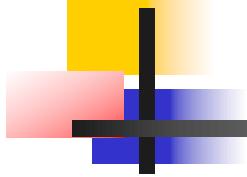


计算机网络

第 5 章 运输层

授课教师：洪锋

<http://osn.ouc.edu.cn/~hong>



第 5 章 运输层

5.1 运输层协议概述

5.1.1 进程之间的通信

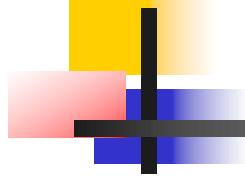
5.1.2 运输层的两个主要协议

5.1.3 运输层的端口

5.2 用户数据报协议 UDP

5.2.1 UDP 概述

5.2.2 UDP 的头部格式



第 5 章 运输层（续）

5.3 传输控制协议 TCP 概述

5.3.1 TCP 最主要的特点

5.3.2 TCP 的连接

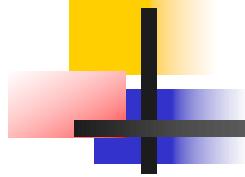
5.9 TCP 的运输连接管理

5.4 可靠传输的工作原理

5.4.1 停止等待协议

5.4.2 连续 ARQ 协议

5.5 TCP 报文段的头部格式



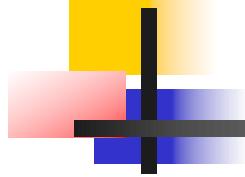
第5章 运输层（续）

5.6 TCP 可靠传输的实现

- 5.6.1 以字节为单位的滑动窗口
- 5.6.2 超时重传时间的选择
- 5.6.3 选择确认 SACK

5.7 TCP的流量控制

- 5.7.1 利用滑动窗口实现流量控制
- 5.7.1 必须考虑传输效率



第5章 运输层（续）

5.8 TCP 的拥塞控制

5.8.1 拥塞控制的一般原理

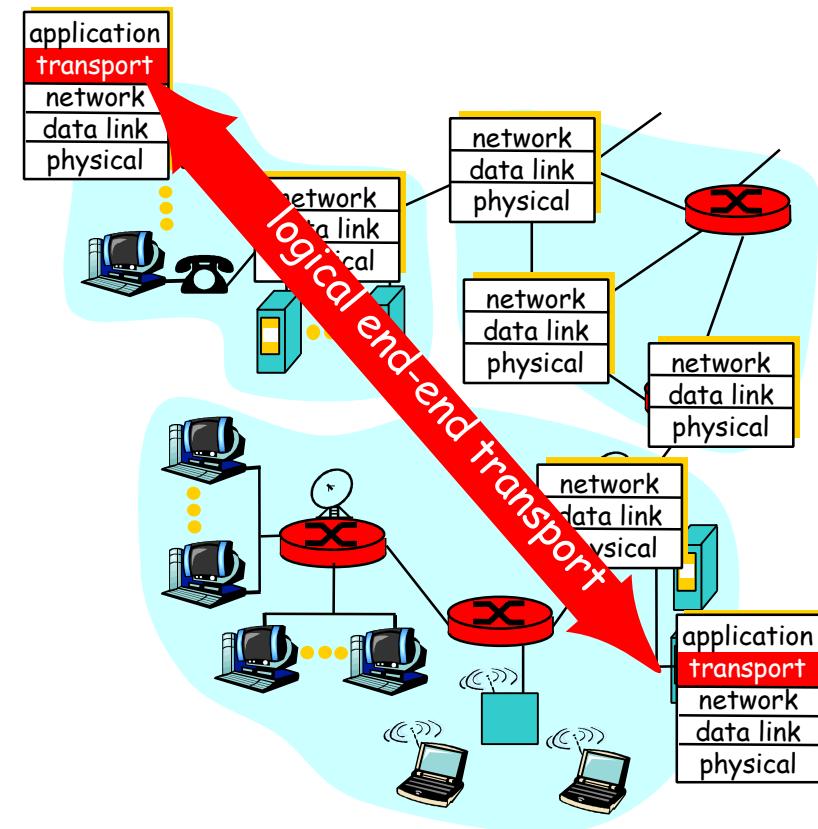
5.8.2 几种拥塞控制方法

5.8.3 随机早期检测 RED

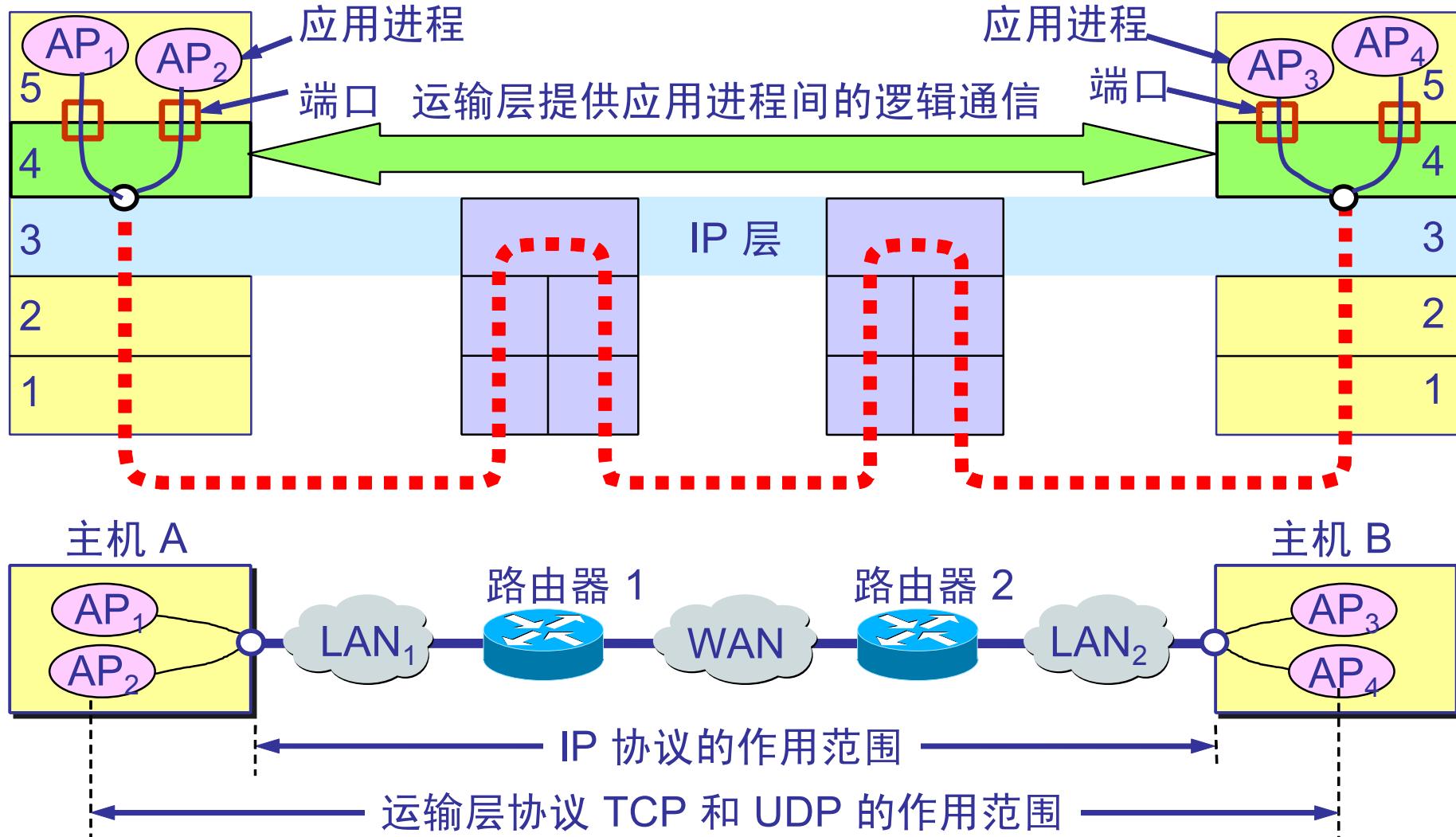
5.1 运输层协议概述

5.1.1 进程之间的通信

- 运输层向应用层提供通信服务
 - 面向通信部分的最高层
 - 用户功能中的最低层。
- 传输层 vs. 网络层
 - 网络层：在端系统间进行通信
 - 传输层：在进程间进行通信
 - 依赖于、加强了 网络层的服务

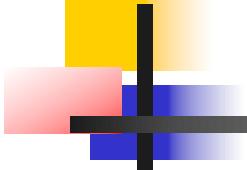


运输层为相互通信的应用进程提供了逻辑通信



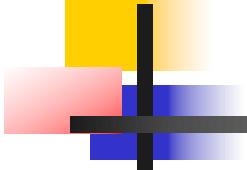
运输层协议和网络层协议的主要区别





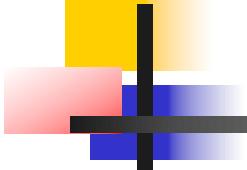
应用进程之间的通信

- 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**。
 - 运输层之间的通信**好像是**沿水平方向传送数据。
 - 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。



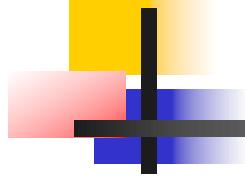
运输层的主要功能

- 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。
- 运输层对收到的报文进行差错检测。
- 运输层有两种不同功能的运输协议
 - 无连接的 UDP
 - 不可靠的（“尽力而为”）、无序的点对点或广播递交
 - 面向连接的 TCP
 - **可靠的、按序点对点递交**
 - 拥塞控制、流量控制、连接建立
- **不能提供的服务：**
 - 实时性、带宽承诺、可靠的广播通信



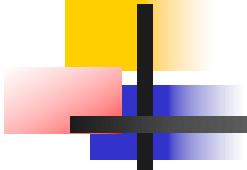
两种不同的运输协议

- 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
 - 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。
- 面向连接的 TCP 协议
 - 尽管下面的**网络是不可靠的**（只提供尽最大努力服务），这种逻辑通信信道就相当于一条全双工的**可靠信道**。
- 无连接的 UDP 协议时，
 - 逻辑通信信道是一条**不可靠信道**。



5.1.2 运输层的两个主要协议

- TCP/IP 的运输层有两个不同的协议
 - 用户数据报协议 UDP (User Datagram Protocol)
 - 传输控制协议 TCP (Transmission Control Protocol)
- 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**



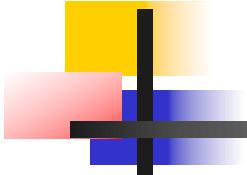
TCP 与 UDP 比较

- UDP 特点:

- 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。
- 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 特点:

- 提供面向连接的服务。
- 不提供广播或多播服务。
- 开销大:
 - 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。
 - 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。

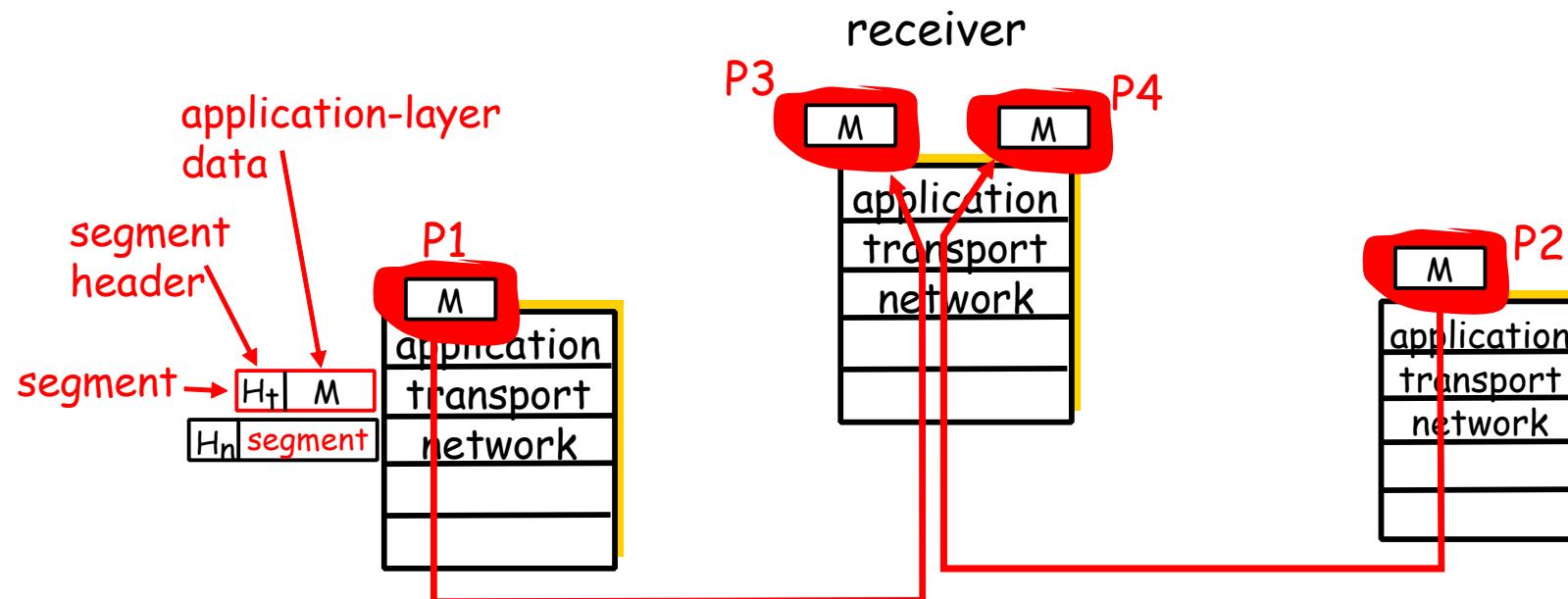


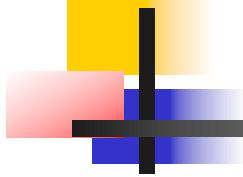
强调两点

- 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。

5.1.3 运输层的端口

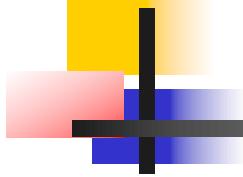
分用: 将接收到的段传递给正确的应用层进程





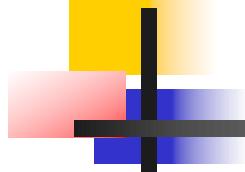
进程的标识和区分问题

- 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。
 - 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

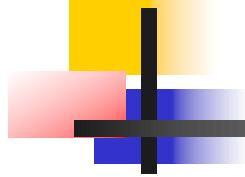
- 由于进程的创建和撤销都是动态的，发送方几乎无法识别其他机器上的进程。
- 有时我们会改换接收报文的进程，但并不需要通知所有发送方。
- 我们往往需要利用目的主机提供的功能来识别终点，而不需要知道实现这个功能的进程。



端口号(protocol port number)

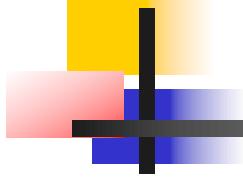
简称为端口(port)

- 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。



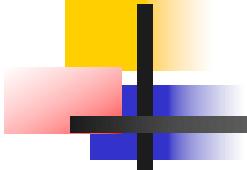
软件端口与硬件端口

- 在协议栈层间的抽象的协议端口是**软件端口**。
 - 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。
- 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。



运输层端口

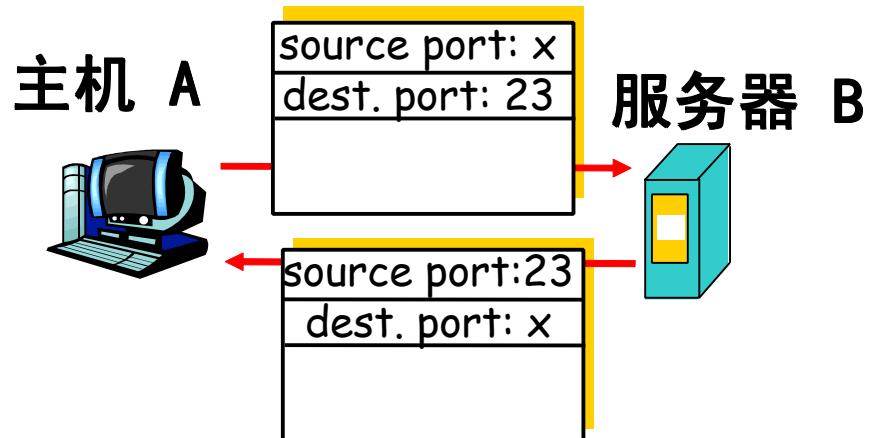
- 端口用一个 16 位端口号进行标志。
- 端口号只具有**本地**意义，即端口号只是为了标志本计算机应用层中的各进程。
- 在因特网中不同计算机的相同端口号是没有联系的。



三类端口

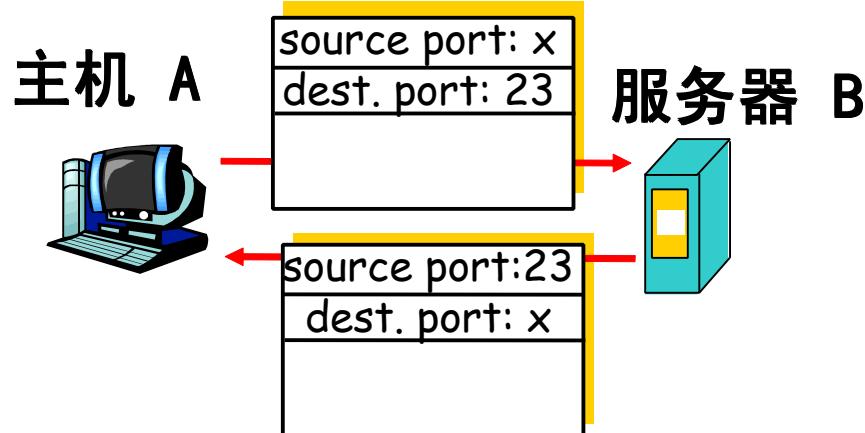
- 熟知端口，数值一般为 0~1023。
 - 20,21 FTP; 22 SSH; 23 Telnet; 25 SMTP
 - 66 Oracle SQL*NET; 80 WWW; 110 PoP3; 443 安全 HTTP
- 登记端口号，数值为 1024~49151
 - 为没有熟知端口号的应用程序使用的。
 - 使用这个范围的端口号必须在 IANA 登记，以防止重复。
 - 1433 Microsoft SQL Server 数据库服务; 8080 WWW (Tomcat, Apache)
- 客户端口号或短暂端口号，
 - 数值为 49152~65535，留给客户进程选择暂时使用。
 - 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。

端口应用举例



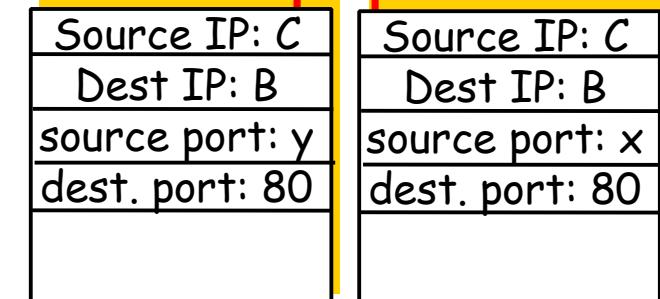
端口的使用：简单的 telnet 应用

端口应用举例

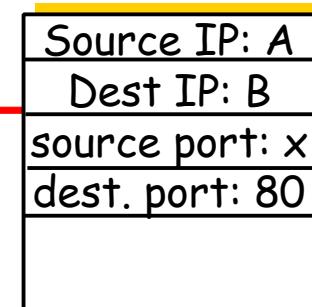


端口的使用：简单的 telnet 应用

Web客户端
主机 C

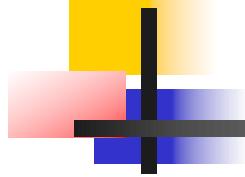


Web客户端
主机 A



Web
服务器 B

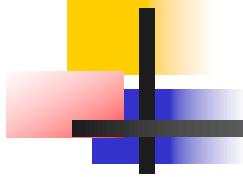
端口的使用： Web 服务器



5.2 用户数据报协议 UDP

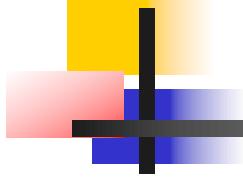
5.2.1 UDP 概述

- “最简约的” Internet 传输协议
 - UDP 在 IP 数据报服务之上增加了很多一点的功能：
 - 端口管理
 - 差错检测
 - “尽力而为的” 服务, UDP 数据段：
 - 丢失
 - 应用数据不按序到达



UDP 的主要特点

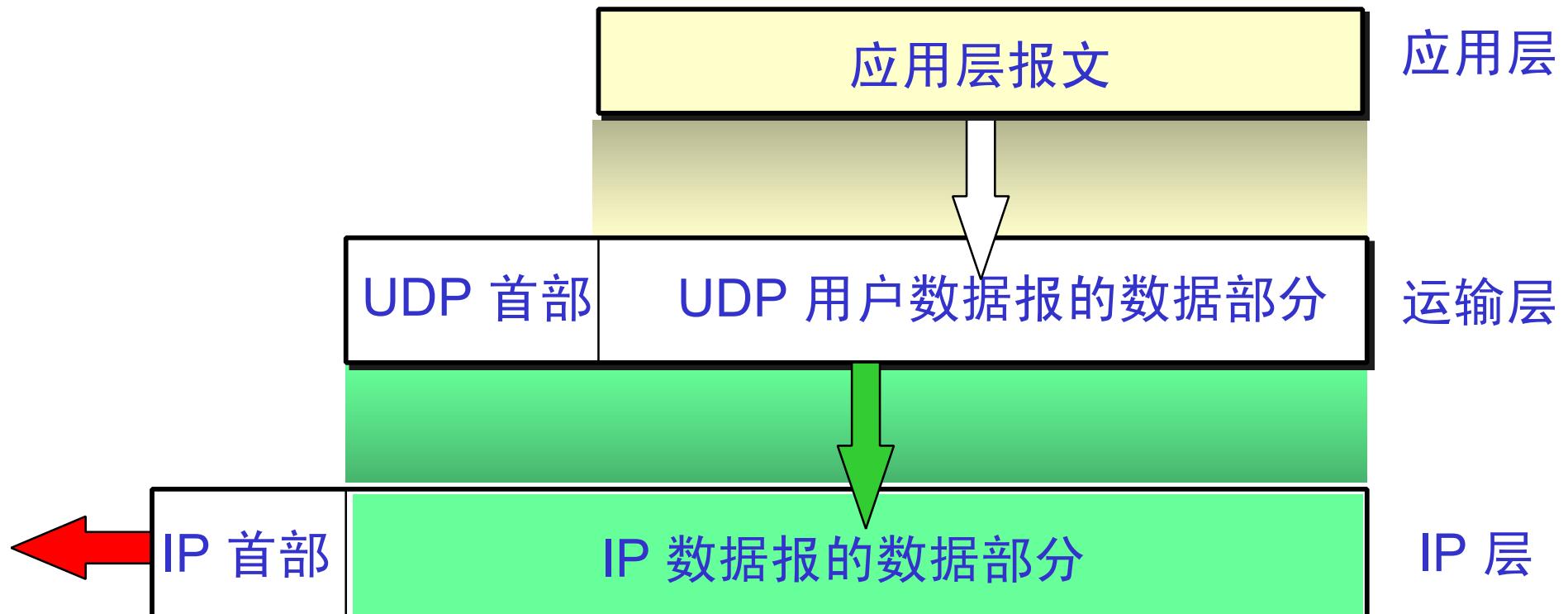
- UDP 是无连接的
 - 发送数据之前不需要建立连接。
- 最大努力交付
 - 不保证可靠交付
 - 不使用拥塞控制。
- UDP 是面向报文的。
- UDP 支持一对一、一对多、多对一和多对多的交互通信。
- UDP 的头部开销小，只有 8 个字节。



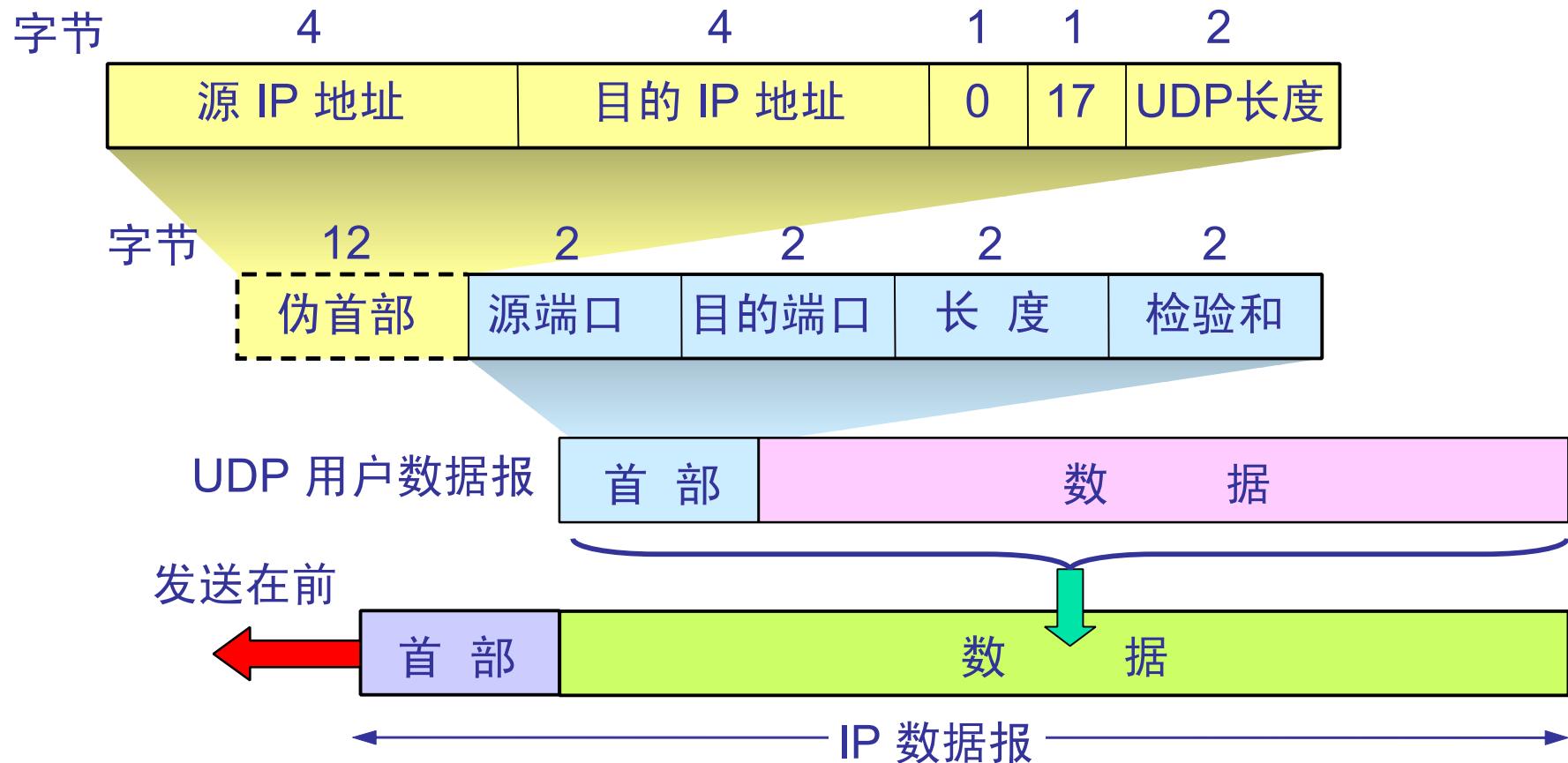
面向报文的 UDP

- 发送方 UDP 对应用程序交下来的报文，在添加头部后就向下交付 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，在去除头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的报文大小。

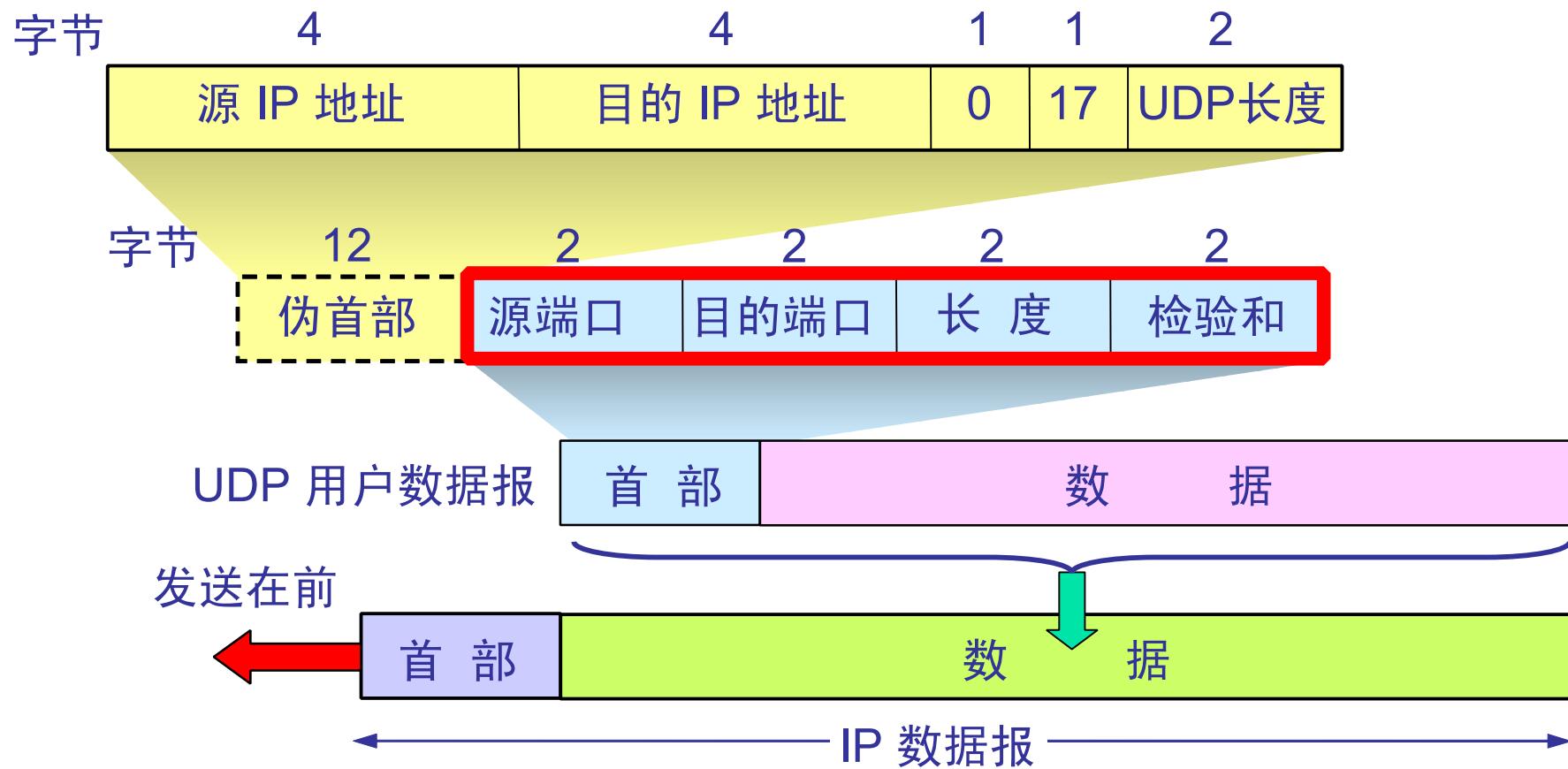
UDP 是面向报文的

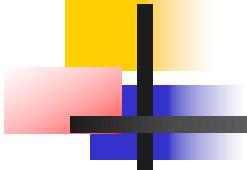


5.2.2 UDP 的头部格式

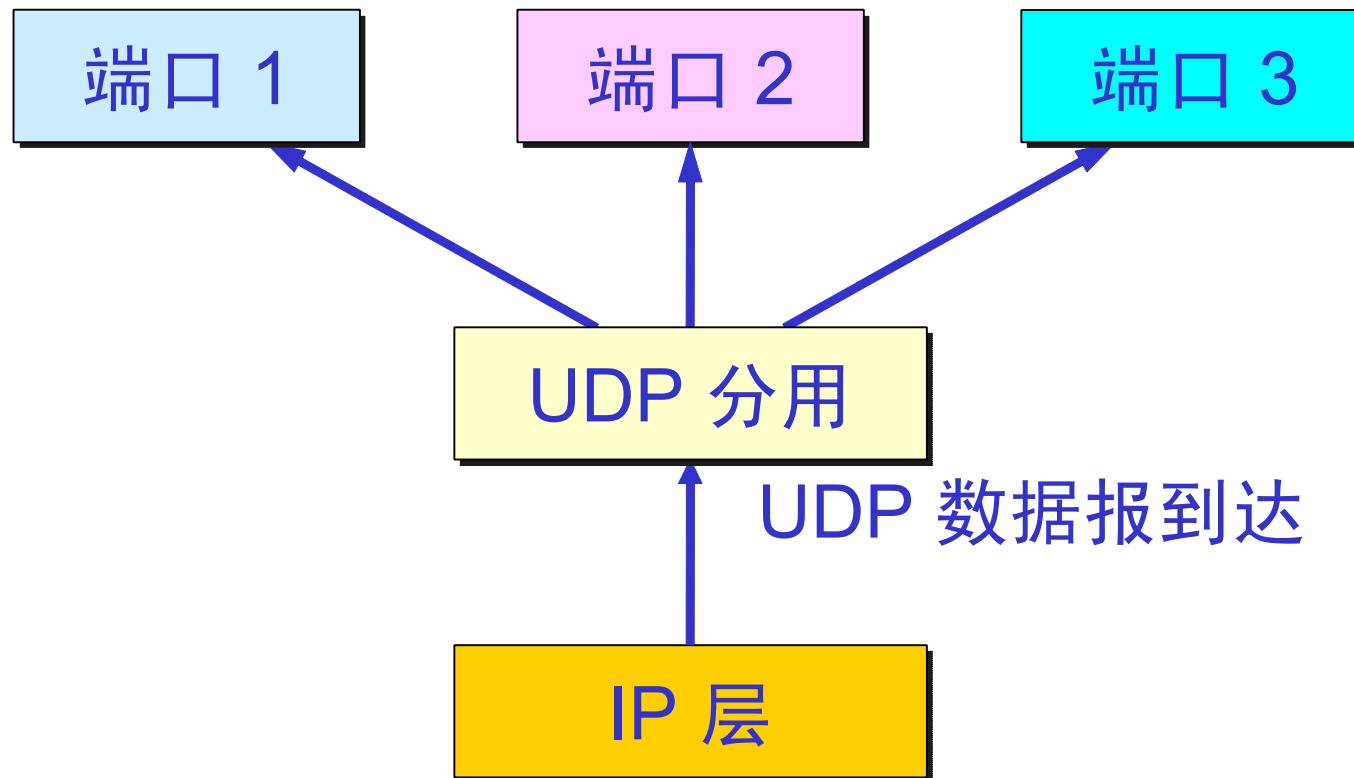


用户数据报 UDP 有两个字段：数据字段和首部字段。首部字段有 8 个字节，由 4 个字段组成，每个字段都是两个字节。

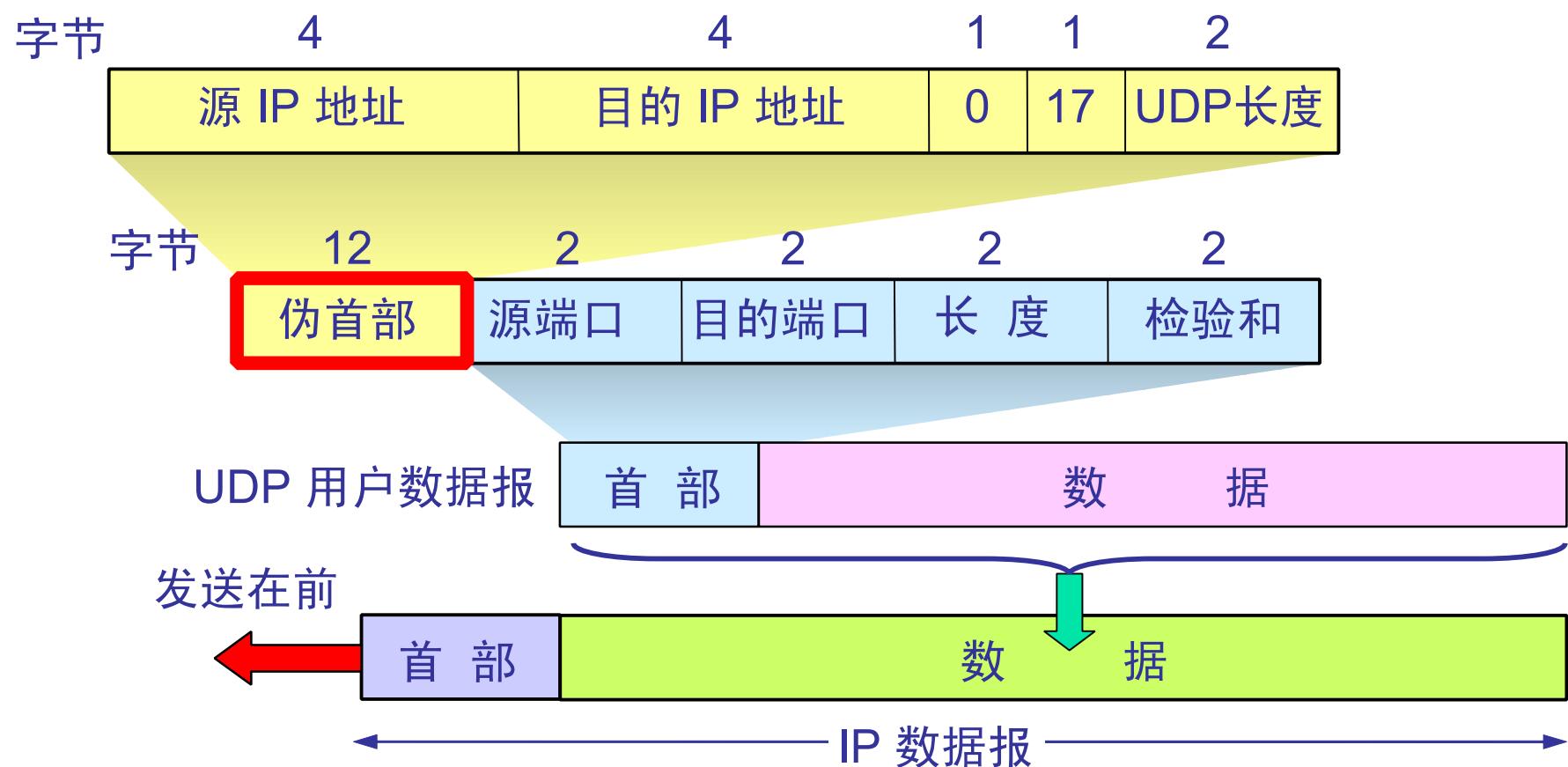




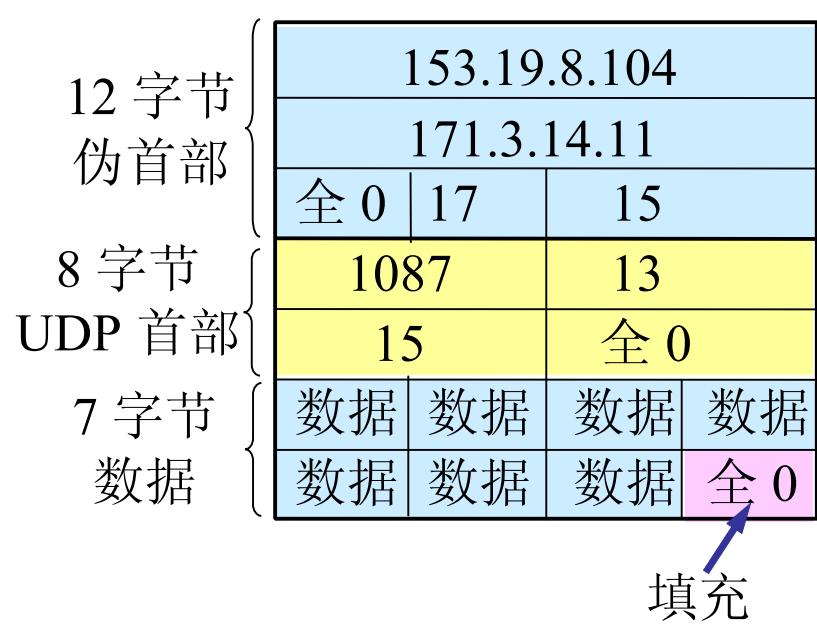
UDP 基于端口的分用



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。



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)

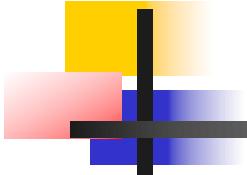
按二进制反码运算求和 10010110 11101101 → 求和得出的结果

将得出的结果求反码 01101001 00010010 → 检验和

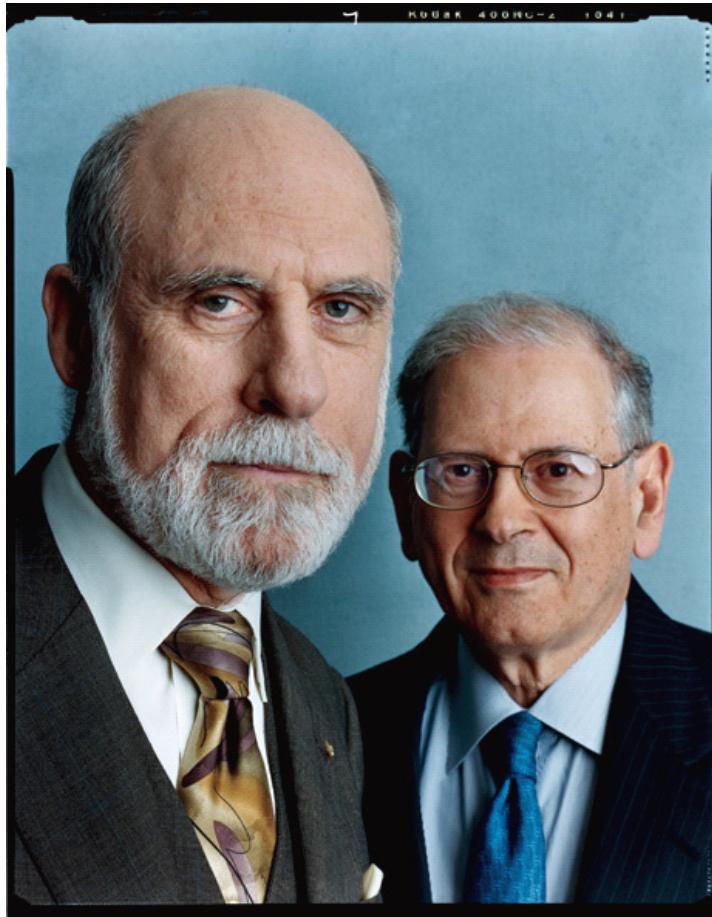


UDP应用

- 经常为流媒体应用使用
 - 允许数据丢失
 - 对传输速率敏感
- 其他 UDP用途 (why?)
 - RIP
 - DNS
- 若需要通过 UDP进行可靠传输——在应用层增加可靠性措施
 - 在应用程序中-专门的出错恢复机制!



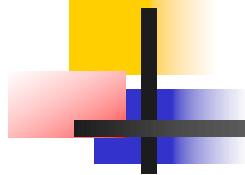
5.3 传输控制协议 TCP 概述



Vinton Cerf & Robert Kahn



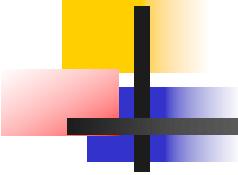
For pioneering work on internetworking, including the design and implementation of the Internet's basic communications protocols, TCP/IP, and for inspired leadership in networking.



