

# 第 5 章 运输层



# 主要内容:



- 5.1 运输层协议概述
  - 运输层的功能
  - 运输层的两个协议
- 5.2 用户数据报协议 UDP
  - UDP的特点和首部格式
- 5.3 传输控制协议 TCP 概述
  - TCP的特点和连接
- 5.4 可靠传输的工作原理
  - 停止等待协议
  - 连续ARQ协议

# 5.1 运输层协议概述

---

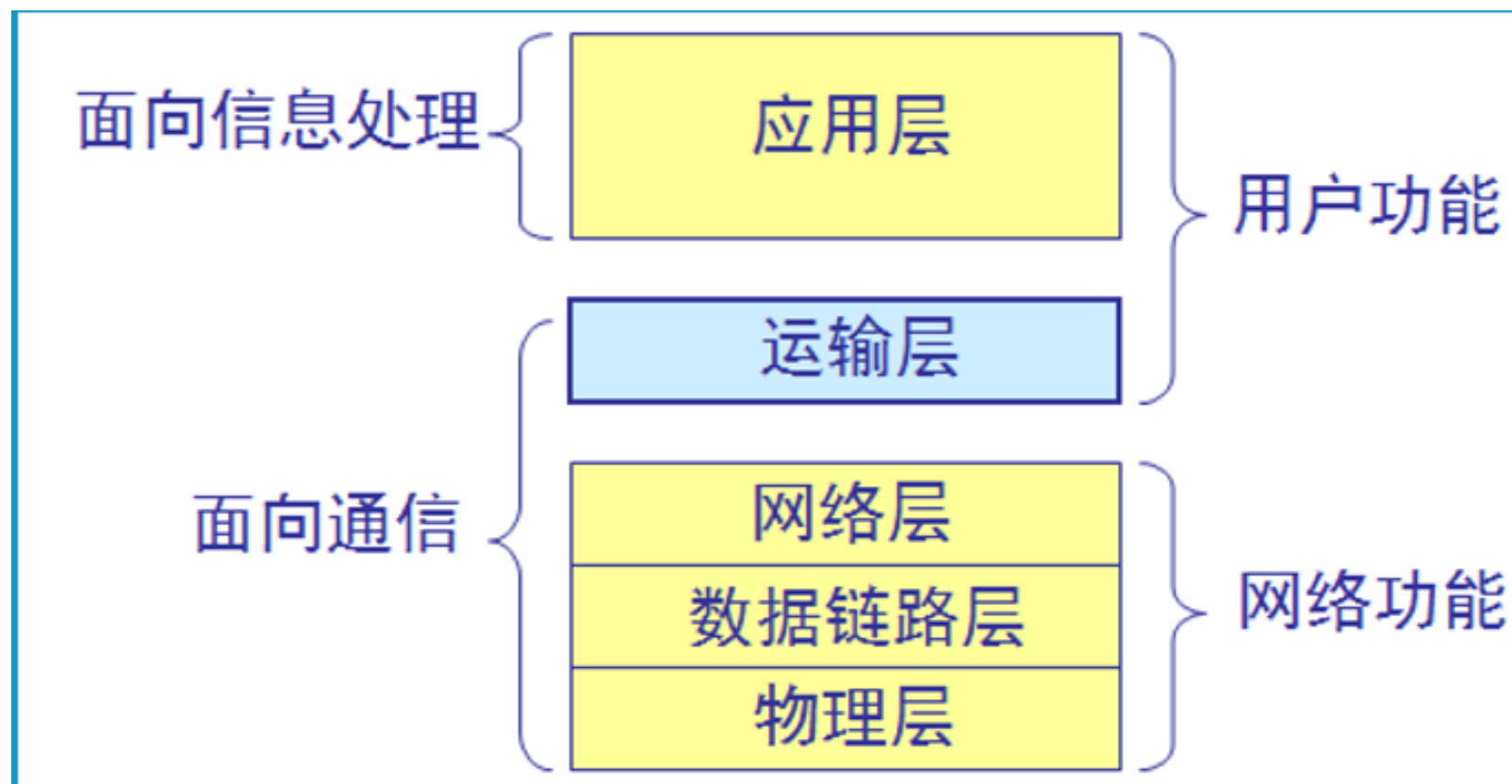


- 5.1.1 运输层的功能 ★
- 5.1.2 运输层的两个主要协议 ★
- 5.1.3 运输层的端口

# 5.1.1 运输层的功能

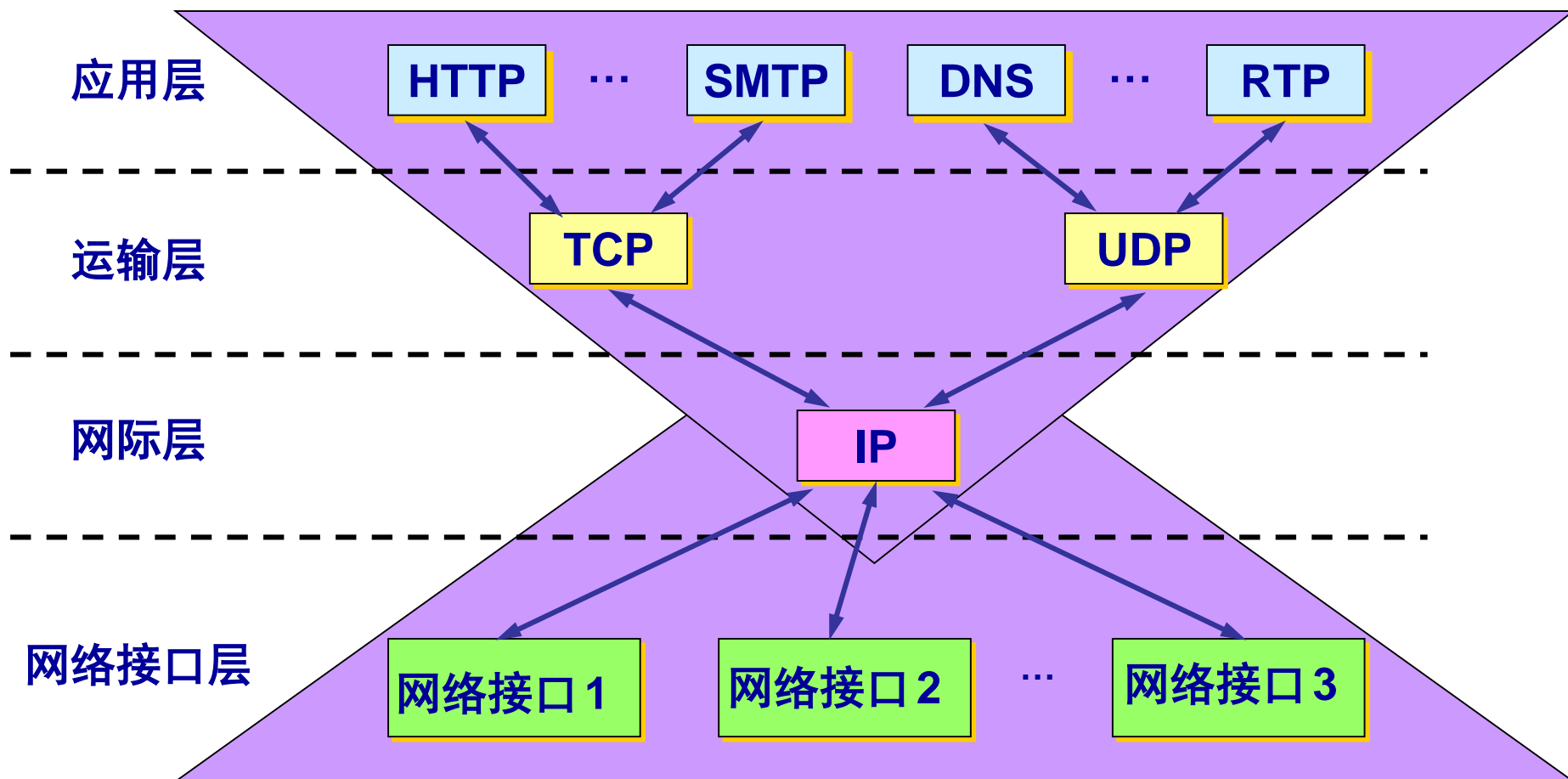


- **运输层的功能**是向上面的应用层提供逻辑通信服务。它属于面向**通信部分**的**最高层**，同时也是**用户功能**中的**最低层**。



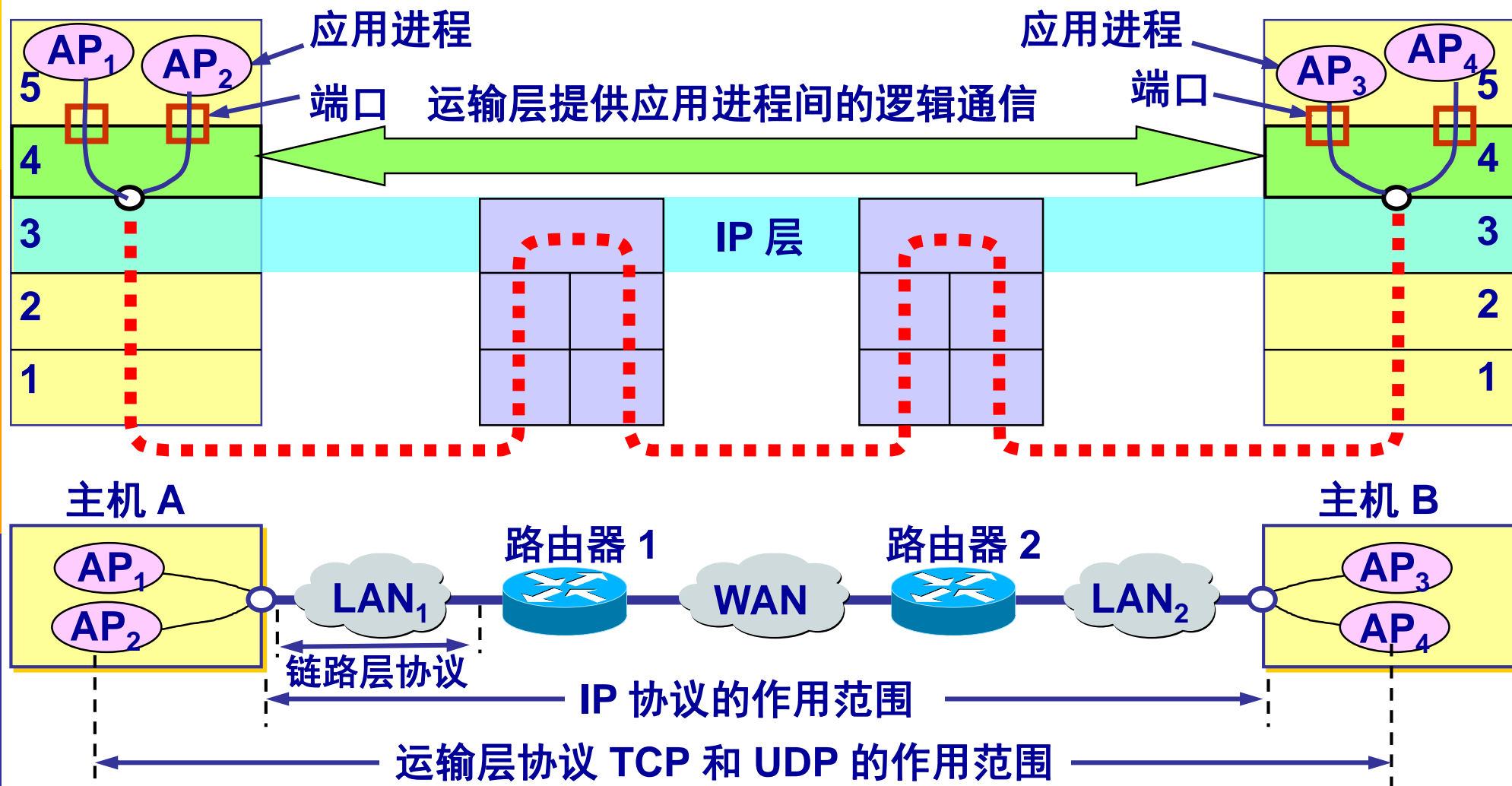
# IP over Everything

## IP可应用到各式各样的网络上



沙漏计时器形状的 TCP/IP 协议族

# 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

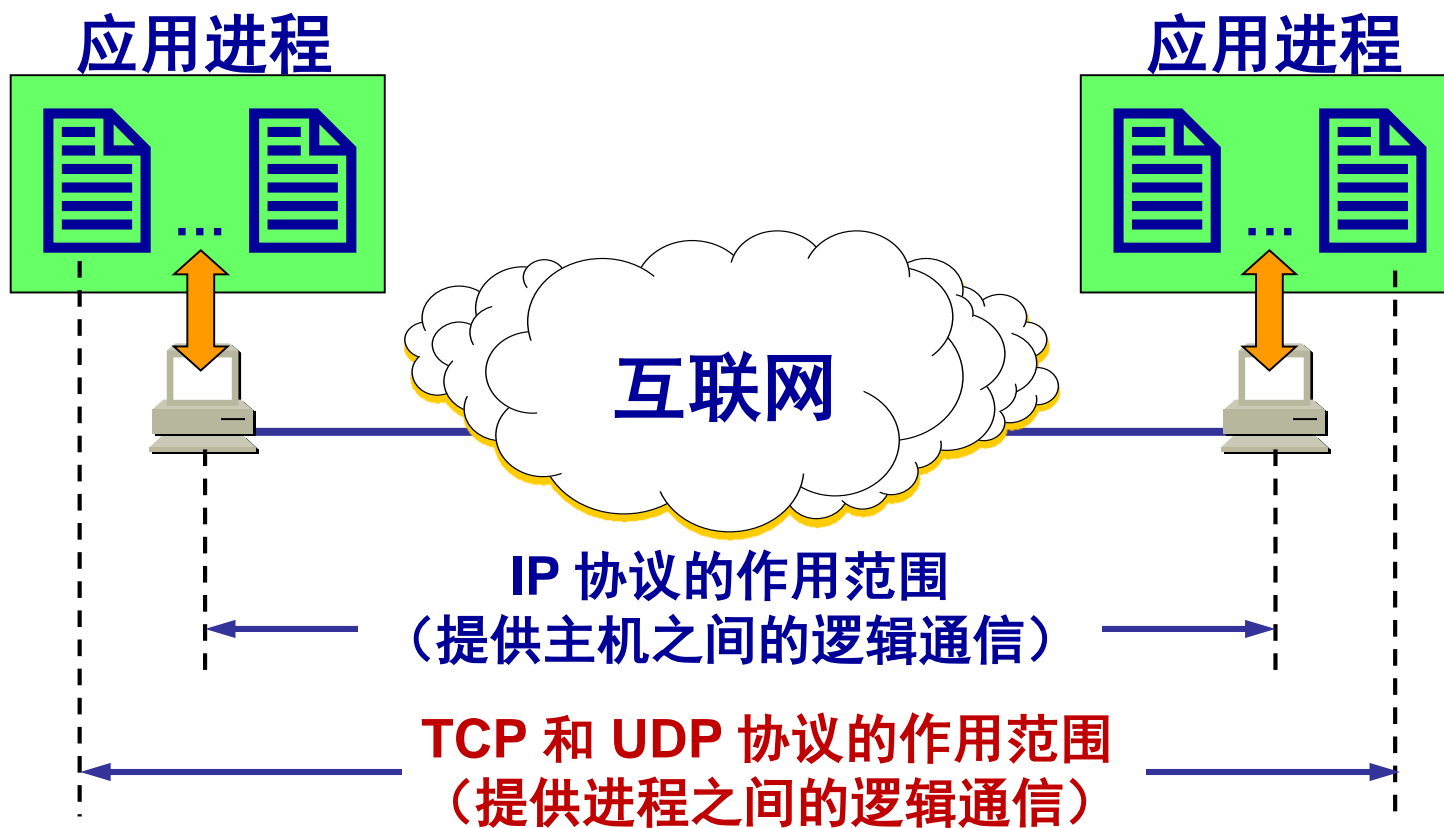


- “**逻辑通信**”的意思是“好像是这样通信，但事实上并非真的这样通信”。
- **从IP层来说，通信的两端是两台主机。**但“两台主机之间的通信”这种说法还不够清楚。
- **从运输层的角度看，通信的真正端点并不是主机而是主机中的进程。**也就是说，端到端的通信是应用进程之间的通信。

# 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

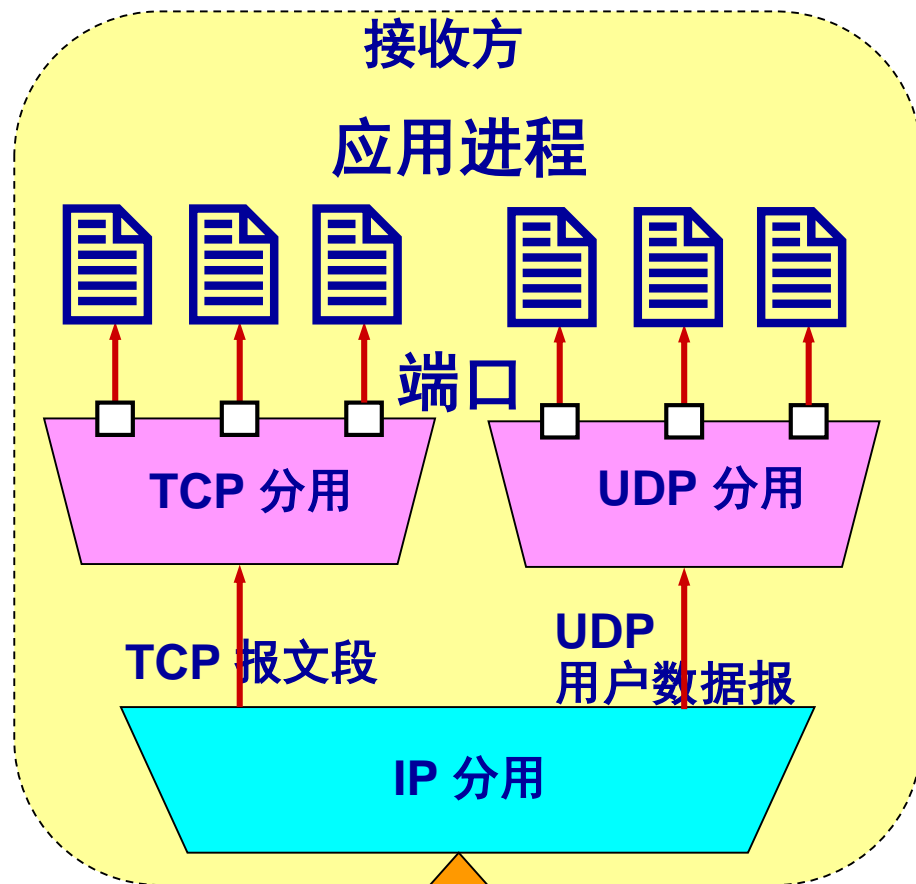
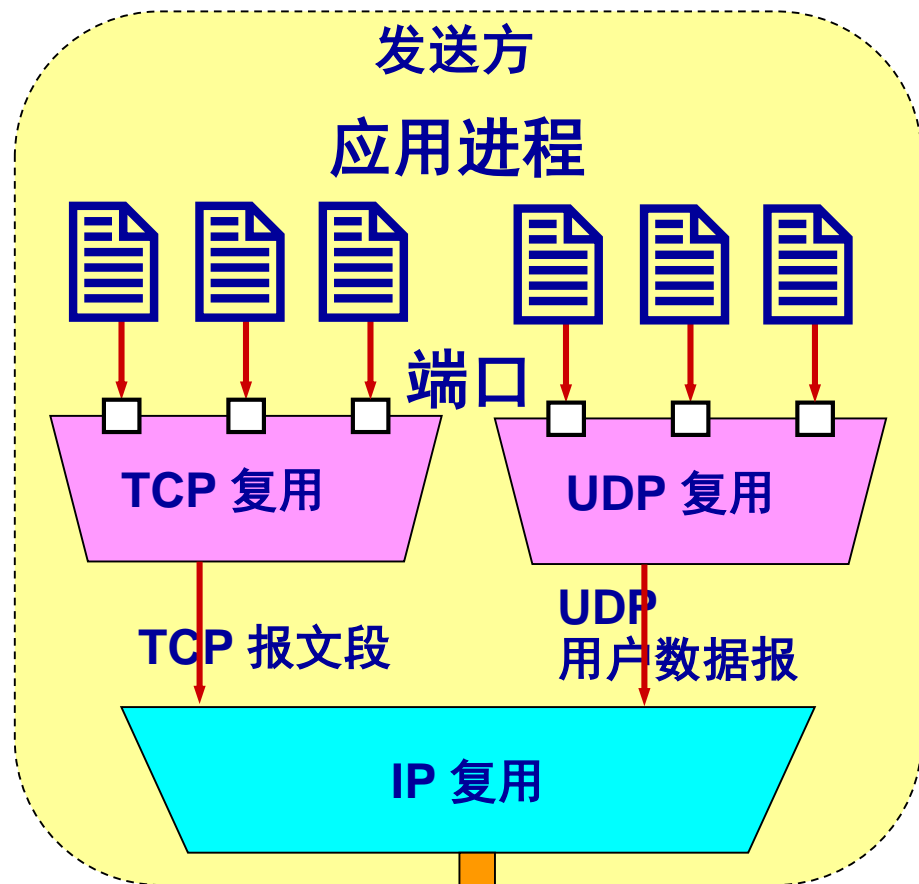


- 在一台主机中经常有**多个应用进程**同时分别和另一台主机中的多个应用进程通信。
- 这表明运输层有一个很重要的功能——**复用** (multiplexing)和**分用** (demultiplexing)。

# 基于端口的复用和分用功能



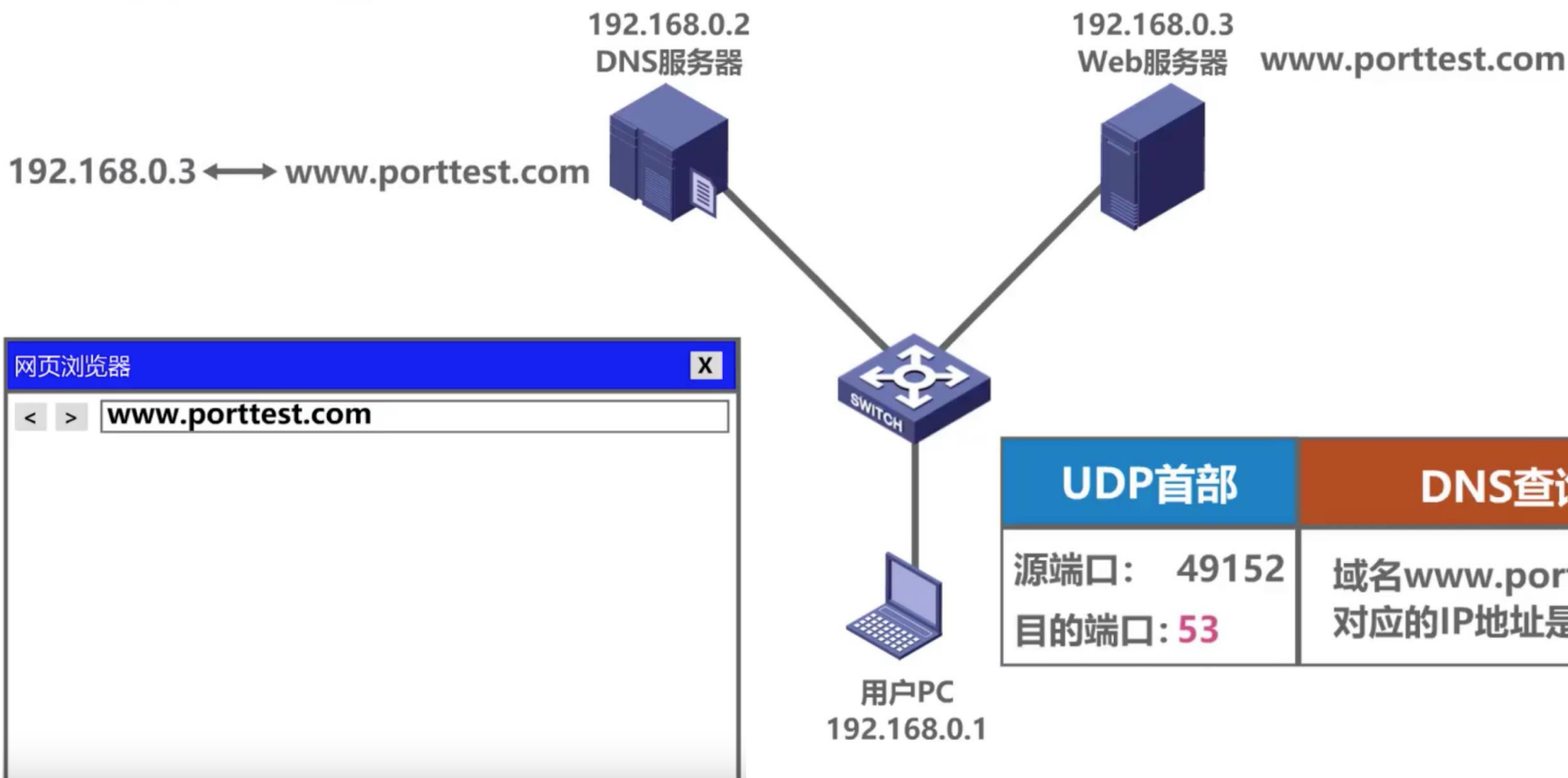
应用层  
运输层  
网络层



# 端口复用和分用示例



【举例】运输层端口号



# 端口复用和分用示例

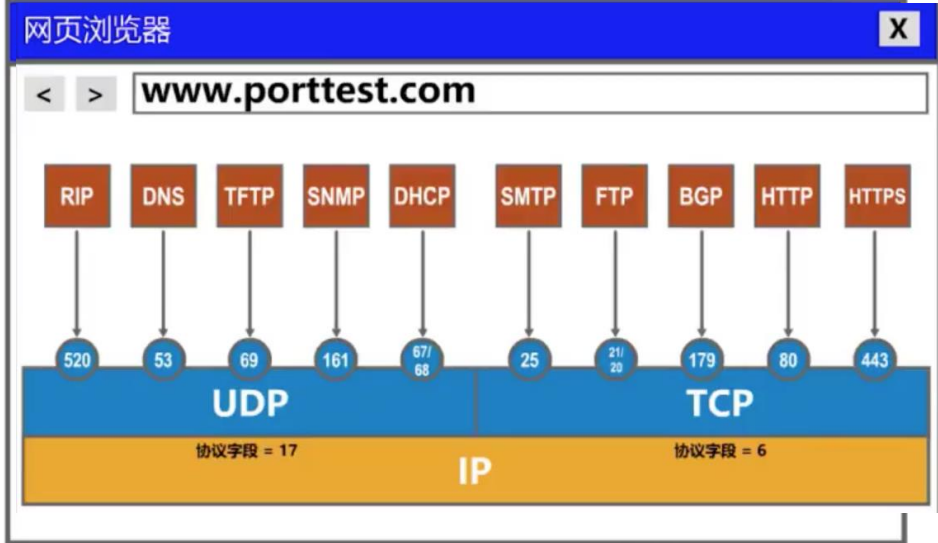


## 【举例】运输层端口号

UDP首部	DNS查询请求
源端口: 49152	域名www.porttest.com
目的端口: 53	对应的IP地址是什么?

192.168.0.3 ↔ www.porttest.com

DNS响应	UDP首部
域名www.porttest.com对应的IP地址是192.168.0.3	源端口: 53
	目的端口: 49152



192.168.0.2  
DNS服务器



192.168.0.3  
Web服务器 www.porttest.com



用户PC  
192.168.0.

HTTP请求	TCP首部
首页内容是什么?	源端口: 49152
	目的端口: 80

TCP首部	HTTP响应
源端口: 80	首页内容是.....
目的端口: 49152	

# 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**。

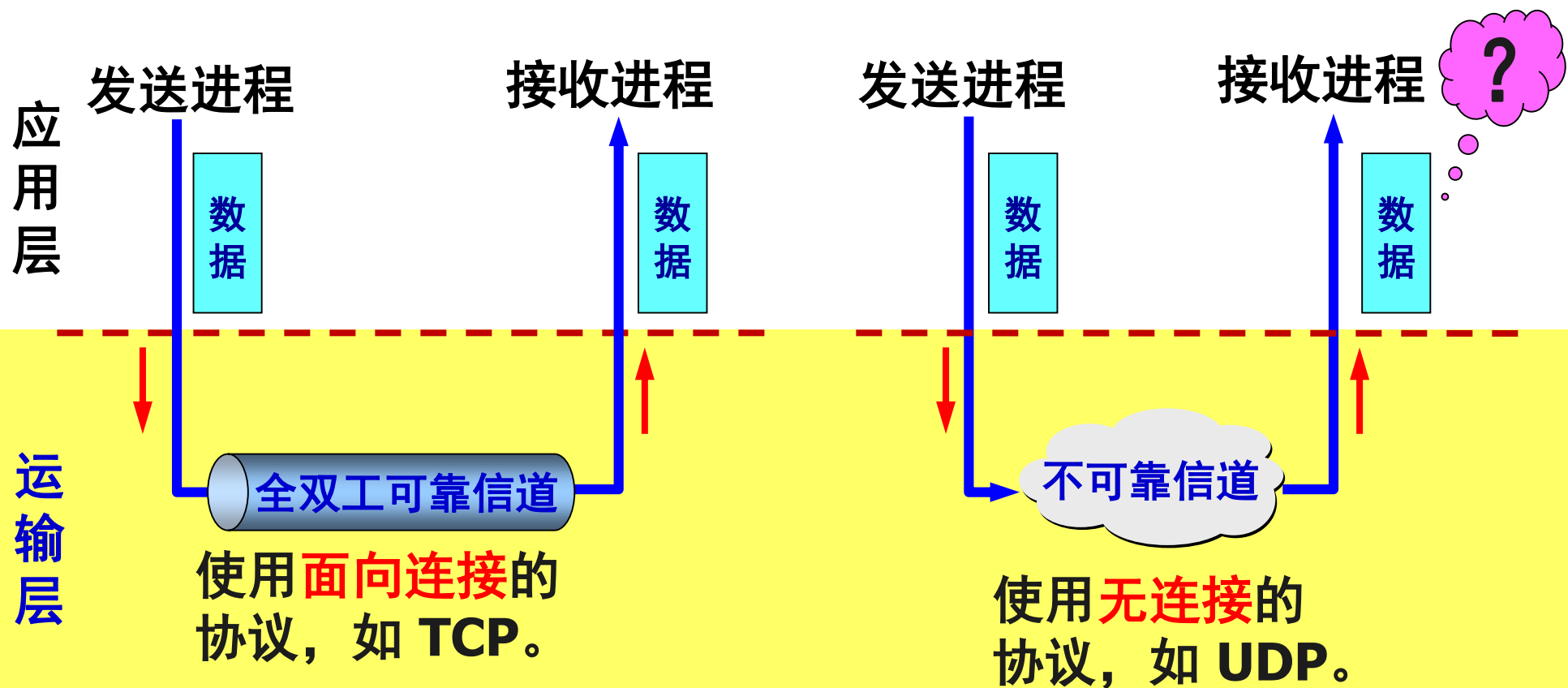


# 两种不同的运输协议



- 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**。
- 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**不可靠半双工信道**。

