《网络存储技术》课程作业四

DHT 技术概述

付容天 学号 2020211616

班级 2020211310

计算机科学与技术系

计算机学院（国家示范性软件学院）

2022 年 11 月 8 日

目录

1 引言

分布式哈希表（Distributed Hash Table，DHT）是一种分布式系统，它提供类似于哈希表的查找服务：键值对存储在 DHT 中，任何参与节点都可以有效地检索与给定键关联的值。分布式哈希表的主要优点是可以在重新分配密钥的过程中以最少的工作量添加或删除节点。键是映射到特定值的唯一标识符，而这些值又可以是任何内容（从地址到文档、再到任意数据）。维护从键到值的映射的责任以分布式的方式由所有节点承担，从而使参与者集合中的更改导致最小量的中断。这允许分布式哈希表扩展到极大量的节点并处理连续的节点到达、离开和故障等情况。

分布式哈希表可以作为用于构建更复杂任务的基础，在分布式哈希表这一基础上，可以构造如任播（Anycast）、协作 Web 缓存、分布式文件系统（Distributed File System，DFS）、域名服务（Domain Name System，DNS）、即时消息传递、多播（Multicast），以及对等文件共享（Peer-to-peer File Sharing）和内容分发（Content Distribution）系统。使用分布式哈希表的著名分布式网络包括 BitTorrent 的分布式跟踪器、Kad 网络、Storm 网络、Tox 即时通讯工具、Freenet、YaCy 搜索引擎和星际文件系统（InterPlanetary File System）。

2 DHT 概要

2.1 CS 网络与 P2P 网络

CS 架构即 Client-Server 架构，由服务器和客户端两部分组成：服务器为服务的提供者，客户端则为服务的使用者。我们如今使用的很多应用程序例如网盘、视频应用、网购平台等都是 CS 架构，它的架构如下图所示：

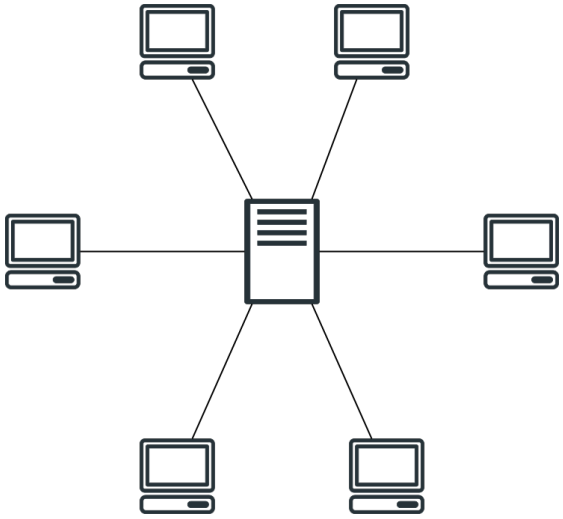


图 1: CS 架构概念图

通常来讲，服务器不是一个单点，而往往是一个集群，但本质上与单点没有显著差别。CS 架构的最大问题在于：一旦服务器因为某种原因关闭（例如宕机、被 DDoS 攻击等），那么客户端就无法使用，服务当然也就失效了，稳健性较差。

为了解决这个问题，人们提出了点到点网络（Peer-to-peer Networking，P2P Network）。在 P2P 网络中，不再由中心服务器提供服务，当然也就不再有“服务器”这一概念，取而代之的是每个人即使服务的提供者同时也是服务的使用者。BT 种子和磁力链接下载服务就是 P2P 架构的。显然，在 P2P 系统中，每个节点都是平等的，同时也是一个自我组织的系统，目的是在避免中心服务的网络环境中共享使用分布式资源。

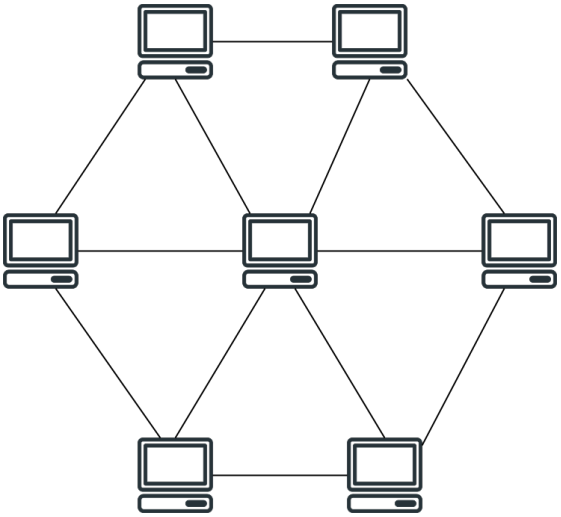


图 2: P2P 网络概念图 1

P2P 系统的架构如上图所示。由于去掉了中心服务器，P2P 系统的稳定性就会比较强：少数几个节点的失效几乎不会影响整个服务；节点多为用户提供，即使受到破坏，往往也不会导致整个系统服务崩溃。P2P 网络需要解决的一个最重要的问题就是：如何知道用户请求的资源位于哪个节点上。

在第一代 P2P 网络（典型代表是：第一个大型点到点内容交付系统 Napster）中，人们设置了一台中央服务器来管理资源所处的位置。每个节点加入网络的同时，会将他们所拥有的文件列表发送给服务器，这使得服务器可以进行搜索并将结果回传给进行查询的节点。也就是说，当一个用户想要发布资源时，他需要告诉中央服务器它发布的资源信息和自己的节点信息；而当其他用户请求资源的时候，需要先请求中央服务器以获取资源发布者的节点信息，再向资源发布者请求资源。

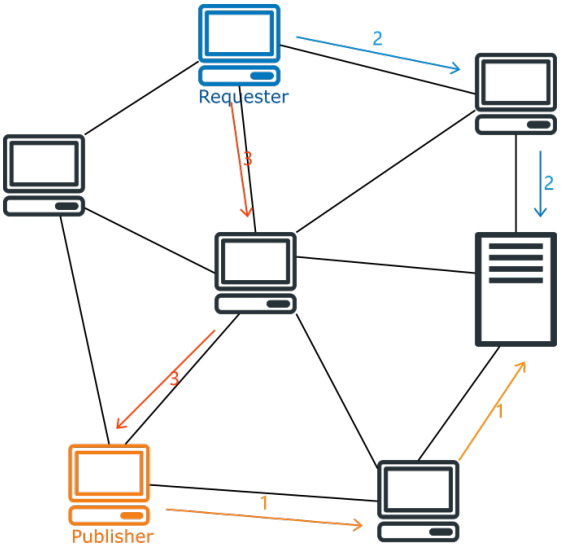


图 3: P2P 网络概念图 2

这种 P2P 网络的好处是效率高，只需要请求一次中央服务器就可以发布或获取资源。然而它的缺点也很明显：中央服务器是这个网络系统最脆弱的地方，它需要存储所有资源的信息，处理所有节点的请求，但是中央索引服务器的这一“系统管理”要求让整个系统易受攻击。一旦中央服务器失效，整个网络就无法使用。

