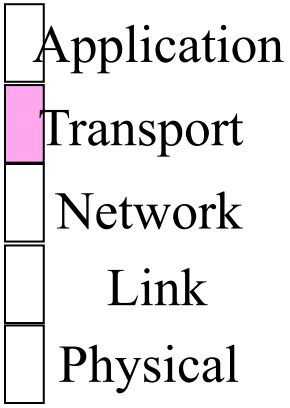
第五章 传输层

林海

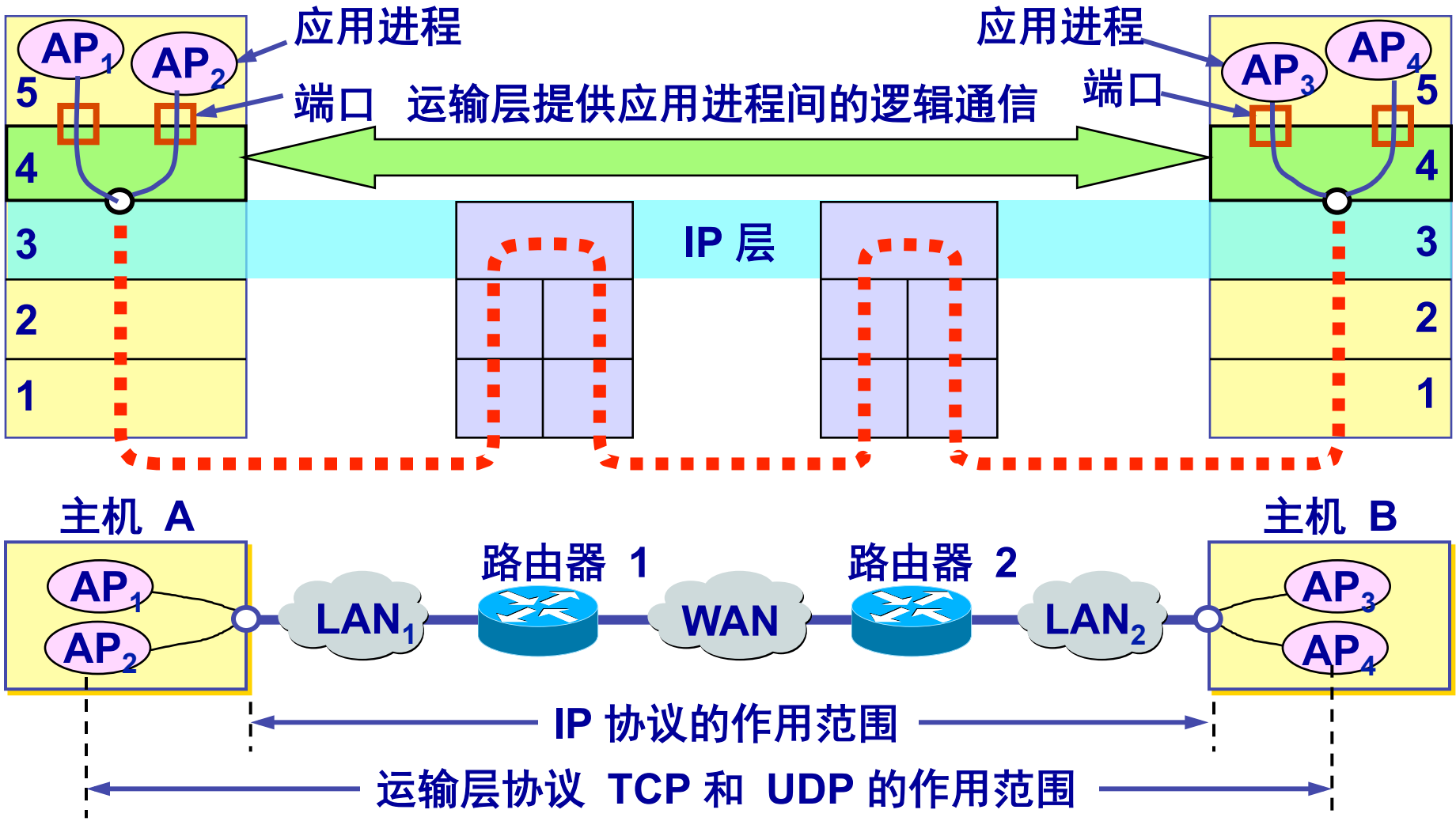
Lin.hai@whu.edu.cn

第 5 章　运输层

- 主要功能：实现端对端传输
  - 可靠性传输
  - 拥塞控制
  - 流量控制



运输层的作用



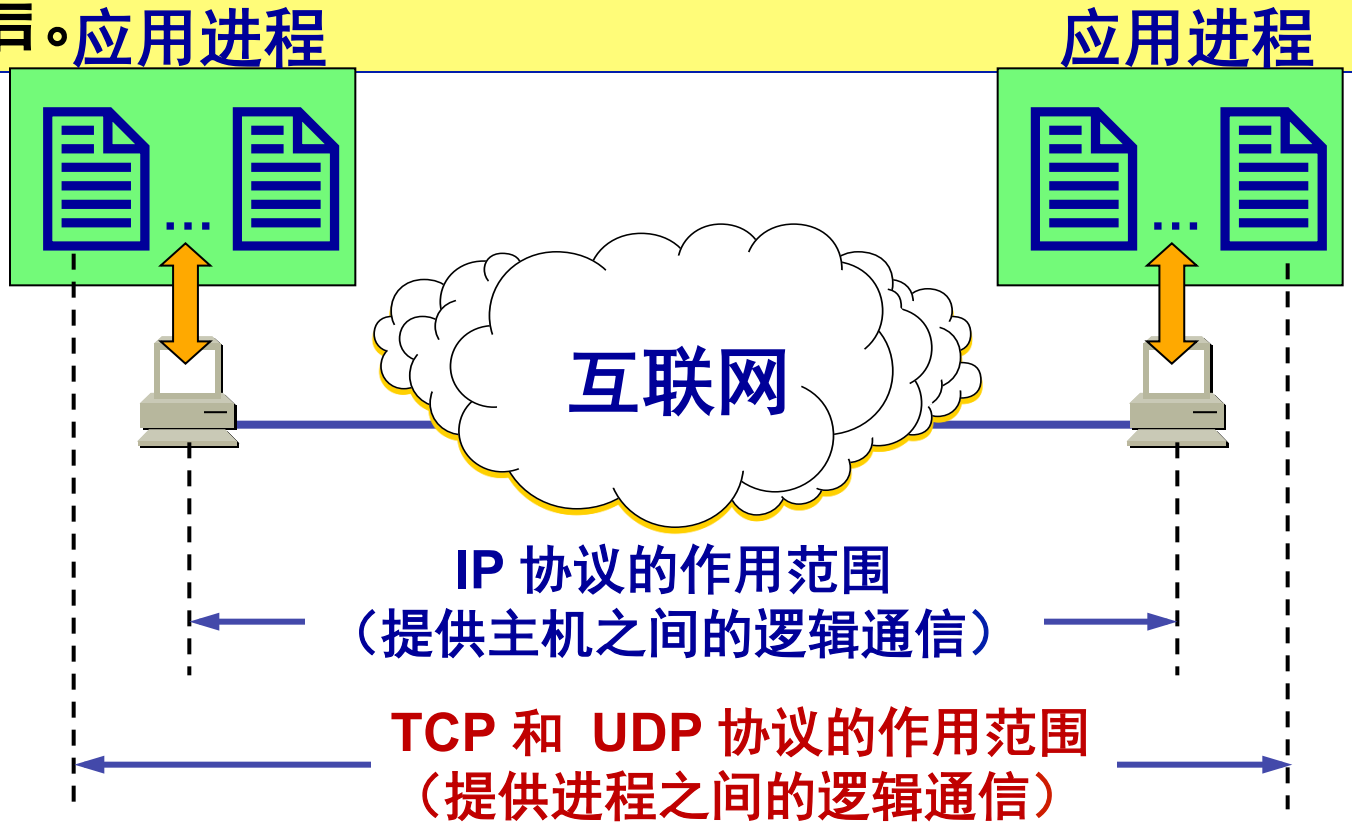
运输层为相互通信的应用进程提供了逻辑通信

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。

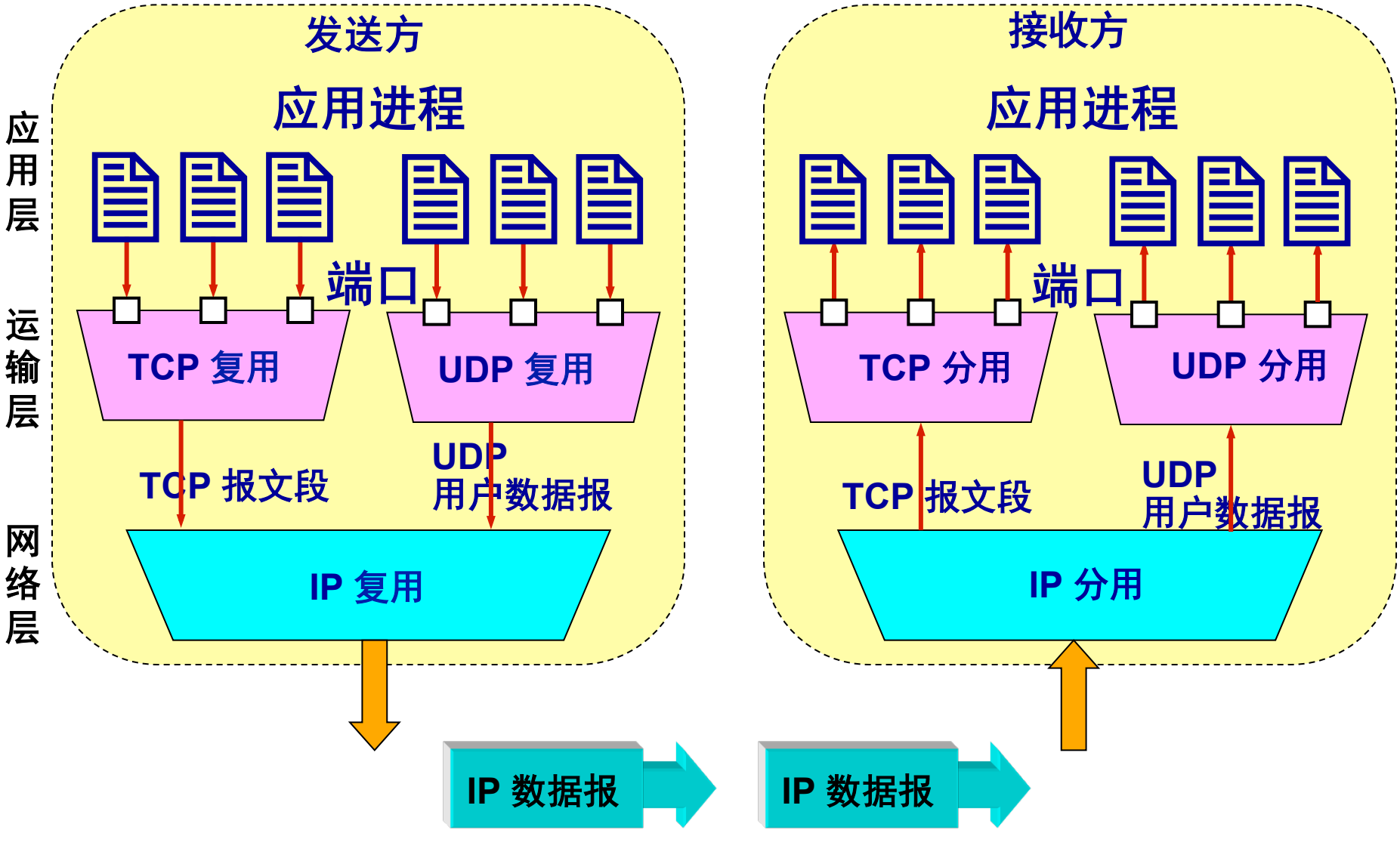


运输层协议和网络层协议的主要区别

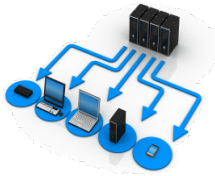
运输层的作用

- 多个应用进
  - 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



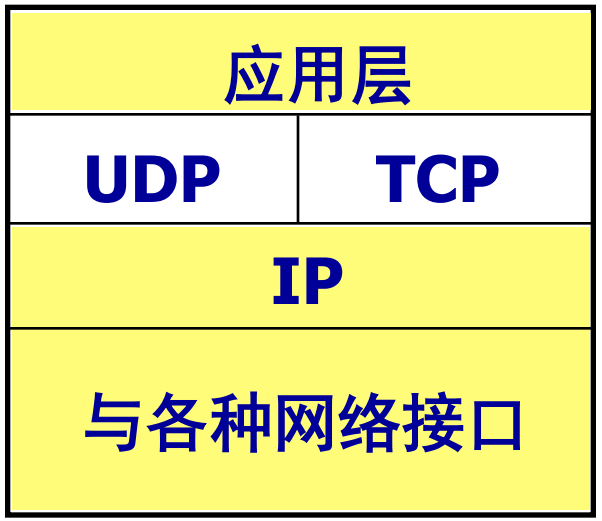
屏蔽作用



- 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

- TCP
  - 面向连接、有确认
  - 全双工的可靠信道。
- UDP
  - 无连接、无确认
  - 不可靠信道。



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。

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。

5.1.3 运输层的端口

- 端口作用（传输层端口）
  - 辨别应用进程
- 和主机上的硬件端口的区别
  - 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。

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

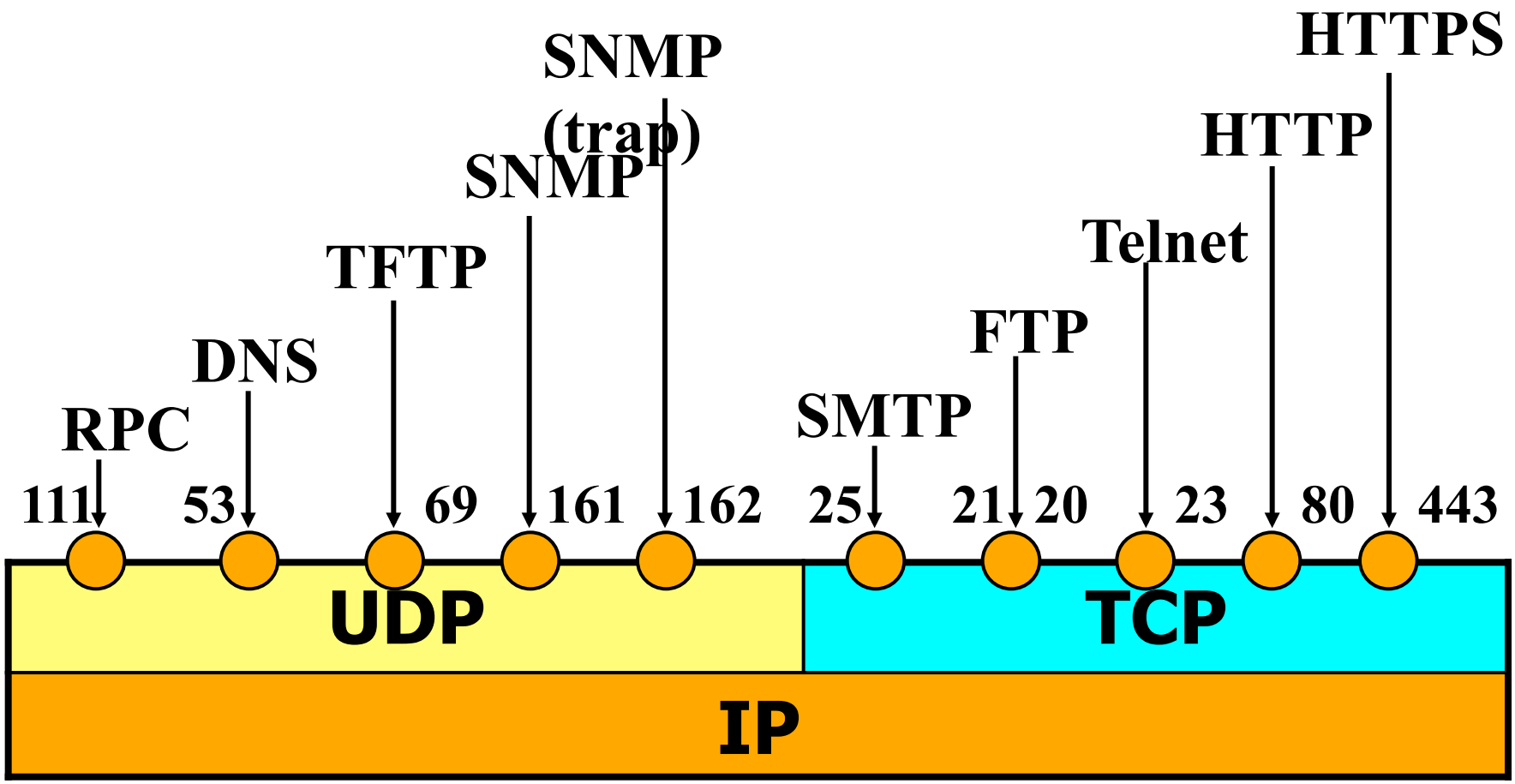
(1) 服务器端使用的端口号

- 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


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

5.2 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。

5.2 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。

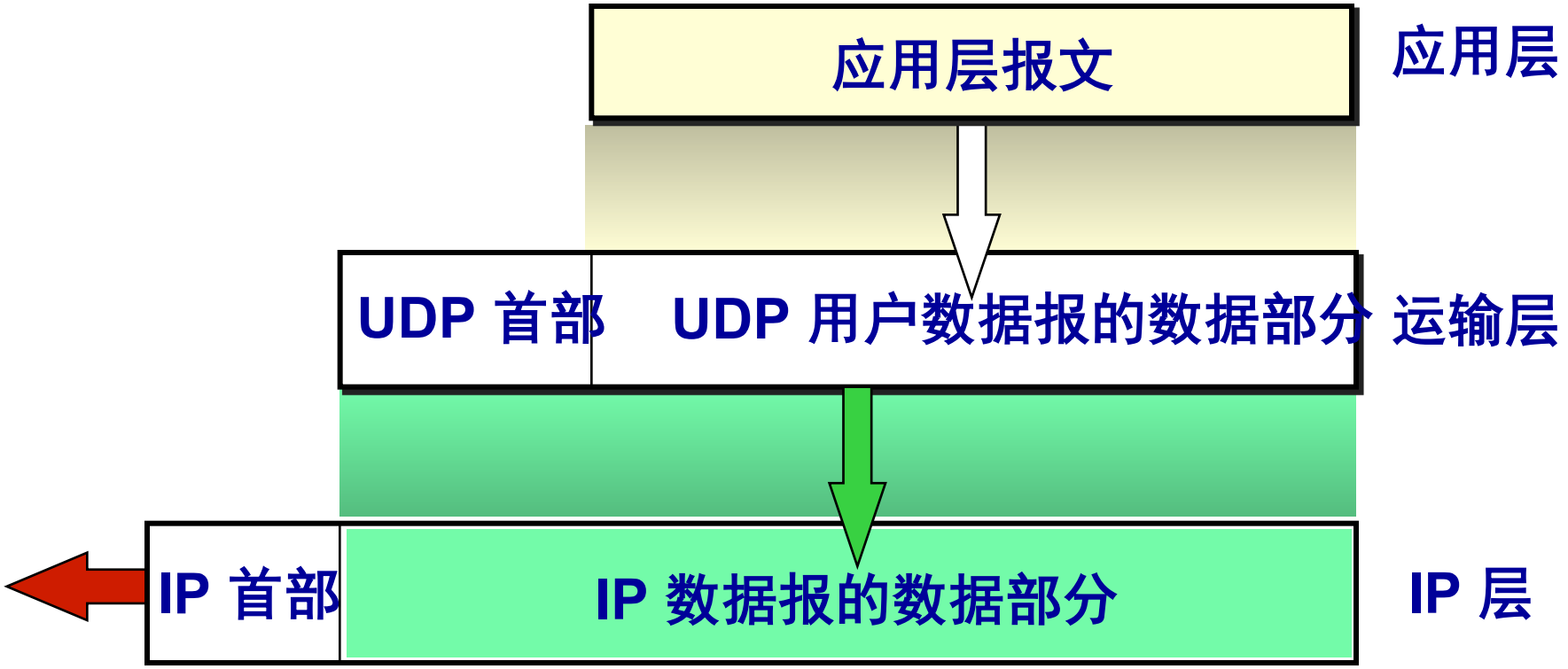
面向报文的 UDP

- 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。

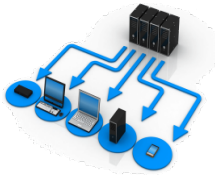
面向报文的 UDP

- 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。

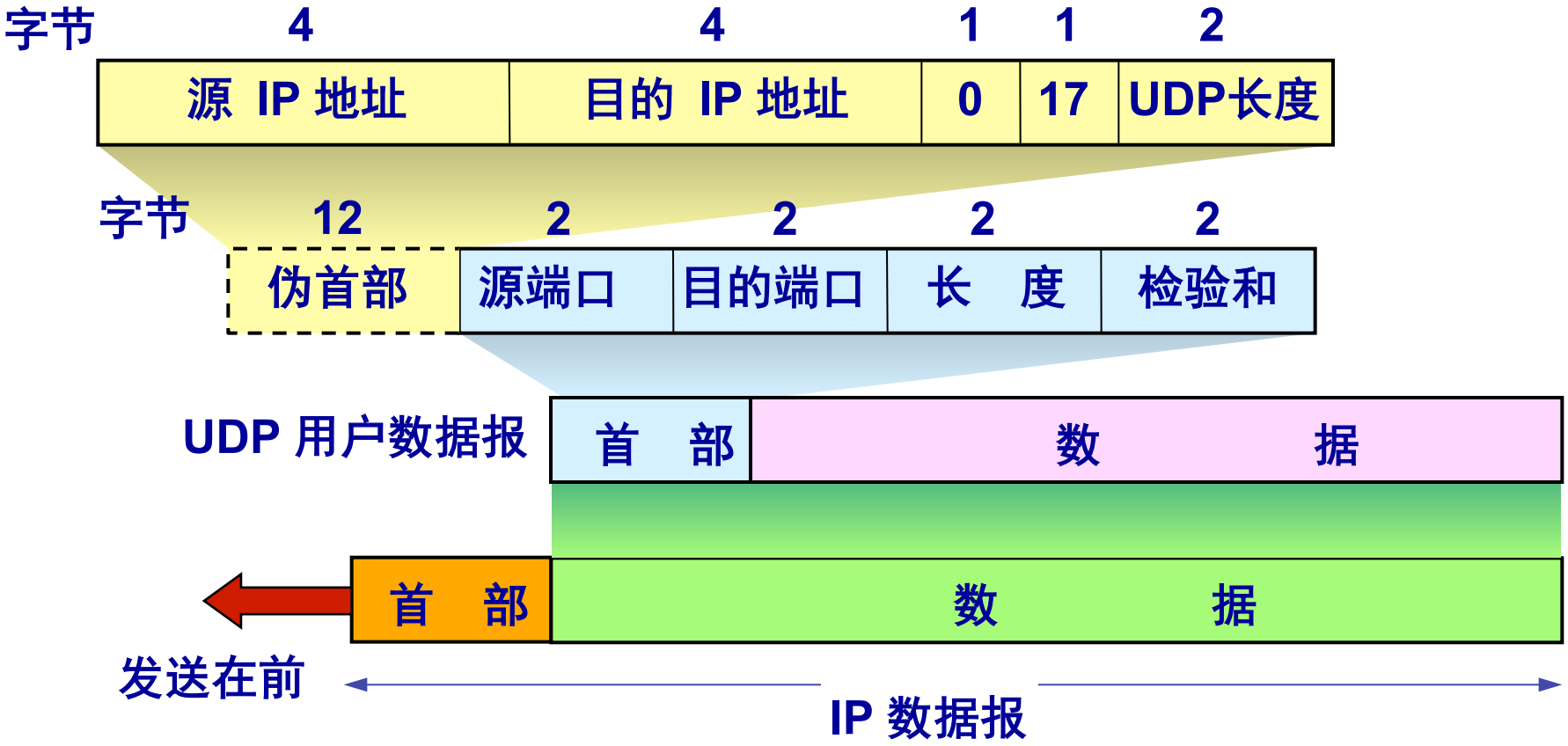
UDP 是面向报文的



5.2.2 UDP 的首部格式

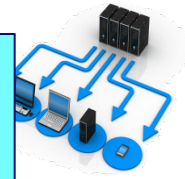


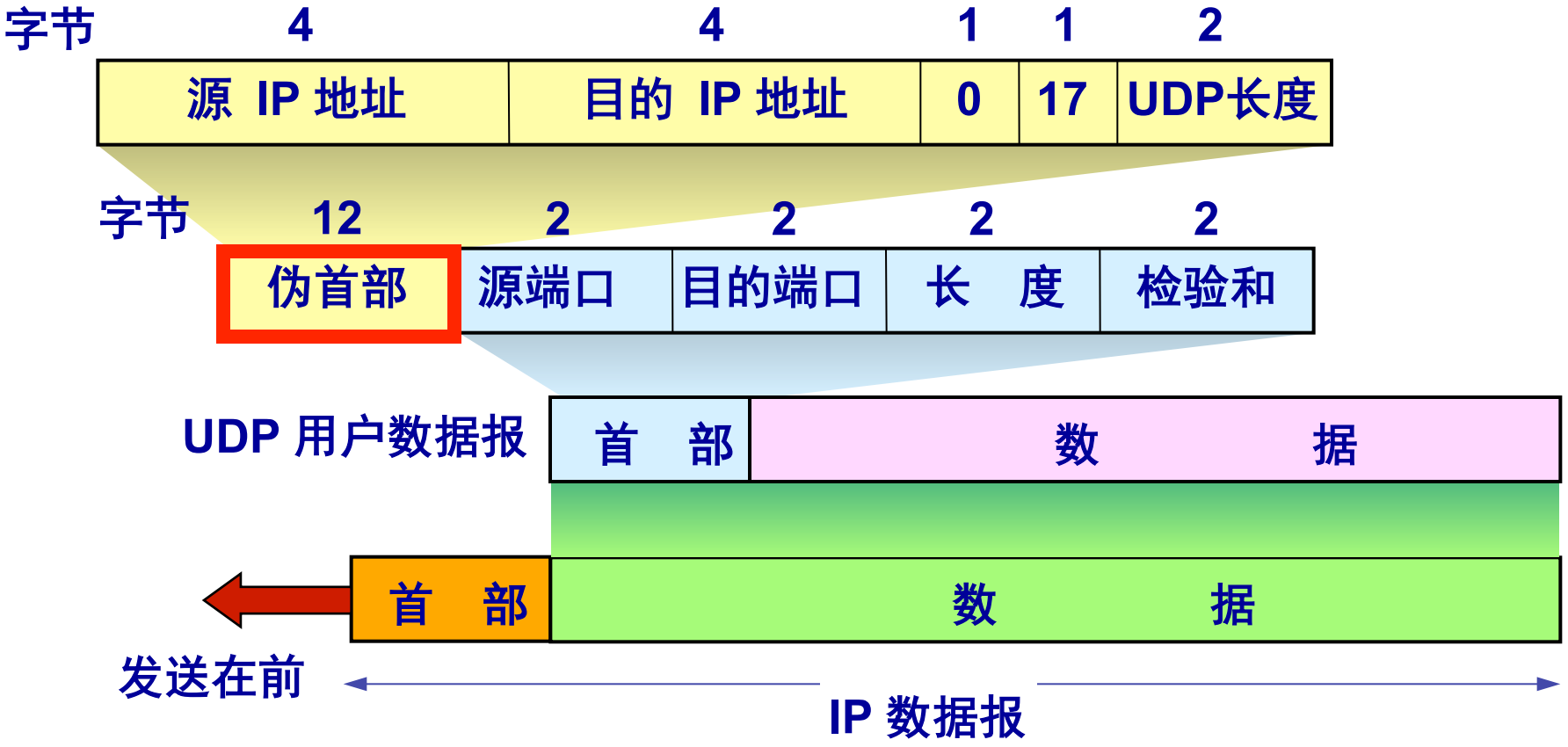
用户数据报 UDP 有两个字段：数据字段和首部字段。首部字段很简单，只有 8 个字节。



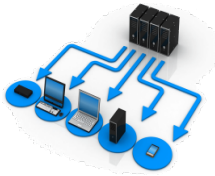
UD P用户数据报的首部和伪首部

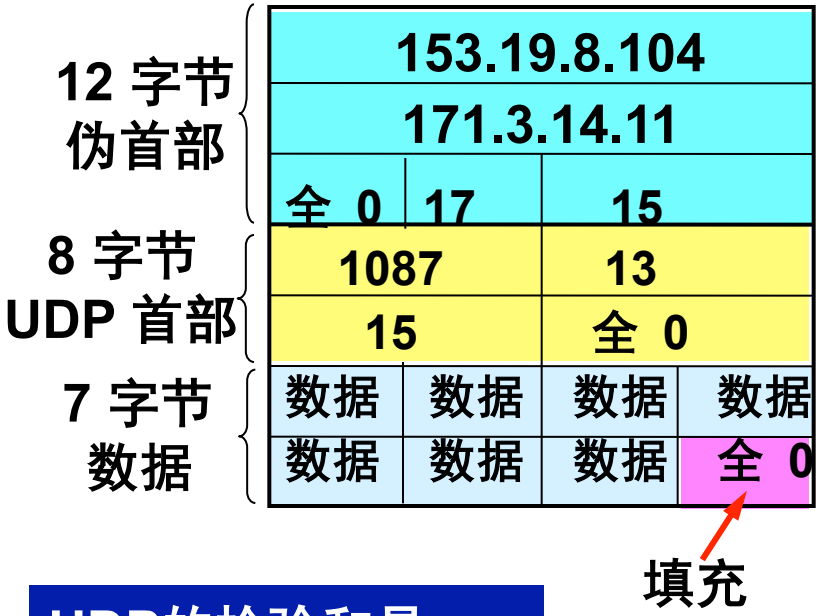
在计算检验和时，临时把“伪首部”和 UDP 用户数据报连接在一起。伪首部仅仅是为了计算检验和。





计算 UDP 检验和的例子





UDP的检验和是把首部和数据部分一起都检验。

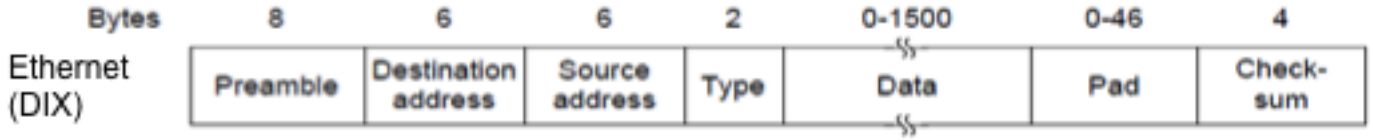
10011001 00010011 → 153.19
00001000 01101000 → 8.104
10101011 00000011 → 171.3
00001110 00001011 → 14.11
00000000 00010001 → 0 和 17
00000000 00001111 → 15
00000100 00111111 → 1087
00000000 00001101 → 13
00000000 00001111 → 15
00000000 00000000 → 0（检验和）
01010100 01000101 → 数据
01010011 01010100 → 数据
01001001 01001110 → 数据
01000111 00000000 → 数据和 0（填充）

按二进制反码运算求和
将得出的结果求反码

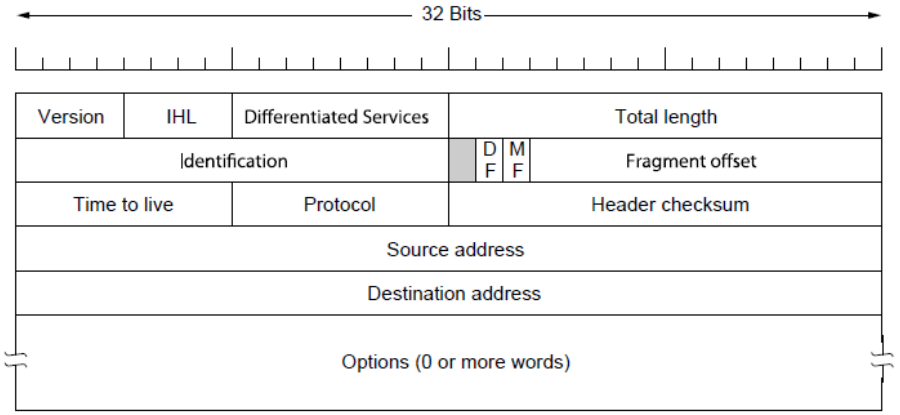
10010110 11101101 → 求和得出的结果
01101001 00010010 → 检验和

Ethernet header:

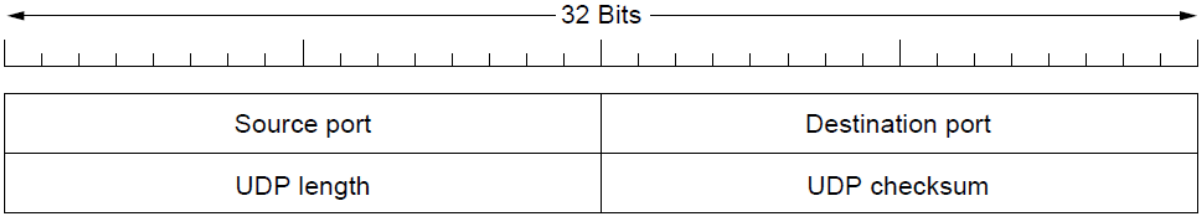
0000 08 18 1a 78 f2 2a 00 06 82 00 39 e4 08 00 45 00
0010 00 c8 ab cd 00 00 ff 11 a0 2d 0a 01 2e 1b 0a 01
0020 2d 0d a0 84 0f a6 00 b4 6c db



IP header:



UDP header:

