# [Kademlia详解](http://blog.chinaunix.net/uid-26548237-id-3067488.html) 2012-02-08 10:27:58

分类： 网络与安全

前两天在网上看到世界知名的电骡服务器Razorback 2被查封、4人被拘禁的消息，深感当前做eMule / BitTorrent等P2P文件交换软件的不易。以分布式哈希表方式(DHT，Distributed Hash Table)来代替集中索引服务器可以说是目前可以预见到的为数不多的P2P软件发展趋势之一，比较典型的方案主要包括：CAN、CHORD、 Tapestry、Pastry、Kademlia和Viceroy等，而Kademlia协议则是其中应用最为广泛、原理和实现最为实用、简洁的一种， 当前主流的P2P软件无一例外地采用了它作为自己的辅助检索协议，如eMule、Bitcomet、Bitspirit和Azureus等。鉴于 Kademlia日益增长的强大影响力，今天特地在blog里写下这篇小文，算是对其相关知识系统的总结。

**1. Kademlia简述**

Kademlia(简称Kad)属于一种典型的结构化P2P覆盖网络(Structured P2P Overlay Network)，以分布式的应用层全网方式来进行信息的存储和检索是其尝试解决的主要问题。在Kademlia网络中，所有信息均以的哈希表条目形式加 以存储，这些条目被分散地存储在各个节点上，从而以全网方式构成一张巨大的分布式哈希表。我们可以形象地把这张哈希大表看成是一本字典：只要知道了信息索 引的key，我们便可以通过Kademlia协议来查询其所对应的value信息，而不管这个value信息究竟是存储在哪一个节点之上。在eMule、 BitTorrent等P2P文件交换系统中，Kademlia主要充当了文件信息检索协议这一关键角色，但Kad网络的应用并不仅限于文件交换。下文的 描述将主要围绕eMule中Kad网络的设计与实现展开。

**2. eMule的Kad网络中究竟存储了哪些信息?**

只要是能够表述成为字典条目形式的信息Kad网络均能存储，一个Kad网络能够同时存储多张分布式哈希表。以eMule为例，在任一时刻，其Kad网络均存储并维护着两张分布式哈希表，一张我们可以将其命名为关键词字典，而另一张则可以称之为文件索引字典。

a. **关键词字典**：主要用于根据给出的关键词查询其所对应的文件名称及相关文件信息，其中key的值等于所给出的关键词字符串的 160比特SHA1散列，而其对应的value则为一个列表，在这个列表当中，给出了所有的文件名称当中拥有对应关键词的文件信息，这些信息我们可以简单 地用一个3元组条目表示：(文件名，文件长度，文件的SHA1校验值)，举个例子，假定存在着一个文件 “warcraft\_frozen\_throne.iso”，当我们分别以“warcraft”、“frozen”、“throne”这三个关键词来查询 Kad时，Kad将有可能分别返回三个不同的文件列表，这三个列表的共同之处则在于它们均包含着一个文件名为 “warcraft\_frozen\_throne.iso”的信息条目，通过该条目，我们可以获得对应iso文件的名称、长度及其160比特的SHA1校 验值。

b. **文件索引字典**：用于根据给出的文件信息来查询文件的拥有者(即该文件的下载服务提供者)，其中key的值等于所需下载文件的 SHA1校验值(这主要是因为，从统计学角度而言，160比特的SHA1文件校验值可以唯一地确定一份特定数据内容的文件)；而对应的value也是一个 列表，它给出了当前所有拥有该文件的节点的网络信息，其中的列表条目我们也可以用一个3元组表示：(拥有者IP，下载侦听端口，拥有者节点ID)，根据这 些信息，eMule便知道该到哪里去下载具备同一SHA1校验值的同一份文件了。

**3. 利用Kad网络搜索并下载文件的基本流程是怎样的?**

基于我们对eMule的Kad网络中两本字典的理解，利用Kad网络搜索并下载某一特定文件的基本过程便很明白了，仍以 “warcraft\_frozen\_throne.iso”为例，首先我们可以通过warcraft、frozen、throne等任一关键词查询关键词 字典，得到该iso的SHA1校验值，然后再通过该校验值查询Kad文件索引字典，从而获得所有提供 “warcraft\_frozen\_throne.iso”下载的网络节点，继而以分段下载方式去这些节点下载整个iso文件。

在上述过程中，Kad网络实际上所起的作用就相当于两本字典，但值得再次指出的是，Kad并不是以集中的索引服务器(如华语P2P源动力、 Razorback 2、DonkeyServer 等，骡友们应该很熟悉吧)方式来实现这两本字典的存储和搜索的，因为这两本字典的所有条目均分布式地存储在参与Kad网络的各节点中，相关文件信息、下载 位置信息的存储和交换均无需集中索引服务器的参与，这不仅提高了查询效率，而且还提高了整个P2P文件交换系统的可靠性，同时具备相当的反拒绝服务攻击能 力；更有意思的是，它能帮助我们有效地抵制FBI的追捕，因为俗话说得好：法不治众…看到这里，相信大家都能理解“分布式信息检索”所带来的好处了吧。但 是，这些条目究竟是怎样存储的呢?我们又该如何通过Kad网络来找到它们?不着急，慢慢来。

**4. 什么叫做节点的ID和节点之间的距离?**

Kad网络中的每一个节点均拥有一个专属ID，该ID的具体形式与SHA1散列值类似，为一个长达160bit的整数，它是由节点自己随机生成的， 两个节点拥有同一ID的可能性非常之小，因此可以认为这几乎是不可能的。在Kad网络中，两个节点之间距离并不是依靠物理距离、路由器跳数来衡量的，事实 上，Kad网络将任意两个节点之间的距离d定义为其二者ID值的逐比特二进制和数，即，假定两个节点的ID分别为a与b，则有：d=a XOR b。在Kad中，每一个节点都可以根据这一距离概念来判断其他节点距离自己的“远近”，当d值大时，节点间距离较远，而当d值小时，则两个节点相距很近。 这里的“远近”和“距离”都只是一种逻辑上的度量描述而已；在Kad中，距离这一度量是无方向性的，也就是说a到b的距离恒等于b到a的距离，因为a XOR b==b XOR a

**5. 条目是如何存储在Kad网络中的?**

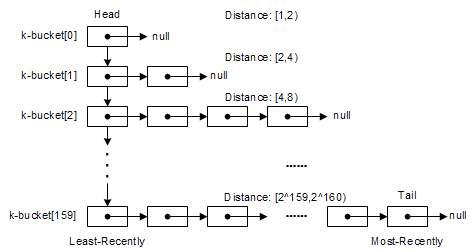
从上文中我们可以发现节点ID与条目中key值的相似性：无论是关键词字典的key，还是文件索引字典的key，都是160bit，而节点ID恰恰 也是160bit。这显然是有目的的。事实上，节点的ID值也就决定了哪些条目可以存储在该节点之中，因为我们完全可以把某一个条目简单地存放在节点ID 值恰好等于条目中key值的那个节点处，我们可以将满足(ID==key)这一条件的节点命名为目标节点N。这样的话，一个查找条目的问题便被简单地转化 成为了一个查找ID等于Key值的节点的问题。

由于在实际的Kad网络当中，并不能保证在任一时刻目标节点N均一定存在或者在线，因此Kad网络规定：任一条目，依据其key的具体取值，该条目 将被复制并存放在节点ID距离key值最近(即当前距离目标节点N最近)的k个节点当中；之所以要将重复保存k份，这完全是考虑到整个Kad系统稳定性而 引入的冗余；这个k的取值也有讲究，它是一个带有启发性质的估计值，挑选其取值的准则为：“在当前规模的Kad网络中任意选择至少k个节点，令它们在任意 时刻同时不在线的几率几乎为0”；目前，k的典型取值为20，即，为保证在任何时刻我们均能找到至少一份某条目的拷贝，我们必须事先在Kad网络中将该条 目复制至少20份。

