计 算 机 网 络

西北工业大学 软件与微电子学院

计算机网络

第3章 运输层

TCP/IP五层模型

- 应用层：包含大量应用普遍需要的协议，支持网络应用
  - FTP, SMTP, HTTP
- 运输层：主机到主机数据传输，负责从应用层接收消息，并传输应用层的message，到达目的后将消息上交应用。
  - TCP, UDP
- 网络层：从源到目的地数据报的选路
  - IP, 选路协议
- 链路层：在邻近网元之间传输数据
  - PPP, 以太网
- 物理层：物理层负责将链路层帧中每一位(bit)从链路的一端传输到另一端。

第3章:运输层

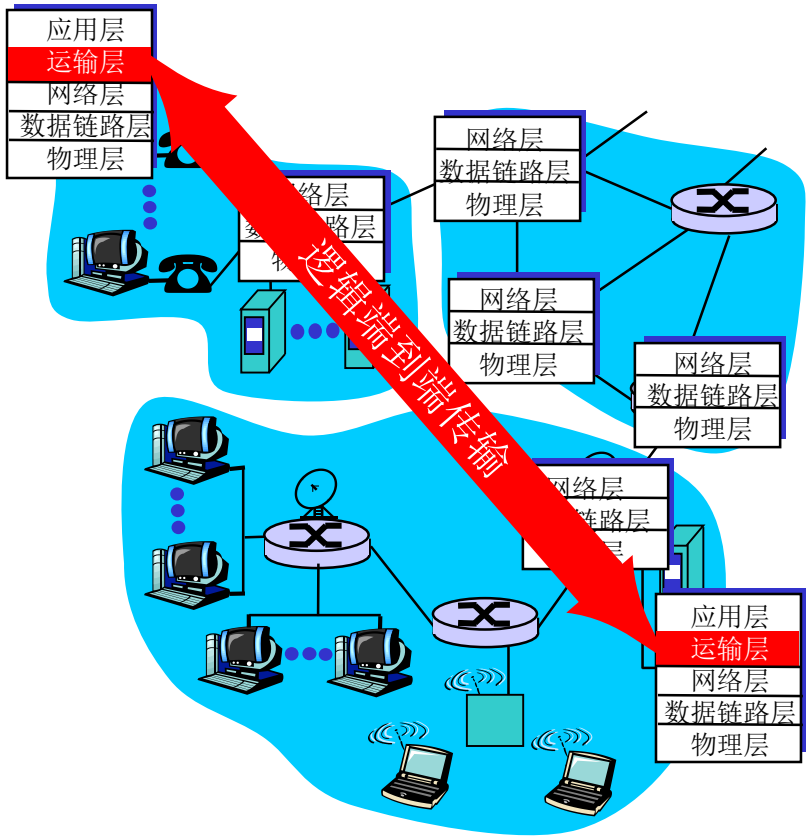
我们的目的:

- 理解运输层服务依据的原理:
  - Multiplexing(多路复用)/demultiplexing（多路分解）
  - 可靠数据传输
  - flow control（流量控制)
  - congestion control（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

- **3.1 运输层服务**
- 3.2 复用与分解
- 3.3 无连接传输：UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的传输：TCP
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制

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（messages）划分为报文段（segments），传向网络层
  - 接收方：将段重新装配为报文，传向应用层
- 应用程序可供使用的运输协议不止一个
  - 因特网：TCP和UDP



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

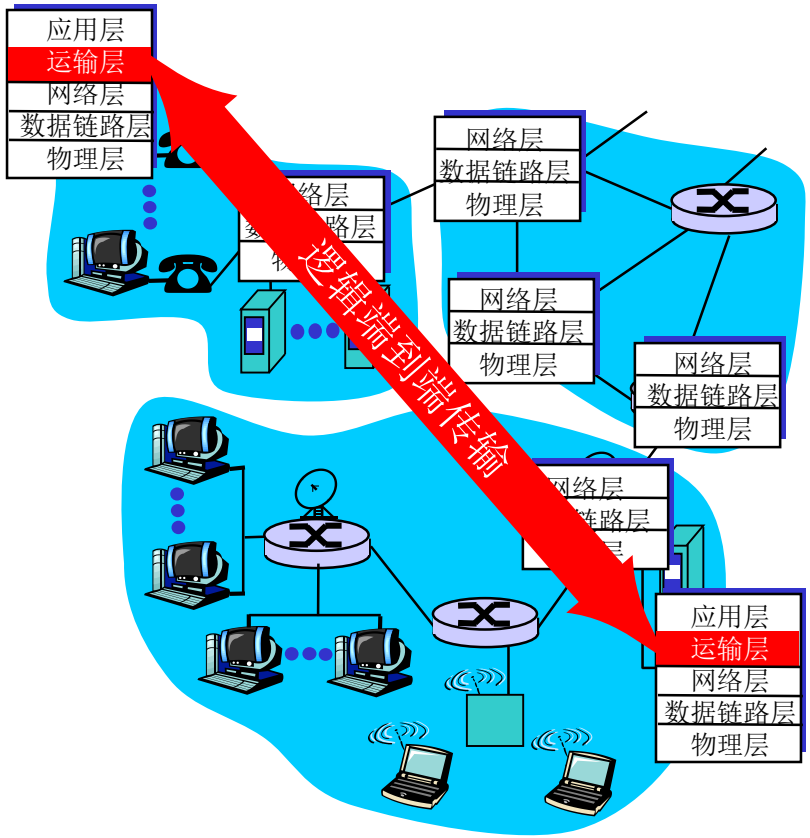
家庭类比：

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

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、无序的交付：UDP
  - 差错检测
- 不可用的服务：
  - 时延保证
  - 带宽保证



- 3.1 运输层服务
- **3.2 复用与分解**
- 3.3 无连接传输：UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的传输：TCP
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制

多路复用/多路分解

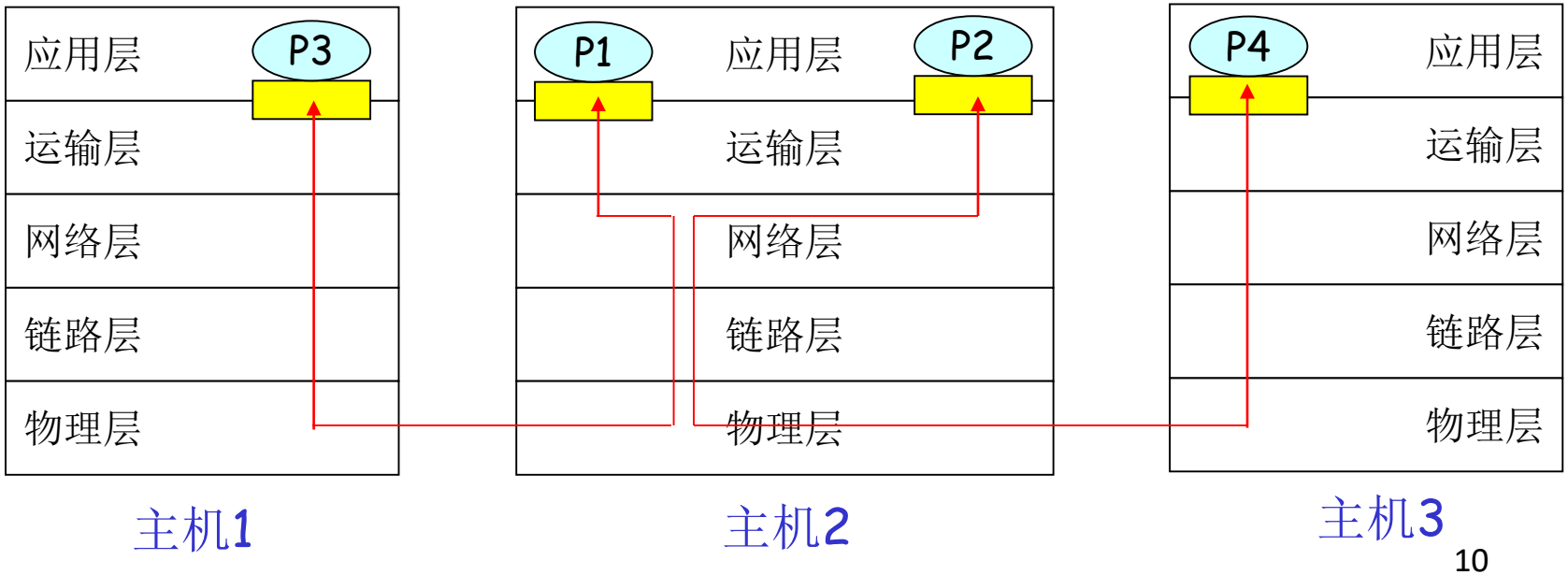
在接收主机分解：

将接收到的段交付给相应的套接字（一路到多路，向上）

在发送主机复用:

从多个套接字收集数据，用首部封装数据（多路到一路，向下）

= 套接字　　= 进程



分解工作过程

- 主机接收IP数据报
  - 每个数据报有源IP地址，目的IP地址
  - 每个数据报承载1个运输层报文段
  - 每个报文段具有源、目的端口号

TCP/UDP 报文段格式

- 主机使用IP地址 &端口号将报文段导向到相应的套接字

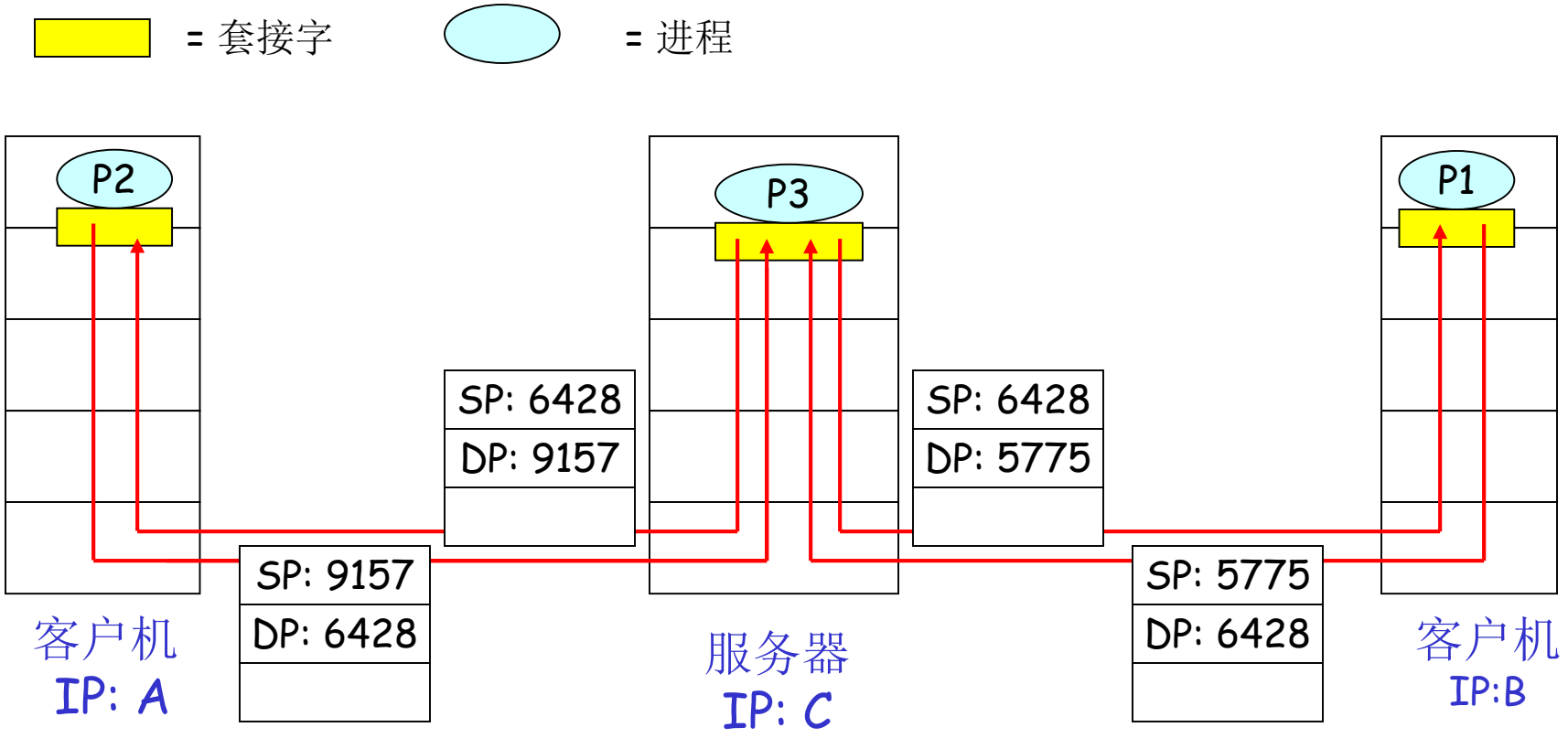
无连接多路复用与分解

- UDP套接字由二元组全面标识 ：

  （目的地IP地址，目的地端口号）

- 当主机接收UDP段时：
  - 在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
- 具有不同源IP地址和/或源端口号的IP数据报（目的IP地址和端口号相同）定向到相同的套接字

无连接多路复用与分解(续)

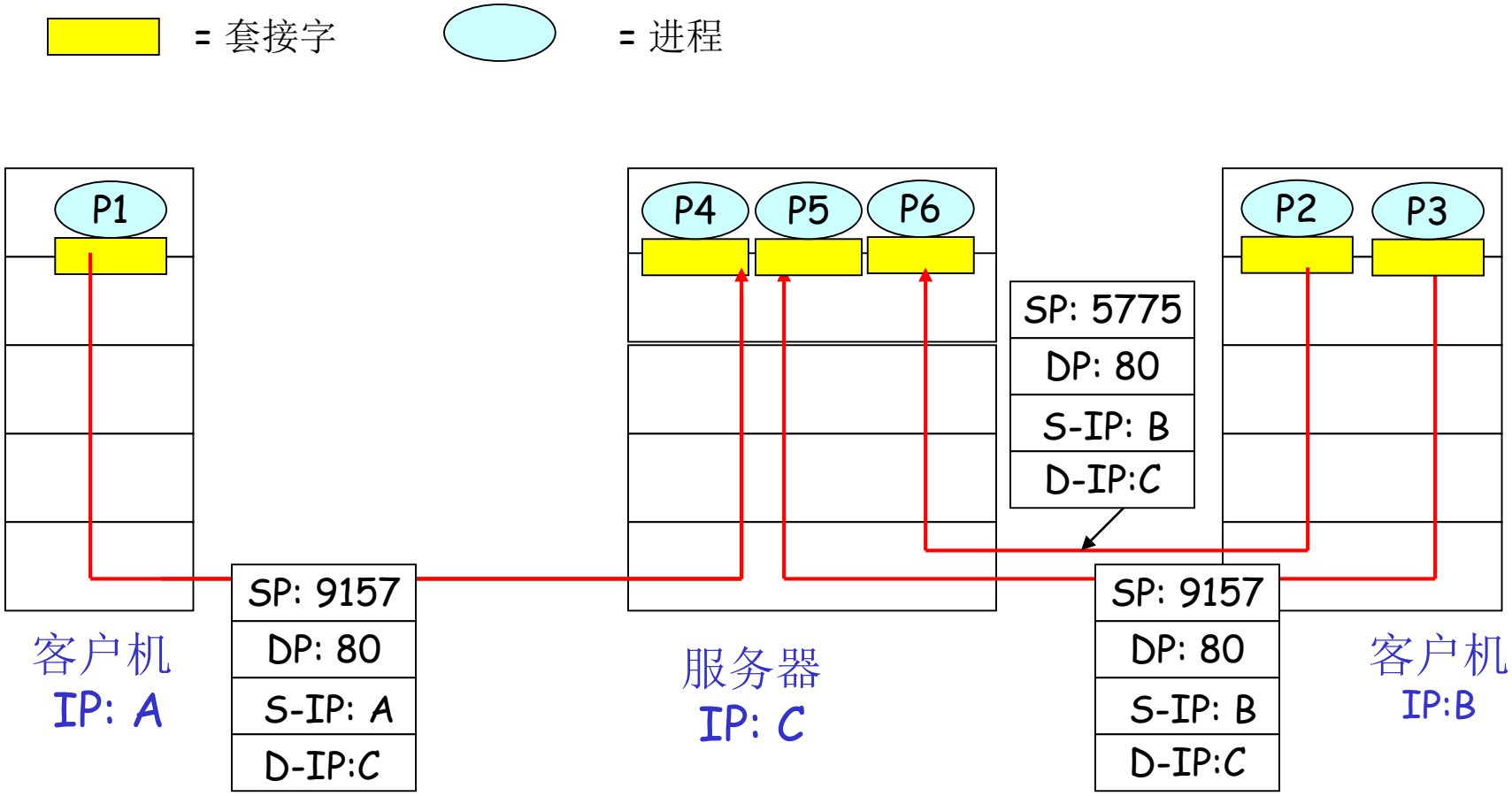


SP提供了“返回地址” How?

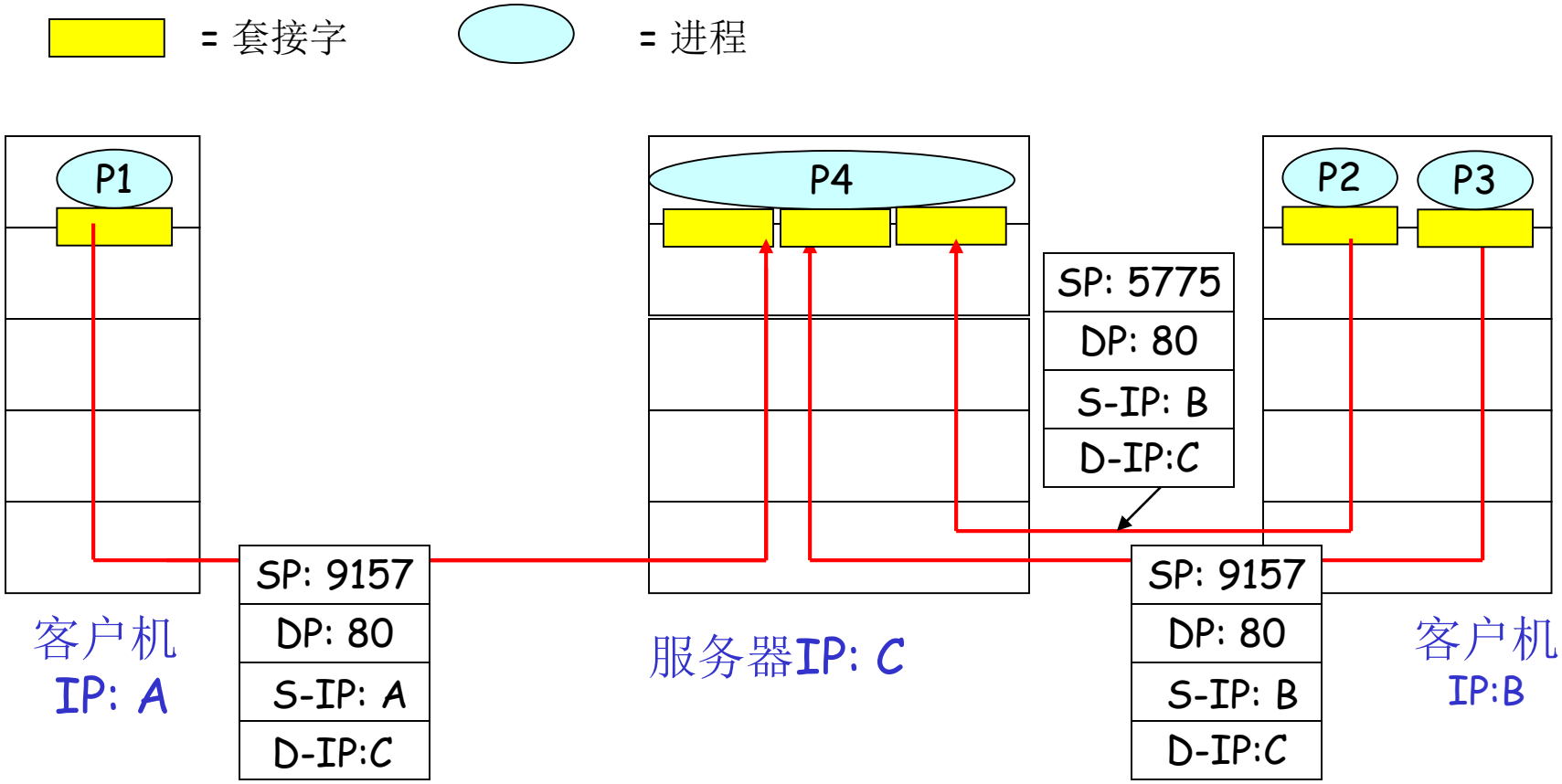
面向连接多路复用与分解

- TCP套接字由四元组（4-tuple）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
- Web服务器对每个连接的客户机具有不同的套接字

面向连接多路复用与分解(续)



面向连接分解：多线程Web服务器



I wonder which is a better choice for implementing tabs in a web browser? (Ex: Firefox use multi-threading for their tabs, while Google Chrome use multi-processes ...)

It depends on what your needs are.
If you are

- Looking for a resource minimal browser
- Not running heavy javascripts
- Not having a lot of memory

Then you are probably going to go with multi threading.
However, if you are
Running many pages at once
Running resource intensive web applications
You probably are going to want multi-process. Since most computers and web applications fit in the second category today, Multi-Process is probably, today the better choice. This is because processes allow you to separate each tab in its own sandbox. That means if one tab crashes, you aren't going to lose the other tabs.

- 3.1 运输层服务
- 3.2 复用与分解
- **3.3 无连接传输：UDP**
- 3.4 可靠数据传输的原理
- 3.5 面向连接的传输：TCP
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制

UDP：用户数据报协议

- “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

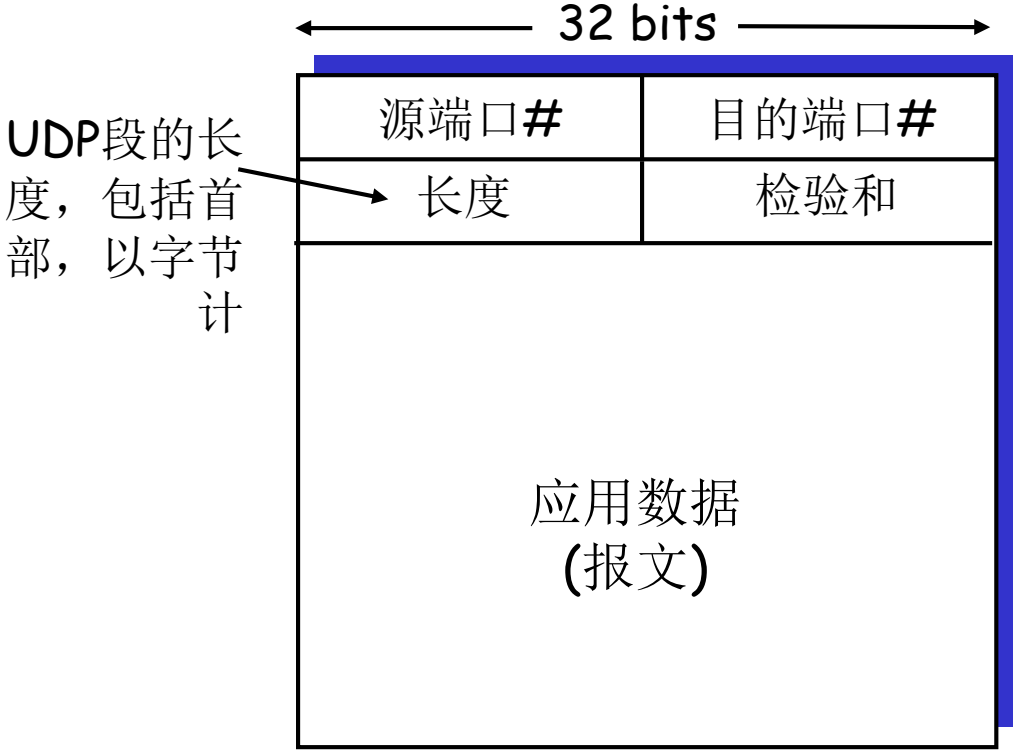
为何要有 UDP协议?

