# CH3 传输层

## 一、传输层概述

### 1. 传输层服务

传输层协议为不同主机上的*进程*直接提供了**逻辑通信**功能。

网络层协议为不同*主机*提供了逻辑通信功能。

UCP，用户数据报协议，提供了一种不可靠、无连接的服务。

TCP，传输控制协议，提供了一种可靠、面向连接的服务，以及拥塞控制。

IP为不可靠服务，不保证报文段的*交付*、*按序交付*、*完整性*。

TCP和UDP将端系统间的IP交付服务扩展到端系统上进程间的交付服务，称为**传输层的多路复用与多路分解**。

## 二、多路分解和多路复用

### 1. 概念

多路分解：将报文交付到正确socket（邮箱→各种邮件）

多路复用：从不同socket中搜集数据封装首部信息并将生成报文传递到网络层（各种邮件→邮箱）

### 2. 传输层报文段

源端口号字段 + 目的端口号字段 + 其他首部数据 + 数据

### 3. 端口

0-1023：周知端口号，受限，提供给特定功能（HTTP : TCP 80，Mail : TCP 25，ftp : TCP 2）

1024-65535（$2^{16}-1$）

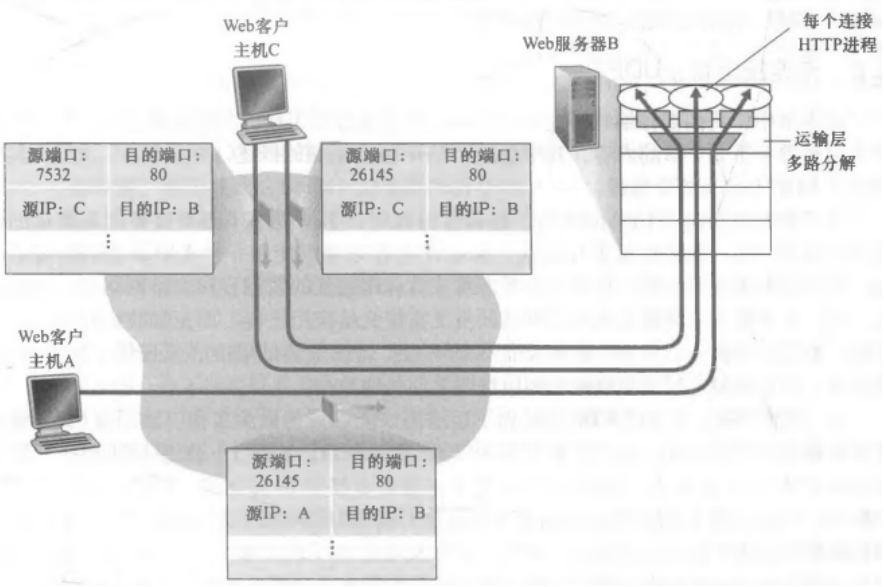
### 5. UDP套接字

(目的IP，目的端口号)二元组标识

### 6. TCP套接字

(ip_s, port_s, ip_s, port_t)四元组标识

主机使用四个值将报文端定向到相应的套接字。




# 三、UDP

## 1. 简介

UDP使用数据+源/目的端口号等封装成UDP报文段，交给网络层，网络层将UDP报文段封装为IP数据报，并尽力交付给正确的进程。

- 要求速度并且容许少量错误的应用推荐使用UDP
- 不需要建立连接
- 没有连接状态
- 分组首部开销小

不提供的服务：可靠、流量控制、拥塞控制、时间、带宽保证、建立连接。

## 2. 报文结构

源端口号+目的端口号+长度+校验和+数据

