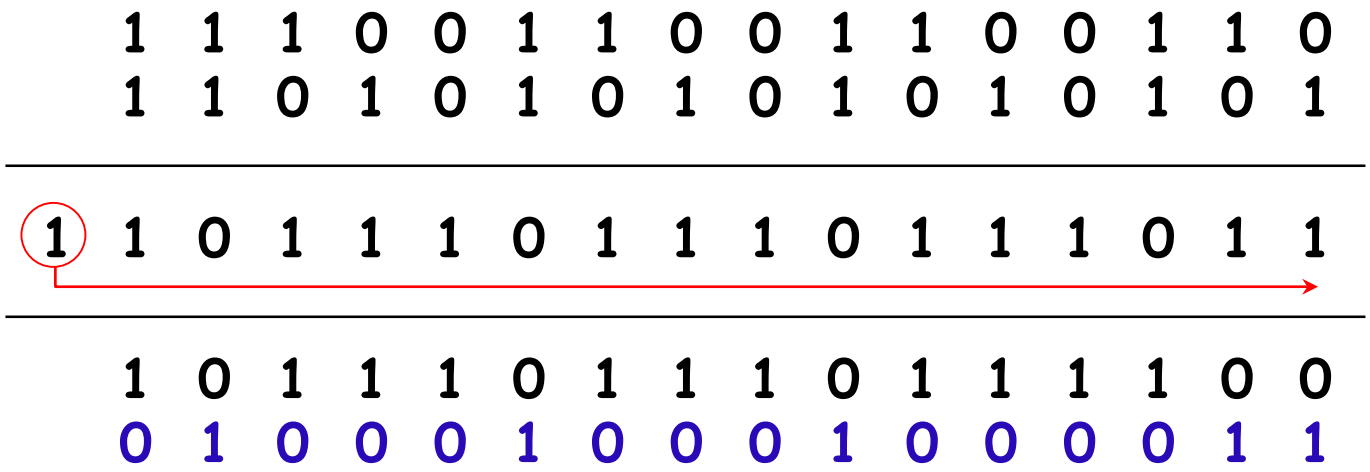
接收端：

- 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



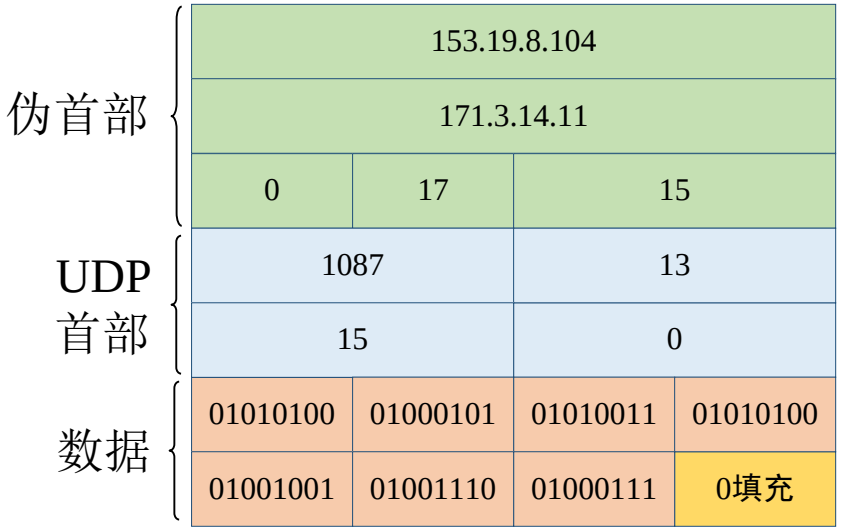
3.3 用户数据报协议UDP

- TCP/IP校验和计算方法示例



3.3 用户数据报协议UDP

计算 UDP校验和示例



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);
  }
```

3.3 用户数据报协议UDP

UDP校验和计算几点说明

- IPv4中UDP校验和是可选项，IPv6中将变成强制性的
  - 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数据报是否真正到达目的地，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

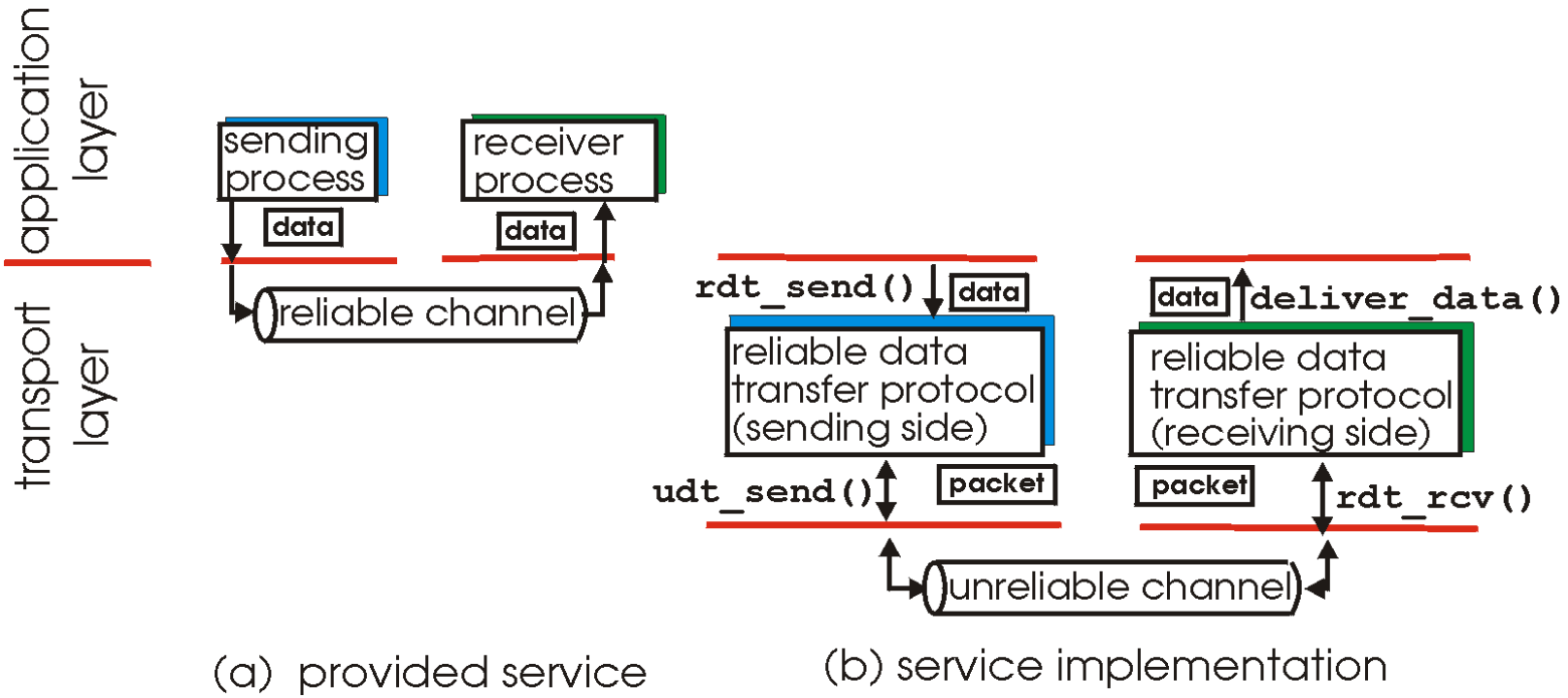
- 流媒体应用（实时音频和视频）通常使用UDP服务
  - 能够容忍一定的丢失
  - 对时延敏感
- 其他使用UDP服务的应用，如：
  - DNS
  - SNMP
- 需要在UDP之上实现可靠传输，即在应用层增加可靠机制

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，特别对与简单的交互应用
- 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

Principles of Reliable data transfer

- important in application, transport, link layers
- top-10 list of important networking topics!

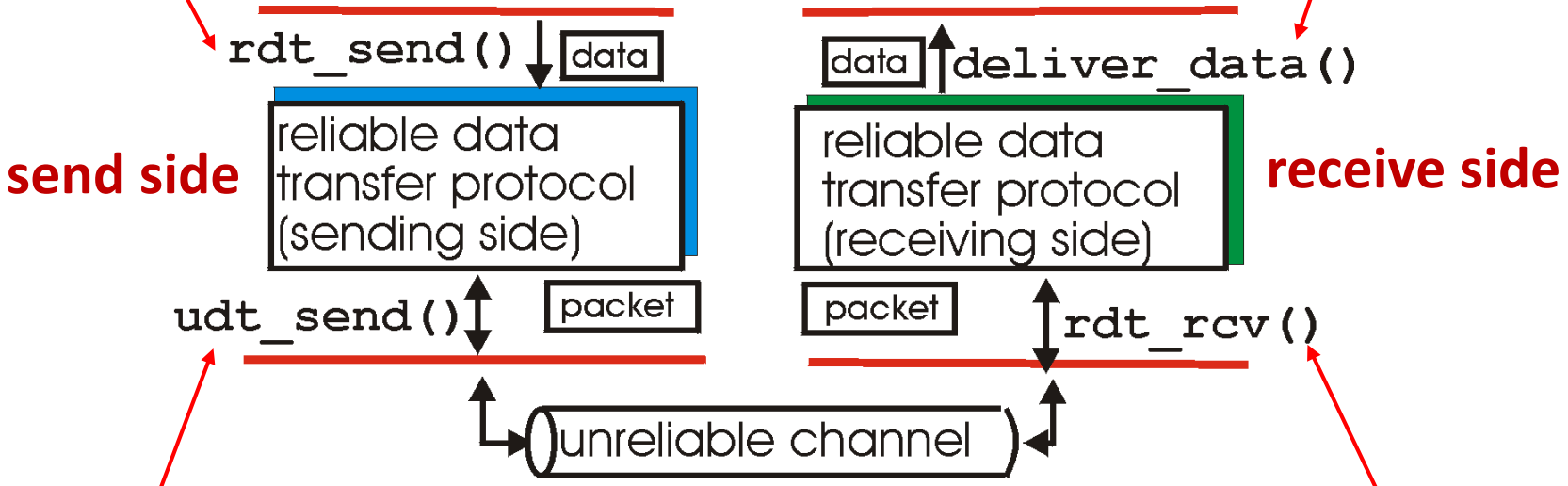


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

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



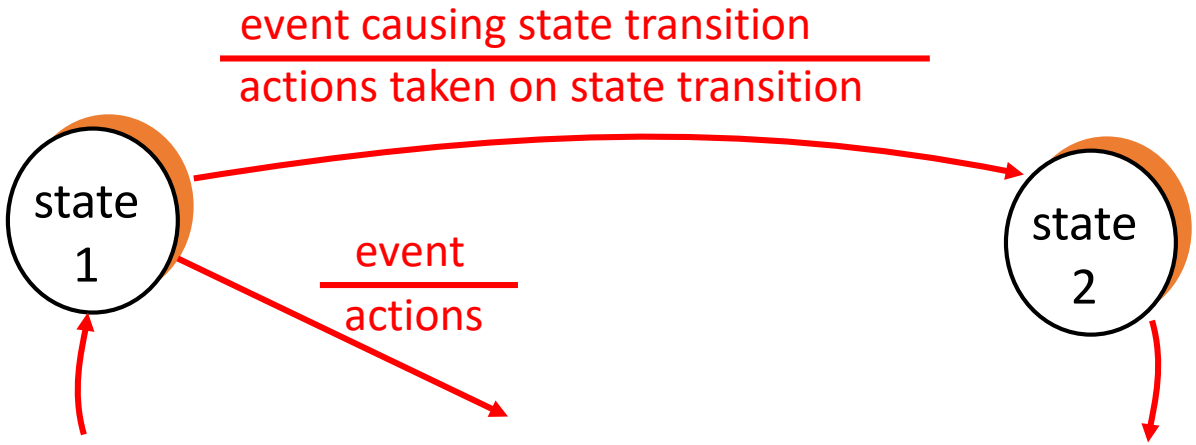
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Reliable data transfer: getting started

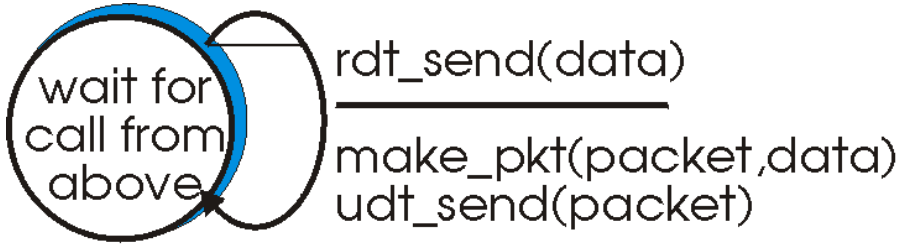
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

state: when in this "state" next state uniquely determined by next event

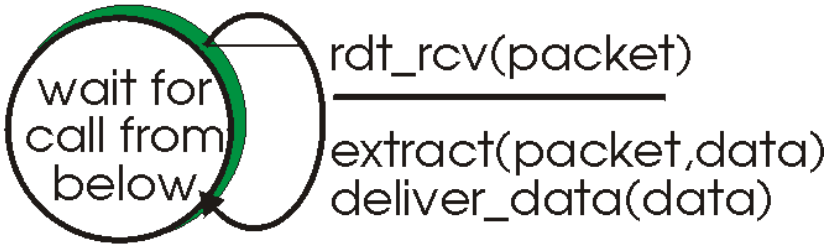


Rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



(a) rdt1.0: sending side

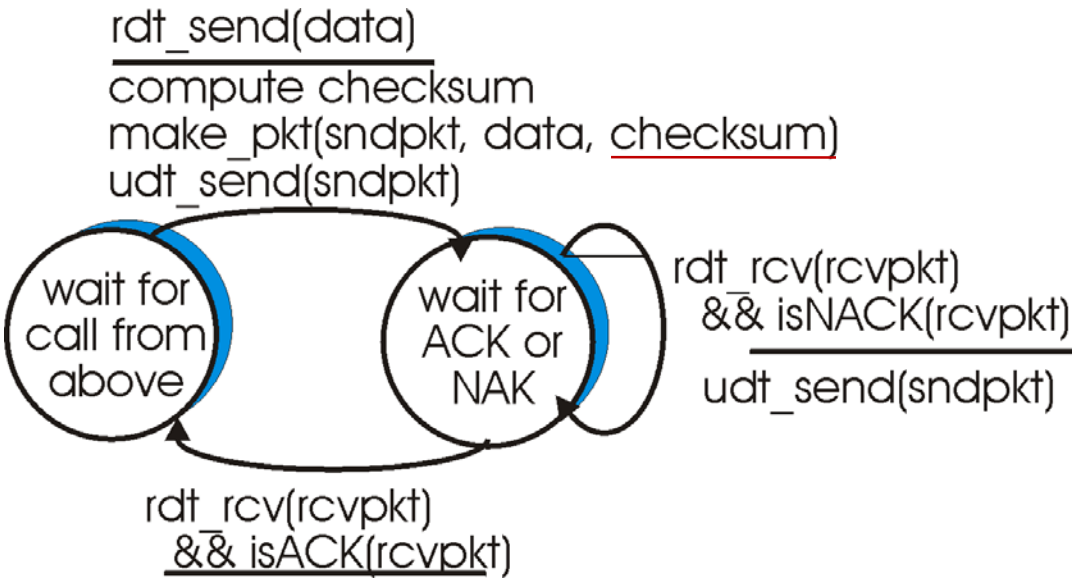


(b) rdt1.0: receiving side

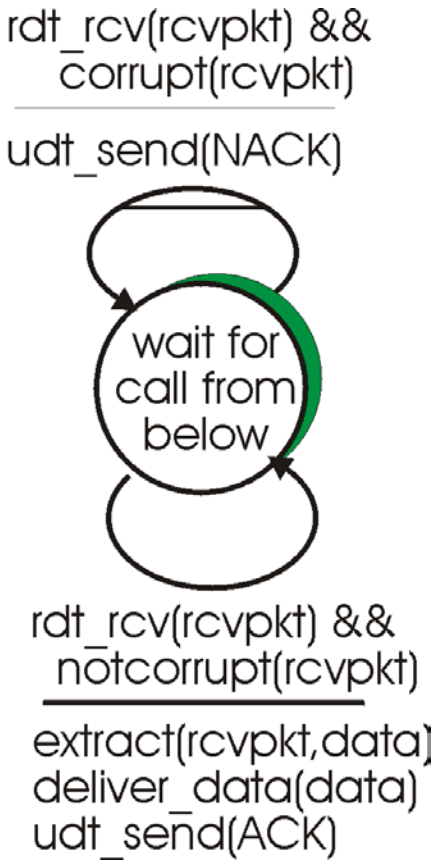
Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - recall: UDP checksum to detect bit errors
- the question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAK or NACKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in **rdt2.0** (beyond **rdt1.0**):
  - error detection
  - receiver feedback: control message (ACK,NAK) rcvr→sender
  - sender retransmits

Rdt2.0: FSM specification

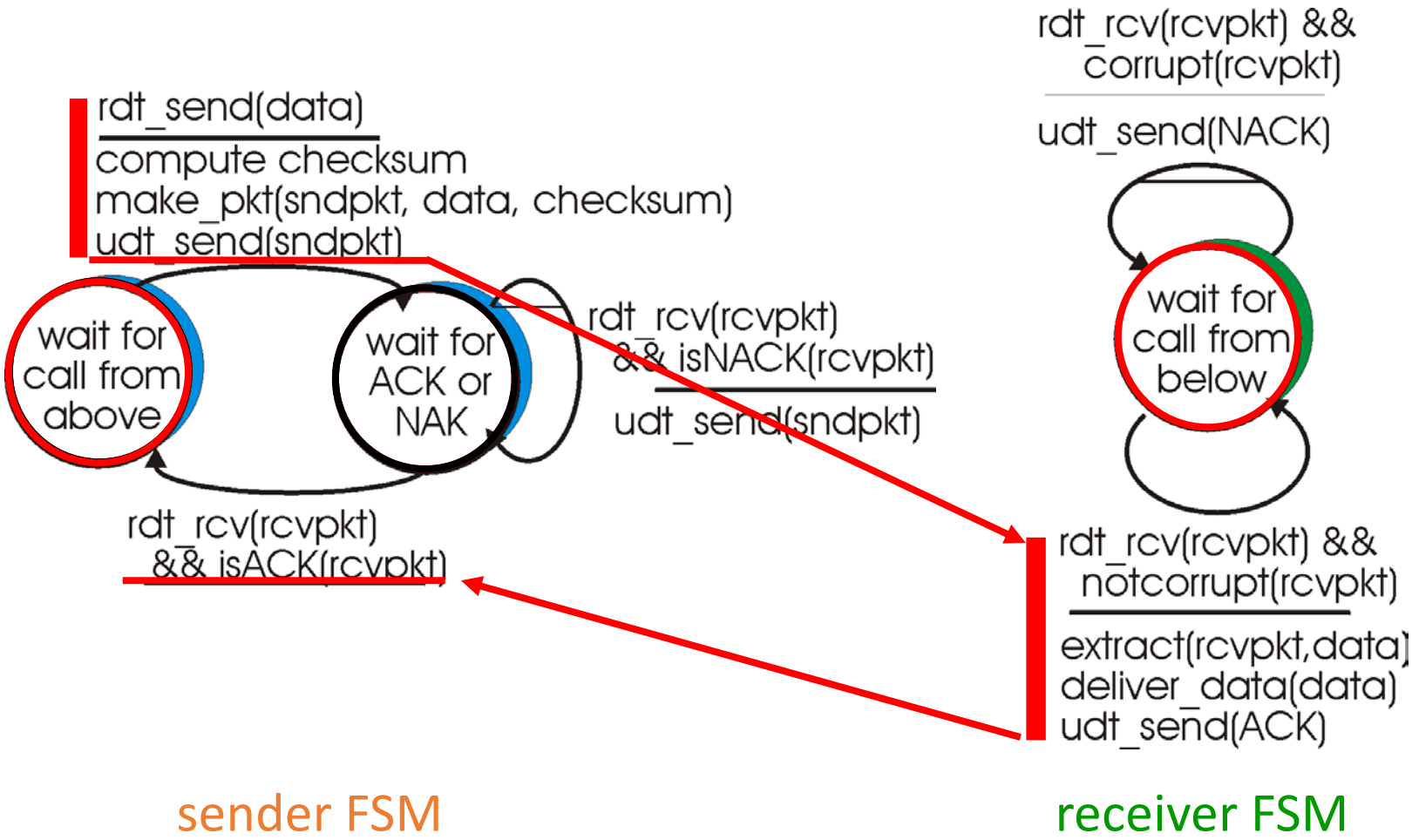


sender FSM

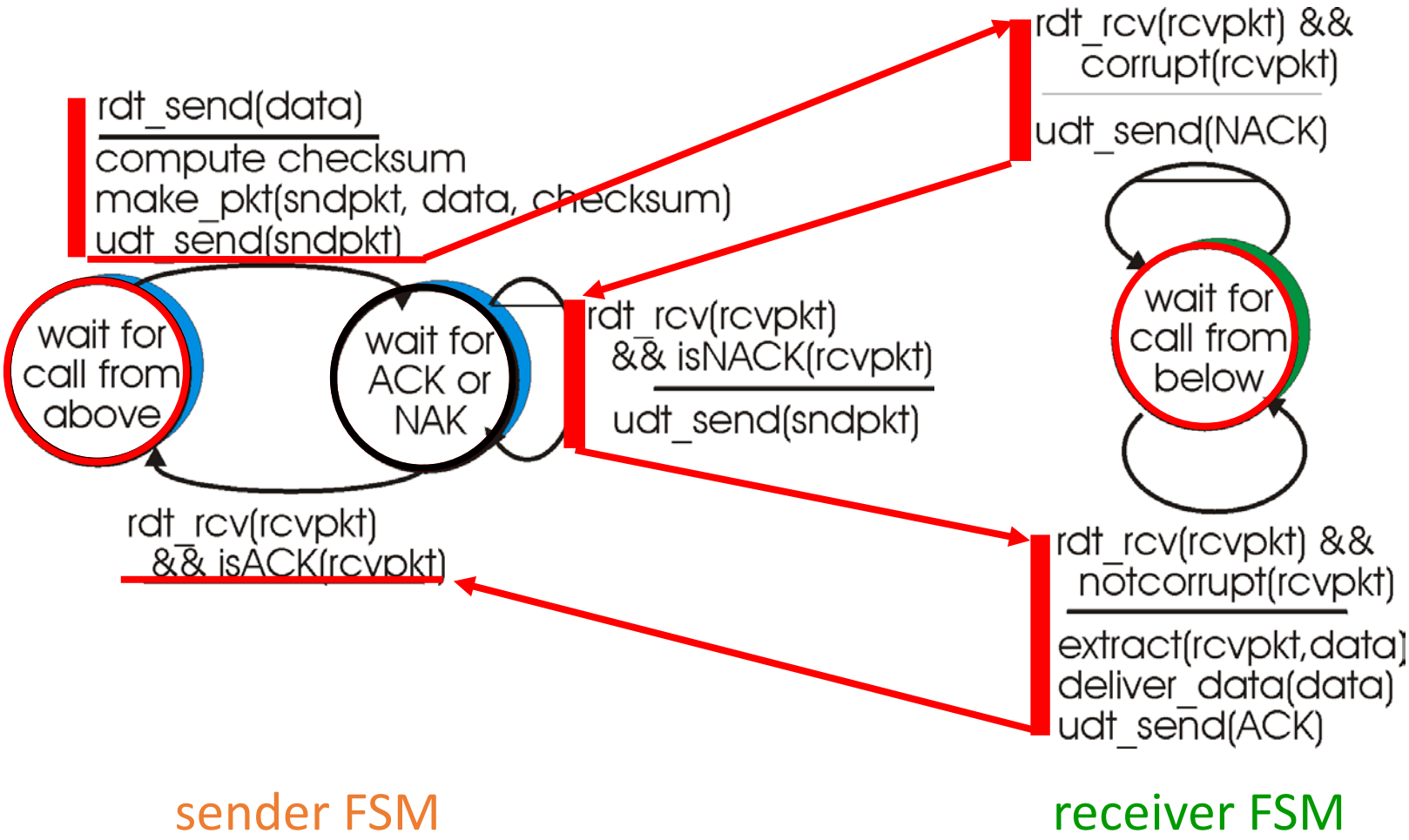


receiver FSM

Rdt2.0: operation with no errors



Rdt2.0: error scenario



Rdt2.0 has a fatal flaw !

What happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

What to do?

- sender ACKs/NAKs receiver's ACK/NAK? What if sender ACK/NAK lost?
- retransmit, but this might cause retransmission of correctly received pkt!

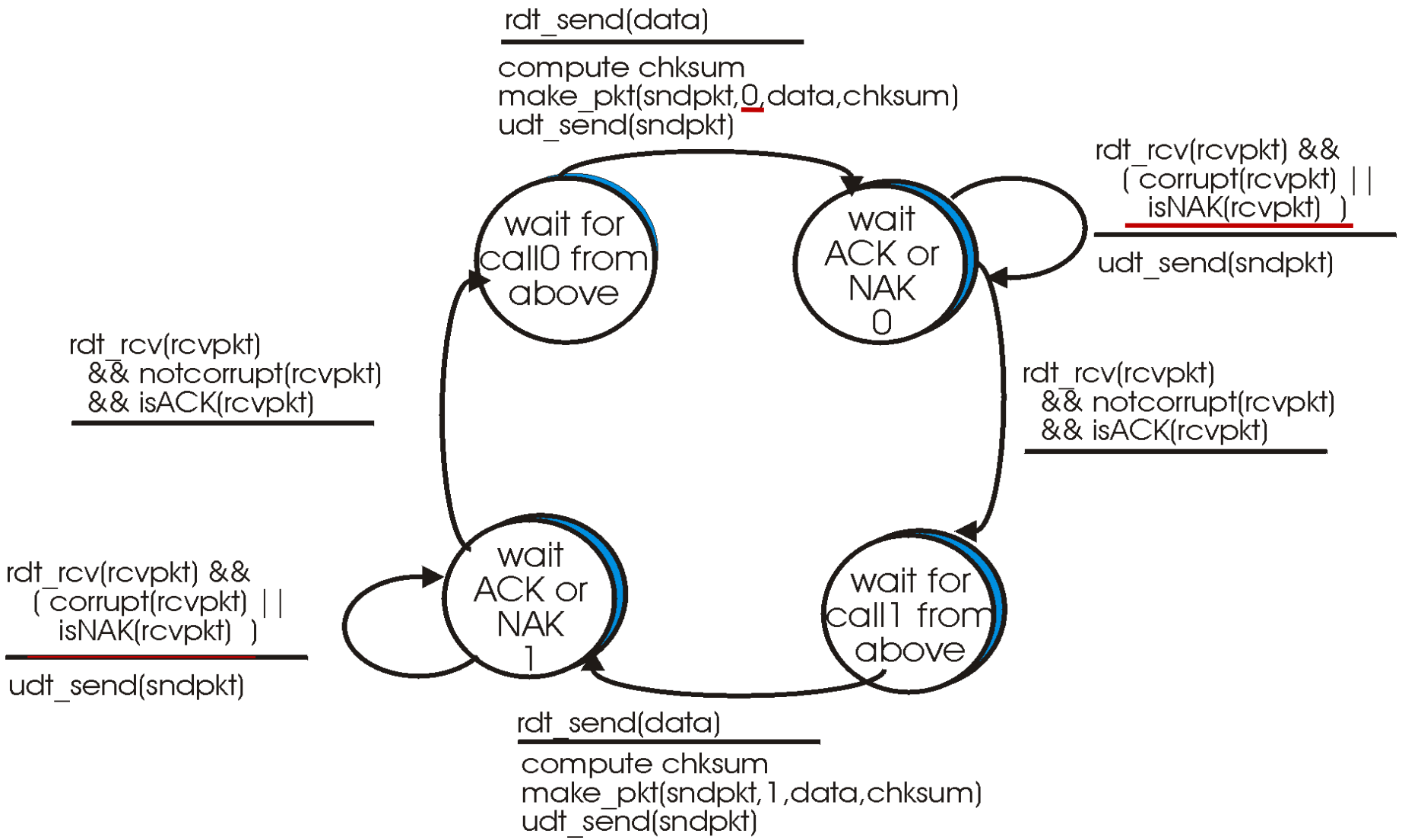
Handling duplicates:

- sender adds *sequence number* to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- receiver discards (doesn't deliver up) duplicate pkt

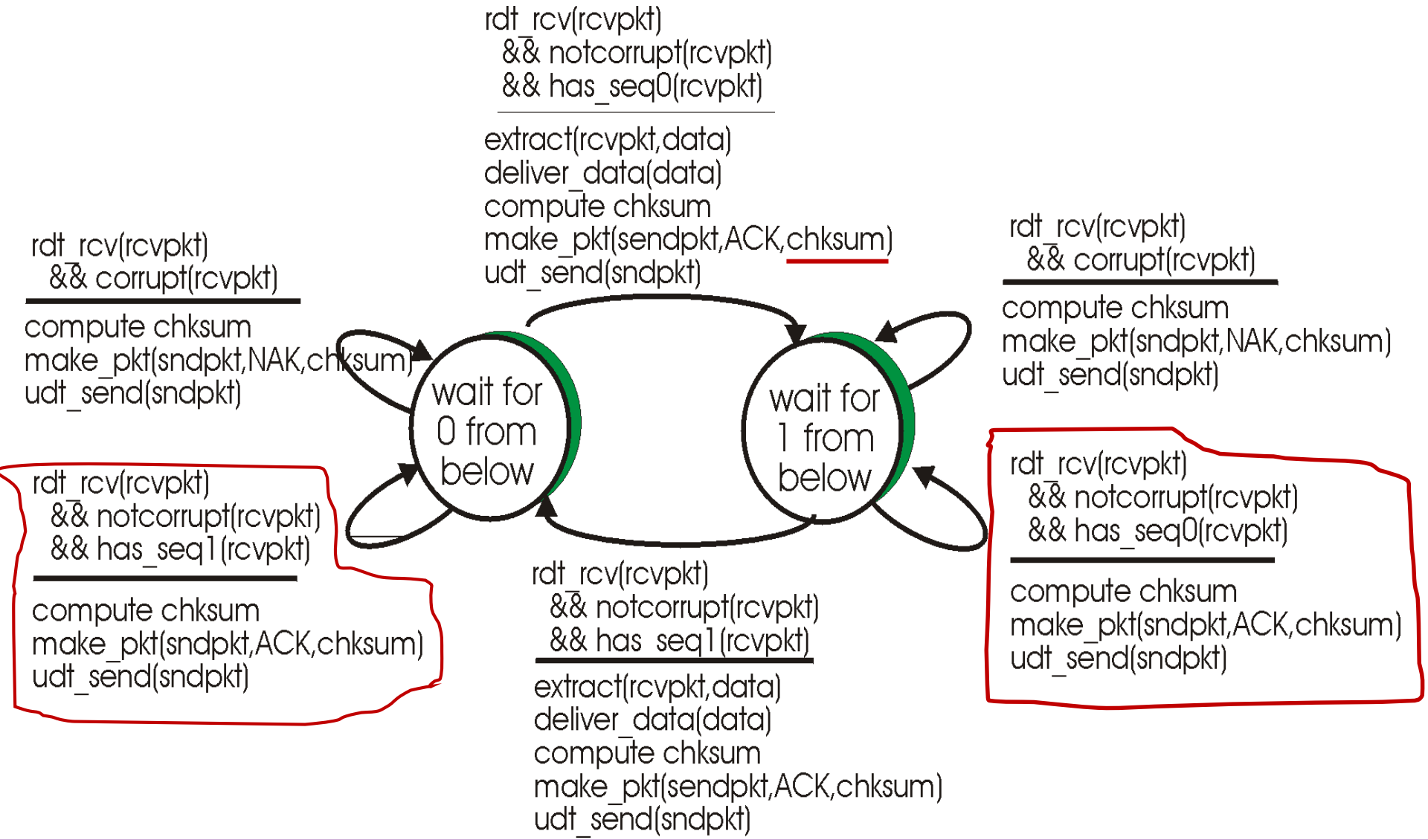
stop and wait

Sender sends one packet, then waits for receiver response

Rdt2.1: sender, handles garbled ACK/NAKs



Rdt2.1: receiver, handles garbled ACK/NAKs



Rdt2.1: discussion

Sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "current" pkt has 0 or 1 seq. #

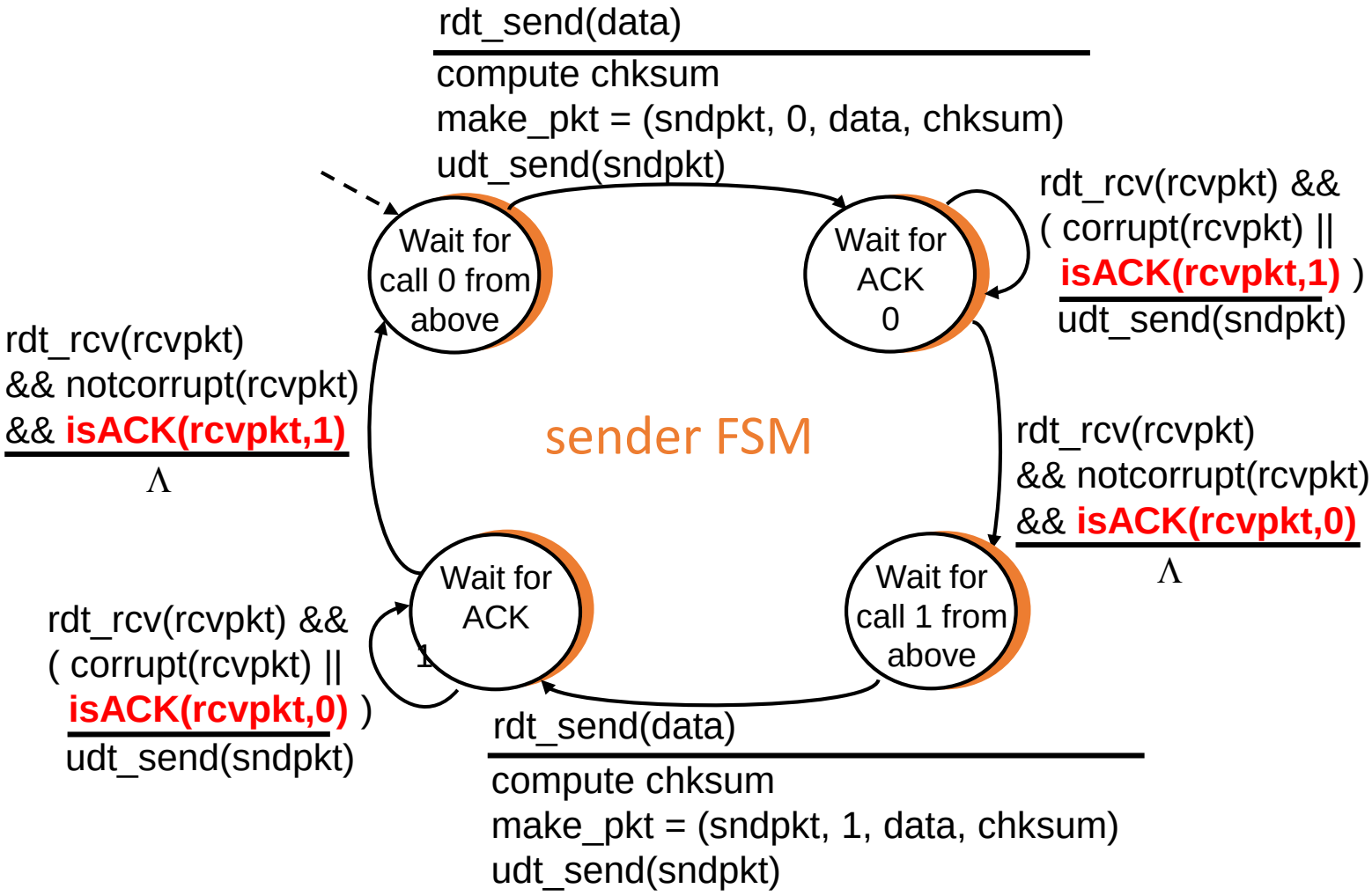
Receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- **note:** receiver can *not* know if its last ACK/NAK received OK at sender

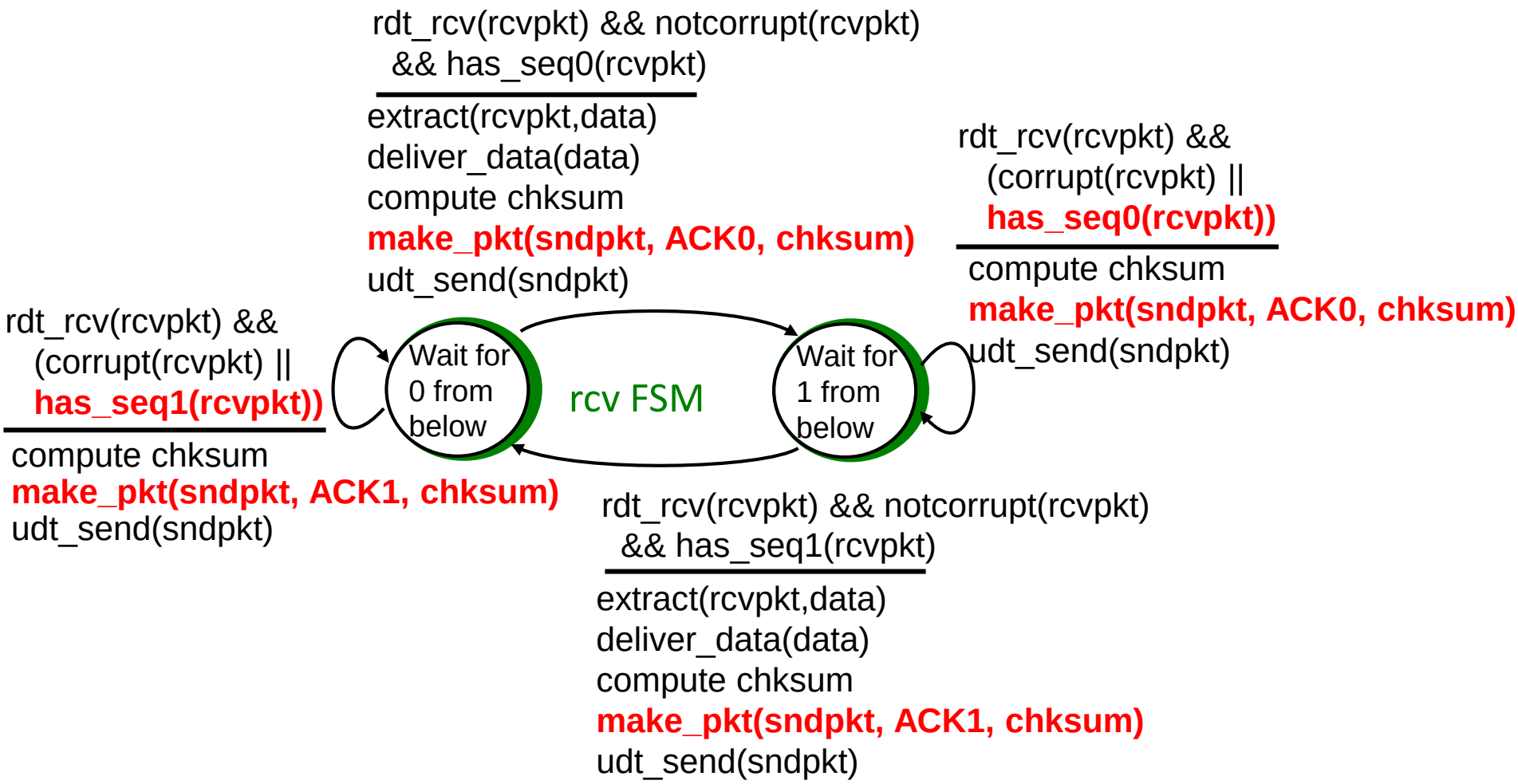
Rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACK only
- instead of NAK, receiver sends ACK for **last pkt received OK**
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

Rdt2.2: sender FSM



Rdt2.2: receiver FSM



Rdt3.0: channels with errors and loss

New assumption: underlying channel can also lose packets (data or ACKs)

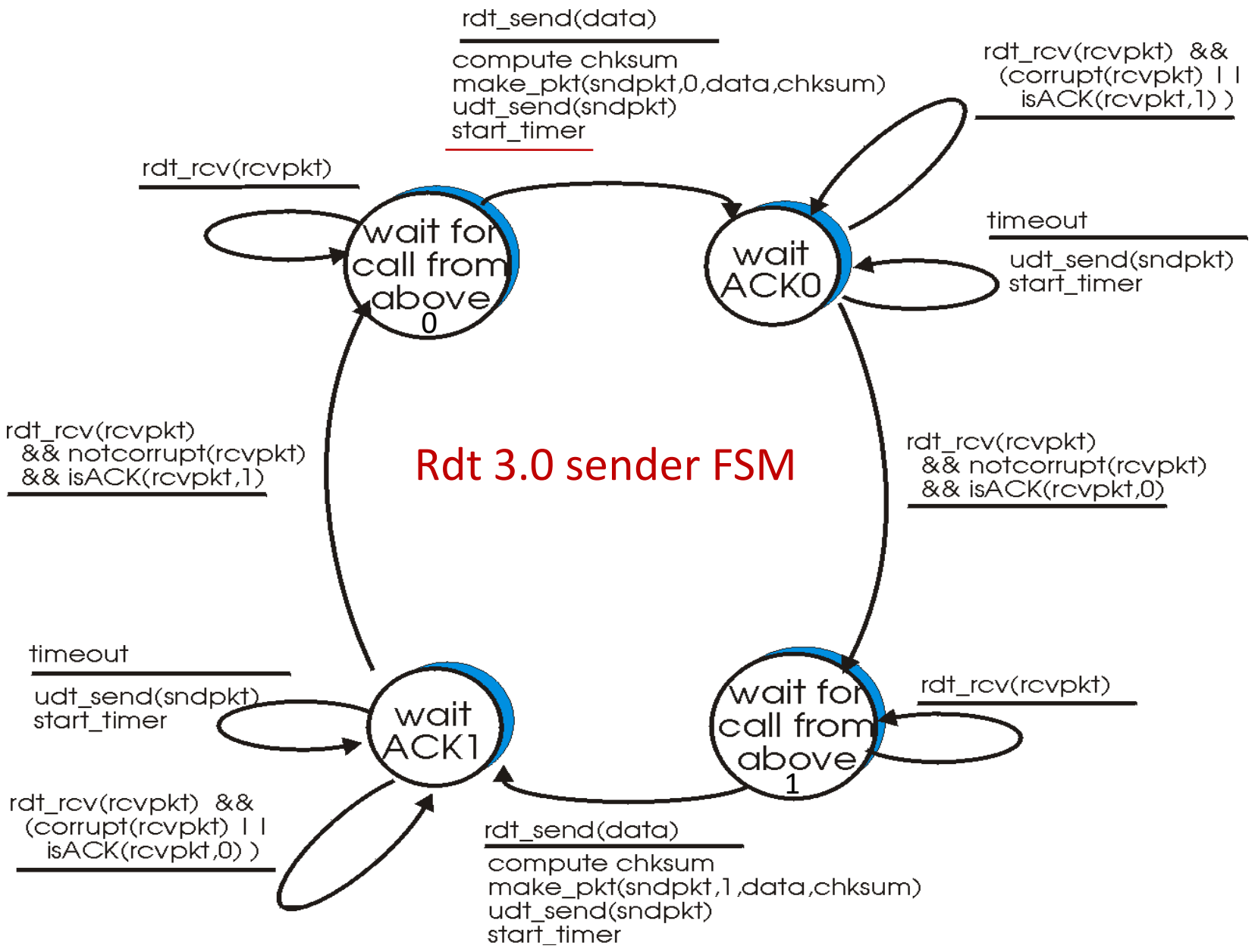
- how to detect packet loss and what to do when packet loss occurs
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Q: how to deal with loss?

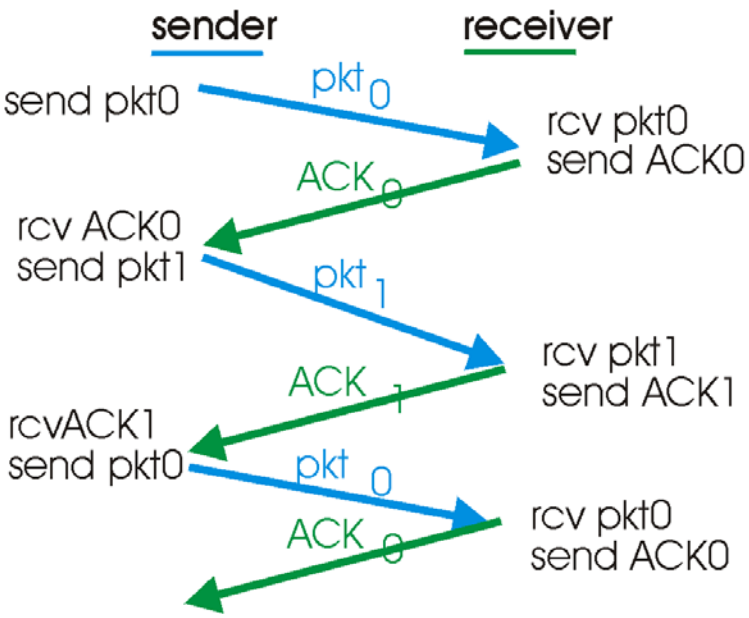
- sender waits until certain data or ACK lost, then retransmits
- drawbacks?

Approach: sender waits "reasonable" amount of time for ACK

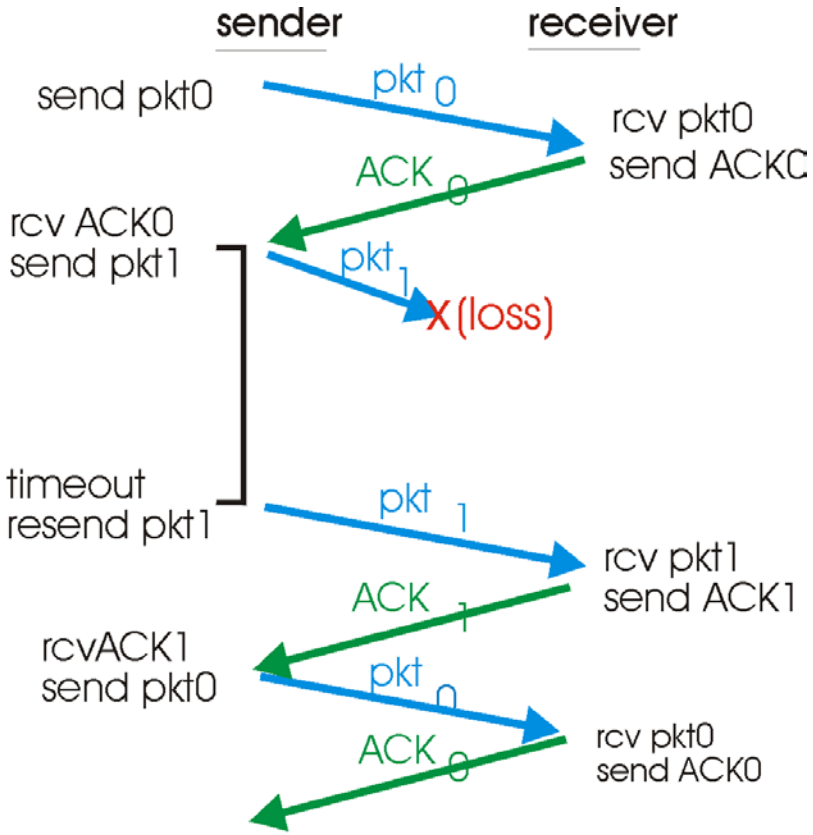
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer


rdt_send(data)
compute chksum
make_pkt(sndpkt,0,data,chksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) | |
isACK(rcvpkt,1))
rdt_rcv(rcvpkt)
wait for
call from
above
0
wait
ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
Rdt 3.0 sender FSM
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
timeout
udt_send(sndpkt)
start_timer
wait
ACK1
wait for
call from
above
1
rdt_rcv(rcvpkt)
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) | |
isACK(rcvpkt,0))
rdt_send(data)
compute chksum
make_pkt(sndpkt,1,data,chksum)
udt_send(sndpkt)
start_timer

Rdt3.0 in action

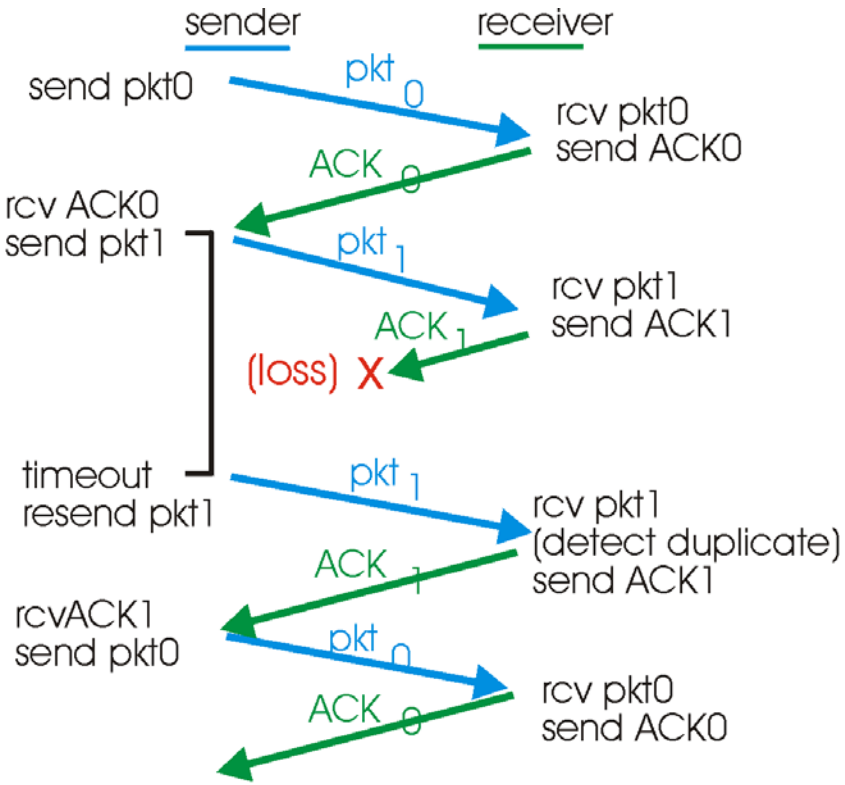


(a) operation with no loss

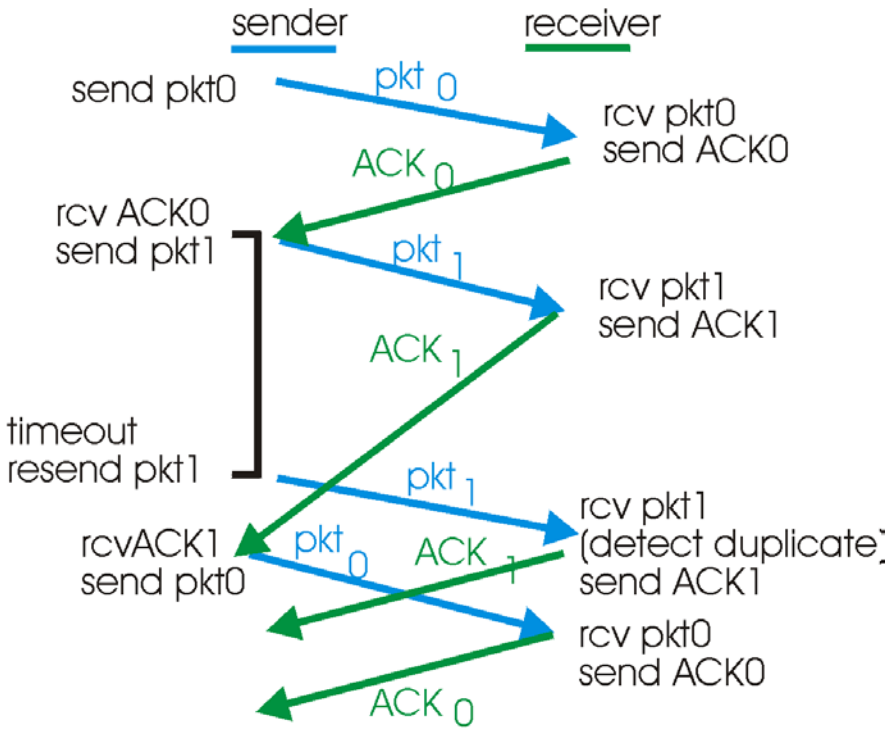


(b) lost packet

Rdt3.0 in action (cont.)



(c) lost ACK



(d) premature timeout

Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1Gbps link, 15ms end-end propagation delay, 8K bits packet:

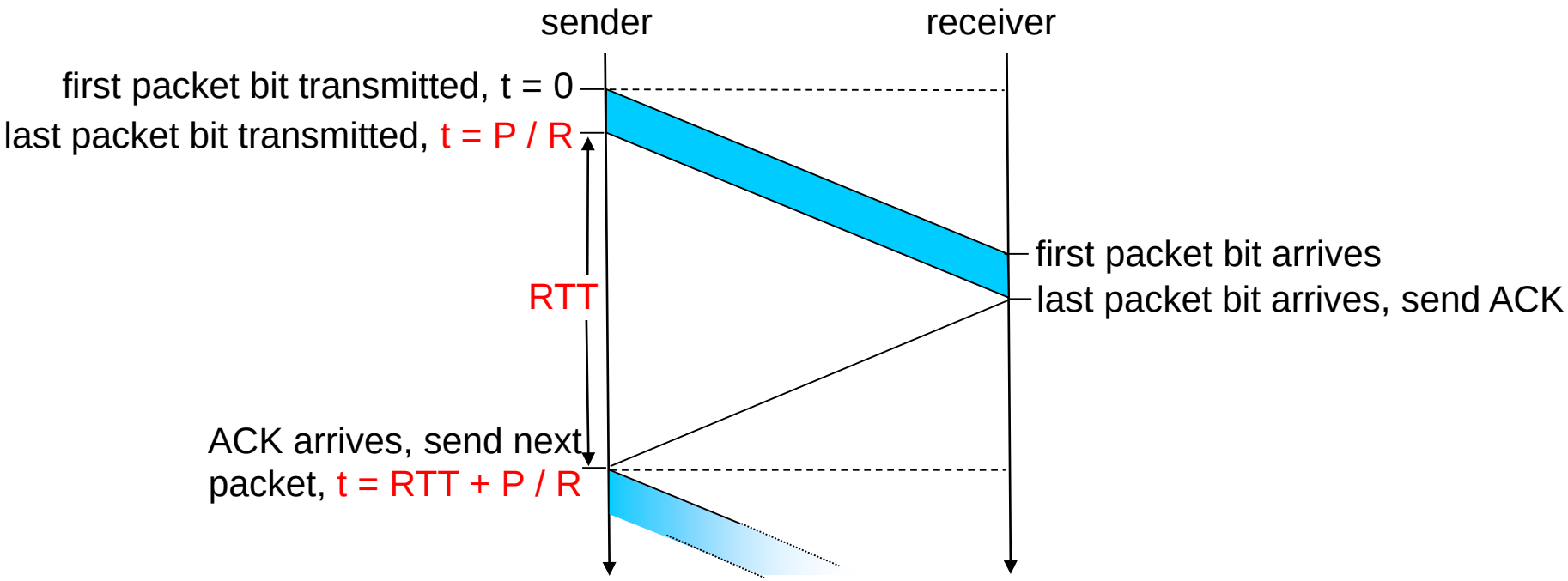
$$T_{transmit} = \frac{P\ (\text{packet length in bits})}{R\ (\text{transmission rate, bps})} = \frac{8000}{10^9 \text{bit/sec}} = 8\ \text{microsec}$$

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

not compute the time of sending an ACK packet

- U_{sender}: utilization – fraction of time sender busy sending
- 1KB pkt every 30 msec → 33kB/sec throughput over 1Gbps link
- network protocol limits use of physical resources!

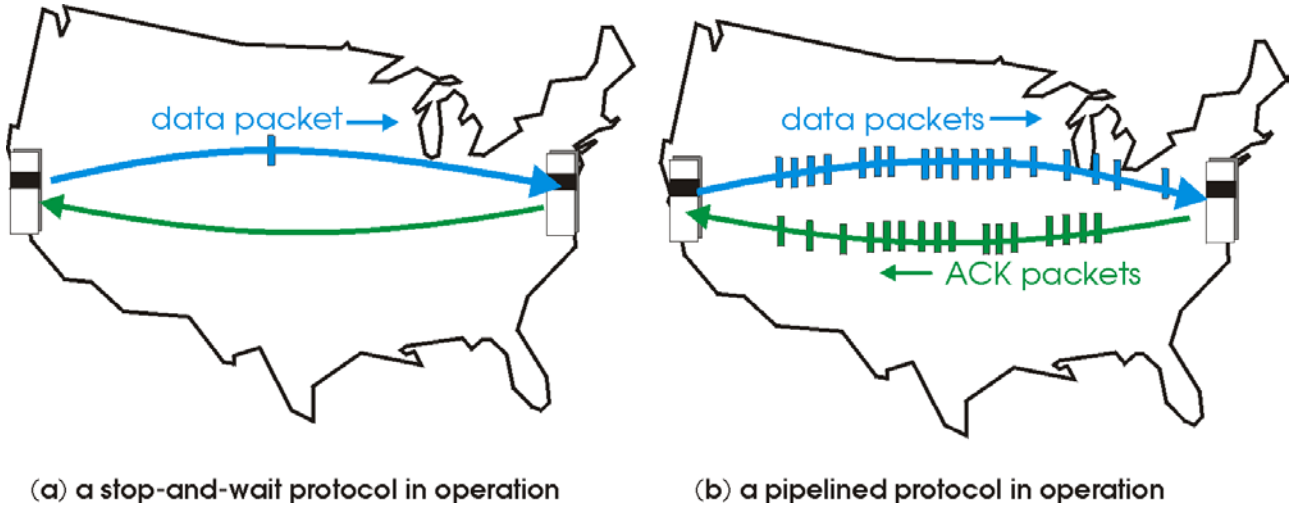
Rdt3.0: stop-and-wait operation



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

Pipelined protocols

- Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts
  - range of sequence numbers must be increased
  - buffering at sender and/or receiver

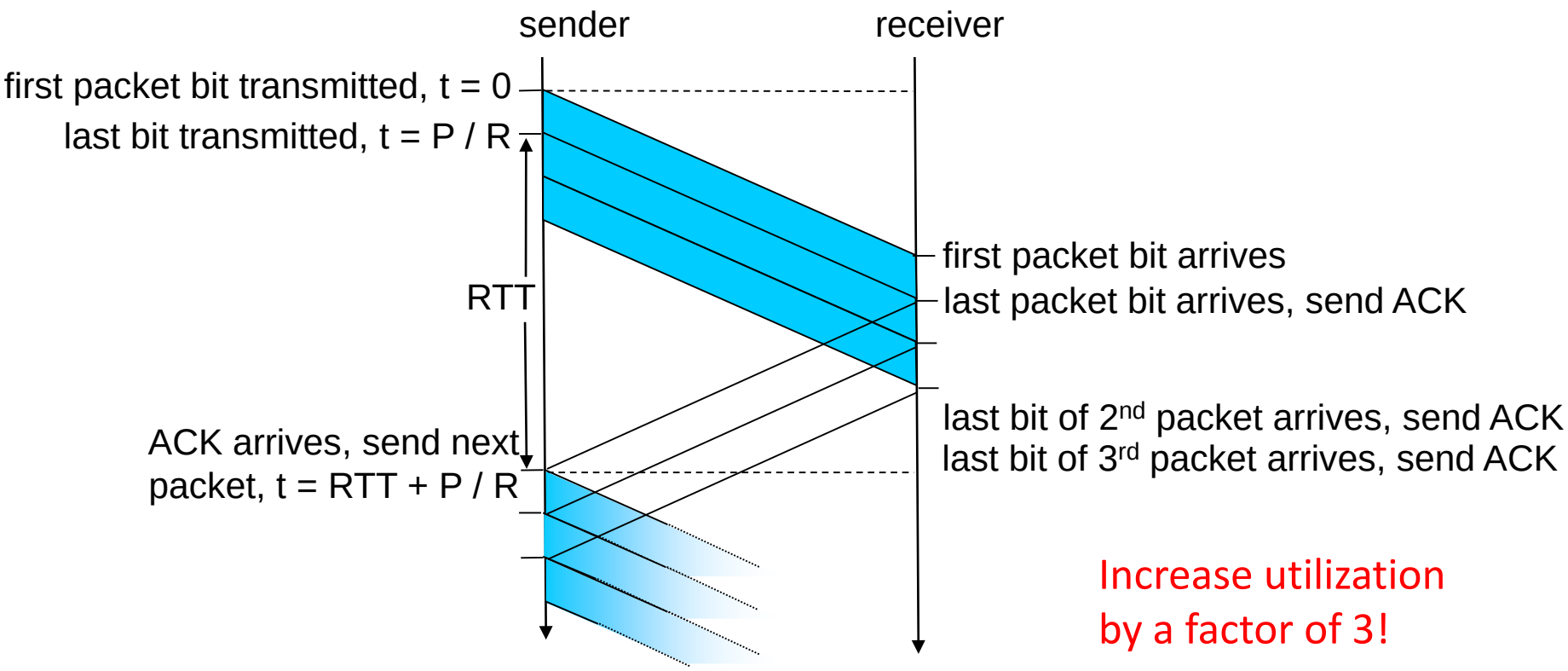


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- Two generic forms of pipelined protocols: *go-Back-N, selective repeat (SR)*

Pipelining: increased utilization



Increase utilization by a factor of 3!

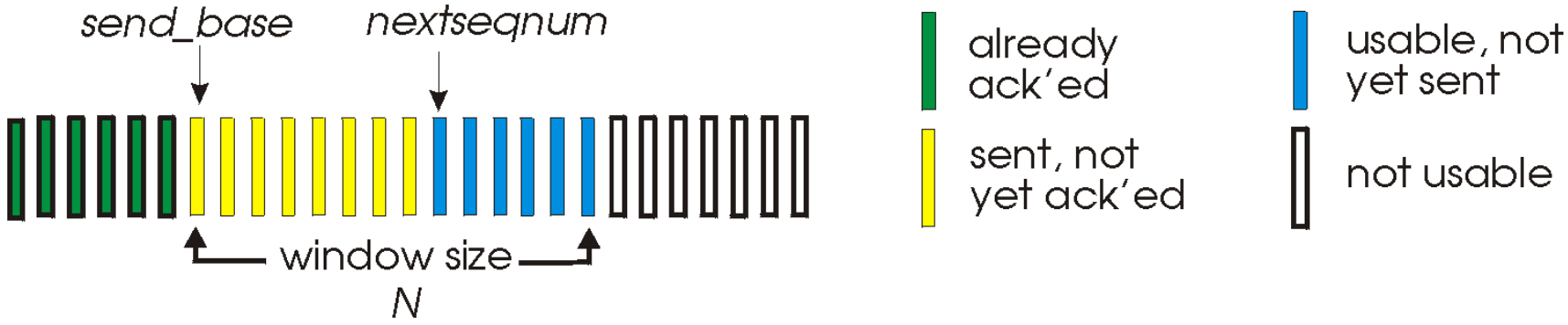
$$U_{sender} = \frac{3 * P / R}{RTT + P / R} = \frac{0.024}{30.008} = 0.0008$$

Go-Back-N

Q: why limit the usable seq# by window size?