- 无连接创建(时延小)
- 头部开销少：8B. vs 20B
- 无拥塞控制：UDP能够尽可能快地传输

简单快捷

UDP报文段结构

- 常用于流媒体应用程序
  - 丢包容忍
  - 速率敏感
- 其他UDP应用
  - DNS
  - SNMP（简单网络管理协议）
- 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

checksum ： 校验和 ， 检验和

UDP检验和

目的: 在传输的段中检测“差错” (如比特翻转)

发送方:

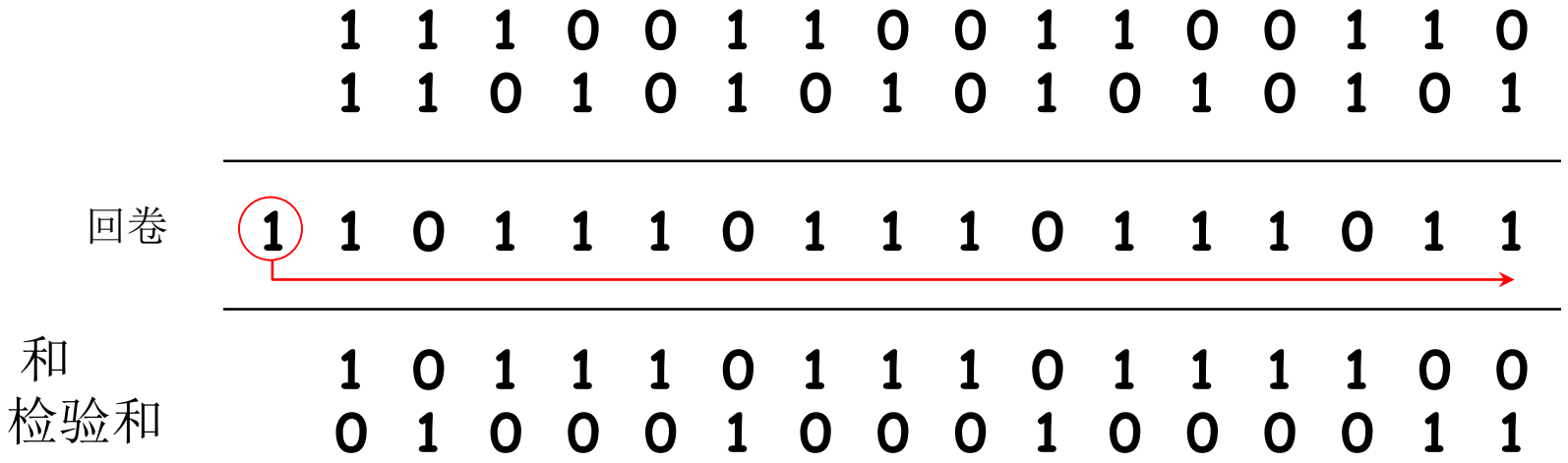
- 将段内容处理为16比特整数序列
- 检验和：段内容的加法（反码和）
- 发送方将检验和放入UDP检验和字段

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验和字段的值:
  - NO – 检测到差错
  - YES – 无差错检测到。

检验和例子

- 注意
  - 当数字作加法时，最高位进比特位的进位需要加到结果中
- 例子：两个16-bit整数相加



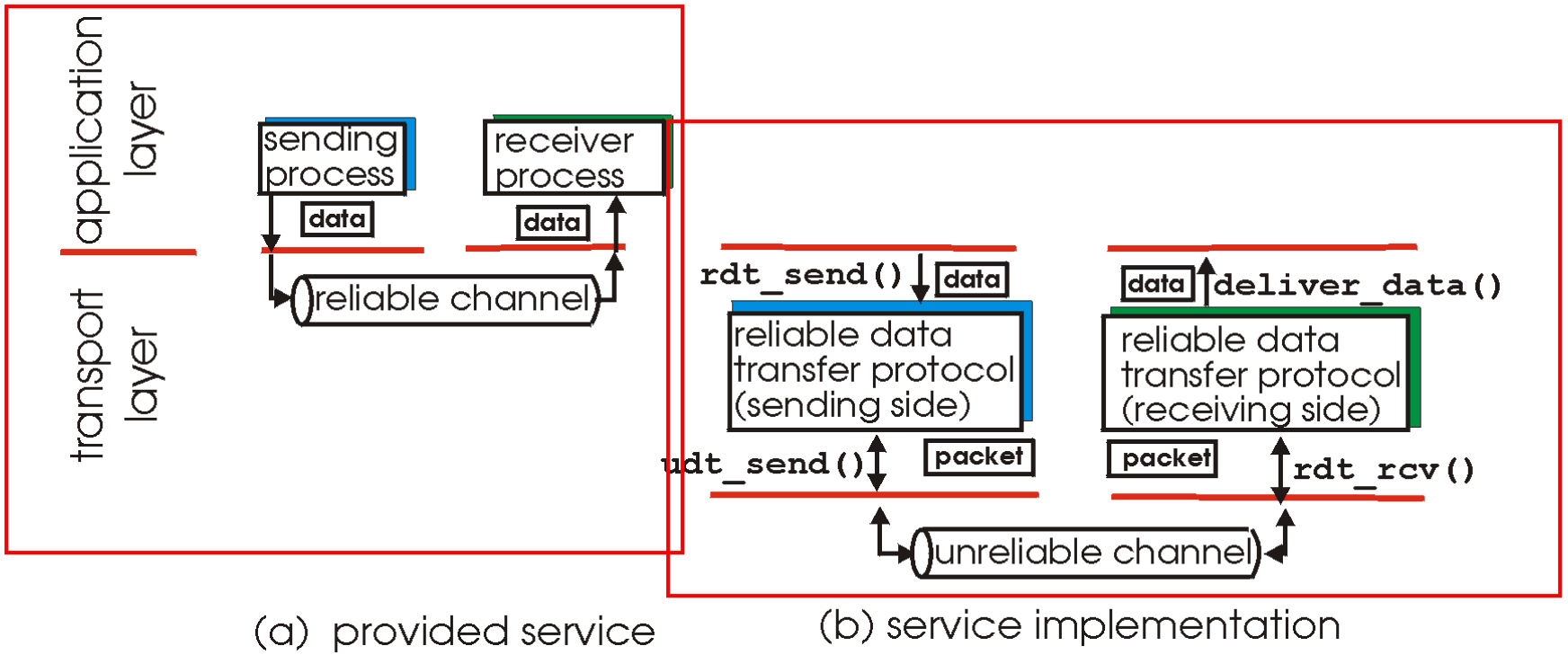
计算步骤: 求和，回卷，求反

检验结果正确，就能保证无错?

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输：UDP
- **3.4 可靠数据传输的原理**
- 3.5 面向连接的传输
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制

可靠数据传输的原理

- 在应用层、运输层、数据链路层的重要性
  - 网络中需解决的最重要的10个问题之一!
  - 不错、不丢、不乱

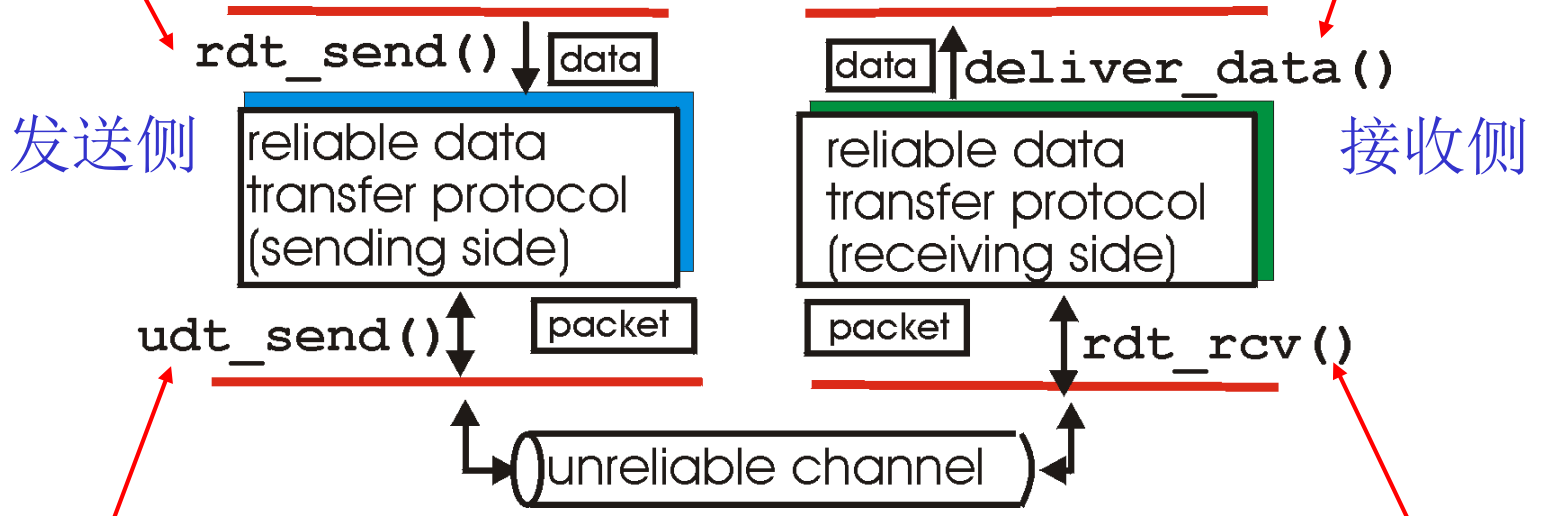


- 不可靠信道的特点决定了可靠数据传输协议的复杂性

可靠数据传输：描述函数熟悉

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



udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

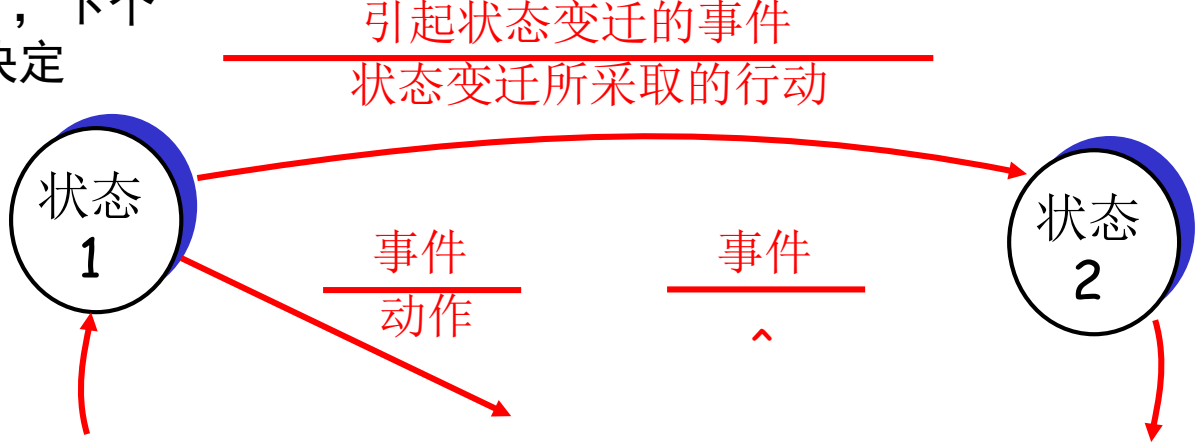
rdt_rcv(): called when packet arrives on rcv-side of channel

有限状态机描述方法

我们将:

- 循序渐进地研究可靠数据传输协议（rdt）的发送方和接收方
  - 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

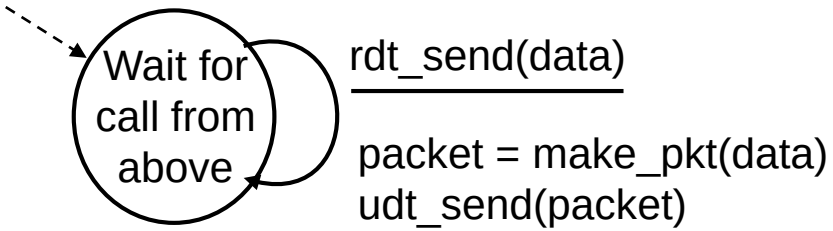
状态: 当位于这个“状态时”，下个状态惟一地由下个事件决定



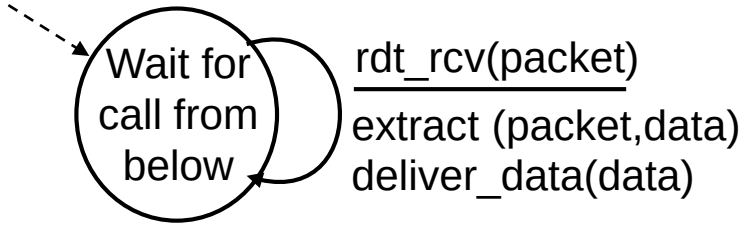
- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输：UDP
- **3.4 可靠数据传输的原理**
  - **rdt1.0，rdt2.0，rdt3.0协议**
- 3.5 面向连接的传输
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制

rdt1.0:完全可靠信道上的可靠数据传输

- 底层信道完全可靠
  - 无比特差错
  - 无分组丢失
- 发送方、接收方具有各自的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

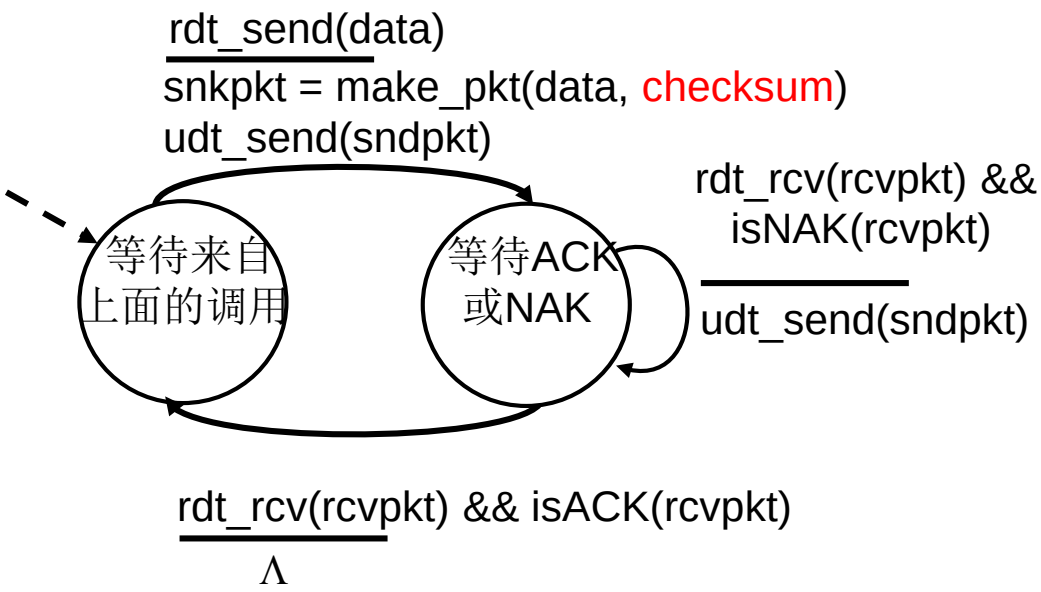


接收方

Rdt2.0：具有比特差错的信道

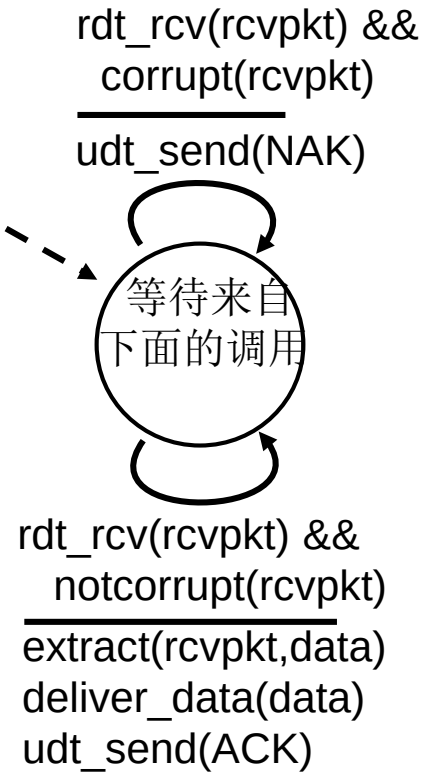
- 具有比特差错的底层信道
  - 有比特差错
- rdt2.0新增加机制（与rdt1.0比较）
  - 检错：Checksum
  - 反馈：ACK, NAK
  - 重传：ARQ
- 数据出错后处理方式
  - 检错重传

rdt2.0: FSM描述



发送方

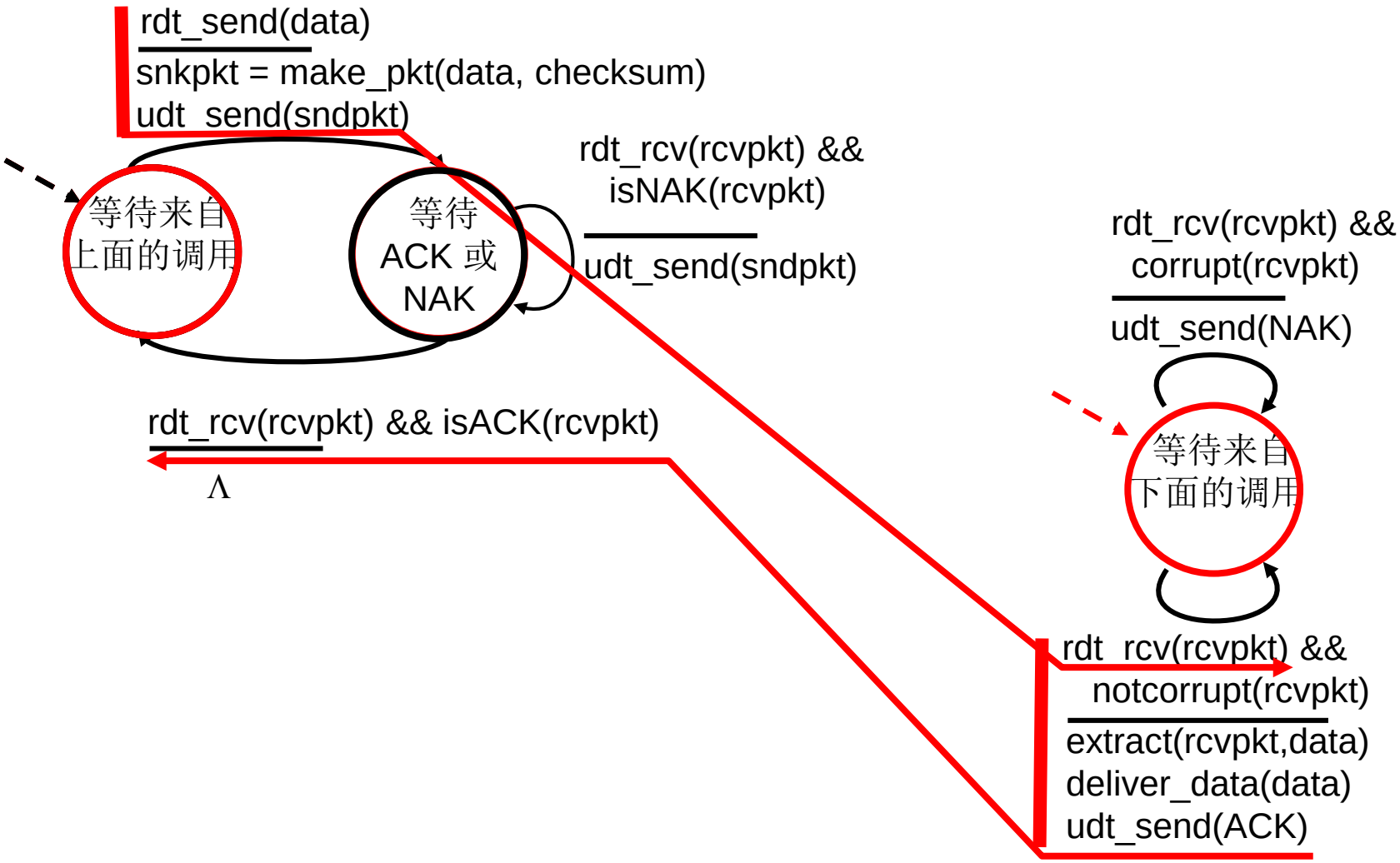
接收方



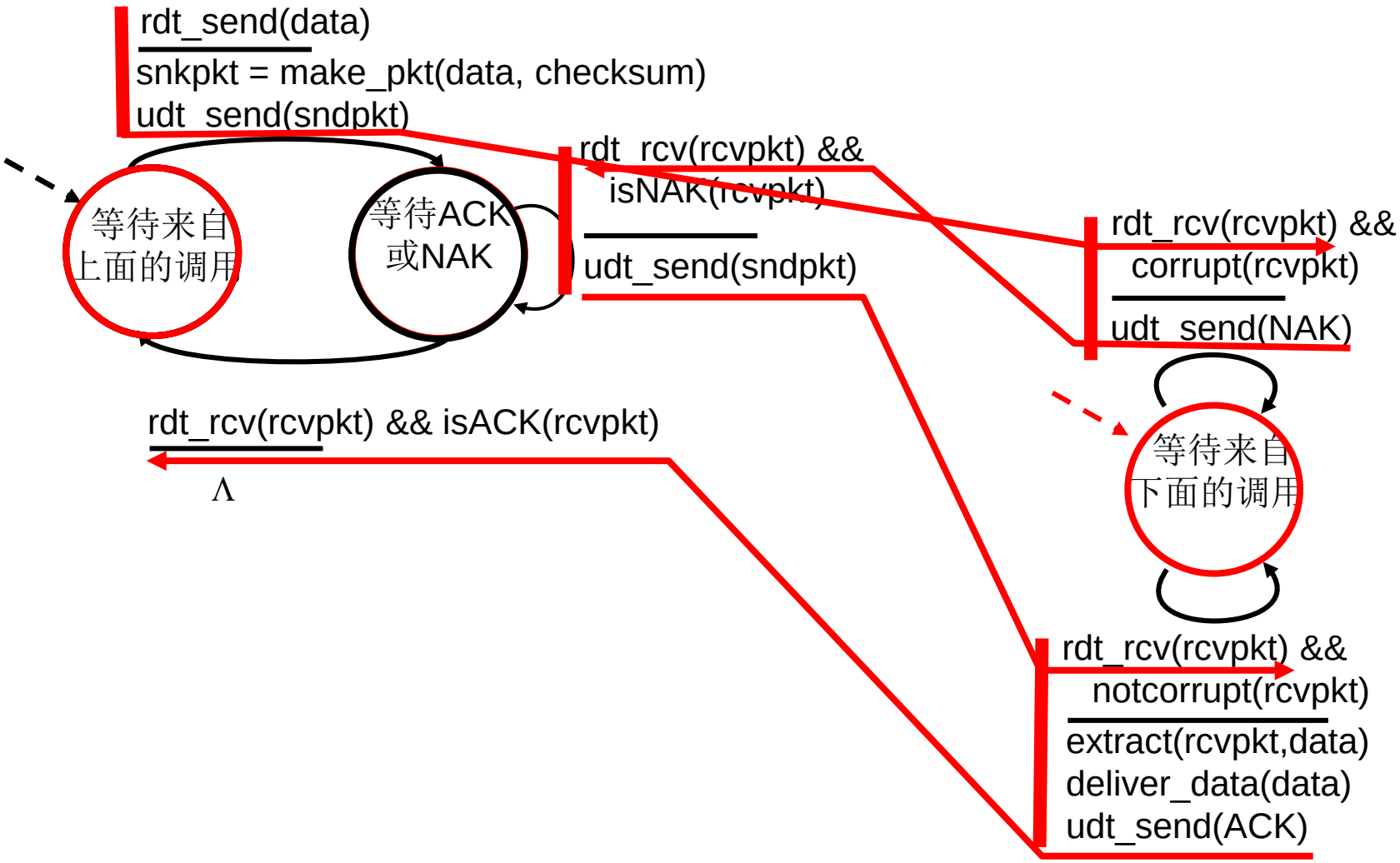
停等协议

发送方发出1个分组，等待接收方响应后再继续发送。

rdt2.0：无差错时的操作



rdt2.0：有差错时的情况



rdt2.0有重大的缺陷!

