

计算机网络

第3章 运输层

目 录

- 概述和运输层服务
- 多路复用与多路分解
- 无连接传输 : UDP
- 可靠数据传输的原理
- 面向连接的传输 : TCP
- 拥塞控制原理
- TCP拥塞控制

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

类似于淘宝网购:

宿舍里的成员要在天猫上买课本

- 进程 = 买家和卖家
- 进程间报文 = 要买的课本，会用快递包裹的方式送达
- 主机 = 宿舍、卖家店铺
- 运输协议 = 买家（张三）和卖家（李四）的约定
- 网络层协议 = 快递公司提供的服务

3.1 概述和运输层服务

■ 上例中的几种特殊场景

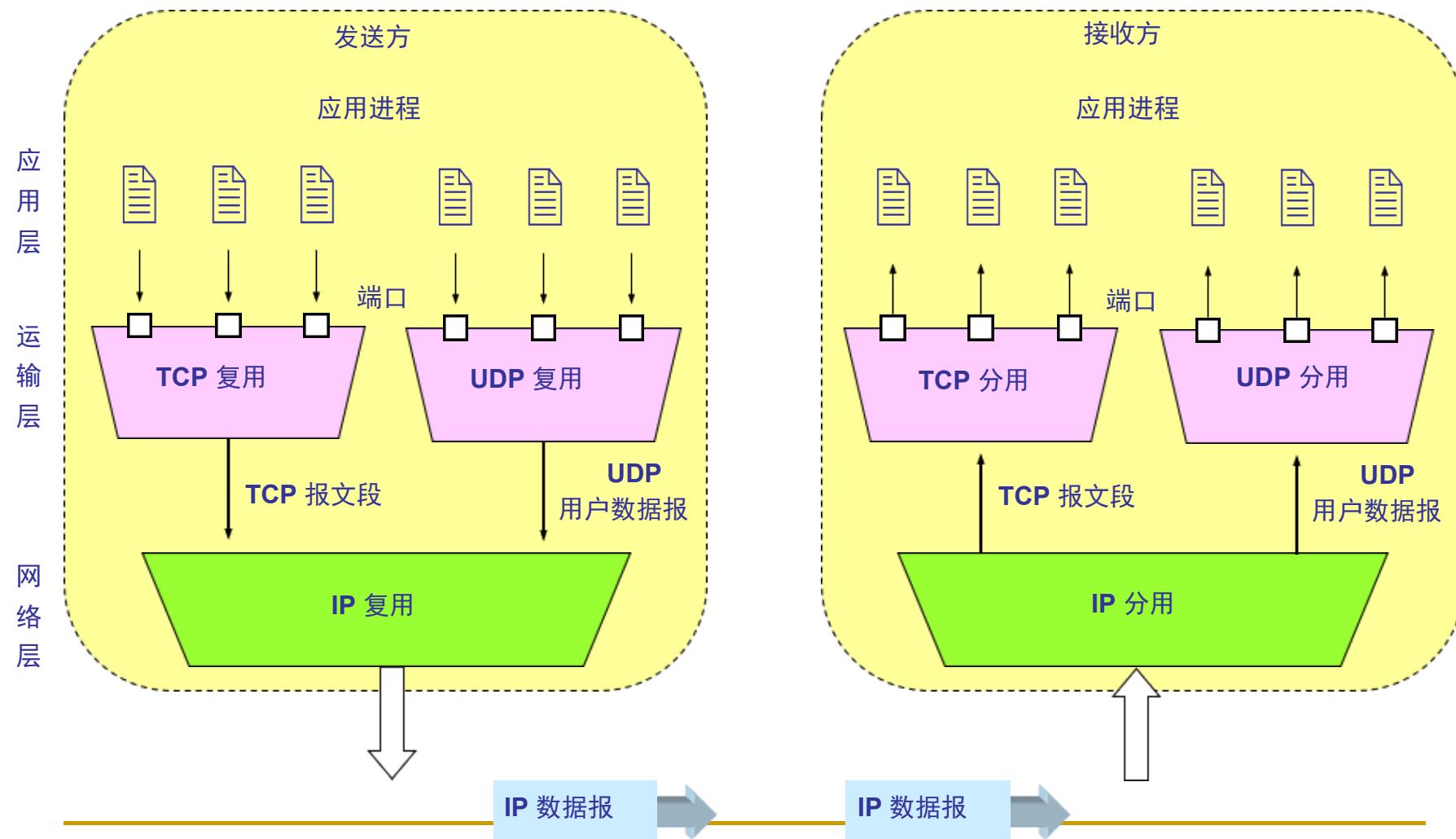
- 张三不买了或者李四不卖了，换成张五和李六
 - 不同的运输层协议可能提供不一样的服务
- 快递公司不承诺快递送达的最长时间
 - 运输层协议能够提供的服务受到底层网络协议的服务模型的限制
- 快递公司不承诺普通包裹一定安全可靠的送达，
可能在路上丢失，但买家、卖家可以约定，在较
长时间内没有收到包裹时，由卖家重发
 - 在网络层不提供某些服务的情况下，运输层自己提供

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



2020年10月6日

7

3.2 多路复用与多路分解

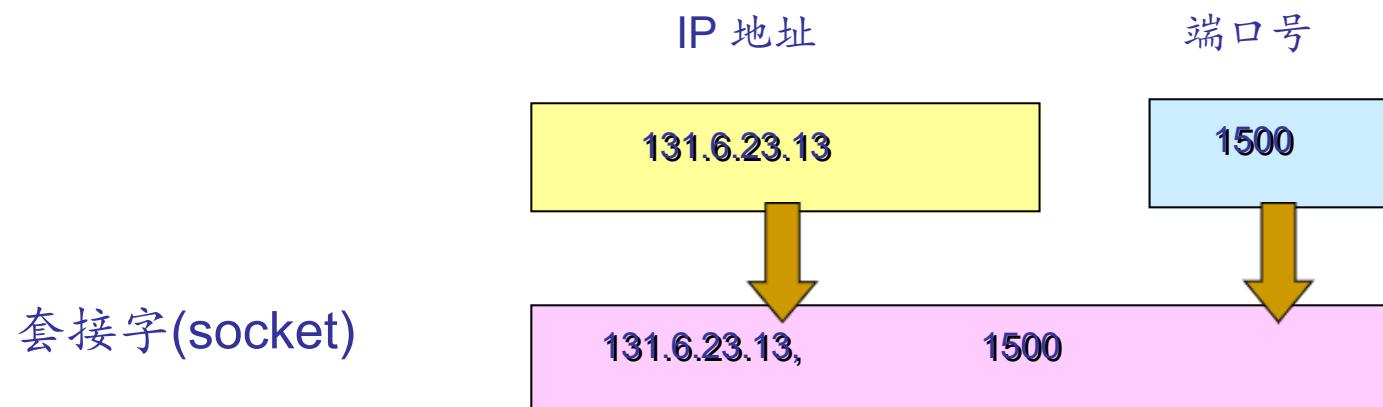
■ 端口

- 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

■ 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：



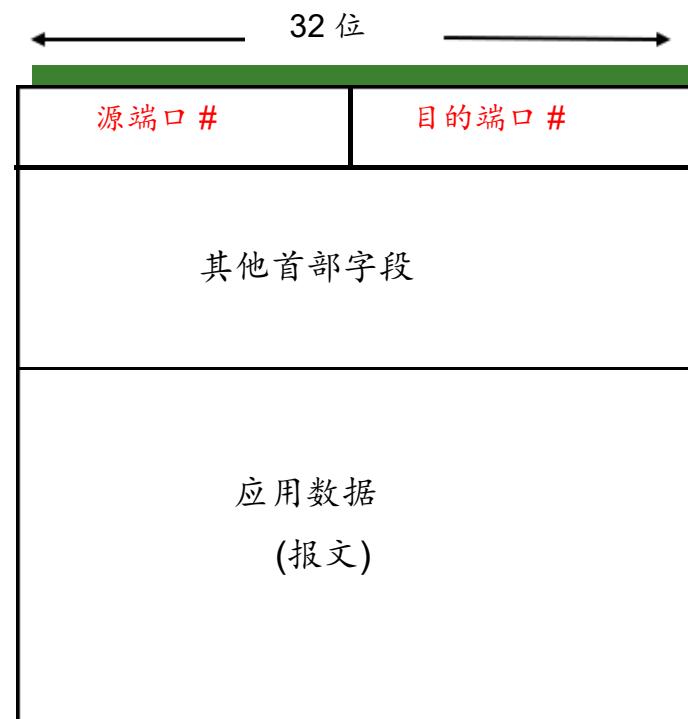
3.2 多路复用与多路分解

■ 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

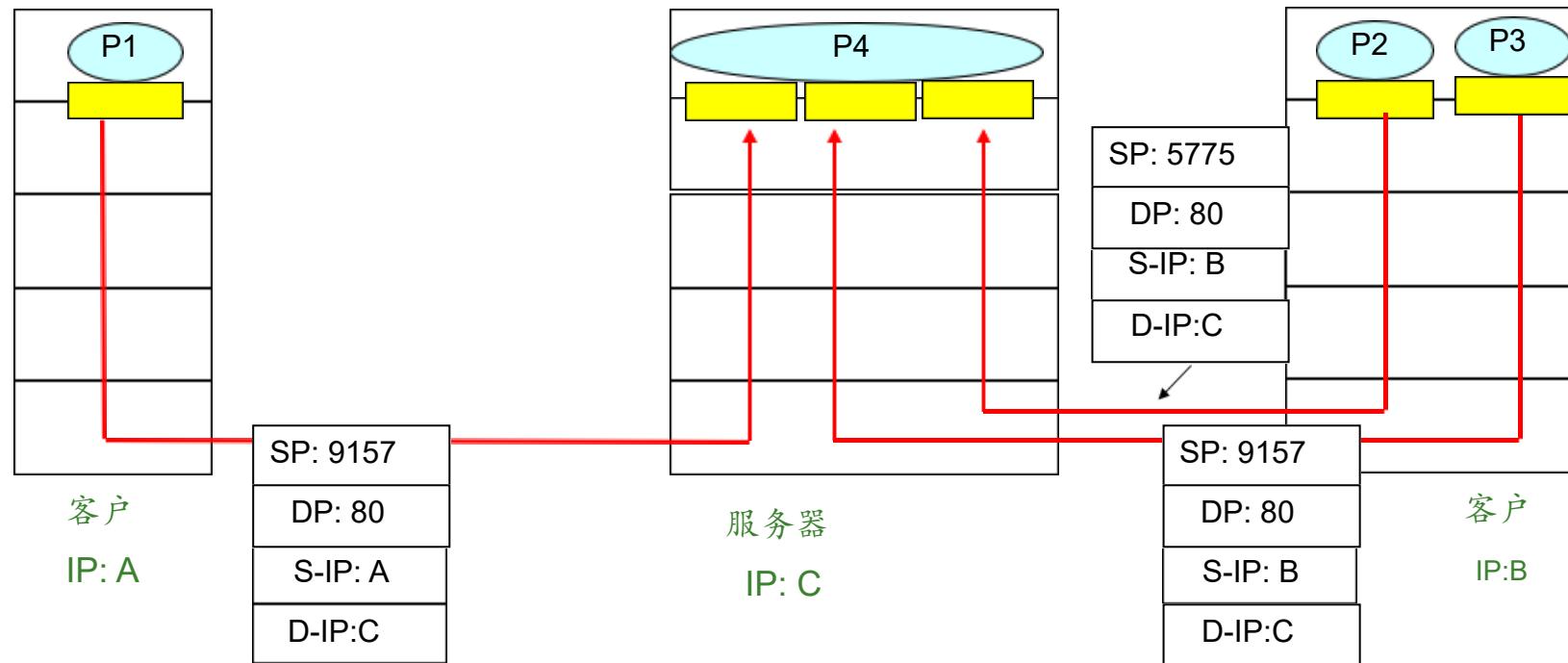
3.2 多路复用与多路分解

■ 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



3.3 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

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

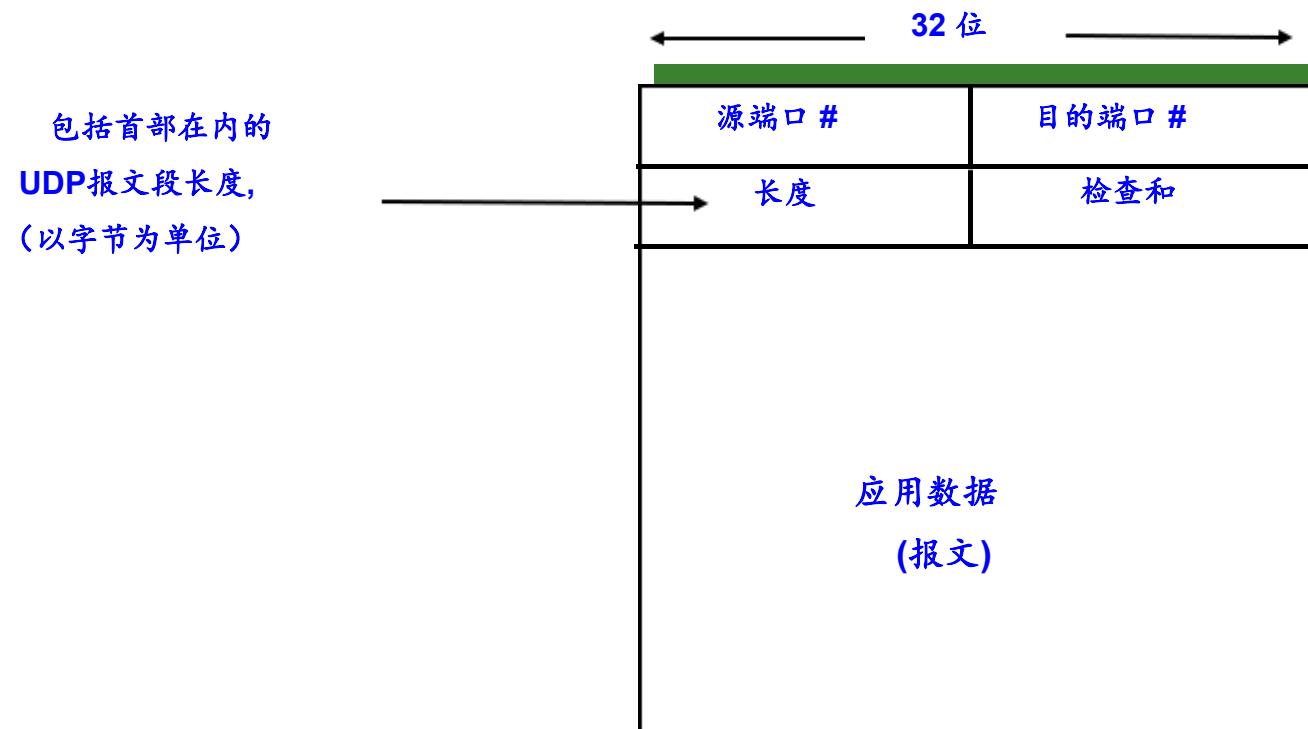
- 路由器中大量的分组溢出
- 显著减小TCP通信的速率，甚至挤垮TCP会话

■ 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

$$\begin{array}{r} 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \\ 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \\ \hline & 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ 1 \ 1 \\ \text{回卷} & \text{和} \\ & \text{检查和} \end{array}$$

1 0 1 1 1 0 1 1 1 0 1 1 1 0 0
0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1

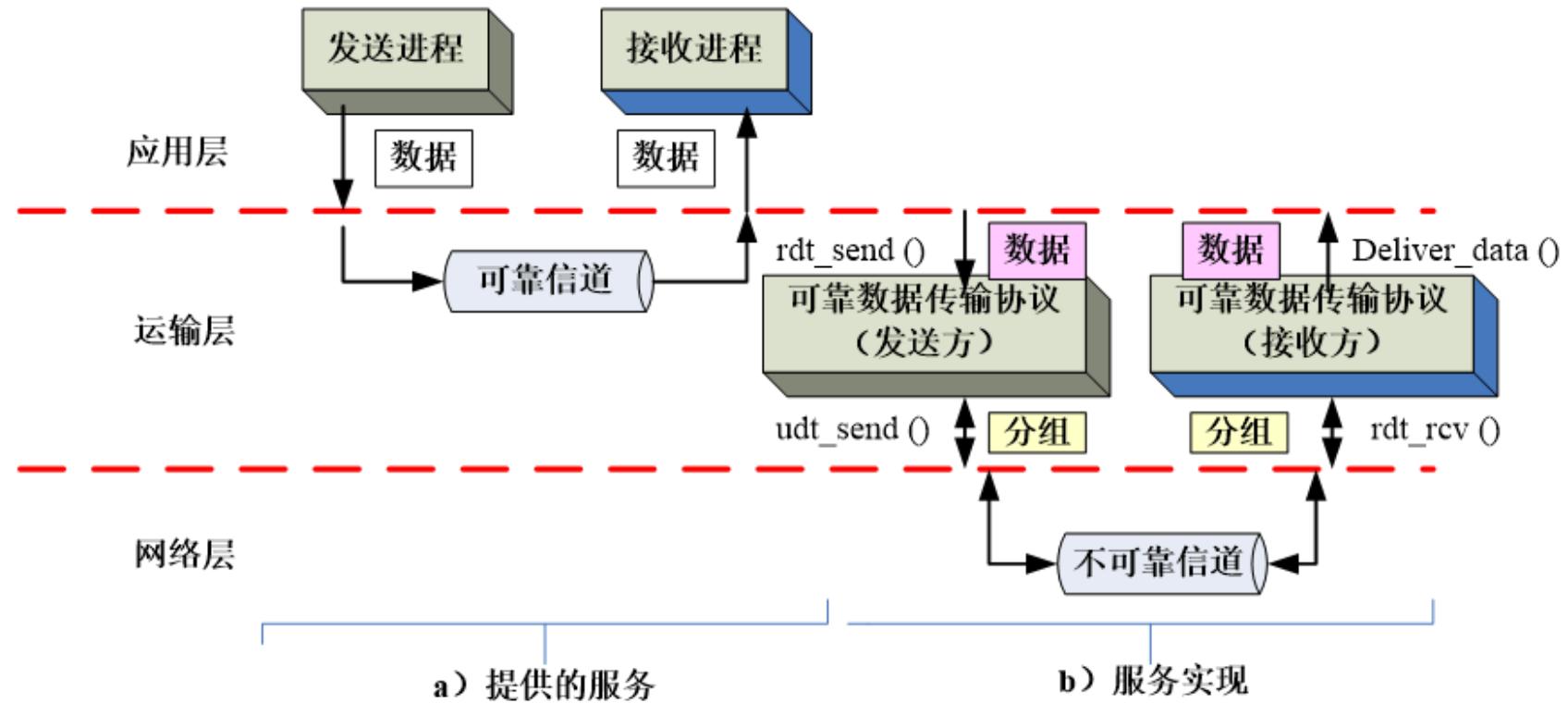
注意：最高有效位的进位要回卷加到结果当中

3.4 可靠数据传输的原理

■ 可靠数据传输

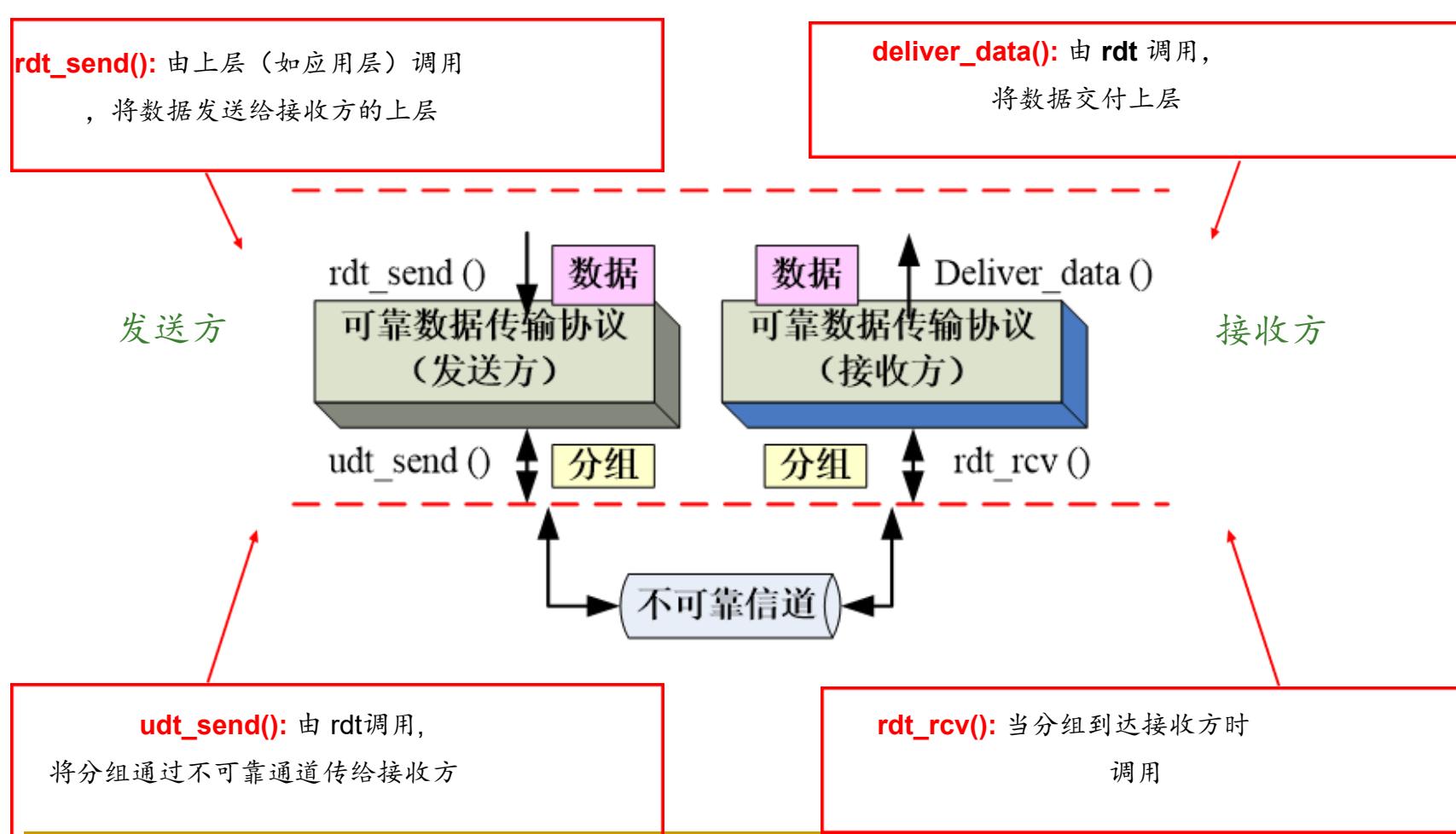
- 在应用层、运输层和链路层都很重要
- 网络中最重要的top-10问题之一!

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

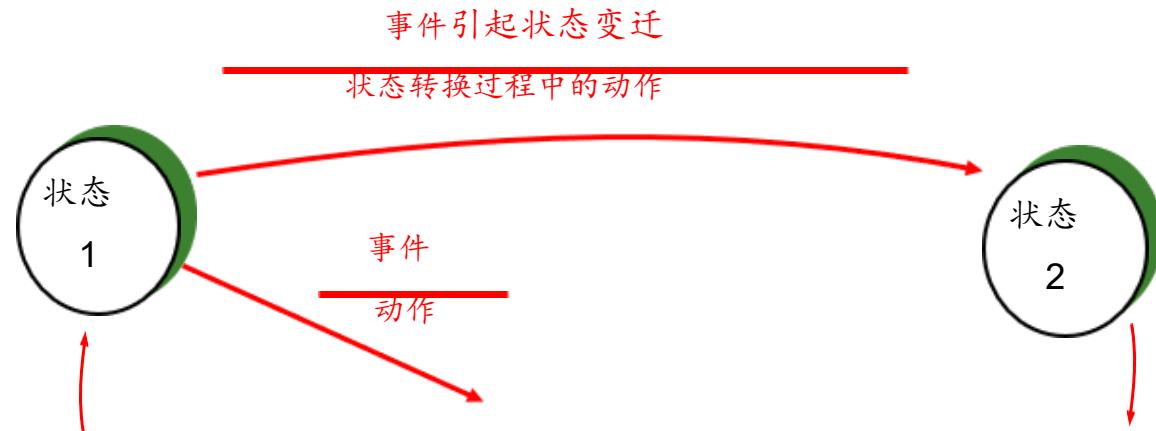


3.4 可靠数据传输的原理

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

■ 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



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

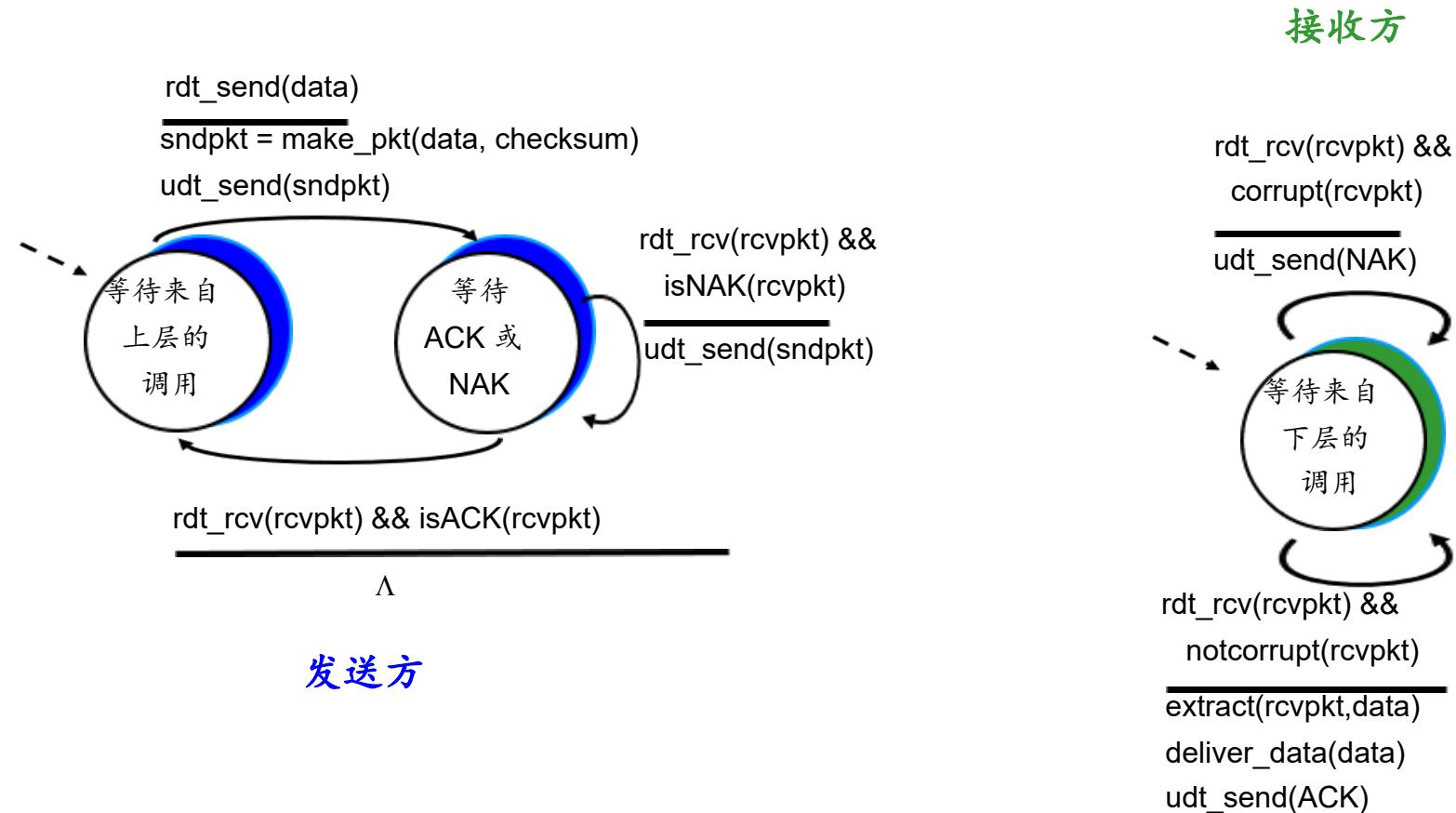
- 分组比特可能受损
- 所有传输的分组都将按序被接收，不会丢失

■ 处理机制

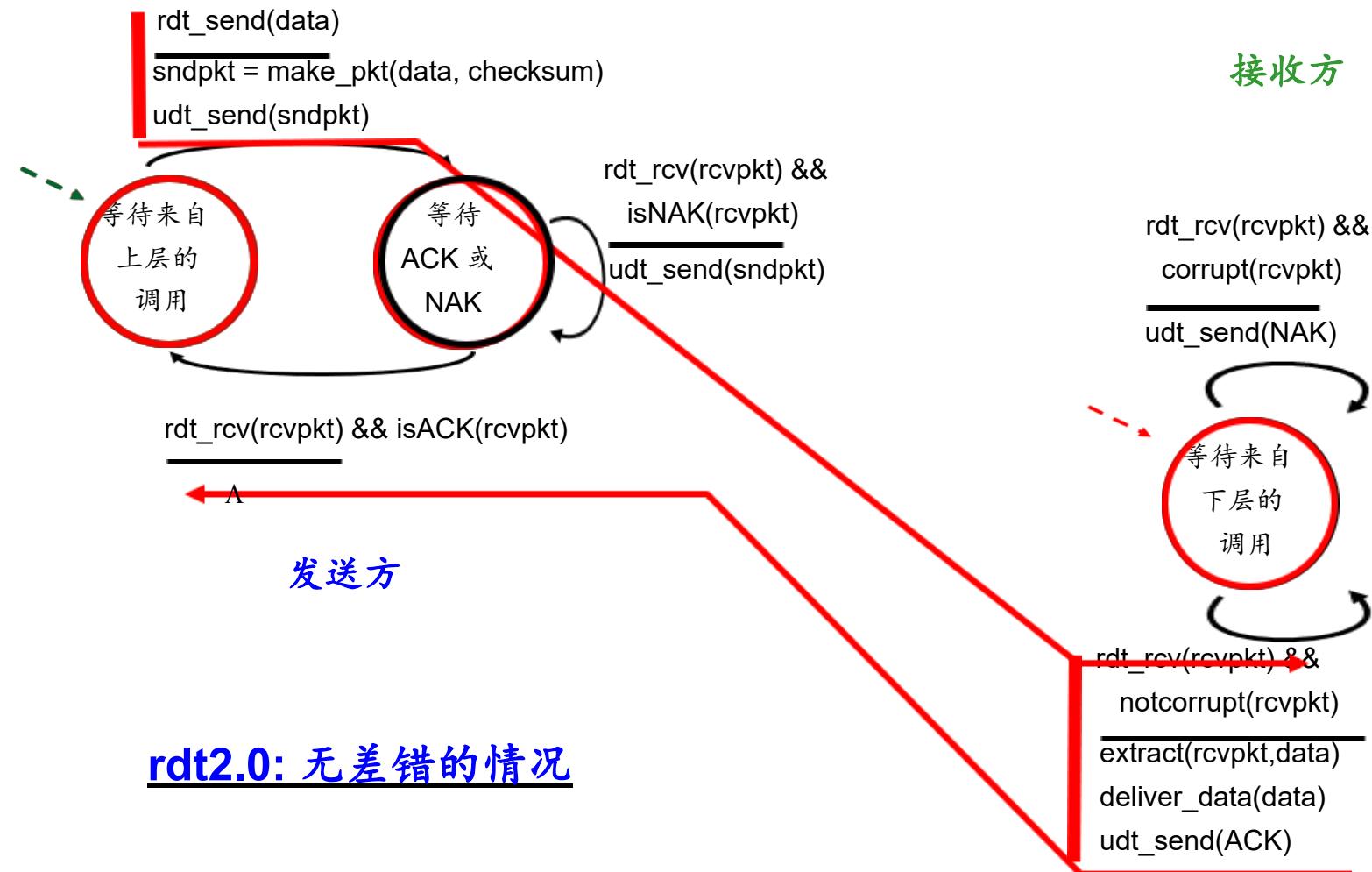
- 如何判断分组受损——差错检测
- 如何通知发送方分组是否受损——接收方反馈（ACK 和 NAK）
- 在得知分组受损后，发送方如何处理——出错重传

