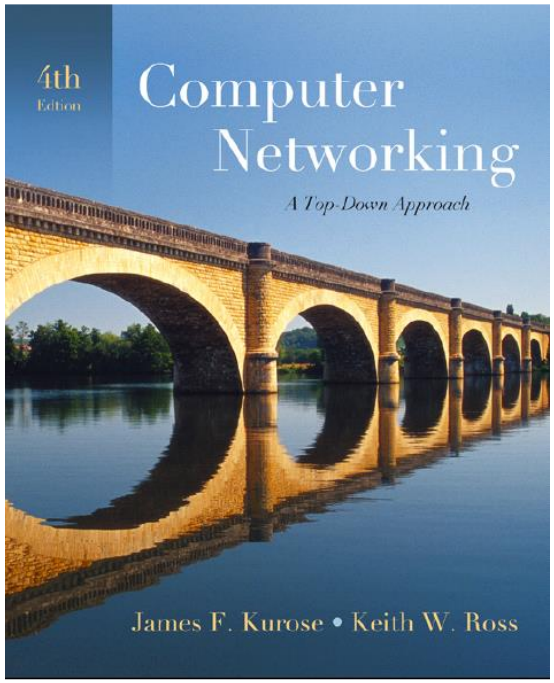
第三章 传输层


4th
Edition
Computer
Networking
A Top-Down Approach
James F. Kurose • Keith W. Ross

学习目标和主要内容

我们的目的:

- 理解运输层服务依据的原理:
  - 复用/分解
  - 可靠数据传输
  - 流量控制
  - 拥塞控制
- 学习因特网中的运输层协议:
  - UDP: 无连接传输
  - TCP: 面向连接传输
  - TCP 拥塞控制

教学内容及要求

本章主要介绍传输层的基本功能和实现技术，包括TCP和UDP两个传输层协议的基本功能、多路复用和多路分解的方法、可靠数据传输技术、面向连接的实现机制、流量控制和拥塞控制的原理和实现方法。

掌握：

- 多路复用和多路分解
- UDP的实现方法和相关传输原理
- 校验和的计算方法
- 可靠数据传输原理
- TCP的实现方法和相关原理
- 拥塞控制原理
- TCP拥塞控制

理解：

- 运输层的基本概念和服务
- TCP拥塞控制的公平性

自学：

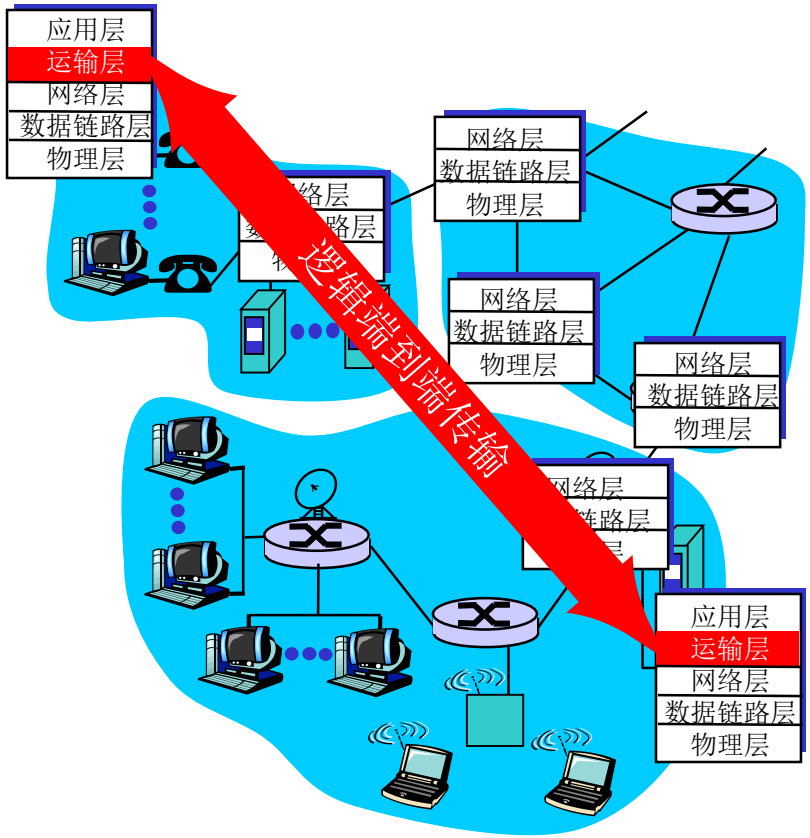
- ATM ABR拥塞控制
- TCP时延建模

第3章 提纲

- 3.1 运输层服务
- 3.2 多路复用与多路分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输原理
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制原理
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

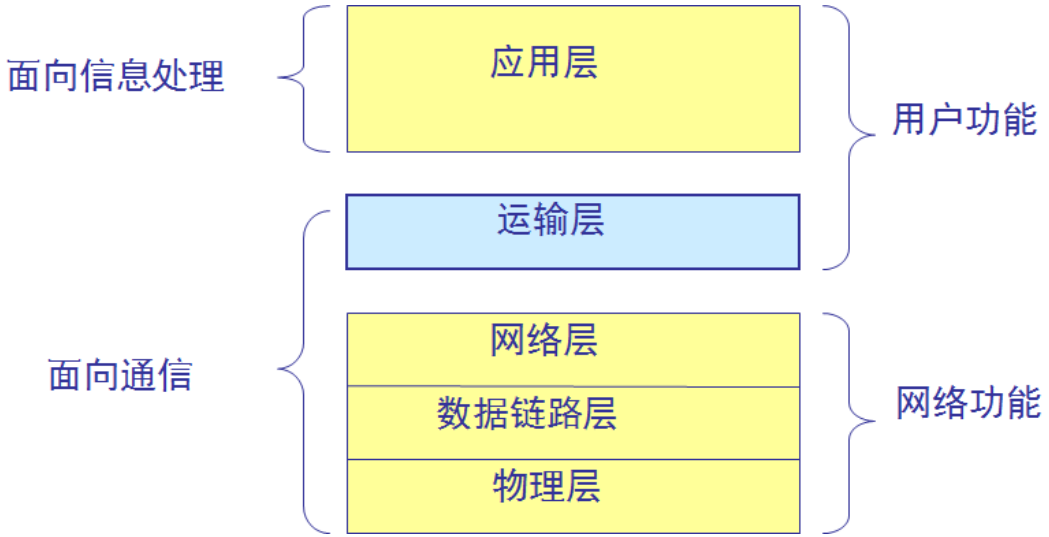
3.1 运输服务和协议

- 在运行不同主机上应用进程之间提供*逻辑通信*
- 运输协议运行在端系统中
  - 发送方：将应用报文划分为*段*，传向网络层
  - 接收方：将段重新装配为报文，传向应用层
- 应用可供使用的运输协议不止一个
  - 因特网：TCP和UDP



3.1 运输服务和协议

- 从通信和信息处理的角度看，运输层向它上面的应用层提供通信服务，它属于面向通信部分的最高层，同时也是用户功能中的最低层。



3.1.1 运输层 vs. 网络层

- *网络层:* 主机间的逻辑通信
- *运输层:* 进程间的逻辑通信
  - 依赖、强化网络层服务

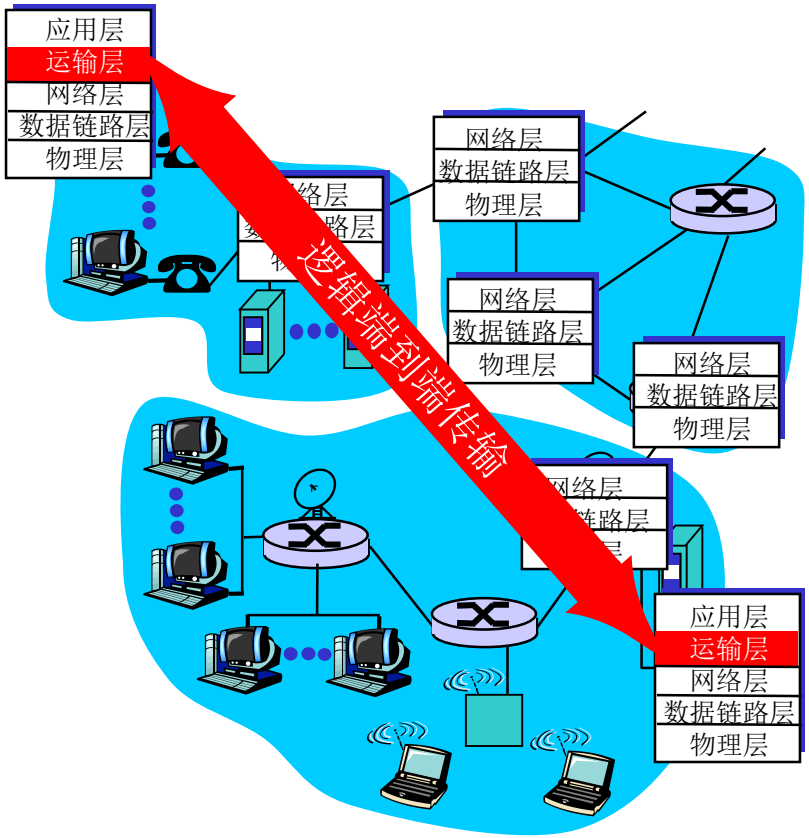
家庭类比:

12个孩子向12个孩子发信

- 进程 = 孩子
- 应用报文= 信封中的信
- 主机 = 家庭
- 运输协议 = Ann和Bill
- 网络层协议= 邮政服务

3.1.2 因特网运输层协议

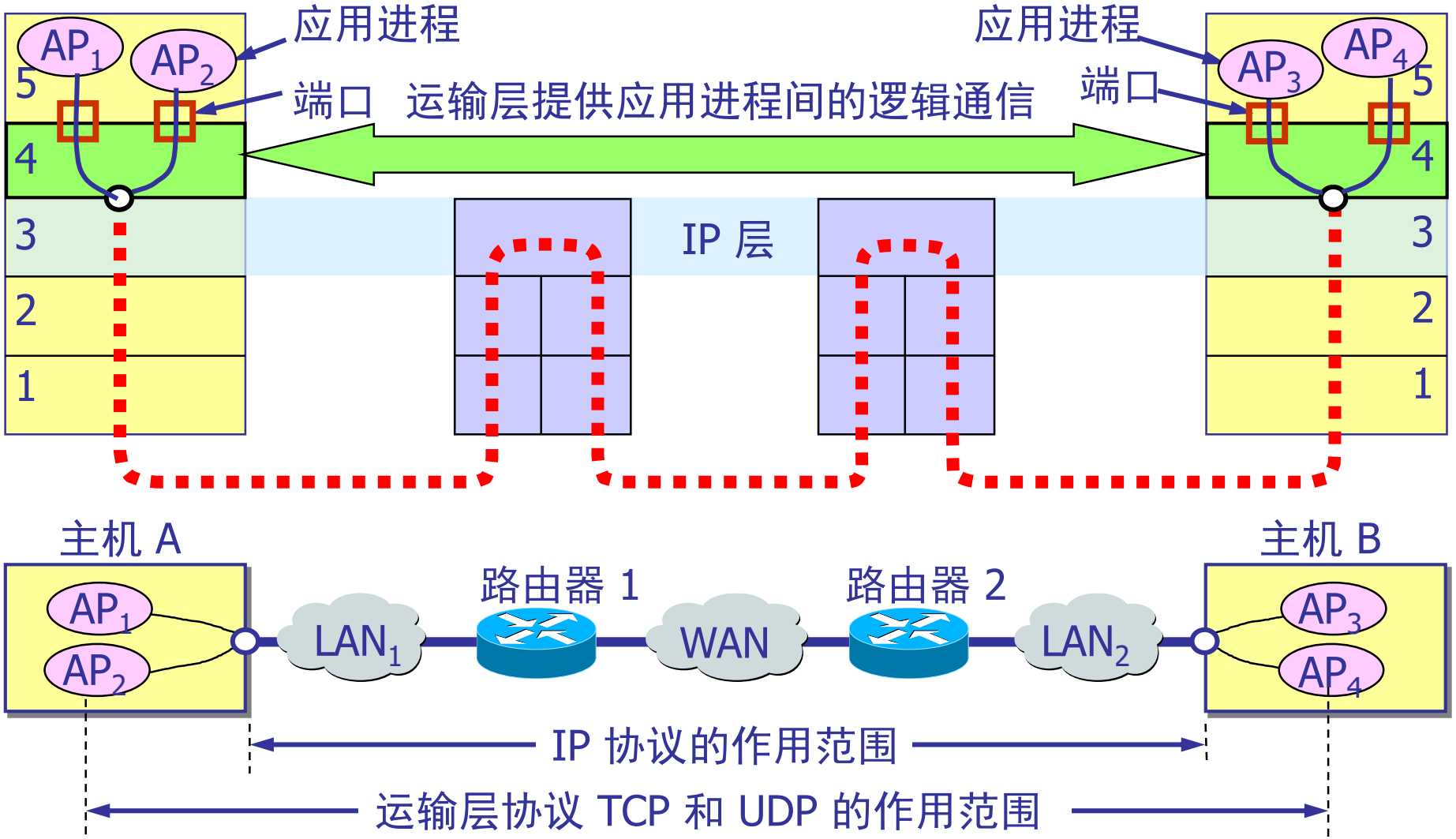
- 可靠的、按序的交付(TCP)
  - 拥塞控制
  - 流量控制
  - 连接建立
- 不可靠、不按序交付: UDP
  - “尽力而为”，不提供不必要服务的扩展
- 不可用的服务:
  - 时延保证
  - 带宽保证



第3章 要点

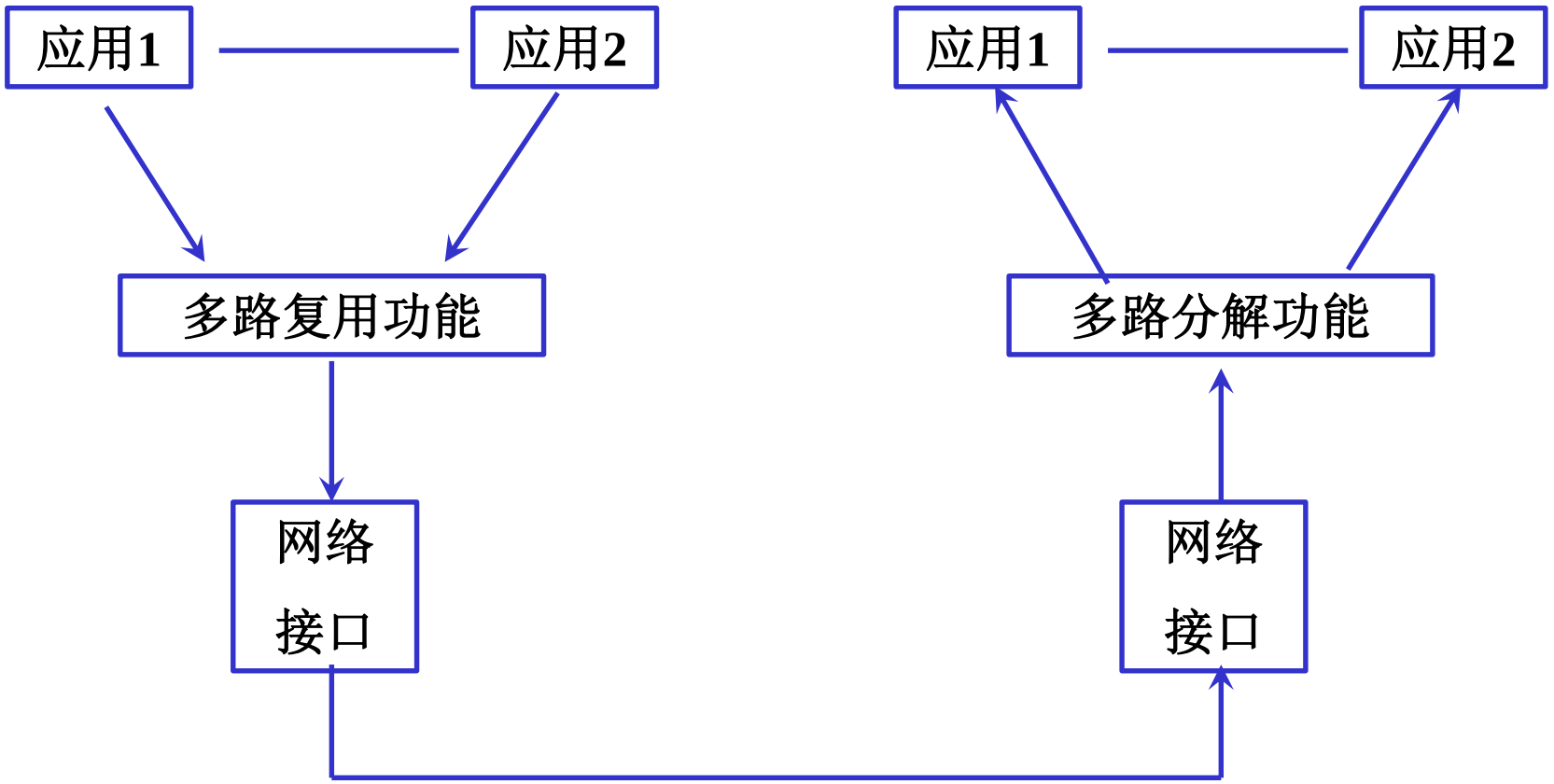
- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

- 运输层解决的是计算机进程到计算机进程之间的通信问题，即所谓的“端”到“端”的通信。



- 网络层实现的是主机到主机之间的通信。

3.2 Internet 层的复用与分解



复用/分解

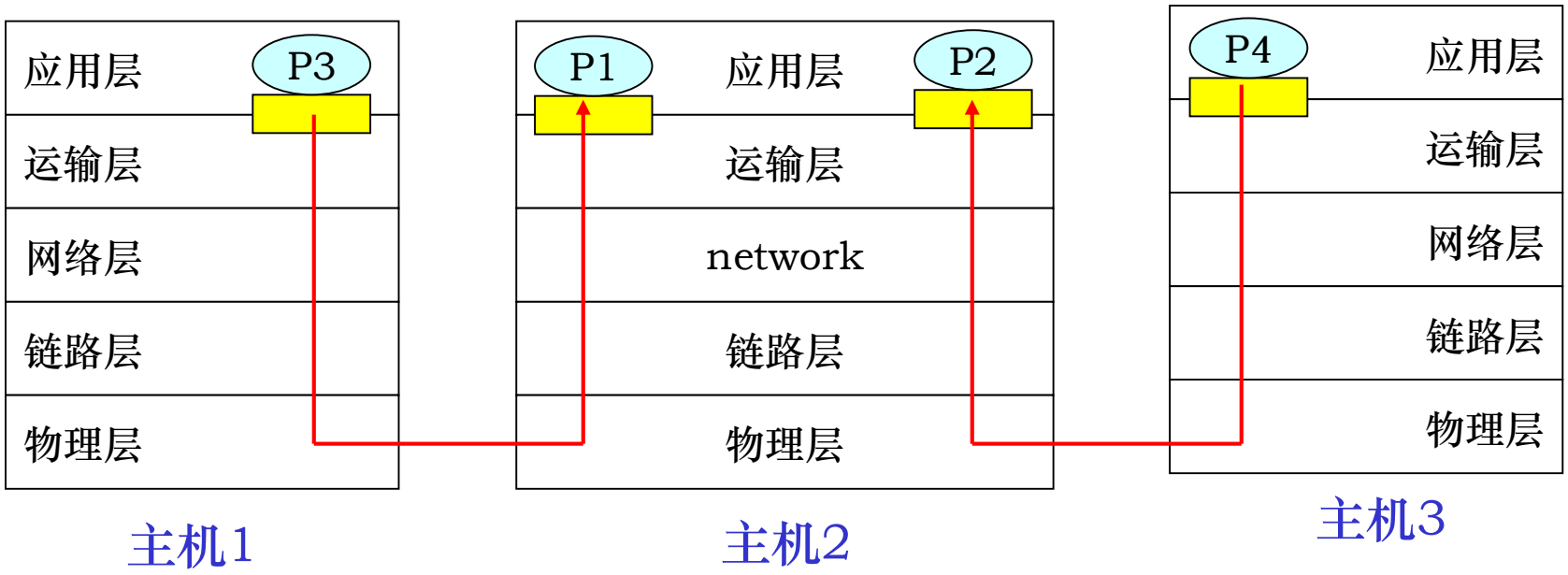
在接收主机分解：

将接收到的段交付给正确的套接字

在发送主机复用:

从多个套接字收集数据，用首部封装数据(以后用于分解)

= 套接字　　= 进程



分解工作过程

- 主机接收IP数据报
  - 每个数据报承载1个运输层段
  - 每个段具有源、目的端口号
    (回想: 对特定应用程序的周知端口号)
- 主机使用IP地址 &端口号将段定向到适当的套接字

TCP/UDP 段格式

1、无连接分解

- 生成具有端口号的套接字:

```
DatagramSocket mySocket1 = new
    DatagramSocket(9911);
serverSocket = socket(AF_INET,
    SOCK_DGRAM)
serverSocket.bind(('', serverPort))
```

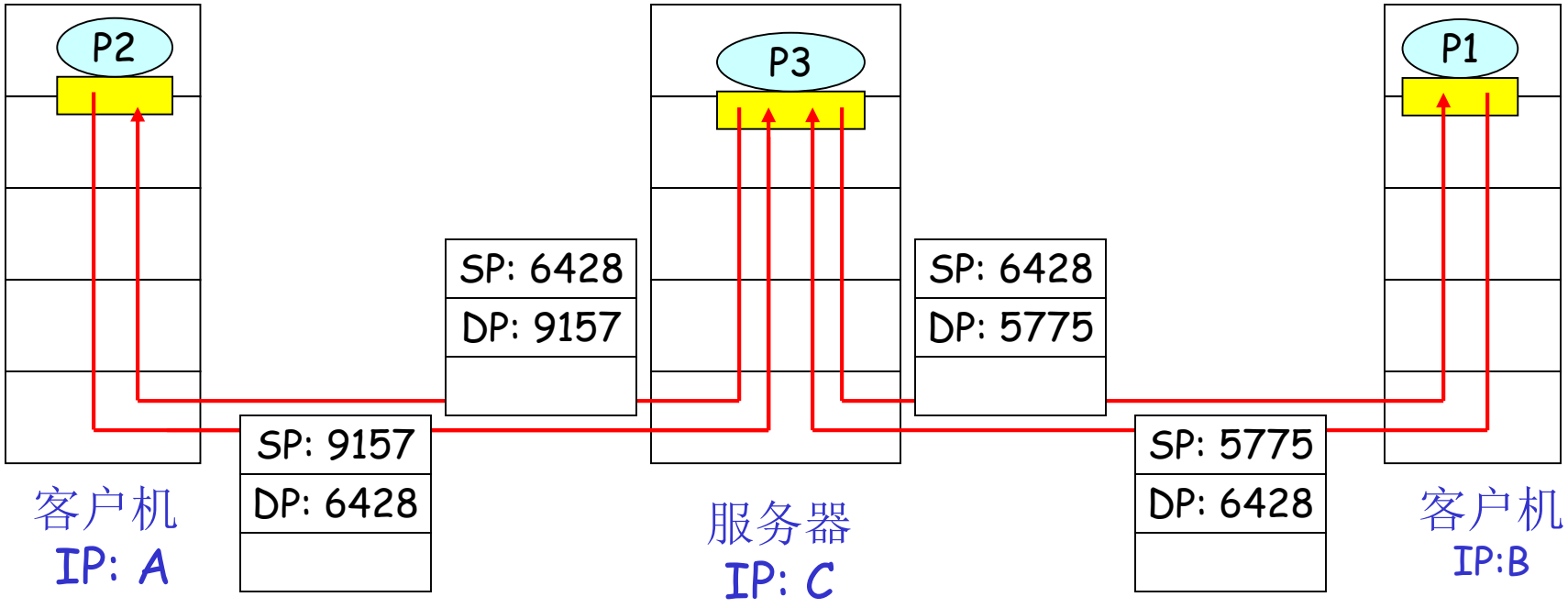
- UDP套接字由二元组标识：

(目的地IP地址，目的地端口号)

- 当主机接收UDP报文段时:
  - 在报文段中检查目的地端口号
  - 将UDP段定向到具有该端口号的套接字
- 具有不同的源IP地址且/或源端口号，但具有相同的目的IP地址和目的端口号的IP报文段指向同样的套接字

1、无连接分解(续)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

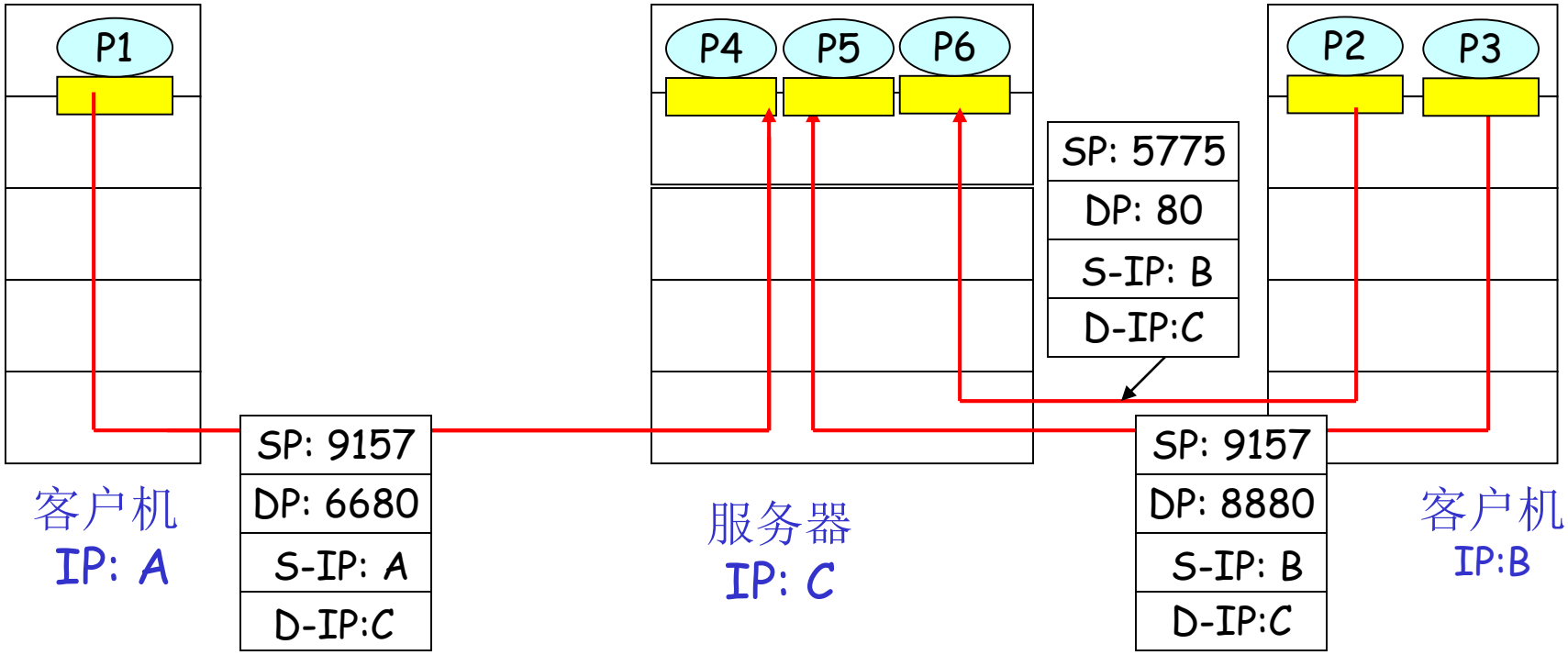


Sp提供了“返回地址”

2、面向连接分解

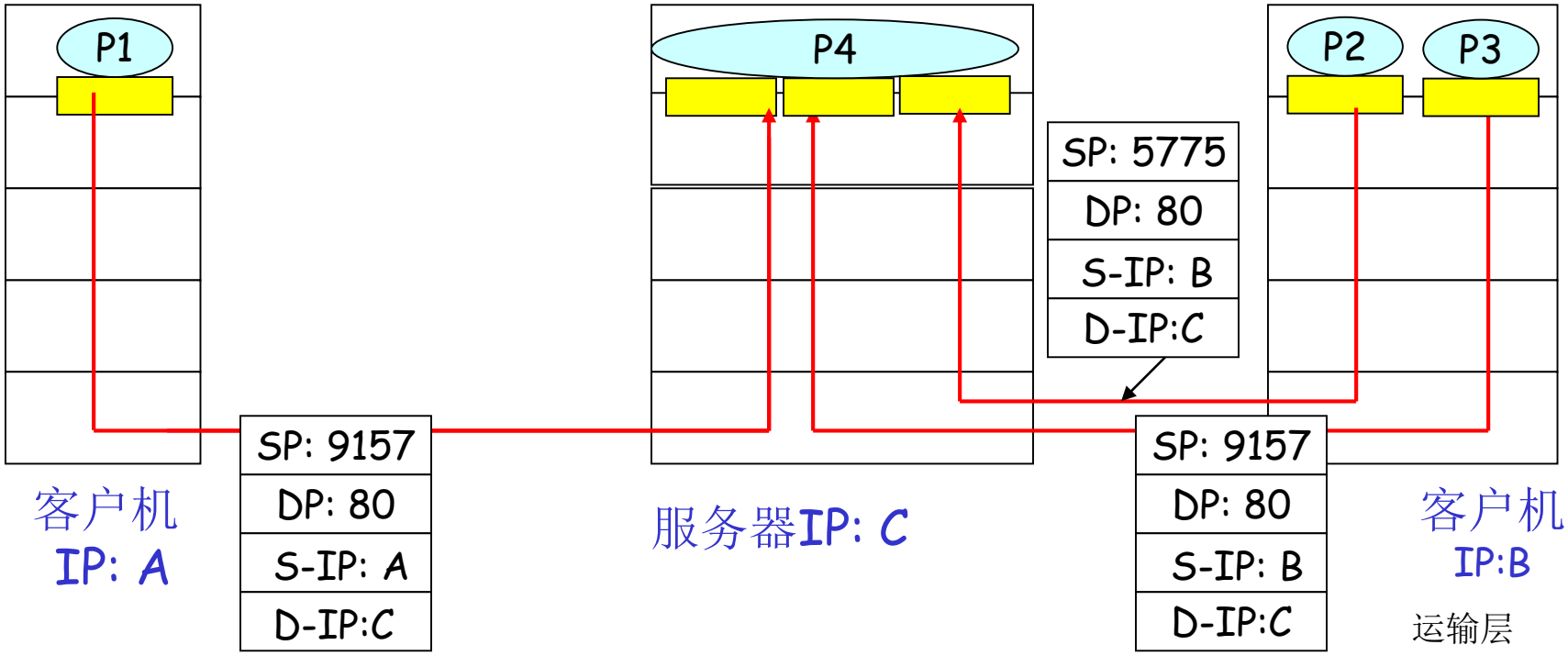
- TCP套接字由四元组标识:
  - 源IP地址
  - 源端口号
  - 目的IP地址
  - 目的端口号
- 接收主机使用这四个值来将段定向到适当的套接字
- 服务器主机可能支持许多并行的TCP套接字:
  - 每个套接字由其自己的四元组标识

2、面向连接分解 (续)


P1
P4
P5
P6
P2
P3
SP: 5775
DP: 80
S-IP: B
D-IP:C
SP: 9157
DP: 6680
S-IP: A
D-IP:C
SP: 9157
DP: 8880
S-IP: B
D-IP:C
客户机
IP: A
服务器
IP: C
客户机
IP:B

3、面向连接分解: 多线程Web服务器

- 所有来至客户机的的连接建立请求报文和HTTP请求报文，都具有相同的目的端口号：80.
- Web服务器对每个连接的客户机具有不同的套接字
- 非持久HTTP将为每个请求具有不同的套接字



第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

3.3 UDP: 用户数据报协议 [RFC 768]

- “没有不必要的,” “基本要素” 互联网传输协议
- “尽力而为”服务，UDP段可能：
  - 丢包
  - 对应用程序交付失序
- *无连接:*
  - 在UDP发送方和接收方之间无握手
  - 每个UDP段的处理独立于其他段

为何要有 UDP协议?

