Chapter 4 Transport Layer

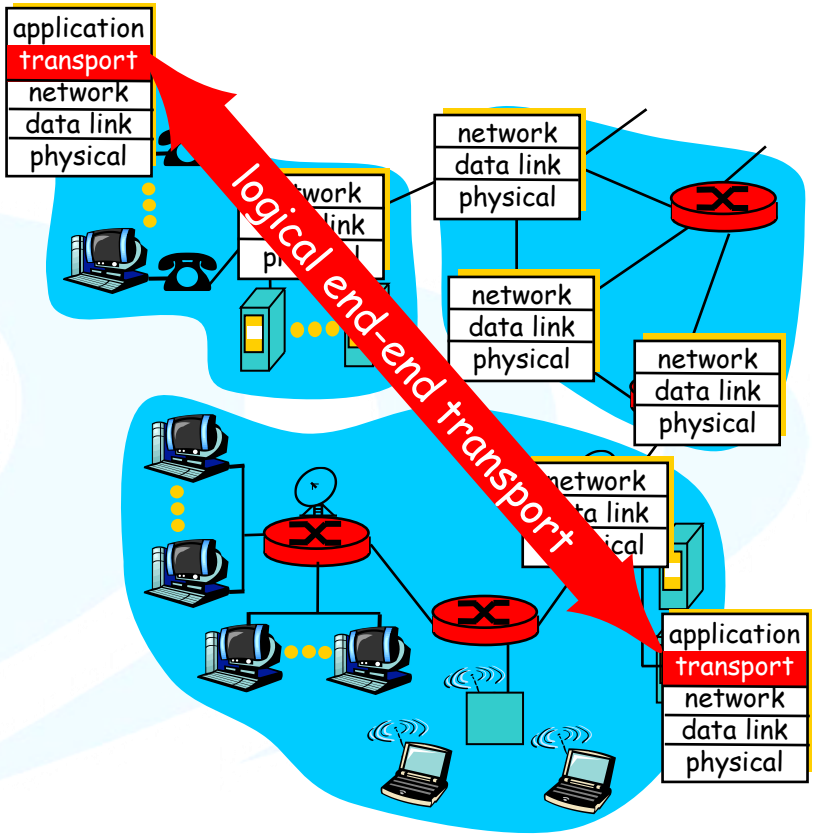
Our goals:

- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

4.1 Transport-layer services

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - 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



Transport vs. network layer

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

Household analogy:

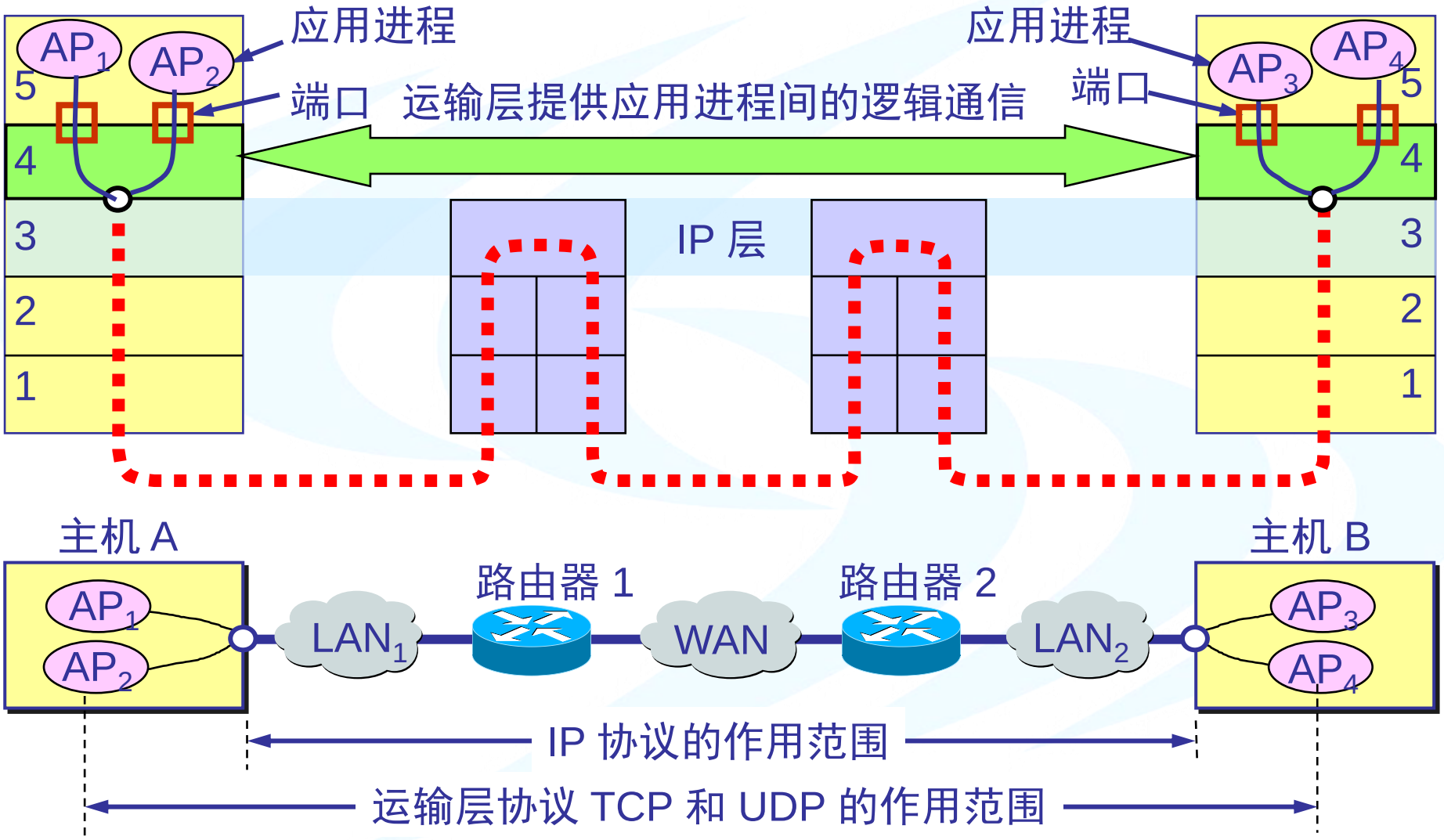
12 kids sending letters to 12 kids

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

进程之间的通信

- 从通信和信息处理的角度看，运输层向它上面的应用层提供通信服务，它属于面向通信部分的最高层，同时也是用户功能中的最低层。
- 当网络的边缘部分中的两个主机使用网络的核心部分的功能进行端到端的通信时，只有位于网络边缘部分的主机的协议栈才有运输层，而网络核心部分中的路由器在转发分组时都只用到下三层的功能。

运输层为相互通信的应用进程提供了逻辑通信



应用进程之间的通信

- 两个主机进行通信实际上就是两个主机中的应用进程互相通信。
- 应用进程之间的通信又称为端到端的通信。
- 运输层的一个很重要的功能就是复用和分用。应用层不同进程的报文通过不同的端口向下交到运输层，再往下就共用网络层提供的服务。
- “运输层提供应用进程间的逻辑通信”。“逻辑通信”的意思是：运输层之间的通信好像是沿水平方向传送数据。但事实上这两个运输层之间并没有一条水平方向的物理连接。

运输层协议和网络层协议的主要区别

运输层的主要功能

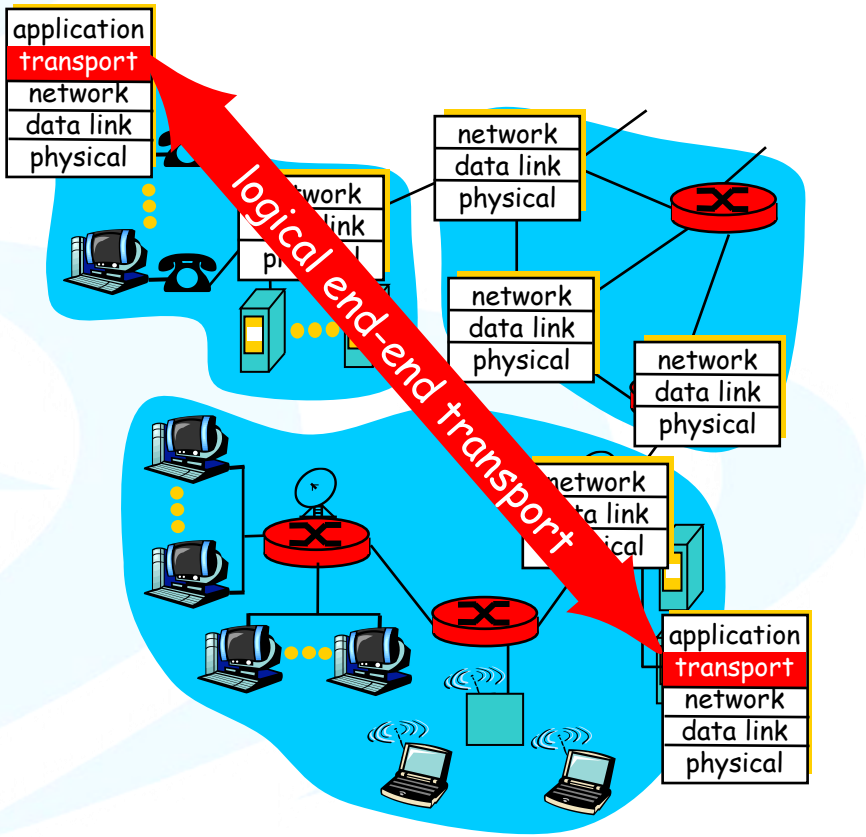
- 运输层为应用进程之间提供端到端的逻辑通信（但网络层是为主机之间提供逻辑通信）。
- 运输层还要对收到的报文进行差错检测。
- 运输层需要有两种不同的运输协议，即面向连接的 **TCP** 和无连接的 **UDP**。

两种不同的运输协议

- 运输层向高层用户屏蔽了下面网络核心的细节（如网络拓扑、所采用的路由选择协议等），它使应用进程看见的就是好像在两个运输层实体之间有一条端到端的逻辑通信信道。
- 当运输层采用面向连接的 **TCP** 协议时，尽管下面的网络是不可靠的（只提供尽最大努力服务），但这种逻辑通信信道就相当于一条全双工的可靠信道。
- 当运输层采用无连接的 **UDP** 协议时，这种逻辑通信信道是一条不可靠信道。

Internet transport-layer protocols

- **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**



运输层的两个主要协议

TCP/IP 的运输层有两个不同的协议：

(1) 用户数据报协议 UDP
(User Datagram Protocol)

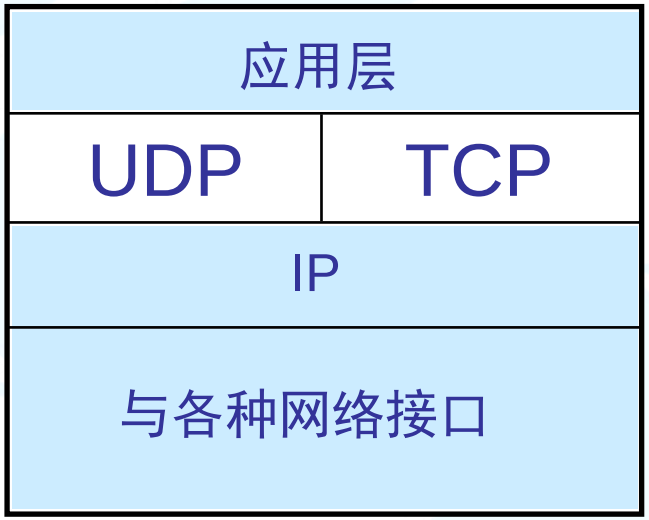
(2) 传输控制协议 TCP
(Transmission Control Protocol)

TCP 与 UDP

- 两个对等运输实体在通信时传送的数据单位叫作运输协议数据单元 **TPDU (Transport Protocol Data Unit)**。
- **TCP** 传送的数据单位协议是 **TCP** 报文段 **(segment)**
- **UDP** 传送的数据单位协议是 **UDP** 报文或用户数据报。

TCP/IP 体系中的运输层协议

运输层



TCP 与 UDP

- **UDP** 在传送数据之前不需要先建立连接。对方的运输层在收到 **UDP** 报文后，不需要给出任何确认。虽然 **UDP** 不提供可靠交付，但在某些情况下 **UDP** 是一种最有效的工作方式。
- **TCP** 则提供面向连接的服务。**TCP** 不提供广播或多播服务。由于 **TCP** 要提供可靠的、面向连接的运输服务，因此不可避免地增加了许多的开销。这不仅使协议数据单元的首部增大很多，还要占用许多的处理机资源。

还要强调两点

- 运输层的 **UDP** 用户数据报与网际层的**IP**数据报有很大区别。**IP**数据报要经过互连网中许多路由器的存储转发，但 **UDP** 用户数据报是在运输层的端到端抽象的逻辑信道中传送的。
- **TCP** 报文段是在运输层抽象的端到端逻辑信道中传送，这种信道是可靠的全双工信道。但这样的信道却不知道究竟经过了哪些路由器，而这些路由器也根本不知道上面的运输层是否建立了 **TCP** 连接。

运输层的端口

- 运行在计算机中的进程是用进程标识符来标志的。
- 运行在应用层的各种应用进程却不应当让计算机操作系统指派它的进程标识符。这是因为在因特网上使用的计算机的操作系统种类很多，而不同的操作系统又使用不同格式的进程标识符。
- 为了使运行不同操作系统的计算机的应用进程能够互相通信，就必须用统一的方法对 **TCP/IP** 体系的应用进程进行标志。

需要解决的问题

- 由于进程的创建和撤销都是动态的，发送方几乎无法识别其他机器上的进程。
- 有时我们会改换接收报文的进程，但并不需要通知所有发送方。
- 我们往往需要利用目的主机提供的功能来识别终点，而不需要知道实现这个功能的进程。

端口号(protocol port number)简称为端口(port)

- 解决这个问题的方法就是在运输层使用协议端口号**(protocol port number)**，或通常简称为端口**(port)**。
- 虽然通信的终点是应用进程，但我们可以把端口想象是通信的终点，因为我们只要把要传送的报文交到目的主机的某一个合适的目的端口，剩下的工作（即最后交付目的进程）就由 **TCP** 来完成。

软件端口与硬件端口

- 在协议栈层间的抽象的协议端口是软件端口。
- 路由器或交换机上的端口是硬件端口。
- 硬件端口是不同硬件设备进行交互的接口，而软件端口是应用层的各种协议进程与运输实体进行层间交互的一种地址。

TCP 的端口

- 端口用一个 **16** 位端口号进行标志。
- 端口号只具有本地意义，即端口号只是为了标志本计算机应用层中的各进程。在因特网中不同计算机的相同端口号是没有联系的。

三类端口

- 熟知端口，数值一般为 **0~1023**。
- 登记端口号，数值为**1024~49151**，为没有熟知端口号的应用程序使用的。使用这个范围的端口号必须在 **IANA** 登记，以防止重复。
- 客户端口号或短暂端口号，数值为**49152~65535**，留给客户进程选择暂时使用。当服务器进程收到客户进程的报文时，就知道了客户进程所使用的动态端口号。通信结束后，这个端口号可供其他客户进程以后使用。

TCP 的连接

- **TCP** 把连接作为最基本的抽象。
- 每一条 **TCP** 连接有两个端点。
- **TCP** 连接的端点不是主机，不是主机的 **IP** 地址，不是应用进程，也不是运输层的协议端口。**TCP** 连接的端点叫做套接字**(socket)**或插口。
- 端口号拼接到**(contatenated with) IP** 地址即构成了套接字。

套接字 (socket)

套接字 **socket = (IP地址: 端口号)**

每一条 TCP 连接唯一地被通信两端的两个端点（即两个套接字）所确定。即：

TCP 连接 ::= {socket1, socket2}
= {(IP1: port1), (IP2: port2)}

同一个名词 socket有多种不同的意思

- 应用编程接口 **API** 称为 **socket API**，简称为 **socket**。
- **socket API** 中使用的一个函数名也叫作 **socket**。
- 调用 **socket** 函数的端点称为 **socket**。
- 调用 **socket** 函数时其返回值称为 **socket** 描述符，可简称为 **socket**。
- 在操作系统内核中连网协议的 **Berkeley** 实现，称为 **socket** 实现。

4.2 Multiplexing and demultiplexing

Multiplexing/demultiplexing

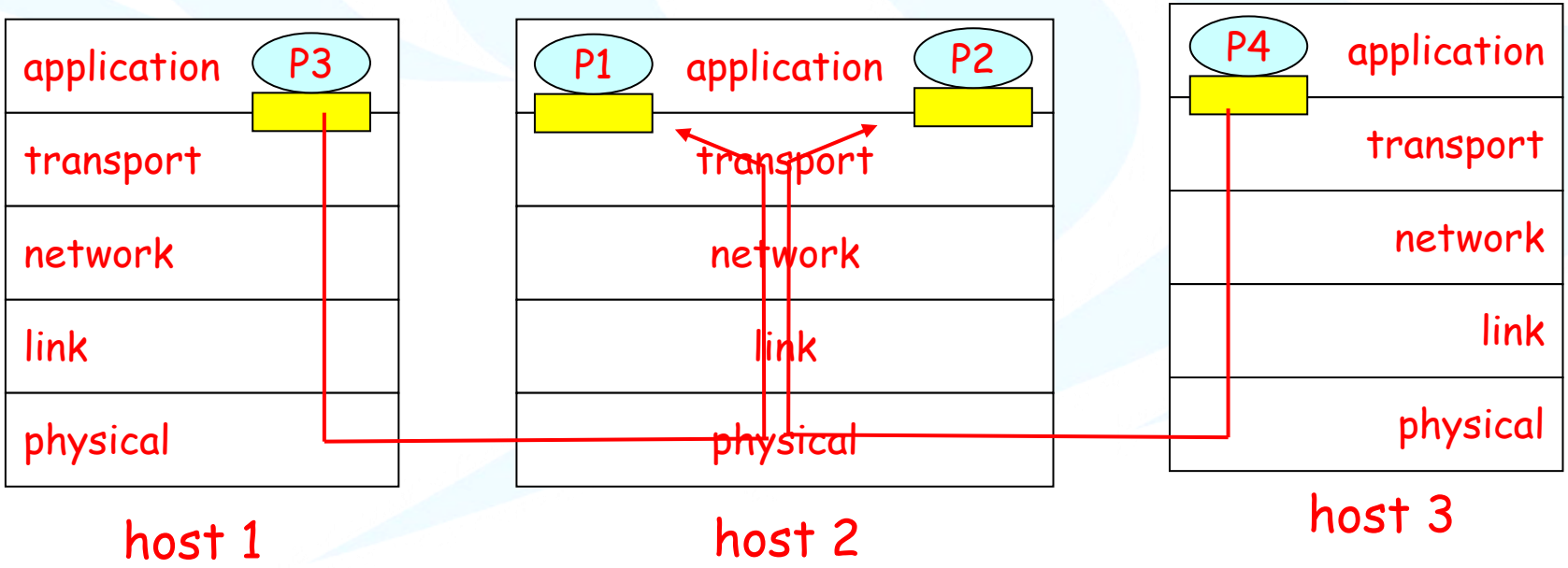
Demultiplexing at rcv host:

delivering received segments to correct socket

Multiplexing at send host:

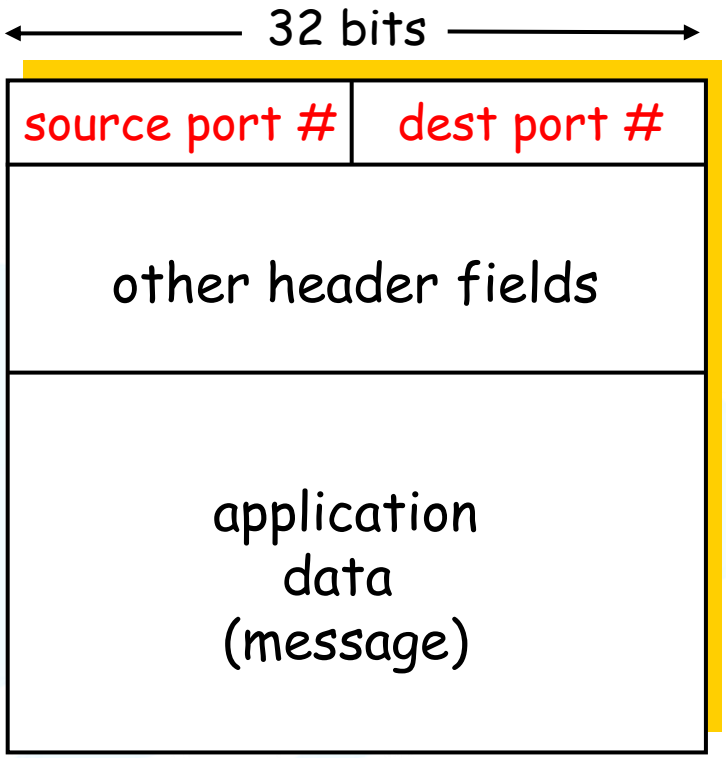
gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

= socket = process



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 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

Connectionless demultiplexing

- **Create sockets with port numbers:**

```
DatagramSocket mySocket1 =
   new DatagramSocket(99111);
DatagramSocket mySocket2 =
   new DatagramSocket(99222);
```

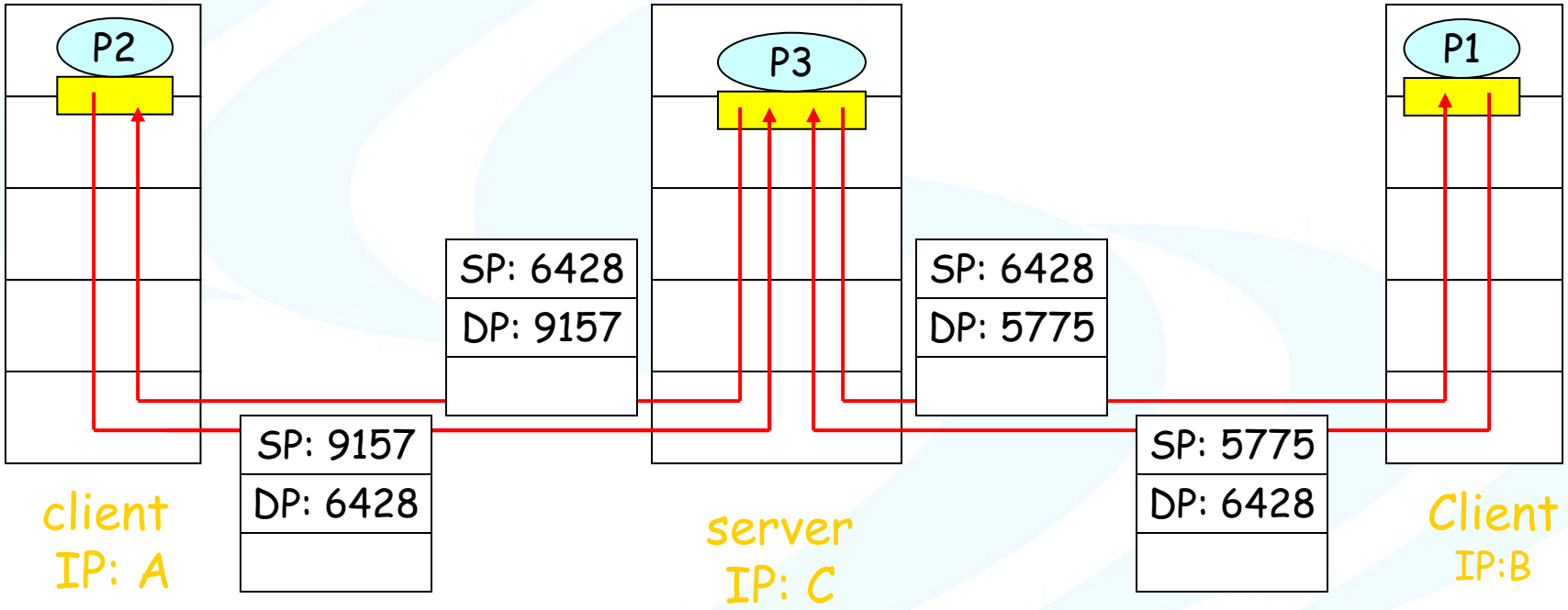
- **UDP socket identified by two-tuple:**

(dest IP address, dest port number)

- **When host receives UDP segment:**
  - **checks destination port number in segment**
  - **directs UDP segment to socket with that port number**
- **IP datagrams with different source IP addresses and/or source port numbers directed to same socket**

Connectionless demux (cont)

```
DatagramSocket serverSocket = new
  DatagramSocket(6428);
```

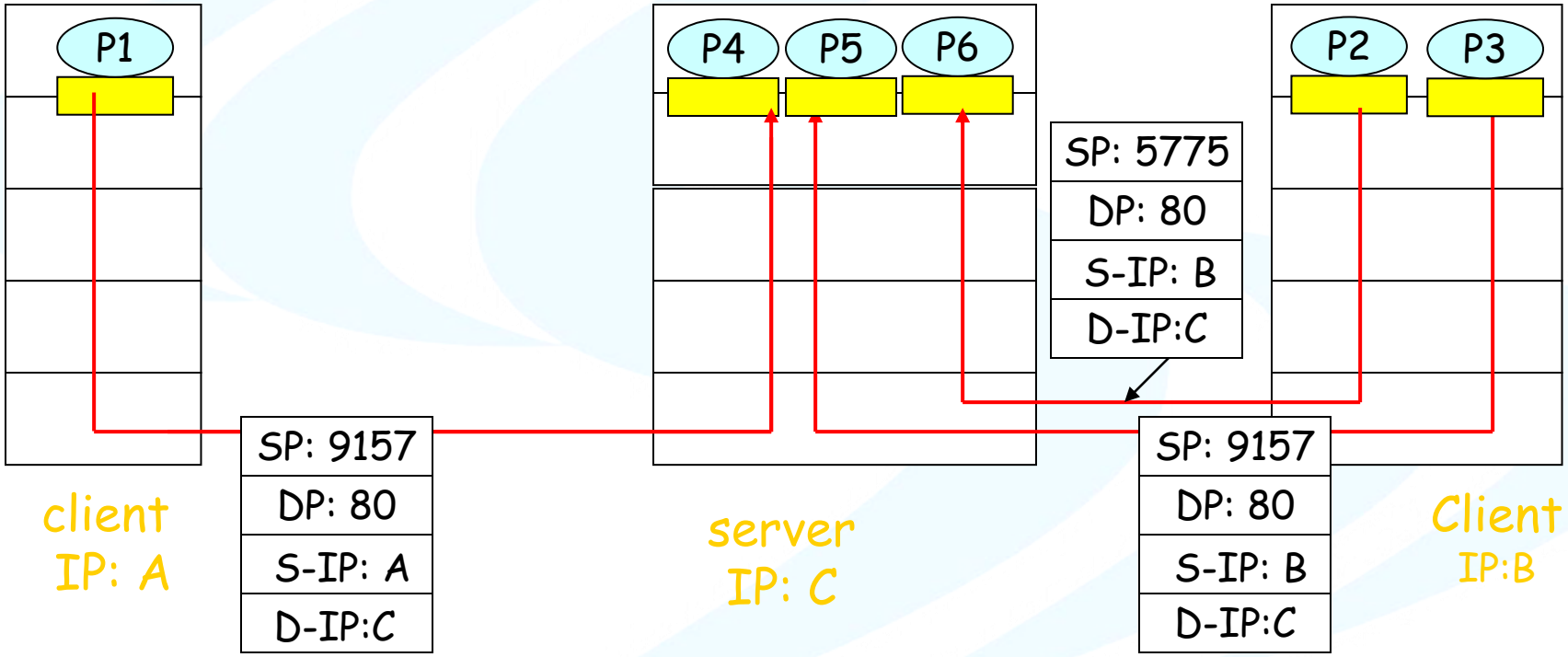


SP provides "return address"

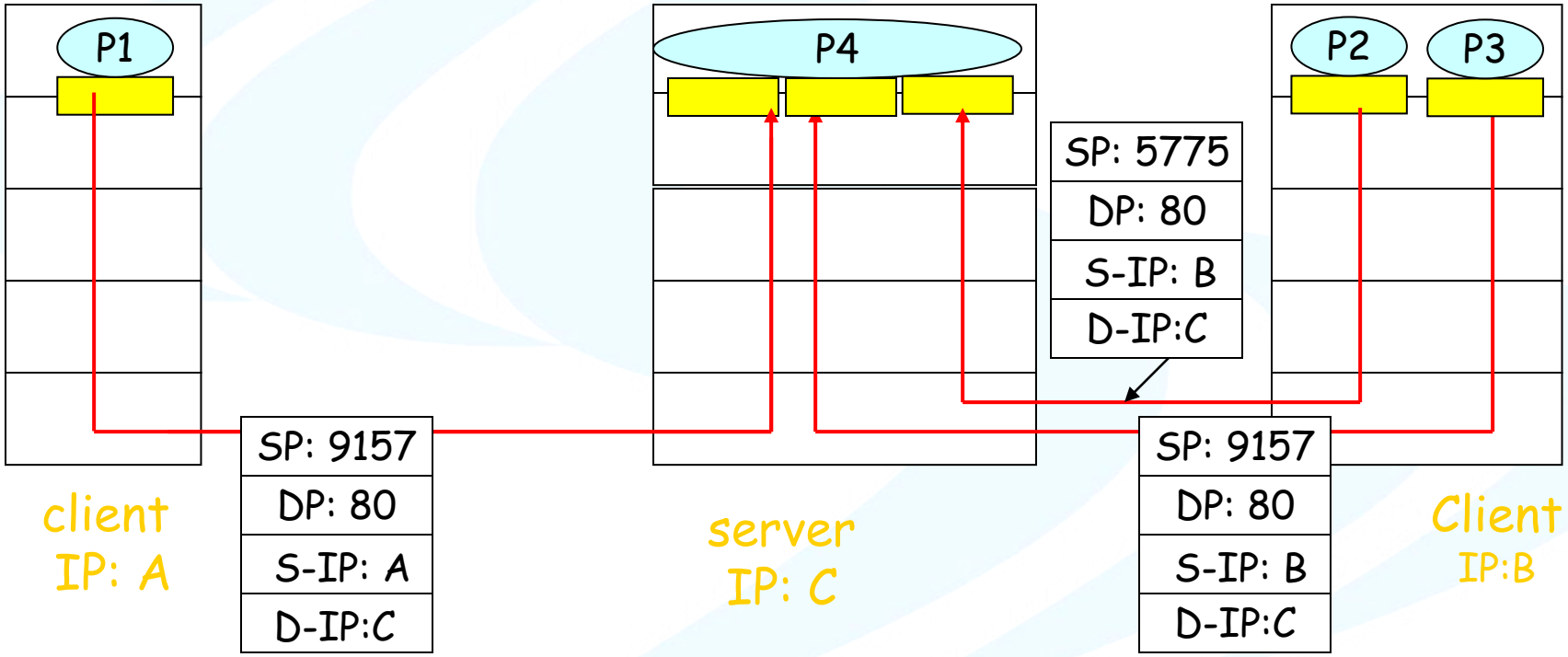
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host 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

Connection-oriented demux (cont)



Connection-oriented demux:Threaded Web Server



4.3 Connectionless transport: UDP

UDP: User Datagram Protocol [RFC 768]

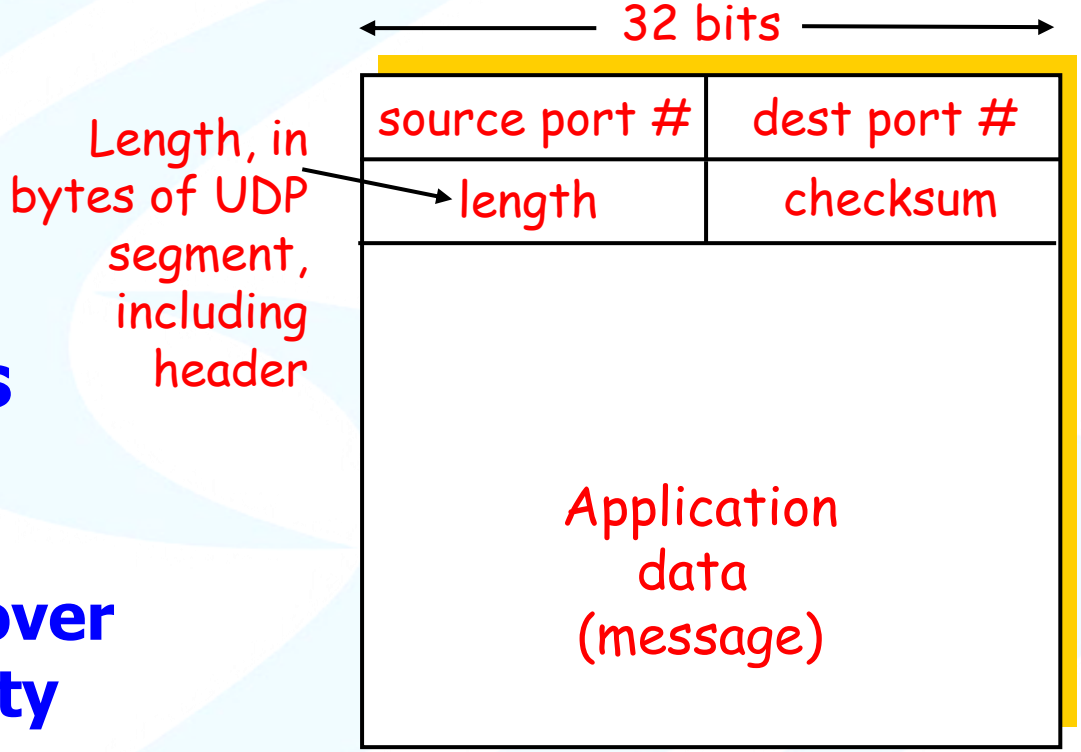
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

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

UDP: more

- **often used for streaming multimedia apps**
  - **loss tolerant**
  - **rate sensitive**
- **other UDP uses**
  - **DNS**
  - **SNMP**
- **reliable transfer over UDP: add reliability at application layer**
  - **application-specific error recovery!**

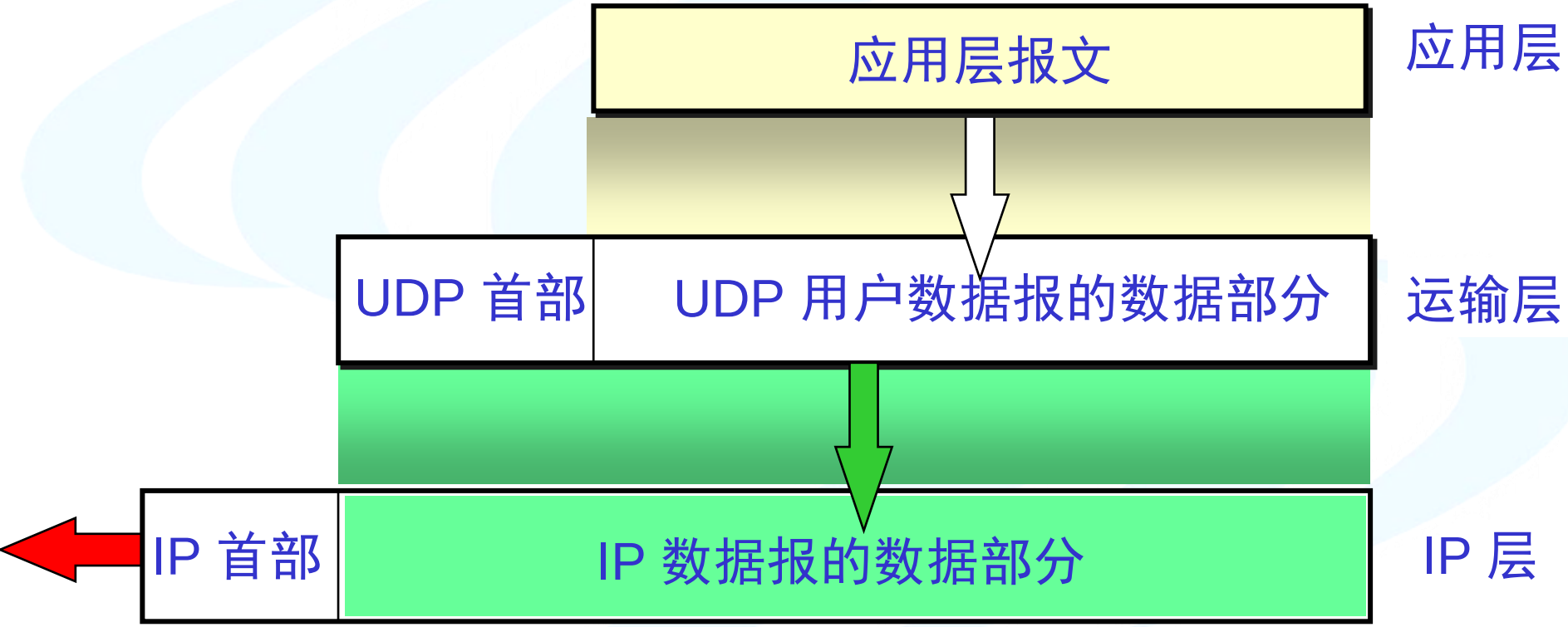


UDP segment format

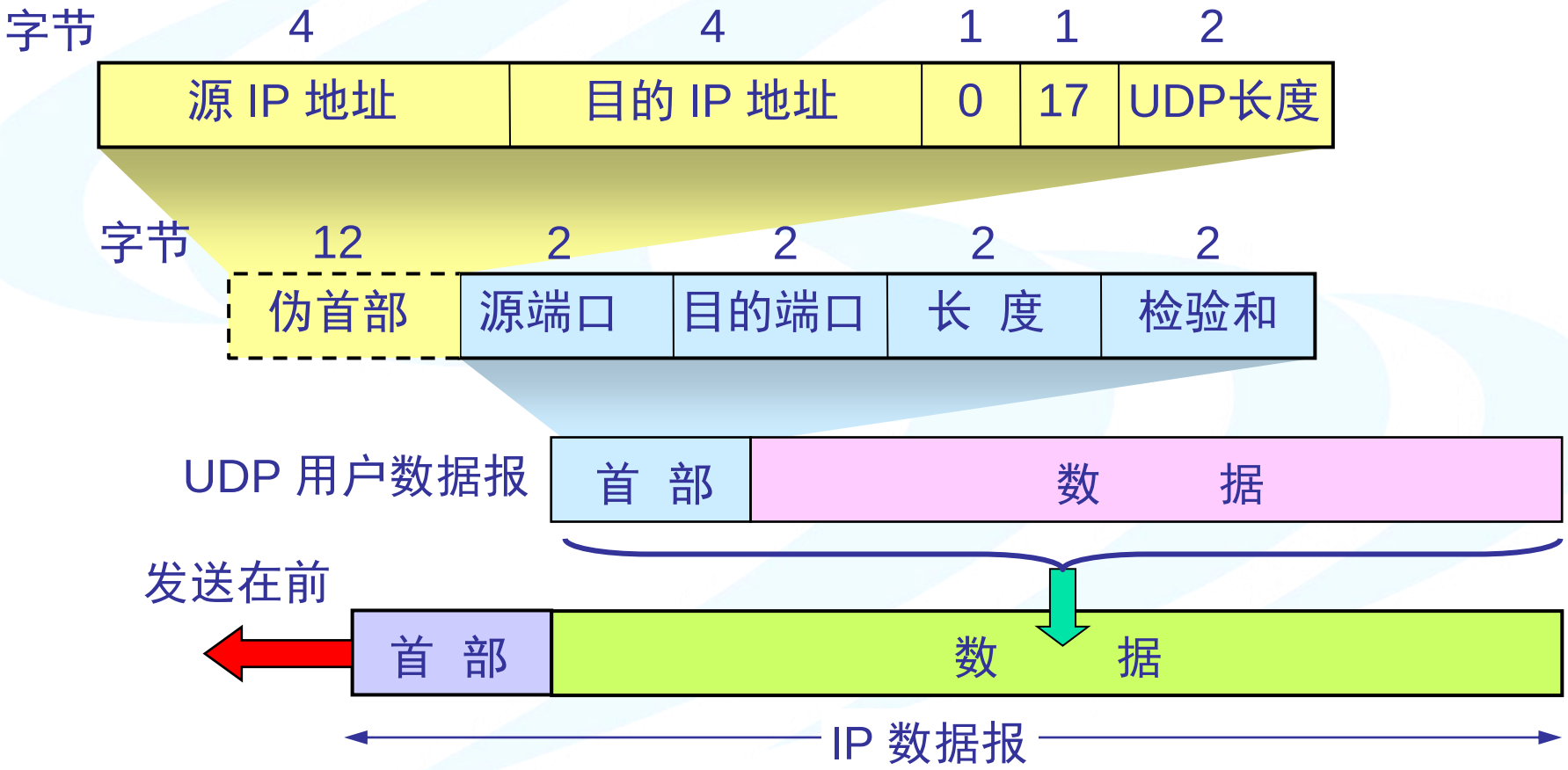
面向报文的 UDP

- 发送方 **UDP** 对应用程序交下来的报文，在添加首部后就向下交付 **IP** 层。**UDP** 对应用层交下来的报文，既不合并，也不拆分，而是保留这些报文的边界。
- 应用层交给 **UDP** 多长的报文，**UDP** 就照样发送，即一次发送一个报文。
- 接收方 **UDP** 对 **IP** 层交上来的 **UDP** 用户数据报，在去除首部后就原封不动地交付上层的应用进程，一次交付一个完整的报文。
- 应用程序必须选择合适大小的报文。

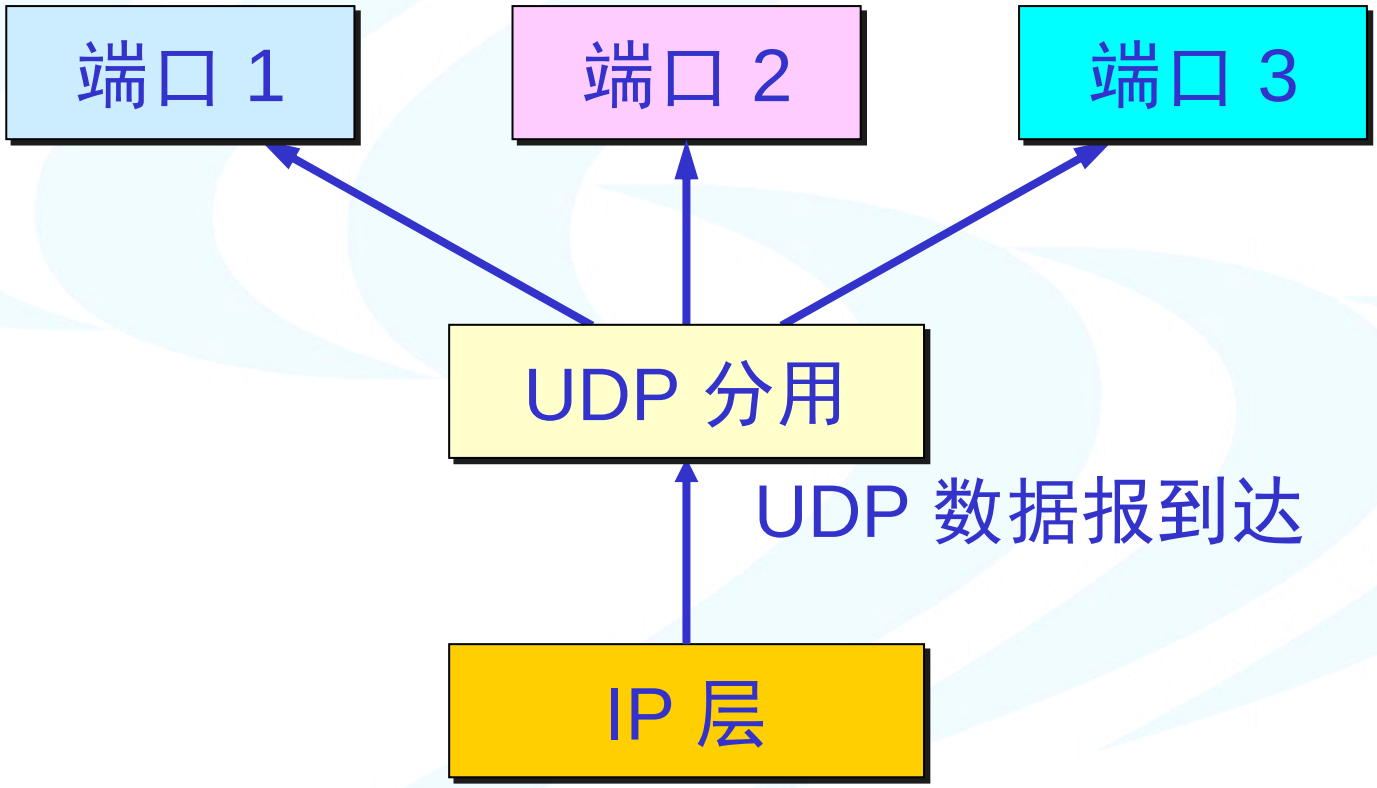
UDP 是面向报文的



UDP 的首部格式


字节
4
4
1
1
2
源 IP 地址
目的 IP 地址
0
17
UDP长度
字节
12
2
2
2
2
伪首部
源端口
目的端口
长 度
检验和
UDP 用户数据报
首 部
数 据
发送在前
首 部
数 据
IP 数据报

UDP 基于端口的分用



UDP checksum

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

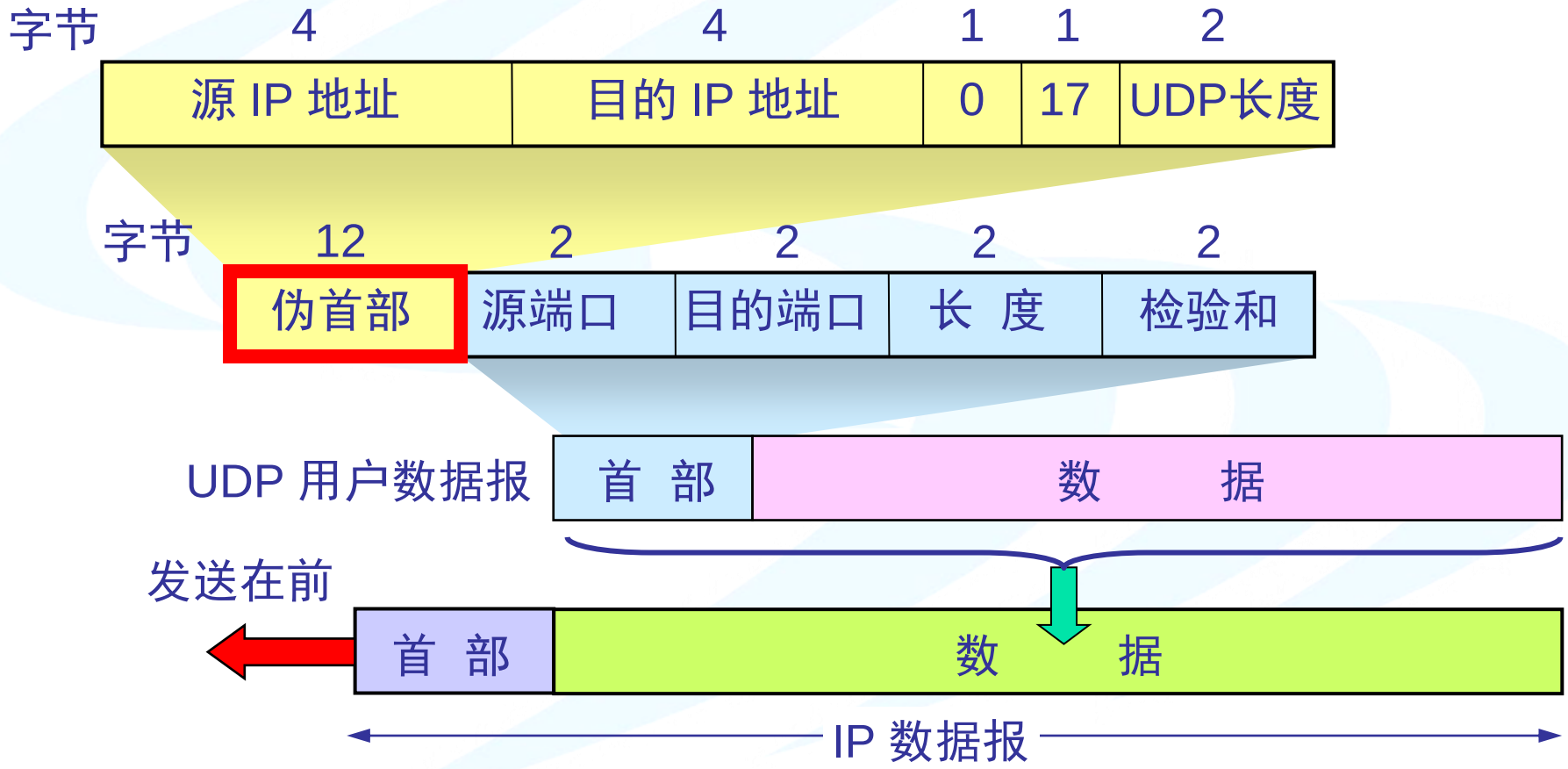
Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1'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 ….