3.4 可靠数据传输的原理

■ rdt 2.0的有限状态机FSM



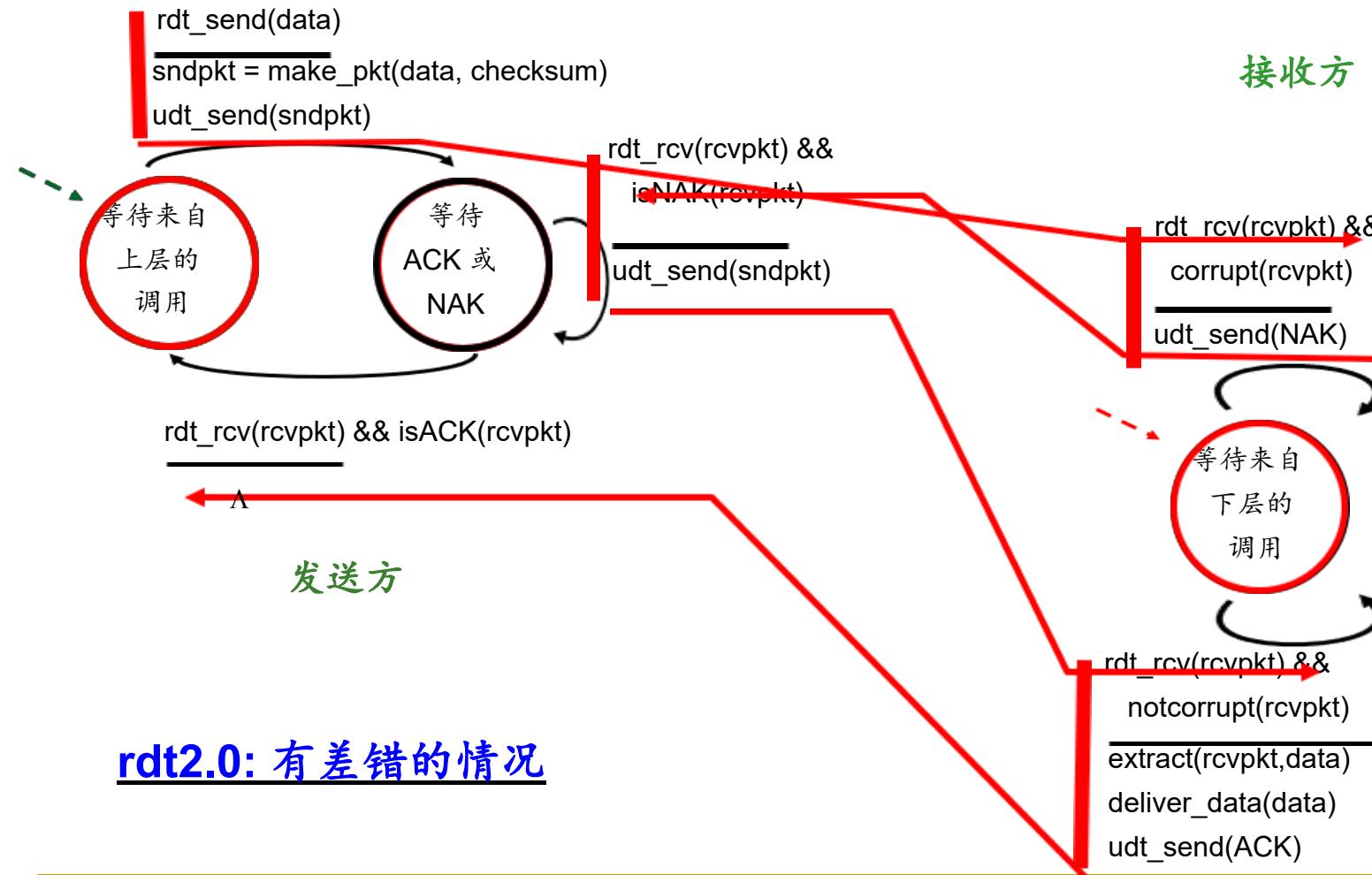
3.4 可靠数据传输的原理



2020年10月6日

28

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

3.4 可靠数据传输的原理

□ 第二个版本——rdt 2.1

■ 问题的引入

- ACK和NAK分组可能受损，而rdt 2.0没有考虑该情况

■ 解决问题的几种思路

- 在人类的对话中，如果听不清楚对方所述，会回问一句

“刚才你说什么来着？”但如果这句话仍然没有听清楚呢？怎么办？双方对着问“刚才你说什么来着？”这就可能进入了一个难以解困的死循环

- 增加足够的检查和比特，使发送方不仅可以检查比特差

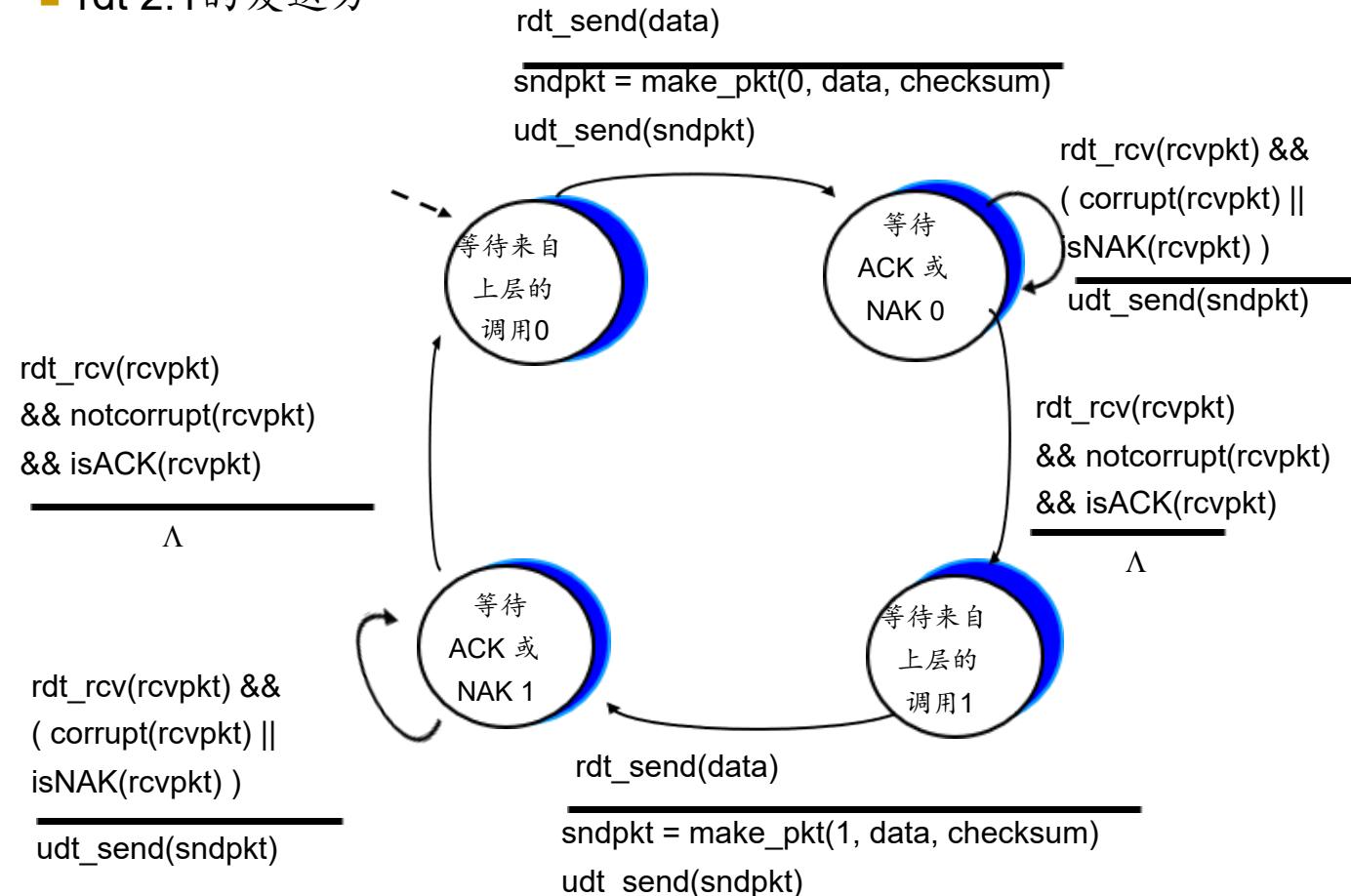
错，还可以恢复比特差错

- 收到出错的反馈时，不管三七二十一，直接重发当前数

据分组，但这就需要对数据分组进行编号，以示识别

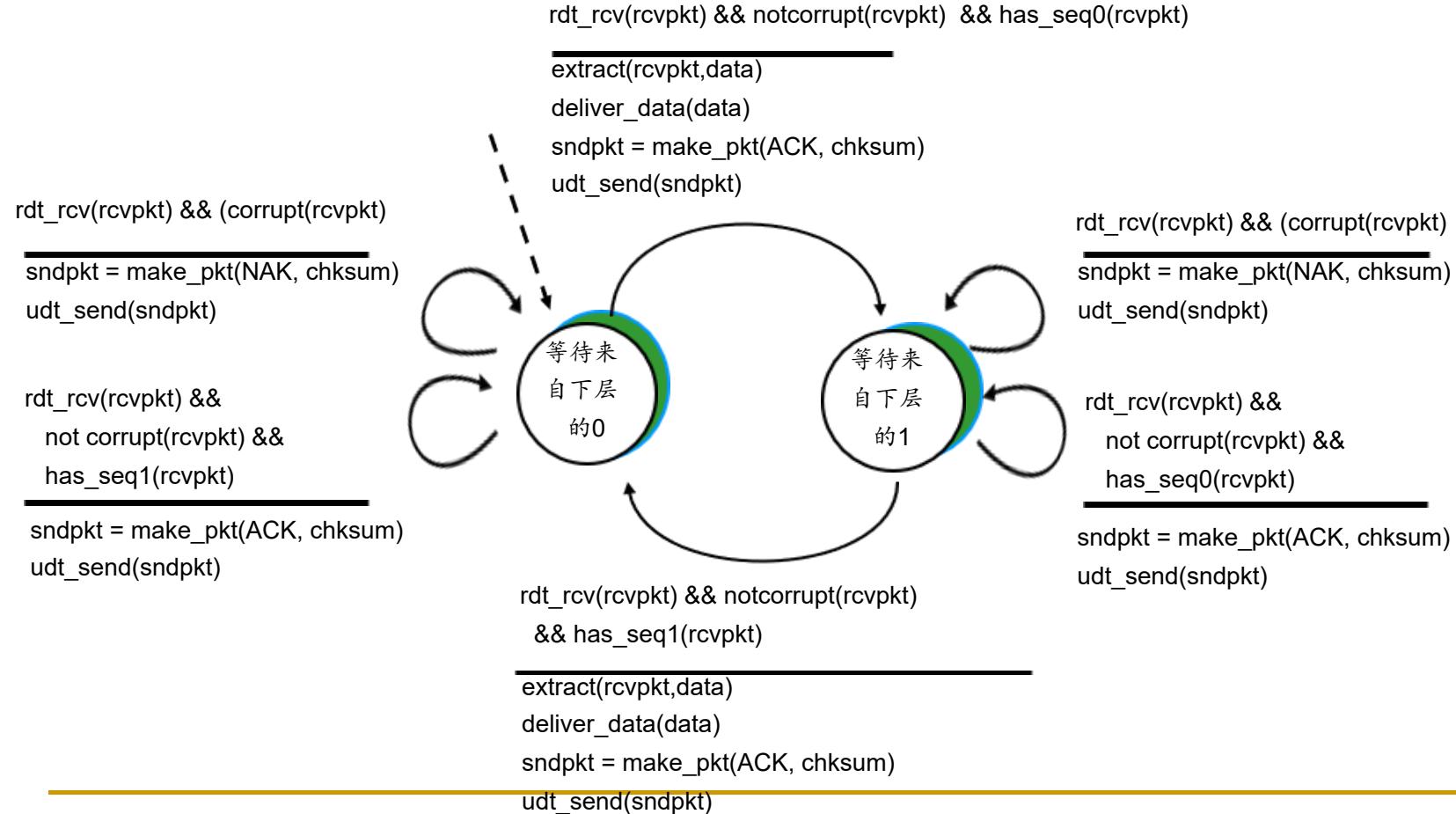
3.4 可靠数据传输的原理

■ 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



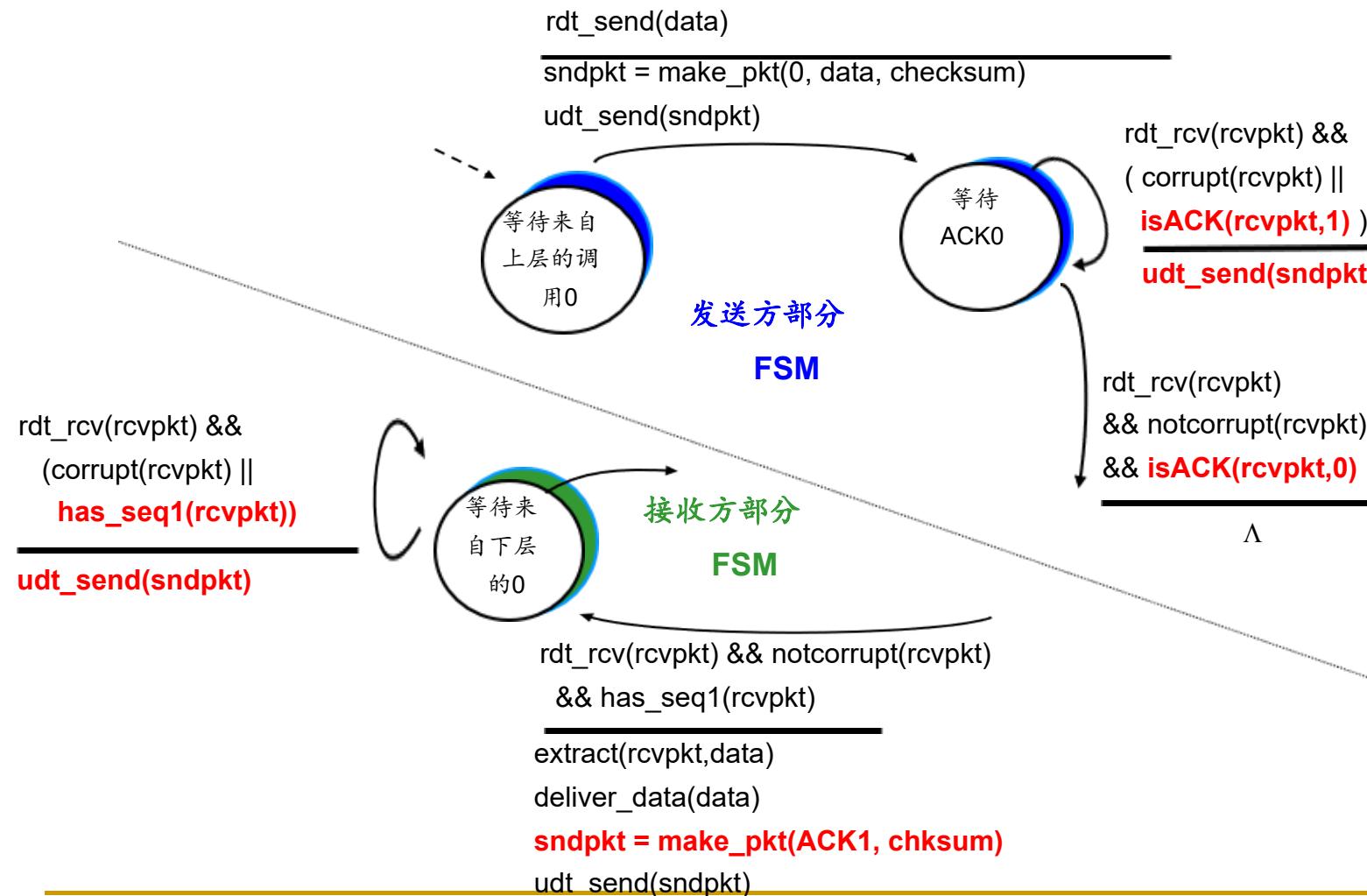
2020年10月6日

33

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



2020年10月6日

35

3.4 可靠数据传输的原理

- 针对rdt 2.x的进一步讨论
 - rdt 2.x实际上也解决了传说中的流控问题

3.4 可靠数据传输的原理

■ 信道不但出错，而且丢包时——rdt 3.0

- 假设

- 底层信道不但可能出现比特差错，而且可能会丢包

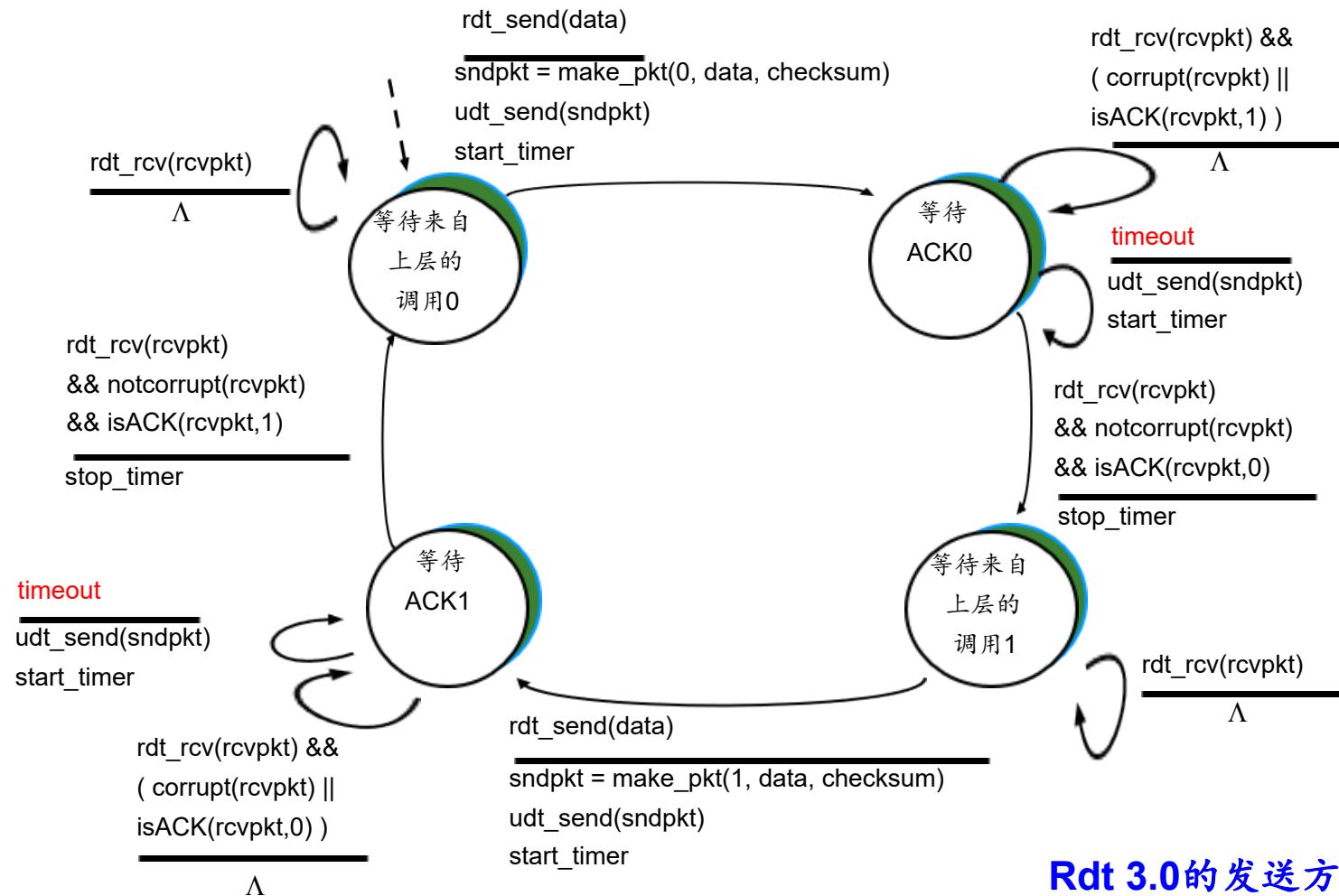
- 需解决的问题

- 怎样检测丢包
 - 发生丢包后，如何处理
 - 检查和技术、序号、ACK、重传

如何判断数据报丢失了呢？

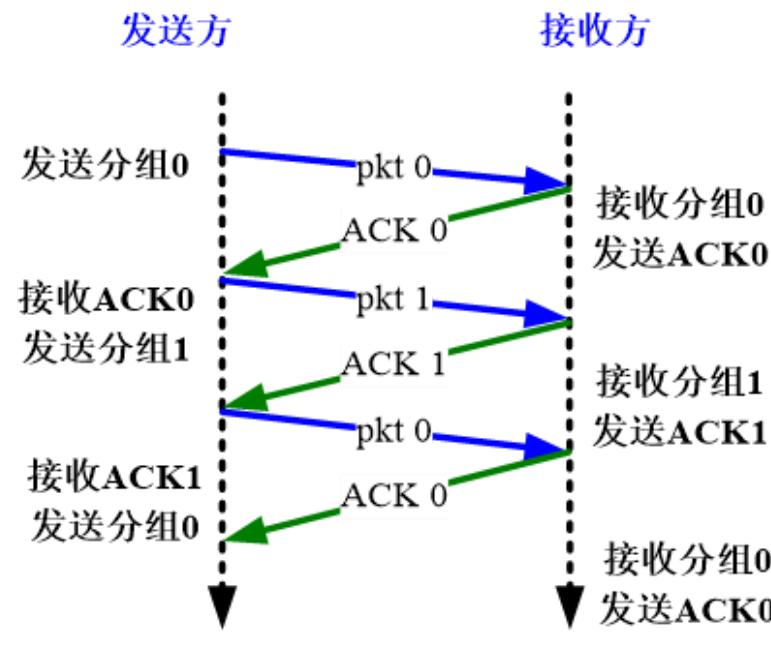
最简单的方法就是：耐心的等待！

3.4 可靠数据传输的原理

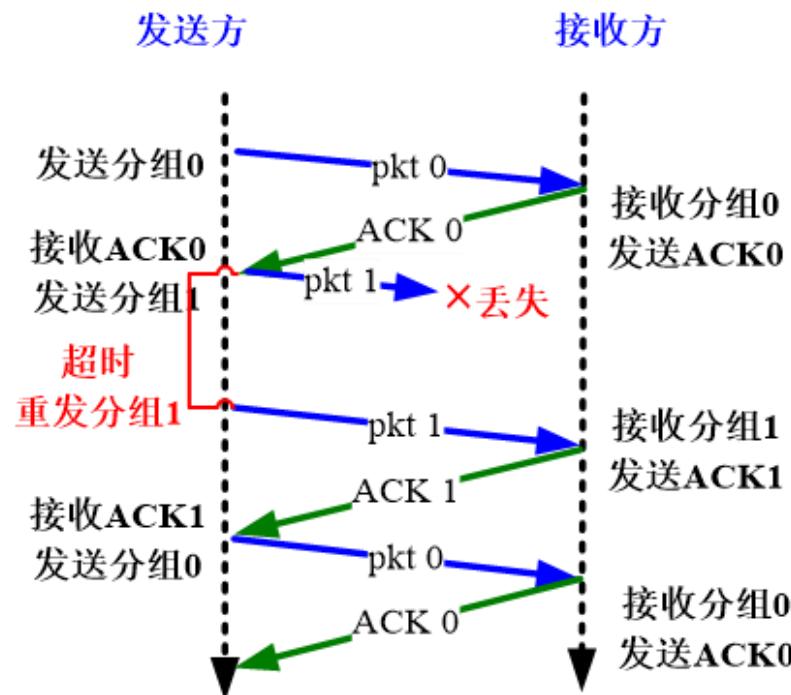


3.4 可靠数据传输的原理

□ rdt 3.0举例



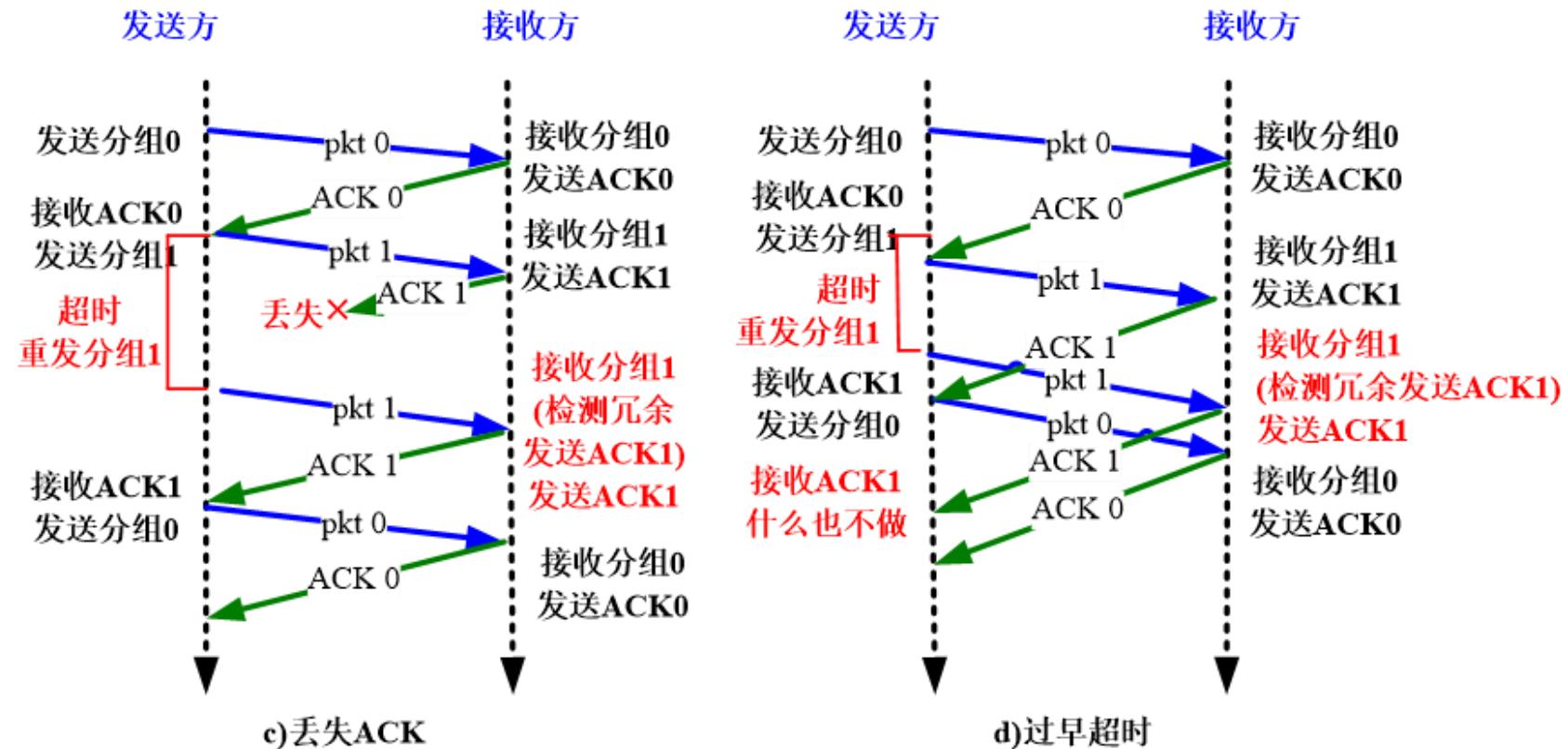
a)无丢包操作



b)分组丢失

下续

3.4 可靠数据传输的原理



3.4 可靠数据传输的原理

■ rdt 3.0的性能分析

- 1Gbps 的链路, 15ms 的端到端延迟, 分组大小为1KB

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

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

- 每30ms内只能发送1KB : 1 Gbps 的链路只有33kB/sec 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

