1. 二阶段提交的问题

a) 同步阻塞问题。执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。

b) 单点故障。由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。(如果是协调者挂掉，可以重新选举一个协调者，但是无法解决因为协调者宕机导致的参与者处于阻塞状态的问题)

c) 数据不一致。在二阶段提交的阶段二中，当协调者向参与者发送 commit 请求之后，发生了局部网络异常或者在发送 commit 请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了 commit 请求。而在这部分参与者接到 commit 请求之后就会执行 commit 操作。但是其他部分未接到 commit 请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据部一致性的现象。

d) 二阶段无法解决的问题：协调者再发出 commit 消息之后宕机，而唯一接收到这条消息的参与者同时也宕机了。那么即使协调者通过选举协议产生了新的协调者，这条事务的状态也是不确定的，没人知道事务是否被已经提交。

2. 三阶段提交

a) 与两阶段提交不同的是，三阶段提交有两个改动点。

i. 引入超时机制。同时在协调者和参与者中都引入超时机制。

ii. 在第一阶段和第二阶段中插入一个准备阶段。保证了在最后提交阶段之前各参与节点的状态是一致的。

也就是说，除了引入超时机制之外，3PC 把 2PC 的准备阶段再次一分为二，这样三阶段提交就有 CanCommit、PreCommit、DoCommit 三个阶段。

iii. 在 preCommit 阶段，假如有任何一个参与者向协调者发送了 No 响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断。

iv. 在 doCommit 阶段，如果参与者无法及时接收到来自协调者的 doCommit 或者 rebort 请求时，会在等待超时之后，会继续进行事务的提交。(其实这个应该是基于概率来决定的，当进入第三阶段时，说明参与者在第二阶段已经收到了 PreCommit 请求，那么协调者产生 PreCommit 请求的前提条件是他在第二阶段开始之前，收到所有参与者的 CanCommit 响应都是 Yes。(一旦参与者收到了 PreCommit，意味他知道大家其实都同意修改了) 所以，一句话概括就是，当进入第三阶段时，由于网络超时等原因，虽然参与者没有收到 commit 或者 abort 响应，但是他有理由相信：成功提交的几率很大。)

b) 2PC 与 3PC 的区别

i. 相对于 2PC，3PC 主要解决的单点故障问题，并减少阻塞，因为一旦参与者无法及时收到来自协调者的信息之后，他会默认执行 commit。而不会一直持有事务资源并处于阻塞状态。但是这种机制也会导致数据一致性问题，因为，由于网络原因，协调者发送的 abort 响应没有及时被参与者接收到，那么参与者在等待超时之后执行了 commit 操作。这样就和其他接到 abort 命令并执行回

滚的参与者之间存在数据不一致的情况。

ii. 了解了 2PC 和 3PC 之后，我们可以发现，无论是二阶段提交还是三阶段提交都无法彻底解决分布式的一致性问题。Google Chubby 的作者 Mike Burrows 说过， there is only one consensus protocol, and that’s Paxos” – all other approaches are just broken versions of Paxos. 意即世上只有一种一致性算法，那就是 Paxos，所有其他一致性算法都是 Paxos 算法的不完整版。后面的文章会介绍这个公认为难于理解但是行之有效的 Paxos 算法。

3. 盐值

a) 盐（Salt），在密码学中，是指通过在密码任意固定位置插入特定的字符串，让散列后的结果和使用原始密码的散列结果不相符，这种过程称之为“加盐”。

b) 安全因素：

通常情况下，当字段经过散列处理（如 MD5），会生成一段散列值，而散列后的值一般是无法通过特定算法得到原始字段的。但是某些情况，比如一个大型的彩虹表，通过在表中搜索该 MD5 值，很有可能在极短的时间内找到该散列值对应的真实字段内容。

