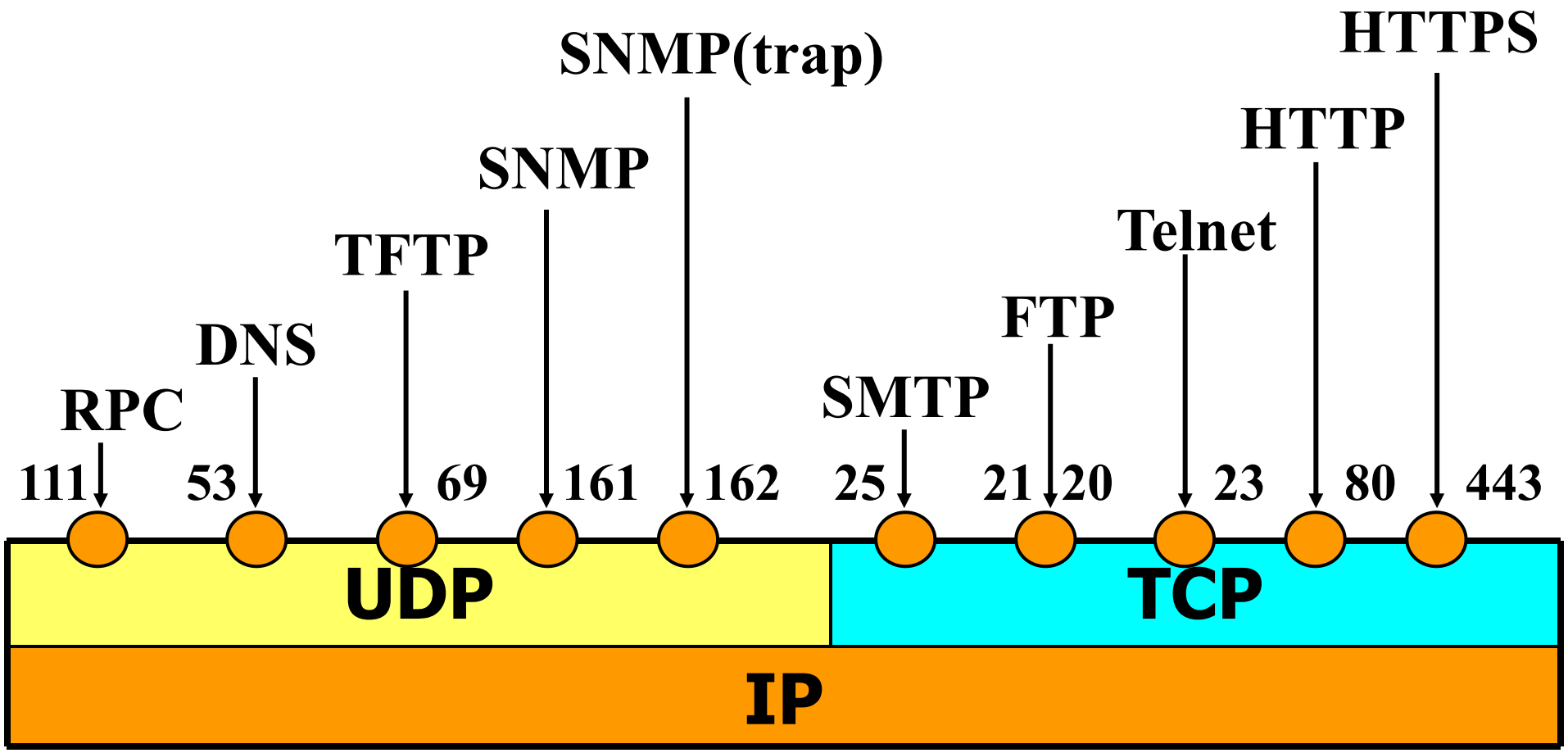
第 5 章 运输层

第 5 章 运输层

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

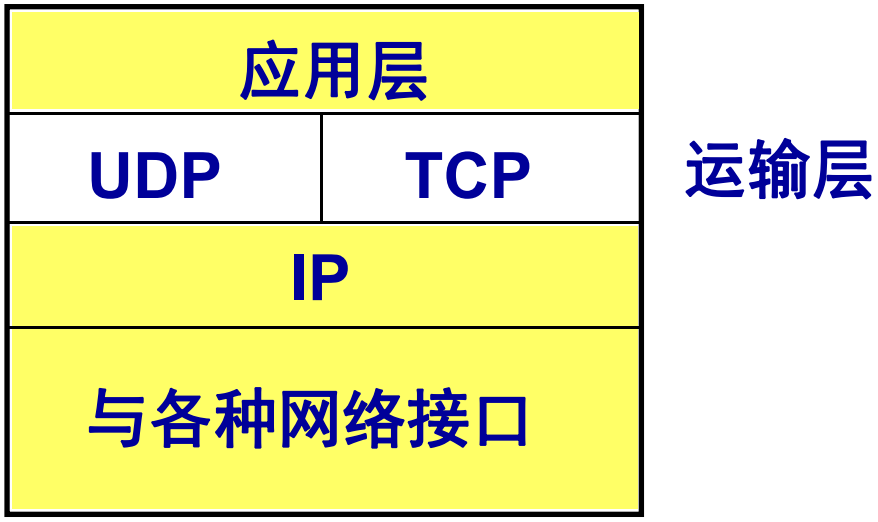
常用的熟知端口


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

5.1.2 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的区别

	UDP	TCP
是否连接	无连接	面向连接
是否可靠	不可靠传输，不使用流量控制和拥塞控制	可靠传输，使用流量控制和拥塞控制
连接对象个数	支持一对一，一对多，多对一和多对多交互通信	只能是一对一通信
传输方式	面向报文	面向字节流
首部开销	首部开销小，仅8字节	首部最小20字节，最大60字节
适用场景	适用于实时应用（IP电话、视频会议、直播等）	适用于要求可靠传输的应用，例如文件传输

5.2.1 UDP概述

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

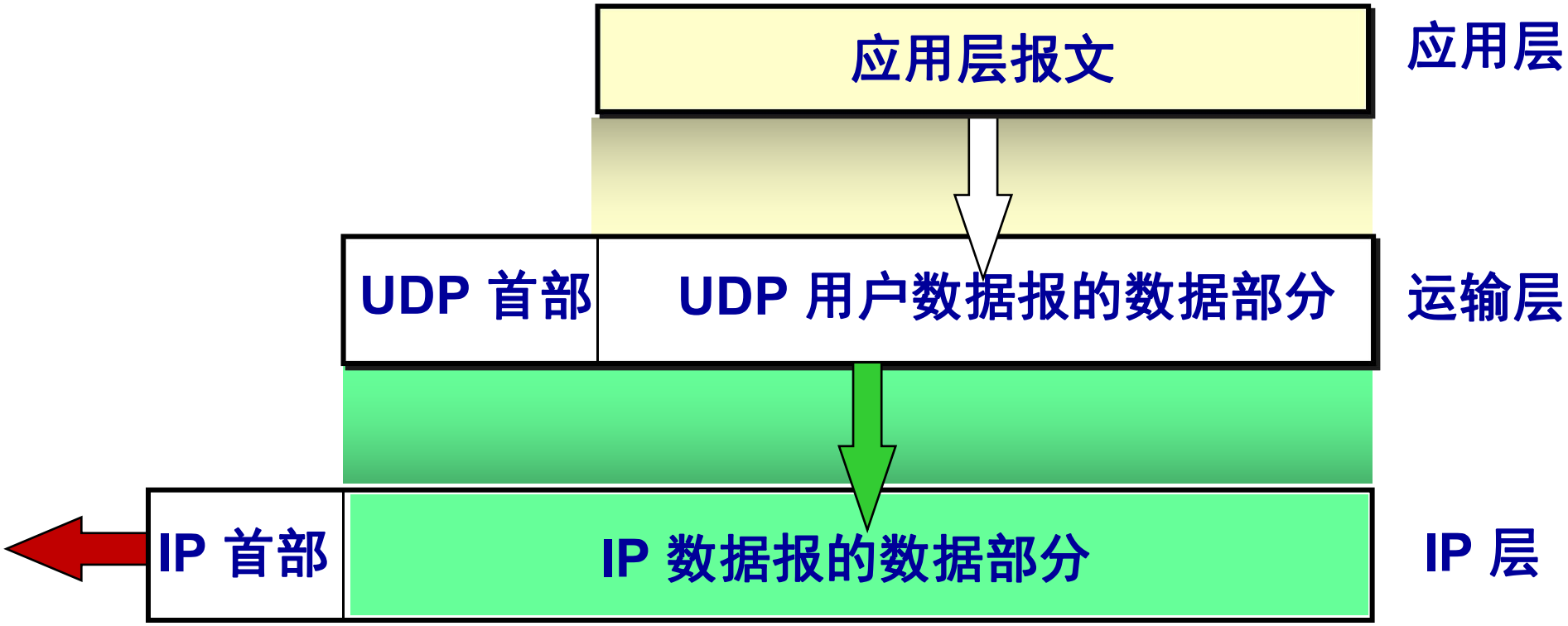
UDP 的主要特点

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

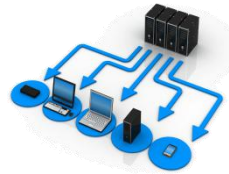
UDP 的主要特点

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

UDP 是面向报文的



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。

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。

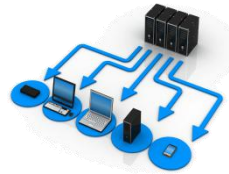
套接字 (socket)

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)