在计算检验和时，临时把“伪首部”和 UDP 用户数据报连接在一起。伪首部仅仅是为了计算检验和。

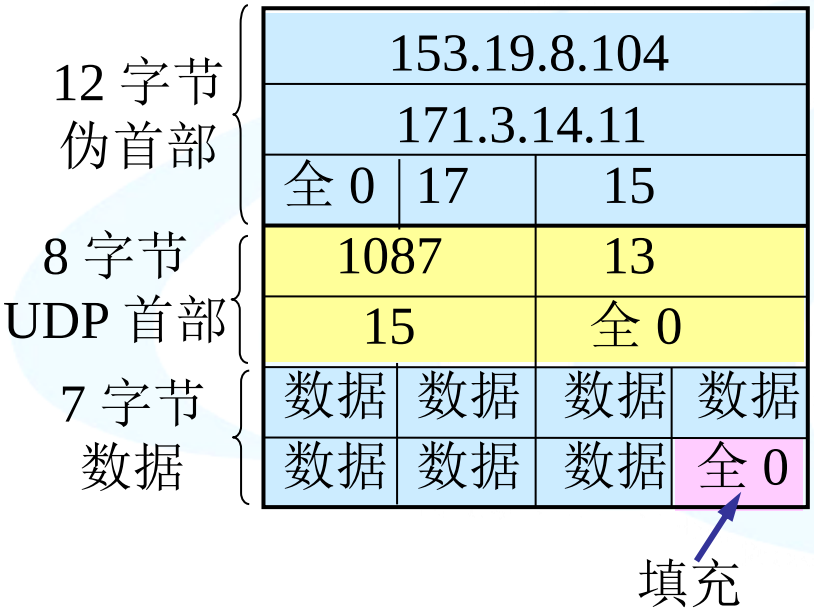


Internet Checksum Example

- **Note**
  - **When adding numbers, a carryout from the most significant bit needs to be added to the result**
- **Example: add two 16-bit integers**

	1	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	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	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	1	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

计算 UDP 检验和的例子



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（检验和）
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 10010110 11101101 → 求和得出的结果
将得出的结果求反码 01101001 00010010 → 检验和

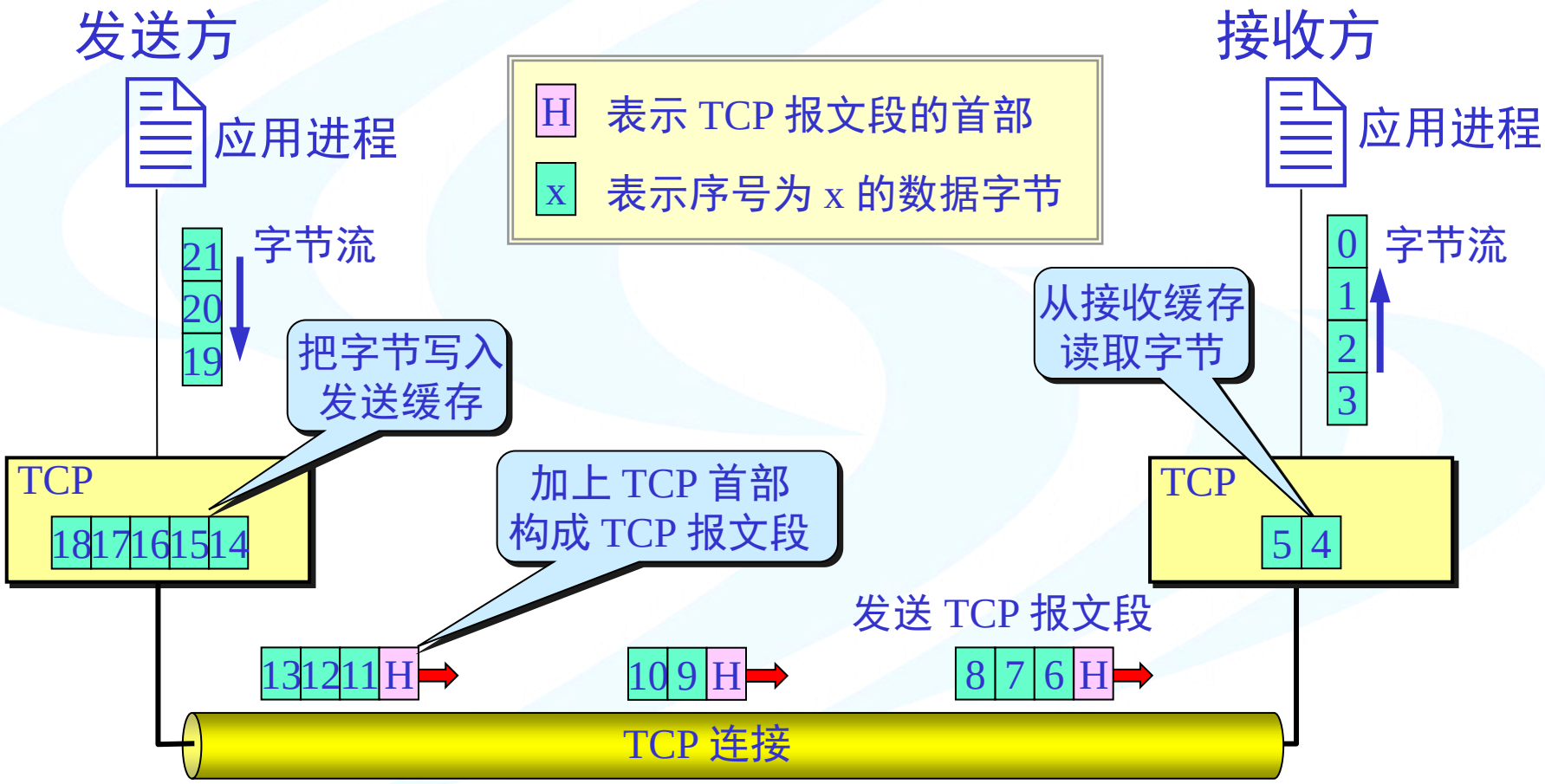
Connection-oriented transport: TCP

- Principles of reliable data transfer
- segment structure
- reliable data transfer of TCP
- flow control
- connection management
- TCP Congestion Control

TCP 最主要的特点

- TCP 是面向连接的运输层协议。
- 每一条 TCP 连接只能有两个端点 (endpoint)，每一条 TCP 连接只能是点对点的（一对一）。
- TCP 提供可靠交付的服务。
- TCP 提供全双工通信。
- 面向字节流。

TCP 面向流的概念


发送方
应用进程
字节流
21
20
19
把字节写入
发送缓存
TCP
18 17 16 15 14
H 表示 TCP 报文段的首部
x 表示序号为 x 的数据字节
加上 TCP 首部
构成 TCP 报文段
13 12 11 H
10 9 H
发送 TCP 报文段
8 7 6 H
TCP 连接
接收方
应用进程
字节流
0
1
2
3
从接收缓存
读取字节
TCP
5 4

应当注意

- **TCP** 连接是一条虚连接而不是一条真正的物理连接。
- **TCP** 对应用进程一次把多长的报文发送到**TCP** 的缓存中是不关心的。
- **TCP** 根据对方给出的窗口值和当前网络拥塞的程度来决定一个报文段应包含多少个字节（**UDP** 发送的报文长度是应用进程给出的）。
- **TCP** 可把太长的数据块划分短一些再传送。**TCP** 也可等待积累有足够多的字节后再构成报文段发送出去。

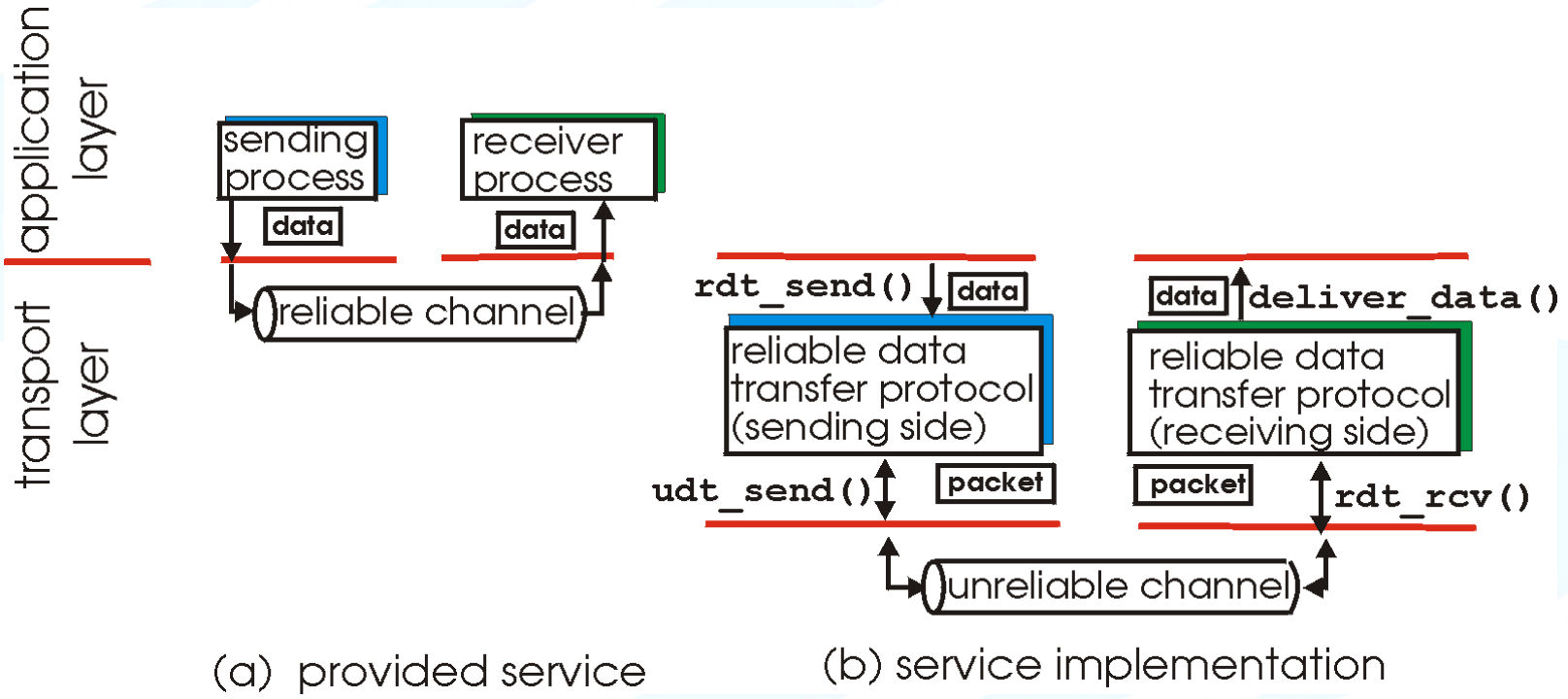
TCP 的连接

- **TCP** 把连接作为最基本的抽象。
- 每一条 **TCP** 连接有两个端点。
- **TCP** 连接的端点不是主机，不是主机的 **IP** 地址，不是应用进程，也不是运输层的协议端口。**TCP** 连接的端点叫做套接字**(socket)**或插口。
- 端口号拼接到**(contatenated with) IP** 地址即构成了套接字。

4.4 Principles of reliable data transfer

Principles of Reliable data transfer

- **important in app., transport, link layers**

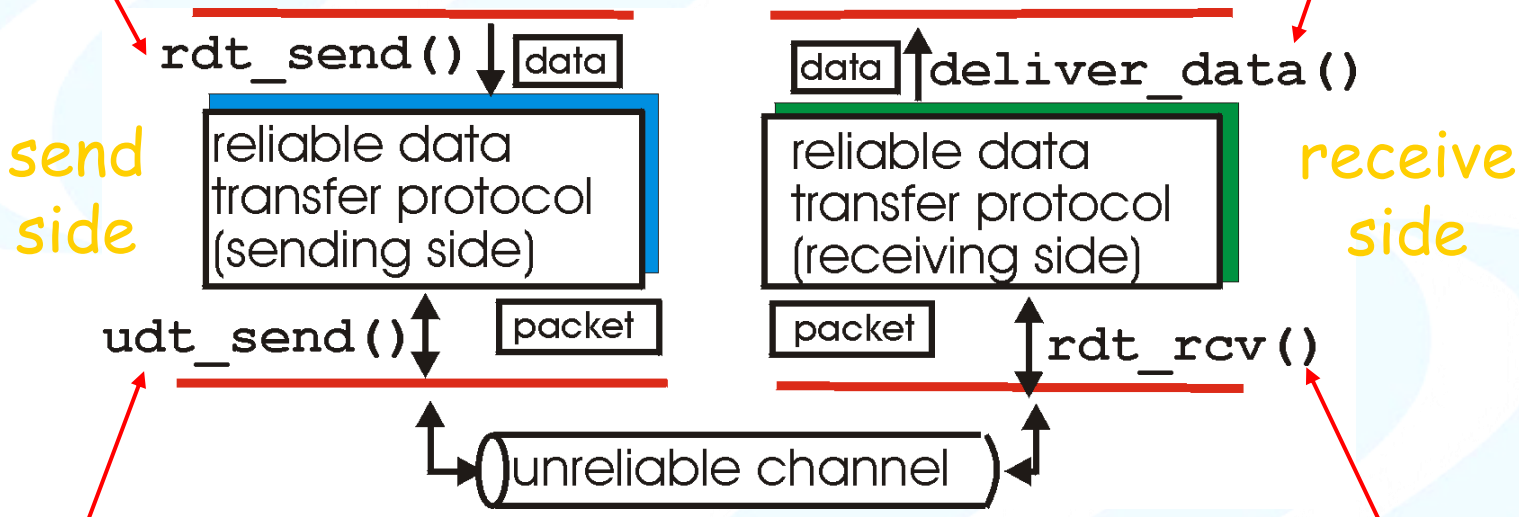


- **characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)**

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

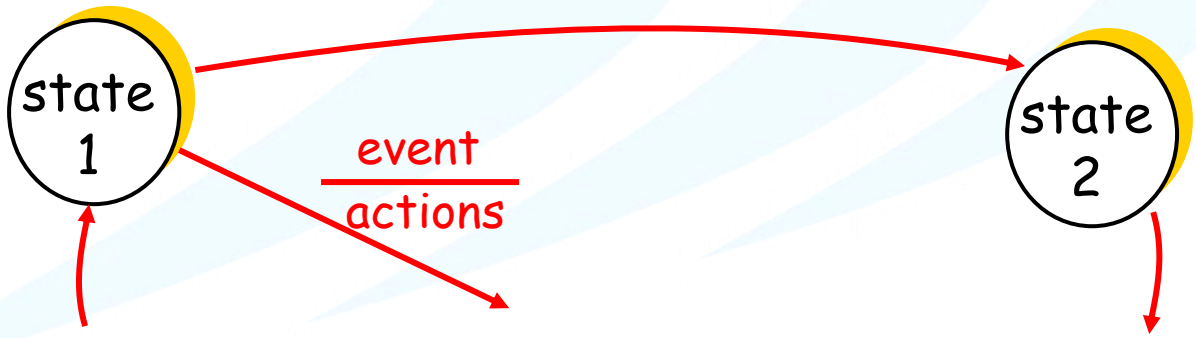
可靠数据传输的规范化接口描述，及对应的功能。

Reliable data transfer: getting started

We'll:

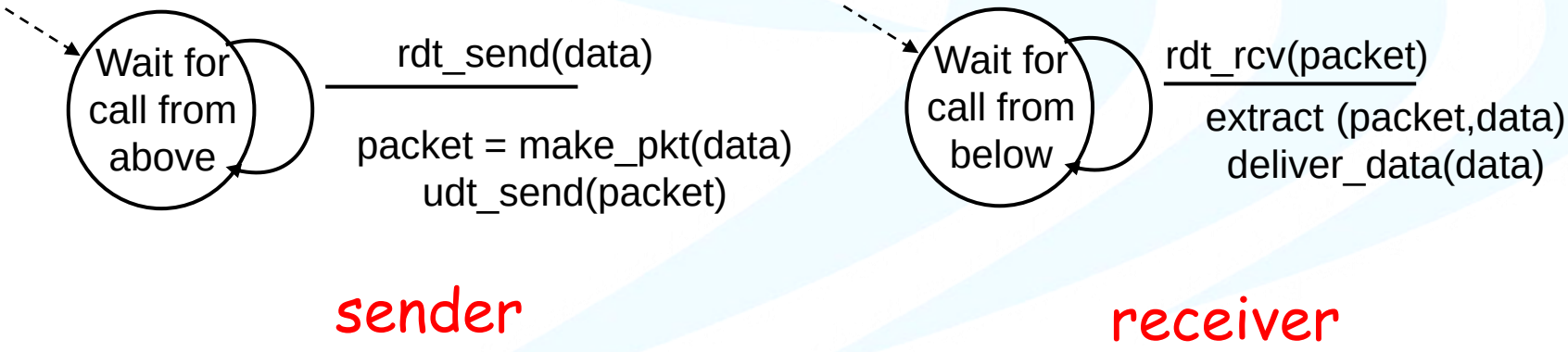
- **incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)**
- **consider only unidirectional data transfer**
  - **but control info will flow on both directions!**
- **use finite state machines (FSM) to specify sender, receiver**

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



Rdt1.0: reliable transfer over a reliable channel

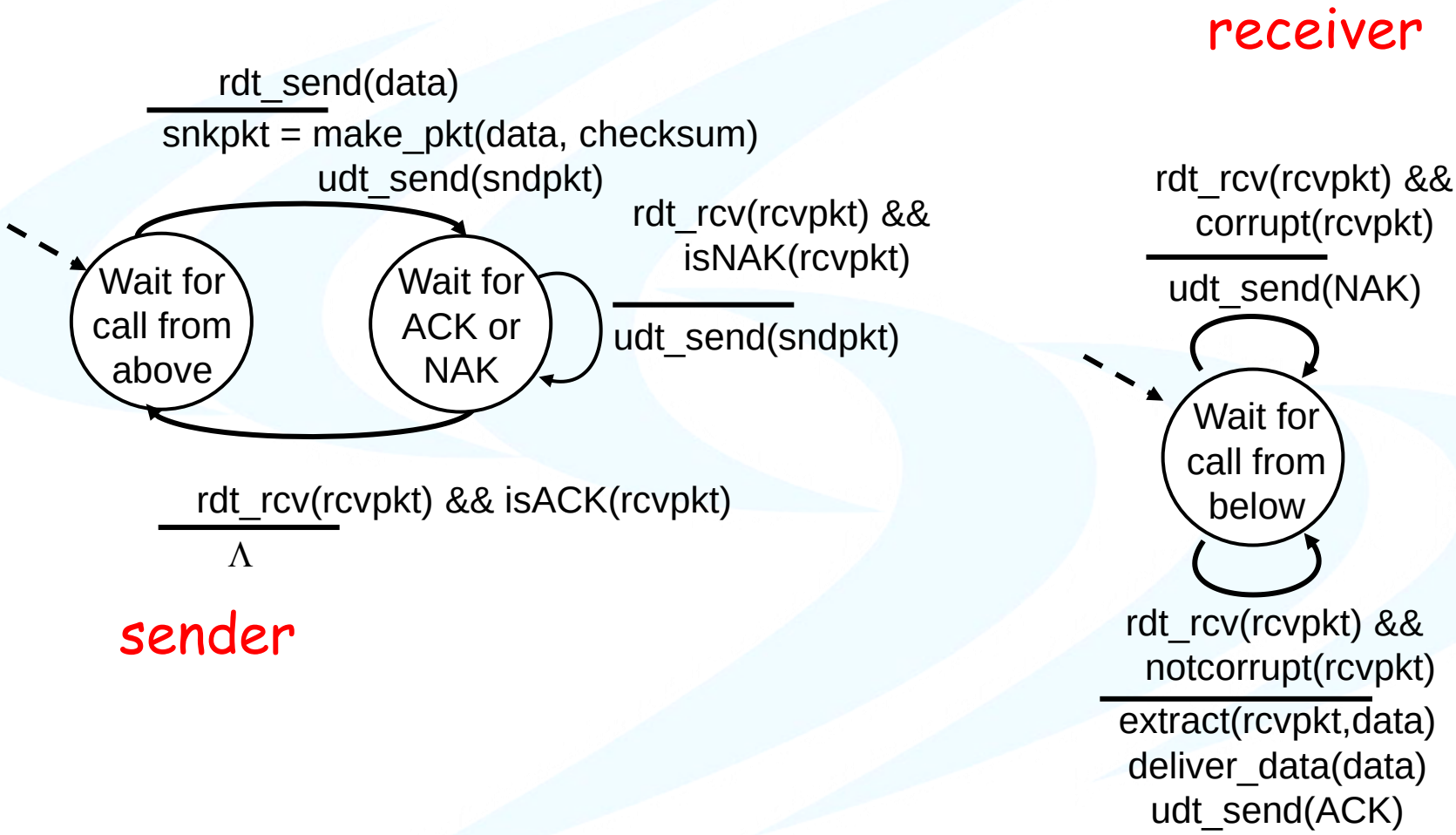
- **underlying channel perfectly reliable**
  - **no bit errors**
  - **no loss of packets**
- **separate FSMs for sender, receiver:**
  - **sender sends data into underlying channel**
  - **receiver read data from underlying channel**



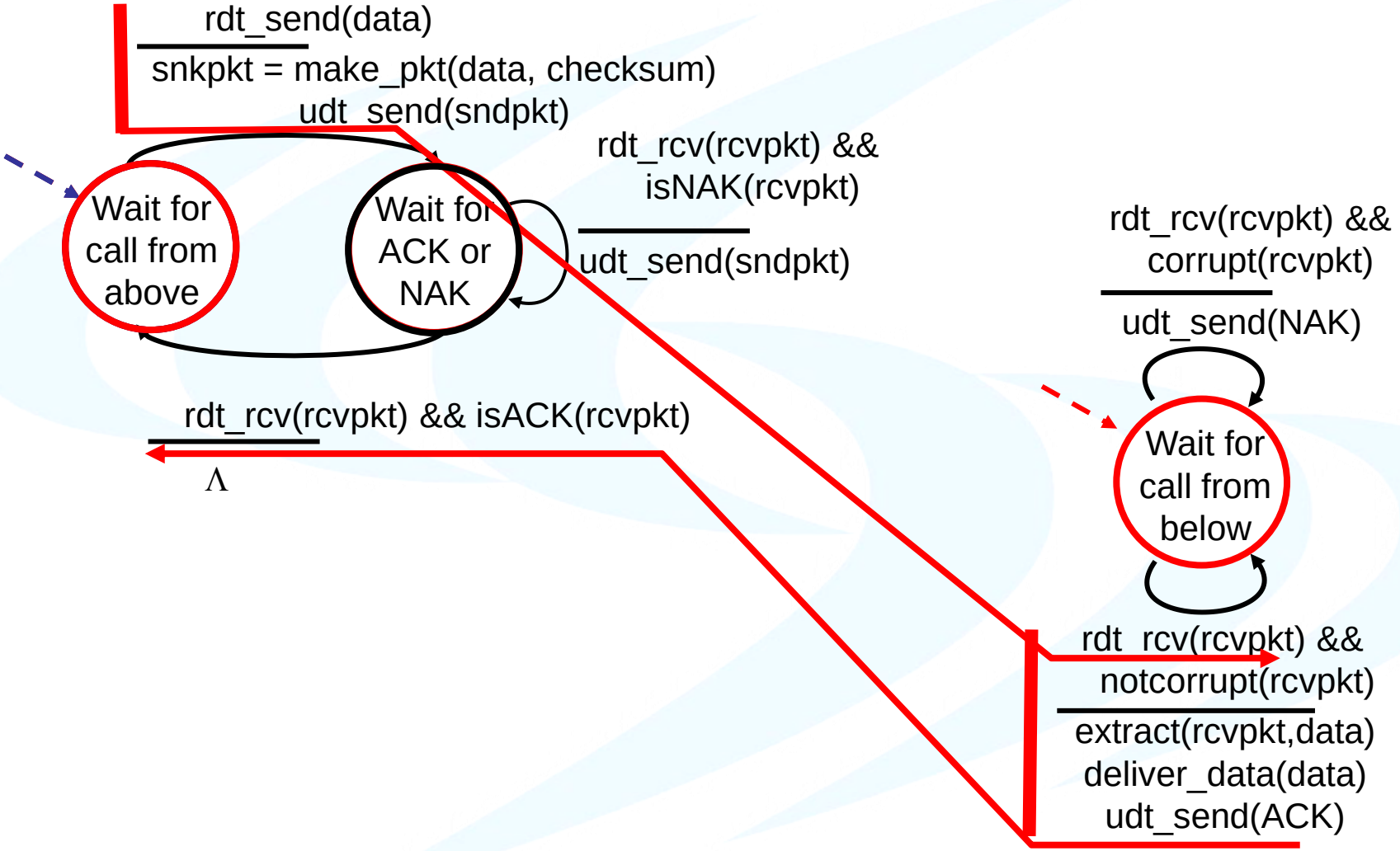
Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* 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 msgs (ACK,NAK) rcvr->sender

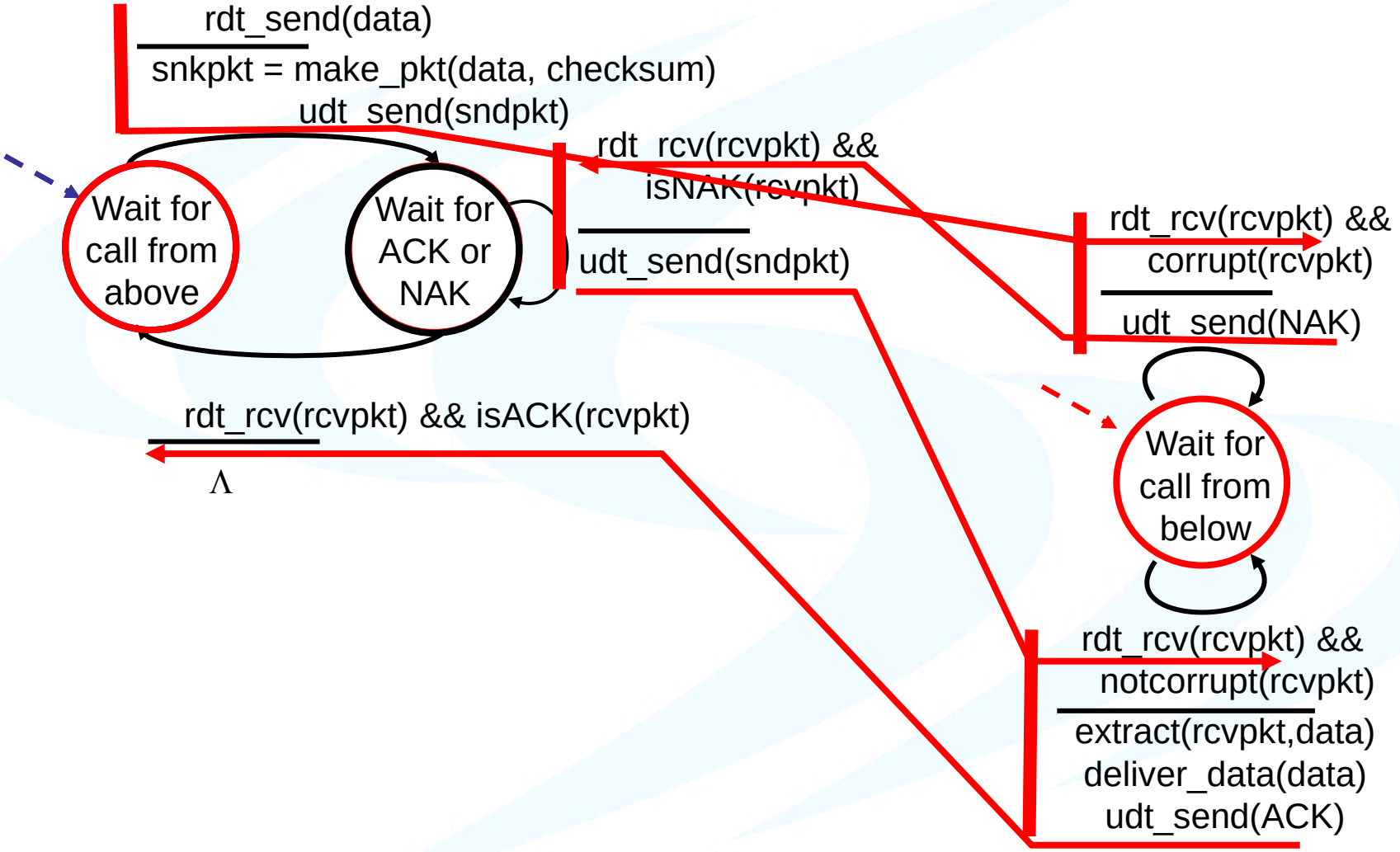
rdt2.0: FSM specification



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

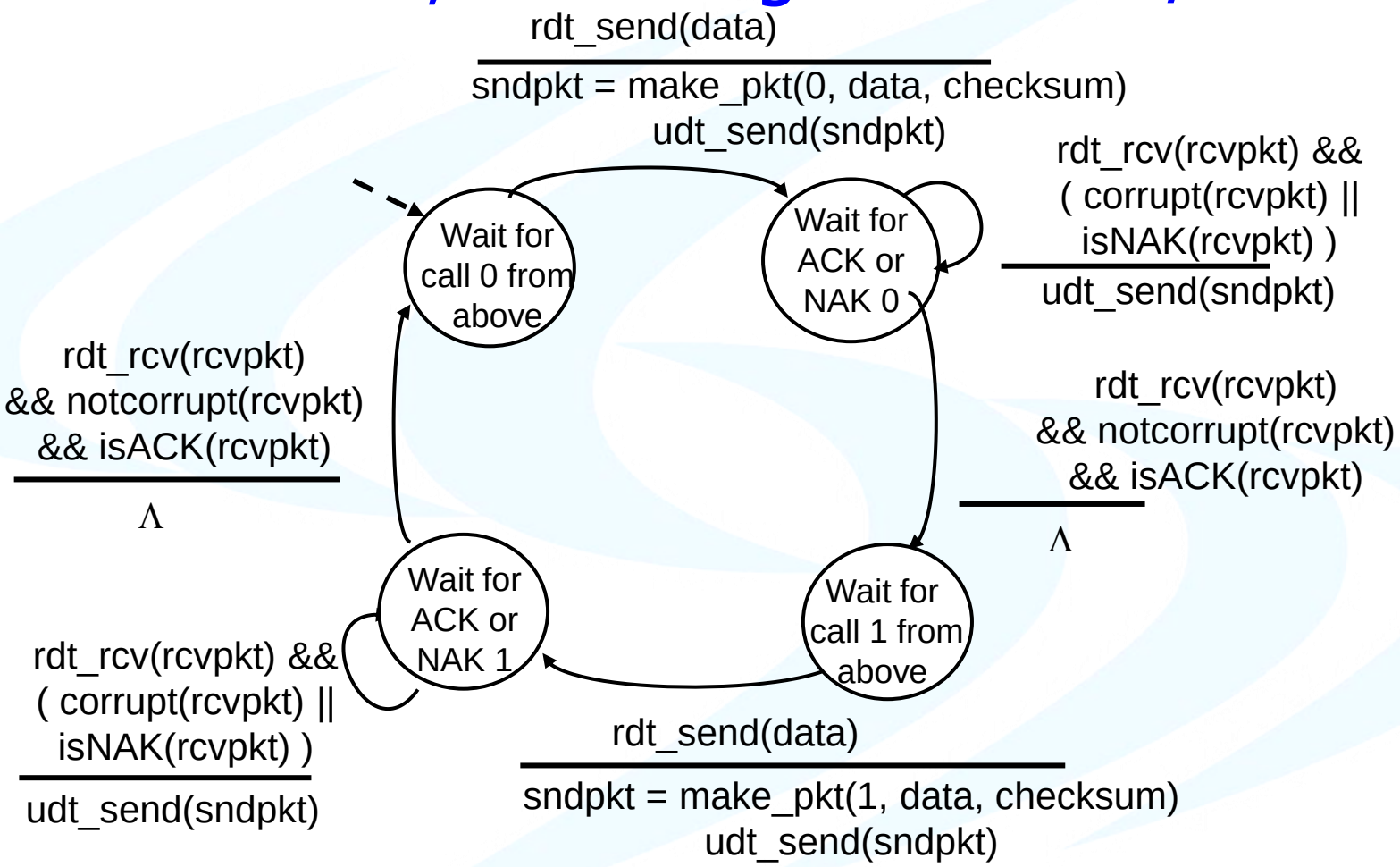
Handling duplicates:

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

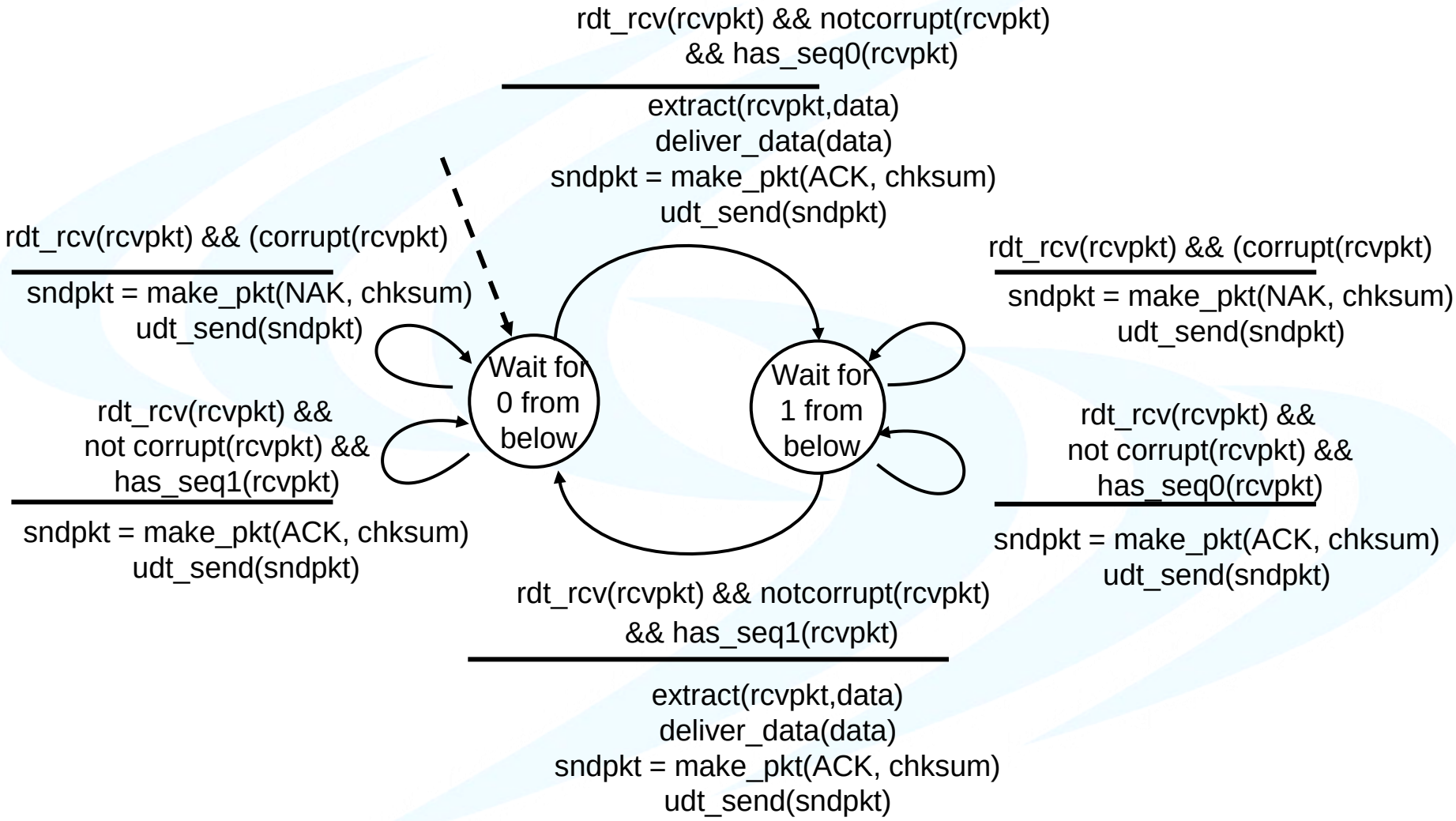
stop and wait

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. #

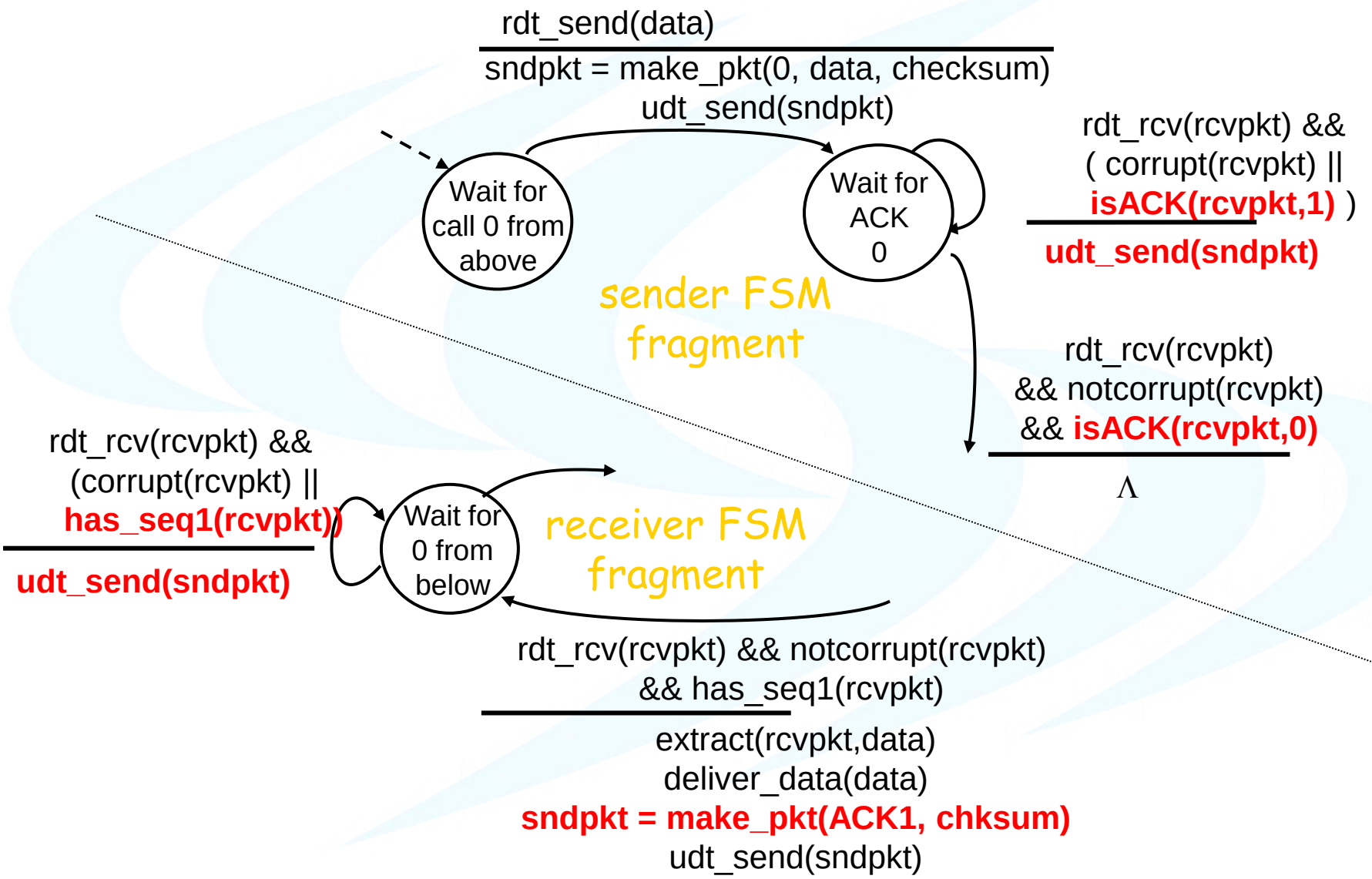
Receiver:

- 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 ACKs 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, receiver fragments



rdt3.0: channels with errors *and* loss

New assumption:

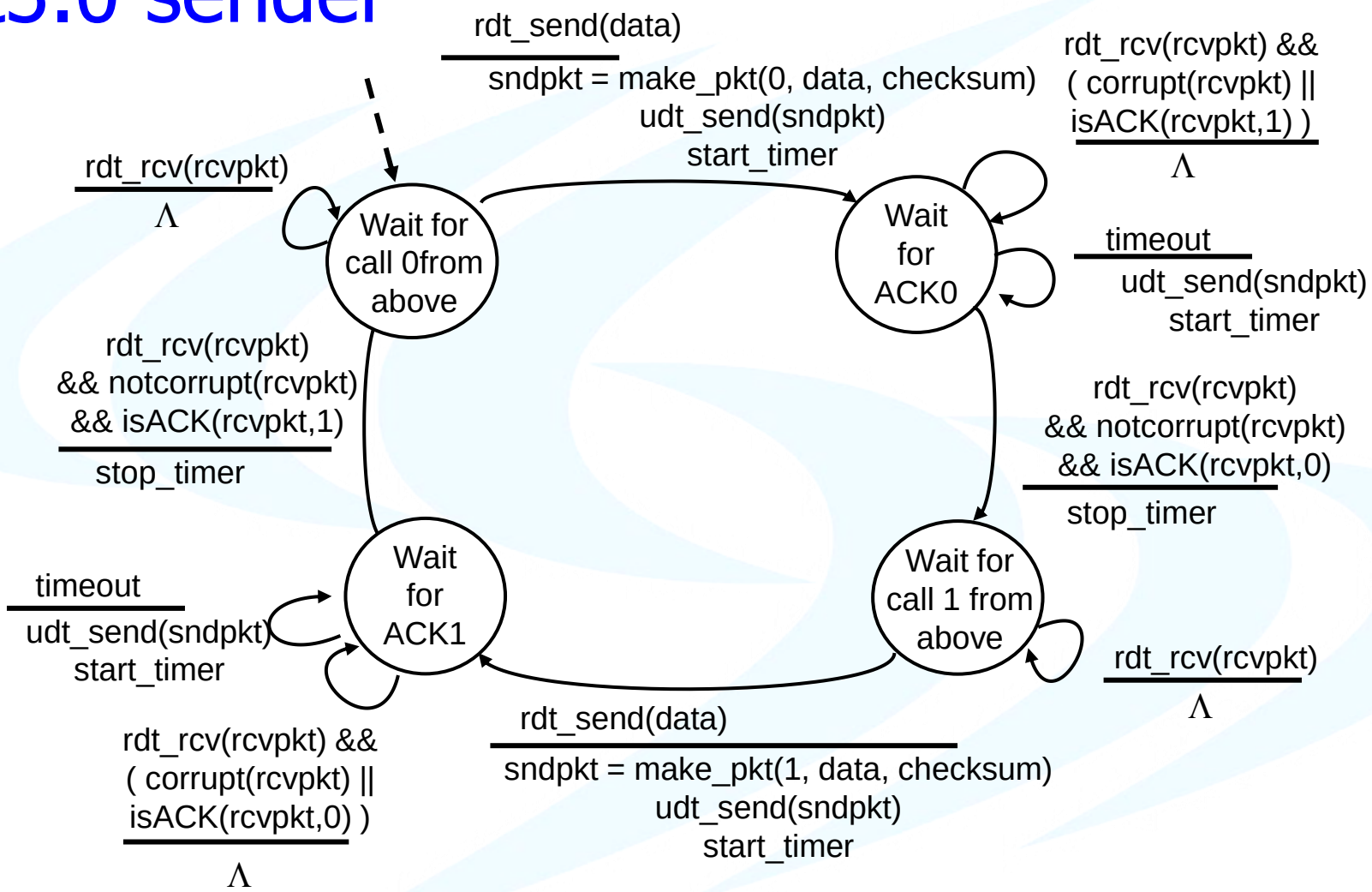
underlying channel can also lose packets (data or ACKs)