加盐后的散列值，可以极大的降低由于用户数据被盗而带来的密码泄漏风险，即使通过彩虹表寻找到了散列后的数值所对应的原始内容，但是由于经过了加盐，插入的字符串扰乱了真正的密码，使得获得真实密码的概率大大降低。

c) 实现原理：加盐的实现过程通常是在需要散列的字段的特定位置增加特定的字符，打乱原始的字符串，使其生成的散列结果产生变化。比如，用户使用了一个密码：

x7faqgjw

经过 MD5 散列后，可以得出结果：455e0e5c2bc109deae749e7ce0cdd397

但是由于用户密码位数不足，短密码的散列结果很容易被彩虹表破解，因此，在用户的密码末尾添加特定字符串（下划线为加盐的字段）：

x7faqgjwabcdefghijklmnopqrstuvwxyz

因此，加盐后的密码位数更长了，散列的结果也发生了变化：

4a1690d5eb6c126ef68606dda68c2f79

以上就是加盐过程的简单描述，在实际使用过程中，还需要通过特定位数插入、倒序或多种方法对原始密码进行固定的加盐处理,使得散列的结果更加不容易被破解或轻易得到原始密码，比如

x7afabqgcjw

4. DDoS 攻击

a) 分布式拒绝服务(DDoS:Distributed Denial of Service)攻击指借助于客户/服务器技术,将多个计算机联合起来作为攻击平台,对一个或多个目标发动 DDoS 攻击，从而成倍地提高拒绝服务攻击的威力。通常，攻击者使用一个偷窃帐号将 DDoS 主控程序安装在一个计算机上，在一个设定的时间主控程序将与大量代理程序通讯，代理程序已经被安装在网络上的许多计算机上。代理程序收到指令时就发动攻击。

利用客户/服务器技术，主控程序能在几秒钟内激活成百上千次代理程序的运行。

b) 攻击方式

DDoS 攻击通过大量合法的请求占用大量网络资源，以达到瘫痪网络的目的。这种攻击方式可分为以下几种：

i. 通过使网络过载来干扰甚至阻断正常的网络通讯；

ii. 通过向服务器提交大量请求，使服务器超负荷；

iii. 阻断某一用户访问服务器；

iv. 阻断某服务与特定系统或个人的通讯。

c) 防范方法

i. 主机设置：所有的主机平台都有抵御 DoS 的设置，总结一下，基本的有几种：

1. 关闭不必要的服务
2. 限制同时打开的 Syn 半连接数目
3. 缩短 Syn 半连接的 time out 时间
4. 及时更新系统补丁

ii. 网络设置：网络设备可以从防火墙与路由器上考虑。这两个设备是到外界的接口设备，在进行防 DDoS 设置的同时，要注意一下这是以多大的效率牺牲为代价的，对你来说是否值得。

1. 防火墙：禁止对主机的非开放服务的访问 限制同时打开的 SYN 最大连接数 限制特定 IP 地址的访问 启用防火墙的防 DDoS 的属性 严格限制对外开放的服务器的向外访问 第五项主要是防止自己的服务器被当做工具去害人。
2. 路由器：设置 SYN 数据包流量速率 升级版本过低的 ISO 为路由器建立 log server

5. DNS 反射放大攻击

DNS 反射放大攻击主要是利用 DNS 回复包比请求包大的特点，放大流量，伪造请求包的源 IP 地址为受害者 IP，将应答包的流量引入受害的服务器。

简单对比下正常的 DNS 查询和攻击者的攻击方式：

正常 DNS 查询：源 IP 地址 - DNS 查询 -> DNS 服务器 - DNS 回复包 -> 源 IP 地址

DNS 攻击：伪造 IP 地址 —- DNS 查询—-> DNS 服务器 —- DNS 回复包—-> 伪造的 IP 地址（攻击目标）

DNS 反射放大攻击的原理也是类似的。网络上有大量的开放 DNS 解析服务器，它们会响应来自任何地址的解析请求。我们发出的解析请求长度是很小的，但是收到的结果却是非常大的，尤其是查询某一域名所有类型的 DNS 记录时，返回的数据量就更大，于是可以利用这些解析服务器来攻击某个目标地址的服务器，而且是利用被控制的机器发起伪造的解析请求，然后解析结果返回给被攻击目标。由于 DNS 解析一般是 UDP 请求，不需要握手，源地址属性易于伪造，而且部分“肉鸡”在平时本来就是合法的 IP 地址，我们很难验证请求的真实性和合法性。DNSSEC 是一种可以防止缓存投毒的机制，另外，如果 DNS 本身抗压能力不行，而且对方请求量过大的话，也会影响到 DNS 本身的服务。目前此类攻击的规模在数百 Gbps 级别。