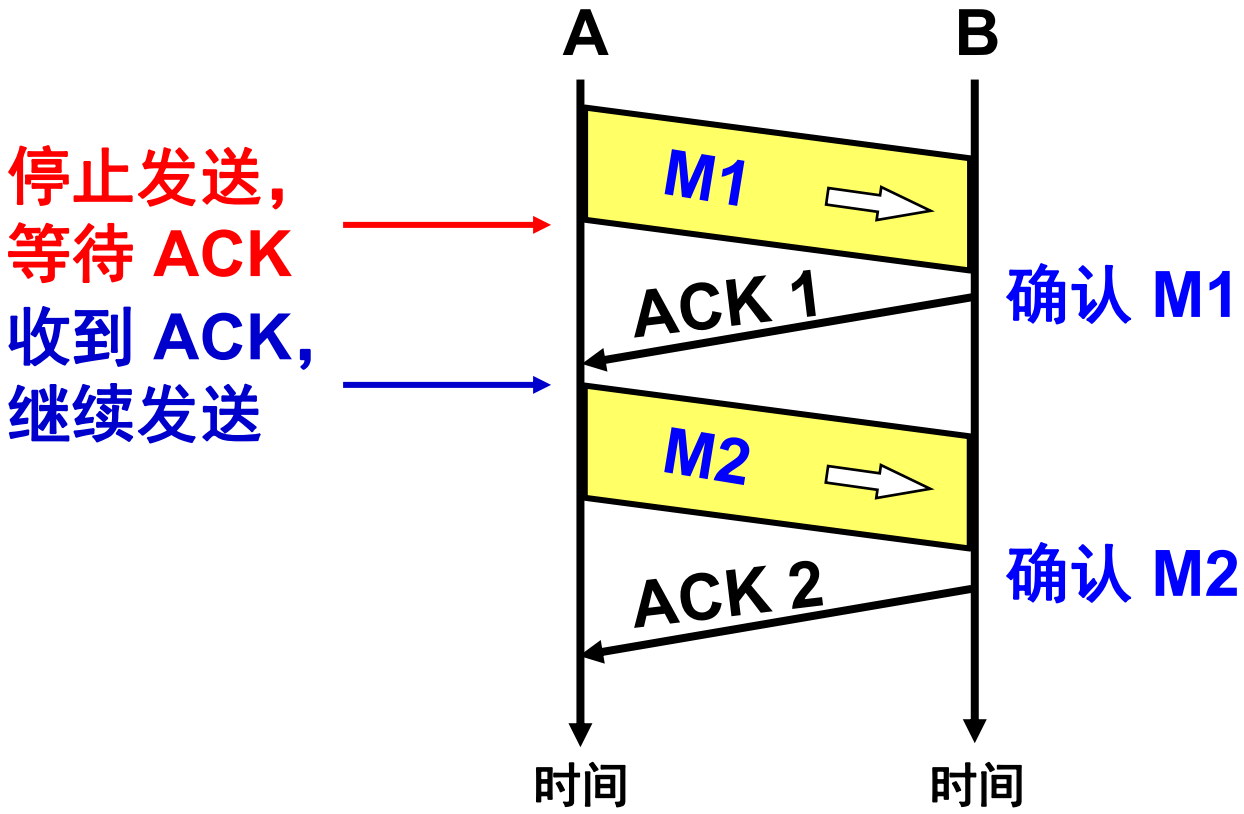
5.4 可靠传输的工作原理



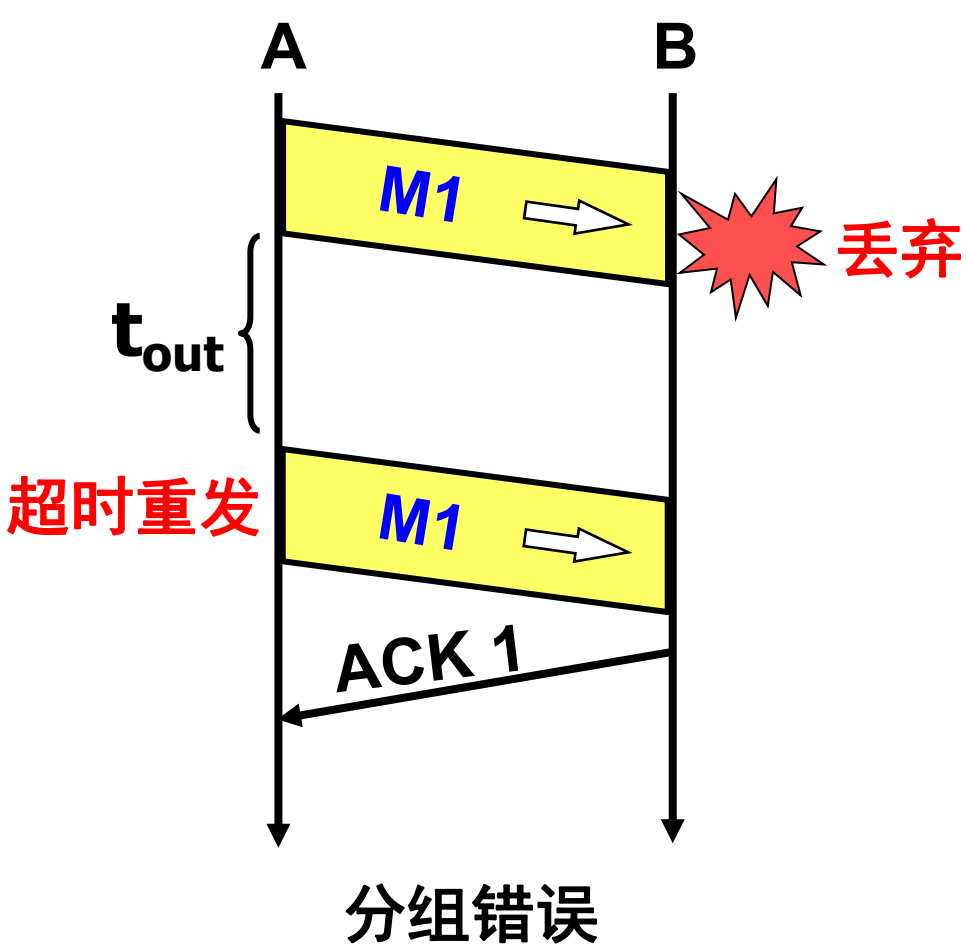
- 5.4.1 停止等待协议
- 5.4.2 连续 ARQ 协议

1. 无差错情况

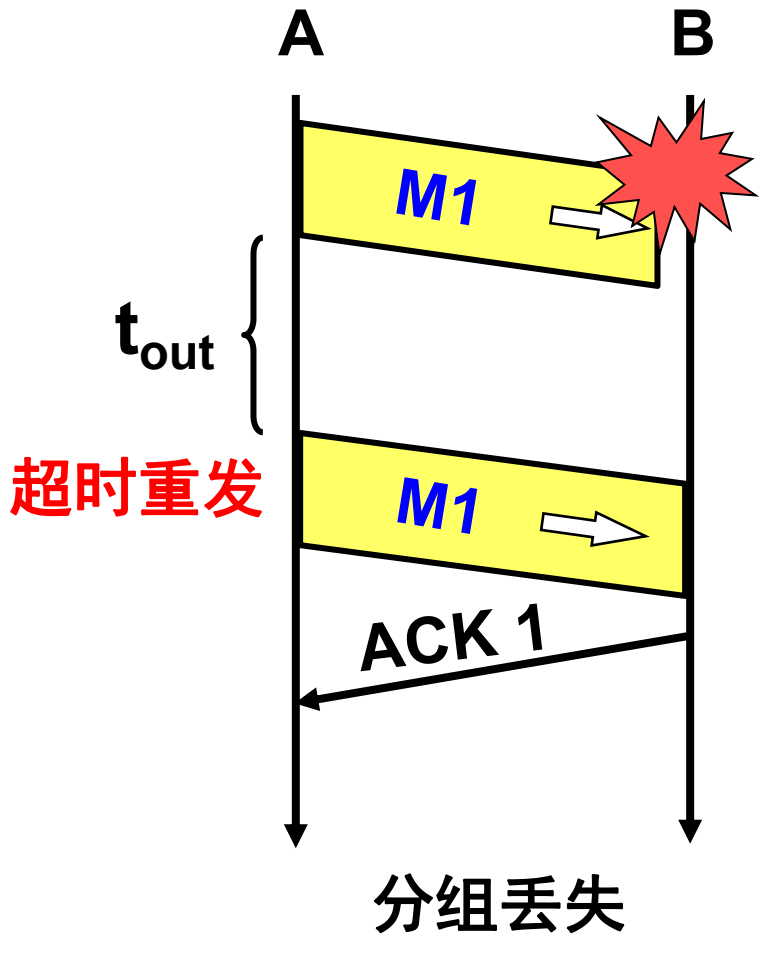
A 发送分组 M1，发完就暂停发送，等待 B 的确认 (ACK)。B 收到了 M1 向 A 发送 ACK。A 在收到了对 M1 的确认后，就再发送下一个分组 M2。



