计算机网络

第3章 运输层

目　　录

3.1 概述和运输层服务

- **运输层的功能**
  - 为不同主机上运行的应用进程之间提供*逻辑通信(logical communication)*
- **运输层协议的工作内容**
  - 发送方：把应用数据划分成 报文段(segments)，交给网络层
  - 接收方：把报文段重组成应用数据，交付给应用层

3.1 概述和运输层服务

- **运输层和网络层的区别**
  - *网络层:* 不同主机之间的逻辑通信
  - *运输层:* 应用进程之间的逻辑通信

类似于家庭间通信:

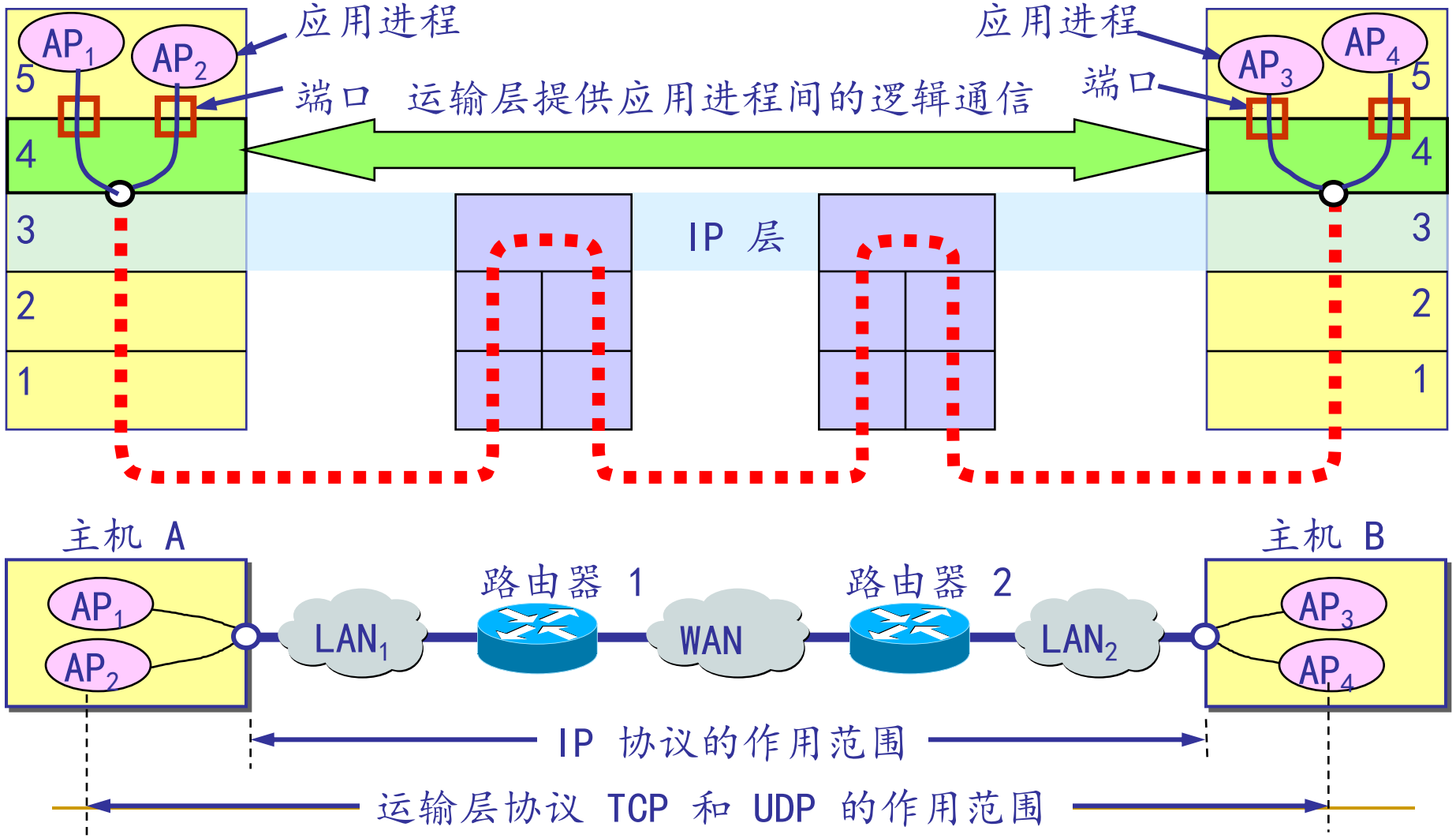
12个孩子要与另一个家庭的12个孩子相互通信

- 进程 = 孩子们
- 进程间报文 = 信封中的信笺
- 主机 = 家庭的房子
- 运输协议 = 张三 和 李四
- 网络层协议 = 邮局提供的服务

3.1 概述和运输层服务

- **上例中的几种特殊场景**
  - 张三和李四生病了，无法工作，换成张五和李六
    - 不同的运输层协议可能提供不一样的服务
  - 邮局不承诺信件送抵的最长时间
    - 运输层协议能够提供的服务受到底层网络协议的服务模型的限制
  - 邮局不承诺平信一定安全可靠的送达，可能在路上丢失，但张三、李四可在较长时间内没有受到对方的回信时，再次誊写信件，寄出
    - 在网络层不提供某些服务的情况下，运输层自己提供

运输层为相互通信的应用进程提供了逻辑通信



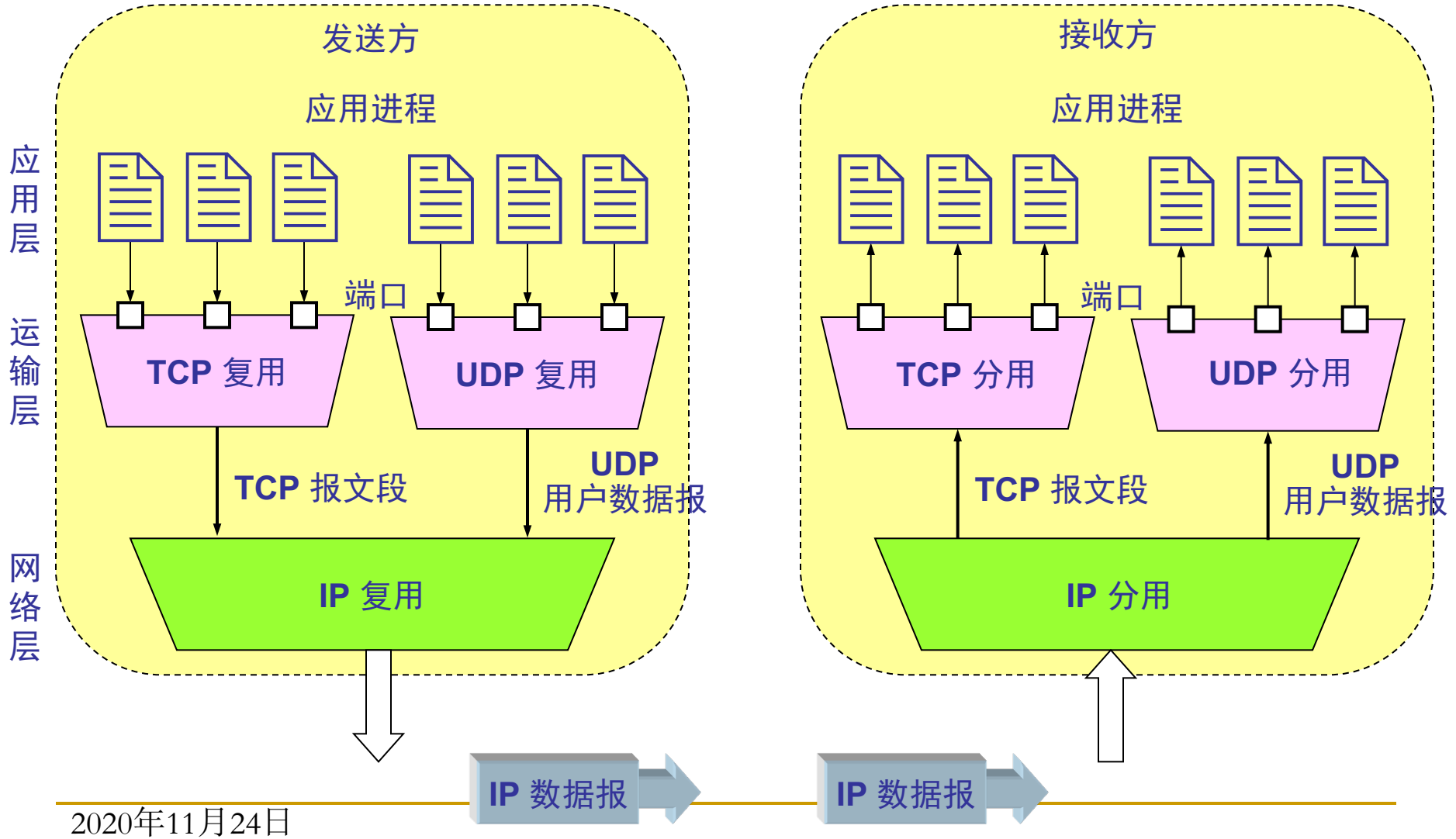
3.1 概述和运输层服务

- **因特网上的运输层协议**
  - 用户数据报协议UDP（*数据报*）
  - 传输控制协议TCP（*报文段*）
  - 所提供的服务
    - 进程间数据交付——详见3.2节
    - 差错检测——详见3.3节和第五章
    - 可靠的数据传输——详见3.4节和3.5节
    - 拥塞控制——详见3.6节和3.7节

3.2 多路复用与多路分解

- **TCP和UDP的最基本任务**
  - 将主机到主机之间数据的交付（由IP层提供）扩展为运行在二个主机上的进程间的数据交付。
  - 一台主机可以同时运行多个网络进程
    - 发送方：多个进程通过各自端口将数据交付给运输层，共同使用运输层的服务。这叫运输层的多路复用。
    - 接收方：当运输层收到从下层网络层传递上来的数据后，通过端口号就数据向上交付给各自的应用进程。这叫运输层的多路分解。

3.2 多路复用与多路分解



3.2 多路复用与多路分解

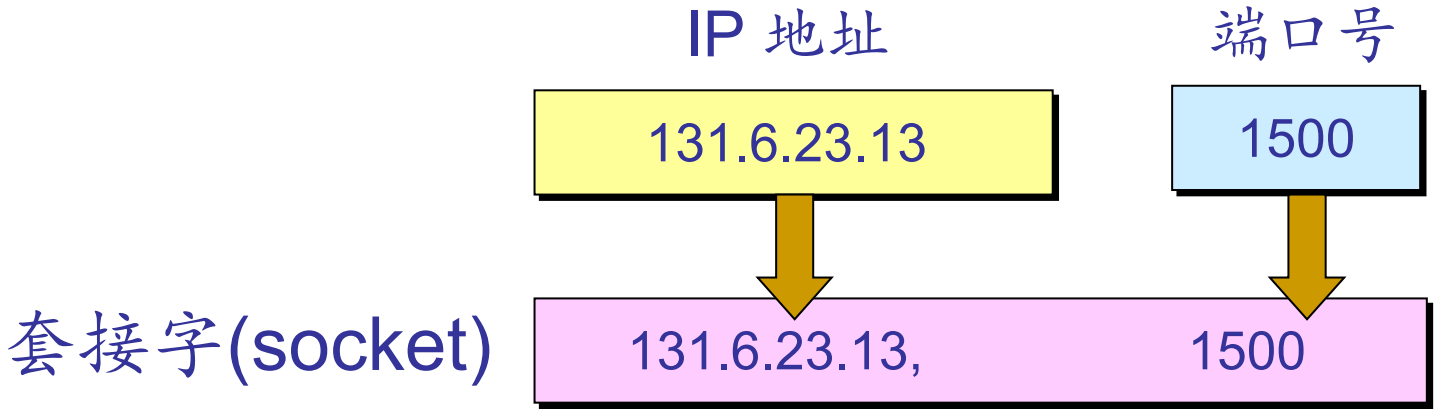
- **端口**
  - 端口的作用就是让应用层的各种应用进程都能将其数据通过端口向下交付给运输层，以及让运输层知道应当将其报文段中的数据向上通过端口交付给应用层相应的进程（或者线程）
  - 从这个意义上讲，端口是用来标志应用层的进程（或者线程）
  - 端口用一个 16 bit 端口号进行标志

3.2 多路复用与多路分解

- 端口
  - 一类是熟知端口，其数值一般为 0~1023。例如HTTP服务进程端口号80，FTP服务端口号20。
  - 另一类则是一般端口，用来随时分配给请求通信的客户进程。
  - 当我们开发一个新的网络应用程序时，应该为服务进程分配一个端口号，而客户端进程端口号一般由系统随机分配
  - 端口号16位，一共有65536个端口

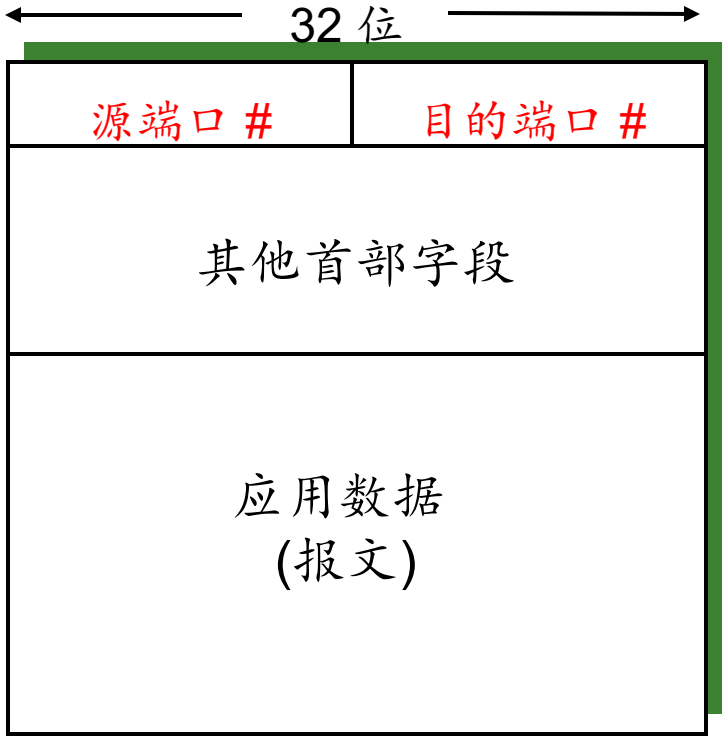
3.2 多路复用与多路分解

- **套接字**
  - TCP 使用“连接”(而不仅仅是“端口”)作为最基本的抽象，同时将 TCP 连接的端点称为套接字(socket) 。
  - 套接字和端口、IP 地址的关系是：



3.2 多路复用与多路分解

- **报文段（数据报）的投送**
  - 主机收到IP包
    - 每个数据包都有源IP地址和目的IP地址
    - 每个数据包都携带一个传输层的数据报文段
    - 每个数据报文段都有源、目的端口号
  - 主机根据“IP地址＋端口号”将报文段定向到相应的套接字



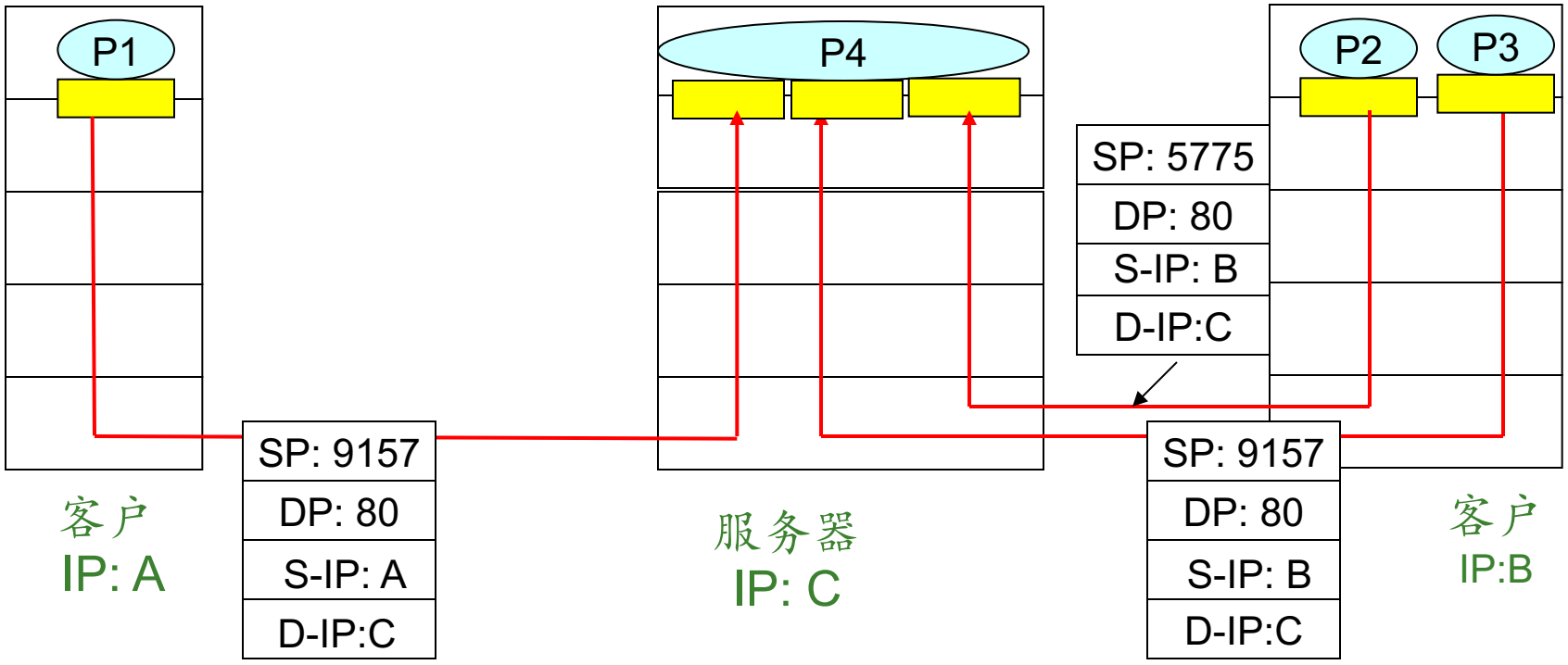
TCP/UDP 报文段格式

3.2 多路复用与多路分解

- **面向连接的复用和分用**
  - TCP 套接字由一个四元组来标识
    （源IP地址，源端口号，目的IP地址，目的端口号）
  - 接收方主机根据这四个值将报文段定向到相应的套接字
  - 服务器主机同时支持多个并发的TCP套接字:
    - 每一个套接字都由其四元组来标识
  - Web服务器为每一个客户连接都产生不同的套接字
    - 非持久HTTP对每一个请求都建立不同的套接字（会影响性能）

3.2 多路复用与多路分解

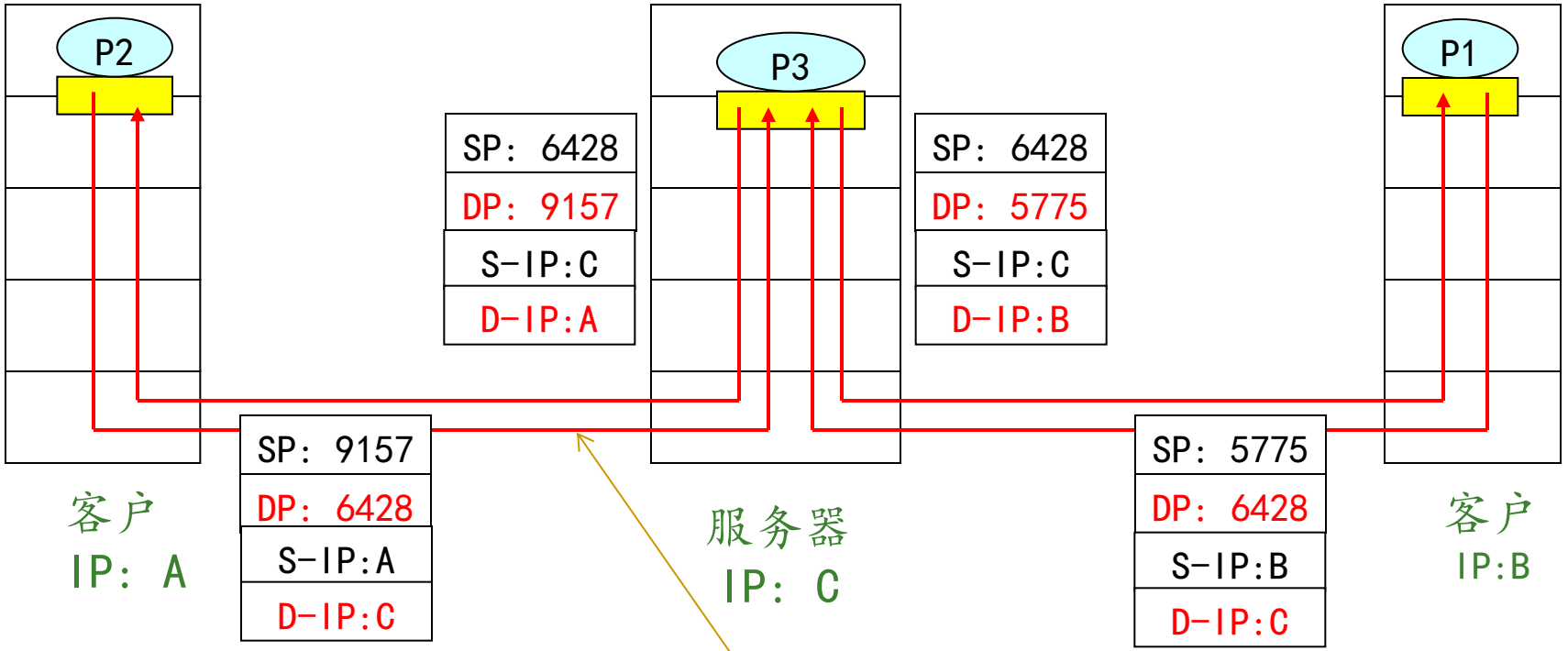
- **举例：多线程的WEB服务器**



3.2 多路复用与多路分解

- **无连接的复用和分用**
  - UDP 套接字由一个二元组来标识
    （目的IP地址，目的端口号）
  - 接收方根据目的端口号将报文段定向到相应的套接字
  - 具有不同源IP地址和/或源端口的UDP报文如果具有相同的目的IP地址和目的端口号，则定向到相同的套接字

无连接的复用与分用(源端口号的作用)



SP+S-IP 提供 “返回地址”

直接发送数据，不需要建立连接
因此图中的红色线代表数据，而不是连接

3.3 无连接传输：UDP

- **一个最简单的运输层协议必须提供**
  - 多路复用/多路分解服务
  - 差错检查（虽然进行差错检测，但不进行差错恢复。只是丢弃出错的UDP报文或交给应用程序但发出警告）
    - 请思考下为什么UDP提供的是无连接的不可靠的传输服务，却还要提供差错检测？
  - 几乎没有对IP增加什么东西。如果程序开发人员选择基于UDP的Socket，则应用程序几乎是直接与IP打交道。

实际上这就是UDP所提供的功能（RFC 768）

3.3 无连接传输：UDP

- **UDP处理数据的流程**
  - 发送方
    - 从应用进程得到数据
    - 附加上为多路复用/多路分解所需的源和目的端口号及差错检测信息，形成报文段（*数据报*）
    - 递交给网络层，尽力而为的交付给接收主机
  - 接收方
    - 从网络层接收报文段（*数据报*）
    - 根据目的端口号，将数据交付给相应的应用进程

UDP通信事先无需握手，是无连接的

3.3 无连接传输：UDP

- **UDP的优势**
  - 无需建立连接——建立连接会增加时延
  - 简单——发送方和接收方无需维护连接状态
  - 段首部开销小——TCP:20Byte vs UDP:8Byte
  - 无拥塞控制——UDP 可按需要随时发送

3.3 无连接传输：UDP

- **部分采用UDP协议的应用**
  - 远程文件服务器（NFS）
  - 流式多媒体
  - 因特网电话
  - 网络管理（SNMP）
  - 选路协议（RIP）
  - 域名解析（DNS）

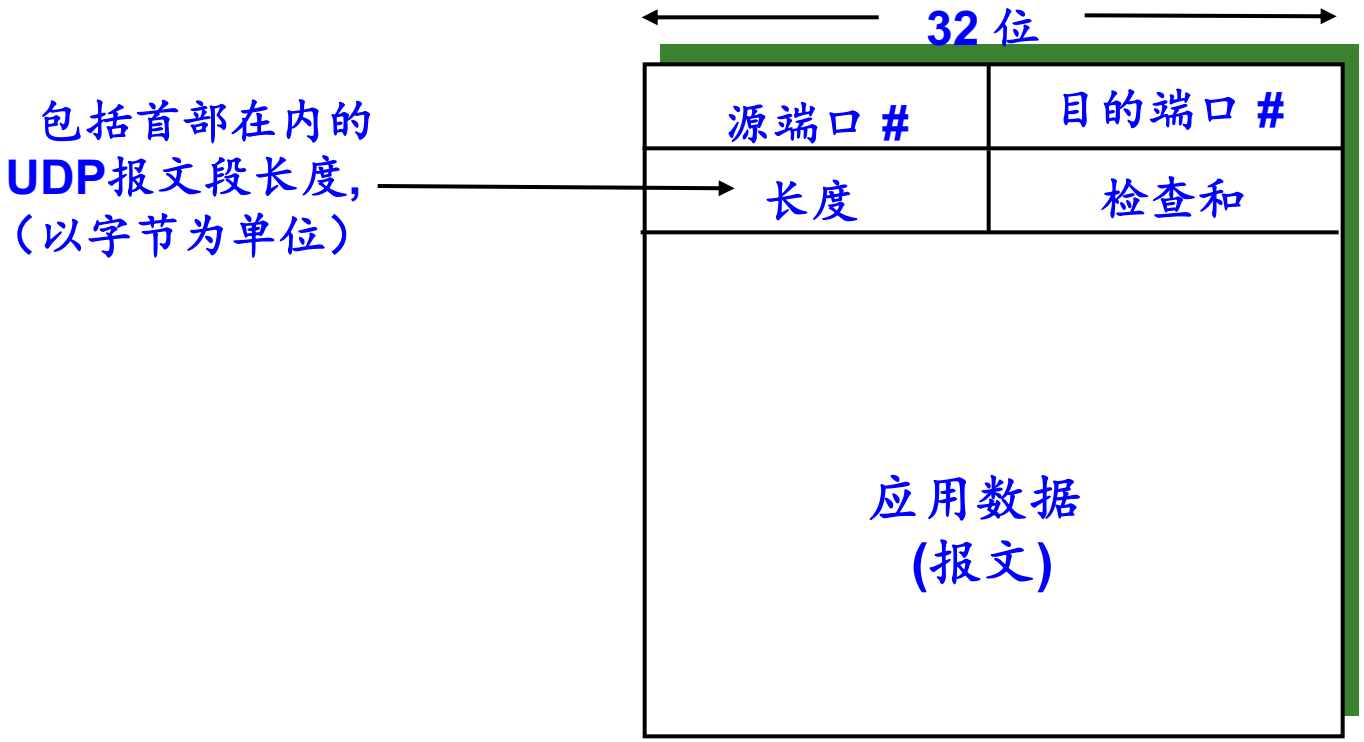
3.3 无连接传输：UDP

- **UDP大量应用可能导致的严重后果**
  - 路由器中大量的分组溢出
  - 显著减小TCP通信的速率，甚至挤垮TCP会话

- **使用UDP的可靠数据传输**
  - 在应用层实现数据的可靠传输
  - 增加了应用进程的实现难度

3.3 无连接传输：UDP

- **UDP报文段（*数据报*）的结构**

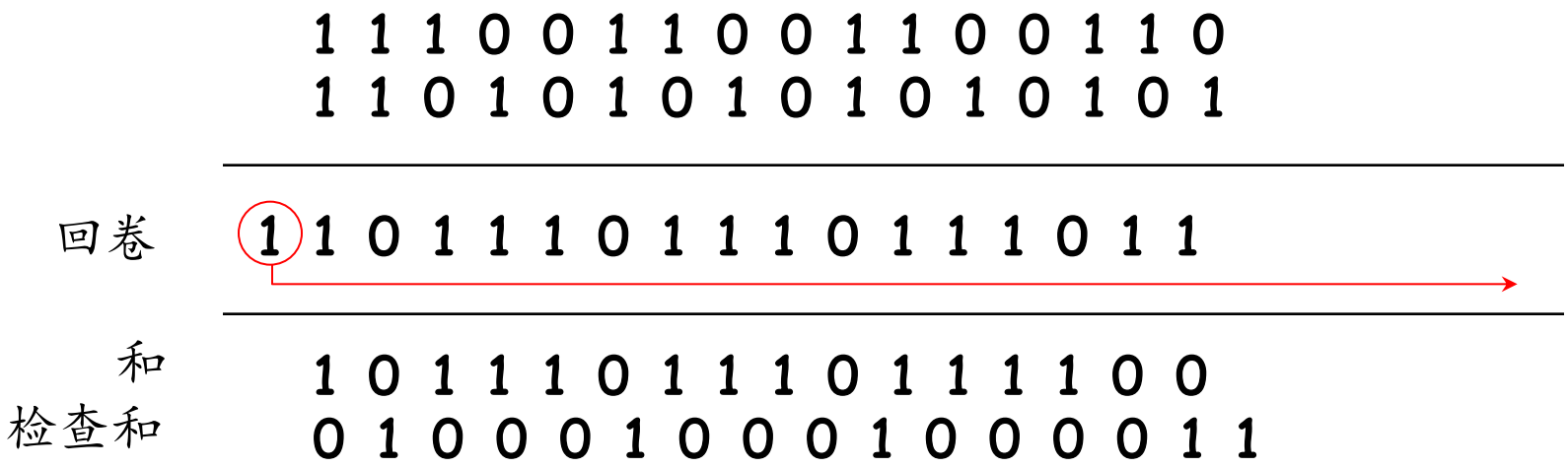


3.3 无连接传输：UDP

- **UDP的检查和**
  - 目标
    - 检测收到的报文段的“差错”（例如，出现突变的比特）
  - 发送方
    - 把报文段看作是16比特字的序列
    - 检查和：对报文段的所有16比特字的和进行1的补运算
    - 发送方将计算校验和的结果写入UDP校验和字段中
  - 接收方
    - 计算接收到的报文段的校验和
    - 检查计算结果是否与收到报文段的校验和字段中的值相同
      - 不同 — 检测到错误
      - 相同 — 没有检测到错误(但仍可能存在错误)

3.3 无连接传输：UDP

- **例子：将两个16比特字相加**

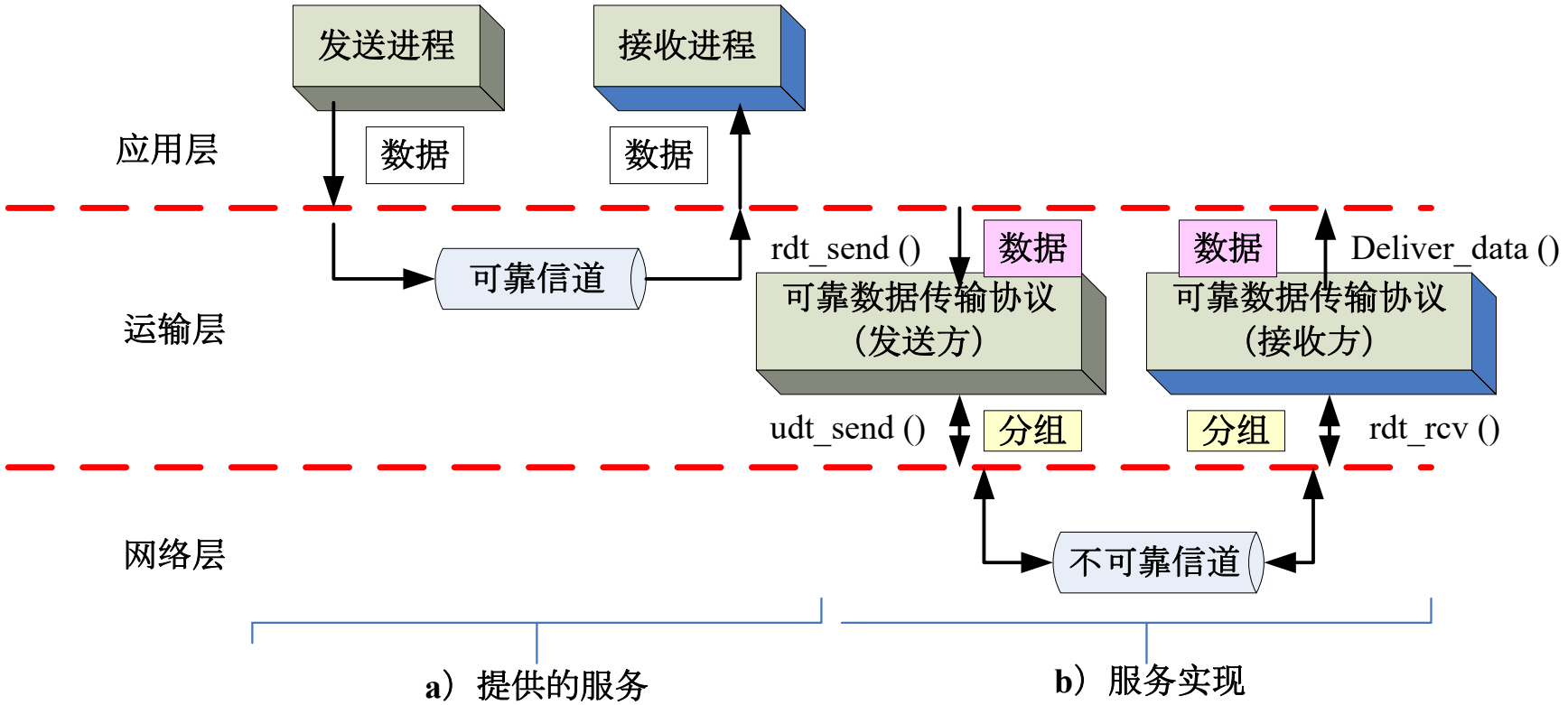


注意： 最高有效位的进位要回卷加到结果当中

3.4 可靠数据传输的原理

- **可靠数据传输**
  - 在应用层、运输层和链路层都很重要
  - 网络中最重要的top-10问题之一!