5.3 传输控制协议 TCP 概述

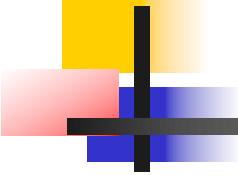
5.3.1 TCP 最主要的特点

- 面向连接的全双工通信
- 点对点
 - 一个发送方，一个接收方
- 可靠传输, 按序字节流
 - 无“报文边界”，无结构但有顺序
 - 发送&接收缓存
- 流水式控制
 - TCP的拥塞和流量控制，设置窗口大小
- 三个关键科学问题
 - 可靠传输
 - 流量控制
 - 拥塞控制



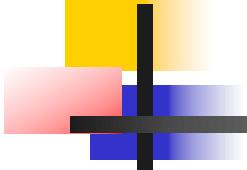
应当注意

- 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的字节后再构成报文段发出去。



5.3.2 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。
 - 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。

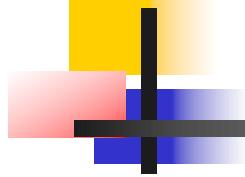


套接字 (socket)

套接字 socket = (IP地址: 端口号) (5-1)

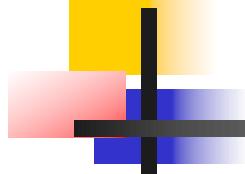
- 每一条 **TCP** 连接唯一地被通信两端的两个端点（即两个套接字）所确定。即：

TCP 连接 ::= {socket1, socket2}
= {(IP1: port1), (IP2: port2)} (5-2)



同一个名词 socket 有多种不同的意思

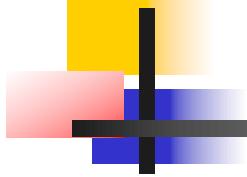
- 应用编程接口API称为 socket API, 简称为 socket。
- socket API 中使用的一个函数名也叫作 socket。
- 调用 socket 函数的端点称为 socket。
- 调用 socket 函数时其返回值称为 socket 描述符, 可简称为 socket。
- 在操作系统内核中连网协议的 Berkeley 实现, 称为 socket 实现。



5-9 TCP 的运输连接管理

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。

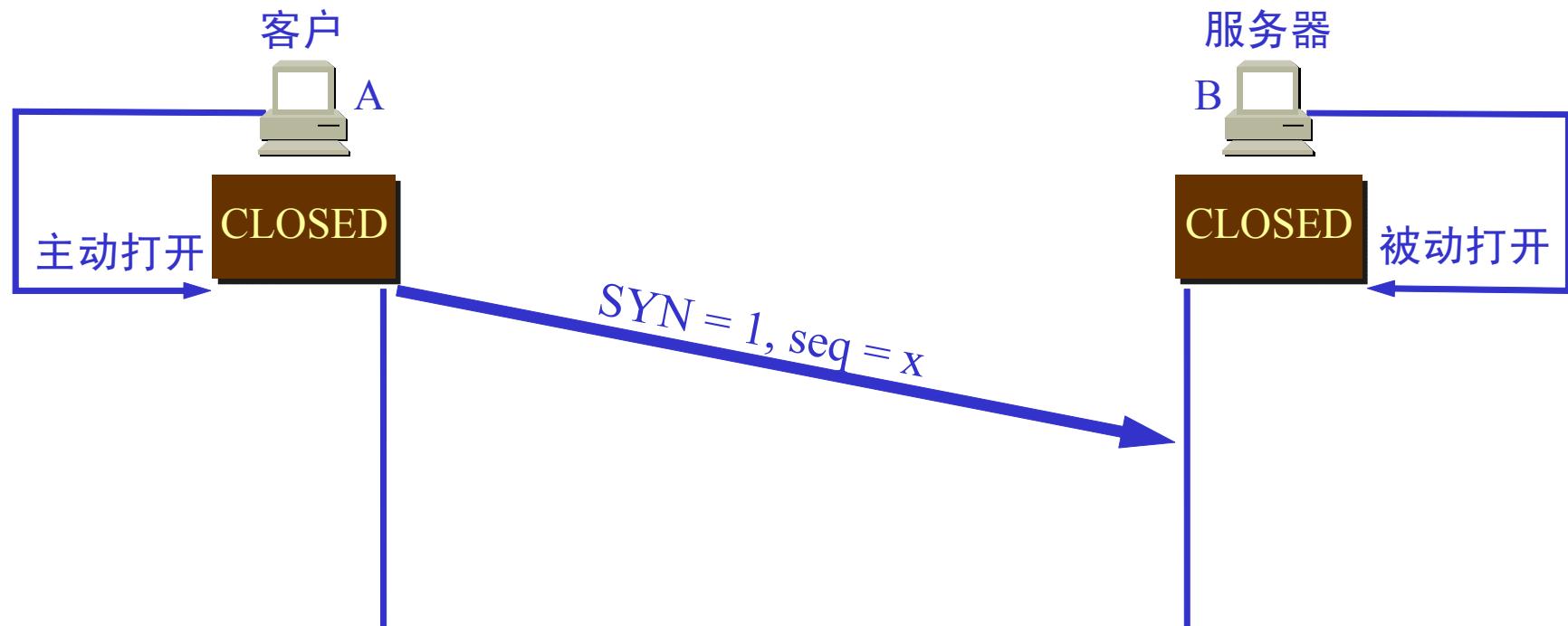


客户服务器方式

- 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)。

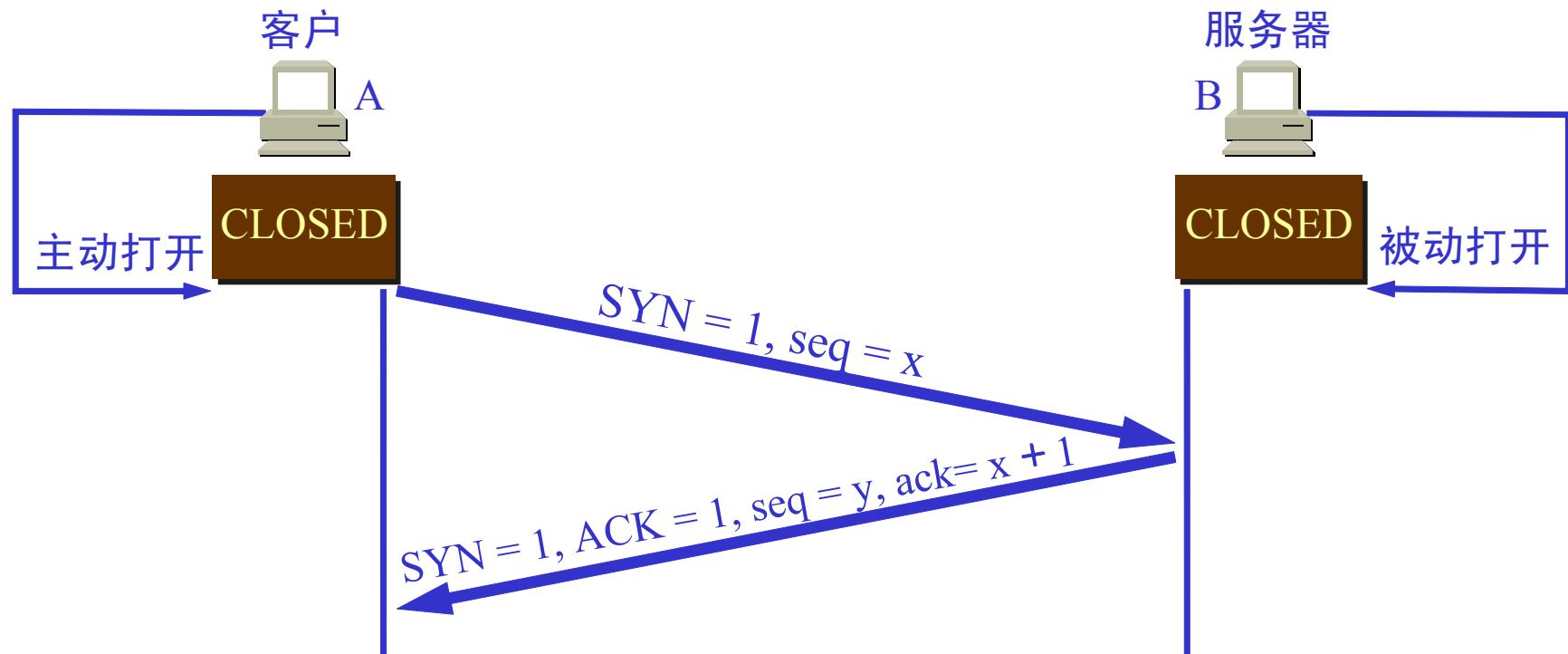
5.9.1 TCP 的连接建立

用三次握手建立 TCP 连接



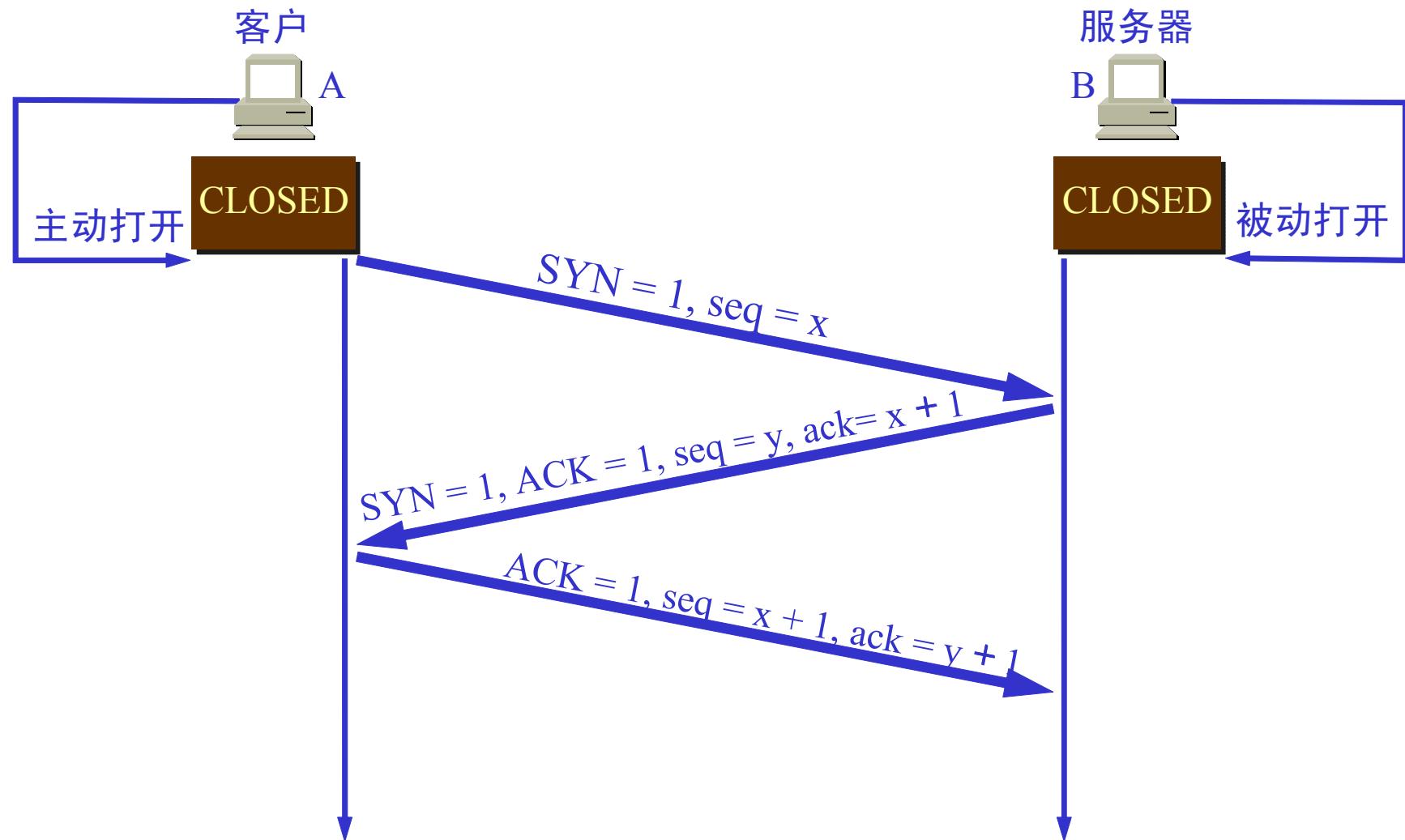
A 的 TCP 向 B 发出连接请求报文段，其首部中的同步位 $SYN = 1$ ，并选择序号 $seq = x$ ，表明传送数据时的第一个数据字节的序号是 x 。

5.9.1 TCP 的连接建立

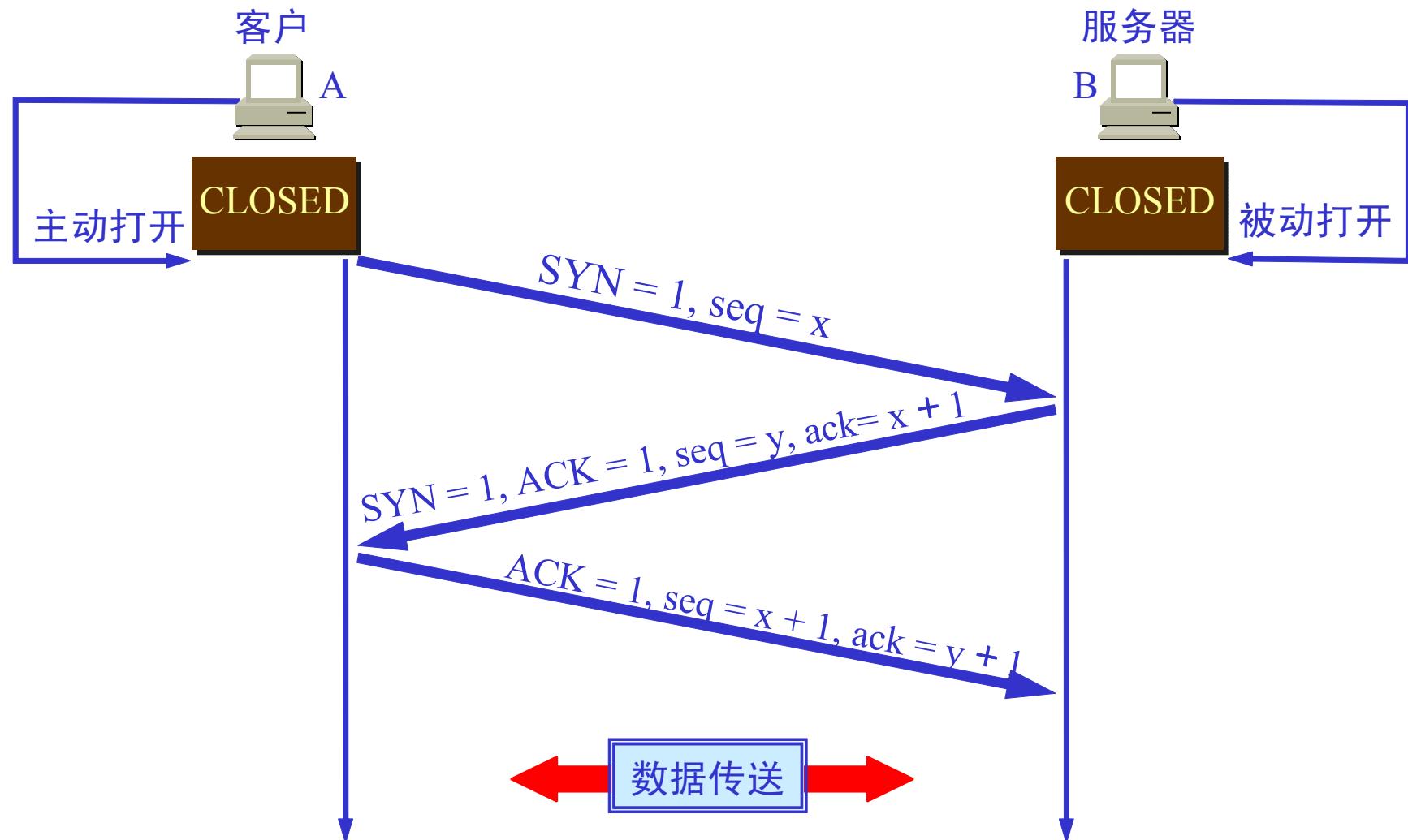


- B 的 TCP 收到连接请求报文段后，如同意，则发回确认。
- B 在确认报文段中应使 $SYN = 1$ ，使 $ACK = 1$ ，其确认号 $ack = x + 1$ ，自己选择的序号 $seq = y$ 。

- 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。

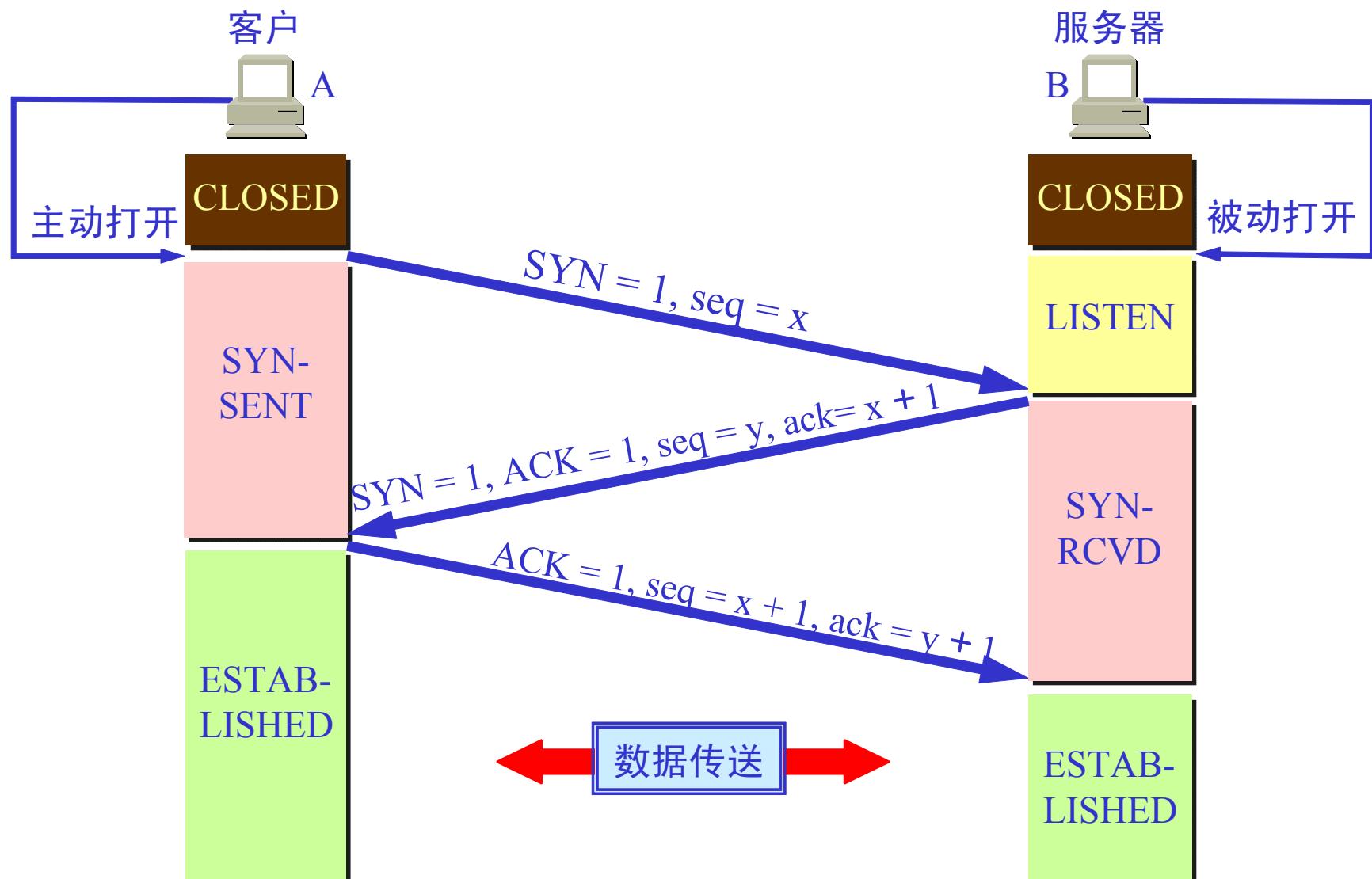


- B 的 TCP 收到主机 A 的确认后，也通知其上层应用进程：TCP 连接已经建立。

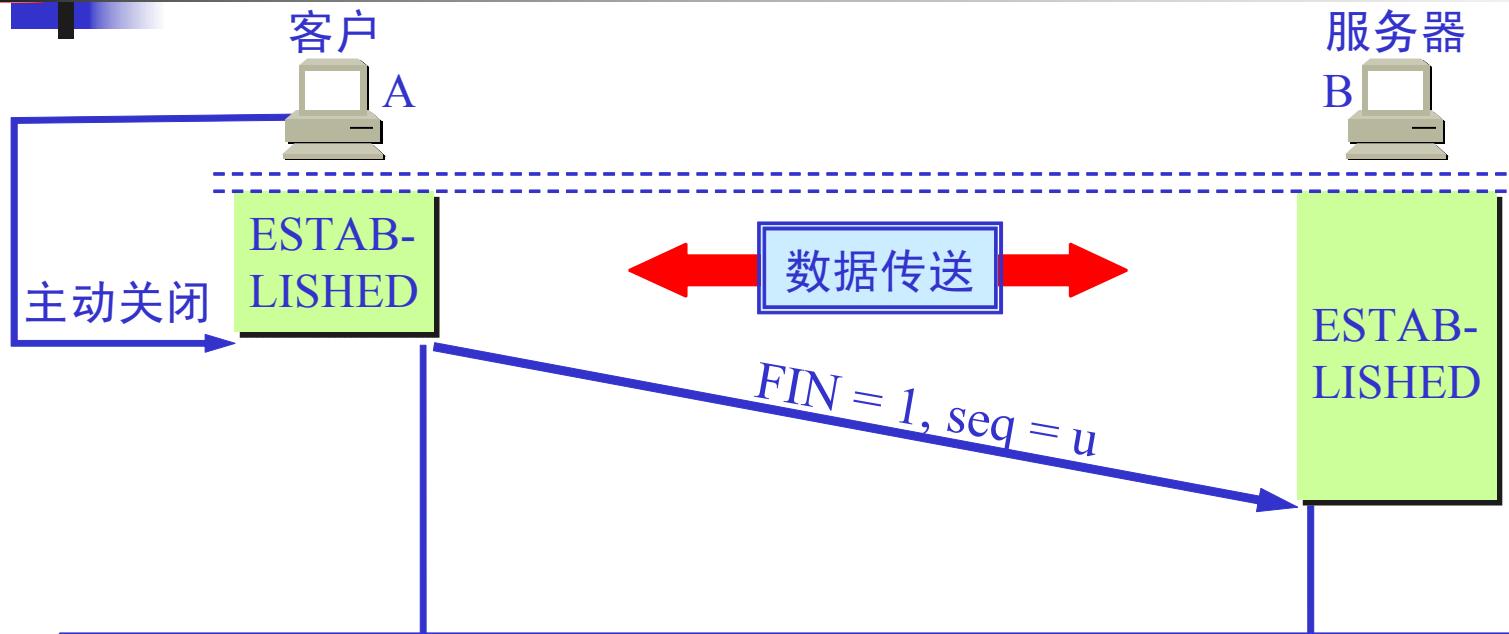


5.9.1 TCP 的连接建立

用三次握手建立 TCP 连接的各状态



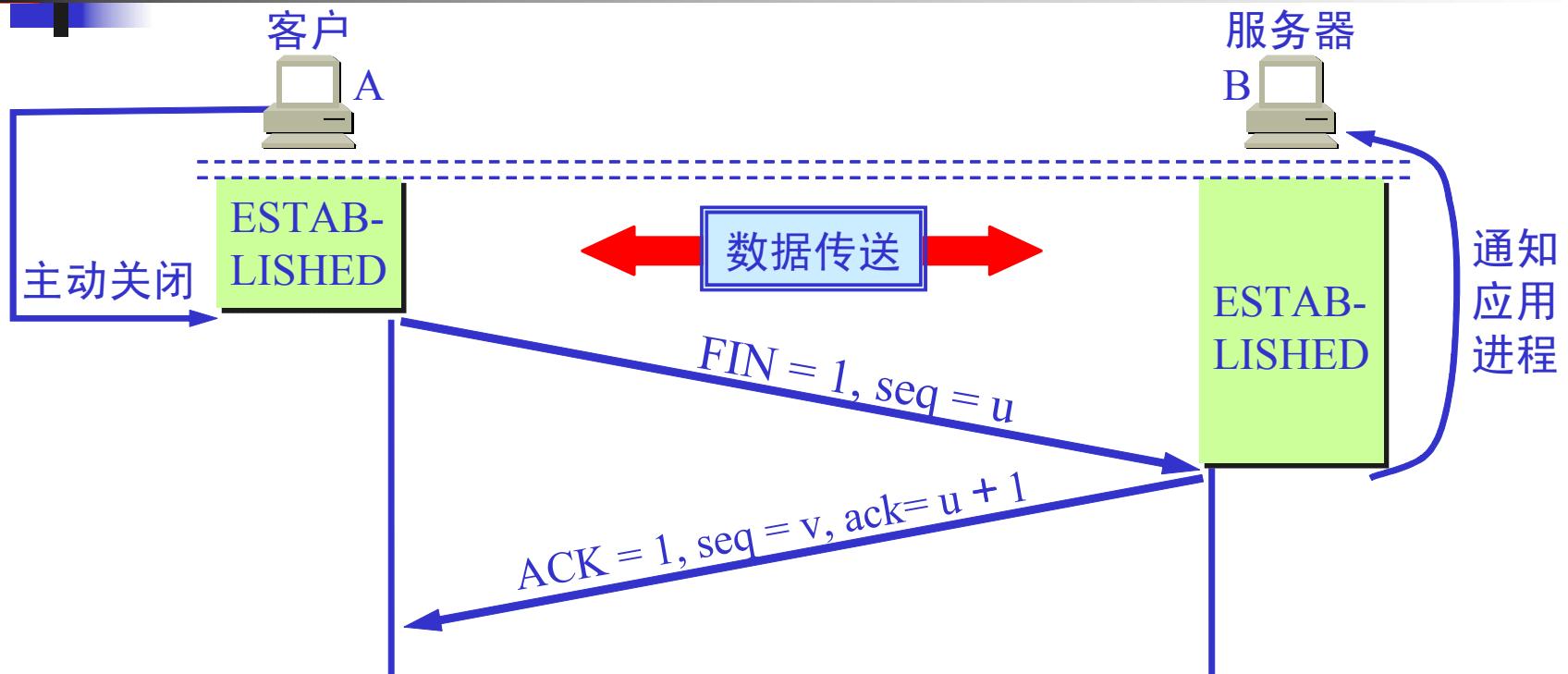
5.9.2 TCP 的连接释放



- 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 把连接释放报文段头部的 $FIN = 1$ ，其序号 $seq = u$ ，等待 B 的确认。

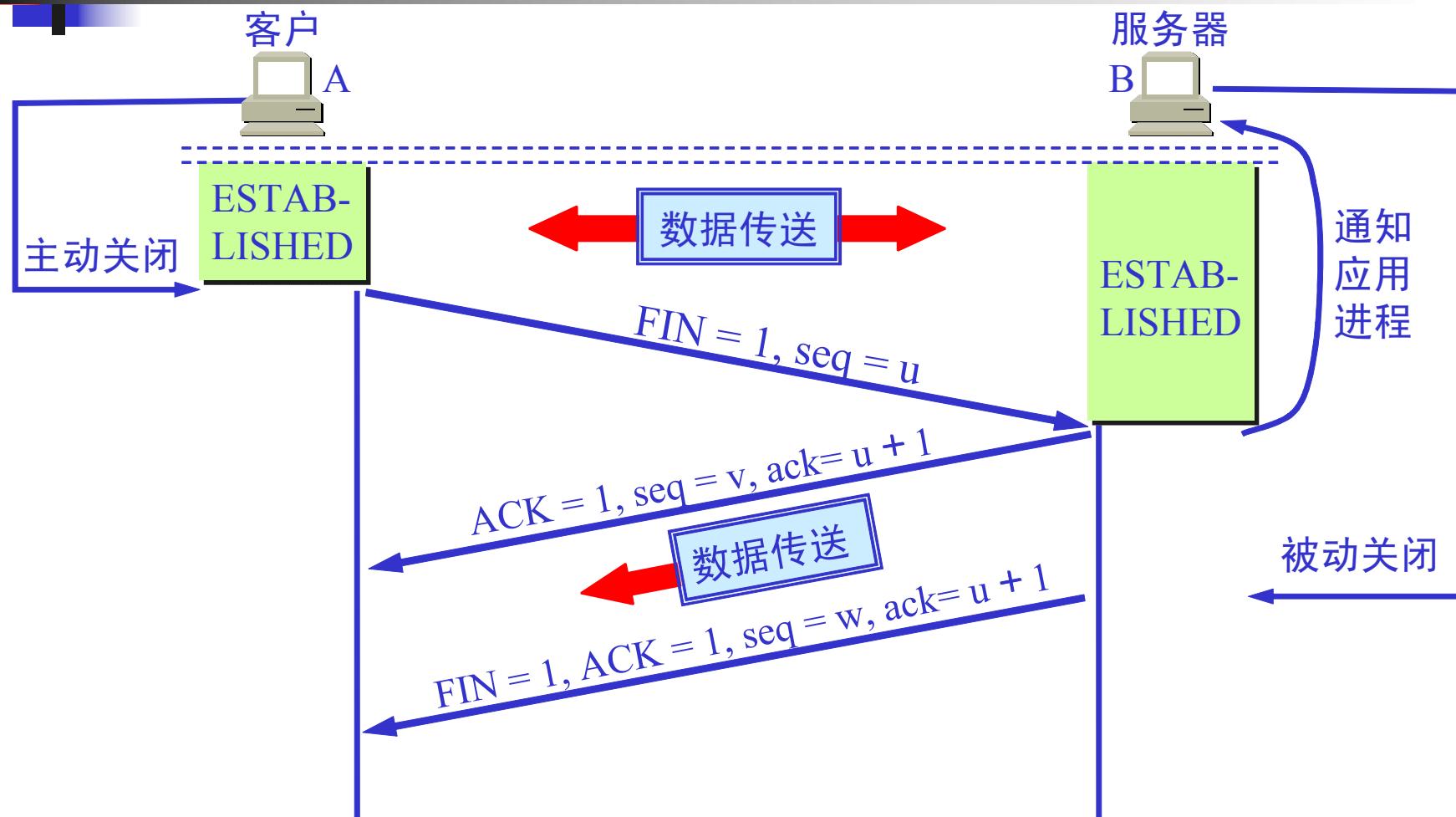
CLOSED

5.9.2 TCP 的连接释放



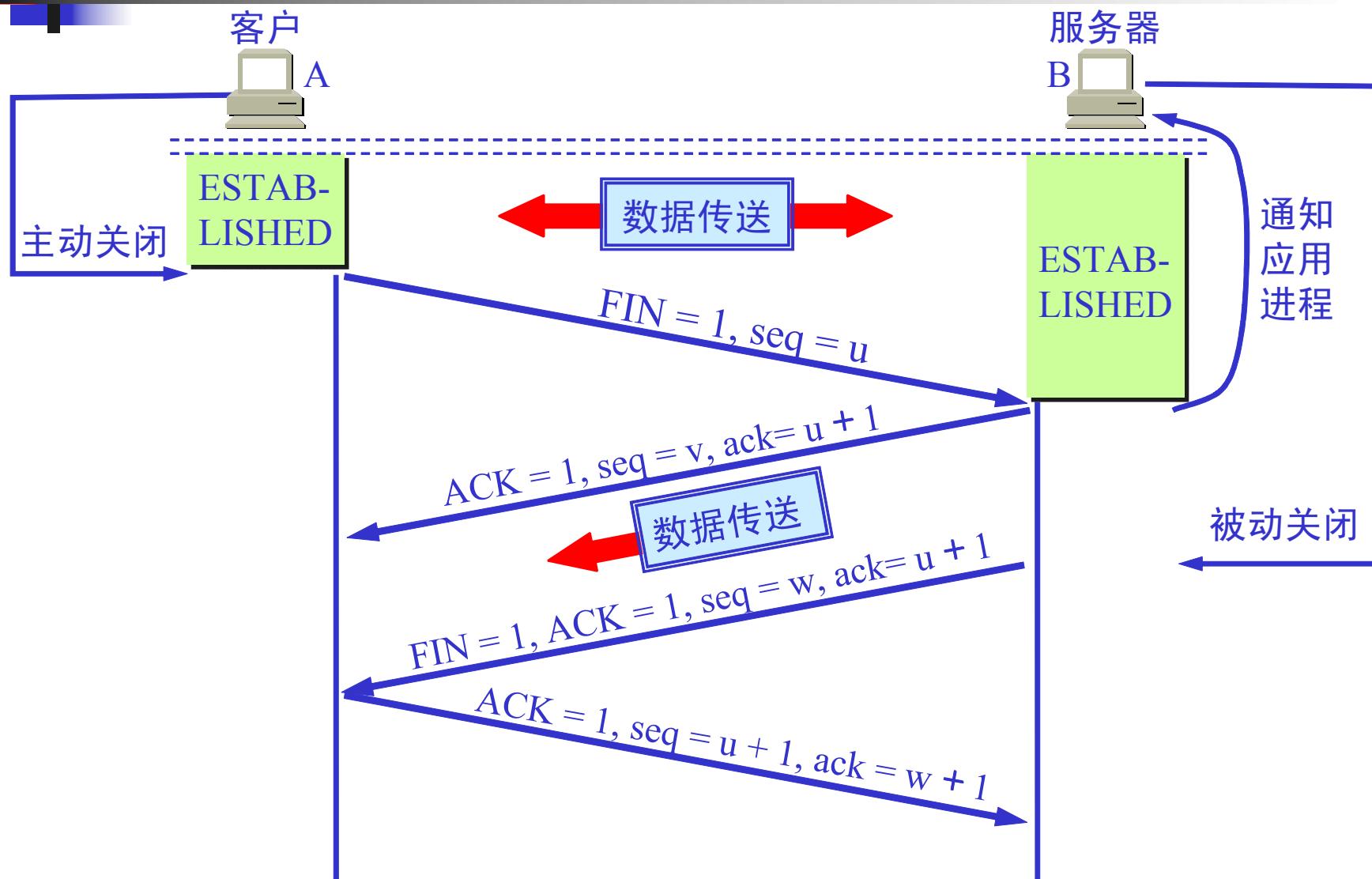
- B 发出确认，确认号 $ack = u + 1$ ，而这个报文段自己的序号 $seq = v$ 。
- TCP 服务器进程通知高层应用进程。
- 从 A 到 B 这个方向的连接就释放了，TCP 连接处于半关闭状态。B 若发送数据，A 仍要接收。

5.9.2 TCP 的连接释放



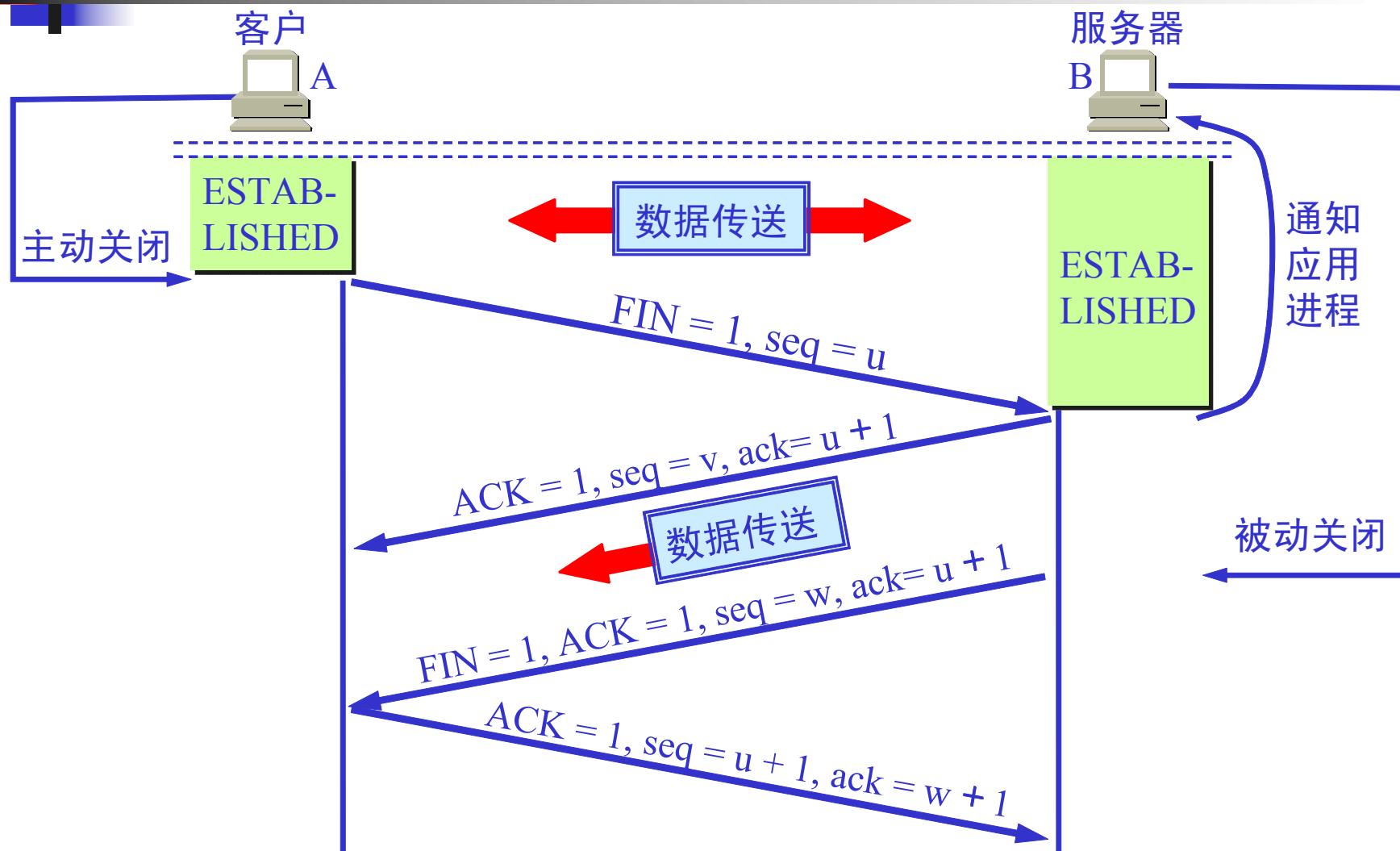
- 若 B 已经没有要向 A 发送的数据，其应用进程就通知 TCP 释放连接。

5.9.2 TCP 的连接释放



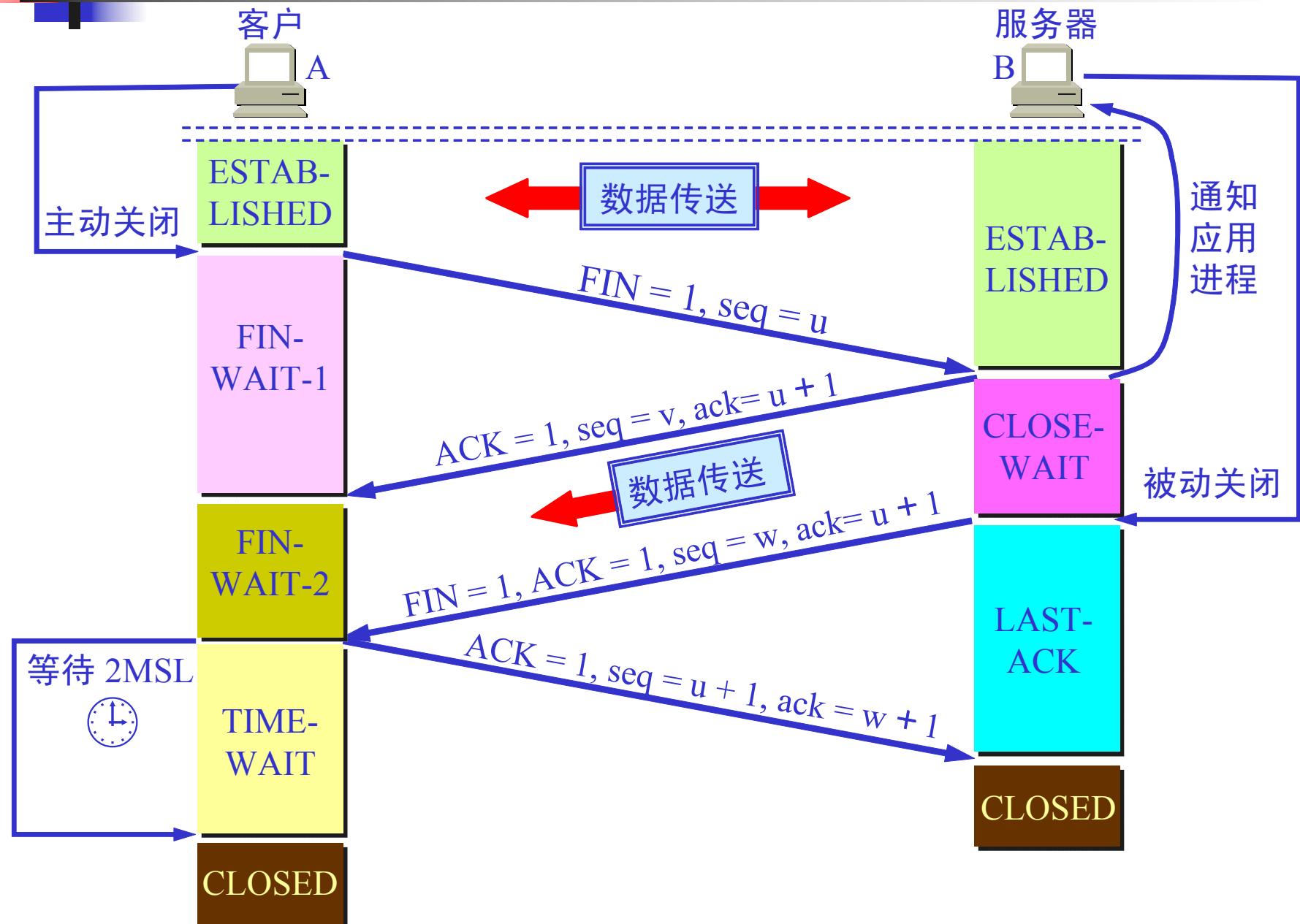
- A 收到连接释放报文段后，必须发出确认。

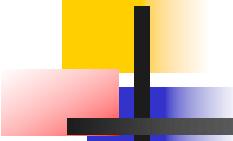
5.9.2 TCP 的连接释放



- 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。



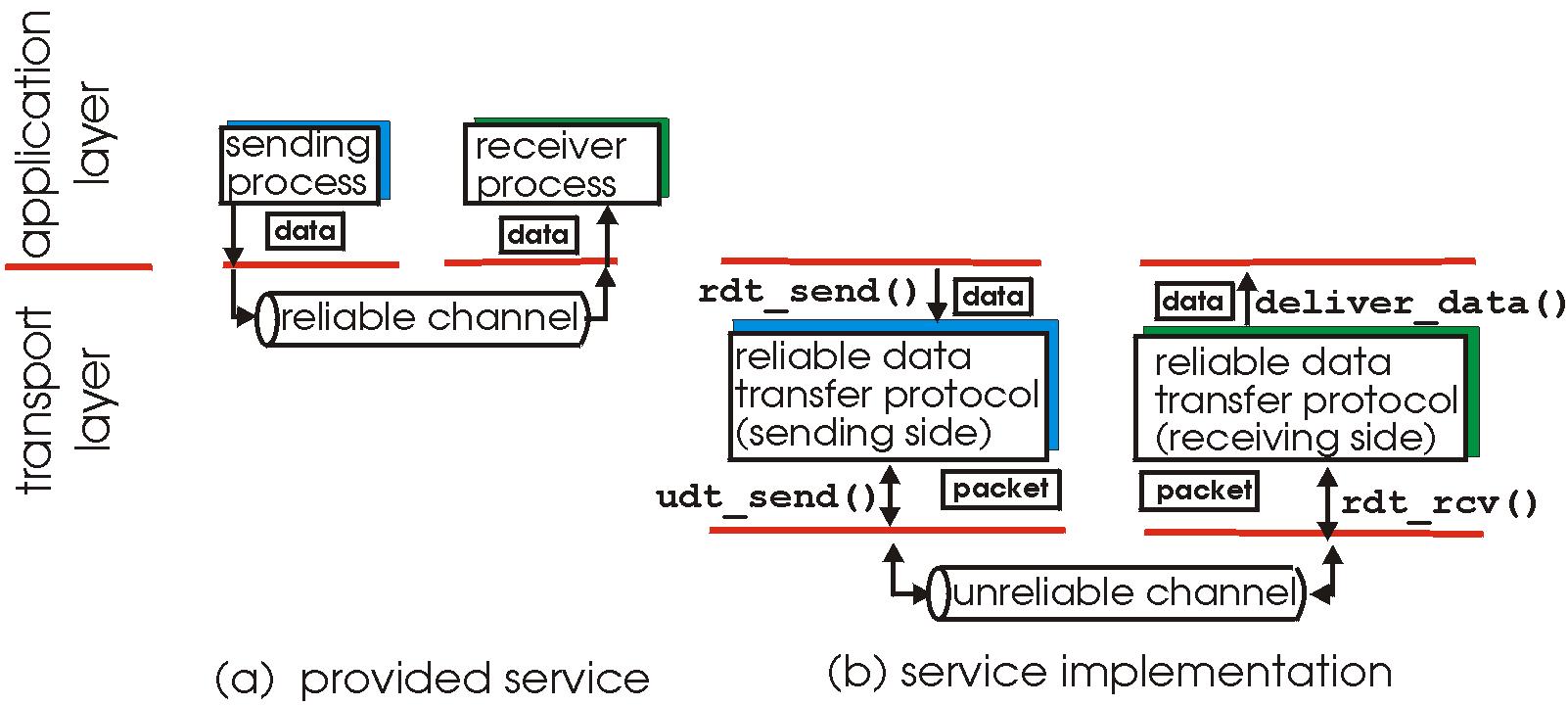


A 必须等待 2MSL 的时间

- MSL：最长报文段寿命，规定为2分钟。
- 为了保证 A 发送的最后一个 ACK 报文段能够到达 B。
- 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。