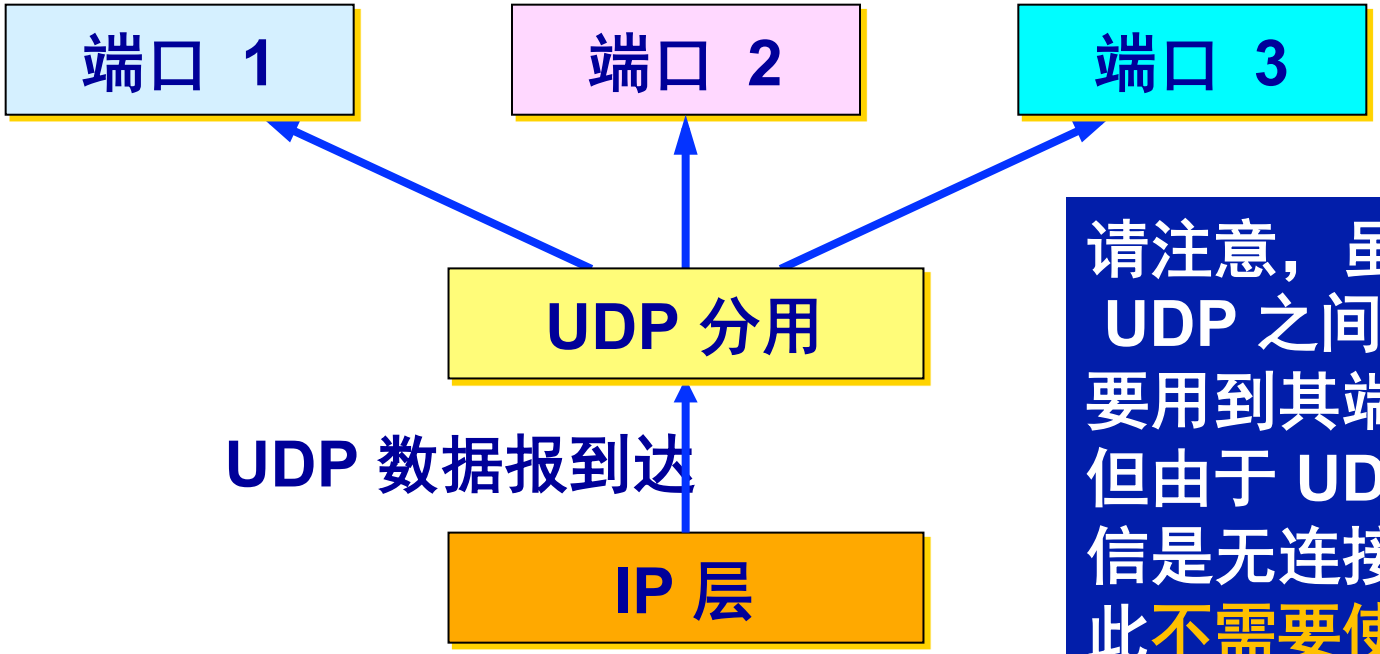


```
0000 08 18 1a 78 f2 2a 00 06 82 00 39 e4 08 00 45 00
0010 00 c8 ab cd 00 00 ff 11 a0 2d 0a 01 2e 1b 0a 01
0020 2d 0d a0 84 0f a6 00 b4 6c db
```

```
▼ Ethernet II, Src: Convedia_00:39:e4 (00:06:82:00:39:e4), Dst: Zte_78:f2:2a (08:18:1a:78:f2:2a)
  ▶ Destination: Zte_78:f2:2a (08:18:1a:78:f2:2a)
  ▶ Source: Convedia_00:39:e4 (00:06:82:00:39:e4)
    Type: IP (0x0800)
▼ Internet Protocol Version 4, Src: 10.1.46.27 (10.1.46.27), Dst: 10.1.45.13 (10.1.45.13)
    Version: 4
    Header Length: 20 bytes
  ▶ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00: Not-ECT (Not ECN-Capable Transport))
    Total Length: 200
    Identification: 0xabcd (43981)
  ▶ Flags: 0x00
    Fragment offset: 0
    Time to live: 255
    Protocol: UDP (17)
  ▶ Header checksum: 0xa02d [correct]
    Source: 10.1.46.27 (10.1.46.27)
    Destination: 10.1.45.13 (10.1.45.13)
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
▼ User Datagram Protocol, Src Port: 41092 (41092), Dst Port: pxc-spvr (4006)
    Source Port: 41092 (41092)
    Destination Port: pxc-spvr (4006)
    Length: 180
  ▶ Checksum: 0x6cdb [validation disabled]
▶ Data (172 bytes)
```

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。

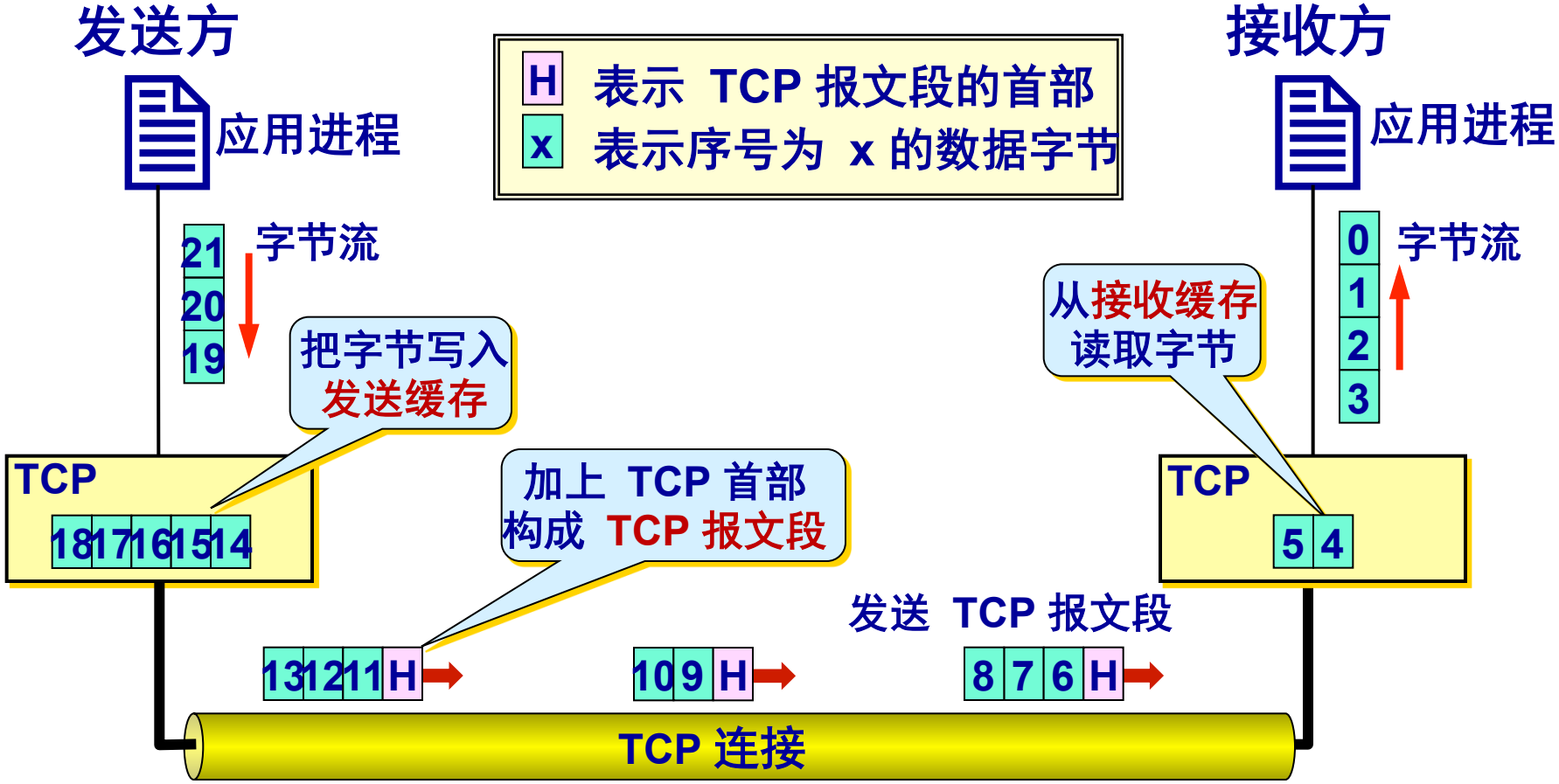
5.3 TCP 最主要的特点

- 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

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

- 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。

传输层原语: Java (android) example

客户端:

Socket的建立:

```
socket=new Socket("192.168.0.167"
,12345);
in = socket.getInputStream();
out=socket.getOutputStream();
```

发送和接收:

```
in.readLine()
out.println(message);
```

Socket 的关闭:

```
in.close(); out.close(); socket.close();
```

服务端:

Socket的建立:

```
ServerSocket server=new
 ServerSocket(12345);
Socket client=server.accept();
in = client.getInputStream();
out=client.getOutputStream();
```

发送和接收:

```
in.readLine()
out.println(message);
```

Socket的关闭:

```
in.close(); out.close(); client.close();
```

5.4 可靠传输

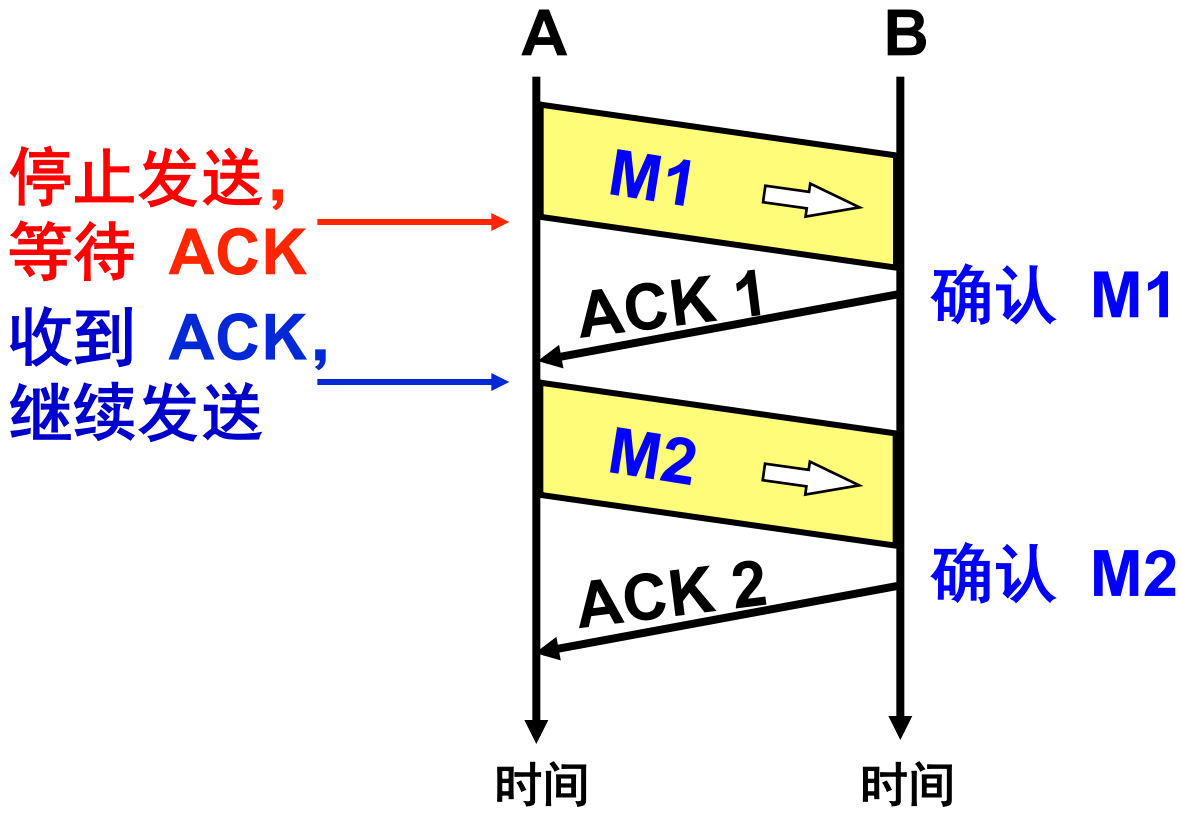
- 计算机网络特点：不可靠的
  - (1) 传输信道会产生差错。
  - (2) 接收方也许来不及处理发送方的数据。
- 但有些应用又有可靠性的需求
- 如果解决这个矛盾
  - 传输层提供可靠性
  - 如何实现：要求每个数据包需要回复ACK
- 简单的可靠传输
  - 停止等待协议

5.4.1 停止等待协议

- “停等协议”
  - 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。
  - 称为自动重传请求 ARQ (Automatic Repeat reQuest)。
- 理想情况下的停等协议与有差错情况下的停等协议

1. 无差错情况

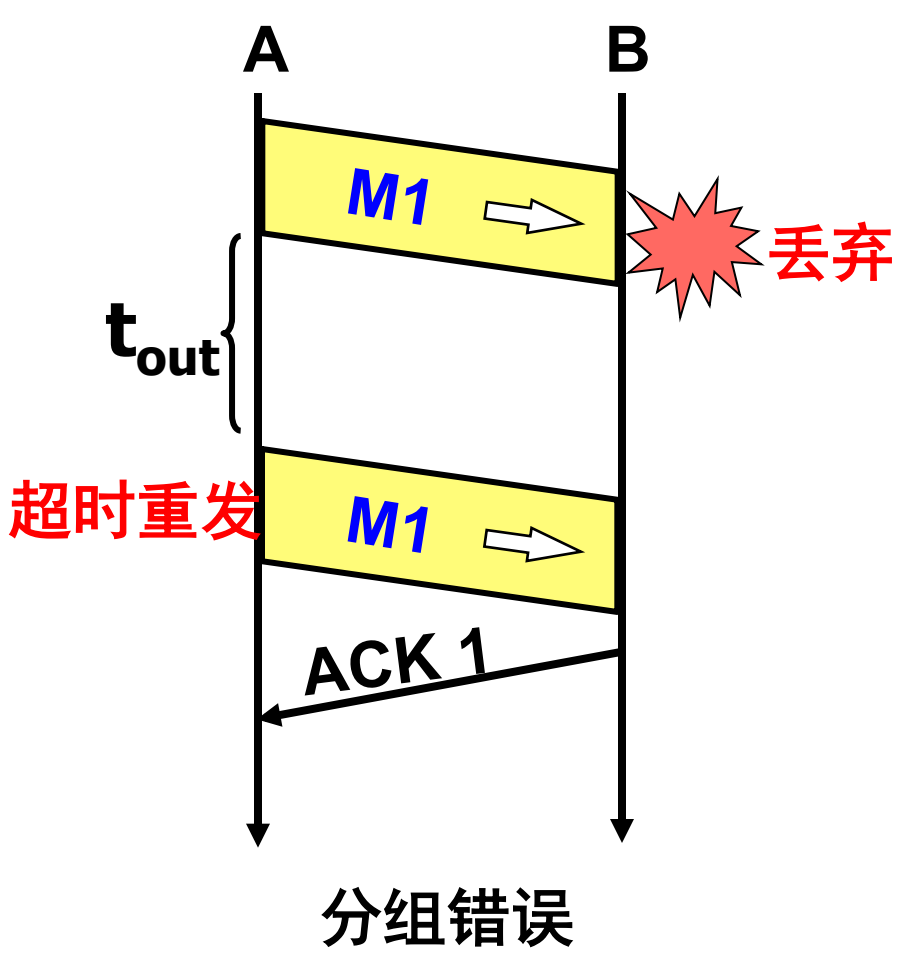
A 发送分组 M1，发完就暂停发送，等待 B 的确认 (ACK)。B 收到了 M1 向 A 发送 ACK。A 在收到了对 M1 的确认后，就再发送下一个分组 M2。



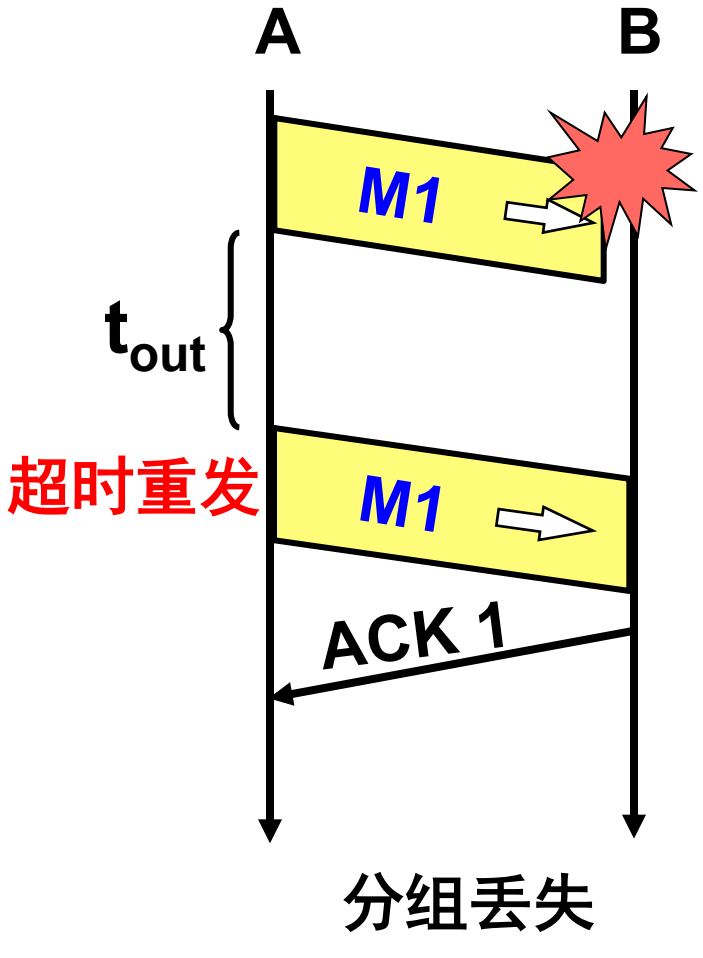
2. 出现差错（数据包）

- 数据包差错
  - 丢失
  - 检测出错
- 流程
  - 接收方无ACK
  - 发送方：超时重传
    - 每一个已发送的分组都设置了一个超时计时器。
    - 只要在计时器到期之前收到了相应的确认，就撤销该超时计时器，否则重传。

2. 出现差错



分组错误

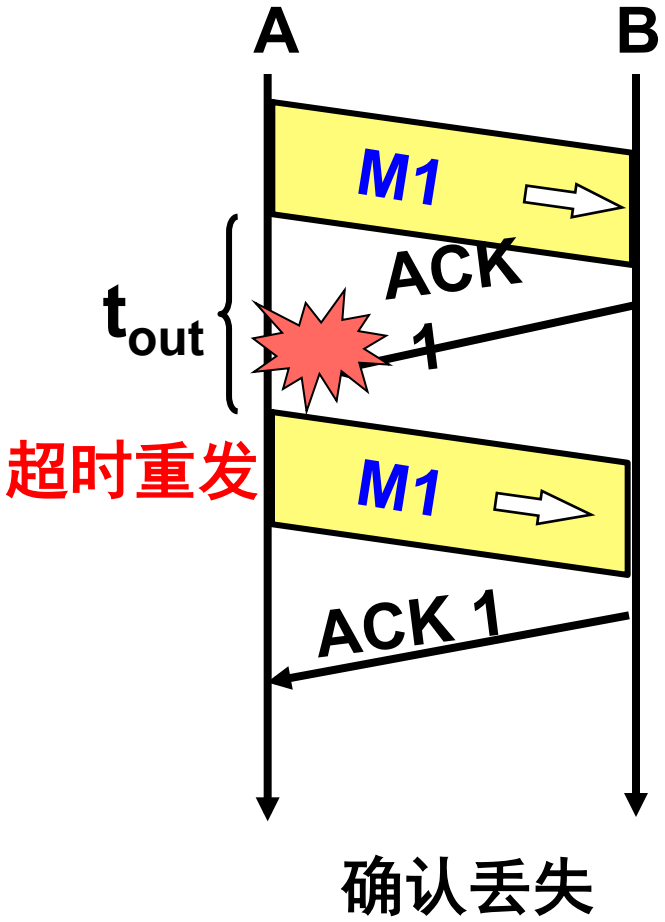


分组丢失

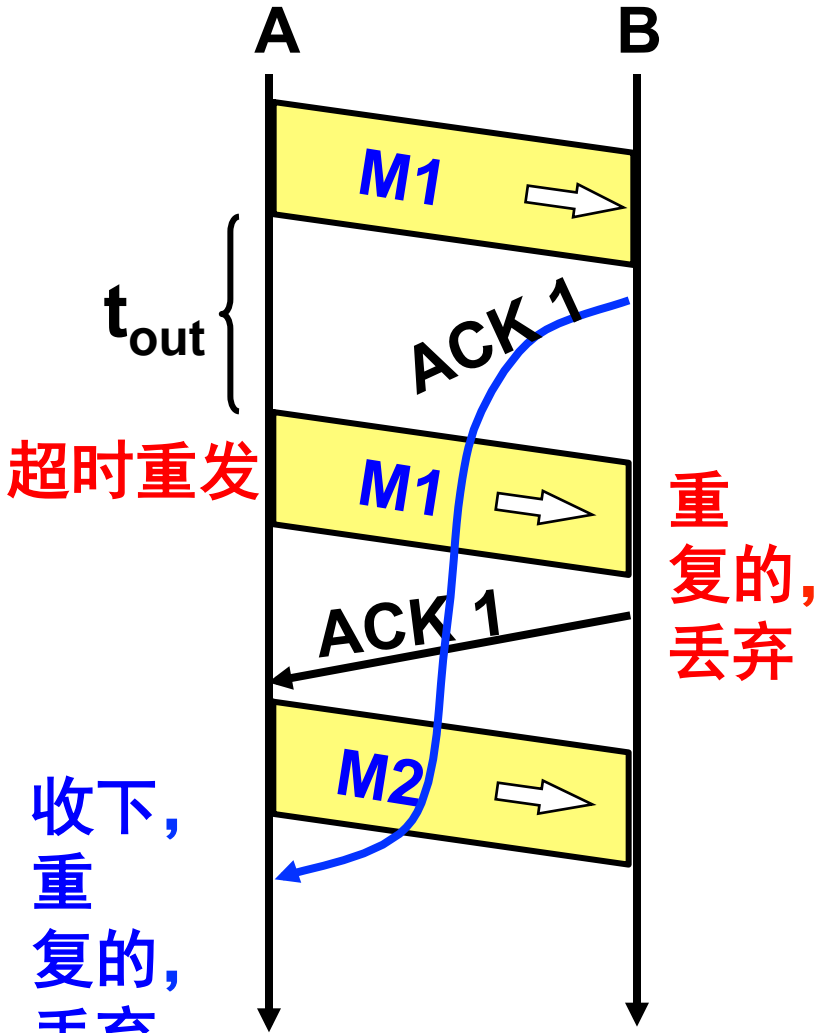
3. 出现差错（ACK）

- ACK出差错
  - ACK丢失
    - 发送方重传数据段。
    - 接收方收到重发的数据，丢弃，回复ACK
  - ACK迟到
    - 发送方重发，
    - 接收方收到重复的数据，丢弃，并回ACK。
    - 发送方会收到重复的ACK，丢弃。

3. 确认丢失和确认迟到



确认丢失



确认迟到

请注意

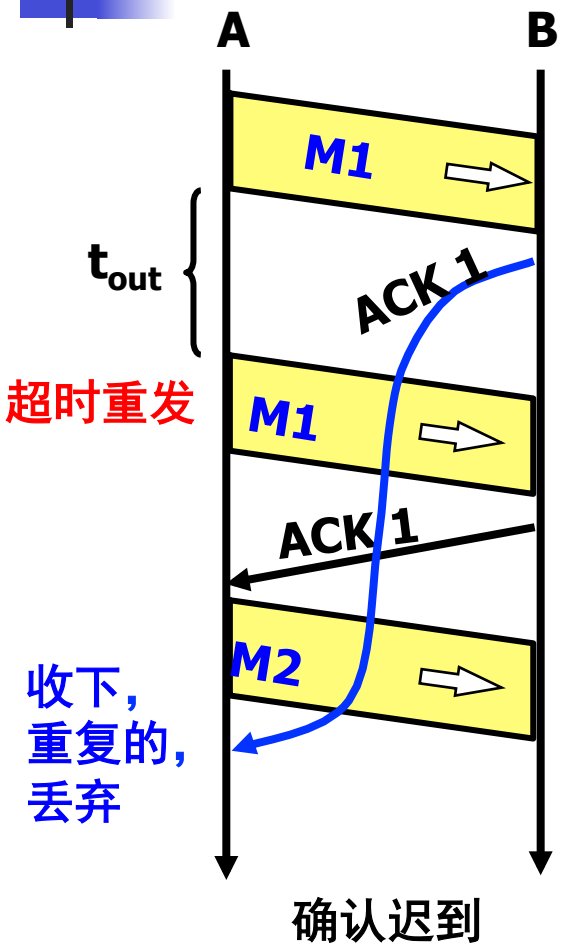
- 在发送完一个分组后，必须暂时保留已发送的分组的副本，以备重发。
- 分组和确认分组都必须进行编号。
- 超时计时器的重传时间应当比数据在分组传输的平均往返时间更长一些。
- 停等协议一般用于链路层，而不是传输层（如果停等协议用在链路层，则只用一个bit的编号即可，用在传输层不行）

滑动窗口的概念

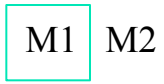
发送方和接收方保持一个窗口用于接收和发送数据

- Ex: window size is 1.

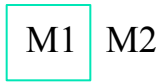
停等协议—滑动窗口



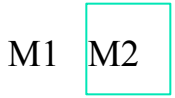
A:



要发送数据包1，等待ACK

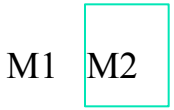


超时重发

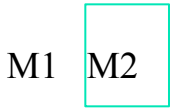


收到ACK，发送M2，等待M2的ACK。此时收到M1的ACK，不是等待的，丢弃

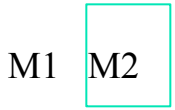
B:



收到M1，发送ACK，等待M2



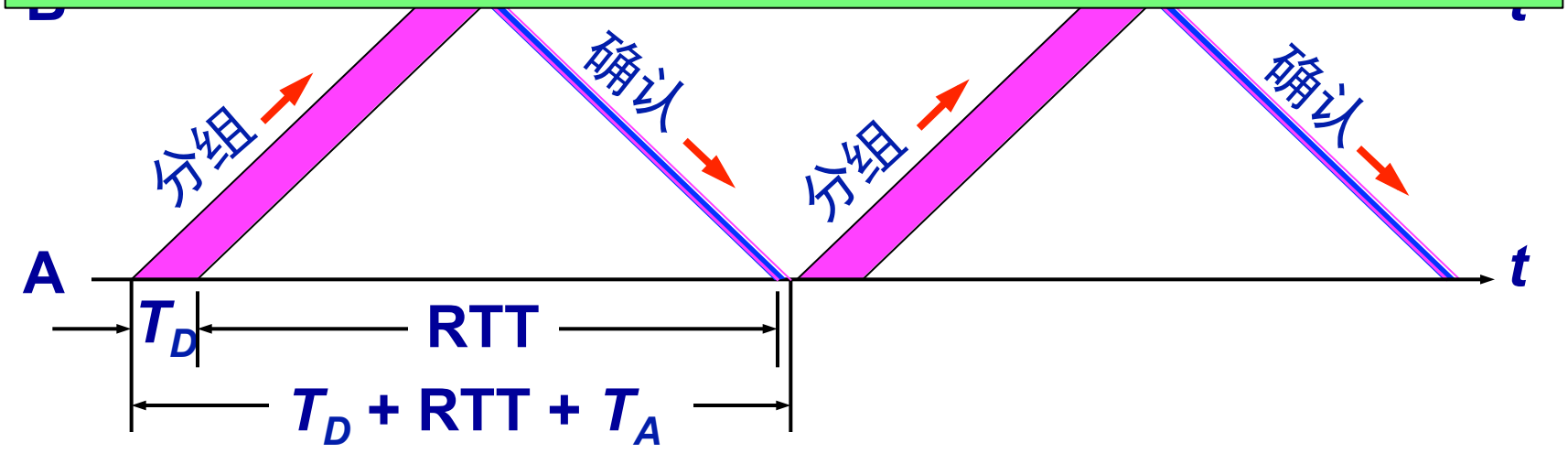
等待M2，收到M1，丢弃，并发送最后收到数据的ACK（M1）



收到M2，发送ACK

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

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

4. 信道利用率

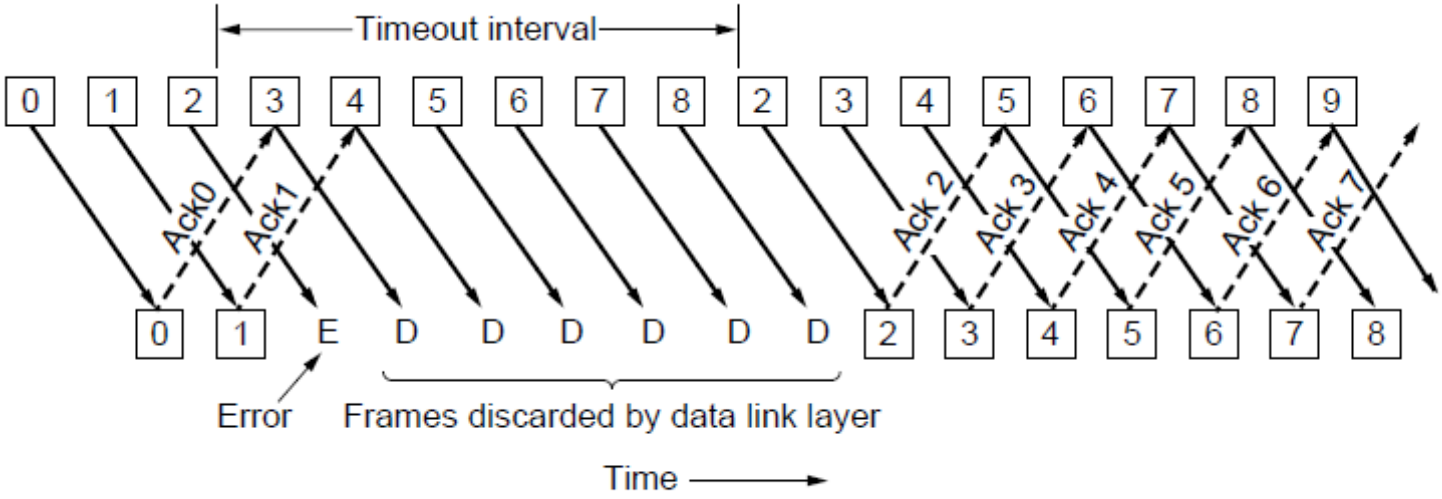
- 一个50 kbps 带宽的信道，500 msec 来回延迟. 那么发送一个1000 bits的数据：
  - 20ms：数据的发送时间
  - 270ms 后, 整个数据到达接收方
  - 520ms 后, ack 到达发送方
  - 所以大多数时间 (500/520), 发送方啥都没做。带宽利用率为20/520=4%

- 问题: 对这么一个信道, 什么大小的数据能使带宽的利用率达到50%？
- Solution (Go-Back-n): 发送方不必非等到ACK才能发下一个数据
  - 加大发送方的窗口，即让发送窗口大于1

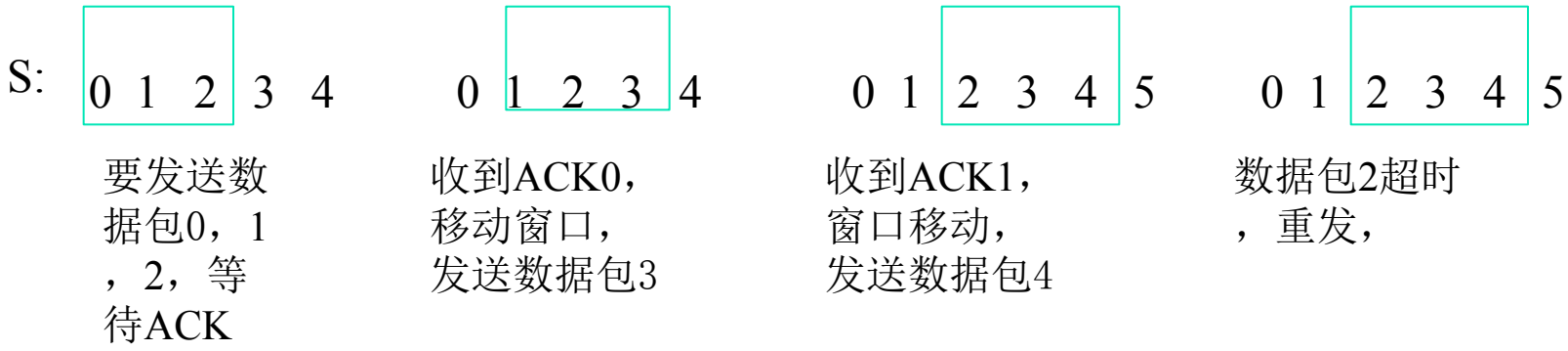
回退n（Go-Back-N）

在go-back-n机制中:

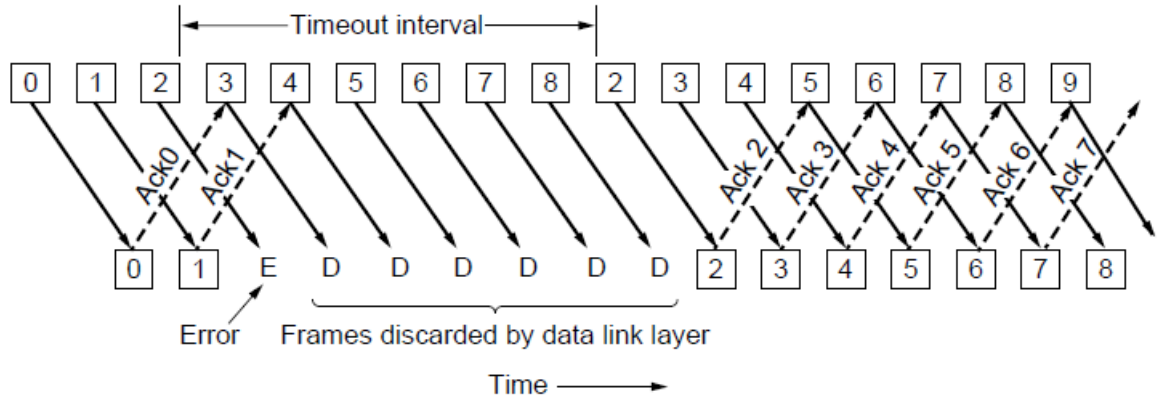
- 接收方丢弃所有非顺序到达的数据
- 发送方的某一数据times out后，重发所有此包以及所有的后续数据



发送方的窗口大于1（下面的例子用3），而接收方的窗口还是1



接后



S: 0 1 2 3 4 5

数据包2超时，重发，

0 1 2 3 4 5

收到ACK2，并重发3，4，5…

D: 0 1 2 3 4

收到数据包2，回ACK2，等待数据包3

依次收到数据包3，4，5…

Go-Back-N

使用go-back-n机制，假设窗口大小为MAX_WIN，那么sequence number的最大值可以设置为MAX_WIN吗？

No, 假设 MAX_WIN=8 (1,2…8)

1. 发送方发送数据 1,2…8
2. 8号数据的piggybacked ACK返回到发送方
3. 发送方发送另外 8 个数据 (1,2…8)
4. 之后，8号数据的另外一个piggybacked ACK 也返回
5. 能确认所有的传输的正常吗？
6. 不能，因为如后传的8个数据都丢失，也是返回8号数据的ACK

Solution: 发送方能发送的最大数目为 MAX_WIN+1，或者说窗口大小为最大数目MAX_SEQ-1

注：Piggybacking总是ACK按顺序到达最后一个帧

Go-Back-N

在go-back-n机制中，如出现错误，那么错误数据后的所有数据都要被重传，所以效率也不是很高

- Solution: 只重传有错的数据

Selective Repeat

两个窗口:

- Sending windows: at sender side for sending frames, which starts from 0 and grows to some predefined maximum
- Receiving window: at receiver side for accepting frames, which is fixed in size and equal to a predefined maximum