2. 出现差错

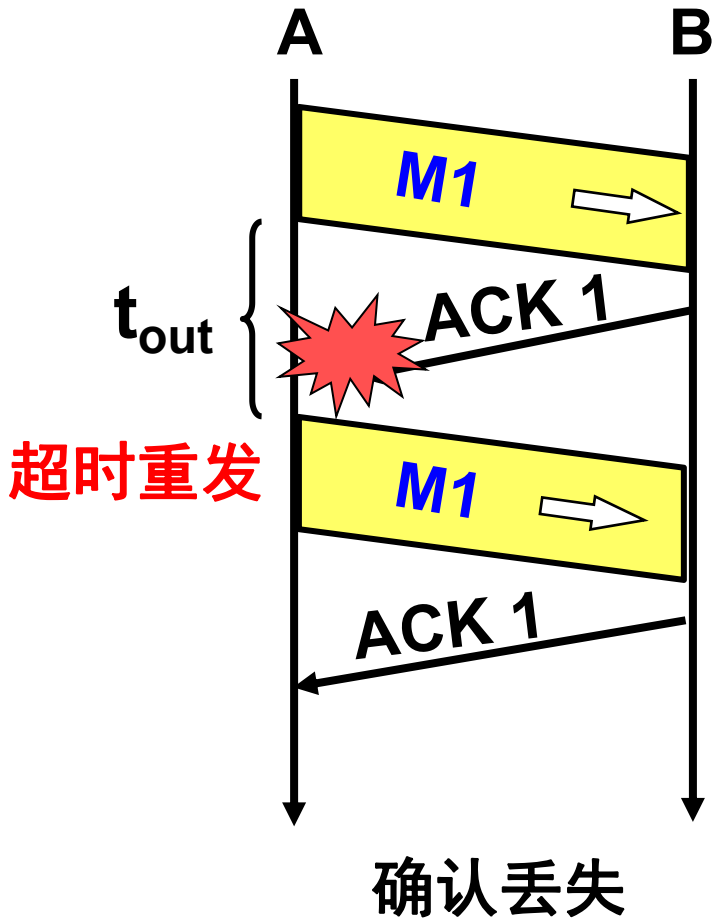


分组错误

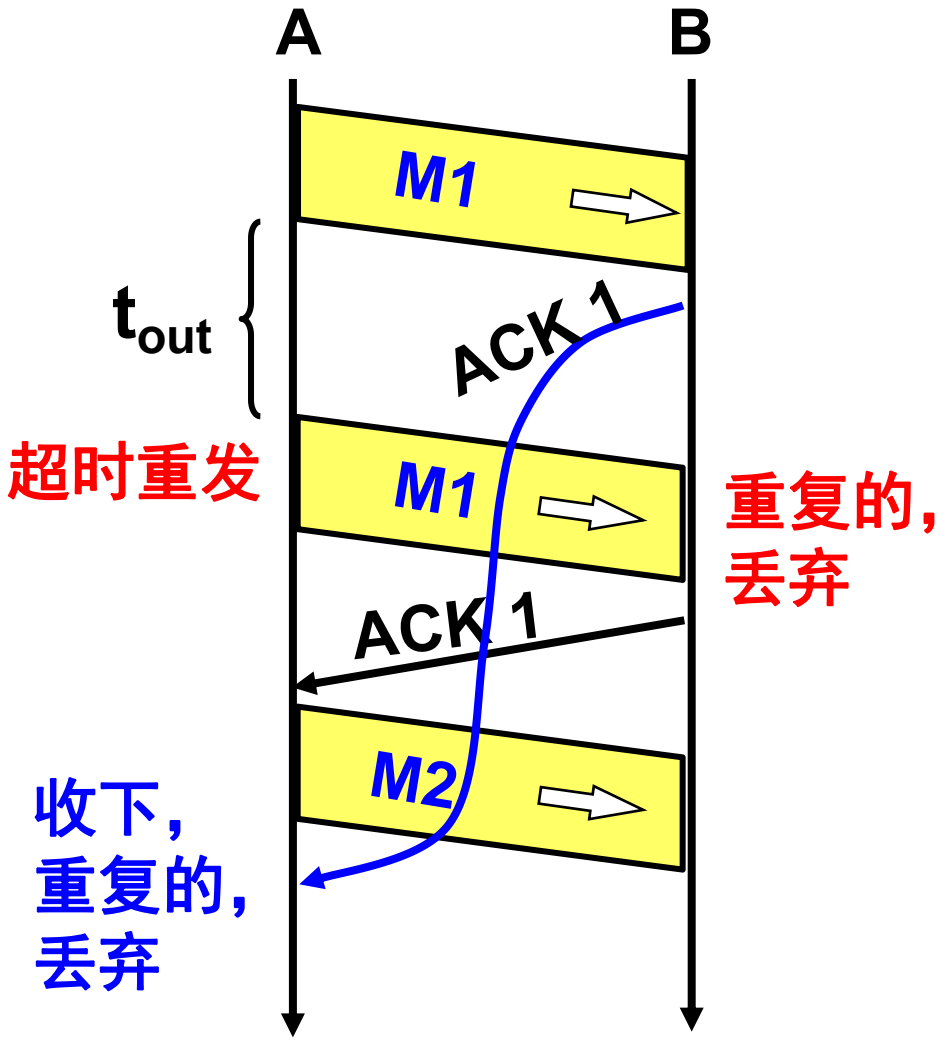


分组丢失

3. 确认丢失和确认迟到



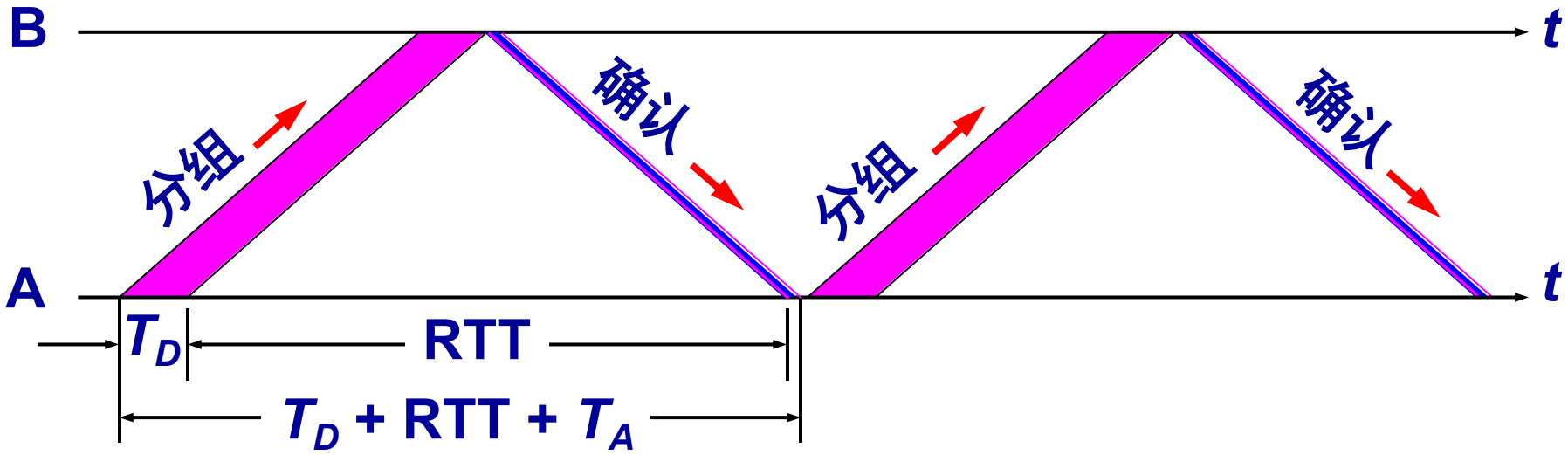
确认丢失



确认迟到

4. 信道利用率

停止等待协议的优点是简单，缺点是信道利用率太低。

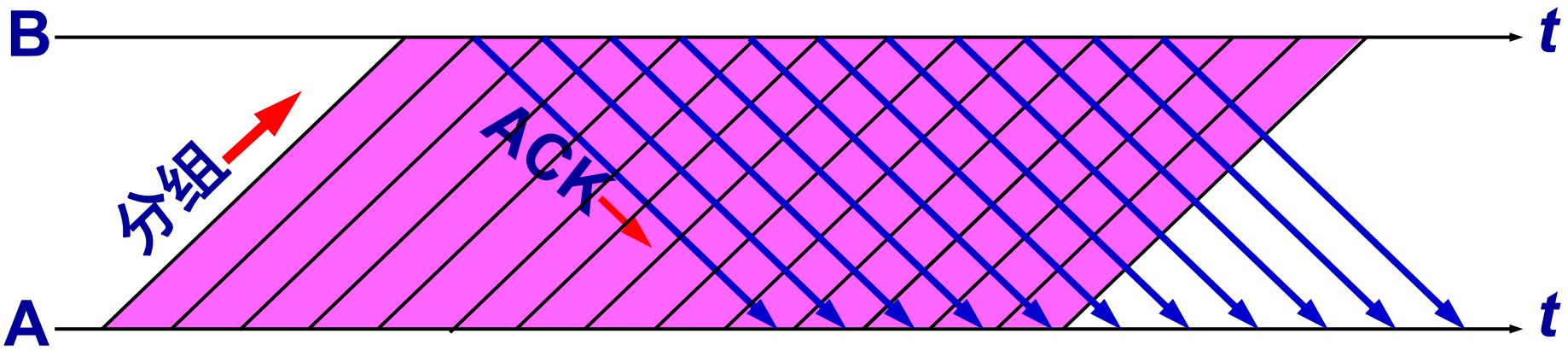


停止等待协议的信道利用率太低

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

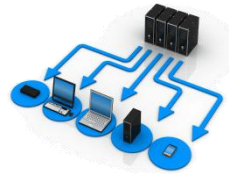
流水线传输

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。



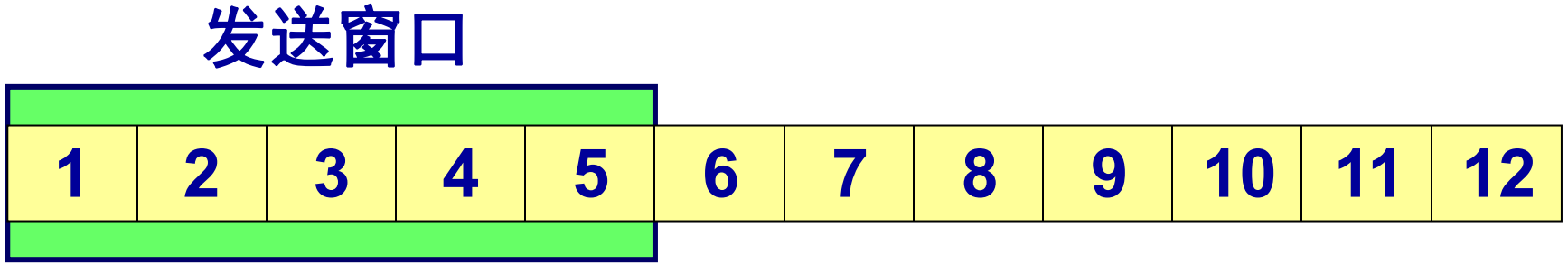
流水线传输可提高信道利用率

5.4.2 连续 ARQ 协议

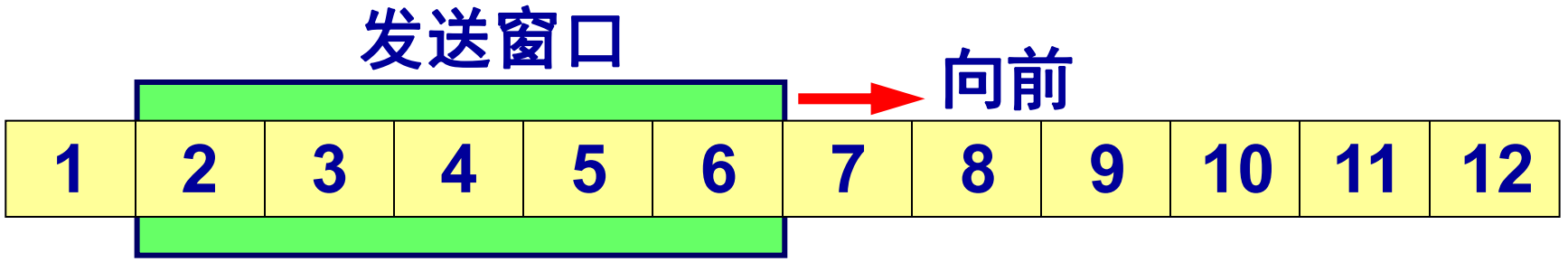


- 滑动窗口协议比较复杂，是 TCP 协议的精髓所在。
- 发送方维持的发送窗口，它的意义是：位于发送窗口内的分组都可连续发送出去，而不需要等待对方的确认。这样，信道利用率就提高了。
- 连续 ARQ 协议规定，发送方每收到一个确认，就把发送窗口向前滑动一个分组的位置。

5.4.2 连续ARQ协议



(a) 发送方维持发送窗口（发送窗口是 5）



(b) 收到一个确认后发送窗口向前滑动

连续 ARQ 协议的工作原理

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。

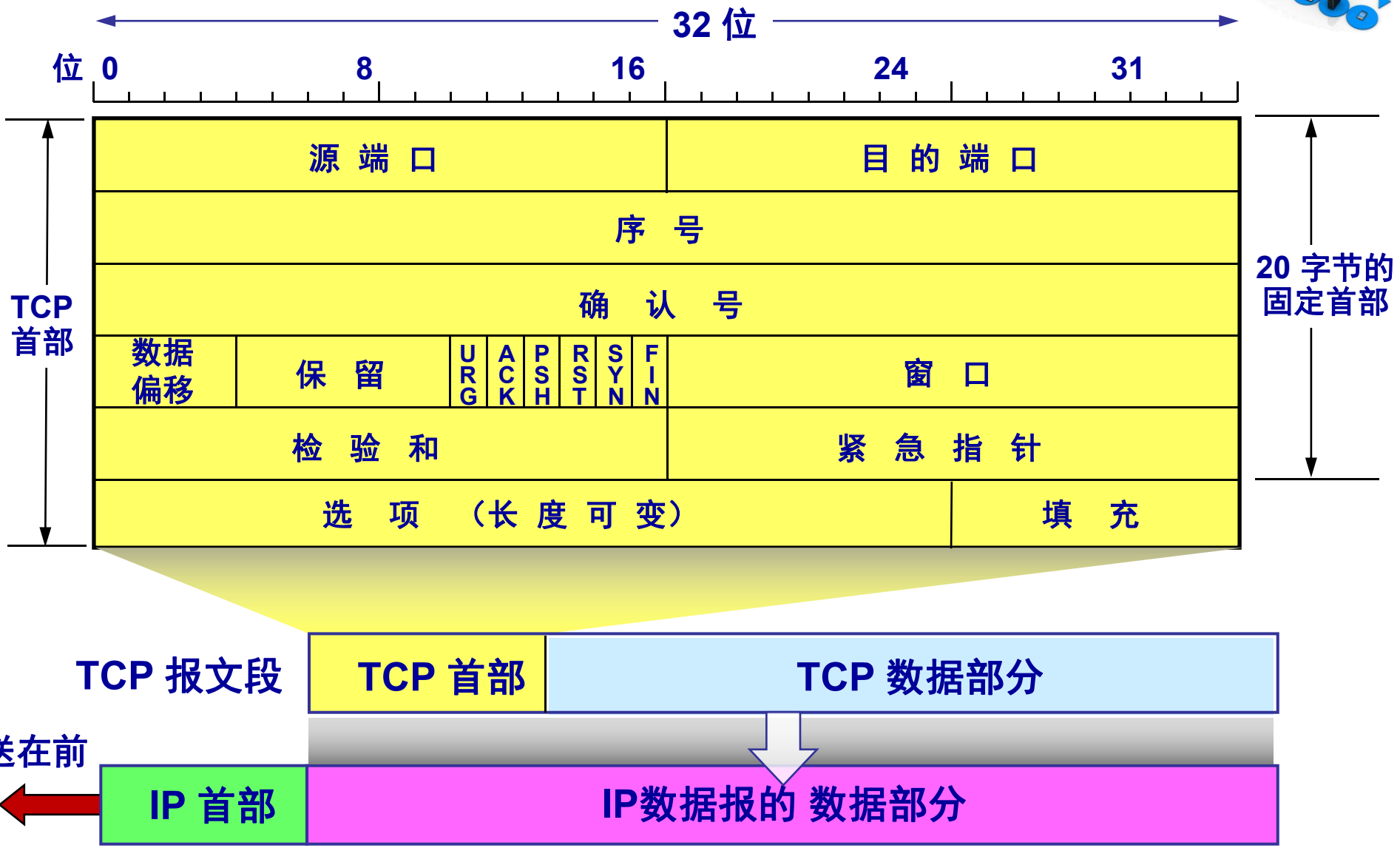
Go-back-N（回退 N）

- 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。

TCP 可靠通信的具体实现

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

TCP 报文段的首部格式



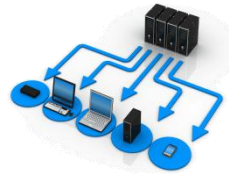
确认 ACK —— 只有当 ACK = 1 时确认号字段才有效。当 ACK = 0 时，确认号无效。