首个分组的第一个比特被传输,

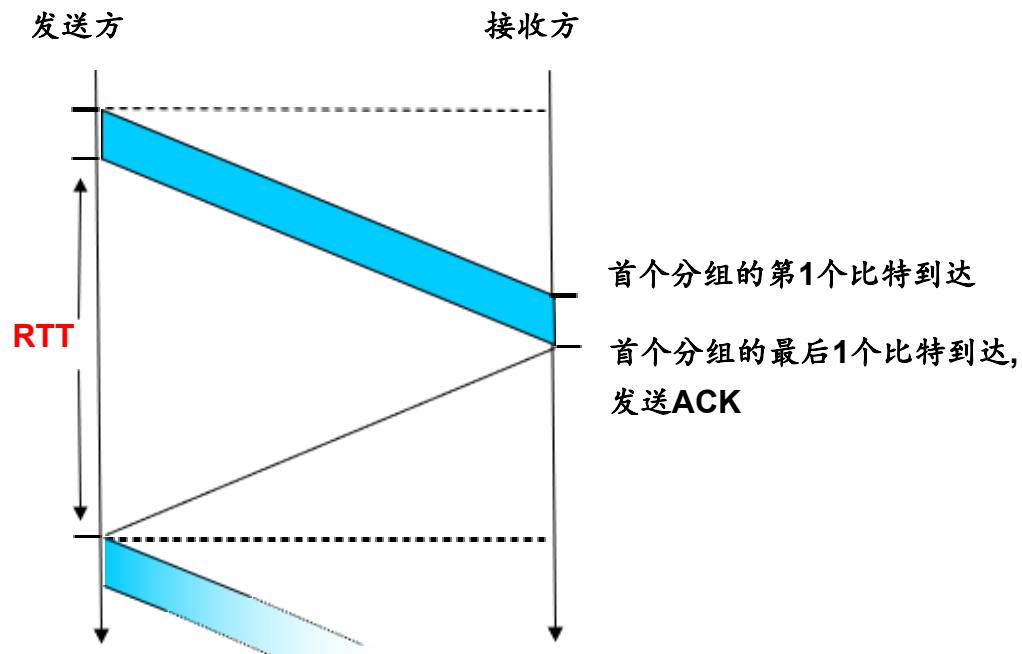
$t = 0$

首个分组的最后一个比特被传输,

$t = L/R$

ACK 到达, 发送下一个分组,

$t = RTT + L/R$



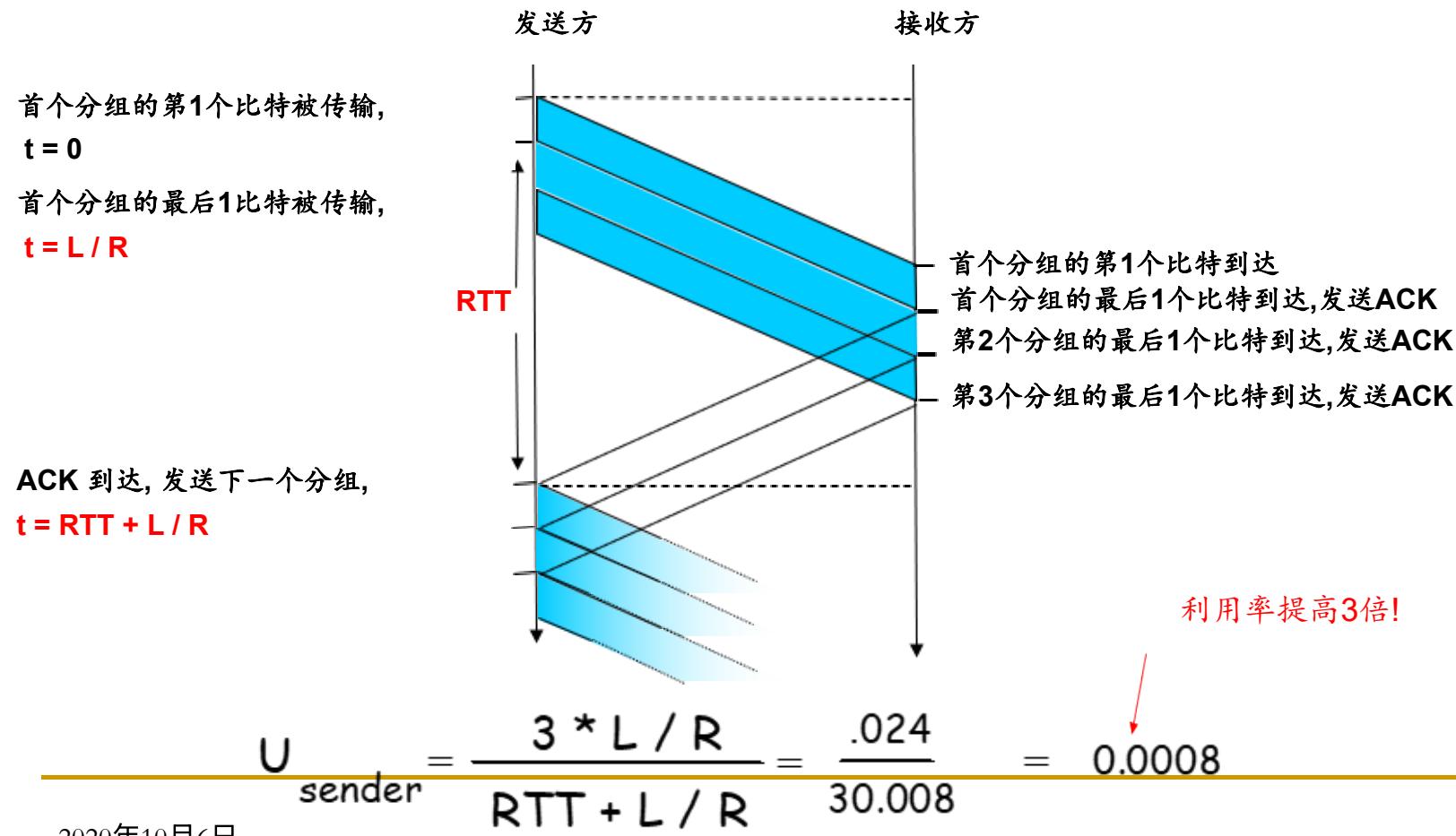
$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

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

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



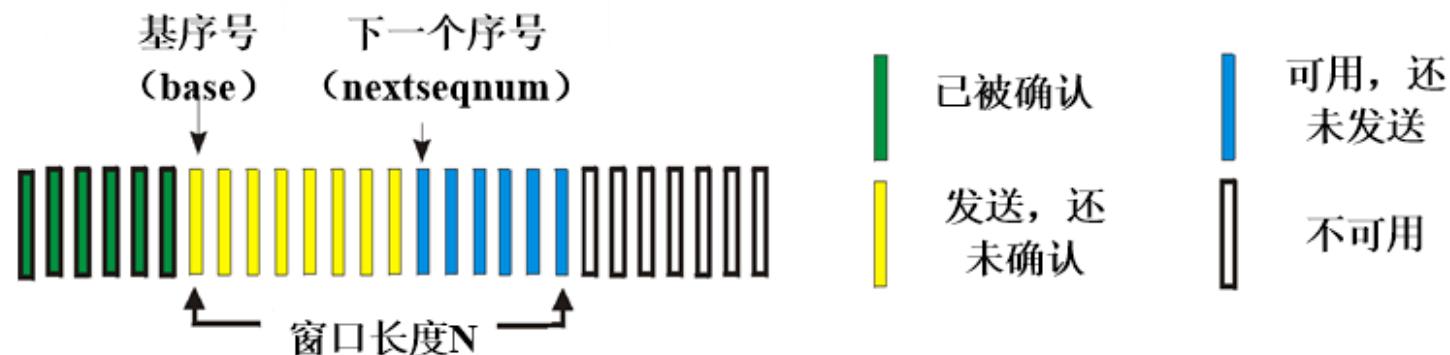
2020年10月6日

44

3.4 可靠数据传输的原理

■ 流水线技术工作原理

- 分组首部用 k -比特字段表示序号
- 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)”

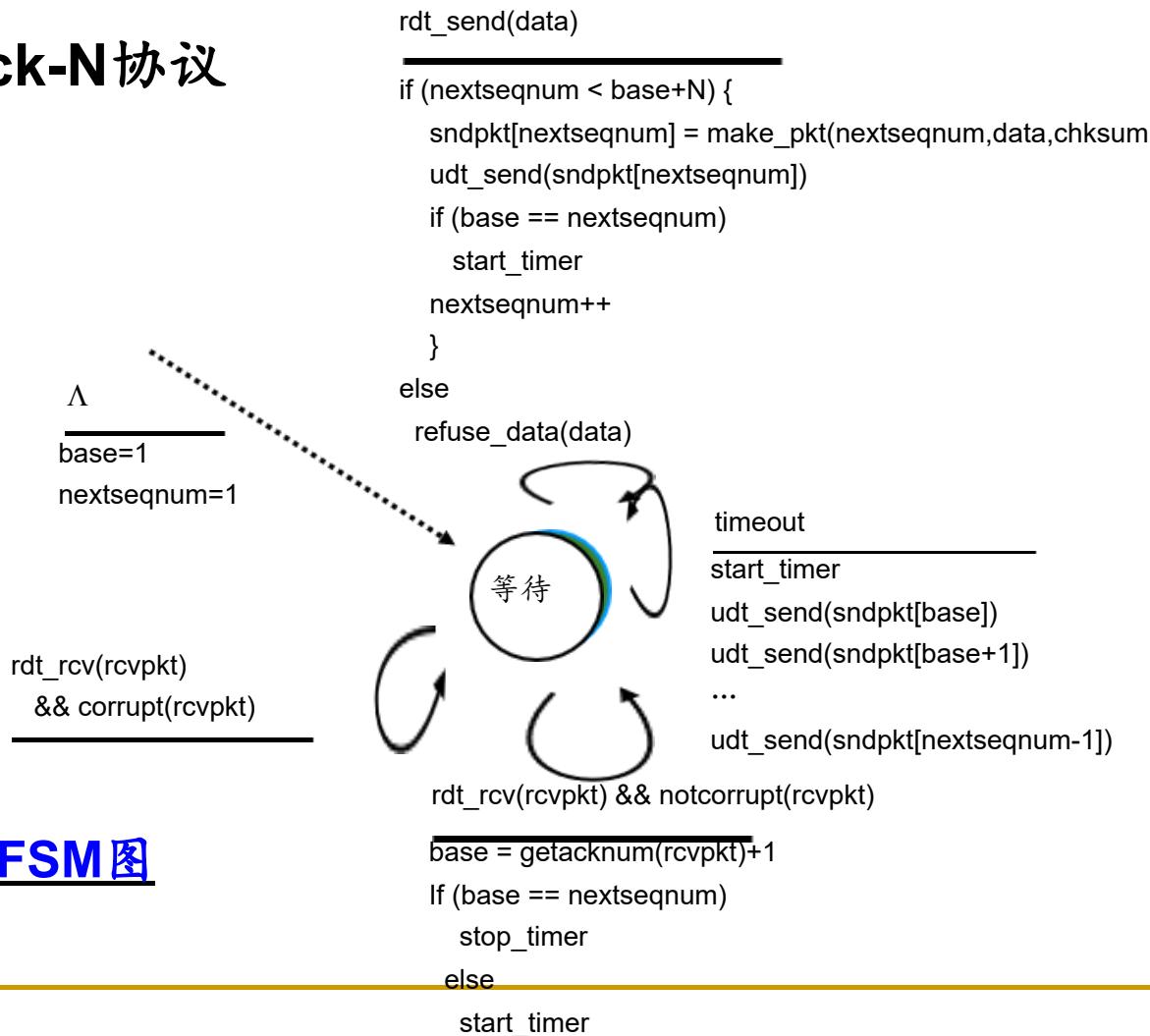


3.4 可靠数据传输的原理

- 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) 协议：其后分组全部重传
 - 选择重传 (SR) 协议：仅重传该分组

3.4 可靠数据传输的原理

■ Go-Back-N协议

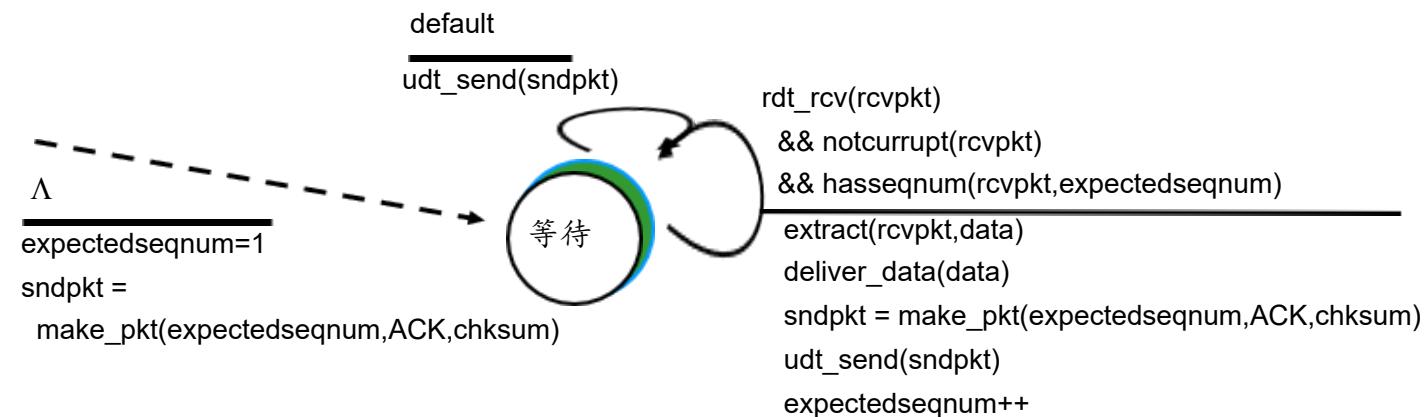


2020年10月6日

47

3.4 可靠数据传输的原理

■ Go-Back-N协议



接收方的FSM图

3.4 可靠数据传输的原理

发送窗口($N=4$)

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

发送方

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

rcv ack0, send pkt4
rcv ack1, send pkt5

接收方

receive pkt0, send ack0
receive pkt1, send ack1

receive pkt3, discard,
(re)send ack1

receive pkt4, discard,
(re)send ack1

receive pkt5, discard,
(re)send ack1



pkt 2 超时

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

send pkt2
send pkt3
send pkt4
send pkt5

X loss

忽略重复ACK

rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

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
- 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

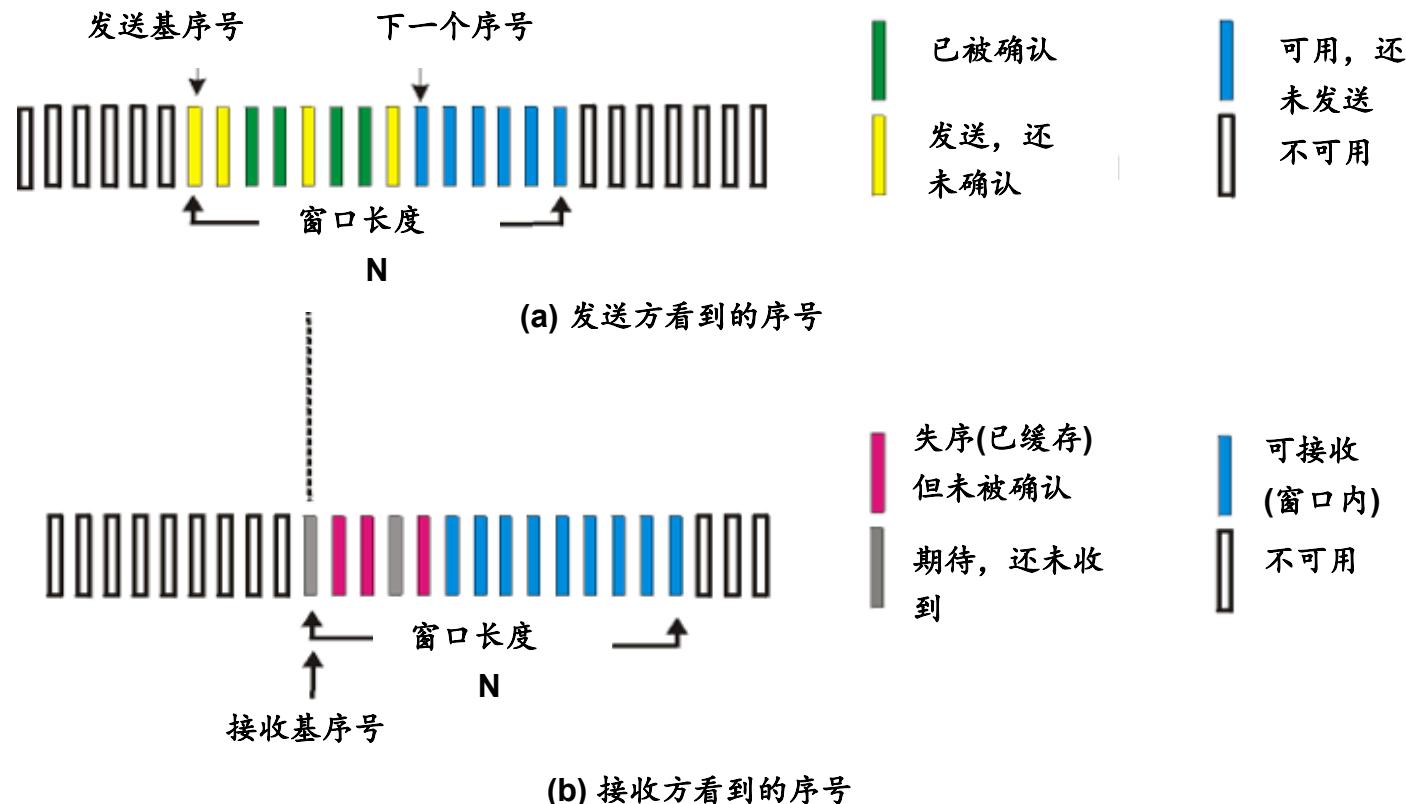
3.4 可靠数据传输的原理

■ Go-Back-N的滑动窗口大小

- 发送端 $\leq 2^k - 1$
- 接收端 = 1

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

超时(n)：

- 重传分组 n , 重置定时器

收到确认(n) 在 $[sendbase, sendbase+N]$ 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

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

- 发送n的确认ACK(n)
- 如果分组序号不连续(失序): 将其缓存
- 按序分组: 将该分组以及以前缓存的序号连续的分组一起交付给上层, 将窗口前推到下一个未收到的分组

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)

其他情况: 忽略该分组

3.4 可靠数据传输的原理

发送窗口(N=4)

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

发送方

send pkt0

send pkt1

send pkt2

send pkt3

(wait)

rcv ack0, send pkt4

rcv ack1, send pkt5

记录ack3到达

pkt 2 超时
send pkt2

记录ack4到达

记录ack5到达

接收方

receive pkt0, send ack0

receive pkt1, send ack1

receive pkt3, buffer,
send ack3

receive pkt4, buffer,
send ack4

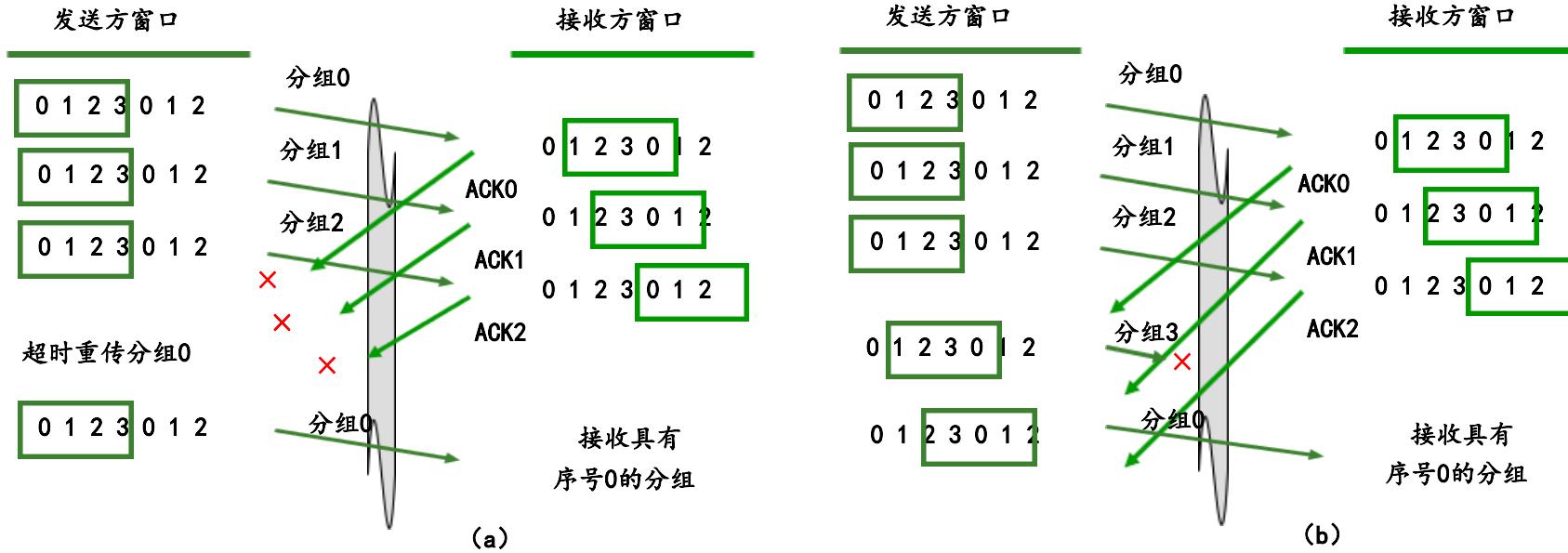
receive pkt5, buffer,
send ack5

rcv pkt2; deliver pkt2,
pkt3, pkt4, pkt5; send ack2



Q: ack2到达后发送方怎么处理?

3.4 可靠数据传输的原理



结论：接收方窗口 $\leq 2^{k-1}$

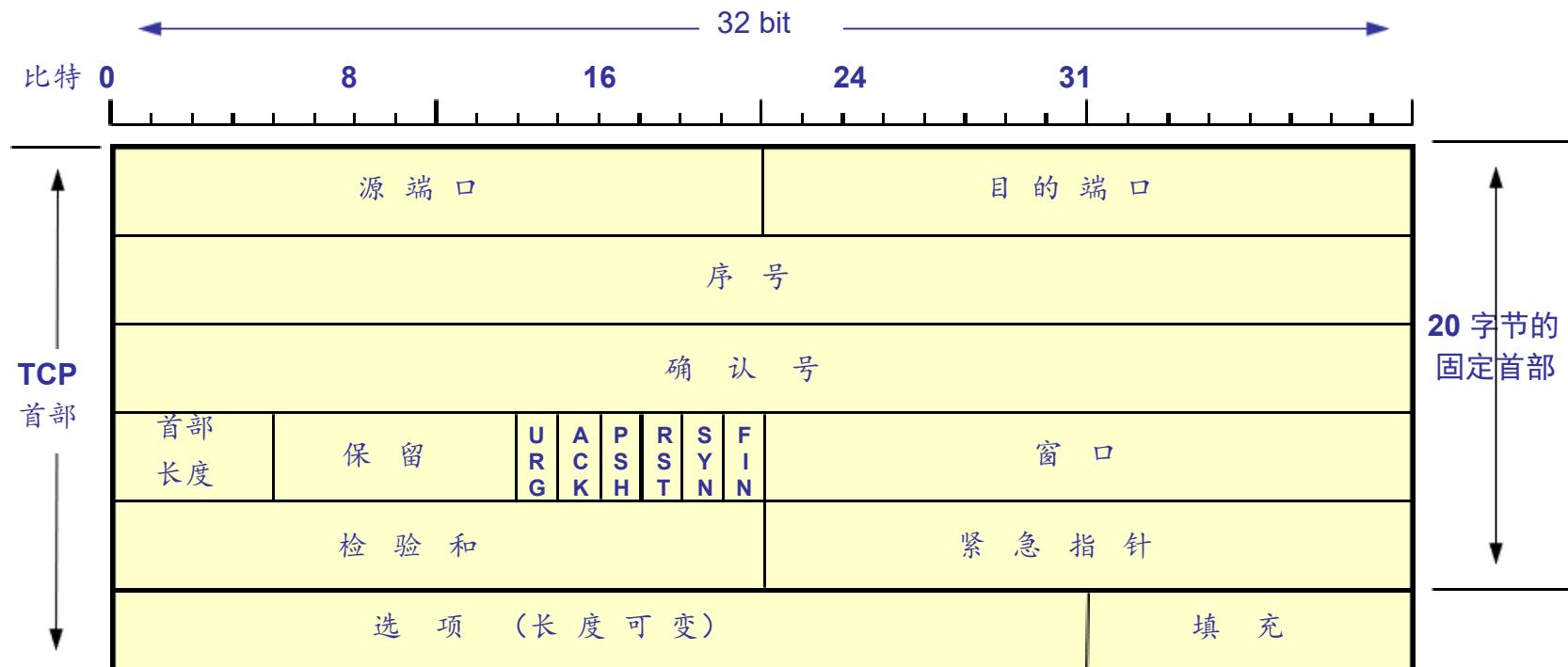
3.5 面向连接的传输：TCP

■ TCP概述—RFCs: 793、1122、1323、2018、2581

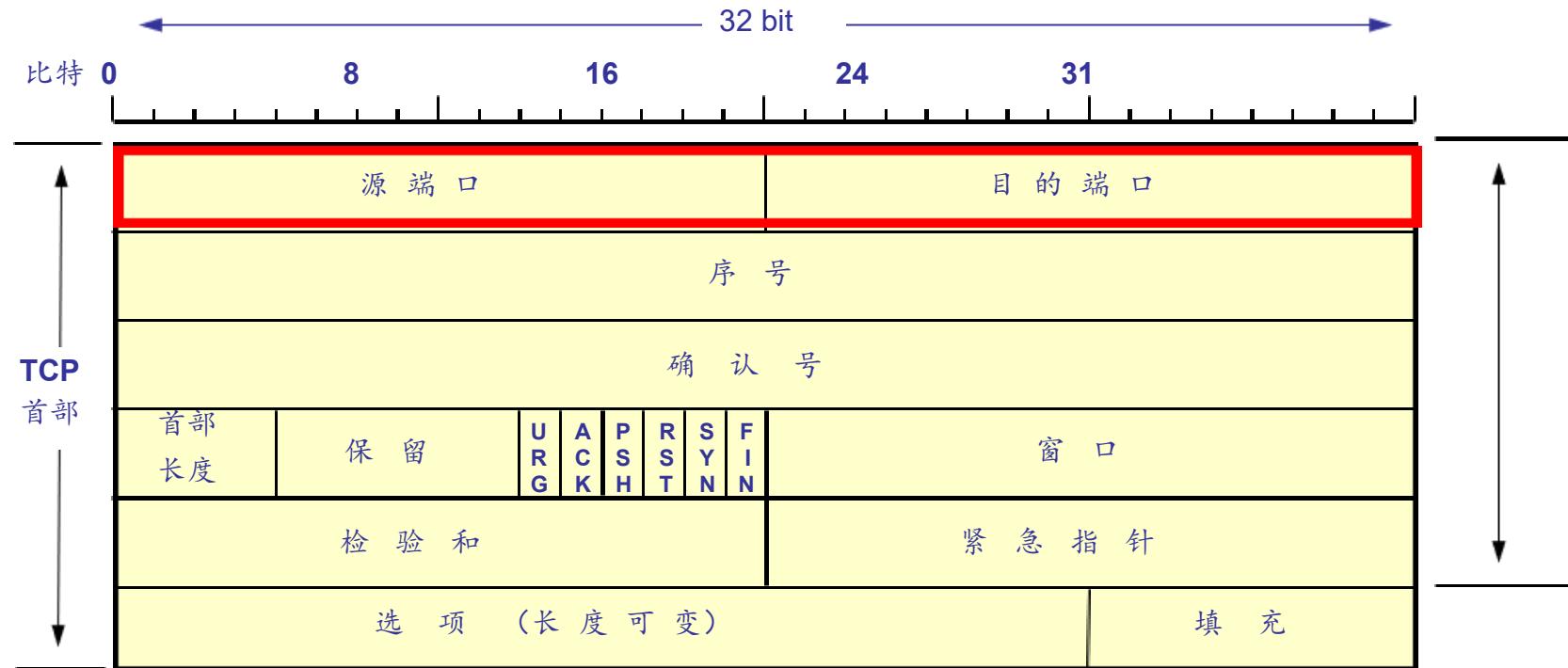
- 面向连接
 - TCP连接仅存于端系统，中间路由器对此毫不知情
- 全双工服务
 - 可双向同时传输数据
- 点对点连接
 - 仅存在于两个端系统之间，无第三者“插足”
- 三次握手
 - 建立连接，协商参数
- 可靠的字节流
 - 最大报文段长MSS