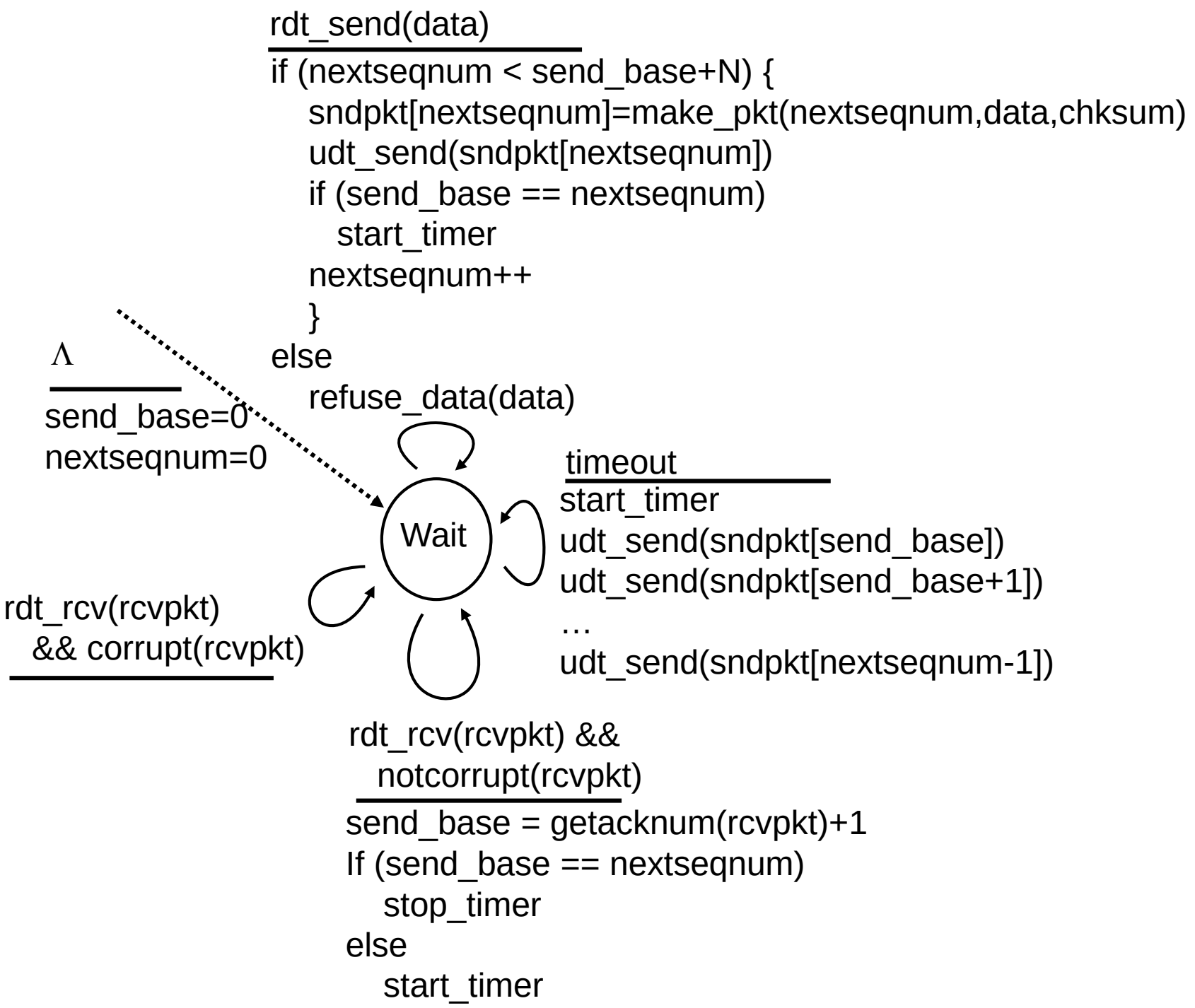
Sender:

- k-bit seq # in pkt header ([0, 2^k-1])
- “window” of up to N, consecutive unack’ed pkts allowed

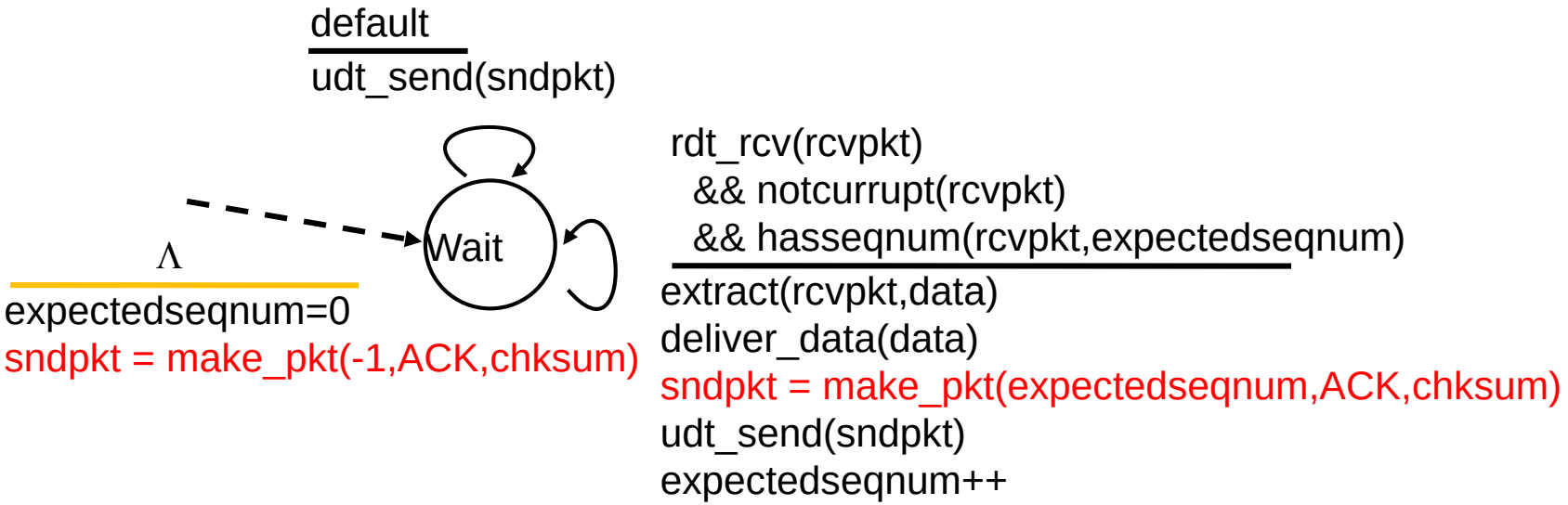


- ACK(n): ACK all pkts up to, including seq #n–cumulative ACK(累积确认)
  - may receive duplicate ACKs (see receiver)
- timer for in-flight pkt
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM

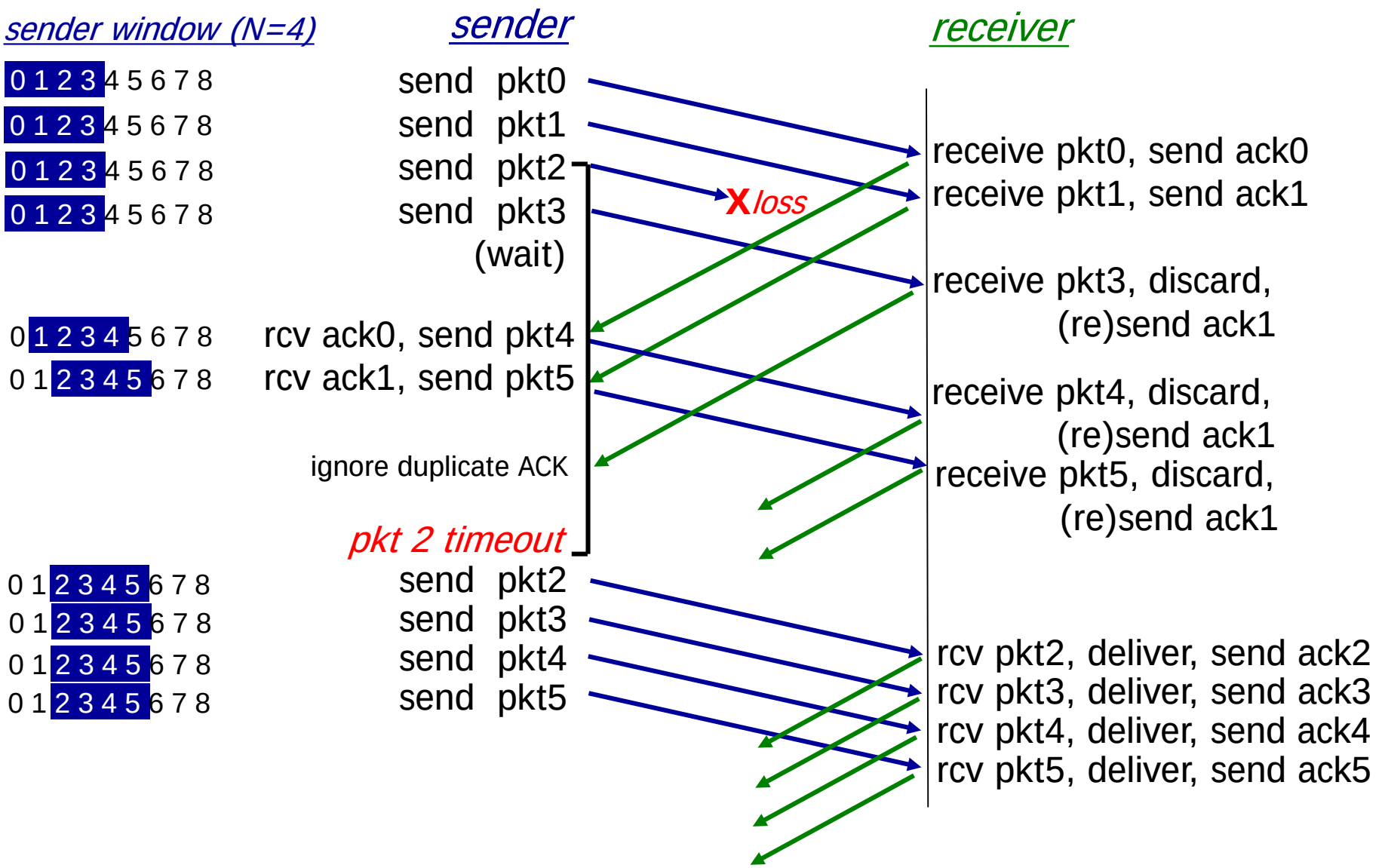


GBN: receiver extended FSM



- ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember **expectedseqnum**
- out-of-order pkt:
  - discard (don't buffer) →no receiver buffering!
  - **Re-ACK pkt with highest in-order seq #**

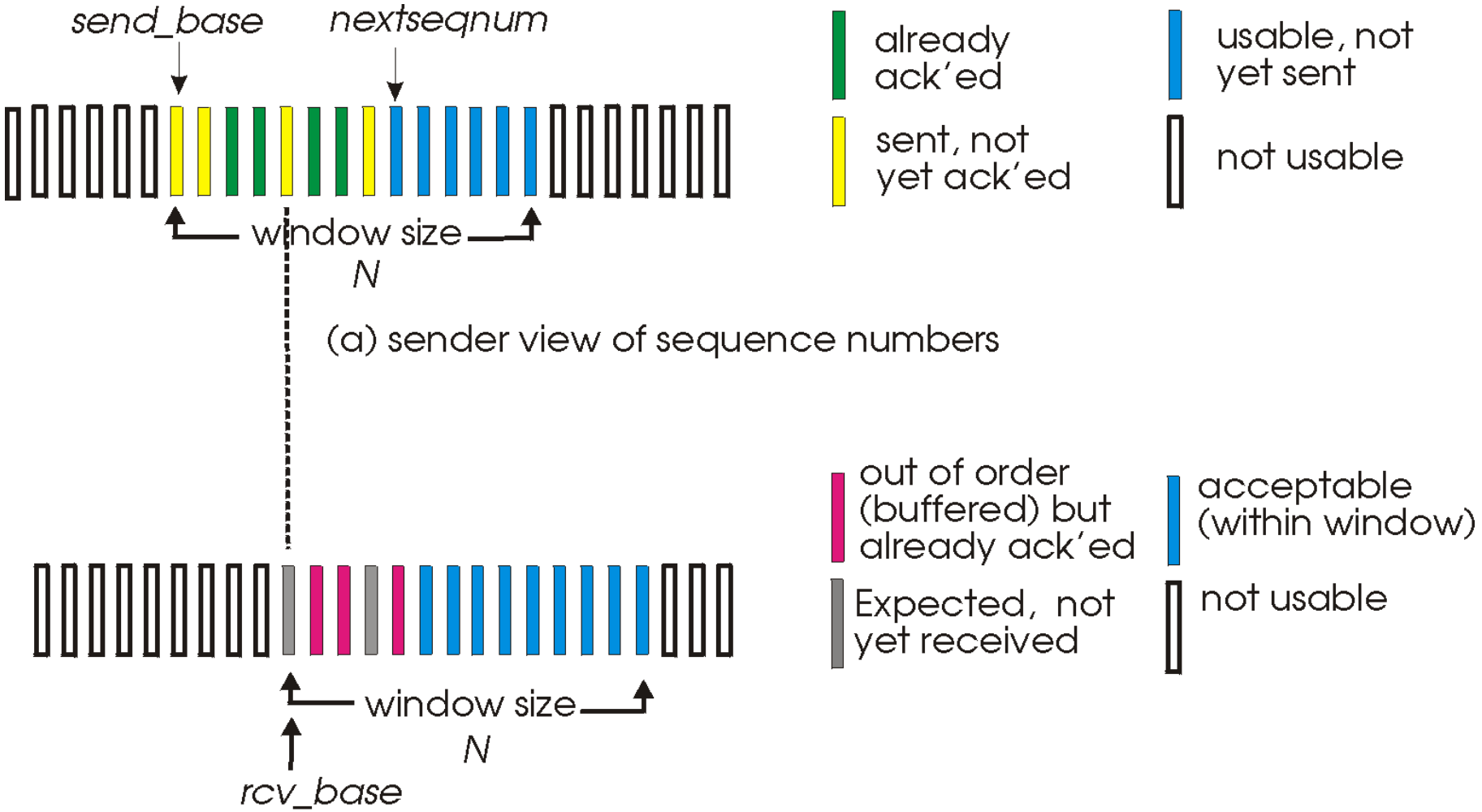
GBN in action



Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #’s
  - limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



Selective repeat

sender

data from above :

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N-1]:

- mark pkt n as received
- if n is the smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

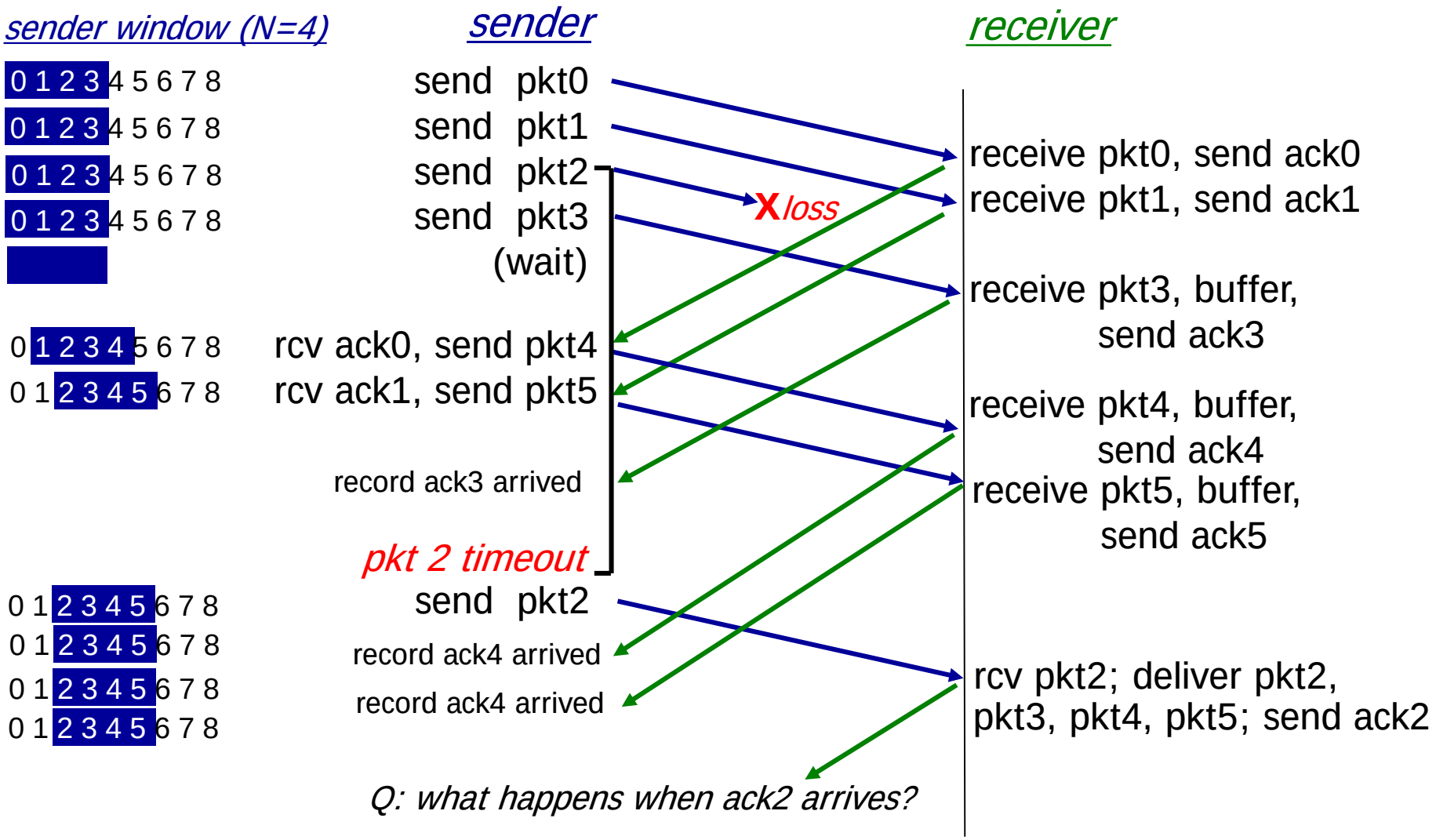
pkt n in [rcvbase-N,rcvbase-1]

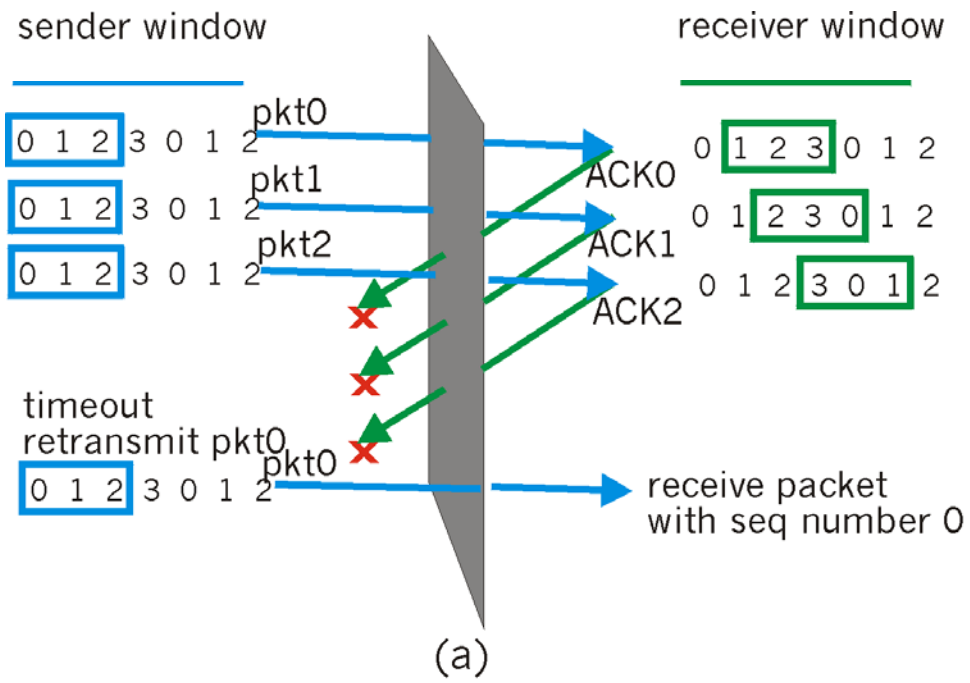
- ACK(n)

otherwise:

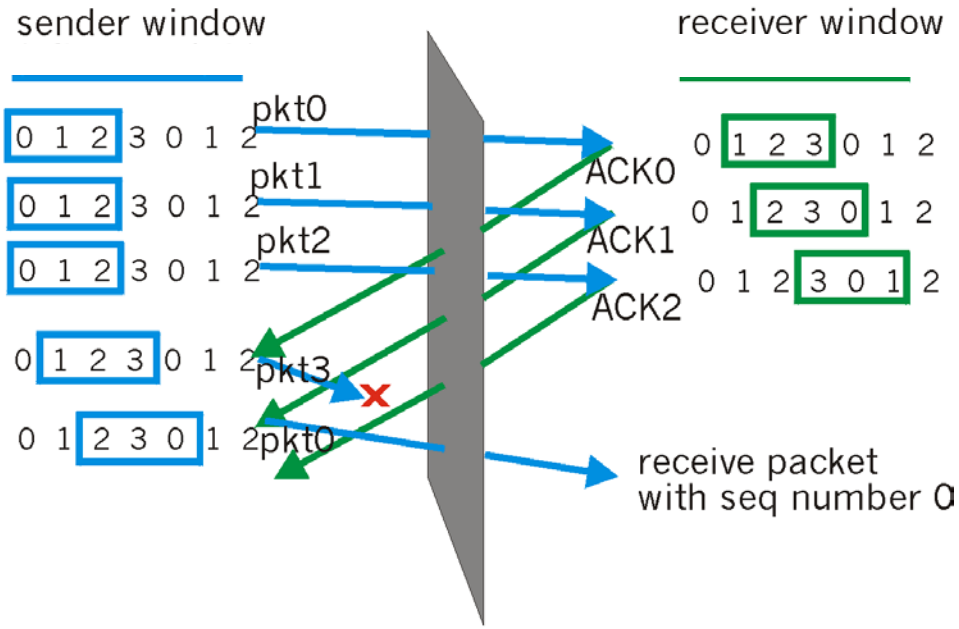
- ignore

Selective repeat in action




sender window
receiver window
0 1 2 3 0 1 2
pkt0
pkt1
pkt2
ACK0
ACK1
ACK2
timeout
retransmit pkt0
pkt0
receive packet
with seq number 0

(a)


sender window
receiver window
pkt0
pkt1
pkt2
ACK0
ACK1
ACK2
pkt3
pkt0
receive packet
with seq number 0

(b)

Selective repeat: dilemma

Example:

- seq #’s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q1: consider the scenario of window size=2

Q2: what relationship between seq # size and window size?

3.5 传输控制协议TCP-特点

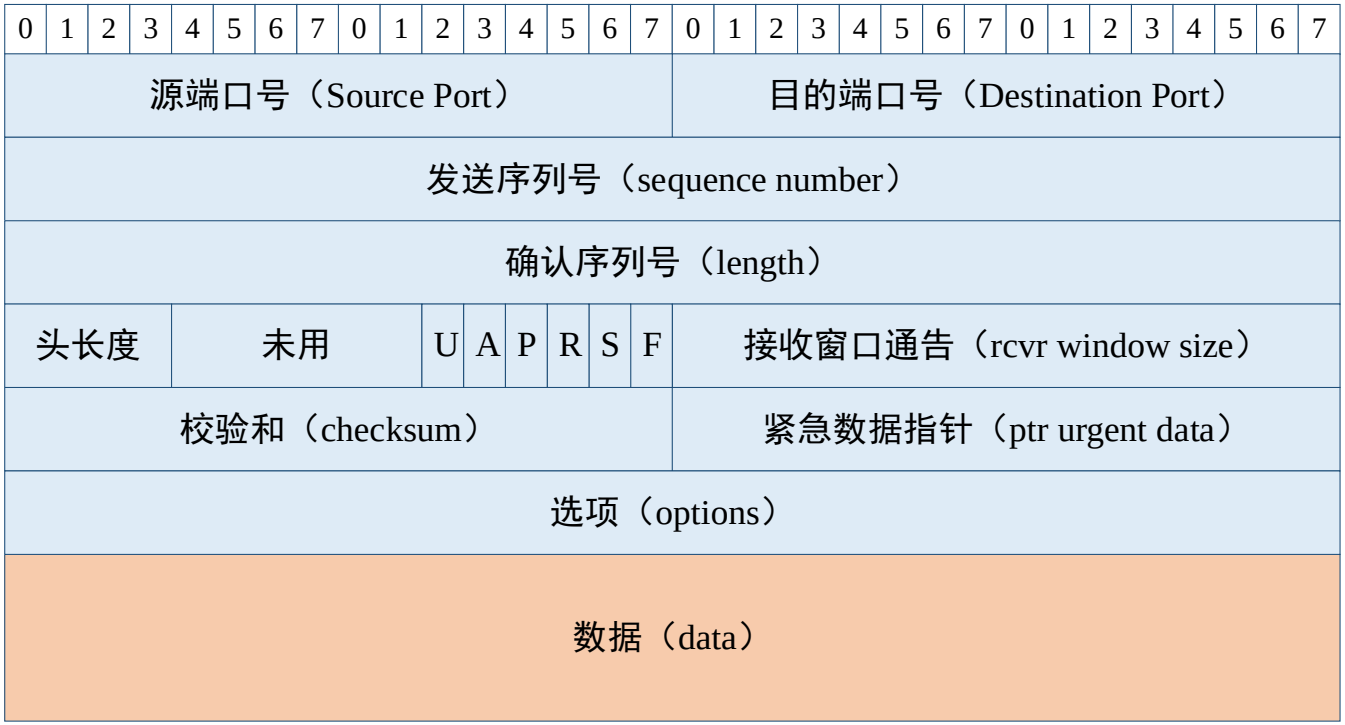
RFCs: 793,1122,1323, 2018, 2581

TCP协议特点

- 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
- 可靠服务：按序、可靠投递
  - 提供字节流服务，不识别消息边界
  - 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
- 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

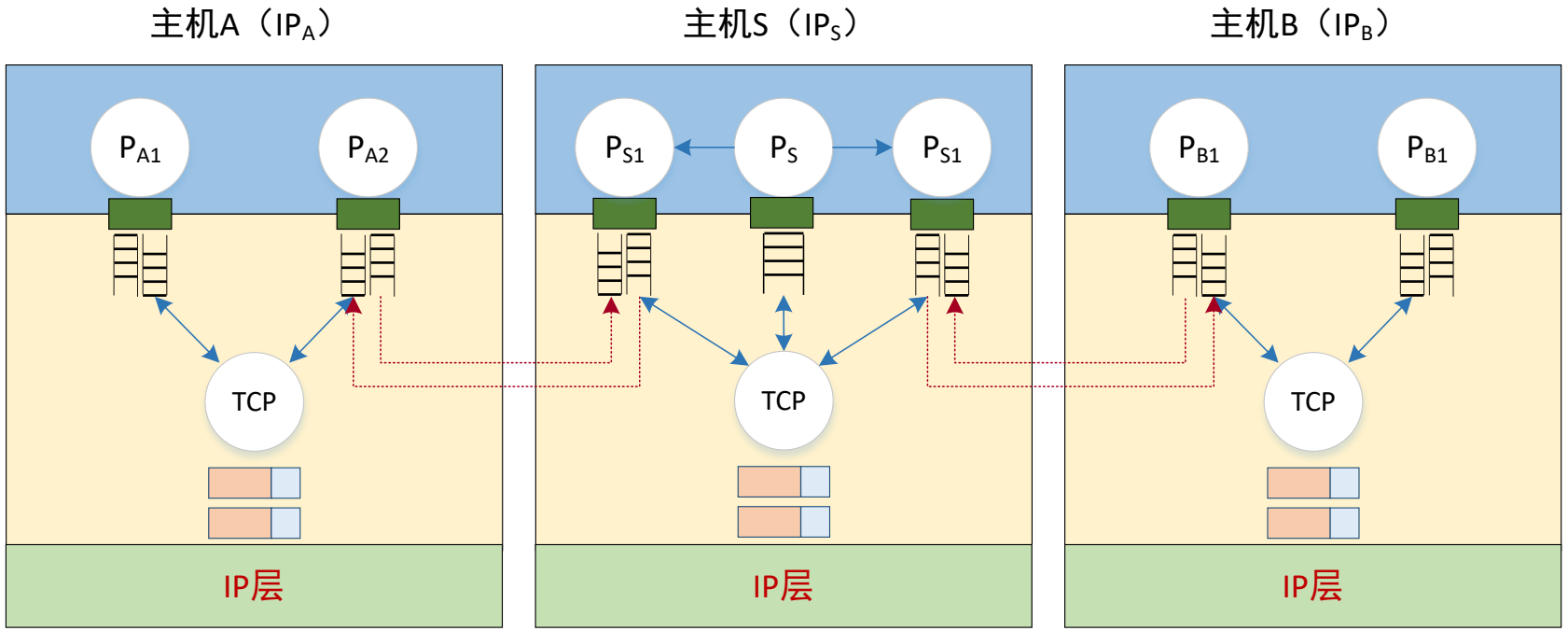


- 头长度：四个字节为计数单位，包含选项部分
- 接收窗口通告：指示接收缓冲区可接收的字节数
- 标志位：URG, ACK, PUSH, RESET, SYN, FIN
- 选项格式：

Kind（1字节）	Length（1字节）	Info（n字节）

3.5 传输控制协议TCP-复用分用

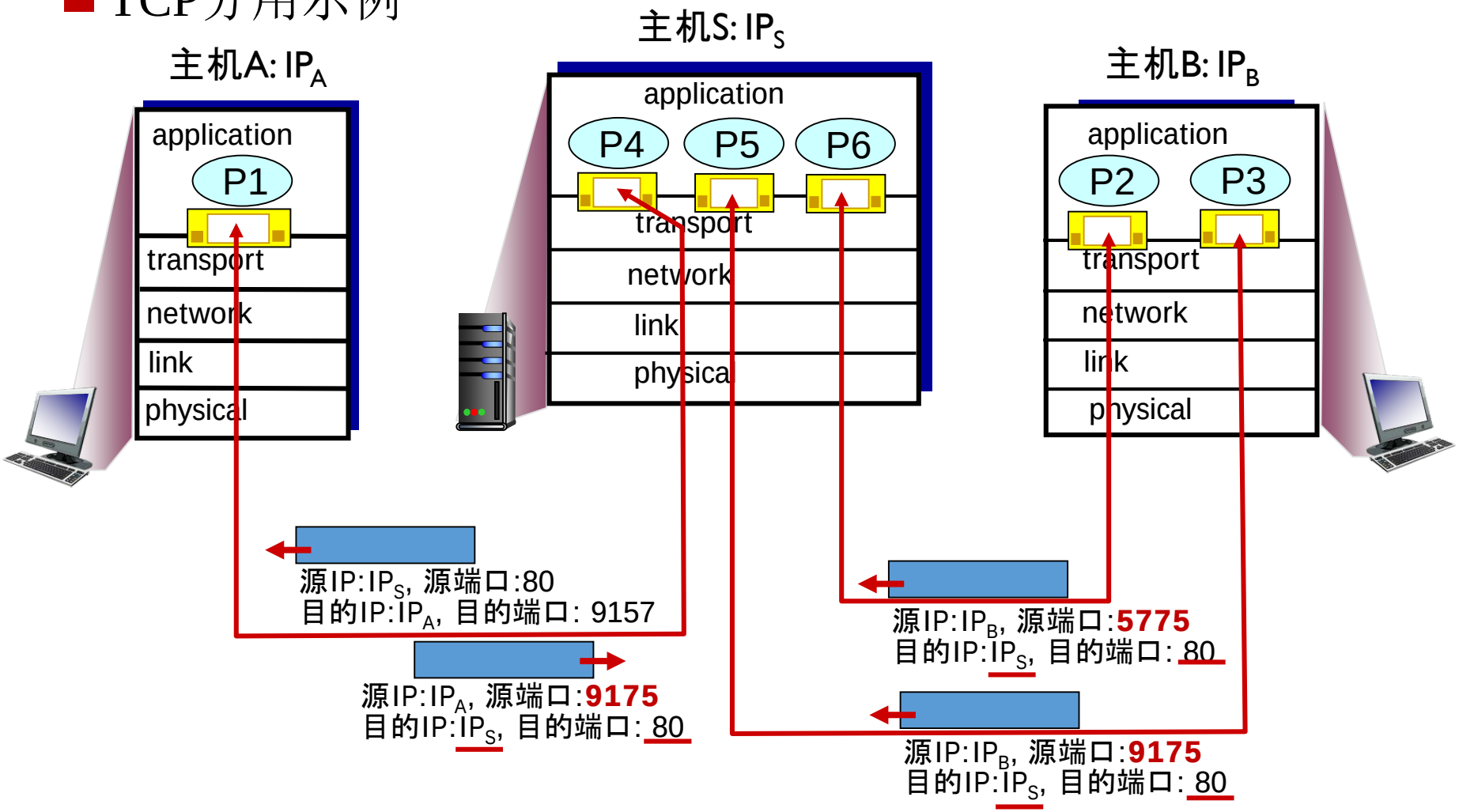
TCP连接与复用、分用机制



- 通信之前通过三次握手建立TCP连接
  - 分配缓冲区、协商参数（初始序列号、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**

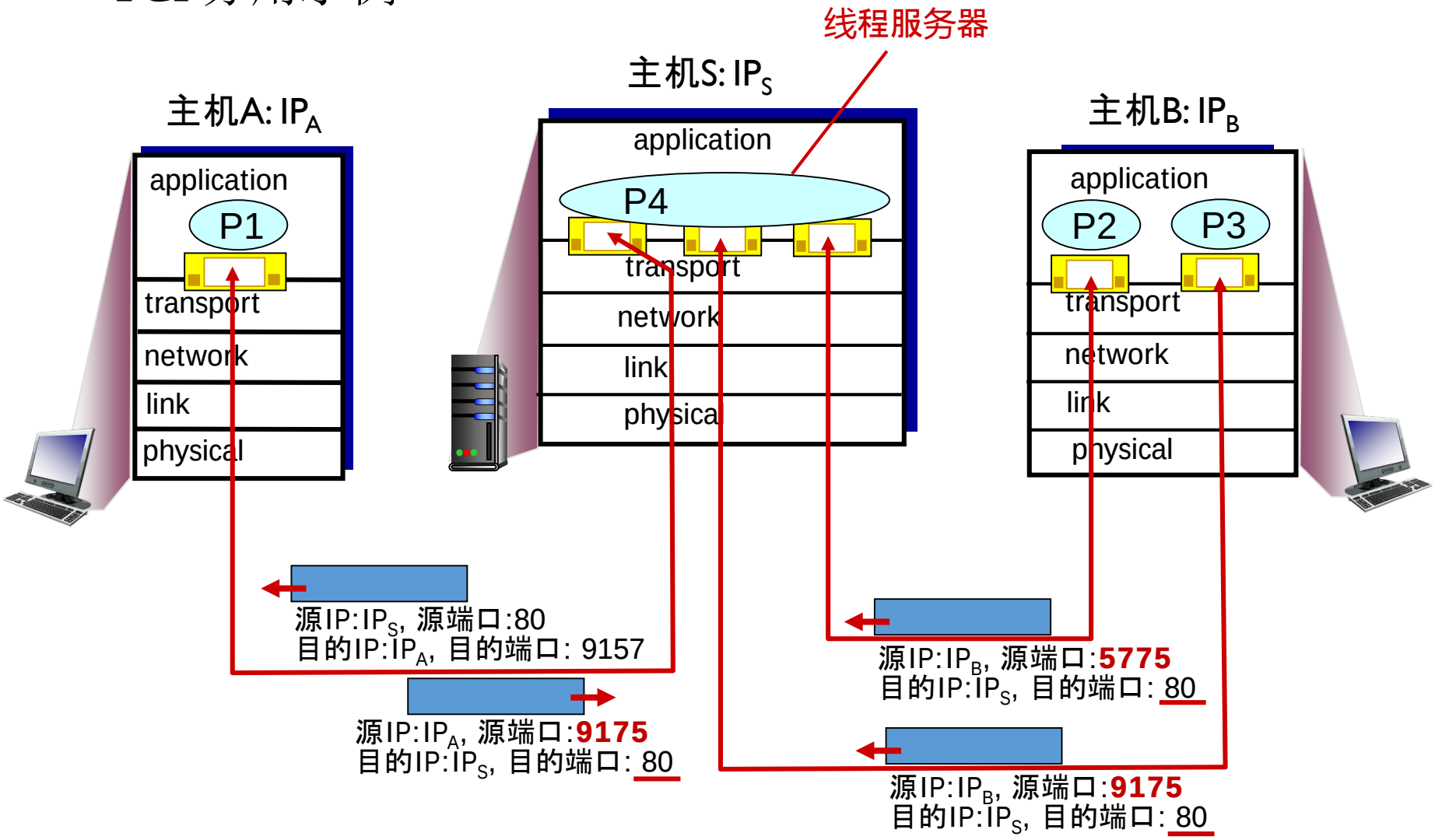
3.5 传输控制协议TCP-复用分用

- TCP分用示例

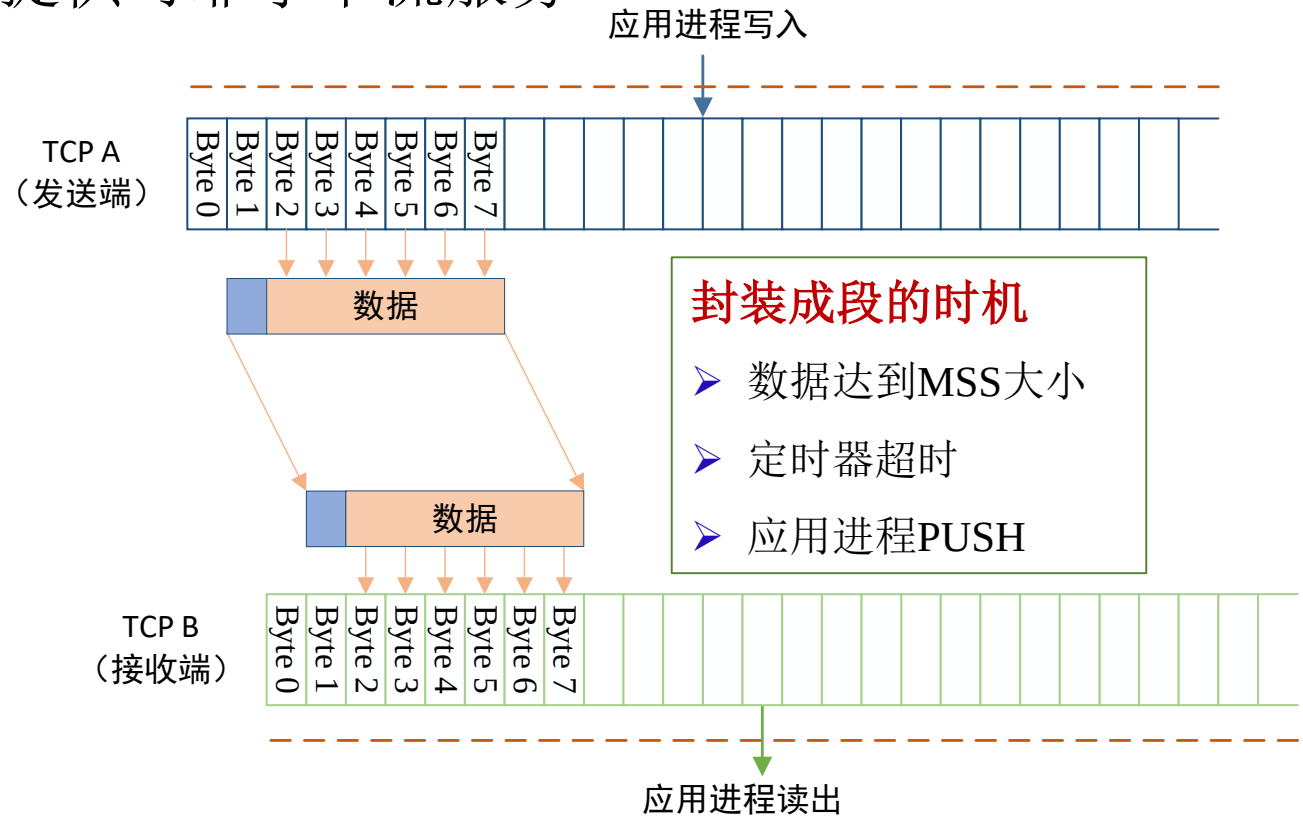


三个段的目的IP地址均为 IP_S，目的端口均为80，但是由于源IP地址或源端口不同，指向不同的套接口

- TCP分用示例


线程服务器
主机S: IP_S
主机A: IP_A
主机B: IP_B
application
P4
transport
network
link
physical
application
P1
transport
network
link
physical
application
P2
P3
transport
network
link
physical
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

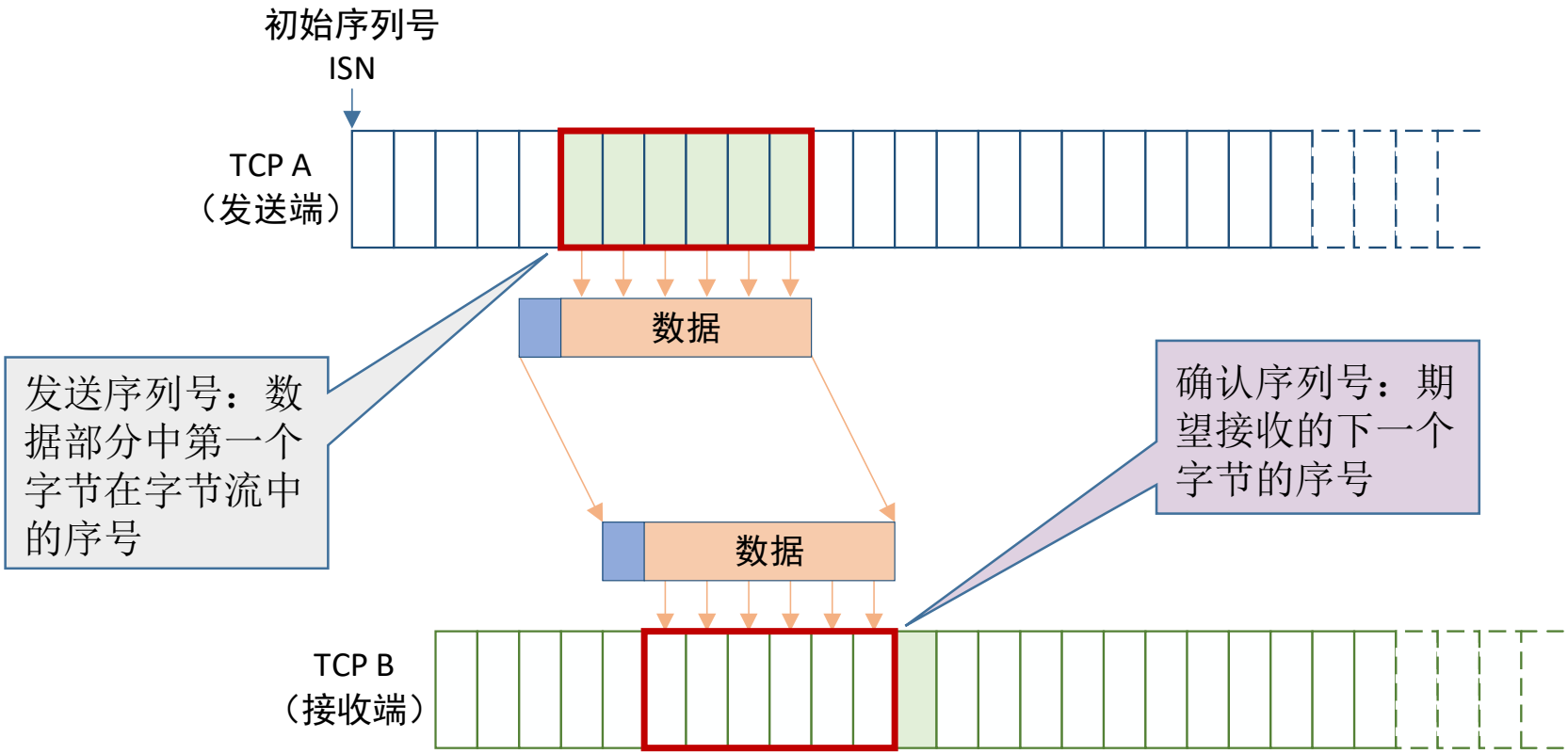
Kind=2	Length=4	最大报文段长度（2字节）

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
  - 发送序列号：32位
  - 确认序列号：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