- 无连接创建(它将增加时延)
- 简单：在发送方、接收方无连接状态
- 段首部小
- 无拥塞控制: UDP能够尽可能快地传输

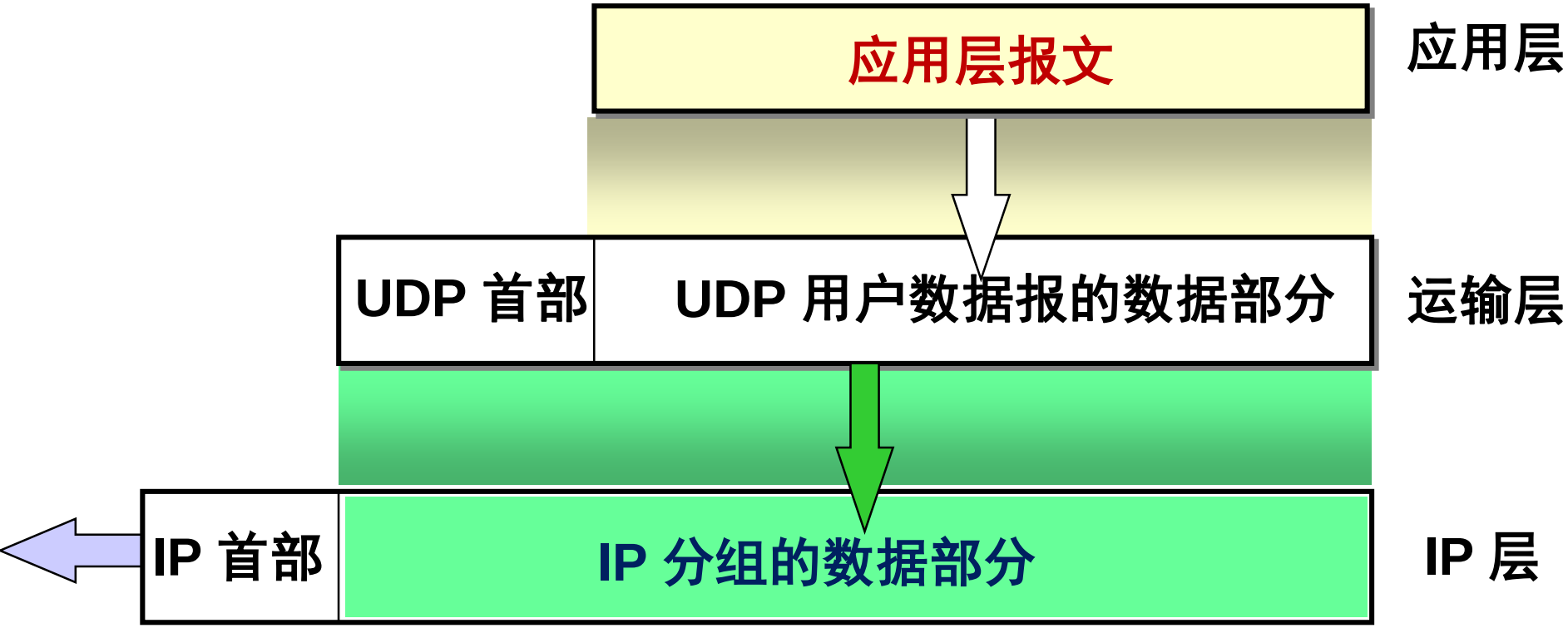
3.3 UDP: 用户数据报协议 [RFC 768]

- UDP 只在 IP 的数据报服务之上增加了很少一点的功能，即端口的复用/分解功能和差错检测的功能。
- 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 是无连接的，即发送数据之前不需要建立连接。
- UDP 使用尽最大努力交付，即不保证可靠交付，同时也不使用拥塞控制。
- UDP 没有拥塞控制，很适合多媒体通信的要求。
- UDP 支持一对一、一对多、多对一和多对多的交互通信。

3.3 UDP: 用户数据报协议 [RFC 768]

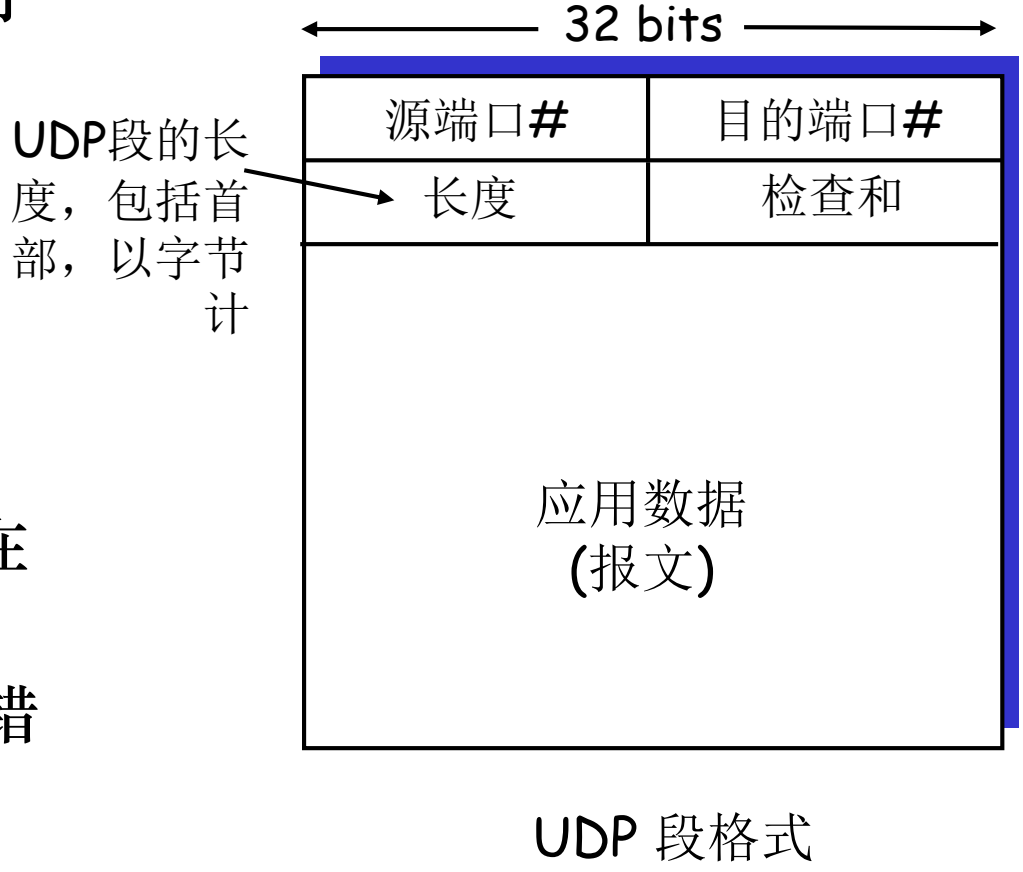
- UDP 的首部开销小，只有 8 个字节。
- UDP 是面向报文的。发送方 UDP 对应用程序交下来的报文，在添加首部后就向下交付 IP 层。UDP 对应用层交下来的报文，既不合并，也不拆分，而是保留这些报文的边界。
- 应用层交给 UDP 多长的报文，UDP 就照样发送，即一次发送一个报文。
- 接收方 UDP 对 IP 层交上来的 UDP 用户数据报，在去除首部后就原封不动地交付上层的应用进程，一次交付一个完整的报文。
- 应用程序必须选择合适大小的报文。

UDP 是面向报文的



3.3.1 UDP: 其他

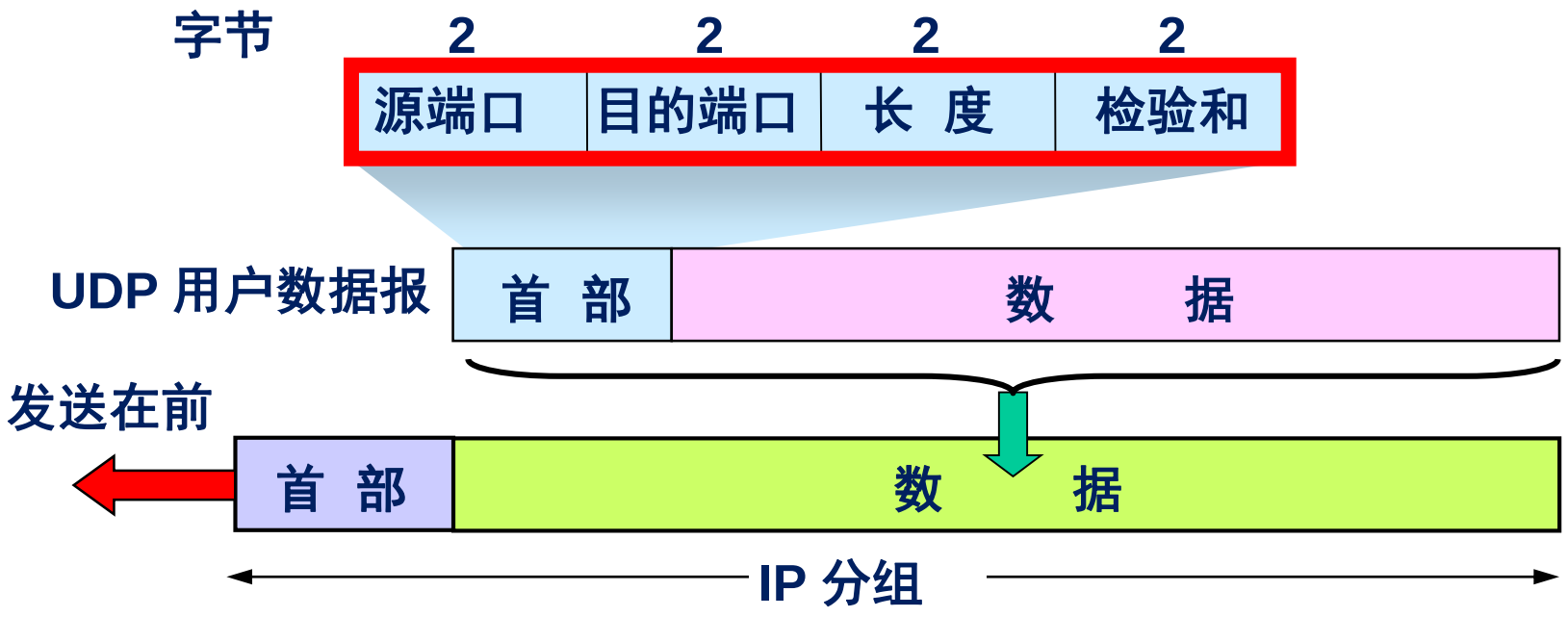
- 常用于流式多媒体应用
  - 丢包容忍
  - 速率敏感
- 其他UDP应用
  - DNS
  - SNMP
- 经UDP的可靠传输：在应用层增加可靠性
  - 应用程序特定的差错恢复!



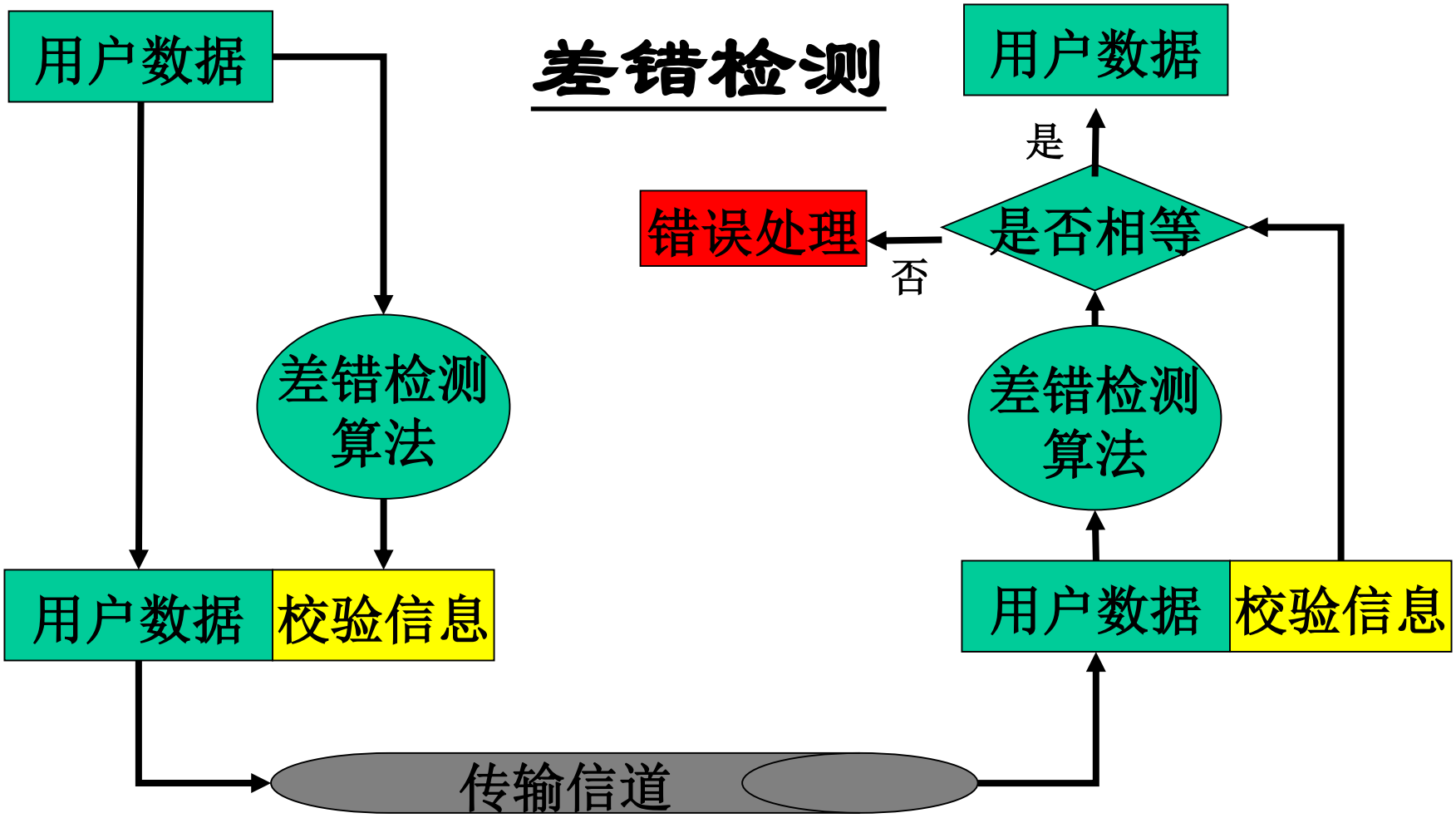
UDP 段格式

UDP 的首部格式

用户数据报 UDP 有两个字段：数据字段和首部字段。首部字段有 8 个字节，由 4 个字段组成，每个字段都是两个字节。长度是首部和数据的总长度



3.3.2 UDP检验和



错误检测不是100%可靠！

- 协议有可能漏掉一些错误，但很少
- 大的校验信息域能提供更好的检错能力

3.3.2 UDP检验和

目的: 在传输的段中检测“差错”（如比特翻转）

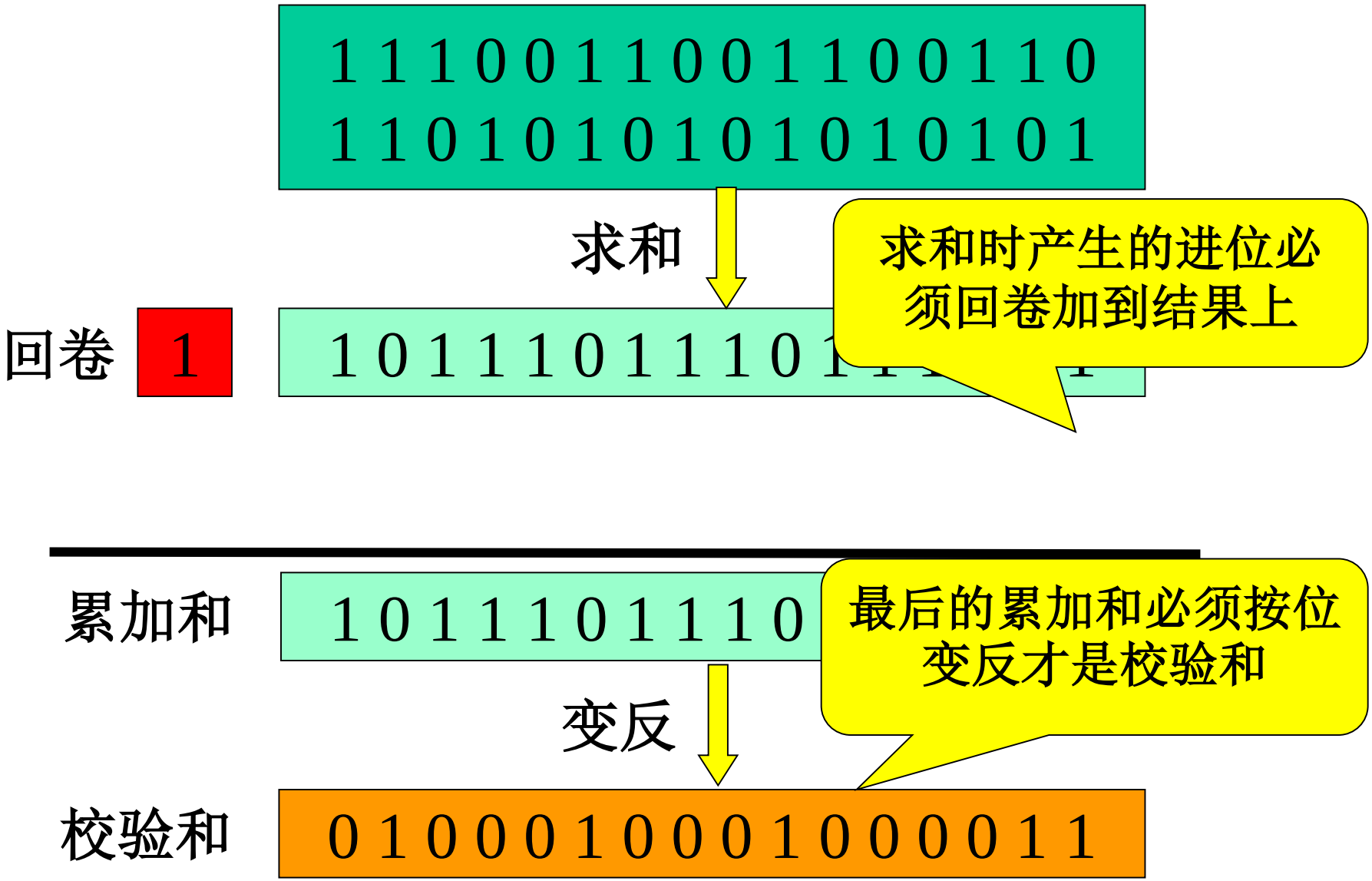
发送方:

- 将段内容处理为16比特整数序列
- 检验和: 段内容的加法(反码和)
- 发送方将检验和放入UDP检查和字段

接收方:

- 计算接收的段的检验和
- 核对计算的检验和是否等于检查和字段的值:
  - NO – 检测到差错
  - YES – 无差错检测到。*虽然如此，还可能有差错吗?* 详情见后……

互联网检验和例子

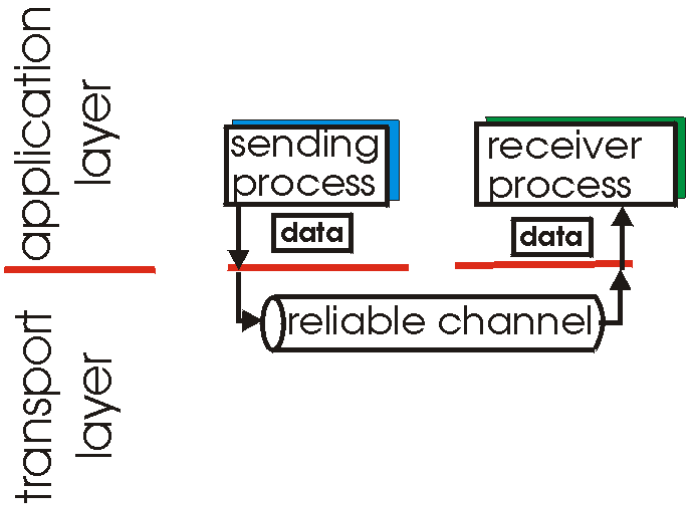

1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
求和
回卷
1
1 0 1 1 1 0 1 1 1 0
求和时产生的进位必须回卷加到结果上
累加和
1 0 1 1 1 0 1 1 1 0
最后的累加和必须按位变反才是校验和
变反
校验和
0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1

第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

3.4 可靠数据传输的原则

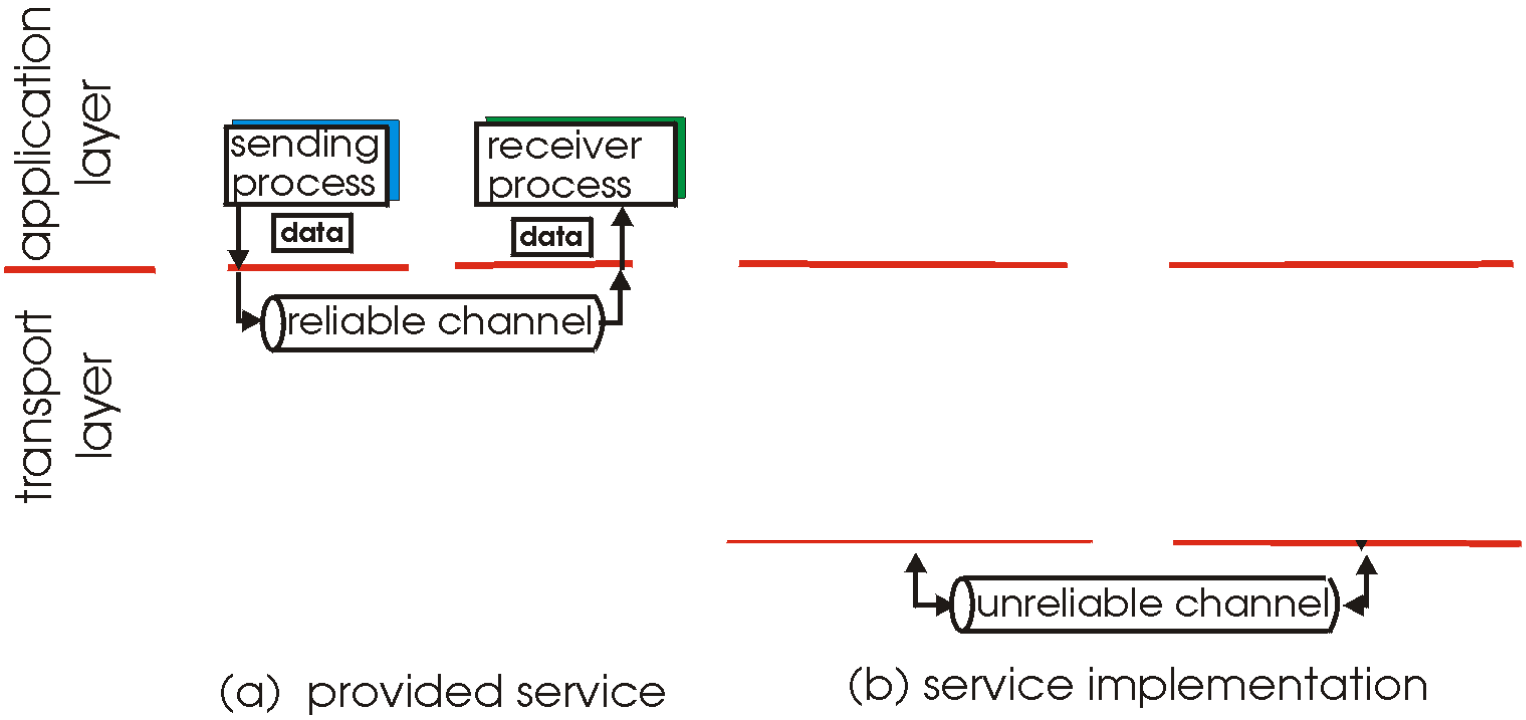
- 在应用层、运输层、数据链路层的重要性
  - **重要的网络主题中的最重要的10个之一!**



(a) provided service

3.4 可靠数据传输的原则

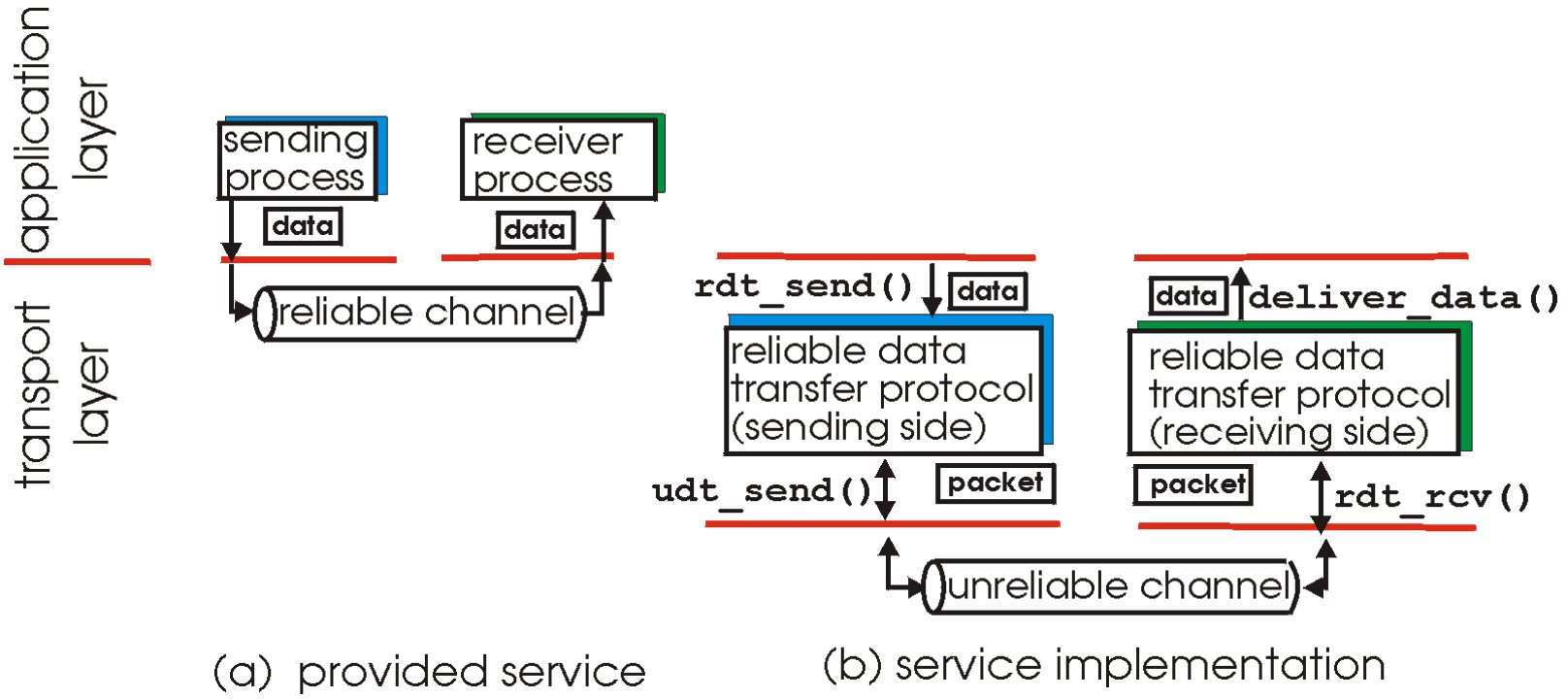
- 在应用层、运输层、数据链路层的重要性
  - **重要的网络主题中的最重要的10个之一!**



- 不可靠信道的特点决定了可靠数据传输 协议 (rdt) 的复杂性

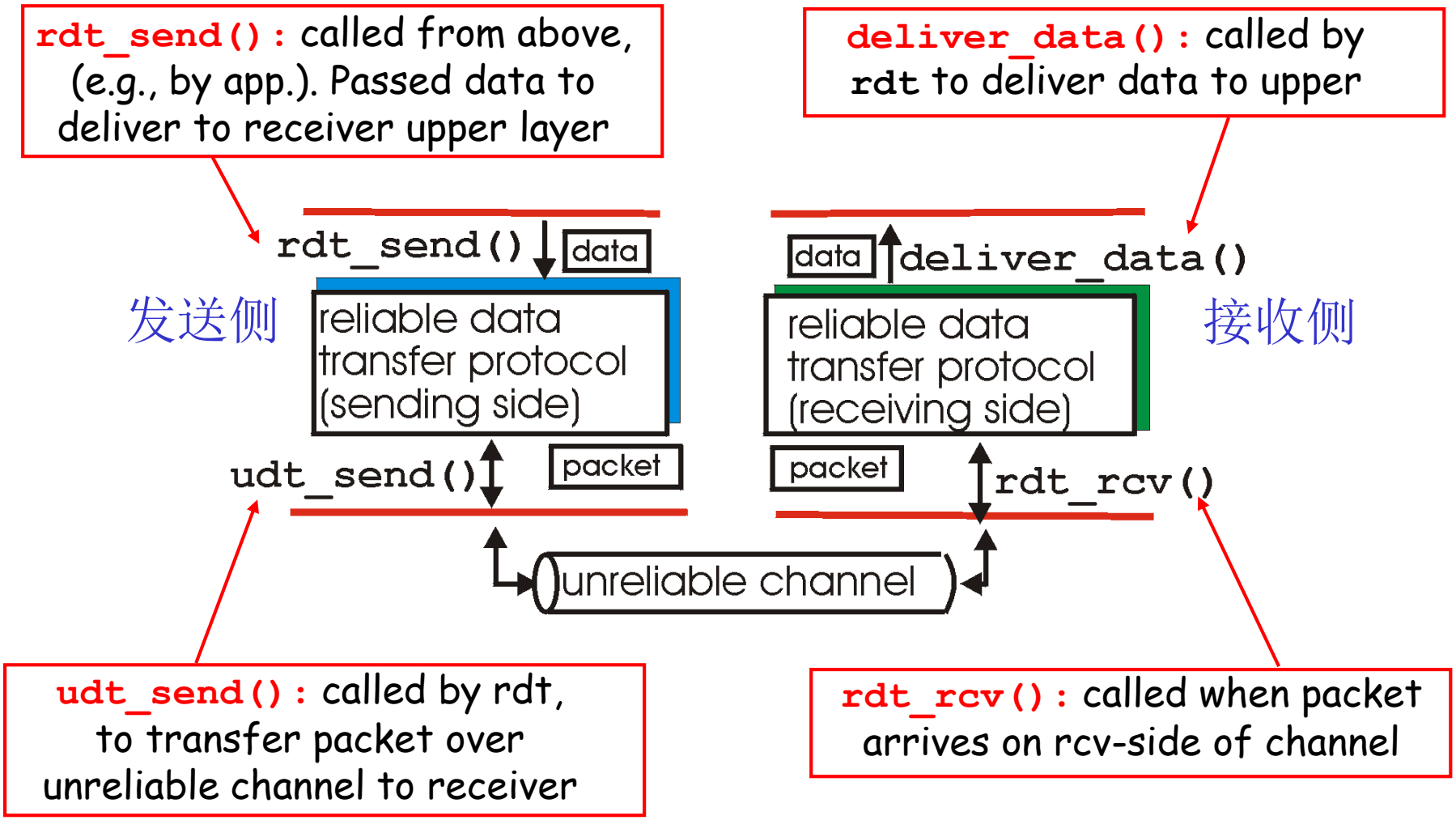
3.4 可靠数据传输的原则

- 在应用层、运输层、数据链路层的重要性
  - **重要的网络主题中的最重要的10个之一!**



- 不可靠信道的特点决定了可靠数据传输 协议 (rdt) 的复杂性

可靠数据传输: 基本概念

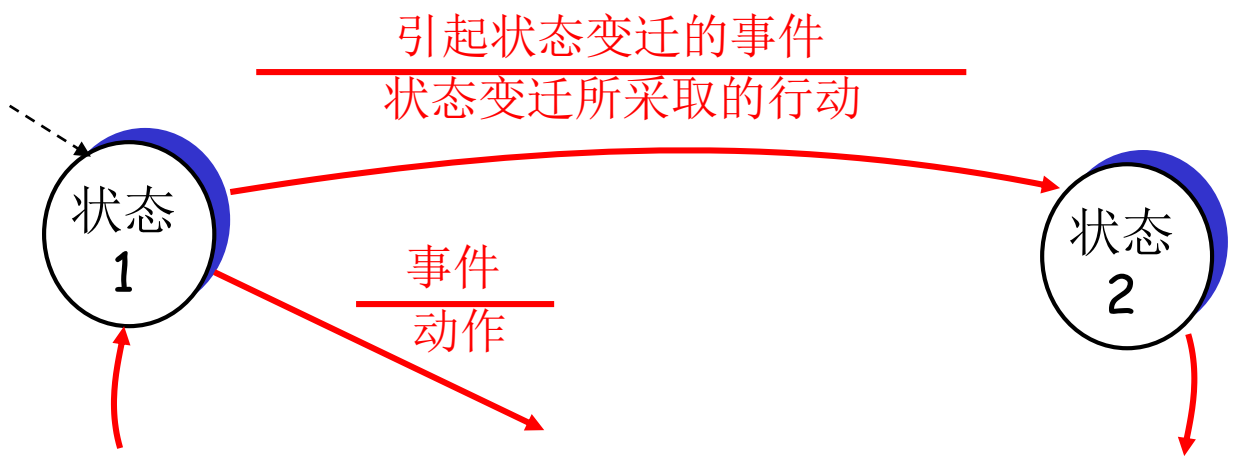

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
deliver_data(): called by rdt to deliver data to upper
rdt_send()
data
发送侧
reliable data transfer protocol (sending side)
udt_send()
packet
data
deliver_data()
reliable data transfer protocol (receiving side)
packet
rdt_rcv()
接收侧
unreliable channel
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver
rdt_rcv(): called when packet arrives on rcv-side of channel

3.4.1 构造可靠数据传输协议

我们将:

- 逐步开发发送方和接收方的可靠数据传输协议 (rdt)
  - 仅考虑单向数据传输
  - 但控制信息将在两个方向流动!
- 使用有限状态机 (FSM)来定义发送方和接收方

状态: 当位于这个“状态时”，下个状态惟一地由下个事件决定

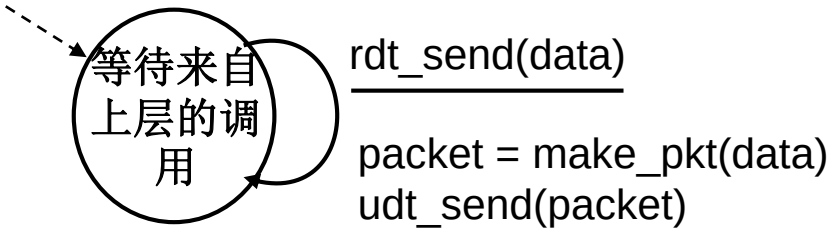


第3章 要点

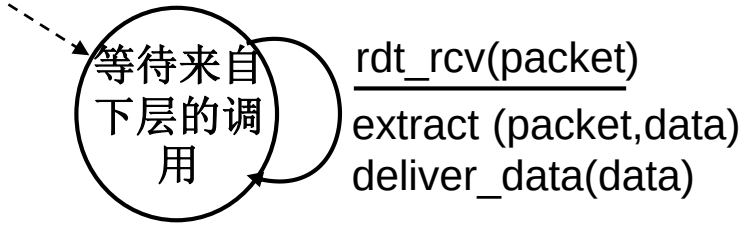
- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

1、Rdt1.0: 经可靠信道的可靠传输

- 底层信道非常可靠
  - 无比特差错
  - 无分组丢失
- 装发送方、接收方的单独FSM:
  - 发送方将数据发向底层信道
  - 接收方从底层信道读取数据



发送方



接收方

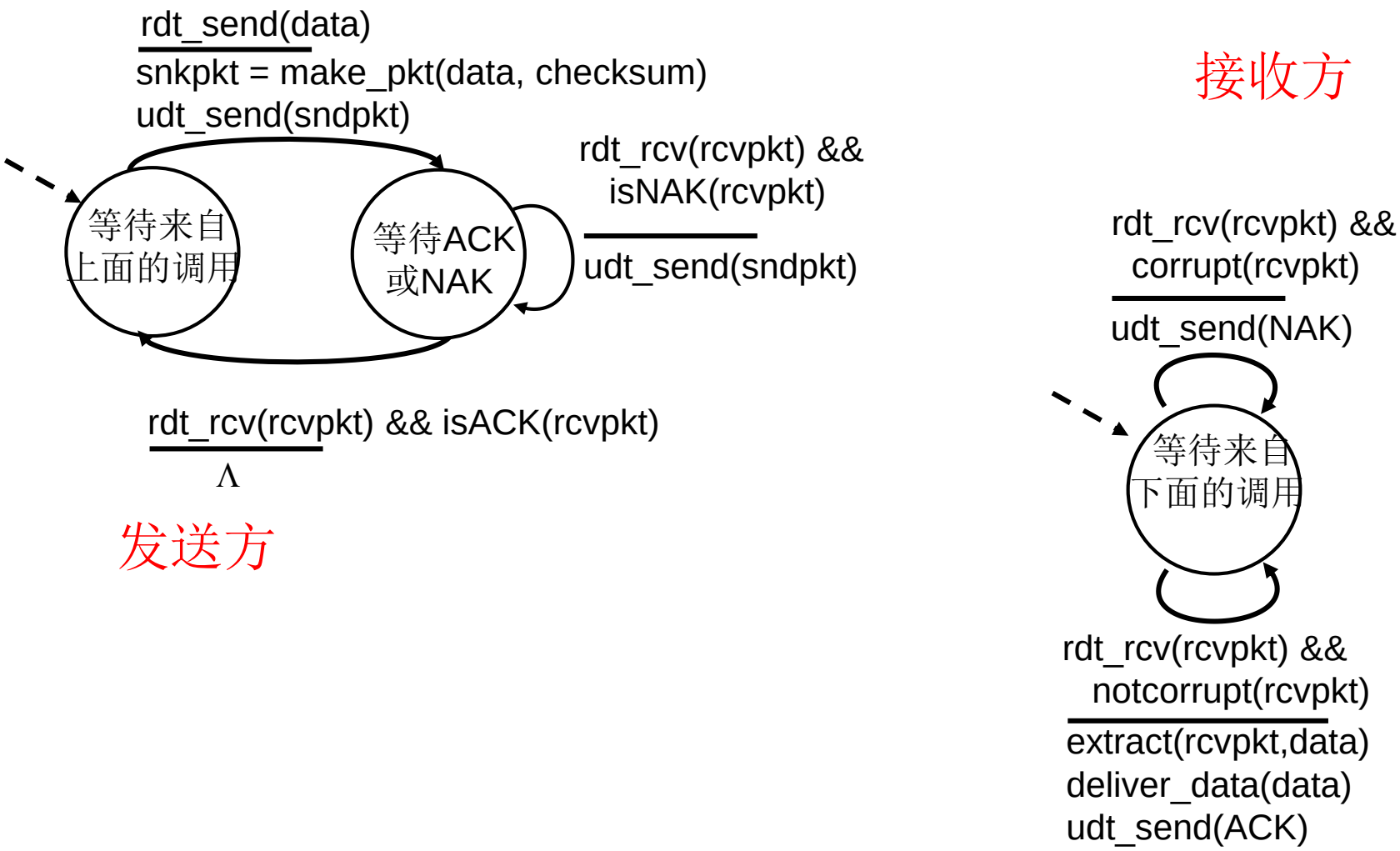
第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

