目录

循环冗余校验码 (P47)

1. 二进制码的多项式

如果我们把一个二进制码字的各位看作是一个多项式的系数，那么一个 n 位码字 $U=U_{n-1}U_{n-2}.....U_1U_0$，可以表示成一个多项式：$U(x)=U_{n-1}x^{n-1}+U_{n-2}x^{n-2}+.....+U_1x+U_0$，那么 U(x)称为 U 的多项式。

2. 产生循环冗余码的方法

（1）约定一个生成多项式 G(x)，设其最高阶次为 m；

（2）设待发送的信息单元为 n 位的 U，其对应的多项式为 U(x)，用 $U(x)\bullet x^m$ 除以生成多项式 G(x)(注意：除法按模 2 运算法则)，得到一个余式 R(x)；

（3）设余式 R(x)对应的 m-1 位二进制码为 R，将 R 放在 U 之后就构成了循环冗余检验码。

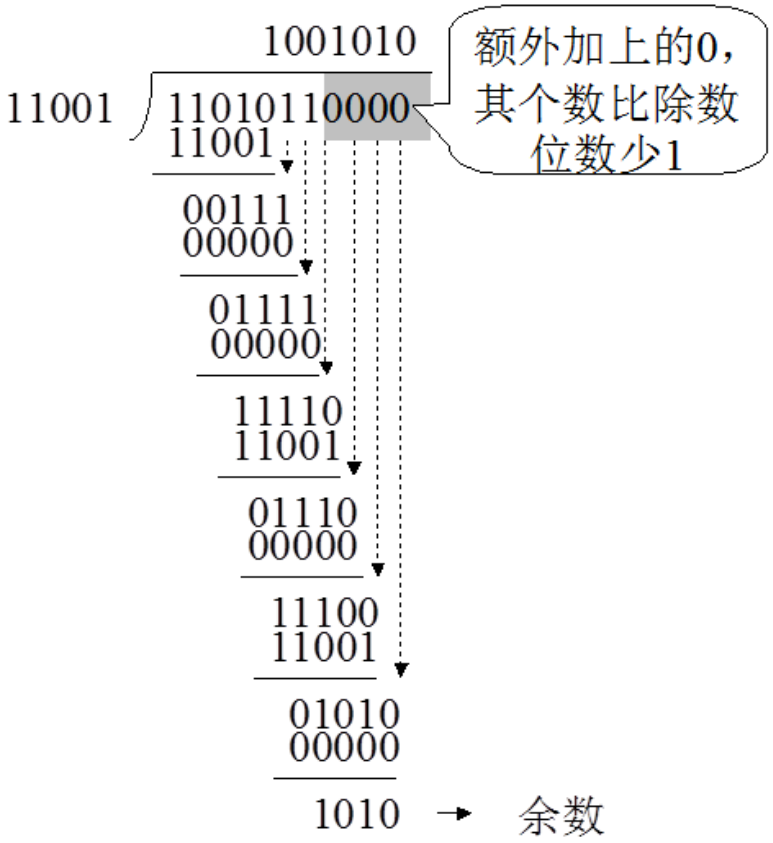
例如，约定的生成多项式为 $G(x)=x^4+x^3+1$，其最高阶次为 4。要发送的信息单元为 U=1101011，其对应的多项式 $U(x)=x^6+x^5+x^3+x+1$，$U(x)\bullet x^4=x^{10}+x^9+x^7+x^5+x^4$。U(x)•x4/G(x)所得的余式为 R(x)。

计算过程（二选一）：

过程一：

$$
\begin{array}{r|l}
 & x^6 \quad +x^3 \quad +x \\
\hline
x^4+x^3+1 & x^{10}+x^9 \quad +x^7 \quad +x^5+x^4 \\
 & x^{10}+x^9 \quad +x^6 \\
\hline
 & x^7+x^6+x^5+x^4 \\
 & x^7+x^6 \quad +x^3 \\
\hline
 & x^5+x^4+x^3 \\
 & x^5+x^4 \quad +x \\
\hline
 & x^3 \quad +x \quad \rightarrow \text{余式}
\end{array}
$$

过程二：



R(x)=x^3+x，R(x)所对应的 4 位二进制码 R=1010。将 R 放在 U 之后就构成了循环冗余检验码 1101011*1010*。

3. 接收方的检错方式

接收方将收到的数据块除以生成多项式所对应的二进制码 G，如果所得到的余数为 0，则是正确的；如果所得到的余数不为 0，则是错误的。当然，这里的除法也必须采用模 2 法则，收发两端采用相同的生成多项式。

网桥路由算法 (P96)

三种路由策略：

① 固定路由策略

② 路由学习策略

③ 源路由策略

三种网桥：

① 固定路由网桥

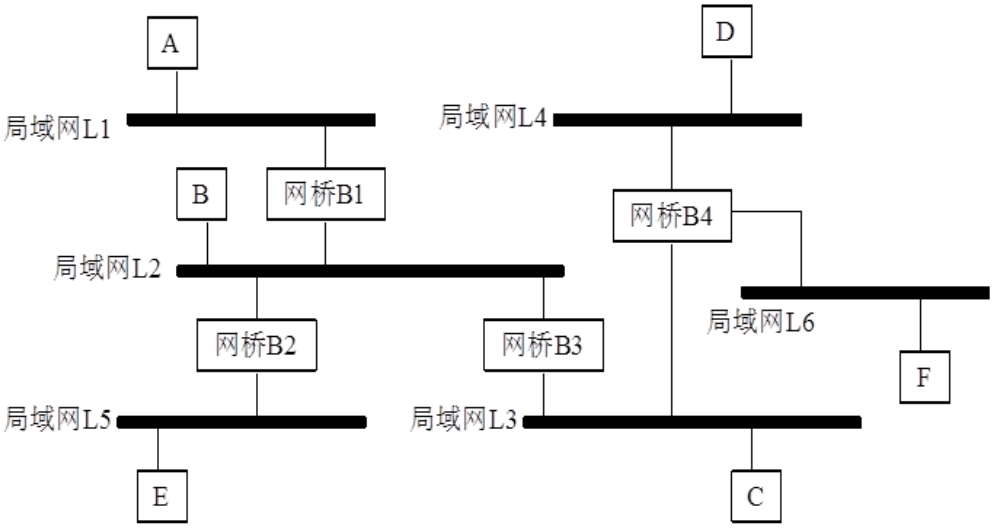
② 透明网桥

③ 源路由网桥

1. 固定路由网桥

每个网桥中都有一张表，这张表中记录了到某个特定站点的帧应该转发到那个哪个局域网中去的信息，这个表称为路由表。

在固定路由网桥中，路由表的生成是由手工配置的，一旦配置完成，路由表不会变动。



源局域网L1		源局域网L2	
目标	下一局域网	目标	下一局域网
A	--	A	L1
B	L2	B	--
C	L2	C	--
D	L2	D	--
E	L2	E	--
F	L2	F	--

(a)网桥B1

源局域网L2		源局域网L5	
目标	下一局域网	目标	下一局域网
A	--	A	L2
B	--	B	L2
C	--	C	L2
D	--	D	L2
E	L5	E	--
F	--	F	L5

(b)网桥B2

源局域网L2		源局域网L3	
目标	下一局域网	目标	下一局域网
A	--	A	L2
B	--	B	L2
C	L3	C	--
D	L3	D	--
E	--	E	L2
F	L3	F	--

(c)网桥B3

源局域网L3		源局域网L4		源局域网L6	
目标	下一局域网	目标	下一局域网	目标	下一局域网
A	--	A	L3	A	L3
B	--	B	L3	B	L3
C	--	C	L3	C	L3
D	L4	D	--	D	L4
E	--	E	L3	E	L3
F	L6	F	L6	F	--

(d)网桥B4

2. 透明网桥

能够根据网络信息自动生成和修改它们自己的路由表的网桥称为透明网桥（Transparent Bridge）。

这种自动修改和生成路由表的能力称为路由学习（Route Learning）或地址学习（Address Learning）。

1) 路由表的自动修改

当网桥接收到一个帧时，检查帧的源地址。确定发送该帧的站点可以通过这个帧刚到达的局域网来访问。

之后修改路由表中关于这个站点的信息。

2) 路由表的初始化

当一个网桥修改一个路由表的某一项时，它包含了对该项修改的时间。每个网桥同时维持一个定时器，只要定时器到期，网桥就检查路由表的每一项，若从定时器被设置以来，该项一直未被修改，网桥就删除该项。