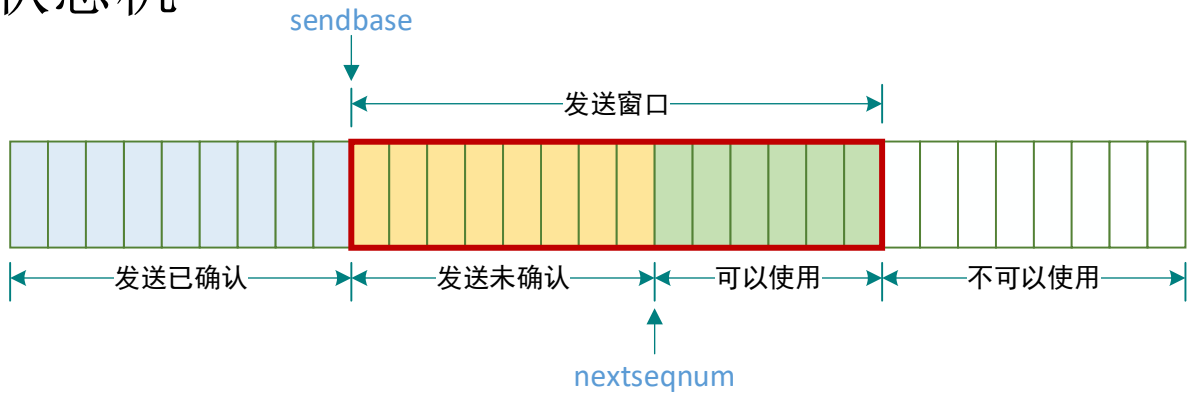
扩展：缓存乱序段，结合选择确认机制（SACK）

- TCP的发送序列号和确认序列号



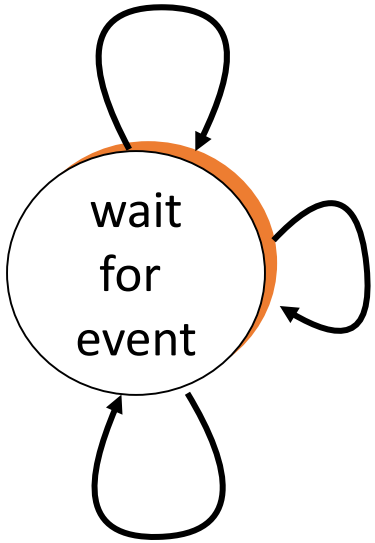
ISN选取: 每个TCP实体维护一个32为计数器，该计数器每4微秒增1，建立连接时从中读取计数器当前值

- 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

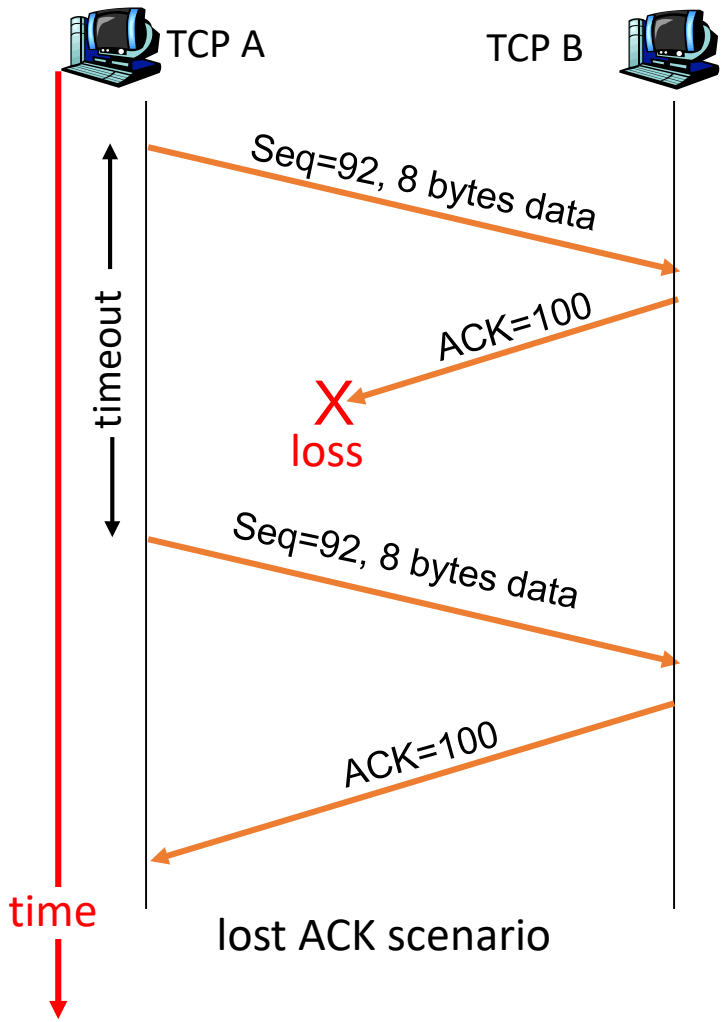
```
00  sendbase = initial_sequence number
01  nextseqnum = initial_sequence number
02
03  loop (forever) {
04     switch(event)
05     event: data received from application above
06          create TCP segment with sequence number nextseqnum
07          compute timeout interval for segment nextseqnum
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 */
```

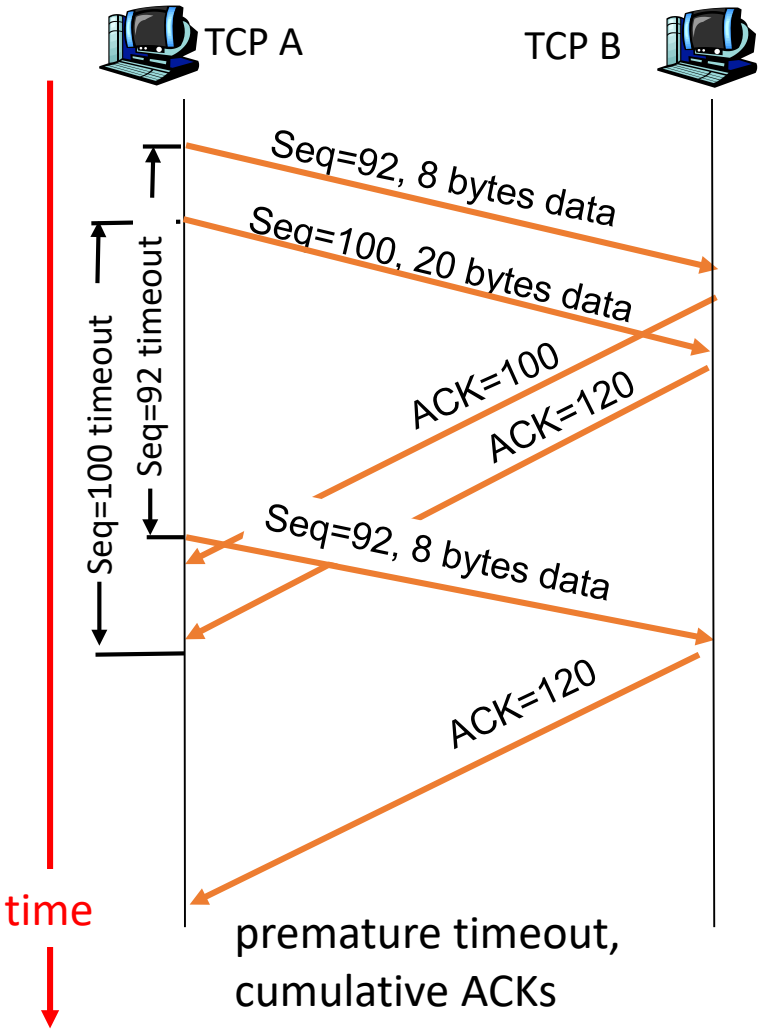
- TCP接收端ACK产生机制（RFC 1122, RFC 2581, RFC 5681）

Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK
out-of-order segment arrival higher-than-expect seq. # gap detected	send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap

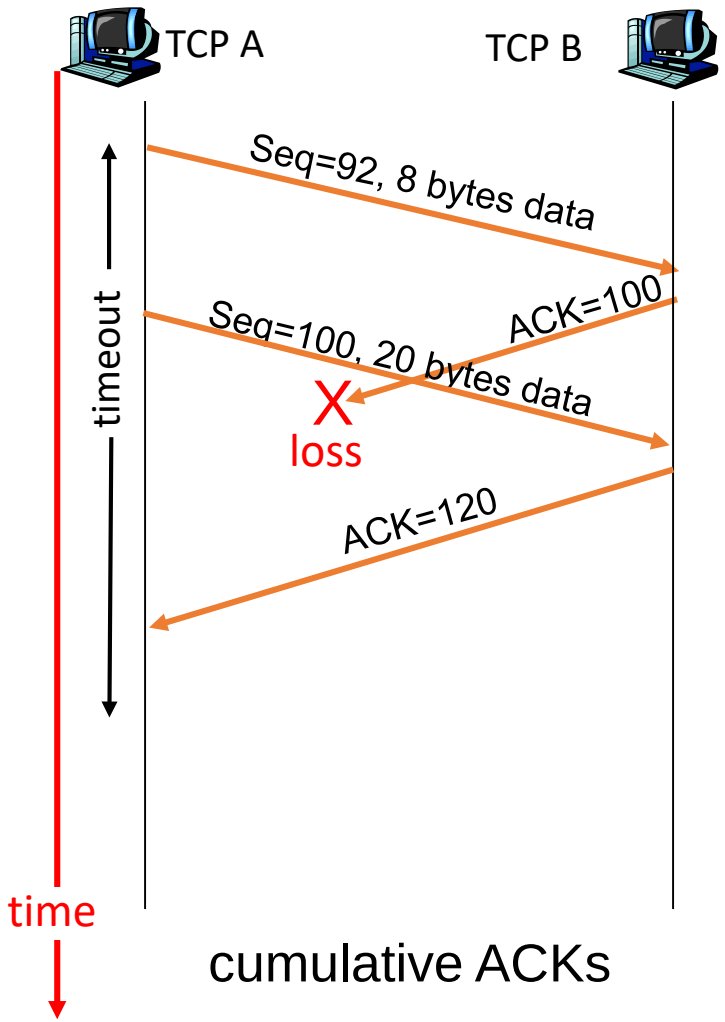
TCP基本的累积确认机制

- TCP重传场景


TCP A
TCP B
Seq=92, 8 bytes data
ACK=100
timeout
X
loss
Seq=92, 8 bytes data
ACK=100
time
lost ACK scenario

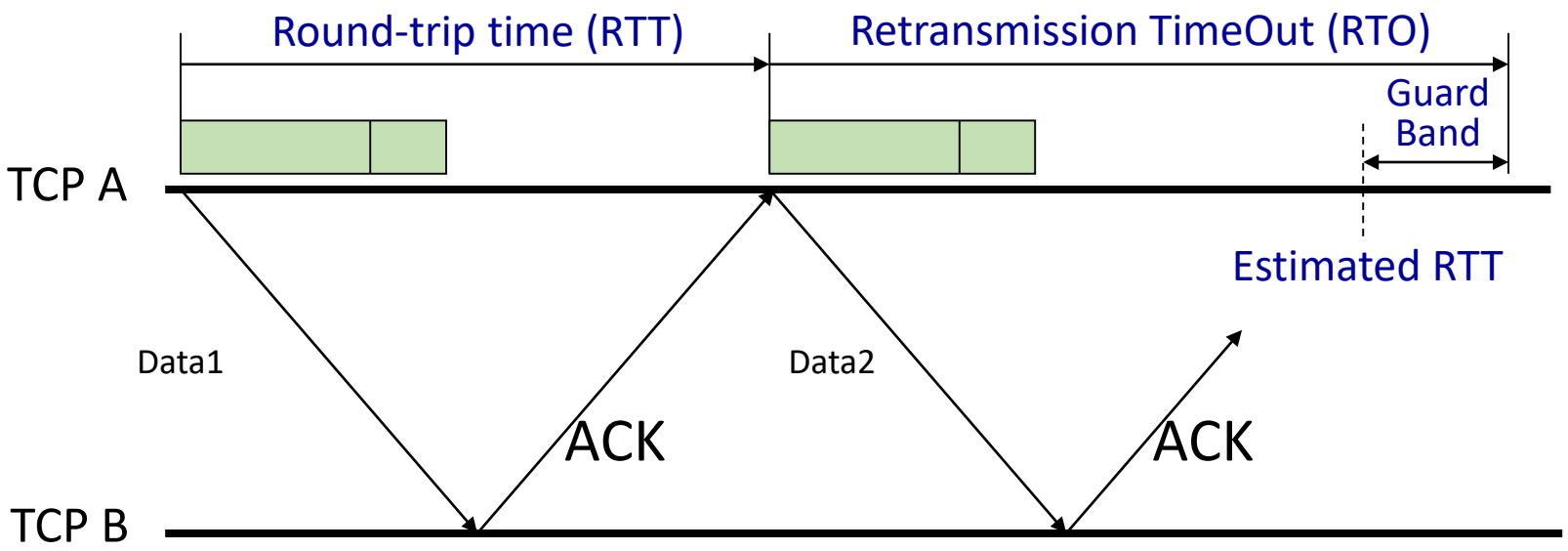

TCP A
TCP B
Seq=92, 8 bytes data
Seq=100, 20 bytes data
ACK=100
ACK=120
Seq=100 timeout
Seq=92 timeout
Seq=92, 8 bytes data
ACK=120
time
premature timeout,
cumulative ACKs

- TCP重传场景



- Retransmission policy is "Go Back N" plus SR
- TCP specification doesn't explain how receiver handles out-of-order segments
- Q: how to select the value of **timeout**?

- TCP重传超时时间考虑



- TCP采用自适应方法计算机重传超时时间
  - ✓ 基于往返时间（RTT）确定重传超时间时间（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

Picking the RTO is important:

- Pick a values that's too big and it will wait too long to retransmit a packet, introduce significant delays into the applications.
- Pick a value too small, and it will unnecessarily retransmit packets.

The original algorithm for picking RTO:

EstimatedRTT=α EstimatedRTT+(1 - α) SampleRTT (usual α=7/8)

RTO = β* EstimatedRTT (β>1, usual β= 2)

- EstimatedRTT is a weighted average of the SampleRTT , i.e., exponential weighted moving average (EWMA)

Characteristics of the original algorithm:

- Variance is assumed to be fixed.
- But in practice, variance increases as congestion increases.

- TCP重传超时时间计算（续）

Jacobson/Karels Algorithm

Newer Algorithm includes estimate of variance in RTT:

Difference = SampleRTT - EstimatedRTT

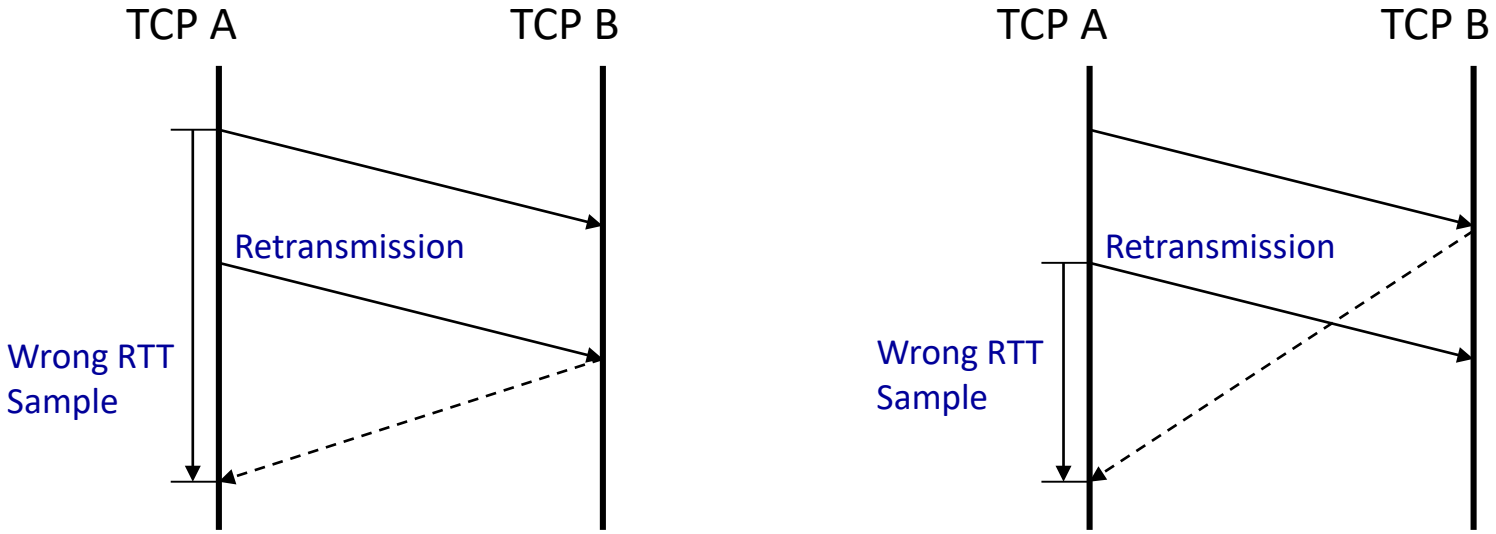
EstimatedRTT = EstimatedRTT + (δ*Difference)
= (1- δ) *EstimatedRTT+ δ* SampleRTT

Deviation = Deviation + δ*(|Difference| - Deviation)
= (1- δ) *Deviation+ δ* |Difference|

RTO = μ * EstimatedRTT + ϕ * Deviation
($\mu \approx 1$, $\phi \approx 4$)

- TCP重传超时时间计算（续）

karn/Partridge Algorithm



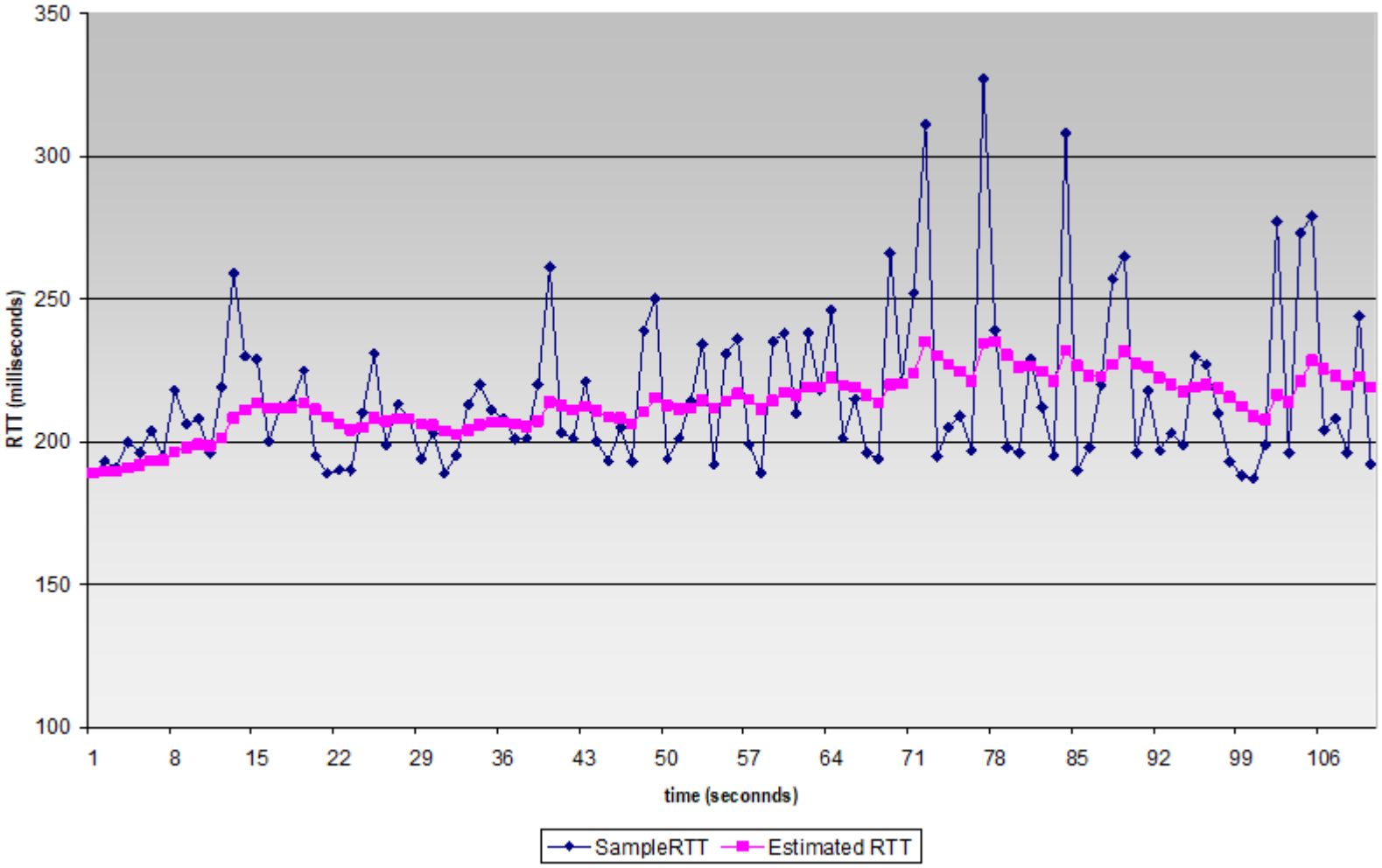
Problem:

How can we estimate RTT when segments are retransmitted?

Solution:

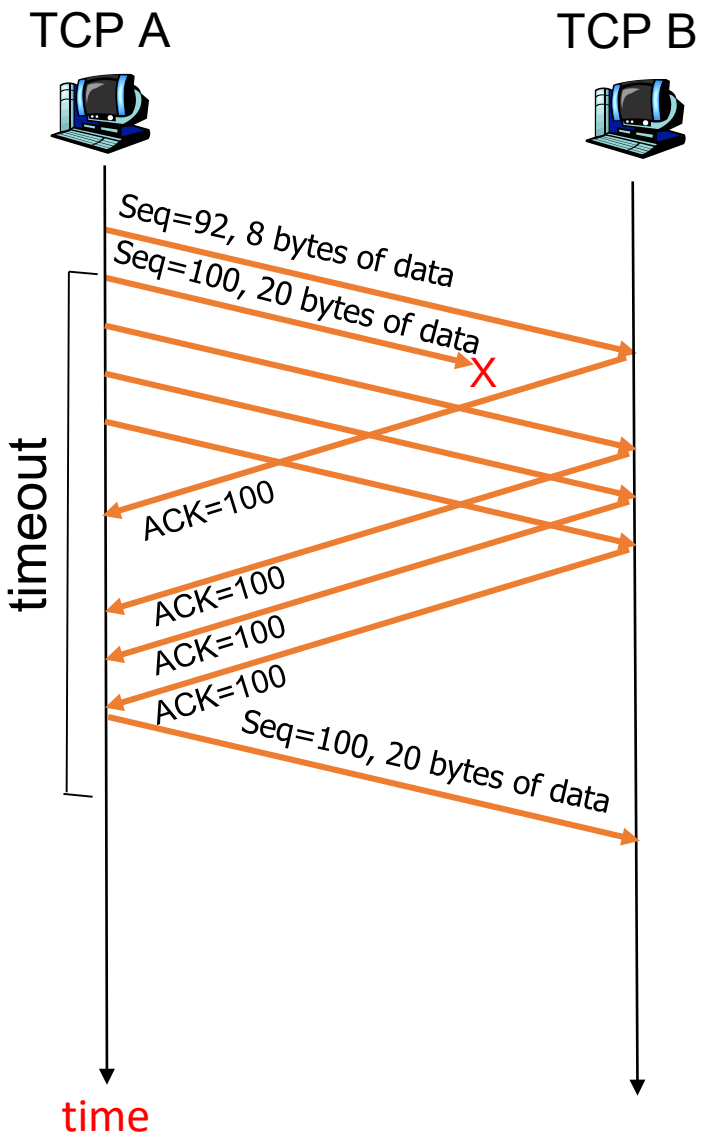
On retransmission, don't update estimated RTT (and double RTO).

- 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 */
            }
```

