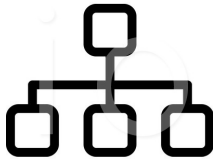
计算机网络

Chaper 3 The Data Link Layer

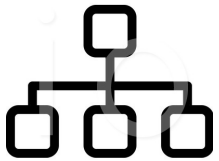
何　军

hejun@ruc.edu.cn

课程内容



第3章 数据链路层



3.1 定义和功能

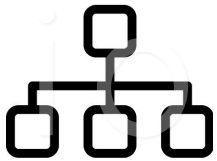
3.2 错误检测和纠正

3.3 基本的数据链路层协议

3.4 滑动窗口协议

3.5 常用的数据链路层协议

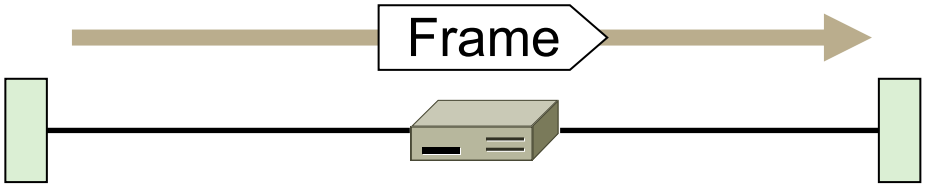
3.1 定义和功能



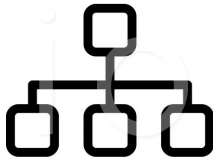
- 定义
- 服务
- 成帧
- 差错控制
- 流量控制

3.1.1 定义

- 要解决的问题
  - 如何在有差错的线路上，进行无差错传输
- ISO关于数据链路层的定义
  - 数据链路层的目的是为了提供功能上和规程上的方法，以便建立、维护和释放网络实体间的数据链路
- Scope of the Link Layer
  - Concerns how to transfer messages over one or more connected links
    - Messages are frames, of limited size
    - Builds on the physical layer

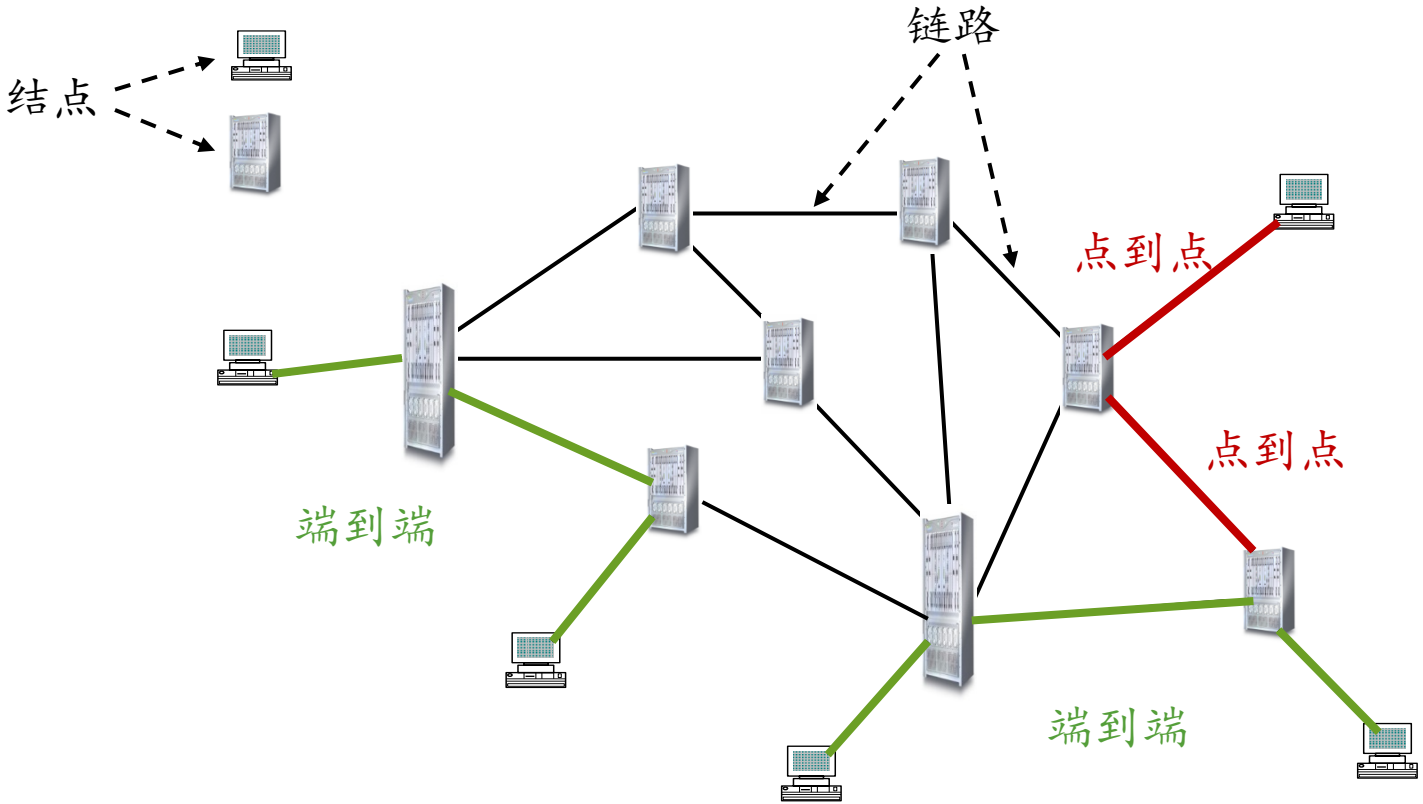


基本概念

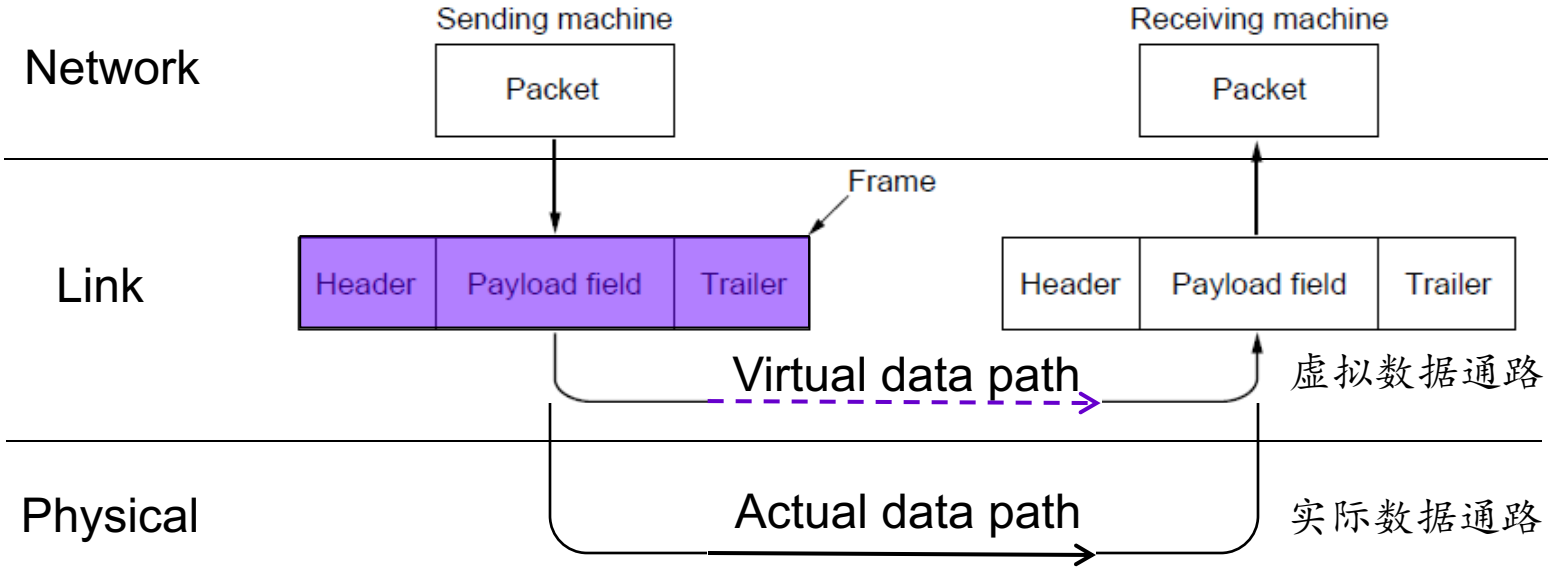


- **结点**（node）：网络中的主机（host）和路由器（router）称为结点
- **链路**（link）：通信路径上连接相邻结点的通信信道称为链路
  - 数据链路层协议定义了一条链路的两个结点间交换的数据单元格式，以及结点发送和接收数据单元的动作
- **端到端**（end to end）：从源结点（source node）到目的结点（destination node）的通信称为端到端通信，通信路径（path）可能由多个链路组成
- **点到点**（point to point）：在相邻结点间的一条链路上的通信称为点到点通信
- **虚拟数据通路**
- **实际数据通路**

基本概念 (2)



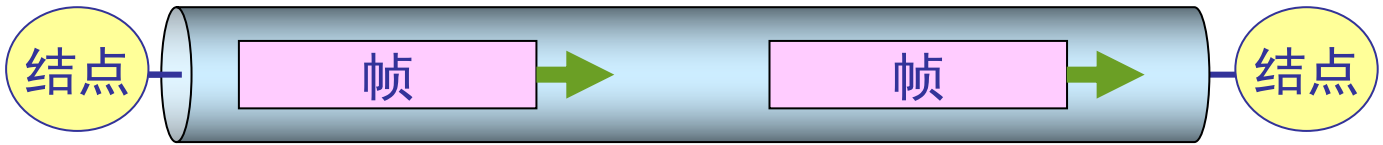
基本概念 (3)


Network
Sending machine
Packet
Receiving machine
Packet
Frame
Link
Header
Payload field
Trailer
Header
Payload field
Trailer
Virtual data path
虚拟数据通路
Physical
Actual data path
实际数据通路

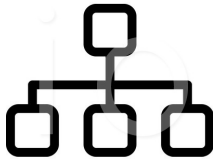
数据链路协议

- 数据链路控制规程（协议）
  - 为使数据能迅速、正确、有效地从发送点到达接收点所采用的控制方式
  - 早期的数据通信协议曾叫作通信规程(procedure)，因此规程和协议是同义语
- 数据链路层协议应提供的最基本功能
  - 数据在数据链路上的正常传输(建立、维护和释放)
  - 定界与同步，也处理透明性问题
  - 差错控制、顺序控制、流量控制

在两个对等的数据链路层之间画出一个数字管道，而在这条数字管道上传输的数据单位就是帧

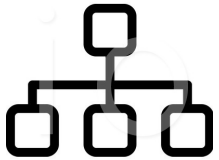


数据链路



- 链路(link)
  - 是一条无源的点到点的物理线路段，中间没有任何其他的交换结点
  - 一条链路只是一条通路的一个组成部分
- 数据链路(data link)
  - 除了物理线路外，还必须有通信协议来控制这些数据的传输。若把实现这些协议的硬件和软件加到链路上，就构成了数据链路
  - 现在最常用的方法是使用适配器（即网卡）来实现这些协议的硬件和软件
  - 一般的适配器都包括了数据链路层和物理层这两层的功能

数据链路层的主要功能点



(1) 链路管理

(2) 帧定界

(3) 流量控制

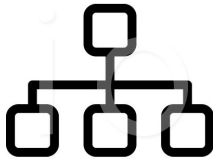
(4) 差错控制

(5) 将数据和控制信息区分开

(6) 透明传输

(7) 寻址

接下来的内容

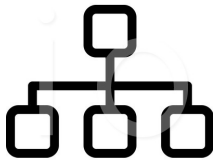


1. Framing
   - Delimiting start/end of frames
2. Error detection and correction
   - Handling errors

3. Retransmissions
   - Handling loss
4. Multiple Access
   - 802.11, classic Ethernet
5. Switching
   - Modern Ethernet

3.3-3.4，第4章

3.1.2 服务

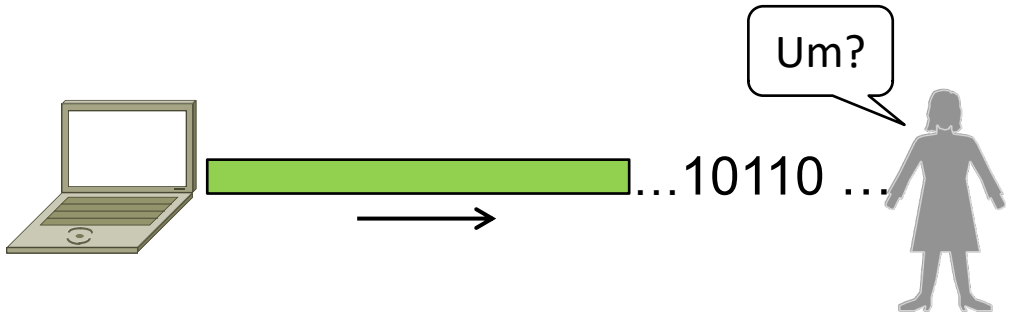


为网络层提供三种合理的服务

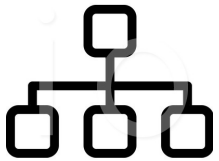
- 无确认无连接服务
  - 发帧 — 完成
  - 适用于：误码率很低的线路，错误恢复留给高层；实时业务；大部分局域网
- 有确认无连接服务
  - 发帧 — 确认 — 无确认重发
  - 适用于不可靠的信道，如无线网。
- 有确认面向连接服务
  - 连接 — 发送 — 释放

3.1.3 成帧

- The Physical layer gives us a stream of bits. How do we interpret it as a sequence of frames?



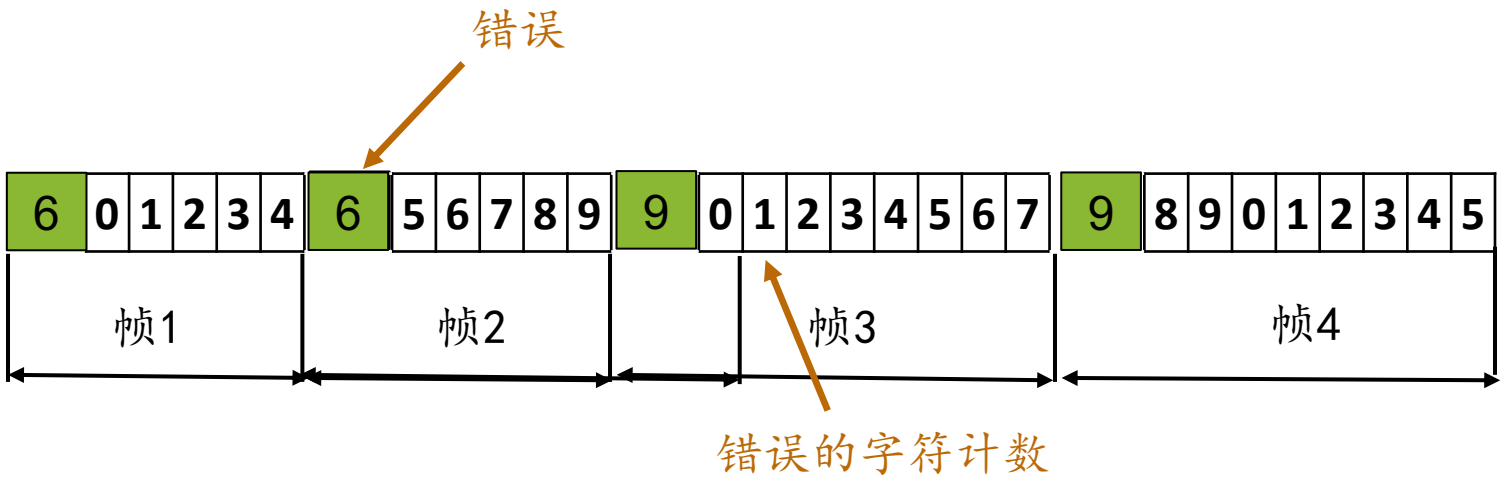
常用的成帧方法



- 将比特流分成离散的帧，并计算每个帧的校验和
- 成帧方法：
  - 字符计数法
  - 带字符填充的首尾字符定界法
  - 带位填充的首尾标记定界法
  - 物理层编码违例法
- 注意
  - 在很多数据链路协议中，使用字符计数法和一种其它方法的组合

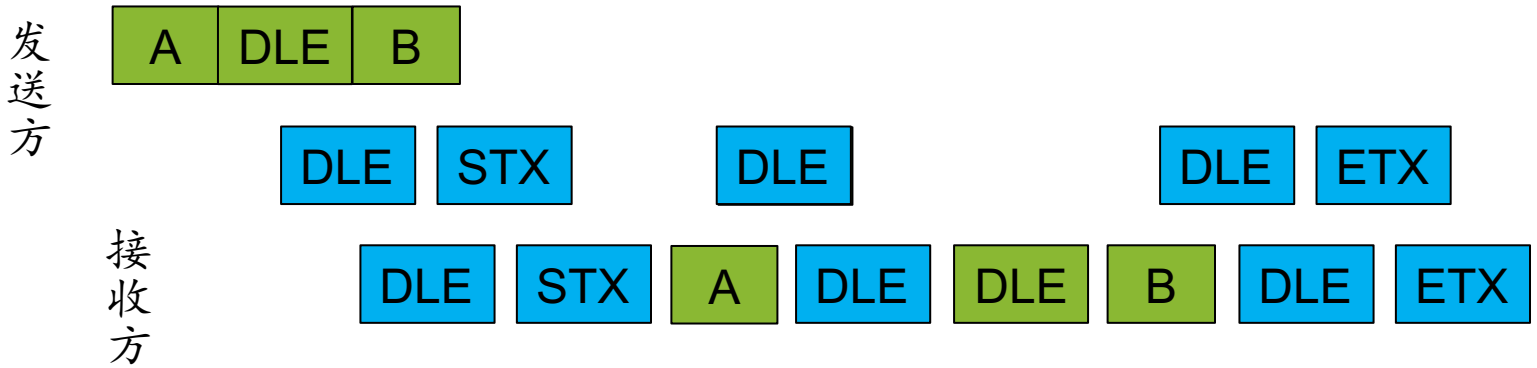
字符计数法

- 在帧头中用一个域来表示整个帧的字符个数
- 缺点：若计数出错，对本帧和后面的帧有影响



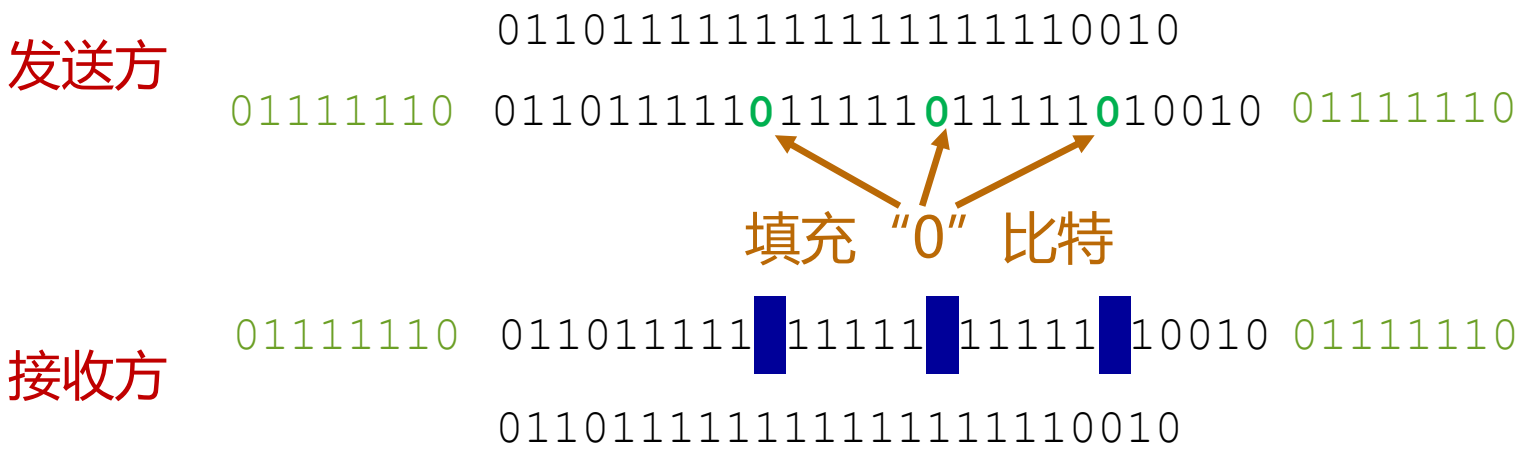
带字符填充的首尾字符定界法

- 起始字符 DLE STX，结束字符DLE ETX
  - DLE : Data Link Escape；STX : Start of Text；ETX : End of Text
- 字符填充：数据中出现标志字符时插入转义字符
- 缺点：局限于8位字符和ASCII字符传送



带位填充的首尾标记定界法

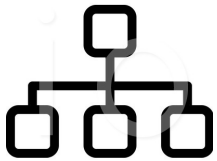
- 帧的起始和结束都用一个特殊的位串“01111110”，称为标记(flag)
- “0”比特插入删除技术



物理层编码违例法

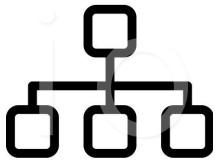
- 只适用于物理层编码有冗余的网络
  - 利用一些不会出现在常规数据中保留的信号来指示帧的开始和结束，即使用”编码违法”来区分帧的边界。
- 802 LAN中
  - 曼彻斯特编码或差分曼彻斯特编码，用high-low pair/low-high pair表示1/0，high-high/low-low不表示数据，可以用来做定界符

3.1.4 差错控制



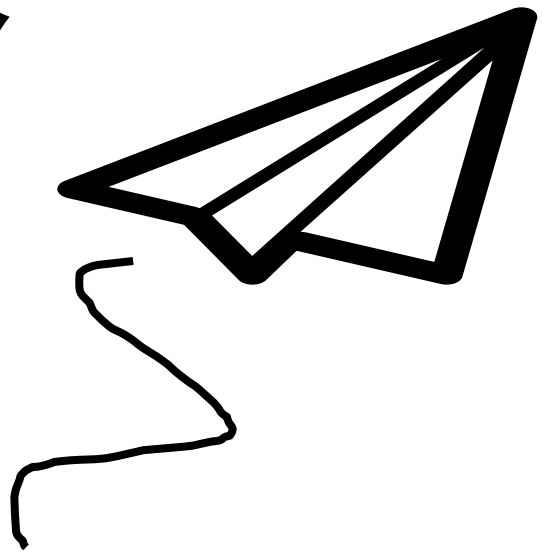
- 一般方法
  - 接收方给发送方一个反馈（响应）
- 出错情况
  - 帧（包括发送帧和响应帧）出错
  - 帧（包括发送帧和响应帧）丢失
- 通过计时器和序号，保证每帧最终交给目的网络层仅一次，是数据链路层的一个主要功能
  - 后续章节将介绍如何使用计时器和序号来保证链路层的正确

3.1.5 流量控制

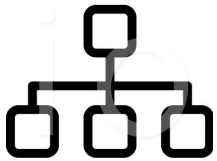


- 基于反馈机制
  - 非允许—不发帧
  - 发方的发送受接收方的控制
- 基于速率控制
  - 协议的设计内置了一种机制，它能限制发送方的速率，而无须利用接收方的反馈
  - 仅在传输层流量控制中使用

本节课程结束

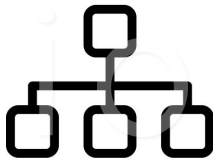


3.2 错误检测和纠正



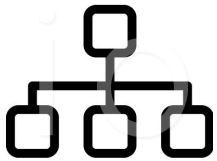
- 差错特点与处理策略
  - 随机、连续突发（burst）
  - 使用纠错码
  - 使用检错码
- 纠错码
  - 发送方在每个数据块中加入足够的冗余信息，使得接收方能够判断接收到的数据是否有错，并能纠正错误
- 检错码
  - 发送方在每个数据块中加入足够的冗余信息，使得接收方能够判断接收到的数据是否有错，但不能判断哪里有错

3.2.1 纠错码



- 码字(codeword)
  - 一个帧包括 m 个数据位，r 个校验位（ $m + r = n$ ），则此 n 比特单元称为 n 位码字
  - n (码字) = m (数据) + r (冗余位)
  - 对于 n 位码字的集合，只有 2^m 个码字是有效的。也就是说，通常并未使用所有 2^n 个码字

海明距离

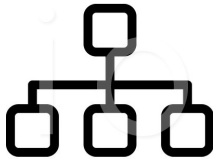


- 海明距离 (Hamming distance)
  - 两个码字中不同比特的个数
  - 例：0000000000 与 0000011111的海明距离为5
  - 如果两个码字的海明距离为d，则只需d个单比特错，就可以把一个码字转换成另一个码字

	检测d比特错	纠正d比特错
编码的海明距离	$d + 1$	$2d + 1$

  - 思考：海明距离为5，能检测几位错，能纠正几位错？

检错方法



- Some bits may be received in error due to noise. How do we detect this?
  - Parity »
  - Checksums »
  - CRCs »
- Detection will let us fix the error, for example, by retransmission (later).