由上述可知，对于某一条目，在Kad网络中ID越靠近key的节点区域，该条目保存的份数就越多，存储得也越集中；事实上，为了实现较短的查询响应 延迟，在条目查询的过程中，任一条目可被cache到任意节点之上；同时为了防止过度cache、保证信息足够新鲜，必须考虑条目在节点上存储的时效性： 越接近目标结点N，该条目保存的时间将越长，反之，其超时时间就越短；保存在目标节点之上的条目最多能够被保留24小时，如果在此期间该条目被其发布源重 新发布的话，其保存时间还可以进一步延长。

**6. Kad网络节点需要维护哪些状态信息?**

在Kad网络中，每一个节点均维护了160个list，其中的每个list均被称之为一个k-桶(k-bucket)，如下图所示。在第i个 list中，记录了当前节点已知的与自身距离为2^i~2^(i+1)的一些其他对端节点的网络信息(Node ID，IP地址，UDP端口)，每一个list(k-桶)中最多存放k个对端节点信息，注意，此处的k与上文所提到的复制系数k含义是一致的；每一个 list中的对端节点信息均按访问时间排序，最早访问的在list头部，而最近新访问的则放在list的尾部。



k-桶中节点信息的更新基本遵循Least-recently Seen Eviction原则：当list容量未满(k-桶中节点个数未满k个)，且最新访问的对端节点信息不在当前list中时，其信息将直接添入list队 尾，如果其信息已经在当前list中，则其将被移动至队尾；在k-桶容量已满的情况下，添加新节点的情况有点特殊，它将首先检查最早访问的队首节点是否仍 有响应，如果有，则队首节点被移至队尾，新访问节点信息被抛弃，如果没有，这才抛弃队首节点，将最新访问的节点信息插入队尾。可以看出，尽可能重用已有节 点信息、并且按时间排序是k-桶节点更新方式的主要特点。从启发性的角度而言，这种方式具有一定的依据：在线时间长一点的节点更值得我们信任，因为它已经 在线了若干小时，因此，它在下一个小时以内保持在线的可能性将比我们最新访问的节点更大，或者更直观点，我这里再给出一个更加人性化的解释：MP3文件交 换本身是一种触犯版权法律的行为，某一个节点反正已经犯了若干个小时的法了，因此，它将比其他新加入的节点更不在乎再多犯一个小时的罪……-\_-b

由上可见，设计采用这种多k-bucket数据结构的初衷主要有二：a. 维护最近-最新见到的节点信息更新；b. 实现快速的节点信息筛选操作，也就是说，只要知道某个需要查找的特定目标节点N的ID，我们便可以从当前节点的k-buckets结构中迅速地查出距离N 最近的若干已知节点。

**7. 在Kad网络中如何寻找某特定的节点?**

已知某节点ID，查找获得当前Kad网络中与之距离最短的k个节点所对应的网络信息(Node ID，IP地址，UDP端口)的过程，即为Kad网络中的一次节点查询过程(Node Lookup)。注意，Kad之所以没有把节点查询过程严格地定义成为仅仅只查询单个目标节点的过程，这主要是因为Kad网络并没有对节点的上线时间作出 任何前提假设，因此在多数情况下我们并不能肯定需要查找的目标节点一定在线或存在。

整个节点查询过程非常直接，其方式类似于DNS的迭代查询：  
a. 由查询发起者从自己的k-桶中筛选出若干距离目标ID最近的节点，并向这些节点同时发送异步查询请求；  
b .被查询节点收到请求之后，将从自己的k-桶中找出自己所知道的距离查询目标ID最近的若干个节点，并返回给发起者；  
c. 发起者在收到这些返回信息之后，再次从自己目前所有已知的距离目标较近的节点中挑选出若干没有请求过的，并重复步骤1；  
d. 上述步骤不断重复，直至无法获得比查询者当前已知的k个节点更接近目标的活动节点为止。  
e. 在查询过程中，没有及时响应的节点将立即被排除；查询者必须保证最终获得的k个最近节点都是活动的。

简单总结一下上述过程，实际上它跟我们日常生活中去找某一个人打听某件事是非常相似的，比方说你是个Agent Smith，想找小李(key)问问他的手机号码(value)，但你事先并不认识他，你首先肯定会去找你所认识的和小李在同一个公司工作的人，比方说小 赵，然后小赵又会告诉你去找与和小李在同一部门的小刘，然后小刘又会进一步告诉你去找和小李在同一个项目组的小张，最后，你找到了小张，哟，正好小李出差 去了(节点下线了)，但小张恰好知道小李的号码，这样你总算找到了所需的信息。在节点查找的过程中，“节点距离的远近”实际上与上面例子中“人际关系的密 切程度”所代表的含义是一样的。

最后说说上述查询过程的局限性：Kad网络并不适合应用于模糊搜索，如通配符支持、部分查找等场合，但对于文件共享场合来说，基于关键词的精确查找 功能已经基本足够了(值得注意的是，实际上我们只要对上述查找过程稍加改进，并可以令其支持基于关键词匹配的布尔条件查询，但仍不够优化)。这个问题反映 到eMule的应用层面来，它直接说明了文件共享时其命名的重要性所在，即，文件名中的关键词定义得越明显，则该文件越容易被找到，从而越有利于其在 P2P网络中的传播；而另一方面，在eMule中，每一个共享文件均可以拥有自己的相关注释，而Comment的重要性还没有被大家认识到：实际上，这个 文件注释中的关键词也可以直接被利用来替代文件名关键词，从而指导和方便用户搜索，尤其是当文件名本身并没有体现出关键词的时候。

**8. 在Kad网络中如何存储和搜索某特定的条目?**

从本质上而言，存储、搜索某特定条目的问题实际上就是节点查找的问题。当需要在Kad网络中存储一个条目时，可以首先通过节点查找算法找到距离 key最近的k个节点，然后再通知它们保存条目即可。而搜索条目的过程则与节点查询过程也是基本类似，由搜索发起方以迭代方式不断查询距离key较近的节 点，一旦查询路径中的任一节点返回了所需查找的value，整个搜索的过程就结束。为提高效率，当搜索成功之后，发起方可以选择将搜索到的条目存储到查询 路径的多个节点中，作为方便后继查询的cache；条目cache的超时时间与节点-key之间的距离呈指数反比关系。

**9. 一个新节点如何首次加入Kad网络?**

当一个新节点首次试图加入Kad网络时，它必须做三件事，其一，不管通过何种途径，获知一个已经加入Kad网络的节点信息(我们可以称之为节点 I)，并将其加入自己的k-buckets；其二，向该节点发起一次针对自己ID的节点查询请求，从而通过节点I获取一系列与自己距离邻近的其他节点的信 息；最后，刷新所有的k-bucket，保证自己所获得的节点信息全部都是新鲜的。

# [P2P之Kademlia (一)](http://blog.csdn.net/cz_hyf/article/details/5076988)

标签： [p2p](http://www.csdn.net/tag/p2p)[网络](http://www.csdn.net/tag/%e7%bd%91%e7%bb%9c)[存储](http://www.csdn.net/tag/%e5%ad%98%e5%82%a8)[算法](http://www.csdn.net/tag/%e7%ae%97%e6%b3%95)[通讯](http://www.csdn.net/tag/%e9%80%9a%e8%ae%af)[数据库](http://www.csdn.net/tag/%e6%95%b0%e6%8d%ae%e5%ba%93)

2009-12-25 17:14 5247人阅读 [评论](http://blog.csdn.net/cz_hyf/article/details/5076988#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/cz_hyf/article/details/5076988#report)

本文章已收录于：

http://static.blog.csdn.net/images/category_icon.jpg分类：

研究基础（13） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpghttp://static.blog.csdn.net/images/arrow_triangle_up.jpg

[作者同类文章](http://blog.csdn.net/cz_hyf/article/category/258047)*X*

参考原文:http://en.wikipedia.org/wiki/Kademlia

参考资料:http://blog.csdn.net/tsingmei/archive/2008/09/13/2924368.aspx

**Kademlia**

        Kademlia是一种通过分布式哈希表实现的协议算法，他是由Petar和David为非集中式P2P计算机网络而设计的。Kademlia规定了网络的结构，也规定了通过节点查询进行信息交换的方式。Kademlia网络节点之间使用UDP进行通讯。参与通讯的所有节点形成一张虚拟网（或者叫做覆盖网）。这些节点通过一组数字（或称为节点ID）来进行身份标识。节点ID不仅可以用来做身份标识，还可以用来进行值定位（值通常是文件的散列或者关键词）。其实，节点ID与文件散列直接对应，它所表示的那个节点存储着哪儿能够获取文件和资源的相关信息。