# 可靠信道与不可靠信道

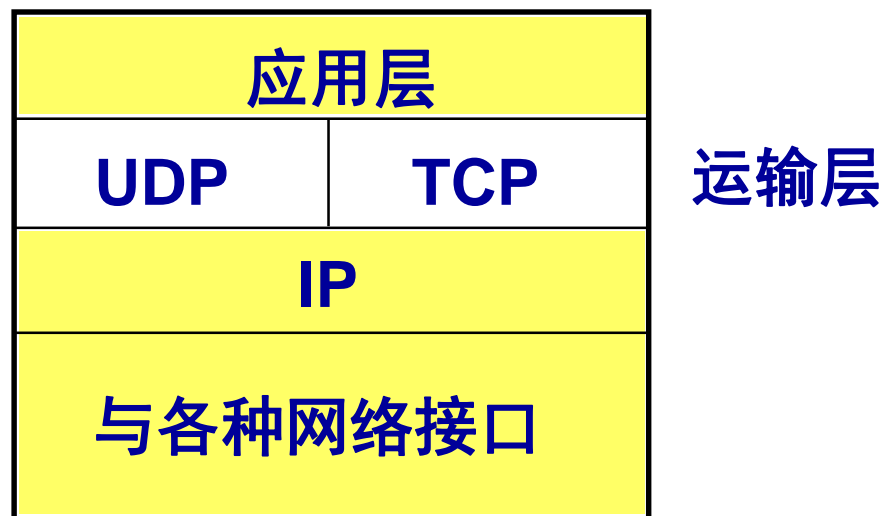


## 5.1.2 运输层的两个主要协议



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

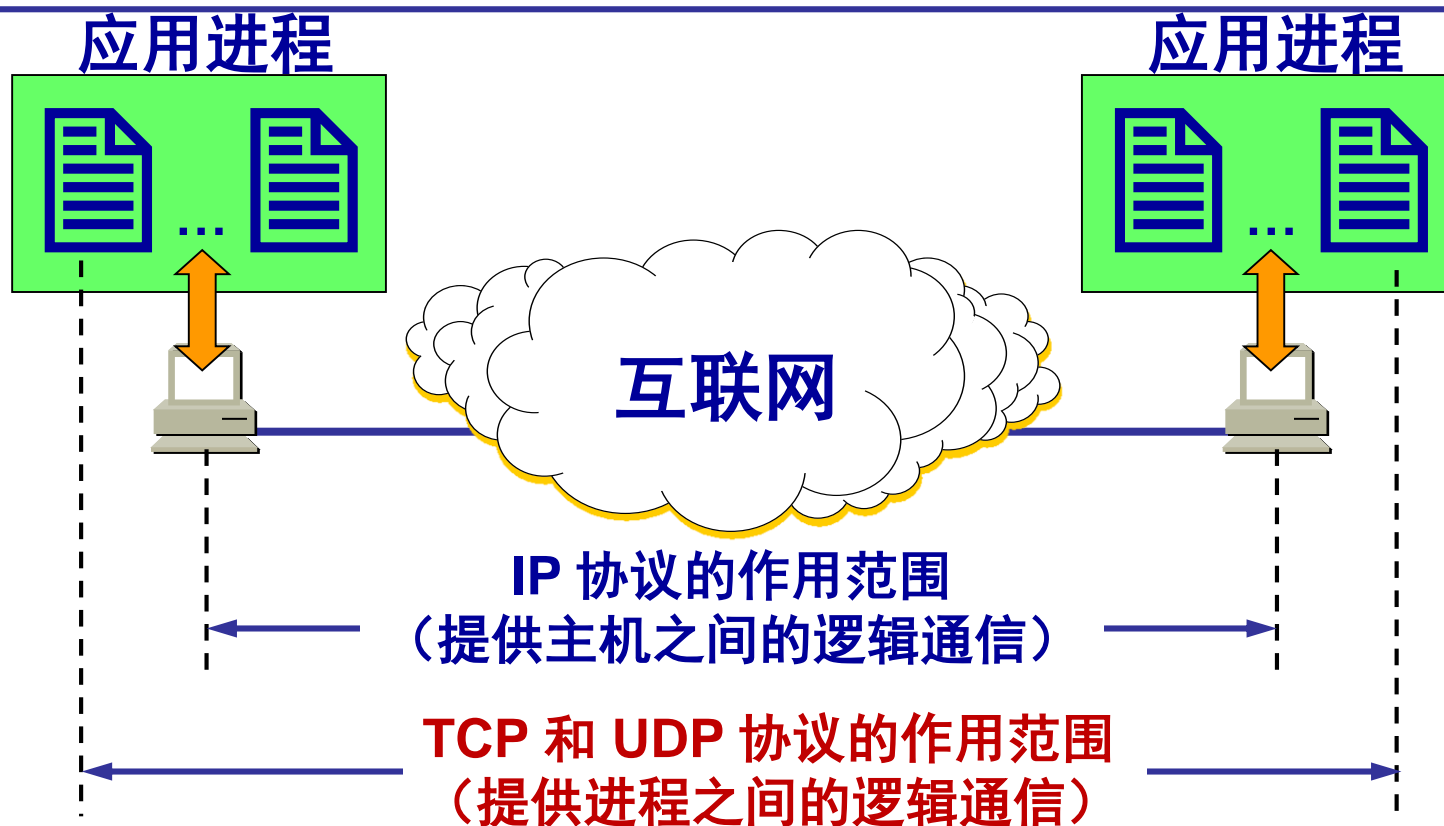
- 提供面向连接的服务。



逻辑连接

- 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。

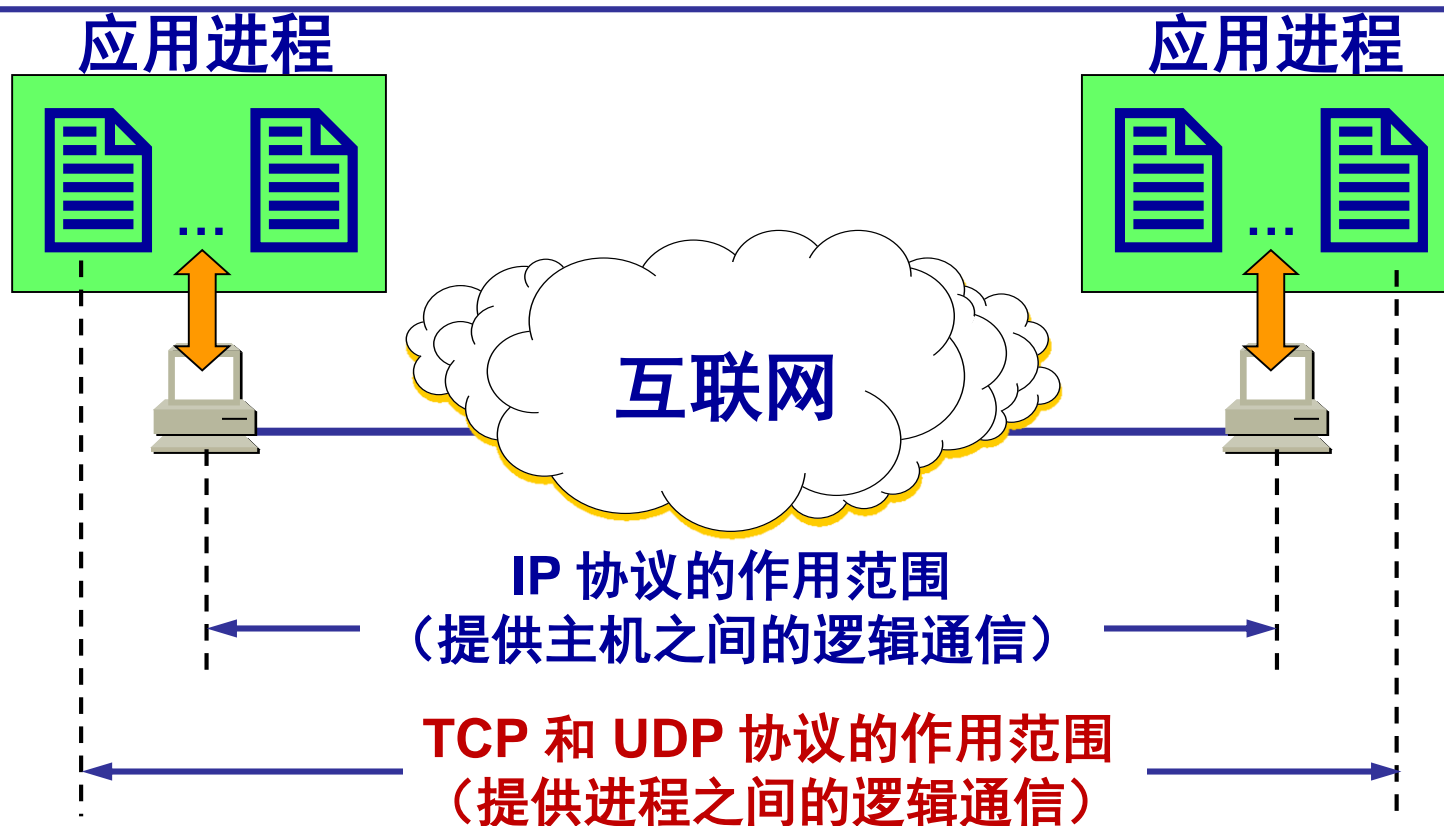
## 5.1.3 运输层的端口



网络层是为主机之间提供逻辑通信。 —————> 主机的身份标志？

↓  
**IP地址**

## 5.1.3 运输层的端口



运输层为应用进程之间提供端到端的逻辑通信。

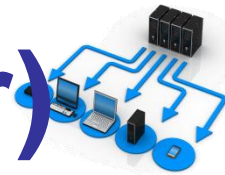


进程的身份标志？



端口

# 端口号 (protocol port number)



- 为了标志主机中的进程使用**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。

# 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

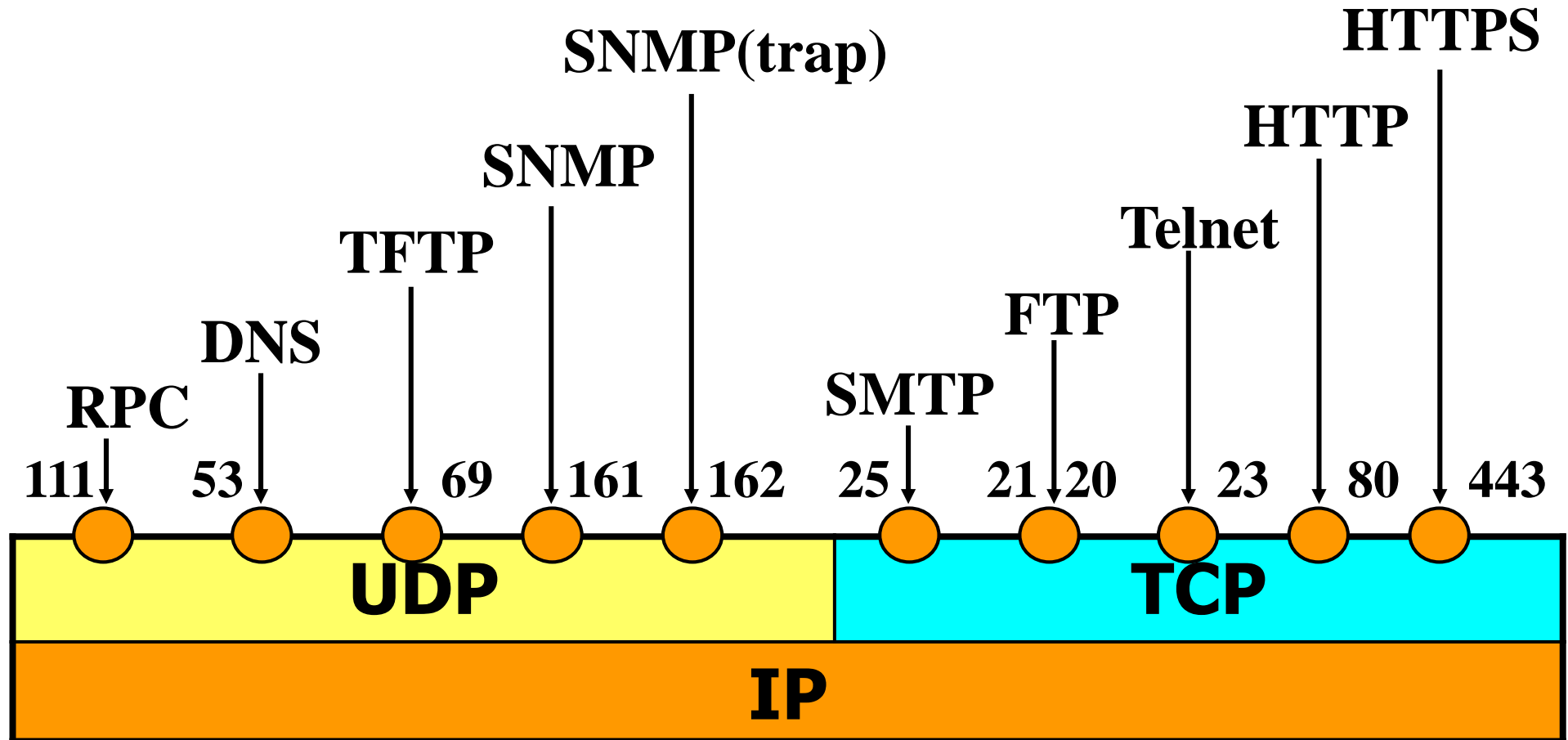
- **熟知端口**，数值一般为 0 ~ 1023。
- **登记端口号**，数值为 1024 ~ 49151，为没有熟知端口号的应用程序使用的。使用这个范围的端口号必须在 IANA 登记，以防止重复。

## (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



# Windows 端口查询



- 通过cmd命令，打开命令行窗口
- 输入命令：netstat -ano，列出所有端口的情况。

```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7600]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\>netstat -ano

活动连接

协议 本地地址 外部地址 状态 PID
TCP 0.0.0.0:135 0.0.0.0:0 LISTENING 1068
TCP 0.0.0.0:445 0.0.0.0:0 LISTENING 4
TCP 0.0.0.0:49152 0.0.0.0:0 LISTENING 680
TCP 0.0.0.0:49153 0.0.0.0:0 LISTENING 1156
TCP 0.0.0.0:49154 0.0.0.0:0 LISTENING 1236
TCP 0.0.0.0:49156 0.0.0.0:0 LISTENING 736
TCP 0.0.0.0:49157 0.0.0.0:0 LISTENING 2720
TCP 0.0.0.0:49158 0.0.0.0:0 LISTENING 756
TCP 10.2.13.64:139 0.0.0.0:0 LISTENING 4
TCP 10.2.13.64:51432 183.60.62.158:443 ESTABLISHED 5628
TCP 10.2.13.64:51497 111.30.131.159:443 ESTABLISHED 4260
```

# Windows 端口查询



- 输入命令：netstat -aon|findstr “49157”，可查指定端口。

```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>netstat -aon|findstr "80"

TCP    0.0.0.0:80          0.0.0.0:*          LISTENING           4
TCP    0.0.0.0:5091       0.0.0.0:*          LISTENING           7680
TCP    0.0.0.0:8000       0.0.0.0:*          LISTENING           5316
TCP    0.0.0.0:18000      0.0.0.0:*          LISTENING           5316
TCP    10.1.76.179:49296 14.17.19.60:80     CLOSE_WAIT          10740
TCP    10.1.76.179:50563 113.96.232.146:80 ESTABLISHED          12512
TCP    10.1.76.179:63507 59.37.96.250:80    ESTABLISHED          1456
TCP    10.1.76.179:63696 110.43.81.33:80    TIME_WAIT            0
TCP    10.1.76.179:63697 117.48.124.216:80  TIME_WAIT            0
TCP    10.1.76.179:63699 117.48.124.216:80  TIME_WAIT            0
TCP    10.1.76.179:63700 219.133.60.246:80  TIME_WAIT            0
TCP    10.1.76.179:63703 139.199.215.55:80  TIME_WAIT            0
TCP    10.1.76.179:63704 220.170.53.37:80   TIME_WAIT            0
TCP    10.1.76.179:63711 124.232.169.96:443 ESTABLISHED          17580
TCP    10.1.76.179:63729 124.232.169.97:443 ESTABLISHED          17580
TCP    10.1.76.179:63754 113.219.136.7:443  ESTABLISHED          17580
TCP    10.1.76.179:63755 113.219.136.8:443  ESTABLISHED          17580
TCP    10.1.76.179:63756 113.219.136.8:443  ESTABLISHED          17580
TCP    10.1.76.179:63772 117.21.189.122:443 ESTABLISHED          17580
TCP    10.1.76.179:63773 117.21.189.123:443 ESTABLISHED          17580
TCP    10.1.76.179:63774 117.21.189.123:443 ESTABLISHED          17580
```

# Windows 端口查询



- 输入命令：tasklist|findstr "2720"，可查指定端口。

```
管理员: C:\Windows\system32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Administrator>netstat -aon|findstr "80"

TCP    0.0.0.0:80          0.0.0.0:0        LISTENING        4
TCP    0.0.0.0:5091       0.0.0.0:0        LISTENING        7680
TCP    0.0.0.0:8000       0.0.0.0:0        LISTENING        5316
TCP    0.0.0.0:18000      0.0.0.0:0        LISTENING        5316
TCP    10.1.76.179:49296 14.17.19.60:80   CLOSE_WAIT       10740
TCP    10.1.76.179:50563 113.96.232.146:8080 ESTABLISHED      12512
TCP    10.1.76.179:63507 59.37.96.250:80  ESTABLISHED      1456
TCP    10.1.76.179:63696 110.43.81.33:80  TIME_WAIT        0
TCP    10.1.76.179:63697 117.48.124.216:80 TIME_WAIT        0
TCP    10.1.76.179:63699 117.48.124.216:80 TIME_WAIT        0
TCP    10.1.76.179:63700 219.133.60.246:80 TIME_WAIT        0
TCP    10.1.76.179:63703 139.199.215.55:80 TIME_WAIT        0
TCP    10.1.76.179:63704 220.170.53.37:80  TIME_WAIT        0
TCP    10.1.76.179:63711 124.232.169.96:443 ESTABLISHED      17580
TCP    10.1.76.179:63729 124.232.169.97:443 ESTABLISHED      17580
TCP    10.1.76.179:63754 113.219.136.7:443 ESTABLISHED      17580
TCP    10.1.76.179:63755 113.219.136.8:443 ESTABLISHED      17580
TCP    10.1.76.179:63756 113.219.136.8:443 ESTABLISHED      17580
TCP    10.1.76.179:63772 117.21.189.122:443 ESTABLISHED      17580
TCP    10.1.76.179:63773 117.21.189.123:443 ESTABLISHED      17580
TCP    10.1.76.179:63774 117.21.189.123:443 ESTABLISHED      17580
```

# 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，既不开并，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（半双工）。
- (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



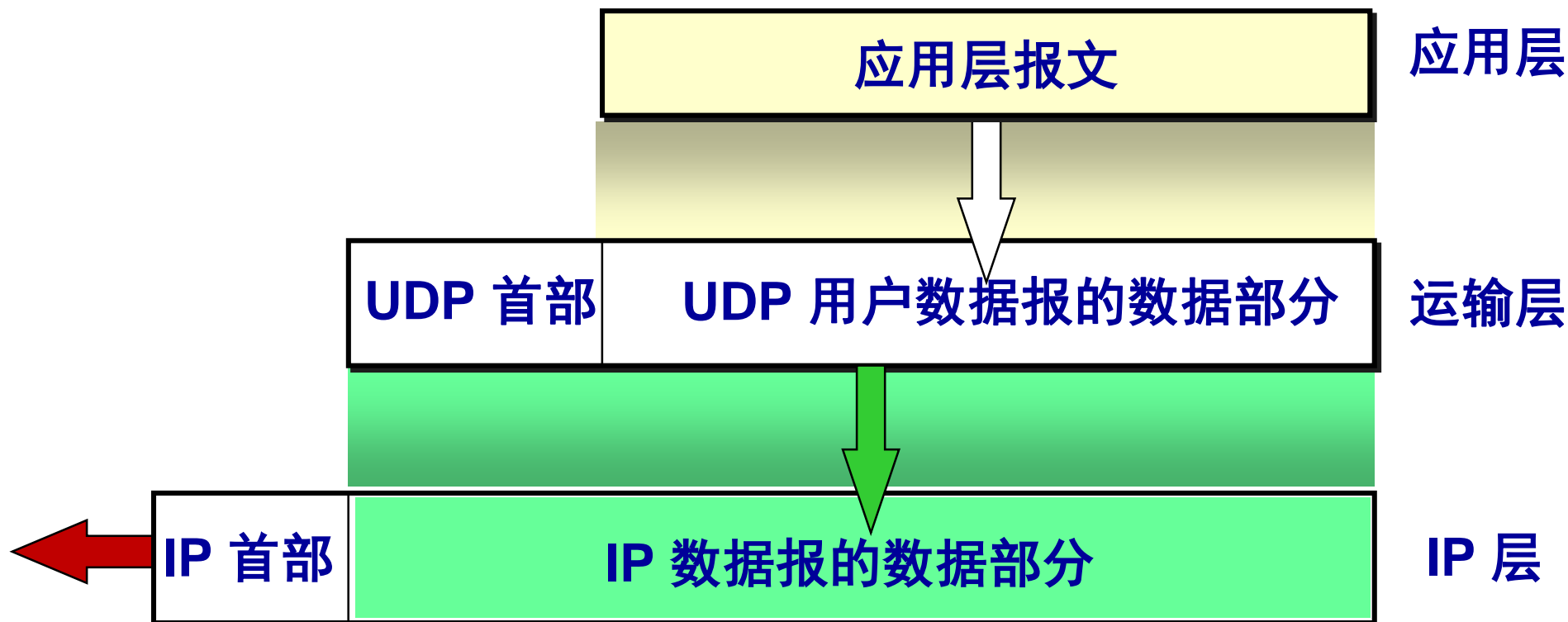
- 发送方 UDP 对应用程序交下来的报文，在添加首部后就向下交付 IP 层。UDP 对应用层交下来的报文，**既不合并，也不拆分**，而是保留这些报文的边界。
- 应用层交给 UDP 多长的报文，UDP 就照样发送，即**一次发送一个报文**。

# 面向报文的 UDP



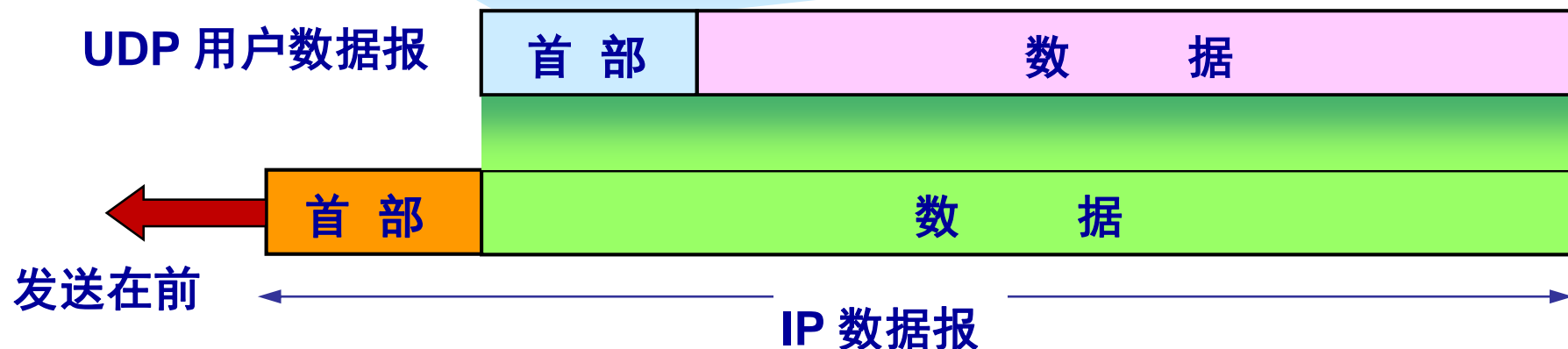
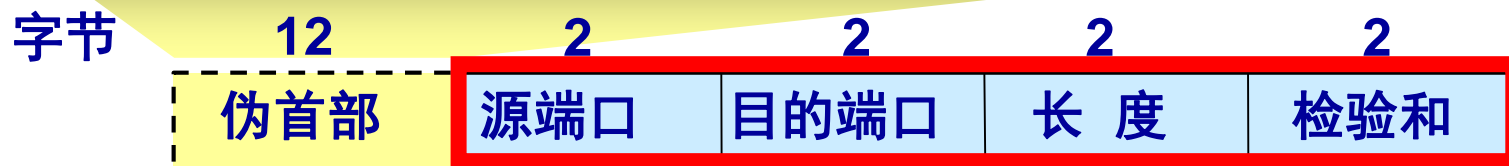
- 接收方 UDP 对 IP 层交上来的 UDP 用户数据报，在去除首部后就原封不动地交付上层的应用进程，**一次交付一个完整的报文。**
- 应用程序必须**选择合适大小的报文。**
  - **若报文太长**，UDP 把它交给 IP 层后，IP 层在传送时可能要进行分片，这会降低 IP 层的效率。
  - **若报文太短**，UDP 把它交给 IP 层后，会使 IP 数据报的首部的相对长度太大，这也降低了 IP 层的效率。

# UDP 是面向报文的



## 5.2.2 UDP 的首部格式

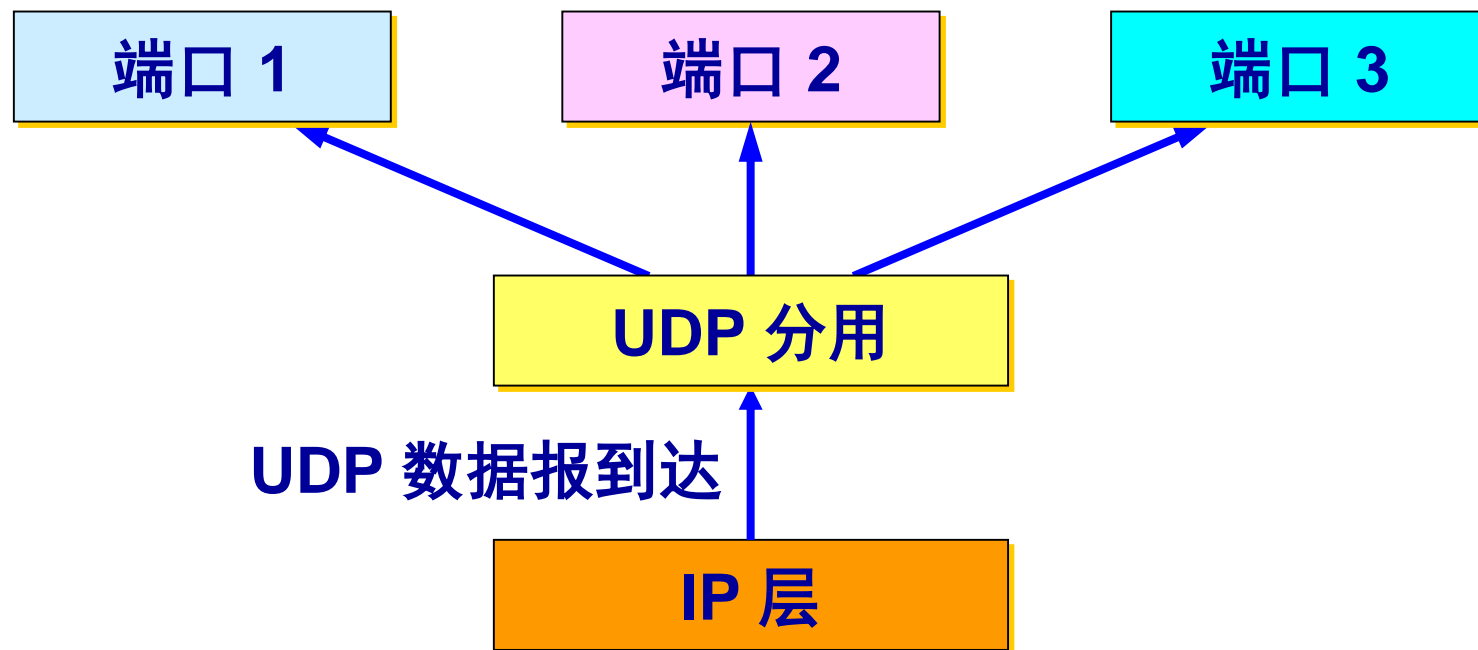
用户数据报 UDP 有两个字段：数据字段和首部字段。首部字段有 8 个字节，由 4 个字段组成，每个字段都是 2 个字节。



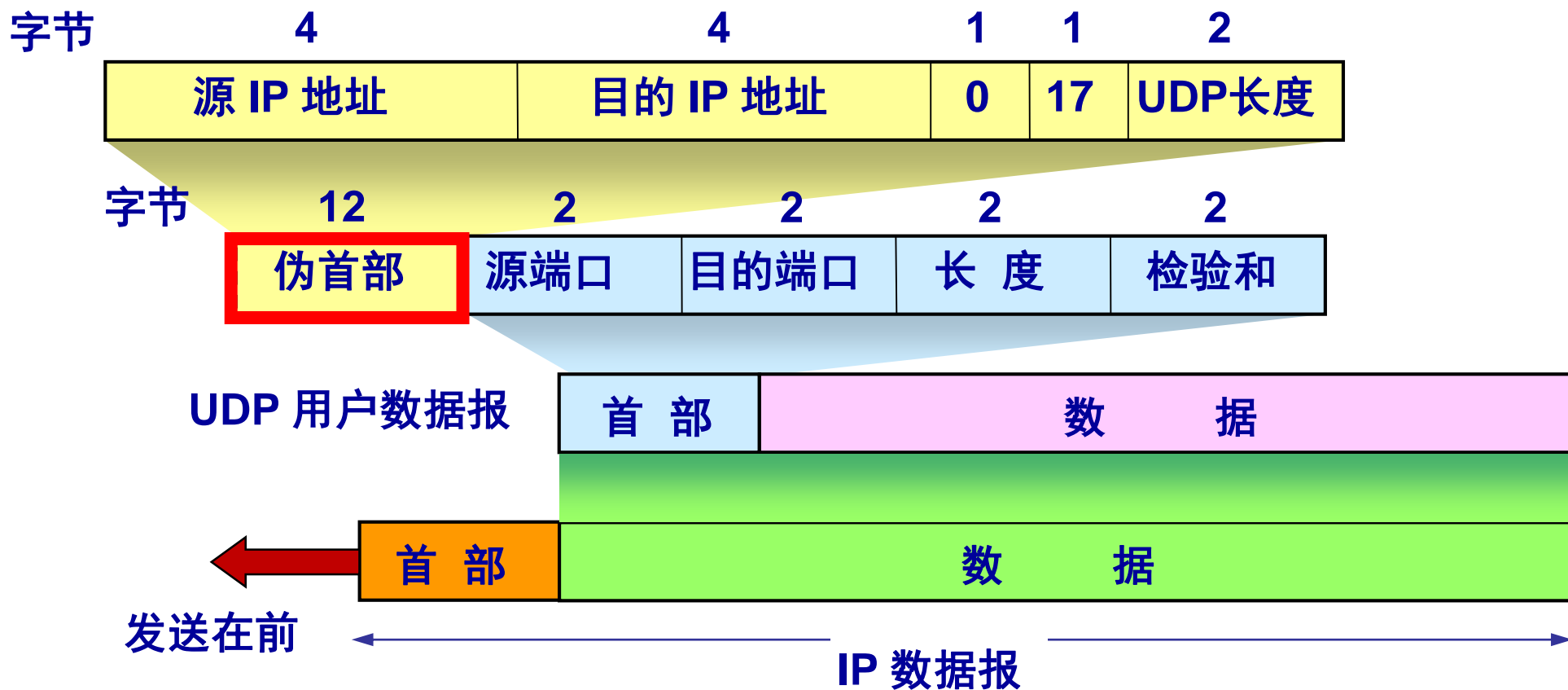
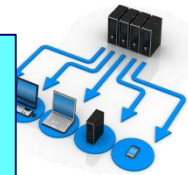
# UDP 基于端口的分用



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。



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



12 字节 伪首部	153.19.8.104			
	171.3.14.11			
	全 0	17	15	
8 字节 UDP 首部	1087		13	
	15		全 0	
7 字节 数据	数据	数据	数据	数据
	数据	数据	数据	全 0

填充

UDP 的检验和是把首部和数据部分一起都检验。

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

# 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



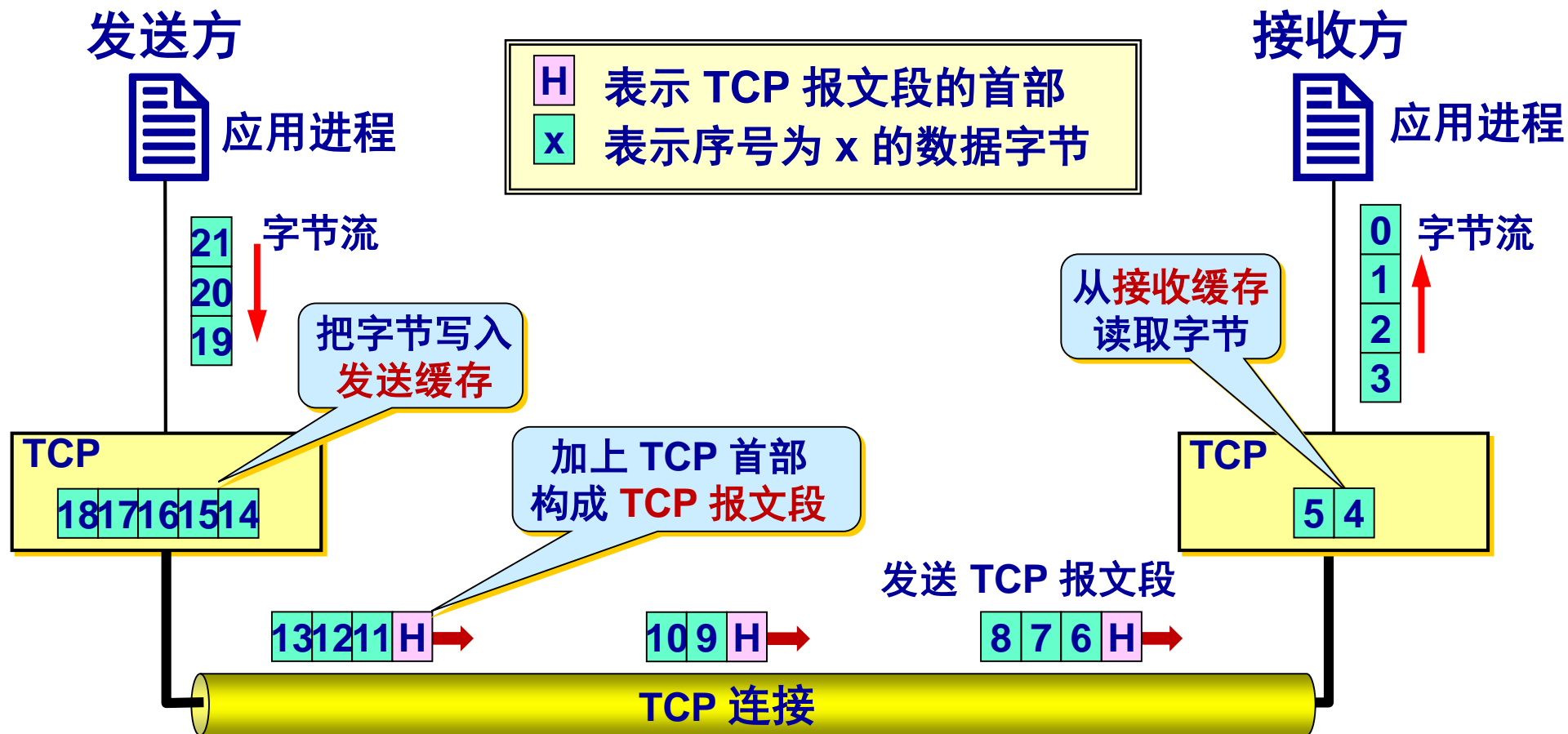
- TCP 是**面向连接**的运输层协议。
- 每一条 TCP 连接**只能有两个端点** (endpoint), 每一条 TCP 连接**只能是点对点的** (一对一)。
- TCP 提供**可靠交付**的服务。
- TCP 提供**全双工**通信。
- **面向字节流**
  - TCP 中的“**流**” (stream) 指的是流入或流出进程的字节序列。
  - “**面向字节流**”的含义是：虽然应用程序和 TCP 的交互是一次一个数据块，但 TCP 把应用程序交下来的数据看成仅仅是一连串无结构的字节流。