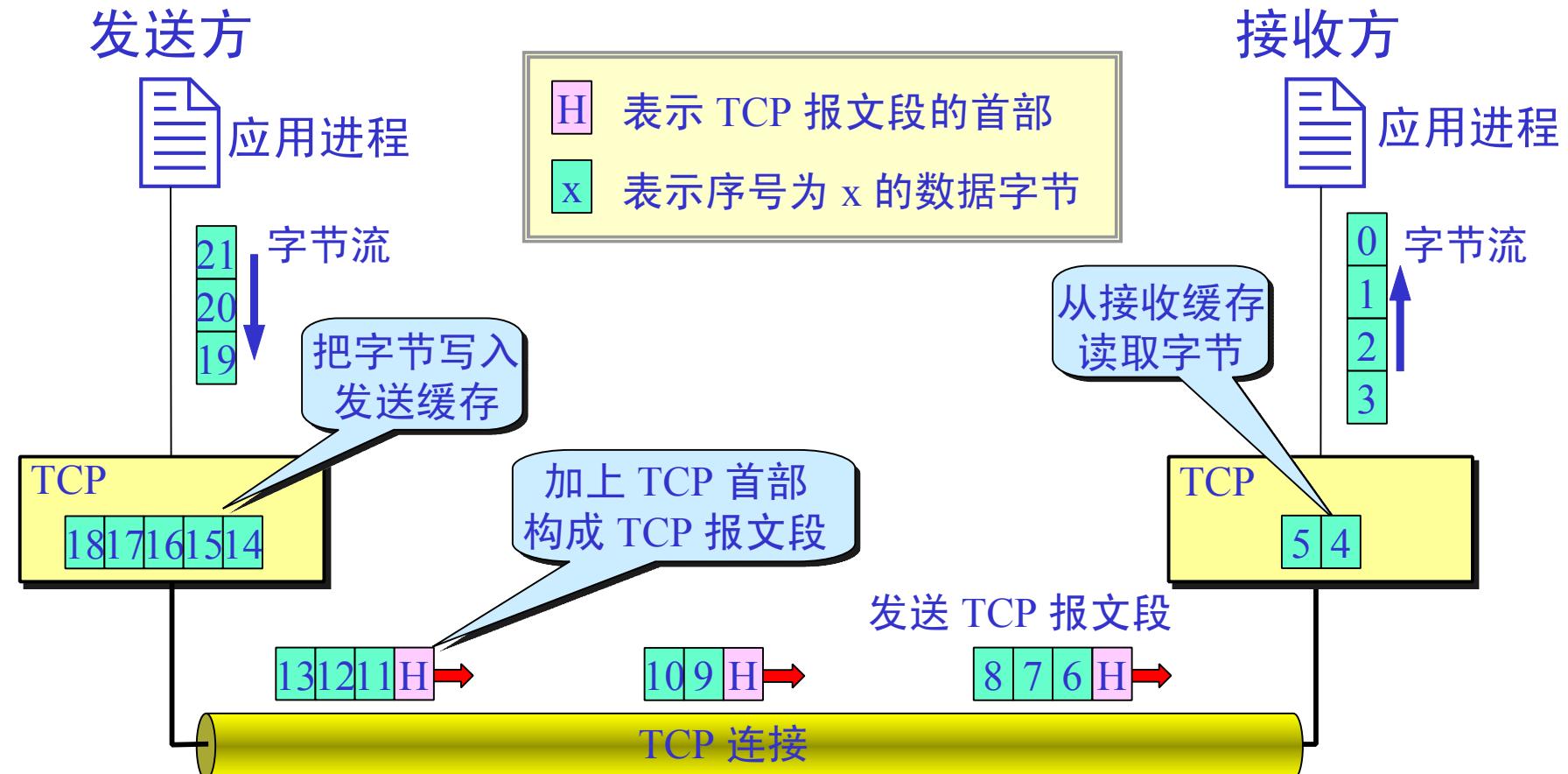
5.4 可靠传输的工作原理

- 在应用、传输、链路层都十分重要，属于网络工程的top-10 课题之一！



- 不可靠传输通道的特性将决定可靠传输协议(RDT)的复杂性

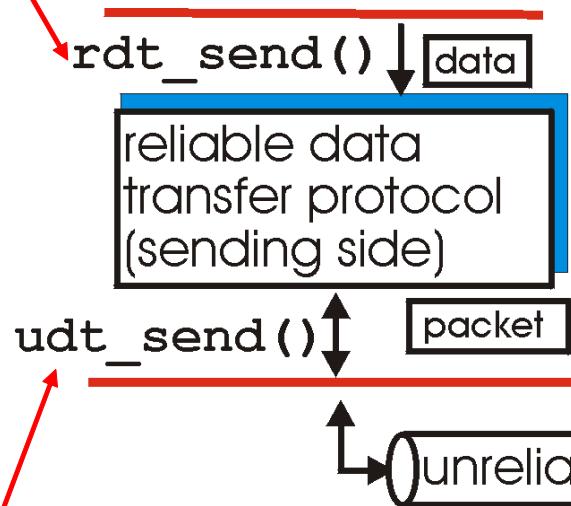
TCP 实现可靠的字节流



可靠数据传输：信道模型

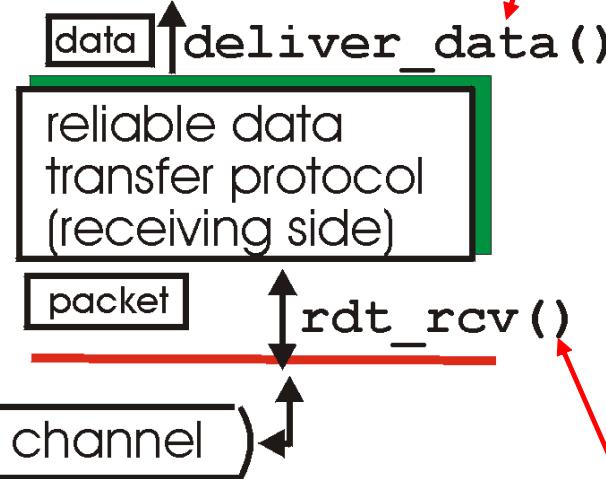
rdt_send(): 由上层进行调用 (如应用进程). 将数据传入发送方并由其传给接收方的上层

发送方



deliver_data(): 由 rdt 调用
将数据递交给上层

接收方



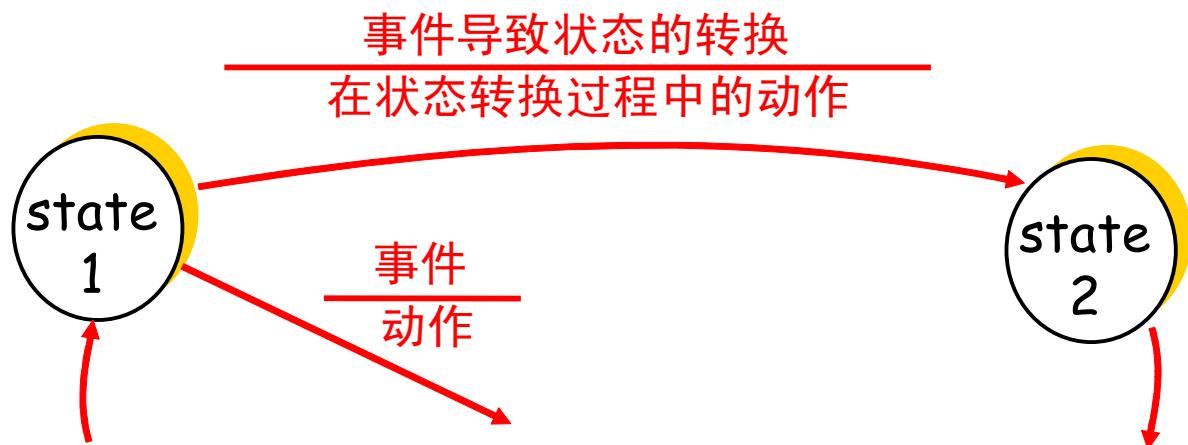
udt_send(): 由 rdt 调用,
将分组通过不可靠的信道传到接
收方

rdt_rcv(): 当分组到达接收方时调用

可靠数据传输: 状态模型

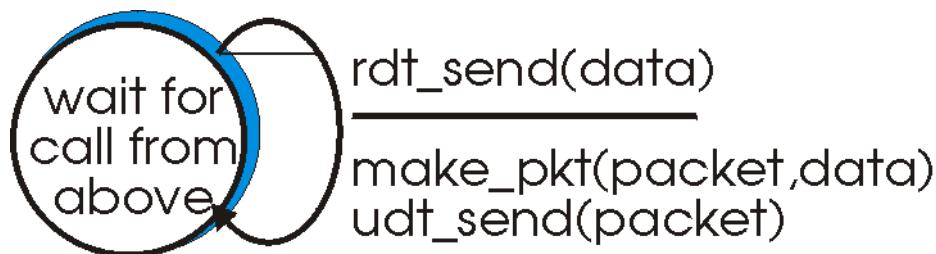
- 逐步发展收发双方的可靠数据传输协议 (**RDT**)
- 仅考虑单向的数据传输
 - 但控制信息将双向流动!
- 使用有限状态机 (**FSM**)来定义发送方、接收方

状态: 当实体处于某个“状态”时，下个状态只能由下个事件来转变

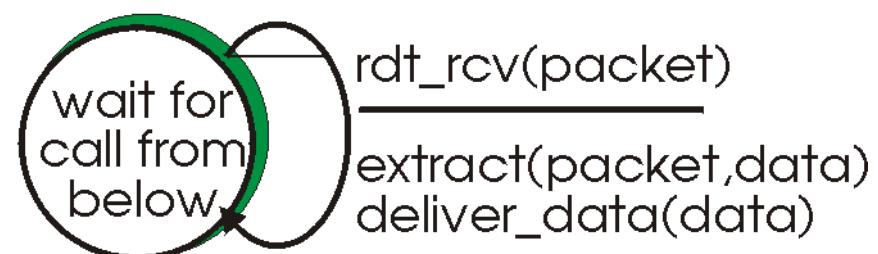


RDT1.0: 在可靠信道上进行可靠的数据传输

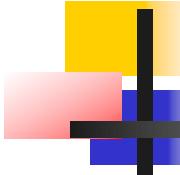
- 所依赖的信道非常可靠
 - 不可能有位错
 - 不会丢失数据
- 分别为发送方和接收方建立 FSMs
 - 发送方将数据送入所依赖的信道
 - 接收方从所依赖的信道读出数据



(a) rdt1.0: sending side



(b) rdt1.0: receiving side



RDT2.0: 在可能出现位错的信道上传输

- 所依赖的信道有可能在分组数据中出现位错
 - 如何发现位错（如UDP）？
- 问题: 如何从错误中恢复?
 - 进行确认 (**ACKs**): 由接收方法送报文向发送方进行确认
 - 发送否认 (**NAKs**): 由接收方法送报文向发送方进行否认，说明分组有错
 - 发送方在收到NAK后进行分组重传
 - 在人类交往中是不是也有 **ACKs, NAKs?**
- RDT2.0的新机制 (在 RDT1.0基础之上):
 - 错误检测
 - 接收方的反馈: 控制信息 (ACK,NAK) rcvr->sender

RDT2.0: 有限状态机定义

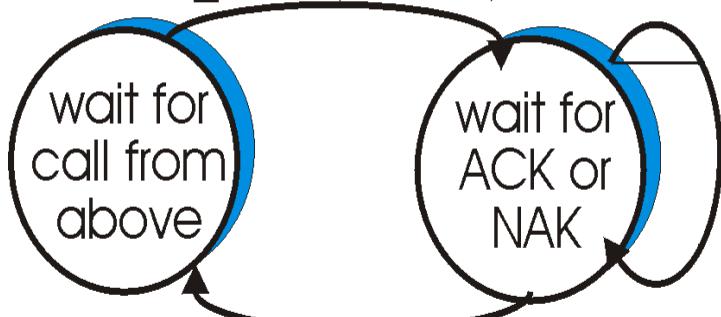
发送方FSM

rdt_send(data)

compute checksum

make_pkt(sndpkt, data, checksum)

udt_send(sndpkt)



rdt_rcv(rcvpkt)
&& isACK(rcvpkt)

rdt_rcv(rcvpkt) && isNACK(rcvpkt)

udt_send(sndpkt)

接收方FSM

rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)

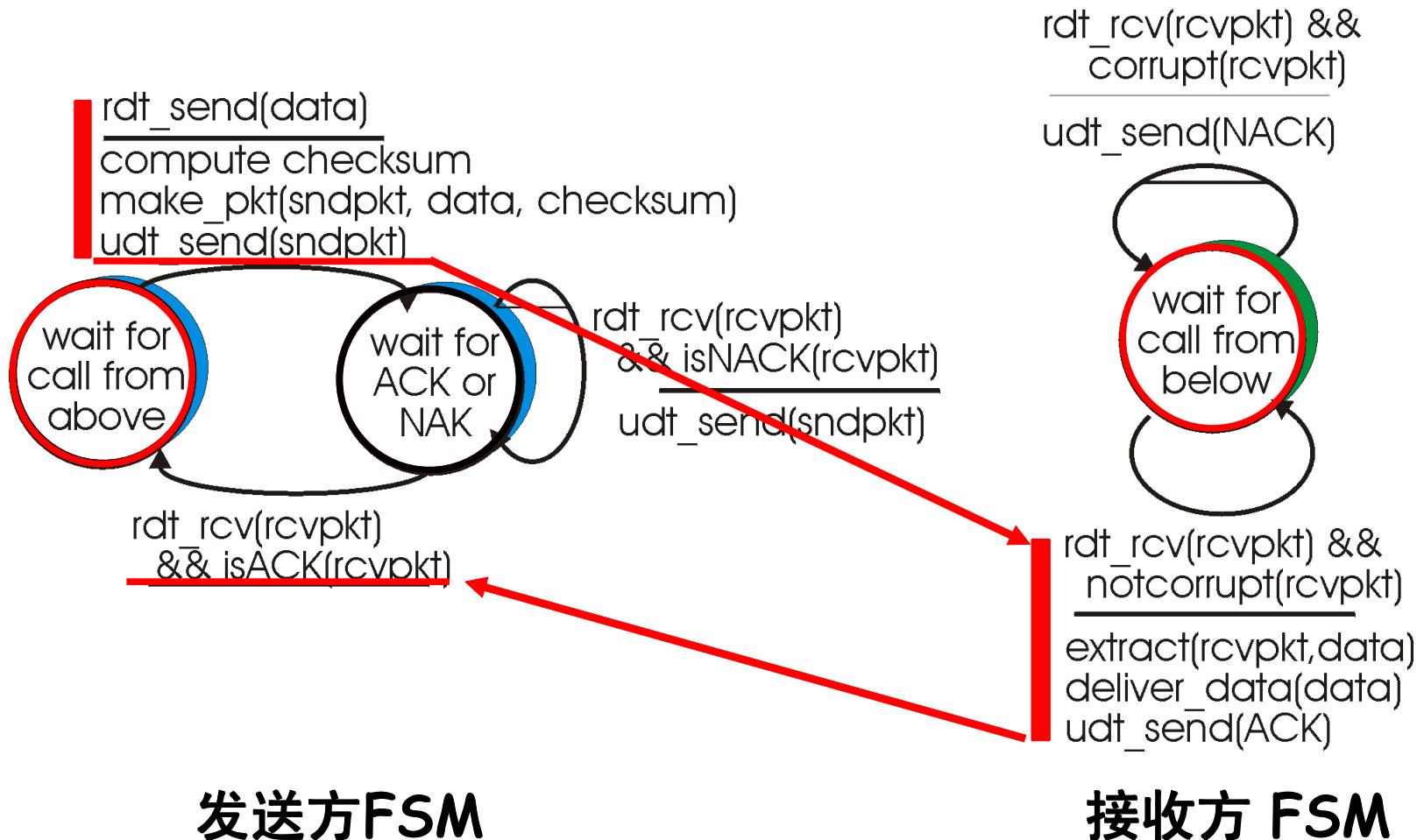
udt_send(NACK)



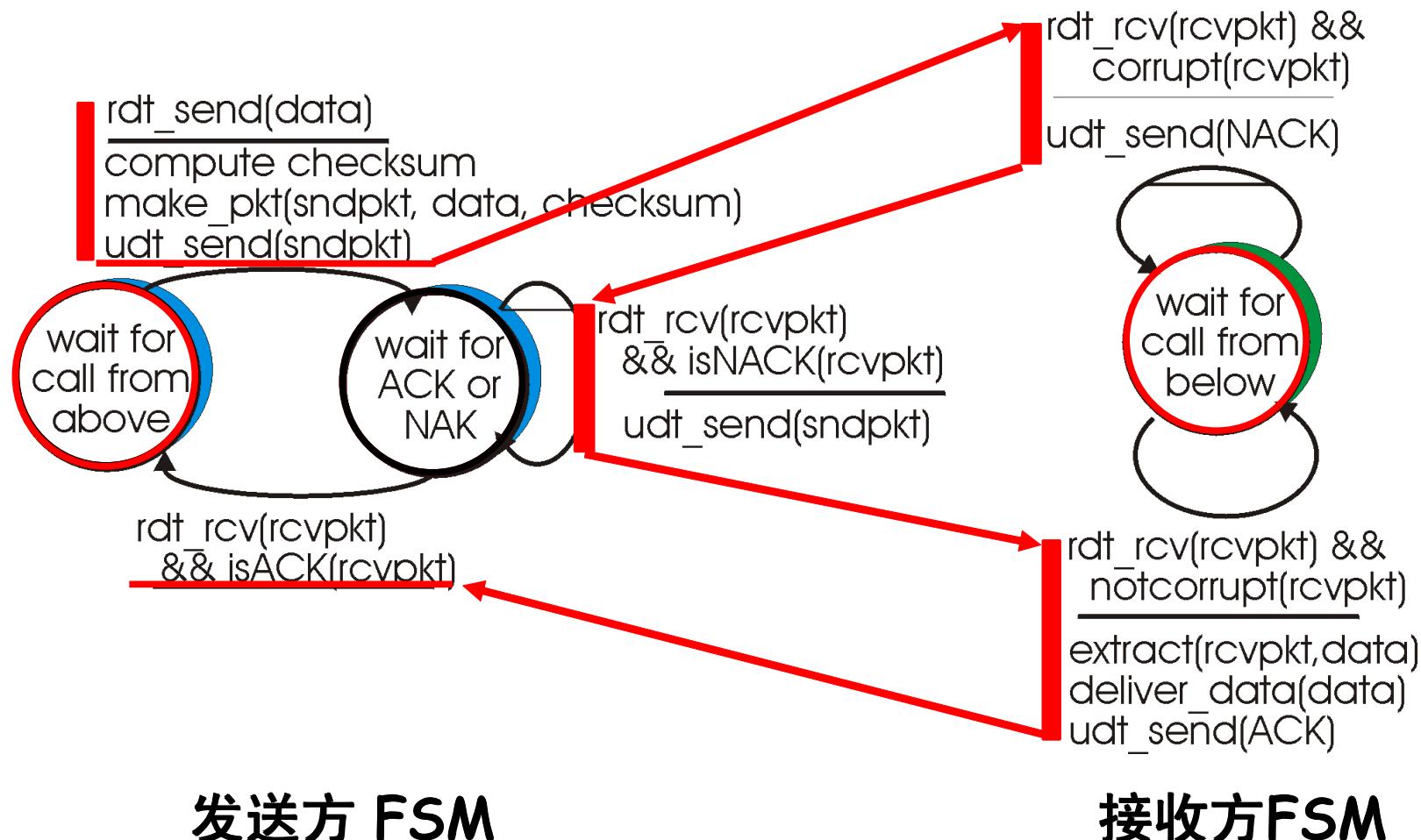
rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

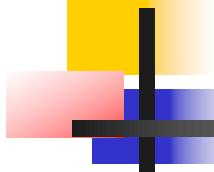
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

RDT2.0: 运行过程 (未发现错误)



rdt2.0:运行过程 (出错情况)





RDT2.0 有一个致命的缺点！

若ACK/NAK 报文出错？

- 发送方将不会知道接收端发生了什么！
- 假如进行重传：可能发生数据重复

怎么办？

- 发送 ACK/NAK 来回应接收方的 ACK/NAK？
 - 那么如果发送方的 ACK/NAK 出错？蓝白困境
- 重传，但可能导致重传了正确的分组！

管理重复的问题：

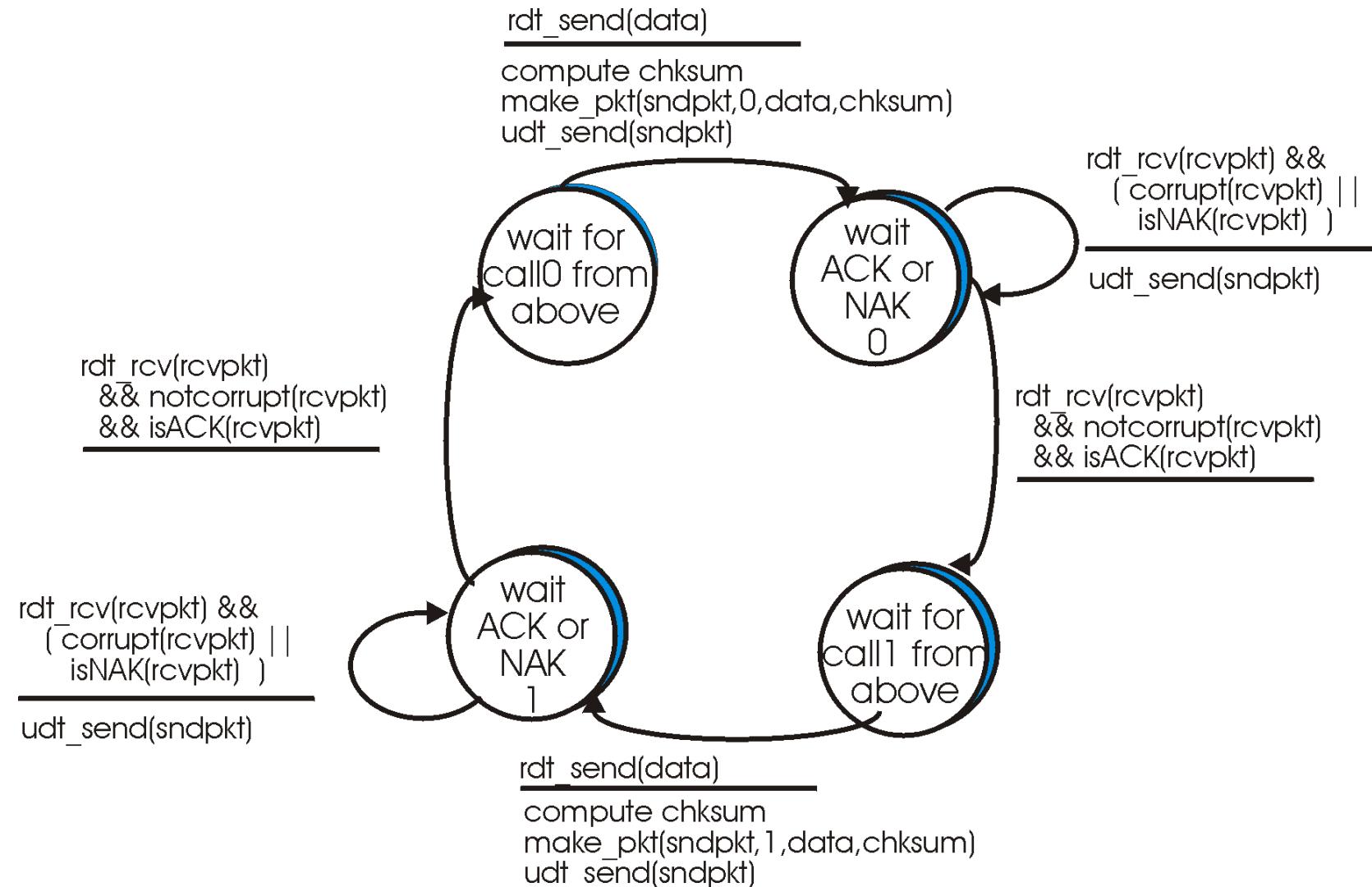
- 发送方给每个分组加上 sequence number（序号）
- 如果ACK/NAK丢失出错，发送方则重传当前分组
- 接收方丢弃重复的分组（不向上递交）

停等策略

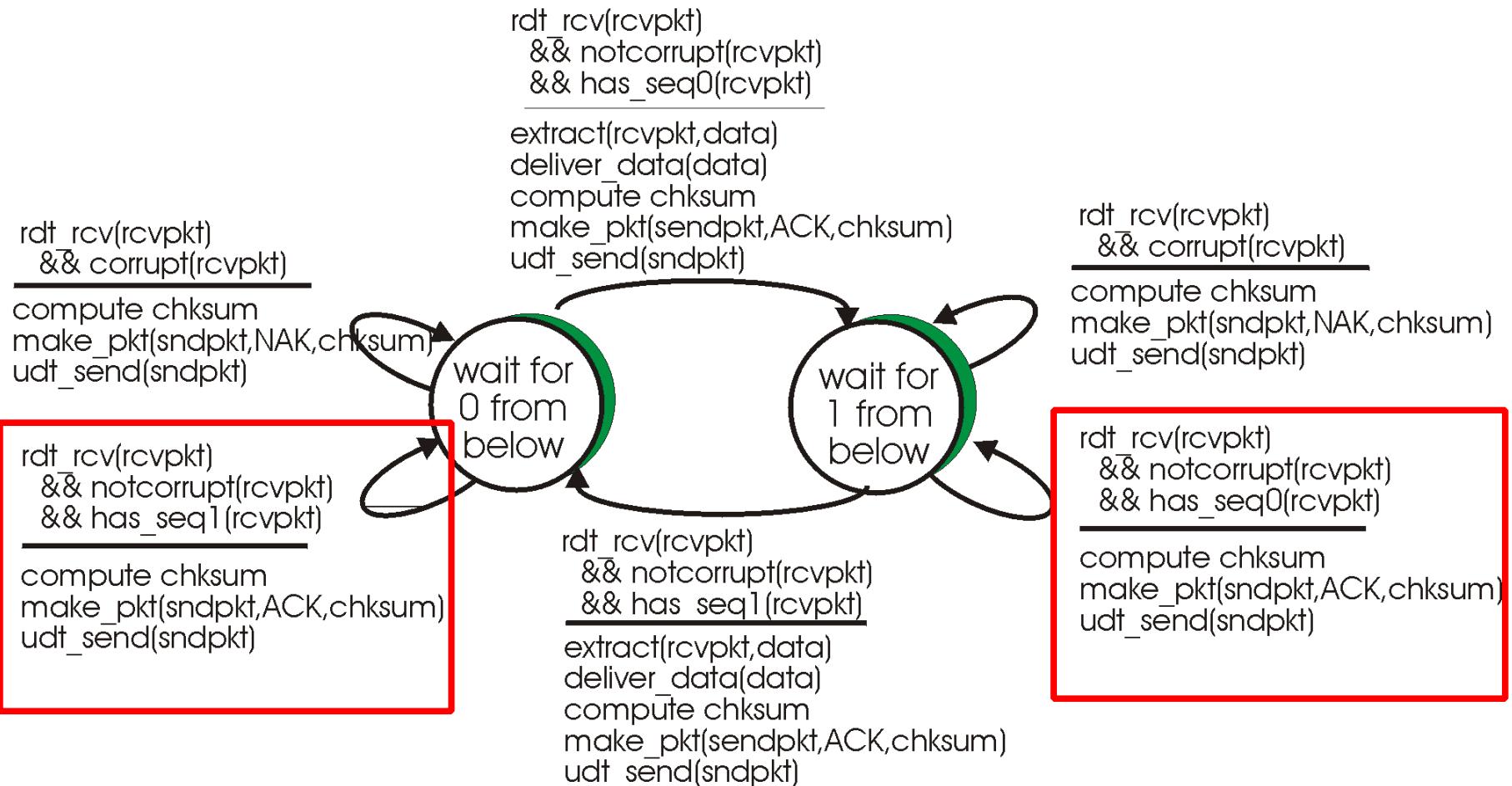
发送方法送一个分组，然后等待接收方的响应

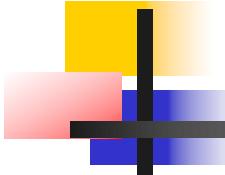


RDT2.1: 管理出错的ACK/NAK, 发送方



RDT2.1: 管理出错的ACK/NAK, 接收方





RDT2.1: 讨论

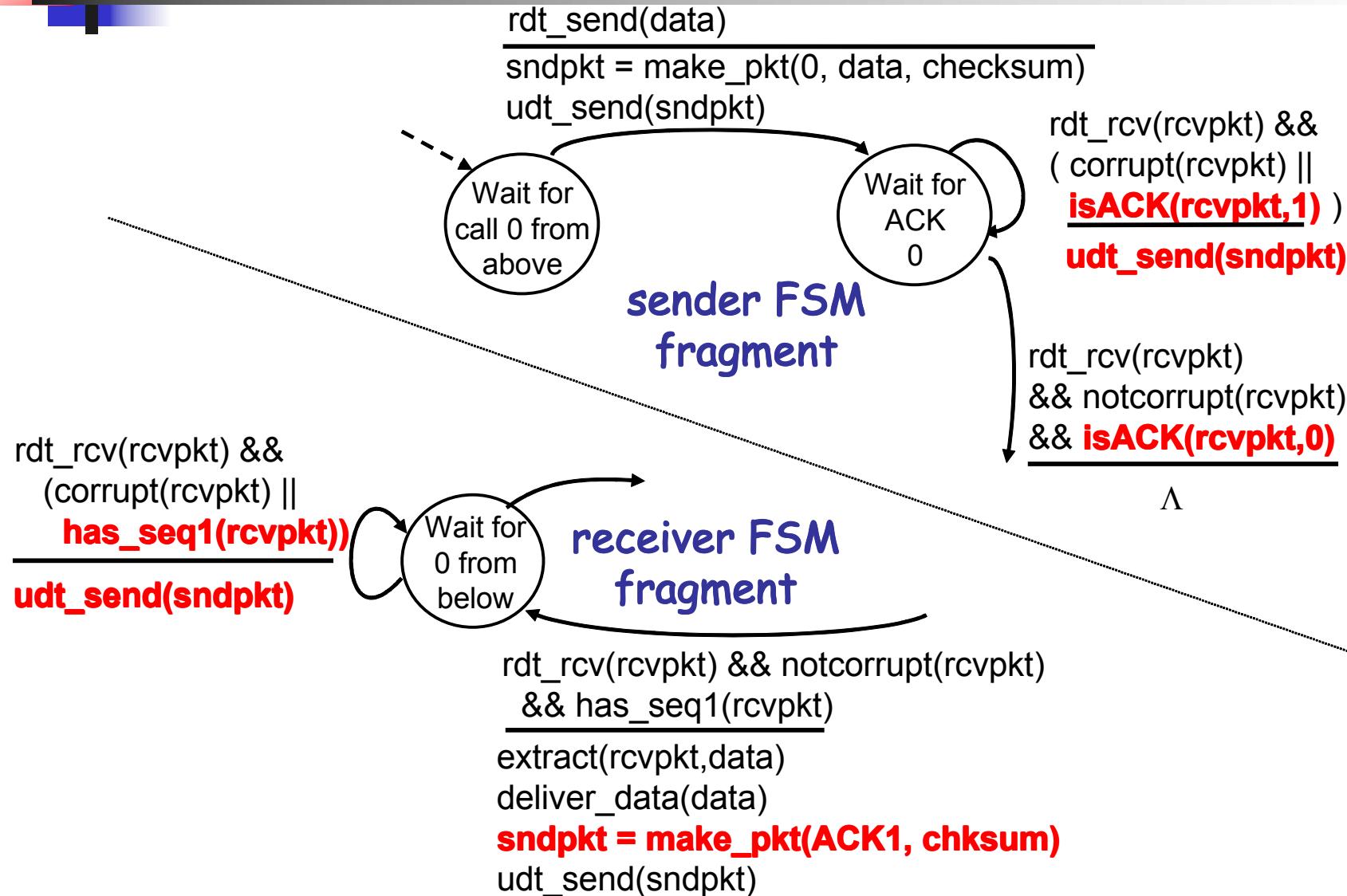
发送方:

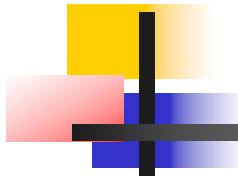
- 给分组加seq. #,
- 两个 seq. # (0,1) 够么? 为什么?
- 必须查收ACK/NAK 是否出错
- 两倍的状态
 - 必须“记忆”状态, 是否当前分组具有 0 或 1 seq. #

接收方:

- 必须查验接收到的分组是否重复
 - 状态可以指出0 或 1 是期望中的seq. #
- 注意: 接收方不会知道最后的ACK/NAK 是否为发送方正确接收

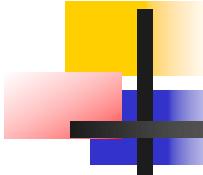
RDT2.2: 无 NAK的协议





RDT2.2讨论

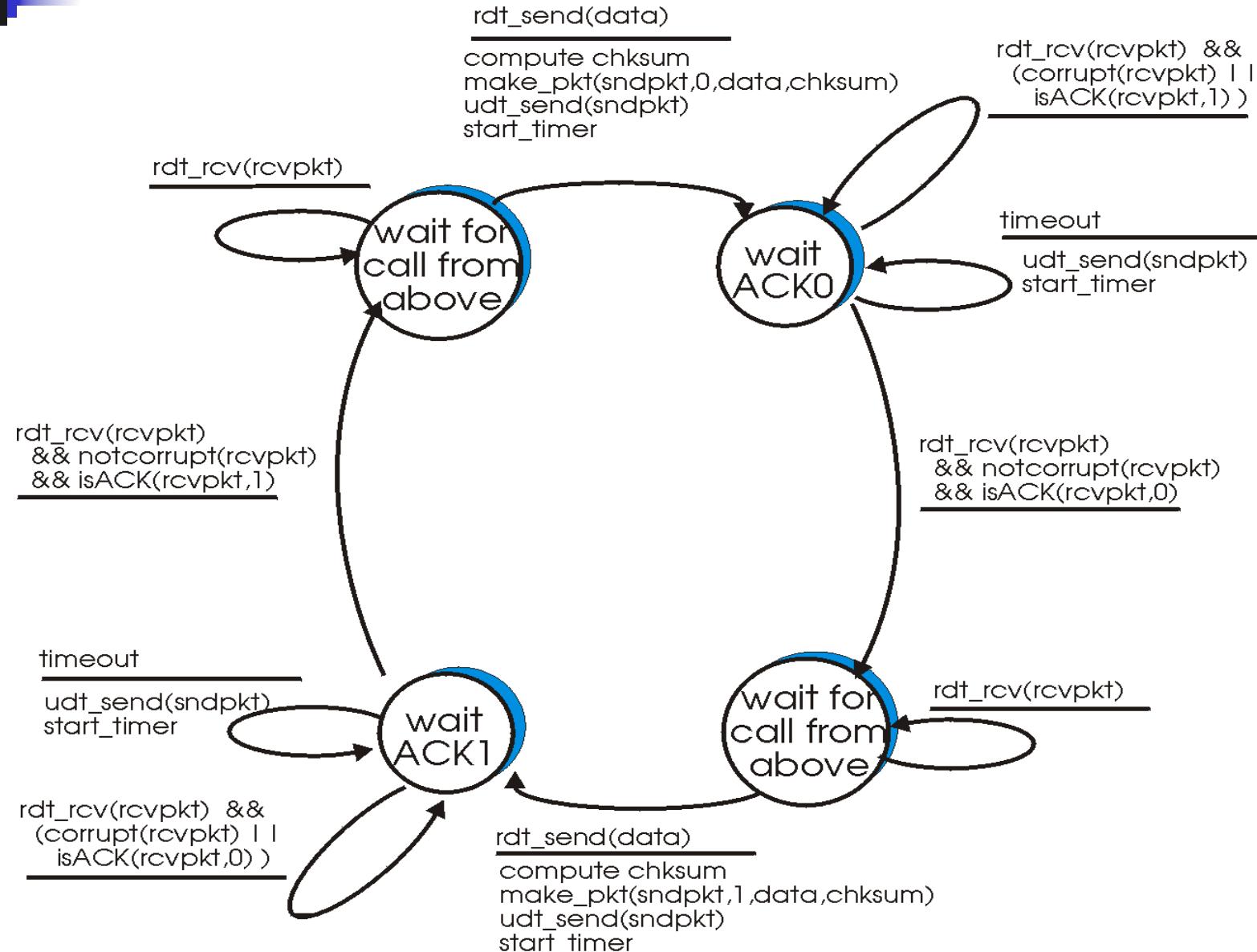
- 和RDT2.1的功能相同，但仅使用ACK
 - 接收方正确接收一个包后，发送ACK
 - 在ACK包中，接收方必须通过序号指明是对哪个数据包的确认
- 重复的ACK包对发送方来说，和收到NACK的效果一样
 - 重发当前数据包



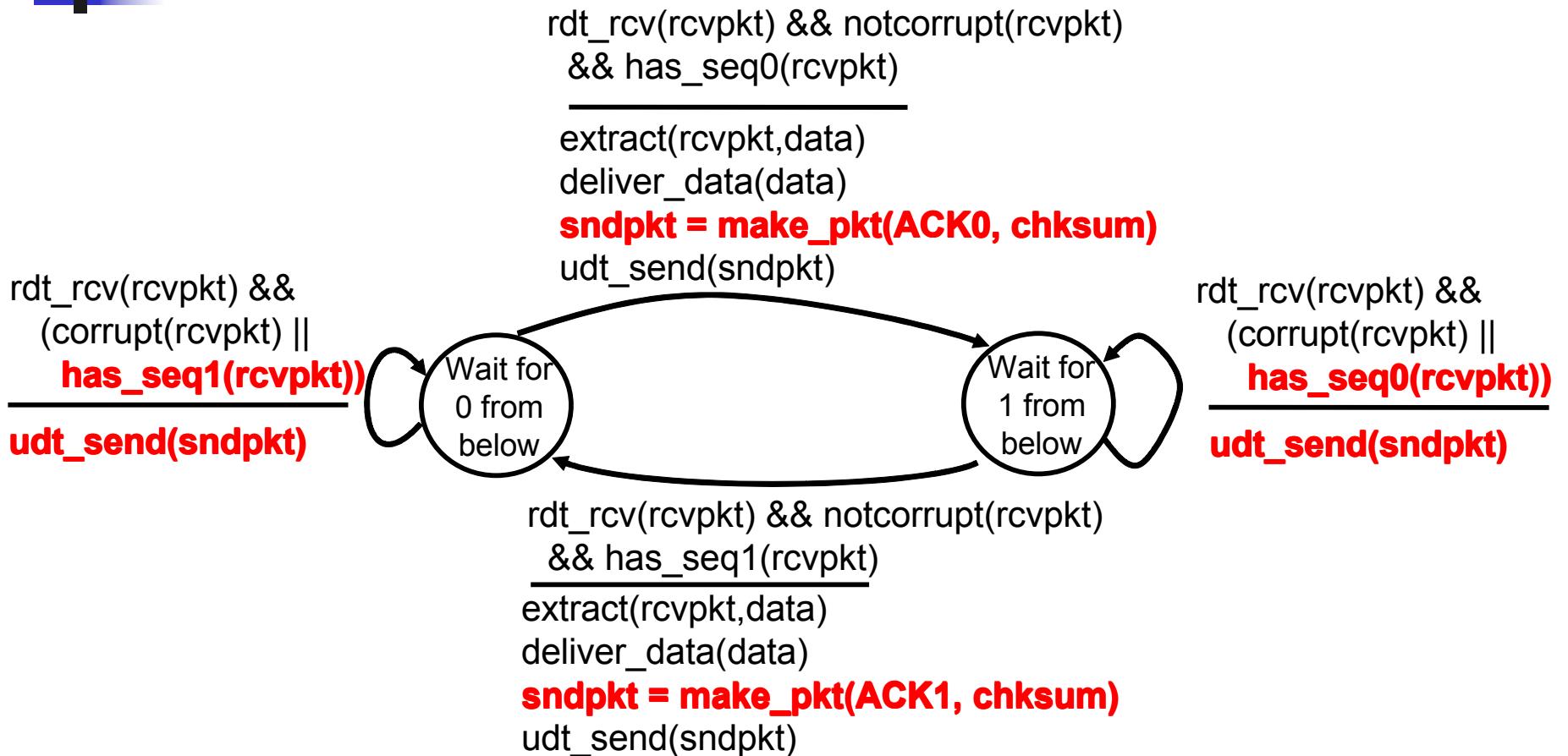
RDT3.0: 通道上可能出错和丢失数据

- 新的假设: 底层信道可能丢包(数据或ACK)
 - checksum, seq. #, ACK, 重发机制会有帮助
- 如何处理数据丢失?
 - 发送方可以等待, 当某些数据或ACK丢失时, 进行重传
 - 想一想: 问题?
- 方法: 发送方等待ACK一段“适当的”时间
 - 如果在这段时间里没有收到ACK, 则进行重传
 - 如果分组(或 ACK)仅仅被延迟了(没有丢失):
 - 重传将导致重复, 但使用seq. # 可以控制
 - 接收方必须指明被 ACK 分组所确认的 seq #
 - 需要进行倒计时

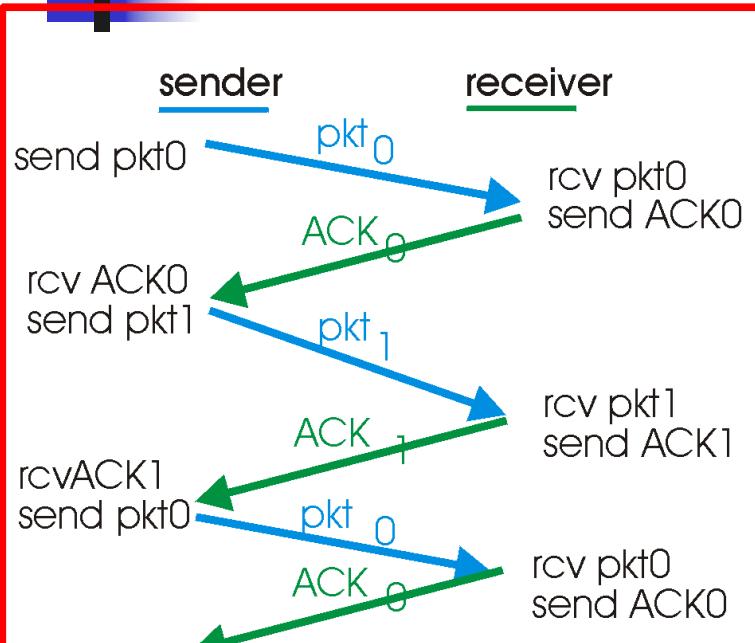
RDT3.0 发送方



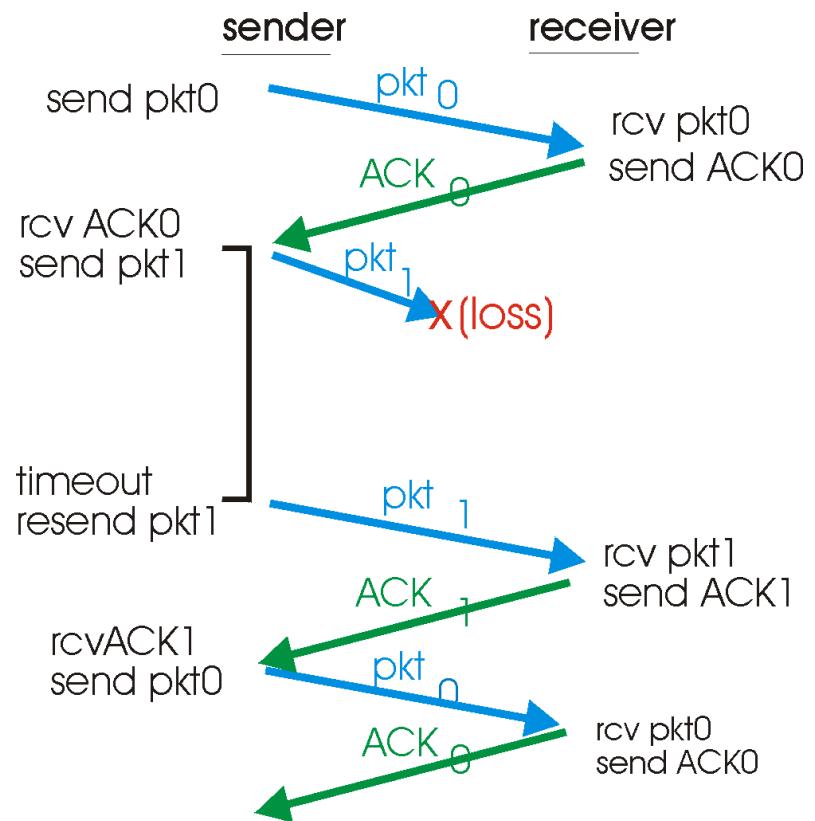
RDT3.0: 接收方



RDT3.0 的运行

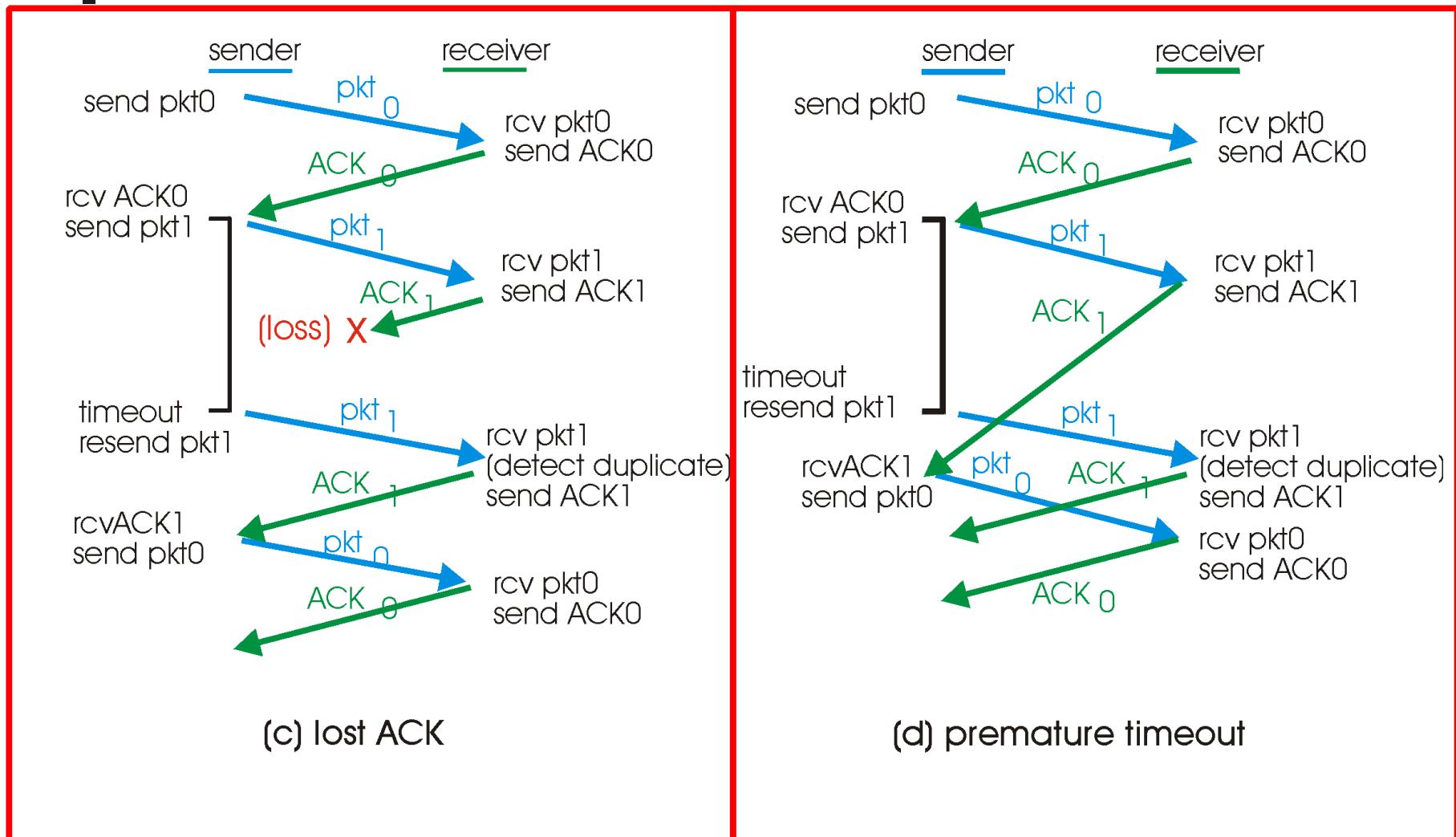


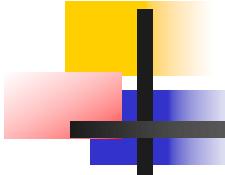
(a) operation with no loss



(b) lost packet

RDT3.0 的运行





RDT3.0的性能

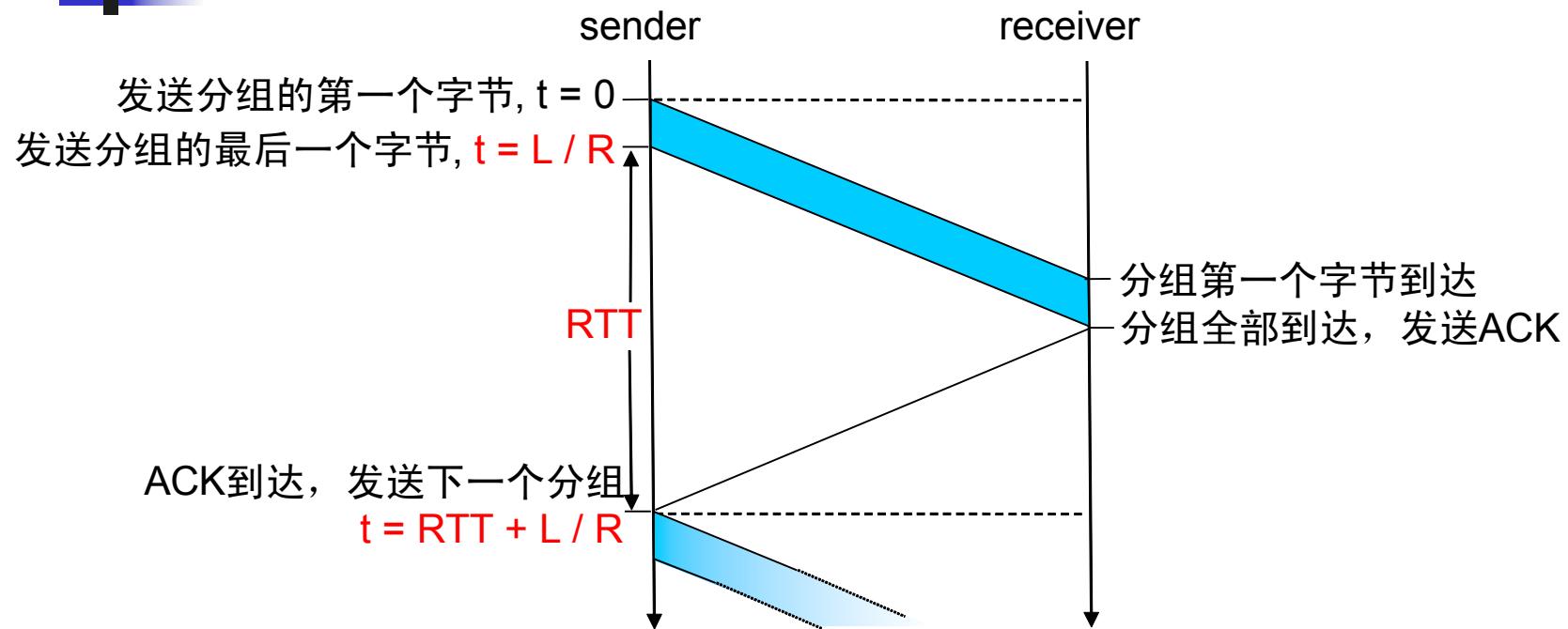
- RDT3.0 可靠传输, 但性能很糟
- 例如: 1 Gb/s 链路, 15 毫秒(millisecond) 端对端的延迟, 1KB 分组, 计算利用率