a duplicate ACK for already ACKed segment

fast retransmit

- 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）

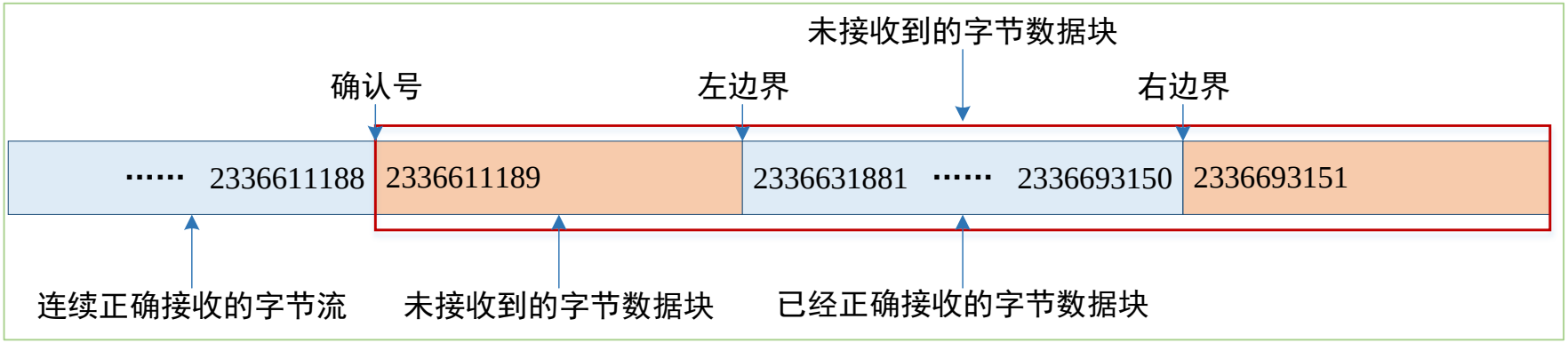
Kind=4	Length=2

  - 确认收到并缓存了的不连续的数据块。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]
```



TCP流量控制

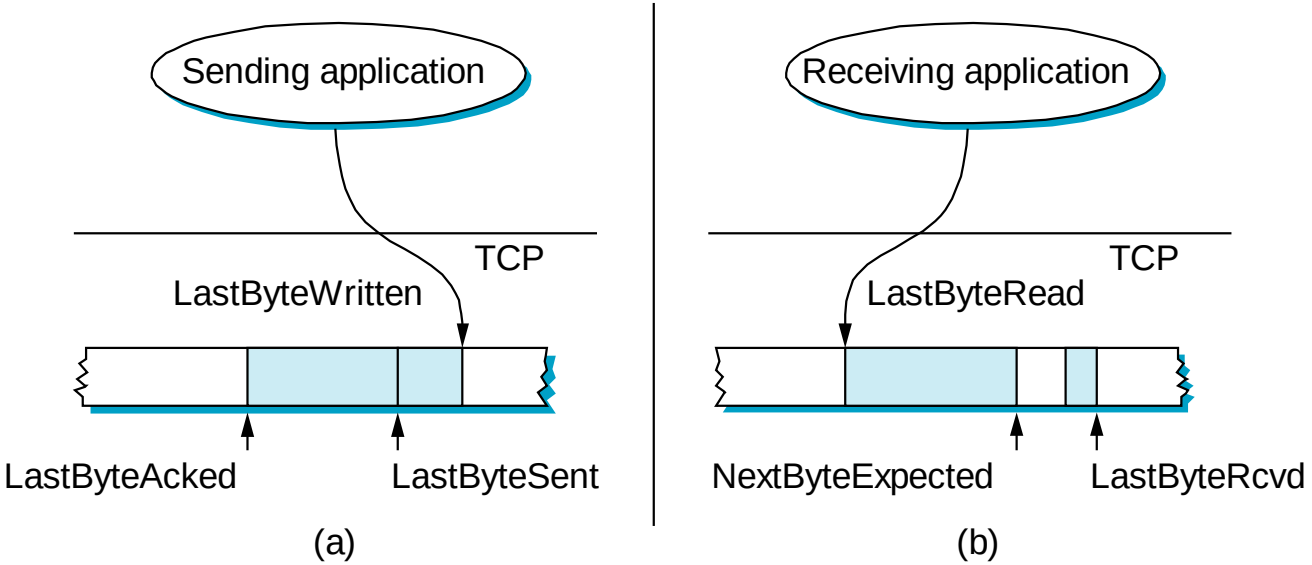
flow control

sender won't overrun receiver's buffers by transmitting too much, too fast

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space using **RcvWindow field** in TCP segment

sender: keeps the amount of transmitted, unACKed data less than most recently received **RcvWindow**

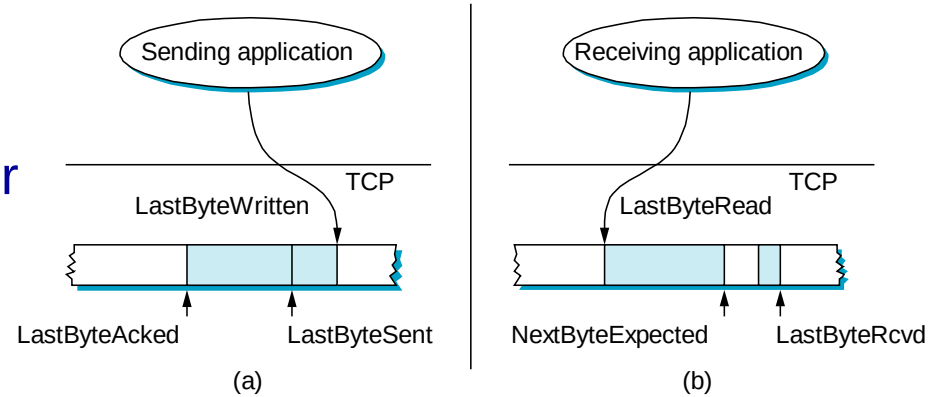
- 滑动窗口（Sliding Window）



- 发送端
  - LastByteAcked < = LastByteSent
  - LastByteSent < = LastByteWritten
  - buffer bytes between LastByteAcked and LastByteWritten
- 接收端
  - LastByteRead < NextByteExpected
  - NextByteExpected < = LastByteRcvd +1
  - buffer bytes between LastByteRead and LastByteRcvd

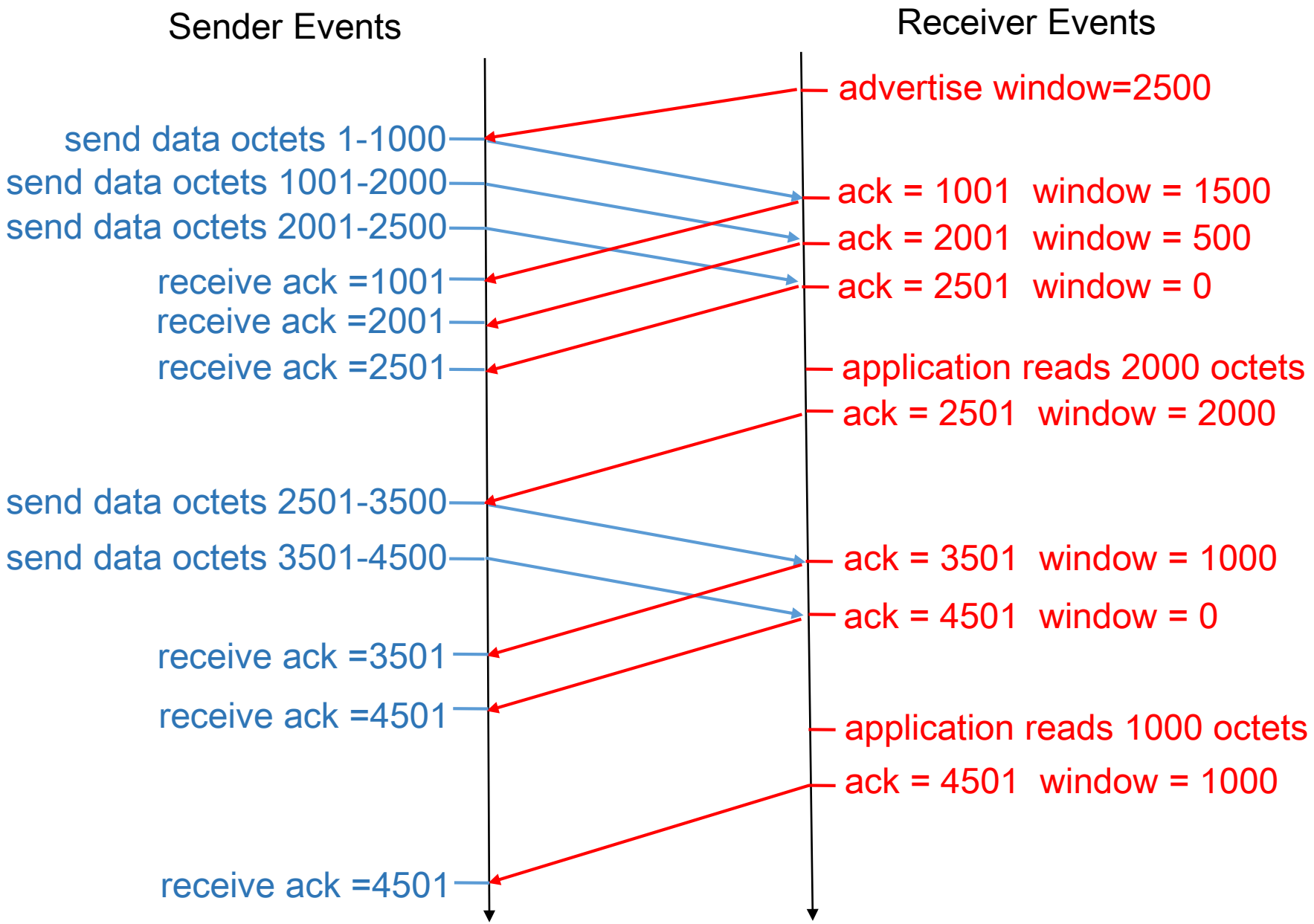
3.5 传输控制协议TCP—流量控制

- 流量控制约束
  - 发送缓冲区大小：MaxSendBuffer
  - 接收缓冲区大小：MaxRcvBuffer

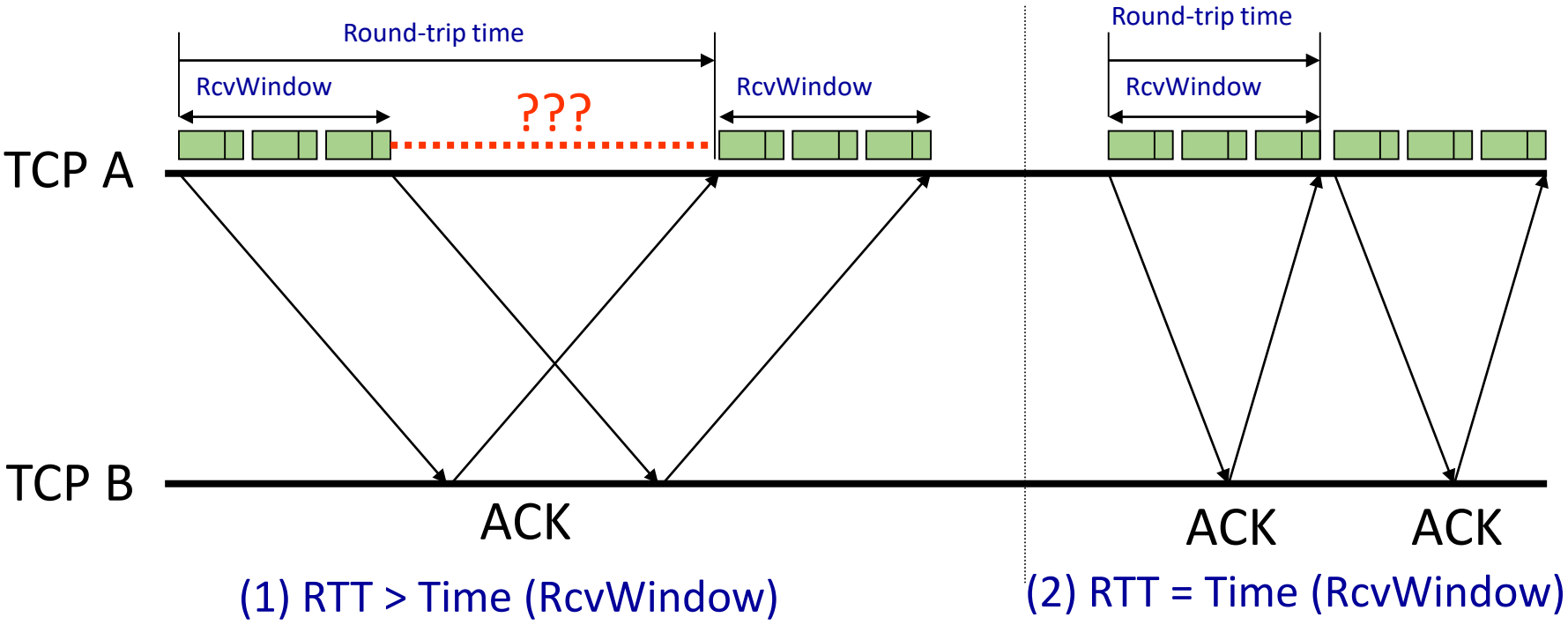


  - 接收端
    - LastByteRcvd - LastByteRead < = MaxRcvBuffer
    - AdvertisedWindow = MaxRcvBuffer - (NextByteExpected - LastByteRead)
  - 发送端
    - LastByteSent - LastByteAcked < = AdvertisedWindow
    - EffectiveWindow=AdvertisedWindow - (LastByteSent - LastByteAcked)
    - LastByteWritten - LastByteAcked < = MaxSendBuffer
    - block sender if (LastByteWritten - LastByteAcked) + y > MaxSenderBuffer

3.5 传输控制协议TCP—流量控制

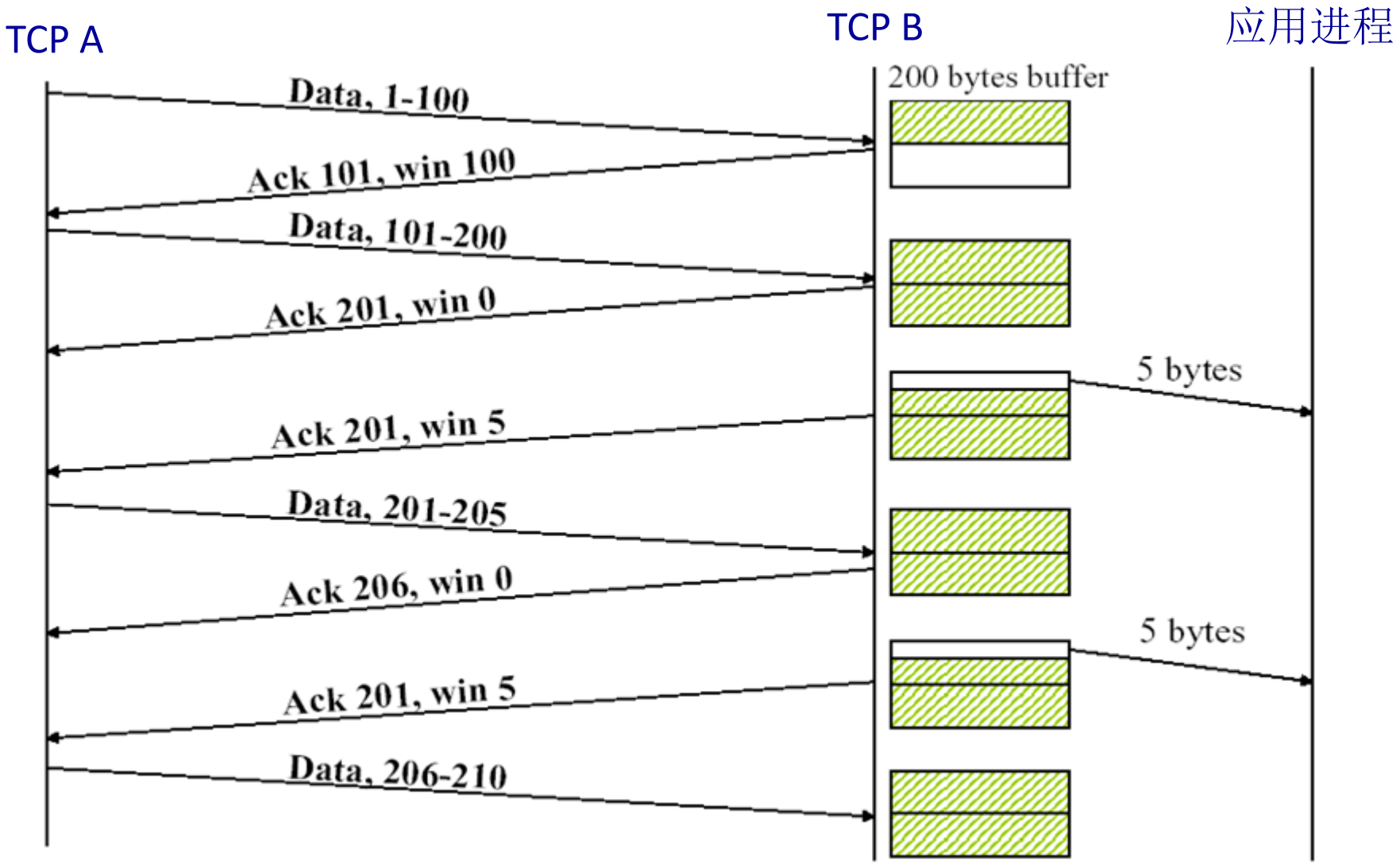


- 接收窗口大小对性能的影响



- usually 4k – 8k Bytes when connection set-up.

- 流量控制的性能问题

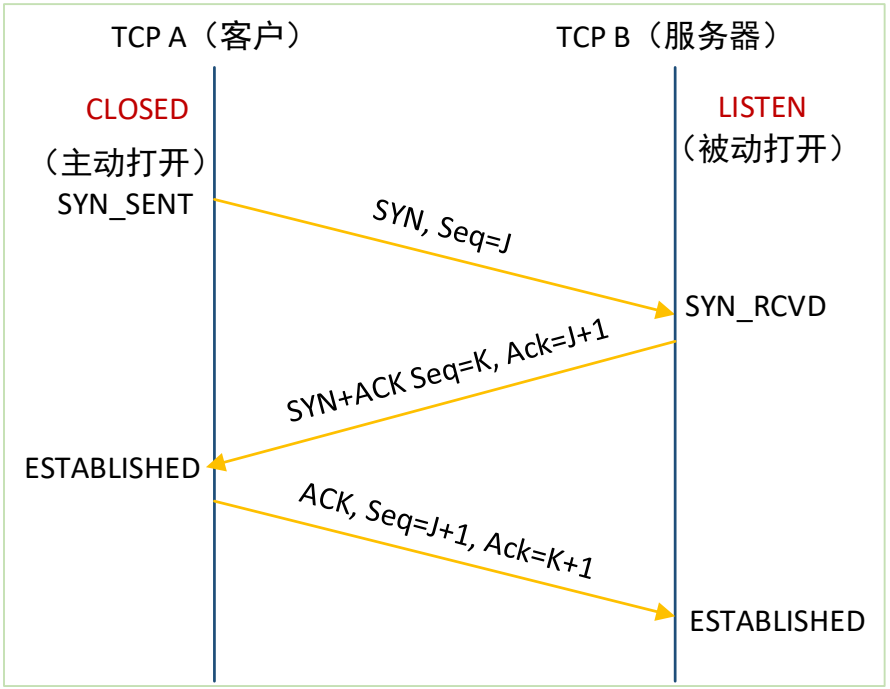

TCP A
TCP B
应用进程
200 bytes buffer
Data, 1-100
Ack 101, win 100
Data, 101-200
Ack 201, win 0
5 bytes
Ack 201, win 5
Data, 201-205
Ack 206, win 0
5 bytes
Ack 201, win 5
Data, 206-210

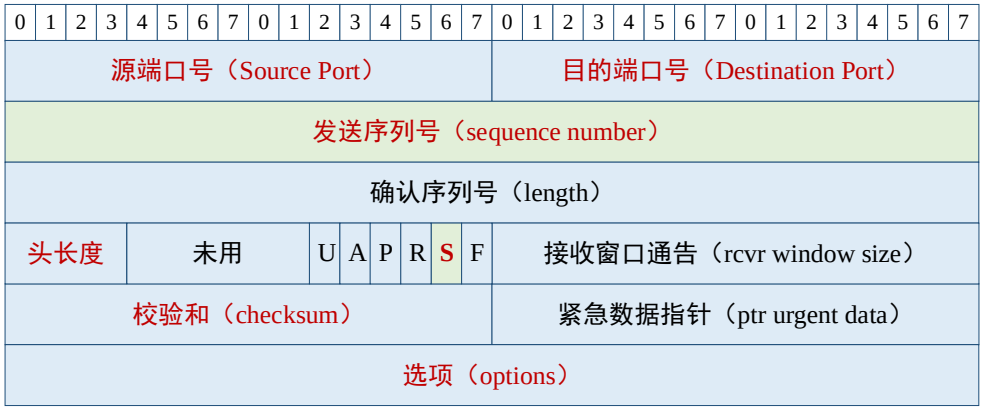
流量控制的性能问题解决策略

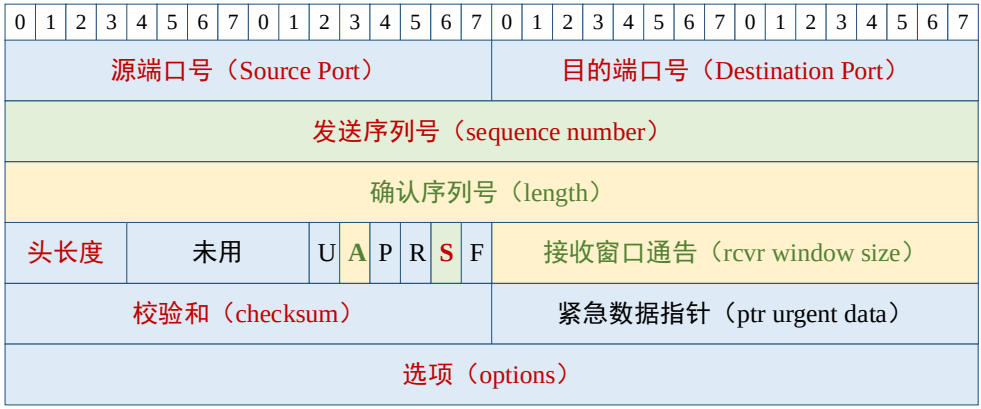
- receiver should not acknowledge when only a small amount of data is removed by application
  - ✓The recommended amount is min (1/2 Receive buffer size, Max segment size)
- receiver can delay the acknowledgement (500ms)
  - ✓TCP acknowledgements can be cumulative
  - ✓ACK can be piggybacked
  - ✓reduce the number of small packets
- Sender can avoid short segments by accumulating data for a while before dispatching

3.5 传输控制协议TCP-连接管理

TCP连接建立





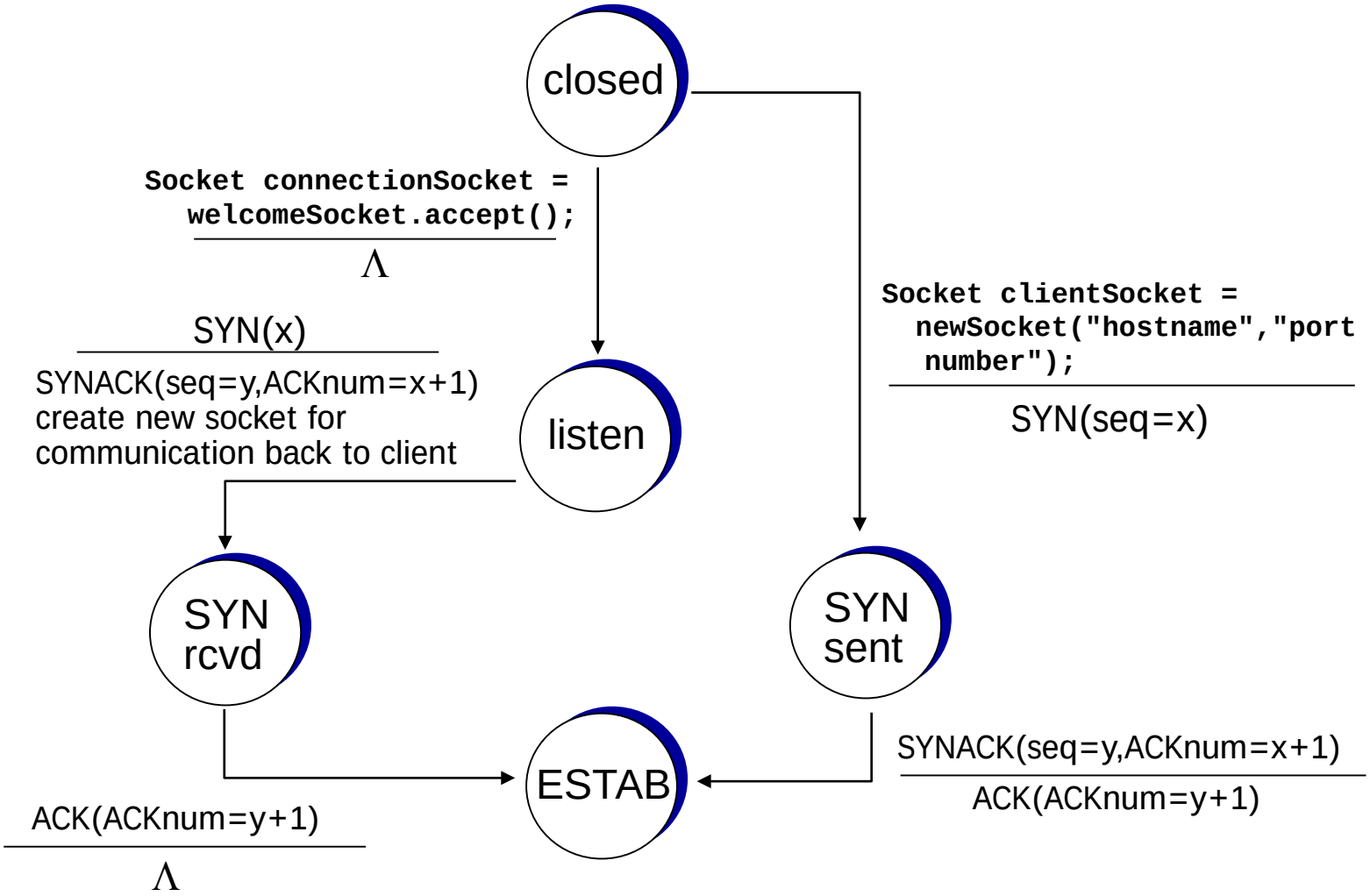


- 步骤一：TCP A端发送SYN段
  - 分配缓冲区，选择A→B初始序列号（ISN， Initial Sequence Number ）
- 步骤二：TCP B接收SYN段，回送SYN+ACK段，
  - 分配缓冲区，选择B→A的初始序列号
- 步骤三：TCP A回送ACK

SYN洪泛攻击？

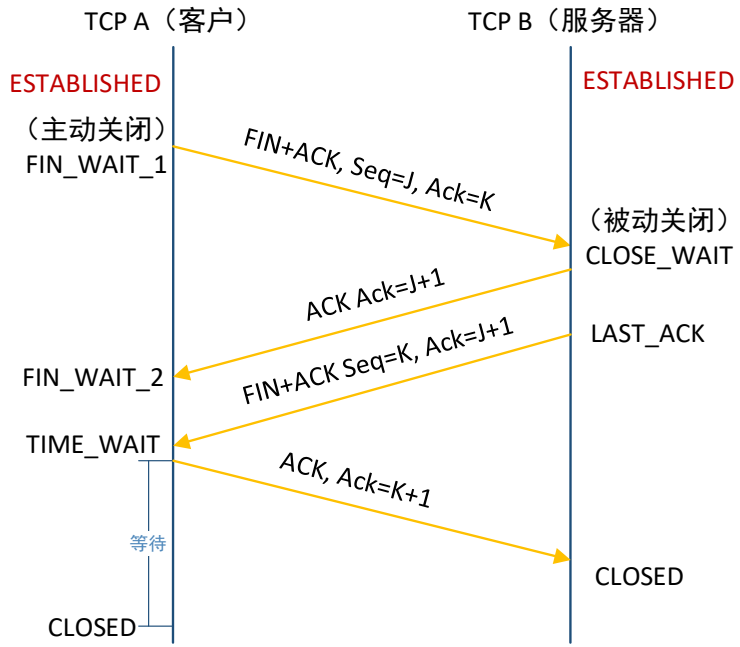
3.5 传输控制协议TCP-连接管理

- 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



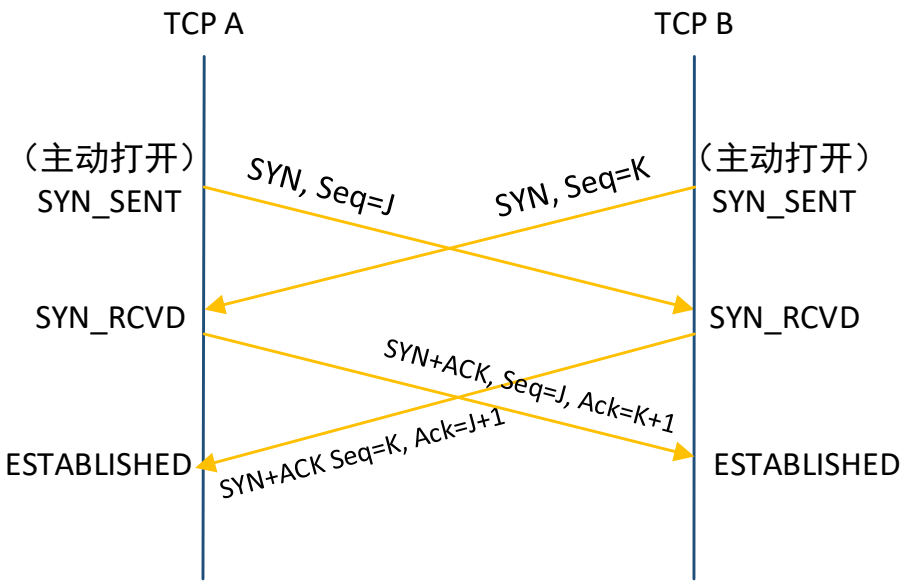
