第 5 章 运输层

第 5 章 运输层

- 5.1 运输层协议概述
- 5.2 用户数据报协议 UDP
- 5.3 传输控制协议 TCP 概述
- 5.4 可靠传输的工作原理
- 5.5 TCP 报文段的首部格式
- 5.6 TCP 可靠传输的实现
- 5.7 TCP 的流量控制
- 5.8 TCP 的拥塞控制
- 5.9 TCP 的运输连接管理

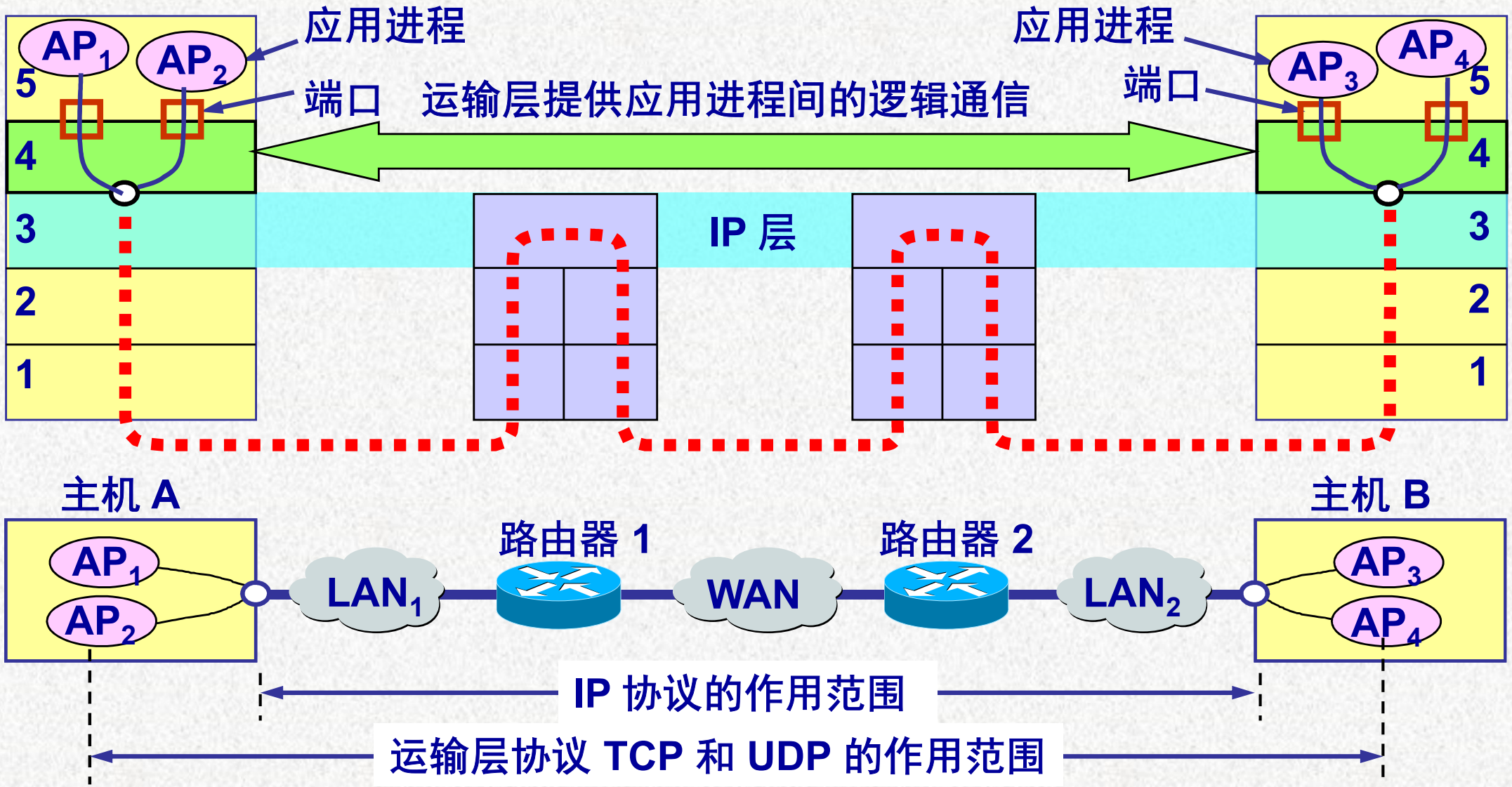
5.1 运输层协议概述

- 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

5.1.1 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。

运输层的作用



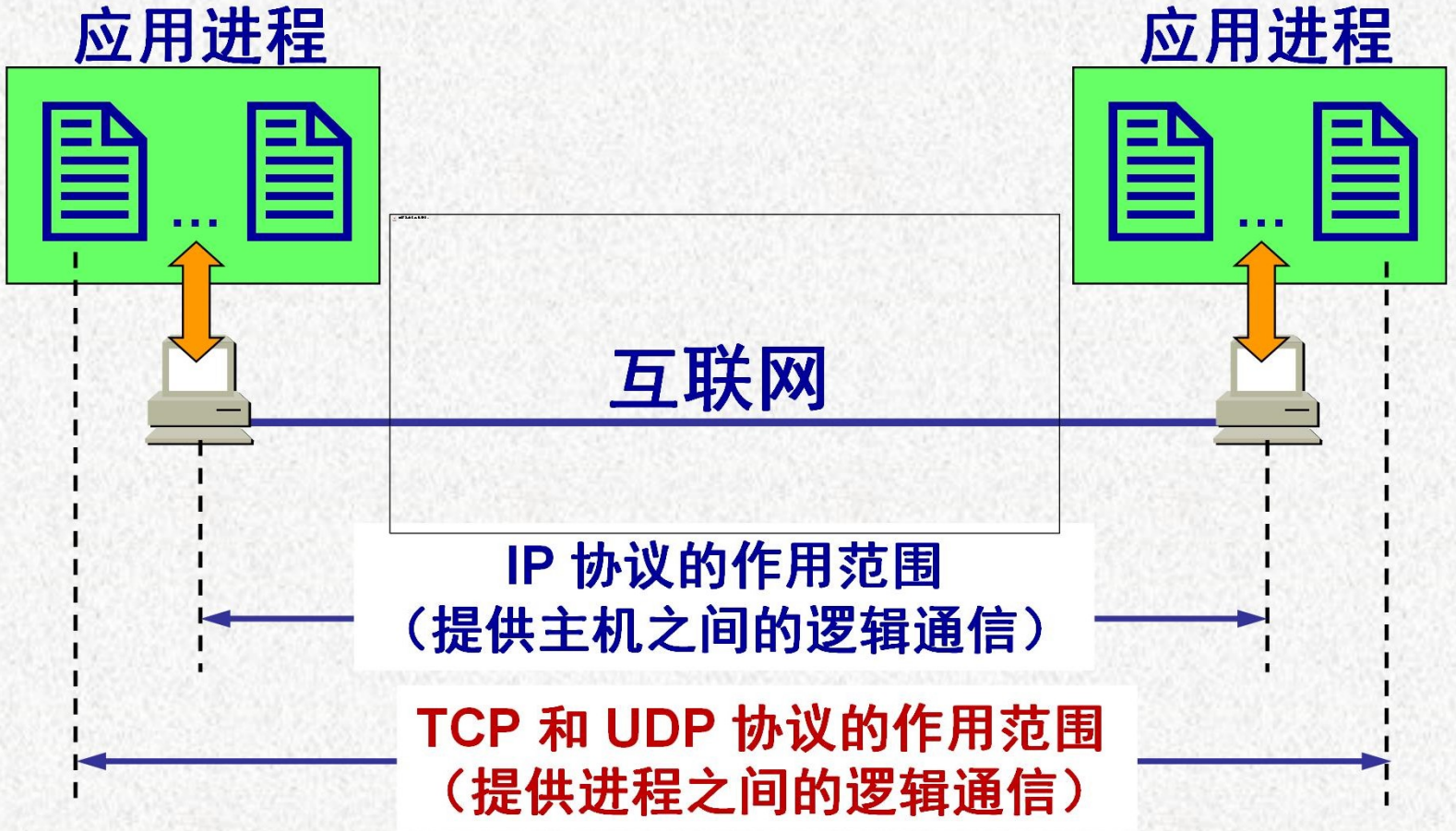
运输层为相互通信的应用进程提供了逻辑通信

运输层的作用

- “逻辑通信”的意思是“好像是这样通信，但事实上并非真的这样通信”。
- 从IP层来说，通信的两端是两台主机。但“两台主机之间的通信”这种说法还不够清楚。
- 严格地讲，两台主机进行通信就是两台主机中的应用进程互相通信。
- 从运输层的角度看，通信的真正端点并不是主机而是主机中的进程。也就是说，端到端的通信是应用进程之间的通信。

网络层和运输层有明显的区别

网络层是为主机之间提供逻辑通信，
而运输层为应用进程之间提供端到端的逻辑通信。

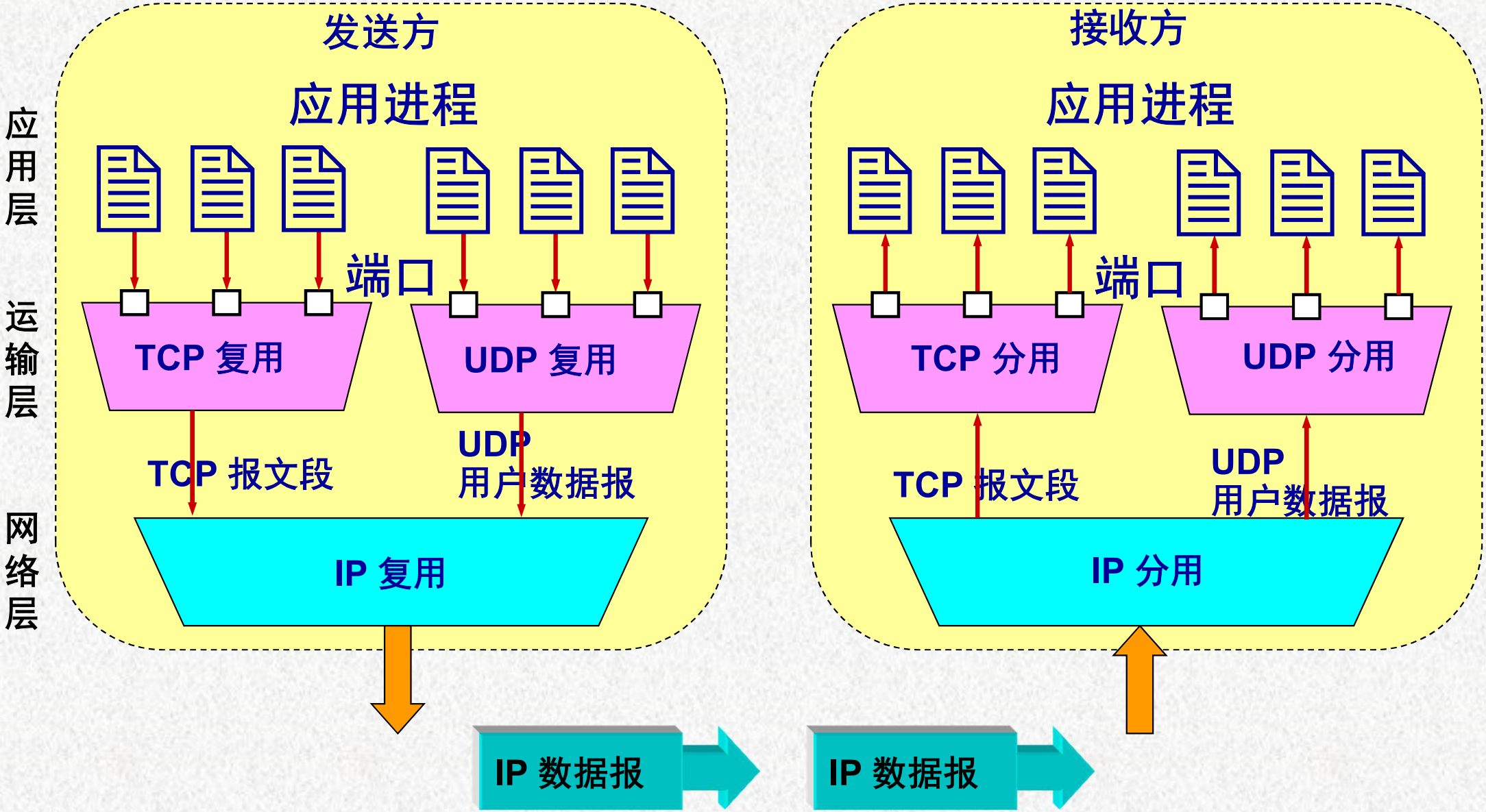


运输层协议和网络层协议的主要区别

运输层的作用

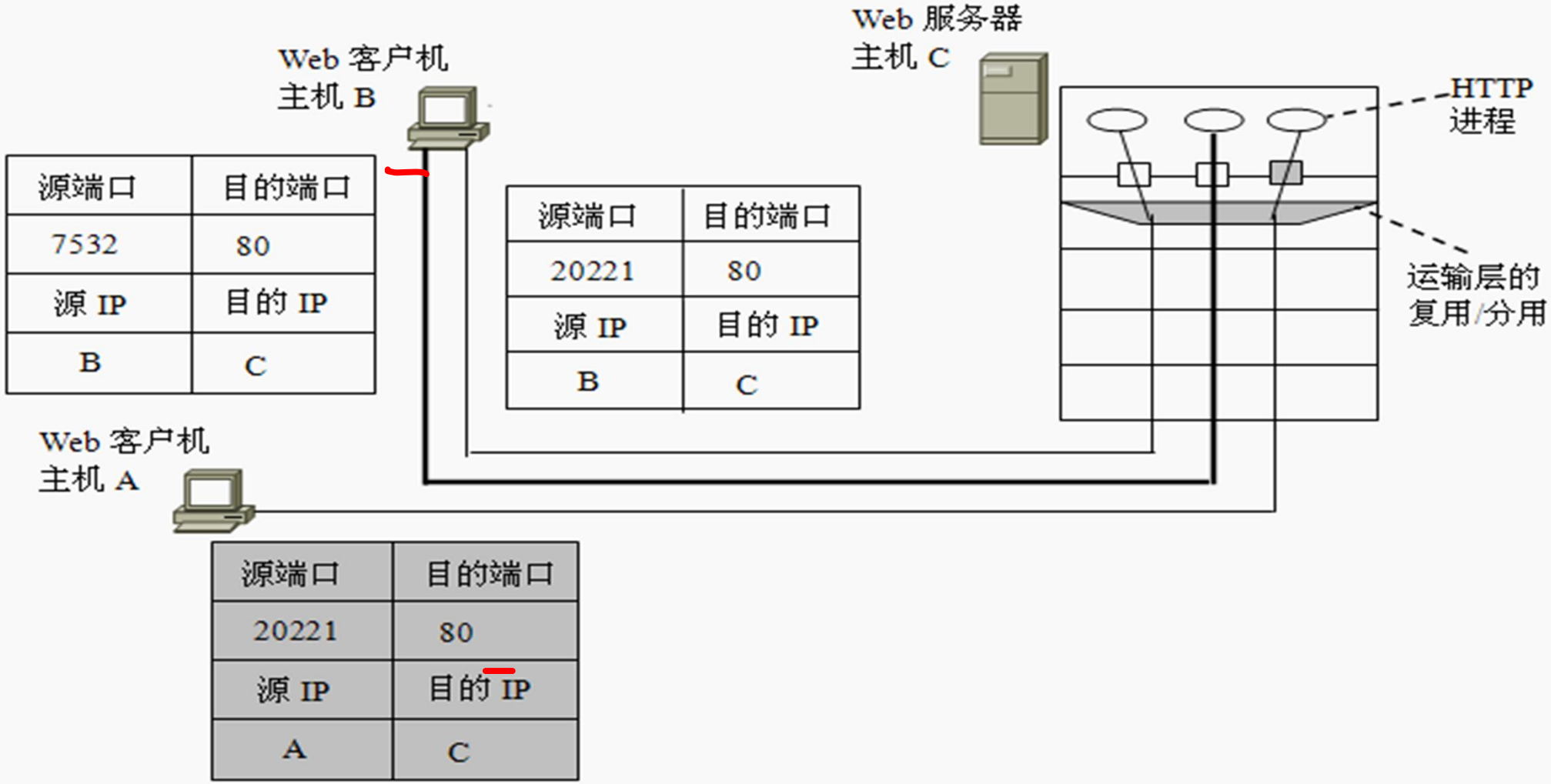
- 在一台主机中经常有多个应用进程同时分别和另一台主机中的多个应用进程通信。
- 这表明运输层有一个很重要的功能——复用 (multiplexing)和分用 (demultiplexing)。
- 根据应用程序的不同需求，运输层需要有两种不同的运输协议，即面向连接的 TCP 和无连接的 UDP 。

基于端口的复用和分用功能



3个应用进程连接的复用和分用

两台客户机主机A、B，一台服务器主机C，A有一个HTTP进程与服务器连接，B有两个HTTP进程与服务器连接



源端口	目的端口
7532	80
源 IP	目的 IP
B	C

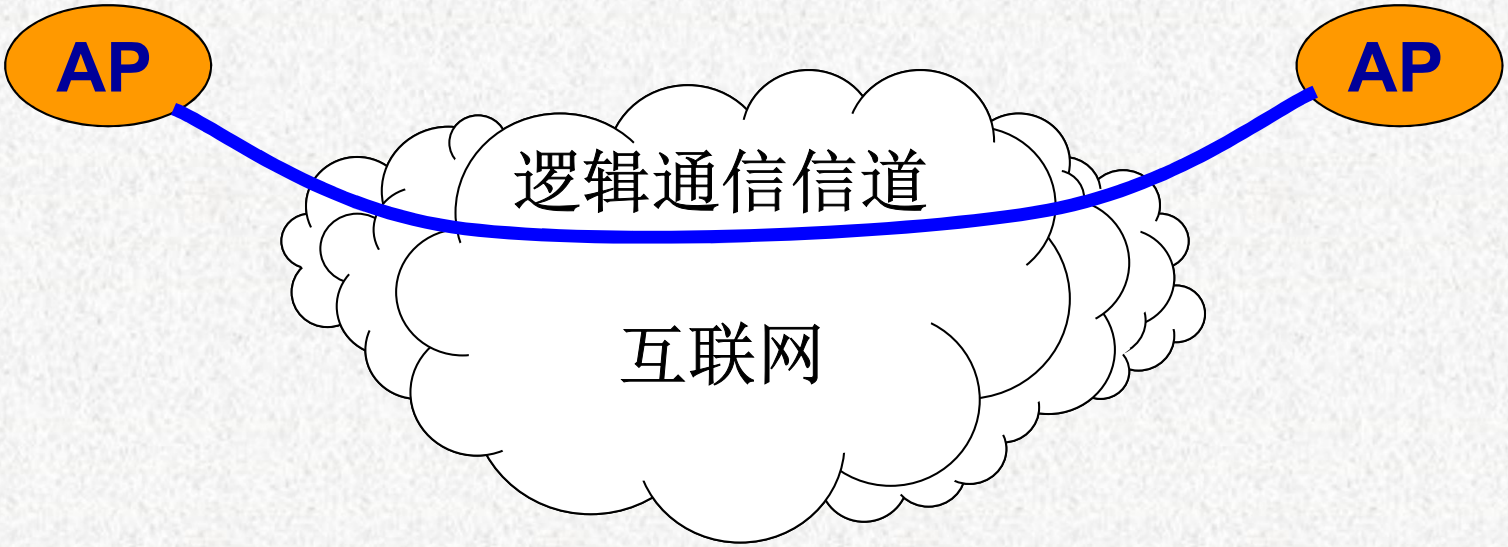
源端口	目的端口
20221	80
源 IP	目的 IP
B	C

源端口	目的端口
20221	80
源 IP	目的 IP
A	C

图 5-6　3 个应用进程连接的复用和分用

屏蔽作用

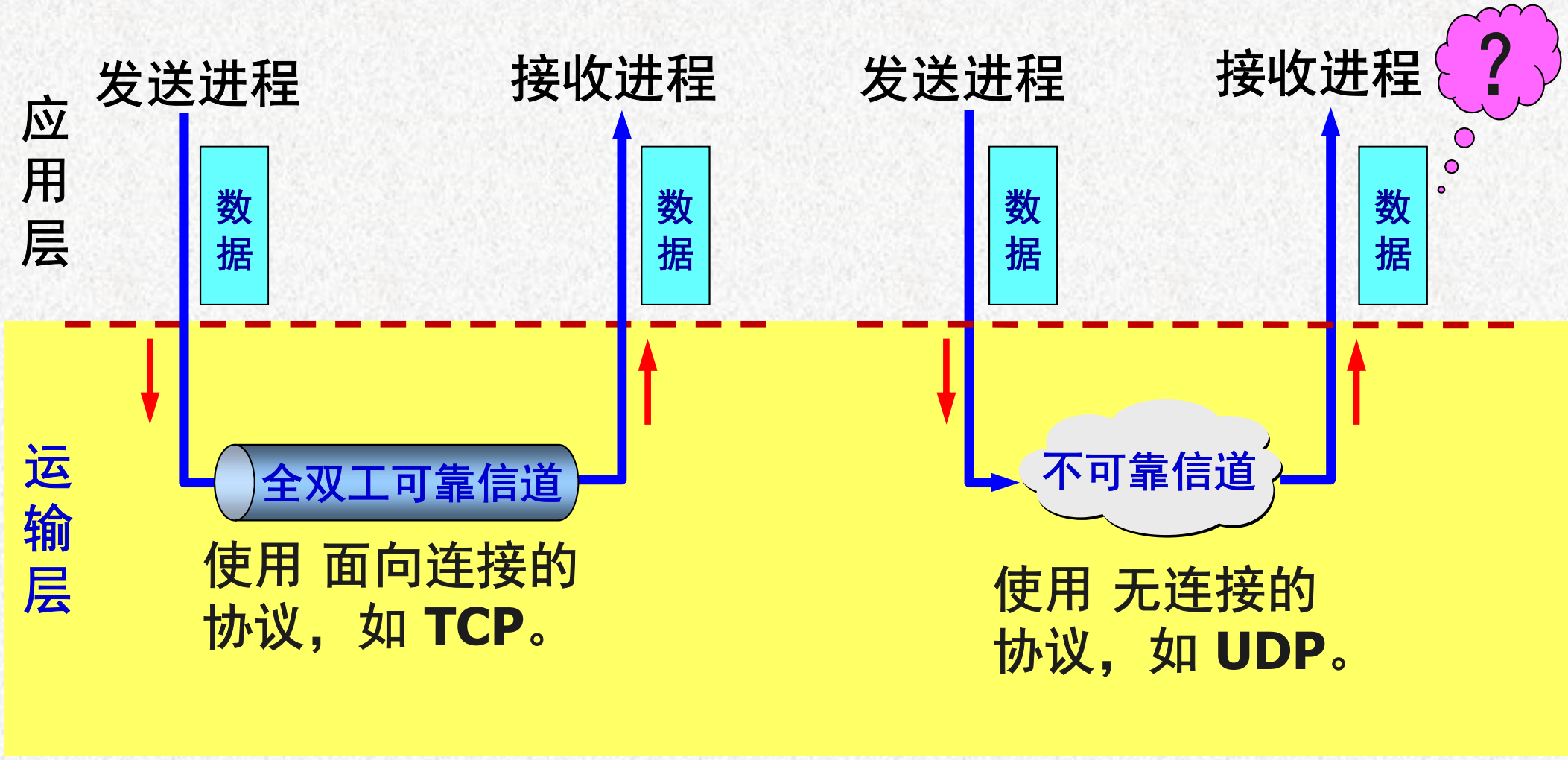
- 运输层向高层用户屏蔽了下面网络核心的细节（如网络拓扑、所采用的路由选择协议等），它使应用进程看见的就是好像在两个运输层实体之间有一条端到端的逻辑通信信道。



两种不同的运输协议

- 但这条逻辑通信信道对上层的表现却因运输层使用的不同协议而有很大的差别。
- 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。

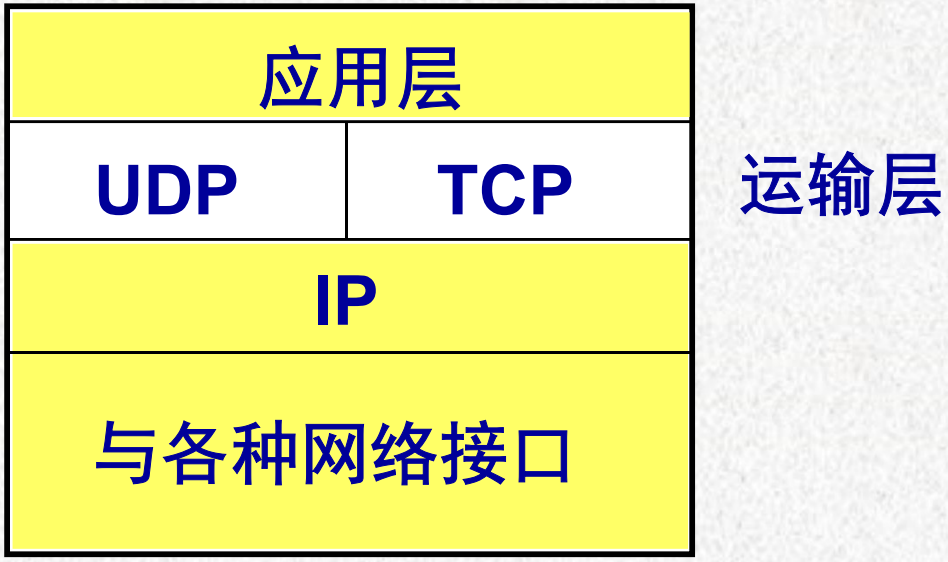
可靠信道与不可靠信道


应用层
发送进程
接收进程
数据
数据
运输层
全双工可靠信道
使用 面向连接的协议，如 TCP。
发送进程
接收进程
?
数据
数据
不可靠信道
使用 无连接的协议，如 UDP。

5.1.2 运输层的两个主要协议

TCP/IP 的运输层有两个主要协议：

- (1) 用户数据报协议 UDP (User Datagram Protocol)
- (2) 传输控制协议 TCP (Transmission Control Protocol)



TCP/IP 体系中的运输层协议

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 与 UDP

- **UDP：一种无连接协议**
  - 提供无连接服务。
  - 在传送数据之前不需要先建立连接。
  - 传送的数据单位协议是 **UDP** 报文或用户数据报。
  - 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 与 UDP

- **TCP：一种面向连接的协议**
  - 提供面向连接的服务。
  - 传送的数据单位协议是 **TCP 报文段** (segment)。
  - **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。
  - 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

- 运行在计算机中的进程是用进程标识符来标志的。
- 但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。这是因为在互联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)

- 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

- 两个不同的概念。
- 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/IP 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。
- 在互联网中，不同计算机的相同端口号是没有联系的。

由此可见，两个计算机中的进程要互相通信，不仅必须知道对方的 IP 地址（为了找到对方的计算机），而且还要知道对方的端口号（为了找到对方计算机中的应用进程）。

两大类端口

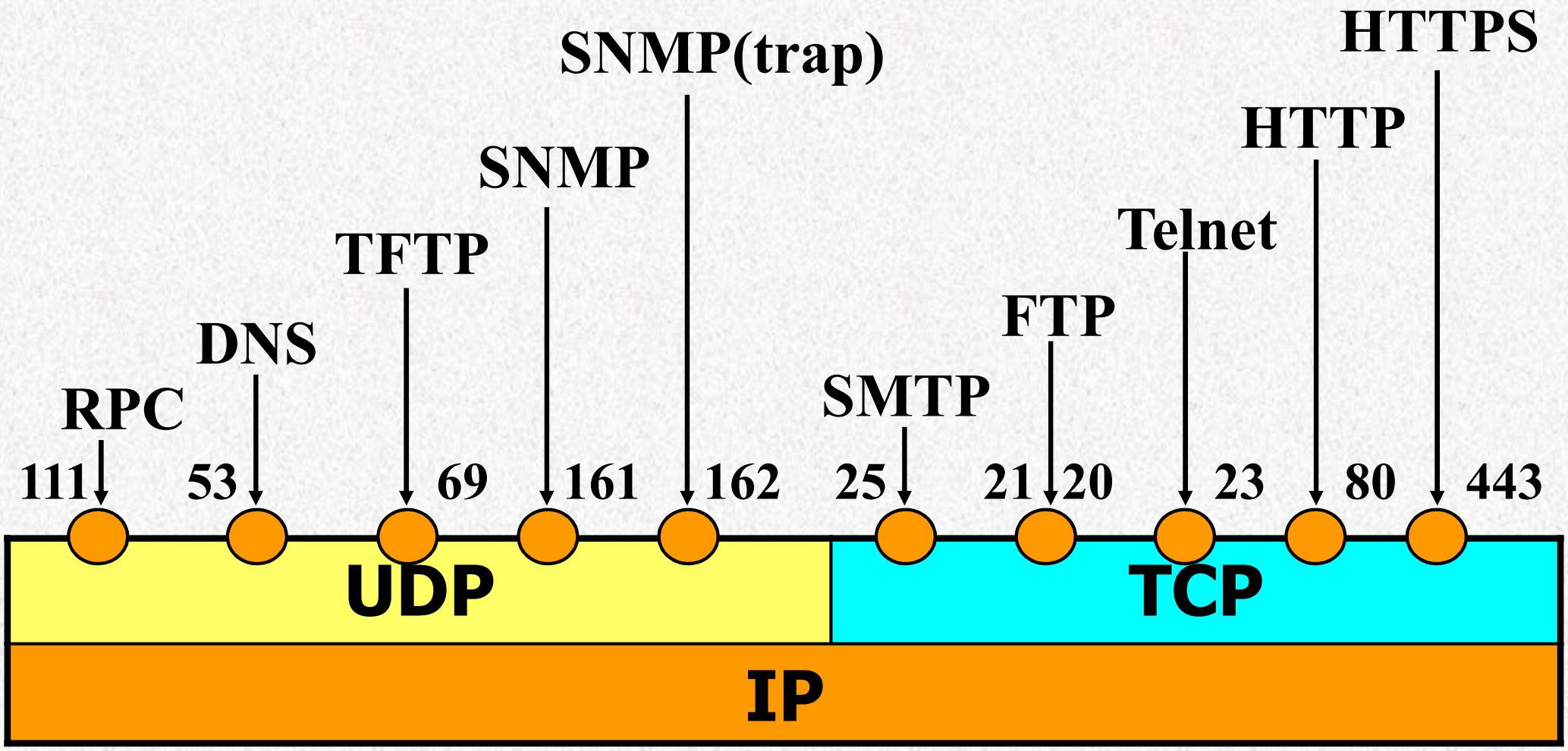
(1) 服务器端使用的端口号

- 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。

(2) 客户端使用的端口号

- 又称为短暂端口号，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。

常用的熟知端口


SNMP(trap)
HTTPS
HTTP
SNMP
Telnet
TFTP
FTP
DNS
SMTP
RPC
111
53
69
161
162
25
21
20
23
80
443
UDP
TCP
IP

5.2 用户数据报协议 UDP

- 5.2.1 UDP 概述
- 5.2.2 UDP 的首部格式

5.2.1 UDP概述

- UDP 只在 IP 的数据报服务之上增加了很少一点的功能：
  - 复用和分用的功能
  - 差错检测的功能
- 虽然 UDP 用户数据报只能提供不可靠的交付，但 UDP 在某些方面有其特殊的优点。

UDP 的主要特点

- (1) UDP 是无连接的，发送数据之前不需要建立连接，，因此减少了开销和发送数据之前的时延。
- (2) UDP 使用尽最大努力交付，即不保证可靠交付，因此主机不需要维持复杂的连接状态表。
- (3) UDP 是面向报文的。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。UDP 一次交付一个完整的报文。
- (4) UDP 没有拥塞控制，因此网络出现的拥塞不会使源主机的发送速率降低。这对某些实时应用是很重要的。很适合多媒体通信的要求。