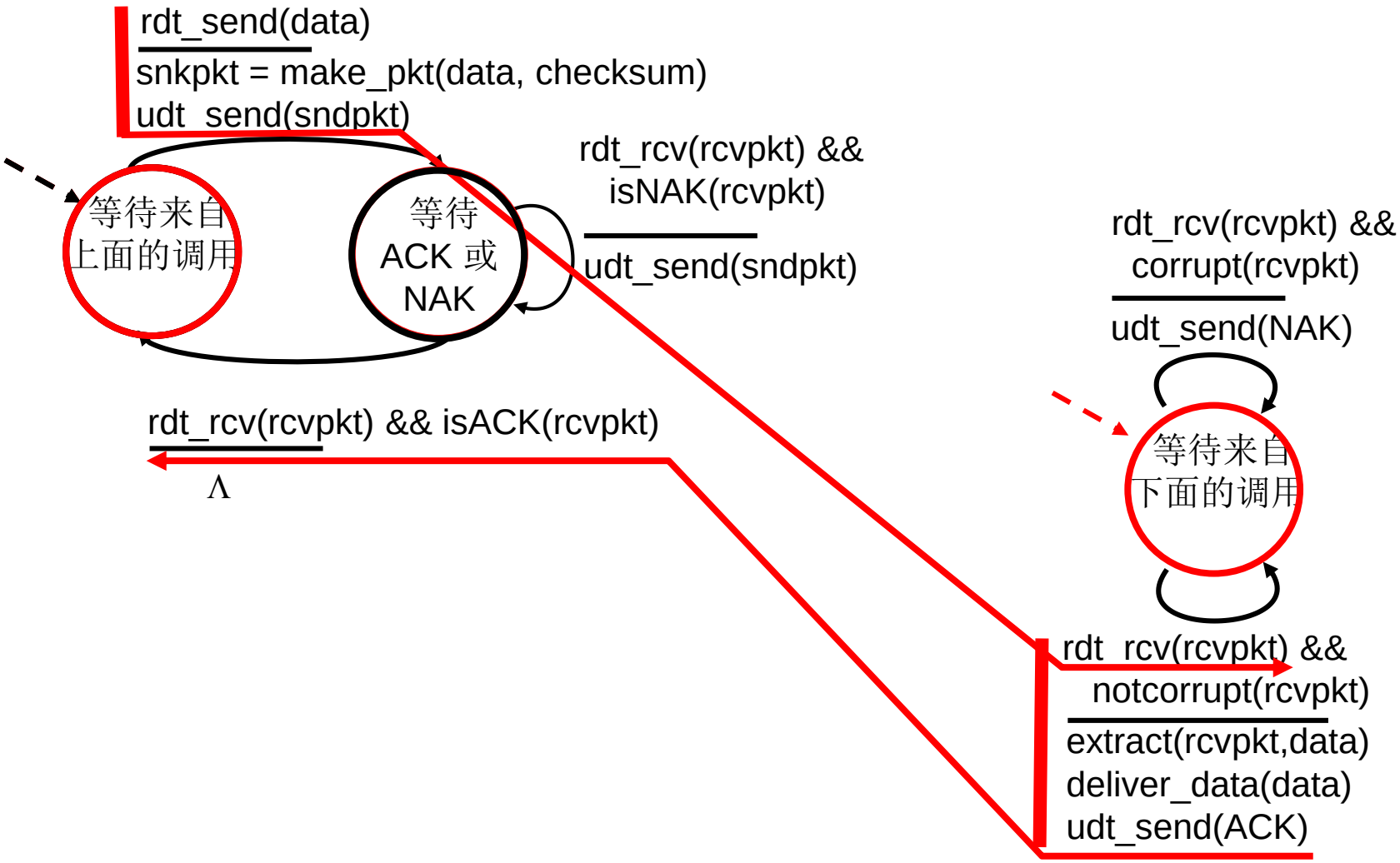
2、Rdt2.0: 具有比特差错的信道

- 下层信道可能让传输分组中的bit受损
  - 校验和将检测到bit错误
- *问题*: 如何从错误中恢复
  - *确认(ACKs):* 接收方明确告诉发送方 分组接收正确
  - *否认 (NAKs):*接收方明确告诉发送方 分组接收出错
  - 发送方收到NAK后重发这个分组
- 在 rdt2.0的新机制 (在 rdt1.0中没有的):
  - 差错检测
  - 接收方反馈: 控制信息 (ACK,NAK)
  - 重传

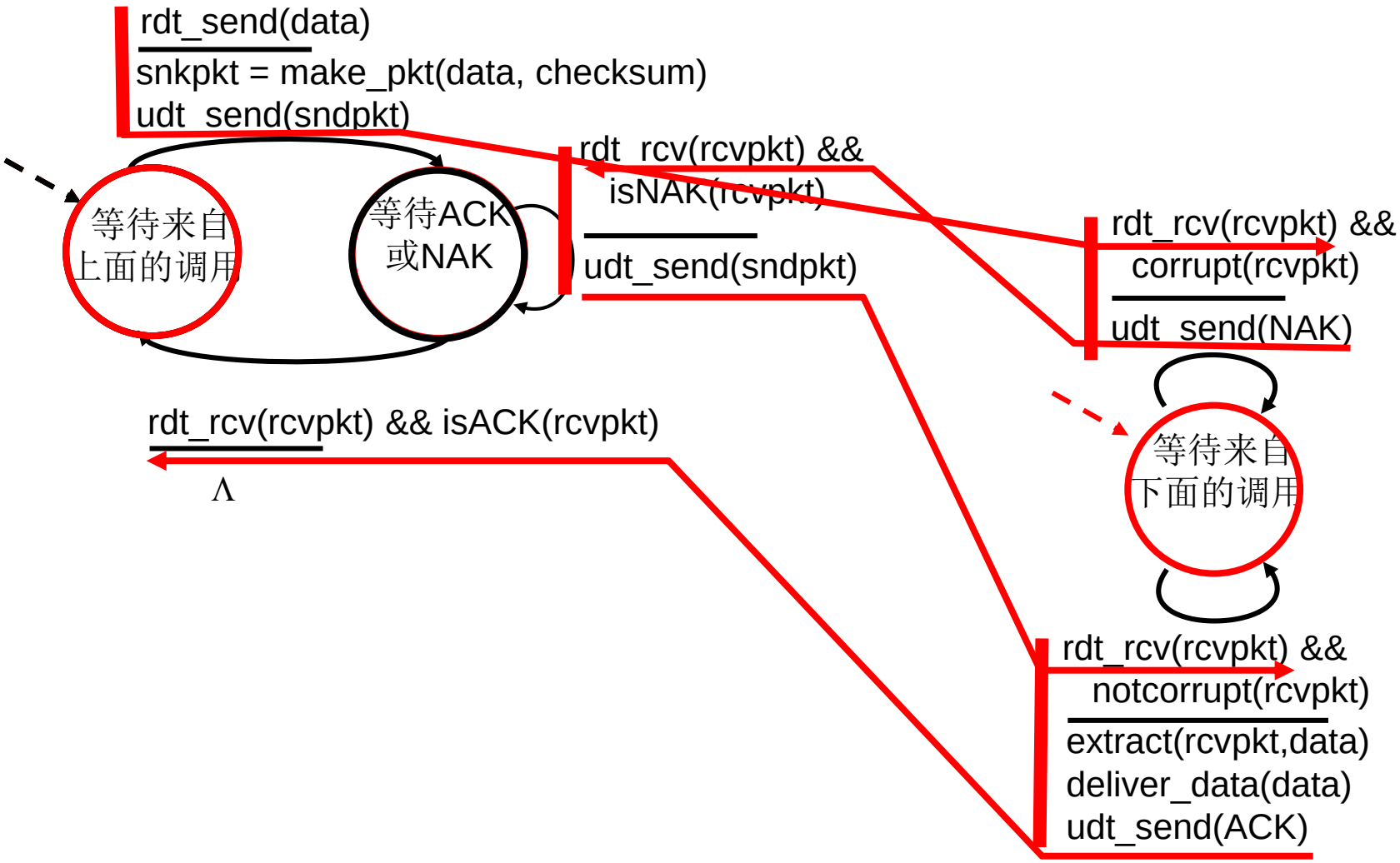
rdt2.0: FSM规格参数



rdt2.0: 无差错时的操作



rdt2.0: 有差错时的情况



停一等协议

发送方发送一个报文，然后

等待接受方的响应

rdt2.0有重大的缺陷!

如果ACK/NAK受损，将会出现何种情况?

- 发送方不知道在接收方会发生什么情况!
- 不能只是重传：可能导致冗余

处理冗余:

- 发送方对每个分组增加*序列号*
- 如果ACK/NAK受损，发送方重传当前的分组
- 接收方丢弃(不再向上交付)冗余分组

停止等待

发送方发送一个分组，然后等待接收方响应

rdt2.1: 讨论

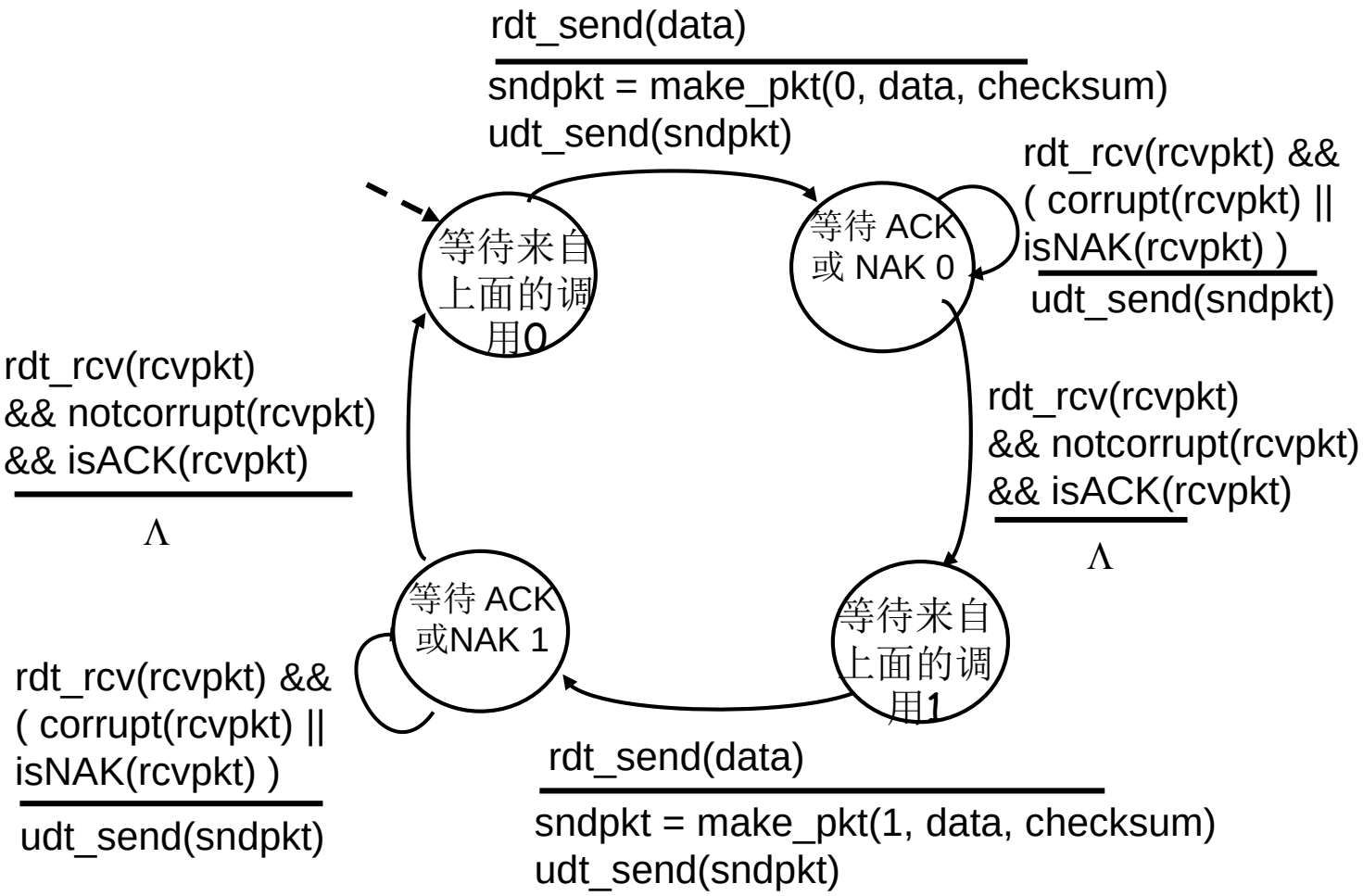
发送方:

- 序号seq # 加入分组中
- 两个序号seq. #’s (0,1) 将够用. (为什么?)
- 必须检查是否收到的ACK/NAK受损
- 状态增加一倍
  - 状态必须“记住”是否“当前的”分组具有0或1序号

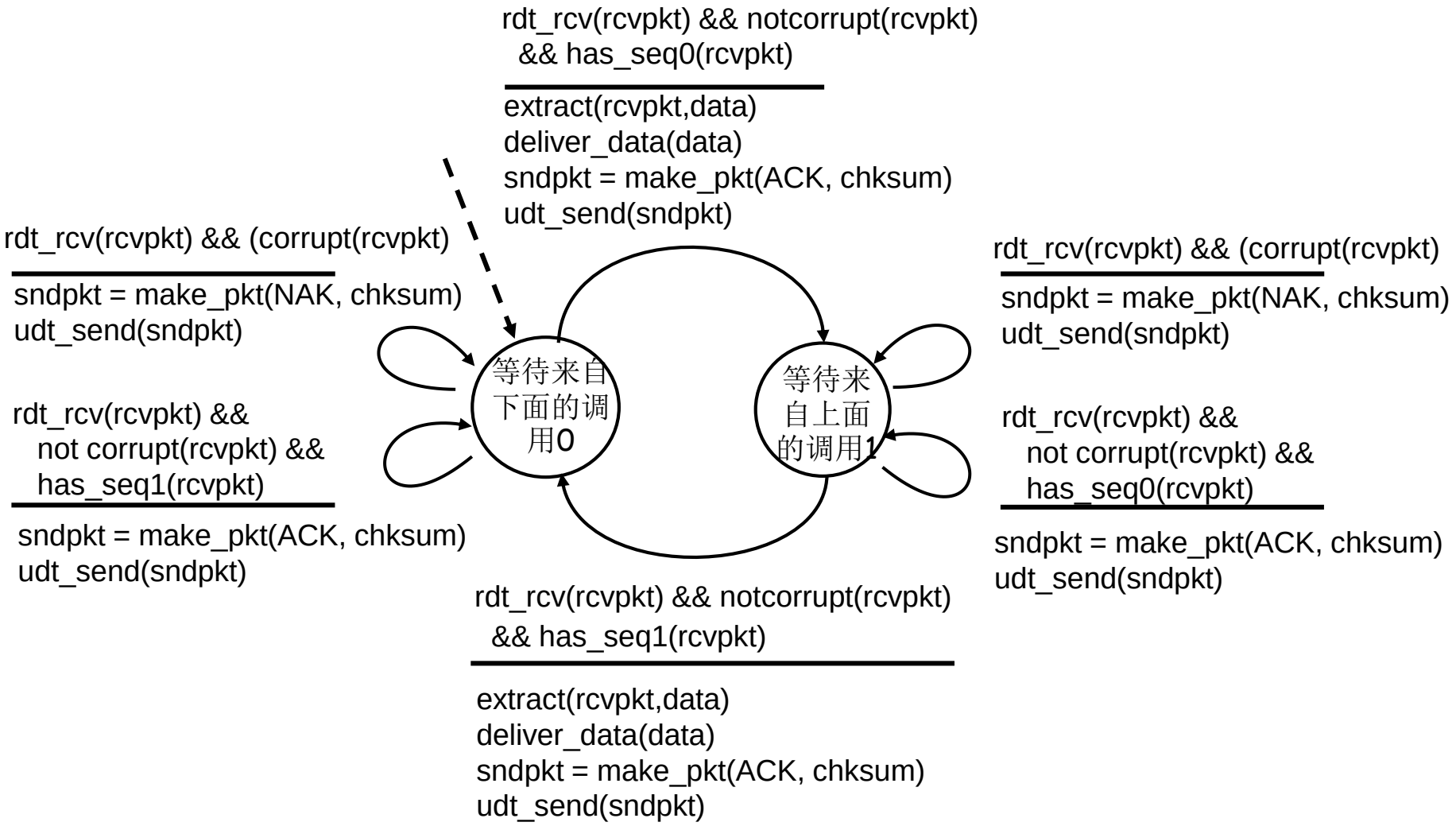
接收方:

- 必须检查是否接收到的分组是冗余的
  - 状态指示是否0或1是所期待的分组序号 seq #
- 注意: 接收方不能知道是否它的最后的ACK/NAK在发送方已经被正确的接收

rdt2.1: 发送方, 处理受损的ACK/NAK



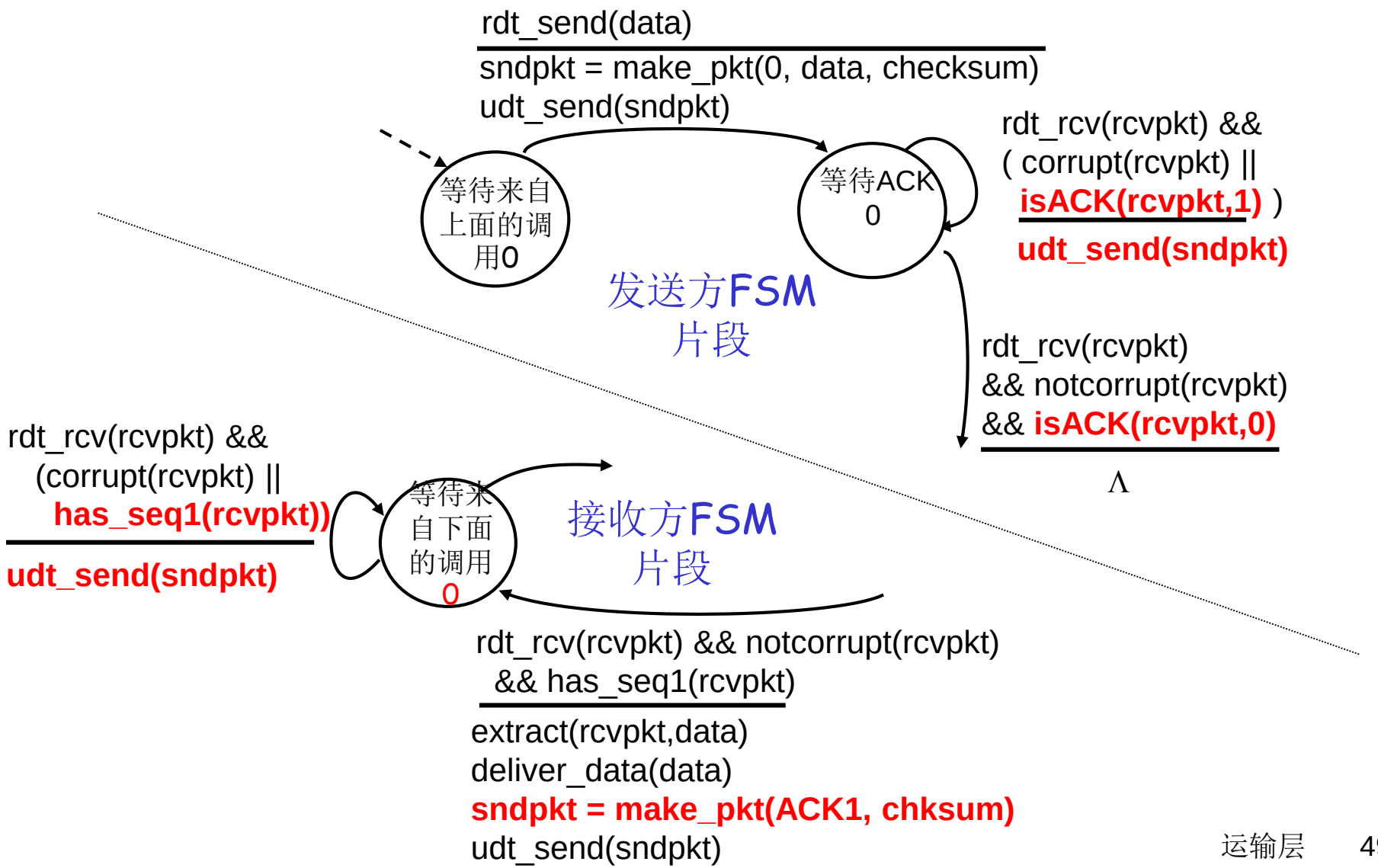
rdt2.1: 接收方,处理受损的ACK/NAK



rdt2.2: 一种无NAK的协议

- 与rdt2.1一样的功能，仅使用ACK
- 代替NAK,接收方对最后正确接收的分组发送ACK
  - 接收方必须明确地包括被确认分组的序号
- 在发送方冗余的ACK导致如同NAK相同的动作：*重传当前分组*

rdt2.2: 发送方, 接收方片段



第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

3、rdt3.0: 具有差错和丢包的信道

新假设: 下面的信道也能丢失分组(数据或ACK)

导致2个问题:

1、怎样检测丢包?

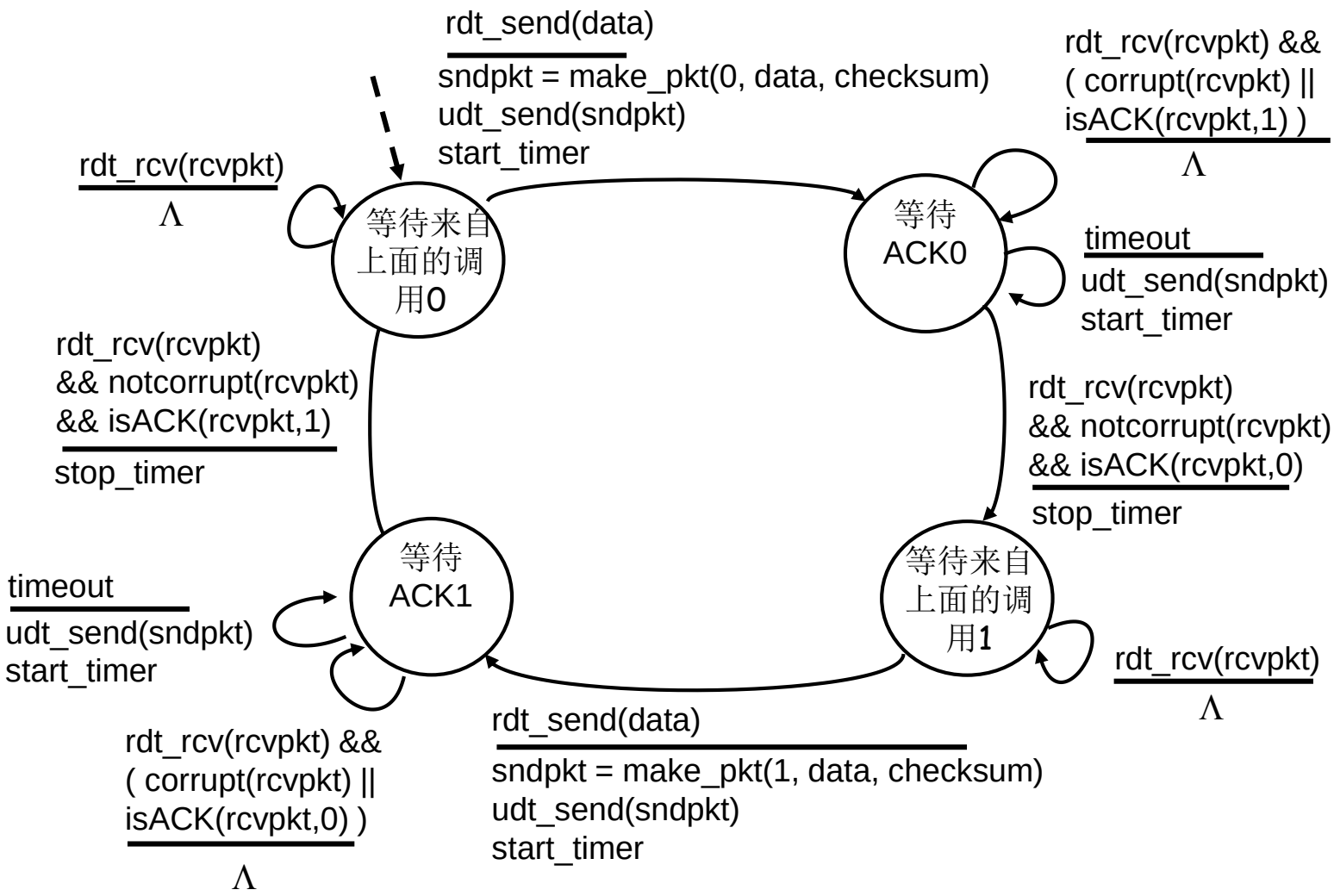
2、丢包后该做些什么?

序号、ACK分组、重传能给出后一个问题的答案。

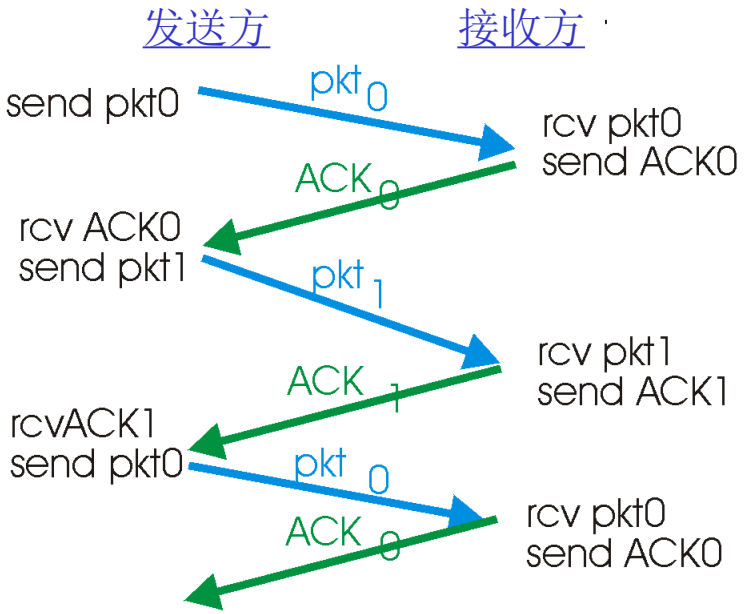
方法: 发送方等待ACK一段“合理的”时间

- 如在这段时间没有收到ACK则重传
- 如果分组(或ACK)只是延迟(没有丢失):
  - 重传将是冗余的，但序号的使用已经处理了该情况
  - 接收方必须定义被确认的分组序号
- 需要倒计时定时器

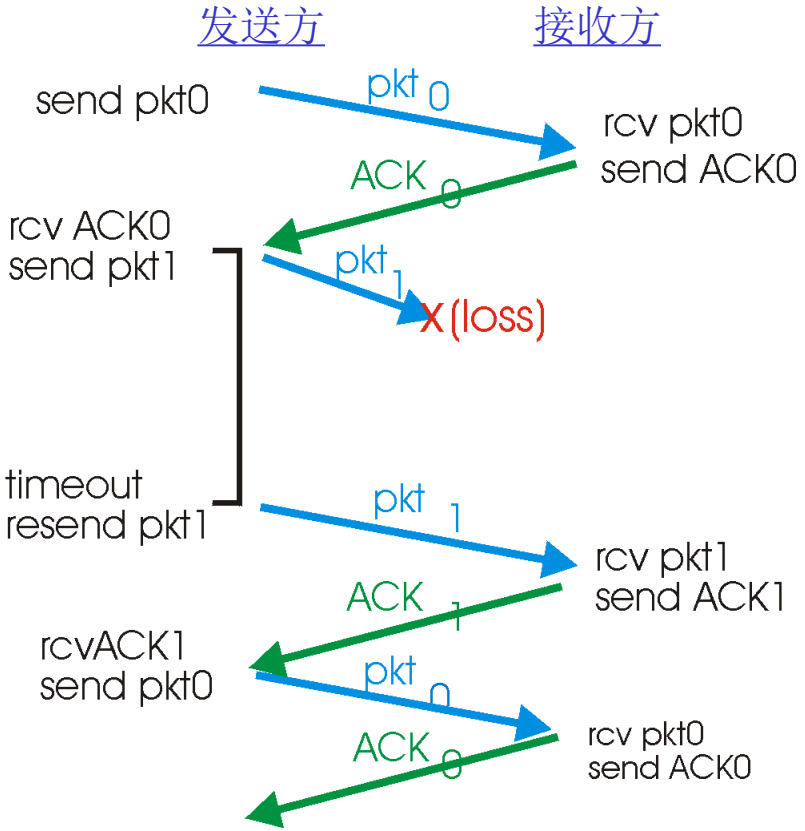
rdt3.0发送方


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
等待来自上面的调用0
等待ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
stop_timer
等待ACK1
等待来自上面的调用1
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
Λ
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,0))
Λ
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

rdt3.0 运行情况

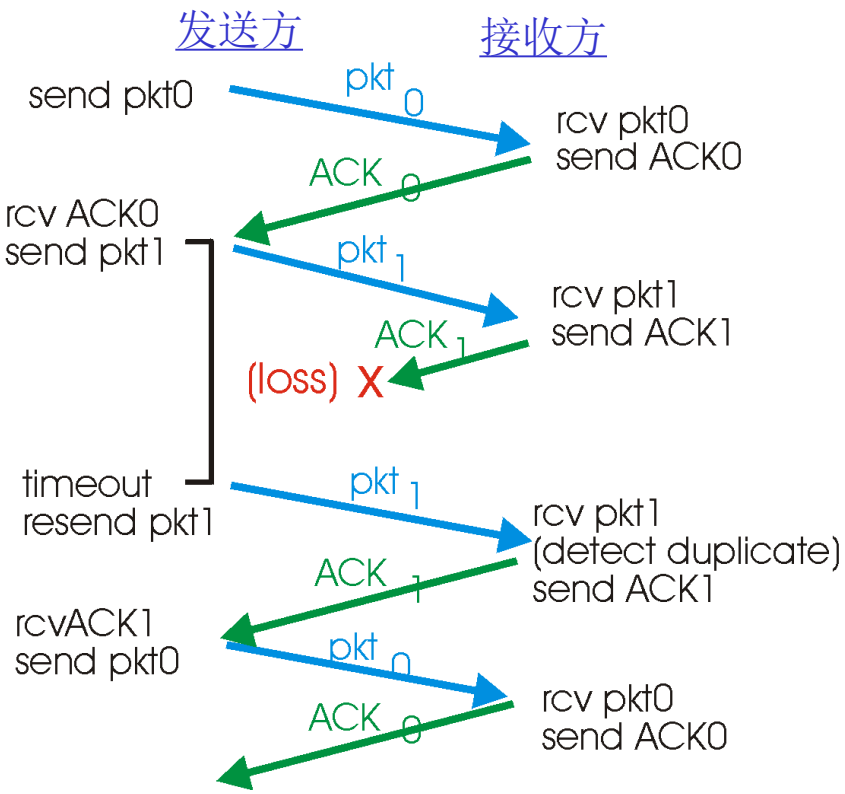


(a) 无丢包时的运行

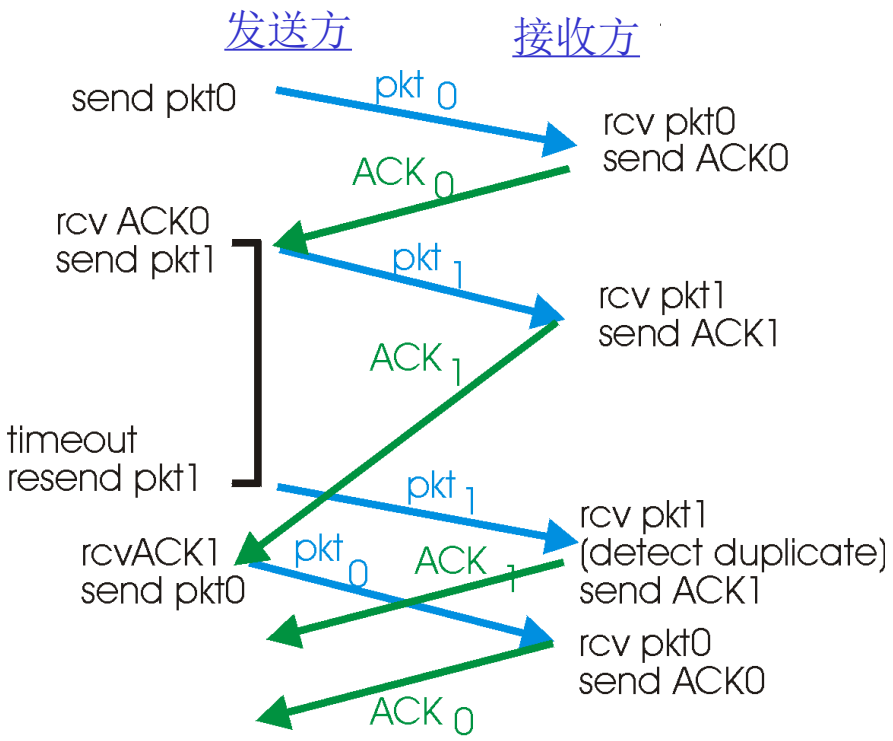


(b) 分组丢失

rdt3.0运行情况



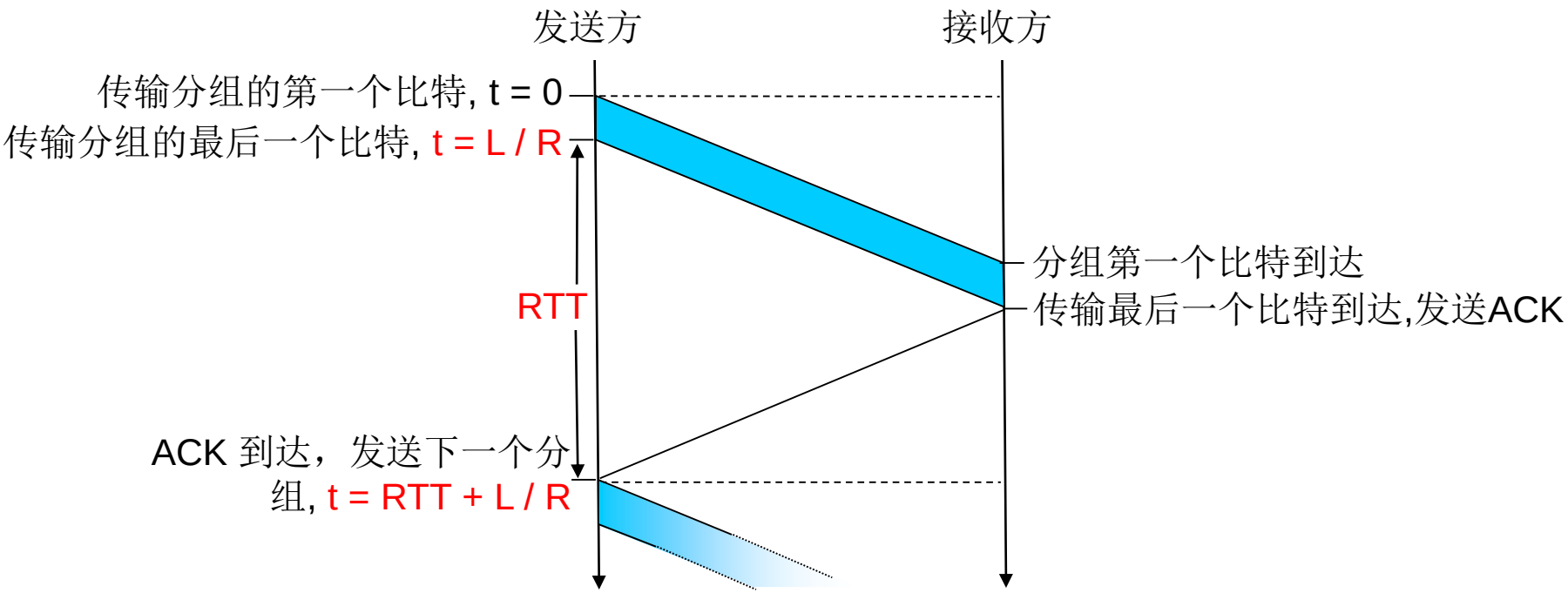
(c) ACK丢失



(d) 过早超时

rdt3.0的性能

- rdt3.0能够工作，但性能不太好
- 例子: 1 Gbps链路, 15 ms端到端传播时延, 1KB分组:



rdt3.0的性能

例子: 1 Gbps链路, 15 ms端到端传播时延, 1KB分组:

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8kb/pkt}{10**9\ b/sec} = 8\ us$$

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

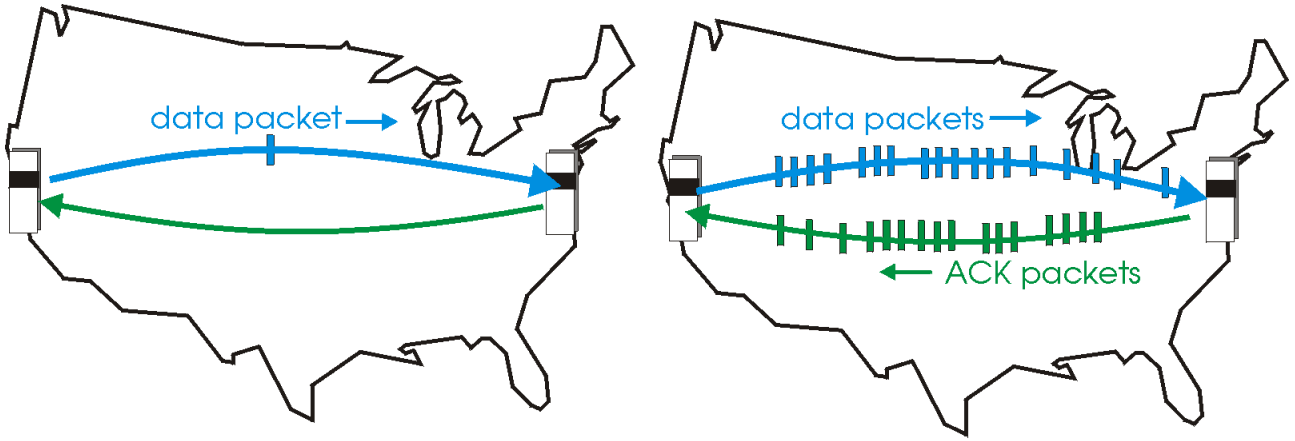
- U_{sender}: 利用率 – 发送方实际用于将发送bit送进信道的时间与发送时间的比率
- 每30.008 ms发送 1KB 分组 -> 经1 Gbps 链路有267kps 吞吐量
- 网络协议限制了物理资源的使用!