## 3. 校验和

计算方法：对数据包的首部以及其他信息每16位相加，如果产生进位，将进位加在后面，然后计算反码得到。

用于检测差错。

检验：校验范围+校验和=ffff则通过校验

## 4. 细节

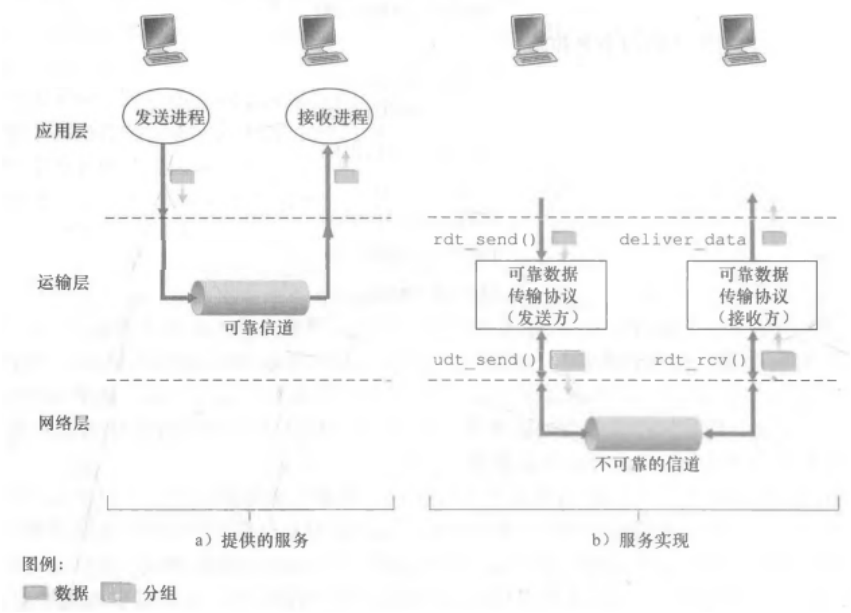
UDP是端到端的。

DNS、SNMP、HTTP/3使用UDP

# 四、可靠数据传输原理

## 1. 简介

在传输层实现可靠信道。




## 2. rdt1.0

假设底层信道完全可靠（没有比特出错，没有分组丢失），使用有限状态机描述。

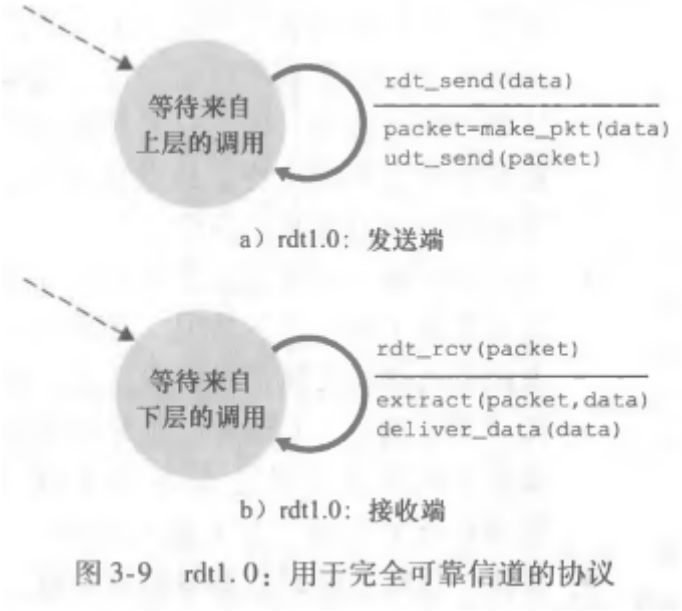



图 3-9　rdt1.0：用于完全可靠信道的协议

## 3. rdt2.0

经过的信道可能有比特差错。

使用**肯定确认(ACK)**和**否定确认(NAK)**实现重传机制。——基于重传机制的可靠数据传输协议称为**自动重传协议(ARQ)**。

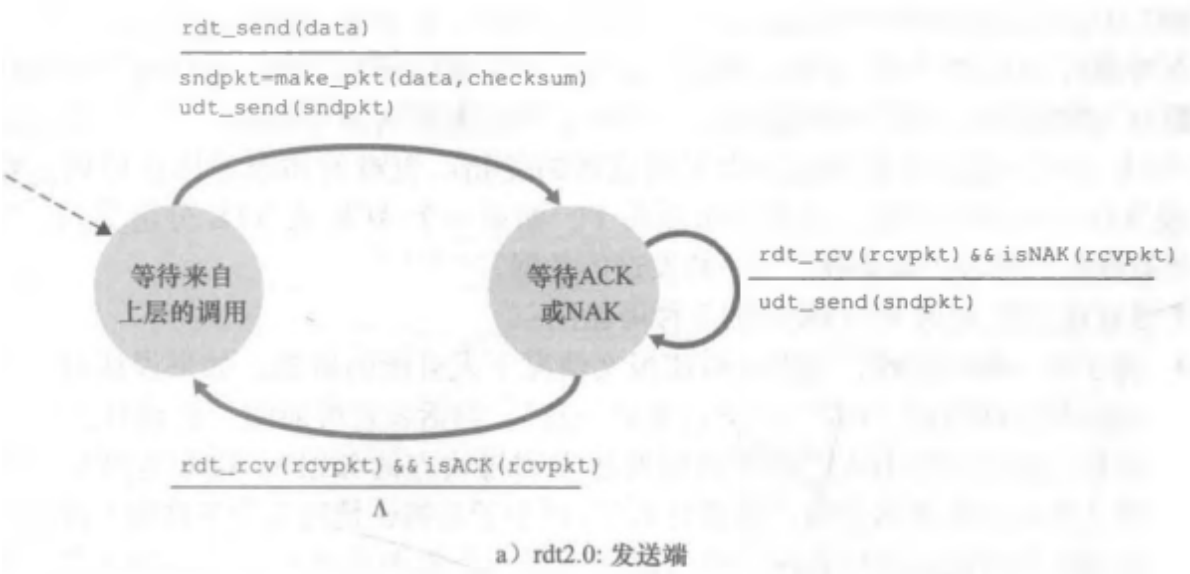
需要实现：差错检测、接收方反馈、重传。

接收方：

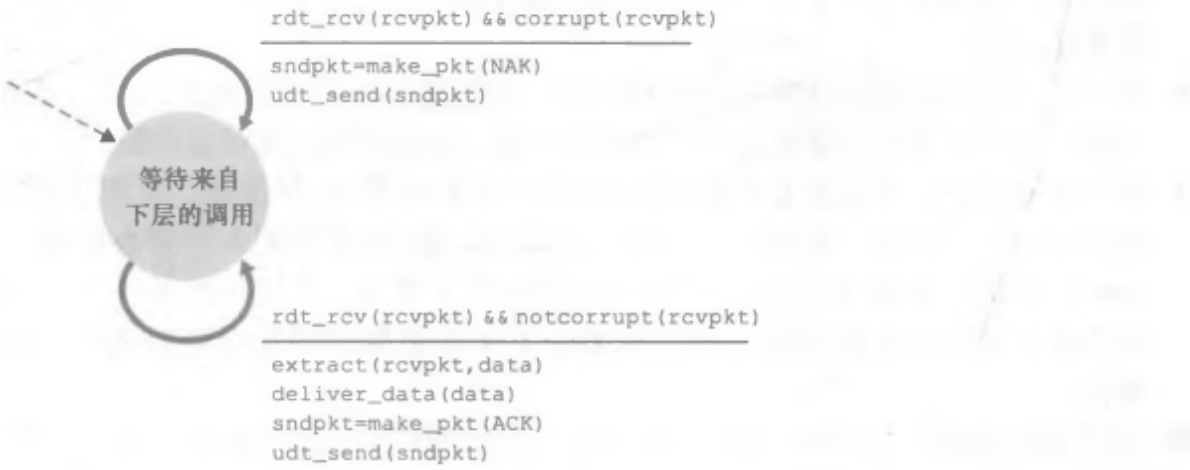
> 收到报文且没有错误：返回ACK
>
> 收到报文且有错误：重发请求

发送方：

> 上层调用发送：将数据打包发送
>
> 收到答复且返回NAK：重发且不能从上次获得数据
>
> 收到答复且返回ACK：等待下一次上层调用




a）rdt2.0: 发送端




## 4. rdt2.1

**停等协议**

为了解决ACK/NAK数据包损坏的问题，在数据分组中引入**序号**，ACK分为ACK0和ACK1，ACK1对ACK0是否收到进行确认，ACK0对ACK1是否收到进行确认。（二者地位相同，产生的ACK包为0-1-0-1-...，0和1之间进行交替）

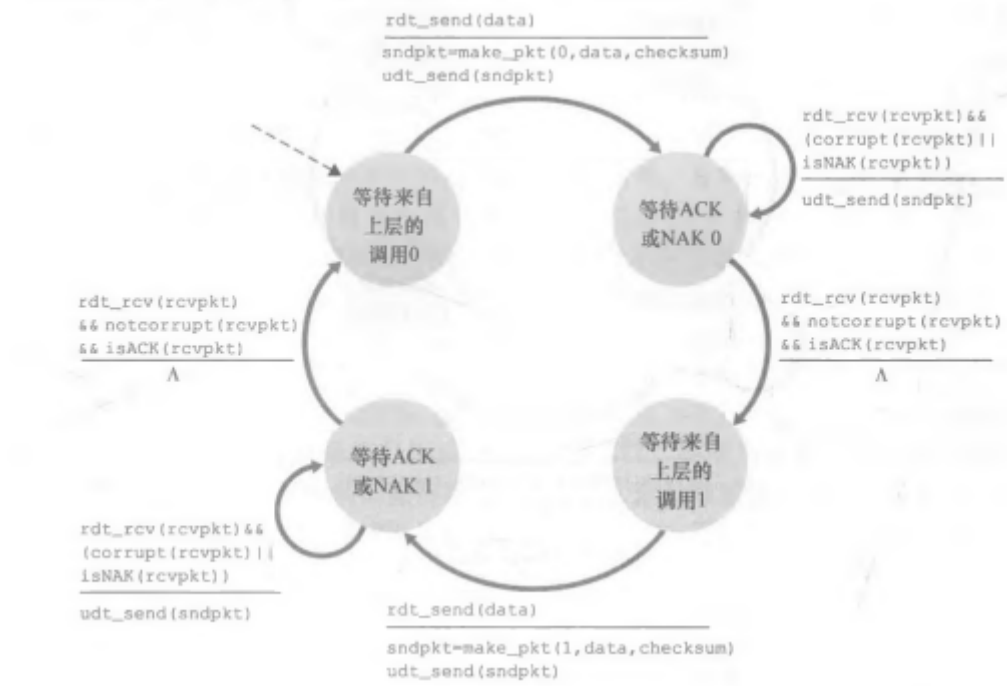
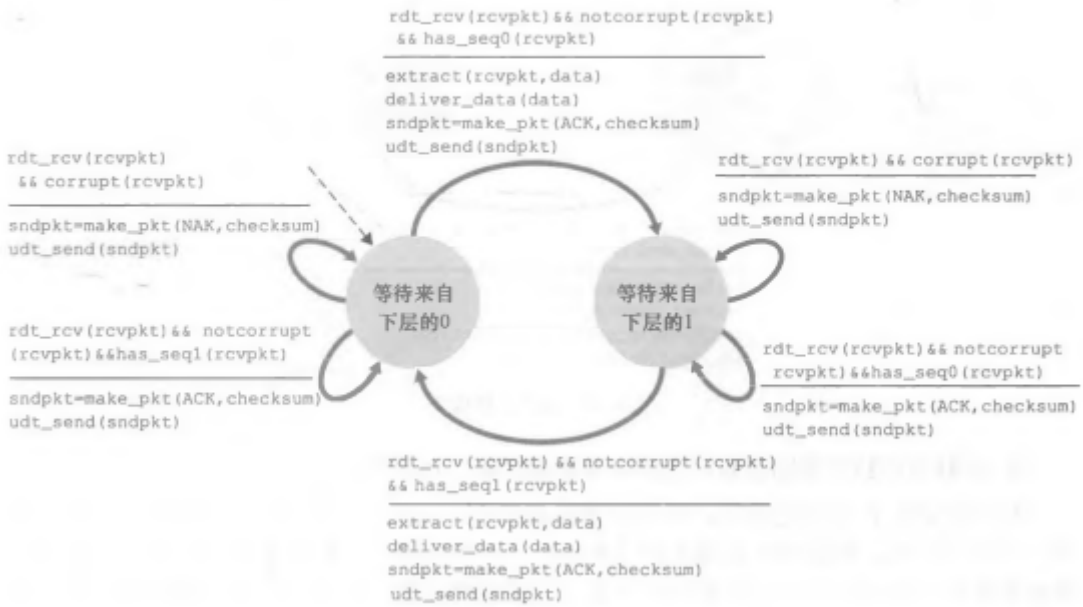