- **checksum, seq. #, ACKs, retransmissions will be of help, but not enough**

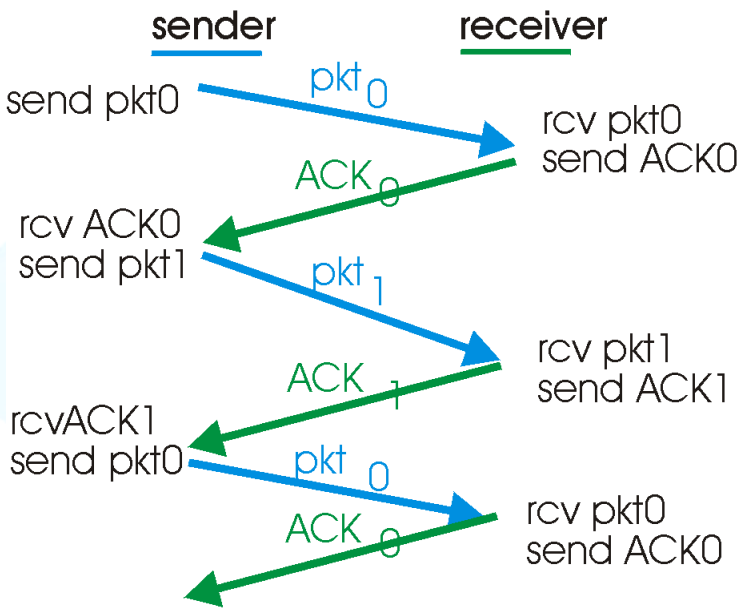
Approach: sender waits "reasonable" amount of time for ACK

- **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**

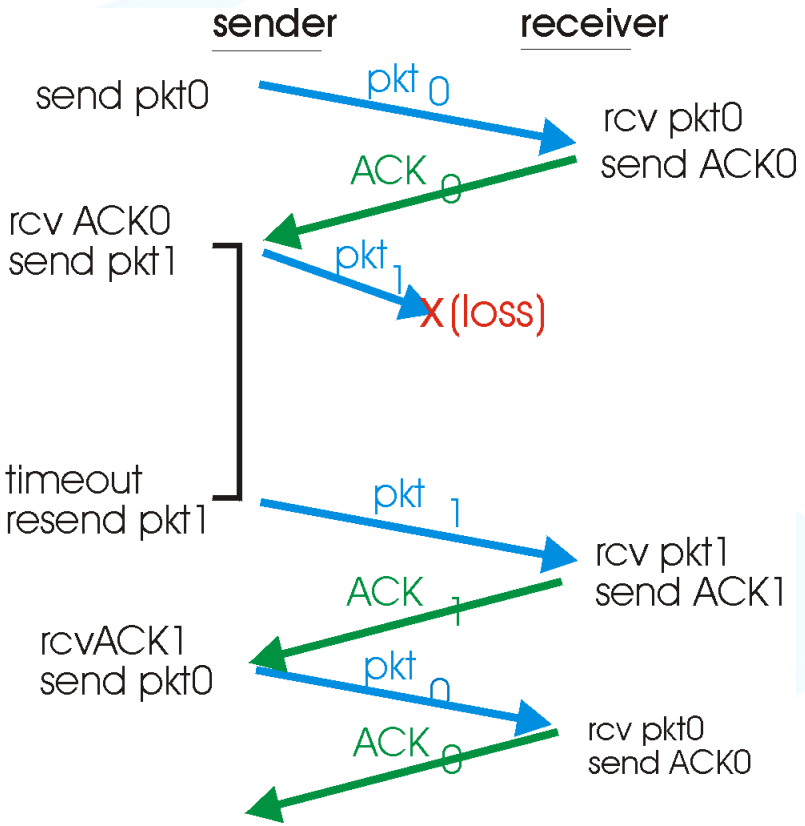
rdt3.0 sender



rdt3.0 in action

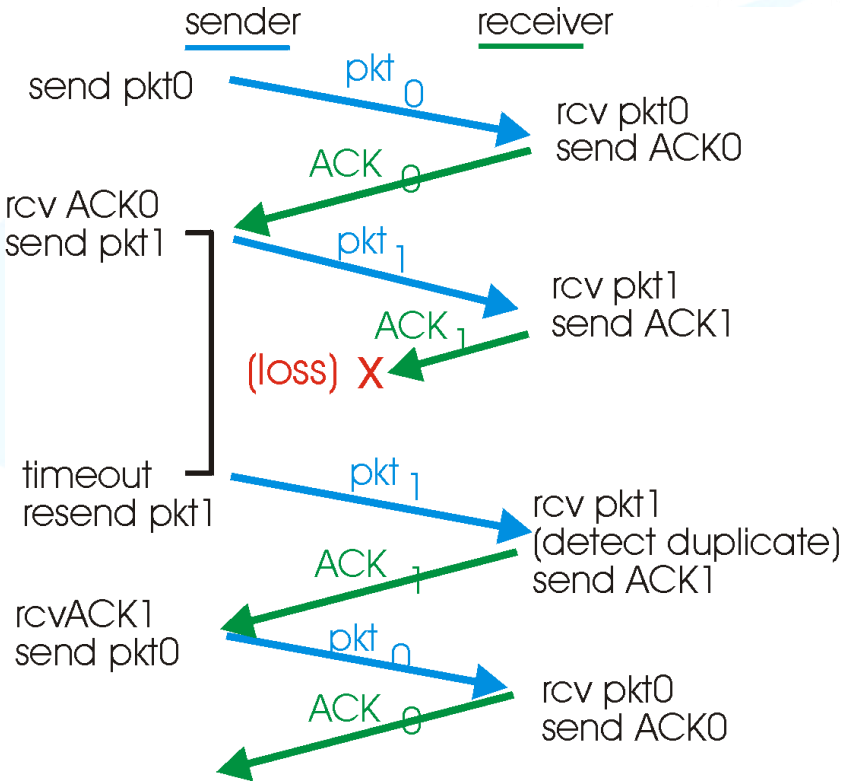


(a) operation with no loss

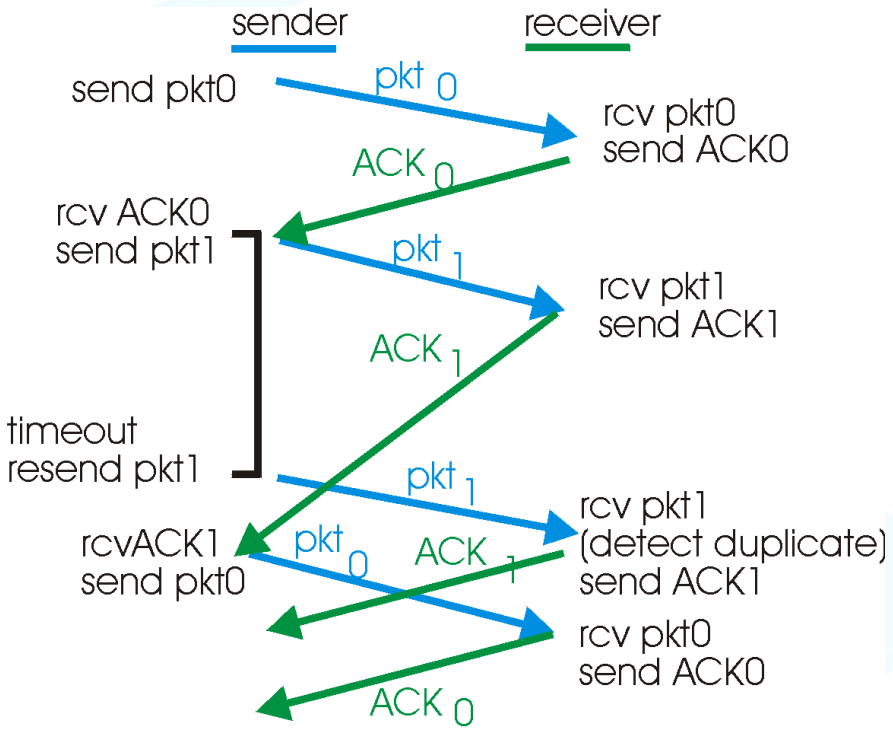


(b) lost packet

rdt3.0 in action

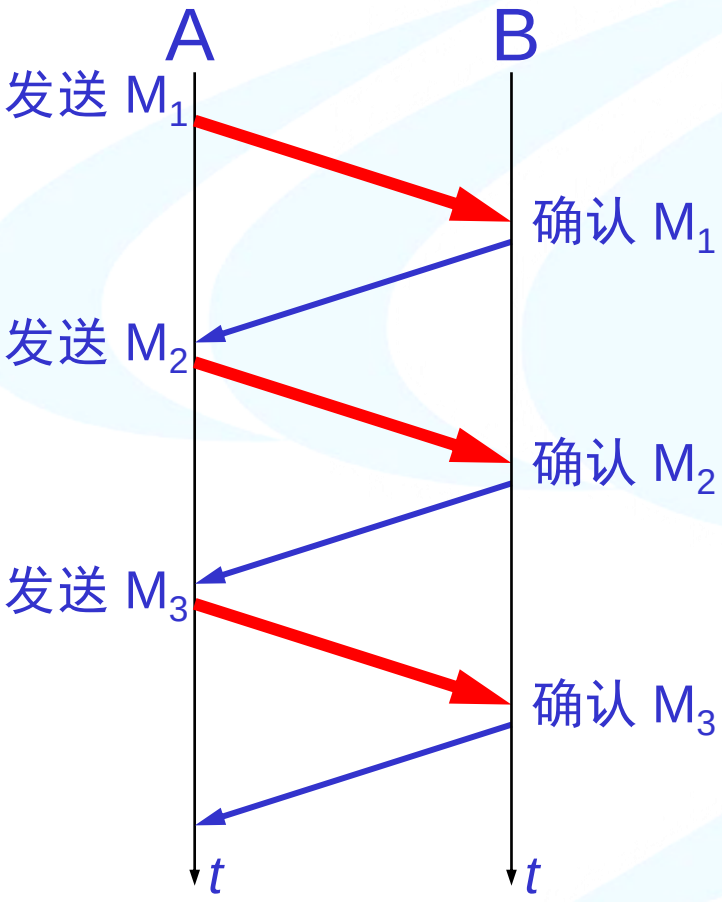


(c) lost ACK

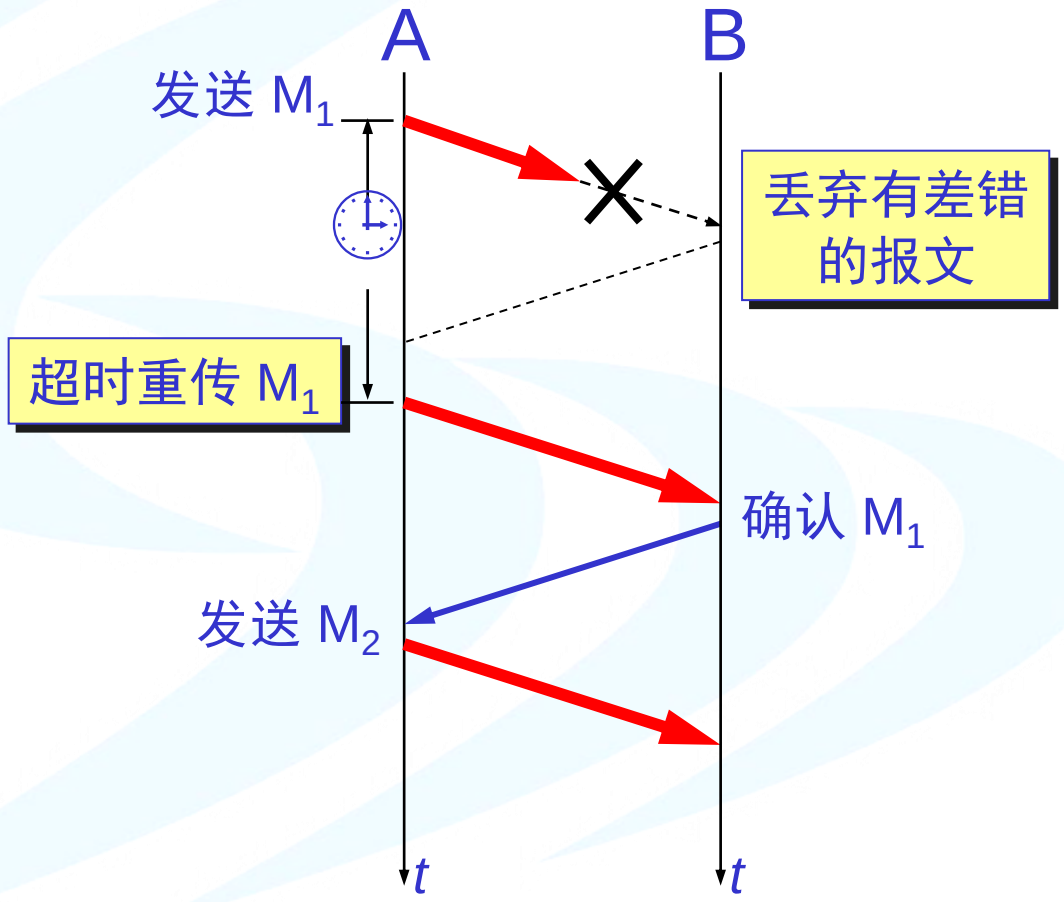


(d) premature timeout

停止等待协议



(a) 无差错情况

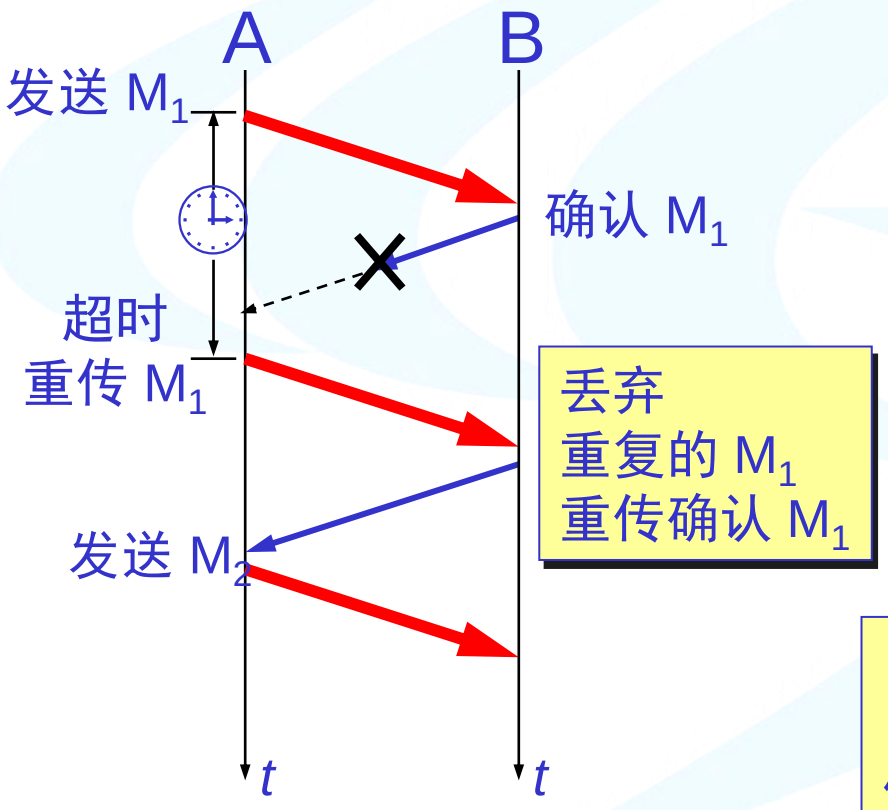


(b) 超时重传

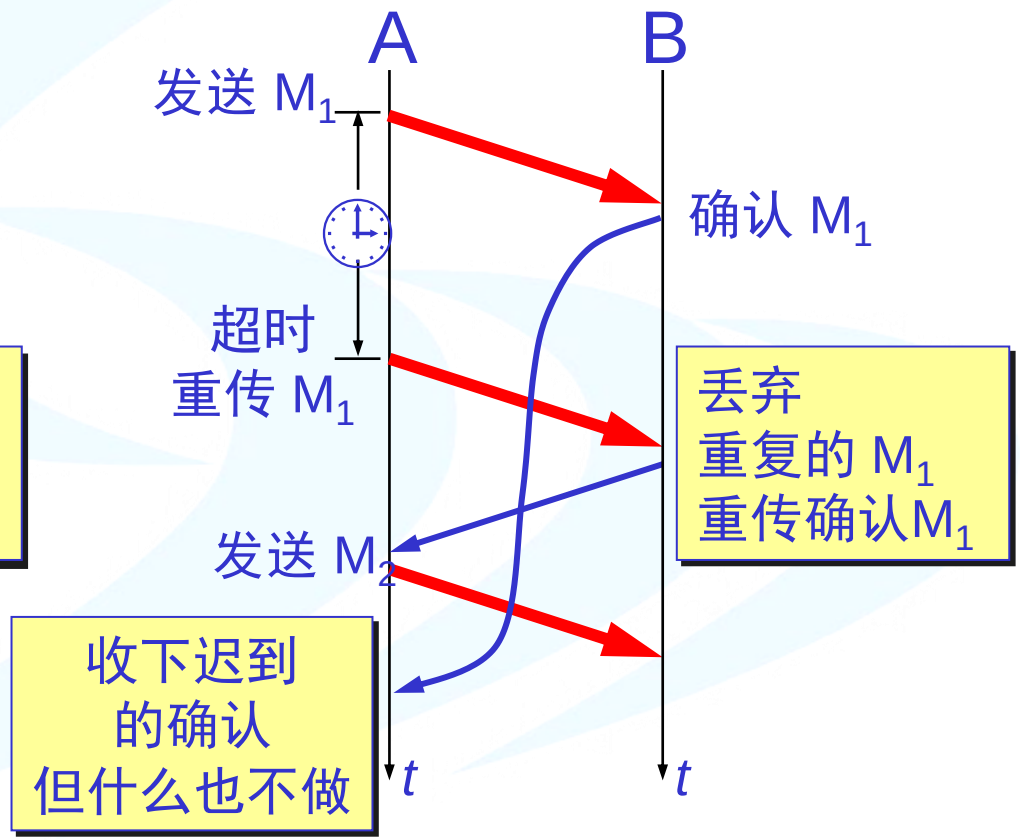
请注意

- 在发送完一个分组后，必须暂时保留已发送的分组的副本。
- 分组和确认分组都必须进行编号。
- 超时计时器的重传时间应当比数据在分组传输的平均往返时间更长一些。

确认丢失和确认迟到



(a) 确认丢失



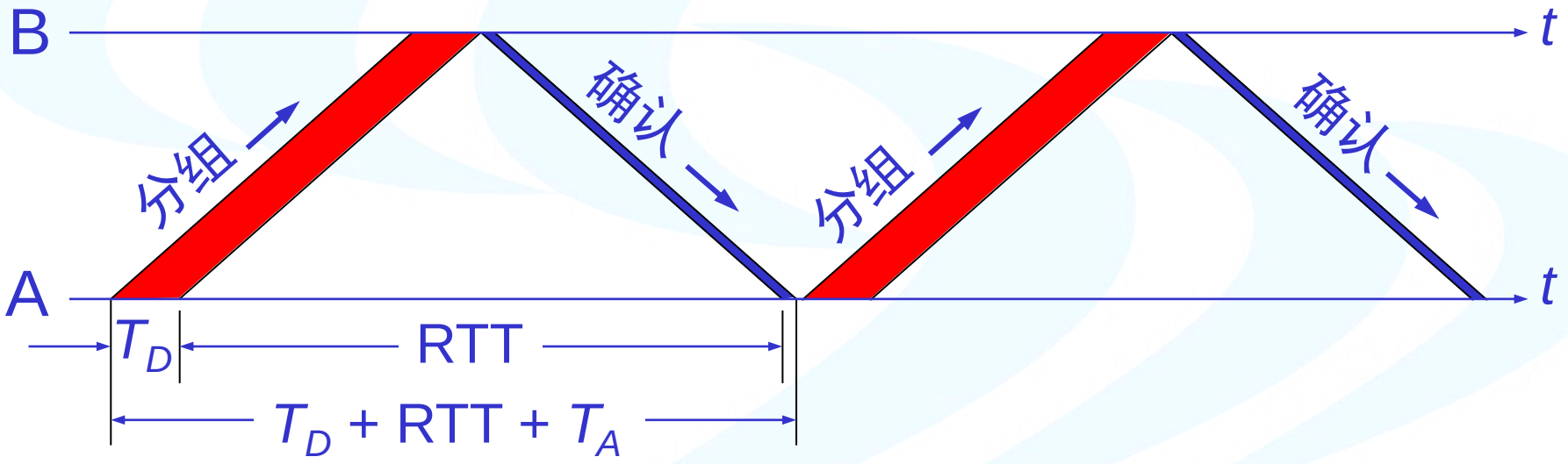
(b) 确认迟到

可靠通信的实现

- 使用上述的确认和重传机制，我们就可以在不可靠的传输网络上实现可靠的通信。
- 这种可靠传输协议常称为自动重传请求 **ARQ (Automatic Repeat reQuest)**。
- **ARQ** 表明重传的请求是自动进行的。接收方不需要请求发送方重传某个出错的分组 。

信道利用率

- 停止等待协议的优点是简单，但缺点是信道利用率太低。



Performance of rdt3.0

- **rdt3.0 works, but performance stinks**
- **ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:**

$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\,\text{bps}} = 8\,\text{microseconds}$$

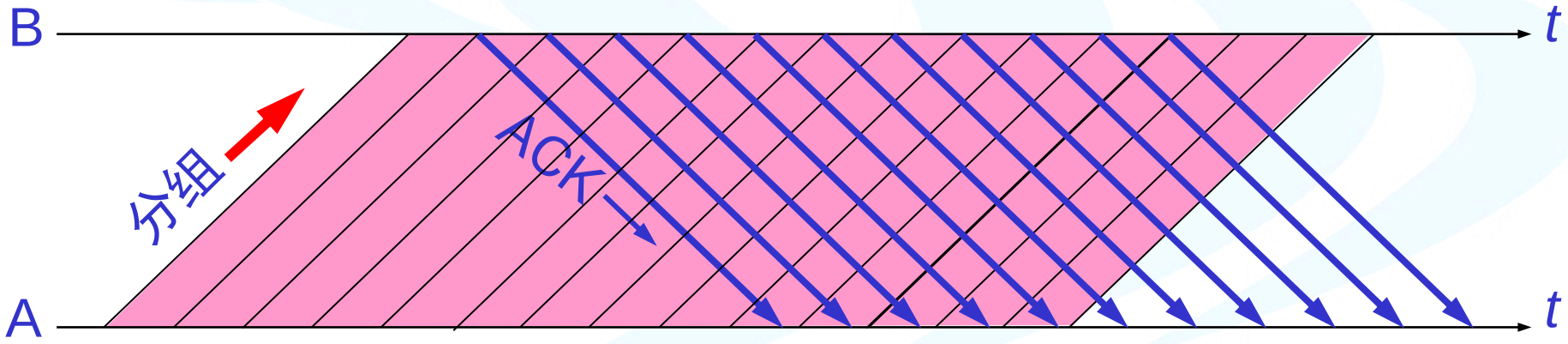
- U_{sender}: utilization – fraction of time sender busy sending

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

- 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources!

流水线传输

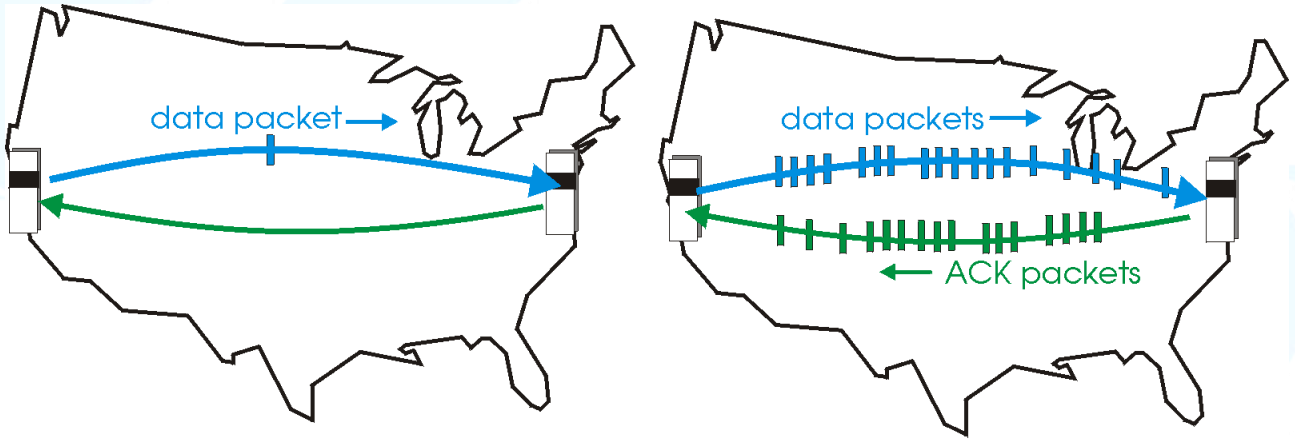
- 发送方可连续发送多个分组，不必每发完一个分组就停顿下来等待对方的确认。
- 由于信道上一直有数据不间断地传送，这种传输方式可获得很高的信道利用率。



Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

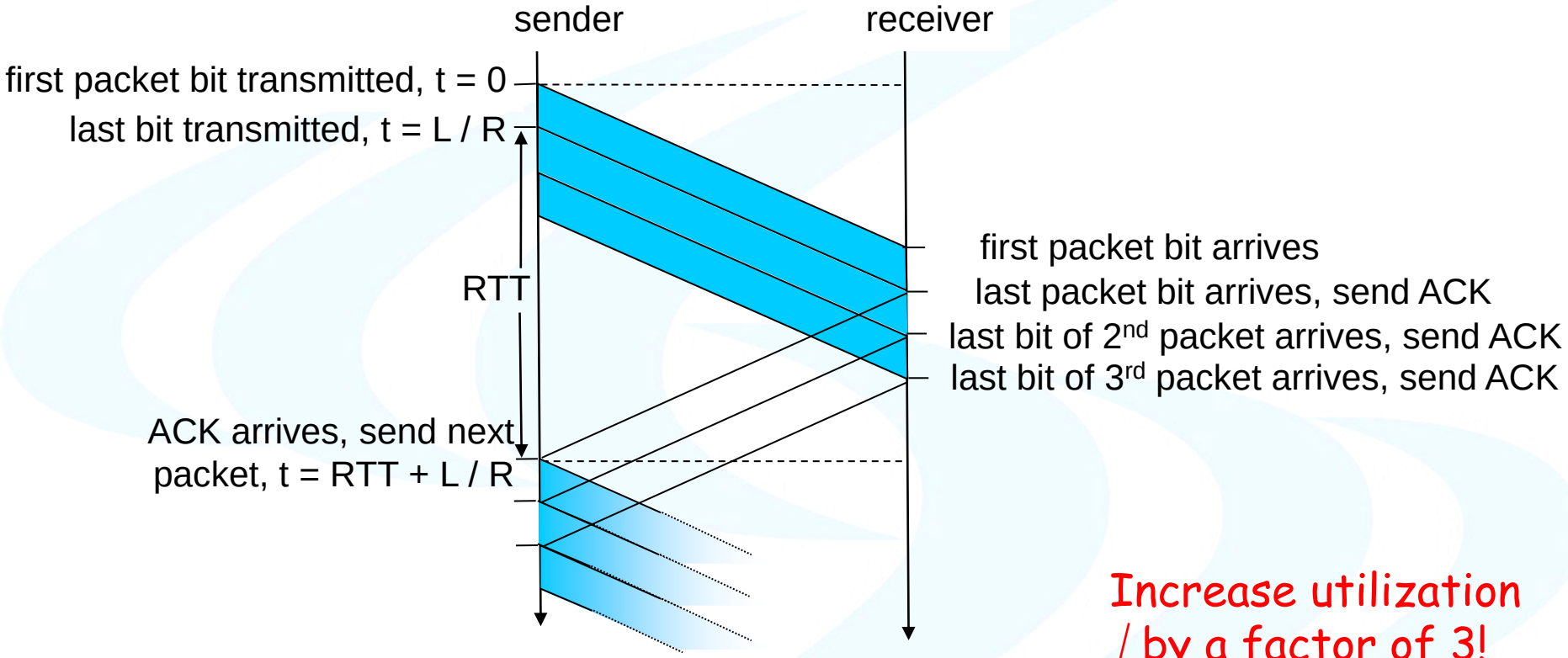
- **range of sequence numbers must be increased**
- **buffering at sender and/or receiver**



(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***

Pipelining: increased utilization



Increase utilization by a factor of 3!

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

Pipelining Protocols

Go-back-N: big picture:

- Sender can have up to N unacked packets in pipeline
- Rcvr only sends cumulative acks
  - Doesn't ack packet if there's a gap
- Sender has timer for oldest unacked packet
  - If timer expires, retransmit all unacked packets

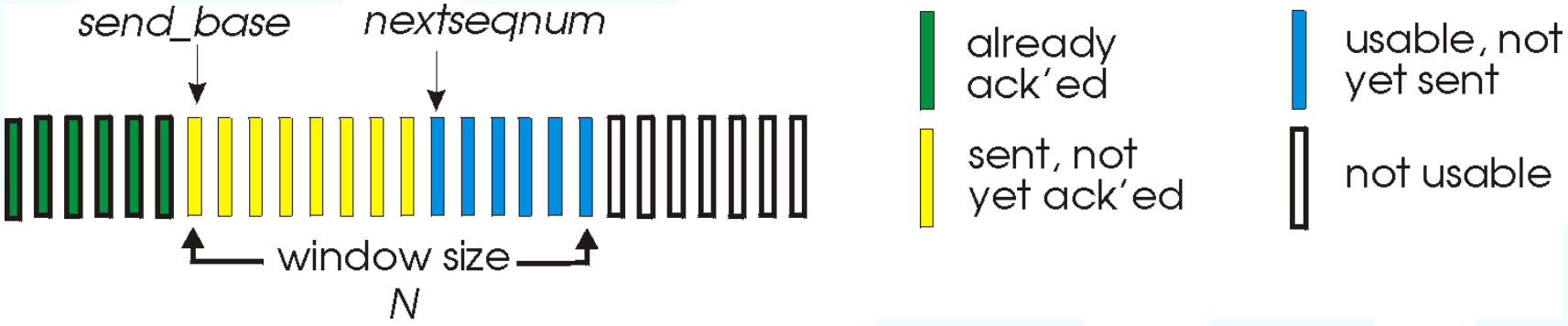
Selective Repeat: big pic

- Sender can have up to N unacked packets in pipeline
- Rcvr acks individual packets
- Sender maintains timer for each unacked packet
  - When timer expires, retransmit only unack packet

Go-Back-N

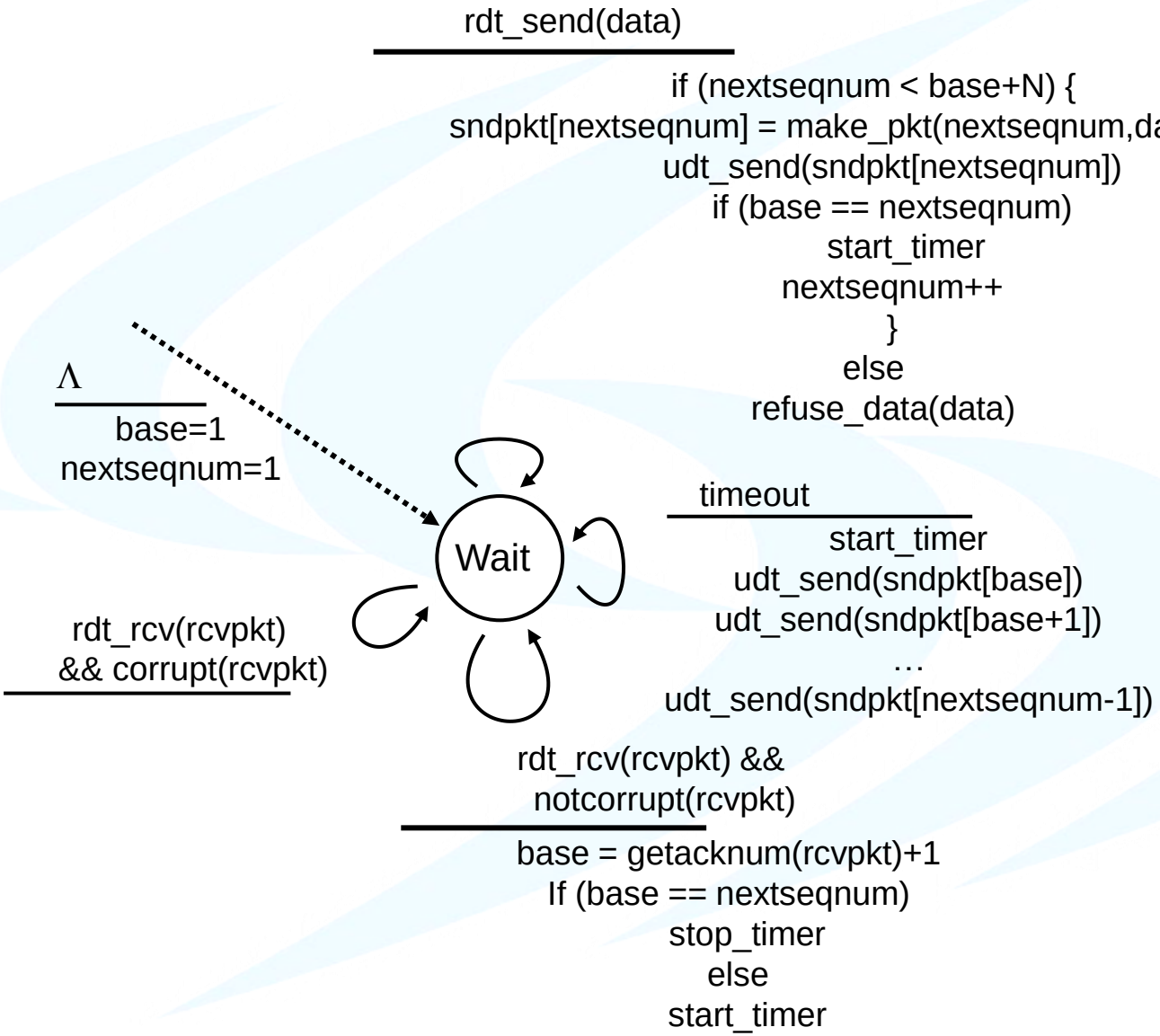
Sender:

- **k-bit seq # in pkt header**
- **"window" of up to N, consecutive unack'ed pkts allowed**

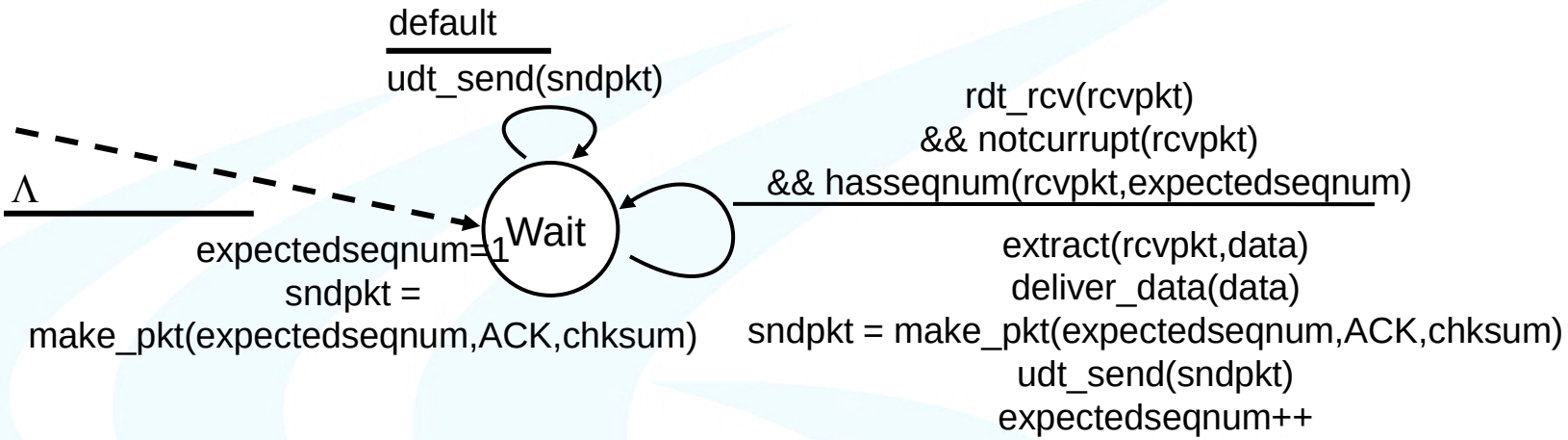


- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
    - timer for the oldest in-flight pkt
- *timeout(n)*: retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM



GBN: receiver extended FSM

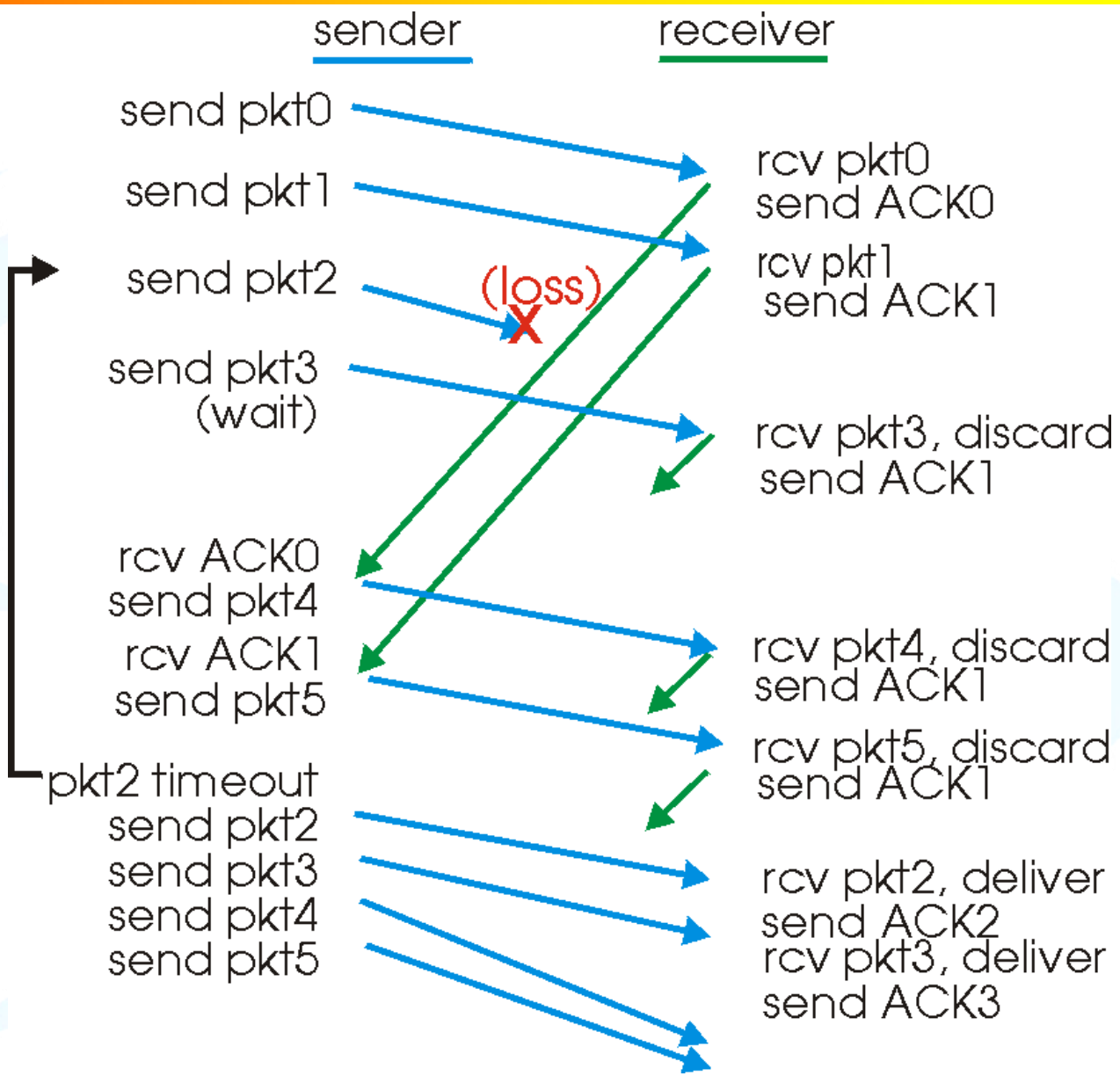


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

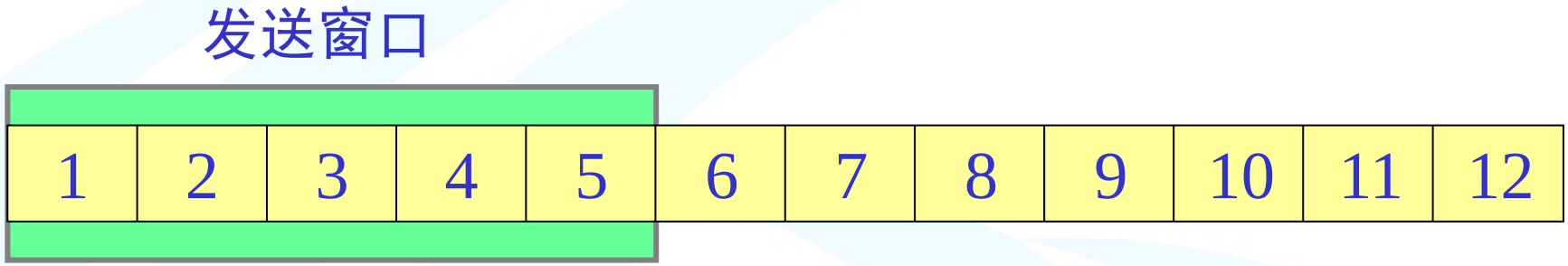
- **may generate duplicate ACKs**
- **need only remember `expectedseqnum`**

- **out-of-order pkt:**
  - **discard (don't buffer) -> no receiver buffering!**
  - **Re-ACK pkt with highest in-order seq #**

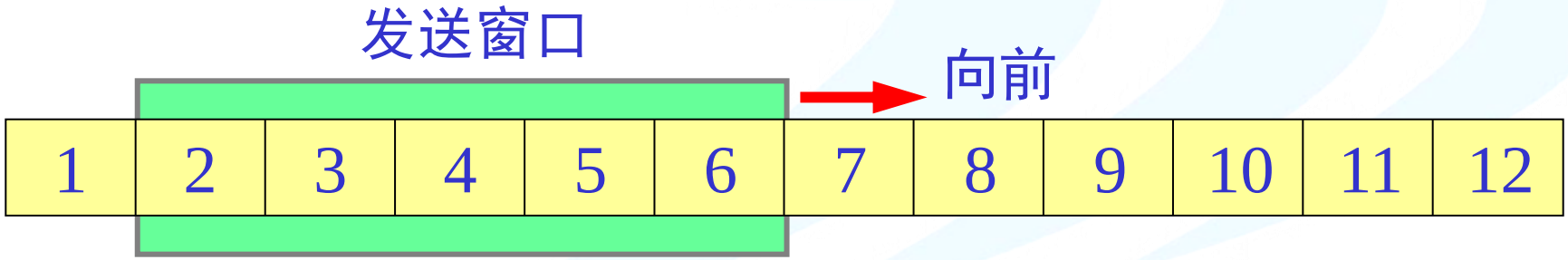
GBN in action



连续 ARQ 协议



(a) 发送方维持发送窗口（发送窗口是 5）



(b) 收到一个确认后发送窗口向前滑动

累积确认

- 接收方一般采用累积确认的方式。即不必对收到的分组逐个发送确认，而是对按序到达的最后一个分组发送确认，这样就表示：到这个分组为止的所有分组都已正确收到了。
- 累积确认有的优点是：容易实现，即使确认丢失也不必重传。缺点是：不能向发送方反映出接收方已经正确收到的所有分组的信息。