第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

3.4.2 流水线协议

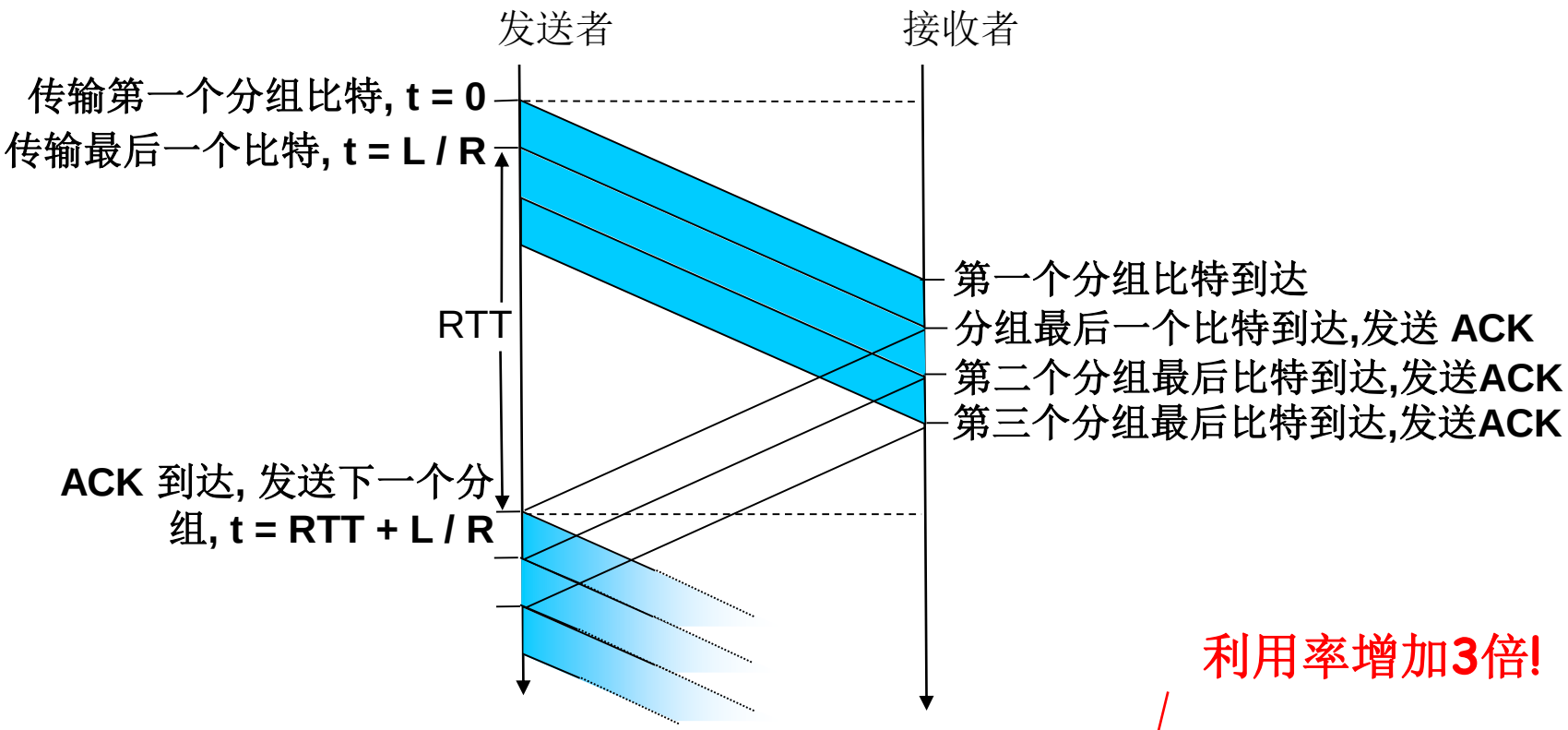
流水线：发送方允许发送多个、“传输中的”,还没有应答的报文段



(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

流水线协议: 增加利用率

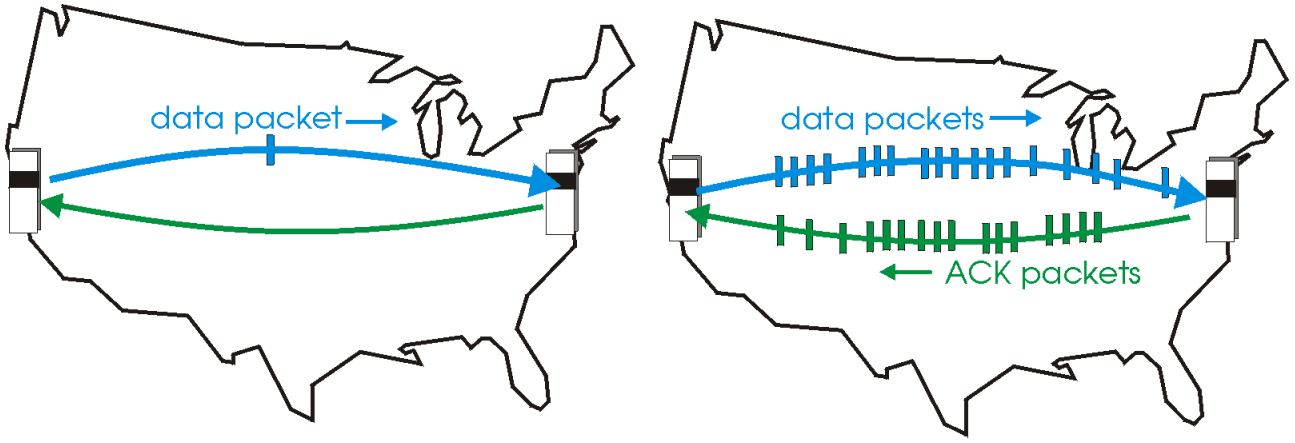


利用率增加3倍!

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

3.4.2 流水线协议

- 流水线技术对可靠数据传输协议带来如下影响:
  - 序号的范围必须增加
  - 发送方和/或接收方设有缓冲
- 流水线协议的两种形式:
  回退N帧法（go-Back-N），选择性重传（S-R），

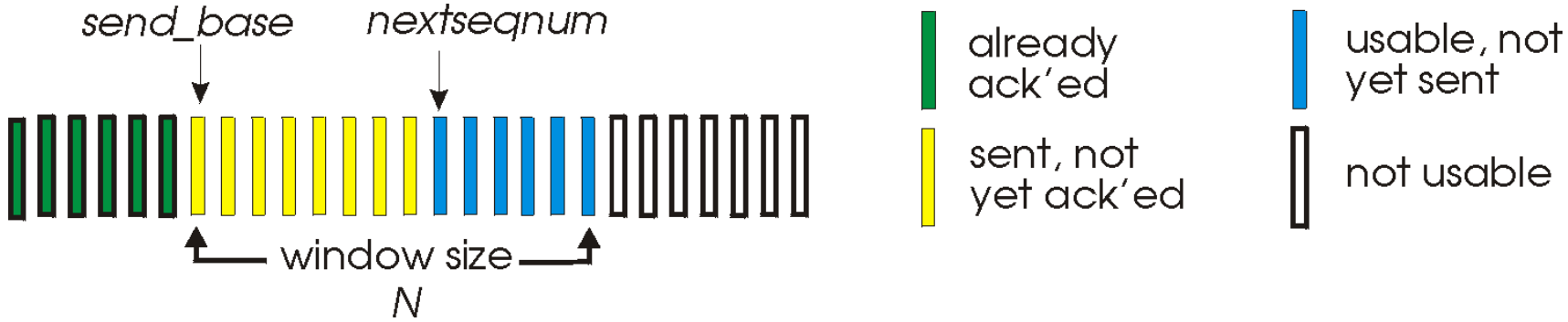


(a) a stop-and-wait protocol in operation　　(b) a pipelined protocol in operation

1、Go-Back-N

发送方:

- 已发送但未被确认的分组数最大为N, 即允许N个连续的没有应答分组



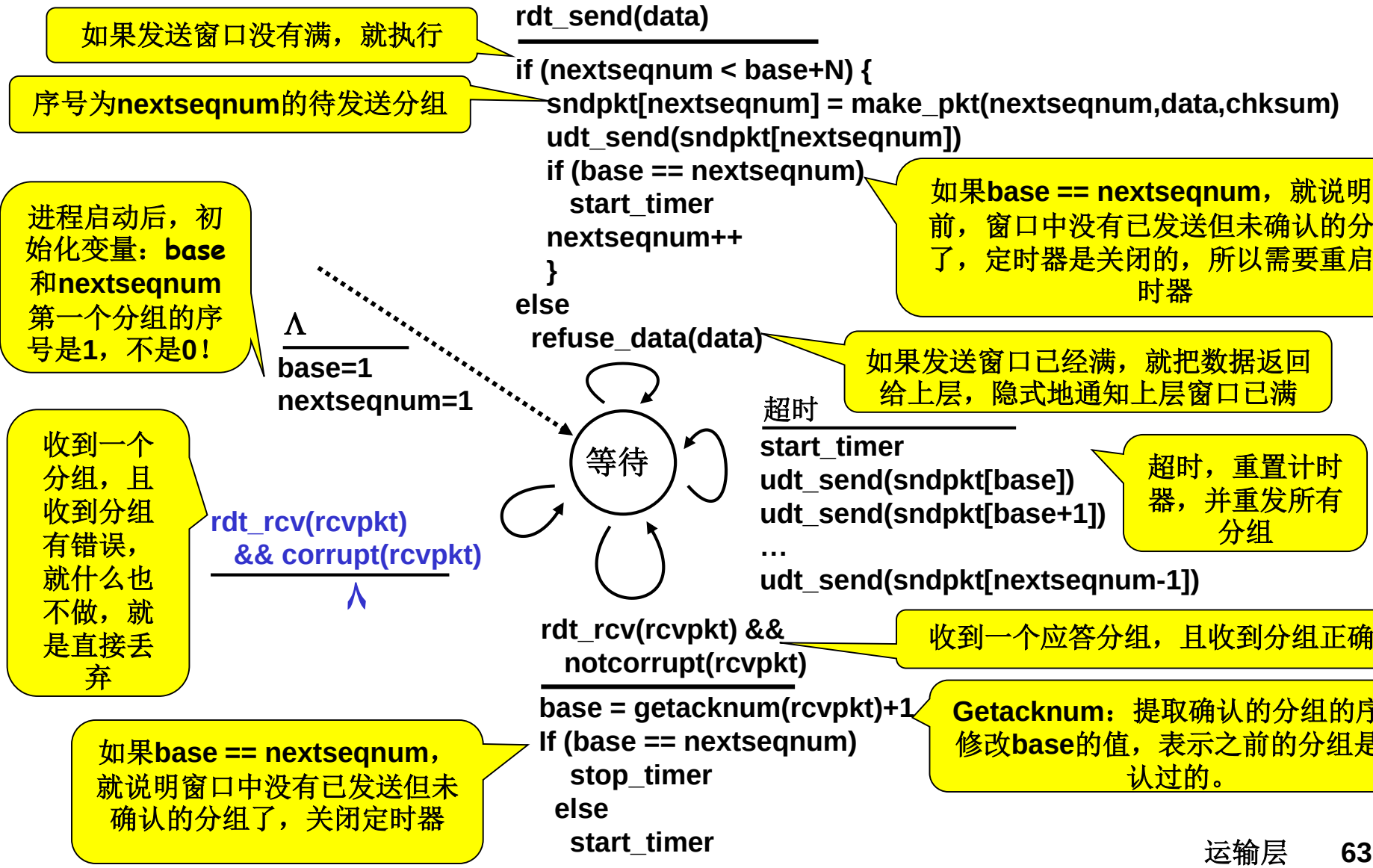
- 滑动窗口协议
  - N：窗口长度
- 分组的序号承载在分组首部的一个固定长度的字段中。
  - 序号空间使用模2^K运算。

1、Go-Back-N

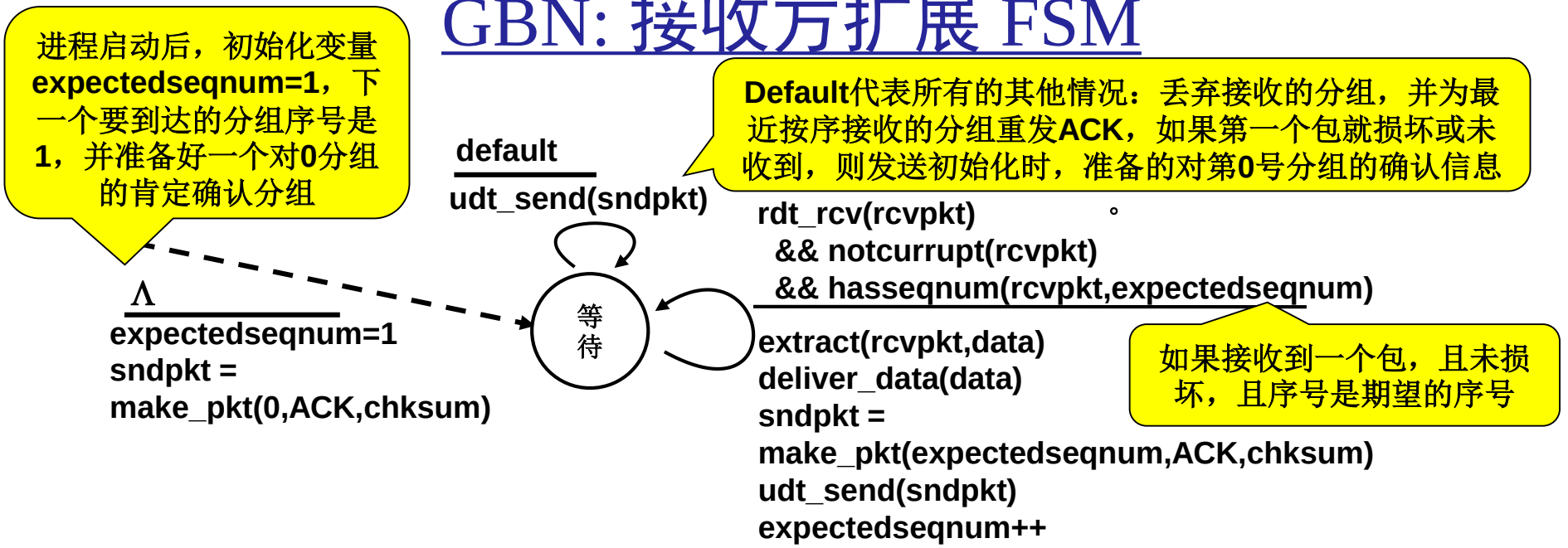
发送方:

- ACK(n): 确认所有的（包括序号n）的分组 - “累计ACK”
  - 可能收到重复的ACKs (见接收方)
- 对每个传输中的分组的用同一个计时器
  - 对第一个发送未被确认的报文定时
- *timeout(n):若*超时，重传窗口中的分组n及所有更高序号的分组

GBN: 发送方扩展的 FSM

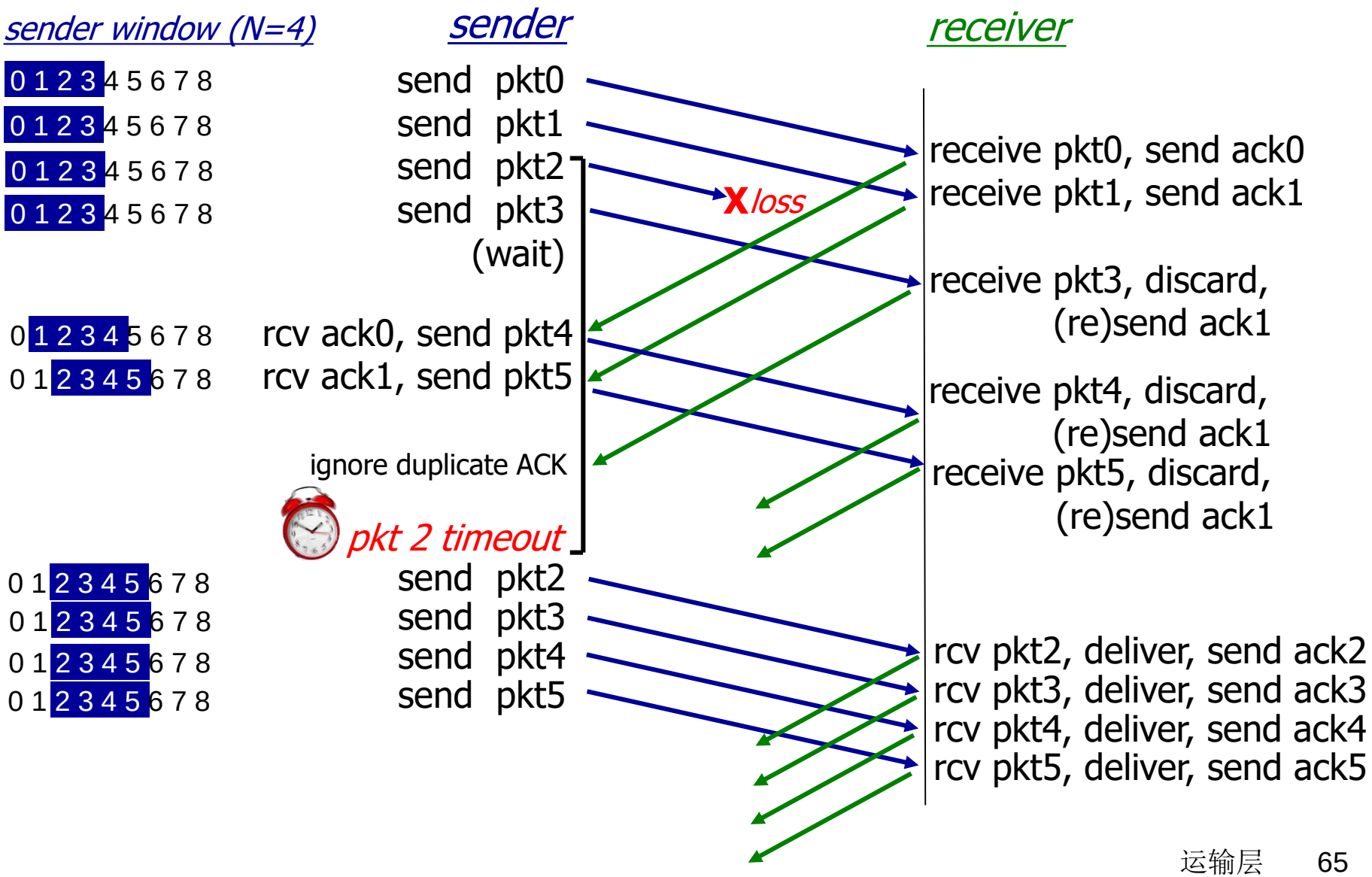


GBN: 接收方扩展 FSM



- 只有ACK: 对发送正确接收的分组总是发送具有最高按序序号的ACK
  - 可能产生冗余的ACKs
  - 仅仅需要记住期望的序号值（expectedseqnum）
- 对失序的分组:
  - 丢弃 (不缓存) -> 没有接收缓冲区!
  - 重新确认具有按序的分组

GBN 操作

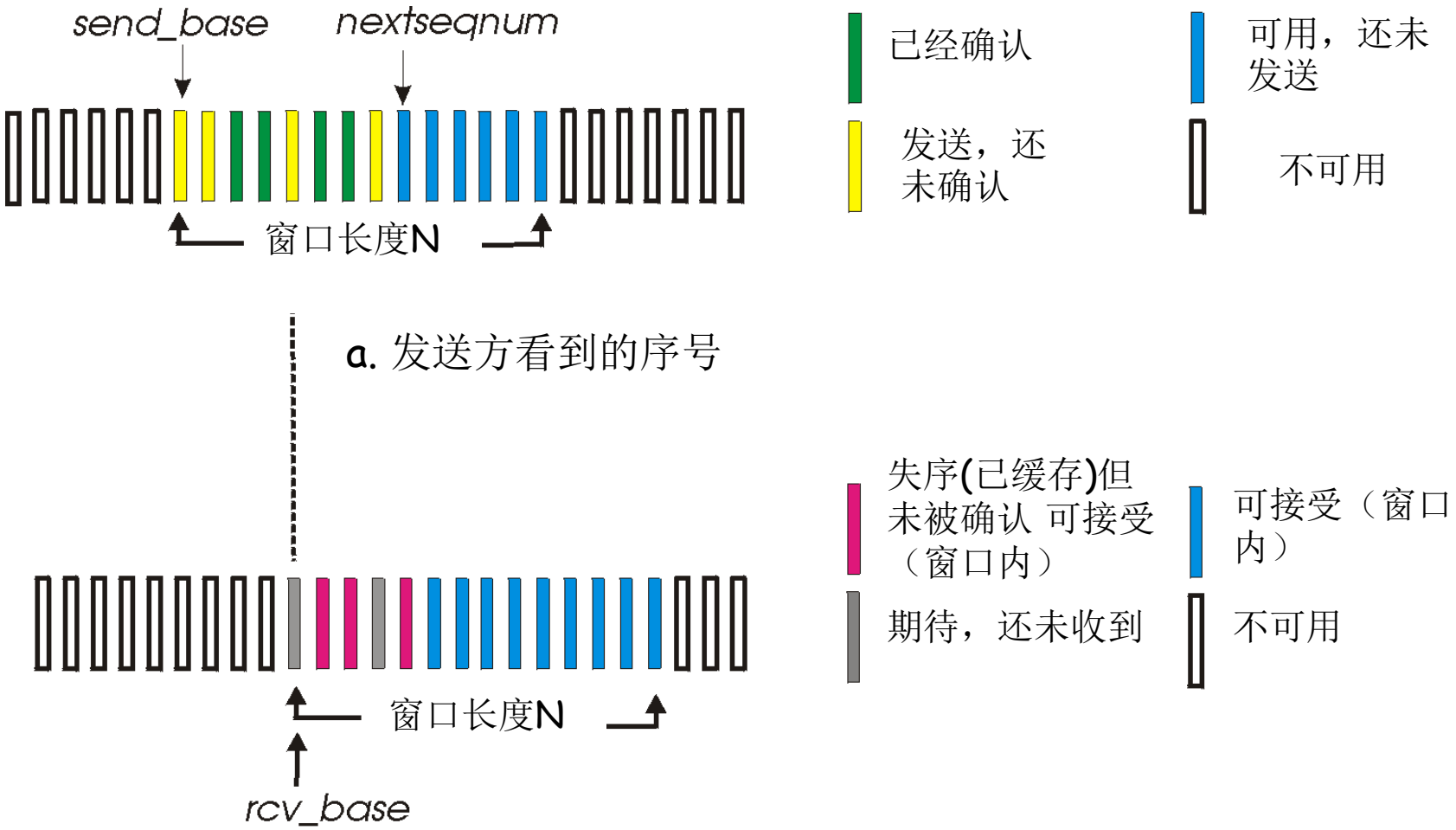

sender window (N=4)
sender
receiver
0 1 2 3 4 5 6 7 8
send pkt0
send pkt1
send pkt2
send pkt3
(wait)
X loss
receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, discard,
(re)send ack1
rcv ack0, send pkt4
rcv ack1, send pkt5
receive pkt4, discard,
(re)send ack1
receive pkt5, discard,
(re)send ack1
ignore duplicate ACK
pkt 2 timeout
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

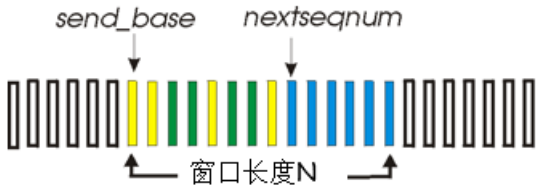
2、选择性重传（Selective Repeat）

GBN改善了信道效率，但仍然有不必要重传问题

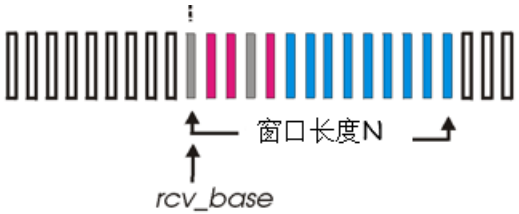
- 接收方分别确认所有正确接收的报文段
  - 需要缓存分组，以便最后按序交付给给上层
- 发送方只需要重传没有收到ACK的分组
  - 发送方定时器对每个没有确认的分组计时
- 发送窗口
  - N个连续的序号
  - 也需要限制已发送但尚未应答分组的序号

选择性重传: 发送方, 接收方窗口





选择性重传



发送方

上层传来数据：

- 如果窗口中下一个序号可用，发送分组

timeout(n):

- 重传分组n, 重启其计时器

ACK(n) 在 [sendbase,sendbase+N]:

- 标记分组 n 已经收到
- 如果n 是最小未收到应答的分组，向前滑动窗口base指针到下一个未确认序号

接收方

分组n在 [rcvbase, rcvbase+N-1]

- 发送 ACK(n)
- 失序: 缓存
- 按序: 交付 (也交付所有缓存的按序分组),向前滑动窗口到下一个未收到分组的序号

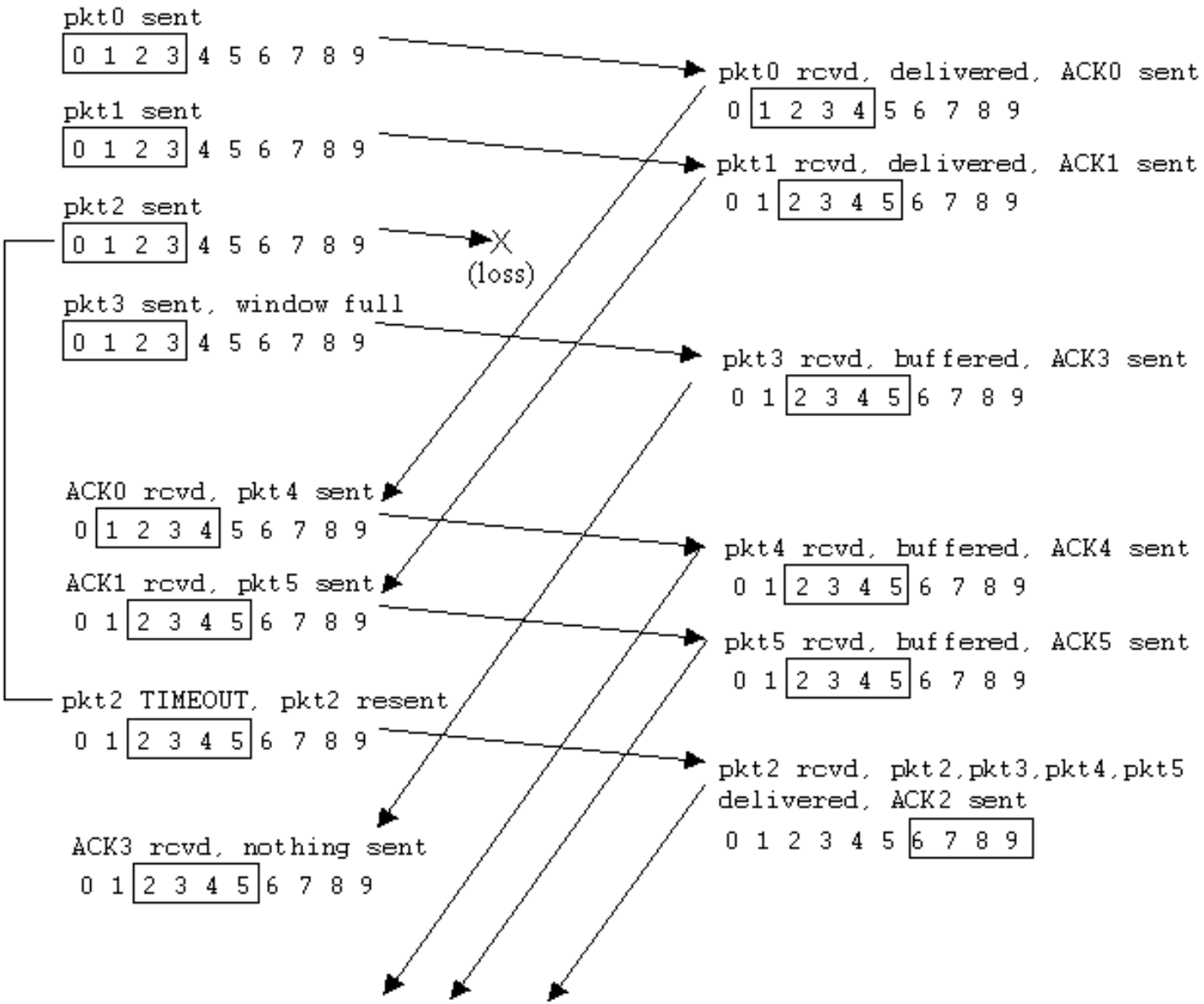
分组n在[rcvbase-N,rcvbase-1]

- ACK(n)

其他:

- 忽略

选择重传的操作


pkt0 sent
0 1 2 3 4 5 6 7 8 9
pkt1 sent
0 1 2 3 4 5 6 7 8 9
pkt2 sent
0 1 2 3 4 5 6 7 8 9
X
(loss)
pkt3 sent, window full
0 1 2 3 4 5 6 7 8 9
pkt0 rcvd, delivered, ACK0 sent
0 1 2 3 4 5 6 7 8 9
pkt1 rcvd, delivered, ACK1 sent
0 1 2 3 4 5 6 7 8 9
pkt3 rcvd, buffered, ACK3 sent
0 1 2 3 4 5 6 7 8 9
ACK0 rcvd, pkt4 sent
0 1 2 3 4 5 6 7 8 9
ACK1 rcvd, pkt5 sent
0 1 2 3 4 5 6 7 8 9
pkt4 rcvd, buffered, ACK4 sent
0 1 2 3 4 5 6 7 8 9
pkt5 rcvd, buffered, ACK5 sent
0 1 2 3 4 5 6 7 8 9
pkt2 TIMEOUT, pkt2 resent
0 1 2 3 4 5 6 7 8 9
pkt2 rcvd, pkt2,pkt3,pkt4,pkt5
delivered, ACK2 sent
0 1 2 3 4 5 6 7 8 9
ACK3 rcvd, nothing sent
0 1 2 3 4 5 6 7 8 9

选择重传: 困难的问题

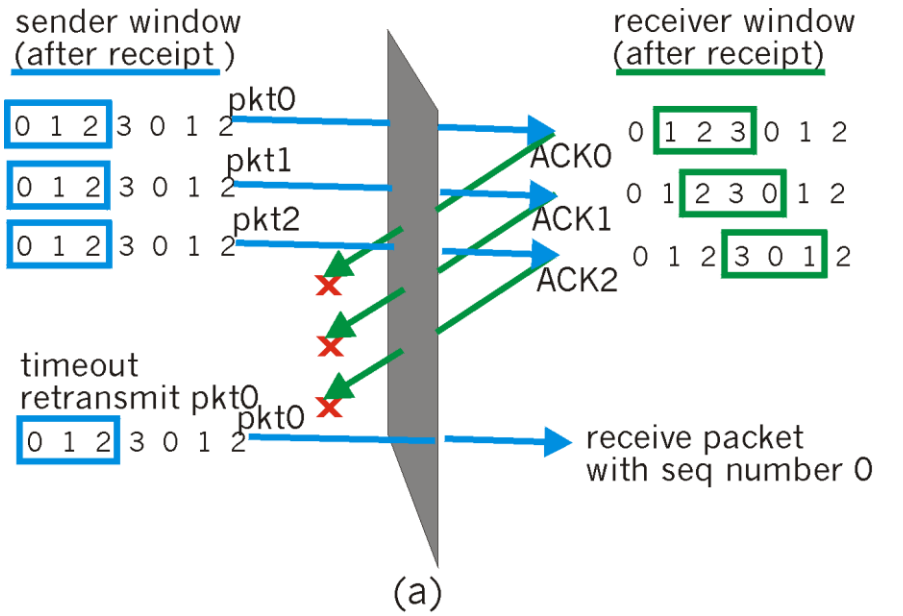
例子:

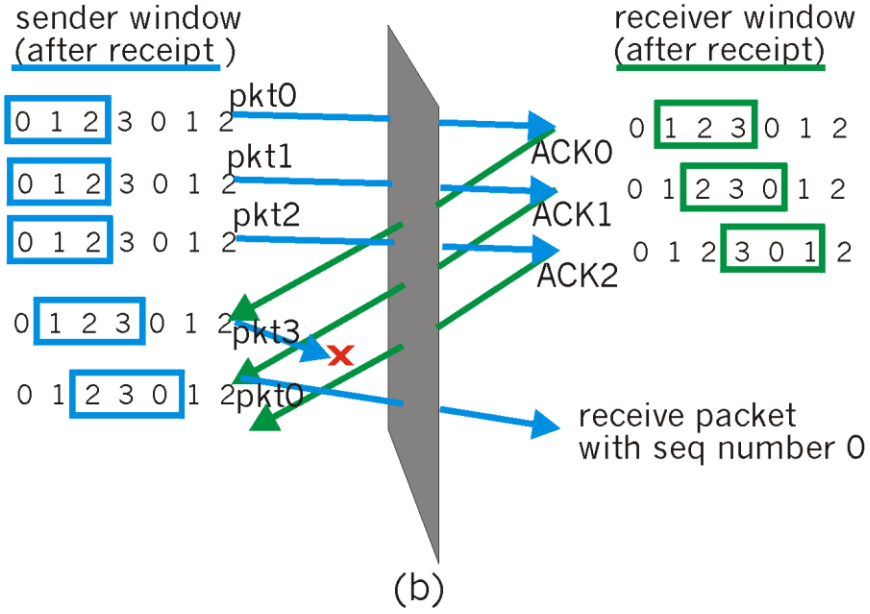
- 序号: 0, 1, 2, 3
- 窗口长度 = 3

- 接收方：在(a)和(b)两种情况下接收方没有发现差别!
- 在 (a) (b)两种情况下，接收方的窗口位置都一样，而且都收到了序号为0的分组，没有任何差别。

问题: 序号长度与窗口长度有什么关系?