UDP 的主要特点

- (5) UDP 支持一对一、一对多、多对一和多对多的交互通信。
- (6) UDP 的首部开销小，只有 8 个字节，比 TCP 的 20 个字节的首部要短。

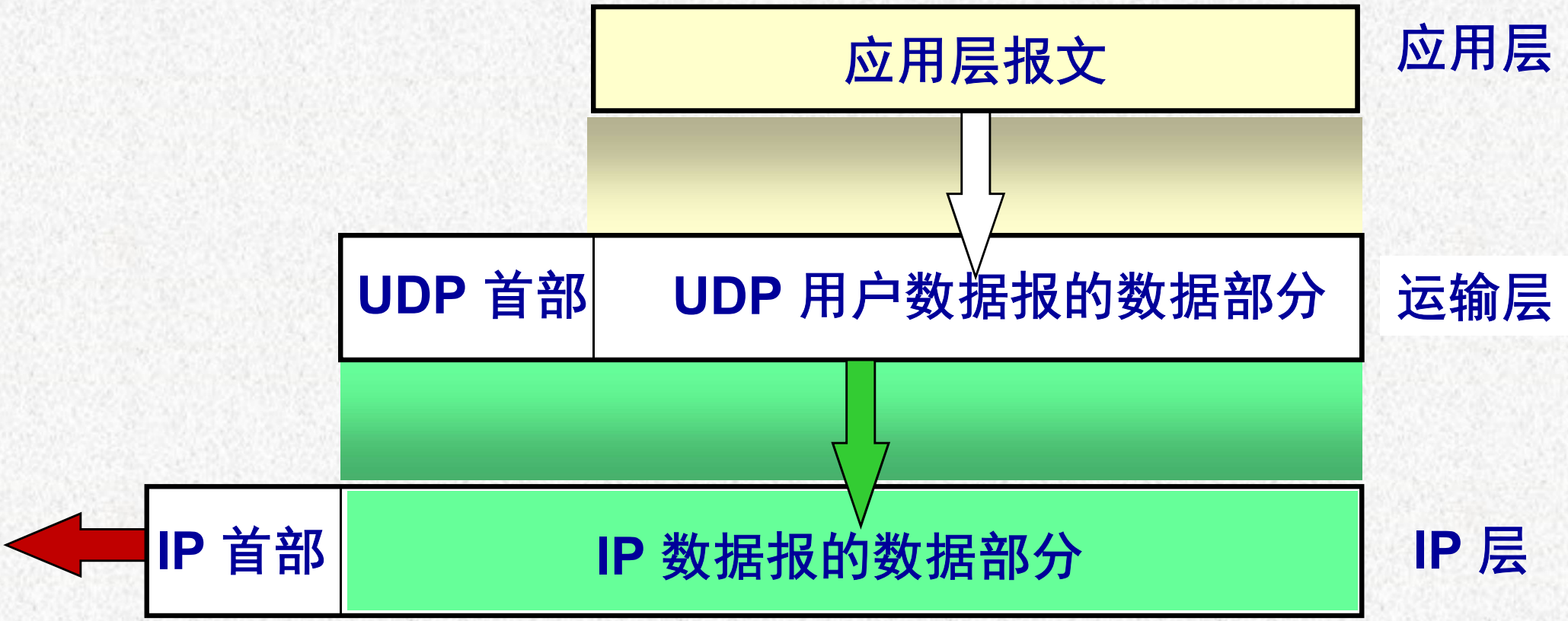
面向报文的 UDP

- 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。

面向报文的 UDP

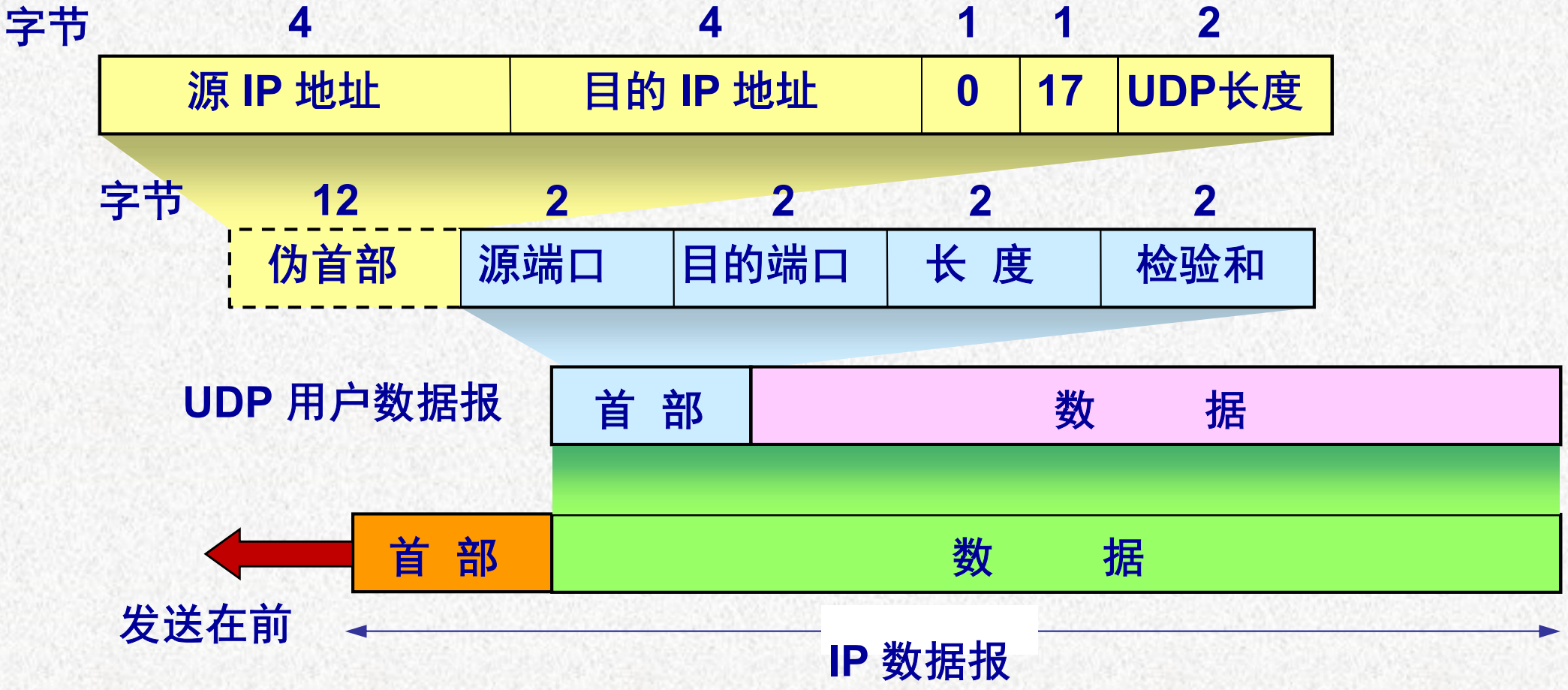
- 接收方 UDP 对 IP 层交上来的 UDP 用户数据报，在去除首部后就原封不动地交付上层的应用进程，一次交付一个完整的报文。
- 应用程序必须选择合适大小的报文。
  - 若报文太长，UDP 把它交给 IP 层后，IP 层在传送时可能要进行分片，这会降低 IP 层的效率。
  - 若报文太短，UDP 把它交给 IP 层后，会使 IP 数据报的首部的相对长度太大，这也降低了 IP 层的效率。

UDP 是面向报文的



5.2.2 UDP 的首部格式

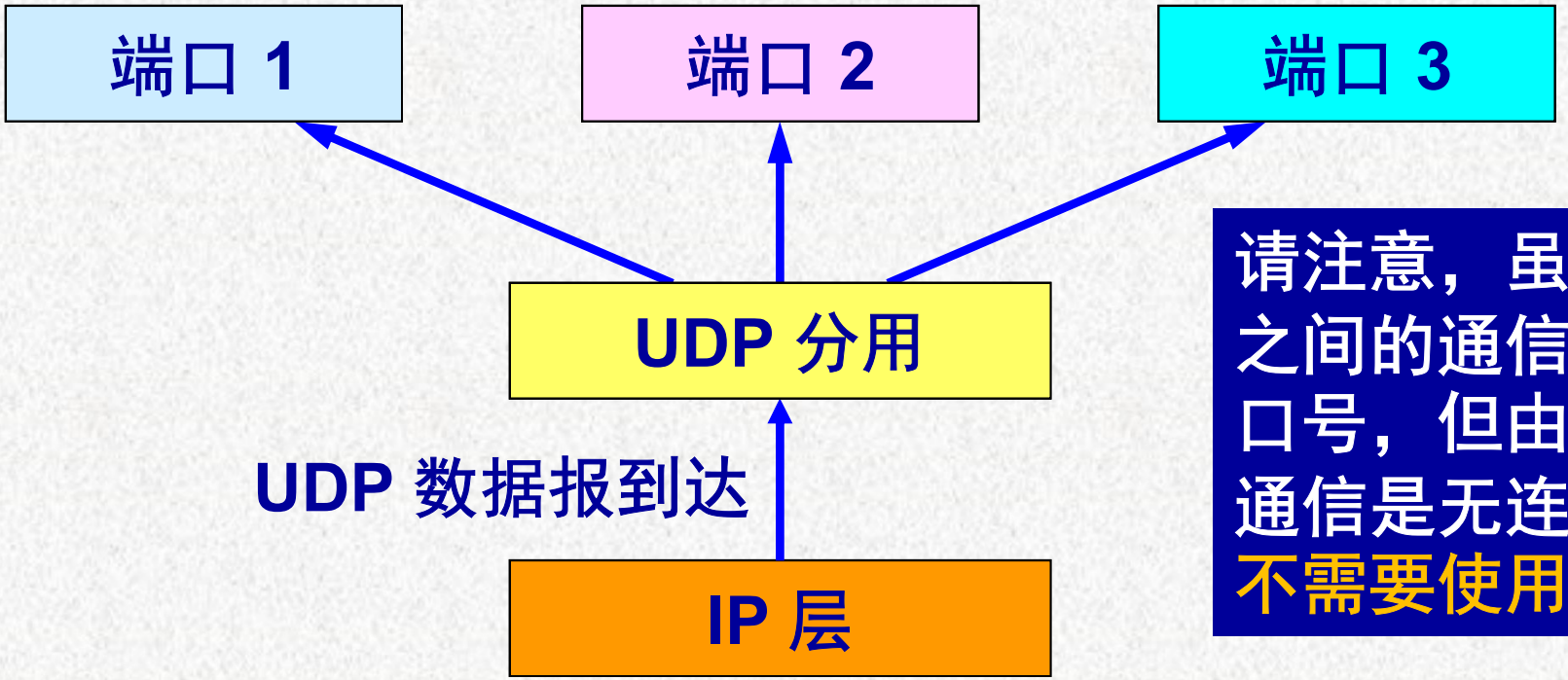
用户数据报 UDP 有两个字段：数据字段和首部字段。
首部字段很简单，只有 8 个字节。



UD P用户数据报的首部和伪首部

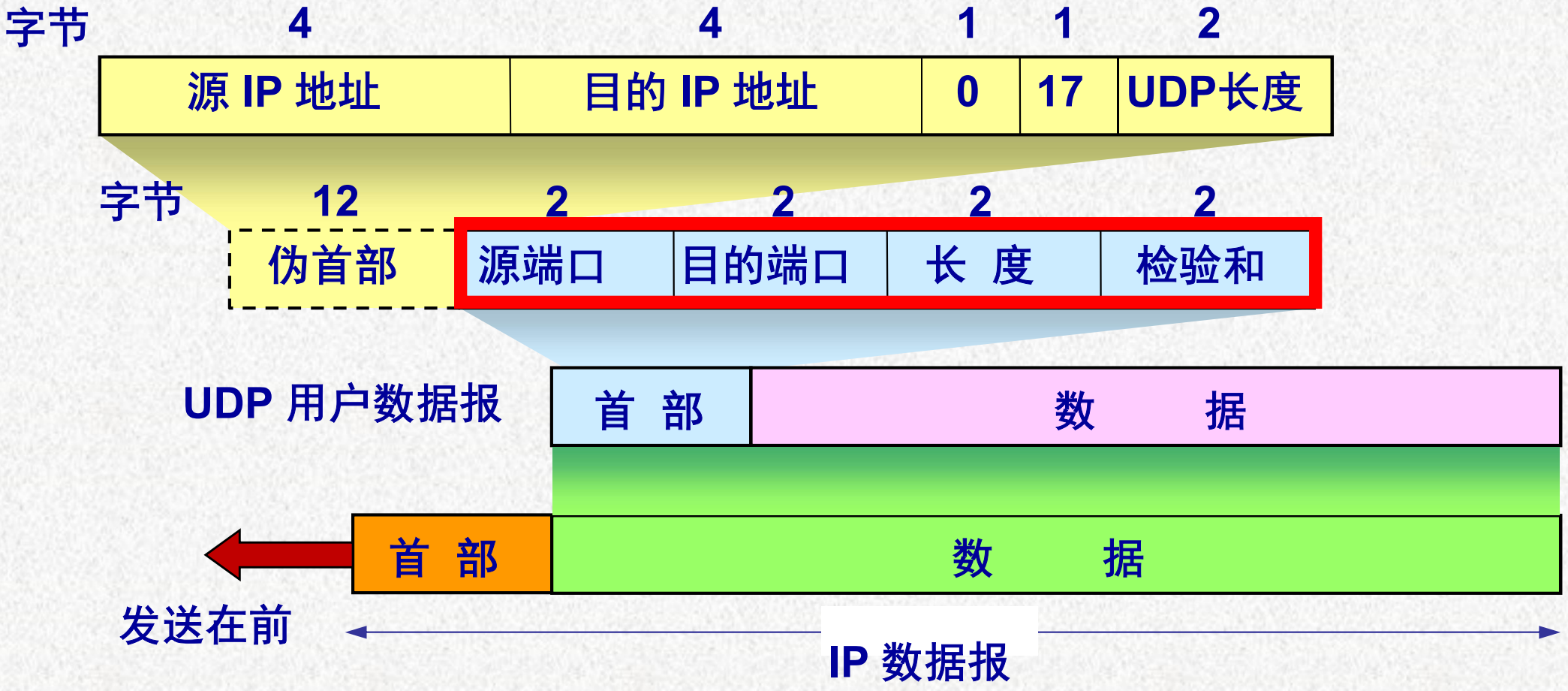
UDP 基于端口的分用

当运输层从 IP 层收到 UDP 数据报时，就根据首部中的目的端口，把 UDP 数据报通过相应的端口，上交最后的终点——应用进程。

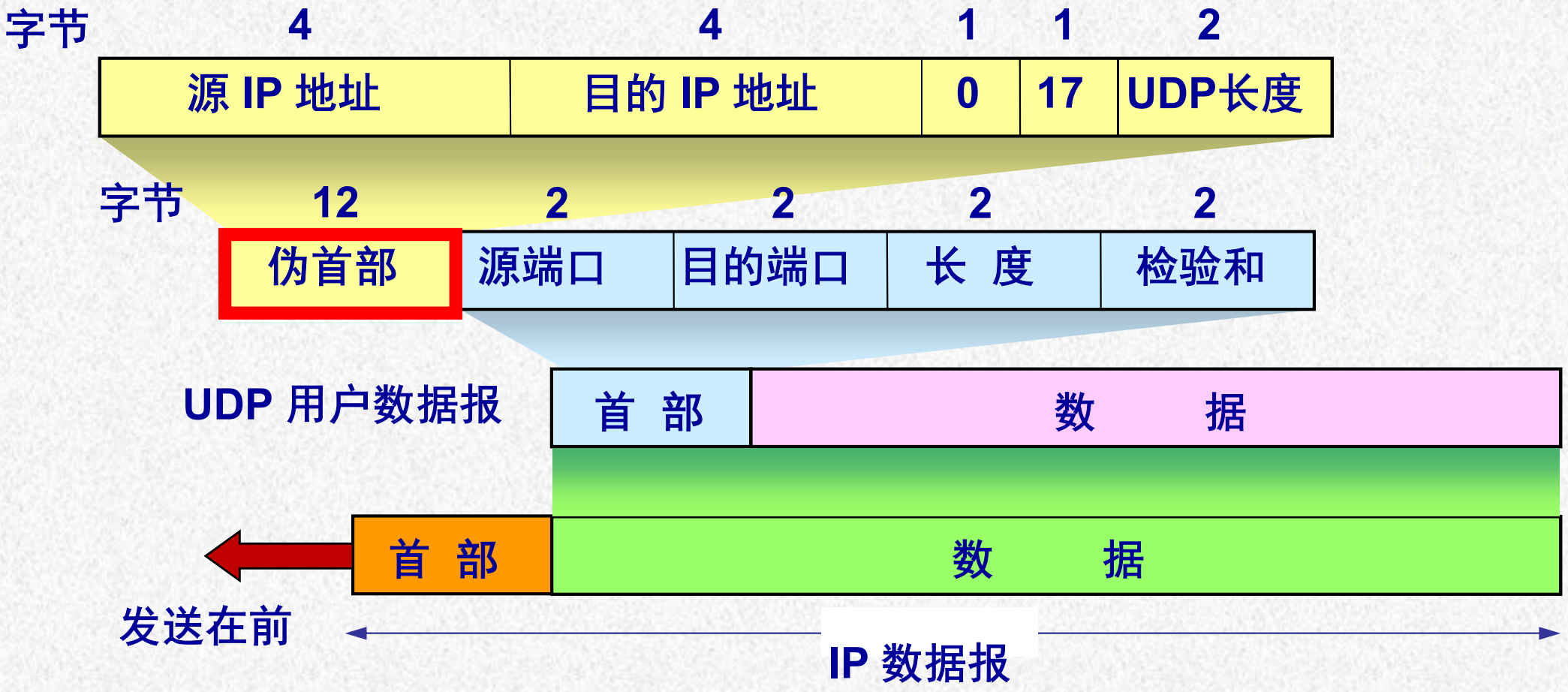


请注意，虽然在 UDP 之间的通信要用到其端口号，但由于 UDP 的通信是无连接的，因此不需要使用套接字。

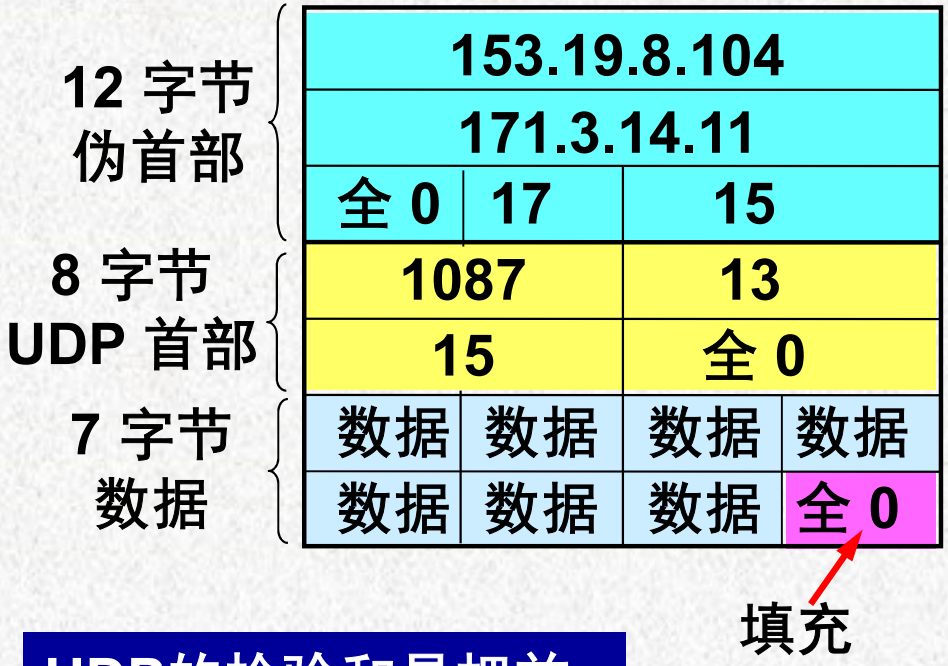
用户数据报 UDP 有两个字段：数据字段和首部字段。首部字段有 8 个字节，由 4 个字段组成，每个字段都是 2 个字节。



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）

UDP的检验和是把首部和数据部分一起都检验。

按二进制反码运算求和
将得出的结果求反码

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

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

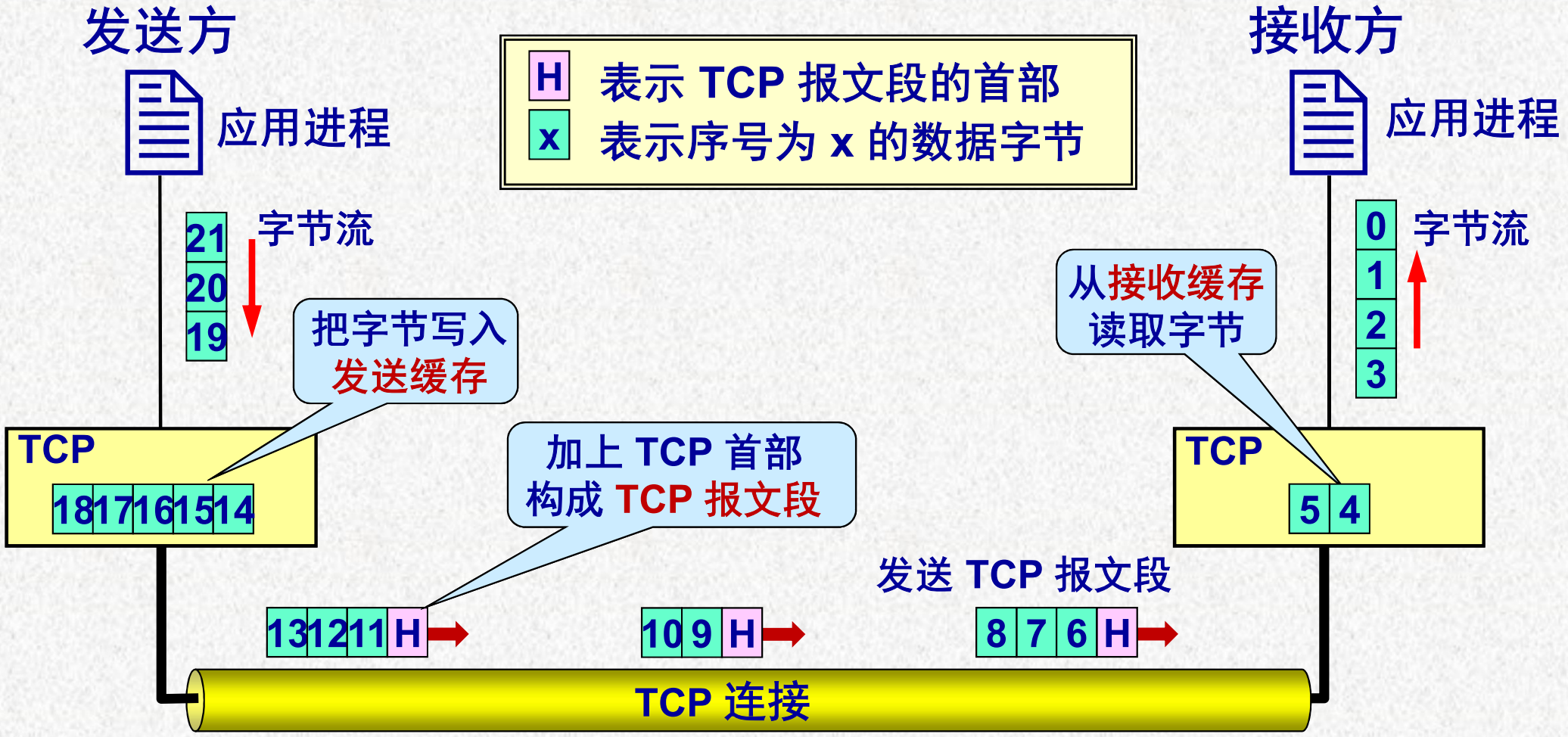
5.3.1 TCP 最主要的特点

- 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 中的“流”(stream)指的是流入或流出进程的字节序列。
  - “面向字节流”的含义是：虽然应用程序和 TCP 的交互是一次一个数据块，但 TCP 把应用程序交下来的数据看成仅仅是一连串无结构的字节流。

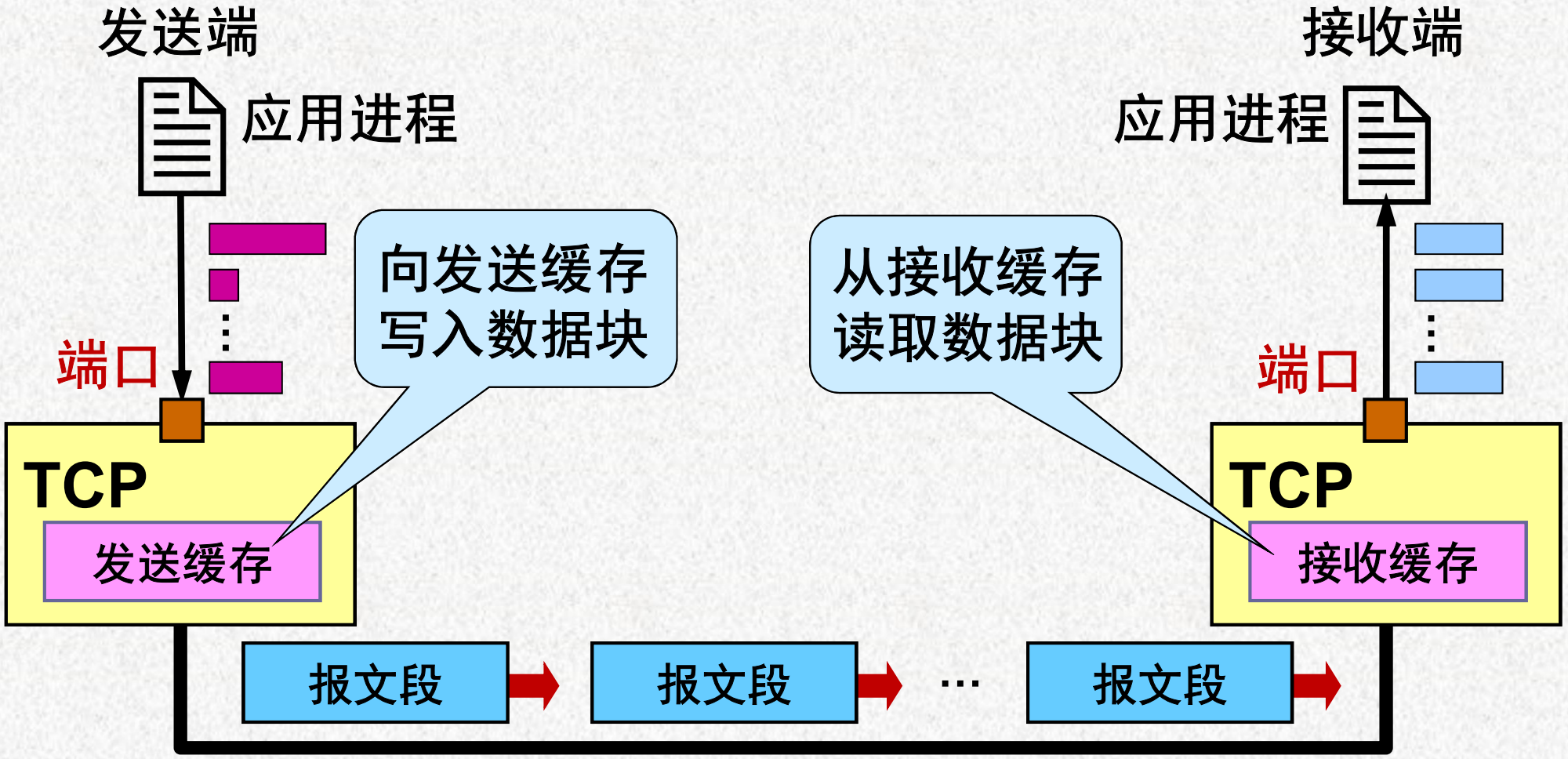
TCP 面向流的概念

- TCP 不保证接收方应用程序所收到的数据块和发送方应用程序所发出的数据块具有对应大小的关系。
- 但接收方应用程序收到的字节流必须和发送方应用程序发出的字节流完全一样。

TCP 面向流的概念


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

TCP 面向流的概念



- TCP 不关心应用进程一次把多长的报文发送到 TCP 缓存。
- TCP 对连续的字节流进行分段，形成 TCP 报文段。

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。

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)　　(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)

TCP 连接，IP 地址，套接字

- TCP 连接就是由协议软件所提供的一种抽象。
- TCP 连接的端点是个很抽象的套接字，即（IP 地址：端口号）。
- 同一个 IP 地址可以有多个不同的 TCP 连接。
- 同一个端口号也可以出现在多个不同的 TCP 连接中。

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。

5.4 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

理想的传输条件特点

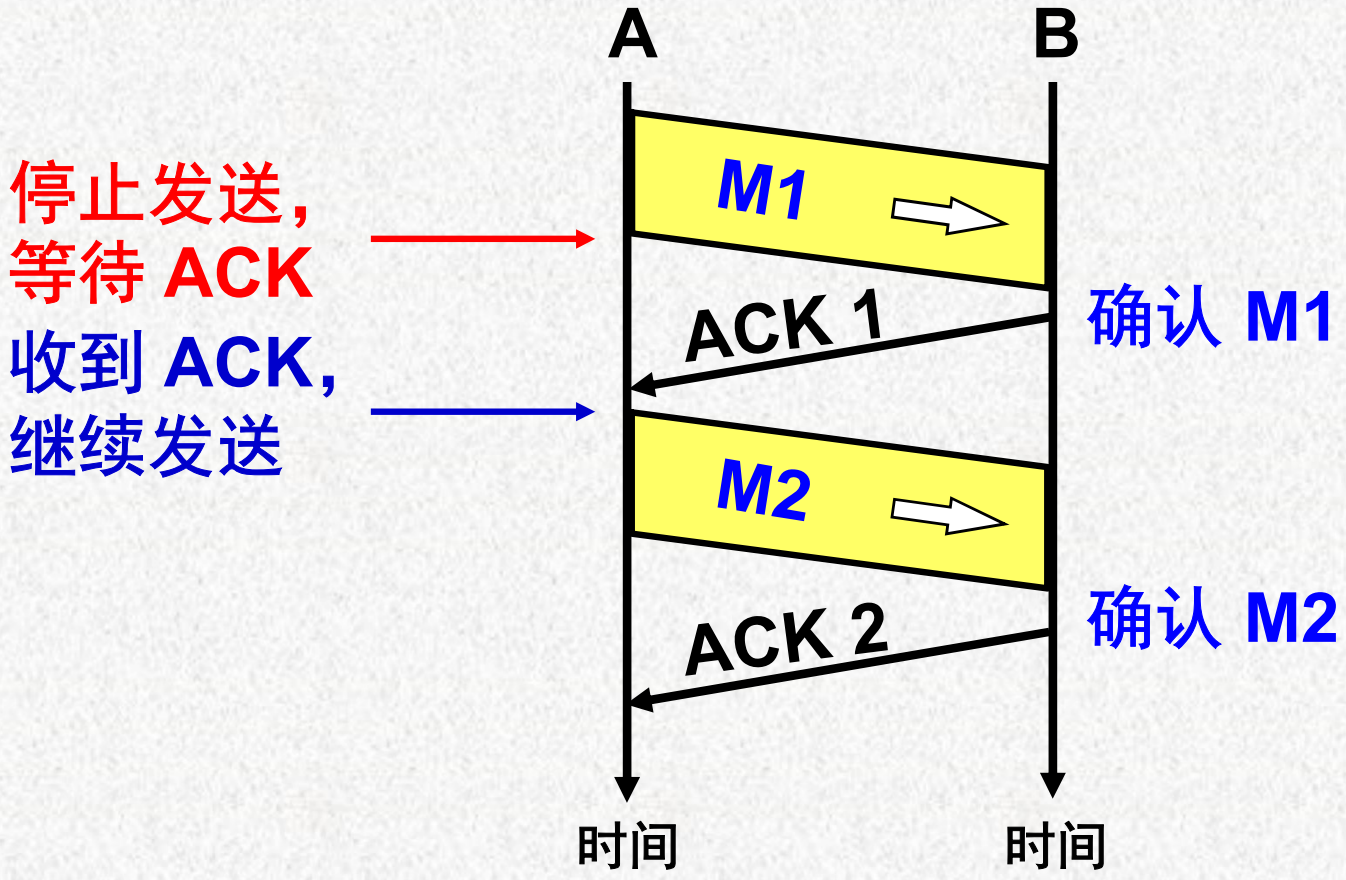
- 理想的传输条件有以下两个特点：
  - (1) 传输信道不产生差错。
  - (2) 不管发送方以多快的速度发送数据，接收方总是来得及处理收到的数据。
- 在这样的理想传输条件下，不需要采取任何措施就能够实现可靠传输。
- 然而实际的网络都不具备以上两个理想条件。必须使用一些可靠传输协议，在不可靠的传输信道实现可靠传输。

5.4.1 停止等待协议

- “停止等待”就是每发送完一个分组就停止发送，等待对方的确认。在收到确认后再发送下一个分组。
- 全双工通信的双方既是发送方也是接收方。
- 为了讨论问题的方便，我们仅考虑 A 发送数据而 B 接收数据并发送确认。因此 A 叫做发送方，而 B 叫做接收方。

1. 无差错情况

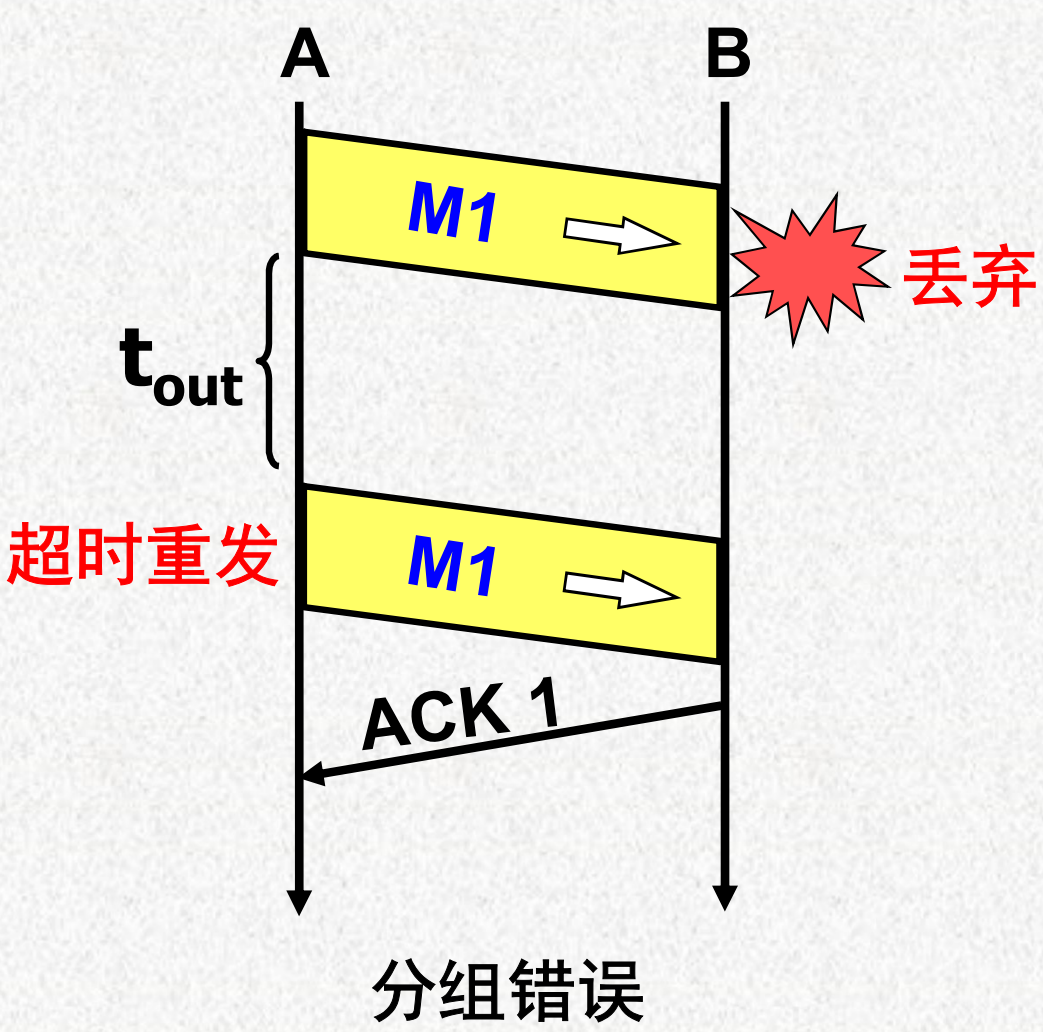
A 发送分组 M1，发完就暂停发送，等待 B 的确认 (ACK)。B 收到了 M1 向 A 发送 ACK。A 在收到了对 M1 的确认后，就再发送下一个分组 M2。