早期的另外一种 P2P 网络（典型代表是：Gnutella）则采取了不同的策略，它不设置中央服务器，而是改用泛洪查询模式（Flooding Query Model）：每次搜索都会把查询消息广播给网络上的所有节点，也就是说，当用户请求资源时，用户会请求它所有的邻接节点，邻接节点再依次请求各自的邻接节点，并使用一些策略防止重复请求，直到找到拥有资源的节点。

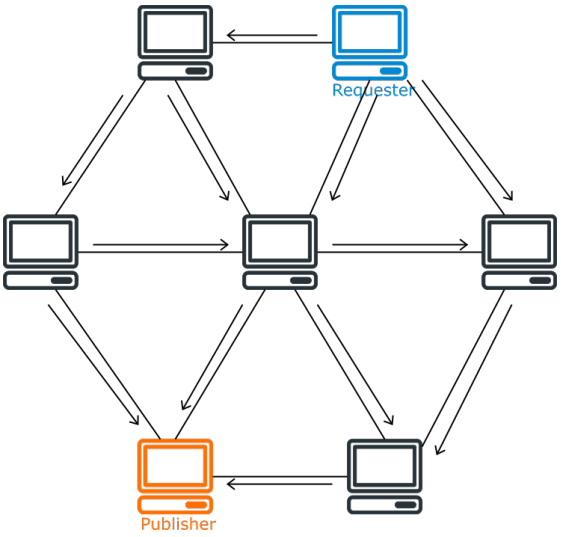


图 4: P2P 网络概念图 3

这种 P2P 网络去除了中央服务器，它的稳定性就强多了。但是，需要注意

的是，虽然这种方式能够防止单点故障（Single Point of Failure），但这种方案和 Napster 相比最主要的问题就是效率问题：它太慢了，一次查找可能会产生大量的请求，可能会有大量的节点卷入其中。一旦整个系统中的的节点过多，性能就会变得很差。所以，更高版本的 Gnutella 客户端为了解决这一问题，迁移到了动态查询模型（Dynamic Querying Model）上，从而大大提高了效率。

进一步，Freenet 则是完全分布式的，并采用了基于启发式密钥的路由，其中每个文件都与一个密钥相关联，具有相似密钥的文件倾向于聚集在一组相似的节点上。查询很可能通过网络路由到这样的集群，而无需访问许多对等点。但是，Freenet 不保证一定可以找到数据。

2.2 DHT 的必要性

从上面的讨论可以知道，在 P2P 系统中各种各样的问题制约着系统性能的发展，因而分布式哈希表的最初动机就是为了开发点对点系统（Peer-to-peer System，例如 Napster、Gnutella、BitTorrent 及 Freenet 等），这些 P2P 系统利用分布在 Internet 上的资源来提供单一有用的应用程序，而且利用增加的带宽和硬盘容量来提供文件共享服务。

分布式哈希表使用严格结构化的基于密钥的路由，以实现 Freenet 和 Gnutella 的去中心化，同时尽可能保证 Napster 的效率和结果查询的有效性。但是，DHT 一个缺点是：尽管 Freenet 的路由算法可以推广到任何可以定义紧密操作的密钥类型，但是 DHT 与 Freenet 一样仅直接支持精确匹配搜索，而不是关键字搜索。

2001 年，内容可寻址网络（Content Addressable Network，CAN）、Chord、Pastry 和 Tapestry 四个项目成为了 DHT 热门研究的开始。2002 年，一个名为弹性互联网系统基础设施（Iris）的项目获得了美国国家科学基金会 1200 万美元的资助。在学术界之外，DHT 技术已被用作 BitTorrent 和 Coral 内容分发网络的组成部分。

可见，DHT 正受到学术界和工业界越来越多相关方面的关注，并不断快速成长与成熟。

2.3 DHT 的属性与结构

DHT 的特点是以下四个属性：

- 自治（Autonomy）和去中心化（Decentralized）：节点共同组成系统，无需任何中央协调
- 容错（Fault Tolerance）：即使节点不断加入、离开和失败，系统也应该是可靠的（在某种意义上）
- 可扩展性（Scale）：即使有数千或数百万个节点，系统也应该有效地运行

用于实现这些目标的一项关键思想是：在一个有 n 个节点的分布式哈希表中，每个节点仅需存储 $O(\log n)$ 个其他节点，因此查找资源时仅需请求 $O(\log n)$ 个节点，并且无需中央服务器，因此是一个完全自组织的系统。分布式哈希表有很多中实现算法，包括：Apache Cassandra、BATON Overlay、Mainline DHT、Content addressable network（CAN）、Chord、Koorde、Kademlia、Pastry、P-Grid、Riak、Tapestry、TomP2P 和 Voldemort 等。我们先来看看它们共通的思想，简单来讲就是任何一个节点只需要与系统中的少数其他节点协调——最常见的是，n 个参与者中的 $O(\log n)$——因此只有有限数量的成员的每一次变化都需要开展工作。

此外，一些 DHT 设计旨在防止恶意参与者并允许参与者保持匿名，但这在许多点对点（尤其是文件共享）系统中并不常见。

2.3.1 地址管理

首先，在分布式哈希表中，每个节点和资源都有一个唯一标识（通常是一个 160 位整数）。为方便起见，我们称节点的唯一标识为 ID，称资源的唯一标识为 Key。我们可以把一个节点的 IP 地址用 SHA-1 散列哈希算法得到这个节点的 ID；同样地，把一个资源文件用 SHA-1 散列哈希算法就能得到这个资源的 Key 了。

定义好 ID 和 Key 之后，就可以发布和存储资源了。每个节点都会负责一段特定范围的 Key，其规则取决于具体的算法。例如，在 Chord 算法中，每个 Key 总是被第一个 ID 大于或等于它的节点负责。在发布资源的的时候，先通过哈希算法计算出资源文件的 Key，然后联系负责这个 Key 的节点，把资源存放在这个节点上。当有人请求资源的时候，自然就联系负责这个 Key 的节点，把资源取回即可。