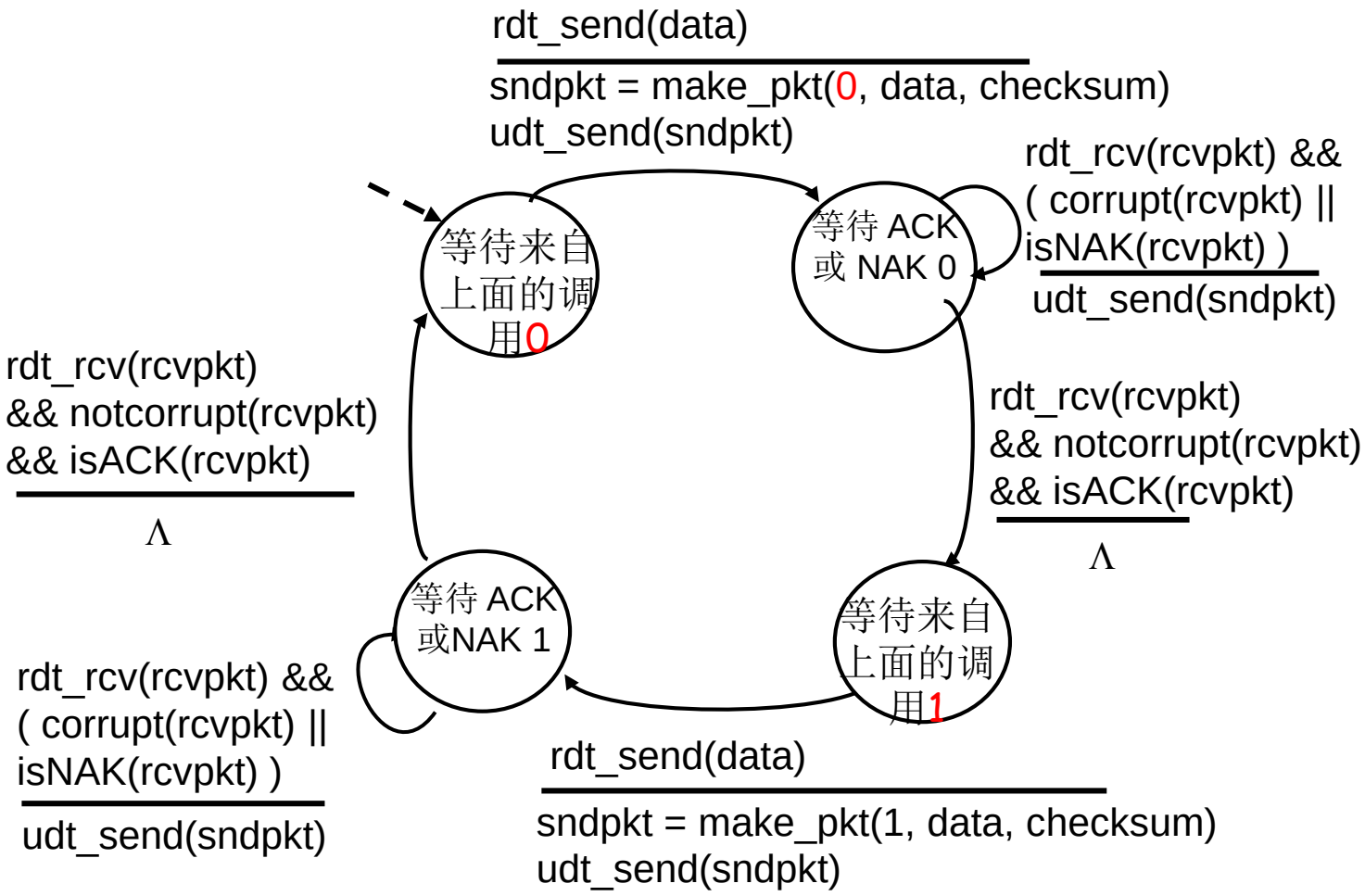
如果ACK/NAK受损，将会出现何种情况？

- 发送方不知道在接收方会发生什么情况!
- 不能只是重传：可能导致重复（ duplicate ）

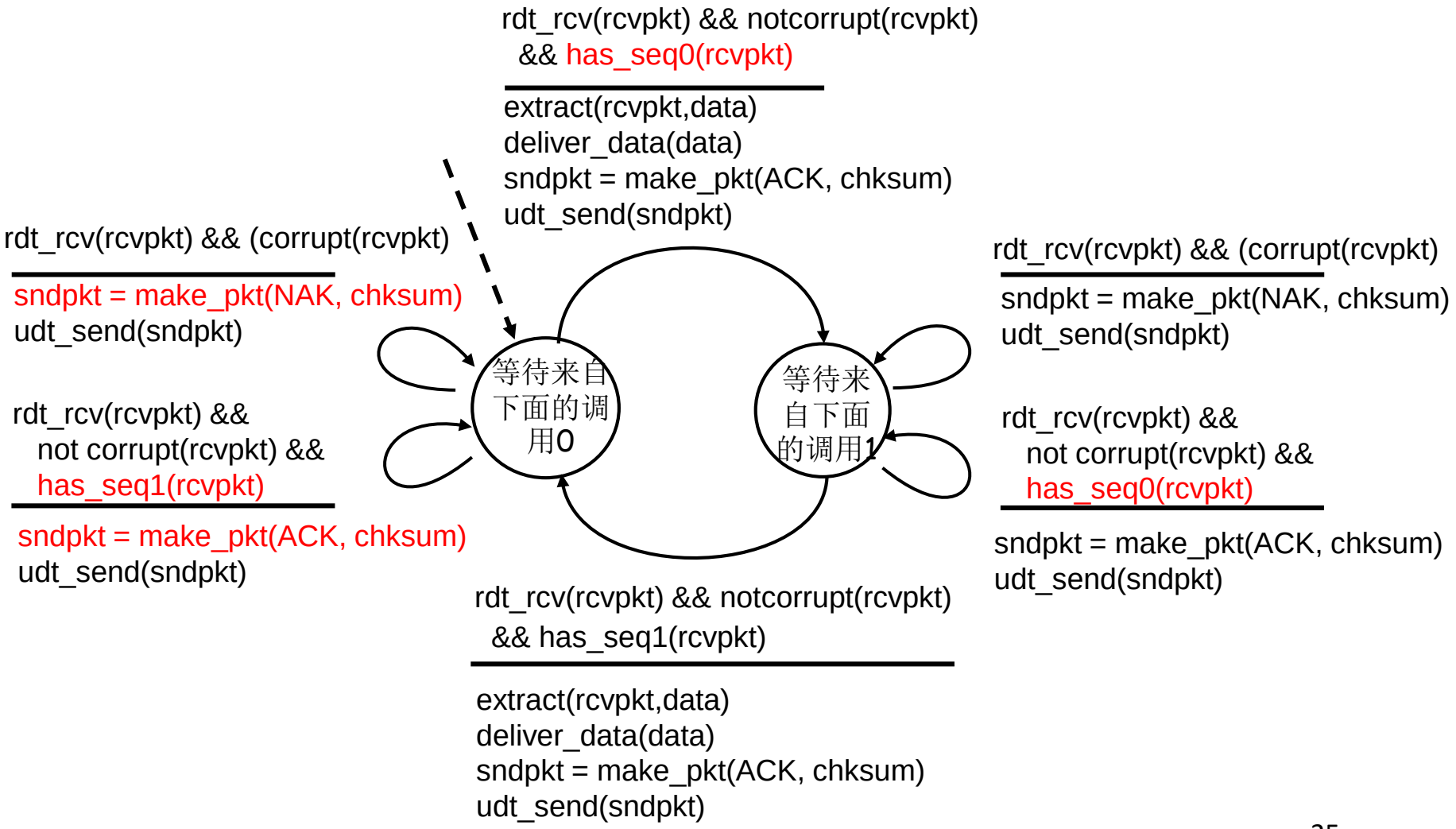
处理重复（序号机制）：

- 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)重复的分组

rdt2.1：发送方，处理受损的ACK/NAK



rdt2.1：接收方，处理受损的ACK/NAK



rdt2.1: 讨论

发送方:

- 序号加入分组中
- 两个序号（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

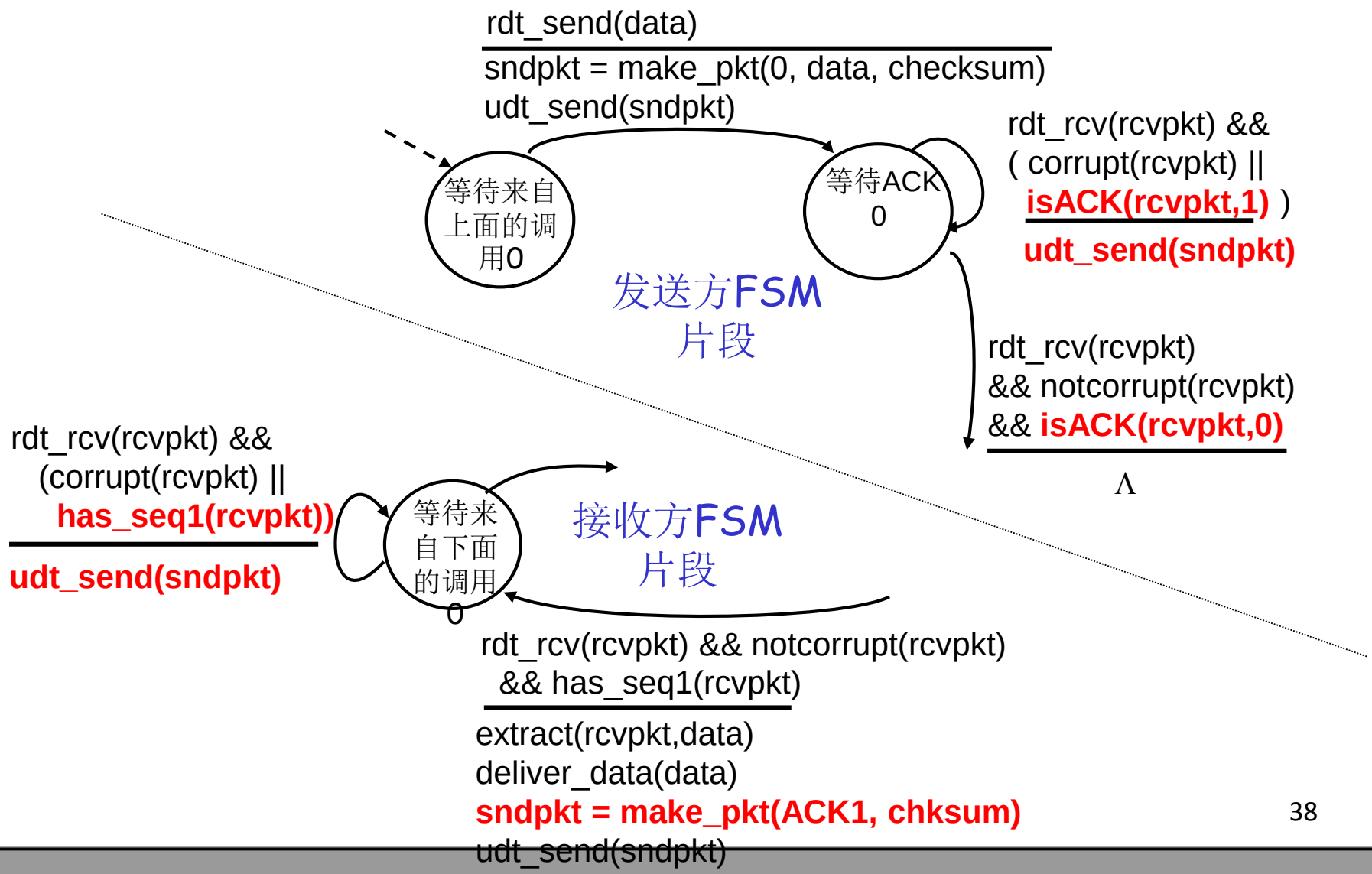
接收方:

- 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
  - 注意: 接收方不能知道它的最后的ACK/NAK在发送方是否接收OK

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
- 在发送方重复的ACK导致如同NAK相同的动作：*重传当前分组*

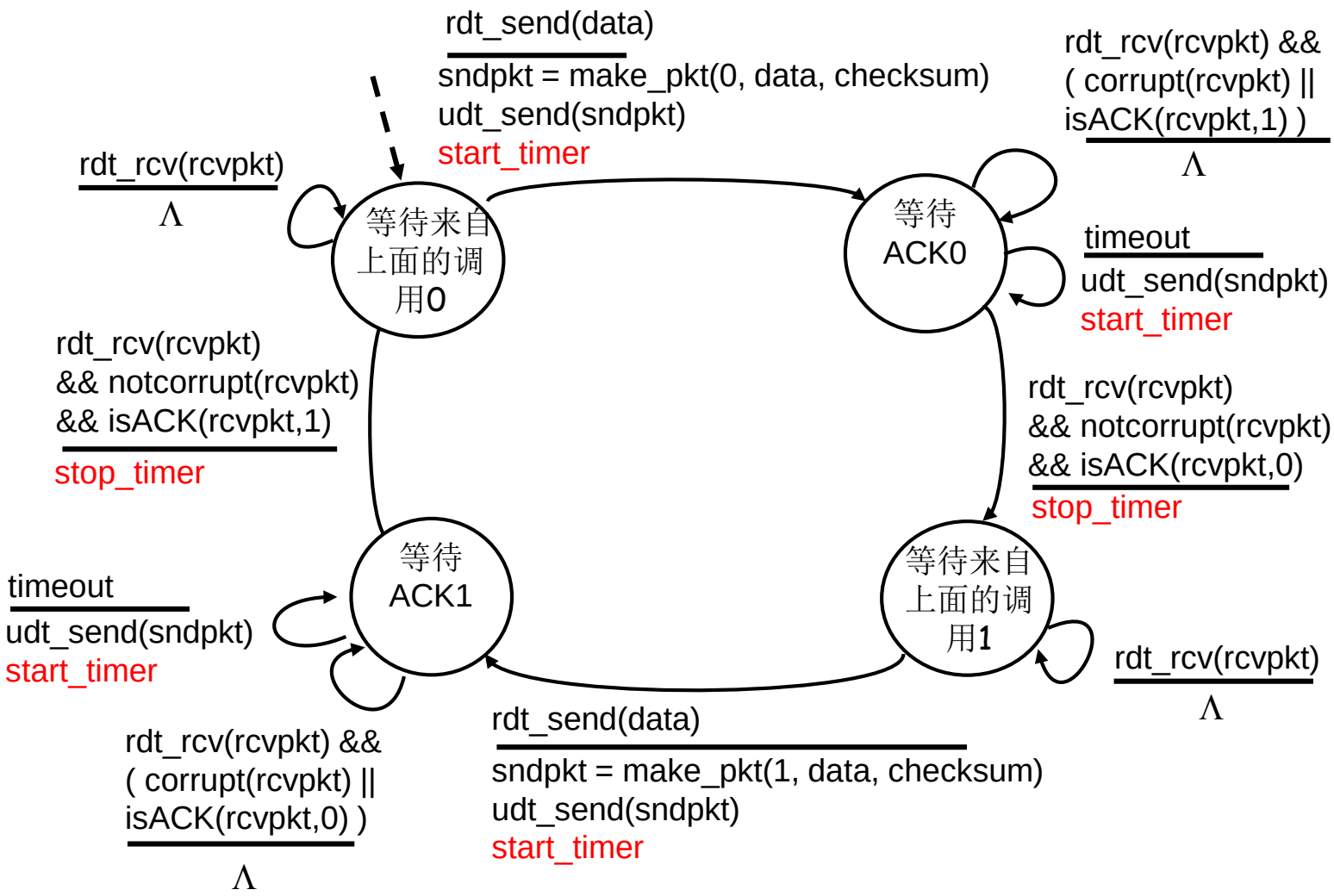
rdt2.2：发送方，接收方片段



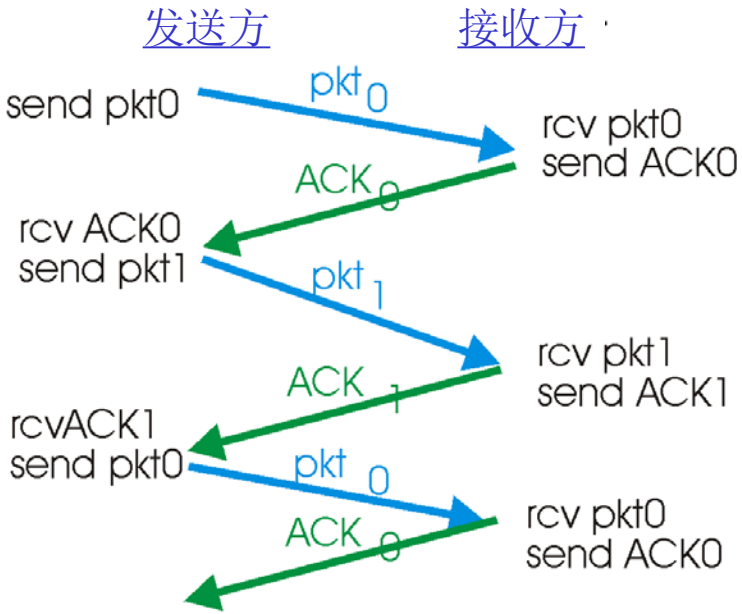
rdt3.0：具有差错和丢包的信道

- 具有差错和丢包的底层信道
  - 有比特差错
  - 有分组丢失
- 现有机制（检错、反馈、重传、序号）还不够
- 增加定时机制：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

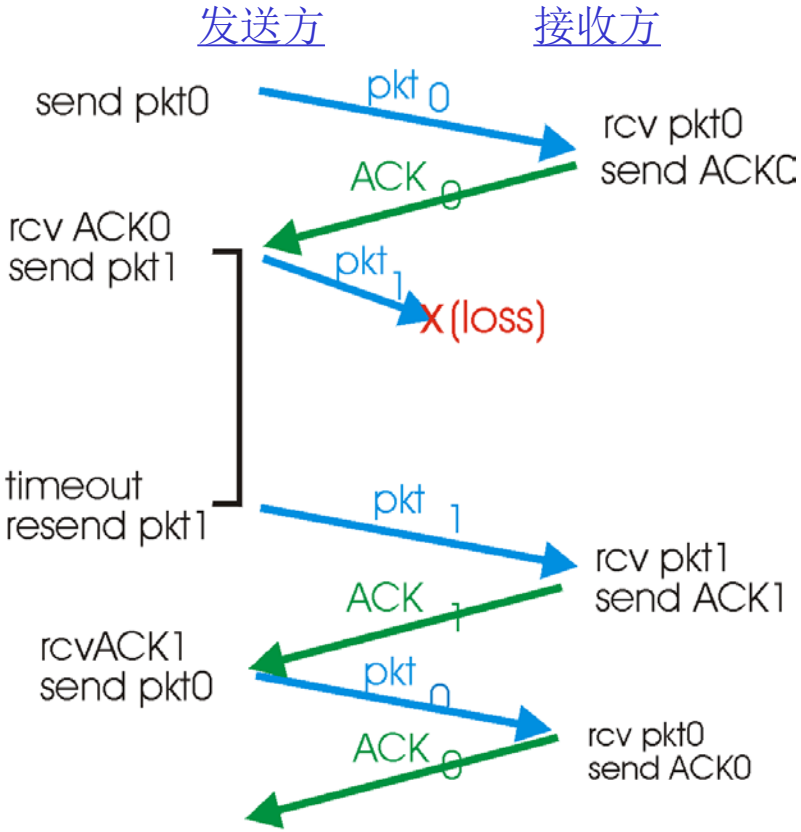
rdt3.0发送方


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

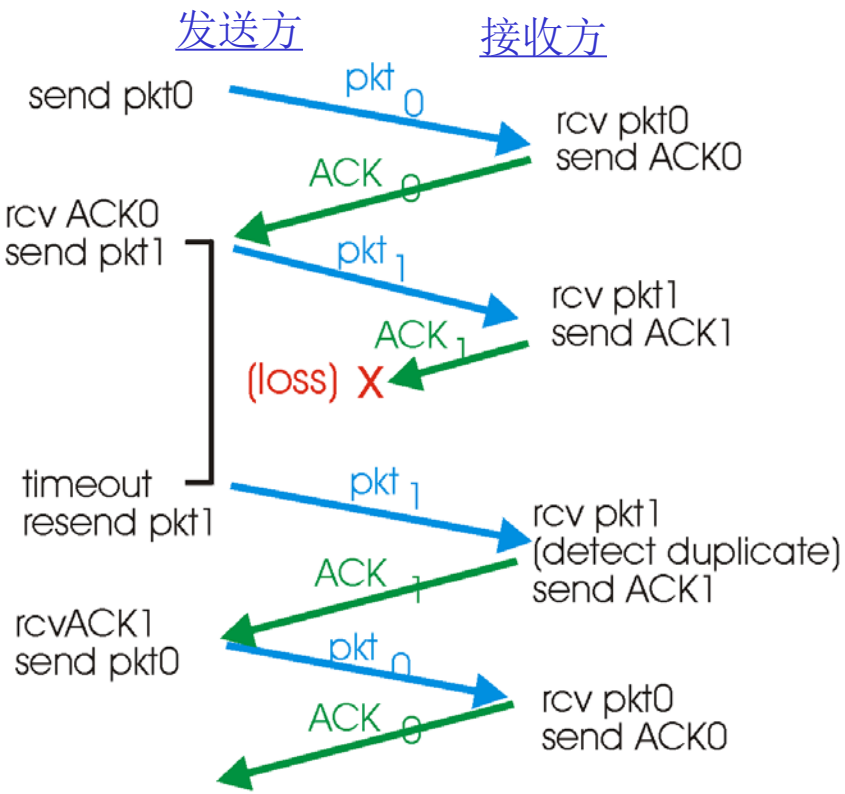


(a) 无丢包时的运行

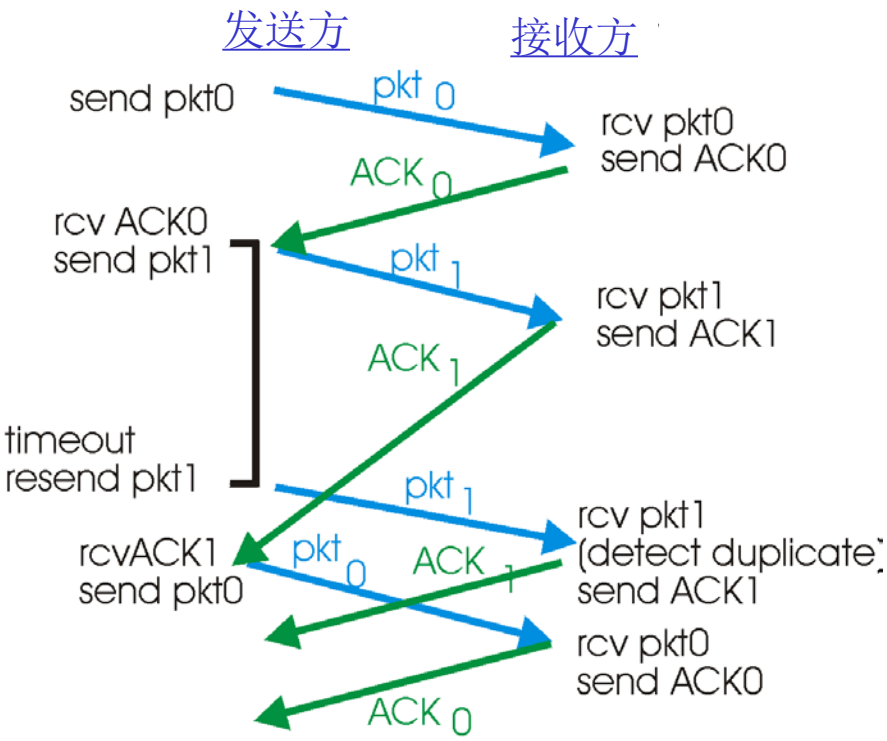


(b) 分组丢失

rdt3.0运行情况



(c) ACK丢失



(d) 过早超时

rdt3.0的性能

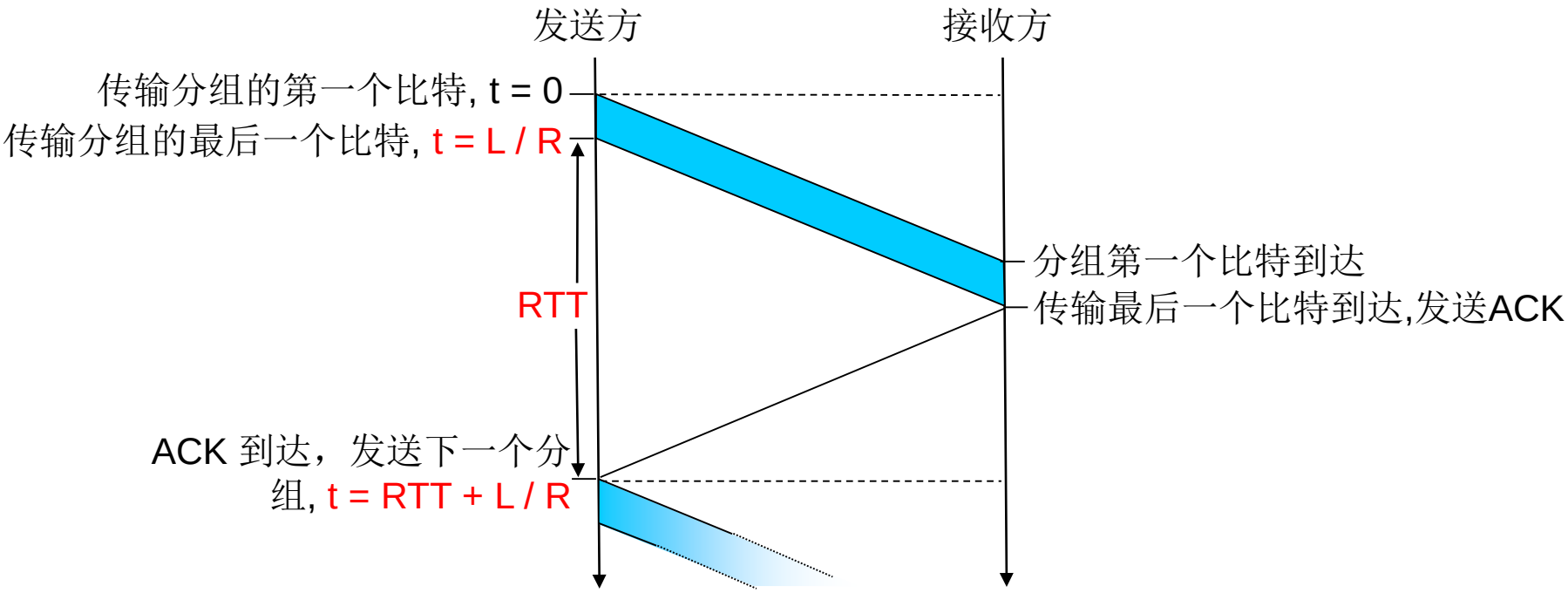
- rdt3.0能够工作，但性能不太好
- 例子：1 Gbps链路，15 ms端到端传播时延，1KB分组：

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8kb/pkt}{10^9 \text{ b/sec}} = 0.008 \text{ microsec}$$

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

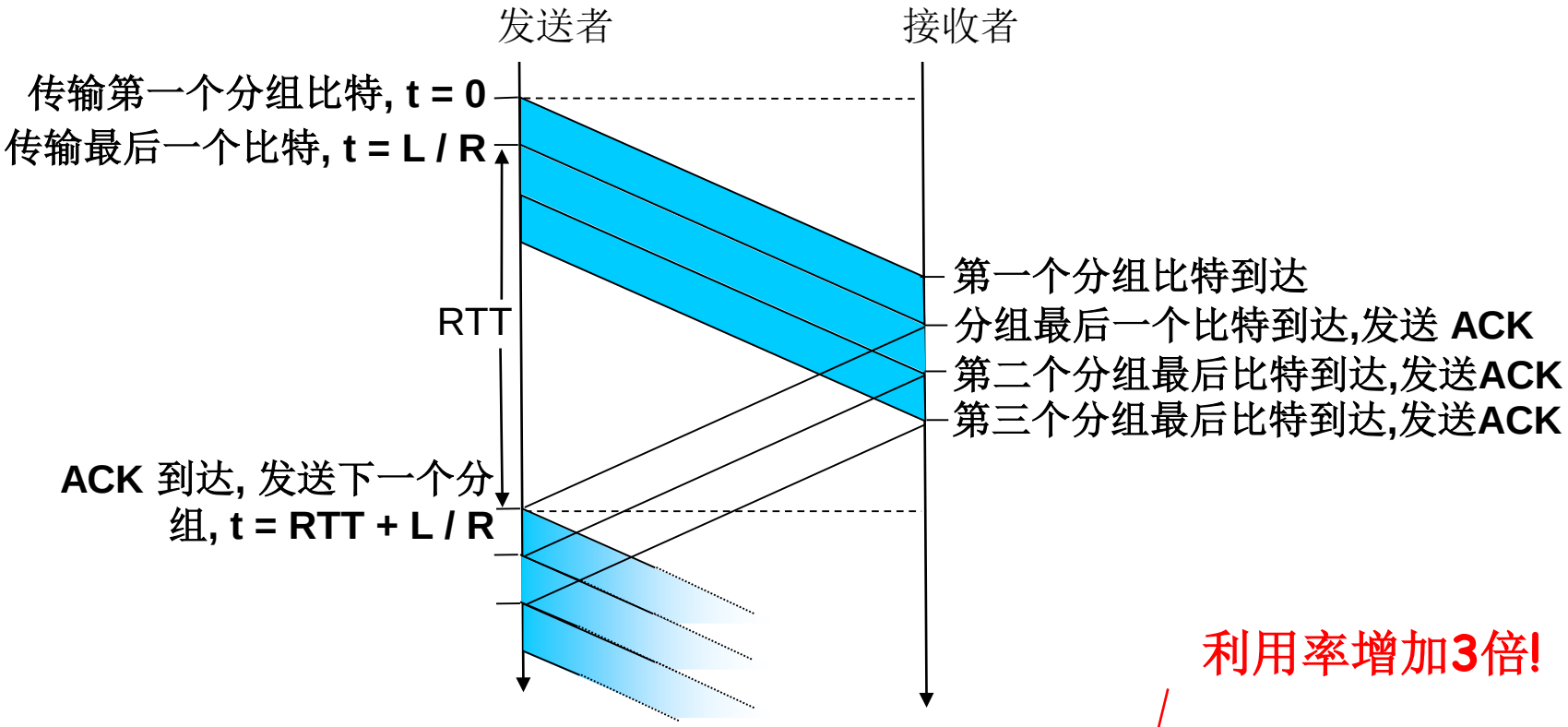
- U_{sender}: 利用率 – 发送方用于发送时间的比率
- 每30 msec 1KB 分组 -> 经1 Gbps 链路有33kB/sec 吞吐量
- 网络协议限制了物理资源的使用!

rdt3.0：停等协议的运行



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

流水线协议：增加利用率



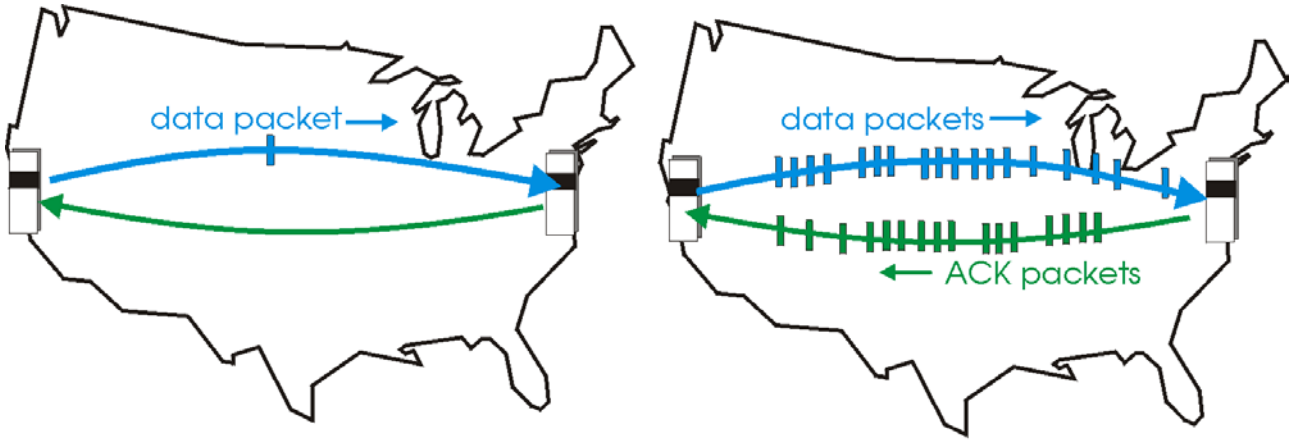
利用率增加3倍!

