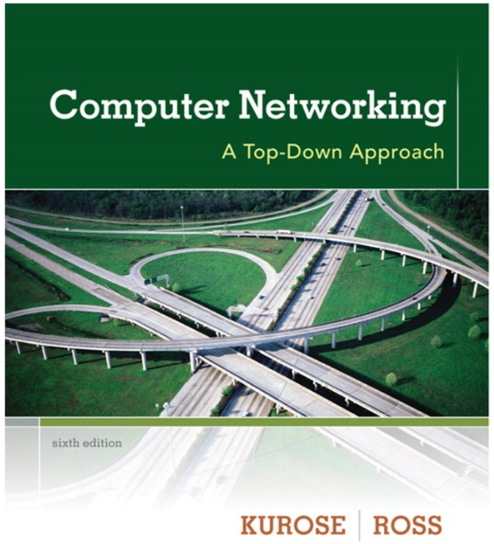
第三章 传输层



Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

注：本PPT 来源于下面的资料，并有所修改。

第三章 传输层

教学目的和要求：

- 理解传输层原理：
  - 多路复用
  - 可靠数据传输
  - 流控制
  - 拥塞控制
- 掌握因特网传输层协议：
  - UDP：无连接的不可靠传输协议
  - TCP：面向连接的可靠传输协议
  - TCP 拥塞控制机制

基本内容

3.1 传输层服务概念

3.2 多路复用

3.3 UDP协议

3.4 可靠传输原理

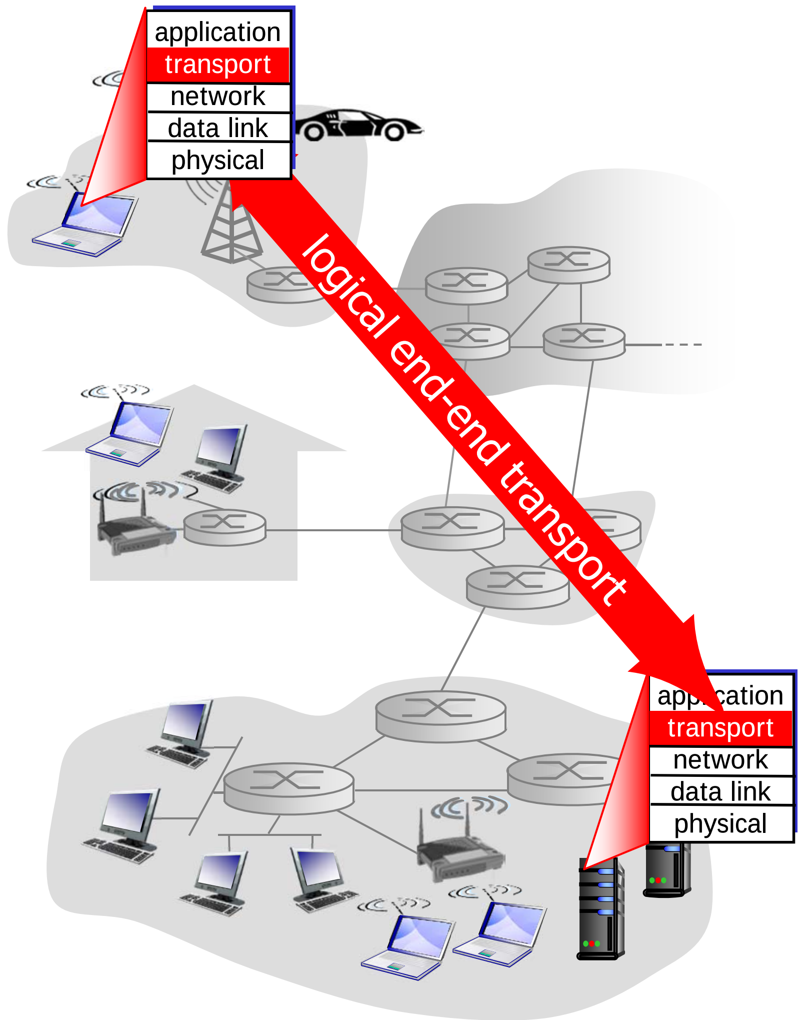
3.5 TCP协议
- segment 结构
- 可靠传输
- 流控制
- 连接管理

3.6 拥塞控制原理

3.7 TCP 拥塞控制机制

3.1 传输层服务和协议

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems （端到端协议）
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



传输层与网络层的区别

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

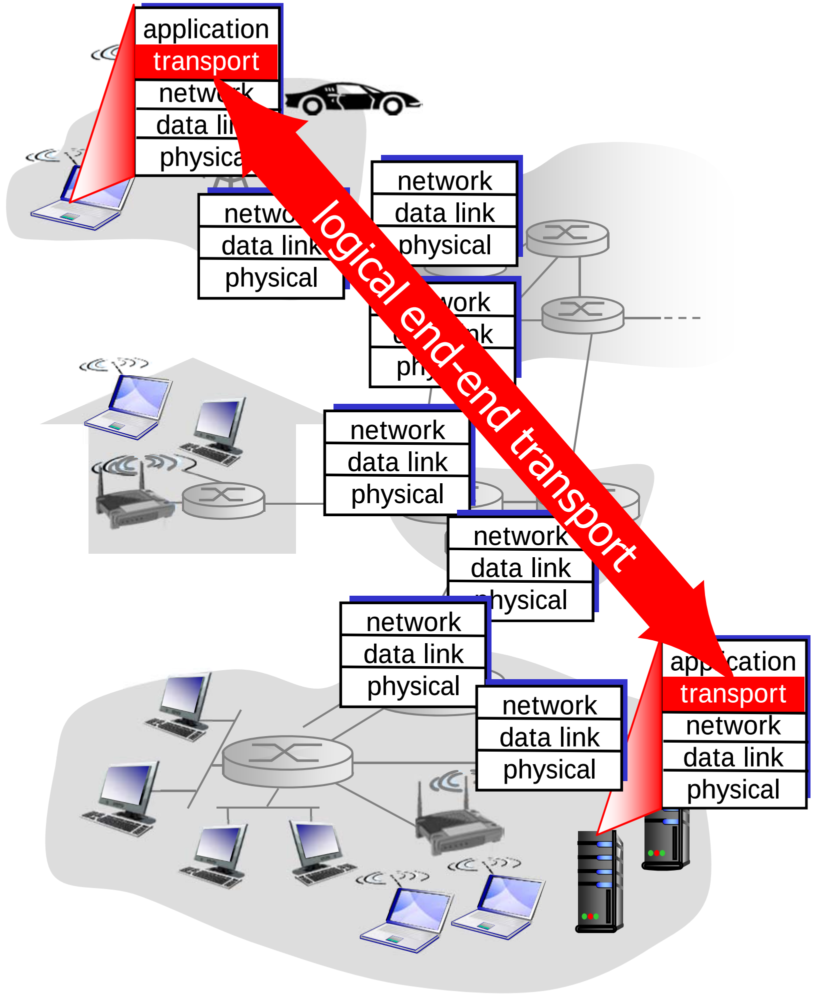
household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

因特网传输层协议

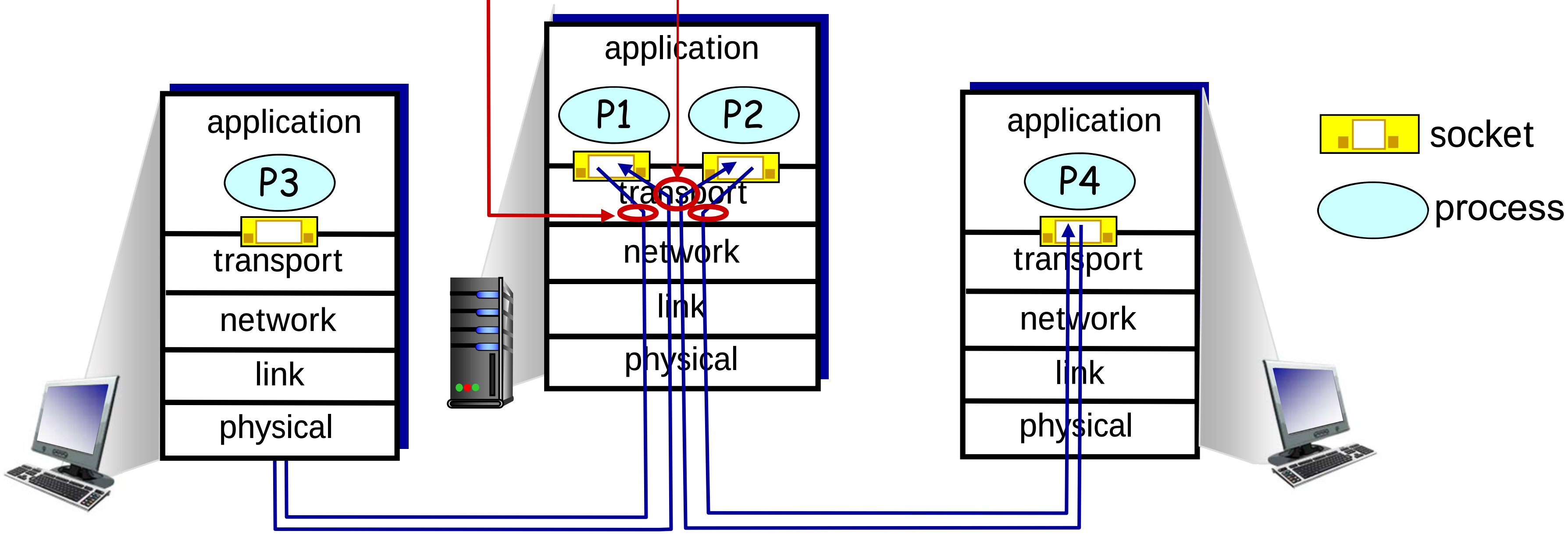
- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



3.2 多路复用/分解技术

multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:
use header info to deliver received segments to correct socket

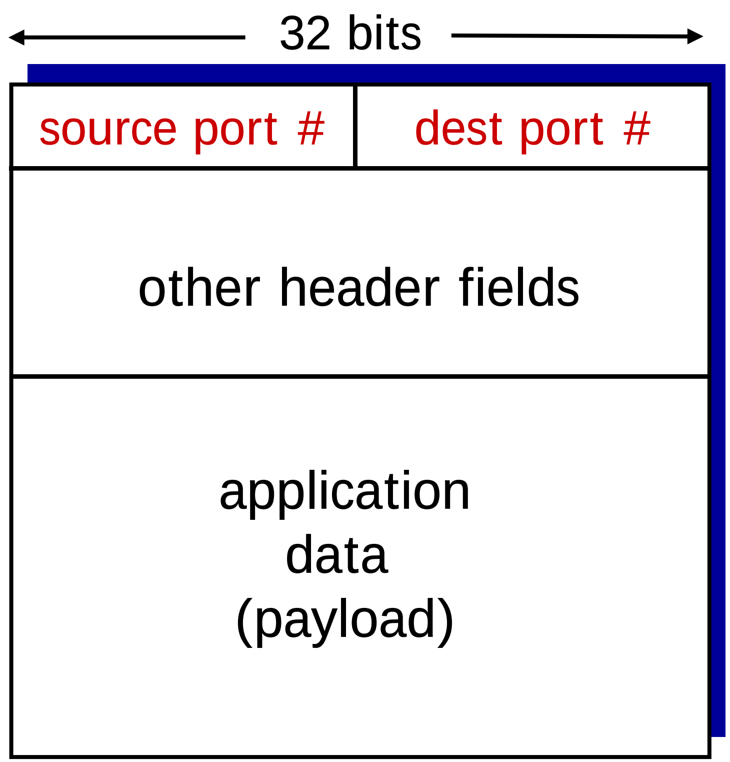


复用：从套接字中收集到数据并封装成传输层的SEGMENT

分解：将传输层SEGMENT定向到适当的套接字

如何多路分解

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

无连接多路分解： UDP 使用目的主机和端口号二元组定向的套接字

- *recall:* created socket has host-local port #:

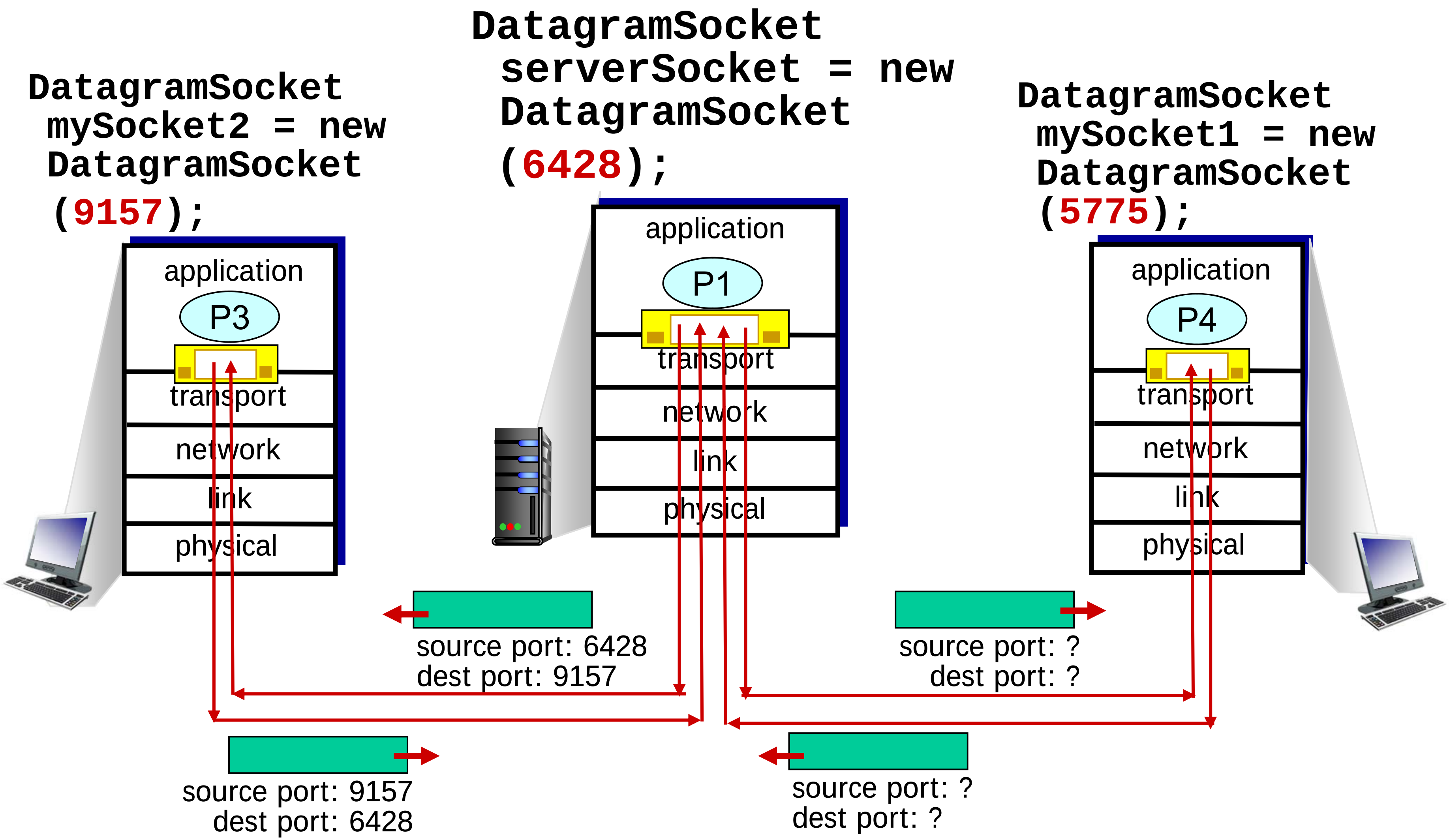
```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

- *recall:* when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

UDP 举例（java）



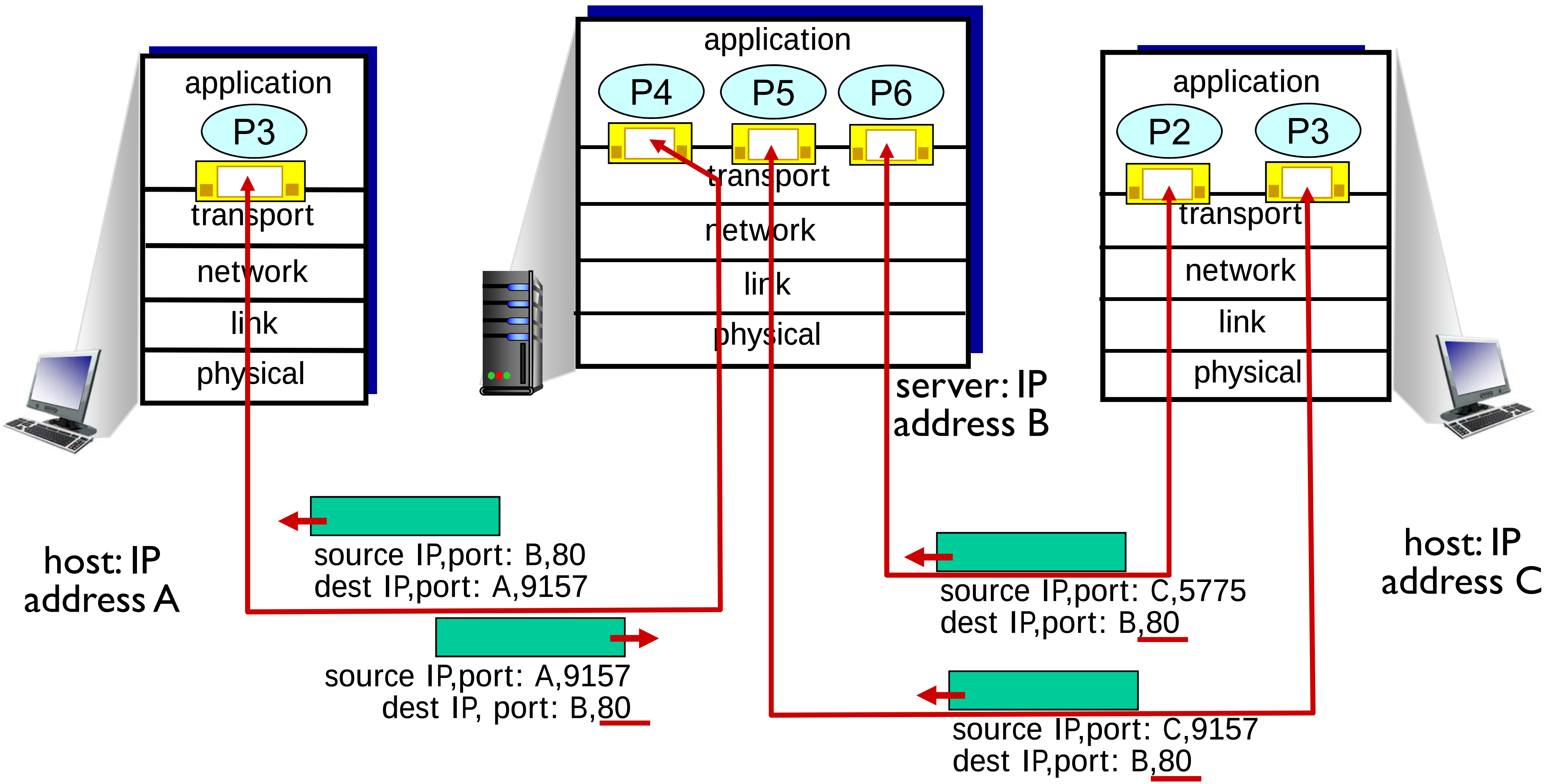
DatagramSocket mySocket2 = new DatagramSocket (9157);
DatagramSocket serverSocket = new DatagramSocket (6428);
DatagramSocket mySocket1 = new DatagramSocket (5775);
application
P3
transport
network
link
physical
application
P1
transport
network
link
physical
application
P4
transport
network
link
physical
source port: 6428
dest port: 9157
source port: ?
dest port: ?
source port: 9157
dest port: 6428
source port: ?
dest port: ?

面向连接的多路分解：TCP采用源目IP地址和端口号四元组定向套接字

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket

- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

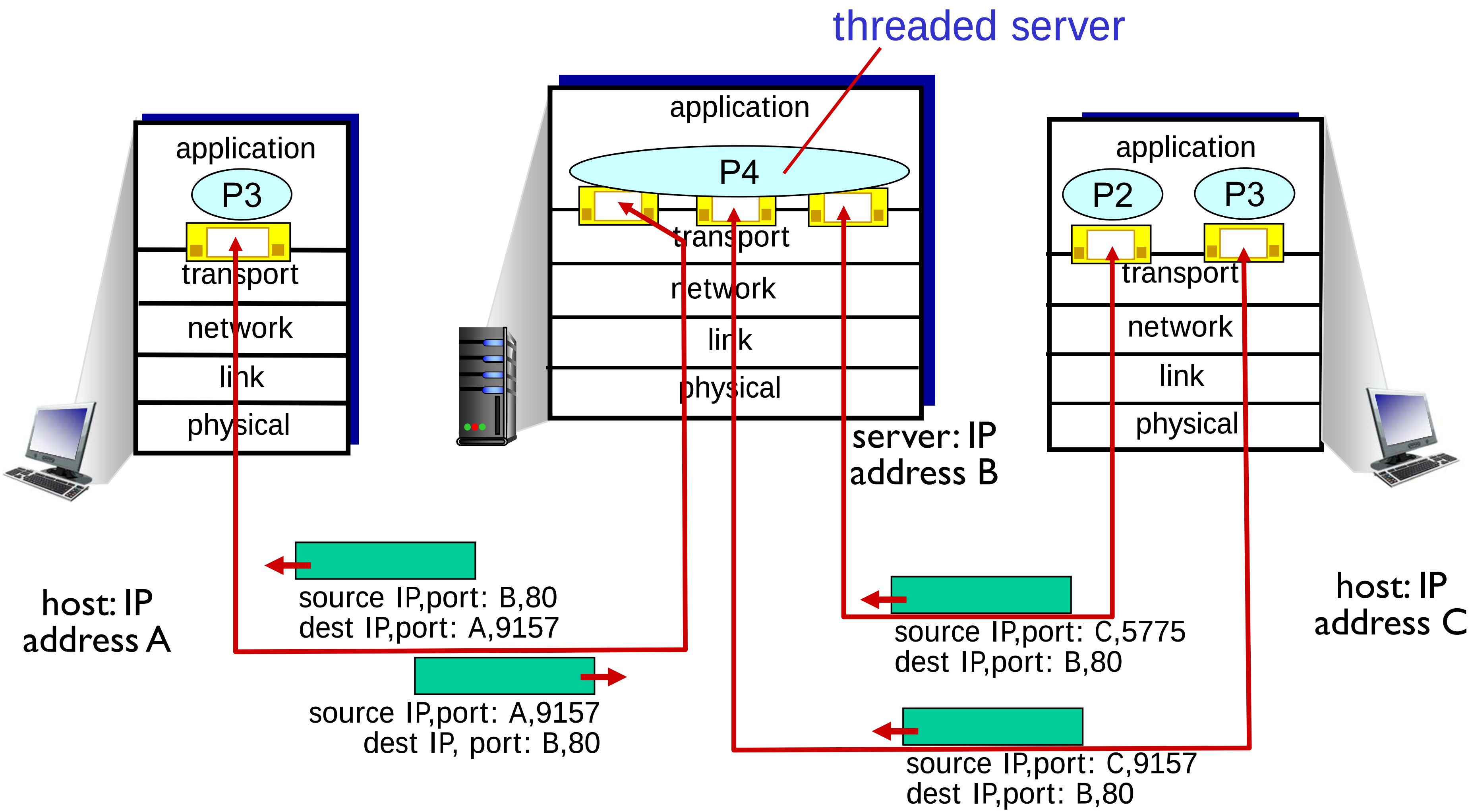
TCP举例：



three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to *different* sockets

TCP 举例

线程服务器：一个进程对应多个套接字
减少建立进程的开销

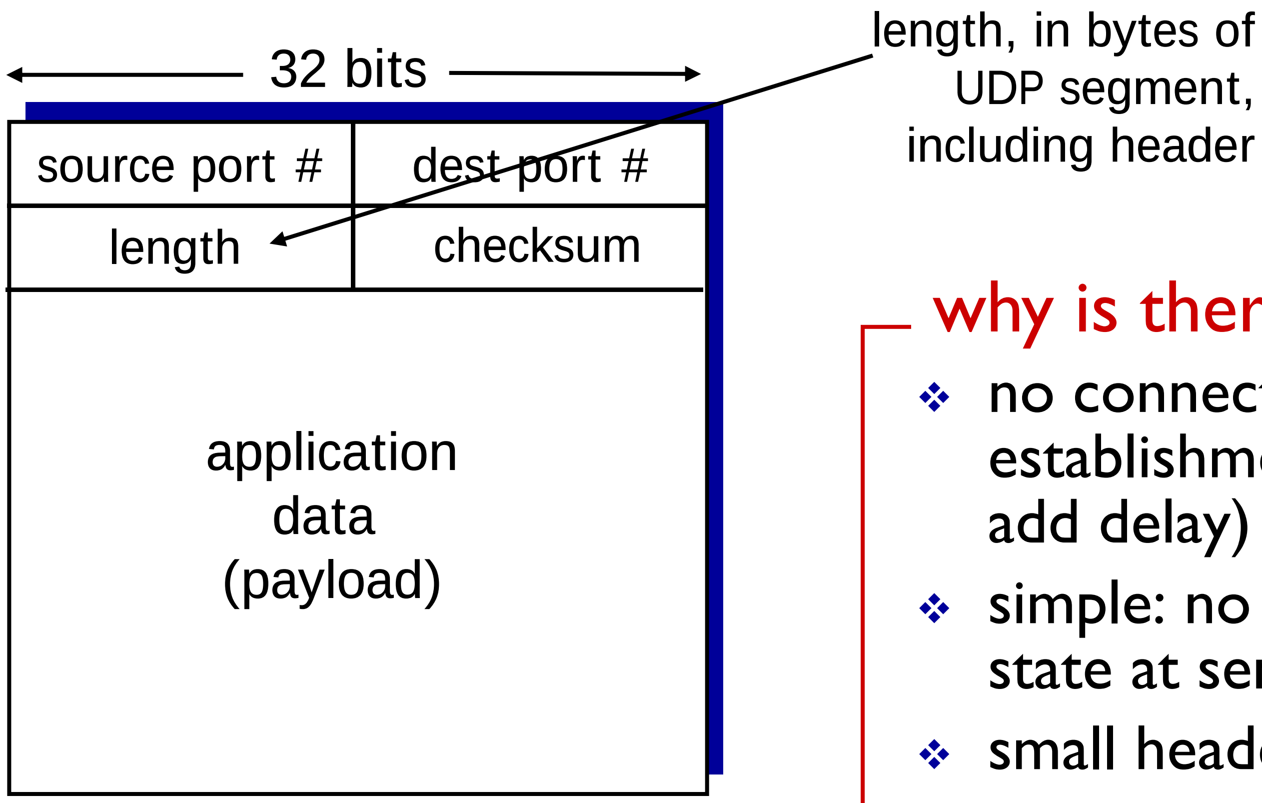


3.3 UDP: User Datagram Protocol

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking 握手 between UDP sender, receiver
  - each UDP segment handled independently of others

- UDP use:
  - streaming multimedia（流媒体）apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP: segment 格式



UDP segment format

why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired

UDP checksum

Goal: detect “errors” (e.g., flipped bits) in transmitted segment

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one’s complement sum) of segment contents
- sender puts checksum value into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ….

校验和举例：（ 模2补码）

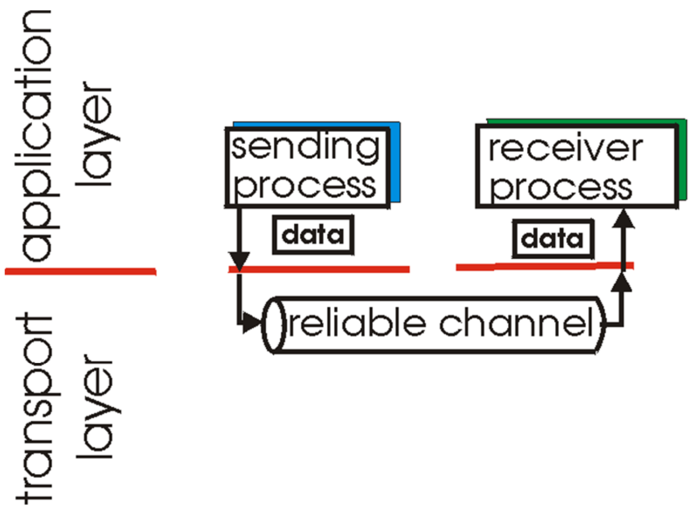
example: add two 16-bit integers

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

3.4 可靠传输原理

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

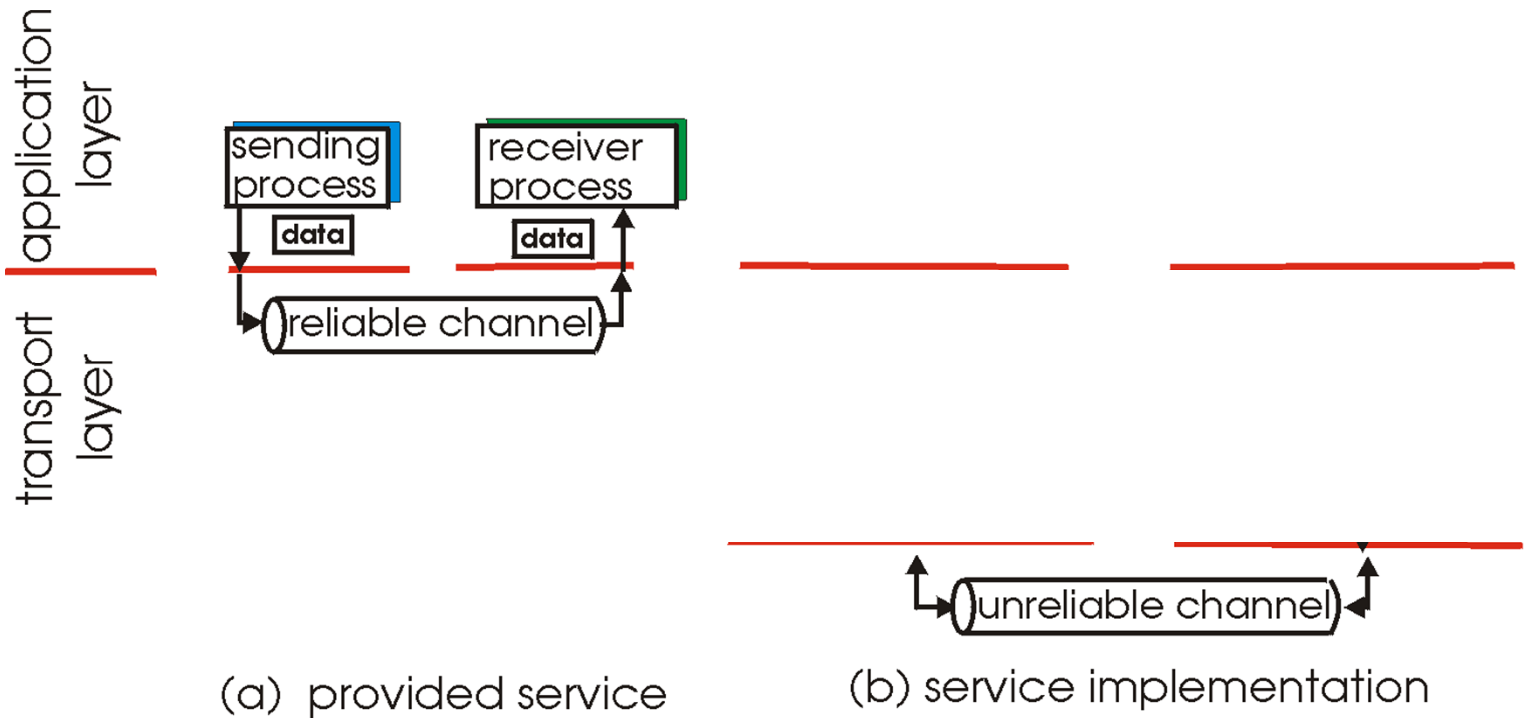


(a) provided service

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

可靠传输原理

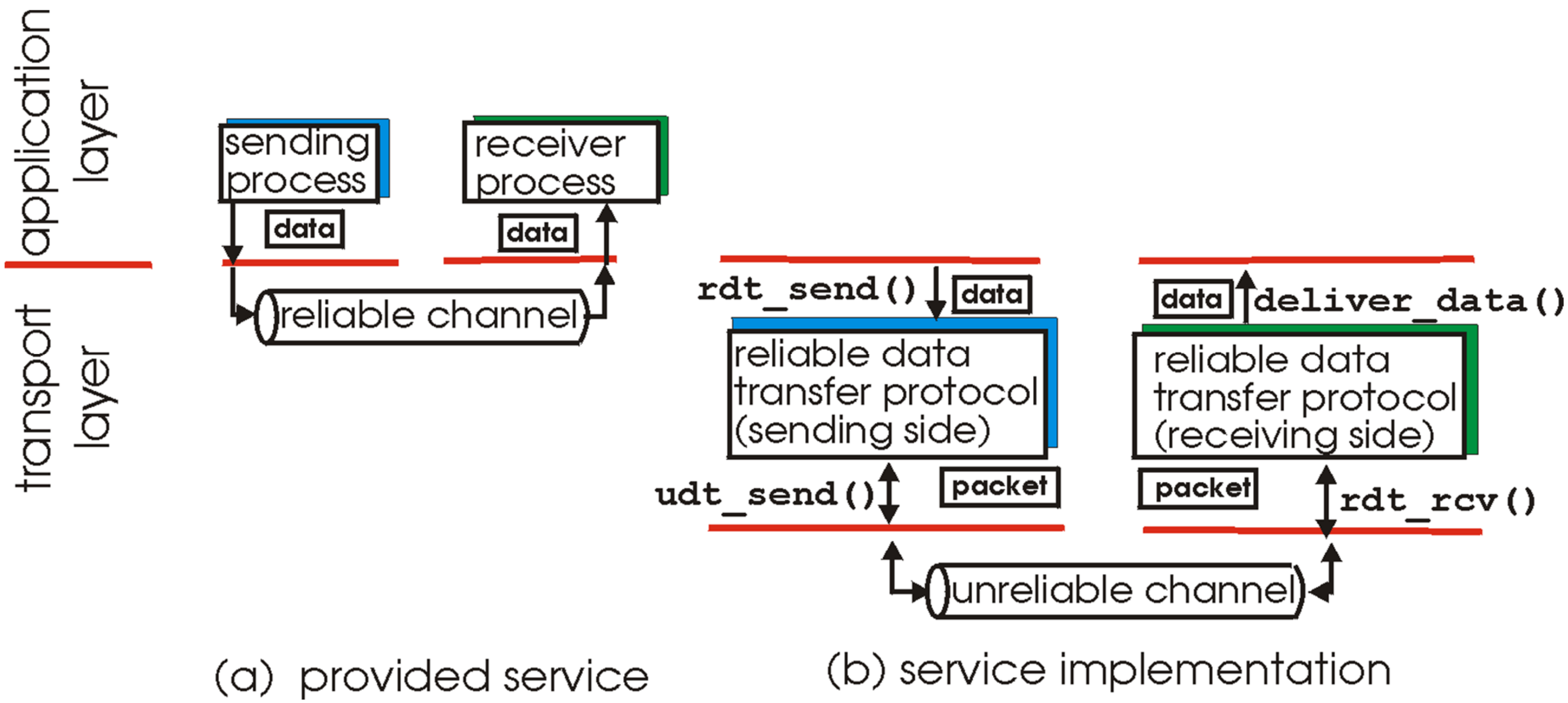
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

可靠传输原理

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

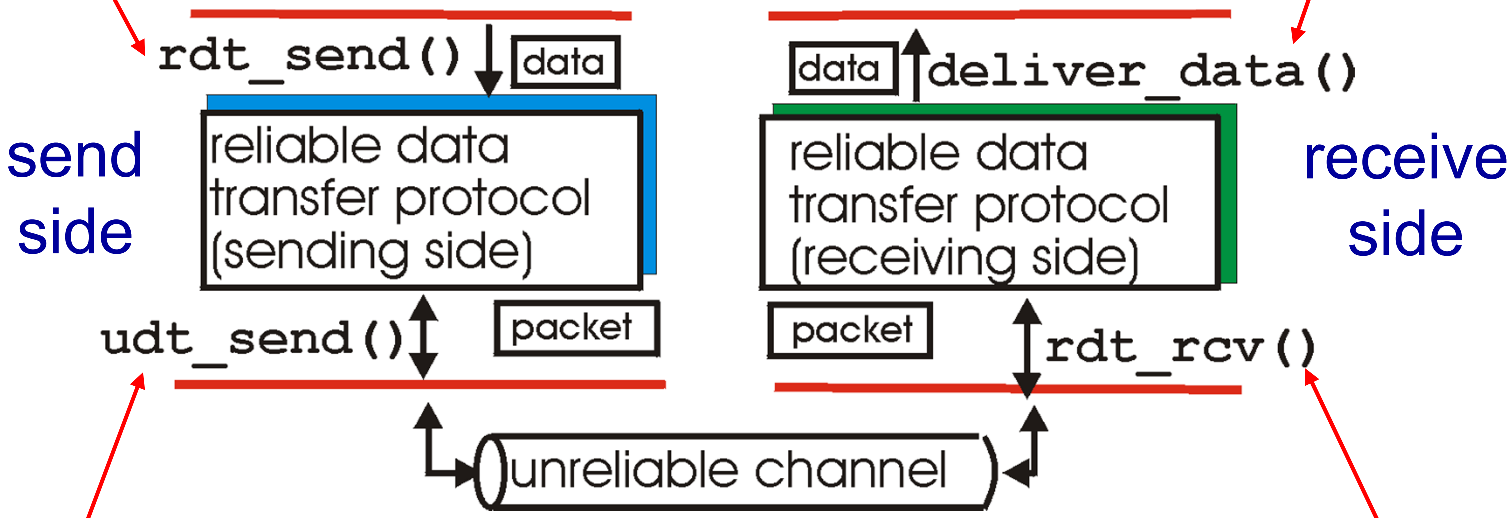


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

3.4.1 可靠传输协议和算法

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

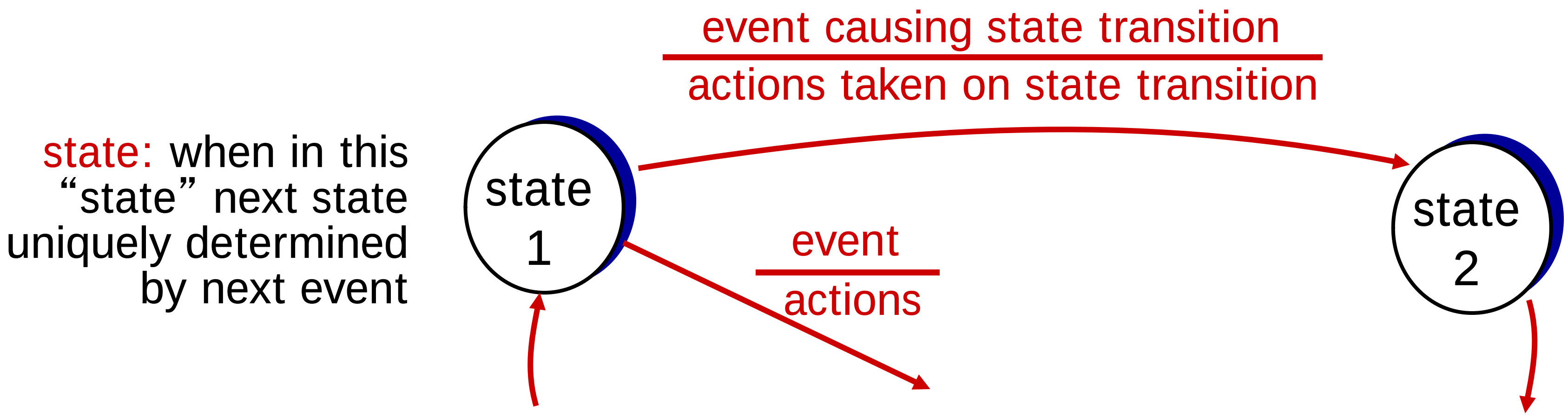


udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

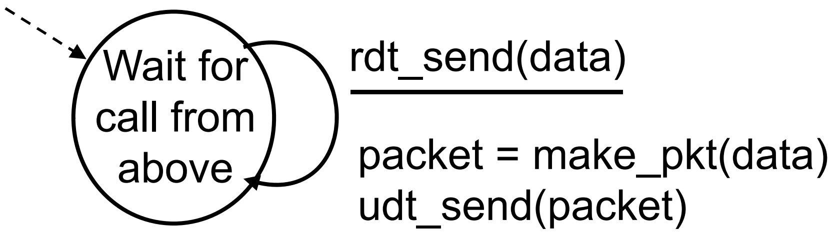
we' ll:

- incrementally develop sender, receiver sides of <u>r</u>eliable <u>d</u>ata <u>t</u>ransfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!（半双工）
- use finite state machines (FSM) to specify sender, receiver

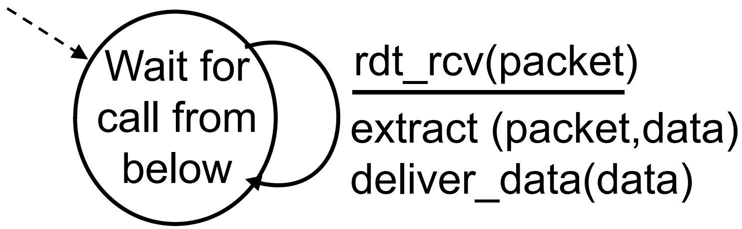


rdt1.0: 可靠信道上的传输协议

- 假设信道完全可靠
  - 无位错
  - 不丢包
  - 有序
- 发送方和接收方分别用一个FSM状态描述:
  - 发送方向信道发送分组（上层调用）
  - 接收方从信道读取分组 （下层调用）



sender



receiver

Sender：