      当我们在网络中搜索某些值（即通常搜索存储文件散列或关键词的节点）的时候，Kademlia算法需要知道与这些值相关的键，然后分步在网络中开始搜索。每一步都会找到一些节点，这些节点的ID与键更为接近，如果有节点直接返回搜索的值或者再也无法找到与键更为接近的节点ID的时候搜素便会停止。这种搜索值的方法是非常高效的：与其他的分布式哈希表的实现类似，在一个包含n个节点的系统的值的搜索中，Kademlia仅访问O(log(n))个节点。

        非集中式网络结构还有更大的优势，那就是它能够显著增强抵御拒绝服务攻击的能力。即使网络中的一整批节点遭受泛洪攻击，也不会对网络的可用性造成很大的影响，通过绕过这些漏洞（被攻击的节点）来重新编织一张网络，网络的可用性就可以得到恢复。

**内容**

**1 系统细节**

**1.1 路由表  
 1.2 系统细节  
 1.3 定位节点  
 1.4 定位资源  
 1.5 加入Kademlia网络  
 1.6 查询加速**

**2 学术意义**

**3 在文件分享网络中的应用**

**1 系统细节**

        第一代P2P文件分享网络，像Napster,依赖于中央数据库来协调网络中的查询，第二代P2P网络，像Gnutella,使用泛洪来查询文件，它会搜索网络中的所有节点，第三代p2p网络使用分布式哈希表来查询网络中的文件，分布式哈希表在整个网络中储存资源的位置，这些协议追求的主要目标就是快速定位期望的节点。

        Kademlia基于两个节点之间的距离计算，该距离是两个网络节点ID号的异或，计算的结果最终作为整形数值返回。关键字和节点ID有同样的格式和长度，因此，可以使用同样的方法计算关键字和节点ID之间的距离。节点ID一般是一个大的随机数，选择该数的时候所追求的一个目标就是它的唯一性（希望在整个网络中该节点ID是唯一的）。异或距离跟实际上的地理位置没有任何关系，只与ID相关。因此很可能来自德国和澳大利亚的节点由于选择了相似的随机ID而成为邻居。

        选择异或是因为通过它计算的距离享有几何距离公式的一些特征，尤其体现在以下几点：

        节点和它本身之间的异或距离是0

        异或距离是对称的：即从A到B的异或距离与从B到A的异或距离是等同的

        异或距离符合三角形不等式：给定三个顶点A B C，假如AC之间的异或距离最大,那么AC之间的异或距离必小于或等于AB异或距离和BC异或距离之和.

        由于以上的这些属性，在实际的节点距离的度量过程中计算量将大大降低。Kademlia搜索的每一次迭代将距目标至少更近1 bit。一个基本的具有2的n次方个节点的Kademlia网络在最坏的情况下只需花n步就可找到被搜索的节点或值。

**1.1 路由表**

        为了说明简单，本部分基于单个bit构建路由表，如需关于实际路由表的更多信息，请看“查询加速”部分。

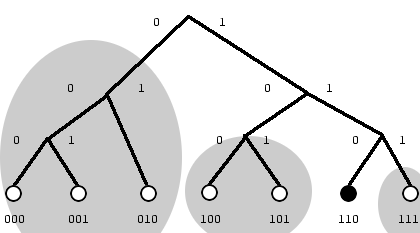
        Kademlia路由表由多个列表组成，每个列表对应节点ID的一位（例如:假如节点ID共有128位，则节点的路由表将包含128个列表），包含多个条目，条目中包含定位其他节点所必要的一些数据。列表条目中的这些数据通常是由其他节点的IP地址，端口和节点ID组成。每个列表对应于与节点相距特定范围距离的一些节点，节点的第n个列表中所找到的节点的第n位与该节点的第n位肯定不同，而前n-1位相同，这就意味着很容易使用网络中远离该节点的一半节点来填充第一个列表（第一位不同的节点最多有一半），而用网络中四分之一的节点来填充第二个列表（比第一个列表中的那些节点离该节点更近一位），依次类推。

        如果ID有128个二进制位，则网络中的每个节点按照不同的异或距离把其他所有的节点分成了128类，ID的每一位对应于其中的一类。

        随着网络中的节点被某节点发现，它们被逐步加入到该节点的相应的列表中，这个过程中包括向节点列表中存信息和从节点列表中取信息的操作，甚至还包括当时协助其他节点寻找相应键对应值的操作。这个过程中发现的所有节点都将被加入到节点的列表之中，因此节点对整个网络的感知是动态的，这使得网络一直保持着频繁地更新，增强了抵御错误和攻击的能力。

        在Kademlia相关的文字作品中，列表也称为K桶，其中K是一个系统变量，如20，每一个K桶是一个最多包含K个条目的列表，也就是说，网络中所有节点的一个列表（对应于某一位，与该节点相距一个特定的距离）最多包含20个节点。

        随着对应的bit位变低（即对应的异或距离越来越短），K桶包含的可能节点数迅速下降（这是由于K桶对应的异或距离越近，节点数越少），因此，对应于更低bit位的K桶显然包含网络中所有相关部分的节点。由于网络中节点的实际数量远远小于可能ID号的数量，所以对应那些短距离的某些K桶可能一直是空的（如果异或距离只有1，可能的数量就最大只能为1，这个异或距离为1的节点如果没有发现，则对应于异或距离为1的K桶则是空的）。



        让我们看上边的那个简单网络，该网络最大可有2^3，即8个关键字和节点，目前共有7个节点加入，每个节点用一个小圈表示（在树的底部）。我们考虑那个用黑圈标注的节点6，它共有3个K桶，节点0，1和2（二进制表示为000，001和010)是第一个K桶的候选节点，节点3目前（二进制表示为011）还没有加入网络，节点4和节点5（二进制表示分别为100和101）是第二个K桶的候选节点，只有节点7（二进制表示为111）是第3个K桶的候选节点。图中，3个K桶都用灰色圈表示，假如K桶的大小（即K值）是2，那么第一个K桶只能包含3个节点中的2个。

        众所周知，那些长时间在线连接的节点未来长时间在线的可能性更大，基于这种静态统计分布的规律，Kademlia选择把那些长时间在线的节点存入K桶，这一方法增长了未来某一时刻有效节点的数量，同时也提供了更为稳定的网络。

        当某个K桶已满，而又发现了相应于该桶的新节点的时候，那么，就首先检查K桶中最早访问的节点，假如该节点仍然存活，那么新节点就被安排到一个附属列表中(作为一个替代缓存).只有当K桶中的某个节点停止响应的时候，替代cache才被使用。换句话说，新发现的节点只有在老的节点消失后才被使用。

**1.2 协议消息**

        Kademlia协议共有四种消息。

        PING消息—用来测试节点是否仍然在线。  
   
        STORE消息—在某个节点中存储一个键值对。  
   
        FIND\_NODE消息—消息请求的接收者将返回自己桶中离请求键值最近的K个节点。

        FIND\_VALUE消息，与FIND\_NODE一样，不过当请求的接收者存有请求者所请求的键的时候，它将返回相应键的值。

        每一个RPC消息中都包含一个发起者加入的随机值，这一点确保响应消息在收到的时候能够与前面发送的请求消息匹配。

# [P2P之Kademlia (二)](http://blog.csdn.net/cz_hyf/article/details/5178330)

标签： [p2p](http://www.csdn.net/tag/p2p)[网络](http://www.csdn.net/tag/%e7%bd%91%e7%bb%9c)[存储](http://www.csdn.net/tag/%e5%ad%98%e5%82%a8)[performance](http://www.csdn.net/tag/performance)[system](http://www.csdn.net/tag/system)[internet](http://www.csdn.net/tag/internet)