回答: 窗口长度小于等于序号空间的一半





可靠数据传输机制及用途总结

机制	用途和说明
检验和	用于检测在一个传输分组中的比特错误。
定时器	用于检测超时/重传一个分组，可能因为该分组（或其ACK）在信道中丢失了。由于当一个分组被时延但未丢失（过早超时），或当一个分组已被接收方收到但从接收方到发送方的ACK丢失时，可能产生超时事件，所以接收方可能会收到一个分组的多个冗余拷贝。
序号	用于为从发送方流向接收方的数据分组按顺序编号。所接收分组的序号间的空隙可使该接收方检测出丢失的分组。具有相同序号的分组可使接收方检测出一个分组的冗余拷贝。
确认	接收方用于告诉发送方一个分组或一组分组已被正确地接收到了。确认报文通常携带着被确认的分组或多个分组的序号。确认可以是逐个的或累积的，这取决于协议。
否定确认	接收方用于告诉发送方某个分组未被正确地接收。否定确认报文通常携带着未被正确接收的分组的序号。
窗口、流水线	发送方也许被限制仅发送那些序号落在一个指定范围内的分组。通过允许一次发送多个分组但未被确认，发送方的利用率可在停等操作模式的基础上得到增加。我们很快将会看到，窗口长度可根据接收方接收和缓存报文的能力或网络中的拥塞程度，或两者情况来进行设置。

第3章 要点

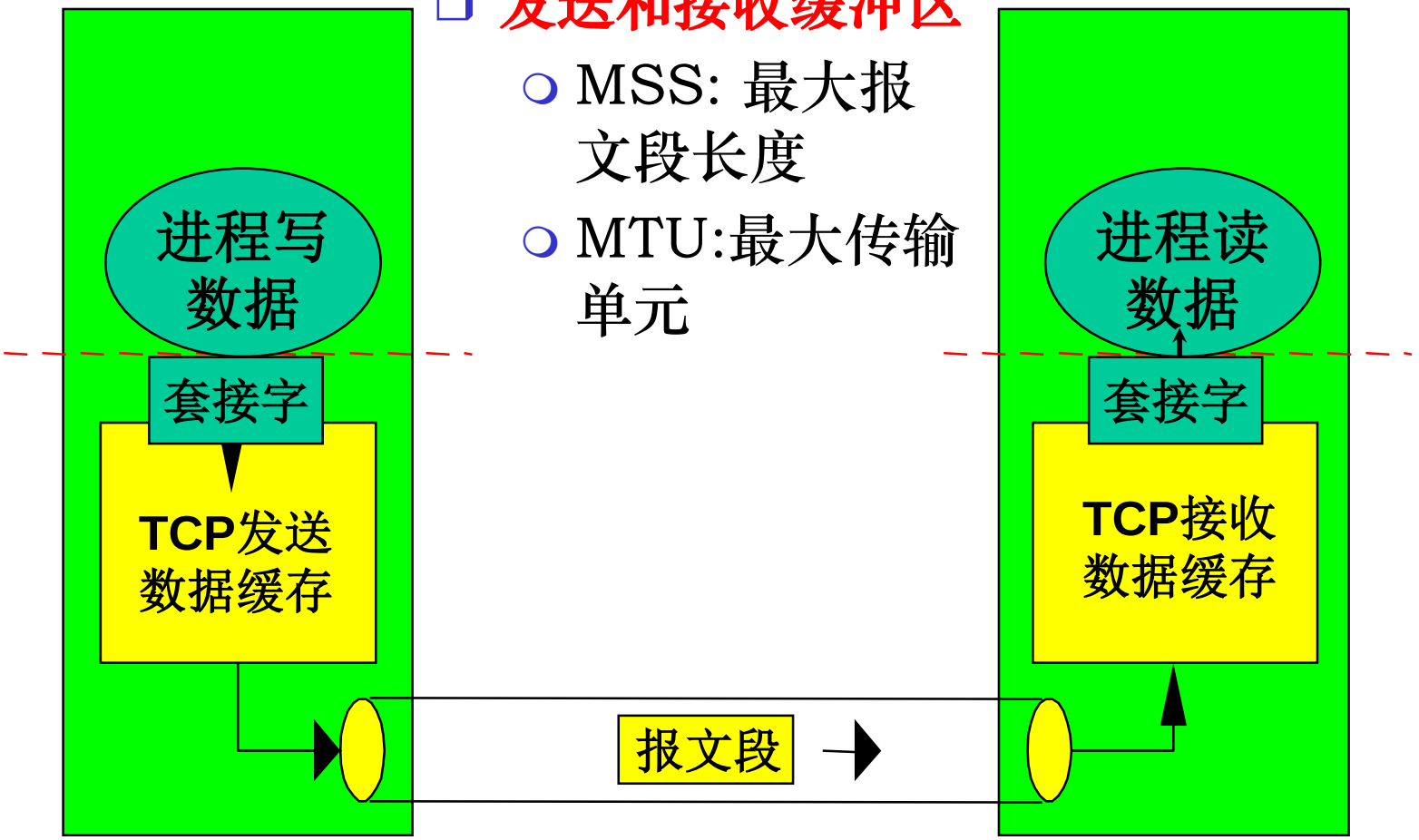
- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

3.5.1 TCP概述 RFCs: 793, 1122, 1323, 2018, 2581

- **点到点:**
  - 一个发送方，一个接收方
  - 连接状态与端系统有关，不为路由器所知
- **面向连接:**
  - 在进行数据交换前，初始化发送方与接收方状态，进行握手(交换控制信息)，
- **全双工数据:**
  - 同一连接上的双向数据流

3.5.1 TCP概述

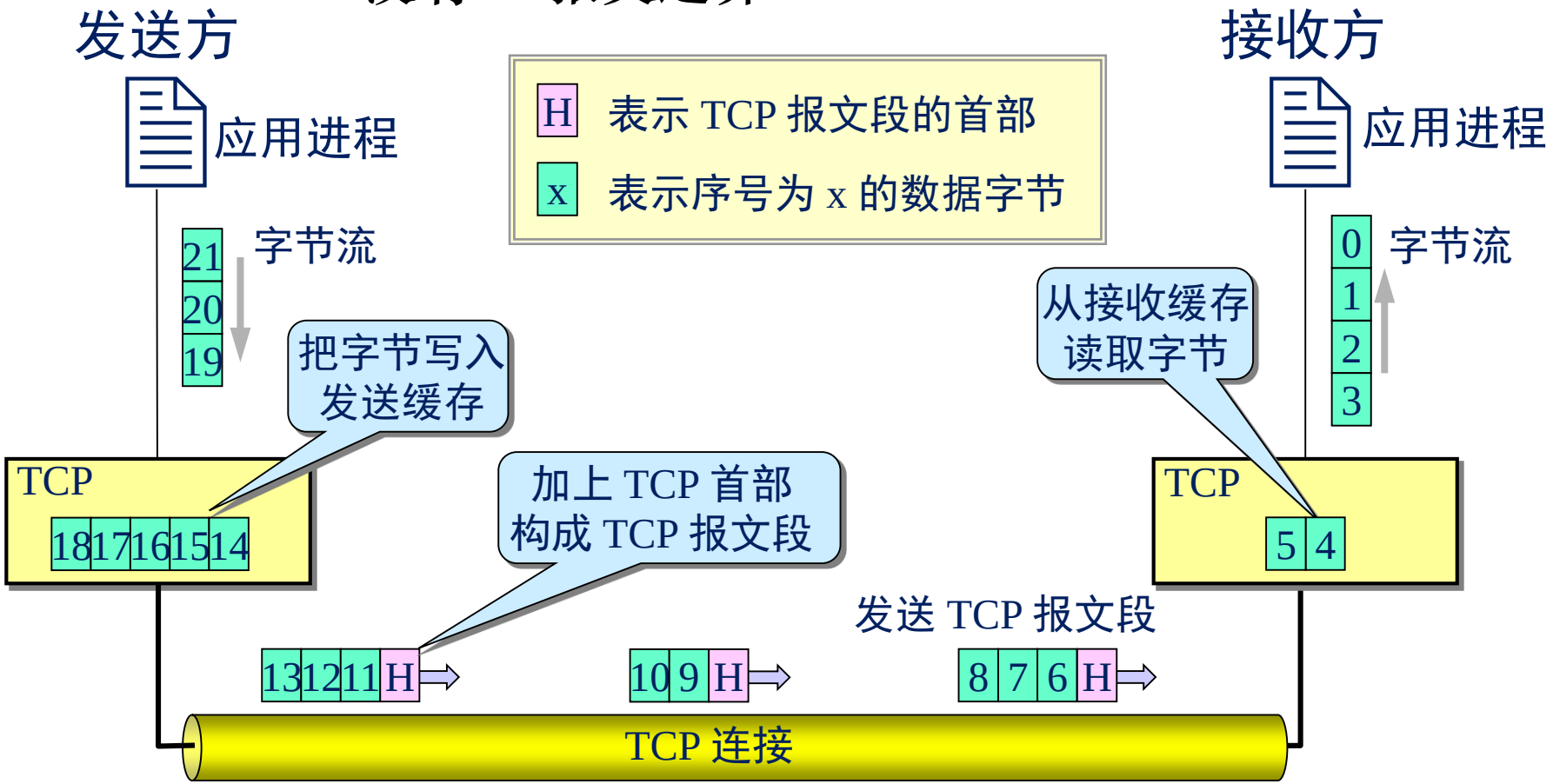
- 发送和接收缓冲区
  - MSS: 最大报文段长度
  - MTU:最大传输单元



收发缓冲区

TCP 面向流的概念

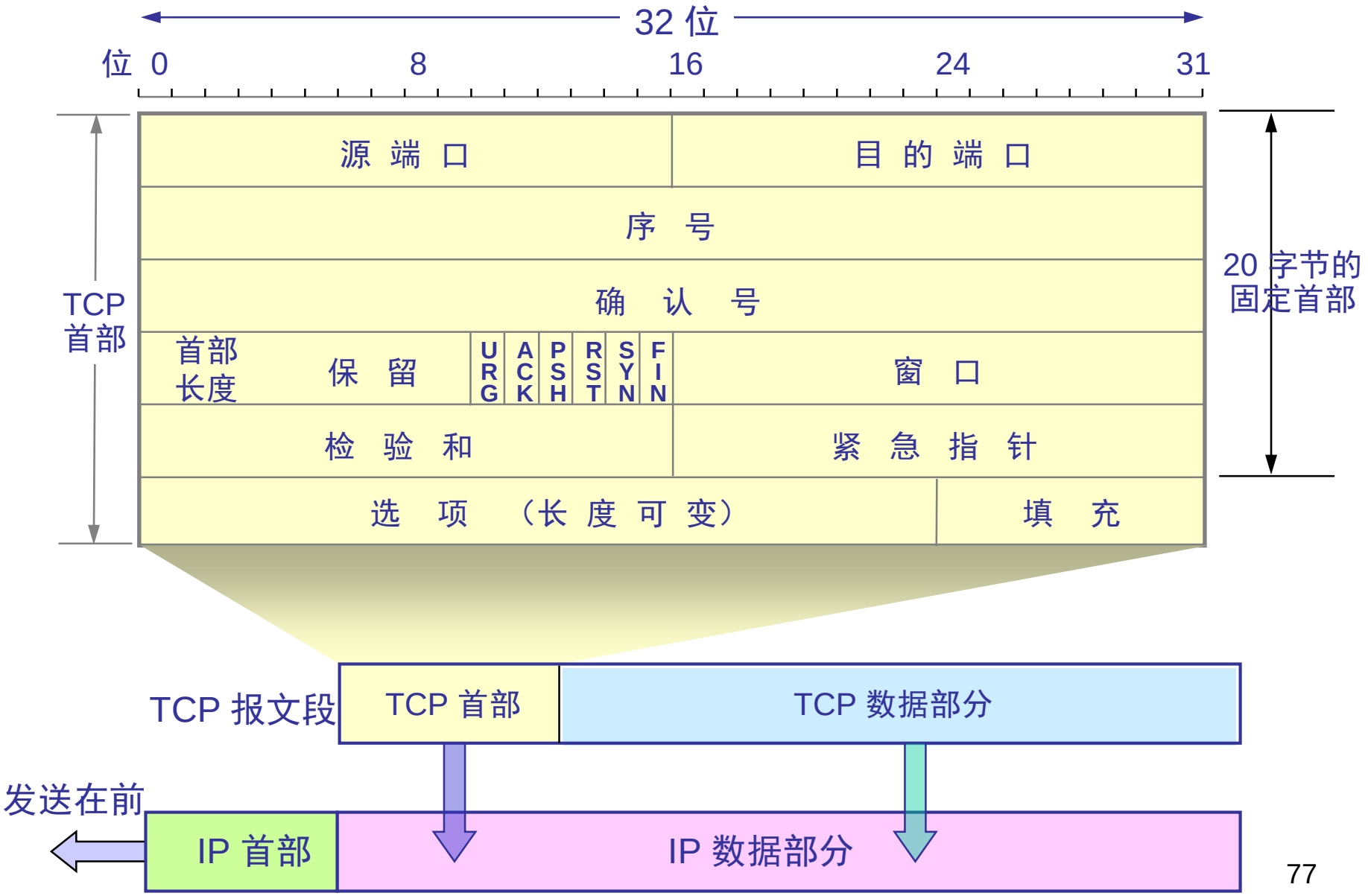
- **可靠、有序的字节流：**
  - 没有 “报文边界”



3.5.1 TCP概述

- **流水线:**
  - TCP拥塞和流量控制设置滑动窗口协议
- **流量控制:**
  - 发送方不能淹没接收方
- **拥塞控制:**
  - 抑止发送方速率来防止过分占用网络资源

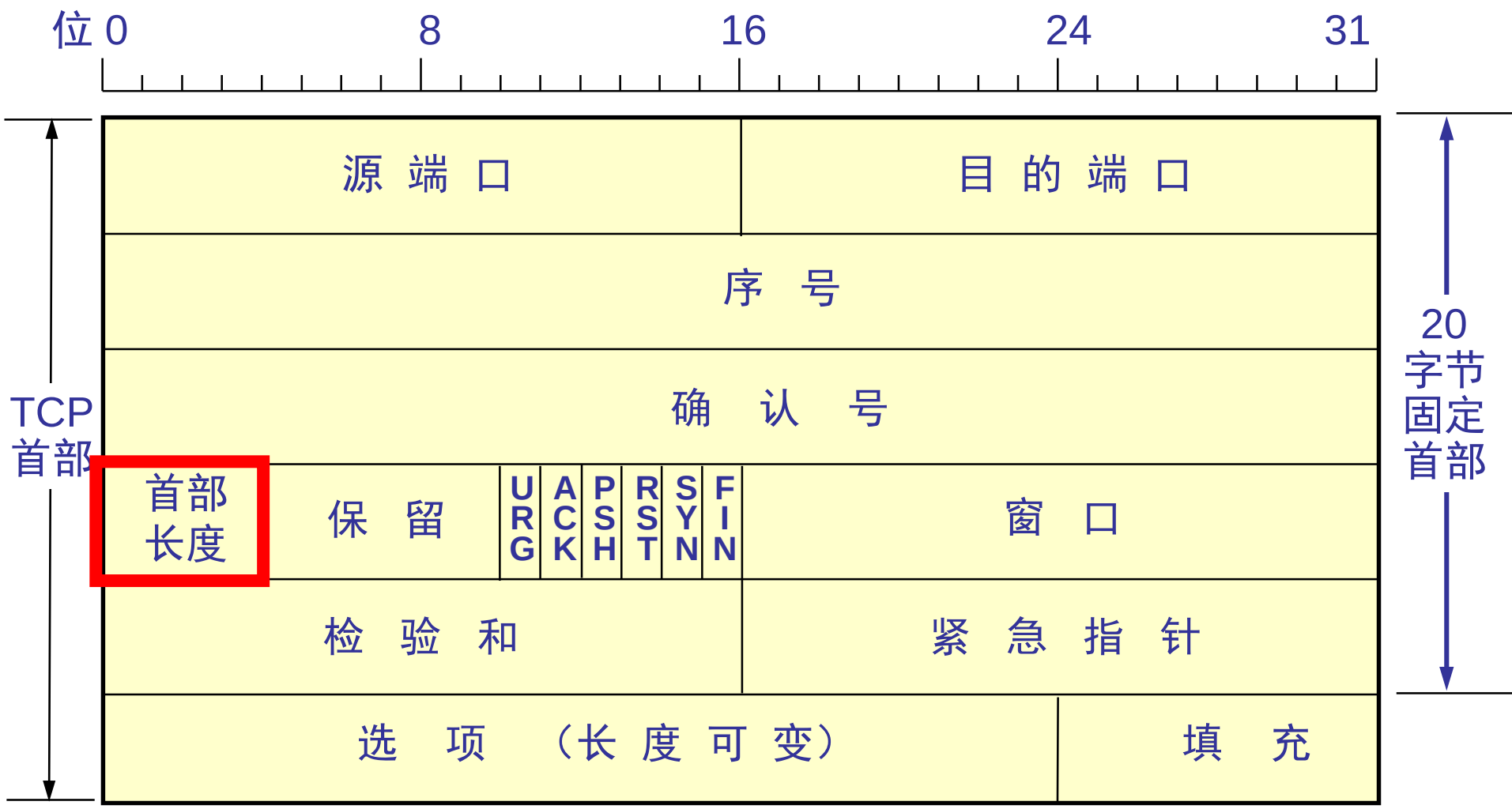
3.5.2 TCP 报文段结构



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

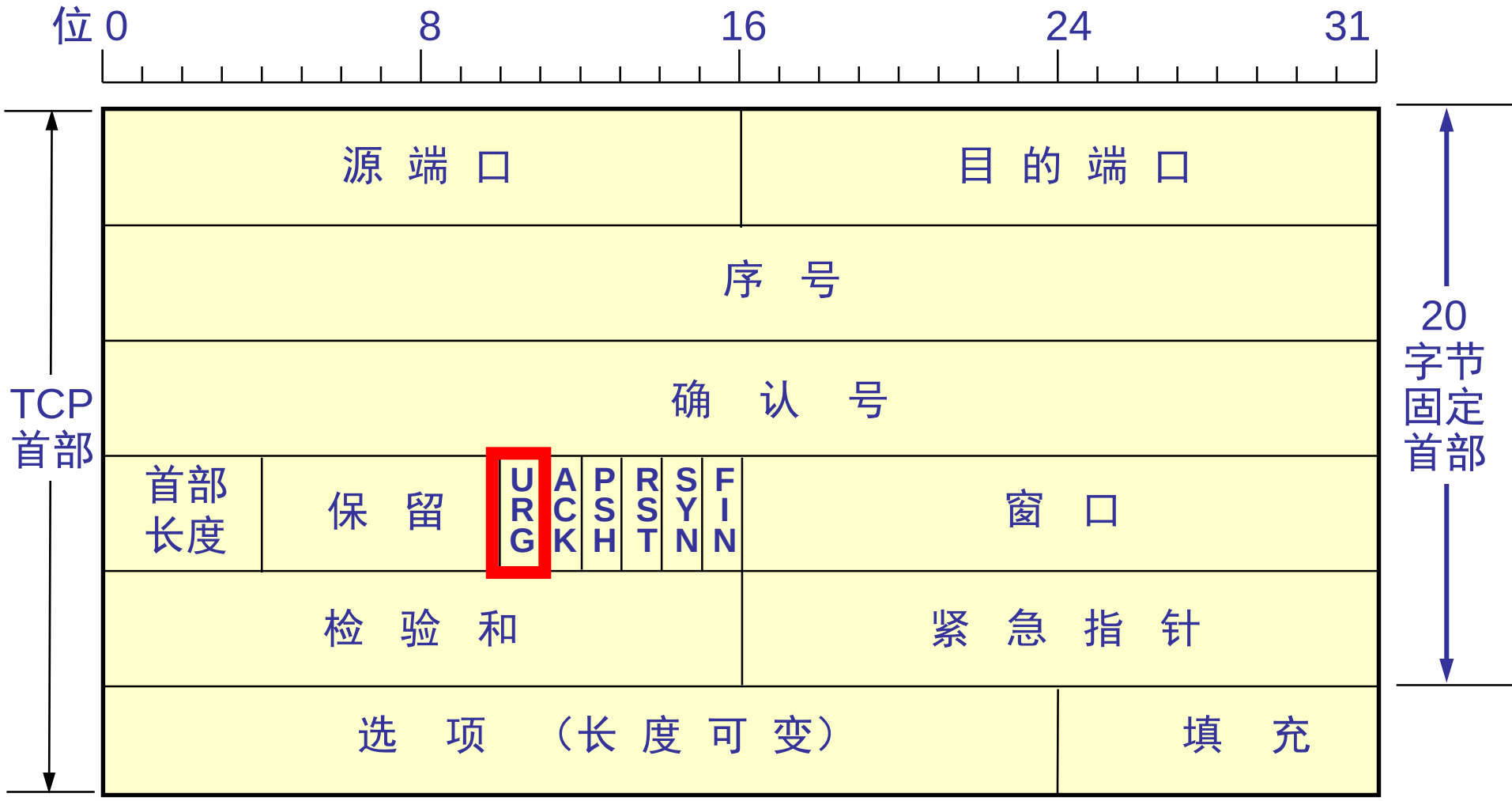
序号字段——占 4 字节。TCP 连接中传送的数据流中的每一个字节都编上一个序号。序号字段的值则指的是本报文段所发送的数据的第一个字节的序号。

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

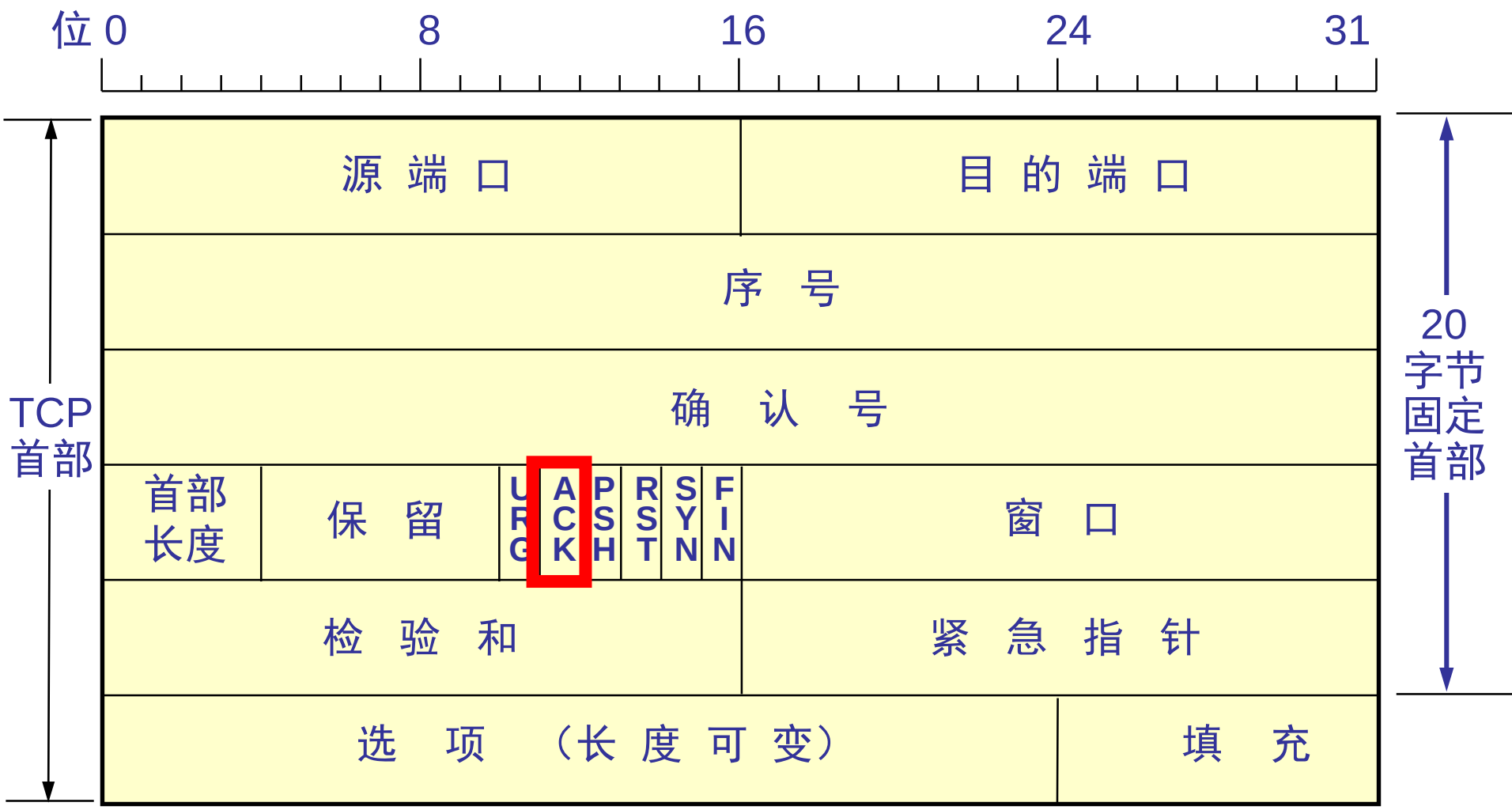


首部长度（即数据偏移）——占 4 位，指示了以32比特的字为单位的TCP首部长度。也就是说，它指出 TCP 报文段的数据起始处距离 TCP 报文段的起始处有多远。“数据偏移”的单位是 32 位字（以 4 字节为计算单位）。

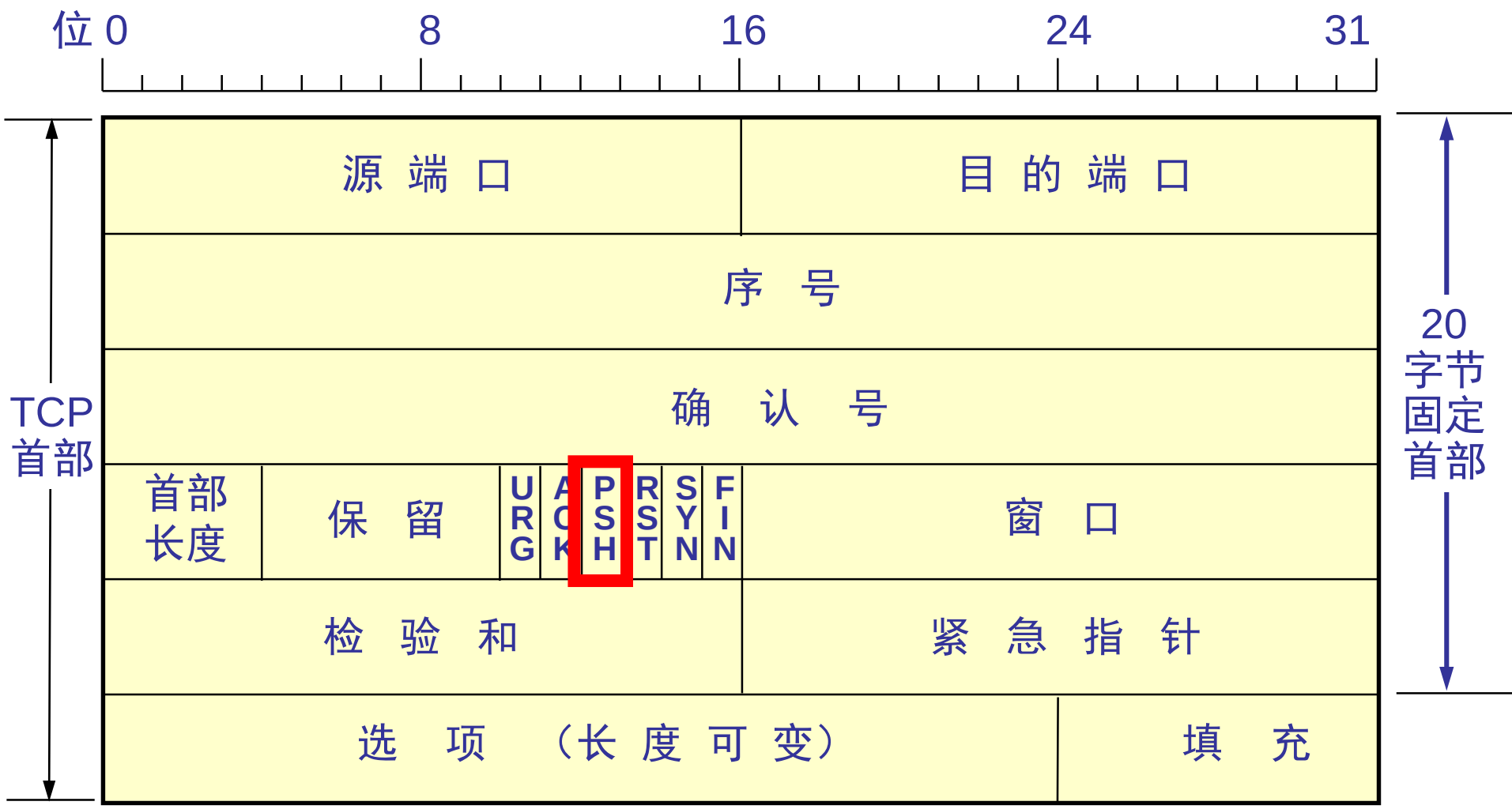
保留字段——占 6 位，保留为今后使用，但目前应置为 0。



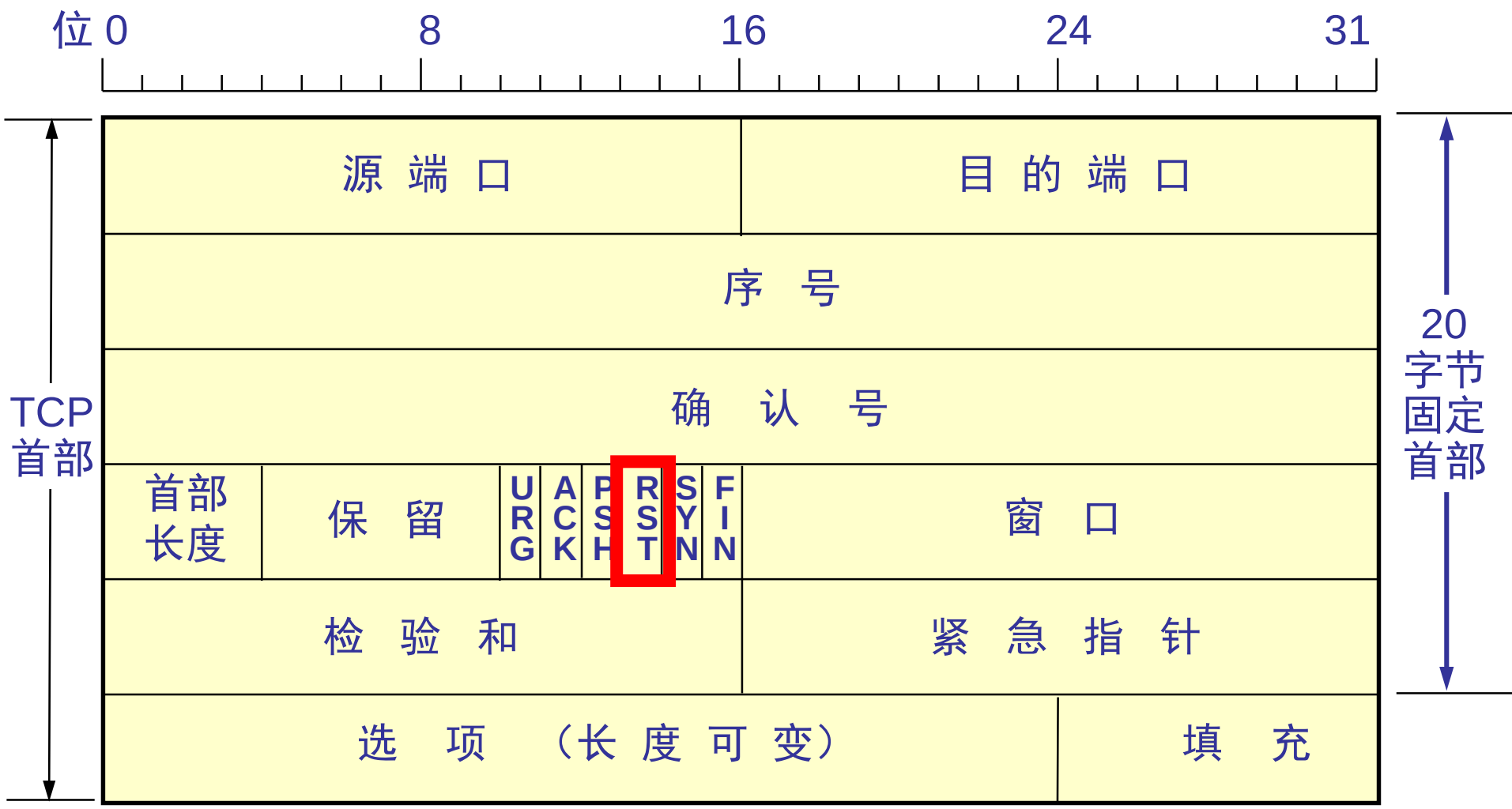
紧急 URG —— 当 URG = 1 时，表明紧急指针字段有效。它告诉系统此报文段中有紧急数据，应尽快传送(相当于高优先级的数据)。



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



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



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

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

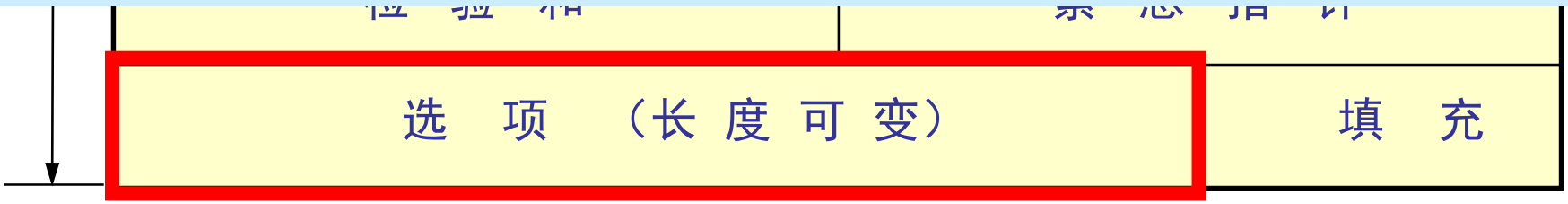
终止 FIN (FINis) —— 用来释放一个连接。FIN = 1 表明此报文段的发送端的数据已发送完毕，并要求释放运输连接。

窗口字段 —— 占 2 字节，用来让对方设置发送窗口的依据，单位为字节。用于流量控制，用于指示作为接受方，愿意接受的字节数量。

检验和 —— 占 2 字节。检验和字段检验的范围包括首部和数据这两部分。在计算检验和时，要在 TCP 报文段的前面加上 12 字节的伪首部。

紧急指针字段 —— 占 16 位，指出紧急数据最后一个字节的位置，也就是指出在本报文段中，紧急数据共有多少个字节（紧急数据放在本报文段数据的最前面）。

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



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

其他选项

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

填充字段 —— 这是为了使整个首部长度是 4 字节的整数倍。

TCP序号和确认号

序号:

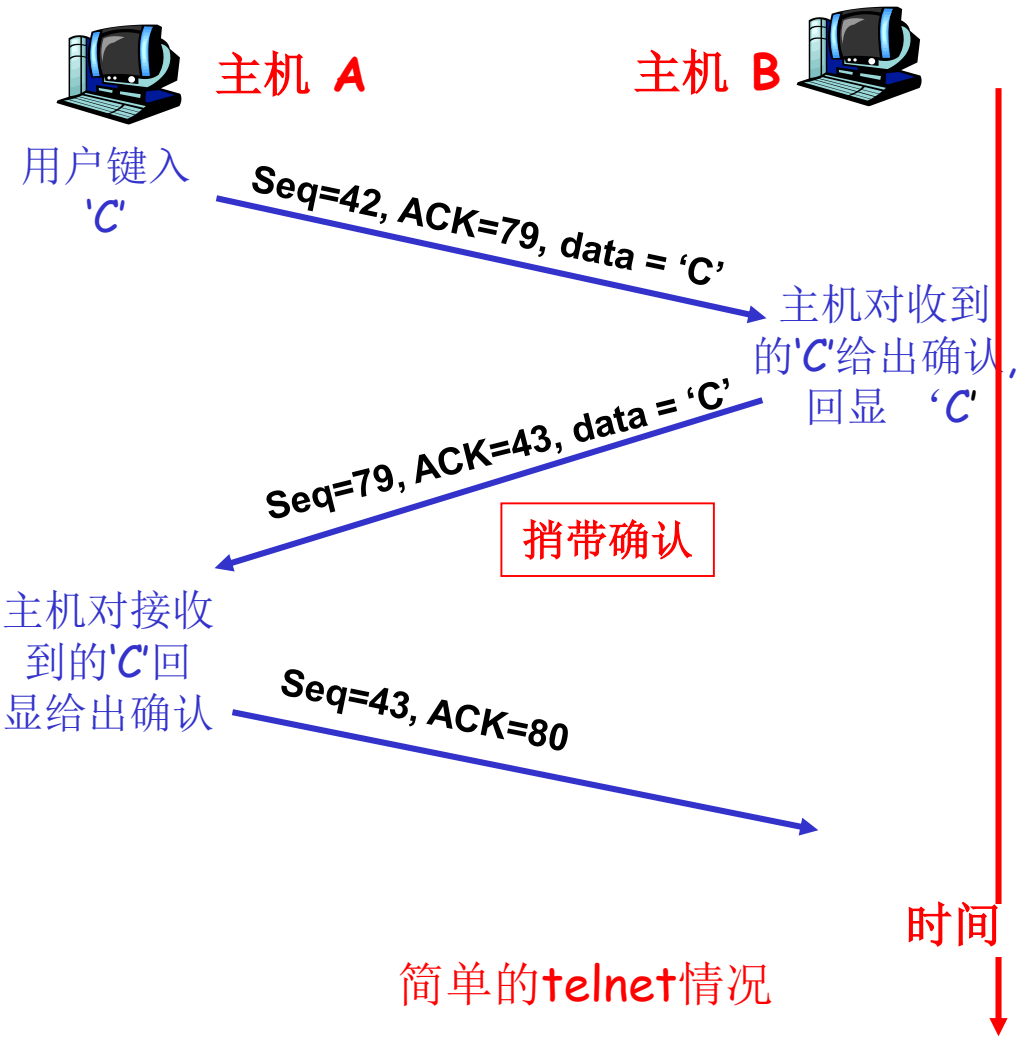
- 报文段中第1个数据字节在字节流中的位置编号

确认号:

- 期望从对方收到下一个字节的序号
- 累计应答

问题：接收方如何处理失序报文段?