当网桥收到一个发往某站点的帧，而在路由表中没有该站点的路由信息时，网桥使用一个扩散算法。

扩散算法有两个目的：

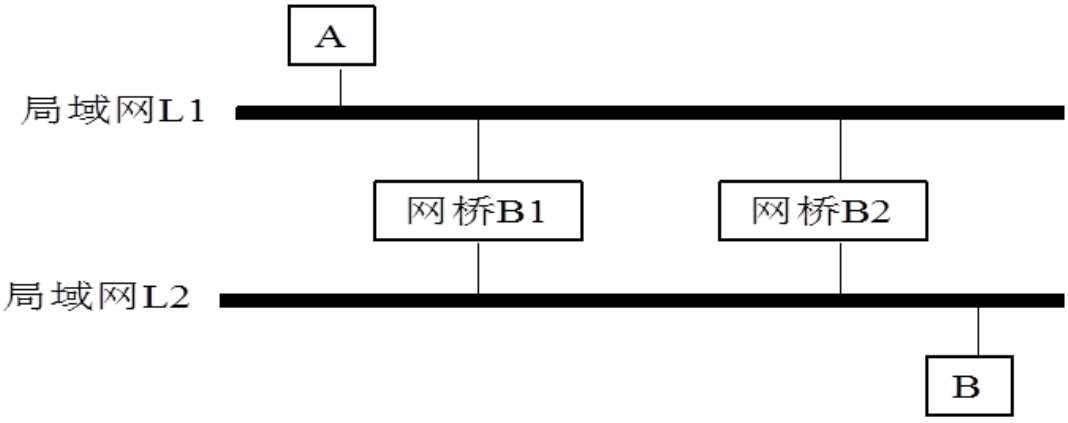
保证该帧到达目的地

让个多的网桥看到这一帧，保证路由信息最新

3) 帧循环问题

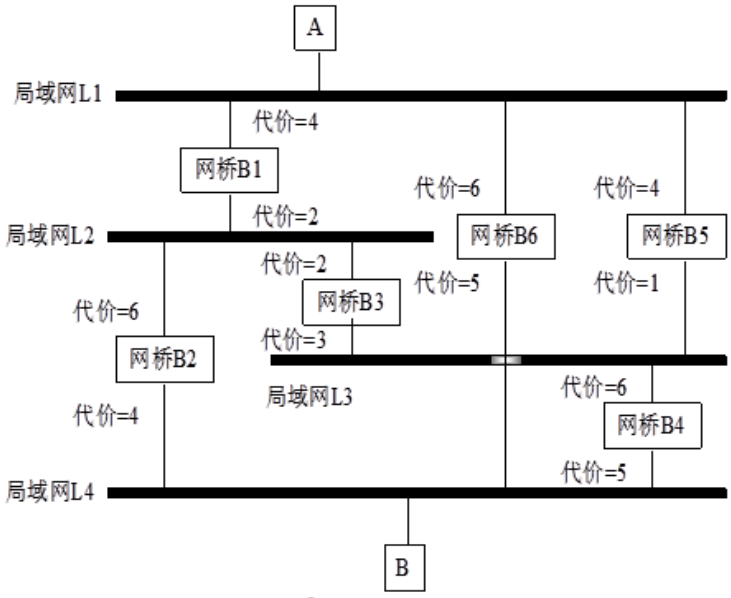
当一个互连局域网有回路时，就可能会产生帧的循环传递问题。

这种过程继续下去，将导致帧的爆炸，最终会阻塞整个系统，使通信停止。

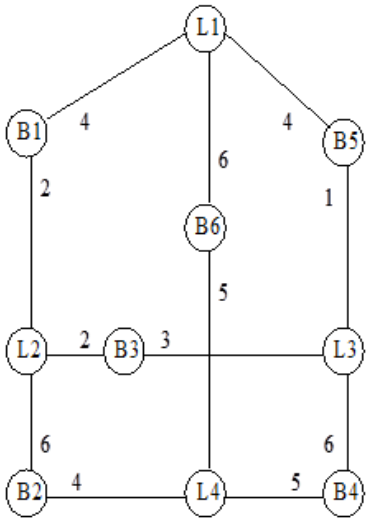


4) 生成树算法（Spanning Tree）

- 对于带有回路的互连局域网，必须停用某些网桥来淘汰循环。
- 不允许某些网桥转发帧，把它们当作别的网桥失效时的备份。
- 网桥执行一种称为生成树的算法来完成这项工作。
- 确定网桥端口费用：每个网桥到局域网的连接（即网桥端口—Bridge Port）有一定的费用。



a



b

- 算法规则

① 先确定根网桥

② 确定每个网桥的根端口（Root Port）

③ 为每个局域网决定一个指定网桥

④ 确定路径，确定生成树

- 步骤一：选择一个网桥作为根网桥：

根网桥是具有最低地址（或标识 ID）的那个网桥，根网桥是生成树的根节点。

根网桥的选择是通过发送网桥协议数据单元（BPDU）这样的特殊帧来完成的。每个网桥协议数据单元包含一个网桥的 ID，帧首次被发送时的端口 ID 和接收该帧的端口的累计费用

- 步骤二：每个网桥确定它的根端口（Root Port）。

这个端口对应于到根网桥的最小费用路径。

随后，每个网桥会使用它的根端口来和根网桥通信

网桥	根端口	费用
B2	B2->L2	6
B3	B3->L2	2
B4	B4->L3	8
B5	B5->L3	3
B6	B6->L1	6

- 步骤三：为每个局域网决定一个指定网桥

  来自这个局域网的帧通过哪一个网桥转发到根网桥的费用最低。

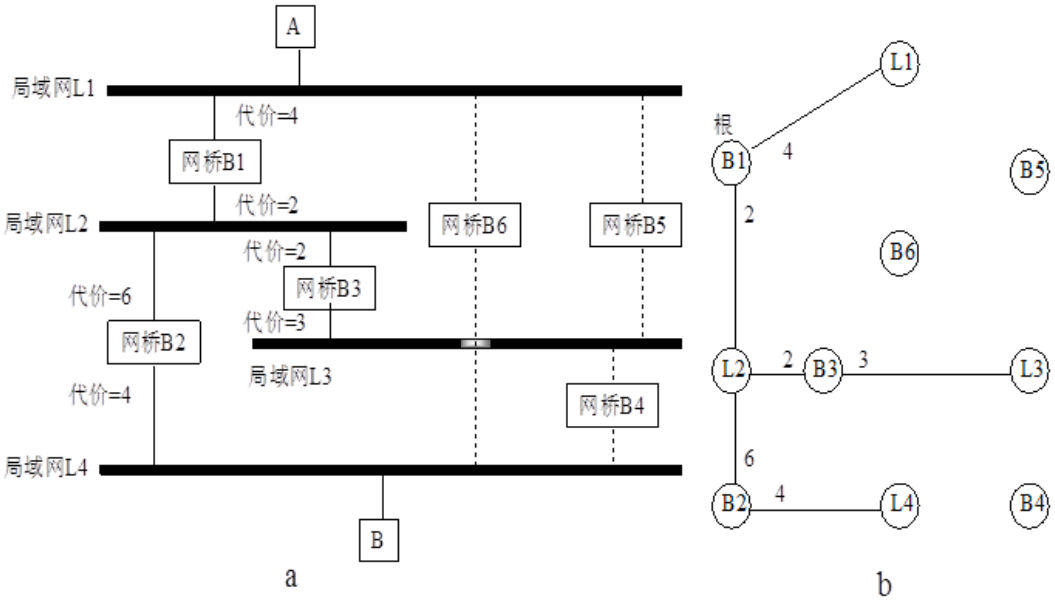
  L1 -> B1

  L2 -> B1

  L3 -> B3

  L4 -> B2

- 步骤四：将局域网和它指定网桥连接起来，同时将指定网桥的根端口和对应的局域网连接起来，就组成了一个生成树。



3. 源路由网桥

链路状态路由算法 (P117)