2010-01-11 22:24 4178人阅读 [评论](http://blog.csdn.net/cz_hyf/article/details/5178330#comments)(3) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/cz_hyf/article/details/5178330#report)

本文章已收录于：

http://static.blog.csdn.net/images/category_icon.jpg分类：

研究基础（13） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpghttp://static.blog.csdn.net/images/arrow_triangle_up.jpg

[作者同类文章](http://blog.csdn.net/cz_hyf/article/category/258047)*X*

**1.3 定位节点**

    节点查询可以异步进行，也可以同时进行，同时查询的数量由α表示，一般是3。在节点查询的时候，

它先得到它K桶中离所查询的键值最近的K个节点，然后向这K个节点发起 FIND\_NODE消息请求，消息

接收者收到这些请求消息后将在他们的K桶中进行查询，如果他们知道离被查键更近的节点，他们就返回

这些节点（最多K个）。消息的请求者在收到响应后将使用它所收到的响应结果来更新它的结果列表，这个

结果列表总是保持K个响应FIND\_NODE消息请求的最优节点（即离被搜索键更近的K个节点）。 然后消息发

起者将向这K个最优节点发起查询，不断地迭代执行上述查询过程。因为每一个节点比其他节点对它周边的节

点有更好的感知能力，因此响应结果将是一次一次离被搜索键值越来越近的某节点。如果本次响应结果中的节

点没有比前次响应结果中的节点离被搜索键值更近了，这个查询迭代也就终止了。 当这个迭代终止的时候，响

应结果集中的K个最优节点就是整个网络中离被搜索键值最近的K个节点（从以上过程看，这显然是局部的，而

非整个网络）

    节点信息中可以增加一个往返时间，或者叫做RTT的参数，这个参数可以被用来定义一个针对每个被查

询节点的超时设置，即当向某个节点发起的查询超时的时候，另一个查询才会发起，当然，针对某个节点的查询

在同一时刻从来不超过α个。

**1.4 定位资源**

    通过把资源信息与键进行映射，资源即可进行定位，哈希表是典型的用来映射的手段。由于以前的STORE

消息，存储节点将会有对应STORE所存储的相关资源的信息。定位资源时，如果一个节点存有相应的资源的

值的时候，它就返回该资源，搜索便结束了，除了该点以外，定位资源与定位离键最近的节点的过程相似。

     考虑到节点未必都在线的情况，资源的值被存在多个节点上（节点中的K个），并且，为了提供冗余，还

有可能在更多的节点上储存值。 储存值的节点将定期搜索网络中与储存值所对应的键接近的K个节点并且把

值复制到这些节点上，这些节点可作为那些下线的节点的补充。另外，对于那些普遍流行的内容，可能有更

多的请求需求，通过让那些访问值的节点 把值存储在附件的一些节点上（不在K个最近节点的范围之类）来

减少存储值的那些节点的负载，这种新的存储技术就是缓存技术。通过这种技术，依赖于请求的数量，资源

的值被存储在离键越来越远的那些节点上，这使得那些流行的搜索可以更快地找到资源的储存者。由于返回

值的节点的NODE\_ID远离值所对应的关键字，网络中的“热点”区域存在的可能性也降低了。依据与键的距

离，缓存的那些节点在一段时间以后将会删除所存储的缓存值。 DHT的某些实现（如Kad）即不提供冗余

（复制）节点也不提供缓存，这主要是为了能够快速减少系统中的陈旧信息。在这种网络中，提供文件的

那些节点将会周期性地更新网络上的信息（通过NODE\_LOOKUP消息和STORE消息）。当存有某个文件

的所有节点都下线了，关于该文件的相关的值（源和关键字）的更新也就停止了，该文件的相关信息也就

从网络上完全消失了。

**1.5 加入网络**

      想要加入网络的节点首先要经历一个引导过程。 在引导过程中，节点需要知道其他已加入该网络的某个

节点的IP地址和端口号（可从用户或者存储的列表中获得）。假如正在引导的那个节点还未加入网络，它

会计算一个目前为止还未分配给其他节点的随机ID号， 直到离开网络，该节点会一直使用该ID号。

      正在加入Kademlia网络的节点在它的某个K桶中插入引导节点（负责加入节点的初始化工作），然后向

它的唯一邻居（引导节点）发起NODE\_LOOKUP操作请求来定位自己，这种“自我定位”将使得Kademlia

的其他节点（收到请求的节点）能够使用新加入节点的Node Id填充他们的K桶，同时也能够使用那些查询过

程的中间节点(位于新加入节点和引导节点的查询路径上的其他节点)来填充新加入节点的K桶。这一自查询过

程使得新加入节点自引导节点所在的那个K桶开始，由远及近，逐个得到刷新，这种刷新只需通过位于K桶范

围内的一个随机键的定位便可达到。

      最初的时候，节点仅有一个K桶（覆盖所有的ID范围），当有新节点需要插入该K桶时，如果K桶已满，K桶

就开始分裂，（参见A Peer-to-peer Information System 2.4）分裂发生在节点的K桶的覆盖范围（表现为二叉

树某部分从左至右的所有值）跨过了该节点本身的ID的时候。对于节点内距离节点最近的那个K桶，Kademlia

可以放松限制（即可以到达K时不发生分裂），因为桶内的所有节点离该节点距离最近，这些节点个数很可能

超过K个，而且节点希望知道所有的这些最近的节点。因此，在路由树中，该节点附近很可能出现高度不平衡

的二叉子树。假如K是20，新加入网络的节点ID为“xxx000011001”，则前缀为“xxx0011……”的节点可能有

21个，甚至更多，新的节点可能包含多个含有21个以上节点的K桶。（位于节点附近的k桶）。 这点保证使得

该节点能够感知网络中附近区域的所有节点。（参见A Peer-to-peer Information System 2.4）

**1.6 查询加速**

    Kademlia使用异或来定义距离。两个节点ID的异或（或者节点ID和关键字的异或）的结果就是两者之间的距

离。对于每一个二进制位来说，如果相同，异或返回0，否则，异或返回1。异或距离满足三角形不等式：任何

一边的距离小于（或等于）其它两边距离之和。

   异或距离使得Kademlia的路由表可以建在多个bit之上，即可使用位组（多个位联合）来构建路由表。位组可

以用来表示相应的K桶，它有个专业术语叫做前缀，对一个m位的前缀来说，可对应2^m-1个K桶。(m位的前缀

本来可以对应2^m个K桶)另外的那个K桶可以进一步扩展为包含该节点本身ID的路由树。一个b位的前缀可以把

查询的最大次数从logn减少到logn/b.这只是查询次数的最大值，因为自己K桶可能比前缀有更多的位与目标键

相同，（这会增加在自己K桶中找到节点的机会，假设前缀有m位，很可能查询一个节点就能匹配2m甚至更多

的位组），所以其实平均的查询次数要少的多。

（参考Improving Lookup Performance over a Widely-Deployed DHT第三部分）

    节点可以在他们的路由表中使用混合前缀，就像eMule中的Kad网络。如果以增加查询的复杂性为代价，

Kademlia网络在路由表的具体实现上甚至可以是有异构的。

**2 学术意义**

    尽管异或标准对于理解Kademlia并不是必要，但是对于协议的分析却至关重要。 异或运算形成了允许闭

合分析的循环群，为了能够预见网络的行为和正确性，其他的一些DHT协议和算法都要求模拟或复杂的形式

分析，而Kademlia并不需要，另外，把位组作为路由信息也简化了Kademlia算法。

**3 在文件分享网络中的应用**

    Kademlia可在文件分享网络中使用，通过制作Kademlia关键字搜索，我们能够在文件分享网络中找到我们

需要的文件以供我们下载。由于没有中央服务器存储文件的索引，这部分工 作就被平均地分配到所有的客户