发布和请求资源有两种做法：

- 一种是直接把文件传输给负责的节点，并由它存储文件资源，请求资源时再由这个节点将文件传输给请求者
- 另一种做法是由发布者自己设法存储资源，发布文件时把文件所在节点的地址传输给负责的节点，负责的节点仅存储一个地址，请求资源的时候会联系负责的节点获取资源文件的地址，然后再取回资源

这两种做法各有优劣：前者的好处是资源的发布者不必在线，请求者也能获取资源；坏处是如果文件过大，就会产生较大的传输和存储成本。后者的好处是传输和存储成本都比较小，但是资源的发布者，或者说资源文件所在的节点必须一直在线。

现在来看 Key 空间是如何进行分区的。大多数 DHT 使用一致性散列（Consistent Hashing）或集合散列（Rendezvous Hashing）的一些变体将 Key 映射到节点。这两种算法是独立并同时设计的，以解决分布式哈希表问题。一致性散列和集合散列都具有基本属性，即删除或添加一个节点只会更改具有相邻 ID 的节点所拥有的密钥集，而不会影响所有其他节点。将此与传统的哈希表进行对比：在

传统哈希表中，添加或删除一个内容会导致几乎整个 Key 空间被重新映射。由于所有权的任何更改通常对应于存储在 DHT 中的对象从一个节点到另一个节点的带宽密集型移动，因此需要最小化这种重组，从而有效地支持高流失率（节点到达和故障）。具体来说：

- 一致性散列（Consistent Hashing）：使用一个函数 $\delta(k_1, k_2)$ 定义了键之间距离的抽象概念 k_1 和 k_1，这与地理距离或网络延迟无关。每个节点都分配有一个称为其标识符（ID）的键。有 ID 的节点 i_x 拥有所有密钥 k_m 为此 i_x 是最接近的 ID（依据是 $\delta(k_m, i_x)$）。例如，Chord DHT 使用一致哈希，将节点视为圆上的点，并且 $\delta(k_1, k_2)$ 是绕圆顺时针方向从 k_1 到 k_2 行进的距离。因此，循环键空间被分成连续的段，其端点是节点标识符
- 集合散列（Rendezvous Hashing）：所有客户端使用相同的散列函数 h（预先选定），将密钥关联到 n 个可用服务器之一，每个客户端都有相同的标识符表 $\{S_1, S_2, \ldots, S_n\}$（每个服务器一个）。给定一些密钥 k，客户端计算 n 个哈希权重 $w_{1\cdots n} = h(S_{1\cdots n}, k)$，客户端将该密钥与对应该密钥哈希权重最高的服务器相关联。有 ID 的服务器所有密钥 k_m 哈希权重 $h(S_m, k_m)$ 高于该密钥的任何其他节点的哈希权重
- 保持位置的散列（Locality-preserving Hashing）：保持位置的散列确保将相似的键分配给相似的对象。这可以更有效地执行范围查询，但是，与使用一致散列相比，无法保证密钥（以及负载）均匀随机分布在密钥空间和参与的对等点上。DHT 协议如 Self-Chord 和 Oscar 解决了这些问题：Self-Chord 将对象键与对等 ID 分离，并使用基于群体智能（Swarm Intelligence）范式的统计方法沿环对键进行排序。排序可确保相邻节点存储相似的密钥，并确保发现过程，包括范围查询（Range Query），可以在对数时间内执行；Oscar 则构建了一个基于随机游走（Random Walk）采样的可导航小世界网络（Small World Network），同时确保对数搜索时间

2.3.2 路由算法

上面我们简述了地址系统，以及如何发布和取回资源。但是现在还有一个大问题：如何找到负责某个特定 Key 的节点呢？这里就要用到路由算法了，不同的分布式哈希表实现有不同的路由算法，但它们的思路是一致的。

首先每个节点会由若干个其他节点的联系方式（IP 地址和端口等），称之为路由表。一般来说一个有着 n 个节点的分布式哈希表中，一个节点的路由表的长度为 $O(\log n)$。每个节点都会按照特定的规则构建路由表，最终所有的节点会形成一张网络。从一个节点发出的消息会根据特定的路由规则，沿着网络逐步接近目标节点，最终达到目标节点。在有着 n 个节点的分布式哈希表中，这个过程的转发次数通常为 $O(\log n)$ 次。具体来讲，在实际应用中：

Max degree	Max route length	Used in	Note
$O(1)$	$O(n)$		最差的查找长度，查找时间可能要慢得多
$O(1)$	$O(\log n)$	Koorde (with constant degree)	实现起来更复杂，但可以通过固定数量的连接找到可接受的查找时间
$O(\log n)$	$O(\log n)$	Chord, Kademlia, Pastry, Tapestry	最常见但不是最佳的（度数/路线长度）。Chord 是最基本的版本，Kademlia 则是最流行的有改进的优化变体
$O(\log n)$	$O(\log n / \log\log(n))$	Koorde (with optimal lookup)	实现起来更复杂，但查找可能更快
$O(\sqrt{n})$	$O(1)$		最差的本地存储需求，在任何节点连接或断开连接后都需要大量通信

2.3.3 自组织

分布式哈希表中的节点都是由各个用户组成，随时有用户加入、离开或失效。并且分布式哈希表没有中央服务器，也就是说着这个系统完全没有管理者。这意味着分配地址、构建路由表、节点加入、节点离开、排除失效节点等操作都要靠自我组织策略实现。

要发布或获取资源，首先要有节点加入。一个节点加入通常有以下几步：首先，一个新节点需要通过一些外部机制联系分布式哈希表中的任意一个已有节点；然后新节点通过请求这个已有节点构造出自己的路由表，并且更新其他需要与其建立连接的节点的路由表；最后这个节点还需要取回它所负责的资源。

此外我们必须认为节点的失效是一件经常发生的事，必须能够正确处理它们。例如，在路由的过程中遇到失效的节点，会有能够替代它的其他节点来完成路由操作；会定期地检查路由表中的节点是否有效；将资源重复存储在多个节点上以对抗节点失效等。另外分布式哈希表中的节点都是自愿加入的，也可以自愿离开。节点离开的处理与节点失效类似，不过还可以做一些更多的操作，比如说立即更新其他节点的路由表，将自己的资源转储到其他节点等。

2.4 DHT 的安全性

由于 DHT 的去中心化、容错性和可扩展性，它们本质上比集中式系统更能抵御恶意攻击者。用于分布式数据存储的开放系统对大规模敌对攻击者更加具

有鲁棒性。精心设计的具有拜占庭容错能力（Byzantine Fault Tolerance）的 DHT 系统可以防御安全漏洞（称为 Sybil 攻击，它会影响大多数当前的 DHT 设计），例如，Whanau 是一种可以在一定程度上抵抗 Sybil 攻击的 DHT。但是，针对 Sybil 攻击的有效防御的研究通常被认为是一个悬而未决的问题。