奇偶校验

- 最简单的检错方法是奇偶校验，在数据后填加一个奇偶位

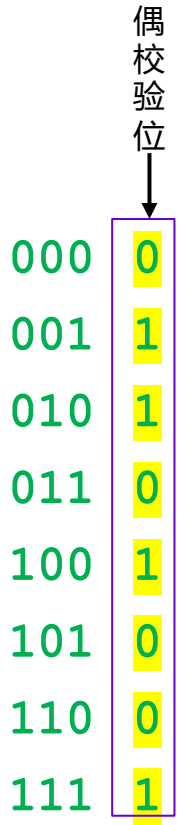
- 检测能力：只能检测出奇数个错误

例1：使用偶校验（“1”的个数为偶数）

10110101——>10110101 1

10110001——>10110001 0

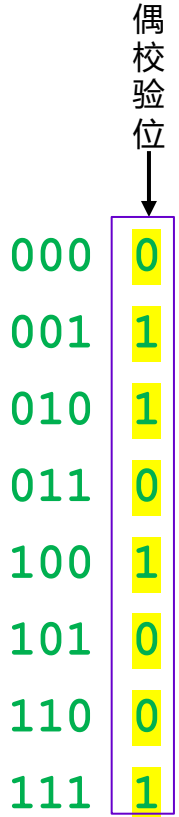
例2：右侧前三个数据位，后一个是冗余的奇偶校验位。



检错

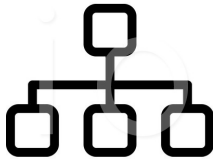
- 在任意两个有效码字间找出具有最小海明距离的两个码字，该海明距离便定义为全部码字的海明距离
- 一种编码的检错和纠错能力取决于编码后码字海明距离的大小
- 为了检测出d个比特的错，需要使用海明距离为d+1的编码
  - 如果数据后加奇偶校验位，编码后海明距离为2，则能检测1比特错

例：右侧示例编码的海明距离为2



说明：右边的编码共有8个有效的码字，你会发现任意两个码字之间最小的不同位数是两位，即海明距离为2，这意味着任何一个有效的码字需要改变两位才能变成另一个有效的码字。当出现一位错误时，接收方看到的比特串不会是这8个（有效）码字中的任何一个，因而知道一定出现了传输错误。

纠错

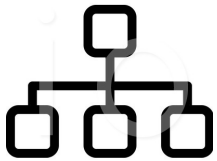


- 为了纠正d个比特的错，必须用海明距离为2d+1的编码

例：有4个有效码字，它们是0000000000，0000011111，1111100000，1111111111，海明距离为5，能纠正2比特错

0000000000	0000000001	0000000011	0000000111	0000001111	0000011111

设计纠错码



- 纠正单比特错的纠错码
  - 要求：m个信息位，r个校验位，当r满足什么条件时，能纠正所有单比特错
  - 对2^m个合法报文的任何一个而言，有n个与该报文距离为1的无效码字，所以2^m个合法报文的每一个都对应有$n+1$个各不相同的位图，n位码字的总的位图是2^n个：

    $(n+1)2^m \le 2^n$，$n=m+r$ 代入

    $(m+r+1)2^m \le 2^{m+r}$

    $\mathbf{2^r \ge m+r+1}$ ⇨ 纠正单比特误码的校验位下界 r

例： 若 $m=7$，则 $r \ge 4$，$n=7+4=11$

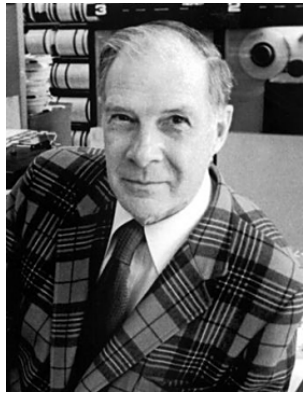
若 $r=4$，则 $m \le 11$，$n=11+4=15$

海明码

- 编码方法
  - 码位从左边开始编号，从“1”开始，标记位号
  - 位号为2的幂的位是校验位，其余是信息位
  - 每个校验位使得包括自己在内的一些位的奇偶值为偶数（或奇数）
  - 位号转换为二进制数， 若其2的某幂位为1，则该位数值计入某校验位组

为看清数据位k对哪些校验位有影响，将k写成2的幂的和，

例：$11 = (1011)_2 = 8+2+1 = 1+2+8$



R.W. Hamming
(1915-1998)

Richard Hamming Worked at Bell Labs
- Developed Hamming Codes to save time on punchcard reading errors(节省打孔卡读错误的时间)
- Mixed message bits and parity bits to detect and correct specific errors(混合消息位和奇偶校验位以检测和纠正特定错误)
- Hamming codes now used for network communications as well as hard drive RAIDs(用于网络通信以及硬盘RAID)

海明码—编码

- 编码方法举例

m = 7，r = 4，n=11，显然 $2^4 \geq 11+1$，采用偶校验

校验位：1、2、4、8；数据位：3、5、6、7、9、10、11

1 ∈ {3，5，7，9，11}

2 ∈ {3，6，7，10，11}

3 = 1+2

4 ∈ {5，6，7}

5 = 1+4

6 = 2+4

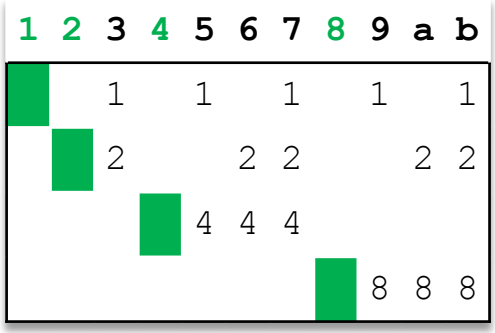
7 = 1+2+4

8 ∈ {9，10，11}

9 = 1+8

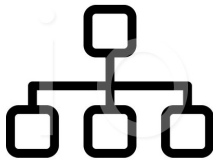
10 = 2+8

11 = 1+2+8



	8	4	2	1
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
a	1	0	1	0
b	1	0	1	1

海明码—校验



- 工作过程
  - 每个码字到来前，接收方计数器清零
  - 接收方检查每个校验位k (k = 1, 2, 4 …)的奇偶值是否正确
  - 若第 k 位奇偶值不对，计数器加 k
  - 所有校验位检查完后，若计数器值为0，则码字有效；若计数器值为m，则第m位出错。例：若校验位1、2、8出错，则第11位变反
- 使用海明码纠正突发错误
  - 可采用k个码字（n = m + r）组成 k×n 矩阵，按列发送，接收方恢复成 k×n 矩阵
  - k × r 个校验位，k × m 个数据位，可纠正最多为k个的突发性连续比特错

海明码—举例

- ASCII码的海明码
- 字符 = H，ASCII = 1001000

  m = 7，r = 4，n=11，显然 $2^4 \geq 11+1$，采用偶校验

  码字： 00110010000

- 若出错：00110110000

  出错位：=2+4=6

  计算可知，k=2位置的偶校验值应为1，k=4位置的偶校验值应为0，这与接收到的情况不符。

1	2	3	4	5	6	7	8	9	a	b		
0	0	1	1	0	1	1	0	0	0	0		
0		1		0		1		0		0	1	√
	0	1			1	1			0	0	2	×
			1	0	1	1					4	×
							0	0	0	0	8	√

海明码

1	2	3	4	5	6	7	8	9	a	b	校验位 k	偶校验值
?	?	1	?	0	0	1	?	0	0	0		
x		1		0		1		0		0	1	0
	x	1			0	1			0	0	2	0
			x	0	0	1					4	1
							x	0	0	0	8	0

海明码—纠正突发性连续比特错

- 纠正不超过12个突发性连续比特错

  传输信息“Hamming code”

  将7位ASCII码加入4位校验码至11位，12×11位块

  如：“a”= $(1100001)_{ASCII}$

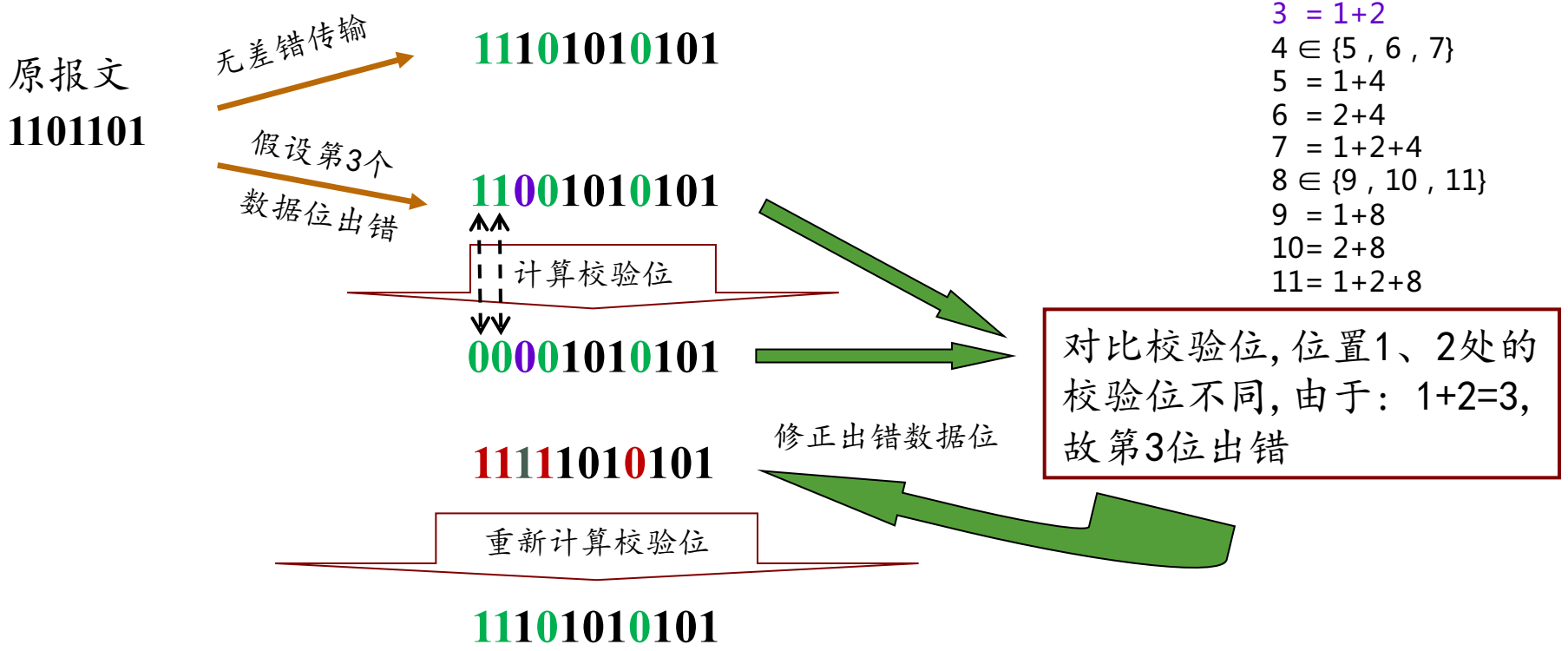
1	2	3	4	5	6	7	8	9	A	b	
?	?	1	?	1	0	0	?	0	0	1	
1		1		1		0		0		1	1
	0	1			0	0			0	1	2
			1	1	0	0					4
							1	0	0	1	8

字符	ASCII	校验位
H	1001000	00110010000
a	1100001	10111001001
m	1101101	11101010101
m	1101101	11101010101
i	1101001	01101011001
n	1101110	01101010110
g	1100111	11111001111
	0100000	10011000000
c	1100011	11111000011
o	1101111	00101011111
d	1100100	11111001100
e	1100101	00111000101

位传输的顺序

海明码—纠正单比特错举例

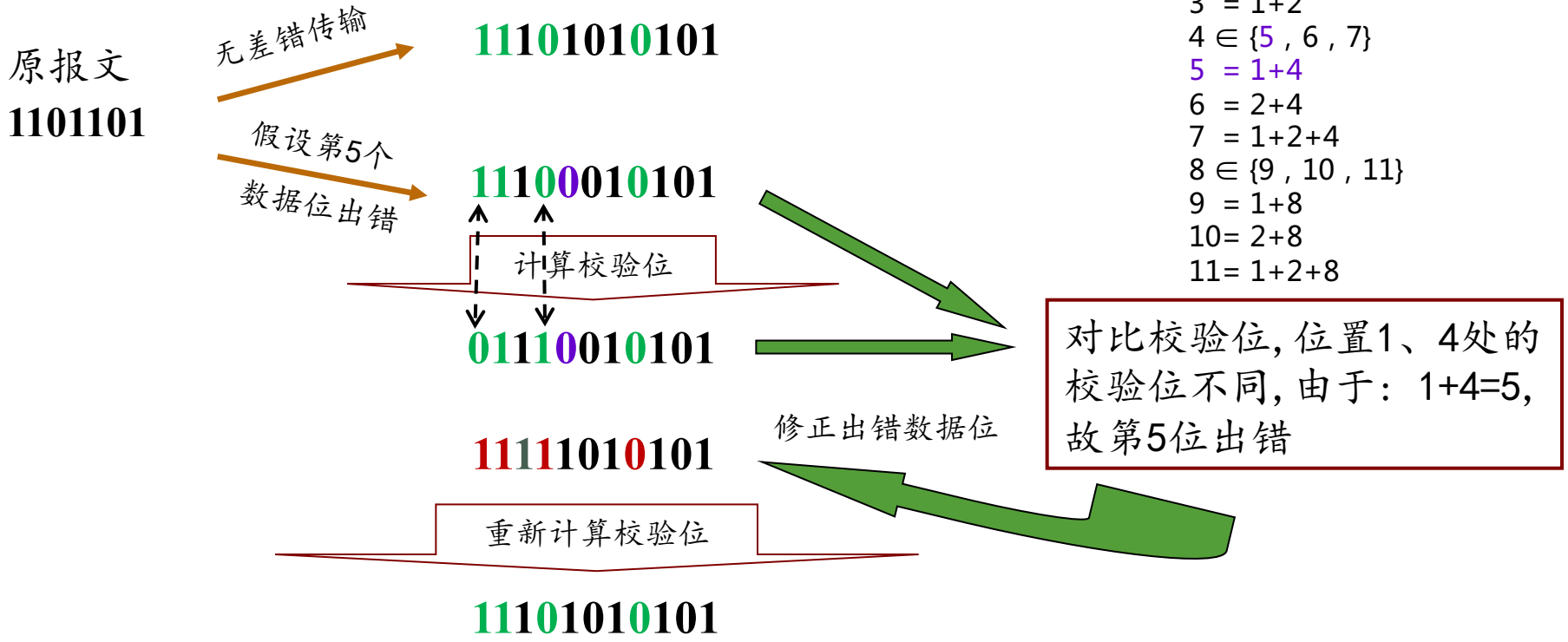
- 纠正码字中的单个数据位错误： “m” = $(1101101)_{ASCII}$



1 ∈ {3 , 5 , 7 , 9 , 11}
2 ∈ {3 , 6 , 7 , 10 , 11}
3 = 1+2
4 ∈ {5 , 6 , 7}
5 = 1+4
6 = 2+4
7 = 1+2+4
8 ∈ {9 , 10 , 11}
9 = 1+8
10= 2+8
11= 1+2+8

海明码—纠正单比特错举例 (2)

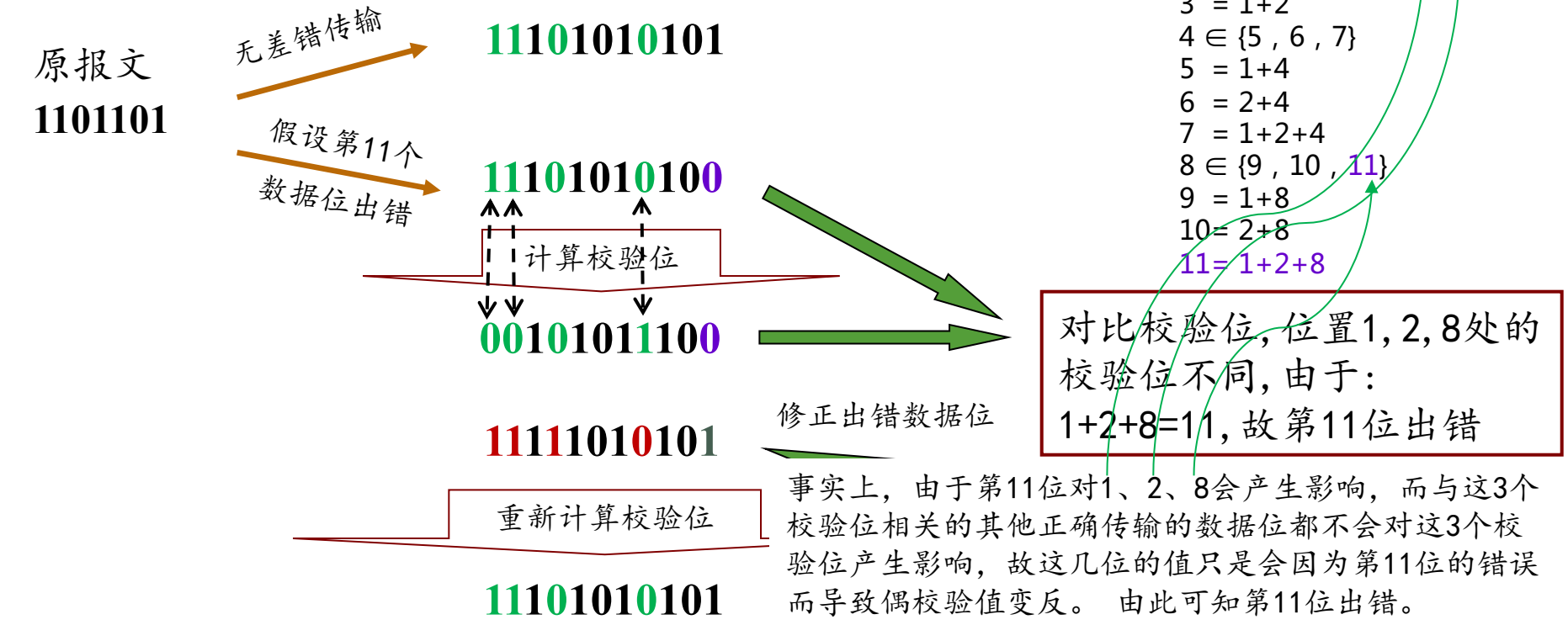
- 假设，码字第5个数据位出错



1 ∈ {3 , 5 , 7 , 9 , 11}
2 ∈ {3 , 6 , 7 , 10 , 11}
3 = 1+2
4 ∈ {5 , 6 , 7}
5 = 1+4
6 = 2+4
7 = 1+2+4
8 ∈ {9 , 10 , 11}
9 = 1+8
10= 2+8
11= 1+2+8

海明码—纠正单比特错举例 (3)

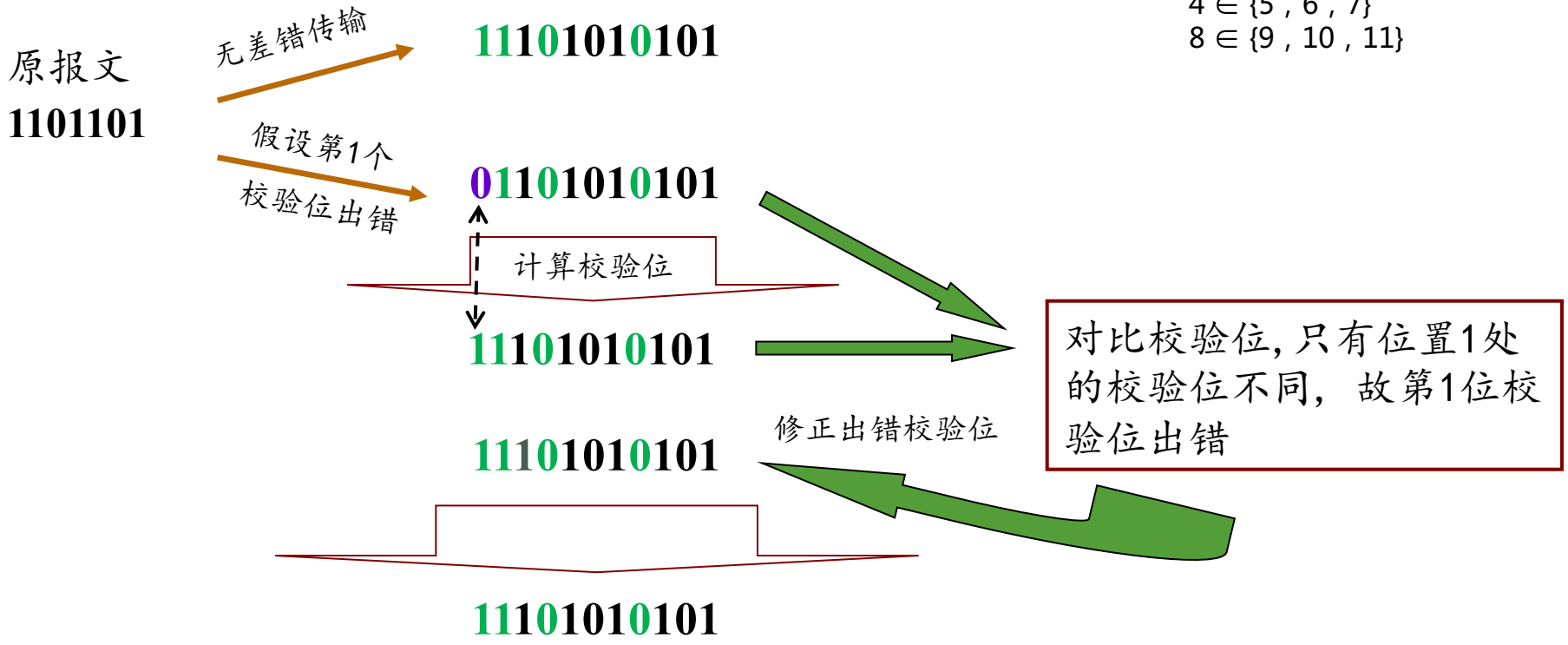
- 假设，码字第11个数据位出错



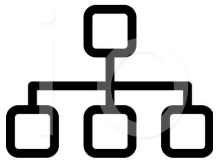
海明码—纠正单比特错举例 (4)

- 假设，校验位出错

1 ∈ {3 , 5 , 7 , 9 , 11}
2 ∈ {3 , 6 , 7 , 10 , 11}
4 ∈ {5 , 6 , 7}
8 ∈ {9 , 10 , 11}