端中去：假如一个节点希望分享某个文件，它先根据文件的内容来处理该文件，通过运算，把文件的内容散列

成一组数字，该数字在文件分享网络中可被用来标识文件。这组散列数字必须和节点ID有同样的长度，然后，

该节点便在网络中搜索ID值与文件的散列值相近的节点，并把它自己的IP地址存储在那些搜索到的节点上，也就

是说，它把自己作为文件的源进行了发布。正在进行文件搜索的客户端将使用Kademlia协议来寻找网络上ID值

与希望寻找的文件的散列值最近的那个节点，然后取得存储在那个节点上的文件源列表。由于一个键可以对应

很多值，即同一个文件可以有多个源，每一个存储源列表的节点可能有不同的文件的源的信息，这样的话，

源列表可以从与键值相近的K个节点获得。

    文件的散列值通常可以从其他的一些特别的Internet链接的地方获得，或者被包含在从其他某处获得的索引文

件中。文件名的搜索可以使用关键词来实现，文件名可以分割成连续的几个关键词，这些关键词都可以散列并

且可以和相应的文件名和文件散列储存在网络中。搜索者可以使用其中的某个关键词，联系ID值与关键词散列

最近的那个节点，取得包含该关键词的文件列表。由于在文件列表中的文件都有相关的散列值，通过该散列值

就可利用上述通常取文件的方法获得要搜索的文件。

# 经典论文系列，供学习和交流，欢迎指正。

# Kademlia ：一种基于 XOR 度量的 P2P 信息系统

Petar Maymounkov and David Mazieres

{petar,dm}@cs.nyu.edu

[http://kademlia.scs.cs.nyu.edu](http://kademlia.scs.cs.nyu.edu/)

New York University

原文： <http://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf>

**邓辉 译**

**摘要：**

       在本文中，我们介绍一种 P2P 的分布式 hash 表，其在不可靠的环境中也具有可证明的一致性和性能特征。我们的系统使用一种新颖的基于 XOR 度量的拓扑方法来路由查询请求和定位节点，该方法简化了算法，且方便了证明。该拓扑方法具有这样的特性：节点间所交换的每条消息都传递或者增强着有用的联系信息。系统利用该信息来发送并行、异步的查询消息以容忍节点故障，同时又不会让用户感到时延。

## 1 介绍

       本文介绍了 Kademlia ，一种 P2P 的分布式 hash 表（ DHT ）。 Kademlia 具有很多所希望的特性，而这些特性是任何先前的 DHT 无法同时拥有的。它最小化了节点为了相互了解而发送的配置消息的数量。配置信息伴随着 key 的查询自动地散布开来。节点具有可以在低响应性的路径上路由查询请求所需要的足够的知识和灵活性。 Kademlia 的查询请求是并行、异步的，从而避免了故障节点所导致的超时延迟问题。节点在记录其他节点是否可用时所采用的算法可以阻止某些基本的拒绝服务工具。最后， Kademlia 的几个重要属性可以仅仅基于一些关于在线时间分布的弱假设就可以被规范地证明（我们通过对当前 P2P 系统的度量证实了这些假设）。

       Kademlia 采用了很多 DHTs 中所使用的基本方法。 Keys 是不透明的、 160 位长的数（比如：某些大数据的 SHA-1 hash 值）。参与其中的计算机都具有一个节点 ID ，位于这 160 位 key 空间之中。根据某种对“接近”的定义， <key ， value> 对被存储在 ID 和 key “接近”的节点之上。最终，一种基于节点 ID 的路由算法可以让每个节点都能够高效地定位出和任意给定目标 key 接近的服务器。

       在 Kademlia 中， key 空间中的点之间的距离是基于 XOR 度量进行的，这种新颖的方法为其带来了诸多好处。 XOR 是对称的，因此 Kademlia 节点所接收到的 lookup 查询请求的发送者就具有和其路由表中的节点完全一样的分布。不具有这个特性的系统（比如： Chord 【 5 】）无法从其所收到的查询请求中学习有用的路由信息。更糟糕的是，缺乏这种对称性会导致呆板的路由表。在 Chord 中，节点 finger 表中的每个条目中所存储的节点，必须恰好是 ID 空间中的某个区间的前驱。实际存在于该区间中的任何一个节点和该区间中的前驱节点之间的距离可能相距甚远。而 Kademlia 则可以向一个区间中的任何节点发送查询请求，这样就可以根据响应时间选择路由，甚至可以向多个合适的节点发送并行、异步的查询请求。

       在找出那些和某个特定 ID 接近的节点的过程中， Kademlia 仅使用了唯一的一个路由算法。其他的系统则不是这样，它们先使用一个算法来接近目标 ID ，然后在最后几步定位中则采用了另外一个算法。在当前的系统中， Pastry 【 1 】中的第一阶段算法与 Kademlia 最为相似，也是采用 Kademlia 的 XOR 度量方法连续查找，以大至折半的速度逼近目标 ID （其作者并没有这样去描述）。但是，在其第二阶段中， Pastry 改变的距离度量方法，转为使用 IDs 间的数值差。在数据复制时， Pastry 同样采用了另外一种数值差度量方法。遗憾的是，在第 2 种度量方法中距离接近的节点用第一种方法来度量可能相距甚远，从而导致针对某个特定节点 ID 的不连续性，造成性能下降，并且难以对其最差情况进行规范化分析。

## 2 系统描述

       总的来说，我们采用了和其他 DHTs 中完全一样的方法：节点的 ID 是 160 位不透明的值，我们的算法也是逐步“逼近”所期望的 ID ，并以对数级的速度收敛到要查询的目标。

       Kademlia 把节点看做是一棵二叉树的叶子，其中每个节点的位置由其 ID 的最短唯一前缀决定。图 1 展示了一棵示例树中唯一前缀为 0011 的节点的位置。对于任何一个给定节点，我们都把树划分成一系列不包含该节点的逐步降低的子树。最高层的子树由二叉树中不含该节点的那一半组成。接下来的子树由剩下的树中不包含该节点的子树组成，以此类推。对于示例图中的节点 0011 ，子树被圆圈标识出来，分别由所有前缀为 1 ， 01 ， 000 以及 0010 的节点组成。

       Kademlia 协议保证了每个节点都至少知晓这些子树中的一个节点（如果子树不为空的话）。有了这个保证，每个节点就都可以通过 ID 来定位其他任何节点。图 2 展示了节点 0011 定位节点 1110 的示例，其中节点 0011 通过逐步查询它所知道的最佳节点来取得和越来越低层次子树的联系；最后查询收敛到目标节点。

       接下来，我们会补充一些细节，并更具体的描述查询算法。首先，我们会给出 ID 接近这个概念的准确定义，这样就可以谈及“在距离 key 最近的 k 个节点上存储或者查询 <key ， value> 对”这样的行为。然后，我们会介绍一个查询协议，该协议即使在任何节点都不和某个 key 具有相同的前缀或者和某个给定节点关联的子树中有一些为空的情况下，都可以正常工作。

### 2.1 XOR 度量

       每个 Kademlia 节点都具有一个 160 位的节点 ID 。我们可以像 Chord 那样对节点 IDs 进行规范的组织，不过，目前它们只是随机的 160 位标示符。节点所发送的每个消息中都包括该节点的 ID ，这样接收者就可以在需要时记录下发送者的在线状态。

       Keys 也是 160 位的标示符。 Kademlia 根据两个标示符之间的距离来把 <key ， value> 对分配给特定的节点。对于两个 160 位标示符 x 和 y ， Kademlia 把它们之间的距离定义为二者按位异或（ XOR ）结果的整数值， d(x,y)=x ⊕y 。

    首先，XOR 是一种有效的度量方法（虽然不是欧几里得几何意义上的）。很显然，d(x,x)=0 ；当x 不等于y 时，d(x,y)>0 ，并且对于任意的x,y 来说，d(x,y)=d(y,x) 。XOR 还具有三角特性：d(x,y)+d(y,z) 大于等于d （x,z ）。该三角特性可以从如下事实得出：d(x,y) ⊕d(y,z)=d(x,z) ，并且对于任意大于等于0 的a 和b ：a+b 都大于等于a ⊕b 。