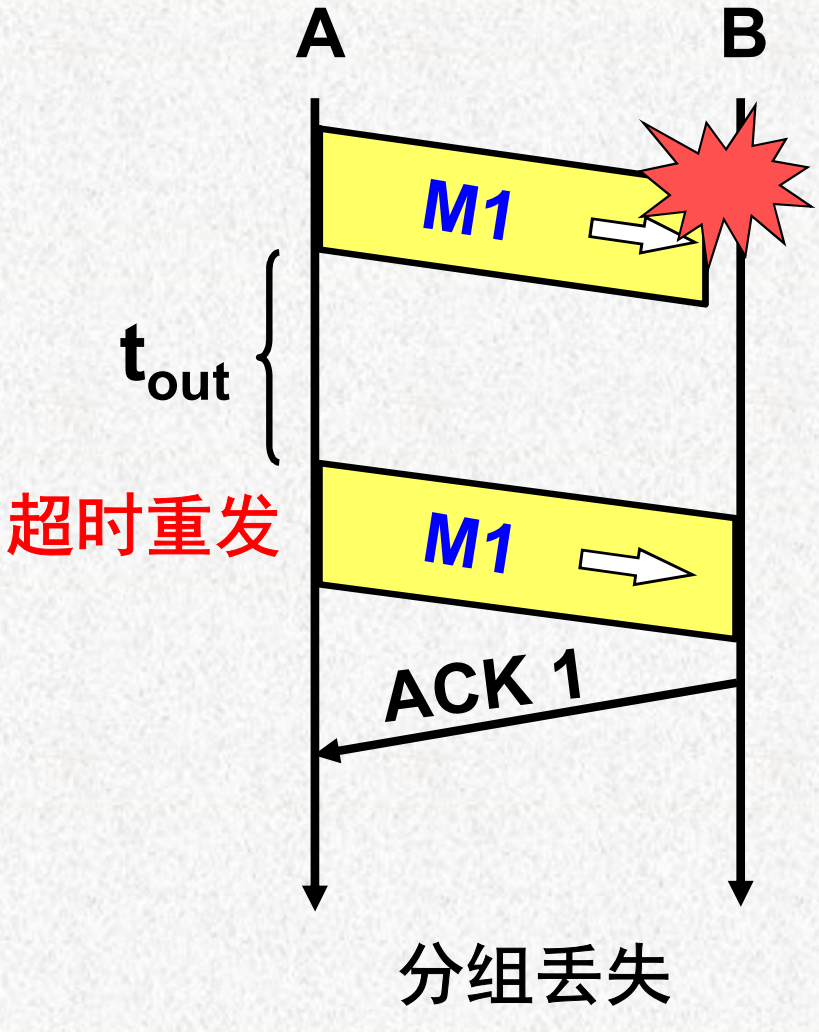
2. 出现差错

- 在接收方 B 会出现两种情况：
  - B 接收 M1 时检测出了差错，就丢弃 M1，其他什么也不做（不通知 A 收到有差错的分组）。
  - M1 在传输过程中丢失了，这时 B 当然什么都不知道，也什么都不做。
- 在这两种情况下，B 都不会发送任何信息。
- 如何保证 B 正确收到了 M1 呢？
- 解决方法：超时重传
  - A 为每一个已发送的分组都设置了一个超时计时器。
  - A 只要在超时计时器到期之前收到了相应的确认，就撤销该超时计时器，继续发送下一个分组 M2 。

2. 出现差错



分组错误



分组丢失

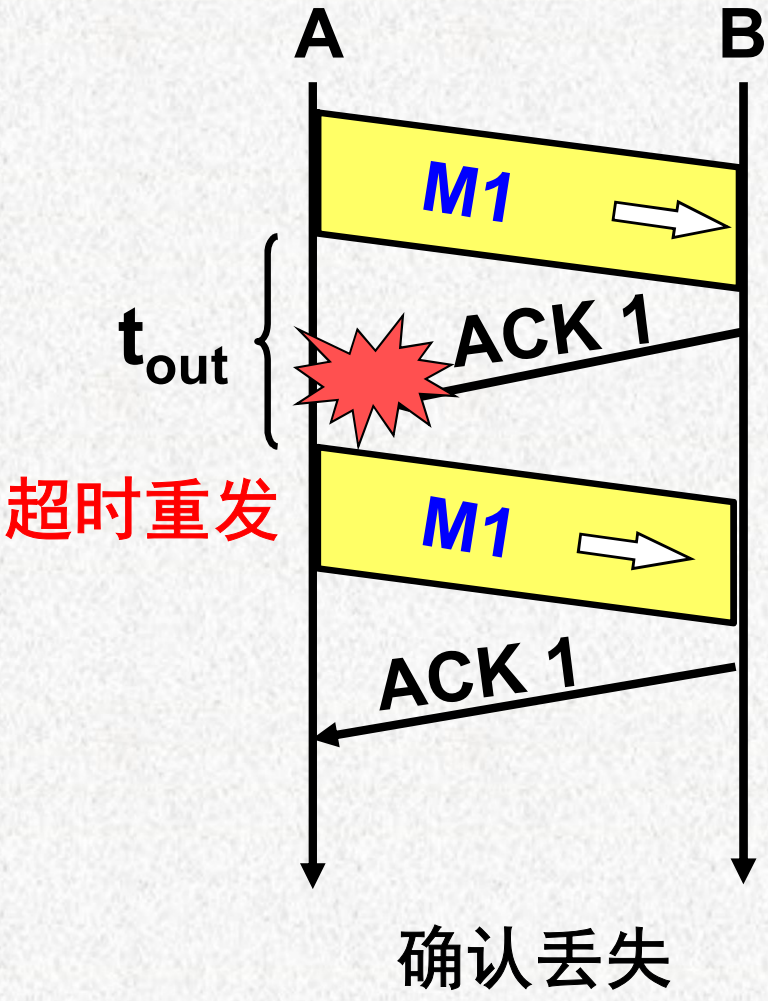
3. 确认丢失和确认迟到

- 确认丢失
  - 若 B 所发送的对 M1 的确认丢失了，那么 A 在设定的超时重传时间内不能收到确认，但 A 并无法知道：是自己发送的分组出错、丢失了，或者 是 B 发送的确认丢失了。因此 A 在超时计时器到期后就要重传 M1。
  - 假定 B 又收到了重传的分组 M1。这时 B 应采取两个行动：
    - 第一，丢弃这个重复的分组 M1，不向上层交付。
    - 第二，向 A 发送确认。不能认为已经发送过确认就不再发送，因为 A 之所以重传 M1 就表示 A 没有收到对 M1 的确认。

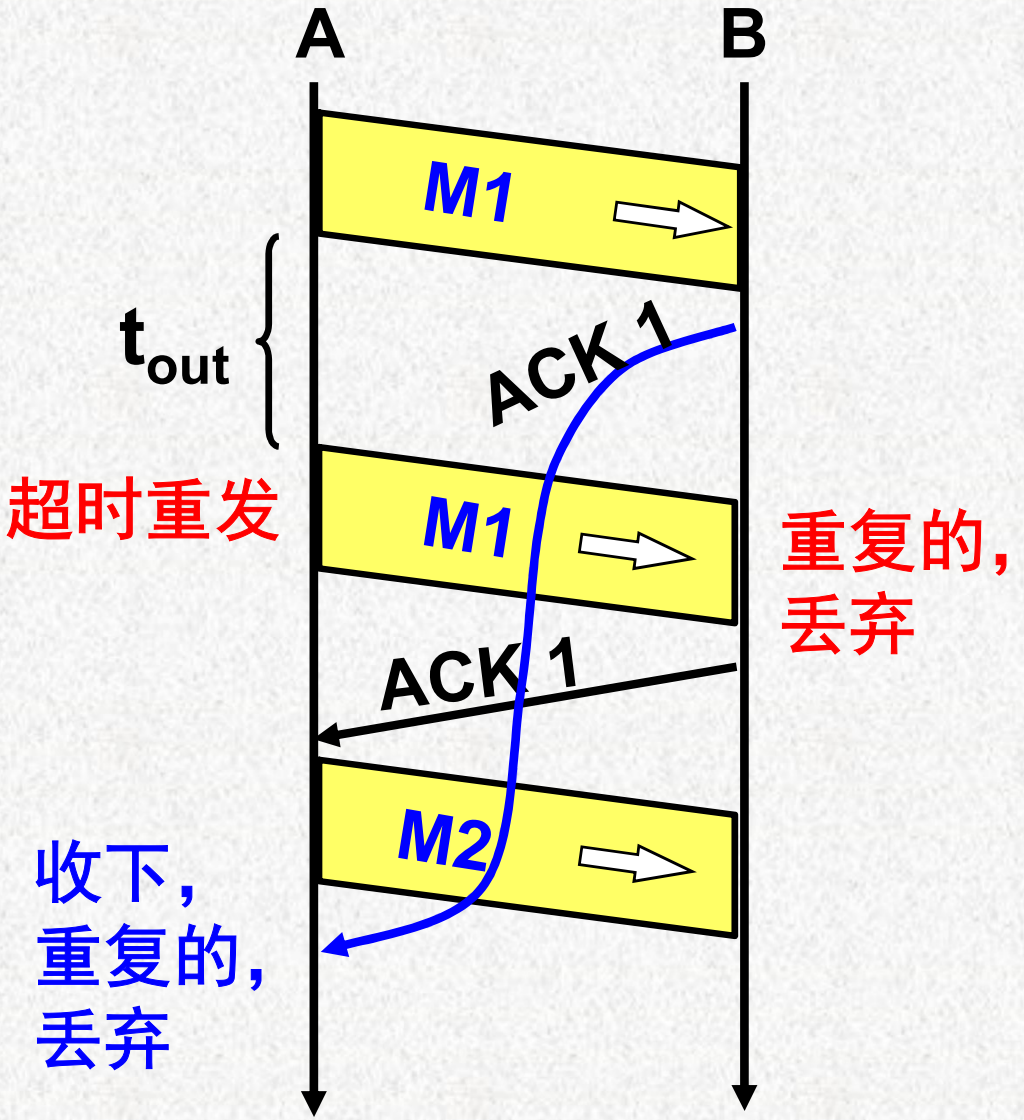
3. 确认丢失和确认迟到

- 确认迟到
  - 传输过程中没有出现差错，但 B 对分组 M1 的确认迟到了。
  - A 会收到重复的确认。对重复的确认的处理很简单：收下后就丢弃。
  - B 仍然会收到重复的 M1，并且同样要丢弃重复的 M1，并重传确认分组。

3. 确认丢失和确认迟到



确认丢失



确认迟到

请注意

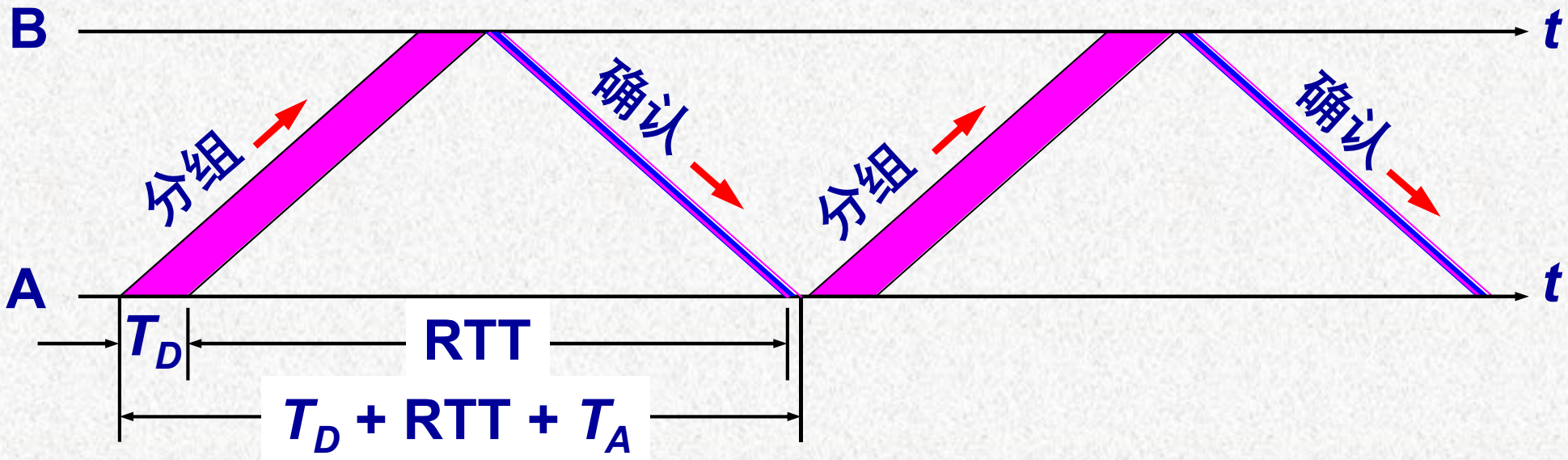
- 在发送完一个分组后，必须暂时保留已发送的分组的副本，以备重发。
- 分组和确认分组都必须进行编号。
- 超时计时器的重传时间应当比数据在分组传输的平均往返时间更长一些。

自动重传请求 ARQ

- 通常 A 最终总是可以收到对所有发出的分组的确认。如果 A 不断重传分组但总是收不到确认，就说明通信线路太差，不能进行通信。
- 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。
- 像上述的这种可靠传输协议常称为自动重传请求 ARQ (Automatic Repeat reQuest)。意思是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。

4. 信道利用率

停止等待协议的优点是简单，缺点是信道利用率太低。



停止等待协议的信道利用率太低

信道利用率 $U = \dfrac{T_D}{T_D + \mathrm{RTT} + T_A}$ **(5-3)**

4. 信道利用率

- 可以看出，当往返时间 RTT 远大于分组发送时间 T_D 时，信道的利用率就会非常低。
- 若出现重传，则对传送有用的数据信息来说，信道的利用率就还要降低。

例子：停止等待协议信道利用率计算

- **例: 1 Gbps 的链路, 15 ms 的传播时延, 8KB的分组:**

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

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

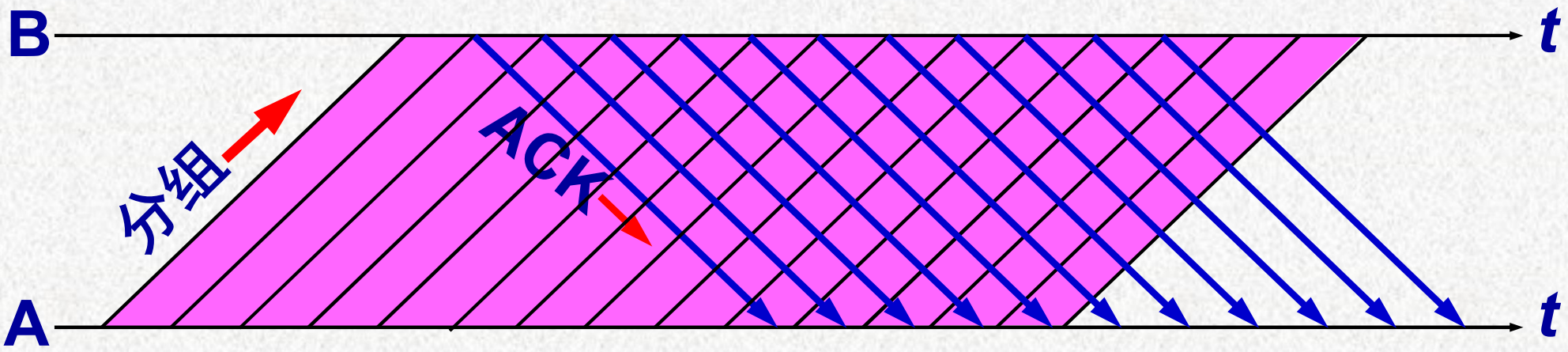
- U_{sender}: 发送方利用率 - 发送方将发送比特送进信道的时间与发送时间之比
- 30.008 ms 只能发送1KB -> 1Gbps的链路上有效的吞吐量为267kb/s
- stop-and-wait协议的通病----造成网络资源的利用率过低
- 网络协议限制底层网络硬件所提供的带宽！

流水线传输

- 为了提高传输效率，发送方可以不使用低效率的停止等待协议，而是采用流水线传输。
- 流水线传输就是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。这样可使信道上一直有数据不间断地传送。
- 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。

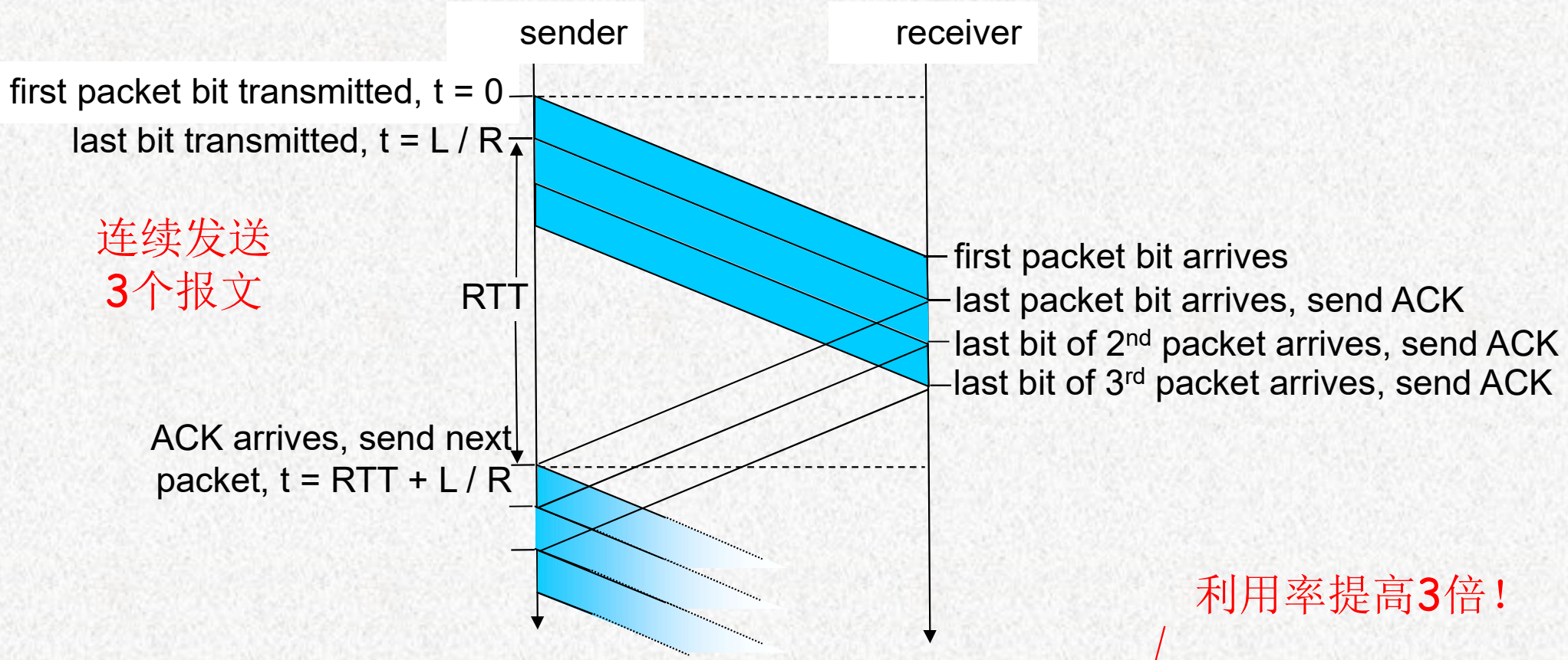
流水线传输

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。



流水线传输可提高信道利用率

例子：流水线传输的信道利用率



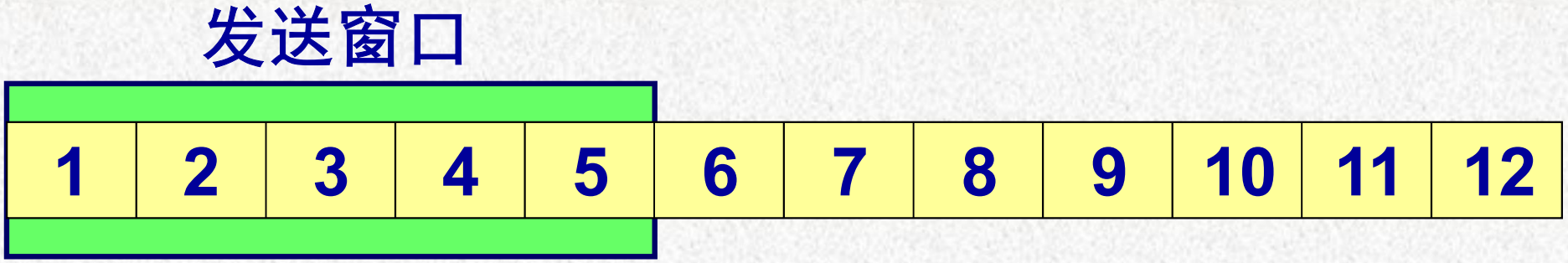
利用率提高3倍！

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

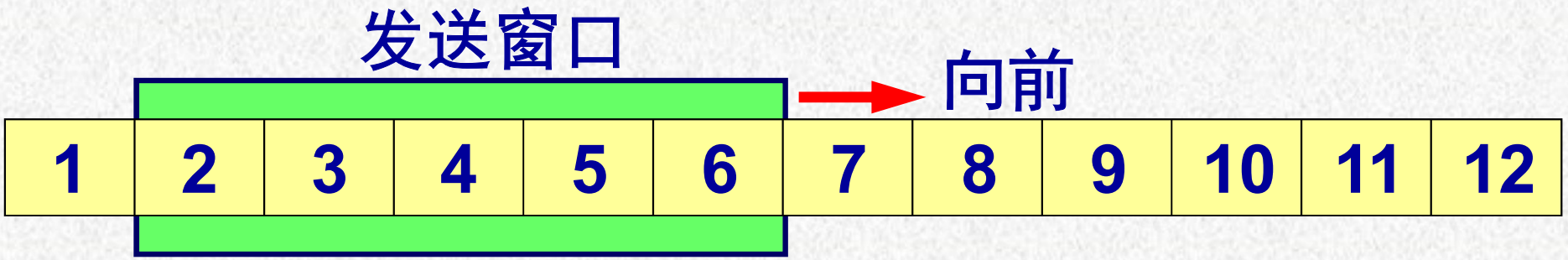
5.4.2 连续 ARQ 协议

- 滑动窗口协议比较复杂，TCP 协议的精髓所在。
- 发送方维持的发送窗口，它的意义是：位于发送窗口内的分组都可连续发送出去，而不需要等待对方的确认。这样，信道利用率就提高了。
- 连续 ARQ 协议规定，发送方每收到一个确认，就把发送窗口向前滑动一个分组的位置。

5.4.2 连续ARQ协议



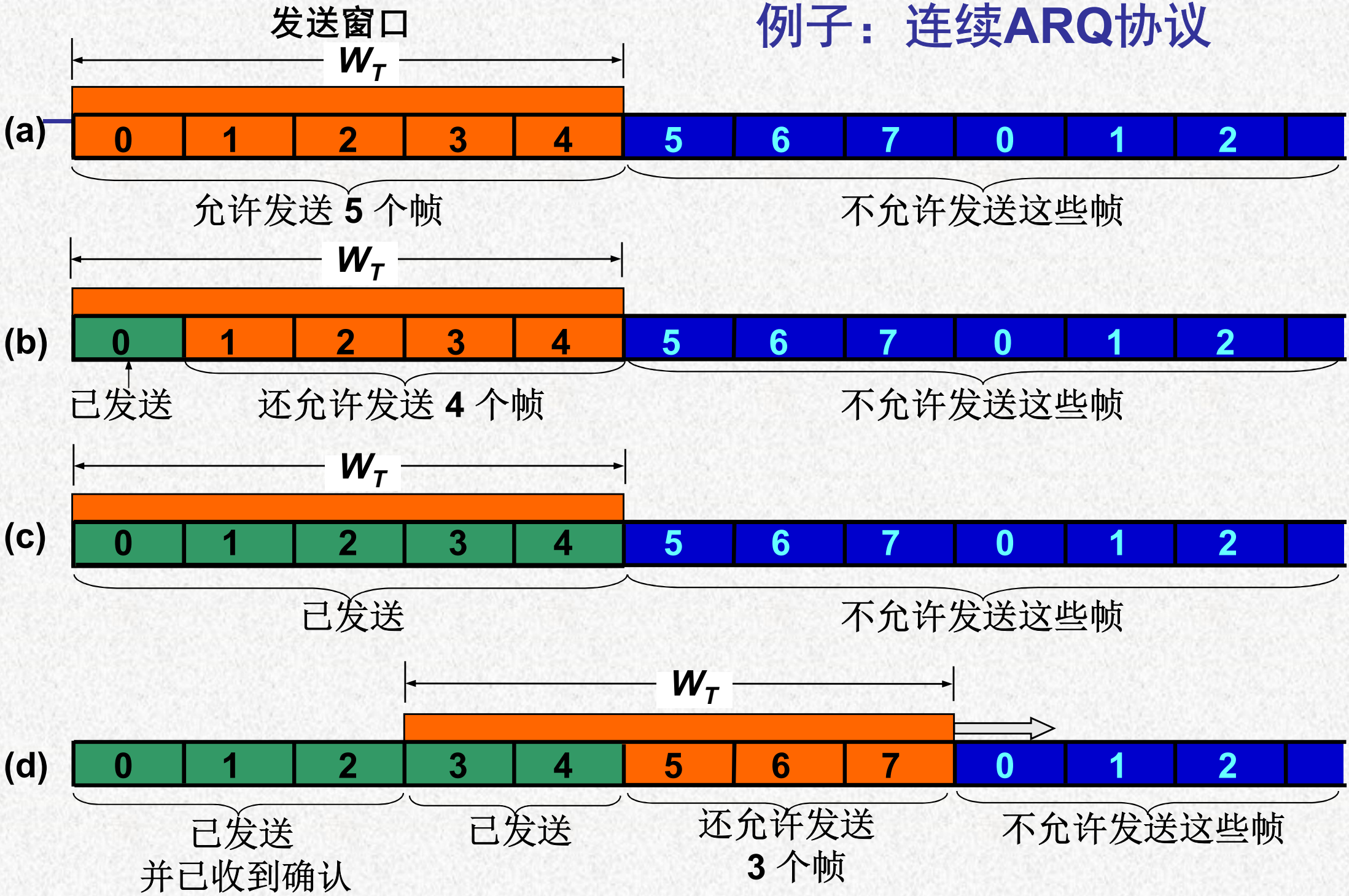
(a) 发送方维持发送窗口（发送窗口是 5）



(b) 收到一个确认后发送窗口向前滑动

连续 ARQ 协议的工作原理

例子：连续ARQ协议


发送窗口
W_T
(a)
0 1 2 3 4 5 6 7 0 1 2
允许发送 5 个帧
不允许发送这些帧
W_T
(b)
0 1 2 3 4 5 6 7 0 1 2
已发送
还允许发送 4 个帧
不允许发送这些帧
W_T
(c)
0 1 2 3 4 5 6 7 0 1 2
已发送
不允许发送这些帧
W_T
(d)
0 1 2 3 4 5 6 7 0 1 2
已发送
并已收到确认
已发送
还允许发送
3 个帧
不允许发送这些帧

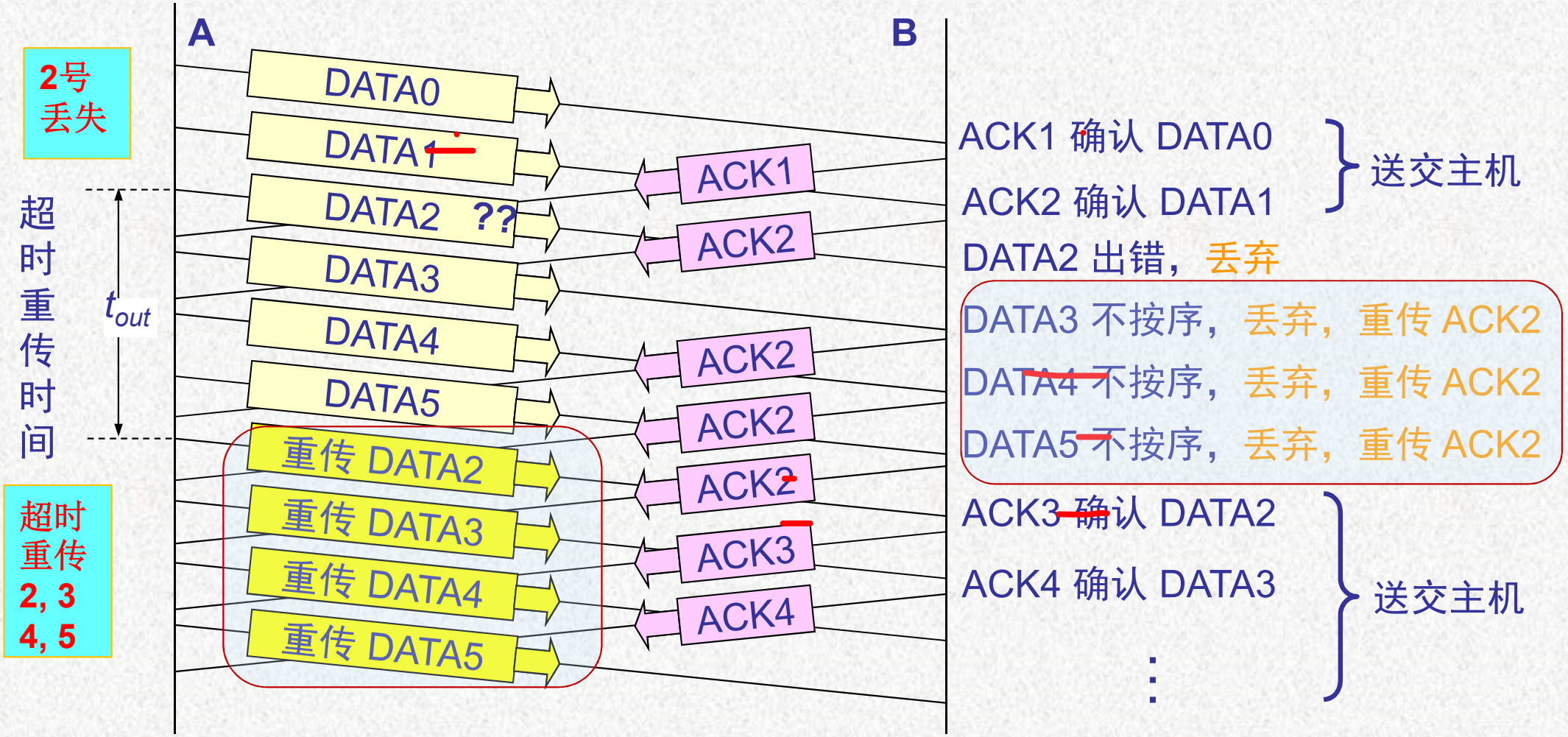
累积确认

- 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。
- 优点：容易实现，即使确认丢失也不必重传。
- 缺点：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）

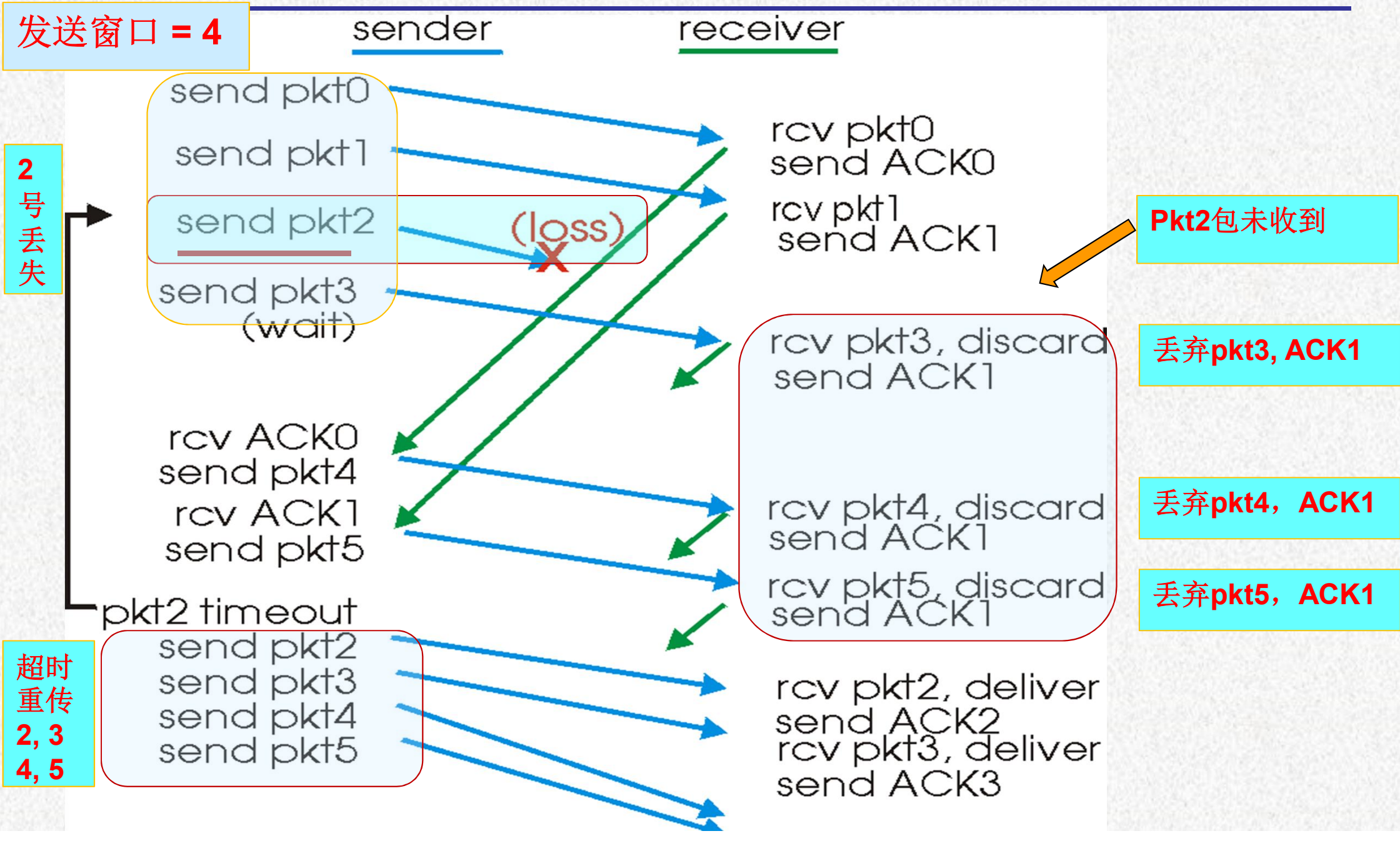
- 如果发送方发送了前 5 个分组，而中间的第 3 个分组丢失了。这时接收方只能对前两个分组发出确认。发送方无法知道后面三个分组的下落，而只好把后面的三个分组都再重传一次。
- 这就叫做 Go-back-N（回退 N），表示需要再退回来重传已发送过的 N 个分组。
- 可见当通信线路质量不好时，连续 ARQ 协议会带来负面的影响。