图 3-11　rdt2. 1 发送方




## 5. rdt2.2

无NAK的协议，对ACK编号。（仍然是01编号）

ACK0 = NAK1；ACK1 = NAK2;.....

与rdt2.1基本同理但是实现方式略有改进，在send函数中加入了参数1/0，指示需要重传/不需要重传

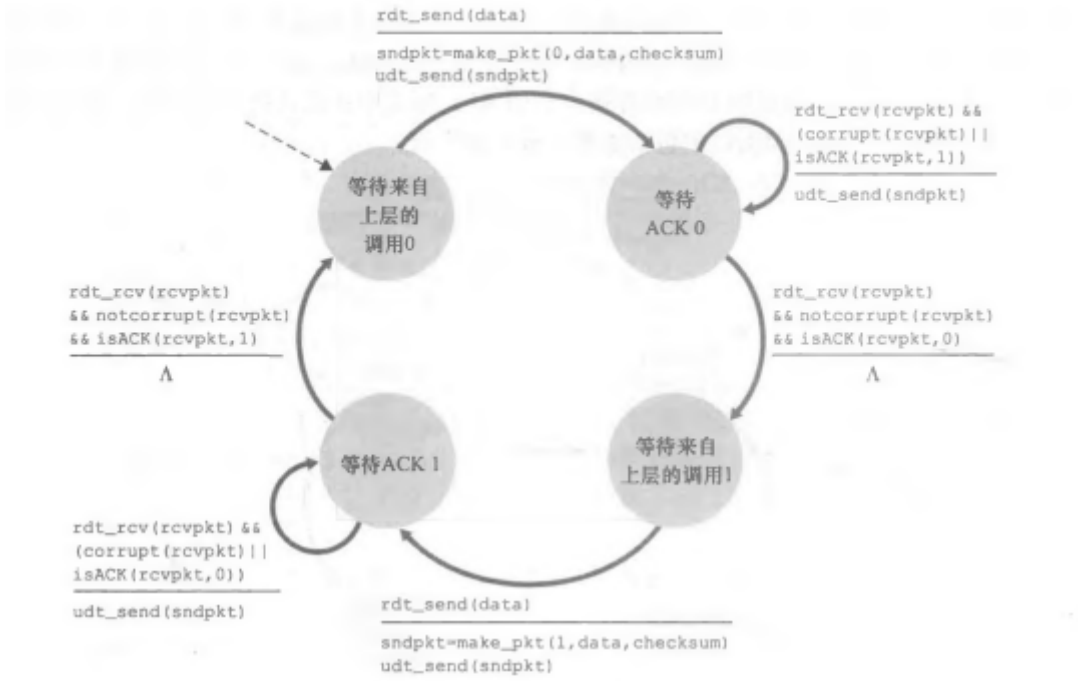
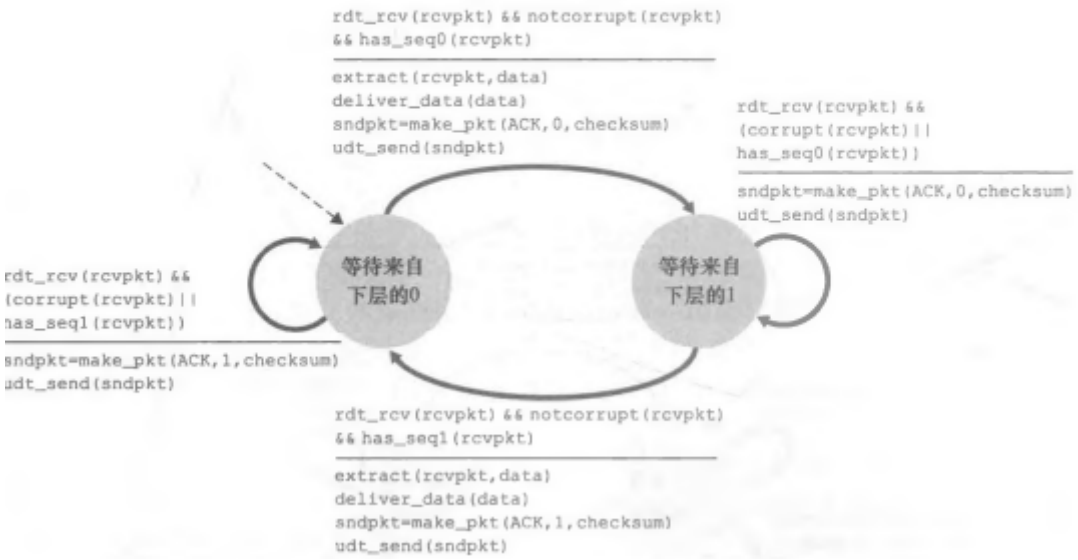