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

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

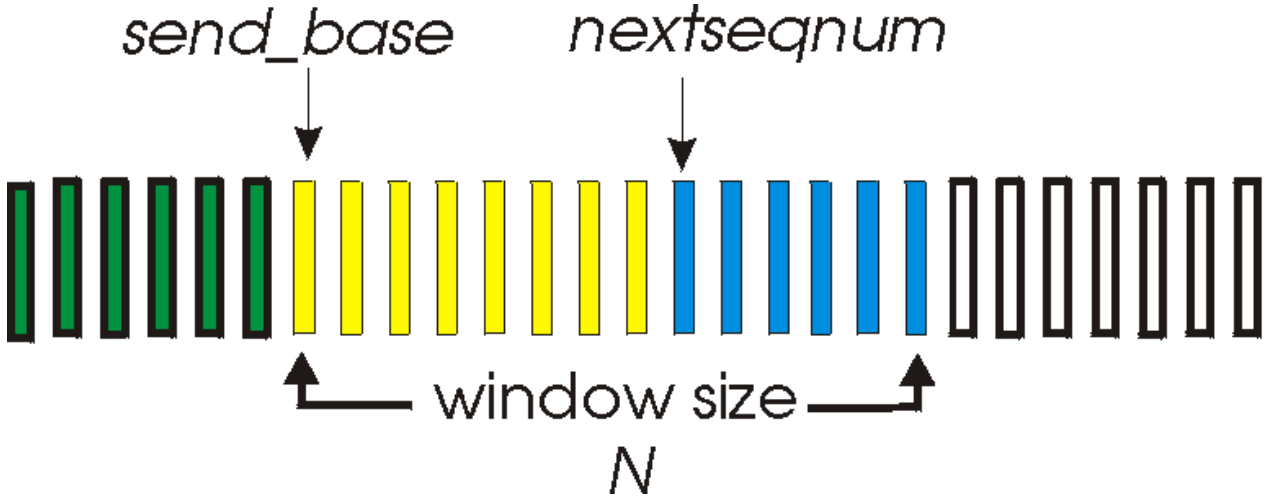
- 序号的范围必须增加
- 发送方和/或接收方设有缓冲



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

(b) a pipelined protocol in operation

滑动窗口协议



- 滑动窗口协议: Sliding-window protocol
- 窗口
  - 允许使用的序列号范围
  - 窗口尺寸为N：最多有N个等待确认的消息
- 滑动窗口
  - 随着协议的运行，窗口在序列号空间内向前滑动

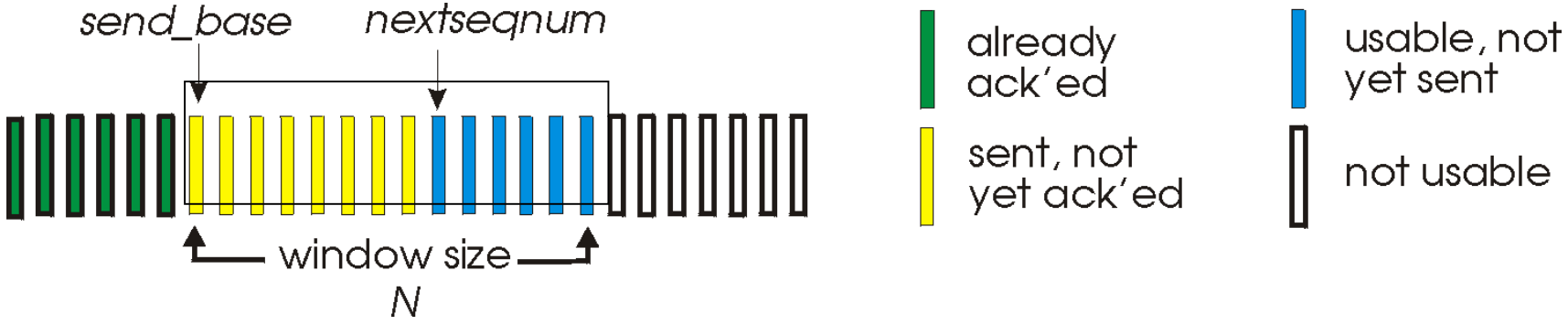
❖滑动窗口协议：GBN, SR

滑动窗口协议

Go-Back-N和SR都是滑动窗口协议

- 发送方和接收方都具有一定容量的缓冲区（即窗口），允许发送站连续发送多个帧而不需要等待应答
- 发送窗口就是发送端允许连续发送的分组的序号，发送端可以不等待应答而连续发送的最大分组数称为发送窗口的尺寸
- 接收窗口是接收方允许接收的分组的序号，凡落在接收窗口内的分组，接收方都必须处理，落在接收窗口外的分组被丢弃。接收方每次允许接收的分组数称为接收窗口的尺寸

Go-Back-N

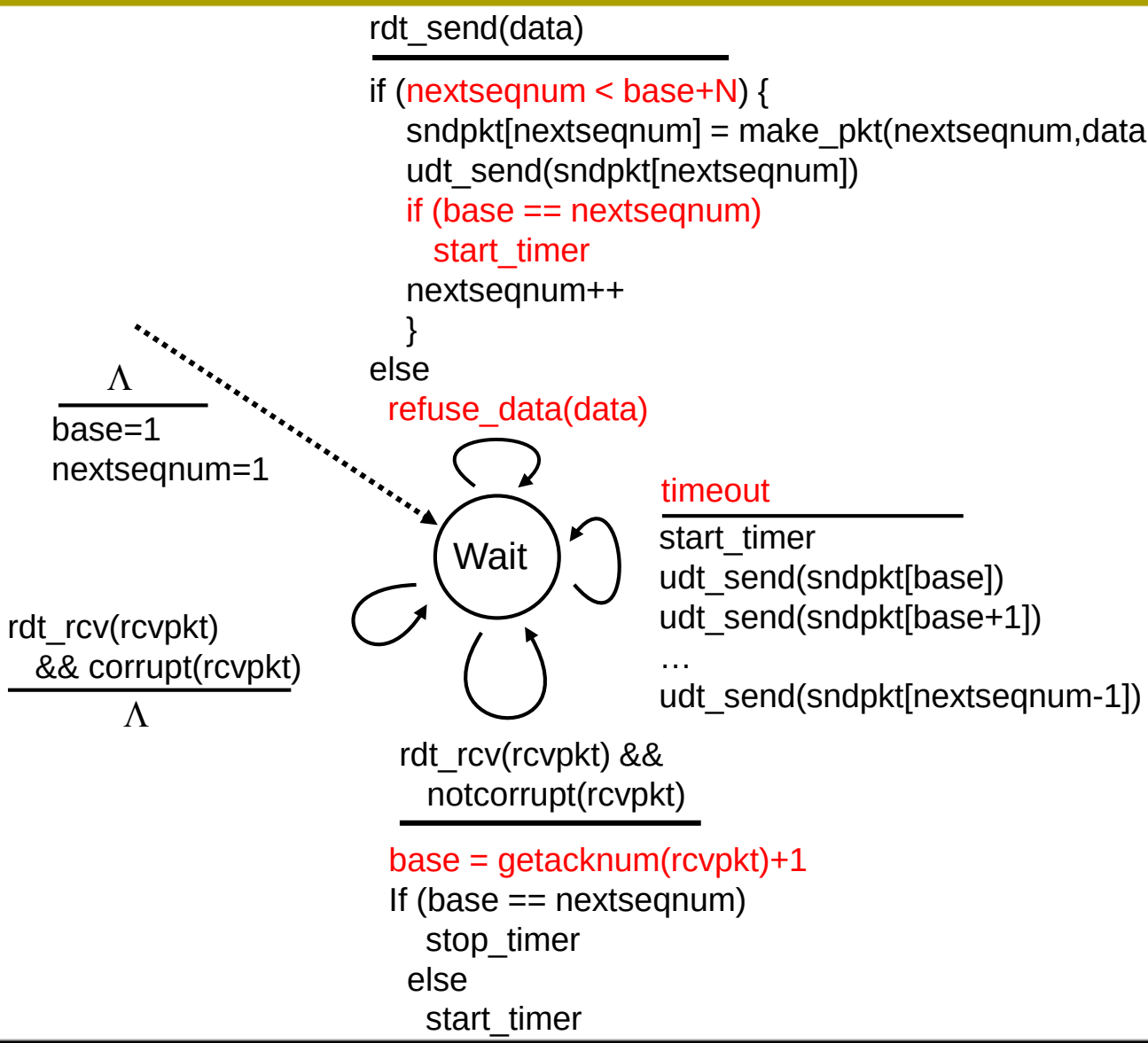


- 简单来说：位于发送窗口内的分组才允许被发送，位于接收窗口内的分组才能被接收，关键是窗口如何滑动。

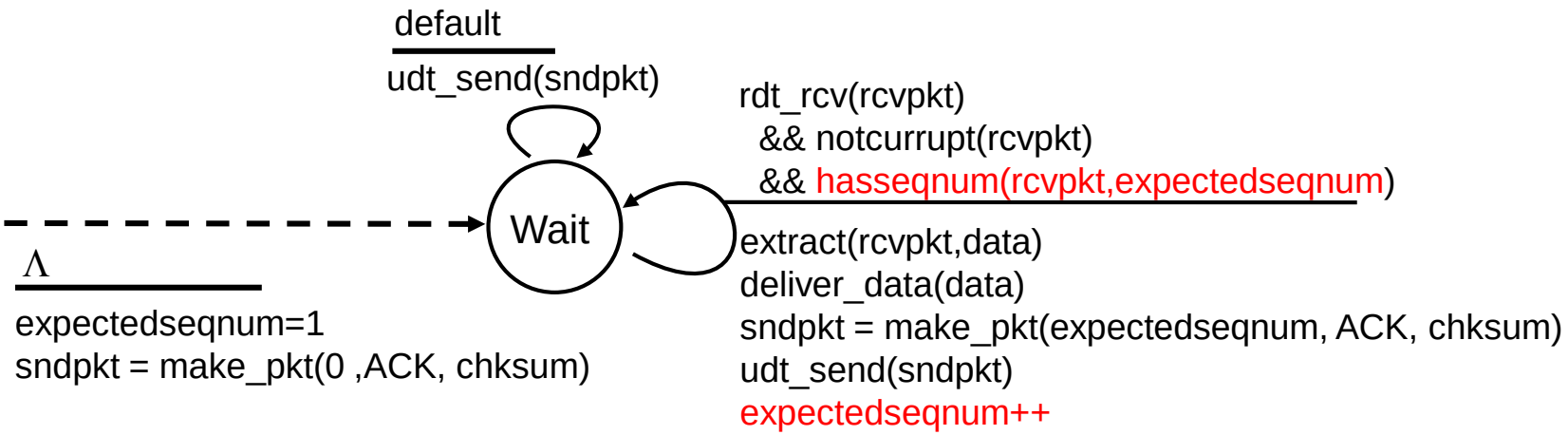
特征：累积ACK，全部重传

- ACK(n): 确认所有的（包括序号n）的分组 - “累积ACK”
- *若*超时，重传窗口中的未被确认的第一个分组n及所有更高序号的分组

GBN: sender extended FSM

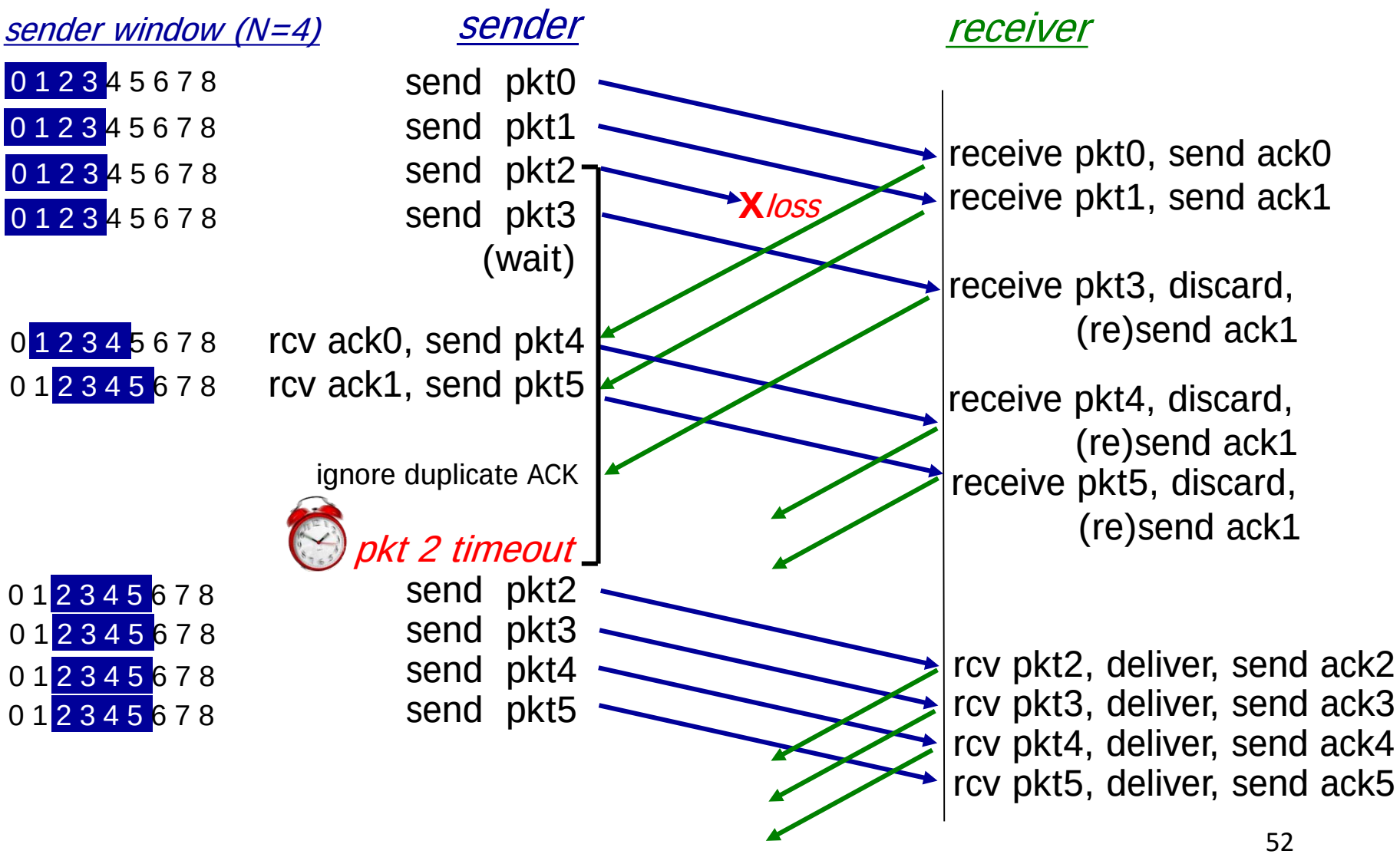


GBN: receiver extended FSM



- ACK机制：发送拥有最高序列号的、已被正确接收的分组的ACK
  - 可能产生重复ACK
  - 只需要记住唯一的**expectedseqnum**
- 乱序到达的分组：
  - 直接丢弃，接收方没有缓存
  - 重新确认序列号最大的、按序到达的分组

Go-Back-N传输示意



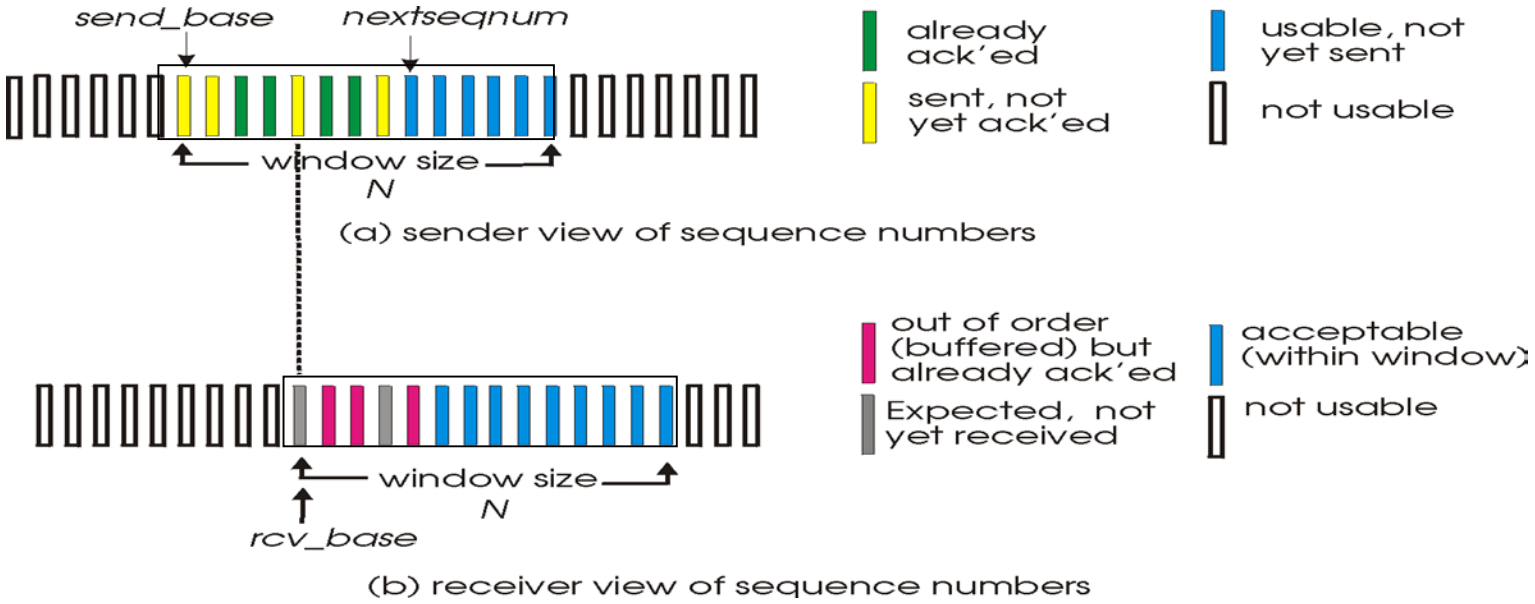
Go-Back-N

理解累积ACK和回退N个重传

- 发送方
  - 发送窗口滑动的条件：收到1个确认分组
  - 超时重传时，回退N个重传，通常重传多个分组
- 接收方
  - 接收窗口滑动的条件：收到期望序号的分组
  - 对失序分组的处理：丢弃，重发（已按序接收分组的）ACK
- Go-Back-N不足：（效率明显高于停等协议）但仍有不必要重传的问题

选择重传SR

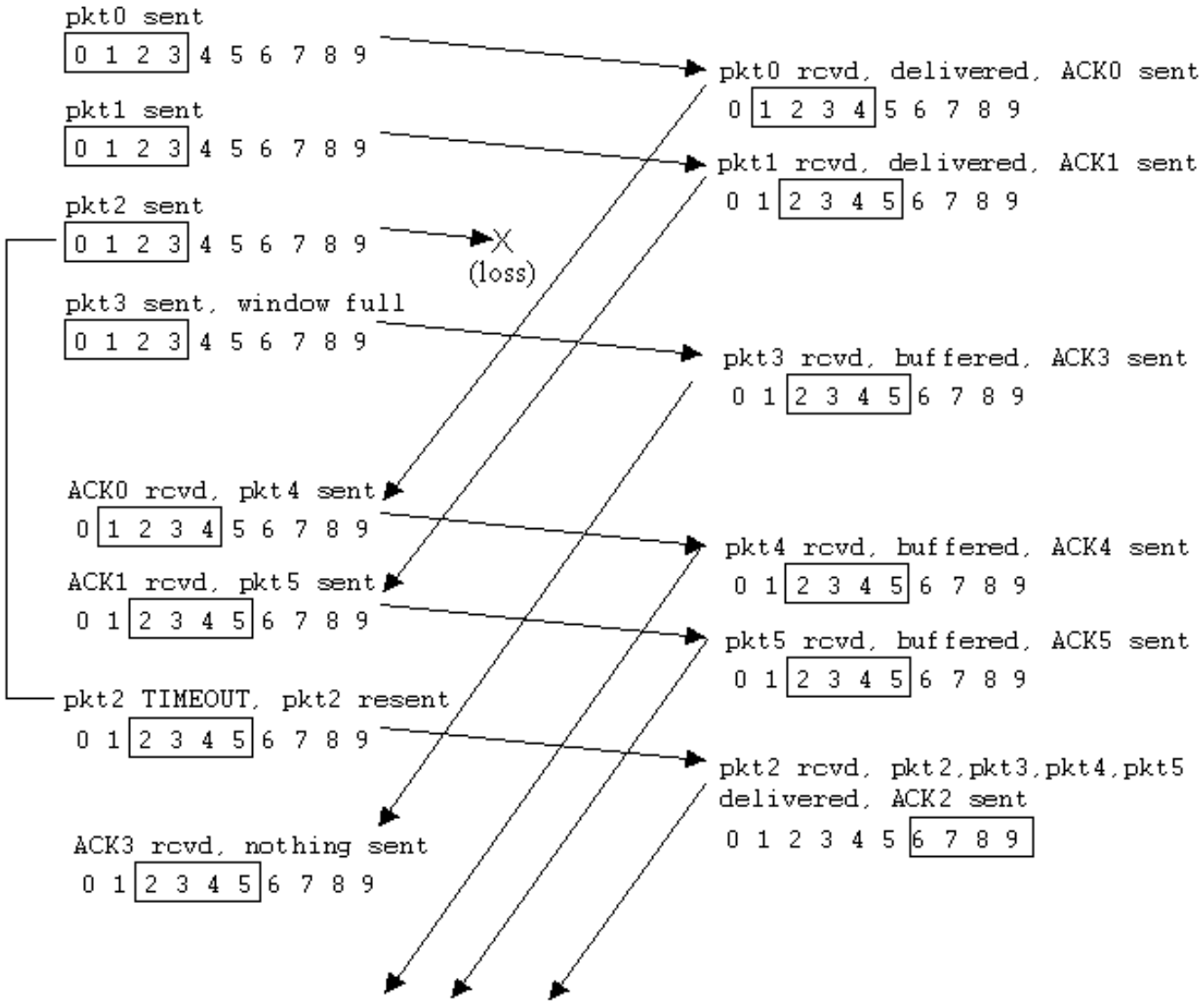
- 发送窗口尺寸为N；接收窗口尺寸为N。



特征：独立ACK，重传单个分组

- 独立ACK: 对每个分组使用单独的确认
- 需N个定时器，若某个分组超时，则重传该分组
- 接收窗口为N，对非按序到达的分组进行缓存

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
(loss)
pkt3 sent, window full
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
pkt2 TIMEOUT, pkt2 resent
0 1 2 3 4 5 6 7 8 9
ACK3 rcvd, nothing sent
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
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 rcvd, pkt2,pkt3,pkt4,pkt5
delivered, ACK2 sent
0 1 2 3 4 5 6 7 8 9

选择重传的理解

理解单独ACK和单个分组重传

- 发送方
  - 发送窗口滑动的条件：收到最低位置分组的确认
  - 超时重传时，仅重传超时的单个分组
- 接收方
  - 接收窗口滑动的条件：收到最低位置的分组
  - 单独ACK
  - 对失序分组的处理：接收窗口内缓存，发对应ACK；接收窗口外丢弃