例子：连续 ARQ 协议的回退 N 工作机制



例子：Go-back-N（回退 N）

发送窗口 = 4



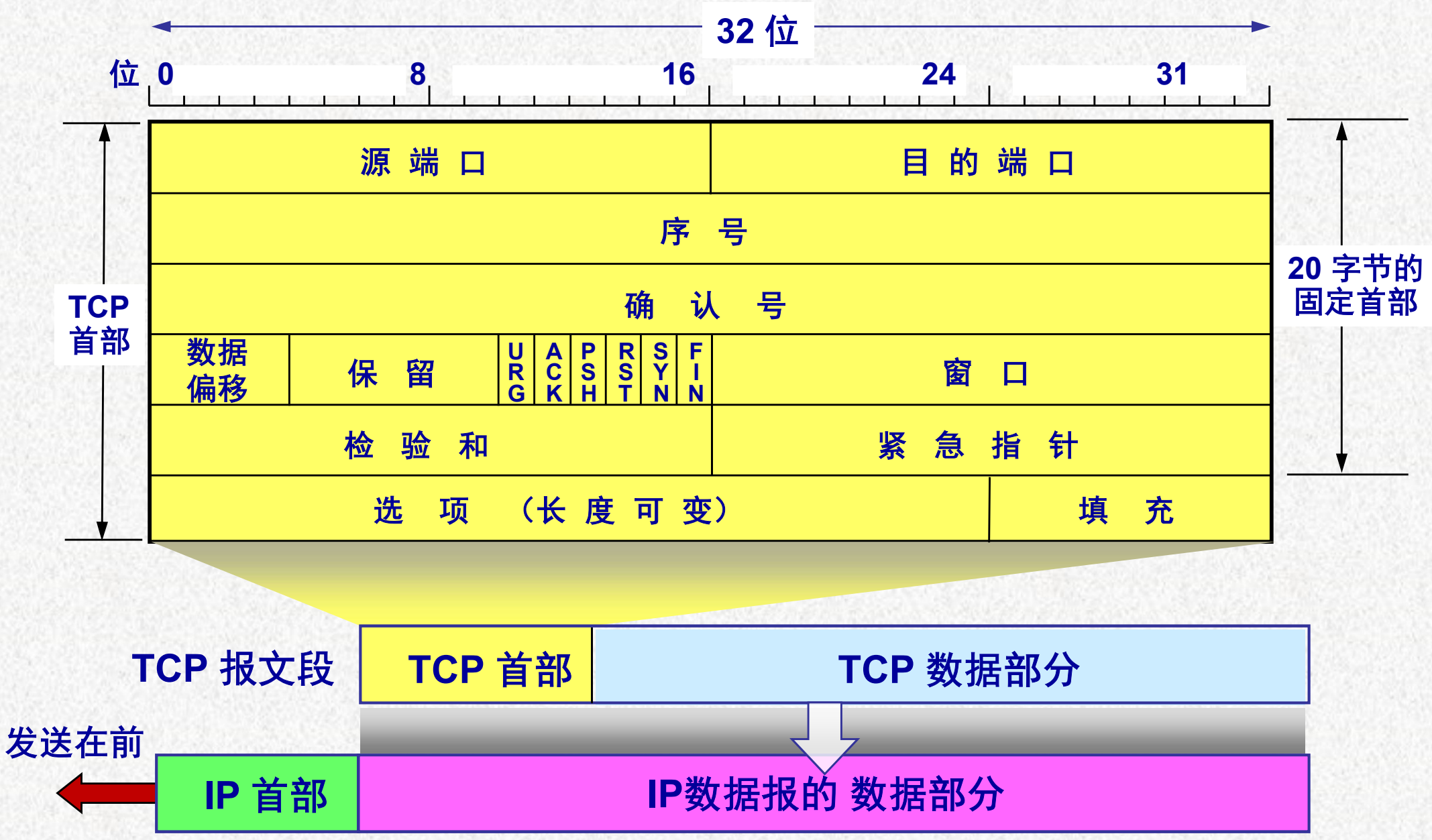
TCP 可靠通信的具体实现

- 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。

5.5 TCP 报文段的首部格式

- TCP 虽然是面向字节流的，但 TCP 传送的数据单元却是报文段。
- 一个 TCP 报文段分为首部和数据两部分，而 TCP 的全部功能都体现在它首部中各字段的作用。
- TCP 报文段首部的前 20 个字节是固定的，后面有 4*n* 字节是根据需要而增加的选项 (*n* 是整数)。因此 TCP 首部的最小长度是 20 字节。

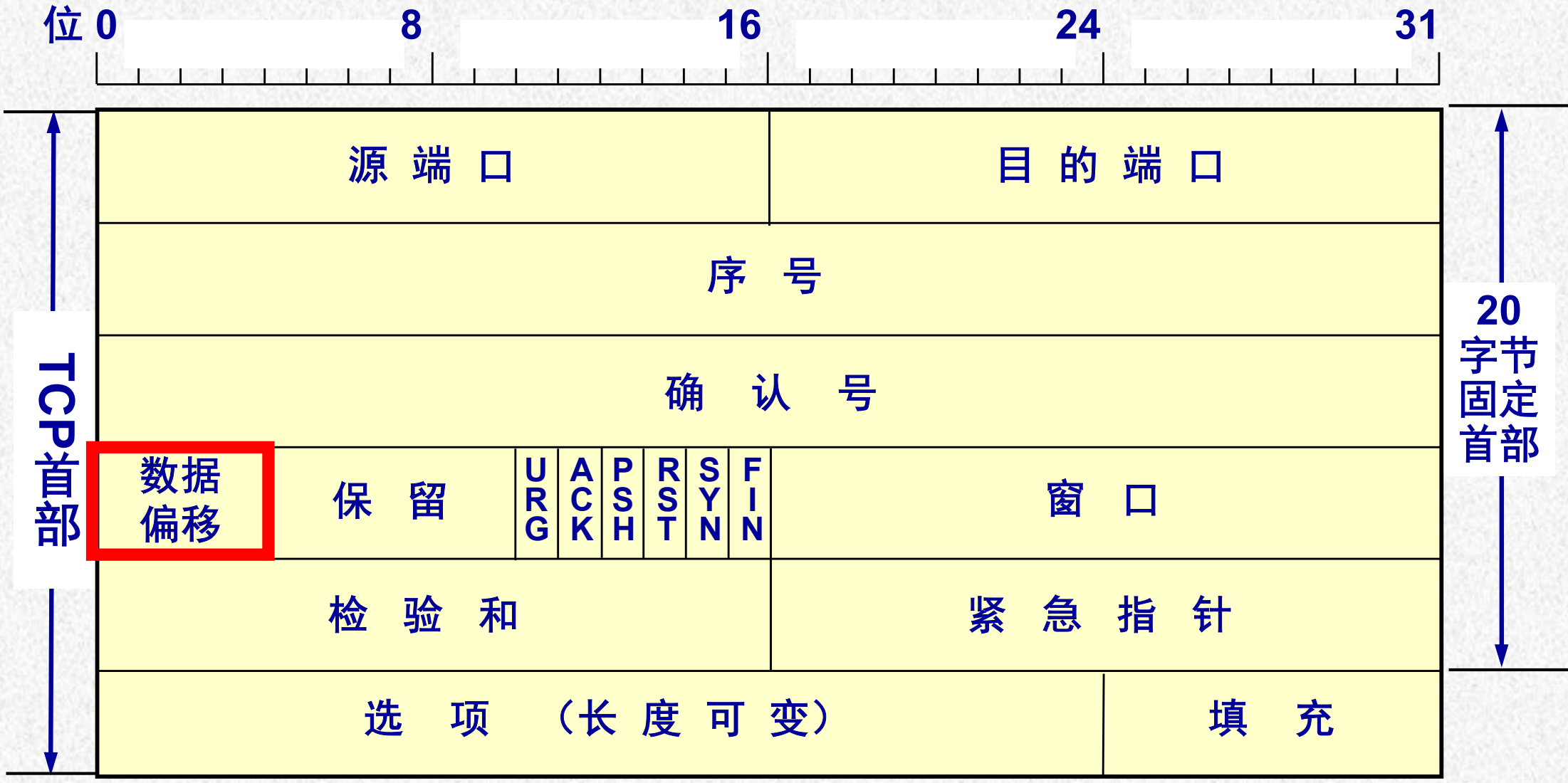
TCP 报文段的首部格式



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。

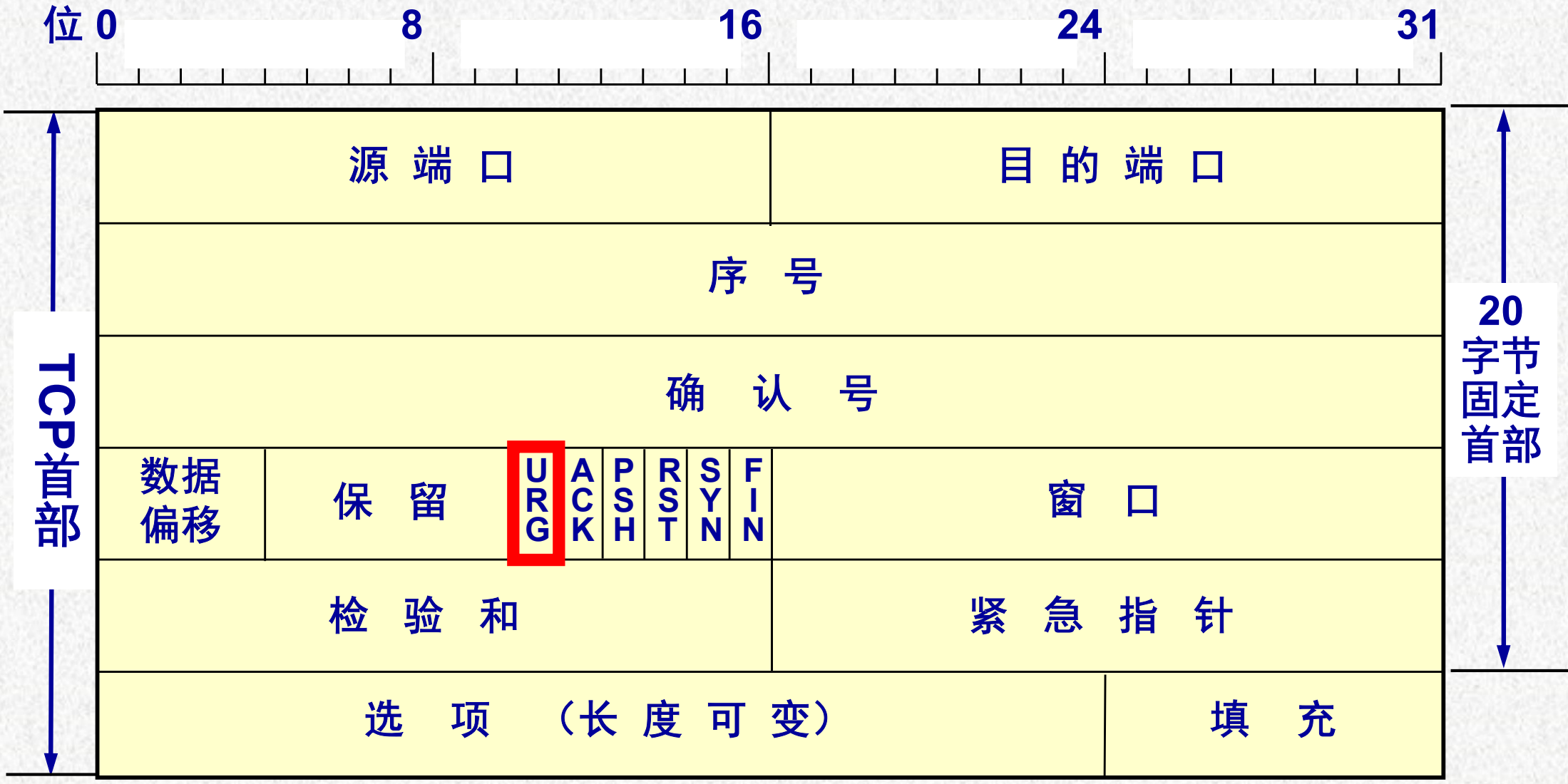
序号字段——占 4 字节。TCP 连接中传送的数据流中的每一个字节都编上一个序号。序号字段的值则指的是本报文段所发送的数据的第一个字节的序号。

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。

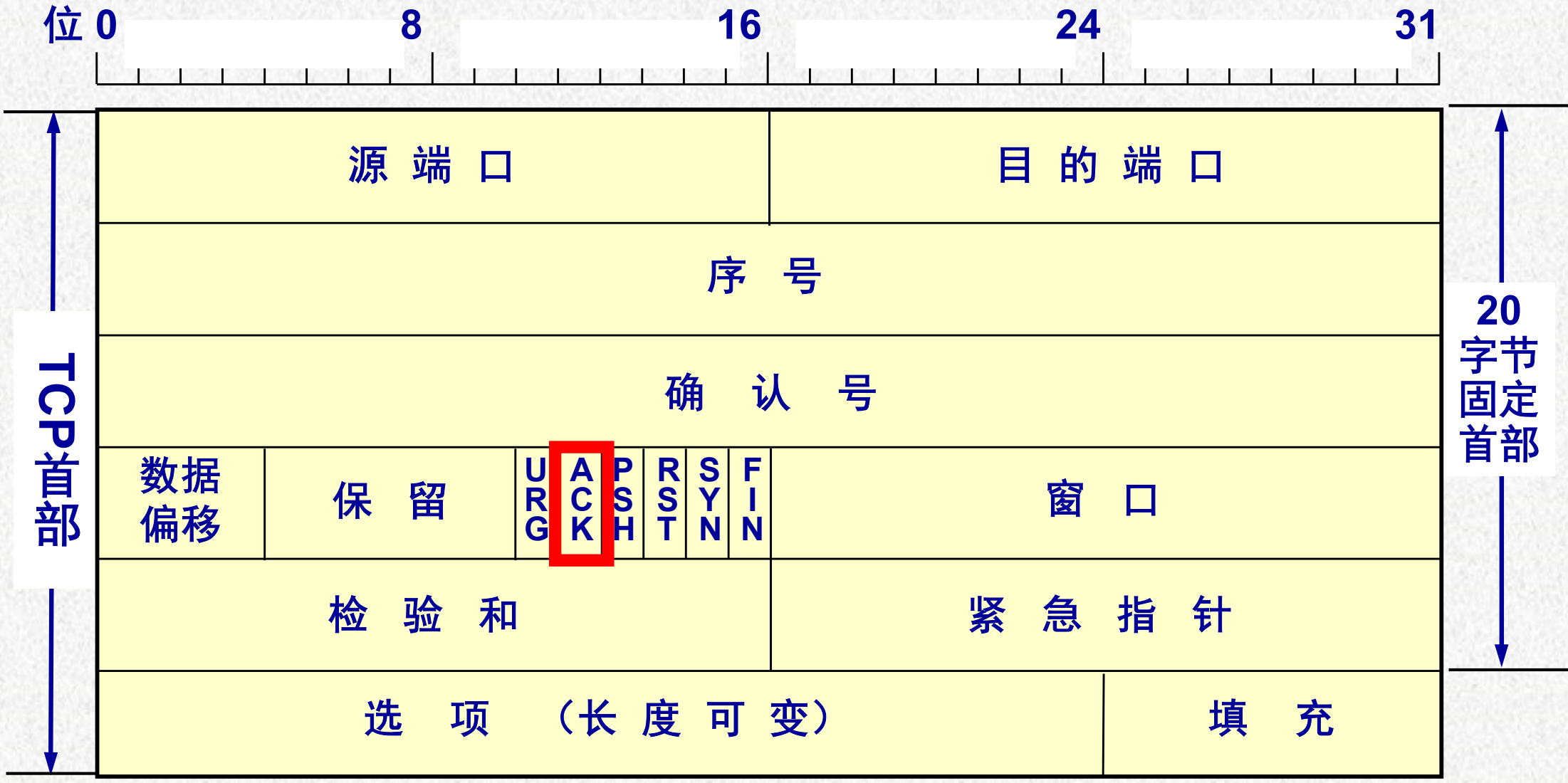


数据偏移（即首部长度）——占 4 位，它指出 TCP 报文段的数据起始处距离 TCP 报文段的起始处有多远。“数据偏移”的单位是 32 位字（以 4 字节为计算单位）。

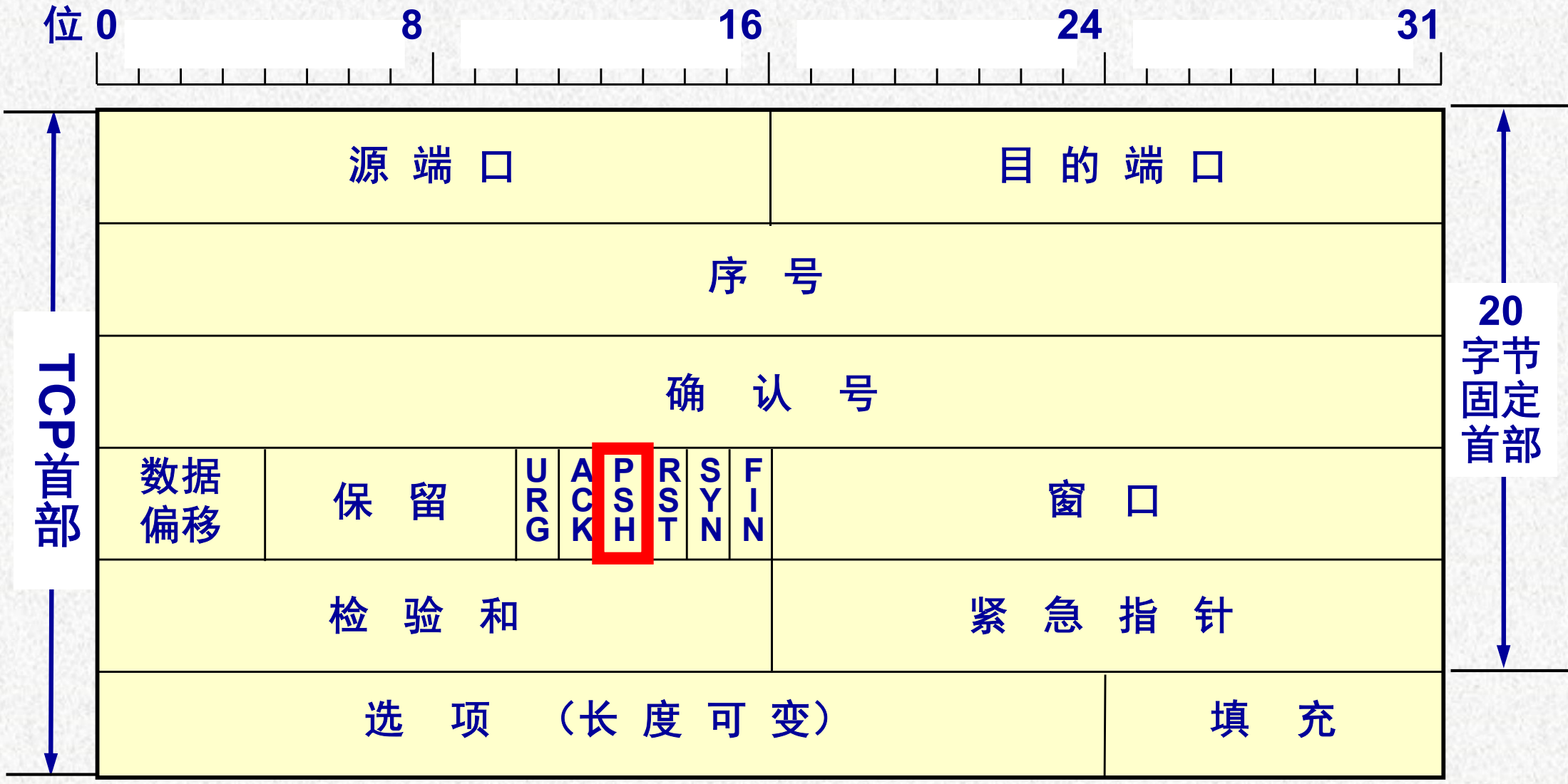
保留字段——占 6 位，保留为今后使用，但目前应置为 0。



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。

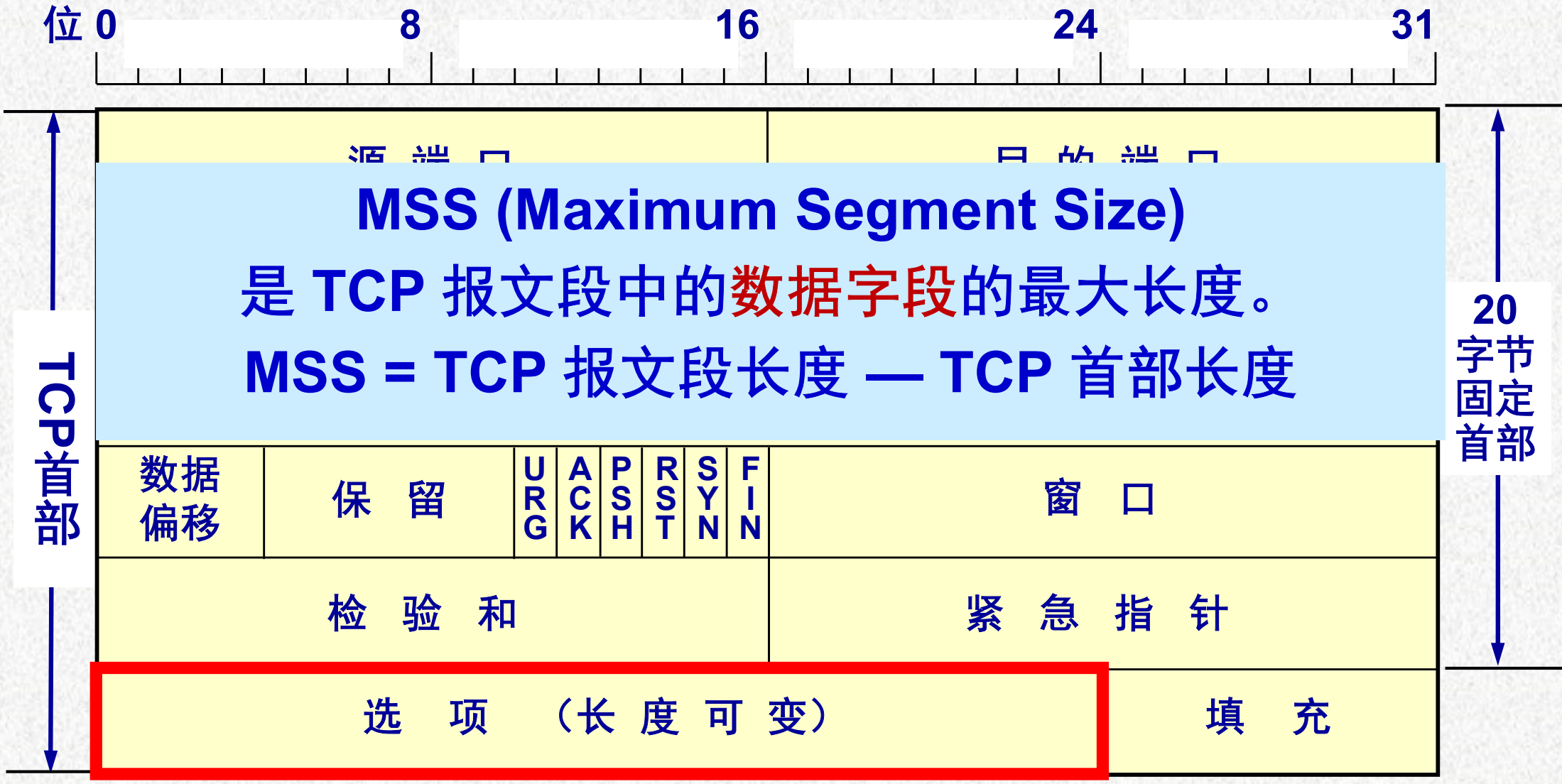
同步 SYN —— 同步 SYN = 1 表示这是一个连接请求或连接接受报文。

终止 FIN (FINish) —— 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。

MSS = TCP 报文段长度 — TCP 首部长度

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。”

为什么要规定 MSS ？

- MSS 与接收窗口值没有关系。
- 若选择较小的 MSS 长度，网络的利用率就降低。
- 当 TCP 报文段只含有 1 字节的数据时，在 IP 层传输的数据报的开销至少有 40 字节(包括 TCP 报文段的首部和 IP 数据报的首部)。这样，对网络的利用率就不会超过 1/41。到了数据链路层还要加上一些开销。
- 若 TCP 报文段非常长，那么在 IP 层传输时就有可能要分解成多个短数据报片。在终点要把收到的各个短数据报片装配成原来的 TCP 报文段。当传输出错时还要进行重传。这些也都会使开销增大。

为什么要规定 MSS ？

- 因此，MSS 应尽可能大些，只要在 IP 层传输时不需要再分片就行。
- 由于 IP 数据报所经历的路径是动态变化的，因此在这条路径上确定的不需要分片的 MSS，如果改走另一条路径就可能需要进行分片。
- 因此最佳的 MSS 是很难确定的。

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——在后面的 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。

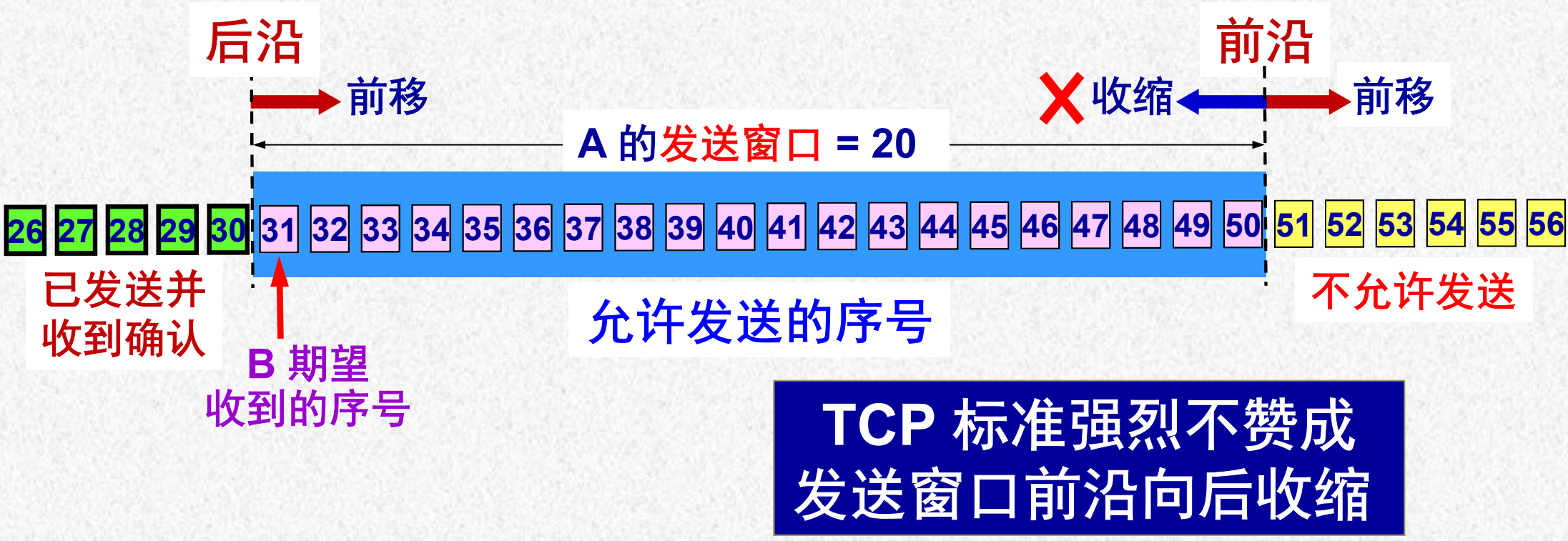
5.6 TCP 可靠传输的实现

- 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

5.6.1 以字节为单位的滑动窗口

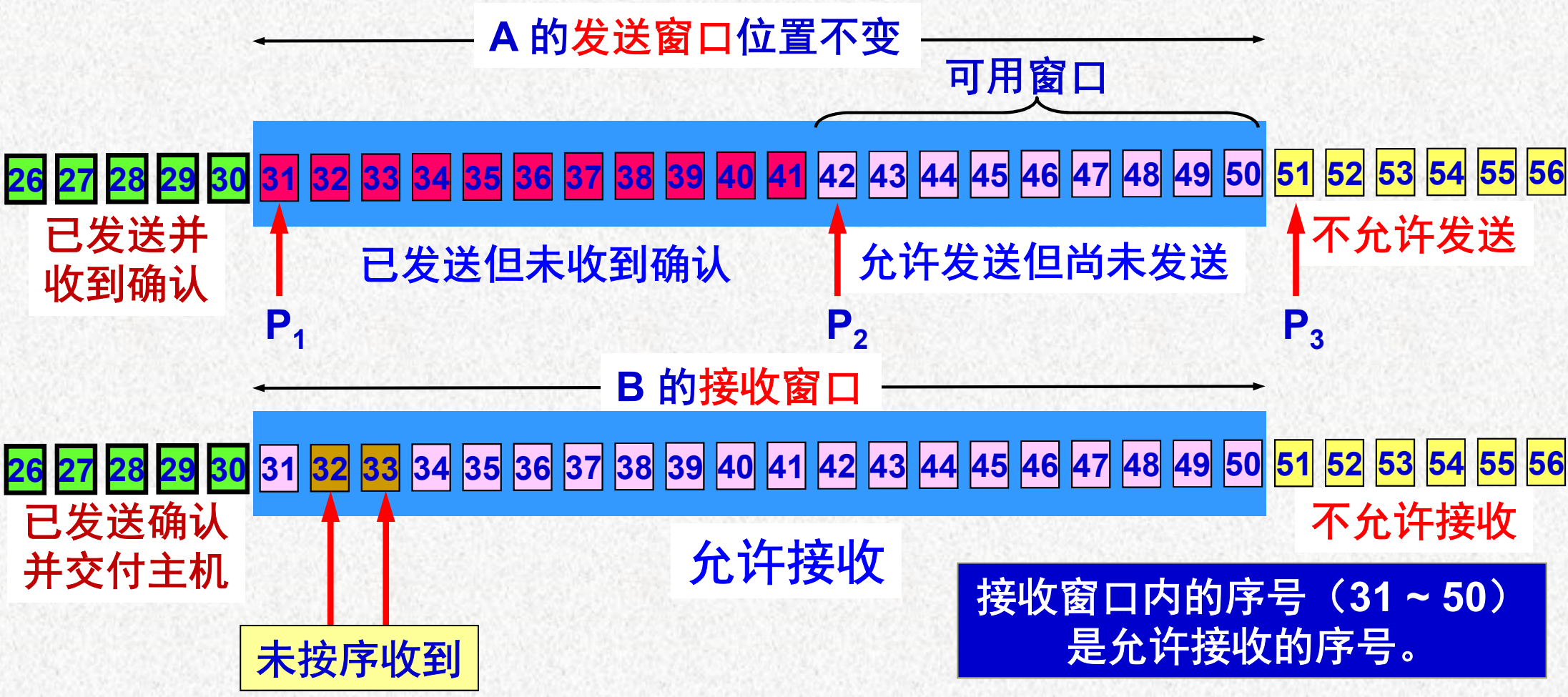
- TCP 的滑动窗口是以字节为单位的。
- 现假定 A 收到了 B 发来的确认报文段，其中窗口是 20 字节，而确认号是 31（这表明 B 期望收到的下一个序号是 31，而序号 30 为止的数据已经收到了）。
- 根据这两个数据，A 就构造出自己的发送窗口，

- 根据 B 给出的窗口值，A 构造出自己的发送窗口。
- 发送窗口表示：在没有收到 B 的确认的情况下，A 可以连续把窗口内的数据都发送出去。
- 发送窗口里面的序号表示允许发送的序号。
- 显然，窗口越大，发送方就可以在收到对方确认之前连续发送更多的数据，因而可能获得更高的传输效率。