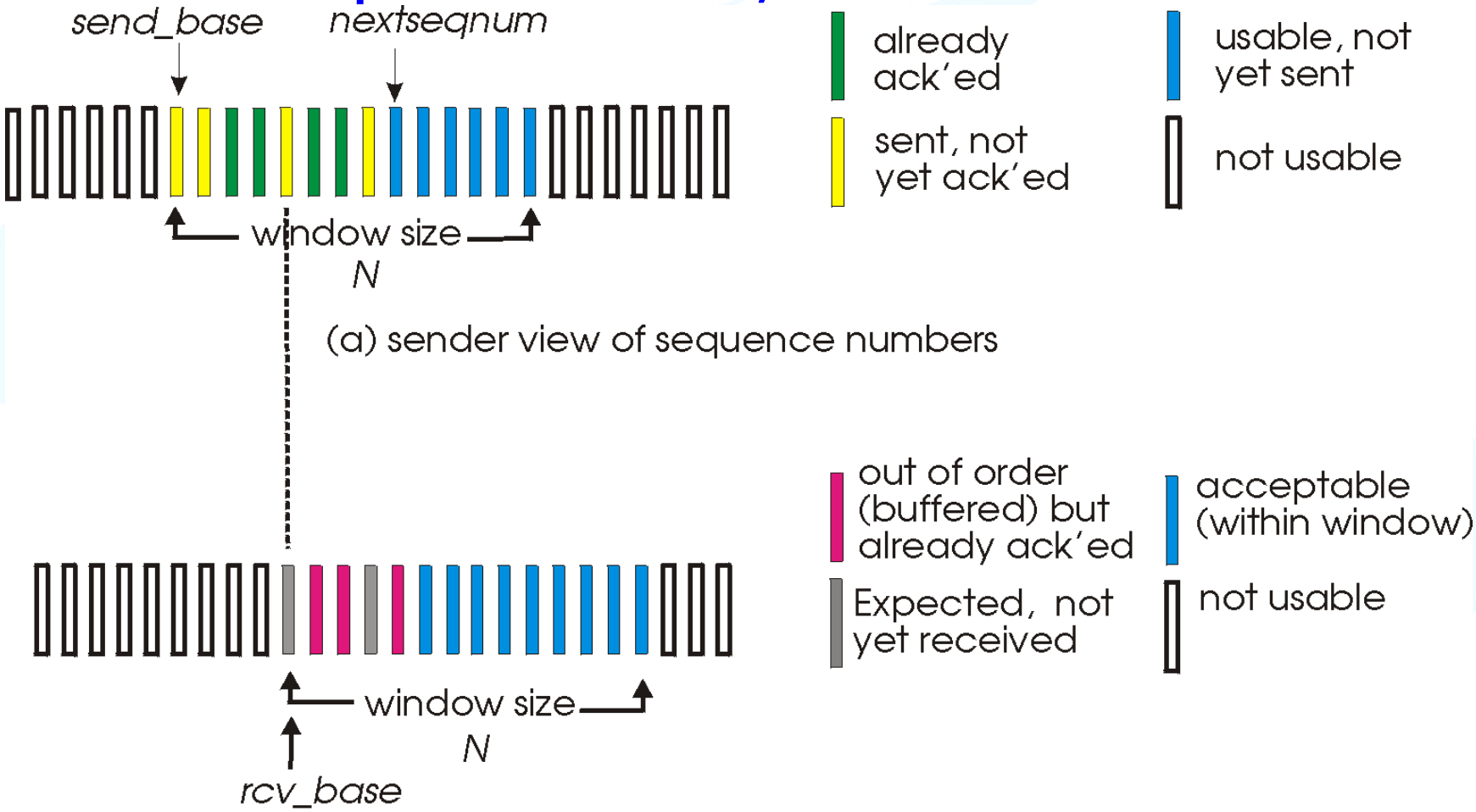
Go-back-N（回退 N）

- 如果发送方发送了前 **5** 个分组，而中间的第 **3** 个分组丢失了。这时接收方只能对前两个分组发出确认。发送方无法知道后面三个分组的下落，而只好把后面的三个分组都再重传一次。
- 这就叫做 **Go-back-N**（回退 **N**），表示需要再退回来重传已发送过的 ***N*** 个分组。
- 可见当通信线路质量不好时，连续 **ARQ** 协议会带来负面的影响。

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
  - again 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]:

- **mark pkt n as received**
- **if n 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

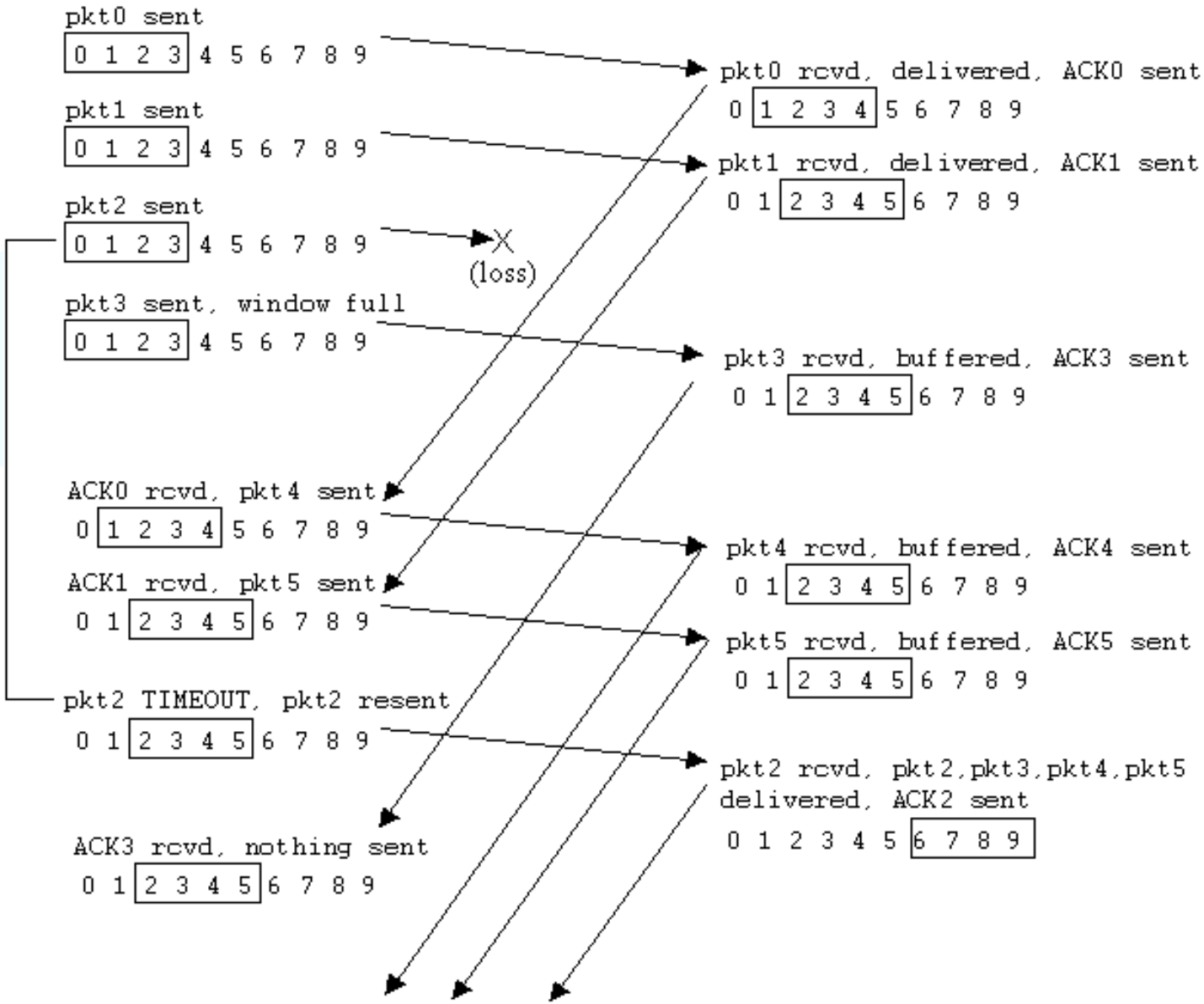
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



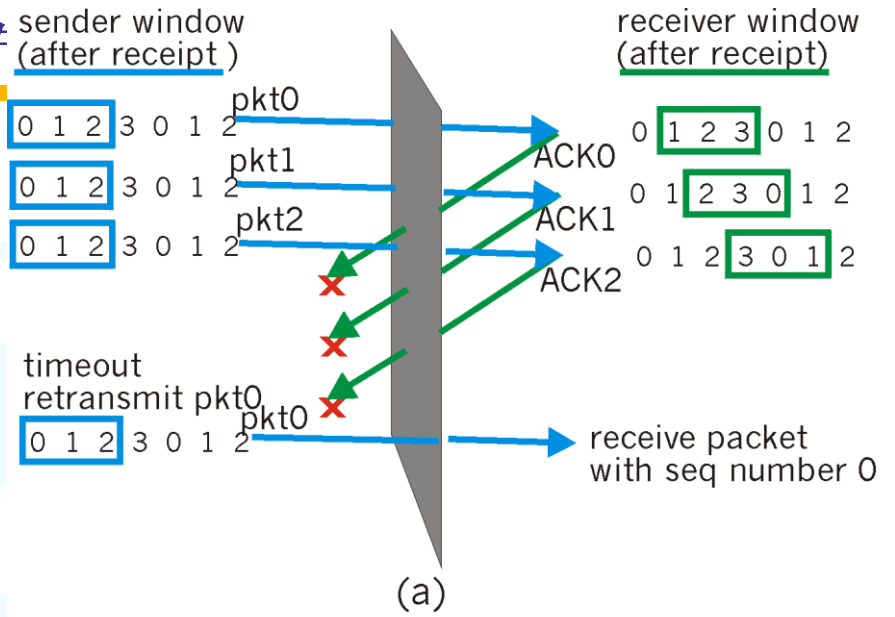
Selective repeat: dilemma

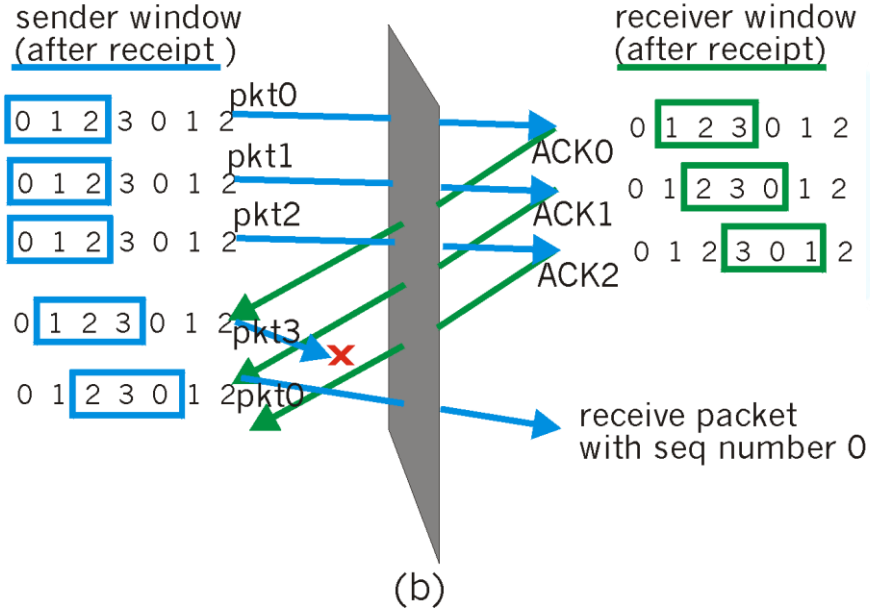
Example:

- **seq #'s: 0, 1, 2, 3**
- **window size=3**

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

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





TCP 可靠通信的具体实现

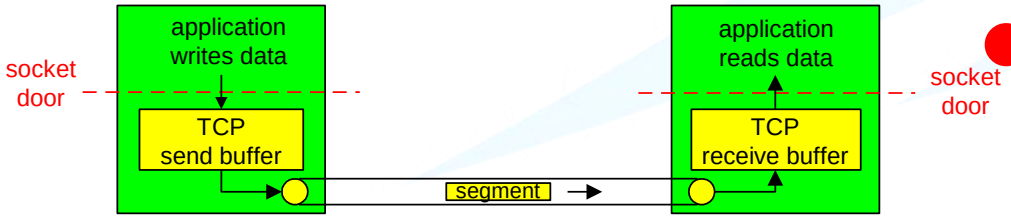
- **TCP** 连接的每一端都必须设有两个窗口——一个发送窗口和一个接收窗口。
- **TCP** 的可靠传输机制用字节的序号进行控制。**TCP** 所有的确认都是基于序号而不是基于报文段。
- **TCP** 两端的四个窗口经常处于动态变化之中。
- **TCP**连接的往返时间 **RTT** 也不是固定不变的。需要使用特定的算法估算较为合理的重传时间。

4.5 Connection-oriented transport: TCP

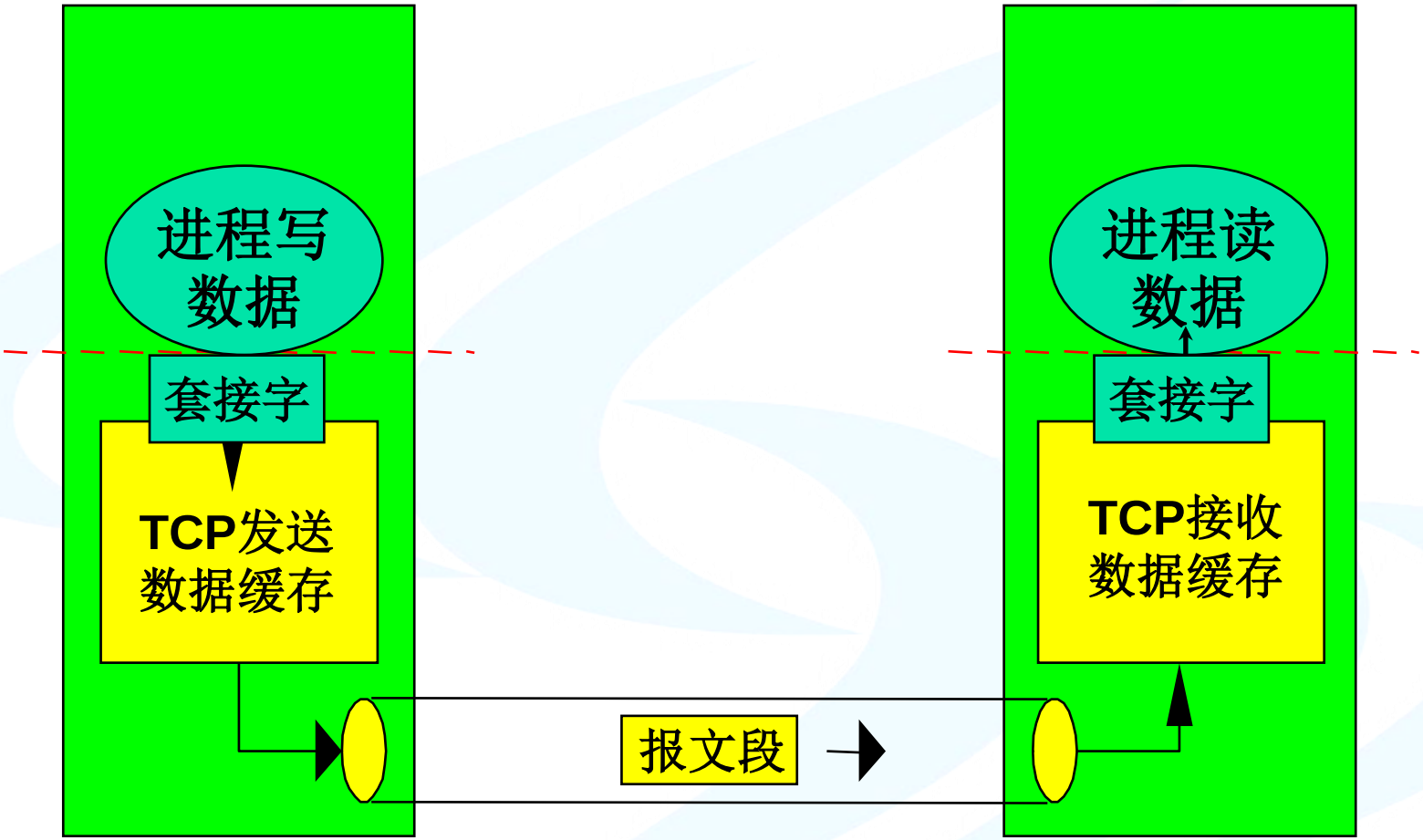
- 4.5.1 segment structure
- 4.5.2 reliable data transfer
- 4.5.3 flow control
- 4.5.4 connection management

TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581

- **point-to-point:**
  - **one sender, one receiver**
- **reliable, in-order *byte steam:***
  - **no "message boundaries"**
- **pipelined:**
  - **TCP congestion and flow control set window size**
- ***send & receive buffers***

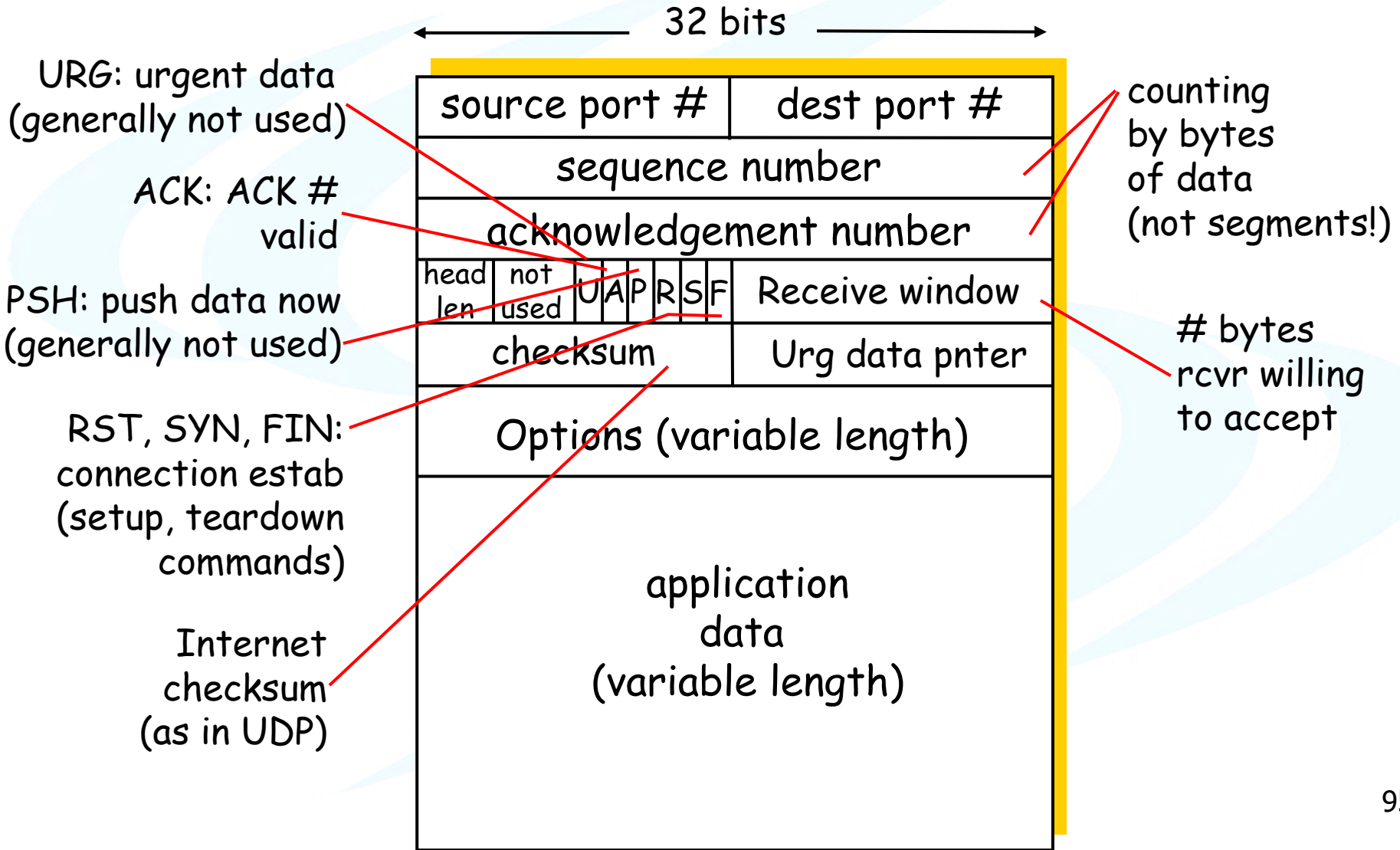


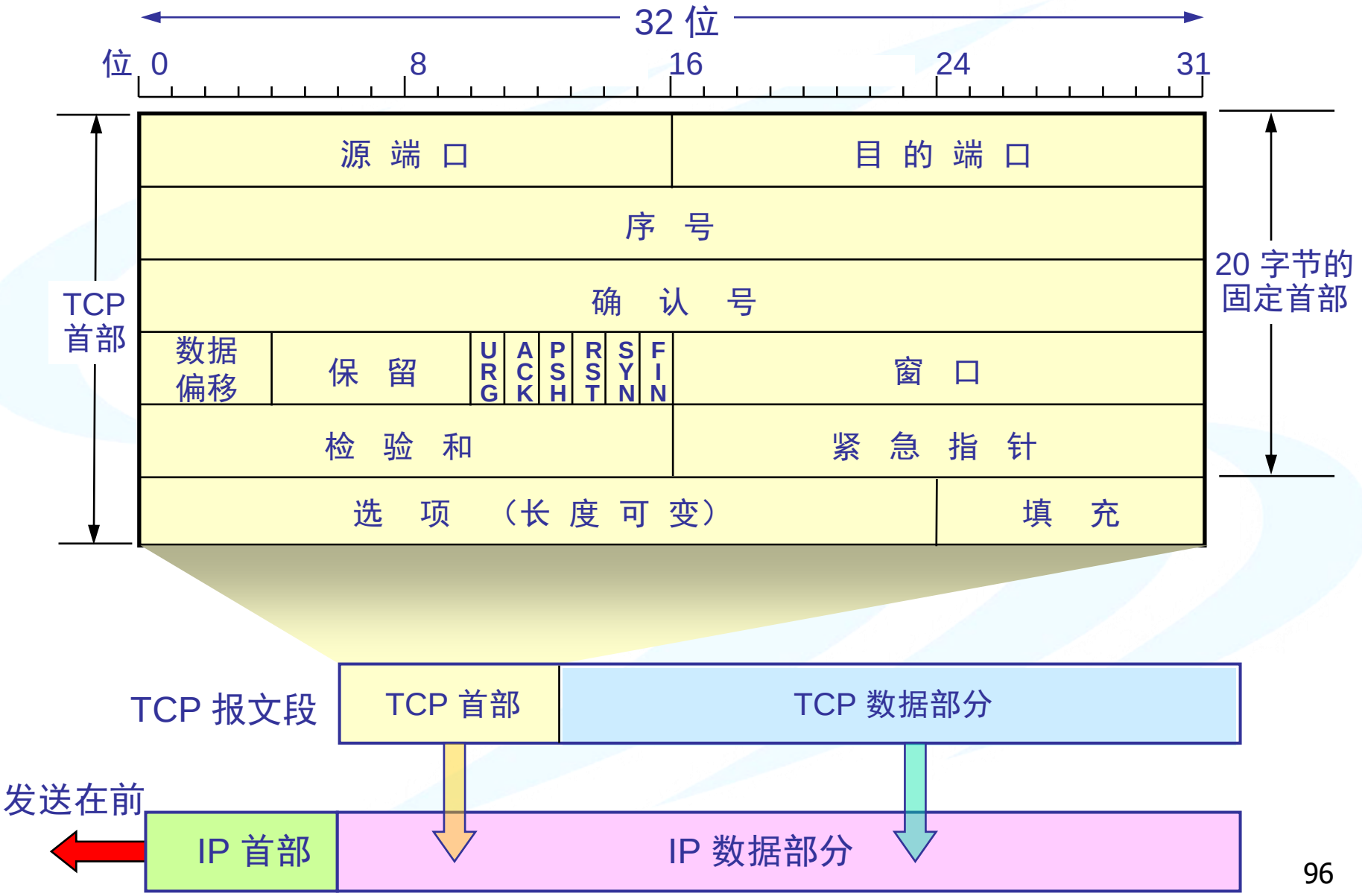
- **full duplex data:**
  - **bi-directional data flow in same connection**
  - **MSS: maximum segment size**
- **connection-oriented:**
  - **handshaking (exchange of control msgs) init's sender, receiver state before data exchange**
- **flow controlled:**
  - **sender will not overwhelm receiver**



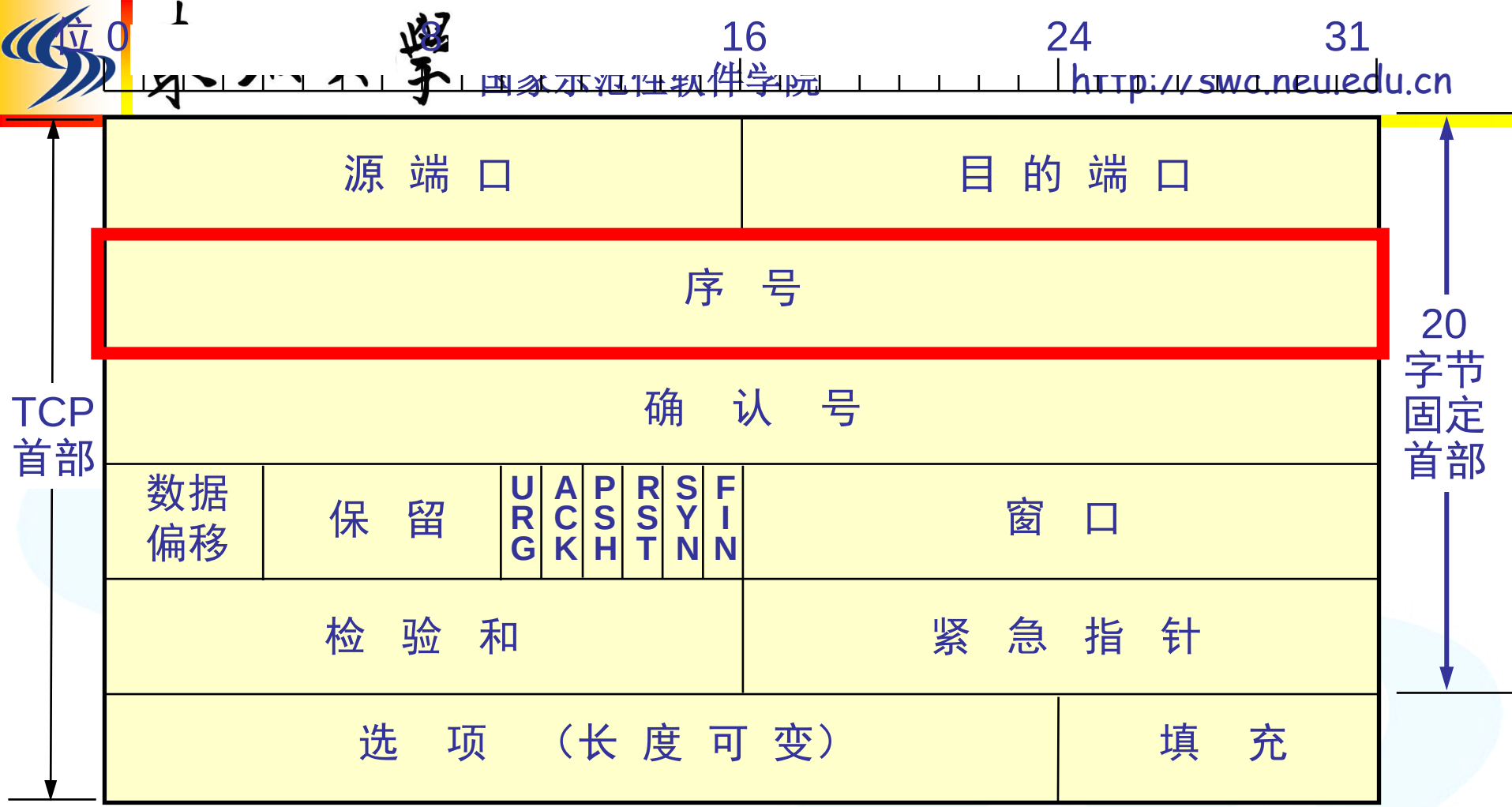
收发缓冲区

4.5.1 TCP segment structure



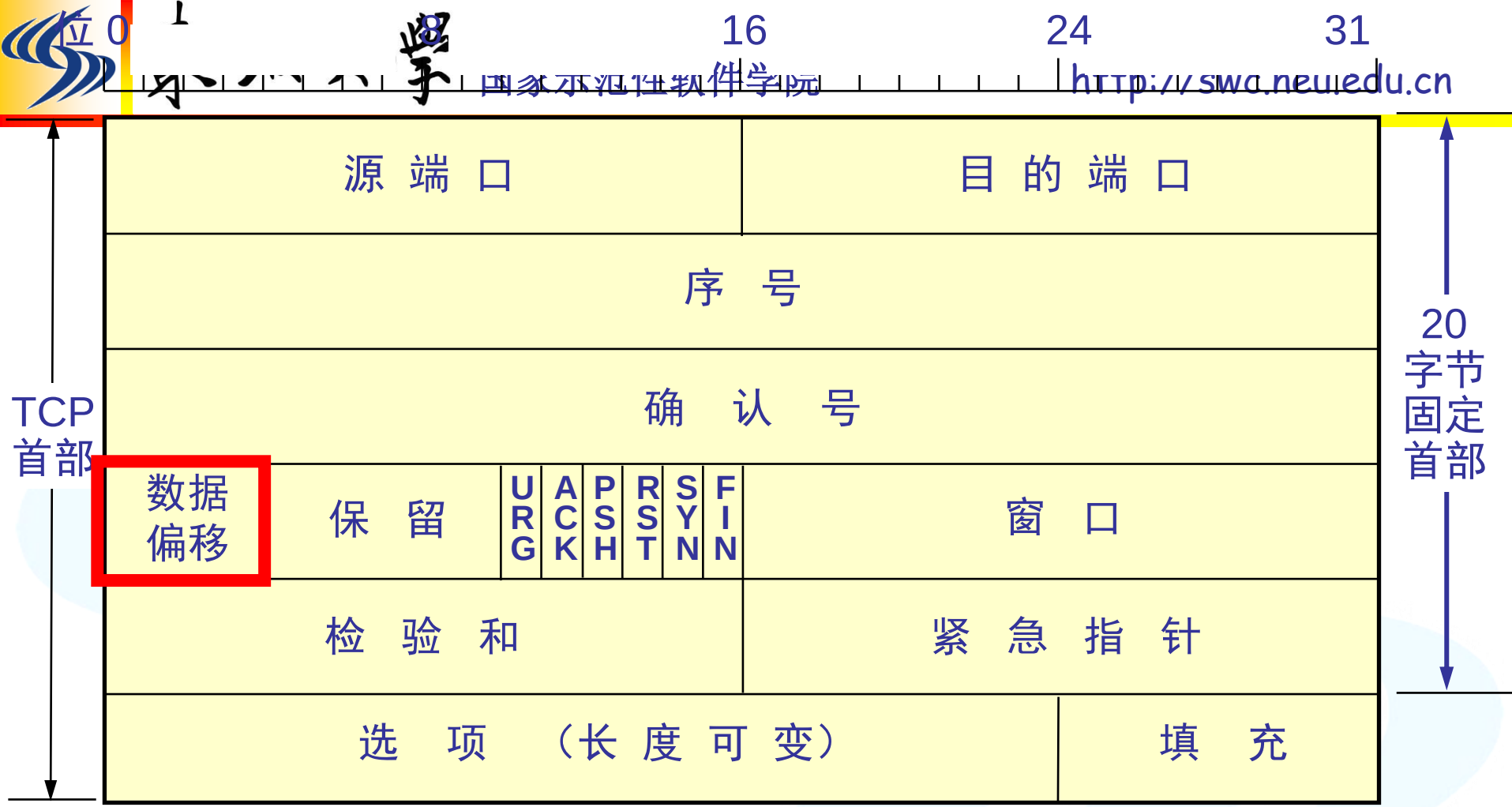

32 位
位 0
8
16
24
31
源 端 口
目 的 端 口
序 号
确 认 号
数据偏移
保 留
URG
ACK
PSH
RST
SYN
FIN
窗 口
检 验 和
紧 急 指 针
选 项 （长 度 可 变）
填 充
TCP 首部
20 字节的固定首部
TCP 报文段
TCP 首部
TCP 数据部分
发送在前
IP 首部
IP 数据部分

源端口和目的端口字段——各占 2 字节。端口是运输层与应用层的服务接口。运输层的复用和分用功能都要通过端口才能实现。



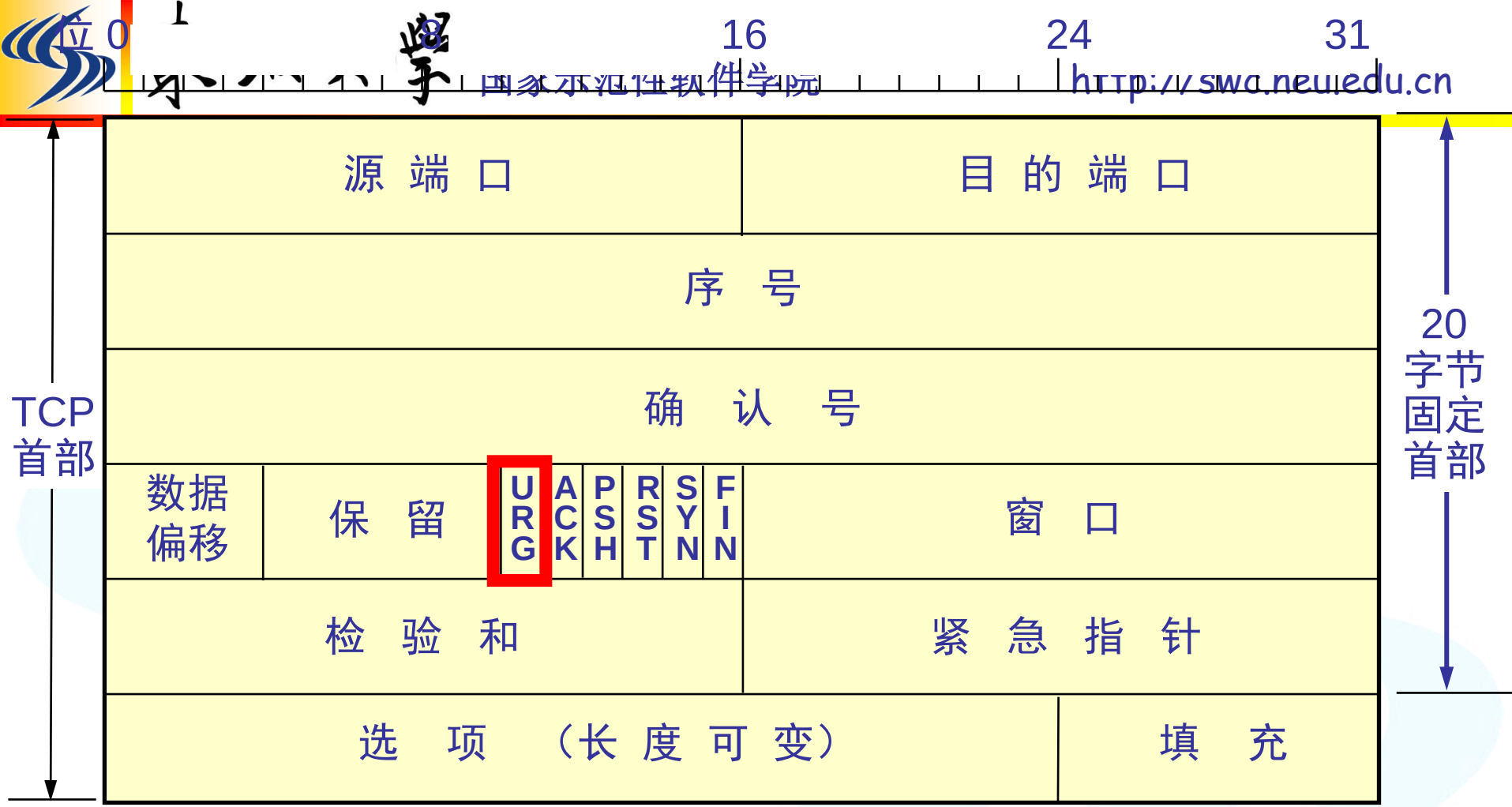
序号字段——占 4 字节。TCP 连接中传送的数据流中的每一个字节都编上一个序号。序号字段的值则指的是本报文段所发送的数据的第一个字节的序号。

确认号字段——占 4 字节，是期望收到对方的下一个报文段的数据的第一个字节的序号。



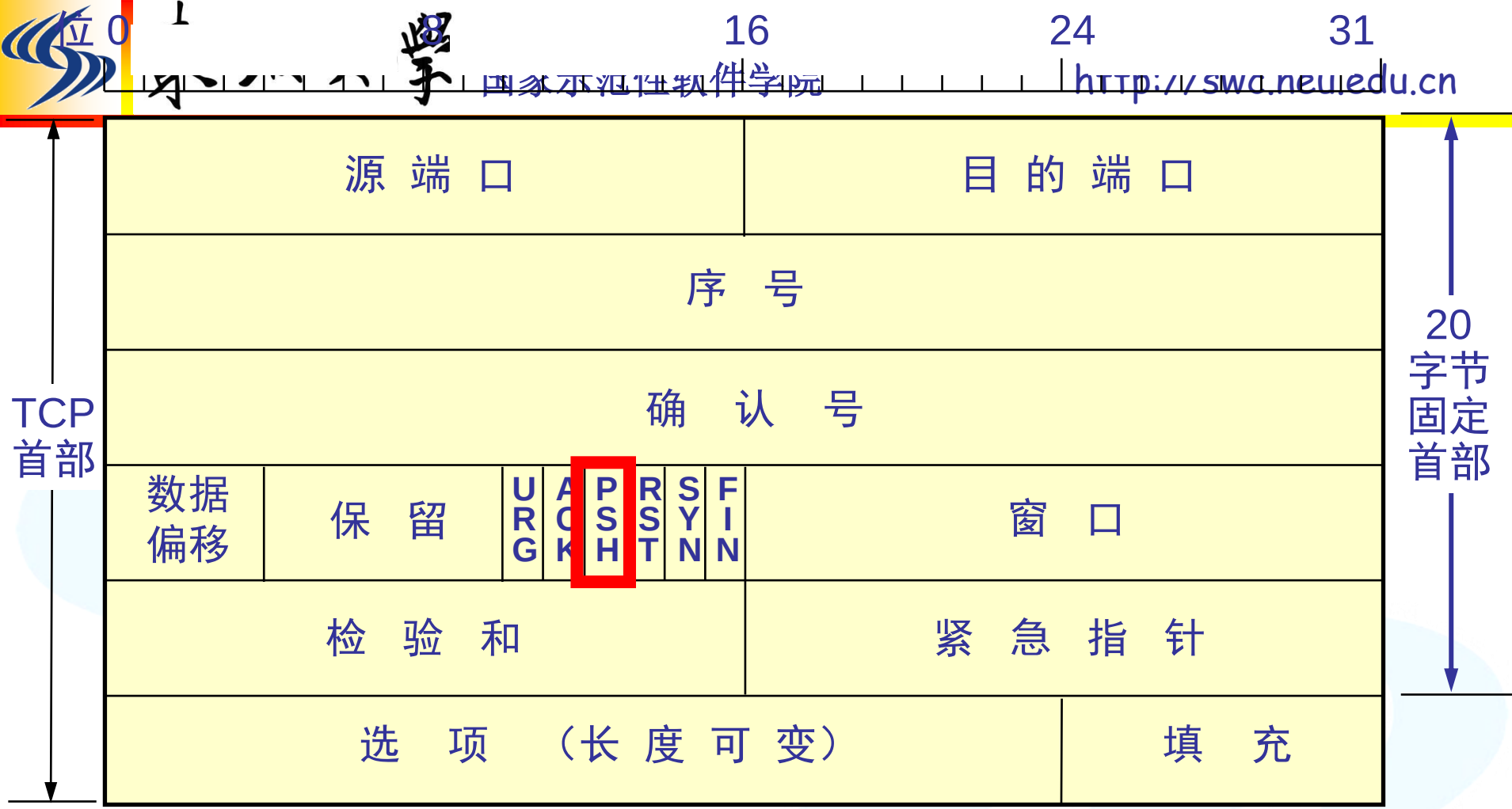
数据偏移（即首部长度）——占 4 位，它指出 TCP 报文段的数据起始处距离 TCP 报文段的起始处有多远。“数据偏移”的单位是 32 位字（以 4 字节为计算单位）。

保留字段——占 6 位，保留为今后使用，但目前应置为 0。



紧急 URG —— 当 URG = 1 时，表明紧急指针字段有效。它告诉系统此报文段中有紧急数据，应尽快传送(相当于高优先级的数据)。

确认 ACK —— 只有当 ACK = 1 时确认号字段才有效。当 ACK = 0 时，确认号无效。

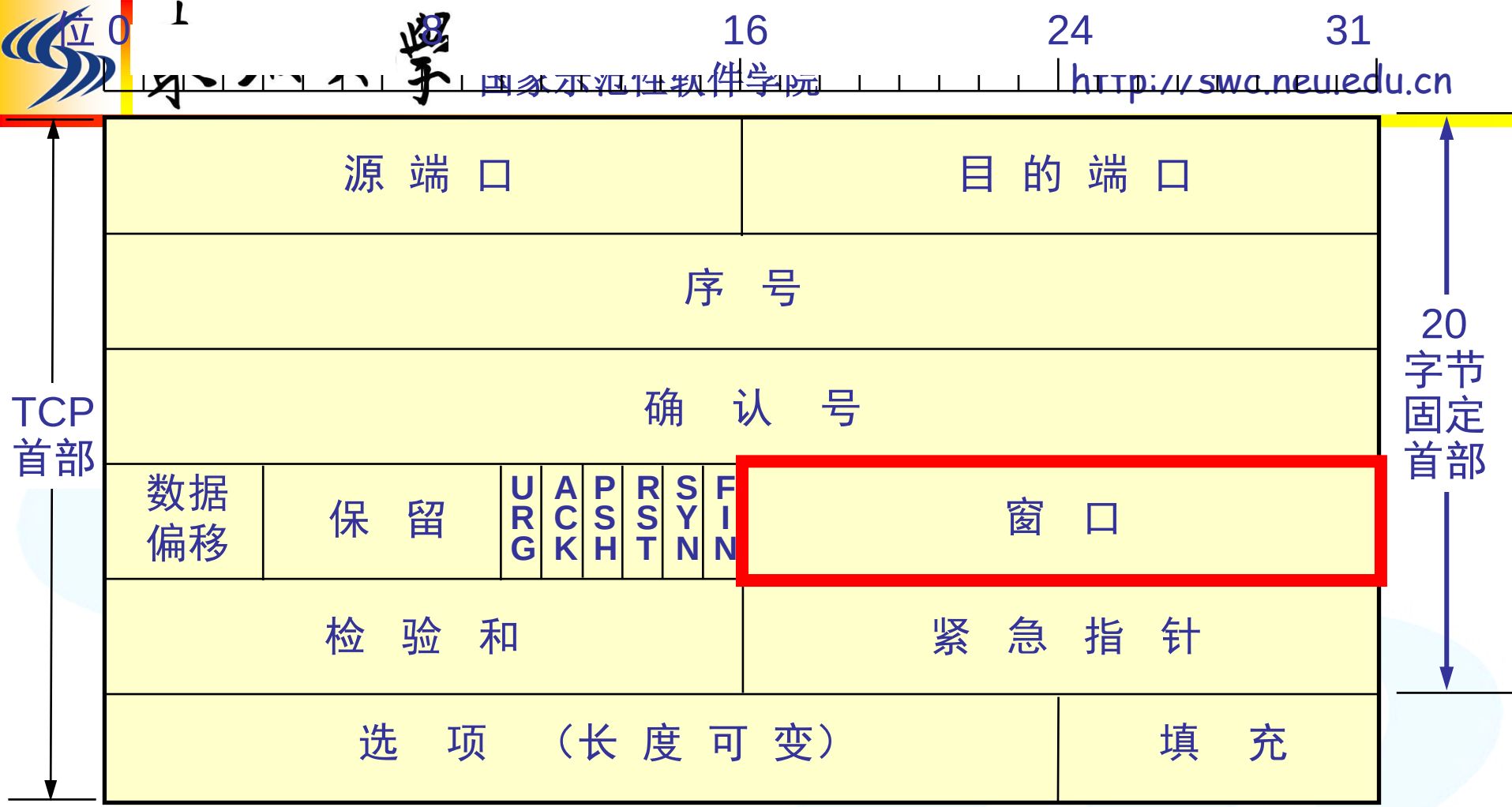


推送 PSH (PuSH) —— 接收 TCP 收到 PSH = 1 的报文段，就尽快地交付接收应用进程，而不再等到整个缓存都填满了后再向上交付。

复位 RST (ReSeT) —— 当 RST = 1 时，表明 TCP 连接中出现严重差错（如由于主机崩溃或其他原因），必须释放连接，然后再重新建立运输连接。

同步 SYN —— 同步 SYN = 1 表示这是一个连接请求或连接接受报文。

终止 FIN (FINis) —— 用来释放一个连接。FIN = 1 表明此报文段的发送端的数据已发送完毕，并要求释放运输连接。

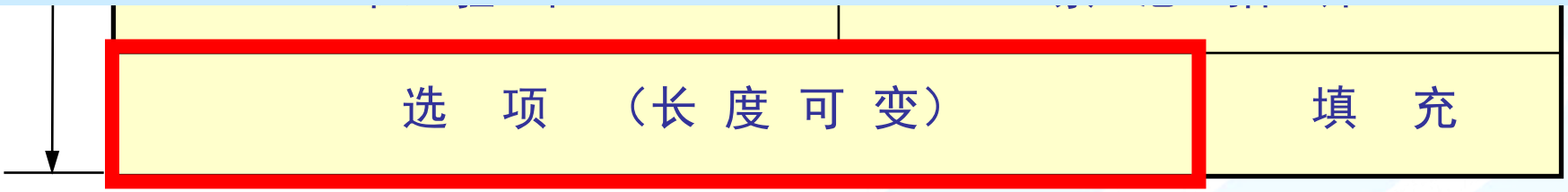


窗口字段 —— 占 2 字节，用来让对方设置发送窗口的依据，单位为字节。

检验和 —— 占 2 字节。检验和字段检验的范围包括首部和数据这两部分。在计算检验和时，要在 TCP 报文段的前面加上 12 字节的伪首部。

紧急指针字段 —— 占 16 位，指出在本报文段中紧急数据共有多少个字节（紧急数据放在本报文段数据的最前面）。

MSS (Maximum Segment Size)
是 TCP 报文段中的数据字段的最大长度。
数据字段加上 TCP 首部
才等于整个的 TCP 报文段。



选项字段 —— 长度可变。TCP 最初只规定了一种选项，即最大报文段长度 MSS。MSS 告诉对方 TCP：“我的缓存所能接收的报文段的数据字段的最大长度是 MSS 个字节。”

其他选项

- 窗口扩大选项——占 3 字节，其中有一个字节表示移位值 S。新的窗口值等于 TCP 首部中的窗口位数增大到(16 + S)，相当于把窗口值向左移动 S 位后获得实际的窗口大小。
- 时间戳选项——占10 字节，其中最主要的字段时间戳值字段（4 字节）和时间戳回送回答字段（4 字节）。
- 选择确认选项——在后面介绍。