# TCP 面向流的概念

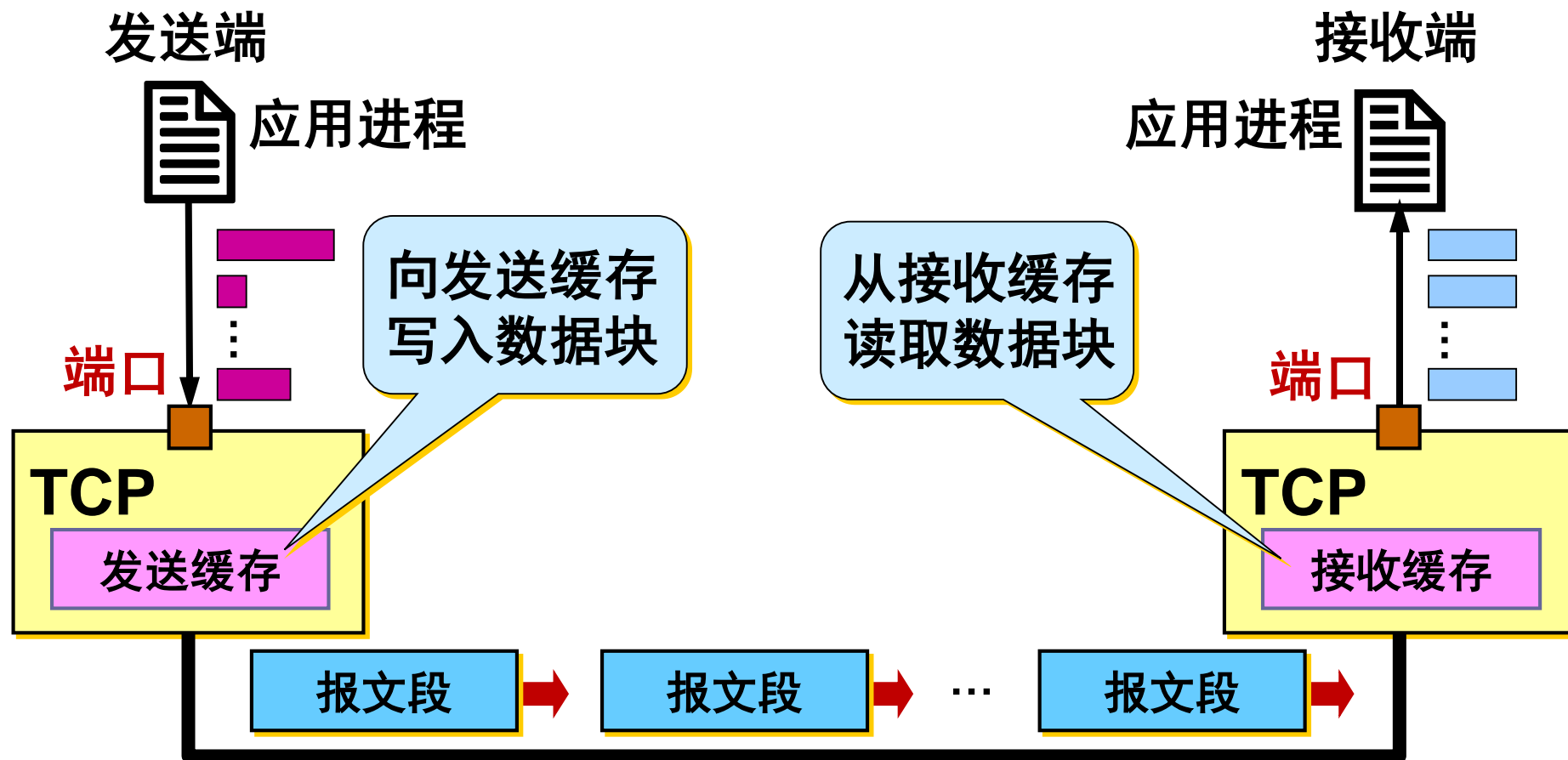


- TCP **不保证**接收方应用程序所收到的数据块和发送方应用程序所发出的**数据块具有对应大小的关系**。
- 但接收方应用程序收到的字节流必须和发送方应用程序发出的**字节流完全一样**。

# TCP 面向流的概念



# 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。



# 第 5 章 运输层（二）



# 复习

---



- **5.1.1 运输层的功能** ★
- **5.1.2 运输层的两个主要协议** ★
- **5.1.3 运输层的端口**

# 本讲重点

---



## ■ 5.4 可靠传输的工作原理

- 停止等待协议

- 连续ARQ协议



## ■ 5.5 TCP 报文段的首部格式

# 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。
- **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。

# 可靠传输的工作原理



TCP发送的报文段是交给IP层传送的。但IP层**只能提供尽最大努力服务**，也就是说，TCP下面的网络所提供的是**不可靠的传输**。因此，TCP必须采用适当的措施才能使得两个运输层之间的通信变得可靠。

可靠传输

传输无差错（不突变）？

不丢失？

不乱序？

不重复？

## 5.4.1 停止等待协议

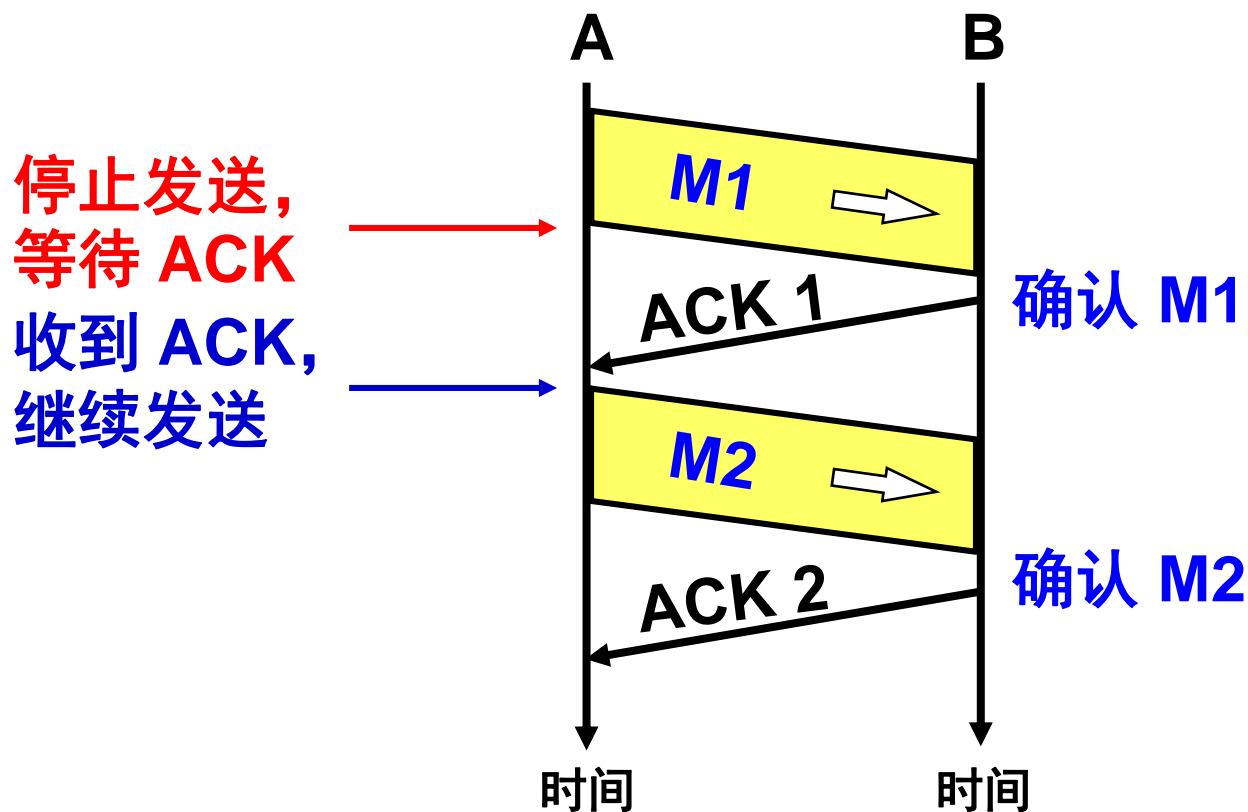


- **“停止等待”** 就是每发送完一个分组就停止发送，等待对方的确认。在收到确认后再发送下一个分组。

# 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。



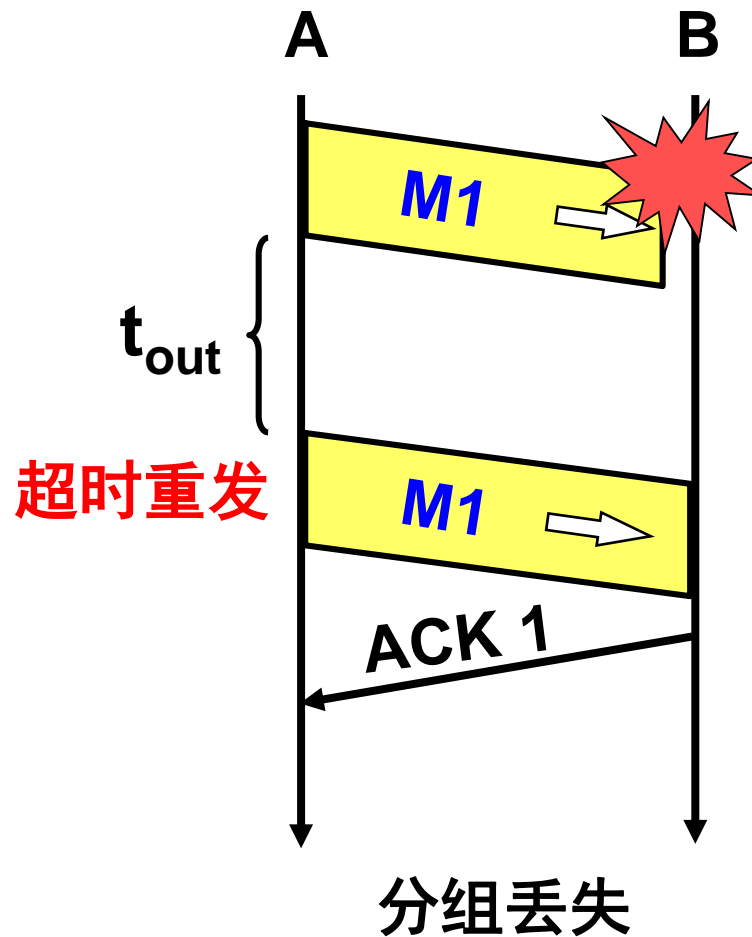
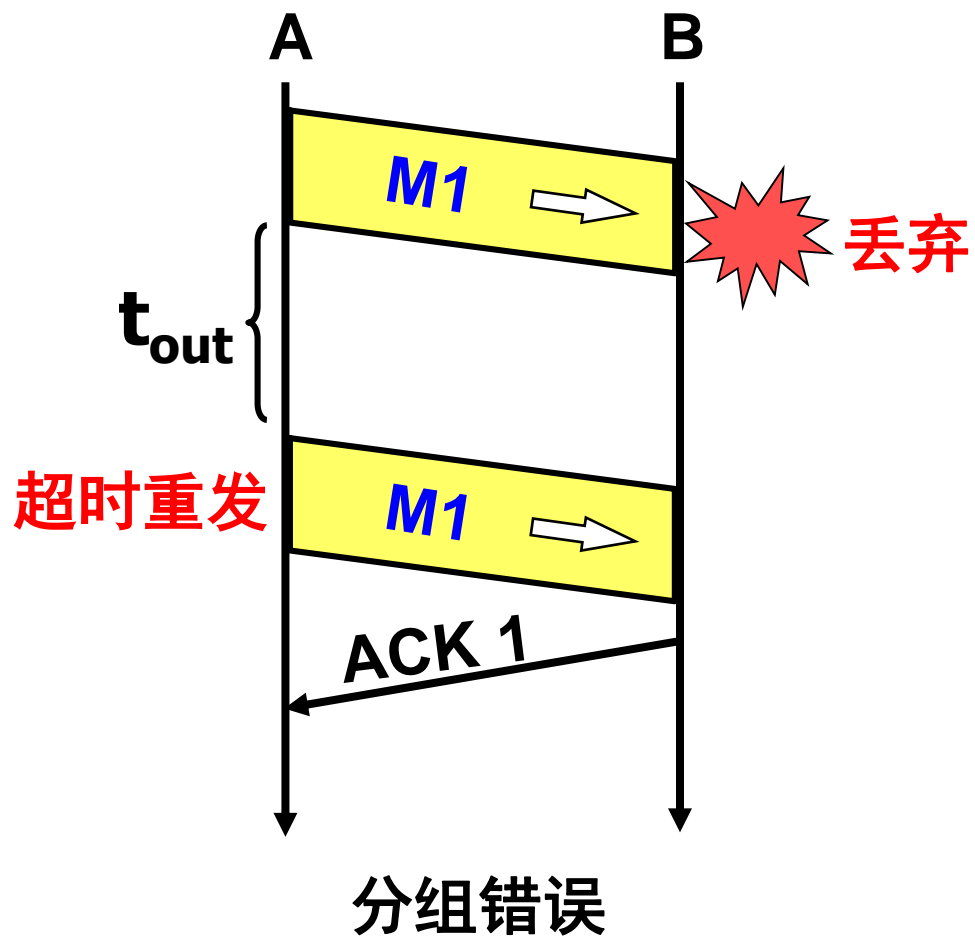


## 2. 出现差错



- 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



# 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。

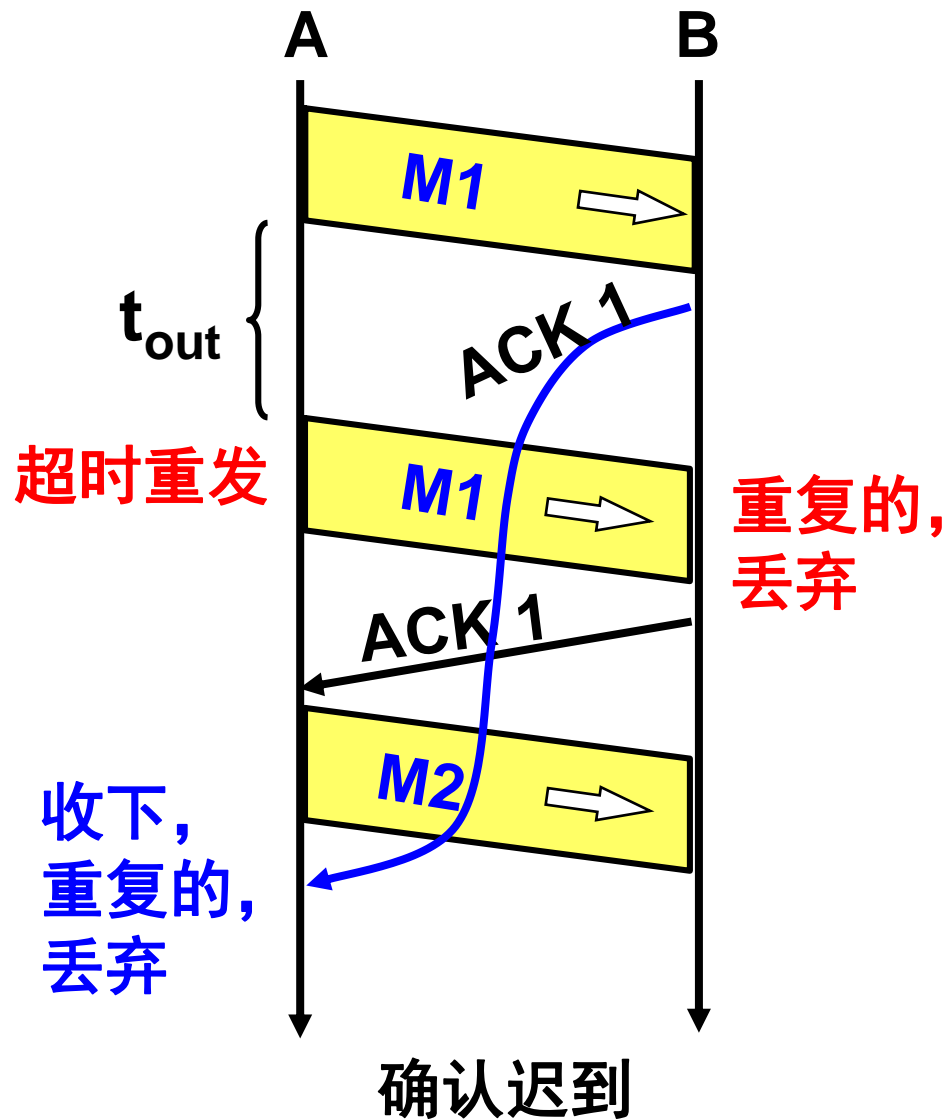
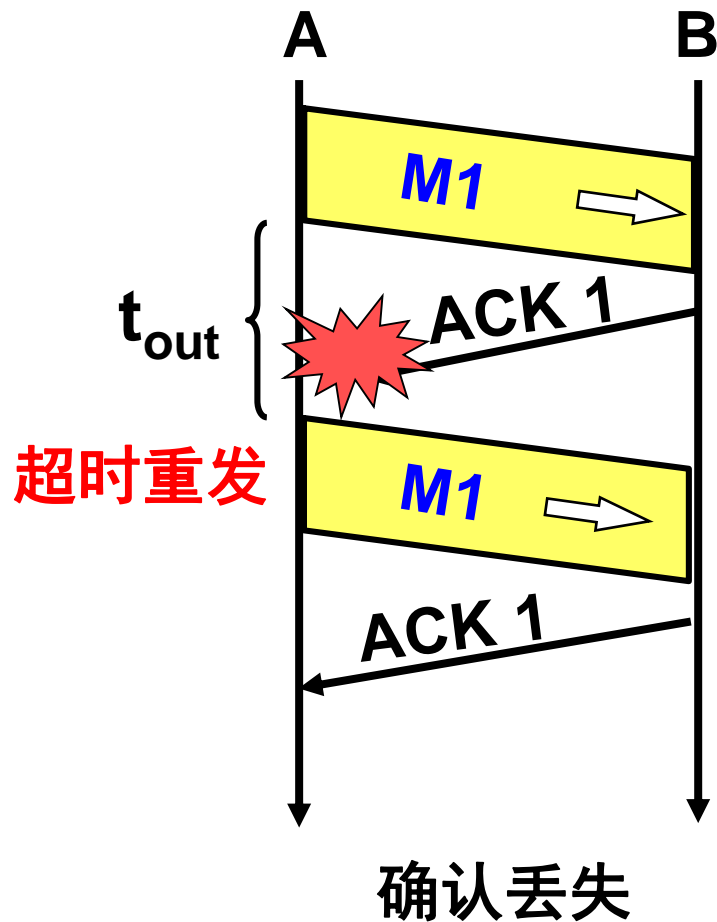
# 3. 确认丢失和确认迟到



## ■ 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



# 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该多长？***
- 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

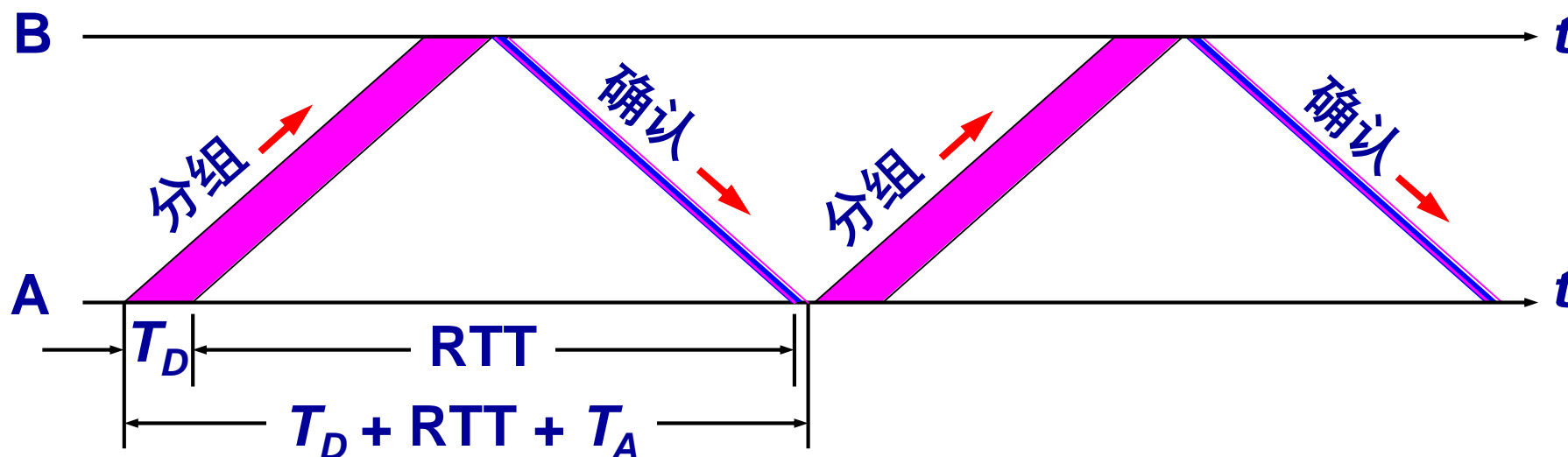


- 使用上述的确认和重传机制，我们就可以在不可靠的传输网络上实现可靠的通信。
- 像上述的这种可靠传输协议常称为**自动重传请求 ARQ** (Automatic Repeat reQuest)。意思是重传的请求是自动进行的，接收方不需要请求发送方重传某个出错的分组。

## 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

$$\text{信道利用率 } U = \frac{T_D}{T_D + RTT + T_A} \quad (5-3)$$



## 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。

**如何提高信道利用率？**

# 流水线传输

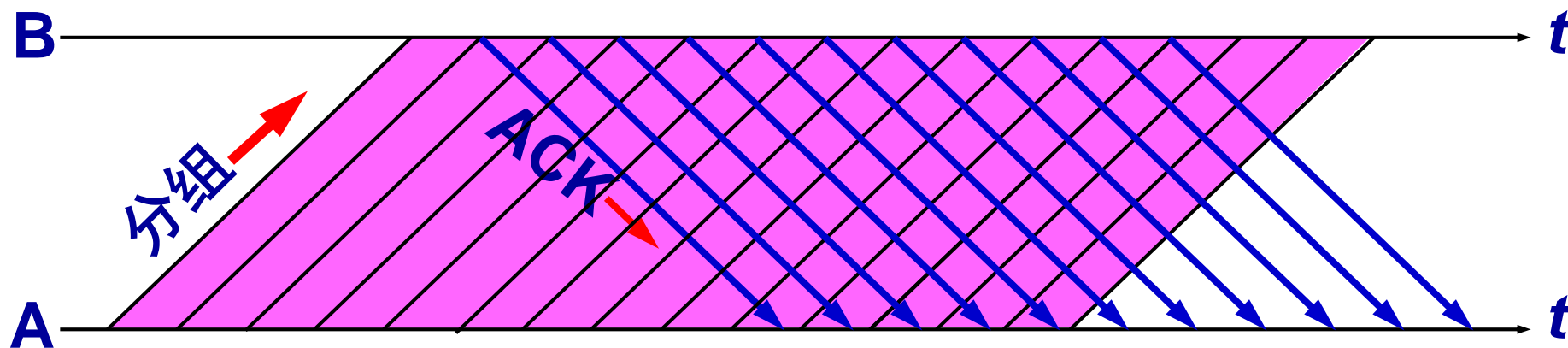


- **流水线传输**就是发送方可连续发送多个分组，不必每发完一个分组就停顿下来等待对方的确认。这样可使信道上一一直有数据不间断地传送。
- 由于信道上一一直有数据不间断地传送，这种传输方式可获得很高的信道利用率。

# 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

一次发送多少分组？ → 滑动窗口

## 5.4.2 连续 ARQ 协议

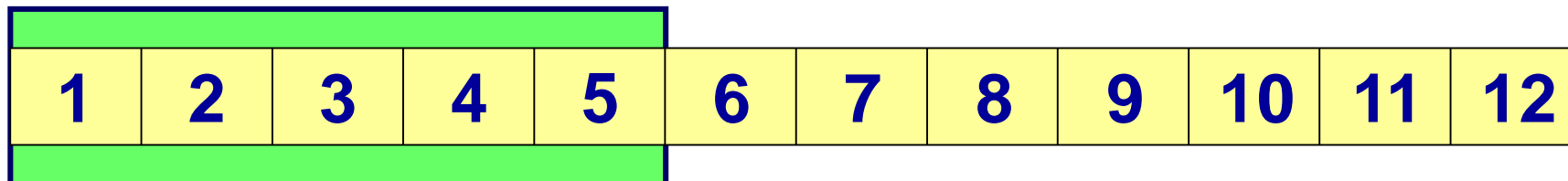


- 滑动窗口协议比较复杂，是 TCP 协议的精髓所在。
- 发送方维持的**发送窗口**，它的意义是：**位于发送窗口内的分组都可连续发送出去，而不需要等待对方的确认**。这样，信道利用率就提高了。
- 连续 ARQ 协议规定，**发送方每收到一个确认，就把发送窗口向前滑动一个分组的位置**。

## 5.4.2 连续ARQ协议

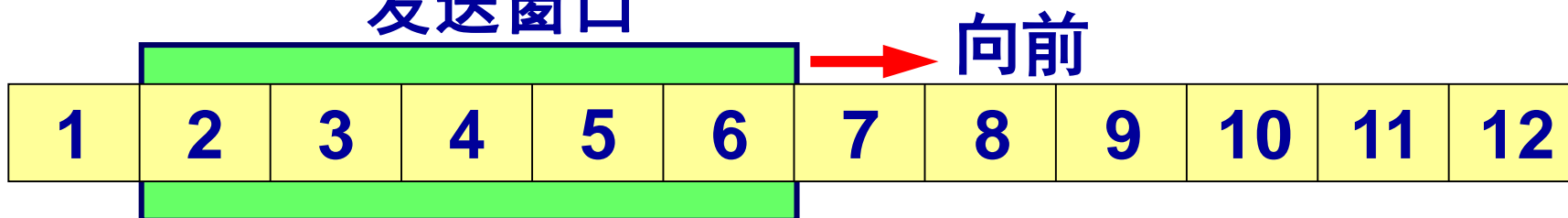


发送窗口



(a) 发送方维持发送窗口（发送窗口是 5）

发送窗口



(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

# 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。

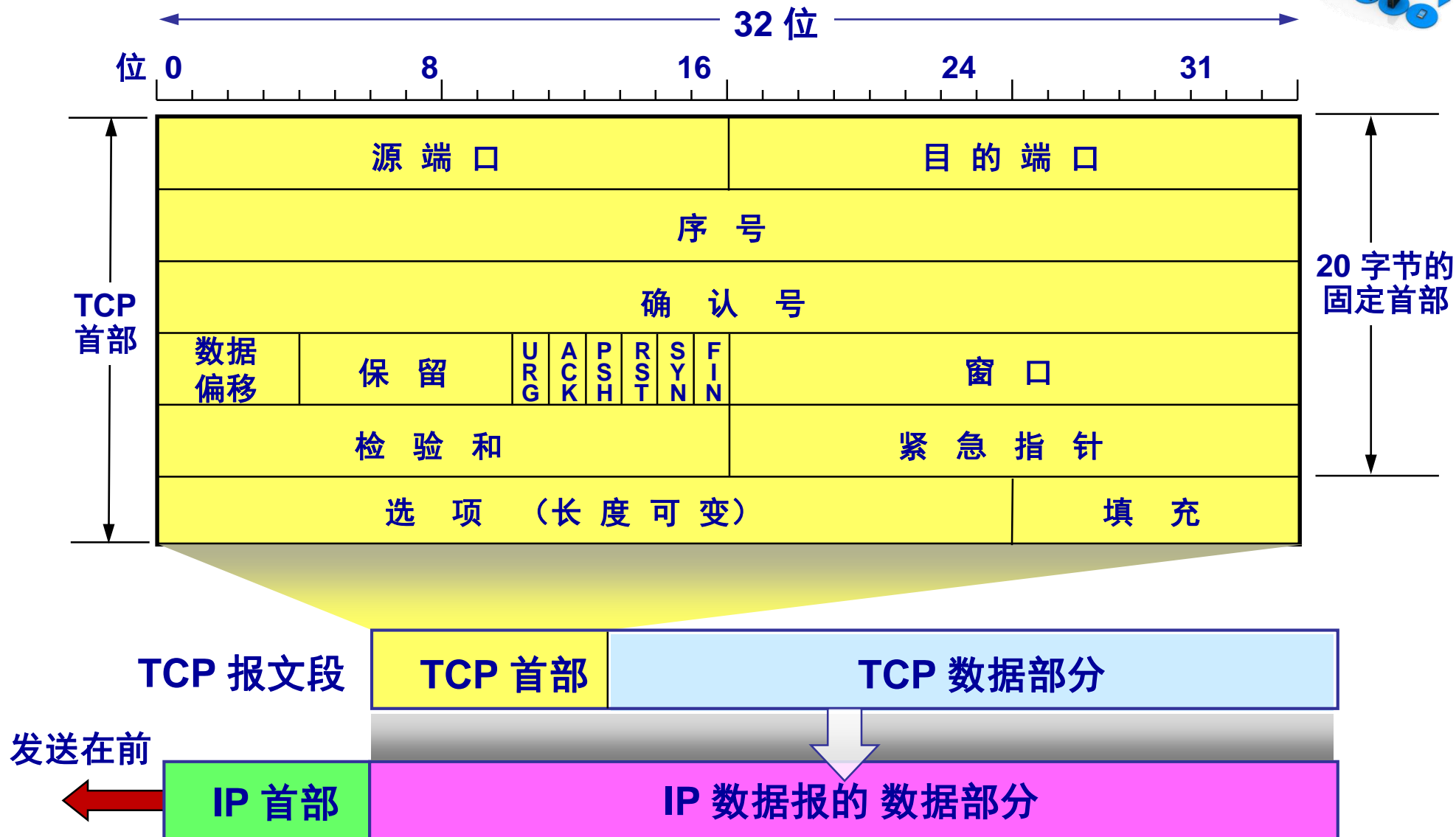
## 5.5 TCP 报文段的首部格式

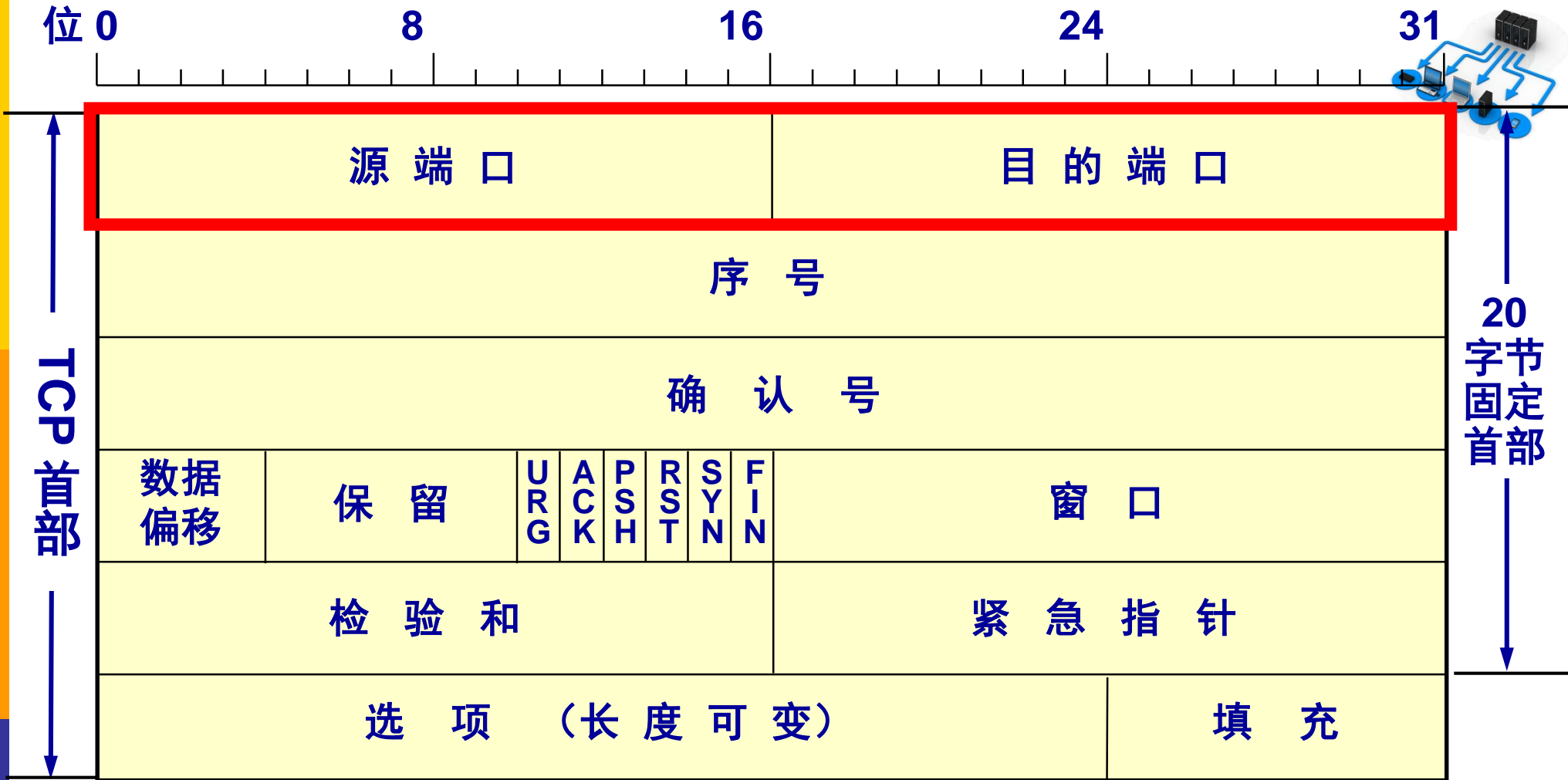


- TCP 虽然是面向字节流的，但 TCP 传送的数据单元却是报文段。
- 一个 TCP 报文段分为**首部和数据**两部分，而 TCP 的全部功能都体现在它首部中各字段的作用。
- TCP 报文段首部的前 **20 个字节**是固定的，后面有  $4n$  字节是根据需要而增加的选项 ( $n$  是整数)。**因此 TCP 首部的最小长度是 20 字节。**



