现代计算机网络

李伟明 lwm@hust.edu.cn

Ch4. P2P原理与技术

4.1 P2P网络基本概念

4.2 混合式P2P网络（第一代）

4.3 无结构P2P网络（第二代）

4.4 结构化P2P网络（第三代）

4.4 结构化P2P网络（第三代）

- P2P网络拓扑演进背景
  - 1999，混合式
  - 2000，无结构
  - 2001，结构化
- 2001年后学术届开始关注
  - IEEE成立P2P专业协会
  - SIGCOMM发表高层次论文（Chord和CAN）
  - 国际会议/刊物发表论文
- 提出第三代模型
  - Chord/CAN/Tapestry
  - Pastry/CFS/PAST

4.4 结构化P2P网络（第三代）

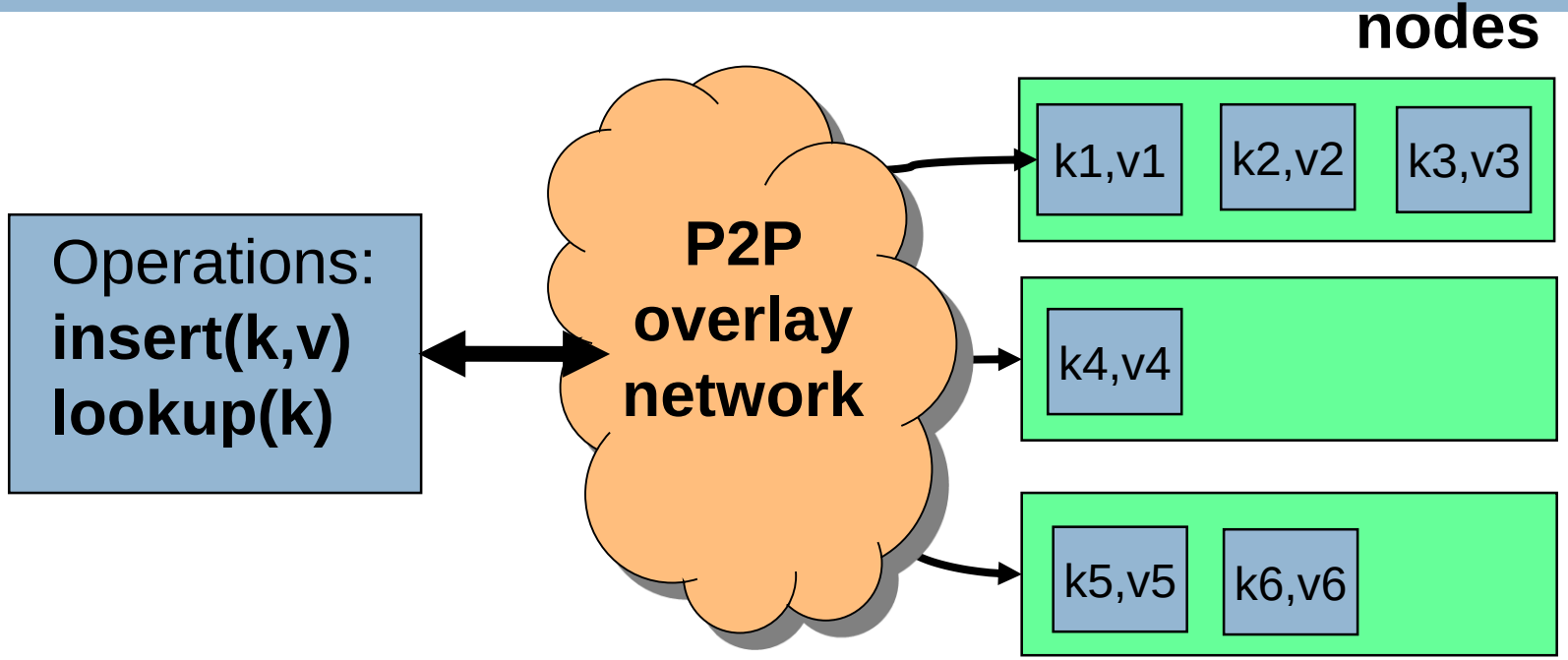
- 成立专门研究机构
  - MIT的Chord和CFS
  - UC Berkeley的Tapestry和OceanStore
  - 微软和Rice大学的Pastry和PAST
  - Stanford的Peers研究组
- 商业领域大发展
  - 基于异或度的Kademlia网络被BT/eDonkey/eMule所使用
  - Azureus/eXeem等下载工具
  - eMule、BitTorrent到了今天是典型结构化P2P网络
  - eMule0.42开始使用Kademlia协议