填充字段 —— 这是为了使整个首部长度是 4 字节的整数倍。

TCP reliable data transfer

- **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

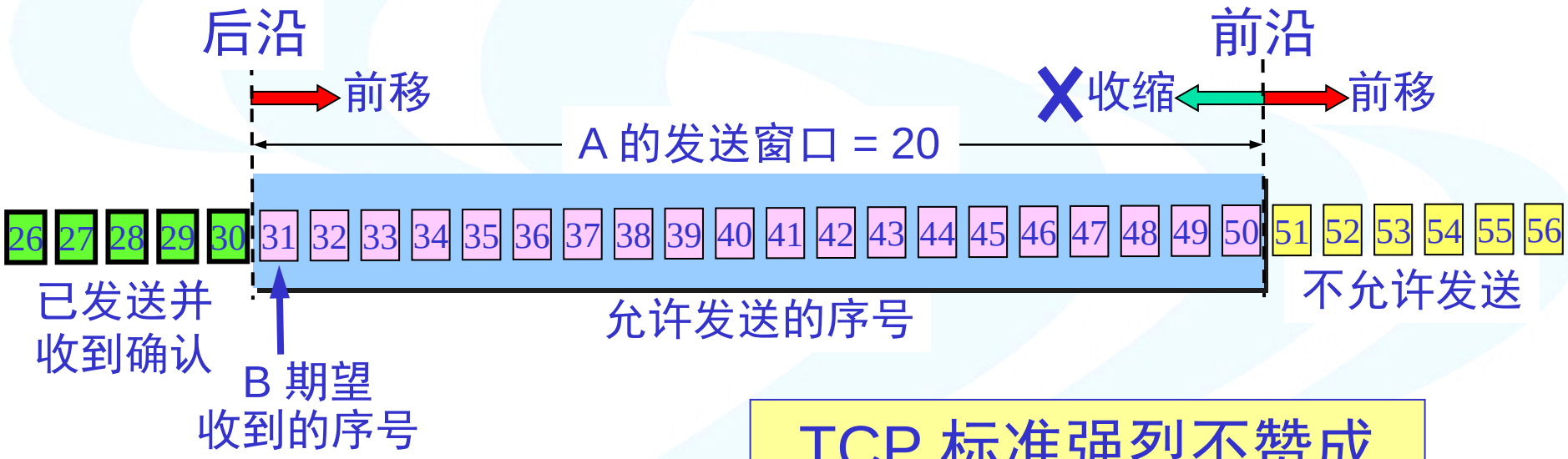
let's initially consider simplified TCP sender:

- **ignore duplicate acks**
- **ignore flow control, congestion control**

TCP 可靠传输的实现

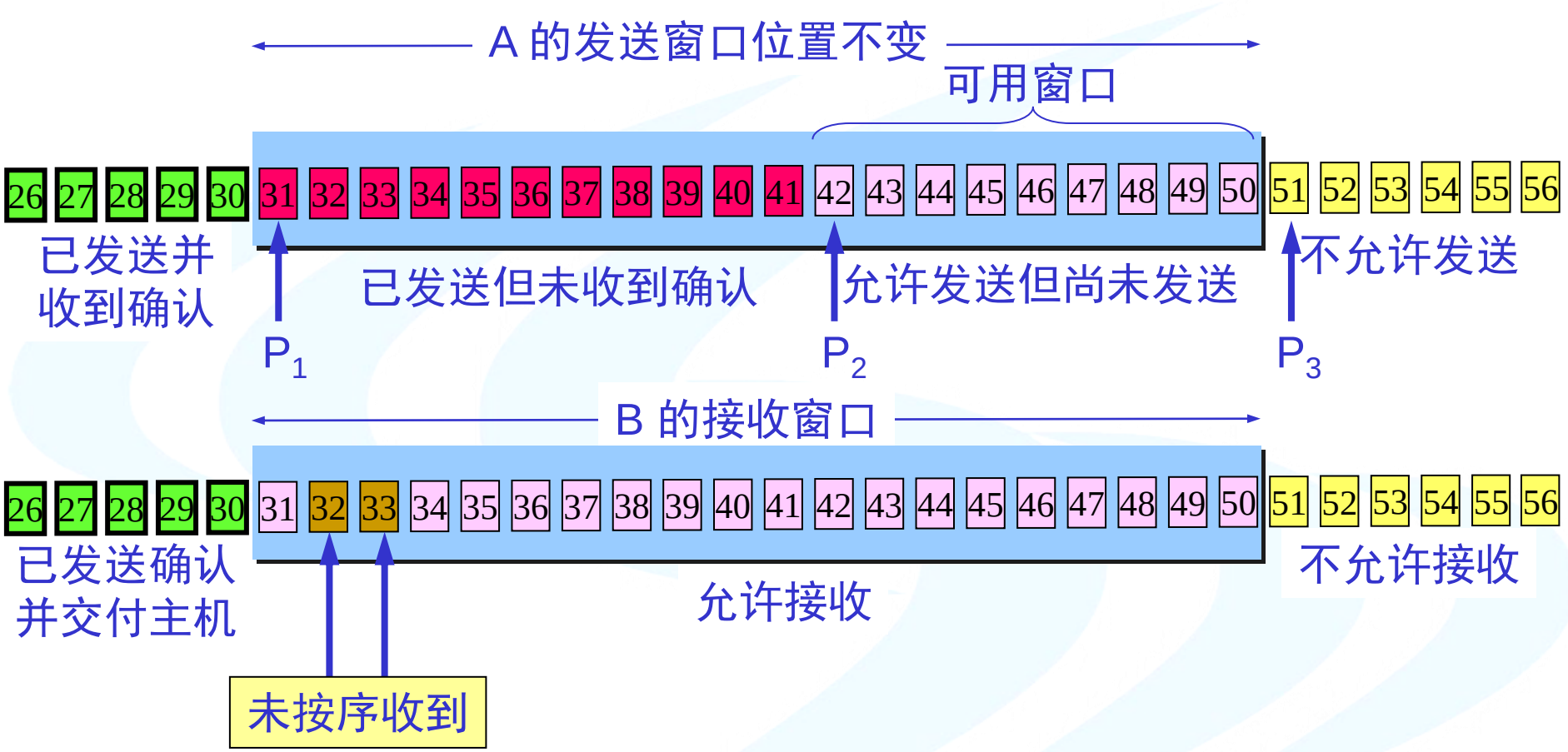
以字节为单位的滑动窗口

根据 B 给出的窗口值
A 构造出自己的发送窗口



TCP 标准强烈不赞成
发送窗口前沿向后收缩

A 发送了 11 个字节的数据

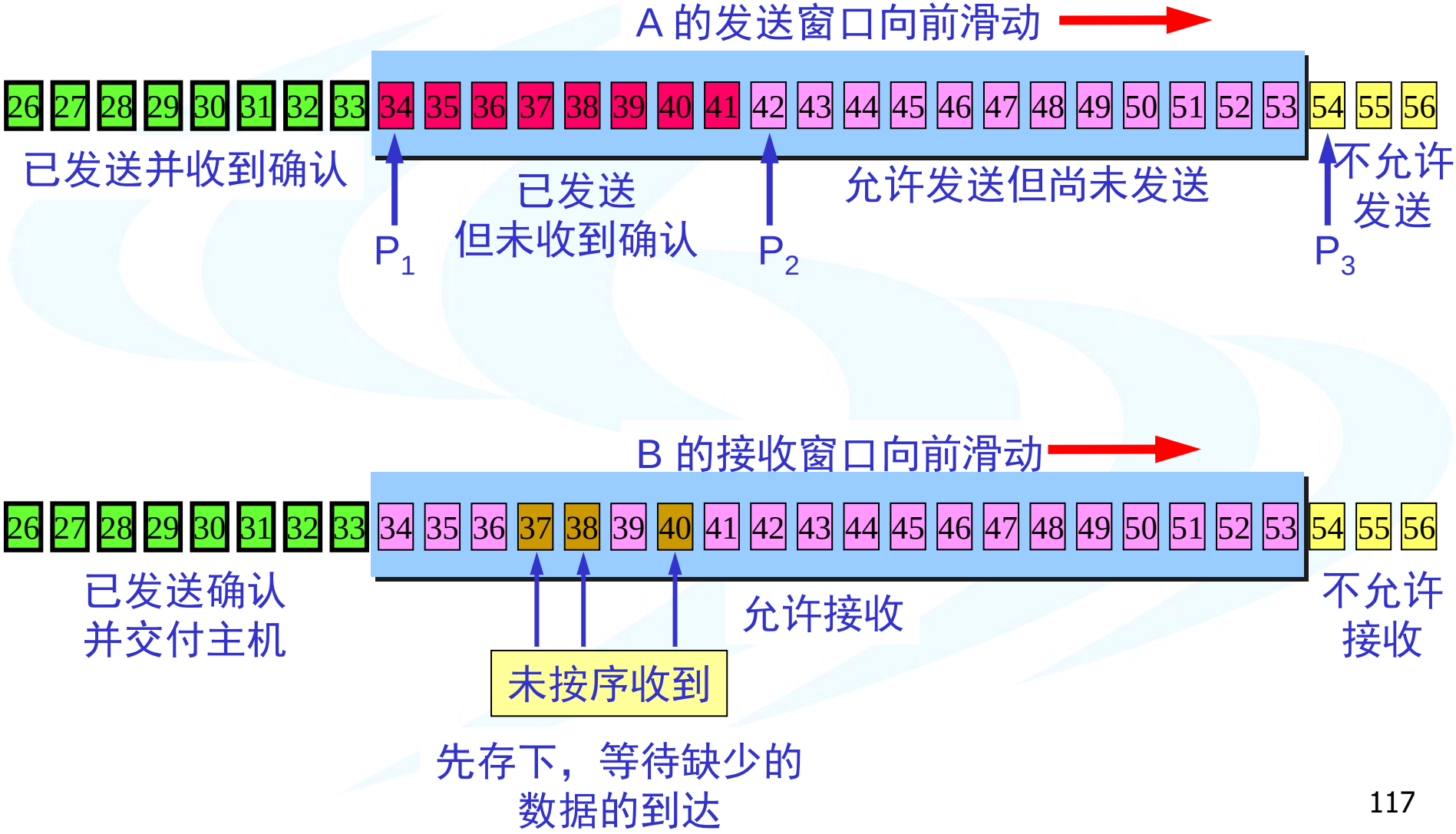


$P_3 - P_1$ = A 的发送窗口（又称为通知窗口）

$P_2 - P_1$ = 已发送但尚未收到确认的字节数

$P_3 - P_2$ = 允许发送但尚未发送的字节数（又称为可用窗口）

A 收到新的确认号，发送窗口向前滑动



A 的发送窗口内的序号都已用完，但还没有再收到确认，必须停止发送。

TCP seq. numbers, ACKs

sequence numbers:

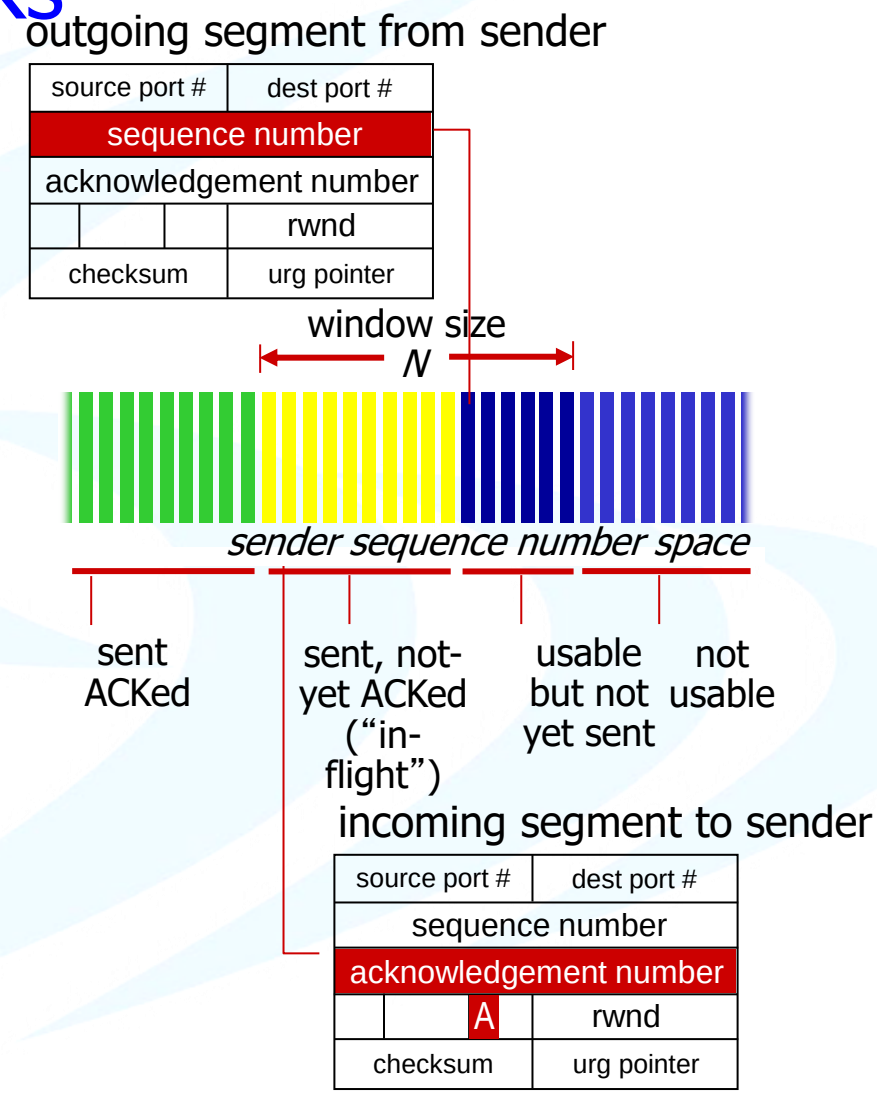
- byte stream "number" of first byte in segment's data

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 seq. #'s and ACKs

Seq. #'s:

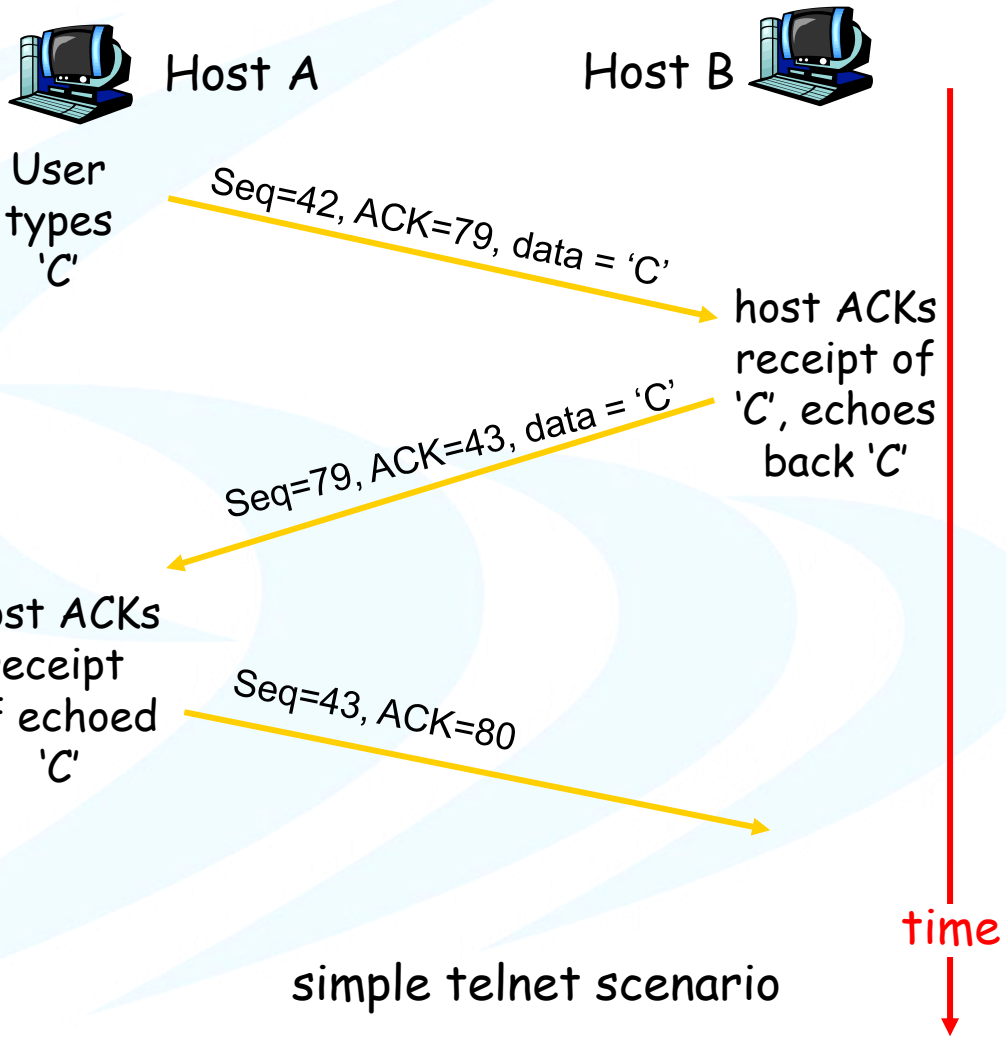
- byte stream "number" of first byte in segment's data

ACKs:

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

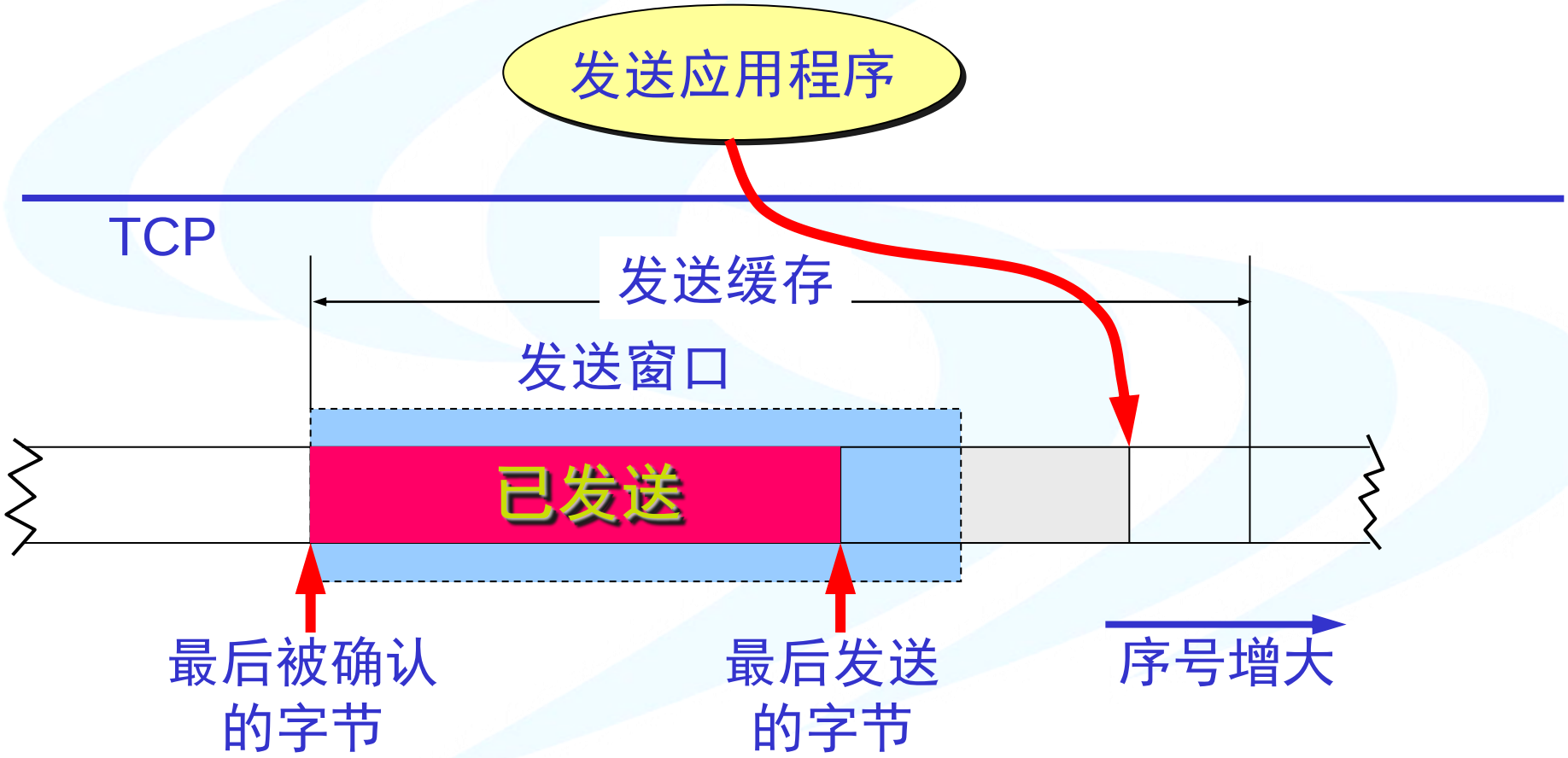
Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementor

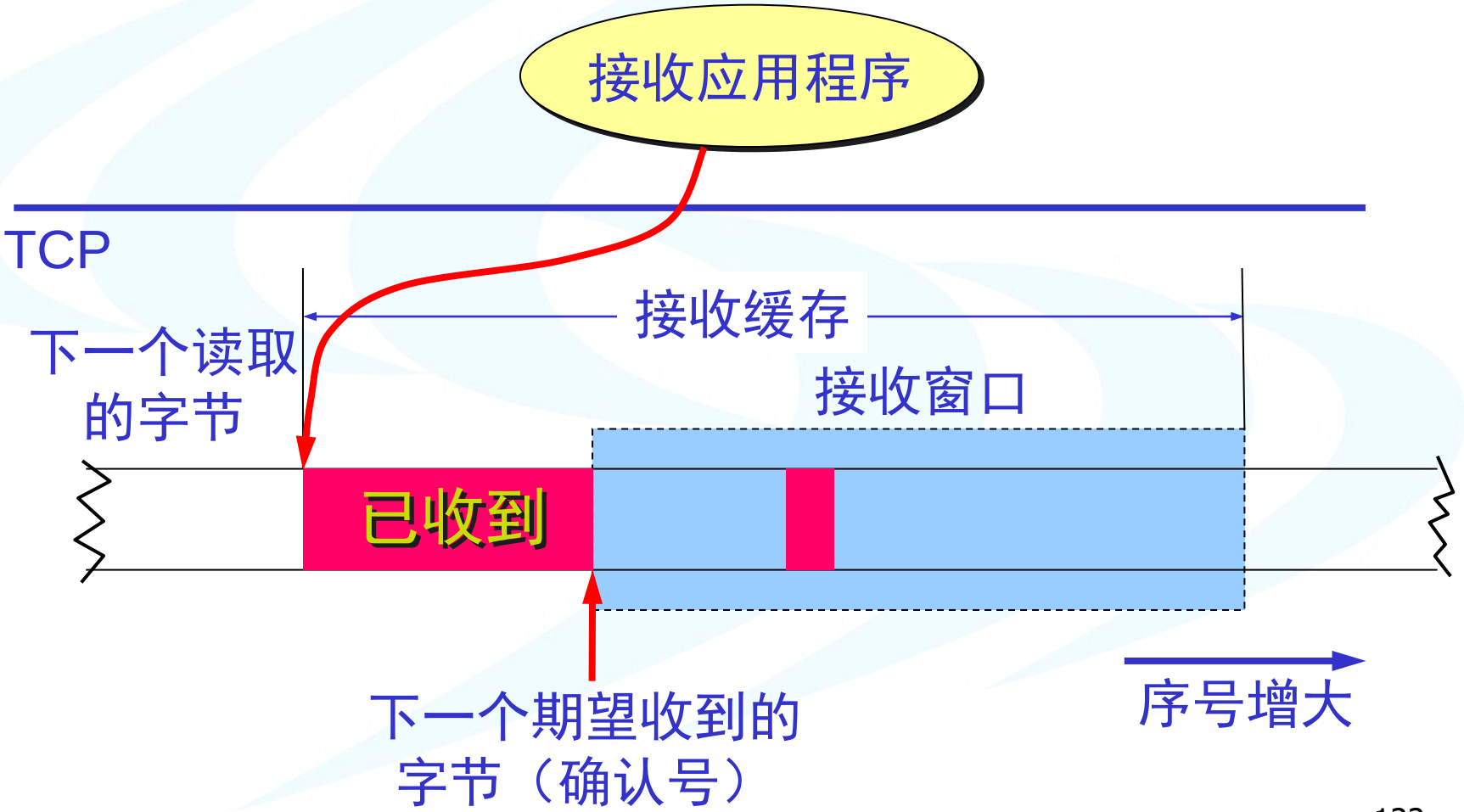


simple telnet scenario

发送缓存



接收缓存



发送缓存与接收缓存的作用

- 发送缓存用来暂时存放：
  - 发送应用程序传送给发送方 TCP 准备发送的数据；
  - TCP 已发送出但尚未收到确认的数据。
- 接收缓存用来暂时存放：
  - 按序到达的、但尚未被接收应用程序读取的数据；
  - 不按序到达的数据。

需要强调三点

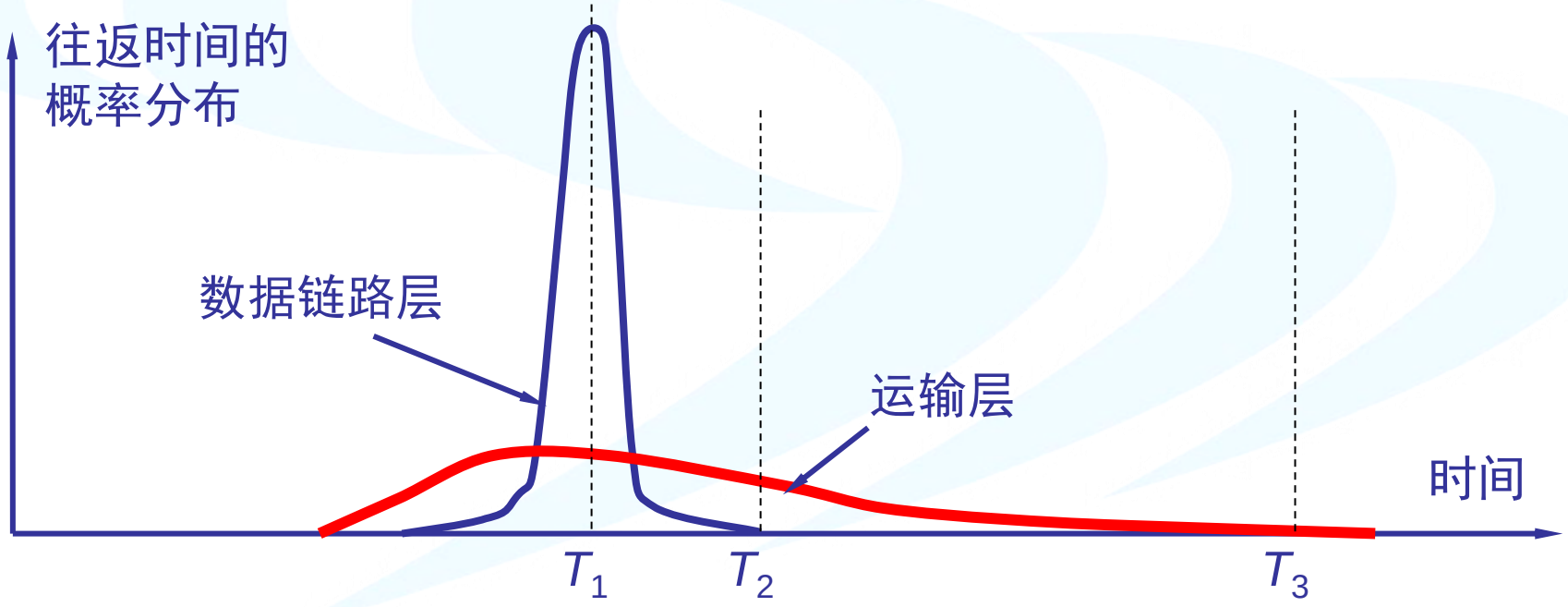
- **A** 的发送窗口并不总是和 **B** 的接收窗口一样大（因为有一定的时间滞后）。
- **TCP** 标准没有规定对不按序到达的数据应如何处理。通常是先临时存放在接收窗口中，等到字节流中所缺少的字节收到后，再按序交付上层的应用进程。
- **TCP** 要求接收方必须有累积确认的功能，这样可以减小传输开销。

超时重传时间的选择

- 重传机制是 **TCP** 中最重要和最复杂的问题之一。
- **TCP** 每发送一个报文段，就对这个报文段设置一次计时器。只要计时器设置的重传时间到但还没有收到确认，就要重传这一报文段。

往返时延的方差很大

- 由于 **TCP** 的下层是一个互联网环境，**IP** 数据报所选择的路由变化很大。因而运输层的往返时间的方差也很大。



加权平均往返时间

- TCP 保留了 RTT 的一个加权平均往返时间 RTT_S（这又称为平滑的往返时间）。
- 第一次测量到 RTT 样本时，RTT_S 值就取为所测量到的 RTT 样本值。以后每测量到一个新的 RTT 样本，就按下式重新计算一次 RTT_S：

新的 $RTT_S = (1 - \alpha) \times$ (旧的 RTT_S)
$+ \alpha \times$ (新的 RTT 样本) 式中，$0 \le \alpha < 1$。

- 若 α 很接近于零，表示 RTT 值更新较慢。若选择 α 接近于 1，则表示 RTT 值更新较快。
- RFC 2988 推荐的 α 值为 1/8，即 0.125。

超时重传时间 RTO (RetransmissionTime-Out)

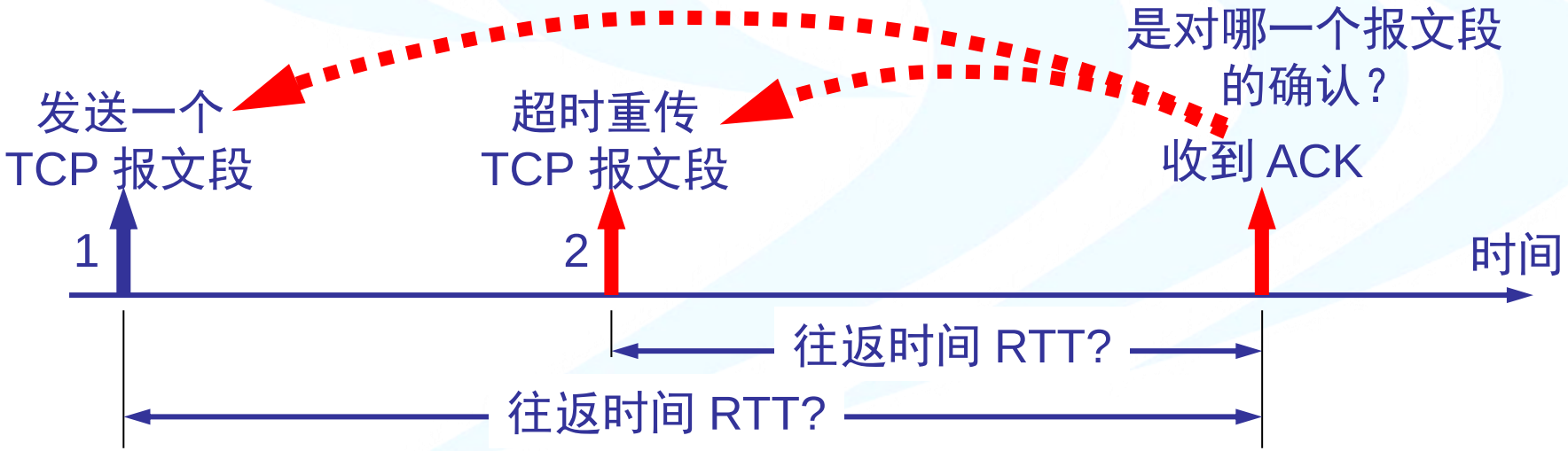
- **RTO** 应略大于上面得出的加权平均往返时间 RTT_S。
- **RFC 2988** 建议使用下式计算 **RTO**：
- $$RTO = RTT_S + 4 \times RTT_D$$
- RTT_D 是 **RTT** 的偏差的加权平均值。
- **RFC 2988** 建议这样计算 RTT_D。第一次测量时，RTT_D 值取为测量到的 **RTT** 样本值的一半。在以后的测量中，则使用下式计算加权平均的 RTT_D：

$$\text{新的 } RTT_D = (1 - \beta) \times (\text{旧的} RTT_D) + \beta \times |RTT_S - \text{新的 RTT 样本}|$$

- β 是个小于 **1** 的系数，其推荐值是 **1/4**，即 **0.25**。

往返时间的测量相当复杂

- **TCP 报文段 1 没有收到确认。重传（即报文段 2）后，收到了确认报文段 ACK。**
- **如何判定此确认报文段是对原来的报文段 1 的确认，还是对重传的报文段 2 的确认？**



Karn 算法

- 在计算平均往返时间 **RTT** 时，只要报文段重传了，就不采用其往返时间样本。
- 这样得出的加权平均平均往返时间 **RTT_S** 和超时重传时间 **RTO** 就较准确。

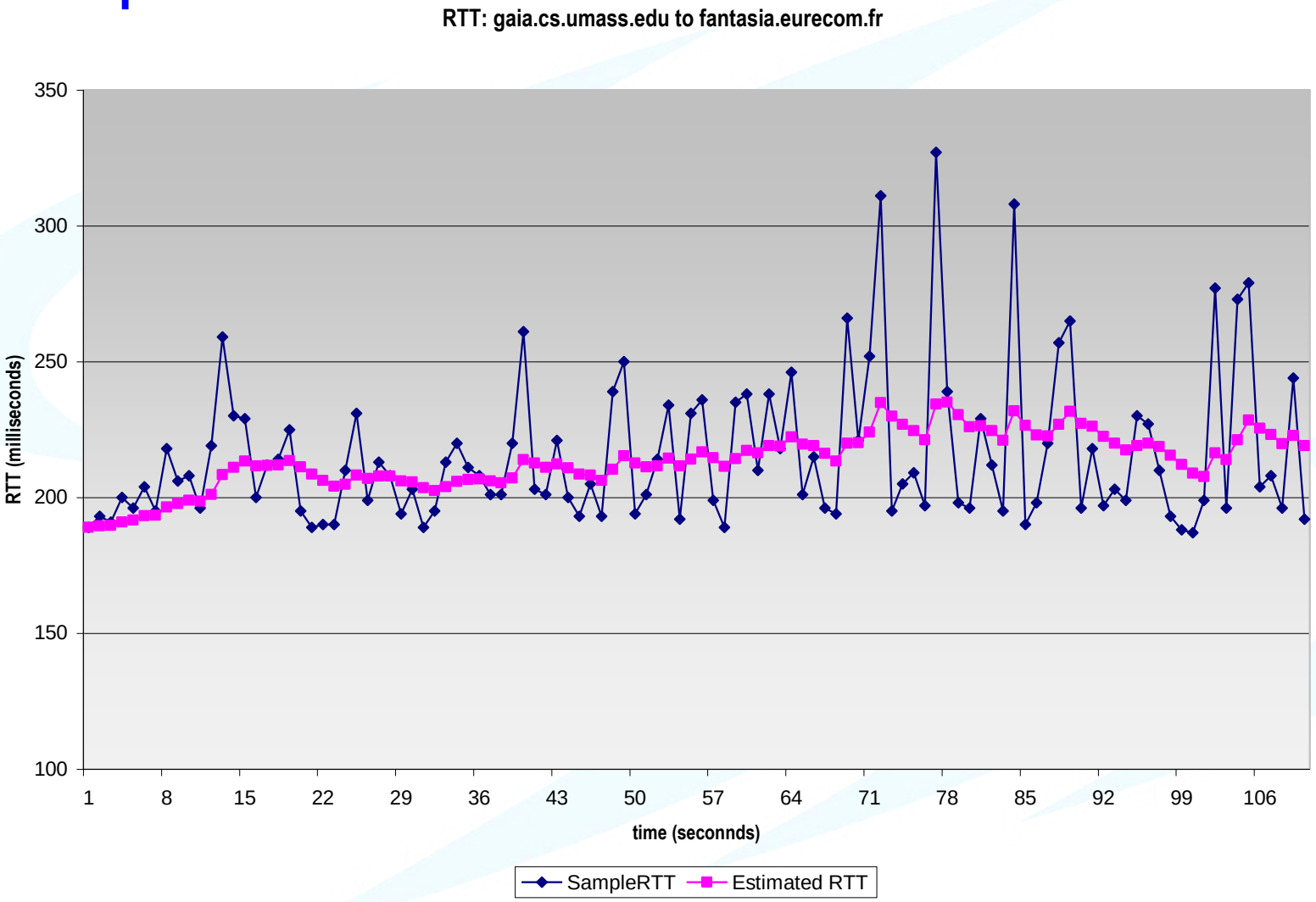
修正的 Karn 算法

- 报文段每重传一次，就把 **RTO** 增大一些：

$$\text{新的 RTO} = \gamma \times (\text{旧的 RTO})$$

- 系数 γ 的典型值是 **2** 。
- 当不再发生报文段的重传时，才根据报文段的往返时延更新平均往返时延 **RTT** 和超时重传时间 **RTO** 的数值。
- 实践证明，这种策略较为合理。

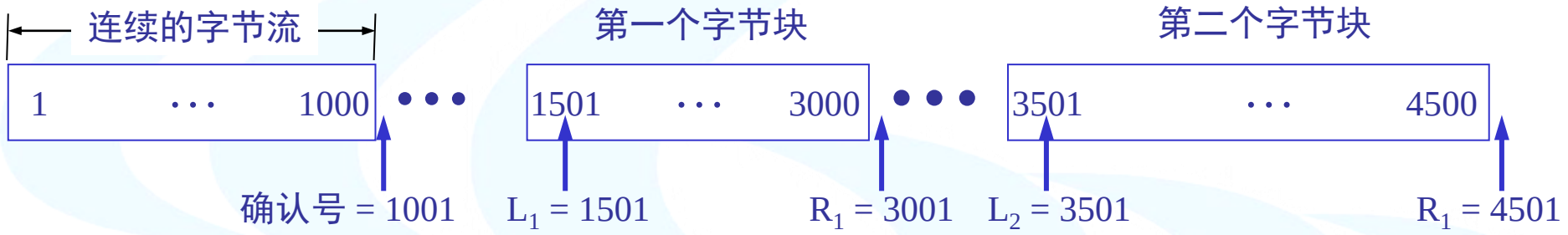
Example RTT estimation:



选择确认 SACK(Selective ACK)

- 接收方收到了和前面的字节流不连续的两个字节块。
- 如果这些字节的序号都在接收窗口之内，那么接收方就先收下这些数据，但要把这些信息准确地告诉发送方，使发送方不要再重复发送这些已收到的数据。

接收到的字节流序号不连续



- 和前后字节不连续的每一个字节块都有两个边界：左边界和右边界。图中用四个指针标记这些边界。
- 第一个字节块的左边界 $L_1 = 1501$，但右边界 $R_1 = 3001$。
- 左边界指出字节块的第一个字节的序号，但右边界减 1 才是字节块中的最后一个序号。
- 第二个字节块的左边界 $L_2 = 3501$，而右边界 $R_2 = 4501$。

RFC 2018 的规定

- 如果要使用选择确认，那么在建立 **TCP** 连接时，就要在 **TCP** 首部的选项中加上“允许 **SACK**”的选项，而双方必须都事先商定好。
- 如果使用选择确认，那么原来首部中的“确认号字段”的用法仍然不变。只是以后在 **TCP** 报文段的首部中都增加了 **SACK** 选项，以便报告收到的不连续的字节块的边界。
- 由于首部选项的长度最多只有 **40** 字节，而指明一个边界就要用掉 **4** 字节，因此在选项中最多只能指明 **4** 个字节块的边界信息。

TCP sender events:

data rcvd from app:

- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (think of timer as for oldest unacked segment)
- expiration interval: `TimeOutInterval`

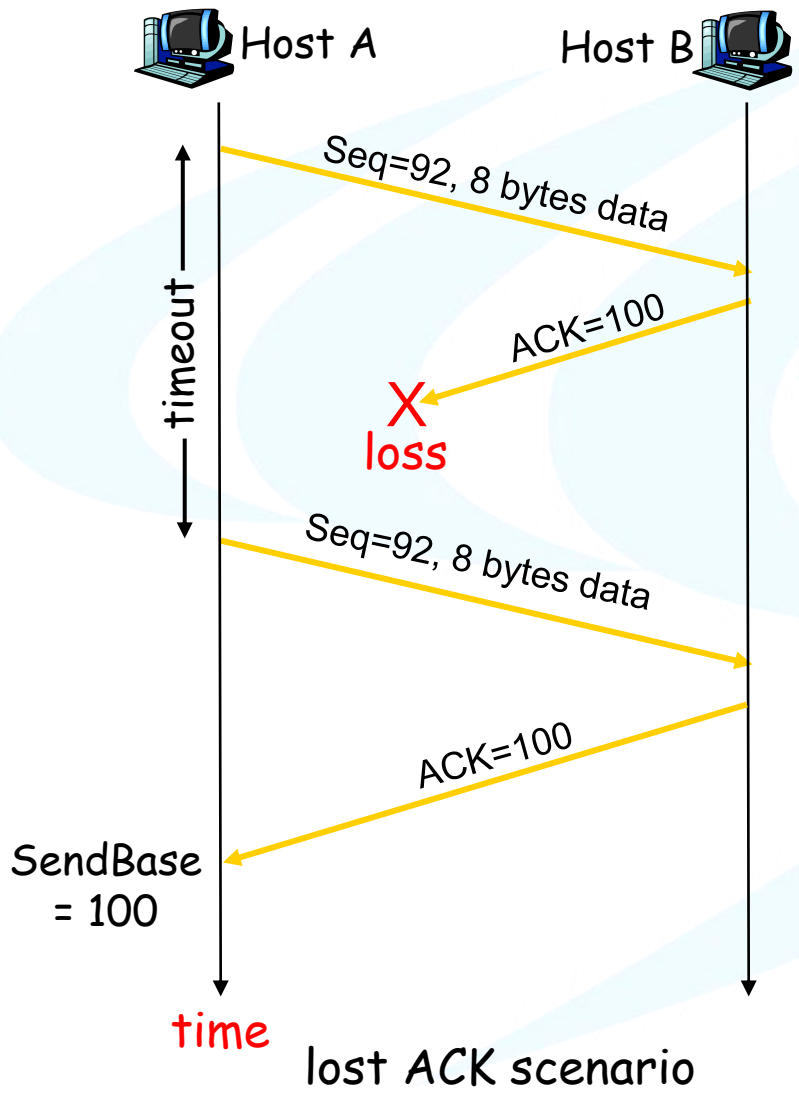
timeout:

- retransmit segment that caused timeout
- restart timer

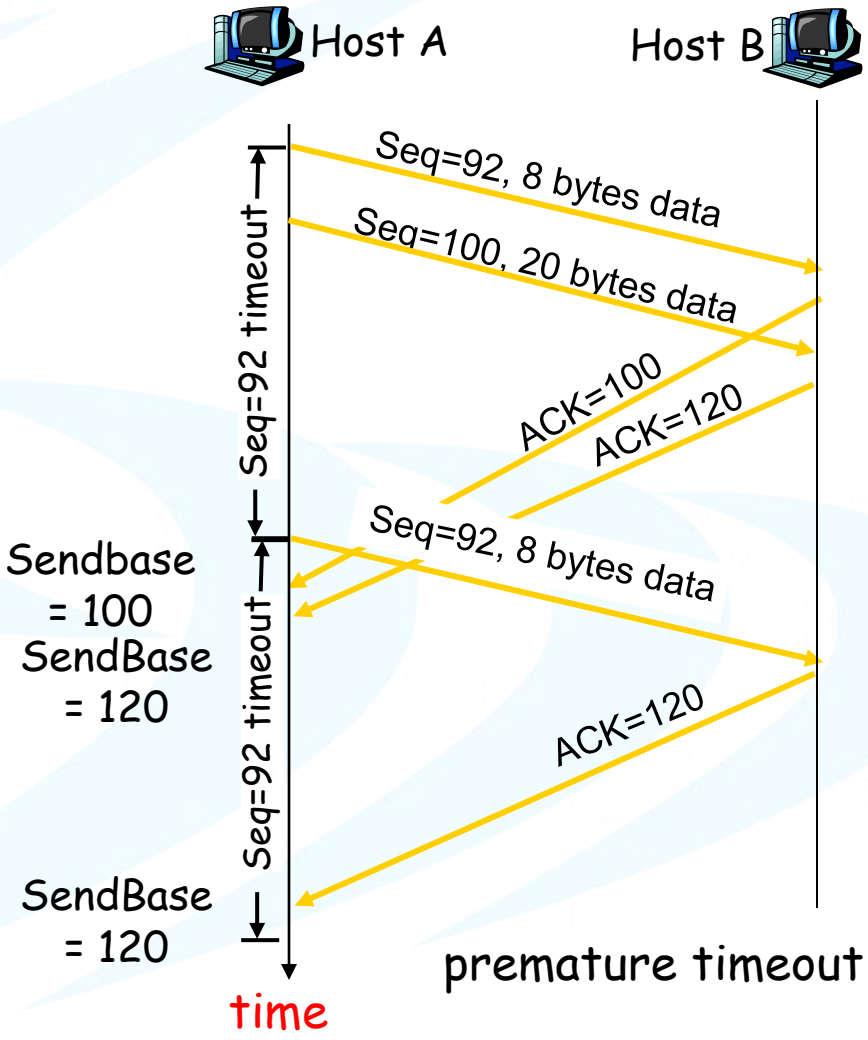
Ack rcvd:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