选择重传（Selective Repeat）

Question one: 怎么可以让发送方在timeout之前就知道一个数据丢了？

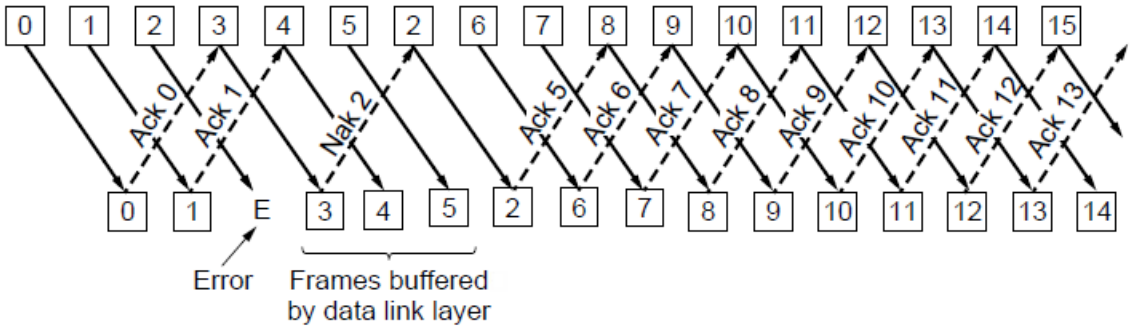
- NAK (negative ack) 机制：如果接收方发现某个数据丢了，就立即发送一个NACK，收到NACK后，发送方就立即重发丢失的数据。NACK通常只发一次，那NAK丢了怎么办？

Question two: 是不是确实需要对每个数据都回复ACK？

- Cumulative ack (累积确认) indicates highest in-order frame

Question three: 通常selective Repeat使用Piggybacks 来回复ACK. 那么如果没有数据来做piggyback呢？

- Set a auxiliary timer for ACK, if no data is sent out after this timer, a ACK packet is sent out



选择重传-滑动窗口

发送方和接收方的窗口都大于1（以下例子为3）

S: 0 1 2 3 4

要发送数据包0，1，2，等待ACK

0 1 2 3 4

收到ACK1，移动滑动窗口，发送3，4

0 1 2 3 4

收到Nak2，重复数据包2

0 1 2 3 4 5 6 7

收到ACK4，移动窗口，发送数据包包5，6，7，

D: 0 1 2 3 4

等待数据包0，1，2

0 1 2 3 4

收到数据包0，1并发送（ACK0）ACK1，等待数据包2，3，4，

0 1 2 3 4

收到数据包3，发送NAk2，之后收到数据包4

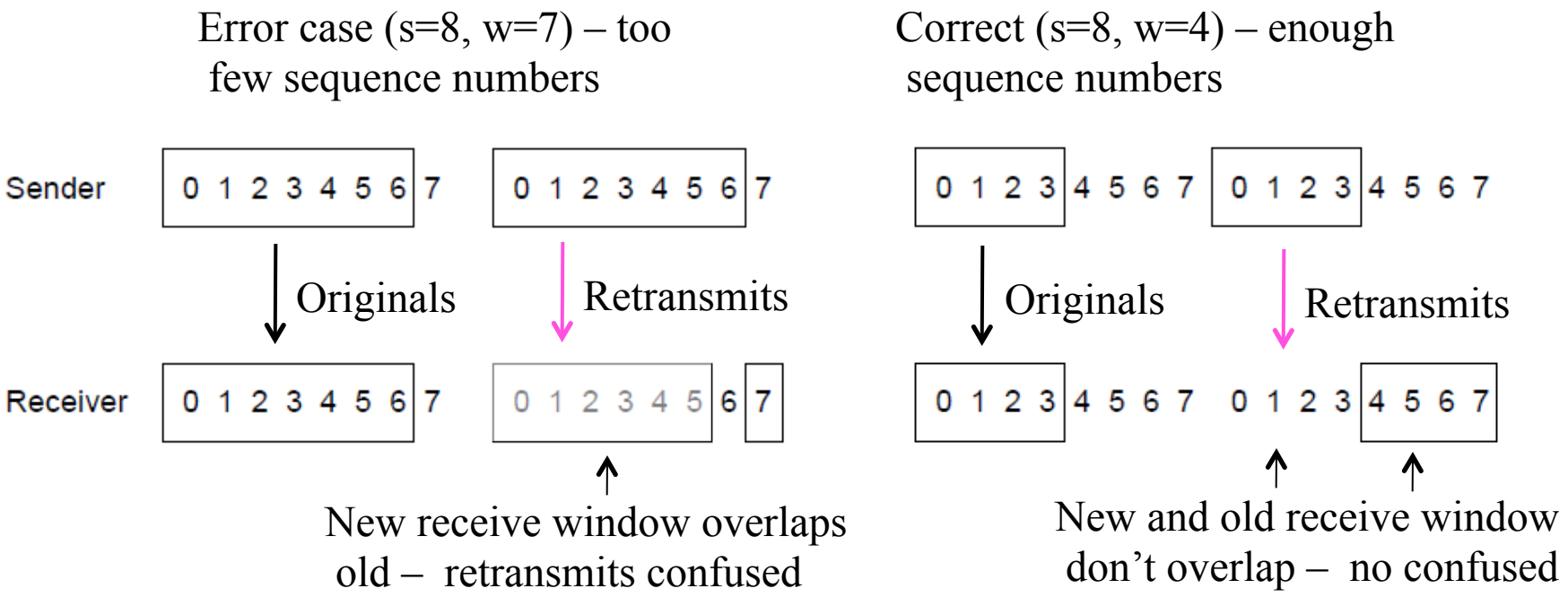
0 1 2 3 4 5 6 7

收到数据包2，发送ACK4，并移动窗口

Selective Repeat（链路层）

在go-back-n中有最大发送数据数目的限制（MAX_SEQ-1），那么seletctive repeat有相同的限制吗？

- Solution: 发送窗口的大小为（MAX_SEQ+1）/2



例子（以链路层为例）

在一个1Mbps的信道上发送长度为10000位的数据（帧），该信道的传播延时为270ms，ACK为1000位，序号使用了4位，试问在下面的协议中，可获得的最大信道利用率分别为多少？（1）停等式 （2）回退n （3）选择重传

答：发送帧需要：10ms，在280ms后，帧到达对方，之后在281ms后，对方发出ACK，551ms后，ACK到达。

（1）10/551=1.8%

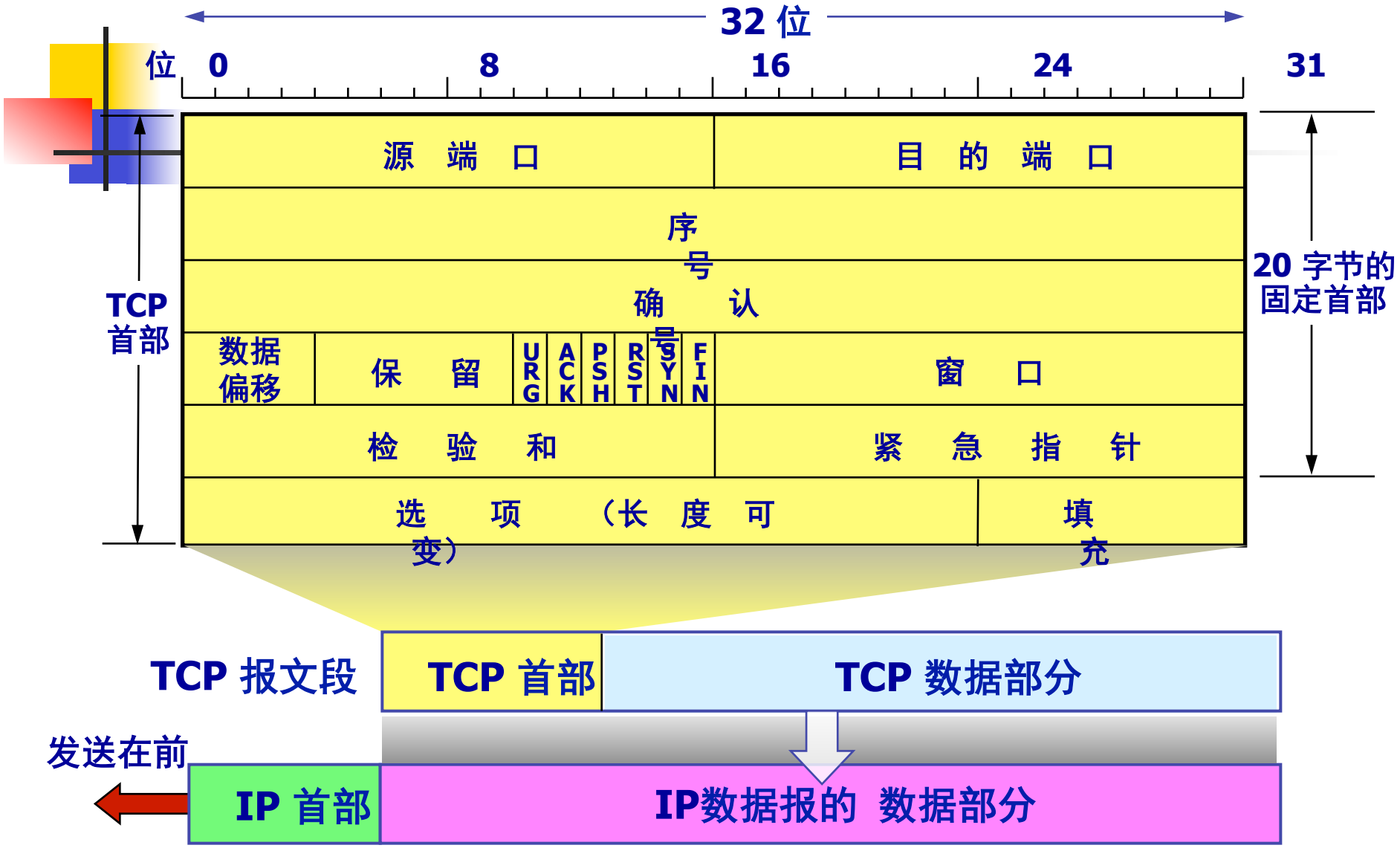
(2)150/551=27.2%

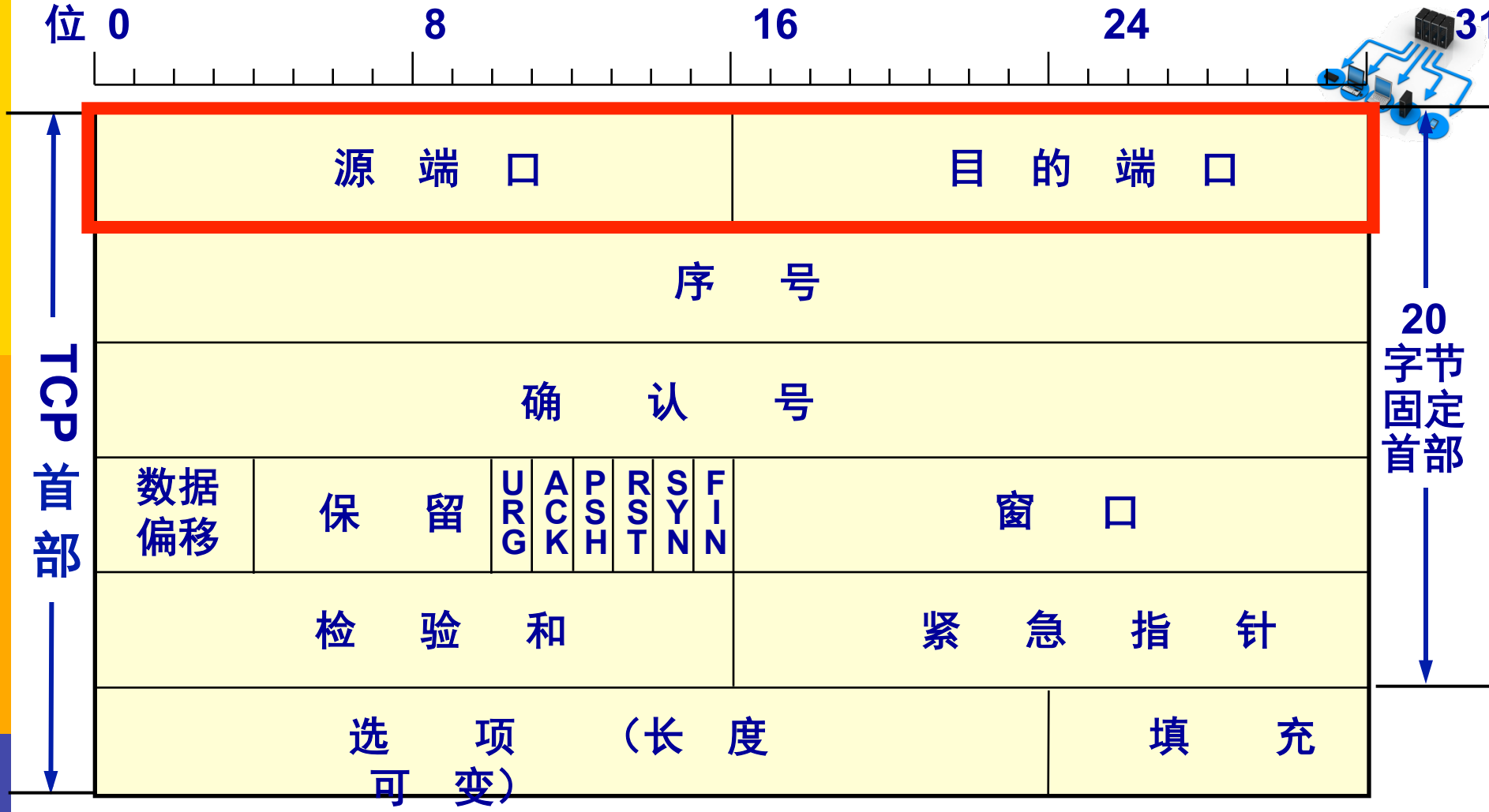
(3)80/551=14.5%

5.5 TCP 报文段的首部格式

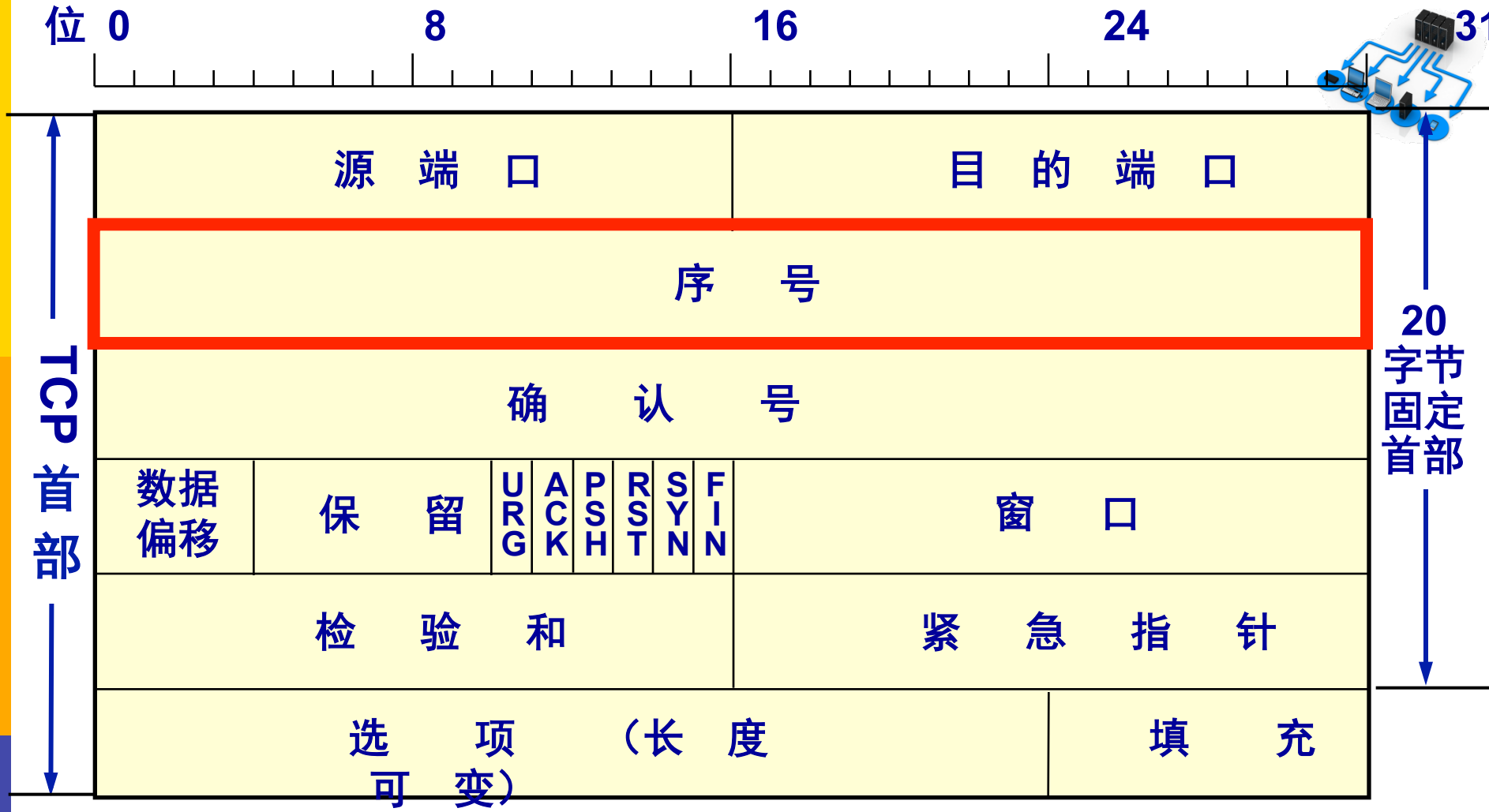
- TCP 报文段首部的前 20 个字节是固定的，后面有 4*n* 字节是根据需要而增加的选项 (*n* 是整数)。因此 TCP 首部的最小长度是 20 字节。

TCP 报文段的首部格式

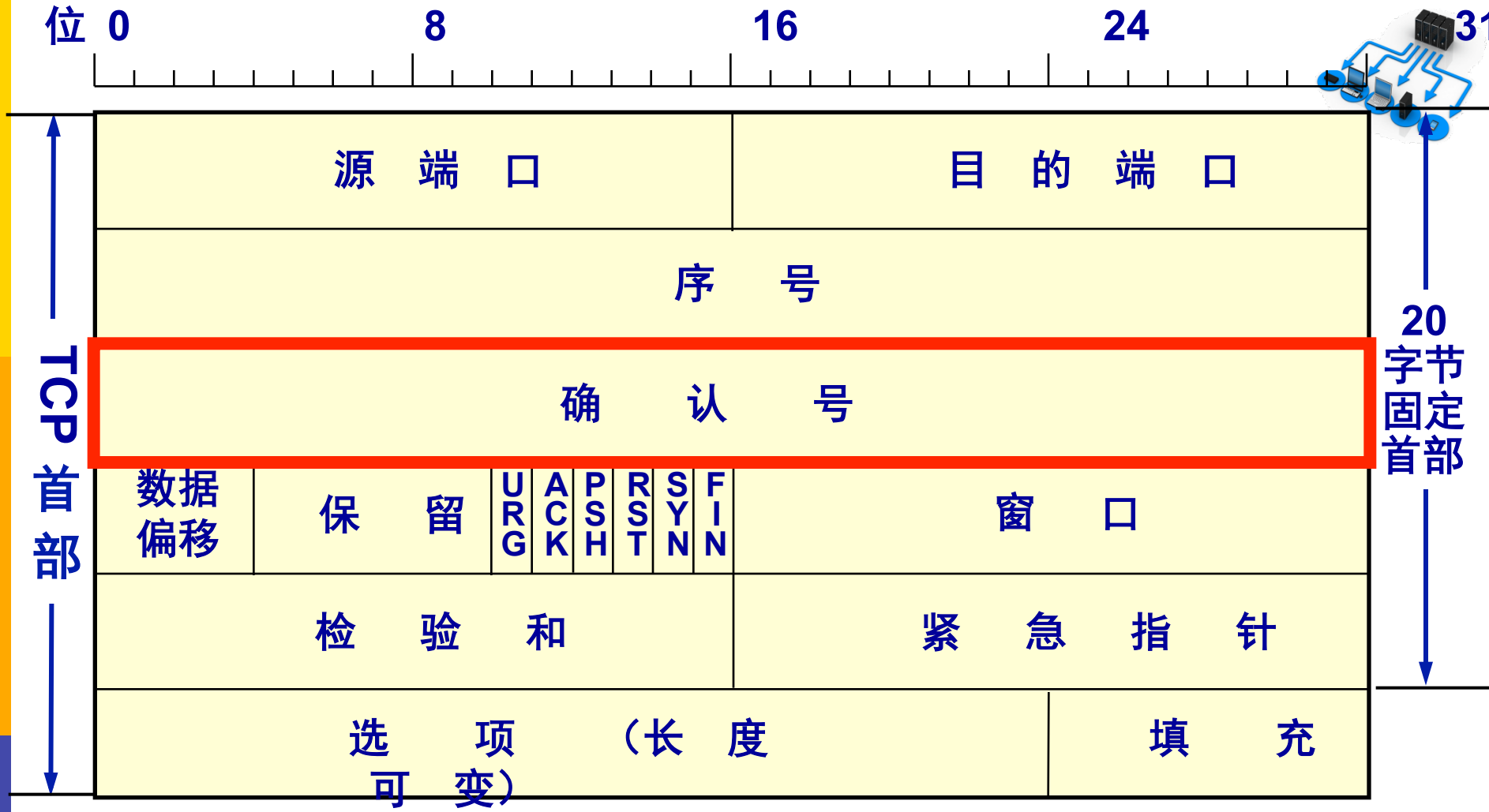




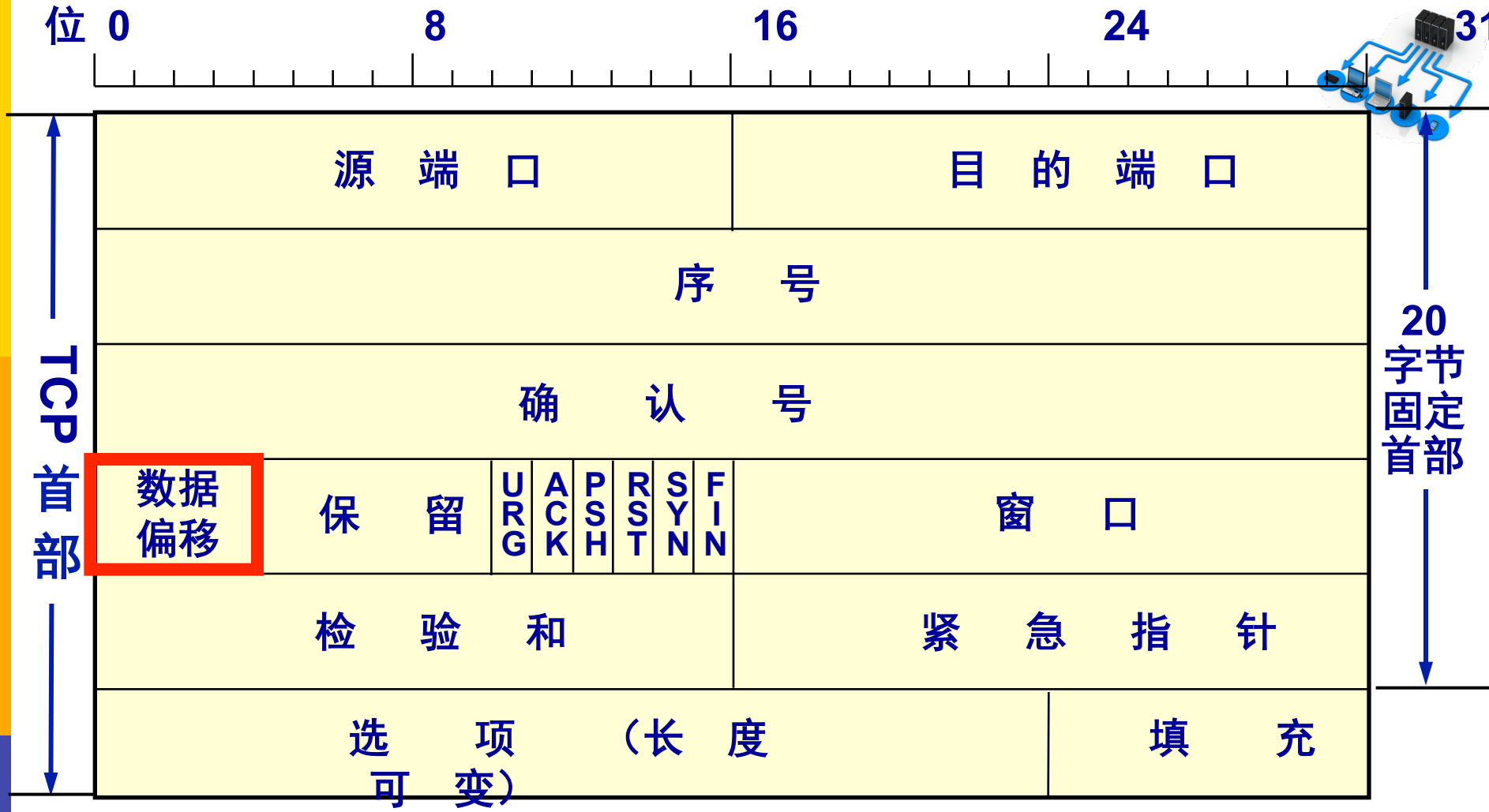
源端口和目的端口字段——各占 2 字节。端口是运输层与应用层的服务接口。运输层的复用和分用功能都要通过端口才能实现。



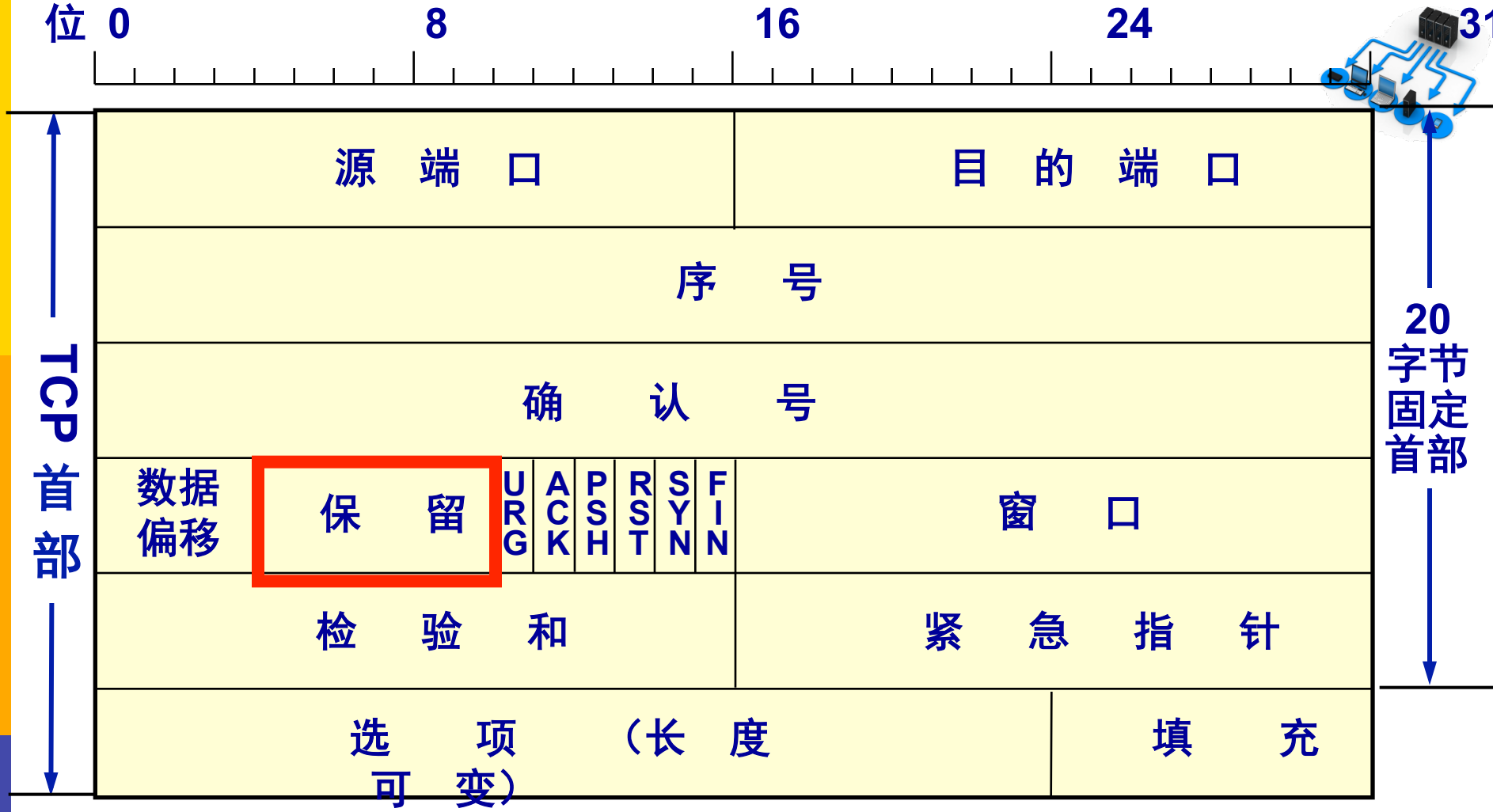
序号字段——占 4 字节。TCP 连接中传送的数据流中的每一个字节都编上一个序号。序号字段的值则指的是本报文段所发送的数据的第一个字节的序号。



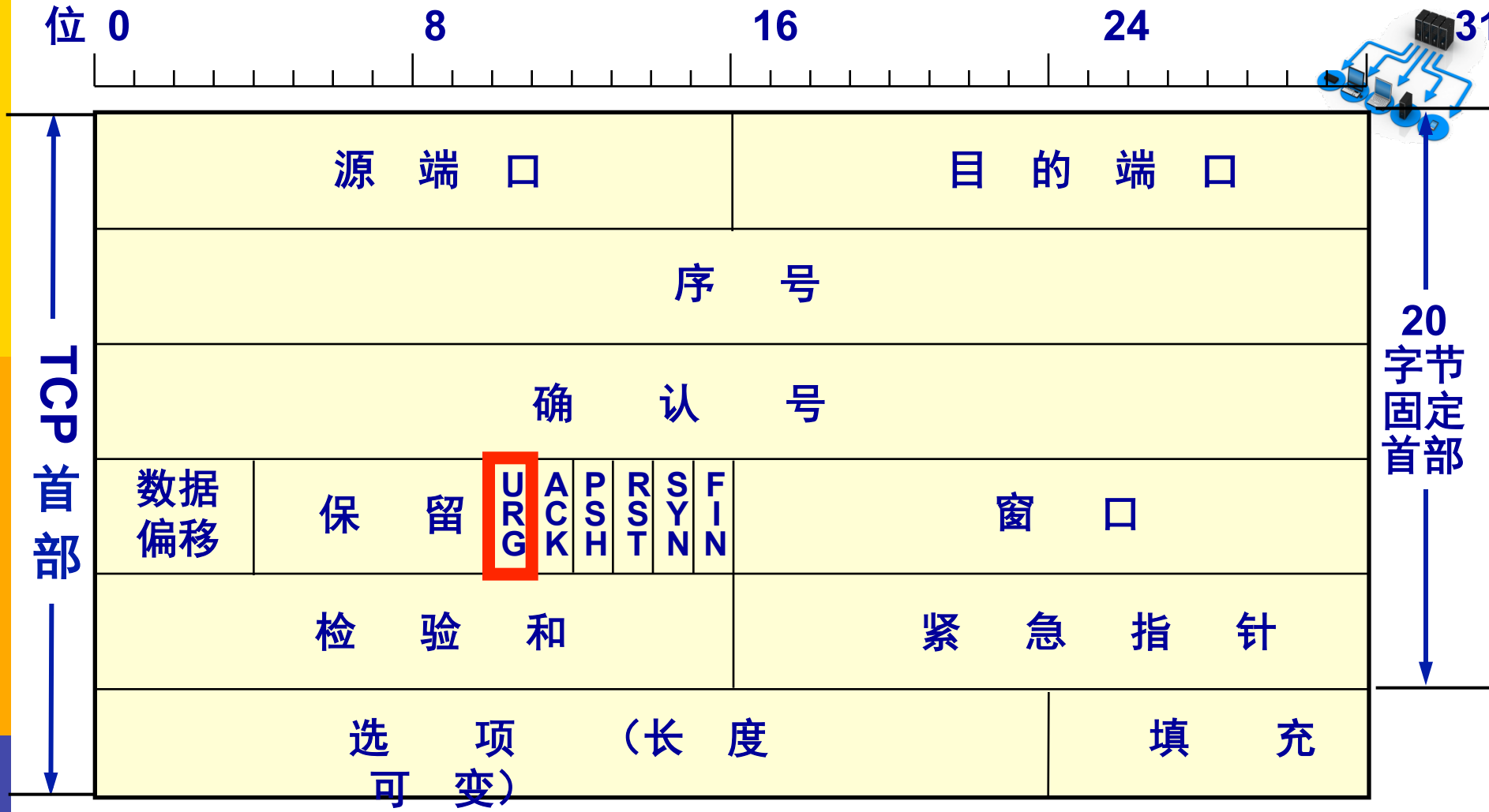
确认号字段——占 4 字节，是期望收到对方的下一个报文段的数据的第一个字节的序号。