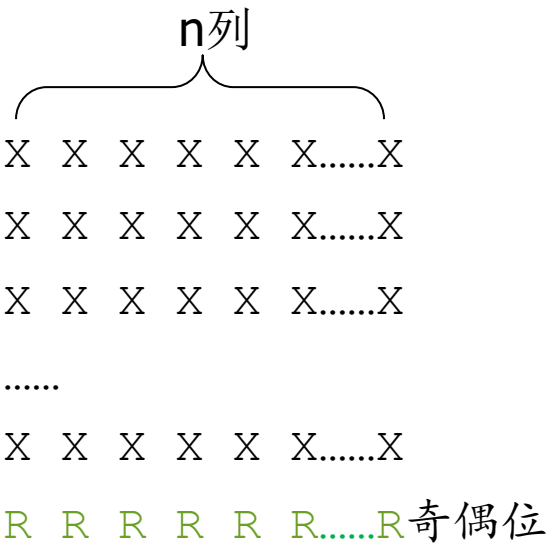
纠错码与检错码



- 在实际通信中使用纠错码好还是检错码好呢？
- 例如
  - 假设一个信道误码率是10^{-6}，且出错是孤立产生的（即只有单比特错），数据块长度为1000比特
  - 如果采用纠错编码，需要10个校验位（$2^{10}>1011$），传送1M数据需要**10000**个校验位
  - 如果采用检错编码，每个数据块只需一个奇偶校验位，传送1M数据只需1000个校验位和一个重传的数据1001位，共需要**2001**比特的额外开销
- 在多数通信中采用检错编码，但在单工信道中需要纠错编码

改进的奇偶校验

- 将数据位组成一个n位宽，k位高的长方形距阵来发送，然后对每一列单独计算奇偶位，并附在最后一行作为冗余位

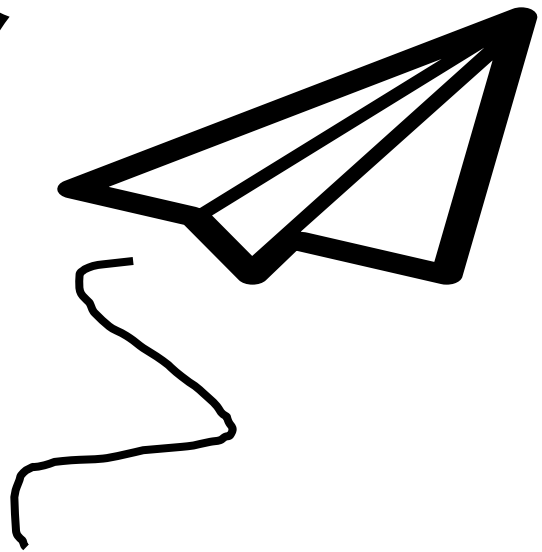


检错率：

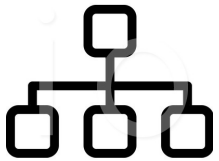
1. 该方法可以检测长度为n的突发性错误，但不能检测长度为n+1的突发性错误
2. 假设n列中任意一列检测出错的概率为1/2，那么，整个数据块的错判率为（1/2）^{n}

该方法用在ICMP分组首部检验中

本节课程结束

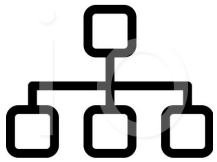


3.2.2 检错码



- 使用纠错码传数据，效率低，适用于不可能重传的场合；大多数情况采用检错码加重传
- 三种主要的检错码
  - 奇偶校验
  - 校验和 ✓
  - 循环冗余码（CRC码，多项式编码）✓

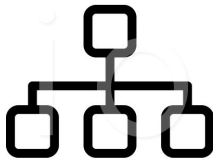
校验和（checksum）



- Idea: sum up data in N-bit words
- 校验和是一个补码运算(放在消息末尾)
  - And it's the negative sum（补码运算中：负数是其正数的按位补）
- "*The checksum field is the 16 bit one's complement of the one's complement sum of all 16 bit words …*" – RFC 791

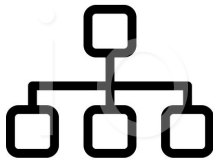
  - Widely used in, e.g., TCP/IP/UDP
- Stronger protection than parity

原码、反码、补码运算



- 原码
  - 最高位为符号位，0代表正数，1代表负数，非符号位为该数字绝对值的二进制表示。例如：+7的原码为0 0000111，-7的原码就是1 0000111
- 反码
  - 正数的反码与原码一致；
  - 负数的反码是对原码按位取反，只是最高位(符号位)不变。
    - 例如：$[+7]_{反}$= 0 0000111 B；$[-7]_{反}$= 1 1111000 B。
- 补码：
  - 正数的补码与原码一致；
  - 负数的补码是该数的反码加1。
    - 例如：$[+7]_{补}$= 0 0000111 B；$[-7]_{补}$= 1 1111001 B。

Internet Checksum (2)



发送端:

1. Arrange data in 16-bit words

```
0001
f203
f4f5
f6f7
```

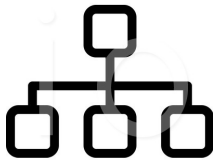
2. Put zero in checksum position, add

3. Add any carryover(进位) back to get 16 bits

4. Negate (complement) to get sum

将发送的数据以两字节为单位进行补码相加，checksum处设置为0，得到32字节的数，再将低16位和高16位相加，结果求反，即为较验和。

Internet Checksum (3)



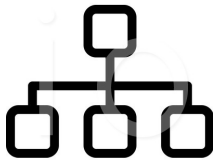
发送端:

1. Arrange data in 16-bit words
2. Put zero in checksum position, add
3. Add any carryover(进位) back to get 16 bits
4. Negate (complement) to get sum

```
  0001
  f203
  f4f5
  f6f7
+(0000)
------
 2ddf0
   ↓
  ddf0
+    2
------
  ddf2
   ↓ 取反码
  220d
```

将发送的数据以两字节为单位进行补码相加，checksum处设置为0，得到32字节的数，再将低16位和高16位相加，结果求反，即为较验和。

Internet Checksum (4)



接收端:

1. Arrange data in 16-bit words
2. Checksum will be non-zero, add

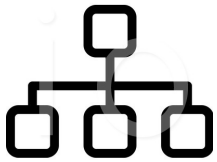
```
  0001
  f203
  f4f5
  f6f7
+ 220d
------
```

3. Add any carryover back to get 16 bits

4. Negate the result and check it is 0

和发送端相同的算法，checksum处使用发送端的结果进行补码相加，结果求反为0x0000，表明数据传输正确，否则数据传输不正确，舍弃。

Internet Checksum (5)



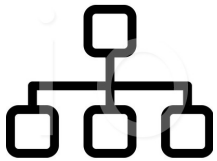
接收端:

1. Arrange data in 16-bit words
2. Checksum will be non-zero, add
3. Add any carryover back to get 16 bits
4. Negate the result and check it is 0

```
  0001
  f203
  f4f5
  f6f7
+ 220d
------
 2fffd
   ↓
  fffd
+    2
------
  ffff
   ↓
  0000
```

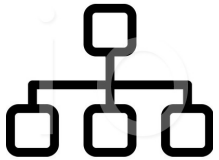
和发送端相同的算法，checksum处使用发送端的结果进行补码相加，结果求反为0x0000，表明数据传输正确，否则数据传输不正确，舍弃。

循环冗余码（CRC）



- 计算准备
  - k位帧作为k次多项式系数序列
  - 多项式模2运算，即加/减不进/借位，等同于异或运算
  - 生成多项式G(x)高位低位为1
    - 若G(x)为 r 阶，即至少$G(x) = x^r + 1$
- 传送m位的帧M(x)
  - G(x)为 r 阶，即 r + 1 位 $(r + 1 \leq m + r)$
  - M(x)帧尾附加 r 位的校验和R(x)，即 $x^r M(x)+R(x) = T(x)$
  - 使得T(x)能被G(x)整除，T(x)即为要传送的带校验和的帧

循环冗余码 (2)

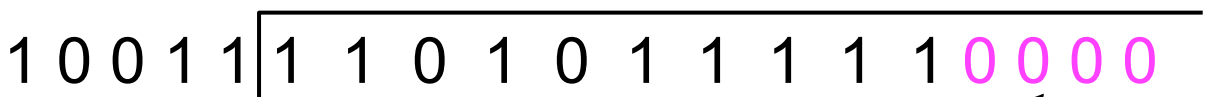


- CRC码基本思想
  - 校验和（checksum）加在帧尾，使带校验和的帧的多项式能被G(x)除尽；收方接收时，用G(x)去除它，若有余数，则传输出错
- 求校验和的方法
  - M(x)帧尾附加r个零得 $x^r M(x)$，再按模2除法除以G(x)
  - 所得余数R(x)即是校验和

CRC示例

Data bits:
1101011111

Check bits:
$G(x)=x^4+x^1+1$
G = 10011
k = 4



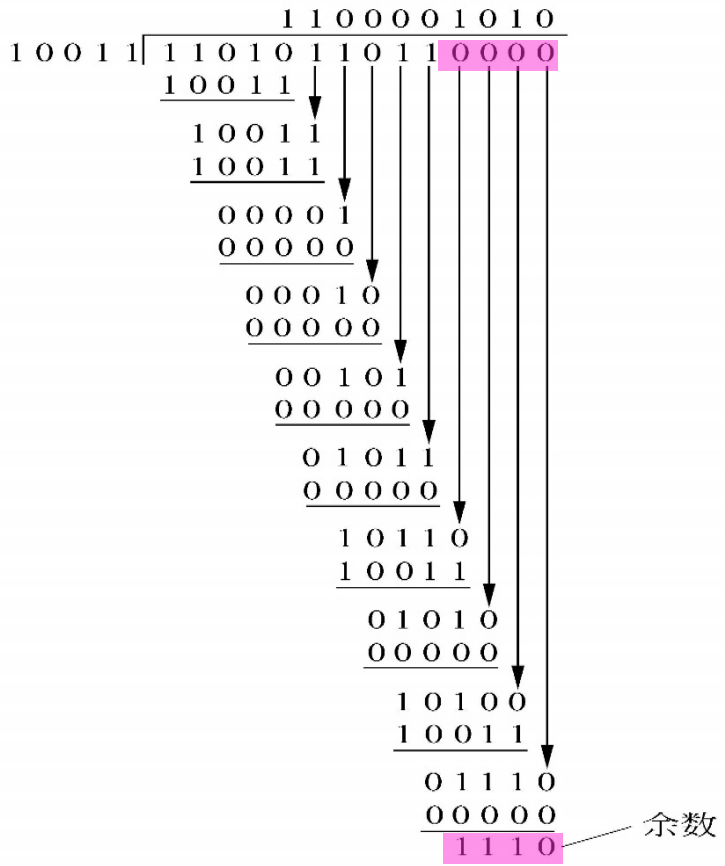
循环冗余码 (3)

- G(x) = 10011 , M(x) = 1101011011

```
          1100001010
      ____________
10011 | 11010110110000
        10011
        -----
         10011
         10011
         -----
          10110
          10011
          -----
           10100
           10011
           -----
            1110
```

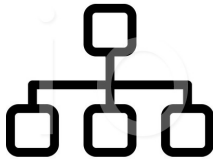
- R(x) = 1110 , T(x) = 11010110111110

帧：1 1 0 1 0 1 1 0 1 1
除数：1 0 0 1 1
附加 4 个零后形成的串：1 1 0 1 0 1 1 0 1 1 0 0 0 0



传输的帧：1 1 0 1 0 1 1 0 1 1 1 1 1 0

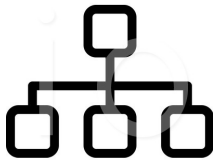
循环冗余码检错能力



- 假设：
  - 发送：T(x)；接收：T(x)+E(x)， E(x)≠0；
  - 余数((T(x)+E(x))/G(x))=0+余数(E(x)/G(x))
- 结论：
  - 若余数(E(x)/G(x))=0，则差错不能发现；否则，可以发现。

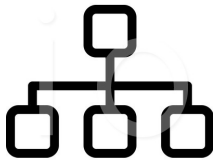
即，G(x)整倍数多项式差错以外的错误均能捕捉到

循环冗余码检错能力 (2)



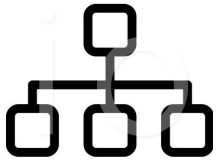
- 单比特错误：$E(x)=x^i$
  - 因$G(x)$不少于2项，不可能除尽 x^i，所以所有单比特错都可检测到
- 两个孤立的单比特错误： $E(x)=x^i+x^j=x^j(x^{i-j}+1)$
  - 若$G(x)$满足对于任何$k(\leq i-j)$，不能除尽 x^k+1，则可检测双位错
    - 例：$x^{15}+x^{14}+1$不能整除 x^k+1 ($k<32768$)，$G(x)=1100000000000001$
- 奇数个错误发生： $E(x)$有奇数项，如 $E(x)=x^5+x^2+1$
  - 由于模2的系统中，奇数项多项式不包含x+1因子，故x+1 不可能整除奇数项的E(x)，只要G(x)包含x+1因子即可检测奇数位错的情况
  - 证明
    - 奇数项的$E(1)<>0$，若$E(x)=(x+1)\times Q(x)$，则$E(1)=(1+1)\times Q(1)=0\times Q(1)=0$

循环冗余码检错能力 (3)



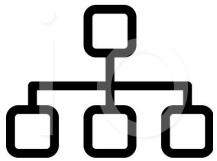
- 突发错误
  - 具有r个校验位的多项式能检查出所有长度 ≤ r 的突发性差错
    - 长度为k的突发错误表达为：$E(x) = x^i (x^{k-1} + \ldots + 1)$
    - 若k ≤ r，因生成 r 位检验和的G(x)为 r+1 位，，余数不为0，可检测
  - 如果突发差错长度为 r + 1
    - 当且仅当突发差错和G(x)一样时，余数 E(x) / G(x) = 0，将坏帧错误接收为有效帧的概率为 $1/2^{r-1}$
  - 长度大于 r + 1的突发差错或几个较短的突发差错
    - 发生后，坏帧被当作有效帧接收的概率为 $1/2^r$

CRC标准



- 四个多项式已成为国际标准
  - CRC-12 $= x^{12} + x^{11} + x^3 + x^2 + x + 1$
  - CRC-16 $= x^{16} + x^{15} + x^2 + 1$
  - CRC-CCITT $= x^{16} + x^{12} + x^5 + 1$
  - CRC-32 $= x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$
- 硬件实现CRC校验
  - 网卡NIC（Network Interface Card）

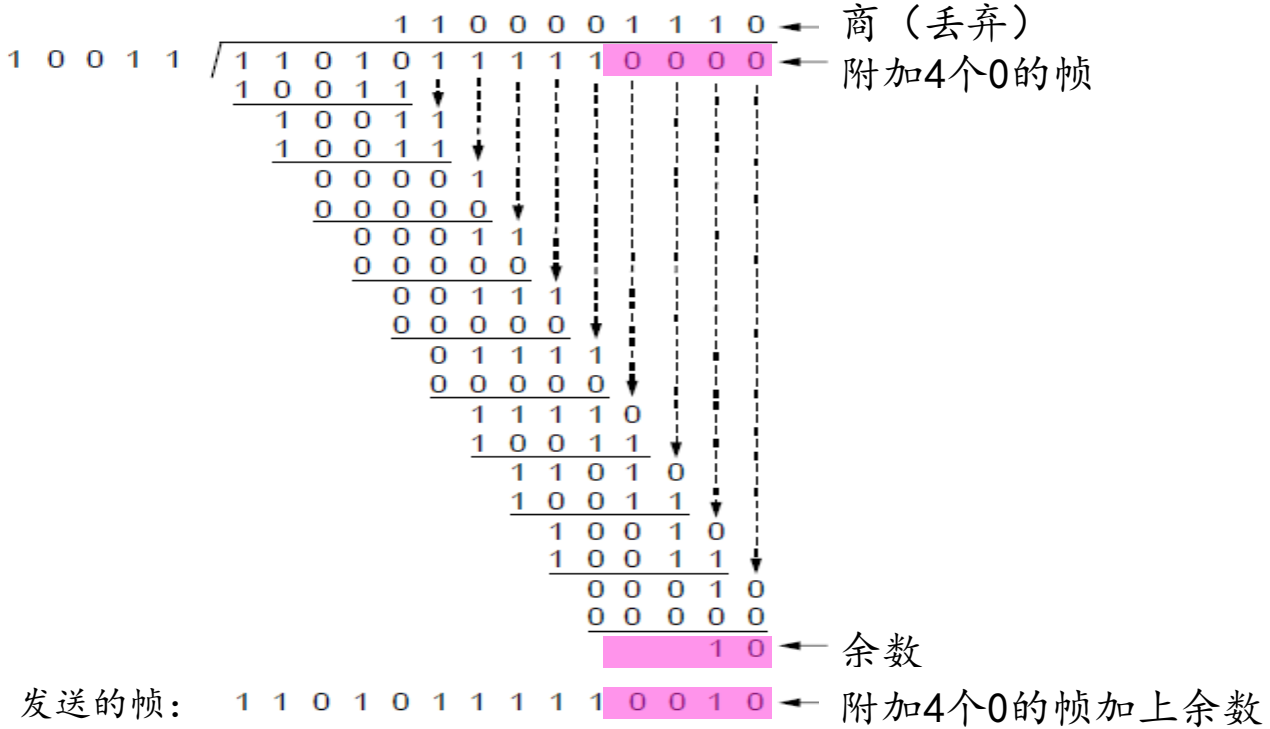
CRC小结



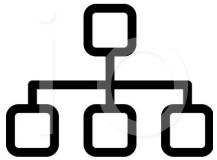
- 在数据链路层传送的帧中，就广泛使用了循环冗余检验 CRC 的检错技术
- 以下我们不以多项式，而以二进制数据串为对象做进一步解释
  - 假设待传送的数据 M = 1010001101（共m bit）。我们在M的后面再添加供差错检测用的 r bit 冗余码一起发送
    - 用二进制的模 2 运算进行 2^r 乘 M 的运算，这相当于在 M 后面添加 r 个 0
  - 得到的 (m + r) bit 的数，除以事先选定好的长度为 (r + 1) bit 的数 P，得出商是 Q ，而余数是 R，余数 R 比除数 P 至少要少1 个比特
    - 设 r = 5, P = 110101，模 2 运算的结果是：商 Q = 1101010110，余数 R = 01110
  - 将余数 R 作为冗余码添加在数据 M 的后面发送出去： $2^r M + R$
    - 即发送的数据是1010001101101110

CRC练习

帧数据为：1101011111，生成多项式为：10011，请写出算式，计算发送的帧数据是什么？

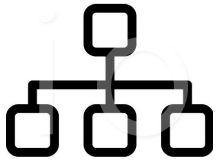


数据链路层帧检验



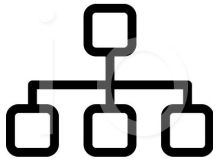
- 在数据后面添加上的冗余码称为帧检验序列 FCS (Frame Check Sequence)
- 循环冗余检验 CRC 和帧检验序列 FCS并不等同
  - CRC 是一种常用的检错方法，而 FCS 是添加在数据后面的冗余码
  - FCS 可以用 CRC 这种方法得出，但 CRC 并非用来获得 FCS 的惟一方法
- 只要得出的余数 R 不为 0，就表示检测到了差错
  - 但这种检测方法并不能确定究竟是哪一个或哪几个比特出现了差错
- 一旦检测出差错，就丢弃这个出现差错的帧
  - 只要经过严格的挑选，并使用位数足够多的除数 P，那么出现检测不到的差错的概率就很小很小

数据链路层帧检验 (2)



- 仅用循环冗余检验 CRC 差错检测技术只能做到无差错接受 (accept)
  - “无差错接受”是指：“凡是接受的帧（即不包括丢弃的帧），我们都能以非常接近于 1 的概率认为这些帧在传输过程中没有产生差错”
  - 也就是说：“凡是接受的帧都没有传输差错”（有差错的帧就丢弃而不接受）
- 要做到“可靠传输”（即发送什么就收到什么）就必须再加上确认和重传机制

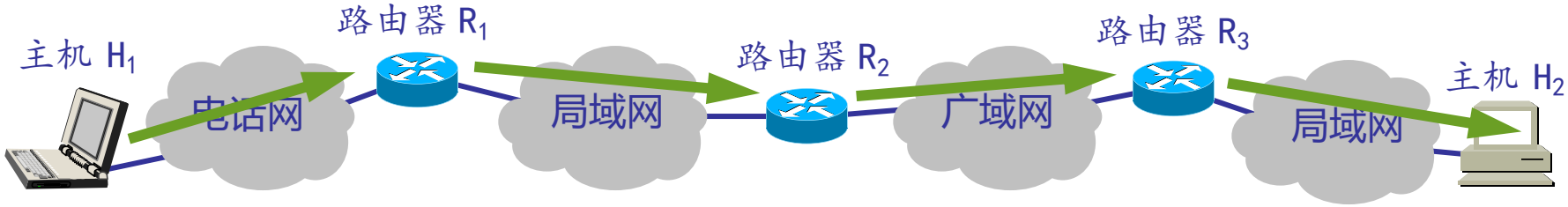
Error Detection in Practice

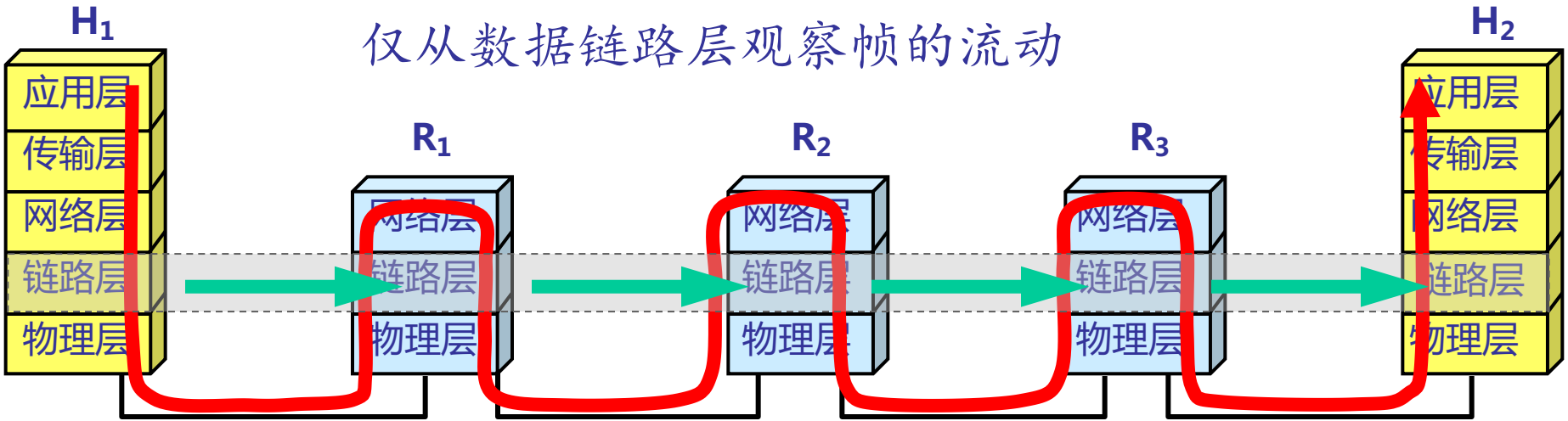


- CRCs are widely used on links
  - Ethernet, 802.11, ADSL, Cable …
- Checksum used in Internet
  - IP, TCP, UDP … but it is weak
- Parity
  - Is little used