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

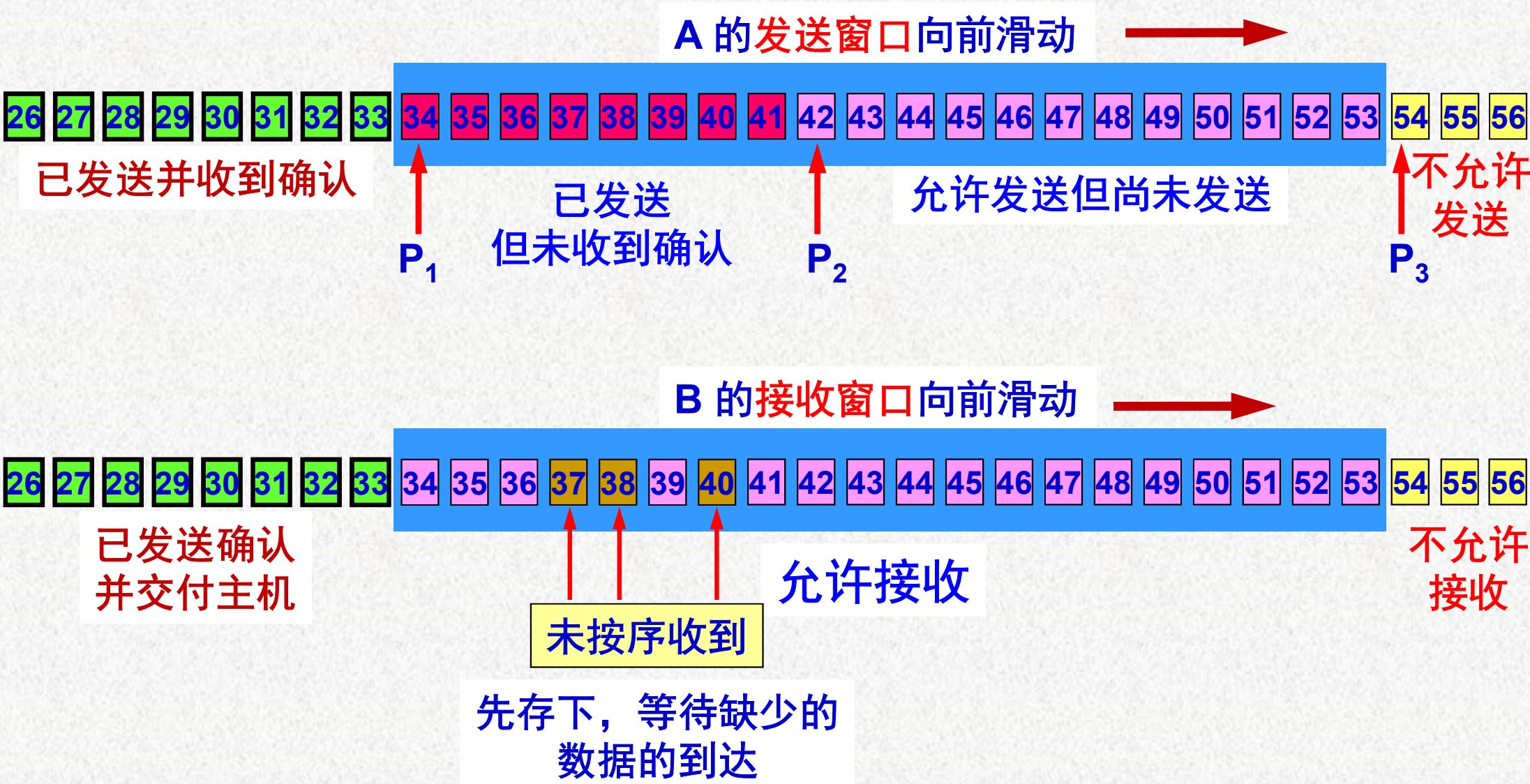


$P_3 - P_1$ = A 的发送窗口（又称为通知窗口）

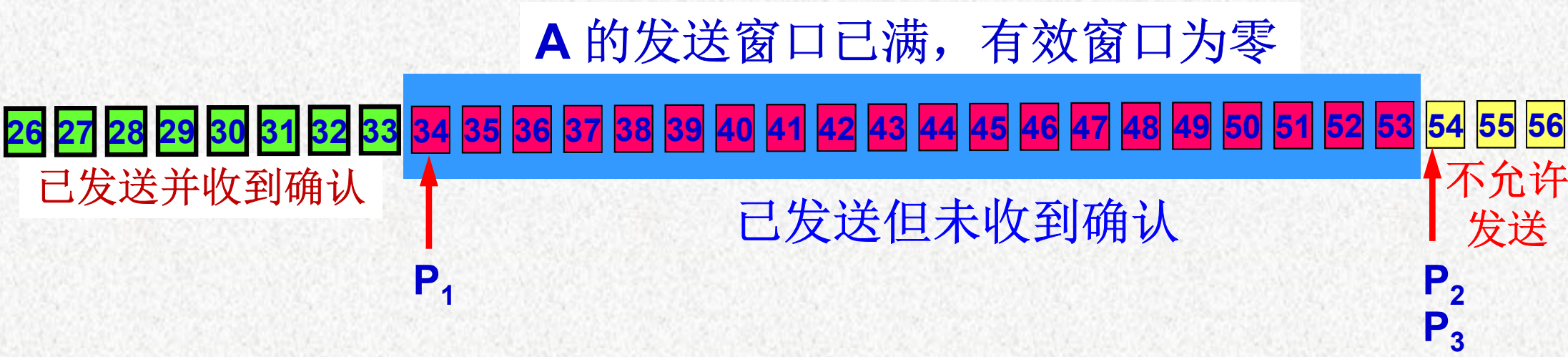
$P_2 - P_1$ = 已发送但尚未收到确认的字节数

$P_3 - P_2$ = 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。

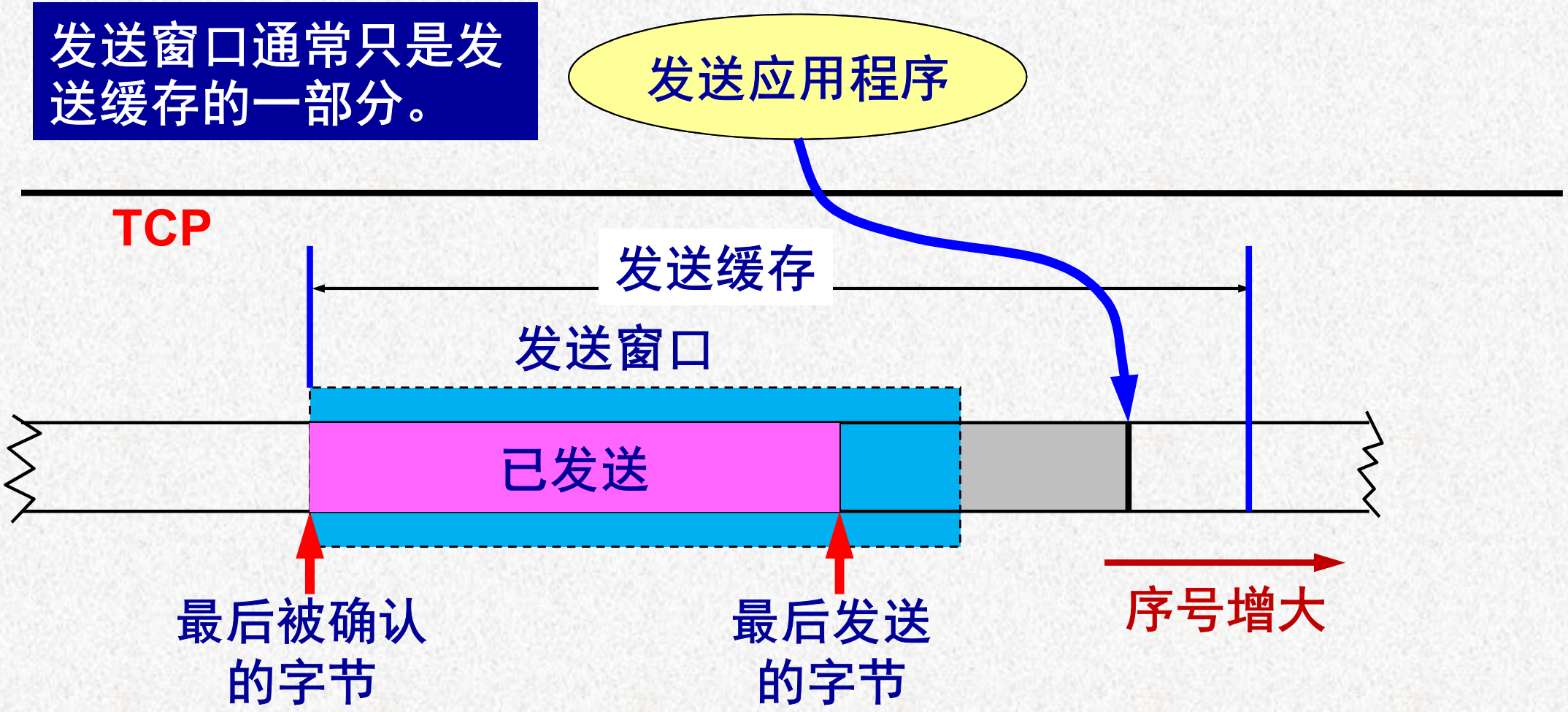


发送窗口内的序号都属于已发送但未被确认

发送缓存

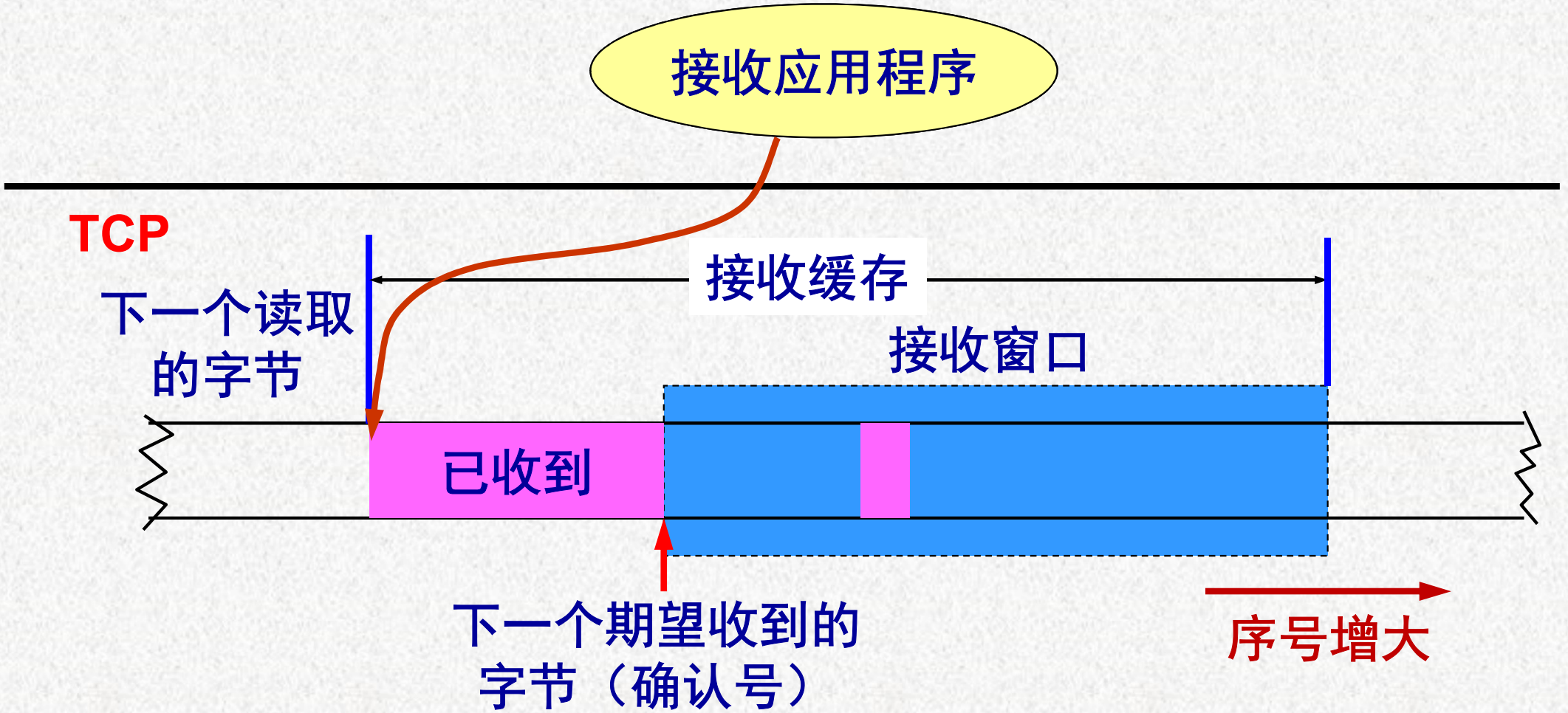
发送方的应用进程把字节流写入 TCP 的发送缓存。

发送窗口通常只是发送缓存的一部分。



接收缓存

接收方的应用进程从 TCP 的接收缓存中读取字节流。



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

- 第一，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）。
- 第二，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。
- 第三，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。

接收方发送确认

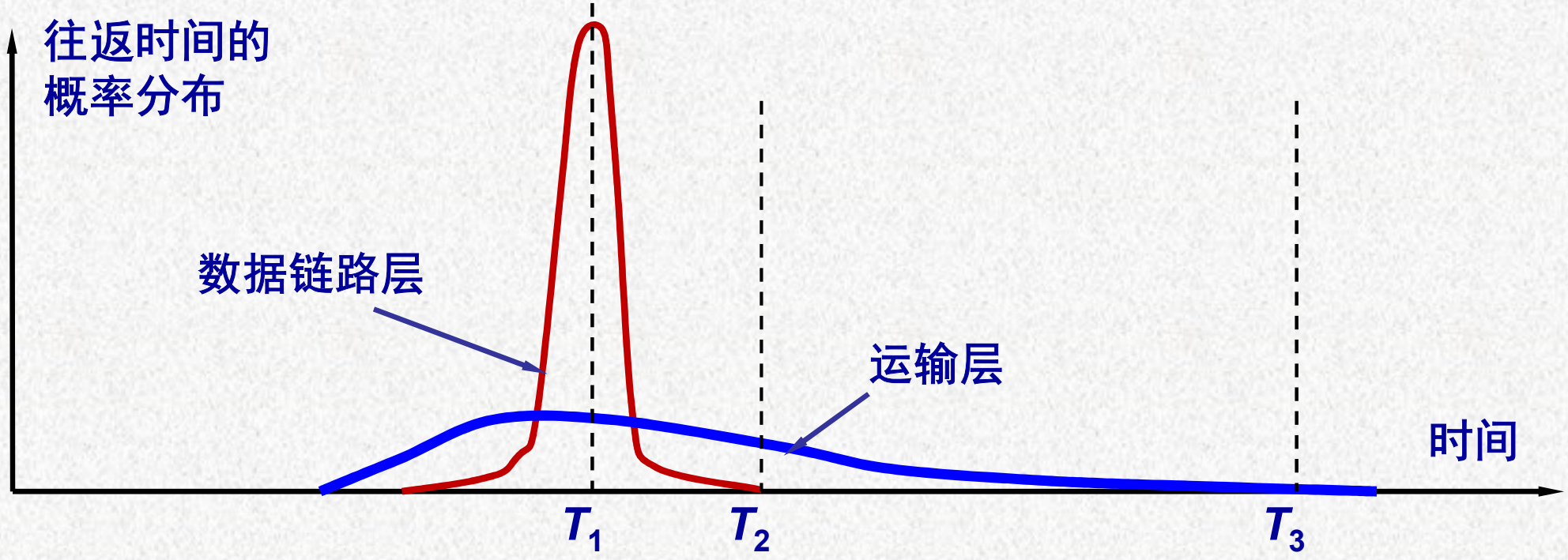
- 接收方可以在合适的时候发送确认，也可以在自己有数据要发送时把确认信息顺便捎带上。
- 但请注意两点：
  - 第一，接收方不应过分推迟发送确认，否则会导致发送方不必要的重传，这反而浪费了网络的资源。。
  - 第二，捎带确认实际上并不经常发生，因为大多数应用程序很少同时在两个方向上发送数据。

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。
- 重传时间的选择是 TCP 最复杂的问题之一。

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 (RTT) 的方差也很大。



TCP 超时重传时间设置

- 如果把超时重传时间设置得太短，就会引起很多报文段的不必要的重传，使网络负荷增大。
- 但若把超时重传时间设置得过长，则又使网络的空闲时间增大，降低了传输效率。
- **TCP 采用了一种自适应算法，**它记录一个报文段发出的时间，以及收到相应的确认的时间。这两个时间之差就是报文段的往返时间 **RTT**。

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：

$$\text{新的}RTT_S = (1-\alpha) \times (\text{旧的}RTT_S) + \alpha \times (\text{新的RTT样本}) \quad (5\text{-}4)$$

- 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

- RTO (Retransmission Time-Out) 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：

$$RTO = RTT_S + 4 \times RTT_D \qquad (5\text{-}5)$$

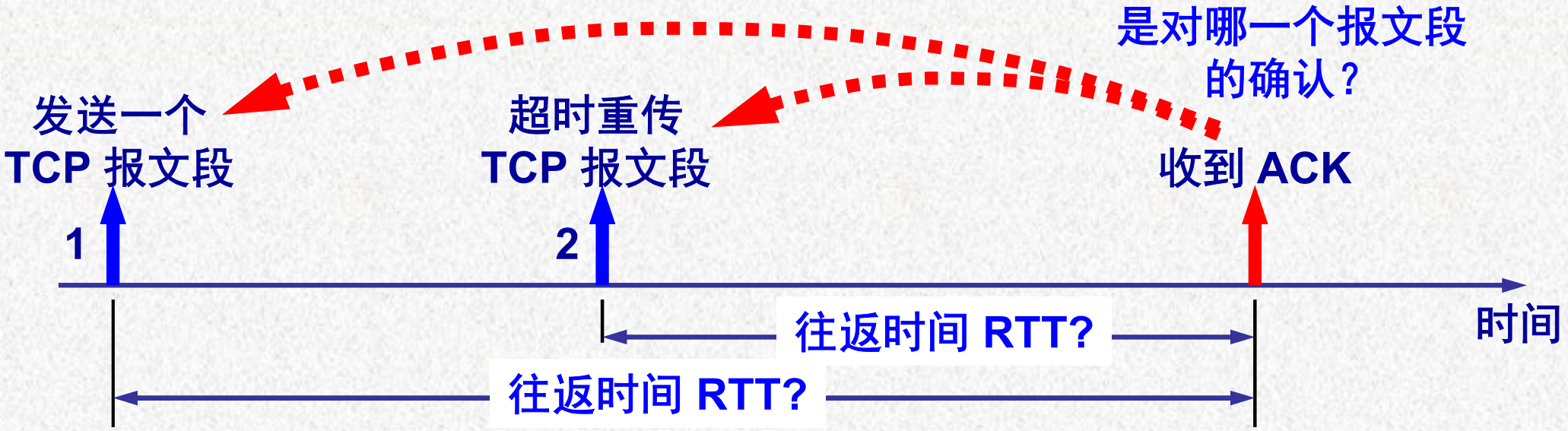
- RTT_D 是 RTT 的偏差的加权平均值。
- RFC 2988 建议这样计算 RTT_D。第一次测量时，RTT_D 值取为测量到的 RTT 样本值的一半。在以后的测量中，则使用下式计算加权平均的 RTT_D：

$$\text{新的 } RTT_D = (1 - \beta) \times (\text{旧的} RTT_D) + \beta \times |RTT_S - \text{新的 RTT 样本}| \qquad (5\text{-}6)$$

- β 是个小于 1 的系数，其推荐值是 1/4，即 0.25。

往返时间 (RTT) 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。
- 但是，这又引起新的问题。当报文段的时延突然增大了很多时，在原来得出的重传时间内，不会收到确认报文段。于是就重传报文段。但根据 Karn 算法，不考虑重传的报文段的往返时间样本。这样，超时重传时间就无法更新。

修正的 Karn 算法

- 报文段每重传一次，就把 RTO 增大一些：

新的 RTO = γ × (旧的 RTO)

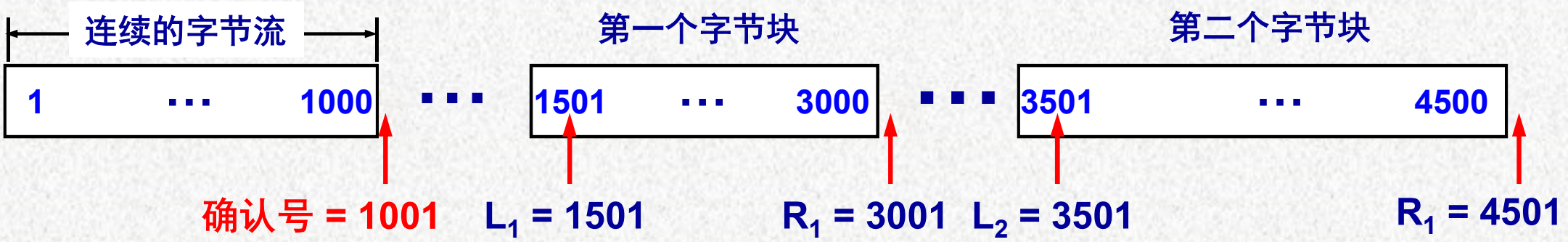
- 系数 γ 的典型值是 2 。
- 当不再发生报文段的重传时，才根据报文段的往返时延更新平均往返时延 RTT 和超时重传时间 RTO 的数值。
- 实践证明，这种策略较为合理。

5.6.3 选择确认 SACK

- 问题：若收到的报文段无差错，只是未按序号，中间还缺少一些序号的数据，那么能否设法只传送缺少的数据而不重传已经正确到达接收方的数据？
- 答案是可以的。选择确认 SACK (Selective ACK) 就是一种可行的处理方法。

接收到的字节流序号不连续

TCP 的接收方在接收对方发送过来的数据字节流的序号不连续，结果就形成了一些不连续的字节块。



和前后字节不连续的每一个字节块都有两个边界：边界和右边界。

- 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 $R_2 = 4501$。

5.6.3 选择确认 SACK

- 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。

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。

5.7 TCP 的流量控制

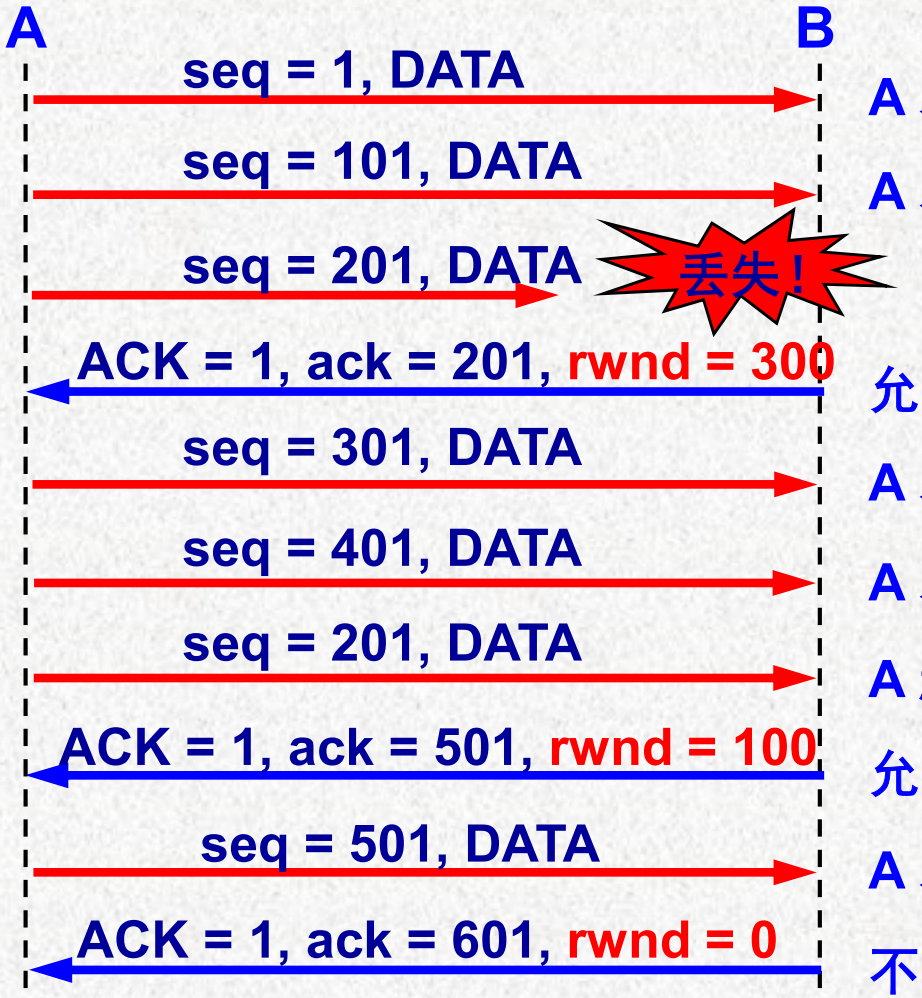
- 5.7.1 利用滑动窗口实现流量控制
- 5.7.2 TCP 的传输效率

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。

利用可变窗口进行流量控制举例

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）

可能发生死锁

- B 向 A 发送了零窗口的报文段后不久，B 的接收缓存又有了一些存储空间。于是 B 向 A 发送了 rwnd = 400 的报文段。
- 但这个报文段在传送过程中丢失了。A 一直等待收到 B 发送的非零窗口的通知，而 B 也一直等待 A 发送的数据。
- 如果没有其他措施，这种互相等待的死锁局面将一直延续下去。
- 为了解决这个问题，TCP 为每一个连接设有一个持续计时器 (persistence timer)。

持续计时器

- TCP 为每一个连接设有一个持续计时器 (persistence timer) 。
- 只要 TCP 连接的一方收到对方的零窗口通知，就启动该持续计时器。
- 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 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。
- 如何控制 TCP 发送报文段的时机仍然是一个较为复杂的问题。

发送方糊涂窗口综合症

- 发送方 TCP 每次接收到一字节的数据后就发送。
- 这样，发送一个字节需要形成 41 字节长的 IP 数据报。若接收方确认，并回送这一字节，就需传送总长度为 162 字节共 4 个报文段。效率很低。
- 解决方法：使用 Nagle 算法。

Nagle算法

- 若发送应用进程把要发送的数据逐个字节地送到 TCP 的发送缓存，则发送方就把第一个数据字节先发送出去，把后面到达的数据字节都缓存起来。
- 当发送方收到对第一个数据字符的确认后，再把发送缓存中的所有数据组装成一个报文段发送出去，同时继续对随后到达的数据进行缓存。
- 只有在收到对前一个报文段的确认后才继续发送下一个报文段。
- 当到达的数据已达到发送窗口大小的一半或已达到报文段的最大长度时，就立即发送一个报文段。

接收方糊涂窗口综合症

- 当接收方的 TCP 缓冲区已满，接收方会向发送方发送窗口大小为 0 的报文。
- 若此时接收方的应用进程以交互方式每次只读取一个字节，于是接收方又发送窗口大小为一个字节的更新报文，发送方应邀发送一个字节的数据（发送的 IP 数据报是 41 字节长），于是接收窗口又满了，如此循环往复。
- 解决方法：让接收方等待一段时间，使得或者接收缓存已有足够空间容纳一个最长的报文段，或者等到接收缓存已有一半空闲的空间。只要出现这两种情况之一，接收方就发出确认报文，并向发送方通知当前的窗口大小。

5.8 TCP 的拥塞控制

- 5.8.1 拥塞控制的一般原理
- 5.8.2 TCP 的拥塞控制方法
- 5.8.3 主动队列管理 AQM

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。这种现象称为**拥塞 (congestion)**。
- 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。
- 出现拥塞的**原因**：

$$\sum \text{对资源需求} > \text{可用资源} \qquad (5\text{-}7)$$

增加资源能解决拥塞吗？

- 不能。这是因为网络拥塞是一个非常复杂的问题。简单地采用上述做法，在许多情况下，不但不能解决拥塞问题，而且还可能使网络的性能更坏。
- 网络拥塞往往是由许多因素引起的。例如：
  - 增大缓存，但未提高输出链路的容量和处理机的速度，排队等待时间将会大大增加，引起大量超时重传，解决不了网络拥塞；
  - 提高处理机处理的速率会会将瓶颈转移到其他地方；

拥塞常常趋于恶化

- 如果一个路由器没有足够的缓存空间，它就会丢弃一些新到的分组。
- 但当分组被丢弃时，发送这一分组的源点就会重传这一分组，甚至可能还要重传多次。这样会引起更多的分组流入网络和被网络中的路由器丢弃。
- 可见拥塞引起的重传并不会缓解网络的拥塞，反而会加剧网络的拥塞。

拥塞控制与流量控制的区别

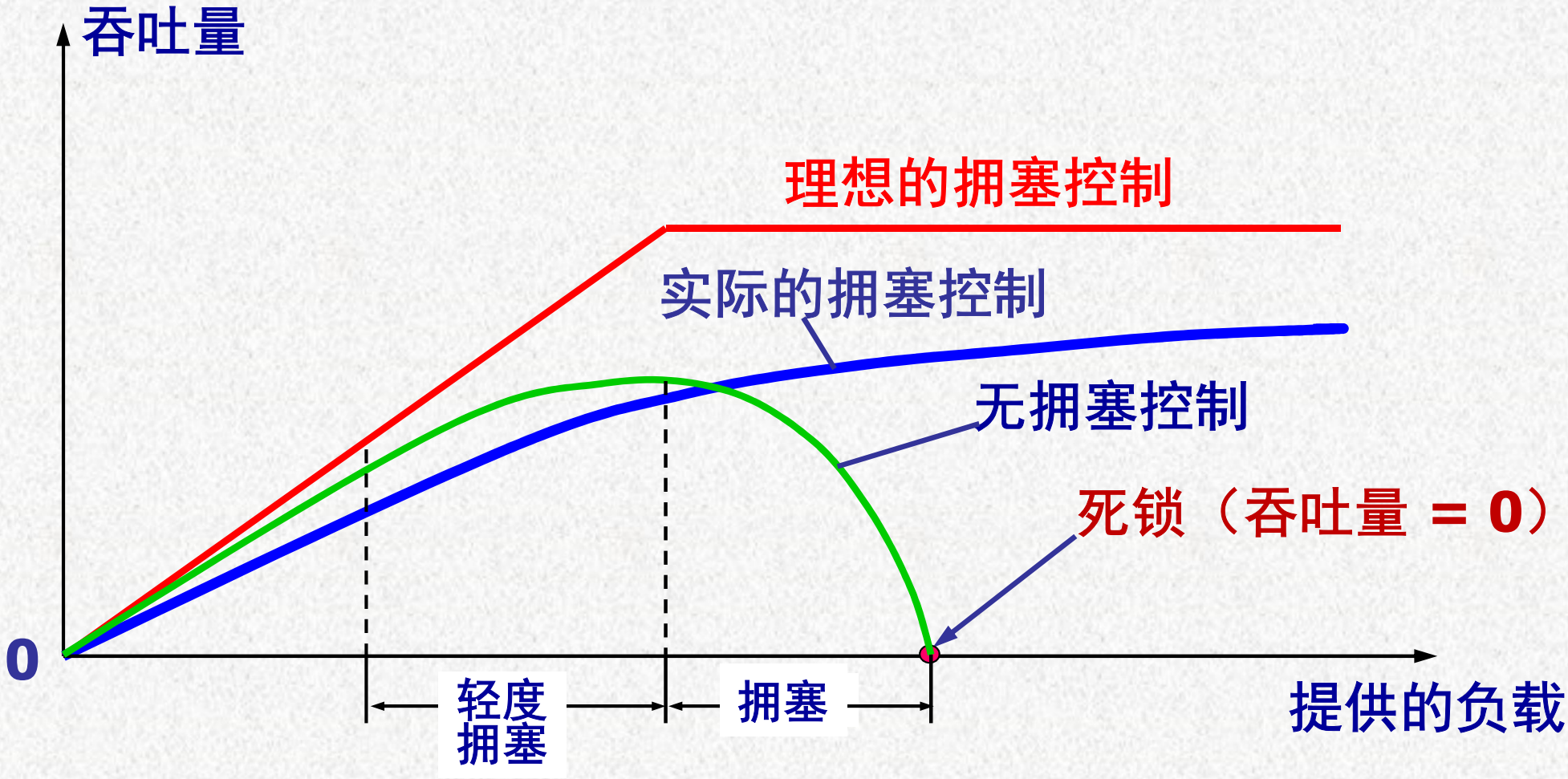
- 拥塞控制就是防止过多的数据注入到网络中，使网络中的路由器或链路不致过载。
- 拥塞控制所要做的都有一个前提，就是网络能够承受现有的网络负荷。
- 拥塞控制是一个全局性的过程，涉及到所有的主机、所有的路由器，以及与降低网络传输性能有关的所有因素。

拥塞控制与流量控制的区别

- 流量控制往往指点对点通信量的控制，是个端到端的问题（接收端控制发送端）。
- 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。

拥塞控制和流量控制之所以常常被弄混，是因为某些拥塞控制算法是向发送端发送控制报文，并告诉发送端，网络已出现麻烦，必须放慢发送速率。这点又和流量控制是很相似的。

拥塞控制所起的作用



拥塞控制的一般原理

- 实践证明，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。
- 闭环控制方法是基于反馈环路的概念。属于闭环控制的有以下几种措施：
  - (1) 监测网络系统以便检测到拥塞在何时、何处发生。
  - (2) 将拥塞发生的信息传送到可采取行动的地方。
  - (3) 调整网络系统的运行以解决出现的问题。

监测网络的拥塞的指标

- 主要指标有：
  - 由于缺少缓存空间而被丢弃的分组的百分数；
  - 平均队列长度；
  - 超时重传的分组数；
  - 平均分组时延；
  - 分组时延的标准差，等等。
- 上述这些指标的上升都标志着拥塞的增长。

5.8.2 TCP 的拥塞控制方法

- TCP 采用基于窗口的方法进行拥塞控制。该方法属于闭环控制方法。
- TCP发送方维持一个拥塞窗口 CWND (Congestion Window)
  - 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。
  - 发送端利用拥塞窗口根据网络的拥塞情况调整发送的数据量。
  - 所以，发送窗口大小不仅取决于接收方公告的接收窗口，还取决于网络的拥塞状况，所以真正的发送窗口值为：

真正的发送窗口值 = Min(公告窗口值，拥塞窗口值)

控制拥塞窗口的原则

- 只要网络没有出现拥塞，拥塞窗口就可以再增大一些，以便把更多的分组发送出去，这样就可以提高网络的利用率。
- 但只要网络出现拥塞或有可能出现拥塞，就必须把拥塞窗口减小一些，以减少注入到网络中的分组数，以便缓解网络出现的拥塞。

拥塞的判断

- 重传定时器超时
  - 现在通信线路的传输质量一般都很好，因传输出差错而丢弃分组的概率是很小的（远小于 1 %）。只要出现了超时，就可以猜想网络可能出现了拥塞。
- 收到三个相同（重复）的 ACK
  - 个别报文段会在网络中丢失，预示可能会出现拥塞（实际未发生拥塞），因此可以尽快采取控制措施，避免拥塞。

TCP拥塞控制算法

- 四种（ **RFC 5681**） ：
  - 慢开始 **(slow-start)**
  - 拥塞避免 **(congestion avoidance)**
  - 快重传 **(fast retransmit)**
  - 快恢复 **(fast recovery)**

慢开始 (Slow start)

- 用来确定网络的负载能力。
- 算法的思路：由小到大逐渐增大拥塞窗口数值。
- 初始拥塞窗口 cwnd 设置：
  - 旧的规定：在刚刚开始发送报文段时，先把初始拥塞窗口 cwnd 设置为 1 至 2 个发送方的最大报文段 SMSS (Sender Maximum Segment Size) 的数值。
  - 新的 RFC 5681 把初始拥塞窗口 cwnd 设置为不超过2至4个 SMSS 的数值。
- 慢开始门限 ssthresh（状态变量）：防止拥塞窗口 cwnd 增长过大引起网络拥塞。

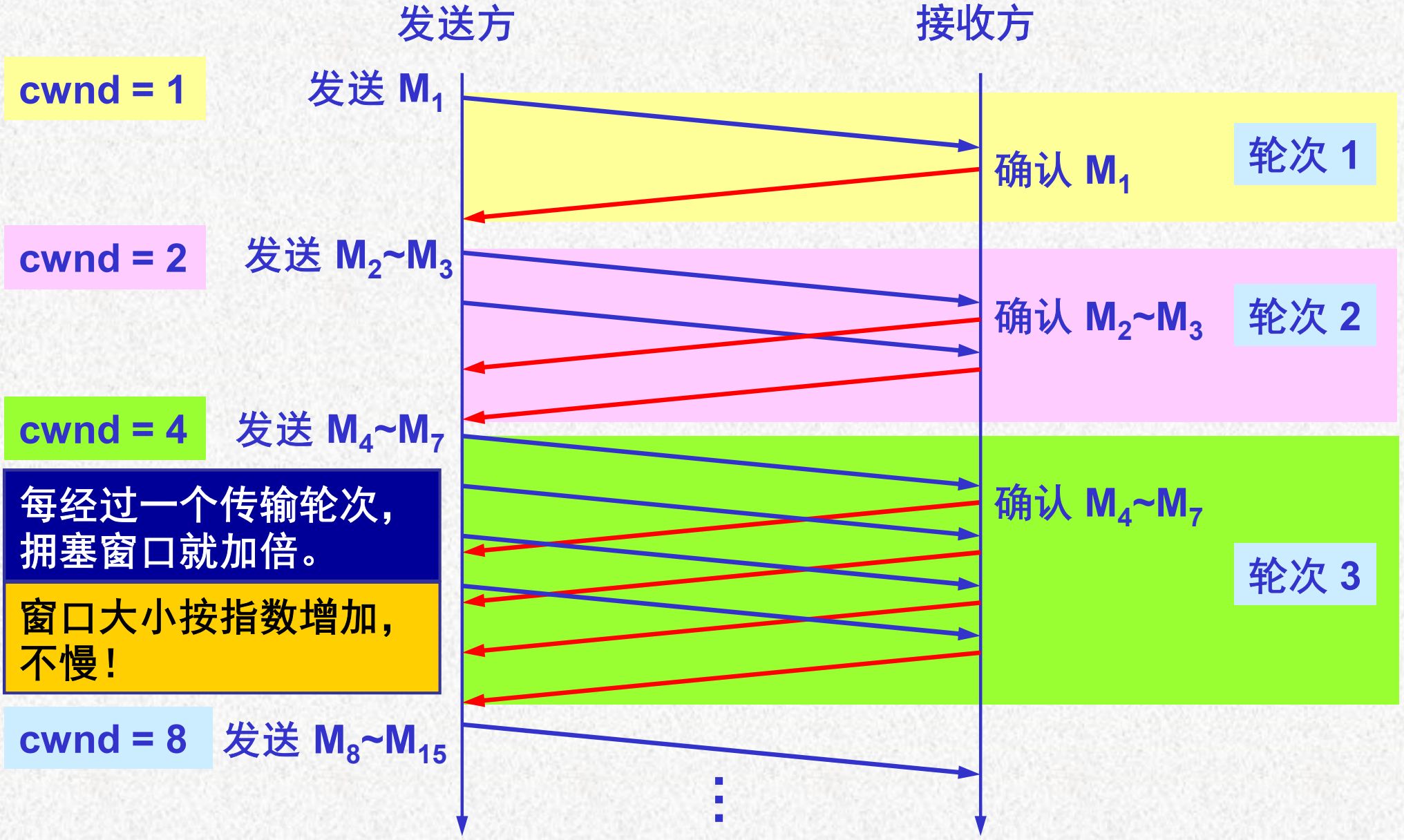
慢开始 (Slow start)