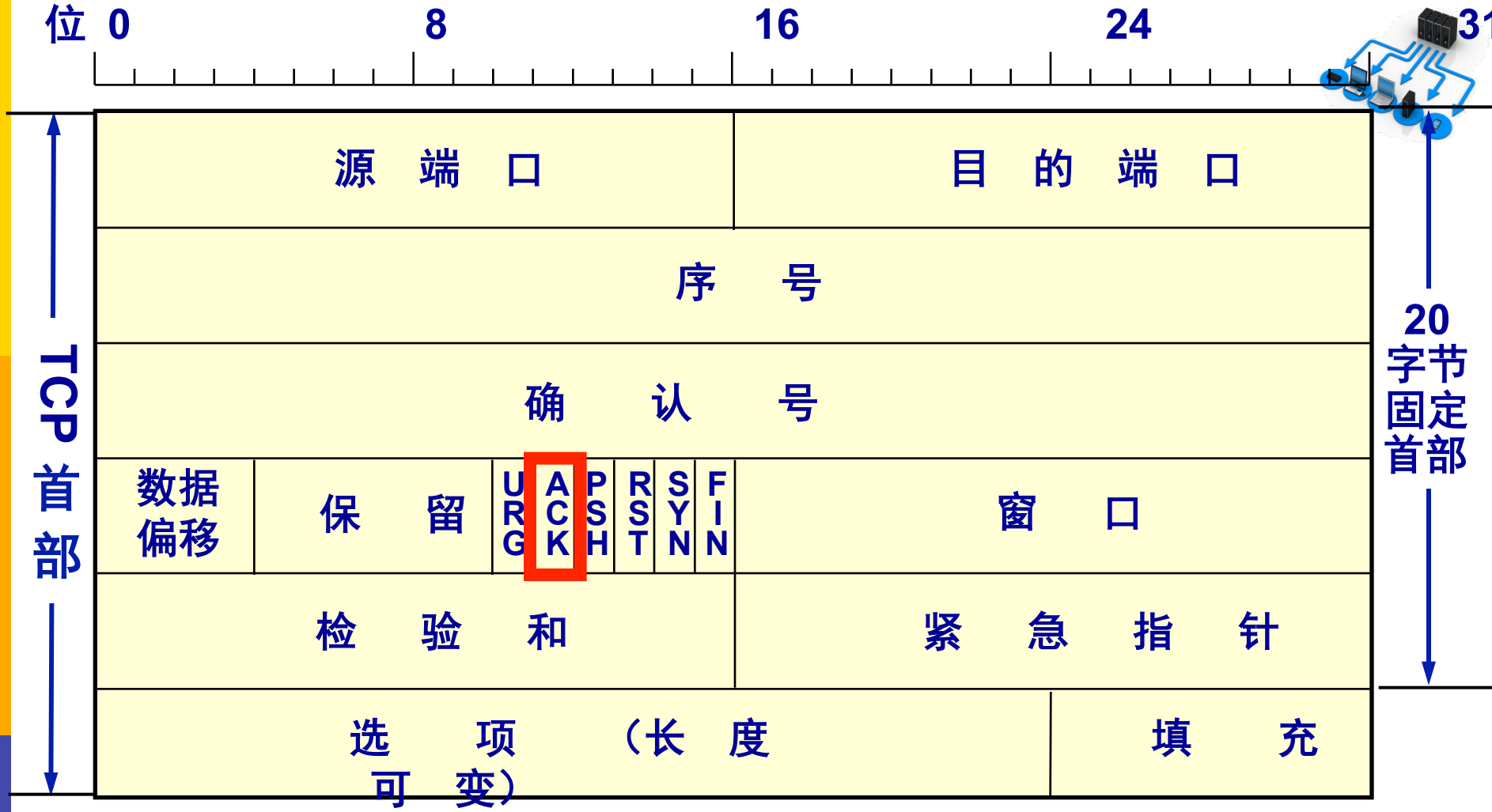
数据偏移（即首部长度）——占 4 位，它指出 TCP 报文段的数据起始处距离 TCP 报文段的起始处有多远。“数据偏移”的单位是 32 位字（以 4 字节为计算单位）。



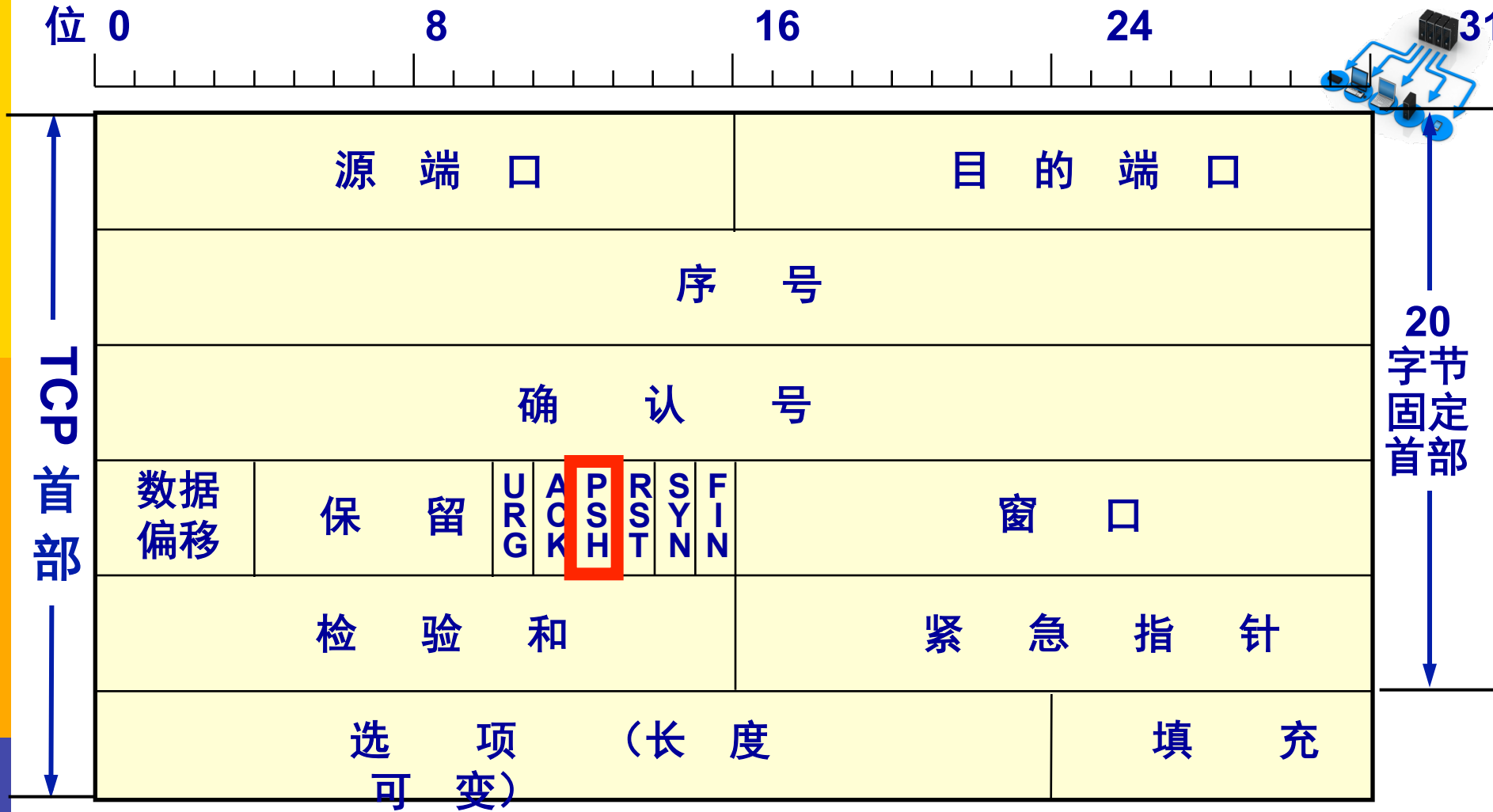
保留字段——占 6 位，保留为今后使用，但目前应置为 0。



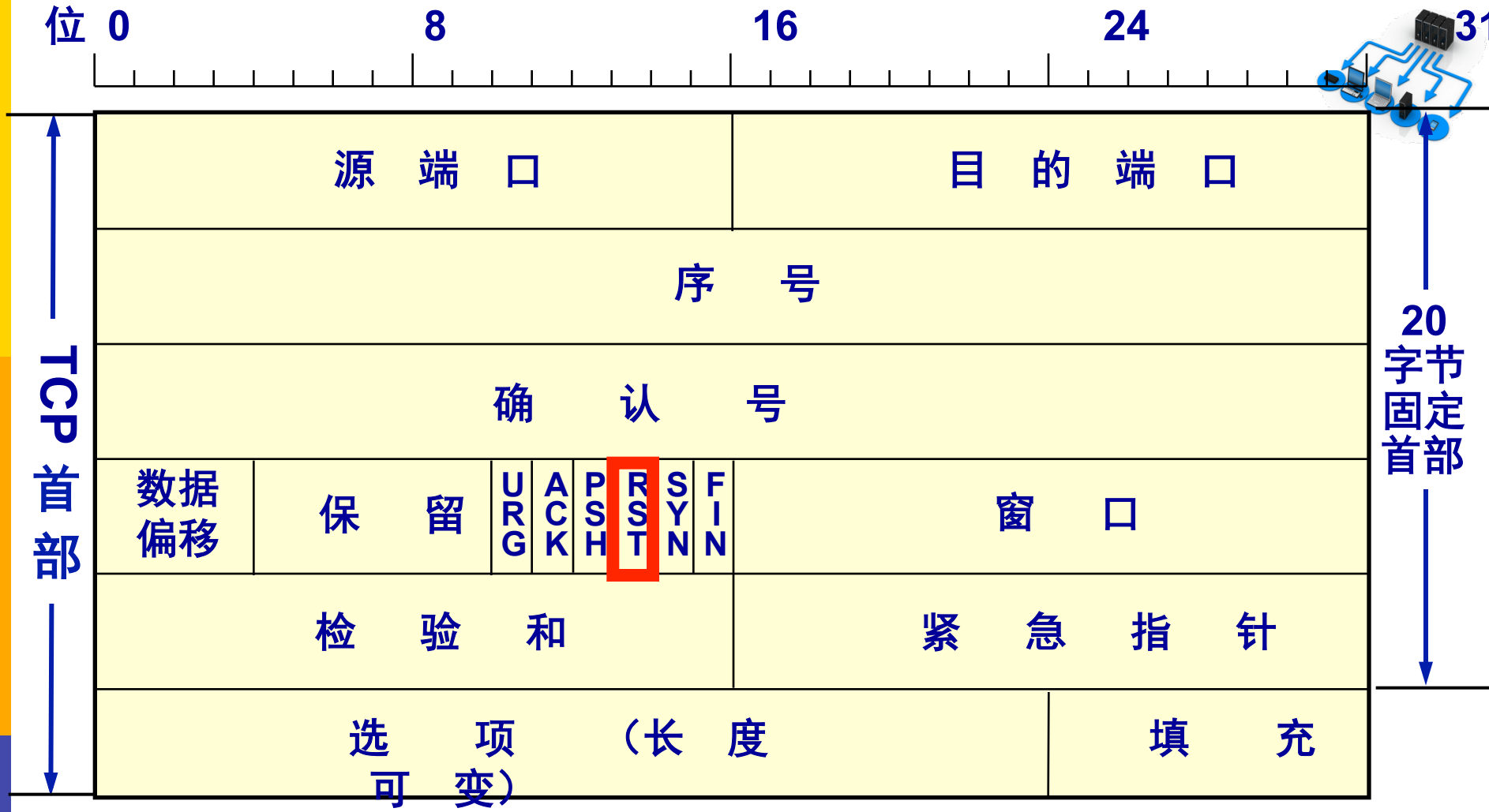
紧急 URG —— 当 URG = 1 时，表明紧急指针字段有效。它告诉系统此报文段中有紧急数据，应尽快传送(相当于高优先级的数据)。



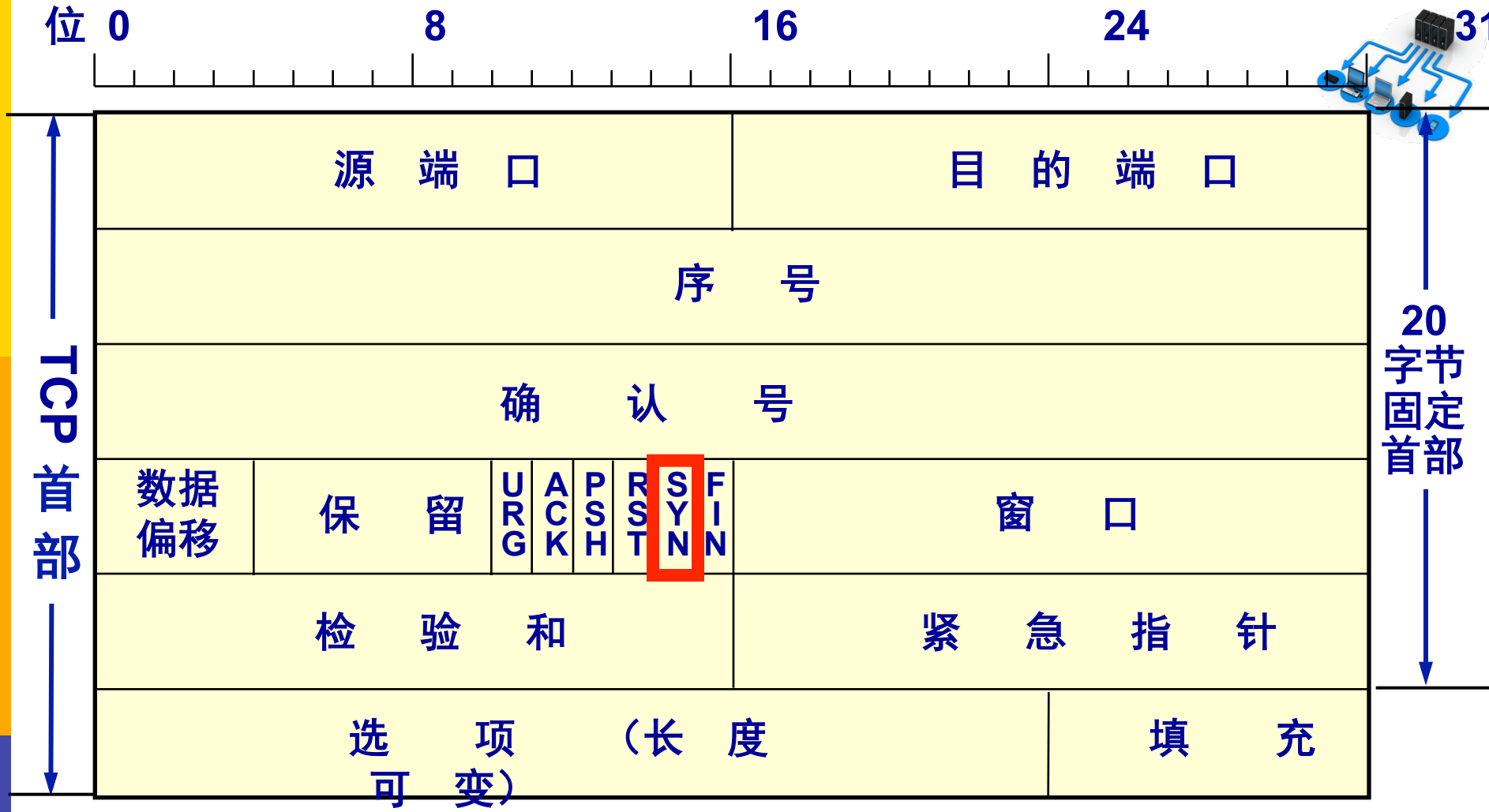
确认 ACK —— 只有当 ACK = 1 时确认号字段才有效。当 ACK = 0 时，确认号无效。



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。

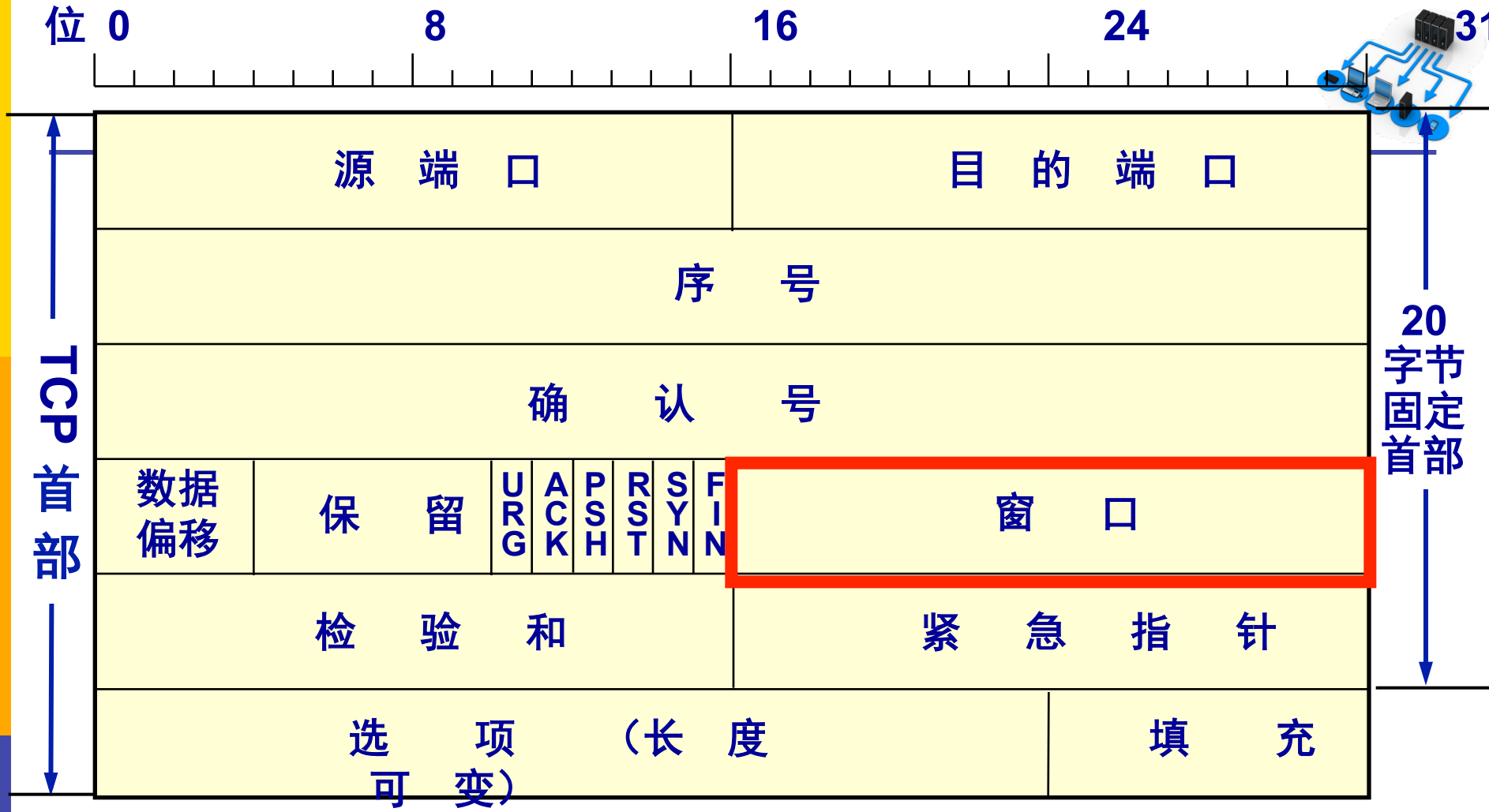


复位 RST (ReSeT) —— 当 RST = 1 时，表明 TCP 连接中出现严重差错（如由于主机崩溃或其他原因），必须释放连接，然后再重新建立运输连接。



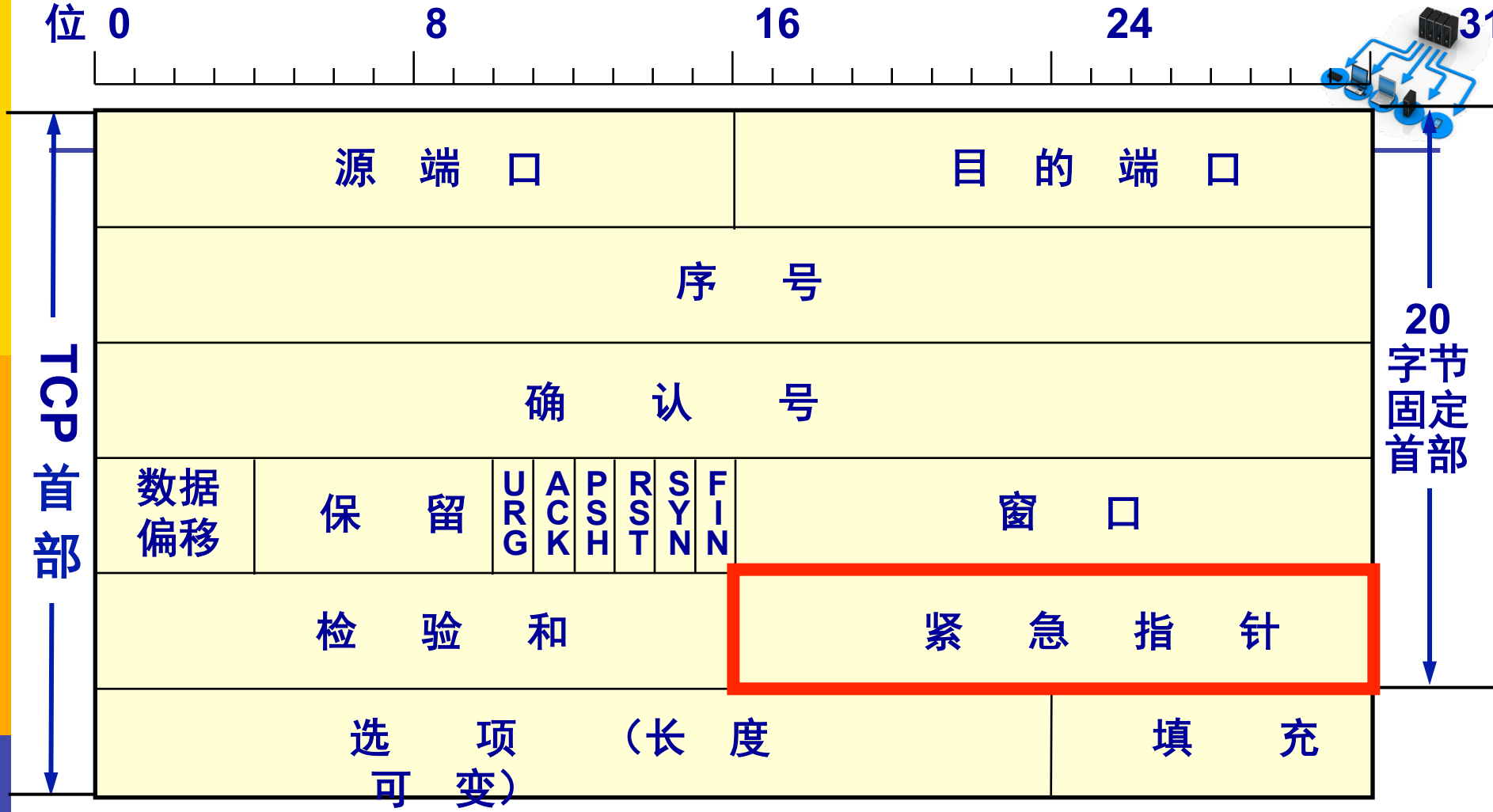
同步 SYN —— 同步 SYN = 1 表示这是一个连接请求或连接接受报文。

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。

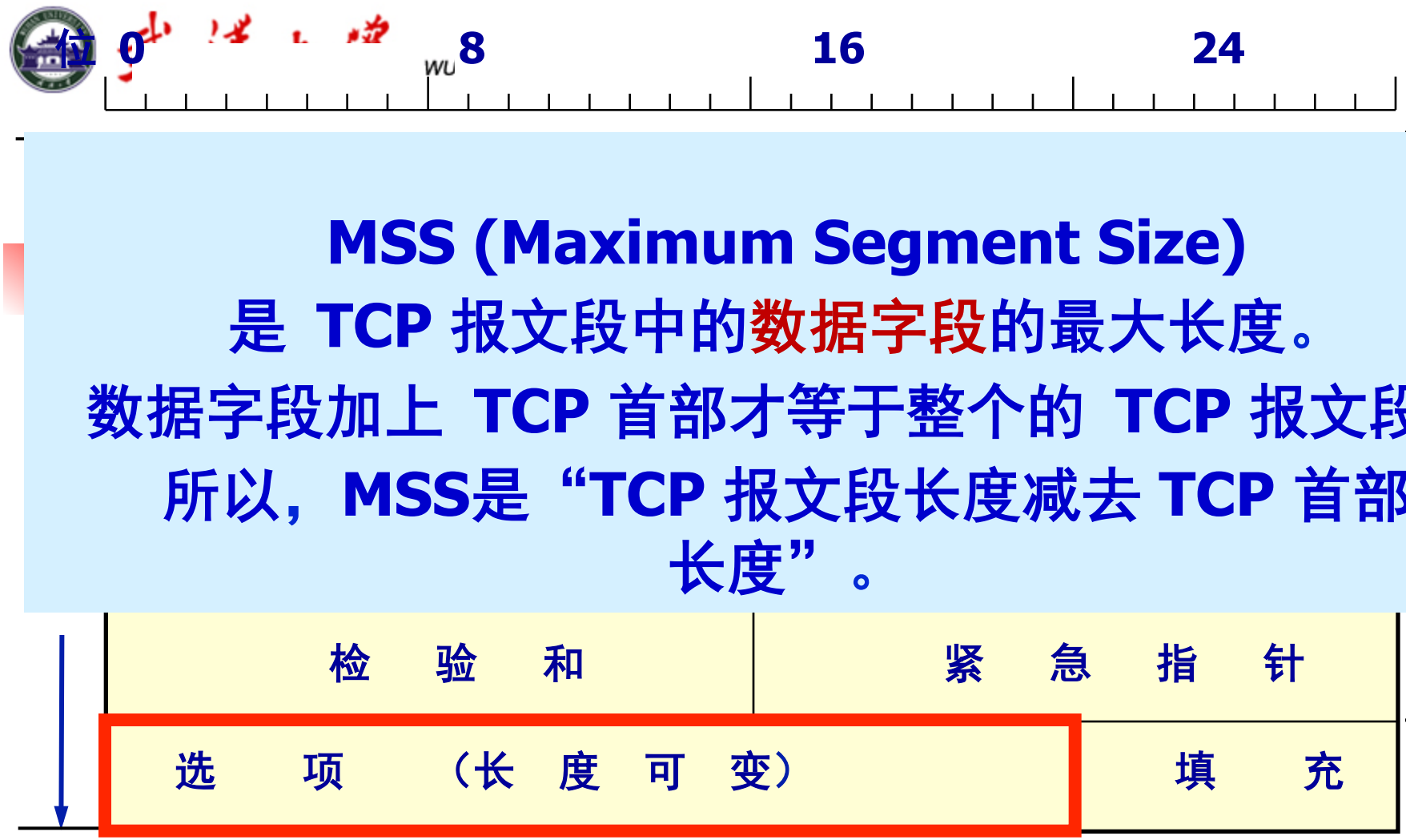


窗口字段 —— 占 2 字节，用来让对方设置发送窗口的依据，单位为字节。

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。

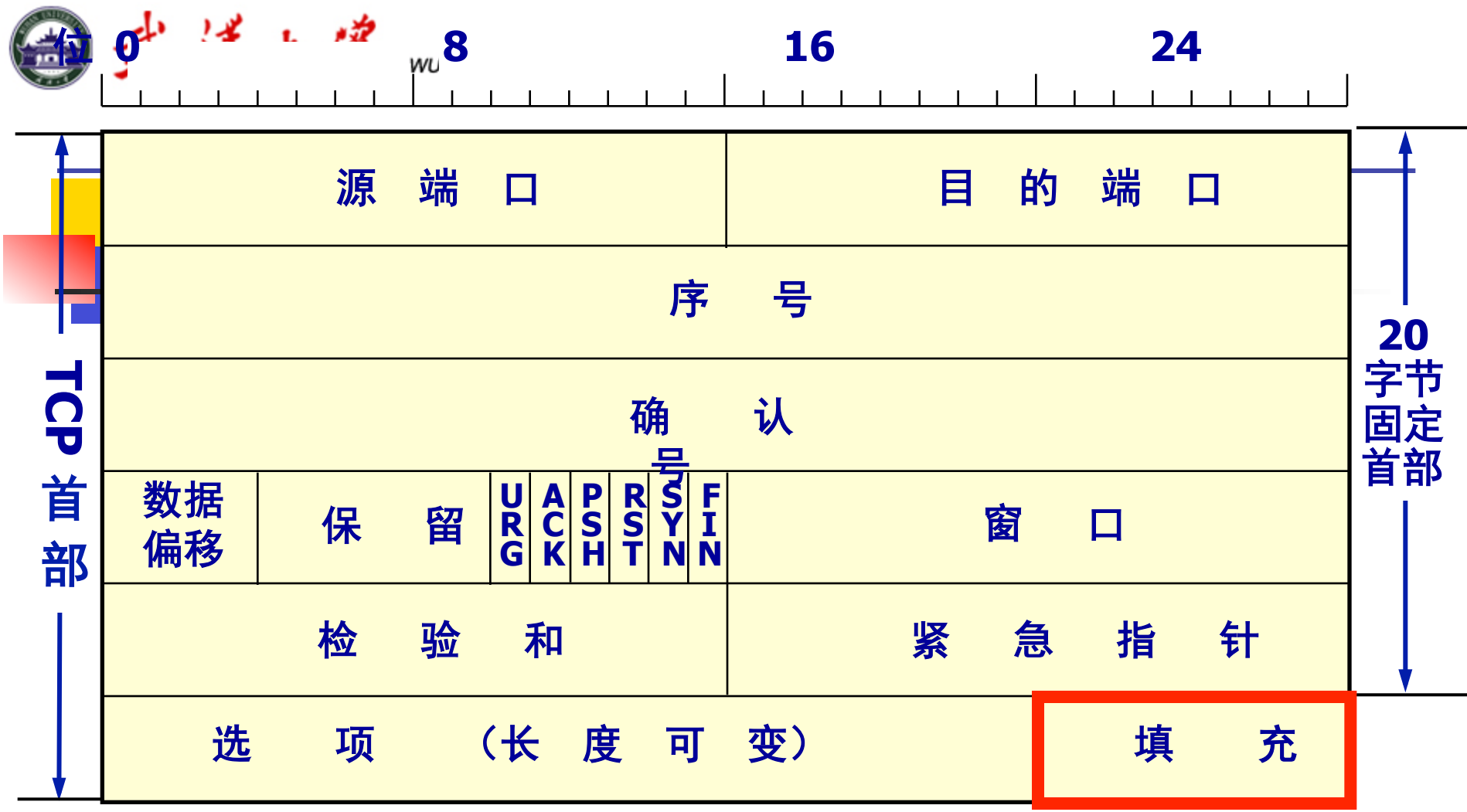
数据字段加上 TCP 首部才等于整个的 TCP 报文段。

所以，MSS是“TCP 报文段长度减去 TCP 首部长度”。

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。

TCP 包

0000 11 75 00 8b e3 54 50 3f 00 00 00 00 70 02 ff ff
0010 5c 39 00 00
这个TCP包是什么类型的包?

02 = 0000 0010 = SYN

```
▼ Transmission Control Protocol, Src Port: 4469 (4469), Dst Port: netbios-ssn (139), Seq: 0, Len: 0
    Source Port: 4469 (4469)
    Destination Port: netbios-ssn (139)
    [Stream index: 1]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    Acknowledgment number: 0
    Header Length: 28 bytes
  ▶ .... 0000 0000 0010 = Flags: 0x002 (SYN)
    Window size value: 65535
    [Calculated window size: 65535]
  ▶ Checksum: 0x5c39 [validation disabled]
    Urgent pointer: 0
  ▶ Options: (8 bytes), Maximum segment size, No-Operation (NOP), No-Operation (NOP), SACK permitted
```

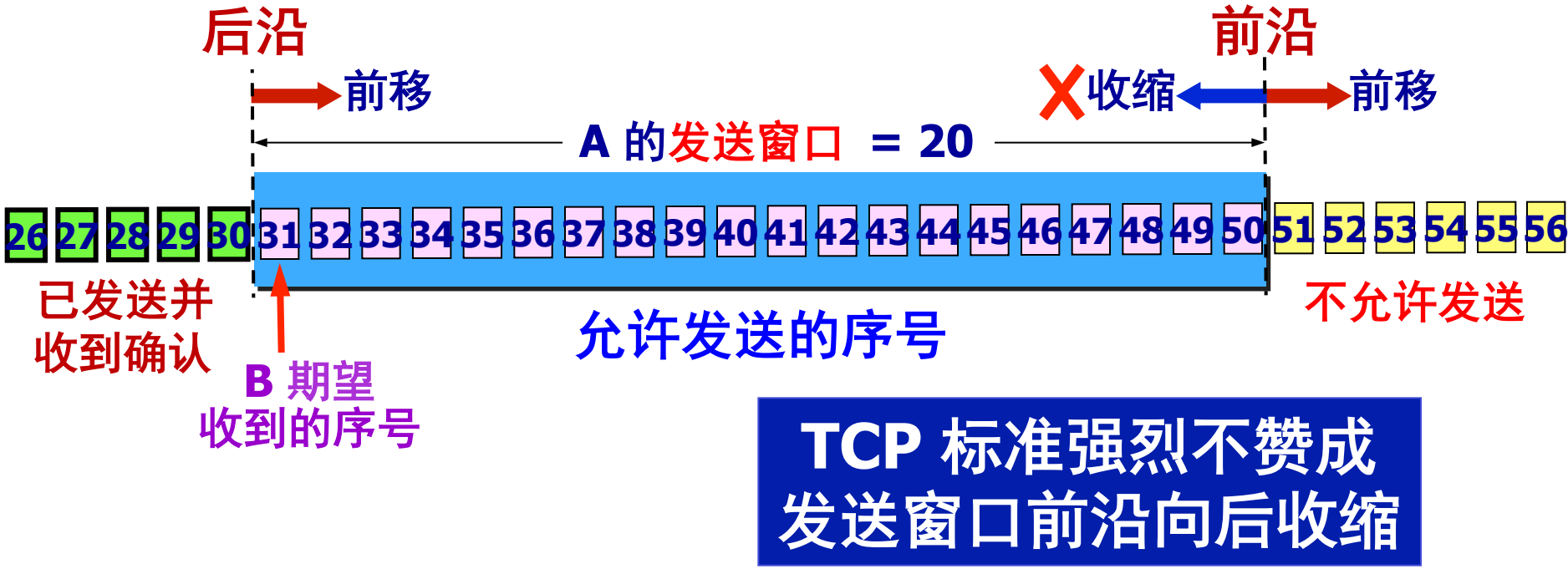
5.6 TCP 可靠传输的实现

- 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

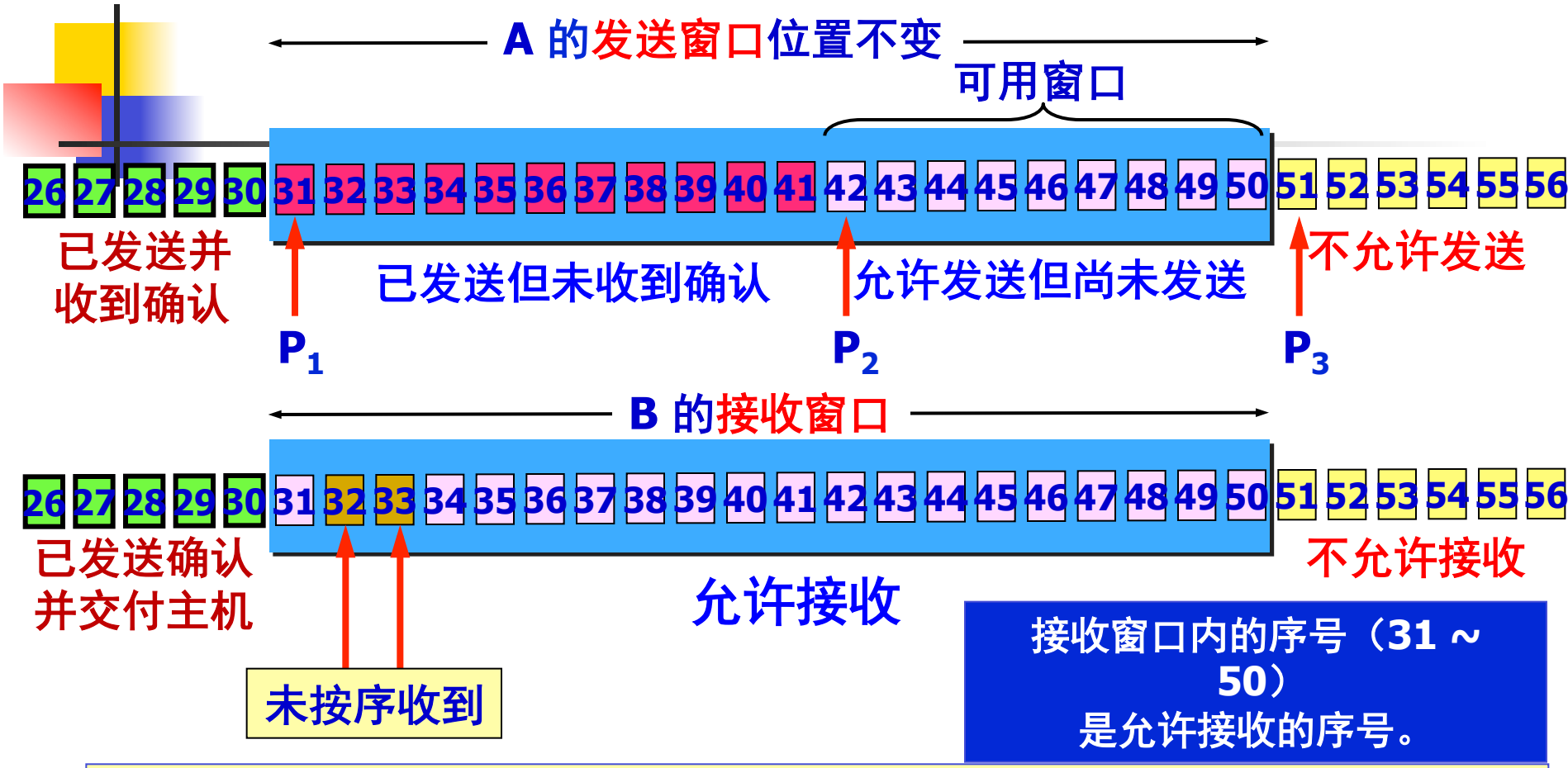
- TCP 的滑动窗口是以字节为单位的。
- 现假定 A 收到了 B 发来的确认报文段，其中窗口是 20 字节，而确认号是 31（这表明 B 期望收到的下一个序号是 31，而序号 30 为止的数据已经收到了）。
- 根据这两个数据，A 就构造出自己的发送窗口，