在链路状态路由中，每个路由器和互连网络中的所有其它路由器共享关于它邻居的信息。

（1）共享关于邻居的信息。路由器仅仅发送关于自己邻居的信息，而不是像距离向量路由那样发送整个路由表。

（2）共享的信息发给所有的路由器。

（3）信息的共享在有规律的时间间隔内进行。链路状态路由中的时间间隔通常是30 分钟。

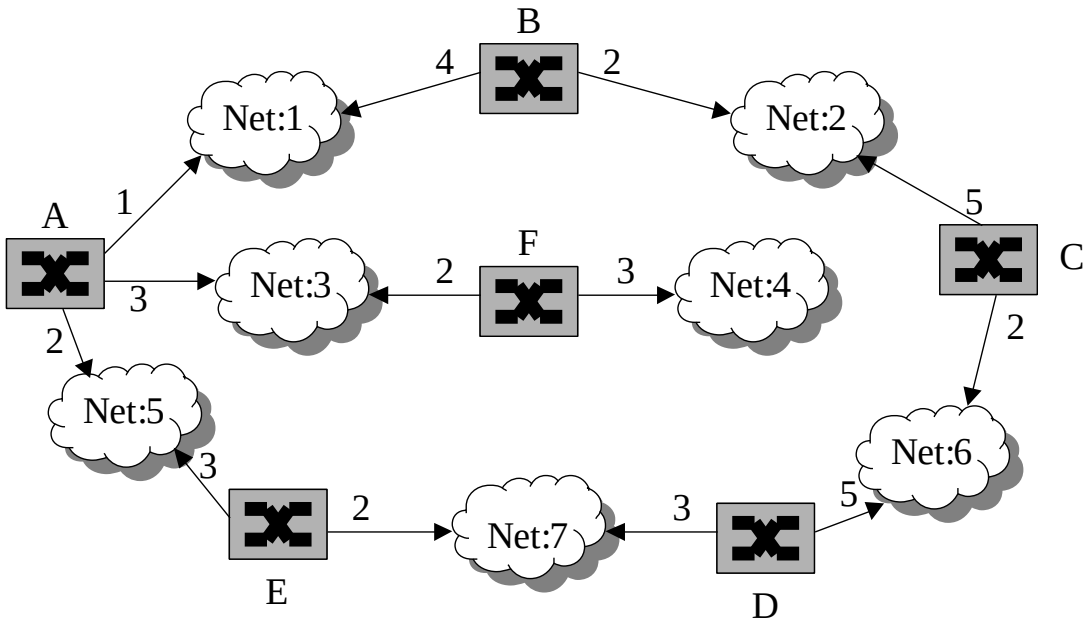
链路状态路由可分为两步完成：

第一步是共享链路状态信息，即每个路由器将它自己和它的所有邻居之间的链路状态信息发送给互连网络中的所有其它路由器。

第二步是每个路由器根据自己所掌握的关于整个网络的链路状态信息计算出到每个网络的路由，建立路由表。

1. 链路状态信息共享

1) 路由器传输包的费用



2) 链路状态包

路由器通过向整个互连网络中的所有路由器发送链路状态包(LSP)，在网络中扩散关于自己邻居的信息。

一个 LSP 通常包含四个信息域：广告者的 ID，所影响的目标网络 ID，费用，邻居路由器的 ID。

3) 获得关于邻居路由器的信息

每个路由器都周期性地发送一个简短的问候包来获取关于它们邻居的信息。

如果邻居按照它们所期待的那样对问候做出响应，它的这个邻居就被认为是活跃的且起作用的

果没有响应，那么它的这个邻居被认为发生了某种变化，发送问候包的路由器将通过它的下一个 LSP 包向网络的其它部分发出警告。

4) 初始化

每个路由器在启动时向它的所有邻居发送一个问候包来获取每条链路的状态信息。然后它基于这些问候的结果准备一个 LSP，并将它扩散到整个网络。

5) 链路状态数据库

每个路由器接收每个其它路由器发送来的 LSP，并将它们的信息存放到一个链路状态数据库中。由于每个路由器接收相同的 LSP，每个路由器将创建相同的链路状态数 据库。如果一个路由器被删除或被加入到网络中，所有的链路状态数据库将被更新。

广告者	相关网络	费用	邻居
A	1	1	B
A	3	3	F
A	5	2	E
B	1	4	A
B	2	2	C
C	2	5	B
C	6	2	D
D	6	5	C
D	7	3	E
E	7	2	D
E	5	3	A
F	3	2	A
F	4	3	-

链路状态数据库

2. Dijkstra 算法计算路由表

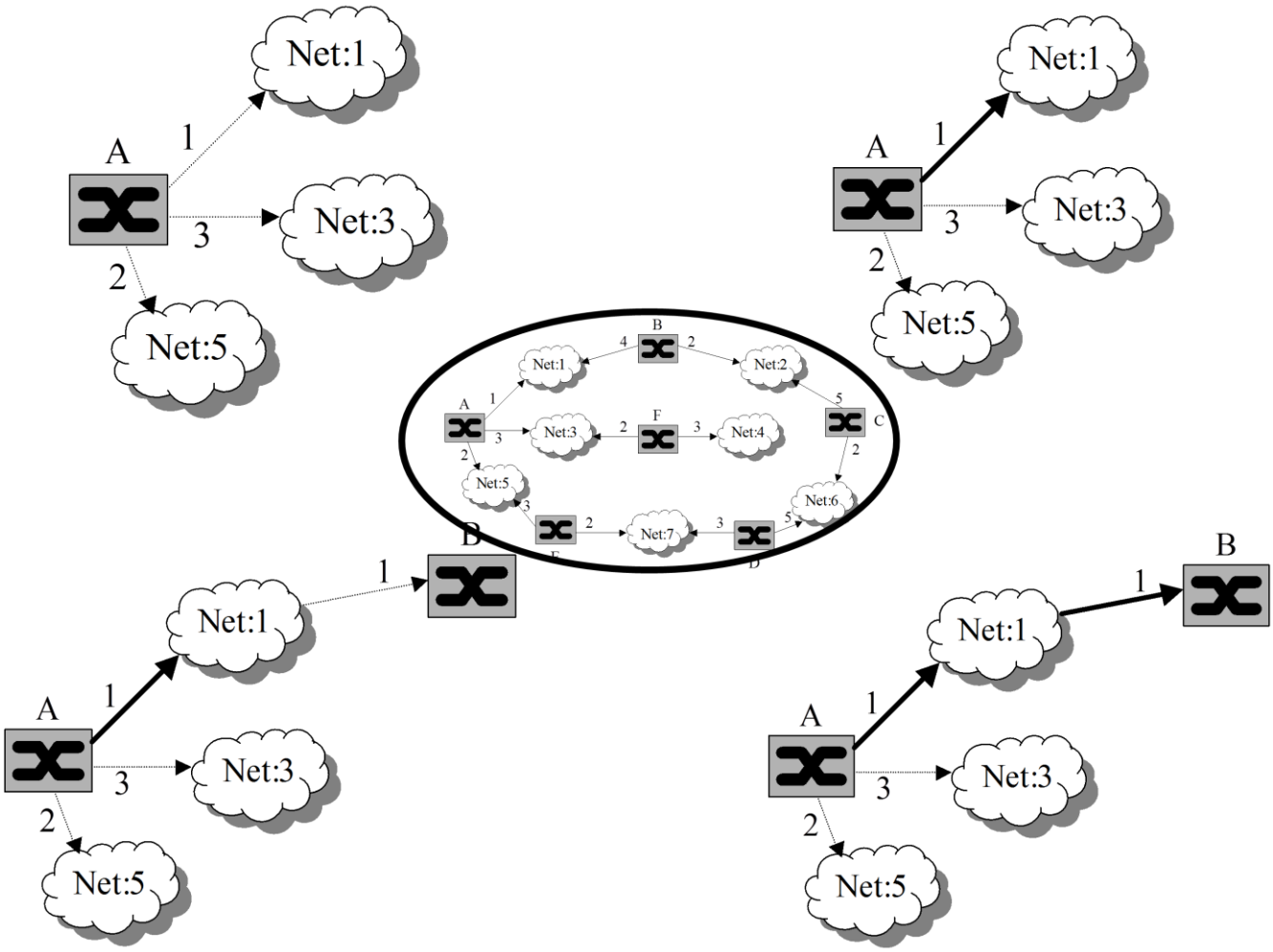
- 每个路由器将 Dijkstra 算法应用到自己的链路状态数据库上 ，为自己创建一个最短路径树，计算出路由表。
- 节点有两种类型：网络和路由器。
- 弧有两类：路由器到网络的链路和网络到路由器的链路。
- 在 Dijkstra 算法中，从路由器到网络的链路的费用才有效，而从网络到路由器的链路的费用总是为 0。
- 每个路由器在使用 Dijkstra 算法时，根据下面四个步骤来形成自己的最短路径树：

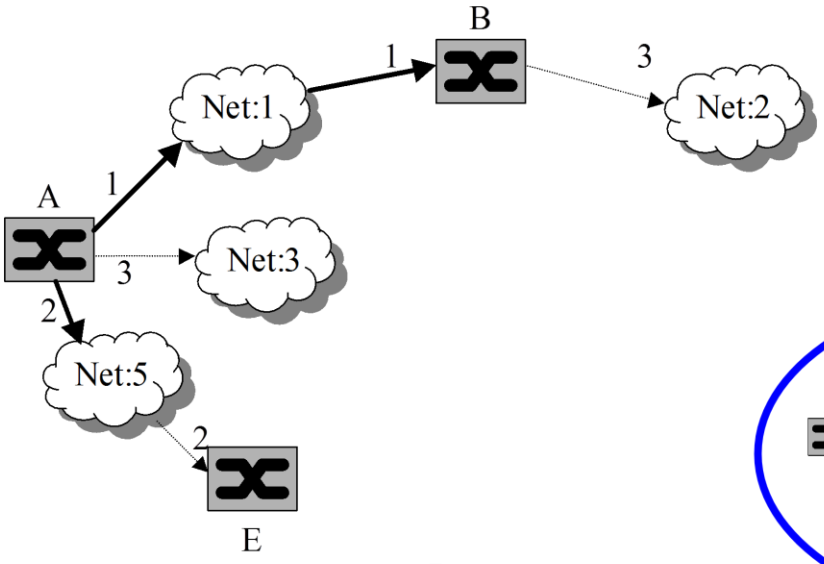
（1）选择自己作为树的根，并将根标记为永久性节点。算法接着从根出发连接它的所有邻居节点。这种连接是临时性的。