3.4 可靠数据传输的原理

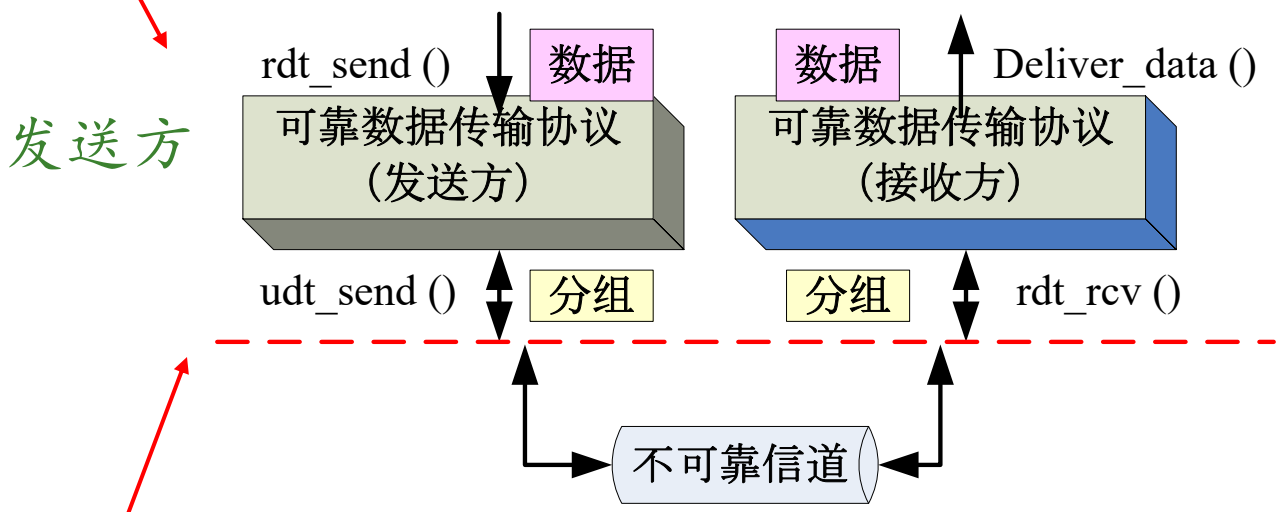


不可靠信道的特性决定了可靠数据传输协议(rdt)的复杂性。

3.4 可靠数据传输的原理

rdt_send(): 由上层（如应用层）调用，将数据发送给接收方的上层

deliver_data(): 由 **rdt** 调用，将数据交付上层



udt_send(): 由 rdt调用，将分组通过不可靠通道传给接收方

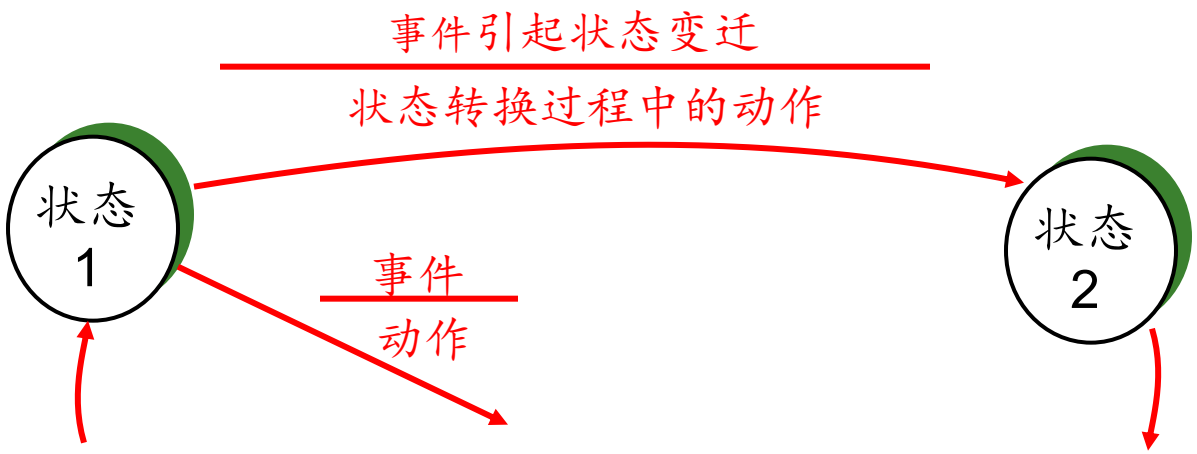
rdt_rcv(): 当分组到达接收方时调用

3.4 可靠数据传输的原理

我们将要:

- 逐步地开发可靠数据传输协议(rdt)的发送方和接收方
- 只考虑单向数据传输的情况
  - 但控制信息是双向传输的!
- 用有限状态机 (FSM) 来描述发送方和接收方

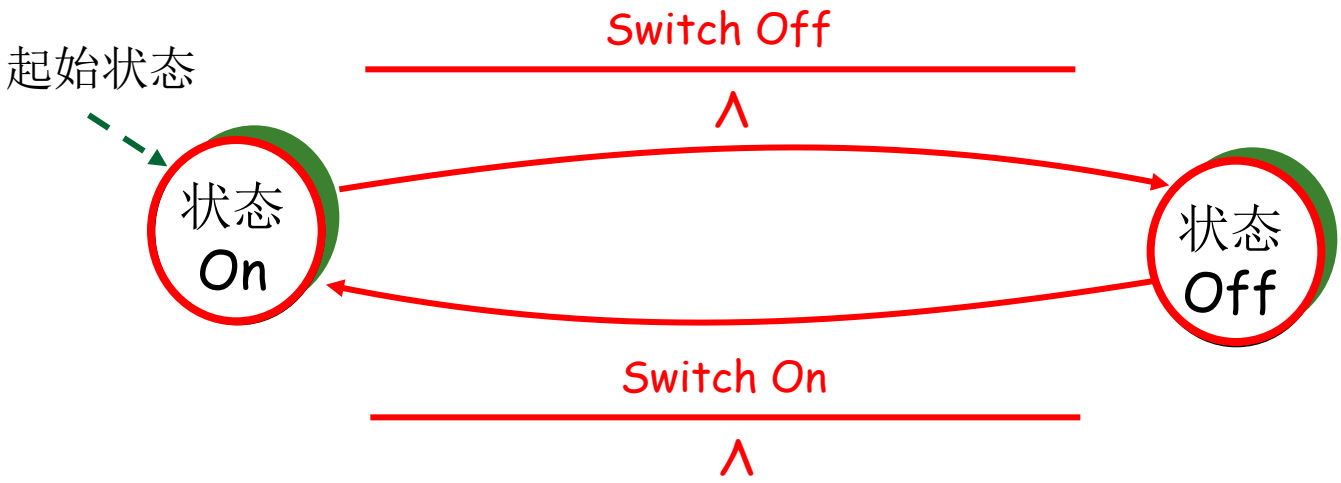
状态: 由事件引起一个状态到另一个状态的变迁。



3.4 可靠数据传输的原理

考虑用FSM来描述一个开关

- 开关有二个状态：On， Off
- 导致状态的变迁的事件：Switch On, Switch Off

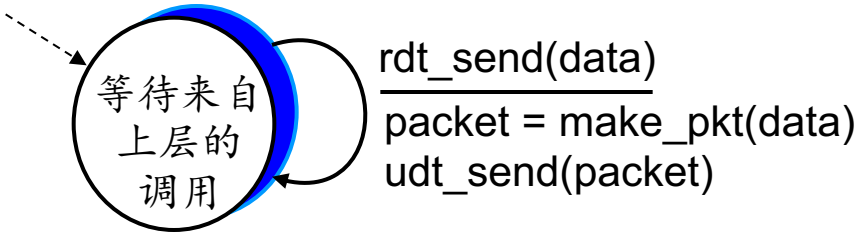


3.4 可靠数据传输的原理

- **可靠信道上的可靠传输—— rdt 1.0**
  - 底层信道完全可靠
    - 不会产生比特错误
    - 不会丢失分组
  - 分别为发送方和接收方建立FSM
    - 发送方将数据发送给底层信道
    - 接收方从底层信道接收数据

packet = make_pkt(data)
将应用层传来的数据data打包成网络层的packet

extract (packet, data)
将网络层传来的packet解包成数据data



发送方

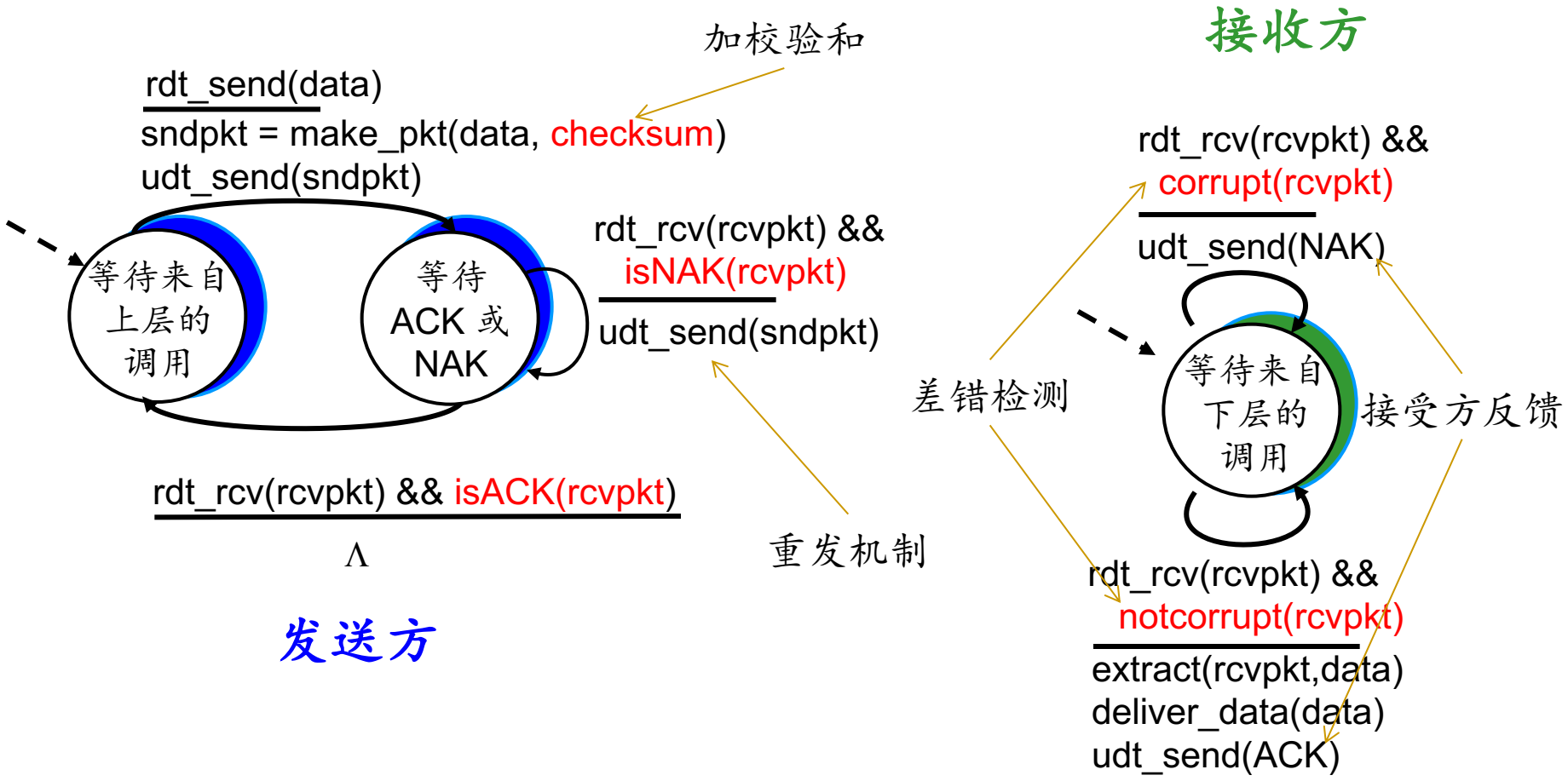
接收方

3.4 可靠数据传输的原理

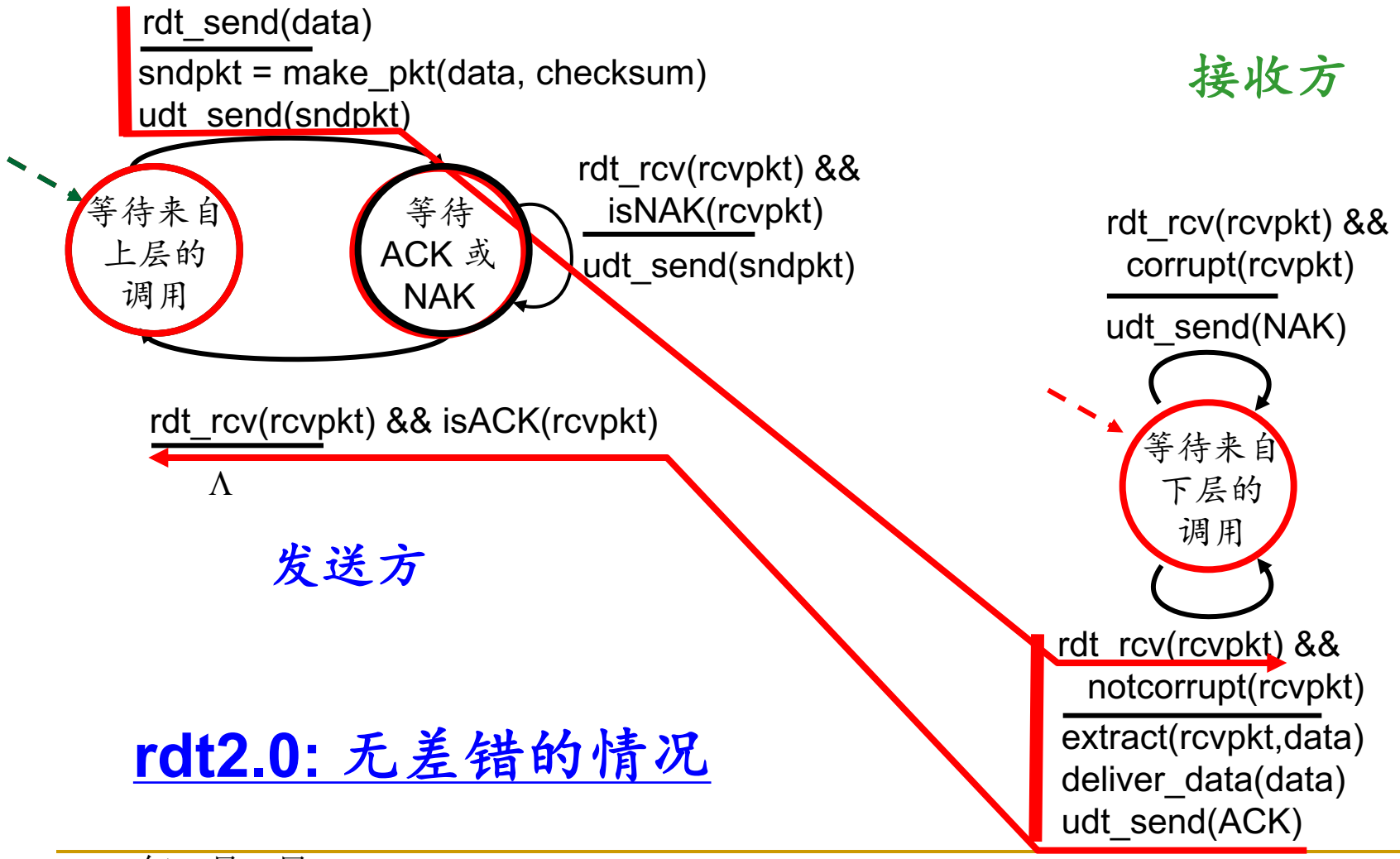
- **信道可能导致比特出现差错时——rdt 2.x**
  - 第一个版本——rdt 2.0
    - 假设
      - 分组比特可能受损
      - 所有传输的分组都将按序被接收，不会丢失
    - 处理机制
      - 如何判断分组受损——差错检测
      - 如何通知发送方分组是否受损——接收方反馈（ACK和NAK）
        - 确认——acknowledgements (ACKs): 接收方明确告诉发送方正确收到分组
        - 否认——negative acknowledgements (NAKs): 接收方明确告诉发送方分组有错
      - 在得知分组受损后，发送方如何处理——出错重传

3.4 可靠数据传输的原理

- rdt 2.0的有限状态机FSM

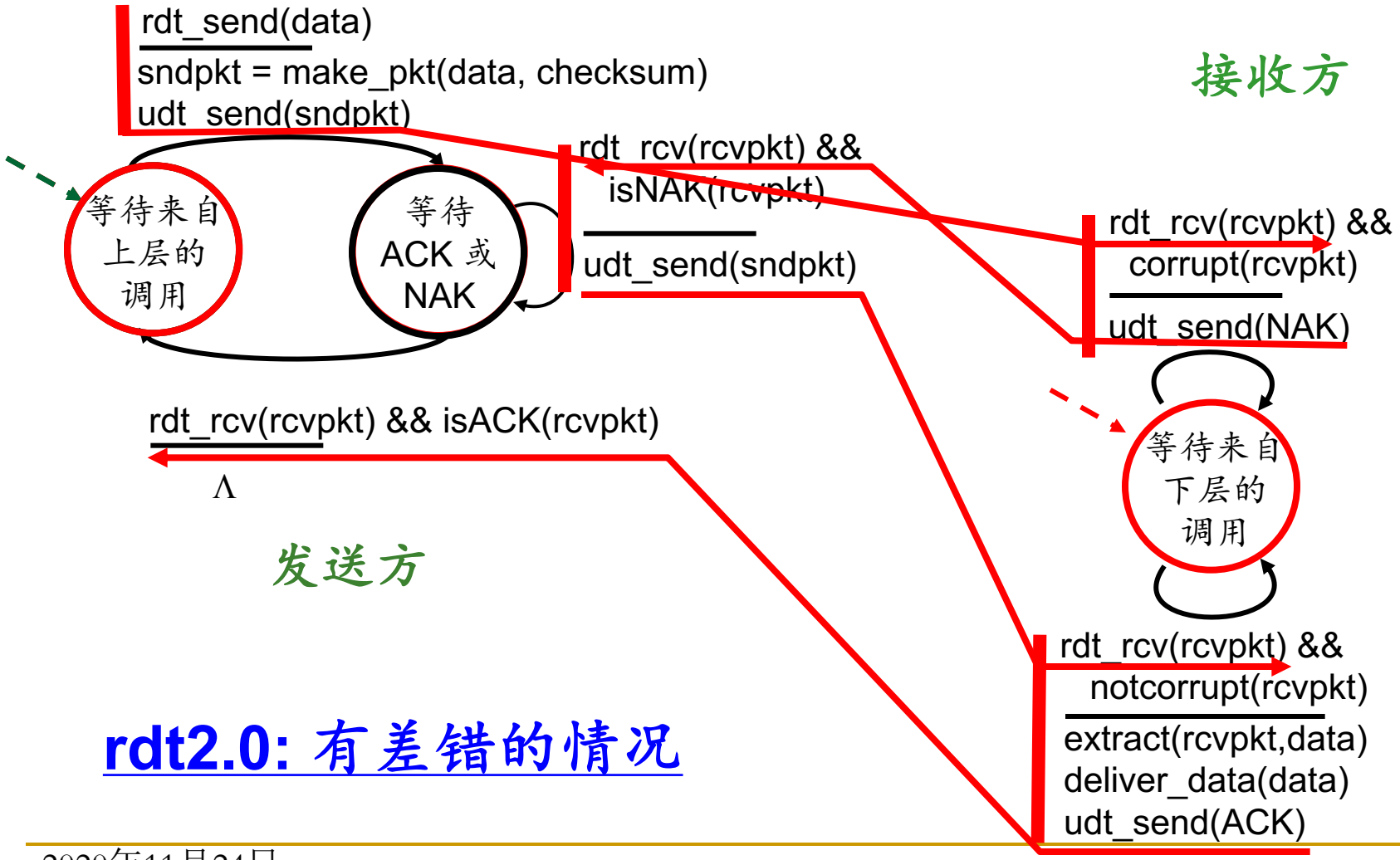


3.4 可靠数据传输的原理



rdt2.0: 无差错的情况

3.4 可靠数据传输的原理



rdt2.0: 有差错的情况

3.4 可靠数据传输的原理

- 如何实现重传
  - 使用缓冲区缓存已发出但未收到反馈的报文段
- 新的问题
  - 需要多大的缓冲区呢?
- 接收方和发送方各一个报文段大小的缓冲区即可

rdt 2.0 存在什么问题?

3.4 可靠数据传输的原理

首先rdt2.0 有一个致命的缺陷!没有考虑ACK/NAK受损的情况

如果ACK/NAK受损会出现什么情况?

- 发送方不知道接收方发生了什么!
- **因此必须对ACK和NAK加校验和**

3.4 可靠数据传输的原理

rdt2.0 另一个问题：只考虑重传可能会出现大量重复分组

- 接受方无法判断接受到的一个分组是重发的还是新的分组

停止等待(stop-and-wait)

发送方发出一个分组，然后等待接收方的应答

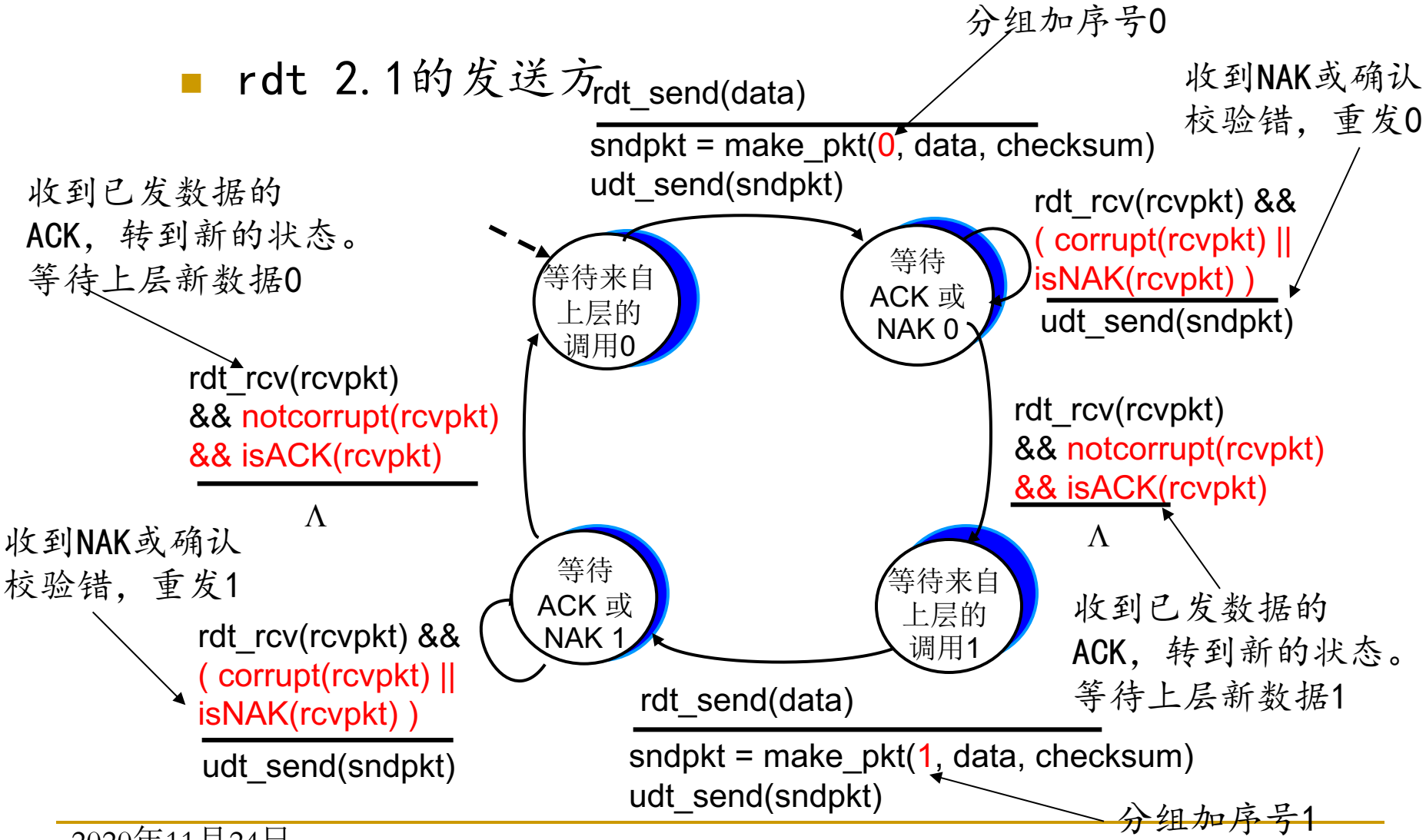
我们开发新的版本rdt 2.1

对重复分组的处理：

- 发送方对每一个分组增加序号（*sequence number*）
- 发送方只重传收到NAK(校验正确)或校验发生错误的ACK/NAK 的分组
- 接收方丢弃重复分组(不向上递交)
- 由于rdt2.0是一个停止等待协议，所以用1bit的序号足够了（区分是在重发前一分组和新的分组），序号变化按模2运算计算。
- 因此发送方和接受方的FSM的状态数是以前的2倍，因为协议状态必须反映出目前正发送和接收的分组序号是0还是1

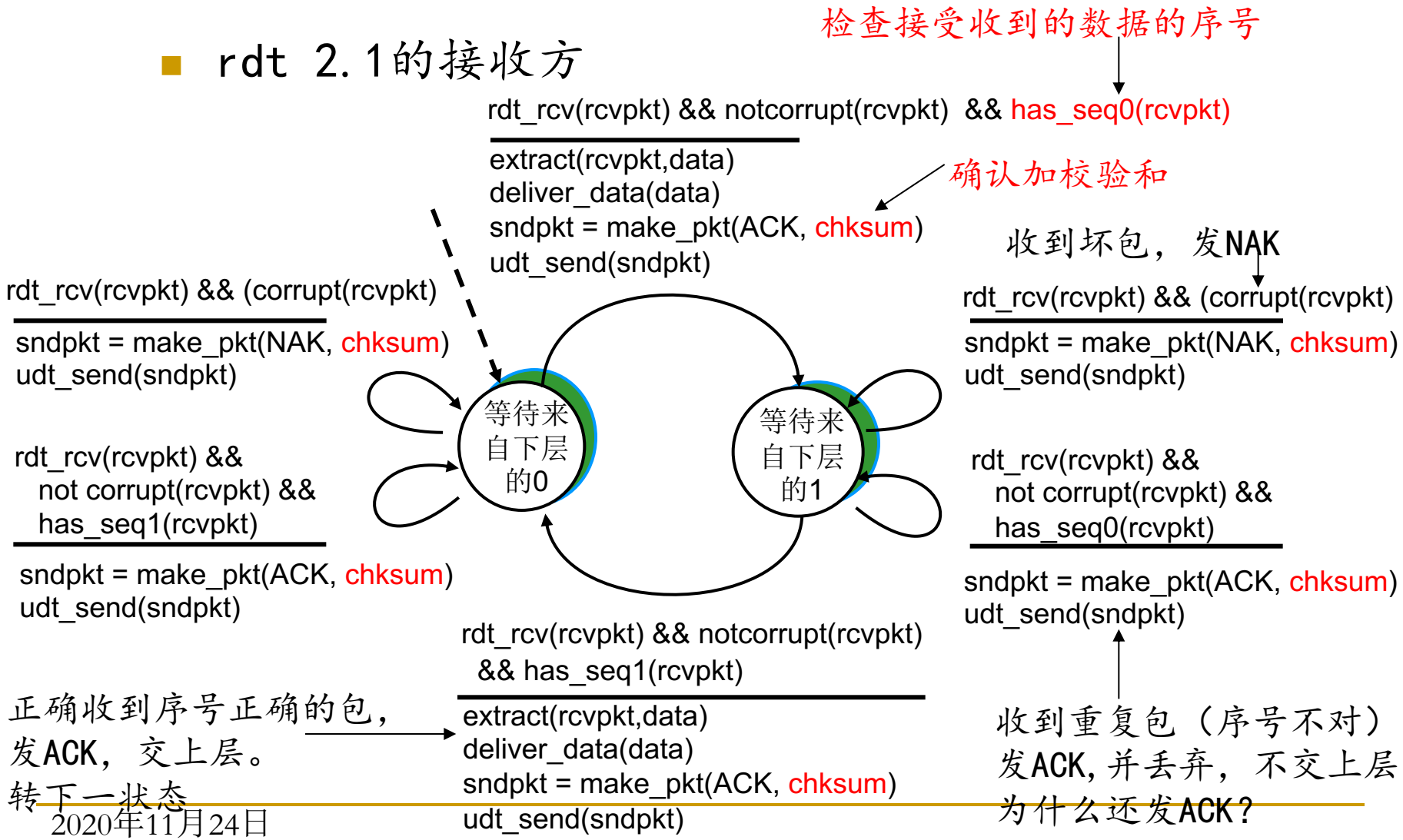
3.4 可靠数据传输的原理

- rdt 2.1的发送方



3.4 可靠数据传输的原理

- rdt 2.1的接收方



rdt2.1: 讨论

发送方：

- 分组中增加序号
- 两个序号 (0,1)是否足够，为什么?
- 必须检查是否收到的ACK/NAK是否损坏，以防止确认错误
- 状态数是以前的两倍
  - 状态必须反映出当前正在发送的分组的序号是0还是1

接收方：

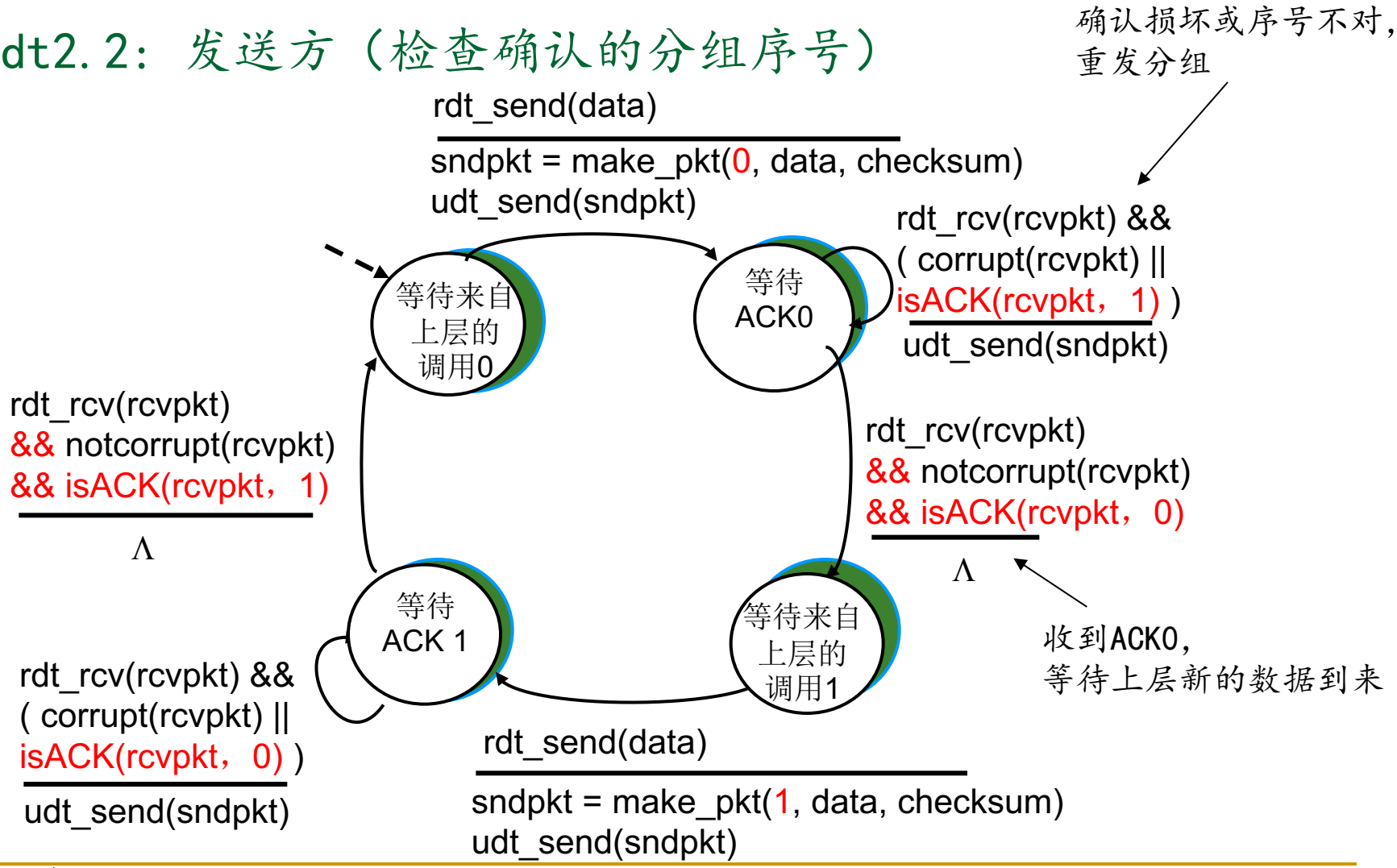
- 必须检查是否收到的是重复的分组
- 状态必须反映出当前希望接收的分组的序号是0还是1
- 注意：接收方不知道它最后发出的ACK/NAK是否被发送方正确接收

3.4 可靠数据传输的原理

- 第三个版本——rdt 2.2
  - 针对rdt 2.1的改进
    - 只使用ACK
    - 取消NAK，接收方对最后一个正确收到的分组发送 ACK
      - 接收方必须明确指出被确认的分组的序号
    - 发送方收到的重复的ACK将按照NAK来进行处理
      - 重传正确的分组

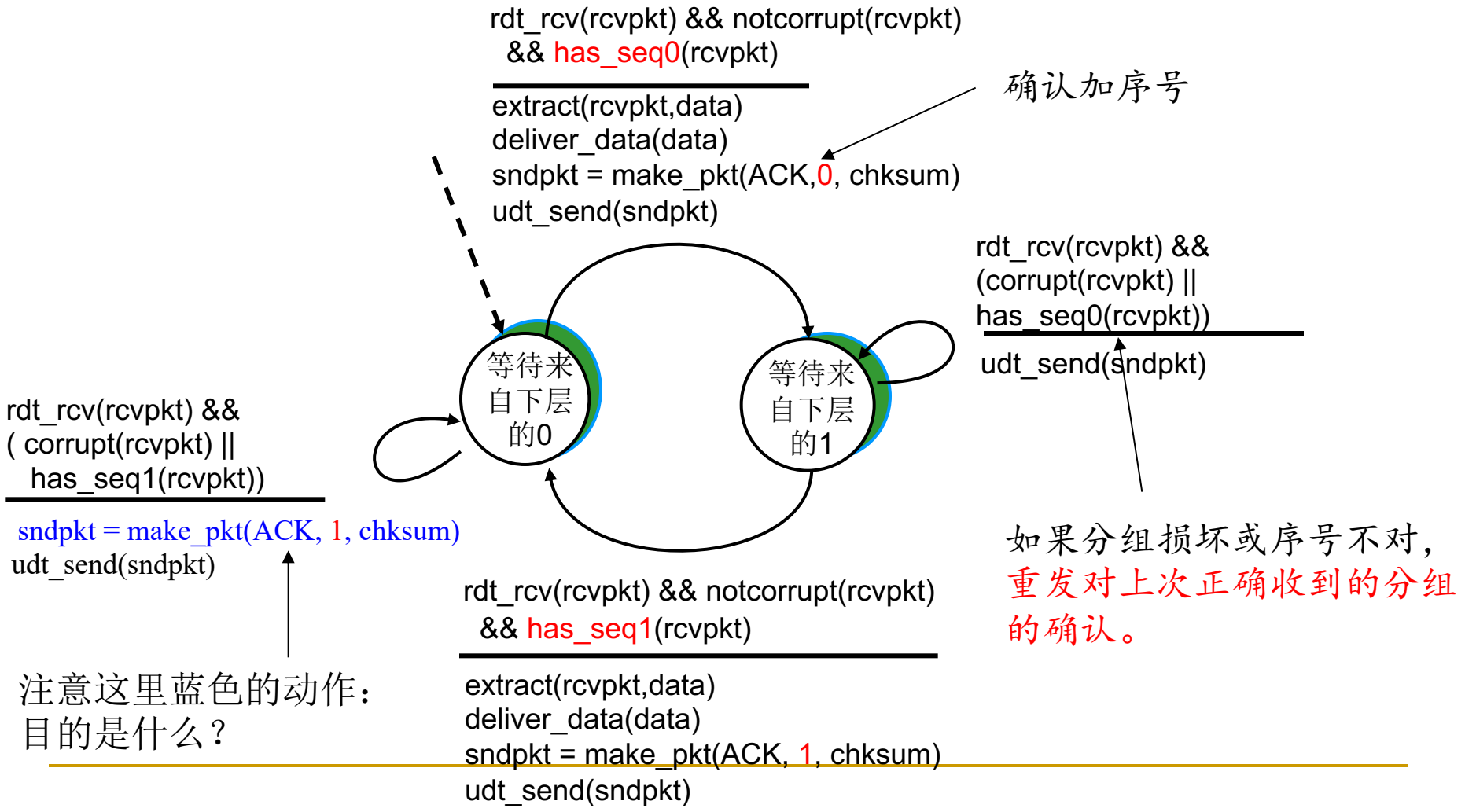
3.4 可靠数据传输的原理

rdt2.2：发送方（检查确认的分组序号）



3.4 可靠数据传输的原理

rdt2.2：接受方(确认指明了所确认的分组序号)