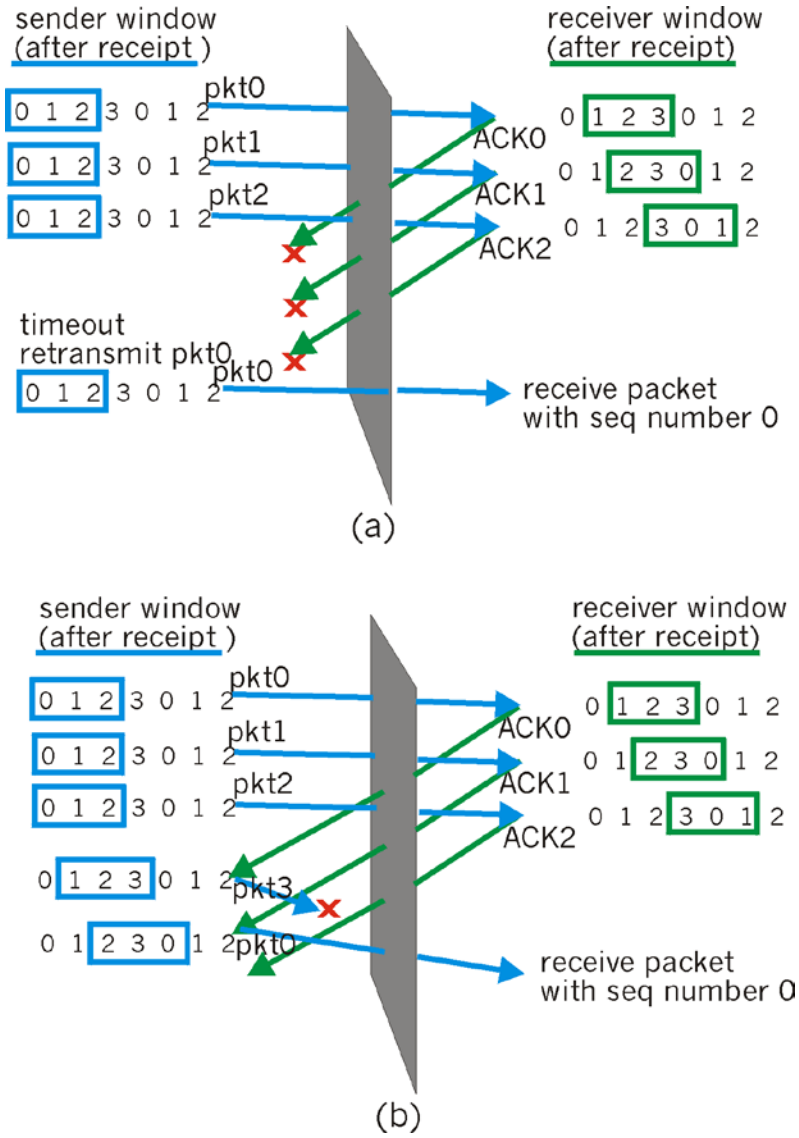
选择重传：困难的问题

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)中不正确地将新的冗余的当为新的，而在(b)中不正确地将新的当作冗余的

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

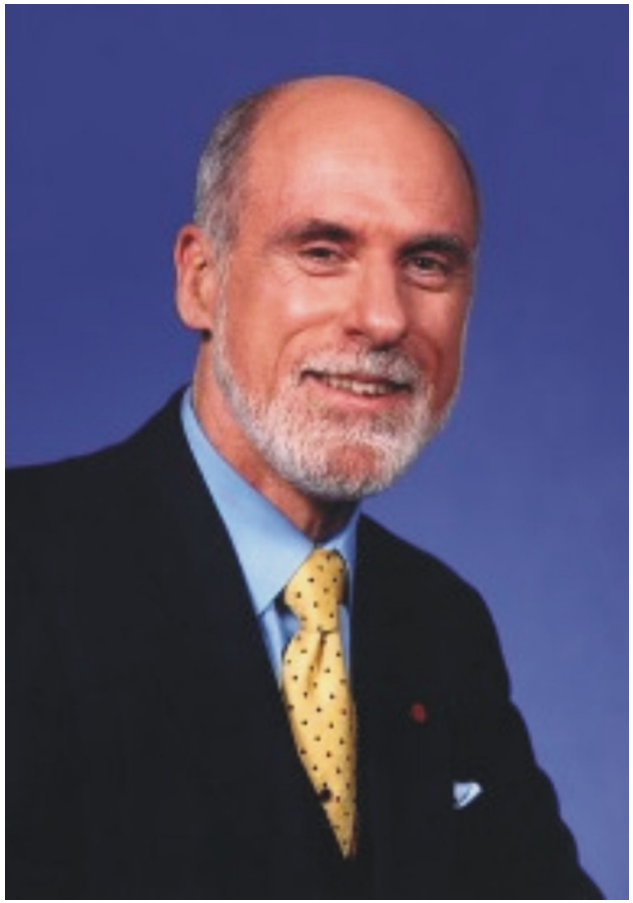


可靠数据传输机制及用途总结

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。

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输：UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的传输：TCP
- 3.6 拥塞控制的原则
- 3.7 TCP拥塞控制

2004年图灵奖 TCP/IP协议发明者



Vinton G. Cerf

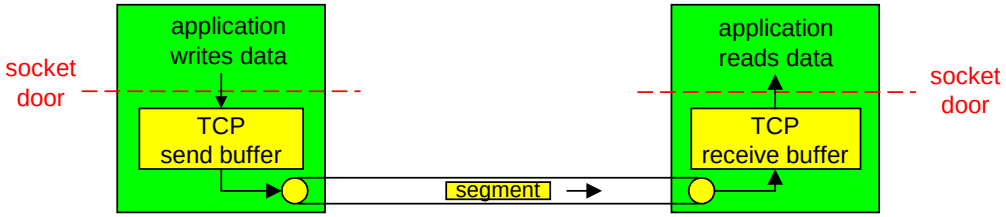
温顿·瑟夫

Robert E. Kahn

罗伯特·卡恩

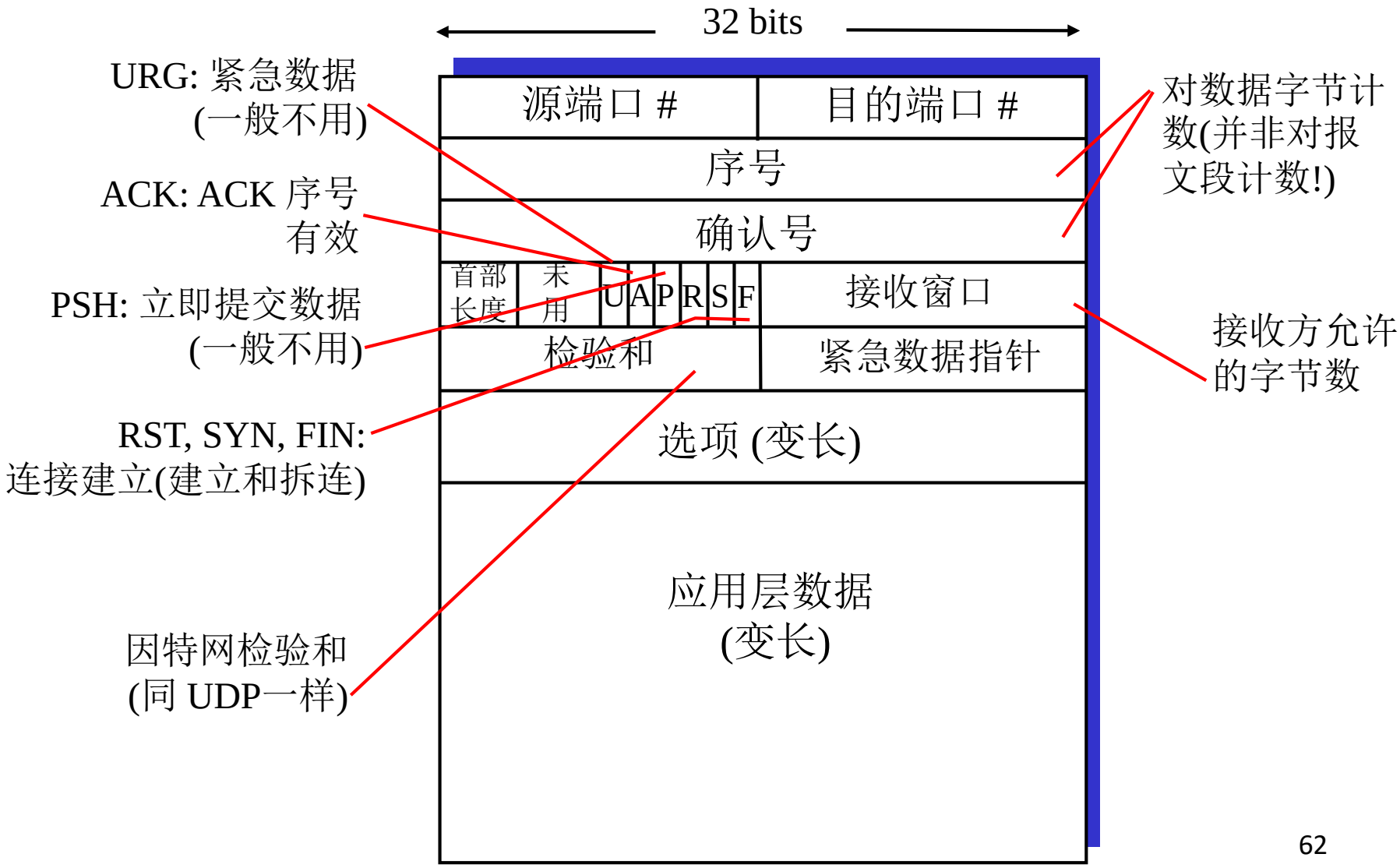
TCP概述

- **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
- **可靠、有序的字节流**
- **流水线：**
  - TCP拥塞和流量控制设置滑动窗口协议
- ***发送和接收缓冲区***



- **全双工数据：**
  - 同一连接上的双向数据流
  - MSS: 最大报文段长度
  - MTU:最大传输单元
- **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),
- **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

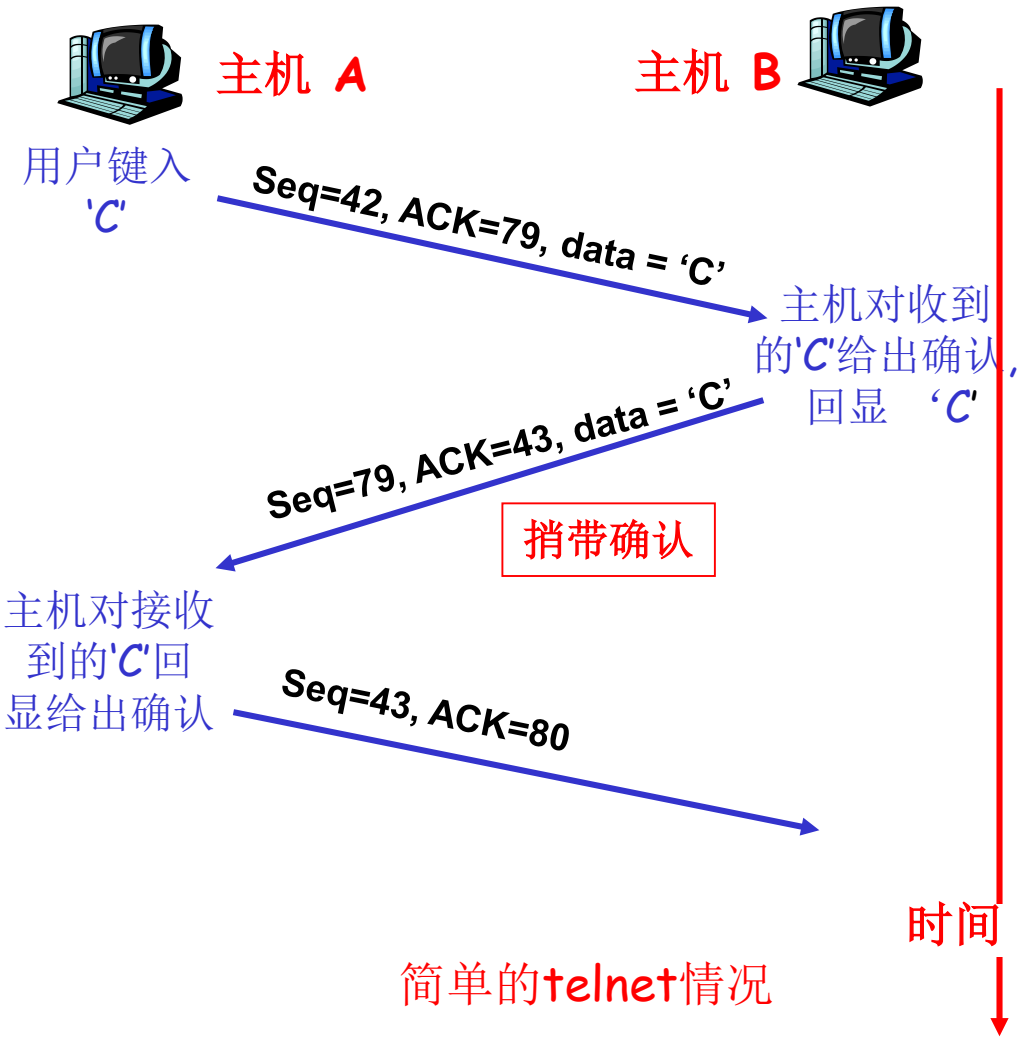
TCP报文段结构


32 bits
源端口 #
目的端口 #
序号
确认号
首部长度
未用
U
A
P
R
S
F
接收窗口
检验和
紧急数据指针
选项 (变长)
应用层数据
(变长)
URG: 紧急数据
(一般不用)
ACK: ACK 序号
有效
PSH: 立即提交数据
(一般不用)
RST, SYN, FIN:
连接建立(建立和拆连)
因特网检验和
(同 UDP一样)
对数据字节计数(并非对报文段计数!)
接收方允许的字节数

TCP序号和确认号

- 序列号：
- 序列号指的是segment中第一个字节的编号，而不是segment的编号
- 建立TCP连接时，双方随机选择序列号
- ACKs：
- 希望接收到的下一个字节的序列号
- 累计确认：该序列号之前的所有字节均已被正确接收到
- Q：接收方如何处理乱序到达的Segment？

A：TCP规范中没有规定，由TCP的实现者做出决策



简单的telnet情况

TCP可靠数据传输特点

- TCP在IP层提供的不可靠服务基础上实现可靠数据传输服务
- 流水线机制
- 累积确认
- TCP使用单一重传定时器

- 触发重传的事件
  - 超时
  - 收到重复ACK
- 渐进式
  - 暂不考虑重复ACK
  - 暂不考虑流量控制
  - 暂不考虑拥塞控制

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?

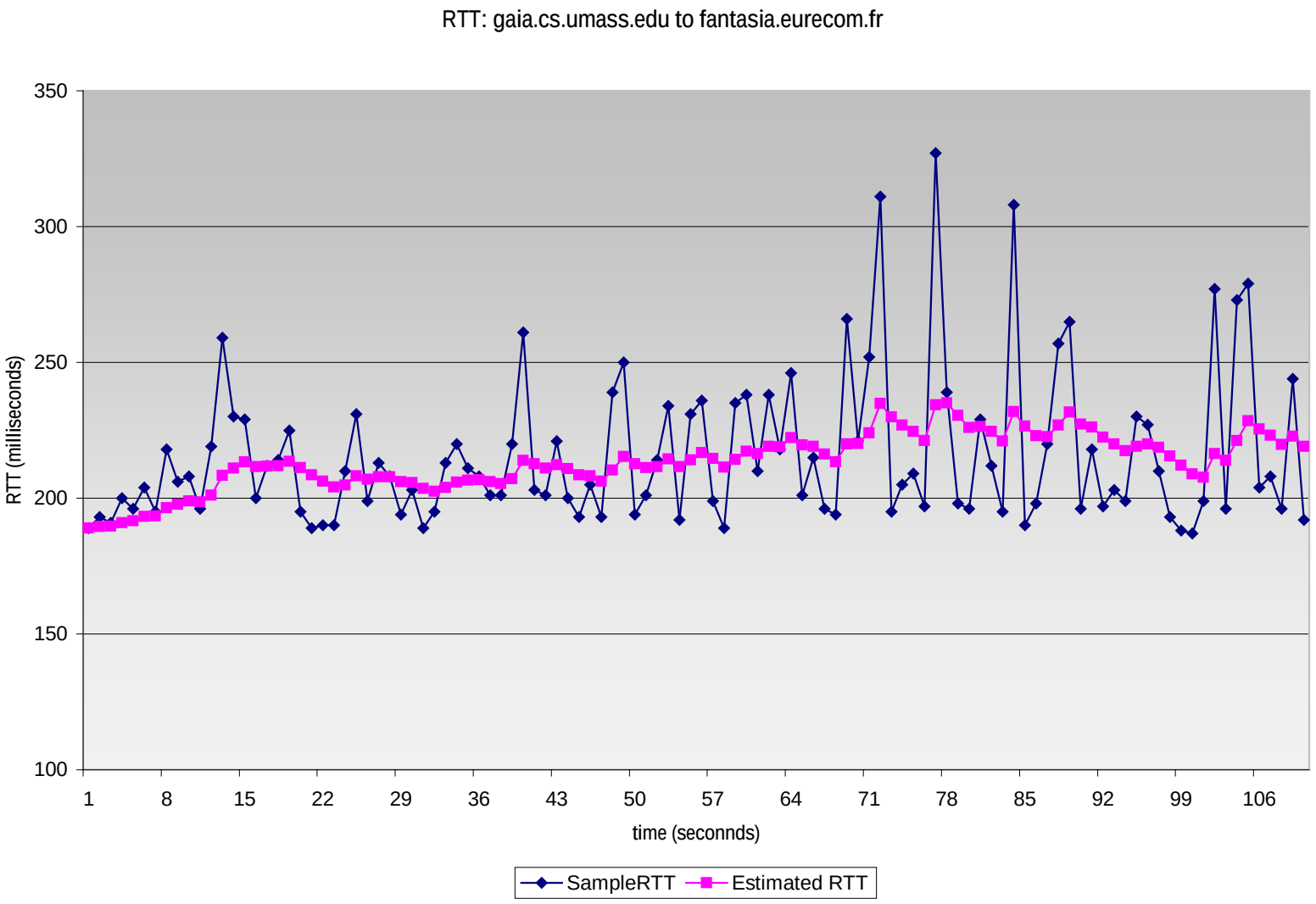
- **SampleRTT**：从发送报文段到接收到ACK的测量时间
  - ◆忽略重传
- **SampleRTT**会变化，希望估计的RTT"较平滑"
  - ◆平均最近的测量值,并不仅仅是当前**SampleRTT**

TCP往返时延估计与超时（续）

EstimatedRTT = (1- α)*EstimatedRTT + α*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: α = 0.125

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

TCP往返时延估计与超时（续）

设置超时间隔

- EstimtedRTT 加 “安全余量”
  - EstimatedRTT大变化-> 更大的安全余量
- 首先估算EstimatedRTT与SampleRTT之间差值有多大 ：

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

(典型地, β = 0.25)

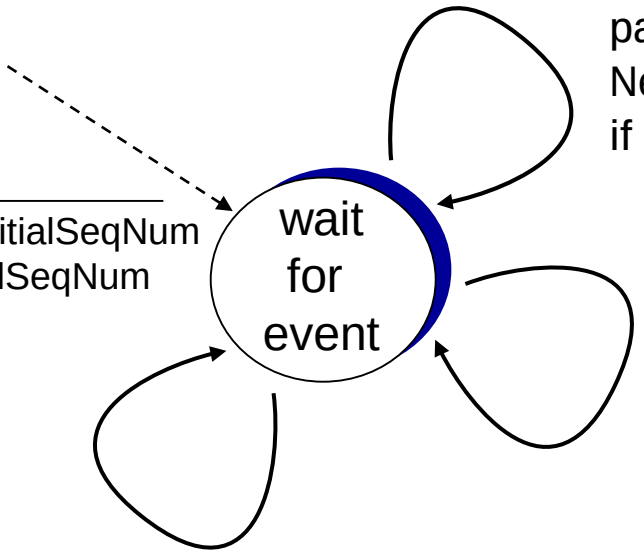
然后估算超时值:

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

TCP发端程序

```
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 */
```

TCP发端程序



Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

data received from application above

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
start timer

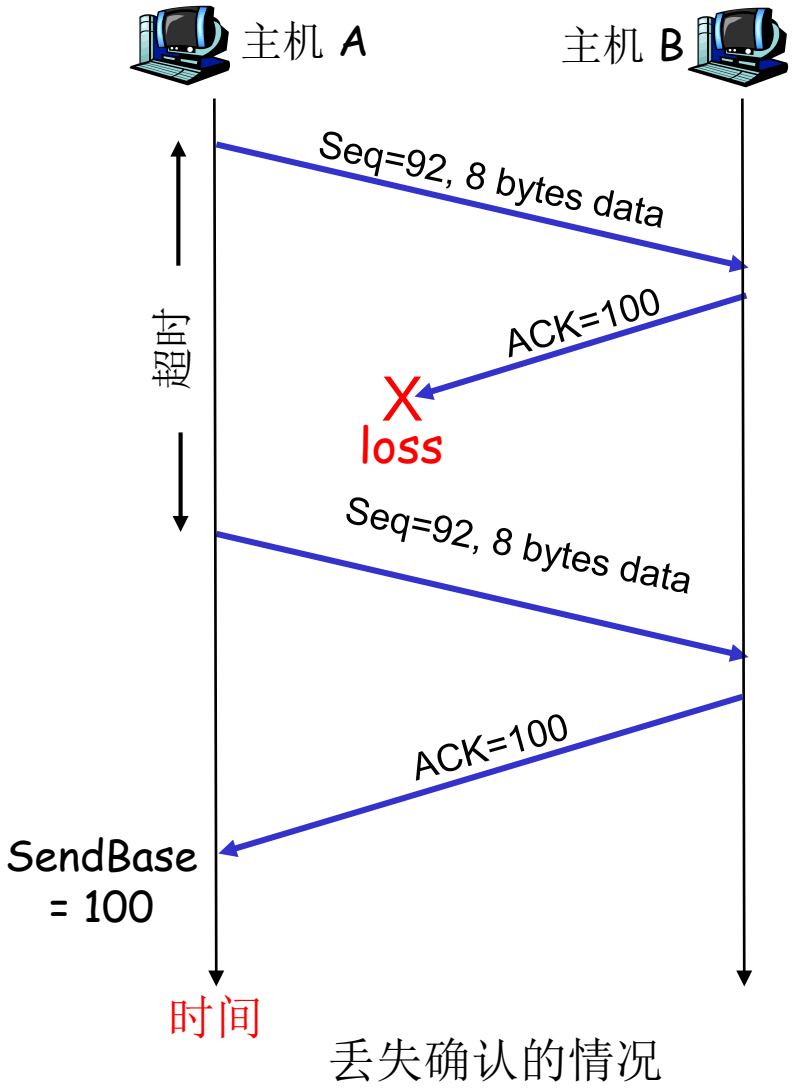
timeout

retransmit not-yet-acked segment
with smallest seq. #
start timer

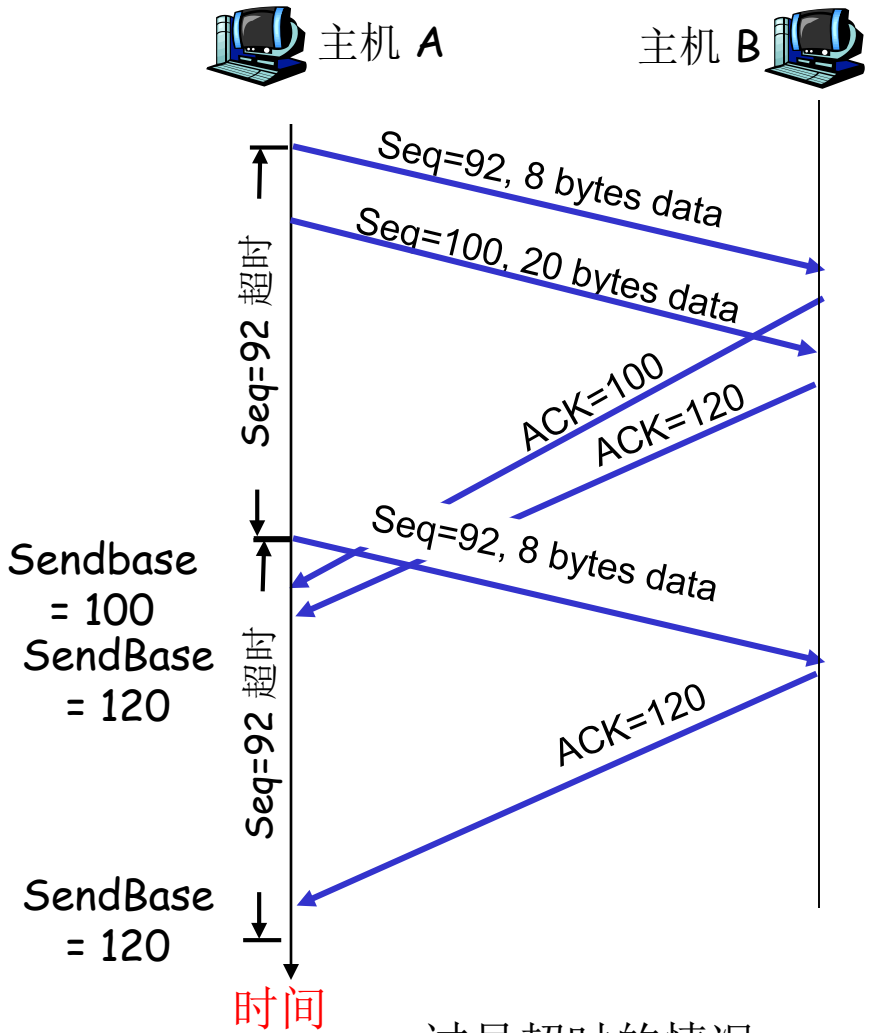
ACK received, with ACK field value y

```
if (y > SendBase) {
    SendBase = y
    /* SendBase–1: last cumulatively ACKed byte */
    if (there are currently not-yet-acked segments)
        start timer
      else stop timer
     }
```