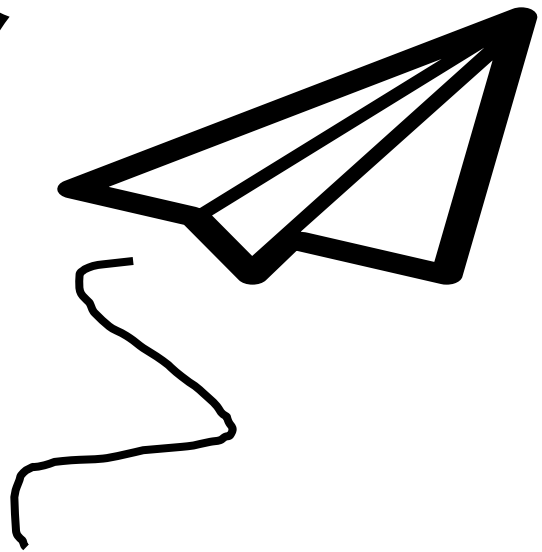
数据链路层的简单模型

- 主机 H1 向 H2 发送数据

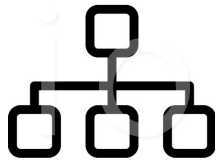




本节课程结束



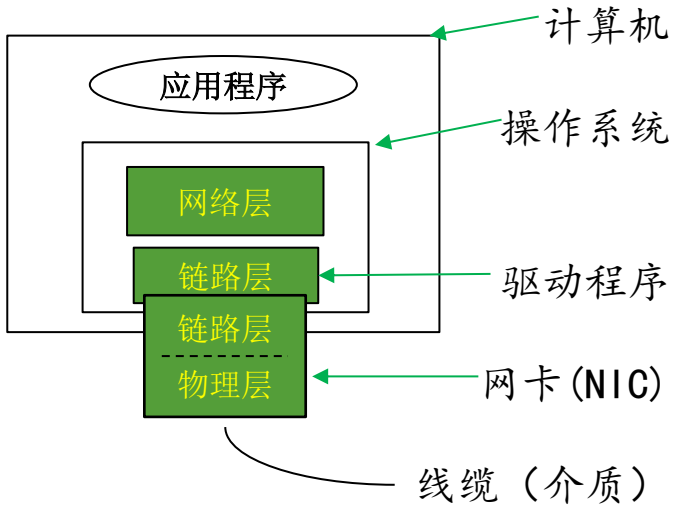
3.3 基本的数据链路层协议



- 前提与准备
- 无约束单工协议
- 单工停等协议
- 有噪声信道的单工协议

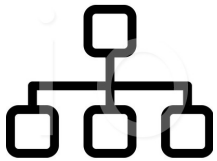
3.3.1 前提与准备

- 前提假设
  - 三层完全独立
    - 物理层和数据链路层：进程在网络I/O芯片上运行
    - 网络层：在主CPU上运行
  - 机器A需要用可靠的、面向连接的服务方式
    - 向机器B发送很长的数据流
  - 机器不会崩溃



物理层、链路层和网络层的实现

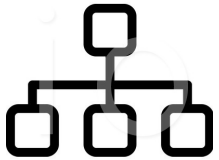
协议准备工作



- 帧的发送与接收
  - (1、2)层：to_physical_layer， from_physical_layer
  - (2、3)层：to_network_layer， from_ network_layer
- 开始：wait_for_event(&event)
  - event = cksum_err | frame_arrival
- 数据类型(C声明)
  - 帧编号 seq_nr : 0~MAX_SEQ
  - 信息交换单位packet长度：变长或MAX_PKT

- struct frame
  - kind : data | arc | nak
  - seq, ack : 序号与确认
  - info 数据

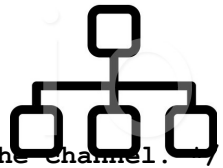
协议准备工作 (2)



- 协议中的定义头文件：protocol.h

```
#define MAX PKT 1024                    /* determines packet size in bytes */

typedef enum {false, true} boolean;     /* boolean type */
typedef unsigned int seq nr;            /* sequence or ack numbers */
typedef struct {unsigned char data
  [MAX PKT];} packet;                   /* packet definition */
typedef enum {data, ack, nak}
  frame kind;                           /* frame kind definition */

typedef struct {                        /* frames are transported in this layer */
  frame kind kind;                      /* what kind of frame is it? */
  seq nr seq;                           /* sequence number */
  seq nr ack;                           /* acknowledgement number */
  packet info;                          /* the network layer packet */
} frame;
```

协议准备工作 (3)

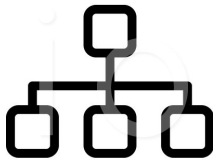


头文件中定义的过程

- 等待事件发生
- 发送或接收网络层分组
- 链路层和物理层之间传递帧
- 打开或关闭计时器
- 用于确认的辅助计时器
- 启用/禁用网络层
- 帧序号加1

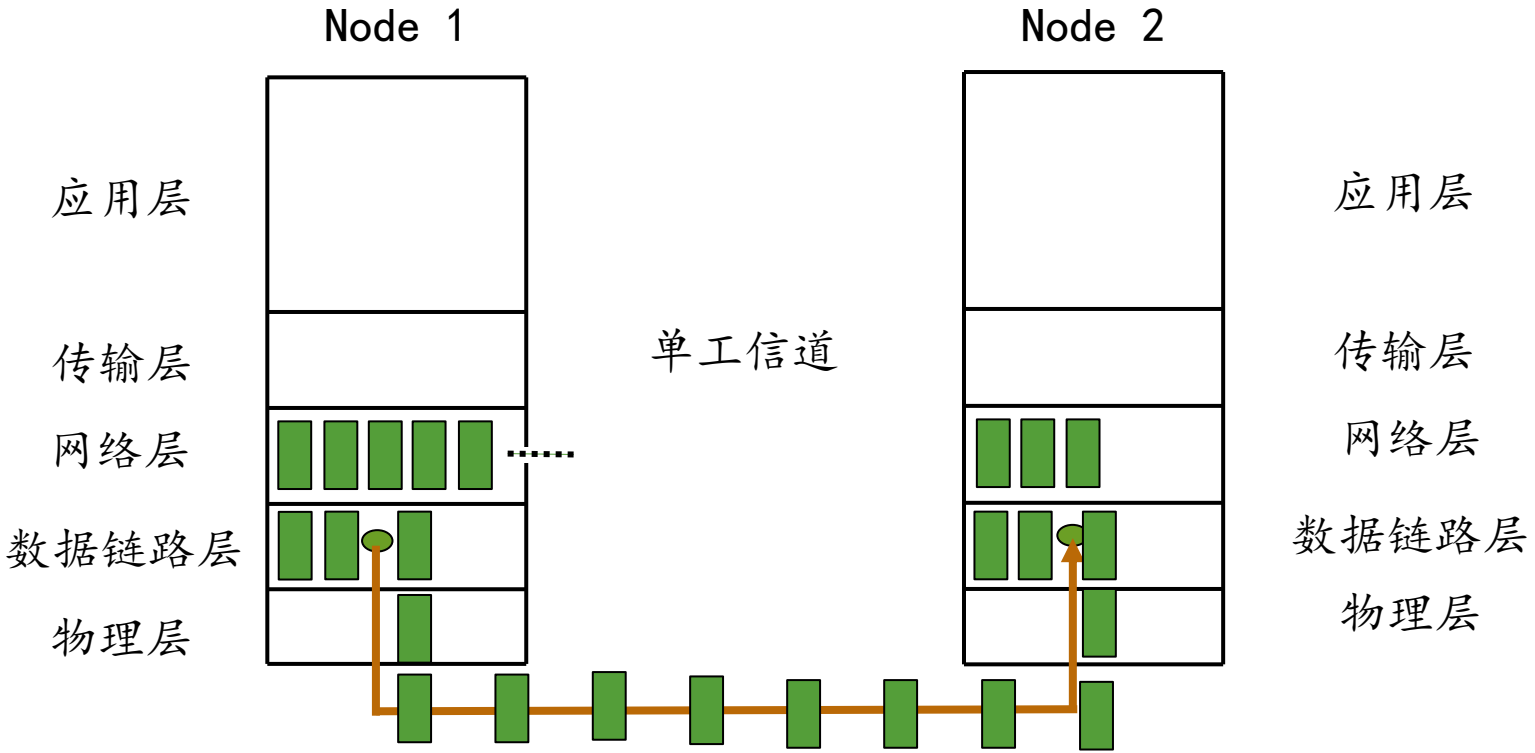
```
void wait for event(event type *event);
/* Fetch a packet from the network layer for transmission on the channel. */
void from network layer(packet *p);
/* Deliver information from an inbound frame to the network layer. */
void to network layer(packet *p);
/* Go get an inbound frame from the physical layer and copy it to r. */
void from physical layer(frame *r);
/* Pass the frame to the physical layer for transmission. */
void to physical layer(frame *s);
/* Start the clock running and enable the timeout event. */
void start timer(seq nr k);
/* Stop the clock and disable the timeout event. */
void stop timer(seq nr k);
/* Start an auxiliary timer and enable the ack timeout event. */
void start ack timer(void);
/* Stop the auxiliary timer and disable the ack timeout event. */
void stop ack timer(void);
/* Allow the network layer to cause a network layer ready event. */
void enable network layer(void);
/* Forbid the network layer from causing a network layer ready event. */
void disable network layer(void);
/* Macro inc is expanded in-line: increment k circularly. */
#define inc(k) if (k < MAX SEQ) k = k + 1; else k = 0
```

3.3.2 无约束单工协议

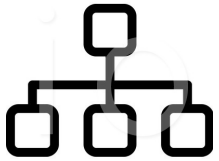


- 无约束单工协议[1] (An Unrestricted Simplex Protocol)
  - 工作在理想情况，几个前提：
    - 单工传输
    - 发送方无休止工作（要发送的信息无限多）
    - 接收方无休止工作（缓冲区无限大或无处理时间）
    - 通信线路（信道）不损坏或丢失信息帧
  - 工作过程
    - 发送程序：取数据，构成帧，发送帧
    - 接收程序：等待，接收帧，送数据给高层

无约束单工协议[1]示意图



乌托邦式单工协议[1]源代码

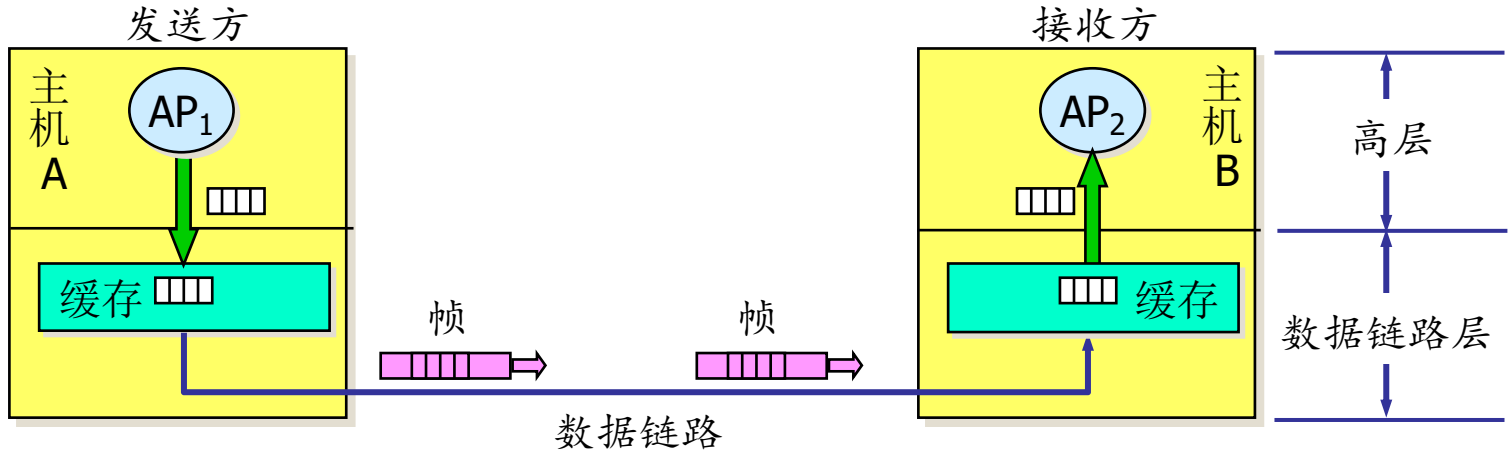


```
/* Protocol 1 (Utopia) provides for data transmission in one direction only, from sender to receiver. The
   communication channel is assumed to be error free and the receiver is assumed to be able to process all the
   input infinitely quickly. Consequently, the sender just sits in a loop pumping data out onto the line as fast
   as it can. */
typedef enum {frame arrival} event type;
#include "protocol.h"
void sender1(void)
{
   frame s;                          /* buffer for an outbound frame */
   packet buffer;                    /* buffer for an outbound packet */
   while (true) {
      from network layer(&buffer);   /* go get something to send */
      s.info = buffer;               /* copy it into s for transmission */
      to physical layer(&s);         /* send it on its way */
}                                    /* Tomorrow, and tomorrow, and tomorrow,
                                     Creeps in this petty pace from day to day
                                     To the last syllable of recorded time.
                                     - Macbeth, V, v */
}
void receiver1(void)
{
   frame r;
   event type event;                 /* filled in by wait, but not used here */
   while (true) {
      wait for event(&event);        /* only possibility is frame arrival */
      from physical layer(&r);       /* go get the inbound frame */
      to network layer(&r.info);     /* pass the data to the network layer */
   }
}
```

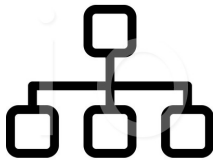
完全理想化的数据传输

- 基于两个假定
  - 假定 1： 链路是理想的传输信道，所传送的任何数据既不会出差错也不会丢失
  - 假定 2： 不管发方以多快速率发送数据，收方总是来得及收下，并及时上交主机

    即：接收端向主机交付数据的速率永远不会低于发送端发送数据的速率

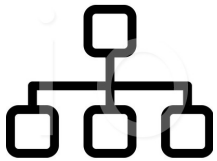


具有最简单流量控制的数据链路层协议



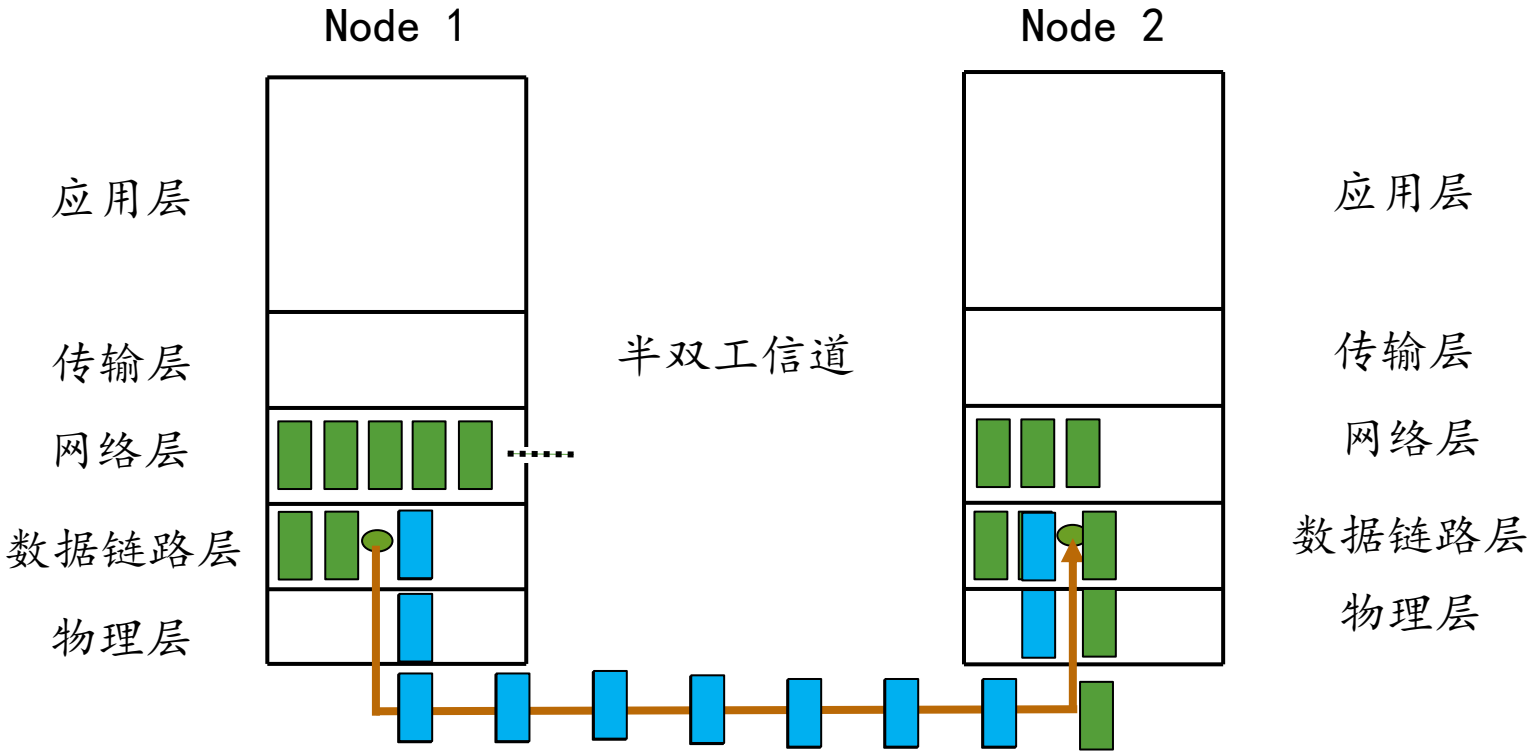
- 现在去掉上述的第二个假定
  - 仍然保留第一个假定，即主机 A 向主机 B传输数据的信道仍然是无差错的理想信道
  - 然而现在不能保证，接收端向主机交付数据的速率永远不低于发送端发送数据的速率
    - 处理速率不一致、缓存不够
    - 需要半双工的物理信道
  - 由收方控制发方的数据流，乃是计算机网络中流量控制的一个基本方法

3.3.3 单工停等协议

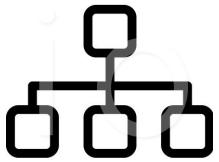


- 单工停等协议[2] (A Simplex Stop-and-Wait Protocol)
  - 增加约束条件
    - 接收方不能无休止接收
    - 半双工传输
  - 解决办法
    - 接收方每收到一个帧后，给发送方回送一个响应
  - 工作过程
    - 发送程序：取数据，成帧，发送帧，等待响应帧
    - 接收程序：等待，接收帧，送数据给高层，回送响应帧

单工停等协议[2]示意图

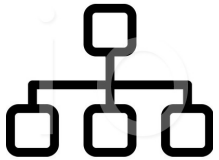


单工协停等协议[2]源代码



```
/* Protocol 2 (Stop-and-wait) also provides for a one-directional flow of data from sender to receiver. The
   communication channel is once again assumed to be error free, as in protocol 1. However, this time the
   receiver has only a finite buffer capacity and a finite processing speed, so the protocol must
   explicitly prevent the sender from flooding the receiver with data faster than it can be handled. */

typedef enum {frame arrival} event type;
#include "protocol.h"
void sender2(void)
{
    frame s;                          /* buffer for an outbound frame */
    packet buffer;                    /* buffer for an outbound packet */
    event type event;                 /* frame arrival is the only possibility */
    while (true) {
       from network layer(&buffer);   /* go get something to send */
       s.info = buffer;               /* copy it into s for transmission */
       to physical layer(&s);         /* bye-bye little frame */
       wait for event(&event);        /* do not proceed until given the go ahead */
    }
}
void receiver2(void)
{
    frame r, s;                       /* buffers for frames */
    event type event;                 /* frame arrival is the only possibility */
    while (true) {
       wait for event(&event);        /* only possibility is frame arrival */
       from physical layer(&r);       /* go get the inbound frame */
       to network layer(&r.info);     /* pass the data to the network layer */
       to physical layer(&s);         /* send a dummy frame to awaken sender */
    }
}
```