```
While（true）do {
 Wait_Event( );
  GetMsgFromApp(data);
  packet=make_pkt(data);
  udt_send (packet);
}
```

Receiver:

```
While ( true ) do {
      Wait_event( );
       GetpktFromNet( packet);
      data=Extract (packet);
       deliver_data(data);
}
```

rdt2.0：只有位错误的可靠传输协议

假设：分组可能出现位错（其余条件和1.0一致）

问题:

1　如何发现错误？

“校验和”检错

2　如何从错误中恢复？

人在对话时候出现的情况？

回答：“OK”或者“说什么？请重复一遍”

前者:确认；后者否定且要求重复。

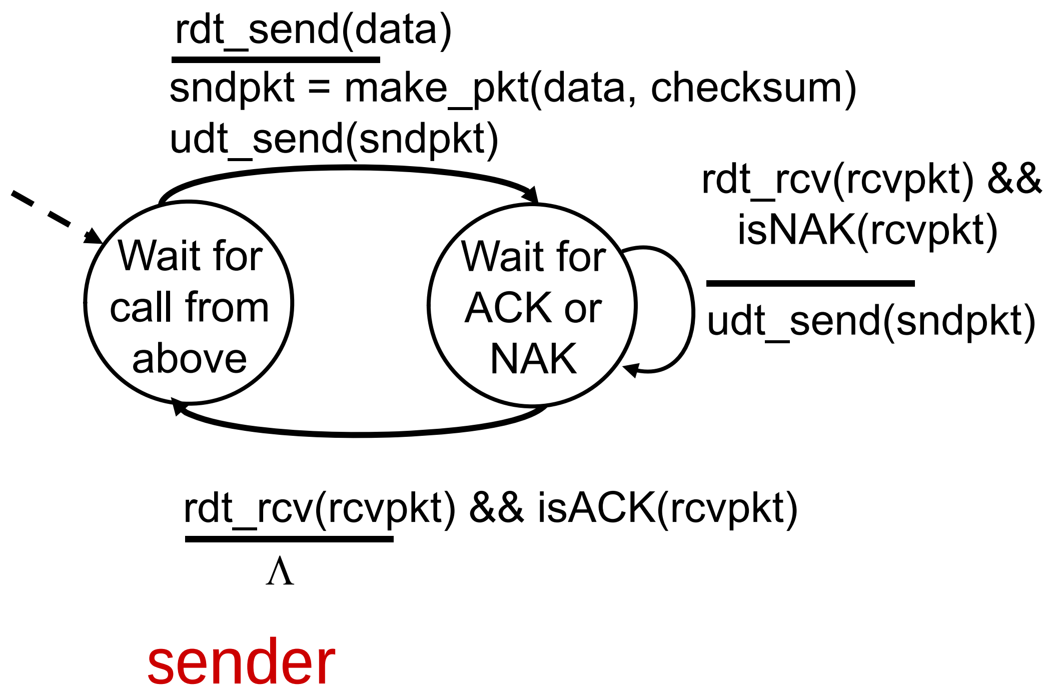
rdt2.0：只有位错误的可靠传输协议

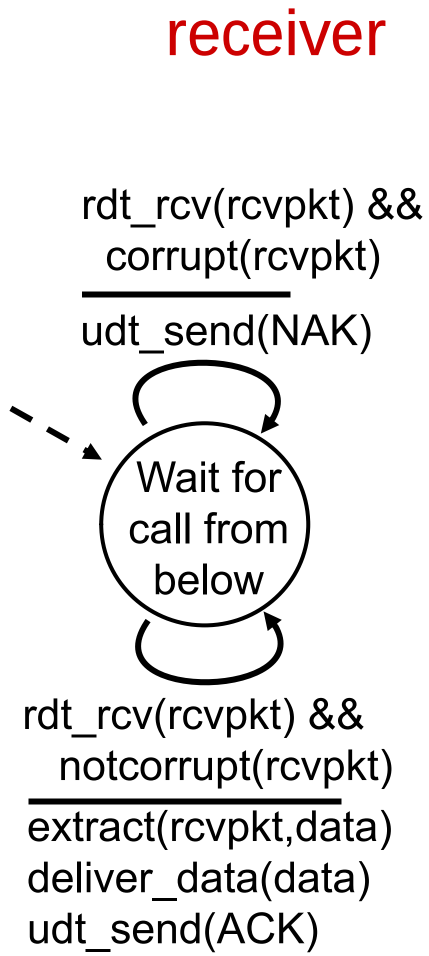
- **假设**：分组可能出现位错（其余条件和1.0一致）
  - 分组中采用校验和检测位错
- **问题**：如何从错误中恢复？
  - 肯定应答 *(ACKs)*: 接收方明确告诉发送方正确收到分组
  - 否定应答 *(NAKs)*: 接收方明确告诉发送方收到的分组有错误
  - 发送方收到NAK，重传分组

- **ARQ实现方法**：
  - 错误检测
  - 接收方反馈控制消息（ACK,NAK） rcvr->sender

ARQ（自动重传请求机制）：
包括检错、接收方反馈（ACK,NAK)和重传等级机制。

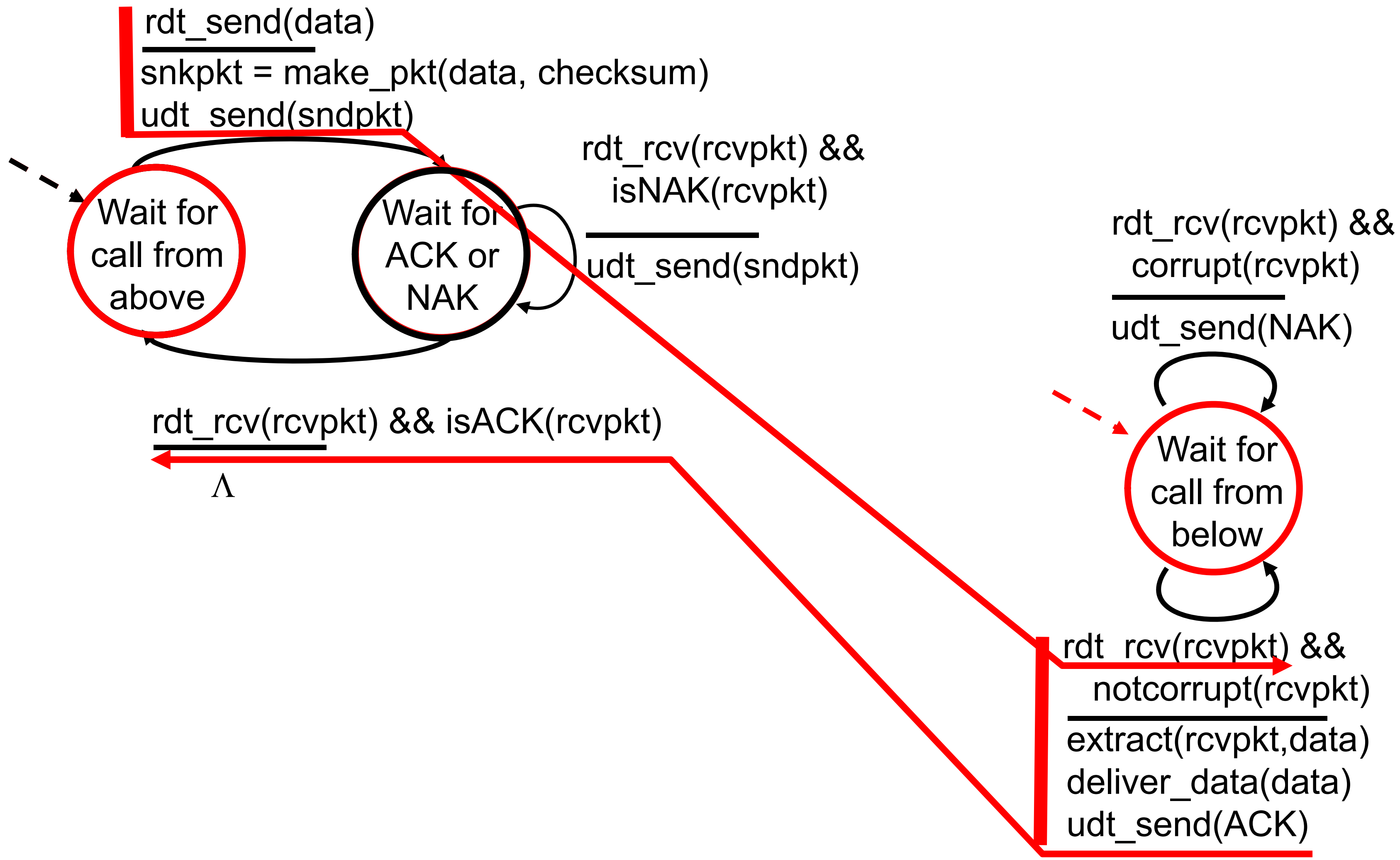
rdt2.0: FSM 状态转换图



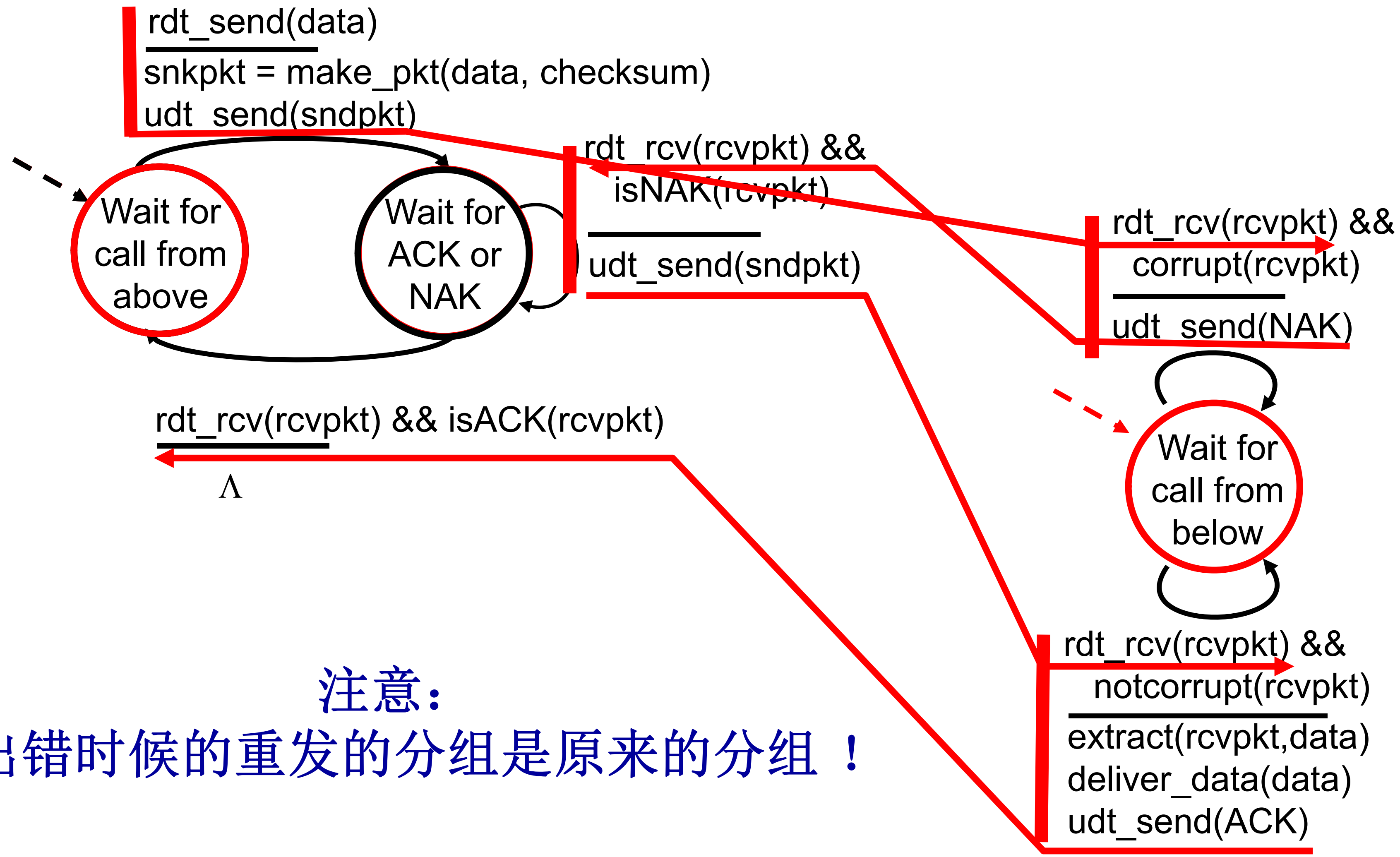


发送方2个状态，
接收方1个状态.

rdt2.0 操作示意图


rdt_send(data)
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)
Wait for call from above
Wait for ACK or NAK
rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ
rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)
udt_send(NAK)
Wait for call from below
rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

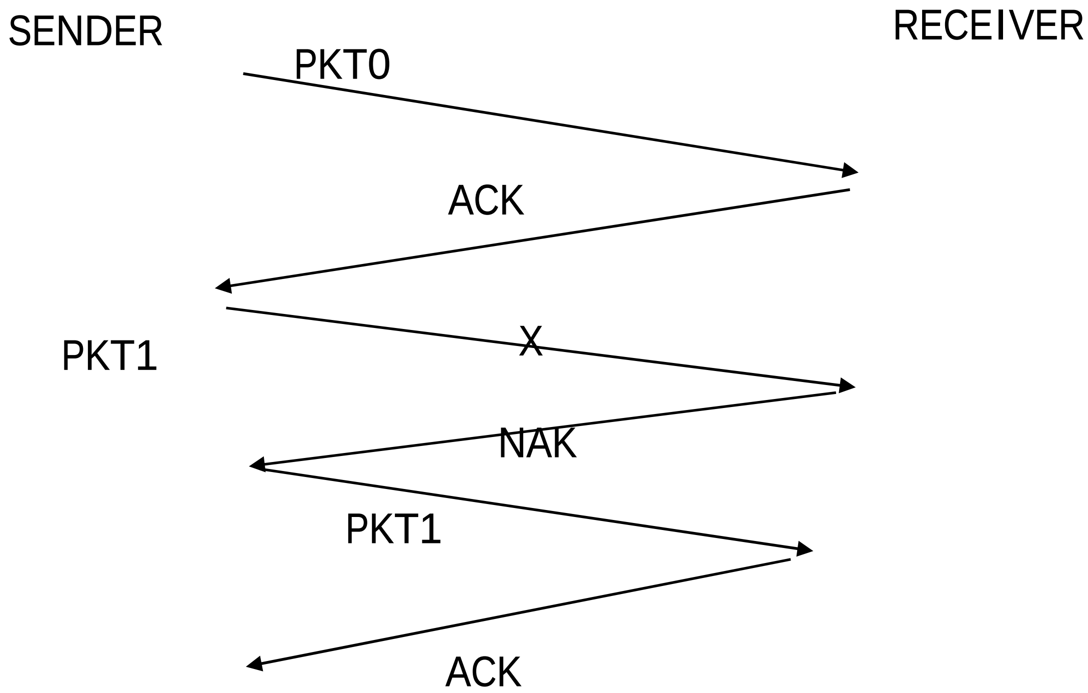
rdt2.0: 位出错场景



注意:

出错时候的重发的分组是原来的分组 !

RDT 2.0 举例



Rdt2.0　伪代码

Sender:

```
While（ true ）do {

 Wait_event();

 case rdt_send():
  { GetMsgFromApp(data);
sndpkt=make_pkt(data,checksum);
   udt_send (sndpkt);
  }
 case  rcv_pkt():
   { if (NAK) udt_send (sndpkt);}