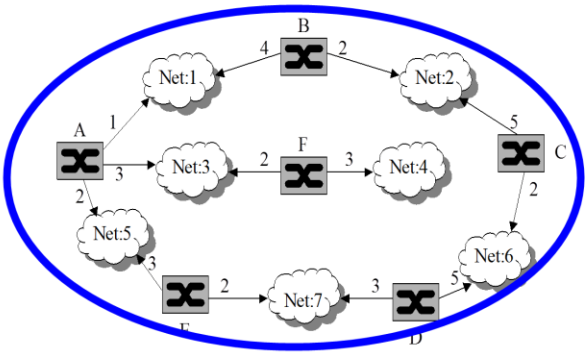
（2）算法比较所有的临时连接，找出费用最小的路径，这个路径上的所有弧和节点被标记为最短路径树上的永久部分。

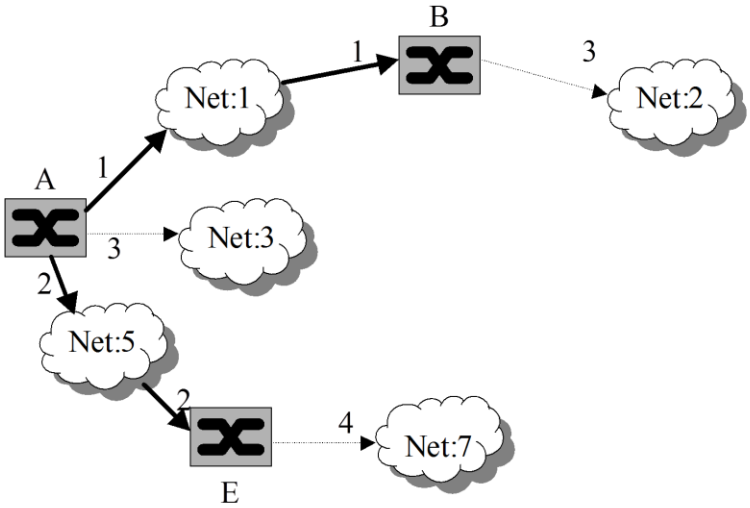
（3）算法考察链路状态数据库，找出从这个选定的最短路径向外延伸所能连接的所有非永久性节点，将这些节点临时性的加到最短路径树上。

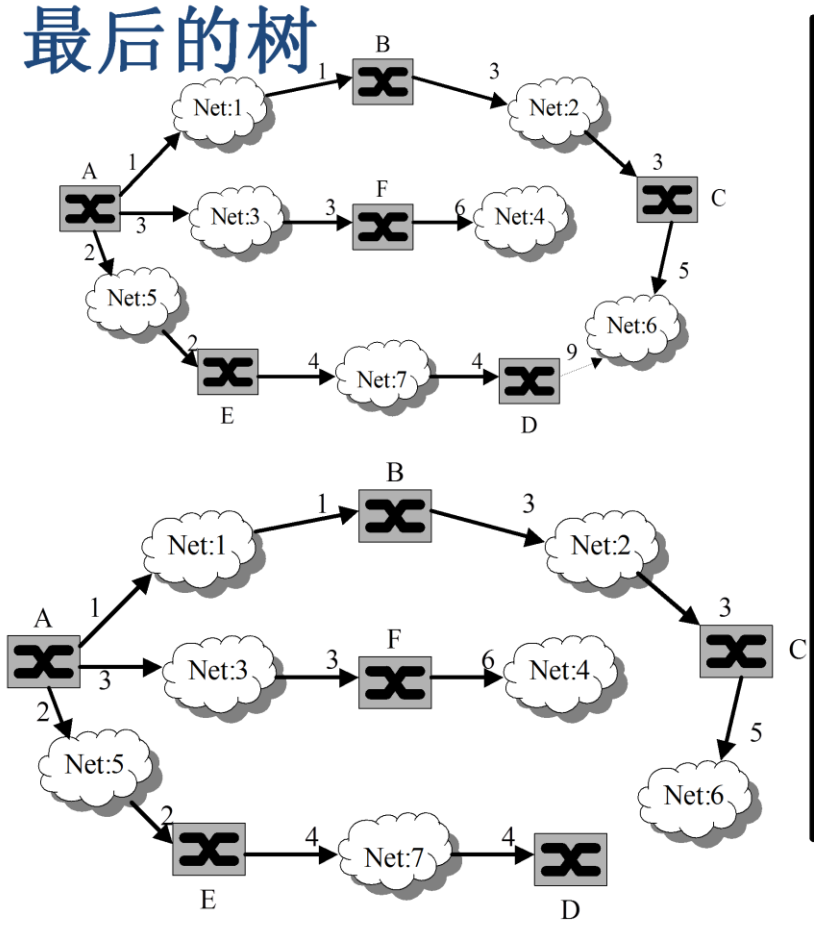
（4）如果所有的节点已经成为最短路径树上的永久部分，则算法结束，去掉非永久性的弧。否则，转步骤（2）继续执行。


Net:1
A
1
Net:3
3
2
Net:5
B
4
2
Net:2
5
C
F
Net:4
Net:6
Net:7


B
1
3
Net:1
Net:2
A
1
3
Net:3
2
Net:5
2
E


B
4
2
Net:1
Net:2
A
1
5
C
F
3
Net:3
2
3
Net:4
2
2
Net:5
3
Net:6
2
Net:7
3
5


B
1
3
Net:1
Net:2
A
1
3
Net:3
2
Net:5
2
4
Net:7
E



目标网络	费用	下一个路由器
1	1	-
2	3	B
3	3	-
4	6	F
5	2	-
6	5	B
7	4	E

路由器A的路由表

以太网访问模式 (P167)

在 LAN 中，多个用户在没有任何控制的情况下同时访问一条线路时，会存在由于不同信号叠加而相互破坏的情况,这就是冲突。为了使冲突发生的可能性最小，需要有一种机制来协调通信。

以太网的媒体访问控制机制称为带有冲突检测的载波侦听多路访问(CSMA/CD)

CSMA/CD 的发展：MA-〉CSMA-〉CSMA/CD

1. 多路访问(MA)

不提供通信管制，“不听就说”

2. 载波帧听多路访问(CSMA)

首先监听链路上是否已经存在通信。 “先听后说”。 由于存在传输延迟，还是会出现冲突

CSMA 可分为三种:

(1) 非坚持 CSMA

(2) 坚持 CSMA

(3) P-坚持 CSMA

非坚持 CSMA 的算法如下：

如果链路是空闲的，则可以发送。

如果链路是忙的，则等待一段时间。

坚持 CSMA 的算法如下：

如果链路是空闲的，则可以发送。

如果链路是忙的，则继续侦听，直到检测到链路空闲，立即发送。

如果有冲突则等待随机的时间。

P-坚持 CSMA 的算法如下：

如果链路是空闲的，则以 P 的概率发送，而以(1-P)的概率延迟一个时间单位。

如果链路是忙的，继续侦听直至链路空闲。

3. 带有冲突检测的载波侦听多路访问(CSMA/CD)

在传输的时候继续监听链路是否发生冲突。 “边听边说”

CSMA/CD 的算法描述：

(1)如果链路是空闲的，则可以发送并同时检测冲突。

(2)如果链路是忙的，则继续侦听，直到检测到链路空闲。

(3)如果在发送过程中检测到冲突，则停止当前帧的发送，发阻塞信号，等待一段选定的时间。

4. 退避算法：

(1)对每一个帧，当第一次发生冲突时，设置参数 L=2；

(2)退避间隔取 1 到 L 个时间片中的一个随机数。一个时间片等于链路上最大传输延迟的 2 倍。

(3)当帧重复发生一次冲突时，则将参数 L 加倍。L 的最大值为 1024。即当 L 增加到 1024 时，L 不再增加。

(4)帧的最大重传次数为 16，超过这个次数，则该帧不再重传，并报告出错。

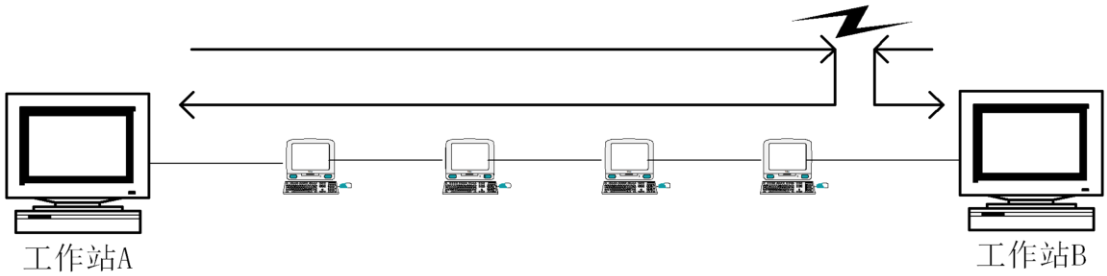
发送站等待的时间:

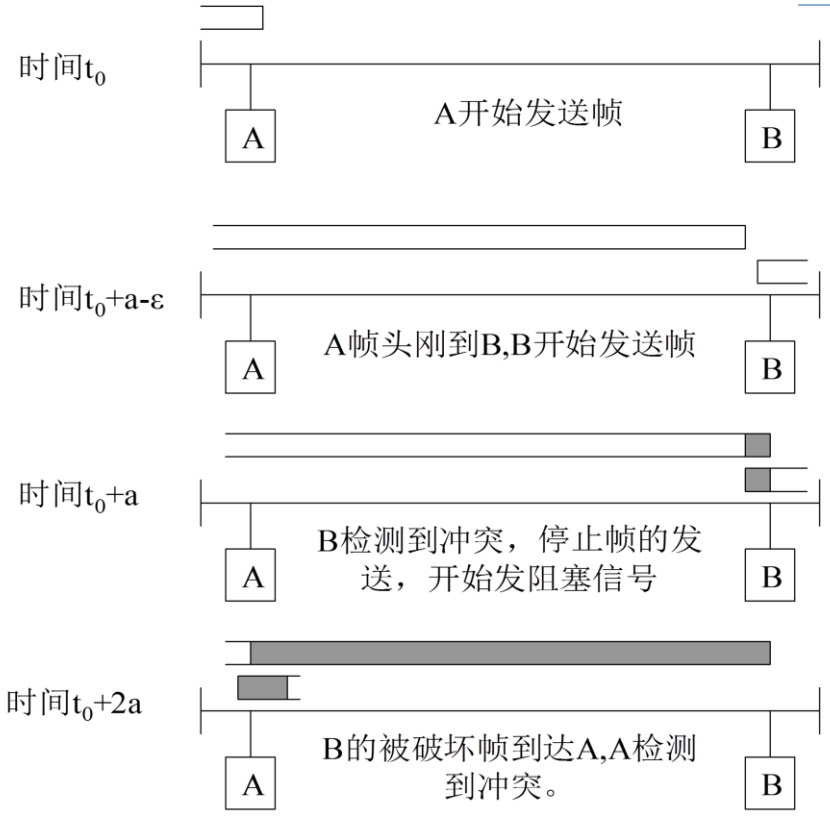
t=R×2T　　其中:

R 是 1～2 $^{min(k,10)}$之间的随机数

K 是冲突次数,最大为 16

冲突示意图





用于检测冲突的时间等于任意两个站点之间最大的传播延迟的两倍。

最短帧长的计算公式：

Lmin=2S×R/V

Lmin：最短数据帧长（bit）

S：任意两站点间的最大距离（m）

R：数据传输速率（Mbps）

V：电子传播速度（200m/us）

IPv4 (P210)

1. IPV4 协议

- IP 协议是 TCP/IP 协议族中的核心协议。

- 所有的 TCP、UDP、ICMP、IGMP 数据都是以 IP 数据报格式传输。
- IP 协议为高层提供不可靠、无连接的数据报通信。
- IP 协议提供的不可靠服务是指它不能保证 IP 数据报能成功地到达目的地。任何要求的可靠性必须由上层来提供。
- 无连接是指 IP 协议并不维护任何关于后续数据报的状态信息。每个数据报的处理是相互独立的，IP 数据报可以不按发送顺序接收。

2. IP 地址

- 网络中的每个独立主机的每个接口必须有一个唯一的 Internet 地址，也称为 IP 地址。
- IP 地址长度为 32 位。表示地址空间是 232，或 4294967296（超过 40 亿个）。
- IP 地址有三种常用的表示方法：

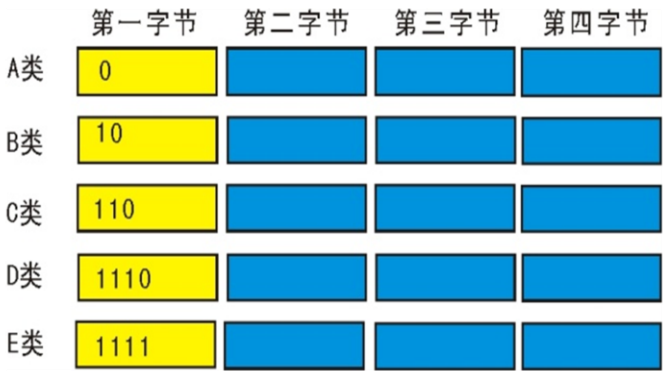
  二进制表示方法,如:10000001 00001110 00000110 00011111

  点分十进制表示方法,如:129.14.6.31

  十六进制表示方法,如：0X810E061F

- IP 地址的分类

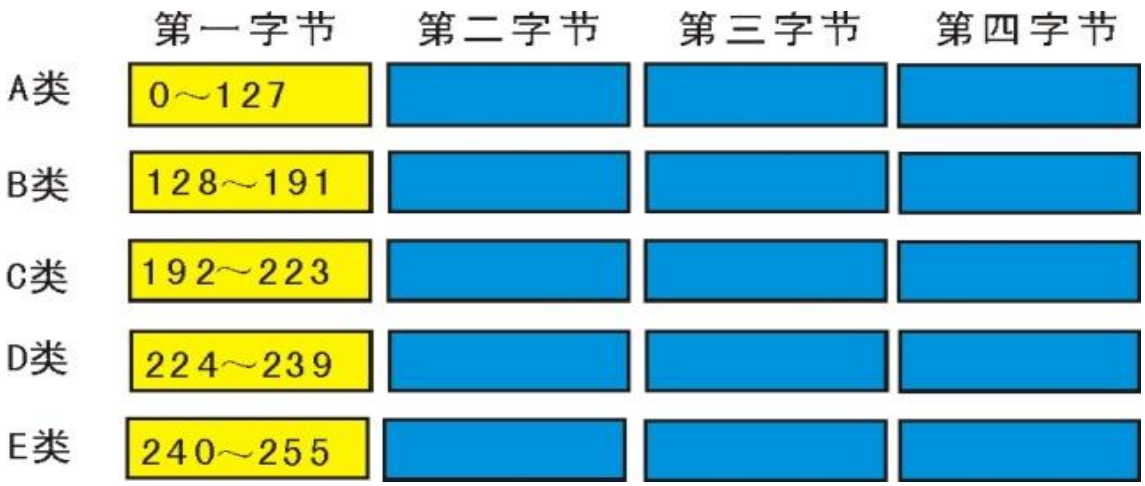
IP地址分成5类：A类，B类，C类，D类和E类。



其中**A**、**B**和**C**类址是基本的**Internet**地址，是用户使用的地址.

D类地址是用于多目标广播的广播地址（也称多播或组播地址）

E类地址为保留地址



类型	范围
A	0.0.0.0 到127.255.255.255
B	128.0.0.0 到191.255.255.255
C	192.0.0.0 到223.255.255.255
D	224.0.0.0 到239.255.255.255
E	240.0.0.0 到247.255.255.255

- 特殊的 IP 地址

  网络地址：主机地址为全“0”的 IP 地址

  直接广播地址：主机地址为全“1”的 IP 地址

  有限广播地址：32 位为全“1”的 IP 地址（255.255.255.255）

  主机本身地址：32 位全“0”的 IP 地址

  回环地址：127.0.0.1 称为回环地址

- 专用 IP 地址

  为解决 IP 地址资源不足,定义两类 IP 地址：

  全局 IP 地址：用于 Internet 上的公共主机

  专用 IP 地址：仅用于专用网内部的本地主机

公共主机和本地主机可以共存于同一网络和进行互访，而大多数路由器不转发携带专用 IP 地址的分组。本地主机必须经网络地址转换服务器（NAT 或代理服务器）才能访问 Internet。

专用 IP 地址为：

10.0.0.0-10.255.255.255　　1 个 A 类网络

172.16.0.0-172.31.255.255　　16 个连续的 B 类网络

192.168.0.0-192.168.255.255　　256 个连续的 C 类网络

企业内部网主机的 IP 地址可以设置成专用 IP 地址，进行企业内部的网络应用；并可通过代理服务器(NAT)访问 Internet。

这样只需要申请少量的全局 IP 地址，既解决了 IP 地址不足的问题，又解决了网络安全问题。

- IP 地址按通信方式分类

单播地址（目的地址为单个主机）、广播地址（目的地址为指定网络上的所有主机）、多播地址（目的地址为同一组中的所有主机）。

3. IPv4 报文格式

（1）版本号 （2）报头长度 （3）服务类型 （4）总长度 （5）标识字段 （6）标志字段 （7）分段偏移 （8）生存时间 （9）协议字段 （10）报头校验和字段 （11）源 IP 地址和目的 IP 地址 （12）任选项