3.4 可靠数据传输的原理：rdt3.0

新的假设：

底层信道会丢包（数据或 ACK）

- 检查和，序号，ACK和重传可以有助于解决问题，但这些远远不够

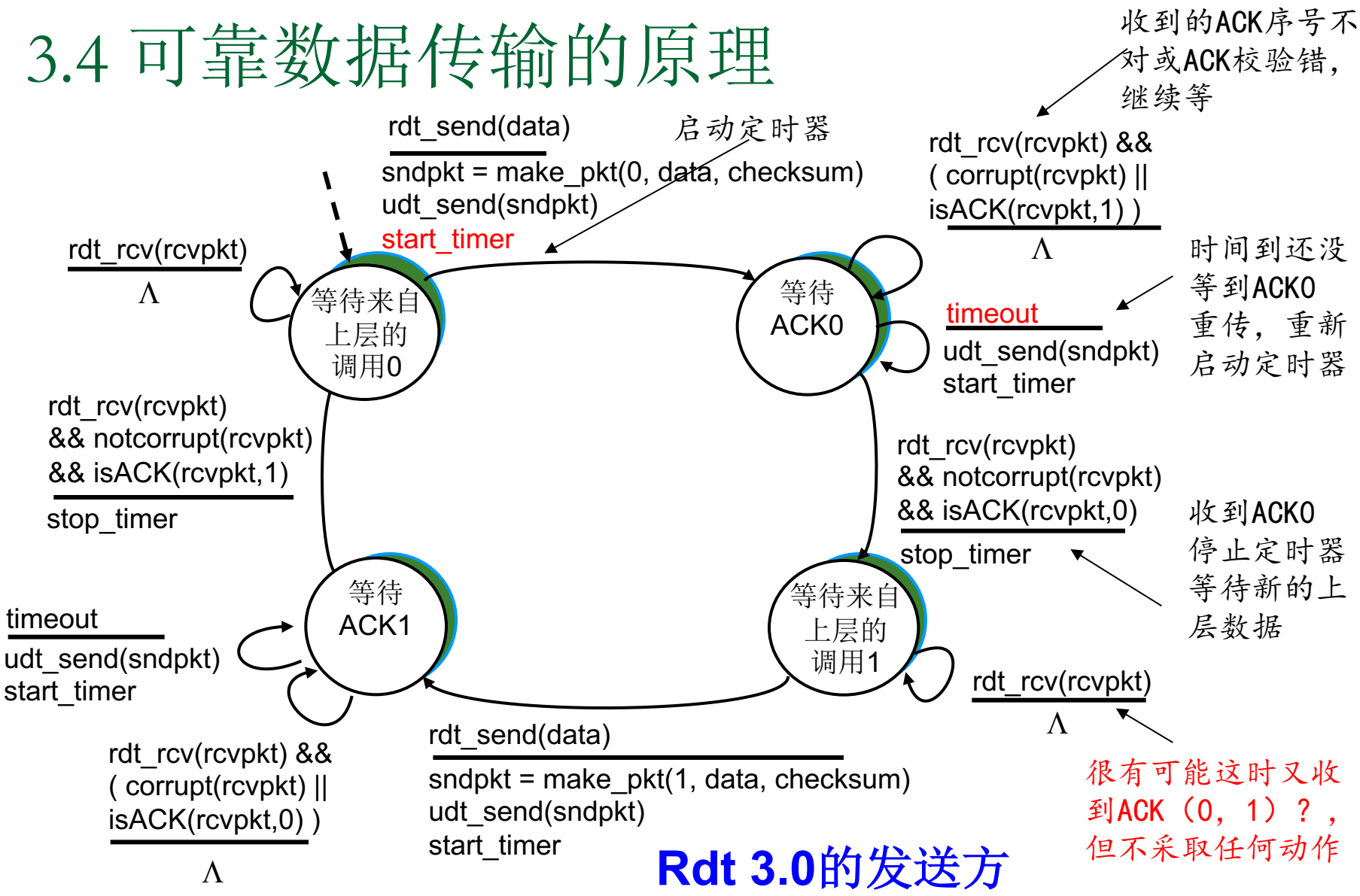
Q：如何处理数据丢失？

- 发送方可以等待，当某些数据或ACK 丢失时，进行重传
- 想一想：等多久？

解决方法：发送方对ACK等待“适当的”时间

- 如果在这个时间内没有收到ACK则重传
- 如果分组或ACK仅仅是延迟到达（而非丢失）：
  - 重传将造成重复，但序号可以解决这个问题
  - 接收方必须指出确认的分组序号
- 需要倒计时的计时器
- 我们只需要修改发送方

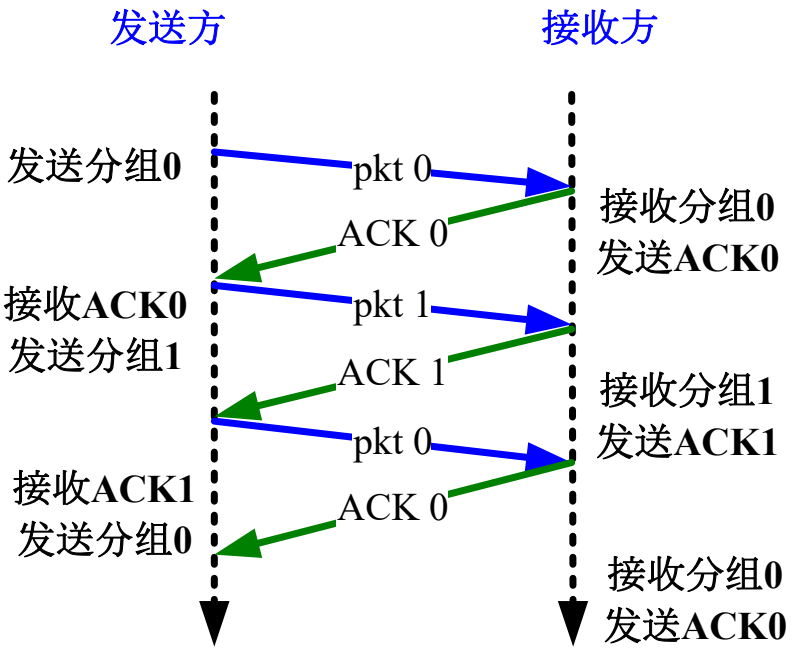
3.4 可靠数据传输的原理



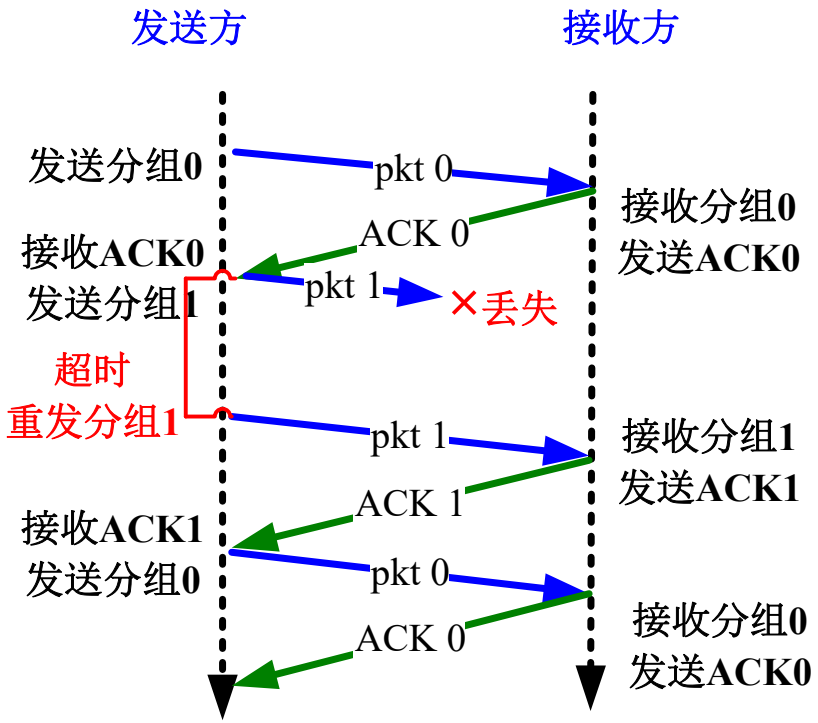
Rdt 3.0的发送方

3.4 可靠数据传输的原理

- rdt 3.0举例



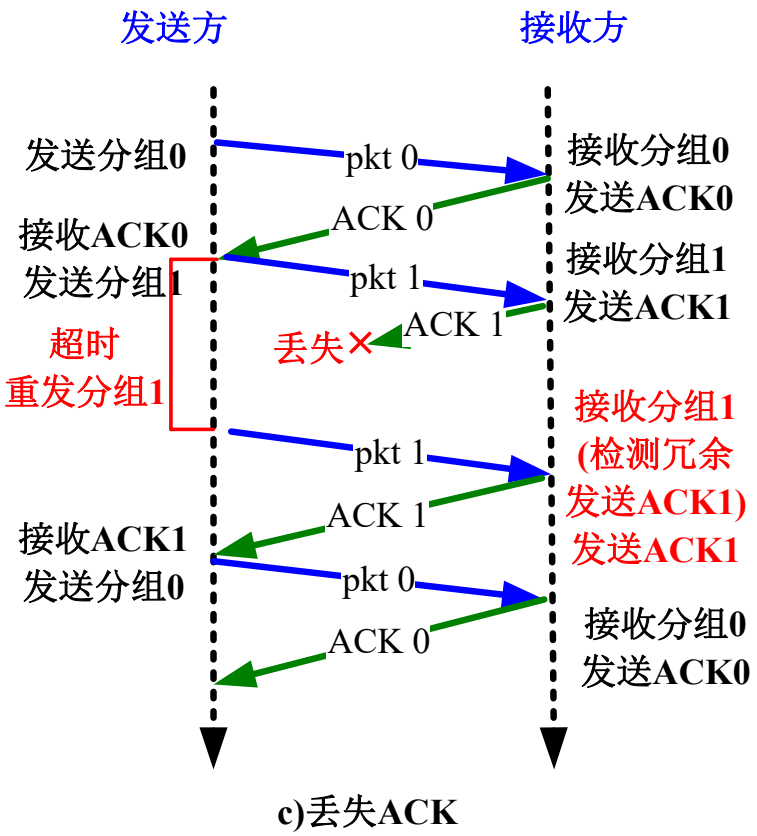
a)无丢包操作



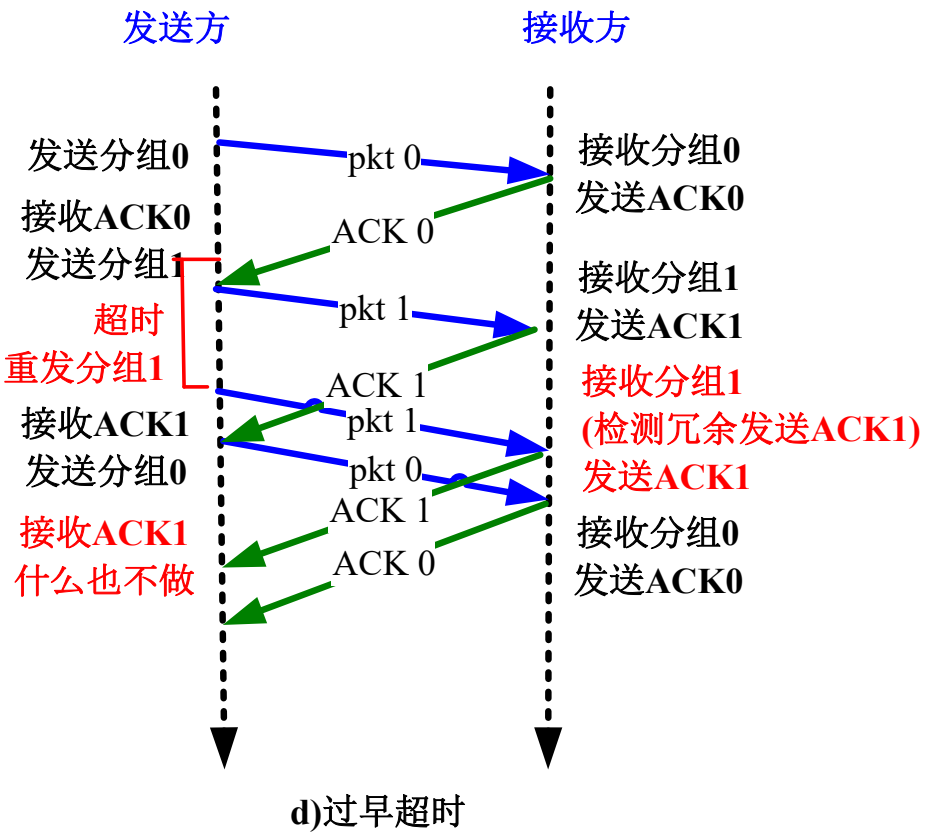
b)分组丢失

下续

3.4 可靠数据传输的原理



c)丢失ACK



d)过早超时

3.4 可靠数据传输的原理

- **rdt 3.0的性能分析：是停止等待协议**
  - 1Gbps 的链路, 15ms 的端到端延迟, 分组大小为1KB

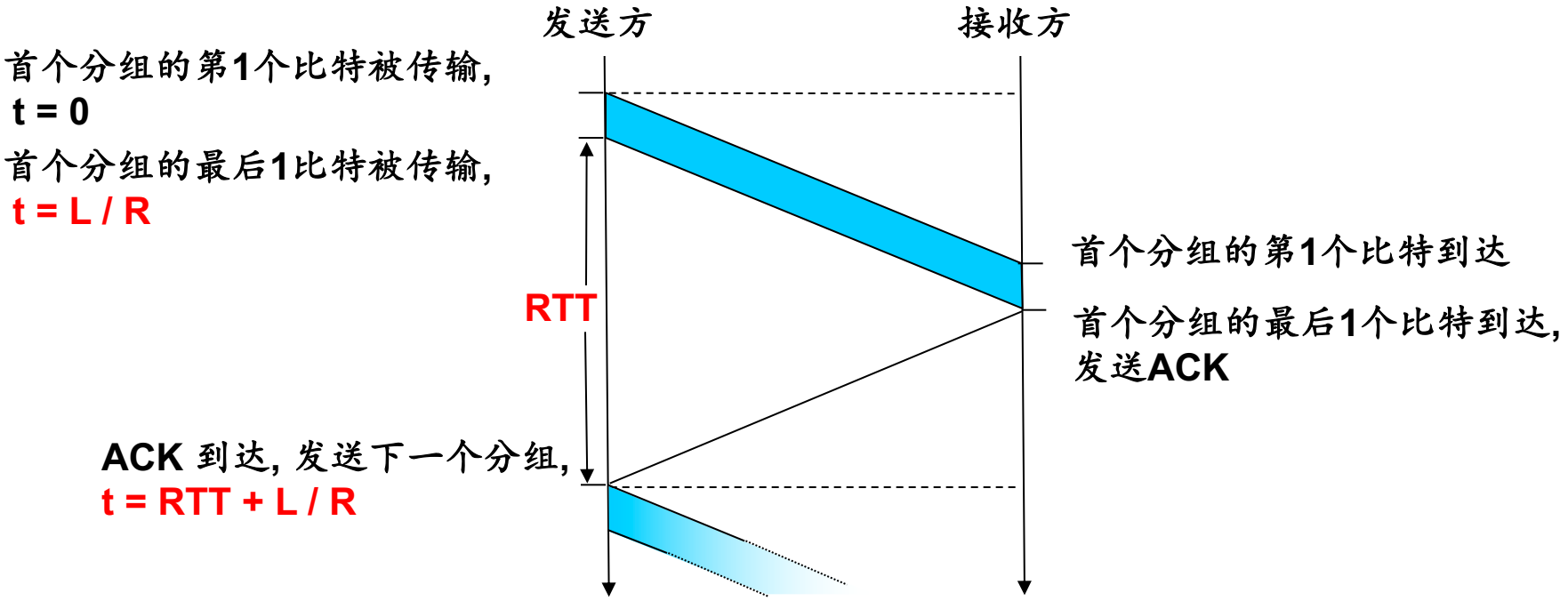
$$T_{transmit} = \frac{L\ (比特为单位的分组大小)}{R\ (传输速率, bps)} = \frac{8kb/pkt}{10^9\ b/sec} = 8\ \mu s$$

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

  - 发送方只有万分之2.7 的时间是忙的
  - 每30ms（1个RTT）内只能发送1KB： 1 Gbps 的链路只有33kB/sec（1KB/30ms） 的吞吐量
  - 网络协议限制了物理资源的利用率

3.4 可靠数据传输的原理

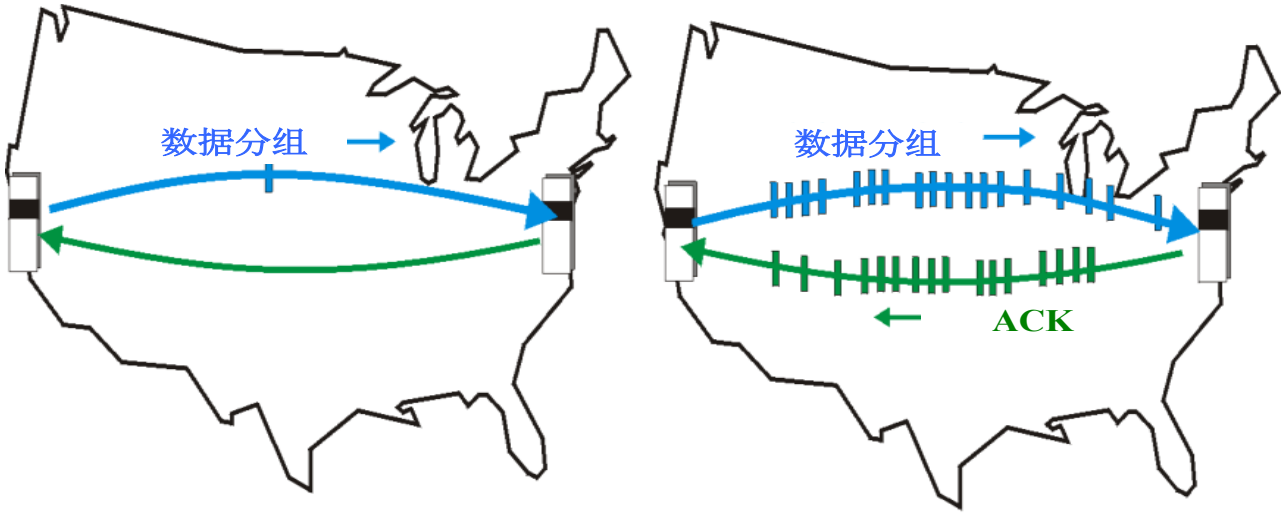
- rdt 3.0性能低下的原因



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

3.4 可靠数据传输的原理

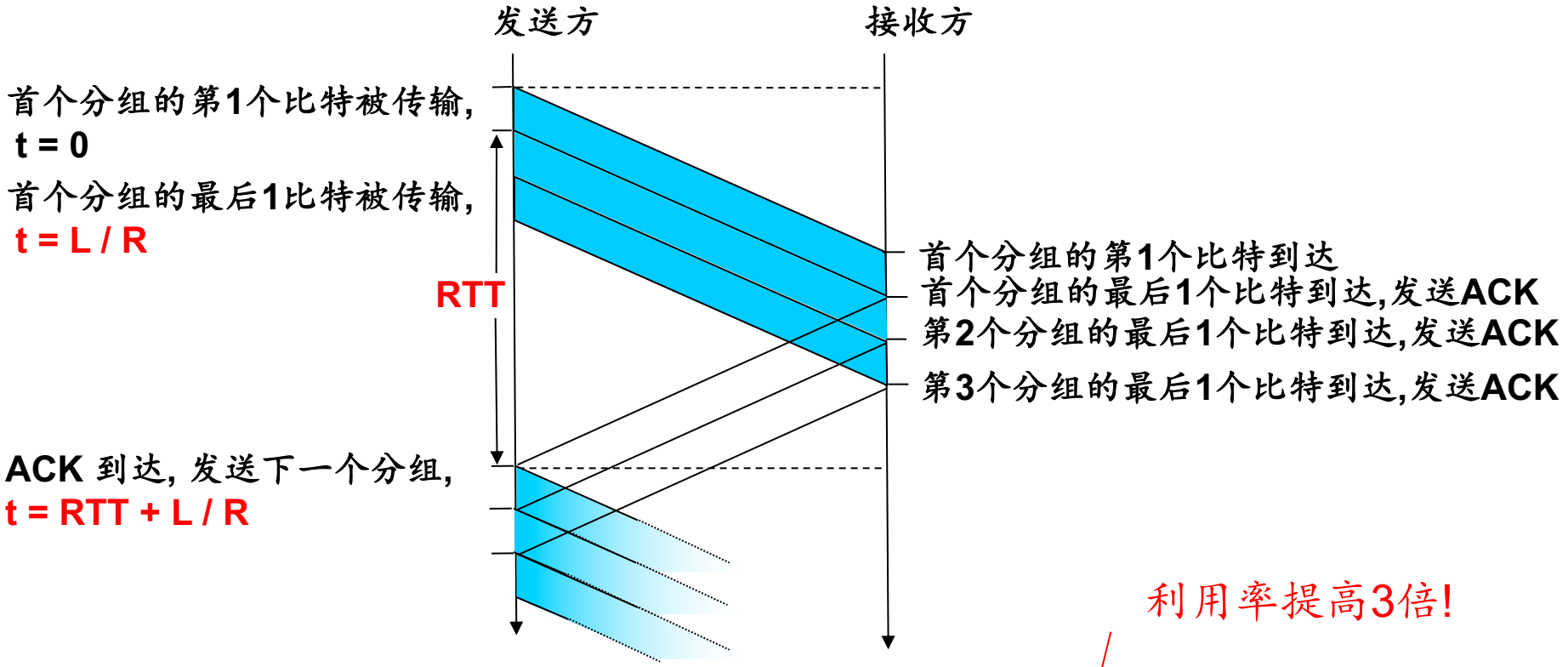
- 提高性能的一种可行方法：流水线技术
  - 允许发送方发送多个分组而无需等待确认
    - 必须增大序号范围
    - 协议的发送方和接收方必须对分组进行缓存



(a) 停等操作　　(b) 流水线操作

3.4 可靠数据传输的原理

- 流水线技术对性能提升的原理图
- 假设发送方可以在等待确认前发送三个分组



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

利用率提高3倍!

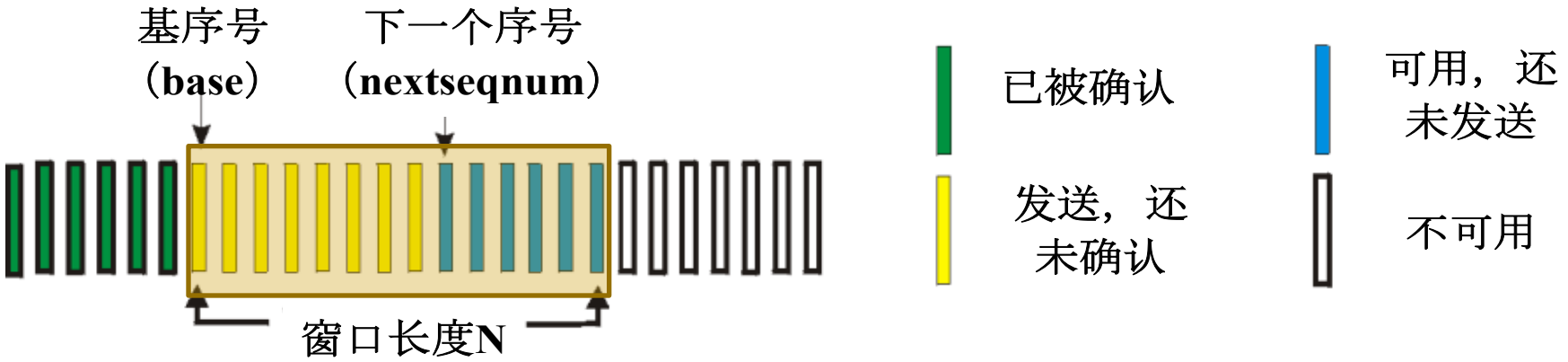
3.4 可靠数据传输的原理

- 问题：当流水线技术中丢失一个分组后，如何进行重传
  - Go-Back-N（GBN，回退N步）协议：其后分组全部重传
  - 选择重传（SR）协议：仅重传该分组

Go-Back-N：允许发送方发送多个分组而不需等待确认，但已发送但未确认的分组数不能超过N

发送方：

- 分组首部用k-比特字段表示序号(二进制)
- 已被传输但还未确认的分组的许可序号范围可以看作是一个在序号范围内大小为N的“窗口(window)”



基序号：最早未确认的分组序号

下一个序号：下一个待发分组的序号

[0,base-1]:已被确认的分组

[base, nextseqnum-1]:已发送但未确认的分组

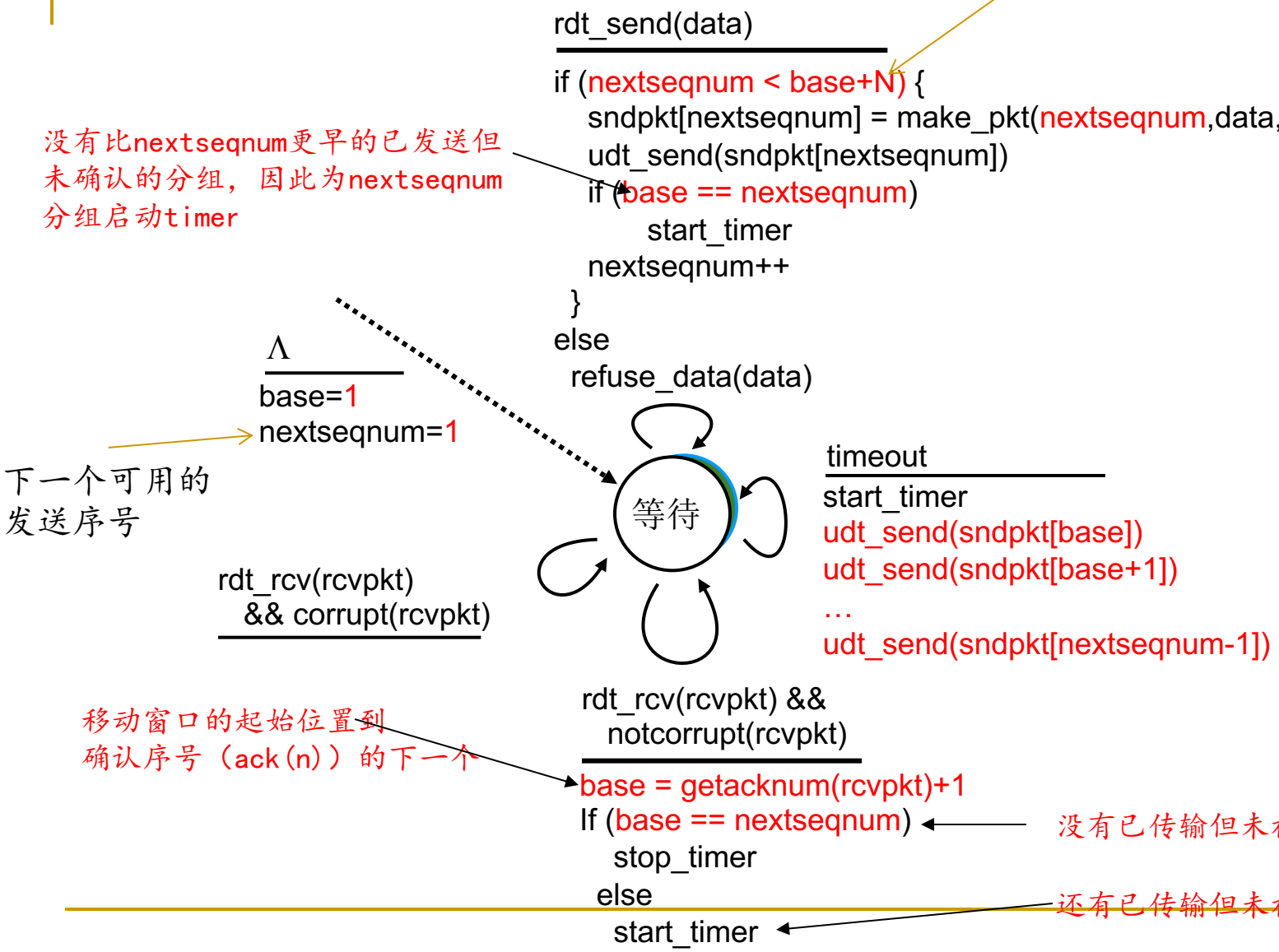
[nextseqnum, base+N-1]:可以被发送的分组

大于等于base+N的分组：不能发送

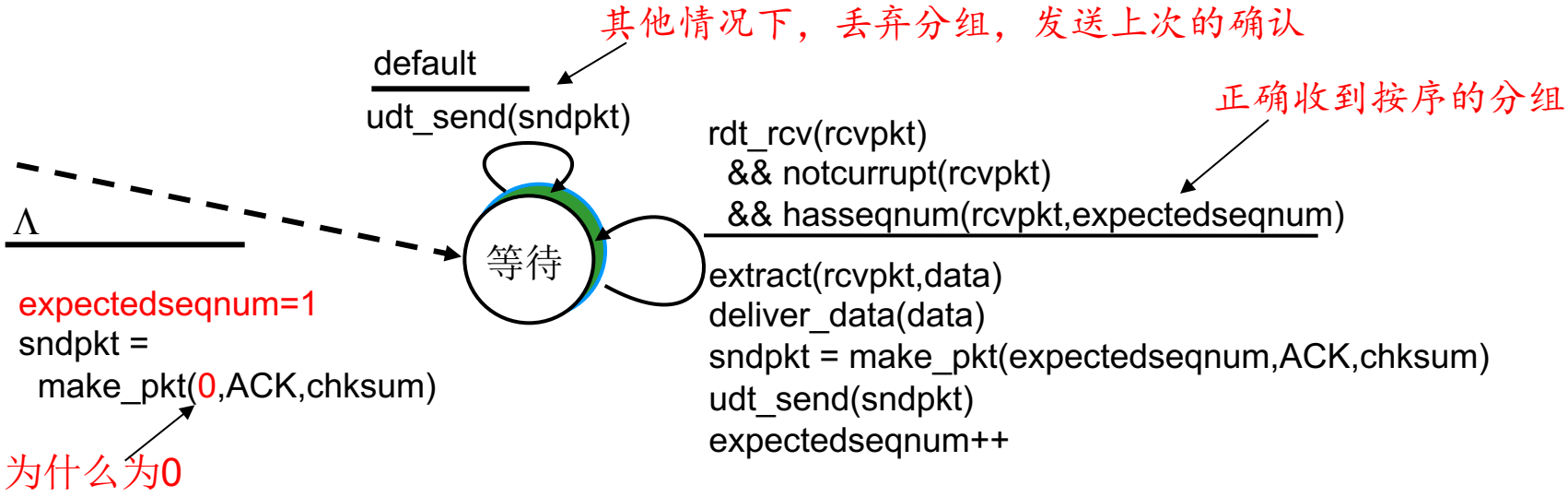
Go-Back-N(续)

- ACK(n)：对序号n之前包括n在内的所有分组进行确认
  - “累积 ACK”（确认分组n一次就足够）
  - ACK-only：只对正确按序到达的分组发送ACK
- 为最早已发送但未确认的分组设置定时器（只需要一个定时器）
- *超时*：重传所有已发送但没被确认的分组
- 为什么要限制滑动窗口大小：为了流量控制

GBN: 发送方的扩展FSM



GBN: 接收方的扩展 FSM

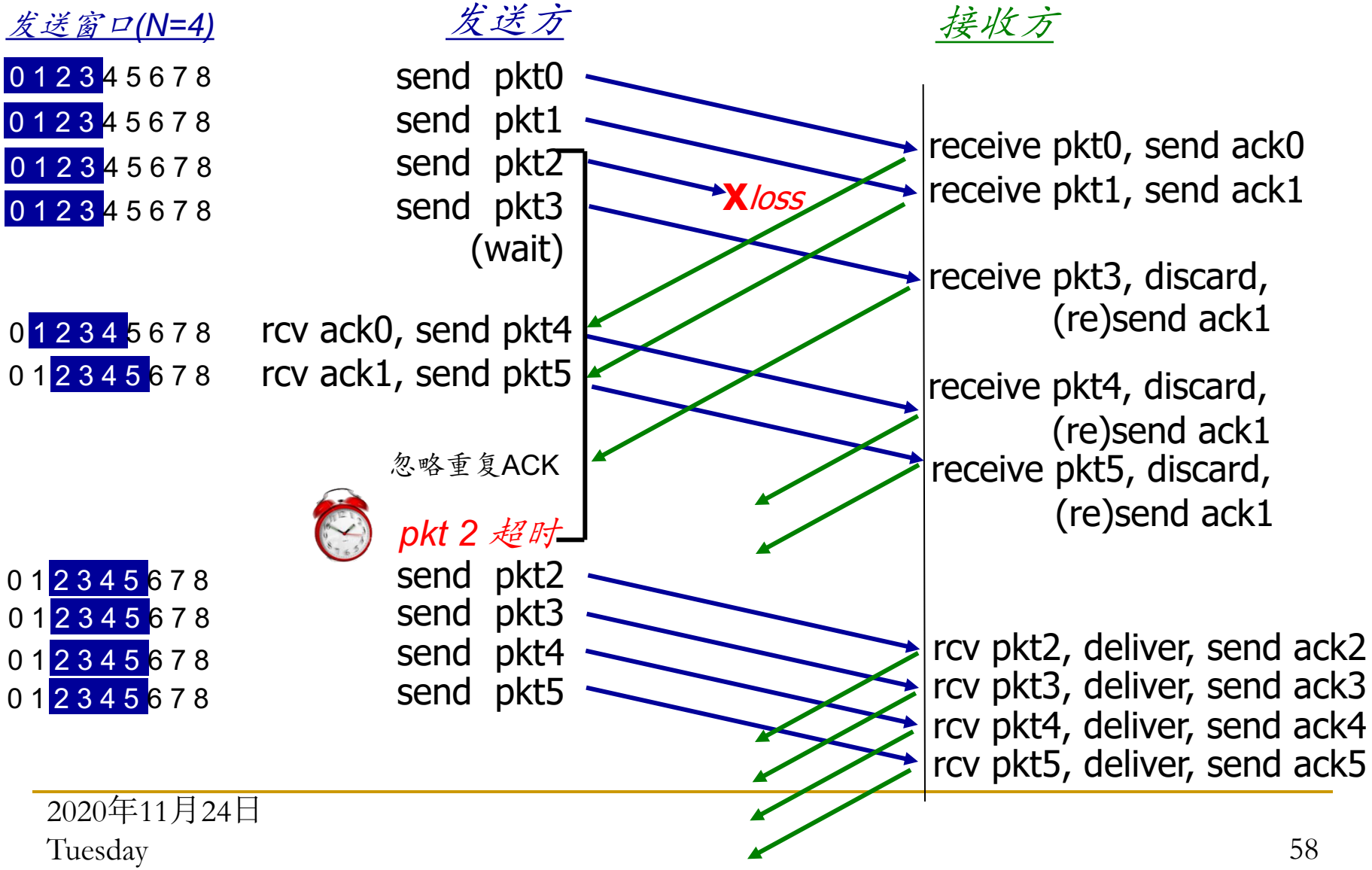


ACK-only：对正确按序到达的分组发送ACK

- 可能会产生重复的ACK
- 需要记住期待序号 expectedseqnum

- 失序分组或损坏分组：
  - 丢弃（不缓存）-> 接收方无缓存！
  - 重发正确按序到达的最高序号分组的ACK
  - 每次发送的ACK一定是对正确按序到达的最高序号分组的确认

3.4 可靠数据传输的原理



3.4 可靠数据传输的原理

- **Go-Back-N协议**
  - 特点
    - ACK(n): 接收方对序号n之前包括n在内的所有分组进行确认 - “累积 ACK”
    - 对所有已发送但未确认的分组统一设置一个定时器
    - *超时(n):* 重传分组n和窗口中所有序号大于n的分组
    - 接收方收到失序分组:
      - 丢弃 (不缓存) -> 接收方无缓存!
      - 重发按序到达的最高序号分组的ACK

3.4 可靠数据传输的原理

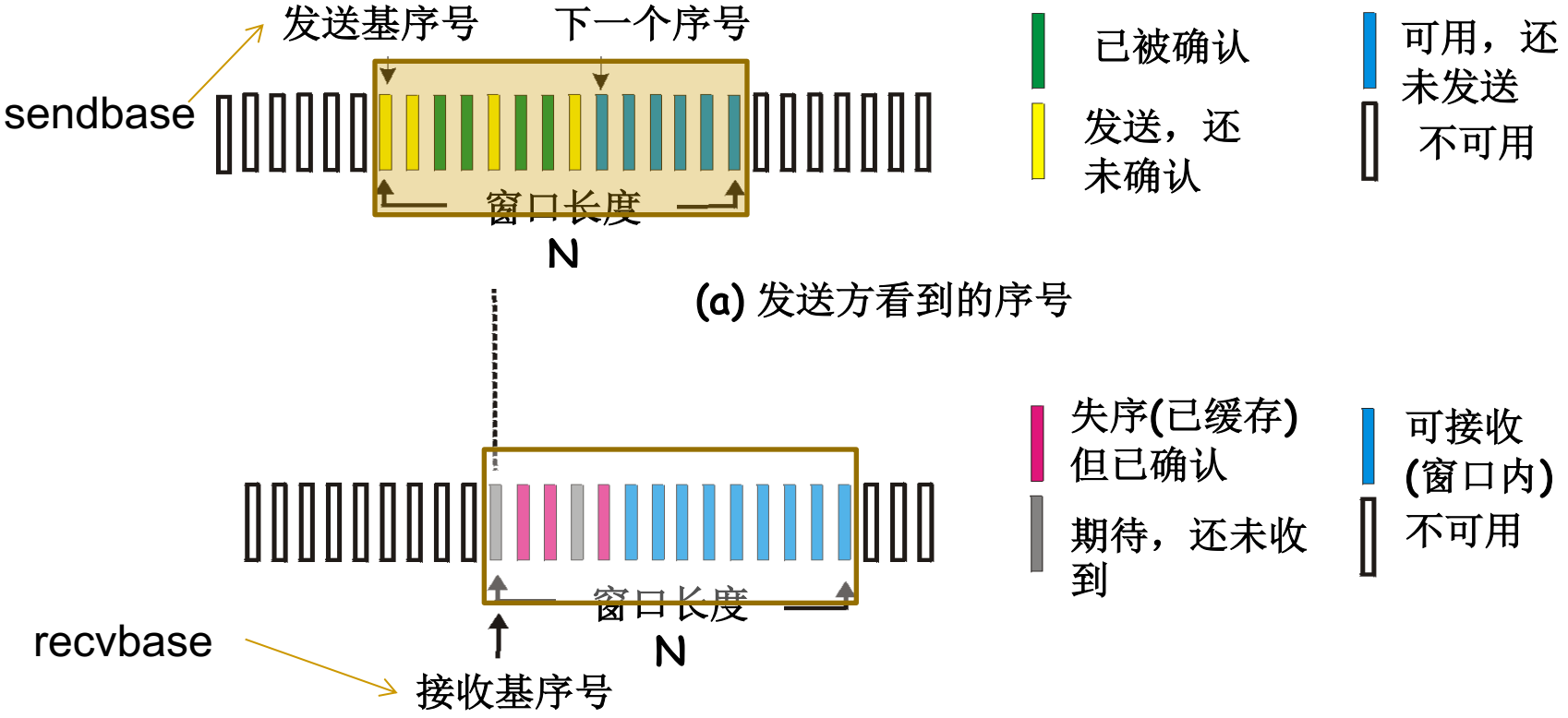
- Go-Back-N的滑动窗口大小
  - 发送端 ≤ 2^k-1
  - 接收端 ＝ 1