                }
```

Receiver:

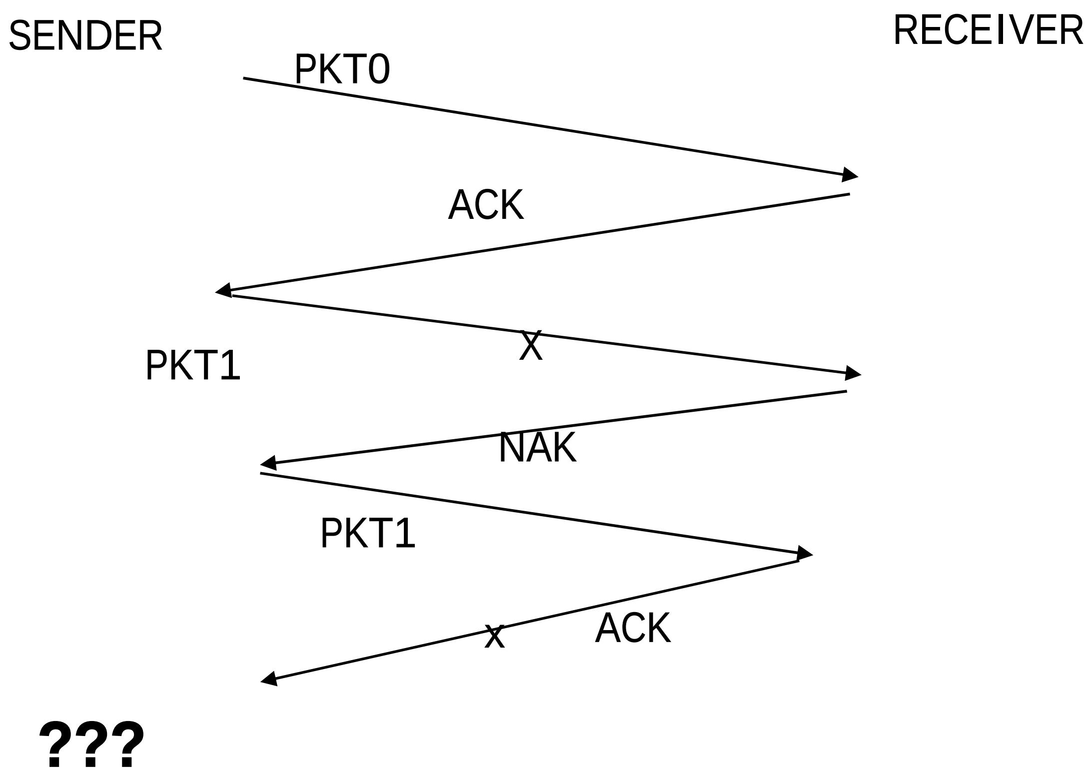
```
While ( true ) do {

       Wait_event();

       if (NO_ERROR)
       {   GetpktFromNet( sndpkt);
           data=Extract (sndpkt);
           deliver_data(data);
            udt_send (ACK);
        }
        else udt_send (NAK);