# TCP 报文段的首部格式

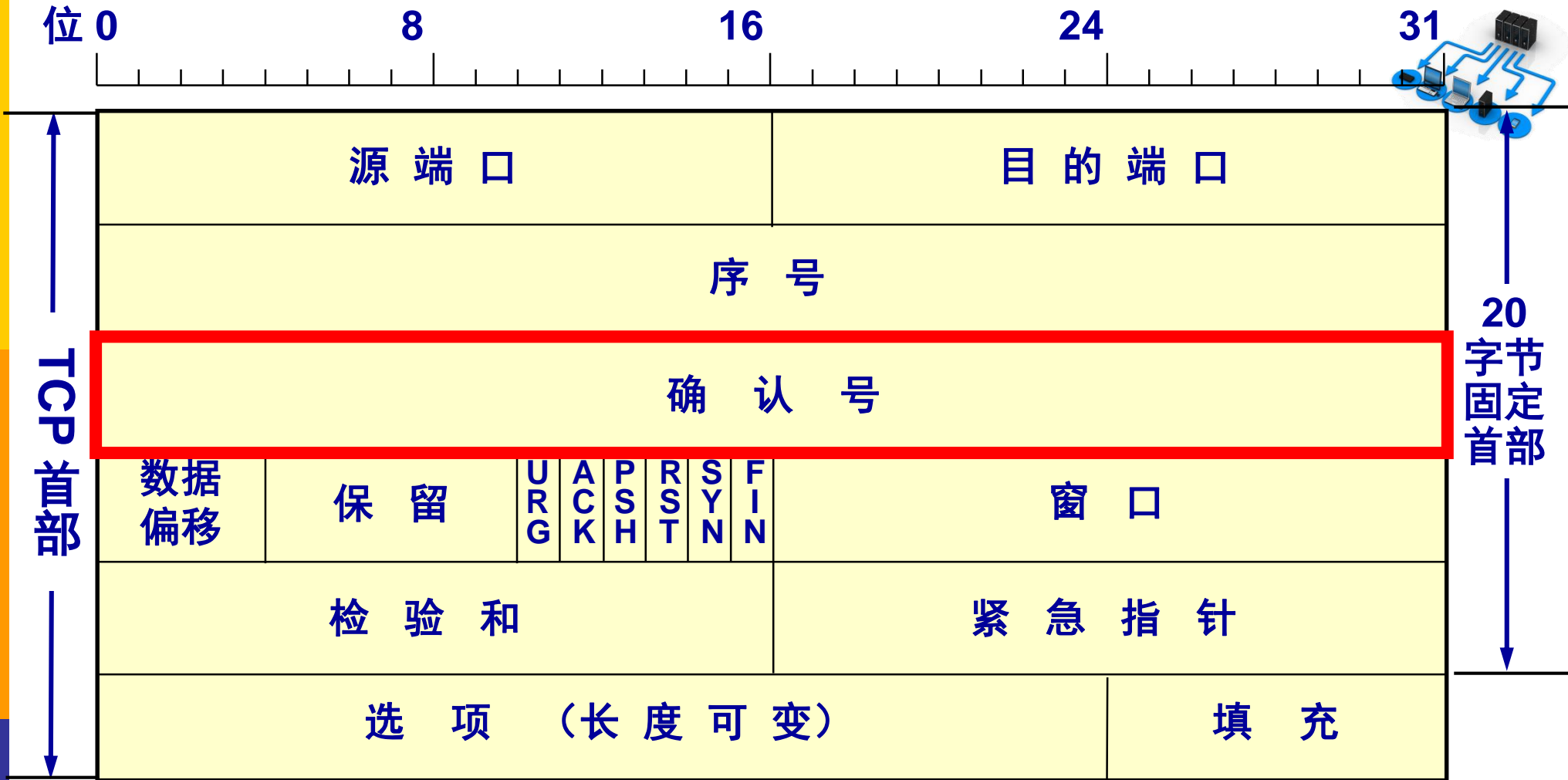




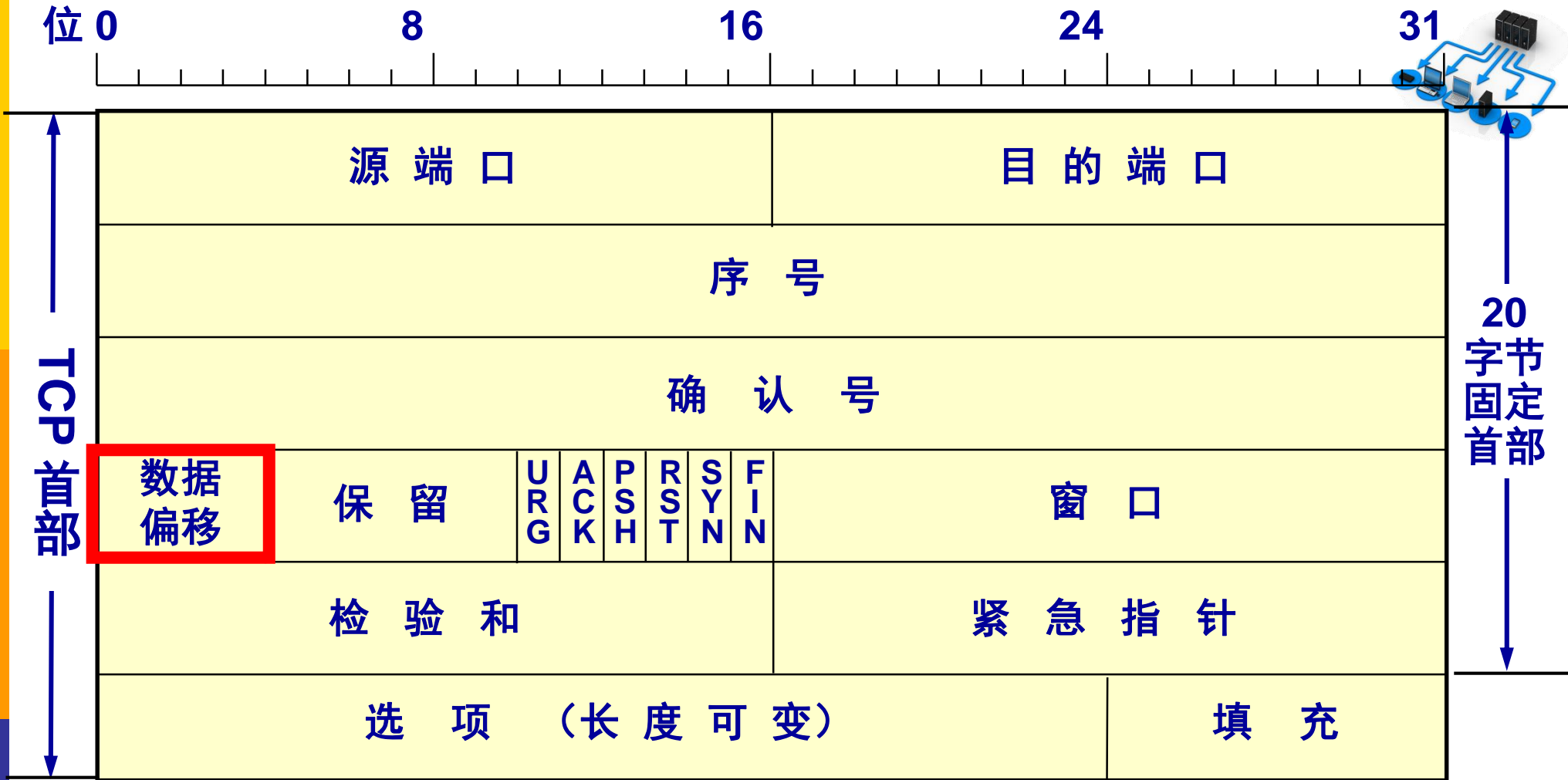
源端口和目的端口字段——各占 2 字节。端口是运输层与应用层的服务接口。运输层的复用和分用功能都要通过端口才能实现。



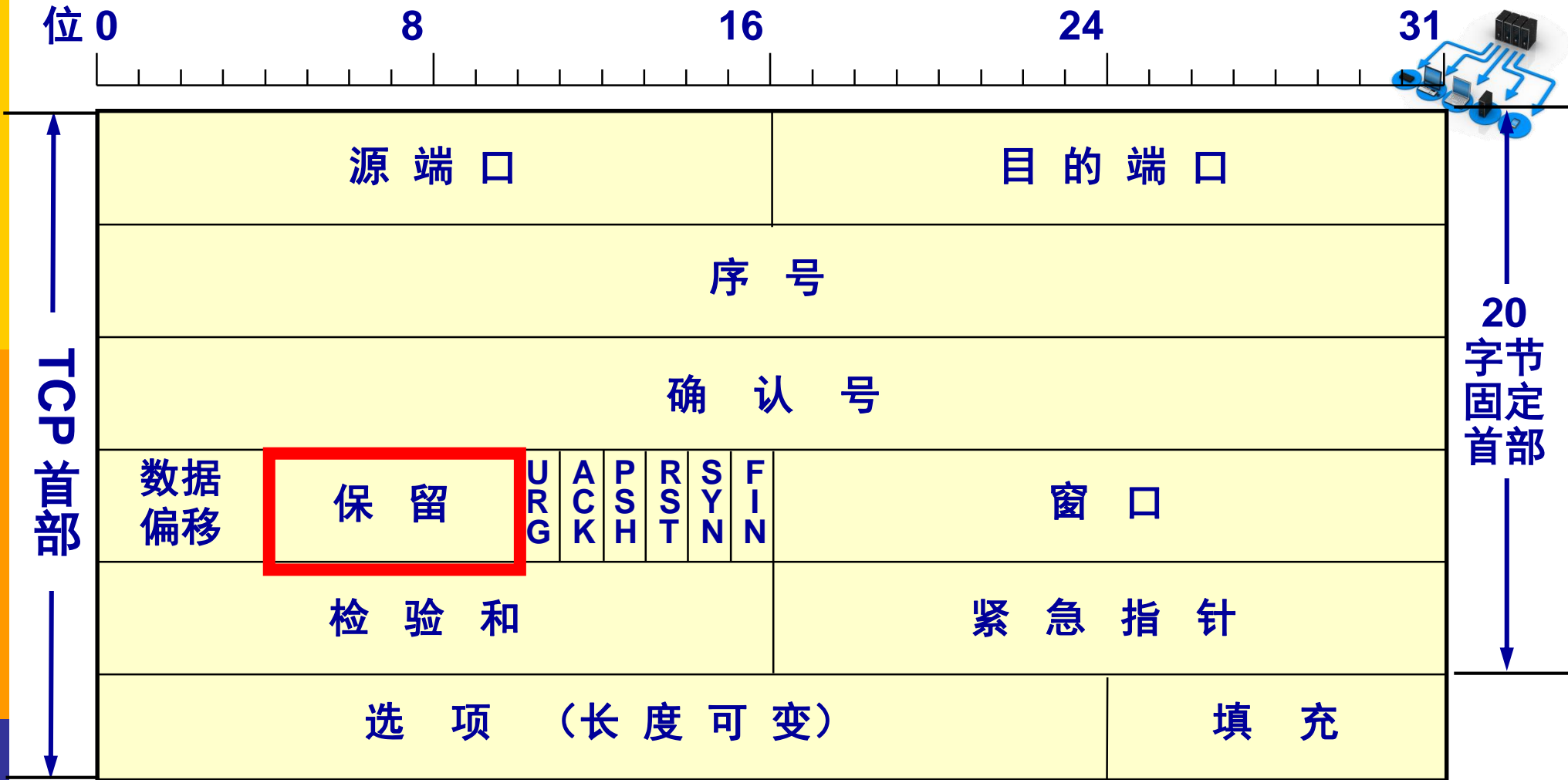
序号字段——占 4 字节。TCP 连接中传送的数据流中的每一个字节都编上一个序号。序号字段的值则指的是本报文段所发送的数据的第一个字节的序号。



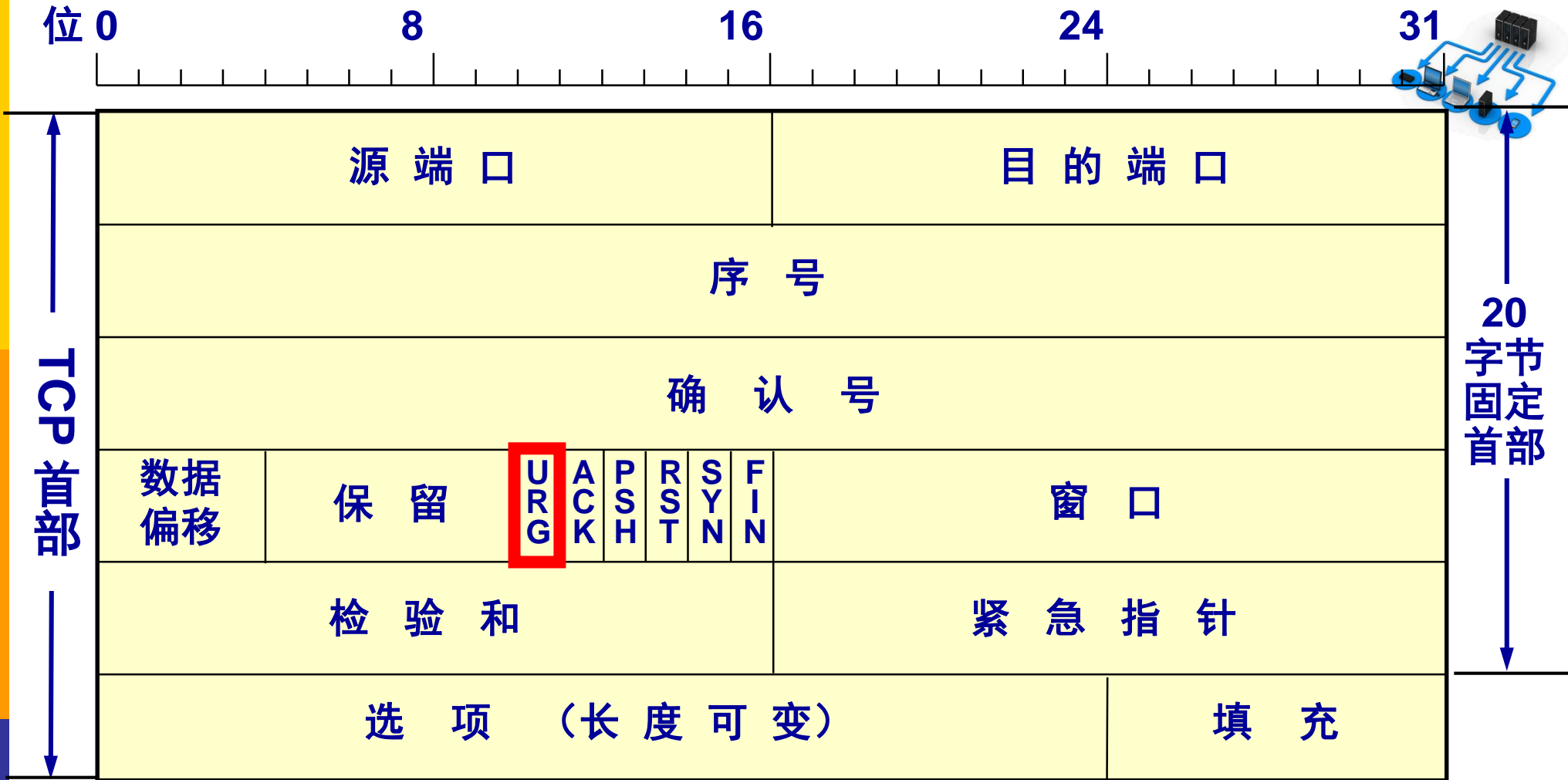
确认号字段——占 4 字节，是期望收到对方的下一个报文段的数据的第一个字节的序号。



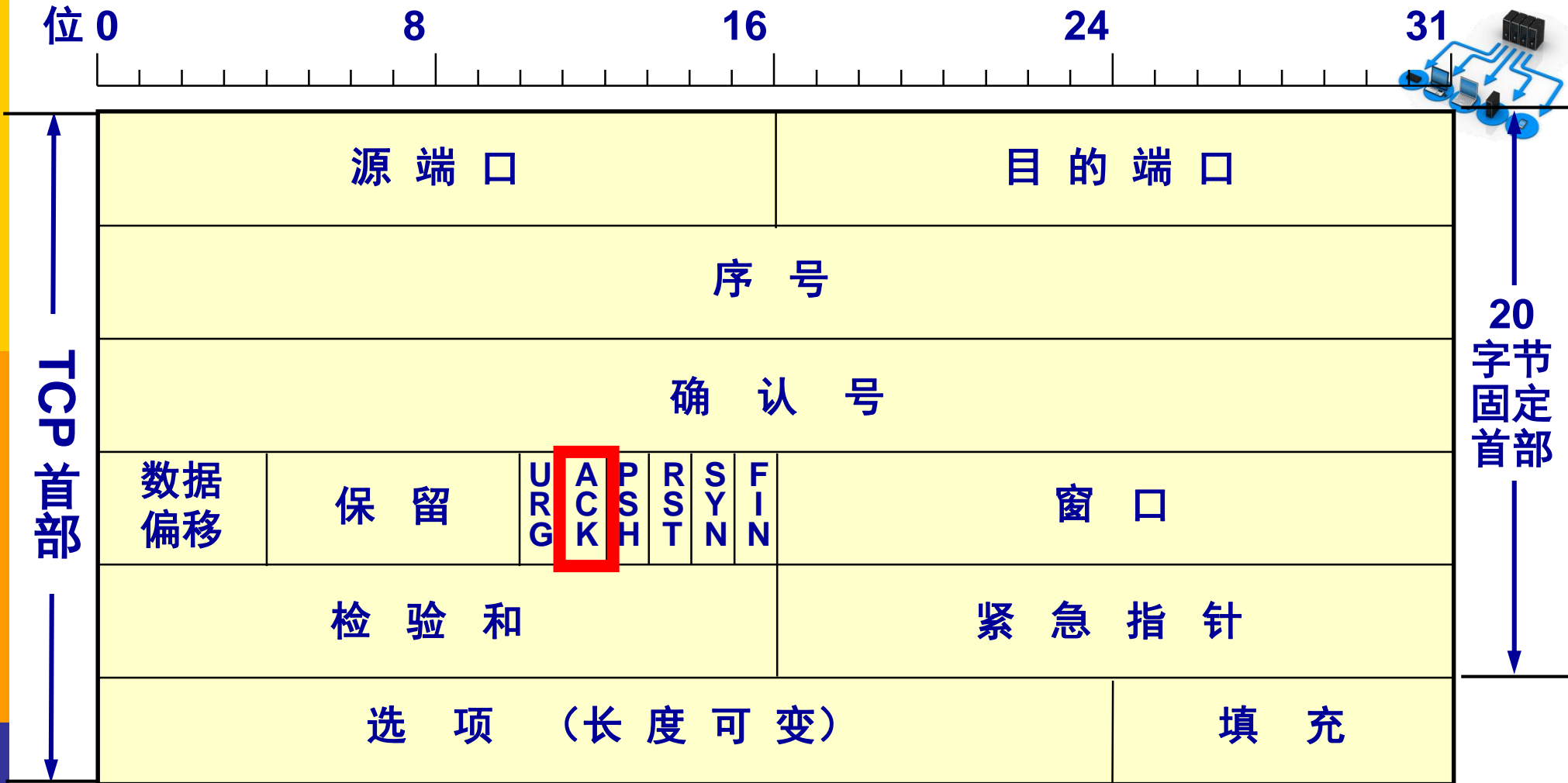
数据偏移（即首部长度的）——占 4 位，它指出 TCP 报文段的数据起始处距离 TCP 报文段的起始处有多远。“数据偏移”的单位是 32 位字（以 4 字节为计算单位）。



保留字段——占 6 位，保留为今后使用，但目前应置为 0。

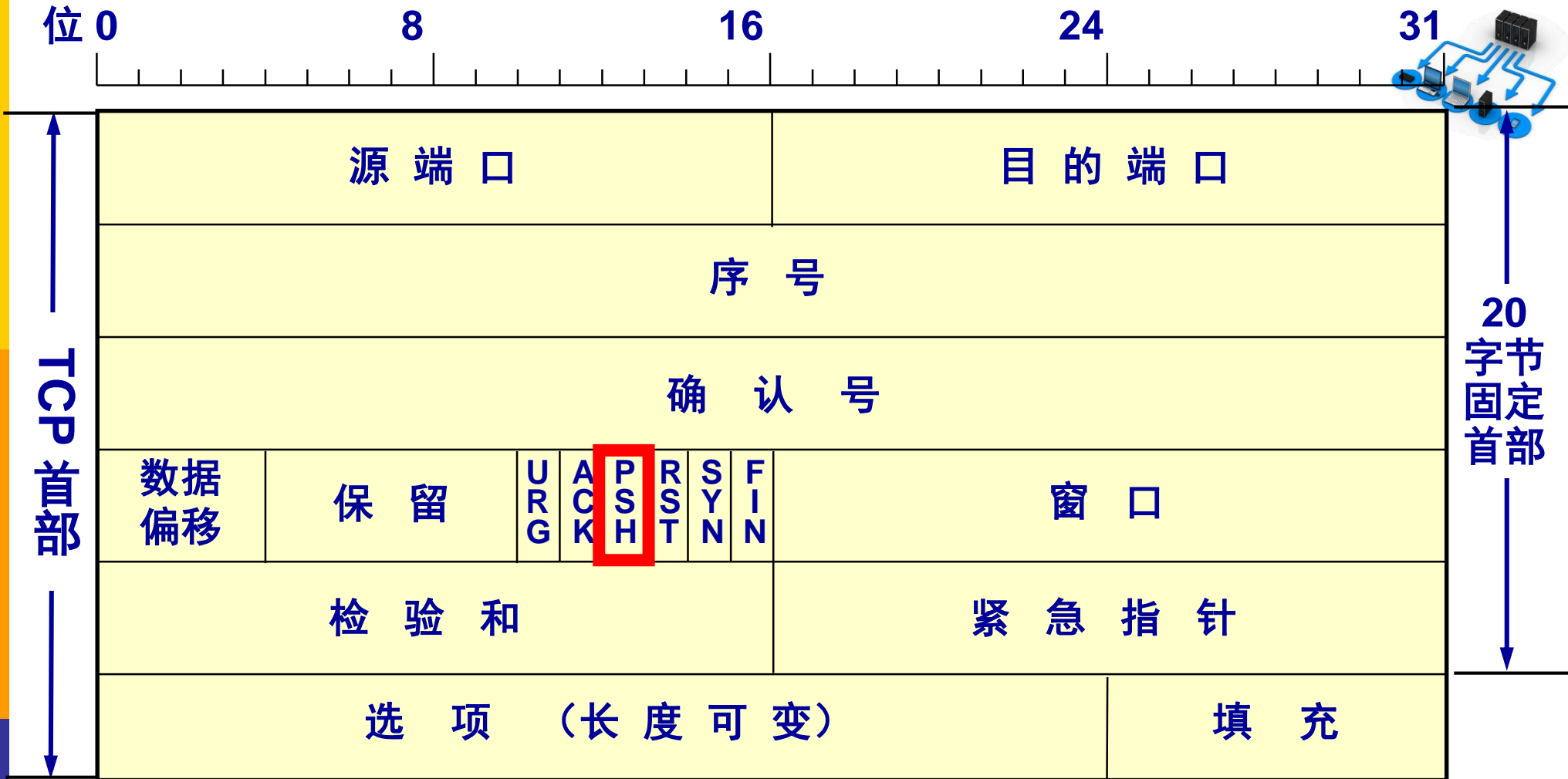


**紧急 URG** —— 当 **URG = 1** 时，表明紧急指针字段有效。它告诉系统此报文段中有紧急数据，应尽快传送(相当于高优先级的数据)。

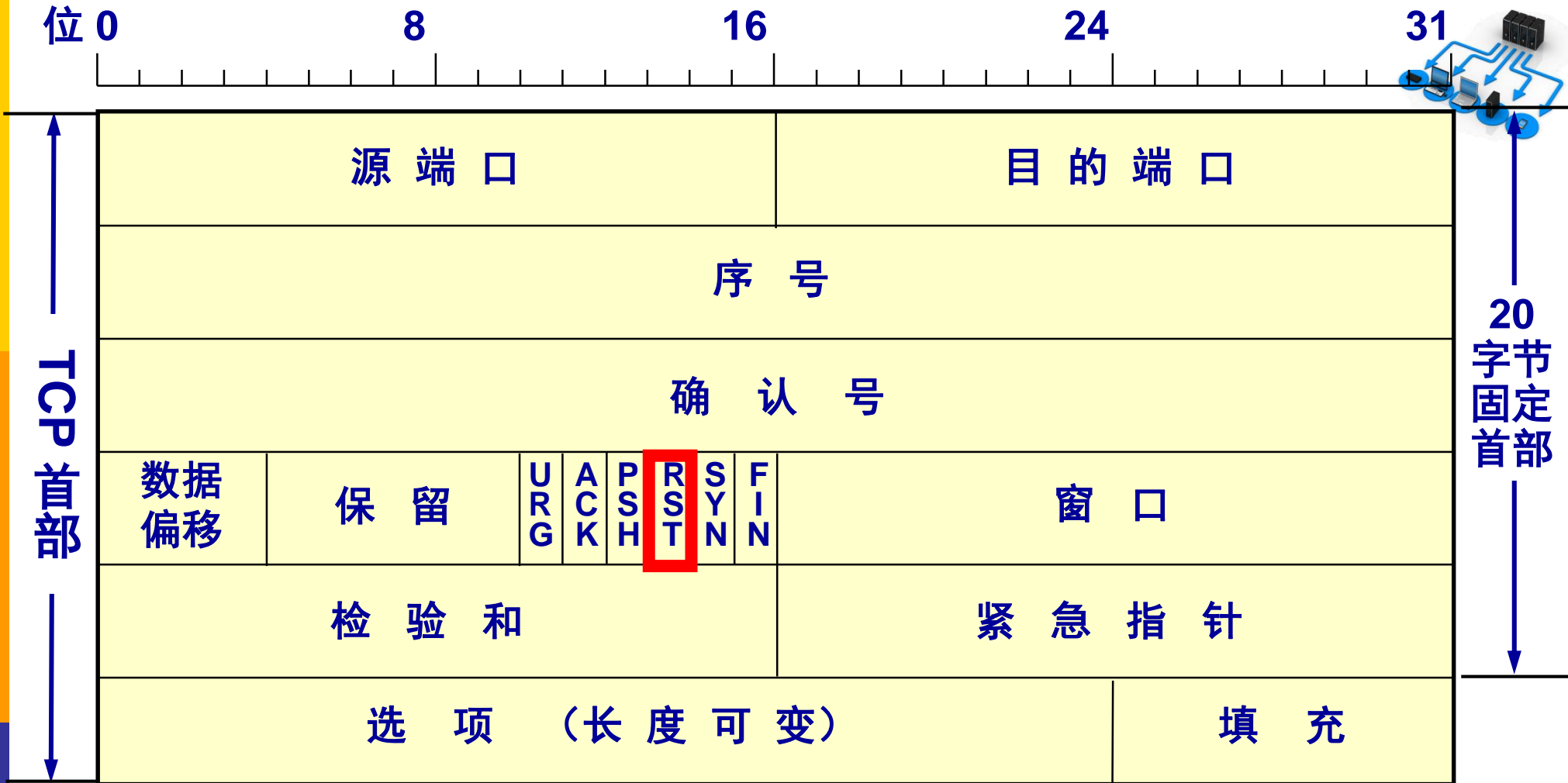


确认 ACK —— 只有当  $ACK = 1$  时确认号字段才有效。当  $ACK = 0$  时，确认号无效。

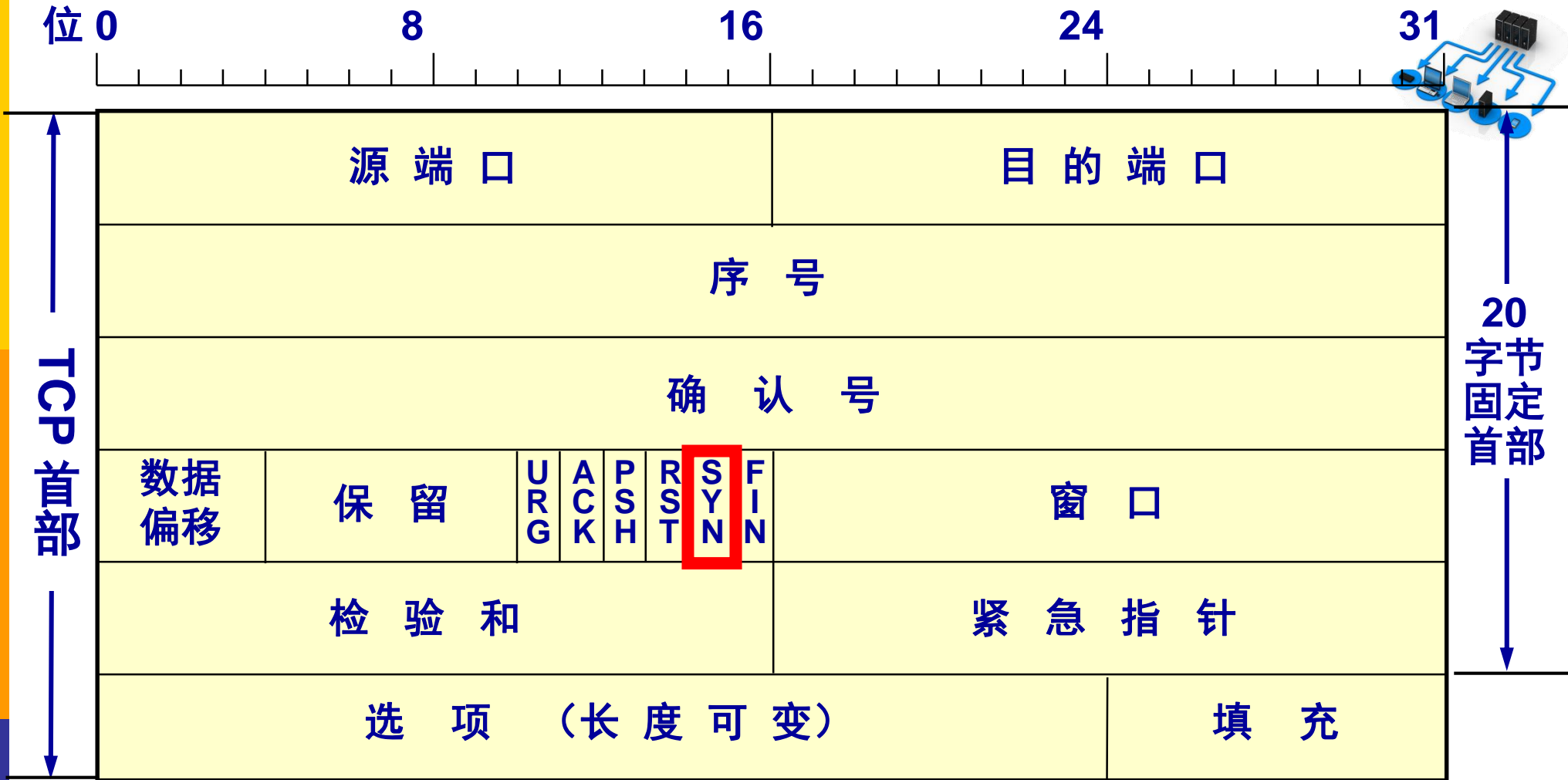




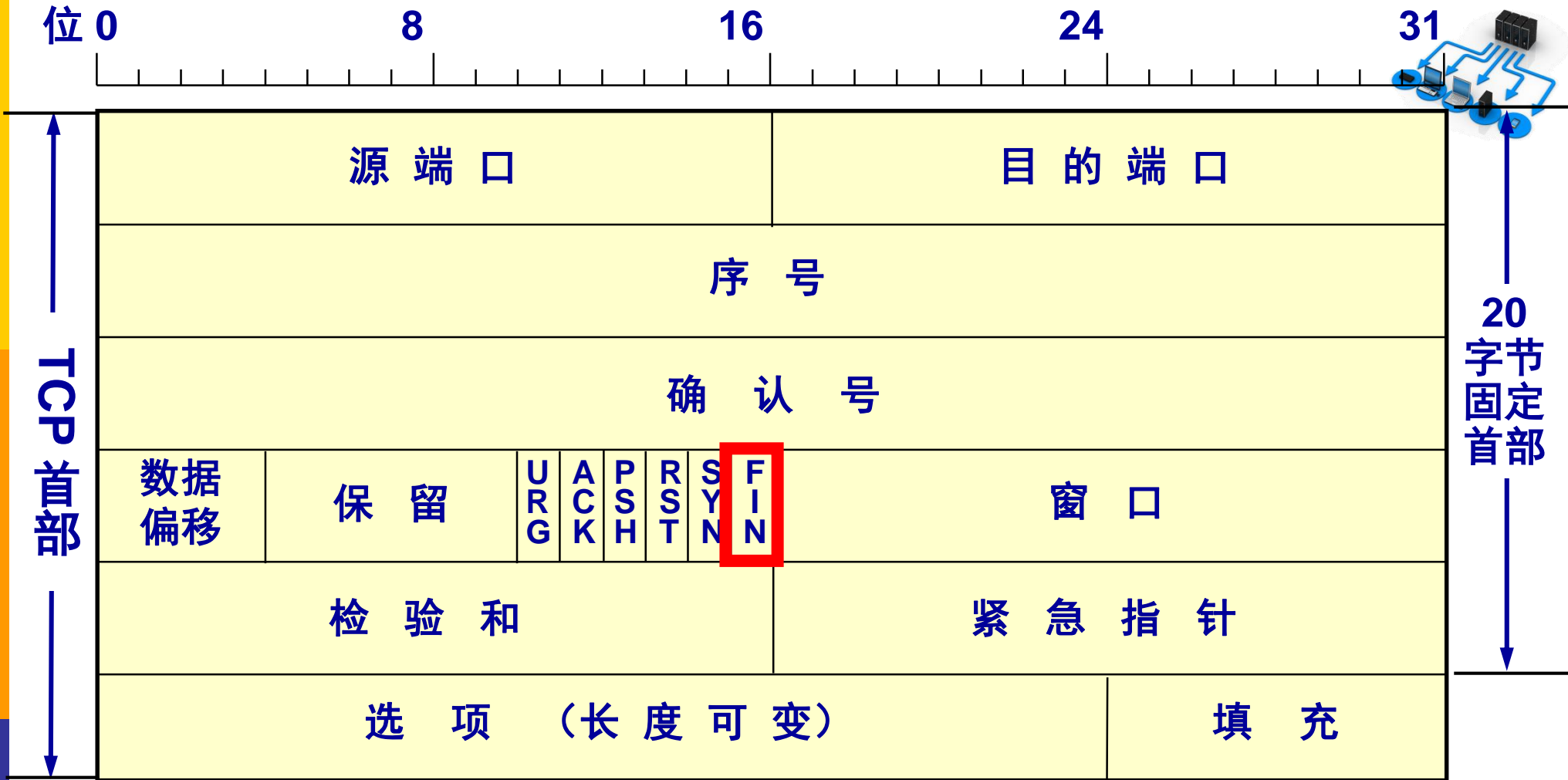
**推送 PSH (PuSH) —— 接收 TCP 收到 PSH = 1 的报文段，就尽快地交付接收应用进程，而不再等到整个缓存都填满了后再向上交付。**