4.4.1 分布式哈希表结构

所谓结构化P2P，核心是采用DHT作为P2P节点和资源的组织方式：

- Distributed Hash Table
  - 普通的Hash Table中key和value保存在一台主机上，而DHT把key和value保存在分散的节点上，并通过Hash Table方法进行插入和查询。
- DHT的特点
  - 能自适应结点的动态加入/退出
  - 有着良好的可扩展性、鲁棒性
  - 结点ID分配的均匀性和自组织能力。
  - 确定性拓扑结构可精确发现

Distributed Hash Tables (DHT)



- p2p overlay maps keys to nodes
- completely decentralized and self-organizing

Distributed Hash Table 历史

- DHT：2000-2001年出现，学术界提出
- 动机
  - 这些不成熟的P2P应用竟然如此流行，“我们可以做得更好”
  - 保证系统中的文件能够被找到
  - 保证搜索时间在算法保证的范围
  - 保证数百万节点的可伸缩性
- 成为研究热点

- DHT的起源：
  - 起源于SDDS（Scalable Distribute Data Structures）研究，Gribble等实现了一个高度可扩展，容错的SDDS集群。
- DHT的特点：

1）一致性hash

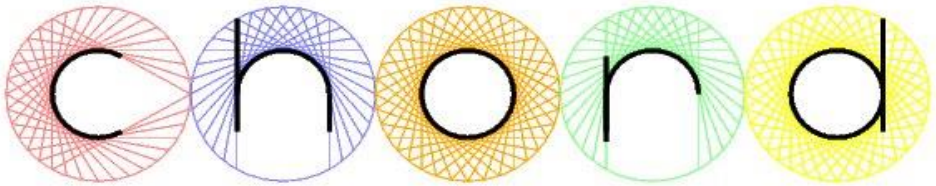
  - The consistent hash function assigns each node and resource an mbit identifier using a base hash function such as SHA-1.
  - 假设资源以（ObjectID，Info）方式保存，每个节点有NodeID，那么NodeID和ObjectID在同一空间
  - 所以一致性hash可以非常方便的建立node和资源之间关系，即哪些node保存了哪些资源可以很容易查询到。

2）结构化路由

- 节点加入：开始时，获得一个NodeID，连接一个“bootstrap”节点，加入P2P网络
- 发布资源：生成资源的ObjectID，查找和ObjectID距离最近的N个Node，向这N个node广播新资源信息，告诉这些节点，我有某某资源
- 资源搜索：找到最靠近资源的N个node(使用NodeID 和ObjectID来计算距离远近)，向这些node发送资源查询信息，如果有这个资源的详细信息，就返回给客户端，否则返回离资源更近的node列表给客户端，直到查询到资源提供者信息。

2）结构化路由（继续）

- 资源下载：如果没查到资源提供者信息，且没有更近的node了，那就说明这个资源没有提供者，如果找到资源提供者信息(NodeID,ip,port)后,向这个资源提供者请求
- 多个节点有相同资源，那么资源搜索得到一个资源提供者列表，可以开始P2P的下载
- 结构化路由和洪泛路由的区别：有目的、有针对性的路由



4.4.2 Chord/CFS

- 环形P2P网络
  - Chord：优美而精确的P2P网络
  - 在N个节点的网络，每个节点保存O(logN)个其它节点的信息
  - 在O(logN)跳内可找到存储数据对象的节点
  - 节点离开或加入网络时，保持Chord自适应所需消息数 $O(\log^2 N)=O((\log N)^2)$
  - 可提供数据对象的存储、查询、复制和缓冲
  - 在其上构架有协同文件系统CFS（Cooperative File System）
- MIT提出

- Stoica, I.; Morris, R.; Kaashoek, M. F.; Balakrishnan, H. (2001). "Chord: A scalable peer-to-peer lookup service for internet applications" (PDF). ACM SIGCOMM Computer Communication Review.