0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7
源端口号（Source Port）		目的端口号（Destination Port）	
发送序列号（sequence number）			
确认序列号（ACK number）			
头长度	未用	U **A** P R S **F**	接收窗口通告（rcvr window size）
校验和（checksum）		紧急数据指针（ptr urgent data）	

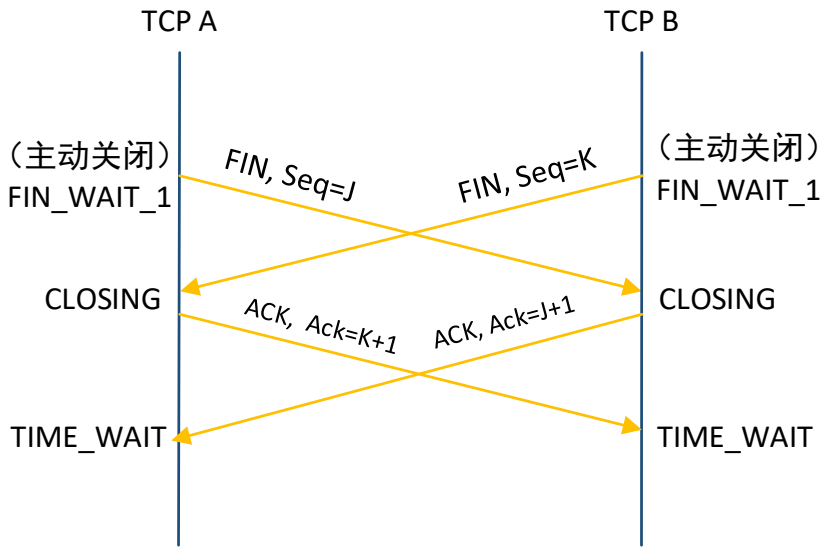
0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7
源端口号（Source Port）		目的端口号（Destination Port）	
发送序列号（sequence number）			
确认序列号（ACK number）			
头长度	未用	U **A** P R S F	接收窗口通告（rcvr window size）
校验和（checksum）		紧急数据指针（ptr urgent data）	

- 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

3.5 传输控制协议TCP-连接管理

- 同时打开与同时关闭连接





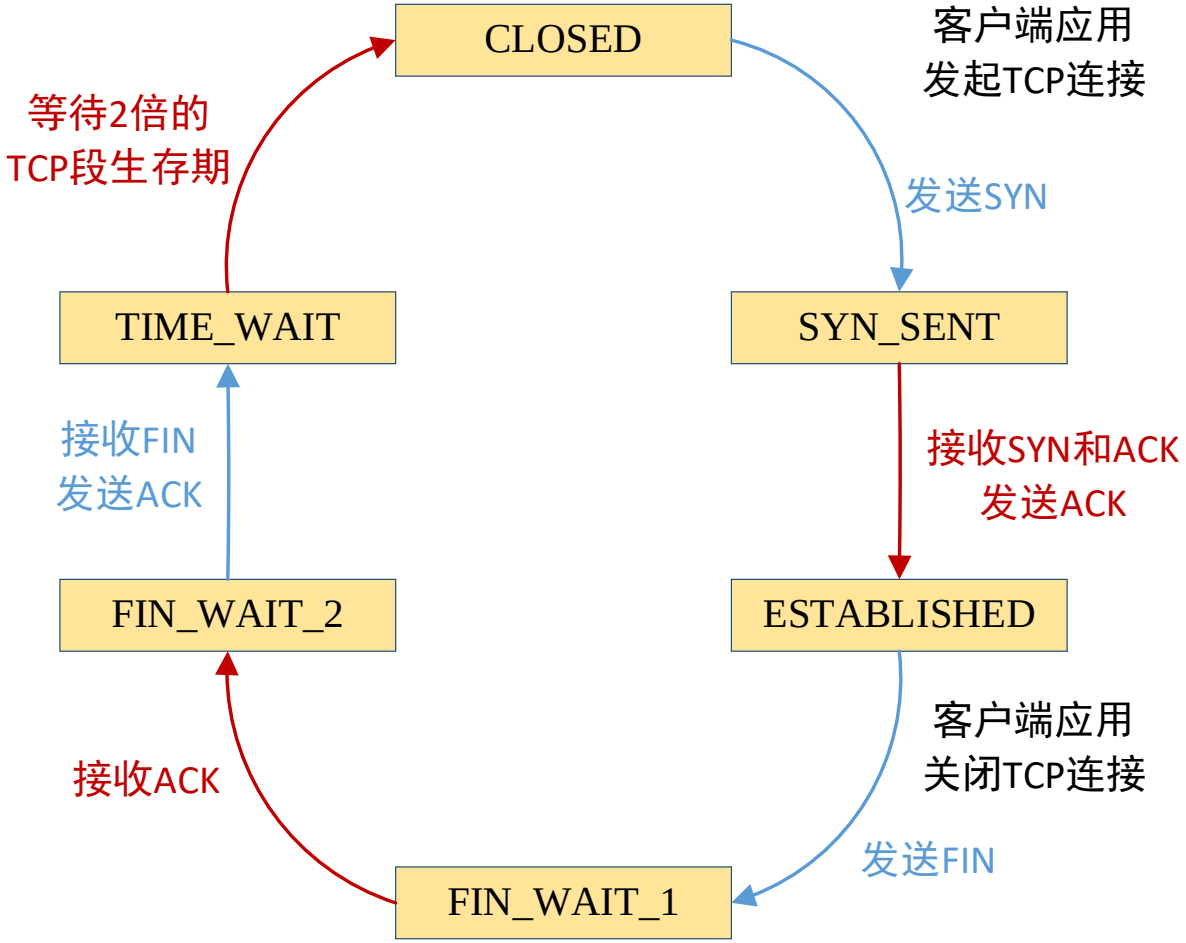
- 连接的半打开状态：连接的一端存在、而另一端不存在
  - ✓ 当一个进程终止连接未能通知到另一方时，例如：掉电、异常关闭等
  - ✓ 如何解决？

TCP定时器:

- Connection Establishment Timer
- Retransmission Timer
- Delayed ACK Timer
- Persistence Timer
- The Keepalive Timer
- The Quiet Timer

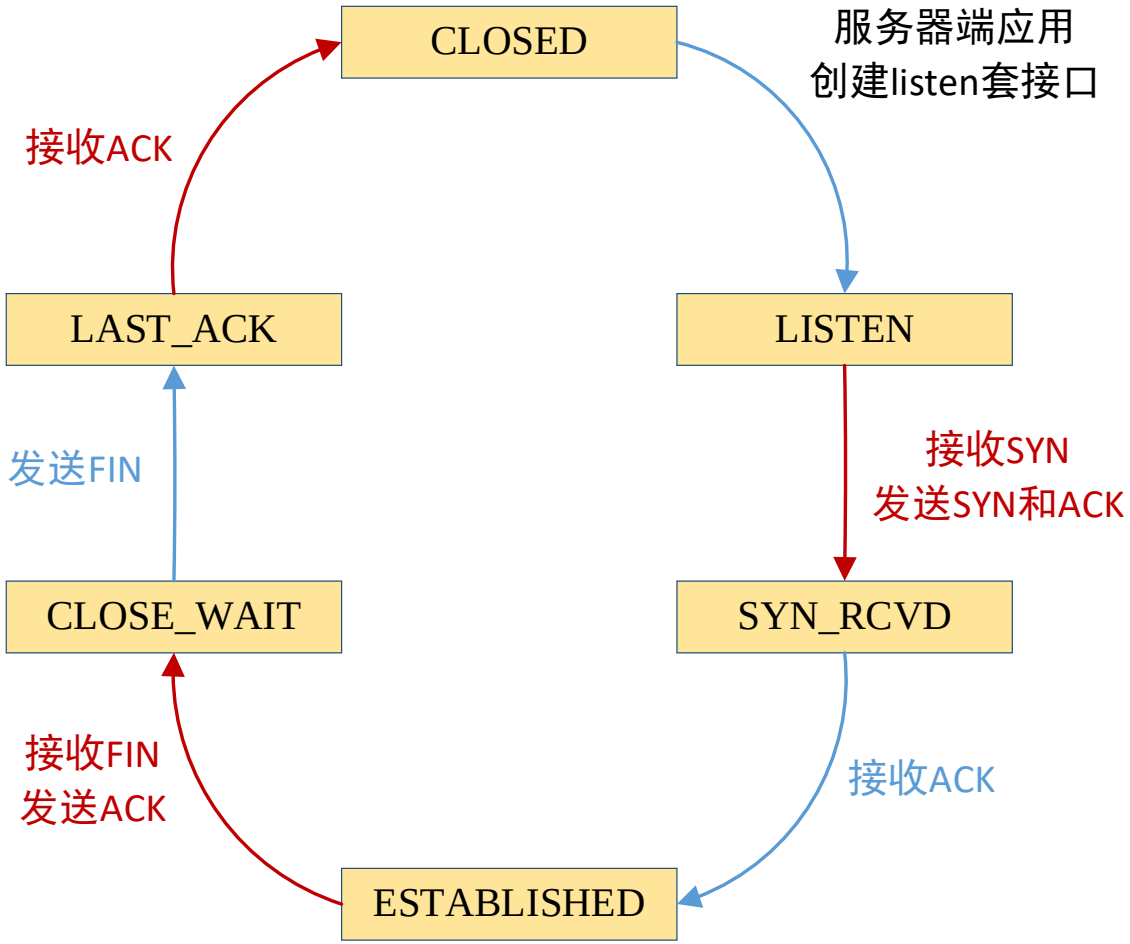
3.5 传输控制协议TCP-连接管理

- TCP客户端生命周期



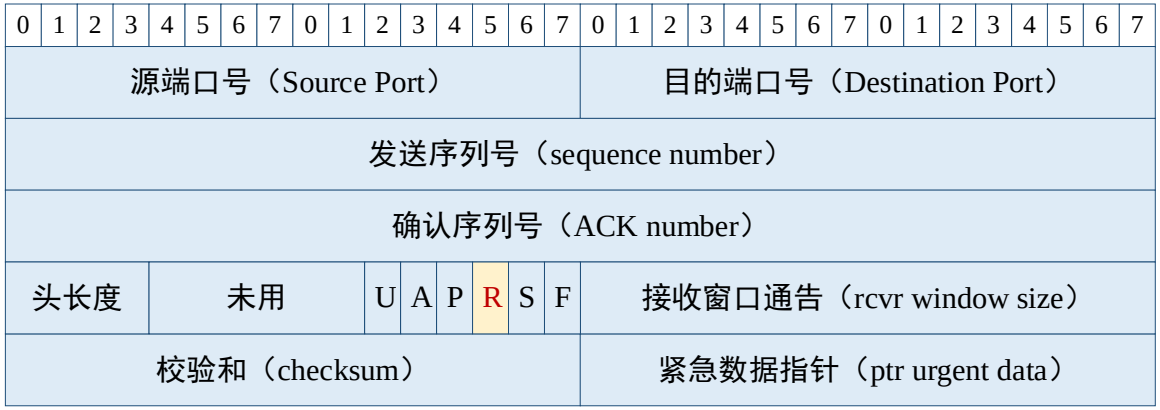
3.5 传输控制协议TCP-连接管理

- 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段
- ✓ 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



3.6 理解网络拥塞

- 拥塞控制
  - ✓too many sources sending too much data too fast for ***network*** to handle
  - ✓different from flow control!
  - ✓manifestations:
    - long delays (queueing in router buffers)
    - lost packets (buffer overflow at routers)
  - ✓a top-10 problem!

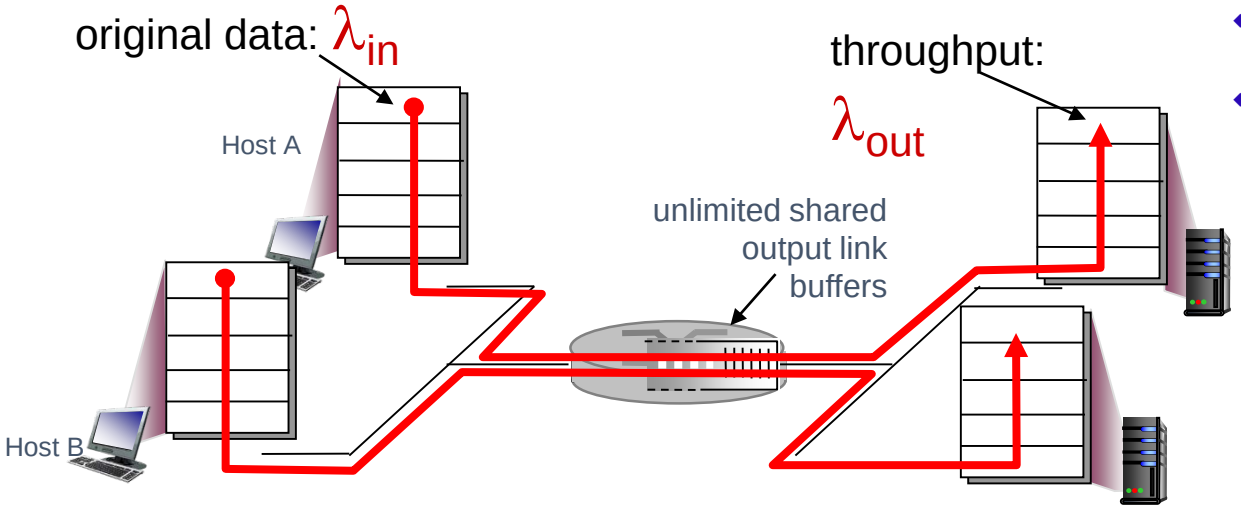
3.6 理解网络拥塞

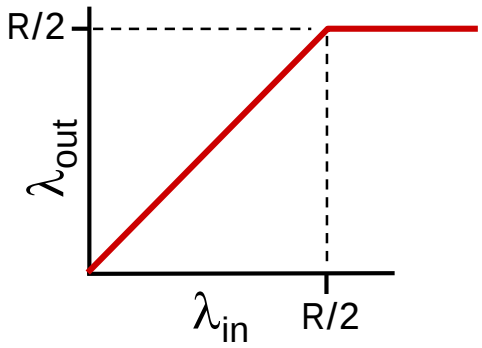
- 分交换网络中拥塞不可避免
  - ✓We use packet switching because it makes efficient use of the links. Therefore, buffers in the routers are frequently occupied
  - ✓If buffers are always empty, delay is low, but our usage of the network is low
  - ✓If buffers are always occupied, delay is high, but we are using the network more efficiently

Q: how much congestion is too much?

拥塞的代价：情景1

- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no retransmission

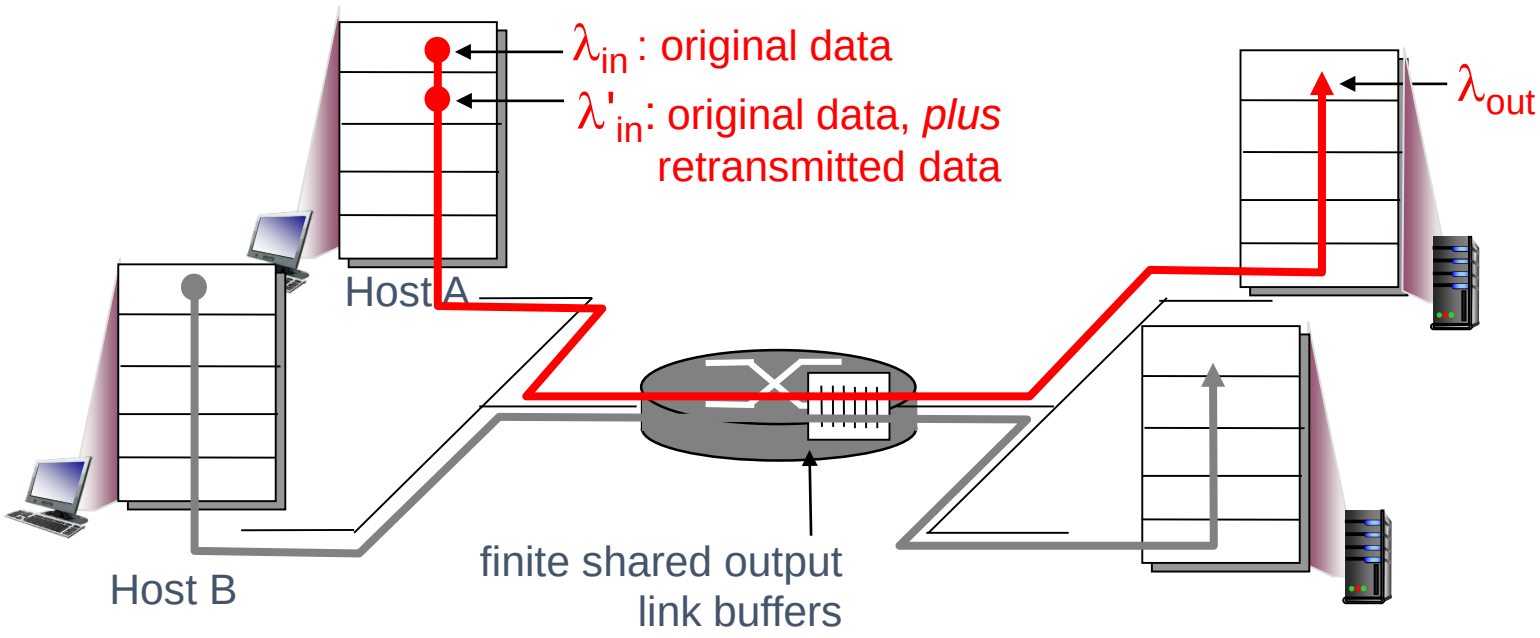




- maximum per-connection throughput: R/2
- large delays as arrival rate, λ_{in}, increases

- 拥塞的代价：情景2

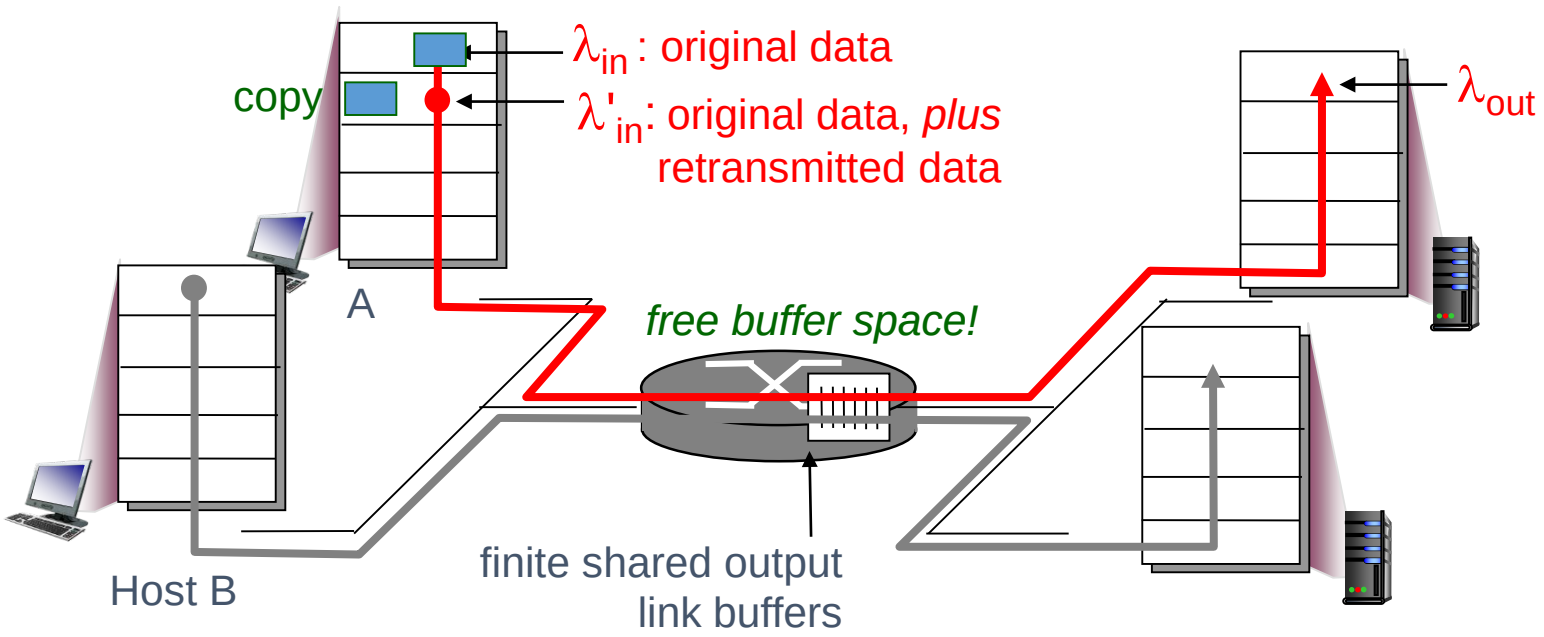
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$



- 拥塞的代价：情景2（续）

Idealization: perfect knowledge

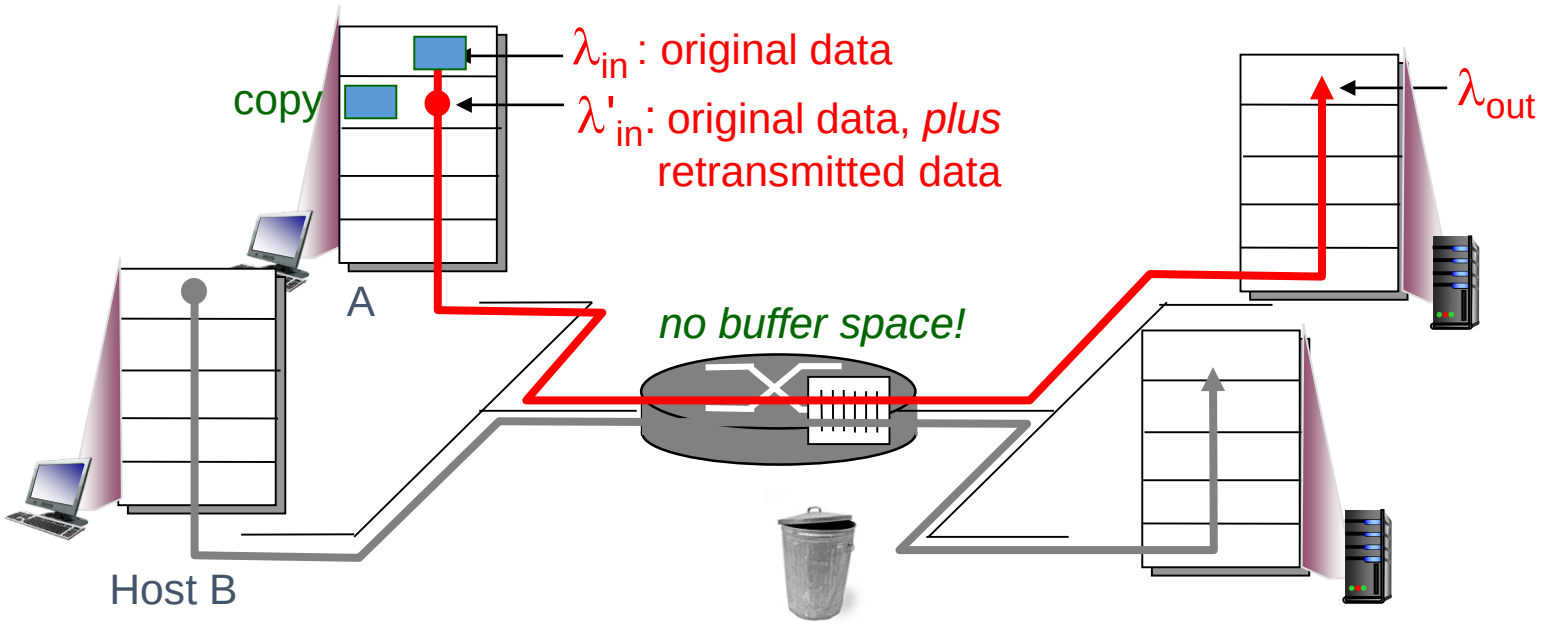
- sender sends only when router buffers available



- 拥塞的代价：情景2（续）

Idealization: known loss

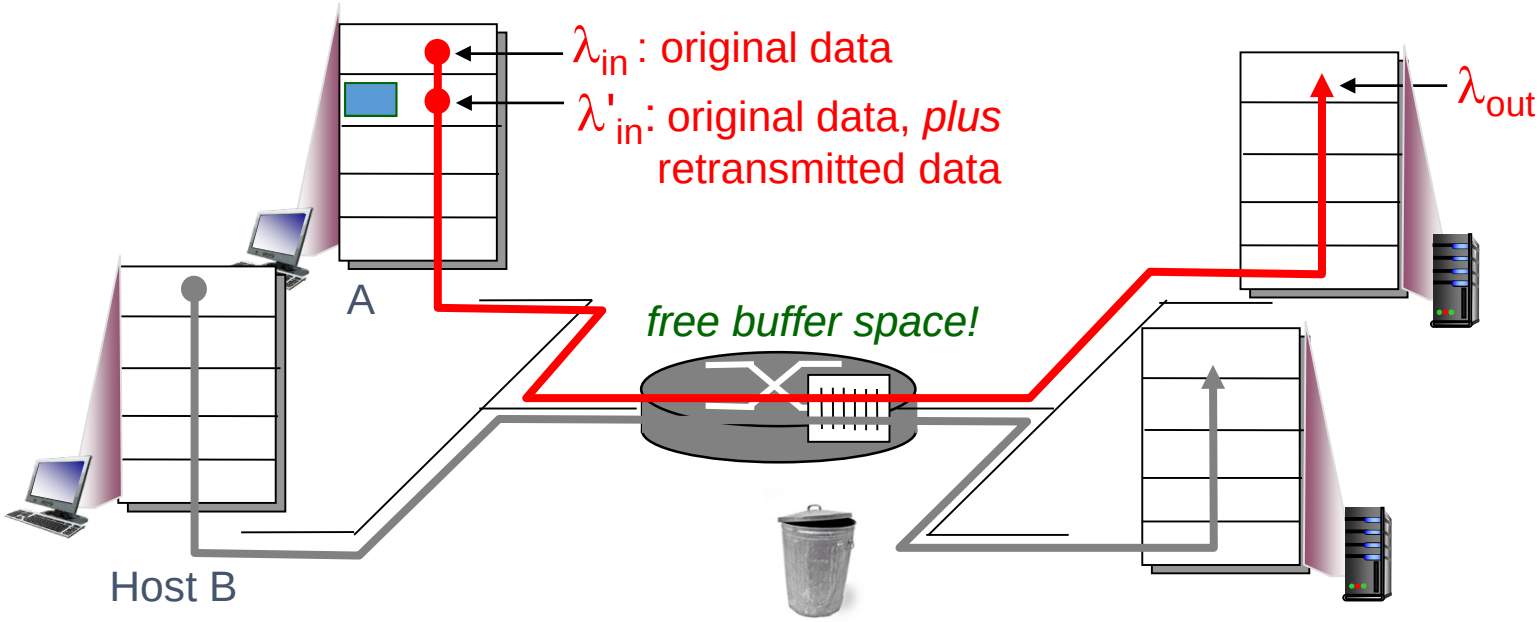
- packets can be lost, dropped at router due to full buffers
- sender only resends if packet *known* to be lost



- 拥塞的代价：情景2（续）

Idealization: known loss

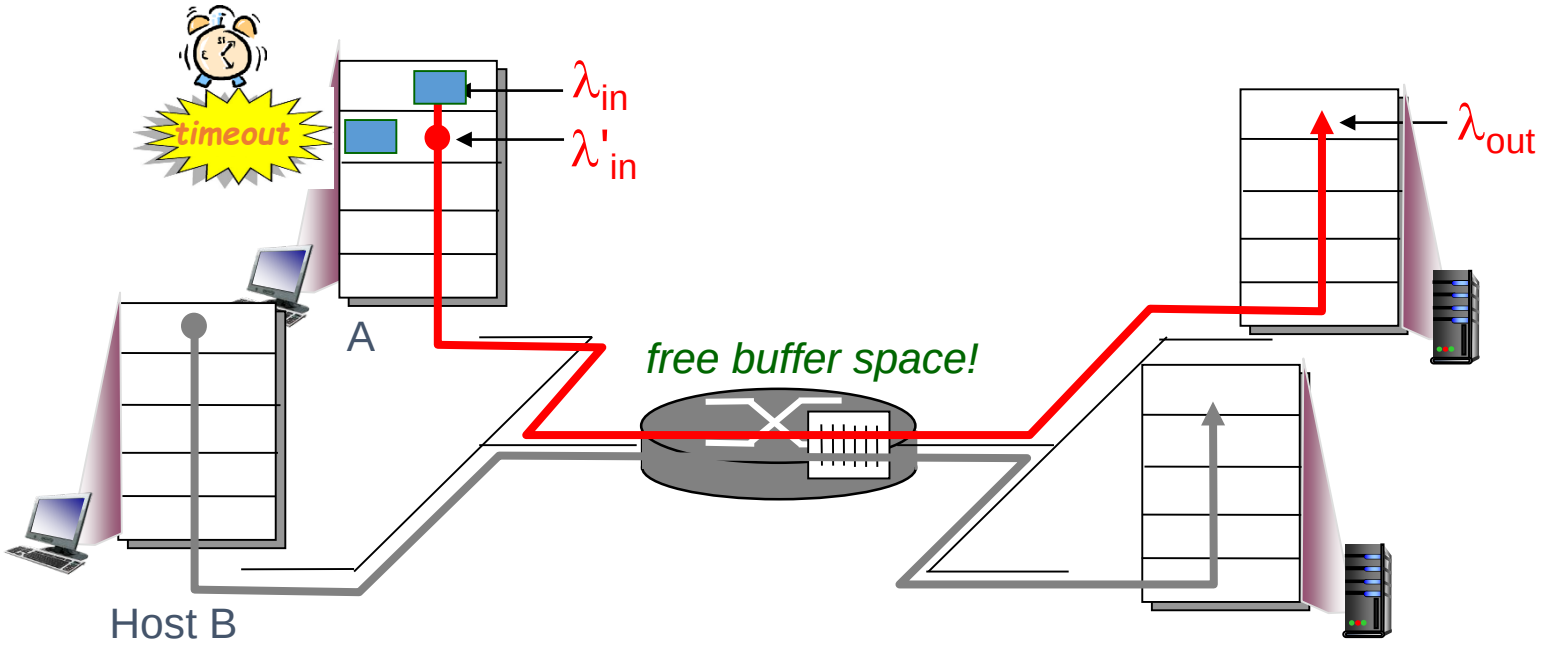
- packets can be lost, dropped at router due to full buffers
- sender only resends if packet *known* to be lost



- 拥塞的代价：情景2（续）

Realistic: duplicates