图 3-13　rdt2. 2 发送方



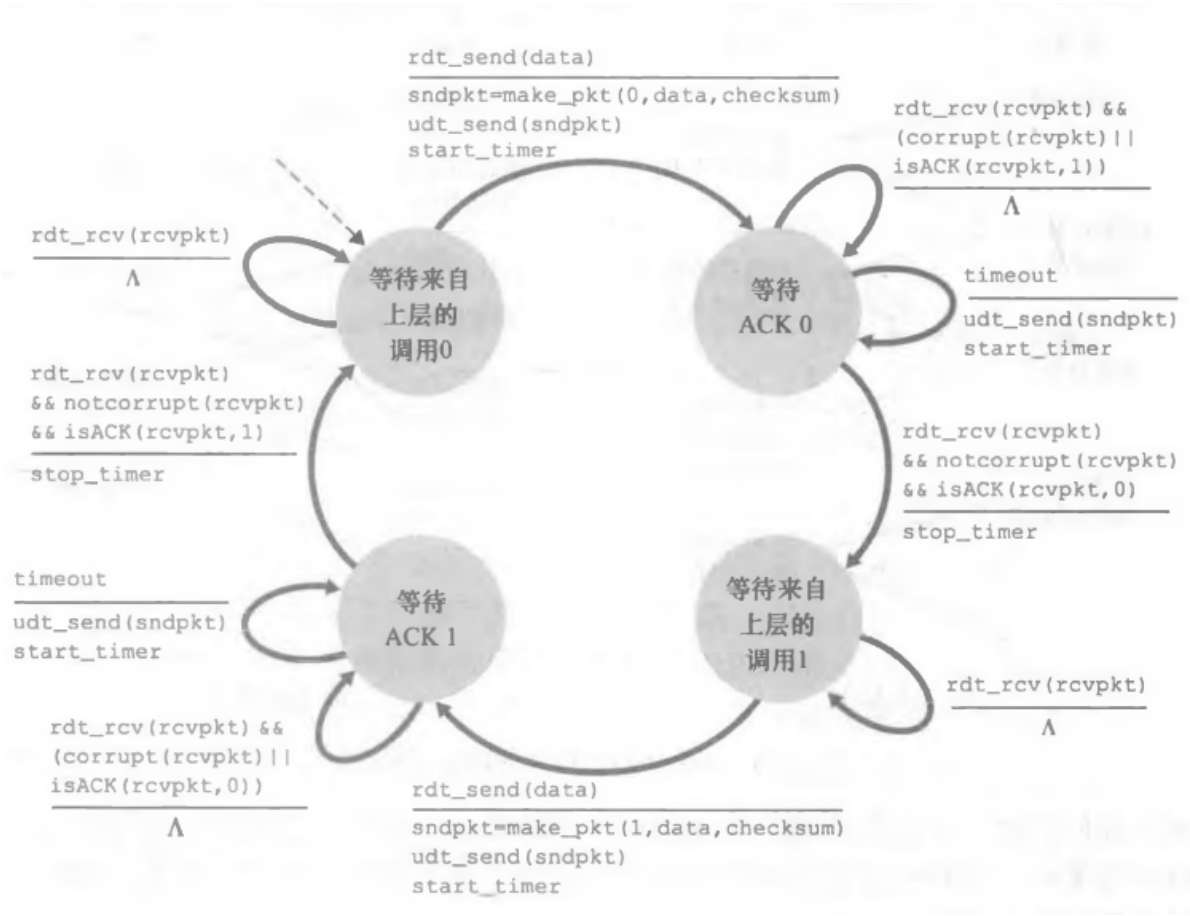

## 6. rdt3.0

又称为**比特交替协议**

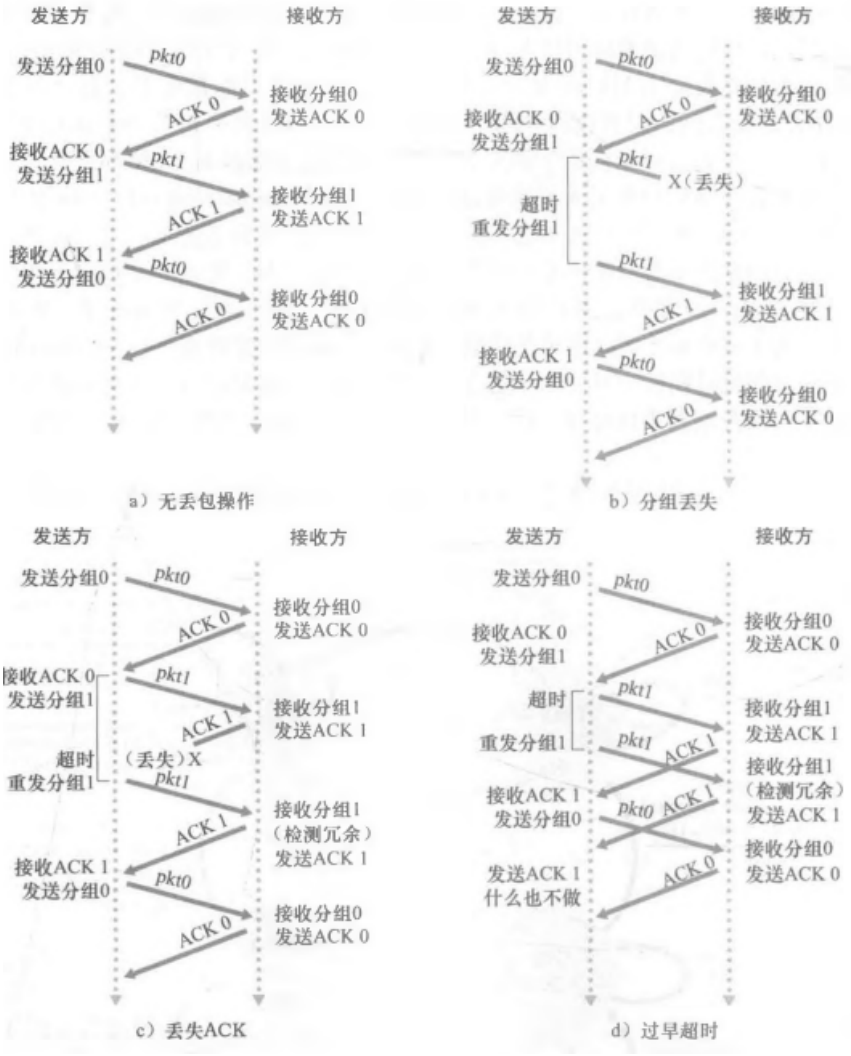
除了比特受损，底层信号还会丢包。

至少需要等待发送方和接收方之间的一个RTT的时间来确认包已丢失，需要倒计时定时器：

> 每发送一个分组，启动一个定时器
>
> 响应定时器中断
>
> 终止定时器


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
上层的
调用0
等待
ACK 0
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
timeout
udt_send(sndpkt)
start_timer
等待
ACK 1
等待来自
上层的
调用1
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运行原理：


发送方
接收方
发送分组0
pkt0
接收分组0
发送ACK 0
ACK 0
接收ACK 0
发送分组1
pkt1
接收分组1
发送ACK 1
ACK 1
接收ACK 1
发送分组0
pkt0
接收分组0
发送ACK 0
ACK 0
a）无丢包操作
发送方
接收方
发送分组0
pkt0
接收分组0
发送ACK 0
接收ACK 0
发送分组1
ACK 0
pkt1
X（丢失）
超时
重发分组1
pkt1
接收分组1
发送ACK 1
ACK 1
接收ACK 1
发送分组0
pkt0
接收分组0
发送ACK 0
ACK 0
b）分组丢失
发送方
接收方
发送分组0
pkt0
接收分组0
发送ACK 0
接收ACK 0
发送分组1
ACK 0
pkt1
接收分组1
发送ACK 1
超时
重发分组1
（丢失）X
pkt1
ACK 1
接收分组1
（检测冗余）
发送ACK 1
接收ACK 1
发送分组0
pkt0
接收分组0
发送ACK 0
ACK 0
c）丢失ACK
发送方
接收方
发送分组0
pkt0
接收分组0
发送ACK 0
接收ACK 0
发送分组1
ACK 0
pkt1
超时
重发分组1
接收分组1
发送ACK 1
pkt1
ACK 1
接收分组1
（检测冗余）
发送ACK 1
接收ACK 1
发送分组0
pkt0 ACK 1
接收分组0
发送ACK 0
发送ACK 1
什么也不做
ACK 0
d）过早超时