回答：TCP规范没有说明，由实现者自行选择实现:抛弃/缓存



简单的telnet情况

3.5.3 TCP往返时延(RTT)的估计与超时

问题: 如何设置TCP 超时值?

- 应大于RTT
  - 但RTT是变化的
- 太短: 过早超时
  - 不必要的重传
- 太长: 对报文段的丢失响应太慢

问题: 如何估计RTT?

- SampleRTT: 从发送报文段到接收到ACK的测量时间
  - 忽略重传
- SampleRTT会变化,希望估计的RTT“较平滑”
  - 平均最近的测量值,并不仅仅是当前SampleRTT

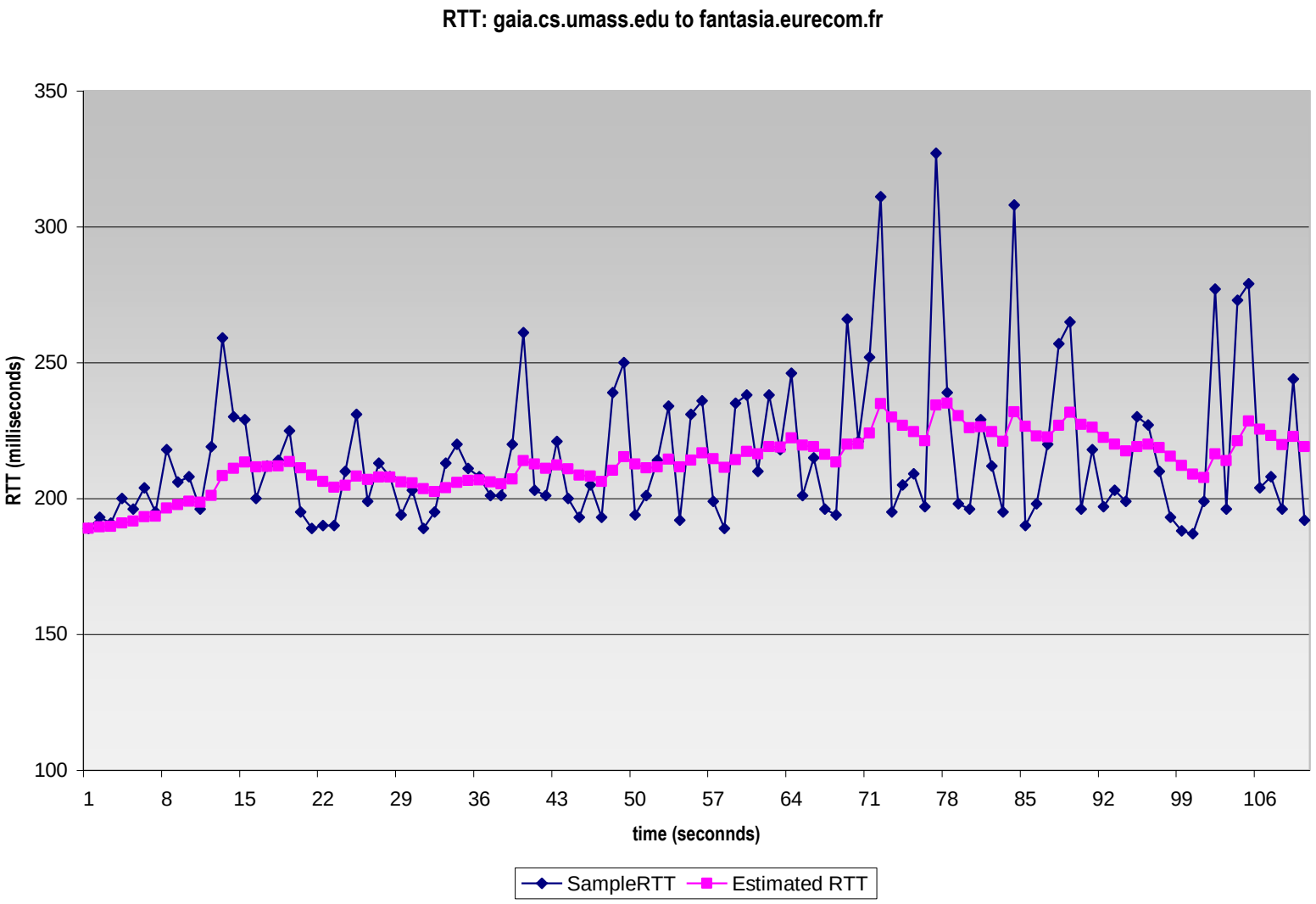
1、TCP往返时延估计

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

- 指数加权移动平均(Exponential weighted moving average)
- 过去的样本指数级衰减来产生影响
- 典型值: $\alpha = 0.125$

$$ERTT_{n+1} = (1-\alpha)*((1-\alpha)*ERTT_{n-1}+\alpha*SRTT_{n})+\alpha*SRTT_{n+1}$$

RTT估计的例子


RTT: gaia.cs.umass.edu to fantasia.eurecom.fr
RTT (milliseconds)
350
300
250
200
150
100
1
8
15
22
29
36
43
50
57
64
71
78
85
92
99
106
time (seconnds)
SampleRTT
Estimated RTT

2、设置和管理重传超时间隔

- 估算EstimatedRTT与SampleRTT之间差值有多大：

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

(典型地, β = 0.25)

设置超时间隔

- EstimtedRTT 加 “安全余量”
  - EstimatedRTT大变化-> 更大的安全余量

然后估算超时值:

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

TCP往返时延的估计和超时初始化

设置超时

- 初始时TimeoutInterval设置为1秒
- 第一个样本RTT获得后，

  EstimatedRTT=SampleRTT，

  DevRTT=SampleRTT/2，

  TimeoutInterval =EstimatedRTT + max (G, K*DevRTT) （K=4，G是用户设置的时间粒度）

第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

3.5.4 TCP 可靠数据传输

- TCP在IP不可靠服务的基础上创建可靠数据传输服务
  - 肯定确认和定时器
  - 序号
- 流水线发送报文段
- 累计确认
- TCP使用单个重传计时器
- 重传被下列事件触发:
  - 超时事件
  - 重复ACK
- 先考虑简化的TCP发送方:
  - 忽略重复ACK
  - 忽略流量控制，拥塞控制

TCP 发送方事件

1.从应用层接收数据:

- 根据序号创建报文段
- 序号是报文段中第一个数据字节的数据流编号
- 如果未启动，启动计时器(考虑计时器用于最早的没有确认的报文段)
- 超时间隔: TimeOutInterval= EstimatedRTT + 4*DevRTT

2.超时:

- 重传导致超时的报文段
- 重新启动计时器

3.收到确认:

- 如果确认了先前未被确认的报文段
  - 更新被确认的报文段序号
  - 如果还有未被确认的报文段，重新启动计时器

TCP 发送方(简化的)

```
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)
```

TCP 发送方(简化的)

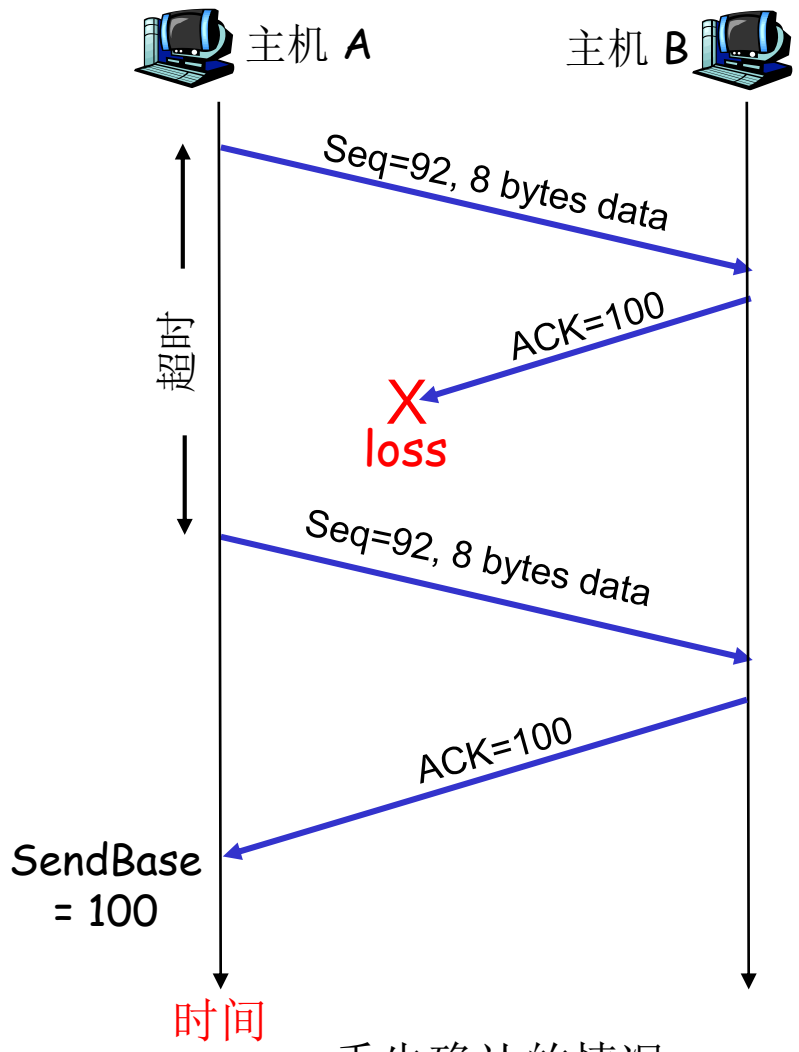
```
event: timer timeout
    retransmit not-yet-acknowledged segment with
        smallest sequence number
    start timer

event: ACK received, with ACK field value of y
    if (y > SendBase) { /* 累计确认到Y */
        SendBase = y
        if (there are currently not-yet-acknowledged segments)
            start timer
        }

} /* end of loop forever */
```

注释: SendBase-1: 上次累计的已确认字节

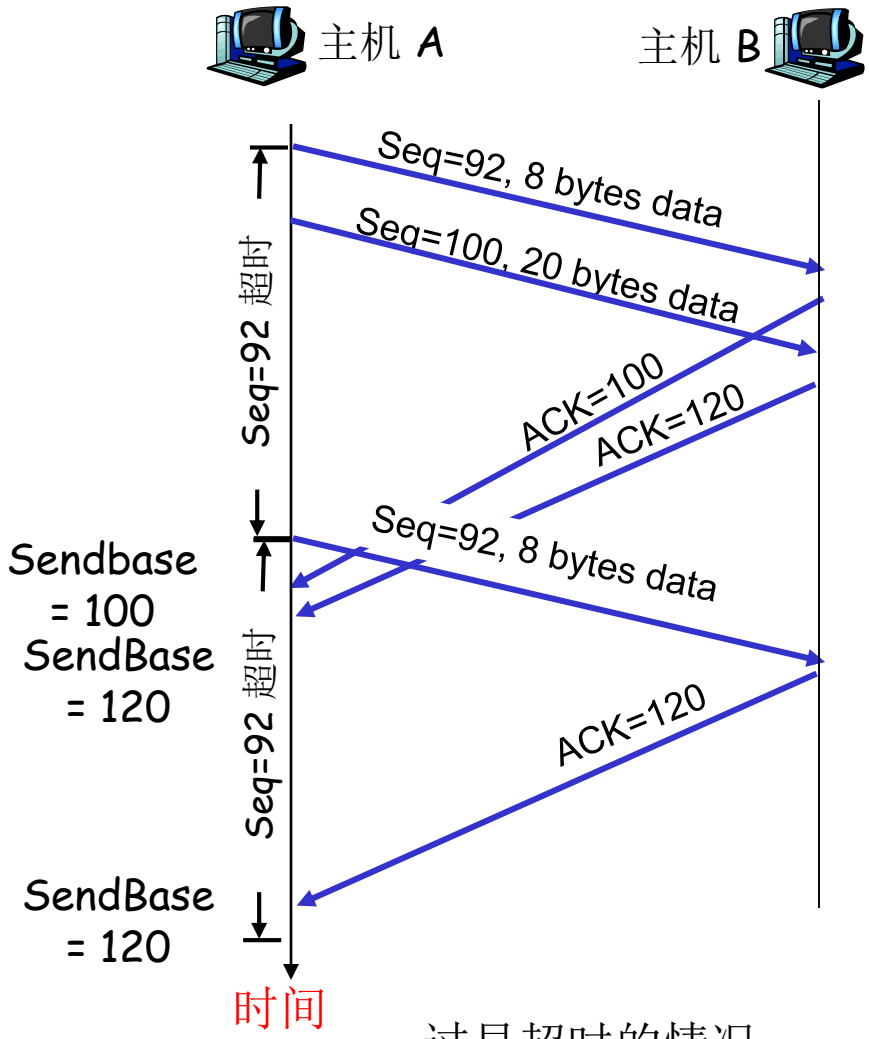
例如:

- SendBase-1 = 71; y= 73, 因此接收方期待73+ ;
  y > SendBase, 因此新数据被确认

1、TCP: 重传的情况

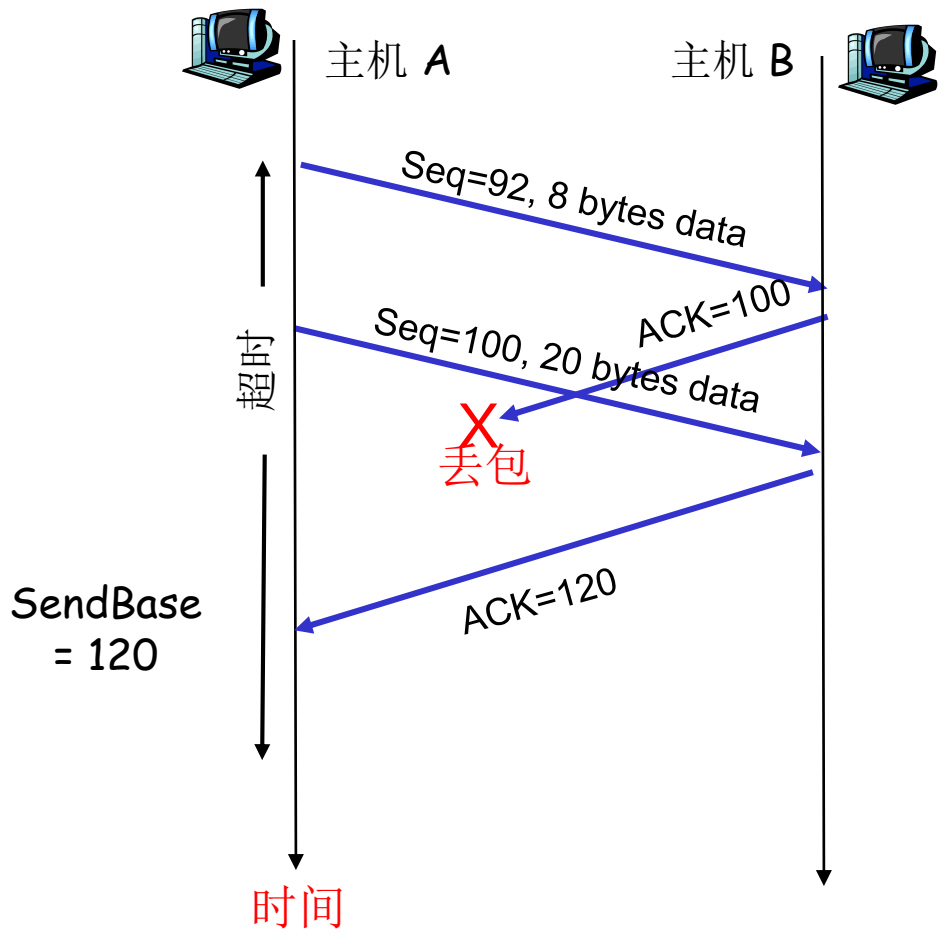


丢失确认的情况



过早超时的情况

1、TCP: 重传的情况



累计确认情况

2、超时间隔加倍

- TCP每次重传，都会把下一次的超时间隔设置为先前值的两倍。
- 但是当收到上层应用的数据和收到ACK两个事件中的任何一个发生时，定时器的`TimeoutInterval`值恢复为由近期的`EstimatedRTT`和`DevRTT`计算得到。
- 这种修改，提供了一种形式受限的拥塞控制。

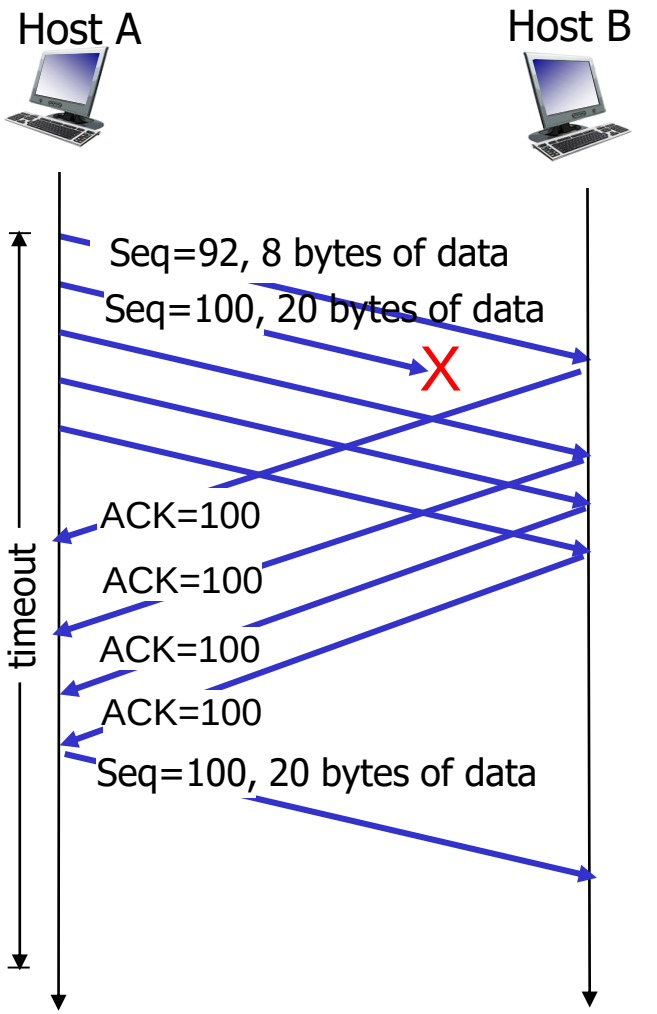
3、快速重传

- 超时间隔常常相对较长:
  - 重传丢失报文段以前有长时延
- 通过冗余ACK，检测丢失的报文段
  - 发送方经常一个接一个的发送报文段
  - 如果报文段丢失，将会收到很多重复ACK
- 如果对相同数据，发送方收到3个冗余的ACK, 假定被确认的报文段以后的报文段丢失了:
  - 快速重传: 在定时器超时之前重传

TCP ACK 产生 [RFC 1122, RFC 2581]

接收方事件	TCP 接收方行为
所期望序号的报文段按序到达。所有在期望序号及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500 ms。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）
部分或者完全填充已接收到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、快速重传



fast retransmit after sender receipt of triple duplicate ACK

快速重传算法:

```
事件: 收到ACK, ACK 域的值为 y
            if (y > SendBase) {
                SendBase = y
                if (当前还有没有确认的报文段)
                    启动定时器
            }
            else {
                值为 y的重复确认的次数加1
                if (值为 y的重复确认的计数= 3) {
                    重传序号为y的报文段
                }
```

对已经确认的报文段
收到一个重复ACK

快速重传

第3章 要点

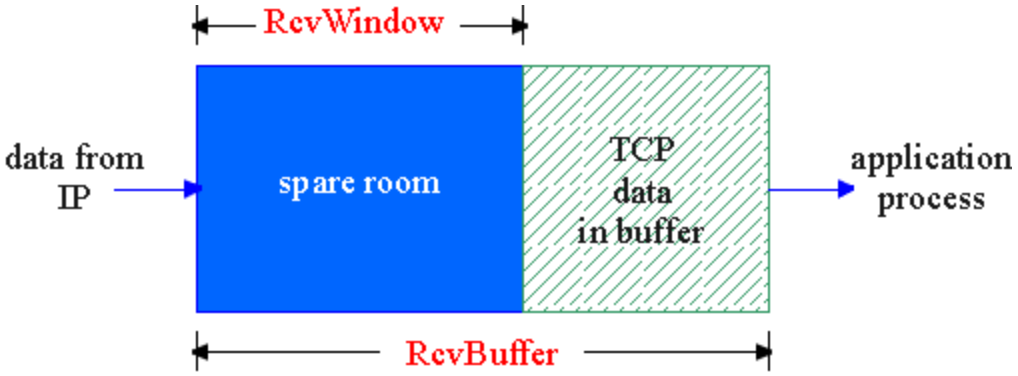
- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

3.5.5 TCP 流量控制

流量控制

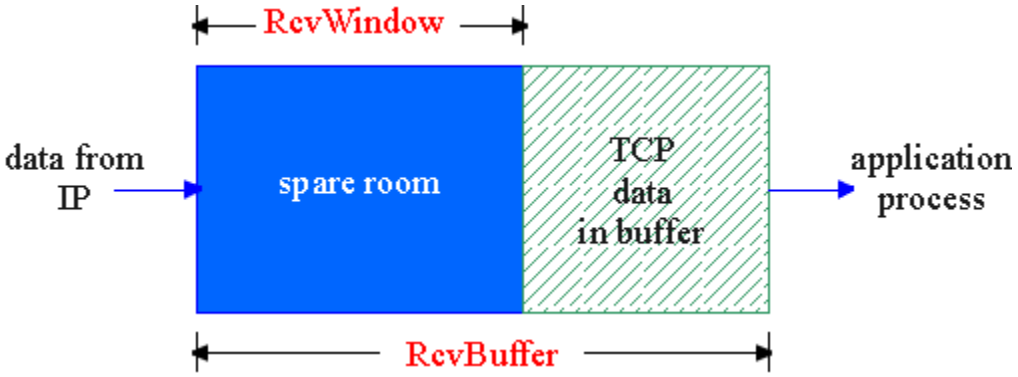
发送方不能发送太多、太快的数据让接收方缓冲区溢出

- TCP连接的接收方有1个接收缓冲区:



- 匹配速度服务: 发送速率需要匹配接收方应用程序的提取速率
- 应用进程可能从接收缓冲区读数据缓慢

TCP流控: 工作原理



(假设 TCP 接收方丢弃失序的报文段)

- 缓冲区的剩余空间

= RcvWindow

= RcvBuffer-[LastByteRcvd - LastByteRead]

- 接收方在报文段接收窗口字段中通告其接收缓冲区的剩余空间
- 发送方要限制未确认的数据不超过RcvWindow

LastByteSent-LastByteAcked <= RcvWindow

  - 保证接收缓冲区不溢出

第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

3.5.6 TCP 连接管理

回想: TCP 发送方与接收方
在交换报文段前要先建连接

- 初始化 TCP 变量:
  - 序号
  - 缓冲区和流控信息 (如RcvWindow)
- *客户机:* 连接的发起方

```
clientSocket.connect((serverName,serverPort));
```

- *服务器:* 接受客户请求

```
connectionSocket, addr = serverSocket.accept();
```

三次握手

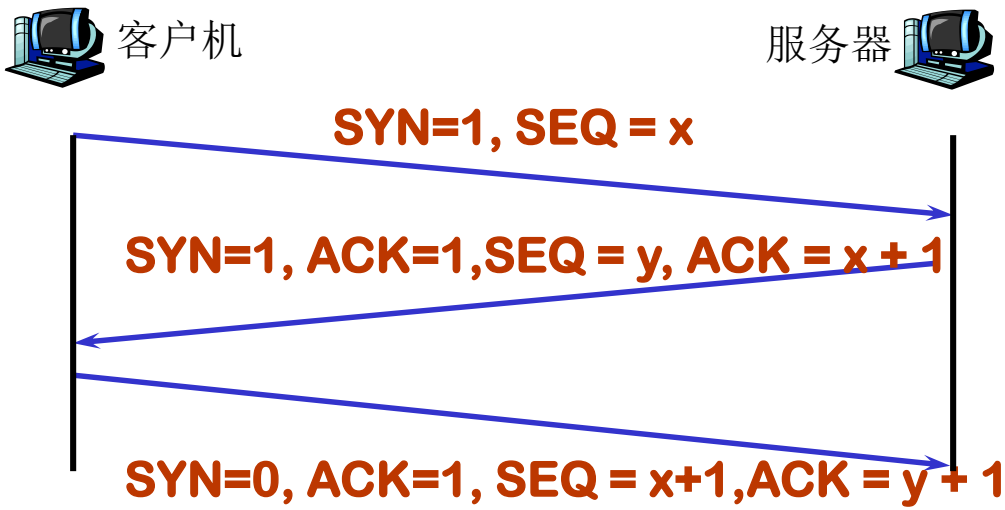
步骤 1: 客户机向服务器发送 TCP SYN报文段

- 指定初始序号
- 没有数据

步骤 2: 服务器收到SYN报文段, 用SYNACK报文段回复

- 服务器为该连接分配缓冲区和变量
- 指定服务器初始序号

步骤 3: 客户机接收到 SYNACK, 用ACK报文段回复,可能包含数据



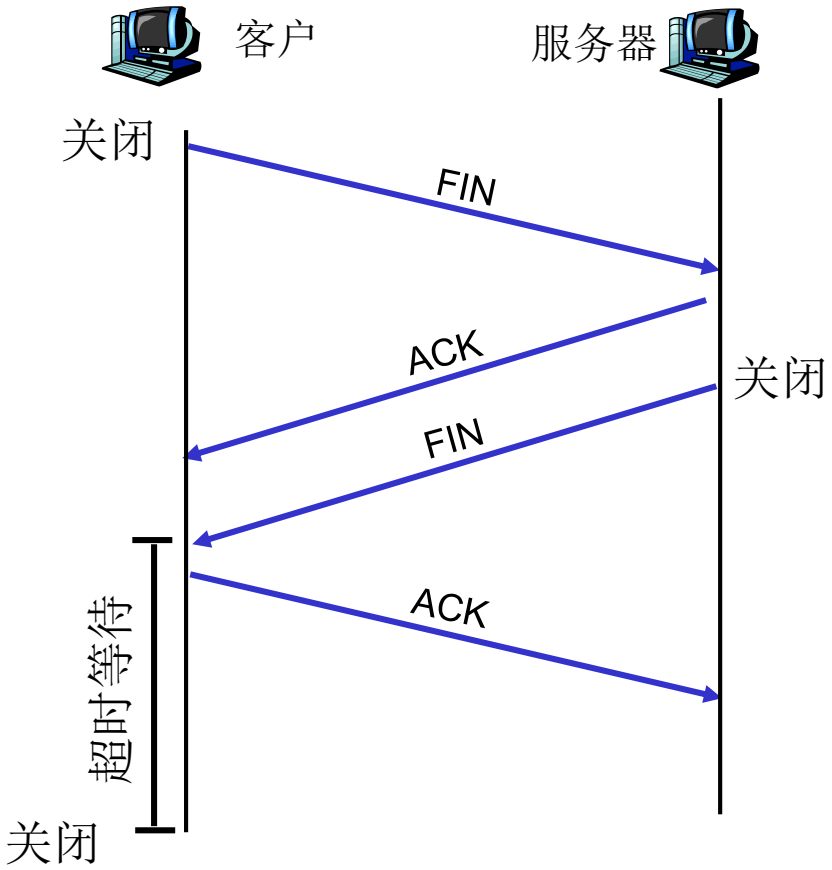
关闭连接

客户关闭套接字:

clientSocket.close();

步骤 1: 客户机向服务器发送TCP FIN控制报文段

步骤 2: 服务器收到FIN，用ACK回答。关闭连接，发送FIN



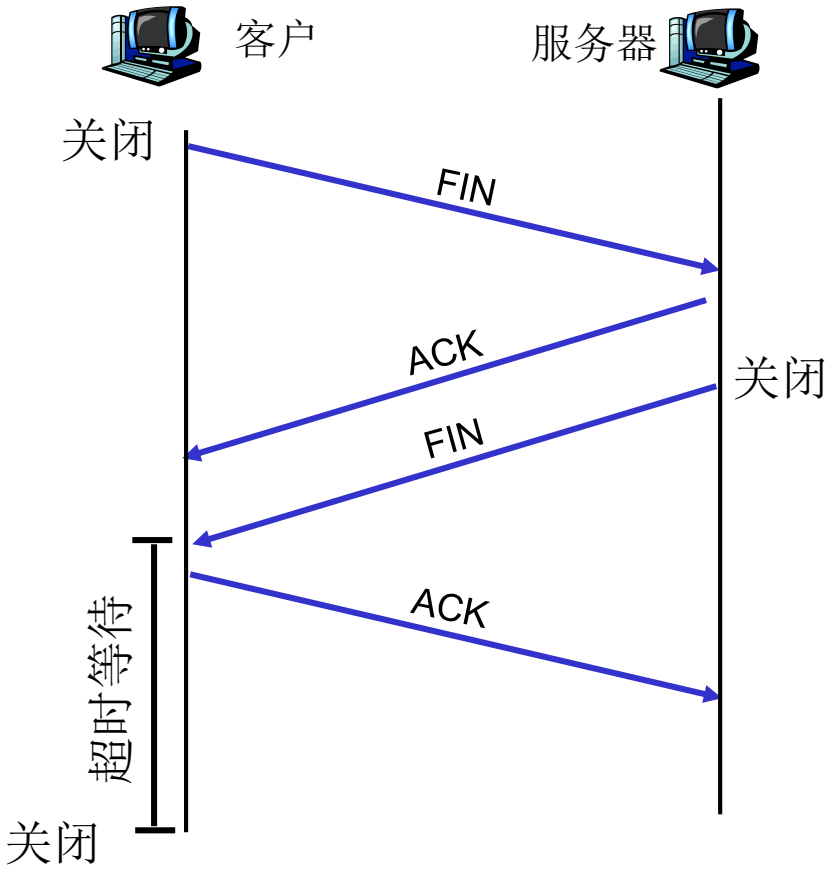
关闭连接

步骤 3: 客户机收到FIN，用ACK回答

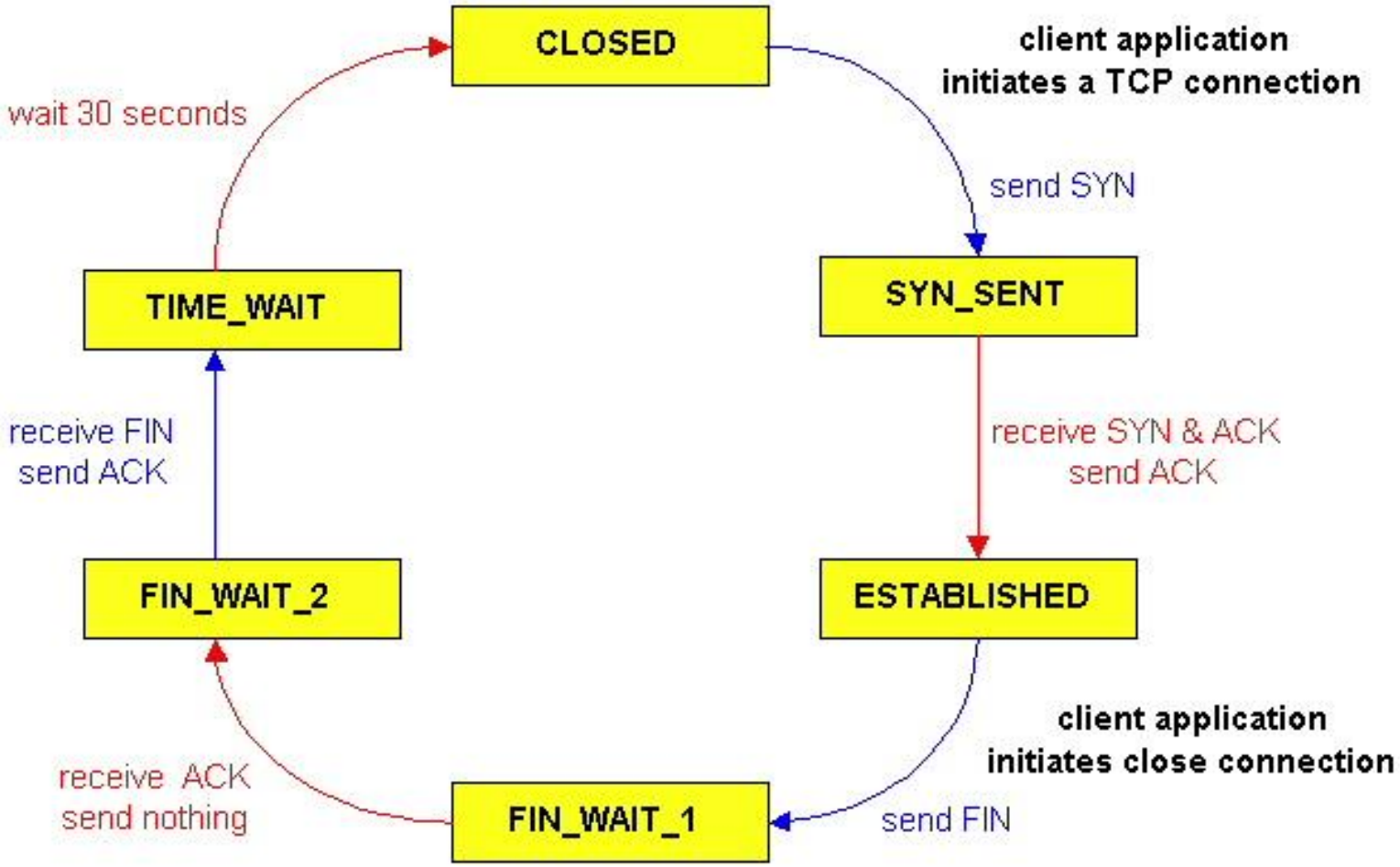
- 进入 “超时等待” – 将对接收到的FIN进行确认

步骤 4: 服务器接收ACK，连接关闭

注意: 少许修改，可以处理并发的FIN

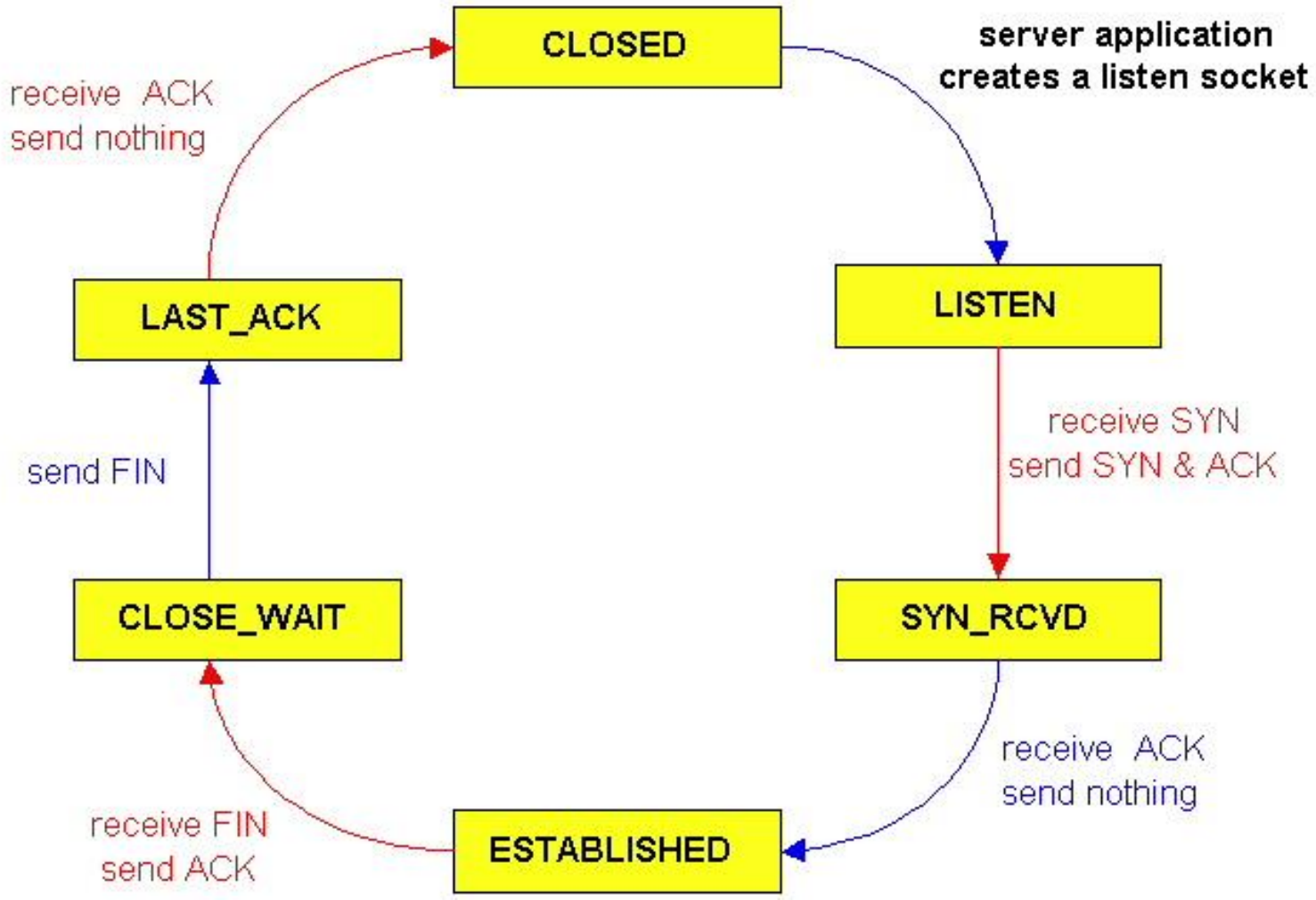


TCP 客户生命周期



TCP 客户端状态转换图

TCP 服务器生命周期



TCP 服务器端状态转换图

第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

3.6 拥塞控制原理

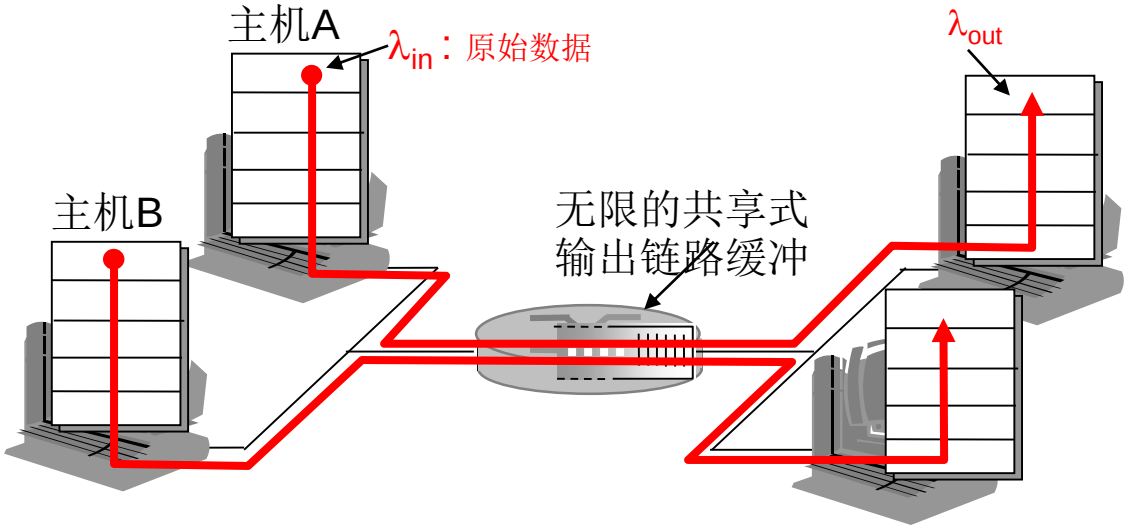
拥塞:

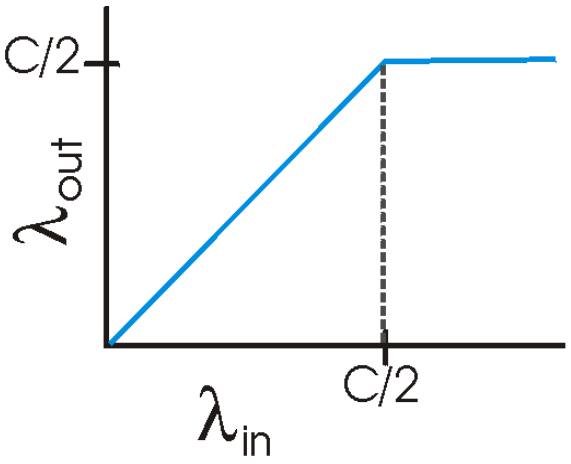
- 非正式地: “太多的源发送太多太快的数据，使网络来不及处理”
- 不同于流量控制!
- 表现:
  - 丢包 (路由器缓冲区溢出)
  - 长时延 (路由器缓冲区中排队)

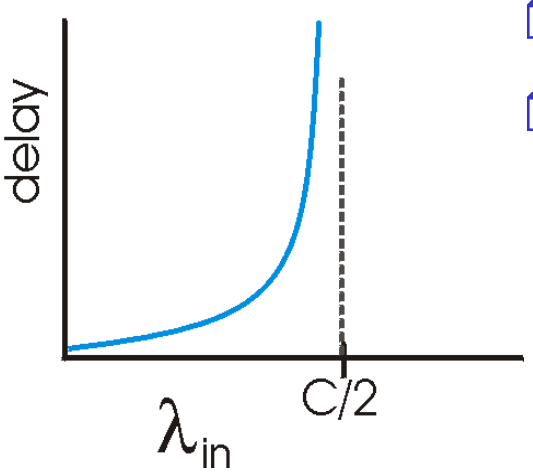
3.6.1 拥塞的原因与开销

1、拥塞的原因与开销: 情况1

- 两个发送方，两个接收方
- 一个路由器，无限缓冲区
- 不重传







- 可达到最大吞吐量
- 拥塞的代价一：当分组的到达率接近链路的容量时，分组将经历较大的排队时延。

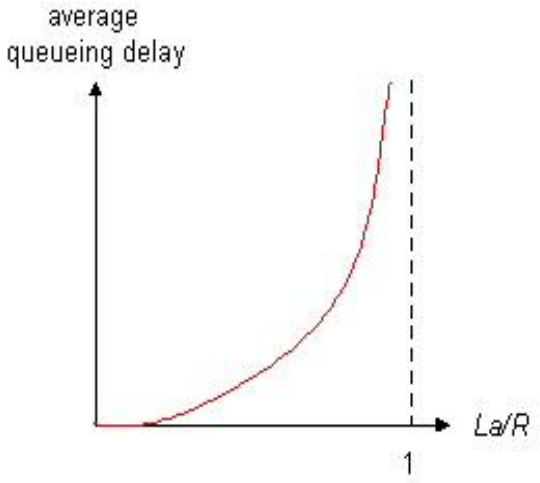
回顾

（二）排队时延和丢包

2、平均排队时延与流量强度关系：

- 设，R= 链路速率 (bps)、L= 分组长度 (比特)、a= 平均分组到达速率（每秒分组，pkt/s）

则流量强度=*La/R*（比特到达队列速率是*La* bit/s）

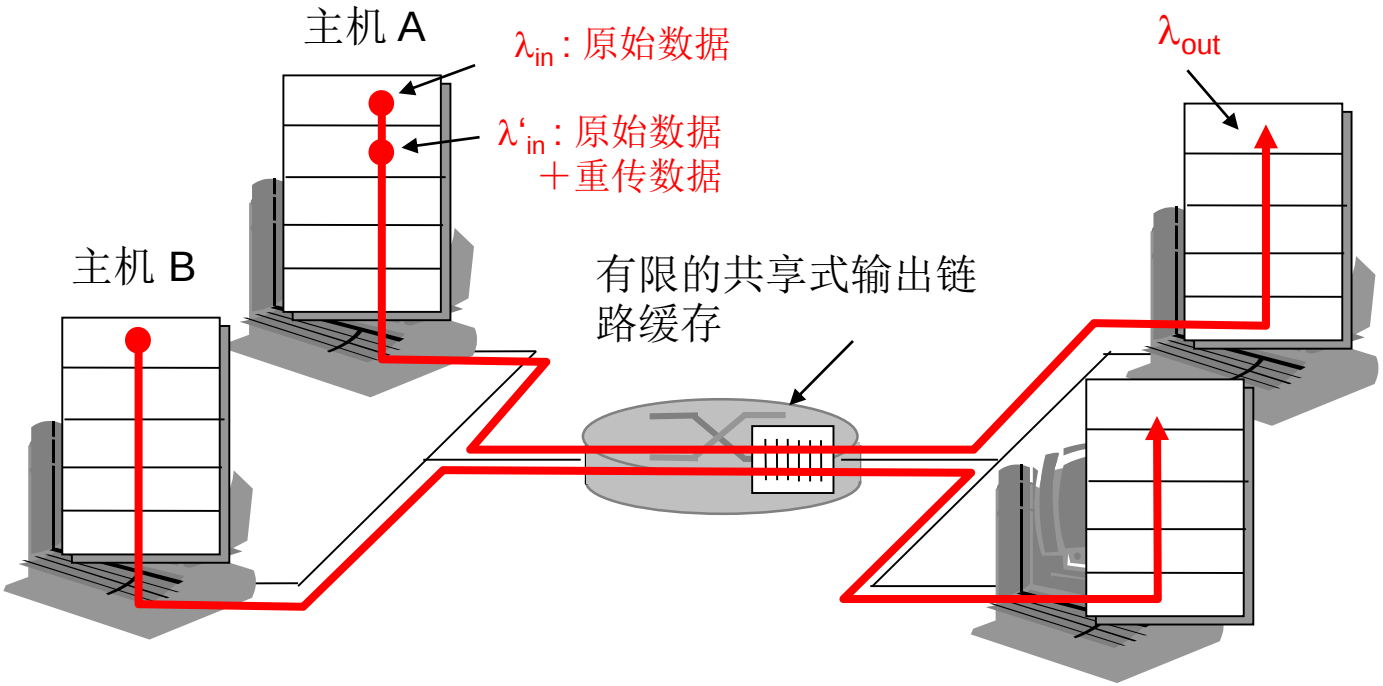


- ✓ La/R ~ 0：平均排队时延小（接近0）。几乎没有分组到达或间隔很大（稀疏），到达的分组几乎不排队。
- ✓ La/R → 1：分组陆续到达，形成队列，时延变大。
- ✓ La/R > 1：更多“分组”到达，超出了服务能力，平均时延无穷大！

设计系统时流量强度不能大于1。

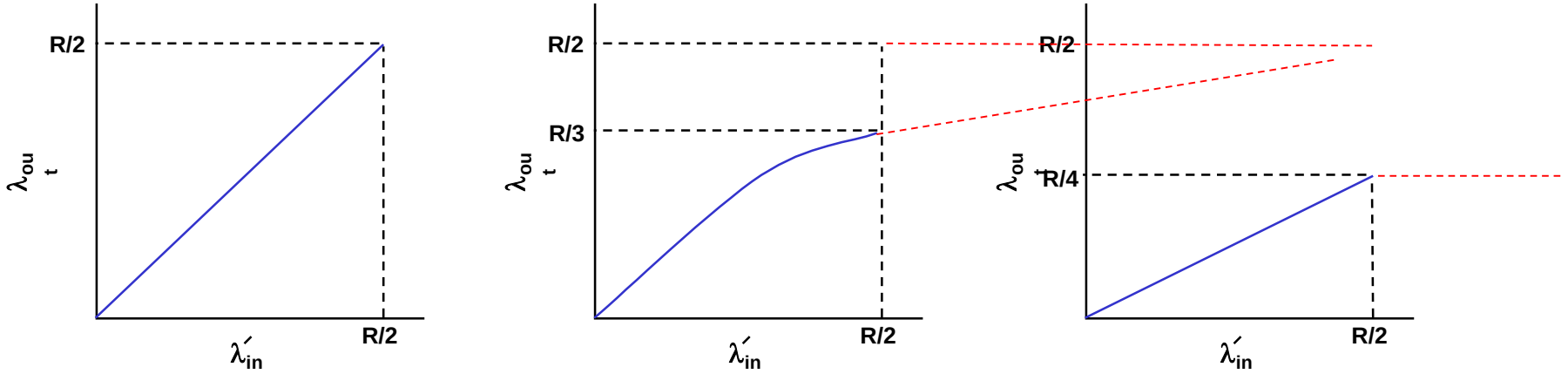
2、拥塞的原因与开销：情况2

- 一个路由器，*有限*缓冲区
- 发送方重传丢失的数据分组



2、拥塞的原因与开销：情况2

- 情形1:假设不会丢包 $\lambda_{in} = \lambda_{out}$ (吞吐量)
- 情形2:仅当丢失丢包时，需要“完美的” 重传：$\lambda'_{in} > \lambda_{out}$
- 情形3:迟延的分组（而不是丢失）的重传



a. b.假设有一半的分组重发 c.假设每个分组重发1一次

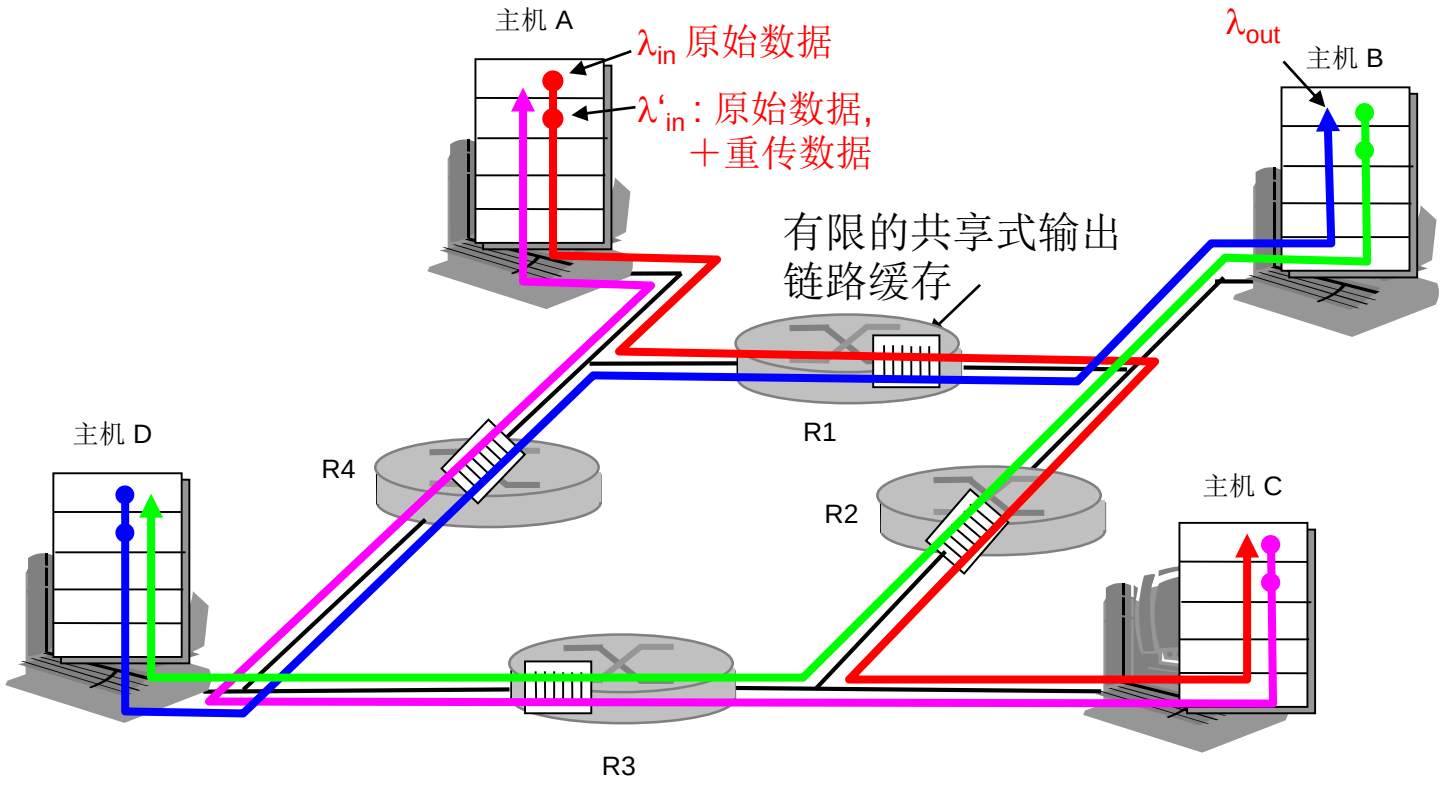
拥塞的“代价”：

- 发送方必须执行重传以补偿因缓存溢出而丢弃的分组。
- 不必要重传: 链路利用其带宽转发不必要的分组副本

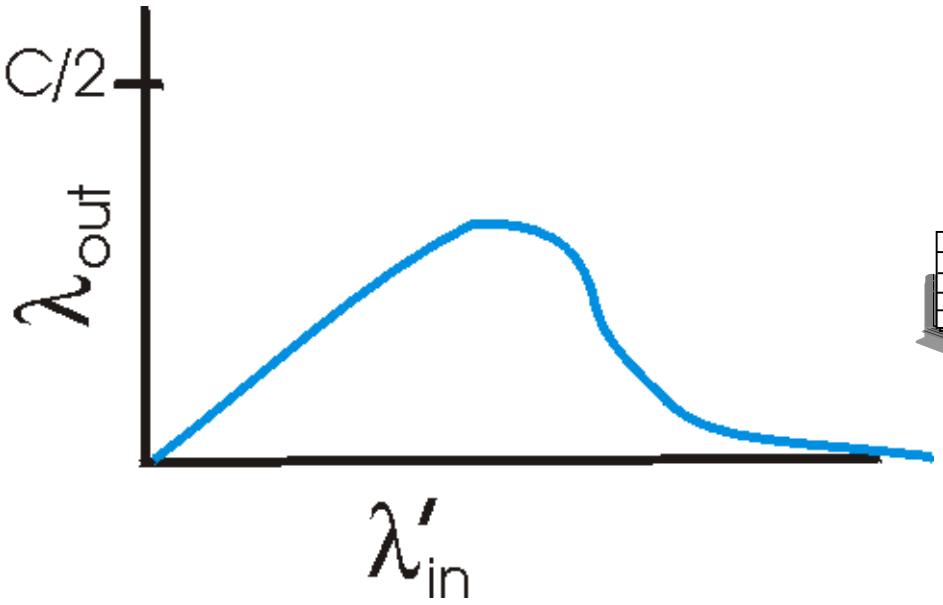
3、拥塞的原因与开销: 情况3

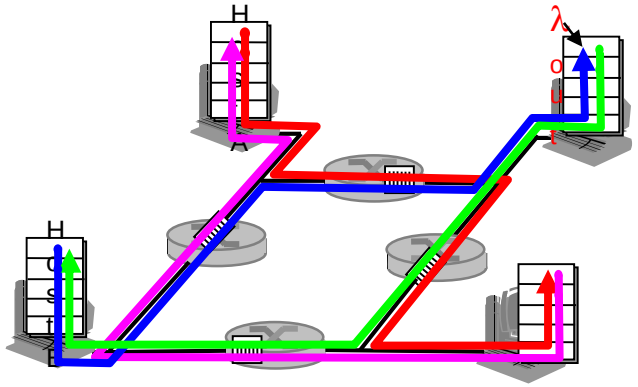
- 四个发送者
- 多跳路径
- 超时/重传

问题: 随着 λ_{in} 和 λ'_{in} 的增加将发生什么情况？



3、拥塞的原因与开销: 情况3





另一个拥塞的“开销”:

- 当分组丢失时, 任何用于传输该分组的上游传输能力都被浪费!

3.6.2 拥塞控制方法

根据网路层是否为运输层拥塞控制提供了显示帮助，来区分拥塞控制方法。分为两类方法:

端到端的拥塞控制:

- 网路层没有为运算层提供显示的支持。
- 从端系统根据观察到的时延和丢失现象推断出拥塞
- 这是TCP所采用的方法

网络辅助的拥塞控制:

- 路由器为端系统提供反馈
  - 一个bit指示一条链路出现拥塞(SNA,DECnet)
  - 指示发送方按照一定速率发送（ATM）
  - 两种形式

3.6.3 案例研究: ATM ABR 拥塞控制

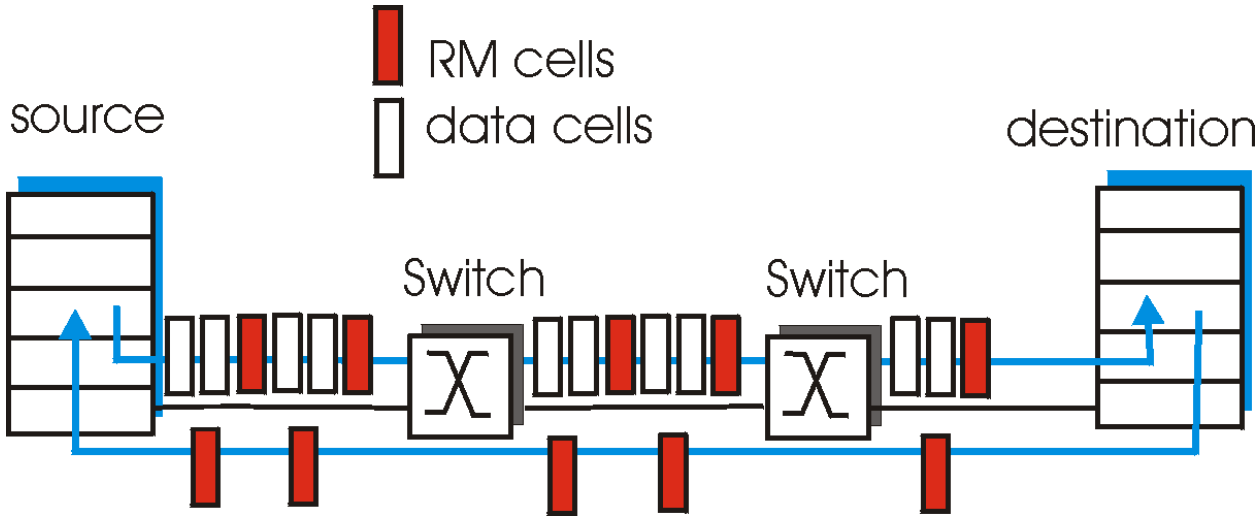
ABR: 可用比特率:

- "弹性服务"
- 如果发送方的路径 "欠载":
  - 发送方应该使用可用的带宽
- 如果发送方的路径拥塞:
  - 发送方被抑制到最小的保证速率

RM (资源管理) 信元:

- 发送方发送RM 信元, 散布在数据信元中
- 由交换机设置 RM 信元中的特定比特("*网络辅助*")
  - NI bit: 速率无增长 (轻度拥塞)
  - CI bit: 拥塞指示
- 接收方向发送方返回RM 信元

案例研究: ATM ABR拥塞控制



- RM信元中的两字节 ER (明确速率)字段
  - 拥塞的交换机会降低RM信元中的ER 值为
  - 发送方以路径上所有交换机的最小支持速率发送
- 数据信元中的EFCI bit : 被拥塞的交换机设置为1
  - 如果比RM信元先到达的数据信元的EFCI位为1，接收方将在返回的RM信元的CI位置1

第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

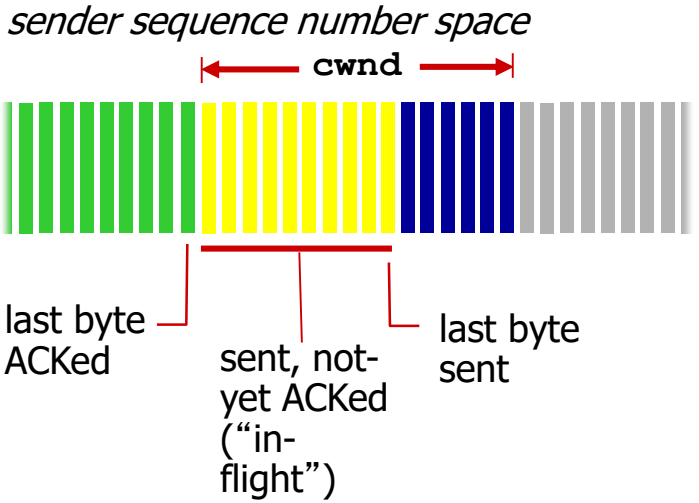
3.7 TCP 拥塞控制

采用端到端控制 (没有网络辅助)

三个问题?

- 一个TCP发送方如何限制它向其连接发送速率的?
- 一个TCP发送方如何感知从它到目的地之间的路径上存在拥塞的?
- 当发送方感知到端到端的拥塞时，采用何种算法来改变其发送速率?

3.7 TCP 拥塞控制



1、发送方如何限制其发送速率?

- 发送方通过CongWin限制传输:

  LastByteSent-LastByteAcked

  ≤min{ CongWin，RecWin}

- 粗略地,

  $$\text{最大平均速率} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- 拥塞窗口是动态的, 通过调节CongWin的值，发送方因此能调整它向连接发送数据的速率。

2、发送方如何感知网络拥塞?

- 丢失事件 = 超时*或者* 3个重复ACK
- 发生丢失事件后，TCP发送方降低速率(拥塞窗口)
- 正常时增加发送速率。
  - 自计时

3.7 TCP 拥塞控制

3、TCP发送方怎样确定它应当发送的速率呢?

既使得网络不会拥塞，与此同时又能充分利用所有可用的带宽。

TCP用下列指导原则回答这些问题:

- 一个丢失的报文段意味着拥塞，应当降低TCP发送速率。
- 当收到未确认报文段的确认到达时，能够增加发送方的速率。
- 带宽探测。
  - 每个TCP发送方根据异步于其他发送的本地信息而行动。

3.7 TCP 拥塞控制

TCP拥塞控制算法，包括3个主要部分:

- 慢启动
- 拥塞避免
- 快速恢复

慢启动和拥塞避免是TCP的强制部分。

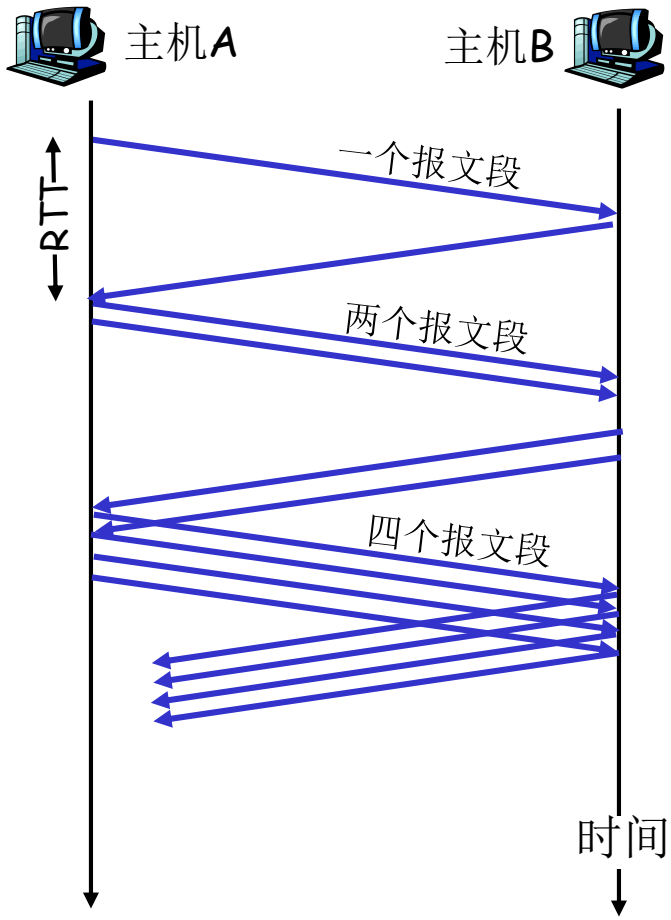
快速恢复是推荐部分，对TCP发送方并非是必需的。

1、TCP慢启动

- 在连接开始时，拥塞窗口值 = 1 MSS
  - 例如: MSS= 500 bytes & RTT = 200 msec
  - 初始化速率 = 20 kbps
- 可获得带宽可能 >> MSS/RTT
  - 希望尽快达到期待的速率
- 当连接开始，以指数快地增加速率，直到第一个丢失事件发生

1、TCP慢启动

- 当连接开始的时候，速率呈指数式上升，直到第1次报文丢失事件发生为止:
  - 每RTT倍增拥塞窗口值
  - 当传输报文段首次被确认，拥塞窗口增加一个MSS。
- 总结: 初始速率很低，但以指数快地增加



1、TCP慢启动

何时结束这种指数增长方式？

- 超时事件以后:
  - CongWin值设置为1 MSS
  - Ssthresh= CongWin /2
- 窗口指数增长(慢启动状态)，到达一个阈值 (ssthresh)后，再线性增长(拥塞避免状态)
- 收到3个冗余确认后:
  - CongWin减半
  - 进入快速恢复阶段

TCP RENO

在TCP Tahoe版本中，收到3个重复ACK后，发送方的处理与超时事件的处理相同

- 3个冗余ACK指示网络还具有某些传送报文段的能力
- 超时则更为“严重”

1、TCP慢启动

问题: 什么时候从指数增长转变为线性增长?

回答: CongWin达到它超时以前1/2的时候。

进入拥塞避免状态。

实现方法:

- 设置一个变的阈值－ssthresh
- 在丢包事件发生时，阈值ssthresh设置为发生丢包以前的CongWin的一半

2、拥塞避免

一旦进入拥塞避免状态，CongWin的值大约是上次遇到拥塞时的一半。

- 每个RTT只将CongWin的值增加一个MSS。

何时结束拥塞避免的线性增长?

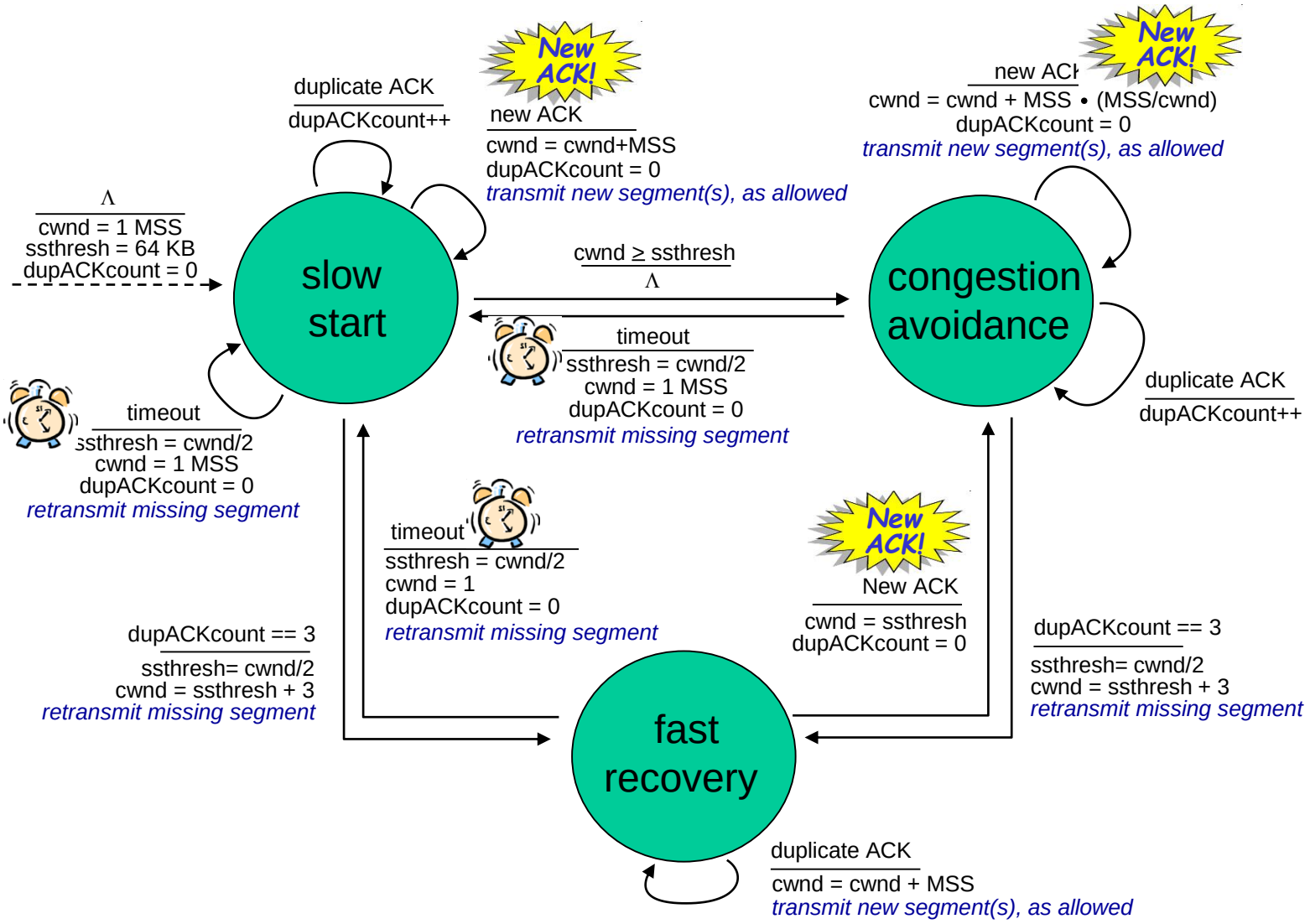
- 超时事件以后，迁移到慢启动状态:
  - CongWin值设置为1 MSS
  - Ssthresh= CongWin /2
- 收到3个冗余确认后:
  - CongWin减半+3个MSS
  - 进入快速恢复阶段

3、快速恢复

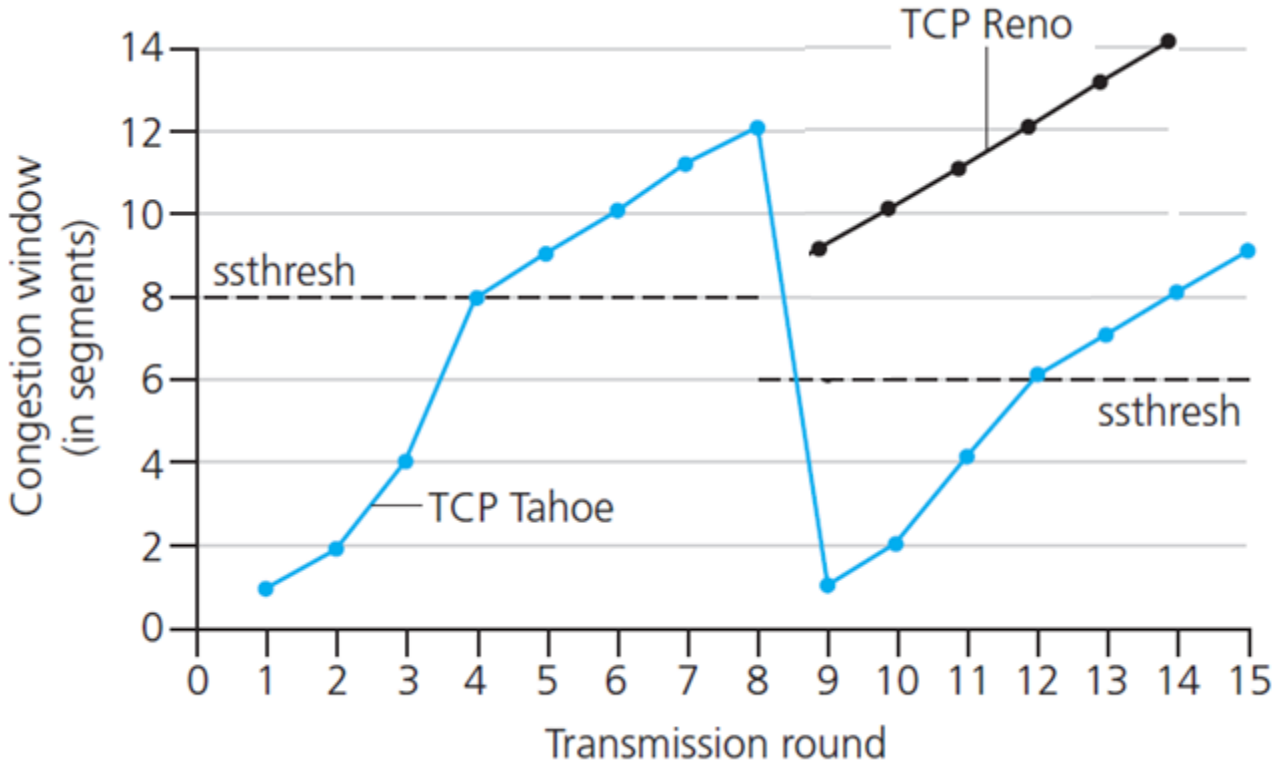
在快速恢复中：

- 对于引起TCP进入快速恢复状态的缺失报文段，每收到一个冗余的ACK，CongWin的值增加一个MSS。
- 最终，当对丢失的报文段的一个ACK到达时，TCP在降低 CongWin后进入拥塞避免状态。
- 如果出现超时事件，迁移到慢启动状态:
  - CongWin值设置为1 MSS
  - Ssthresh= CongWin /2

TCP拥塞控制的FSM描述



TCP拥塞窗口的演化


TCP Reno
ssthresh
ssthresh
TCP Tahoe
Congestion window
(in segments)
Transmission round
0
2
4
6
8
10
12
14
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

TCP 拥塞控制：小结

- 当CongWin < ssthresh时，发送者处于慢启动阶段，CongWin指数增长
- 当CongWin > ssthresh时，发送者处于拥塞避免阶段，CongWin线性增长
- 当出现3个冗余确认时，发送者处于快速恢复阶段。阈值ssthresh设置为CongWin/2，且CongWin设置为ssthresh+3*MSS
- 当超时发生时，阈值ssthresh设置为CongWin/2，并且CongWin设置为1 MSS.