- 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

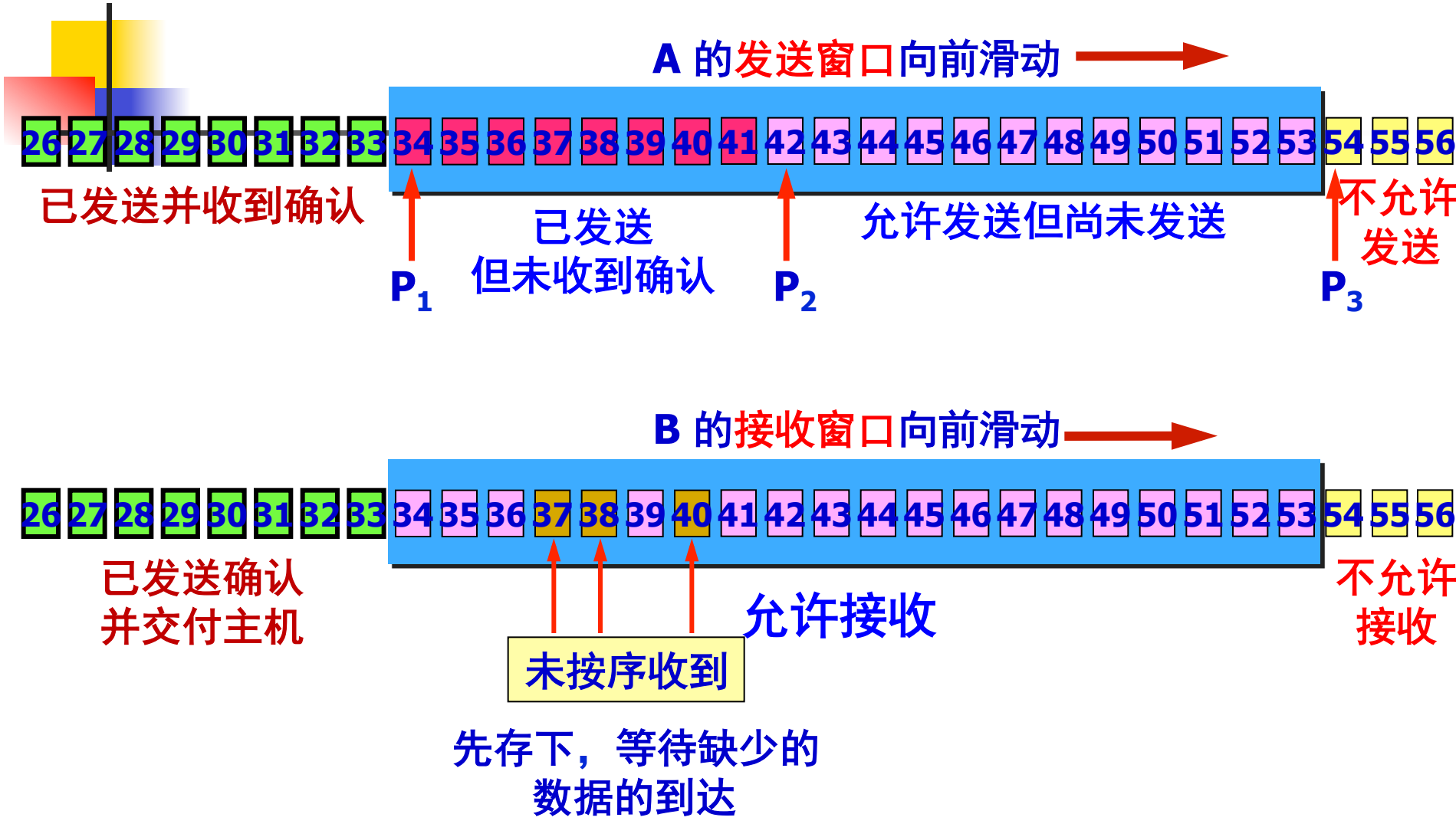
A 发送了 11 个字节的数据



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

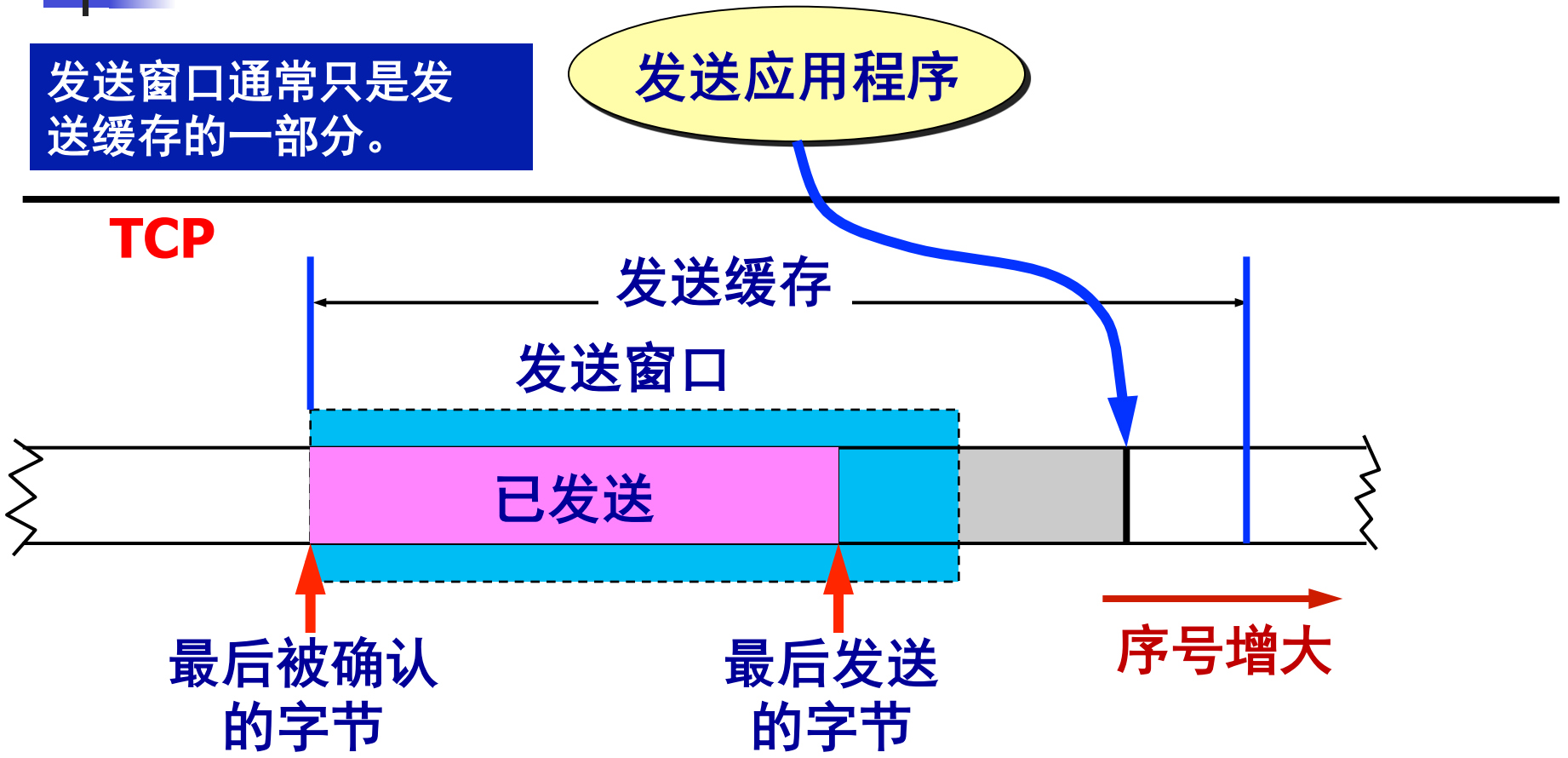


A 的发送窗口内的序号都已用完，但还没有再收到确认，必须停止发送。

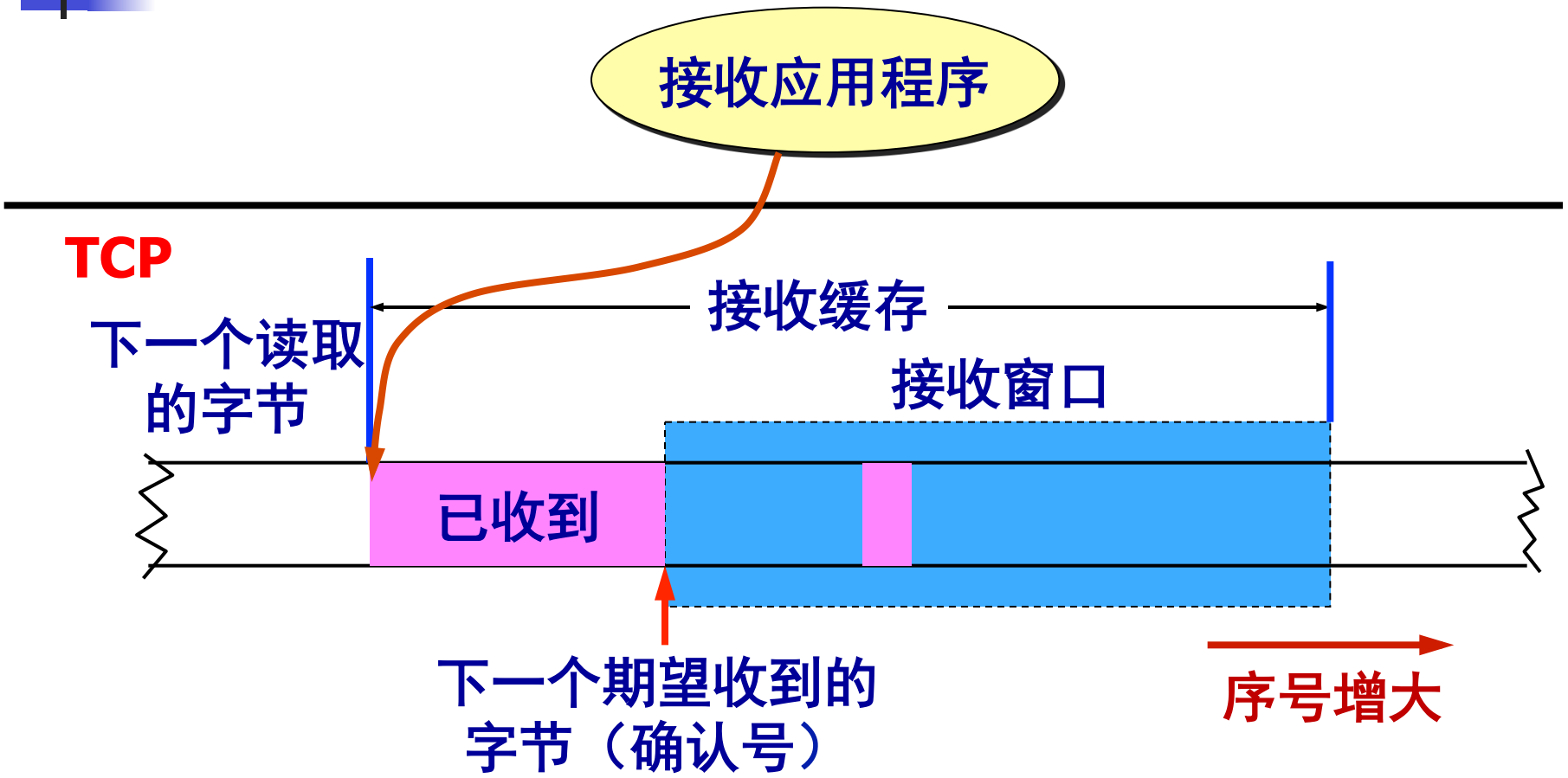
发送窗口内的序号都属于已发送但未被确认

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。



接收方的应用进程从 TCP 的接收缓存中读取字节流。



累积确认

- 确认发送时间
  - Piggyback
  - 接收缓存满
  - 上层读取数据（接收缓存变化）

5.6.2 超时重传时间的选择

- 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。
  - 平均时间RTT_S
  - 偏差时间RTT_D

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

加权平均往返时间

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

新的RTT_S = (1 - α) × (旧的RTT_S)
+ α × (新的RTT样本)　　(5-4)

- 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。

偏差时间RTT_D

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

新的 RTT_D = (1 - β) × (旧的RTT_D)
+ β × |RTT_S - 新的 RTT 样本|　　(5-6)

- β 是个小于 1 的系数，其推荐值是 1/4，即 0.25。

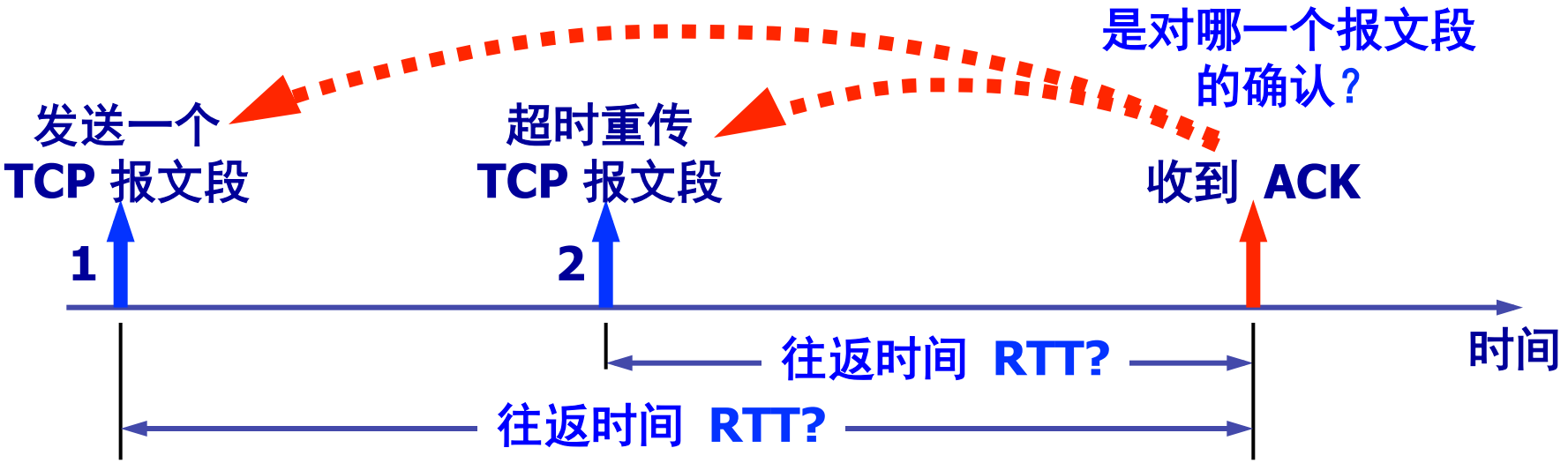
超时重传时间 RTO

- 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：

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

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)

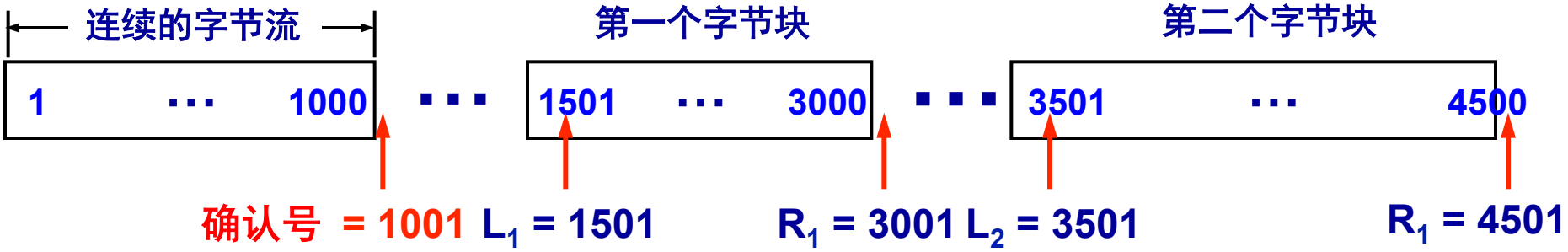
- 系数 γ 的典型值是 2 。
- 当不再发生报文段的重传时，才根据报文段的往返时延更新平均往返时延 RTT 和超时重传时间 RTO 的数值。
- 实践证明，这种策略较为合理。

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 $R_2 = 4501$。

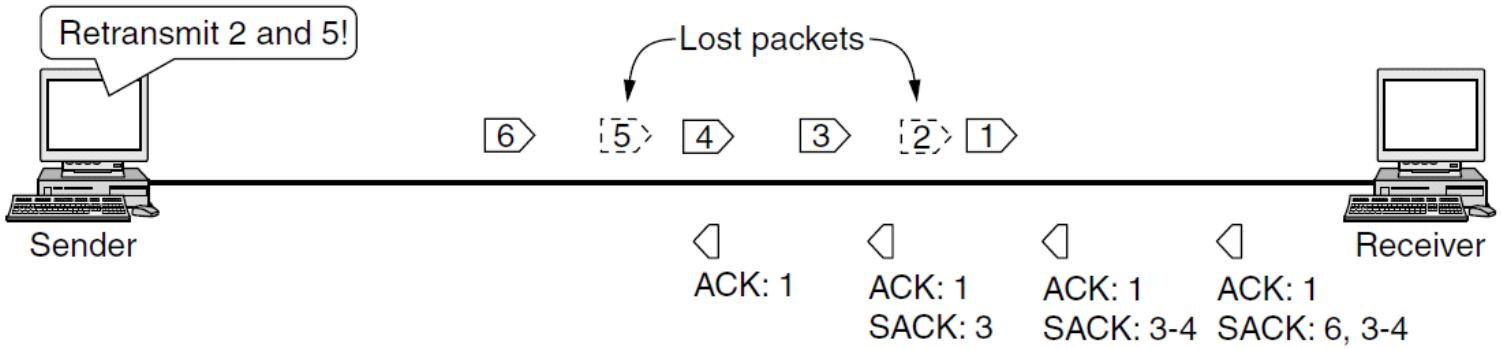
5.6.3 选择确认 SACK

- 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。

5.6.3 选择确认 SACK

SACK (Selective ACKs)：

- 扩展ACK，即添加一个域用来描述收到的包，这样就知道哪些丢失了
- 接收方通过SACK，能够知道哪些包丢失了，从而只发送那些丢失了的包

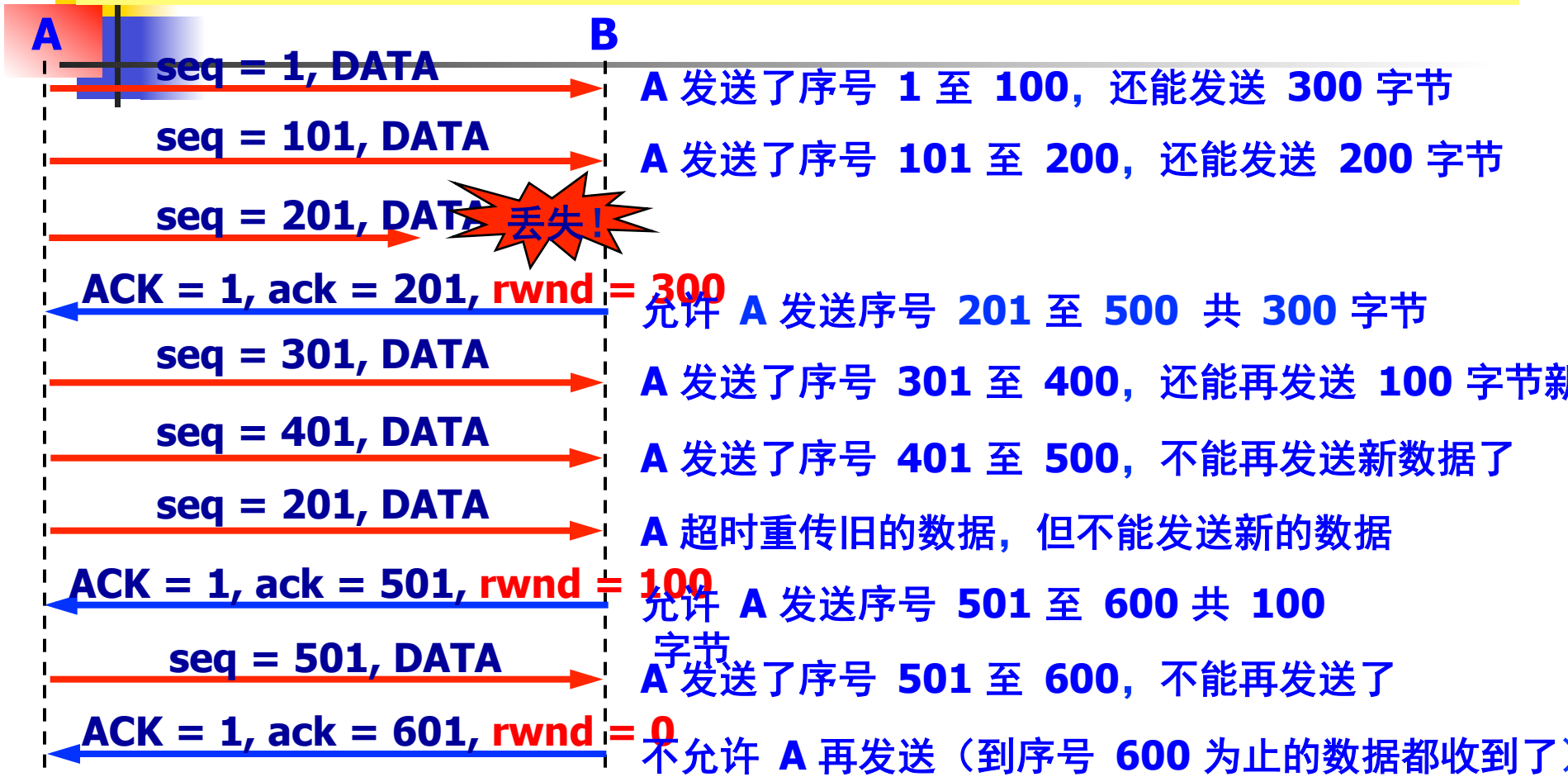


No way for us to know that 2 and 5 were lost with only ACKs

5.7 利用滑动窗口实现流量控制

- 流量控制 (flow control) 就是让发送方的发送速率不要太快，要让接收方来得及接收。
- 利用滑动窗口机制实现流量控制
  - 接收方告诉发送方它的接收能力（大小）

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）”。



这个时候B的应用程序读取了所有的数据，B发送ack=601,rwnd=400。如果这个数据包丢了怎么办?

可能发生死锁

- 发生死锁
  - B 向 A 发送了零窗口，
  - B的接收缓存释放， B 向 A 发送了 rwnd = 400
  - 此报文丢失，造成“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”。
  - 死锁
- 解决方案
  - TCP 为每一个连接设有一个持续计时器 (persistence timer)。

持续计时器

- 只要 TCP 连接的一方（设A）收到对方（B）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，A就发送一个零窗口探测报文段（仅携带 1 字节的数据）。
- 若B的窗口仍然是零，回复零窗口，A重新设置持续计时器。若B的窗口不是零，则死锁的僵局就可以打破了。

5.7.2 传输效率

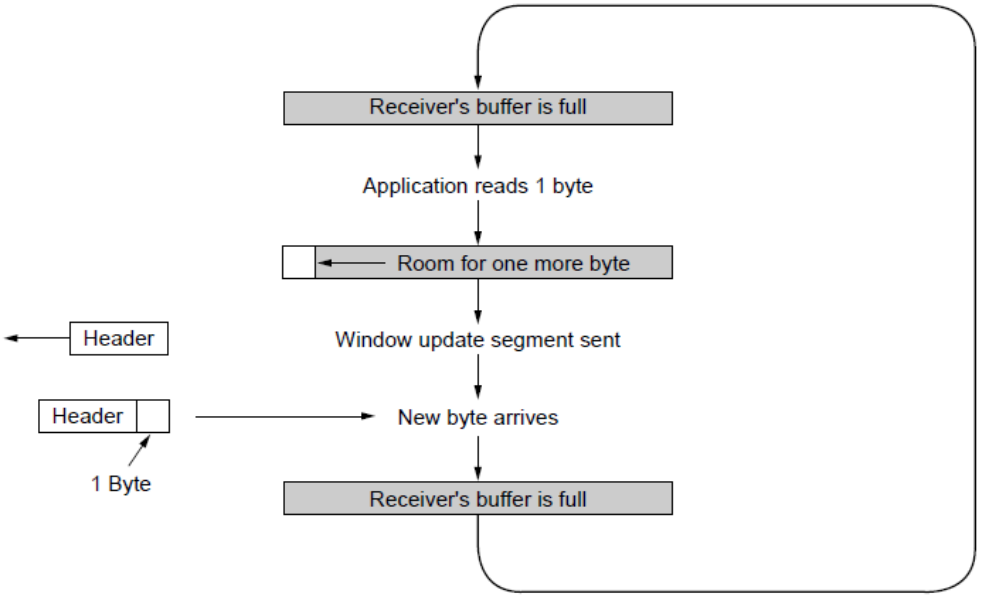
- 对某些应用，TCP的性能会大大降低，如：
  - SSH或者Telnet：对于用户输入的每个字符，都需要4个数据段：带字符的TCP段（41字节的IP包），此段的ACK（40字节的IP包），回显TCP段（接收方处理了这个字符，41字节的IP包），回显确认（40字节IP包）
  - 如果没有clark方案（见后），还存在滑动窗口变化ACK段（因接收方应用读取了一个字符，接收方窗口增大，发送窗口更新ACK）
- 解决方法：使用 Nagle 算法。

Nagle算法

- 发送方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。

接收方低能窗口综合症

- 当大批数据到达接收方，而接收方的上层每次只取一个数据，导致频繁的发送因滑动窗口变化的ACK。
- clark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

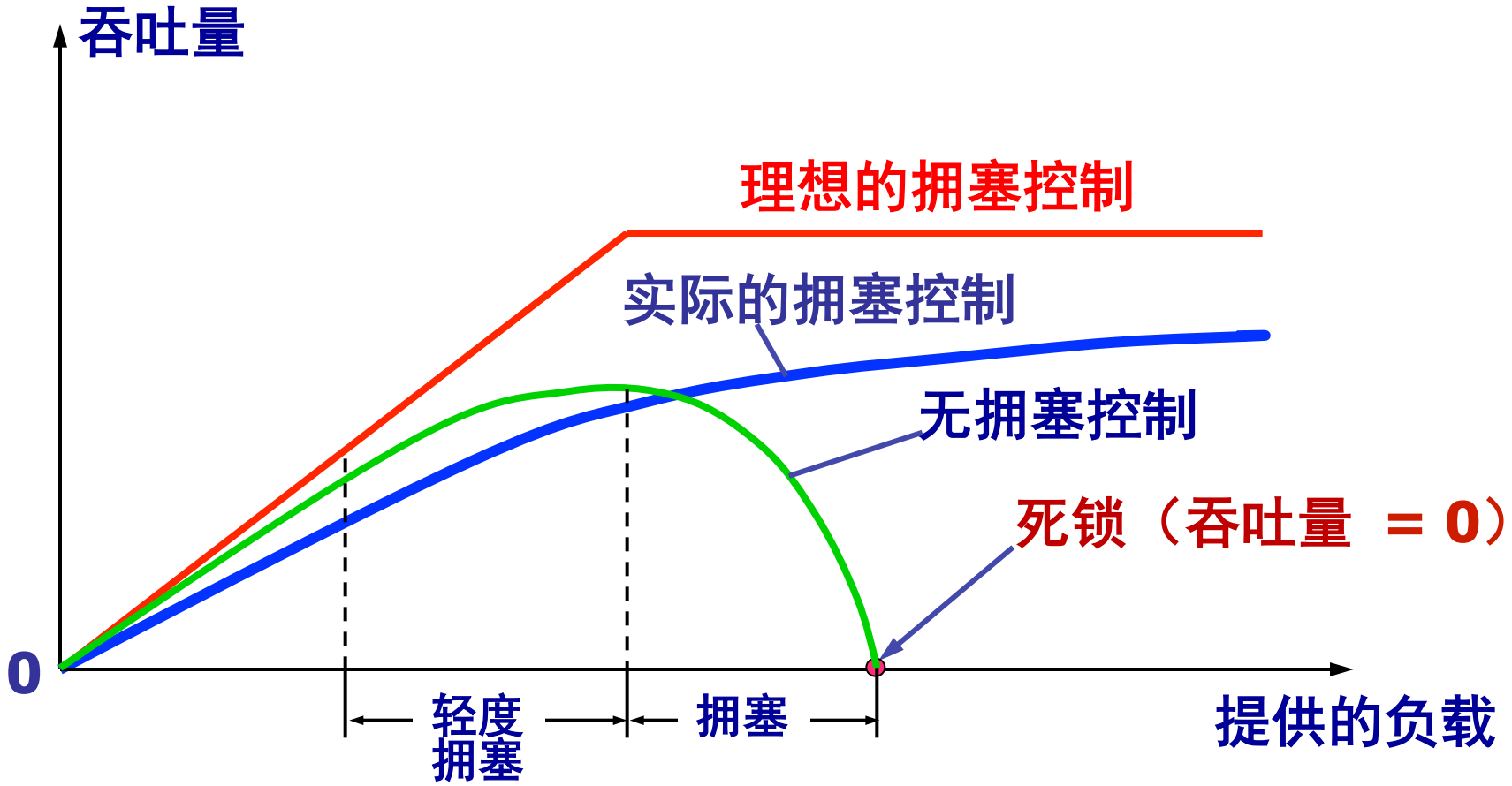
- 在某段时间，若对网络中某资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏。这种现象称为拥塞(congestion)。
- 若网络中有许多资源同时产生拥塞，网络的性能就要明显变坏，整个网络的吞吐量将随输入负荷的增大而下降。
- 出现拥塞的原因：

Σ对资源需求 > 可用资源 (5-7)

增加资源能解决拥塞吗？

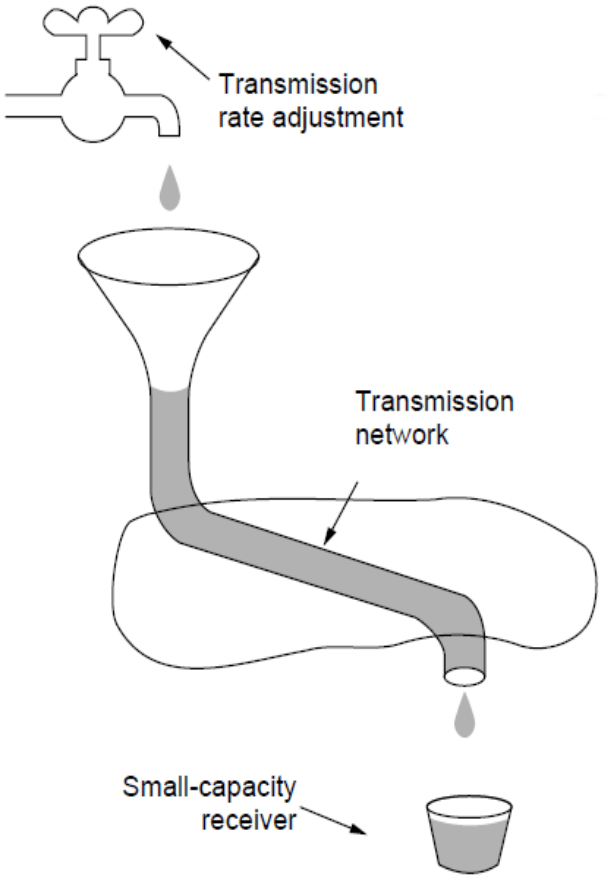
- 不能。这是因为网络拥塞是一个非常复杂的问题。简单地采用上述做法，在许多情况下，不但不能解决拥塞问题，而且还可能使网络的性能更坏。
- 网络拥塞往往是由许多因素引起的。例如：
  - 增大缓存，但未提高输出链路的容量和处理机的速度，排队等待时间将会大大增加，引起大量超时重传，解决不了网络拥塞；
  - 提高处理机处理的速率会会将瓶颈转移到其他地方；
- 拥塞不控制，使整个网络瘫痪

拥塞控制所起的作用



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。
- 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。



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（路由器）；
  - 平均队列长度（路由器）；
  - 超时重传的分组数（终端）；
  - 平均分组时延（终端）；
  - 分组时延的标准差，等等（终端）。
- 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） ：
  - 慢启动 (slow-start)
  - 拥塞避免 (congestion avoidance)
  - 快重传 (fast retransmit)
  - 快恢复 (fast recovery)

慢启动 (Slow start)