6. Log-Structured File System (LFS)

a) LFS 文件系统的主要算法就是首先**把所有的更新（包括元数据）缓存在内存中的成为 segment 的单位中**。当 segment 填满之后，里面的数据就写入到磁盘中未使用的地方。 特别要注意的是：**LFS 并不会覆写已有的数据，而是把 segment 中的数据写入到磁盘中新的位置**。

b) 把所以对文件系统状态的修改更新转换为一个序列操作

当我们在新建文件或文件夹，以及向文件中添加或删除数据时后，不仅仅是文件数据在磁盘的写入，与文件相关的 i-node 也要写入磁盘，i-node 和文件所在磁盘位置相差可能很远。

仅仅只保证数据序列化写入还不够，比如在 T 时刻在地址 A 写入一个数据，准备在 T+δ 时刻往地址 A+1 写入另一个数据，但是在时间 T 和 T + δ 之间磁盘旋转了，假设磁盘旋转了 Trotation 时间，那么第二次写入实际上要等待 Trotation – δ 时间才能真正实施。**所以为了保证高性能的写入，除了序列化写入外，还要保证连续性写入**。而 LFS 就通过把数据更新缓存到内存中，然后一起写入到磁盘中，**将文件数据和 i-node 放在磁盘相邻的位置**，减少磁盘 seek time.LFS 在某一时刻写入的更新单元称为 segment.当要往磁盘写入时，LFS 把所有的更新缓存到内存中的 segment,然后在某一时刻把 segment 的所有数据一次写入磁盘。只要 segment 设置的足够大，就能够取得很好的性能。

c) 缓存大小的计算

To obtain a concrete answer, let's assume we are writing out D MB. The time to write out this chunk of data (T_{write}) is the positioning time $T_{position}$ plus the time to transfer D ($\frac{D}{R_{peak}}$), or:

$$T_{write} = T_{position} + \frac{D}{R_{peak}} \tag{43.1}$$

And thus the effective *rate* of writing ($R_{effective}$), which is just the amount of data written divided by the total time to write it, is:

$$R_{effective} = \frac{D}{T_{write}} = \frac{D}{T_{position} + \frac{D}{R_{peak}}}. \tag{43.2}$$

What we're interested in is getting the effective rate ($R_{effective}$) close to the peak rate. Specifically, we want the effective rate to be some fraction F of the peak rate, where $0 < F < 1$ (a typical F might be 0.9, or 90% of the peak rate). In mathematical form, this means we want $R_{effective} = F \times R_{peak}$.

At this point, we can solve for D:

$$R_{effective} = \frac{D}{T_{position} + \frac{D}{R_{peak}}} = F \times R_{peak} \tag{43.3}$$

$$D = F \times R_{peak} \times (T_{position} + \frac{D}{R_{peak}}) \tag{43.4}$$

$$D = (F \times R_{peak} \times T_{position}) + (F \times R_{peak} \times \frac{D}{R_{peak}}) \tag{43.5}$$

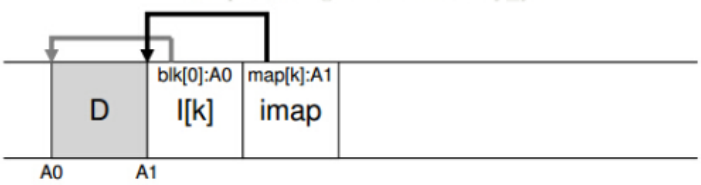
$$D = \frac{F}{1-F} \times R_{peak} \times T_{position} \tag{43.6}$$

d) 在 LFS 中如何找到 i-node

为什么要关心这个问题呢，因为在传统的文件系统比如 UNIX 系统中，i-node 存在在一个数据中，保存在磁盘中固定的位置。因此只要给定 inode 的序号以及 i-node 所在磁盘的地址，就可以通过 i-node 序号乘以 i-node 结构的长度，然后再加上磁盘地址就可以得到所需 i-node 的精确地址。但是在 LFS 系统中，i-node 是

分布在磁盘中的，并且由于在 LFS 系统中并不覆写数据，**所以一个文件的最新版本的 i-node 在磁盘中的位置是不断变化**的。

LFS 通过引入 imap 来解决 i-node 查找问题。imap 是这样的一个结构，输入一个 inode number，生成最新版本的 inode 所在磁盘地址。基于此，imap 可以实现为一个数组，每一项是 4 字节作为磁盘指针，当一个 inode 写入磁盘后，imap 就更新相应的 inode 地址。

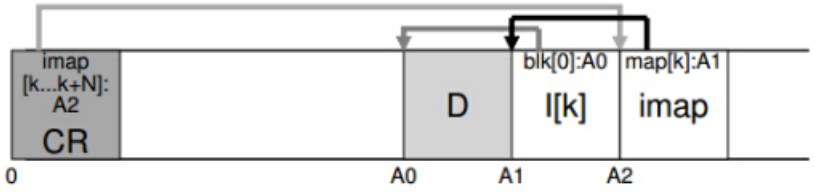


e) 使用时该如何找到这些 imap

系统中必须在一个固定磁盘的位置存储这些 imap 的信息。在 LFS 中这个固定保存 Imap 地址信息的块成为 CR.