- 拥塞窗口 cwnd 控制方法：在每收到一个对新的报文段的确认后，可以把拥塞窗口增加最多一个 SMSS 的数值。

拥塞窗口cwnd每次的增加量 = min (N, SMSS)　　(5-8)

- 其中 *N* 是原先未被确认的、但现在被刚收到的确认报文段所确认的字节数。
- 不难看出，当 *N* < SMSS 时，拥塞窗口每次的增加量要小于 SMSS。
- 用这样的方法逐步增大发送方的拥塞窗口 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

- 使用慢开始算法后，每经过一个传输轮次 (transmission round)，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。

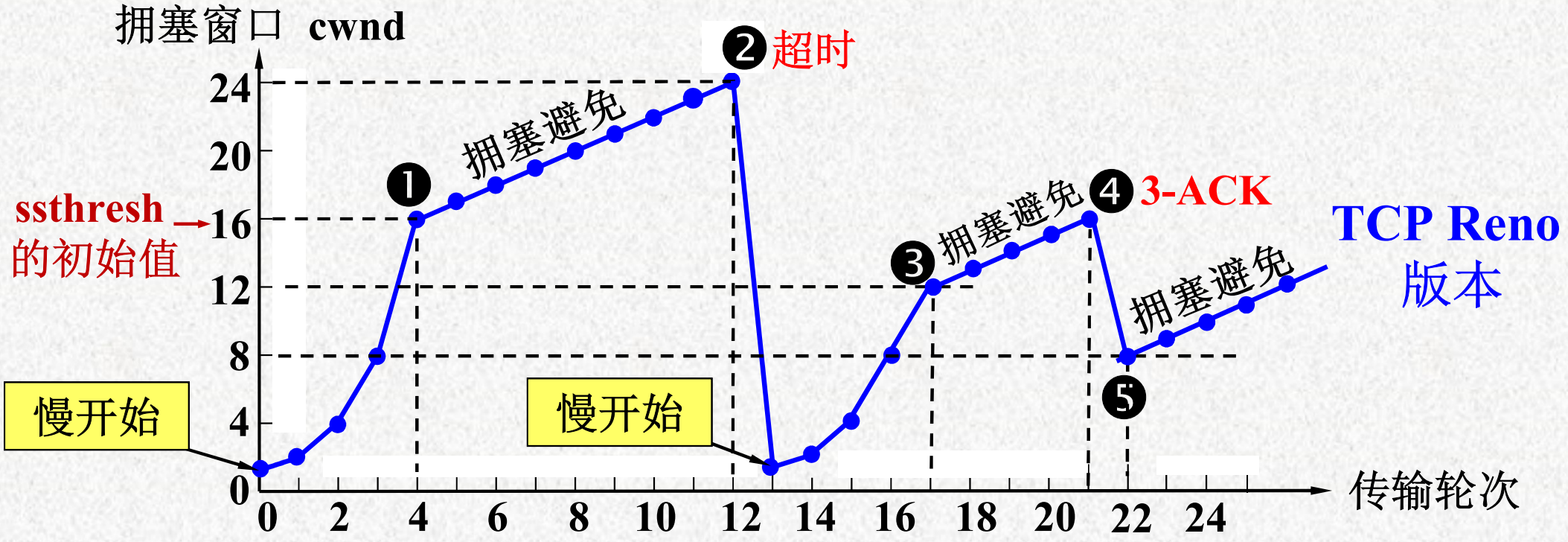
拥塞避免算法

- 思路：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。
- 因此在拥塞避免阶段就有“加法增大”(Additive Increase) 的特点。这表明在拥塞避免阶段，拥塞窗口 cwnd 按线性规律缓慢增长，比慢开始算法的拥塞窗口增长速率缓慢得多。

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（重传定时器超时）：
  - ssthresh = max(cwnd/2，2)
  - cwnd = 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。

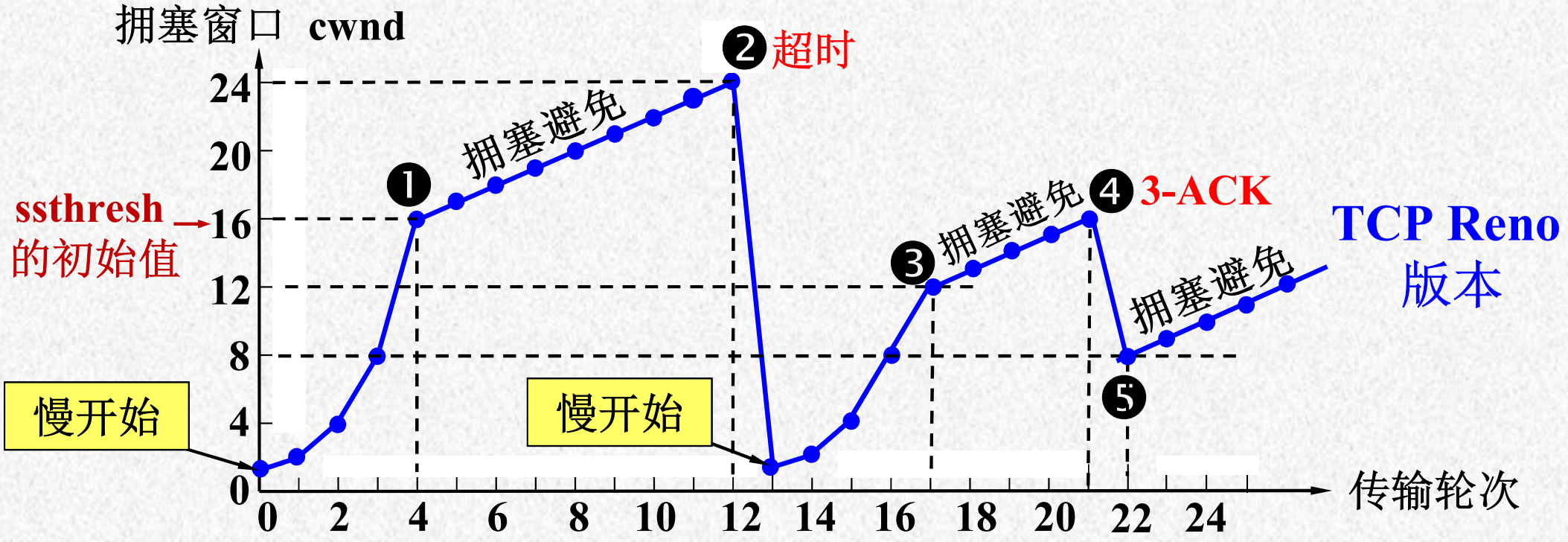
慢开始和拥塞避免算法的实现举例



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。

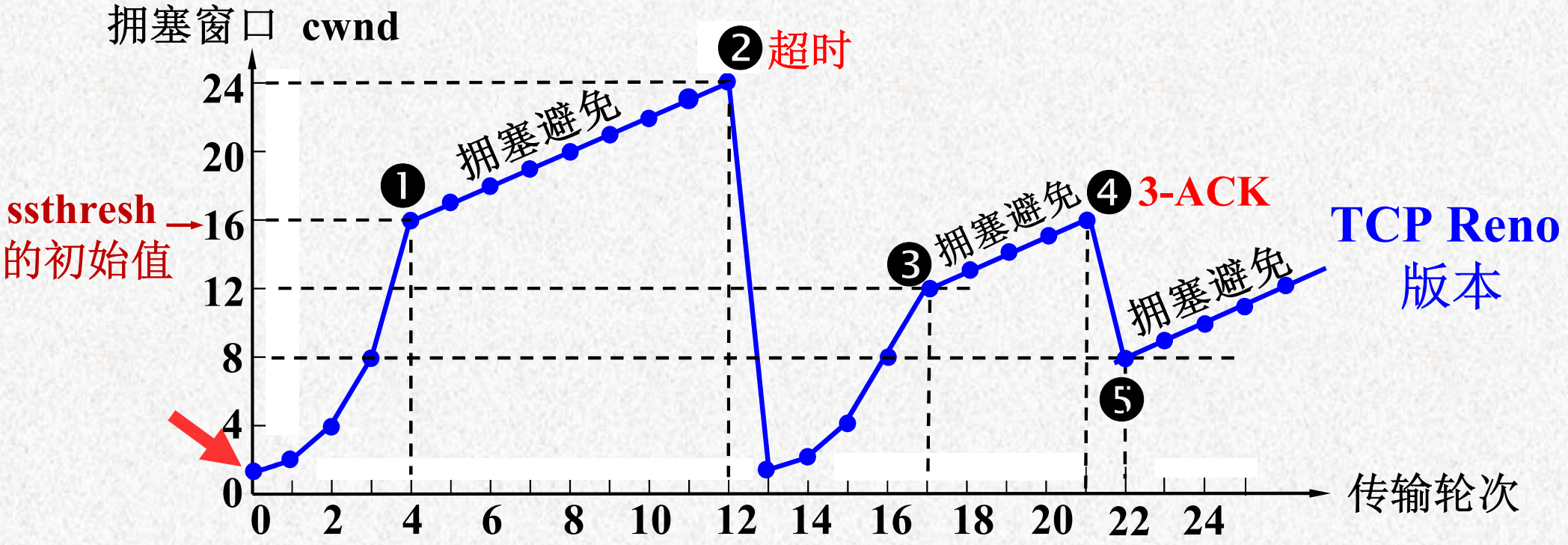
慢开始门限的初始值设置为 16 个报文段，即 ssthresh = 16。

慢开始和拥塞避免算法的实现举例



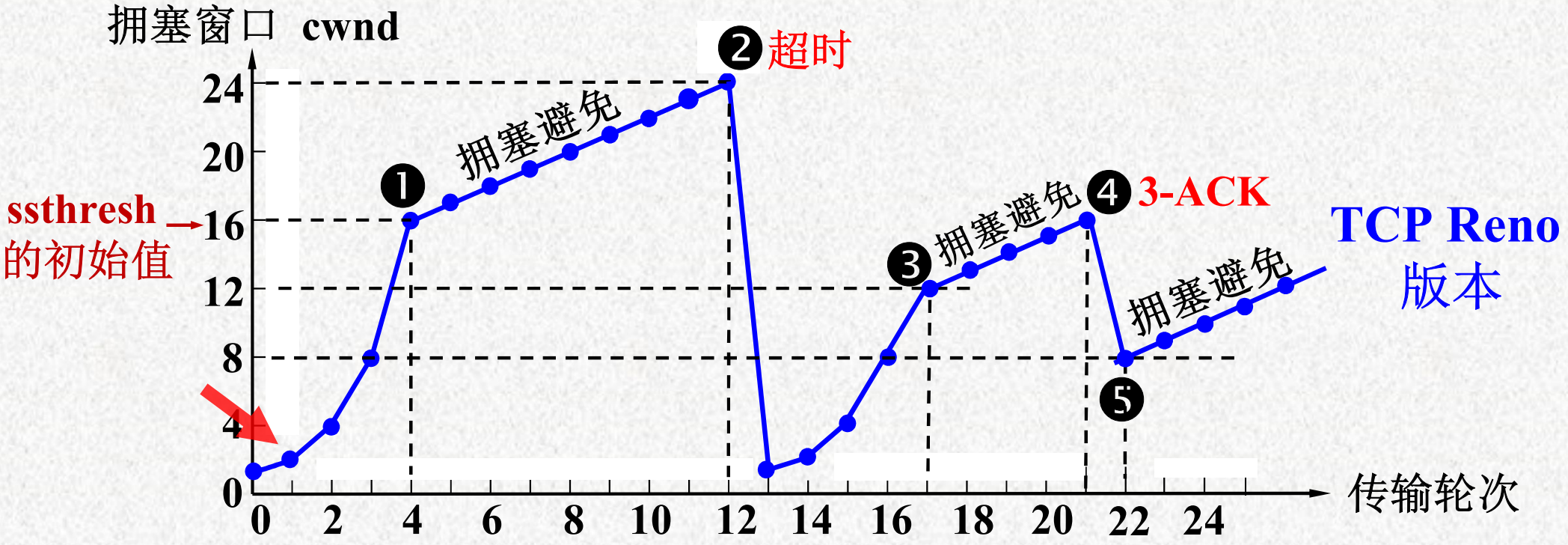
发送端的发送窗口不能超过拥塞窗口 cwnd 和接收端窗口 rwnd 中的最小值。我们假定接收端窗口足够大，因此现在发送窗口的数值等于拥塞窗口的数值。

慢开始和拥塞避免算法的实现举例



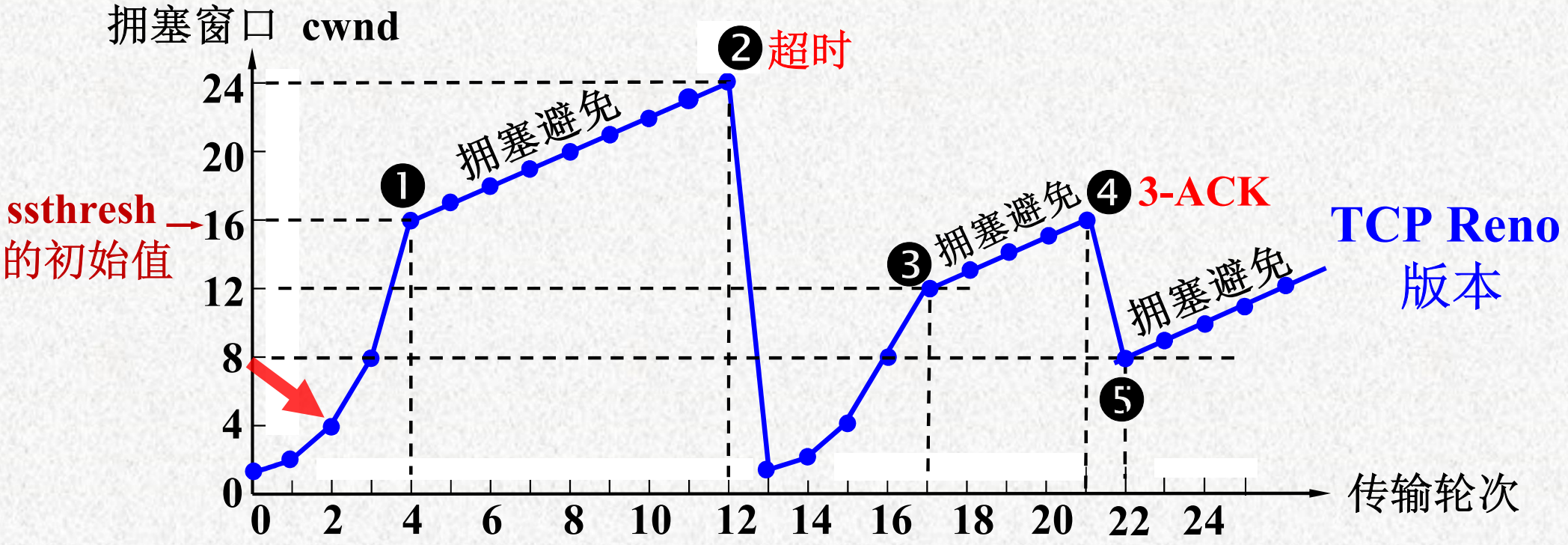
在执行慢开始算法时，拥塞窗口 cwnd=1，发送第一个报文段。

慢开始和拥塞避免算法的实现举例



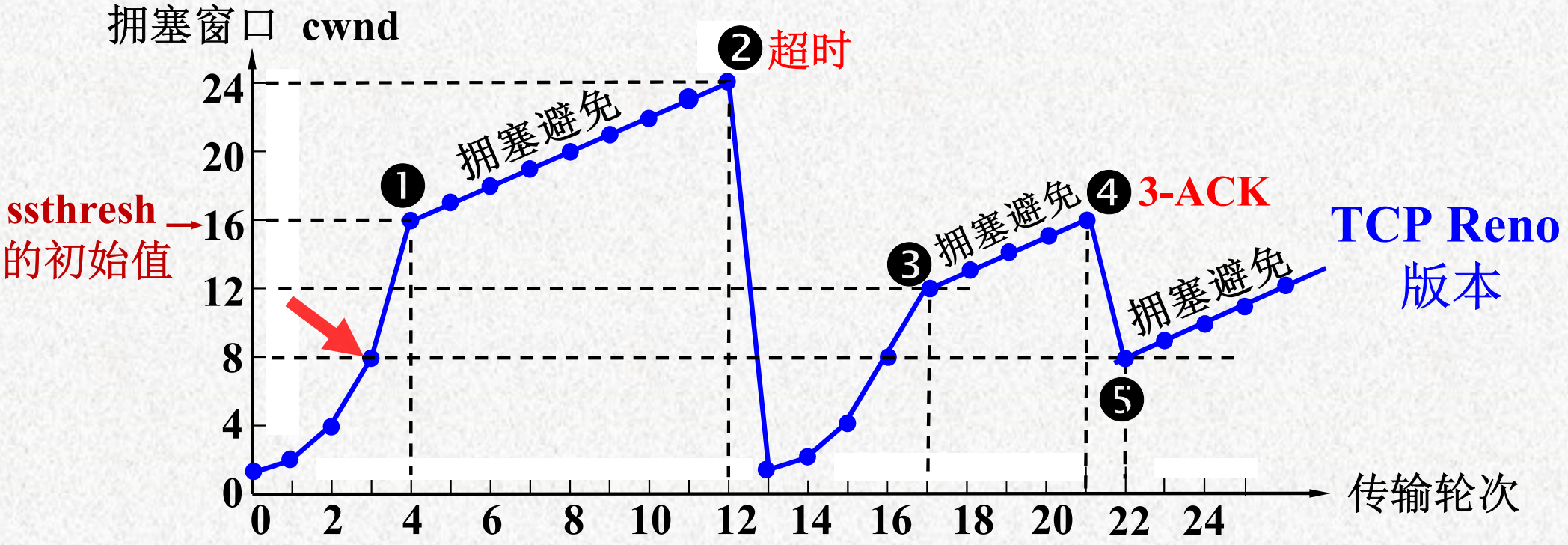
发送方每收到一个对新报文段的确认 ACK，就把拥塞窗口值加 1，然后开始下一轮的传输（请注意，横坐标是传输轮次，不是时间）。因此拥塞窗口 cwnd 随着传输轮次按指数规律增长。

慢开始和拥塞避免算法的实现举例



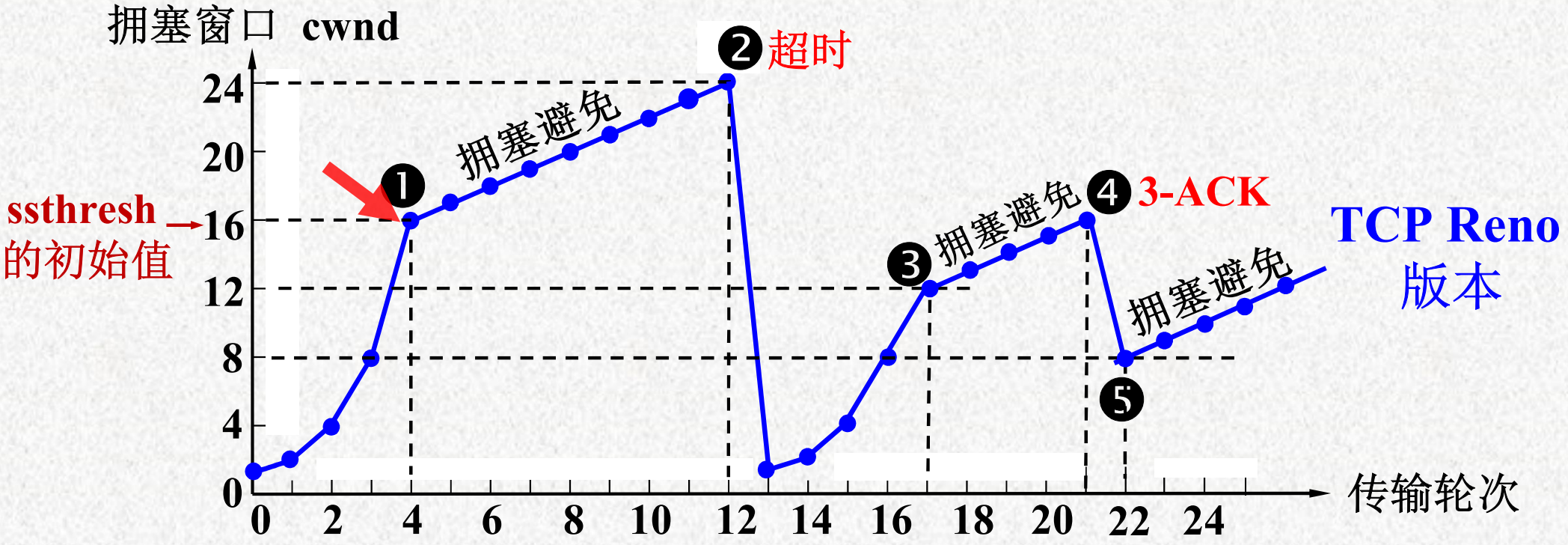
发送方每收到一个对新报文段的确认 ACK，就把拥塞窗口值加 1，然后开始下一轮的传输（请注意，横坐标是传输轮次，不是时间）。因此拥塞窗口 cwnd 随着传输轮次按指数规律增长。

慢开始和拥塞避免算法的实现举例



发送方每收到一个对新报文段的确认 ACK，就把拥塞窗口值加 1，然后开始下一轮的传输（请注意，横坐标是传输轮次，不是时间）。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

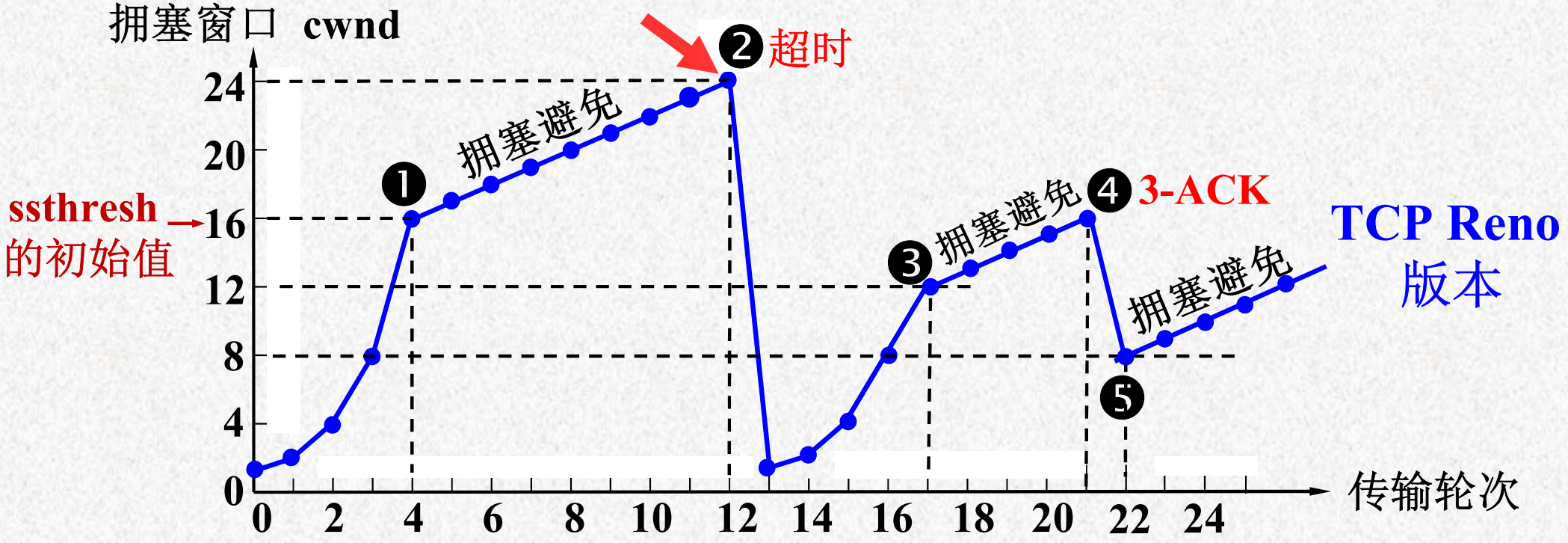


当拥塞窗口 cwnd 增长到慢开始门限值s sthresh 时（图中的点❶，此时拥塞窗口cwnd = 16），就改为执行拥塞避免算法，拥塞窗口按线性规律增长。

必须强调指出

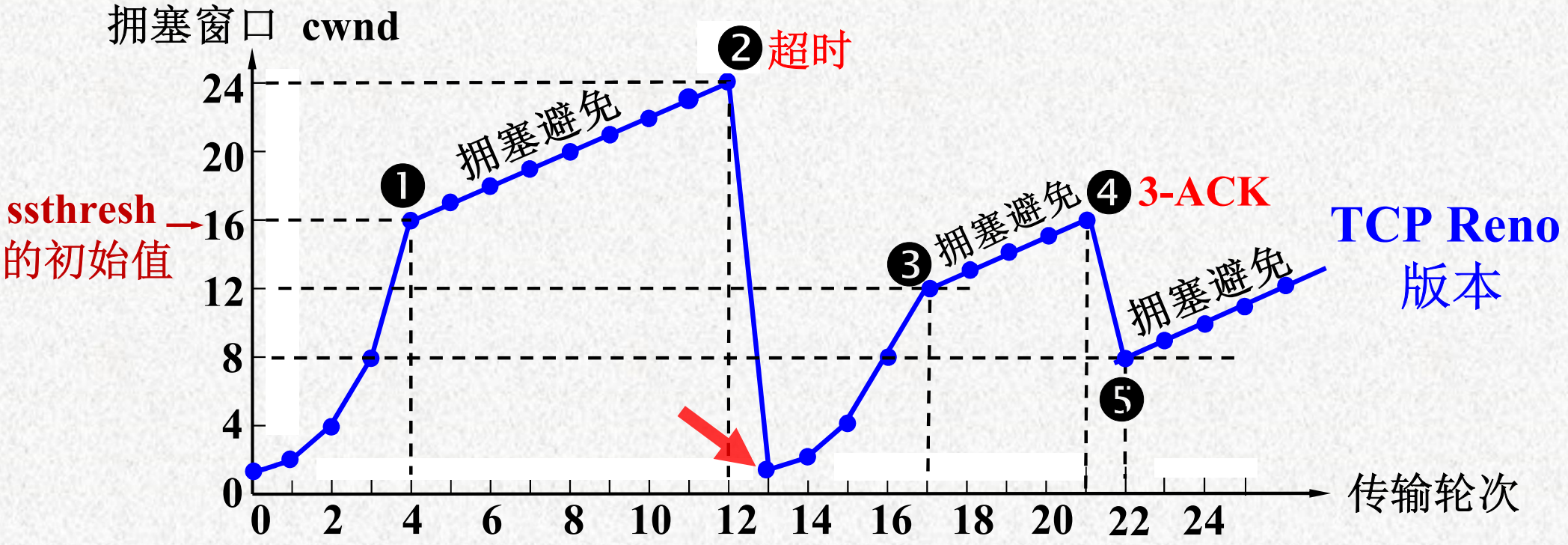
- “拥塞避免”并非指完全能够避免了拥塞。利用以上的措施要完全避免网络拥塞还是不可能的。
- “拥塞避免”是说在拥塞避免阶段把拥塞窗口控制为按线性规律增长，使网络比较不容易出现拥塞。

慢开始和拥塞避免算法的实现举例



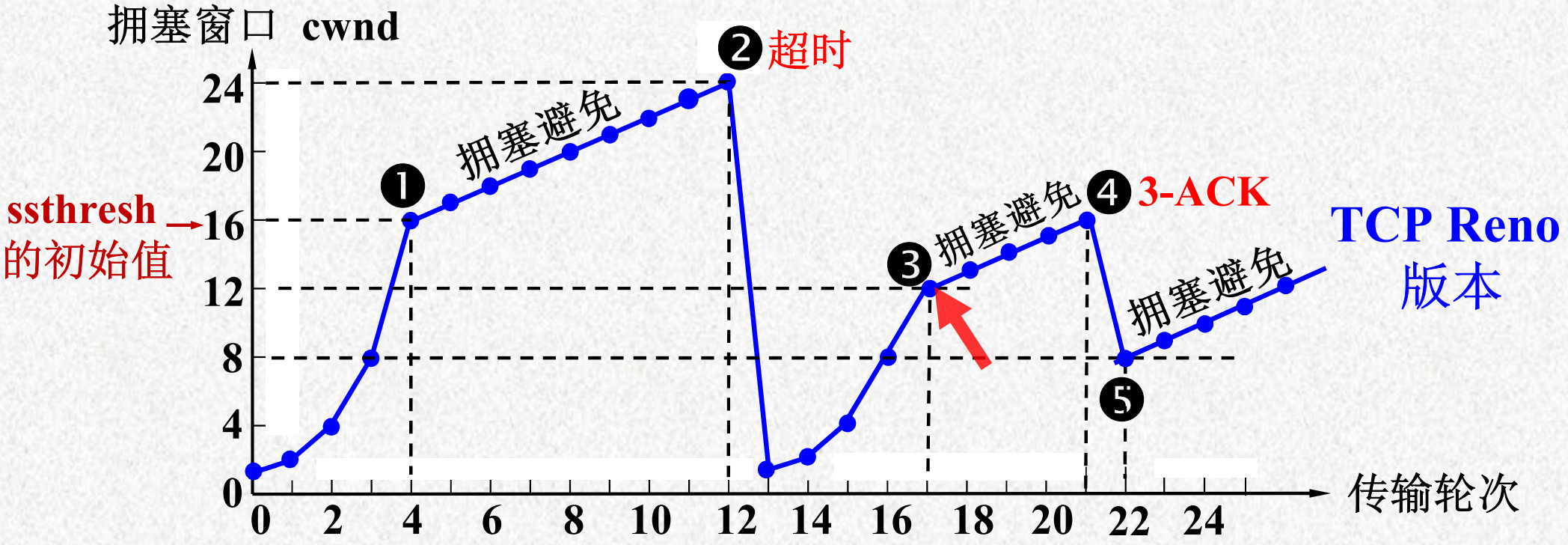
当拥塞窗口 cwnd = 24 时，网络出现了超时（图中的点❷），发送方判断为网络拥塞。于是调整门限值 ssthresh = cwnd / 2 = 12，同时设置拥塞窗口 cwnd = 1，进入慢开始阶段。

慢开始和拥塞避免算法的实现举例



当拥塞窗口 cwnd = 24 时，网络出现了超时（图中的点❷），发送方判断为网络拥塞。于是调整门限值 ssthresh = cwnd / 2 = 12，同时设置拥塞窗口 cwnd = 1，进入慢开始阶段。

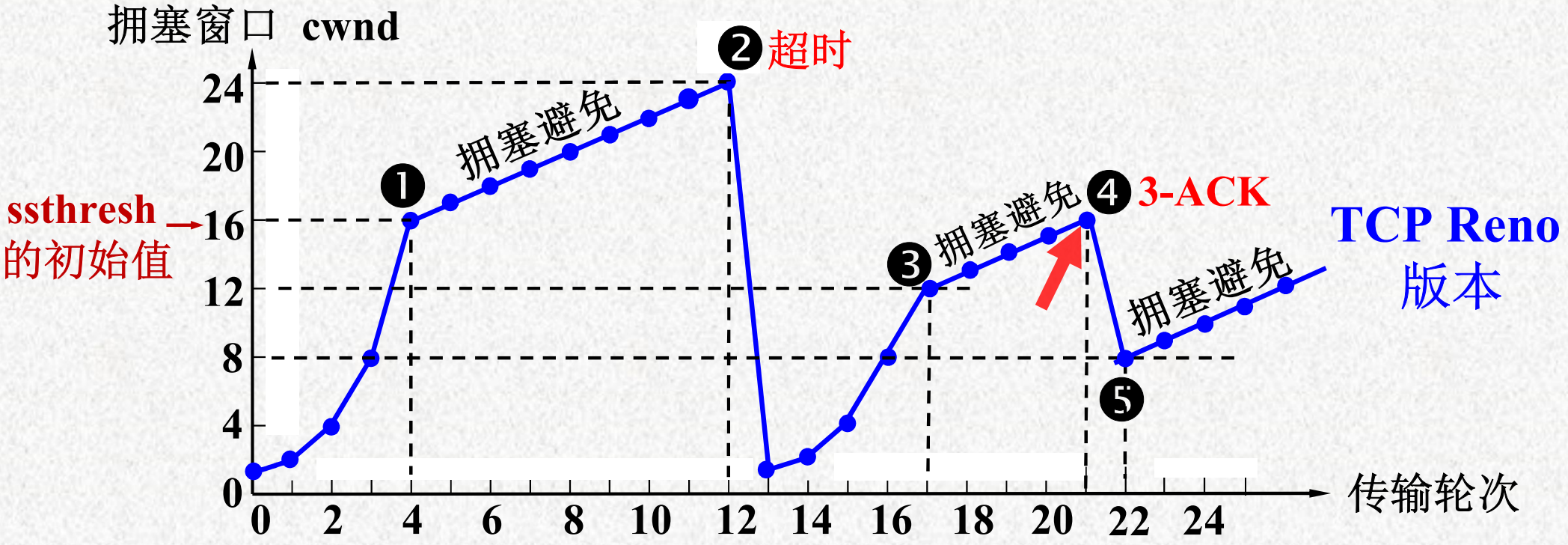
慢开始和拥塞避免算法的实现举例



按照慢开始算法，发送方每收到一个对新报文段的确认ACK，就把拥塞窗口值加1。

当拥塞窗口cwnd = ssthresh = 12时（图中的点❸，这是新的ssthresh值），改为执行拥塞避免算法，拥塞窗口按线性规律增大。

慢开始和拥塞避免算法的实现举例



当拥塞窗口cwnd = 16时（图中的点❹），出现了一个新的情况，就是发送方一连收到 3 个对同一个报文段的重复确认（图中记为3-ACK）。发送方改为执行快重传和快恢复算法。