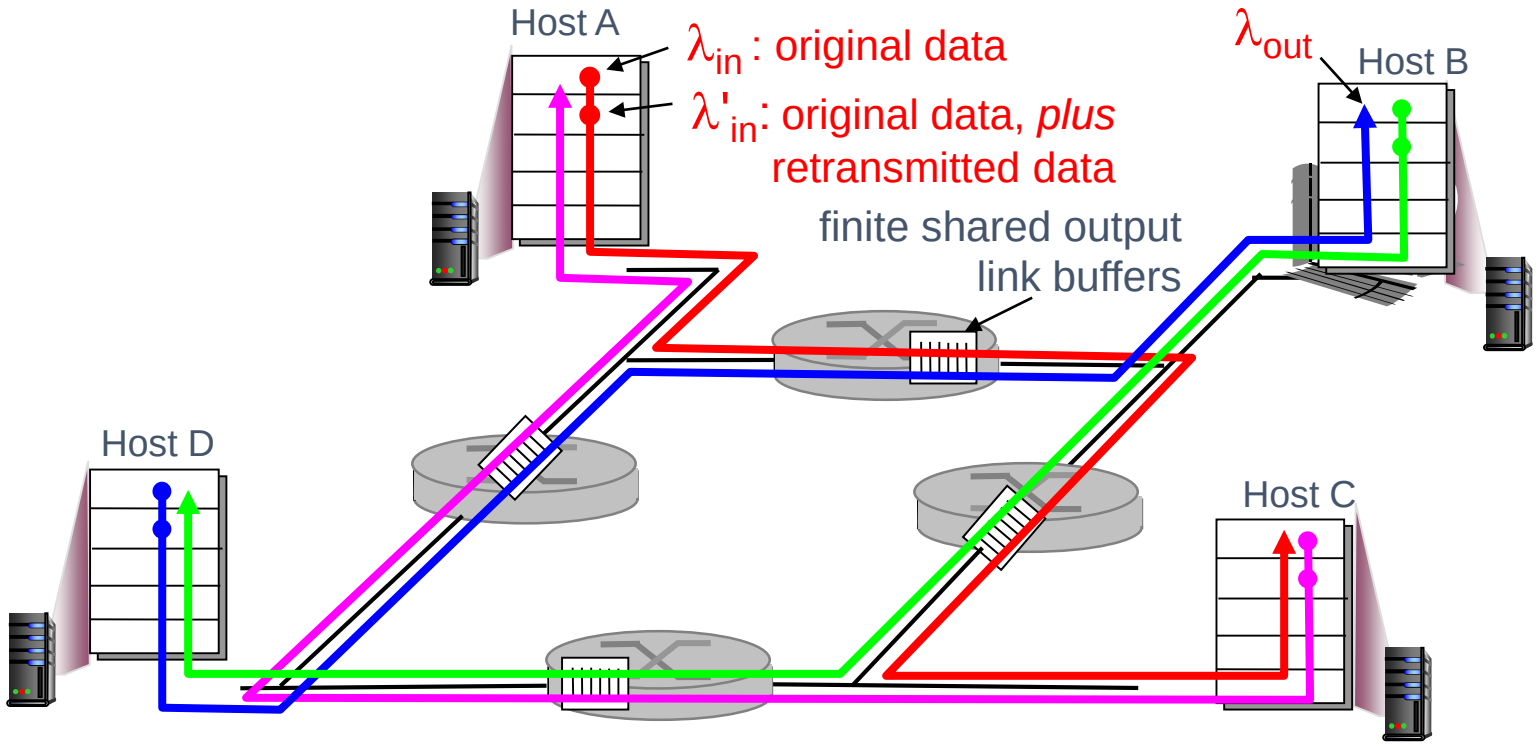
- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered



- 拥塞的代价：情景3
  - four senders
  - multihop paths
  - timeout/retransmit

Q: what happens as λ_{in} and λ_{in}' increase ?

A: as red λ_{in}' increases, all arriving blue pkts at upper queue may be dropped, blue throughput → 0



- 拥塞控制方法

Two broad approaches towards congestion control:

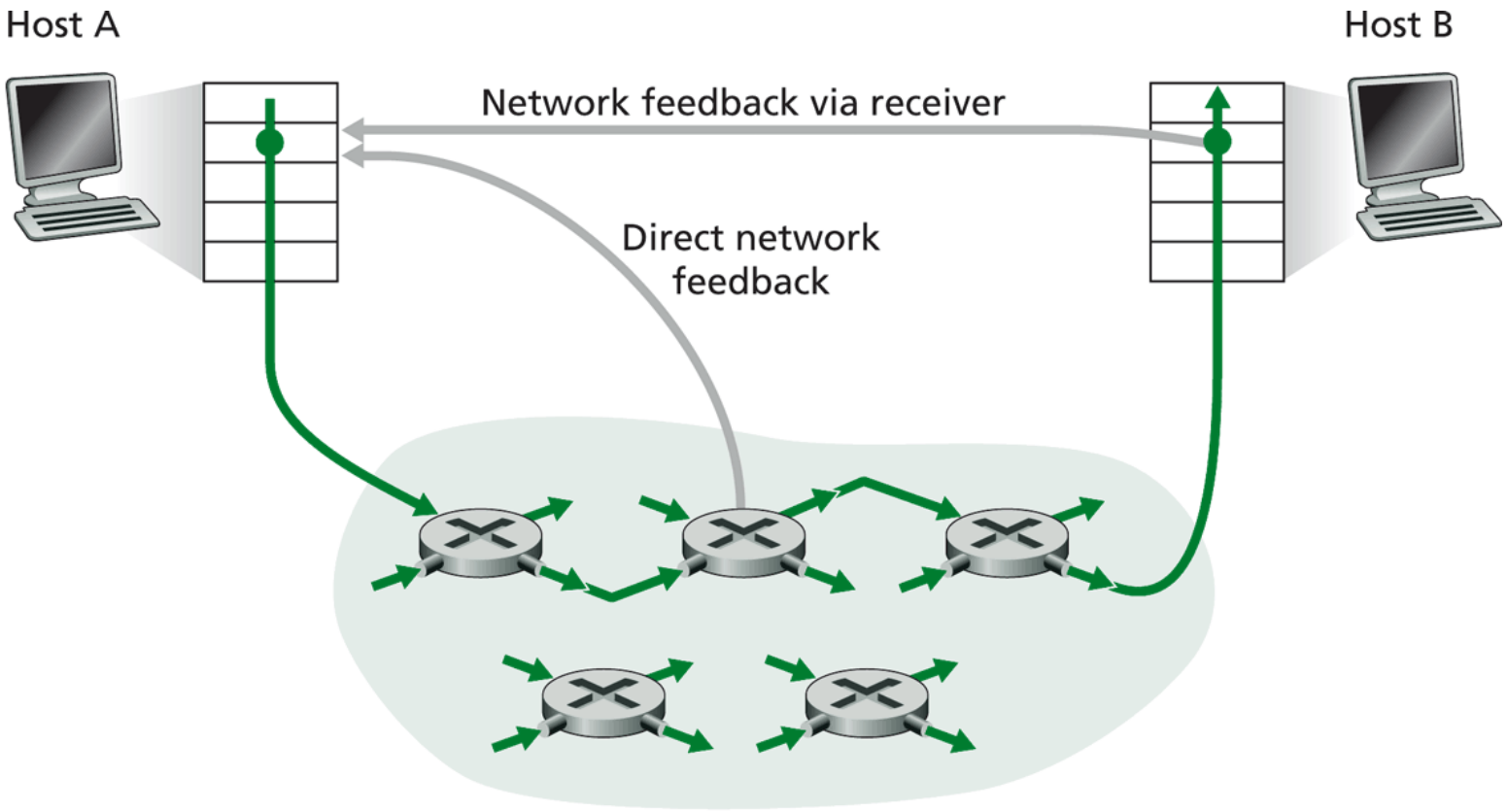
end-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken **by TCP**

network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (X.25, ATM)
  - explicit rate for sender to send at

Network-Assisted Congestion Control

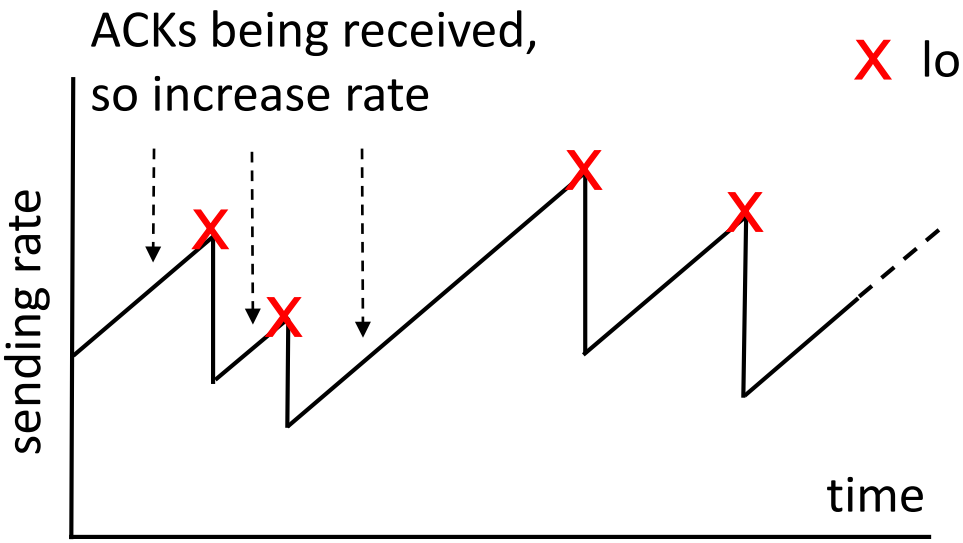


Two feedback pathways for network-indicated congestion information

- *Goal:* TCP sender should transmit as fast as possible, but without congesting network
  - Q: how to find rate *just* below congestion level
- Decentralized solution: each TCP sender sets its own rate, based on *implicit* feedback:
  - *ACK:* segment received (a good thing!), network not congested, so increase sending rate
  - *lost segment:* **assume loss due to congested network**, so decrease sending rate

TCP congestion control: bandwidth probing

- "probing for bandwidth": increase transmission rate on receipt of ACK, until eventually loss occurs, then decrease transmission rate



- Q: how fast to increase/decrease?

TCP Congestion Control: congestion window

- TCP implements window-based congestion control
- Transmission rate limited by congestion window size, **cwnd**

  Window Size= min{Advertized window, Congestion Window}

  Advertized window = rcvwin; Congestion Window = cwnd

- In other words, send at the rate of the slowest component: network or receiver

TCP Congestion Control: slow start, congestion avoidance

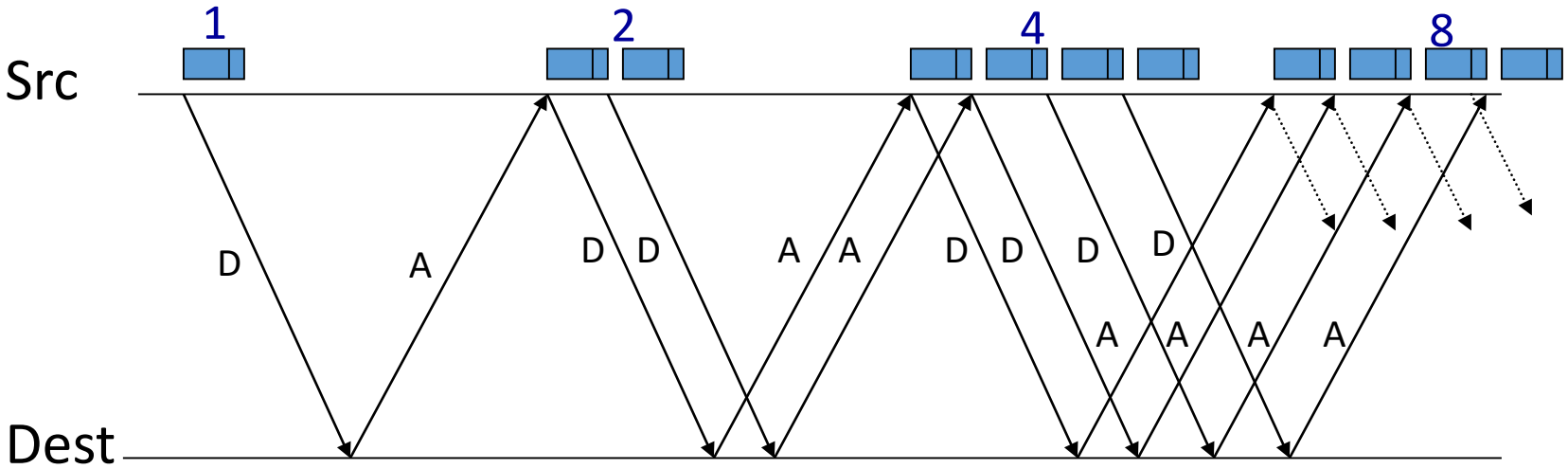
- slow start
  - **cwnd** is set to 1 (1 MSS) at start
  - **cwnd** increase window by 1 per ACK
- congestion avoidance : **cwnd** follows “Additive Increase, Multiplicative Decrease (AIMD)”
  - **cwnd** increase window by 1 per RTT
  - **cwnd** decrease window by factor of 2 on loss event
- important variables:
  - **cwnd**
  - **threshold(ssthresh) :** defines threshold between slow start phase, congestion control phase

TCP Slowstart

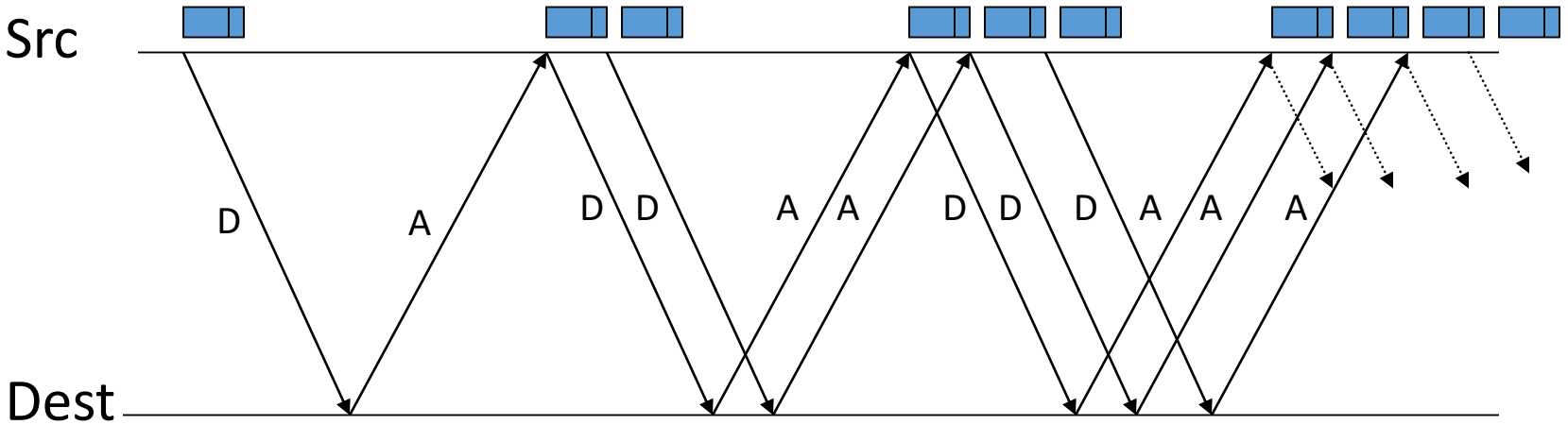
Slowstart algorithm

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

- exponential increase (per RTT) in window size (not so slow!)
- **loss event**: timeout (Tahoe TCP) and/or three duplicate ACKs (Reno TCP)



Additive Increase



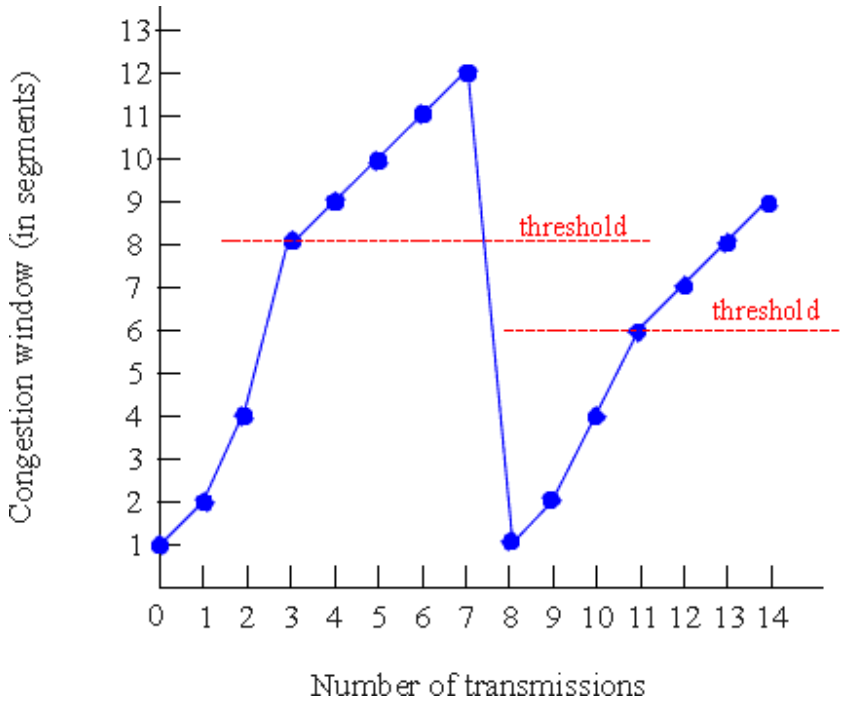
Note: Actually, TCP uses bytes, not segments to count, When ACK is received:

cwnd=cwnd+MSS(MSS/cwnd)

TCP Congestion Avoidance: Tahoe

TCP Tahoe Congestion avoidance