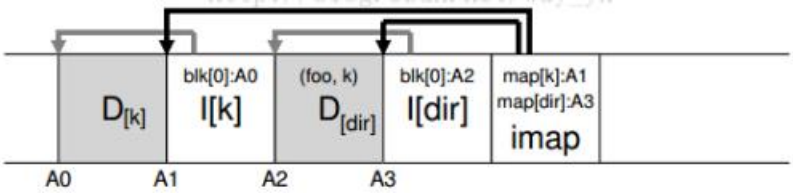
在 CR 中保存指向最新版本的 imap 的指针，系统就可以通过读取 CR 查找获取想要的 imap 信息。CR 中的信息周期性的进行更新，可能是 30 秒一次，所以对系统的性能影响不大。

所以 LFS 最终在磁盘上的整体结构包含一个 CR，imap（将 inode number 映射为 Inode 的地址），以及指向文件的 inode.



f) LFS 系统如何保存目录数据

和 UNIX 系统类似，一个目录就是(name,inode number)的结构体的集合。举例来说，当在磁盘上创建一个文件，LFS 必须写入一个新的 inode，新的数据，以及指向这个文件的目录数据和目录 inode(目录数据和 Inode 是被更新了，不过在 LFS 不会覆写数据，所以这些数据都要在磁盘新的位置中写入)。



g) LFS 还解决了另外一个问题 递归更新问题。

举例来说，当一个文件的 inode 被更新后，它在磁盘上的位置改变了,如果我们不仔细处理这种情况的话，指向这个文件的目录的也会更新，这个目录的目录也会更新，沿着文件系统树向上更新。LFS 很漂亮的避免了这个问题。即使一个 inode 的磁盘位置改变,这个改变不会反映到他的目录中,因为更新的只是 inode 的 imap，目录仍然保存一样的(name,inode number)映射。通过非直接的方法，LFS 避免了

递归更新问题。

h) 既然 LFS 不覆写数据，只是在新的磁盘位置写入数据，磁盘总有写满的时候，同时在磁盘中有大量的老版本的数据，LFS 怎么解决这个问题

如原文中所示，假设有个 Inode number 为 k 的 inode 指向磁盘块 D0，我们现在更新了文件数据，那么在磁盘中生成一个新的 inode 和数据磁盘块。结构磁盘中就会出现这样一种情况,inode,数据磁盘块有两个版本,一个老版本,一个新版本。

如原文中另外一个事例，如果在一个文件中添加数据，在磁盘中为原始文件增加了一个新的磁盘块用于保存添加的数据，我们可以看到，磁盘中增加了一个新的 inode，但是原有的数据块除了被新的 inode 指向外，还依然被老版本的 inode 指向。

对于这些老版本的 inode，数据块等应该怎么处理？一个方法是保存这些老版本的数据然后允许用户恢复到老版本（比如当如用误删除或覆写了一个文件，用这种方法就会非常方便的恢复）。这样的文件系统称为 versioning file system.

在 LFS 中只保存最新的版本和之前的老版本，LFS 周期性的扫描磁盘检测老的版本数据然后清除掉，为后续数据的腾出磁盘空间。

i) LFS 怎么清除这些老版本数据

如果仅仅只是检查时把老版本的 inode ,data,imap 等占用空间释放掉，那么在磁盘中就会产生许多的 hole，磁盘的写入性能会收到极大的影响，因为到后来可能找不到合适大小的磁盘空间来放置新的数据，虽然有许多的空磁盘，但是这些空间都太小。