4. IP 封装

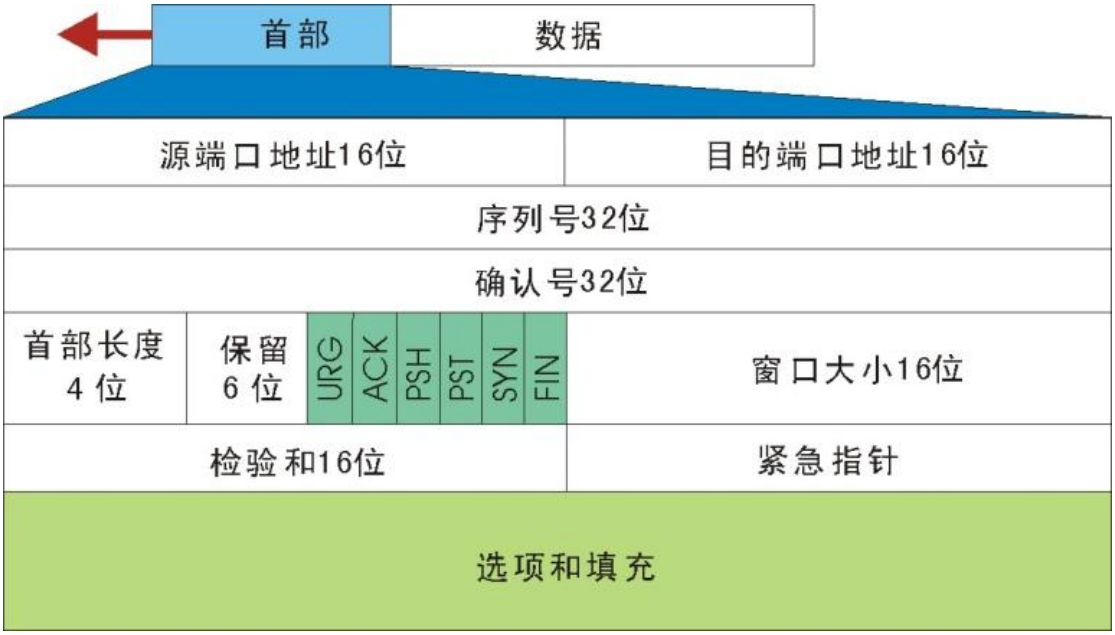
TCP (P229)

- TCP 提供了一种可靠的面向连接的字节流传输层服务；
- 它发送数据后启动一个定时器；通信的另一端对收到的数据进行确认，对乱序的数据重新排序，丢弃重复数据；
- TCP 提供端到端的流量控制，并计算和验证一个强制性的端到端检查和。