                       }
```

RDT 2.0 存在的问题



只能处理分组出现错误的情况，
不能处理应答消息出现错误 ！

rdt2.0 存在的问题

若ACK/NAK 出错，会出现什么情况？

- 发送方不清楚接收方是否正确接收分组。
- 单重发分组可能导致接收方有重复的分组。

处理重复分组：

- 如果 ACK/NAK 出错，发送方重传分组
- 每个分组编制序号
- 接收方依据接收分组序号识别重复分组，不提交重复的分组给上层

stop and wait

发送方发送一个分组，然后等待接收方应答。

停等协议和流水线协议区别！

rdt2.1：处理数据和控制消息出错的协议

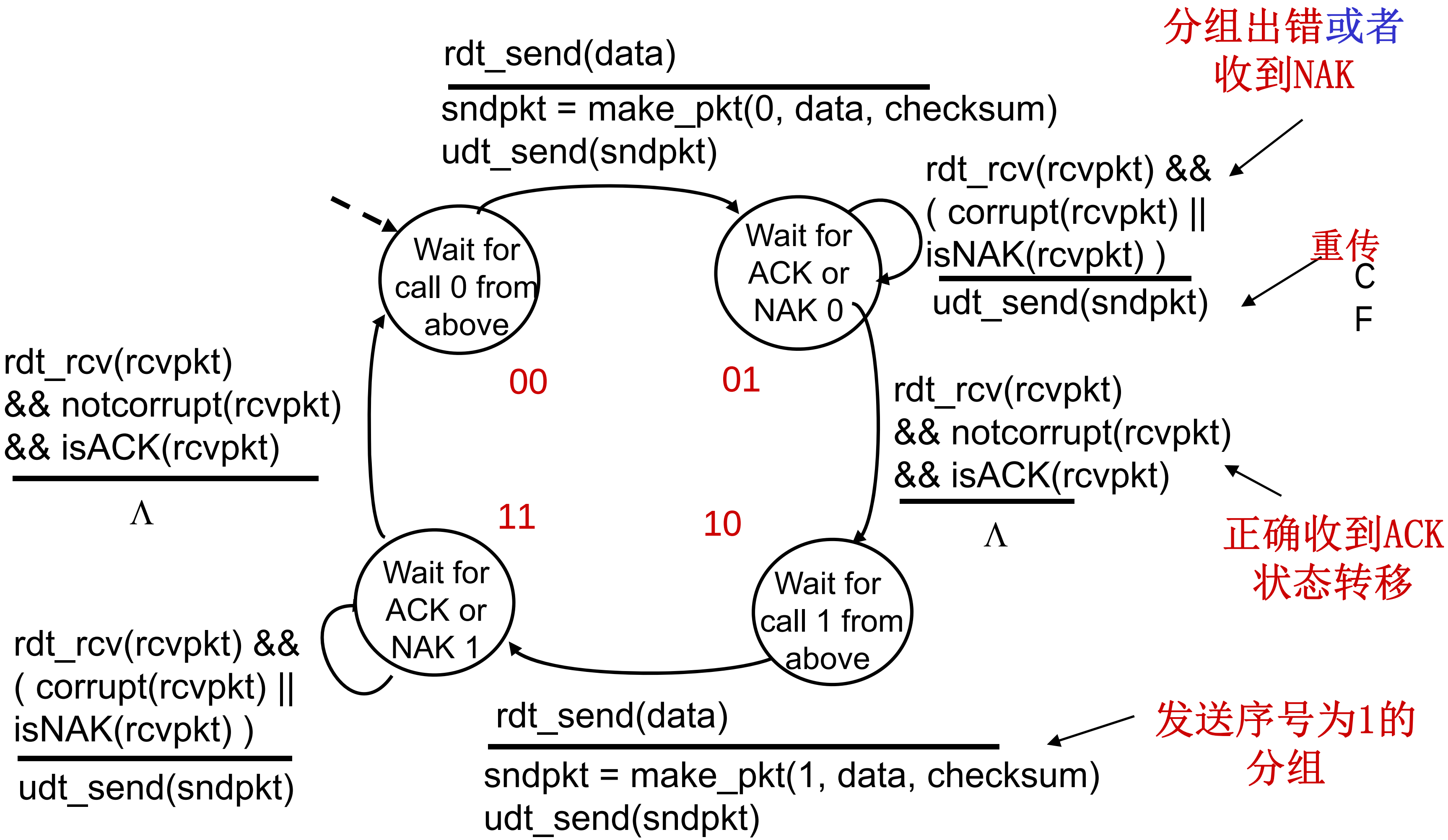
sender:

- 给分组编制序号（**sequence number)**
- 停等协议只需1位序号
- 需要检测 ACK/NAK 是否出错（校验码）
- 四个转态
  - 对应发送和等待两种转态，需要区别序号0和1，组合数为4。

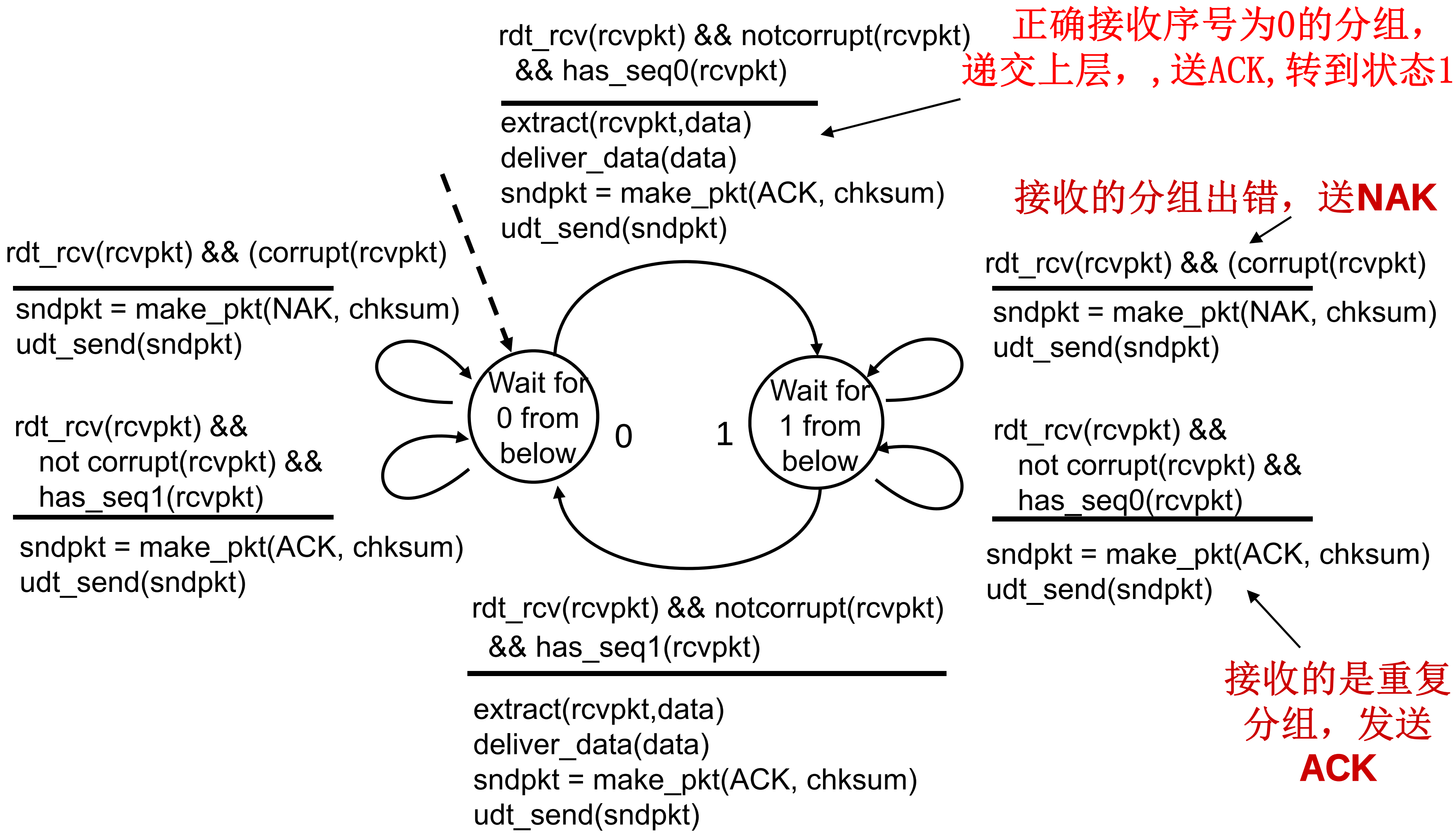
receiver:

- 需要检测分组是否重复（检测序号）
  - 所在状态表示下一个期待接收的序号，因此需要2个转态。
- 注意：接收者并不知道上次发送的ACK/NAK，在发送方是否正确接收。

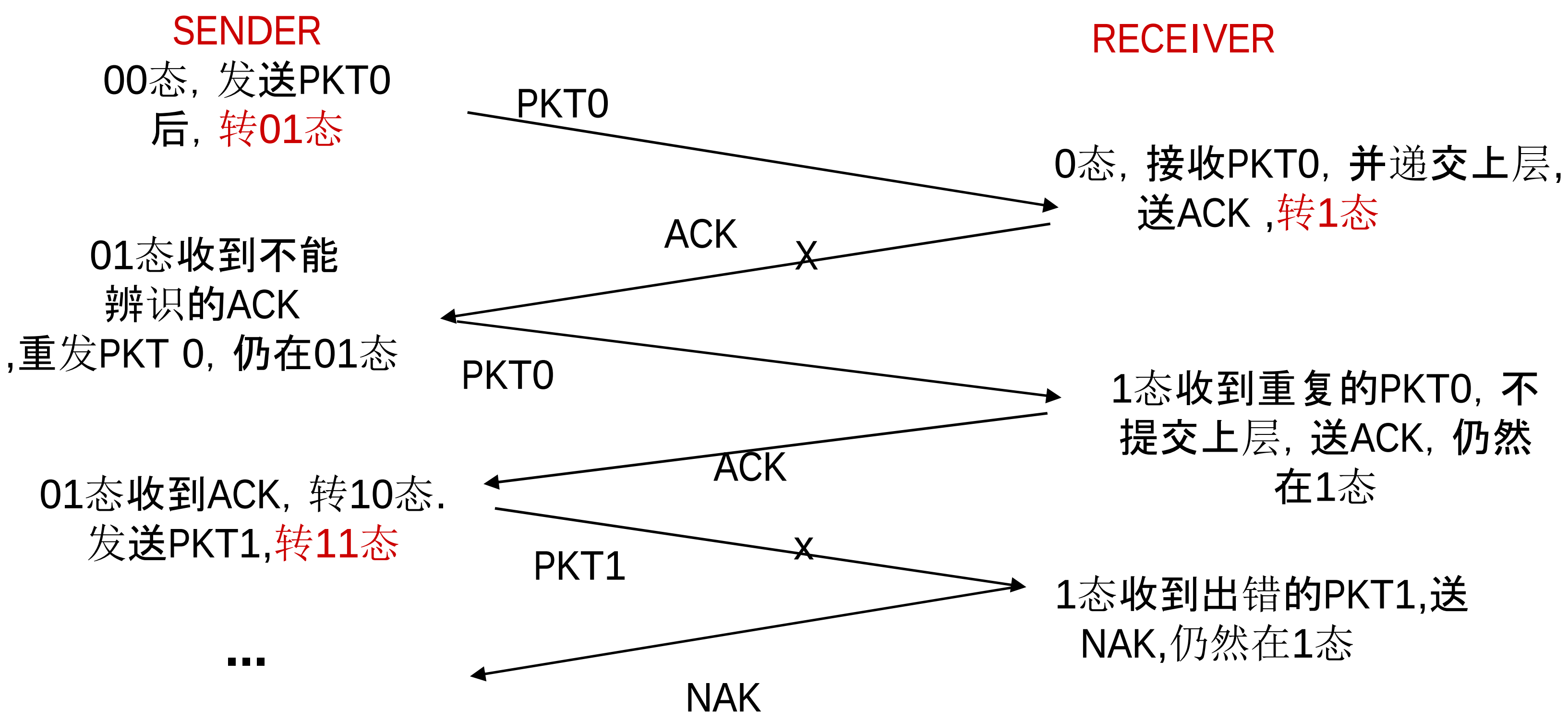
rdt2.1: Sender



rdt2.1: Receiver



RDT 2.1 举例



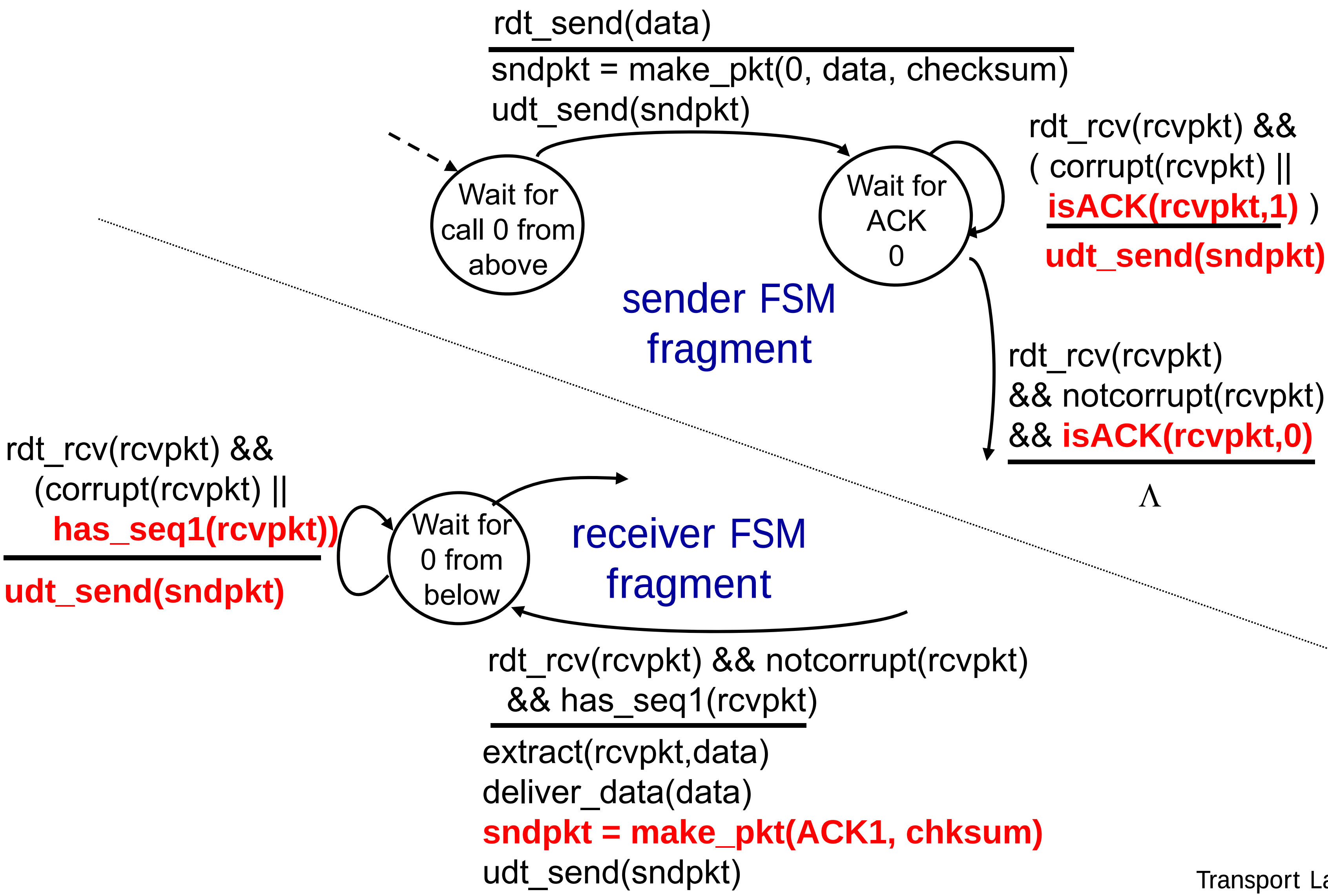
结合转态图，设计场景，构造FSM所有可能的转态转换

rdt2.2 不采用NAK的协议

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

课后作业

rdt2.2: sender, receiver fragments



rdt3.0: 处理信道有错和丢包的协议

假设:

信道有错、丢包，无序

- 校验和检错，ＡＣＫ应答决定重发，序号判定是否重复分组。

问题:

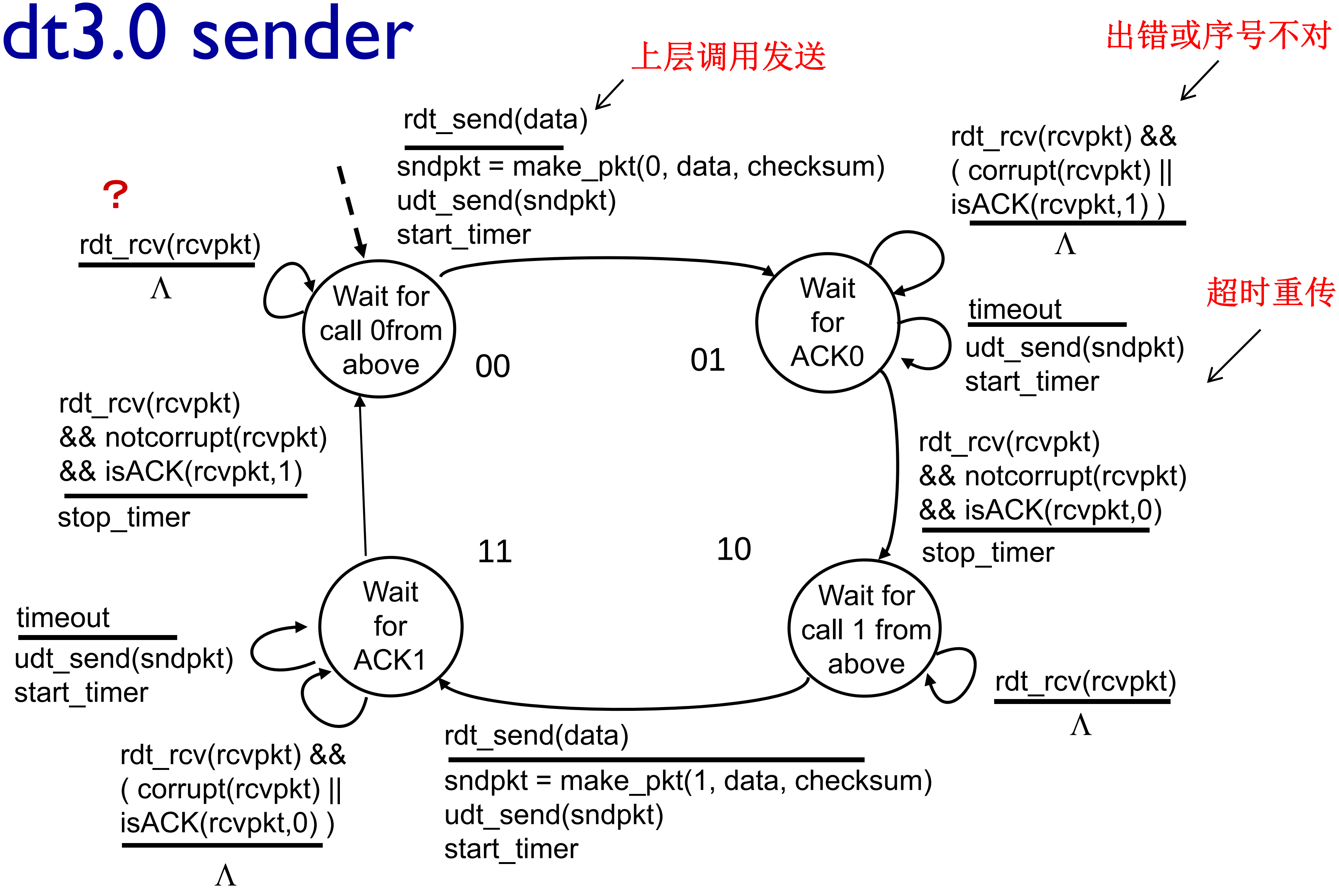
1 如何检测丢包?

2 何时重发丢失分组?

方法: 发送分组后等待时间T

- 发送方若T时间内没有收到ACK（timeout:**超时**），重发分组。
- 发送方如果T时间后又收到超时对应的分组ACK，说明接收方存在重复分组，可通过序号判定是否重复。
- 接收方必须说明应答（ACK n)对应的分组序号
- 需要定时器记录分组发送后的时间。

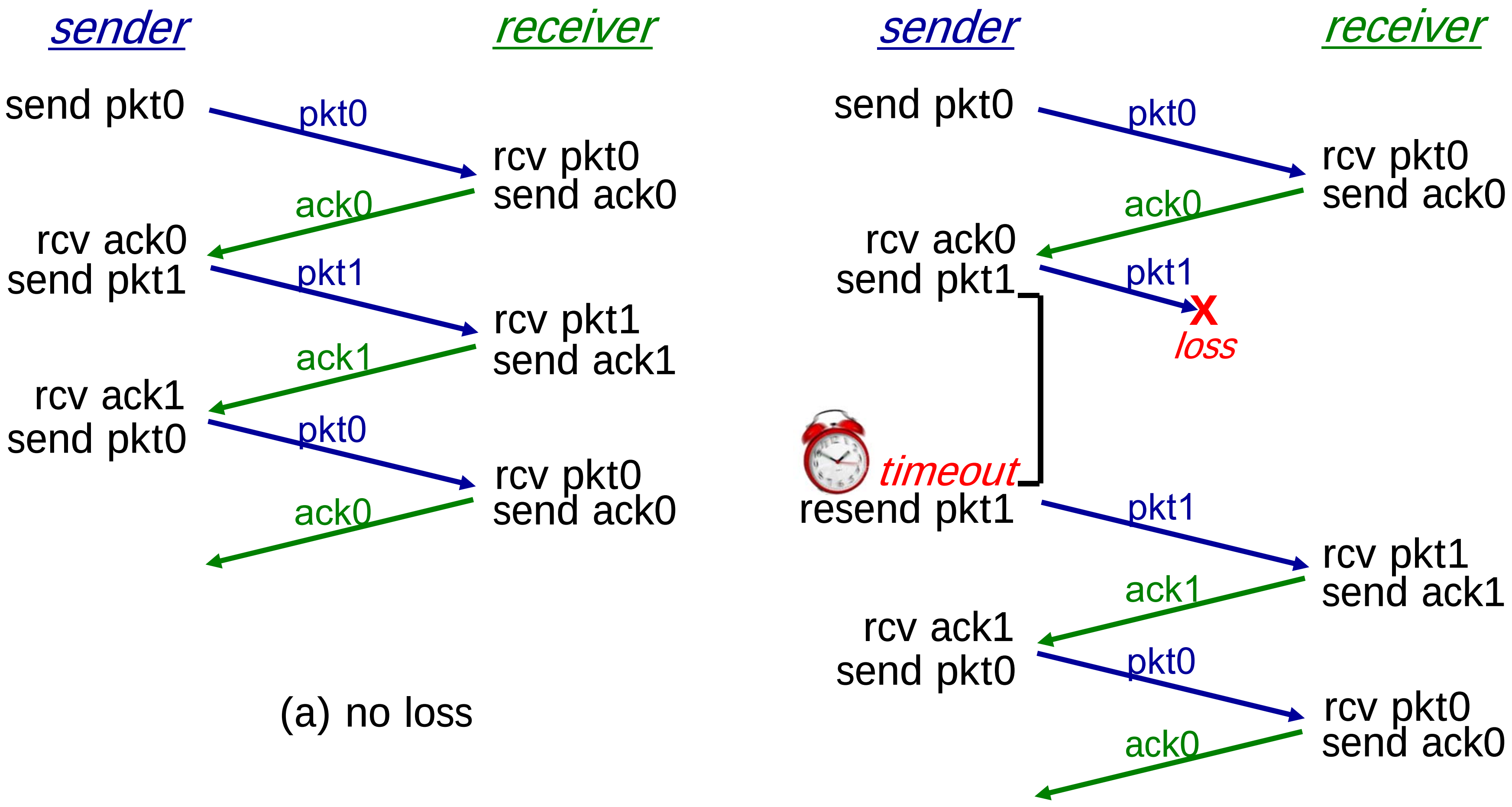
rdt3.0 sender



rdt3.0 receiver

- 参考rdt 2.2,课后作业.

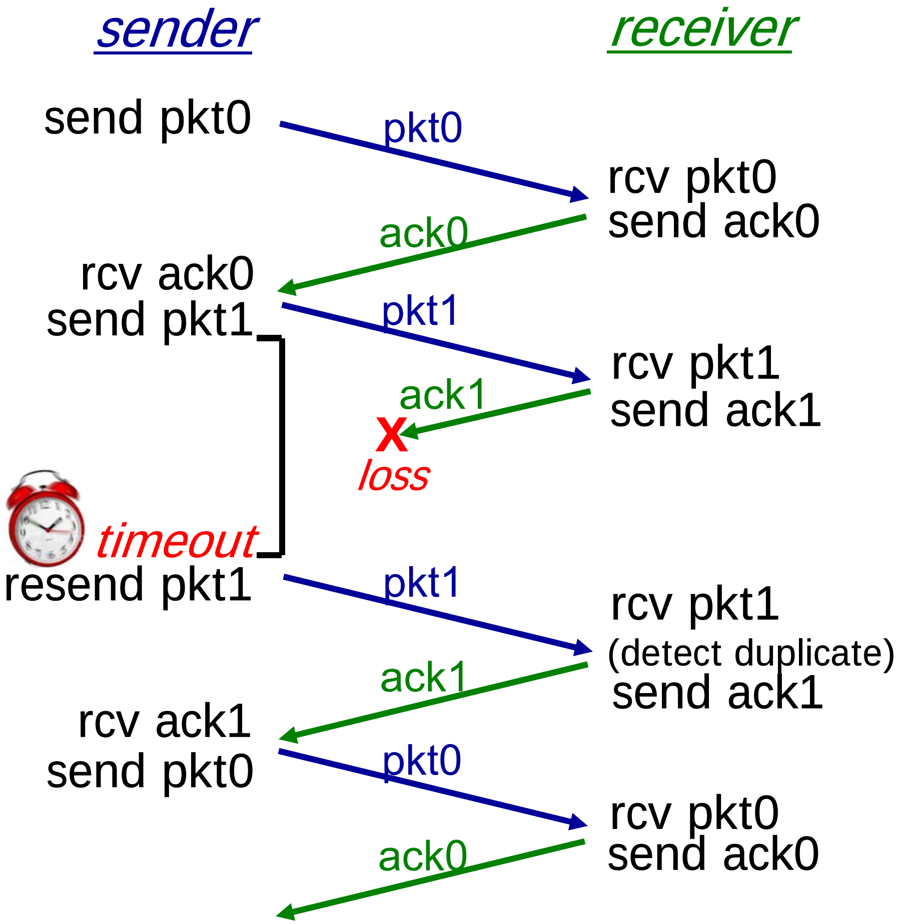
rdt3.0 举例



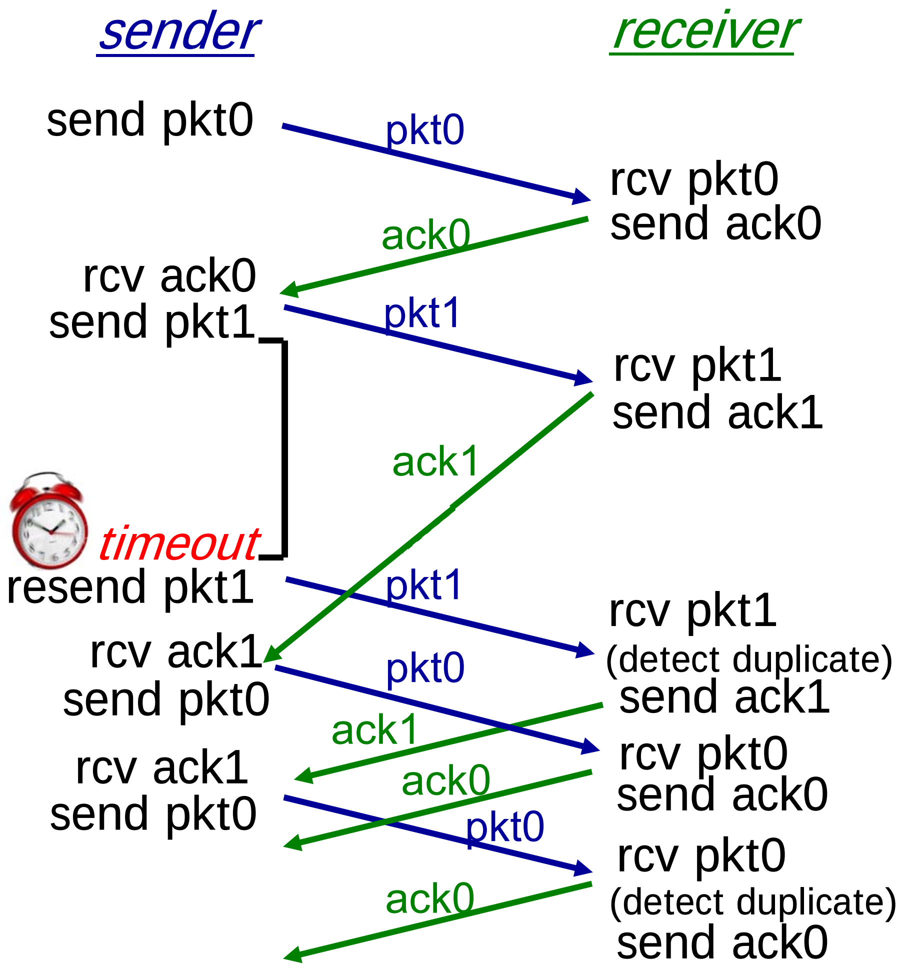
(a) no loss

(b) packet loss

rdt3.0 举例



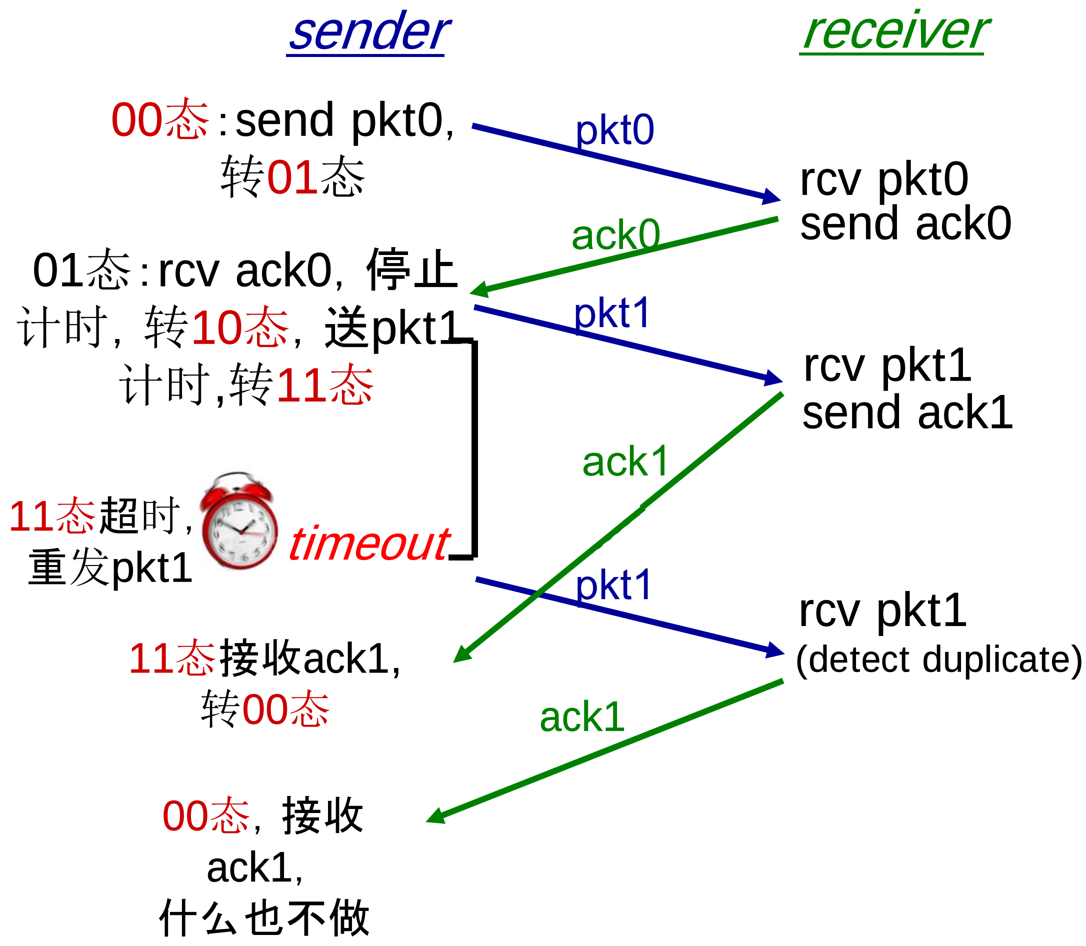
(c) ACK loss



(d) premature timeout/ delayed ACK

rdt3.0 举例

为什么发送方在等待上层调用状态（00或者10），收到接收方应答消息后，什么动作都不做？



所在转态不同接收消息后的动作可能不同！需要结合状态分析转移动作。

rdt3.0 （停等协议）的性能

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bits packet:

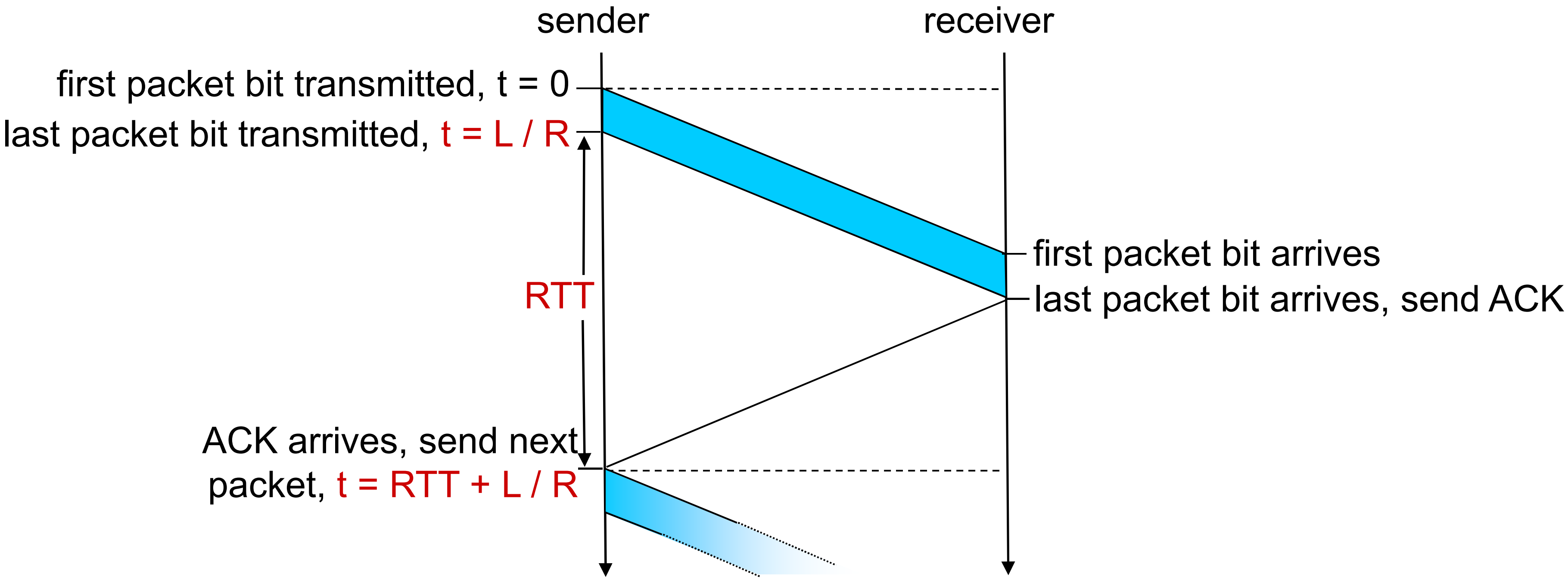
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

  - U $_{sender}$: *利用率* – 发送方用于分组传输的时间比例

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

  - if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link
- 停等协议限制了物理资源的使用！

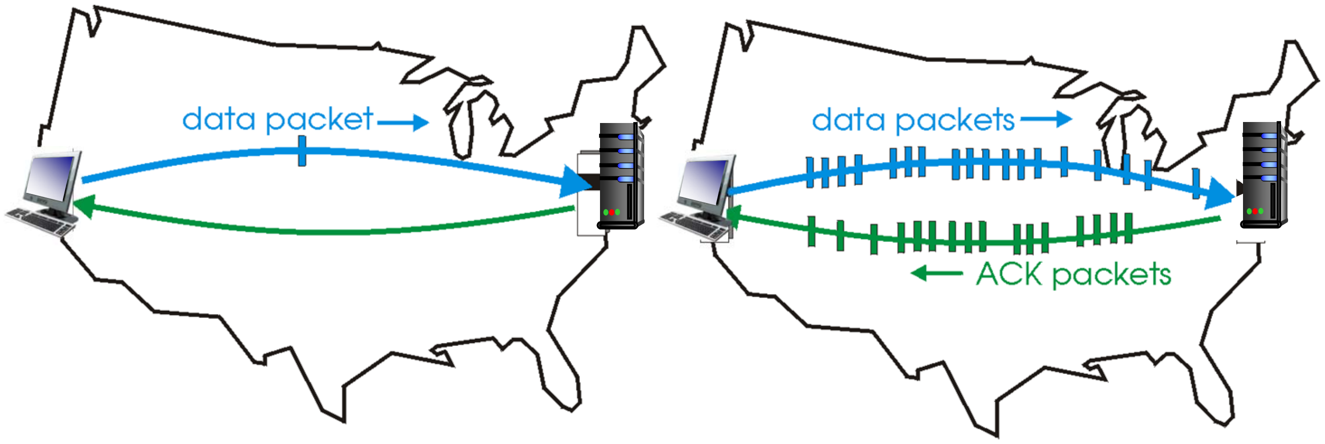
停等协议的性能



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

3.4.2 流水线（pipeline）协议

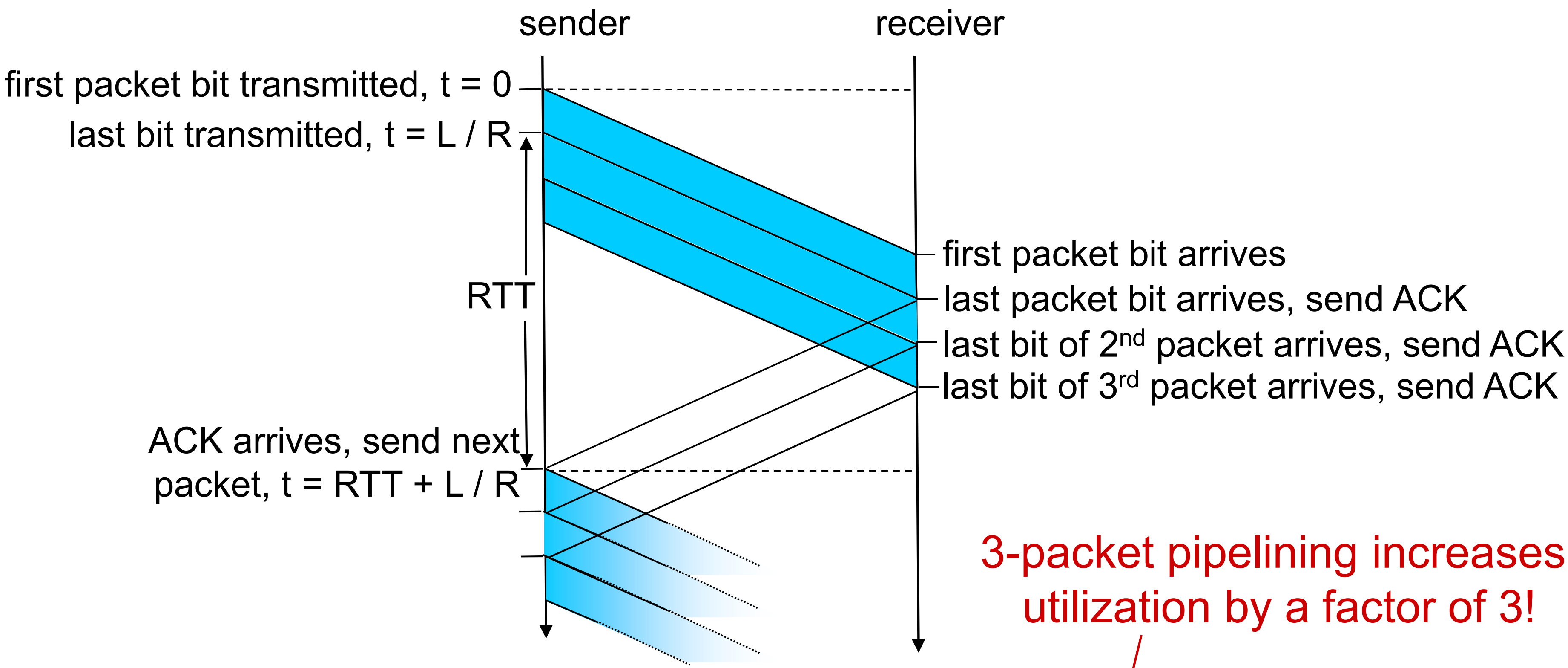
pipelining: 允许发送方不等待应答，连续发送N个分组
- 需要扩大分组序号范围
- 在发送方和接收方需要缓存分组



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

- 两种流水线协议: *go-Back-N, selective repeat*

流水线协议提高性能



3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

流水线协议：概述

Go-back-N:

- 发送方可以连续发送N个未被应答的分组
- 接收方只发送累计ACK *(cumulative ack)*
  - 不缓存和递交失序分组
- 发送方对未被应答的最早分组计时（定时）
  - 定时器溢出时，需要重传所有未被应答的分组。

Selective Repeat:

- 发送方可以连续发送N个未被应答的分组
- 接收方对每一个分组发送独立的ACK
- 缓存失序的分组，但不递交
- 发送方对每一个未被应答的分组计时（定时）
  - 定时器超时，只重传对应的一个分组。

3.4.3 Go-Back-N：滑动窗口协议

发送方采用流水线方式，接收方只接受有序的分组

举例：可以发送多个缆车，但只接收有序的缆车

- 发送方需**N**个缓存，缓存**N**个已发送但未被应答的分组，也就是发送窗的大小。
- 若超时，发送方重传所有未被应答分组。

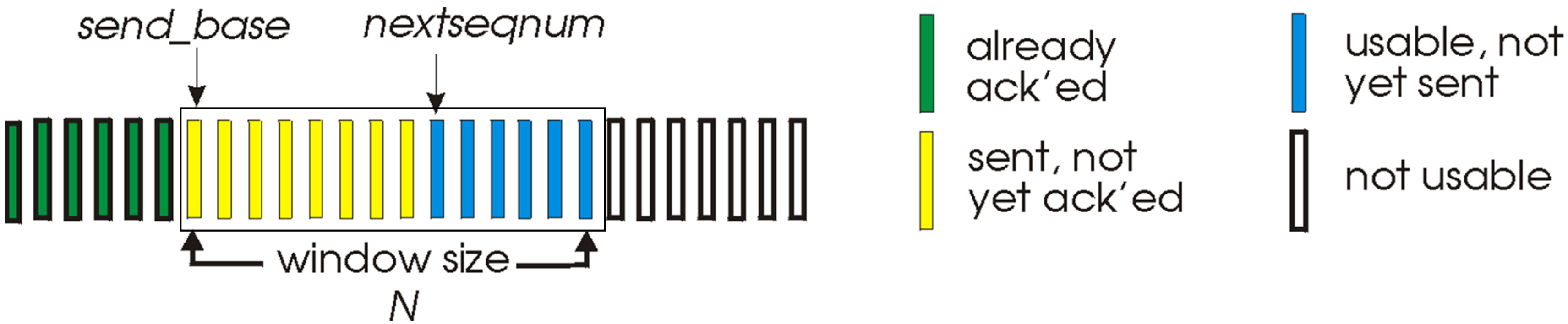
- 接收方只接收正确和有序的分组，不需提供接收 缓存。
- 接收方收到分组（无论错误与否），只应答已经接收分组中的最高序号**ACK**。

优点：接收方处理简单。

缺点：一个分组出错或者丢掉，必须重传多个分组。

Go-Back-N: 发送方滑动窗口协议

- 分组头中设置K位序号
- “发送窗”允许多达N个序号连续的可发送的分组



- ACK(n): 累计ACK，应答包括序号n在内的所有未被应答的分组
  - 发送方可能接收到重复的ACK
- 对最早发送的分组定时
- *timeout(n)*:定时器超时，重传序号n在内的所有未被应答的分组

Base：窗口中最早发送且未被应答的分组。
Nextsequence：窗口中未发送分组的最小序号

Go-Back-N: 发送方处理事件

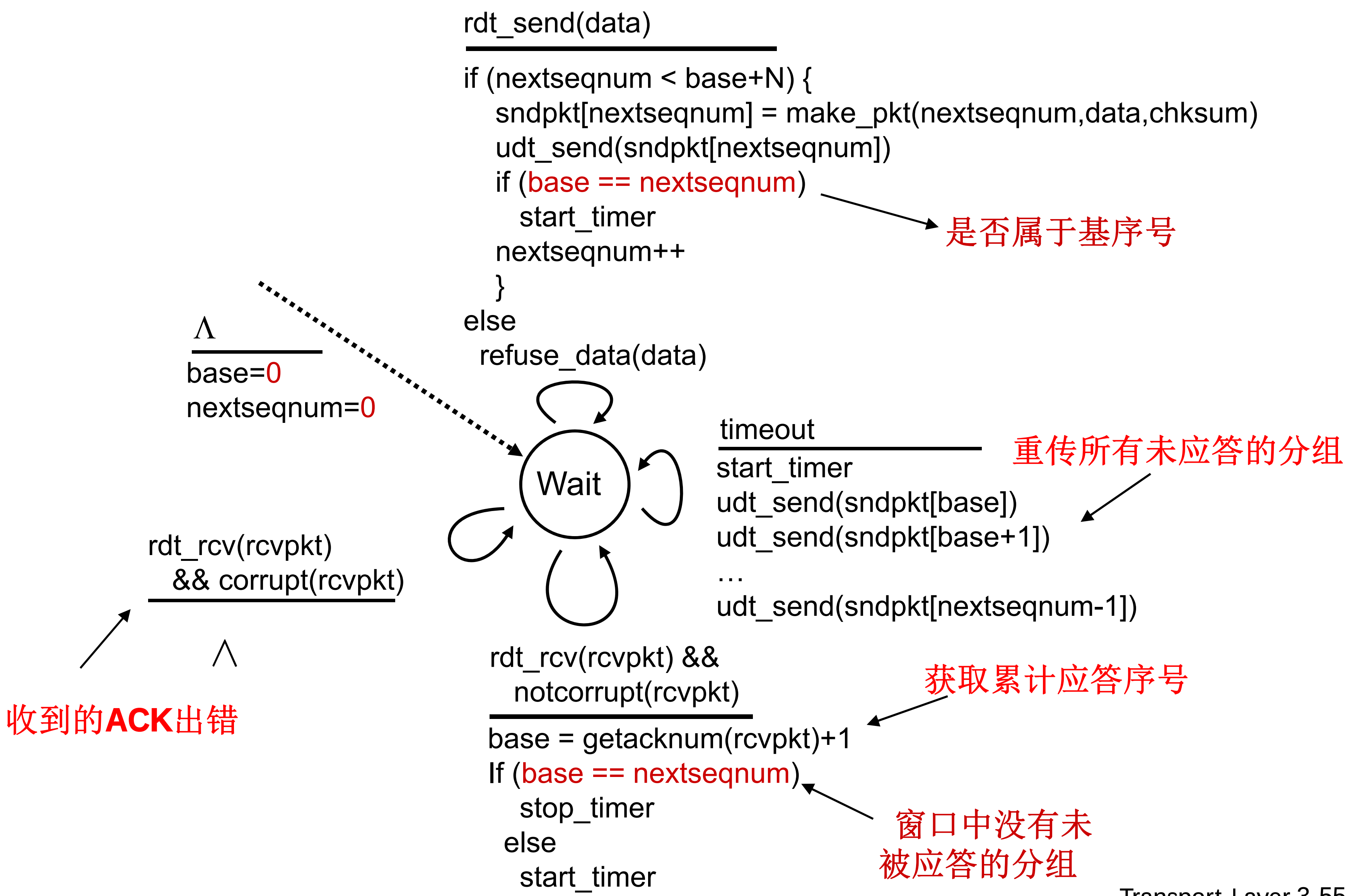
- 初始化：基序号base和下一个待发送的序号nextseqnum
- 发送分组：若窗口已满，拒绝发送；如果序号为基序号，需要定时。
- 收到的应答出错：忽略，不做事情。
- 收到正确的应答：获取应答序号X，表示序号小于等于X的分组都已正确接收，将这些分组移出窗口，BASE=X+1。如果窗口非空，启动新的定时器。
- 超时：重传所有未被应答的分组。

Go-Back-N: 接收方处理事件

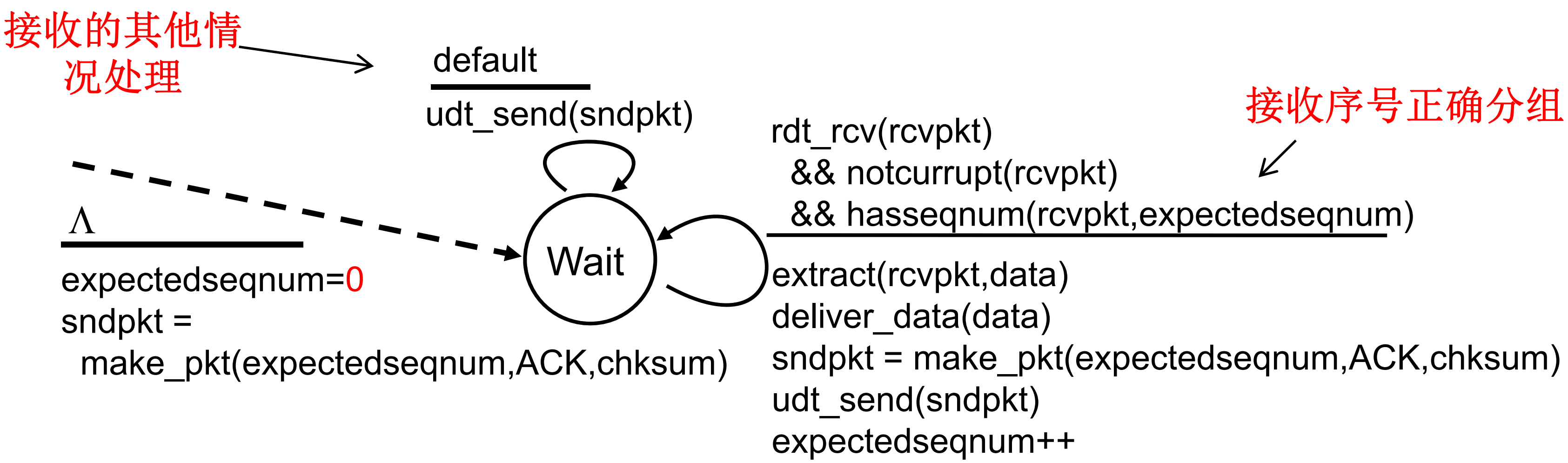
- 初始化：期待接收分组的序号（expectedseqnum)
- 接收序号正确的分组：递交，发送ACK(expectedseqnum)，expectedseqnum++。
- 接收到失序或者错误的分组：废弃（不保存），发送ACK(expectedseqnum)，表示期望收到分组的序号是expectedseqnum+1。

不保存失序的分组，保持expectedseqnum变量。

GBN: 发送方扩展 FSM

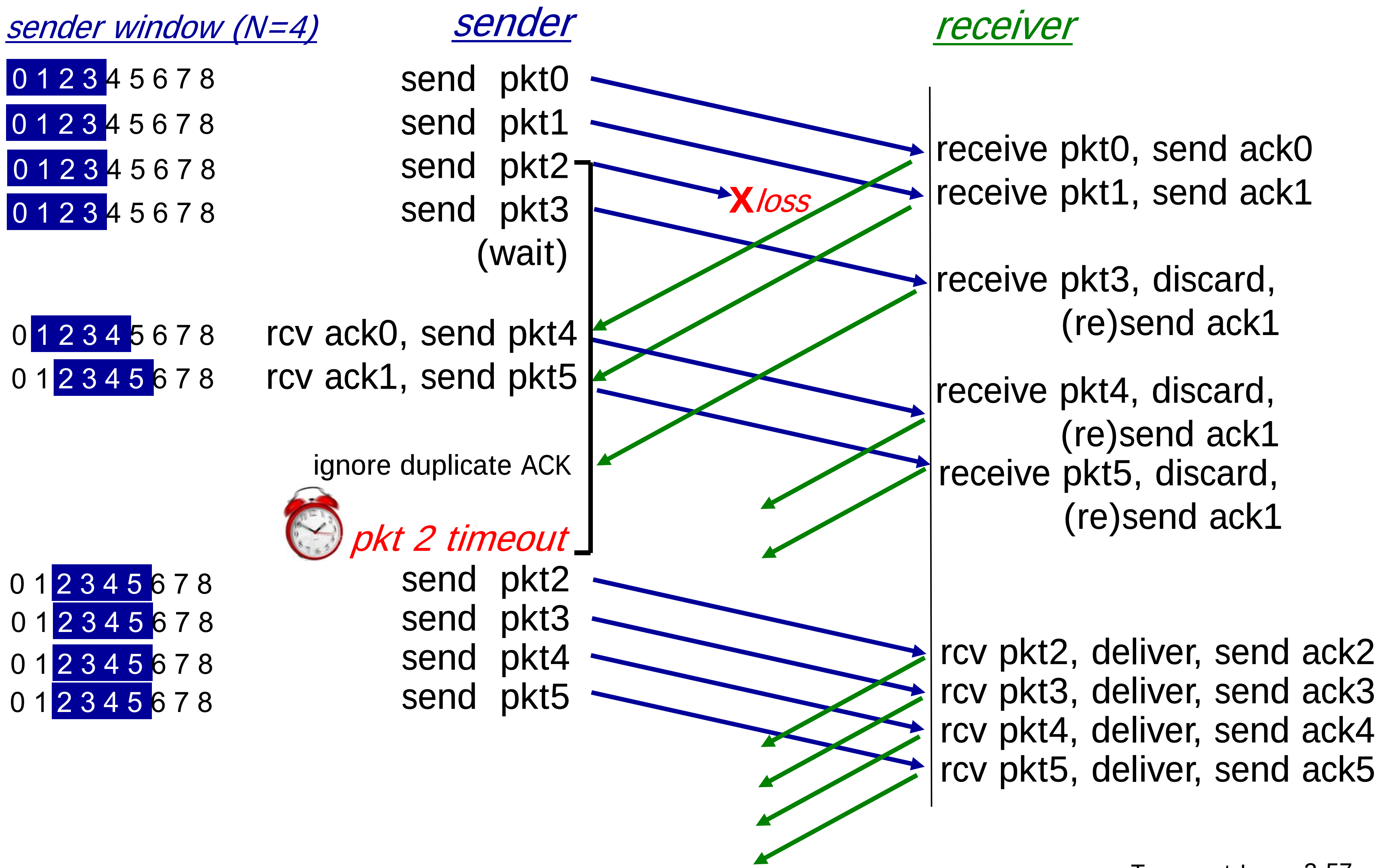


GBN: 接收方扩展 FSM



- ACK-only: 对于接收的分组（无论出错与否）都是发送目前正确收到分组中的最高序号ACK
  - 因此在发送方可能出现重复的ACK
  - 接收方只需要记住 **expectedseqnum**