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

$$\text{利用率 } U = \frac{T_{transmit}}{T_{transmit} + RTT} = \frac{8 \text{ microsec}}{30.008 \text{ msec}} = 0.00027$$

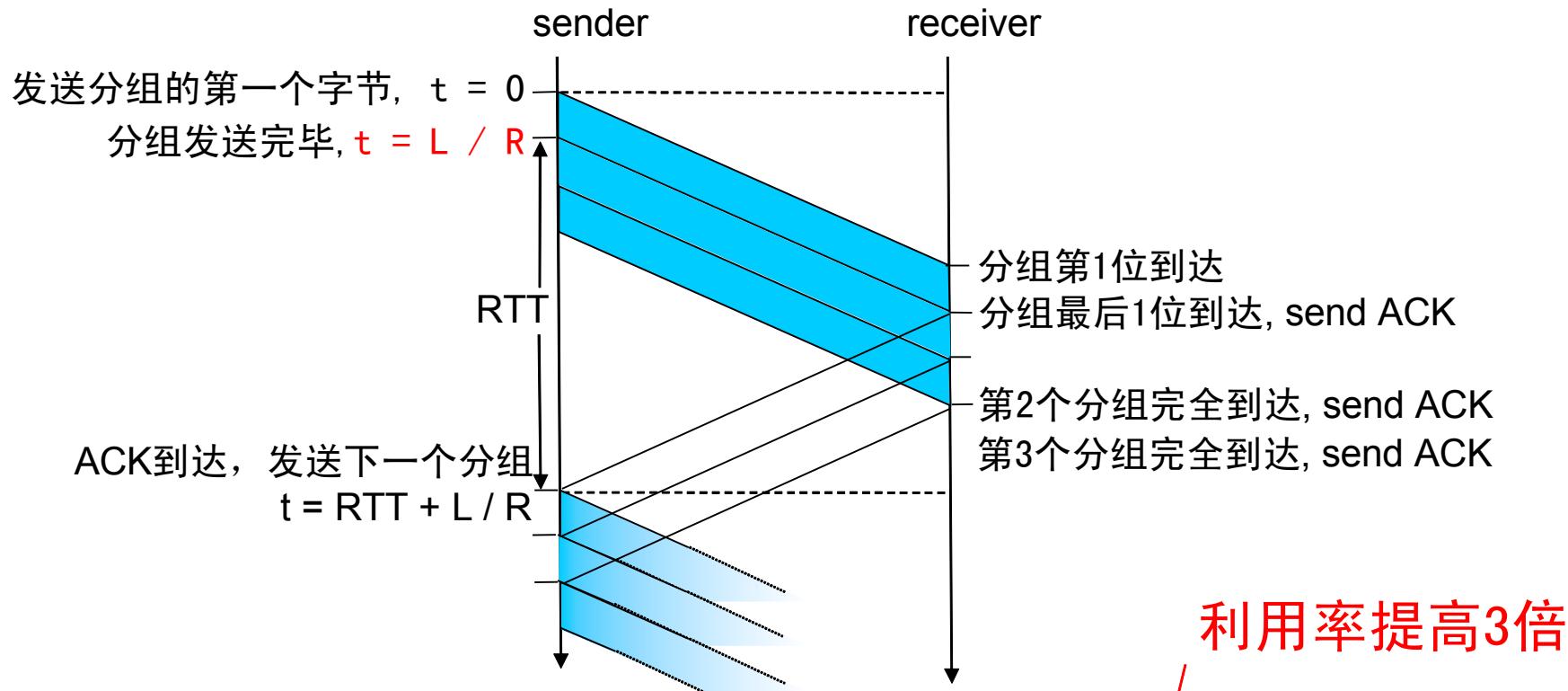
- 每 30 ms 发送1KB分组, 1s 发送 33个分组
 - 在1Gb/s 链路上的吞吐量仅为33kB/sec
- 网络协议限制了物理资源的利用!

RDT3.0: 停止等待协议



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

流水线：提高利用率

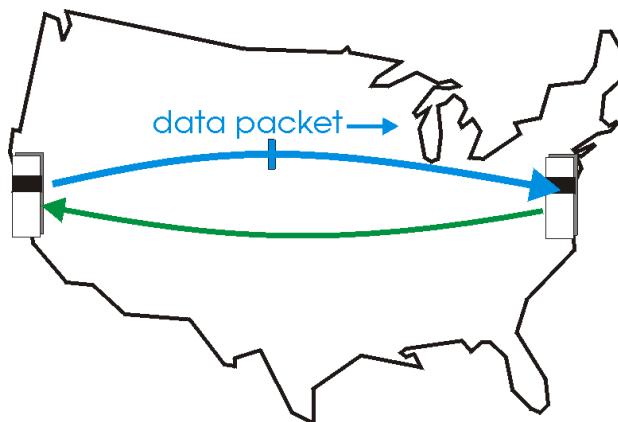


$$U_{\text{sender}} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

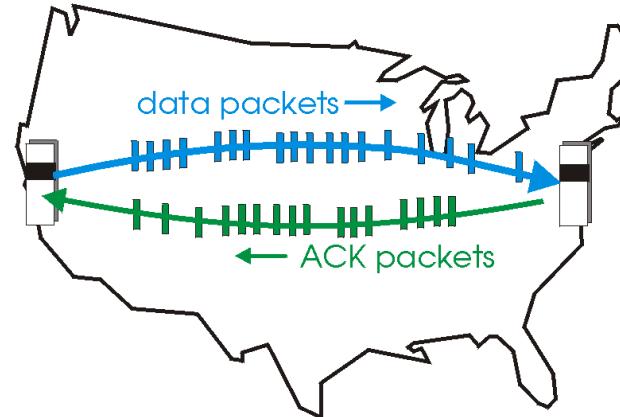
流水线协议

流水作业：发送端允许发送多个——“悬在空中”等待应答的分组

- 必须增加顺序号的位数
- 在发送和接收端增加缓存
- 第N个分组重发(**go-Back-N**)，选择应答 (**Select Response**)



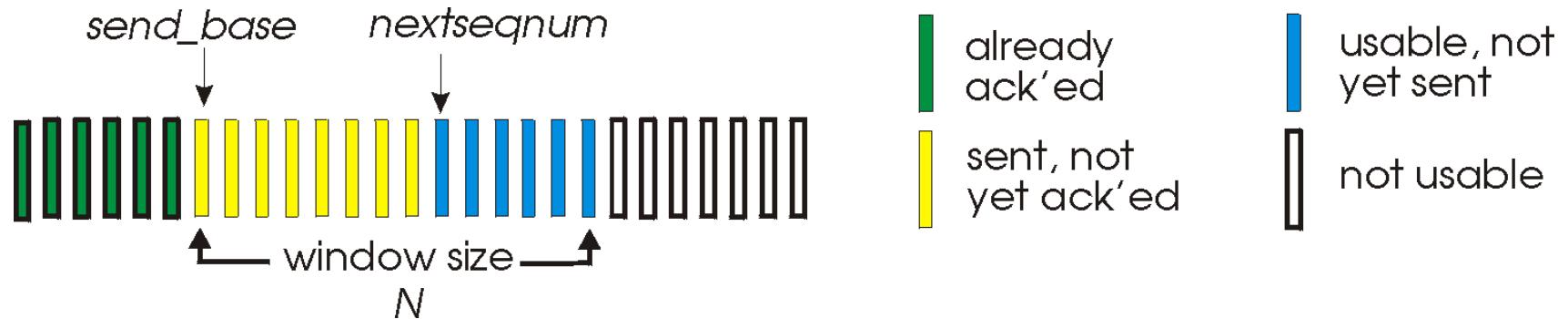
(a) a stop-and-wait protocol in operation



(b) a pipelined protocol in operation

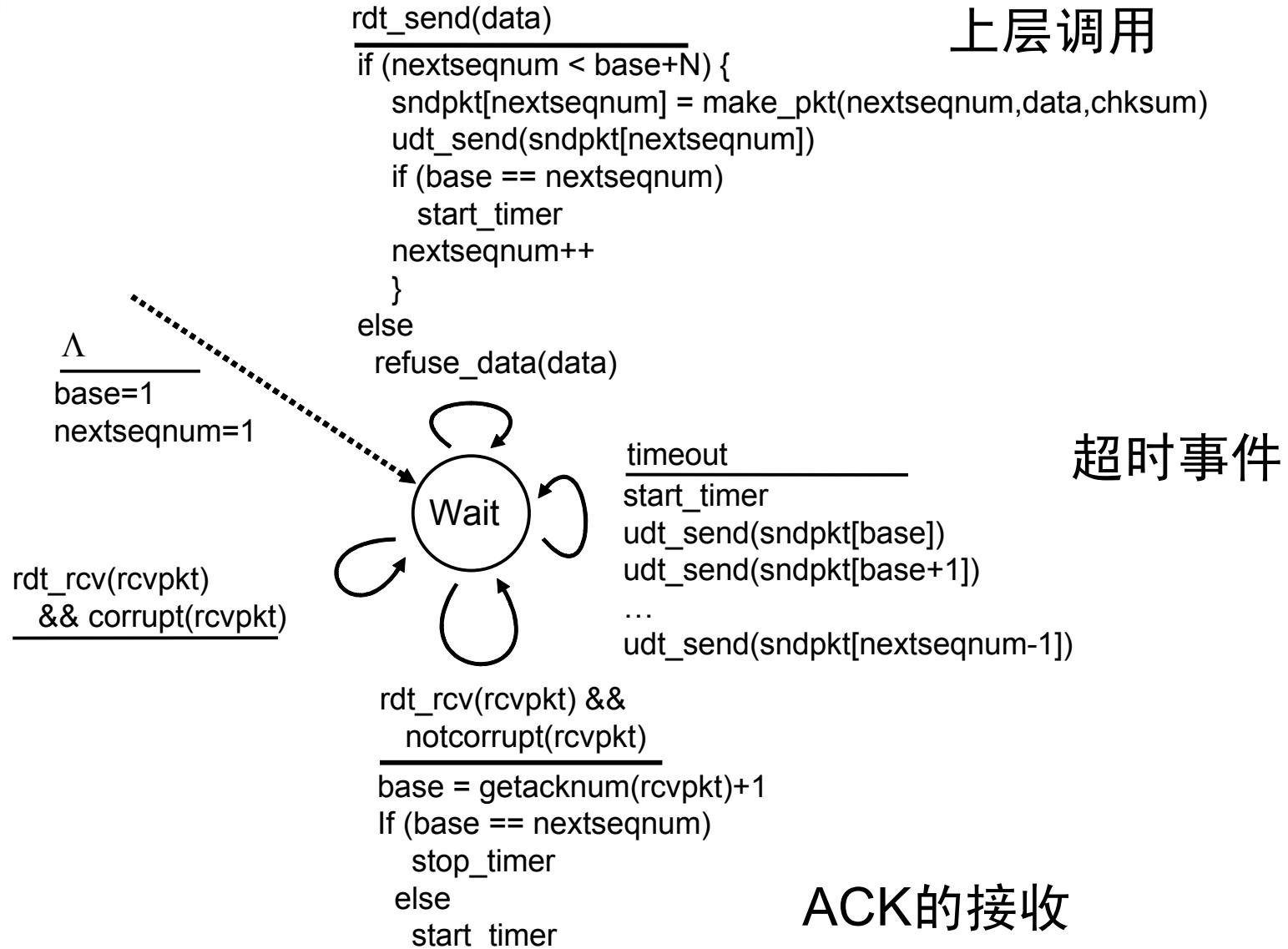
RDT4.0:从第N个分组重发(Go-Back-N)

- 发送方: 在分组首部设置k位 seq #
- 使用尺寸为N的“滑动窗口”, 允许连续的多个分组不被应答

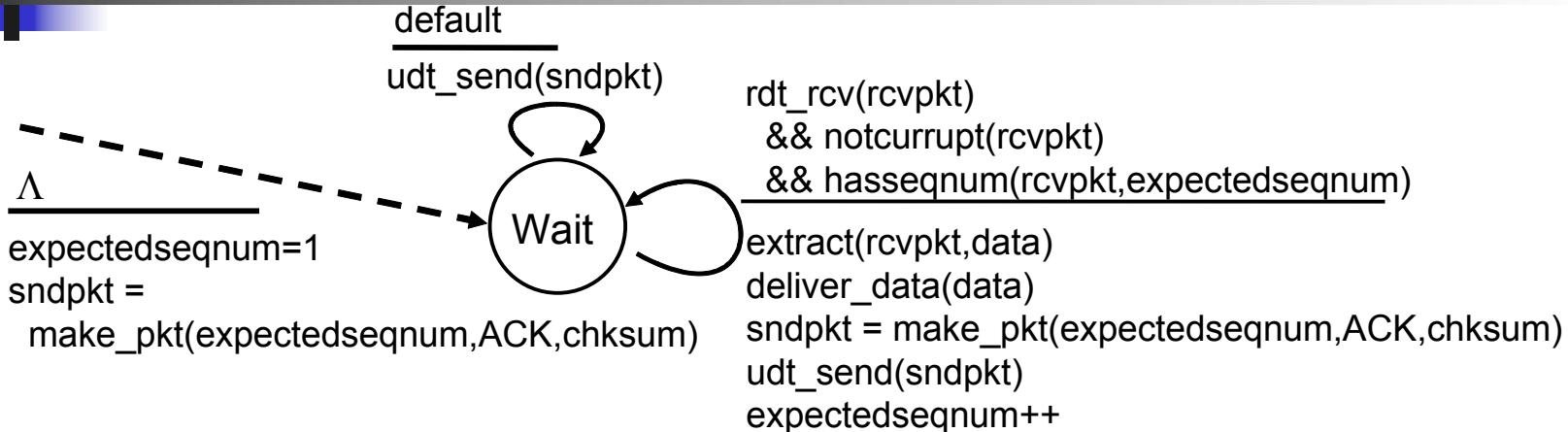


- $ACK(n)$: ACK所有n号之前, 包括n号在内的分组-- “积累式 ACK”
- 为每个未应答 (in-flight) 的分组设置计时器(timer)
- 当发生超时: $timeout(n)$: 重传n号和n号以后的所有在当前发送窗口中的分组

GBN: 发送方扩展的 FSM



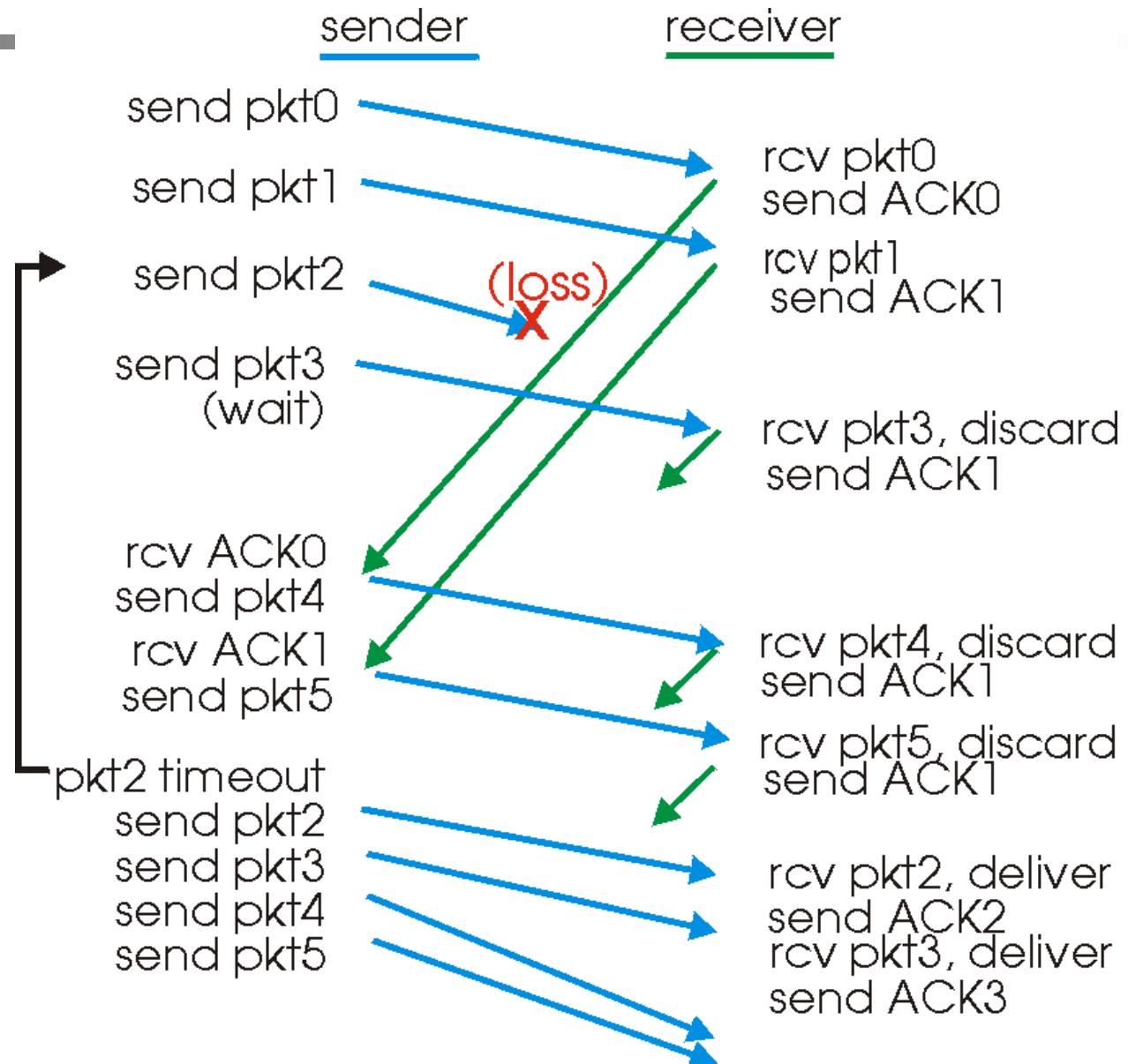
GBN: 接收方的 FSM

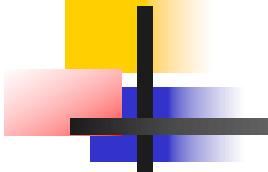


接收方举例:

- **ACK-only:** 总是对正确接收到的分组中按序（*in-order*）对最高 seq # 进行ACK
 - 可以产生重复的ACKs
 - 仅仅需要记住expectedseqnum（预期的序号）
- 失序分组:
 - 丢弃(不缓存) -> 不进行接收缓存!
 - 接收到的分组中按序对最高 seq # 进行ACK

GBN运行

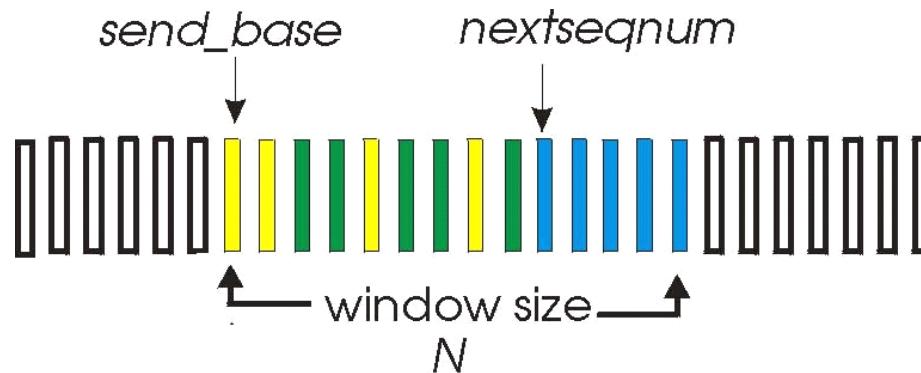




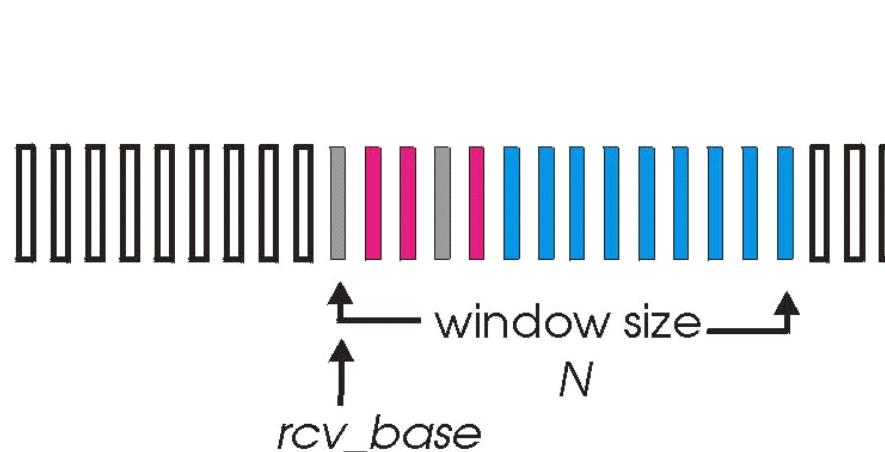
RDT4.2:选择响应协议(Select Response)

- 接收方逐个对所有正确收到的分组进行应答
 - 对接收到的（失序）分组进行缓存，以便最后对上层进行有序递交
- 发送方仅对未收到应答的分组进行重发
 - 发送方为每个unACKed 分组设置计时器
- 发送方的窗口
 - N 个连续的 seq #'s
 - 同样对已发送的seq #'s但未收到ACK的分组进行限制

选择响应：发送方，接收方的窗口

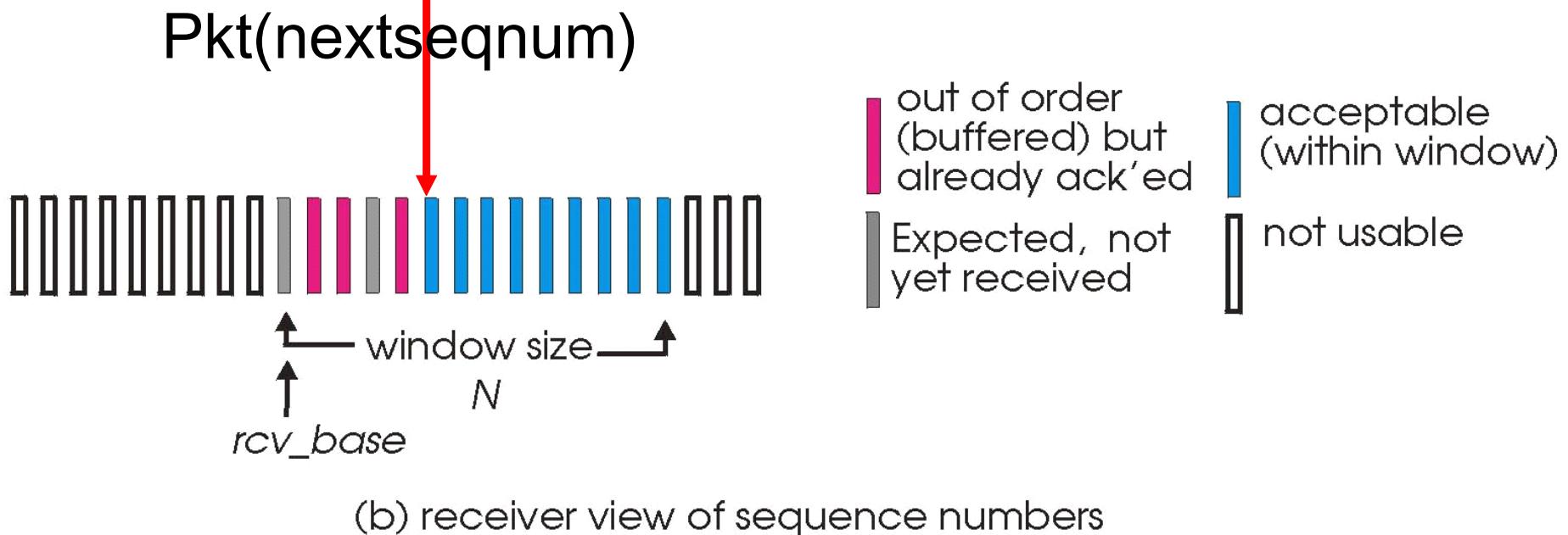
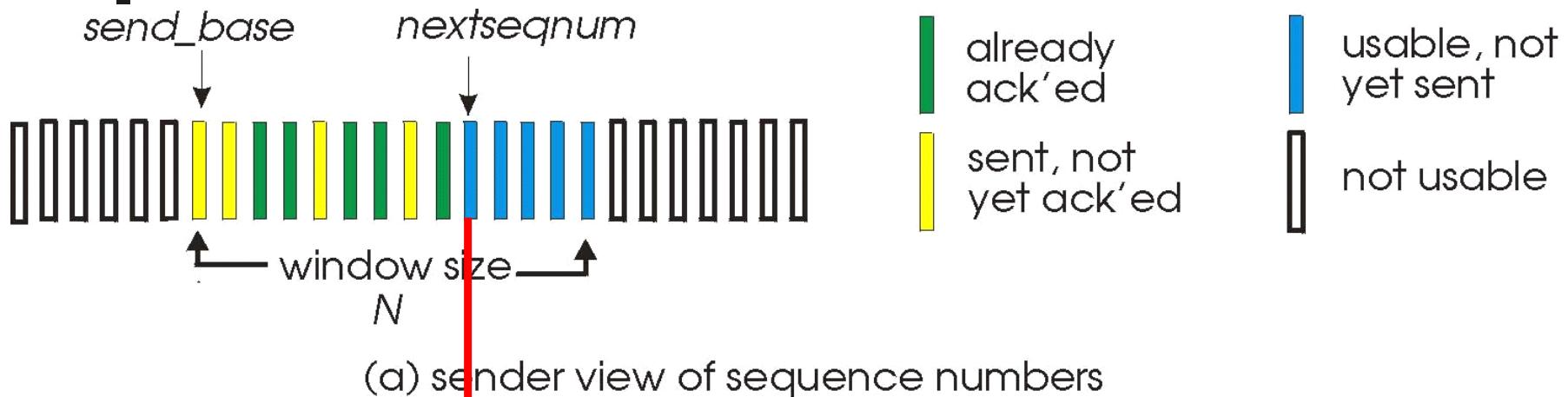


(a) sender view of sequence numbers

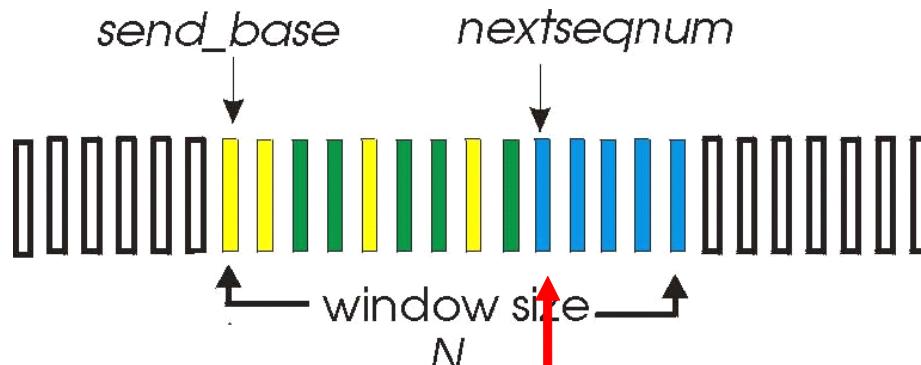


(b) receiver view of sequence numbers

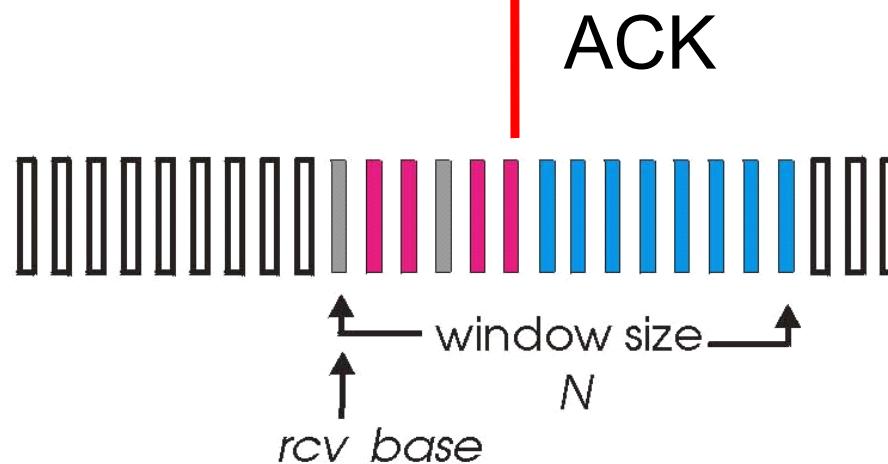
发送方发送Pkt(nextseqnum)



接收方收到Pkt(nextseqnum), 应答

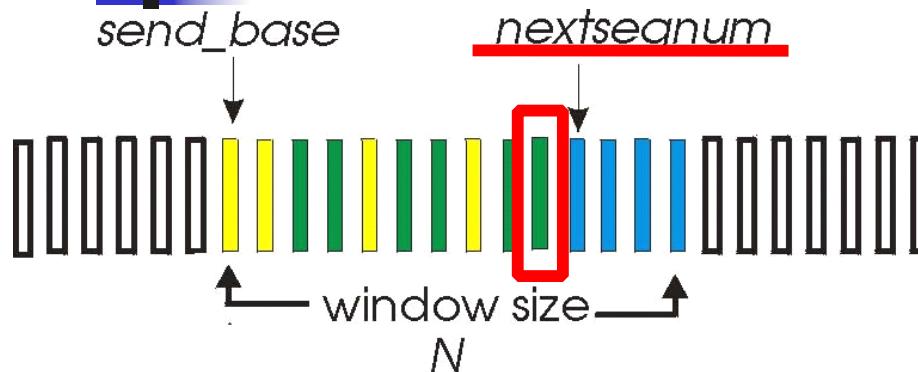


(a) sender view of sequence numbers

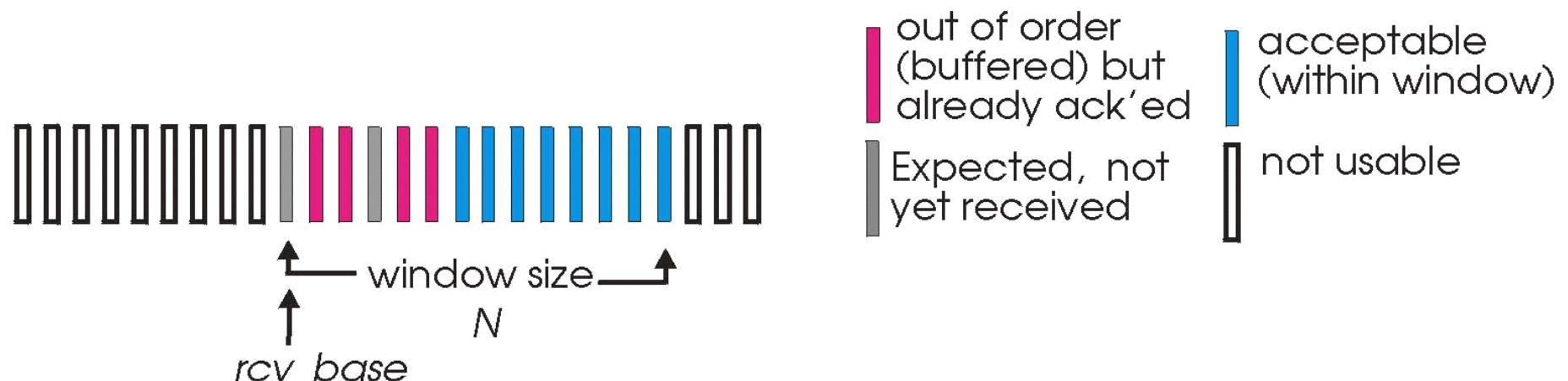


(b) receiver view of sequence numbers

发送方收到ACK(nextseqnum), 移动指针

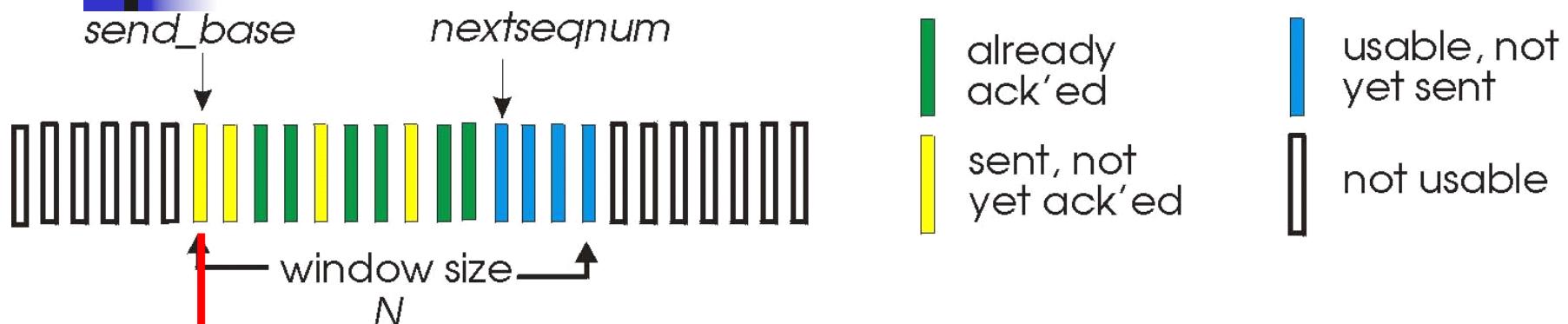


(a) sender view of sequence numbers

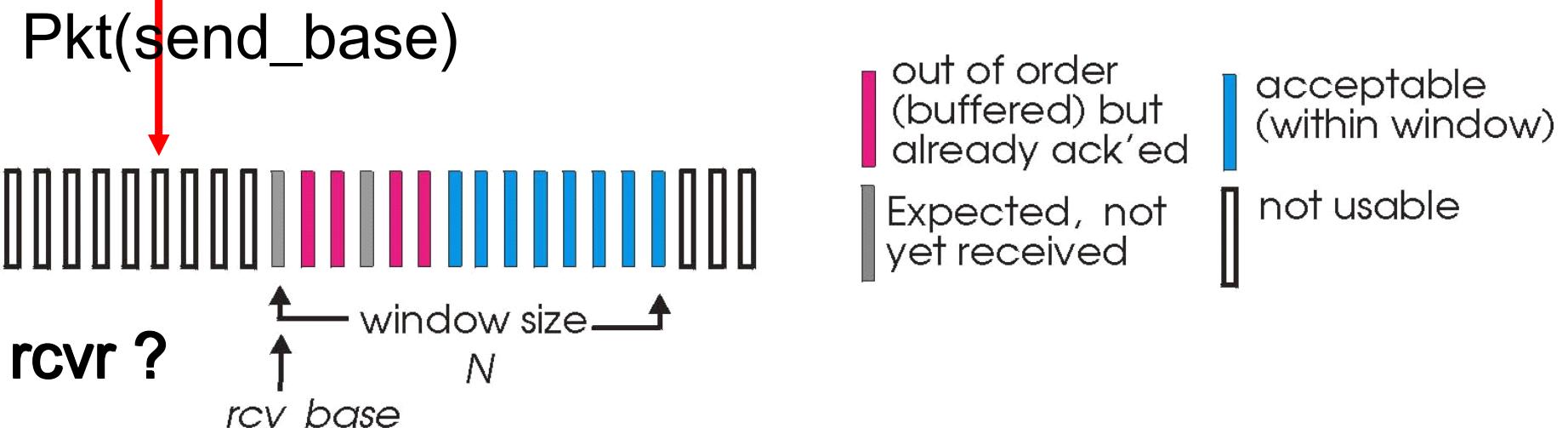


(b) receiver view of sequence numbers

发送方重发Pkt (send_base)

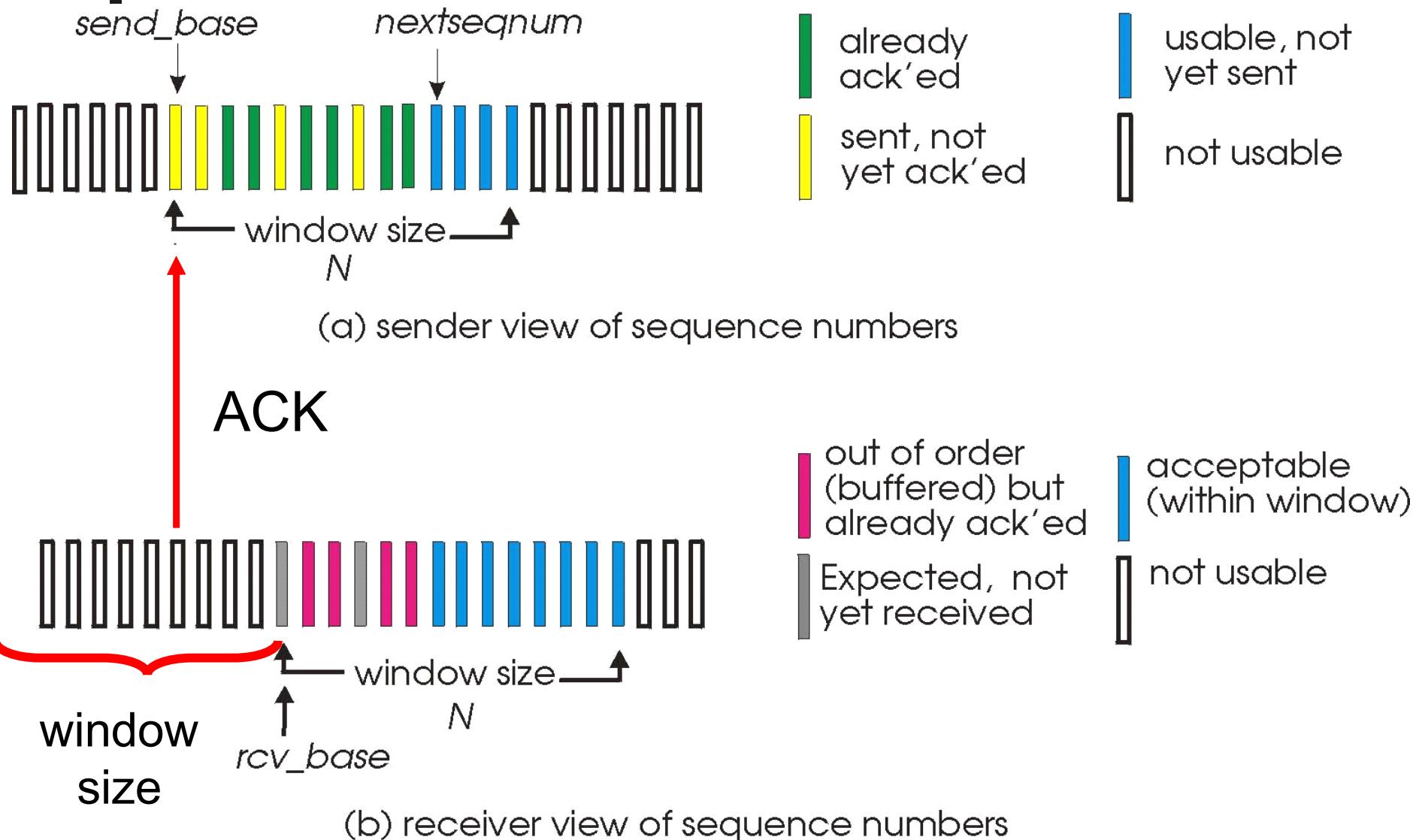


(a) sender view of sequence numbers

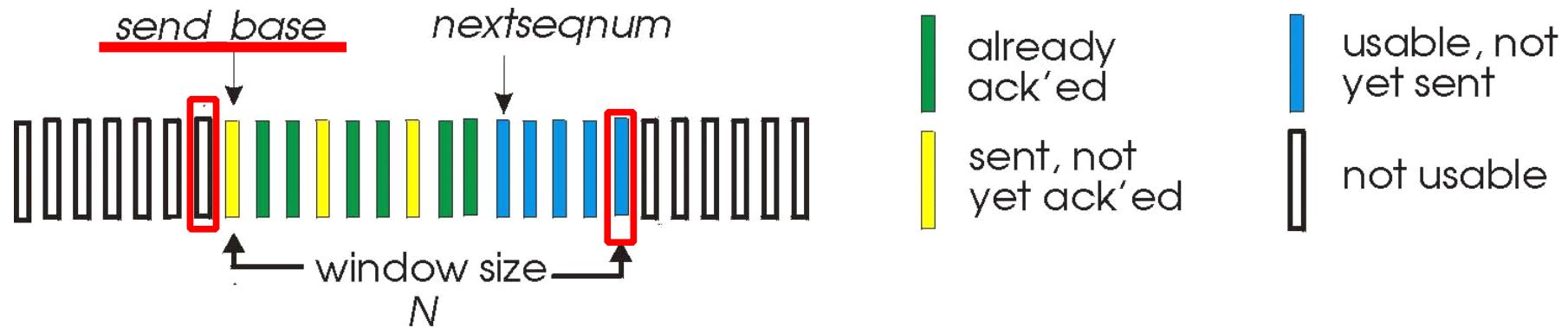


(b) receiver view of sequence numbers

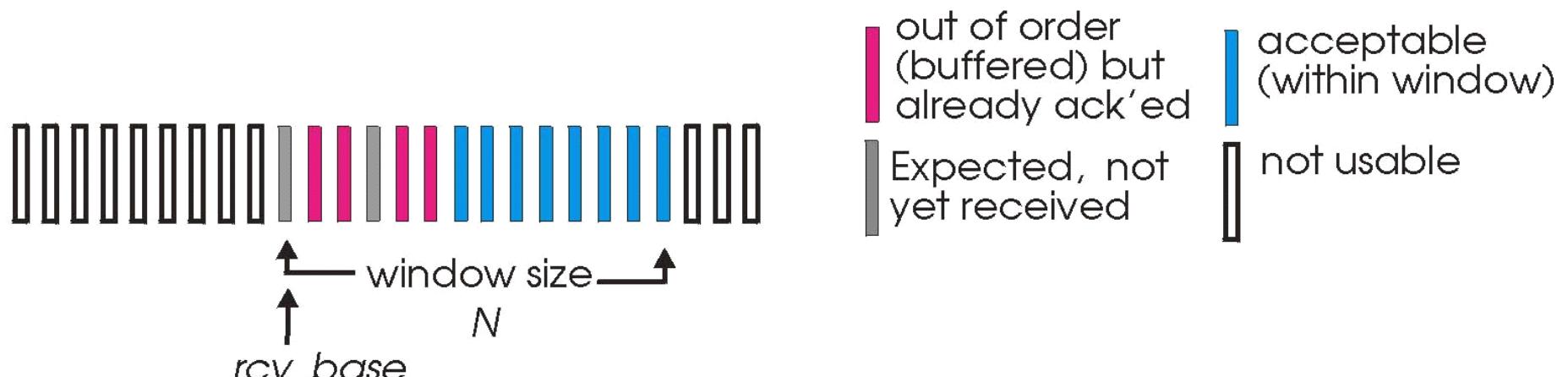
接收方收到Pkt (send_base), 应答



发送方收到ACK(`send_base`)，移动窗口

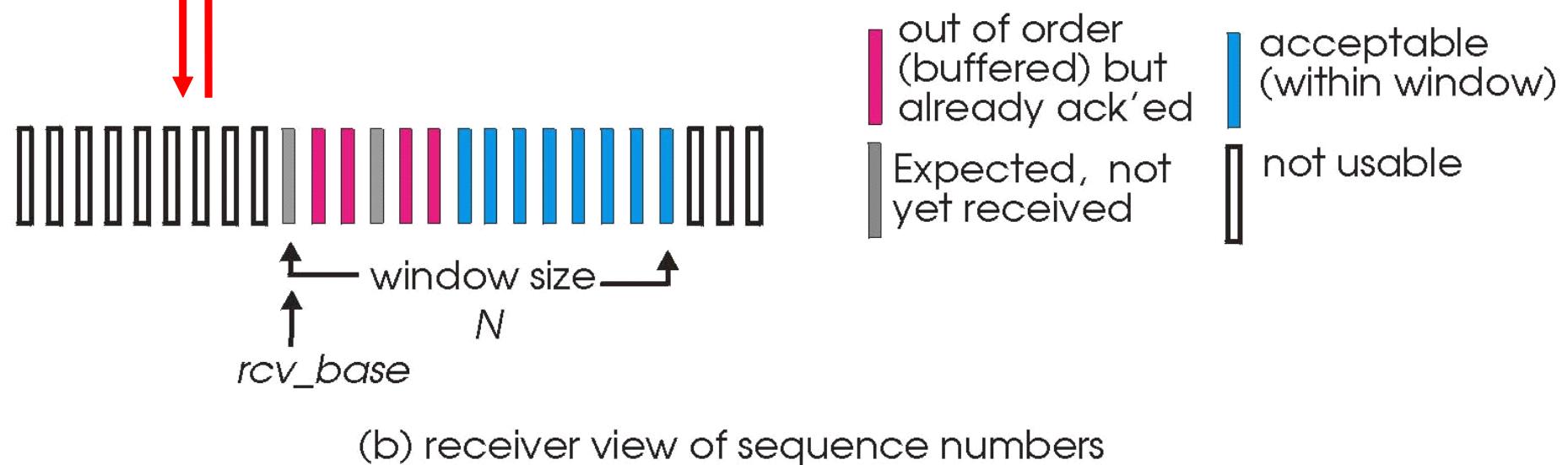
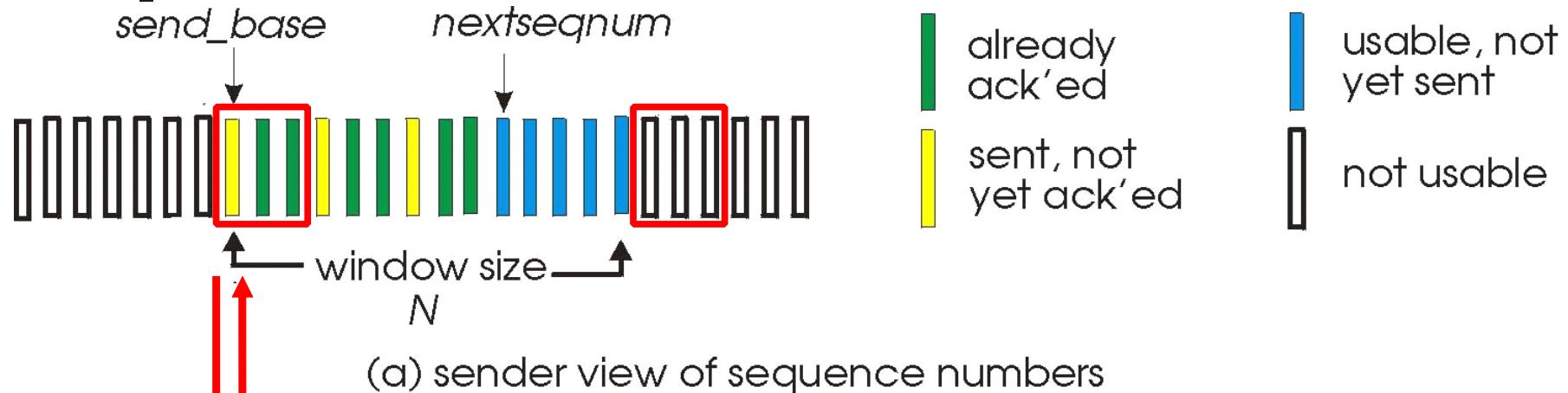