- 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。

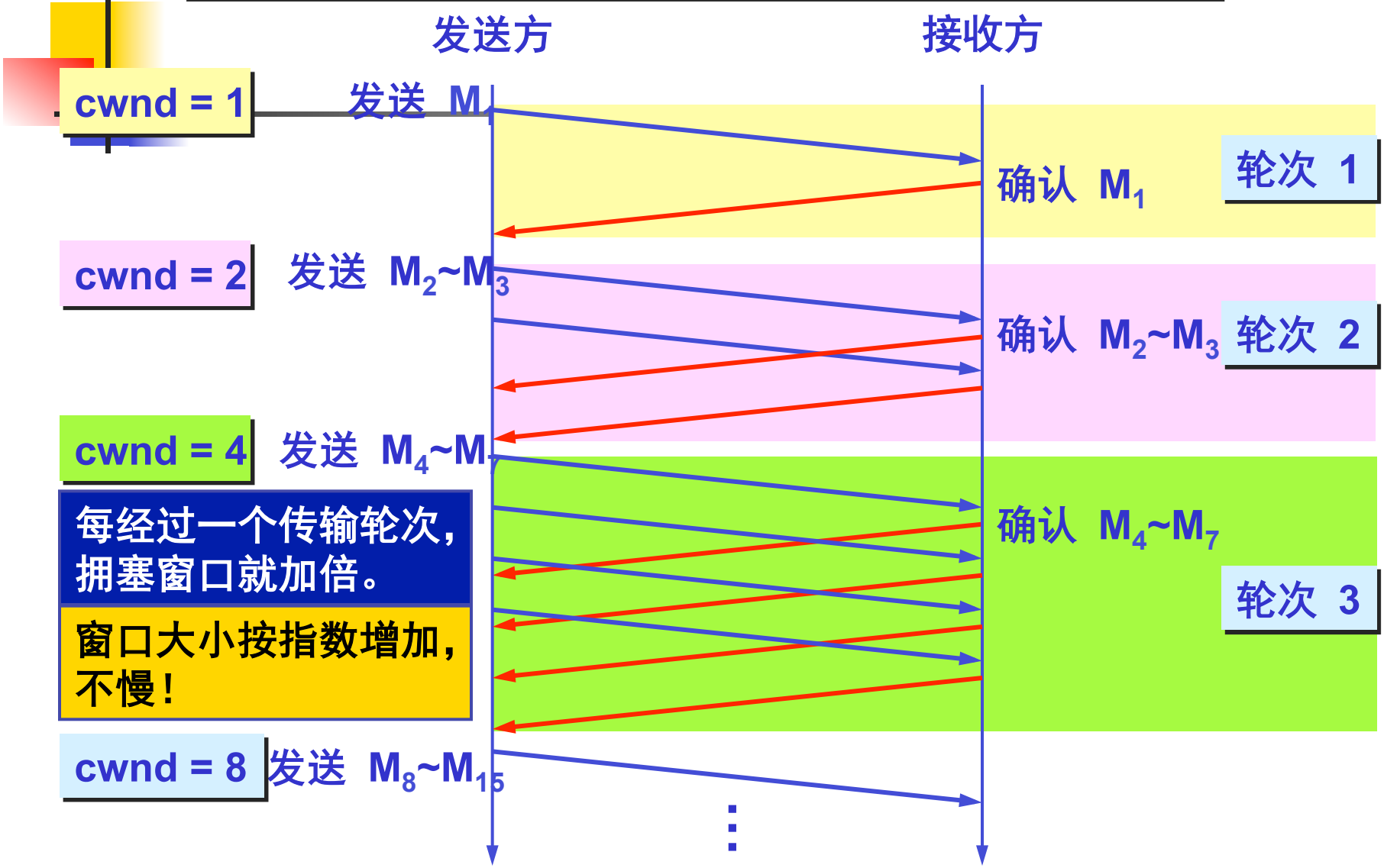
慢启动 (Slow start)

- 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

- 使用慢启动算法后，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。

设置慢启动门限状态变量 ssthresh

- 慢启动门限 ssthresh 的用法如下：
  - 当 cwnd < ssthresh 时，使用慢启动算法。
  - 当 cwnd > ssthresh 时，慢启动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。

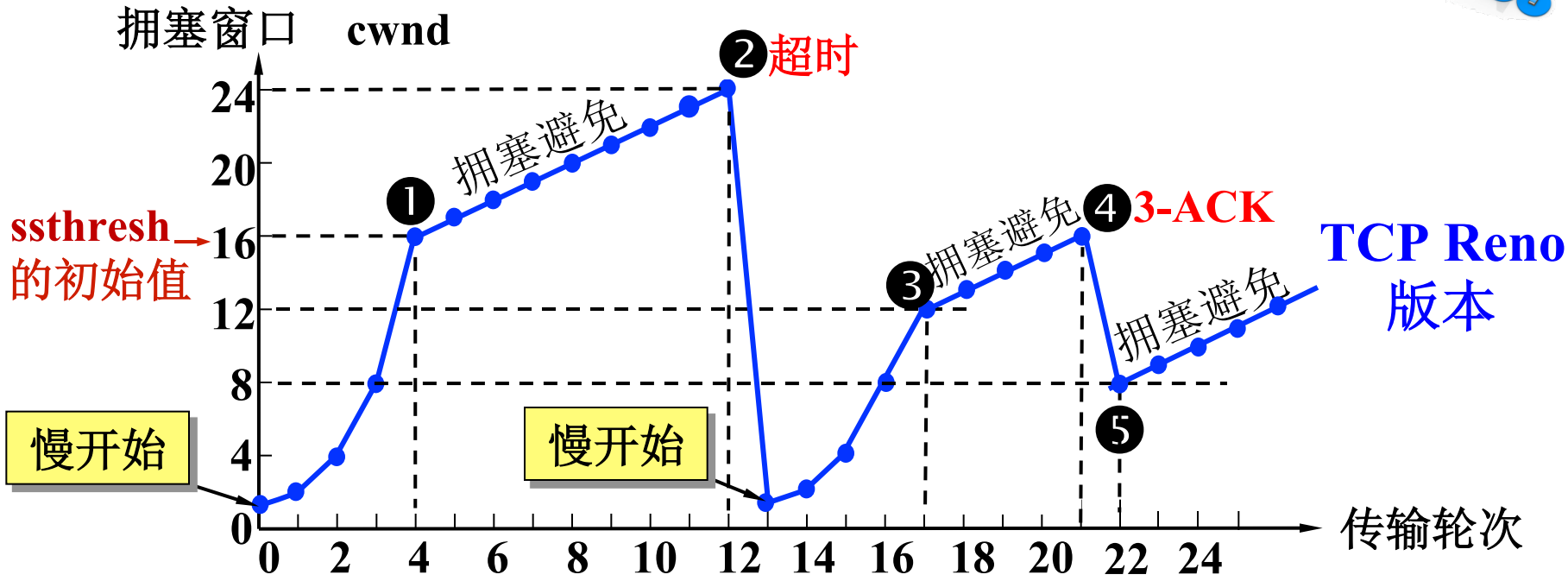
拥塞避免算法

- 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“加法递增” (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，比慢启动算法的拥塞窗口增长速率缓慢得多。

当网络出现拥塞时

- 无论在慢启动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
  - 执行慢启动算法
- 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。

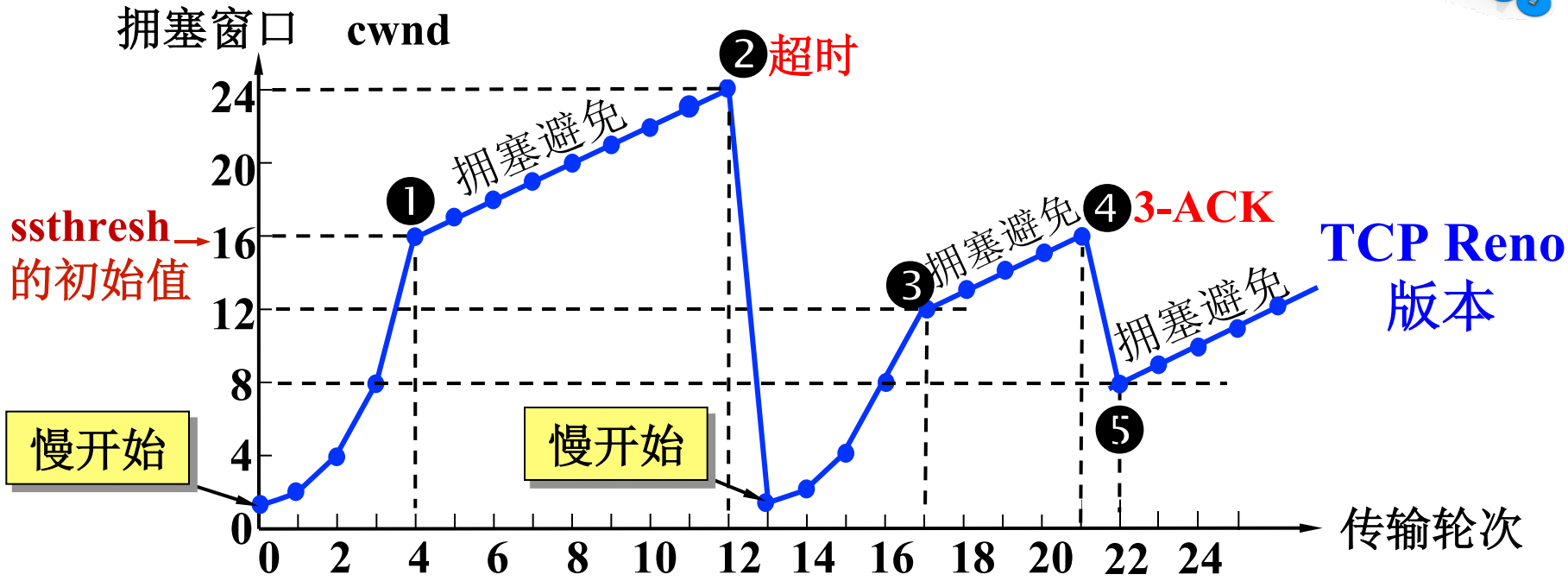
慢开始和拥塞避免算法的实现举例



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。

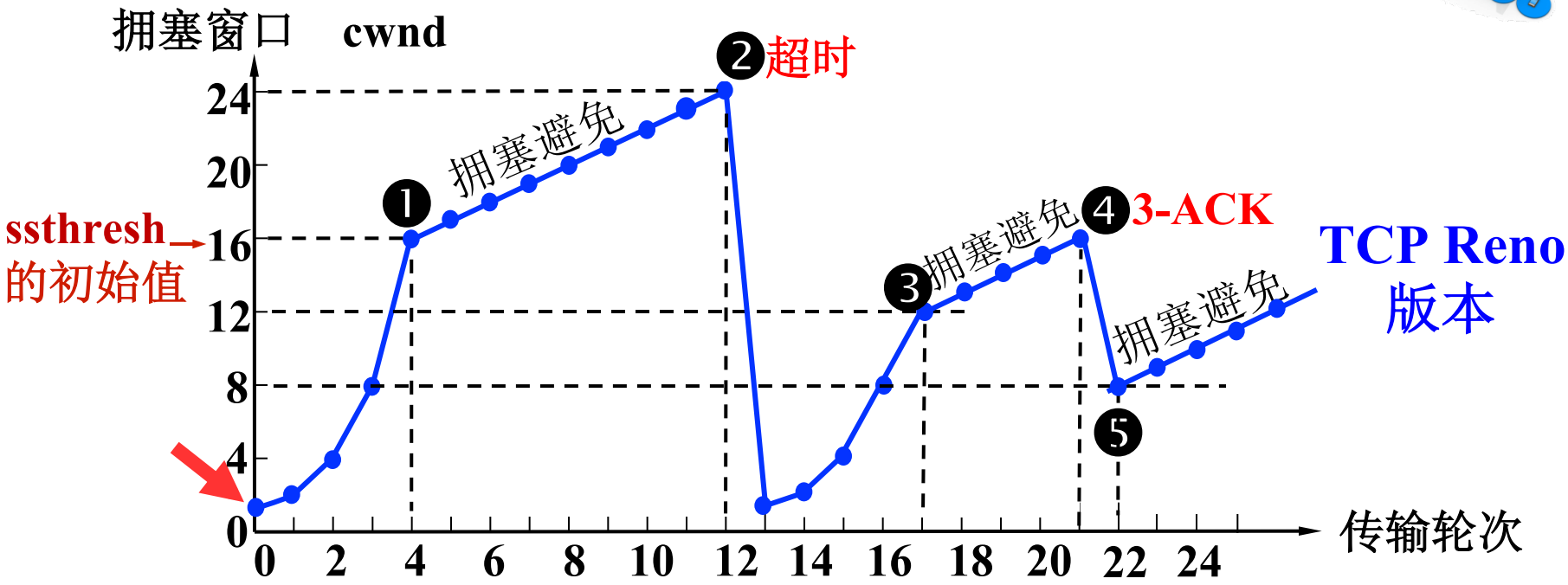
慢开始门限的初始值设置为 16 个报文段，即 ssthresh = 16。

慢开始和拥塞避免算法的实现举例



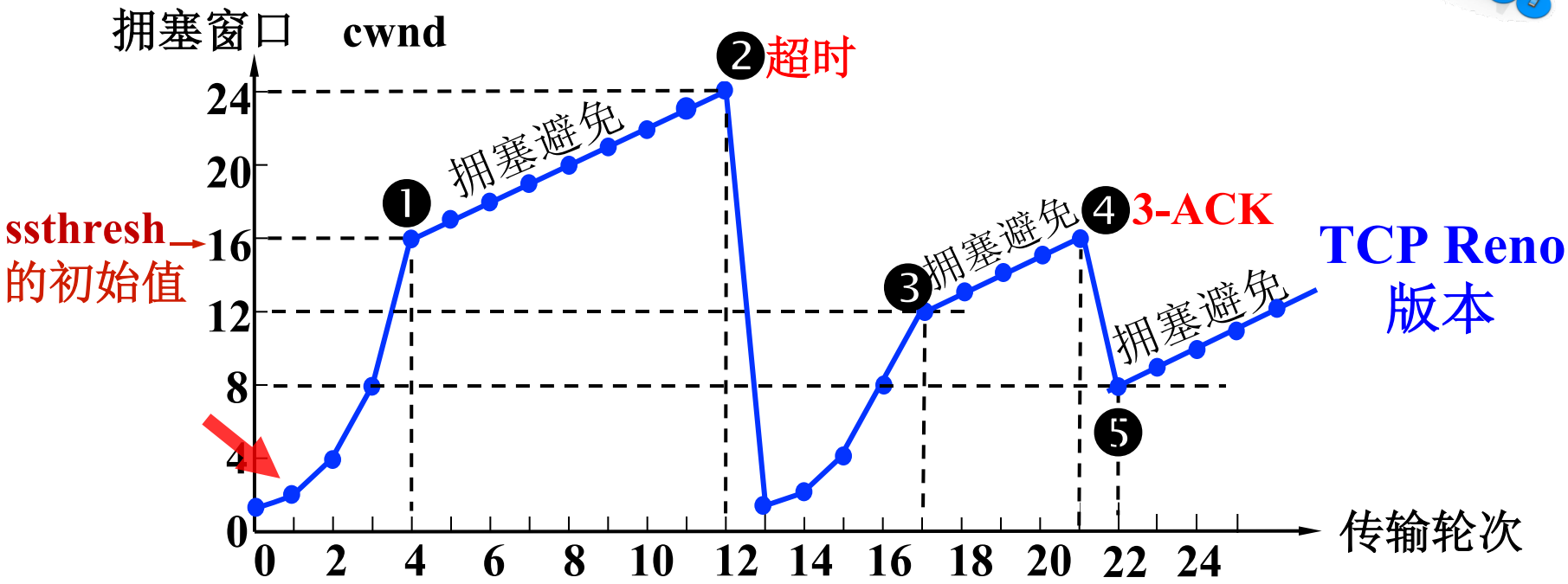
发送端的发送窗口不能超过拥塞窗口 cwnd 和接收端窗口 rwnd 中的最小值。我们假定接收端窗口足够大，因此现在发送窗口的数值等于拥塞窗口的数值。

慢开始和拥塞避免算法的实现举例



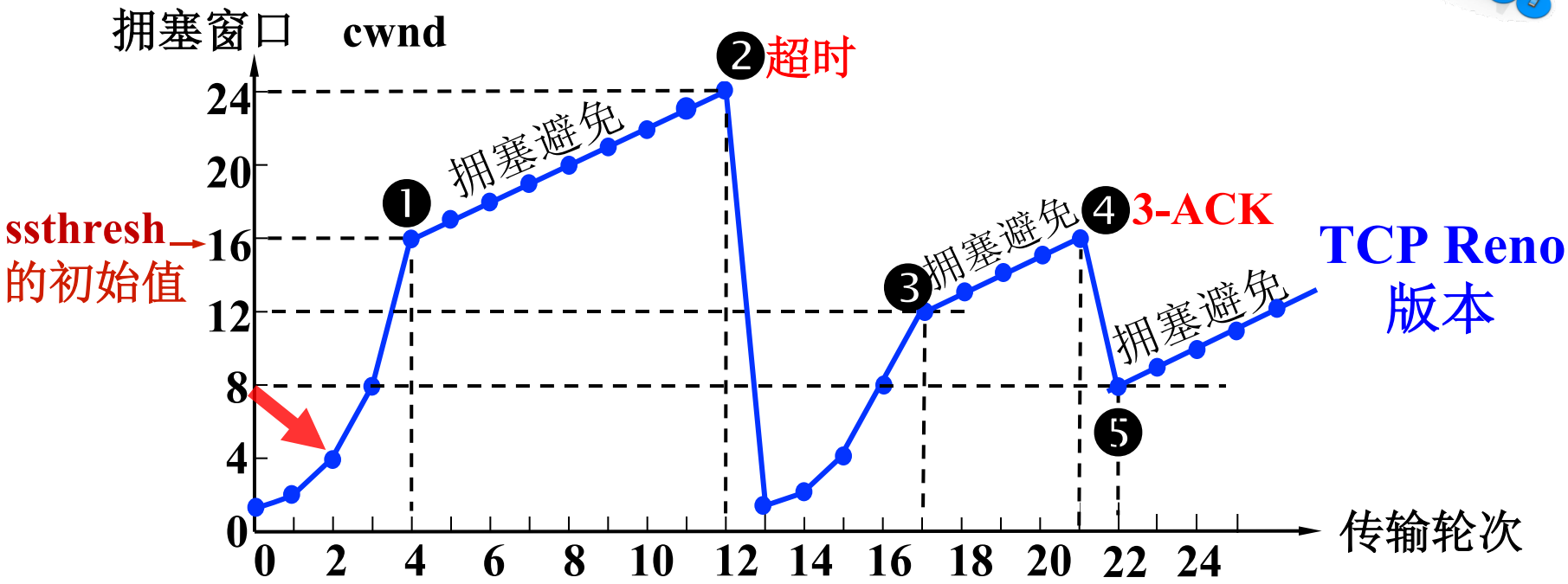
在执行慢开始算法时，拥塞窗口 cwnd=1，发送第一个报文段。

慢开始和拥塞避免算法的实现举例



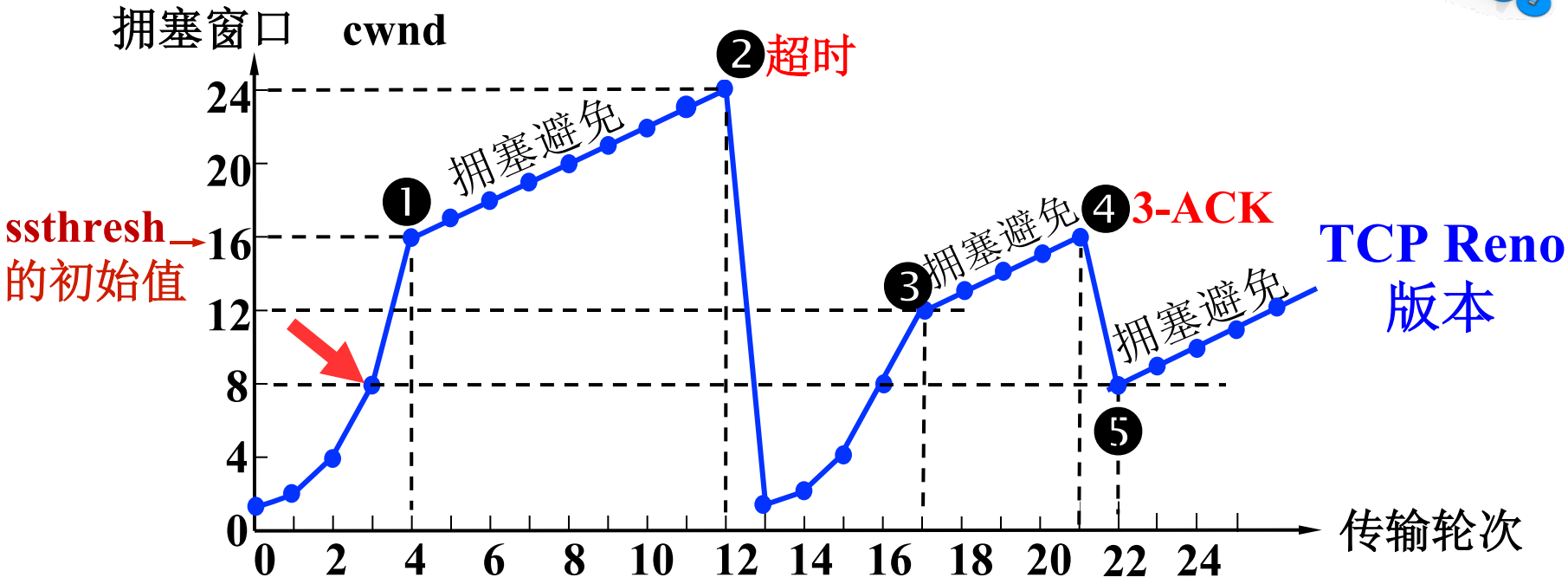
发送方每收到一个对新报文段的确认 ACK，就把拥塞窗口值加 1，然后开始下一轮的传输（请注意，横坐标是传输轮次，不是时间）。因此拥塞窗口 cwnd 随着传输轮次按指数规律增长。

慢开始和拥塞避免算法的实现举例



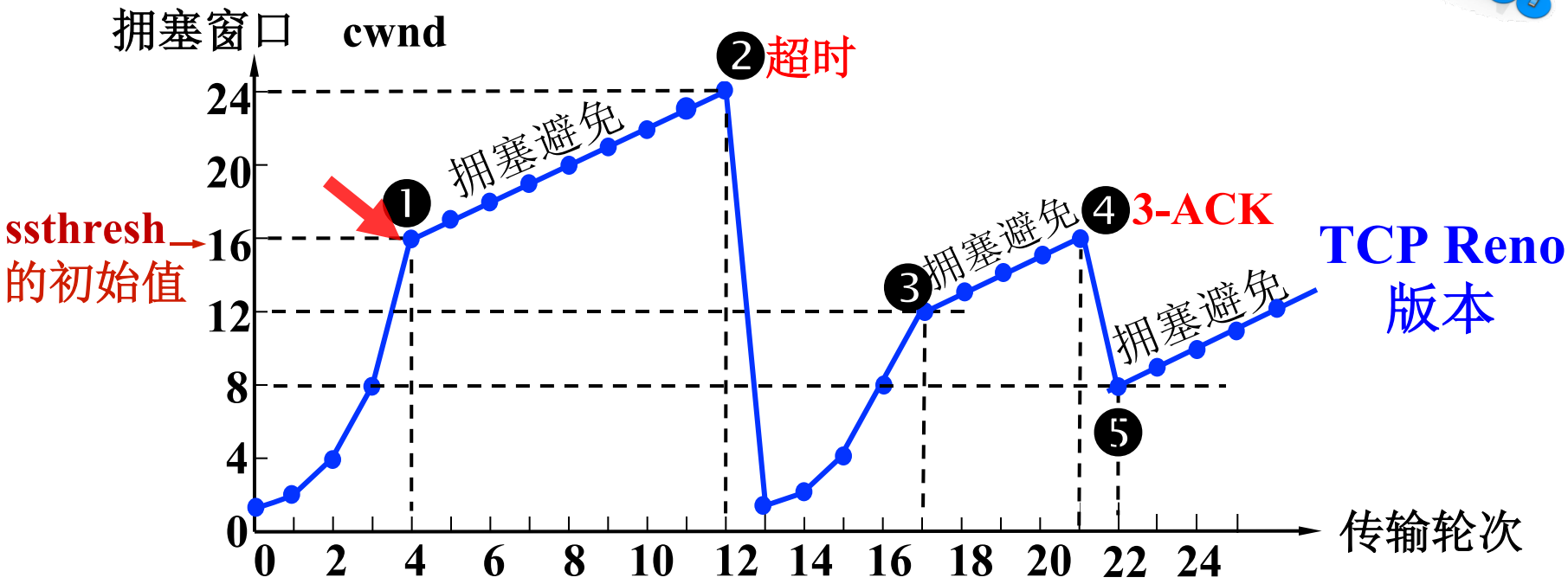
发送方每收到一个对新报文段的确认 ACK，就把拥塞窗口值加 1，然后开始下一轮的传输（请注意，横坐标是传输轮次，不是时间）。因此拥塞窗口 cwnd 随着传输轮次按指数规律增长。

慢开始和拥塞避免算法的实现举例



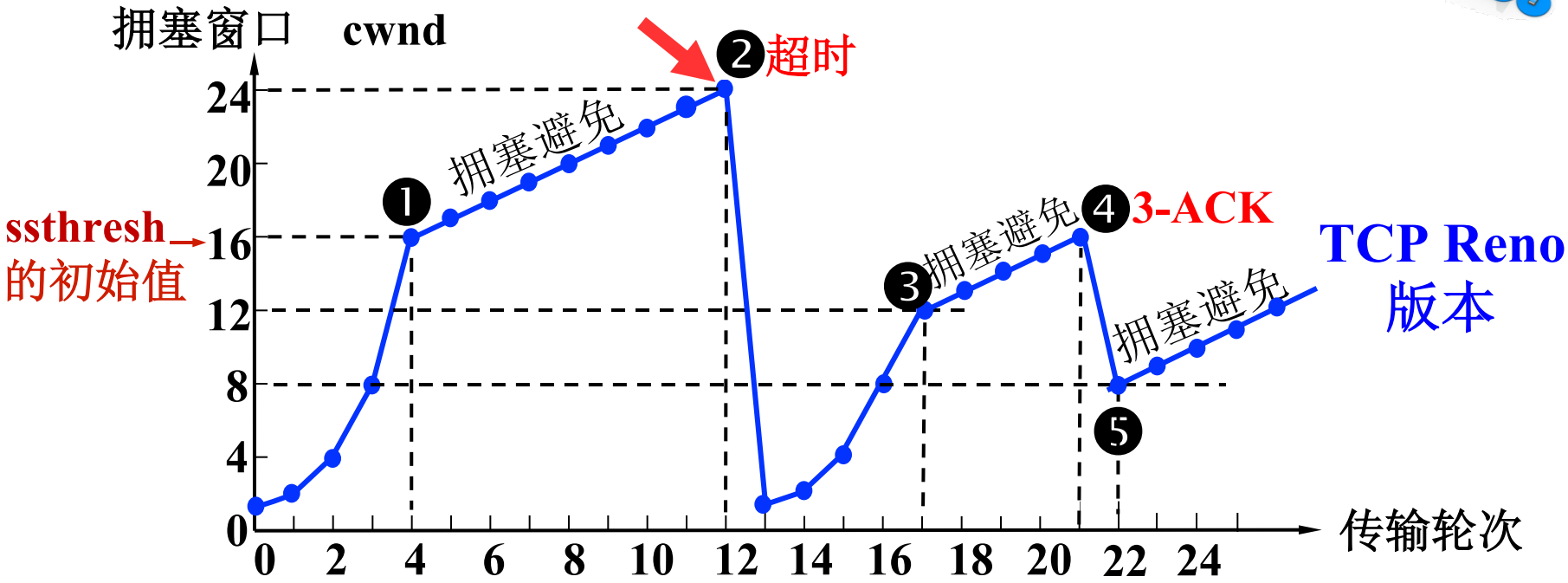
发送方每收到一个对新报文段的确认 ACK，就把拥塞窗口值加 1，然后开始下一轮的传输（请注意，横坐标是传输轮次，不是时间）。因此拥塞窗口 cwnd 随着传输轮次按指数规律增长。

慢开始和拥塞避免算法的实现举例



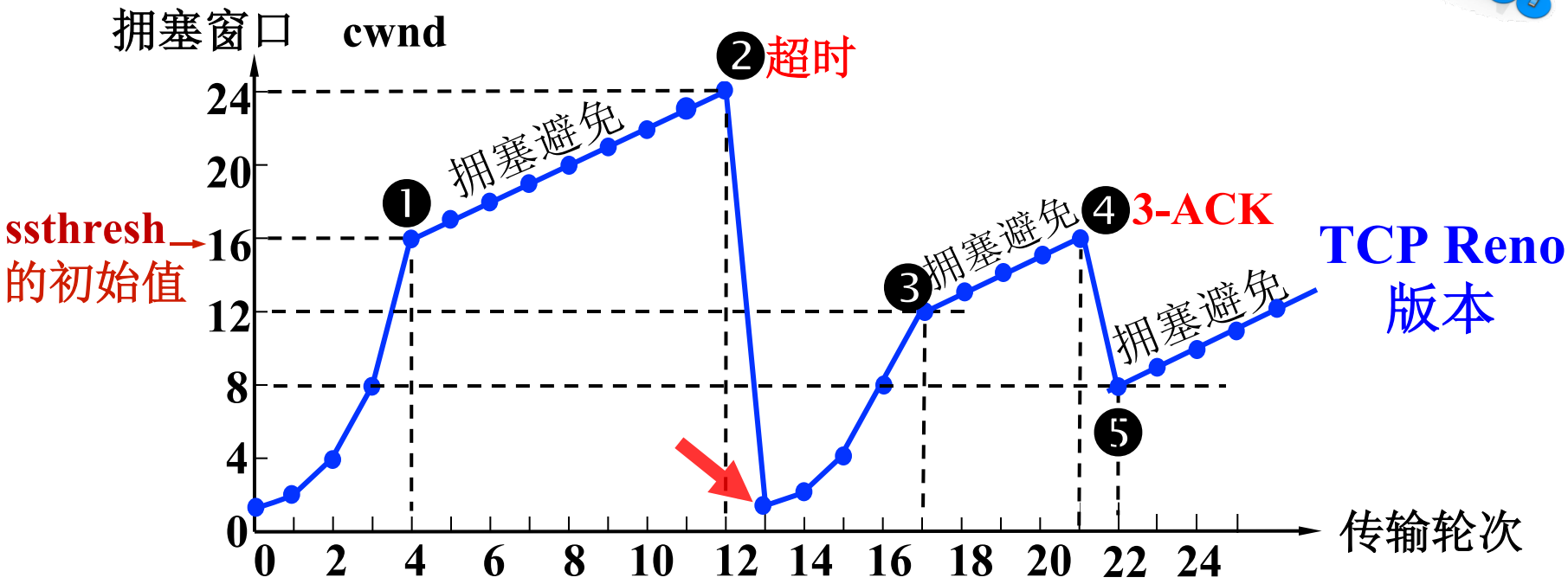
当拥塞窗口 cwnd 增长到慢开始门限值s sthresh时（图中的点❶，此时拥塞窗口cwnd = 16），就改为执行**拥塞避免**算法，拥塞窗口**按线性规律增长**。

慢开始和拥塞避免算法的实现举例



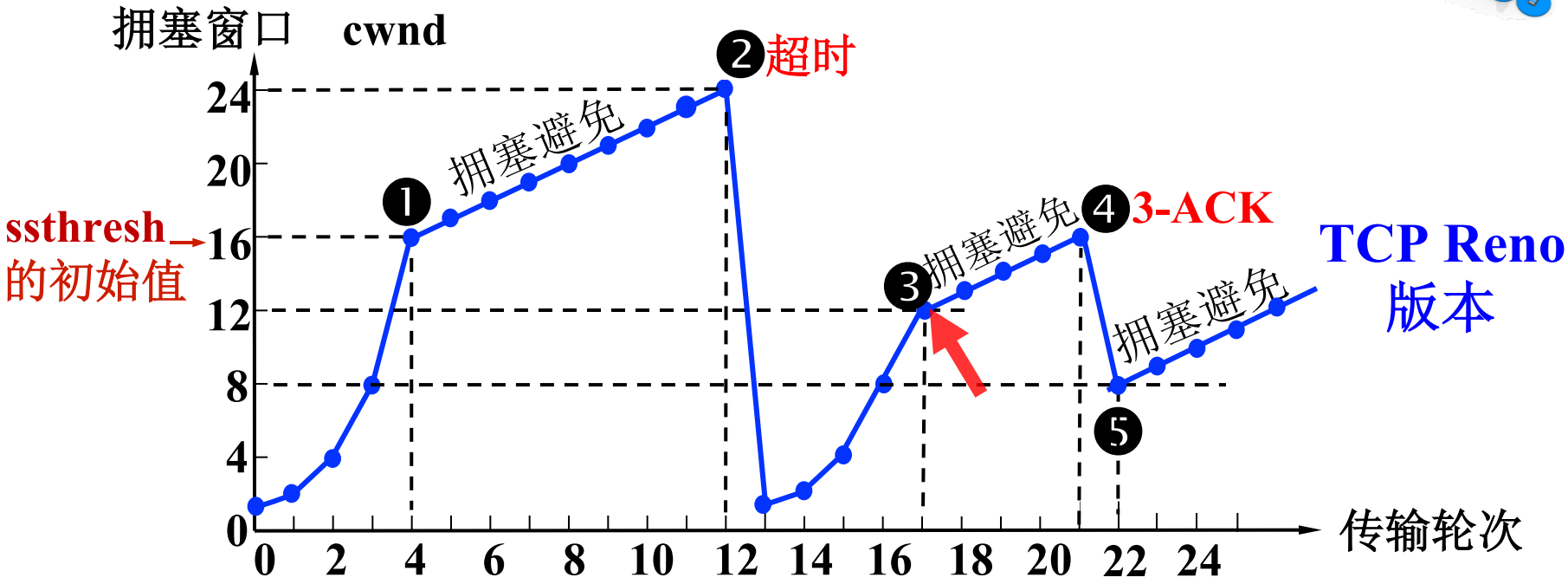
当拥塞窗口 cwnd = 24 时，网络出现了超时（图中的点❷），发送方判断为网络拥塞。于是调整门限值 ssthresh = cwnd / 2 = 12，同时设置拥塞窗口 cwnd = 1，进入慢开始阶段。