3.5 面向连接的传输：TCP

■ TCP报文段头部结构

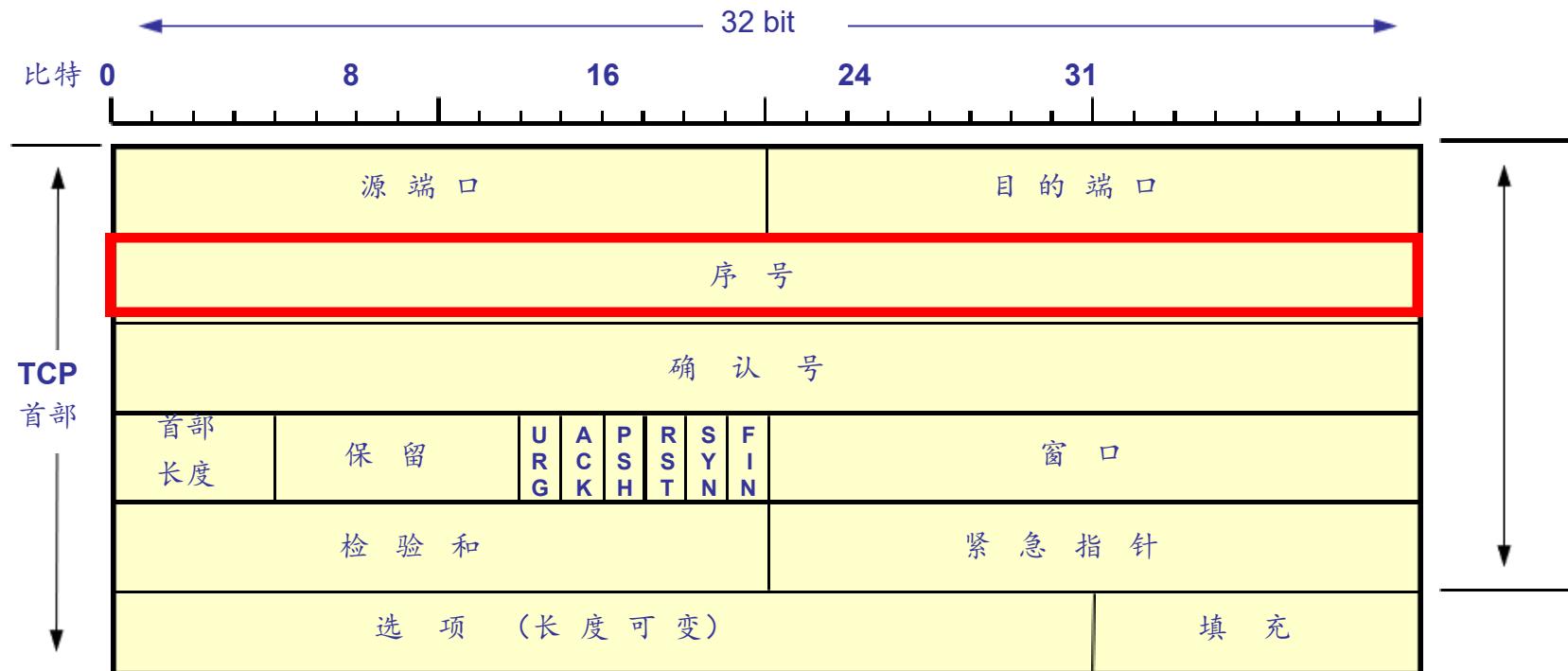


3.5 面向连接的传输：TCP



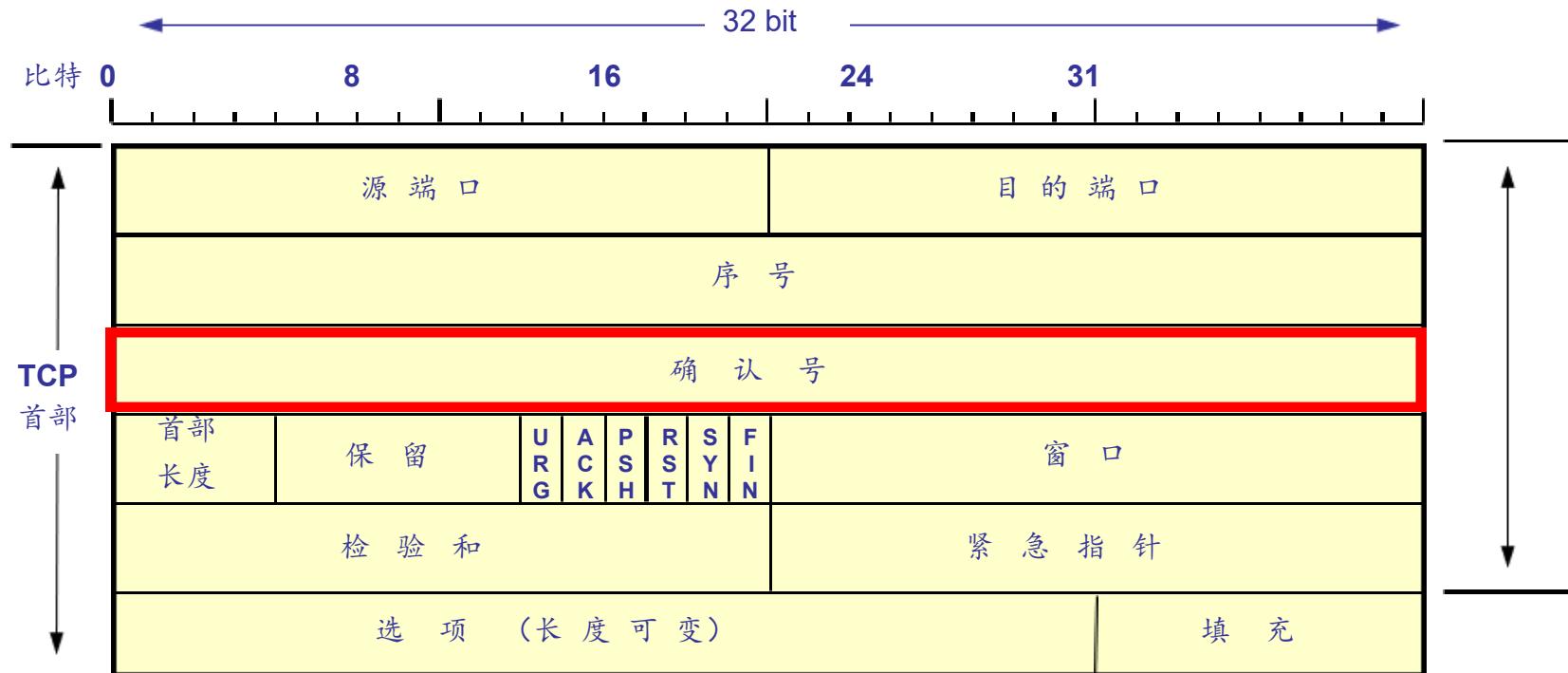
源端口和目的端口字段——各占 2 字节。端口是运输层与应用层的服务接口。运输层的复用和分用功能都要通过端口才能实现。

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。

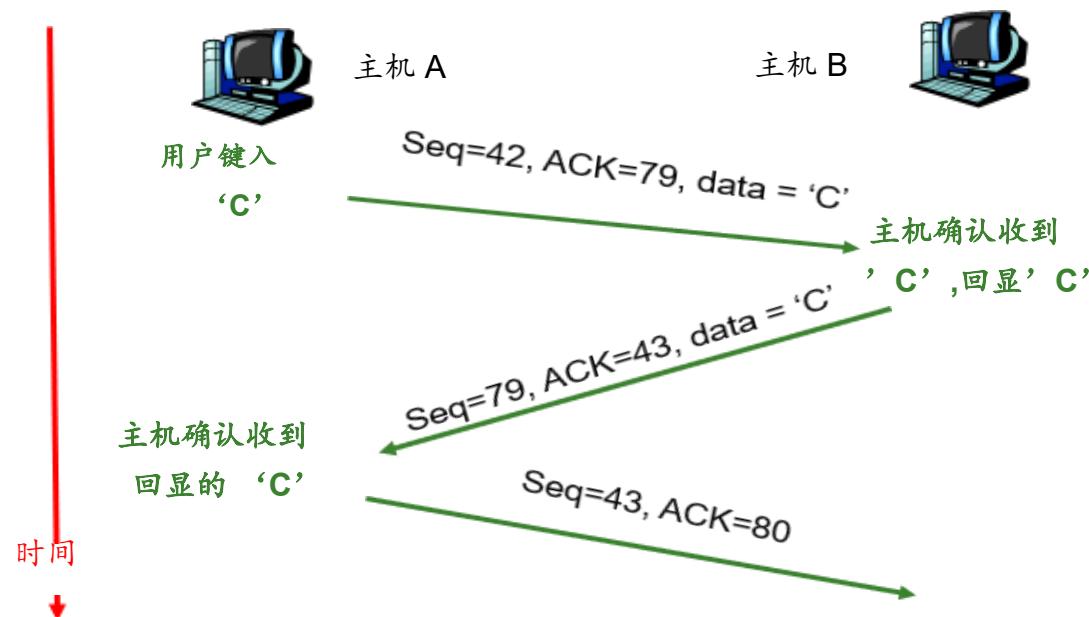
3.5 面向连接的传输：TCP



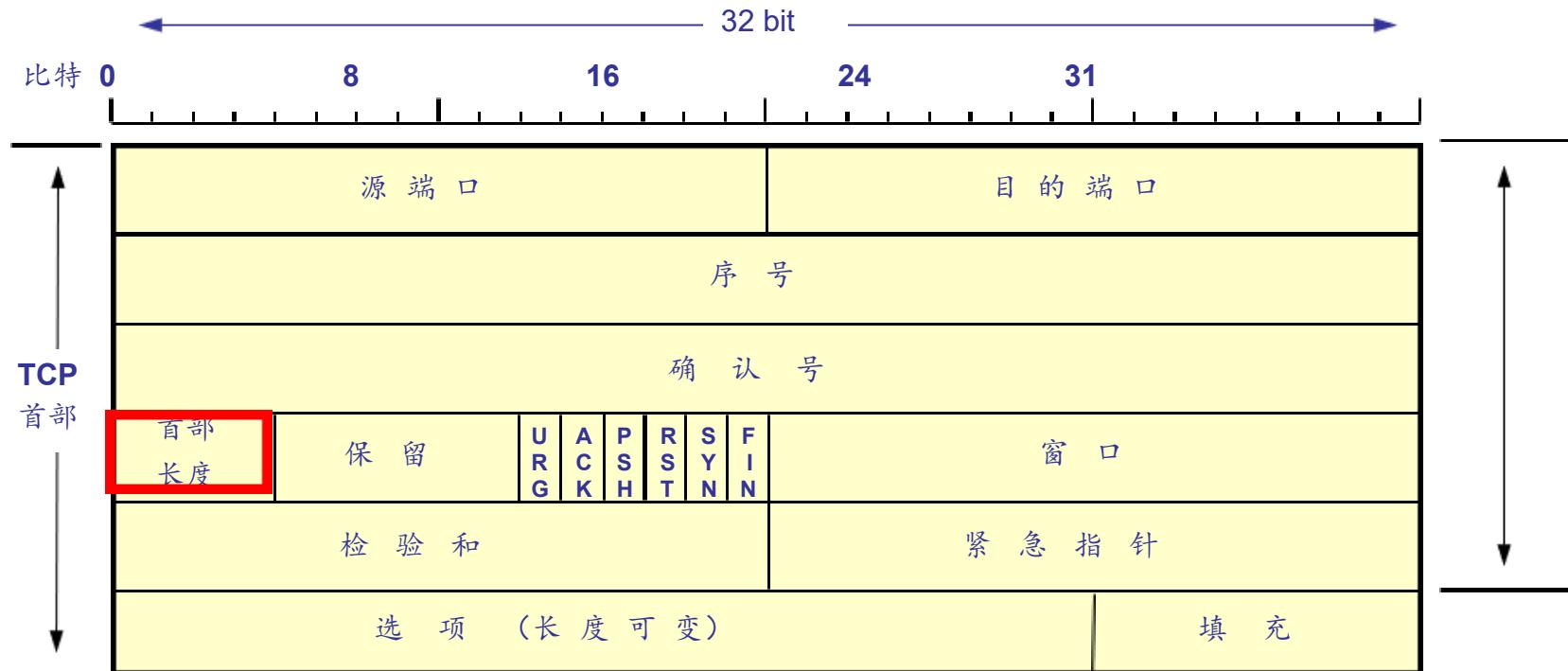
确认号字段——占 4 字节，是期望收到对方的下一个报文段的数据的第一个字节的序号。

3.5 面向连接的传输：TCP

■ TCP序列号和确认序列号

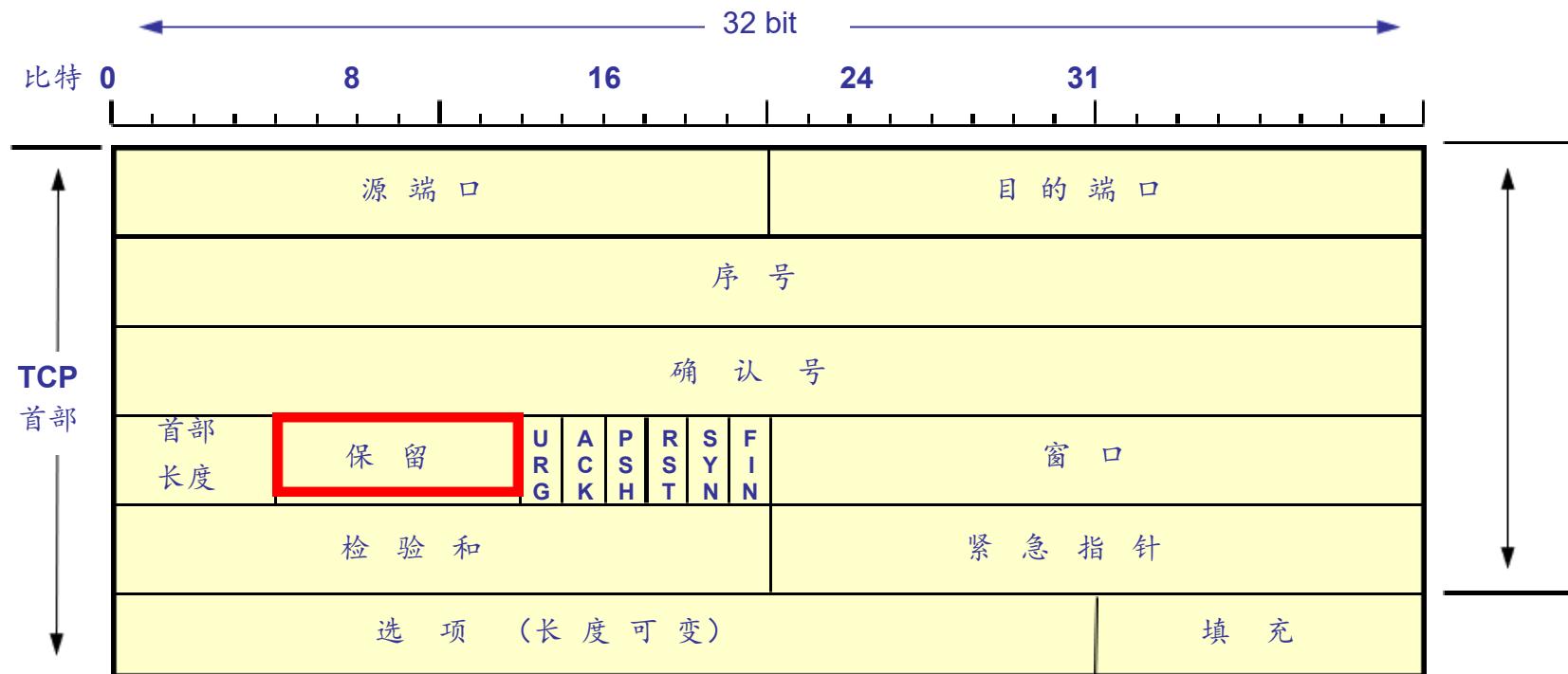


3.5 面向连接的传输：TCP



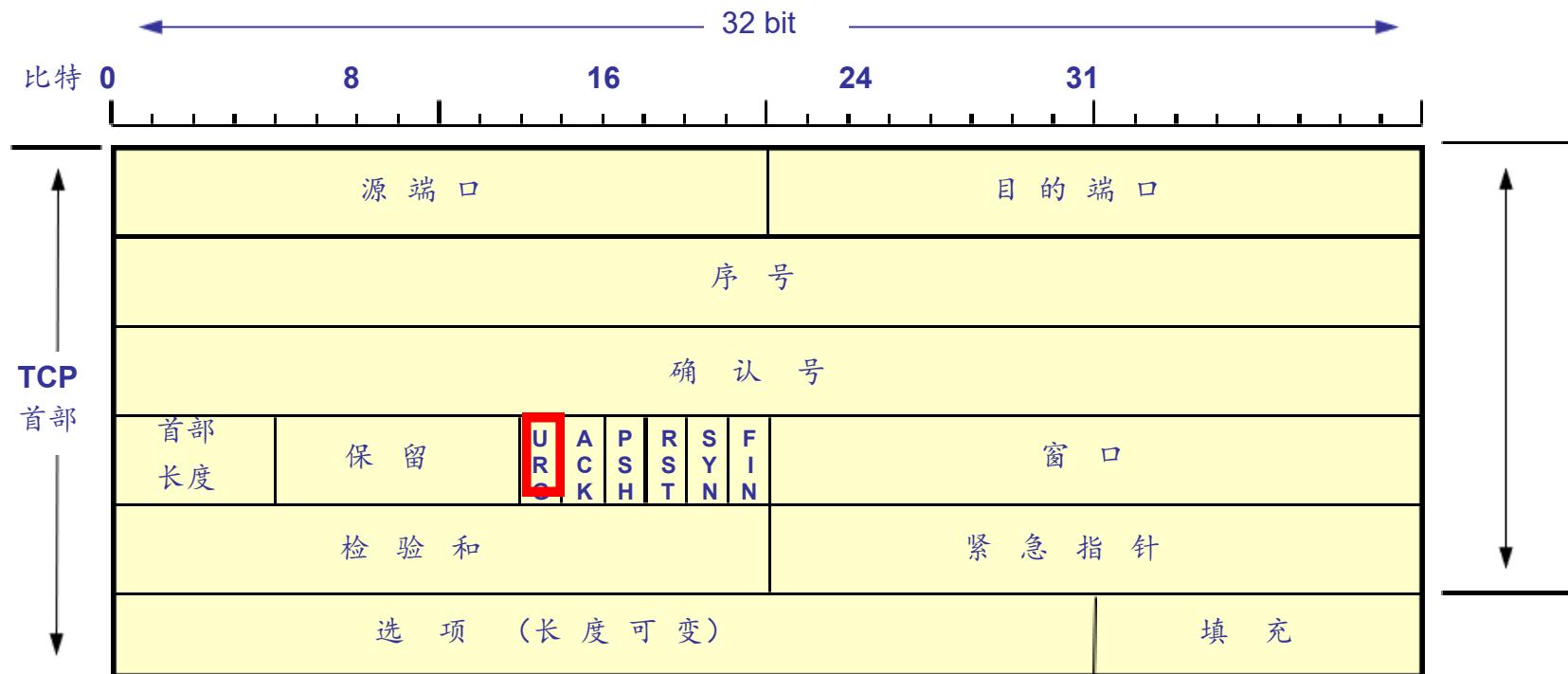
首部长度——占 4 bit，它指示以 32bit 为单位的 TCP 首部长度。

3.5 面向连接的传输：TCP



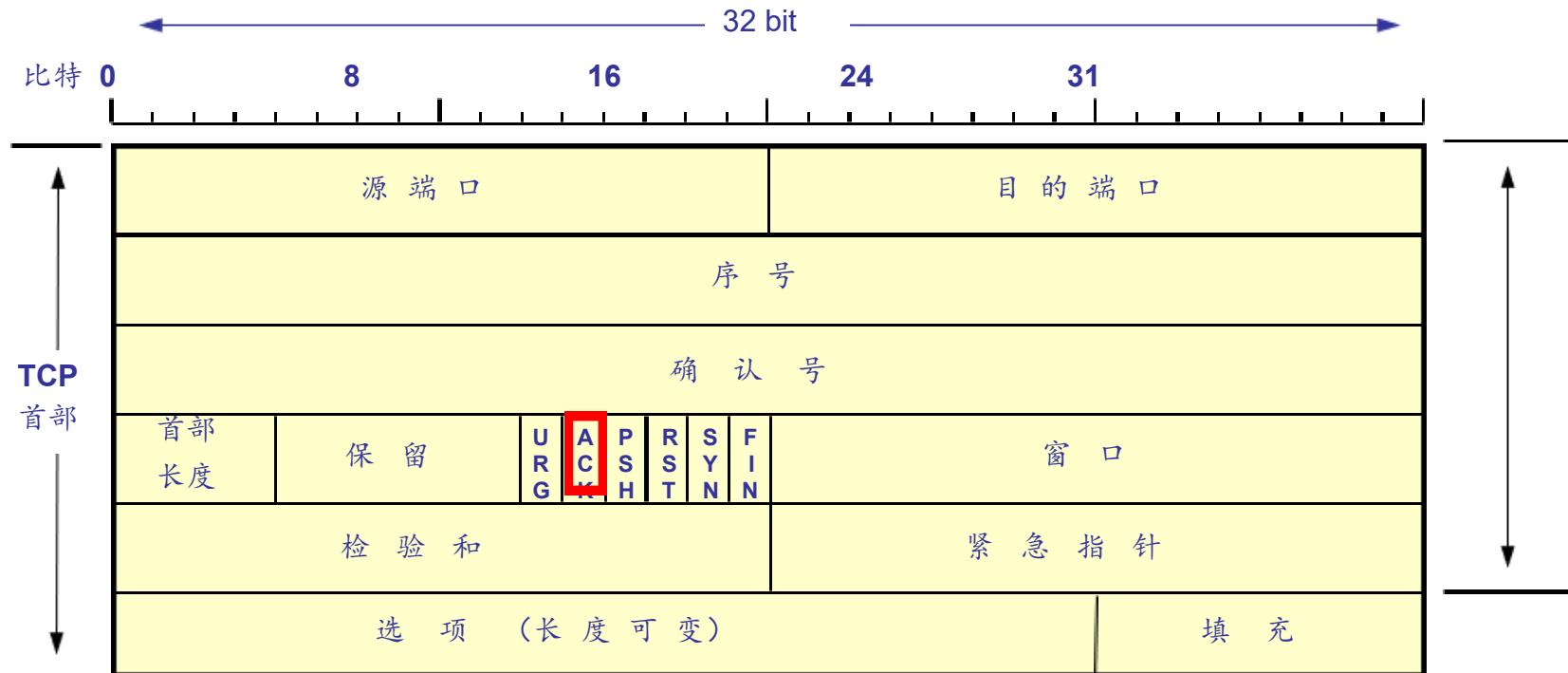
保留字段——占 6 bit，保留为今后使用，但目前应置为 0。

3.5 面向连接的传输：TCP



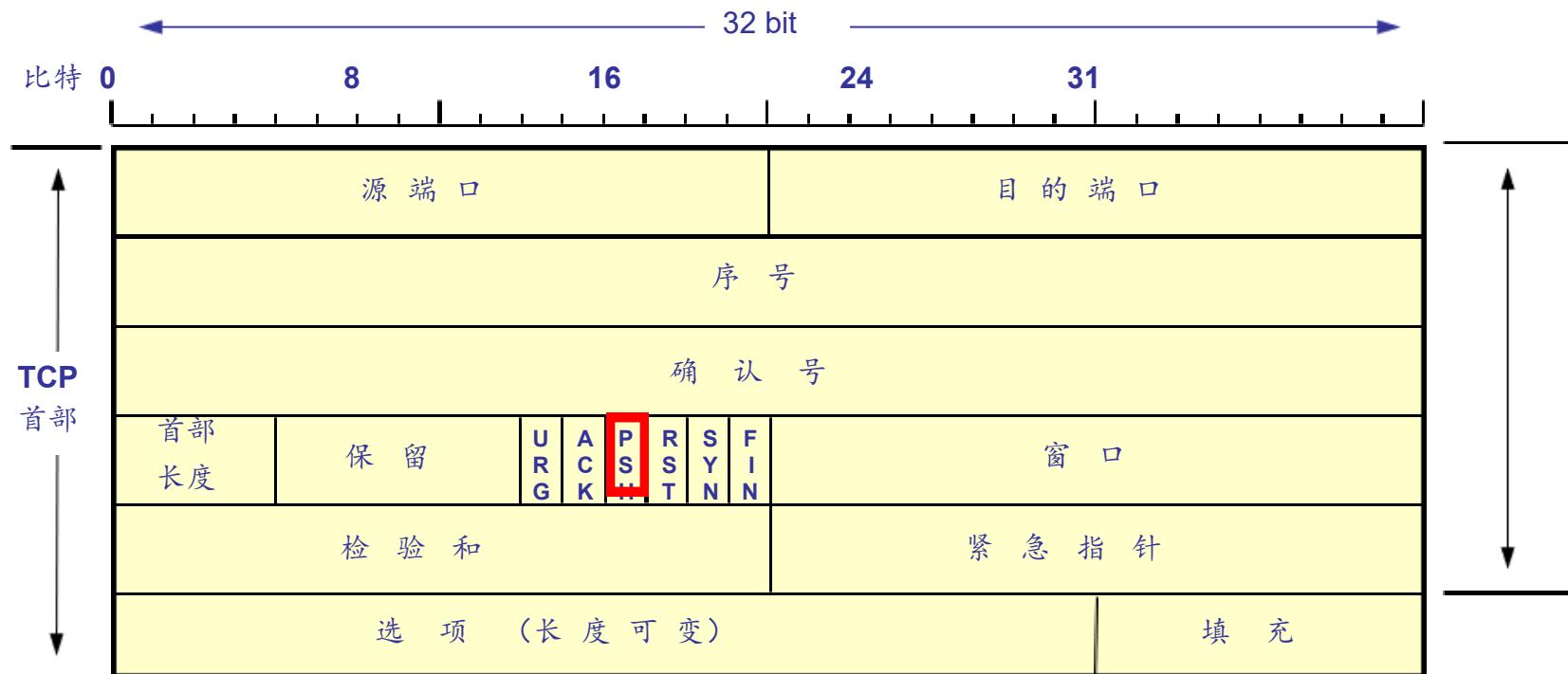
紧急比特 URG —— 当 $URG = 1$ 时，表明紧急指针字段有效。它告诉系统此报文段中有紧急数据，应尽快传送。（一般不使用）

3.5 面向连接的传输：TCP



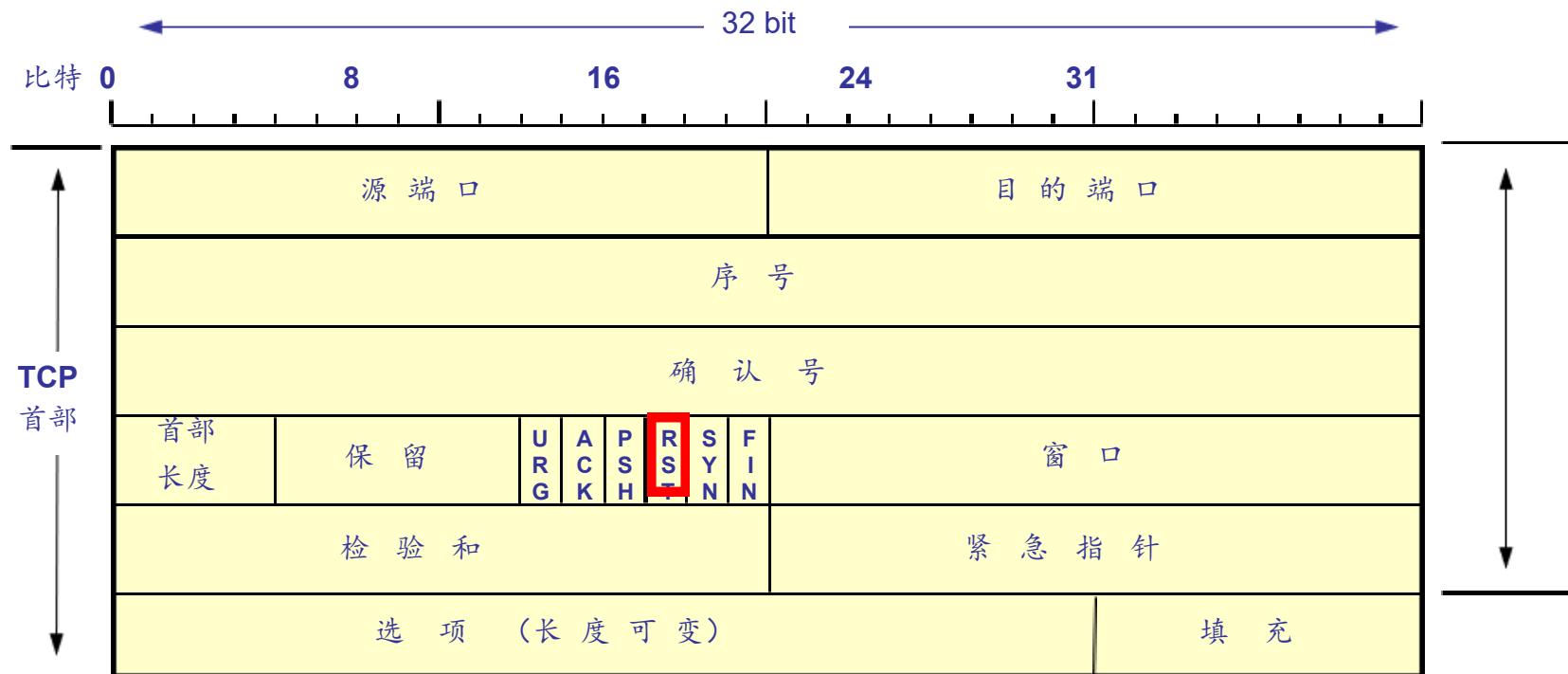
确认比特 ACK —— 只有当 $ACK = 1$ 时确认号字段才有效。
当 $ACK = 0$ 时，确认号无效。

3.5 面向连接的传输：TCP



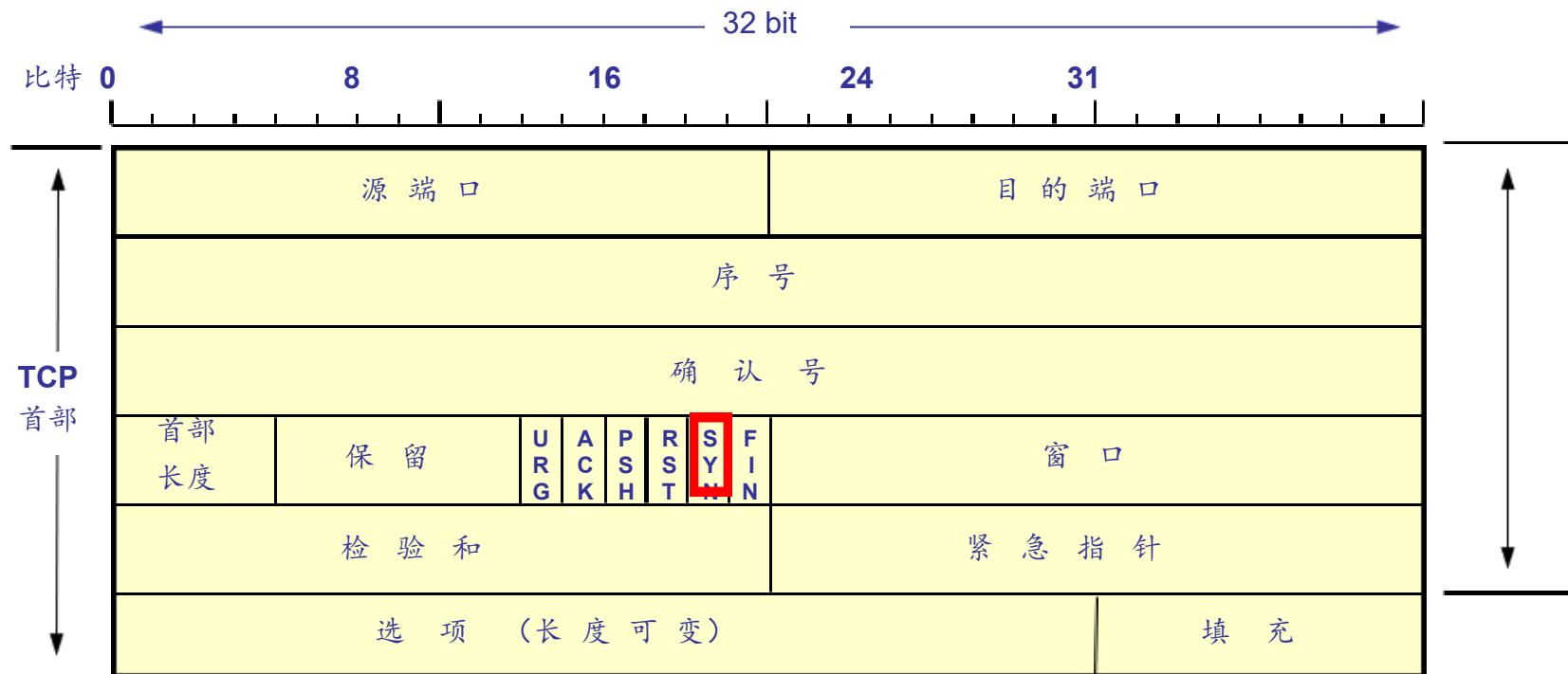
推送比特 PSH (PuSH) —— 接收 TCP 收到推送比特置 1 的报文段，就尽快地交付给接收应用进程，而不再等到整个缓存都填满了后再向上交付。