3.4 可靠数据传输的原理

- 选择重传:解决GBN大量重传分组的问题
- 接收方逐个对所有正确收到(即使失序)的分组进行确认（不是累积确认）
  - 对接收到的（失序）分组进行缓存（GBN不缓存），以便最后对上层进行有序递交
- 发送方只重发怀疑丢失或损坏的分组
  - 发送方为每一个没有收到ACK的分组设置定时器
- 发送窗口
  - 大小为N，范围[sendbase，sendbase + N - 1]
  - 限制已发送但未被确认的分组数最多为N
  - sendbase以前的分组都被确认
- 接受窗口窗口
  - 大小为N,范围[recvbase，recvbase + N - 1]
  - 落在窗口内的序号都是期待收到的分组序号
  - recvbase前都是按序到达，已发出确认，且已递交给上层

3.4 可靠数据传输的原理

- **选择重传（SR）协议**



3.4 可靠数据传输的原理

选择重传（SR）协议

发送方

从上层收到数据 ：

- 如果下一个可用于该分组的序号在窗口内，则将数据打包并发送。否则要么缓存，要么拒绝上层

超时(n)：序号为n的分组超时

- 重传分组n，重置定时器

收到确认(n) 在 [sendbase, sendbase+N−1]范围内

- 标记分组 n 为已接收
- 如果n是发送窗口基序号sendbase，则将窗口基序号前推到下一个未确认序号（因此，sendbase前的分组一定都被确认过了），如果不是窗口不动
- 发送方可能会收到比sendbase还小的序号的分组确认，这时什么都不做(因为比sendbase还早的分组都被确认)

3.4 可靠数据传输的原理

- 选择重传（SR）协议
- 为什么发送方会收到比sendbase还小的分组确认？
  - 例如窗口位于sendbase-1时，序号为sendbase-1的分组定时器时间到还没收到Ack(sendbase-1)，则发送方重发该分组。
  - 刚重发，收到了Ack(sendbase-1)，窗口前移到sendbase
  - 窗口移动后，又收到了Ack(sendbase-1)

3.4 可靠数据传输的原理

■ 选择重传（SR）协议

接收方

分组序号n在[rcvbase, rcvbase+N-1]范围内

- 分组正确接收
- 发送n的确认ACK(n)（不管是否为重复分组及是否失序）
- 如果分组是以前没收到过的：将其缓存
- 如果该分组序号=recvbase，则将从该分组开始序号连续的分组一起交给上层，然后，窗口按向上交付的分组的数量向前移动

分组序号n 在 [rcvbase-N, rcvbase-1]范围内：

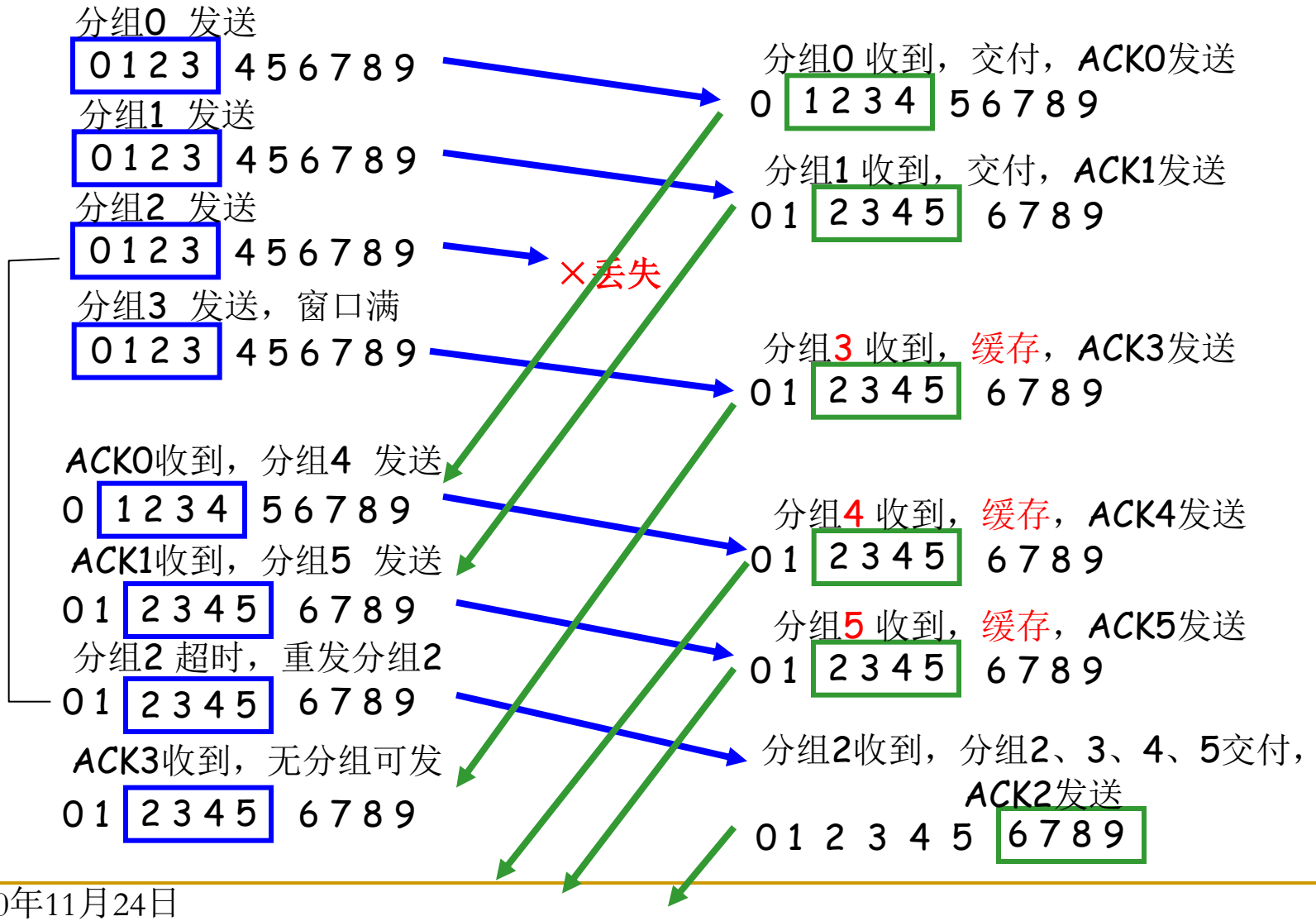
- 虽然曾经确认过，仍再次发送n的确认ACK(n)
- 不发确认，发送方窗口无法向前移动

其他情况：忽略该分组

3.4 可靠数据传输的原理

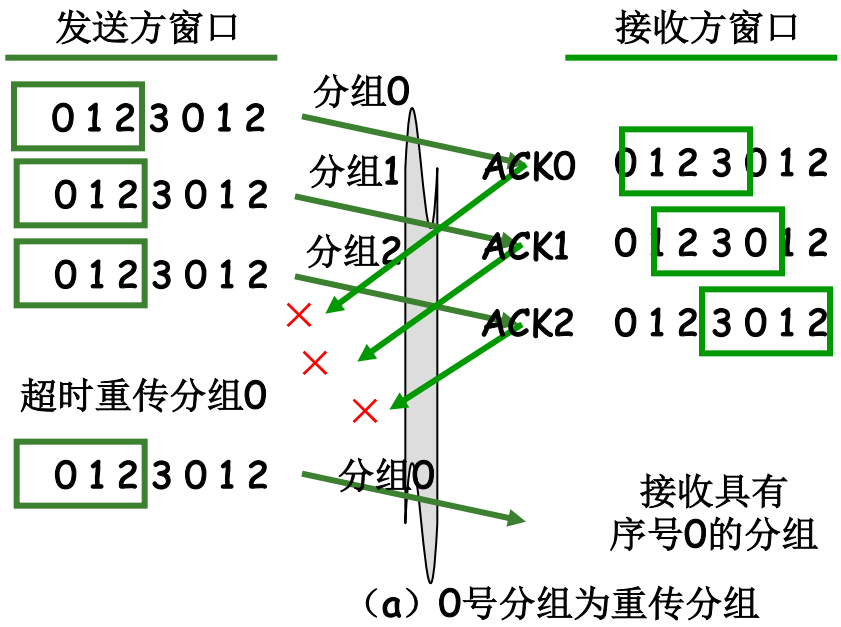
- 选择重传（SR）协议
- 为什么接受方会收到[recvbase-N，recvbase ﹣ 1]范围内的分组？并且必须给出确认？
  - 因为确认可能会丢失。假设接受方按序收到N个分组，向发送方发送确认后接受窗口向前移动N位。
  - 假设确认分组全部丢失，导致发送方重发。发送方最多只能发N个，因此接受方会收到[recvbase-N，recvbase ﹣ 1]范围内的分组
  - 接收方这时必须给出确认，否则发送方窗口无法向前移动
- 为什么接受方收到比recvbase-N更早的分组后不用发确认了？
  - 因为比recvbase-N更早的分组（如recvbase-N-1 ），发送方一定收到确认了。
  - 当接受窗口位于recvbase时，意味着接受方一定按序收到了从[recvbase-N，recvbase ﹣ 1]的分组
  - 这意味着发送方窗口一定到了recvbase-N。因此发送方一定收到了比recvbase-N更早的确认

3.4 可靠数据传输的原理



3.4 可靠数据传输的原理

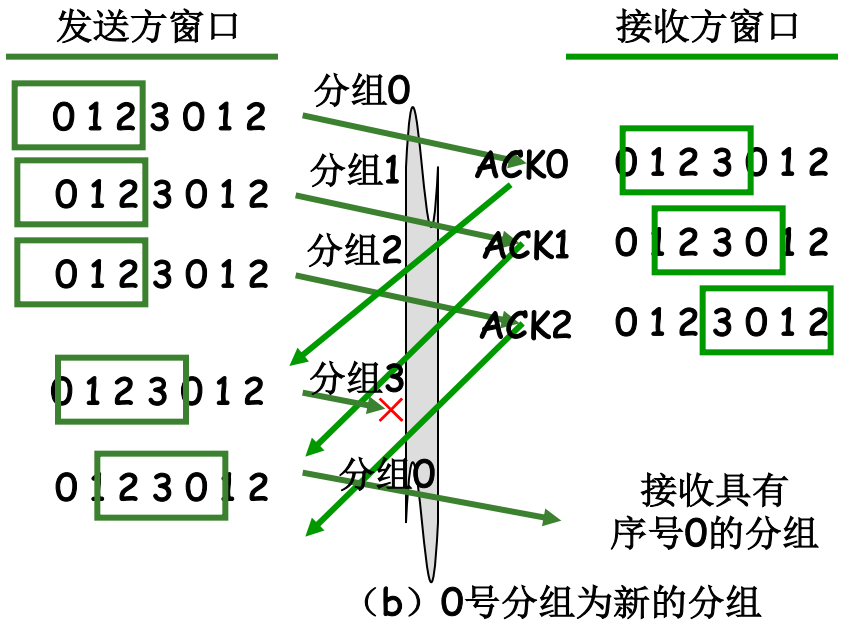
- 当序号范围有限时，发送窗口和接受窗口不同步会产生严重后果
- 假设序号采用2位二进制编码(k=2)，则序号空间为0, 1, 2, 3
- 假设窗口长度为3
- 接受方
  - 按序收到分组0，1，2
  - 接受窗口移动3位
- 发送方
  - 确认都没收到
  - 超时重发0



（a）0号分组为重传分组

3.4 可靠数据传输的原理

- 第二种情况
- 接受方
  - 按序收到分组0，1，2
  - 接受窗口移动3位
- 发送方
  - 收到ACK0，窗口移动1位，发送分组3，但分组3丢失
  - 收到ACK1，窗口移动1位，发送分组0
- 这是接受方在窗口太大时的两难：最后收到的分组0是新的分组还是重传的？（接受方无法判断是情况一还是二）
- 原因：窗口太大



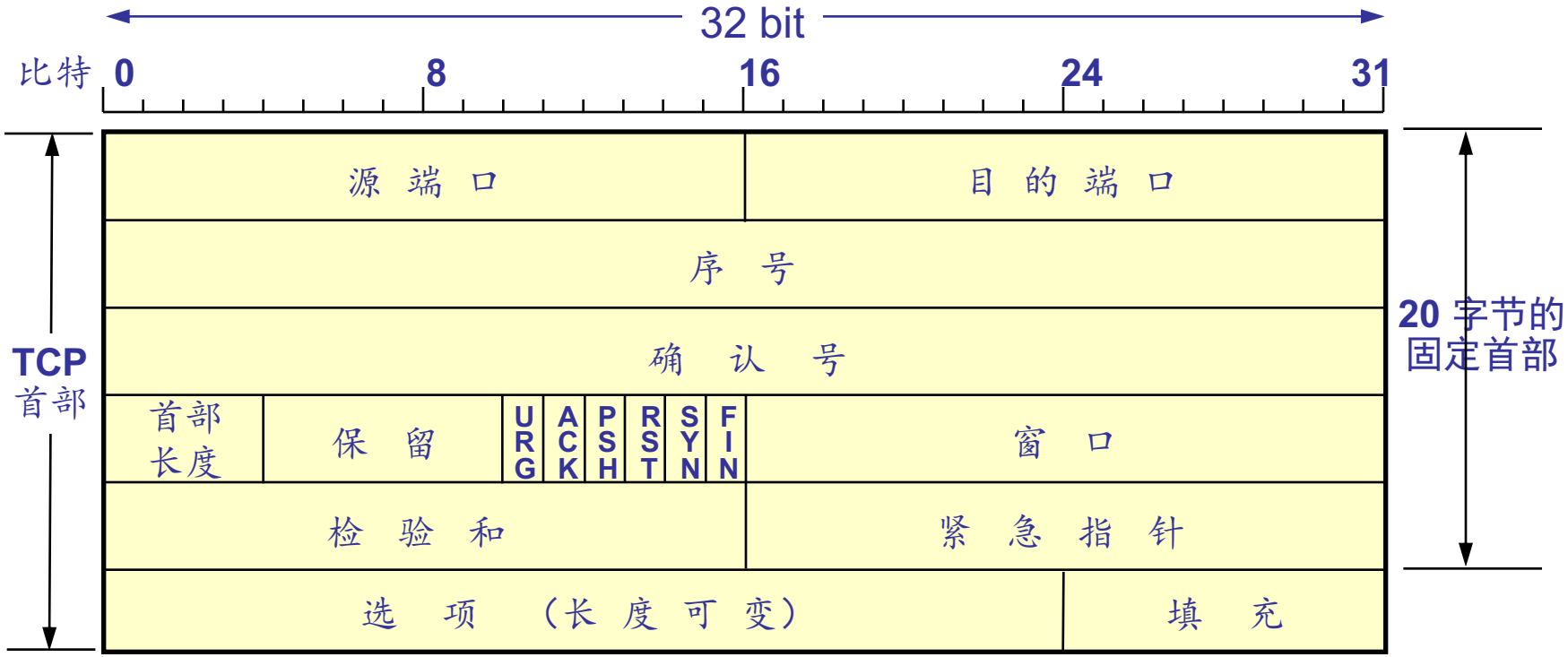
结论：$N \leq 2^{k-1}$

3.5 面向连接的传输：TCP

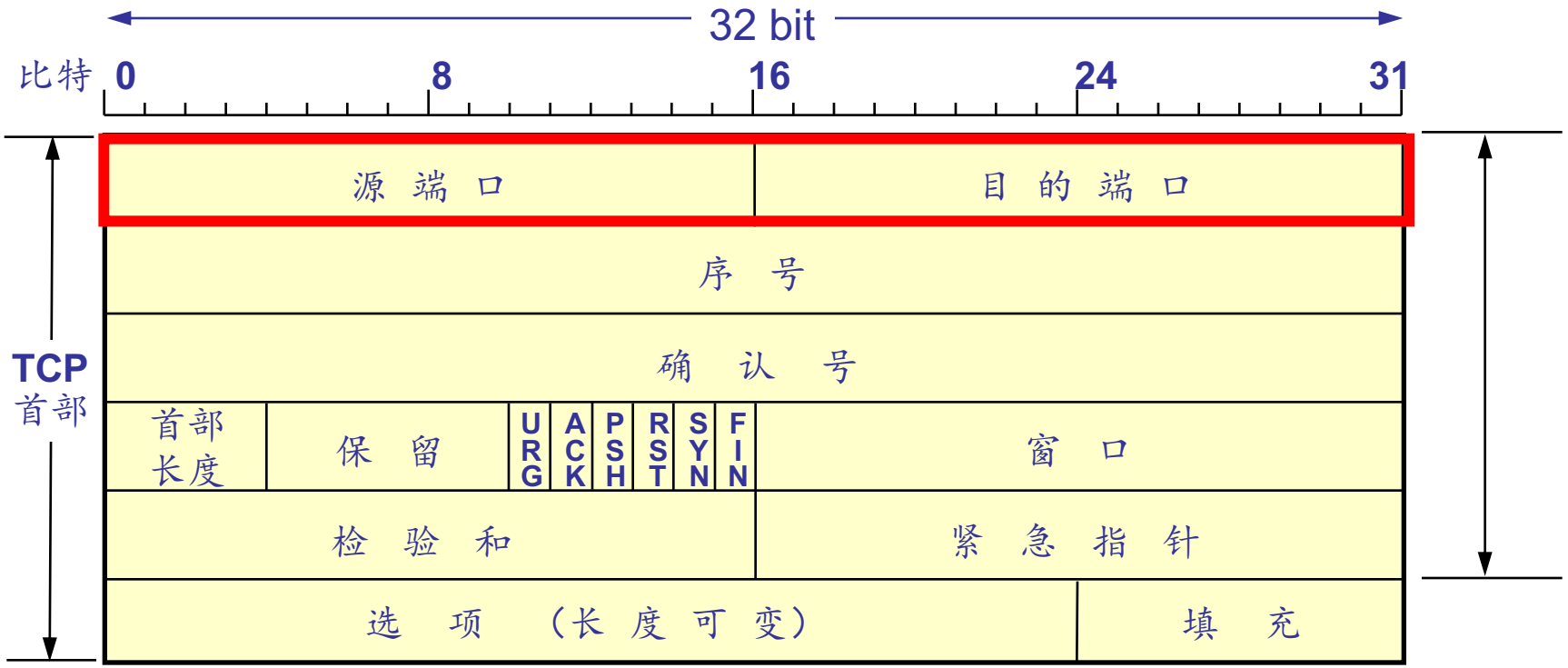
- **TCP概述—RFCs：793、1122、1323、2018、2581**
  - 面向连接
    - TCP连接仅存于端系统，中间路由器对此毫不知情
  - 全双工服务
    - 可双向同时传输数据
  - 点对点连接
    - 仅存在于两个端系统之间，无第三者“插足”
  - 三次握手
    - 建立连接，协商参数
  - 可靠的字节流
    - 最大报文段长MSS
  - 流量控制
  - 拥塞控制

3.5 面向连接的传输：TCP

- **TCP报文段首部结构**

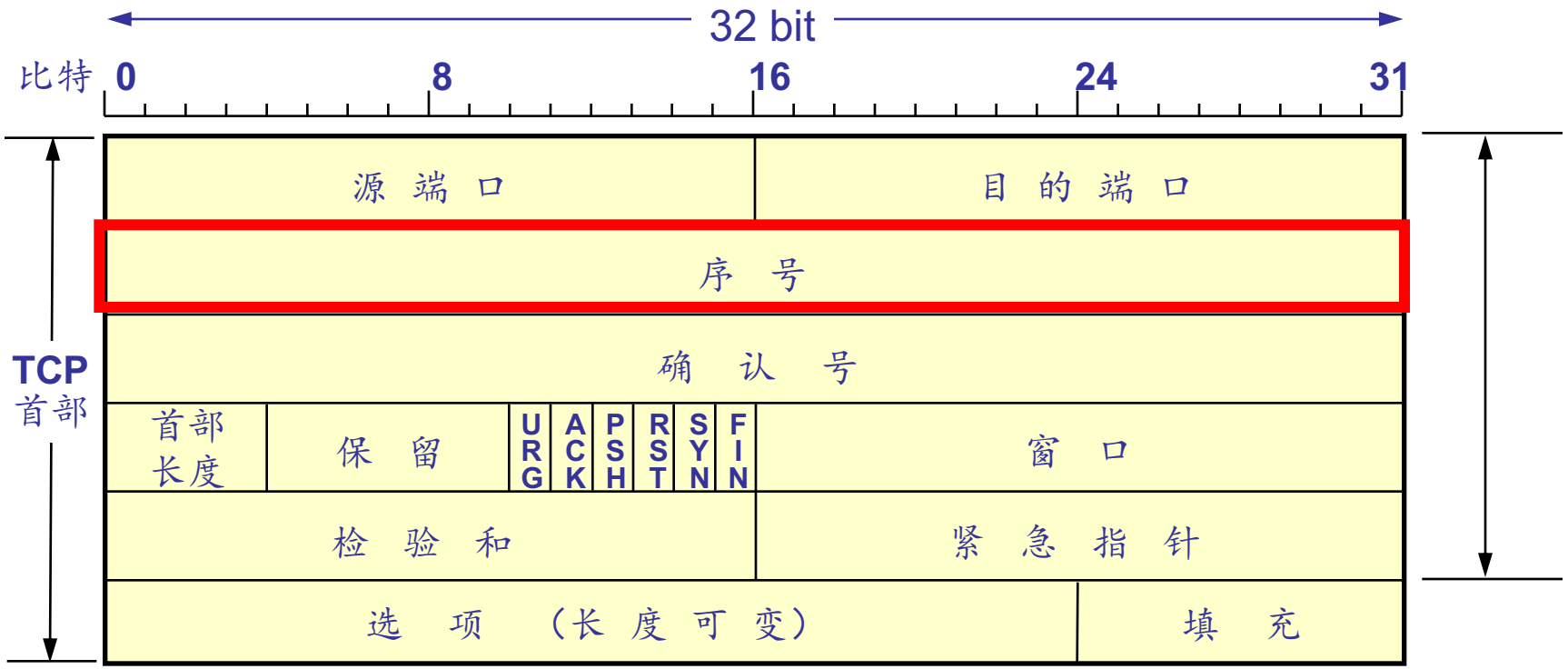


3.5 面向连接的传输：TCP



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。

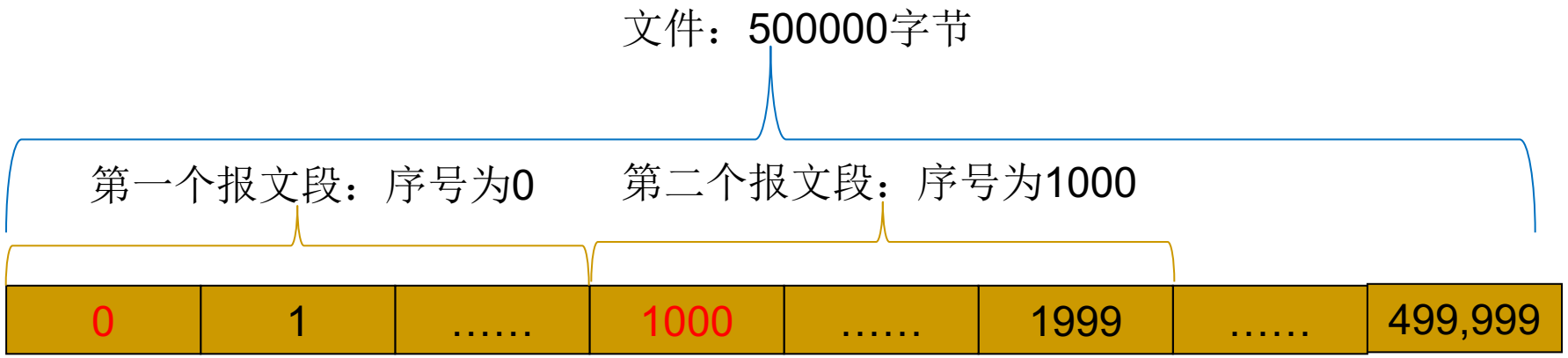
3.5 面向连接的传输：TCP



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在整个报文字节流中的序号。

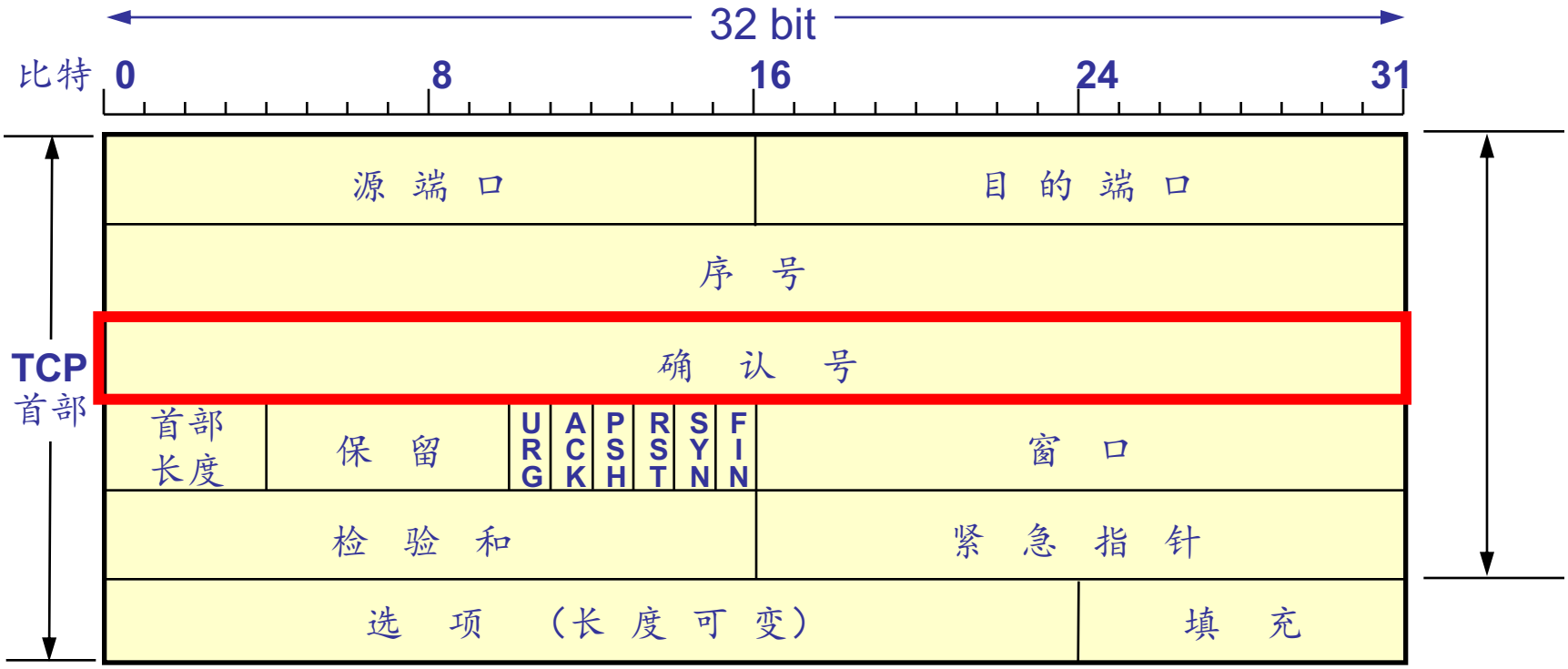
3.5 面向连接的传输：TCP

- TCP报文段序号
  - TCP把数据看成是无结构但有序的字节流
  - 字节流中的每个字节编上序号
  - 报文段序号=该报文段首字节的字节流编号



假设MSS为1000字节，该文件分500个报文段发送

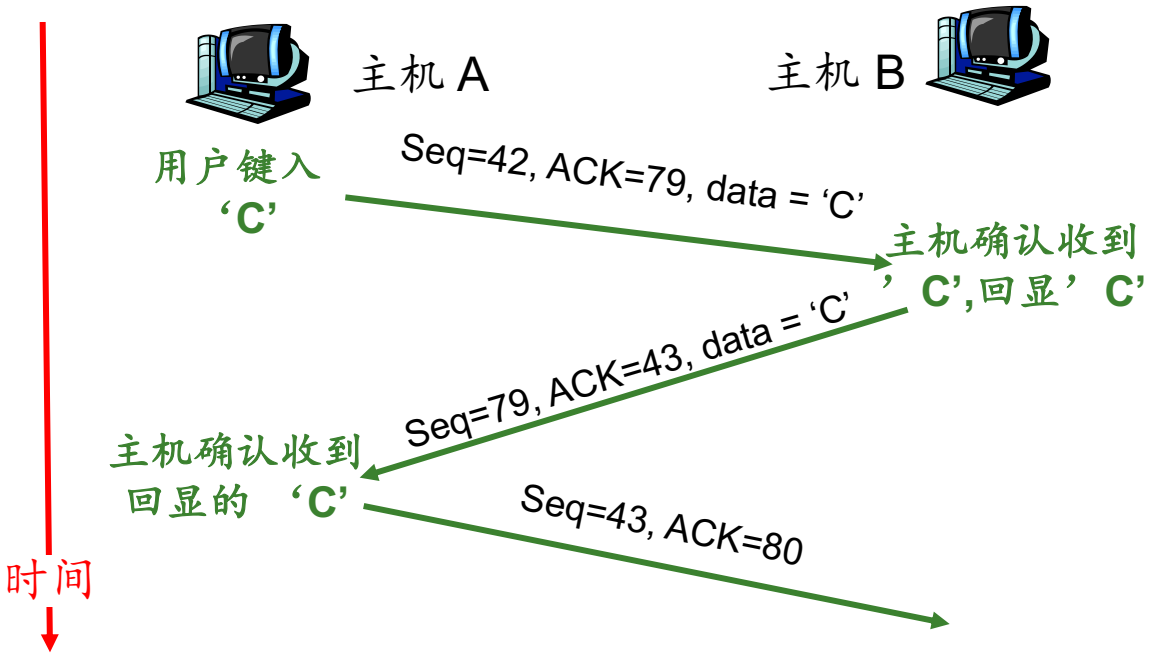
3.5 面向连接的传输：TCP



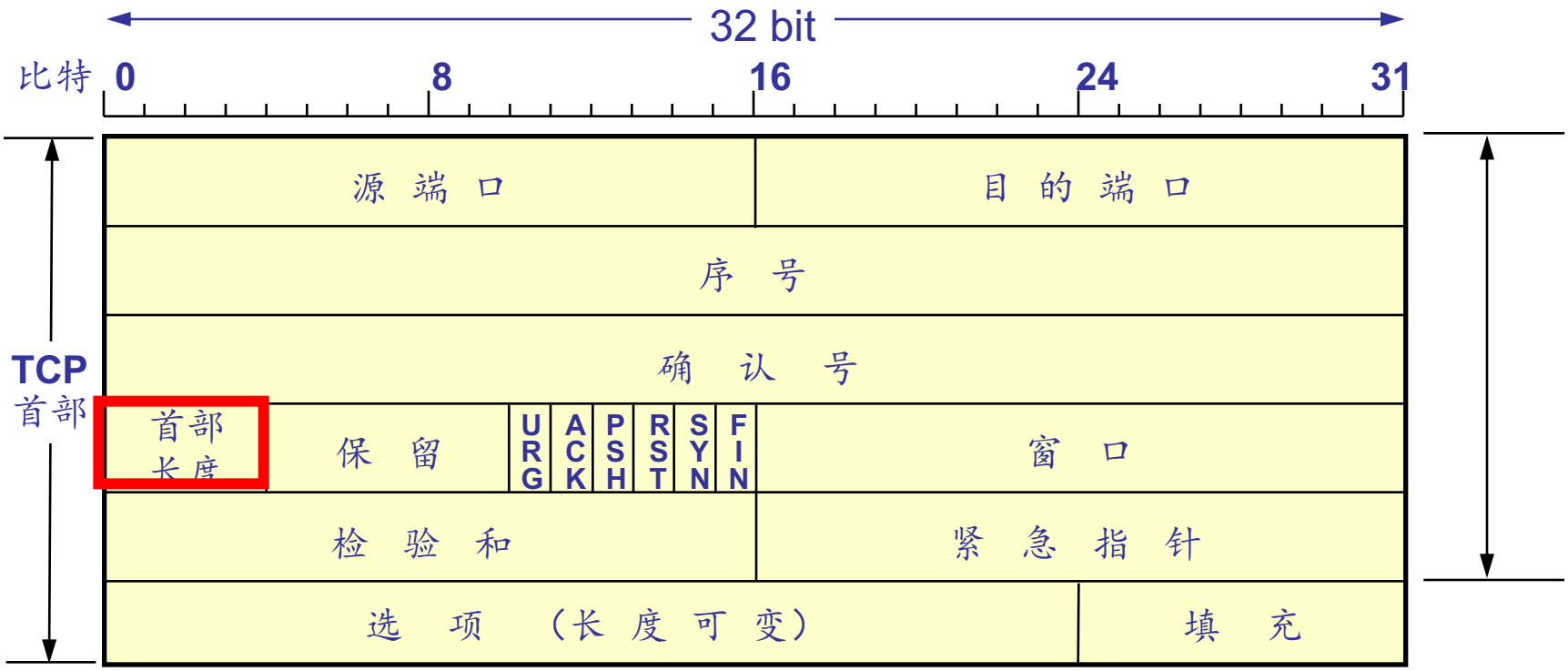
确认号字段——占 4 字节，是期望收到对方的下一个报文段的数据的第一个字节的序号。