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。

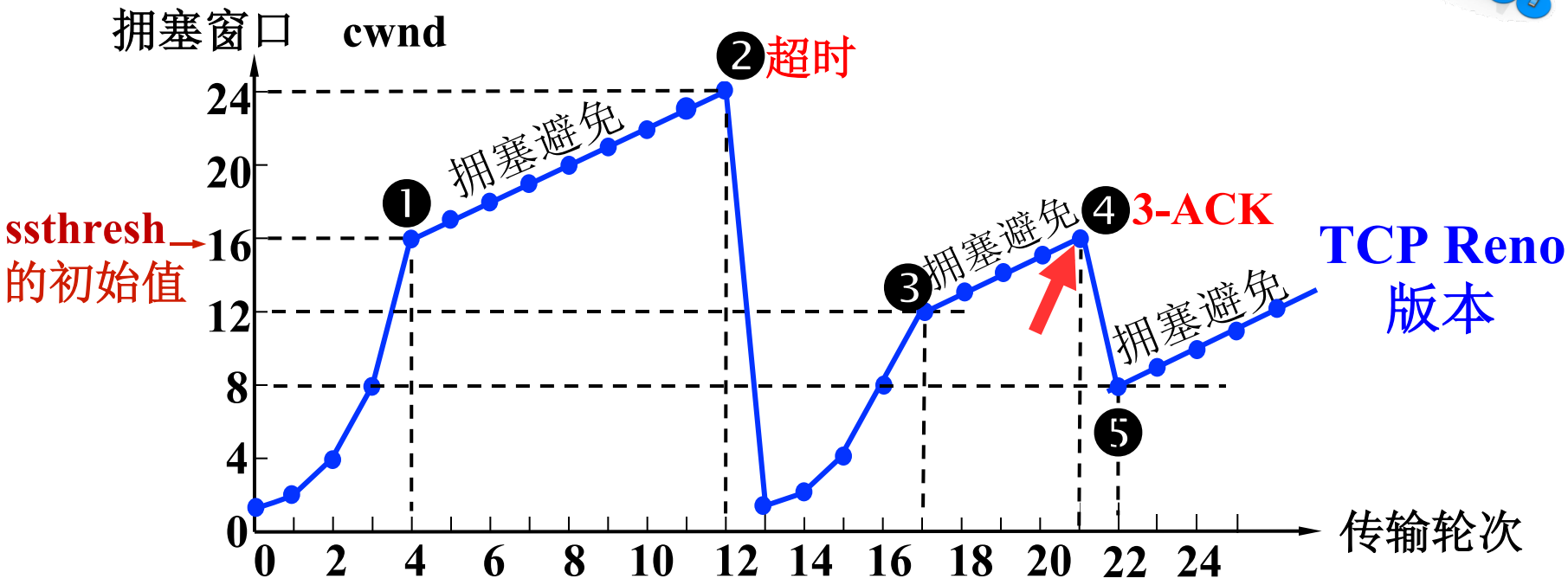
慢开始和拥塞避免算法的实现举例



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

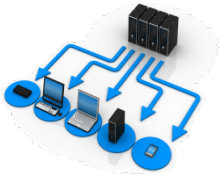


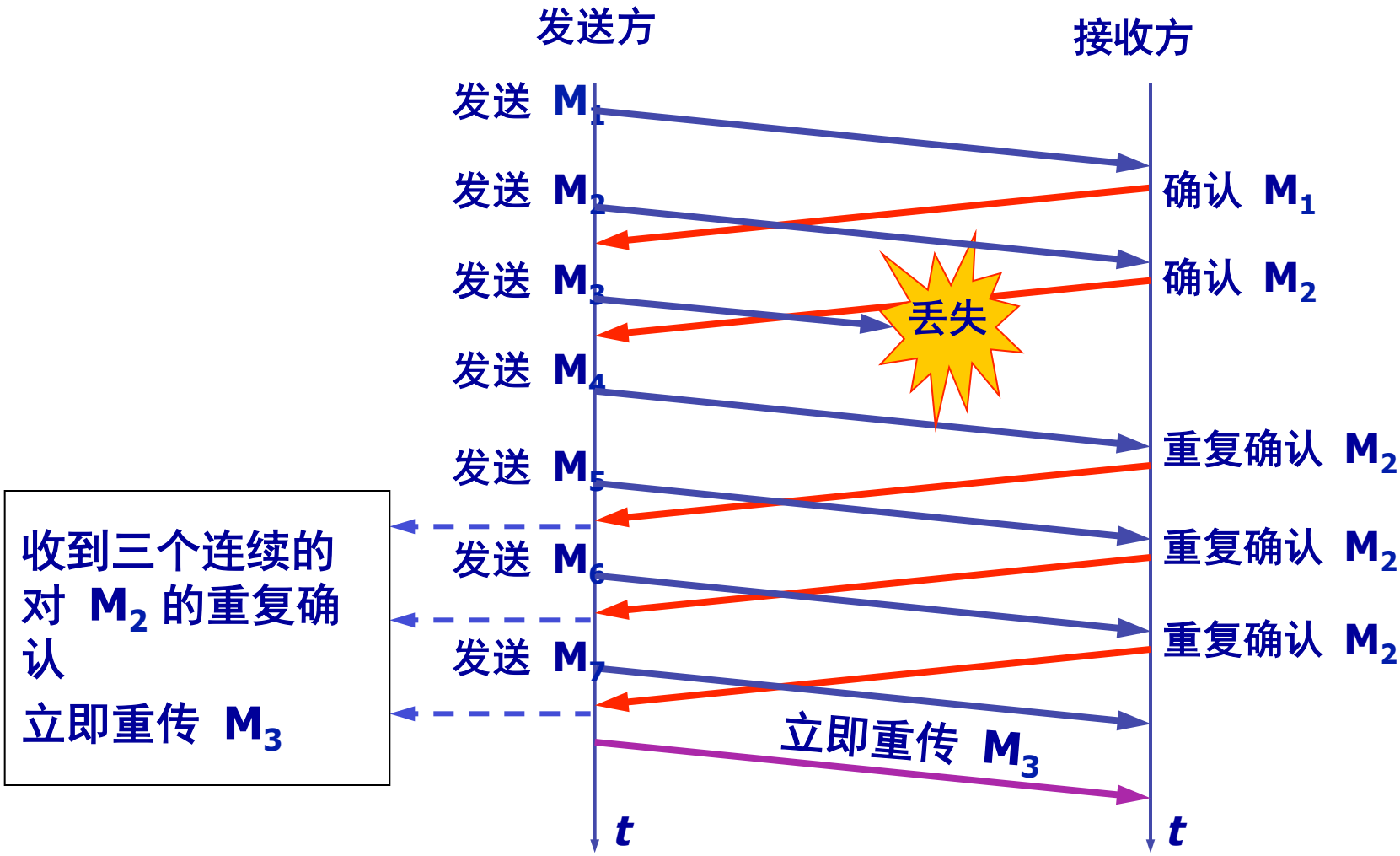
当拥塞窗口cwnd = 16时（图中的点❹），出现了一个新的情况，就是发送方一连收到 3 个对同一个报文段的重复确认（图中记为3-ACK）。发送方改为执行快重传和快恢复算法。

快重传算法

- 发送方只要一连收到三个重复确认，就认为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。
  - 另一种认为快速重传只是代替丢包，发送方还是从慢启动开始
- 使用快重传可以使整个网络的吞吐量提高约20%。

快重传举例




发送方
接收方
发送 M1
发送 M2
发送 M3
发送 M4
发送 M5
发送 M6
发送 M7
确认 M1
确认 M2
丢失
重复确认 M2
重复确认 M2
重复确认 M2
立即重传 M3
t
t
收到三个连续的对 M2 的重复确认
立即重传 M3

快恢复算法

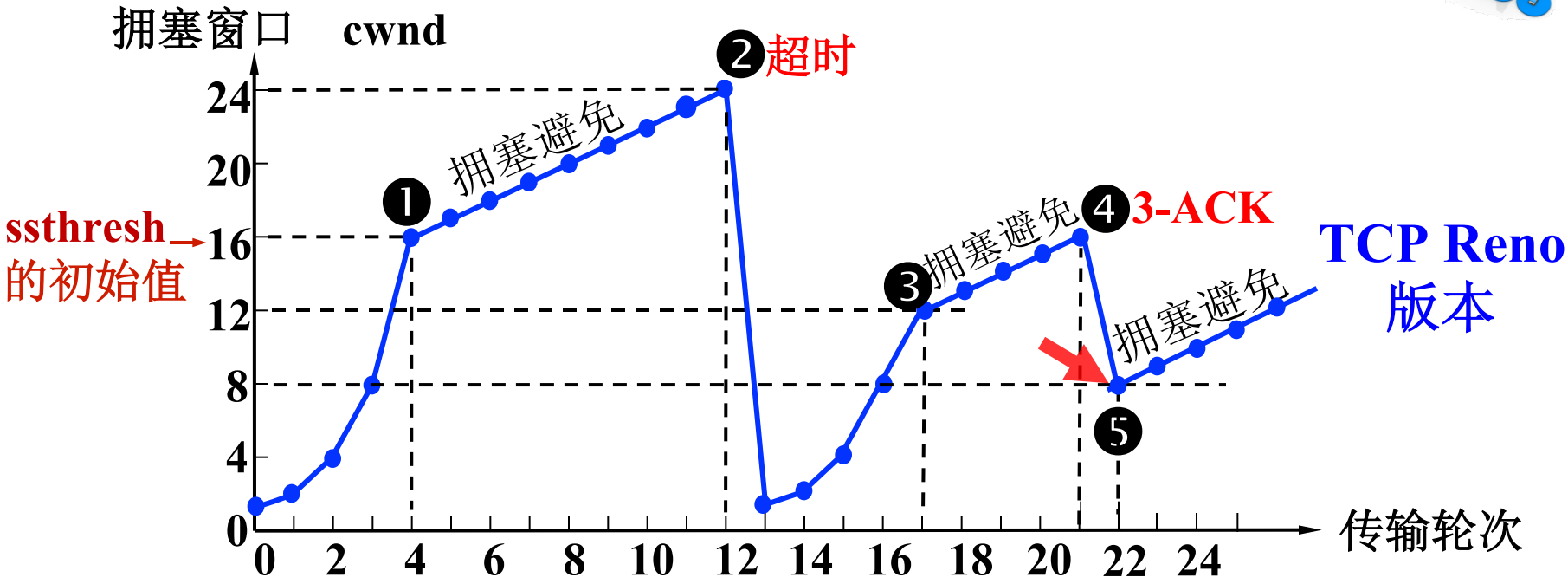
- 当发送端收到连续三个重复的确认时，由于发送方现在认为网络很可能没有发生拥塞，因此现在不执行慢开始算法，而是执行快恢复算法 FR (Fast Recovery) 算法：

  (1) 慢启动门限 ssthresh = 当前拥塞窗口 cwnd / 2；

  (2) 新拥塞窗口 cwnd = 慢启动门限 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。

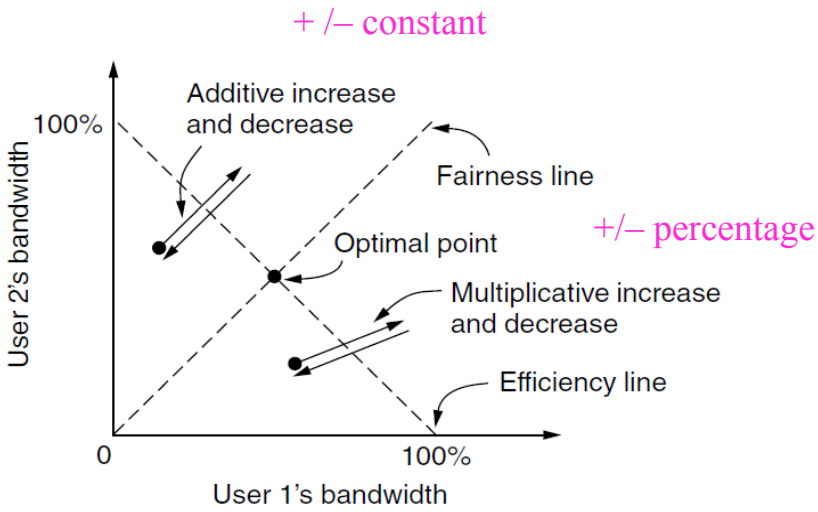
加法递增，乘法递减（AIMD）

- 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“加法递增” 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“乘法递减”MD (Multiplicative Decrease)。
- 二者合在一起就是所谓的 AIMD 算法。

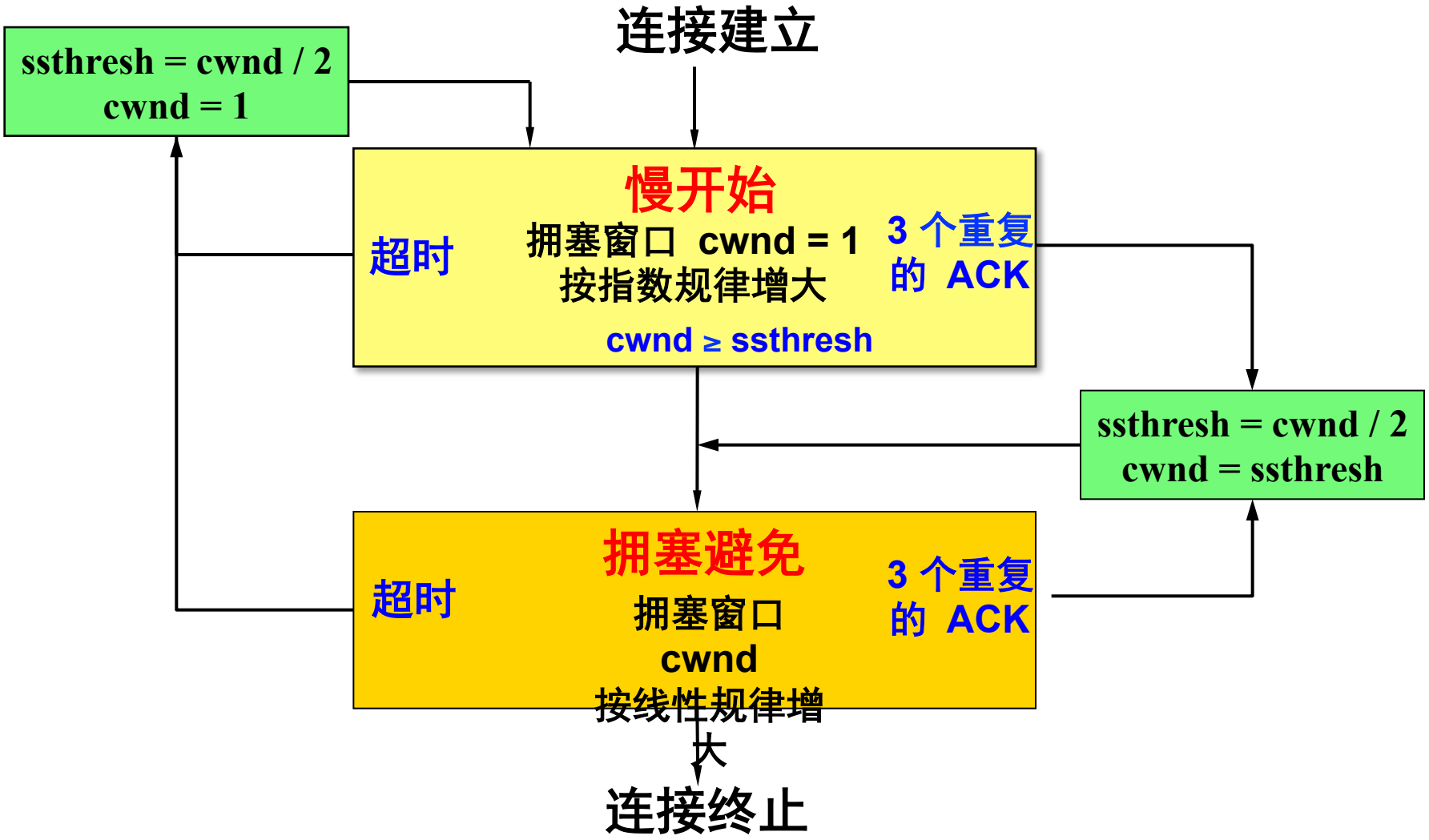
为什么加法递增乘法递减

为了达到公平性

- 加法递增加法递减（AIAD）：在没有拥塞的时候，逐一增加数据流量，当出现拥塞的时候，逐一递减数据流量 (AIAD 无法达到公平)
- 乘法递增乘法递减（MIMD）：在没有拥塞的时候，成倍增加（如增加10%），当出现拥塞的时候成倍相减 (MIMD依然无法达到公平)
- 怎么样可以达到公平呢?



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

5.8.3 主动队列管理 AQM

- 丢包
  - 传输错误
  - 队列满了
- 队列满了，怎么丢包
  - FIFO
  - RED
- 顺便提个问题
  - 有了端对端的可靠性，还需要点对点的可靠性吗？
  - 或者有了点对点的可靠性，还需要端对端的可靠性吗？

“先进先出”FIFO 处理规则

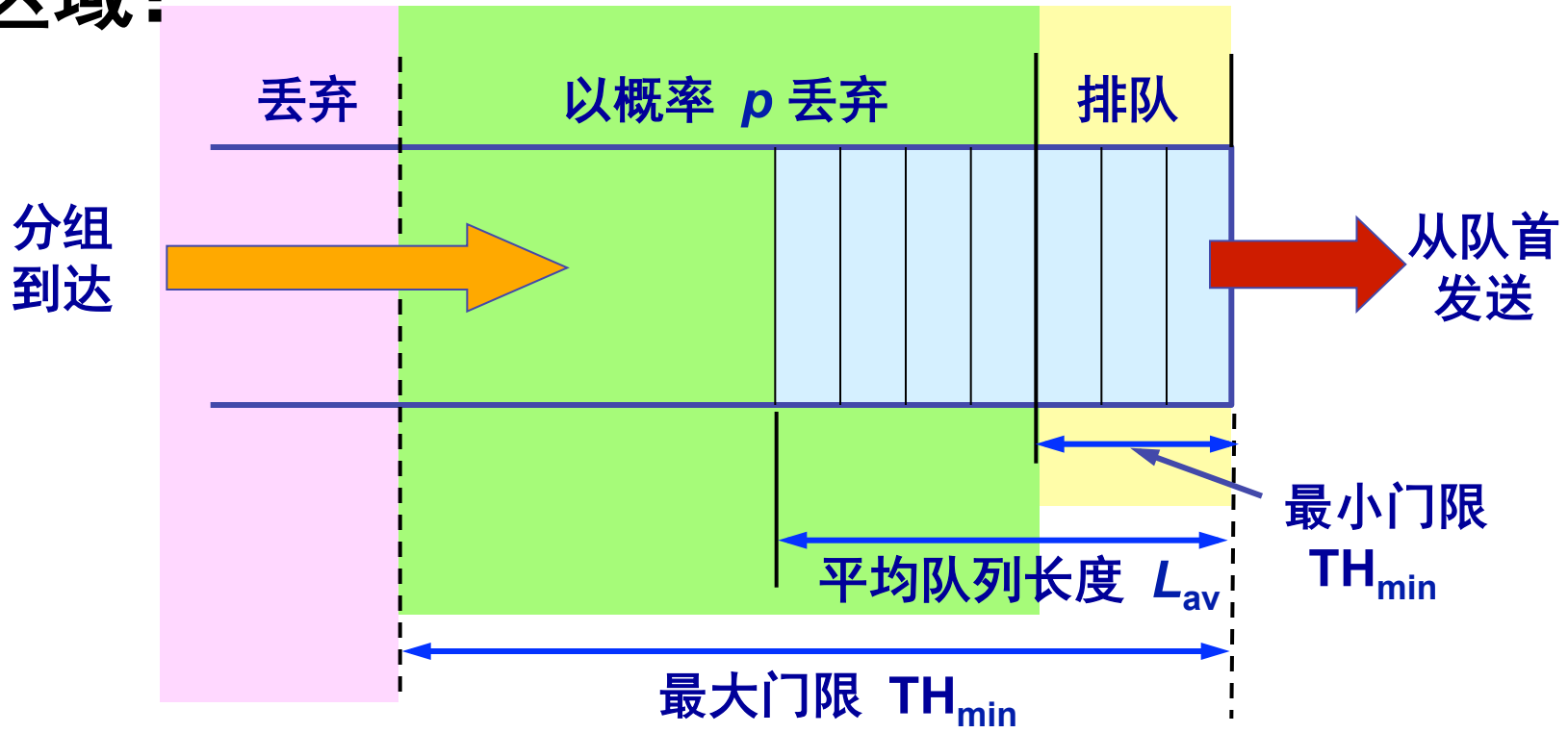
- 路由器的队列通常都是按照“先进先出”FIFO (First In First Out) 的规则处理到来的分组。
  - 尾部丢弃队列策略 (tail-drop policy)。
  - 被丢弃包的TCP连接进入慢启动。
- 全局同步
  - 尾部丢弃，使许多 TCP 连接在同一时间突然都进入到慢开始状态。
  - 全局同步使得全网的通信量突然下降了很多，而在网络恢复正常后，其通信量又突然增大很多

主动队列管理 AQM-RED

- 思想：不等到队列满的时候再丢弃包，提早随机丢包
  - 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} 。
  - (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

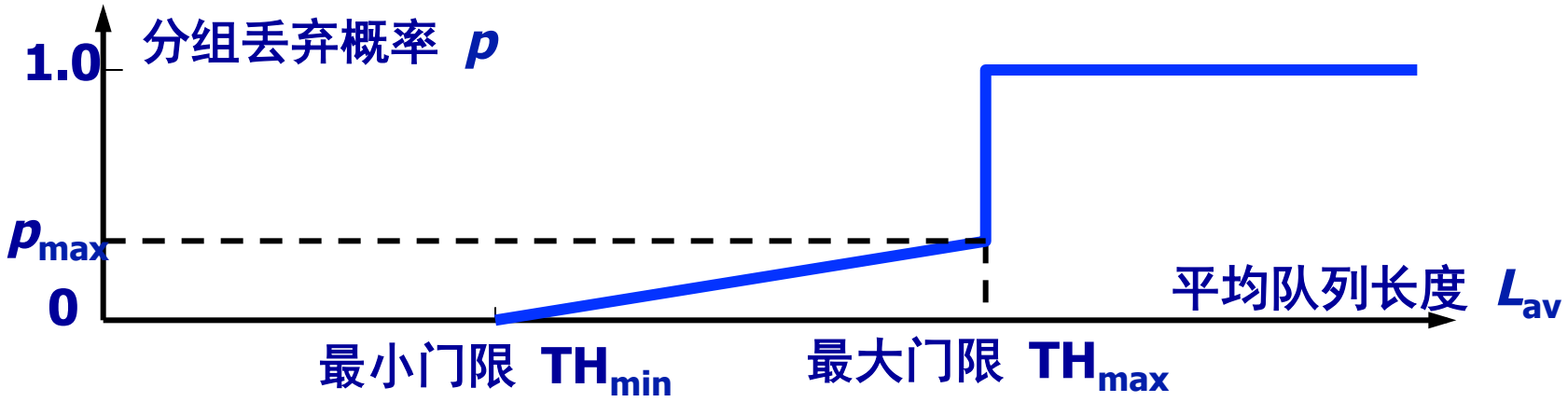
RED 将路由器的到达队列划分成为三个区域：



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。

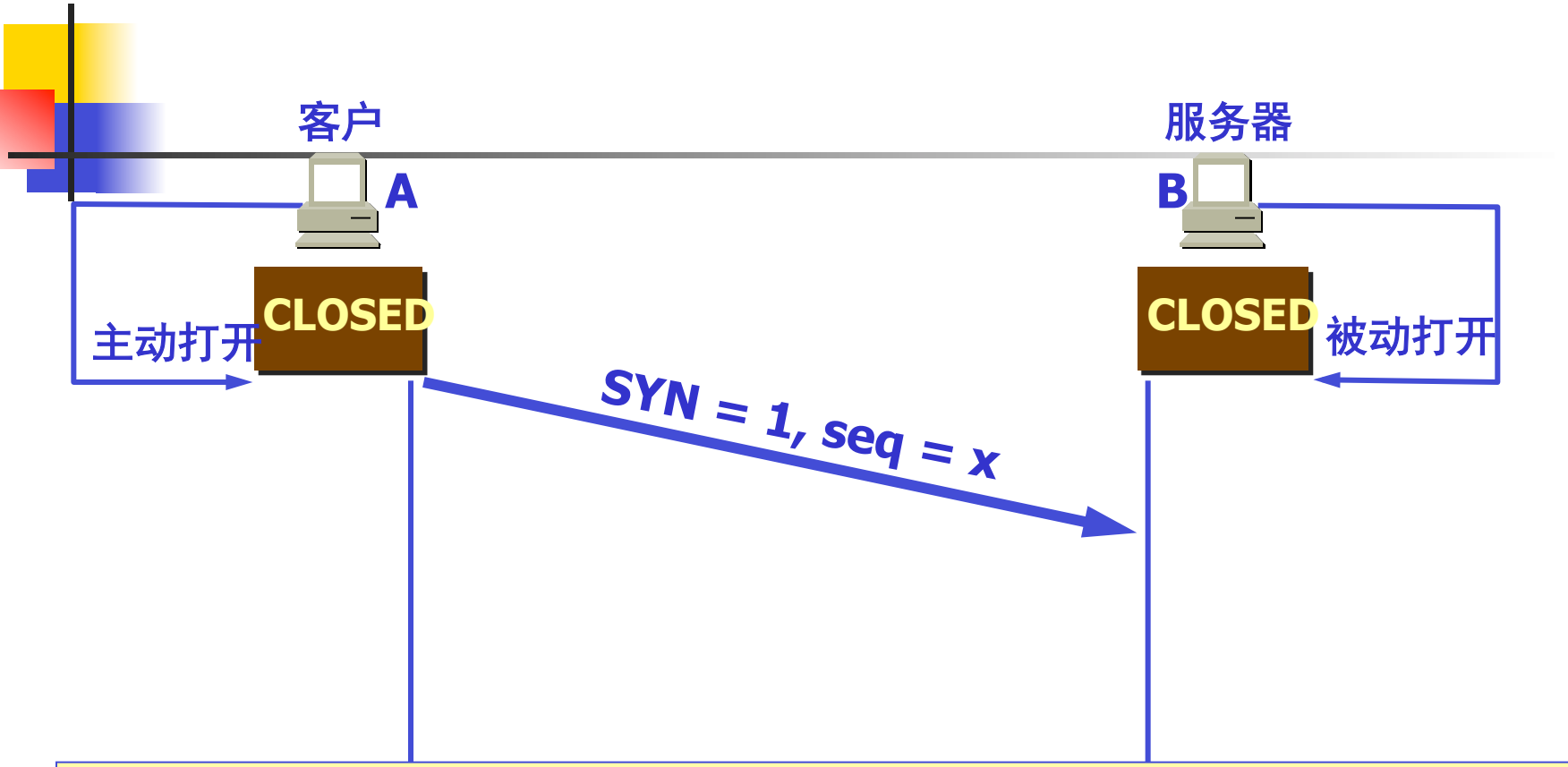
TCP 连接建立

- 建立链接本质上是做什么。
  - 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）。
  - 对资源（如缓存大小、连接表中的项目等）进行分配。
- 采用客户服务器方式建立链接

5.9.1 TCP 的连接建立

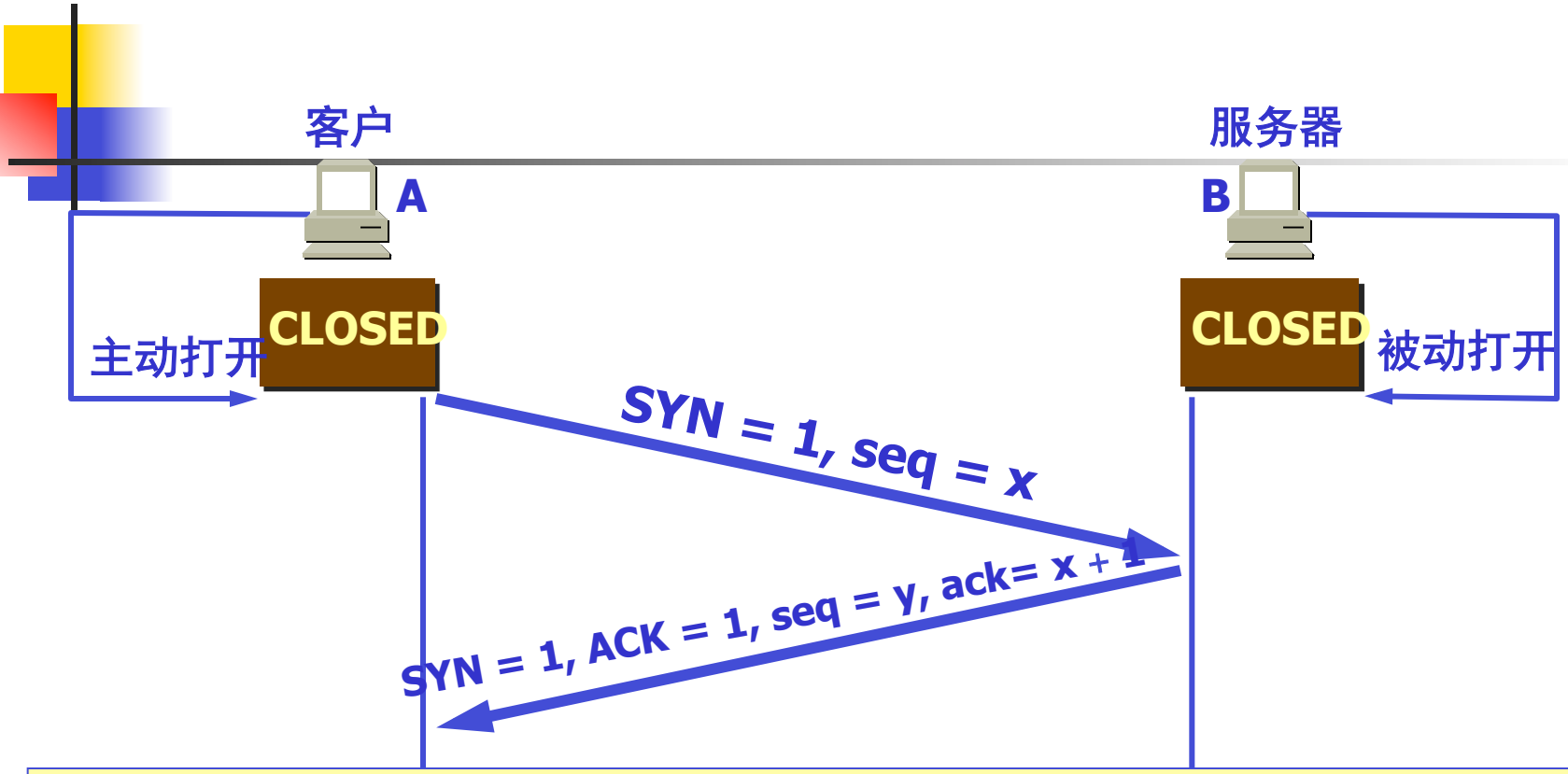
- TCP 建立连接的过程叫做握手。
- 握手需要在客户和服务器之间交换三个 TCP 报文段。称之为三报文握手。
- 采用三报文握手主要是为了防止已失效的连接请求报文段突然又传送到了，因而产生错误。

TCP 的连接建立：采用三报文握手



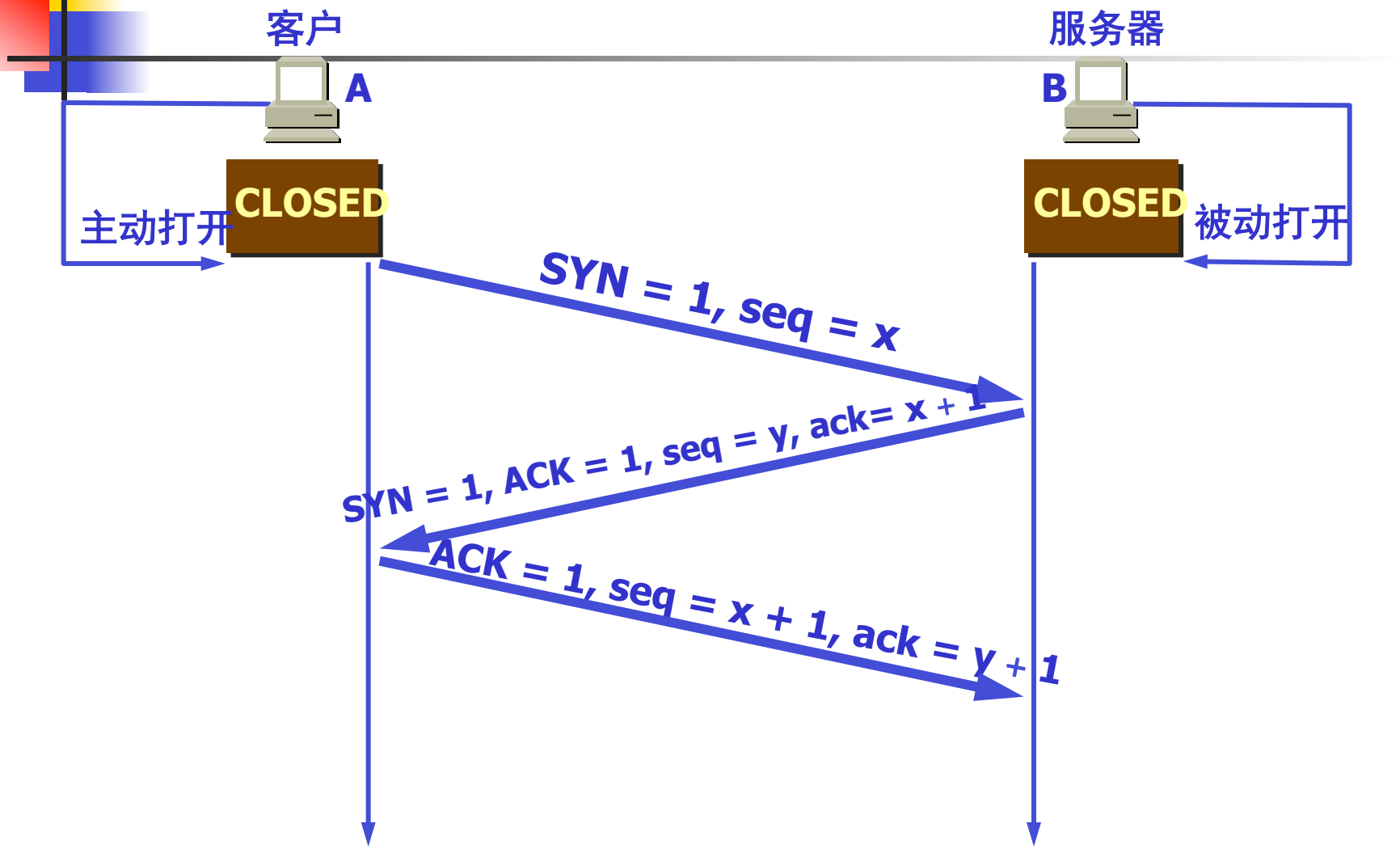
A 的 TCP 向 B 发出连接请求报文段，其首部中的同步位 SYN = 1，并选择序号 seq = x，表明传送数据时的第一个数据字节的序号是 x。

TCP 的连接建立：采用三报文握手



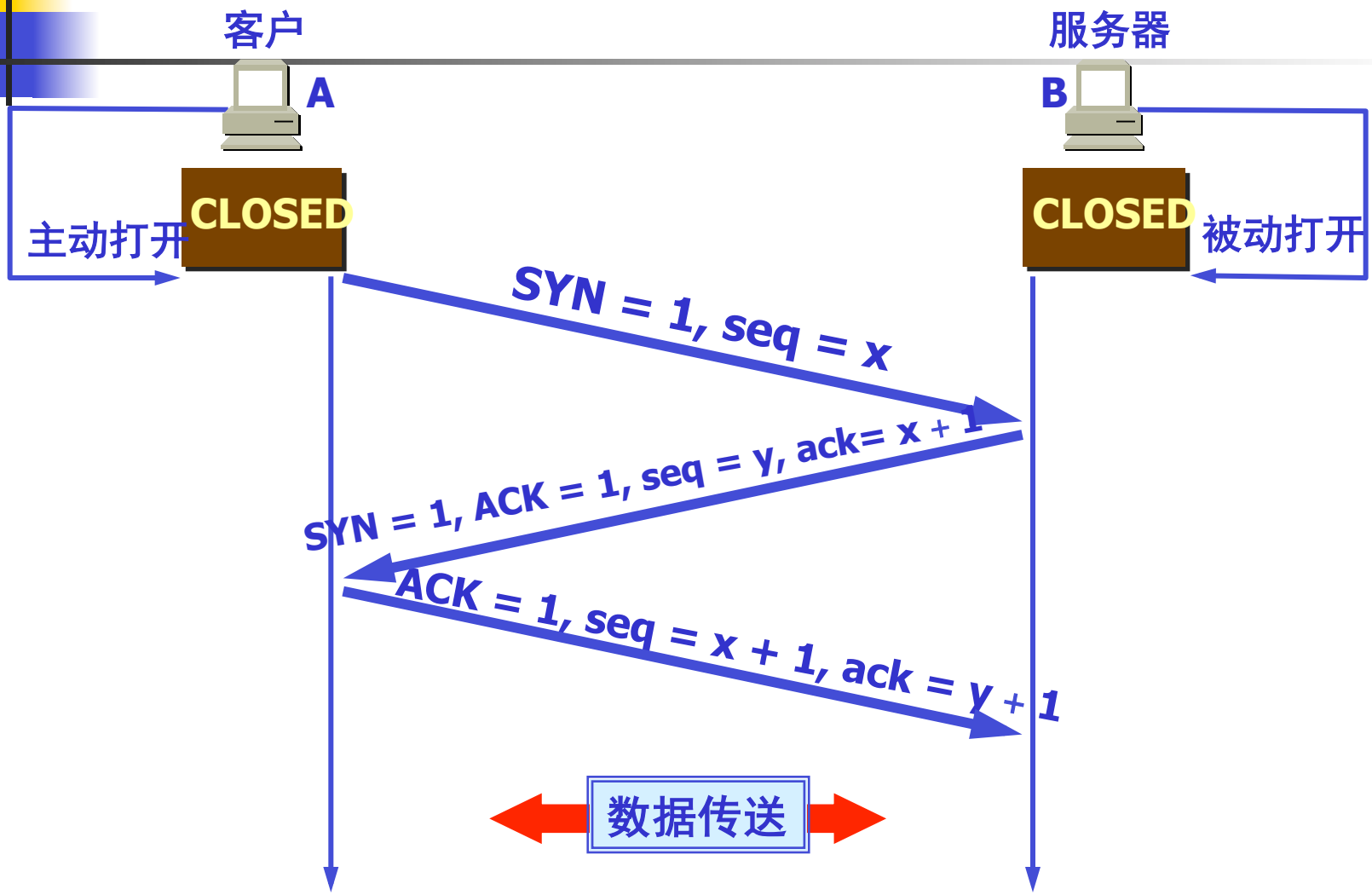
- B 的 TCP 收到连接请求报文段后，如同意，则发回确认。
- B 在确认报文段中应使 SYN = 1，使 ACK = 1，其确认号ack = x + 1，自己选择的序号 seq = y。

- 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

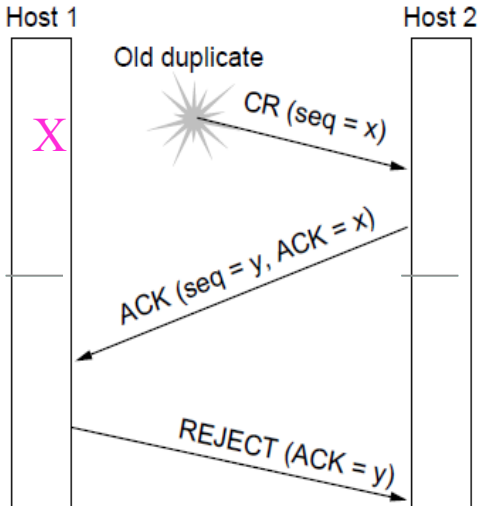
  应用进程：TCP 连接已经建立。



TCP连接的建立

- 为什么要三次握手，两次不行吗?