## 7. 流水线可靠传输协议

多个数据包一起发送，不需要确认收到后再发送第二个。

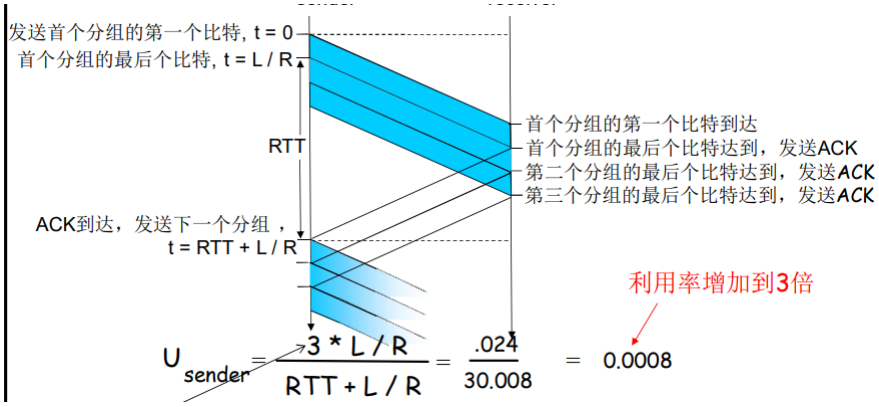
需要改进的地方（特点）：

- 增加序号范围，使得分组序号唯一化
- 发送方和接收方需要缓存多个分组
- 解决流水线差错的基本方法是**回退N步(GBN)**和**选择重传(SR)**

## 8. 利用率的计算

非持续连接：$\eta = \frac{L/R}{L/R+RTT}$

持续连接：$\eta = \frac{n \times L/R}{L/R+RTT}$，$n$为流水线支持的并发发送数（不能超过100%）




## 9. 滑动窗口协议（通用）

发送缓冲区：存储已发送，但是未确认的分组。

缓冲区的大小：停等协议为1，流水线大于1

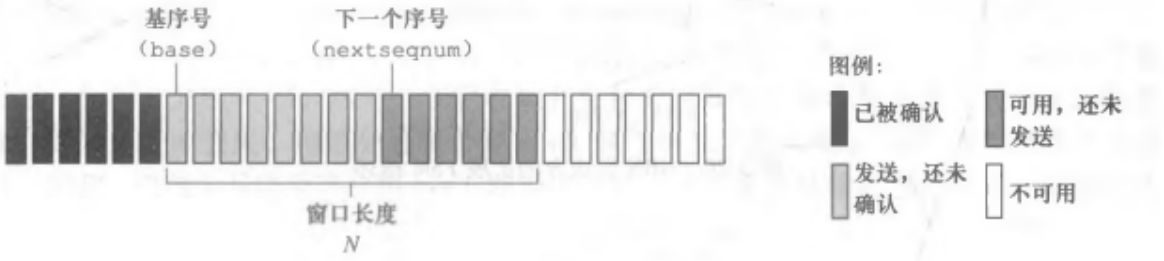
使用前沿和后沿来维护发送窗口，同时还存在接收窗口

> 初始状态下，前沿等于后沿
>
> 发送一个分组，前沿移动一步（不能超过窗口极限）
>
> 接收窗口内的分组确认时，后沿随之移动
>
> > 接收窗口大小为1时，只能顺序接收
> >
> > 大于1时，可以乱序接收
>
> 滑动：低序号分组到来，移动窗口，高序号分组到来，缓存但不交付不滑动。（不允许失序）

## 10. 回退N步（GBN）

流水线中的分组数不能超过某个最大允许值N，接收窗口尺寸为1。

将序列号分为四段：




随着协议的运行，窗口向前滑动。

k位序号的序号空间看成一个长度为$2^k$的环，使用$(i+1+2^k)\%2^k$更新窗口。

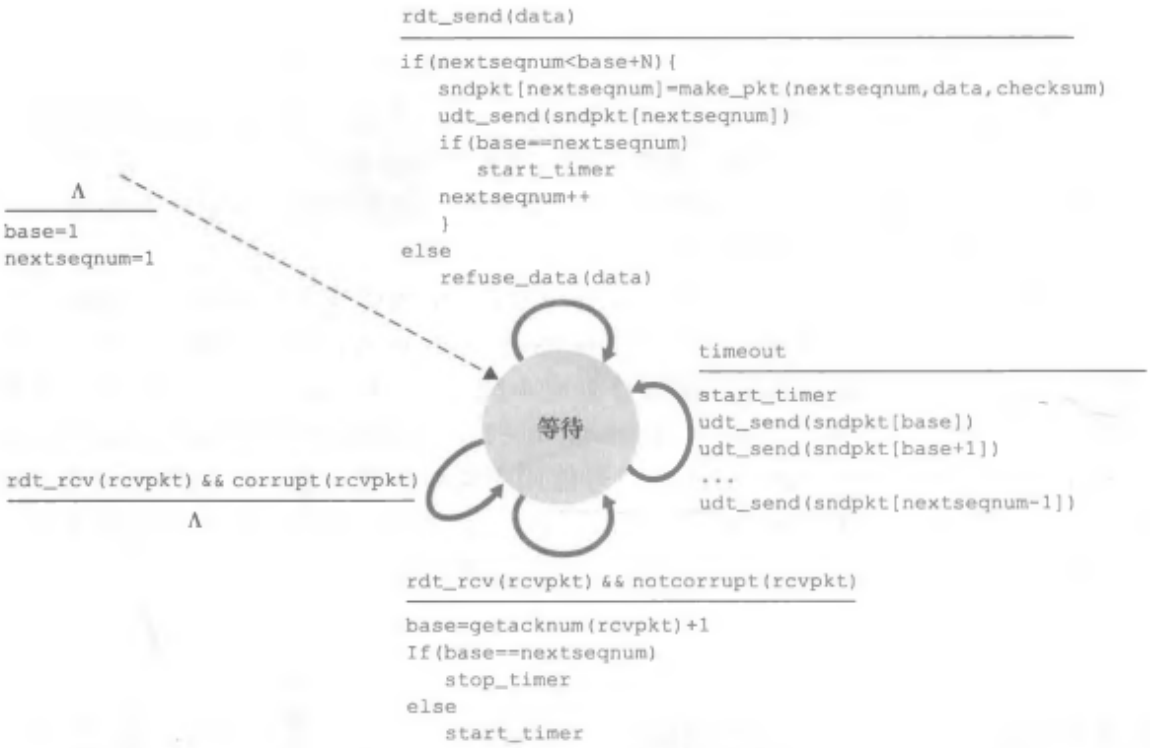
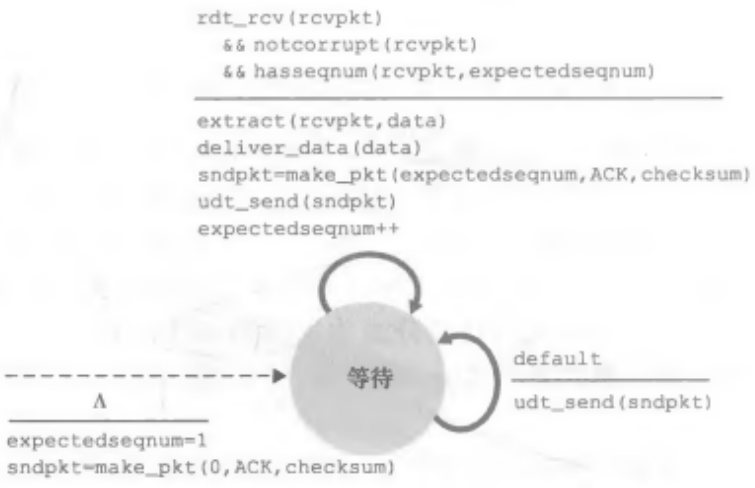