TCP: retransmission scenarios

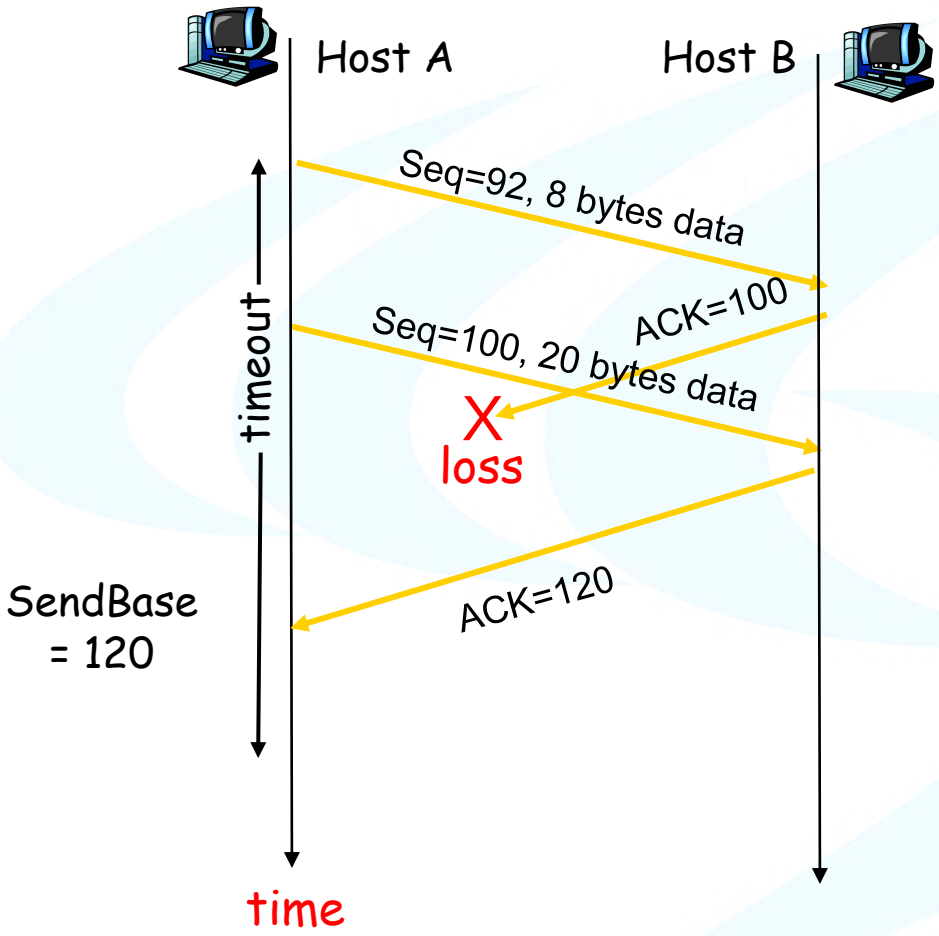


lost ACK scenario



premature timeout

TCP retransmission scenarios (more)



Cumulative ACK scenario

Fast Retransmit

- Time-out period often relatively long:
  - long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

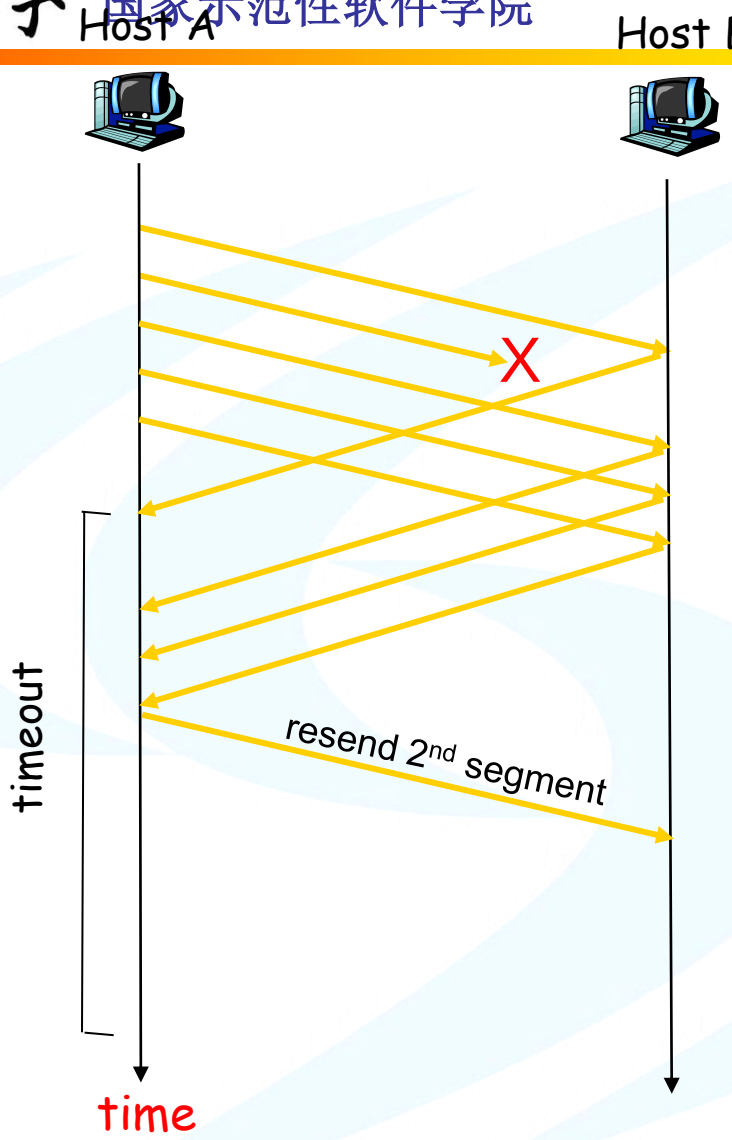


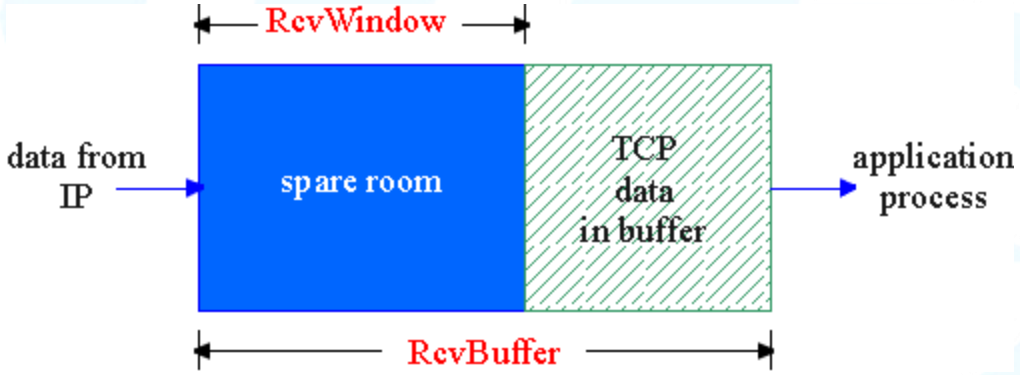
Figure 3.37 Resending a segment after triple duplicate ACK

Chapter 4 outline

- 4.1 Transport-layer services
- 4.2 Multiplexing and demultiplexing
- 4.3 Connectionless transport: UDP
- 4.4 Principles of reliable data transfer
- 4.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 4.6 Principles of congestion control
- 4.7 TCP congestion control

4.5.3 TCP Flow Control

- **receive side of TCP connection has a receive buffer:**



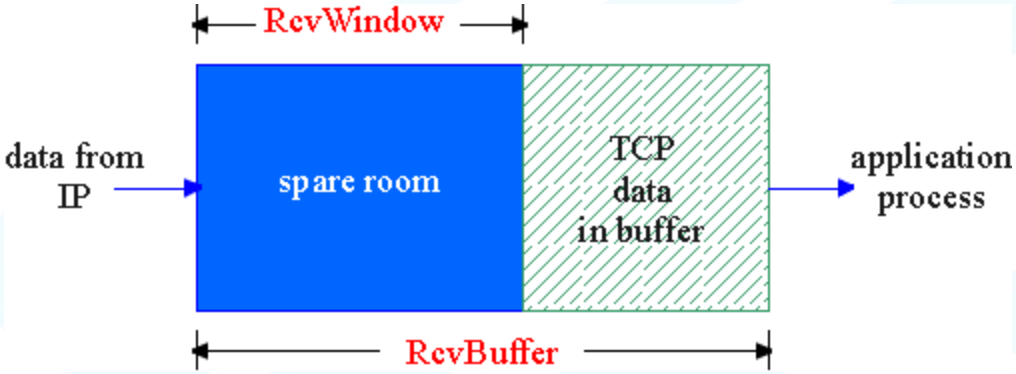
- **app process may be slow at reading from buffer**

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

- **speed-matching service: matching the send rate to the receiving app's drain rate**

TCP Flow control: how it works



(Suppose TCP receiver discards out-of-order segments)

- **spare room in buffer**

= RcvWindow
= RcvBuffer-[LastByteRcvd - LastByteRead]

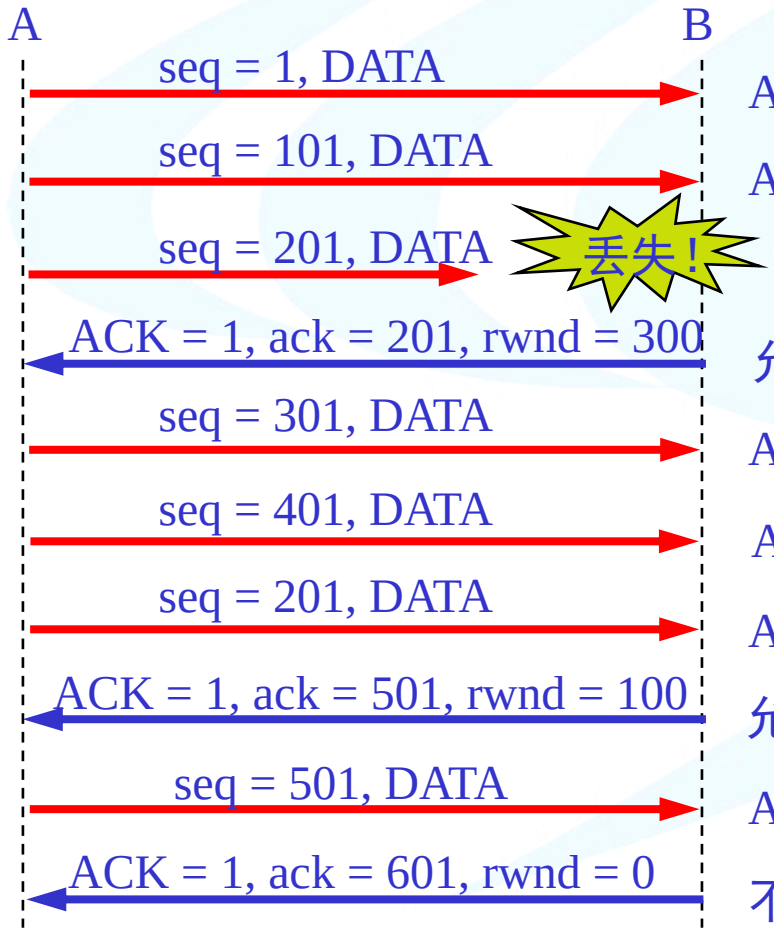
- **Rcvr advertises spare room by including value of** `RcvWindow` **in segments**
- **Sender limits unACKed data to** `RcvWindow`
  - **guarantees receive buffer doesn't overflow**

利用滑动窗口实现流量控制

- 一般说来，我们总是希望数据传输得更快一些。但如果发送方把数据发送得过快，接收方就可能来不及接收，这就会造成数据的丢失。
- 流量控制(flow control)就是让发送方的发送速率不要太快，既要让接收方来得及接收，也不要使网络发生拥塞。
- 利用滑动窗口机制可以很方便地在 TCP 连接上实现流量控制。

流量控制举例

A 向 B 发送数据。在连接建立时，
B 告诉 A：“我的接收窗口 rwnd = 400（字节）”。



A 发送了序号 1 至 100，还能发送 300 字节

A 发送了序号 101 至 200，还能发送 200 字节

允许 A 发送序号 201 至 500 共 300 字节

A 发送了序号 301 至 400，还能再发送 100 字节新数据

A 发送了序号 401 至 500，不能再发送新数据了

A 超时重传旧的数据，但不能发送新的数据

允许 A 发送序号 501 至 600 共 100 字节

A 发送了序号 501 至 600，不能再发送了

不允许 A 再发送（到序号 600 为止的数据都收到了）

持续计时器(persistence timer)。

- **TCP** 为每一个连接设有一个持续计时器。
- 只要 **TCP** 连接的一方收到对方的零窗口通知，就启动持续计时器。
- 若持续计时器设置的时间到期，就发送一个零窗口探测报文段（仅携带 **1** 字节的数据），而对方就在确认这个探测报文段时给出了现在的窗口值。
- 若窗口仍然是零，则收到这个报文段的一方就重新设置持续计时器。
- 若窗口不是零，则死锁的僵局就可以打破了。

必须考虑传输效率

- 可以用不同的机制来控制 **TCP** 报文段的发送时机**:**
- 第一种机制是 **TCP** 维持一个变量，它等于最大报文段长度 **MSS**。只要缓存中存放的数据达到 **MSS** 字节时，就组装成一个 **TCP** 报文段发送出去。
- 第二种机制是由发送方的应用进程指明要求发送报文段，即 **TCP** 支持的推送**(push)**操作。
- 第三种机制是发送方的一个计时器期限到了，这时就把当前已有的缓存数据装入报文段（但长度不能超过 **MSS**）发送出去。

4.6 Principles of congestion control

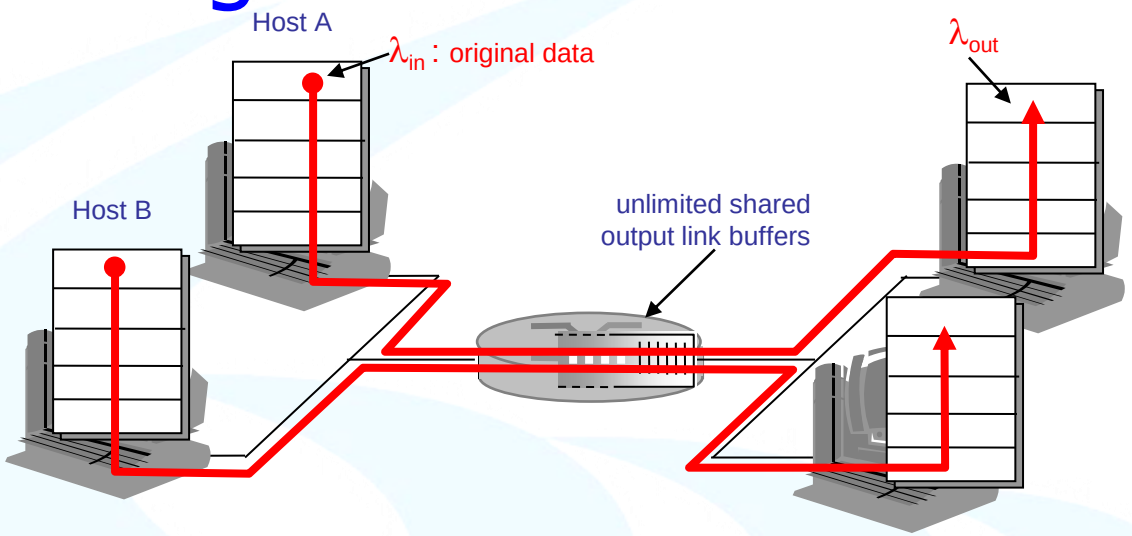
Principles of Congestion Control

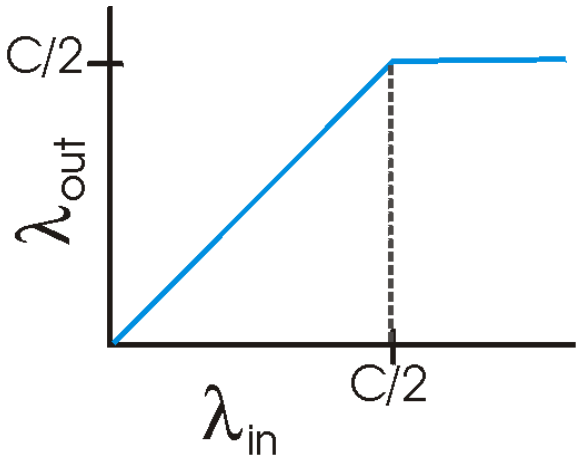
Congestion:

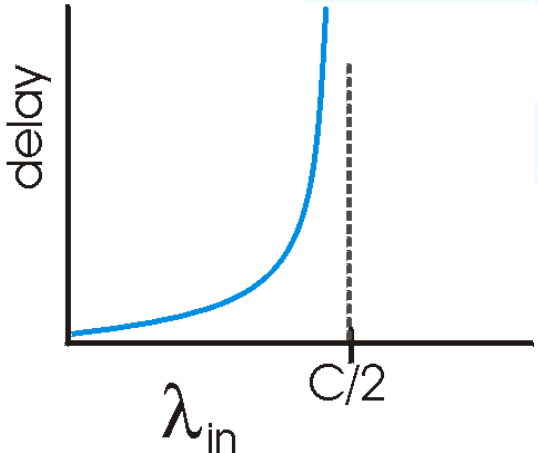
- **informally: "too many sources sending too much data too fast for *network* to handle"**
- **different from flow control!**
- **manifestations:**
  - **lost packets (buffer overflow at routers)**
  - **long delays (queueing in router buffers)**
- **a top-10 problem!**

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission



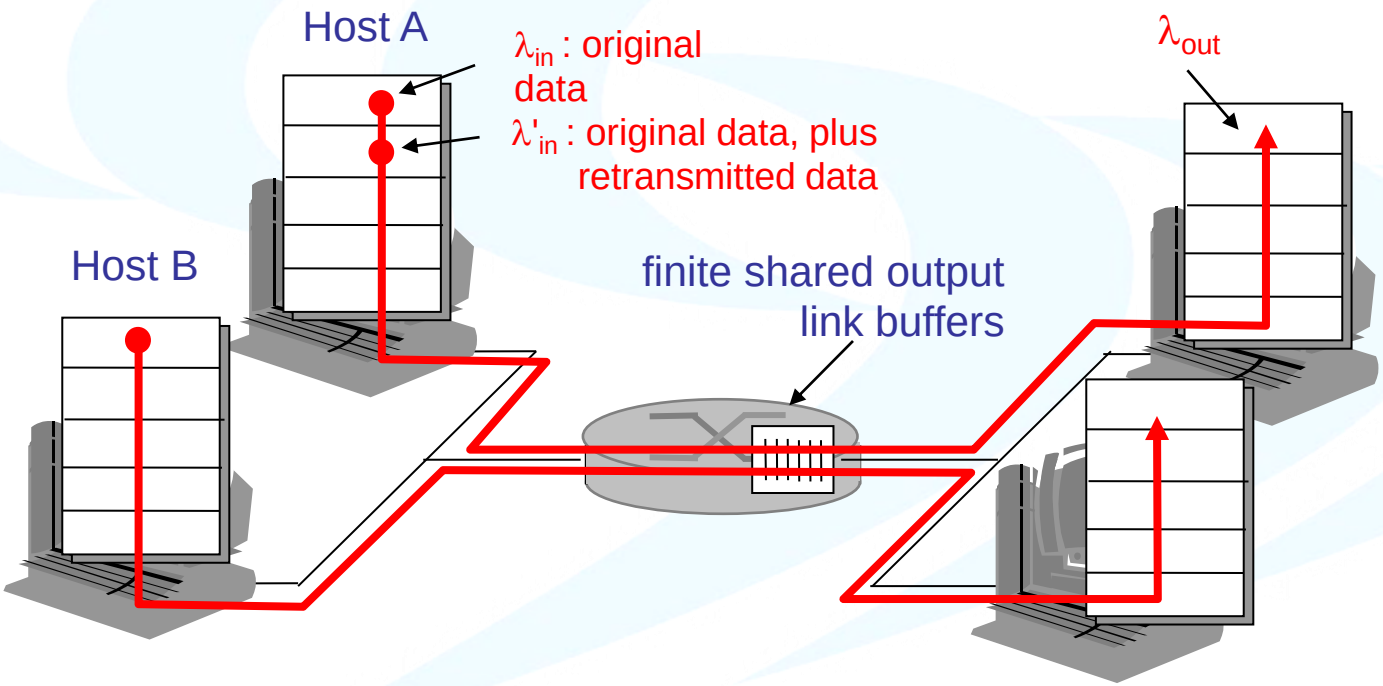




- large delays when congested
- maximum achievable throughput

Causes/costs of congestion: scenario 2

- **one router, *finite* buffers**
- **sender retransmission of lost packet**



拥塞控制的一般原理

- 在某段时间，若对网络中某资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏——产生拥塞(congestion)。
- 出现资源拥塞的条件：

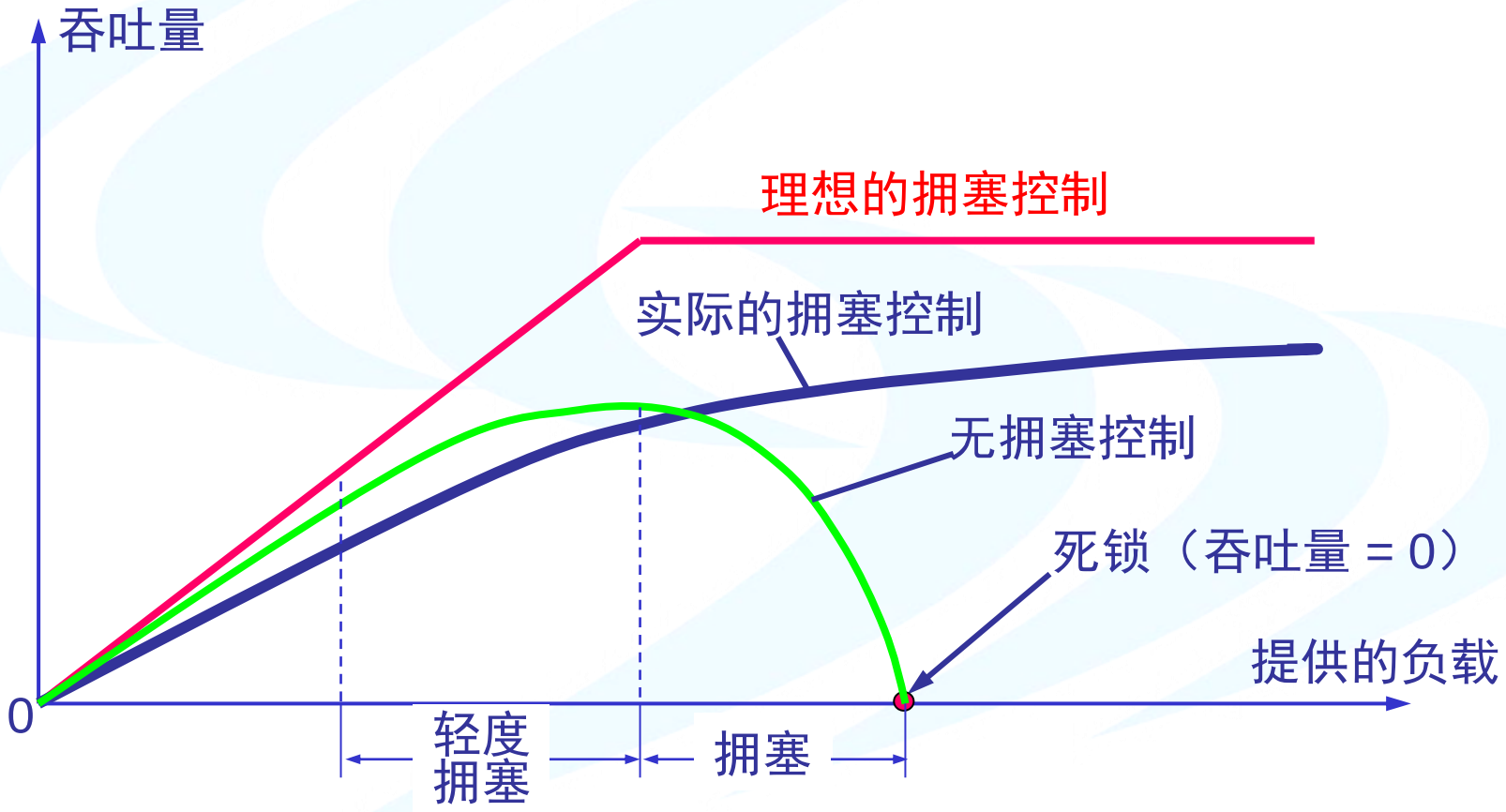
  对资源需求的总和 > 可用资源

- 若网络中有许多资源同时产生拥塞，网络的性能就要明显变坏，整个网络的吞吐量将随输入负荷的增大而下降。

拥塞控制与流量控制的关系

- 拥塞控制所要做的都有一个前提，就是网络能够承受现有的网络负荷。
- 拥塞控制是一个全局性的过程，涉及到所有的主机、所有的路由器，以及与降低网络传输性能有关的所有因素。
- 流量控制往往指在给定的发送端和接收端之间的点对点通信量的控制。
- 流量控制所要做的就是抑制发送端发送数据的速率，以便使接收端来得及接收。

拥塞控制所起的作用



拥塞控制的一般原理

- 拥塞控制是很难设计的，因为它是一个动态的（而不是静态的）问题。
- 当前网络正朝着高速化的方向发展，这很容易出现缓存不够大而造成分组的丢失。但分组的丢失是网络发生拥塞的征兆而不是原因。
- 在许多情况下，甚至正是拥塞控制本身成为引起网络性能恶化甚至发生死锁的原因。这点应特别引起重视。

开环控制和闭环控制

- 开环控制方法就是在设计网络时事先将有关发生拥塞的因素考虑周到，力求网络在工作时不产生拥塞。
- 闭环控制是基于反馈环路的概念。属于闭环控制的有以下几种措施：
  - 监测网络系统以便检测到拥塞在何时、何处发生。
  - 将拥塞发生的信息传送到可采取行动的地方。
  - 调整网络系统的运行以解决出现的问题。

TCP Congestion Control: details

- **sender limits transmission:**
  `LastByteSent-LastByteAcked`
  `≤ CongWin`
- **Roughly,**

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- `CongWin` **is dynamic, function of perceived network congestion**

How does sender perceive congestion?

- **loss event = timeout *or* 3 duplicate acks**
- **TCP sender reduces rate (`CongWin`) after loss event**

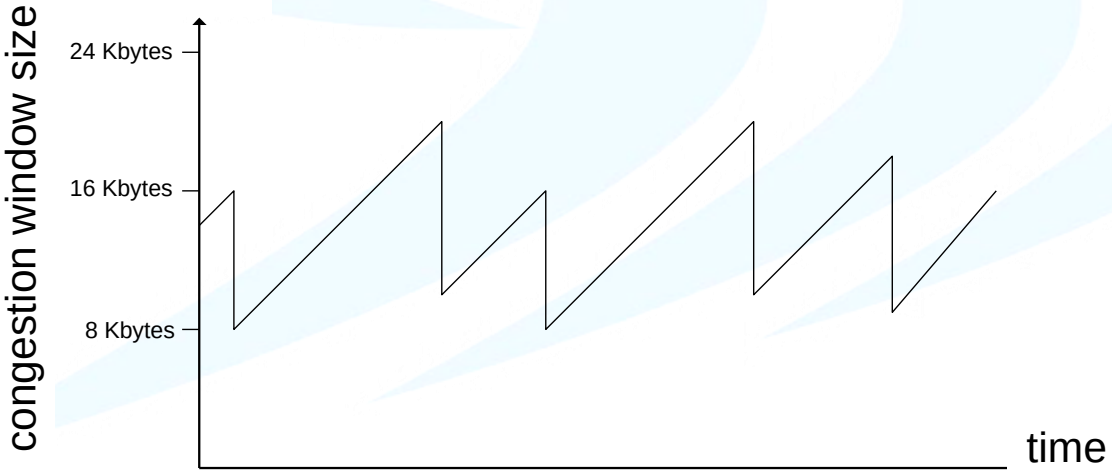
three mechanisms:

- **AIMD**
- **slow start**
- **conservative after timeout events**

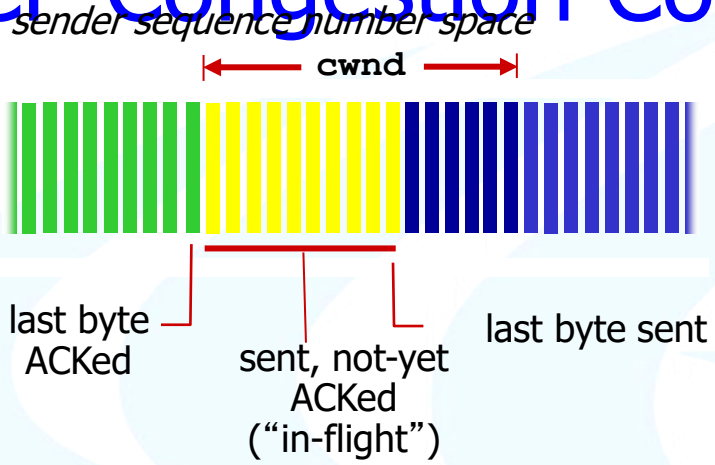
TCP congestion control: additive increase, multiplicative decrease

- *Approach:* increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **CongWin** by 1 MSS every RTT until loss detected
  - *multiplicative decrease:* cut **CongWin** in half after loss

Saw tooth behavior: probing for bandwidth



TCP Congestion Control: details



- sender limits transmission:

$$\texttt{LastByteSent} - \texttt{LastByteAcked} \leq \texttt{cwnd}$$

- cwnd is dynamic, function of perceived network congestion

TCP sending rate:

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

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

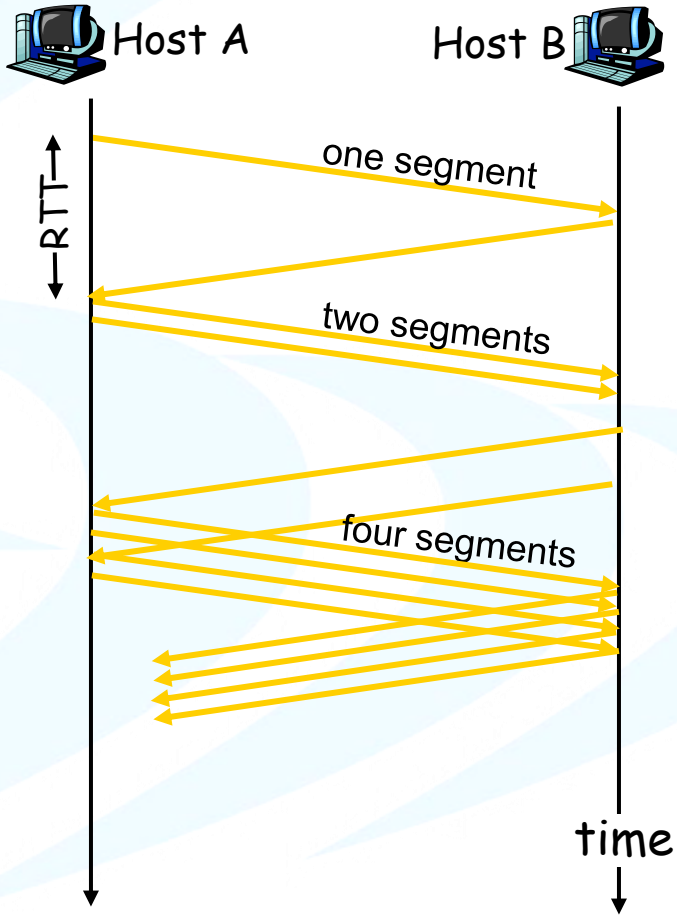
TCP Slow Start

- **When connection begins, `CongWin` = 1 MSS**
  - **Example: MSS = 500 bytes & RTT = 200 msec**
  - **initial rate = 20 kbps**
- **available bandwidth may be >> MSS/RTT**
  - **desirable to quickly ramp up to respectable rate**

r When connection begins, increase rate exponentially fast until first loss event

TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  - double `CongWin` every RTT
  - done by incrementing `CongWin` for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



几种拥塞控制方法

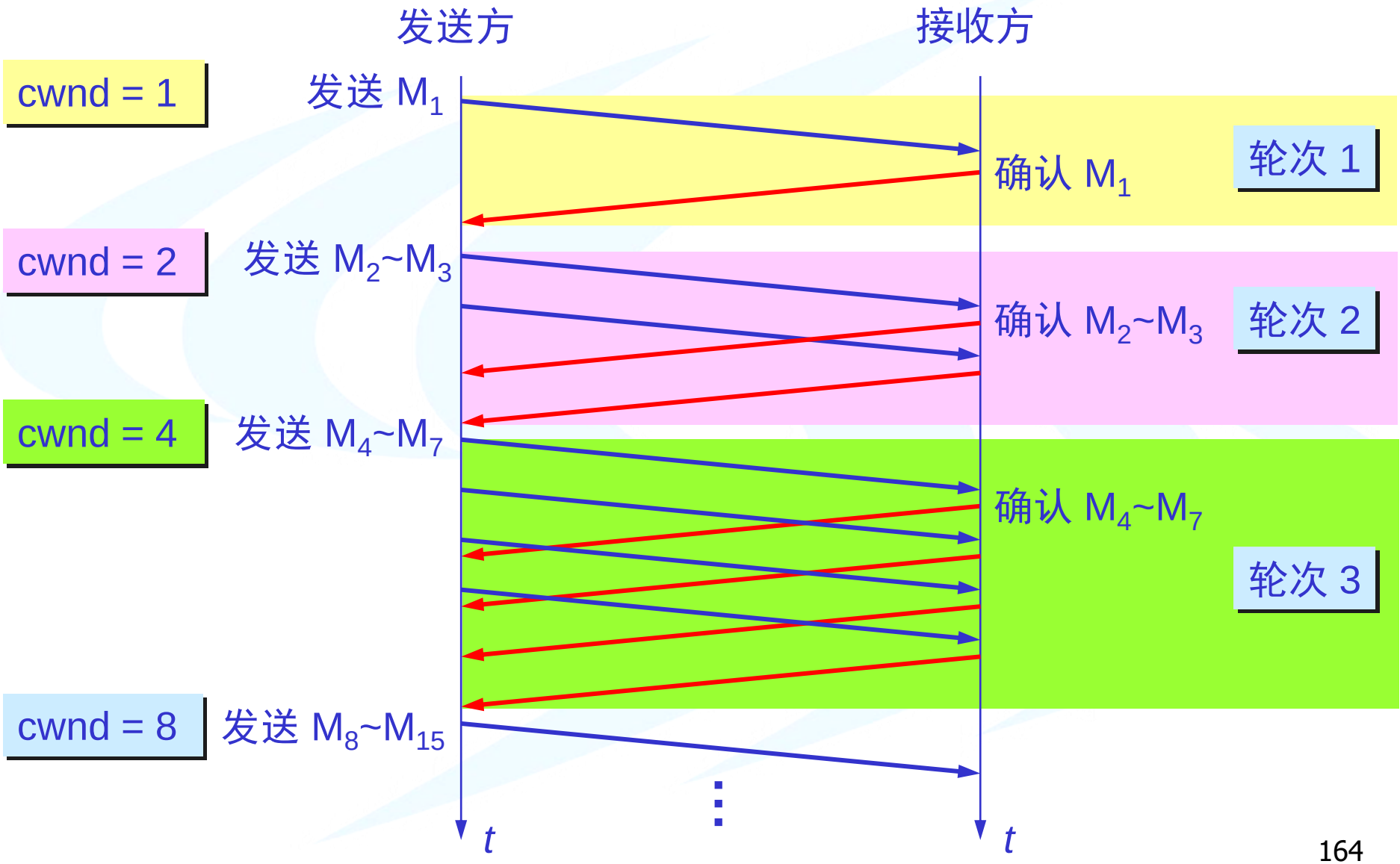
1. 慢开始和拥塞避免

- 发送方维持一个叫做拥塞窗口 **cwnd (congestion window)**的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方让自己的发送窗口等于拥塞窗口。如再考虑到接收方的接收能力，则发送窗口还可能小于拥塞窗口。
- 发送方控制拥塞窗口的原则是：只要网络没有出现拥塞，拥塞窗口就再增大一些，以便把更多的分组发送出去。但只要网络出现拥塞，拥塞窗口就减小一些，以减少注入到网络中的分组数。

慢开始算法的原理

- 在主机刚刚开始发送报文段时可先设置拥塞窗口 **cwnd = 1**，即设置为一个最大报文段 **MSS** 的数值。
- 在每收到一个对新的报文段的确认后，将拥塞窗口加 **1**，即增加一个 **MSS** 的数值。
- 用这样的方法逐步增大发送端的拥塞窗口 **cwnd**，可以使分组注入到网络的速率更加合理。

发送方每收到一个对新报文段的确认（重传的不算在内）就使 cwnd 加 1。



传输轮次(transmission round)

- 使用慢开始算法后，每经过一个传输轮次，拥塞窗口 **cwnd** 就加倍。
- 一个传输轮次所经历的时间其实就是往返时间 **RTT**。
- “传输轮次”更加强调：把拥塞窗口 **cwnd** 所允许发送的报文段都连续发送出去，并收到了对已发送的最后一个字节的确认。
- 例如，拥塞窗口 **cwnd = 4**，这时的往返时间 **RTT** 就是发送方连续发送 **4** 个报文段，并收到这 **4** 个报文段的确认，总共经历的时间。

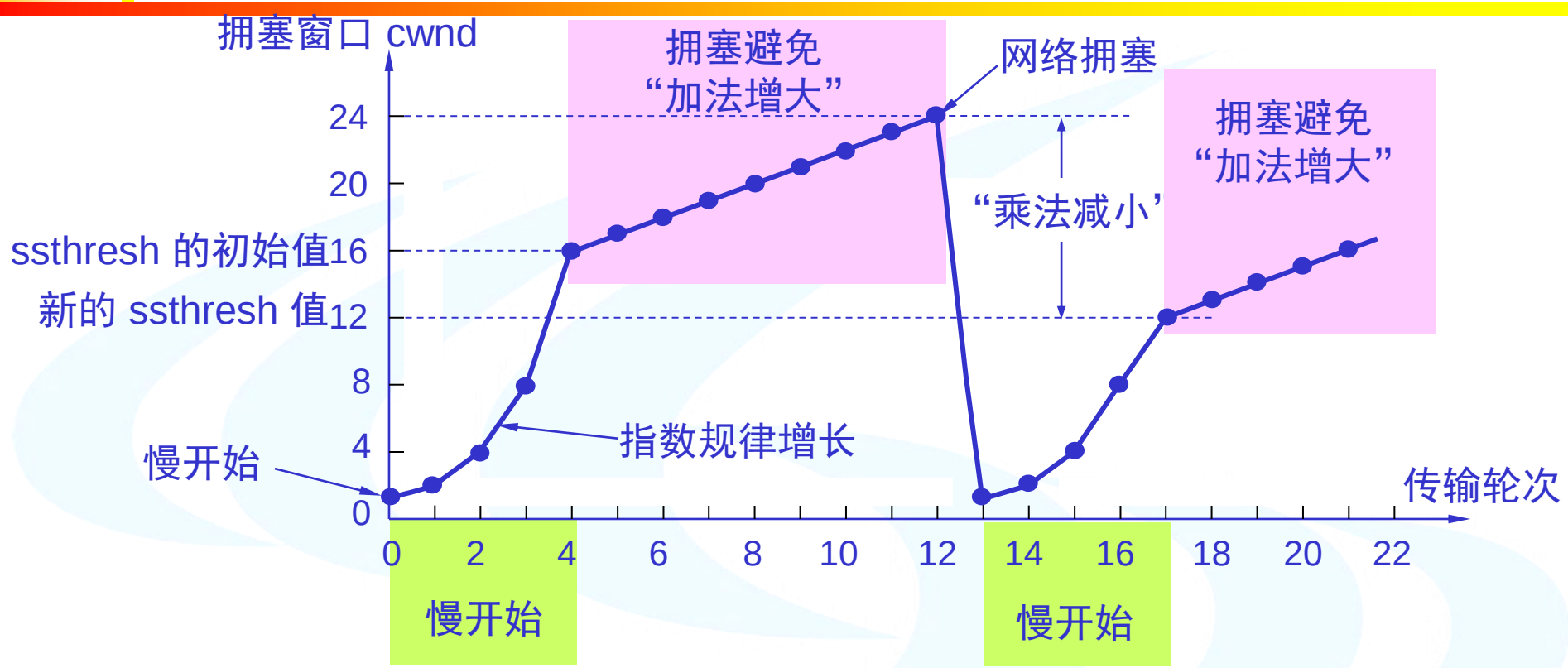
设置慢开始门限状态变量ssthresh

- 慢开始门限 **ssthresh** 的用法如下：
- 当 **cwnd < ssthresh** 时，使用慢开始算法。
- 当 **cwnd > ssthresh** 时，停止使用慢开始算法而改用拥塞避免算法。
- 当 **cwnd = ssthresh** 时，既可使用慢开始算法，也可使用拥塞避免算法。
- 拥塞避免算法的思路是让拥塞窗口 **cwnd** 缓慢地增大，即每经过一个往返时间 **RTT** 就把发送方的拥塞窗口 **cwnd** 加 **1**，而不是加倍，使拥塞窗口 **cwnd** 按线性规律缓慢增长。

当网络出现拥塞时

- 无论在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞（其根据就是没有按时收到确认），就要把慢开始门限 **ssthresh** 设置为出现拥塞时的发送方窗口值的一半（但不能小于**2**）。
- 然后把拥塞窗口 **cwnd** 重新设置为 **1**，执行慢开始算法。
- 这样做的目的就是要迅速减少主机发送到网络中的分组数，使得发生拥塞的路由器有足够时间把队列中积压的分组处理完毕。

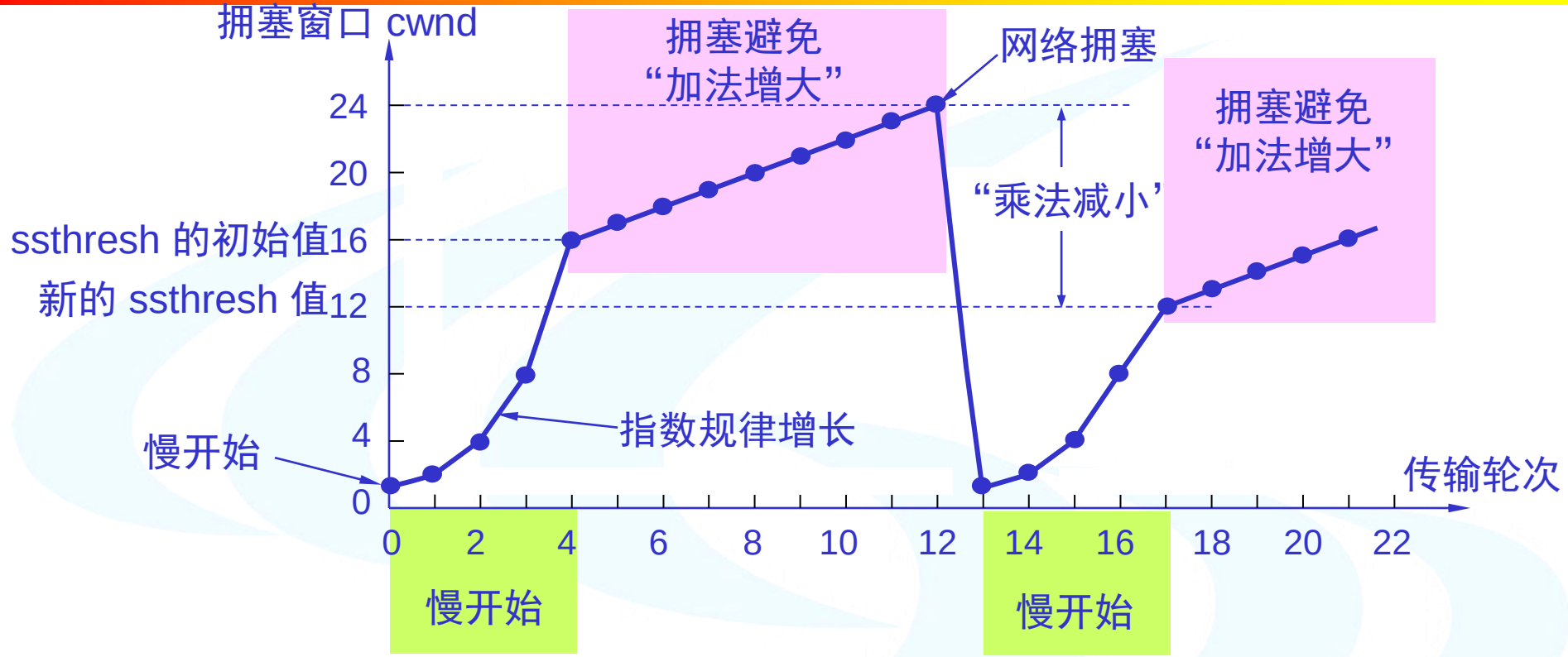
慢开始和拥塞避免算法的实现举例



当 TCP 连接进行初始化时，将拥塞窗口置为 1。图中的窗口单位不使用字节而使用报文段。

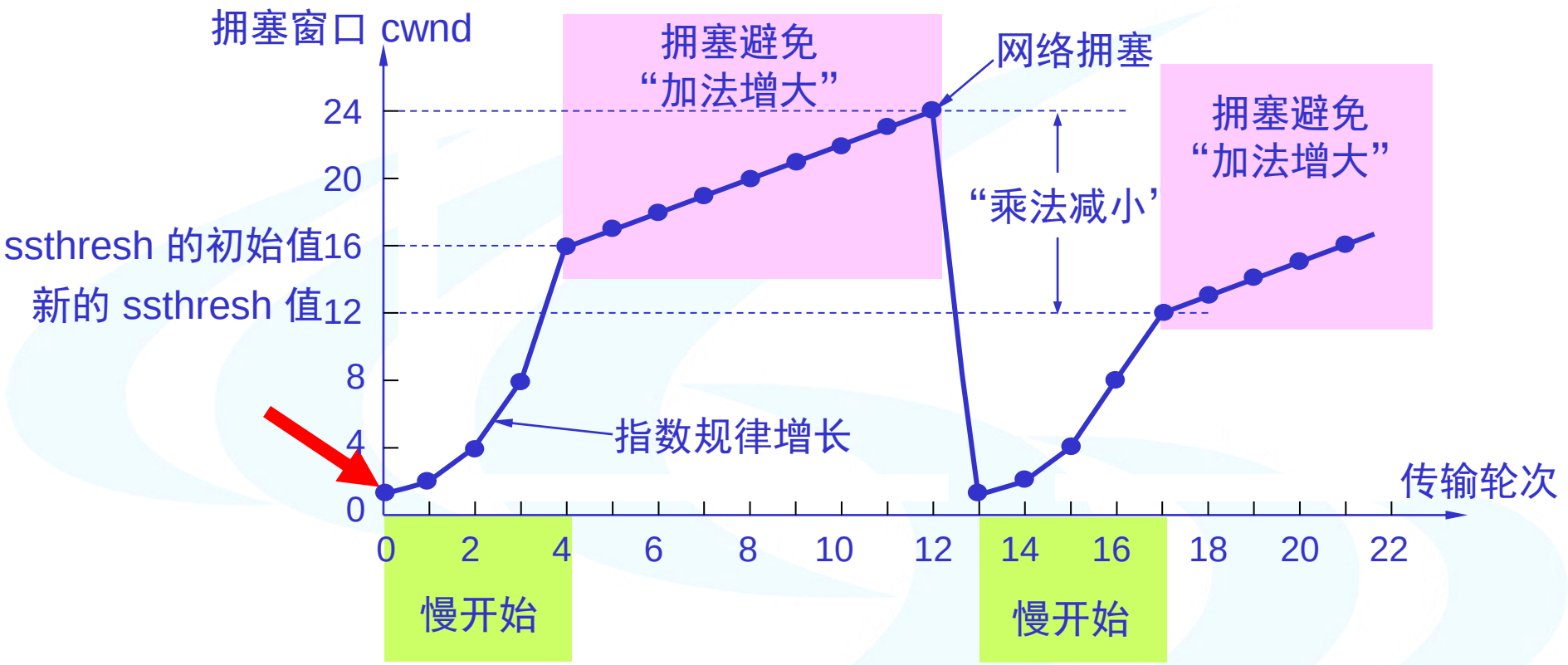
慢开始门限的初始值设置为 16 个报文段，即 ssthresh = 16。

慢开始和拥塞避免算法的实现举例



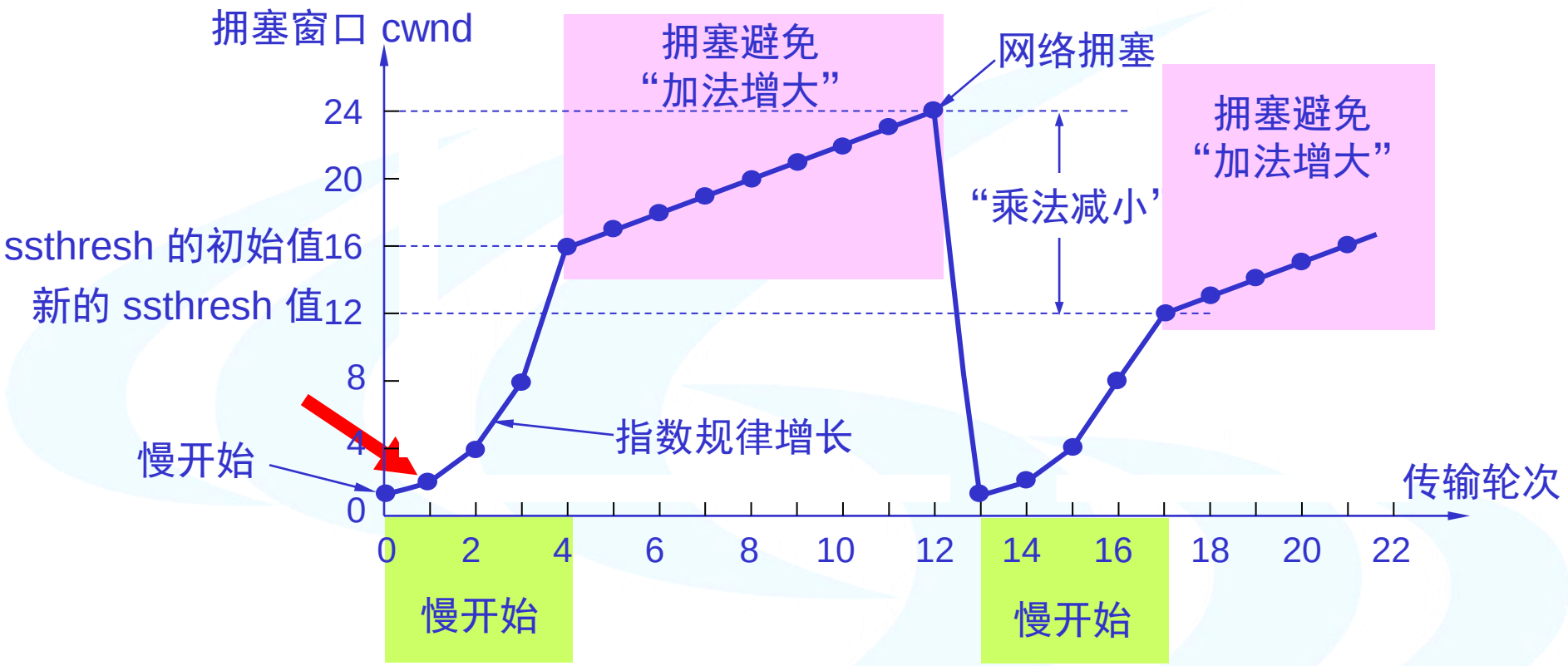
发送端的发送窗口不能超过拥塞窗口 cwnd 和接收端窗口 rwnd 中的最小值。我们假定接收端窗口足够大，因此现在发送窗口的数值等于拥塞窗口的数值。