- 对于收到的失序分组处理:
  - 废弃 （不缓存）: 没有接收缓存区！
  - 发送目前正确收到分组中的最高序号ACK

GBN 举例



sender window (N=4)
sender
receiver
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
send pkt0
send pkt1
send pkt2
send pkt3
(wait)
X loss
receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, discard,
(re)send ack1
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
rcv ack0, send pkt4
rcv ack1, send pkt5
receive pkt4, discard,
(re)send ack1
receive pkt5, discard,
(re)send ack1
ignore duplicate ACK
pkt 2 timeout
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

3.4.4 选择重传（ Selective repeat ）

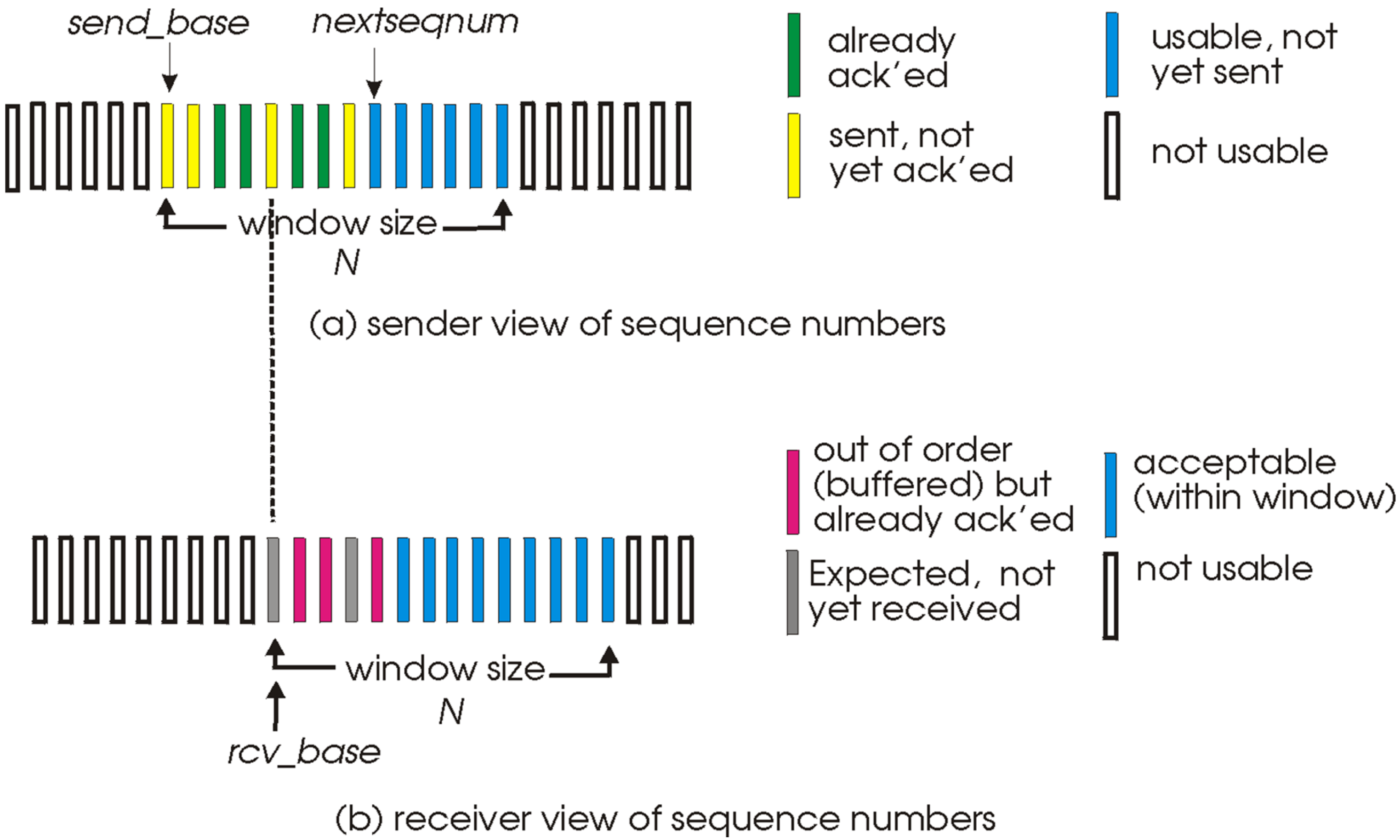
接收方对失序分组缓存，不提交。
发送方按需重传丢失和出错的分组

- 接收方对接收的无错分组（无论是否有序）分别发送ACK。

缓存失序分组，当失序分组都达到之后，再有序提交。

- 发送方只需重传没有ACK应答的分组。
  - 需要对每一个发送的分组定时。
- 发送窗口：
  - N 个连续序号的分组
  - 限制发送和未应答的分组

选择重传：发送和接收窗口



rcv_base:指向X+1，其中X是目前正确有序接收的序号。

选择重传算法

sender

上层有消息发送:

- 如果下一个要发送的分组序号属于窗口内，发送该分组。

timeout(*n*):

- 重传序号为n 的分组， 重新启动定时器

ACK(*n*)：若序号对应的分组属于 [sendbase,sendbase+N]:

- 标记分组n为已接收
- 如果分组 n 是最小序号的未应答分组， 移动基序号到下一个最小未应答分组位置。

receiver

无误接收分组pkt n 属于 [rcvbase, rcvbase+N-1]

- 发送 ACK(n)
- 是失序分组：缓存
- 是当前期待接受的有序分组 ：提交包括缓存区在内所有有序分组， 移动基序号到下一个失序的分组。

无误接收分组pkt n属于 [rcvbase-N,rcvbase-1]

- ACK(n)

Q ??

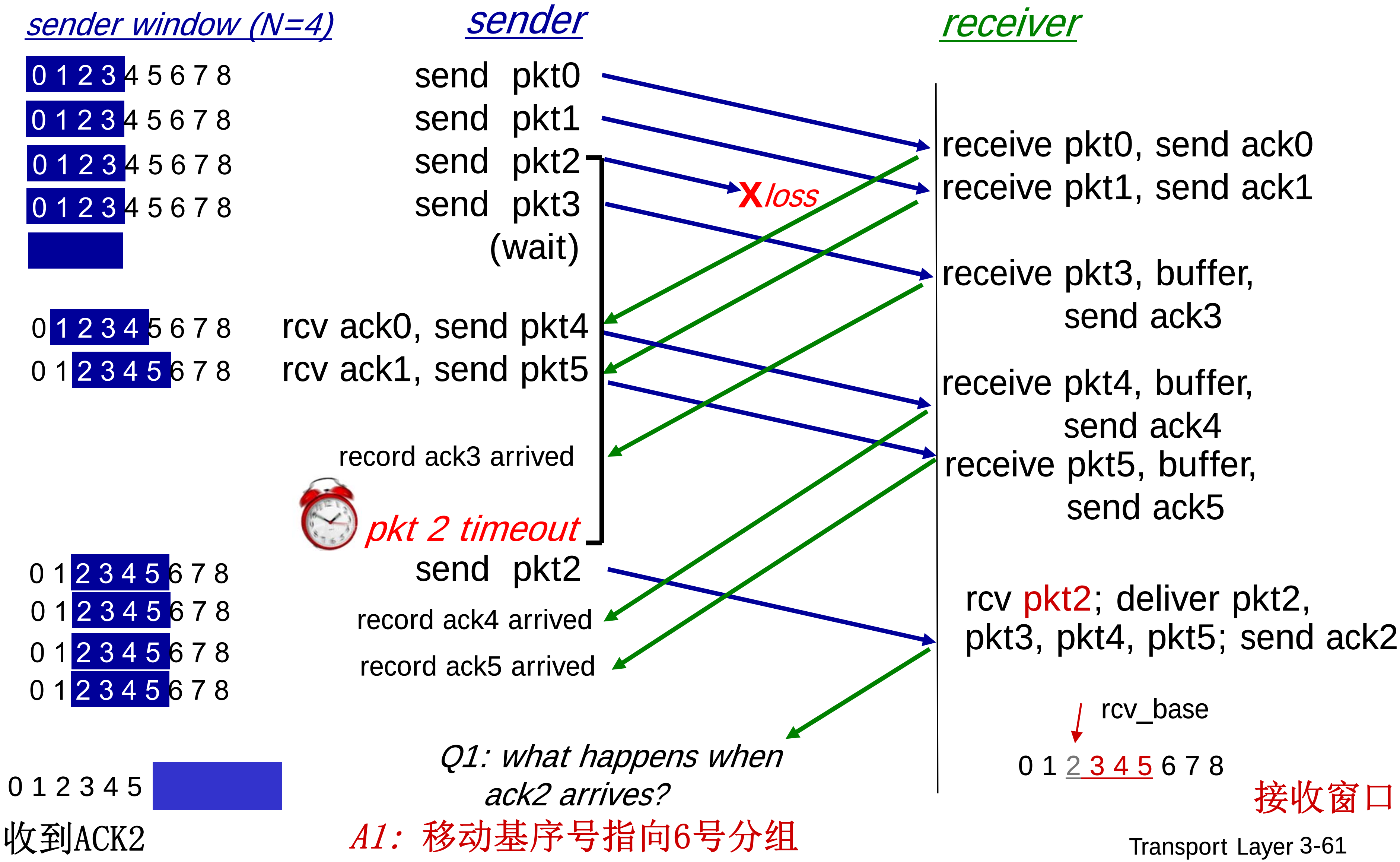
其他事件(分组出错）：

- 忽略（不做动作）

选择重传举例

Q2： what happens if ack2 lost ？

A2：会出现接收方重复收到PKT2情况，接收方必须再次回送ACK2，使得发送窗口移动！



思考：

1 接收方至少要有多少BUFFER?

2 接收方收到分组的序号可能在一个什么范围？

3 序号大小（循环使用）和窗口大小存在什么关系？

选择重传：

序号和窗口大小关系

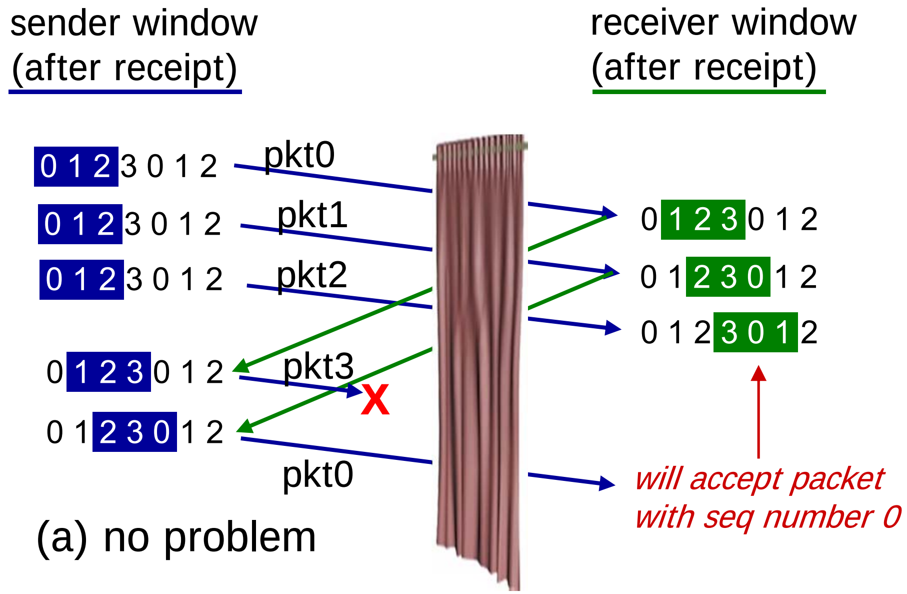
（序号可能需要重复使用）

example:

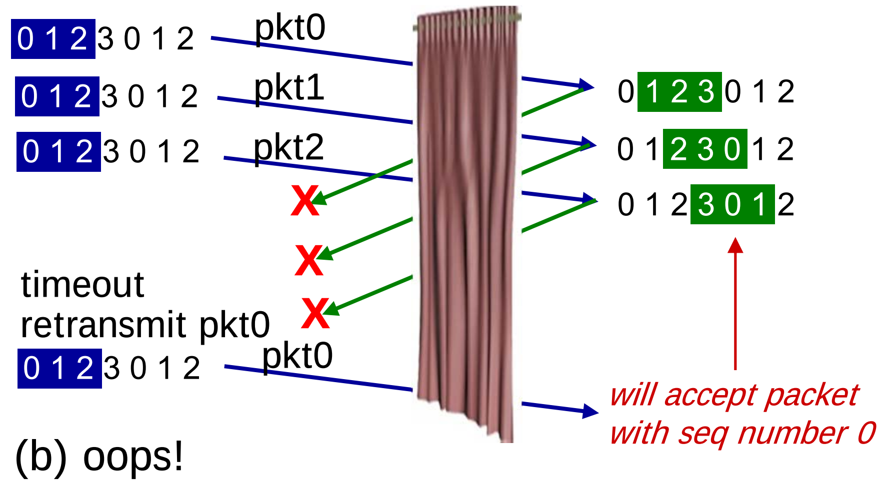
- 序号为0, 1, 2, 3
- 窗口大小为3
- 右边两种情况下接收方认为没有区别！
- （b）图中，接收方将重复分组识别为新的分组。

Q: 怎样才能避免出现(b)中的错误？

A: N<=(maxseq+1)/2



receiver can't see sender side.
receiver behavior identical in both cases!
something's (very) wrong!

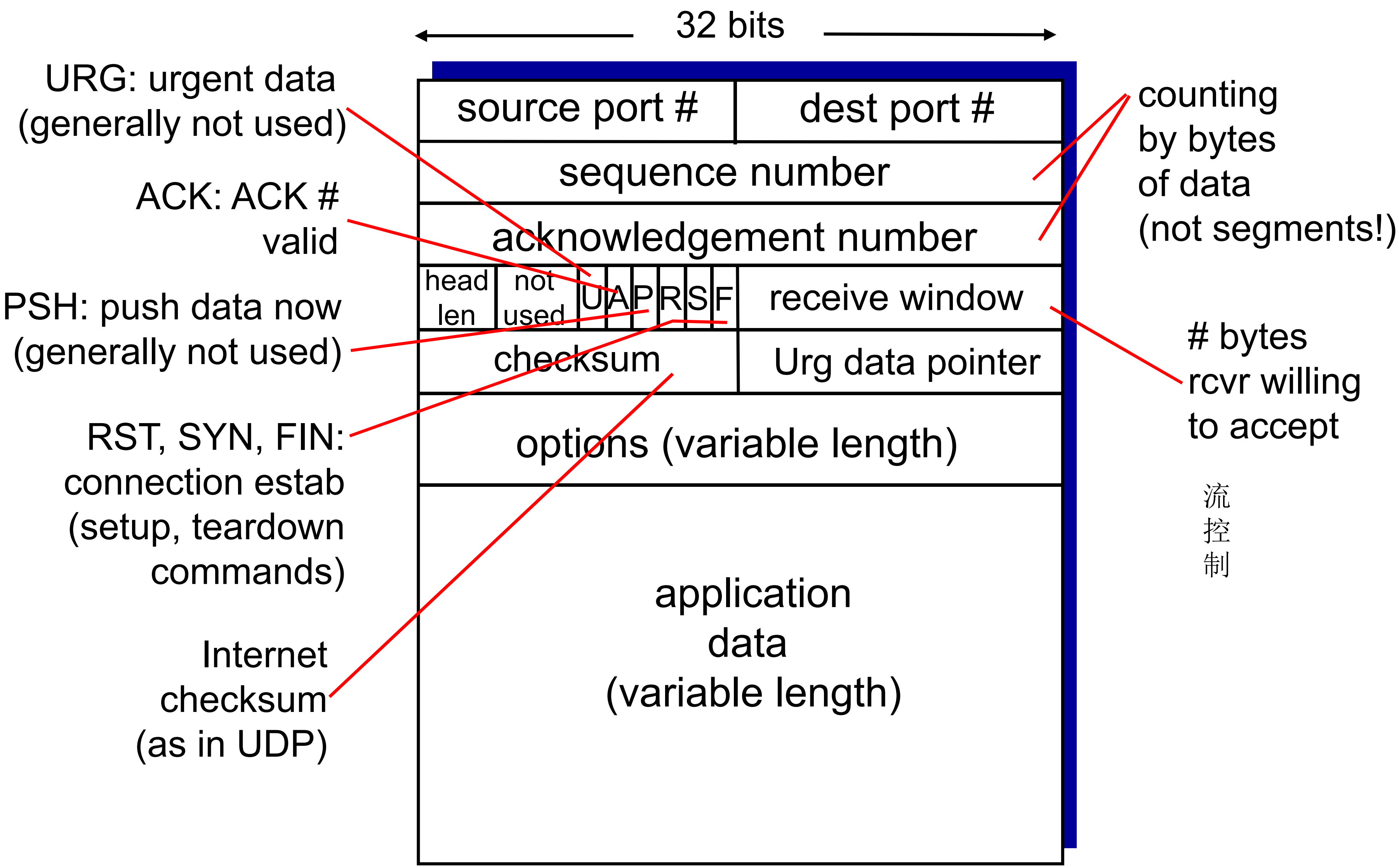


3.5 TCP协议

3.5.1 概述

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

3.5.2 TCP segment structure

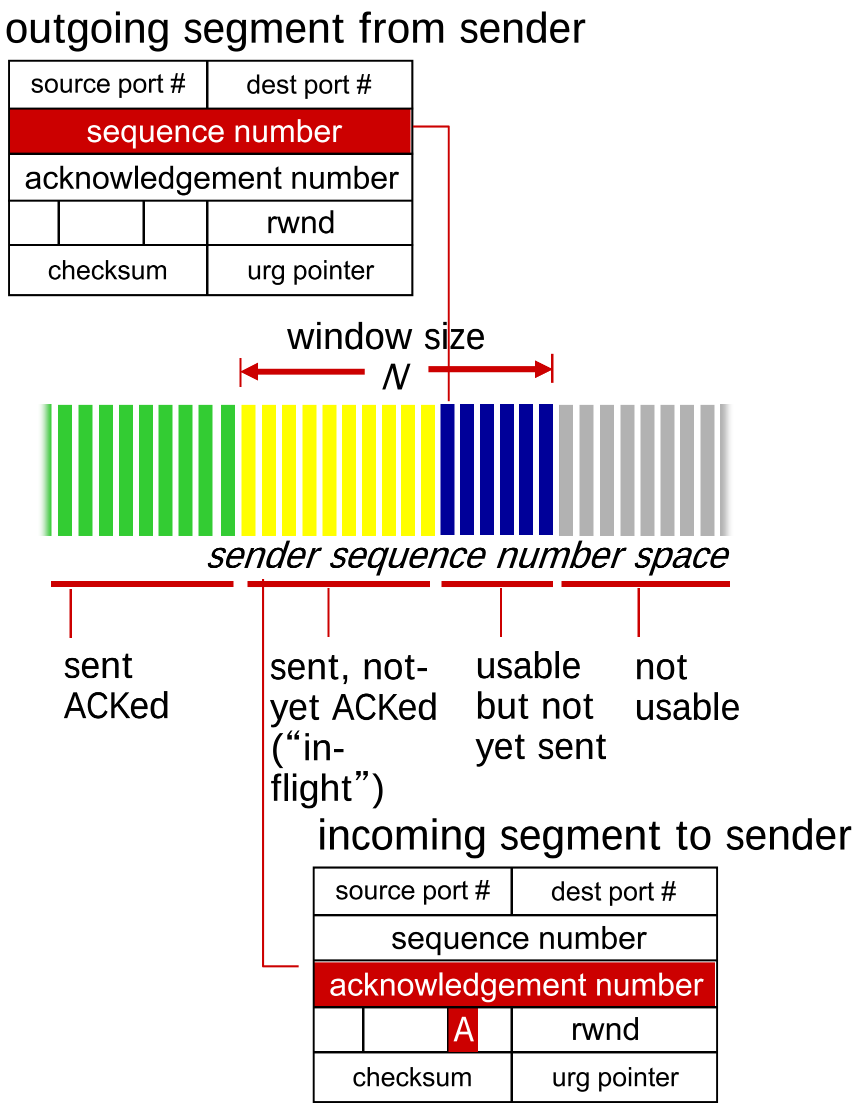


TCP 分组序号和应答序号

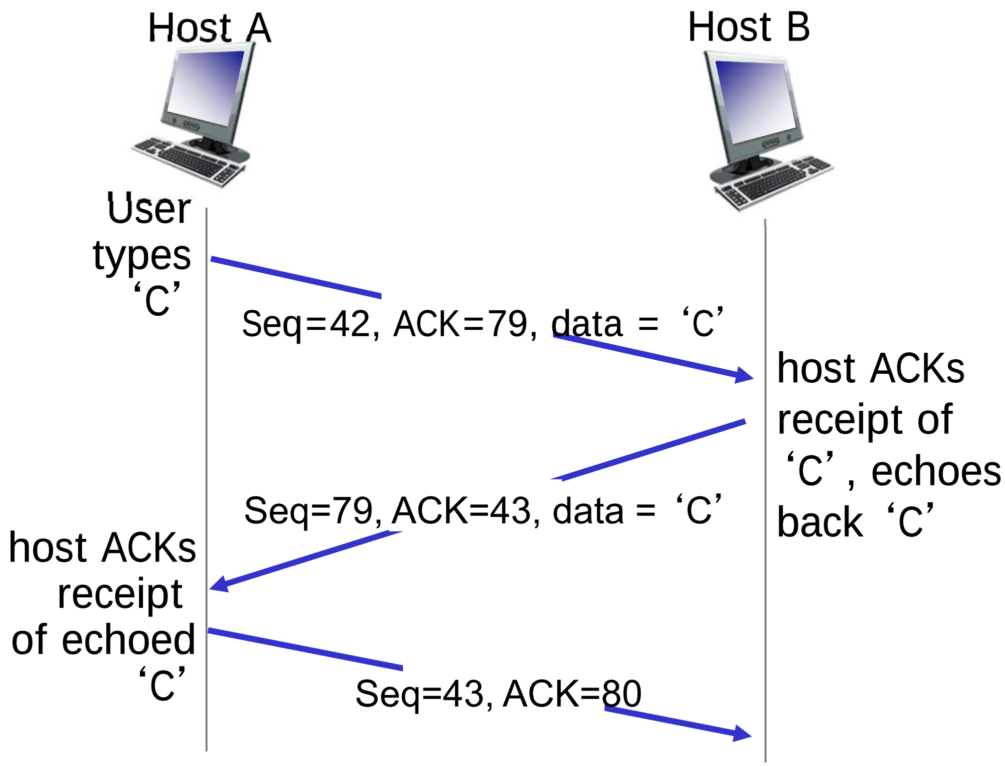
sequence numbers:
- byte stream "number" of first byte in segment's data

acknowledgements:
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments
- A: TCP spec doesn't say, - up to implementor



TCP 分组序号和应答序号



simple telnet scenario

3.5.3 TCP 超时区间

Q: how to set TCP timeout value?

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

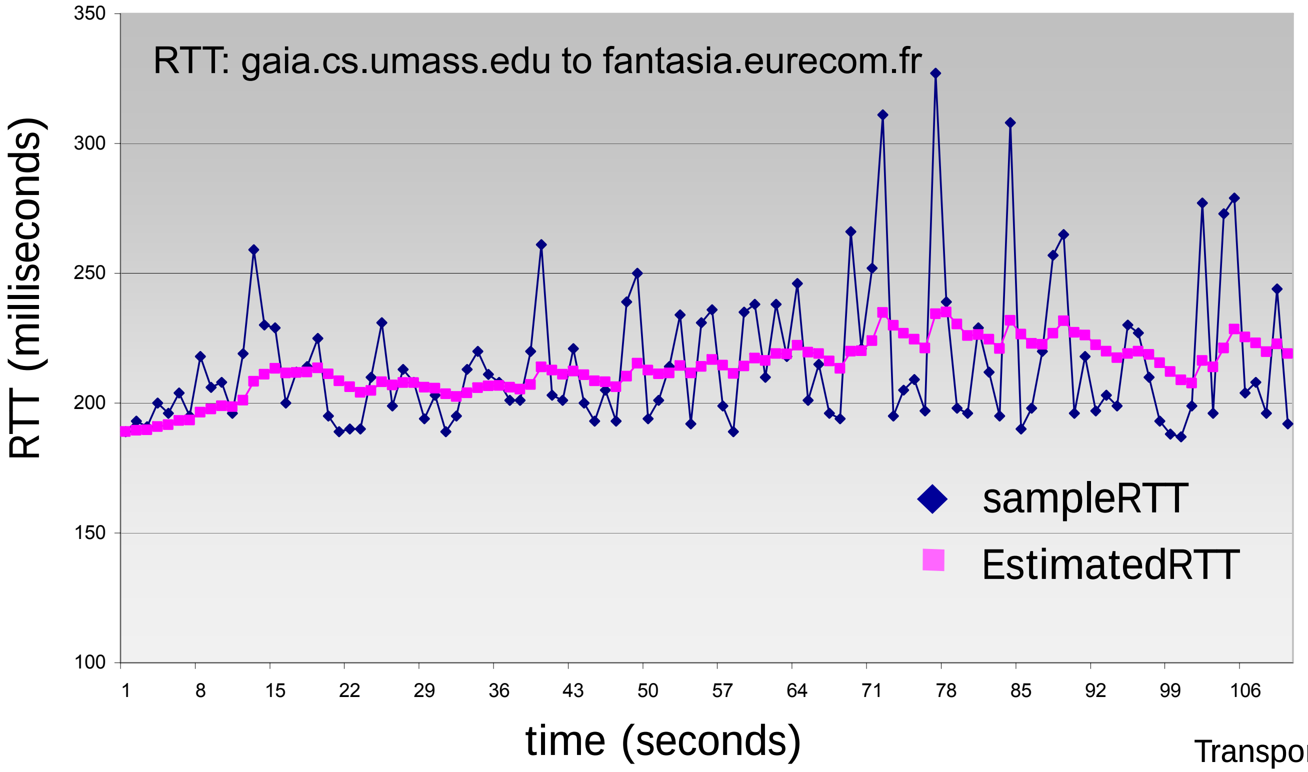
Q: how to estimate RTT?

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

TCP RTT 评估

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

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



TCP RTT 评估

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

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

(typically, $\beta = 0.25$)

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

estimated RTT

"safety margin"

3.5.4 TCP 可靠传输

- TCP creates rdt service on top of IP’s unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

let’s initially consider simplified TCP sender:
- ignore duplicate acks
- ignore flow control, congestion control

TCP sender events:

E1 :收到来自应用层的数据

- 封装成SEGEMEN，加上序号等头信息。
- 序号等于SEGEMENT中第一个字节的顺序数。
- 启动定时器
  - 可以认为定时器对应的是最小序号未应答的SEGMENT
  - 超时区间:

`TimeOutInterval`

E2 :timeout:

- 重传引起超时的SEGMENT
- 重新计时

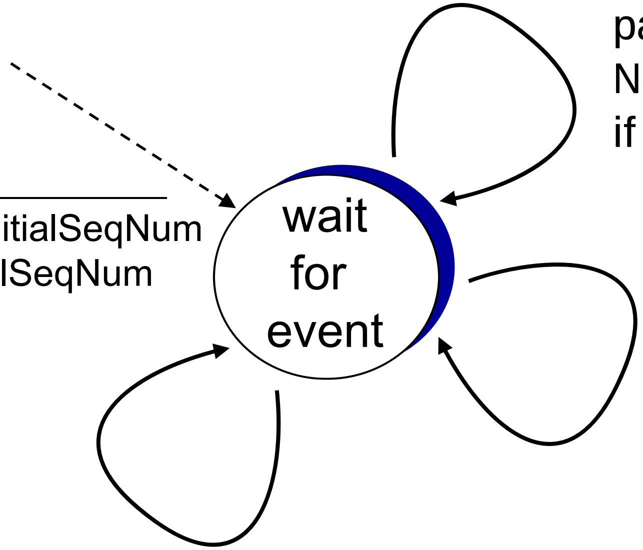
E3 :收到ACK

- 如果是应答未被应答SEGMENT的ACK:
  - 更新操作
  - 如果还有未被应答的SEGMENT，重新定时

TCP 发送方（简化）

Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum



data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., "send")
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

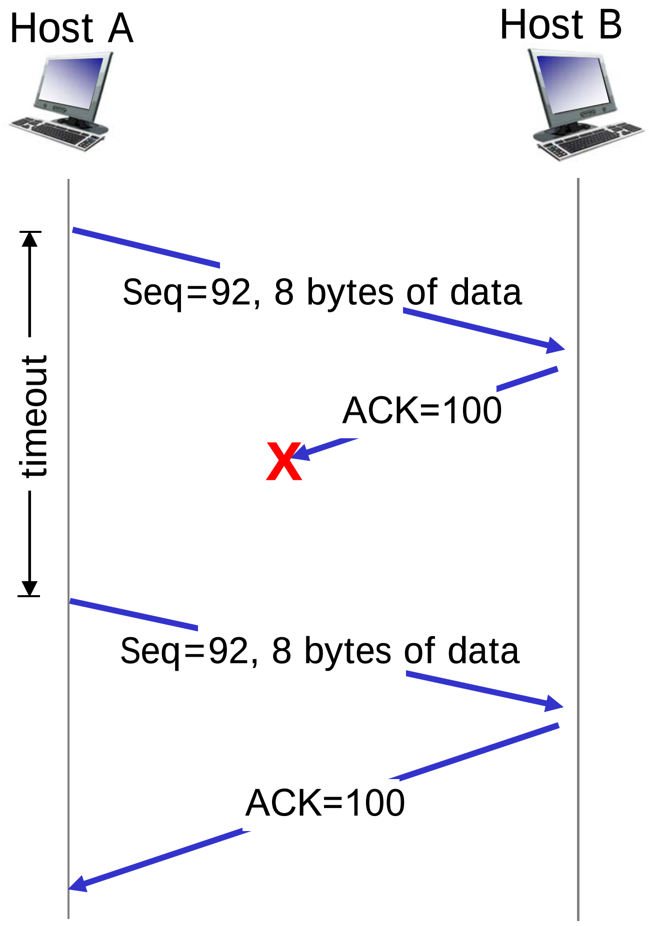
timeout

retransmit not-yet-acked segment
    with smallest seq. #
start timer

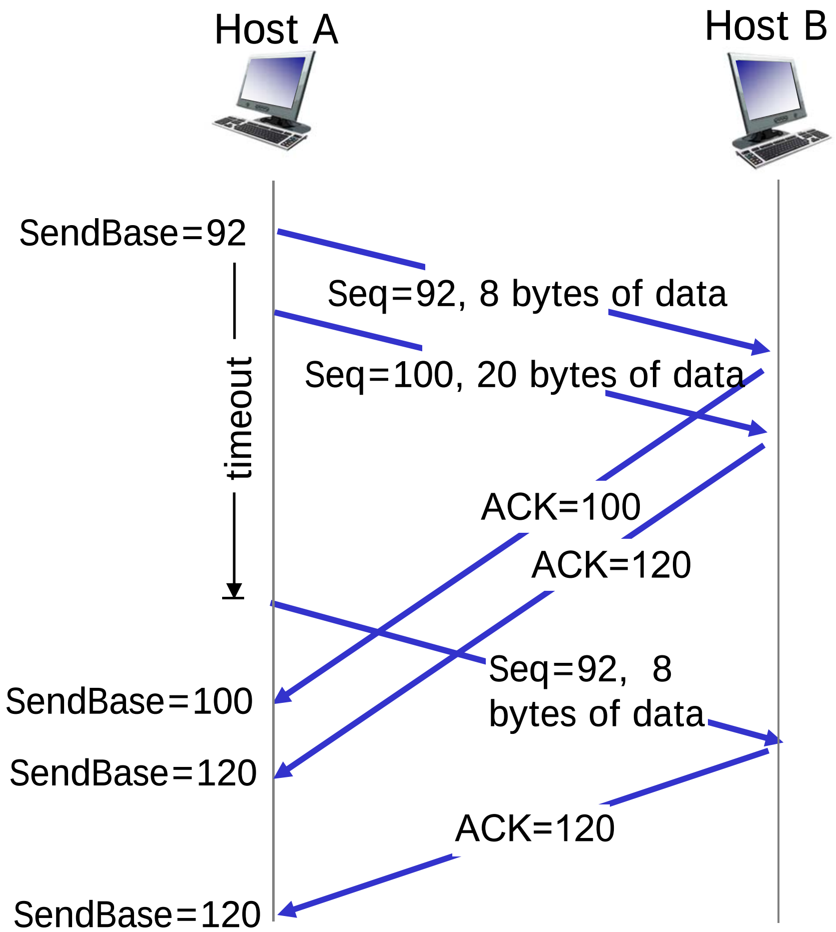
ACK received, with ACK field value y