最简单流量控制的数据链路层协议算法



- 在发送结点：

① 从主机取一个数据帧

② 将数据帧送到数据链路层的发送缓存

③ 将发送缓存中的数据帧发送出去

④ 等待

⑤ 若收到由接收结点发过来的信息(此信息的格式与内容，可由双方事先商定好)，转到①

- 在接收结点：

① 等待

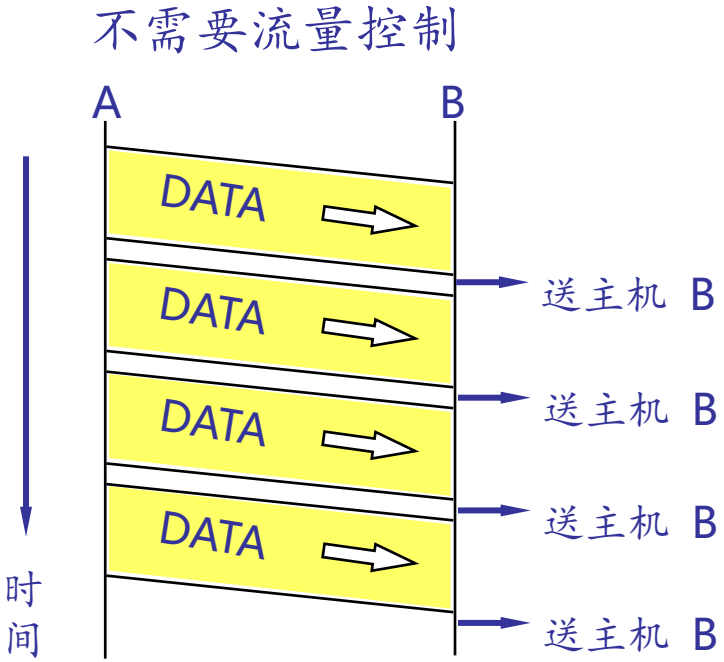
② 若收到由发送结点发过来的数据帧，则将其放入数据链路层的接收缓存

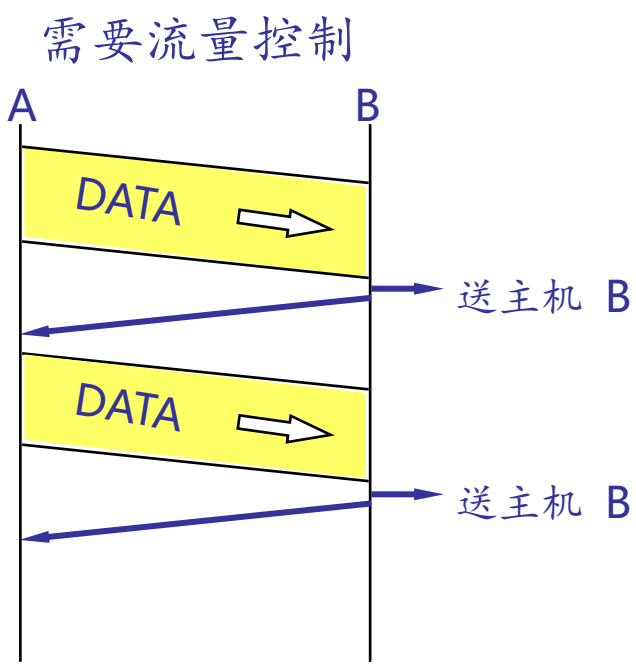
③ 将接收缓存中的数据帧上交主机

④ 向发送结点发一信息，表示数据帧已经上交给主机

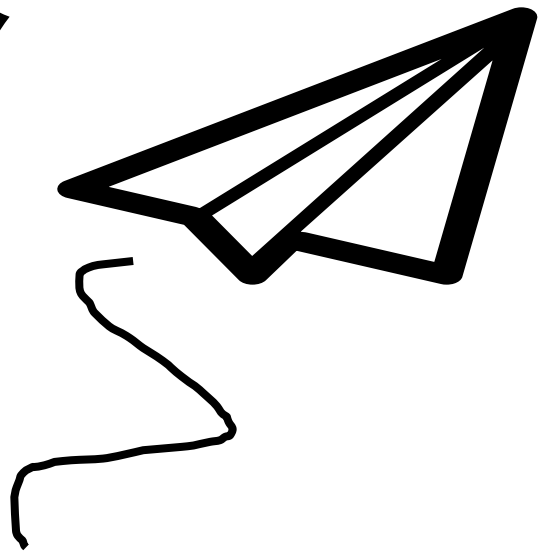
⑤ 转到①

两种情况的对比（传输均无差错）

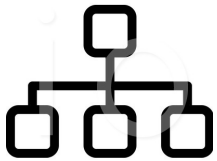




本节课程结束

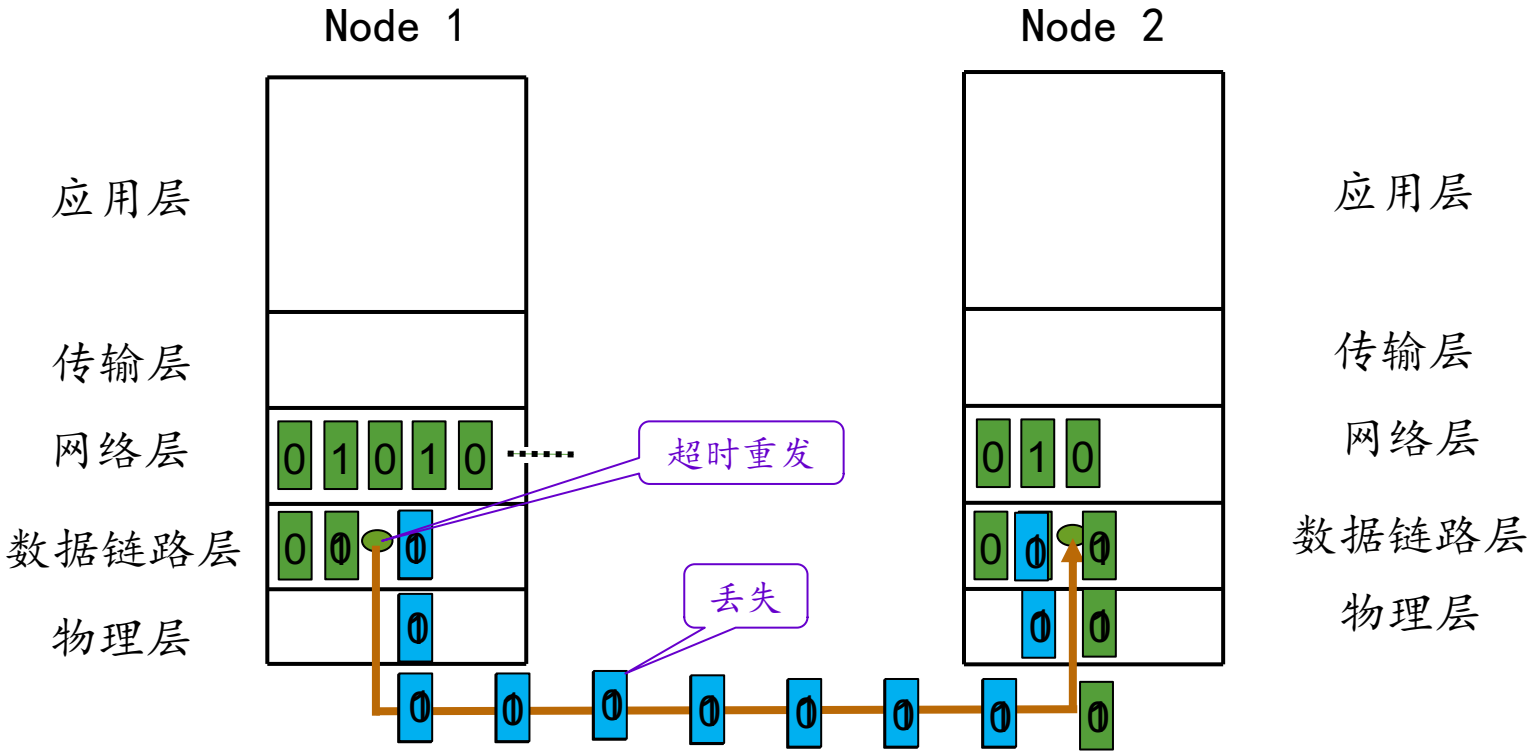


3.3.4 有噪声信道的单工协议



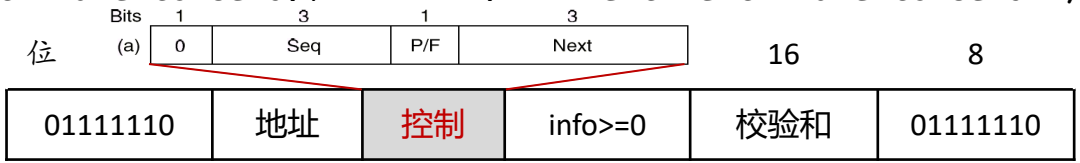
- 有噪声信道的单工协议[3] (A Simplex Protocol for a Noisy Channel)
  - 增加约束条件：信道（线路）有差错，信息帧可能损坏或丢失
  - 解决办法：出错重传
  - 带来的问题
    - 什么时候重传 —— 定时
    - 响应帧损坏怎么办(重复帧)——发送帧头中放入序号
    - 为了使帧头精简，序号取多少位 —— 1位
  - 自动重传请求ARQ（Automatic Repeat reQuest）协议：发方在发下一个帧之前等待一个肯定确认，也称带重传的肯定确认PAR（Positive Acknowledgement with Retransmission）协议
  - 工作过程

有噪声信道的单工协议[3]示意图



带重传的肯定确认协议(PAR), 简称协议3，源代码

```
/* Protocol 3 (PAR) allows unidirectional data flow over an unreliable channel. */
  #define MAX SEQ 1                          /* must be 1 for protocol 3 */
  typedef enum {frame arrival, cksum err, timeout} event type;
  #include "protocol.h"
void sender3(void)
{
  seq nr next frame to send;                 /* seq number of next outgoing frame */
  frame s;                                   /* scratch variable */
  packet buffer;                             /* buffer for an outbound packet */
  event type event;
  next frame to send = 0; /* initialize outbound sequence numbers */
  from network layer(&buffer);               /* fetch first packet */
  while (true) {
    s.info = buffer;                         /* construct a frame for transmission */
    s.seq = next frame to send;              /* insert sequence number in frame */
    to physical layer(&s);                   /* send it on its way */
    start timer(s.seq);                      /* if answer takes too long, time out */
    wait for event(&event);                  /* frame arrival, cksum err, timeout */
    if (event == frame arrival) {
      from physical layer(&s);               /* get the acknowledgement */
      if (s.ack == next frame to send) {
        stop timer(s.ack);                   /* turn the timer off */
        from network layer(&buffer);         /* get the next one to send */
        inc(next frame to send);             /* invert next frame to send */
      }
    }
  }
}
```

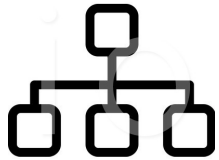


HDLC帧结构

带重传的肯定确认协议(PAR), 简称协议3，源代码(续)

```
void receiver3(void)
{
  seq nr frame expected;
  frame r, s;
  event type event;
  frame expected = 0;
  while (true) {
    wait for event(&event);                 /* possibilities: frame arrival, cksum err */
    if (event == frame arrival) {           /* a valid frame has arrived */
      from physical layer(&r);              /* go get the newly arrived frame */
      if (r.seq == frame expected) {        /* this is what we have been waiting for */
         to network layer(&r.info);         /* pass the data to the network layer */
         inc(frame expected);               /* next time expect the other sequence nr */
      }
      s.ack = 1 - frame expected;           /* tell which frame is being acked */
      to physical layer(&s);                /* send acknowledgement */
    }
  }
}
```

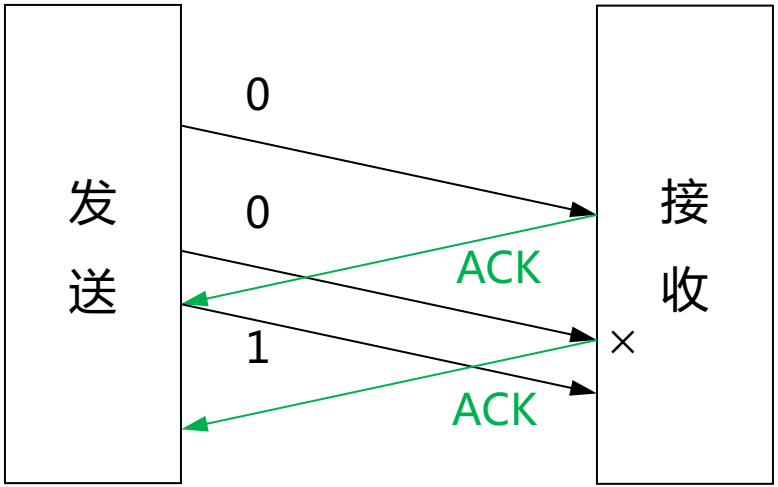
ARQ协议[3]讨论



- 实用的停等协议
- 超时计时器
- 重复帧处理
- 帧的编号

ARQ协议[3]讨论—协议可能的漏洞

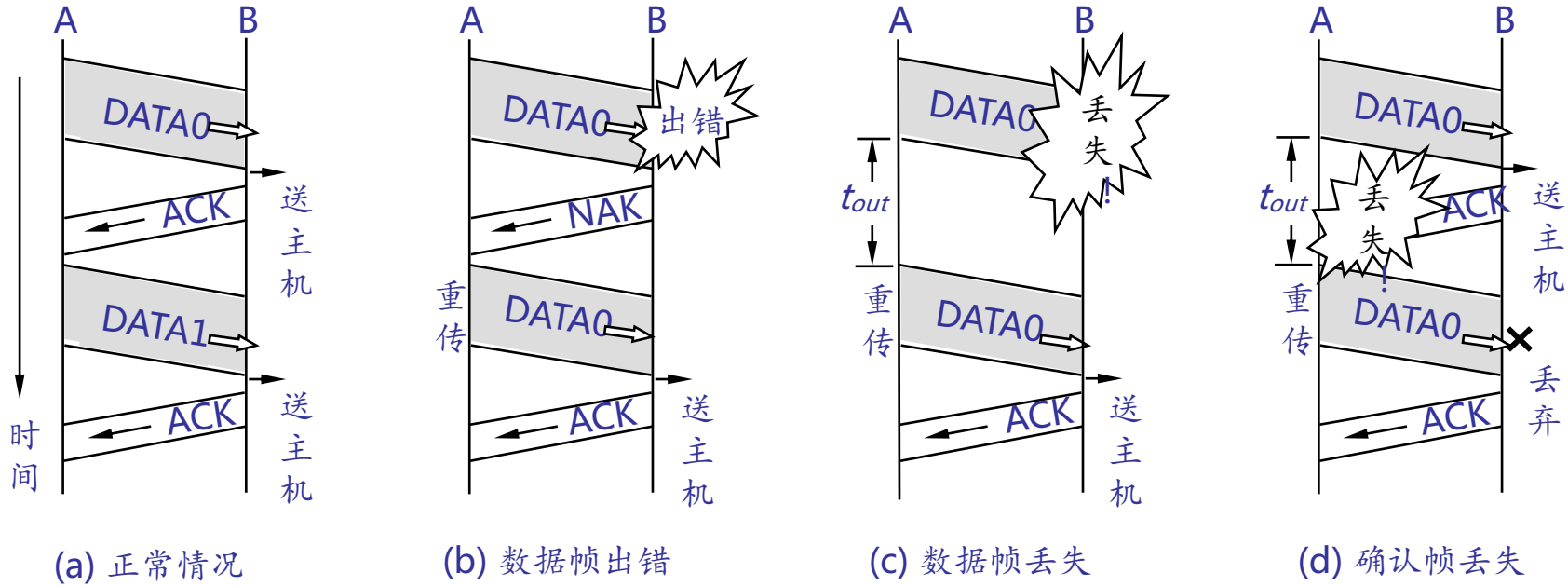
- 如果确认帧中没有序号，则超时时间不能太短，否则协议失败
  - 因此，没有序号的协议，其发送和接收必须严格交替进行



- 即使加上序号，若超时时间设置太短，也会严重影响协议性能

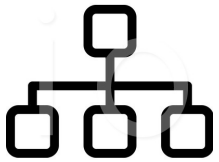
ARQ协议[3]讨论—实用的停等协议

- 有四种情况



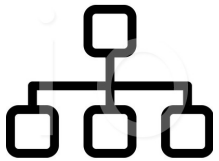
(a) 正常情况　(b) 数据帧出错　(c) 数据帧丢失　(d) 确认帧丢失

ARQ协议[3]讨论—超时计时器



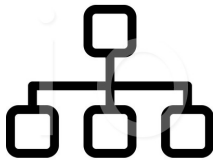
- 结点A发送完一个数据帧时，就启动一个超时计时器(timeout timer)
  - 计时器又称为定时器
- 若到了超时计时器所设置的重传时间 t_{out} 而仍收不到结点 B 的任何确认帧，则结点 A 就重传前面所发送的这一数据帧
- 一般可将重传时间选为，略大于“从发完数据帧到收到确认帧所需的平均时间”

ARQ协议[3]讨论—重复帧处理



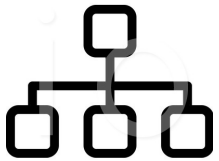
- 使每一个数据帧带上不同的发送序号。每发送一个新的数据帧就把它的发送序号加 1
- 若结点 B 收到发送序号相同的数据帧，就表明出现了重复帧。这时应丢弃重复帧，因为已经收到过同样的数据帧并且也交给了主机 B
- 但此时结点 B 还必须向 A 发送确认帧 ACK，因为 B 已经知道 A 还没有收到上一次发过去的确认帧 ACK

ARQ协议[3]讨论—帧的编号



- 任何一个编号系统的序号所占用的比特数一定是有限的。因此，经过一段时间后，发送序号就会重复
  - 序号占用的比特数越少，数据传输的额外开销就越小
- 对于停止等待协议，由于每发送一个数据帧就停止等待，因此用一个比特来编号就够了
  - 一个比特可表示 0 和 1 两种不同的序号
- 数据帧中的发送序号 seq_nr(S) 以 0 和 1 交替的方式出现在数据帧中
  - 每发一个新的数据帧，发送序号就和上次发送的不一样。用这样的方法就可以使收方能够区分开新的数据帧和重传的数据帧了

ARQ协议[3]分析

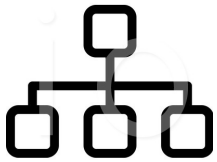


- 可靠传输

  虽然物理层在传输比特时会出现差错，但由于数据链路层的停止等待协议采用了有效的检错重传机制，数据链路层对上面的网络层就可以提供可靠传输的服务

  - 算法
  - 协议的要点
  - 协议的定量分析
  - 重传时间分析
  - 协议的优缺点

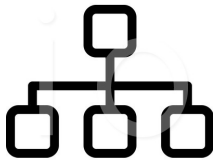
ARQ协议[3]分析—算法的确认环节



- 通常不使用否认帧（实用的数据链路层协议大都是这样的），而且确认帧带有序号 n；
- 确认帧序号 n，一般有两种表示方法：第一种，确认第n号帧已经收到；第二种，确认第 n – 1 号帧已经收到；
- 按照第二种表示法，ACK n 表示“第 n – 1 号帧已经收到，现在期望接收第 n 号帧”
  - ACK1 表示“0 号帧已收到，现在期望接收的下一帧是 1 号帧”
  - ACK0 表示“1 号帧已收到，现在期望接收的下一帧是 0 号帧”

思考：书上P175的协议3代码，采用哪种表示法？

ARQ协议[3]分析—算法描述



- **发送结点**

① 从主机取一个数据帧，送交发送缓存

② V(S)←0

③ seq_nr(S)←V(S)

④ 将发送缓存中的数据帧发送出去

⑤ 设置超时计时器

⑥ 等待 {等待以下⑦、⑧两个事件中最先出现的一个}

⑦ 收到确认帧 ACK *n*

若 *n*=1–V(S)，则从主机取一个新的数据帧，放入发送缓存；

V(S)←[1 - V(S)]，转到 ③

否则，丢弃这个确认帧，转到 ④

⑧ 若超时计时器时间到，则转到 ④

- **接收结点**

V(S/R)：发送/接收状态变量

seq_nr(S)：发送序号

① V(R)←0

② 等待

③ 收到一个数据帧，用CRC检验有无错误

若 seq_nr(S) = V(R) 且无错，则执行 ④

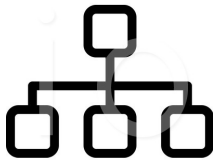
否则(传输有错误或重复帧)丢弃此数据帧，然后转到 ⑥

④ 将收到的数据帧中的数据部分送交上层软件（也就是数据链路层模型中的主机）

⑤ V(R)←[1 - V(R)]

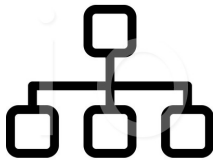
⑥ *n* ←V(R)，发送确认帧 ACK *n*，转到 ②

ARQ协议[3]分析—协议的要点



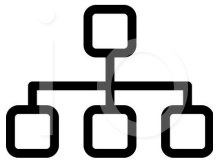
- 只有收到正确序号确认帧 ACK *n* 后，才更新发送状态变量 V(S)一次，并发送新的数据帧
- 接收端接收到数据帧时，就要将发送序号seq_nr(S) 与本地的接收状态变量 V(R) 相比较
  - 若二者相等就表明是新的数据帧，就收下，并发送确认
  - 否则为重复帧，就必须丢弃。但这时仍须向发送端发送确认帧 ACK *n*，而接收状态变量 V(R) 和确认序号 *n* 都不变
- 连续出现相同发送序号的数据帧，表明发送端进行了超时重传。连续出现相同序号的确认帧，表明接收端收到了重复帧
- 发送端在发送完数据帧时，必须在其发送缓存中暂时保留这个数据帧的副本。这样才能在出差错时进行重传。只有确认对方已经收到这个数据帧时，才可以清除这个副本

ARQ协议[3]分析—协议的要点 (2)



- 实用的 CRC 检验器都是用硬件完成的
- CRC 检验器能够自动丢弃检测到的出错帧。因此所谓的“丢弃出错帧”，对上层软件或用户来说都是感觉不到的
- 发送端对出错的数据帧进行重传是自动进行的，因而这种差错控制体制常简称为 ARQ (Automatic Repeat reQuest)，直译是自动重传请求，但意思是自动请求重传

ARQ协议[3]分析—协议的定量分析



- 设 t_f 是一个数据帧的发送时间，且数据帧的长度是固定不变的。显然，数据帧的发送时间 t_f 是数据帧的长度 l_f (bit)与数据的发送速率 C (bit/s)之比，即

$$t_f = l_f / C \text{ (s)}$$

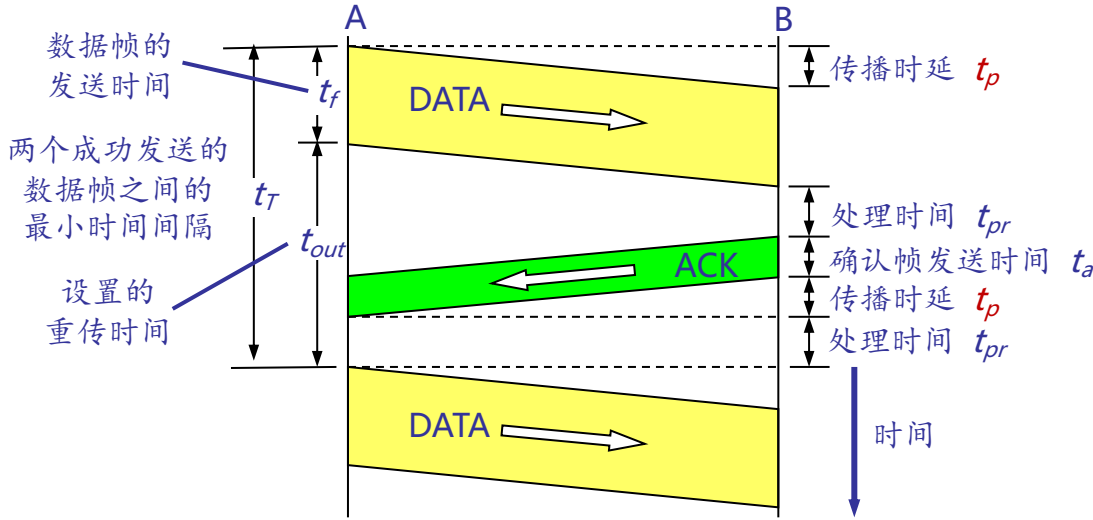
发送时间 t_f 也就是数据帧的发送时延

- 数据帧沿链路传到结点B还要经历一个传播时延 t_p
- 结点 B 收到数据帧要花费时间进行处理，此时间称为处理时间 t_{pr}
- 发送确认帧 ACK 的发送时间为 t_a

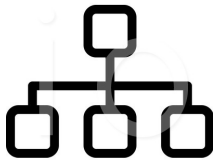
ARQ协议[3]分析—重传时间分析

- 重传时间的作用
  - 数据帧发送完毕后若经过了这样长的时间还没有收到确认帧，就重传这个数据帧
- 为方便起见，我们设重传时间为：$t_{out} = t_p + t_{pr} + t_a + t_p + t_{pr}$
- 设上式右端的处理时间t_{pr}和确认帧的发送时间 t_a都远小于传播时延 t_p，因此可将重传时间取为两倍的传播时延，即

$$t_{out} = 2t_p$$



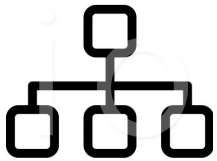
ARQ协议[3]分析—协议的优缺点



- 协议的优缺点
  - 优点：比较简单
  - 缺点：通信信道的利用率不高，也就是说，信道还远远没有被数据比特填满；

    数据吞吐量（时延时段）远小于时延带宽积（参见第一章讲稿）
- 为了克服这一缺点，就产生了另外两种协议
  - 即退后n帧协议和选择重传协议
  - 接下来进一步讨论