LFS 最终还是基于 segment 进行清理操作。基本的清理过程如下：LFS 首先读入一些老版本的 segment,然后判断在 segment 中哪些磁盘块中的数据是 Live 的，在把这些 live 的数据写入到一个新的 segment 集合中，释放原有 segment 占用的磁盘空间。**我们期望能够读取 M 个存在的 segment,把他们的 live 内容压缩到 N 个新的 segment 中，然后释放掉老的 M 个 segment 占用的磁盘空间**，用于后续的写入。

j) 如何判定 segment 中哪些磁盘是 live 的

在 LFS 中为每一个磁盘块 D 记录了新的信息，包含 inode number(这个磁盘块属于哪个文件)和 offset（属于文件第几个磁盘块）.这个信息保存在 segment 的的头部的一个结构中，称为 segment summary block。

有了这个信息，就可以很直接的判定一个磁盘块是 live 或 dead。假设有一个磁盘块 D 位于地址 A，通过查看 segment summary block 可以找到这个磁盘块属于的 inode number N 和 offset T. 接下来,通过查看 imap 可以找到 N 对应的 inode 的地址，根据 inode 信息找到文件所占用的磁盘块，查看这个文件的第 T 个磁盘块所在地址，比较通过 inode 查找得到的地址和 segment summary block 中记录的地址，如果两个不一样，那么该磁盘块 D 就是 old 版本，不再被使用。

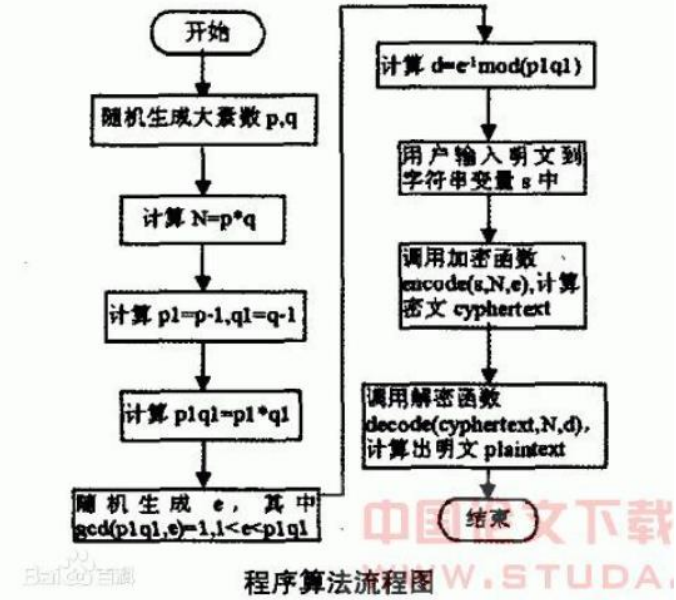
除此之外，还有一些更简单的判断磁盘 live 的方法。比如，当一个文件增加数据或者被删除了，LFS 增加这个文件的版本号，然后记录在 Imap 中，同时也记录在磁盘的 segment 中。LFS 可以通过比较磁盘块的 version number 和 imap 中的 version number 来判定这个磁盘块是否 live.

k) LFS 如何处理数据写入时系统崩溃的情况

为了确保 CR 数据写入的原子性，LFS 其实保存了两个 CR,保存在磁盘的两端，CR 更新到磁盘中时是随机挑选一个 CR 写入。LFS 同样实现了一个非常小心的协议来更新 CR。首先写入一个 header(附带写入一个 timestamp)，然后写入 CR 的主要内容，然后写入最后一个磁盘块数据以及一个 timestamp。如果在 CR 更新时系统奔溃了，LFS 通过观察到一对不一致的 timestamp 检测到出现了系统崩溃，LFS 就会采用含有一致性 timestam 的离崩溃时间最接近 CR。

l) Benchmark 证明 Sprite LFS 针对**小文件的写速度**比 Unix 文件系统要快得多。即使是针对其他的负载，例如包含**读和大文件访问**的负载，Sprite LFS 能够获得不比 Unix 文件系统差的速度，除了在**文件被随机写入后顺序读出**的测试中略有不如。我们还测试了系统进行 cleaning 的性能开销,总的来说,Sprite LFS 允许大约 60-75%的磁盘带宽进行新的数据写入(剩下的用于 cleaning)。与之相对应的是，Unix 系统只能使用 5-10%的磁盘带宽用于新的数据的写入，剩下的是用于寻道。

7. 公钥和私钥的计算方法



程序算法流程图

a) 公钥（e,N），私钥（d，N）

加密过程 cyphertext = encode(message，e，N) = message^e (mod N)