(a) sender view of sequence numbers

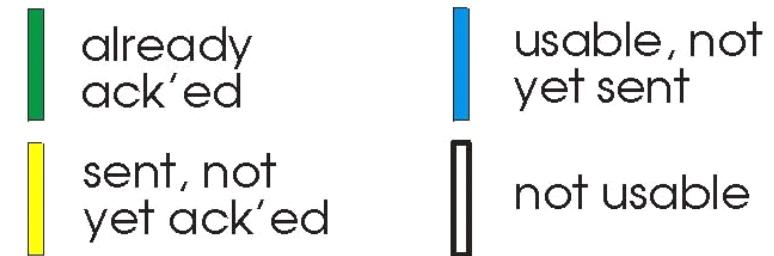
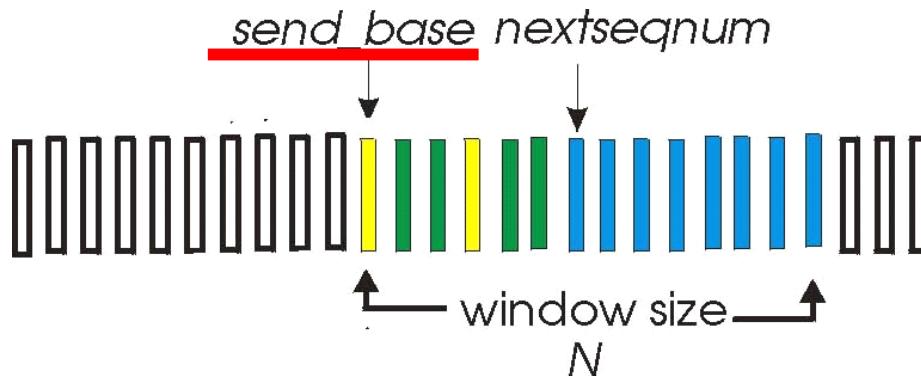


(b) receiver view of sequence numbers

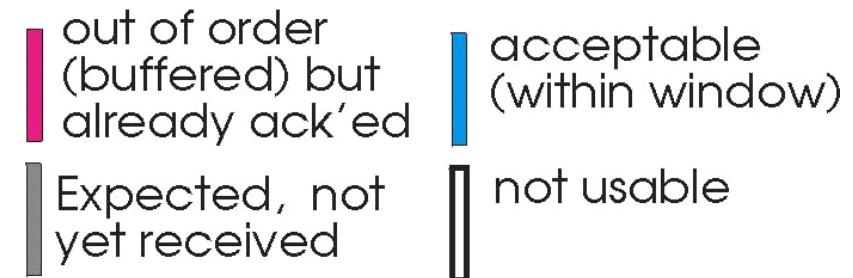
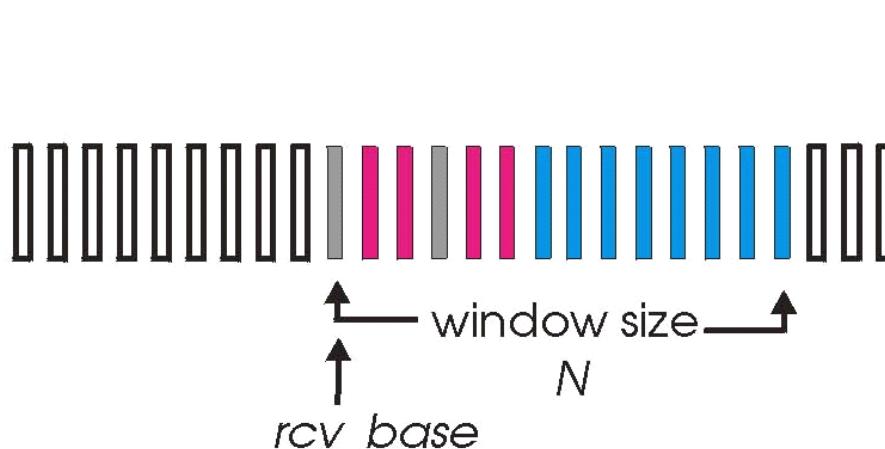
发送方发送新Pkt (send_base) , 接收方应答



发送方收到新的ACK($send_base$)，移动窗口

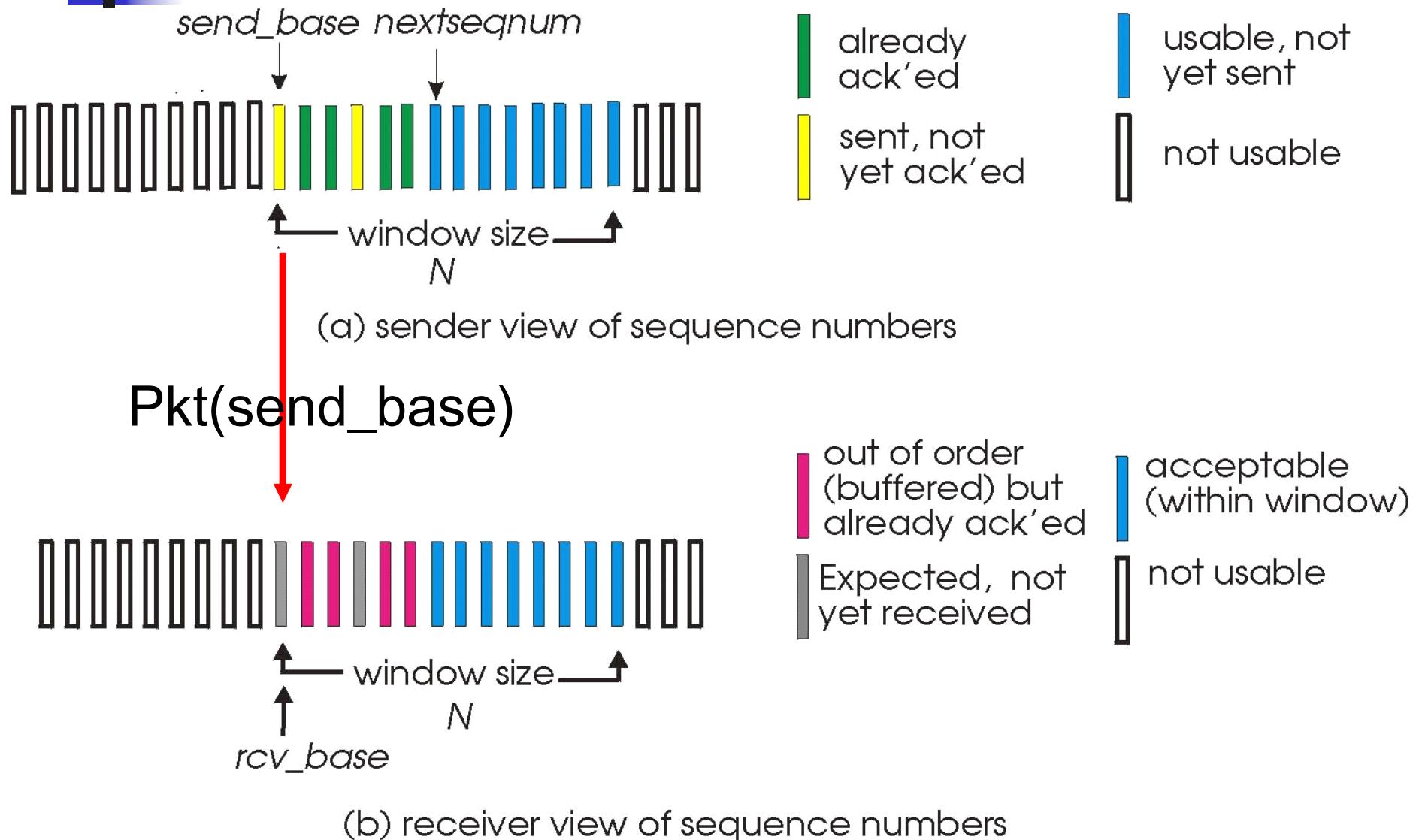


(a) sender view of sequence numbers

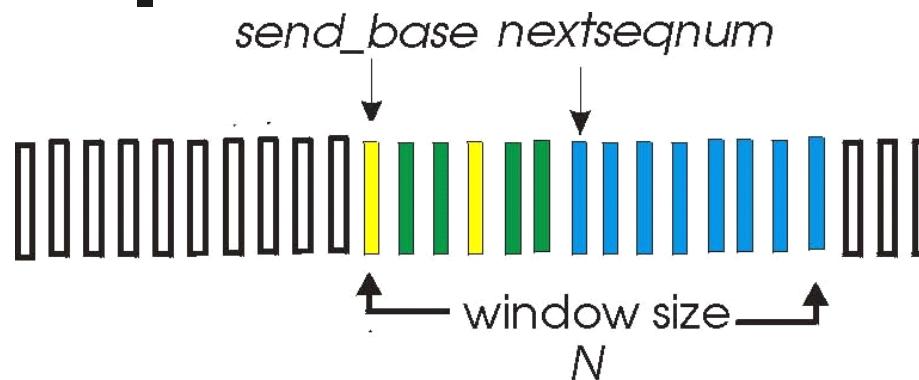


(b) receiver view of sequence numbers

发送方重发新的Pkt(send_base)



接收方收到应答，移动窗口



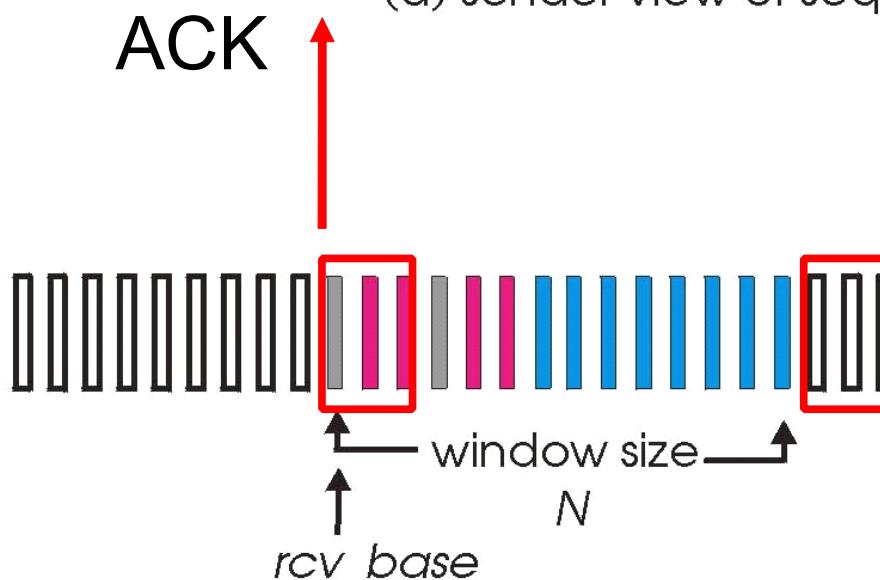
already
ack'ed

sent, not
yet ack'ed

usable, not
yet sent

not usable

(a) sender view of sequence numbers



out of order
(buffered) but
already ack'ed

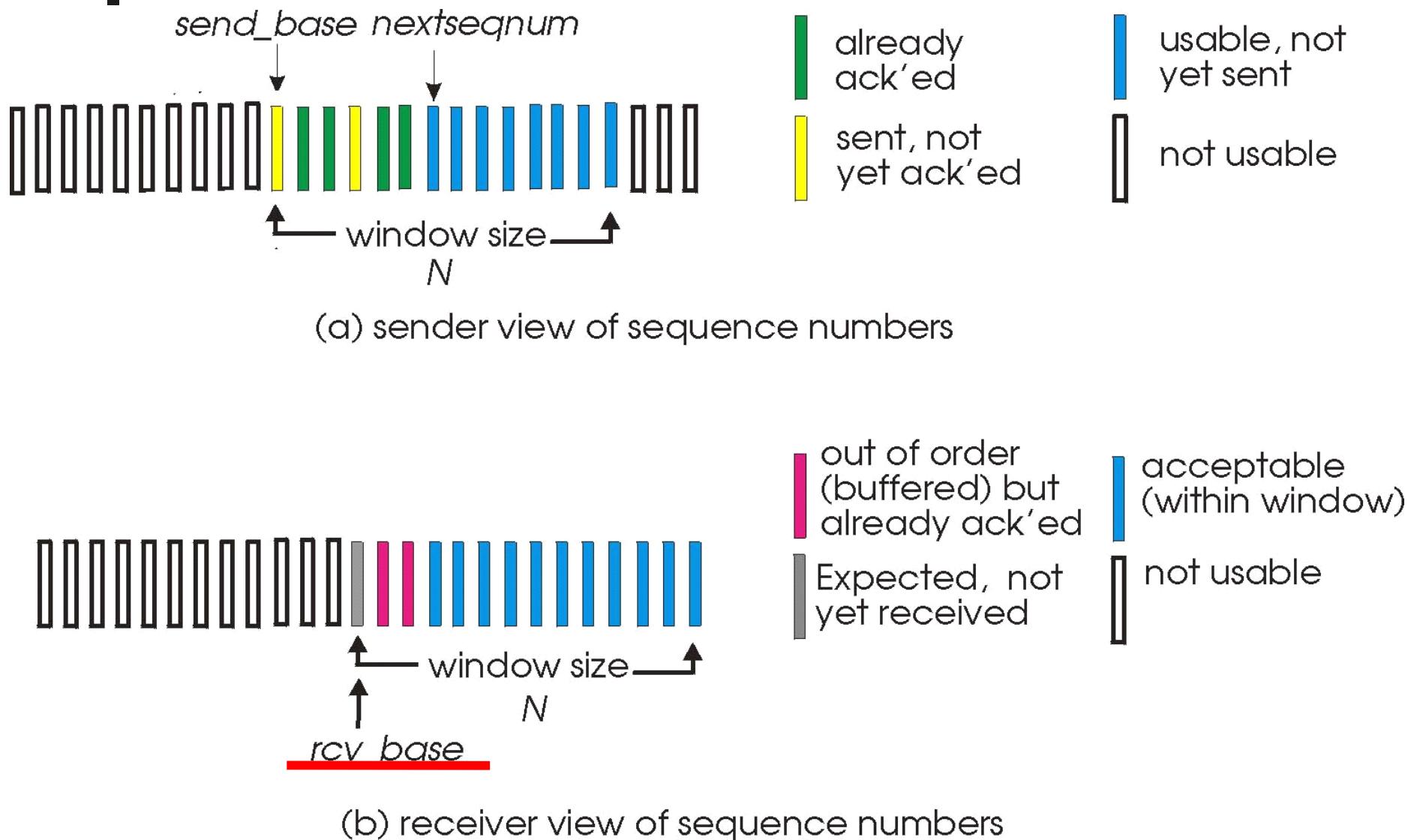
Expected, not
yet received

acceptable
(within window)

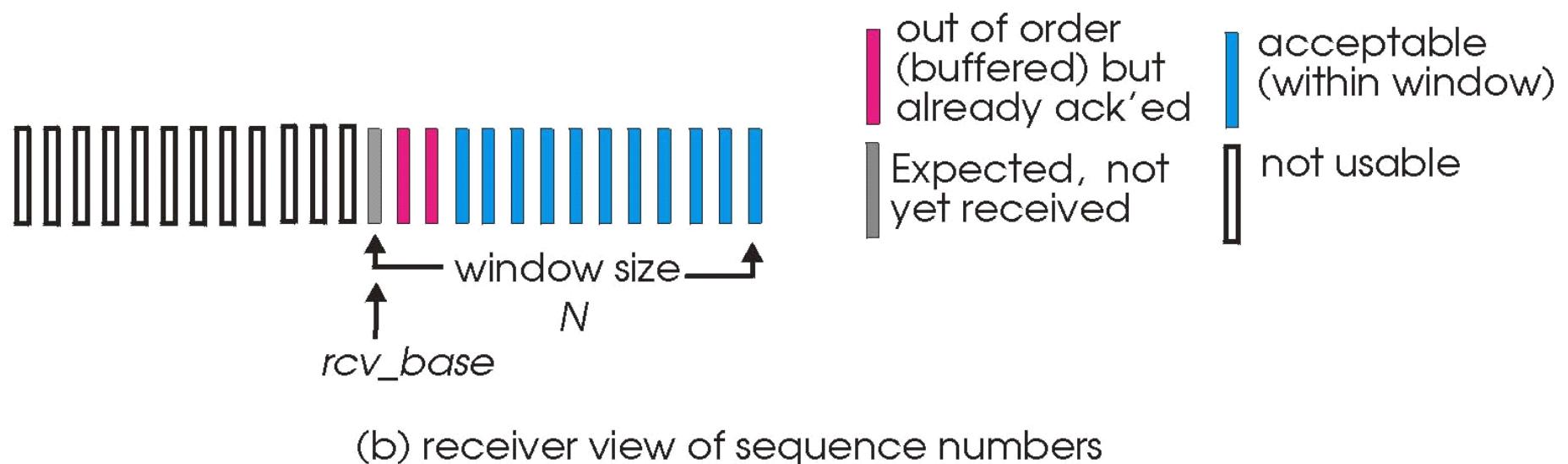
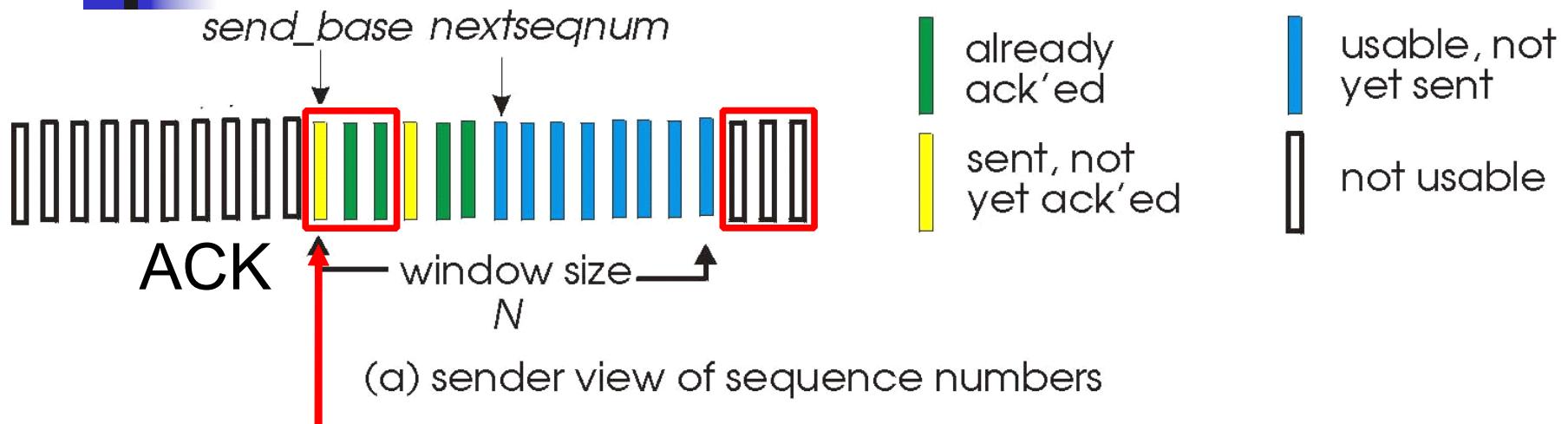
not usable

(b) receiver view of sequence numbers

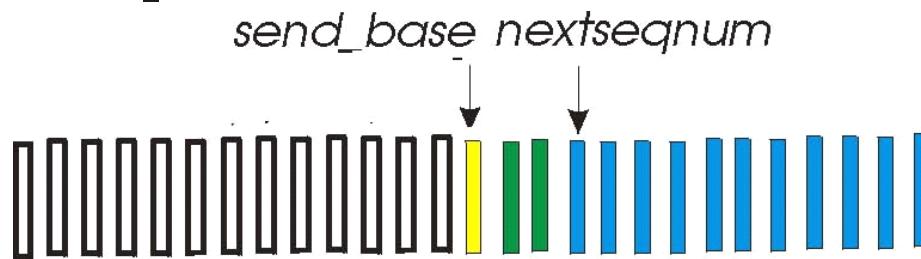
接收方移动窗口后



发送方收到ACK，移动窗口



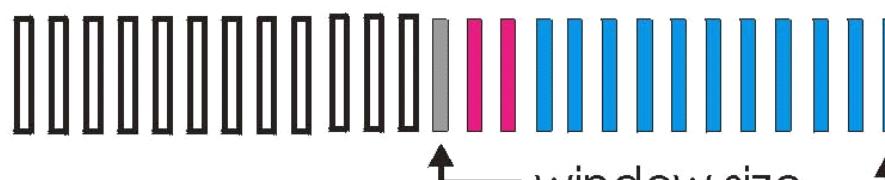
发送方移动窗口后(一致状态)



already
ack'ed
sent, not
yet ack'ed
not usable

usable, not
yet sent
not usable

(a) sender view of sequence numbers



out of order
(buffered) but
already ack'ed
Expected, not
yet received
not usable

acceptable
(within window)
not usable

rcv_base

(b) receiver view of sequence numbers

选择响应协议

发送方

上层数据到达：

- 如果窗口中的下一个序号可用，发送分组

timeout(n):

- 第n个计时器超时
- 重发分组n, 计时器复位

ACK(n) 到达

[sendbase, sendbase+N]:

- 标记分组 n 已经收到
- 如n为unACKed分组中的最小值, 将窗口下沿前推到下一个unACKed seq #

接收方

分组n到达

[rcvbase, rcvbase+N-1]

- 发送 ACK(n)
- 失序: 缓存
- 有序: 递交到上层 (同时递交缓存中的其他有序分组), 将窗口前推到下一个尚未收到的分组

分组n到达

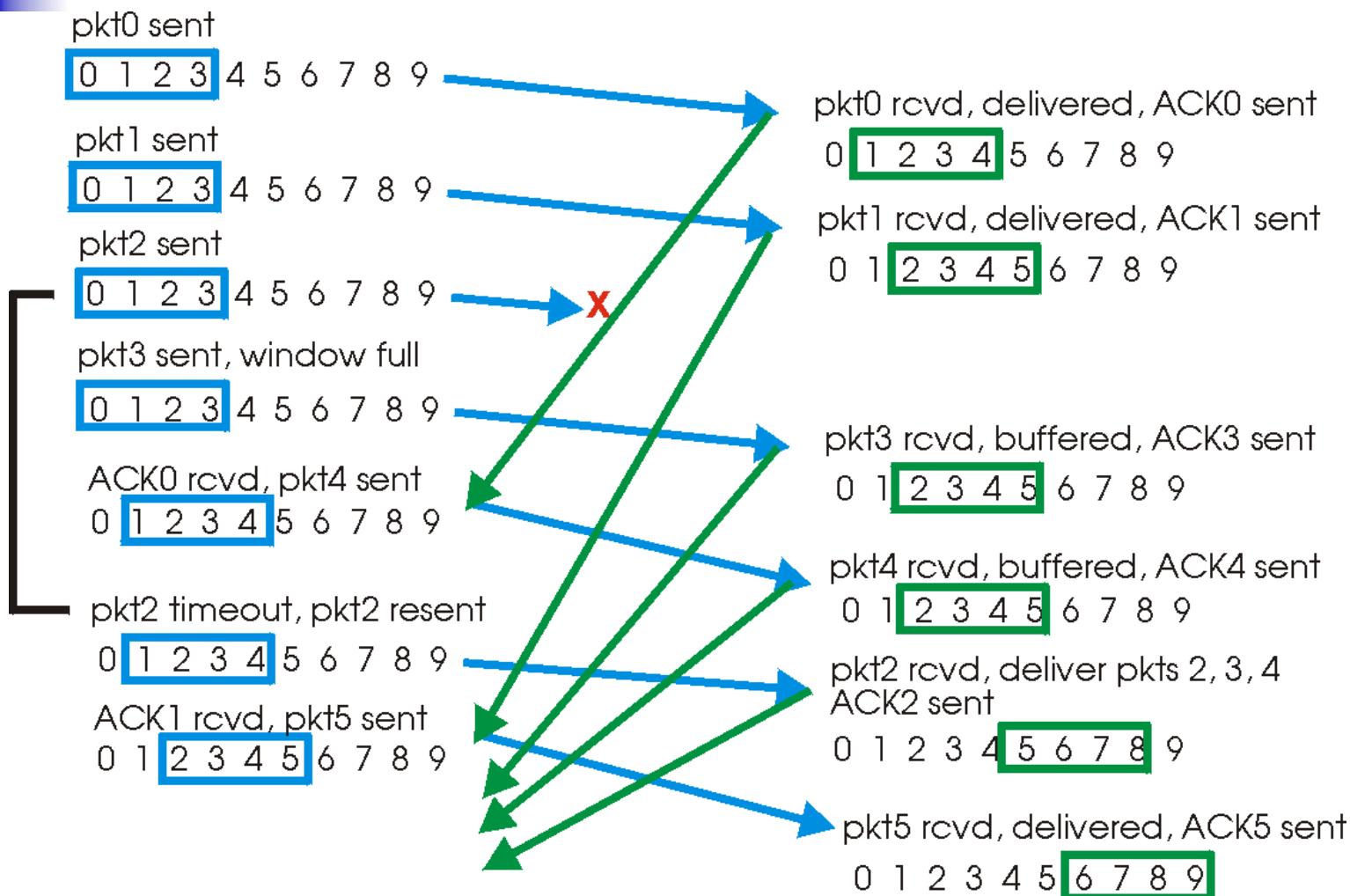
[rcvbase-N, rcvbase-1]

- 虽然接收方曾经确认, 但仍然需要ACK(n)

其他情况:

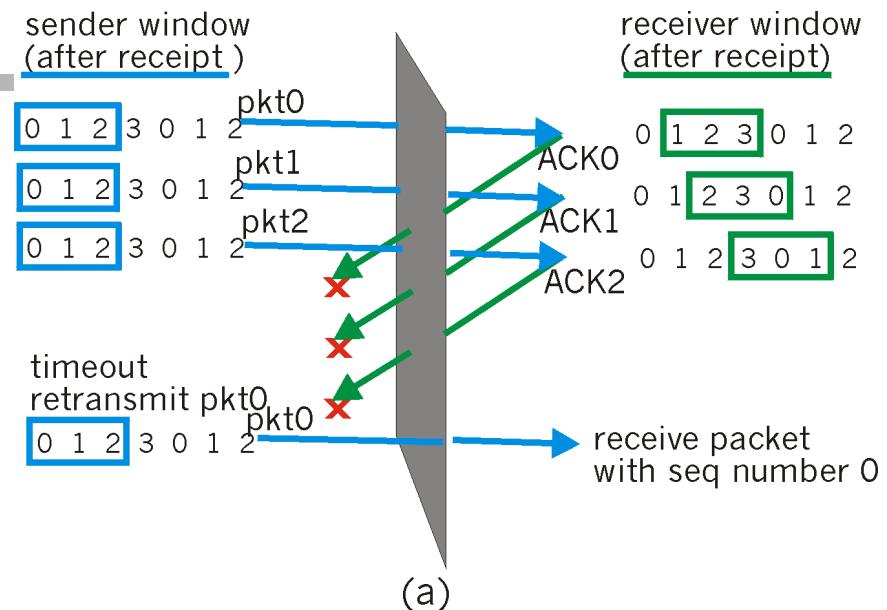
- 忽略分组

选择响应协议的运行

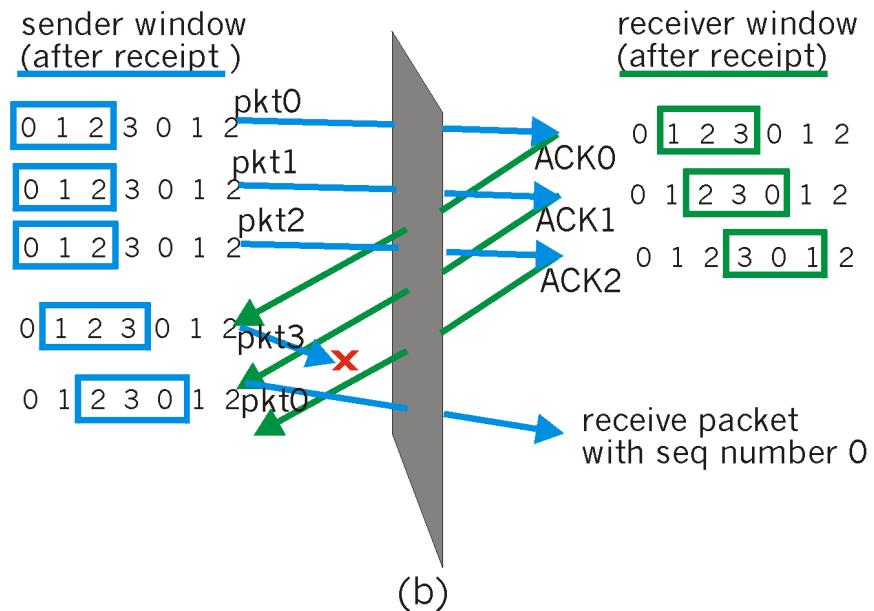


选择响应困境

- 问题环境
 - seq #: 0, 1, 2, 3
 - window size=3
- 右图两种情况没有区别
 - Pkt0是重发分组，还是新分组？
- Key: seq # size 和 window size 的关系?
 - $window\ size = seq\# \ size / 2$
- 还有问题么?
 - 分组失序到达的间隔太久
 - 分组有寿命，不能活太长， 3分钟

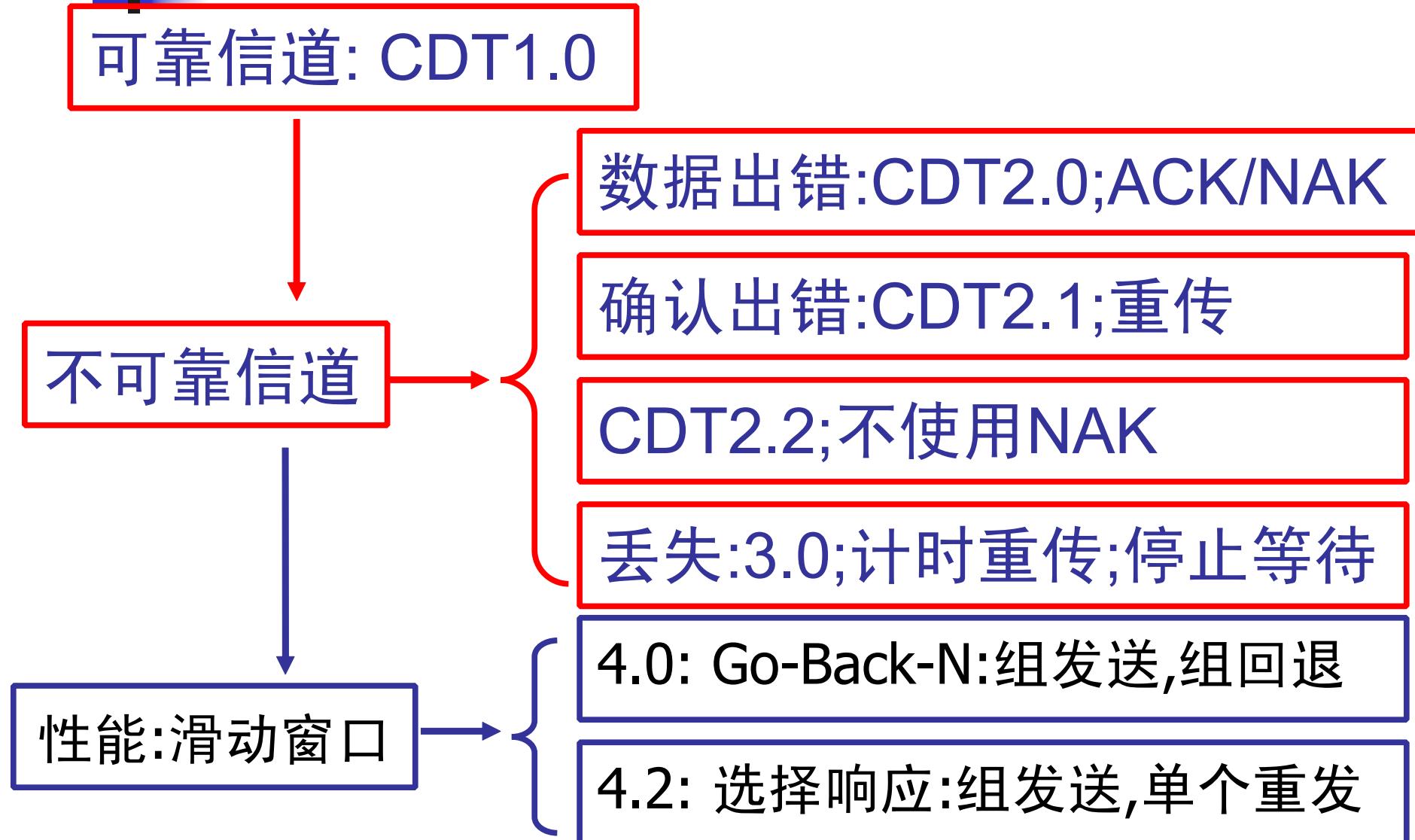


(a)



(b)

可靠传输的实现

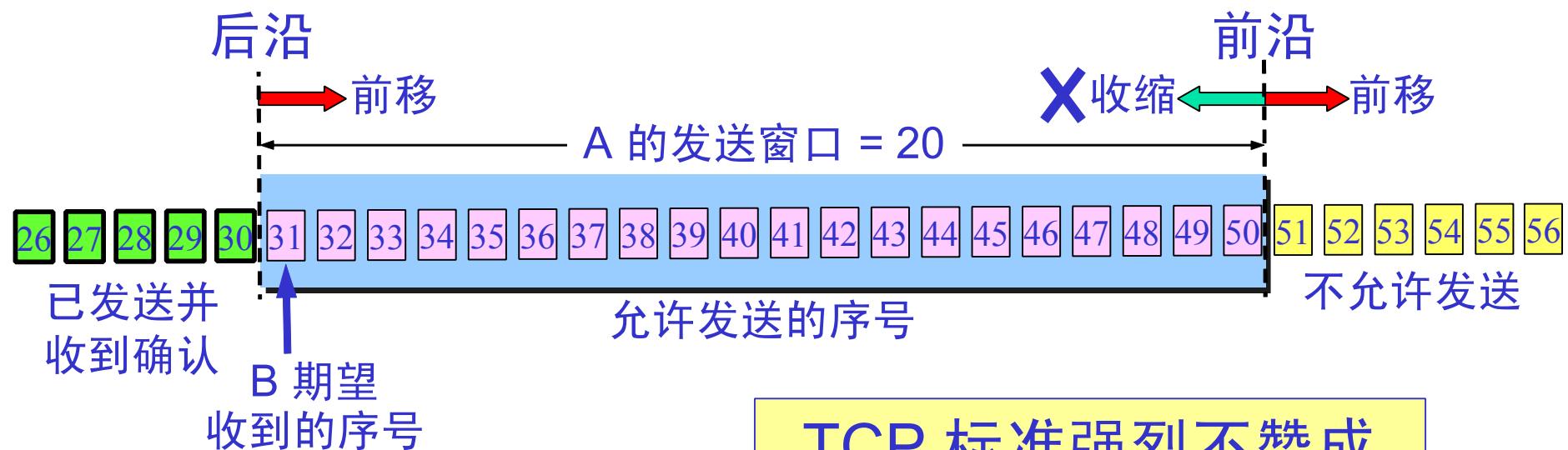


5.6 TCP 可靠传输的实现

5.6.1 以字节为单位的滑动窗口

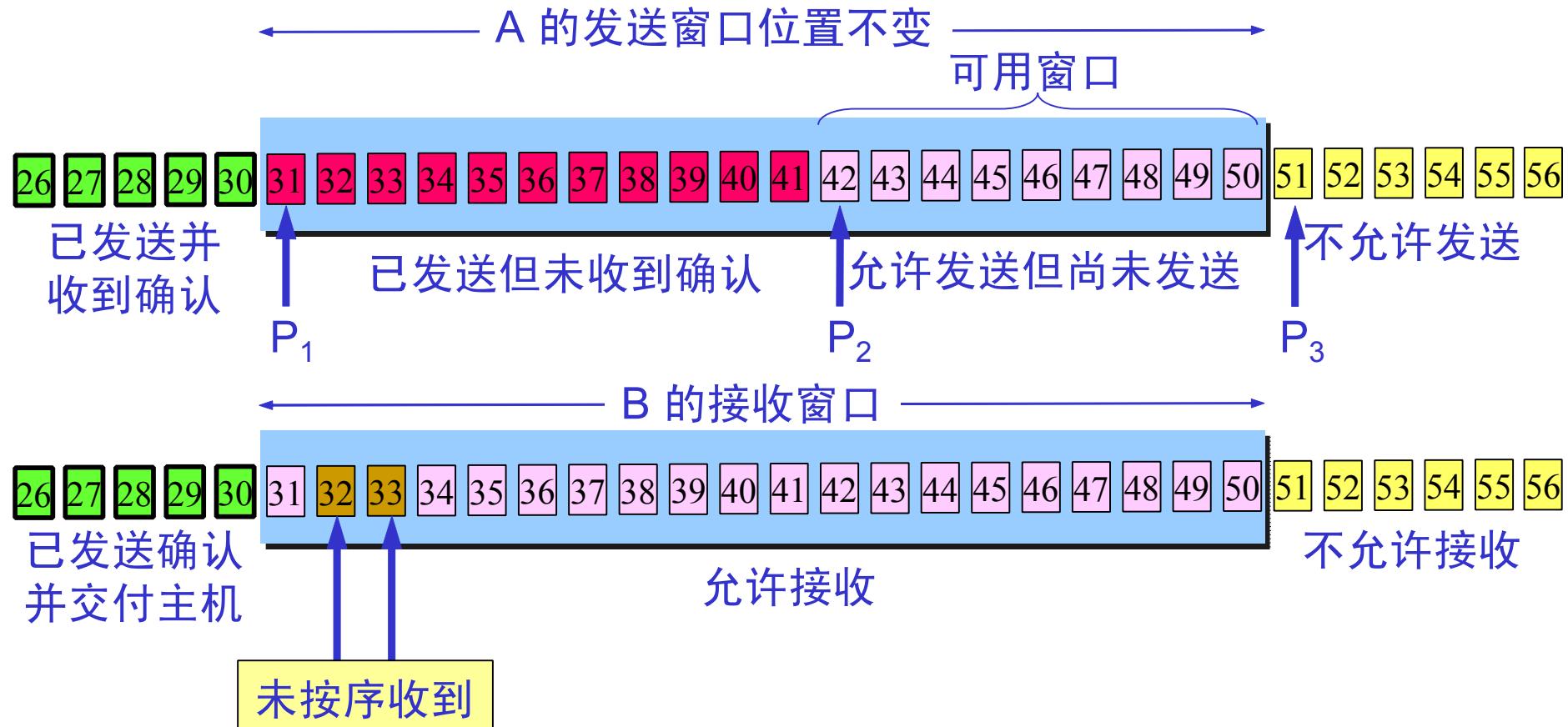
选择响应协议实现

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 个分组的数据

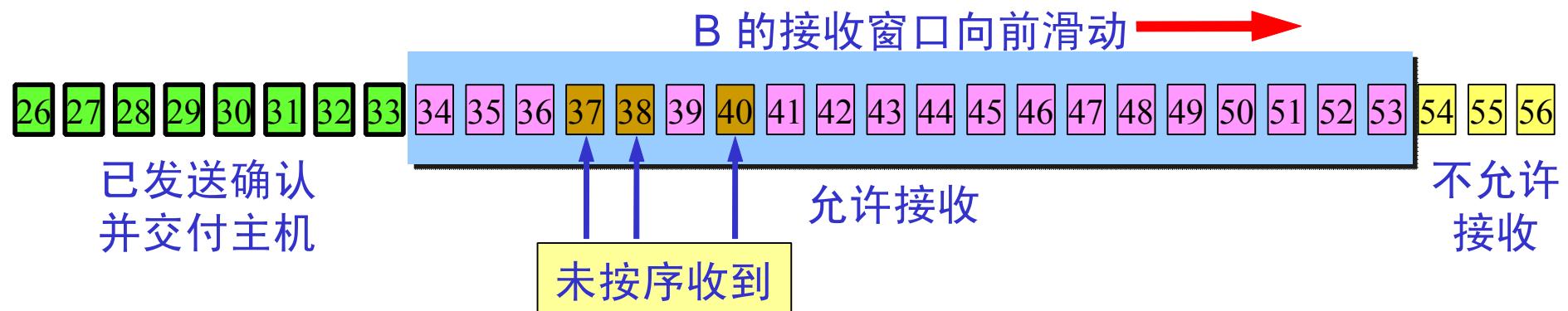
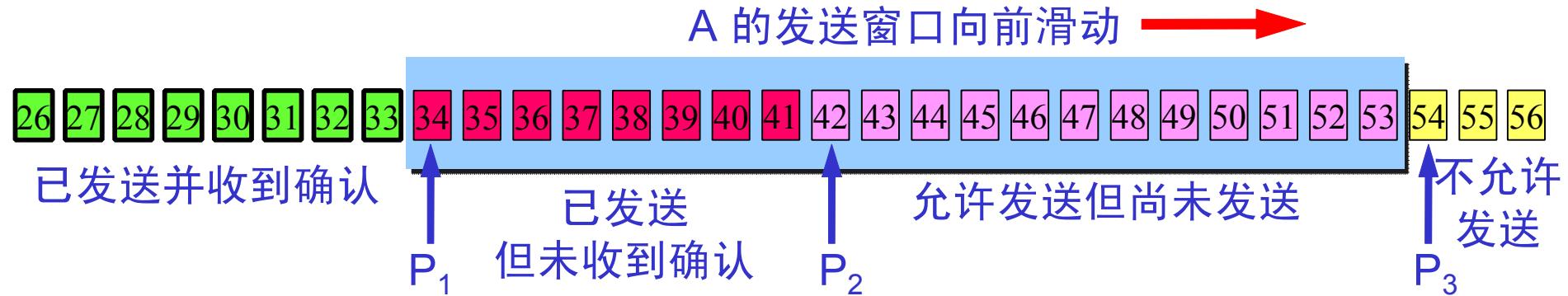