图 3-20　GBN 发送方的扩展 FSM 描述




- 上层调用rdt_send()时，发送方先检查窗口是否已满
  - 未满：产生一个分组并将其发送，nextsetnum ++
  - 满：等待

- 收到一个ACK，使用**累积确认**的方式，表明接收方已经正确接收到序号为n的以前包括n在内的所有分组
- 超时：如果收到一个ACK；重新发送base到nextsetnum-1标号的分组（所有发送窗口内未确认的分组）
    - 但仍有已发送但未被确认的分组，重启计时器
    - 如果没有已发送但未确认的分组，停止计时器

GBN运行示意图：

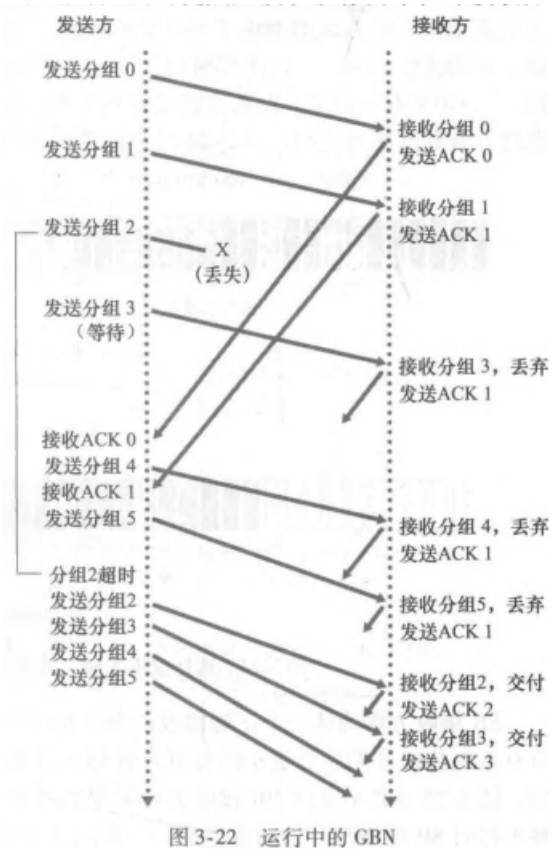


图 3-22　运行中的 GBN

GBN会丢弃失序分组。

## 11. 选择重传（SR）

GBN的问题：窗口长度和带宽时延积过大时，单个分组出错导致重传大量分组，同样使用长度为N的滑动窗口来限制流水线中未完成、未被确认的分组。

SR的接收窗口尺寸大于1。

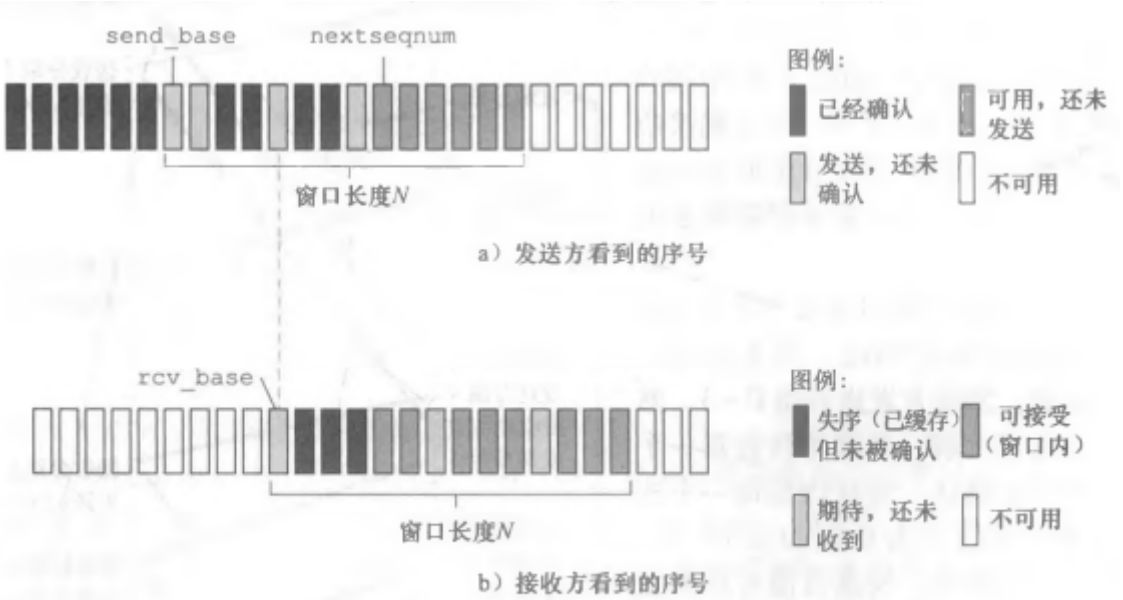



图 3-23　选择重传（SR）发送方与接收方的序号空间

失序的分组将被缓存直到所有丢失分组都被收到为止，这时才可以将一批分组按序交付给上层。

1. 从上层收到数据。当从上层接收到数据后，SR 发送方检查下一个可用于该分组的序号。如果序号位于发送方的窗口内，则将数据打包并发送；否则就像在 GBN 中一样，要么将数据缓存，要么将其返回给上层以便以后传输。

2. 超时。定时器再次被用来防止丢失分组。然而，现在每个分组必须拥有其自己的逻辑定时器，因为超时发生后只能发送一个分组。可以使用单个硬件定时器模拟多个逻辑定时器的操作［Varghese 1997］。

3. 收到 ACK。如果收到 ACK，倘若该分组序号在窗口内，则 SR 发送方将那个被确认的分组标记为已接收。如果该分组的序号等于 send_base，则窗口基序号向前移动到具有最小序号的未确认分组处。如果窗口移动了并且有序号落在窗口内的未发送分组，则发送这些分组。

图 3-24　SR 发送方的事件与动作

1. 序号在［rcv_base，rcv_base + $N$ − 1］内的分组被正确接收。在此情况下，收到的分组落在接收方的窗口内，一个选择 ACK 被回送给发送方。如果该分组以前没收到过，则缓存该分组。如果该分组的序号等于接收窗口的基序号（图 3-23 中的 rcv_base），则该分组以及以前缓存的序号连续的（起始于 rcv_base 的）分组交付给上层。然后，接收窗口按向前移动分组的编号向上交付这些分组。举例子来说，考虑一下图 3-26。当收到一个序号为 rcv_base = 2的分组时，该分组及分组 3、4、5 可被交付给上层。