    同时，XOR 也刻画出了隐含在我们基于二叉树描绘的系统中距离的概念。在160 位ID 的满二叉树中，两个IDs 间的距离大小就是包含它们的最小子树的高度。当树不是满的时，距离ID x 最近的叶子就是其ID 和x 具有最长的公共前缀的那个叶子。如果树中有空的分支，那么具有最长公共前缀的叶子就会有多个。此时，我们基于该树的空分支，把x 对应的位取反得到ID x’ ，那么距离x’ 最近的叶子即为距离x 最近的叶子。

    和Chord 中沿圆周顺时针方向的度量方法一样，XOR 也是单向的。对于任意给定的点x 和距离Δ>0 ，仅存在一个点y 使得d(x,y)= Δ。单向性保证了无论从哪个节点开始查询，都会沿着同样的路径进行收敛。因此，沿着查询路径对<key ，value> 对进行cache 就可以避免热点问题。基于XOR 的拓扑也是对称的（对于所有的x ，y ：d(x,y)=d(y,x) ），这一点和Pastry 一样，在Chord 中则不具有对称性。

### 2.2 节点状态

       Kademlia 节点中保存有路由查询消息所需要的相互联系信息。对于位于 [0,160) 中的每个 i ，每个节点都保存有一个 <IP 地址， UDP 端口，节点 ID> 三元组列表，三元组中的节点 ID 为那些和其距离位于 2 i 和 2 i+1 之间的节点的 ID 。我们称这些列表为 k-buckets 。每个 k-bucket 都按照节点最近联系时间排序——最近未联系的节点放在表头，最近联系的节点放在表尾。对于小的 i 值， k-buckets 一般都是空的（因为不存在没有合适节点）。对于大一些的 i 值，列表的长度可以扩展到 k ， k 是系统范围内定义的复制参数。 k 的值按照这样的方式给出：任意 1 小时内，任意给定的 k 个节点同时失效的可能性很小（比如， k=20 ）。

       当 Kademlia 节点收到任何来自其他节点的消息（请求或者回应）时，就会更新和消息发送者的节点 ID 对应的 k-bucket 。如果发送节点已经存在于接收节点的 k-bucket 中，接收节点就会把其移到列表的尾部。如果发送节点还没有出现在相应的 k-bucket 中，并且该 k-bucket 中的元素个数小于 k ，那么接收节点就把发送着放到列表的尾部。如果相应的 k-bucket 中的元素达到上限，此时，接收节点会 ping 该 k-bucekt 中最近最少联系的节点来决定该如何做。如果最近最少联系的节点没有响应，那么就把其从 k-bucket 中去除，然后把发送节点插入列表尾部。否则，如果收到了最近最少联系的节点的响应，那么就把其移到列表尾部，并丢弃掉发送者节点的联系信息。

       k-buckets 有效地实现了一种最近最少联系的置换策略，只是活动的节点永远不会被从列表中去除。这种对于老联系信息的偏爱来自于我们对 Saroiu 等人【 4 】所收集的 Gnutella 节点跟踪数据的分析。图 3 中展示了那些已经在线一段时间，还会继续在线 1 小时的 Gnutella 节点的百分比。节点已经在线的时间越长，再保持在线一小时的可能性就越高。通过保留那些最老的、活动的联系信息， k-bucekts 最大化了其所保存的节点依然在线的可能性。

       k-buckets 的另外一个好处是，其可以阻止某些 DoS 攻击。攻击者无法通过向系统中洪泛新节点的方法清空节点的路由状态。 Kademlia 节点只会在老节点离开系统时才插入新的节点。

### 2.3 Kademlia 协议

       Kademlia 协议有 4 个 RPCs 组成，分别为： PING ， STORE ， FIND\_NODE 以及 FIND\_VALUE 。 PING RPC 会对一个节点进行探测以确定其是否在线。 STORE 指示一个节点存储一个 <key ， value> 对，以用于以后的获取。

       FIND\_NODE 的参数是一个 160 位的 ID 。该 RPC 的接收者要返回它所知道的距离目标 ID 最近的 k 个节点的三元组 <IP 地址， UDP 端口，节点 ID> 列表。这些三元组可以来自一个 k-bucket ，也可以来自多个（最接近的那个 k-bucket 不满的话）。无论如何，该 RPC 的接收者都必须返回 k 个结果（除非所有的 k-buckets 中的个数加起来也不足 k 个，此时返回知道的所有节点）。

       FIND\_VALUE 和 FIND\_NODE 的行为相似，也是返回三元组 <IP 地址， UDP 端口，节点 ID> 列表，不过有一个例外。如果该 RPC 的接收者曾经收到过针对该 key 的 STORE RPC ，那么它仅返回所存储的值。

       对于所有的 RPCs ，接收者必须在回应中复制一个 160 位的随机 RPC ID ，这可以阻止一些地址欺骗行为。 RPC 的接收者可以把 PING 附带在 RPC 的回应中以进一步确认发送者网络地址的有效性。

       Kademlia 节点必须要做的最重要的事情是找出距离某个给定节点 ID 最近的 k 个节点。我们把这个过程叫做节点查询。 Kademlia 采用了一个递归算法进行节点查询。查询的发起者首先从最近的非空 k-bucket 中选出 a 个节点（如果该 bucket 中元素的个数小于 a ，那么就选出它所知道的 a 个最近的节点）。接着，发起者向选出的这 a 个节点同时发送异步的 FIND\_NODE RPC 。 a 是一个系统范围内的并发参数，比如， 3 。

       在递归步骤中，发起者向从前面 RPC 中学到的节点重新发送 FIND\_NODE 。（这个递归过程可以在前面发起的 a 个 RPCs 都返回前就开始）。从学到的距离目标最近的 k 个节点中，发起者从中选出 a 个还没有询问过的节点，并向它们重发 FIND\_NODE RPC[[1]](http://blog.csdn.net/hoping/article/details/5307320" \l "_ftn1) 。没有相应的节点则立即不予考虑，直到（除非）其响应为止。如果一轮 FIND\_NODES 下来没有发现更为接近的节点，发起者就向还没有查询过的 k 个最近节点重发 FIND\_NODE 。当发起者询问过它所知道的最近的 k 个节点，并且得到了它们的回应时，查询过程结束。当 a=1 时，在消息开销和故障节点的检测时延方面，该算法和 Chord 的接近。不过， Kademlia 可以做到低时延路由，因为它具有选择 k 个节点中的任何一个向其发送请求的灵活性。

       Kademlia 的绝大部分操作都是基于上述查询过程实现的。为了存储一个 <key ， value> 对，节点先找出距离 key 最近的 k 个节点，然后向它们发送 STORE RPCs 。此外，节点在必要时都会重新发布 <key ， value> 对，以保持数据可用（会在 2.5 小节中介绍）。该行为可以确保 <key ， value> 对以非常高的概率持续存在于系统中（我们会在概要证明中介绍）。针对 Kademlia 的当前应用（文件共享），我们同样要求 <key ， value> 的原始发布者每 24 小时就要重新发布一次。否则，在发布的 24 小时之后， <key ， value> 对就会过期，这样就限制了系统中过期索引信息的数量。对其他应用来说（比如数字认证或者用于值映射的加密 hash ），更长一点过期时间也许会更合适一些。

       要想找到一个 <key ， value> 对，节点首先要执行一个查询过程以找到距离 key 最近的 k 个节点。不过，在查找值时使用的是 FIND\_VALUE RPC 而不是 FIND\_NODE 。此外，当任何一个节点返回了 value 时，该过程就立即结束。出于 caching 的目的，一旦查询成功，发起请求的节点会把 <key ， value> 对存储在它所观察到的距离 key 最近但是没有返回 value 的节点上。

       由于拓扑结构的单向性，后面再次去查找该 key 时，很可能在去查询最接近的节点之前就在 cache 中找到。对于某个特别受欢迎的 key ，系统会在许多节点上对其进行 cache 。为了避免“过度 caching ”，我们把 <key ， value> 对在节点数据库中的过期时间设置成与当前节点和距离 key ID 最近的节点之间的节点个数成指数级反比关系 [[2]](http://blog.csdn.net/hoping/article/details/5307320" \l "_ftn2) 。虽然简单的 LRU 置换策略也能产生出类似的生存期分布，但是我们没有好的方法来选择 cache 的大小，因为节点无法预先知道系统将会存储多少值。