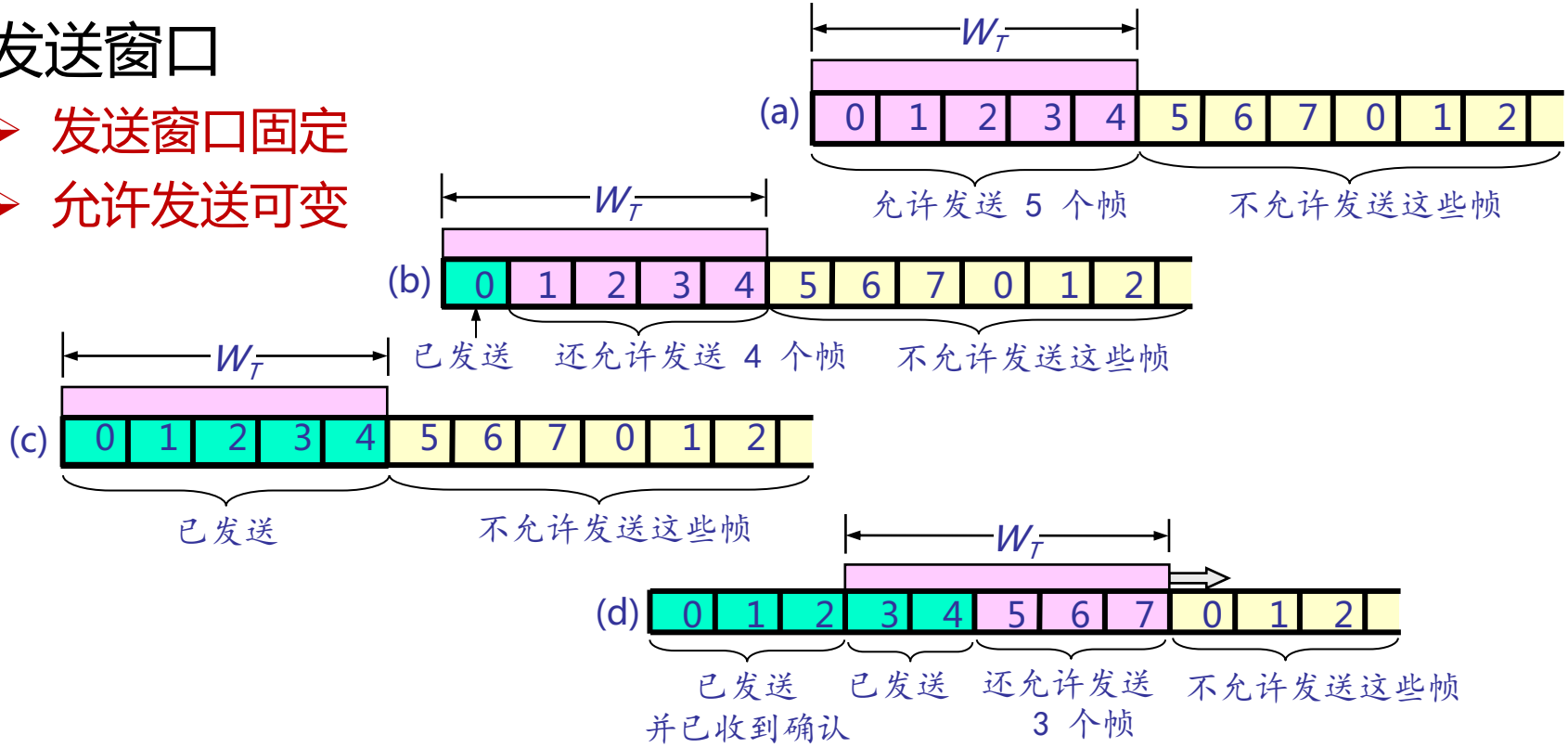
连续ARQ协议



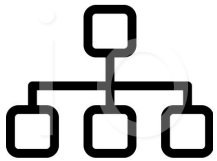
- 连续 ARQ 协议的工作原理
  - 在发送完一个数据帧后，不是停下来等待确认帧，而是连续再发送若干个数据帧
  - 如果这时收到了接收端发来的确认帧，那么还可以接着发送数据帧
  - 由于减少了等待时间，整个通信的吞吐量就提高了
- 滑动窗口的引入
  - 发送端和接收端分别设定发送窗口和接收窗口
  - 发送窗口用来对发送端进行流量控制
  - 发送窗口的大小 W_T，代表在还没有收到对方确认信息的情况下，发送端最多可以发送多少个数据帧

连续ARQ协议—发送窗口

- 发送窗口
  - 发送窗口固定
  - 允许发送可变



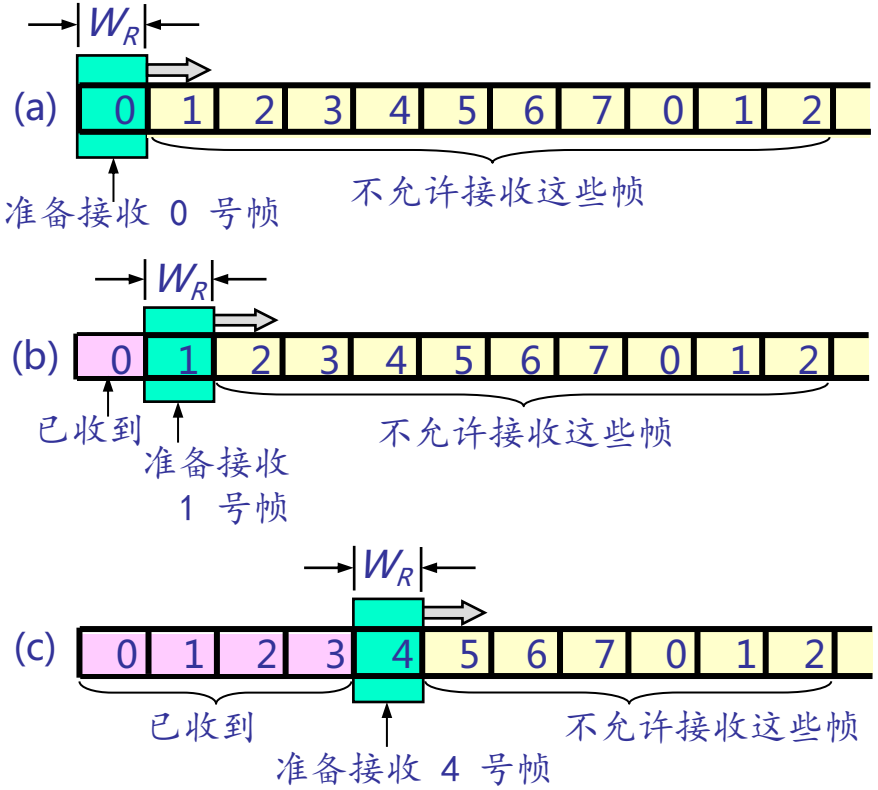
连续ARQ协议—接收窗口



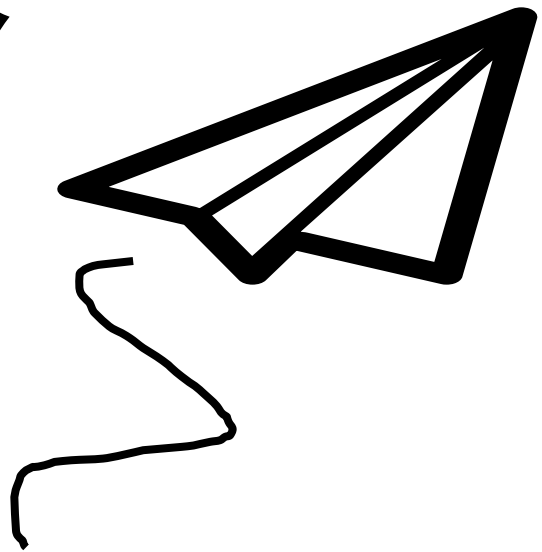
- 接收窗口
  - 在接收端，只有当收到的数据帧的发送序号落入接收窗口内，才允许将该数据帧收下
  - 若接收到的数据帧落在接收窗口之外，则一律将其丢弃
  - 在连续 ARQ 协议中，接收窗口的大小 $W_R = 1$
    - 只有当收到的帧的序号与接收窗口一致时才能接收该帧。否则，就丢弃它
    - 每收到一个序号正确的帧，接收窗口就向前（即向右方）滑动一个帧的位置，同时发送对该帧的确认

连续ARQ协议—滑动窗口重要特性

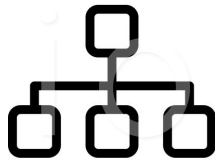
- 只有在接收窗口向前滑动时（与此同时也发送了确认），发送窗口才有可能向前滑动
- 收发两端的窗口按照以上规律不断地向前滑动，因此这种协议又称为滑动窗口协议
- 当发送窗口和接收窗口的大小都等于 1时，就是停止等待协议



本节课程结束

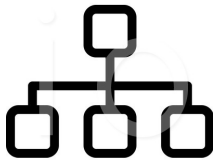


3.4 滑动窗口协议



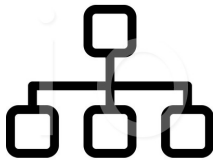
- 滑动窗口原理
- 1位滑动窗口协议
- 退后n帧的协议
- 选择性重传的协议

3.4.1滑动窗口原理



- 信道要求
  - 单工 ——> 全双工
- 捎带 (piggybacking)：暂时延迟待发确认，以便附加在下一个待发数据帧的技术
  - 优点：充分利用信道带宽，减少帧的数目意味着减少“帧到达”中断
  - 带来的问题：复杂
- 本节的三个协议统称滑动窗口协议，都能在实际（非理想）环境下正常工作，区别仅在于效率、复杂性和对缓冲区的要求

滑动窗口协议工作原理



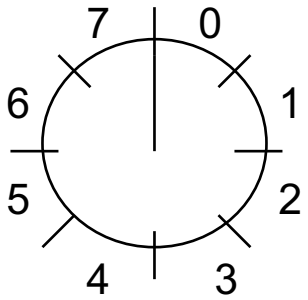
- 滑动窗口协议（Sliding Window Protocol）工作原理
  - 发送的信息帧都有一个序号，从0到某个最大值，$0 \sim 2^n - 1$，一般用n个二进制位表示
  - 发送端始终保持一个已发送但尚未确认的帧的序号表，称为发送窗口
    - 发送窗口的上界表示要发送的下一个帧的序号，下界表示未得到确认的帧的最小编号
    - 发送窗口大小 = 上界 – 下界，大小可变
    - 发送端每发送一个帧，序号取上界值，上界加1；每接收到一个正确响应帧，下界加1
  - 接收端有一个接收窗口，大小固定，但不一定与发送窗口相同
    - 接收窗口的上界表示允许接收的序号最大的帧，下界表示希望接收的帧
    - 接收窗口容纳允许接收的信息帧，落在窗口外的帧均被丢弃。序号等于下界的帧被正确接收，并产生一个响应帧，上界/下界都加1
    - 接收窗口大小不变

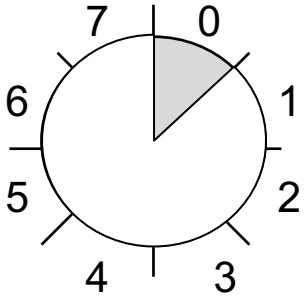
滑动窗口协议工作原理 (2)

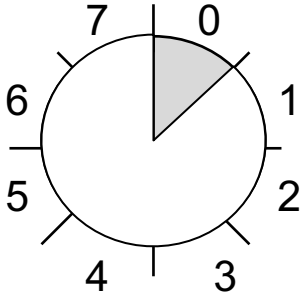
- 三位序号大小为1的滑动窗口

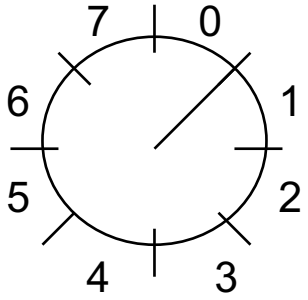
开　始　　发送第一帧　　收到第一帧　　收到第一个确认

发送方

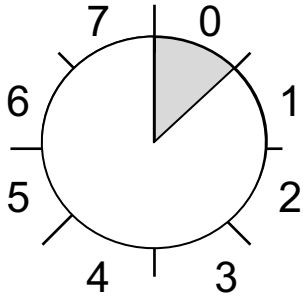


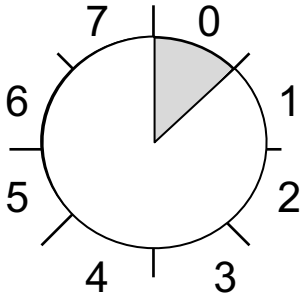


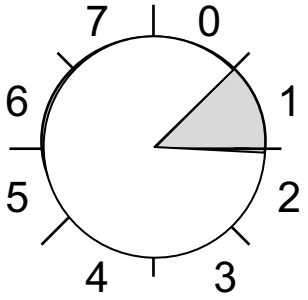


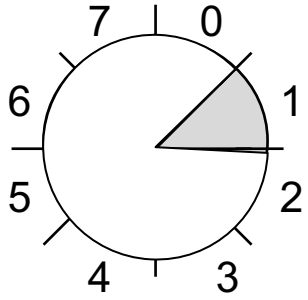


接收方







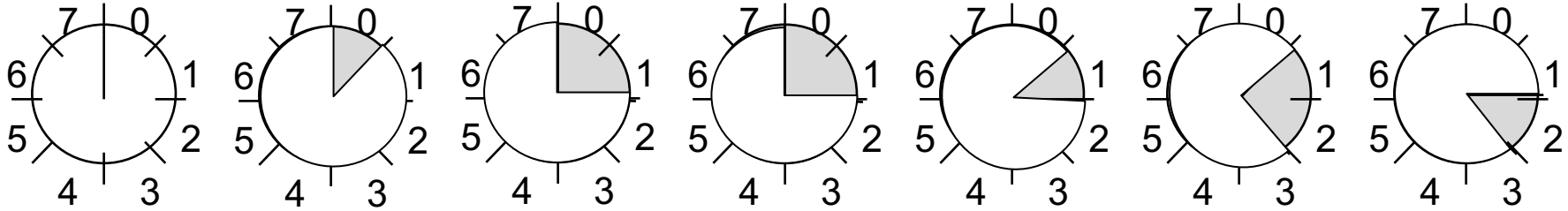


滑动窗口协议工作原理 (3)

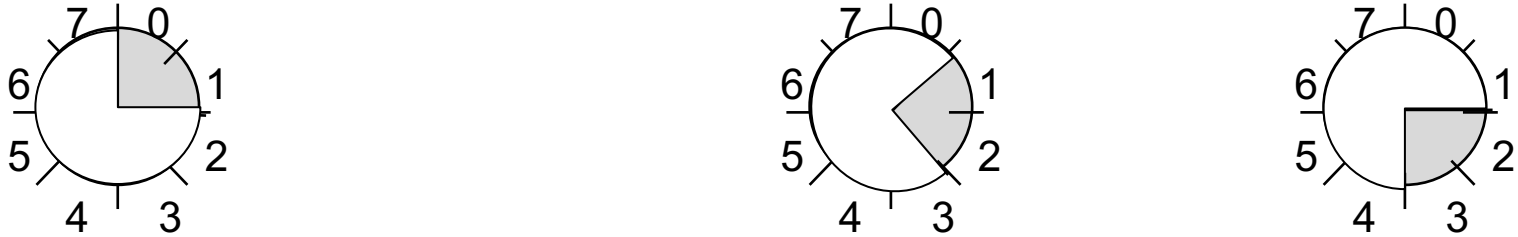
- 三位序号大小为2的滑动窗口

开　始　　发送第一帧　　发送第二帧　　收到第一帧　　收到第一个确认　　收到第二帧　　收到第二个确认　　发送第三帧

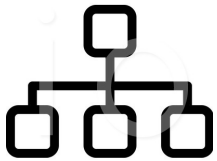
发送方



接收方

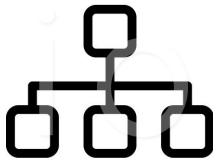


3.4.2 1位滑动窗口协议



- 1位滑动窗口协议[4] (A One Bit Sliding Window Protocol)
  - 协议特点
    - 窗口大小：N = 1
    - 发送序号和接收序号的取值范围：0，1
    - 可进行数据双向传输，信息帧中可含有确认信息（捎带技术）
    - 信息帧中包括两个序号域：发送序号和接收序号（已经正确收到的帧的序号）
  - 工作过程

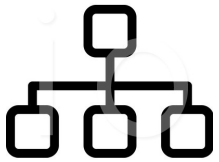
1位滑动窗口协议[4]源代码



```
/* Protocol 4 (Sliding window) is bidirectional. */
   #define MAX SEQ 1                          /* must be 1 for protocol 4 */
   typedef enum {frame arrival, cksum err, timeout} event type;
   #include "protocol.h"
void protocol4 (void)
{
   seq nr next frame to send;                 /* 0 or 1 only */
   seq nr frame expected;                     /* 0 or 1 only */
   frame r, s;                                /* scratch variables */
   packet buffer;                             /* current packet being sent */
   event type event;
   next frame to send = 0;                    /* next frame on the outbound stream */
   frame expected = 0;                        /* frame expected next */
   from network layer(&buffer);               /* fetch a packet from the network layer */
   s.info = buffer;                           /* prepare to send the initial frame */
   s.seq = next frame to send;                /* insert sequence number into frame */
   s.ack = 1 - frame expected;                /* piggybacked ack */
   to physical layer(&s);                     /* transmit the frame */
   start timer(s.seq);                        /* start the timer running */
```

待续

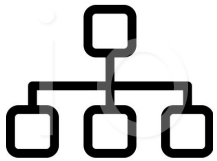
1位滑动窗口协议[4]源代码（续）



```
    while (true) {
        wait for event(&event);                 /* frame arrival, cksum err, or timeout */
        if (event == frame arrival) {           /* a frame has arrived undamaged */
            from physical layer(&r);            /* go get it */
            if (r.seq == frame expected) {      /* handle inbound frame stream */
                to network layer(&r.info);      /* pass packet to network layer */
                inc(frame expected);            /* invert seq number expected next */
            }
            if (r.ack == next frame to send) {  /* handle outbound frame stream */
                stop timer(r.ack);              /* turn the timer off */
                from network layer(&buffer);    /* fetch new pkt from network layer */
                inc(next frame to send);        /* invert sender's sequence number */
            }
        }
        s.info = buffer;                        /* construct outbound frame */
        s.seq = next frame to send;             /* insert sequence number into it */
        s.ack = 1 - frame expected;             /* seq number of last received frame */
        to physical layer(&s);                  /* transmit a frame */
        start timer(s.seq);                     /* start the timer running */
    }
}
```

结束

1位滑动窗口协议[4]问题

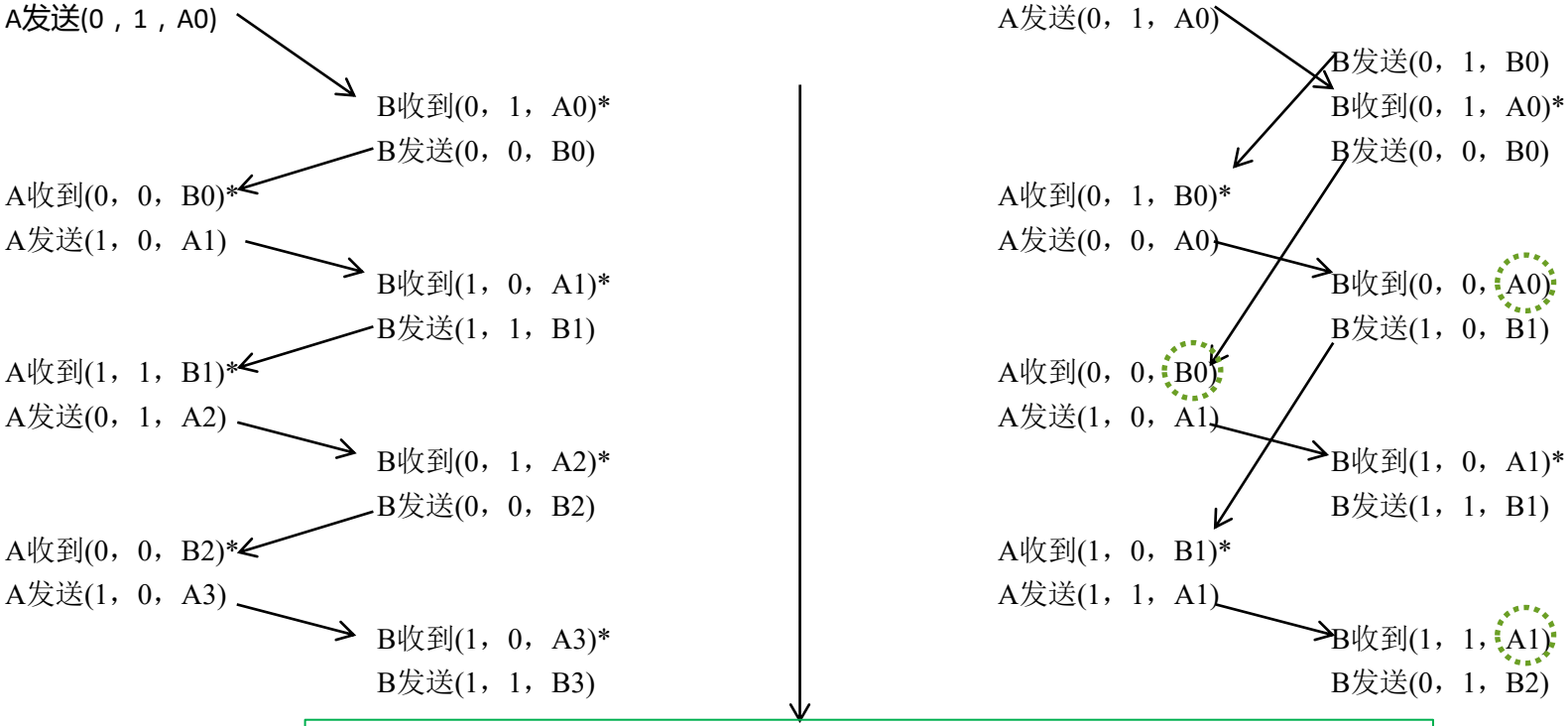


- 存在问题
  - 能保证无差错传输，但是基于停等方式
  - 若双方同时开始发送，则会有一半重复帧

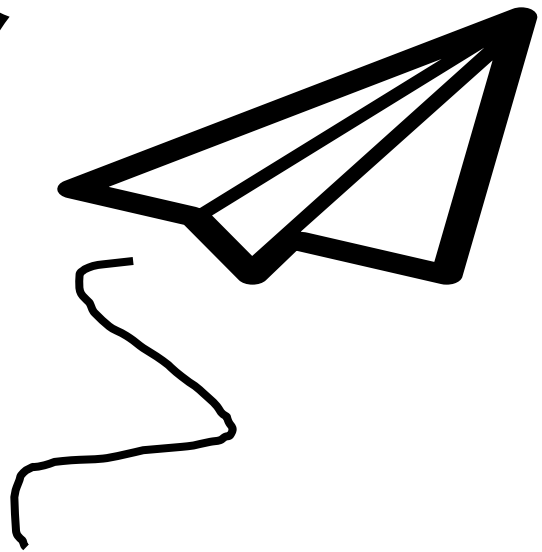
    注意：图 3-17 书上有误(P180)，正确见下页图
  - 效率低，传输时间长

结束

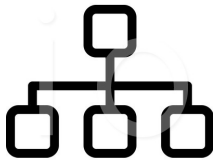
1位滑动窗口协议[4]的两种场景



本节课程结束



3.4.3 退后n帧的协议



- 退后n帧协议[5] (A Protocol Using Go Back n)
  - 为提高传输效率而设计
  - 例如

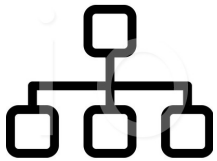
    卫星信道传输速率50kbps，往返传播时延500ms，若传1000bit的帧，使用协议4，则传输一个帧所需时间为：

    发送时延 + 往返传播时延 + 确认发送时延（确认帧很短，忽略发送时间）

    = 1000bit / 50kbps + 250ms + 250ms

    = 20 + 500 = 520ms
  - 信道利用率 = 20 / 520 ≈ 4%

信道利用率计算



- 信道带宽 B 比特/秒，帧长度 L 比特，往返传播时延 2D 秒，则协议4的信道利用率为

$$(L/B) / (L/B + 2D) = L / (L + 2BD)$$

时延带宽积

- 结论：传播时延越大、信道带宽越高、帧长越小时，信道利用率越低
- 解决办法
  - 连续发送多帧后，再等待确认，连续多帧同时传送称为管线化 (pipelining) 技术
  - 连续发送W帧，使得确认刚好返回；将时延带宽积BD转化为帧数：$[BD]_{帧数} = BD / L$

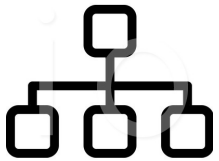
$$W_{MAX} = [BD] + 1 + [BD] = 2[BD] + 1$$

  - 假设滑动窗口协议的发送窗口为W，则：　信道利用率 $\leq \frac{W}{1+2[BD]}$

$W = 1$

- 带来的问题：信道误码率高时，帧的重传非常多

信道利用率计算 (2)