2. 序号在［rcv_base − $N$，rcv_base − 1］内的分组被正确收到。在此情况下，必须产生一个 ACK，即使该分组是接收方以前已确认过的分组。

3. 其他情况。忽略该分组。

图 3-25　SR 接收方的事件与动作

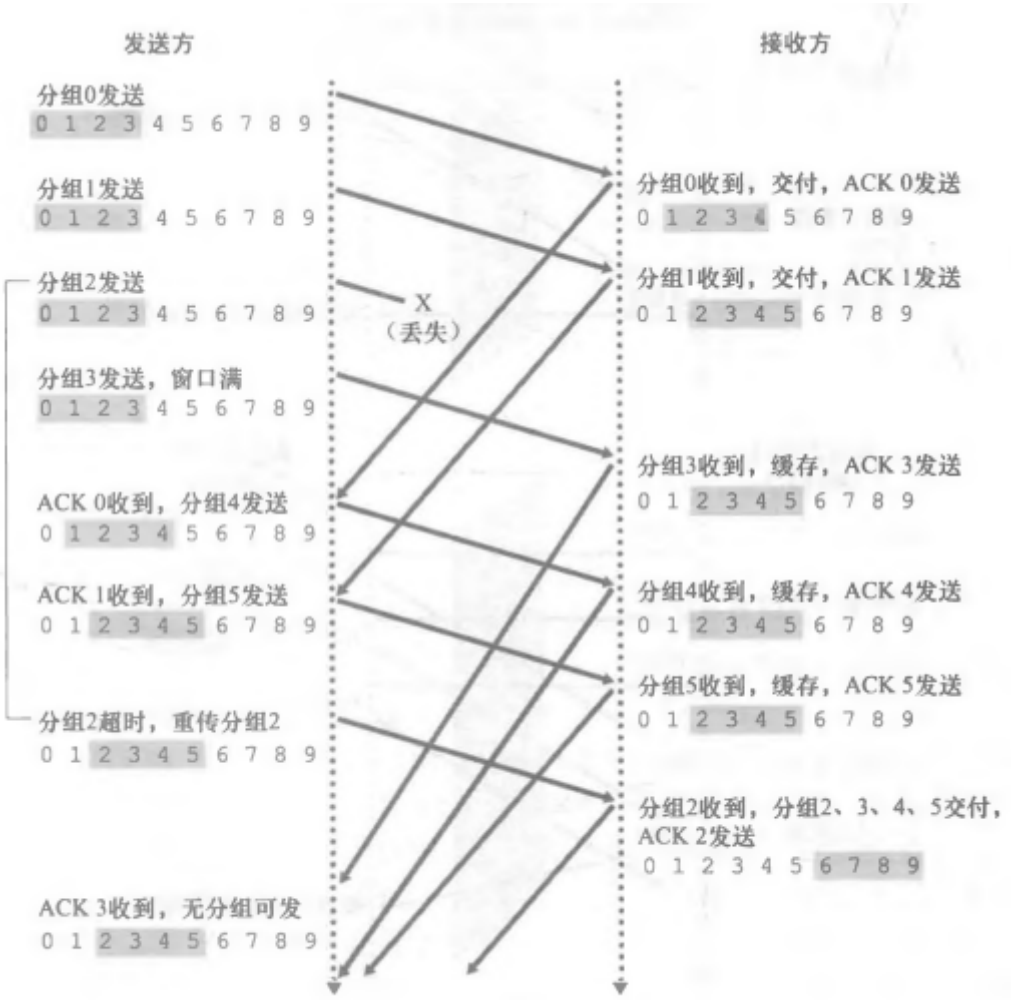



图 3-26　SR 操作

窗口大小不能超过缓冲区的一半因为：

> 假设序号空间大小为2m - 1，窗口长度为m，则当发送方发送m个分组时，假设其中第一个分组发送失败，未收到ACK，则发送剩余m-1个分组时不会移动窗口，若此时收到了第一个分组的ACK，则窗口需要移动m - 1个位置，导致越界或产生重复序号，导致服务器无法判断是重传分组还是新分组

### 12. GBN和SR的对比

ACK序号：$0 \sim 2^n$，即n位二进制码

GBN：简单、所需资源少，回退代价大，窗口的最大尺寸为$2^n - 1$

SR：重传代价小，复杂，所需资源多，窗口的最大尺寸为$2^{n-1}$

## 五、TCP

### 1. TCP简介

可靠的传输服务。

提供的服务：流量控制、拥塞控制、面向连接

不提供的服务：时间保证、最小吞吐保证和安全

**SSL**：基于TCP实现的安全可靠数据传输

## 2. TCP连接

TCP连接是点对点的。

> 客户端发送一个特殊的TCP报文段，服务器用另一个特殊的TCP报文段来响应（这两个报文段不包含应用层数据）
>
> 客户再用第三个特殊报文段作为响应，可以承载应用层数据
>
> 这个过程称为“三次握手”

发送缓存：客户端TCP发送数据使用的缓存

**最大报文长度（MSS）：**从缓存中去除并放入报文段中的数据数量限制

## 3. TCP报文段

序号和确认号对字节计数，而不是报文段。

> 序号指示首字节在字节流的编号
>
> 确认号确认n-1及以前字节（累计确认）。
>
> 例如：HostA(seq = 42, ack=79)$\rightarrow$HostB(seq = 79, ack = 43)$\rightarrow$HostA(seq = 43, ack = 80)$\rightarrow\cdots$

RST/SYN/FIN：建立/拆除/连接

## 4. TCP往返延时和超时

超时：大于RTT（短：不必要的重传，长：反应慢）

RTT估计

> SampleRTT：测量从报文段发出到确认的时间（重传则忽略）
>
> 对最近几个SampleRTT求平均，而不是仅仅使用当前的SampleRTT

**估计公式：**

$$EstimatedRTT_{i+1} = (1-\alpha)EstimatedRTT_i + \alpha SampleRTT$$

$\alpha$一般取0.125

**安全边界时间公式：**

$$DevRTT_{i+1} = (1-\beta)DevRTT_i + \beta \times |SampleRTT_i - EstimatedRTT_i|$$

$\beta$一般取0.25

**超时公式：**

$$TimeoutInterval = EstimatedRTT + 4 \times DevRTT$$

## 5. 可靠数据传输

- 在IP不可靠的基础上建立rdt（使用管道、累积确认、单个重传计时器、没有规范乱序判断）
- 超时和重复确认触发重传

运行机制：（只有一个状态）

> 从应用层接收到消息：创建报文段，发送至IP层，如果此时计时器没有启动，启动计时器。
>
> 如果超时：重传未确认的最小序号的报文段，重新启动计时器

> 收到ACK：如果在发送窗口内，将其设置为已确认状态
>
> - 如果此时有未确认的报文段：启动计时器
> - 否则关闭计时器

```
def SimpleTCPSender():
    NextSeqNum = InitSeqNum()
    SendBase = InitialSeqNum()
    timer = Time.time()
    while 1:
        if((data = FromApplicationAbove()) != -1):
            seg = CreateTCPseg(NextSeqNum)
            if(timer.NotWork()):
                timer.start()
            TCP.sendtoIP(seg)
            NextSeqNum = NextSeqNum + len(data)
        elif(timer.timeout()):
            seg = GetMinNAK()
            TCP.sendtoTP(seg)
            timer.start()
        elif((ack = ReceivedFromClient()) != -1):
            if(ack > SendBase):
                SendBase = ack
                if(ExistsNAKSegs):
                    timer.start()
    #end of def
```