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。

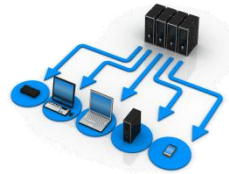
选项字段 —— 长度可变。TCP 最初只规定了一种选项，即最大报文段长度 MSS。MSS 告诉对方 TCP：“我的缓存所能接收的报文段的数据字段的最大长度是 MSS 个字节。”

5.6 TCP 可靠传输的实现



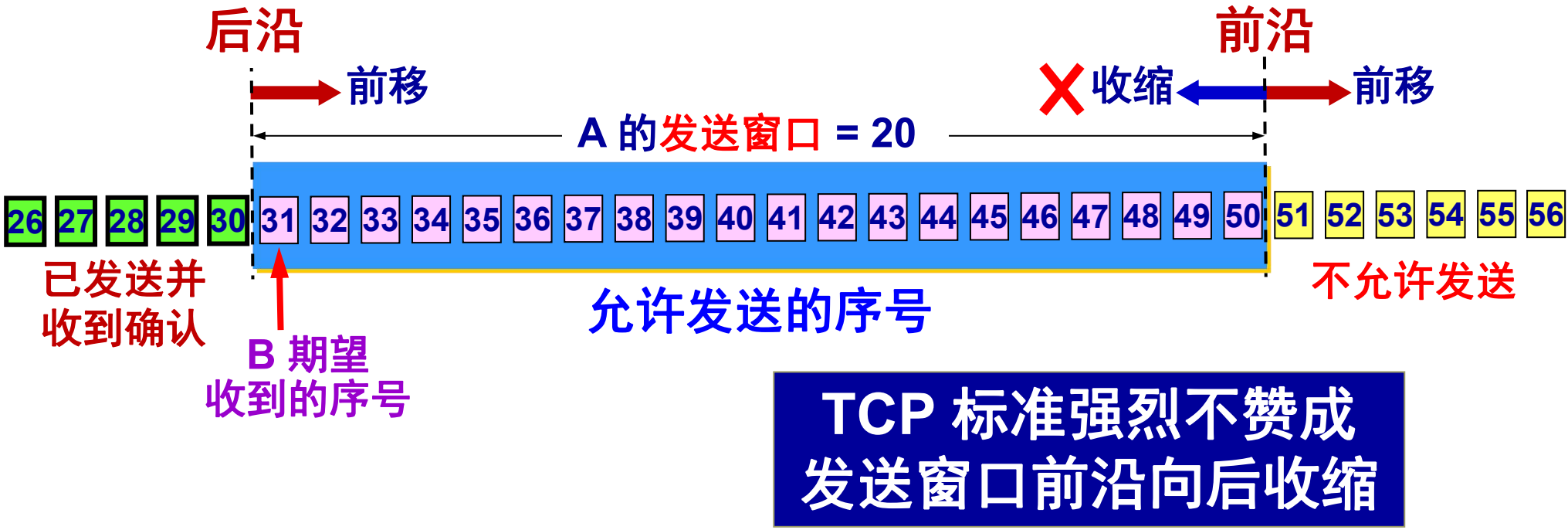
- 5.6.1 以字节为单位的滑动窗口
- 5.6.2 超时重传时间的选择
- 5.6.3 选择确认 SACK

5.6.1 以字节为单位的滑动窗口



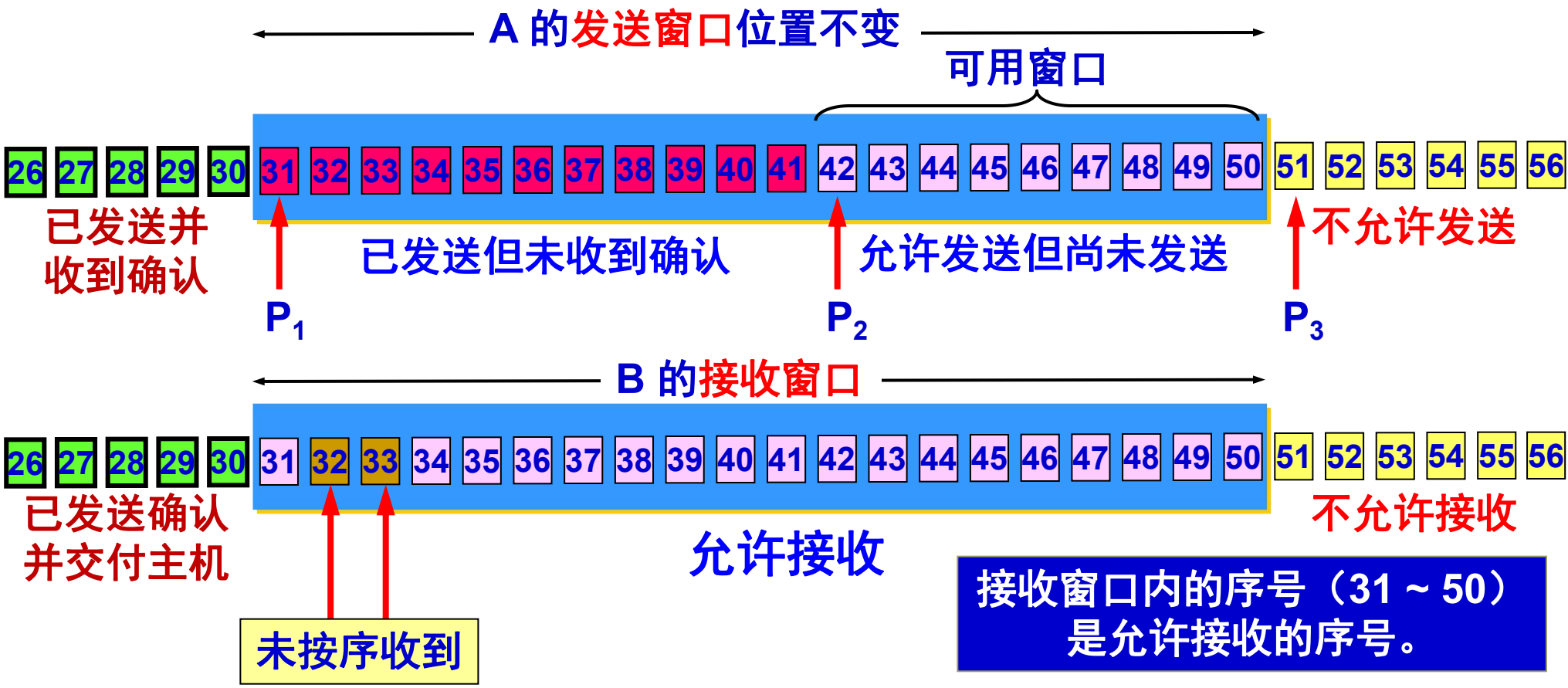
- TCP 的滑动窗口是以字节为单位的。
- 现假定 A 收到了 B 发来的确认报文段，其中窗口是 20 字节，而确认号是 31（这表明 B 期望收到的下一个序号是 31，而序号 30 为止的数据已经收到了）。
- 根据这两个数据，A 就构造出自己的发送窗口，