Chord基础工作原理

- ID的分配：通过hash函数(如SHA-1) (Secure Hash Standard)
  - NodeID=H(node属性)=H(IP地址/端口号/公钥/随机数/或其组合)
  - ObjectID=H(object属性)=H(数据名称/内容/大小/发布者/或其组合)
  - SHA-1的长度值m=160 Bits，从而保证其唯一性和几乎不重复性，故nodeID/objectID均可在[0...2^m）中选取

- 索引的分配
  - NodeID从小到大、顺时针排列于1个环上
  - 数据对象k(即ObjectID=k)也按环上顺时针方向，分配到节点k或第一个比k大 (因为环，需要mod 2^m) 的节点上。这个节点称为Successor(k)节点
  - 形式化表示 Successor($object_k$) = $node_k$ 或 $node_x$ ，x是现存网上顺时针第一个大于k的节点

Chord基础工作原理

论文图2，假设m=3，描述节点（Node ID）和key（Object ID）的分布：

- Successor(1)=1；Successor(2)=3；Successor(6)=0

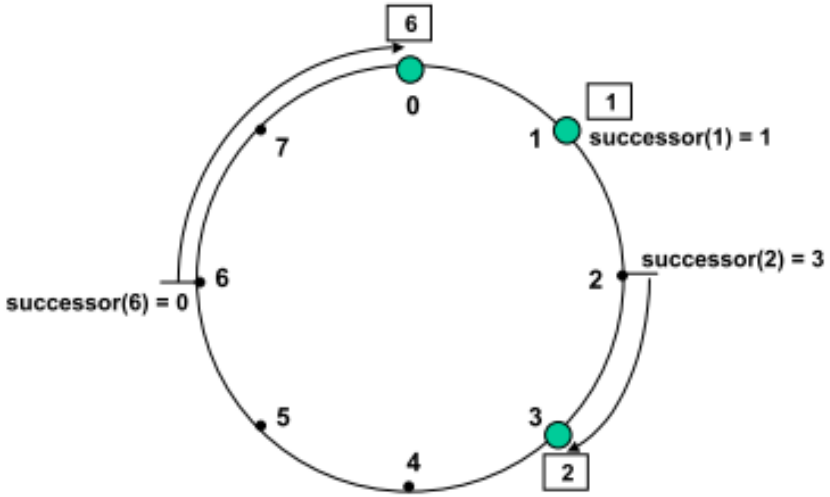


Figure 2: An identifier circle consisting of the three nodes 0, 1, and 3. In this example, key 1 is located at node 1, key 2 at node 3, and key 6 at node 0.

Chord基础工作原理

- 查询索引
  - 按上述后继关系，Chord显然可以正确工作
  - 但查询效率低下，比如要找ObjectID=K的资源，首先要询问K节点是否存在，然后找K+1，最坏情况要顺序查询O(N)个节点。
  - 所以Chord引入finger table来优化， finger table实际类似路由表：使每步更快接近目的节点

节点路由表的分配

- 指向表：finger table
  - 每个node存储大小为m的路由表（finger table），以减少路由跳数
  - 节点n的路由表中，第i项指向节点 s = Successor($n+2^{i-1}$),$1 \le i \le m$
  - 故s是在顺时针方向到节点n的距离至少为2^{i-1}的第一个节点：记作 n.finger[i].node
  - 除了对象k有Successor，节点n也有Successor节点就是 n.finger[1].node