```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
       start timer
     else stop timer
 }
```

TCP: 重传例1和例2

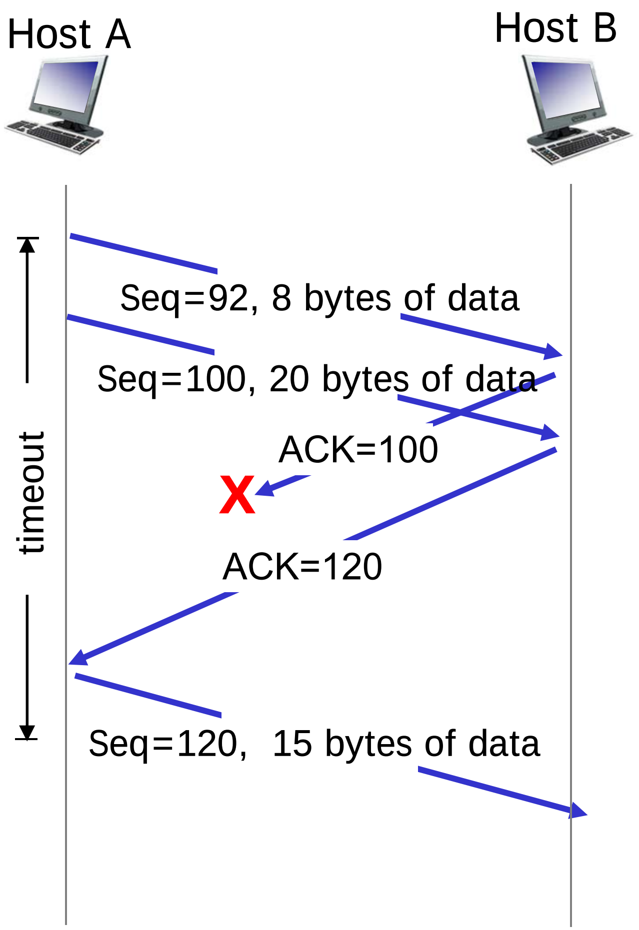


lost ACK scenario



premature timeout

TCP: 例3



cumulative ACK

TCP 接收方 [RFC 1122, RFC 2581]

event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments （见前例3）
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK,* indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

TCP 快速重传

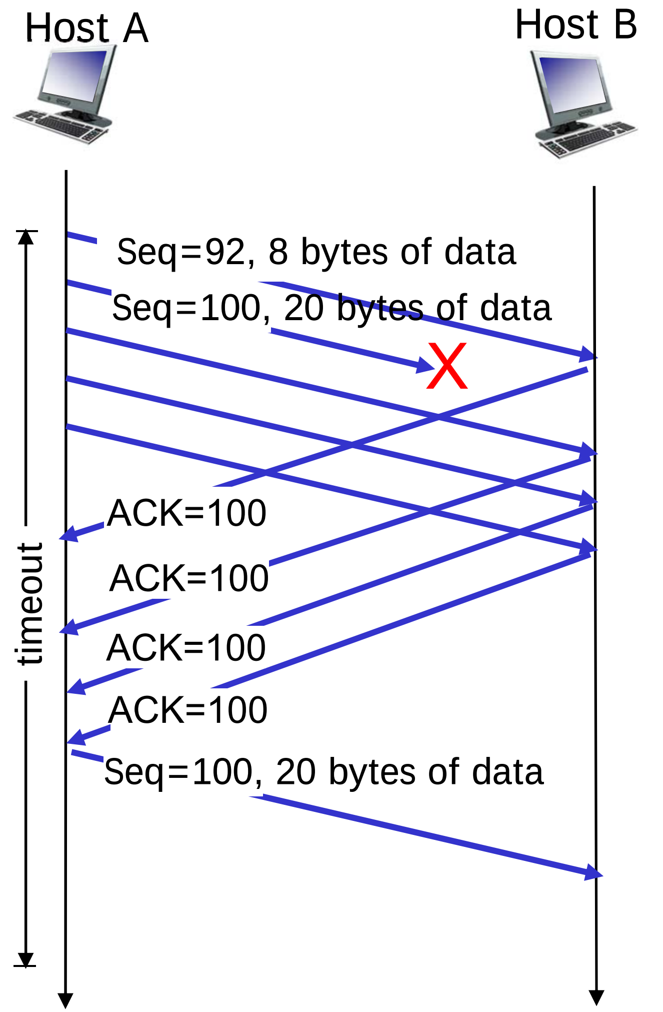
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives 3 ACKs for same data

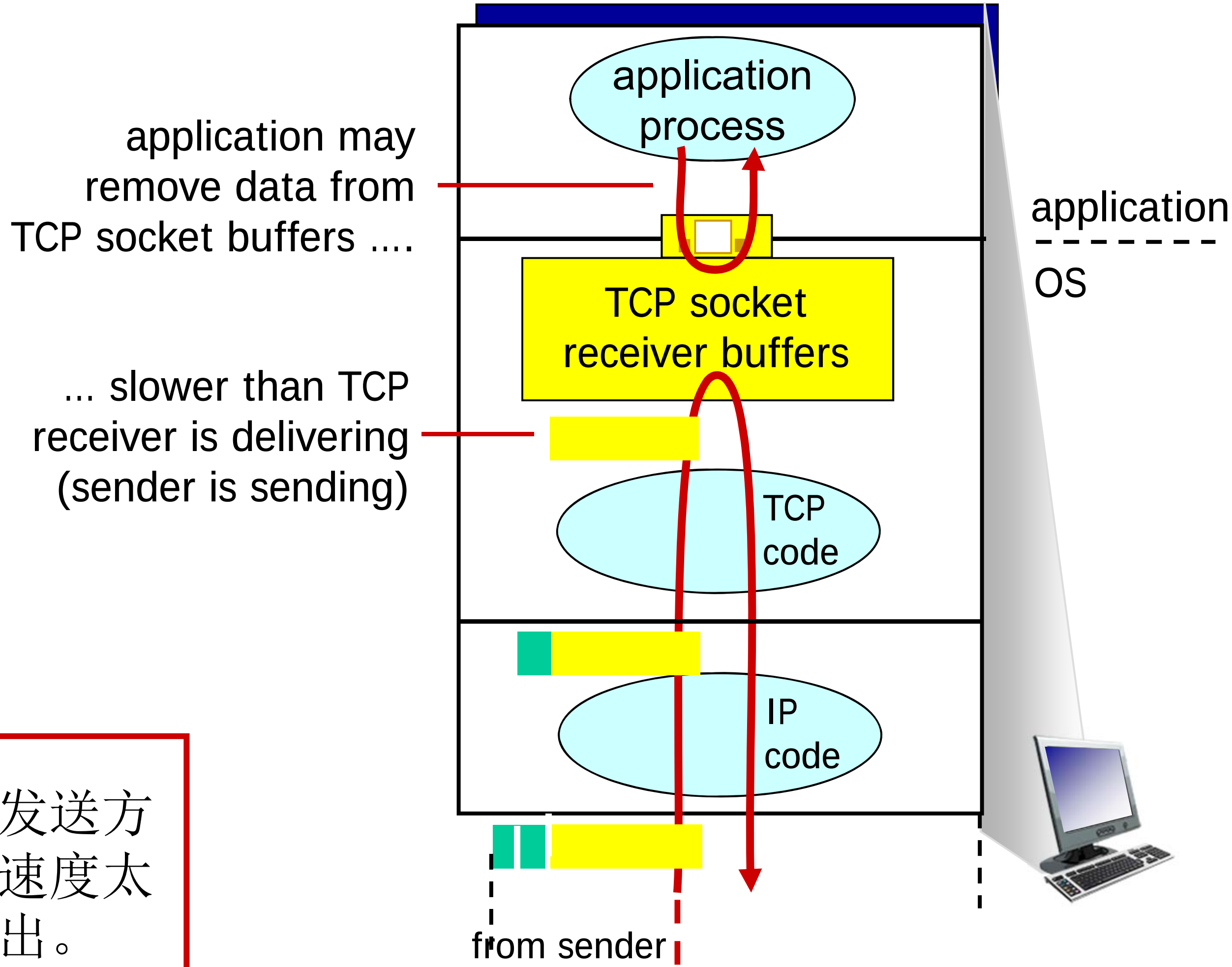
- resend unacked segment with smallest seq #
  - likely that unacked segment lost, so don't wait for timeout

TCP 快速重传举例



fast retransmit after sender receipt of triple duplicate ACK

TCP 流控制



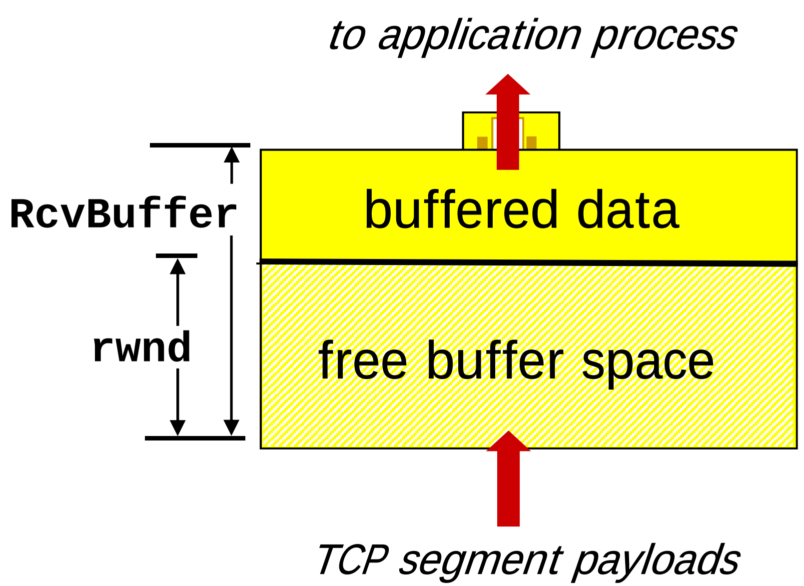
receiver protocol stack

flow control

接收方通过发送消息给发送方控制其速度。避免发送速度太快导致其接收缓冲区溢出。

TCP 流控制

- receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments
  - **RcvBuffer** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **RcvBuffer**
- sender limits amount of unacked ("in-flight") data to receiver's **rwnd** value
- guarantees receive buffer will not overflow



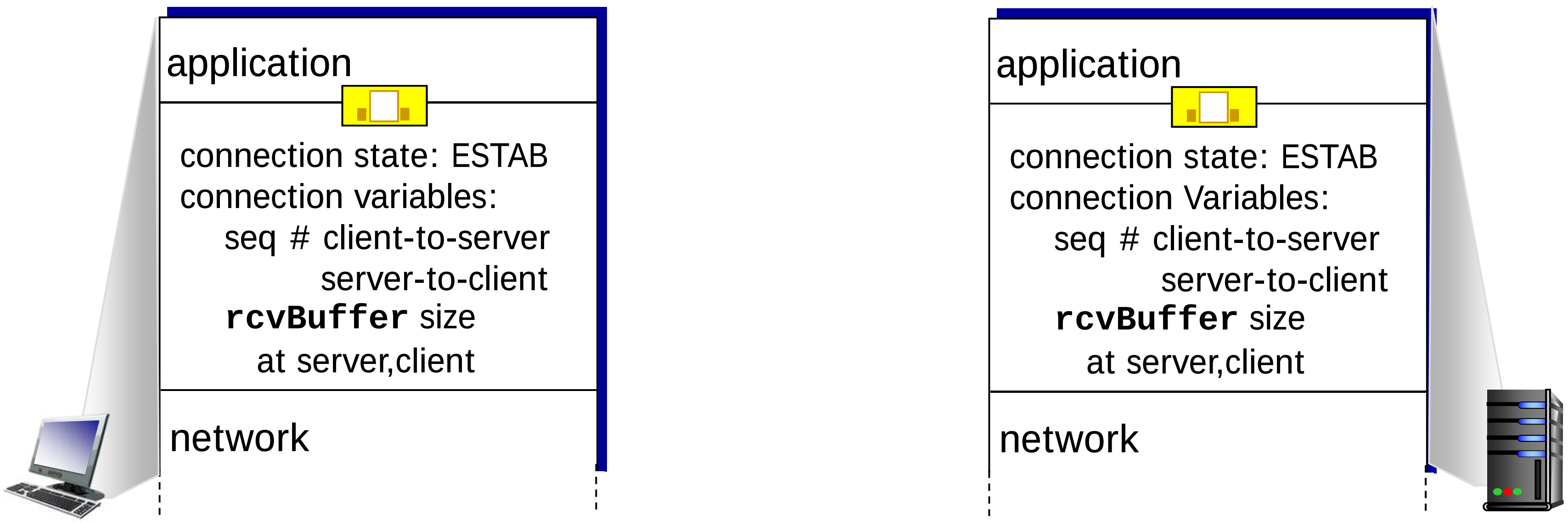
receiver-side buffering

Q:如果**B**告知**Arwnd=0**后，**B**又清空了缓冲区，如何让**A**及时知晓？

TCP 连接管理

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

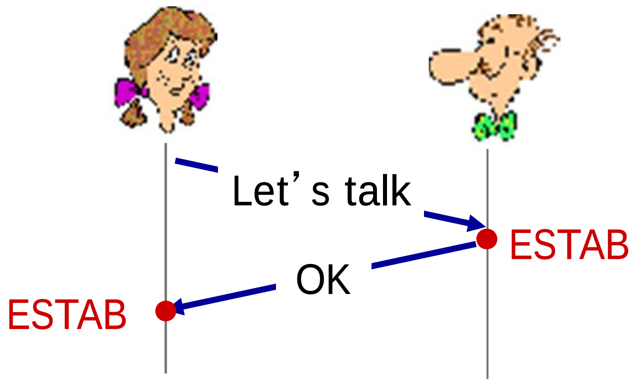


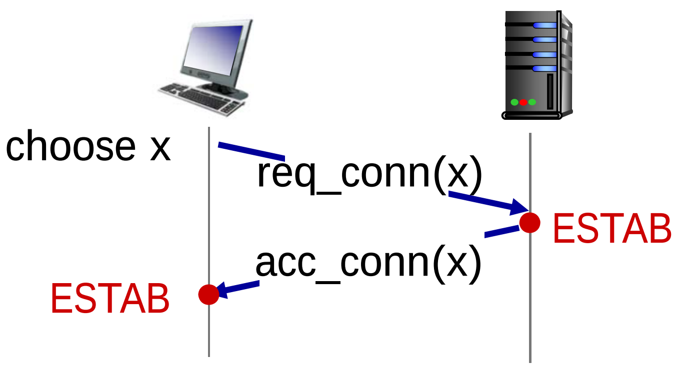

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