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



TCP 标准强烈不赞成发送窗口前沿向后收缩

A 发送了 11 个字节的数据



$P_3 - P_1$ = A 的发送窗口（又称为通知窗口）

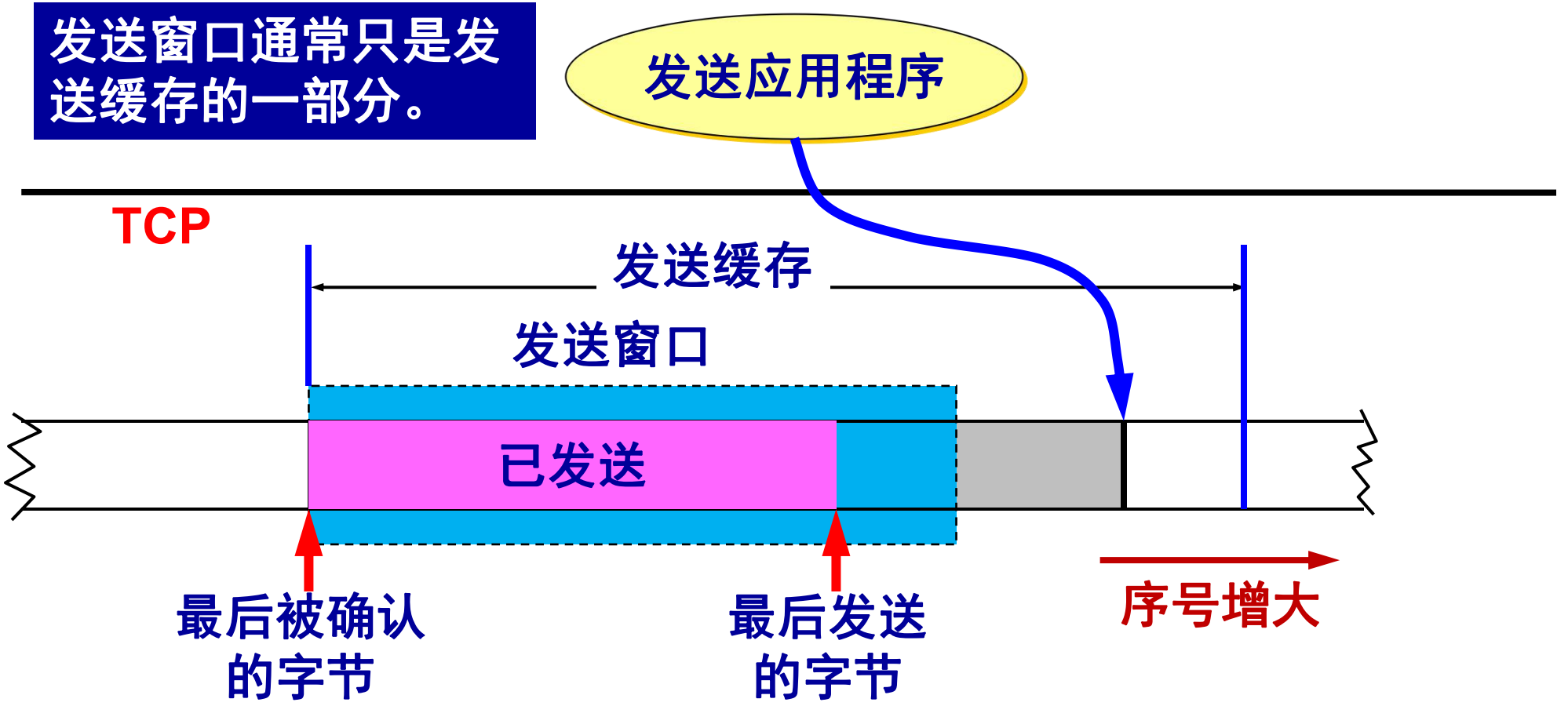
$P_2 - P_1$ = 已发送但尚未收到确认的字节数

$P_3 - P_2$ = 允许发送但尚未发送的字节数（又称为可用窗口）

发送缓存

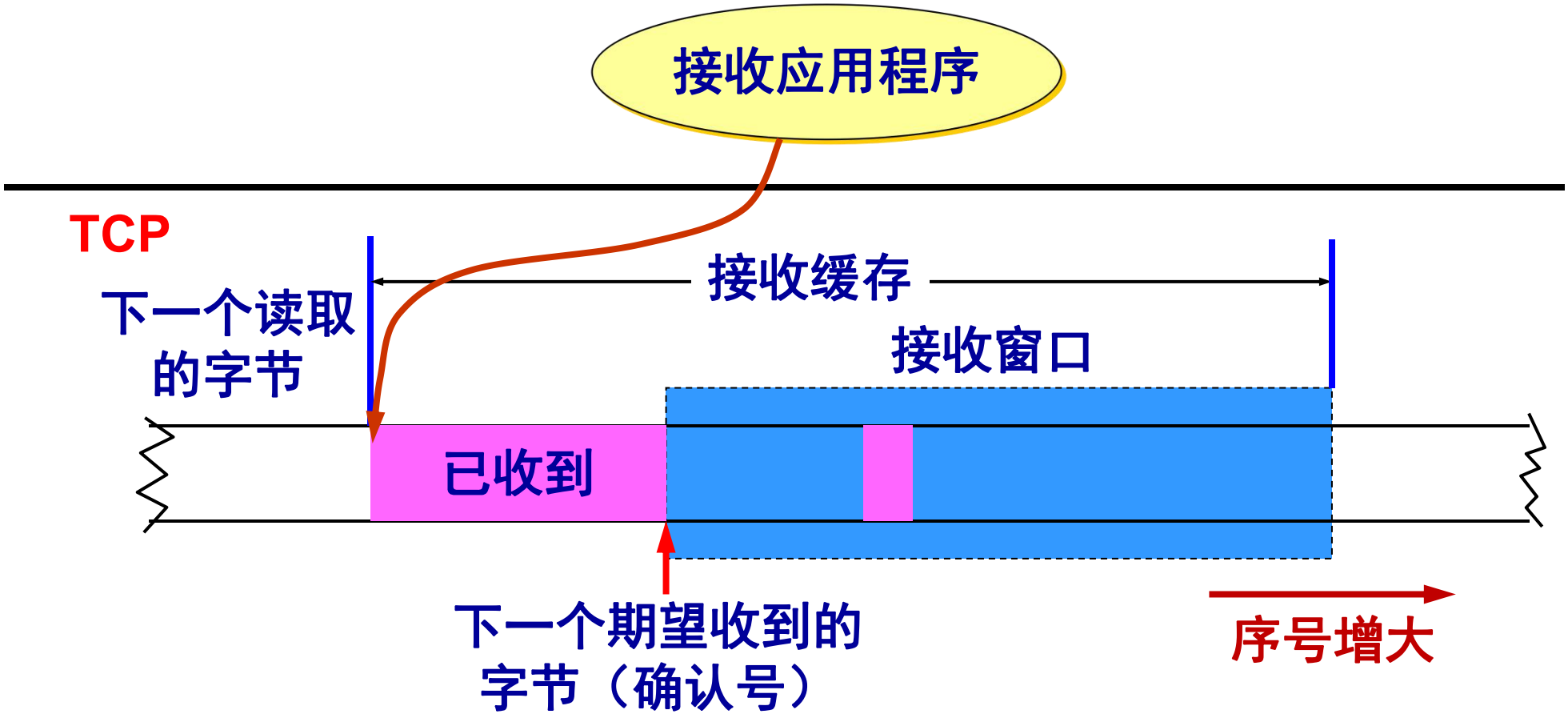
发送方的应用进程把字节流写入 TCP 的发送缓存。

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。



61. 在本题中列出的 8 种情况下，画出发送窗口的变化。并标明可用窗口的位置。已知主机 A 要向主机 B 发送 3 KB 的数据。在 TCP 连接建立后，A 的发送窗口大小是 2 KB。A 的初始序号是 0。

(1) 一开始 A 发送 1 KB的数据。

(2) 接着 A 就一直发送数据，直到把发送窗口用完。

(3) 发送方 A 收到对第 1000 号字节的确认报文段。

(4) 发送方 A 再发送 850 B的数据。

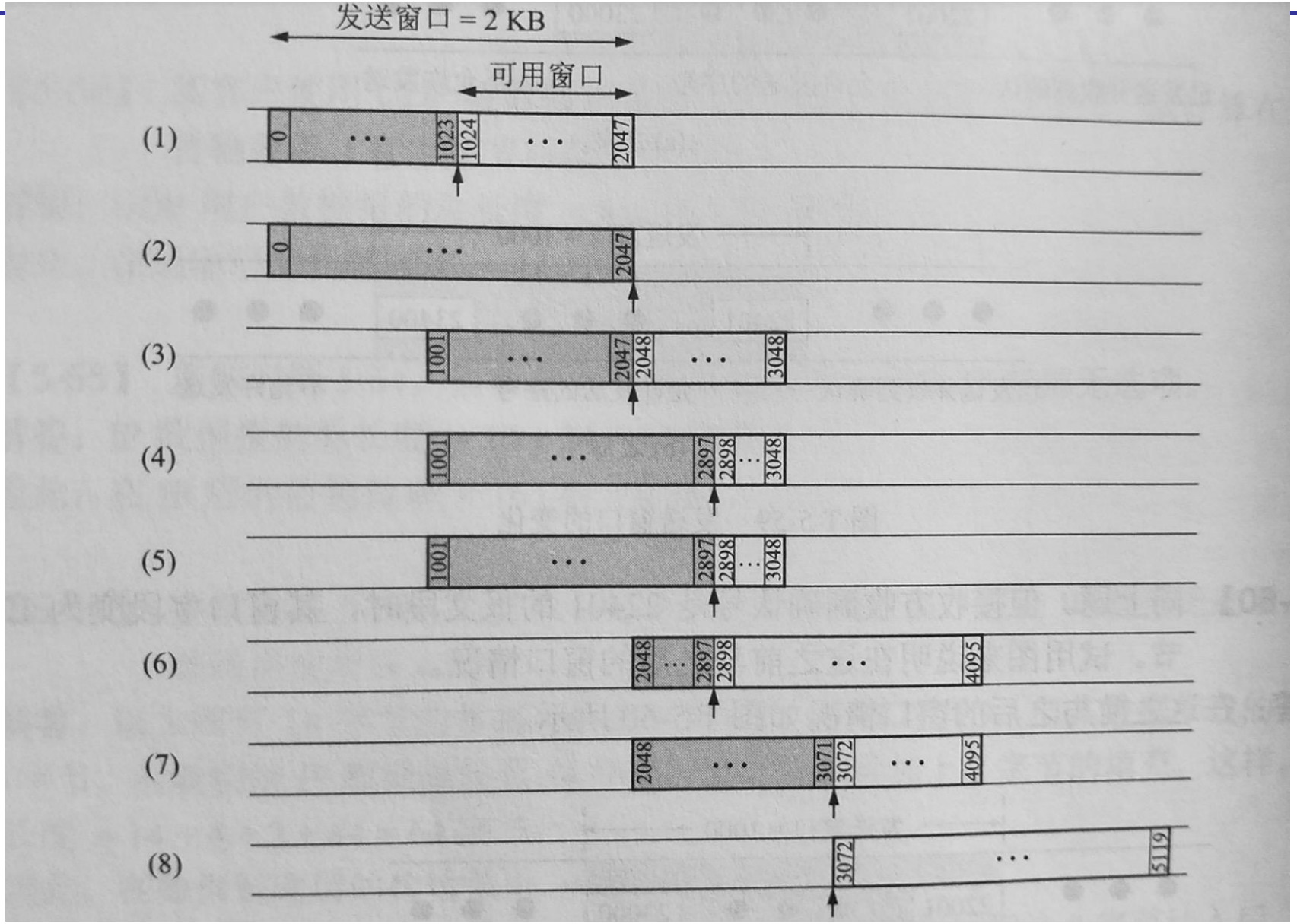
(5) 发送方 A 收到 ack = 900 的确认报文段。

(6) 发送方 A收到对第 2047 号字节的确认报文段。

(7) 发送方 A把剩下的数据全部都发送完。

(8) 发送方 A 收到 ack = 3072 的确认报文段。

下图是发送窗口和可用窗口的变化情况。


发送窗口 = 2 KB
可用窗口
(1)
0
1023
1024
2047
(2)
0
2047
(3)
1001
2047
2048
3048
(4)
1001
2897
2898
3048
(5)
1001
2897
2898
3048
(6)
2048
2897
2898
4095
(7)
2048
3071
3072
4095
(8)
3072
5119

（1）我们应当注意到，发送窗口 = 2 KB 就是 $2 * 1024 = 2048$ 字节。因此，发送窗口应当是从 0 到第 2047 字节为止，长度是 2048 字节。A 开始就发送了 1024 字节，因此发送窗口中左边的 1024 个字节已经用掉了（窗口的这部分为灰色），而可用窗口是白色的，从第 1024 字节到第 2047 字节位置。请注意，不是到第 2048 字节位置，因此第一个编号是 0 而不是 1。

（2）发送方 A 一直发送数据，直到把发送窗口用完。这时，整个窗口都已用掉了，可用窗口的大小已经是零了，一个字节也不能发送了。

（3）发送方 A 收到对第 1000 字节的确认报文段，表明 A 收到确认号 ack = 1001 的确认报文段。这时，发送窗口的后沿向前移动，发送窗口从第 1001 字节（不是从第 1000 字节）到第 3048 字节（不是第 3047 字节）为止。可用窗口从第 2048 字节到第 3048 字节。【注意，因为从 1001 起，3048 - 1001 + 1 = 2048】

（4）发送方 A 再发送 850 字节，使得可用窗口的后沿向前移动 850 字节，即移动到 2898 字节。现在的可用窗口从第 2898 字节到 3048 字节。

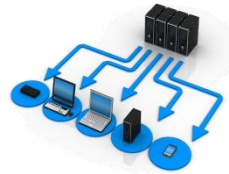
（5）发送方 A 收到 ack = 900 的确认报文段，不会对其窗口状态有任何影响。这是个迟到的确认。

（6）发送方 A 收到对第 2047 号字节的确认报文段。A 的发送窗口再向前移动。现在的发送窗口从第 2048 字节开始到第 4095 字节。可用窗口增大了，从第 2898 字节第 4095 字节。

（7）发送方 A 把剩下的数据全部发送完。发送方 A 共有 3 KB（即 3072 字节）的数据，其编号从 0 到 3071。因此现在的可用窗口变小了，从第 3072 字节到第 4095 字节。

（8）发送方 A 收到 ack = 3072 的确认报文段，表明序号在 3071 和这以前的报文段都收到了，后面期望收到的报文段的序号熊 3072 开始。因此新的发送窗口的位置又向前移动，从第 3072 号字节到第 5119 号字节。整个发送窗口也就是可用窗口。

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。

TCP 超时重传时间设置

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

加权平均往返时间

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

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

- 式中，$0 \le \alpha < 1$。若 α 很接近于零，表示 RTT 值更新较慢。若选择 α 接近于 1，则表示 RTT 值更新较快。
- RFC 2988 推荐的 α 值为 1/8，即 0.125。

1. 已知，当前 TCP 连接的 RTT 值为 35ms，连续收到 3 个确认报文段，它们比相应的数据报文段的发送时间滞后了 27ms，30ms 与 21ms。设α=0.2，计算第三个确认报文段到达后新的 RTT 估算值。

34. **已知第一次测得 TCP 的往返时延的当前值 RTT 是 30 ms。现在收到了三个接连的确认报文段，它们比相应的数据报文段的发送时间分别滞后的时间是：26 ms，32 ms 和24 ms。设 α = 0.1。试计算每一次的新的加权平均往返时间值 RTT_S。讨论所得出的结果。**

公式：新的$RTT_S = (1-\alpha) * (\text{旧的}RTT_s) + \alpha * (\text{新的}RTT\text{样本})$

第一次算出：$RTT_S = (1-0.1) * 30 + 0.1 * 26 = 29.6ms$

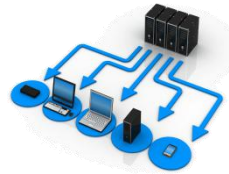
第二次算出：$RTT_S = (1-0.1) * 29.6 + 0.1 * 32 = 29.86ms$

第三次算出：$RTT_S = (1-0.1) * 29.86 + 0.1 * 24 = 29.256ms$

三次算出加权平均往返时间分别为 29.6，29.84 和 29.256 ms。

可以看出，RTT 的样本值变化多达 20 % 时($\frac{(30-24)}{30} = \frac{6}{30} = \frac{1}{5} = 20\ \%$)，加权平均往返 RTT_S 的变化却很小。