3.5 面向连接的传输：TCP



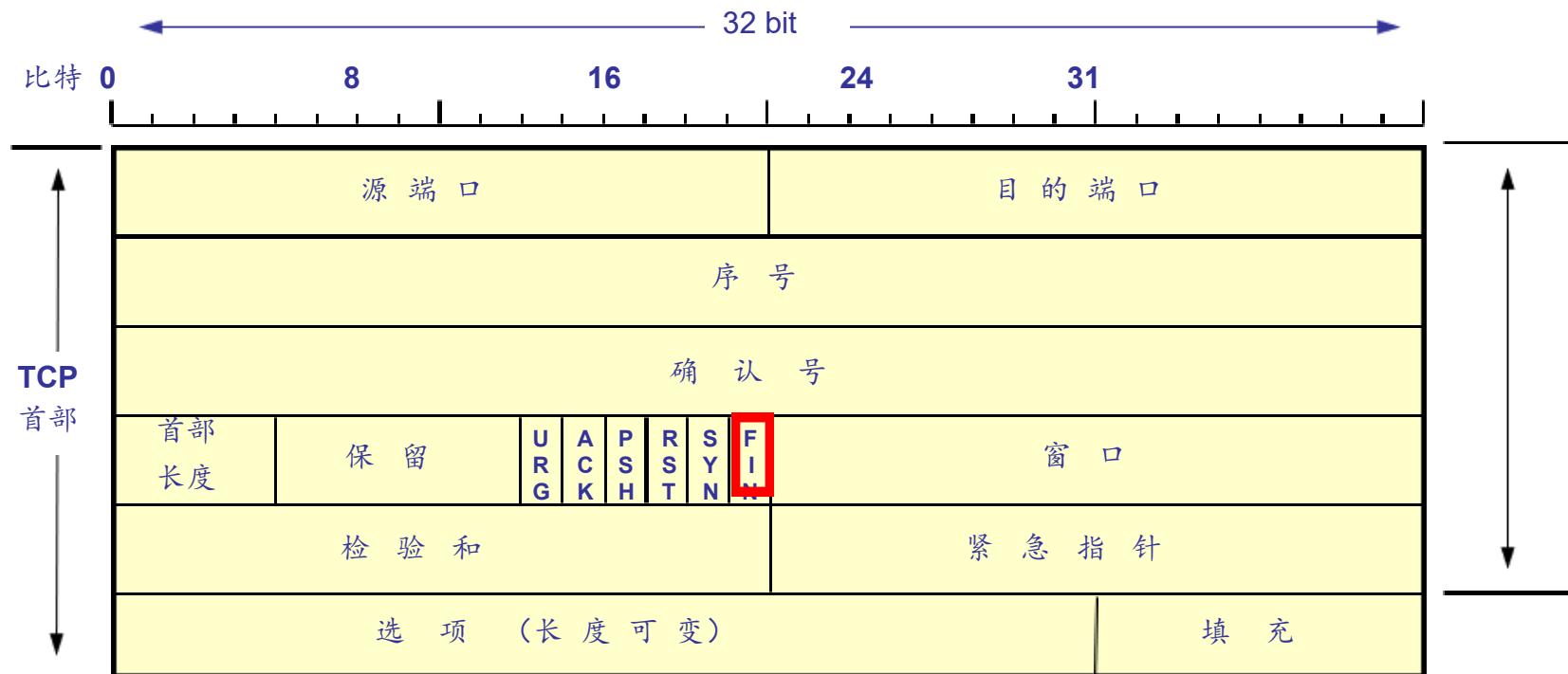
复位比特 RST (ReSeT) —— 当 $RST = 1$ 时，表明 TCP 连接中出现严重差错（如由于主机崩溃或其他原因），必须释放连接，然后再重新建立运输连接。

3.5 面向连接的传输：TCP



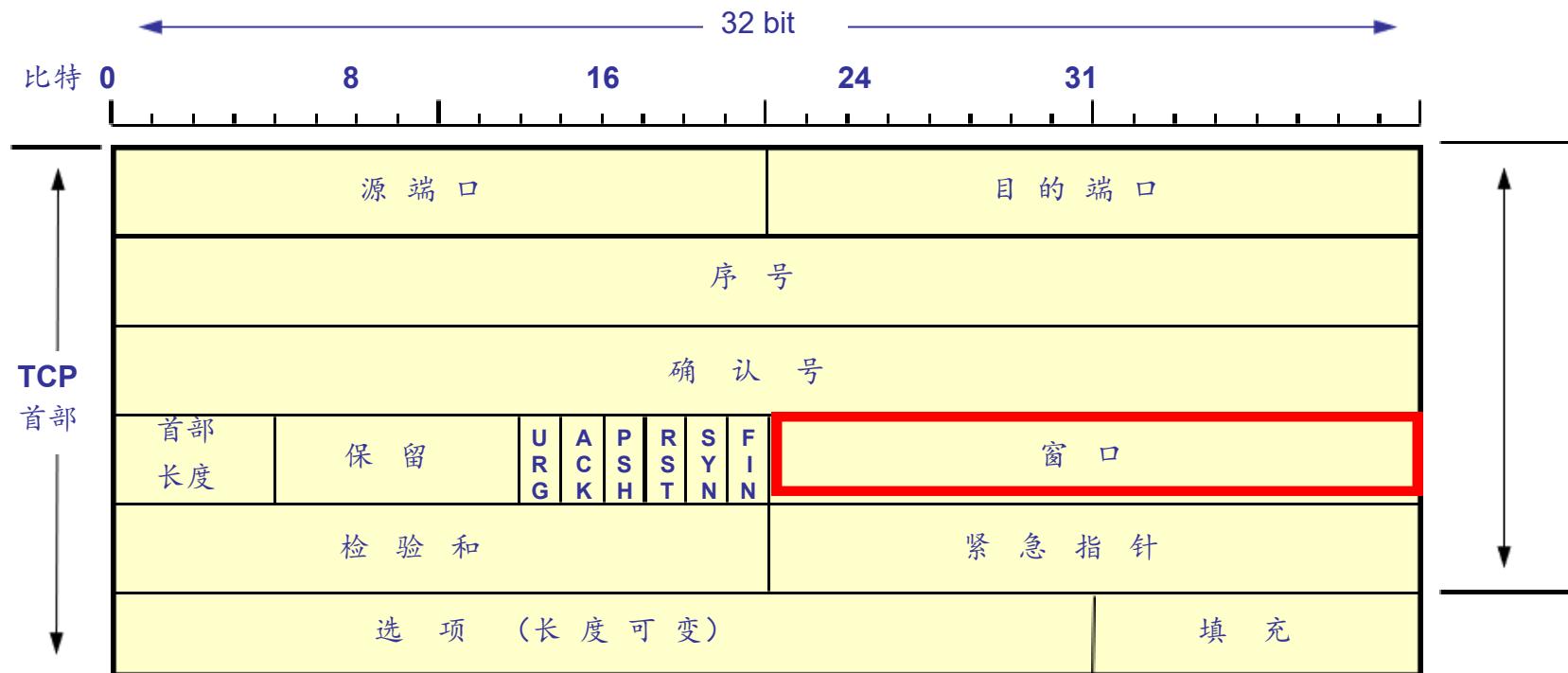
同步比特 SYN —— 同步比特 SYN 置为 1，就表示这是一个连接请求或连接接受报文。

3.5 面向连接的传输：TCP



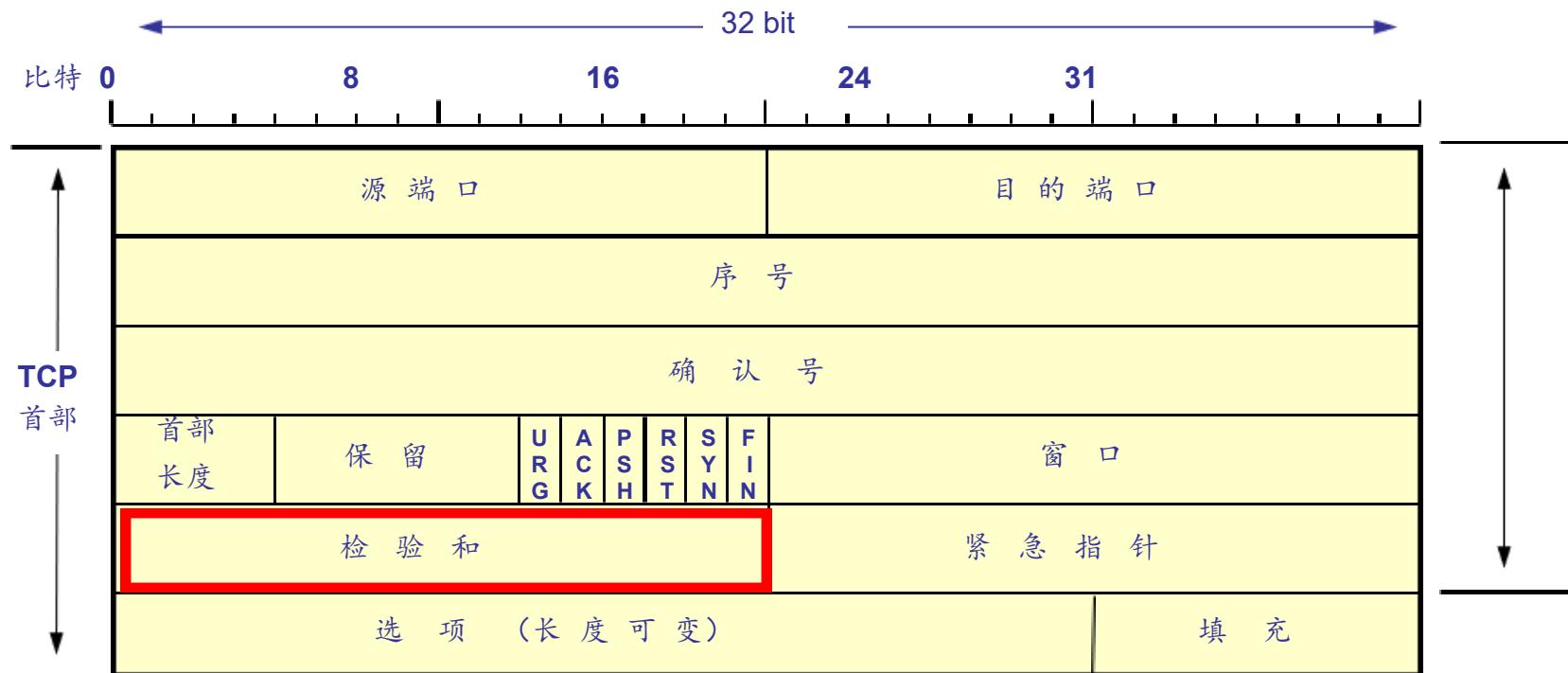
终止比特 **FIN (FINal)** —— 用来释放一个连接。当 **FIN = 1** 时，表明此报文段的发送端的数据已发送完毕，并要求释放运输连接。

3.5 面向连接的传输：TCP



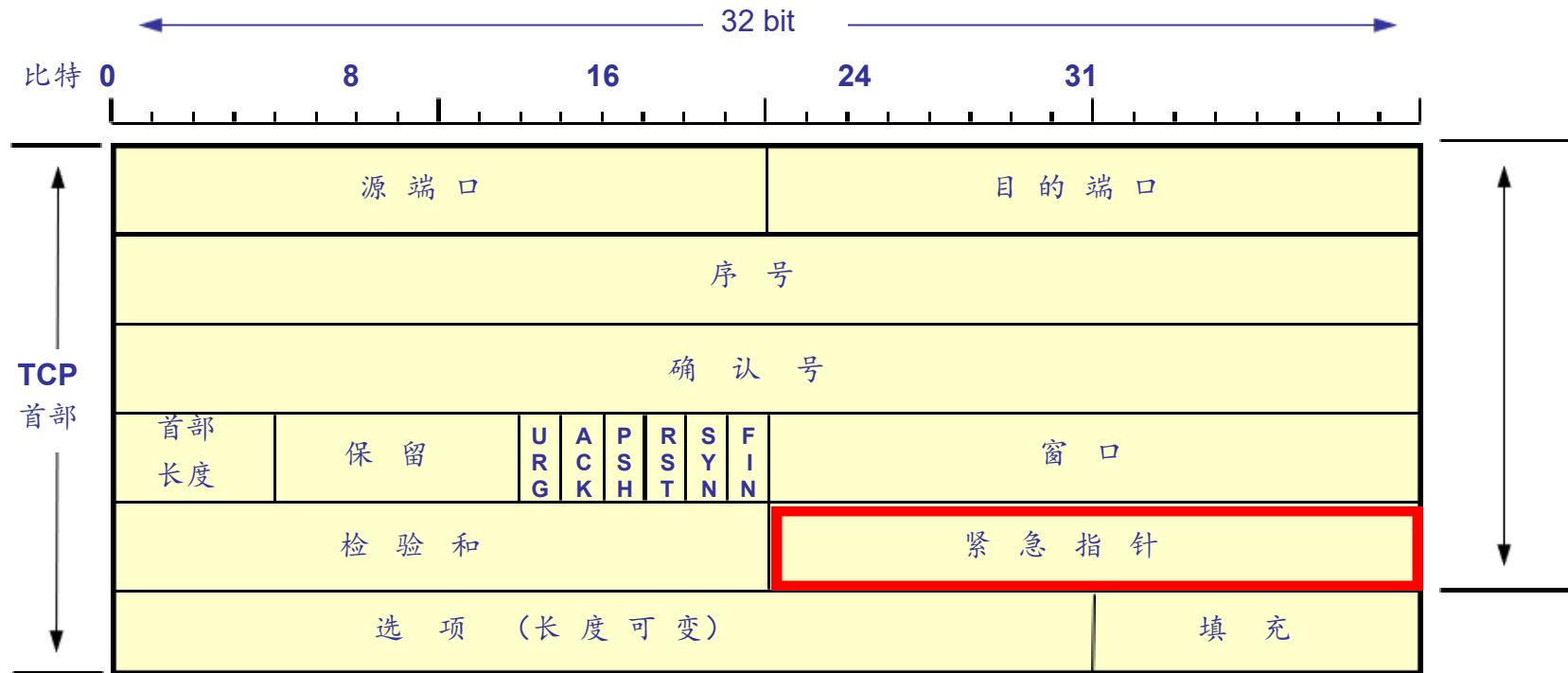
窗口字段——占 2 字节。窗口字段用来控制对方发送的数据量，单位为字节。TCP 连接的一端根据设置的缓存空间大小确定自己的接收窗口大小，然后通知对方以确定对方的发送窗口的上限。

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

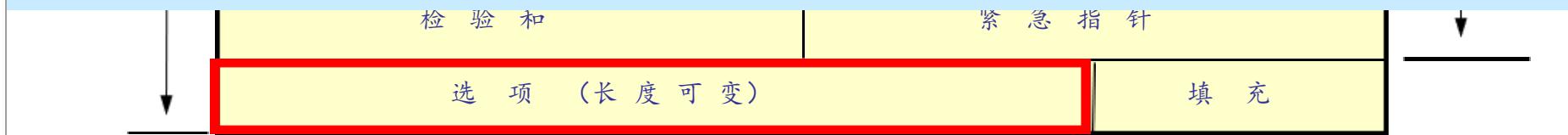


紧急指针字段——占 16 bit。紧急指针指出在本报文段中的紧急数据的最后一个字节的序号。

3.5 面向连接的传输 : TCP

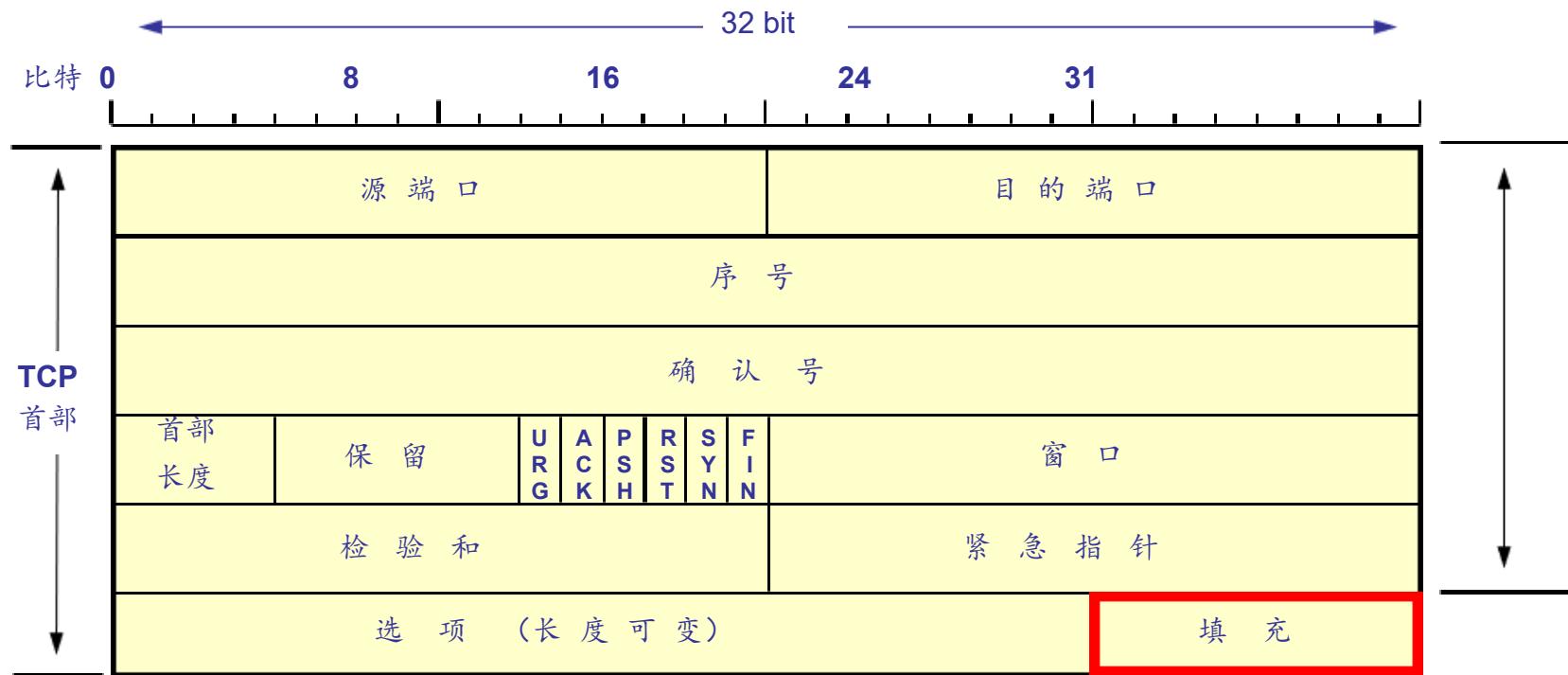
MSS 是 TCP 报文段中的数据字段的最大长度。

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

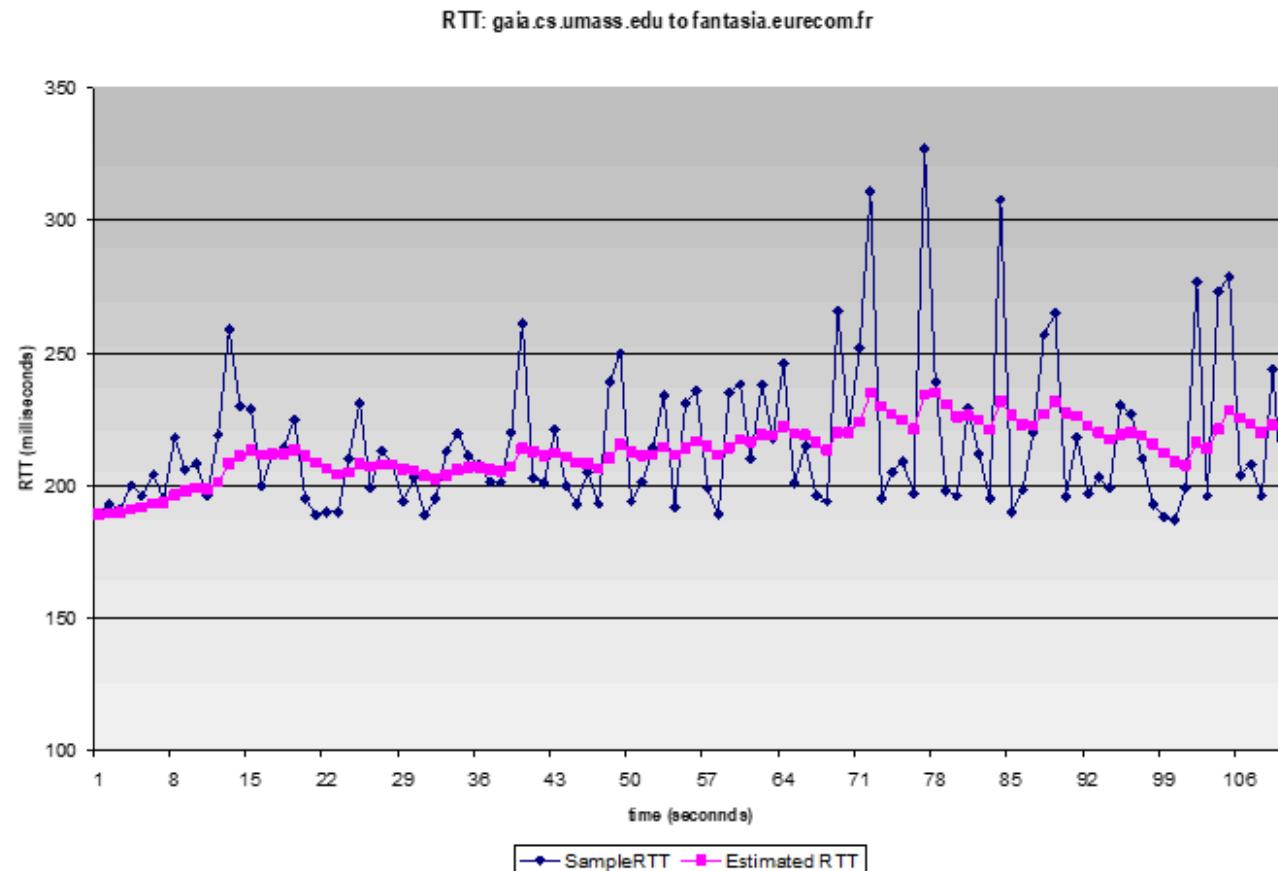
- 样本RTT(SampleRTT): 对报文段被发出到收到该报文段的确认之间的时间进行测量
 - 忽略重传
- 样本RTT会有波动，要使得估算RTT更平滑，需要将最近几次的测量进行平均，而非仅仅采用最近一次的 SampleRTT

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

参考值: $\alpha = 0.125$

3.5 面向连接的传输：TCP

■ RTT估计的一个例子



2020年10月6日

78

3.5 面向连接的传输：TCP

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

$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$

(参考值, $\beta = 0.25$)

TCP中的超时间隔为

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$

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

```

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
            start timer
        event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase = y
                if (there are currently not-yet-acknowledged segments)
                    start timer
            }
    } /* end of loop forever */

```

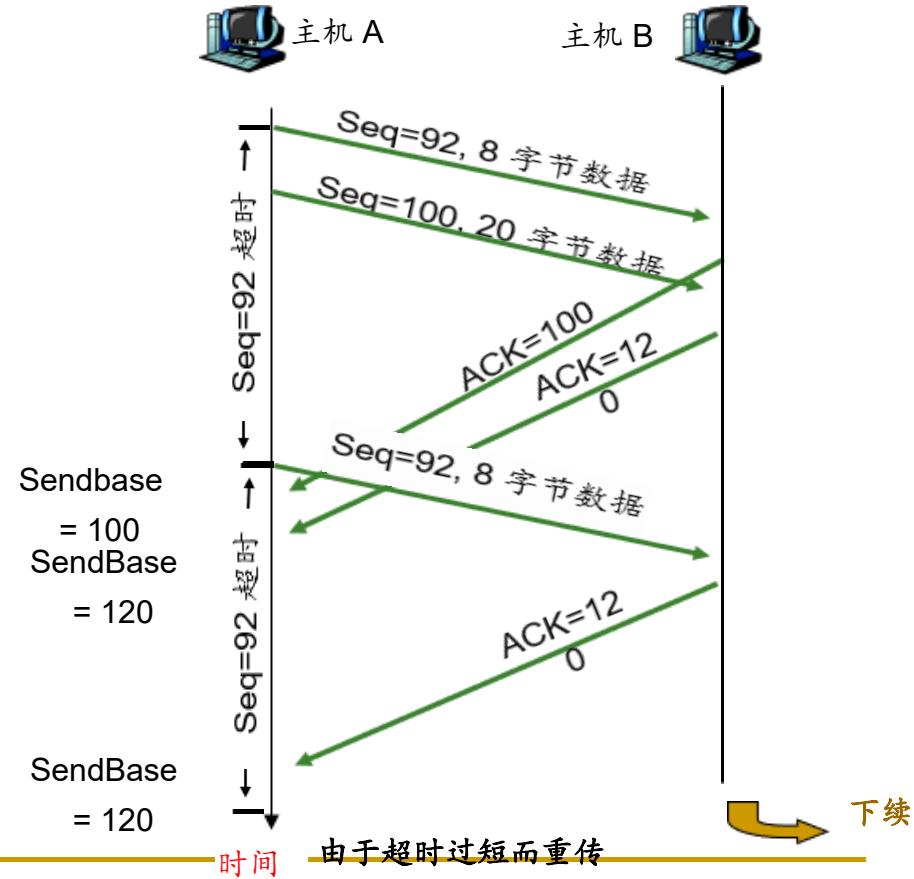
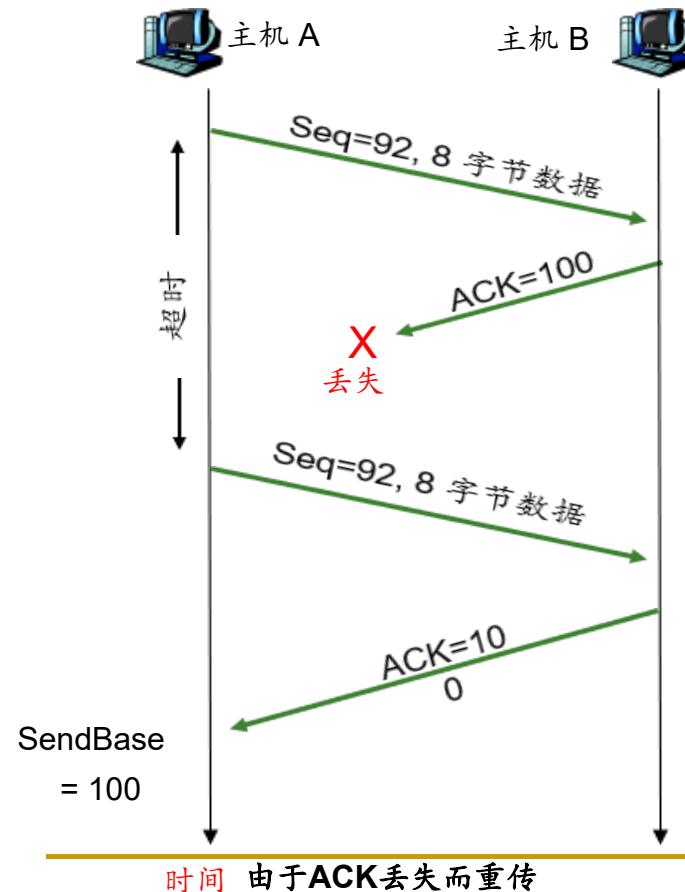
2020年10月6日

从应用到程序接收

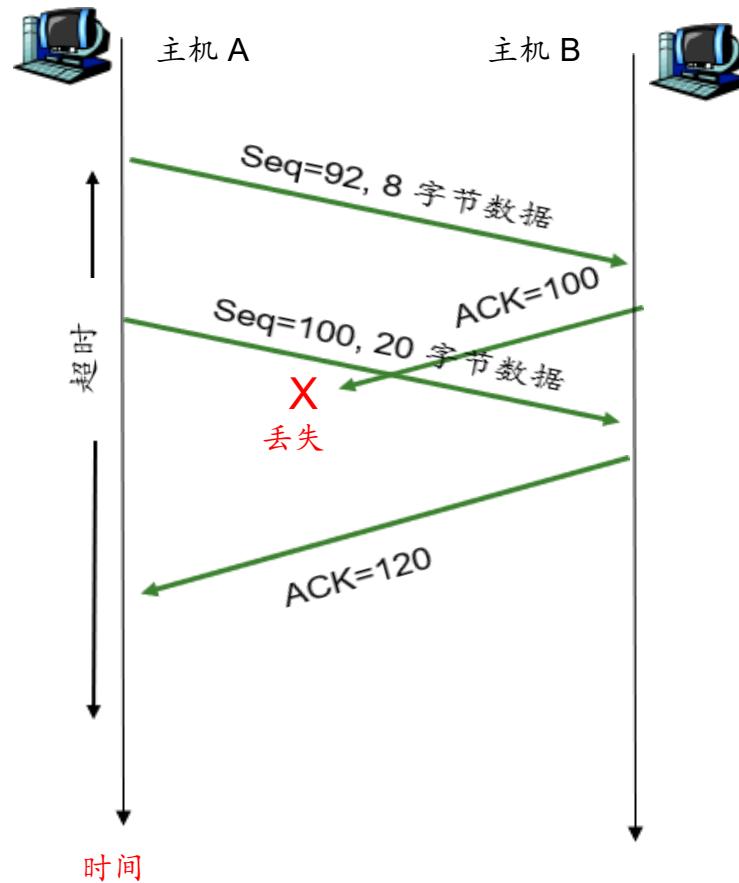
- 数据报是前超对报文段的桶式报文段中，每一个报文段都包含一个序号
- 序号是该报文段第如数据有字节的字节流的编号
- 启动定时器，TCP还要重启动定时器
- 超时间隔：启动定时器 TimeOutInterval

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



3.5 面向连接的传输：TCP

■ 产生TCP ACK的建议 (RFC1122、2581)

接收方事件	TCP接收方 动作
所期望序号的报文段按序到达。所有在期望序号及其以前的数据都已经被确认	延迟的ACK。对另一个按序报文段的到达最多等待 500ms 。如果下一个按序报文段在这个时间间隔内没有到达，则发送一个ACK
有期望序号的报文段按序到达。 另一个按序报文段等待发送ACK	立即发送单个累积ACK，以确认两个按序报文段
比期望序号大的失序报文段到达，检测出数据流中的间隔	立即发送冗余ACK，指明下一个期待字节的序号(也就是间隔的低端字节序号)
能部分或完全填充接收数据间隔的报文段到达	倘若该报文段起始于间隔的低端，则立即发送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