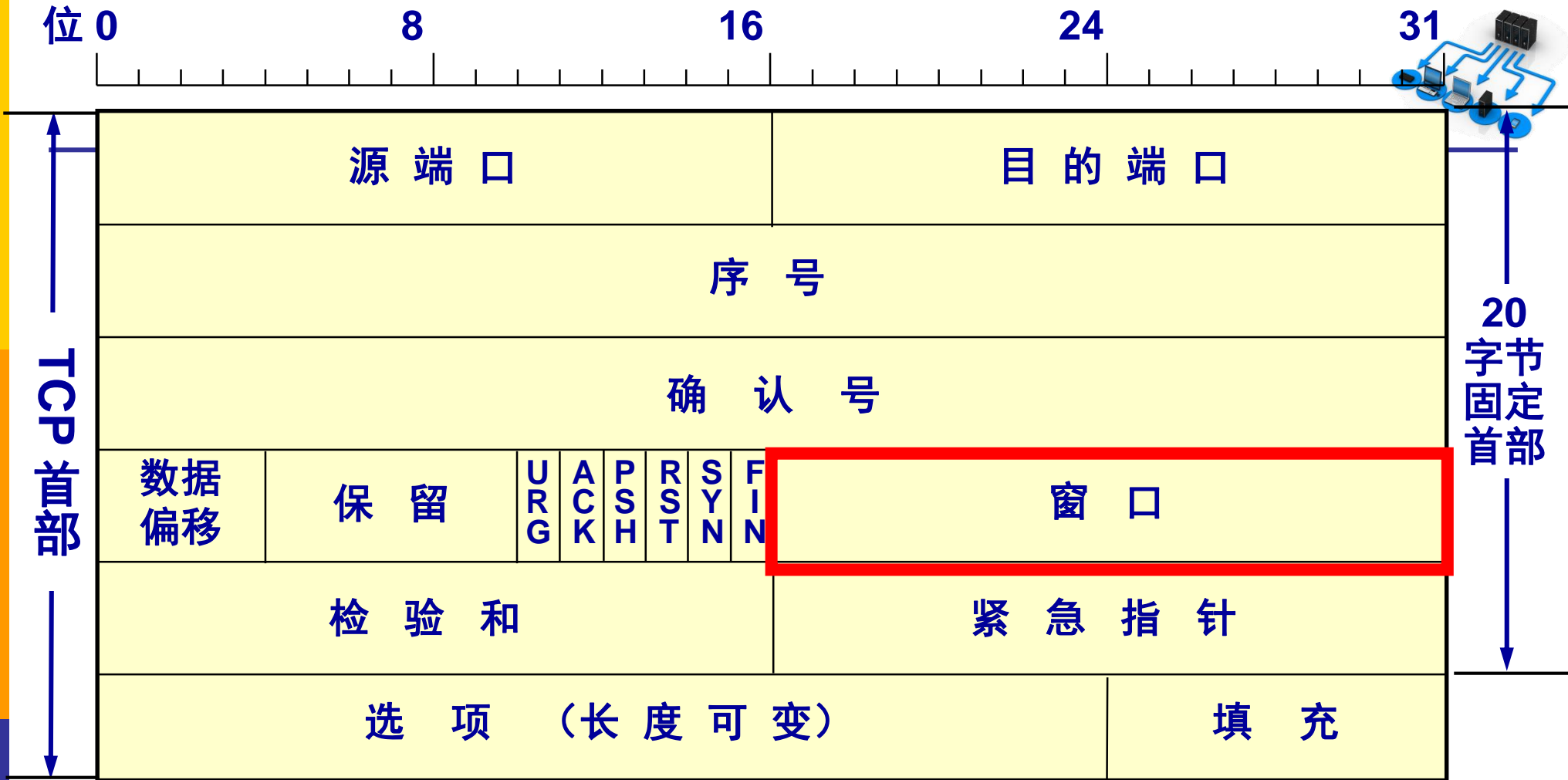
**复位 RST (ReSeT) ——** 当  $RST = 1$  时，表明 TCP 连接中出现严重差错（如由于主机崩溃或其他原因），必须释放连接，然后再重新建立运输连接。



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。



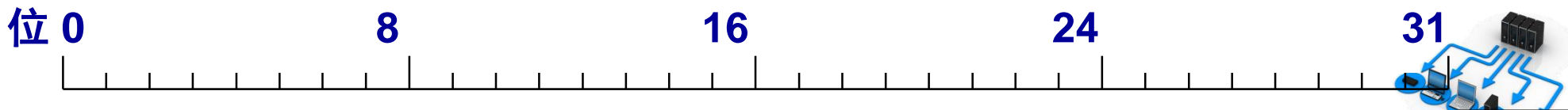
窗口字段——占 2 字节，用来让对方设置发送窗口的依据，单位为字节。



校验和 —— 占 2 字节。校验和字段检验的范围包括**首部**和**数据**这两部分。在计算校验和时，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）。

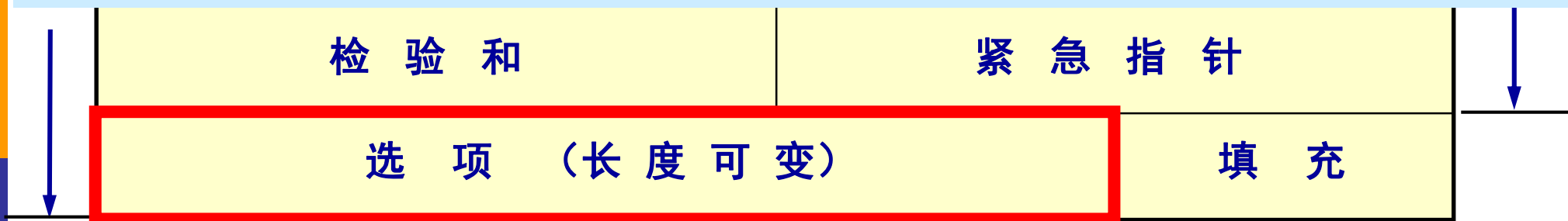


## MSS (Maximum Segment Size)

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。

所以，MSS是“TCP 报文段长度减去 TCP 首部长度的”。



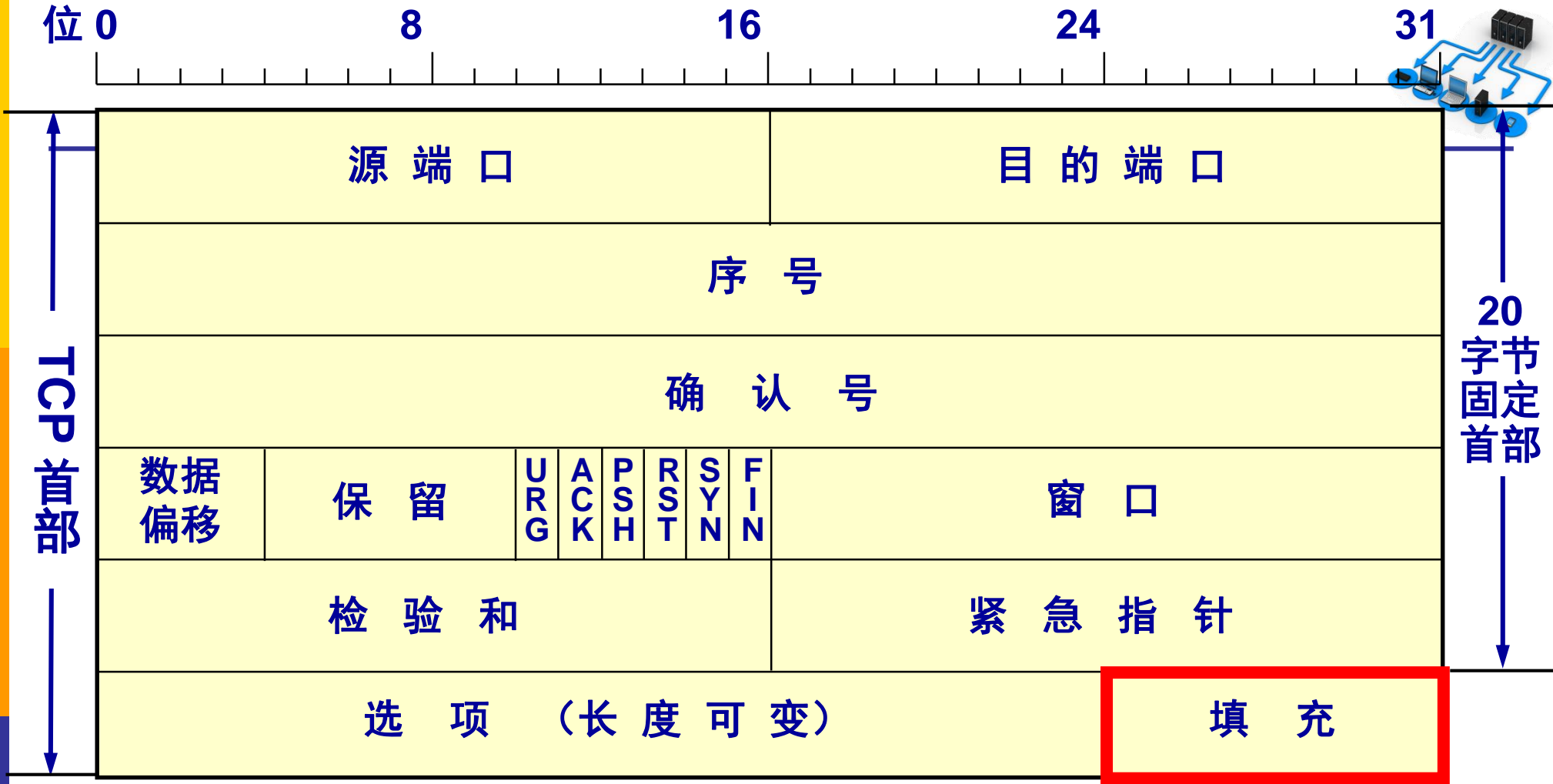
选项字段 —— 长度可变。TCP 最初只规定了一种选项，即**最大报文段长度 MSS**。MSS 告诉对方 TCP：“我的缓存所能接收的报文段的数据字段的最大长度是 MSS 个字节。”



# 其他选项



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

# 第 5 章 运输层（三）



# 复习



## ■ 5.4 可靠传输的工作原理

- 停止等待协议

- 连续ARQ协议



## ■ 5.5 TCP 报文段的首部格式

# 本讲重点



## ■ 5.6 TCP 可靠传输的实现

- 以字节为单位的滑动窗口
- 超时重传时间的选择
- 选择确认 SACK



## ■ 5.7 TCP 的流量控制



# 5.6 TCP 可靠传输的实现

---



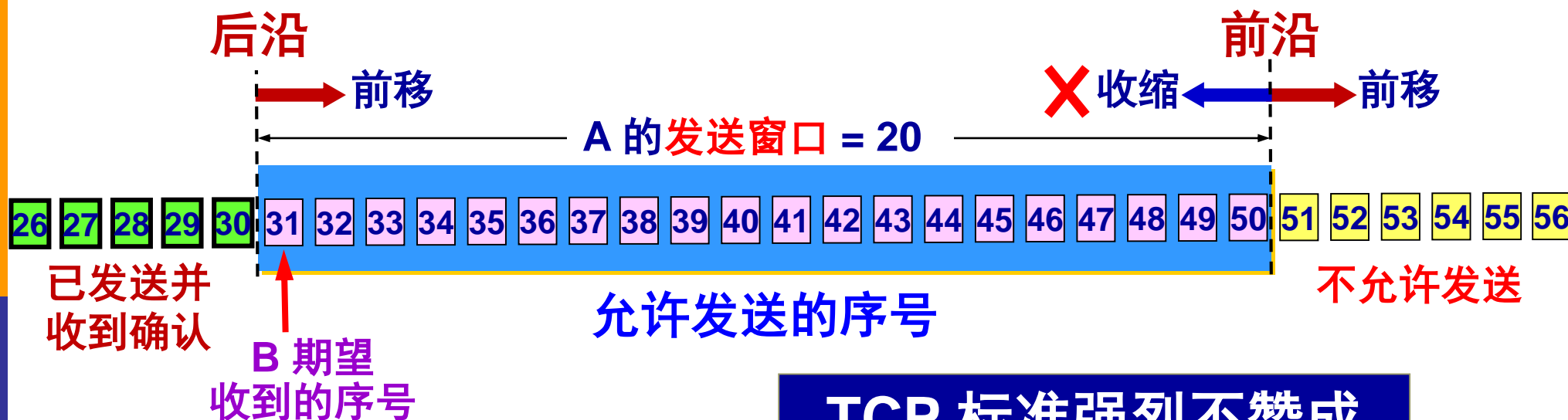
- 5.6.1 以字节为单位的滑动窗口
- 5.6.2 超时重传时间的选择
- 5.6.3 选择确认 SACK

## 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。

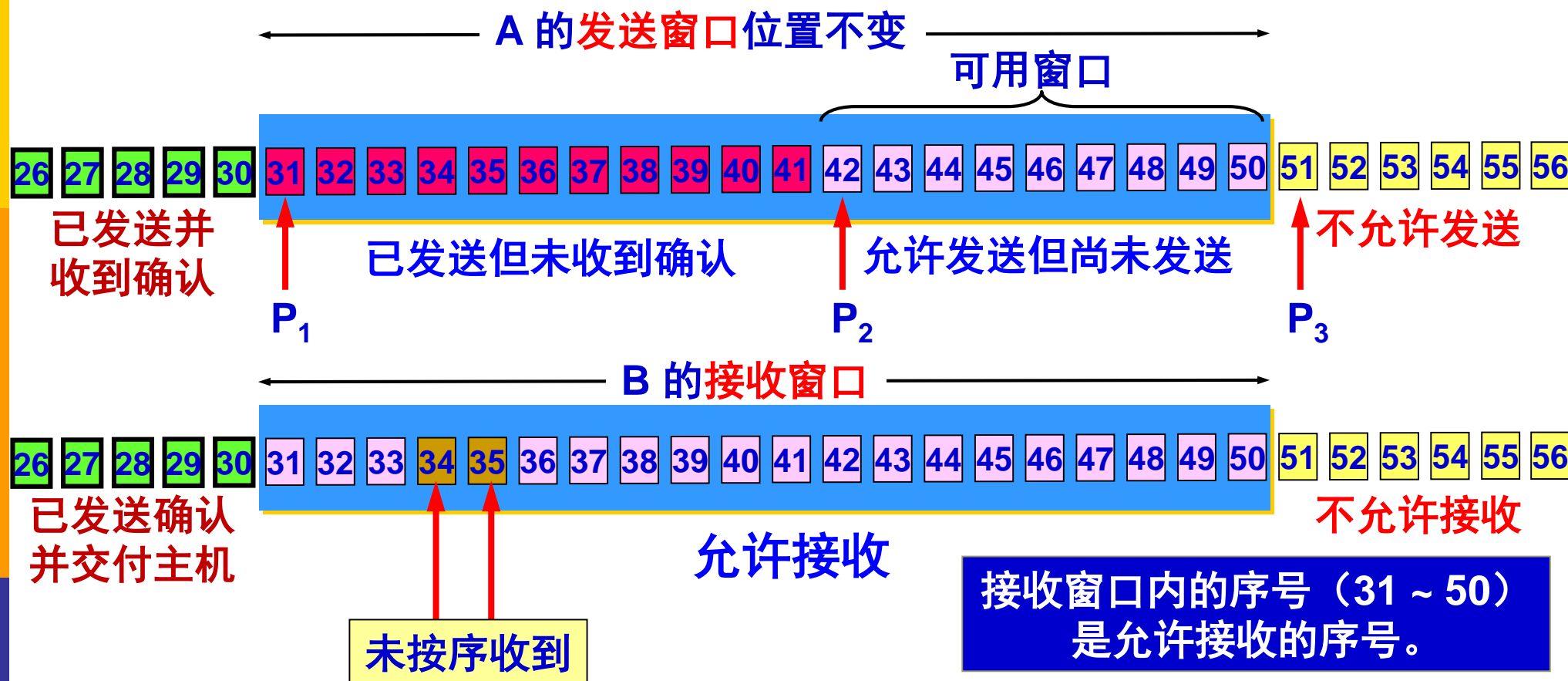
- 根据 B 给出的窗口值，A 构造出自己的发送窗口。
- 发送窗口表示：在没有收到 B 的确认的情况下，A 可以连续把窗口内的数据都发送出去。
- 发送窗口里面的序号表示允许发送的序号。
- 显然，窗口越大，发送方就可以在收到对方确认之前连续发送更多的数据，因而可能获得更高的传输效率。



**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



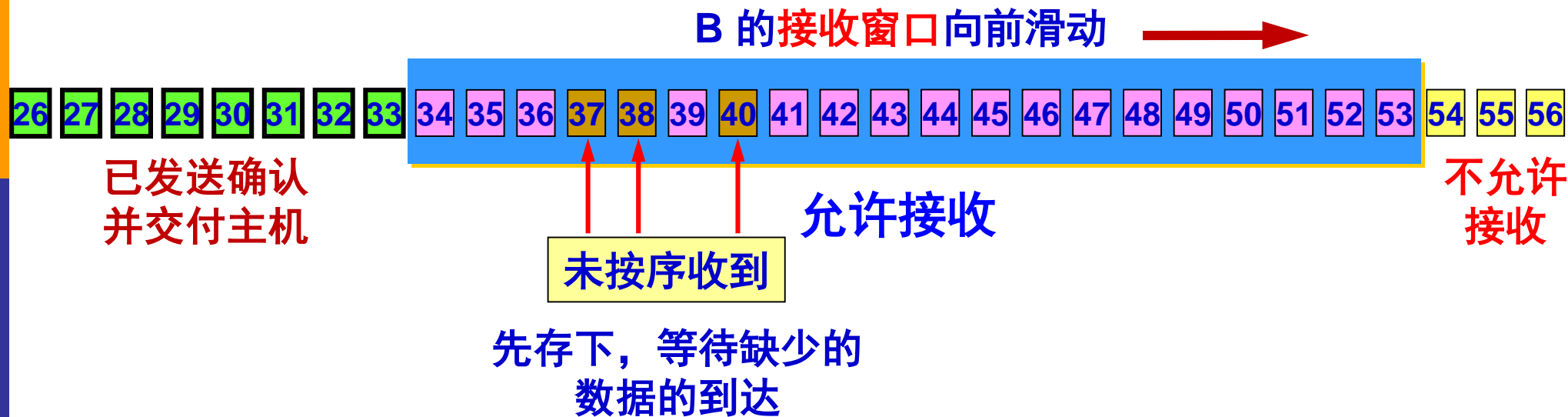
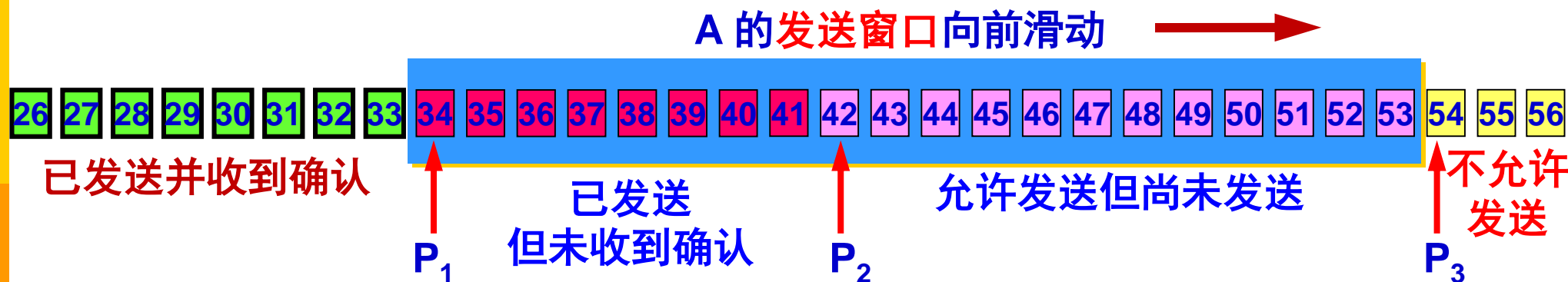
接收窗口内的序号 (31 ~ 50) 是允许接收的序号。

$P_3 - P_1 = A$  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)

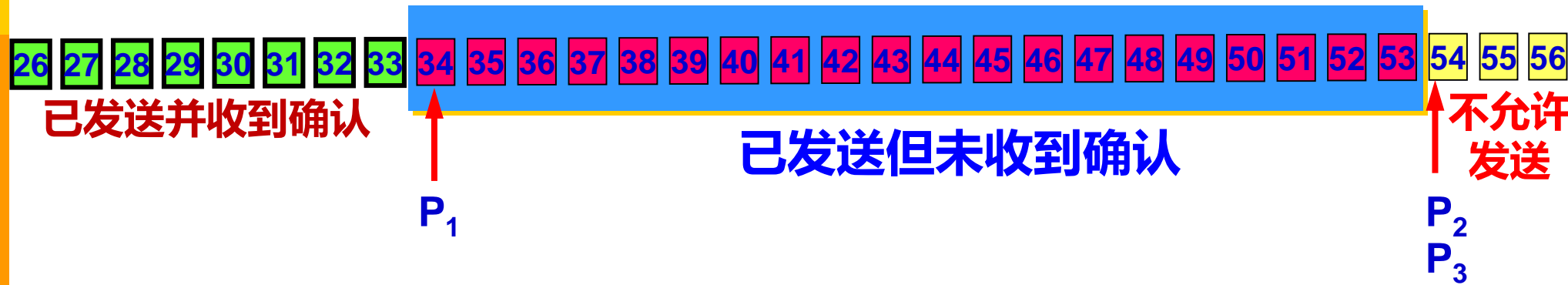
# A 收到新的确认号，发送窗口向前滑动



**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。**



**A 的发送窗口已满，有效窗口为零**



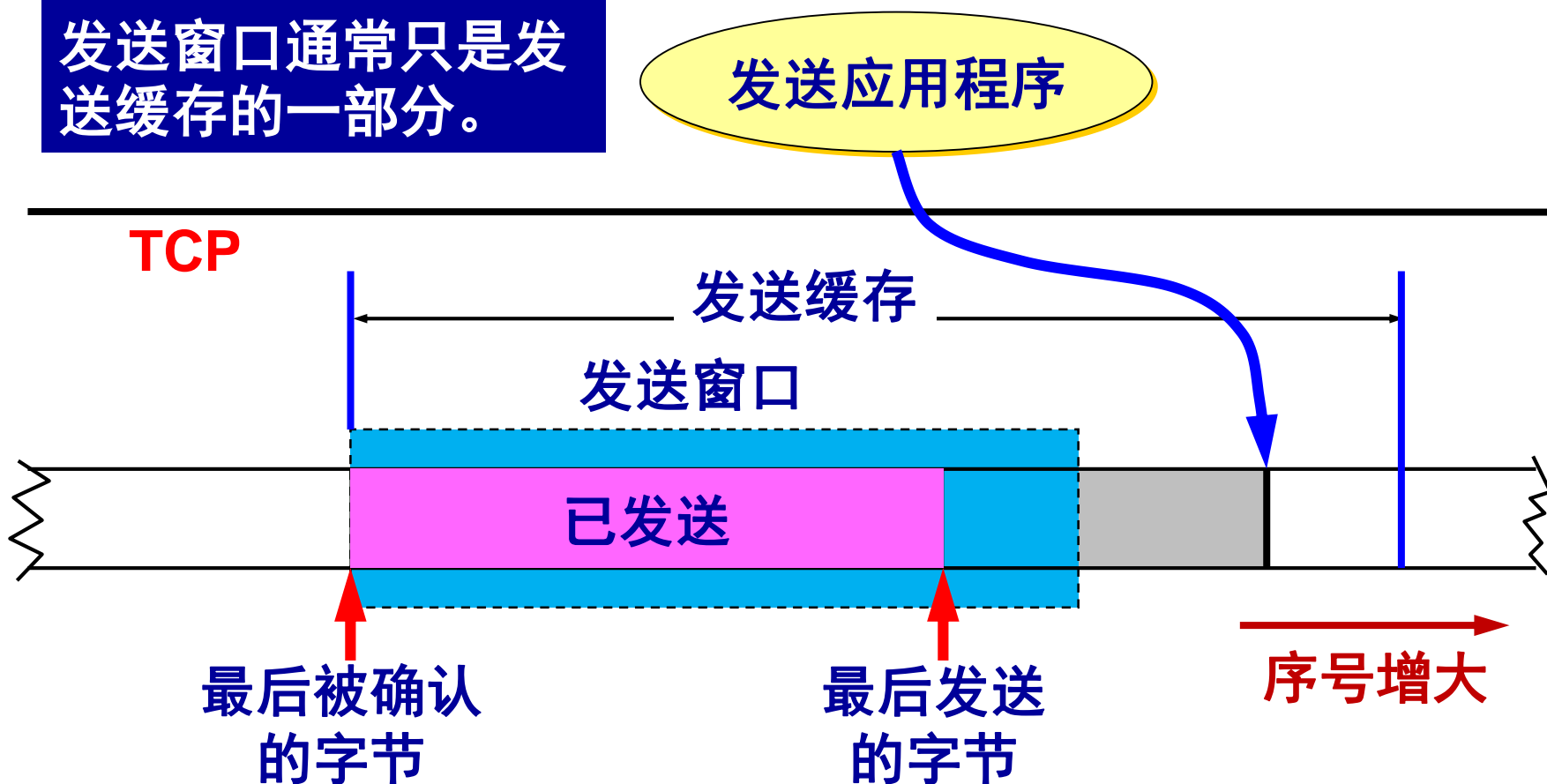
**发送窗口内的序号都属于已发送但未被确认**

# 发送缓存



发送方的应用进程把字节流写入 TCP 的发送缓存。

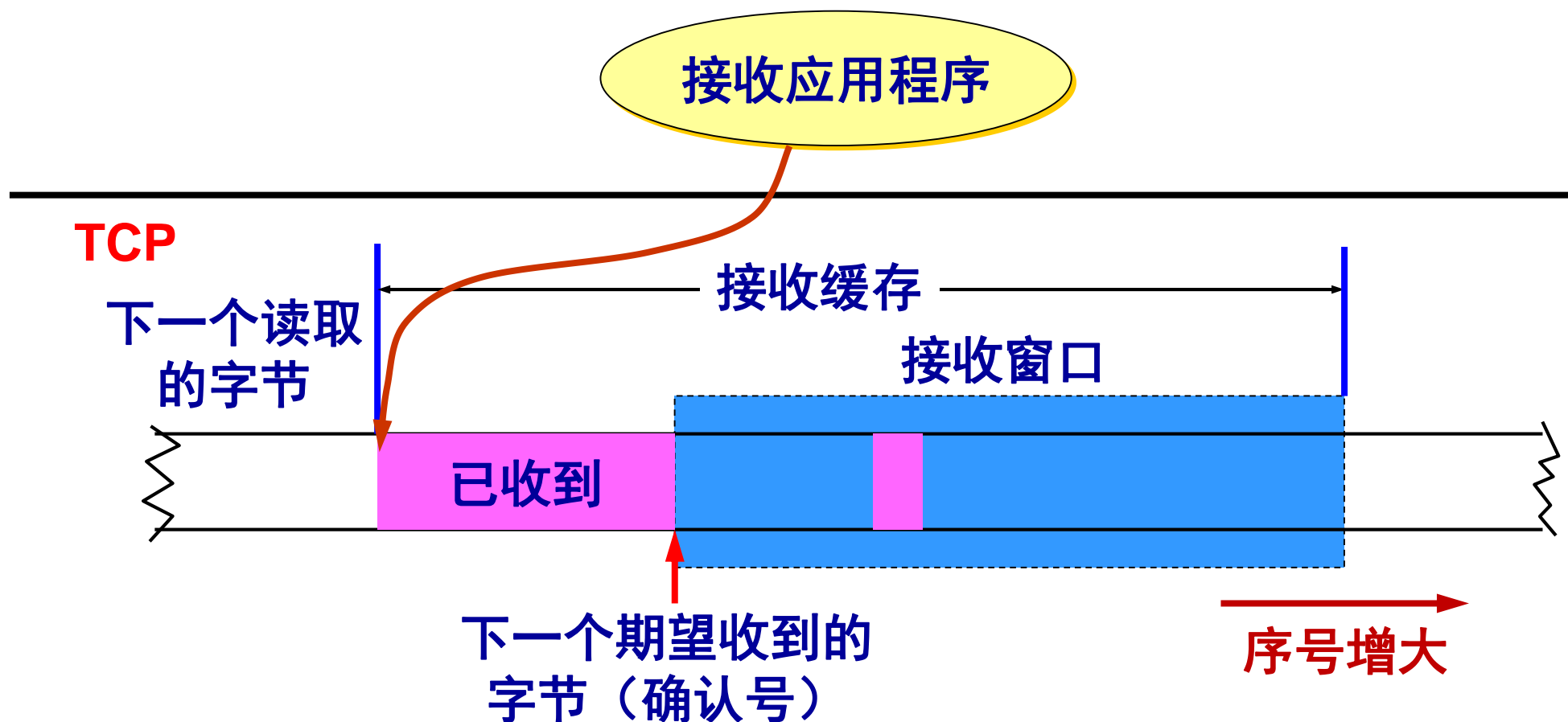
发送窗口通常只是发送缓存的一部分。



# 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



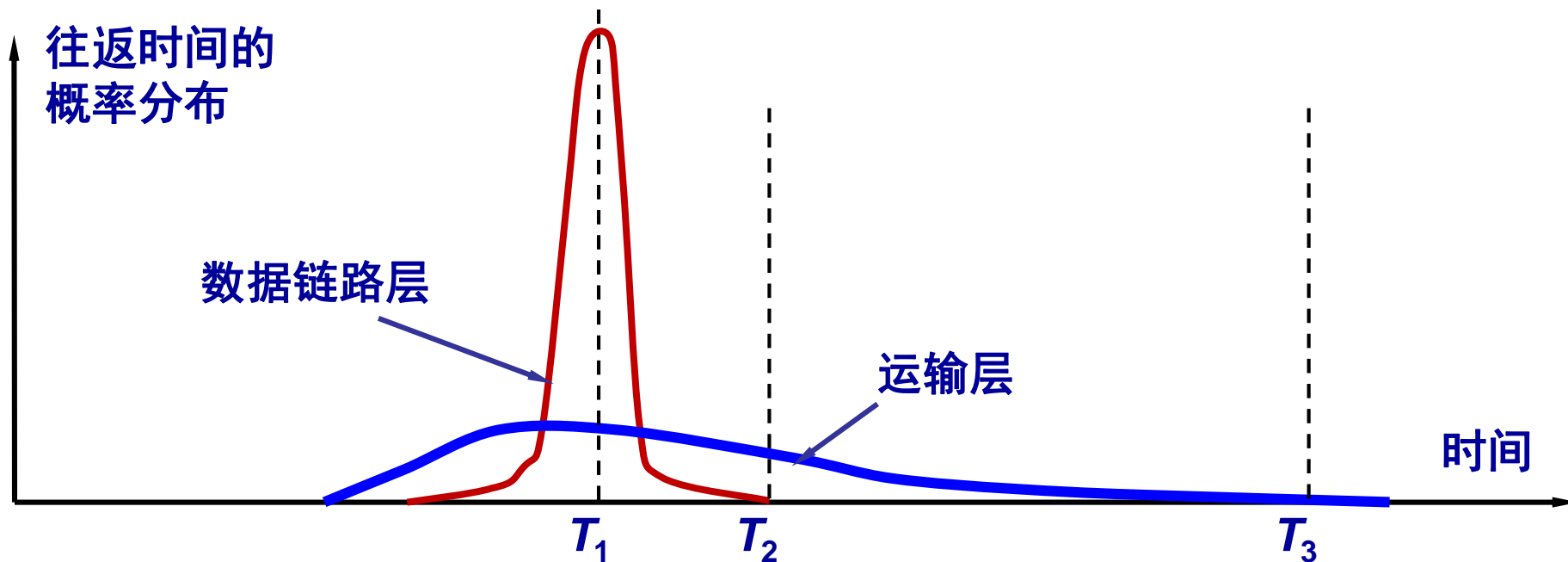
- 重传机制是 TCP 中最重要和最复杂的问题之一。
- TCP 每发送一个报文段，就对这个报文段设置一次计时器。
- 只要计时器设置的重传时间到但还没有收到确认，就要重传这一报文段。
- 重传时间的选择是 TCP 最复杂的问题之一。

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**。

# 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  $RTT_s$ ：

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

- 式中， $0 \leq \alpha < 1$ 。若  $\alpha$  很接近于零，表示 RTT 值更新较慢。若选择  $\alpha$  接近于 1，则表示 RTT 值更新较快。
- RFC 2988 推荐的  $\alpha$  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:

$$RTO = RTT_S + 4 \times RTT_D \quad (5-5)$$

- $RTT_D$  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  $RTT_D$ :

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

- $\beta$  是个小于 1 的系数, 其推荐值是 1/4, 即 0.25。

# 练习：



## ■ 假定tcp建立连接时，RTO=6s：

- 请问当发送方收到对方的连接确认报文段时，测量出RTT样本值为1.5s，试计算现在的RTO值；
- 请问当发送方发送数据报文段并收到确认时，测量出RTT样本值为2.5s，试计算现在的RTO值；

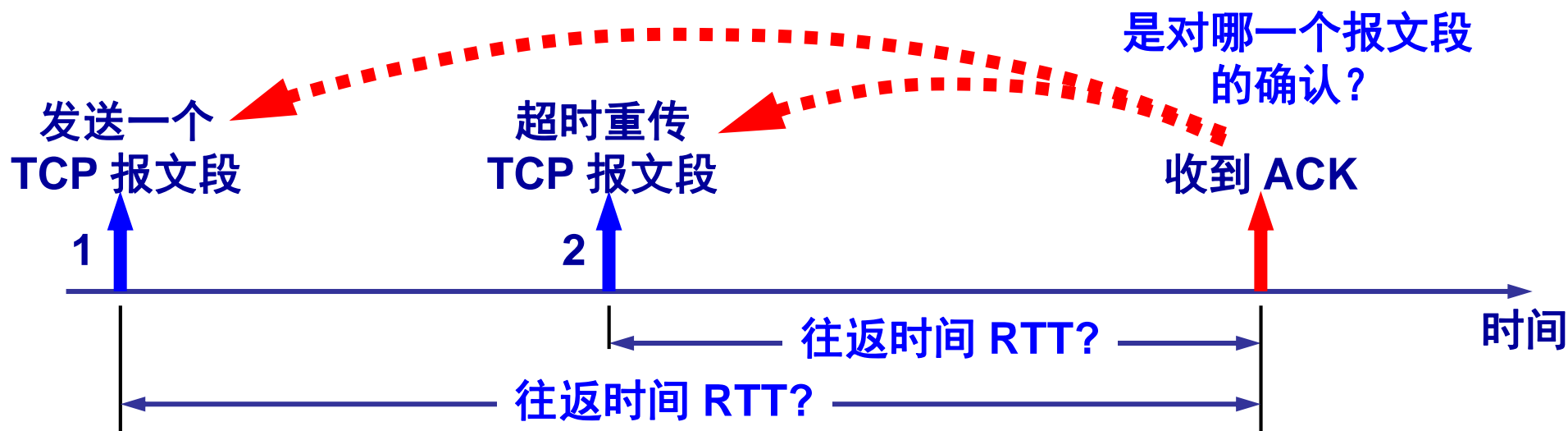
$$(1) \quad 1.5 + 4 * 0.75 = 4.5s$$

$$(2) \quad 1.5 * 0.875 + 2.5 * 0.125 + 4 * (0.75 * 0.75 + 0.25 * (2.5 - 1.5)) = ?$$

# 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：

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

- 系数  $\gamma$  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

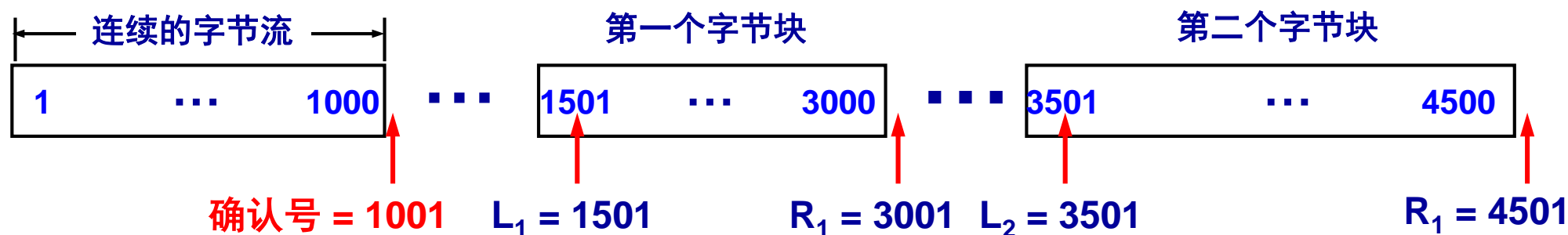


- **问题：**若收到的报文段无差错，只是未按序号，中间还缺少一些序号的数据，那么能否设法只传送缺少的数据而不重传已经正确到达接收方的数据？
- 答案是可以的。**选择确认 SACK** (Selective ACK) 就是一种可行的处理方法。