慢开始和拥塞避免算法的实现举例



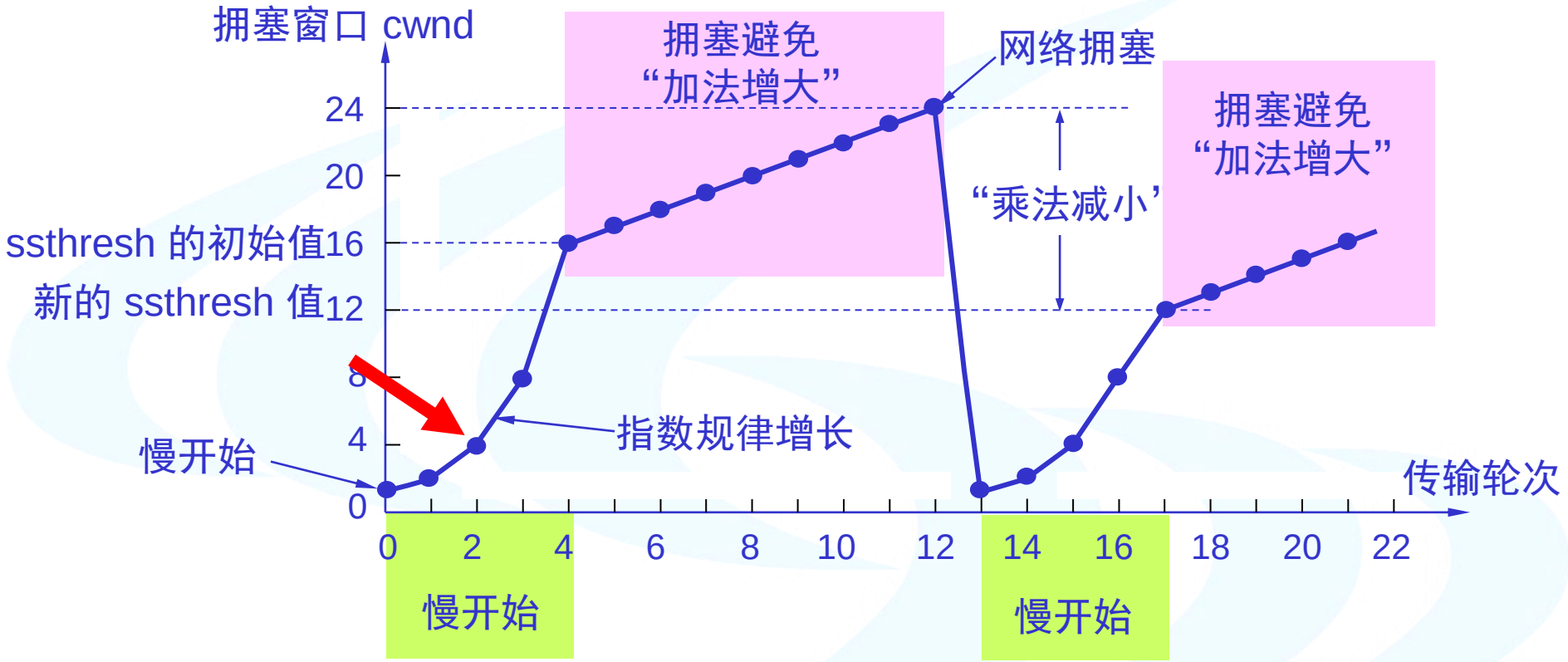
在执行慢开始算法时，拥塞窗口 cwnd 的初始值为 1，发送第一个报文段 M_0。

慢开始和拥塞避免算法的实现举例



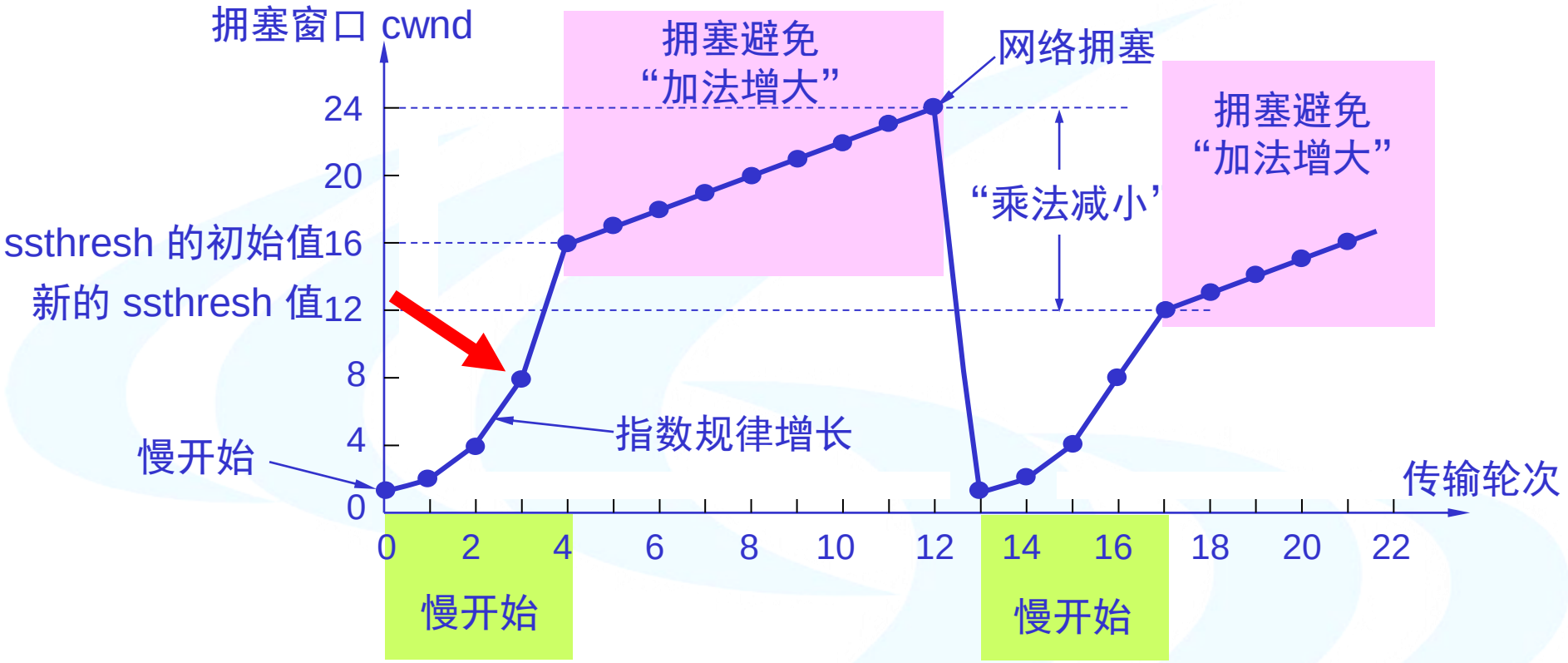
发送端每收到一个确认，就把 cwnd 加 1。于是发送端可以接着发送 M_1 和 M_2 两个报文段。

慢开始和拥塞避免算法的实现举例



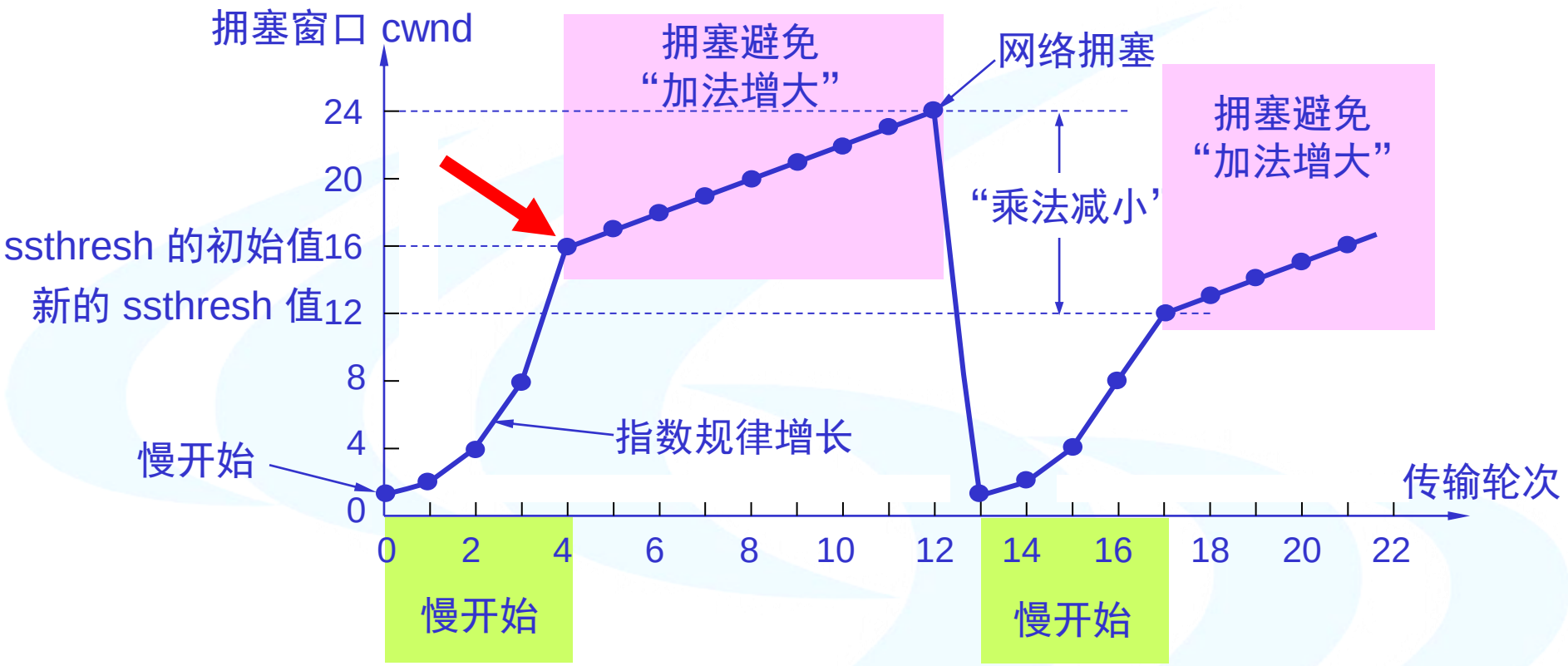
接收端共发回两个确认。发送端每收到一个对新报文段的确认，就把发送端的 cwnd 加 1。现在 cwnd 从 2 增大到 4，并可接着发送后面的 4 个报文段。

慢开始和拥塞避免算法的实现举例



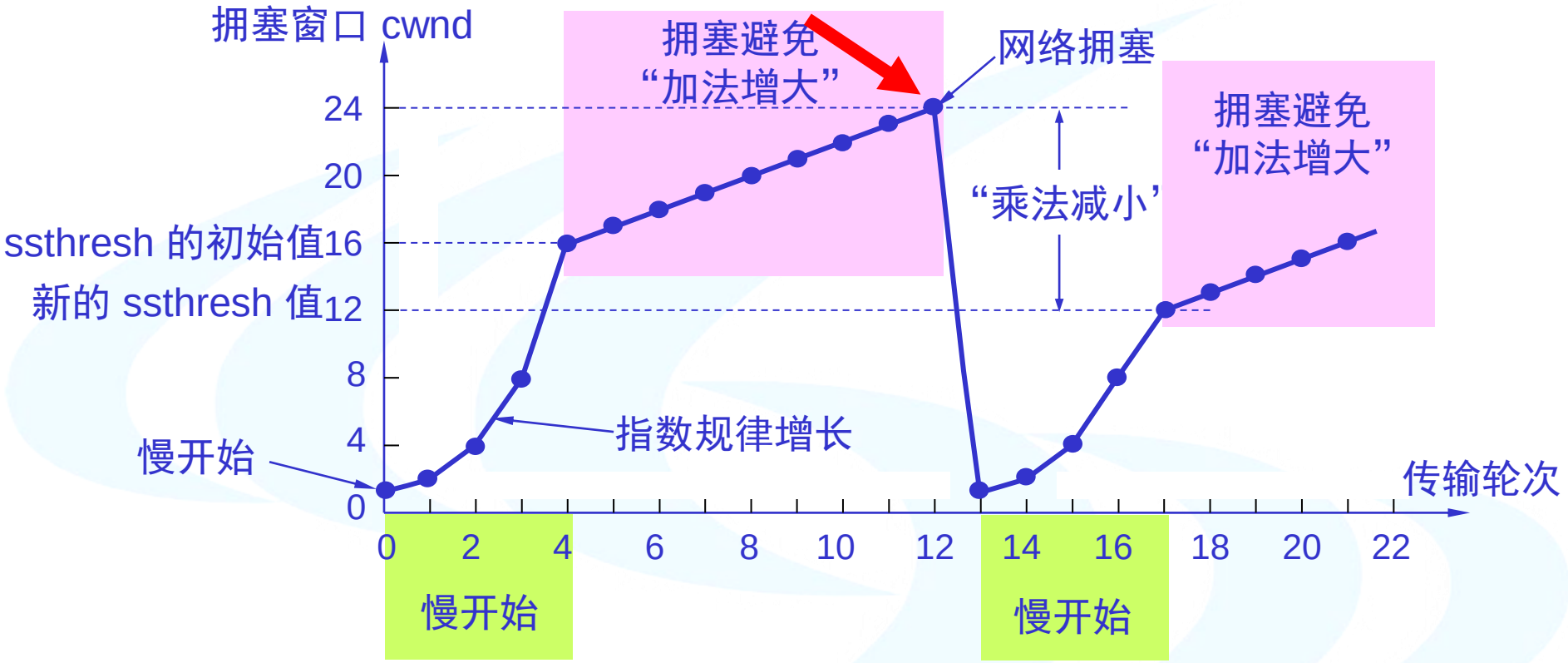
发送端每收到一个对新报文段的确认，就把发送端的拥塞窗口加 1，因此拥塞窗口 cwnd 随着传输轮次按指数规律增长。

慢开始和拥塞避免算法的实现举例



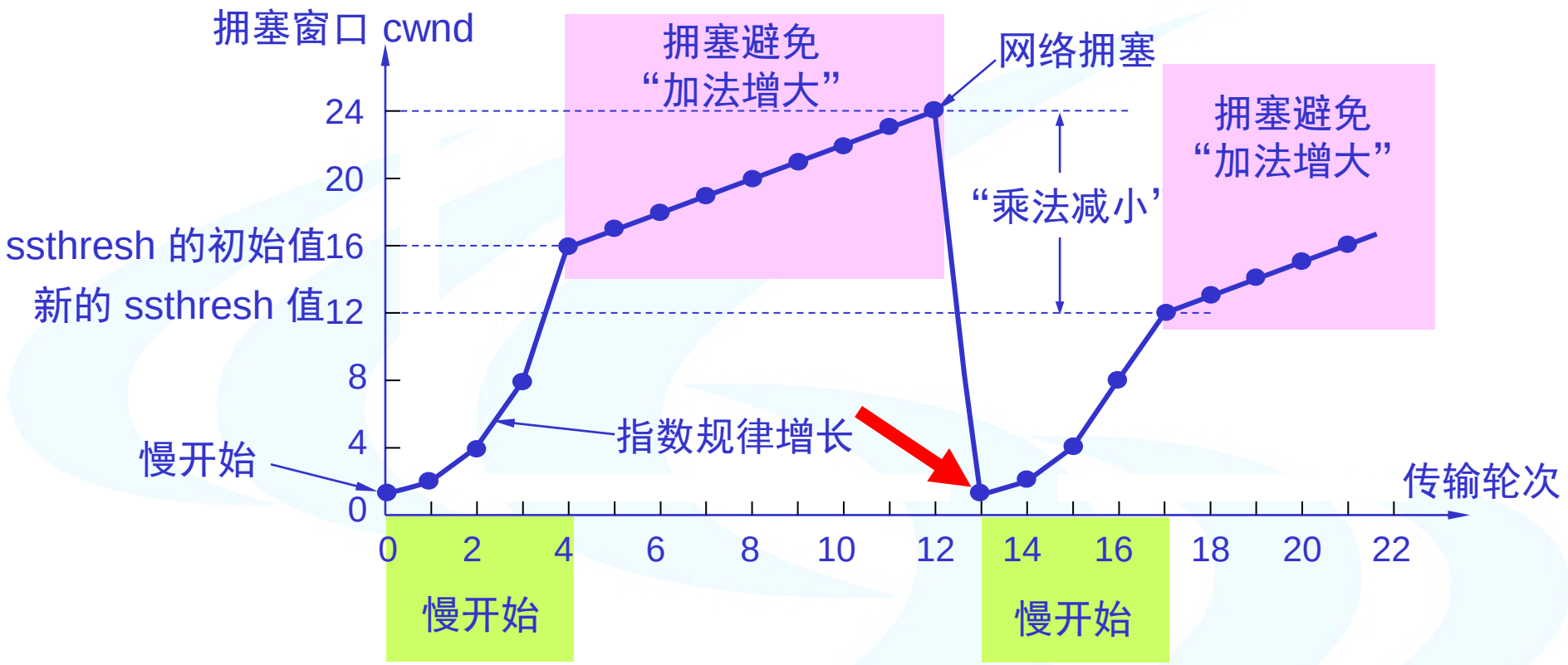
当拥塞窗口 cwnd 增长到慢开始门限值 ssthresh 时（即当 cwnd = 16 时），就改为执行拥塞避免算法，拥塞窗口按线性规律增长。

慢开始和拥塞避免算法的实现举例



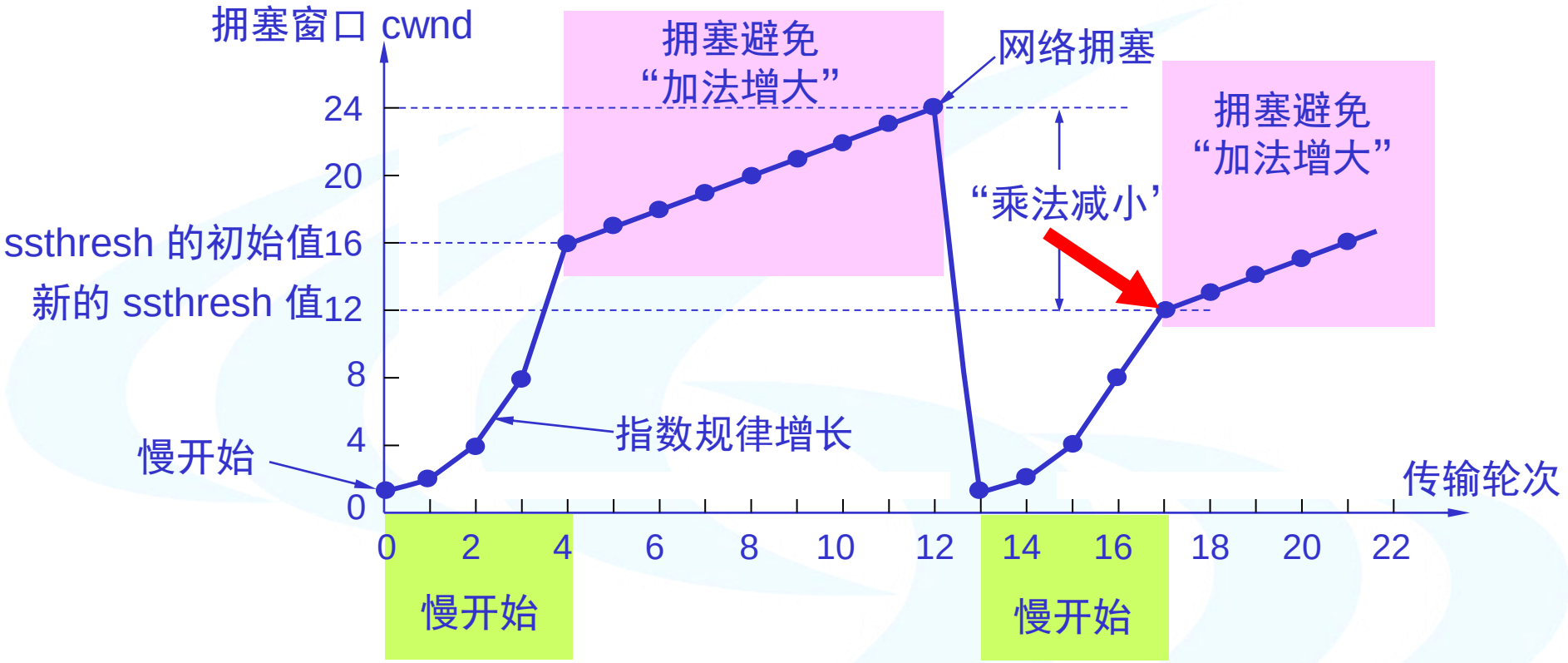
假定拥塞窗口的数值增长到 24 时，网络出现超时，表明网络拥塞了。

慢开始和拥塞避免算法的实现举例



更新后的 ssthresh 值变为 12（即发送窗口数值 24 的一半），拥塞窗口再重新设置为 1，并执行慢开始算法。

慢开始和拥塞避免算法的实现举例



当 cwnd = 12 时改为执行拥塞避免算法，拥塞窗口按按线性规律增长，每经过一个往返时延就增加一个 MSS 的大小。

乘法减小(multiplicative decrease)

- “乘法减小“是指不论在慢开始阶段还是拥塞避免阶段，只要出现一次超时（即出现一次网络拥塞），就把慢开始门限值 **ssthresh** 设置为当前的拥塞窗口值乘以 **0.5**。
- 当网络频繁出现拥塞时，**ssthresh** 值就下降得很快，以大大减少注入到网络中的分组数。

加法增大(additive increase)

- “加法增大”是指执行拥塞避免算法后，在收到对所有报文段的确认后（即经过一个往返时间），就把拥塞窗口 **cwnd**增加一个 **MSS** 大小，使拥塞窗口缓慢增大，以防止网络过早出现拥塞。

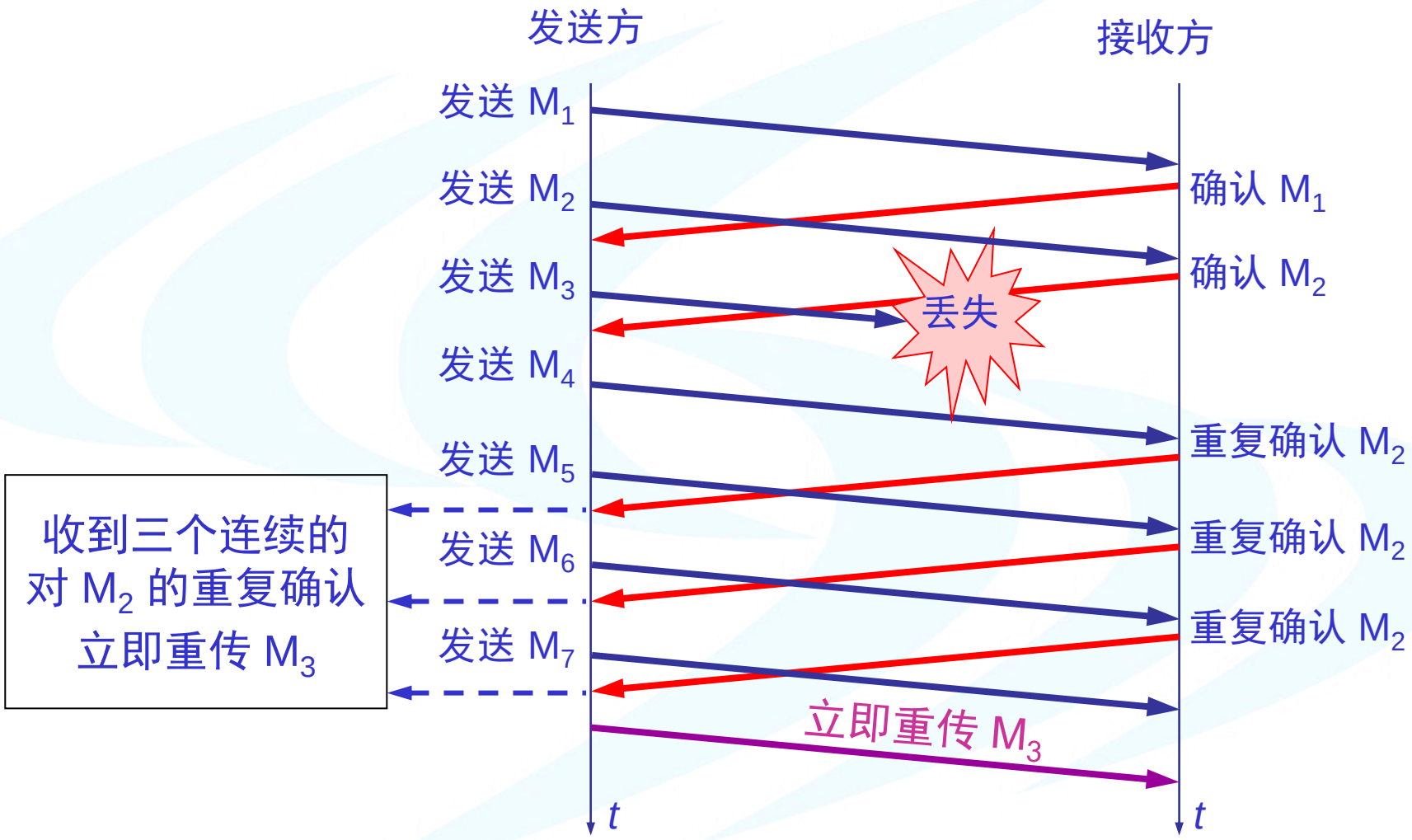
必须强调指出

- “拥塞避免”并非指完全能够避免了拥塞。利用以上的措施要完全避免网络拥塞还是不可能的。
- “拥塞避免”是说在拥塞避免阶段把拥塞窗口控制为按线性规律增长，使网络比较不容易出现拥塞。

2. 快重传和快恢复

- 快重传算法首先要求接收方每收到一个失序的报文段后就立即发出重复确认。这样做可以让发送方及早知道有报文段没有到达接收方。
- 发送方只要一连收到三个重复确认就应当立即重传对方尚未收到的报文段。
- 不难看出，快重传并非取消重传计时器，而是在某些情况下可更早地重传丢失的报文段。

快重传举例

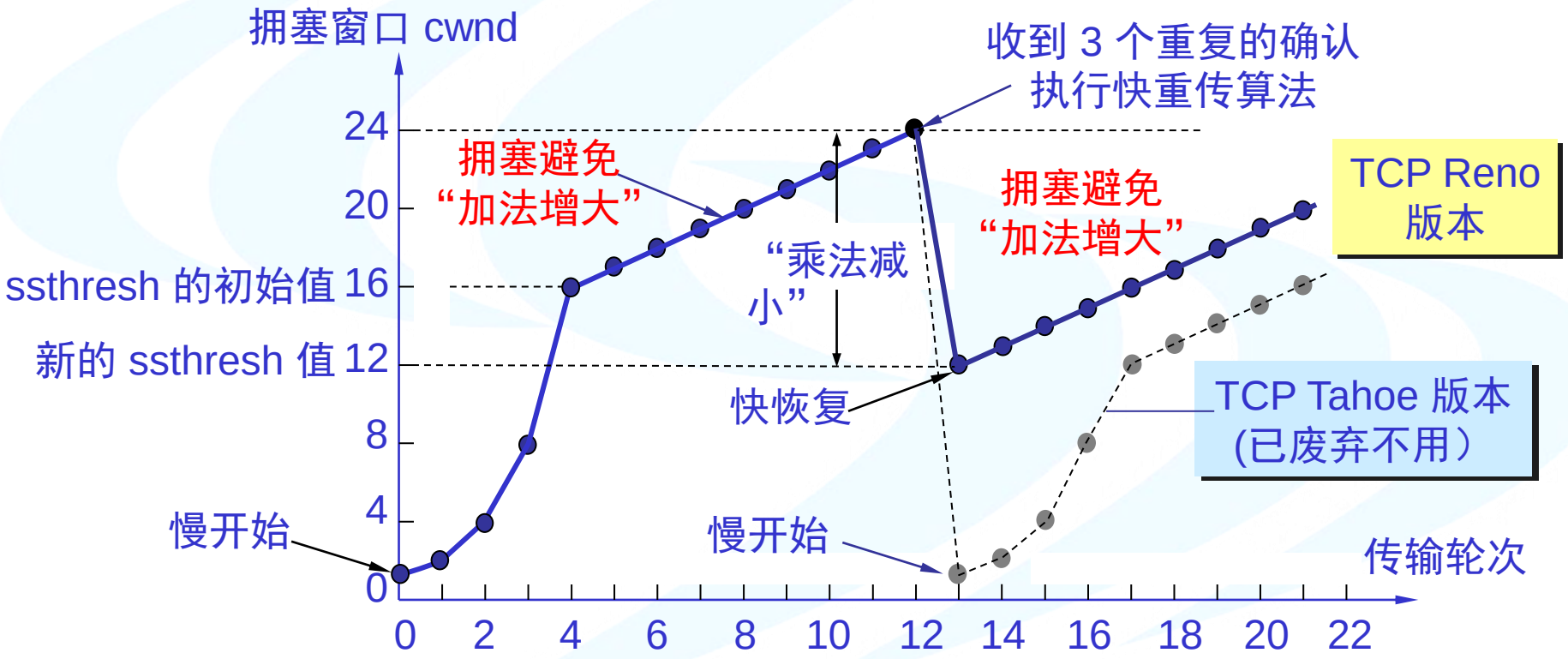

发送方
接收方
发送 M_1
发送 M_2
发送 M_3
发送 M_4
发送 M_5
发送 M_6
发送 M_7
确认 M_1
确认 M_2
丢失
重复确认 M_2
重复确认 M_2
重复确认 M_2
立即重传 M_3
收到三个连续的
对 M_2 的重复确认
立即重传 M_3
t
t

快恢复算法

(1) 当发送端收到连续三个重复的确认时，就执行“乘法减小”算法，把慢开始门限 **ssthresh** 减半。但接下去不执行慢开始算法。

(2)由于发送方现在认为网络很可能没有发生拥塞，因此现在不执行慢开始算法，即拥塞窗口 **cwnd** 现在不设置为 **1**，而是设置为慢开始门限 **ssthresh** 减半后的数值，然后开始执行拥塞避免算法（“加法增大”），使拥塞窗口缓慢地线性增大。

从连续收到三个重复的确认转入拥塞避免



Refinement: inferring loss

- **After 3 dup ACKs:**
  - **CongWin is cut in half**
  - **window then grows linearly**
- **But after timeout event:**
  - **CongWin instead set to 1 MSS;**
  - **window then grows exponentially**
  - **to a threshold, then grows linearly**

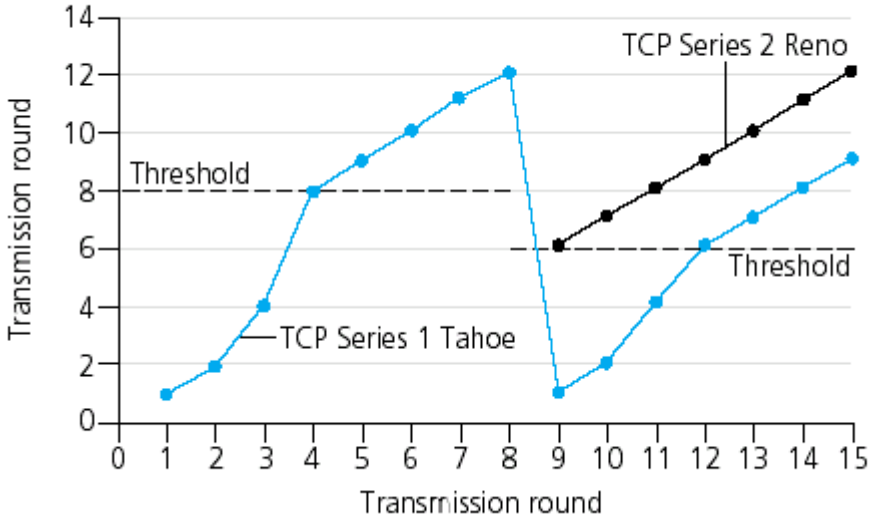
Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a "more alarming" congestion scenario

Refinement

Q: When should the exponential increase switch to linear?

A: When `CongWin` gets to 1/2 of its value before timeout.



Implementation:

- **Variable Threshold**
- **At loss event, Threshold is set to 1/2 of CongWin just before loss event**

发送窗口的上限值

- 发送方的发送窗口的上限值应当取为接收方窗口 **rwnd** 和拥塞窗口 **cwnd** 这两个变量中较小的一个，即应按以下公式确定：

发送窗口的上限值 = **Min [rwnd, cwnd]**

当 **rwnd < cwnd** 时，是接收方的接收能力限制发送窗口的最大值。

- 当 **cwnd < rwnd** 时，则是网络的拥塞限制发送窗口的最大值。

Summary: TCP Congestion Control

- When `CongWin` is below `Threshold`, sender in slow-start phase, window grows exponentially.
- When `CongWin` is above `Threshold`, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, `Threshold` set to `CongWin/2` and `CongWin` set to `Threshold`.
- When timeout occurs, `Threshold` set to `CongWin/2` and `CongWin` is set to 1 MSS.

4.5.4 TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. `RcvWindow`)
- *client:* connection initiator

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

- *server:* contacted by client

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

Three way handshake:

Step 1: client host sends TCP SYN segment to server
- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

1. 运输连接的三个阶段

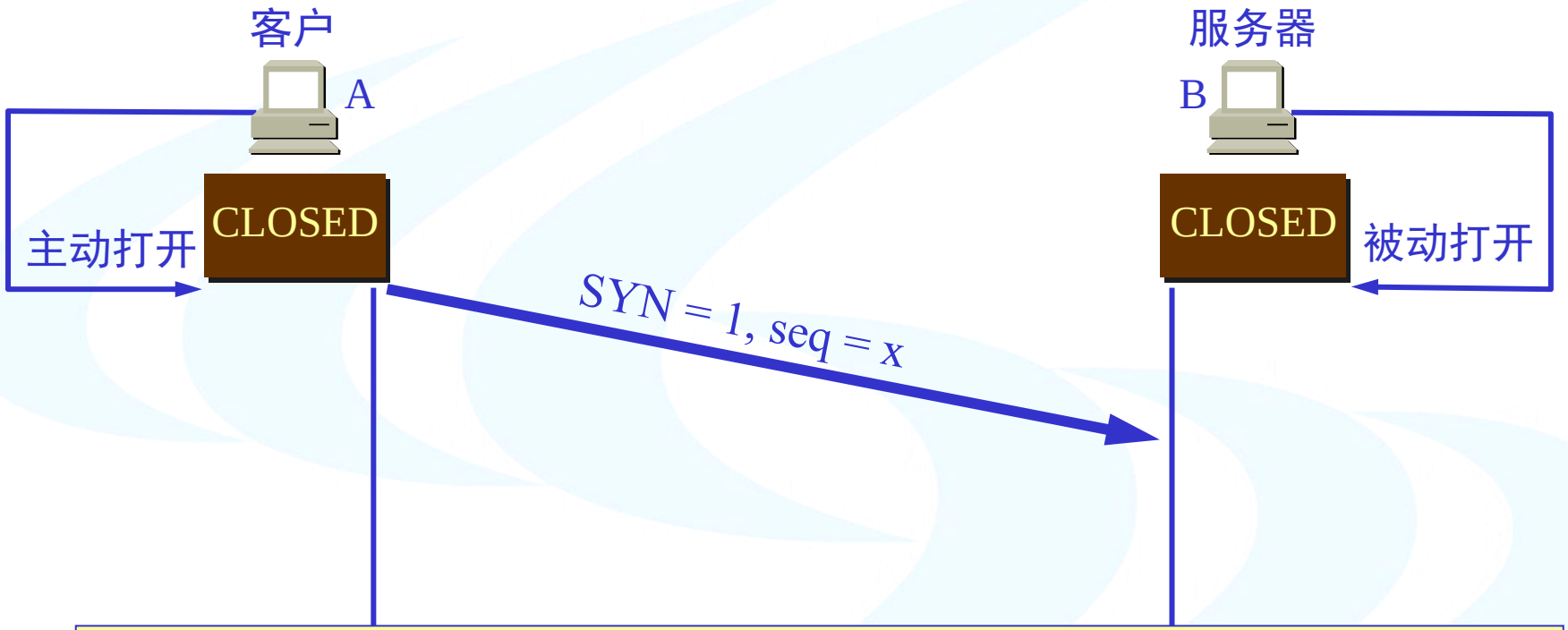
- 运输连接就有三个阶段，即：连接建立、数据传送和连接释放。运输连接的管理就是使运输连接的建立和释放都能正常地进行。
- 连接建立过程中要解决以下三个问题：
  - 要使每一方能够确知对方的存在。
  - 要允许双方协商一些参数（如最大报文段长度，最大窗口大小，服务质量等）。
  - 能够对运输实体资源（如缓存大小，连接表中的项目等）进行分配。

客户服务器方式

- **TCP** 连接的建立都是采用客户服务器方式。
- 主动发起连接建立的应用进程叫做客户**(client)**。
- 被动等待连接建立的应用进程叫做服务器**(server)**。

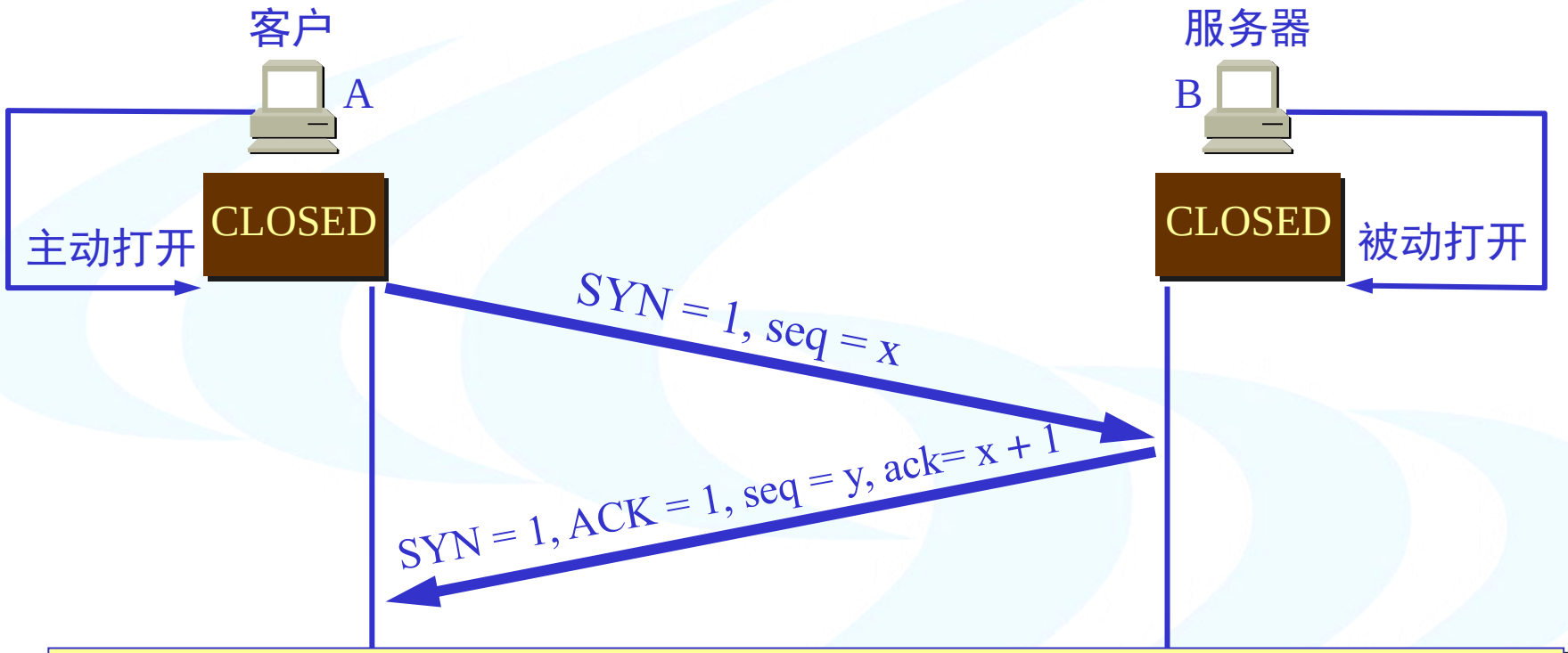
TCP 的连接建立

用三次握手建立 TCP 连接



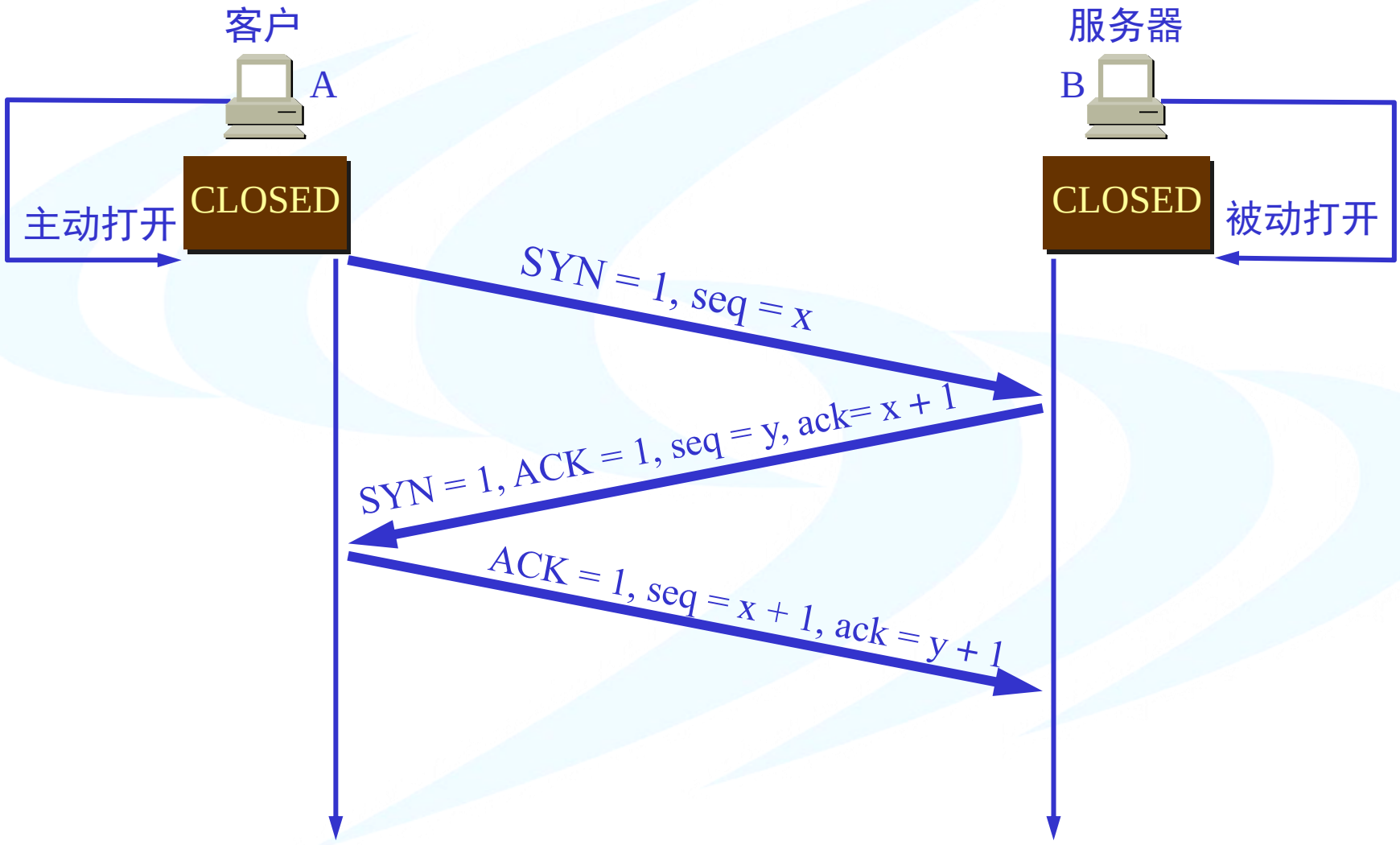
A 的 TCP 向 B 发出连接请求报文段，其首部中的同步位 SYN = 1，并选择序号 seq = x，表明传送数据时的第一个数据字节的序号是 x。