2.5 DHT 的实际应用特点

在 DHT 实际应用的过程中，往往有以下的特点：

- 地址空间应当作为 DHT 的一个参数，这是因为一些 DHT 使用 128 位或 160 位密钥空间
- 一些 DHT 使用 SHA-1 以外的散列函数
- 在实际情况下，密钥 k 可以是文件内容的哈希而不是文件名的哈希，以提供内容可寻址存储（Content Addressable Storage），因此文件的重命名不会阻止用户找到它
- 一些 DHT 也可能发布不同类型的对象，例如密钥 k 可以是节点 ID，相关数据可以描述如何联系该节点。这允许发布存在信息，并且经常用于 IM 应用程序等。在最简单的情况下，ID 只是一个直接用作密钥 k 的随机数（因此在 160 位 DHT 中，ID 将是一个 160 位数字，通常是随机选择的）。在一些 DHT 中，节点 ID 的发布也用于优化 DHT 操作
- 可以添加冗余以提高可靠性，(k, data) 密钥对可以存储在多个与该密钥对应的节点中。通常，实际使用的 DHT 算法不是只选择一个节点，而是选择 i 个合适的节点（其中 i 是 DHT 的特定于实现的参数）。在一些 DHT 设计中，节点同意处理某个键空间范围，其大小可以动态选择，而不是硬编码
- 一些先进的 DHT（如 Kademlia）首先通过 DHT 执行迭代查找，以选择一组合适的节点并将 put(k, data) 消息仅发送到这些节点，从而大大减少无用流量，因为发布的消息仅发送到节点似乎适合存储密钥 k；并且迭代查找只覆盖一小部分节点而不是整个 DHT，从而减少了无用的转发。在此类 DHT 中，put(k, data) 消息的转发可能仅作为自愈算法的一部分发生：如果目标节点接收到 put(k, data) 消息（但认为 k 超出其处理范围并且已知更接近的节点（就 DHT 密钥空间而言）），则将消息转发到该节点。否则，数据将在本地建立索引。这导致了某种程度上的自我平衡 DHT 行为。当然，这样的算法需要节点在 DHT 中发布他们的存在数据，以便可以执行迭代查找
- 由于在大多数机器上发送消息比本地哈希表访问昂贵得多，因此将与特定节点有关的许多消息捆绑到一个批次中是有意义的。假设每个节点有一个本地批次，最多包含 b 个操作，捆绑过程如下。每个节点首先根据负责操作的节点的标识符对其本地批次进行排序。使用桶排序，这可以在 $O(b+n)$ 中完成，其中 n 是 DHT 中的节点数。当一个批次中有多个操作针对同一个键时，批次在被发送出去之前会被压缩。例如，同一键的多次查找可以减

少到一个或多个增量可以减少到单个添加操作。这种减少可以在临时本地哈希表的帮助下实现。最后，操作被发送到各个节点

- 每条文件索引被表示成一个 (K, V) 对，K 称为关键字，可以是文件名（或文件的其他描述信息）的哈希值，V 是实际存储文件的节点的 IP 地址（或节点的其他描述信息）。所有的文件索引条目（即所有的 (K, V) 对）组成一张大的文件索引哈希表，只要输入目标文件的 K 值，就可以从这张表中查出所有存储该文件的节点地址。然后，再将上面的大文件哈希表分割成很多局部小块，按照特定的规则把这些小块的局部哈希表分布到系统中的所有参与节点上，使得每个节点负责维护其中的一块。这样，节点查询文件时，只要把查询报文路由到相应的节点即可（该节点维护的哈希表分块中含有要查找的 (K, V) 对）

3 DHT 实现技术介绍

3.1 Chord

Chord 是一种点对点分布式哈希表的协议和算法。分布式哈希表通过将键分配给不同的计算机（称为“节点”）来存储键值对；一个节点将存储它负责的所有键的值。Chord 指定如何将键分配给节点，以及节点如何通过首先定位负责该键的节点来发现给定键的值。

Chord 是与 CAN、Tapestry 和 Pastry 一起的四种原始分布式哈希表协议之一。它由 Ion Stoica、Robert Morris、David Karger、Frans Kaashoek 和 Hari Balakrishnan 于 2001 年推出，并由 MIT 开发。2001 年发表的 Chord 论文在 2011 年获得 ACM SIGCOMM 时间测试奖。

3.1.1 Chord 概述

节点和键被分配一个 m 位标识符使用一致的散列。SHA-1 算散列法是一致性哈希的基本哈希函数。一致的散列对于 Chord 的稳健性和性能是不可或缺的，因为键和节点（实际上是它们的 IP 地址）均匀分布在相同的标识符空间中，故冲突的可能性可以忽略不计。因此，它还允许节点加入和离开网络而不会中断。在协议中，术语节点用于指代节点本身及其标识符（ID），没有歧义。

使用 Chord 查找协议，节点和键被排列在一个标识符圈中，最多有 2^m 个节点，范围从 0 至 $2^m - 1$（m 应该足够大以避免冲突），其中一些节点将映射到机器或键，而其他（大多数）将是空的。每个节点都有一个后继者和前任者。节点的后继节点是标识符圈中顺时针方向的下一个节点。前身是逆时针。如果每个可能的 ID 都有一个节点，则节点 0 的后继是节点 1，节点 0 的前驱是节点 $2^m - 1$。但是，通常序列中存在“漏洞”，例如，节点 153 的后继节点可能是节点 167（这是因为从 154 到 166 的节点不存在），在这种情况下，节点 167 的前任将是节点 153。后继者的概念也可以用于键。

3.1.2 Chord 地址管理

如前所述，Chord 使用一个 m 位整数作为节点和资源的唯一标识，也就是说标识的取值范围为 0 至 $2^m - 1$ 的整数。Chord 把所有的 ID 排列成一个环，从小到大顺时针排列，首尾相连。为了方便起见，我们不妨称每个 ID 都先于它逆时针方向的 ID 且后于它顺时针方向的 ID。每个节点都能在这个环中找到自己的位置，而对于 Key 值为 k 的资源来说，它总是会被第一个 ID 先于或等于 k 的节点负责。我们把负责 k 的节点称为 k 的后继，记作 successor(k)。例如，如下图所示，在一个 $m = 4$ 的 Chord 环中，有 ID 分别为 $\{0, 1, 4, 8, 11, 14\}$ 的六个节点。Key 为 1 的资源被 ID 为 1 的节点负责，Key 为 5 的资源被 ID 为 8 的节点负责，Key 为 15 的资源被 ID 为 0 的节点负责。也就是有：successor$(1) = 1$，successor$(5) = 8$，successor$(15) = 0$。