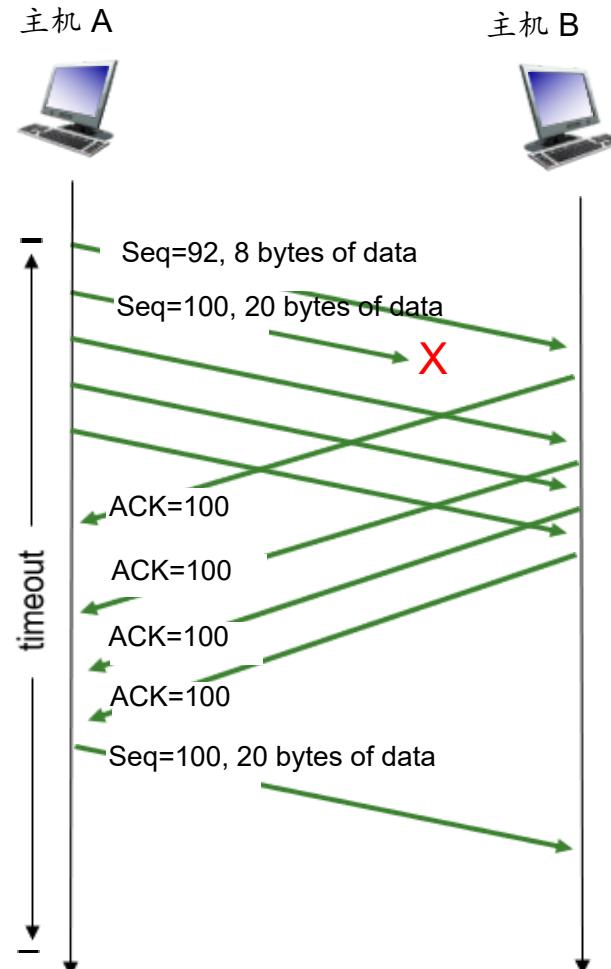
```
event: ACK received, with ACK field value of y
    if (y > SendBase) {
        SendBase = y
        if (there are currently not-yet-acknowledged segments)
            start timer
    }
    else {
        increment count of dup ACKs received for y
        if (count of dup ACKs received for y = 3)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



3.5 面向连接的传输：TCP

- 超时间隔加倍
 - 每一次TCP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
 - 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

■ 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

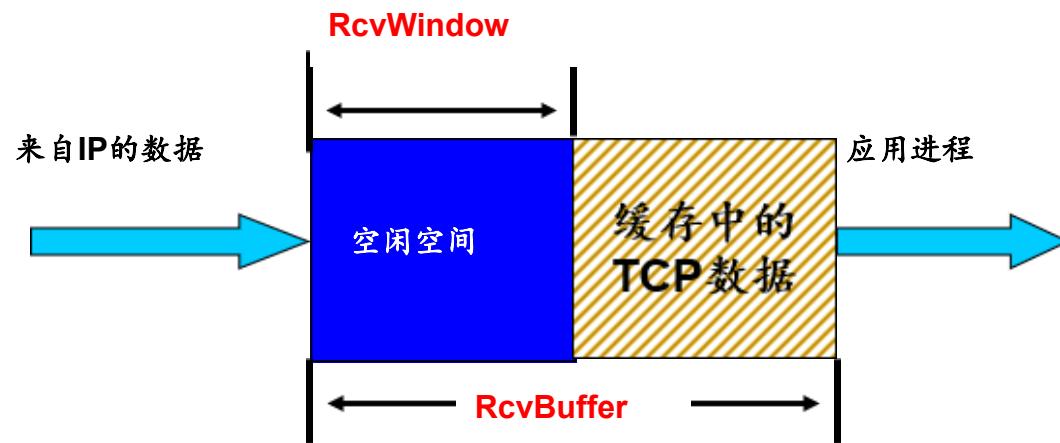
- 发送方不会由于传得太多太快而使得接收方缓存溢出

□ 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

□ 窗口值的计算



接收方： $\text{LastByteRcvd} - \text{LastByteRead} \leq \text{RcvBuffer}$

$$\text{RcvWindows} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$$

发送方： $\text{LastByteSent} - \text{LastByteAcked} \leq \text{RcvWindow}$

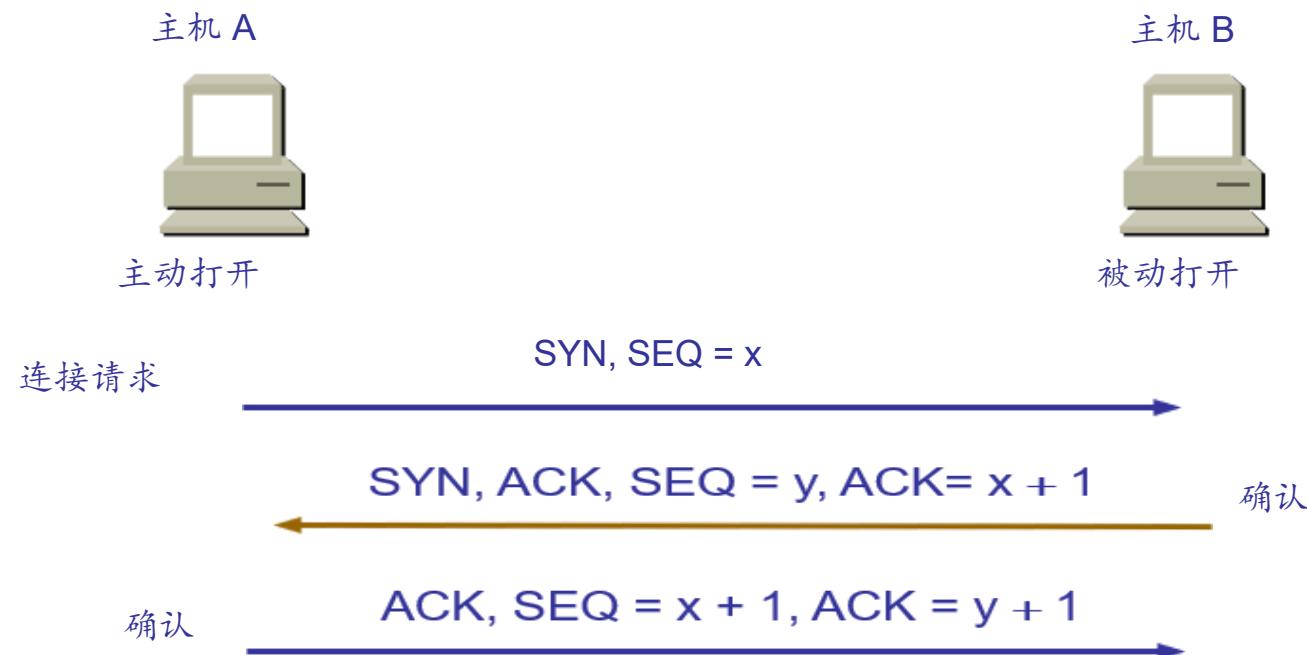
3.5 面向连接的传输：TCP

□ 一种特殊的情况

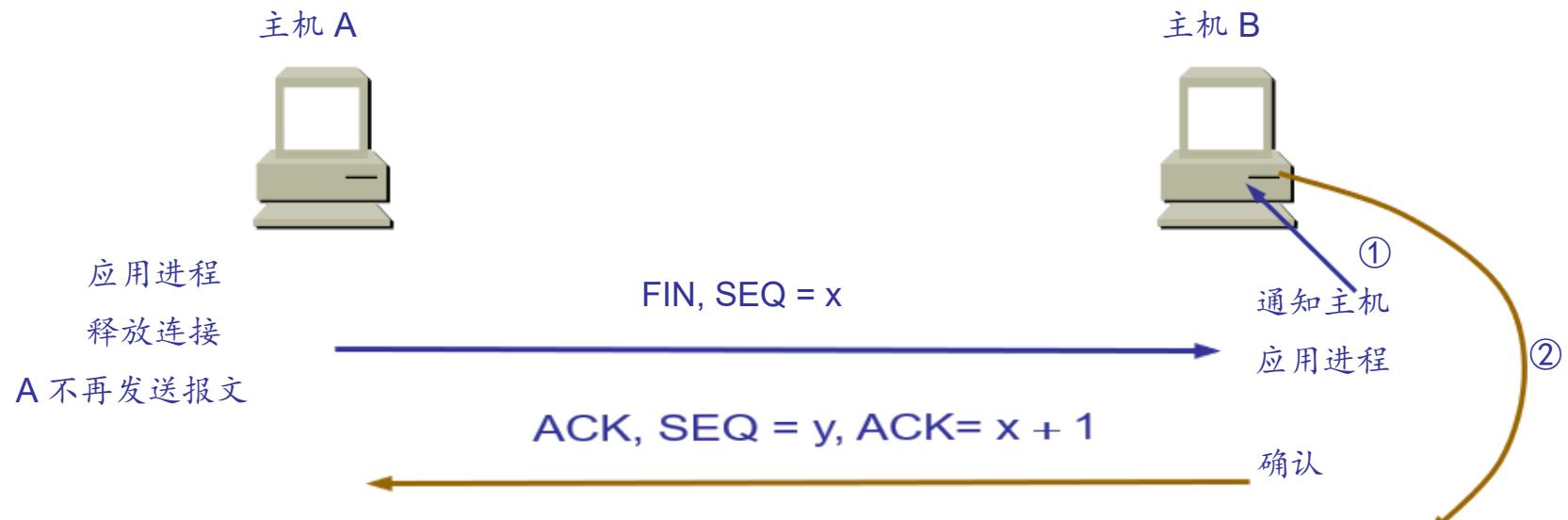
- 接收方通知发送方RcvWindow为0，且接收方无任何数据传送给发送方
- 发送方持续向接受方发送只有一个字节数据的报文段，目的是试探

3.5 面向连接的传输 : TCP

■ TCP连接的建立



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。

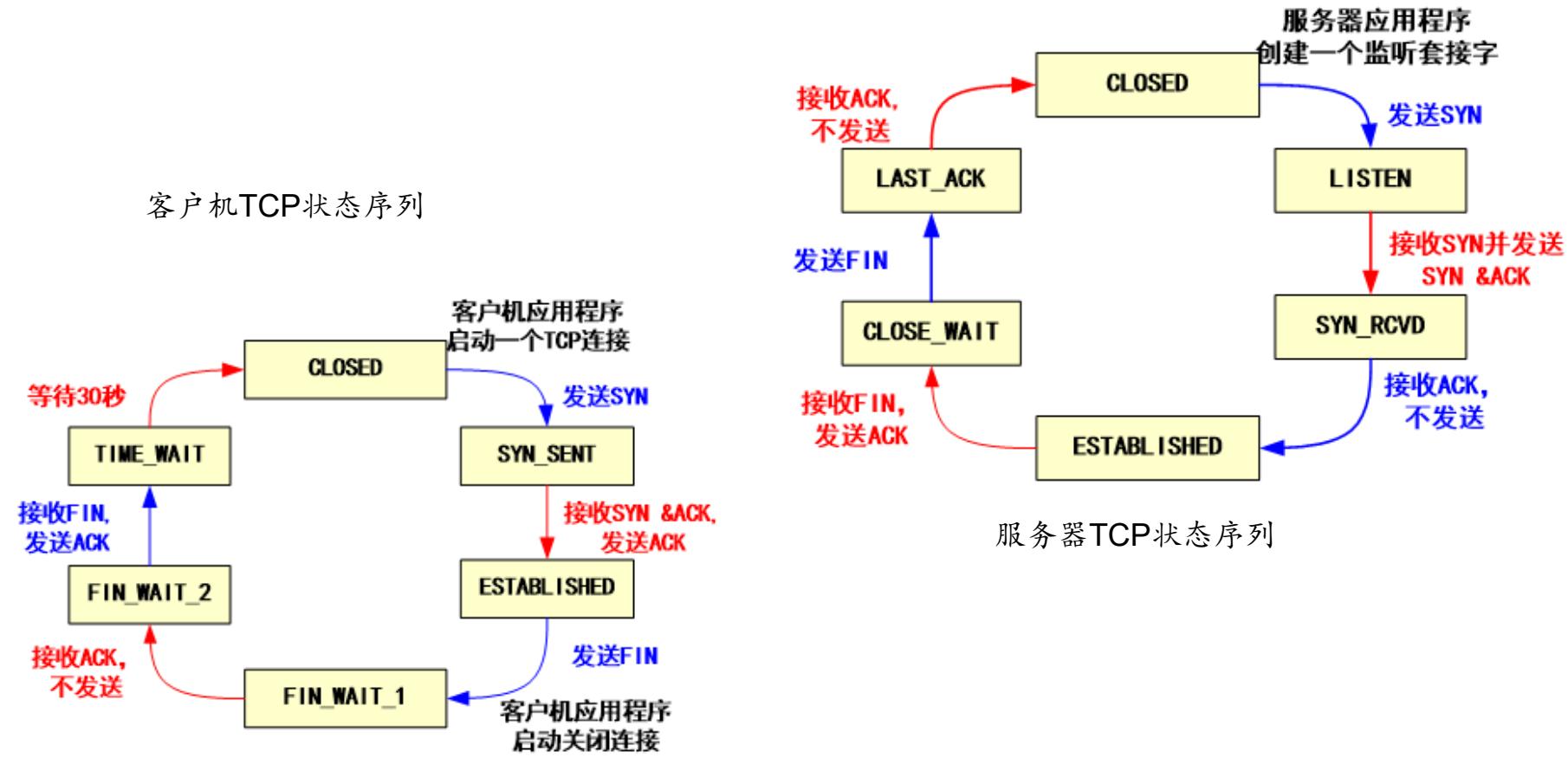
相当于 A 向 B 说：

“我已经没有数据要发送了。

但你如果还发送数据，我仍接收。”

3.5 面向连接的传输：TCP

TCP连接管理的状态序列



3.6 拥塞控制原理

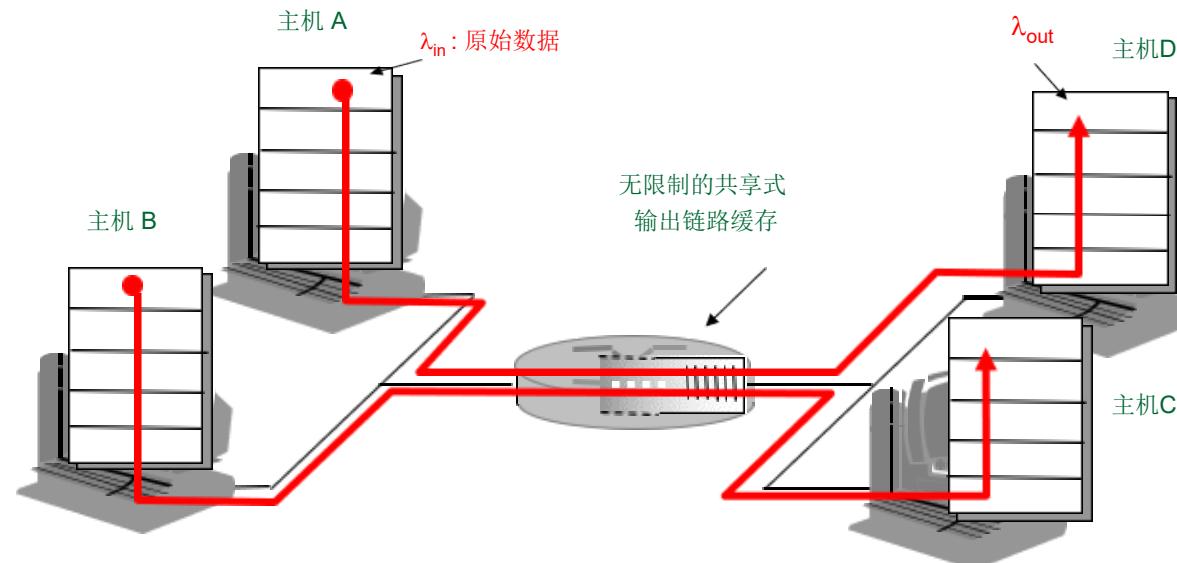
■ 拥塞的基本知识

- 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！
- 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

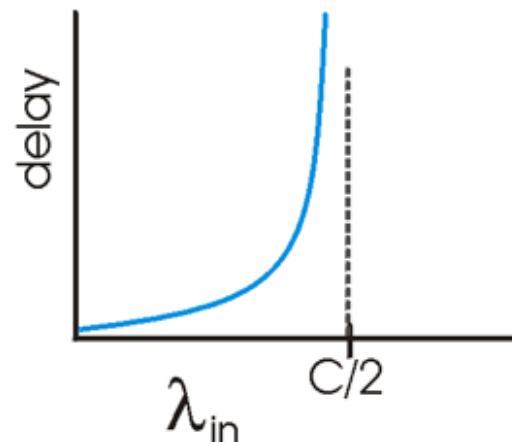
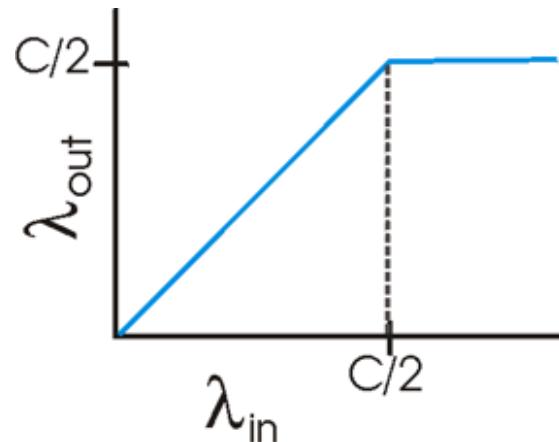
- 两个发送方，两个接受方
- 一个具有无限大缓存的路由器
- 没有重传



3.6 拥塞控制原理

■ 情境1

- 最大可获得的吞吐量
- 出现拥塞时延时变大

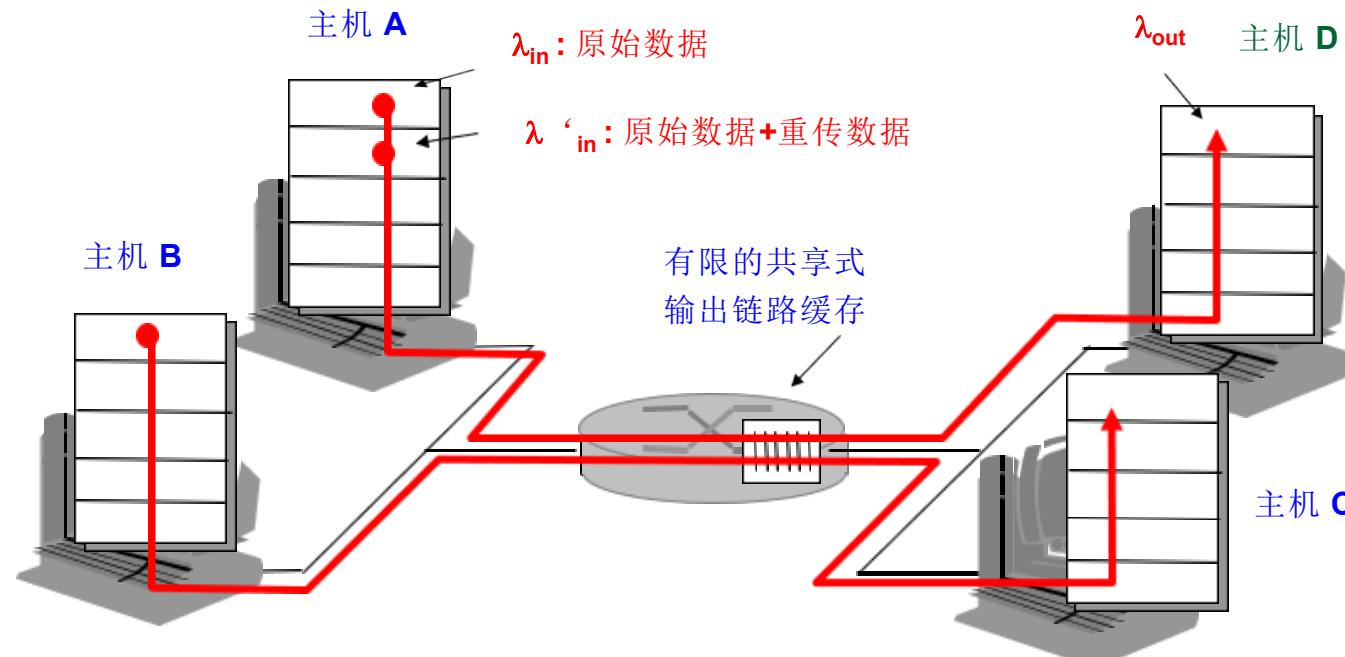


拥塞代价：当分组到达速率接近链路容量时，分组经历的巨大排队时延

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

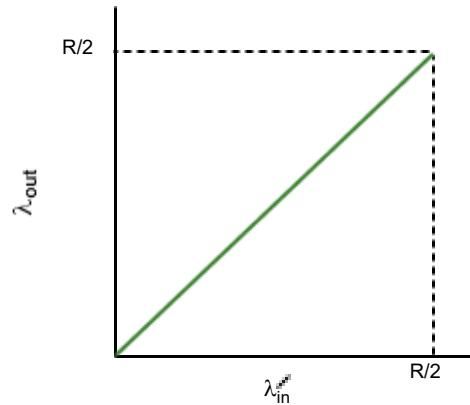


3.6 拥塞控制原理

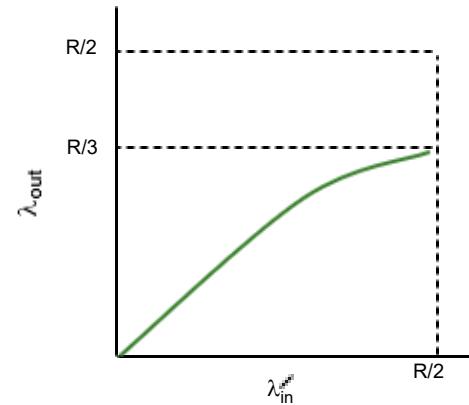
- 设计期望: $\lambda_{in} = \lambda_{out}$
- “理想”的重传是仅仅在丢包时才发生重传:
- 对延迟到达(而非丢失)的分组的重传使得 比理想情况下更大于 $\lambda_{in} > \lambda_{out}$

$$\lambda' > \lambda_{out}$$

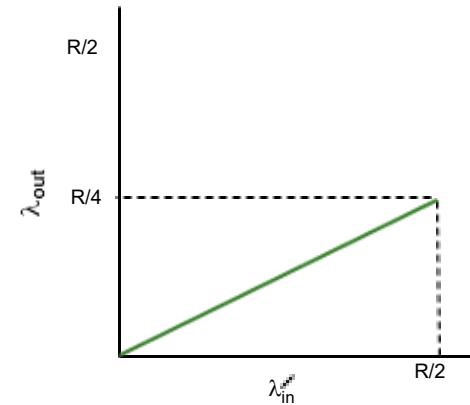
$$\lambda_{out}$$



a.



b.



c.

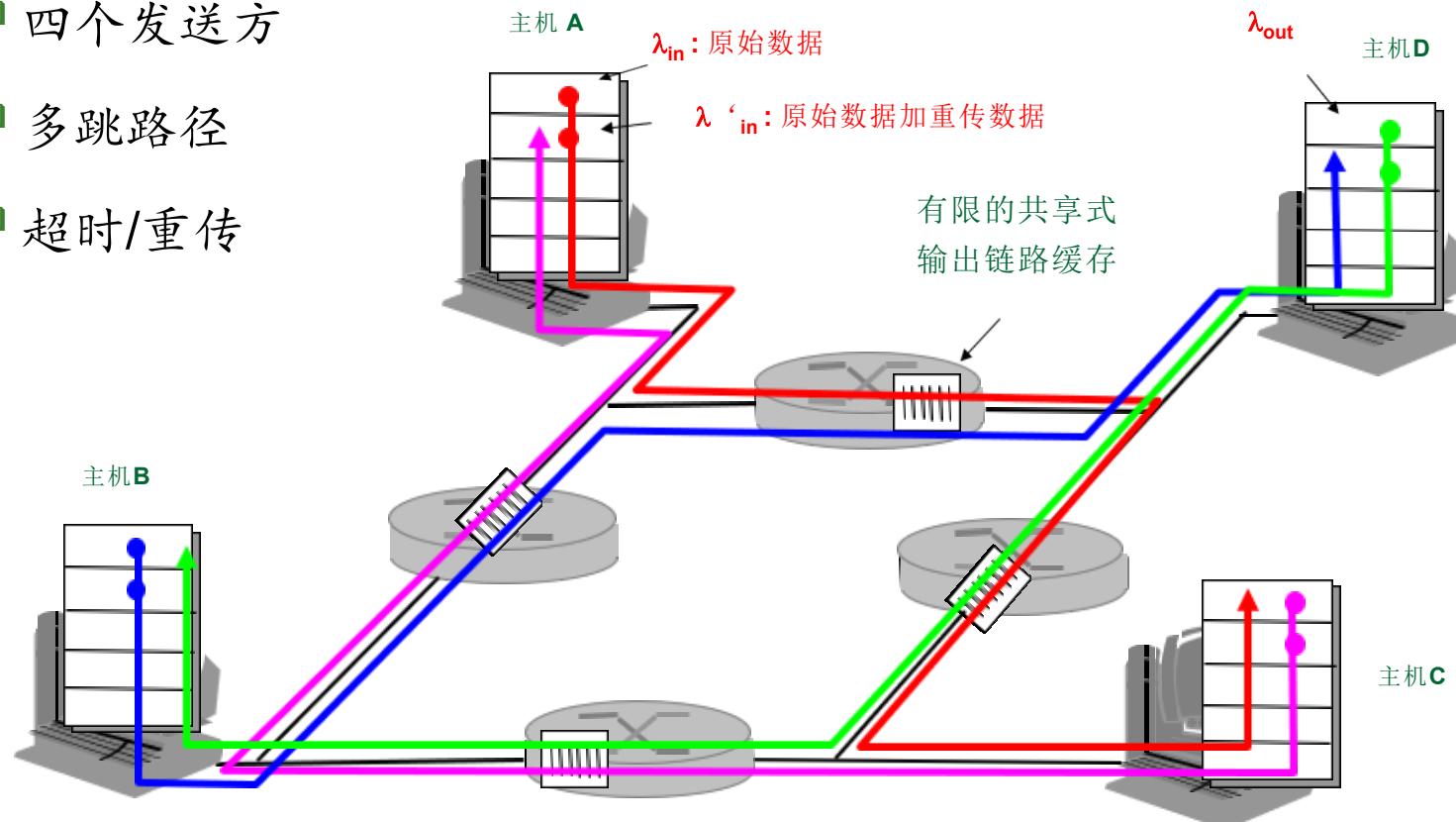
拥塞的“开销”：

- 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

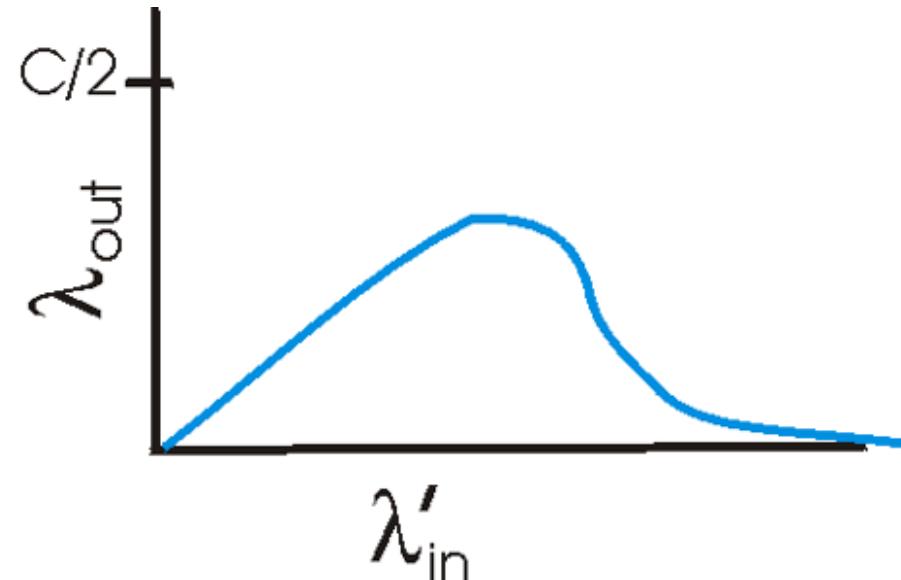


2020年10月6日

100

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

3.6 拥塞控制原理

■ 例子：ATM ABR拥塞控制

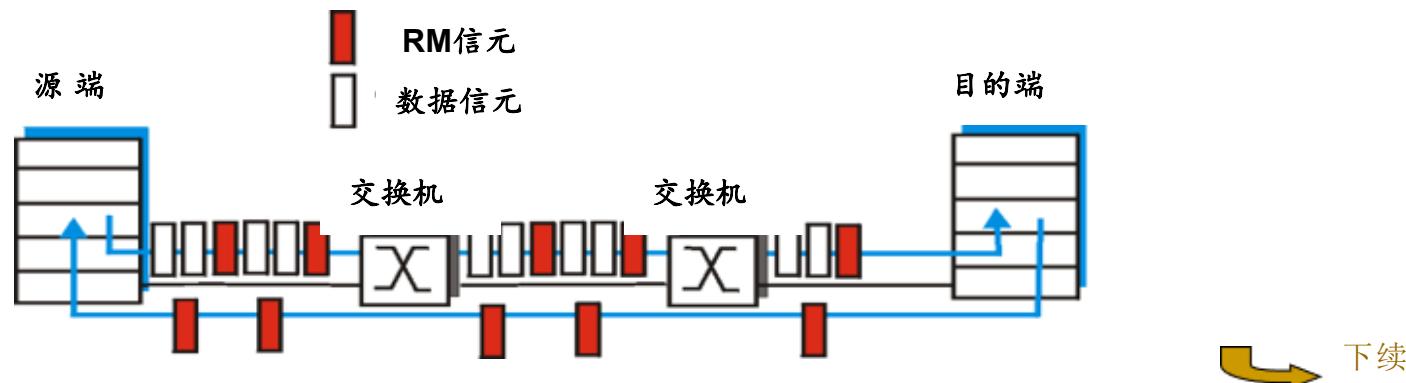
□ 特点

- “弹性服务”
- 当发送方网络“轻载”时：
 - 发送方应充分利用空闲的可用带宽
- 当发送方网络“拥塞”时：
 - 发送方应将传输速率抑制为最小承诺传输速率

3.6 拥塞控制原理

□ 工作机制

- 发送方散布在数据信元中会发出若干个RM信元
- 交换机可以设置RM信元中的比特（“网络辅助”）
 - NI比特：不得增大发送速率（轻度拥塞）
 - CI比特：拥塞指示（严重拥塞）
- 接收方会将RM信元完整地回送给发送方



3.6 拥塞控制原理

- RM信元中包含一个两字节的显式速率ER (explicit rate) 字段
 - 出现拥塞的交换机会减小经过自己的信元中ER字段的值
 - 发送方的发送速率也因之减小为路径支持的最小速率，可以得到最低程度得支持