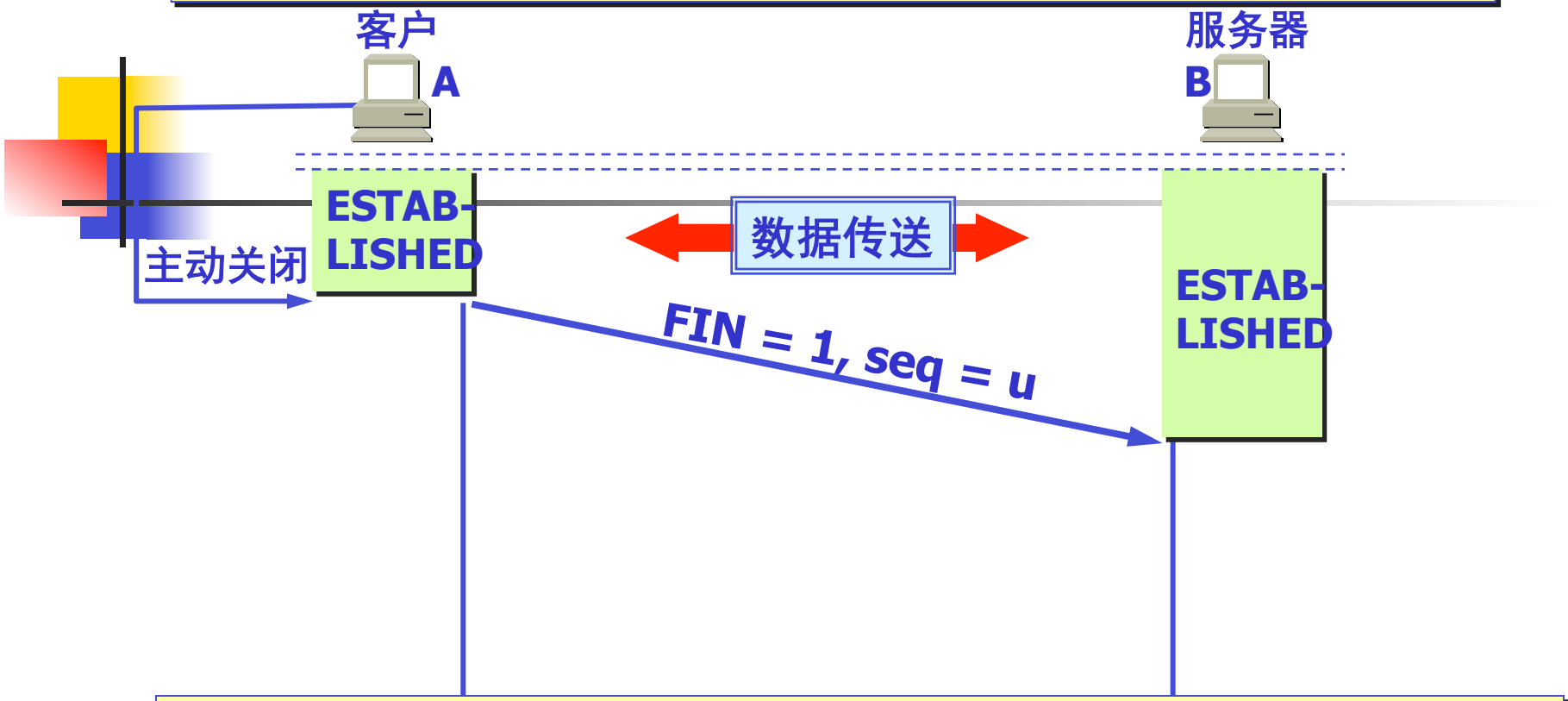
a) **重复的CR（connection request）.**
   - **两次握手，Old的连接建立请求会被B端响应，并建立连接（二次握手，B端连接在发出ACK后建立）**
   - **三次握手，A端收到对Old的ACK，但A端知道这是旧的连接请求，不会对此ACK进行确认，所以B端不会建立连接（三次握手，B端在收到对ACK的确认后建立连接）**



5.9.2 TCP 的连接释放

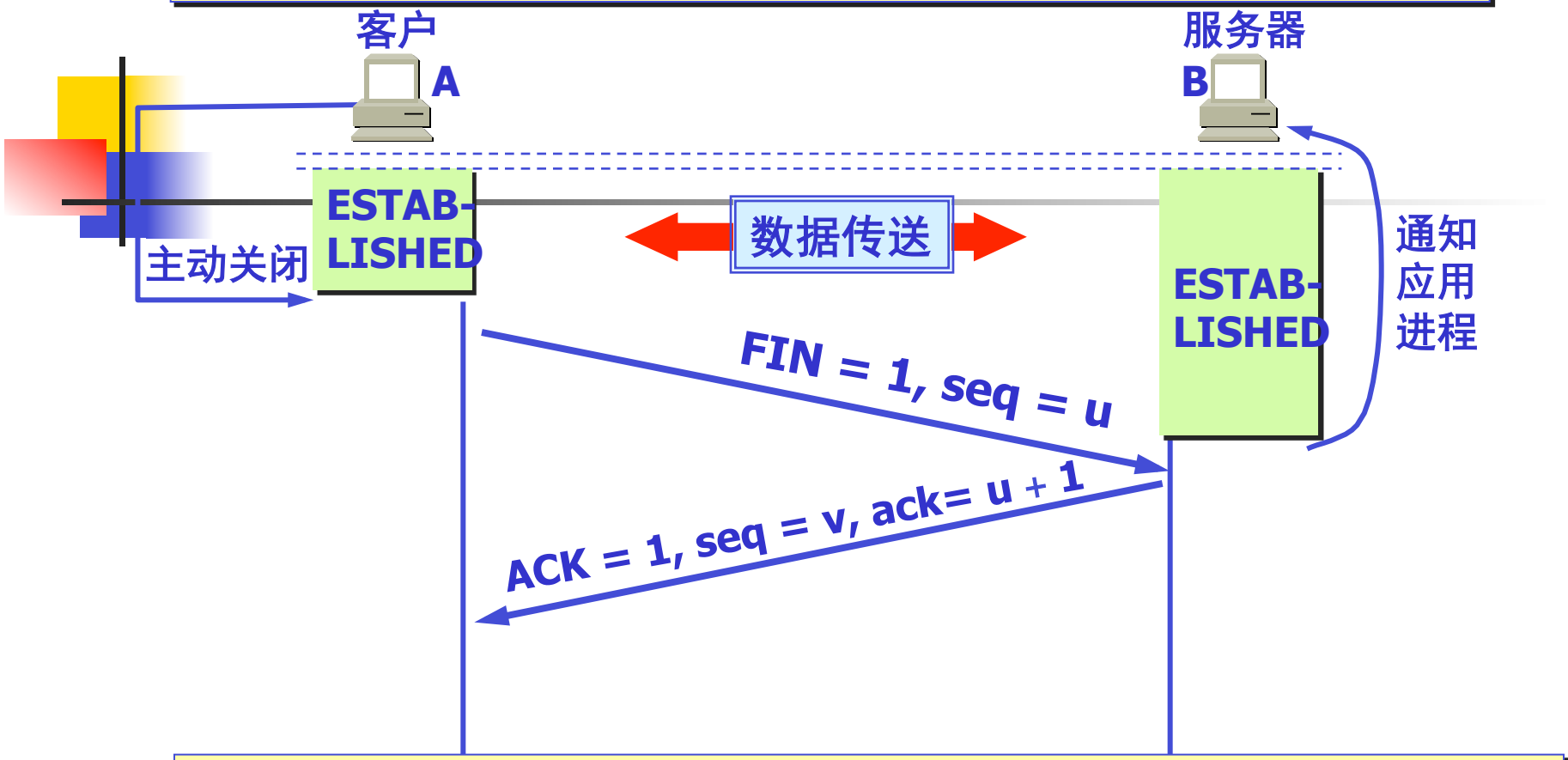
- TCP 连接释放过程比较复杂。
- 数据传输结束后，通信的双方都可释放连接。
- TCP 连接释放过程是四报文握手。

TCP 的连接释放：采用四报文握手



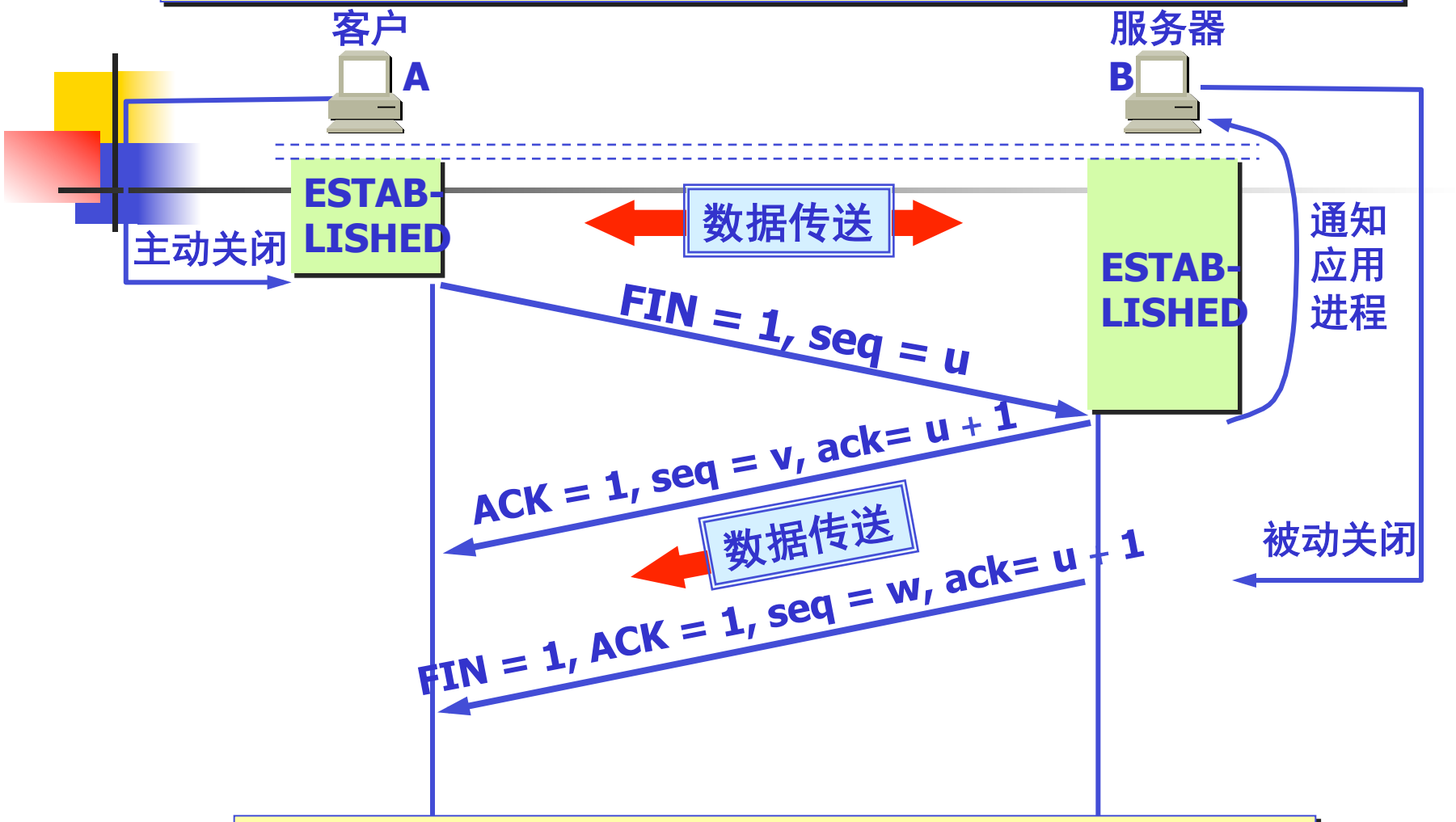
- 数据传输结束后，通信的双方都可释放连接。
- 现在 A 的应用进程先向其 TCP 发出连接释放报文段，并停止再发送数据，主动关闭 TCP 连接。
- A 把连接释放报文段首部的 FIN = 1，其序号 seq = u，等待 B 的确认。

TCP 的连接释放：采用四报文握手



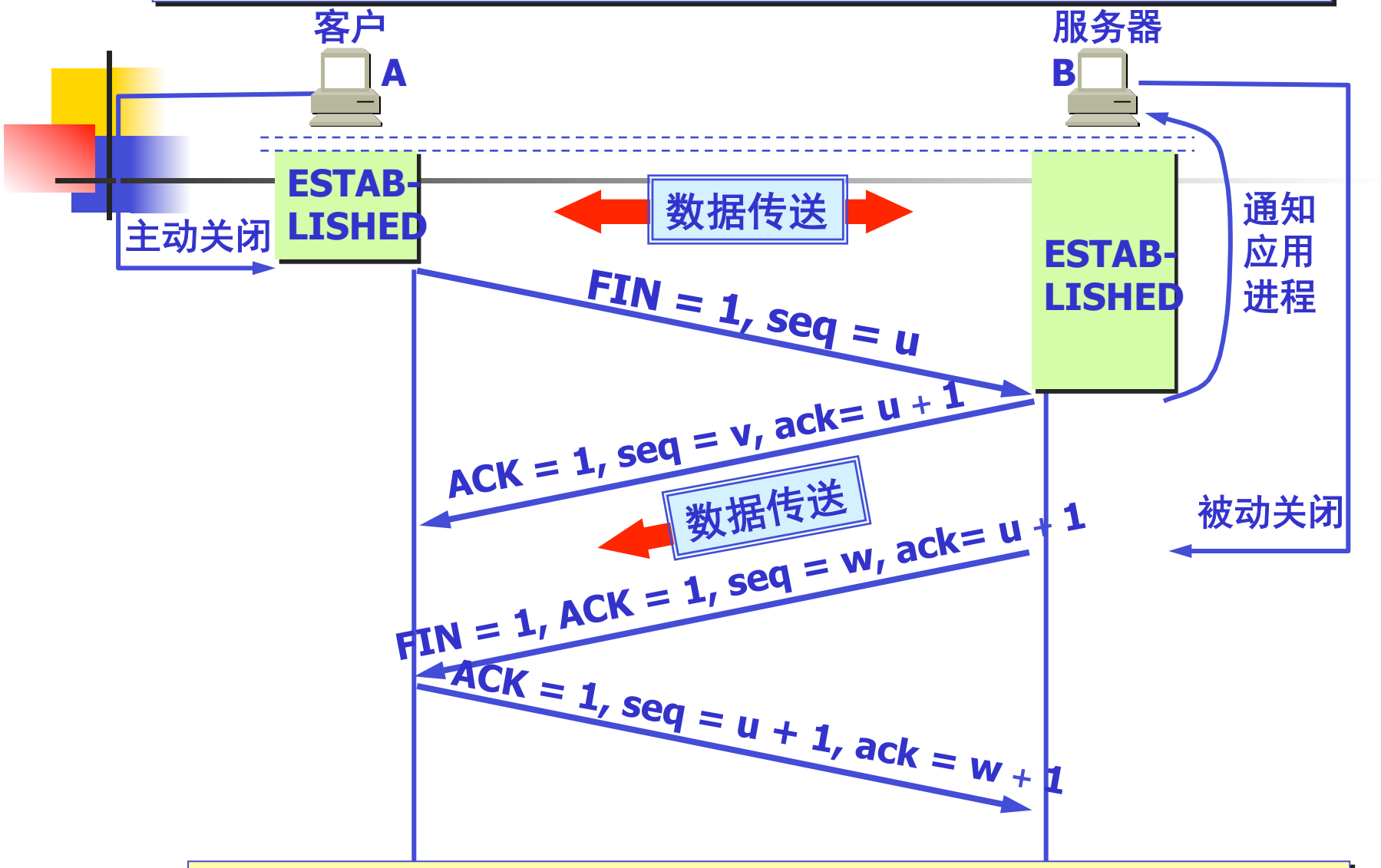
- B 发出确认，确认号 ack = u + 1，而这个报文段自己的序号 seq = v。
- TCP 服务器进程通知高层应用进程。
- 从 A 到 B 这个方向的连接就释放了，TCP 连接处于半关闭状态。B 若发送数据，A 仍要接收。

TCP 的连接释放：采用四报文握手



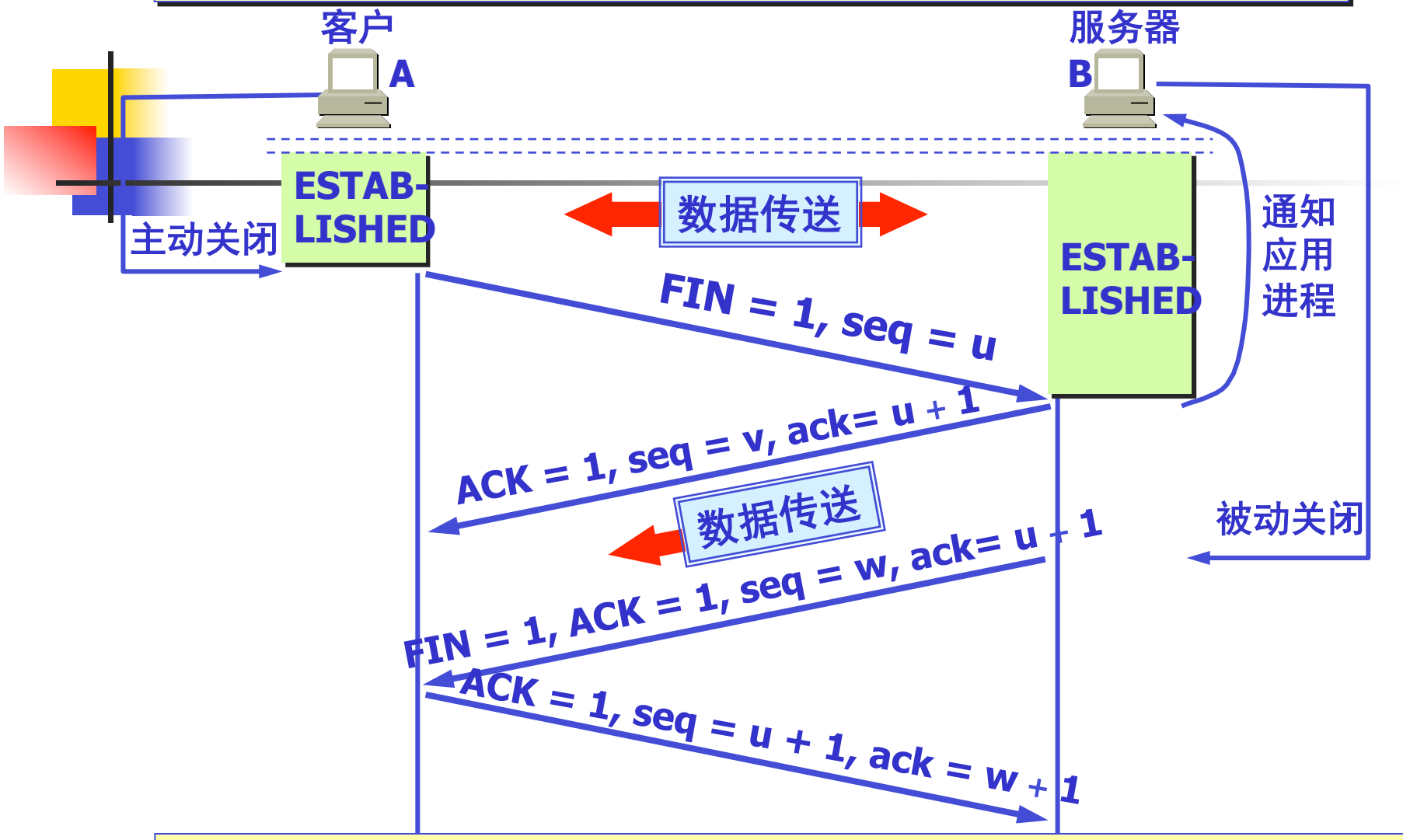
- 若 B 已经没有要向 A 发送的数据，其应用进程就通知 TCP 释放连接。

TCP 的连接释放：采用四报文握手



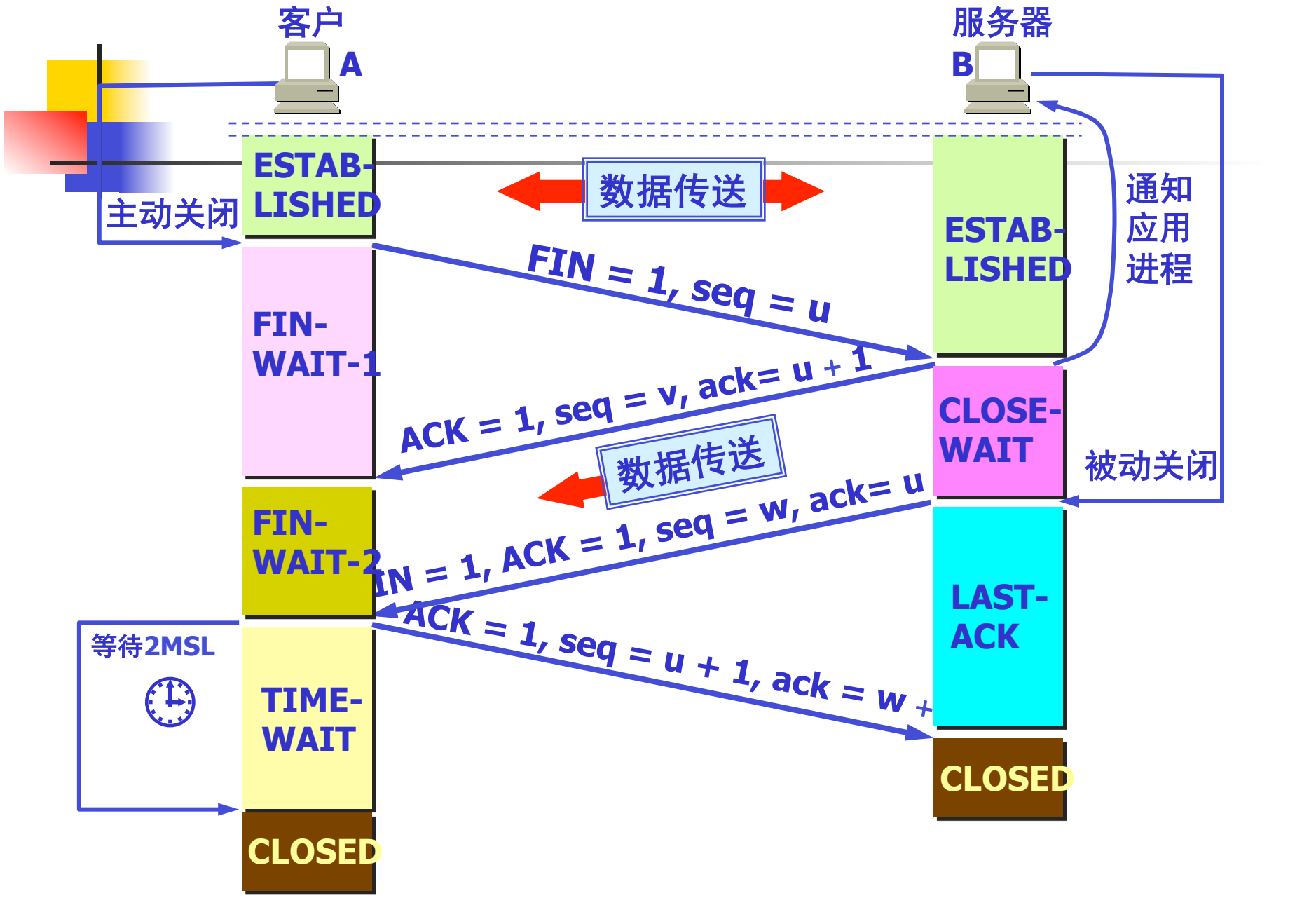
- A 收到连接释放报文段后，必须发出确认。

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。



问题1

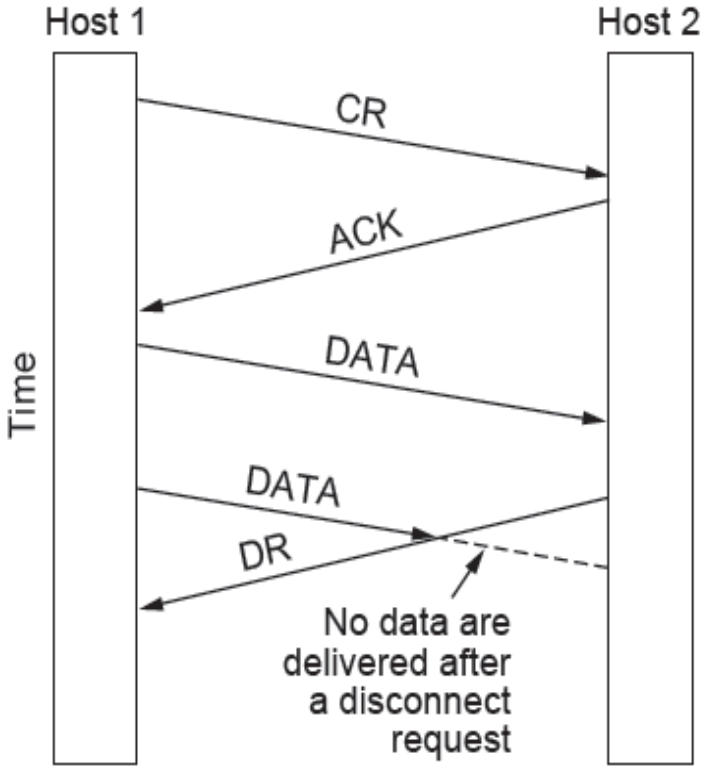
- A 必须等待 2MSL 的时间
  - 第一，为了保证 A 发送的最后一个 ACK 报文段能够到达 B，即ACK丢失，会重发。如果没有2MSL时间，直接关闭，则ACK丢失，A无法重发（连接已经关闭），B端的连接无法关闭。
  - 第二，防止 “已失效的连接请求报文段”出现在本连接中。A 在发送完最后一个 ACK 报文段后，再经过时间 2MSL，就可以使本连接持续的时间内所产生的所有报文段，都从网络中消失。这样就可以使下一个新的连接中不会出现这种旧的连接请求报文段。

问题2

- 为什么在TCP协议里，建立连接是三次握手，而关闭连接却是四次挥手？
  - 如果因为当处于LISTEN状态的服务器端收到来自客户端的SYN报文(客户端希望新建一个TCP连接)时，它可以把ACK(确认应答)和SYN(同步序号)放在同一个报文里来发送给客户端。但在关闭TCP连接时，当收到对方的FIN报文时，对方仅仅表示对方已经没有数据发送给你了，但是你自己可能还有数据需要发送给对方，则等你发送完剩余的数据给对方之后，再发送FIN报文给对方来表示你数据已经发送完毕，并请求关闭连接，所以通常情况下，这里的ACK报文和FIN报文都是分开发送的。

问题3

- 两次握手呢？
  - 如果ACK丢失，需要再次发送Fin，但此时对方链接已经关闭



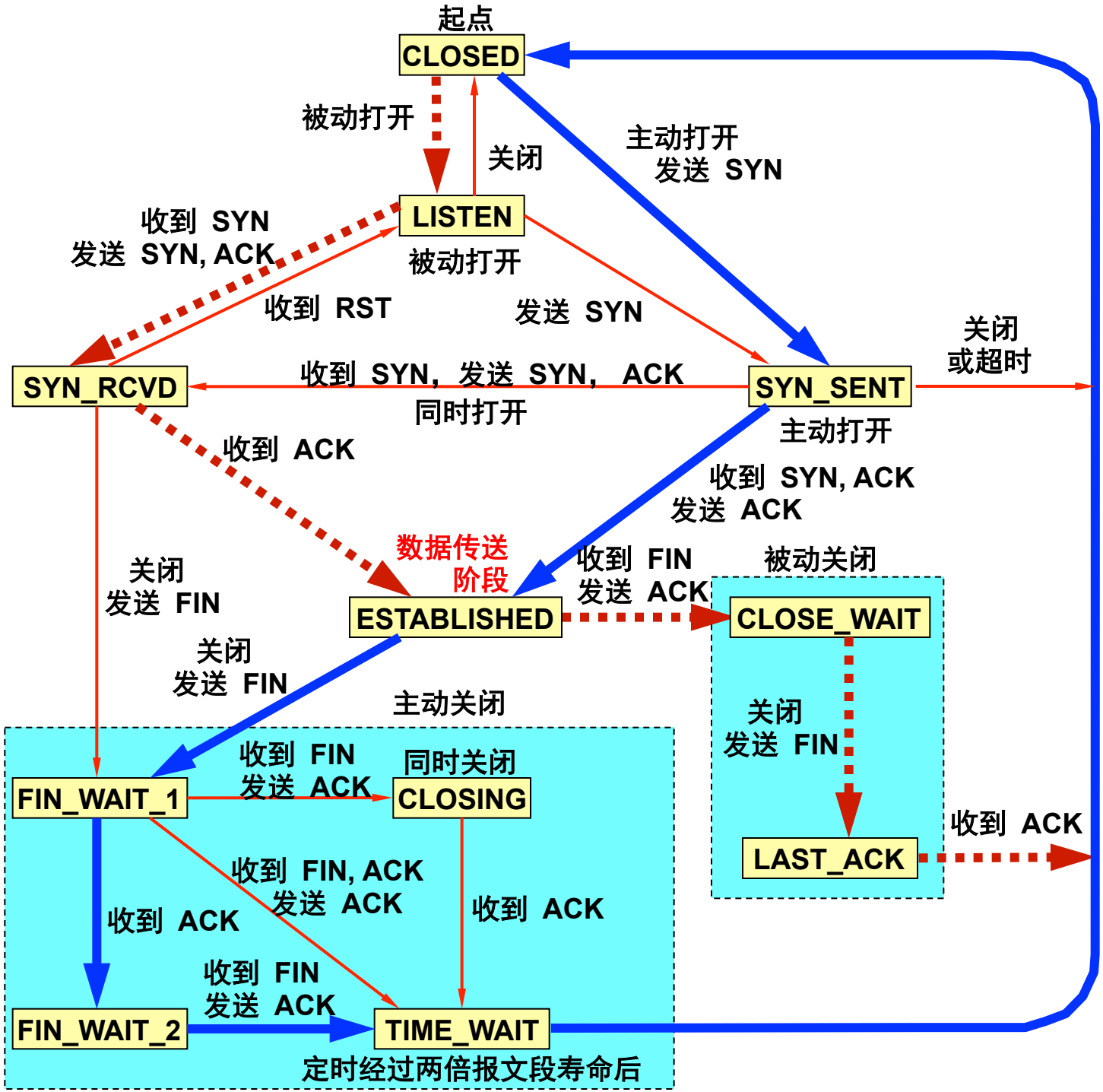
5.9.3 TCP 的有限状态机

- 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


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

作业

- 5-02, 05, 07, 08, 09, 11, 13, 14, 16, 18, 19, 22, 23, 24, 28, 29, 31, 32, 34, 37, 39, 45, 46, 47, 49