TCP：重传的情况

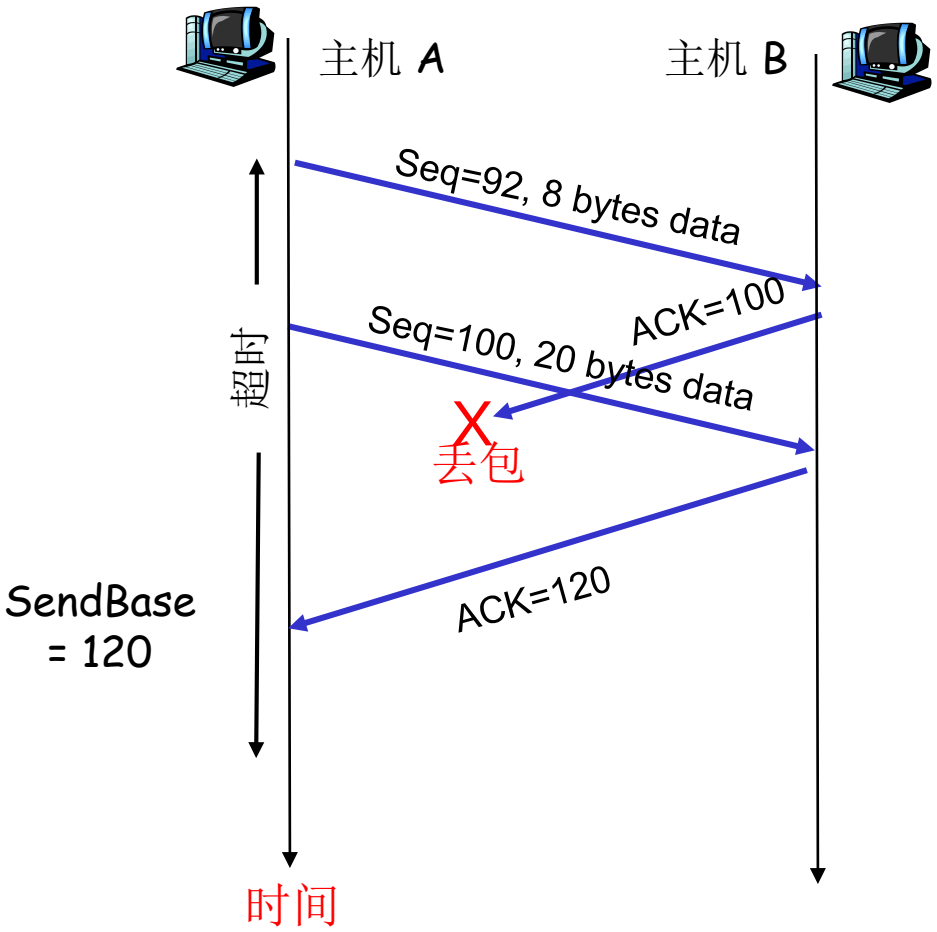


丢失确认的情况



过早超时的情况

TCP 重传情况(续)

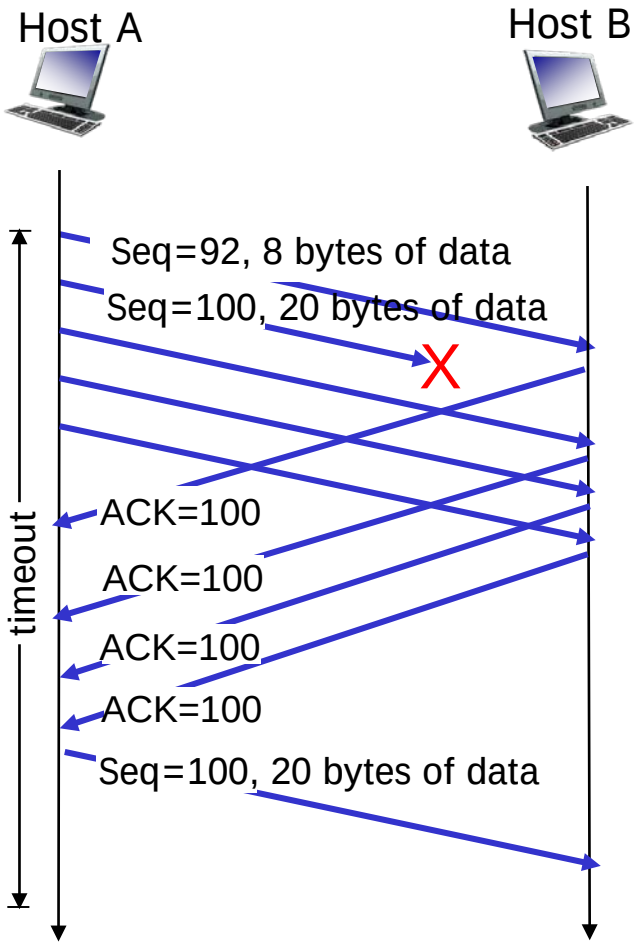


累积确认情况

event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK*, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

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



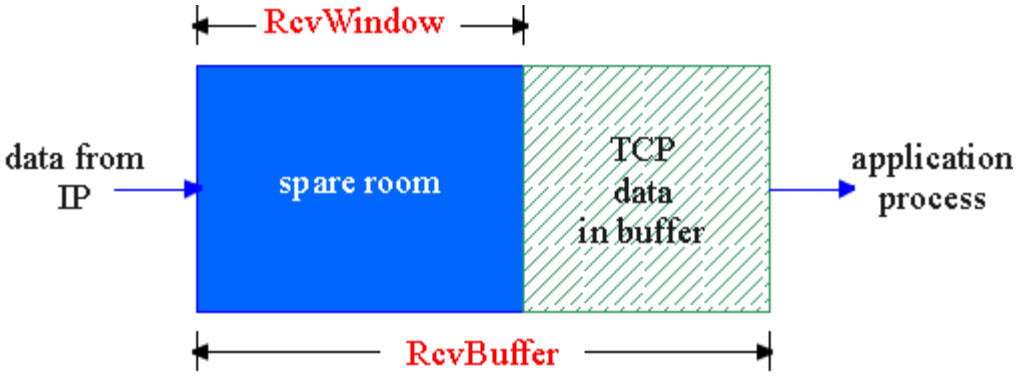
fast retransmit after sender receipt of triple duplicate ACK

TCP流量控制

流量控制

发送方发送数据太快，导致接收方来不及接收时，接收方需进行流量控制

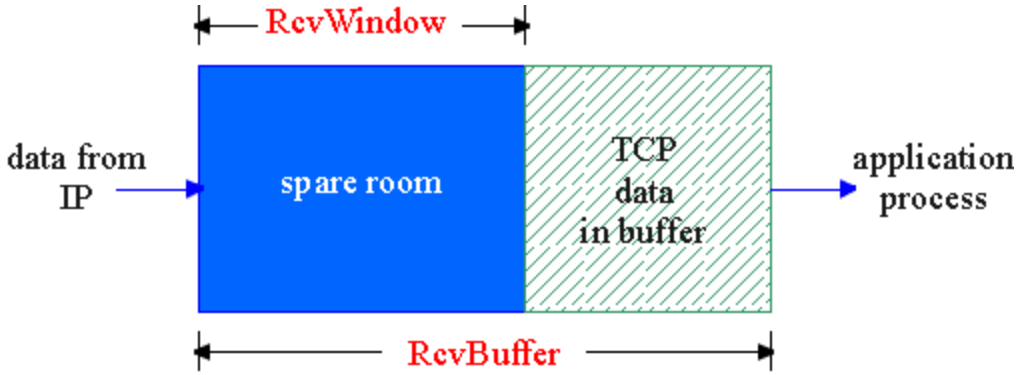
- TCP连接的接收方有1个接收缓冲区：
- 应用进程可能从接收缓冲区读数据缓慢
- 匹配速度服务：发送速率需要匹配接收方应用程序的提取速率



TCP流量控制：工作原理

- TCP流控通过RcvWindow字段实现
  - 接收方计算缓存区的剩余空间，即接收窗口大小：

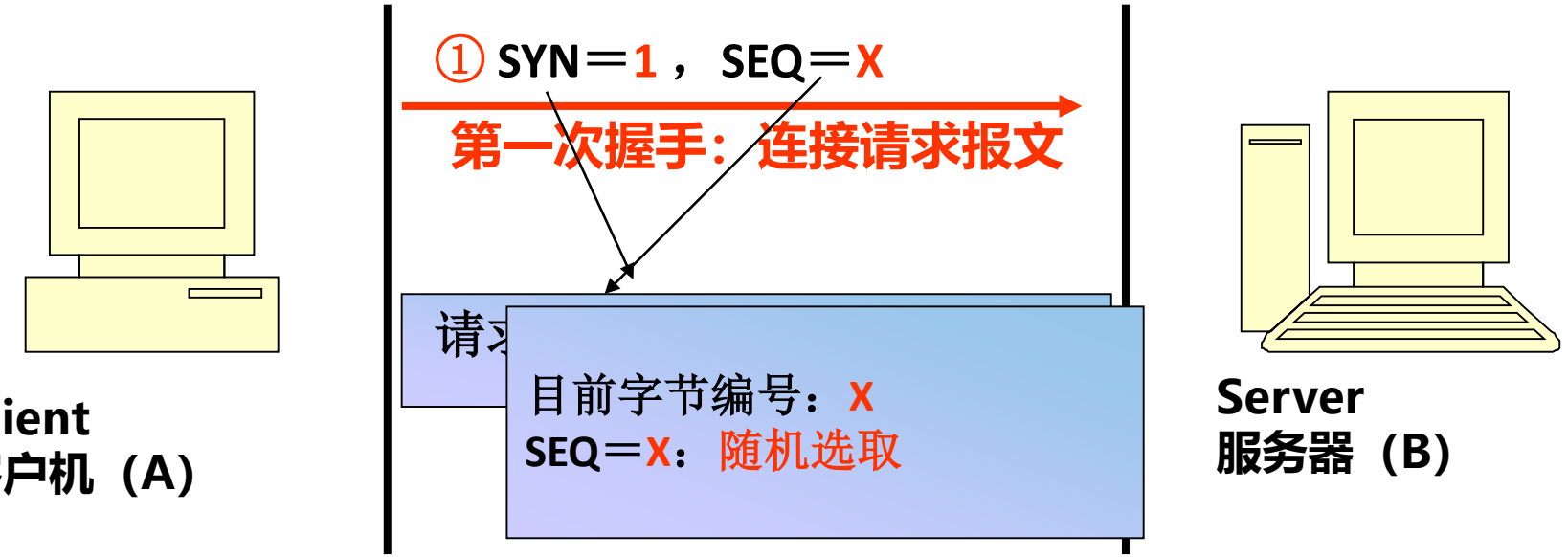
    RcvWindow= RcvBuffer-[LastByteRcvd -LastByteRead]
  - 接收方通过TCP首部的接收窗口字段反馈给发送方
  - 发送方根据接收窗口字段来限制发送窗口大小，以保证接收方缓存不溢出



TCP连接管理

- TCP是面向连接的协议，TCP连接的建立和释放是每次TCP传输中必不可少的过程。
- TCP的传输连接包括三个状态
  - 连接建立
  - 数据传输
  - 连接释放

建立连接（三次握手）



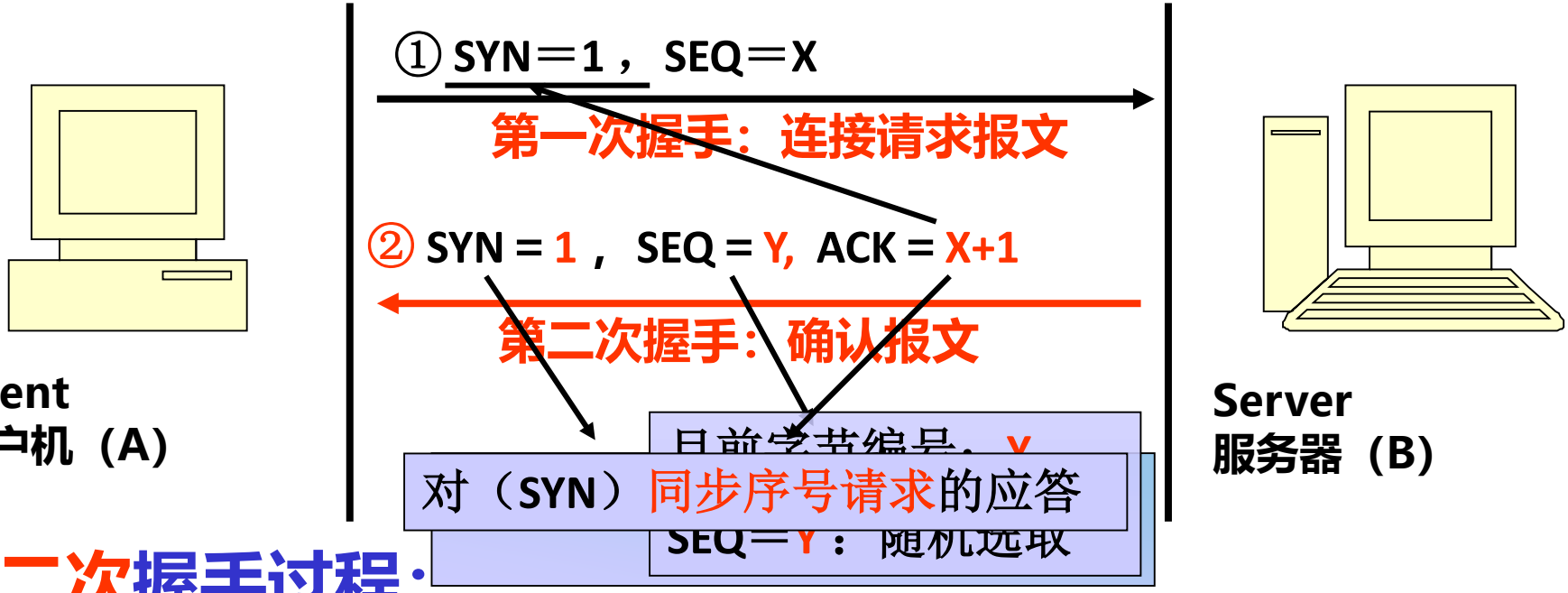
第一次握手过程：

注：

SYN： 同步序列编号(Synchronize Sequence Number)

SEQ： 序列号(Sequence Number), 表示当前数据传输字节的编号为X。

建立连接（三次握手）



第二次握手过程：

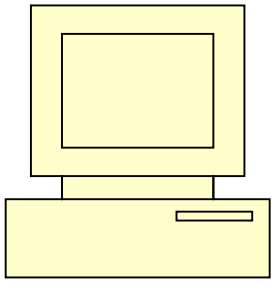
注：

SYN： 同步序列编号(Synchronize Sequence Numbers)

SEQ： 序列号(Sequence Number)

ACK： 确认编号(Acknowledgement Number)