1. TCP 封装

帧首部	IP 首部	TCP 报文段

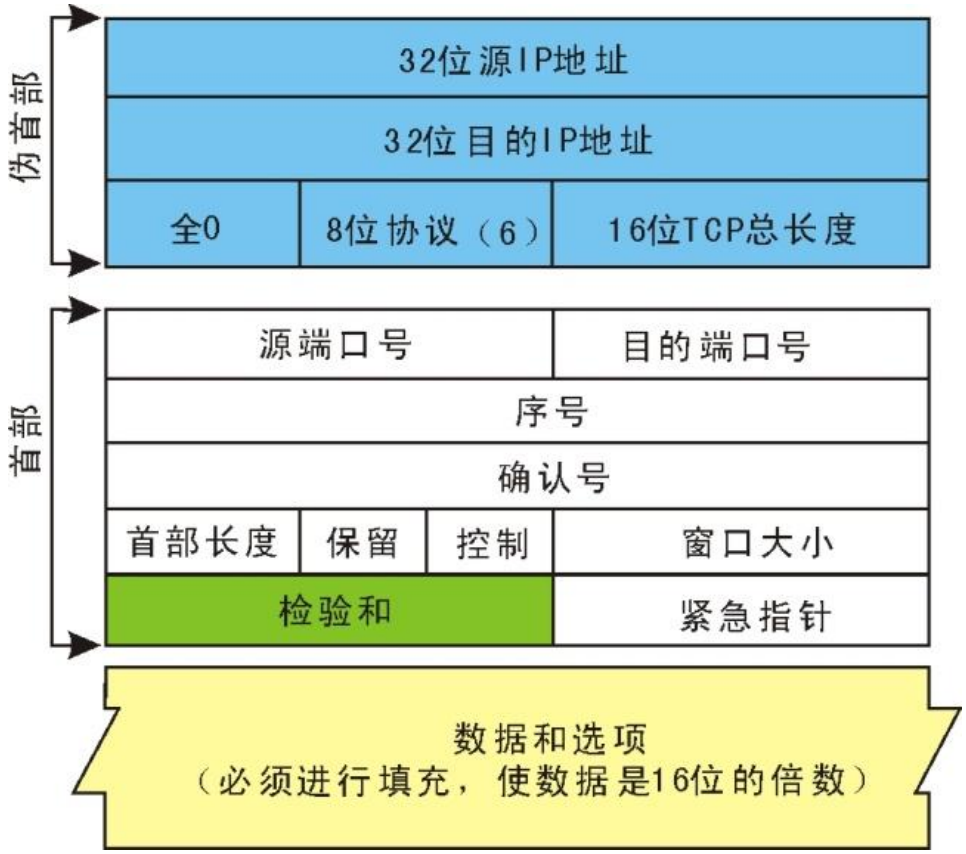
2. TCP 报文格式



3. TCP 标志位

URG：紧急指针有效
ACK：确认是有效的
PSH：请求推送
PST：连接复位
SYN：同步序号
FIN：终止连接

4. TCP 校验和



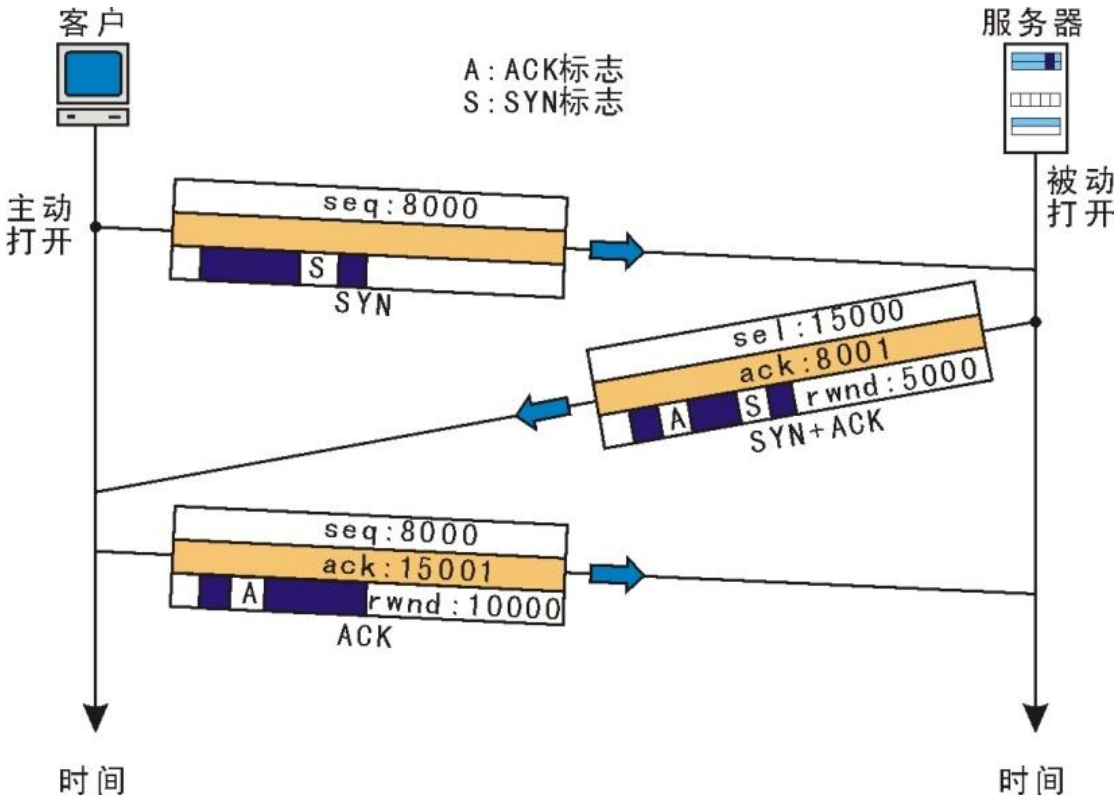
伪首部加到 TCP 报文段上

5. TCP 连接建立与释放

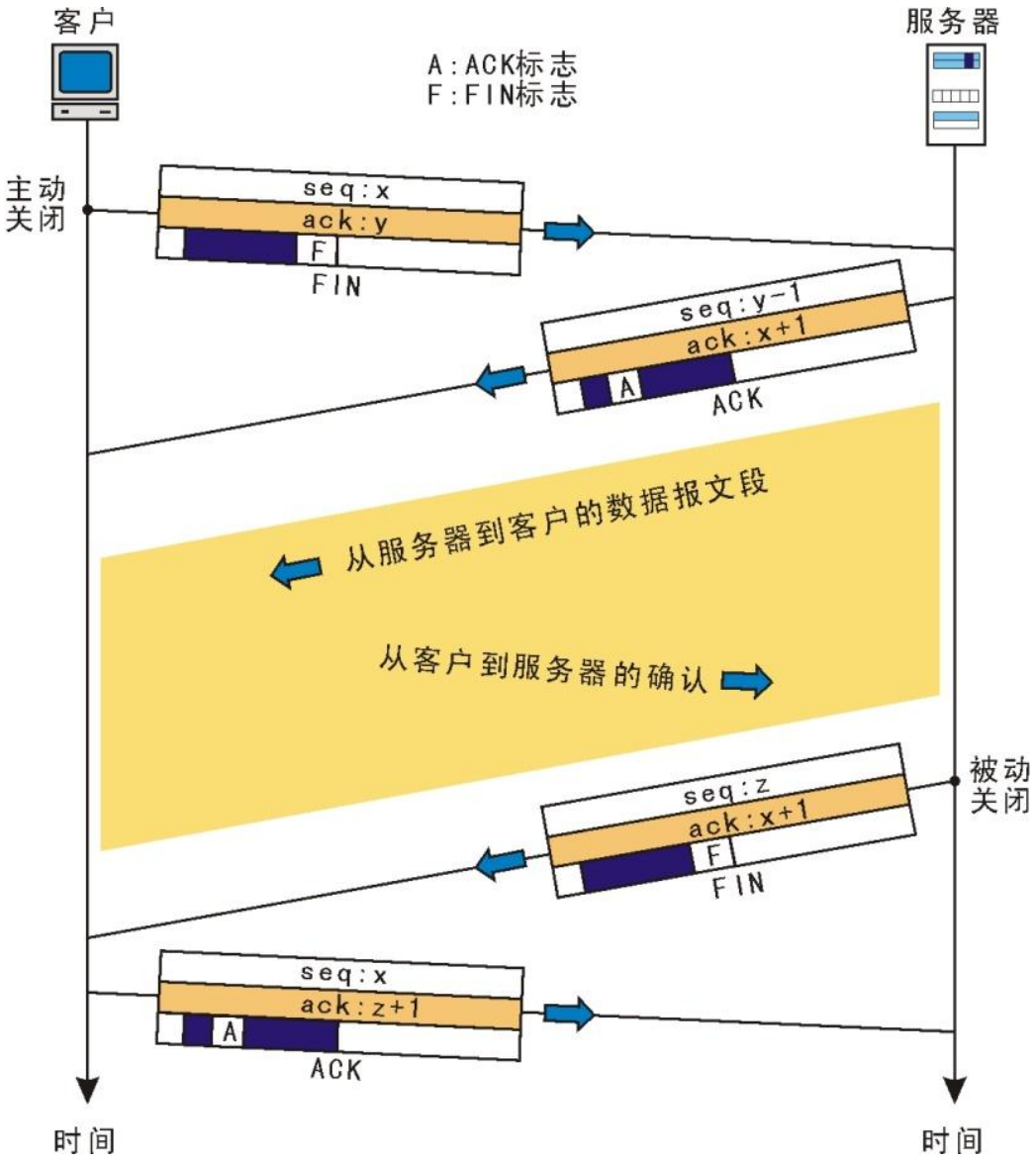
TCP 是一个面向连接的协议，通信双方在发送数据之前都必须建立一个 TCP 连接。在数据发送完毕之后断开连接。

TCP 使用三次握手建立连接、四次握手断开连接 。

三次握手建立连接


客户
服务器
A：ACK标志
S：SYN标志
主动打开
被动打开
seq:8000
S
SYN
sel:15000
ack:8001
A
S
rwnd:5000
SYN+ACK
seq:8000
ack:15001
A
rwnd:10000
ACK
时间
时间

四次握手断开连接



6. TCP 的应用

应用程序	TCP 端口
超文本传输协议（HTTP）	80
文件传输协议（FTP）	21，20
Telnet	23
NetBIOS 会话	139

传输层端口 (P241)

1. 端口号

RFC1700 含有一张由因特网已分配数值权威机构（IANA）定义的端口列表。端口号分成三段：

众所周知的端口为 0—1023。这些端口由 IANA 分配和控制。可能时，相同端口号分配给 TCP 和 UDP 的同一给定服务。

注册的端口（registered ports）为 1024—49151。这些端口不受 IANA 控制。但由 IANA 登记并提供它们的使用情况和清单。可能时，相同端口号分配给 TCP 和 UDP 的同一给定服务。

49152—65535 是动态的（dynamic）或私有的（private）端口。IANA 不管理这些端口，它们就是我们所说的临时端口。

2. 套接字

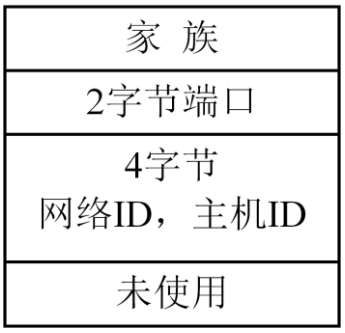
- 网络 socket 的结构如下：

  Code:

```
#include <netinet/in.h>
struct sockaddr_in {
  sa_family_t     sin_family;   /*internet address family*/
  in_port_t                sin_port;      /*port number*/
  struct in_addr  sin_addr;     /*holds the IP address*/
  unsigned char   sin_zero[8];  /*filling*/
};
```

View:

Struct sockaddr_in



- 句柄的建立是通过 socket 系统调用完成的，具体如下：

#include <sys/socket.h>

int socket(int domain, int type, int protocol);

Parameters:

domain 参数告诉系统这个 socket 用于何处。例如，值 AF_INET 指网络上的 socket。当通信进程运行于同一台机器时，则使用另一个 domain 参数 AF_UNIX。

Type 参数指定了将建立的端口是用于面向连接模型还是无连接模型的。参数值 SOCK_STREAM 为面向连接模型，SOCK_DGRAM 为面向无连接类型。

protocol，指定端口所使用的协议。该值一般被设置为 0，这种情况下 SOCK_STREAM 将使用 TCP 协议，而 SOCK_DGRAM 将使用 UDP 协议。

面向连接的编程模型 (P245)

1. 面向连接编程模型使用的主要系统调用

1．s = socket（domain，type，protocol）；

建立给定类型的 TSAP。

2．bind（s，localaddr，addrlen）；

为前面创建的 socket 分配一个本地地址。

3．connect（s，server，serverlen）；

发起与远端 socket 的一个连接。

4．listen（s，n）：

建立一个队列以存储外来的连接请求。

5．accept（s，from，fromlen）；

将一个连接请求从队列中移出或等待一个连接请求。

6．send（s，buf，buflen，flags）；

在给定的 socket 上发送一个报文（有连接）。

7．recv(s，buf，buflen，flags)；

在给定的 socket 上接收一个报文(有连接)。

8．close(s)；

拆除一个 socket 上的连接。

9．shutdown(s，how)；

终止一个 socket 上的连接。

10．sendto(s，buf，buflen，flags，to，tolen);

在无连接的 socket 上发送一个报文。

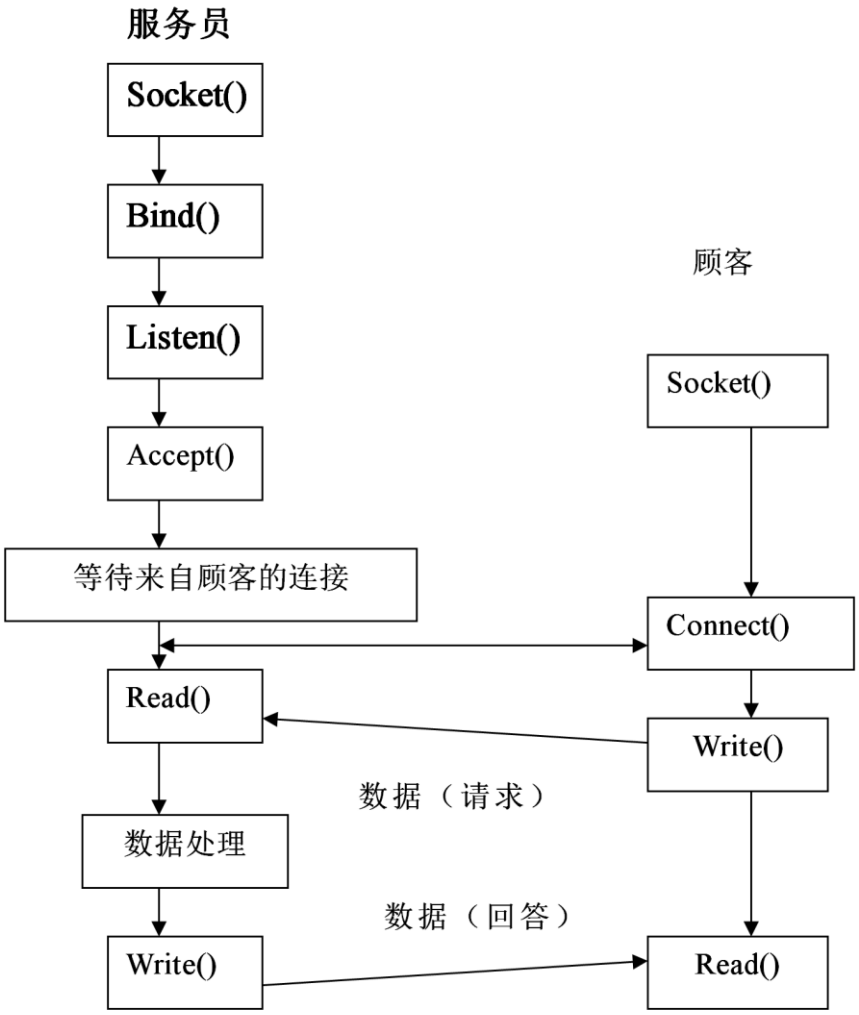
11．recvfrom(s，buf,buflen，flags，from，fromlen);