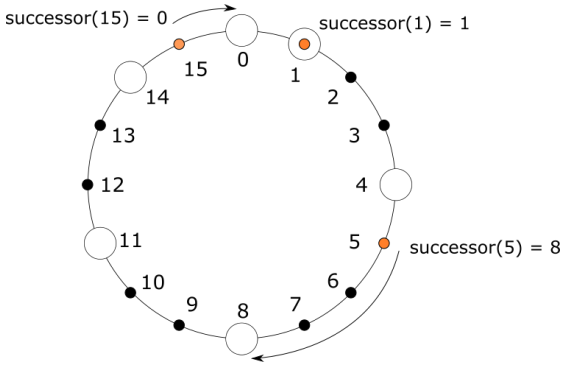
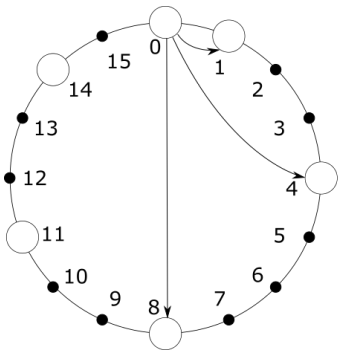


图 5: Chord 地址管理概念图

3.1.3 Chord 路由算法

在 Chord 算法中，为了更好地进行路由操作，每个节点都保存一个长度为 m 的路由表，存储至多 m 个其他节点的信息，这些信息能让我们联系到这些节点。假设一个节点的 ID 为 n（以下简称节点 n），那么它的路由表中第 i 个节点应为第一个 ID 先于或等于 $n + 2^{i-1}$ 的节点，即 successor$(n + 2^{i-1})$，此处 $1 \leq i \leq m$。我们把 n 的路由表中的第 i 个节点记作 $n.finger[i]$。显然，路由表中的第一个节点为顺时针方向紧挨着 n 的节点，我们通常称它为节点 n 的后继节点，记作 n.successor。



i	$n+2^{i-1}$	successor$(n+2^{i-1})$
1	1	1
2	2	4
3	4	4
4	8	8

图 6: Chord 路由算法概念图 1

有了这个路由表，我们就可以快速地寻址了。假设一个节点 n 需要寻找 Key 值为 k 的资源所在的节点，即寻找 successor(k)，它首先判断 k 是否落在区间 $(n, n.\text{successor}]$ 内：如果是，则说明这个 Key 由它的后继负责；否则 n 从后向前遍历自己的路由表，直到找到一个 ID 后于 k 的节点，然后把 k 传递给这个节点由它执行上述查找工作，直到找到为止。

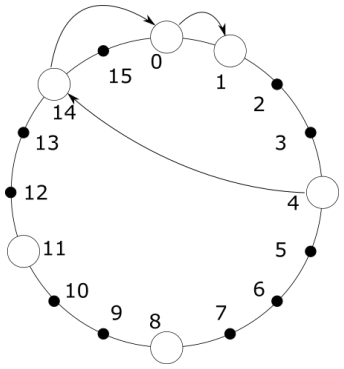


图 7: Chord 路由算法概念图 2

上图展示了从节点 4 寻找节点 1 的过程。节点 4 的路由表中的节点分别为 $\{8, 11, 14\}$，它首先会从后向前找到路由表中第一个后于 1 的节点为 14，然后就请求 14 帮忙寻找 1，节点 14 的路由表分别为 $\{0, 4, 8\}$，同样地，14 就会请求节点 0 帮忙寻找 1，最终节点 0 找到它的后继即为节点 1。

路由表的设计使得每个节点存储的信息相对总节点数量来说仍然只是很少的一部分。不过，这种设计并不保证任意一个 Key k 都能被节点从自己的路由表中找到。在绝大多数情况下，Key lookup 仍需要几次跳跃（hop）来定位所对应的节点。可以看到，整个查找过程是逐步逼近目标目标节点的：离目标节点越远，跳跃的距离就越长；离目标节点越近，跳跃的距离就越短、越精确。若整个系统有 N 个节点，查找进行的跳跃次数就为 $O(\log N)$。

3.1.4 Chord 自组织

首先来看节点加入问题。

一个节点要想加入 Chord 并不难。我们已经说明了：Chord 系统中的任意一个节点都能对任意 Key k 找到它的后继 successor(k)。首先，新节点 n 使用一些算法生成自己的 ID，然后它需要联系系统中的任意一个节点 n'，让他帮忙寻找 successor(n)。显然，successor(n) 即是 n 的后继节点，同时也是它路由表中的第一个节点。接下来它再请求 n' 帮忙分别寻找 successor($n + 2^{1,2,\cdots}$) 等，直到构建完自己的路由表。

构建完自己的路由表后，n 基本上就可以算是加入 Chord 了，不过这时其他的节点还不知晓它的存在。n 加入后，有些节点的路由表中本该指向 n 的指针却仍指向 n 的后继，这个时候就需要提醒这些节点更新路由表。为了得到哪些节点需要更新路由表，我们只需把操作反过来，即对所有的 $1 \le i \le m$，找到第一个

后于 $n-2i-1$ 的节点，这与寻找后继相同，不再赘述了。

最后，新节点 n 还需要取回它负责的所有资源。n 只需联系它的后继取回它后继所拥有的所有 Key 后于（注意！和“大于”不同!）或等于 n 的资源即可。

然后来看处理并发问题。

考虑多个节点同时加入 Chord，如果使用上述方法，就有可能导致路由表不准确。为此，我们不能在新节点加入的时候一次性更新所有节点的路由表。Chord 算法非常重要的一点就是保证每个节点的后继节点都是准确的。为了保证这一点，我们希望每个节点都能和它的后继节点双向通信、彼此检查、因此我们给每个节点添加一个属性 n.predecessor 指向它的前序节点。然后，我们让每个节点定期执行一个操作，称之为 stabilize。在 stabilize 操作中，每个节点 n 都会向自己的后继节点 s 请求获取 s 的前序节点 x，然后 n 会检查 x 是否更适合当它的后继，如果是，n 就它自己的后继更新为 x，同时 n 告诉自己的后继 s 自己的存在，s 又会检查 n 是否更适合当它的前序，如果是，s 就把自己的前序更新为 n。这个操作足够简单，又能够保证 Chord 环的准确性。