解密过程 message = decode(cyphertext，d，N) = cyphertext^d (mod N)

b) 公钥 e 和私钥 d 的生成过程

随机生成大素数 p,q，计算 N=p*q，M=(p-1)(q-1)，有关系式 e*d (mod M) = 1

随机生成与数字 e，gcd(M,e)=1，1<e<M

计算 d = (1/e) mod M

c) 已知 rsa 算法中的两个素数 p=7,q=11,公钥部分 e=7,密文 c=5 请算出私钥部分 d

解答：

由$p=7,q=11$,

可得：$n=77,\varphi(n)=\varphi(pq)=\varphi(p)\varphi(q)=(p-1)(q-1)=6\times10=60$

又由$e=7$及$de\equiv1\ mod\ \varphi(n)$

可得：$7\times d\equiv1\ mod\ 60$

也就是：$7\times d=1+60k$

易知：当k取5时，可以得到$d=43$

故现在所有的未知量都已求出

$d=43,\ \varphi(n)=60,n=77$,

则 $M=C^d\ mod\ n=5^{43}\%\ 77=26$

注：$5^{43}\%\ 77$的计算,你可以通过编程实现，也可以用快速幂来手算，快速幂又叫做逐次平方法，代码如下。

```
int fast_mod( int a, int b ){
    int res = 1;
    while( b ){
        if( b & 1 ) res = ( res * a ) % 77;
        b >>= 1;
        a = ( a * a ) % 77;
    }
    return res;
}
```

手动计算如下：

$res=1,\ a=5,\ b=43$

$res=(1\times5)\%77=5,\ a=(5\times5)\%77=25,\ b=21$

$res=(5\times25)\%77=48,\ a=(25\times25)\%77=9,\ b=10$

$res=48,\ a=(9\times9)\%77=4,\ b=5$

$res=(48\times4)\%77=38,\ a=(4\times4)\%77=16,\ b=2$

$res=38,\ a=(16\times16)\%77=25,\ b=1$

$res=(38\times25)\%77=26,\ a=(25\times25)\%77=9,\ b=0$

$b=0$，计算结束，$res=26$即为所求.

8. Diffie-Hellman 方法

基于原根的定义及性质，可以定义 Diffie-Hellman 密钥交换算法.该算法描述如下：

a) 有两个全局公开的参数，一个素数 q 和一个整数 a,a 是 q 的一个原根.

b) 假设用户 A 和 B 希望交换一个密钥，用户 A 选择一个作为私有密钥的随机数 XA(XA<q)，并计算公开密钥 YA=a^XA mod q。A 对 XA 的值保密存放而使 YA 能被 B 公开获得。类似地，用户 B 选择一个私有的随机数 XB<q，并计算公开密钥 YB=a^XB mod q。B 对 XB 的值保密存放而使 YB 能被 A 公开获得.

c) 用户 A 产生共享秘密密钥的计算方式是 K = (YB)^XA mod q.同样，用户 B 产生共享秘密密钥的计算是 K = (YA)^XB mod q.这两个计算产生相同的结果： K = (YB)^XA mod q = (a^XB mod q)^XA mod q = (a^XB)^XA mod q （根据取模运算规则得到） = a^(XBXA) mod q = (a^XA)^XB mod q = (a^XA mod q)^XB mod q = (YA)^XB mod q 因此相当于双方已经交换了一个相同的秘密密钥.

d) 因为 XA 和 XB 是保密的，一个敌对方可以利用的参数只有 q,a,YA 和 YB.因而敌对方被迫取离散对数来确定密钥.例如，要获取用户 B 的秘密密钥，敌对方必须先计算 XB = inda,q(YB) 然后再使用用户 B 采用的同样方法计算其秘密密钥 K. Diffie-Hellman 密钥交换算法的安全性依赖于这样一个事实：虽然计算以一个素数为模的指数相对容易，但计算离散对数却很困难.对于大的素数，计算出离散对数几乎是不可能的. 下面给出例子.密钥交换基于素数 q = 97 和 97 的一个原根 a = 5.A 和 B 分别选择私有密钥 XA = 36 和 XB = 58.每人计算其公开密钥 YA = 5^36 = 50 mod 97 YB = 5^58 = 44 mod 97 在他们相互获取了公开密钥之后，各自通过计算得到双方共享的秘密密钥如下： K = (YB)^XA mod 97 = 44^36 = 75 mod 97 K = (YA)^XB mod 97 = 50^58 = 75 mod 97 从|50,44|出发，攻击者要计算出 75 很不容易.