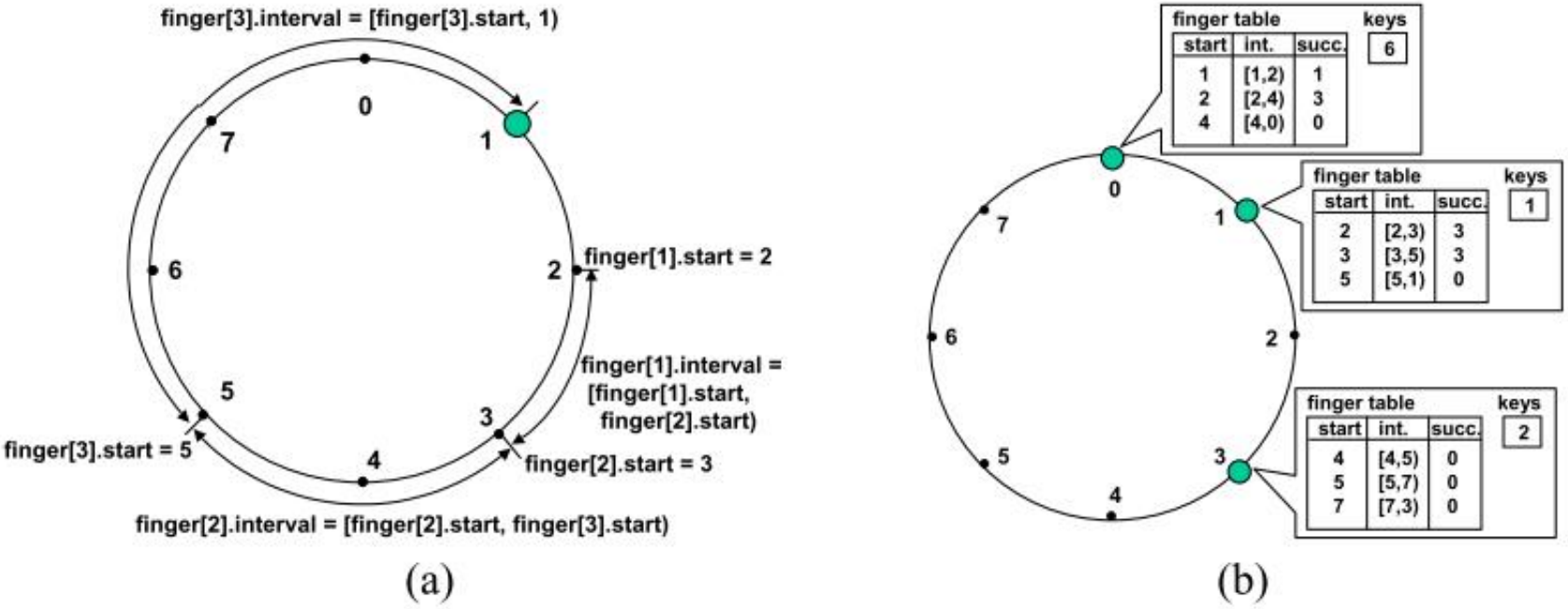
```
#define successor finger[1].node
```

  - 一个路由表项还包括相关节点的NodeID和IP地址、端口号

- 特点
  - 每节点只保存很少其它节点信息，且对离它越远的节点所知越少
  - 节点不能从自己的路由表中直接看出对象k的后继节点

Chord的finger table

- 论文图3，左图是Node 1的finger table每一项的范围
- 右图是如果有三个节点，那么他们的finger table内容



查找资源（不修改finger）

- 有了finger table后，就可以快速查找一个key的Successor：
  - 假设k是ObjectID，要找到保存k的Succesor(k)节点，从n节点开始查找 (n可以是任意一个节点)
  - 节点n在自己的路由表中寻找在k之前且离k最近的节点j，让j去找离k进一步最近的节点，如此递归下去，j将离k越来越近，最终找到n’
  - n’就是在k之前而且离k最近的节点
  - 那么n'的Successor就是要找到的节点Succesor(k)
  - 这个算法对应论文图4的伪代码

查找资源

- 论文图4的伪代码

```
// ask node n to find id's successor
n.find_successor(id)
  n' = find_predecessor(id);
  return n'.successor;

// ask node n to find id's predecessor
n.find_predecessor(id)
  n' = n;
  while (id ∉ (n', n'.successor])
    n' = n'.closest_preceding_finger(id);
  return n';

// return closest finger preceding id
n.closest_preceding_finger(id)
  for i = m downto 1
    if (finger[i].node ∈ (n, id))
      return finger[i].node;
  return n;
```

先找id之前的最近节点n'

迭代寻找节点n'

在finger表中找到合适的node，从m到1是因为finger[m]覆盖范围大

Figure 4: The pseudocode to find the successor node of an identifier *id*. Remote procedure calls and variable lookups are preceded by the remote node.