3.5 面向连接的传输：TCP

- **TCP序列号和确认序列号**

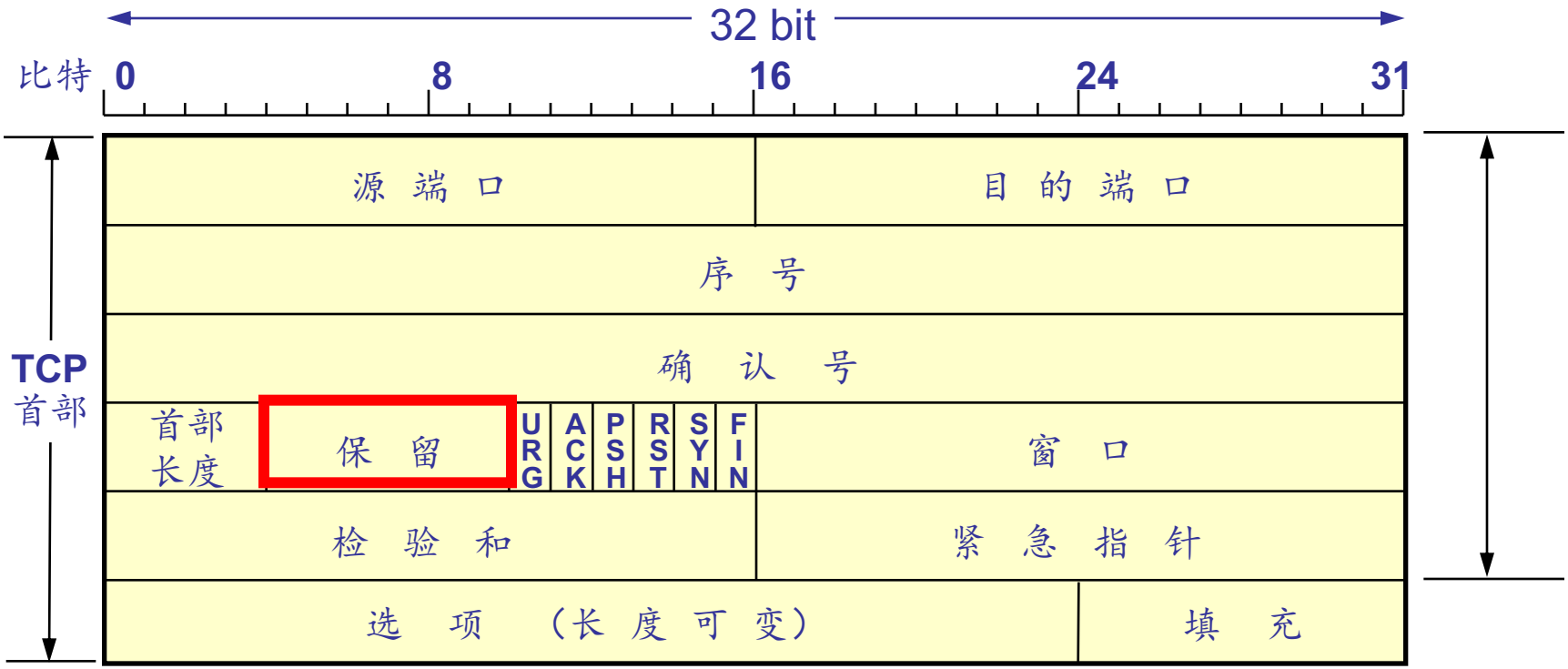


3.5 面向连接的传输：TCP



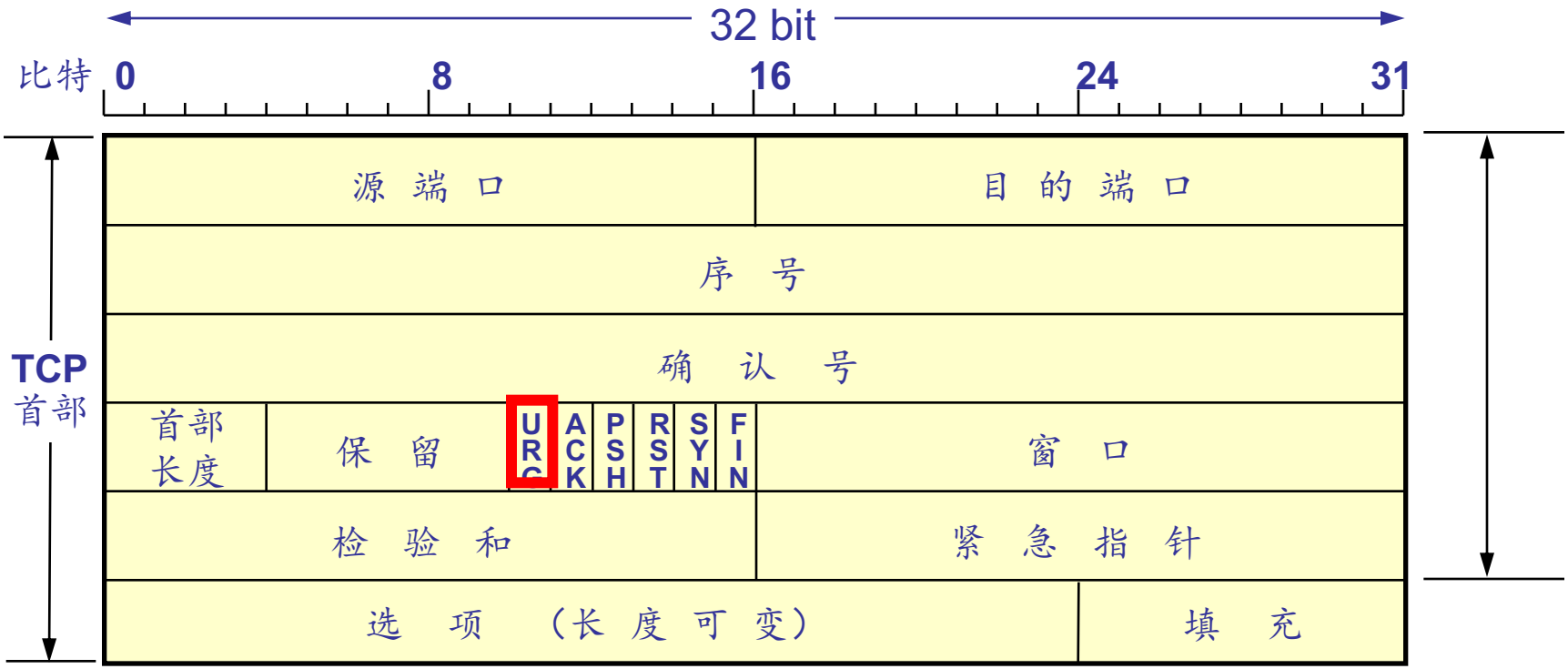
首部长度——占 4 bit，它指示意32bit为单位的TCP首部长度。

3.5 面向连接的传输：TCP



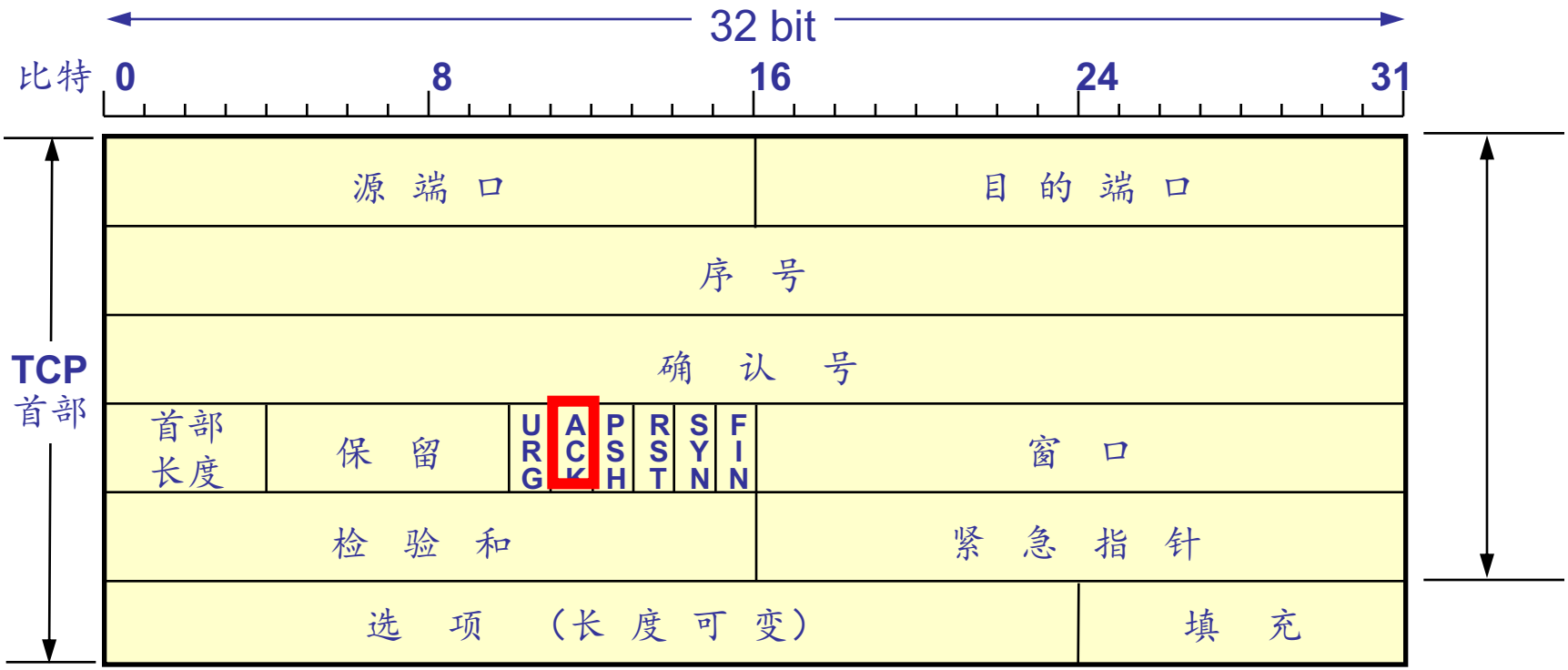
保留字段——占 6 bit，保留为今后使用，但目前应置为 0。

3.5 面向连接的传输：TCP



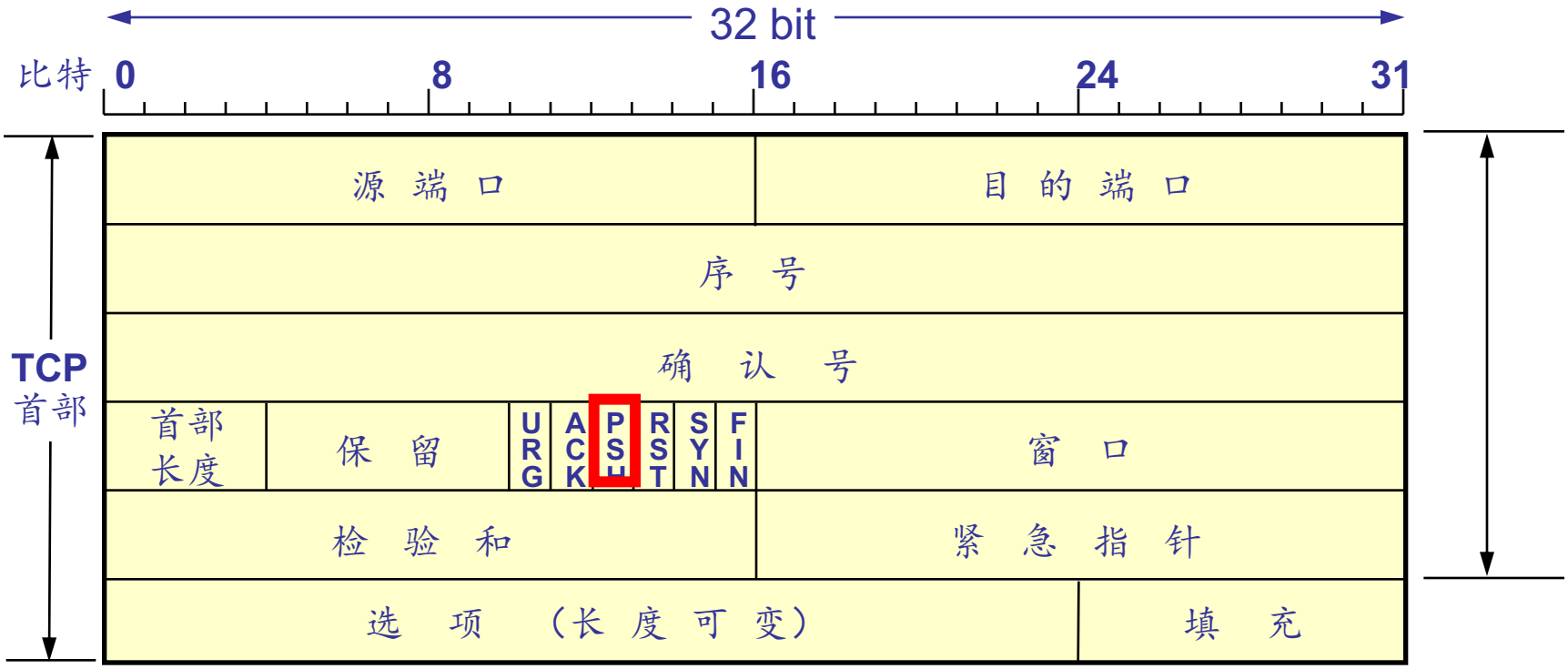
紧急比特 URG —— 当 URG = 1 时，表明紧急指针字段有效。它告诉系统此报文段中有紧急数据，应尽快传送。（一般不使用）

3.5 面向连接的传输：TCP



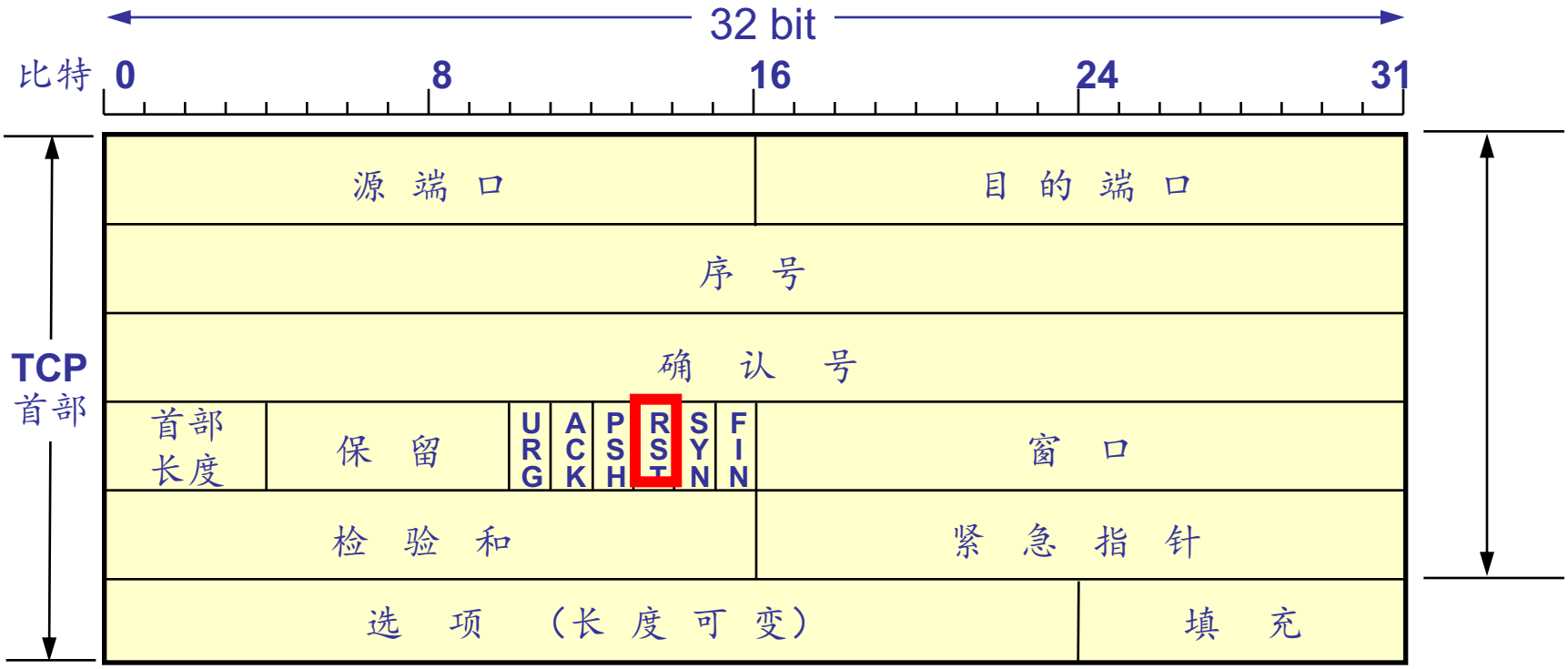
确认比特 ACK —— 只有当 ACK = 1 时确认号字段才有效。
当 ACK = 0 时，确认号无效。

3.5 面向连接的传输：TCP



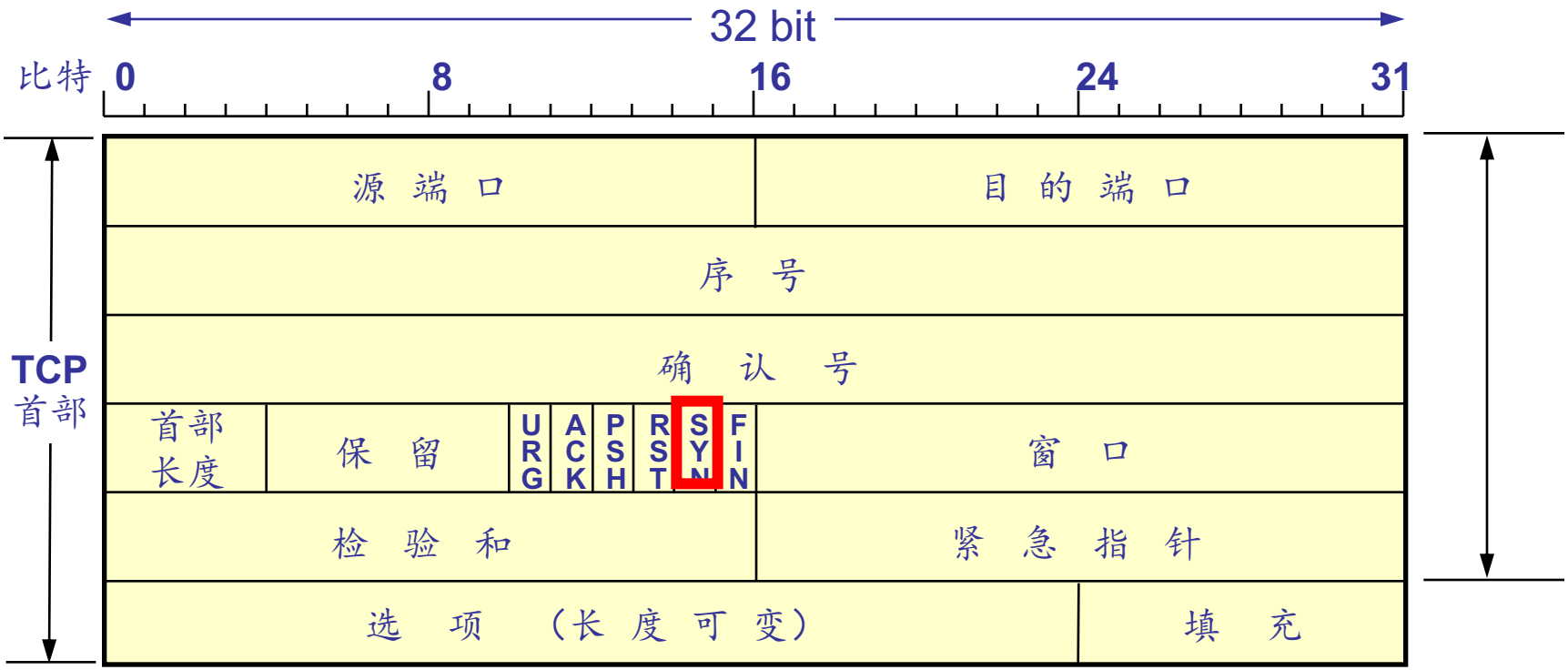
推送比特 PSH (PuSH) —— 接收 TCP 收到推送比特置 1 的报文段，就尽快地交付给接收应用进程，而不再等到整个缓存都填满了后再向上交付。

3.5 面向连接的传输：TCP



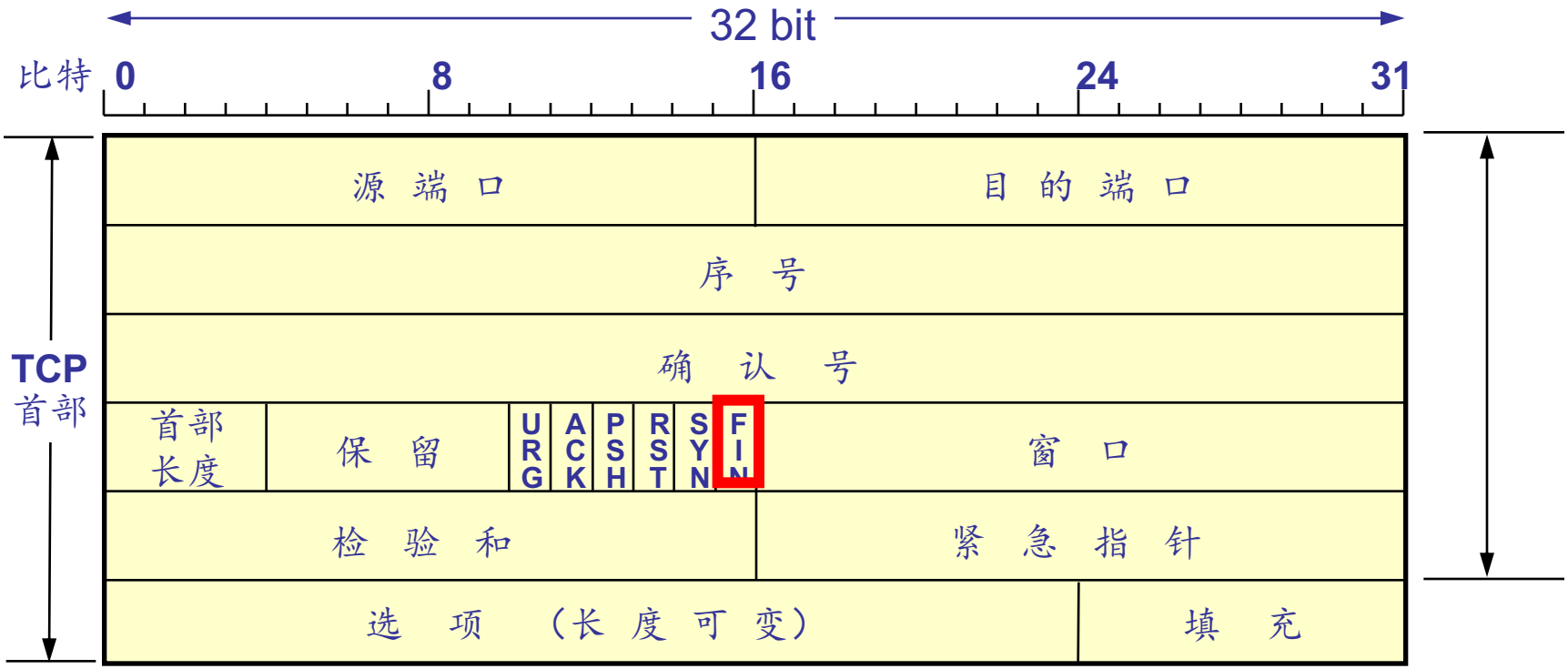
复位比特 RST (ReSeT) —— 当 RST = 1 时，表明 TCP 连接中出现严重差错（如由于主机崩溃或其他原因），必须释放连接，然后再重新建立运输连接。

3.5 面向连接的传输：TCP



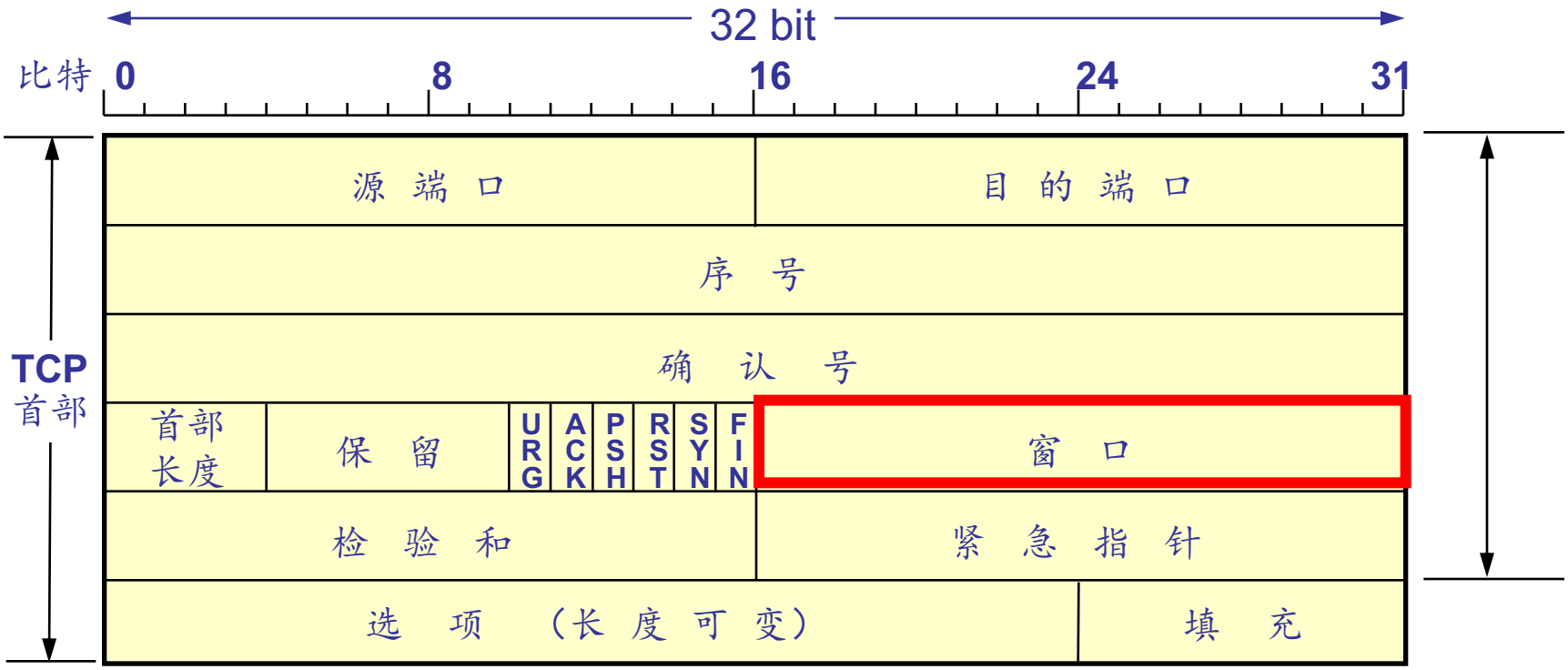
同步比特 SYN —— 同步比特 SYN 置为 1，就表示这是一个连接请求或连接接受报文。

3.5 面向连接的传输：TCP



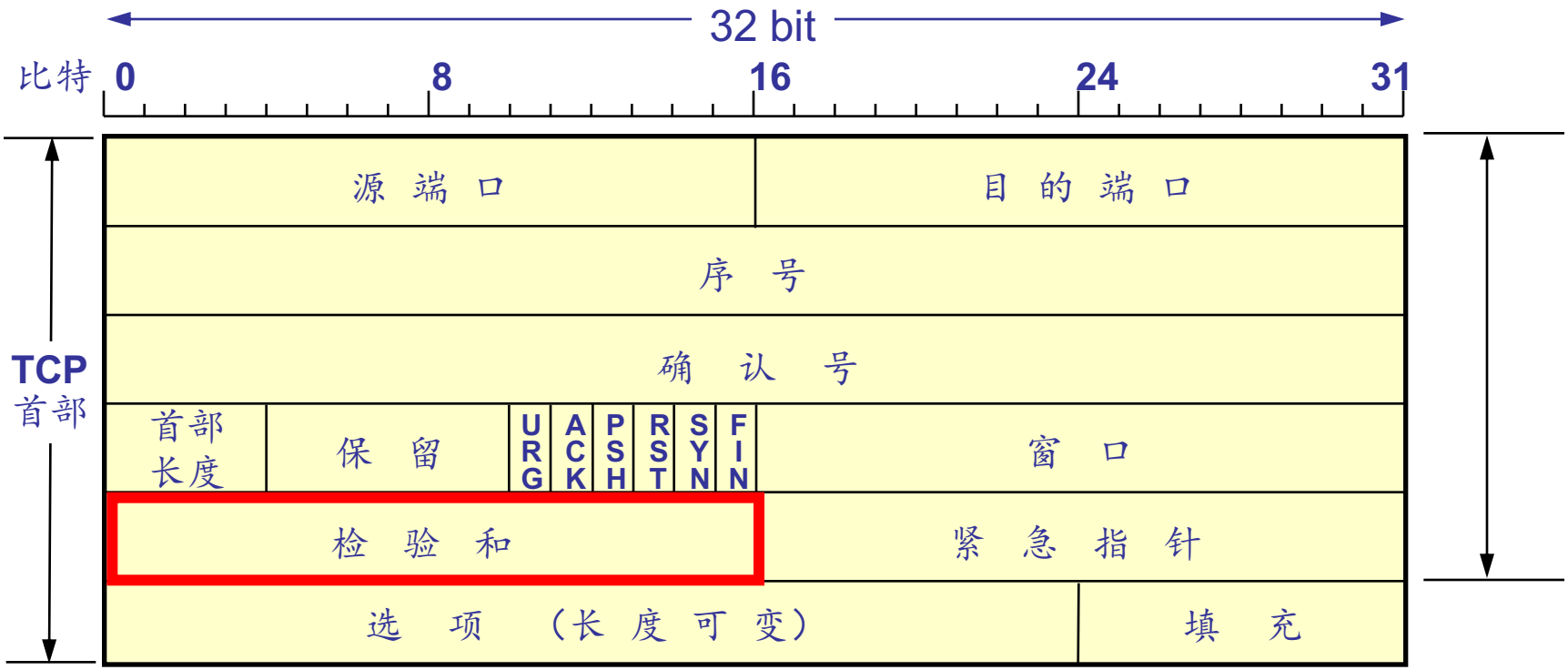
终止比特 FIN (FINal) —— 用来释放一个连接。当FIN = 1 时，表明此报文段的发送端的数据已发送完毕，并要求释放运输连接。

3.5 面向连接的传输：TCP



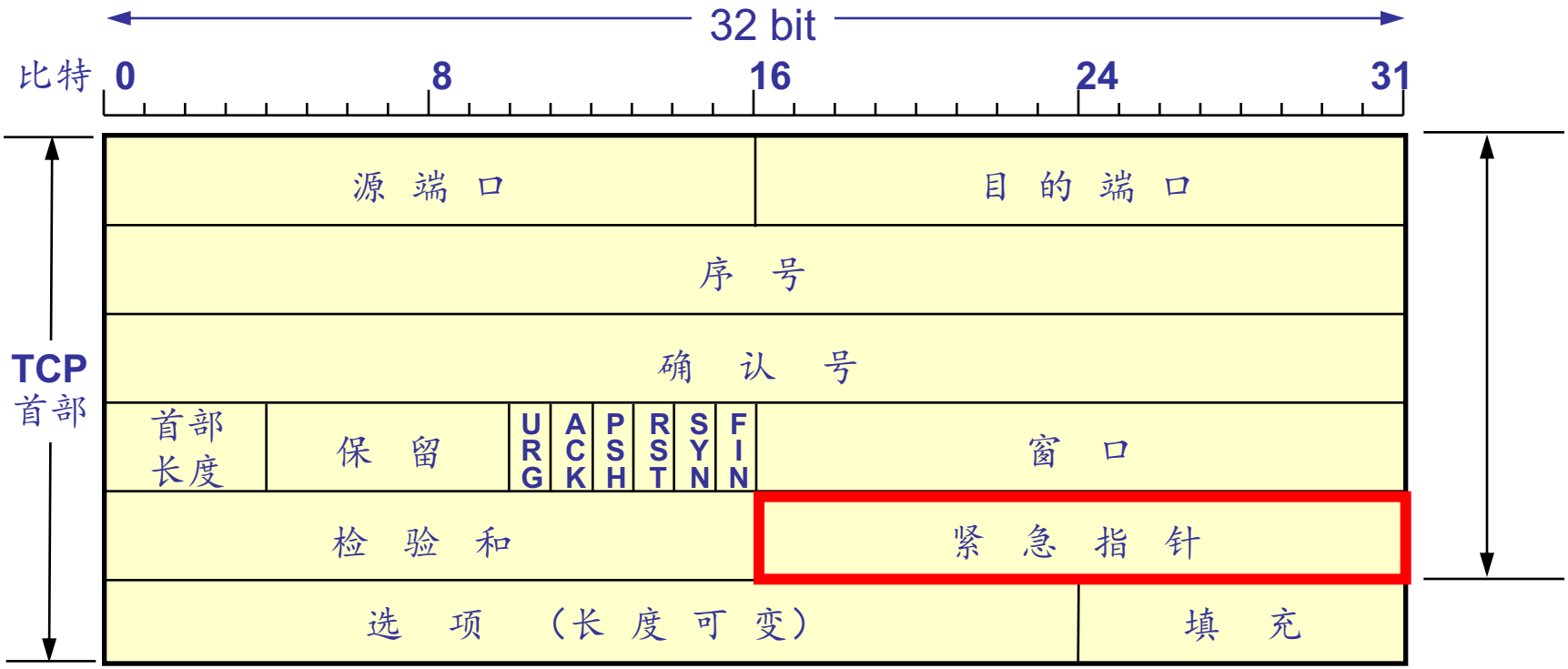
窗口字段 —— 占 2 字节。窗口字段用来控制对方发送的数据量，单位为字节。TCP 连接的一端根据设置的缓存空间大小确定自己的接收窗口大小，然后通知对方以确定对方的发送窗口的上限。

3.5 面向连接的传输：TCP



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。

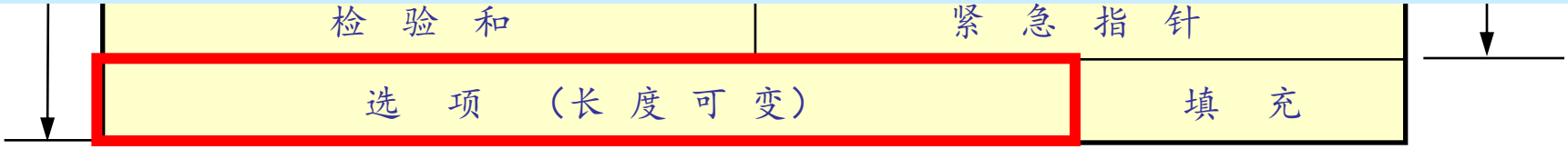
3.5 面向连接的传输：TCP



紧急指针字段 —— 占 16 bit。紧急指针指出在本报文段中的紧急数据的最后一个字节的序号。

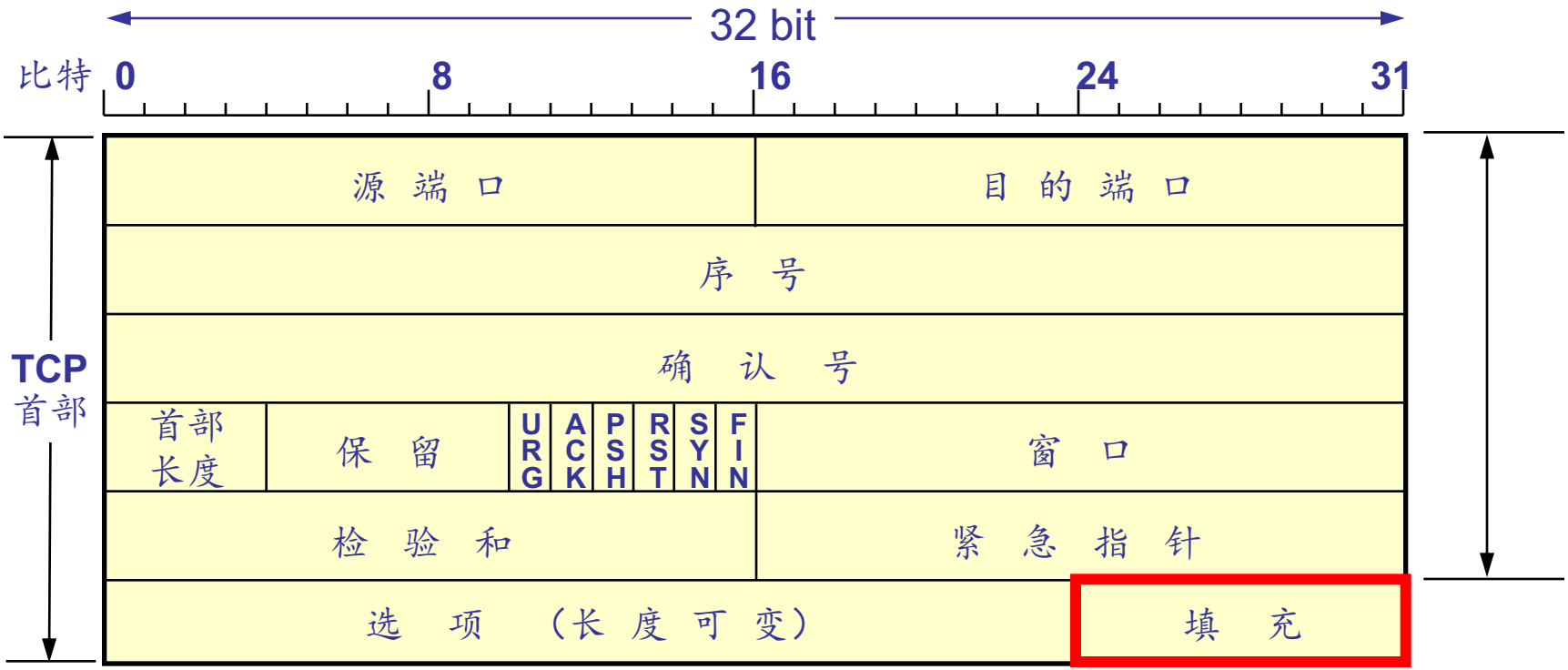
3.5 面向连接的传输：TCP

MSS 是 TCP 报文段中的数据字段的最大长度。
数据字段加上 TCP 首部
才等于整个的 TCP 报文段。



选项字段 —— 长度可变。TCP 只规定了一种选项，即最大报文段长度 MSS (Maximum Segment Size)。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。”