$P_3 - P_1 = A$ 的发送窗口（又称为通知窗口）

$P_2 - P_1 =$ 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 收到确认号 (31) , 发送窗口向前滑动



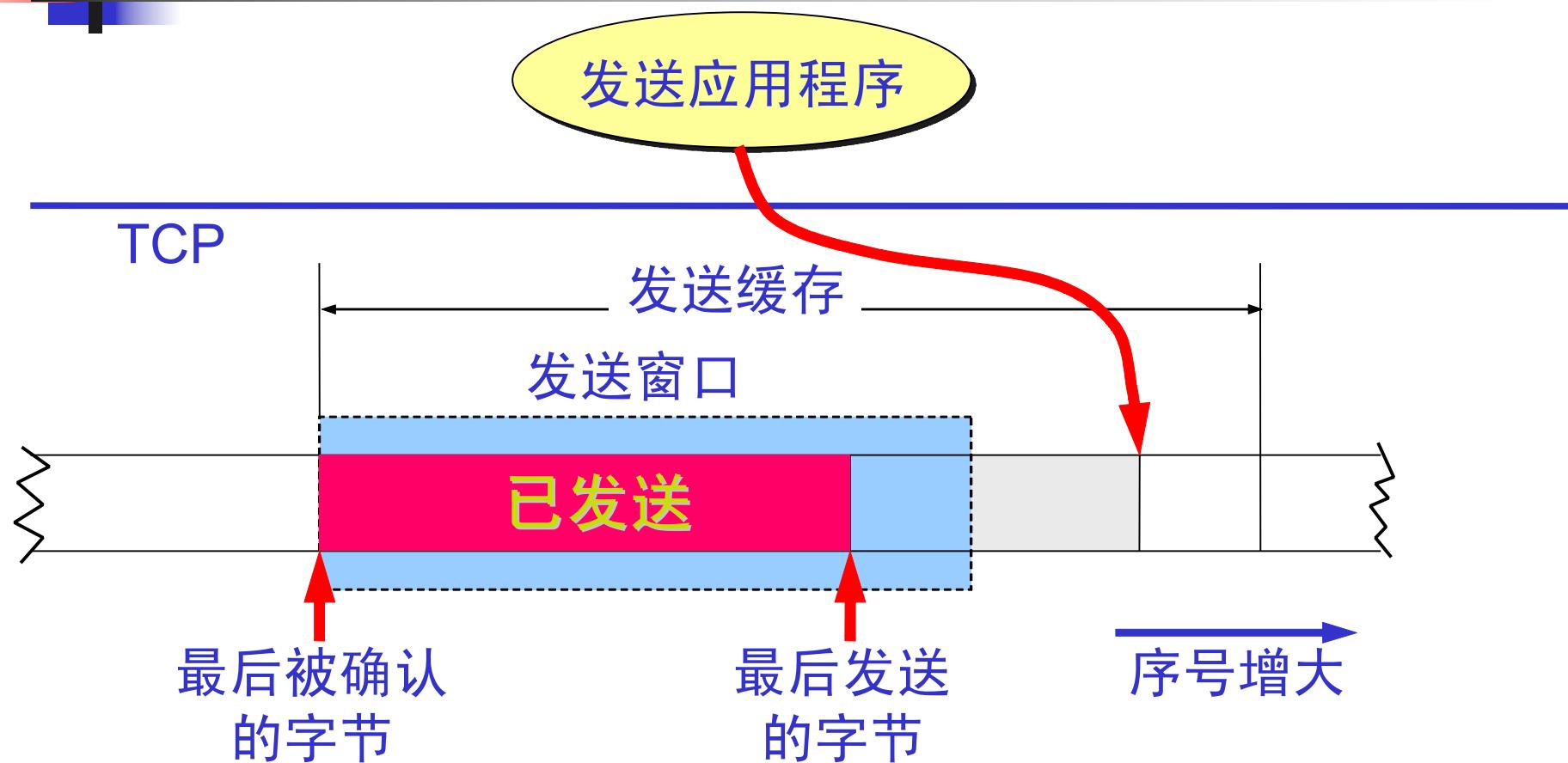
先存下，等待缺少的数据的到达

未按序收到

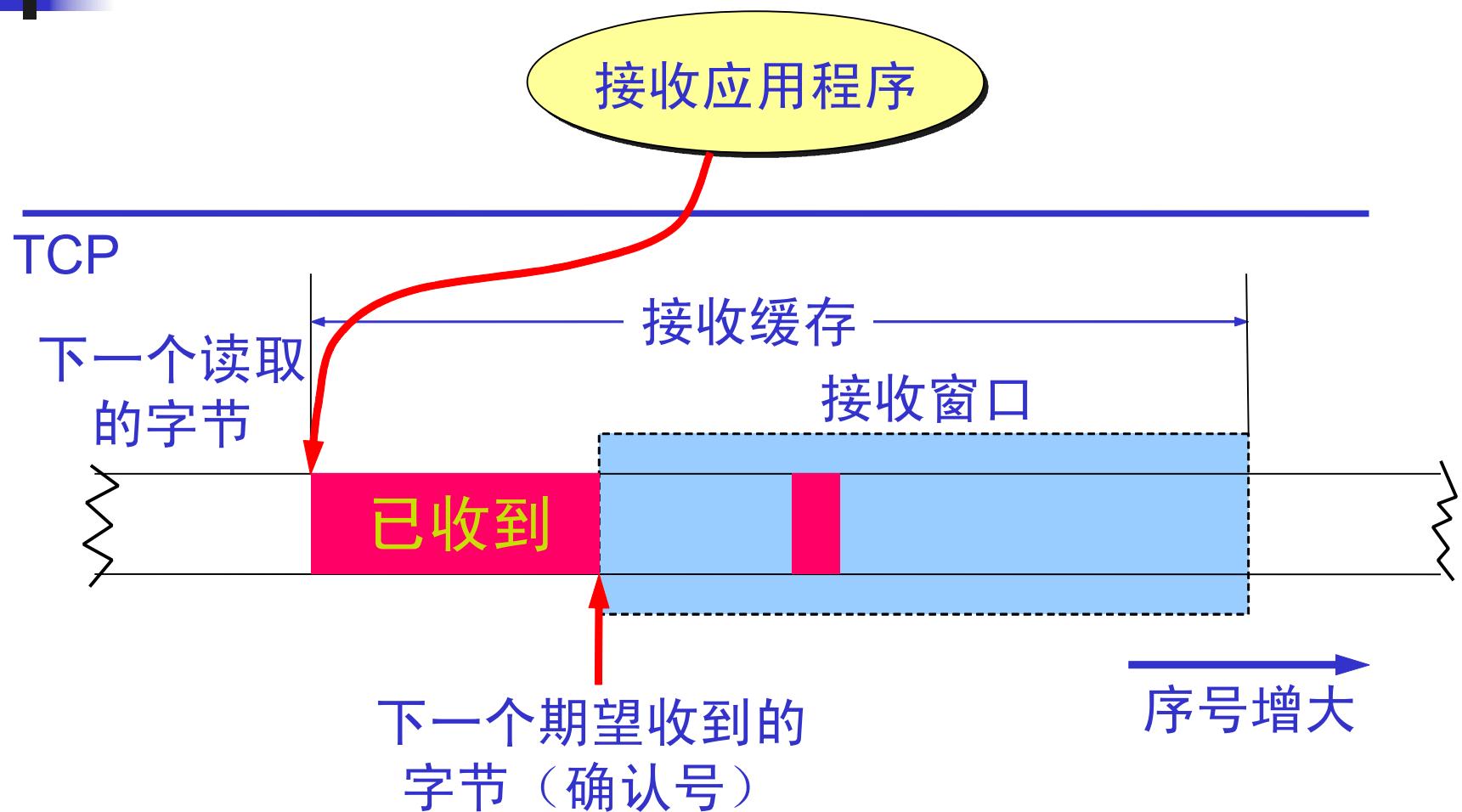
A 的发送窗口内的序号都已用完，
但还没有再收到确认，必须停止发送。

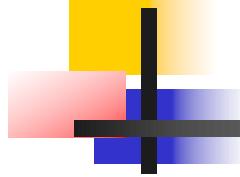


发送缓存



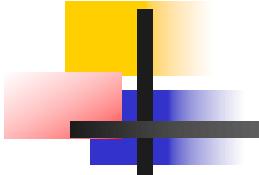
接收缓存





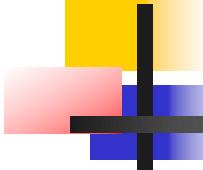
发送缓存与接收缓存的作用

- 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

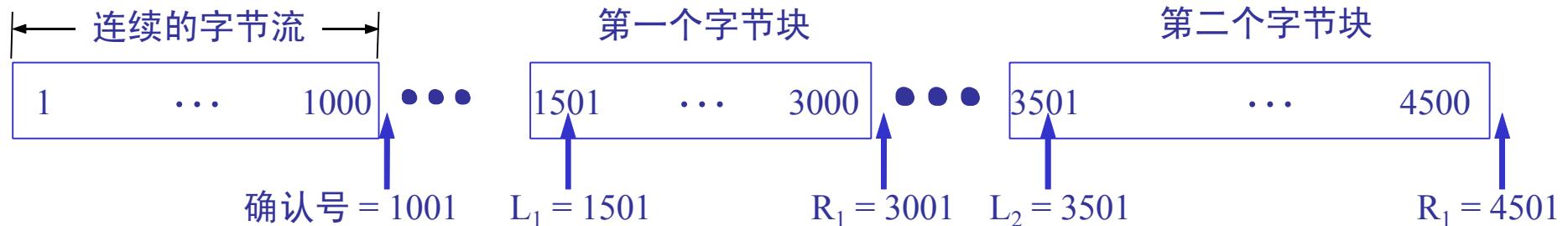
- A 的发送窗口并不总是和 B 的接收窗口一样大（因为有一定的时间滞后）。
- TCP 标准没有规定对不按序到达的数据应如何处理。
 - 主流实现参考选择应答算法：
 - 先临时存放在接收窗口中，等到字节流中所缺少的字节收到后，再按序交付上层的应用进程。
- TCP 要求接收方必须有累积确认的功能，这样可以减小传输开销。



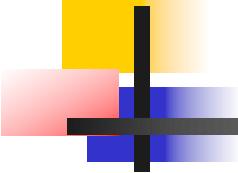
5.6.3 选择确认 SACK(Selective ACK)

- 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

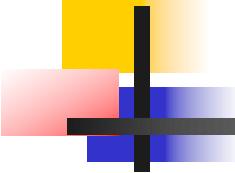


- 和前后字节不连续的每一个字节块都有两个边界：左边界和右边界。图中用四个指针标记这些边界。
- 第一个字节块的左边界 $L_1 = 1501$ ，但右边界 $R_1 = 3001$ 。
- 左边界指出字节块的第一个字节的序号，但右边界减 1 才是字节块中的最后一个序号。
- 第二个字节块的左边界 $L_2 = 3501$ ，而右边界 $R_2 = 4501$ 。



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。

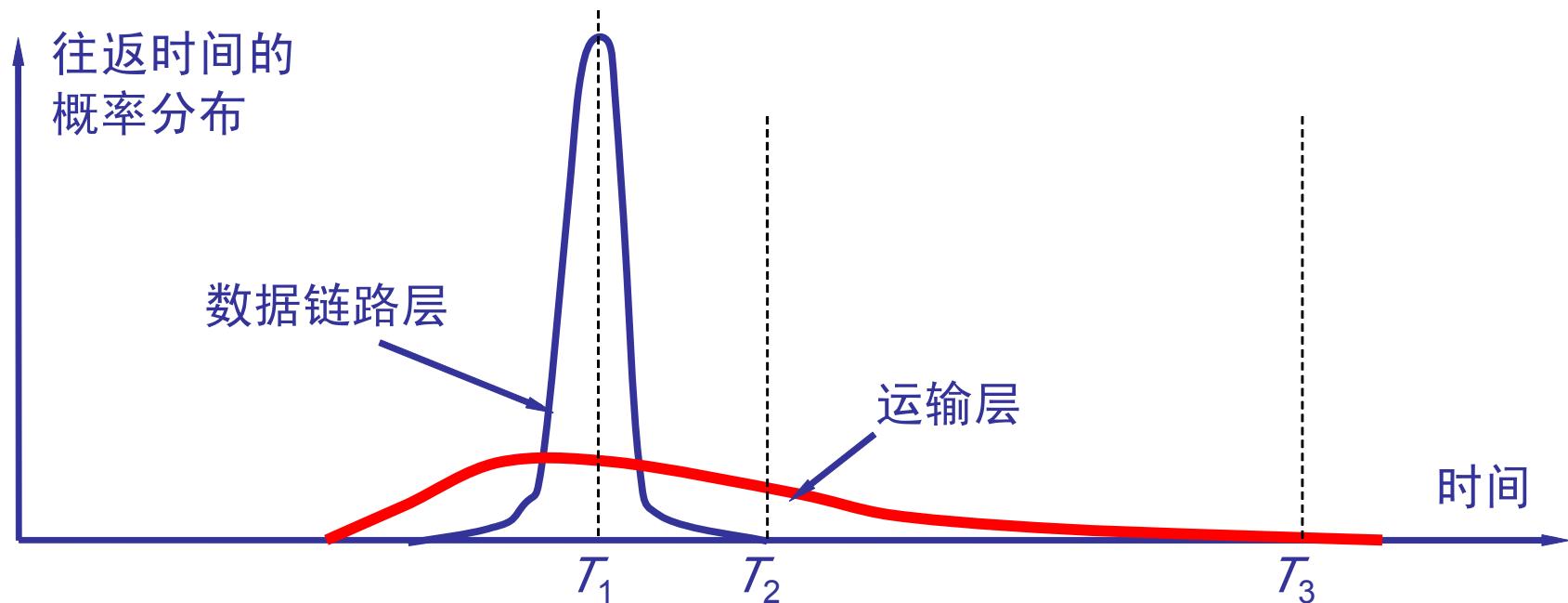


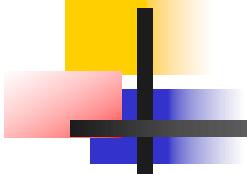
5.6.2 超时重传时间的选择

- 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。



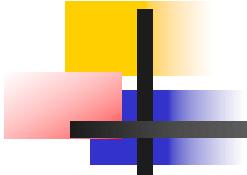


加权平均往返时间

- 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 ：

$$\begin{aligned} \text{新的 } RTT_s = & (1 - \alpha) \times (\text{旧的 } RTT_s) \\ & + \alpha \times (\text{新的 RTT 样本}) \end{aligned} \quad (5-4)$$

- 式中， $0 \leq \alpha < 1$ 。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

(Retransmission Time-Out)

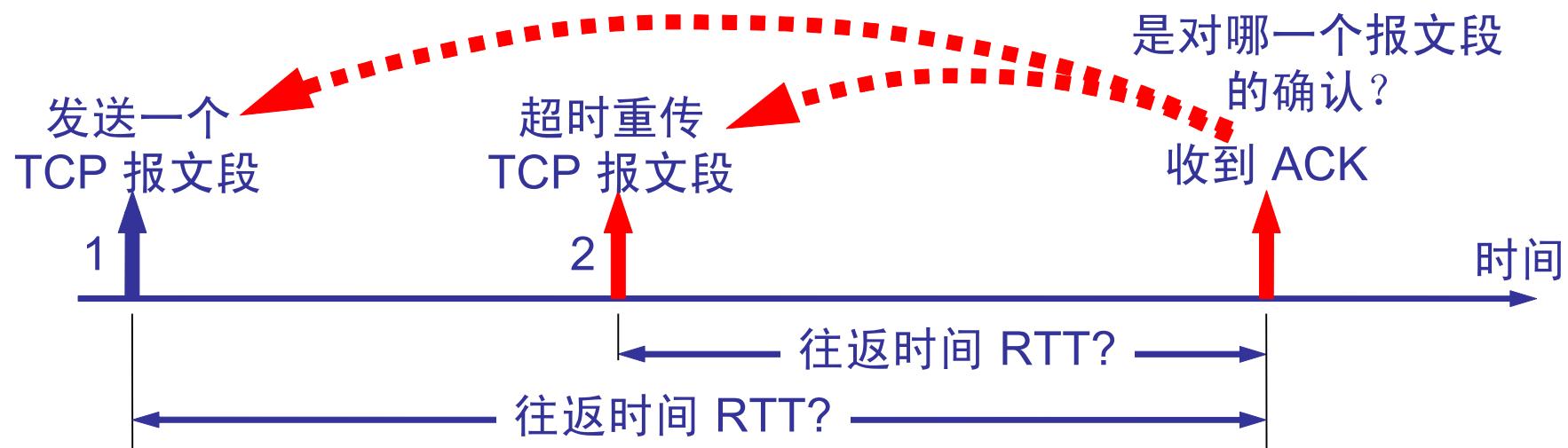
- RTO 应略大于上面得出的加权平均往返时间 RTT_S 。
- RFC 2988 建议使用下式计算 RTO:
- $$RTO = RTT_S + 4 \times RTT_D \quad (5-5)$$
- RTT_D 是 RTT 的**偏差**的加权平均值。
- RFC 2988 建议这样计算 RTT_D 。第一次测量时， RTT_D 值取为测量到的 RTT 样本值的一半。在以后的测量中，则使用下式计算加权平均的 RTT_D :

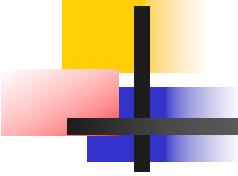
$$\begin{aligned} \text{新的 } RTT_D &= (1 - \beta) \times (\text{旧的 } RTT_D) \\ &\quad + \beta \times |RTT_S - \text{新的 RTT 样本}| \end{aligned} \quad (5-6)$$

- 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

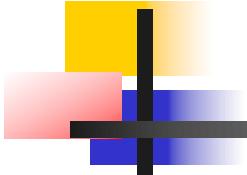
- TCP 报文段 1 没有收到确认。重传（即报文段 2）后，收到了确认报文段 ACK。
- 如何判定此确认报文段是对原来的报文段 1 的确认，还是对重传的报文段 2 的确认？





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。

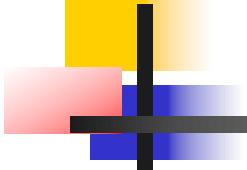


修正的 Karn 算法

- 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。



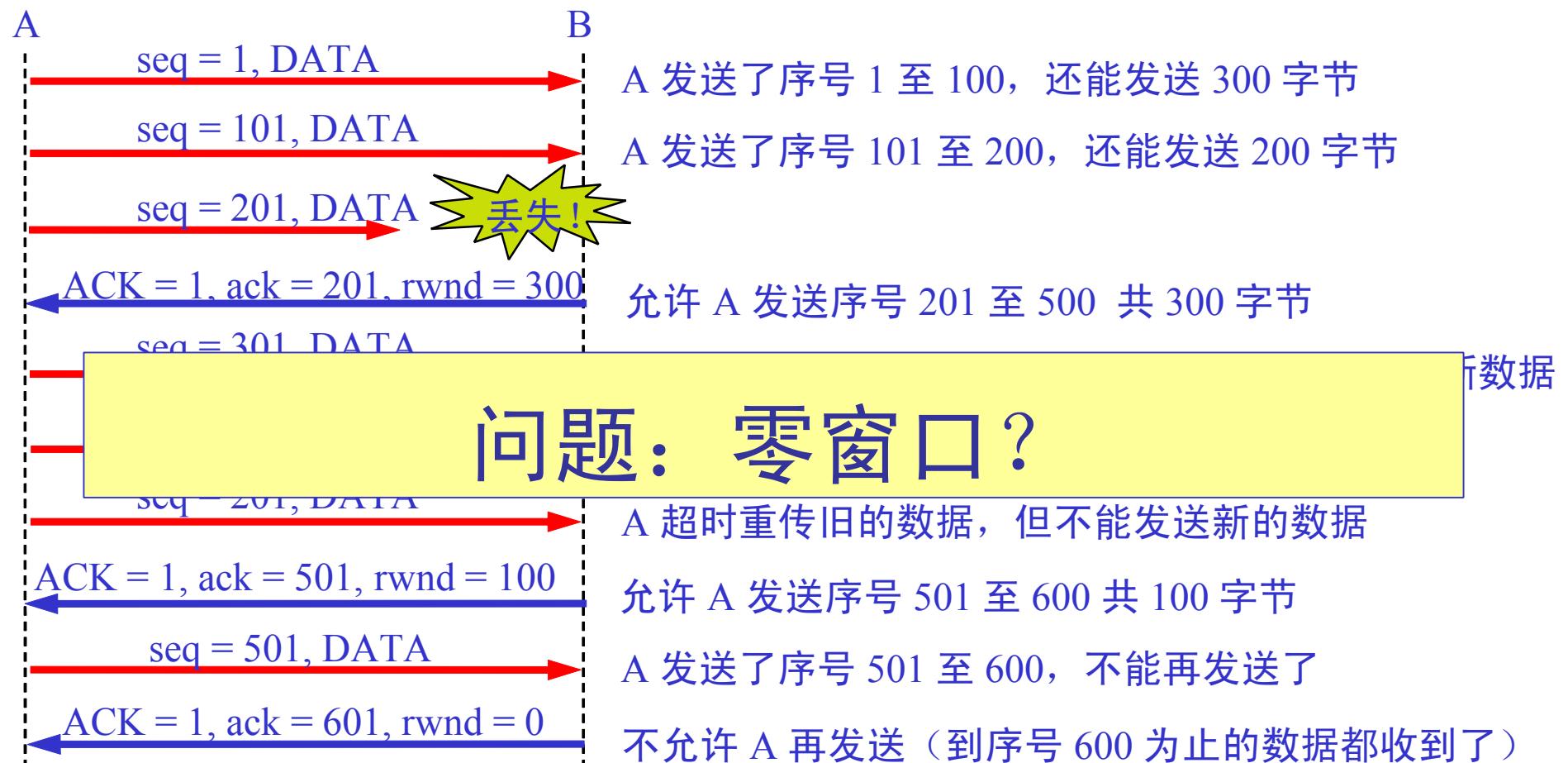
5.7 TCP 的流量控制

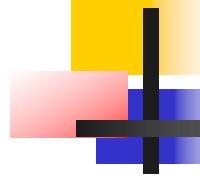
5.7.1 利用滑动窗口实现流量控制

- 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

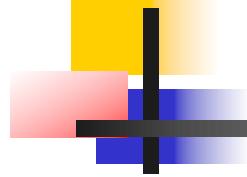
A 向 B 发送数据。在连接建立时，
B 告诉 A：“我的接收窗口 $rwnd = 400$ （字节）”。





持续计时器(persistence timer)

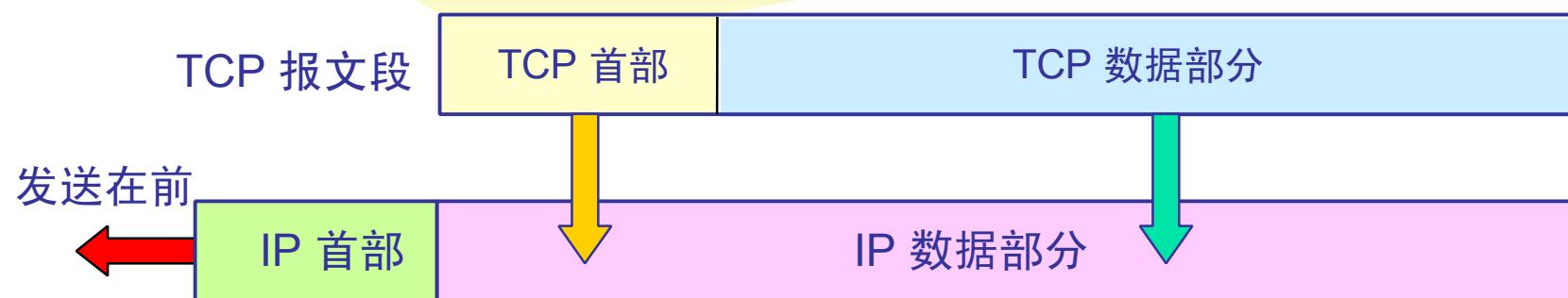
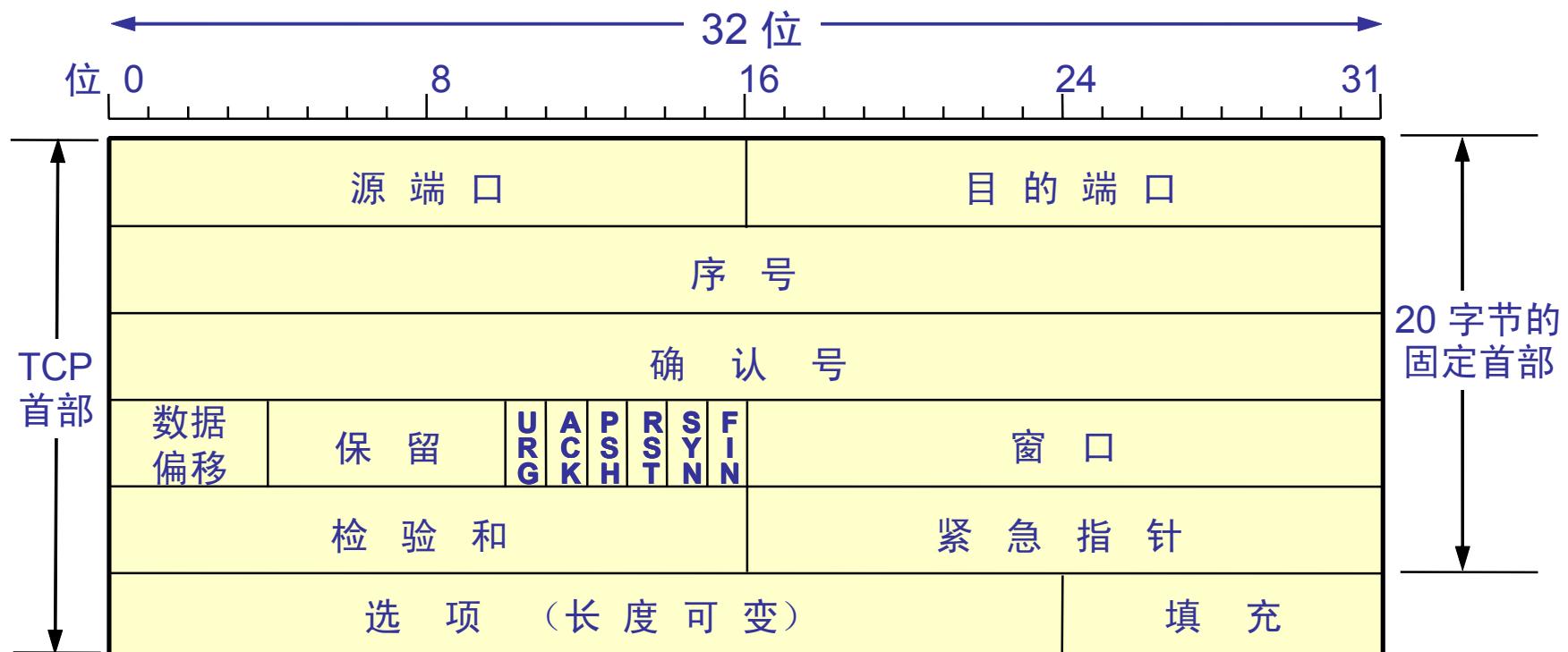
- TCP 为每一个连接设有一个**持续计时器**。
 - 只要 TCP 连接的一方收到对方的**零窗口**通知，就启动持续计时器。
 - 若持续计时器设置的时间到期，就发送一个零窗口探测报文段（仅携带 1 字节的数据），而对方就在确认这个探测报文段时给出了现在的窗口值。
 - 若窗口仍然是零，则收到这个报文段的一方就重新设置持续计时器。
 - 若窗口不是零，则死锁的僵局就可以打破了。

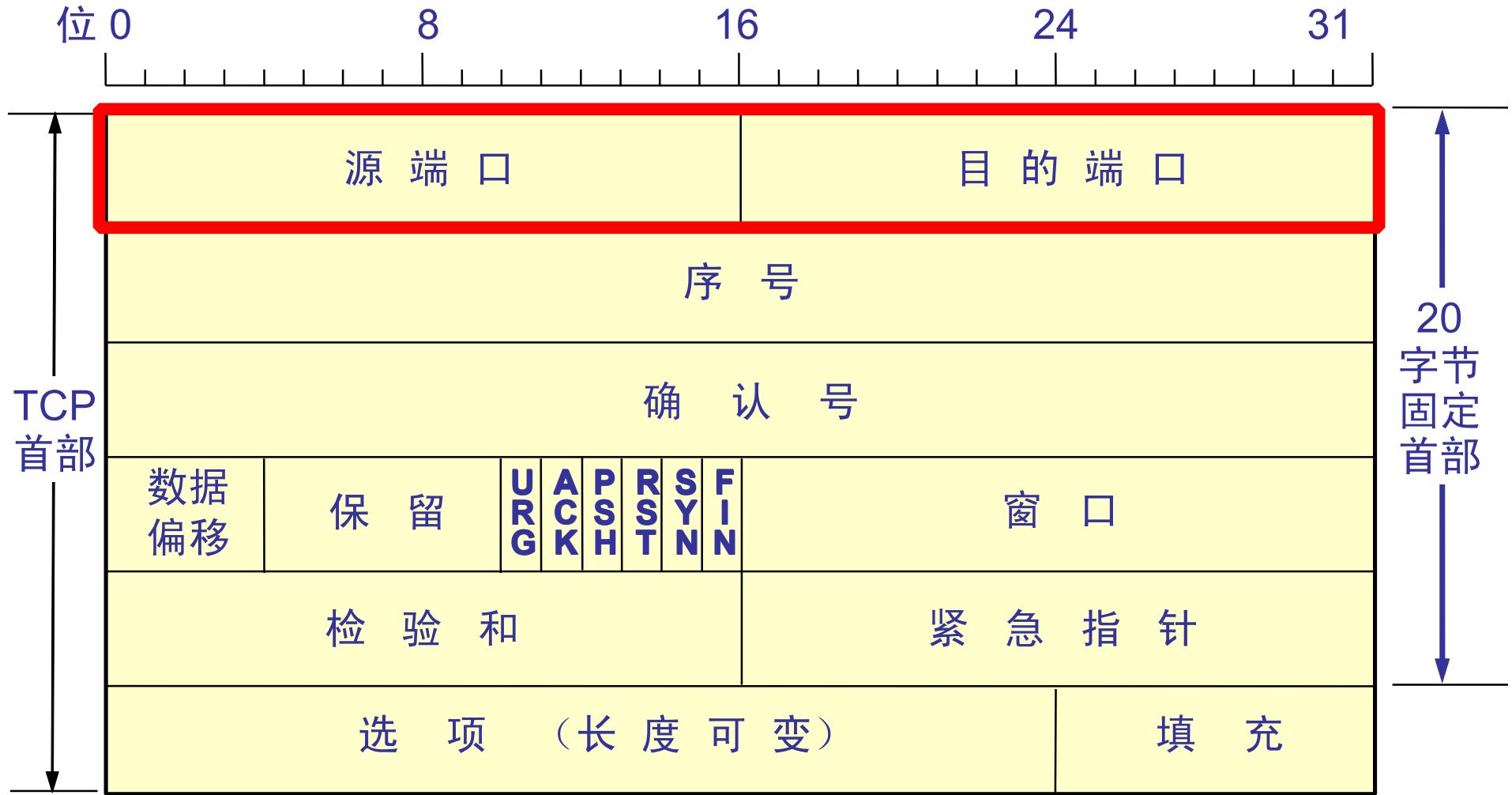


5.7.2 TCP 报文段发送时机

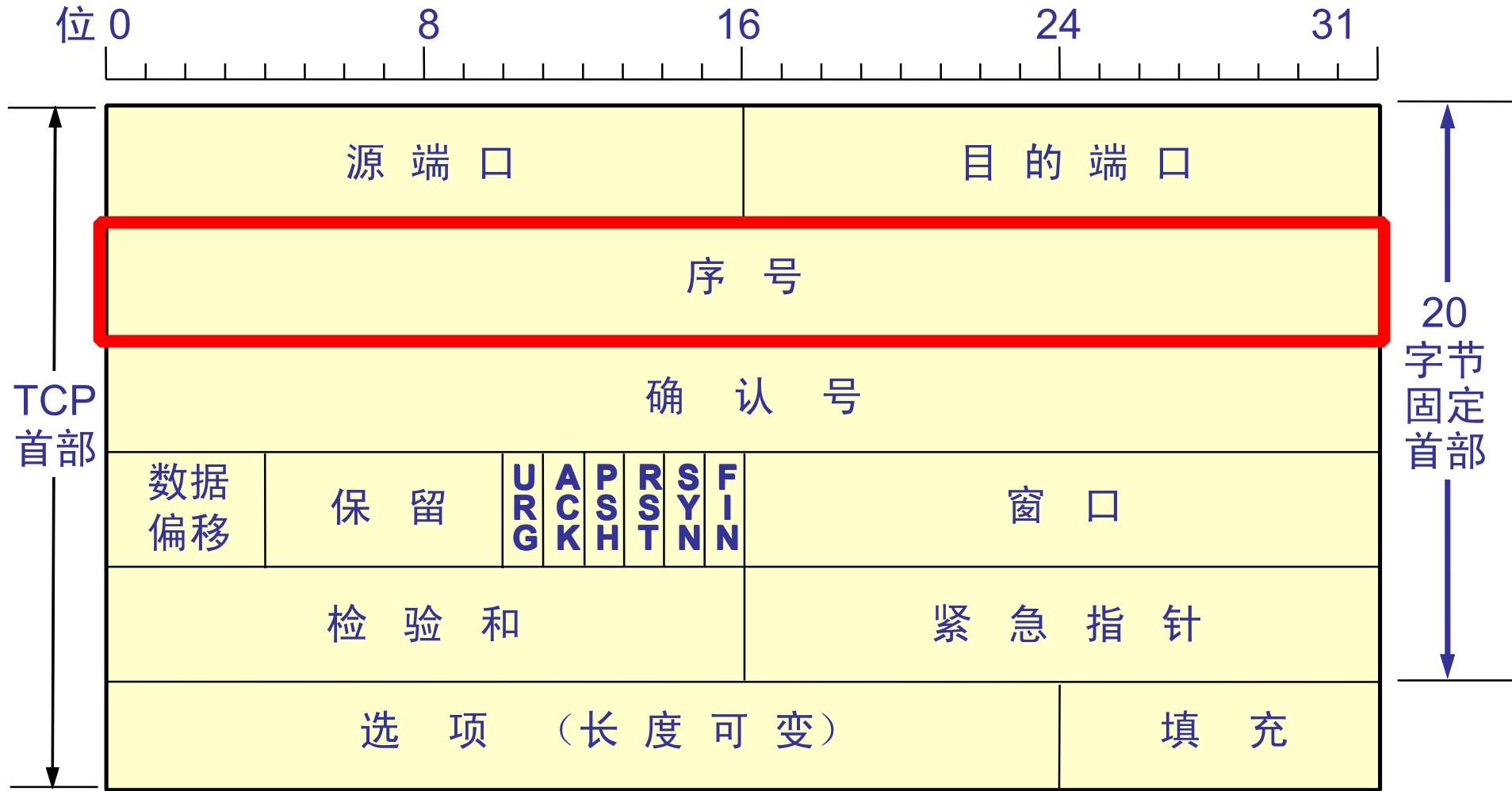
- TCP 报文段的发送时机必须考虑传输效率：
 - 第一种机制是 TCP 维持一个变量，它等于最大报文段长度 **MSS**。只要缓存中存放的数据达到 **MSS** 字节时，就组装成一个 TCP 报文段发送出去。
 - 第二种机制是由发送方的应用进程指明要求发送报文段，即 TCP 支持的推送(push)操作。
 - 第三种机制是发送方的一个计时器期限到了，这时就把当前已有的缓存数据装入报文段（但长度不能超过 **MSS**）发送出去。

5.5 TCP 报文段的头部格式

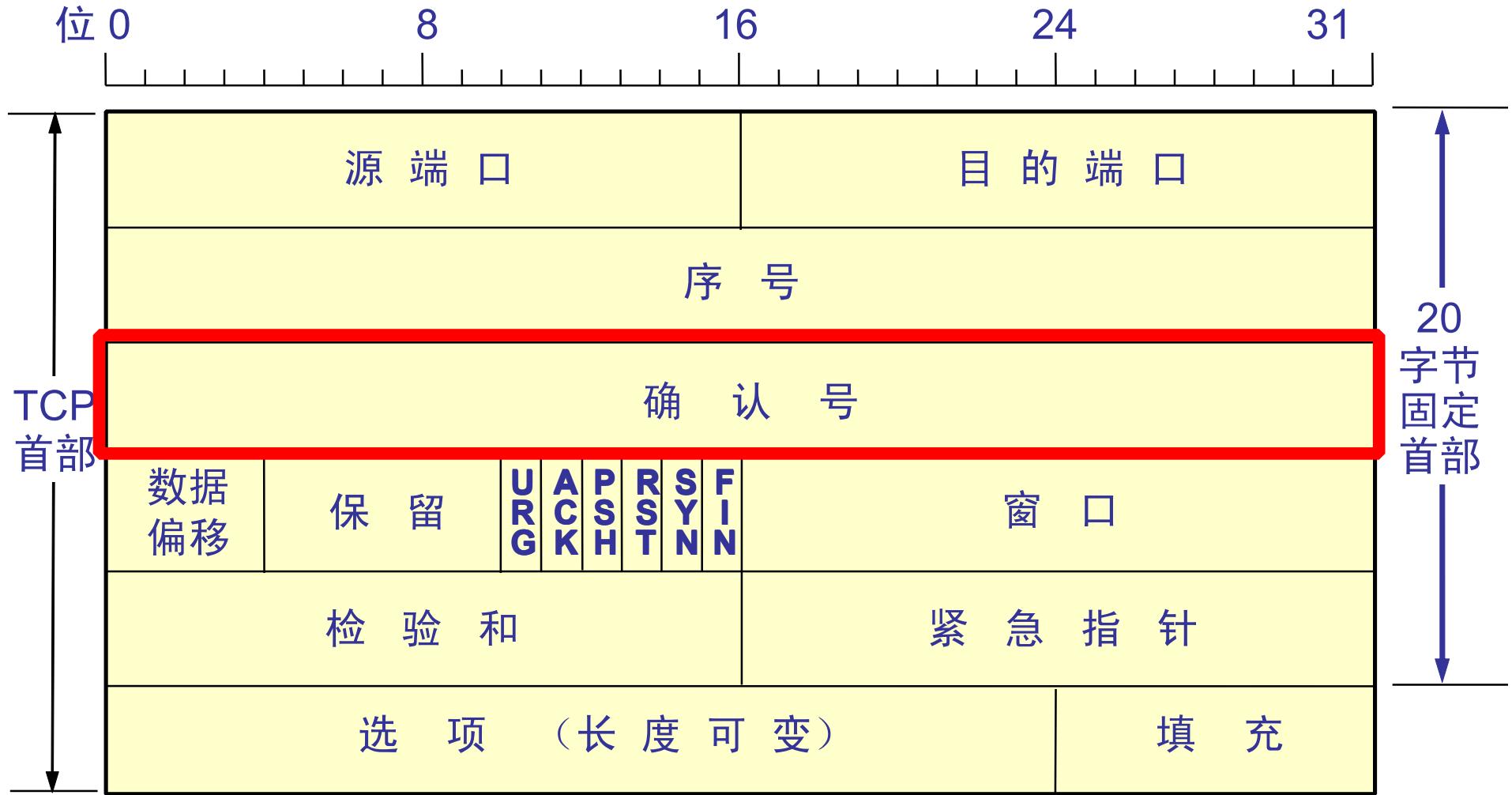




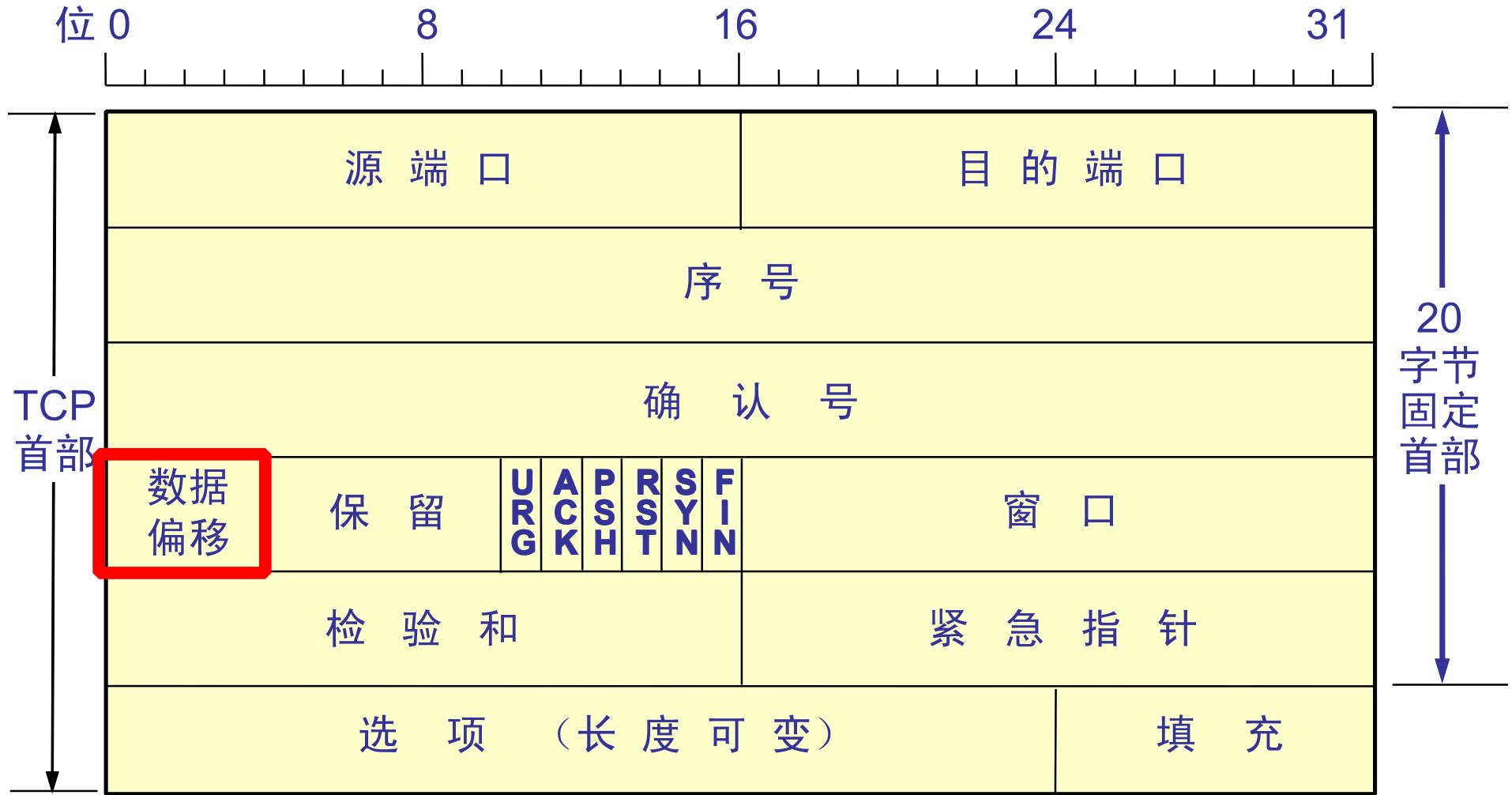
源端口和目的端口字段——各占 2 字节。端口是运输层与应用层的服务接口。运输层的复用和分用功能都要通过端口才能实现。