       一般来讲，节点之间往来的请求消息流就可以保持 buckets 最新。为了应对某些极端情况，比如：某个节点区间从来没有进行过查询请求，节点会对过去 1 小时内没有进行过节点查询的所有 kucket 进行更新操作。更新意味着从 bucket 中随机选择一个 ID ，然后执行一次针对该 ID 的节点查询。

       如果一个节点 u 想要加入 Kademlia 网络，它必须要与已经存在于网络中的某个节点 w 取得联系。 u 把 w 插入到相应的 k-bucket 中。接着， u 发起一次针对自己节点 ID 的查询。最后， u 更新比最近相邻节点远的所有 k-buckets 。在更新过程中， u 既填充了自己的 k-buckets ，同时也把自己加入到其他节点的 k-buckets 中。

### 2.4 路由表

       基于 Kademlia 的协议，我们可以很容易地定义出其基本的路由表结构，只是在某些高度不平衡的树中，需要处理一些微妙的情况。路由表是一颗二叉树，其叶子是 k-buckets 。每个 k-bucket 存放着具有某些公共 ID 前缀的节点。前缀就是该 k-bucket 在二叉树中的位置。了因此，每个 k-bucket 覆盖了 ID 空间的某个范围，所有 k-bucket 合起来完整覆盖了整个 160 位 ID 空间。

       节点是根据需要动态分配到路由树中的。图 4 说明了这个过程。一开始，节点 u 的路由树只有一个节点——覆盖整个 ID 空间的单个 k-bucket 。当 u 学到一个新节点的联系信息时，就会试图把其插入到相应的 k-bucket 中。如果该 bucket 不满，简单将其插入即可。否则，如果该 k-bucket 的区间范围包含了 u 自己的节点 ID ，那么该 bucket 会分裂为两个新的 buckets ，原有的内容会被划分到这两个 buckets 中，接着重复插入过程。如果 k-bucket 已满且不含有 u 的节点 ID ，那么就直接丢弃这个新的联系信息。

       当树高度不平衡时，就会出现一个复杂情况。假设节点 u 加入了一个系统，并且是系统那个中仅有的一个 ID 以 000 为前缀的节点。进一步假设系统中 ID 以 001 为前缀的节点个数超过 k 。每个具有前缀 001 的节点都会具有一个空的、应将 u 插入其中的 k-bucket ，但是 u 在对其 bucket 进行更新时只会通知到其中的 k 个节点。为了避免这种情况发生， Kademlia 节点会在一颗多于 k 个节点的子树中保存所有有效的联系星系，即使这会要求对不含有节点自己 ID 的 buckets 进行分割。图 5 说明了这些附加的分割过程。当 u 更新这些分割过的 buckets 时，所有具有 001 前缀的节点都会得到通知。

### 2.5 高效的 key 重新发布

       为了保证 key-value 对在系统中的持久性，节点必须对 key 进行周期性地重新发布。否则，有两种情况会导致对有效 key 的查询失败。首先，在发布时，最初获得 key-value 对的 k 个节点中的一些会离开网络。其次，新加入节点的 ID 相比 key-value 对的原始发布节点，可能距离该 key 更近一些。在这两种情况下，拥有 key-value 对的节点必须要对其进行重新发布，这样就再次保证了从距离该 key 最近的 k 个节点上可以获取该 key 。

       为了对节点离开造成的问题进行弥补， Kademlia 每一小时就对每个 key-value 对进行重新发布。该策略的一种幼稚的实现会导致很多的消息往来——存储 key-value 对的 k 个节点每小时都会先进行一次节点查询然后再进行 k-1 次 STORE RPC 调用。幸运的是，可以对这个重新发布过程进行深度的优化。首先，当节点收到针对某个 key-value 对的 STORE RPC 时，它可以认为该 RPC 也发给了其他 k-1 个最近节点，因此在下一个小时就不会重新发布该 key-value 对。这就保证了只要重新发布间隔不是精确同步的，对于任何一个 key-value 对来说，每小时只会有一个节点对其进行重新发布。

       第 2 个优化可以避免在重新发布 key 之前进行节点查询。如 2.4 小节中所述，为了应对不平衡树，节点在需要时可以分裂 k-buckets 以保证其具有关于一个节点个数超过 k 的边缘子树的全部知识。在重新发布 key-value 对之前，如果节点 u 更新了该子树中 k 个节点的所有 k-buckets ，那么它将自动获取了关于距离某个 key 值最近的 k 个节点的信息。对这些 bucket 的更新代价可以分摊到许多 key 的重新发布上面。

       要想知道为何在对规模大于 k 的子树中的 buckets 进行更新后，就无需再进行节点查询操作，就得考虑两种情况。如果要被重新发布的 key 位于该子树覆盖的 ID 区间内，那么由于该子树的规模至少为 k ，并且 u 具有关于该子树的全部知识，因此 u 一定知道距离该 key 最近的 k 个节点。另一方面，如果 key 位于子树范围之外，而 u 是距离该 key 最近的 k 个节点之一，那么按照 u 的 k-buckets 规则，所有距离该 key 比子树更近一些的区间中的元素都少于 k 。因此， u 将知道这些 k-buckets 中的所有节点，再加上关于子树的知识，就可以得到距离该 key 最近的 k 个节点。

       当一个新节点加入系统时，对于每个 key-vaule 对来说，如果该节点为其 k 个最近节点之一，那么必须对其进行存储。系统中原有的节点同样可以通过其边缘子树的完整知识，知道哪些 key-value 对需要存储在该新增节点上。每个了解到新节点的节点都会发起 STORE RPC 把相关的 key-value 对传送到新节点之上。为了避免重复的 STORE RPC ，只有那些自身 ID 比其他节点 ID 更接近 key 的节点才会进行 key-value 对的传送。

## 3 概要证明

       为了说明我们的系统是完全可用的，我们必须证明： 1 、绝大多数操作都可以在 [logn]+c 的时间内完成（ c 是一个小常量）； 2 、一个 <key ， value> 查询能够以极高的概率得到系统中的存储结果。

       我们先来给出一些定义。对于一个覆盖了距离区间 [2 i ， 2 i+1 ) 的 k-bucket 来说，称 i 为该 bucket 的索引。我们把节点的深度 h 定义为 160-i ，其中 i 为其非空 k-bucket 的最小索引。节点 y 在节点 x 中的 bucket 高度被定义为 y 将被插入到的 bucket 的索引减去 x 的最低有效空 bucket 的索引。因为节点 ID 是随机选择的，所以不太可能出现高度不均衡分布的情况。因此，对于一个具有 n 个节点的系统来说，每个节点的高度很可能都位于 logn 之内。此外，对于一个 ID 来说，第 k 个和其最接近的节点中和其最接近的节点的 bucket 高度很可能位于 logk 之内。

       接下来，我们将假定系统具有一个不变性：每个节点中的每个 k-bucket 都保存至少一个节点的联系信息，如果在相应的区间范围内存在有一个节点的话。有了这个假定，我就可以证明节点查询过程是正确的，并且仅花费对数时间。假设距离目标 ID 最近的节点的深度为 h 。如果该节点的 h 个最高有效 k-buckets 都不为空，那么查询过程将会在每一步中都找到一个近一半的节点（或者说距离短一位），因此在 h-logk 步找到目标节点。如果节点的 k-bucket 中有一个为空，那么目标节点有可能位于空 bucket 所覆盖的区间中。此时，最后的几步就不会折半递减。不过，搜索的过程继续正确进行，就好像 key 中和空 bucket 对应的位被反转过一样。因此，查询算法将总是在 h-logk 步内返回最近的节点。此外，一旦找到最近的节点，并发的粒度就从 a 变成 k 。找到剩余 k-1 个最近节点的步骤的数目就不会超过第 k 个最近节点中最接近节点的 bucket 高度，基本上不太可能大于 logk 加上一个常量。