在无连接的 socket 上接收一个报文。

12．select(nfds，readfds，writefds，exceptfds，timeout)；

检查一组 socket，看它们是否可读或可写。

2. 面向连接服务流程



3. 面向连接服务编程实例

- 服务端程序 server using TCP protocol.

```
#include"inet.h"
    main(argc , argv)
    int argc ;
char *argv[] ;
{
```

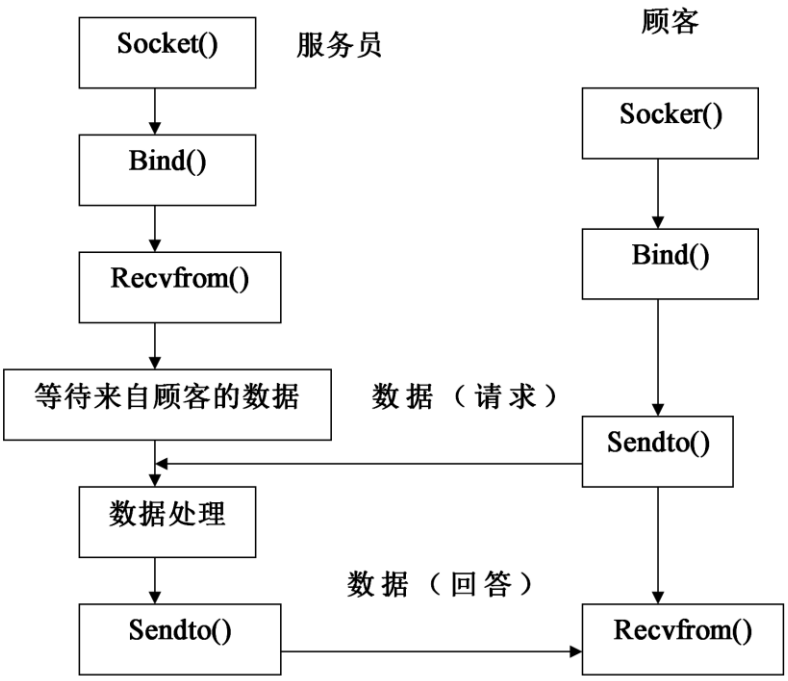
```
        Int sockfd , newsockfd , cliden , Childpid ;
        Struct sockaddf_in   cli_addr , serv_addr ;
        pname=argv[0] ;
        if((sockfd = socket(AF_INET , SOCK_STREAM , 0))<0)
                err_dump("server : can't open stream socker") ;
        /* Bind our local address so that the client can send to us. */
        bzero((Char *)&serv_addr , sizeof(serv_addr)) ;
        serv_addr.sin_family = AF_INET ;
        serv_addr.sin_addr.s_addr=htonl(INADDR_ANY) ;
        serv_addr.sin_port=htons(SERV_TCP_PORT) ;
        if(bind(sockfd , (struct sockaddr *)&serv_addr , sizeof(serv_addr))<0)
                err_dump("server : can't bind local address") ;
        listen(sockfd , 5) ;
        for( ; ; ){    ·
                /* Wait for a connection from a client process ,
                * This is an example of a concurrent server . */
                clilen = sizeof(cli_addr) ;
                newsockfd = accept(sockfd , (struct sockaddr *)&cli_addr , &clilen) ;
                if(newsockfd<0)
                        err_dump("server : accept error") ;
                if((childpid = fork())<0)
                        err_dump("server : fork error") ;
                else if(childpid = = 0){    /*child process*/
                close(sockfd) ;   /*close original socket*/
                str_echo(newsockfd) ; /*process the request*/
                exit(0) ;
                close(newsockfd) ;   /*parent process*/
                }
        }
}
```

- 客户端程序 client using TCP protocol

```
#include"inet.h"
main(argc , argv)
int argc ;
char *argv[] ;
  {
        int  sockfd ;
        struct sockaddr_in  serv_addr ;
        pname=argv[0] ;
        /*Fill in the structure"serv_addr"with the address of the
        * server that we want to connect with . */
        bzero((char *)&serv_addr , sizeof(serv_addr)) ;
        serv_addr.sin_family=AF_INET ;
        serv_addr.sin.addr.s_addr = inet_addr(SERV_HOST_ADDR) ;
        serv_addr.sin_port = htons(SERV_TCP_PORT) ;
        /*Open a TCP socket(an Internet stream socket) . */
        if((sockfd = socket(AF_INET , SOCK_STREAM , 0)<0)
                err_sys("client : can't open stream socket") ;     '
        /*Connect to the server . */
        if(connect(sockfd , (struct sockaddr *)&serv_addr , sizeof(serv_addr))<0)
                err_sys("client : can't connect to server") ;
        str_cli(Stdin , sockfd) ;   /*do it all*/
        close(sockfd) ;
        exit(0) ;
  }
```

面向无连接的编程模型 (P251)

1. 面向无连接服务流程



2. 面向无连接服务编程实例

- 无连接服务服务端程序

```
#define LOCAL_SERVER_PORT 1500
#define MAX_MSG 100
int main(int argc, char *argv[])
{
        int sd, rc, n, cliLen;
        struct sockaddr_in cliAddr, servAddr;
        char msg[MAX_MSG];
```

```
        /* socket creation */
        sd=socket(AF_INET, SOCK_DGRAM, 0);
        if(sd<0) {
                printf("%s: cannot open socket \n",argv[0]);
        exit(1);
        }
        /* bind local server port */
        servAddr.sin_family = AF_INET;
        servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
        servAddr.sin_port = htons(LOCAL_SERVER_PORT);
        rc = bind (sd, (struct sockaddr *) &servAddr,sizeof(servAddr));
        if(rc<0) {
        printf("%s: cannot bind port number %d \n", argv[0], LOCAL_SERVER_PORT);
                exit(1);
        }
        printf("%s: waiting for data on port UDP %u\n", argv[0],LOCAL_SERVER_PORT);
        /* server infinite loop */
        while(1) {
                /* init buffer */
                memset(msg,0x0,MAX_MSG);
                /* receive message */
        cliLen = sizeof(cliAddr);
        n = recvfrom(sd, msg, MAX_MSG, 0, (struct sockaddr *) &cliAddr, &cliLen);
        if(n<0) {
                        printf("%s: cannot receive data \n",argv[0]);
                continue;
                }
        /* print received message */
                printf("%s: from %s:UDP%u : %s \n",
                argv[0],inet_ntoa(cliAddr.sin_addr),
                        ntohs(cliAddr.sin_port),msg);
        }/* end of server infinite loop */
        return 0;
}
```

- 无连接服务顾客端程序

```
int main(int argc, char *argv[]) {
        int sd, rc, i;
        struct sockaddr_in cliAddr, remoteServAddr;
        struct hostent *h;
        /* check command line args */
        if(argc<3) {
                printf("usage : %s <server> <data1> ... <dataN> \n", argv[0]);
        exit(1);
        }
        /* get server IP address (no check if input is IP address or DNS name */
        h = gethostbyname(argv[1]);
        if(h==NULL) {
                printf("%s: unknown host '%s' \n", argv[0], argv[1]);
        exit(1);
        }
        printf("%s: sending data to '%s' (IP : %s) \n", argv[0], h->h_name,inet_ntoa(*(struct in_addr *)h-
>h_addr_list[0]));
        remoteServAddr.sin_family = h->h_addrtype;
        memcpy((char *) &remoteServAddr.sin_addr.s_addr,
                h->h_addr_list[0], h->h_length);
        remoteServAddr.sin_port = htons(REMOTE_SERVER_PORT);
        /* socket creation */
        sd = socket(AF_INET,SOCK_DGRAM,0);
        if(sd<0) {
                printf("%s: cannot open socket \n",argv[0]);
        exit(1);
        }
        /* bind any port */
        cliAddr.sin_family = AF_INET;
        cliAddr.sin_addr.s_addr = htonl(INADDR_ANY);
        cliAddr.sin_port = htons(0);
        rc = bind(sd, (struct sockaddr *) &cliAddr, sizeof(cliAddr));
        if(rc<0) {
                printf("%s: cannot bind port\n", argv[0]);
```

```
                exit(1);

        }

        for(i=2;i<argc;i++) {

                rc = sendto(sd, argv[i], strlen(argv[i])+1, 0, (struct sockaddr *) &remoteServAddr,
 sizeof(remoteServAddr));

                if(rc<0) {

                printf("%s: cannot send data %d \n",argv[0],i-1);

                        close(sd);

                exit(1);

                }

                }

        return 1;

}
```

3. 两种模型的区别

从程序设计的角度看，两种模型中，服务器进程都需要创建端口，并把自己的本地地址绑定到这个端口上。在面向连接模型中，服务器接下来必须监听进入的连接。在面向无连接模型的情况下这一步是没有必要的，因为客户端会做更多的工作。

从客户端的角度看，面向连接模型中客户端只需要连接到服务器就可以了；但是在面向无连接的模型中，客户端必须创建端口并把它的本地地址绑定到这个端口上。

不同的系统调用的目的都是传输数据。send 和 recv 系统调用可以用于两种模型中。然而，为了使服务器获得发送端的消息并作出适当的回复，通常在面向无连接模型中使用 sendto 和 recvfrom 代替 send 和 recv。