- 卫星信道举例

  B = 50kbps，D = 250ms，L = 1000bit

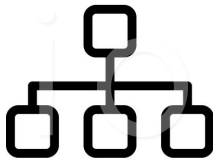
  BD = 12.5kb，[BD] = 12.5

  W_{MAX} = 2[BD] + 1 = 26

$$\text{信道利用率} \le \frac{w}{1+2[BD]}$$

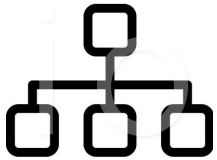
- 如果延迟高，或带宽高，发送方都会很快耗尽其窗口，所以无论时延带宽积多大，发送窗口一定要大
- 高带宽，短延迟，也将导致 W_{MAX} 急速增大
  - 千兆带宽信道，传播延迟为1ms
  - B = 1Gbps，D = 1ms，[BD] = 1Mb
  - 若L=1000b，则W_{MAX} = 2001

滑动窗口协议的改进



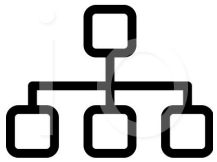
- 协议4的主要问题是信道利用率太低
  - 发送端等待发送下一帧的时间，至少是发送端到接收端信号传播时间的两倍，没有充分利用两条信道的传输能力
- 信道的利用率定义为
  - 数据发送时间，除以从数据开始发送到ACK返回的总耗时
- 协议5是一个发送管线化的协议
  - 在等待ACK的时间内，连续发送
- 出错后重发必须后退n帧
  - 一旦重发定时器超时，必须将已发的n帧全部重发

滑动窗口协议的改进 (2)



- 两种基本方法
  - 退后n帧（go back n）
    - 接收方从出错帧起丢弃所有后继帧
    - 接收窗口为1
    - 对于出错率较高的信道，浪费带宽
  - 选择性重传（selective repeat）
    - 接收窗口大于1，先暂存出错帧的后继帧
    - 只重传坏帧
    - 对最高序号的帧进行确认
    - 接收窗口较大时，需较大缓冲区

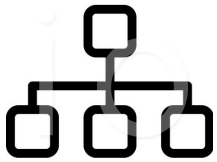
滑动窗口协议的改进 (3)



- 发送窗口的最大值
  - 当用 n 个比特进行编号时，接收窗口的大小为 1，则只有在发送窗口的大小 $W_T \leq 2^n - 1$时，协议才能正确运行
  - 例如，当采用 3 bit 编码时，发送窗口的最大值是 7 而不是 8
  - 为什么？
- 发送、接收窗口之间的关系
  - 序号编码为n比特
  - 发送窗口 W_T 和接收窗口 W_R 的关系

结束

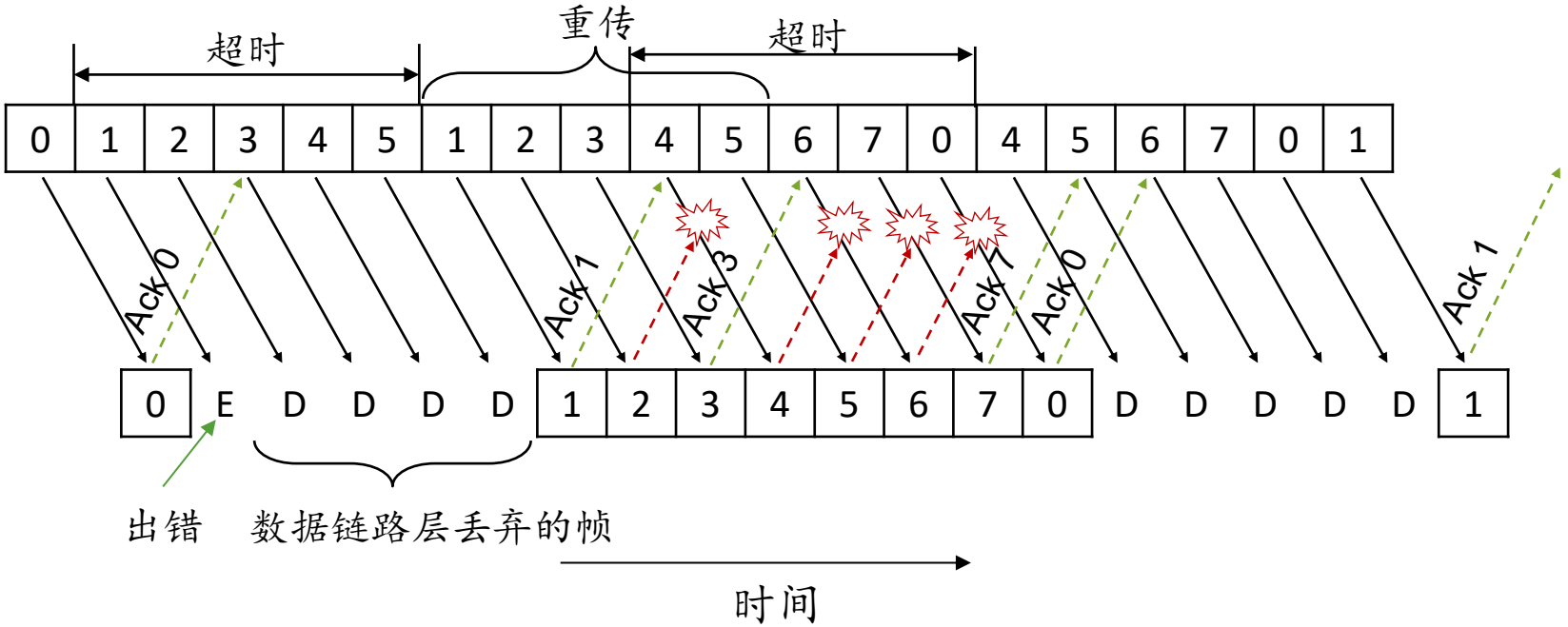
退后n帧协议[5]说明



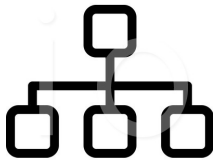
- 设帧序号由3个bit表示，即0 ~ 7，并且$W_T = 7$，$W_R = 1$
- 设发送方有大量数据待发送，由于$W_T = 7$，即有7个发送缓冲区
  - 可连续发送7帧，并每发送一帧将启动一个重发定时器，但缓冲区的覆盖（窗口的旋转）必须在收到Ack之后，因为一旦该帧的重发定时器超时，必须将原缓冲区内的帧重发
- 接收方的 $W_R = 1$
  - 接收方采用捎带确认，并假设需确认时，接收方总能从网络层取到待发送给发送方的数据，以便捎带确认
  - 如期待接收的帧出错，则丢弃此帧及以后所有收到的帧，不发确认（无NAK机制）

退后n帧协议[5]的示意图

- 例：3位序号，发送窗口为7，接收窗口为1

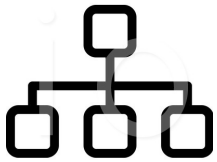


退后n帧协议[5]说明 (2)



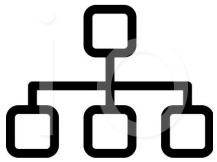
- 当发送方出现超时（错帧的 TimeOut）后，重发自该帧起的所有已发送帧（在当前的缓冲区中）
  - 如发送方连续发送了5帧（1～5），而1#帧无确认，超时后，必须从1#帧起全部重发
- 确认帧序号代表在此之前的帧均无差错
  - 因此，发送方收到Ack3后，表示之前2#帧无差错，即使之前未收到Ack2
  - 如果确认帧连续丢失，也将导致超时，图中将从4#帧重发4#帧及之后的所有帧
- 协议5源代码
  - 参见教材P182，图 3-19

发送窗口的最大值为Max_seq



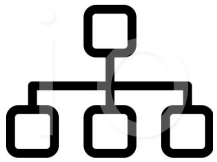
- 当用 n 个比特进行编号时，接收窗口的大小为 1，则发送窗口的大小 $W_T \leq 2^n - 1$，协议才能正常工作，我们分析一下原因。
- 例如，当采用 3 bit 编码时，如果发送窗口的最大值是 $8 = 2^3$，请考虑如下场景：
  - 第一次，发送方发送了0~7个帧
  - 7号帧的捎带确认返回到发送方
  - 发送方发送另外0~7号，共8个帧，这8个帧有可能全部正确到达，也有可能全部出错
  - 现在7号帧的另一个捎带确认返回
  - 问题：发送方如何能判别发送的情况？即第二个确认是对第一次发送的确认，还是对第二次发送的确认？

退后n帧协议[5]源代码—（1）



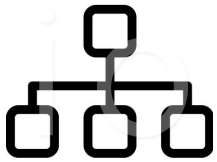
```
/* Protocol 5 (Go-back-n) allows multiple outstanding frames. The sender may transmit up
   to MAX SEQ frames without waiting for an ack. In addition, unlike in the previous
   protocols, the network layer is not assumed to have a new packet all the time. Instead,
   the network layer causes a network layer ready event when there is a packet to send. */

#define MAX SEQ 7
typedef enum {frame arrival, cksum err, timeout, network layer ready} event type;
#include "protocol.h"

static boolean between(seq nr a, seq nr b, seq nr c)
{
/* Return true if a <= b < c circularly; false otherwise. */
  if (((a <= b) && (b < c)) || ((c < a) && (a <= b)) || ((b < c) && (c < a)))
     return(true);
   else
     return(false);
}

static void send data(seq nr frame nr, seq nr frame expected, packet buffer[ ])
{
/* Construct and send a data frame. */
  frame s; /* scratch variable */

  s.info = buffer[frame nr];                          /* insert packet into frame */
  s.seq = frame nr;                                   /* insert sequence number into frame */
  s.ack = (frame expected + MAX SEQ) % (MAX SEQ + 1); /* piggyback ack */
  to physical layer(&s);                              /* transmit the frame */
  start timer(frame nr);                              /* start the timer running */
}
```

退后n帧协议[5]源代码—（2）



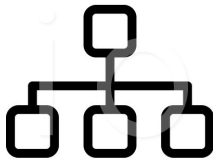
```
void protocol5(void)
{
 seq nr next frame to send;          /* MAX SEQ > 1; used for outbound stream */
 seq nr ack expected;                /* oldest frame as yet unacknowledged */
 seq nr frame expected;              /* next frame expected on inbound stream */
 frame r;                            /* scratch variable */
 packet buffer[MAX SEQ + 1];         /* buffers for the outbound stream */
 seq nr nbuffered;                   /* number of output buffers currently in use */
 seq nr i;                           /* used to index into the buffer array */
 event type event;

 enable network layer();             /* allow network layer ready events */
 ack expected = 0;                   /* next ack expected inbound */
 next frame to send = 0;             /* next frame going out */
 frame expected = 0;                 /* number of frame expected inbound */
 nbuffered = 0;                      /* initially no packets are buffered */

 while (true) {
  wait for event(&event);            /* four possibilities: see event type above */
```

退后n帧协议[5]源代码—（3）



```
while (true) {
 wait for event(&event);                          /* four possibilities: see event type above */

 switch(event) {
  case network layer ready:                        /* the network layer has a packet to send */
     /* Accept, save, and transmit a new frame. */
     from network layer(&buffer[next frame to send]); /* fetch new packet */
     nbuffered = nbuffered + 1;                    /* expand the sender's window */
     send data(next frame to send, frame expected, buffer);/* transmit the frame */
     inc(next frame to send);                      /* advance sender's upper window edge */
     break;

  case frame arrival:                              /* a data or control frame has arrived */
     from physical layer(&r);                      /* get incoming frame from physical layer */

     if (r.seq == frame expected) {
        /* Frames are accepted only in order. */
        to network layer(&r.info);                 /* pass packet to network layer */
        inc(frame expected);                       /* advance lower edge of receiver's window */
     }
```

退后n帧协议[5]源代码—（4）

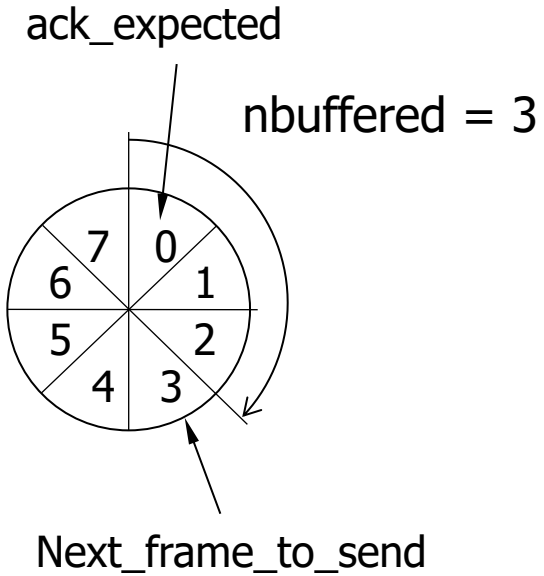


```
          /* Ack n implies n - 1, n - 2, etc. Check for this. */
        while (between(ack expected, r.ack, next frame to send)) {
            /* Handle piggybacked ack. */
            nbuffered = nbuffered - 1;          /* one frame fewer buffered */
            stop timer(ack expected);           /* frame arrived intact; stop timer */
            inc(ack expected);                  /* contract sender's window */
        }
        break;

    case cksum err: break;                      /* just ignore bad frames */

    case timeout:                               /* trouble; retransmit all outstanding frames */
        next frame to send = ack expected;      /* start retransmitting here */
        for (i = 1; i <= nbuffered; i++) {
            send data(next frame to send, frame expected, buffer);/* resend frame */
            inc(next frame to send);            /* prepare to send the next one */
        }
  }
  if (nbuffered < MAX SEQ)
      enable network layer();
  else
      disable network layer();
  }
}
```

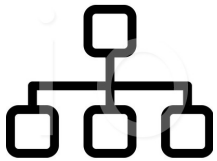
退后n帧协议[5]代码说明

- 初始化

```
void protocol5(void)
{
  enable_network_layer();
  ack_expected = 0;
  next_frame_to_send = 0;
  frame_expected = 0;
  nbuffered = 0;
```



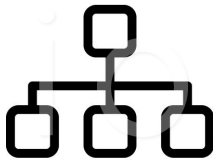
退后n帧协议[5]代码说明 (2)



- 主循环

```
while(true)
 {wait_for_event(&event);
    switch(event)
      {case network_layer_ready:    network_layer_ready处理；
       case frame_arrival:          frame_arrival处理；
       case cksum_err: break;       忽略坏帧
       case timeout:                timeout处理；
      }
    if nbuffered < MAX_SEQ)
        enable_network_layer();     /* 如窗口未满，则允许网络层事件 */
    else disable_network_layer();   /* 如窗口已满，则不允许网络层事件 */
  }
}
```

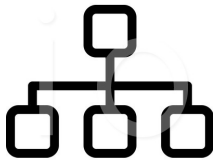
退后n帧协议[5]代码说明 (3)



- network_layer_ready 处理
  - 装配一个数据帧并发送
  - 发送缓冲区数+1
  - 准备发送下一数据帧

```
from_network_layer(&buffer[next_frame_to_send]);
nbuffered = nbuffered + 1;
send_data(next_frame_to_send, frame_expected, buffer);
inc(next_frame_to_send);
```

退后n帧协议[5]代码说明 (4)



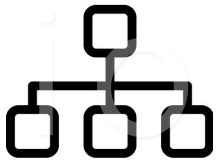
- frame_arrival 处理

```
from_physical_layer(&r);
if(r.seq == frame_expected)
    {to_network_layer(&r.info);
    inc(frame_expected);
}
while (between(ack_expected, r.ack, next_frame_to_send))
   {nbuffered = nbuffered - 1;
    stop_timer(ack_expected);
    inc(ack_expected);
   }
```

如收到一个数据帧，则交网络层，并期待接收下一数据帧

如收到一个ACK，则释放一个缓冲区，定时器复位，并期待接收下一ACK

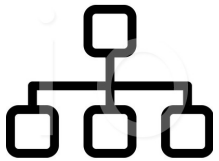
退后n帧协议[5]代码说明 (5)



- Timeout 处理
  - 从等待确认的帧开始全部重发

```
next_frame_to_send = ack_expected;
for ( i = 1; i <= nbuffered; i++)
  {
    send_data(next_frame_to_send, frame_expected, buffer);
    inc(next_frame_to_send);
}
```

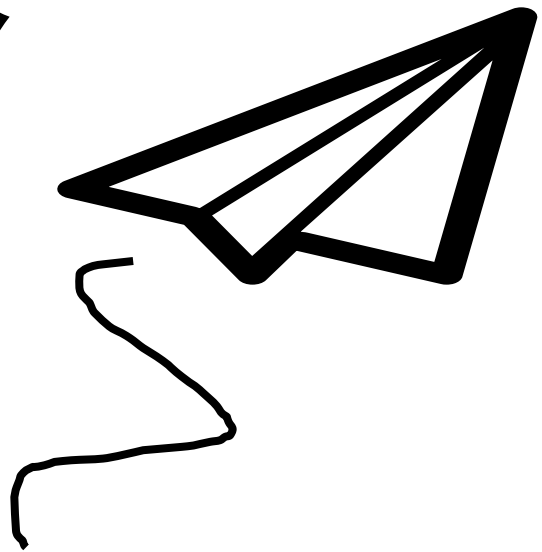
退后n帧协议小结



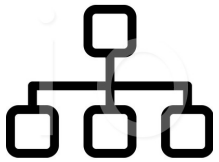
$$（W_T = 2^n - 1，\ W_R = 1）$$

- 协议5，即管线化协议
  - 是一个很实用的点对点可靠传输的协议，特别适用于差错率较低的信道，此时，信道利用率很高
- 如果帧序号由n位组成，则发送窗口$W_T = 2^n - 1$，接收窗口$W_R = 1$

本节课程结束



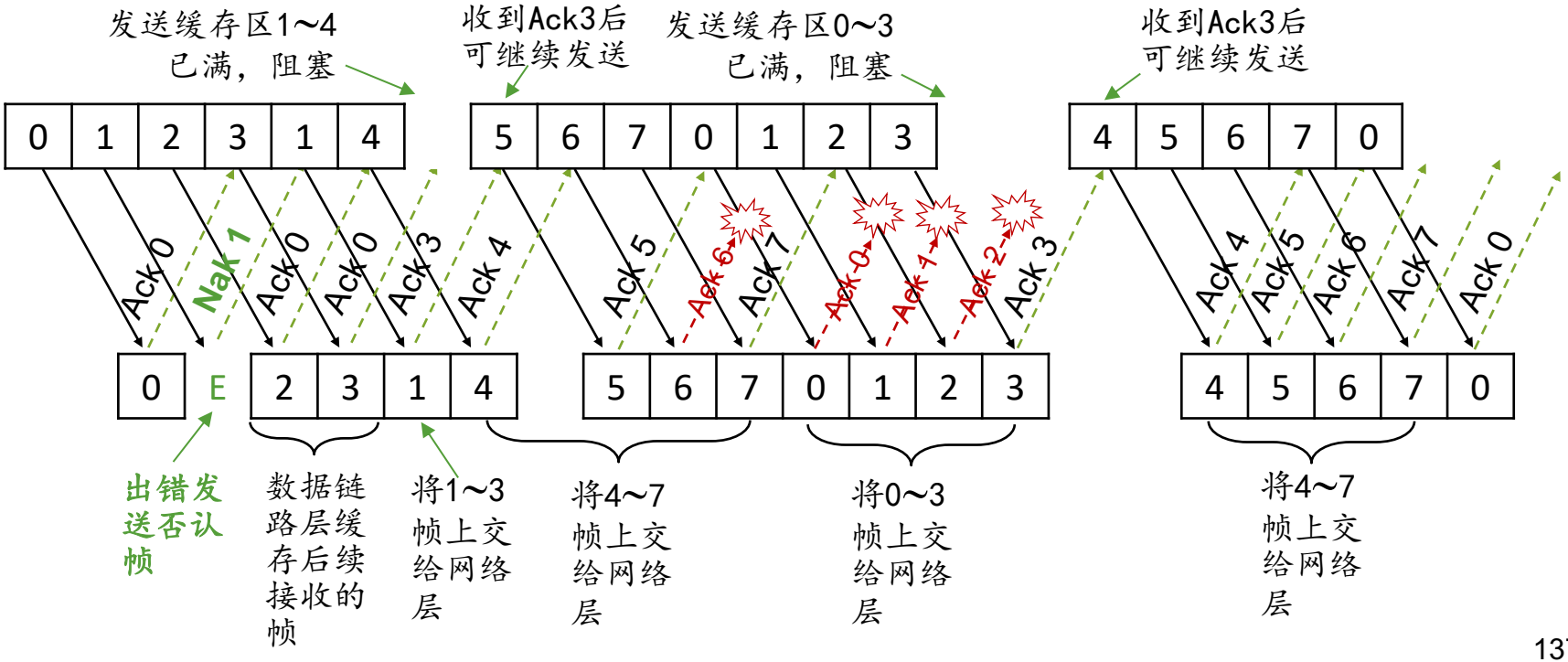
3.4.4 选择性重传的协议



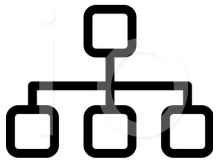
- 在差错率较高的信道上，协议5信道利用率低
  - 使用协议5可能导致大量的重发，从而使信道利用率大幅度降低
  - 设帧长为1500 Byte，连续发送7帧共 84000 bit，当信道的误码率高于 $1.2X10^{-5}$，信道的利用率将非常低
- 协议6是一个选择性重发的滑动窗口协议
  - 发送方在某帧的重发定时器超时（没有收到该帧的Ack）后，只要重发该帧即可，而不必重发所有已发送的帧
- 发送方可根据所定义的发送窗口大小，连续发送
  - 通常发送方将当前缓冲区内的帧连续发送完后，等待确认，发送缓冲区的覆盖（窗口的旋转）将依据收到的Ack的序号，该序号的帧及其以前的所有帧都可被覆盖

选择性重传协议[6]的示意图

例：3位序号，有差错时仅重发出现差错的帧（W_T =4，W_R = 4）

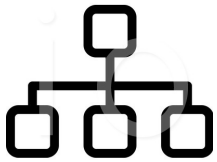


选择性重传协议[6]说明



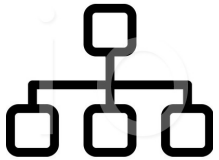
- 设帧序号由3个bit表示，即0 ~ 7，并且假设$W_T = 4$，$W_R = 4$
- 设发送方有大量数据等待发送给对方，由于$W_T = 4$，即有4个发送缓冲区
  - 可连续发送4帧，并每发送一帧将启动一个（带帧序号的）重发定时器，但缓冲区的覆盖必须在收到确认之后，因为一旦该帧的重发定时器超时，必须将原缓冲区内的帧重发
- 协议6中，接收方的$W_R = 4$
  - 如期待接收的帧n#丢失或出错， 接收方发送否认帧Nak，确认号为n#，对后继到达正确的 (n+1)#帧可照常接收
  - 由于(n+1)#帧正确接收，也需要确认，但此确认应有别于对正期待接收帧的确认，此时Ack确认号为(n-1)，即期待接收第n帧
  - 如果采用期待的(n+1)#帧序号，来确认当前n#帧，则无论是Ack或Nak，其中的确认号都必须为n

辅助定时器和否定性确认



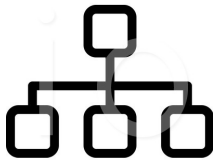
- 协议6中，接收方定义了一个辅助定时器
  - 接收方采用捎带确认的前提是有数据帧发送给发送方
  - 然而并非需要捎带确认时，接收方上层总是有数据需发送的，所以接收方定义了一个辅助定时器，凡收到一个正确的数据帧并需发送确认时，立即启动辅助定时器
  - 在辅助定时器溢出前，上层有数据帧发送，则可捎带确认，如辅助定时器溢出，则立即发送一个单独的确认，以免发送方的重发定时器超时而导致的数据帧的重发
- 协议6中，定义了一个否定性确认的帧格式
  - 当接收方接收到一个有问题的帧时，发送一个否定性确认NAK，包括以下两种情况
    1. CRC校验错
    2. CRC校验正确但帧序号错