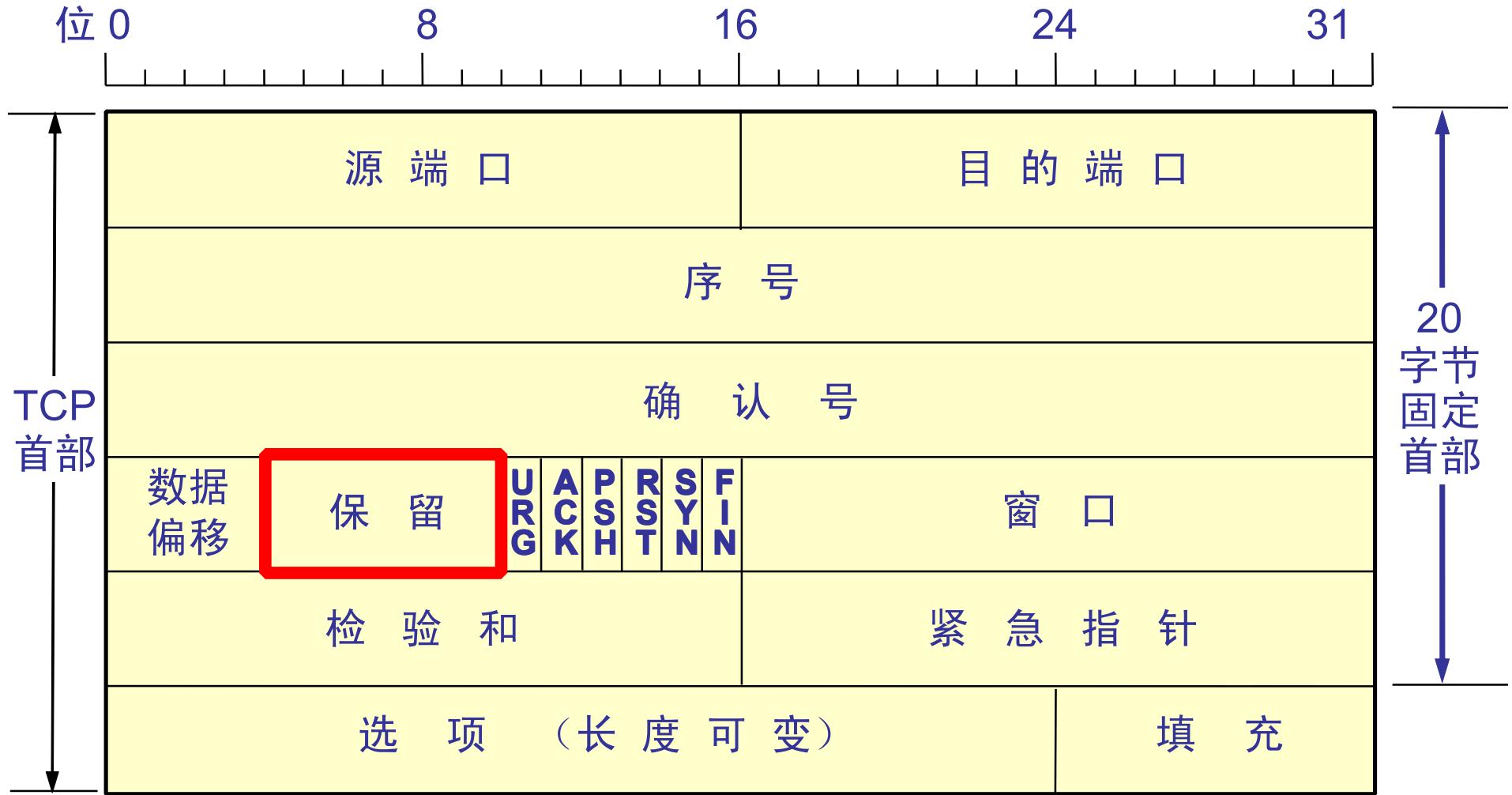
序号字段——占 4 字节。TCP 连接中传送的数据流中的每一个字节都编上一个序号。序号字段的值则指的是本报文段所发送的数据的第一个字节的序号。



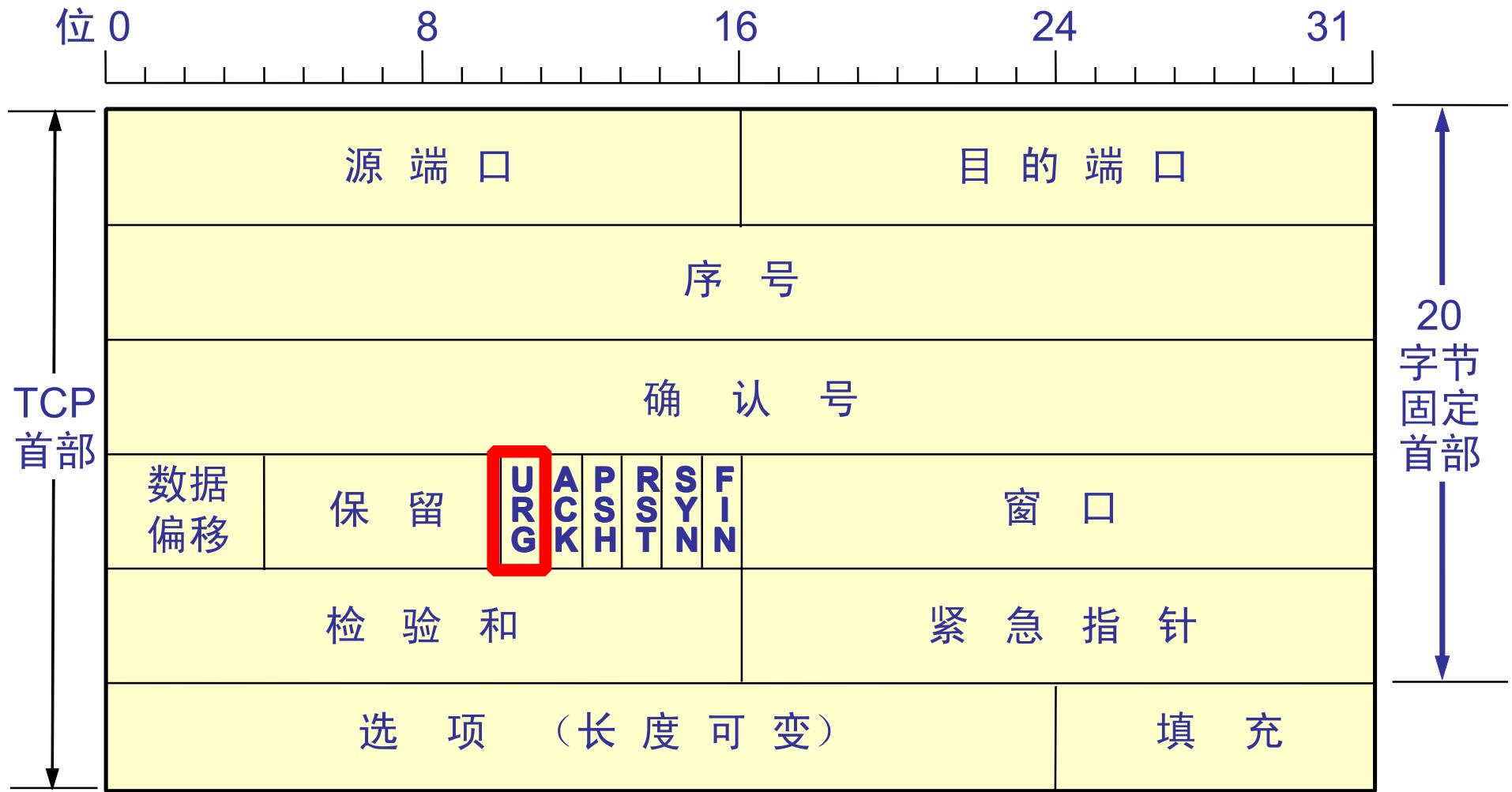
确认号字段——占 4 字节，是期望收到对方的下一个报文段的数据的第一个字节的序号。



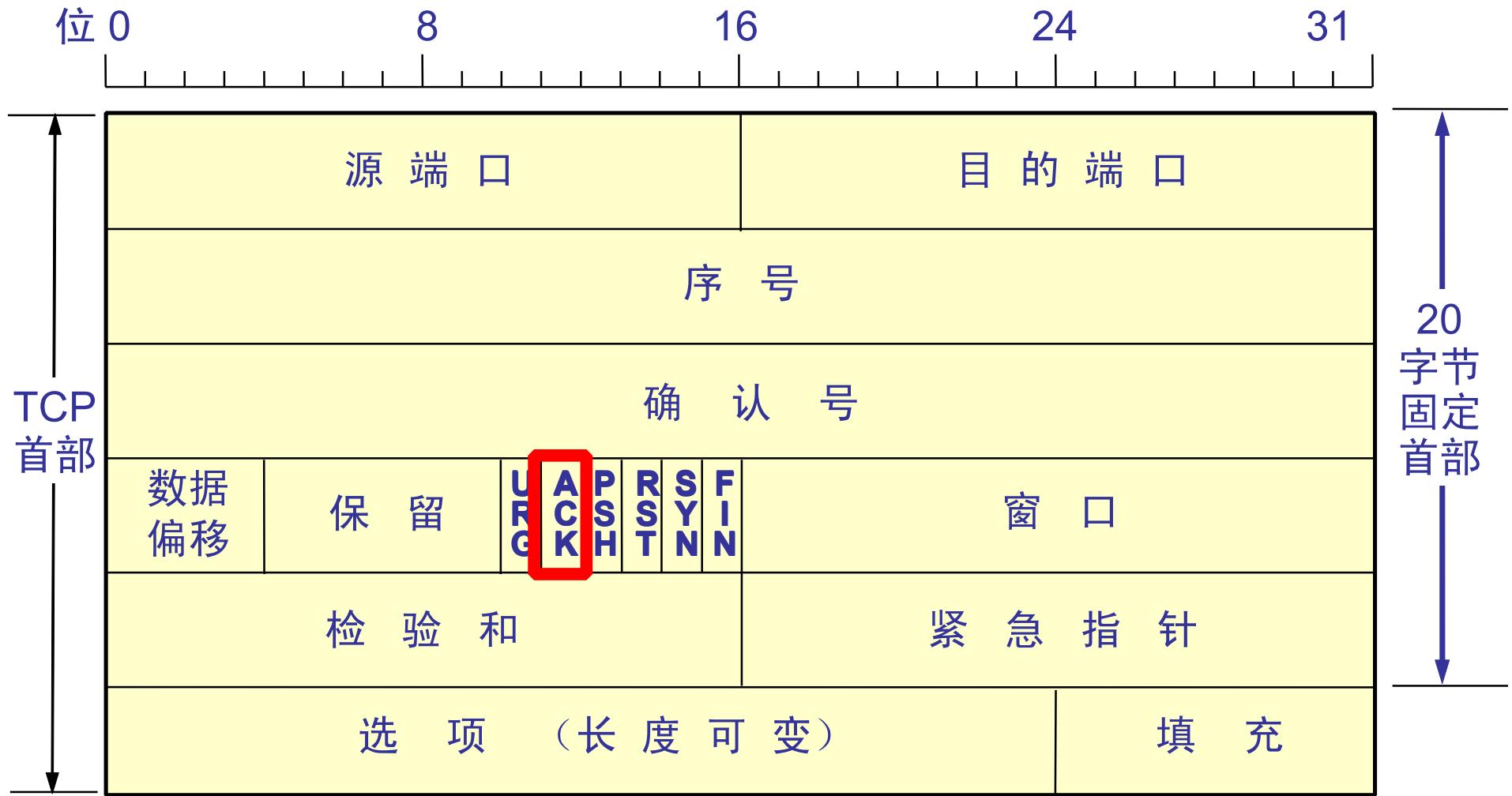
数据偏移（即首部长度）——占 4 位，它指出 TCP 报文段的数据起始处距离 TCP 报文段的起始处有多远。“数据偏移”的单位是 32 位字（以 4 字节为计算单位）。



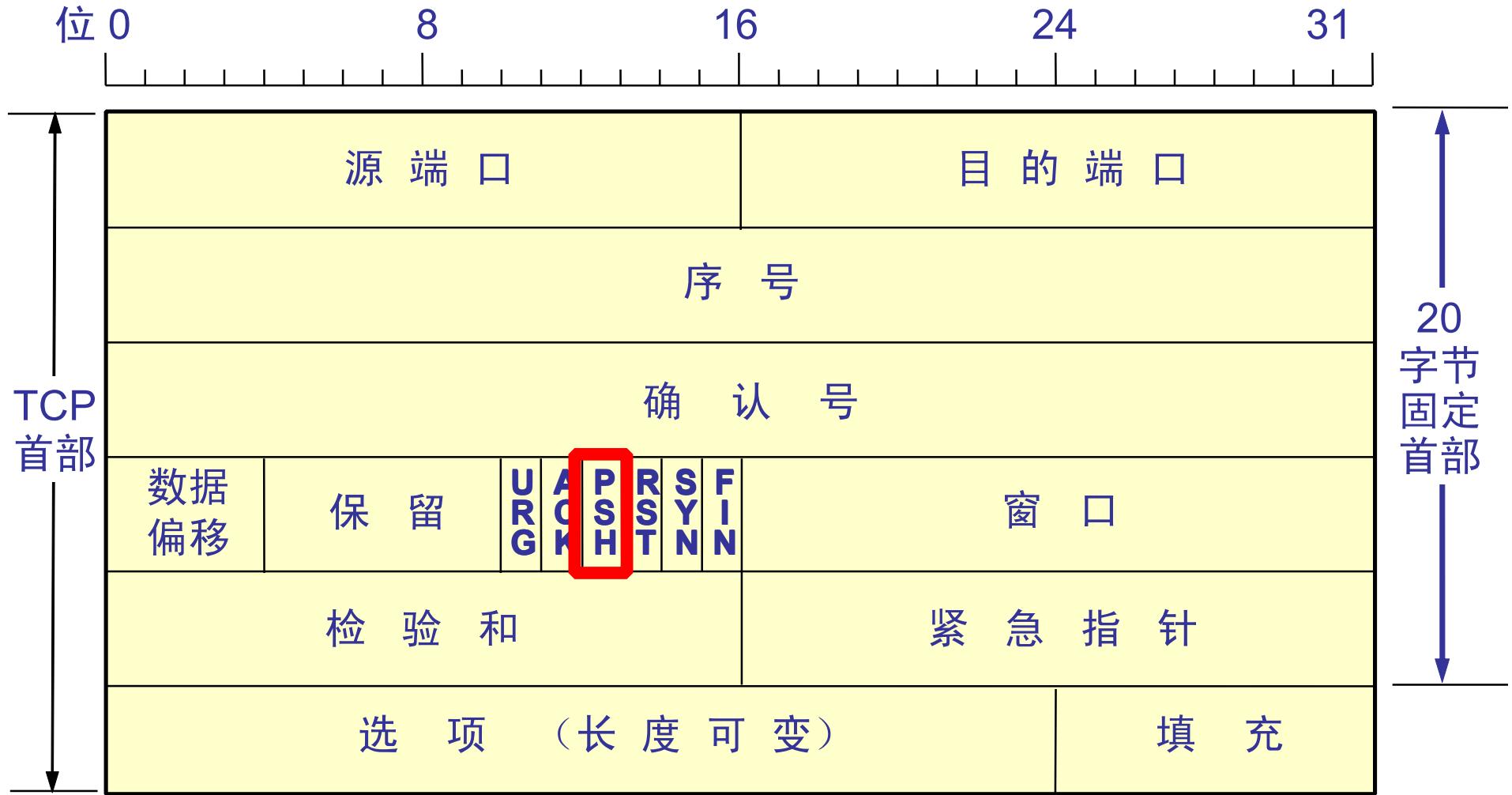
保留字段——占 6 位，保留为今后使用，但目前应置为 0。



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)。



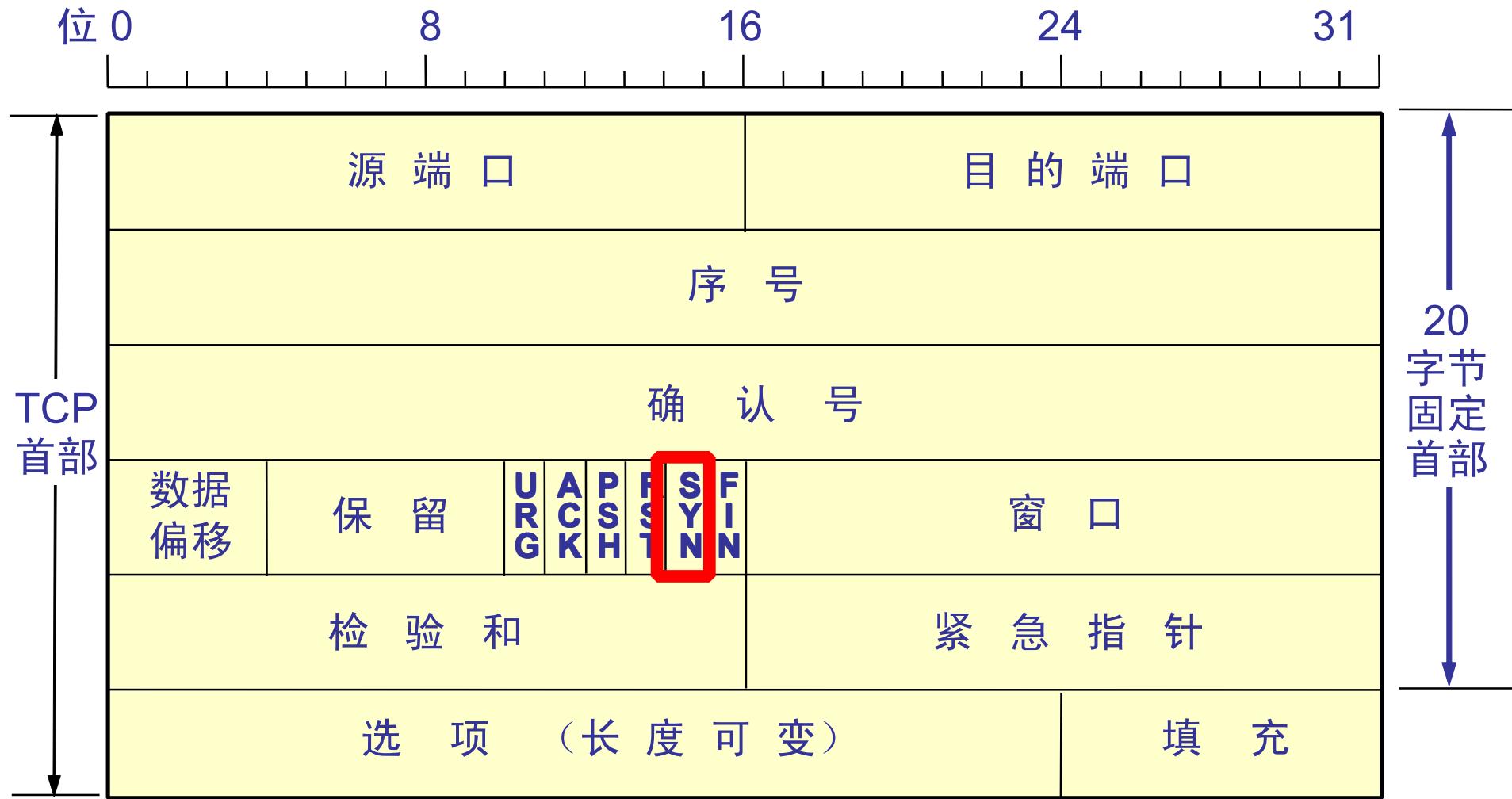
确认 ACK —— 只有当 $ACK = 1$ 时确认号字段才有效。当 $ACK = 0$ 时，确认号无效。



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。



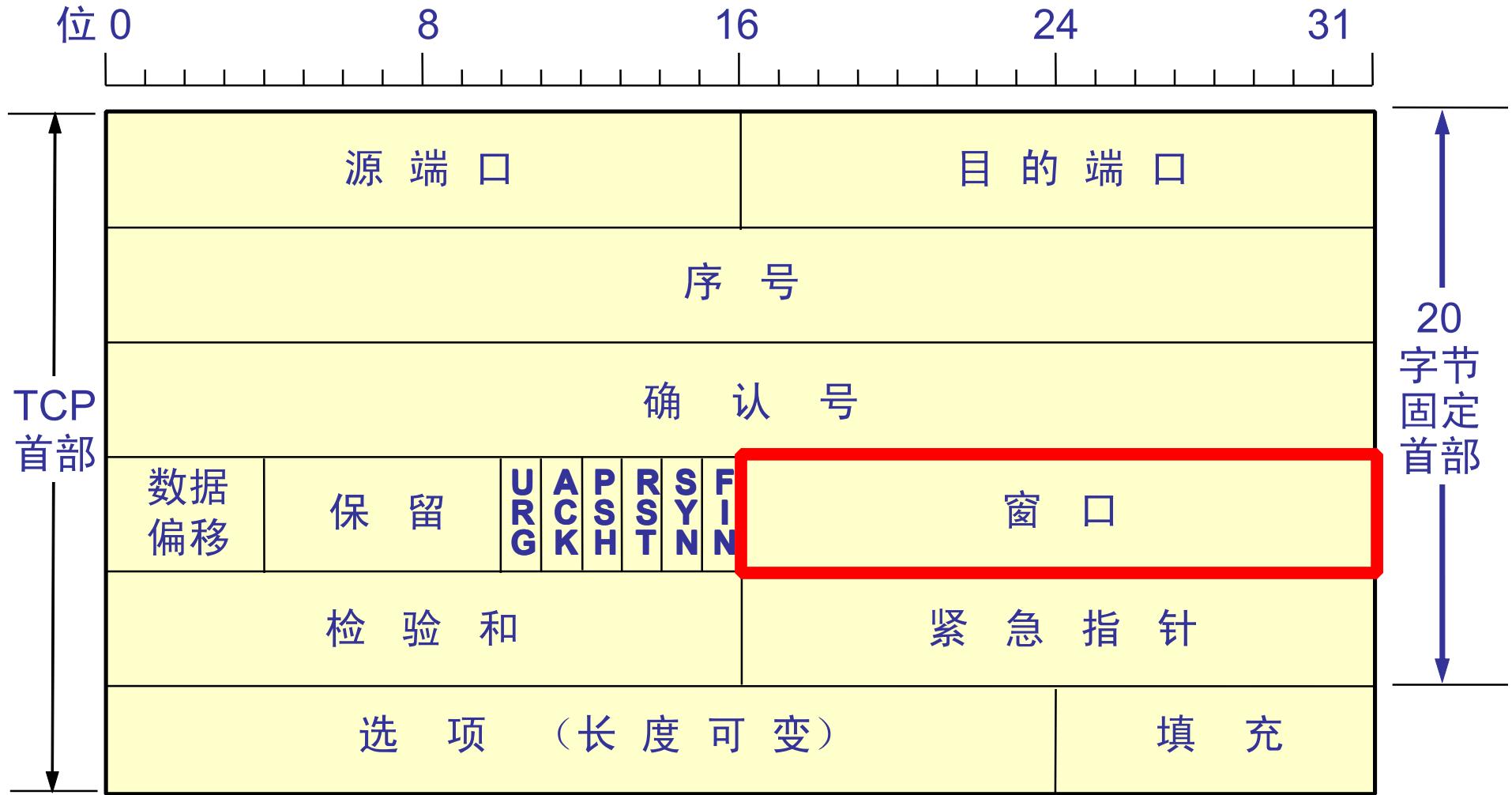
复位 RST (ReSeT) ——当 $RST = 1$ 时，表明 TCP 连接中出现严重差错（如由于主机崩溃或其他原因），必须释放连接，然后再重新建立运输连接。



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



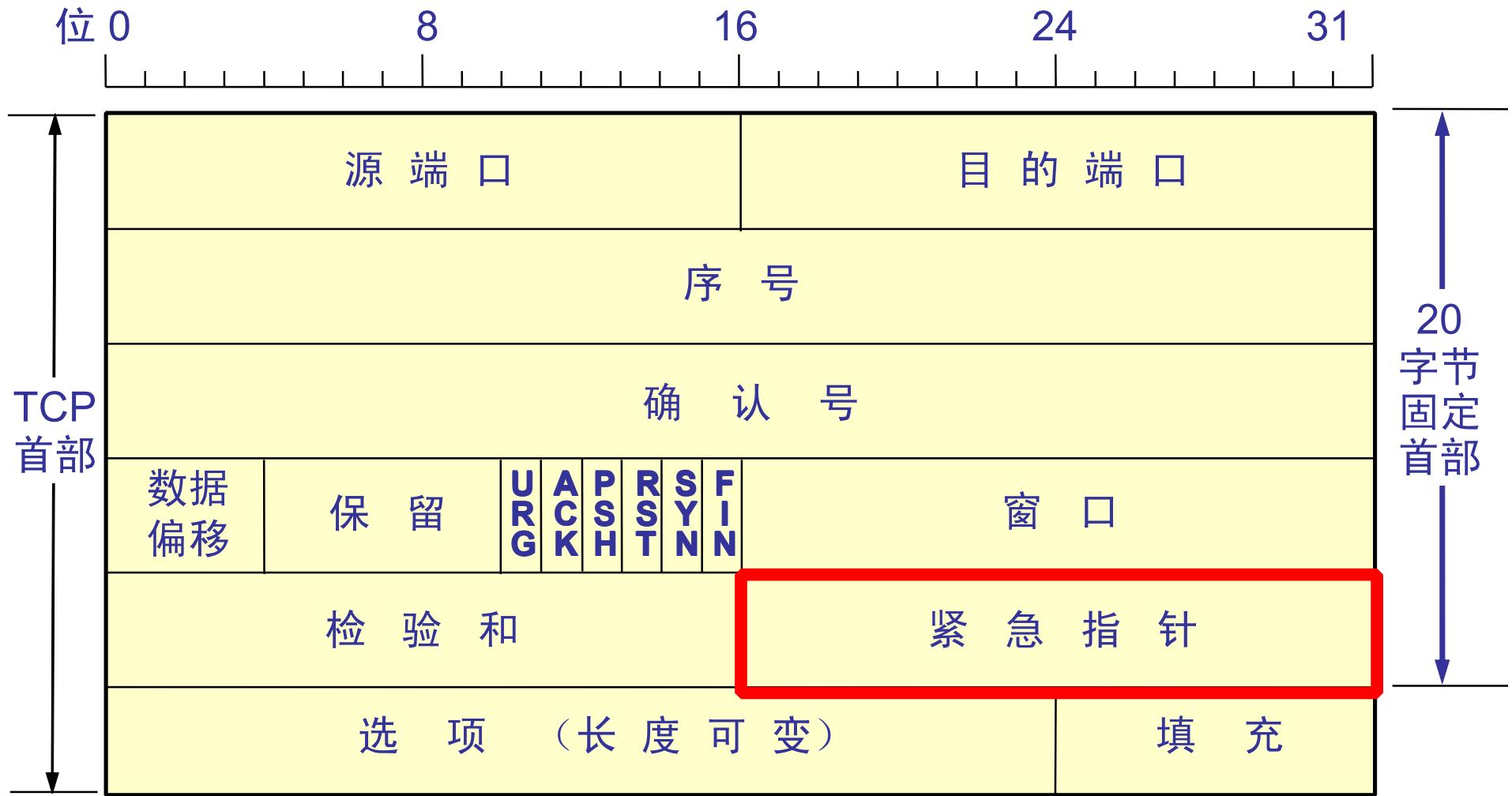
终止 FIN (FINis) —— 用来释放一个连接。FIN = 1 表明此报文段的发送端的数据已发送完毕，并要求释放运输连接。



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。

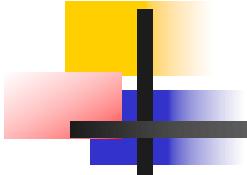


紧急指针字段 —— 占 16 位，指出在本报文段中紧急数据共有多少个字节（紧急数据放在本报文段数据的最前面）。

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。”

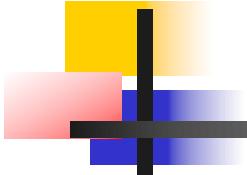


其他选项

- 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（SACK）——在 5.6.3 节介绍。



填充字段 —— 这是为了使整个首部长度是 4 字节的整数倍。



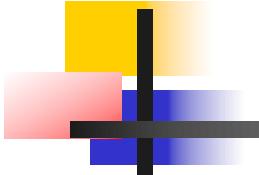
5.8 TCP的拥塞控制

5.8.1 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：

$$\text{对资源需求的总和} > \text{可用资源} \quad (5-7)$$

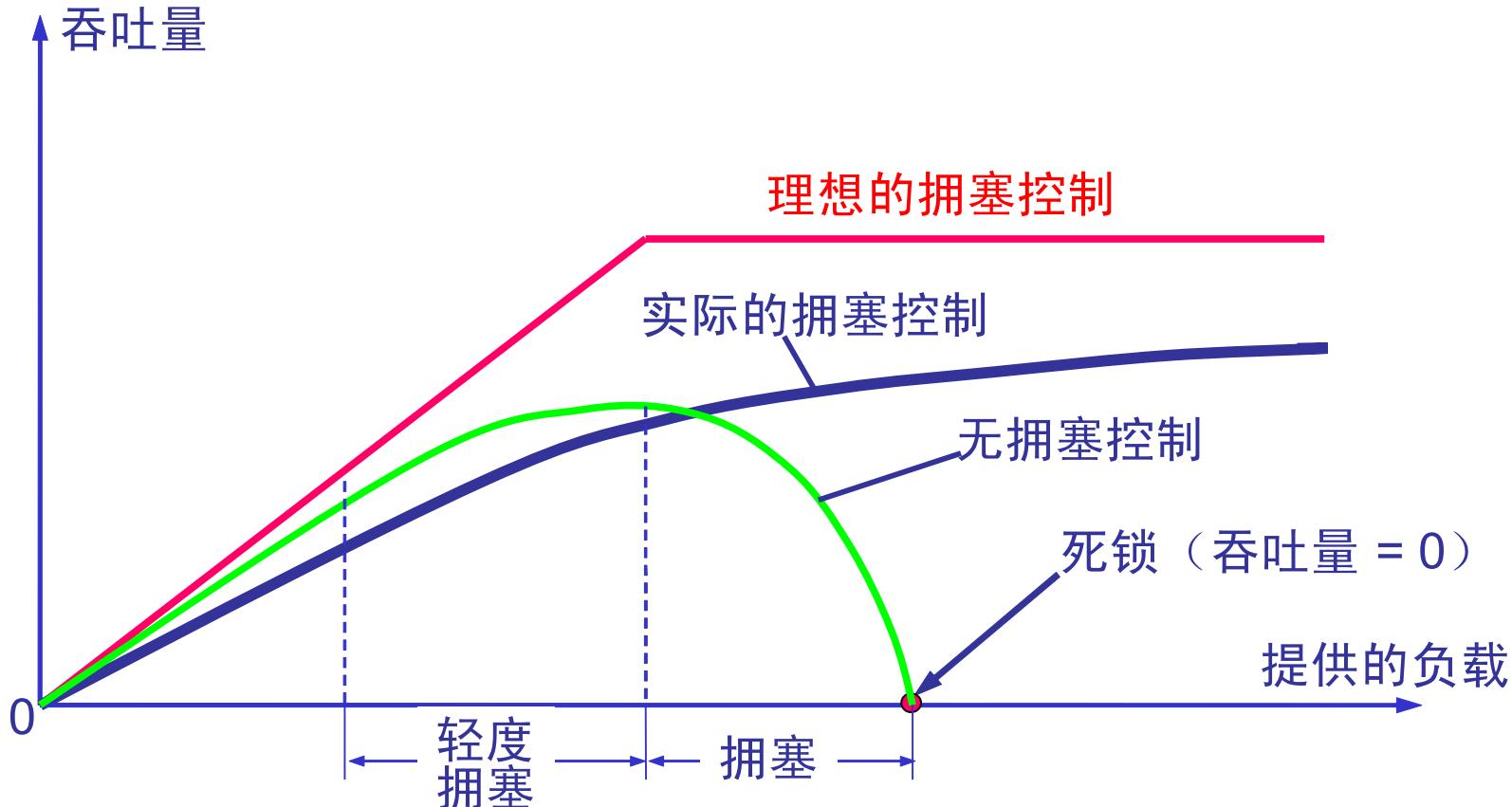
- 若网络中有许多资源同时产生拥塞，网络的性能就要明显变坏，整个网络的吞吐量将随输入负荷的增大而下降。

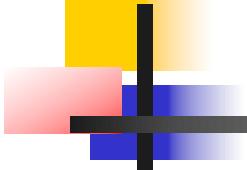


拥塞控制与流量控制的关系

- **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。

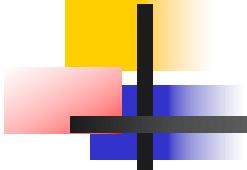
拥塞控制所起的作用





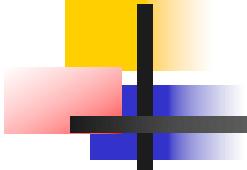
拥塞控制的一般原理

- 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。
 - 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。



开环控制和闭环控制

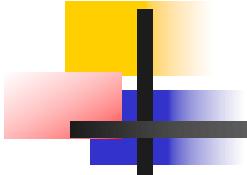
- 开环控制方法就是在设计网络时事先将有关发生拥塞的因素考虑周到，力求网络在工作时不产生拥塞。
- 闭环控制是基于反馈环路的概念。属于闭环控制的有以下几种措施：
 - 监测网络系统以便检测到拥塞在何时、何处发生。
 - 将拥塞发生的信息传送到可采取行动的地方。
 - 调整网络系统的运行以解决出现的问题。



5.8.2 拥塞控制方法

1. 慢开始和拥塞避免

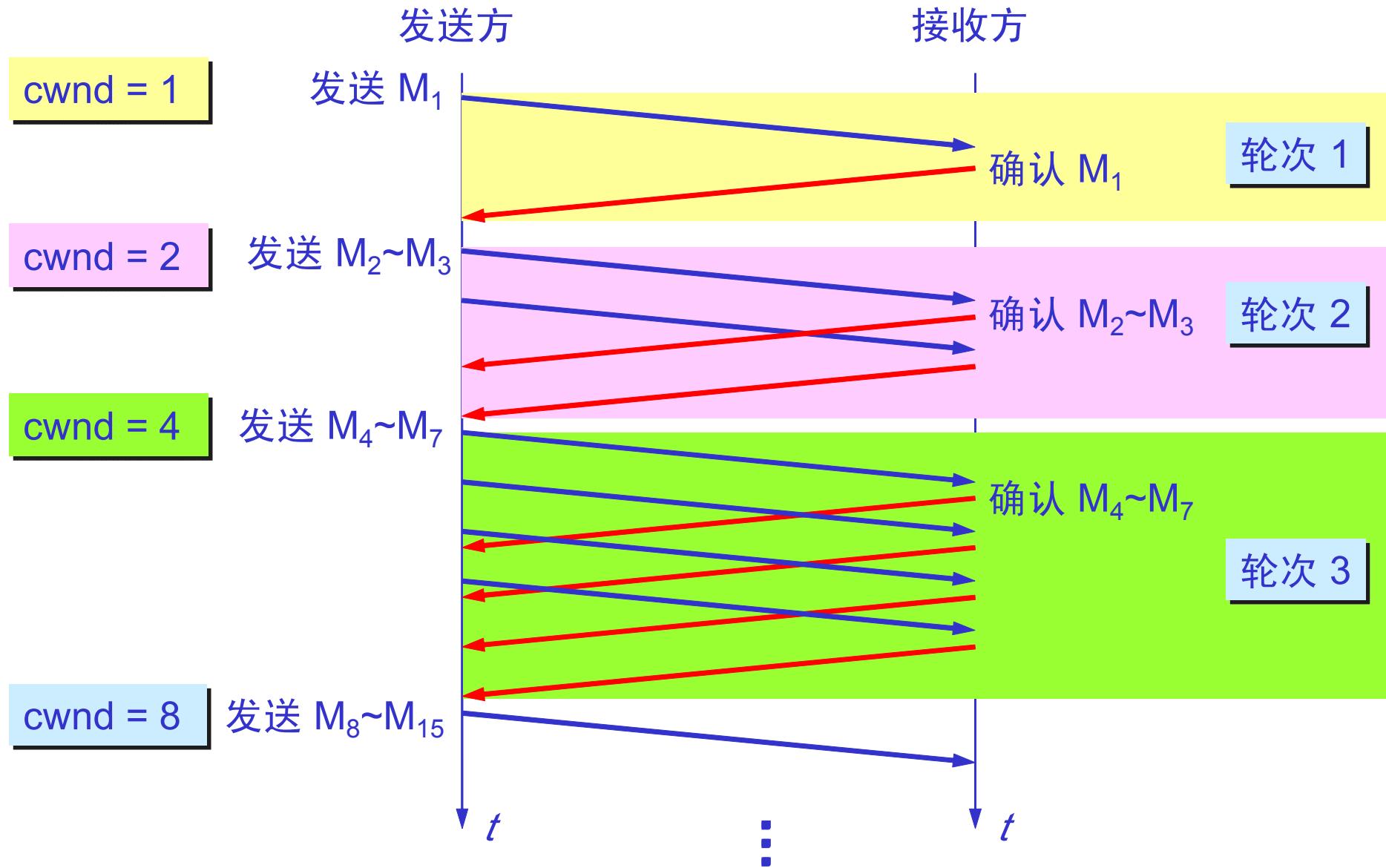
- 发送方维持一个叫做**拥塞窗口 cwnd (congestion window)**的状态变量。
 - 拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。
 - 发送方让自己的发送窗口等于拥塞窗口。如再考虑到接收方的接收能力，则发送窗口还可能小于拥塞窗口。
- 发送方控制拥塞窗口的原则是：
 - 只要网络没有出现拥塞，拥塞窗口就再增大一些，以便把更多的分组发送出去。
 - 只要网络出现拥塞，拥塞窗口就减小一些，以减少注入到网络中的分组数。

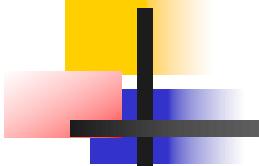


慢开始算法

- 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。
- 这样逐步增大发送端的拥塞窗口 $cwnd$ ，使分组注入到网络的速率更加合理。

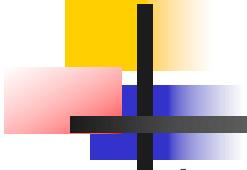
发送方每收到一个对新报文段的确认
(重传的不算在内) 就使 cwnd 加 1。





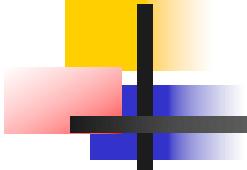
传输轮次(transmission round)

- 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**”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。
- 举例：
 - 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。



慢开始算法与拥塞避免算法

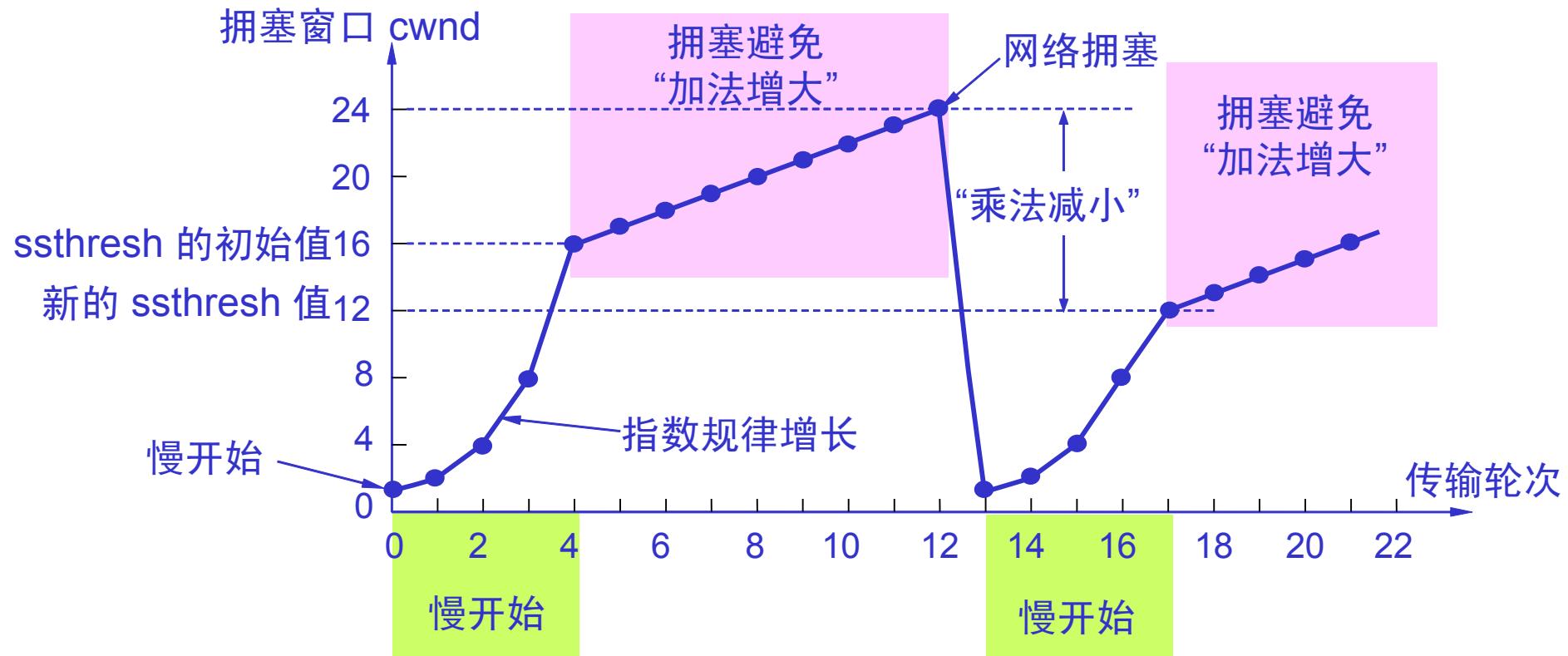
- 慢开始门限 $ssthresh$ 的用法：
 - 当 $cwnd < ssthresh$ 时，使用慢开始算法。
 - 当 $cwnd > ssthresh$ 时，停止使用慢开始算法而改用拥塞避免算法。
 - 当 $cwnd = ssthresh$ 时，既可使用慢开始算法，也可使用拥塞避免算法。
- 拥塞避免避免算法
 - 让拥塞窗口 $cwnd$ 缓慢地增大，即每经过一个往返时间 RTT 就把发送方的拥塞窗口 $cwnd$ 加 1，而不是加倍，使拥塞窗口 $cwnd$ 按线性规律缓慢增长。



当网络出现拥塞

- 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，就把慢开始门限 $ssthresh$ 设置为出现拥塞时的发送方窗口值的一半（但不能小于2）。
 - 判断网络拥塞的根据是没有按时收到确认
- 把拥塞窗口 $cwnd$ 重新设置为 1，执行慢开始算法。
 - 目的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。

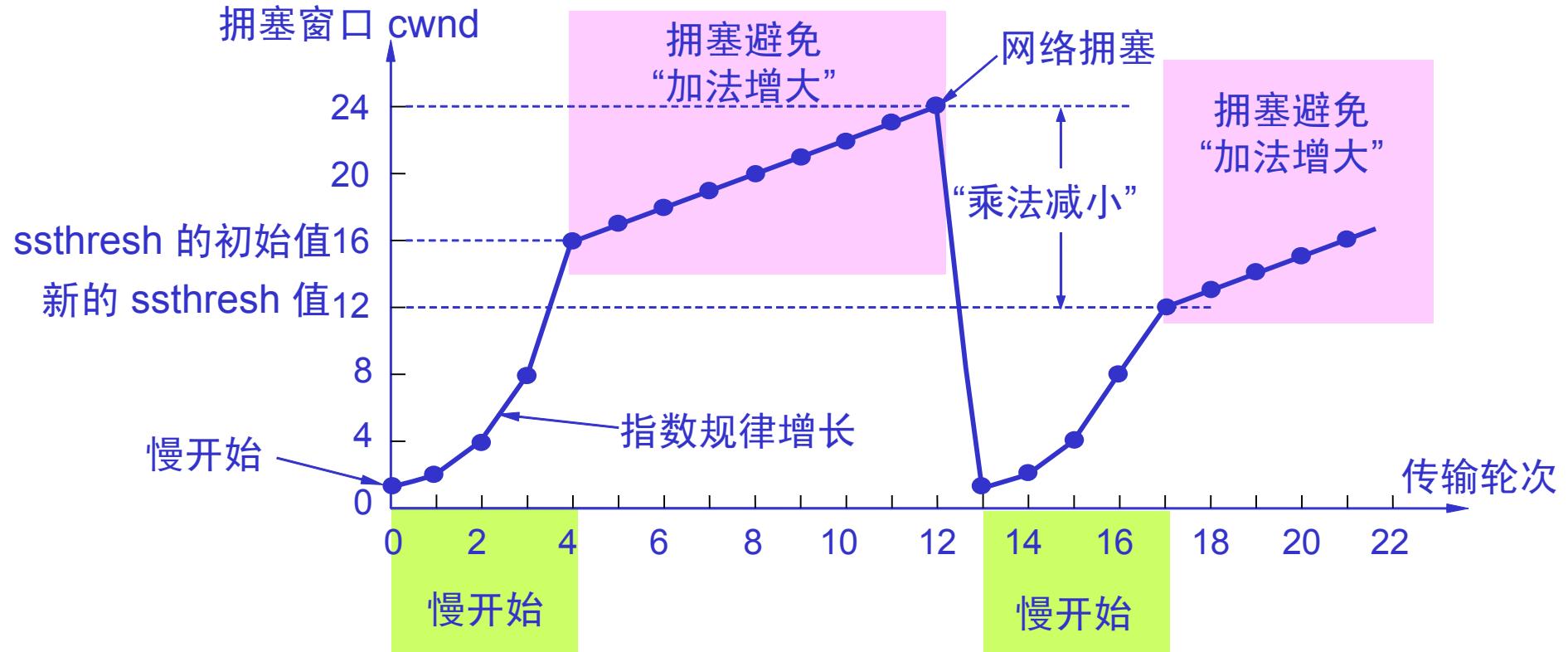
慢开始和拥塞避免算法的实现举例



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。

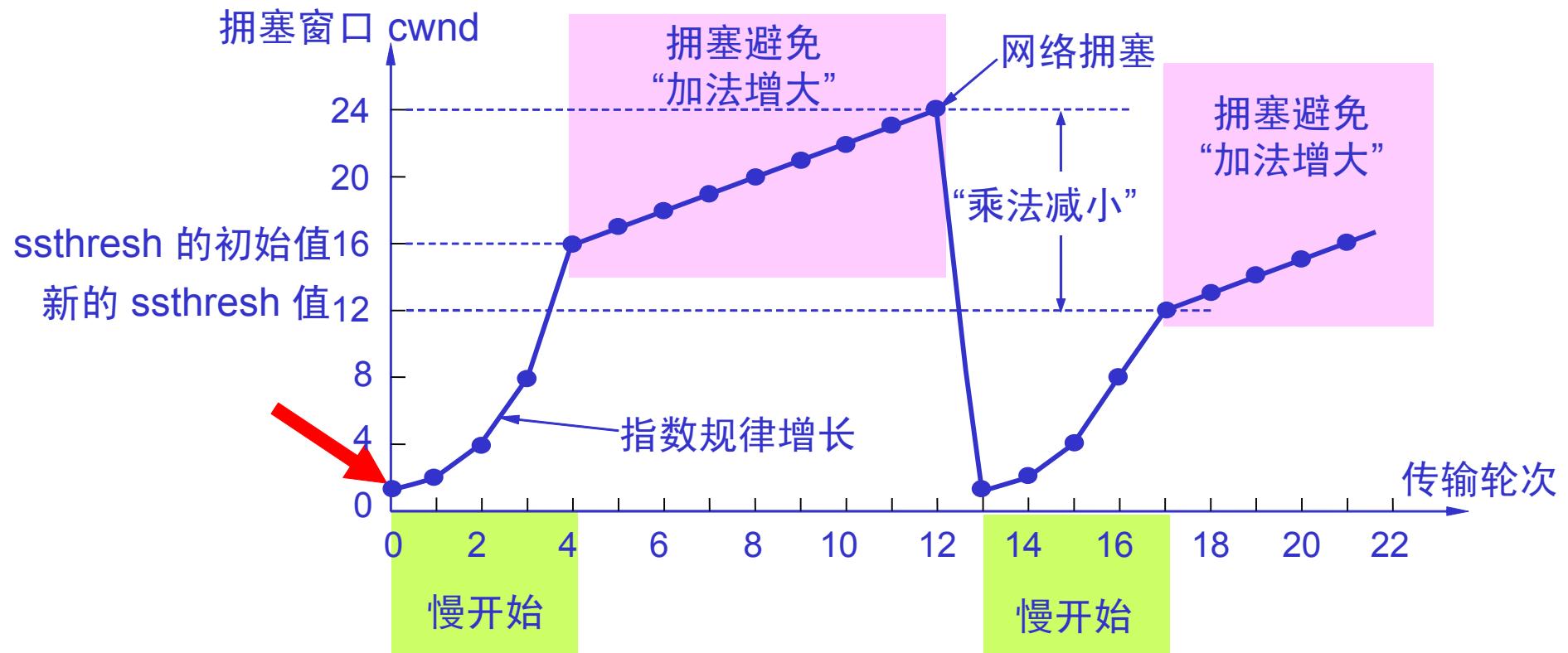
慢开始门限的初始值设置为 16 个报文段，即 $ssthresh = 16$ 。