# 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：左边界和右边界。

- 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

# 知识点



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。

滑动窗口+重传      流量控制

发送速率 $\leq$ 接收速率？

何时发送分组？

何时发送确认？

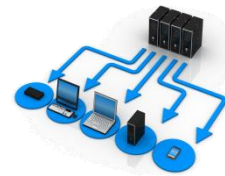
## 5.7.1 利用滑动窗口实现流量控制



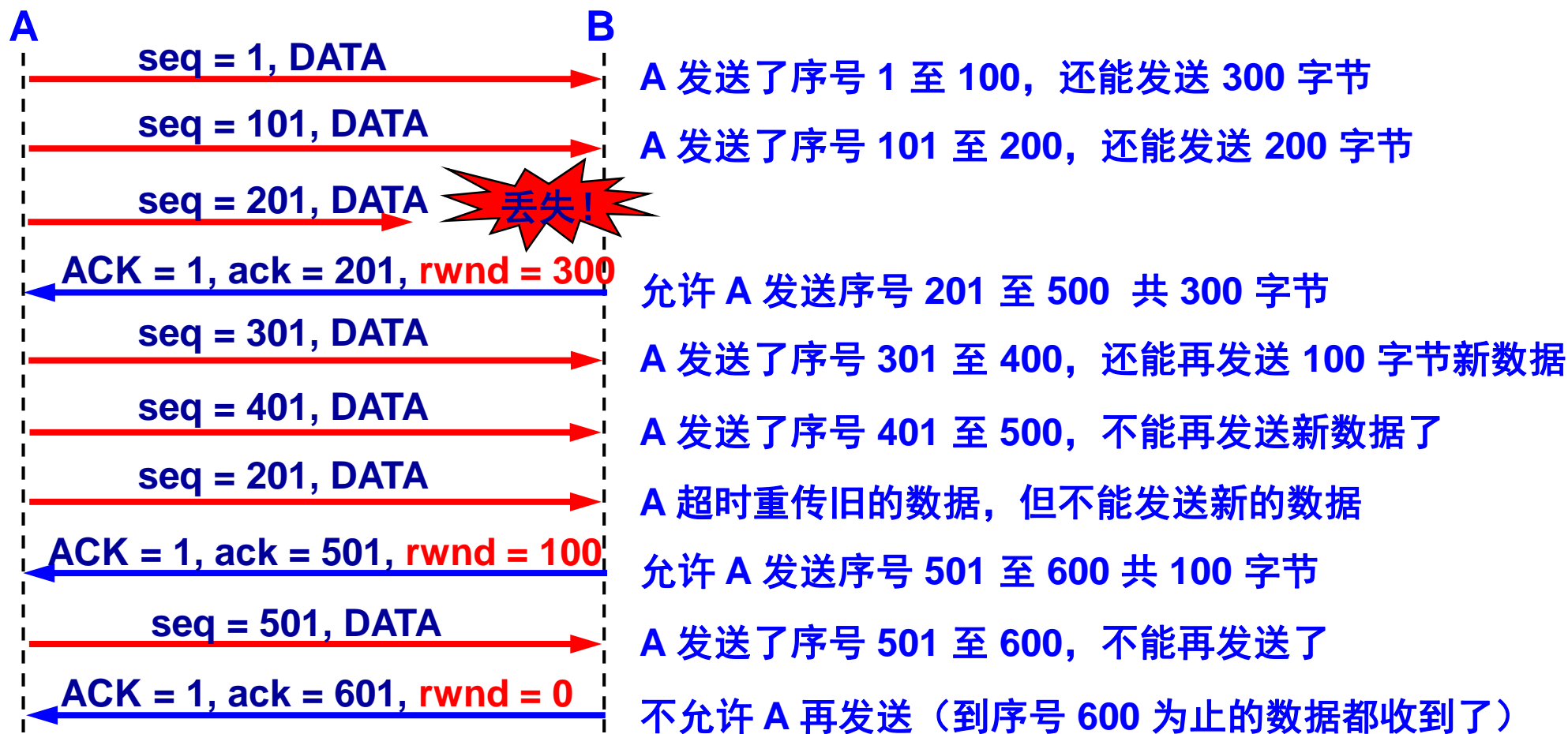
- 一般说来，我们总是希望数据传输得更快一些。但如果发送方把数据发送得过快，接收方就可能来不及接收，这就会造成数据的丢失。
- **流量控制** (flow control) 就是让发送方的发送速率不要太快，既要让接收方来得及接收，也不要使网络发生拥塞。
- 利用**滑动窗口机制**可以很方便地在 TCP 连接上实现流量控制。



# 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）”。



# 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 何时发送分组？



- 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。

# 何时发送确认报文？



- 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。**

# 第 5 章 运输层（四）



# 复习



## ■ 5.6 TCP 可靠传输的实现

- 以字节为单位的滑动窗口
- 超时重传时间的选择
- 选择确认 SACK



## ■ 5.7 TCP 的流量控制



# 本讲重点



## ■ 5.8 TCP 的拥塞控制

- 慢开始 (slow-start)
- 拥塞避免 (congestion avoidance)
- 快重传 (fast retransmit)
- 快恢复 (fast recovery)



## ■ 5.9 TCP 的运输连接管理

- 连接建立 (三次握手)
- 连接释放(四次握手)





# 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（四种方法）
- 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)**。
- 出现拥塞的**原因**：

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

**如何解决拥塞？**

# 增加资源能解决拥塞吗？



不能。

- 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将瓶颈转移到其他地方；

# 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。

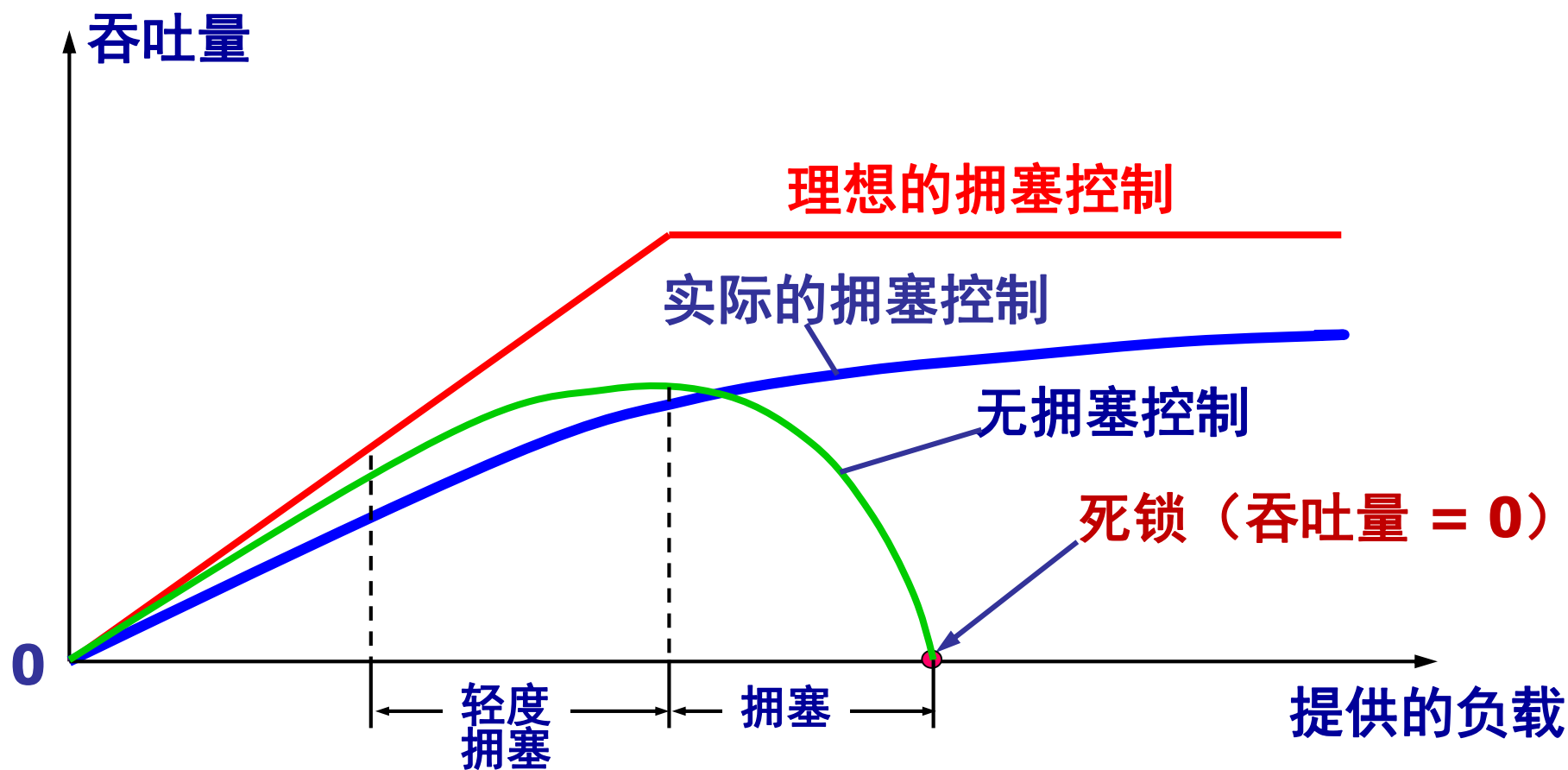
# 拥塞控制与流量控制的区别



- **流量控制**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



# 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 如何进行拥塞控制?**
- TCP 采用**基于窗口的方法**进行拥塞控制。
- 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 (接收方窗口值, 拥塞窗口值)**



# 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



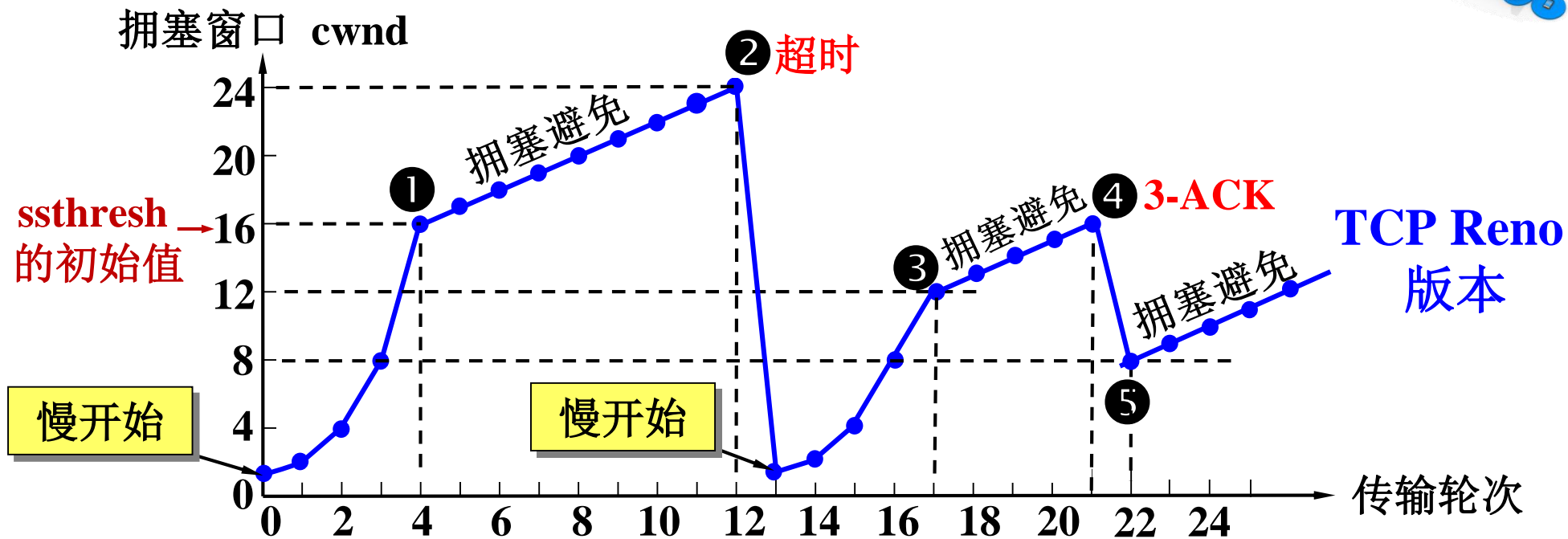
## ■ 四种拥塞控制算法（RFC 5681）：

- 慢开始 (slow-start)
- 拥塞避免 (congestion avoidance)
- 快重传 (fast retransmit)
- 快恢复 (fast recovery)



**注意：** 此处的拥塞控制算法并未考虑流量控制，认为接收方和发送方的缓存是无限大的。

# 慢开始和拥塞避免算法的实现举例



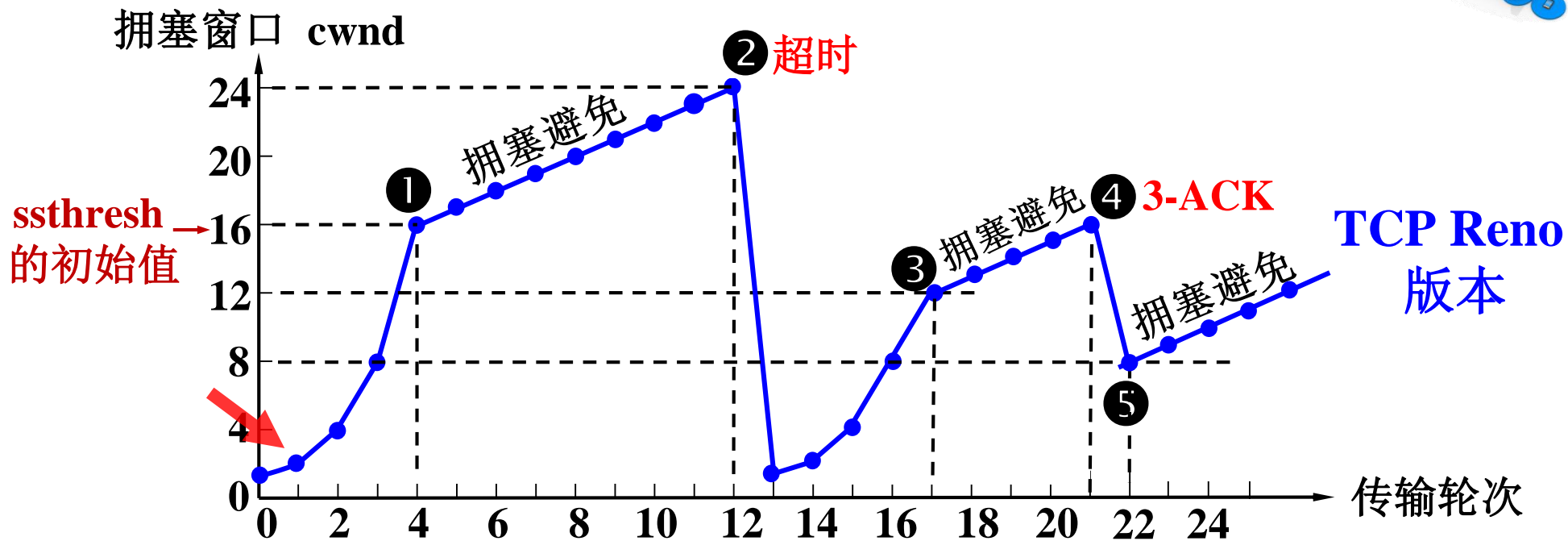
**慢开始：**当 TCP 连接进行初始化时，将拥塞窗口置为 1。图中的窗口单位不使用字节而使用报文段。

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



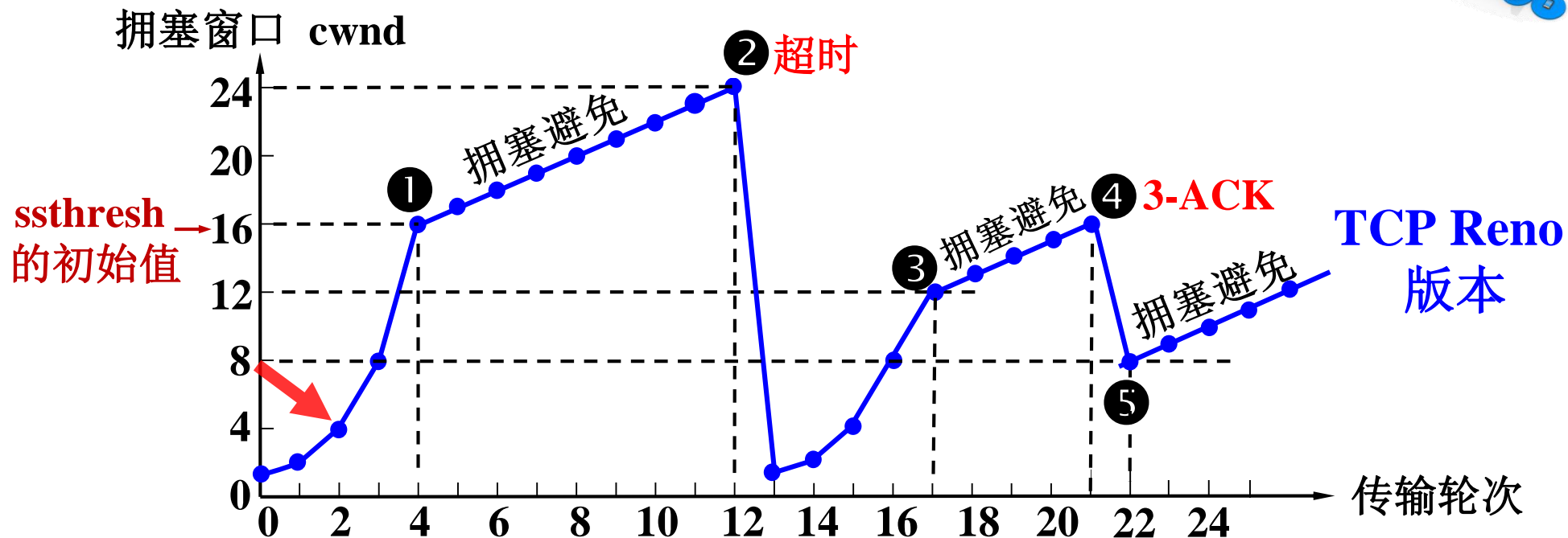
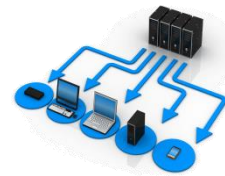
**慢开始：**在执行慢开始算法时，拥塞窗口  $cwnd=1$ ，发送第一个报文段。

# 慢开始和拥塞避免算法的实现举例



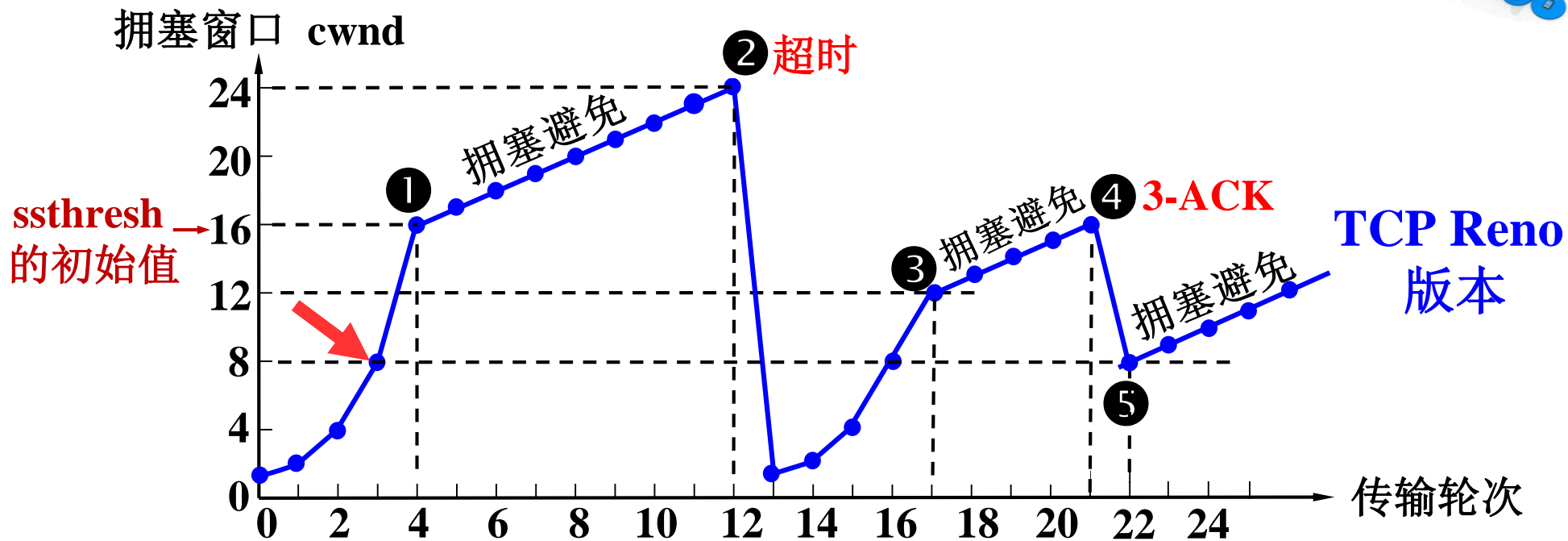
**慢开始：**发送方每收到一个对新报文段的确认 ACK，就把拥塞窗口值加 1 倍，然后开始下一轮的传输（请注意，横坐标是传输轮次，不是时间）。因此拥塞窗口 cwnd 随着传输轮次按指数规律增长。

# 慢开始和拥塞避免算法的实现举例



**慢开始：**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 倍，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



**慢开始：**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 倍，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。



**拥塞避免算法：**当拥塞窗口  $cwnd$  增长到慢开始门限值  $ssthresh$  时（图中的点①，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**。

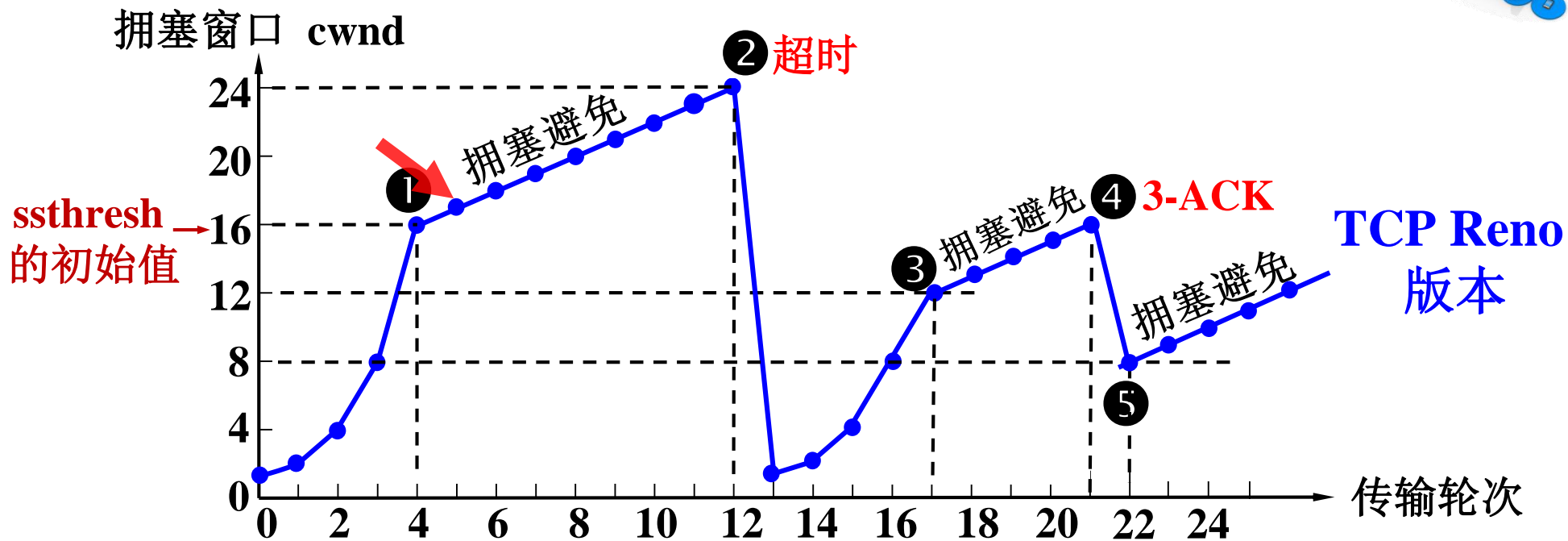


# 必须强调指出



- “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

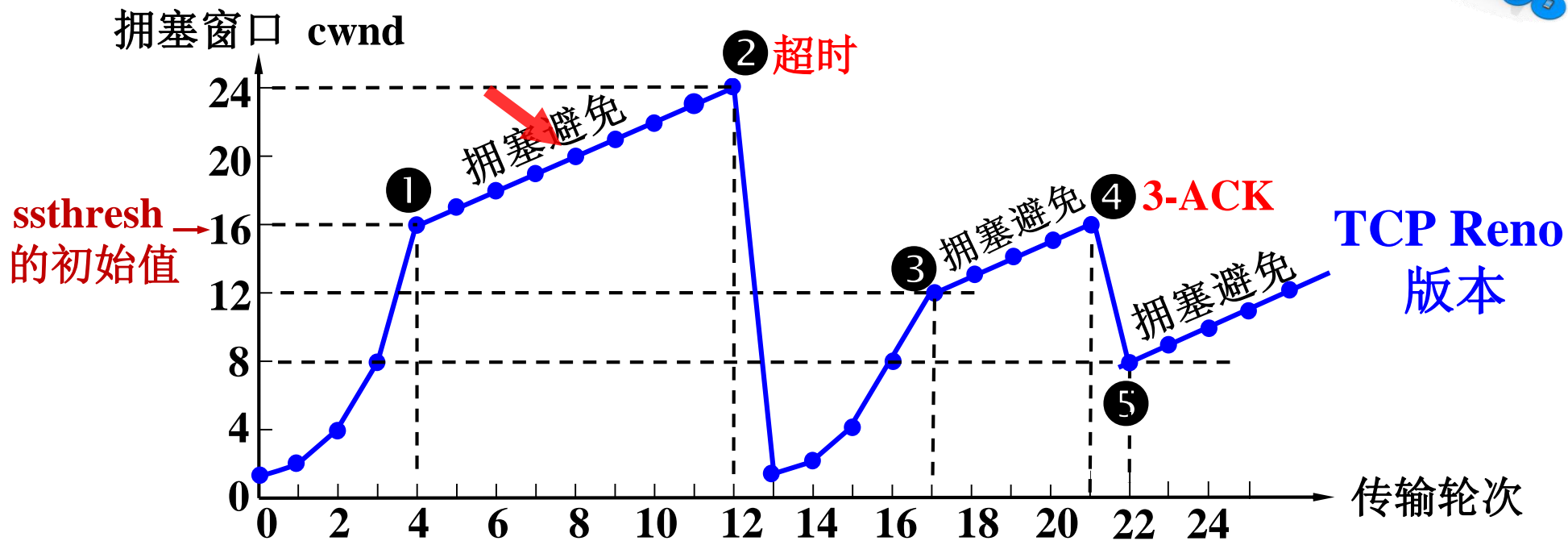


**拥塞避免算法：**当拥塞窗口  $cwnd > 16$  时，进入拥塞避免，拥塞窗口每一轮次只加1。



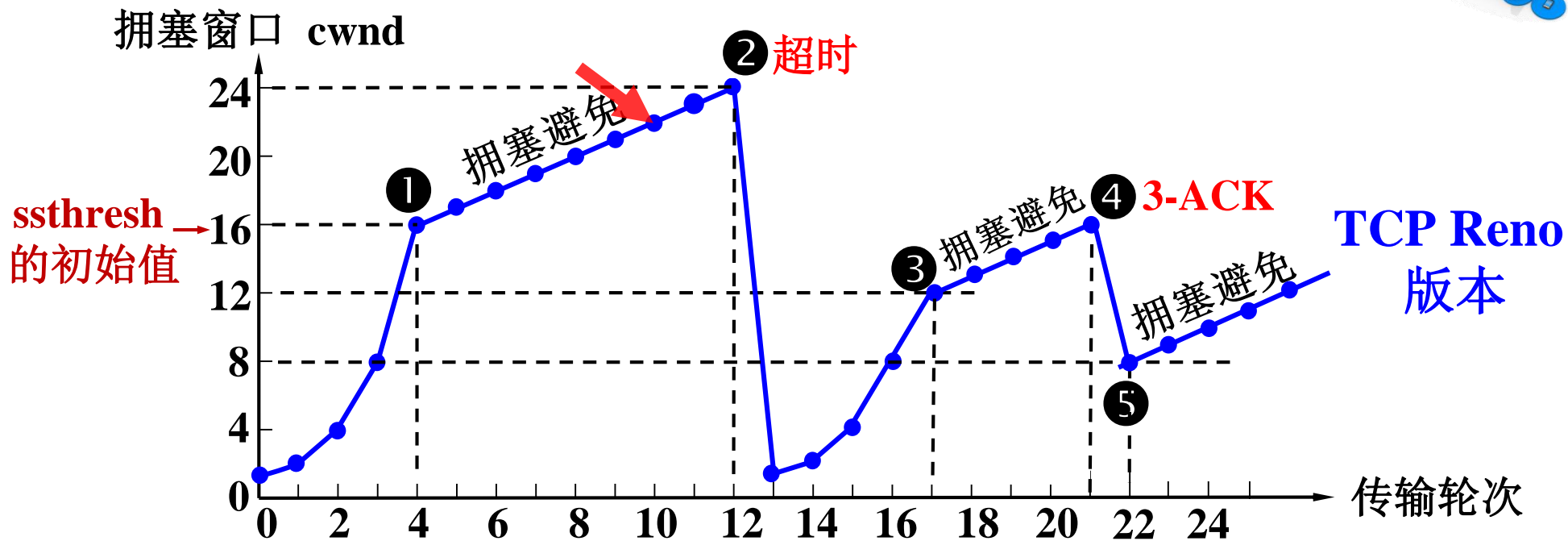
**拥塞避免算法：**当拥塞窗口  $cwnd > 16$  时，进入拥塞避免，拥塞窗口每一轮次只加1。

# 慢开始和拥塞避免算法的实现举例



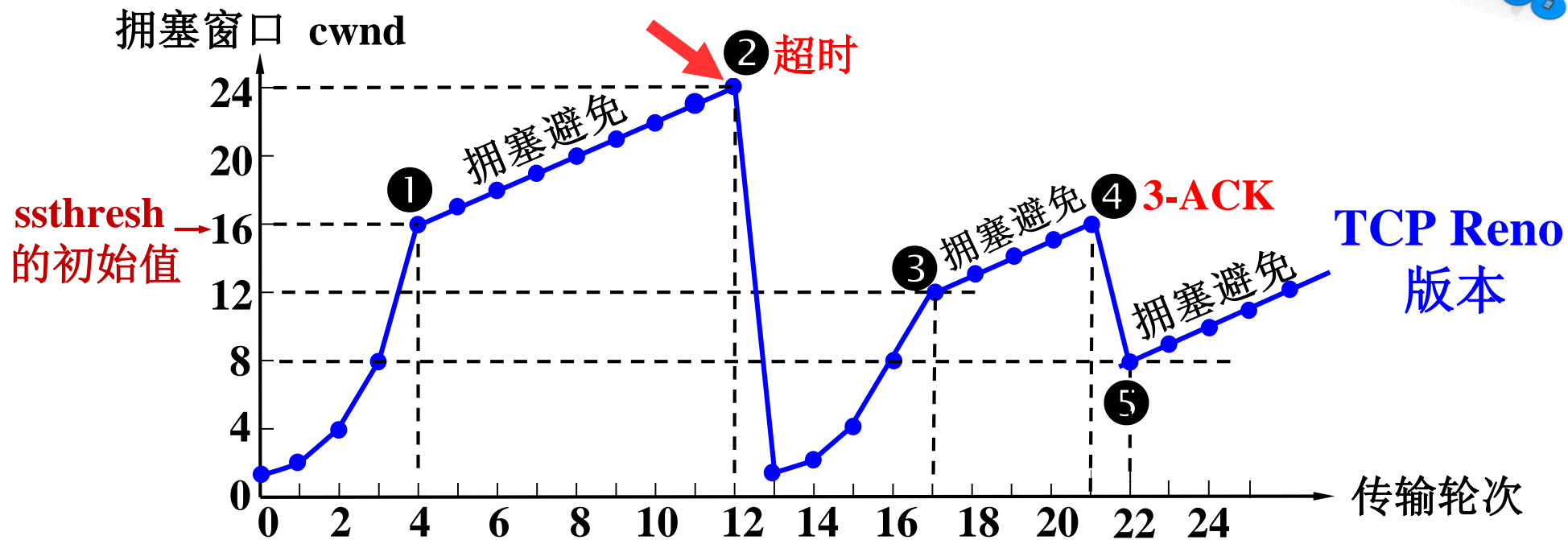
**拥塞避免算法：**当拥塞窗口  $cwnd > 16$  时，进入拥塞避免，拥塞窗口每一轮次**只加1**。