5.7 TCP 的流量控制



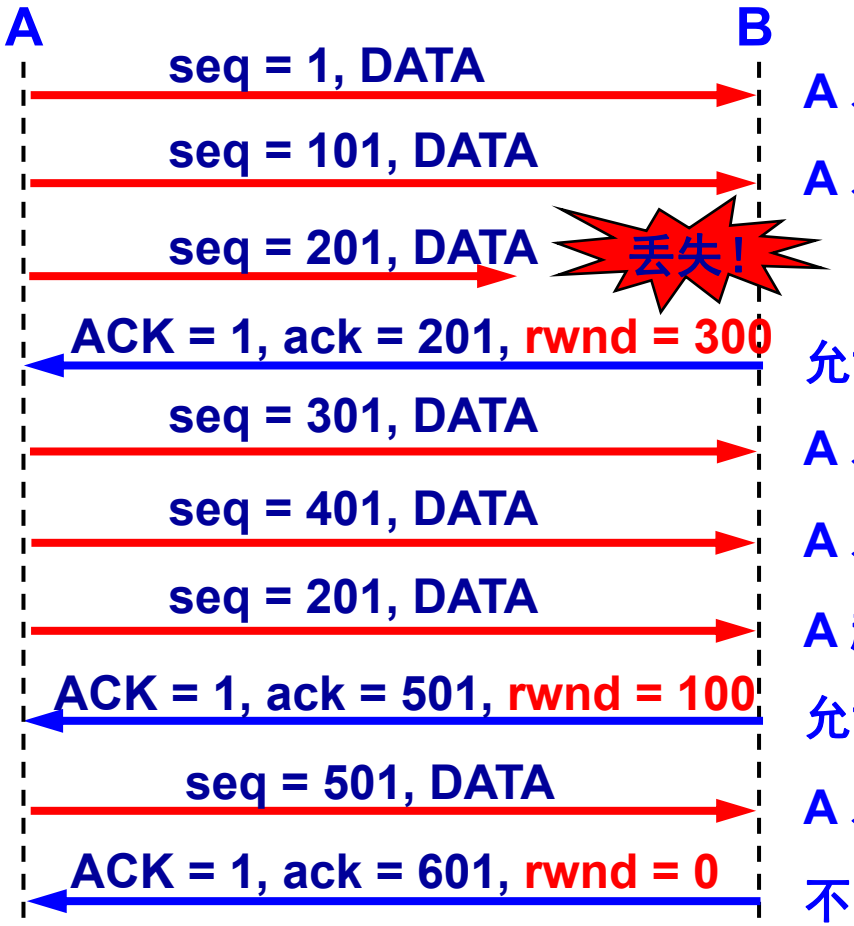
- 5.7.1 利用滑动窗口实现流量控制
- 5.7.2 TCP 的传输效率

5.7.1 利用滑动窗口实现流量控制

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

利用可变窗口进行流量控制举例

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



A 发送了序号 1 至 100，还能发送 300 字节

A 发送了序号 101 至 200，还能发送 200 字节

允许 A 发送序号 201 至 500 共 300 字节

A 发送了序号 301 至 400，还能再发送 100 字节新数据

A 发送了序号 401 至 500，不能再发送新数据了

A 超时重传旧的数据，但不能发送新的数据

允许 A 发送序号 501 至 600 共 100 字节

A 发送了序号 501 至 600，不能再发送了

不允许 A 再发送（到序号 600 为止的数据都收到了）

5.8 TCP 的拥塞控制

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

5.8.1 拥塞控制的一般原理

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

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

拥塞控制与流量控制的区别

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

拥塞控制与流量控制的区别

- **流量控制**往往指点对点通信量的控制，是个端到端的问题（接收端控制发送端）。
- **流量控制**所要做的就是抑制发送端发送数据的速率，以便使接收端来得及接收。

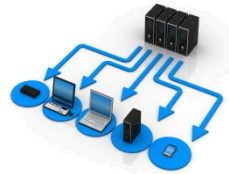
拥塞控制和流量控制之所以常常被弄混，是因为某些拥塞控制算法是向发送端发送控制报文，并告诉发送端，网络已出现麻烦，必须放慢发送速率。这点又和流量控制是很相似的。

5.8.2 TCP 的拥塞控制方法

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

真正的发送窗口值 = Min(公告窗口值，拥塞窗口值)

控制拥塞窗口的原则



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

拥塞的判断

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

TCP拥塞控制算法

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

慢开始 (Slow start)

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

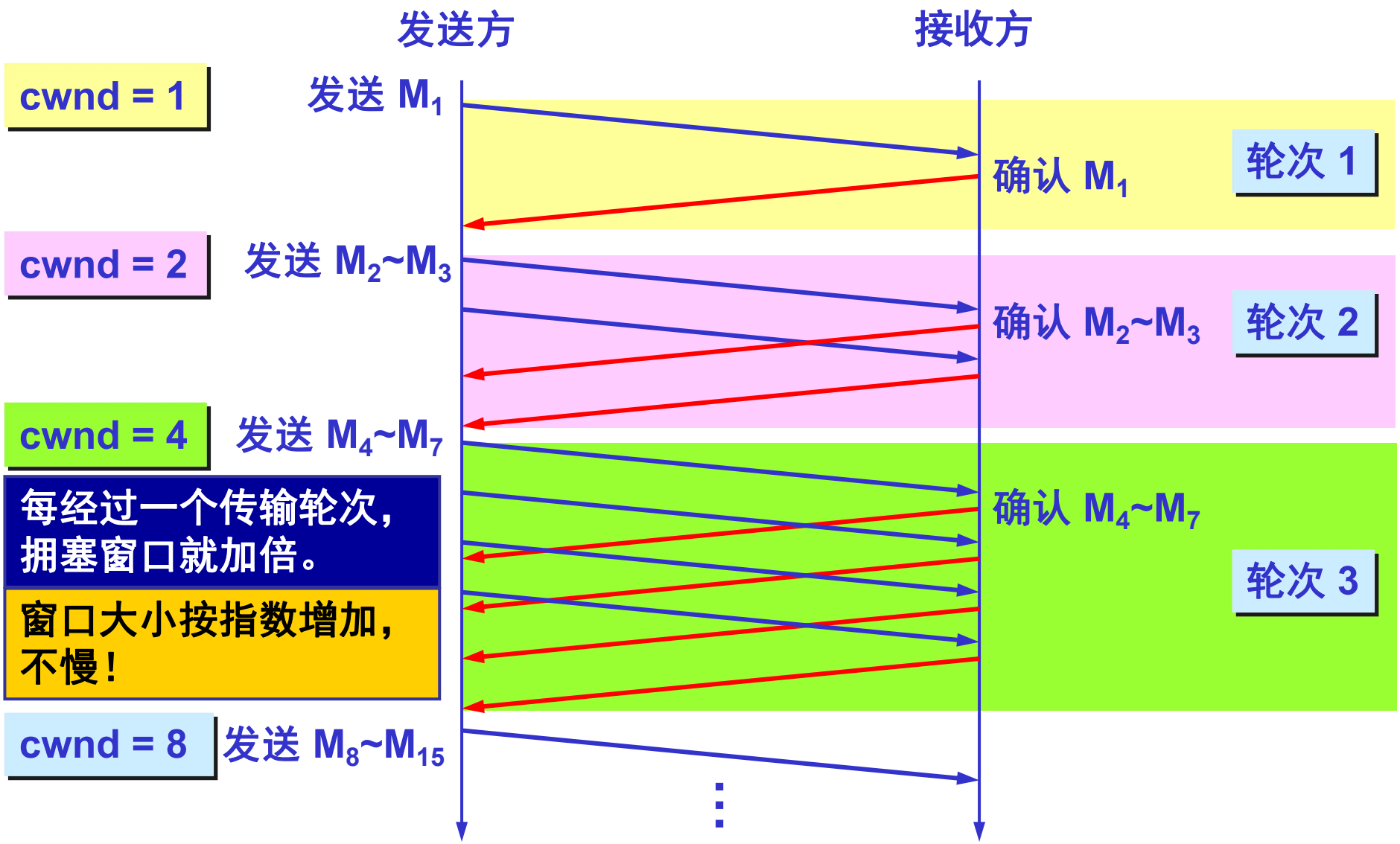
慢开始 (Slow start)

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

拥塞窗口cwnd每次的增加量 = min (N, SMSS)　　(5-8)

- 其中 *N* 是原先未被确认的、但现在被刚收到的确认报文段所确认的字节数。
- 不难看出，当 *N* < SMSS 时，拥塞窗口每次的增加量要小于 SMSS。
- 用这样的方法逐步增大发送方的拥塞窗口 cwnd，可以使分组注入到网络的速率更加合理。

发送方每收到一个对新报文段的确认（重传的不算在内）就使 cwnd 加 1。



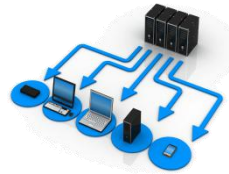
传输轮次

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

设置慢开始门限状态变量 ssthresh

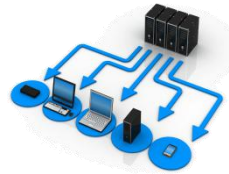
- 慢开始门限 ssthresh 的用法如下：
  - 当 cwnd < ssthresh 时，使用慢开始算法。
  - 当 cwnd > ssthresh 时，停止使用慢开始算法而改用拥塞避免算法。
  - 当 cwnd = ssthresh 时，既可使用慢开始算法，也可使用拥塞避免算法。

拥塞避免算法



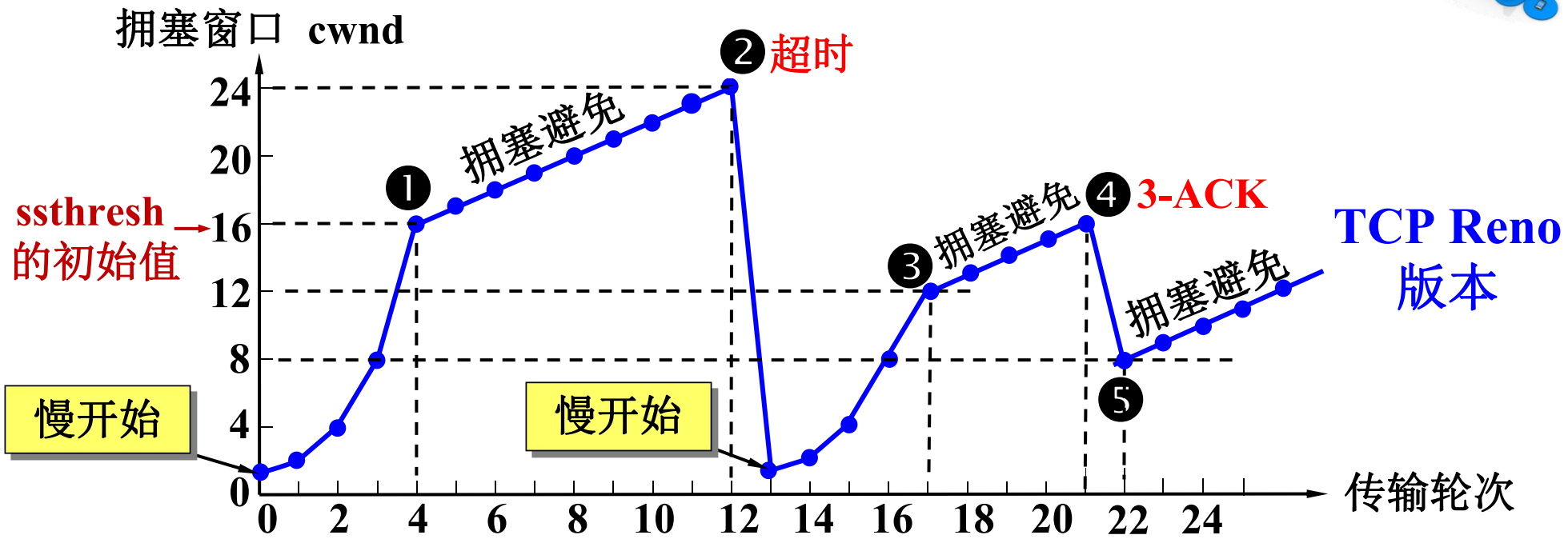
- **思路：**让拥塞窗口 cwnd **缓慢地增大，**即每经过一个往返时间 RTT 就把发送方的拥塞窗口 cwnd 加 1，而不是加倍，使拥塞窗口 cwnd **按线性规律缓慢增长。**
- 因此在拥塞避免阶段就有“**加法增大**”(Additive Increase) 的特点。这表明在拥塞避免阶段，拥塞窗口 cwnd 按线性规律缓慢增长，比慢开始算法的拥塞窗口增长速率缓慢得多。