```
/* slowstart is over    */
/* cwnd >= ssthresh  */
Until (loss event) {
  every w segments ACKed:
        cwnd ++
  }
ssthresh = cwnd /2
cwnd = 1
perform slowstart
```



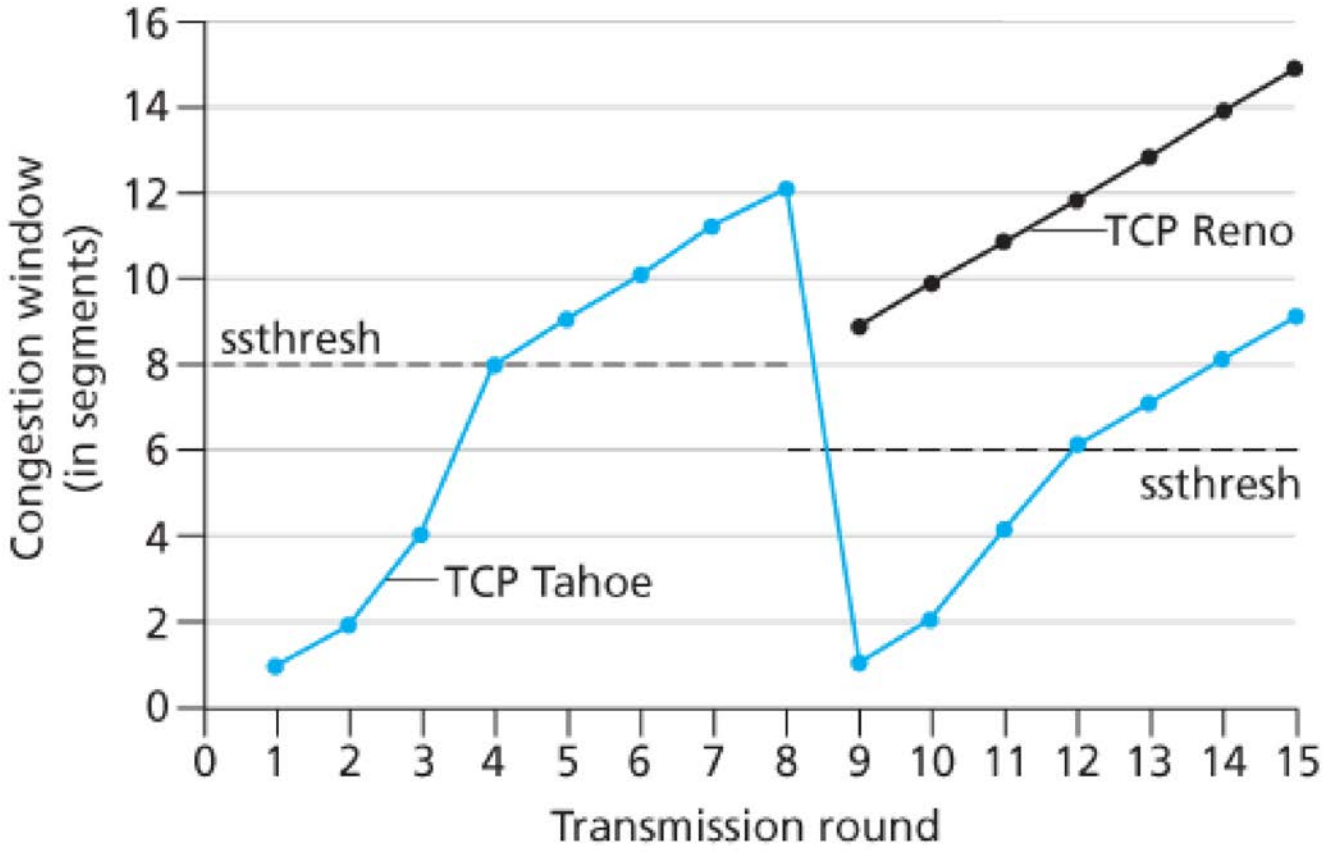
TCP Congestion Avoidance: Reno

- three duplicate ACKs (Reno TCP)
- some segments are getting through correctly!
- don't "overreact" by decreasing window to 1 as in Tahoe
  - decrease window size by half

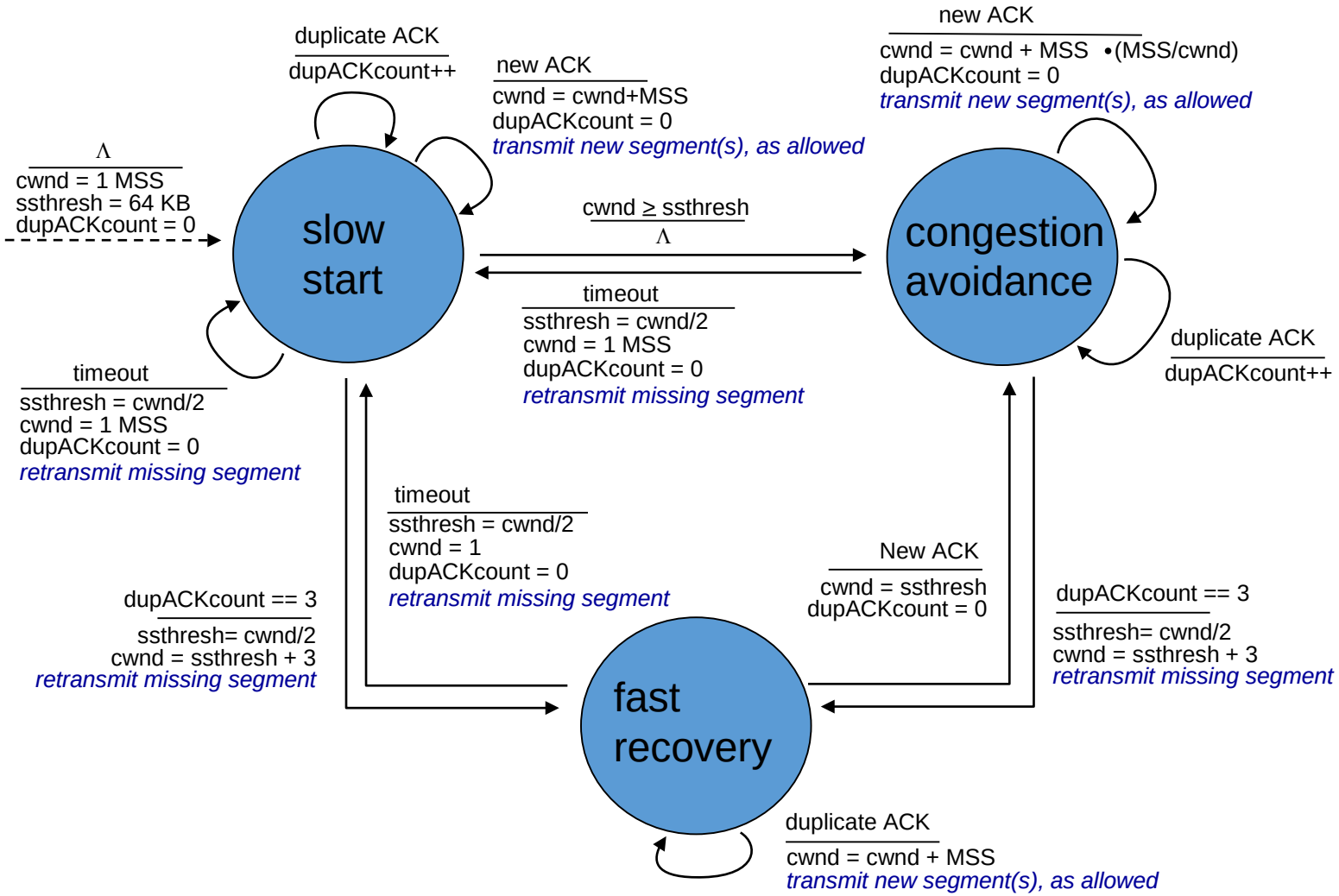
TCP Reno Congestion avoidance

```
/* 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 Reno versus TCP Tahoe



Summary: TCP Congestion Control



Summary

- principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet
  - UDP
  - TCP

Next:

- leaving the network "edge" (application, transport layers)
- into the network "core"