# 慢开始和拥塞避免算法的实现举例



**拥塞避免算法：**当拥塞窗口  $cwnd > 16$  时，进入拥塞避免，拥塞窗口每一轮次只加1。

# 慢开始和拥塞避免算法的实现举例



**拥塞发生：**当拥塞窗口  $cwnd = 24$  时，网络出现了**超时**（图中的点②），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。



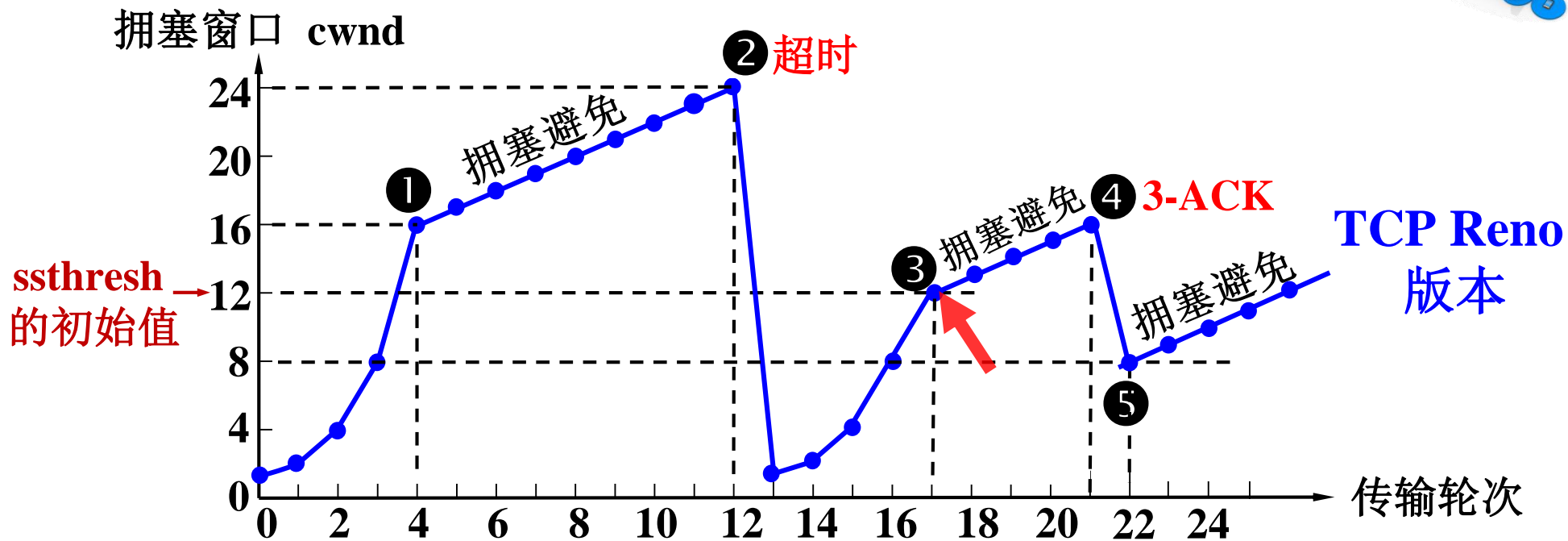
当拥塞窗口  $cwnd = 24$  时，网络出现了**超时**（图中的点②），发送方判断为网络拥塞。于是**调整门限值**  
 $ssthresh = cwnd / 2 = 12$ ，同时设置拥塞窗口  $cwnd = 1$ ，进入**慢开始**阶段。



当拥塞窗口  $cwnd = 24$  时，网络出现了**超时**（图中的点②），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  $cwnd = sssthresh = 12$  时（图中的点③，这是新的  $sssthresh$  值），改为执行**拥塞避免**算法，拥塞窗口**按线性规律增大**。

# 问题

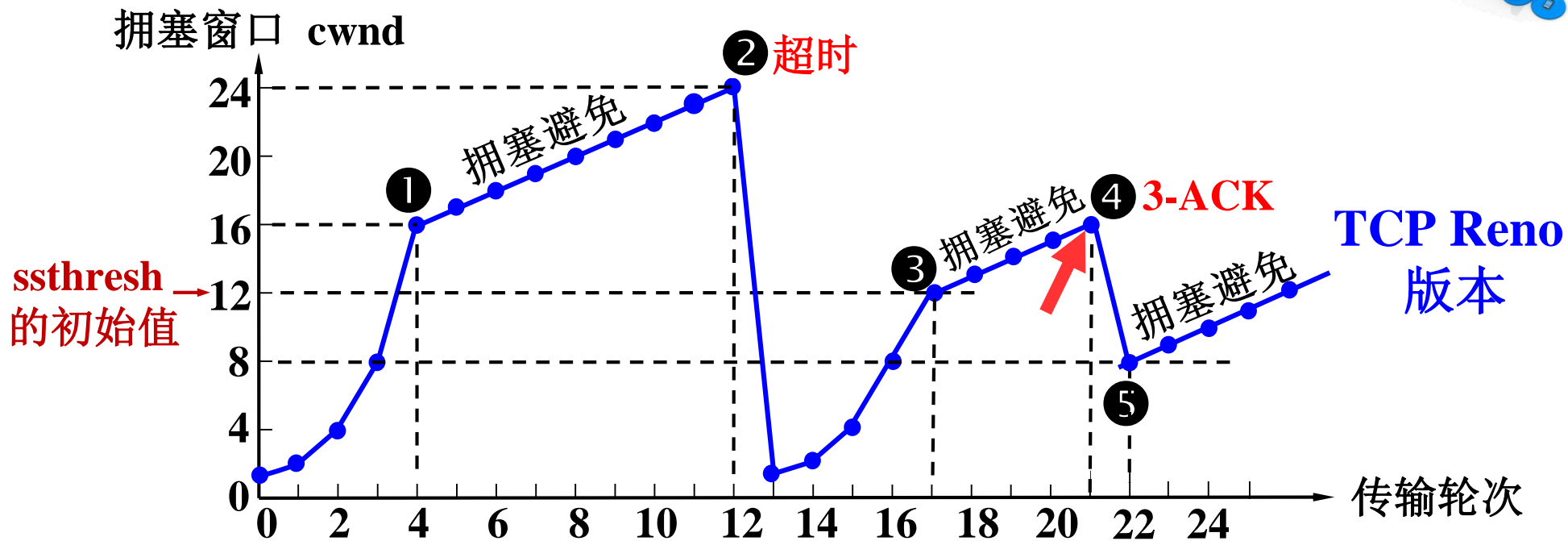


- **拥塞避免**，认为网络出现大面积超时重传时，则网络网络拥塞，执行慢开始算法。但是，很多情况下，网络只出现一个小小的分组丢失，就会引起“小题大做”的结果？
- **慢开始**，每次拥塞窗口都从1开始，下降速度太快，网络性能波动太大。



快重传和快恢复

# 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④），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。

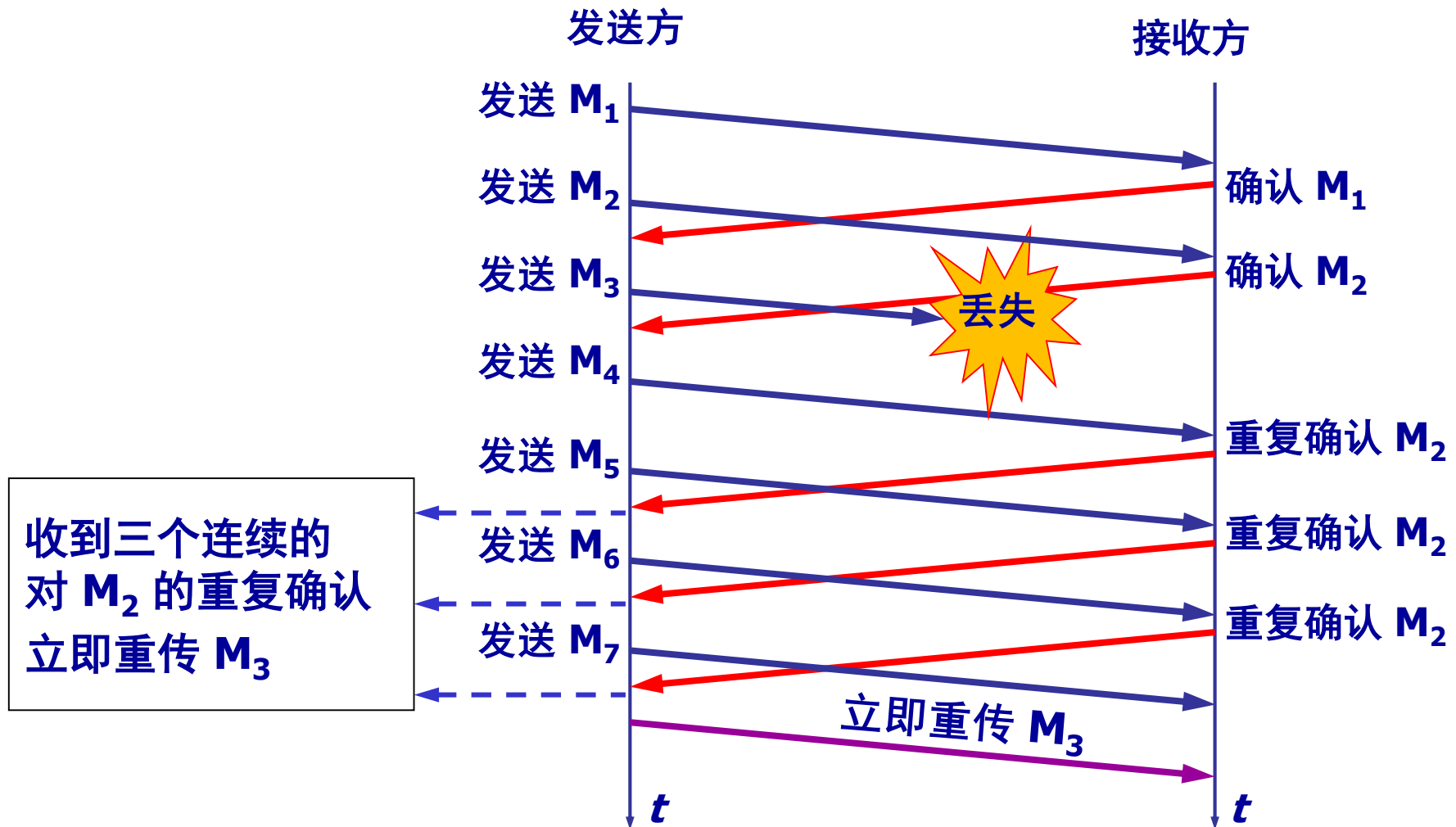
# 快重传算法



- 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”），这样就不会出现因少数几个分组超时，导致发送方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

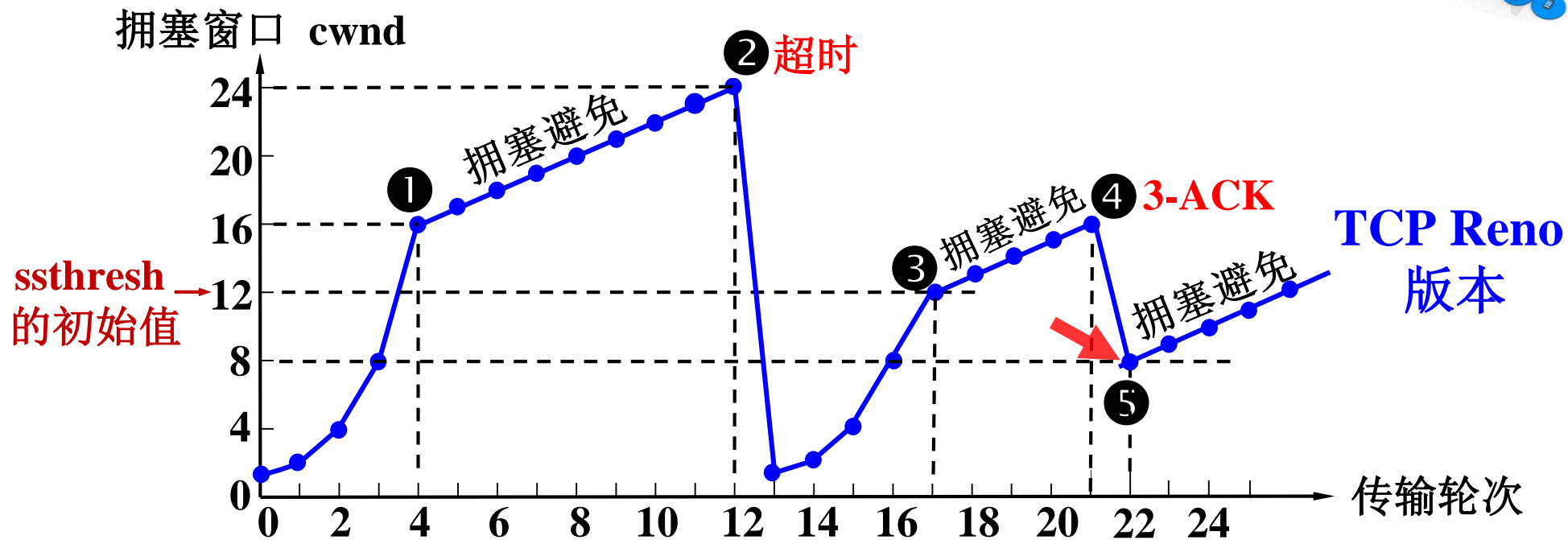


# 快恢复算法



- 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  $ssthresh = \text{当前拥塞窗口 } cwnd / 2$  ；
  - (2) 新拥塞窗口  $cwnd = \text{慢开始门限 } ssthresh$  ；
  - (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④，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  $sssthresh = cwnd / 2 = 8$ ，同时设置拥塞窗口  $cwnd = sssthresh = 8$ （见图中的点⑤），并开始执行拥塞避免算法。

# 练习：



【2009年 题39】一个TCP连接总是以1KB的最大段长发送TCP段，发送方有足够多的数据要发送。当拥塞窗口为16KB时发生了超时，如果接下来的4个RTT（往返时间）内的TCP段的传输都是成功的，那么当第4个RTT时间内发送的所有TCP段都得到肯定应答时，拥塞窗口大小是

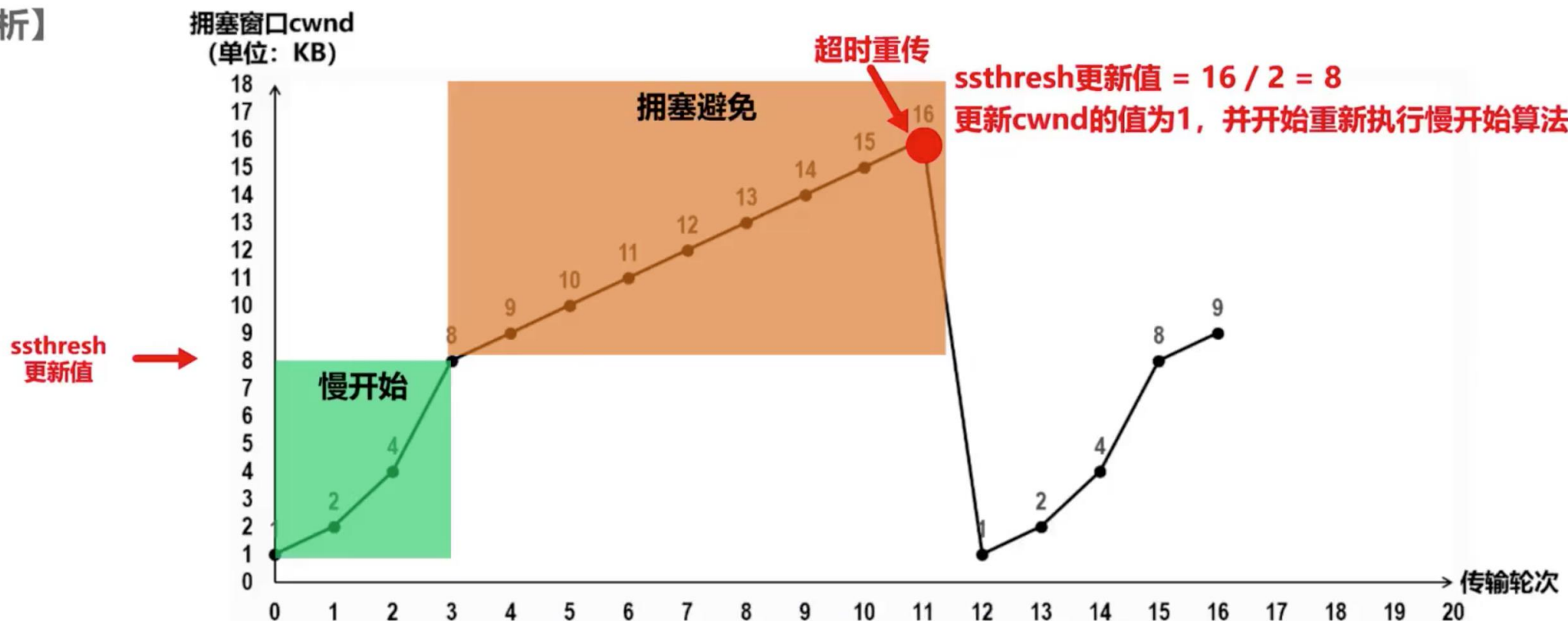
A. 7KB

B. 8KB

C. 9KB

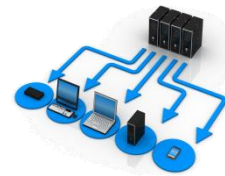
D. 16KB

【解析】



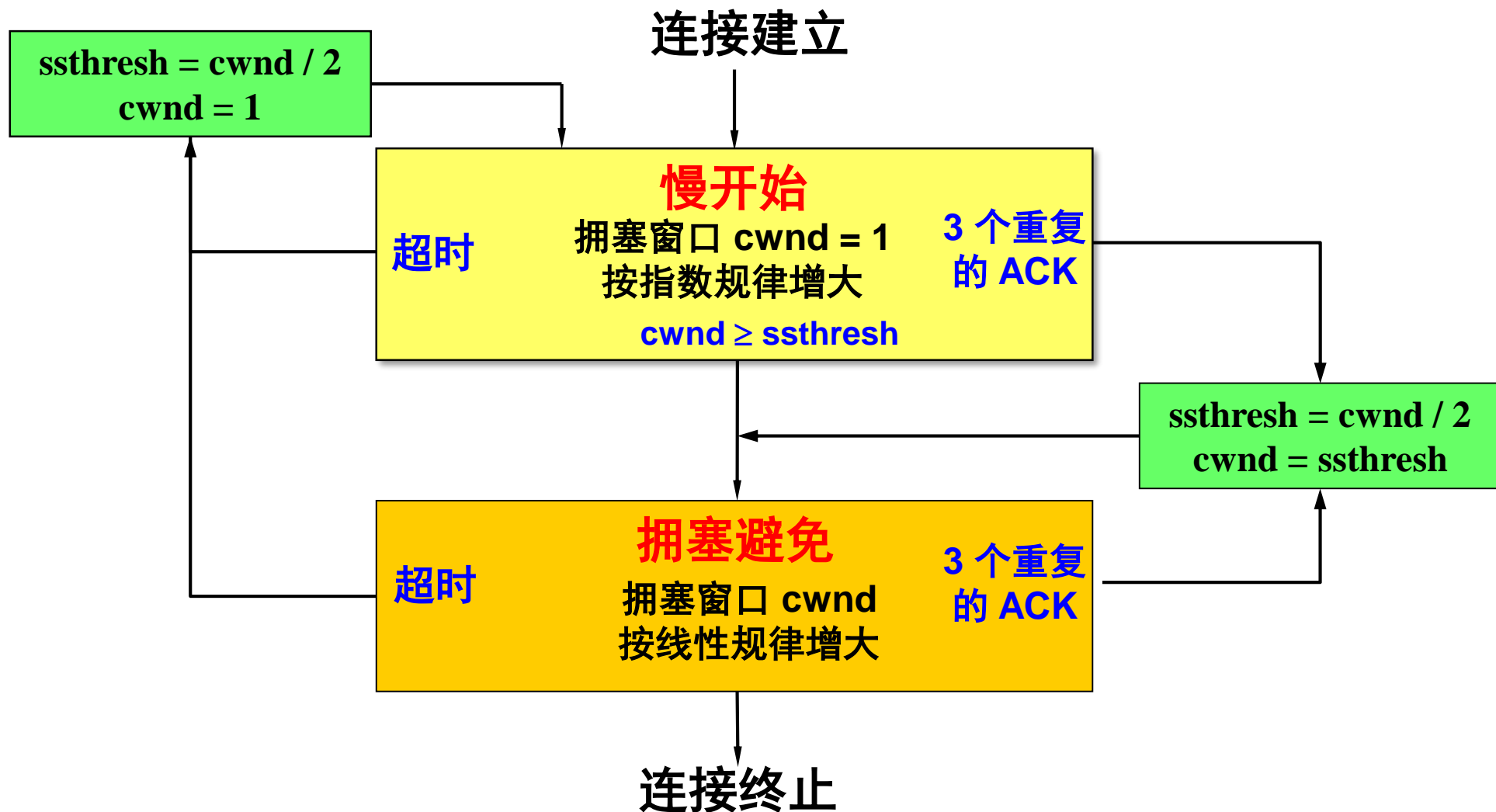


# 加法增大，乘法减小 (AIMD)



- 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



# 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：

$$\text{发送窗口的上限值} = \text{Min} [rwnd, cwnd] \quad (5-9)$$

- 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



- 对 TCP 拥塞控制**影响最大**的就是路由器的分组丢弃策略。
- 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)。

# 主动队列管理AQM

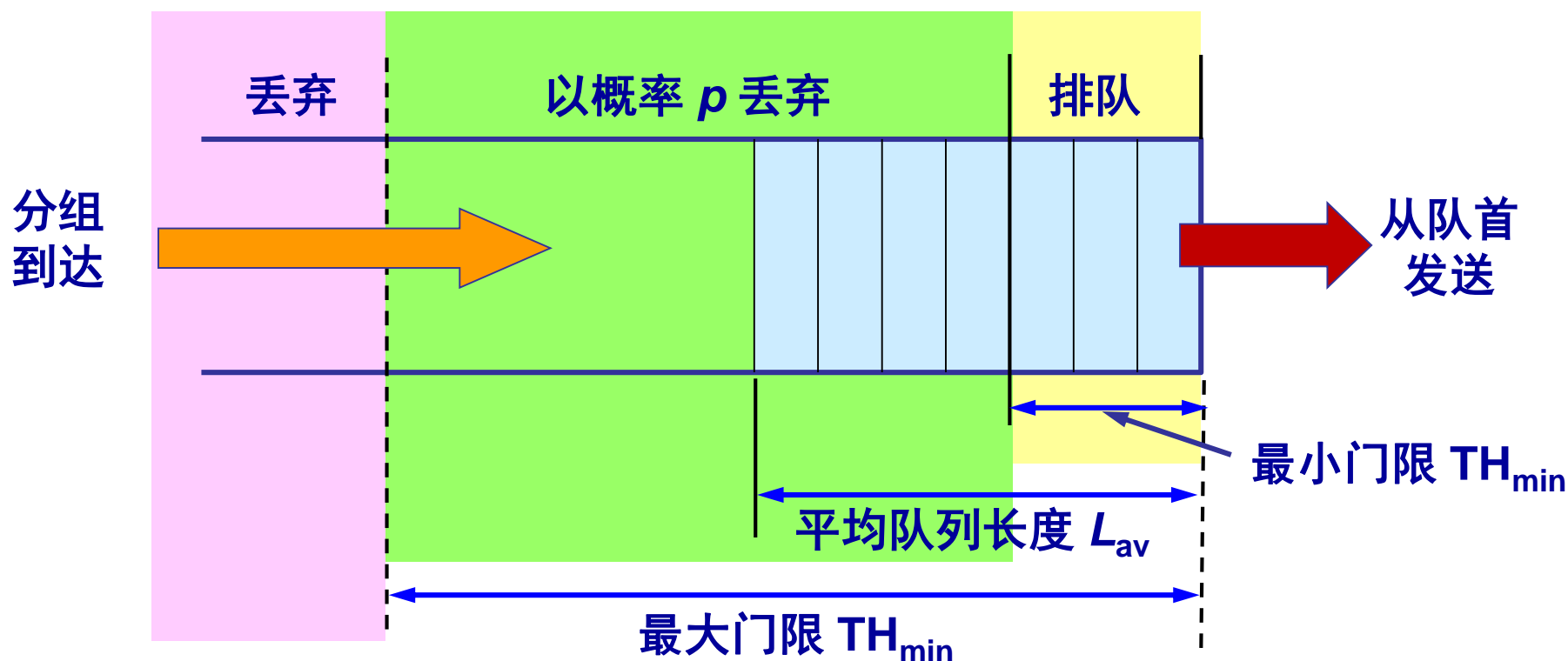


- 1998 年提出了主动队列管理 AQM (Active Queue Management)。
- 所谓“主动”就是不要等到路由器的队列长度已经达到最大值时才不得不丢弃后面到达的分组。这样就太被动了。应当在队列长度达到某个值得警惕的数值时（即当网络拥塞有了某些拥塞征兆时），就**主动丢弃**到达的分组。
- AQM 可以有不同实现方法，其中曾流行多年的就是**随机早期检测 RED** (Random Early Detection)。

# 随机早期检测 RED



RED 将路由器的到达队列划分成为三个区域：



# 随机早期检测 RED



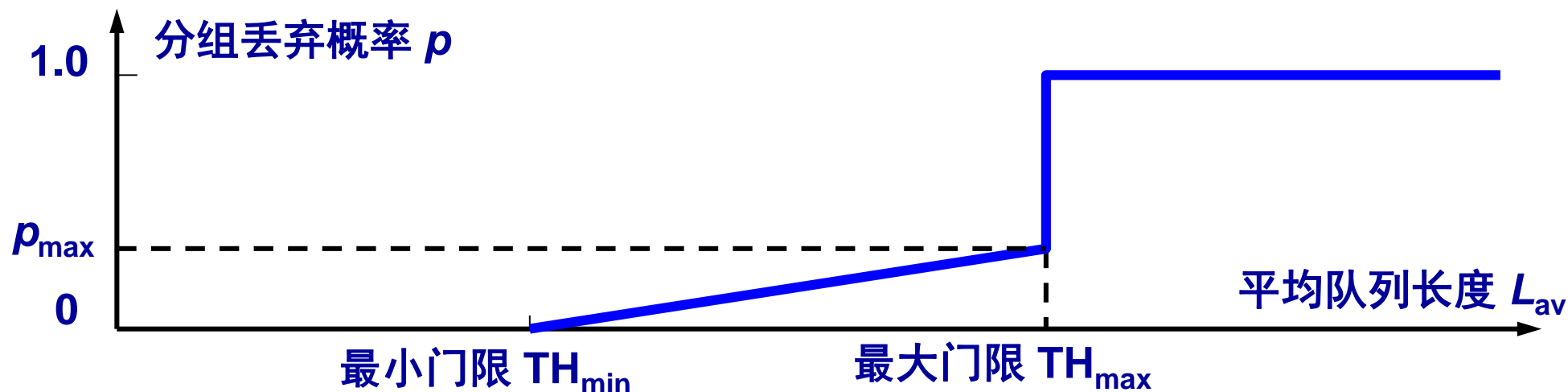
- 使路由器的队列维持两个参数：队列长度最小门限  $TH_{min}$  和最大门限  $TH_{max}$ 。
- RED 对每一个到达的分组都先计算平均队列长度  $L_{AV}$ 。
- (1) 若平均队列长度小于最小门限  $TH_{min}$ ，则将新到达的分组放入队列进行排队。
- (2) 若平均队列长度超过最大门限  $TH_{max}$ ，则将新到达的分组丢弃。
- (3) 若平均队列长度在最小门限  $TH_{min}$  和最大门限  $TH_{max}$  之间，则按照某一概率  $p$  将新到达的分组丢弃。

# 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。

# 5.9 TCP 的运输连接管理

---



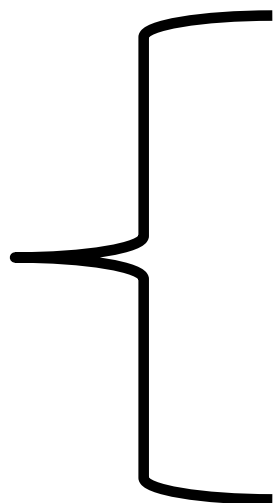
- 5.9.1 TCP 的连接建立
- 5.9.2 TCP 的连接释放
- 5.9.3 TCP 的有限状态机

# 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



超时重传

流量控制

拥塞控制

# 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。

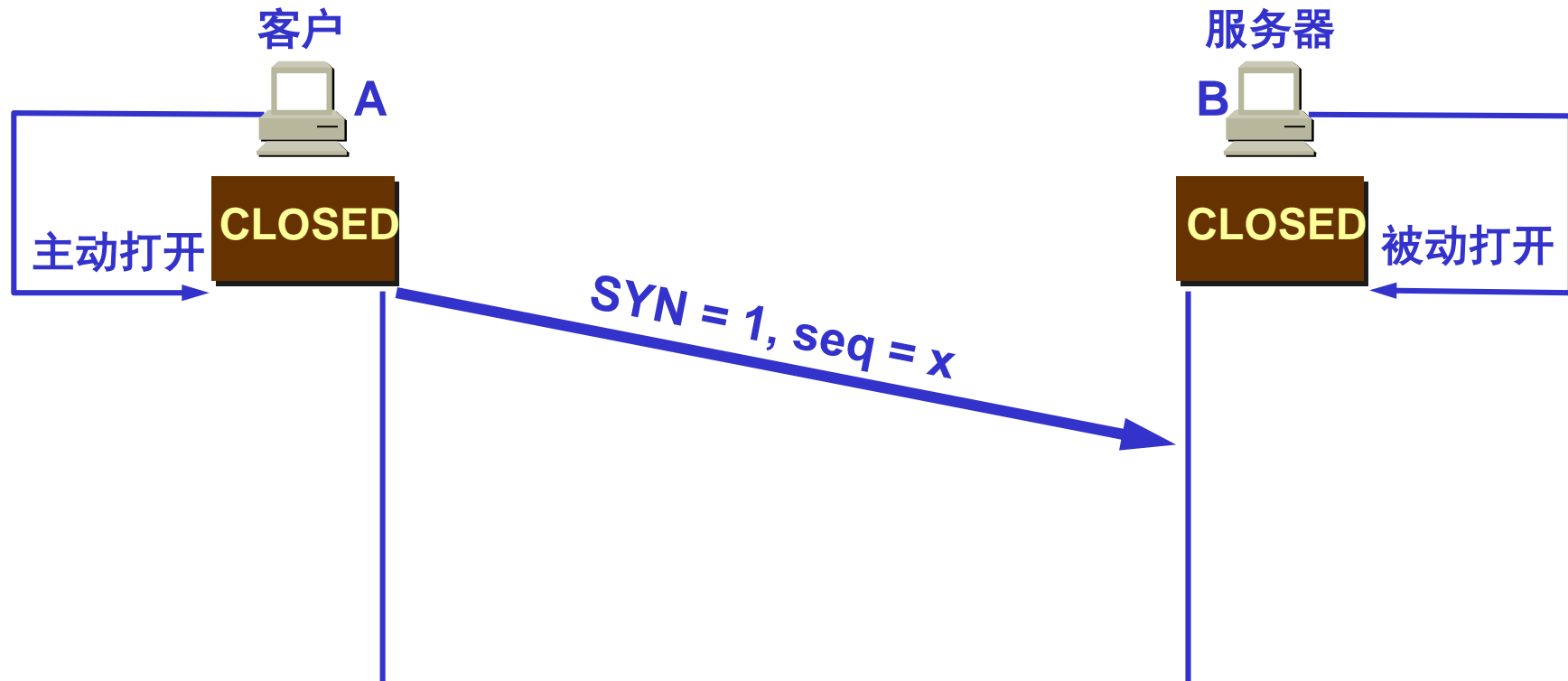
## 5.9.1 TCP 的连接建立



- TCP 建立连接的过程叫做**握手**。
- 握手需要在客户和服务端之间交换三个 TCP 报文段。称之为**三报文握手**。

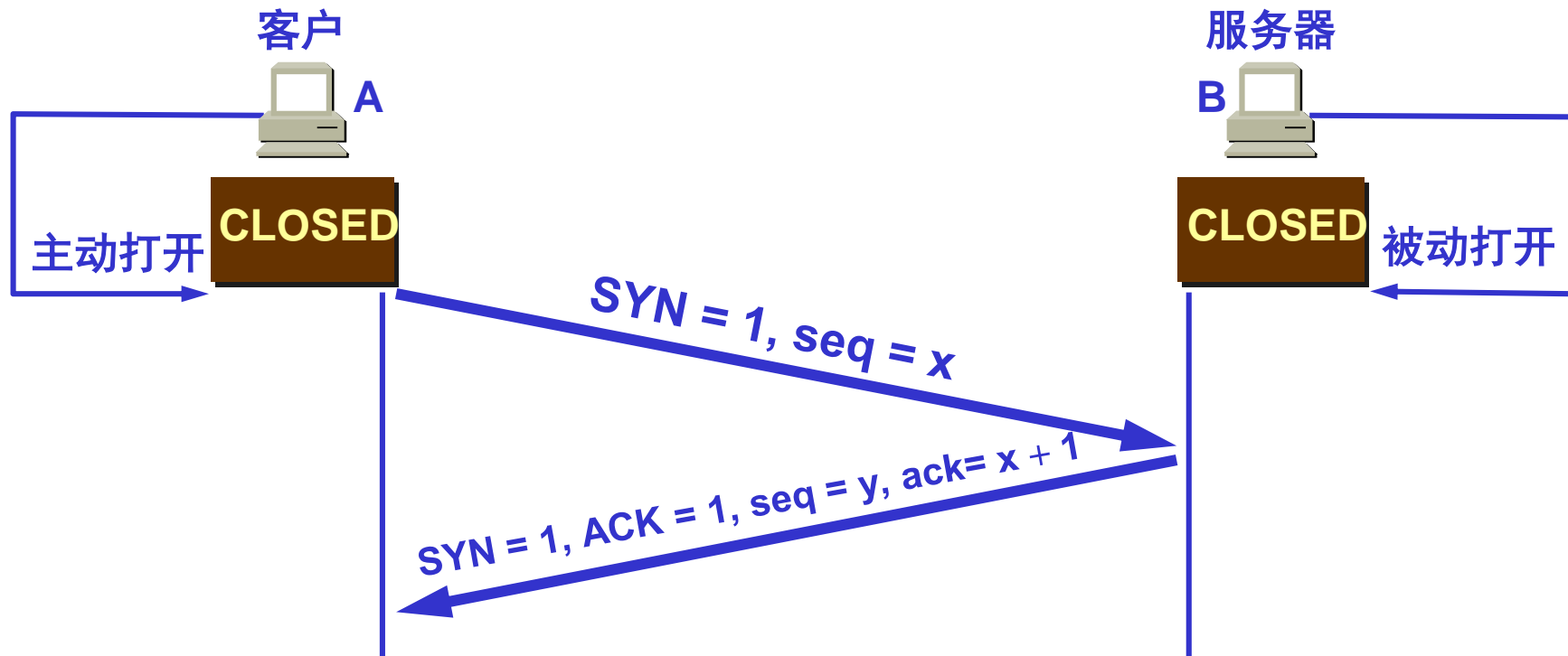
**三次握手**

# TCP 的连接建立：采用三报文握手



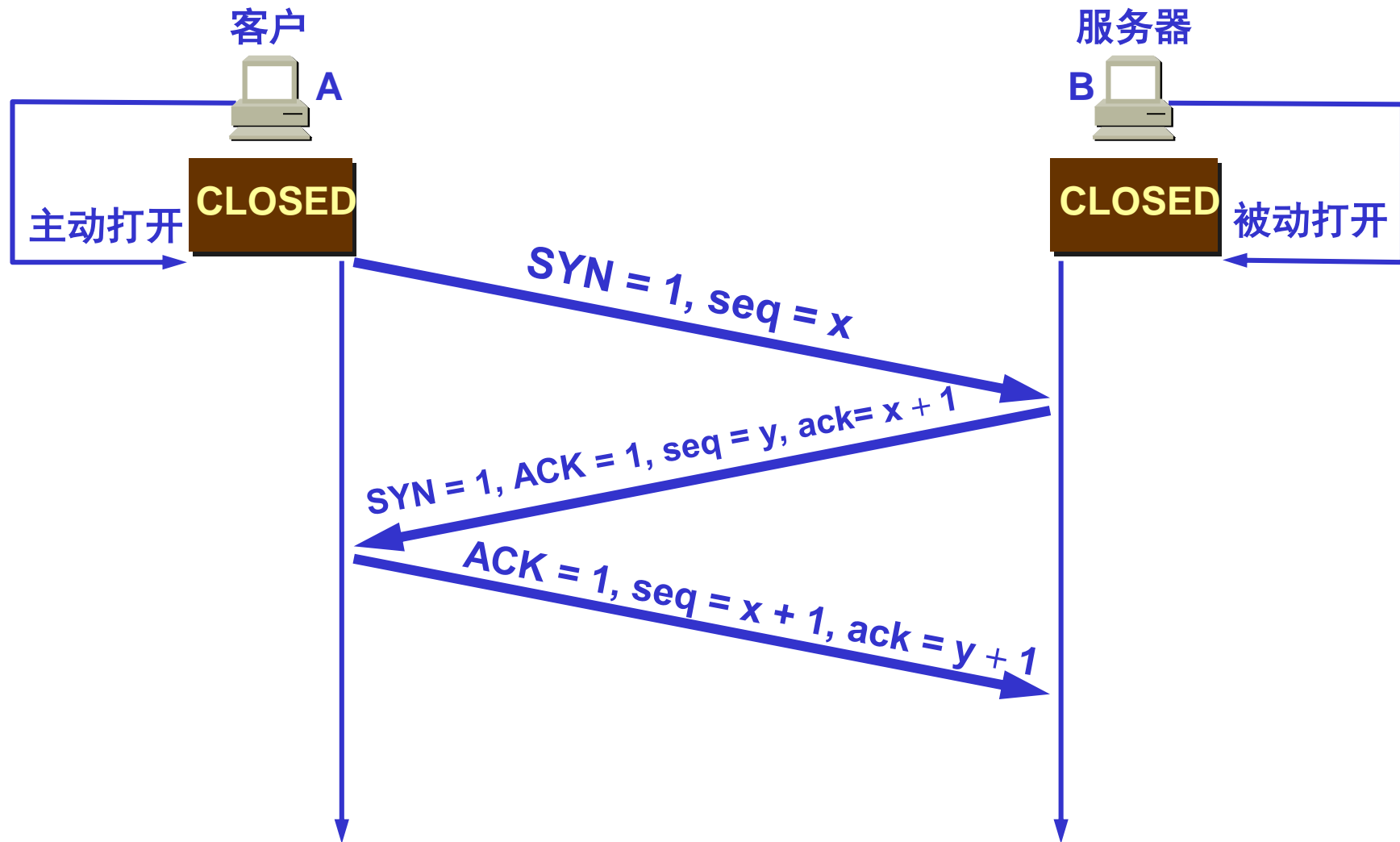
A 的 TCP 向 B 发出连接请求报文段，其首部中的同步位  $SYN = 1$ ，并选择序号  $seq = x$ ，表明传送数据时的第一个数据字节的序号是  $x$ 。