慢开始和拥塞避免算法的实现举例



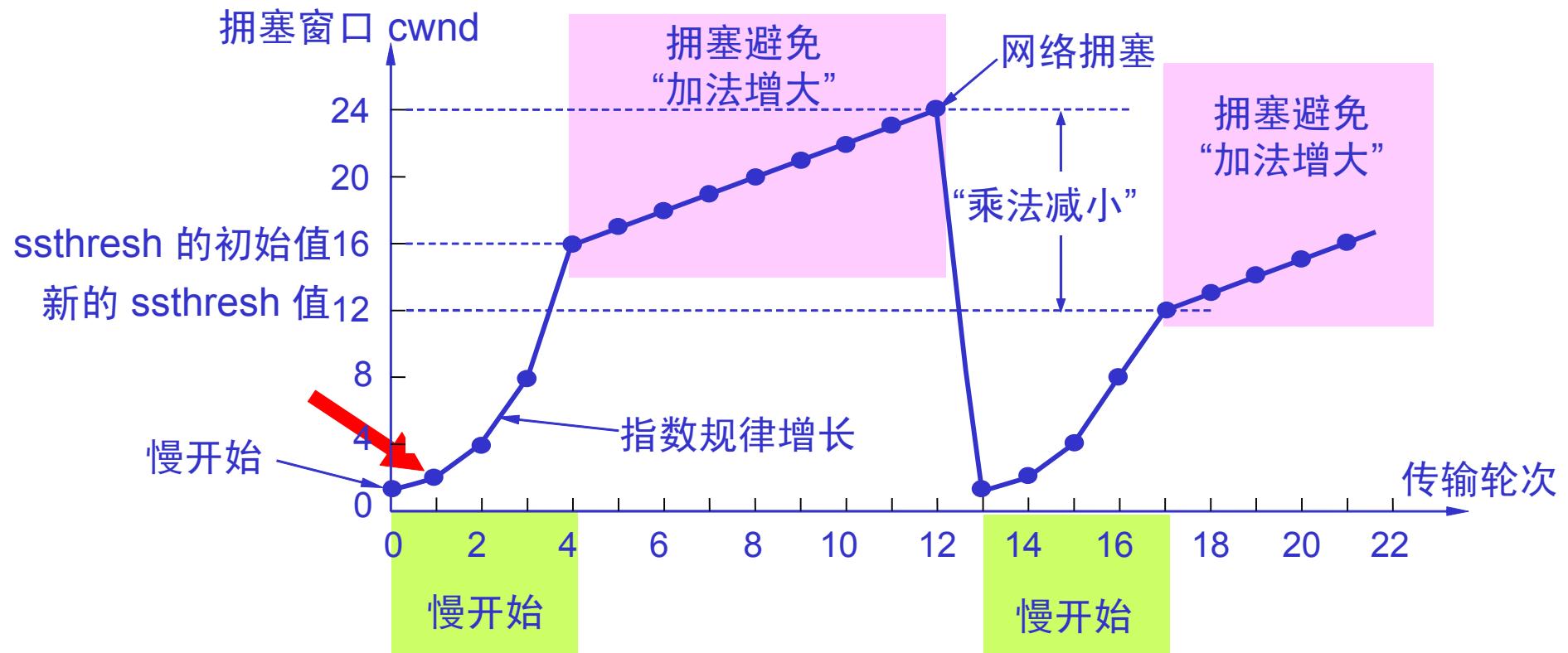
发送端的发送窗口不能超过拥塞窗口 cwnd 和接收端窗口 rwnd 中的最小值。我们假定接收端窗口足够大，因此现在发送窗口的数值等于拥塞窗口的数值。

慢开始和拥塞避免算法的实现举例



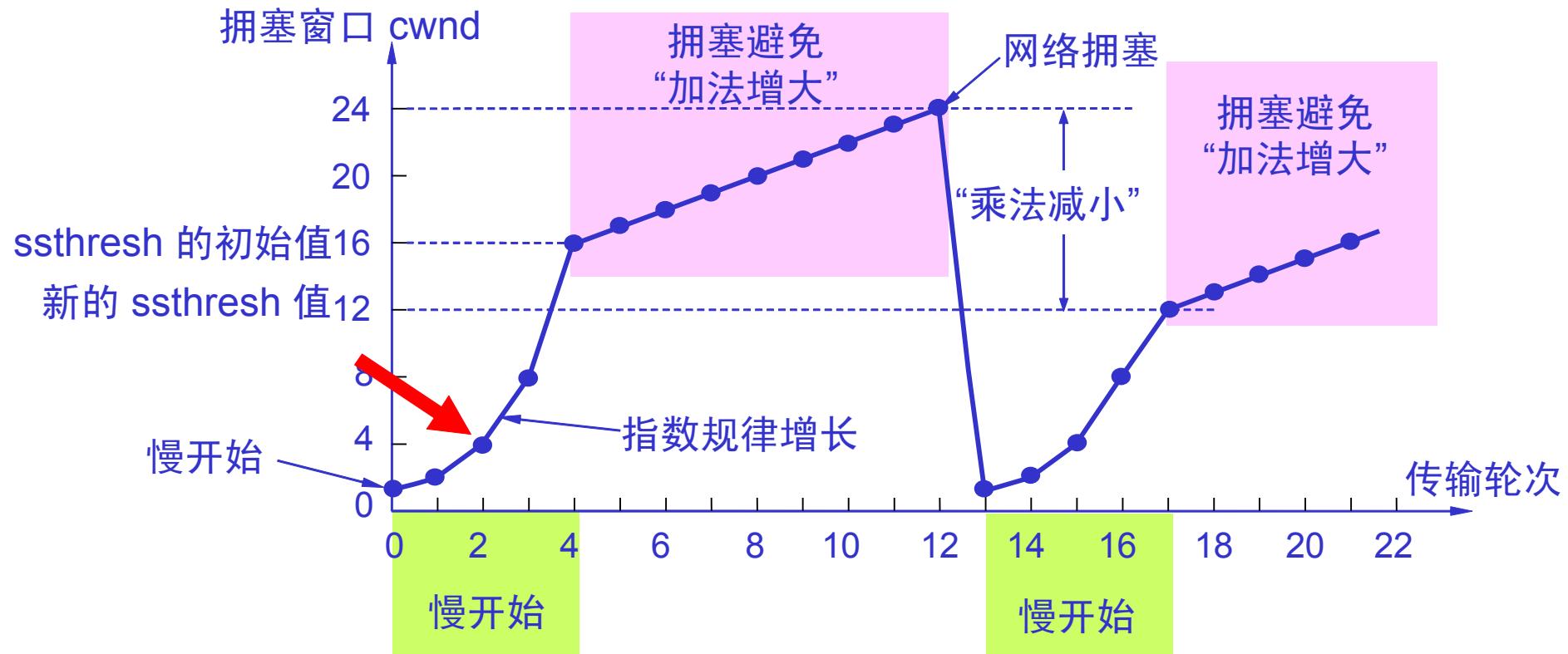
在执行慢开始算法时，拥塞窗口 cwnd 的初始值为 1，发送第一个报文段 M_0 。

慢开始和拥塞避免算法的实现举例



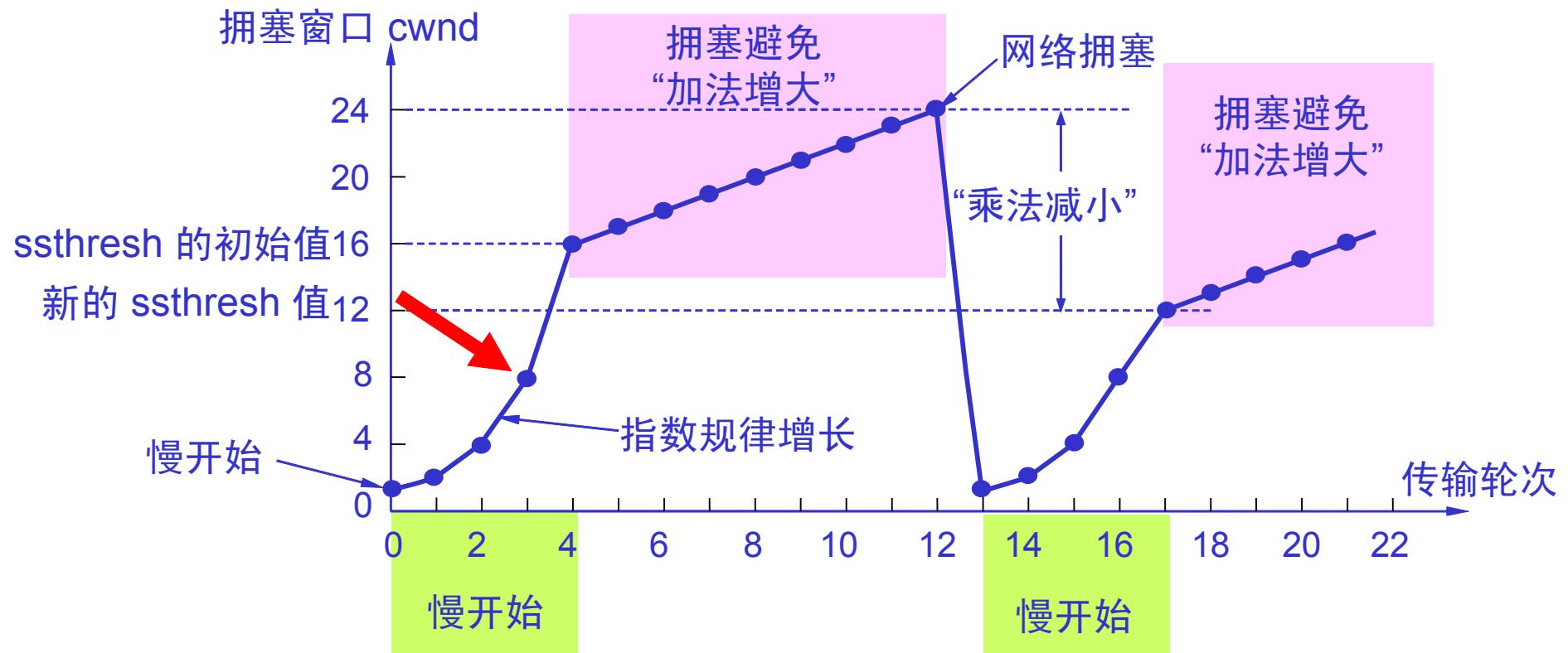
发送端每收到一个确认，就把 cwnd 加 1。于是发送端可以接着发送 M_1 和 M_2 两个报文段。

慢开始和拥塞避免算法的实现举例



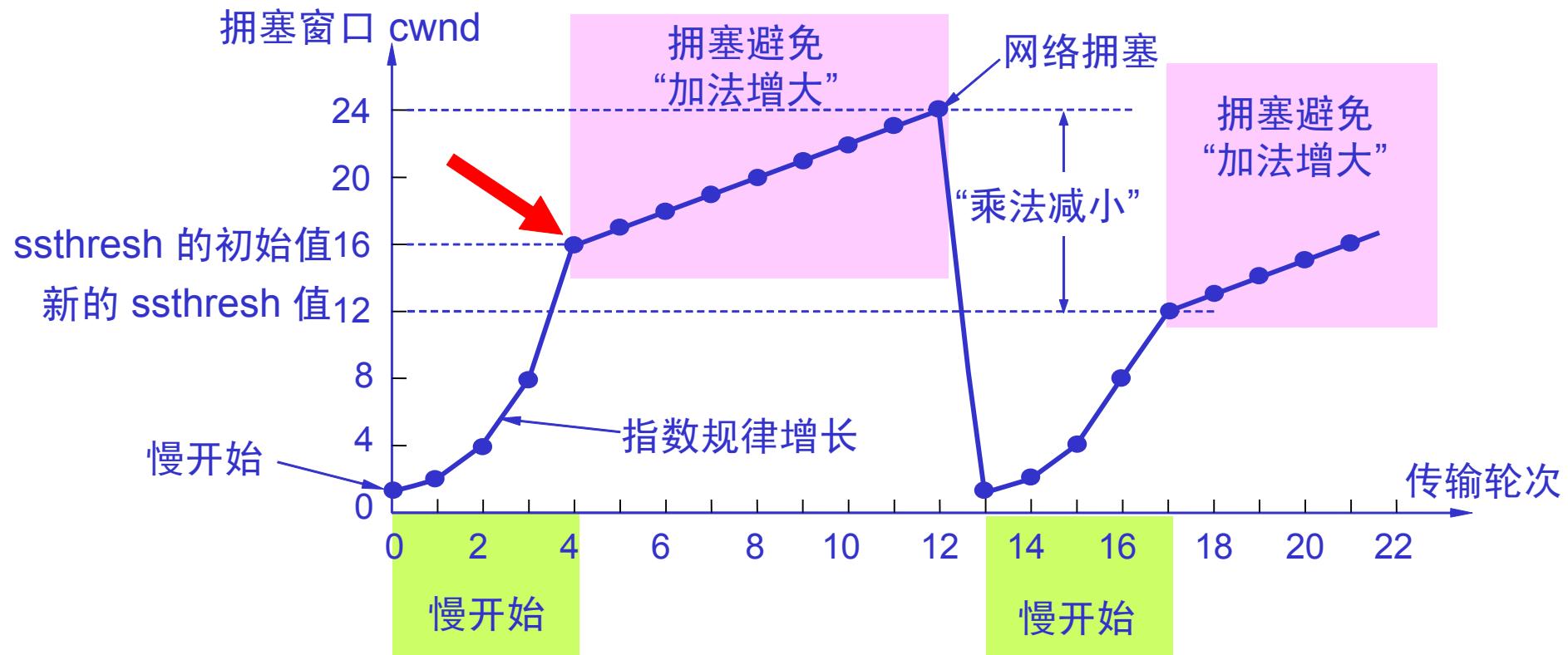
接收端共发回两个确认。发送端每收到一个对新报文段的确认，就把发送端的 cwnd 加 1。现在 cwnd 从 2 增大到 4，并可接着发送后面的 4 个报文段。

慢开始和拥塞避免算法的实现举例



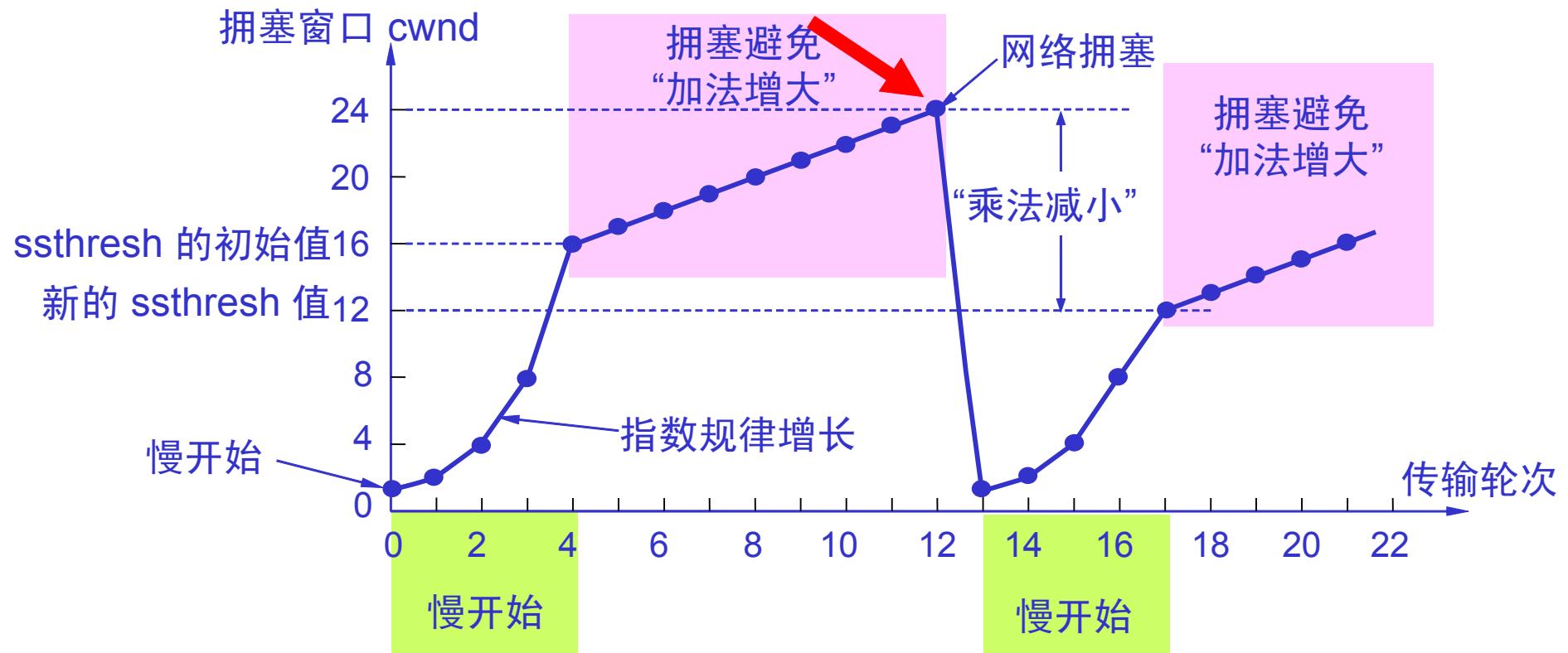
发送端每收到一个对新报文段的确认，就把发送端的拥塞窗口加 1，因此拥塞窗口 cwnd 随着传输轮次按指数规律增长。

慢开始和拥塞避免算法的实现举例



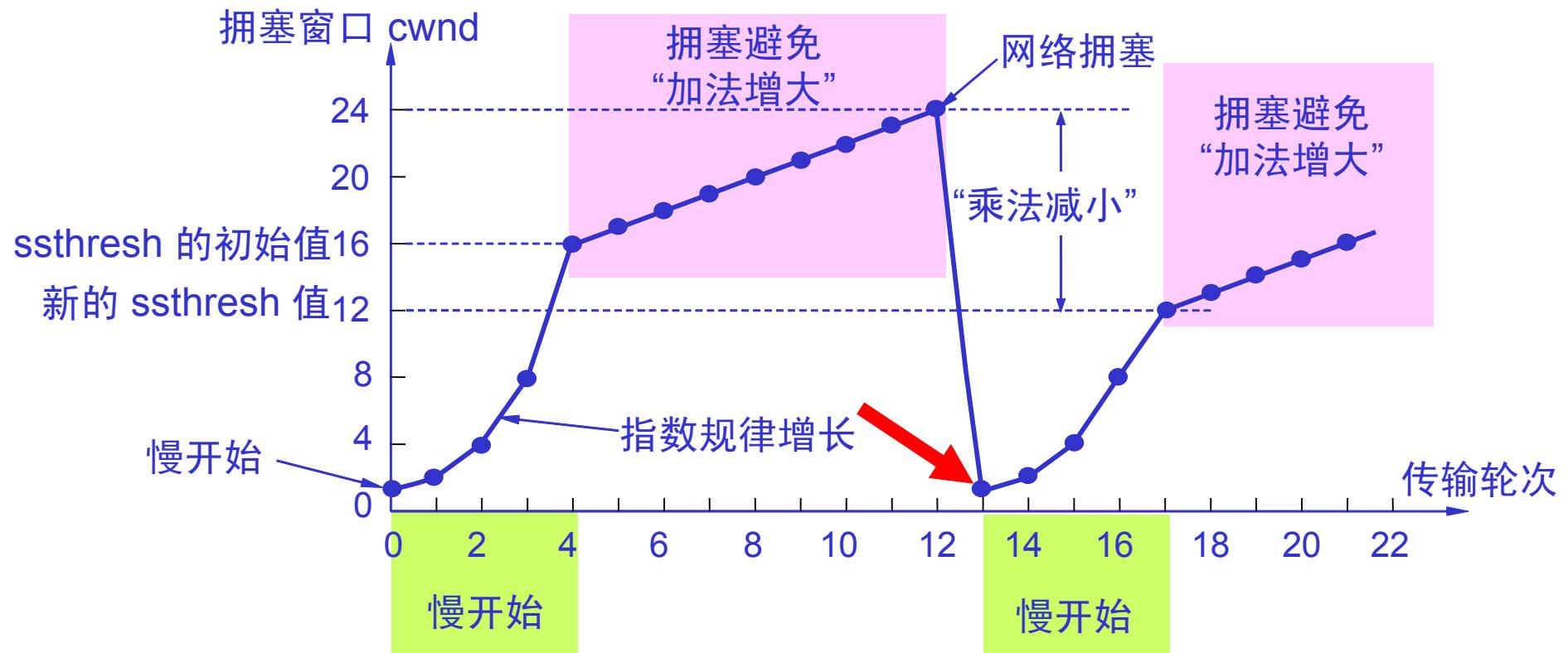
当拥塞窗口 cwnd 增长到慢开始门限值 ssthresh 时
(即当 $cwnd = 16$ 时)，就改为执行拥塞避免算法，
拥塞窗口按线性规律增长。

慢开始和拥塞避免算法的实现举例



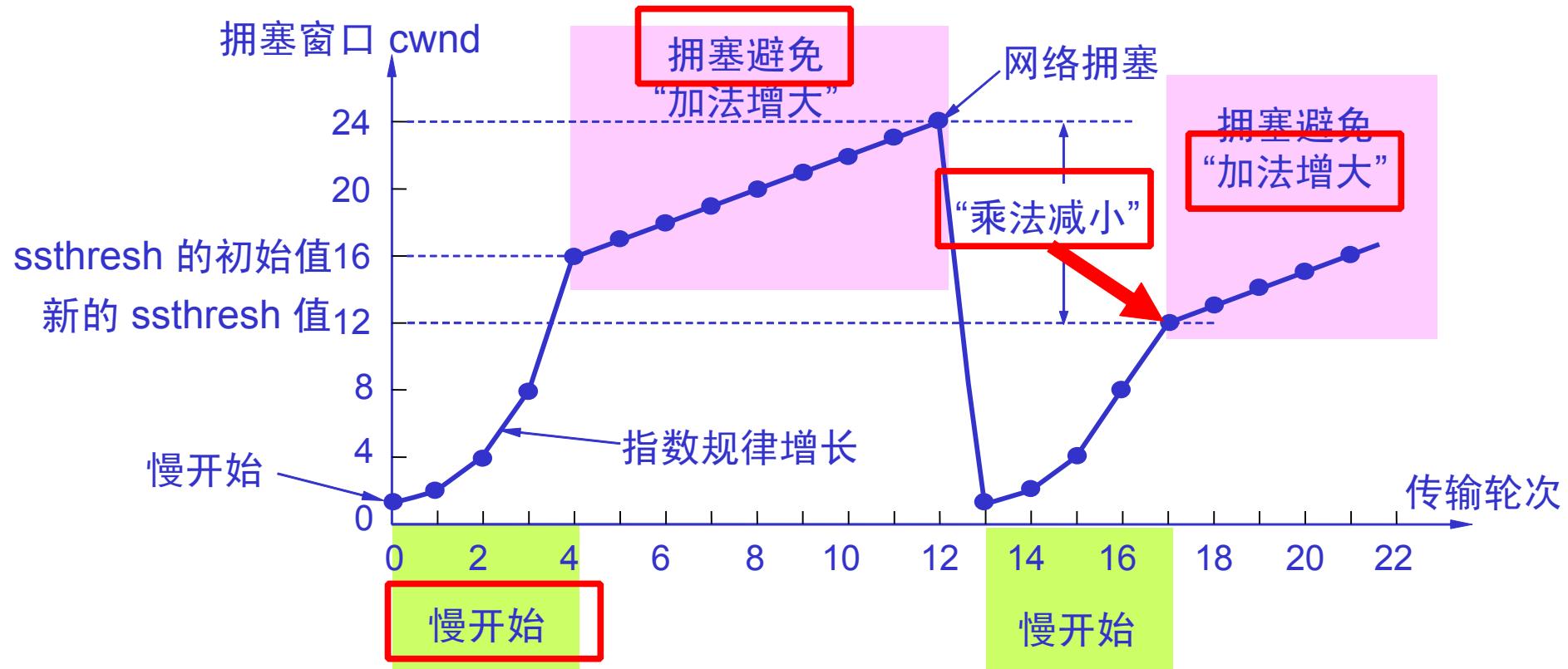
假定拥塞窗口的数值增长到 24 时，网络出现超时，表明网络拥塞了。

慢开始和拥塞避免算法的实现举例

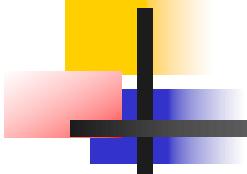


更新后的 ssthresh 值变为 12（即发送窗口数值 24 的一半），拥塞窗口再重新设置为 1，并执行慢开始算法。

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。



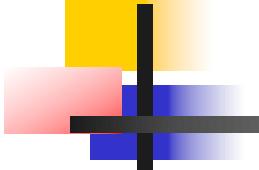
乘法减小和加法增大

■ 乘法减小(multiplicative decrease)

- 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。

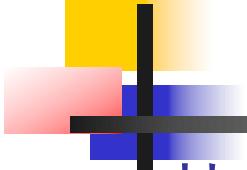
■ 加法增大(additive increase)

- 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

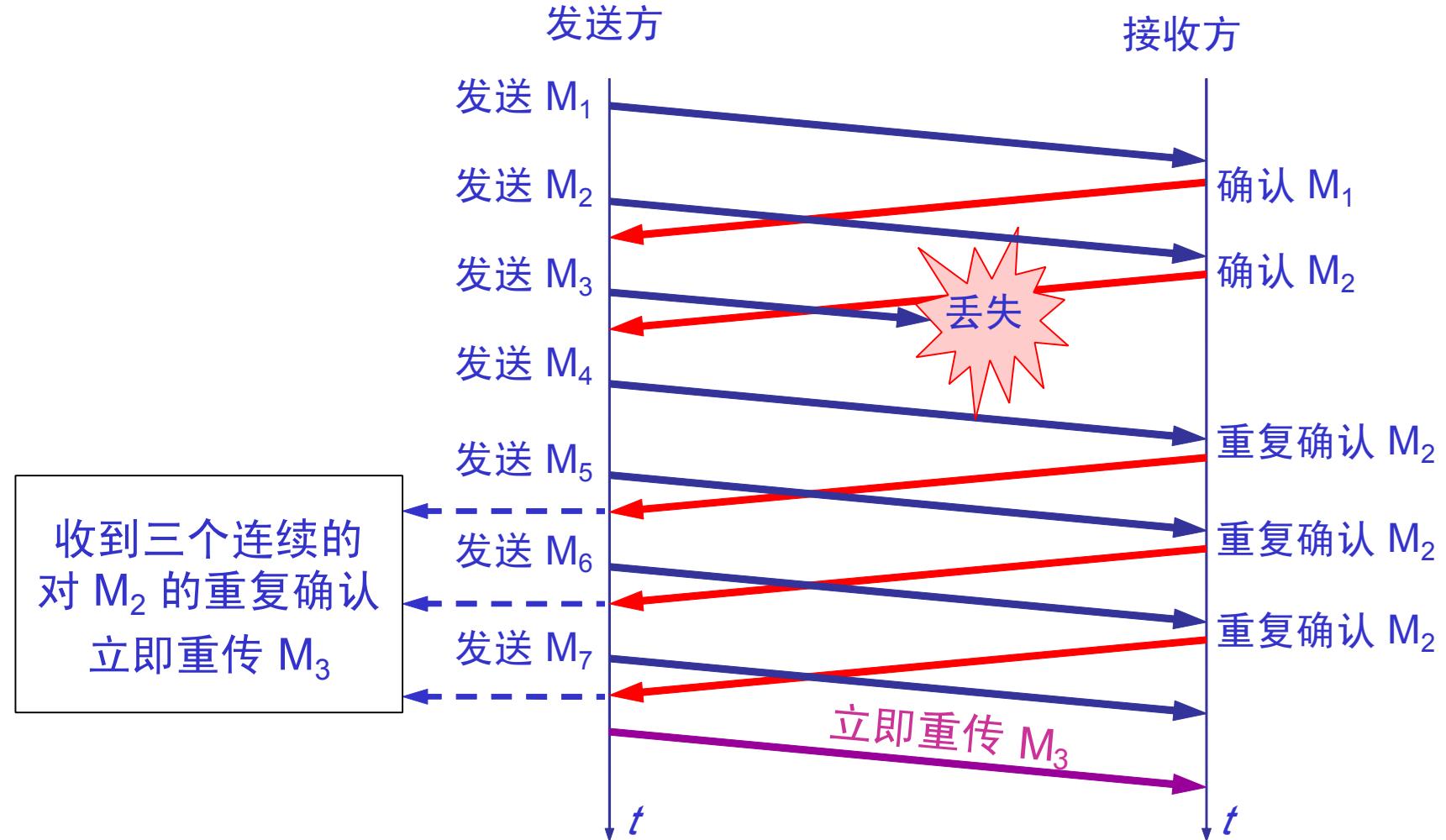
- “拥塞避免”并非指完全能够避免了拥塞，利用以上的措施要完全避免网络拥塞还是不可能的。
- “拥塞避免”是说在拥塞避免阶段把拥塞窗口控制为按线性规律增长，使网络比较不容易出现拥塞。

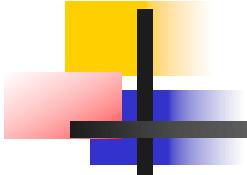


2. 快重传和快恢复

- 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

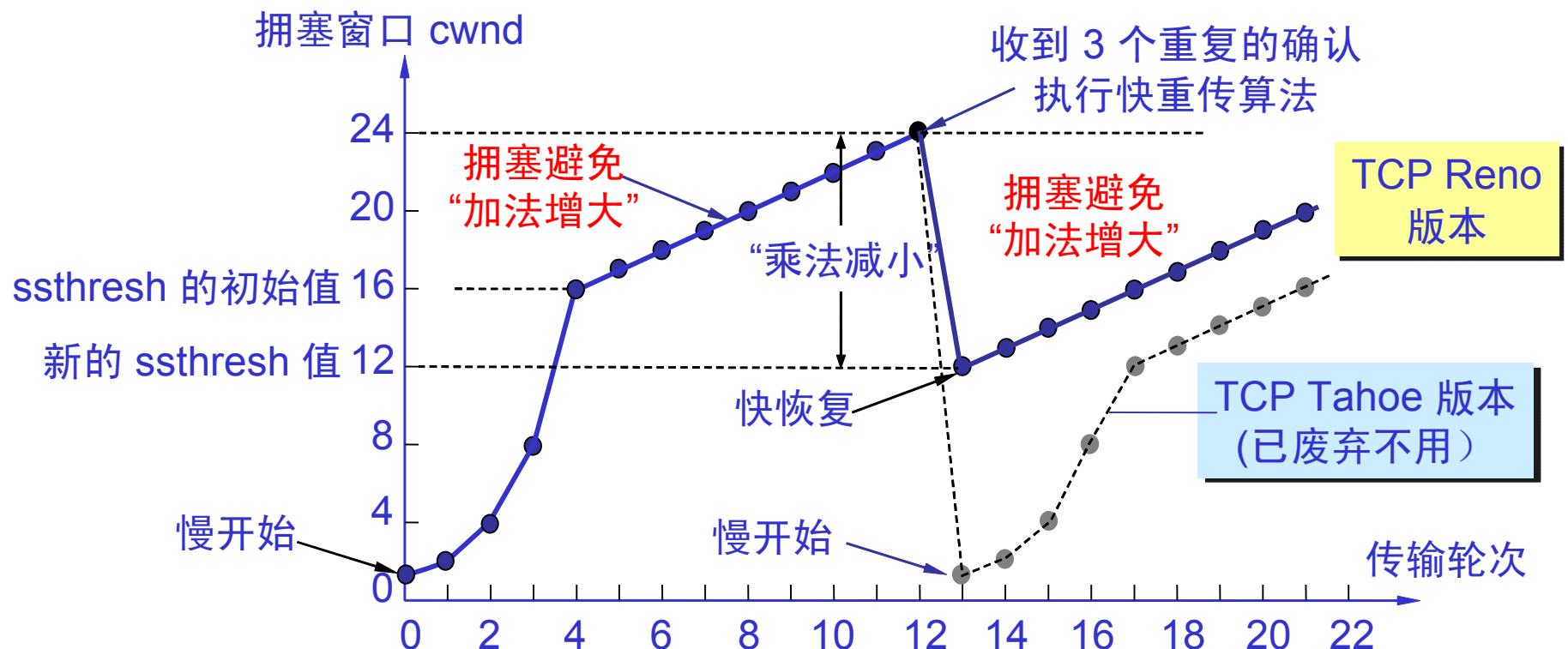


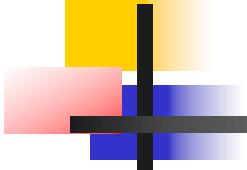


快恢复算法

- 当发送端收到连续三个重复的确认时，直接转入拥塞避免：
 - 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，而是执行拥塞避免算法（“加法增大”），使拥塞窗口缓慢地线性增大。
 - 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。

从连续收到三个重复的确认 转入拥塞避免





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：

$$\text{发送窗口的上限值} = \text{Min} [\text{rwnd}, \text{cwnd}] \quad (5-8)$$

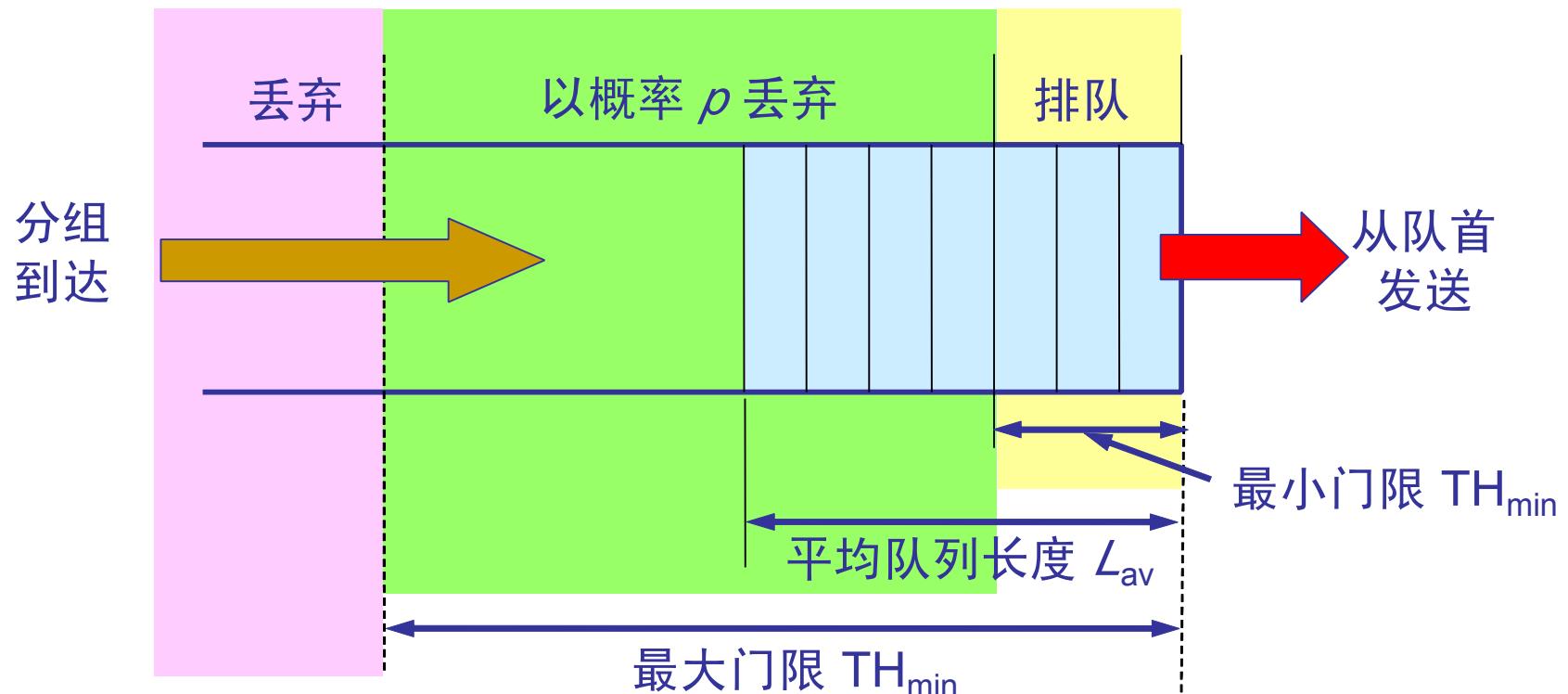
- 当 $\text{rwnd} < \text{cwnd}$ 时，是接收方的接收能力限制发送窗口的最大值。
- 当 $\text{cwnd} < \text{rwnd}$ 时，则是网络的拥塞限制发送窗口的最大值。

路由器处理拥塞的方法

5.8.3 随机早期检测 RED

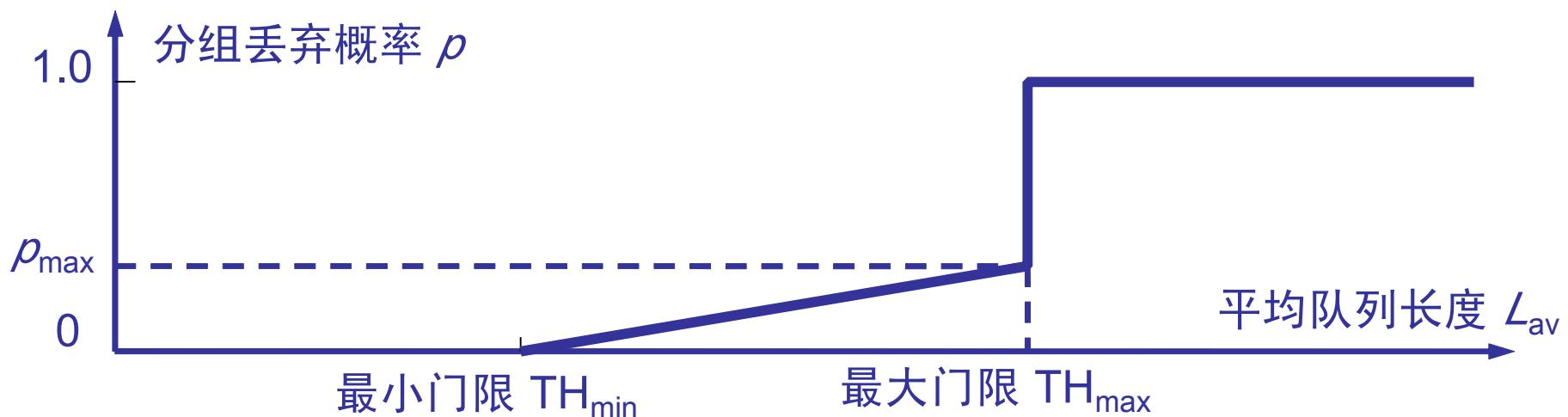
- 使路由器的队列维持两个参数，即队列长度最小门限 TH_{min} 和最大门限 TH_{max} 。
- RED 对每一个到达的数据报都先计算平均队列长度 L_{AV} 。
- 若平均队列长度小于最小门限 TH_{min} ，则将新到达的数据报放入队列进行排队。
- 若平均队列长度超过最大门限 TH_{max} ，则将新到达的数据报丢弃。
- 若平均队列长度在最小门限 TH_{min} 和最大门限 TH_{max} 之间，则按照某一概率 ρ 将新到达的数据报丢弃。

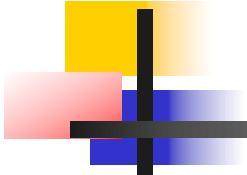
RED 将路由器的到达队列 划分成为三个区域



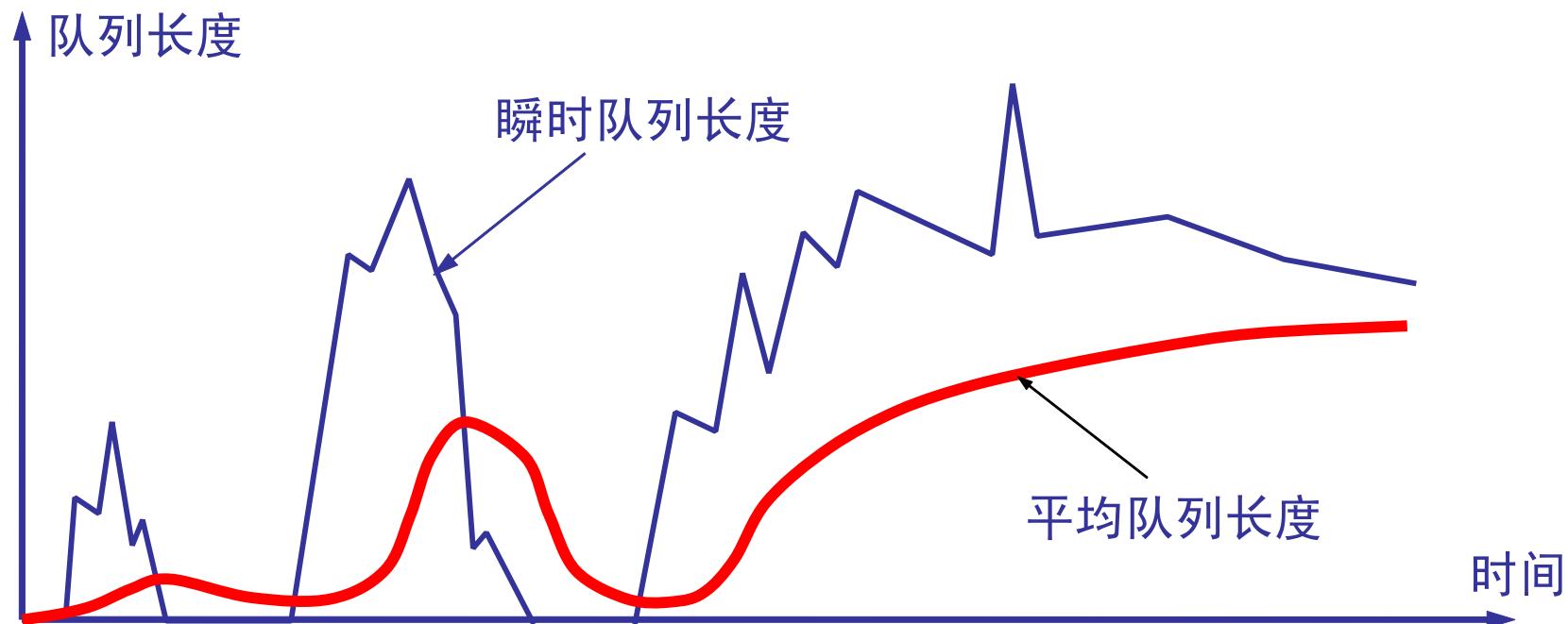
丢弃概率 ρ 与 TH_{min} 和 Th_{max} 的关系

- 当 $L_{AV} < TH_{min}$ 时，丢弃概率 $\rho = 0$ 。
- 当 $L_{AV} > Th_{max}$ 时，丢弃概率 $\rho = 1$ 。
- 当 $TH_{min} < L_{AV} < TH_{max}$ 时， $0 < \rho < 1$ 。
例如，按线性规律变化，从 0 变到 ρ_{max} 。





瞬时队列长度和 平均队列长度的区别





作业

5-9、10、13、19、20、22、24、31、34、
37、38、46