3.5 面向连接的传输：TCP



填充字段 —— 这是为了使整个首部长度是 4 字节的整数倍。

3.5 面向连接的传输：TCP

- **TCP超时的设置**
  - 如何设置TCP的超时
    - 应该大于RTT
      - 但 RTT是变化的
    - 太短:
      - 造成不必要的重传
    - 太长:
      - 对丢包反应太慢

3.5 面向连接的传输：TCP

- 如何估算 RTT
  - 样本RTT(SampleRTT): 对报文段被发出到收到该报文段的确认之间的时间进行测量
    - 忽略重传
  - 样本RTT会有波动，要使得估算RTT更平滑，需要将最近几次的测量进行平均，而非仅仅采用最近一次的SampleRTT

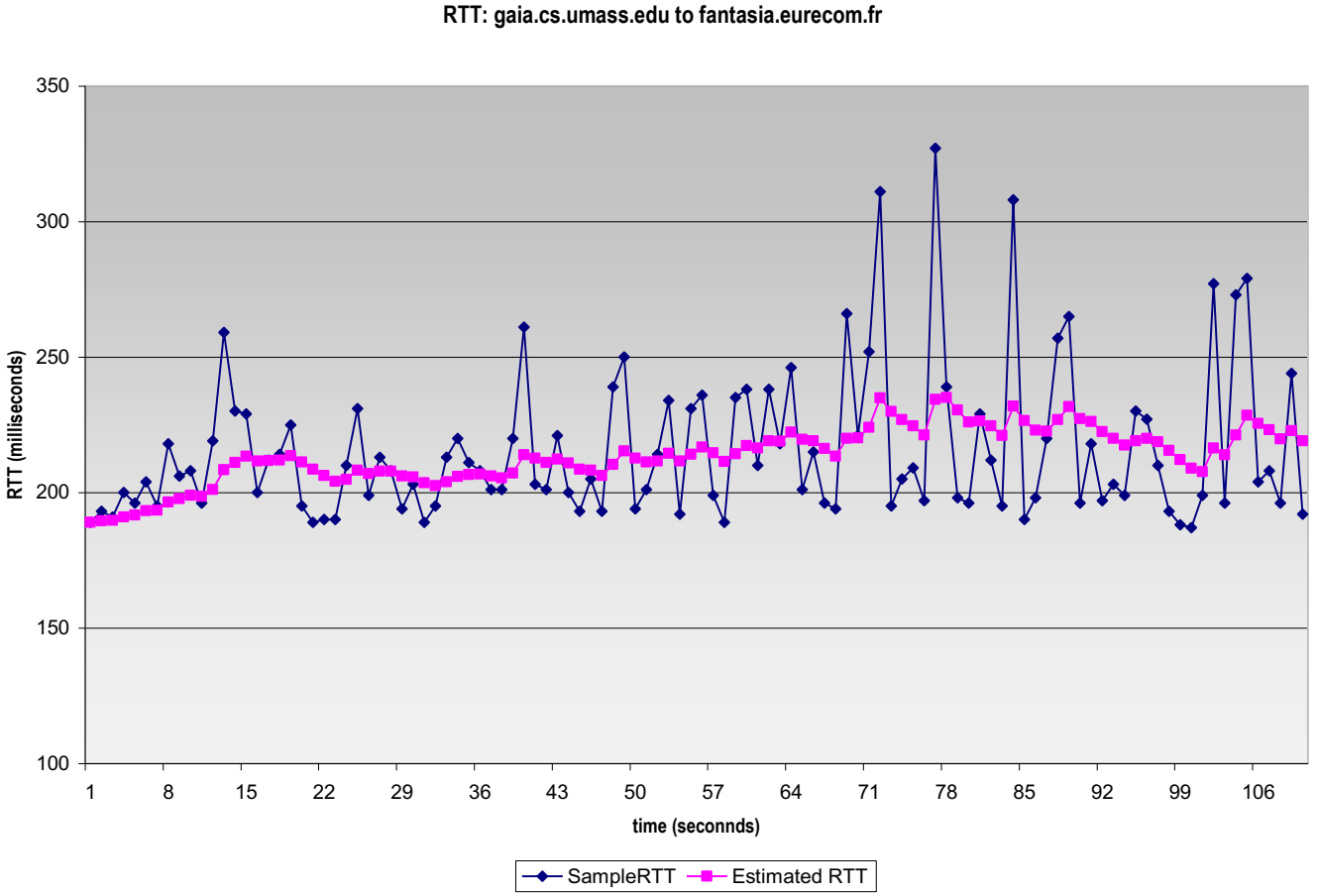
$$EstimatedRTT = (1-\alpha)*EstimatedRTT + \alpha*SampleRTT$$

参考值: $\alpha = 0.125$

第一次计算时：EstimatedRTT=SampleRTT

3.5 面向连接的传输：TCP

- RTT估计的一个例子



3.5 面向连接的传输：TCP

- 考虑RTT的波动，估计EstimatedRTT与SampleRTT的偏差

DevRTT = (1-β)*DevRTT +
β*|SampleRTT-EstimatedRTT|

(参考值, β = 0.25)
注意，第一次计算时，DevRTT=0.5*SampleRTT

TCP中的超时间隔为

TimeoutInterval = EstimatedRTT + 4*DevRTT

3.5 面向连接的传输：TCP

- **可靠的TCP数据传输**
  - IP协议是不可靠的
  - TCP采用了3.4节阐述的数据可靠传输的方法
  - 特别之处
    - TCP编号采用按字节编号，而非按报文段编号
    - TCP仅采用唯一的超时定时器

3.5 面向连接的传输：TCP

- TCP在IP的不可靠服务基础上提供可靠数据传输服务
- 流水线方式发送报文段
- 累积确认：只确认最后一个正确按序到达的报文段
- TCP使用一个重传定时器（最早未确认的报文段）
- 从上面几点看，TCP采用的机制很像GBN
- 重传由以下情况触发：
  - 超时
  - 重复的ACK
- 先考虑TCP发送方的简化描述：
  - 忽略重复的ACK
  - 忽略流量控制和拥塞控制

3.5 面向连接的传输：TCP

- TCP发送方事件

从应用程序接收数据

- 将数据封装入报文段中，每个报文段都包含一个序号
- 序号是该报文段第一个数据字节的字节流编号
- 启动定时器
- 超时间隔：TimeOutInterval

超时：

- 重传认为超时的报文段（最早未确认的报文段）
- 重启定时器

收到Ack：

- 如果是对以前的未确认报文段的确认(Ack>Sendbase)
  - 更新SendBase
  - 如果当前有未被确认的报文段，TCP还要重启定时器

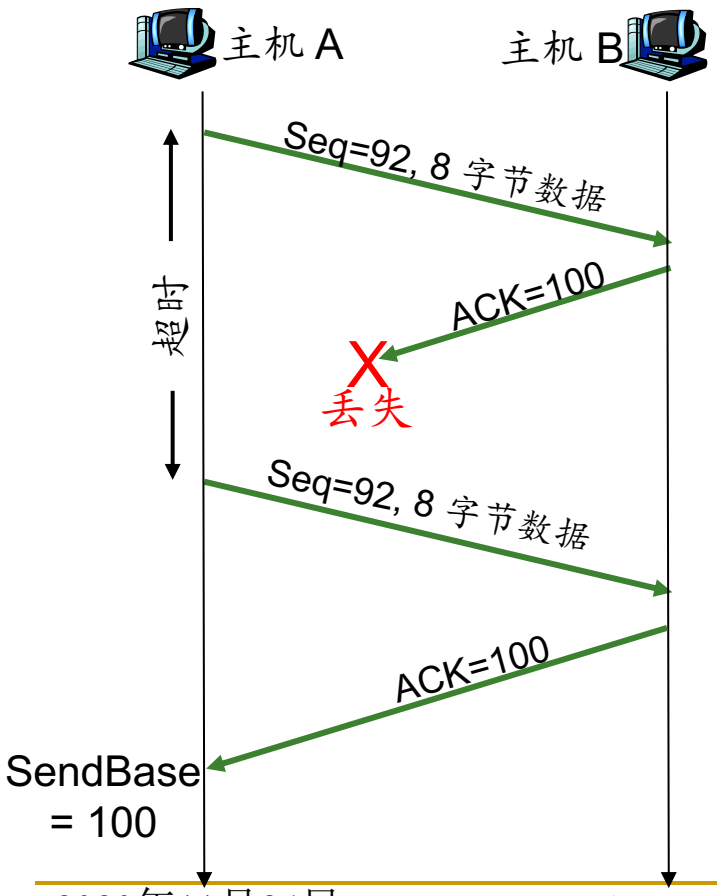
3.5 面向连接的传输：TCP

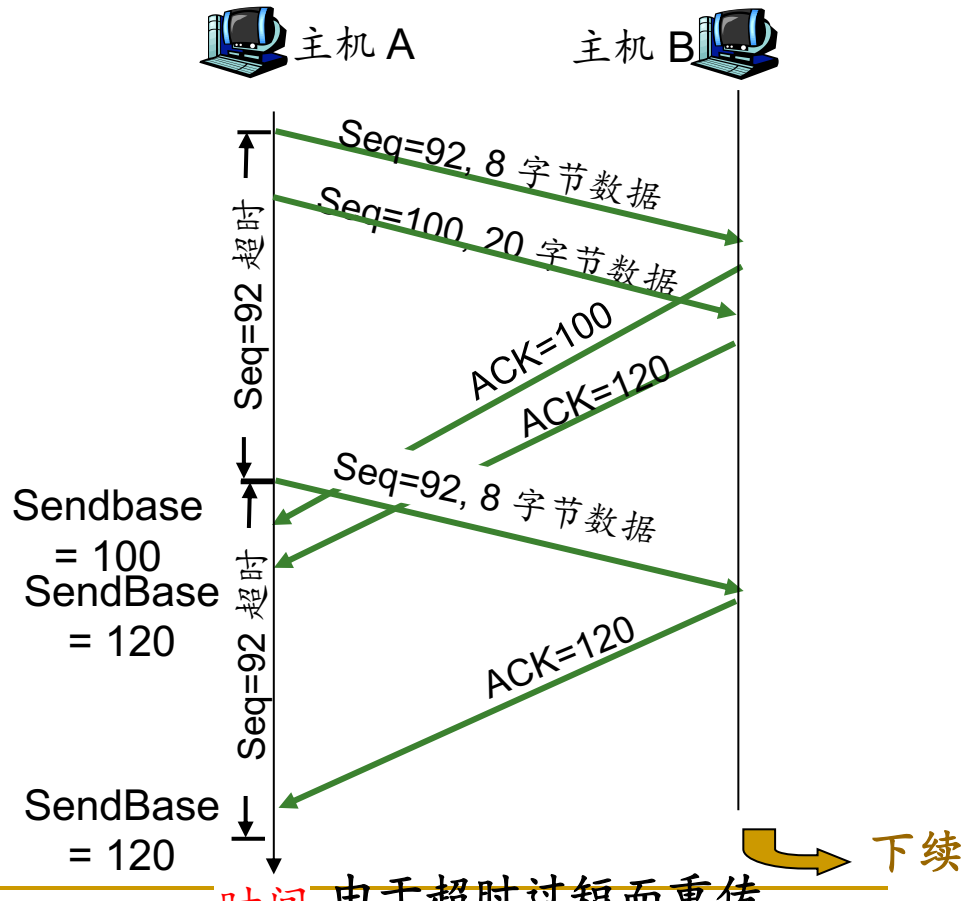
```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
loop (forever) {
   switch(event)
   event: data received from application above
        create TCP segment with sequence number NextSeqNum
        if (timer currently not running)
              start timer
        pass segment to IP
        NextSeqNum = NextSeqNum + length(data)
   event: timer timeout
        retransmit not-yet-acknowledged segment with
              smallest sequence number （和GBN不同）
        start timer
   event: ACK received, with ACK field value of y
        if (y > SendBase) {
             SendBase = y //窗口移动到y,因为是累积确认
            if (there are currently not-yet-acknowledged segments)
                  start timer
        }
} /* end of loop forever */
```

- 重传是对超前的确认报文段中，每个报文段都包含一个序号
- 序号是该报文段第一个数据字节的字节流编号
- 启动定时器
- 超时间隔：TimeOutInterval

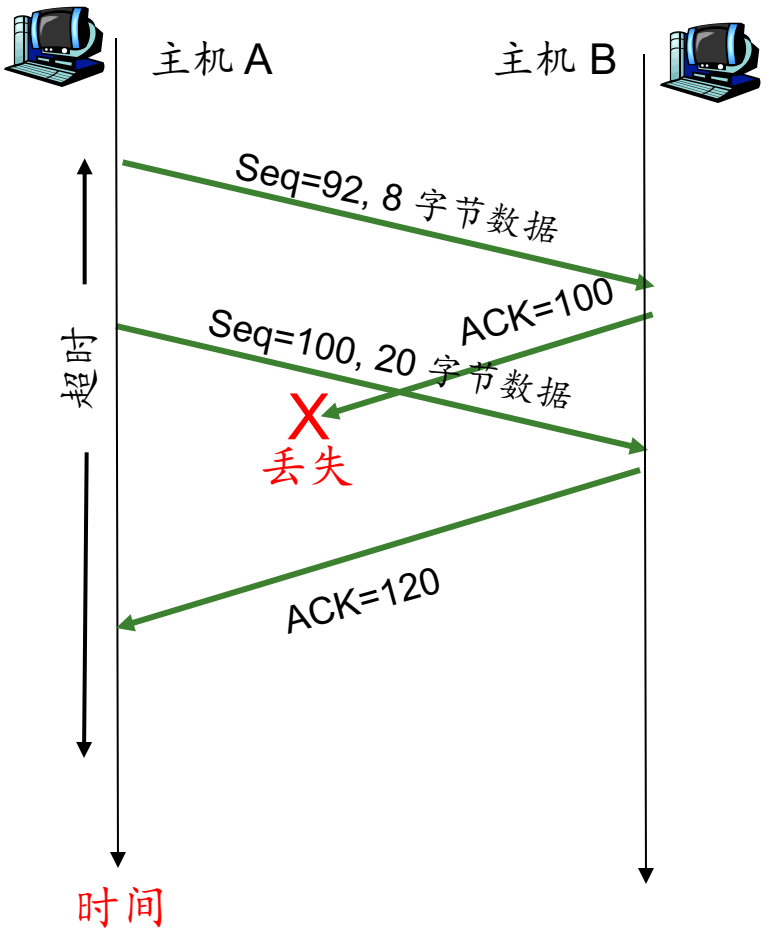
3.5 面向连接的传输：TCP

- **TCP的几种重传情况**





3.5 面向连接的传输：TCP



累积确认避免了第一个报文的重传

3.5 面向连接的传输：TCP

- 超时间隔加倍
  - 每一次TCP超时重传均将下一次超时间隔设为先前值的两倍
  - 超时间隔由EstimatedRTT和DevRTT决定，每当发生下列事件之一是重新计算超时间隔
    - 收到上层应用的数据
    - 收到对未确认数据的ACK

3.5 面向连接的传输：TCP

- **快速重传**
  - 超时周期往往太长
    - 增加重发丢失分组的延时
  - 通过重复的ACK检测丢失报文段
    - 发送方常要连续发送大量报文段
    - 如果一个报文段丢失，会引起很多连续的重复ACK.
  - 如果发送收到一个数据的3个重复ACK，它会认为确认数据之后的报文段丢失
    - 快速重传：在超时到来之前重传报文段

3.5 面向连接的传输：TCP

- **快速重传的算法**

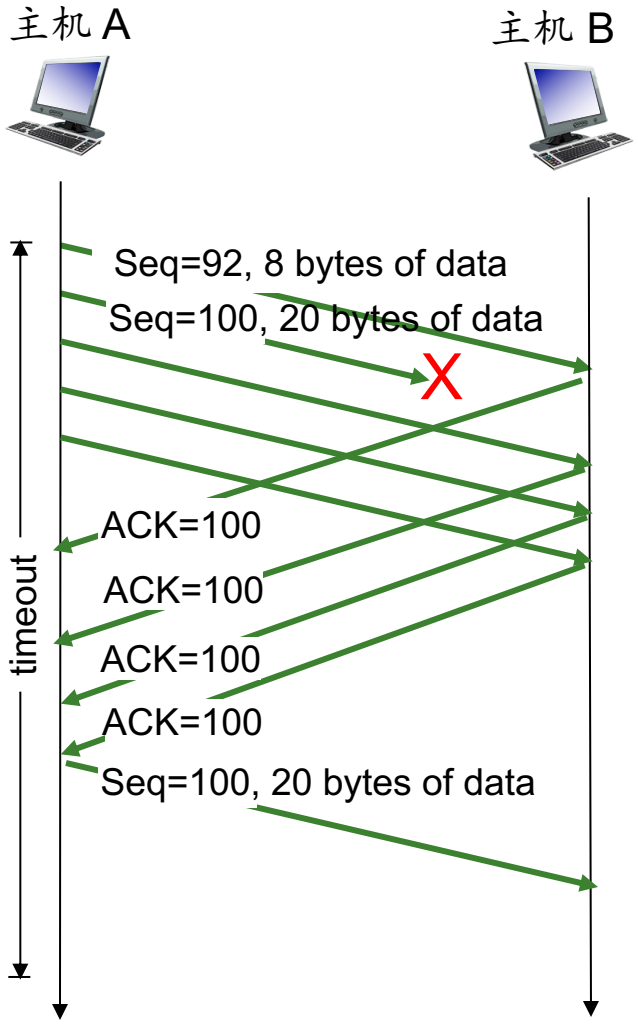
```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase = y
                if (there are currently not-yet-acknowledged segments)
                    start timer
              }
            else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                 }
```

重复的ACK报文

快速重传

3.5 面向连接的传输：TCP

- 快速重传示例



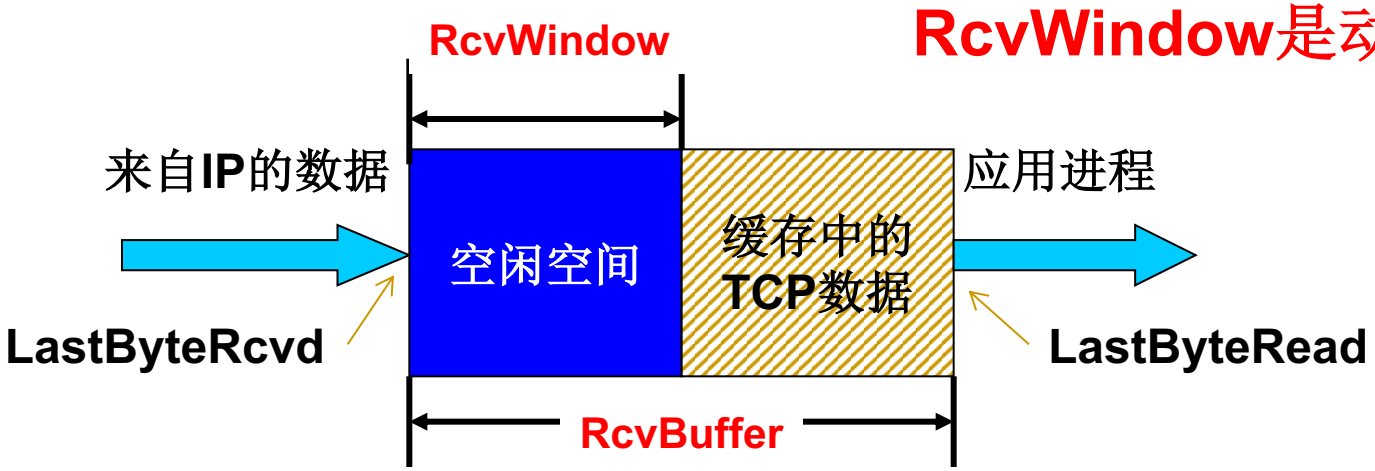
是Go-Back-N还是选择重传

- TCP积累是确认式的，只用一个定时器，很像GBN
- 但有区别：
  - 很多TCP实现缓存失序的报文段。
  - GBN在报文段n超时时，会重发从n开始所有未确认的报文段。而TCP只会重传报文段n。甚至如果在报文段n超时前收到了对报文段n+1的确认，TCP连报文段n都不会重传。
  - TCP还有快速重传机制。

3.5 面向连接的传输：TCP

- **TCP流量控制**
  - 背景
    - TCP接收方有一个缓存，所有上交的数据全部缓存在里面
    - 应用进程从缓冲区中读取数据可能很慢
  - 目标
    - 发送方不会由于传得太多太快而使得接收方缓存溢出
  - 手段
    - 接收方在反馈时，将缓冲区剩余空间的大小填充在报文段首部的窗口字段中，通知发送方

3.5 面向连接的传输：TCP



LastByteRead：接受主机上的应用进程从缓存中读出数据流最后一个字节

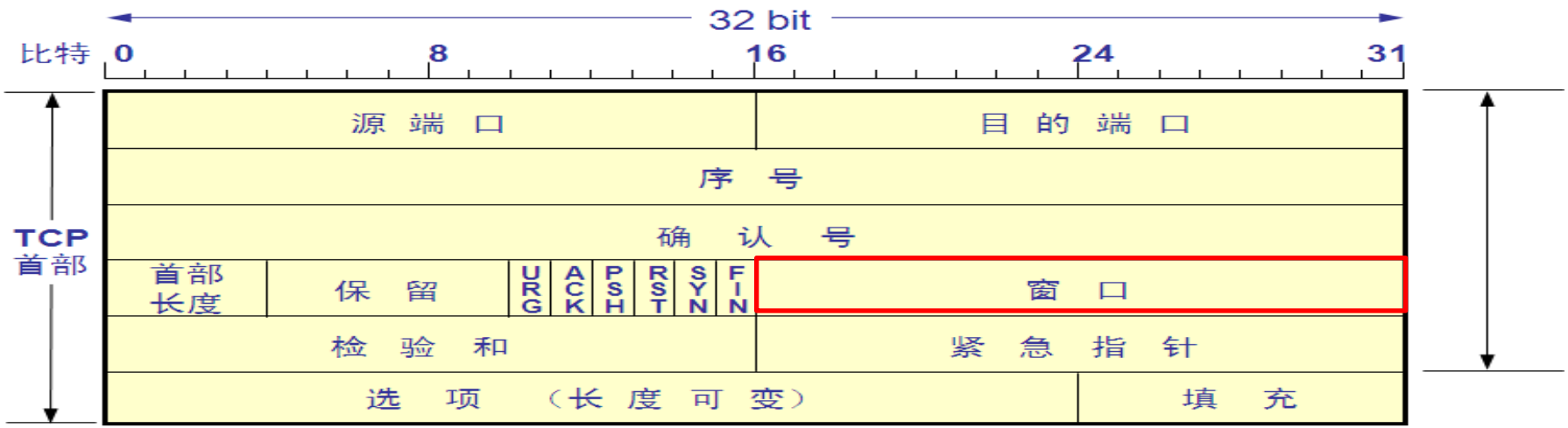
LastByteRcvd：从网络层（IP层）到达的并且已放入接受缓存中的数据流的最后一个字节

为了不使接受缓冲区溢出，必须满足：

接收方： $LastByteRcvd - LastByteRead \leq RcvBuffer$

$RcvWindows = RcvBuffer - [LastByteRcvd - LastByteRead]$

3.5 面向连接的传输：TCP



TCP连接如何使用RcvWindow进行流量控制？

接受方：把当前的RcvWindow值放入它发给发送方的报文段的窗口字段中以通知发送方自己在该连接的缓冲还有多少空间

发送方：LastByteSend：发送主机上发送的最后一个字节序号

LastByteAcked：发送主机上被确认的最后一个字节序号

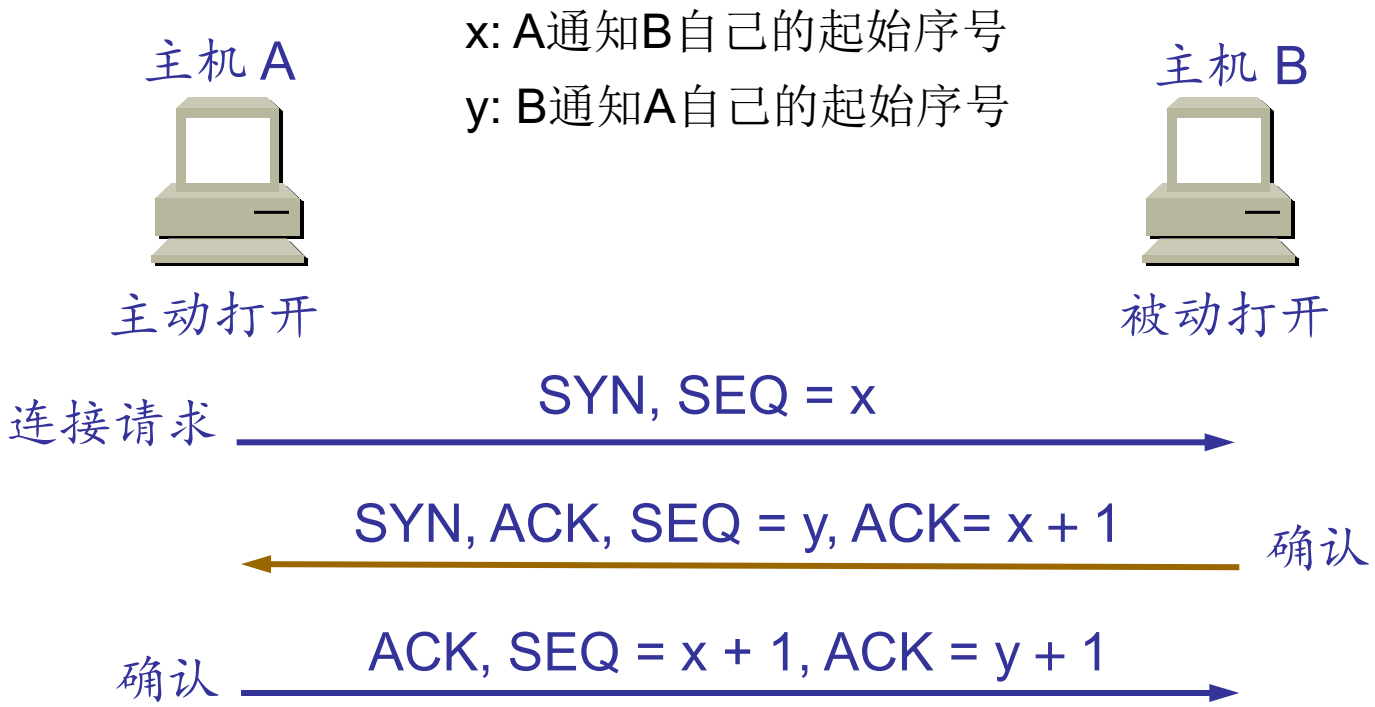
发送方必须保证：LastByteSent – LastByteAcked ≤ RcvWindow

3.5 面向连接的传输：TCP

- 一种特殊的情况
  - 接收方通知发送方RcvWindow为0，且接收方无任何数据传送给发送方
  - 当接受方的缓冲区被应用进程清空后，接收方没有办法主动发送报文段通知发送方（假设接受方没有数据要发送，而发送方因为得知接收方传递接受窗口为0而被阻塞）
  - 解决办法：当接收端的接收窗口为0时，发送方持续向接受方发送只有一个字节数据的报文段，目的是试探

3.5 面向连接的传输：TCP

- TCP连接的建立



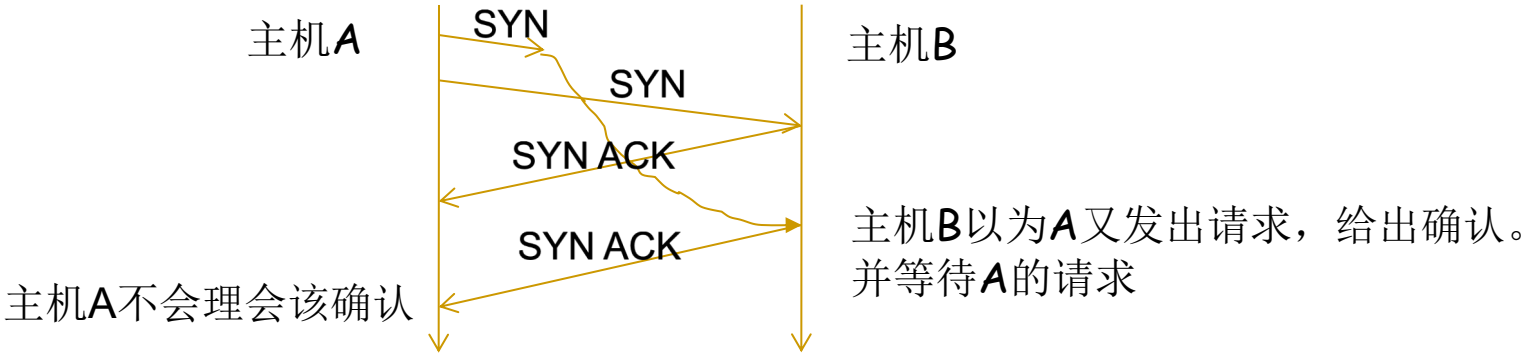
3.5 面向连接的传输：TCP

■ 为什么需要三次握手

假设只需要二次握手，考虑这种情况：

1：主机A发出的请求报文段在某些网络节点滞留时间太长，主机A由于超时重发连接请求，B收到重发的连接请求后给出同意连接的确认，主机A收到B的确认建立连接。数据传输完毕释放连接。

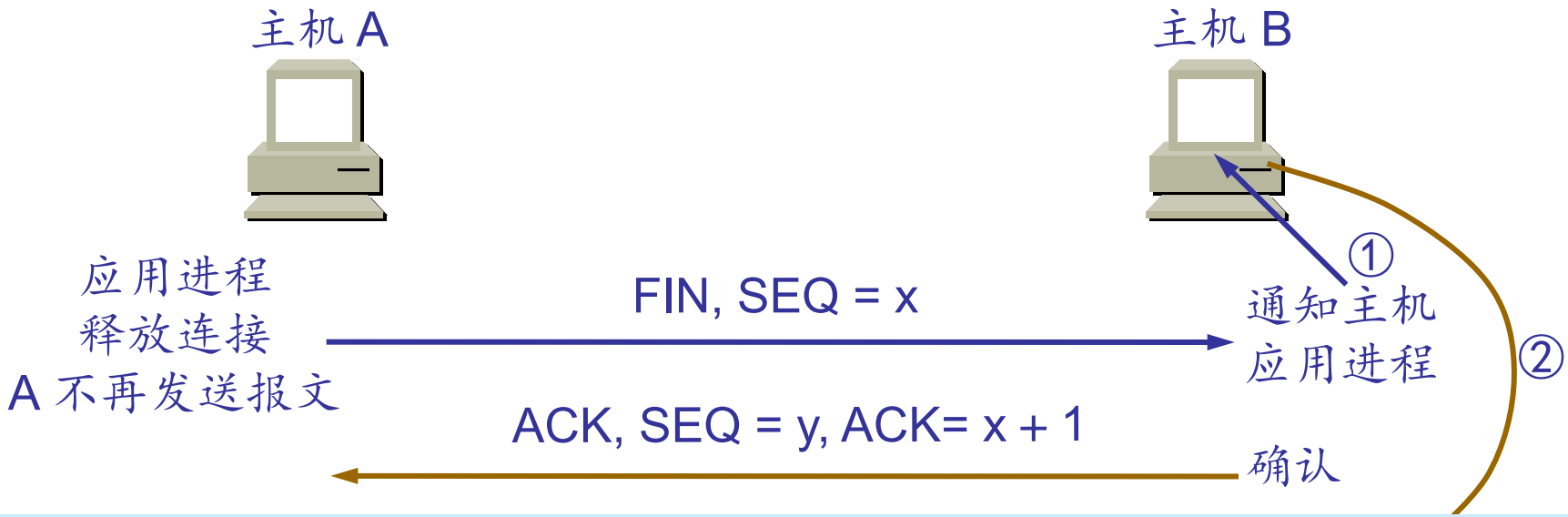
2：这时第一个请求才到达B，主机B收到该失效的请求后，误以为A又发出请求，于是向主机A发出确认，同意建立连接。主机A则不会理睬该确认。主机B则苦等A的数据。三次握手就可以防止这种情况的发生。（主机A不会对主机B的确认发出确认，连接就建立不起来）



3.5 面向连接的传输：TCP

- TCP SYN洪泛攻击
  - TCP服务器为了响应一个SYN连接请求报文，需要初始化连接变量及分配缓存（需要消耗服务器的资源），然后发送SYNACK报文进行响应，并等待客户端的ACK报文段。
  - 如果客户机不给出第三次ACK响应来完成第三次握手的最后一步，服务器在等待一段时间后（通常为一分钟）服务器将终止该半开连接并释放资源。
  - 三次握手机制为TCP SYN洪泛攻击搭建了舞台
  - 攻击者大量发送TCP SYN报文段，而不发送三次握手的第三步。
  - 为了加大攻击力度，从多个源主机发起攻击，称为DDOS（Distributed Dos），分布式拒绝服务攻击
  - 服务器不断地为这些半开连接分配资源，直到资源耗尽。

至此，整个连接已经全部释放。



从 A 到 B 的连接就释放了，连接处于半关闭状态。

相当于 A 向 B 说：

“我已经没有数据要发送了。

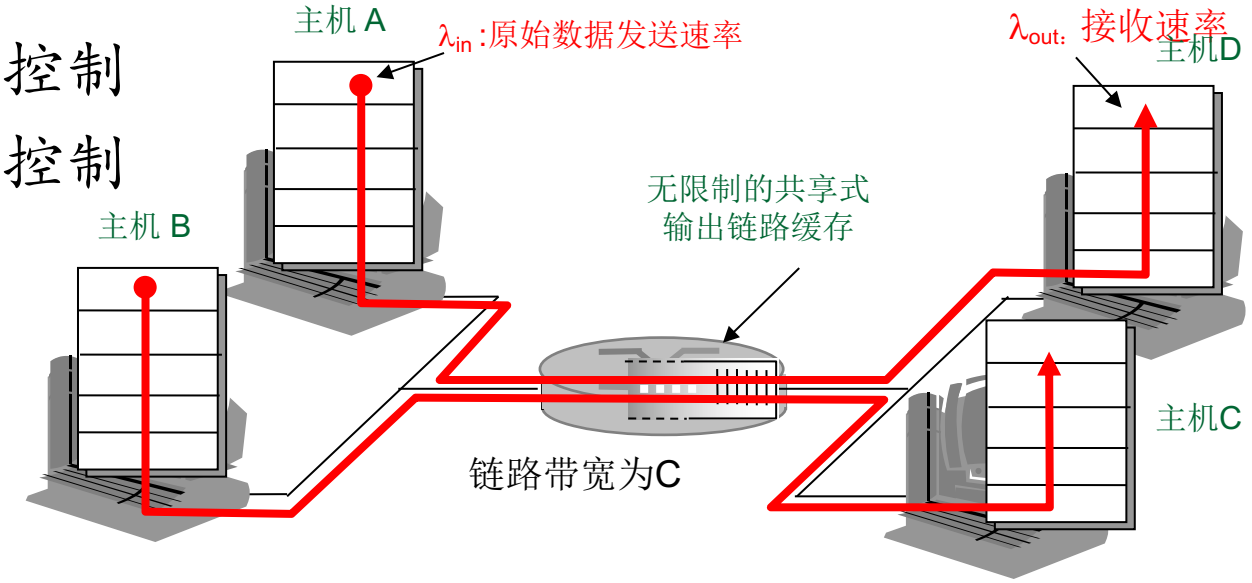
但你如果还发送数据，我仍接收。”

3.6 拥塞控制原理

- **拥塞的基本知识**
  - 非正式定义:“过多的源发送了过多的数据，超出了网络的处理能力”
  - 不同于流量控制!
    - 流量控制是为了匹配发送方和接收方的速度，只发生在发送方和接收方之间
    - 拥塞控制：网络拥塞时，所有发送方都要抑制发送速度
  - 现象:
    - 丢包 (路由器缓冲区溢出)
    - 延时长 (在路由器缓冲区排队)