建立连接（三次握手）



Client
客户机（A）

① SYN＝1， SEQ＝X

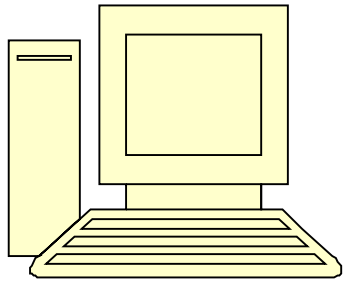
第一次握手：连接请求报文

② SYN = 1， SEQ = Y, ACK = X+1

第二次握手确认报文

③ SYN = 0， SEQ = X+1, ACK = Y+1

第三次握手：确认报文

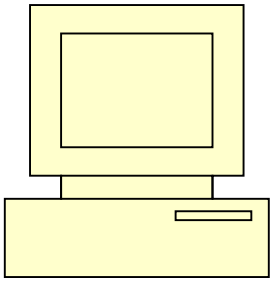


Server
服务器（B）

第三次握手过程

对（SYN）同步序号请求的应答

建立连接（三次握手）



Client
客户机（A）

① SYN＝1， SEQ＝X

请求

② SYN = 1， SEQ = Y, ACK = X+1

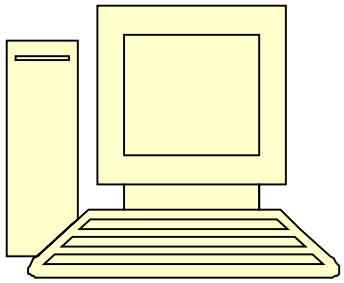
确认

③ SYN = 0， SEQ = X+1, ACK = Y+1

确认

连接已建立

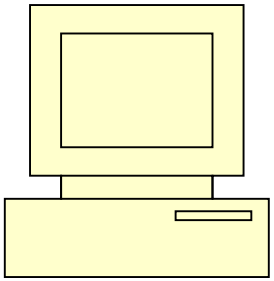
数据



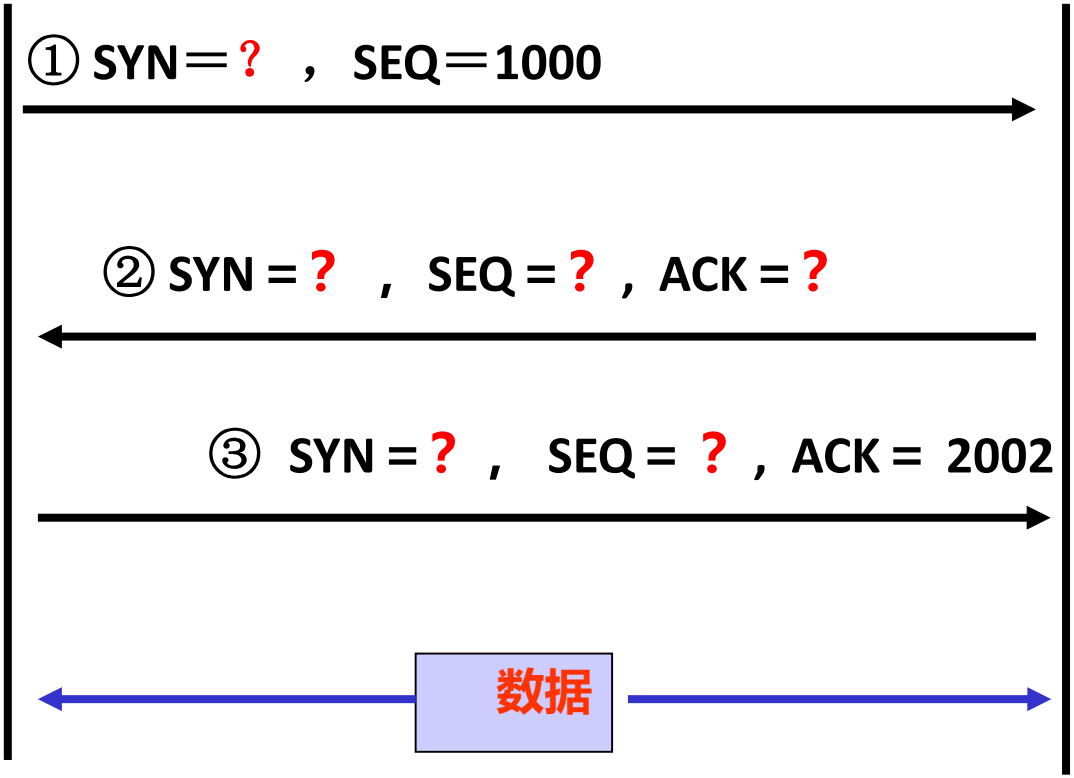
Server
服务器（B）

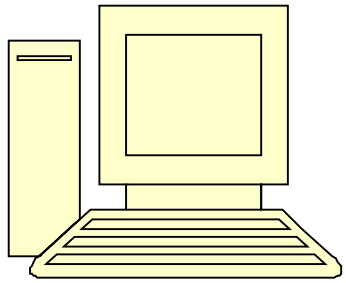
三次握手过程：一个请求，两个确认

练习



Client
客户机（A）





Server
服务器（B）

三次握手过程：一个请求，两个确认

释放连接

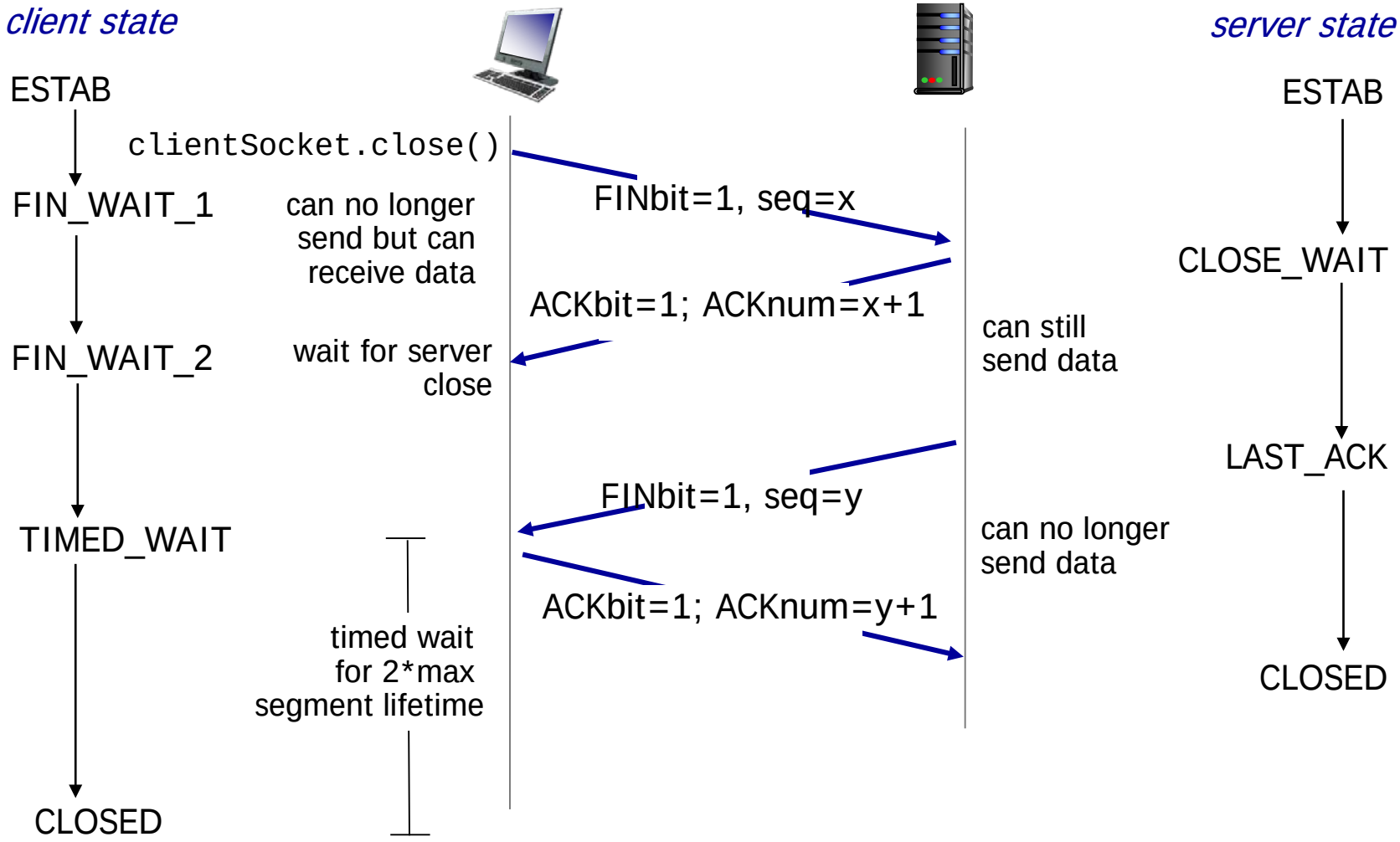
步骤 1: 客户机向服务器发送TCP FIN控制报文段

步骤 2: 服务器收到FIN，用ACK回答。关闭连接，发送FIN

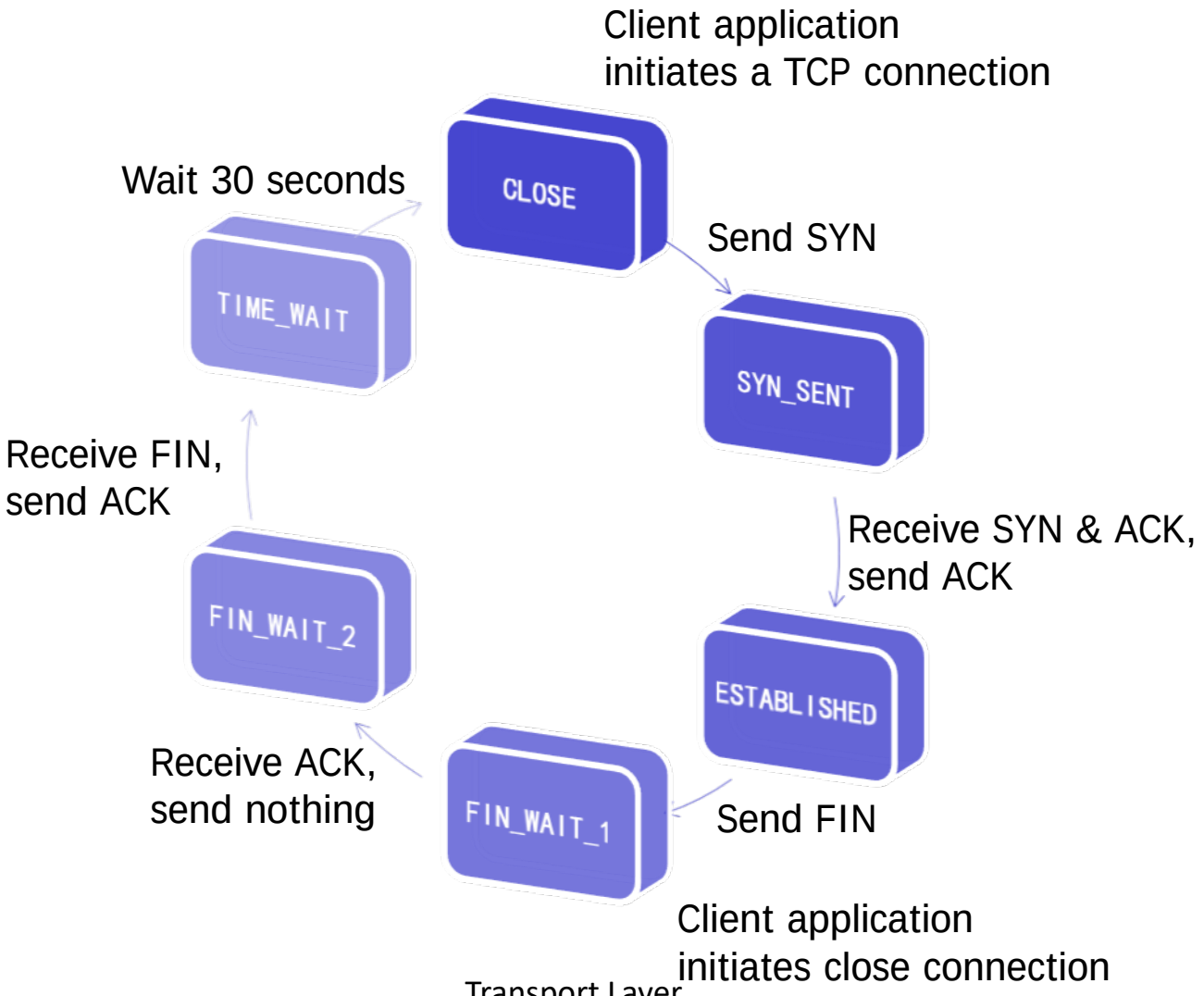
步骤 3: 客户机收到FIN，用ACK答

- 进入“定时等待” - 将对接收到的FIN进行确认

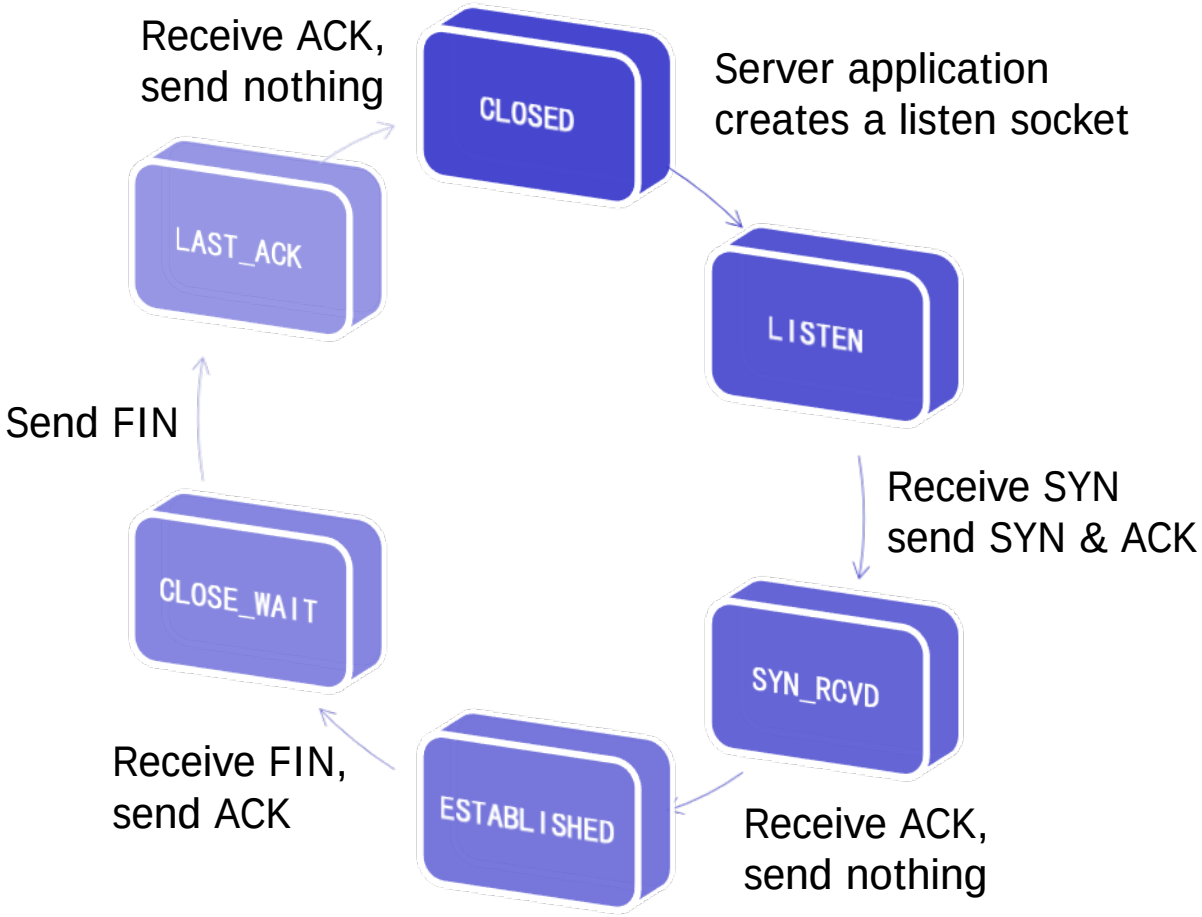
步骤 4: 服务器接收ACK，连接关闭


client state
server state
ESTAB
clientSocket.close()
FIN_WAIT_1
can no longer send but can receive data
FINbit=1, seq=x
ACKbit=1; ACKnum=x+1
FIN_WAIT_2
wait for server close
can still send data
CLOSE_WAIT
LAST_ACK
FINbit=1, seq=y
TIMED_WAIT
can no longer send data
ACKbit=1; ACKnum=y+1
timed wait for 2*max segment lifetime
CLOSED
CLOSED

A typical sequence of TCP states visited by a client TCP



A typical sequence of TCP states visited by a server-side TCP



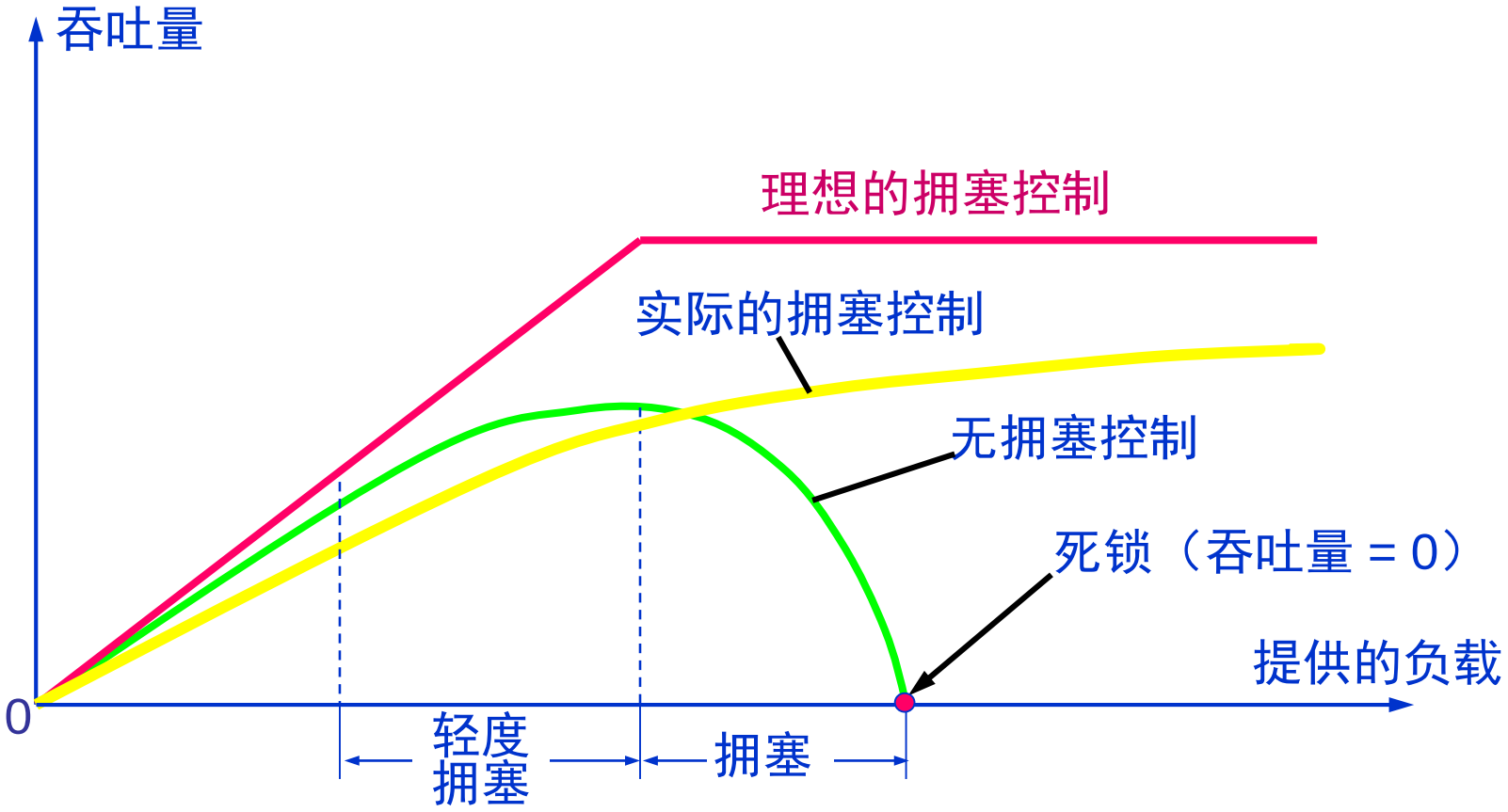
- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输：UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的传输：TCP
- **3.6 拥塞控制的原则**
- 3.7 TCP拥塞控制

拥塞控制原理

拥塞(Congestion):

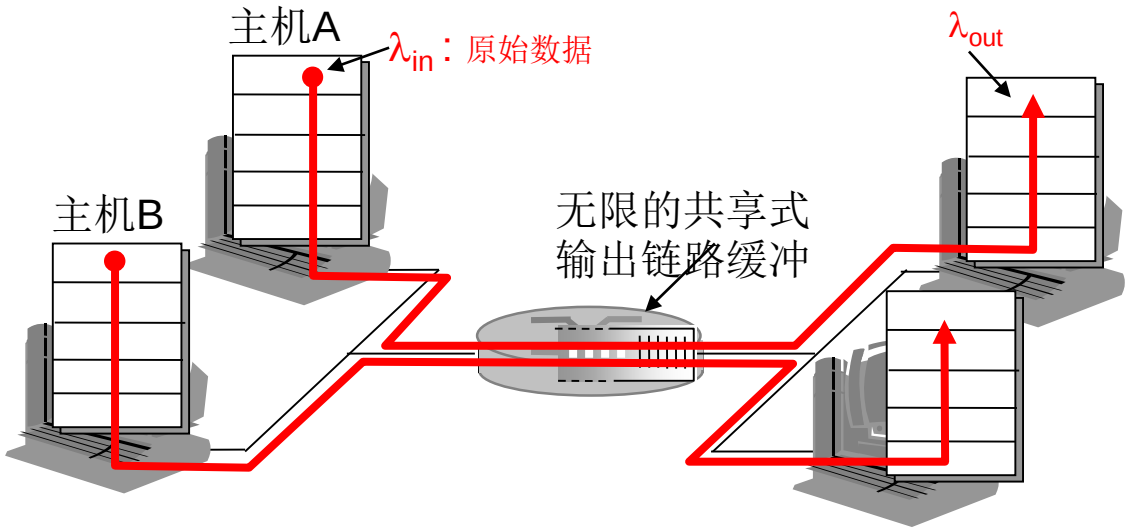
- 当大量的分组进入网络，超出了网络的处理能力时，会引起网络局部或整体性能下降，这种现象称为拥塞。不加控制的拥塞甚至导致整个网络瘫痪。
- 表现:
  - 丢包 (路由器缓冲区溢出)
  - 长时延 (路由器缓冲区中排队)
  - 拥塞控制 vs 流量控制

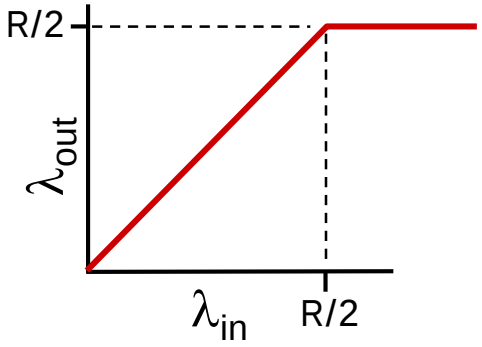
拥塞控制起的作用

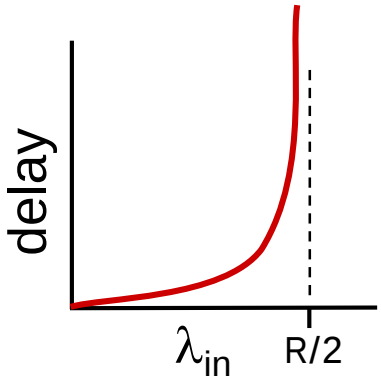


拥塞的原因与开销：情况1

- 两个发送方，两个接收方
- 一个路由器，无限缓冲区
- 无重传



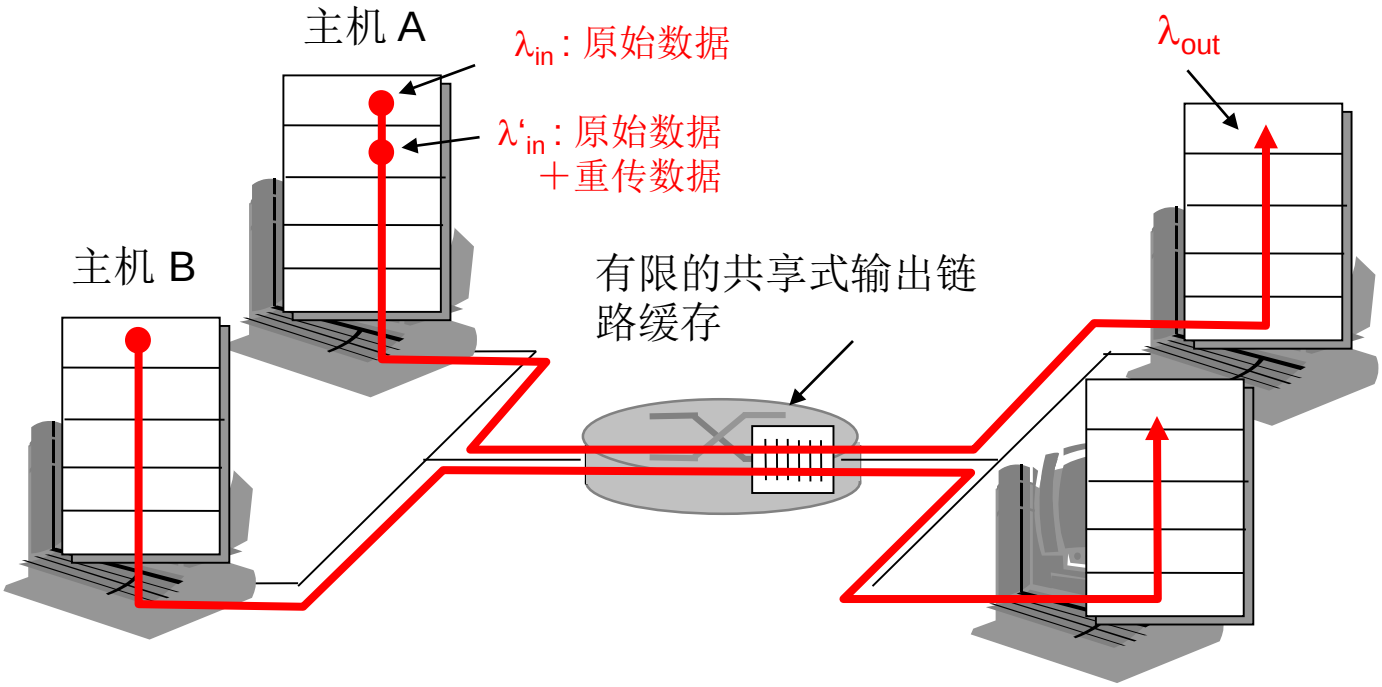




- 拥塞时时延增大
- 可达到最大吞吐量

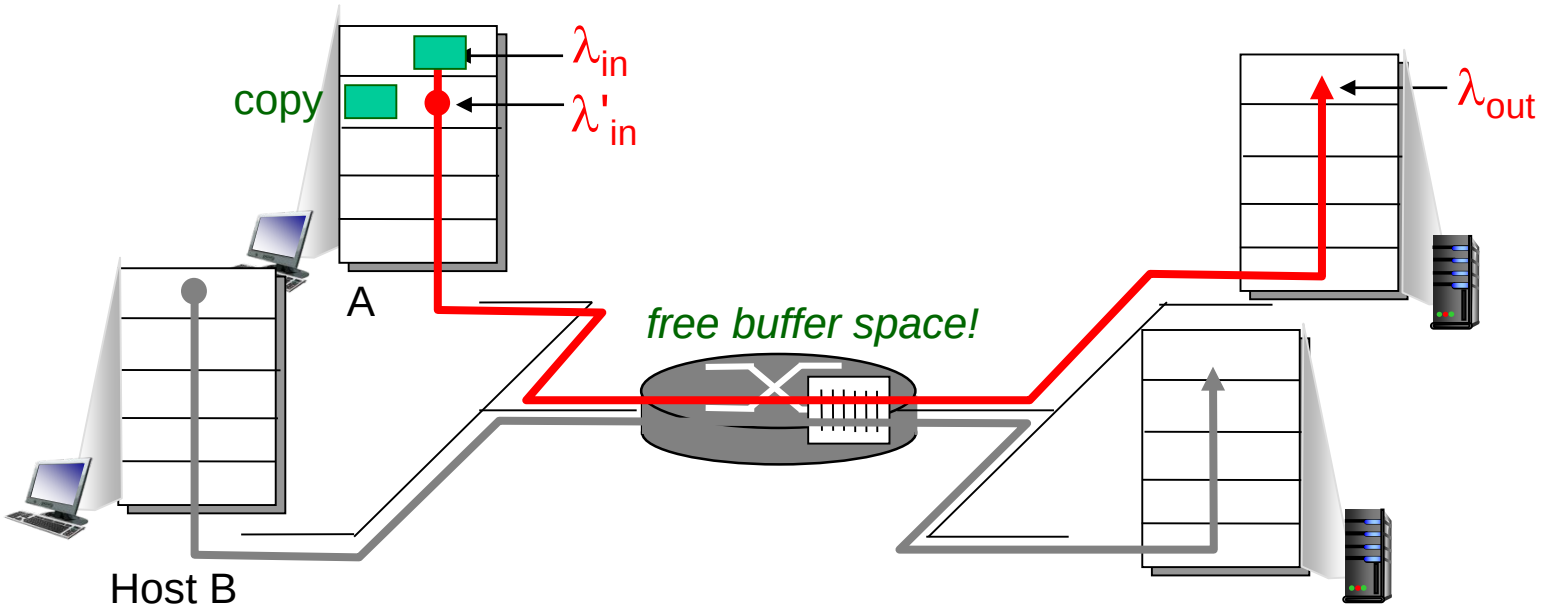
拥塞的原因与开销 ： 情况2

一个路由器，有限缓冲区

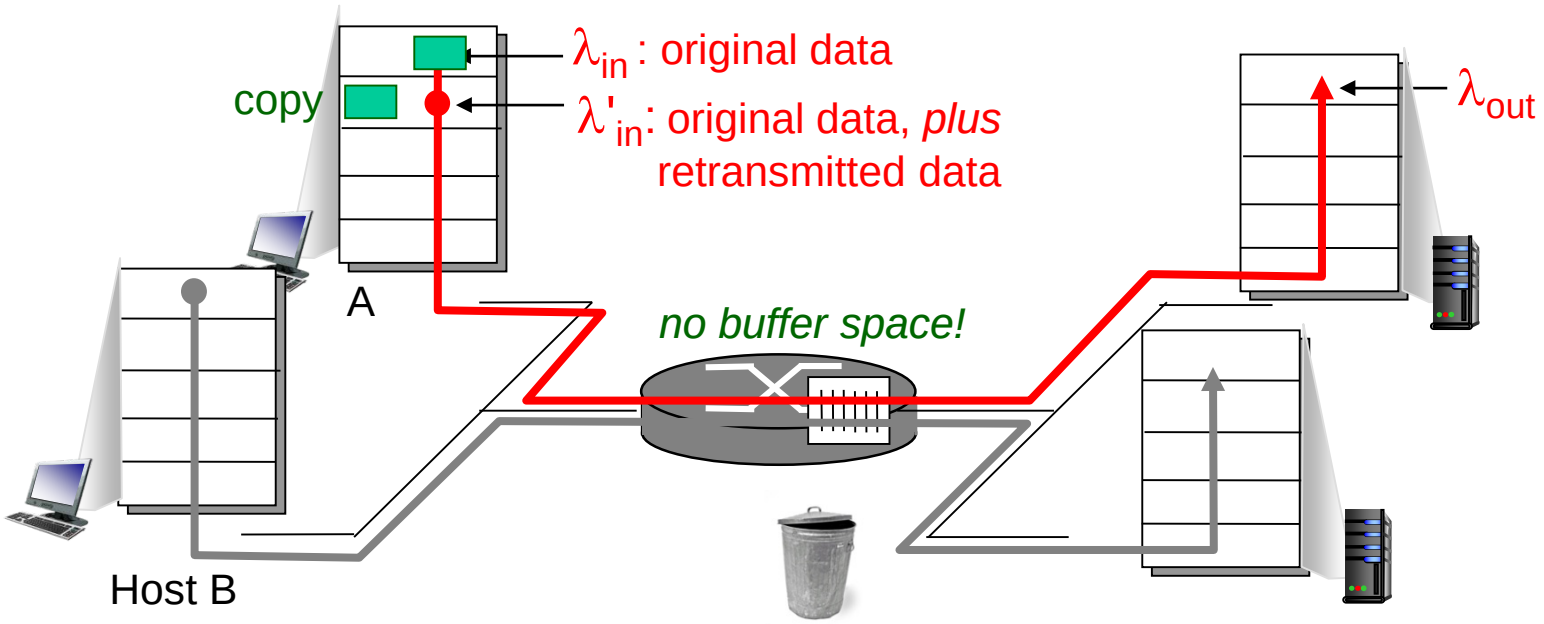


拥塞的原因与开销 ： 情况2

一个路由器，有限缓冲区



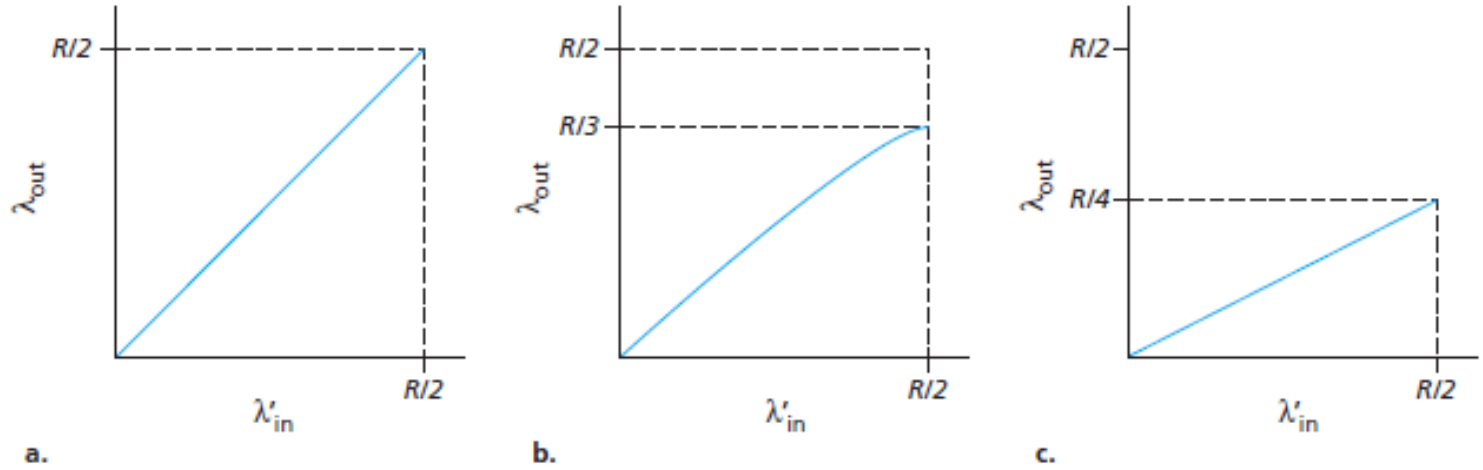
代价：发送方重传超时的数据分组



代价：发送方重传丢失的数据分组

拥塞的原因与开销：情况2（续）

- a：空闲： $\lambda_{in} = \lambda_{out}$
- 丢包需要重传 ： $\lambda'_{in} > \lambda_{out}$
- 丢包和超时都要重传： $\lambda'_{in} > \lambda_{out}$