当网络出现拥塞时



- 无论在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞（**重传定时器超时**）：
  - ssthresh = max(cwnd/2，2)
  - cwnd = 1
  - 执行慢开始算法
- 这样做的目的就是要迅速减少主机发送到网络中的分组数，使得发生拥塞的路由器有足够时间把队列中积压的分组处理完毕。

慢开始和拥塞避免算法的实现举例



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

慢开始门限的初始值设置为 16 个报文段，即 ssthresh = 16。

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。

快重传算法

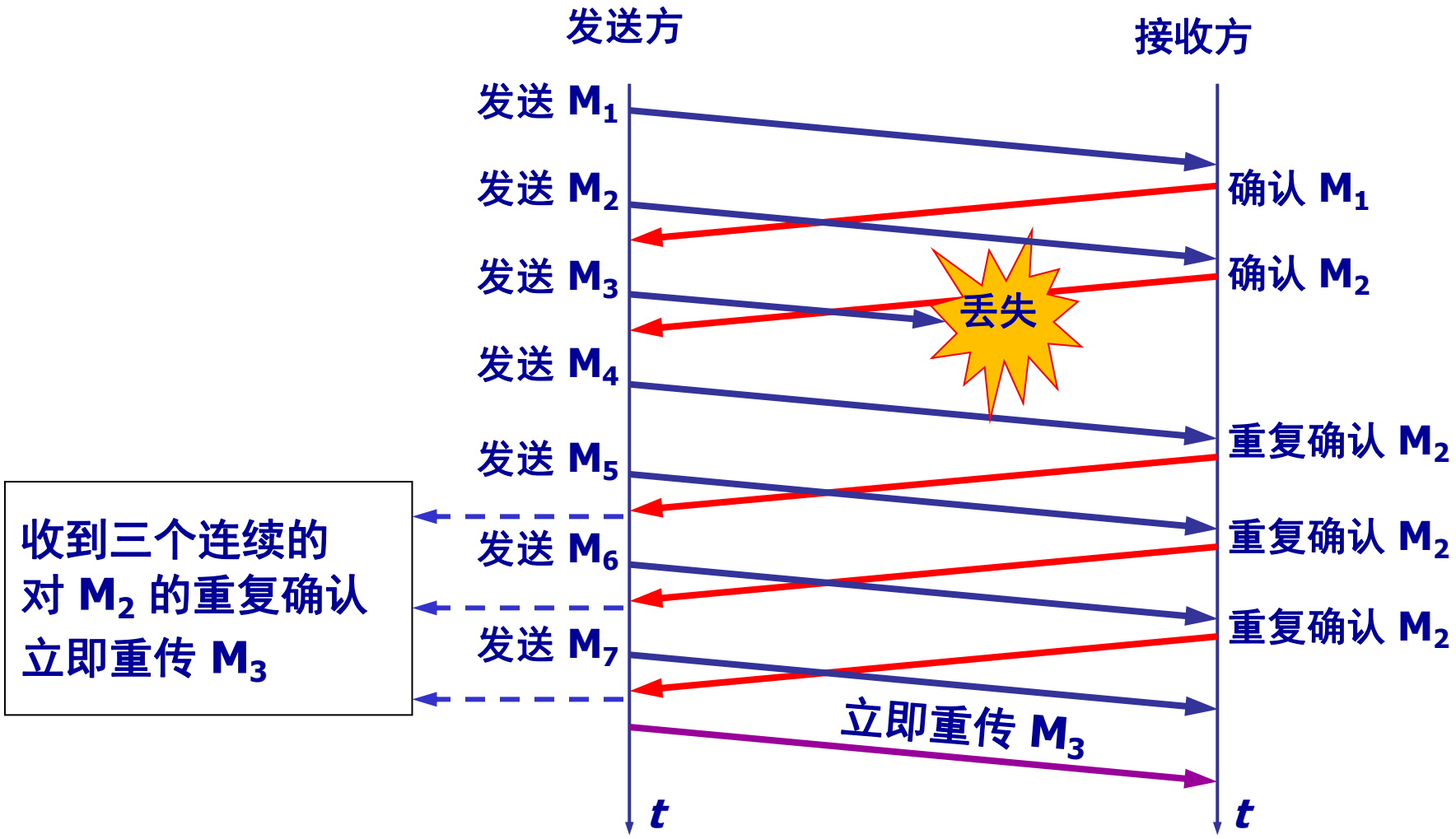
- 采用快重传FR (Fast Retransmission) 算法可以让发送方尽早知道发生了个别报文段的丢失。
- 快重传 算法首先要求接收方不要等待自己发送数据时才进行捎带确认，而是要立即发送确认，即使收到了失序的报文段也要立即发出对已收到的报文段的重复确认。

快重传算法

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

不难看出，快重传并非取消重传计时器，而是在某些情况下可更早地重传丢失的报文段。

快重传举例


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

快恢复算法

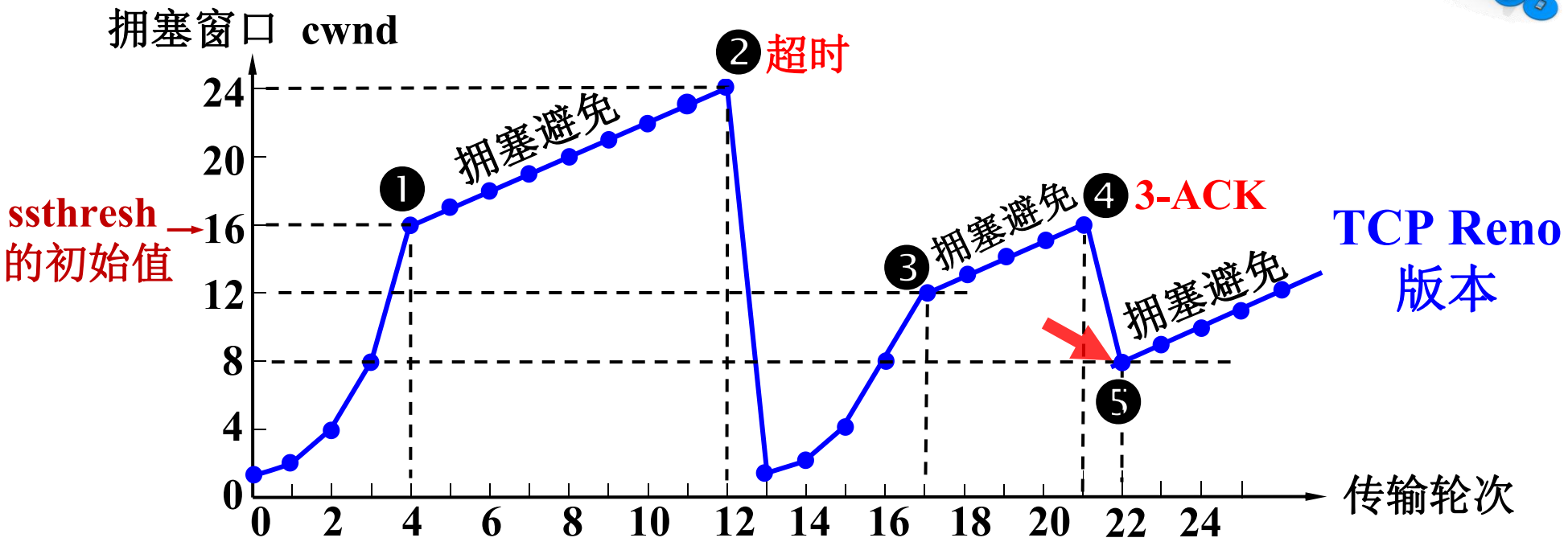
- 当发送端收到连续三个重复的确认时，由于发送方现在认为网络很可能没有发生拥塞，因此现在不执行慢开始算法，而是执行快恢复算法 FR (Fast Recovery) 算法：

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

  (2) 新拥塞窗口 cwnd = 慢开始门限 ssthresh ；

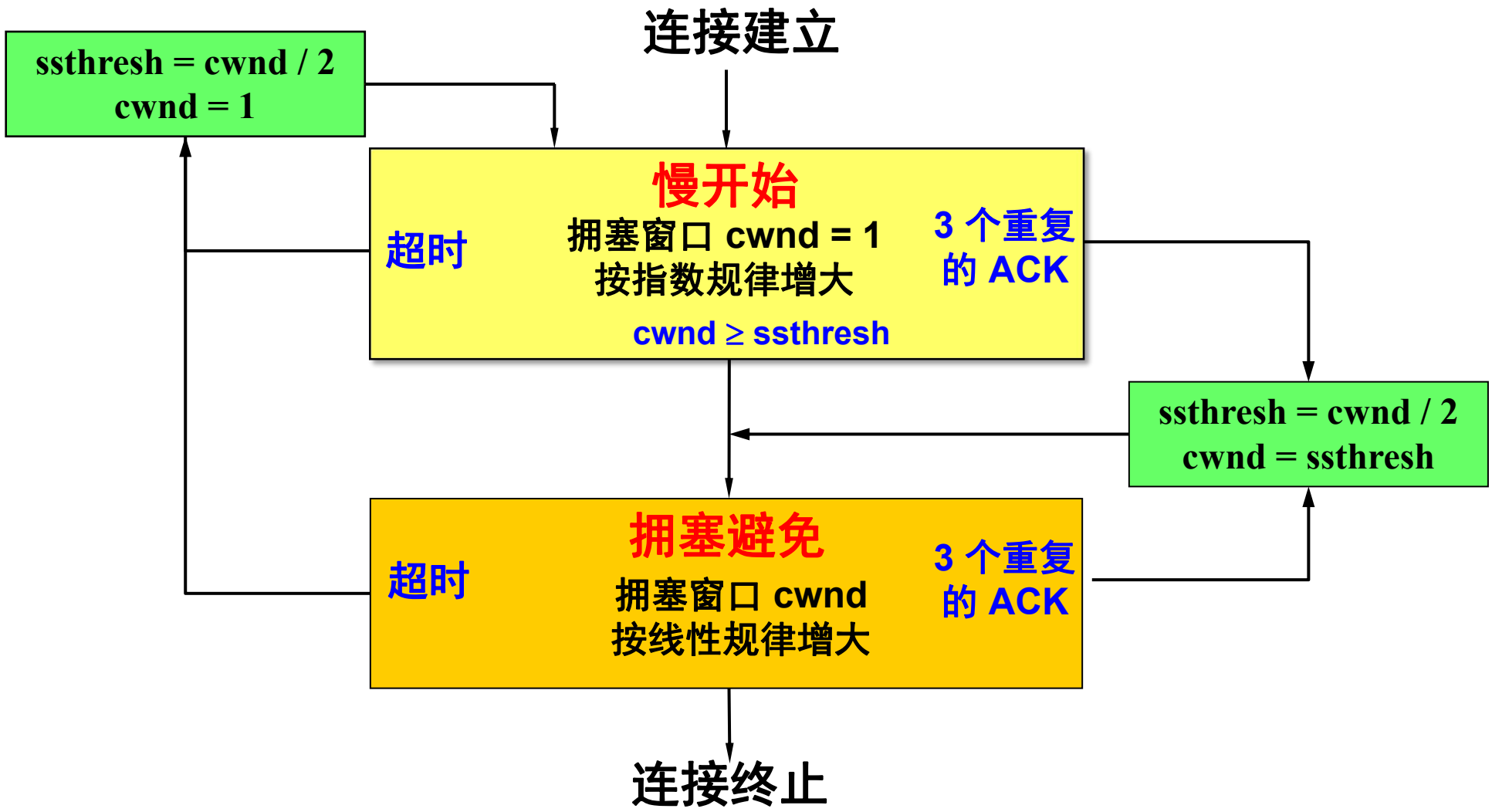
  (3) 开始执行拥塞避免算法，使拥塞窗口缓慢地线性增大。

慢开始和拥塞避免算法的实现举例



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

TCP拥塞控制流程图


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

发送窗口的上限值

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

发送窗口的上限值 = Min [rwnd, cwnd]　　(5-9)

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

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

39. **TCP 的拥塞窗口 cwnd 大小与传输轮次 n 的关系如表 5.1 所示：**

（1）试画出如教材中图 5-25 所示的拥塞窗口与传输轮次的关系曲线。

（2）指明 TCP 工作在慢开始阶段的时间间隔。

（3）指明 TCP 工作在拥塞避免阶段的时间间隔。

（4）在第 16 轮次和第 22 轮次之后发送方是通过收到三个重复的确认还是通过超时检测到丢失了报文段?