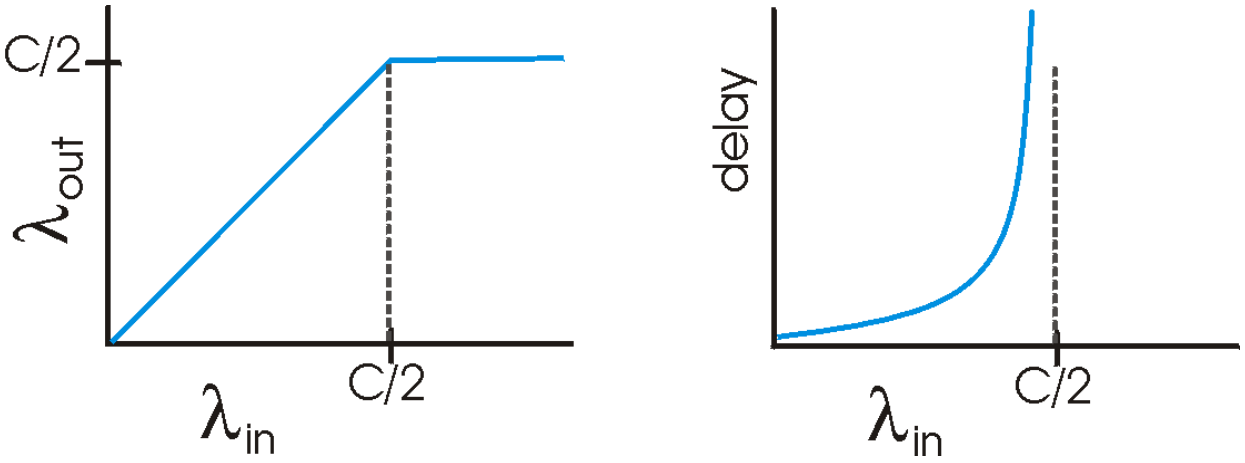
3.6 拥塞控制原理

- **情境1**
  - 两个发送方，两个接受方
  - 一个具有无限大缓存的路由器
  - 没有重传
  - 没有流量控制
  - 没有拥塞控制



3.6 拥塞控制原理

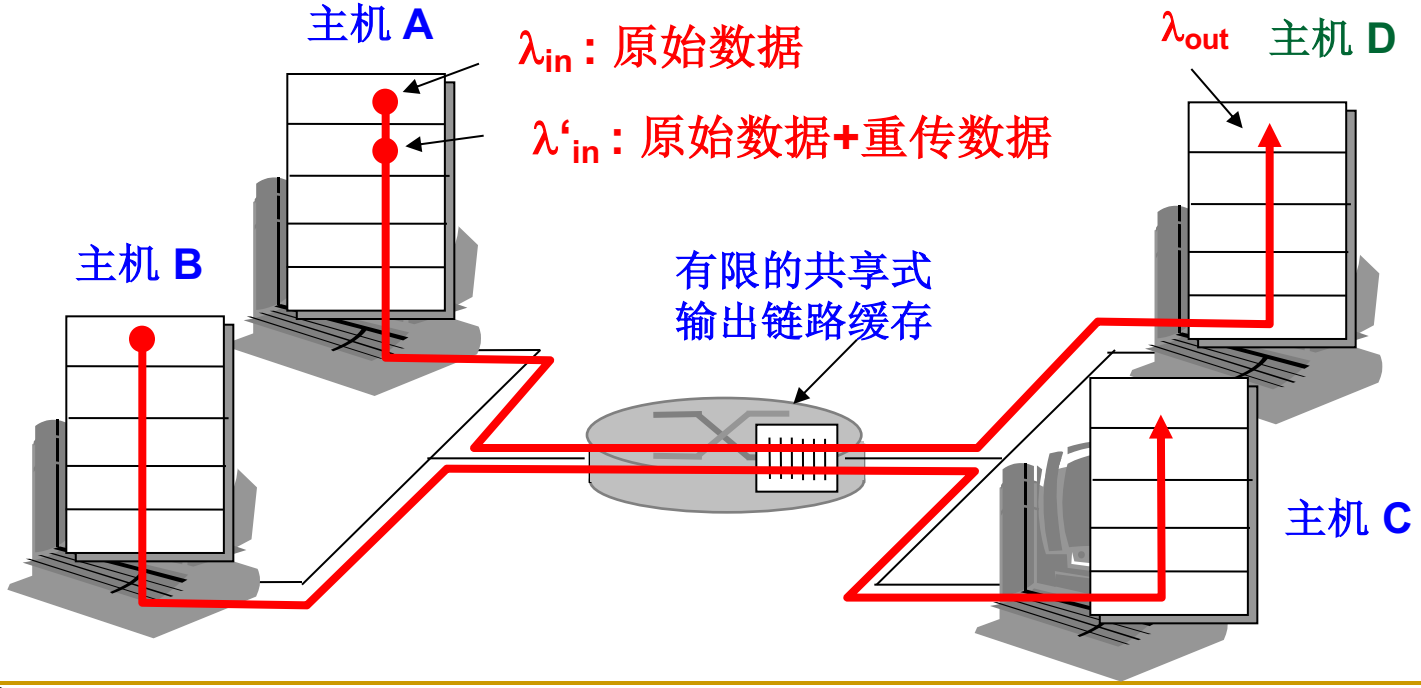
- **情境1**
  - 最大可获得的每连接吞吐量：C/2
  - 出现拥塞时延时变大，当发送速率超过C/2，出现拥塞



拥塞代价：当分组到达速率接近链路容量时，分组经历的巨大排队时延

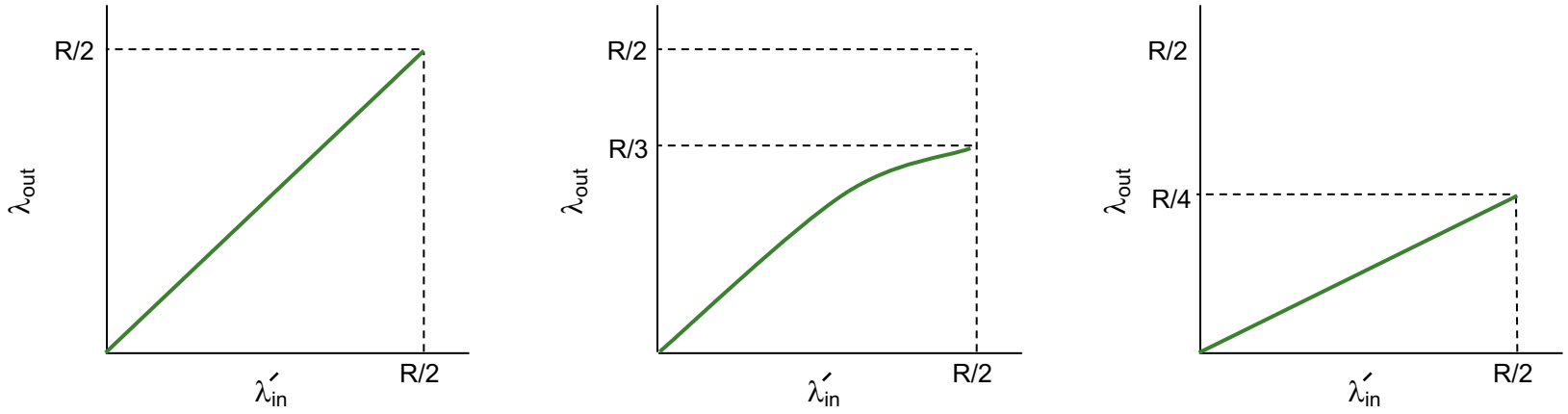
3.6 拥塞控制原理

- **情境2**
  - 一个具有 *有限* 缓存的路由器
  - 发送方对丢失的分组进行重传



3.6 拥塞控制原理

- 设计期望：$\lambda_{in} = \lambda'_{in} = \lambda_{out}$ （不重传）
- “理想” 的重传是仅仅在丢包时才发生重传：$\lambda'_{in} > \lambda_{out}$
- 对延迟到达(而非丢失)的分组的重传使得 λ'_{in} 比理想情况下更大于λ_{out}

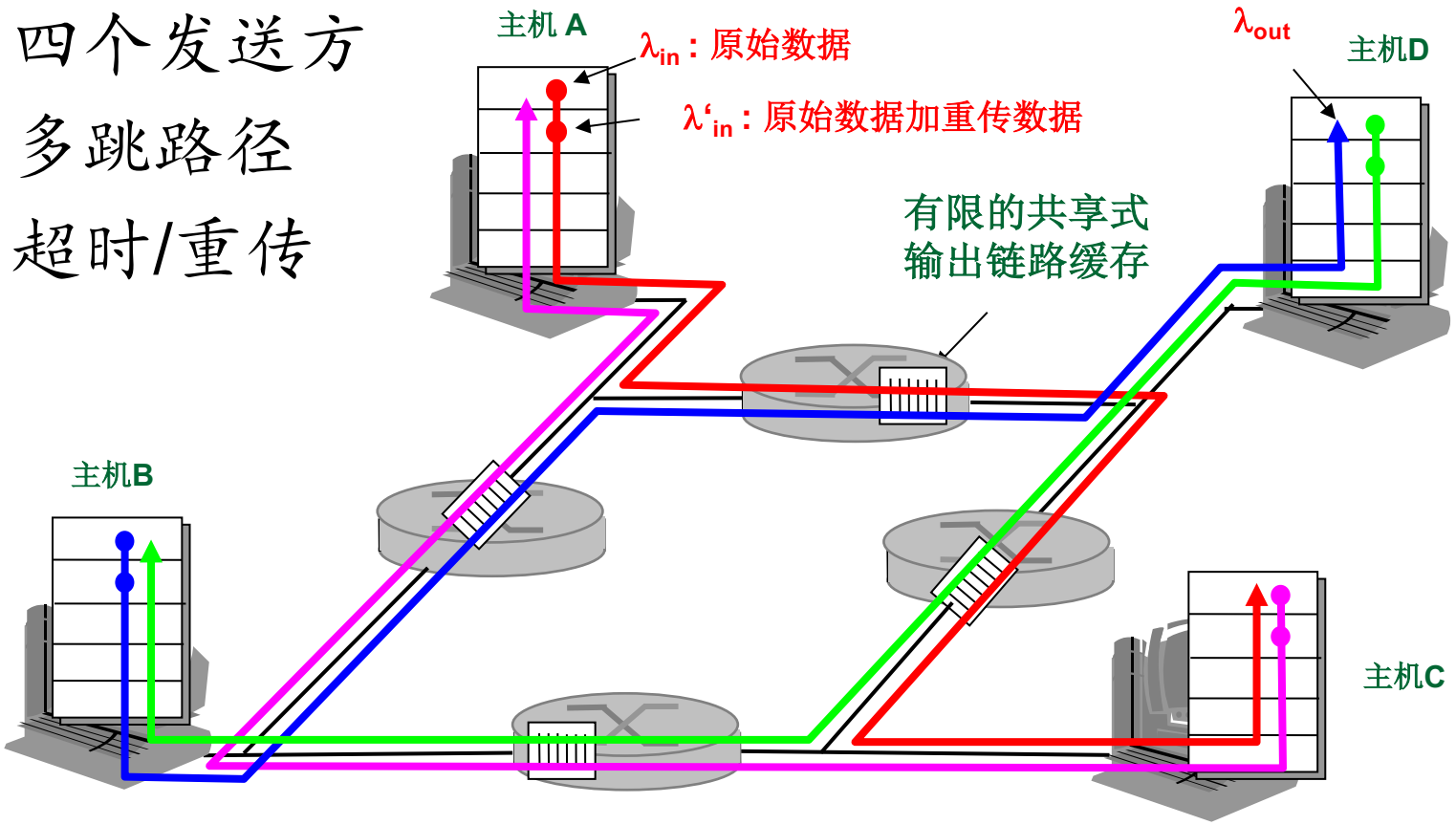


拥塞的“开销”：

- 发送方必须重传以补偿因为缓存溢出而丢失的分组
- 发送方在遇到大的时延时所进行的不必要重传会引起路由器转发不必要的分组拷贝而占用其链路带宽

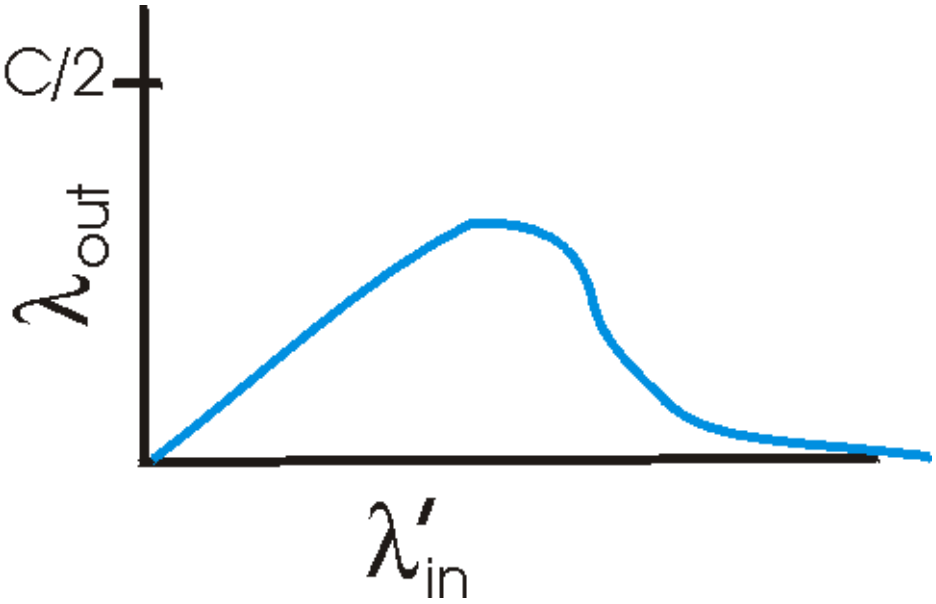
3.6 拥塞控制原理

- **情境3**
  - 四个发送方
  - 多跳路径
  - 超时/重传



3.6 拥塞控制原理

- **情境3**



拥塞的另一个”开销”：

- 当分组被丢弃时，该分组曾用到的所有“上游”传输容量被浪费了！

3.6 拥塞控制原理

- **拥塞控制的方法**
  - 网络辅助的拥塞控制
    - 直接网络反馈：路由器以阻塞分组的形式通知发送方“网络拥塞了”
    - 经由接收方的网络反馈：路由器标识从发送方流向接收方分组中的某个字段以指示拥塞的产生，由接收方通知发送方“网络拥塞了”
  - 端到端拥塞控制
    - 网络层不为拥塞控制提供任何帮助和支持
    - 端系统通过对网络行为（丢包或时延增加）的观测判断网络是否发生拥塞
    - 目前TCP采用该种方法

3.7 TCP拥塞控制

- **TCP拥塞控制为端到端拥塞控制**

- **TCP进行拥塞控制的方法**
  - 每个发送方自动感知网络拥塞的程度
  - 发送方根据感知的结果限制外发的流量
    - 如果前方路径上出现了拥塞，则降低发送速率
    - 如果前方路径上没有出现拥塞，则增加发送速率

3.7 TCP拥塞控制

TCP拥塞控制需要解决的三个问题

- TCP发送方如何限制外发流量的速率
  - 引入拥塞窗口

为了方便讨论，我们假设接收缓存足够大，因此足够大，可以不考虑RcvWindow

LastByteSent-LastByteAcked ≤ min{CongWin,RcvWindow}

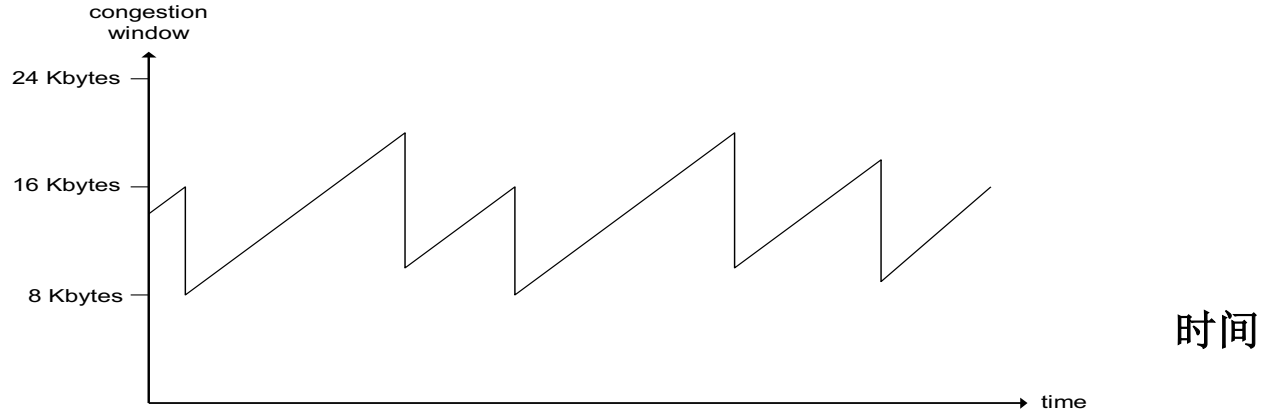
粗略地讲，该限制条件允许在一个RTT内发送方发送CongWin字节的数据，在该RTT结束时收到一个确认报文，这时已发送发且没被确认的字节数<=CongWin-1

$$rate = \frac{CongWin}{RTT} \text{ Bytes/sec}$$

- 发送方如何感知拥塞
  - 超时
  - 三个冗余ACK
- 在感知到拥塞后，发送方如何调节发送速率

3.7 TCP拥塞控制

- **TCP拥塞控制算法（Reno算法）**
  - 加性增，乘性减（AIMD）
    - 出现丢包事件后将当前 CongWin 大小减半，可以大大减少注入到网络中的分组数
    - 当没有丢包事件发生了，每个RTT之后将CongWin增大1个MSS，使拥塞窗口缓慢增大，以防止网络过早出现拥塞，线性增长阶段称为拥塞避免（CA，Congestion Avoidance）阶段



3.7 TCP拥塞控制

- **如何做到每个RTT将CongWin增加一个MSS**
  - 每收到一个ACK，将CongWin增加MSS(MSS/CongWin)
  - 因为Rate = CongWin/RTT,一个报文段大小为MSS字节
  - 因此一个RTT内，一共发送CongWin/MSS个报文段
  - 即一个RTT内，有CongWin/MSS个ACK
  - 每个ACK增加MSS(MSS/CongWin)，一个RTT内共增加

    MSS * (MSS / CongWin) * (CongWin/MSS) = 1个MSS

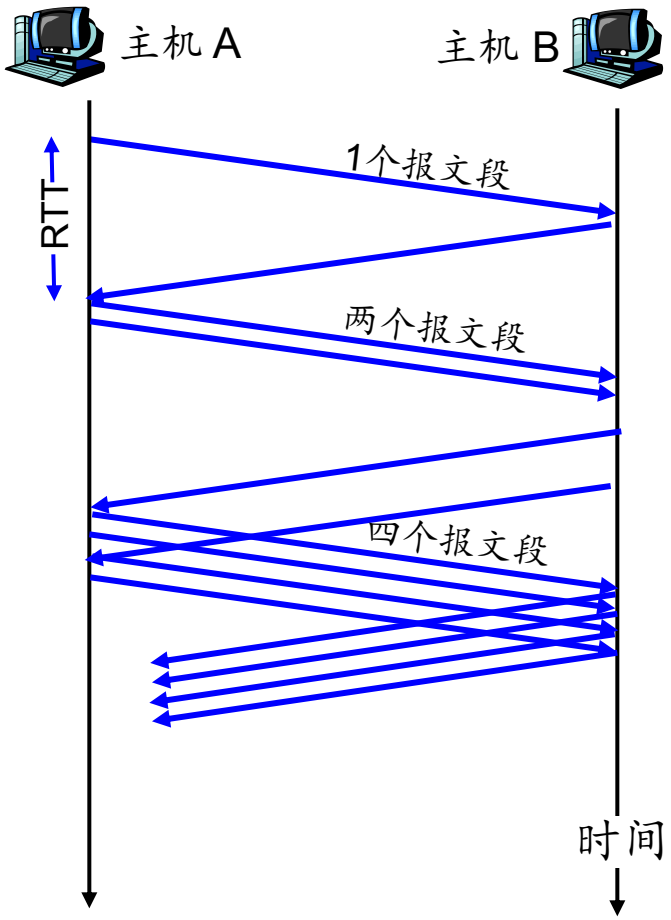
3.7 TCP拥塞控制

- 慢启动
  - 建立连接时, **CongWin** = 1 MSS
    - 例如: MSS = 500 bytes & RTT = 200 msec
    - 初始速率 = 20 kbps
  - 可用带宽 >> MSS/RTT
    - 初始阶段以指数的速度增加发送速率
  - 连接初始阶段，以指数的速度增加发送速率，直到发生一个丢包事件为止
    - 每收到一次确认则将CongWin的值增加一个MSS

总结: 初始速率很低但速率的增长速度很快

3.7 TCP拥塞控制

- 慢启动

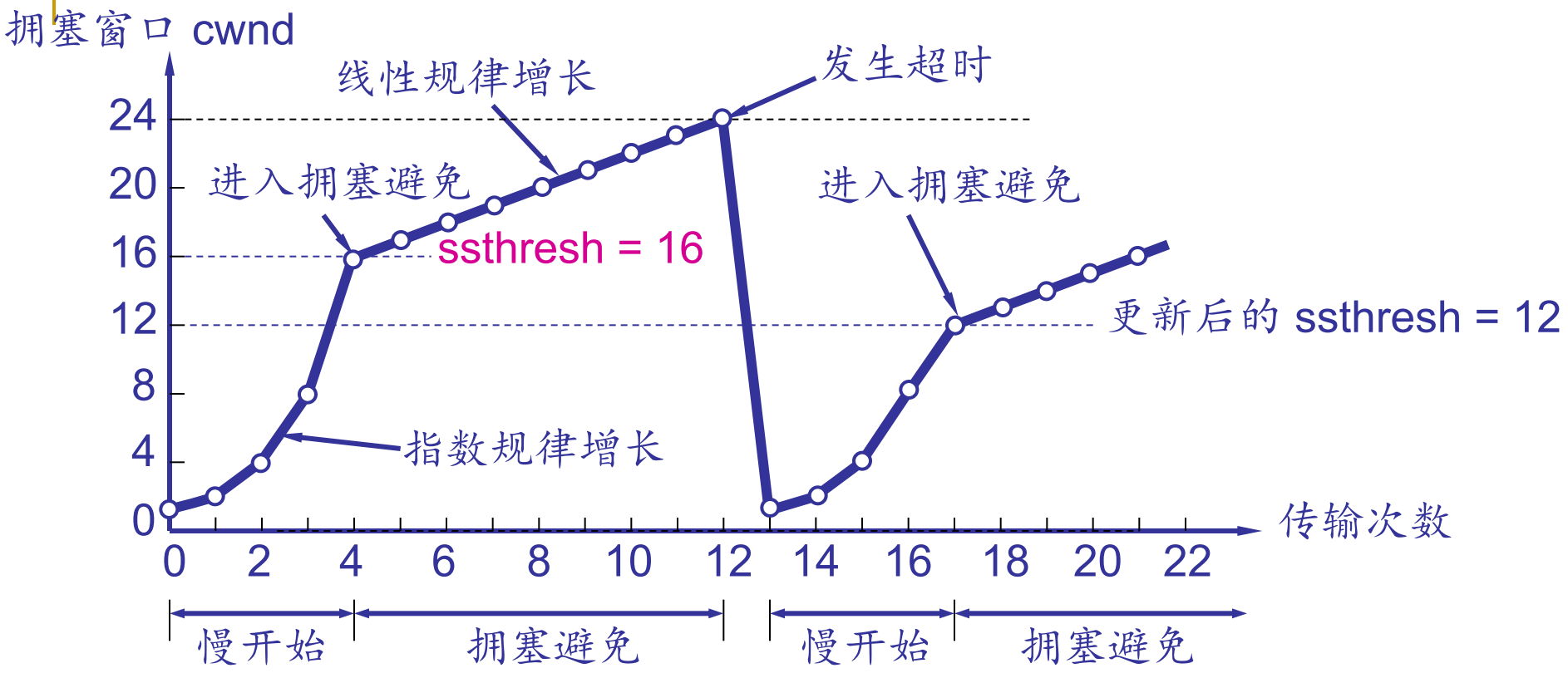


3.7 TCP拥塞控制

- 对收到3个重复ACK的反应——快速重传
  - 门限值设为当前CongWin的一半（门限值初始值65kB）
  - 将CongWin减为新的门限值+3MSS
  - 线性增大拥塞窗口
- 对超时事件的反应：设置门限（Threshold）
  - 门限值设为当前CongWin的一半（门限值初始值65kB）
  - 将CongWin设为1个 MSS大小;
  - 窗口以指数速度增大
  - 窗口增大到门限值之后，再以线性速度增大

特别说明：早期的TCP Tahoe版本对上述两个事件并不区分，统一将CongWin降为1。实际上，3个重复的ACK相对超时来说是一个预警信号，因此在Reno版中作了区分

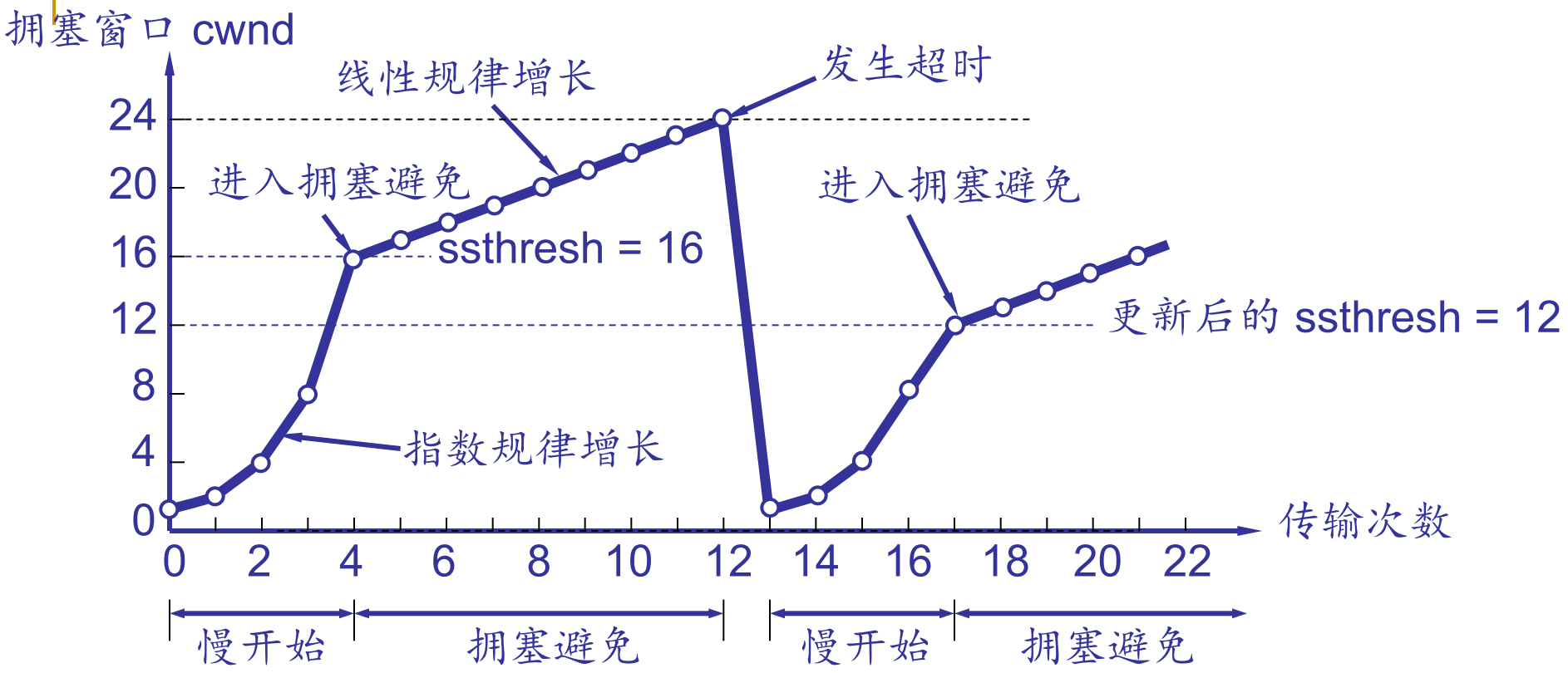
慢开始和拥塞避免算法的实现举例



当 TCP 连接进行初始化时，将拥塞窗口置为 1(单位为MSS)。图中的窗口单位不使用字节而使用MSS。水平坐标单位为RTT

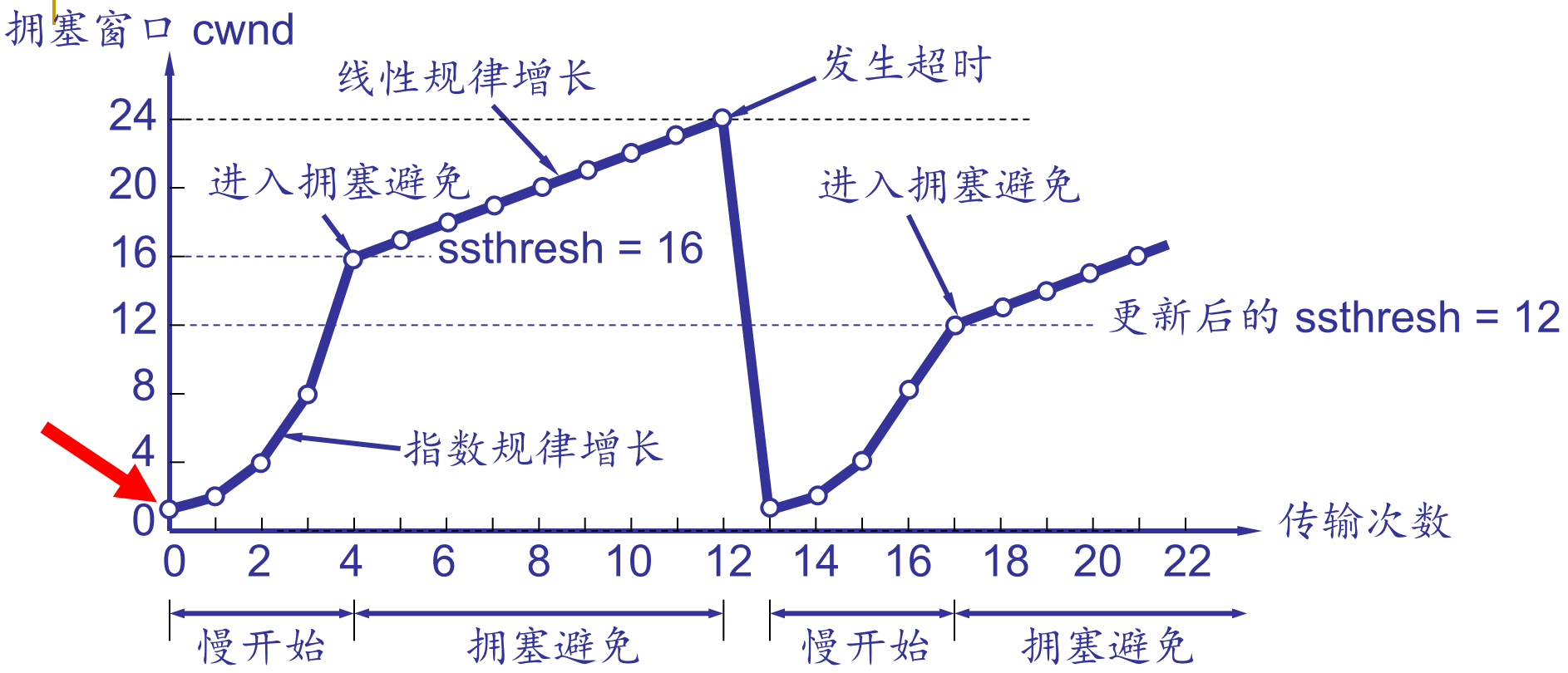
慢启动门限的初始值设置为 16 个报文段，即 ssthresh = 16。

慢开始和拥塞避免算法的实现举例



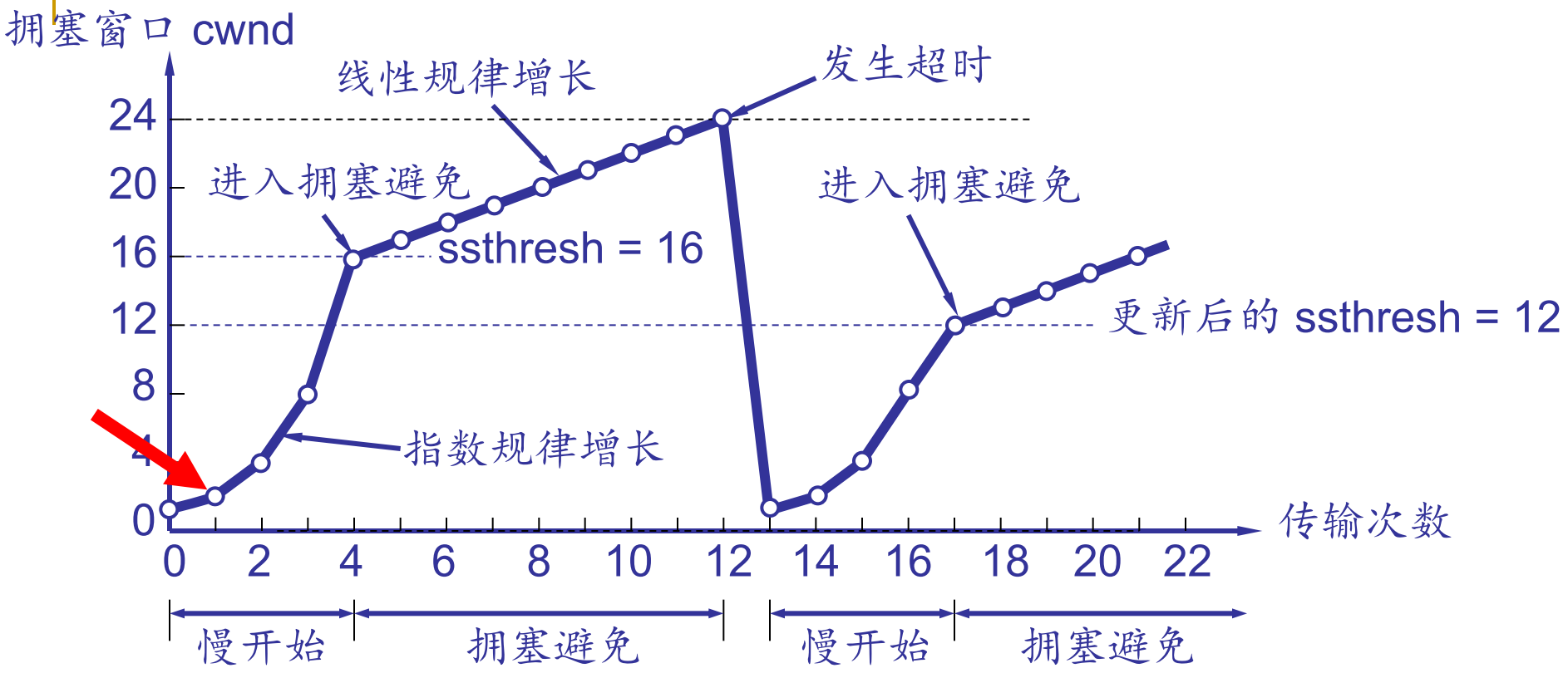
发送端的发送窗口不能超过拥塞窗口 cwnd 和接收端窗口 rwnd 中的最小值。我们假定接收端窗口足够大，因此现在发送窗口的数值等于拥塞窗口的数值。