- 数据信元中的EFCI 比特: 拥塞交换机将之置1来通知目的主机网络已经出现拥塞
 - 如果一个RM信元到达时，之前多数数据信元的EFCI比特置1， 目的方会将返回给发送方的RM信元中CI比特置1

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
 - 拥塞窗口

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin}$$

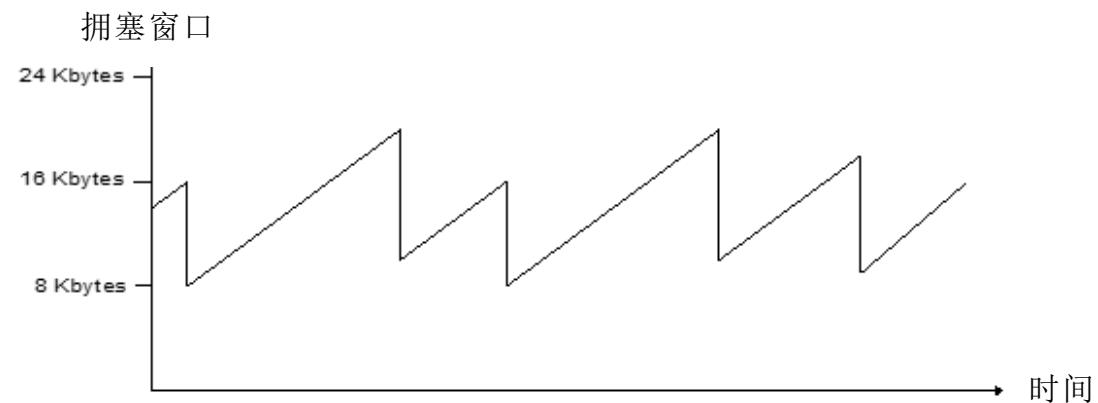
$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- 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



2020年10月6日

108

3.7 TCP拥塞控制

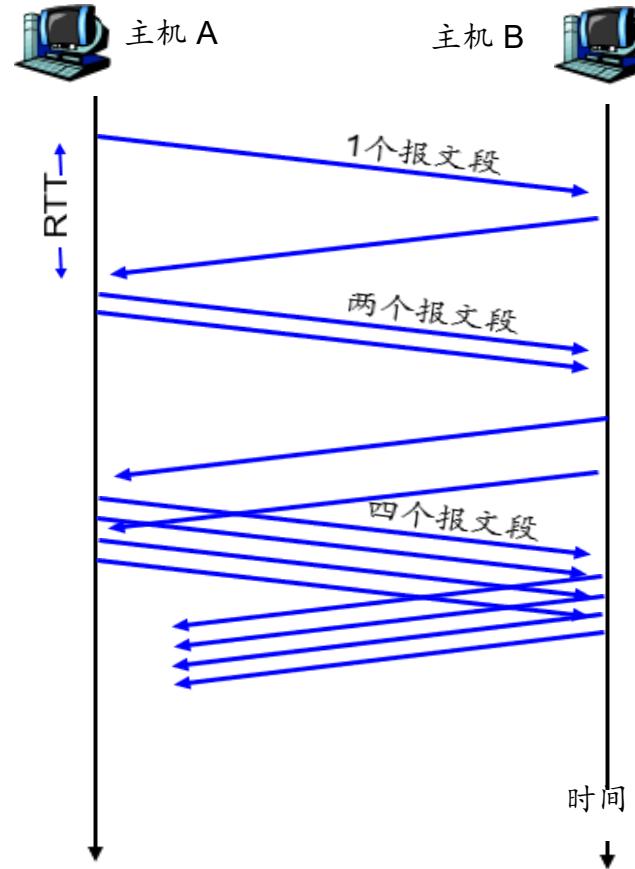
□ 慢启动

- 建立连接时, **CongWin = 1 MSS**
 - 例如: $MSS = 500 \text{ bytes}$ & $RTT = 200 \text{ msec}$
 - 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



2020年10月6日

110

3.7 TCP拥塞控制

□ 对收到3个重复ACK的反应

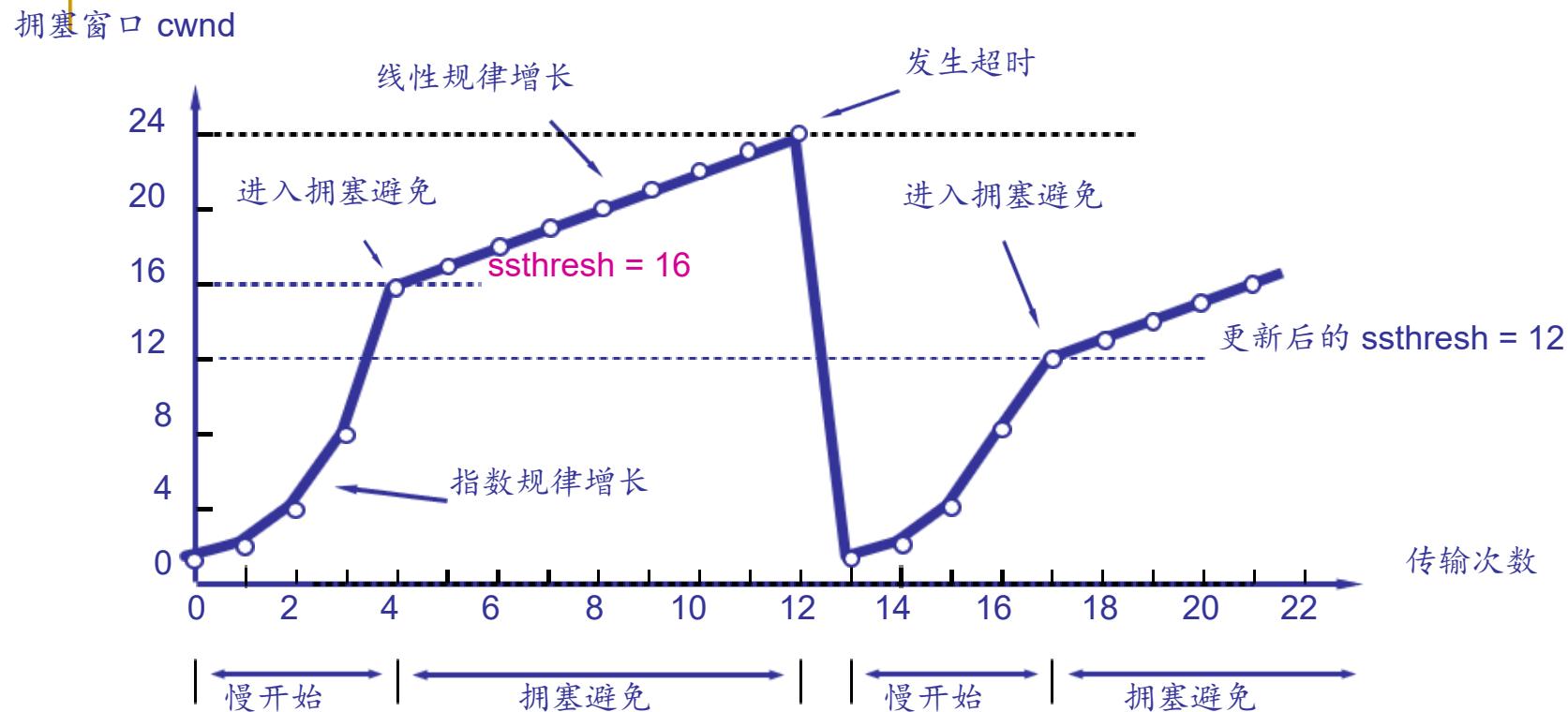
- 将CongWin减为原来的一半
- 线性增大拥塞窗口

□ 对超时事件的反应

- 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

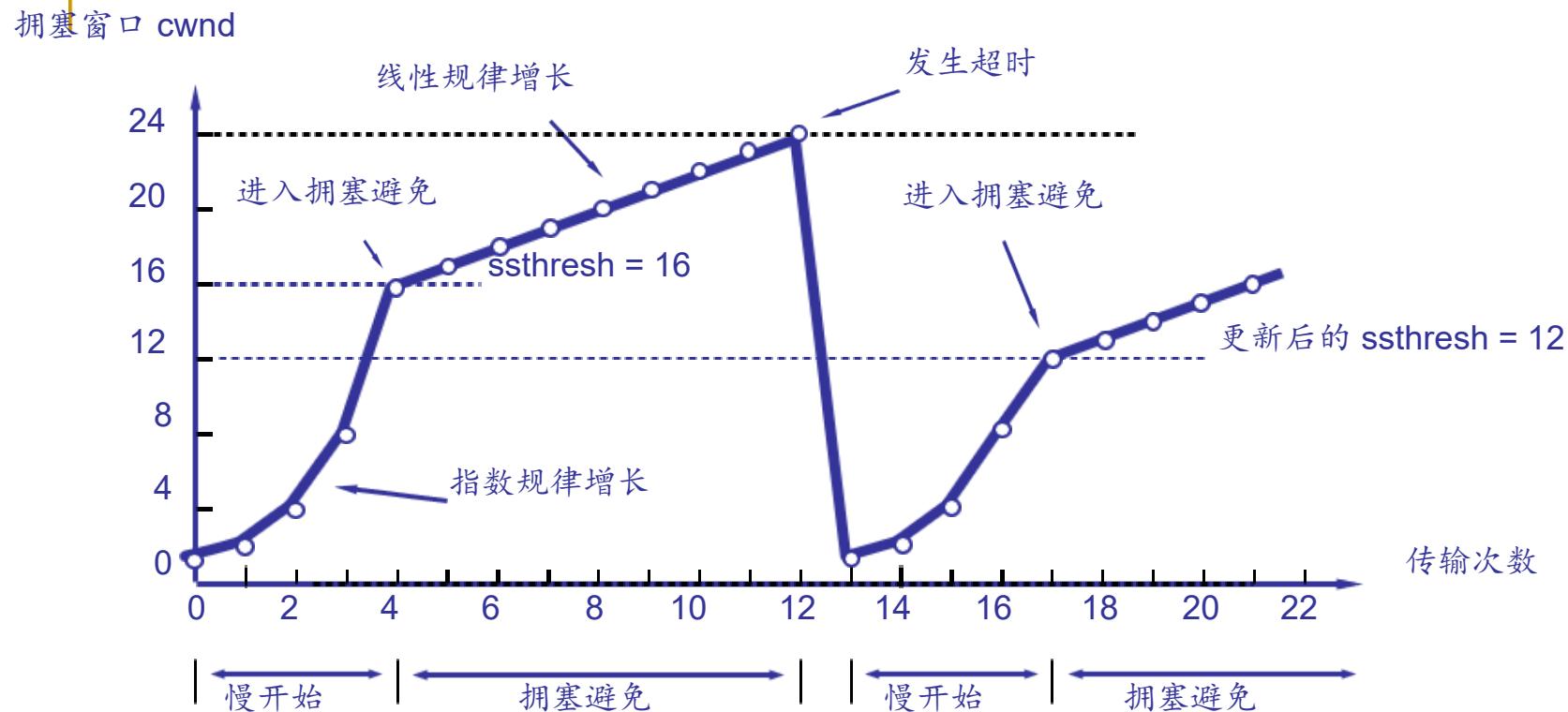
慢开始和拥塞避免算法的实现举例



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

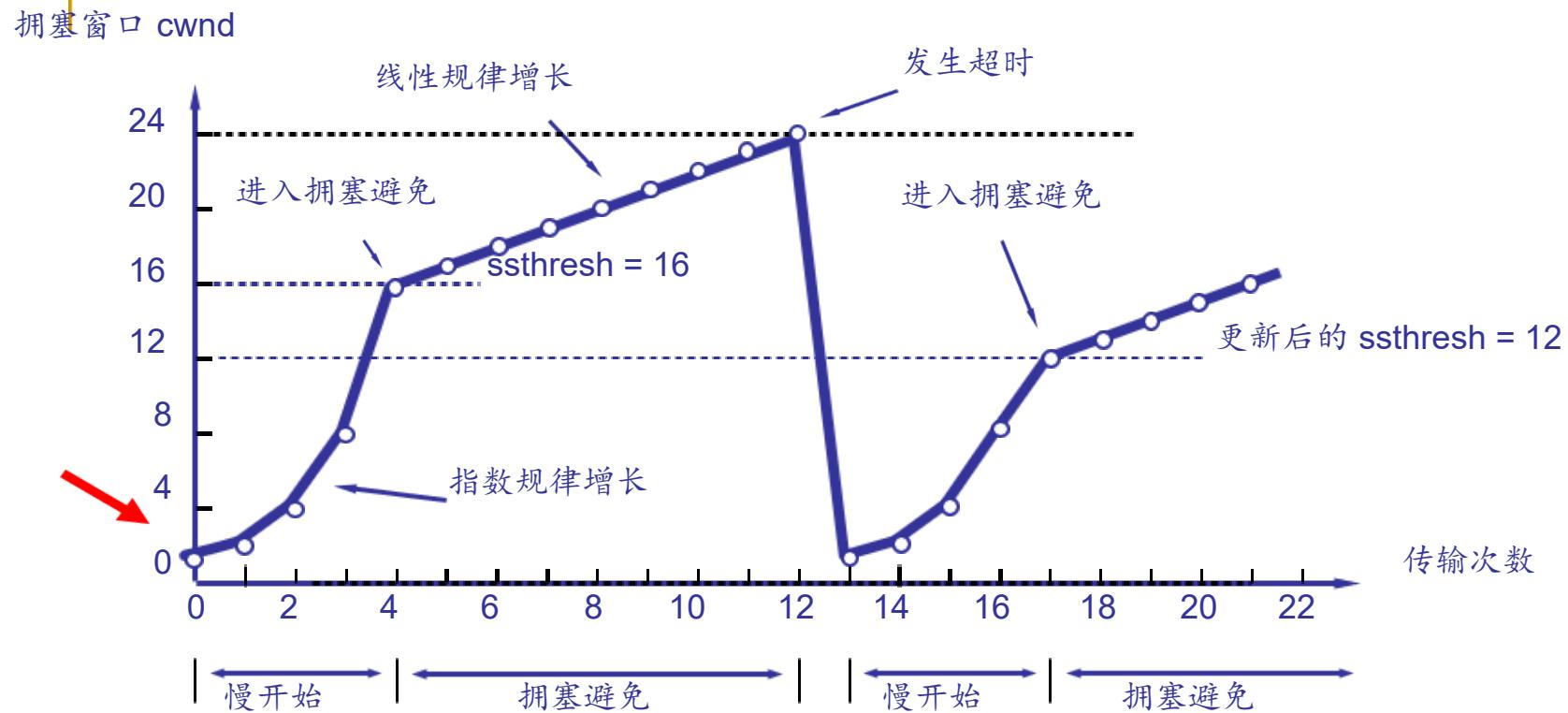
慢开始门限的初始值设置为 16 个报文段，即 $ssthresh = 16$ 。

慢开始和拥塞避免算法的实现举例



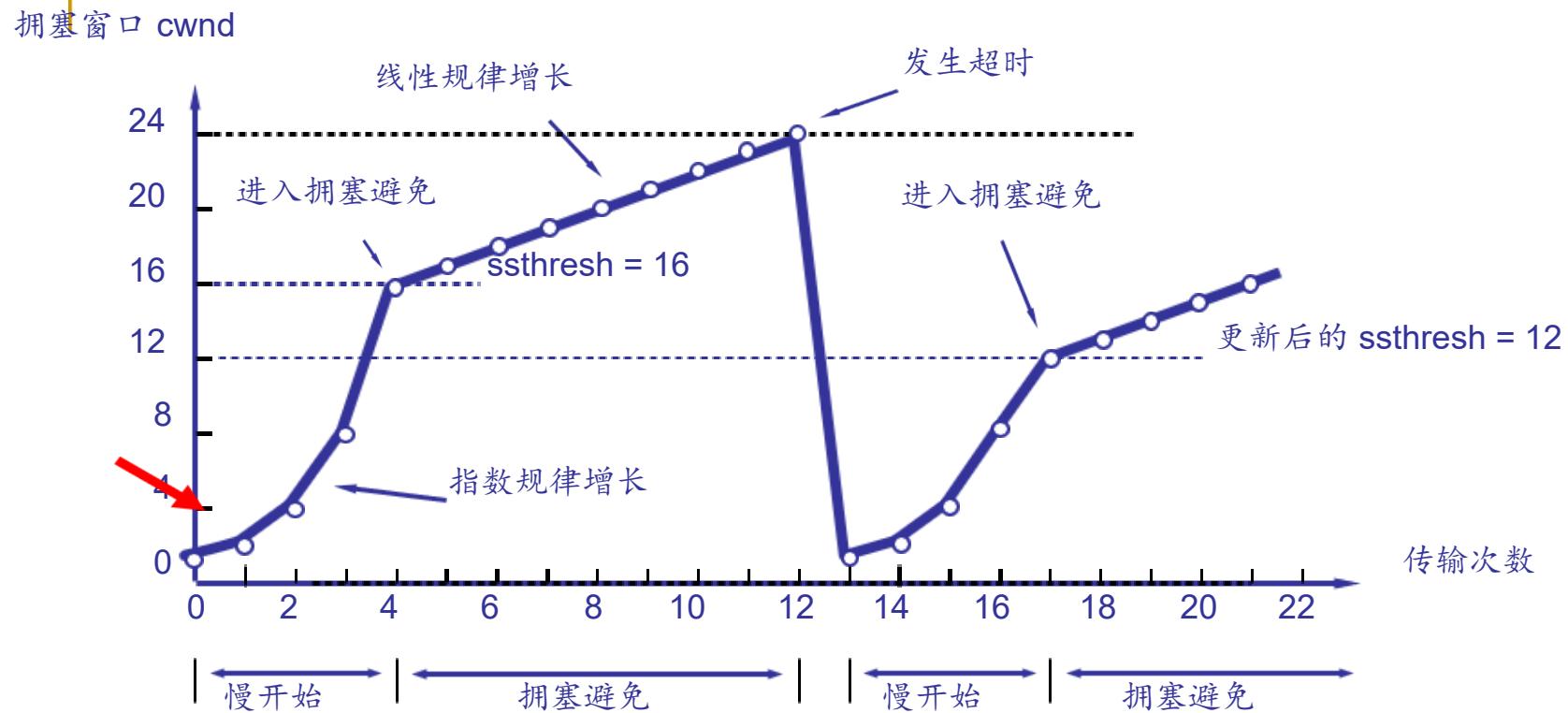
发送端的发送窗口不能超过拥塞窗口 cwnd 和接收端窗口 rwnd 中的最小值。我们假定接收端窗口足够大，因此现在发送窗口的数值等于拥塞窗口的数值。

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 的初始值为 1，发送第一个报文段 M_0 。

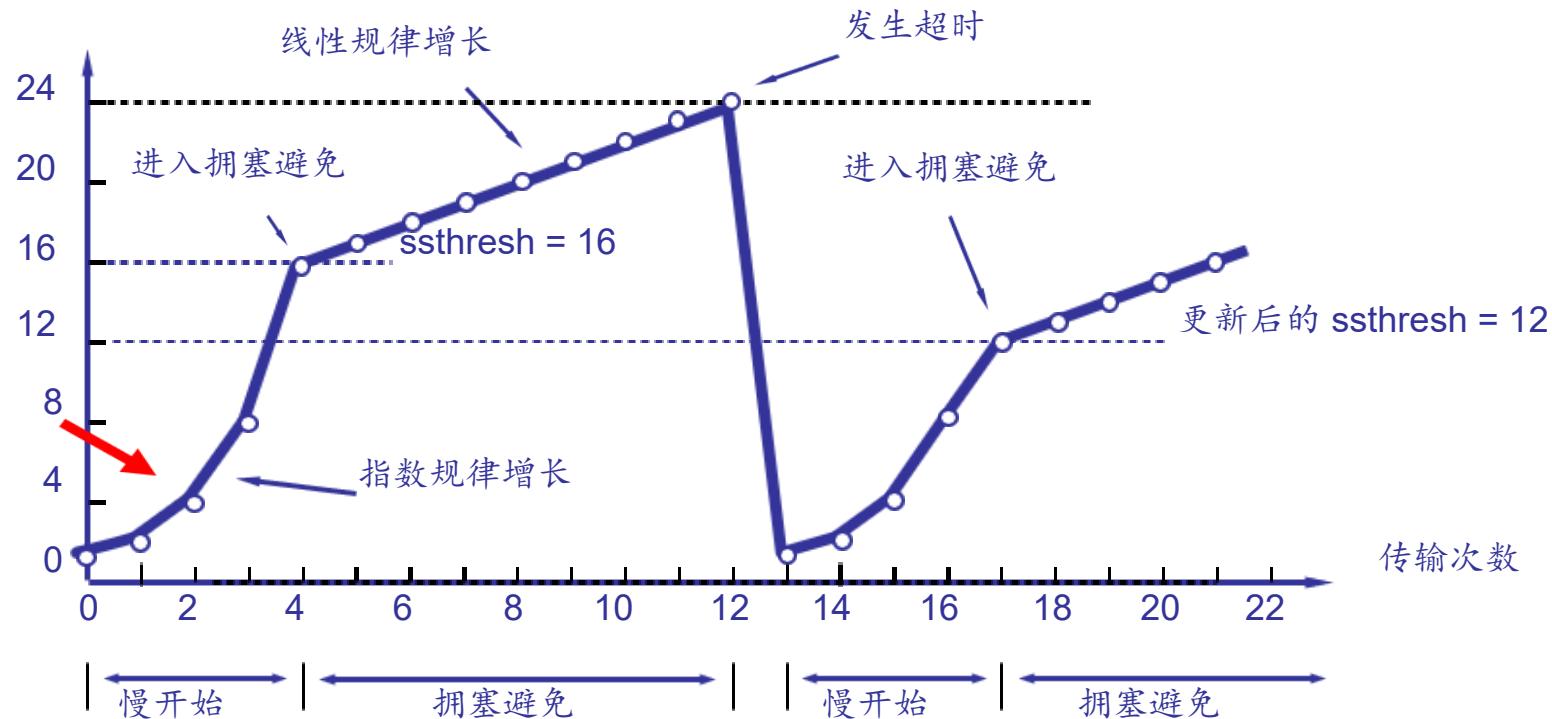
慢开始和拥塞避免算法的实现举例



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。

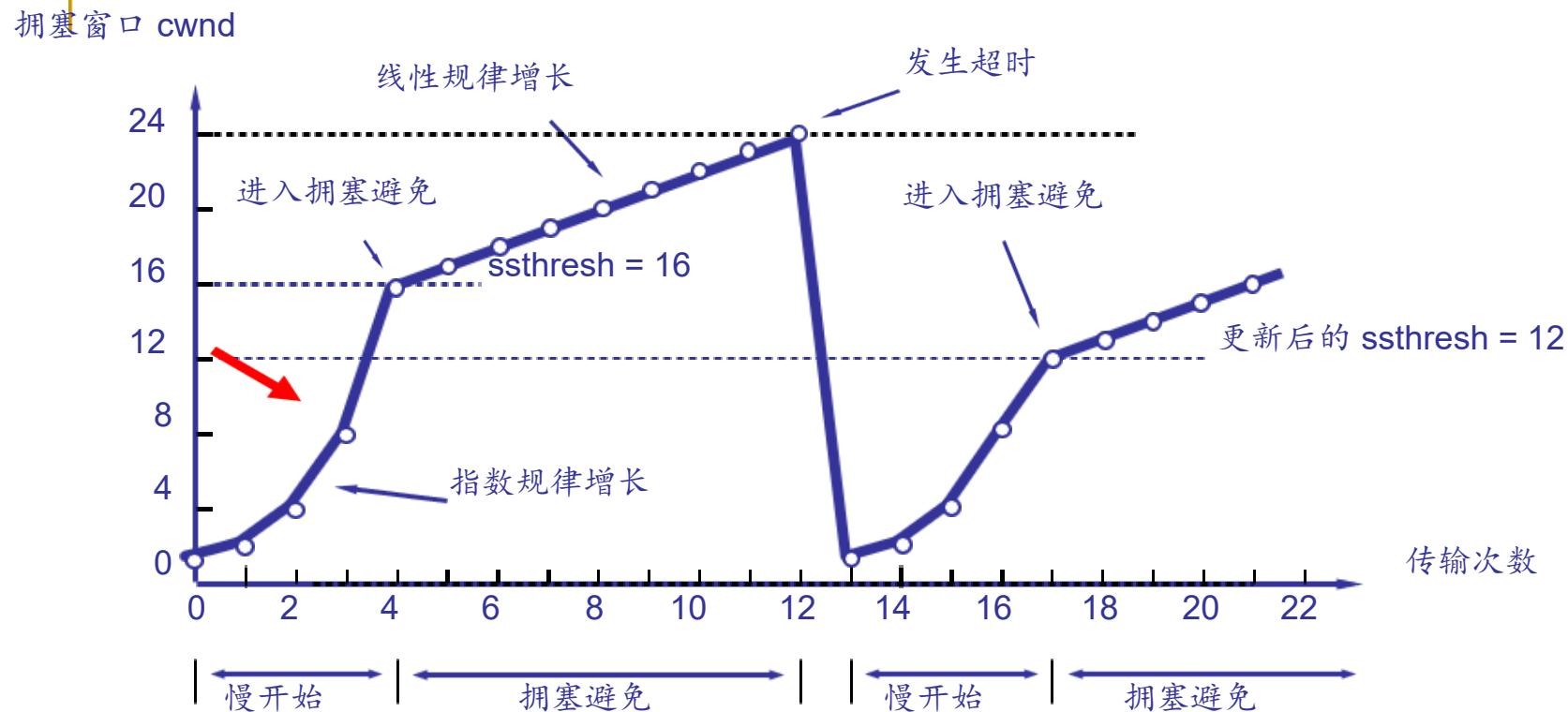
拥塞窗口 cwnd

慢开始和拥塞避免算法的实现举例



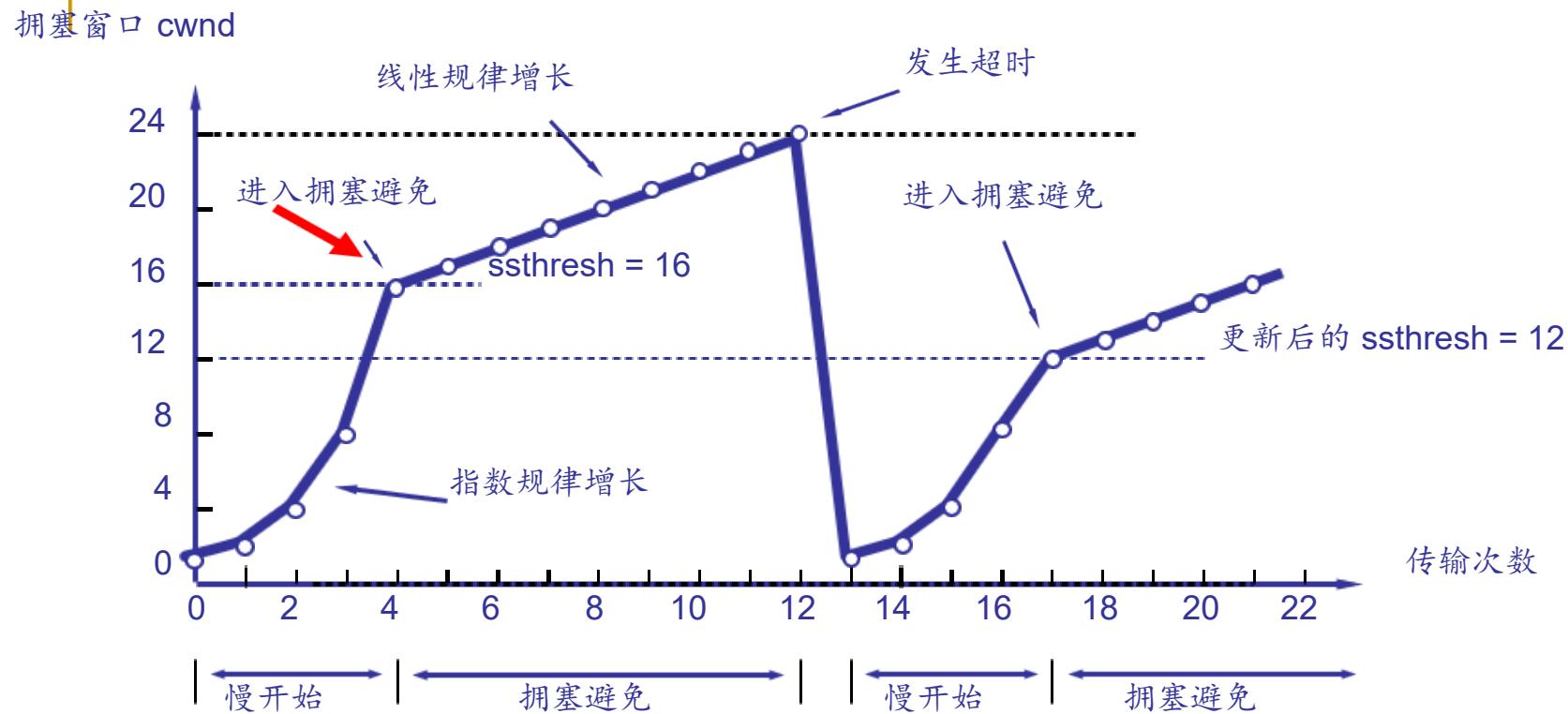
接收端发回 ACK_2 和 ACK_3 。发送端每收到一个对新报文段的确认 ACK , 就把发送端的拥塞窗口翻倍。现在发送端的 $cwnd$ 从 2 增大到 4, 并可发送 M_4 ~ M_7 共 4 个报文段。

慢开始和拥塞避免算法的实现举例

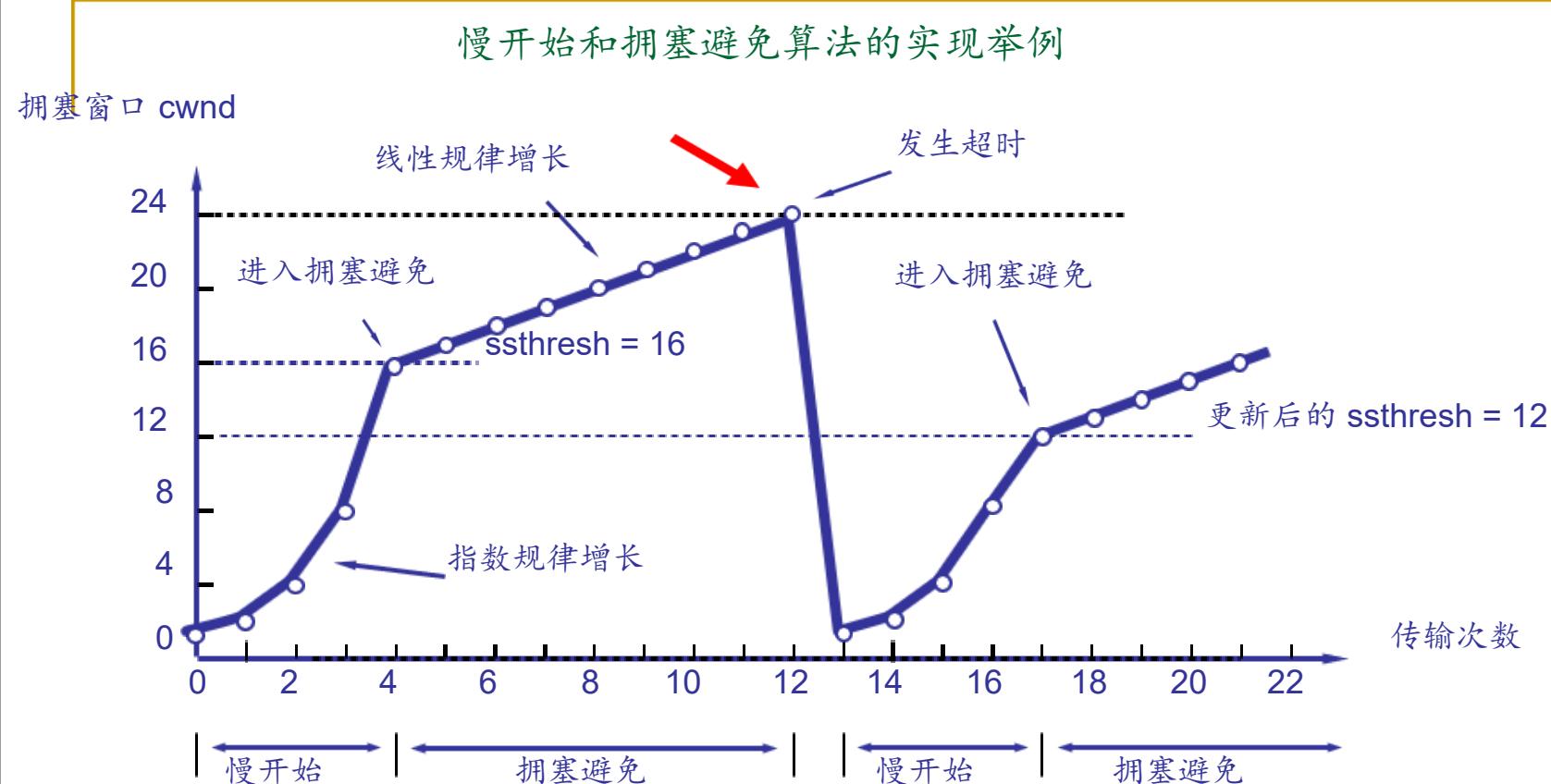


发送端每收到一个对新报文段的确认 ACK，就把发送端的拥塞窗口翻倍，因此拥塞窗口 cwnd 随着传输次数按指数规律增长。

慢开始和拥塞避免算法的实现举例



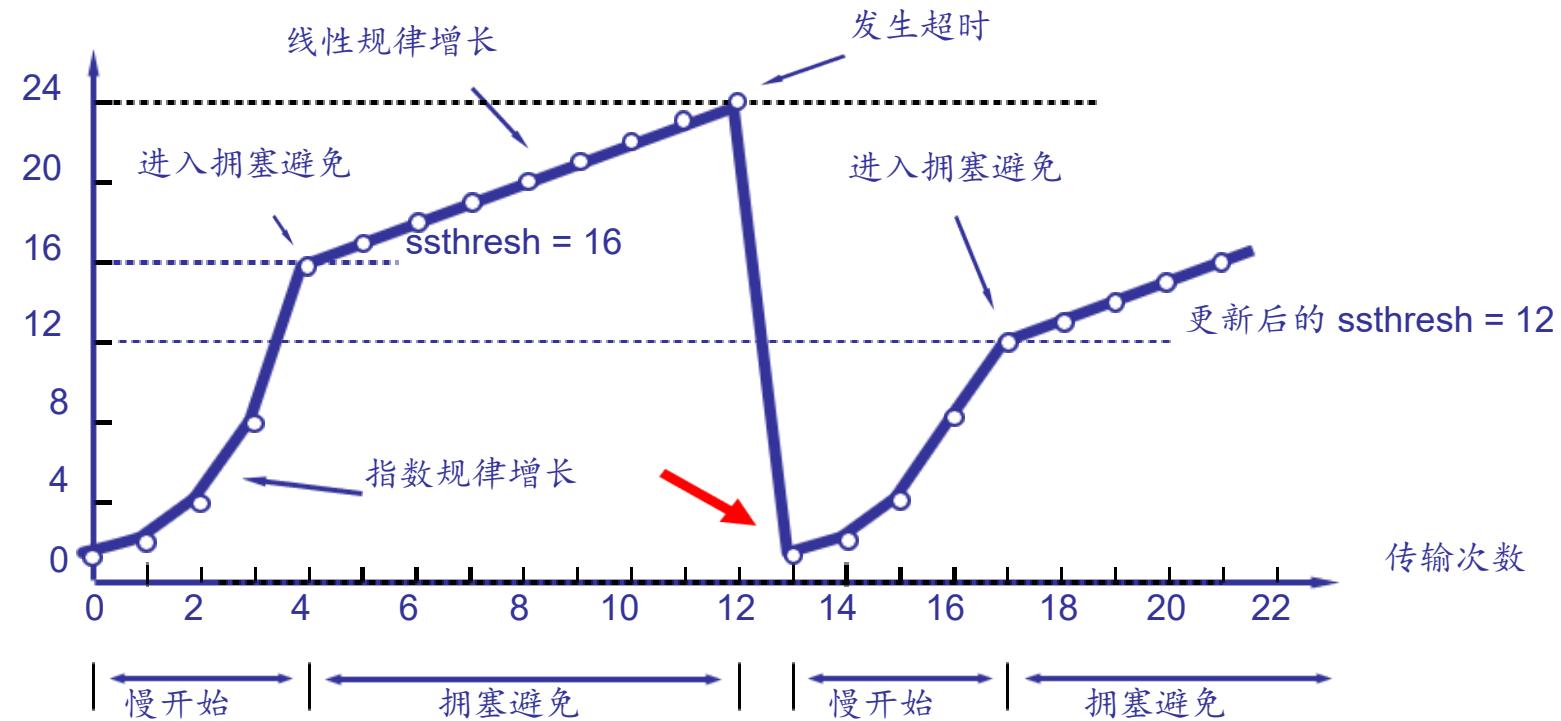
当拥塞窗口 cwnd 增长到慢开始门限值 ssthresh 时（即当 cwnd = 16 时），就改为执行拥塞避免算法，拥塞窗口按线性规律增长。



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）。

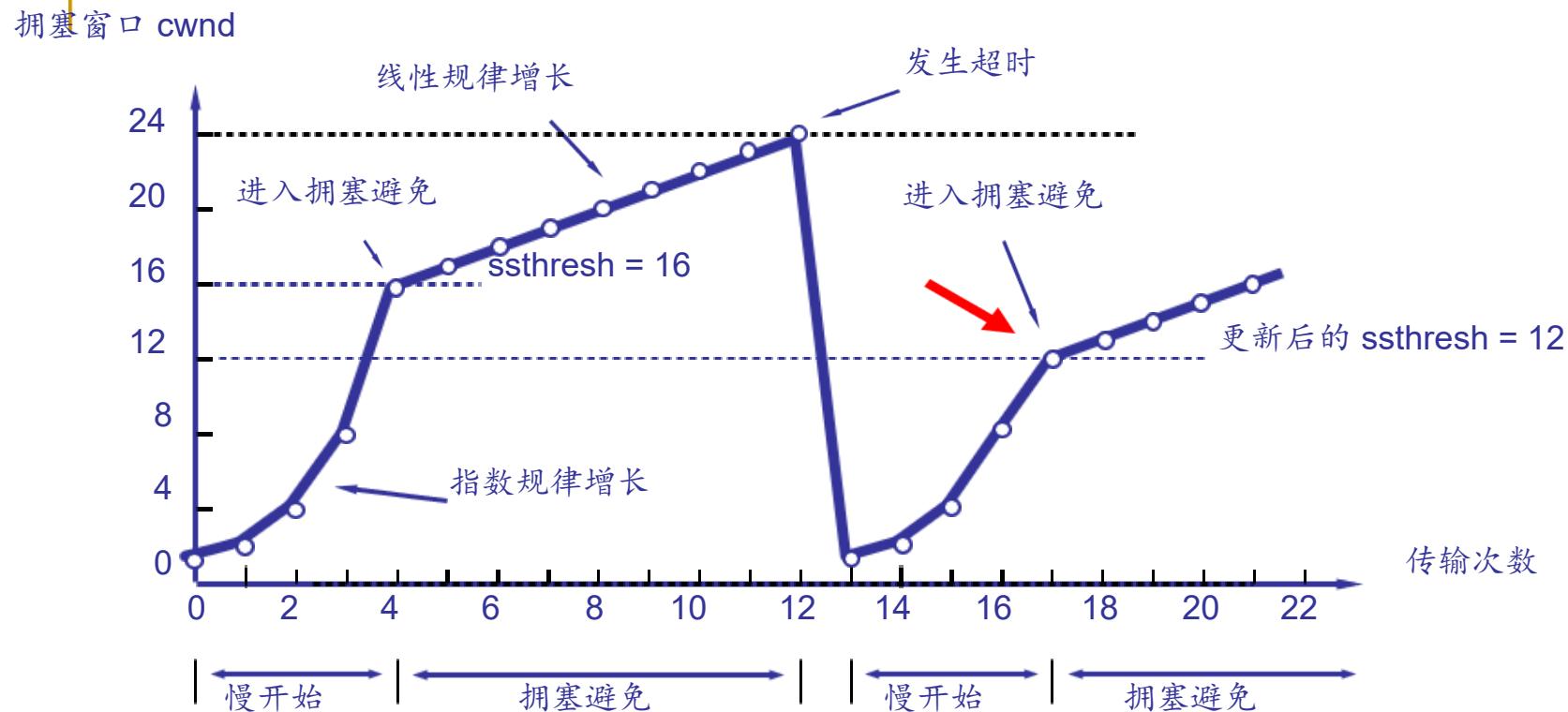
拥塞窗口 cwnd

慢开始和拥塞避免算法的实现举例



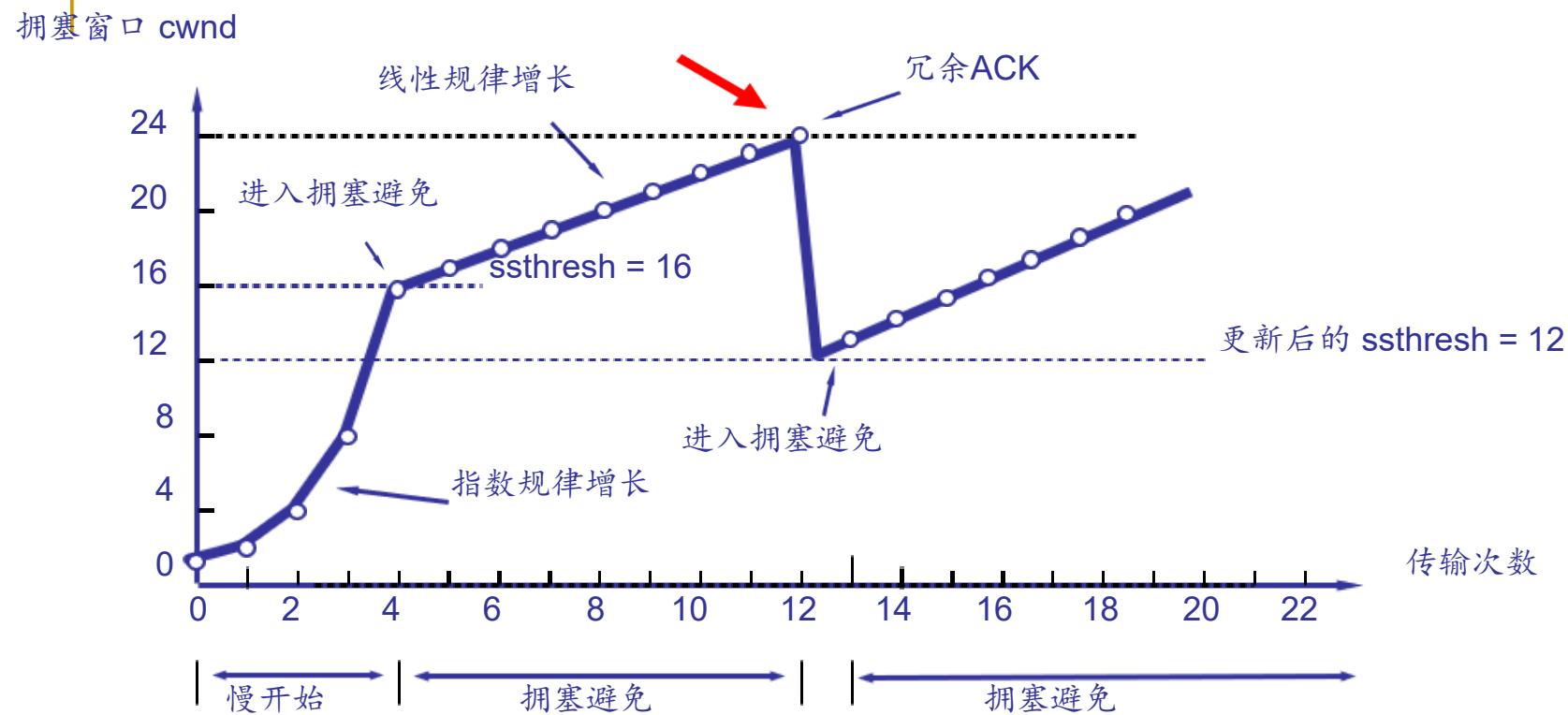
更新后的 ssthresh 值变为 12（即发送窗口数值 24 的一半），拥塞窗口再重新设置为 1，并执行慢开始算法。

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

□ 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。
- 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	$Threshold = CongWin/2,$ $CongWin = Threshold,$ 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) 每发送一组报文段, 仅返回一个ACK;
- (4) 仅考虑传播时延, 不考虑传输时延;
- (5) 第9、36个报文段在传输过程中发生超时重传;
- (6) 连续四次收到了对第24个报文段的ACK;
- (7) 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

- Reno算法的演进——Vegas算法
 - 通过往返时延的变化检测拥塞的严重程度
 - 往返时延越长，拥塞越严重
 - 当检测的拥塞达到一定程度时，有意识的线性降低发送速率以避免拥塞

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：

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

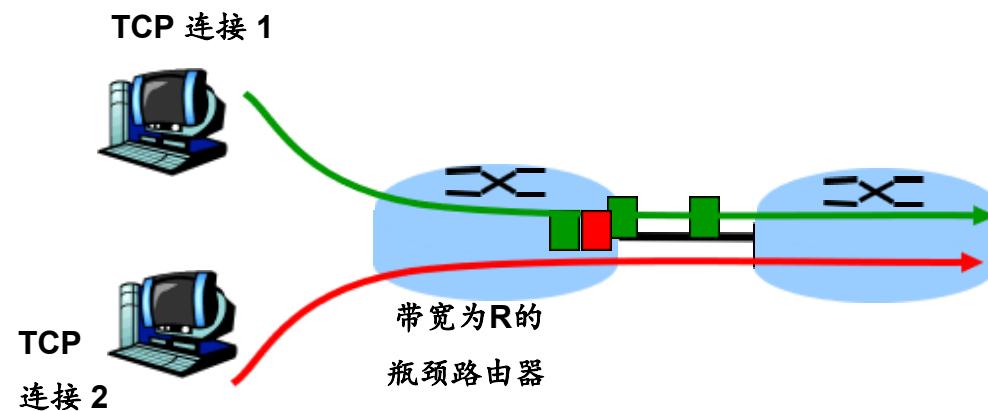
- 对于一条MSS=1500字节，RTT=100ms的TCP连接而言，如果希望达到10Gbps的吞吐量，那么丢包率 L 不能高于 2×10^{-10}