当新节点 n 加入 Chord 时，首先联系已有节点 n' 获取到 n.successor。n 除了设置好自己的后继之外，什么都不会做。此时 n 还没有加入 Chord 环，这要等到 stabilize 执行之后：当 n 执行 stabilize 时，会通知它的后继更新前序为 n；当 n 真正的前序 p 执行 stabilize 时会把自己的后继修正为 n，并通知 n 设置前序为 p。这样 n 就加入到 Chord 环中了。这样的操作在多个节点同时加入时也是正确有效的。

此外，每个节点 n 还会定期修复自己的路由表，以确保能指向正确的节点。具体的做法是随机选取路由表中的第 i 个节点，查找并更新 $n.finger[i]$ 为 successor$(n+2^{i-1})$。由于 n 的后继节点始终是正确的，所以这个查找操作始终是有效的。

最后考察节点失效与离开问题。

一旦有节点失效，势必会造成某个节点的后继节点失效。而后继节点的准确性对 Chord 来说至关重要，它的失效意味着 Chord 环断裂，可能导致查找失效，stabilize 操作无法进行。为了解决这个问题，一个节点通常会维护多个后继节点，形成一个后继列表，它的长度通常是 m。这样的话，当一个节点的后继节点失效后，它会在后继列表中寻找下一个替代的节点。此外一个节点的失效意味着这个节点上资源的丢失，因此一个资源除了存储在负责它的节点 n 上之外，还会被重复地存储在 n 的若干个后继节点上。

节点离开的处理与节点失效相似，不同的是节点离开时可以做一些额外的操作，例如通知它周围的节点立即执行 stabilize 操作，把资源转移到它的后继等。

3.2 Kademlia

Kademlia 是由 Petar Maymounkov 和 David Mazières 在 2002 年设计的分布式对等计算机网络的分布式哈希表，它指定了网络的结构和通过节点查找进行的信息交换。Kademlia 节点之间使用 UDP 进行通信。参与节点形成一个虚拟或覆盖网络。每个节点都由一个数字或节点 ID 标识。节点 ID 不仅用作标识，而且 Kademlia 算法还使用节点 ID 来定位值（通常是文件哈希或关键字）。

为了查找与给定键关联的值，该算法分几个步骤探索网络。每一步都会找到离键更近的节点，直到被联系的节点返回值或没有找到更近的节点，这非常有效。与许多其他 DHT 一样，Kademlia 仅需要 $O(\log n)$ 复杂度即可搜索节点总数为 n 的系统中的节点。

Kademlia 还增加了对拒绝服务攻击（Denial-of-service Attack）的抵抗力。即使一整套节点被淹没，这对网络可用性的影响也有限，因为网络将通过围绕这些“漏洞”编织网络来恢复自身。匿名网络层（Invisible Internet Project，I2P）还对 Kademlia 的实施进行了修改，以缓解 Kademlia 的漏洞（例如 Sybil 攻击）。

在 Kademlia 中一共有四种消息：

- PING：用来测试节点是否仍然在线
- STORE：在某个节点中存储一个键值对
- FIND_NODE：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：与 FIND_NODE 一样，但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

3.2.1 Kademlia 地址管理

Kademlia 同样使用 m 位整数作为节点和资源的唯一标识，但与 Chord 中的“区间负责制”不同，Kademlia 中的资源都是被离它最近的节点负责。出于容错考虑，每个资源通常都被距离它最近的 k 个节点负责，这里 k 是一个常量，通常取 k 使得在系统中任意 k 个节点都不太可能在一小时之内同时失效，比如取 $k = 20$。有趣的是，这里的“距离”并不是数值之差，而是通过异或运算得出的。在 Kademlia 中，每个节点都可以看作一颗高度为 $m + 1$ 的二叉树上的叶子节点。把 ID 二进制展开，从最高位开始、自根节点逢 1 向左逢 0 向右、直到抵达叶子节点，如下图所示。

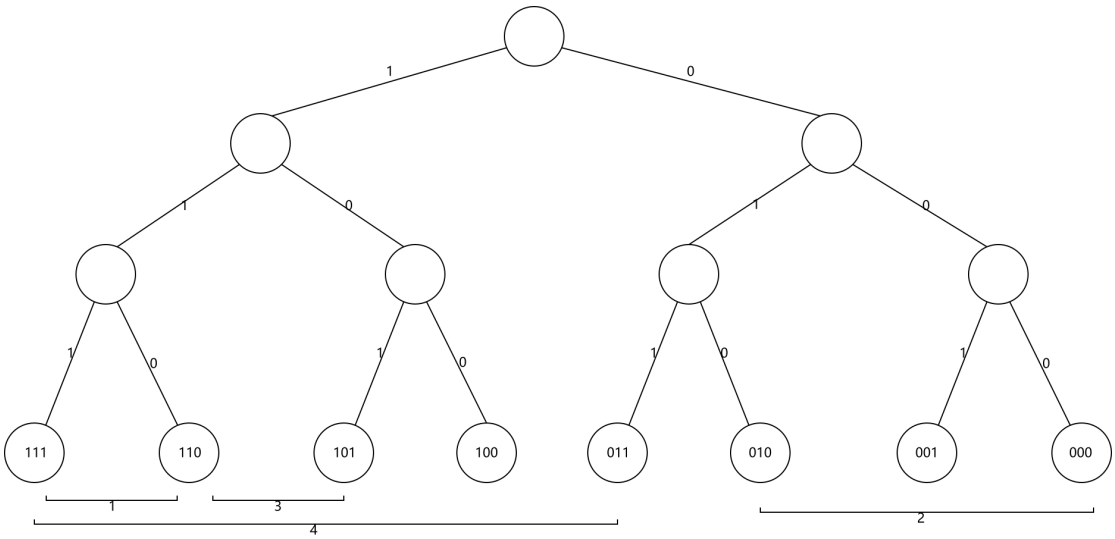


图 8: Kademlia 地址管理概念图

Kademlia 的巧妙之处就是定义两个 ID x 和 y 之间的距离为 $x \oplus y$。异或运算的特点是异为真同为假，如果两个 ID 高位相异低位相同，它们异或的结果就大；如果它们高位相同低位相异，异或的结果就小。这与二叉树中叶子的位置分布是一致的：如果两个节点共有的祖先节点少（高位相异），它们的距离就远；反之，如果共有的祖先节点多（高位相同），它们的距离就近。上图中也标注了一些节点之间的距离。