慢开始和拥塞避免算法的实现举例



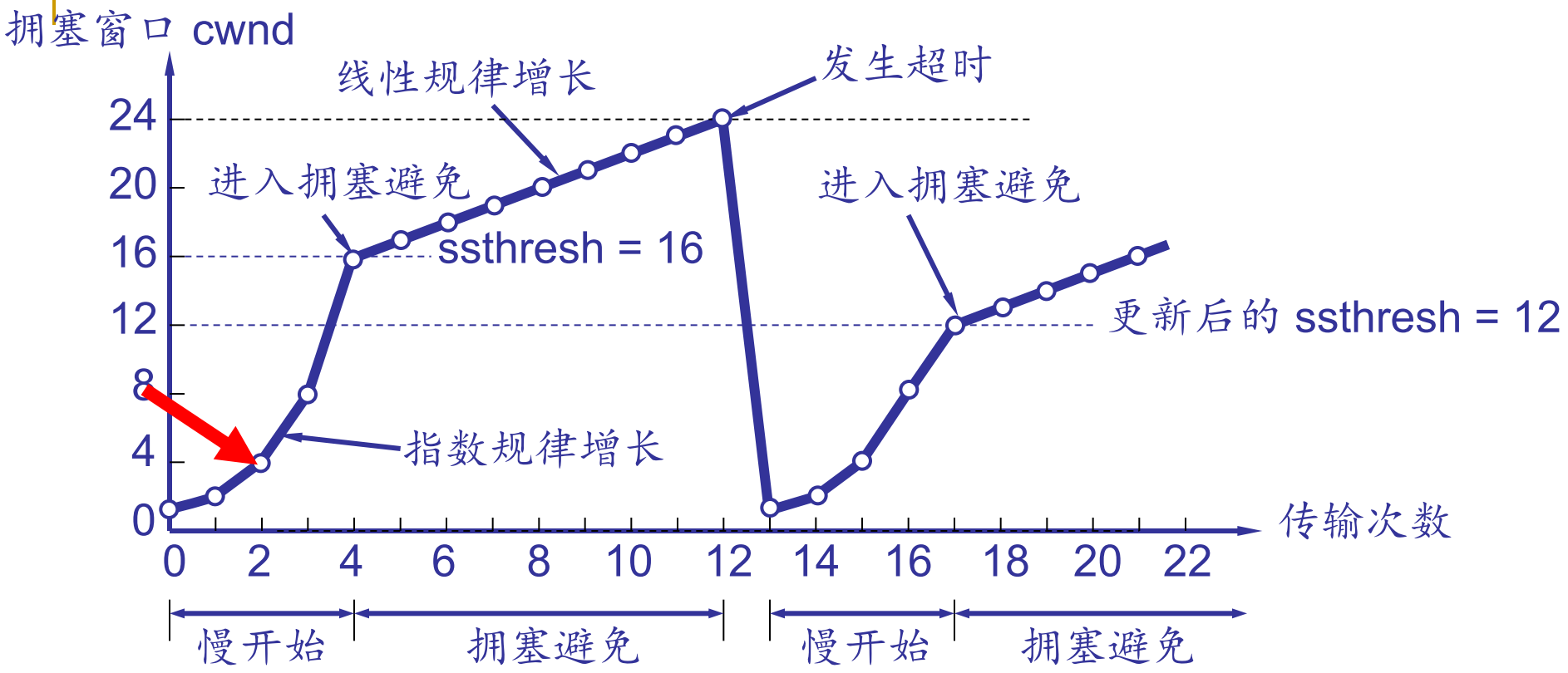
在执行慢开始算法时，拥塞窗口 cwnd 的初始值为 1，发送第一个报文段 M_0。

慢开始和拥塞避免算法的实现举例



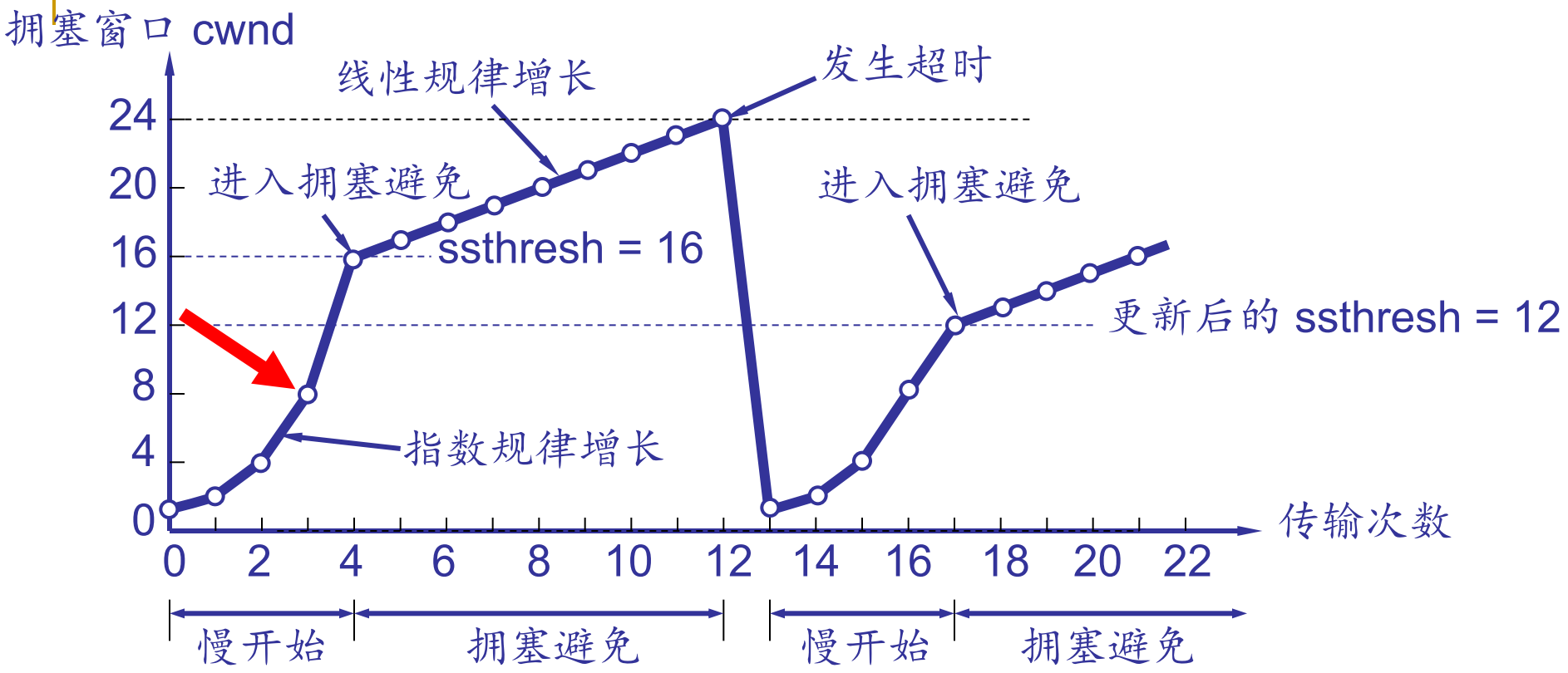
发送端收到 ACK_1（确认 M_0，期望收到 M_1）后，将 cwnd 从 1 增大到 2，于是发送端可以接着发送 M_1 和 M_2 两个报文段。

慢开始和拥塞避免算法的实现举例



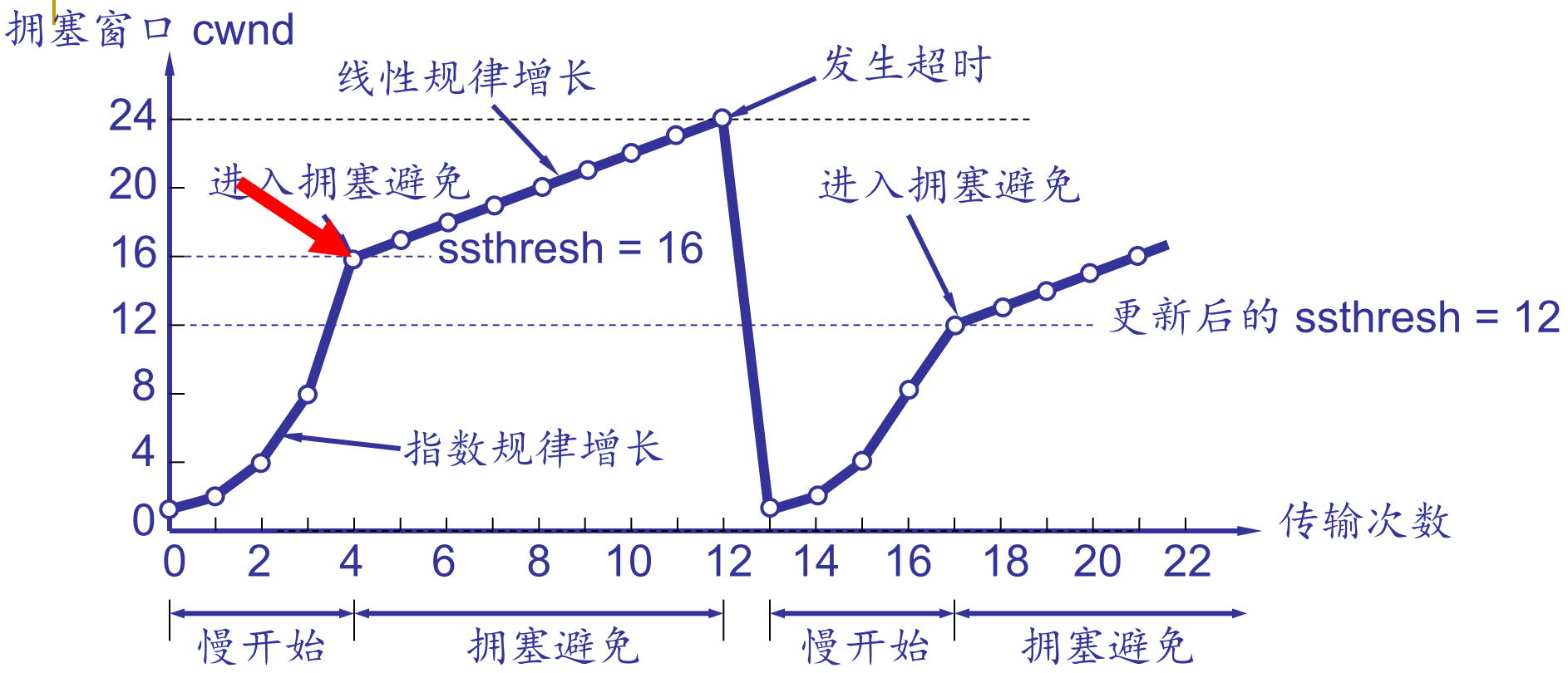
接收端发回 ACK_2 和 ACK_3。发送端每收到一个对新报文段的确认 ACK，就把发送端的拥塞窗口+1MSS。现在发送端的 cwnd 从 2 增大到 4，并可发送 M_3 ~ M_6共 4个报文段。

慢开始和拥塞避免算法的实现举例



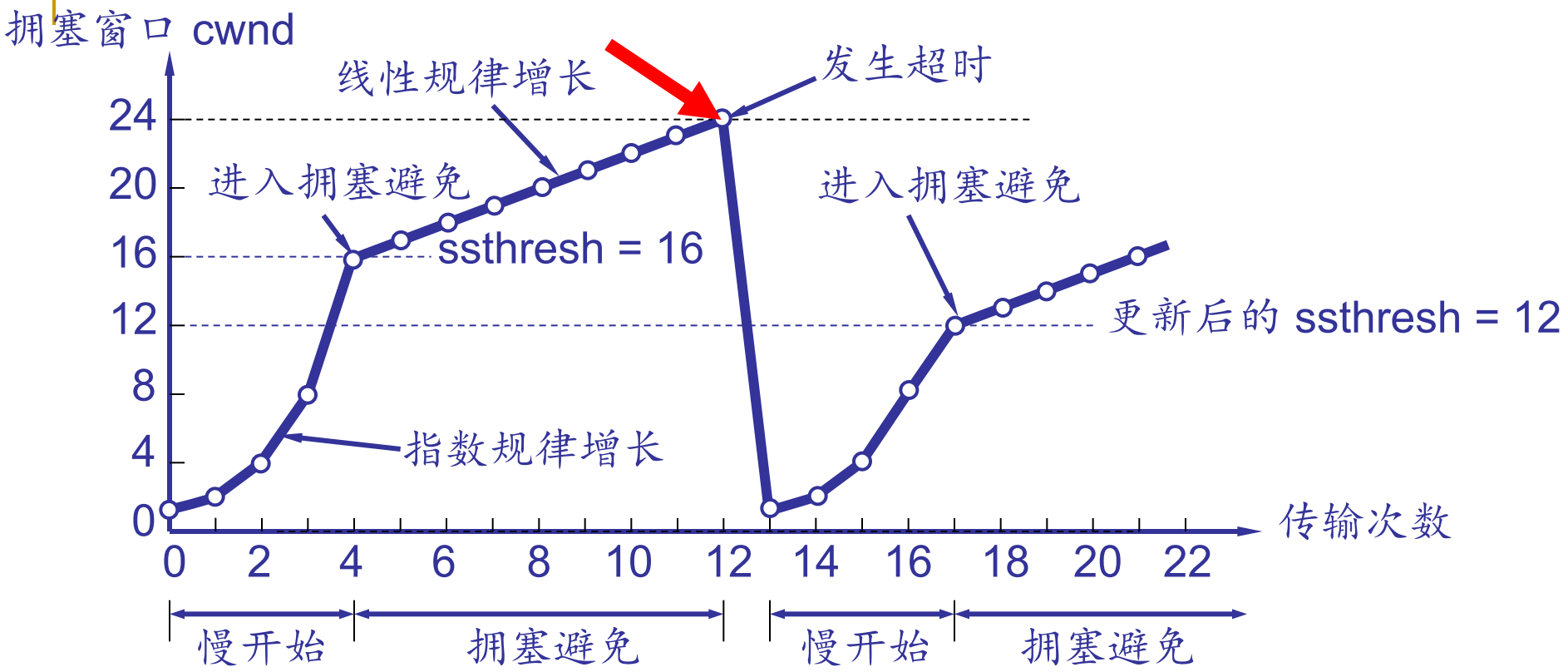
发送端每收到一个对新报文段的确认 ACK，就把发送端的拥塞窗口+1MSS，因此拥塞窗口 cwnd 随着传输次数按指数规律增长。

慢开始和拥塞避免算法的实现举例



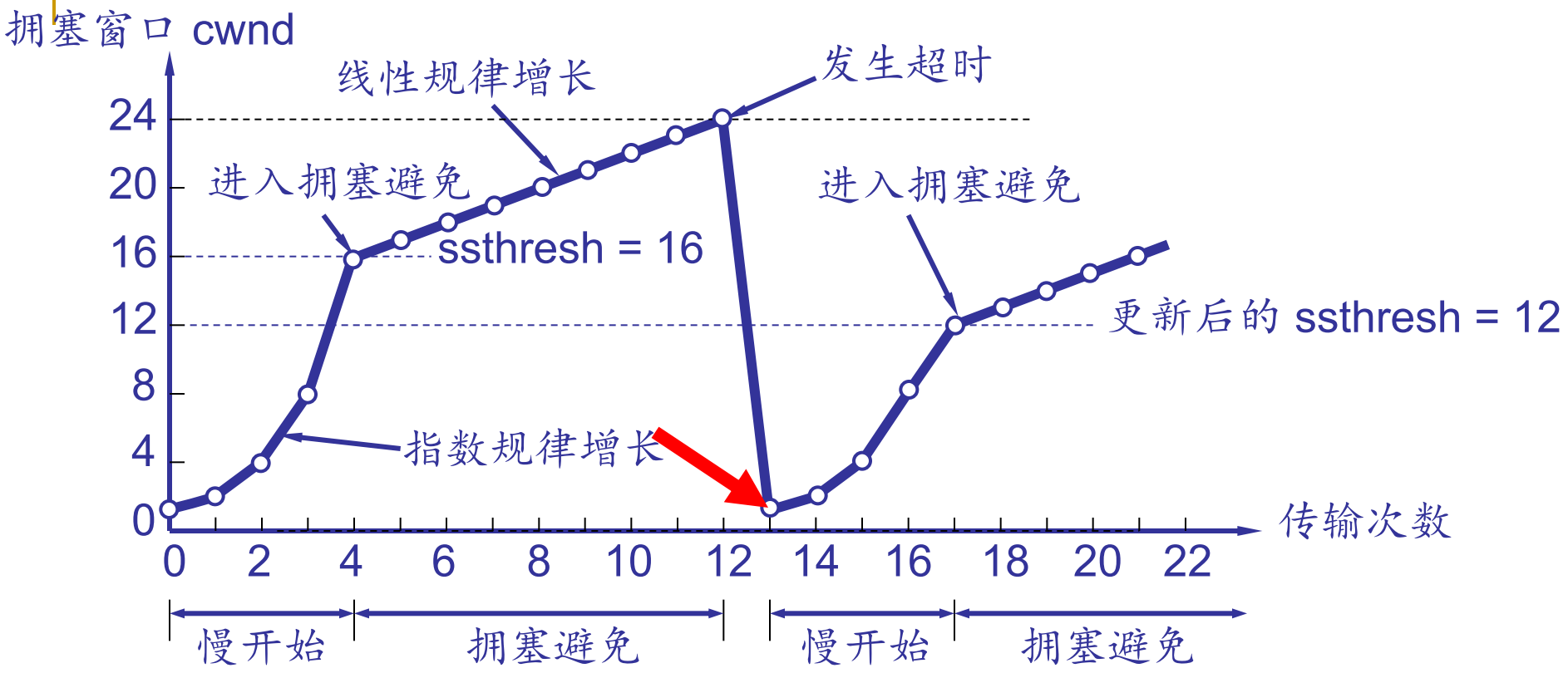
当拥塞窗口 cwnd 增长到慢开始门限值 ssthresh 时（即当 cwnd = 16 时），就改为执行拥塞避免算法，拥塞窗口按线性规律增长。

慢开始和拥塞避免算法的实现举例



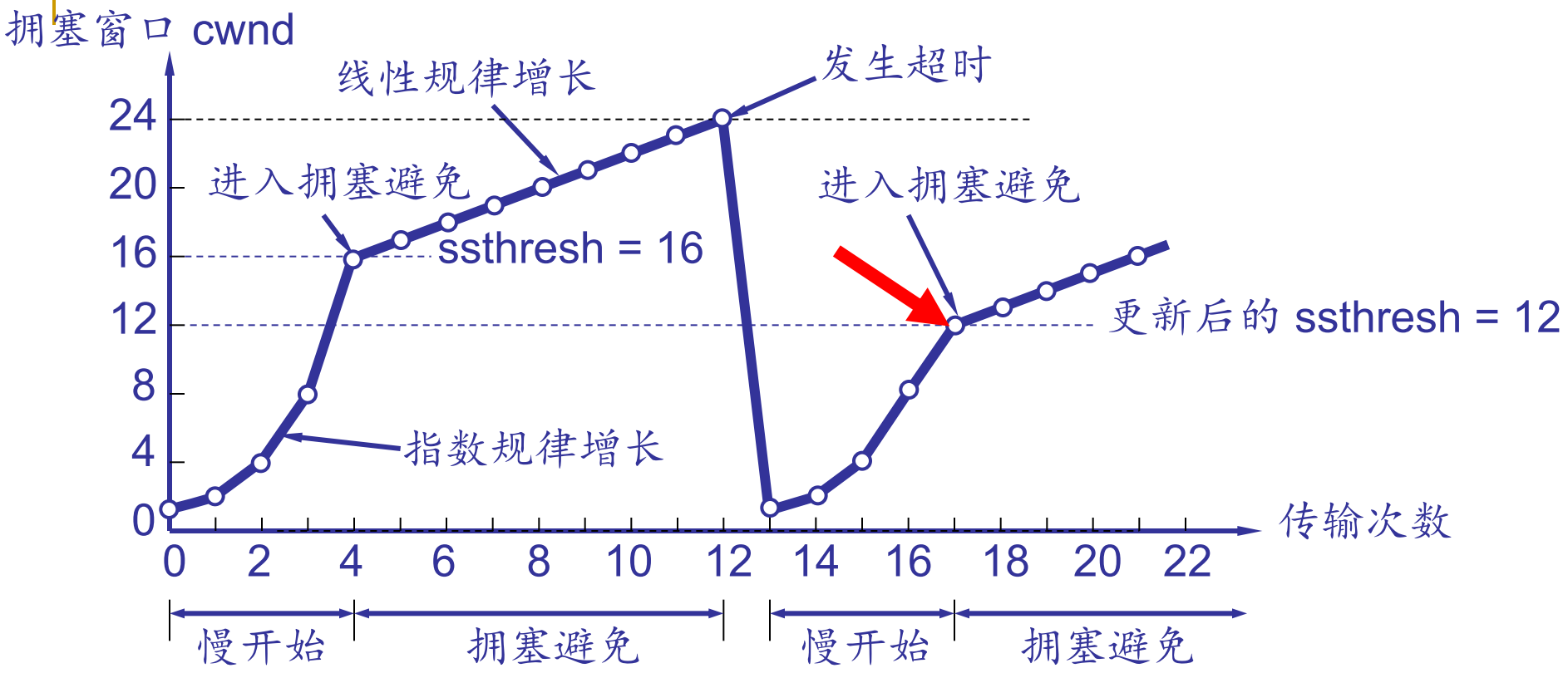
假定拥塞窗口的数值增长到 24 时，网络出现超时（表明网络拥塞了）。

慢开始和拥塞避免算法的实现举例



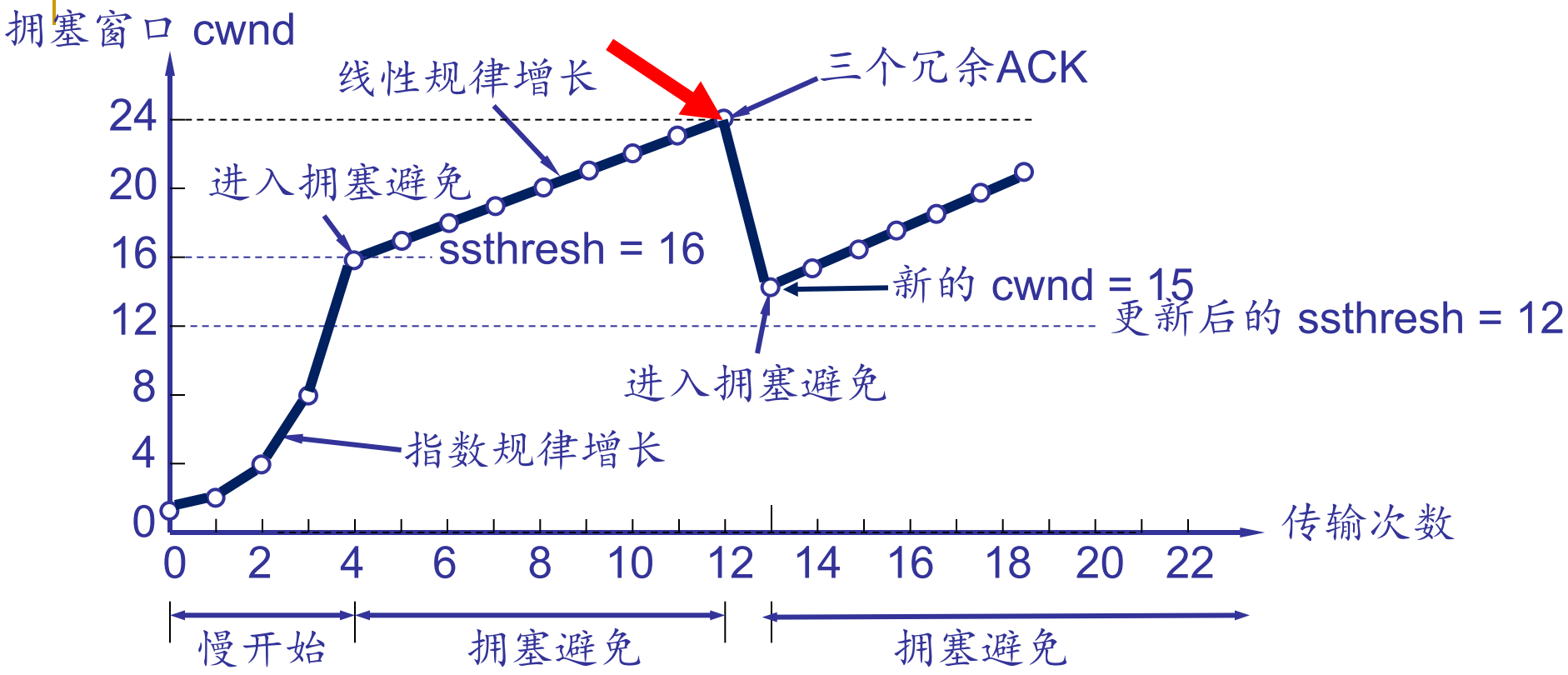
更新后的 ssthresh 值变为 12（即发送窗口数值 24 的一半），拥塞窗口再重新设置为 1，并执行慢开始算法。

慢开始和拥塞避免算法的实现举例



当 cwnd = 12 时改为执行拥塞避免算法，拥塞窗口按按线性规律增长，每经过一个往返时延就增加一个 MSS 的大小。

慢开始和拥塞避免算法的实现举例



假定拥塞窗口的数值增长到 24 时，网络出现冗余ACK

3.7 TCP拥塞控制

- 快速恢复（TCP推荐但非必须实现）
  - 3个冗余ACK进入快速重传后
  - 每收到一个冗余ACK：CongWin++
  - 直至收到一个新的ACK：CongWin=门限值，重新进入拥塞避免
  - 在进入快速恢复之后及重新进入拥塞避免之间，如果出现超时现象，直接按照前述超时事件进行处理

说明：本页内容不纳入考试范围

3.7 TCP拥塞控制

- 额外说明
  - 快速恢复和超时中，门限值并不总等于CongWin/2
  - 门限值=Max(flightSize/2 , 2MSS)
  - flightsize:当时发送窗口中已发出但未确认的报文段数目

  - 门限值=Max(min(拥塞窗口,通知窗口) , 2MSS)　–微软

说明：本页内容不纳入考试范围

3.7 TCP拥塞控制

- TCP拥塞控制算法（Reno）总结
  - 当 拥塞窗口CongWin小于门限值Threshold时，发送方处于 慢启动 阶段，窗口以指数速度增大。
  - 当 拥塞窗口CongWin大于门限值Threshold时，发送方处于 拥塞避免 阶段，窗口线性增大。
  - 当收到 3个重复的ACK 时,门限值Threshold设为拥塞窗口的1/2，而拥塞窗口CongWin设为门限值Threshold+3个MSS。
  - 当 超时 事件发生时，门限值Threshold设为拥塞窗口的1/2，而拥塞窗口CongWin设为1个 MSS。

3.7 TCP拥塞控制

事件	状态	TCP发送方动作	说明
收到前面未确认数据的ACK	慢启动(SS)	CongWin = CongWin + MSS, If (CongWin > Threshold) 设置状态为 “拥塞避免”	导致每过一个RTT则CongWin翻倍
收到前面未确认数据的ACK	拥塞避免 (CA)	CongWin = CongWin+MSS * (MSS/CongWin)	加性增，每个RTT导致 CongWin 增大1个MSS
由3个重复ACK检测到丢包事件	SS 或 CA	Threshold = CongWin/2, CongWin = Threshold+3*MSS, 设置状态为 “拥塞避免”	快速恢复，实现乘性减. CongWin不低于1个MSS.
超时	SS 或 CA	Threshold = CongWin/2, CongWin = 1 MSS, 设置状态为 “慢启动”	进入慢启动
重复ACK	SS 或 CA	对确认的报文段增加重复ACK的计数	CongWin 和 Threshold不变

3.7 TCP拥塞控制

考虑TCP Reno算法,现做出如下假定：

（1）拥塞窗口的计量单位采用报文段，而不采用字节；

（2）初始Threshold值设为25个报文段；

（3）仅考虑传播时延，不考虑传输时延；

（4）第9、36个报文段在第一次传输过程中发生了超时；

（5）连续四次收到了对第21个报文段的ACK；

（6）重传方式为回退N步重传方式。

请画出拥塞窗口相对往返时延RTT的函数图。

3.7 TCP拥塞控制

- **TCP的吞吐量**
  - 作为窗口大小和RTT的函数TCP的平均吞吐量应该是什么样的?
    - 忽略慢启动
  - 假定当丢包事件发生时，窗口大小为 W.
    - 此时 吞吐量为W/RTT
  - 丢包事件发生后，窗口大小减为W/2, 吞吐量为W/2RTT.
  - 因此平均吞吐量为: 0.75 W/RTT

3.7 TCP拥塞控制

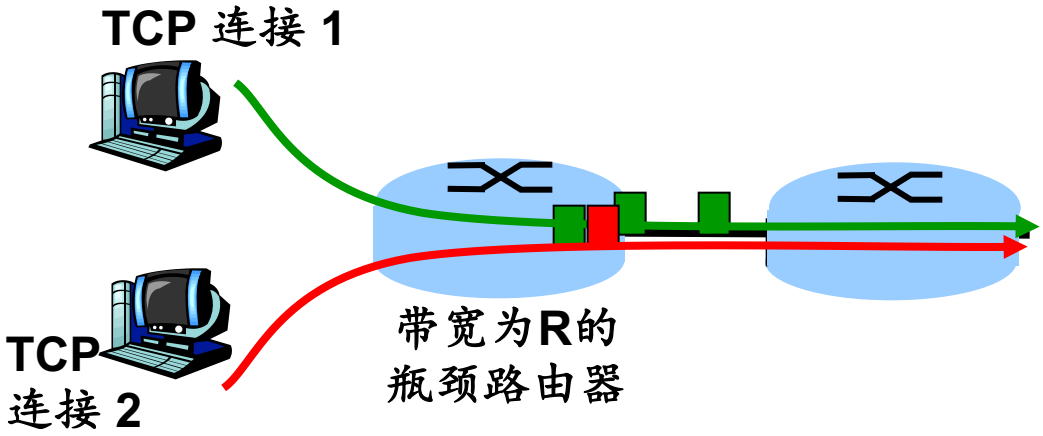
- **TCP吞吐量的进一步讨论**
  - 吞吐量是丢包率(L)的函数:

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

  - 对于一条MSS=1500字节，RTT=100ms的TCP连接而言，如果希望达到10Gbps的吞吐量，那么丢包率L不能高于$2x10^{-10}$

3.7 TCP拥塞控制

- **TCP拥塞控制的公平性分析**
  - 公平性的目标
    - 如果K个TCP连接共享同一个带宽为R的瓶颈链路，每个连接的平均传输速率为 R/K



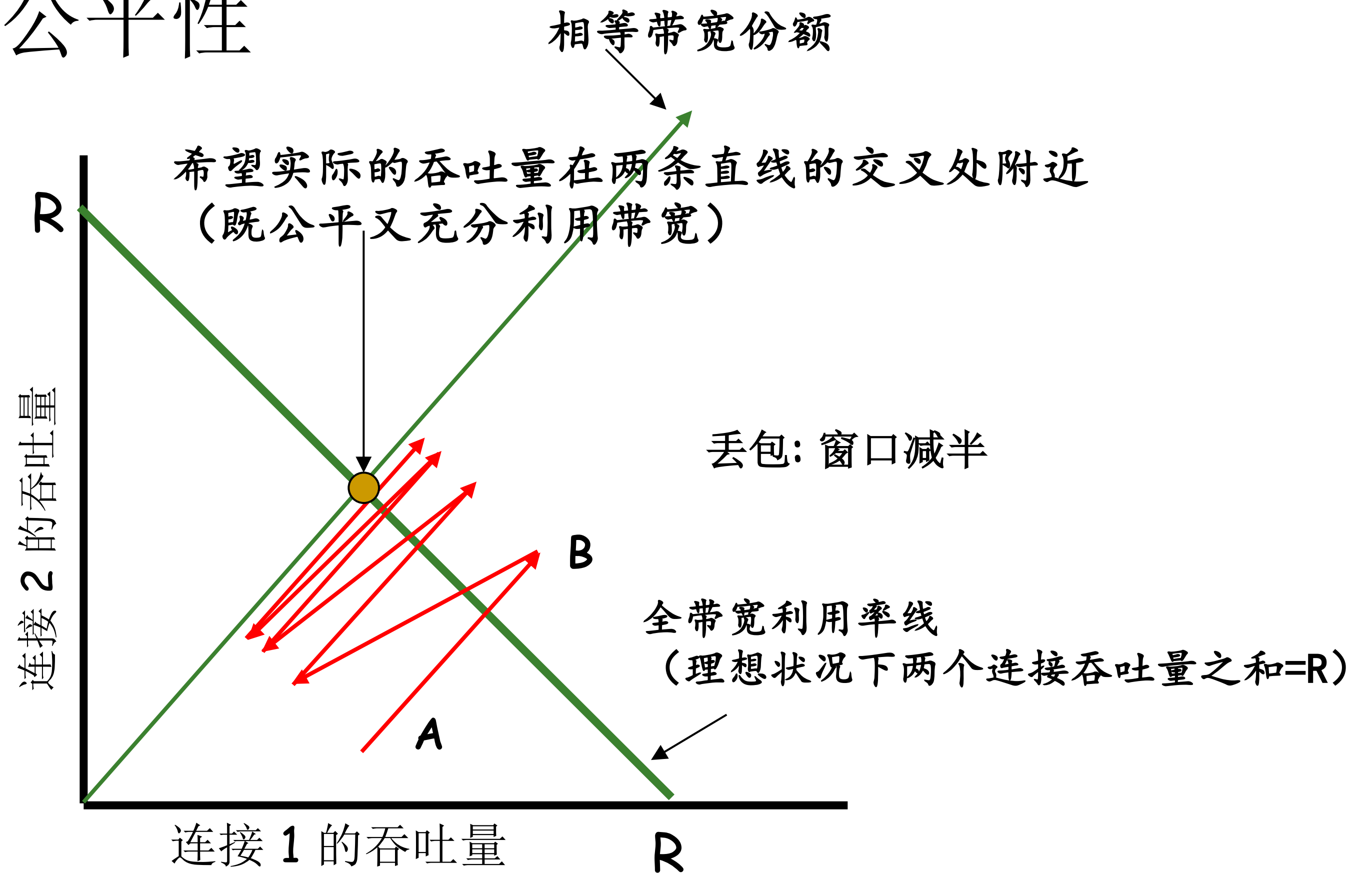
3.7 TCP拥塞控制

- TCP的公平性

以此类推，最后两个连接的带宽始终是沿着相同带宽份额线波动

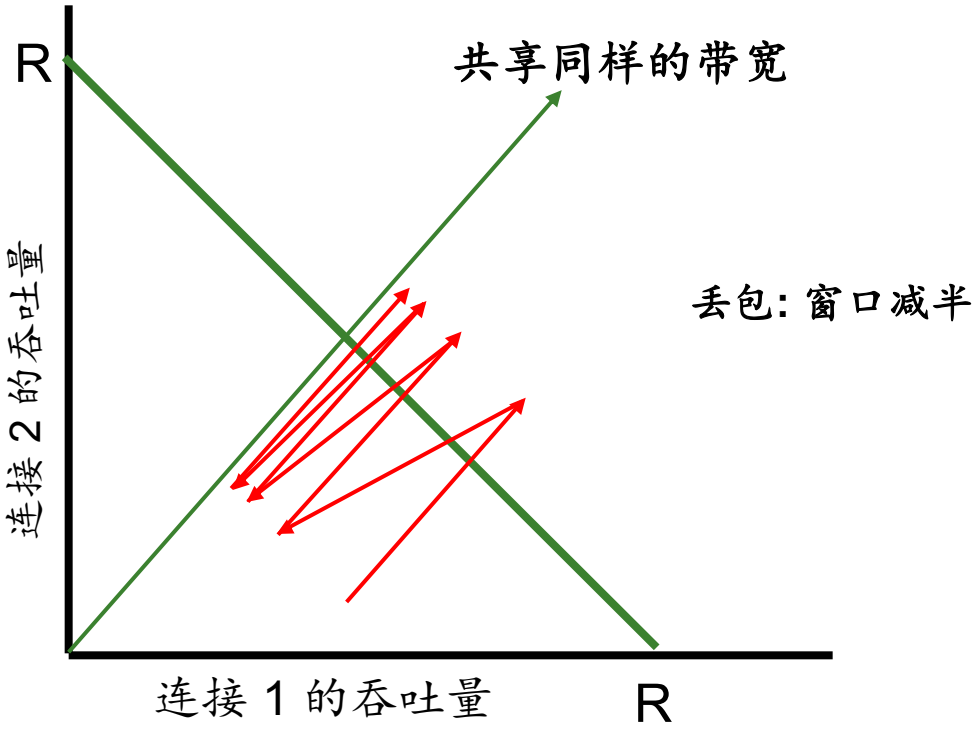
TCP的拥塞控制会保证公平性

吞吐量（都是在一个RTT内增加一个MSS），因此直线以45度增长



3.7 TCP拥塞控制

- TCP的公平性



3.7 TCP拥塞控制

- 公平性和UDP
  - 从TCP的观点看，UDP是不公平的：UDP抢了TCP的带宽
  - 多媒体应用一般不使用 TCP
    - 不希望因为拥塞控制影响其速率
  - 多媒体应用采用UDP：
    - 恒定的速率传输音频和视频数据，可容忍丢包
- 公平性和并行TCP连接
  - 当一个应用使用多个并行TCP连接时，会占用更多的带宽。（从应用的角度看，不能保证TCP是公平的）
  - 无法阻止应用在两个主机之间建立多个并行的连接.
  - Web浏览器就是如此
    - 例子：速率为R的链路支持10个并发连接
    - 应用请求一个TCP连接，获得R/10的速率
    - 应用请求11个TCP连接，将获得超过R/2的速率

课后思考题

- 复习题 1、4、8、12~15、17
- 习　　题 1、6、14、18、22、27、31、32、37、40、50、52、56

作 业

- 习题
  - 18、27、31、40、45、56