```
Socket connectionSocket =
   welcomeSocket.accept();
```

TCP 连接建立

2-way handshake:





Q: will 2-way handshake always work in network?

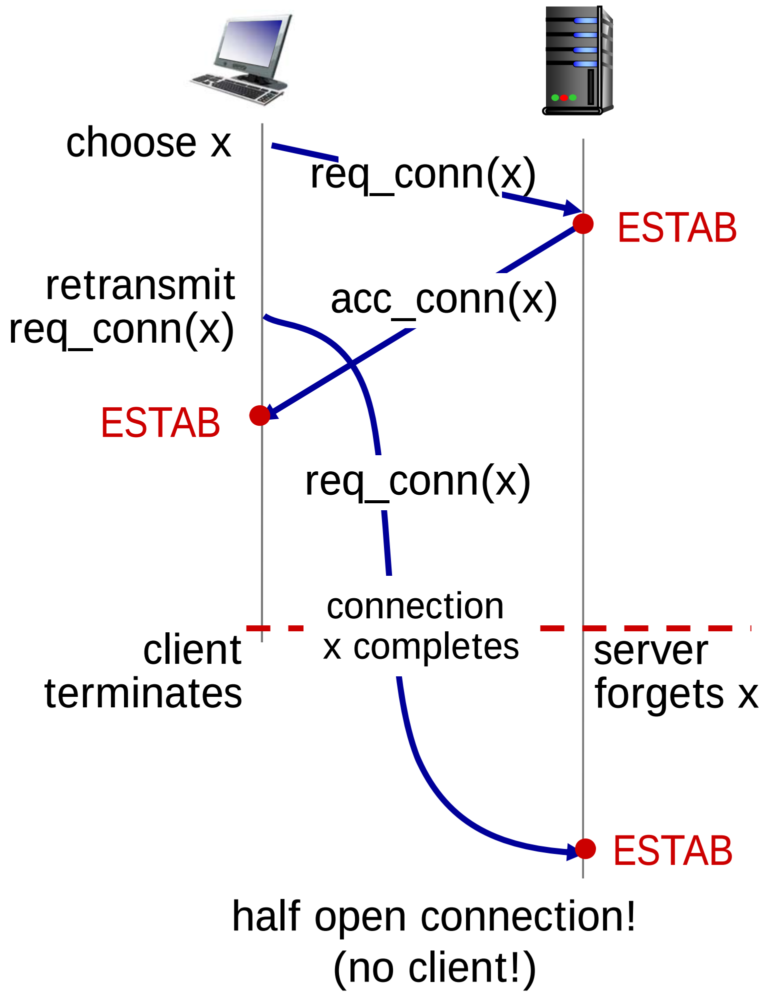
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can’t “see” other side

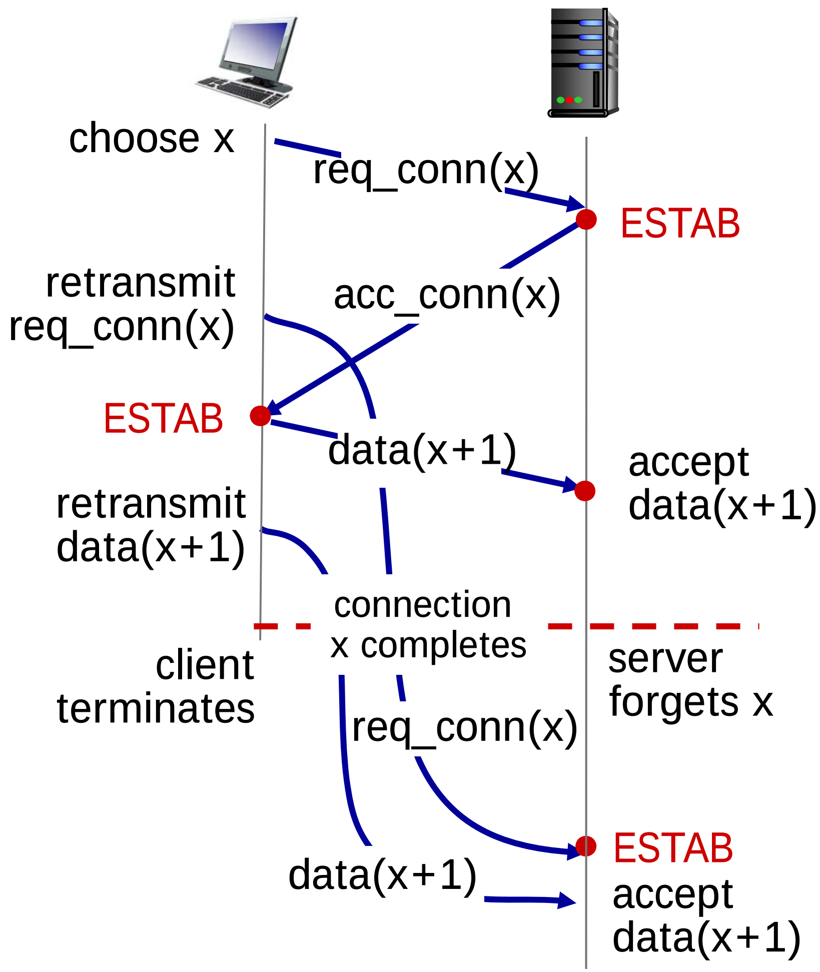
思考:

为什么要进行三次而不是二次握手 ?

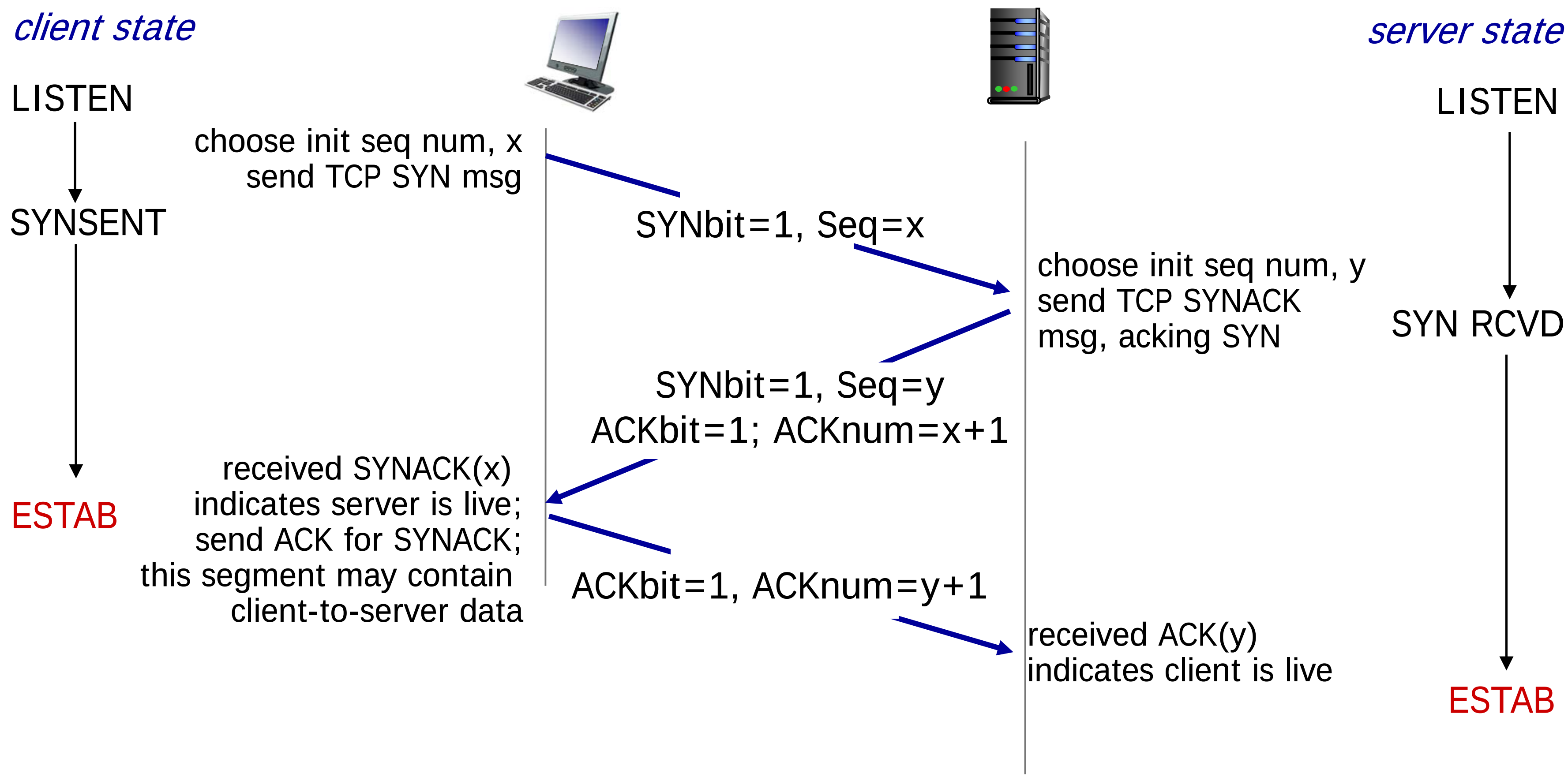
建立连接

2-way handshake failure scenarios:



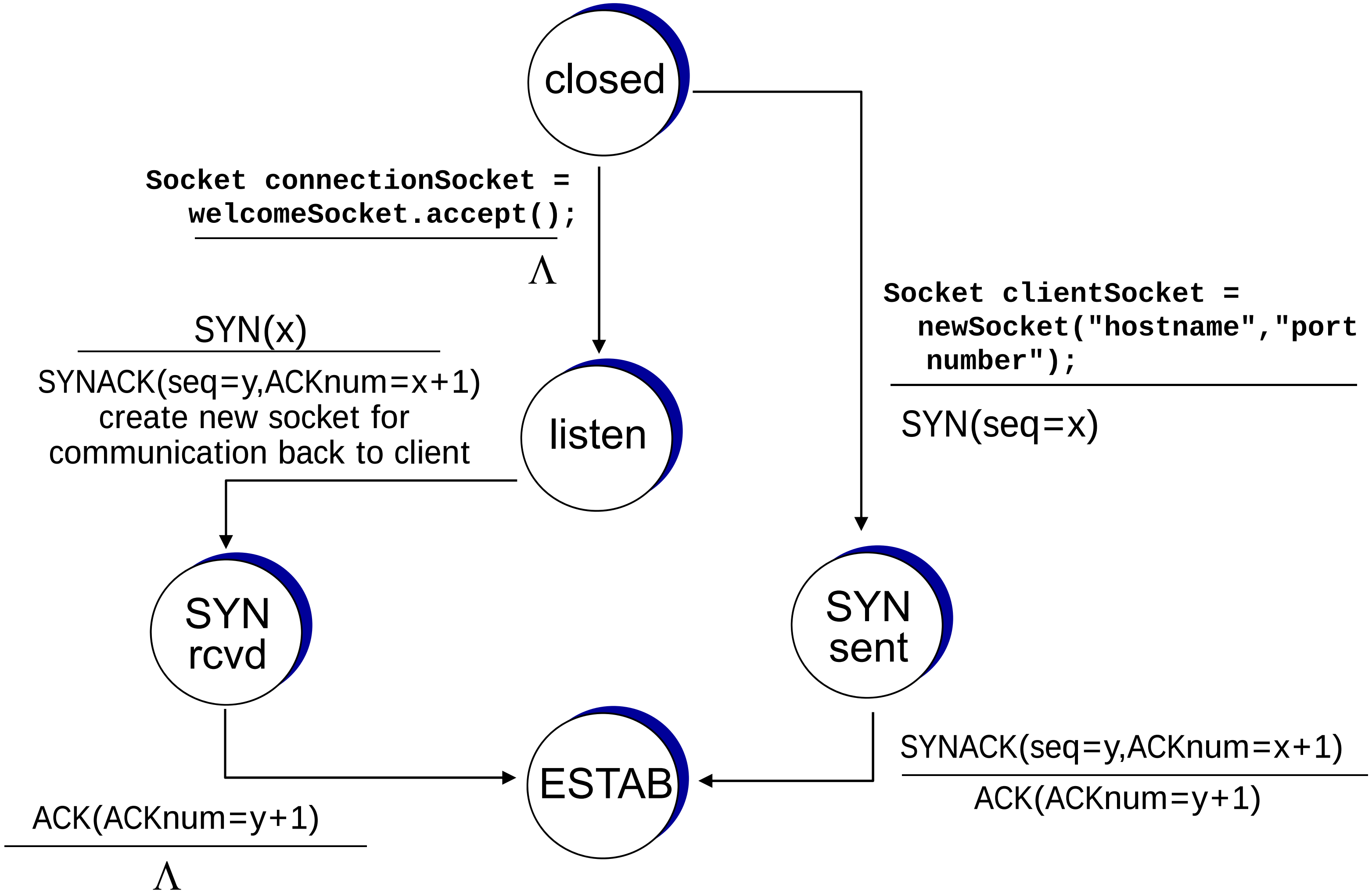


TCP 三次握手



client state
server state
LISTEN
LISTEN
choose init seq num, x
send TCP SYN msg
SYNSENT
SYNbit=1, Seq=x
choose init seq num, y
send TCP SYNACK
msg, acking SYN
SYN RCVD
SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1
received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data
ESTAB
ACKbit=1, ACKnum=y+1
received ACK(y)
indicates client is live
ESTAB

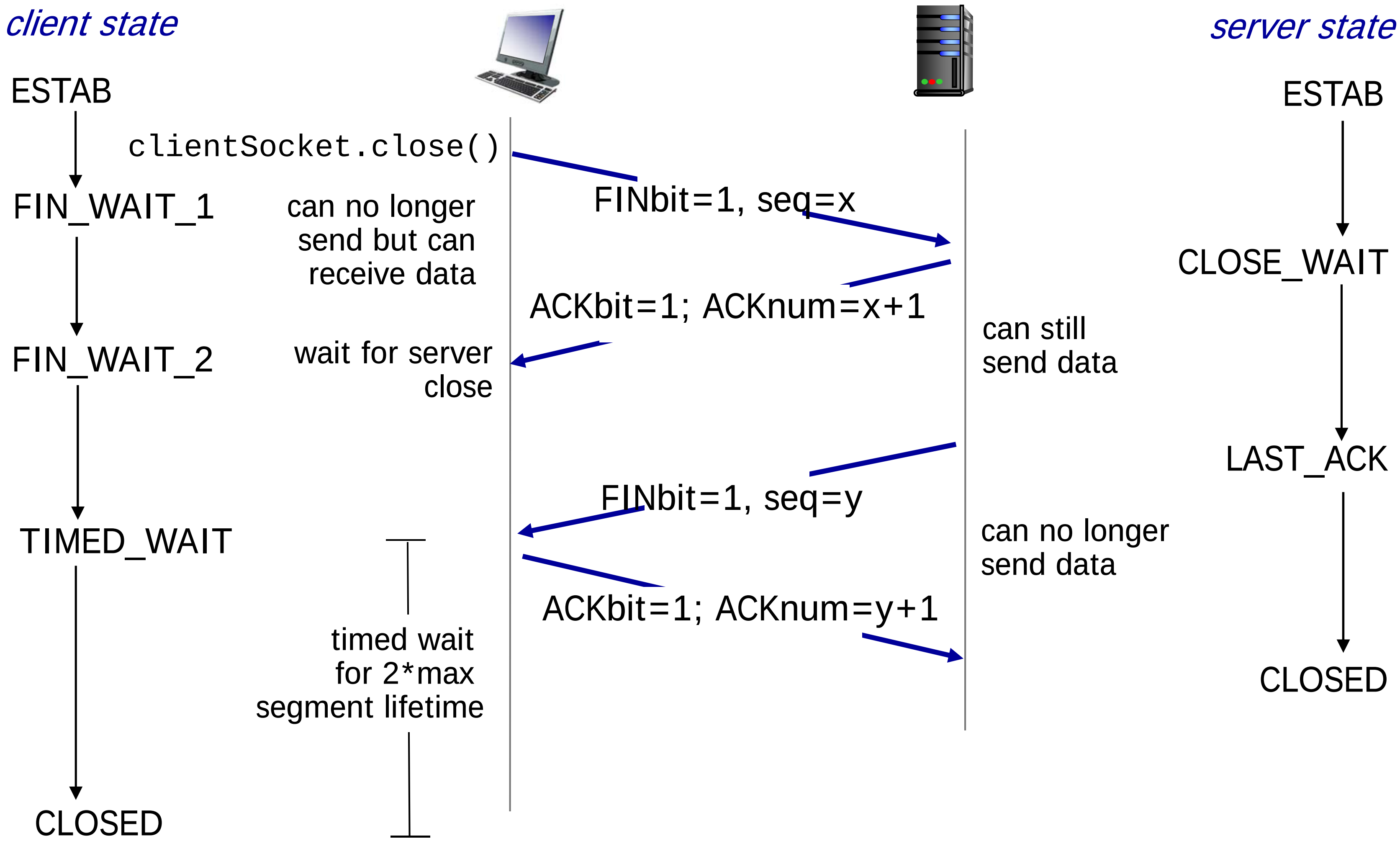
TCP三次握手 FSM


closed
Socket connectionSocket = welcomeSocket.accept();
Λ
listen
SYN(x)
SYNACK(seq=y,ACKnum=x+1)
create new socket for communication back to client
SYN rcvd
ACK(ACKnum=y+1)
Λ
ESTAB
Socket clientSocket = newSocket("hostname","port number");
SYN(seq=x)
SYN sent
SYNACK(seq=y,ACKnum=x+1)
ACK(ACKnum=y+1)

TCP：关闭连接

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP：关闭连接举例



client state
server state
ESTAB
ESTAB
clientSocket.close()
FIN_WAIT_1
can no longer send but can receive data
FINbit=1, seq=x
CLOSE_WAIT
ACKbit=1; ACKnum=x+1
can still send data
FIN_WAIT_2
wait for server close
LAST_ACK
FINbit=1, seq=y
TIMED_WAIT
can no longer send data
ACKbit=1; ACKnum=y+1
timed wait for 2*max segment lifetime
CLOSED
CLOSED

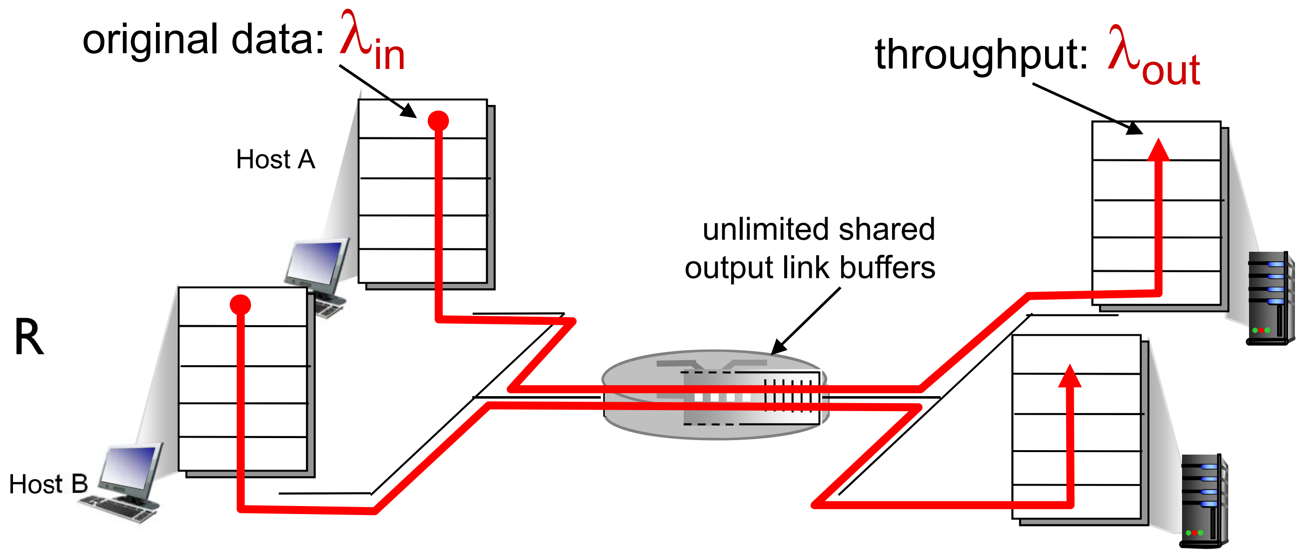
3.6 拥塞控制原理

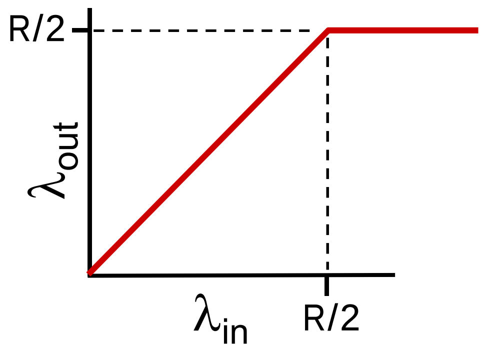
congestion:

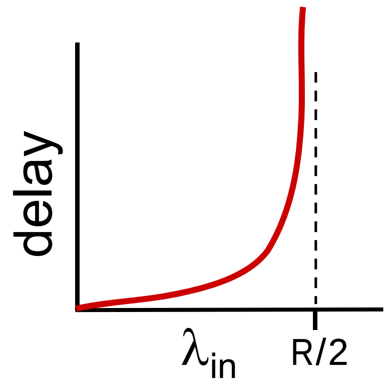
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

拥塞控制原因和代价：场景 1

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



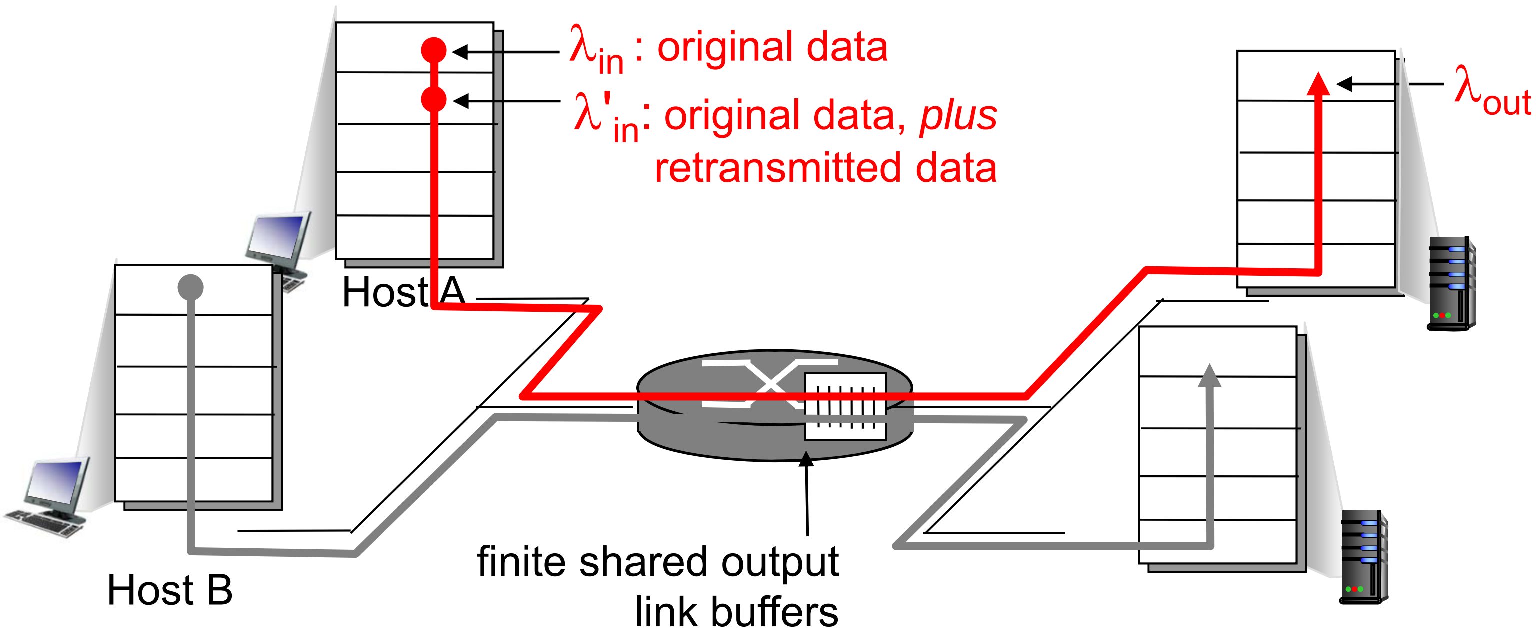




- maximum per-connection throughput: R/2
- large delays as arrival rate, λ_{in}, approaches capacity

拥塞控制原因和代价: 场景2

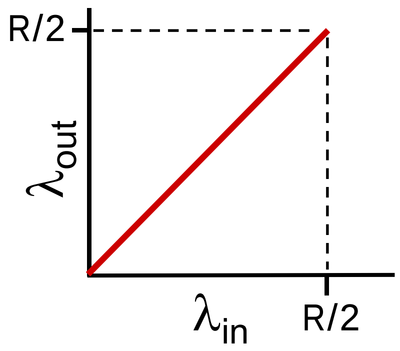
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

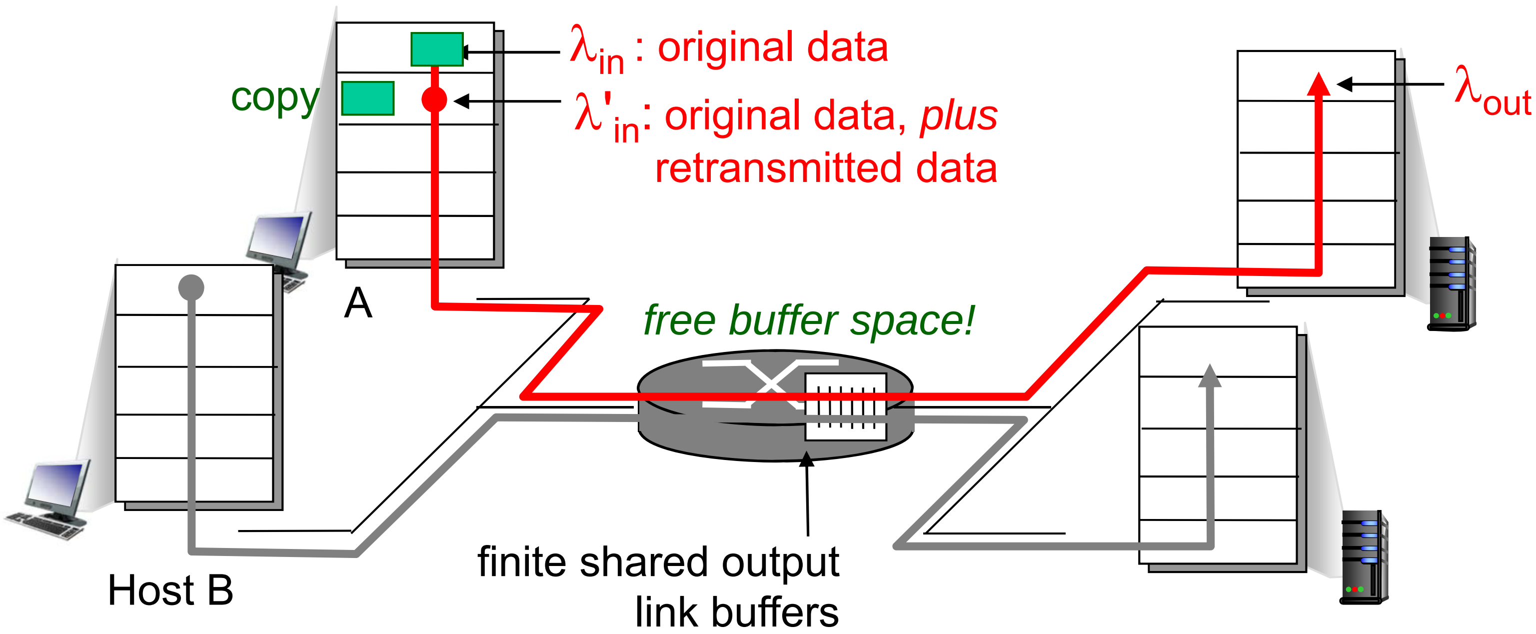


拥塞控制原因和代价: 场景2

idealization: perfect knowledge

- sender sends only when router buffers available

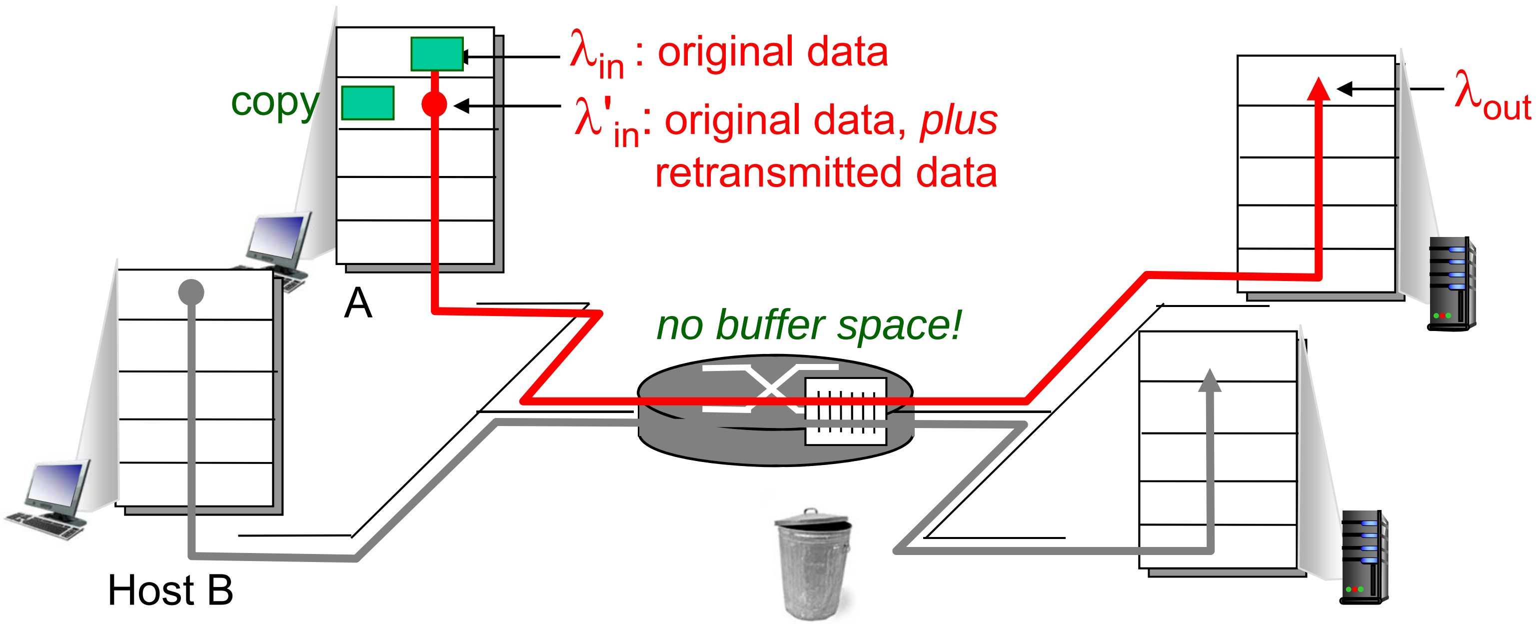




拥塞控制原因和代价: 场景2

Idealization: known loss packets can be lost, dropped at router due to full buffers

- sender only resends if packet *known* to be lost

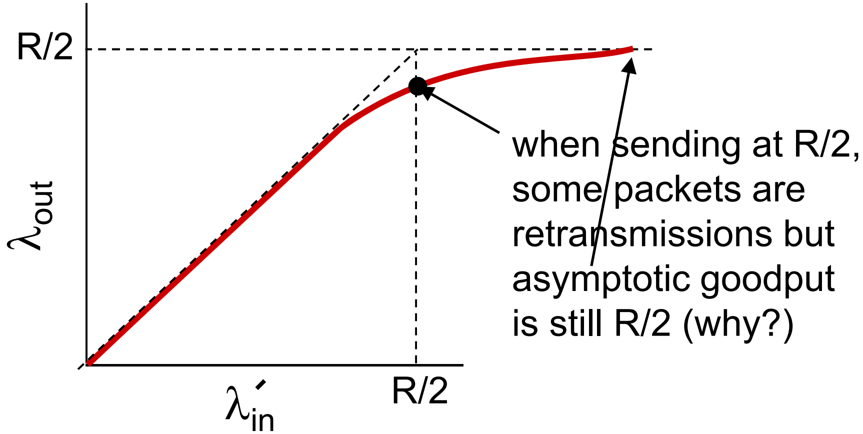


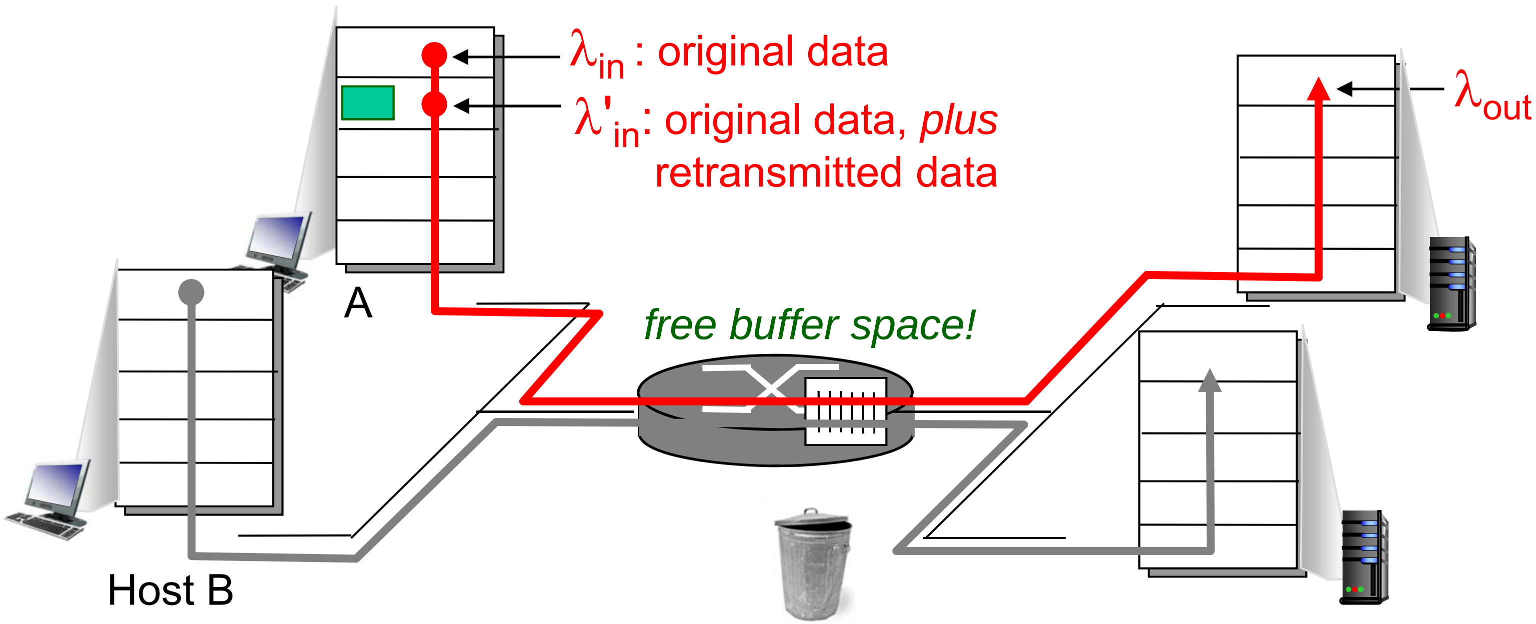
拥塞控制原因和代价: 场景2

Idealization: known loss

packets can be lost, dropped at router due to full buffers

- sender only resends if packet *known* to be lost

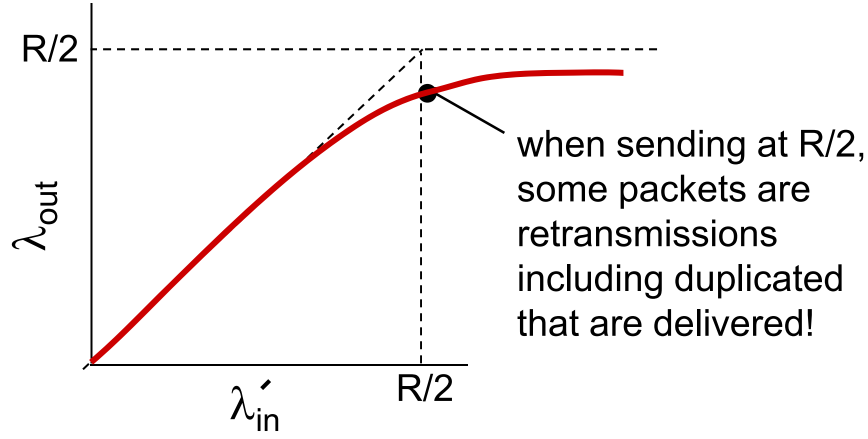


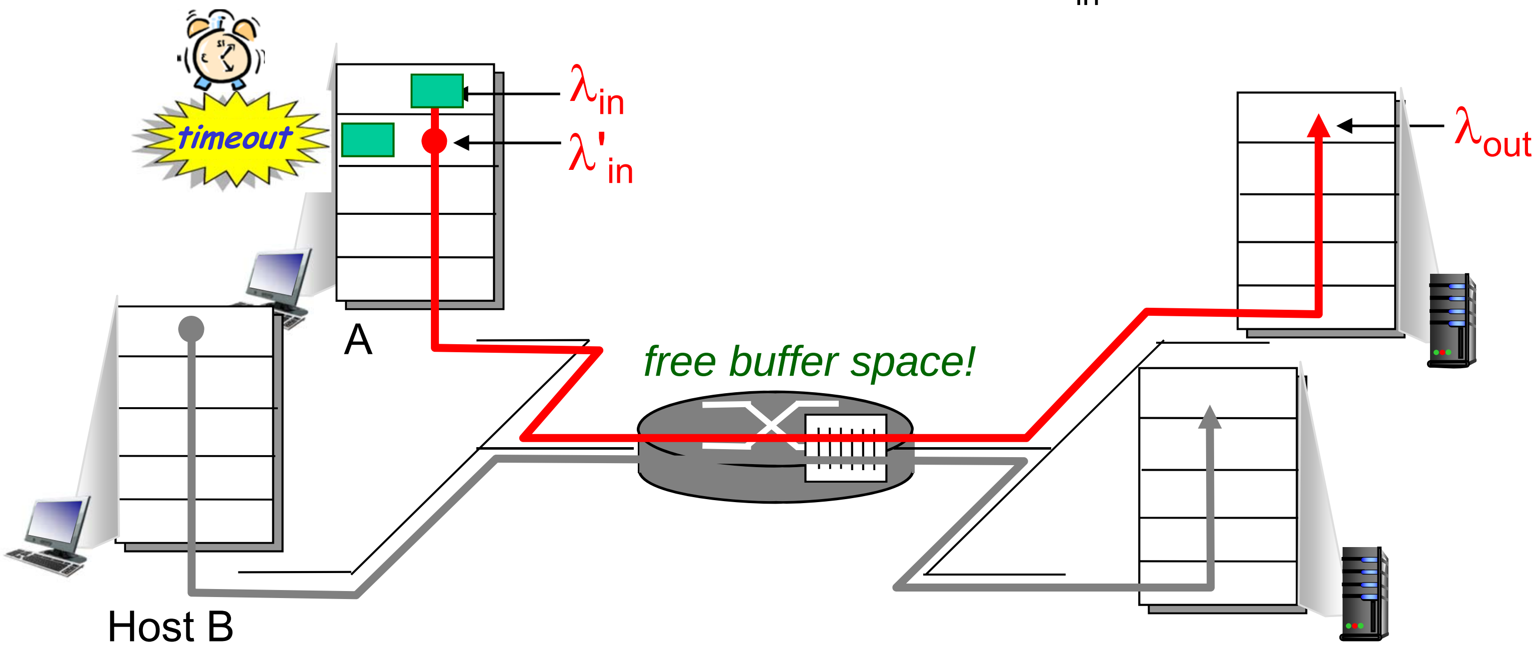


拥塞控制原因和代价: 场景2

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

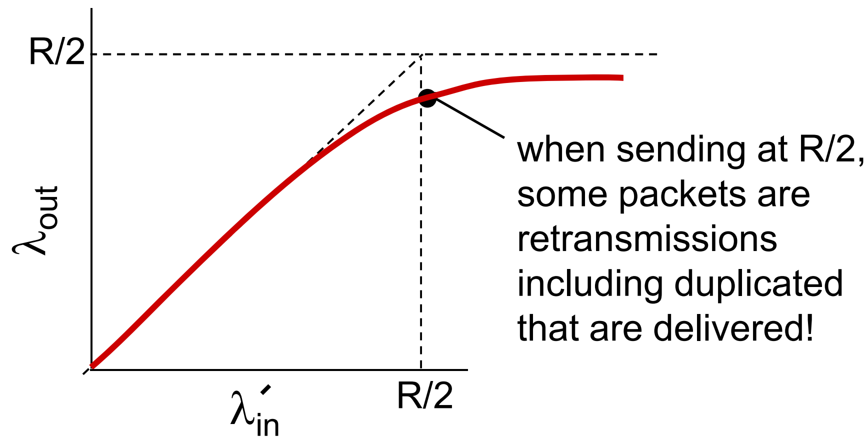




拥塞控制原因和代价: 场景2

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



"costs" of congestion:

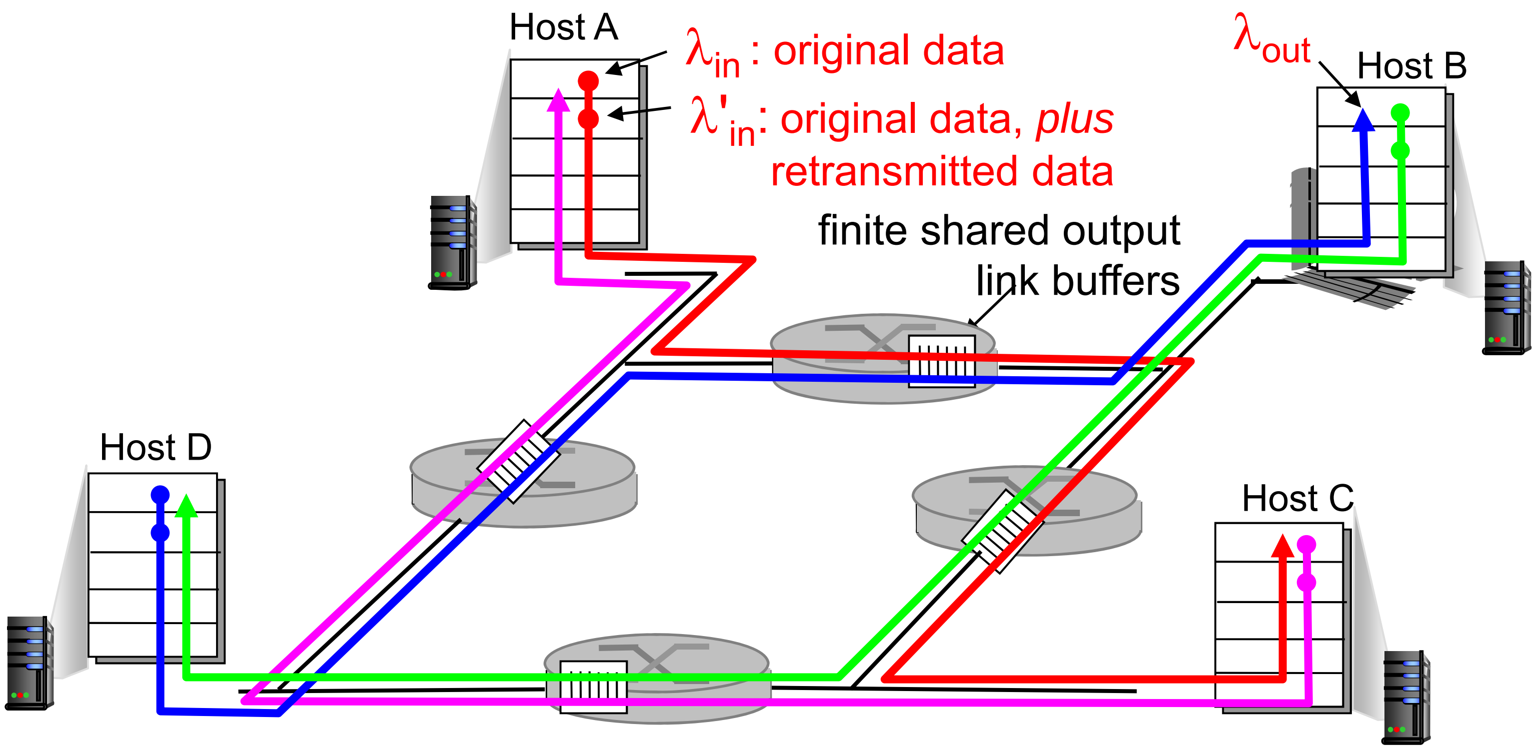
- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

拥塞控制原因和代价: 场景3

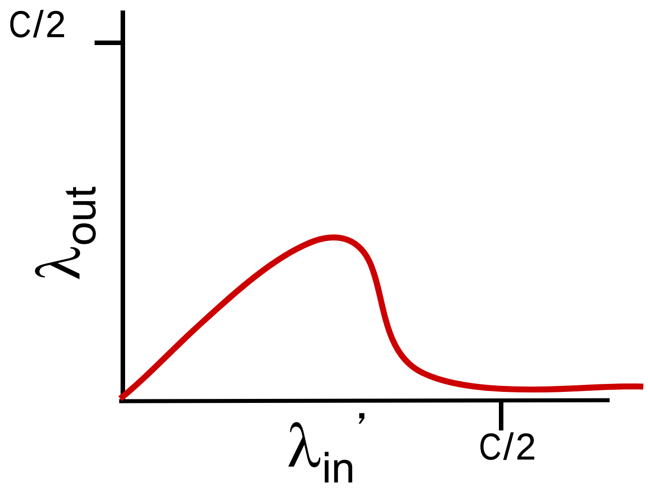
- four senders
- multihop paths
- timeout/retransmit

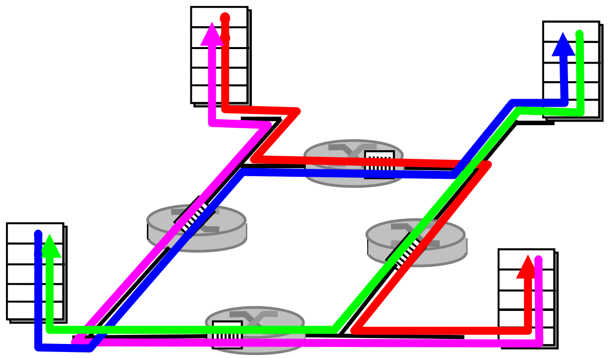
Q: what happens as λ_{in} and λ_{in}' increase ?

A: as red λ_{in}' increases, all arriving blue pkts at upper queue are dropped, blue throughput $\rightarrow$ 0



拥塞控制原因和代价: 场景3





another "cost" of congestion:

- when packet dropped, any "upstream transmission capacity used for that packet was wasted!

3.6.2 拥塞控制发方法

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 (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate for sender to send at

3.6.3 ATM ABR congestion control （略）

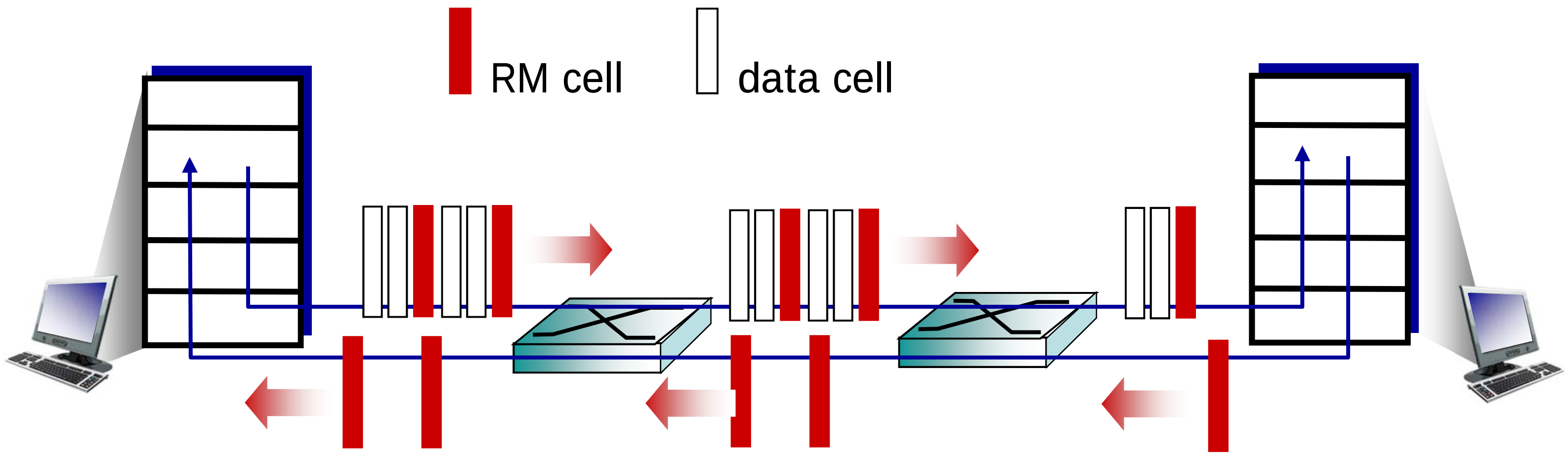
ABR: available bit rate:

- "elastic service"
- if sender's path "underloaded":
  - sender should use available bandwidth
- if sender's path congested:
  - sender throttled to minimum guaranteed rate

RM (resource management) cells:

- sent by sender, interspersed with data cells
- bits in RM cell set by switches ("*network-assisted*")
  - *NI bit:* no increase in rate (mild congestion)
  - *CI bit:* congestion indication
- RM cells returned to sender by receiver, with bits intact

Case study: ATM ABR congestion control

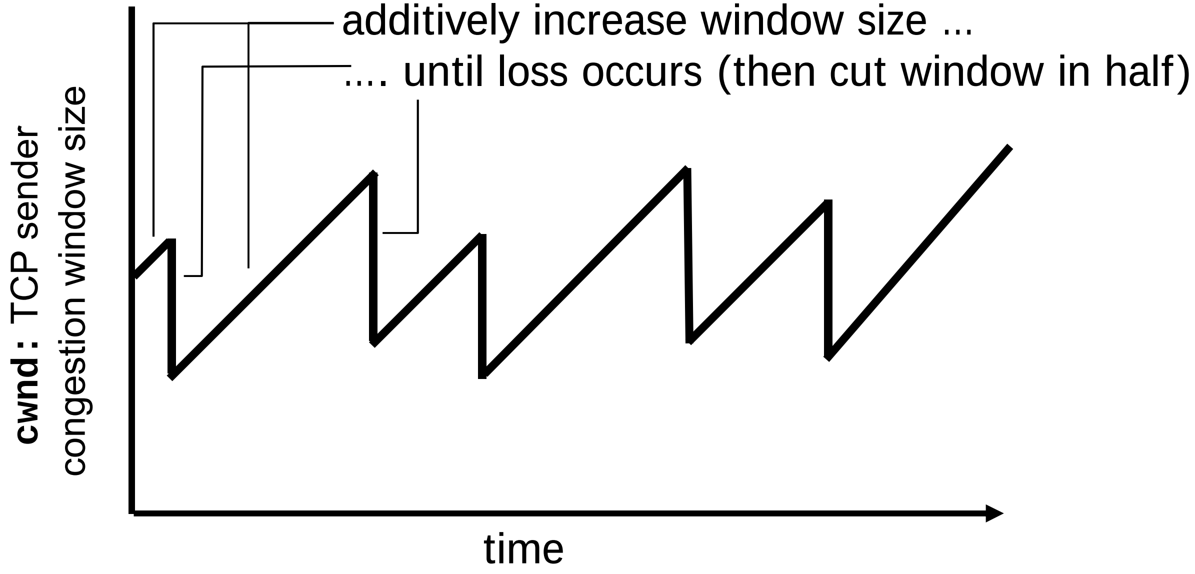


- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - senders' send rate thus max supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell

3.7 TCP 拥塞控制：AIMD（加法增，乘法减）

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs, then decreases the window size
  - *additive increase:* increase **cwnd** by 1 MSS every RTT until loss detected
  - *multiplicative decrease:* cut **cwnd** in half after loss

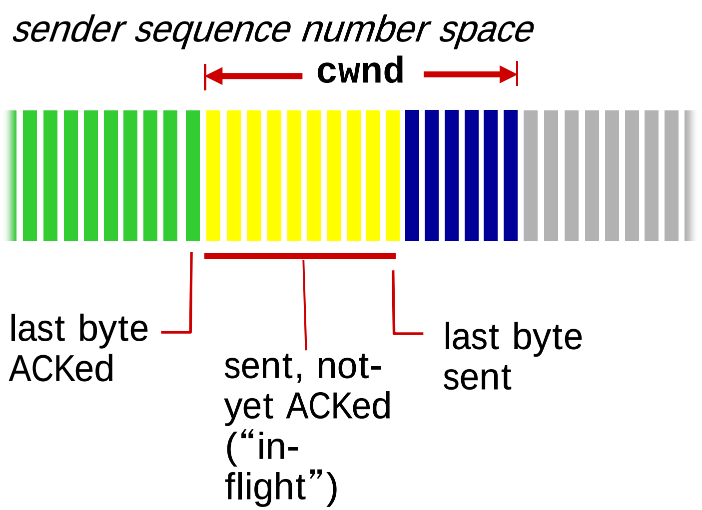
AIMD saw tooth behavior: probing for bandwidth



cwnd :

拥塞窗口

TCP 拥塞控制



- sender limits transmission:

$$\texttt{LastByteSent-LastByteAcked} \le \texttt{Min(cwnd, rwnd)}$$

- **cwnd** is dynamic, function of perceived network congestion

TCP sending rate:

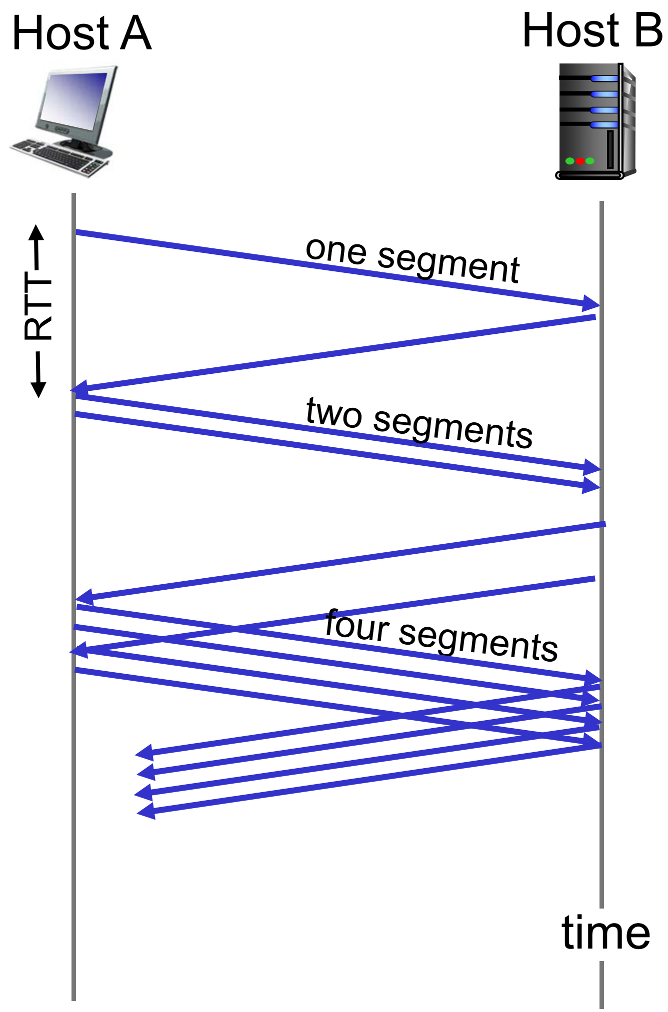
- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

TCP Slow Start

- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- *summary:*

初始速率低，但以指数增长快。



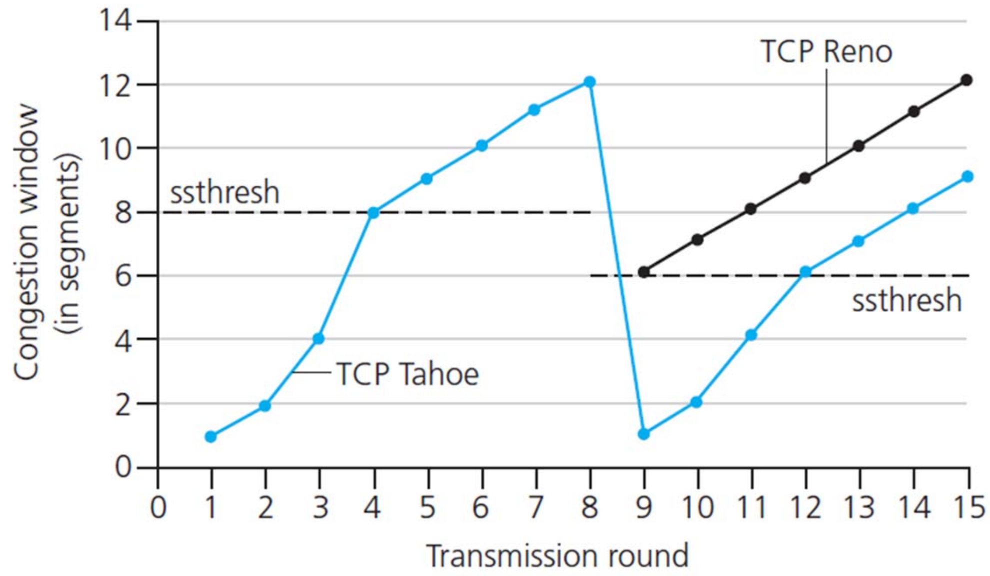
TCP 对丢包的反应和措施

- 超时引起的事件：
  - **cwnd** set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- 三个重复ACK引发的事件（TCP RENO）
  - dup ACKs indicate network capable of delivering some segments
  - **cwnd** is cut in half window then grows linearly
- TCP Tahoe always sets **cwnd** to 1 (timeout or 3 duplicate acks)

TCP: 冲突避免

Q: when should the exponential increase switch to linear?

A: when **cwnd** gets to 1/2 of its value before timeout.

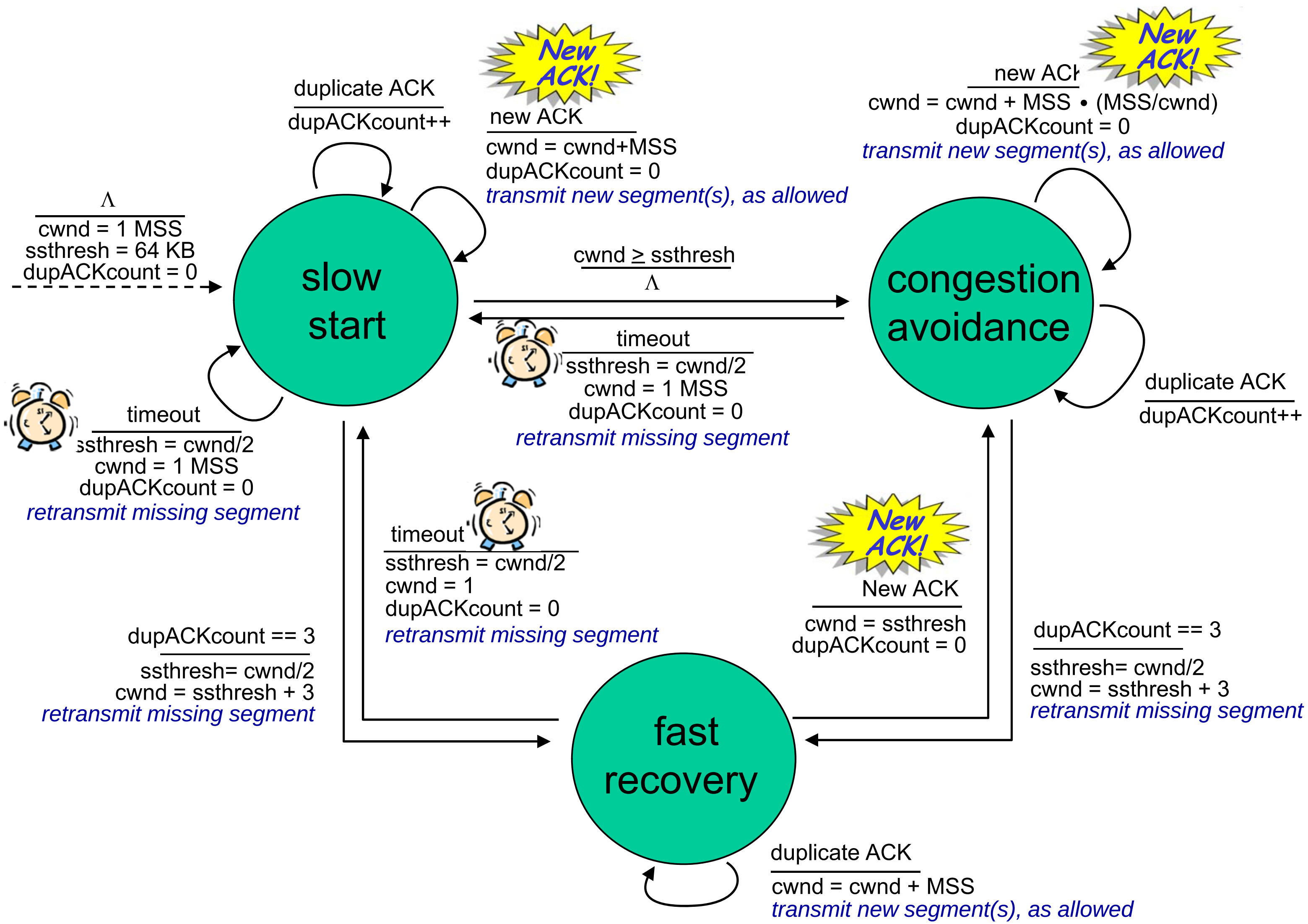


Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

Q: 第0个回合，CWND=？

Summary: TCP Congestion Control

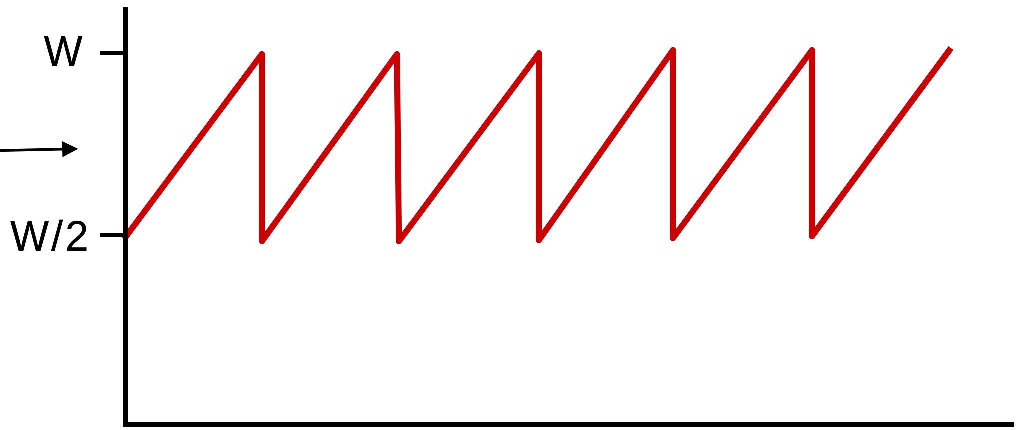


TCP 吞吐量

- avg. TCP thruput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

$$\text{avg TCP thruput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$

平均为0.75



TCP 未来

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires W = 83,333 in-flight segments
- throughput in terms of segment loss probability, L [Mathis 1997]:

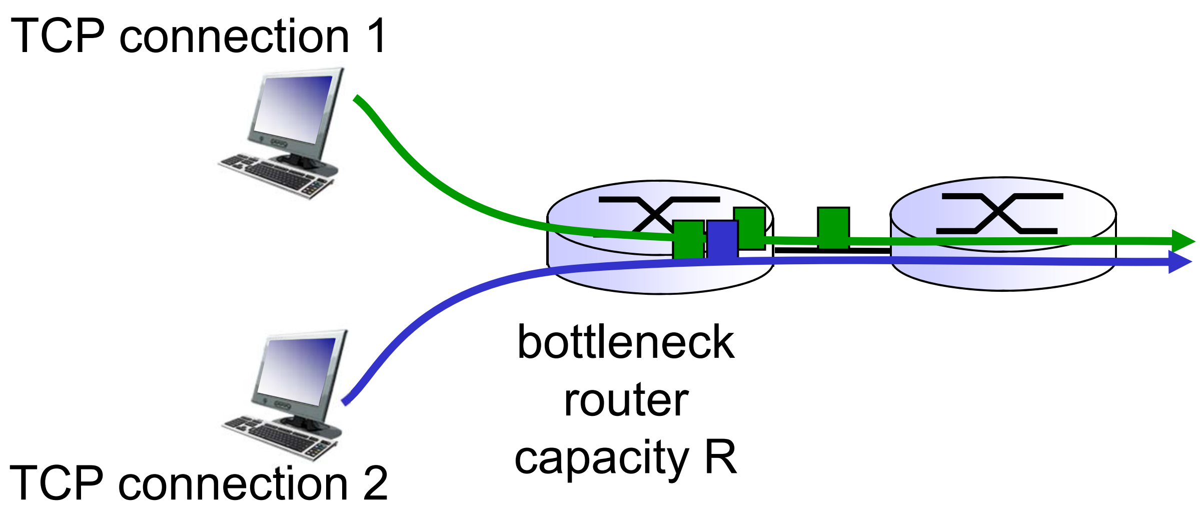
$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT}\sqrt{L}}$$

➜ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$ *– a very small loss rate!*

- new versions of TCP for high-speed

TCP 公平性

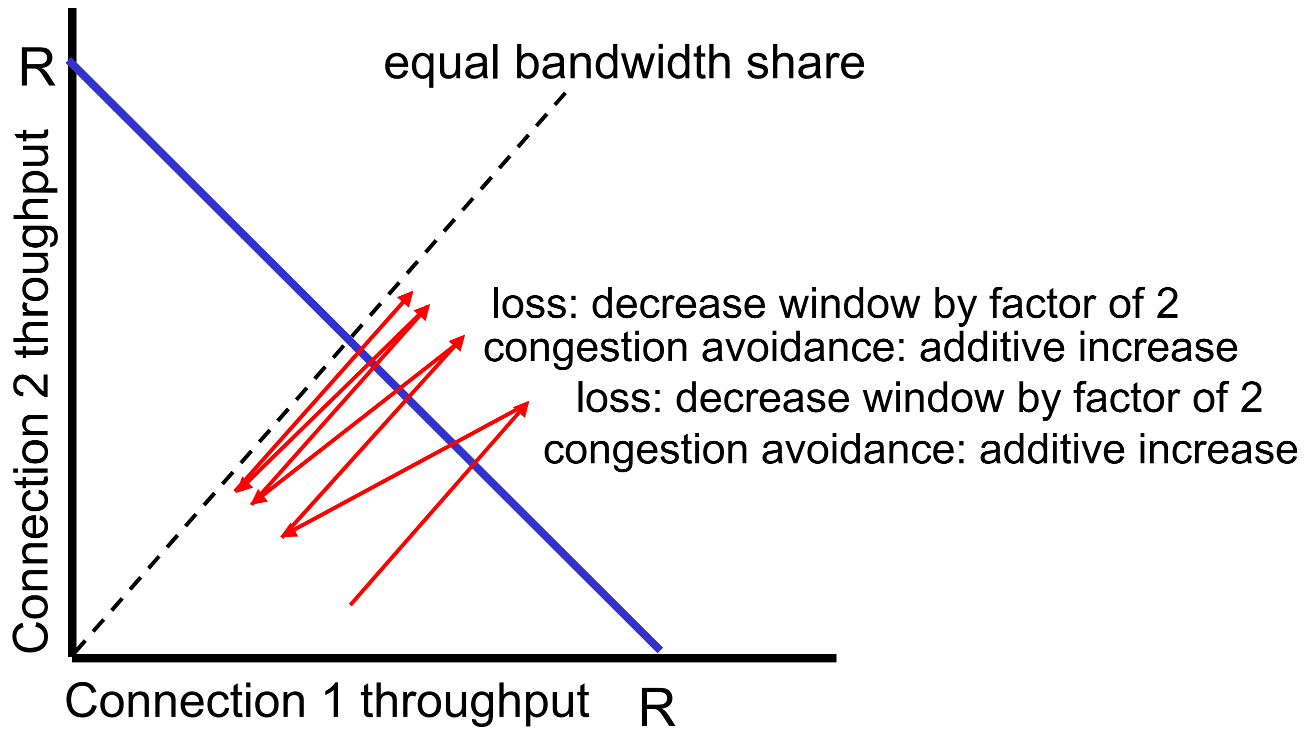
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



TCP 的公平性问题

two competing sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



TCP 的公平性问题

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

- application can open multiple parallel connections between two hosts
- web browsers do this
- e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

本章小结

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

next:

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