快重传算法

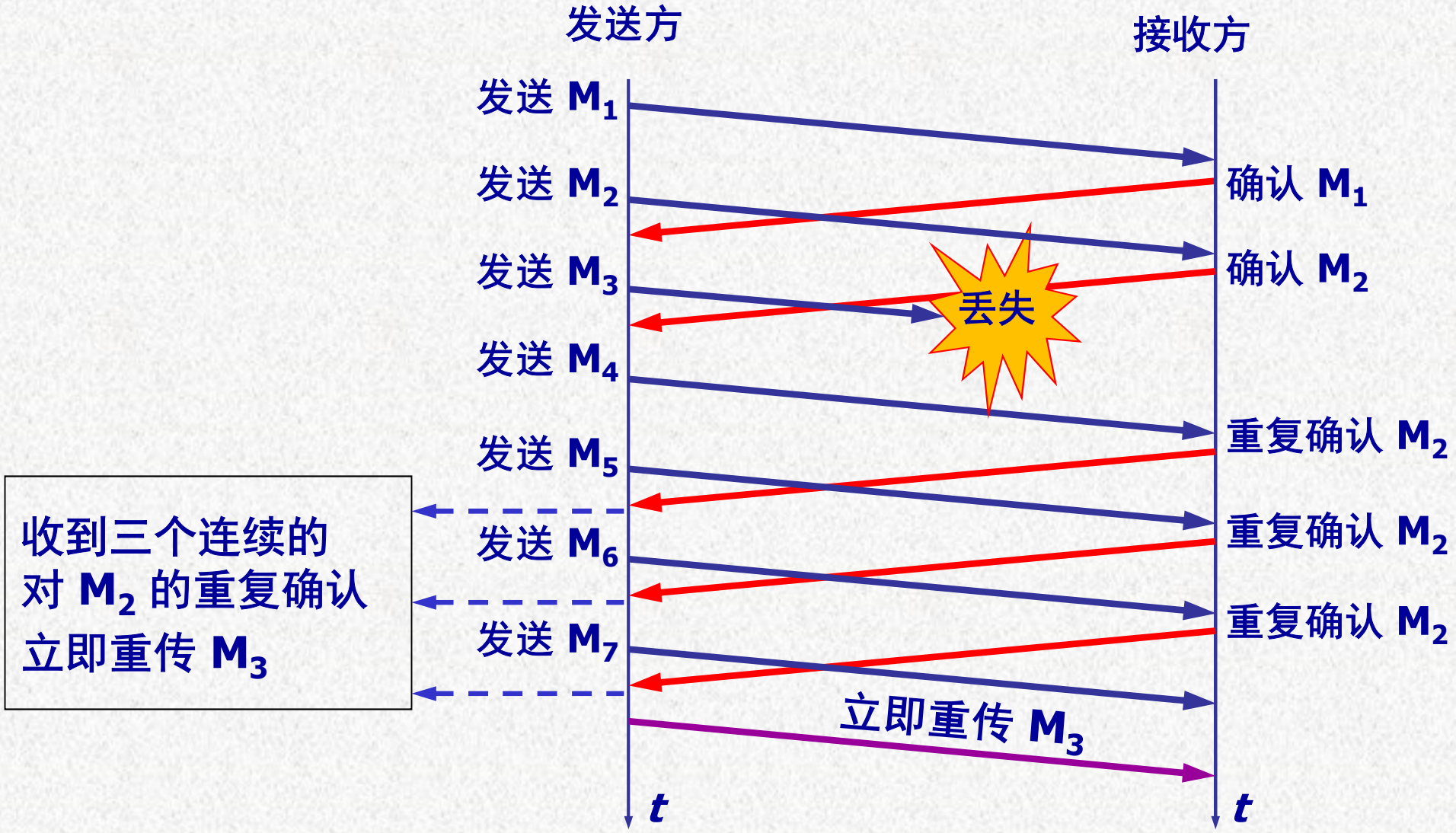
- 采用快重传FR (Fast Retransmission) 算法可以让发送方尽早知道发生了个别报文段的丢失。
- 快重传 算法首先要求接收方不要等待自己发送数据时才进行捎带确认，而是要立即发送确认，即使收到了失序的报文段也要立即发出对已收到的报文段的重复确认。

快重传算法

- 发送方只要一连收到三个重复确认，就知道接收方确实没有收到报文段，因而应当立即进行重传（即“快重传”），这样就不会出现超时，发送方也不就会误认为出现了网络拥塞。
- 使用快重传可以使整个网络的吞吐量提高约20%。

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
t
t
收到三个连续的
对 M_2 的重复确认
立即重传 M_3

快恢复算法

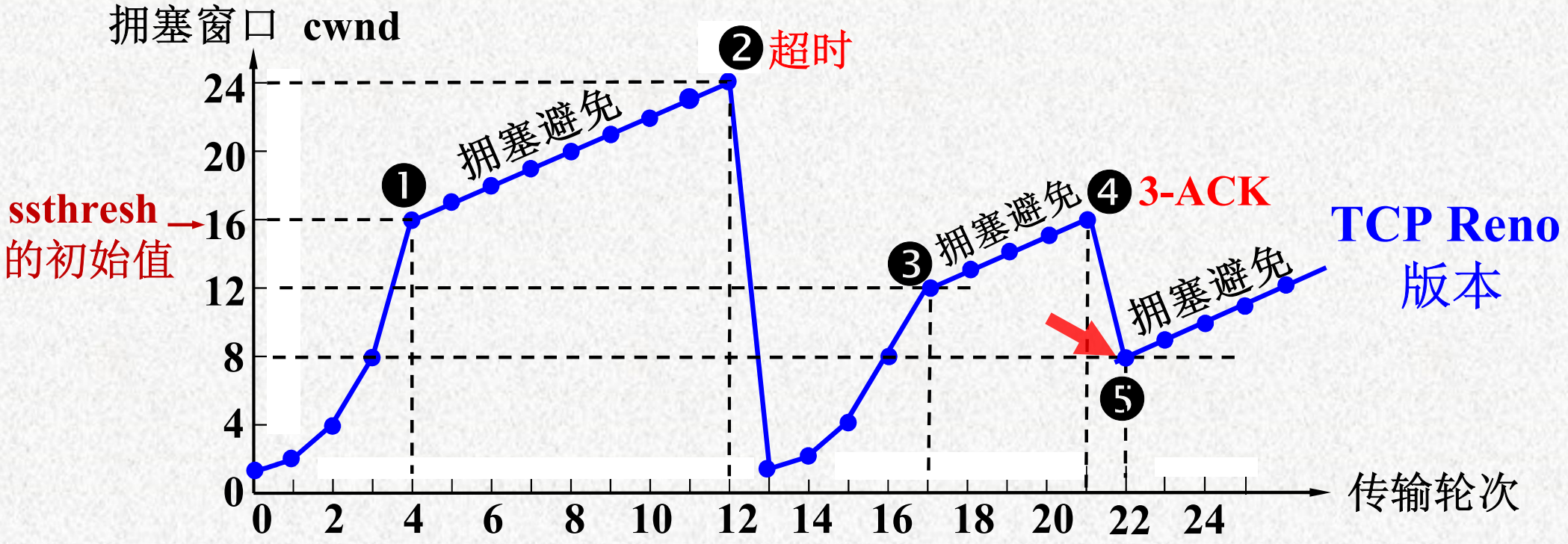
- 当发送端收到连续三个重复的确认时，由于发送方现在认为网络很可能没有发生拥塞，因此现在不执行慢开始算法，而是执行快恢复算法 **FR (Fast Recovery)** 算法：

  (1) 慢开始门限 ssthresh = 当前拥塞窗口 cwnd / 2 ；

  (2) 新拥塞窗口 cwnd = 慢开始门限 ssthresh ；

  (3) 开始执行拥塞避免算法，使拥塞窗口缓慢地线性增大。

慢开始和拥塞避免算法的实现举例

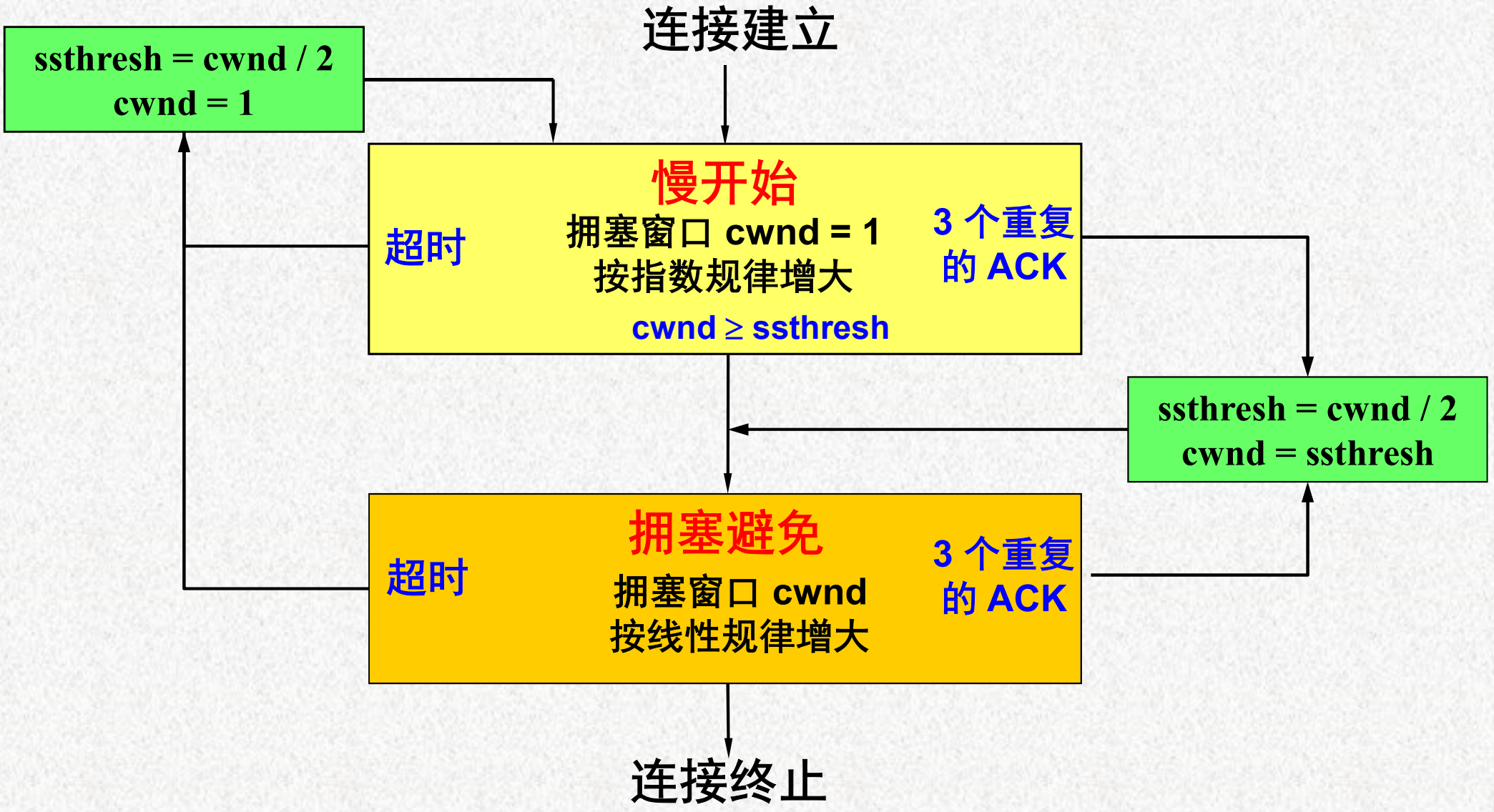


因此，在图的点❹，发送方知道现在只是丢失了个别的报文段。于是不启动慢开始，而是执行快恢复算法。这时，发送方调整门限值ssthresh = cwnd / 2 = 8，同时设置拥塞窗口cwnd = ssthresh = 8（见图中的点❺），并开始执行拥塞避免算法。

加法增大，乘法减小 (AIMD)

- 可以看出，在拥塞避免阶段，拥塞窗口是按照线性规律增大的。这常称为“加法增大” AI (Additive Increase)。
- 当出现超时或3个重复的确认时，就要把门限值设置为当前拥塞窗口值的一半，并大大减小拥塞窗口的数值。这常称为“乘法减小”MD (Multiplicative Decrease)。
- 二者合在一起就是所谓的 AIMD 算法。

TCP拥塞控制流程图


连接建立
ssthresh = cwnd / 2
cwnd = 1
慢开始
拥塞窗口 cwnd = 1
按指数规律增大
cwnd ≥ ssthresh
超时
3 个重复
的 ACK
ssthresh = cwnd / 2
cwnd = ssthresh
拥塞避免
拥塞窗口 cwnd
按线性规律增大
超时
3 个重复
的 ACK
连接终止

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]** **(5-9)**

- 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。

也就是说，**rwnd** 和 **cwnd** 中数值较小的一个，控制了发送方发送数据的速率。

5.8.3 主动队列管理 AQM

- TCP 拥塞控制和网络层采取的策略有密切联系。
- 重传会使 TCP 连接的发送端认为在网络中发生了拥塞。于是在 TCP 的发送端就采取了拥塞控制措施，但实际上网络并没有发生拥塞。
- 网络层的策略对 TCP 拥塞控制影响最大的就是路由器的分组丢弃策略。

“先进先出”FIFO 处理规则

- 路由器的队列通常都是按照“先进先出”FIFO (First In First Out) 的规则处理到来的分组。
- 当队列已满时，以后再到达的所有分组（如果能够继续排队，这些分组都将排在队列的尾部）将都被丢弃。这就叫做尾部丢弃策略 (tail-drop policy)。
- 路由器的尾部丢弃往往会导致一连串分组的丢失，这就使发送方出现超时重传，使 TCP 进入拥塞控制的慢开始状态，结果使 TCP 连接的发送方突然把数据的发送速率降低到很小的数值。

全局同步

- 更为严重的是，在网络中通常有很多的 TCP 连接，这些连接中的报文段通常是复用在网络层的 IP 数据报中传送的。
- 在这种情况下，若发生了路由器中的尾部丢弃，就可能会同时影响到很多条 TCP 连接，结果使这许多 TCP 连接在同一时间突然都进入到慢开始状态。这在 TCP 的术语中称为全局同步 (global syncronization)。
- 全局同步使得全网的通信量突然下降了很多，而在网络恢复正常后，其通信量又突然增大很多。

主动队列管理AQM

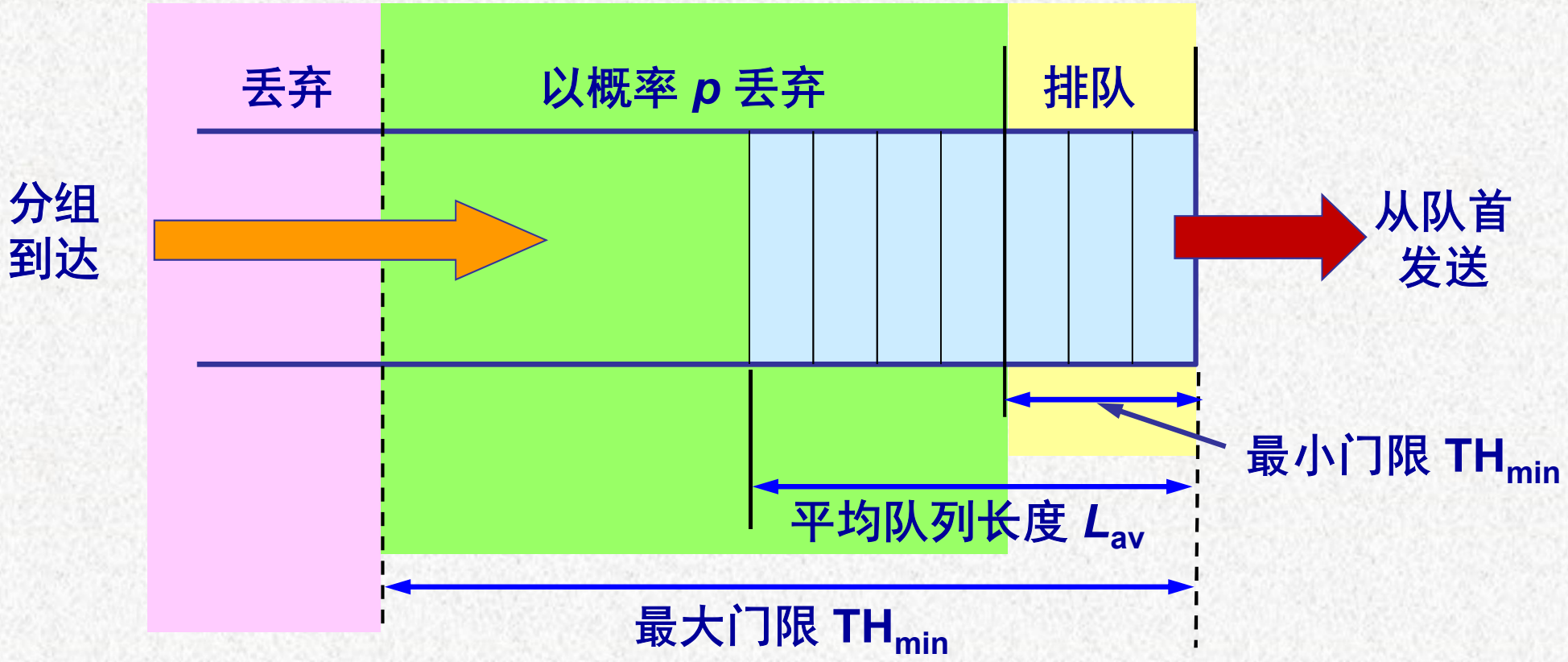
- 1998年提出了主动队列管理 AQM (Active Queue Management)。
- 所谓“主动”就是不要等到路由器的队列长度已经达到最大值时才不得不丢弃后面到达的分组。这样就太被动了。应当在队列长度达到某个值得警惕的数值时（即当网络拥塞有了某些拥塞征兆时），就主动丢弃到达的分组。
- AQM 可以有不同实现方法，其中曾流行多年的就是随机早期检测 RED (Random Early Detection)。

随机早期检测 RED

- 使路由器的队列维持两个参数：队列长度最小门限 TH_{min} 和最大门限 Th_{max} 。
- RED 对每一个到达的分组都先计算平均队列长度 L_{AV} 。
- (1) 若平均队列长度小于最小门限 TH_{min}，则将新到达的分组放入队列进行排队。
- (2) 若平均队列长度超过最大门限 TH_{max}，则将新到达的分组丢弃。
- (3) 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 p 将新到达的分组丢弃。

随机早期检测 RED

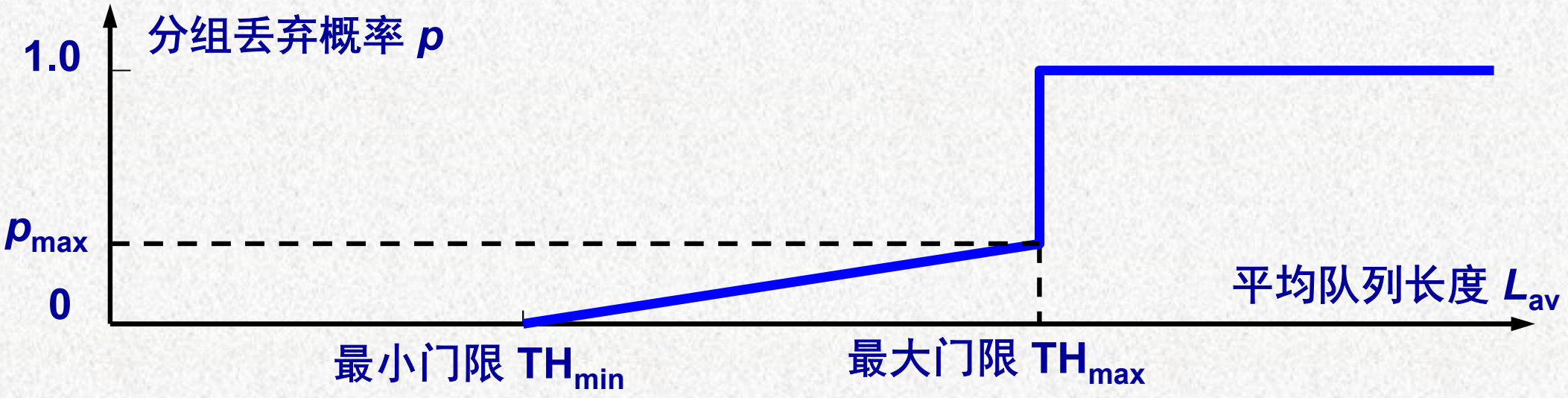
RED 将路由器的到达队列划分成为三个区域：



丢弃概率 p 与 TH_{min} 和 Th_{max} 的关系

- 当 $L_{AV} < Th_{min}$ 时，丢弃概率 $p = 0$。
- 当 $L_{AV} > Th_{max}$ 时，丢弃概率 $p = 1$。
- 当 $TH_{min} < L_{AV} < TH_{max}$ 时，$0 < p < 1$ 。

在 RED 的操作中，最难处理的就是丢弃概率 p 的选择，因为 p 并不是个常数。例如，按线性规律变化，从 0 变到 p_{max}。



随机早期检测 RED

- 多年的实践证明，RED 的使用效果并不太理想。
- 2015年公布的 RFC 7567 已经把 RFC 2309列为陈旧的，并且不再推荐使用 RED。
- 对路由器进行主动队列管理 AQM 仍是必要的。
- AQM 实际上就是对路由器中的分组排队进行智能管理，而不是简单地把队列的尾部丢弃。
- 现在已经有几种不同的算法来代替旧的 RED，但都还在实验阶段。

5.9 TCP 的运输连接管理

- 5.9.1 TCP 的连接建立
- 5.9.2 TCP 的连接释放
- 5.9.3 TCP 的有限状态机

运输连接的三个阶段

- TCP 是面向连接的协议。
- 运输连接有三个阶段：
  - 连接建立
  - 数据传送
  - 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。

TCP 连接建立过程中要解决的三个问题

- (1) 要使每一方能够确知对方的存在。
- (2) 要允许双方协商一些参数（如最大窗口值、是否使用窗口扩大选项和时间戳选项以及服务质量等）。
- (3) 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。

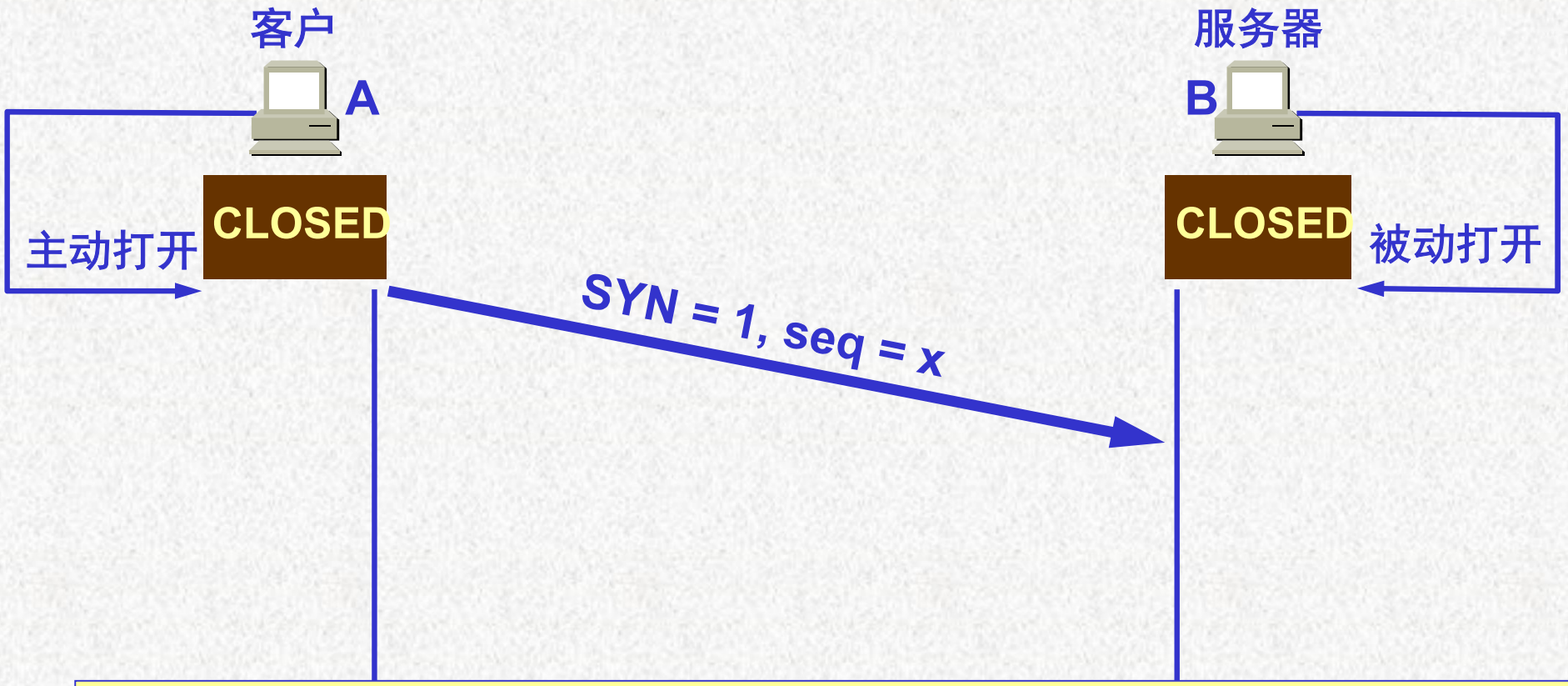
客户–服务器方式

- 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

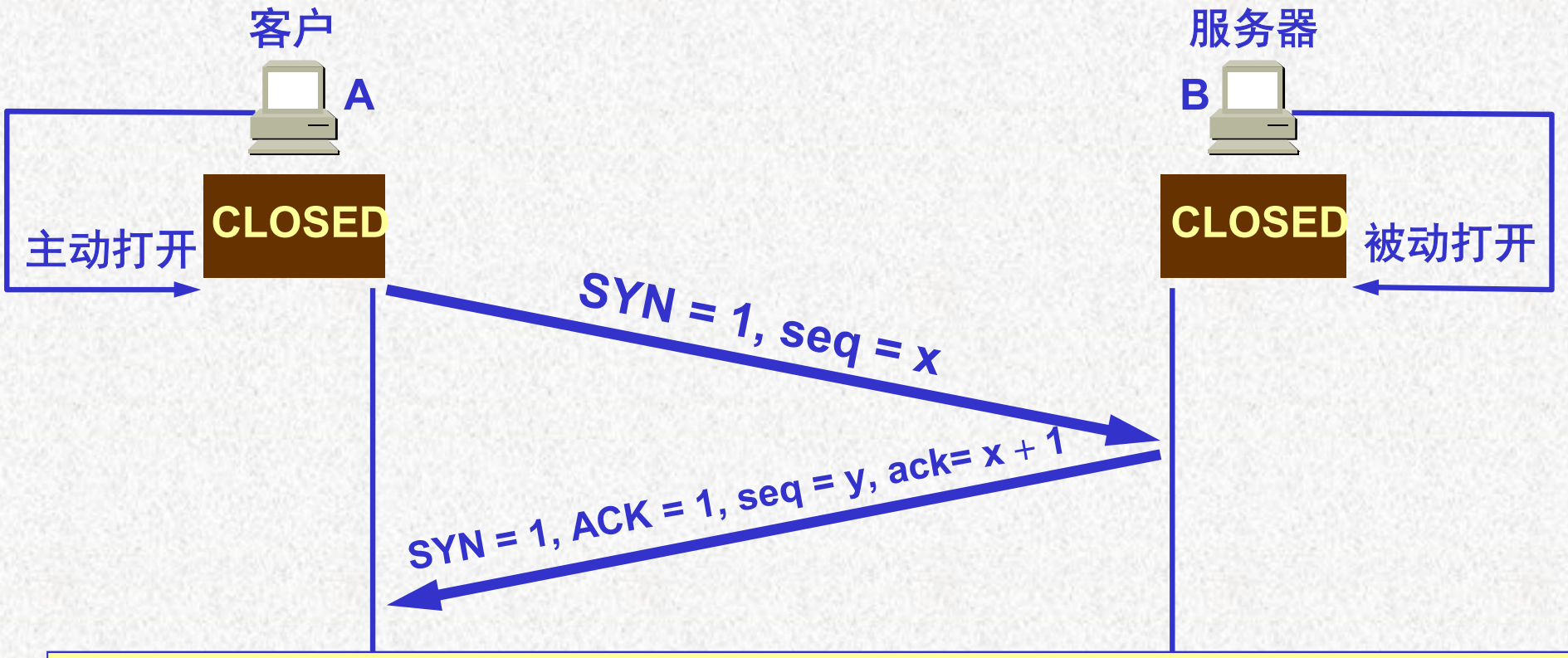
- TCP 建立连接的过程叫做握手。
- 握手需要在客户和服务器之间交换三个 TCP 报文段。称之为三报文握手。
- 采用三报文握手主要是为了防止已失效的连接请求报文段突然又传送到了，因而产生错误。

TCP 的连接建立：采用三报文握手



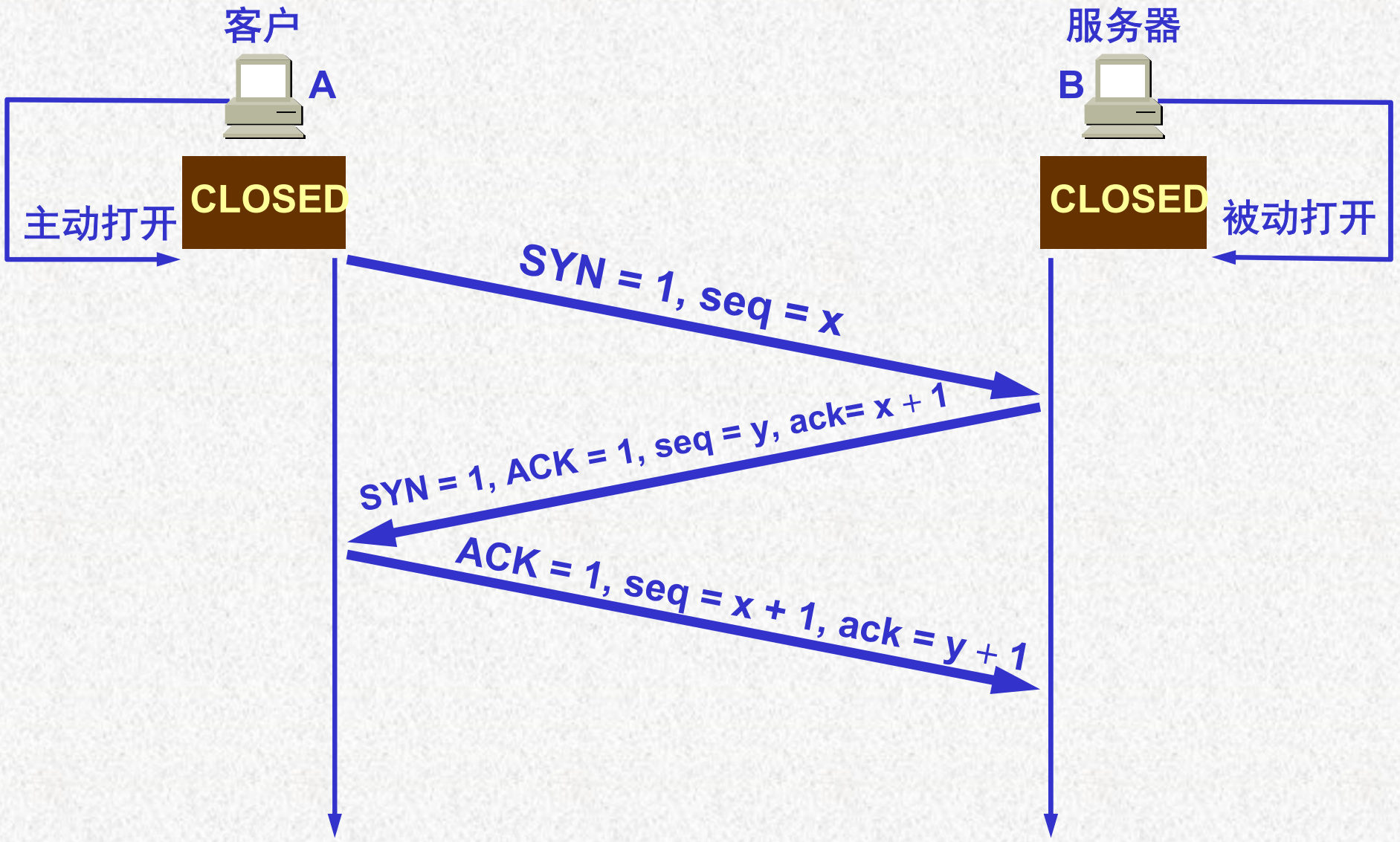
A 的 TCP 向 B 发出连接请求报文段，其首部中的同步位 SYN = 1，并选择序号 seq = x，表明传送数据时的第一个数据字节的序号是 x。