## 6. 快速重传

- 超时周期太长
- 通过重复ACK检测报文段丢失
- 收到3条冗余ACK，重传最小序号的段
  - 快速重传：定时器过时之前快速重发报文段
  - 假设被确认的数据后的数据丢失

## 7. 流量控制

流量控制：接收方控制发送方，不让其发送的太多、太快导致缓冲区溢出。

接收方通过在TCP段的头部rwnd字段向发送方通告剩余buffer大小。

## 8. 连接管理

两次握手不可行：连接请求的段超时、丢失造成重传、报文乱序、看不见对方，即半连接和接收老数据问题

请求建立连接→服务器回应表明自己活跃→客户端回应表明自己活跃

客户端(SYN = 1, Seq = x)→服务器(SYN = 1, seq = y，ACK = x + 1)→客户端(ACK = y + 1)

**关闭TCP连接：**

让FIN字段为1，发送段，服务器回应，然后断开TCP连接。

客户端(FIN = 1, Seq = x)→服务器(ACK = x + 1)→服务器(FIN = 1, Seq = y)→客户端(ACK = y + 1)→连接关闭

# 六、TCP的拥塞控制

## 1. 拥塞控制简介

拥塞非正式定义：数据传输需求超过了网络负载能力

拥塞的表现：分组丢失、分组延时过长

现实情况：重复+丢失，没必要的重传导致有效输出率降低

## 2. 拥塞控制方法

**端到端拥塞控制**：

- 没有来自网络的显示反馈
- 端系统根据延迟和丢失事件判断是否拥塞
- TCP采用的方法

**网络辅助的拥塞控制：**

- 路由器提供给端系统反馈信息
- 显示提供发送端可以采用的速率

## 3. TCP拥塞控制原理

cwnd即CongWin

**拥塞感知：**超时事件（拥塞）、三次重复ACK（轻微拥塞）

**速率控制**：维护拥塞窗口（cwnd），发送端限制已发送但是未确认的数据量

- 超时：cwnd = 1MSS，进入慢启动(SS)状态再倍增。
- 三次重复ACK：cwnd /= 2，进入拥塞避免(CA)阶段。
- 正常收到ACK，加倍增加(SS)或线性增加(CA)

**联合控制**：SendWin  = min{cwnd, RecvWin}，同时满足拥塞控制和流量控制要求

## 4. 慢启动

超时或复位时发生

建立连接后，cwnd = 1MSS，指数增加cwnd：（无超时和重复ACK）

> 每一个RTT，cwnd *= 2
>
> 每一个ACK，cwnd += 1

达到阈值（上次发生拥塞的窗口大小ssthresh）的一半时，进入拥塞避免阶段。

## 5. 拥塞避免

cwnd > Threshold时发生

> 每一个RTT，cwnd += 1

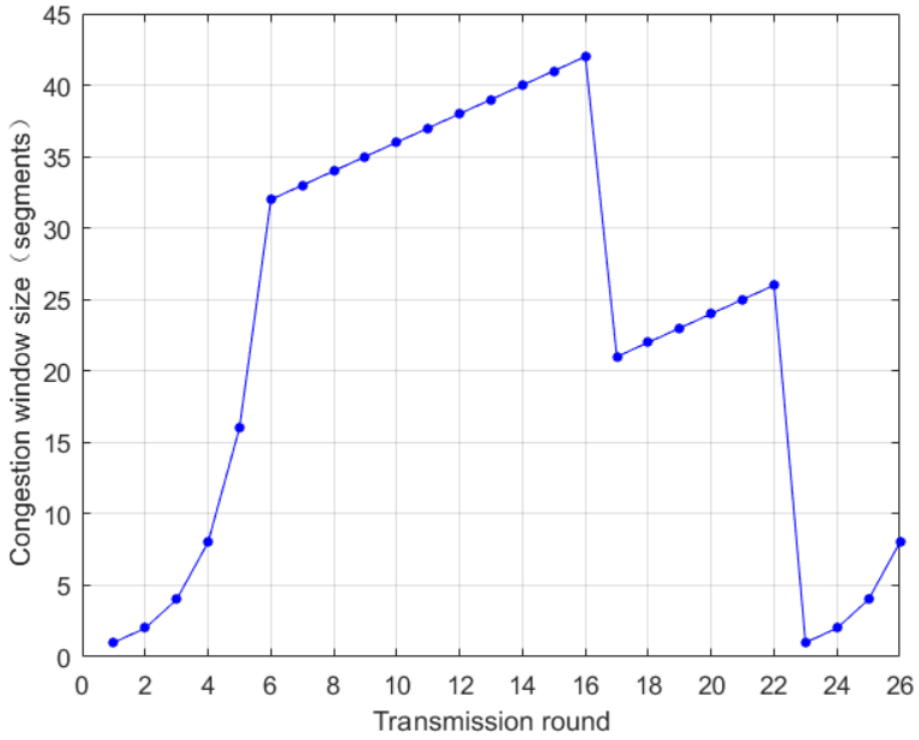
## 6. 快速重传

> ssthresh设置为cwnd / 2
>
> cwnd = ssthresh + 3

## 7. AIMD

**乘性减：**丢失事件后将cwnd降为1，将cwnd0作为阈值，进入慢启动阶段

**加性增：**当cwnd大于阈值时，一个RTT若没有丢失事件，cwnd增加1MSS。




| 事件 | 状态 | TCP 发送端行为 | 解释 |
| --- | --- | --- | --- |
| 以前没有收到ACK的data被ACKed | 慢启动(SS) | CongWin = CongWin + MSS<br>If (CongWin > Threshold)<br>状态变成“CA” | 每一个RTT CongWin 加倍 |
| 以前没有收到ACK的data被ACKed | 拥塞避免 (CA) | CongWin = CongWin+MSS * (MSS/CongWin) | 加性增加, 每一个RTT对CongWin 加一个 1 MSS |
| 通过收到3个重复的ACK，发现丢失的事件 | SS or CA | Threshold = CongWin/2,<br>CongWin = Threshold+3,<br>状态变成“CA” | 快速重传, 实现乘性的减. CongWin 没有变成1 MSS. |
| 超时 | SS or CA | Threshold = CongWin/2,<br>CongWin = 1 MSS,<br>状态变成“SS” | 进入slow start |
| 重复的 ACK | SS or CA | 对被ACKed 的segment，增加重复ACK的计数 | CongWin and Threshold 不变 |

## 8. TCP吞吐量

$\frac{W_{avg}}{RTT}$

## 七、常用应用功能使用的传输层协议

| 应用 | 应用层协议 | 下面的运输协议 |
| --- | --- | --- |
| 电子邮件 | SMTP | TCP |
| 远程终端访问 | Telnet | TCP |
| Web | HTTP | TCP |
| 文件传输 | FTP | TCP |
| 远程文件服务器 | NFS | 通常 UDP |
| 流式多媒体 | 通常专用 | UDP 或 TCP |
| 因特网电话 | 通常专用 | UDP 或 TCP |
| 网络管理 | SNMP | 通常 UDP |
| 名字转换 | DNS | 通常 UDP |

图 3-6　流行的因特网应用及其下面的运输协议