用三次握手建立 TCP 连接

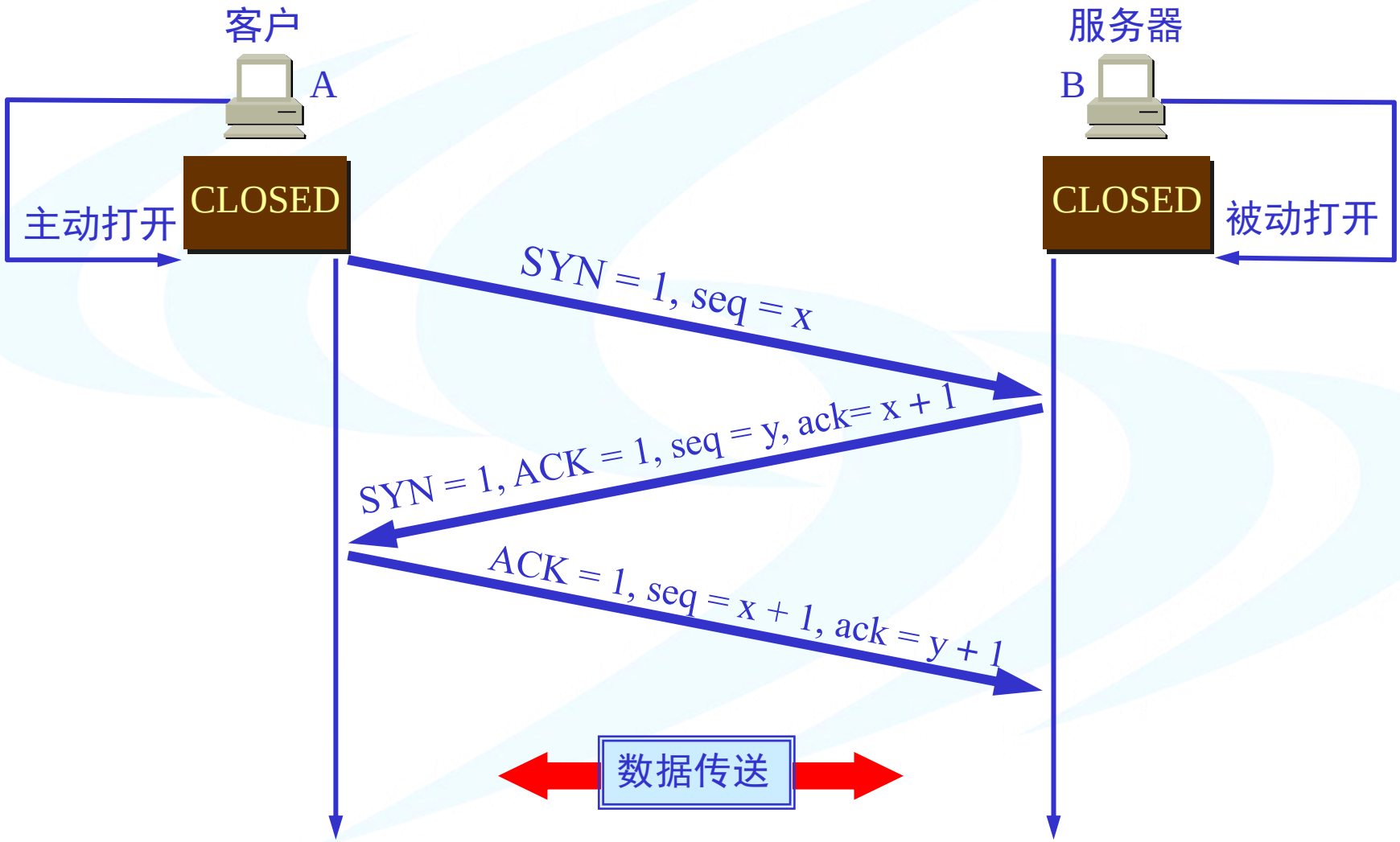


- B 的 TCP 收到连接请求报文段后，如同意，则发回确认。
- B 在确认报文段中应使 SYN = 1，使 ACK = 1，其确认号ack = x + 1，自己选择的序号 seq = y。

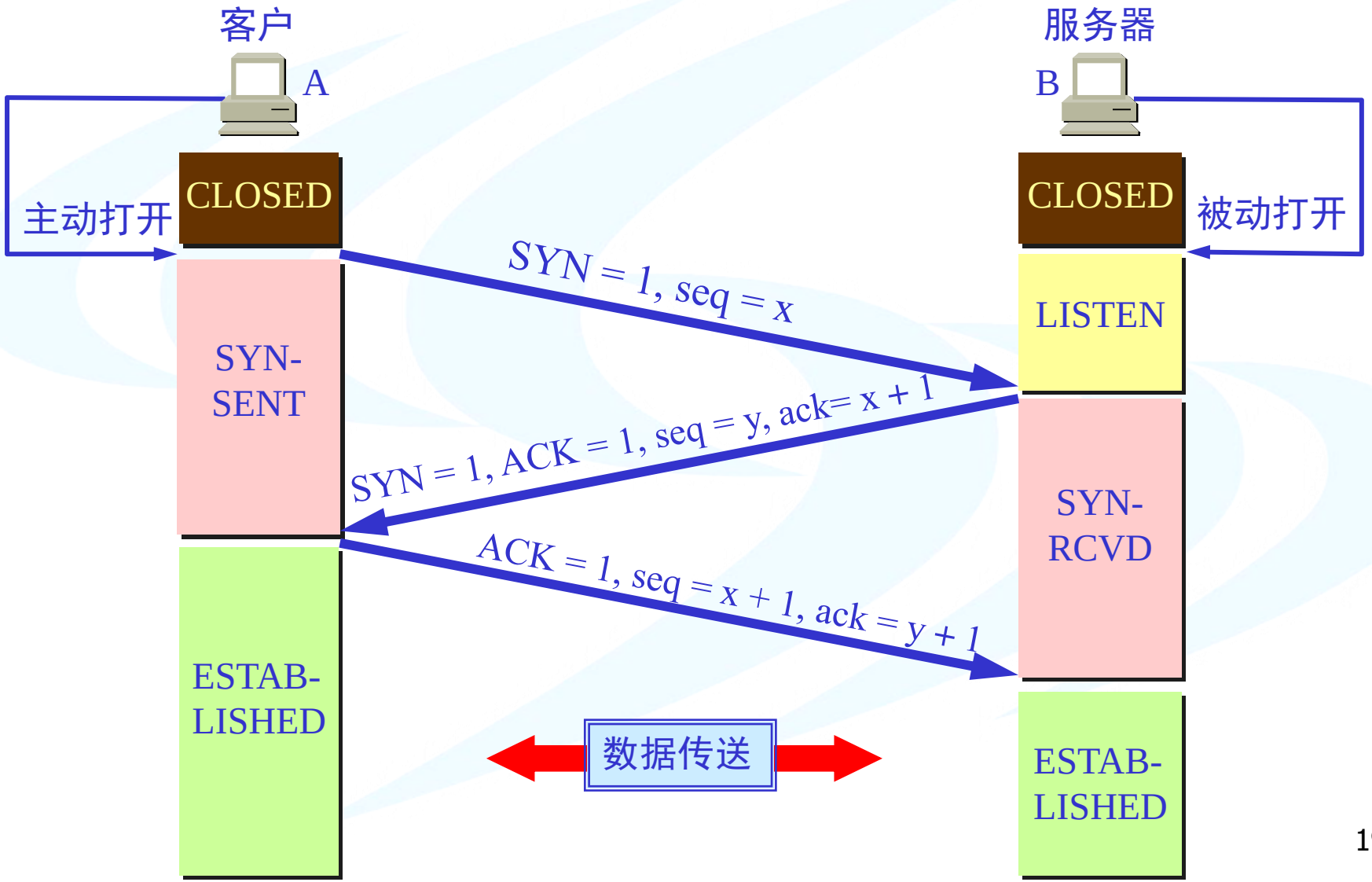
- A 收到此报文段后向 B 给出确认，其 ACK = 1，确认号 ack = y + 1。
- A 的 TCP 通知上层应用进程，连接已经建立。



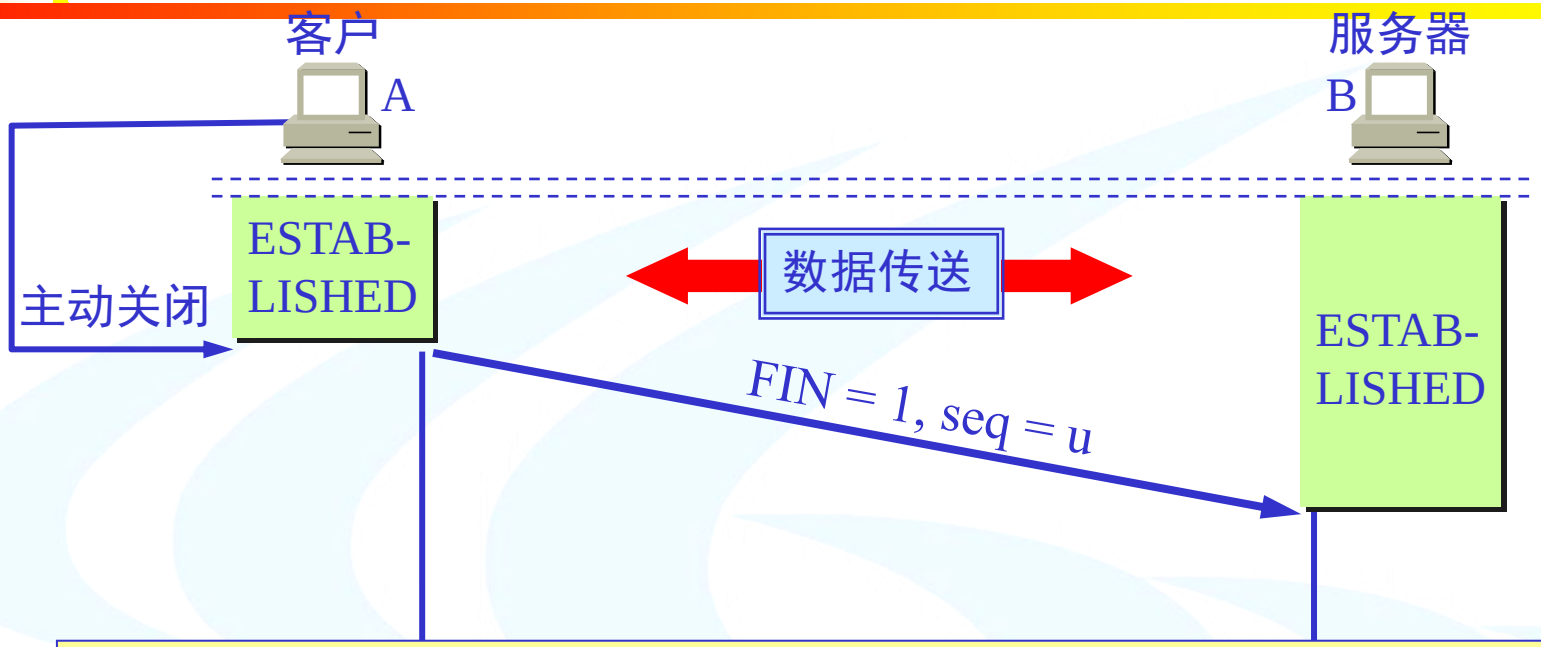
- B 的 TCP 收到主机 A 的确认后，也通知其上层应用进程：TCP 连接已经建立。



用三次握手建立 TCP 连接的各状态

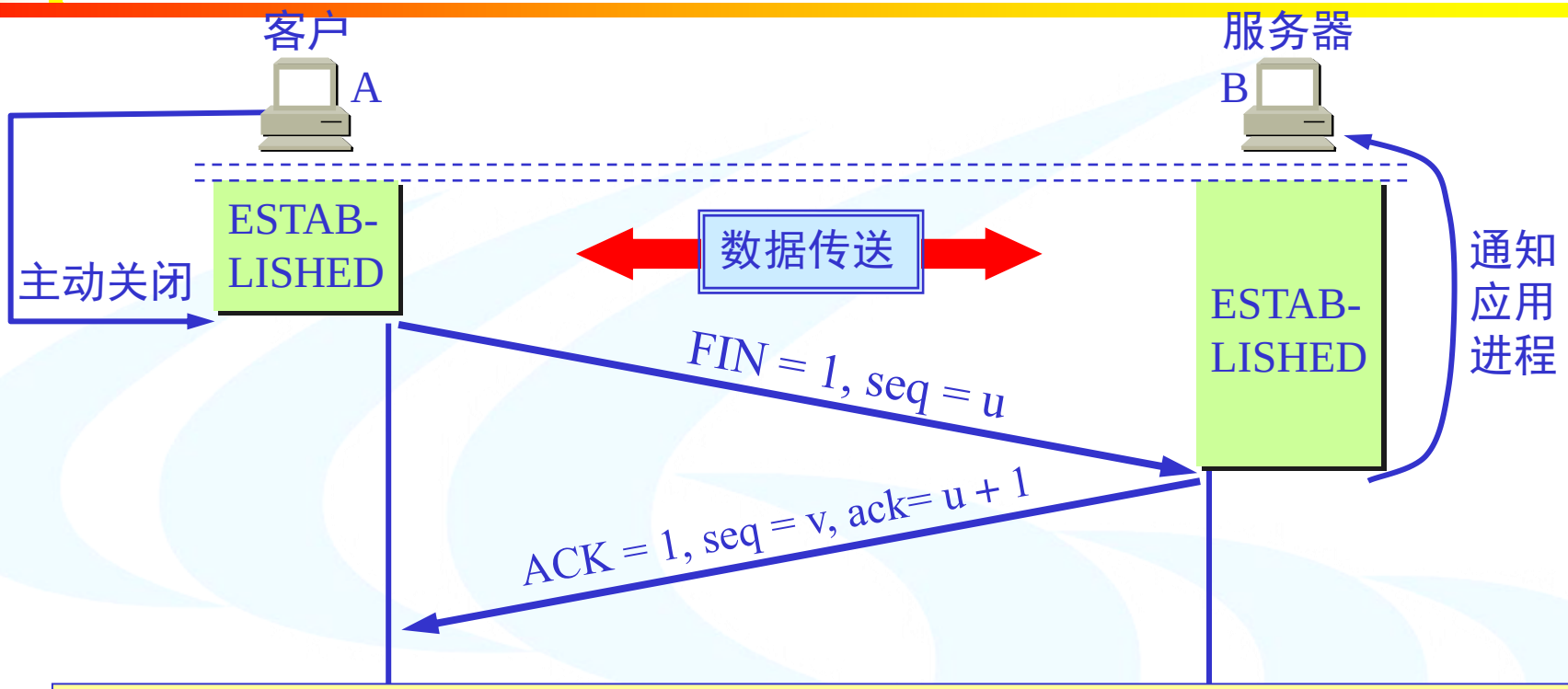


TCP 的连接释放

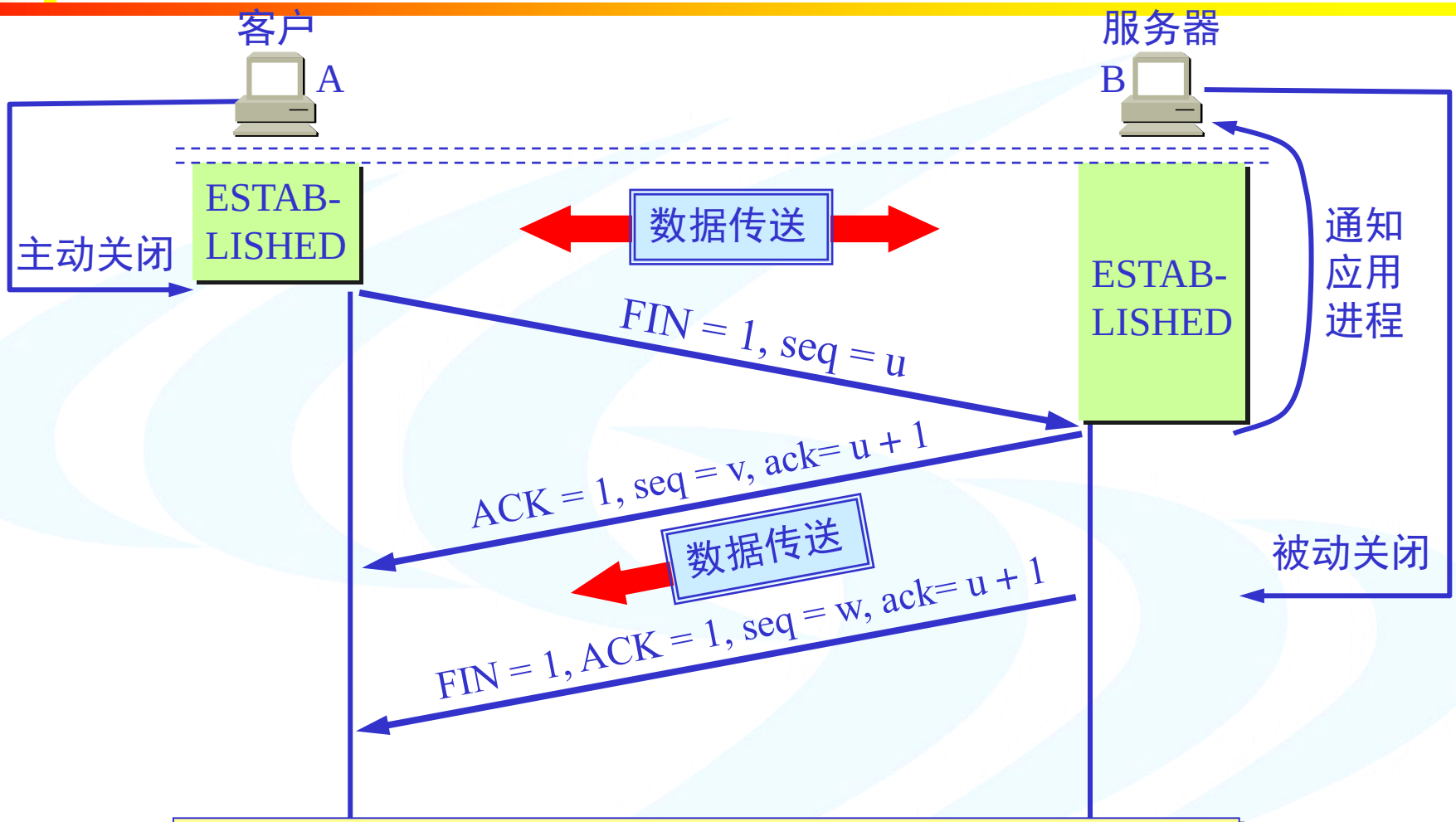


- 数据传输结束后，通信的双方都可释放连接。现在 A 的应用进程先向其 TCP 发出连接释放报文段，并停止再发送数据，主动关闭 TCP 连接。
- A 把连接释放报文段首部的 FIN = 1，其序号 seq = u，等待 B 的确认。

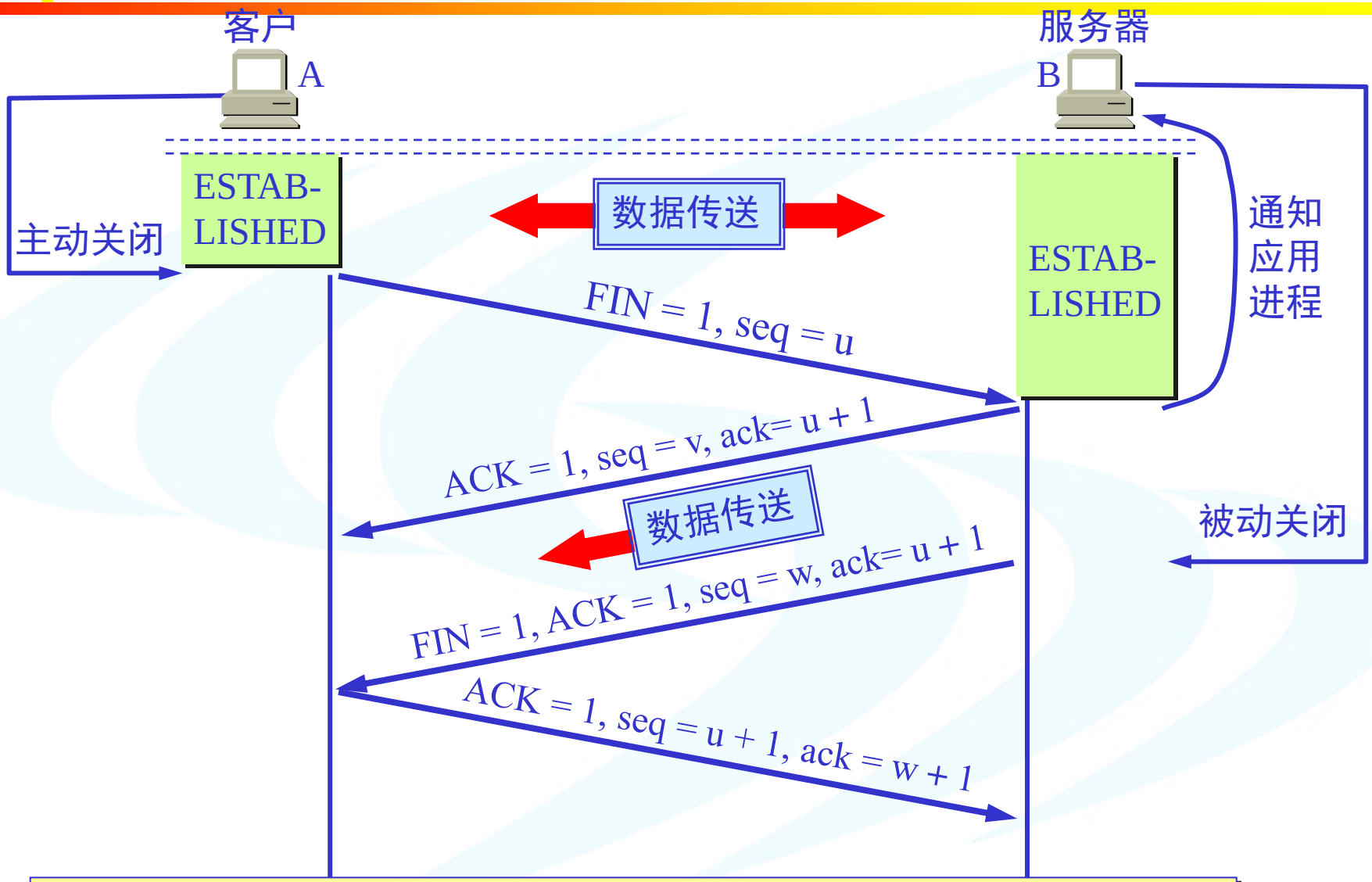
CLOSED



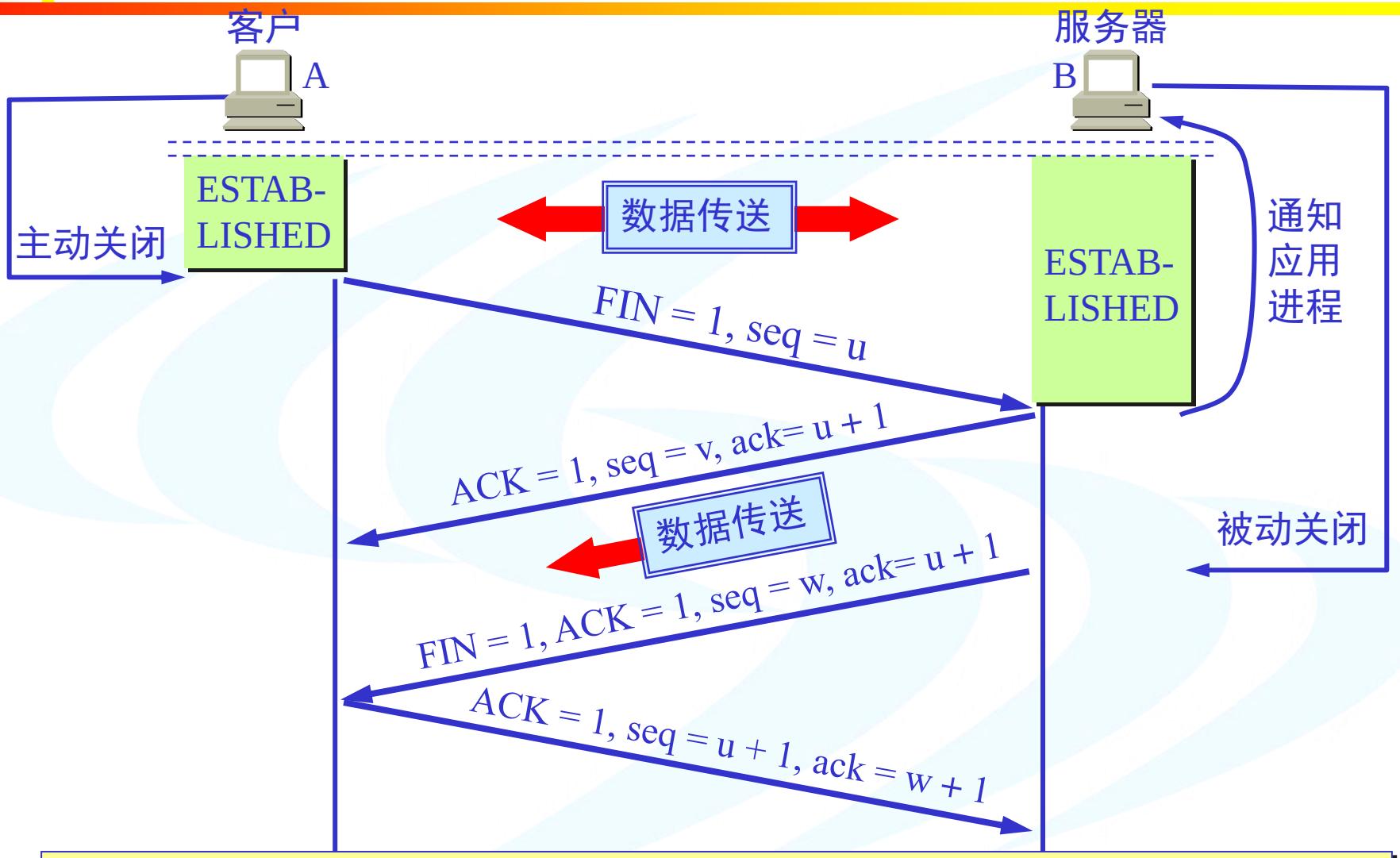
- B 发出确认，确认号 ack = u + 1，而这个报文段自己的序号 seq = v。
- TCP 服务器进程通知高层应用进程。
- 从 A 到 B 这个方向的连接就释放了，TCP 连接处于半关闭状态。B 若发送数据，A 仍要接收。



- 若 B 已经没有要向 A 发送的数据，其应用进程就通知 TCP 释放连接。

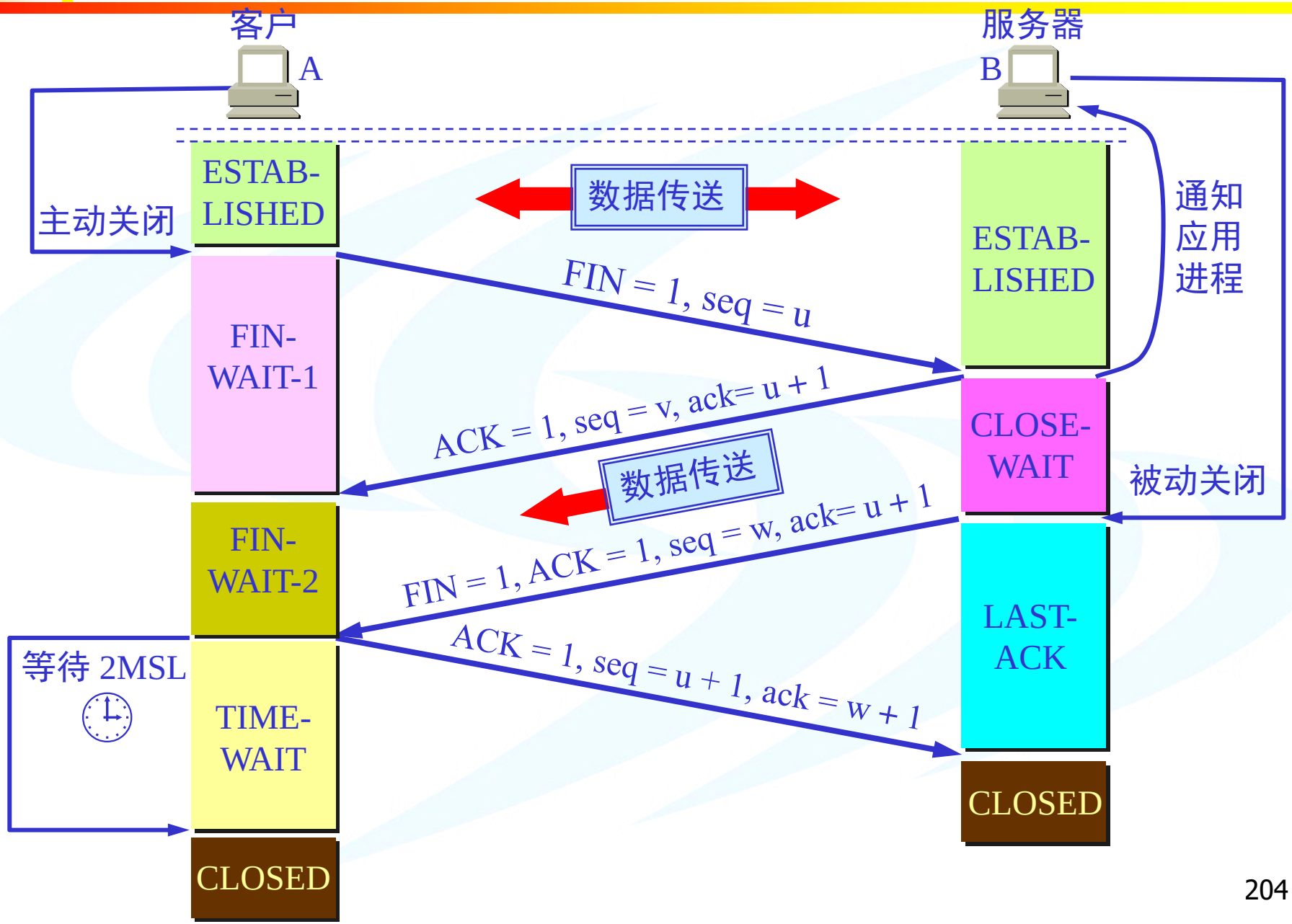


- A 收到连接释放报文段后，必须发出确认。



- 在确认报文段中 ACK = 1，确认号 ack = w + 1，自己的序号 seq = u + 1。

TCP 连接必须经过时间 2MSL 后才真正释放掉。



A 必须等待 2MSL 的时间

- 第一，为了保证 **A** 发送的最后一个 **ACK** 报文段能够到达 **B**。
- 第二，防止“已失效的连接请求报文段”出现在本连接中。**A** 在发送完最后一个 **ACK** 报文段后，再经过时间 **2MSL**，就可以使本连接持续的时间内所产生的所有报文段，都从网络中消失。这样就可以使下一个新的连接中不会出现这种旧的连接请求报文段。

TCP Connection Management (cont.)

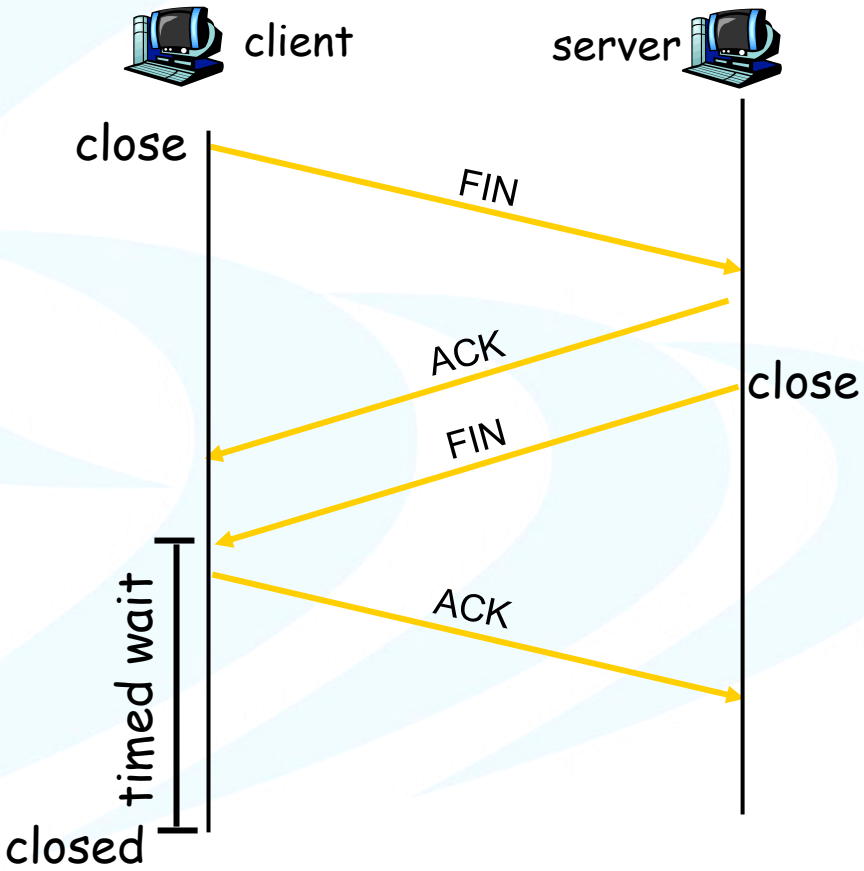
Closing a connection:

client closes socket:

```
clientSocket.close(
);
```

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.



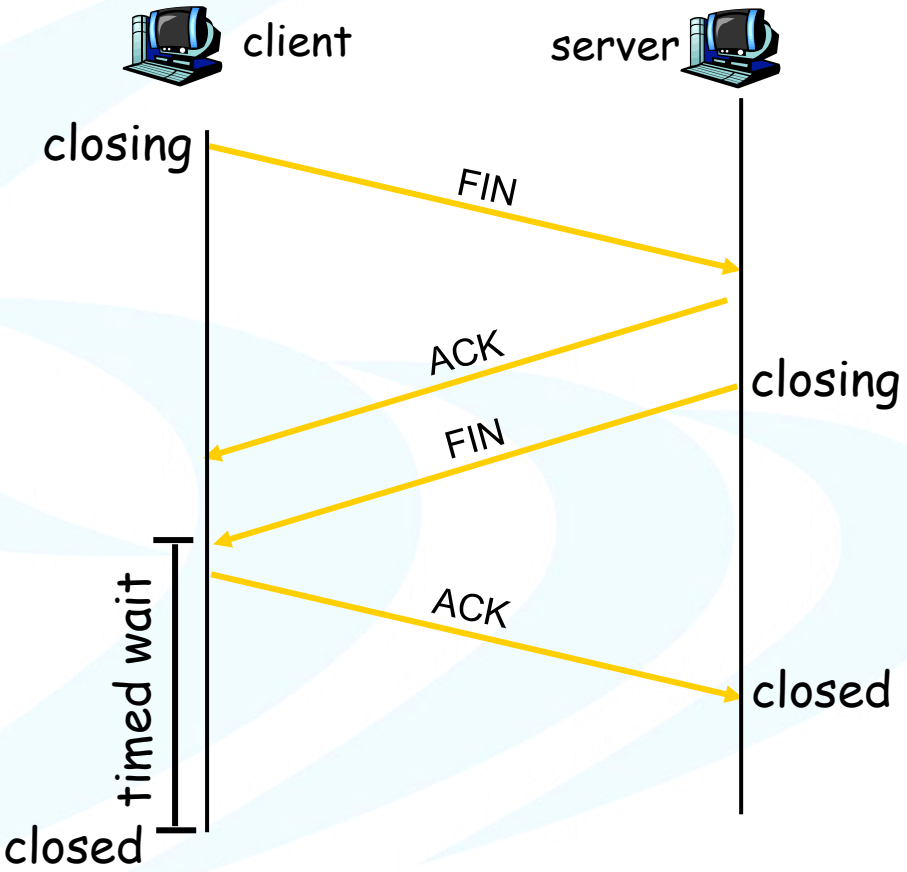
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

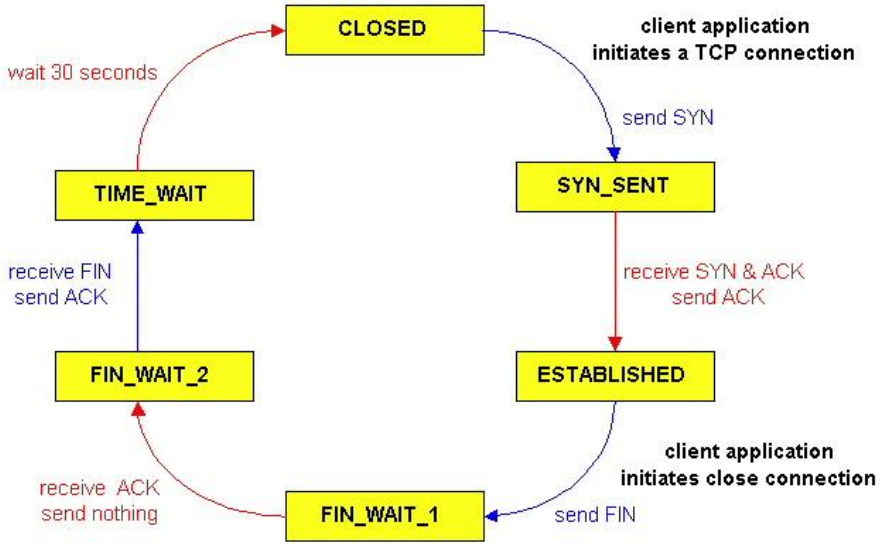
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.

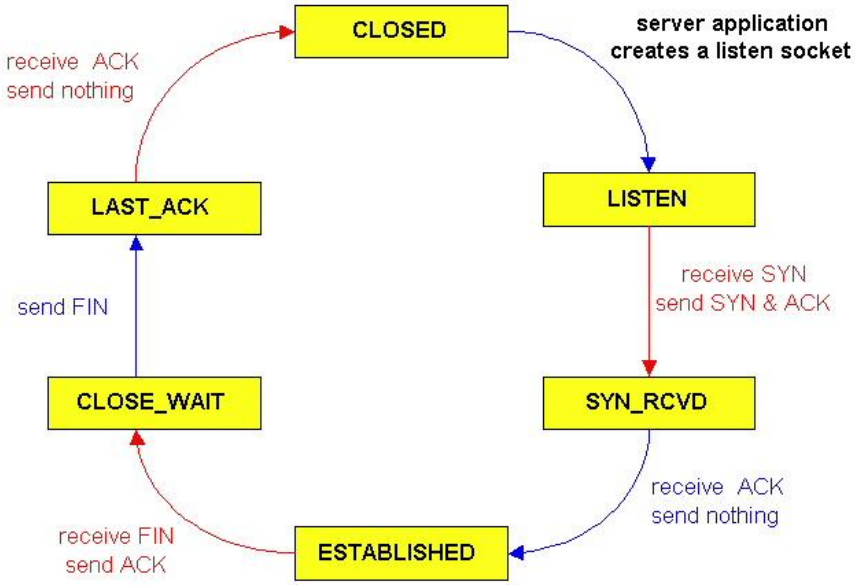


TCP Connection Management



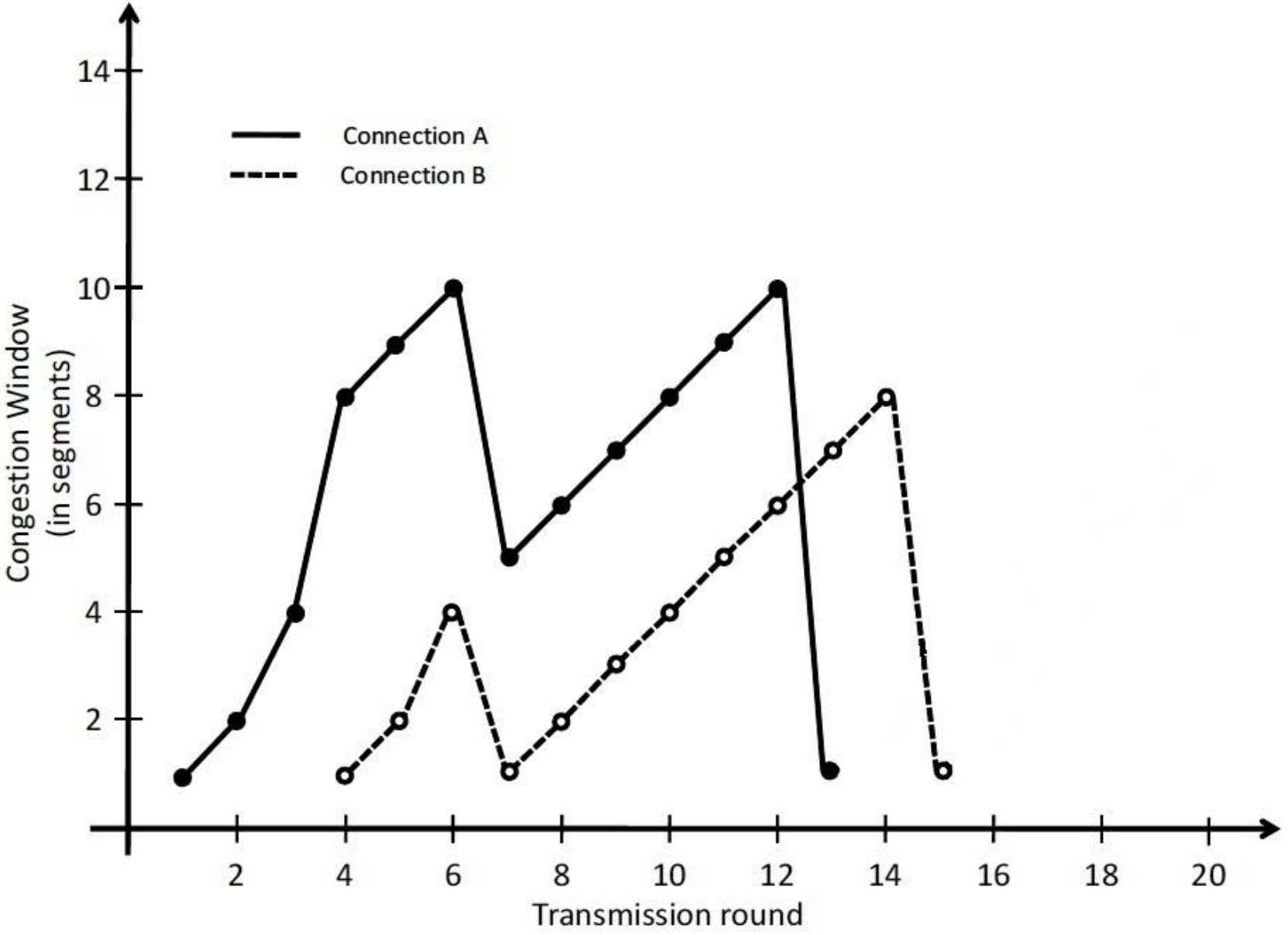
TCP client lifecycle

TCP server lifecycle



Chapter 4: Summary

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


Connection A
Connection B
Congestion Window
(in segments)
14
12
10
8
6
4
2
2
4
6
8
10
12
14
16
18
20
Transmission round

Consider the following plot of TCP window size as a function of time for two TCP connections A and B. In this problem we will suppose that both TCP senders are sending large files. We also assume that the packet loss events are independent in connection A and B.

Question 1: Considering the above values of congestion window (CongWin) for these connections, can we identify the type of TCP connections (Reno or Tahoe) that have been used by connection A and B?

Question 2: What are the values of the Threshold parameter between the 1st and the 14th transmission rounds for each connection?

Question 3: Assume that the segment size is 1460 bytes and that a total of 87600 bytes have been successfully transmitted over connection A before the 13th transmission round. At which transmission round the cumulative amount of the successful transmitted data is equal to 163520 bytes? Again we assume that there is neither timeout nor duplicate ACK after the 13th transmission round.