TCP 的连接建立：采用三报文握手

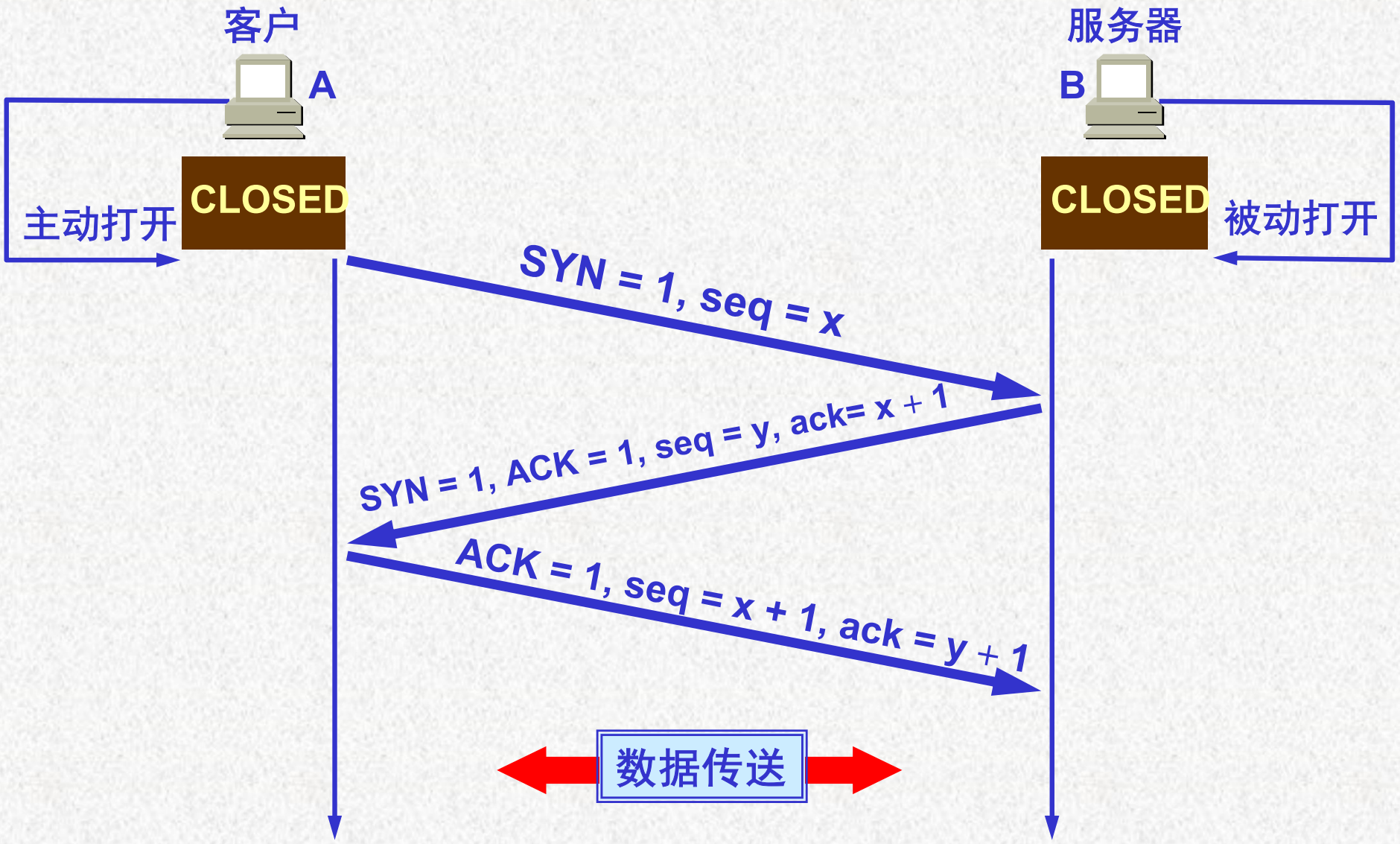


- B 的 TCP 收到连接请求报文段后，如同意，则发回确认。
- B 在确认报文段中应使 SYN = 1，使 ACK = 1，其确认号ack = x + 1，自己选择的序号 seq = y。

- 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。

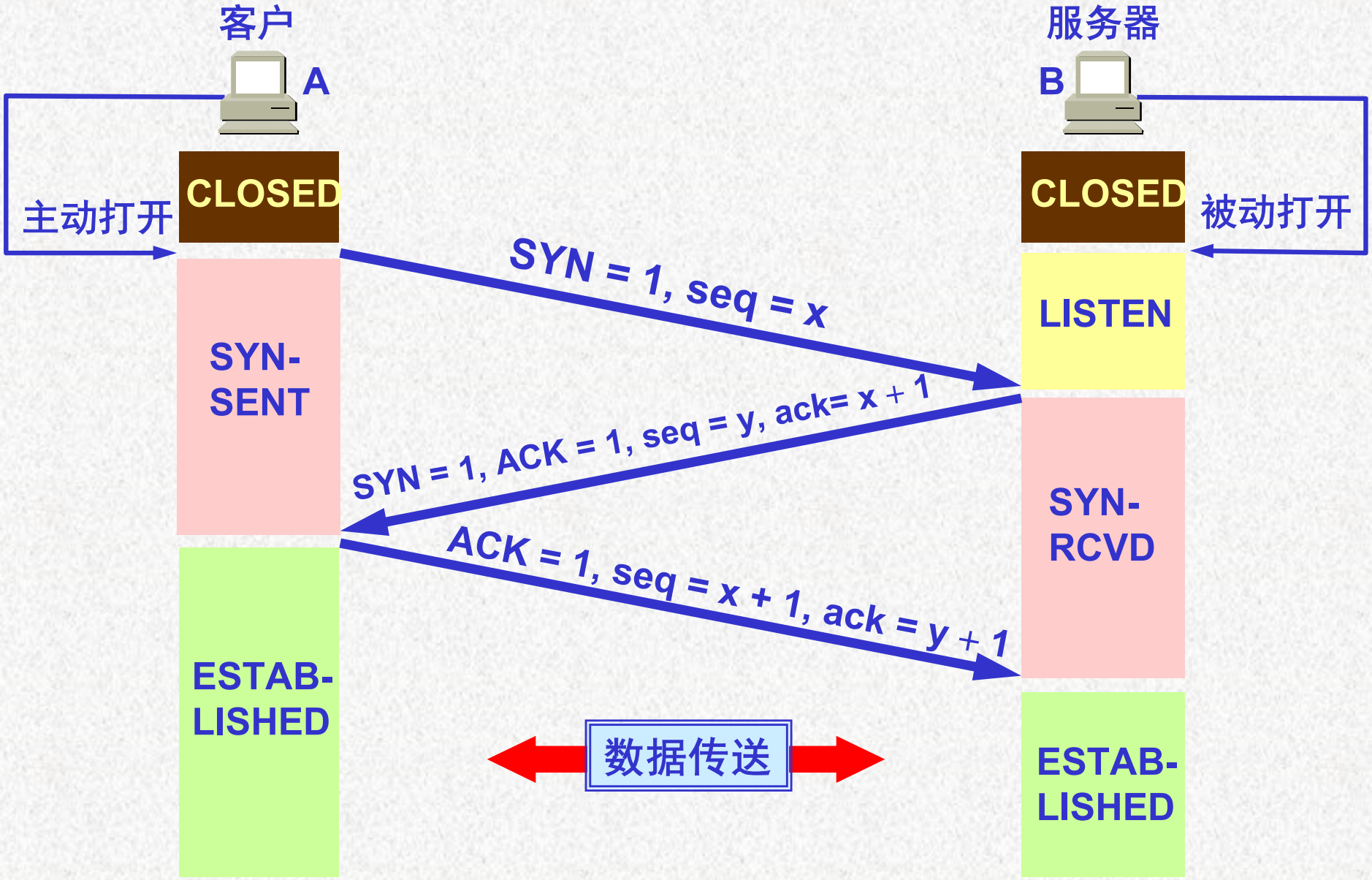


- B 的 TCP 收到主机 A 的确认后，也通知其上层应用进程：TCP 连接已经建立。



TCP 的连接建立：采用三报文握手

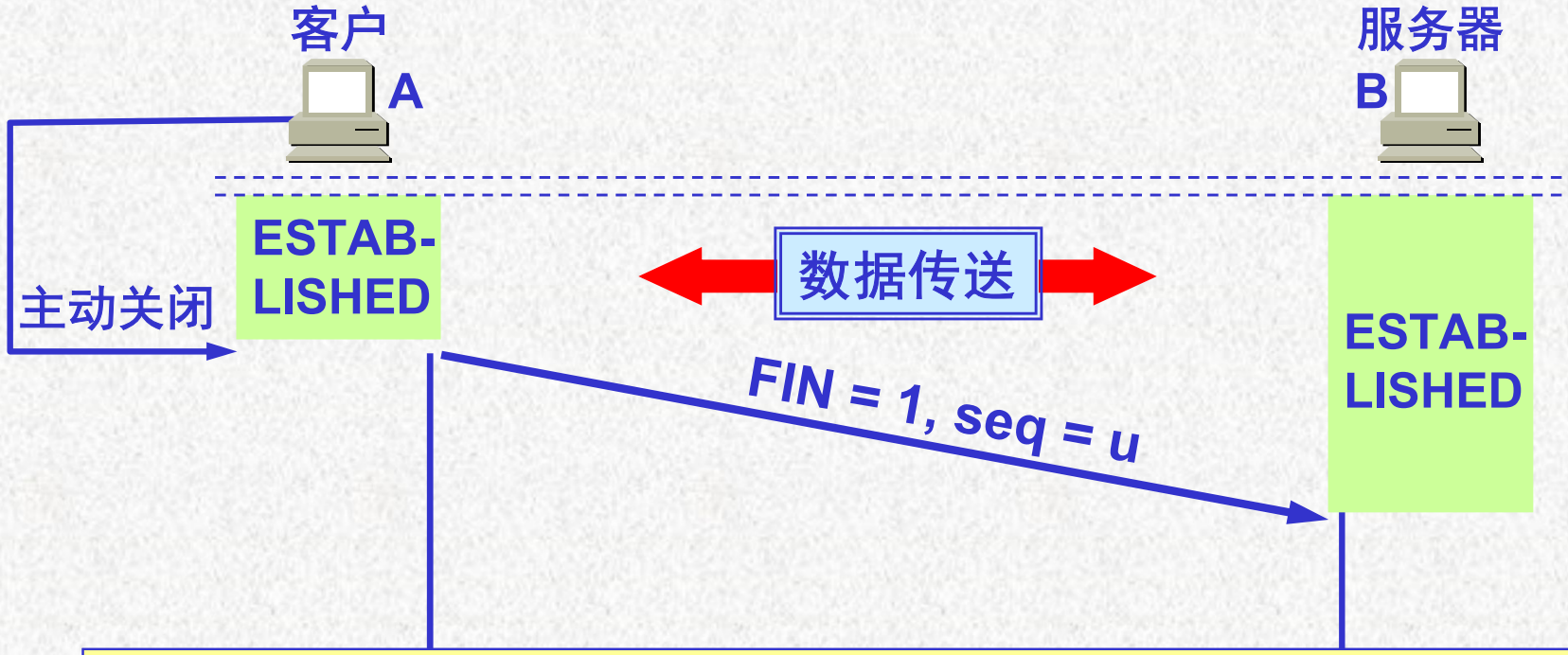
采用三报文握手建立 TCP 连接的各状态



5.9.2 TCP 的连接释放

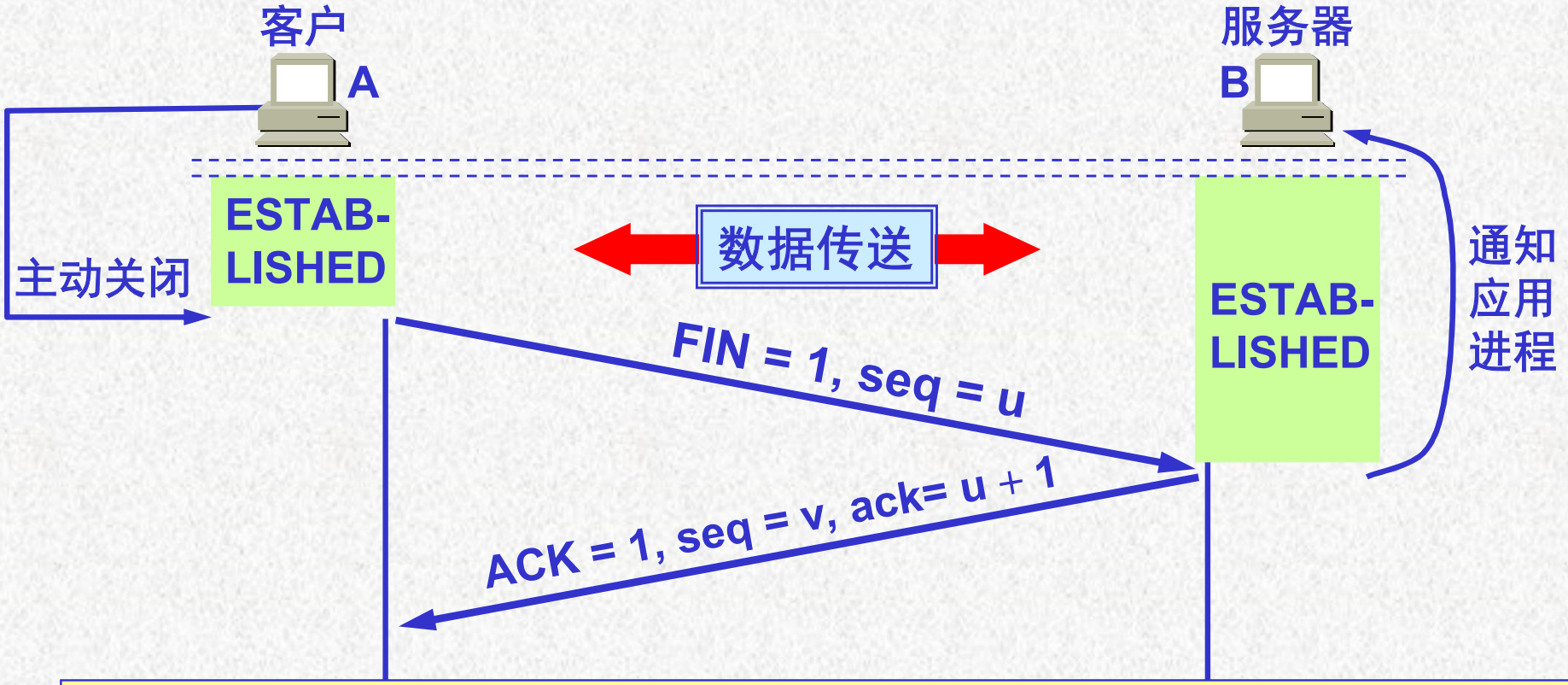
- TCP 连接释放过程比较复杂。
- 数据传输结束后，通信的双方都可释放连接。
- TCP 连接释放过程是四报文握手。

TCP 的连接释放：采用四报文握手



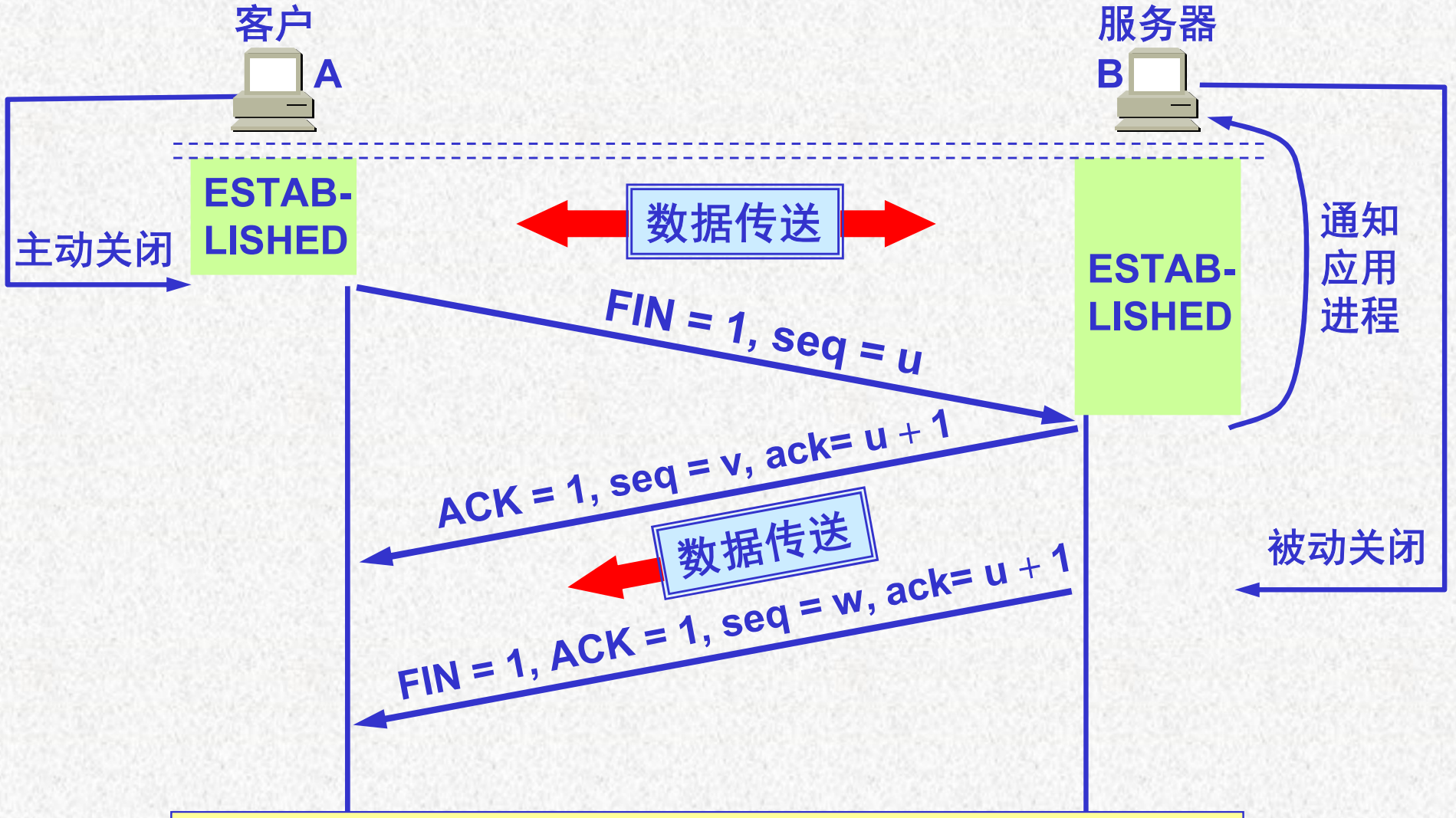
- 数据传输结束后，通信的双方都可释放连接。
- 现在 A 的应用进程先向其 TCP 发出连接释放报文段，并停止再发送数据，主动关闭 TCP 连接。
- A 把连接释放报文段首部的 FIN = 1，其序号 seq = u，等待 B 的确认。

TCP 的连接释放：采用四报文握手



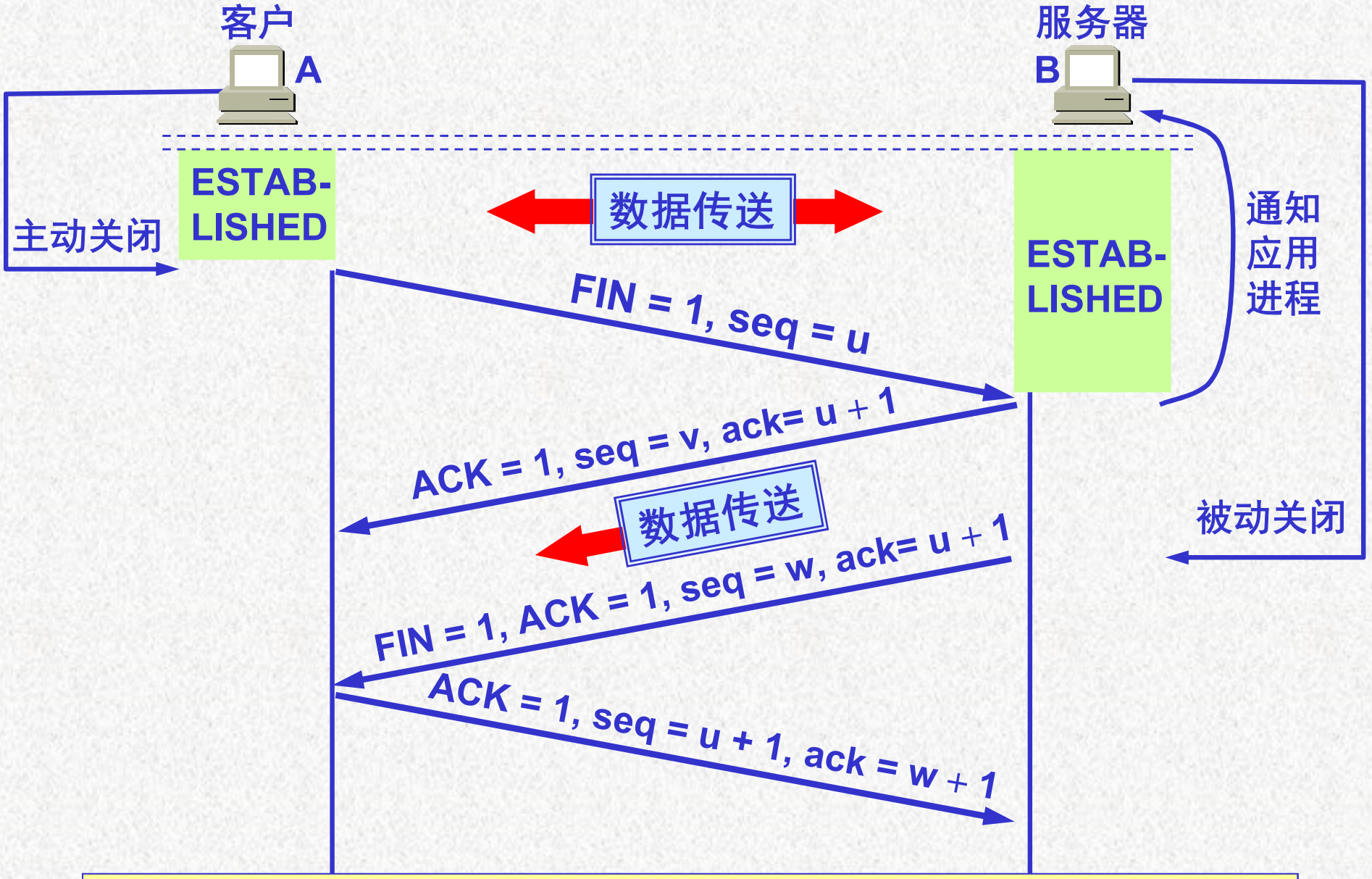
- B 发出确认，确认号 ack = u + 1，而这个报文段自己的序号 seq = v。
- TCP 服务器进程通知高层应用进程。
- 从 A 到 B 这个方向的连接就释放了，TCP 连接处于半关闭状态。B 若发送数据，A 仍要接收。

TCP 的连接释放：采用四报文握手



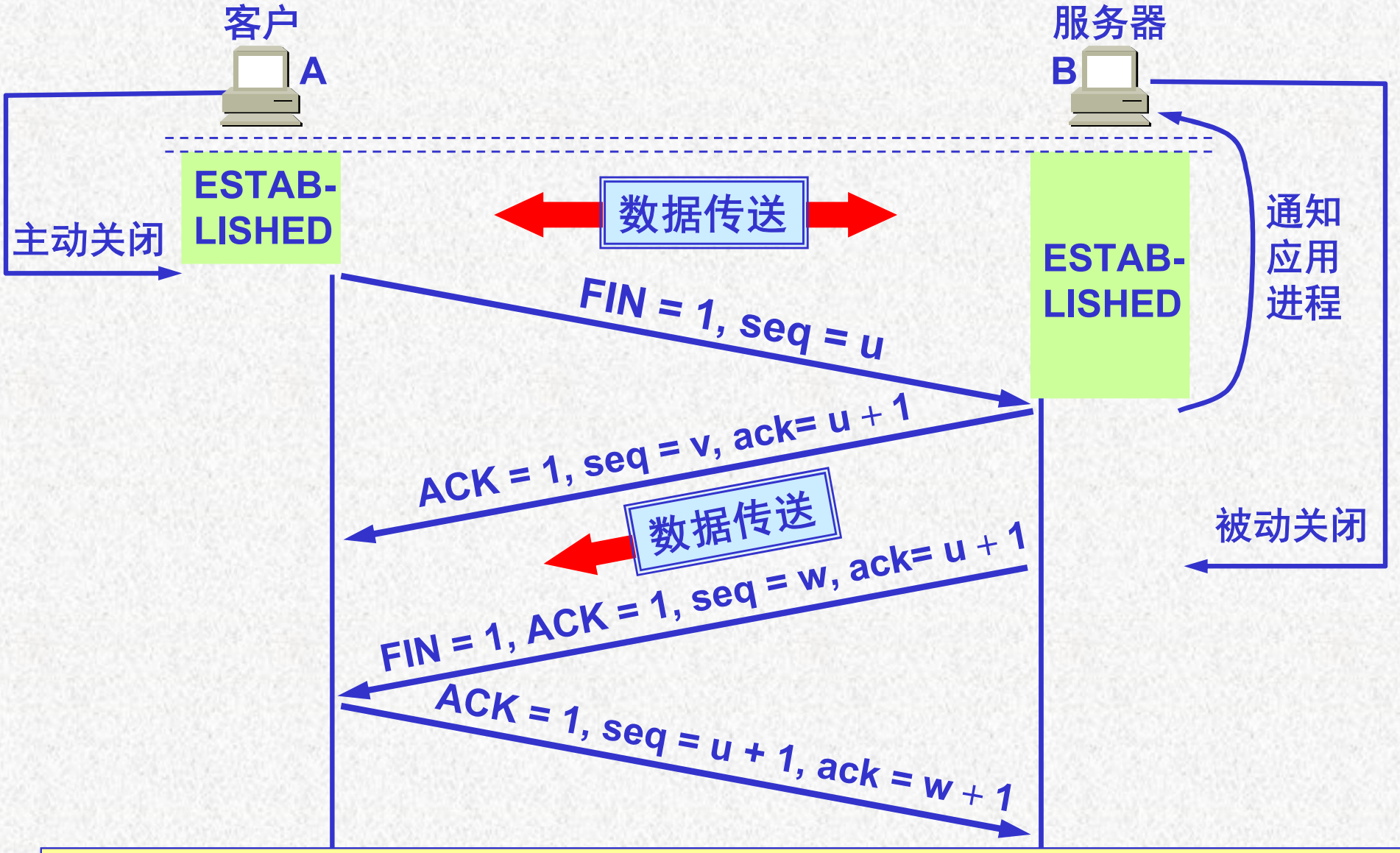
- 若 B 已经没有要向 A 发送的数据，其应用进程就通知 TCP 释放连接。

TCP 的连接释放：采用四报文握手



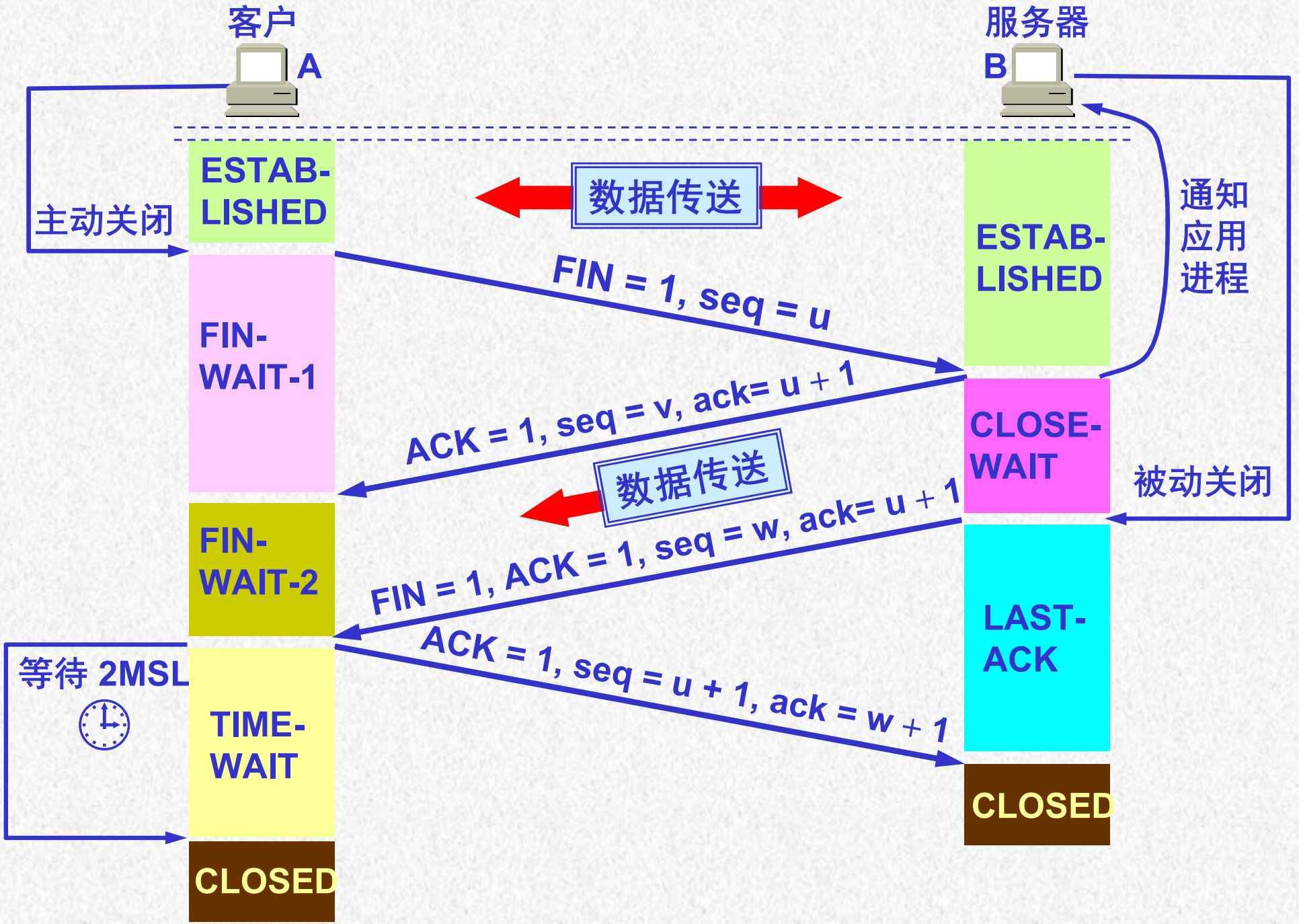
- A 收到连接释放报文段后，必须发出确认。

TCP 的连接释放：采用四报文握手



- 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

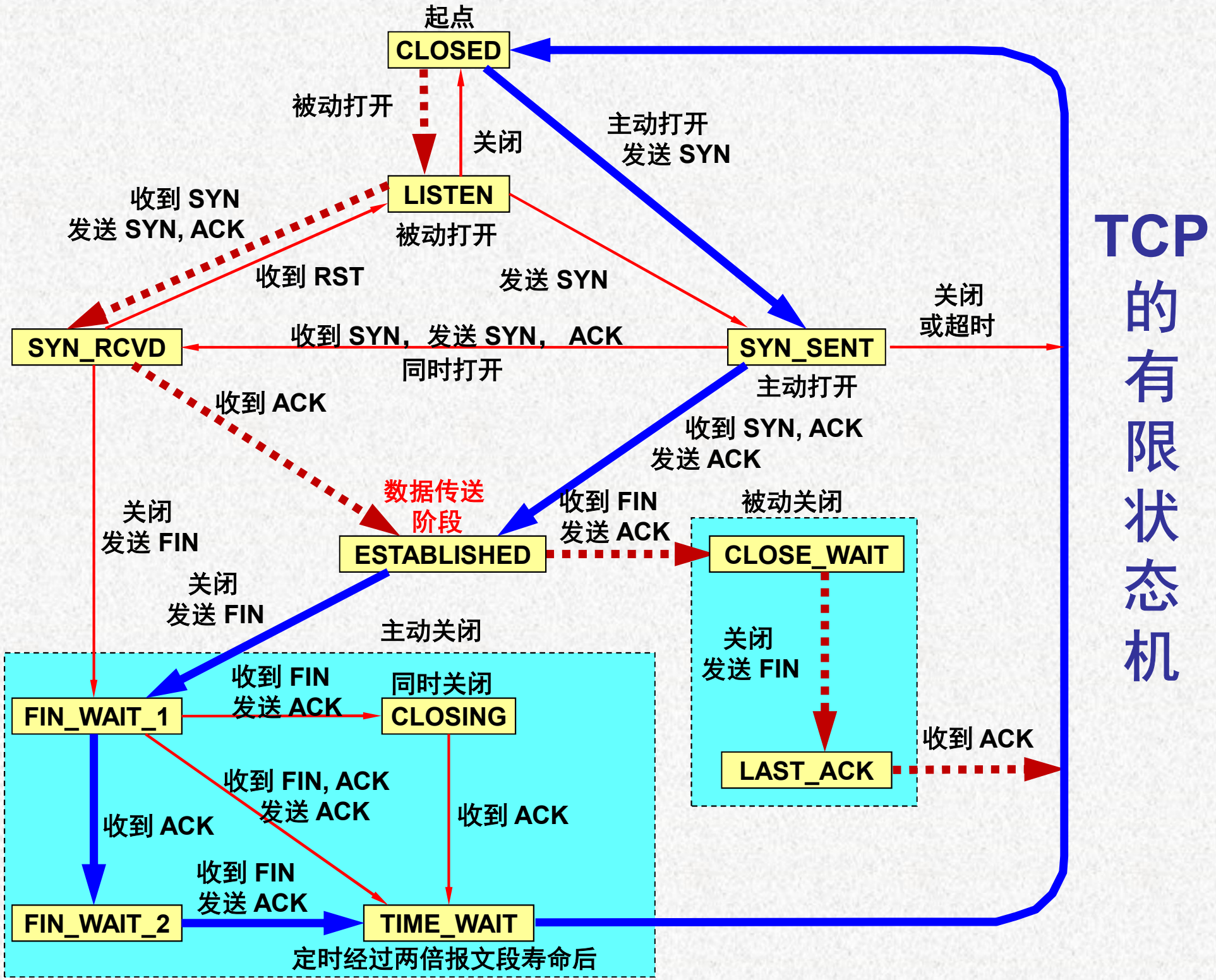
- 第一，为了保证 A 发送的最后一个 ACK 报文段能够到达 B。
- 第二，防止“已失效的连接请求报文段”出现在本连接中。A 在发送完最后一个 ACK 报文段后，再经过时间 2MSL，就可以使本连接持续的时间内所产生的所有报文段，都从网络中消失。这样就可以使下一个新的连接中不会出现这种旧的连接请求报文段。

5.9.3 TCP 的有限状态机

- TCP 的有限状态机可以更清晰地看出 TCP 连接的各种状态之间的关系。
- TCP 有限状态机的图中每一个方框都是 TCP 可能具有的状态。
- 每个方框中的大写英文字符串是 TCP 标准所使用的 TCP 连接状态名。
- 状态之间的箭头表示可能发生的状态变迁。

5.9.3 TCP 的有限状态机

- 箭头旁边的字，表明引起这种变迁的原因，或表明发生状态变迁后又出现什么动作。
- 图中有三种不同的箭头。
  - 粗实线箭头表示对客户进程的正常变迁。
  - 粗虚线箭头表示对服务器进程的正常变迁。
  - 细线箭头表示异常变迁。


起点
CLOSED
被动打开
关闭
主动打开
发送 SYN
LISTEN
收到 SYN
发送 SYN, ACK
被动打开
收到 RST
发送 SYN
关闭
或超时
SYN_RCVD
收到 SYN，发送 SYN， ACK
同时打开
SYN_SENT
主动打开
收到 ACK
收到 SYN, ACK
发送 ACK
数据传送
阶段
关闭
发送 FIN
收到 FIN
发送 ACK
被动关闭
ESTABLISHED
CLOSE_WAIT
关闭
发送 FIN
主动关闭
收到 FIN
发送 ACK
同时关闭
FIN_WAIT_1
CLOSING
关闭
发送 FIN
收到 ACK
收到 FIN, ACK
发送 ACK
收到 ACK
LAST_ACK
收到 ACK
收到 FIN
发送 ACK
FIN_WAIT_2
TIME_WAIT
定时经过两倍报文段寿命后

TCP 的有限状态机