       现在来证明系统不变性这个假设是正确的。首先，我们来看看当这个不变性成立时， bucket 更新造成的影响。在被更新后， bucket 要么包含有 k 个有效的节点，要么包含其覆盖区间内的所有节点（如果小于 k 个）。（可以根据节点查询过程得出这个结论）。新加入的节点也会被插入到不满的 buckets 中。因此，违反该不变性的唯一方法就是，某个 bucket 覆盖的区间内有 k+1 或者更多的节点，并且 bucket 中实际包含的 k 个节点全部故障，且在其间没有发生过任何查询或者更新操作。但是， k 正好是基于在一个小时（最大刷新时间）内同时故障的概率非常小这个概率选取的。

       实际上， k 个节点 在一个小时内失效的概率要远小于其 离开的概率，因为每个进出的请求都会更新节点的 buckets 。这个结果来自于 XOR 度量的对称性，因为在入或者出请求期间，和一个给定节点通信的节点的 ID 的分布和该节点的 bucket 区间范围完全一致。

       另外，即使该不变性在某个节点中的某个 bucket 中失效，也只会对查找时间造成影响（对某些查询来说，增加了一个 hop ），而不会对节点查询的正确性造成影响。要想使某个查询失败，查询路径上的 k 个节点必须在相同的 bucket 中都丢掉 k 个节点，并且其间没有任何查询或者更新操作。如果不同节点的 buckets 之间没有重叠，这件事发生的概率为 2 -k2 。否则，出现在多个节点 bucket 中的节点很可能具有更长一些的在线时间，因此失效的概率也更低一些。

       现在，我们来看看 <key ， value> 对的恢复问题。当 <key ， value> 对被发布后，会被存储在距其最近的 k 个节点上。并且每小时都会被重新发布。因为即使新增的节点（最不可靠）也具有 50% 的概率再在线一个小时，所以一小时后，该 <key ， value> 仍然存在于距其最近的 k 个节点中的一个上的概率为： 1-2 -k 。这个性质不会由于新增节点距离该 key 近而改变，因为只要这种节点一加入，就会联系距其最近的节点以填充其 buckets ，并收到任何它们应该存储的临近的 <key ， value> 对。当然，当距离某个 key 最近的 k 个节点都故障并且 <key ， value> 对也没有在别处 cache 的话， Kademlia 将无法存储该对，因此该 key 会丢失。

## 4 实现要点

       在本小节中，我们将介绍两个用来改进 Kademlia 实现性能的重要技术。

### 4.1 优化联系信息记录方法

       对 k-buckets 的最基本要求是能够完成 LRU 检查策略，并且能够在不丢失任何有效联系信息的情况下去除那些失效的联系信息。如 2.2 小节中所述，如果 k-bucket 已满，那么每当收到一条来自该 bucket 区间范围内的未知节点的消息时，都得发送一条 PING RPC 。这条 PING RPC 用来检查 k-bucket 中最近最少使用的联系信息是否仍然有效。如果已经失效，就用新的联系信息取代之。糟糕的是，该算法会导致大量的 PINGs 网络消息。

       为了减少这些流量， Kademlia 会延迟这个探测行为，直到需要向它们发送有用的信息。当 Kademlia 节点收到来自一个未知节点的 RPC ，并且针对该节点的 k-bucket 已经满时，该节点就把新的联系信息放在一个节点“置换 cache ”中（具有替换 k-bucket 中过期条目的资质）。当该节点再次查询该 k-bucket 中的节点时，没用响应的节点都会被清除，并由置换 cache 中的元素取代。置换 cache 是按照最近联系时间排序的，最近联系的节点具有最高的置换优先级。

       有一个和 Kademlia 使用 UDP 相关的问题，当网络丢包时，会丢失一些有效节点的响应信息。因为丢包通常意味着网络拥塞，所以 Kademlia 会锁定那些未能响应的节点，并在一个以指数级增加的退避时间区间内不向它们发送任何 RPCs 。因为在 Kademlia 查询过程中，大部分情况下只需联系到 k 个节点中的一个，所以一般情况下，系统不会向同一节点重传被丢弃的 RPCs 。

       如果对于连续的 5 条 RPCs ，节点都没有响应时，就被认为过期。如果此时 k-bucket 不满，或者其置换 cache 为空，那么 Kademlia 就只是把过期的联系信息打上标记，而不是把它们去除。这就保证了当节点自己的网络连接暂时出现问题时，不会完全清除其所有 k-buckets 。

### 4.2 加速查询

       实现中的另外一个优化技术为：通过增加路由表的规模来减少每个查询涉及的步数。从概念层面来讲，可以通过每次使用 ID 中的 b 位而不是一位来实现。如前面所介绍的，每个查询的期望步数为 log 2 n 。如果把路由表的规模扩大到 2 b log 2 b n 个 k-buckets ，那么就可以把期望的步数减至 log 2 b n 。

       2.4 小节中介绍了当 Kademlia 节点的 k-bucket 满且其区间包含了节点自己的 ID 时，如何去分裂该 k-bucket 。不过，在实现中，也会把不包含节点自己 ID 的区间分裂成 b-1 层。比如，如果 b=2 ，不包含节点 ID 的那一半 ID 空间会分裂一次（变成两个区间）；如果 b=3 ，会分裂成两层，最多四个区间，以此类推。大致的分裂规则为：如果一个满的 k-bucket 包含了节点自身的 ID 或者其在路由树中深度 d 满足 d ≡ 0(mob b) ，就会被分裂。在当前的实现中 b=5 。

       虽然基于 XOR 的路由算法和 Pastry 【 1 】、 Tapestry 【 2 】以及 Plaxton 分布式搜素算法【 3 】中第一阶段的路由算法很相似，但是当它们一般化到 b>1 时就都变得非常复杂。如果没有基于 XOR 的拓扑，就需要另外使用一个算法结构来从具有相同前缀但是后 b 位又不同的节点中找出目标节点。这三个算法采用了不同的方法来解决这个问题，各具缺点；除了大小为 O(2 b log 2 b n) 的主路由表外，它们都需要一个大小为 O(2 b ) 的二级路由表。这增加了启动和维护的成本，也使协议变得复杂，并且对于 Pastry 和 Tapestry 来说，也使得对其进行正确性和一致性方面的规范分析变得困难或不可能。有一个针对 Plaxton 的证明，但是该系统难以适应像 P2P 这样的高故障概率环境。

## 5 总结

       由于采用了基于 XOR 度量的拓扑结构， Kademlia 是第一个同时具有可证明的一致性和性能、最小延迟路由以及对称的、单向拓扑等优点的 P2P 系统。更进一步， Kademlia 还引入了一个并发参数， a ，可以让我们通过调整带宽中的一个常数因子来达成可以异步地选取具有最小时延的下一跳以及无延迟的错误恢复。最后， Kademlia 是第一个利用了节点失效概率和其在线时间反相关这个事实的 P2P 系统。

## 参考文献

[1] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. Accepted for Middleware, 2001, 2001. http://research.microsoft.com/˜antr/pastry/.   
[2] Ben Y. Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: an infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, U.C. Berkeley, April 2001.   
[3] Andr´ea W. Richa C. Greg Plaxton, Rajmohan Rajaraman. Accessing nearby copies of replicated objects in a distributed environment. In Proceedings of the ACM SPAA, pages 311–320, June 1997.   
[4] Stefan Saroiu, P. Krishna Gummadi and Steven D. Gribble. A Measurement Study of Peer-to-Peer File Sharing Systems. Technical Report UW-CSE-01-06-02, University of Washington, Department of Computer Science and Engineering, July 2001.   
[5] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In Proceedings of the ACM SIGCOMM ’01 Conference, San Diego, California, August 2001.

[[1]](http://blog.csdn.net/hoping/article/details/5307320" \l "_ftnref1) 可以在 bucket 元素以及 FIND 的回应中增加 round trip 时间信息，用来挑选这 a 个节点。

[[2]](http://blog.csdn.net/hoping/article/details/5307320" \l "_ftnref2) 该数目可以从当前节点的 bucket 结构中推导出来。