（5）在第 1 轮次，第 18 轮次和第 24 轮次发送时，门限 ssthresh 分别被设置为多大?

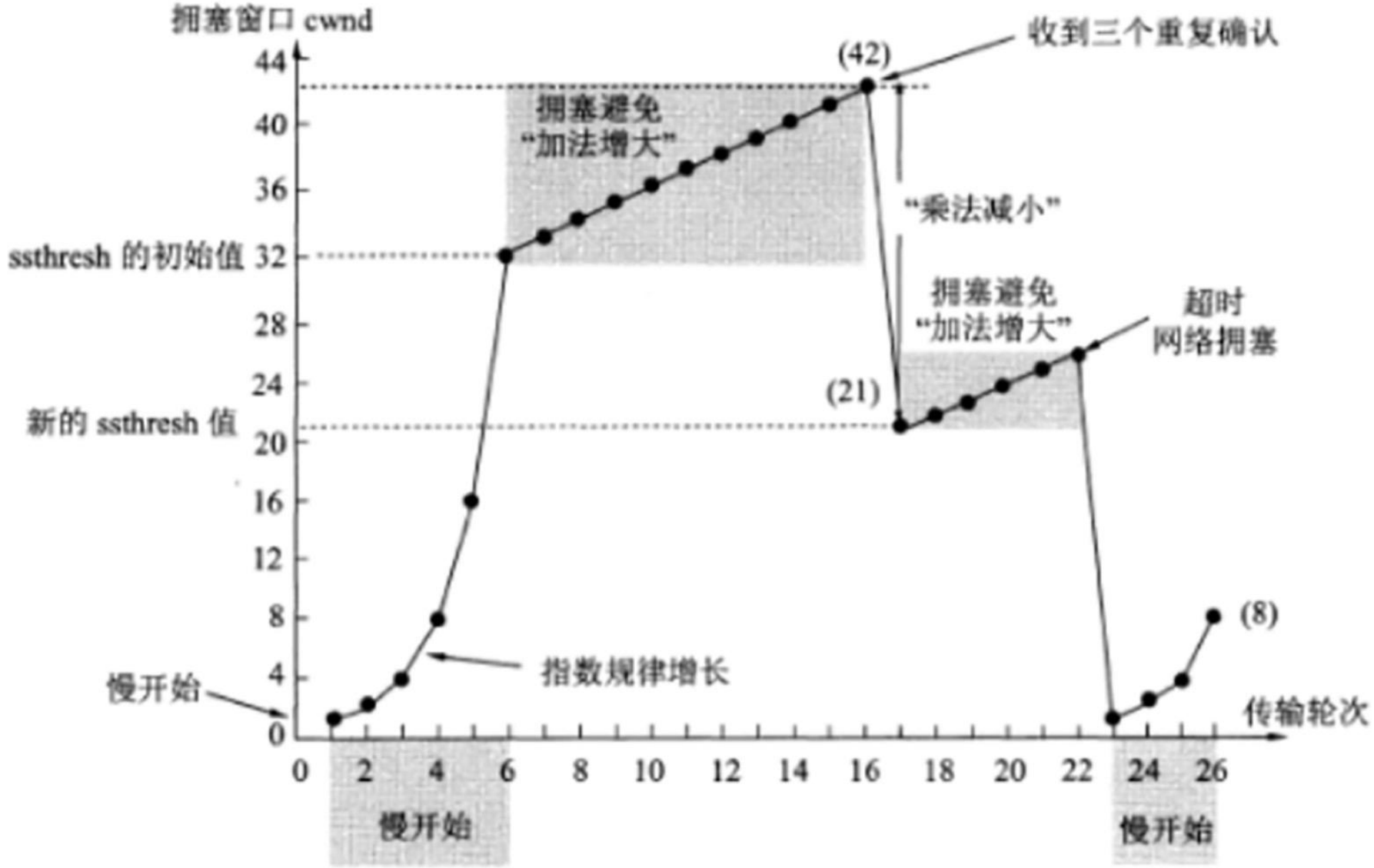
（6）在第几轮次发送出第 70 个报文段?

（7）假定在第 26 轮次之后收到了三个重复的确认，因而检测出了报文段的丢失，那么拥塞窗口 cwnd 和门限 ssthresh 应设置为多大?

拥塞窗口 cwnd 大小与传输轮次 n 的关系

n	1	2	3	4	5	6	7	8	9	10	11	12	13
cwnd	1	2	4	8	16	32	33	34	35	36	37	38	39
n	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**
cwnd	40	41	42	21	22	23	24	25	26	1	2	3	4

（1）拥塞窗口与传输轮次的关系曲线如下图所示：



（2）慢开始时间间隔：[1, 6] 和 [23, 26]

（3）拥塞避免时间间隔：[6, 16] 和 [17, 22]

（4）在第 16 轮次之后发送方通过收到三个重复的确认，检测到丢失了报文段，因为题目给出，下一个轮次的拥塞窗口减半了。

在第 22 轮次之后发送方通过超时，检测到丢失了报文段，因为题目给出，下一个轮次的拥塞窗口下降到 1了。

（5）在第 1 轮次发送时，门限 ssthresh 被设置为 32，因为从第 6 轮次起就进入了拥塞避免状态，拥塞窗口每个轮次加 1。

在第 18 轮次发送时，门限 ssthresh 被设置为发生拥塞时拥塞窗口 42 的一半，即 21。

在第 24 轮次发送时，门限 ssthresh 被设置为发生拥塞时拥塞窗口 26 的一半，即 13。

（6）第 1 轮次发送报文段 1。（cwnd = 1）

第 2 轮次发送报文段 2， 3。（cwnd = 2）

第 3 轮次发送报文段 4 ~ 7。（cwnd = 4）

第 4 轮次发送报文段 8 ~ 15。（cwnd = 8）

第 5 轮次发送报文段 16 ~ 31。（cwnd = 16）

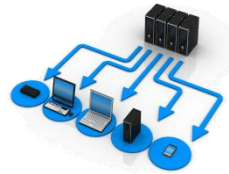
第 6 轮次发送报文段 32 ~ 63。（cwnd = 32）

第 7 轮次发送报文段 64 ~ 96。（cwnd = 33）

因此第 70 报文段在第 7 轮次发送出。

（7）检测出了报文段的丢失时拥塞窗口 cwnd 是 8，因此拥塞窗口 cwnd 的数值应当减半，等于 4，而门限 ssthresh 应设置为检测出报文段丢失时的拥塞窗口 8 的一半，即 4。

5.9 TCP 的运输连接管理

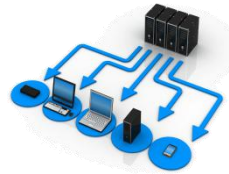


- 5.9.1 TCP 的连接建立
- 5.9.2 TCP 的连接释放
- 5.9.3 TCP 的有限状态机

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。

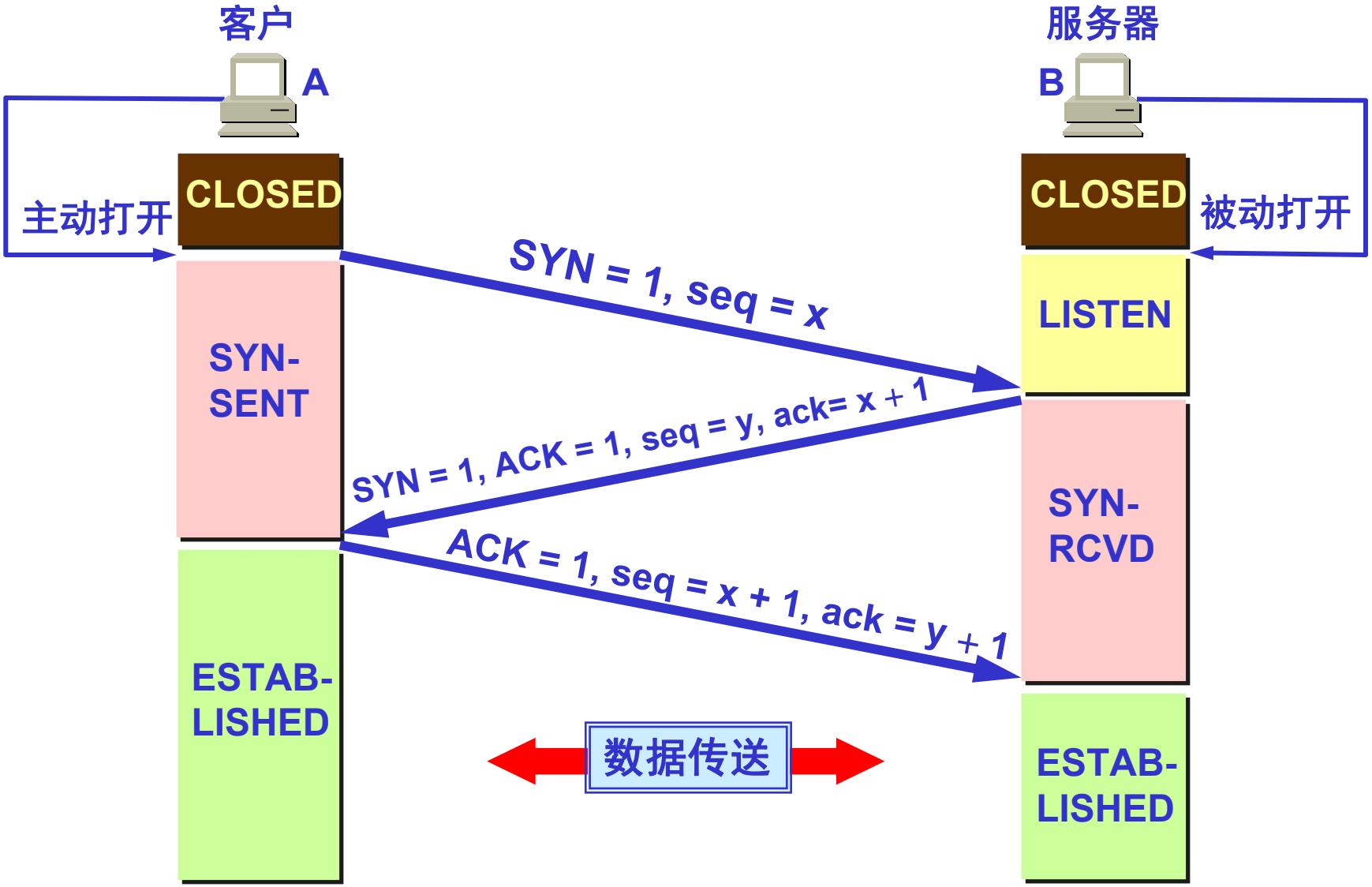
5.9.1 TCP 的连接建立



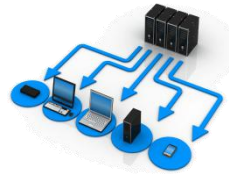
- TCP 建立连接的过程叫做握手。
- 握手需要在客户和服务器之间交换三个 TCP 报文段。称之为三报文握手。
- 采用三报文握手主要是为了防止已失效的连接请求报文段突然又传送到了，因而产生错误。

TCP 的连接建立：采用三报文握手

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。

5.9.2 TCP 的连接释放