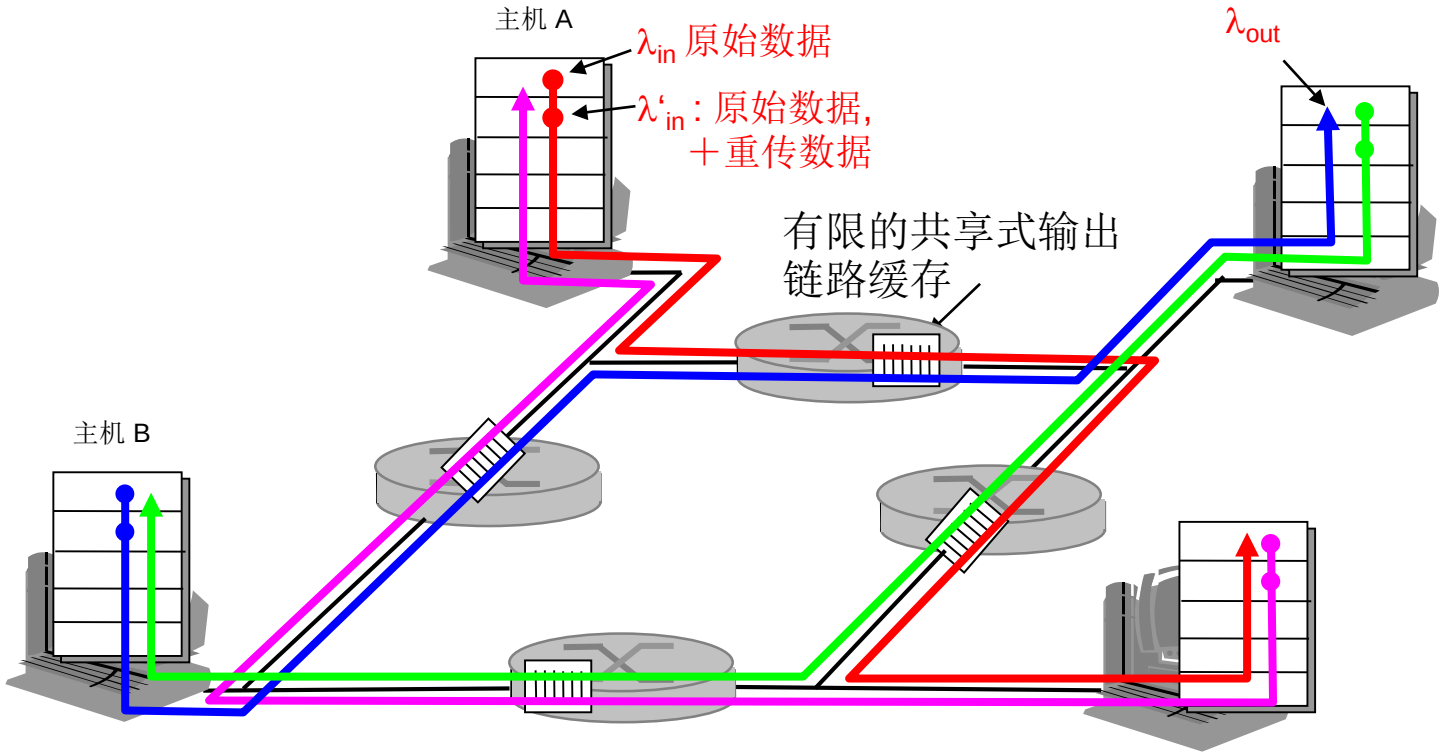


拥塞的代价：
由于重传造成传输效率的降低和资源的浪费

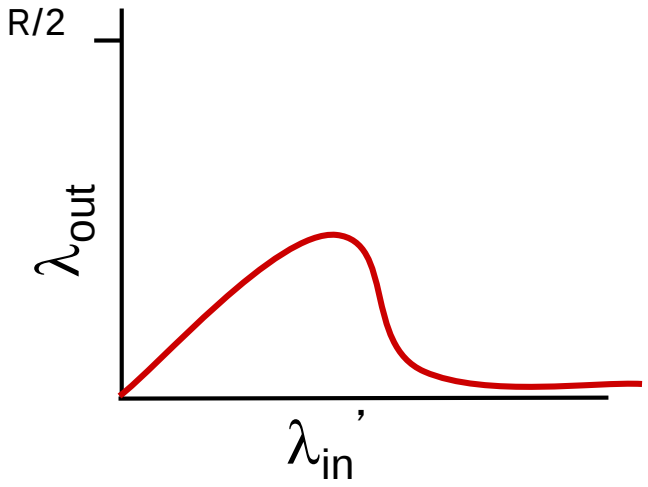
拥塞的原因与开销：情况3

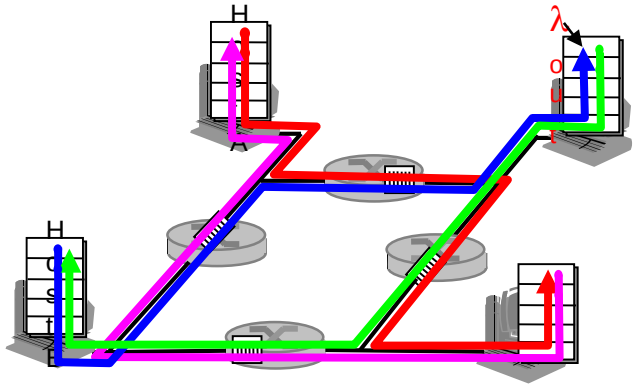
- 四个发送者
- 多跳路径
- 超时/重传

问题：随着 λ_{in} 和 λ'_{in} 的增加将发生什么情况？



拥塞的原因与开销：情况3（续）





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！

拥塞控制方法

控制拥塞的两类方法（直接的和间接的）：

端到端的拥塞控制

- 不能从网络得到明确的反馈
- 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,TCP/IP ECN, ATM)
  - ◆指示发送方按照一定速率发送

案例研究：ATM ABR拥塞控制

ABR:available bit rate

- “弹性服务”
- 如果发送方路径 “underloaded”

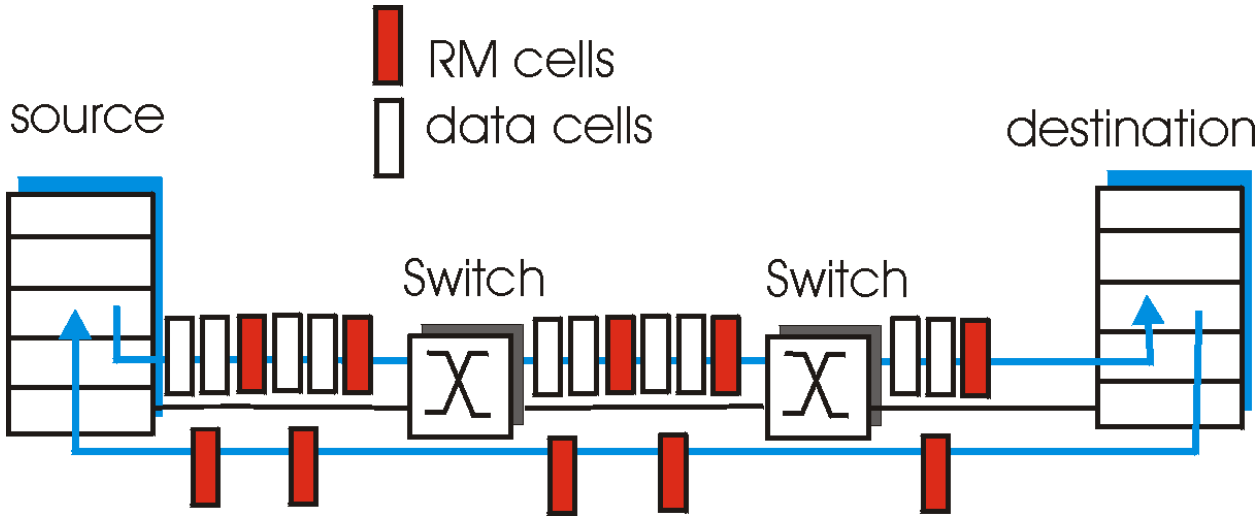
  使用可用带宽
- 如果发送方路径拥塞

  将发送速率降到最低保障速率

RM(resource management) cells

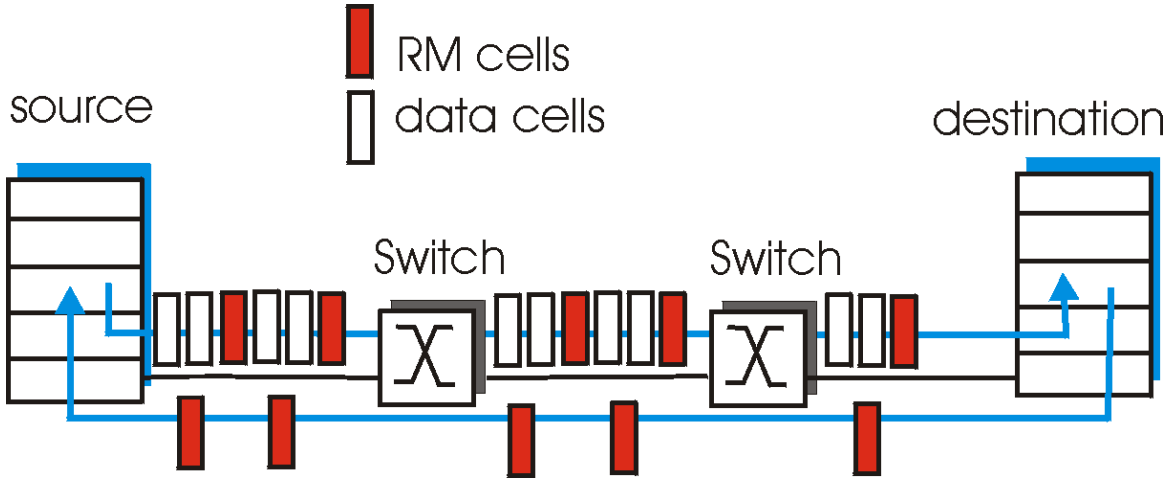
- 发送方发送
- 交换机设置RM cell位(网络辅助)
  - NI bit: rate不许增长
  - CI bit:拥塞指示
- RM cell由接收方返回给发送方

ATM ABR拥塞控制



- 通信过程简要描述
  - 发送方沿（建立好连接的）路径上不断传输数据信元和管理信元，到达接收方
  - 接收方将管理信元（内容修改调整后）研路径返回（反馈）到发送方

案例:ATM ABR拥塞控制



- 采取网络辅助的拥塞控制（包括多种机制）
  - RM信元中的特定bit：NI bit速率无增长 (轻度拥塞)，CI bit拥塞指示
  - RM信元中的两字节 ER (明确速率)字段：确定路径上所有交换机的最小支持速率
  - 数据信元中的EFCI bit：被拥塞的交换机设置为1 ， 接收方将在返回的RM信元的CI位置1

- 3.1 运输层服务
- 3.2 复用与分解
- 3.3 无连接传输：UDP
- 3.4 可靠数据传输的原理
- 3.5 面向连接的传输：TCP
- 3.6 拥塞控制的原则
- **3.7 TCP拥塞控制**

TCP拥塞控制

- 端到端控制（没有网络辅助）
- 发送方限制传输:滑动窗口法，发送方未被确认的数据量小于拥塞窗口 cwnd ：

LastByteSent-LastByteAcked≤CongWin

rate ≈ CongWin/RTT

- 拥塞窗口是动态的，具有感知到的网络拥塞的函数

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)
- 如何合理地调整发送速率?

加性增—乘性减： AIMD

慢启动: SS

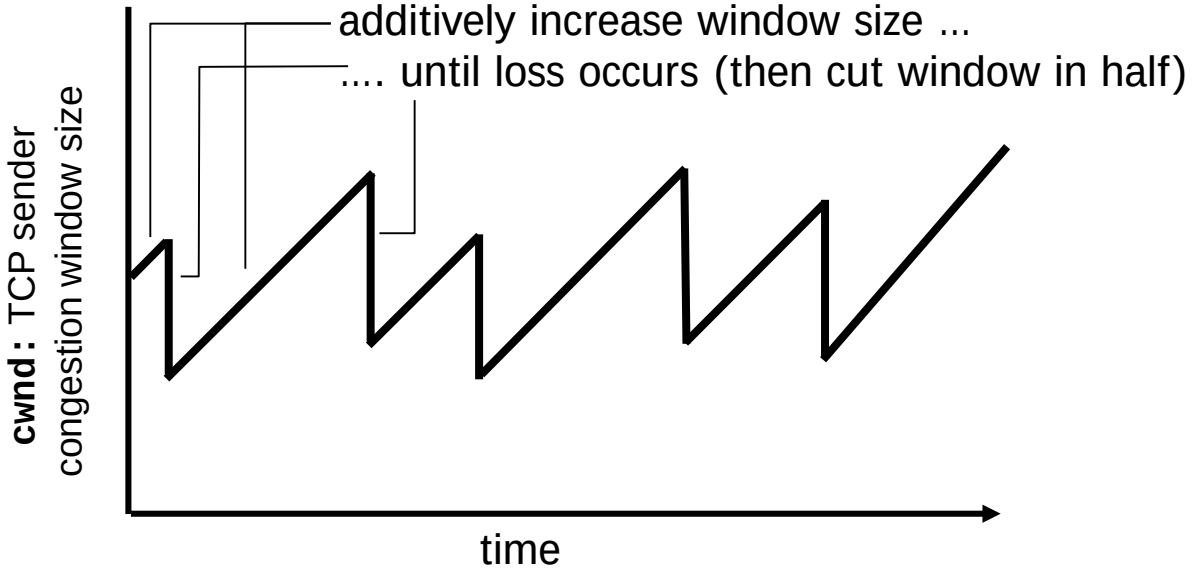
TCP加性增乘性减AIMD

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)

乘性减:

丢包事件后，拥塞窗口值减半

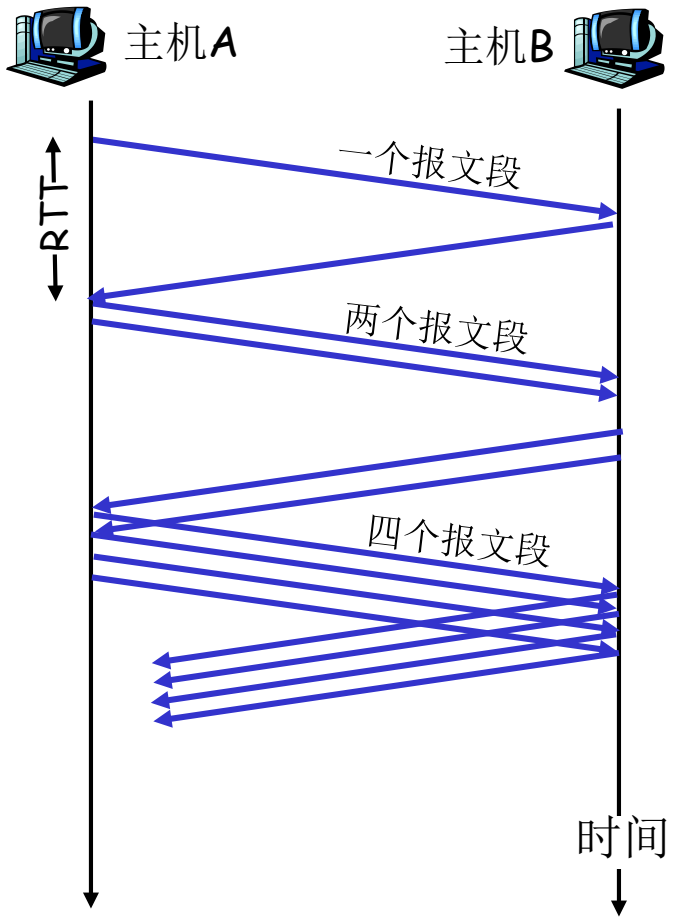


TCP慢启动

- 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

TCP慢启动(续)

- 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
  - 每收到ACK，增加拥塞窗口
- 总结：初始速率很低，但以指数快地增加



拥塞避免机制：Threshold

- Q:何时应该指数性增长切 换为线性增长(拥塞避免)?

  A: 当CongWin达到Loss事 件前值的1/2时.

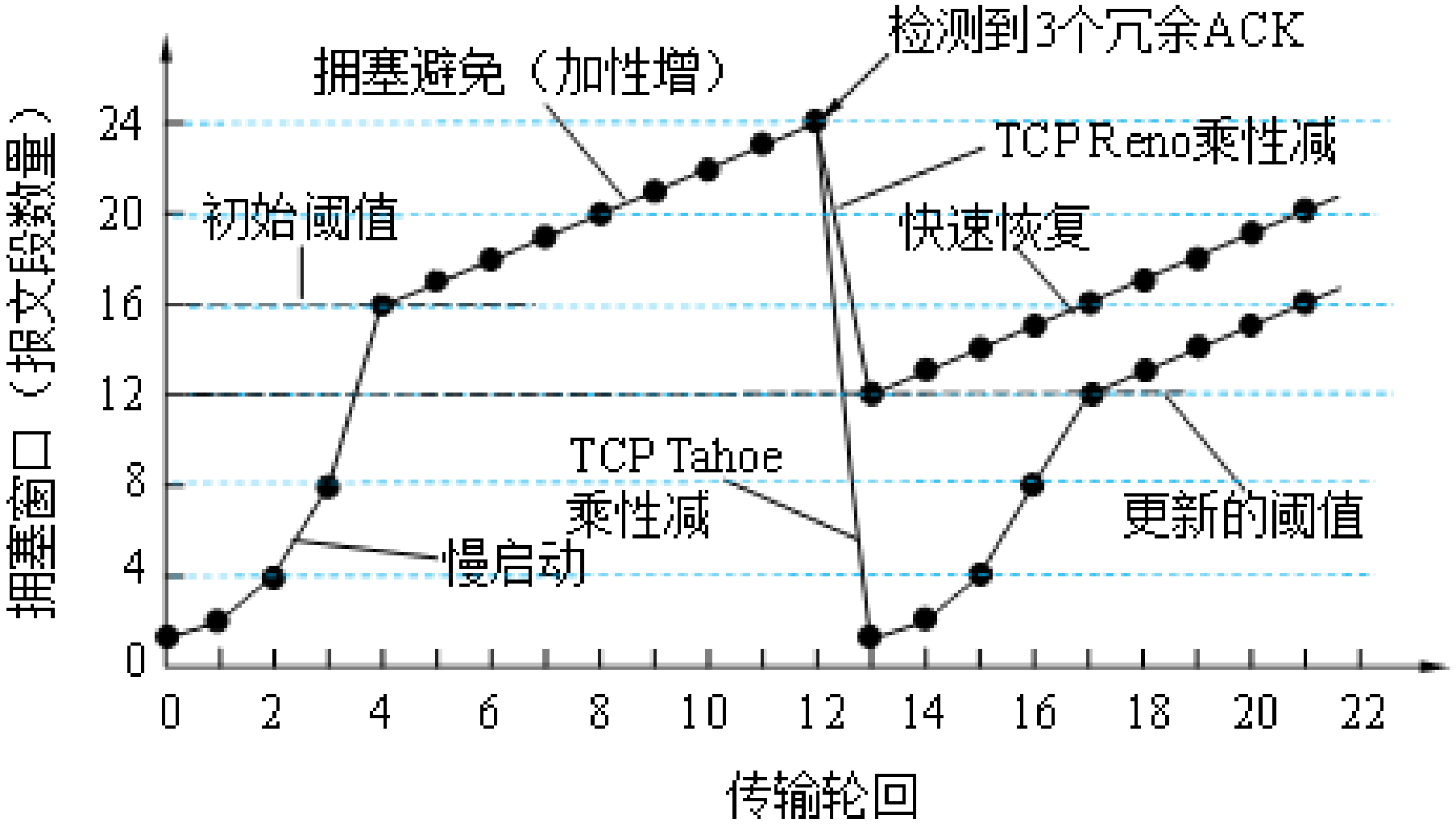
- 实现方法:

- 变量 Threshold
- Loss事件发生时, Threshold 被设为Loss事件前 CongWin 值的1/2。

拥塞避免机制：Loss事件处理机制

- 基本思想
  - 3个冗余ACK指示网络还具有某些传送报文段的能力
  - 直接超时，则更为 “严重”
- 收到3个冗余确认后：
  - **cwnd**减半
  - 窗口再线性增加
- 但是超时事件以后：
  - **cwnd**值设置为1 MSS
  - 窗口再指数增长
  - 到达一个阈值 (Threshold) 后，再线性增长

TCP拥塞控制与拥塞窗口变化：例子



TCP 拥塞控制：小结

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

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?

  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

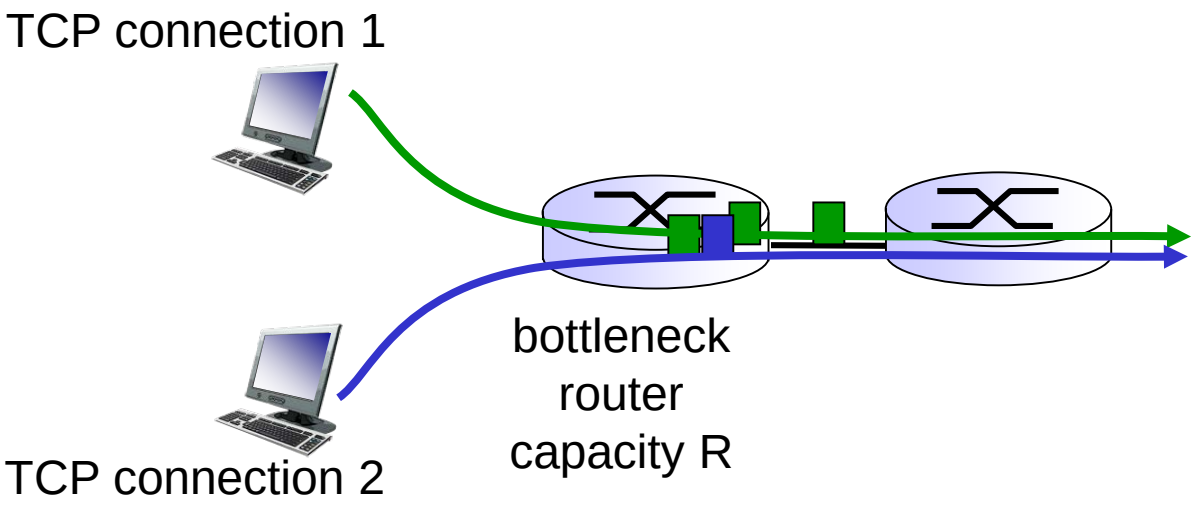
- 实际网络性能分析中，TCP吞吐量是信噪比，窗口大小，分组长，重传次数，往返时延等参数的复杂函数

$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT}\sqrt{L}}$$

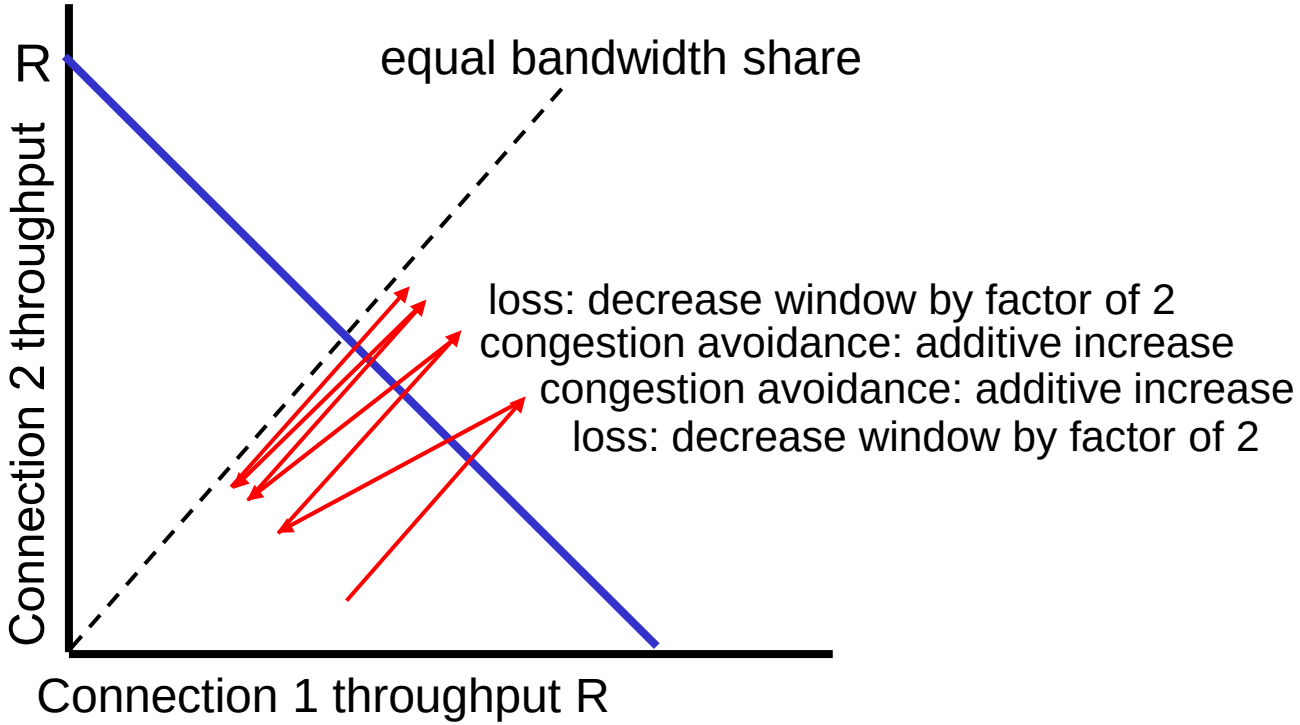
TCP 公平性

公平性?

- 如果K个TCP Session共享相同的瓶颈带宽R，那么每个Session的平均速率为R/K



TCP 公平性


R
equal bandwidth share
Connection 2 throughput
loss: decrease window by factor of 2
congestion avoidance: additive increase
congestion avoidance: additive increase
loss: decrease window by factor of 2
Connection 1 throughput R

TCP 公平性

- 公平性与UDP
- 多媒体应用通常不使用TCP， 以免被拥塞控制机制限制速率
- 使用UDP:以恒定速率发送， 能够容忍丢失 产生了不公平
- 研究:TCP friendly
- 公平性与并发TCP连接
  - 某些应用会打开多个并发连接 （Web浏览器）
  - 产生公平性问题
- 例子:链路速率为R，已有9个 连接
- 新来的应用请求1个TCP，获得 R/10的速率
- 新来的应用请求11个TCP，获得 R/2的速率

本章小结与作业

重要内容：

- 多路复用与多路分解。
- UDP和TCP的比较。
- 可靠的数据传输服务模型rdt1.0，rdt2.0，rdt3.0，底层信道模型，停等协议与流水线协议，Go-Back-N与选择重传。
- TCP序号、确认、可靠传输机制，TCP连接的建立。
- TCP拥塞控制

作业：

- R6、R8、R12、R13、R15、R17
- P27、P40、P42
- 思考题：