TCP拥塞控制(回顾)：加增倍减 AIMD

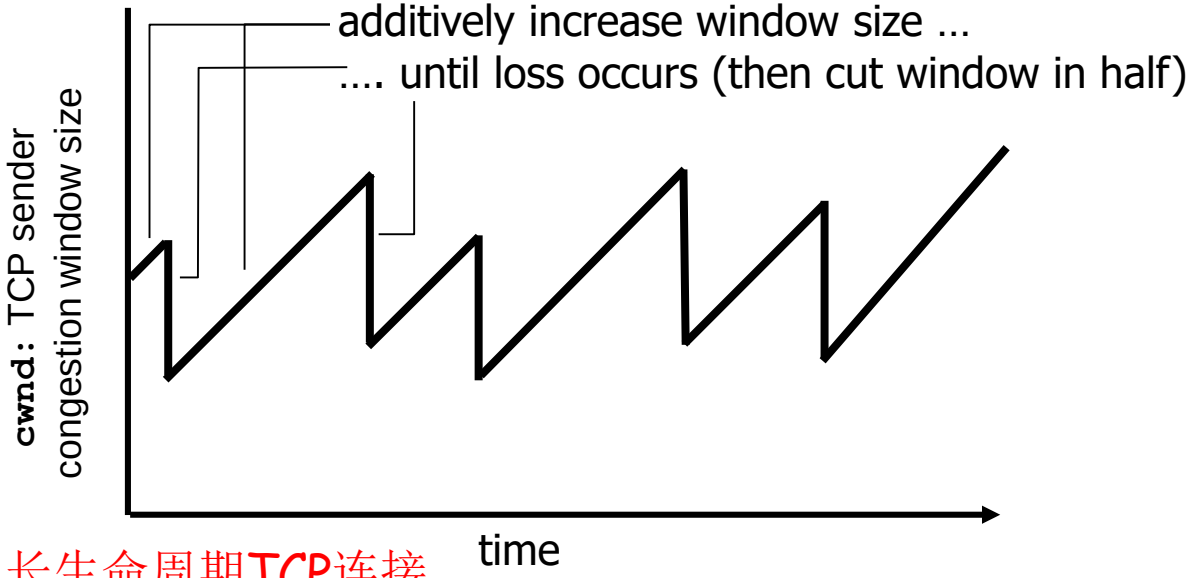
乘性减:

丢包事件后，拥塞窗口值减半

加性增:

如没有检测到丢包事件，每个RTT时间拥塞窗口值增加一个MSS (最大报文段长度)

AIMD saw tooth behavior: probing for bandwidth



长生命周期TCP连接

第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

2、TCP 吞吐量

- 作为窗口长度和RTT的函数，TCP的平均吞吐量是什么?
  - 忽略慢启动
- 设当丢包发生时窗口长度是W
- 如果窗口为 W，吞吐量是 W/RTT
- 当丢包发生后，窗口降为 W/2，吞吐量为 W/2RTT.
- 一个连接的平均吞吐量为0 .75 W/RTT

3、TCP未来

- 举例: 1500 字节的报文段, 100ms RTT, 要达到10 Gbps 的吞吐量
- 要求平均拥塞窗口长度 W = 83,333 包括传输中的报文段
- 根据丢包率，则一个连接的平均吞吐量为:

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

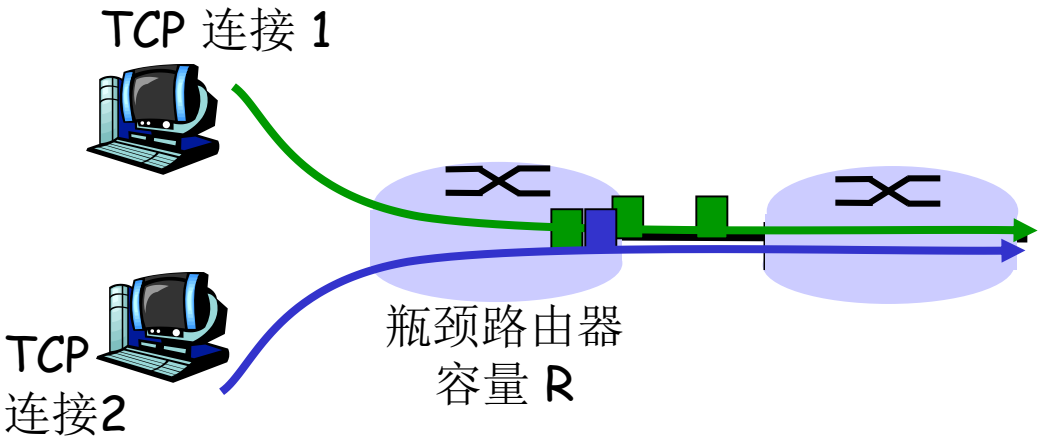
- ➜丢包率 L = $2 \cdot 10^{-10}$ *(难以达到，5亿个报文段丢失1个)*
- 需要高速下的TCP新版本!

第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

4、TCP 公平

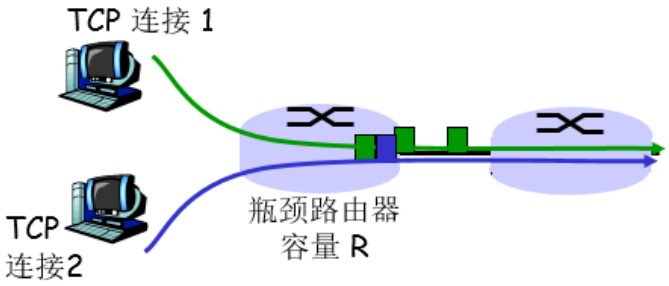
公平目标: 如果K个 TCP 会话 共享带宽为 R的链路瓶颈, 每个会话应有R/K的平均链路速率

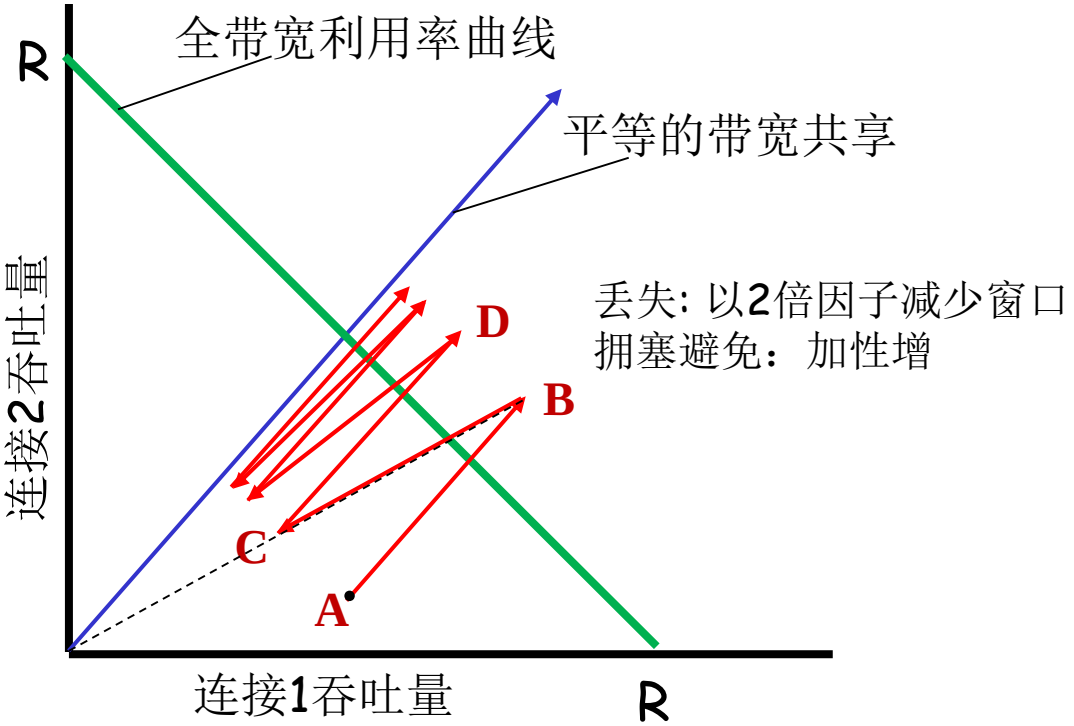


为什么TCP能保证公平性?

两个竞争会话:

- 假设两条连接具有相同的MSS和RTT。忽略慢启动，一直以AIMD方式运行。没有其他干扰流量。
- 随着吞吐量的增加，按照斜率1加性增加
- 等比例地乘性降低吞吐量





当多条连接共享一个共同的瓶颈链路时，那些具有较小RTT的连接能够在链路空闲时抢到可用带宽，因而比那些具有较大RTT的连接享用更高的吞吐量。

公平性(续)

公平性和UDP

- 多媒体应用通常不用TCP
  - 不希望拥塞控制抑制速率
- 使用UDP
  - 音频/视频以恒定速率发送，能容忍报文丢失
- 研究领域:

  TCP友好(TCP friendly)

公平性和并行TCP 连接

- 不能防止2台主机之间打开多个并行连接.
- Web浏览器以这种方式工作
- 例子:支持9个连接的速率R的链路:
  - 新应用请求一个TCP连接，则得到 R/10的带宽。
  - 新应用请求11个TCP连接，则得到R/2 的带宽!

第3章 要点

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输: UDP
- 3.4 可靠数据传输的原则
  - rdt1
  - rdt2
  - rdt3
  - 流水线协议
- 3.5 面向连接的传输: TCP
  - 报文段结构
  - 可靠数据传输
  - 流量控制
  - 连接管理
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制
  - 机制
  - TCP吞吐量
  - TCP公平性
  - 时延模型

5、时延模型

问题：从发送一个请求到从该Web服务器收到一个对象，需要多长时间?

忽略拥塞，时延受如下影响:

- 创建TCP连接
- 数据传输时延
- 慢启动

假设如下符号：

- 假定客户机和服务器间有一条速率为R的链路
- S: MSS (bits)
- O: 对象大小 (bits)
- 没有重传(没有丢失, 没有破坏)

窗口长度:

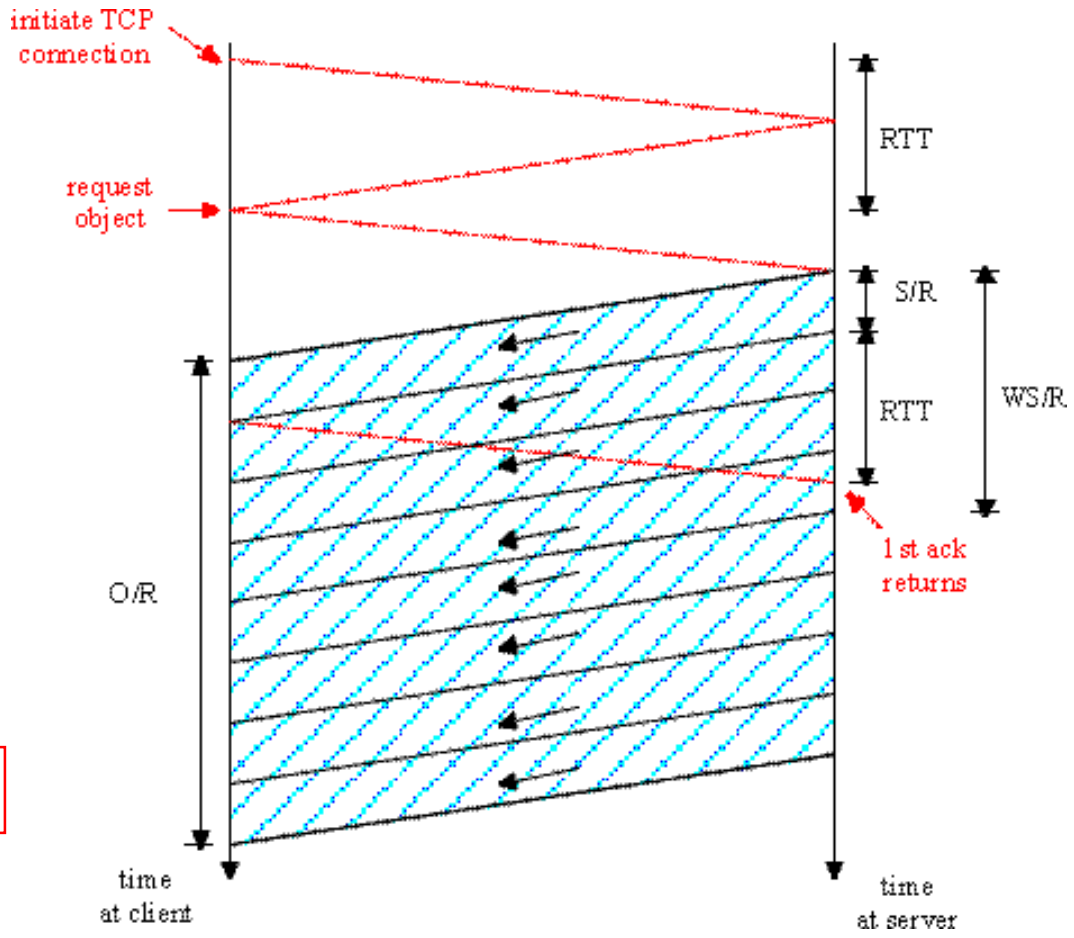
- 首先假设: 固定的拥塞窗口, W个报文段
- 然后动态窗口, 构模慢启动过程

固定的拥塞窗口(1)

第一种情况:

WS/R > RTT + S/R:

窗口中所有数据发送完之前,收到对第一个报文段的确认

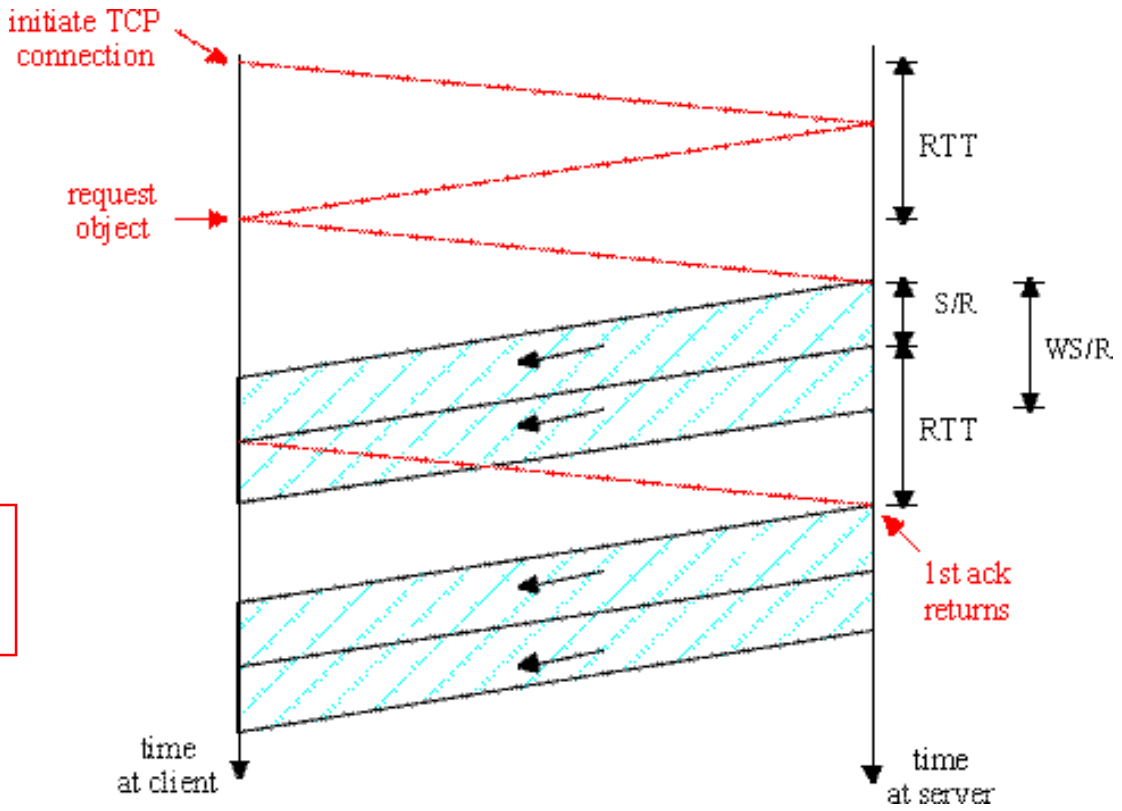


时延 = 2RTT + O/R

固定的拥塞窗口(2)

第二种情况:

- WS/R < RTT + S/R: 发送完窗口中的报文段，等待确认的到来



时延 = 2RTT + O/R
+ (K-1)[S/R + RTT - WS/R]

多减部分

TCP 时延模型: 慢启动 (1)

现在假设窗口根据慢启动方式增长

一个对象的传输时延是:

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

其中 P 是 TCP 在服务器中闲置的时间数量:

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

- 其中 Q 是服务器闲置的时间数量
  如果对象是无限大小.

- 并且 K 是对象占用窗口的数量.

TCP 时延模型: 慢启动(2)

时延组成:

- 2 RTT 用于连接建立和请求
- O/R 用于传输对象
- 由于慢启动导致服务器的空闲

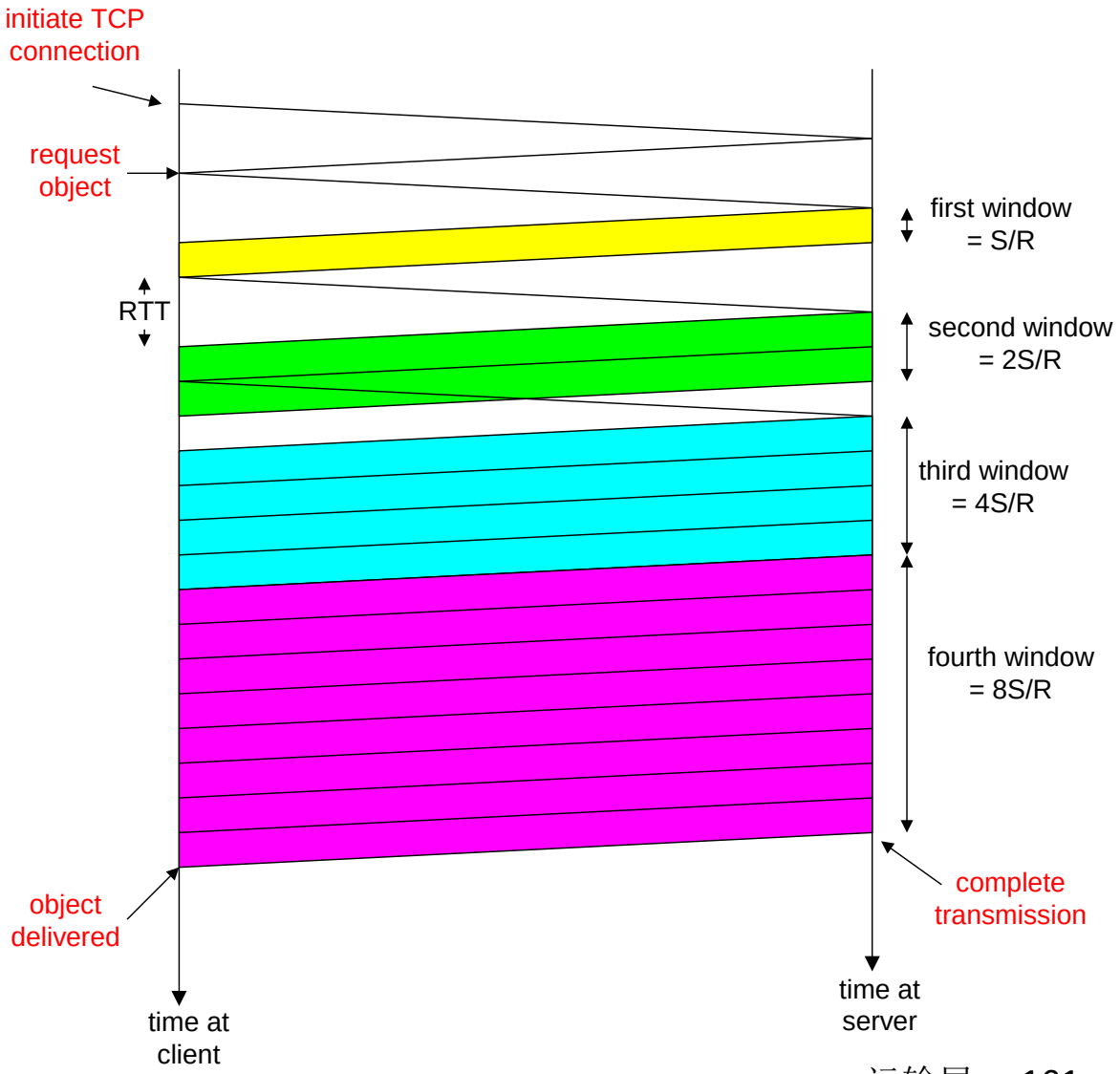
服务器空闲:

P = min{K-1,Q} times

例子:

- O/S = 15 报文段
- K = 4 窗口
- Q = 2
- P = min{K-1,Q} = 2

服务器空闲 P=2次



TCP时延模型 (3)

$\frac{S}{R}+RTT=$服务器开始发送数据到

服务器收到确认的时间

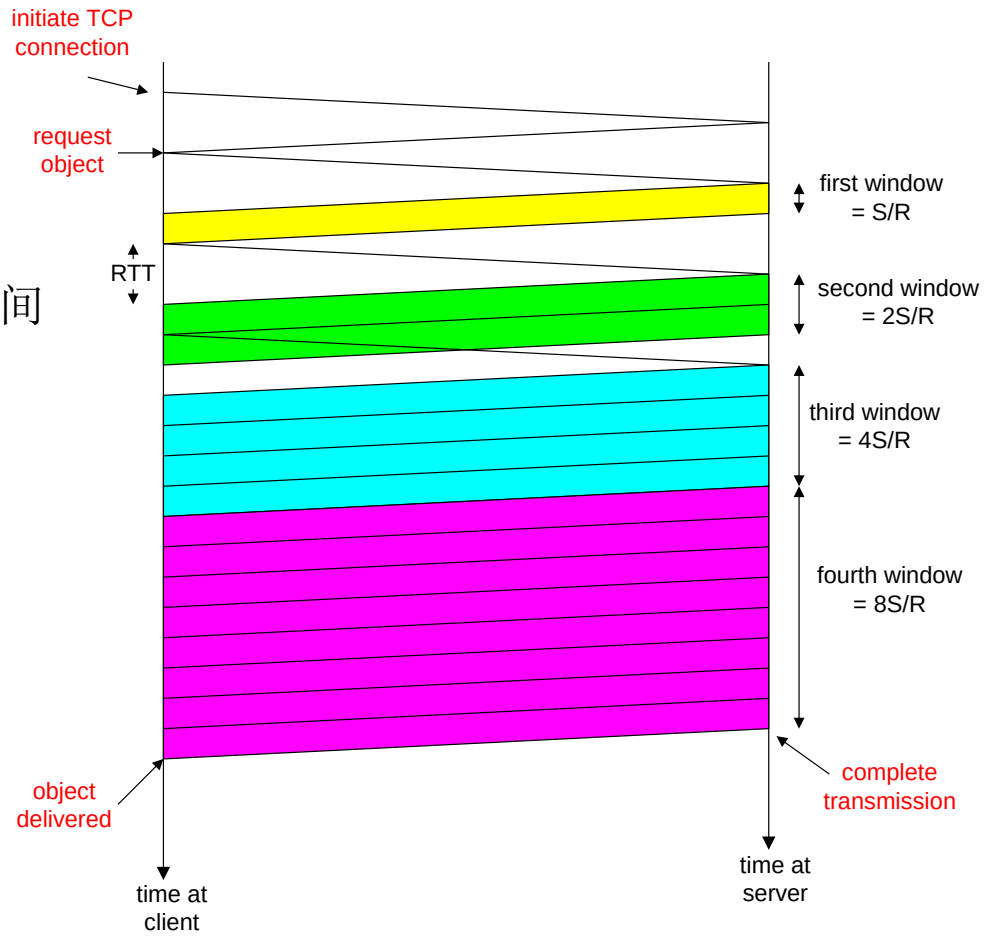
$2^{k-1}\frac{S}{R}=$传输第K个窗口的时间

$\left[\frac{S}{R}+RTT-2^{k-1}\frac{S}{R}\right]^{+}=$第K个窗口以后空闲时间

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

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

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



TCP时延模型 (4)

回想 K = 包含对象的窗口数量

我们如何计算 K？

$$
\begin{aligned}
K &= \min\{k : 2^0 S + 2^1 S + \cdots + 2^{k-1} S \geq O\} \\
&= \min\{k : 2^0 + 2^1 + \cdots + 2^{k-1} \geq O/S\} \\
&= \min\{k : 2^k - 1 \geq \frac{O}{S}\} \\
&= \min\{k : k \geq \log_2(\frac{O}{S} + 1)\} \\
&= \left\lceil \log_2(\frac{O}{S} + 1) \right\rceil
\end{aligned}
$$

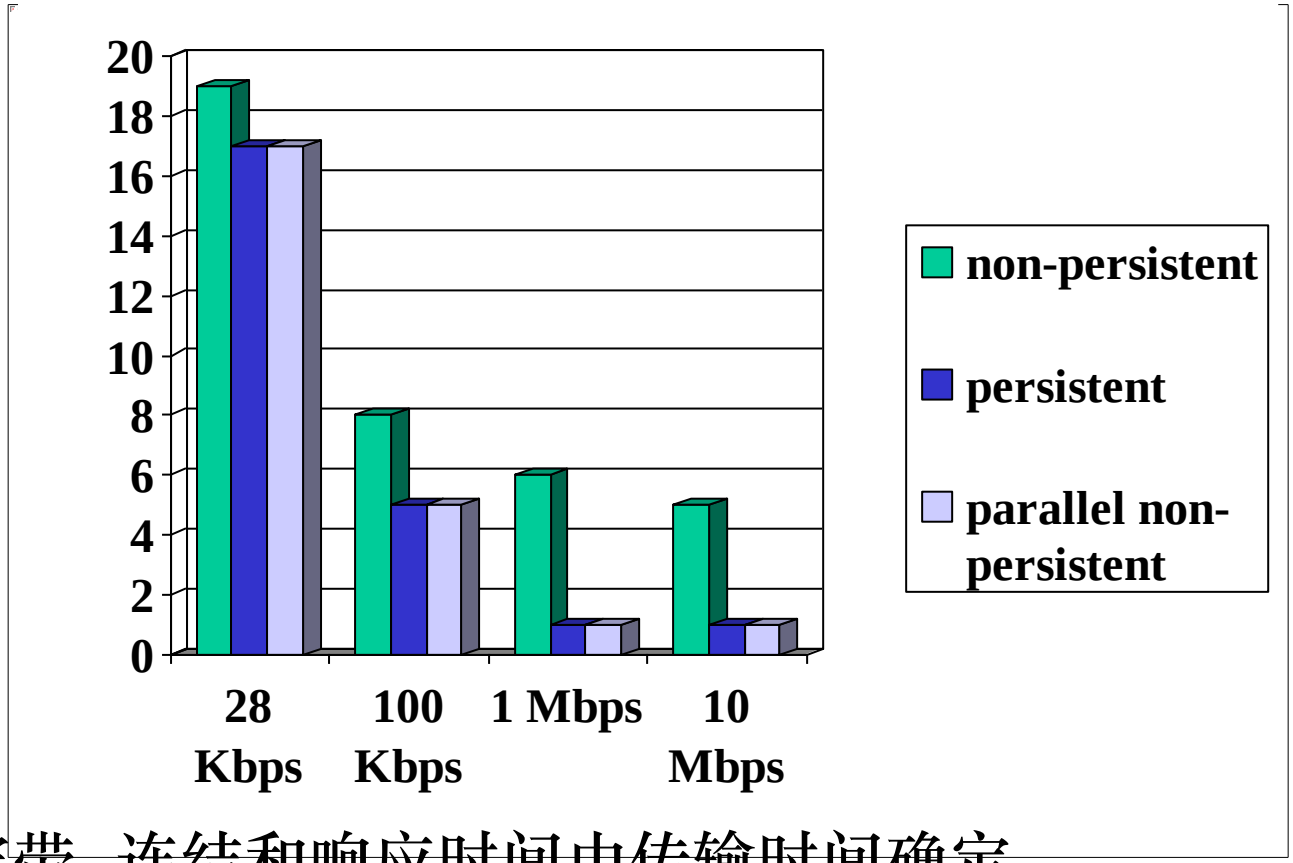
对无穷大对象，计算空闲次数Q是类似的.

HTTP构模

- 假设Web 页面有下列组成:
  - *1* 个基本的HTML页面 (*O* bits的大小)
  - *M*个图片 (每个 *O* bits的大小)
- 非持久HTTP:
  - *M+1*个连续的TCP 连接
  - *响应时间 = (M+1)O/R + (M+1)2RTT + 空闲时间总和*
- 持久HTTP:
  - *2 RTT* 用于请求和接受基本HTML 文件
  - *1 RTT*用于请求和接受 M个图片
  - *响应时间 = (M+1)O/R + 3RTT + 空闲时间总和*
- 有X个并行连接的非持久HTTP
  - 假设 M/X为整数.
  - 1 TCP 连接用于基本文件
  - M/X 个并行连接的集合用于图片.
  - *响应时间 = (M+1)O/R + (M/X + 1)2RTT + 空闲时间总和*

HTTP 响应时间 (秒)

RTT = 100 msec, O = 5 Kbytes, M=10 and X=5

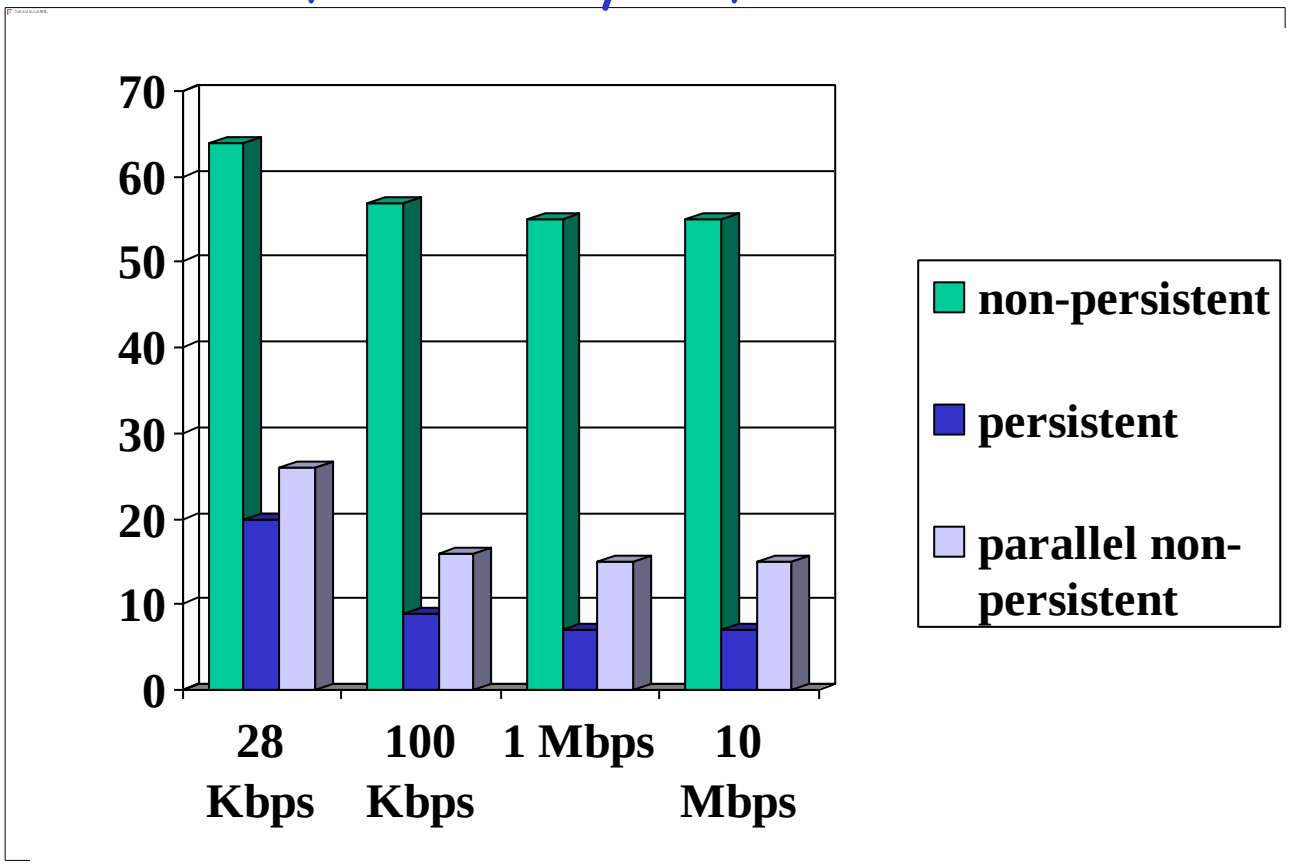


对于窄带，连结和响应时间由传输时间确定

持久连接仅比并行连接有少许改进.

HTTP 响应时间 (秒)

RTT =1 sec, O = 5 Kbytes, M=10 and X=5



对较大的 RTT, 响应时间是TCP建立连接和慢启动时延决定。

持久连接此时则能有重大改进: 特别对于高时延宽带积的网络

第三章 小结

- 运输层服务依据的原则:
  - 多路复用与多路分解
  - 可靠数据传送
  - 流量控制
  - 拥塞控制
  - 连接管理
- 因特网中的实例和实现
  - UDP
  - TCP

下一章:

- 离开网络的 “边缘” (应用层，运输层)
- 进入网络的“核心”

第三章 复习大纲

- 传输层提供的服务
  - 进程通信
  - 面向连接和无连接（是否建立连接）
  - 可靠传输实现原理
- UDP协议特性
- 校验和的实现思想
- TCP协议特性及其实现
  - TCP报文，固定报头为20字节
  - 连接管理
  - 可靠传输
  - 流量控制
  - 拥塞控制