异或运算的另一个重要性质是：异或的逆运算仍是异或，即：如果有 $x \oplus y = d$ 则 $x \oplus d = y$，这就意味着对于每个节点，给的一个距离 d，至多有一个与其距离为 d 节点。这样一来 Kademlia 的拓扑结构便是单向的（unidirectional）。单向性很重要，因为单向性可以确保不管查找从哪个节点开始，同一 Key 的所有查找都会沿着同一路径收敛。

3.2.2 Kademlia 路由算法

对于任意一个给定节点，我们将二叉树从根节点开始不断向下分成一系列不包含该节点的子树。最高的子树由不包含该节点的二叉树的一半组成，下一个子树又由不包含该节点的剩余树的一半组成，以此类推。如果这个二叉树的高度为 $m+1$，我们最终会得到 m 个子树。接着在每个子树中任取 k 个节点，形成 m 个 k 桶（k-bucket），这 m 个 k 桶就是 Kademlia 节点的路由表。我们定义最小子树中取得的节点为第 0 个 k 桶，次小的子树中取得的节点为第 1 个 k 桶，以此类推。不难看出，对于每个 $0 \leq i < m$，第 i 个 k 桶中节点与当前节点的距离总是在区间 $[2^i, 2^{i+1})$ 之内。下图展示了 $m = 3$ 且 $k = 2$ 时节点 101 的 k 桶。

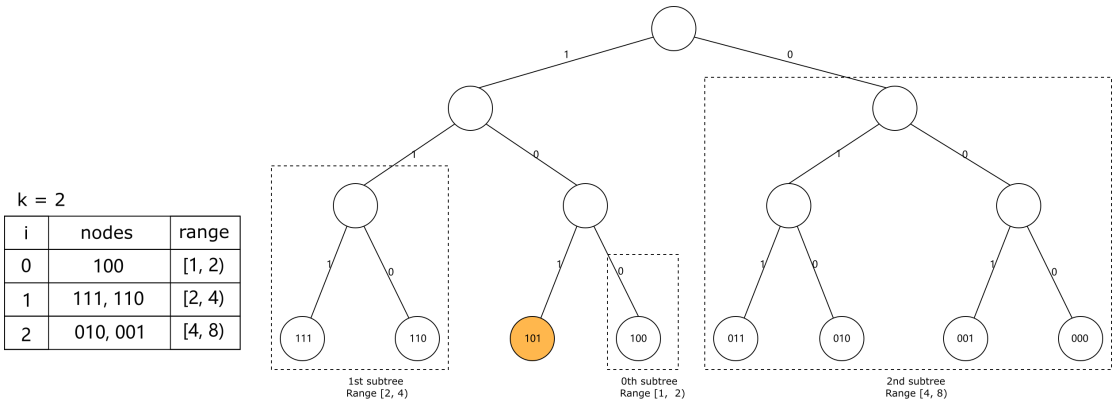


图 9: k 桶概念图 1

Kademlia 中每个节点都有一个基础操作，称为 FIND_NODE 操作。FIND_NODE 接受一个 Key 作为参数，返回当前节点所知道的 k 个距离这个 Key 最近的节点。基于 k 桶，找到这 k 个最近的节点很容易：先求出这个 Key 与当前节点的距离 d。因为第 i 个 k 桶中节点与当前节点的距离总是在区间 $[2^i, 2^{i+1})$ 之内，且这些区间都不会互相重叠，那么显然 d 落在的区间所属的 k 桶中的节点就是距离这个 Key 最近的节点。如果这个 k 桶中的节点不足 k 个，则在后一个 k 桶中取节点补充，如果还不够就再在后一个 k 桶中取。如果这个节点所有的 k 桶中的节点数之和都不足 k 个，就返回它所知道的所有节点。

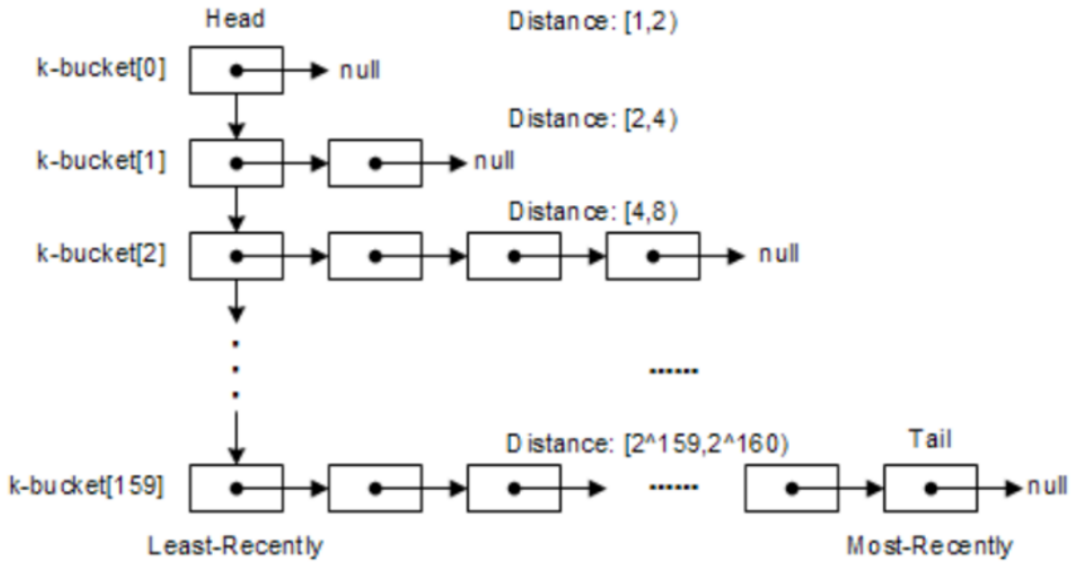


图 10: k 桶概念图 2

有了 FIND_NODE 操作，我们就可以定义 Kademlia 中最重要的一个过程：节点查找（node lookup）。节点查找要做的是给定一个 Key，找出整个系统中距离它最近的 k 个节点。这是一个递归过程：首先初始节点调用自己的 FIND_NODE，找到 k 个它所知的距离 Key 最近的节点。接下来我们在这 k 个节点中取 α 个最

近的节点，同时请求它们为 Key 执行 FIND_NODE。这里的 α 也是一个常量，作用是同时请求提高效率，常见的选择是取 $\alpha = 3$。

在接下来的递归过程中，初始节点每次都在上一次请求后返回的节点中取 α 个最近的，并且未被请求过的节点，然后请求它们为 Key 执行 FIND_NODE，以此类推。每执行一次，返回的节点就距离目标近一点。如果某次请求返回的节点不比上次请求返回的节点距目标 Key 近，就向所有未请求过的节点请求执行 FIND_NODE。如果还不能获取更近的节点，过程就终止。这时我们在其中取 k 个距离 Key 最近的节点，就是节点查找的结果。