选择性重传协议[6]源代码



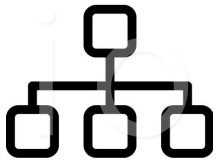
```
void protocol6(void)
{
        enable_network_layer();
        ack_expected = 0;
        next_frame_to_send = 0;
        frame_expected = 0;
        too_far = NR_BUFS;
        nbuffered = 0;
        for (i = 0; i < NR_BUFS;i++)  arrived[i] = false;
        while (true)
        {
            wait_for_event(&event);
```

选择性重传协议[6]源代码 (2)



```
switch (event)
 { case network_layer_ready :      发送处理;
   case frame_arrival:             帧到达处理;
   case cksum_err:          /* 收到一个坏帧且没有发过nak，则发nak */
        if (no_nak)    send_frame(nak,0, frame_expected, out_buf);
   case timeout:            /* 等待确认帧超时，重发 */
        send_frame(data, oldest_frame, frame_expected, out_buf);
  case ack_timeout:         /* 辅助定时器超时，发一单独的ack */
        send_frame(ack, 0, frame_expected, out_buf);
 }
if (nbuffered < NR_BUFS)
        enable_network_layer(); /* 如窗口未满，则允许网络层事件 */
else
        disable_network_layer(); /* 如窗口满，则不允许网络层事件 */
}
}
```

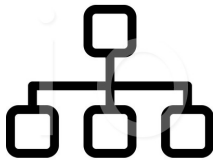
选择性重传协议[6]源代码 (3)



- 发送处理

```
nbuffered = nbuffered + 1;          /* 已用窗口数 + 1 */

from_network_layer(&out_buf [next_frame_to_send % NR_BUFS]); /* 取新的分组 */

send_frame(data,next_frame_to_send, frame_expected,out_buf); /* 发送该帧 */

inc(next_frame_to_send);            /* 发送窗口的前沿+1 */
```

选择性重传协议[6]源代码 (4)



- 帧到达处理

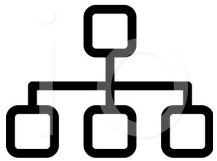
```
from_physical_layer(&r);
if (r.kind == data)
{if((r.seq != frame_expected) && no_nak)
         send_frame(nak, 0, frame_expected, out_buf);
     else start_ack_timer();
 if (between(frame_expected, r.seq, too_far)&&(arrived[r.seq % NR_BUFS]==false))
     {arrived[r.seq % NR_BUFS] = true;
      in_buf[r.seq % NR_BUFS] = r.info;
      while (arrived[frame_expected % NR_BUFS])
         {to_network_layer(&in_buf[frame_expected % NR_BUFS]);
          no_nak = true;   arrived[frame_expected % NR_BUFS] = false;
          inc(frame _expected);   inc(too_far);  start_ack_timer();
 } } }
```

如序号错则发nak，否则启动辅助定时器

如序号在接收窗口范围内，接受该帧

如顺序正确，则交网络层，调整参数，启动辅助定时器

选择性重传协议[6]源代码 (5)



- 续帧到达处理

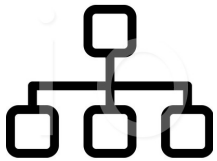
```
if (r.kind == nak) &&
 (between(ack_expected,(r.ack + 1)%( MAX_SEQ+1), next_frame_to_send))
   send_frame(data,(r.ack+1)%( MAX_SEQ+1),frame_expected, out_buf);

while (between(ack_expected, r.ack, ext_frame_to_send ))
  {nbuffered = nbuffered - 1;
      stop_timer(ack_expected % NR_BUFS) ;
      inc(ack _expected);
 }
```

r.ack+1正是接收方期待接收的帧

处理ack

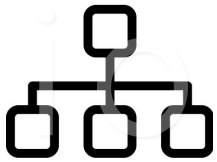
已用窗口数-1，启动辅助定时器，等待下一个ack

选择性重传协议[6]分析



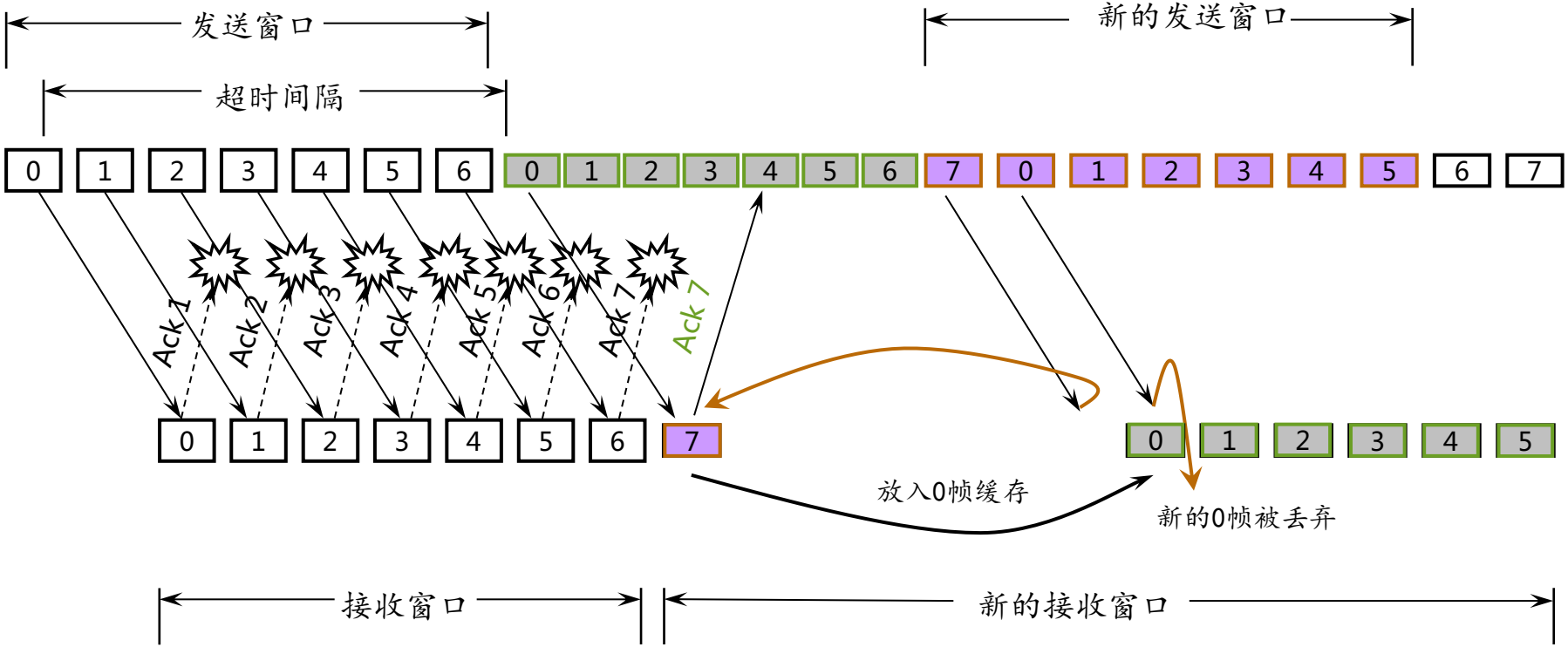
- 极端情况分析：当 n=3，即 帧号为 0 ~ 7，且 发送窗口 W_T = 接收窗口 W_R= 7
  - 发送方当前的发送窗口为0~6，连续发送了7帧，帧号为0 1 2 3 4 5 6，然后等待确认
  - 接收方在初始化后，接收窗口为0~6
    - 在正确收到0#帧后，由于捎带确认，所以立即启动辅助定时器
    - 在辅助定时器没有超时之前，发送方发送的7帧都正确地收到
    - 辅助定时器超时，接收方立即发送Ack7，意即0 ~ 6帧全部收到，并期待接收第7号帧
    - 然后取出分组交网络层，清缓冲区并调整窗口为7 0 1 2 3 4 5
  - 发送方一直在等待确认，但接收方发送的Ack7由于某种原因丢失了
    - 在重发定时器陆续超时的过程中，发送方又重发了0 1 2 3 4 5 6帧，并继续等待确认

选择性重传协议[6]分析 (2)



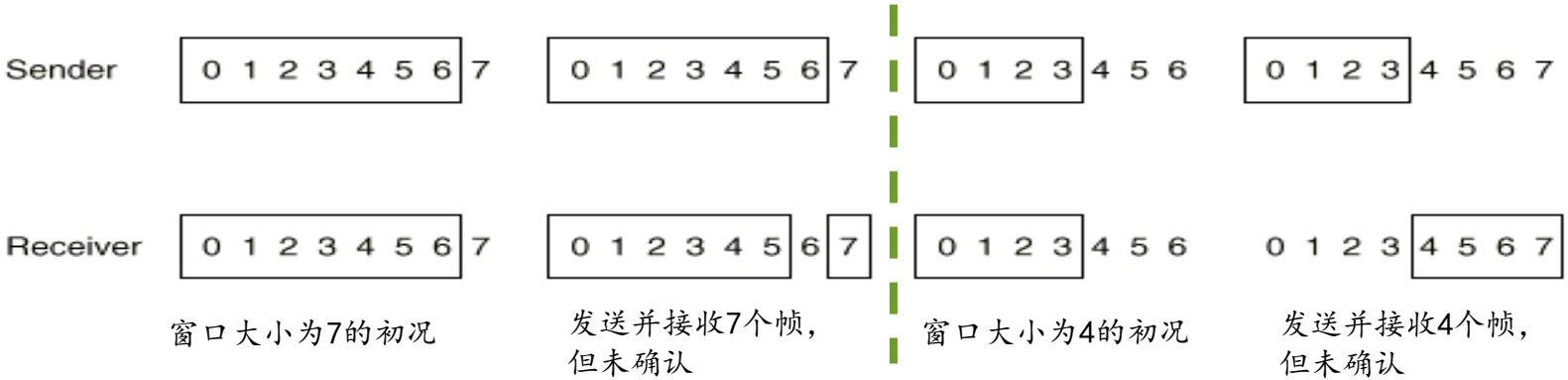
- 接收方收到0 1 2 3 4 5 6帧，认为是第二批来的帧
  - 按正常处理，发现0 1 2 3 4 5均在其接收窗口内，当然接收并存入缓冲，6丢弃
  - 由于期待接收的7号帧未到，所以只能仍发Ack7，意即再次确认上次收到的0～6
    - 同时由于7号帧未到，所以，已收到的0 1 2 3 4 5帧不能上交网络层
- 在发送方看来，收到了Ack7后会认为，重发的0～6号帧总算收到了
  - 于是，调整窗口为7 0 1 2 3 4 5，又从网络层取分组，并发送第二批帧
- 接收方在收到7 0 1 2 3 4 5后，发现0～5帧已在缓冲区中，重复应丢弃
  - 第7帧接收，发送Ack6，然后将7 0 1 2 3 4 5交网络层，清缓冲区、调整接收窗口
- 此时，接收方的网络层发现：数据链路层交来的第二批分组中的0 1 2 3 4 5与原来的重复
  - 协议失败！！！

极端情况图示


发送窗口
新的发送窗口
超时间隔
0 1 2 3 4 5 6 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
Ack 1
Ack 2
Ack 3
Ack 4
Ack 5
Ack 6
Ack 7
Ack 7
0 1 2 3 4 5 6 7
0 1 2 3 4 5
放入0帧缓存
新的0帧被丢弃
接收窗口
新的接收窗口

选择性重传协议[6]分析 (3)

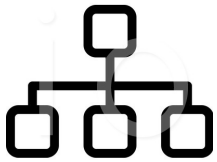
- 当$W_T=W_R=7$时，协议6失败的原因
  - 原因在于接收窗口过大，窗口中的有效顺序号在调整前和调整后有重叠



  - 所以，通常

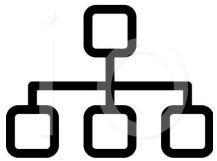
发送窗口+ 接收窗口 $\leq 2^n$　　且：　发送窗口 = 接收窗口

选择性重传协议小结



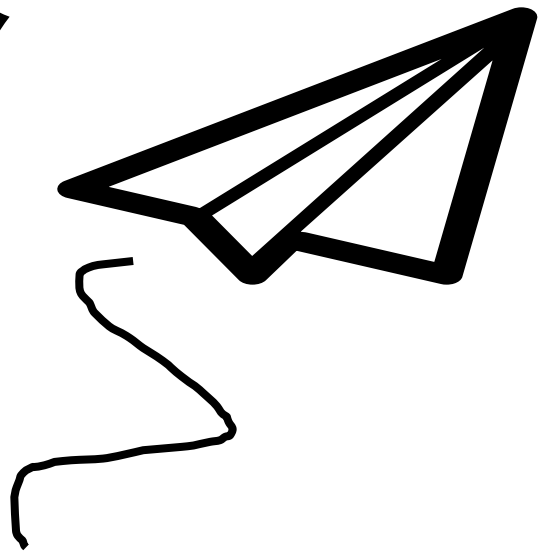
- 协议6中：如果帧序号由n位组成，即0 ~ $2^n - 1$
  - 发送窗口 = 接收窗口 = $(MAX\_SEQ + 1) / 2 = 2^{n-1}$
- 协议6增加了否定性确认Nak，其中的确认号为当前所期待接收的帧的序号
  - 当收到一个CRC校验错的帧，则发一个Nak
  - 当首次收到一个CRC校验正确、但序号错的帧，则发送一个Nak
- 协议6增加了一个辅助定时器：当辅助定时器超时，则立即发送一个Ack，其中的确认号为当前所期待接收的帧的序号
  - 当收到一个CRC校验正确、序号也正确的帧，将启动一个辅助定时器（为等待捎带）
  - 当再次收到一个CRC校验正确、但序号错的帧，因为已发送过Nak ，所以不再发Nak，此时也将启动一个辅助定时器

可靠性传输

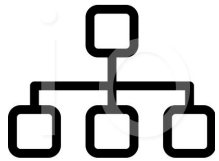


- 差错控制：校验、重发和序号
  - 避免帧错误的保证：帧的校验
  - 避免帧丢失的保证：超时和重发
  - 避免帧重复的保证：帧有序号
- 流量控制：窗口协议
  - 发送方和接收方之间的协调
- 可靠传输
  - 通过确认和重传机制
  - 传输层协议，如TCP，也提供可靠传输服务
  - 链路层的可靠传输服务通常用于高误码率的连路上，如无线链路

本节课程结束

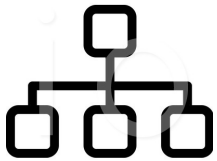


3.5 常用的数据链路层协议



- 数据链路层协议分类
- SONET上的数据包
- 非对称数字用户线路

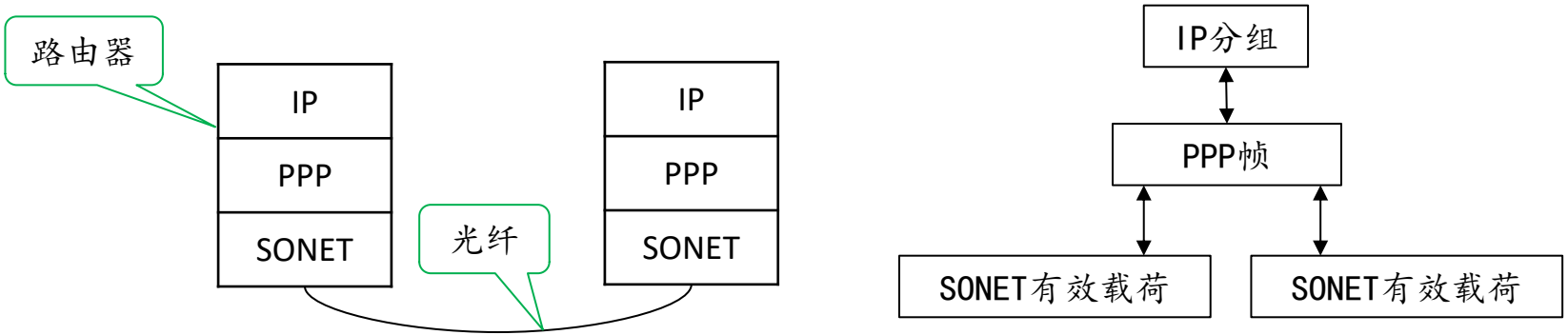
3.5.1 数据链路层协议分类



- 主要的数据链路层协议
  - 面向字符的链路层协议
    - IBM的二进制同步通信规程（BSC）
    - DEC的数字数据通信报文协议（DDCMP）
    - PPP
  - 面向比特的链路层协议
    - IBM的SNA使用的数据链路协议SDLC（Synchronous Data Link Control protocol）
    - ISO修改SDLC，提出HDLC（High-level Data Link Control）
- 因特网的数据链路层连接
  - 广播式：建筑物内（局域网）
  - 点到点：光纤连接与电话线连接（广域网）

3.5.2 SONET上的数据包

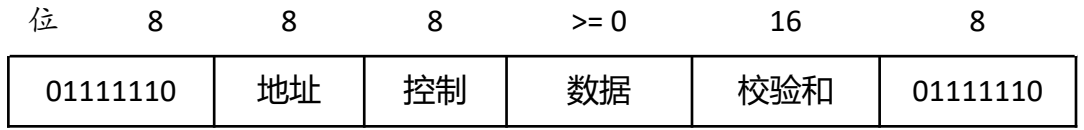
- 同步光纤网 SONET (Synchronous Optical NETwork)
- 点到点协议 PPP(Point-to-Point Protocol)



SONET之上的数据包

点到点协议 PPP

- 高级数据链路控制 HDLC(High-Level Data Link Control)
  - 面向比特
  - 帧格式



- 串行线路的因特网协议 SLIP(Serial Line Internet Protocol)
  - 字符填充、不纠错、固定IP、无身份验证
- 点到点协议 PPP
  - 成帧方法
  - 链路控制协议LCP
  - 网络控制协议NCP

点到点协议 PPP (2)

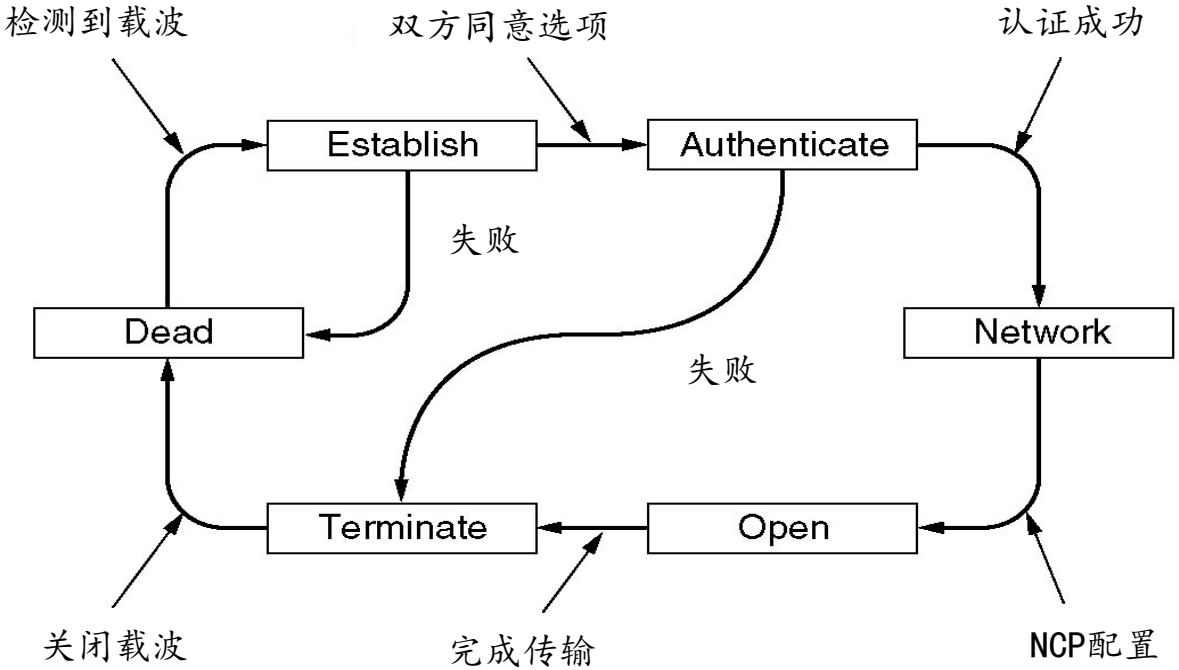
- PPP帧格式

字节	1	1	1	1或2	可变	2	1
	标记 01111110	地址 11111111	控制 00000011	协议	有效载荷	校验和	标记 01111110

  - 控制字段
    - 缺省为00000011，表示无序号帧，不提供使用序号和确认的可靠传输
    - 不可靠线路上，也可使用有序号的可靠传输
  - 协议域
    - 以0开始编码：表示IPv4、IPv6、IPX、AppleTalk等协议
    - 以1开始编码用于PPP配置协议
  - 有效载荷字段：变长，缺省为1500字节
  - 校验和字段：可协商使用4字节，一般用16位CRC或32位CRC

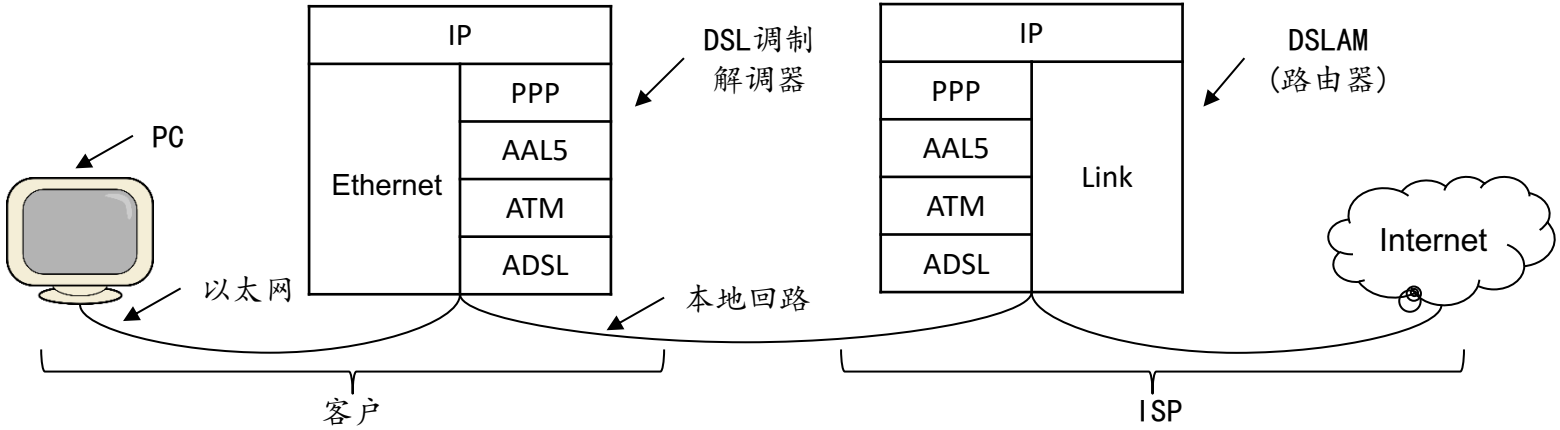
点到点协议 PPP (3)

- PPP 链路 建立/释放 状态转换图

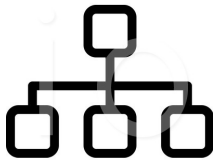


3.5.3 非对称数字用户线路

- ADSL协议栈
- 异步传输模式 ATM (Asynchronous Transfer Mode)
  - 信元：53 B =5 + 48
- ATM适应层5 (AAL5，ATM Adaptation Layer 5)



本章小结



- 数据链路层的功能
- 组帧
- 差错控制
  - 检错编码
  - 纠错编码
- 基本数据链路层协议
  - 理想信道单工协议
  - 无错信道上单工停-等协议、有错信道上单工停-等协议
- 滑动窗口协议
  - 1位滑动窗口、回退N帧、选择重传
- 常见的数据链路层协议
  - PPP协议、HDLC协议

本章课程结束