# TCP 的连接建立：采用三报文握手



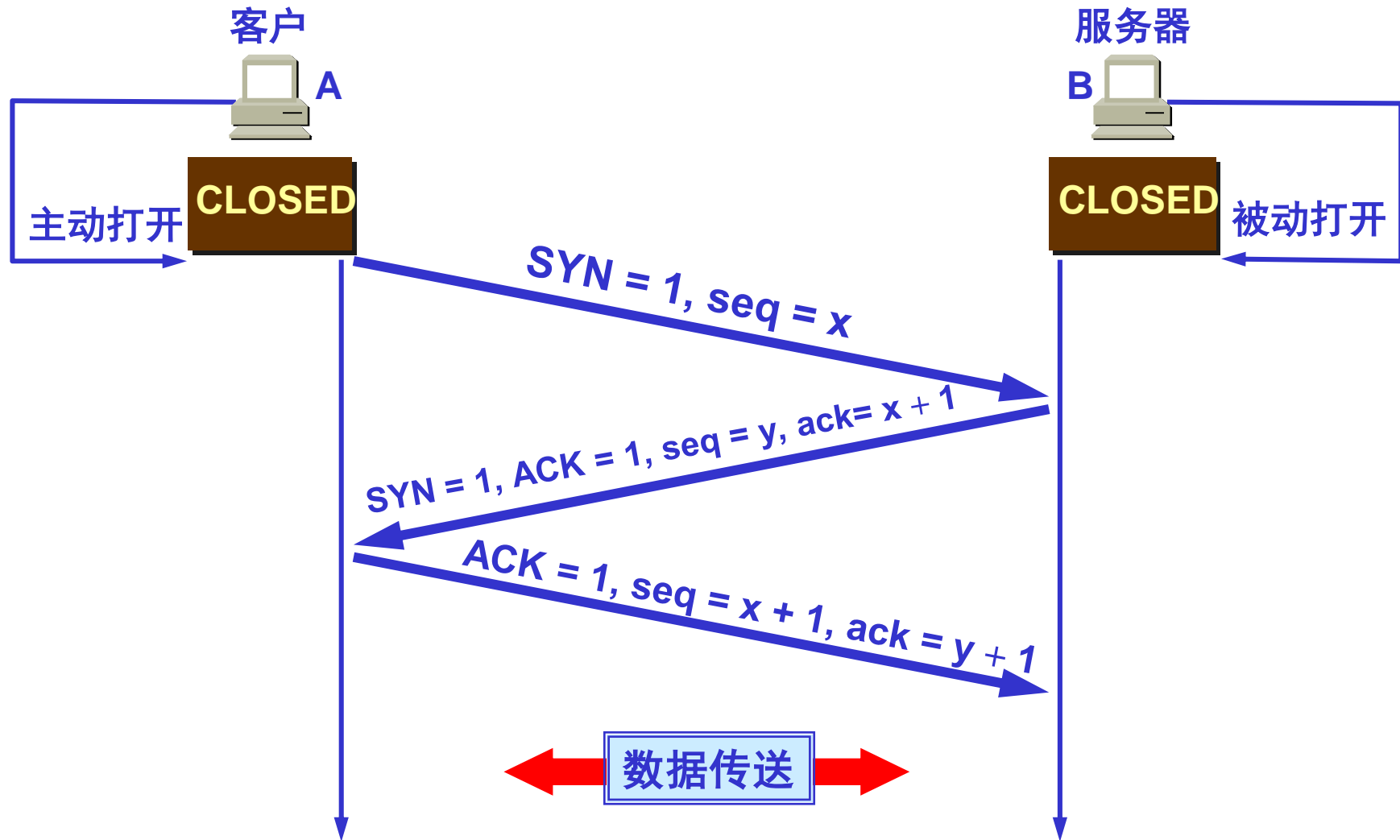
- B 的 TCP 收到连接请求报文段后，如同意，则发回确认。
- B 在确认报文段中应使  $SYN = 1$ ，使  $ACK = 1$ ，其确认号  $ack = x + 1$ ，自己选择的序号  $seq = y$ 。

- 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。

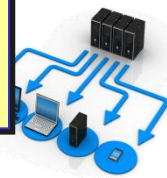




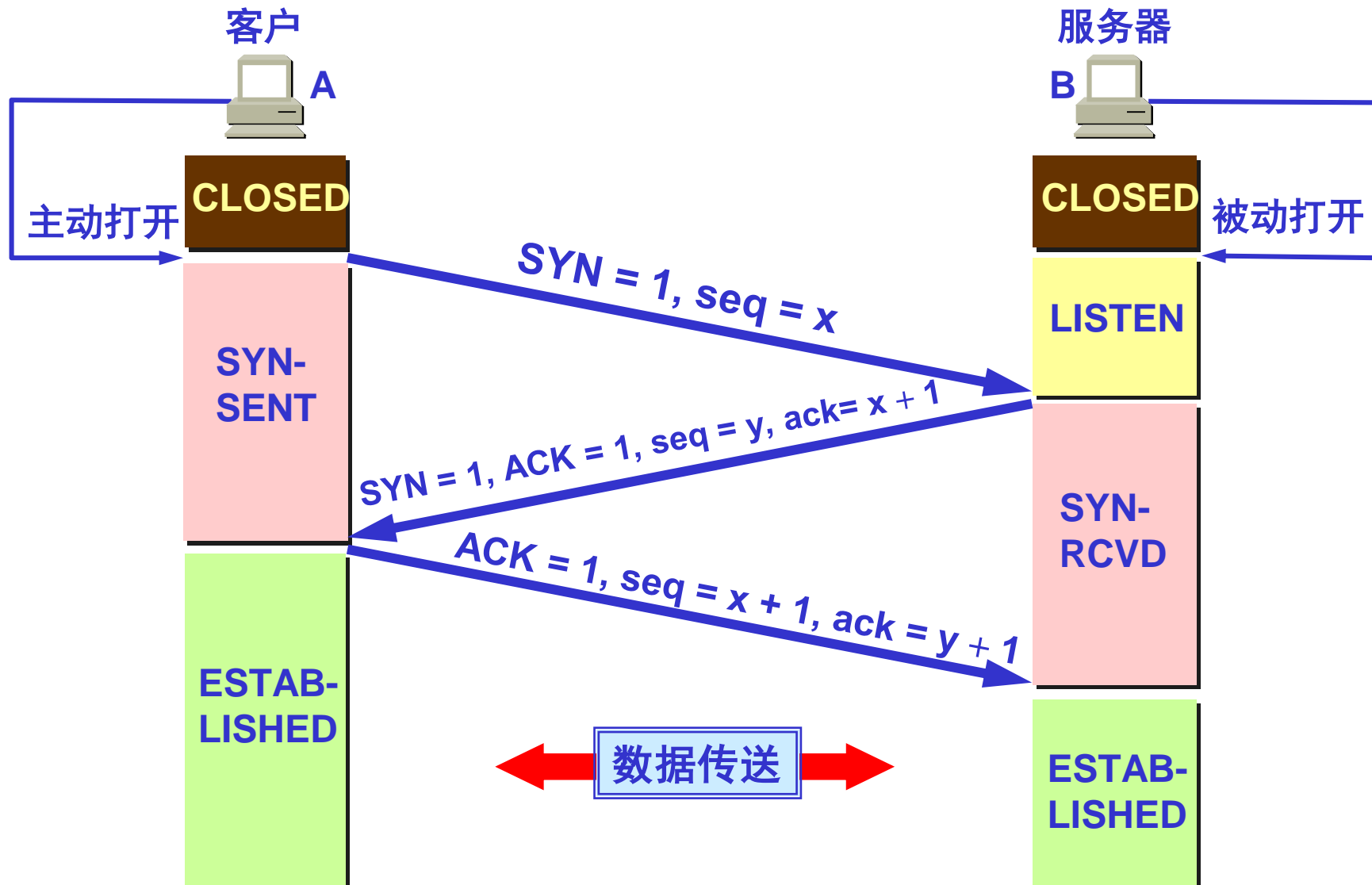
- B 的 TCP 收到主机 A 的确认后，也通知其上层应用进程：TCP 连接已经建立。



# 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



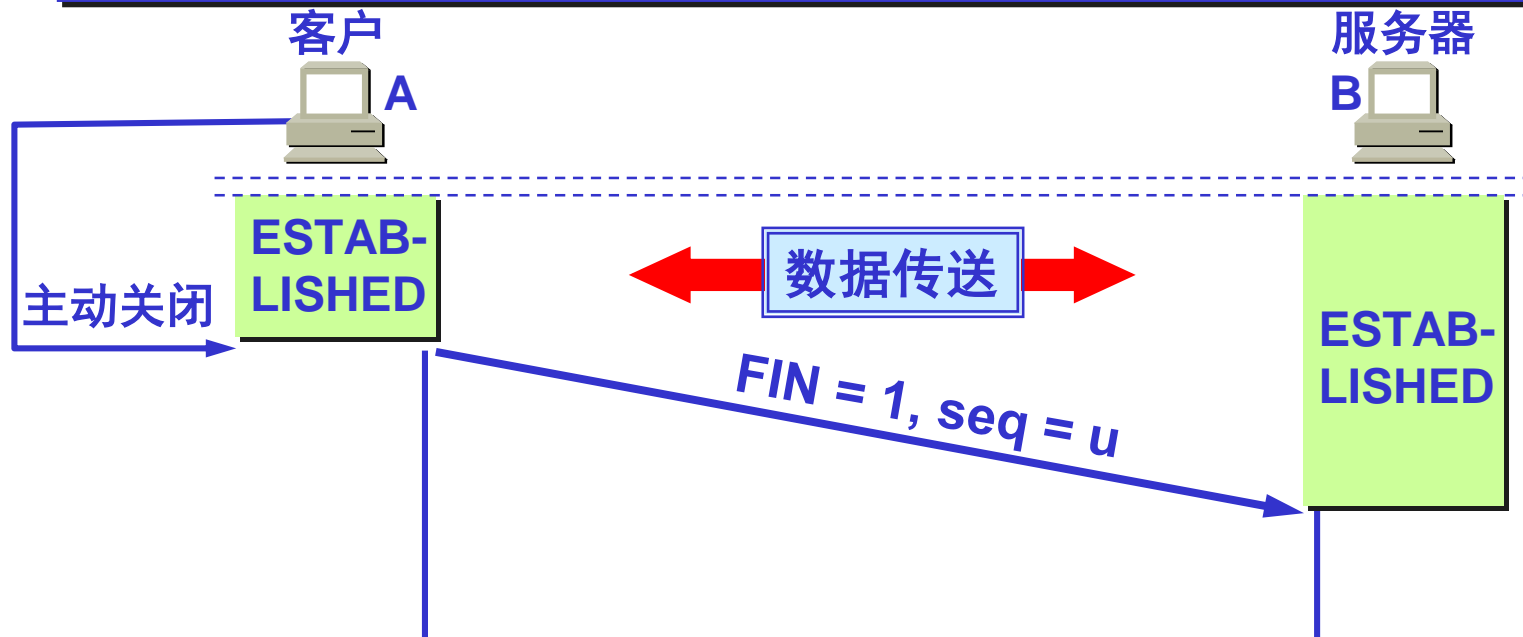
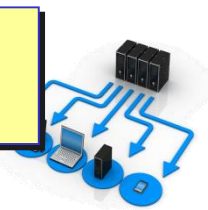
## 5.9.2 TCP 的连接释放



- 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**。

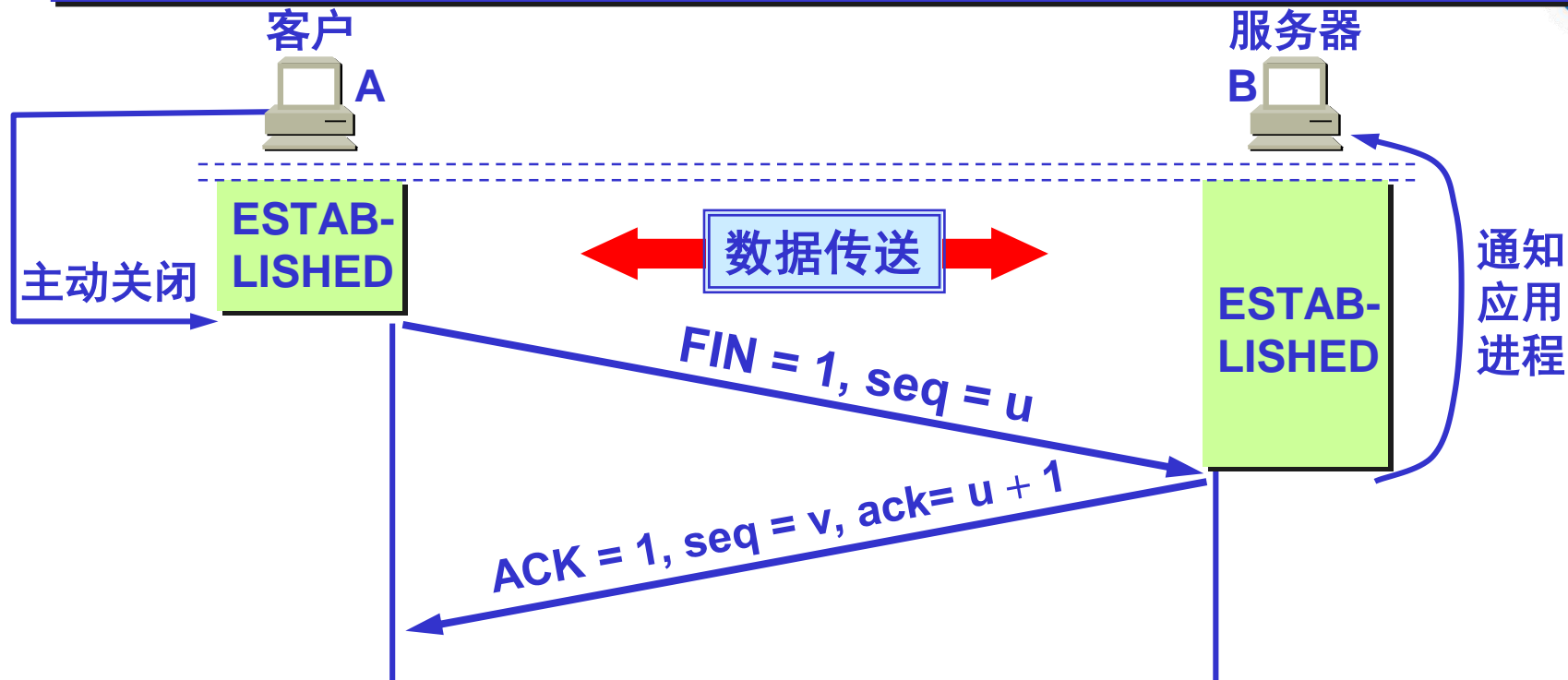
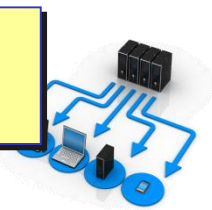
**四次握手**

# TCP 的连接释放：采用四报文握手



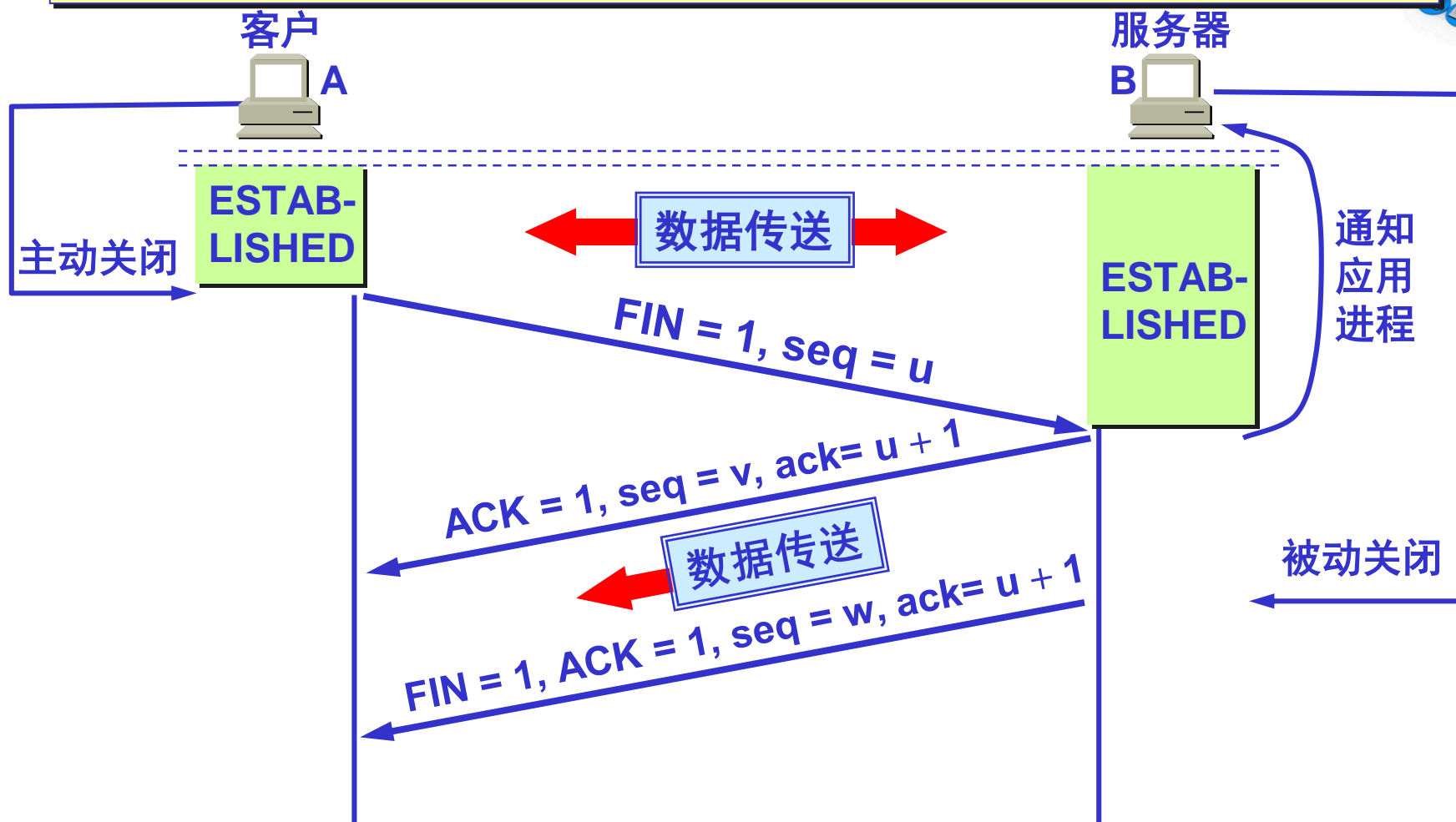
- 数据传输结束后，通信的双方都可释放连接。
- 现在 A 的应用进程先向其 TCP 发出连接释放报文段，并停止再发送数据，主动关闭 TCP 连接。
- A 把连接释放报文段首部的  $FIN = 1$ ，其序号  $seq = u$ ，等待 B 的确认。

# TCP 的连接释放：采用四报文握手



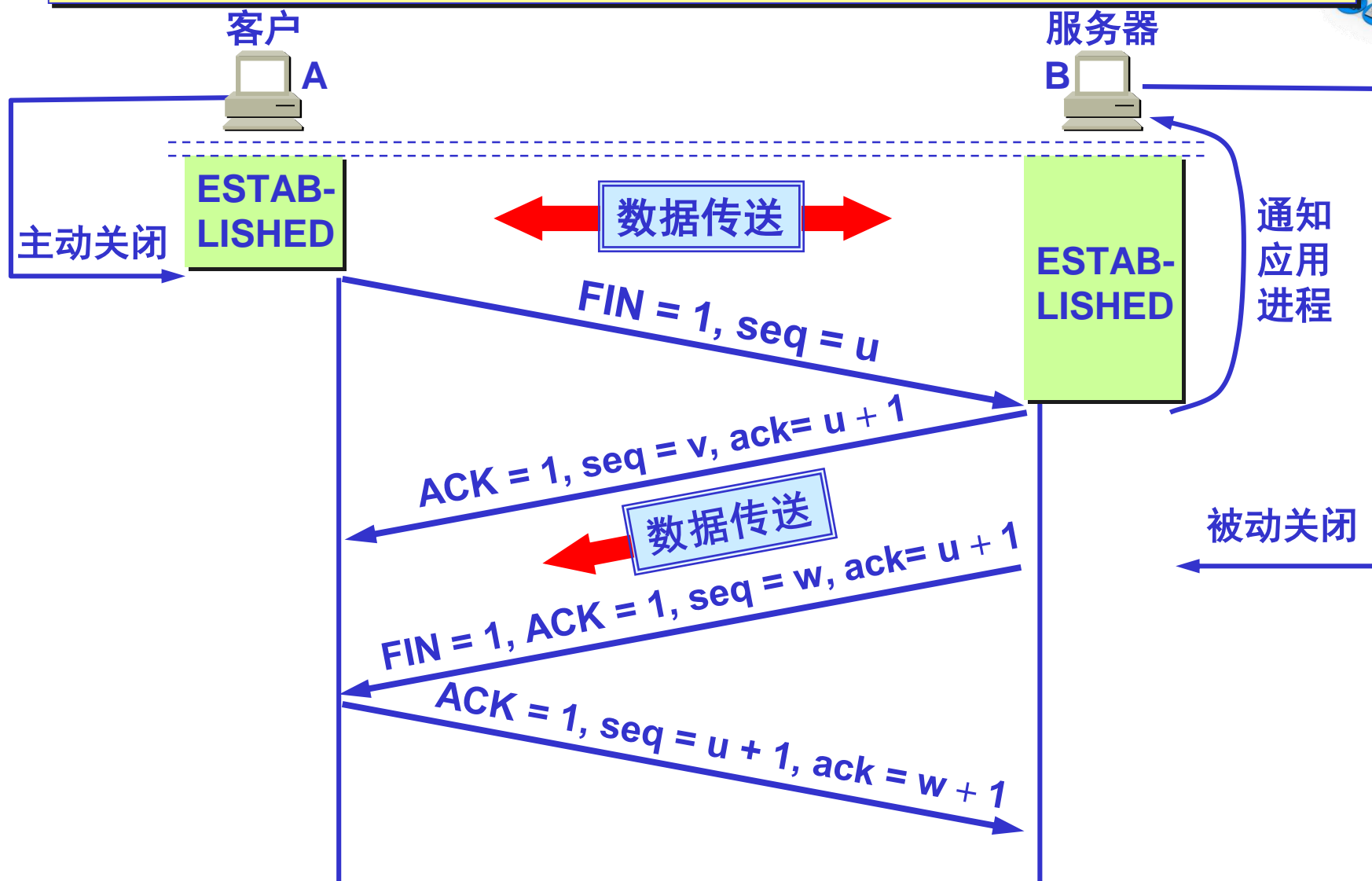
- B 发出确认，确认号  $ack = u + 1$ ，而这个报文段自己的序号  $seq = v$ 。
- TCP 服务器进程通知高层应用进程。
- 从 A 到 B 这个方向的连接就释放了，TCP 连接处于**半关闭**状态。B 若发送数据，A 仍要接收。

# TCP 的连接释放：采用四报文握手



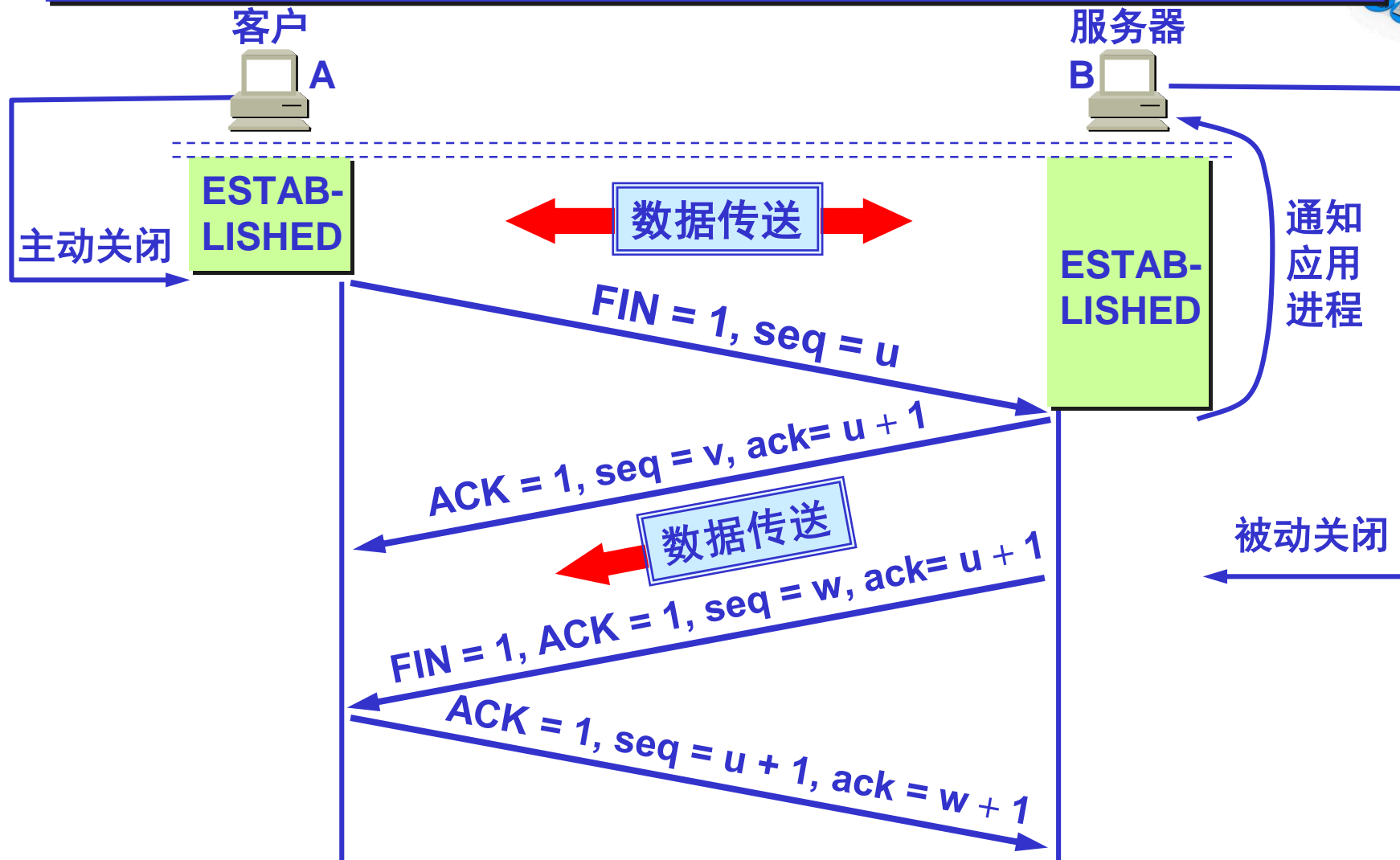
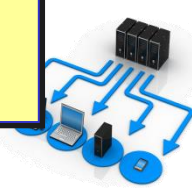
- 若 B 已经没有要向 A 发送的数据，其应用进程就通知 TCP 释放连接。

# 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。

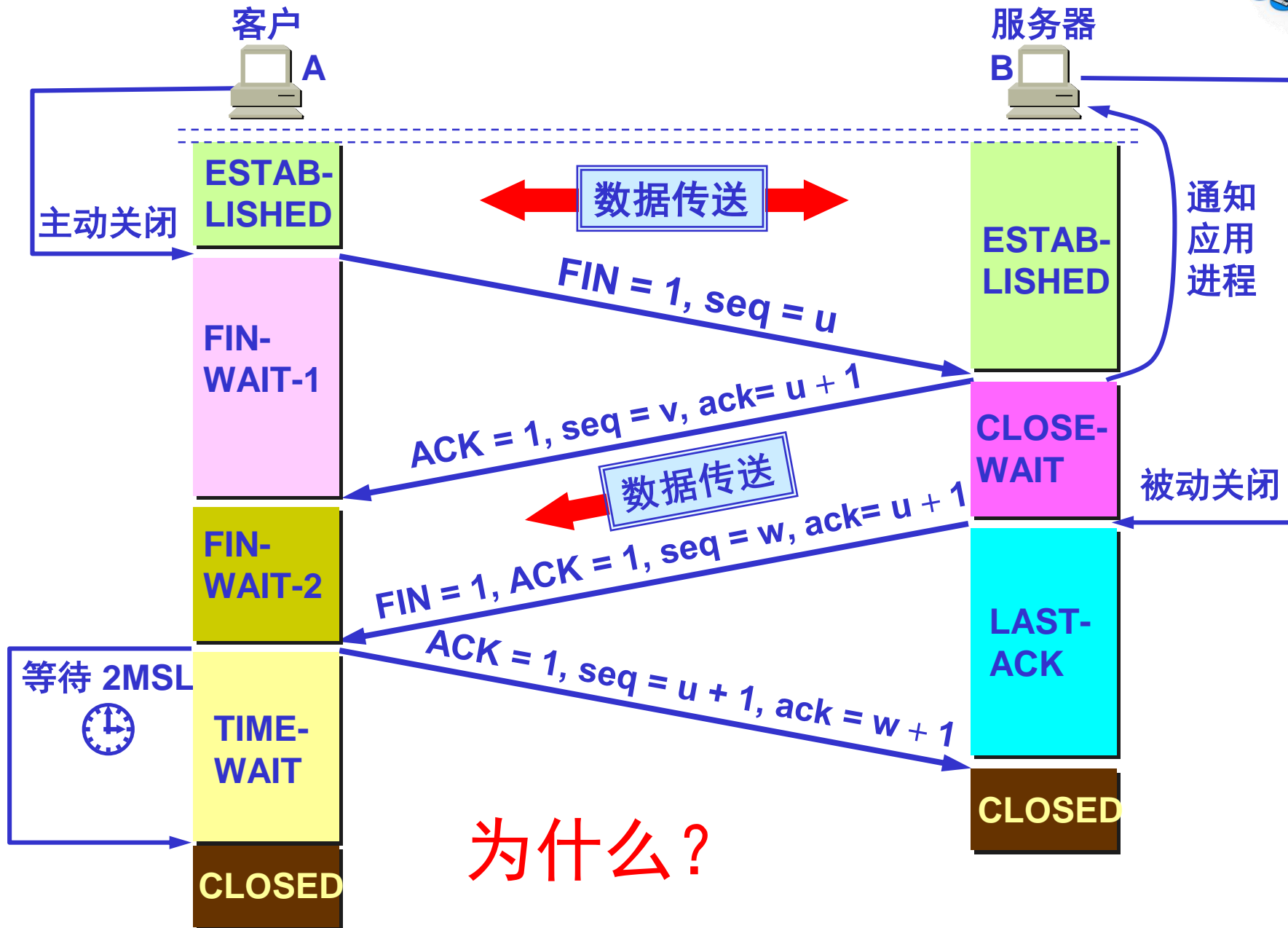
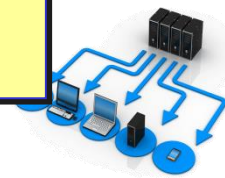
# TCP 的连接释放：采用四报文握手



- 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。

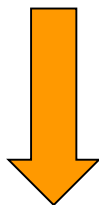


为什么？

# 问题



- 如何正常连接的过程中，客户端或者服务器端突然出现故障，如何处理？



设置计时器

## 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。



# TCP 的有限状态机