3.7 TCP拥塞控制

■ TCP拥塞控制的公平性分析

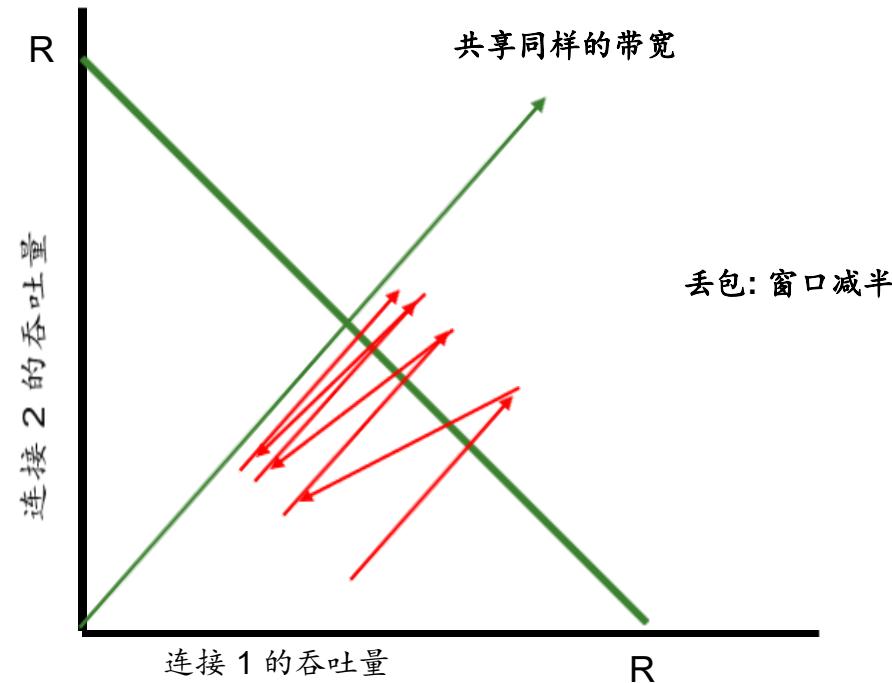
□ 公平性的目标

- 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



2020年10月6日

130

3.7 TCP拥塞控制

□ 公平性和UDP

- 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

- 无法阻止应用在两个主机之间建立多个并行的连接。

- Web浏览器就是这样

- 例子：速率为R的链路当前支持9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，获得R/2的速率！

3.7 TCP拥塞控制

■ TCP时延建模

□ 假定

- 假设客户机与服务器间链路传输速率为 R
- MSS大小为 S (比特), 对象大小为 O (比特)
- 没有重传(无丢包和损坏)
- 拥塞窗口大小为 W 个报文段

□ 影响时延的因素

- TCP建立连接
- 数据传输延时
- 慢启动

3.7 TCP拥塞控制

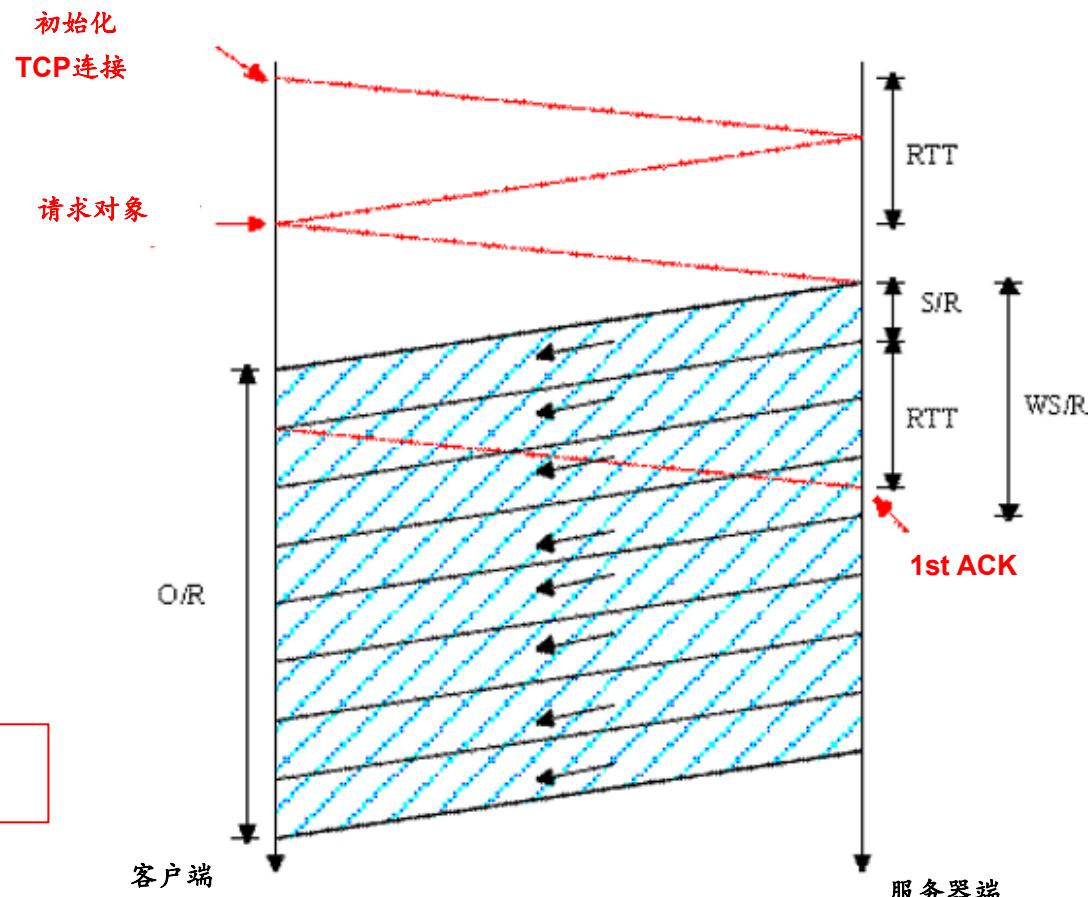
□ 静态拥塞窗口

■ 情况1：服务

器在完成窗口
的传输之前就
已经收到了窗
口中第一个报
文段的确认

$$WS/R > RTT + S/R$$

$$delay = 2RTT + O/R$$



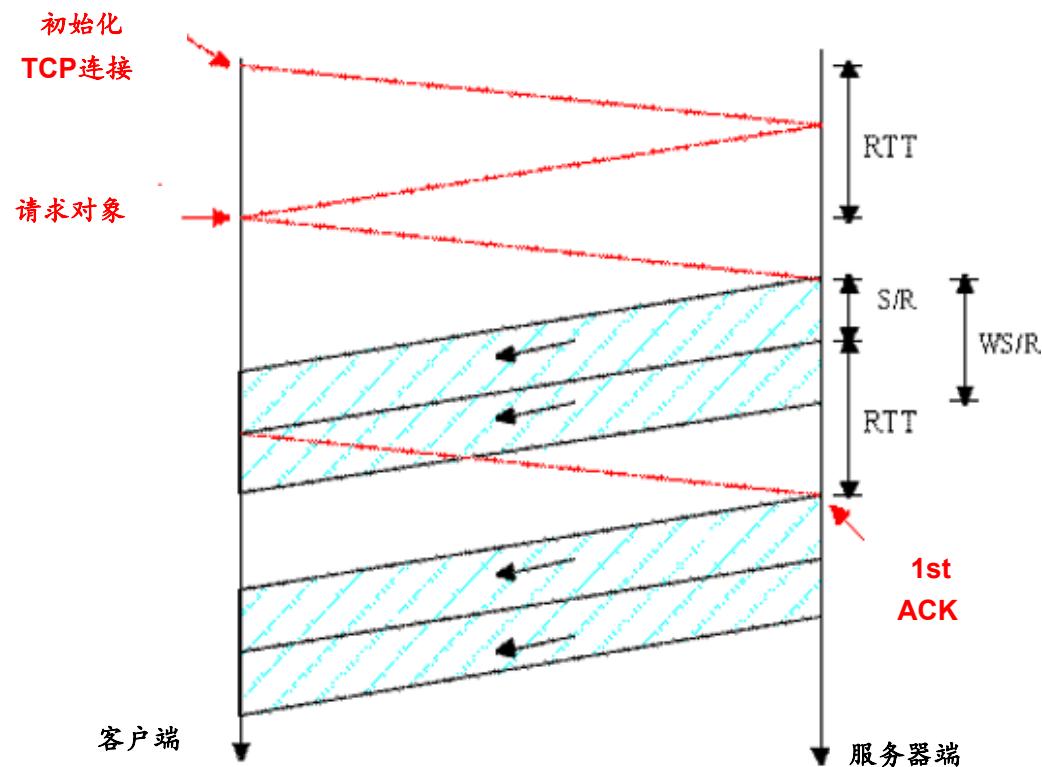
3.7 TCP拥塞控制

□ 静态拥塞窗口

- 情况2：服务
器收到窗口中
第一个报文段
的确认之前已
经将窗口中所
有报文段传输
完毕

$$WS/R < RTT + S/R$$

$$\begin{aligned} delay = & 2RTT + O/R \\ & + (K-1)[S/R + RTT - WS/R] \end{aligned}$$



3.7 TCP拥塞控制

□ 动态拥塞窗口

■ 引入三个新的变量

- K : 涵盖传输对象的窗口数量
- Q : 当对象包含无数个报文段时服务器的空闲次数
- P : 服务器TCP空闲的次数

$$P = \min\{Q, K - 1\}$$

3.7 TCP拥塞控制

$\frac{S}{R} + RTT =$ 从服务器开始发送窗口 内的报文段到

收到窗口第一个报文段 的确认所经历的时间

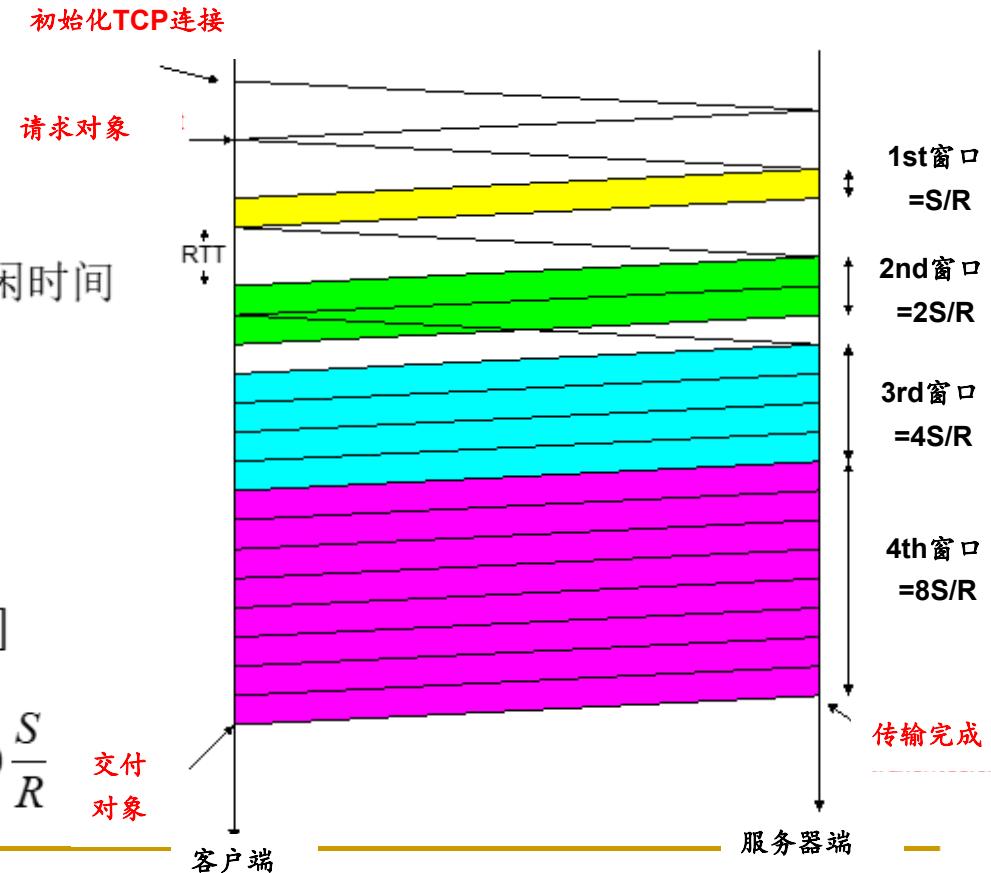
$$2^{k-1} \frac{S}{R} = \text{第 } k \text{ 个窗口的传输时间}$$

$$\left[\frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right]^+ = \text{第 } k \text{ 个窗口后的空闲时间}$$

$$\text{delay} = \frac{O}{R} + 2RTT + \sum_{p=1}^P \text{idleTime}_p$$

$$= \frac{O}{R} + 2RTT + \sum_{k=1}^P \left[\frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right]$$

$$= \frac{O}{R} + 2RTT + P \left[RTT + \frac{S}{R} \right] - (2^P - 1) \frac{S}{R}$$



2020年10月6日

136

3.7 TCP拥塞控制

□ 动态拥塞窗口

$$Latency = 2RTT + \frac{O}{R} + P \left[RTT + \frac{S}{R} \right] - (2^P - 1) \frac{S}{R}$$

$$P = \min \{Q, K - 1\}$$

$$K = \left\lceil \log_2 \left(\frac{O}{S} + 1 \right) \right\rceil \quad Q = \left\lfloor \log_2 \left(1 + \frac{RTT}{S/R} \right) \right\rfloor + 1$$

3.7 TCP拥塞控制

■ 时延的进一步讨论

- 无拥塞控制时，时延为 $2RTT+O/R$ ，该时延定义为最小延迟时间

$$\frac{\text{时延}}{\text{最小时延}} \leq 1 + \frac{P}{[(O/R)/RTT] + 2}$$

- 如果 $RTT \ll O/R$, TCP慢启动几乎不会显著的增加延迟时间

3.7 TCP拥塞控制

■ 几个例子

□ 假设: $S=536$ 字节、 $RTT=100ms$ 、 $O=100kB$ 、

$K=8$

R	O/R	P	最小时延	具慢启动的时延
28 kb/s	28.6 s	1	28.8 s	28.9 s
100 kb/s	8 s	2	8.2 s	8.4 s
1 Mb/s	800 ms	5	1 s	1.5 s
10 Mb/s	80 ms	7	0.28 s	0.98 s

□ 速率越高, 相对而言, 慢启动的影响越大

3.7 TCP拥塞控制

■ 几个例子

□ 假设：S=536字节、RTT=100ms、O=5kB、

K=4

R	O/R	P	最小时延	具慢启动的时延
28 kb/s	1.43 s	1	1.63 s	1.73 s
100 kb/s	0.4 s	2	0.6 s	0.76 s
1 Mb/s	40 ms	3	0.24 s	0.52 s
10 Mb/s	4 ms	3	0.20 s	0.50 s

□ 对象越小，慢启动的影响越大

3.7 TCP拥塞控制

■ 几个例子

- 假设: $S=536$ 字节、 $RTT=1s$ 、 $O=5kB$ 、 $K=4$

R	O/R	P	最小时延	具慢启动的时延
28 kb/s	1.43 s	3	3.4 s	5.8 s
100 kb/s	0.4 s	3	2.4 s	5.2 s
1 Mb/s	40 ms	3	2.0 s	5.0 s
10 Mb/s	4 ms	3	2.0 s	5.0 s

- RTT越大, 慢启动的影响越大

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