Kademlia 的大多数操作都基于节点查找。发布资源时，只需为资源的 Key 执行节点查找，从而获取 k 个距离资源最近的节点，把资源存储在这些节点上。获取资源也类似，也只需为目标资源的 Key 执行节点查找，不同的是一旦遇到拥有目标资源的节点就停止查找。此外，一旦一个节点查找成功，它就会把资源缓存在离自己最近的节点上。因为 Kademlia 的拓扑结构是单向的，其他离目标资源比自己远的节点在查找时就很可能会经由缓存的节点，这样就能提前终止查找，提高了查找效率。

总结来说，Kademlia 查找节点过程一共有四步：

1. 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
2. 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
3. 发起者在收到这些返回信息之后，更新自己的结果列表，再次从自己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
4. 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

而 Kademlia 定位指定资源时，也是类似的：

1. 首先发起者会查找自己是否存储了 $<$ key, value $>$ 数据对，如果存在则直接返回，否则就返回 k 个距离 Key 值最近的节点，并向这 k 个节点 ID 发起 FIND_VALUE 请求
2. 收到 FIND_VALUE 请求的节点，首先也是检查自己是否存储了 $<$ key, value $>$ 数据对，如果有直接返回 value，如果没有，则在自己的对应的 k 桶中返回 k 个距离 Key 值最近的节点
3. 发起者如果收到 value 则结束查询过程，否则发起者在收到这些节点后，更新自己的结果列表，并再次从其中 k 个距离 Key 值最近的节点，挑选未发送请求的节点再次发起 FIND_VALUE 请求
4. 上述步骤不断重复，直到获取到 value 或者无法获取比发起者当前已知的 k 个节点更接近 Key 值的活动节点为止，这时就表示未找到 value 值

3.2.3 Kademlia 自组织

先来看 k 桶的维护问题，k 桶的维护分为三个问题：

- 主动收集节点：任何节点都可以发起 FIND_NODE 请求，从而刷新 k 桶中的节点信息
- 被动收集节点：当收到其它节点发来的请求（FIND_NODE、FIND_VALUE），会将对方的节点 ID 加入到某 k 桶中
- 检测失效节点：通过发起 PING 请求，判断 K 桶中的某个节点是否在线，然后清除失效的节点

所有的 k 桶都遵循最近最少使用（Least Recently Used，LRU）淘汰法。最近最少活跃的节点排在 k 桶的头部，最近最多活跃的节点排在 k 桶尾部。当一个 Kademlia 节点接收到任何来自其他节点的消息（请求或响应）的时候，都会尝试更新相应的 k 桶。如果这个节点在接收者对应的 k 桶中，接收者就会把它移动到 k 桶的尾部。如果这个节点不在相应的 k 桶中，并且 k 桶的节点小于 k 个，那么接收者就会直接把这个节点插入到这个 k 桶的尾部。如果相应的 k 桶是满的，接收者就会尝试 PING 这个 k 桶中最近最少活跃的节点。如果最近最少活跃的节点失效了，那么就移除它并且将新节点插入到 k 桶的尾部；否则就把最近最少活跃的节点移动到 k 桶尾部，并丢弃新节点。

通过这个机制就能在的通信的同时刷新 k 桶。为了防止某些范围的 Key 不易被查找的情况，每个节点都会手动刷新在上一小时未执行查找的 k 桶，做法是在这个 k 桶中随机选一个 Key 执行节点查找。

再来看新节点的加入问题，新节点加入需要以下步骤：

1. 新节点 A 首先需要一个种子节点 B 作为引导，并把该种子节点加入到对应的 k 桶中
2. 首先生成一个随机的节点 ID 值，直到离开网络，该节点会一直使用该 ID
3. 向节点 B 发起 FIND_NODE 请求，请求定位的节点时自己的节点 ID
4. 节点 B 在收到节点 A 的 FIND_NODE 请求后，会根据 FIND_NODE 请求的约定，找到 k 个距离 A 最近的节点，并返回给 A 节点
5. A 收到这些节点以后，就把它们加入到自己的 k 桶中
6. 然后节点 A 会继续向这些刚拿到节点发起 FIND_NODE 请求，如此往复，直到 A 建立了足够详细的路由表

具体来讲，如果一个新节点 n 要想加入 Kademlia，它首先使用一些算法生成自己的 ID，然后它需要通过一些外部手段获取到系统中的任意一个节点 n'，把 n' 加入到合适的 k 桶中。然后对自己的 ID 执行一次节点查找。根据上述的 k 桶维

护机制，在查找的过程中新节点 n 就能自动构建好自己的 k 桶，同时把自己插入到其他合适节点的 k 桶中，而无需其他操作。

节点加入时除了构建 k 桶之外，还应该取回这个节点应负责的资源。Kademlia 的做法是每隔一段时间（例如一个小时），所有的节点都对其拥有的资源执行一次发布操作。此外每隔一段时间（例如 24 小时）节点就会丢弃这段时间内未收到发布消息的资源。这样新节点就能收到自己须负责的资源，同时资源总能保持被 k 个距离它最近的节点负责。

如果每个小时所有节点都重发它们所拥有的资源，就有些浪费了。为了优化这一点，当一个节点收到一个资源的发布消息，它就不会在下一个小时重发它。因为当一个节点收到一个资源的发布消息，它就可以认为有 $k-1$ 个其他节点也收到了这个资源的发布消息。只要节点重发资源的节奏不一致，这就能保证每个资源都始终只有一个节点在重发。

最后考察节点失效与离开问题。

有了上述 k 桶维护和资源重发机制，我们不需要为节点的失效和离开做任何其它的工作。这也是 Kademlia 算法的巧妙之处，它的容错性要高于其他分布式哈希表算法。

3.2.4 Kademlia 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。

4 结语

随着互联网系统日益复杂，分布式概念变得越来越重要，分布式哈希表（Distributed Hash Table，DHT）更是其中一个非常重要的分布式概念，DHT 提供类似于哈希表的查找服务。DHT 技术有效地提高了分布式数据服务的性能、增加了系统的鲁棒性，这使其实际性能表现超出了我们的预期！

在本文中，我简要梳理了 DHT 技术的一些常见内容，感觉收获颇丰，同时又惊叹于计算机工业界实践成果之富，这值得我们持续不断地学习与探索！