新节点加入（修改finger）

论文在一个新节点加入到Chord后，为了更新finger table，每个节点除了Successor还有一个Predecessor节点

- 前驱节点：Predecessor(n)
  - 在节点n之前，不等于n且离n最近的节点Predecessor(n)
- 论文图6描述Join算法

新节点加入，必然影响finger表

- 论文图6描述Join算法（1）

```
#define successor finger[1].node

// node n joins the network;
// n' is an arbitrary node in the network
n.join(n')
  if (n')
    init_finger_table(n');
    update_others();
    // move keys in (predecessor, n] from successor
  else // n is the only node in the network
    for i = 1 to m
      finger[i].node = n;
    predecessor = n;

// initialize finger table of local node;
// n' is an arbitrary node already in the network
n.init_finger_table(n')
  finger[1].node = n'.find_successor(finger[1].start);
  predecessor = successor.predecessor;
  successor.predecessor = n;
  for i = 1 to m - 1
    if (finger[i + 1].start ∈ [n, finger[i].node))
      finger[i + 1].node = finger[i].node;
    else
      finger[i + 1].node =
          n'.find_successor(finger[i + 1].start);
```

n依靠另外任意节点n'，建立自己的finger table。然后由于n的加入，更新其他节点的finger

n为唯一节点，那么finger表中node都是自己，predecessor也是自己

找到finger表第一个item，是n的successor，然后修改predecessor关系

这部分是优化

找到finger表其它item的node

新节点加入，必然影响finger表

- 论文图6描述Join算法（2）

```
// update all nodes whose finger
// tables should refer to n
n.update_others()
  for i = 1 to m
    // find last node p whose i^th finger might be n
    p = find_predecessor(n − 2^(i−1));
    p.update_finger_table(n, i);

// if s is i^th finger of n, update n's finger table with s
n.update_finger_table(s, i)
  if (s ∈ [n, finger[i].node))
    finger[i].node = s;
    p = predecessor; // get first node preceding n
    p.update_finger_table(s, i);
```

因为n的加入，所以需要修改所有可能指向n的节点的finger表

find_predecessor(n-2^{i-1}的含义是找到最后一个节点其finger[i]指向n

注意这里是个递归过程，用来不断修改可能节点的finger表

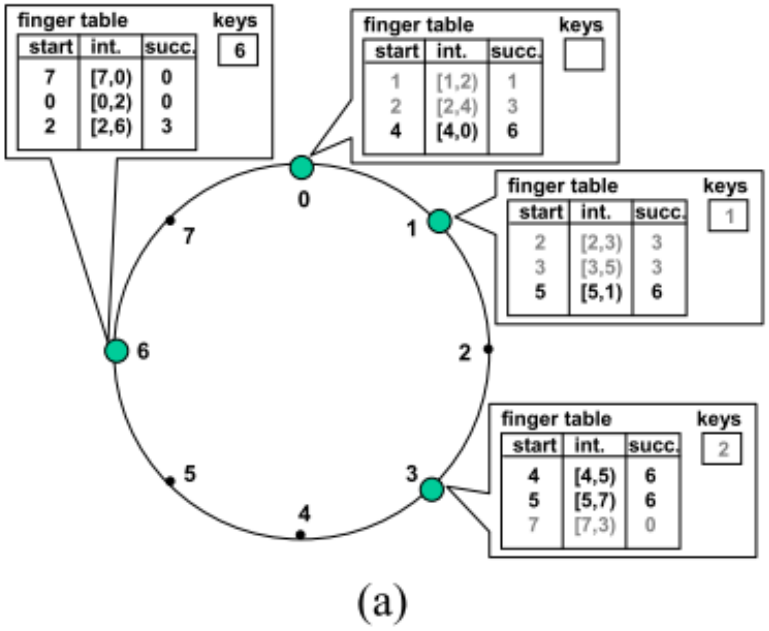
Chord中节点离开

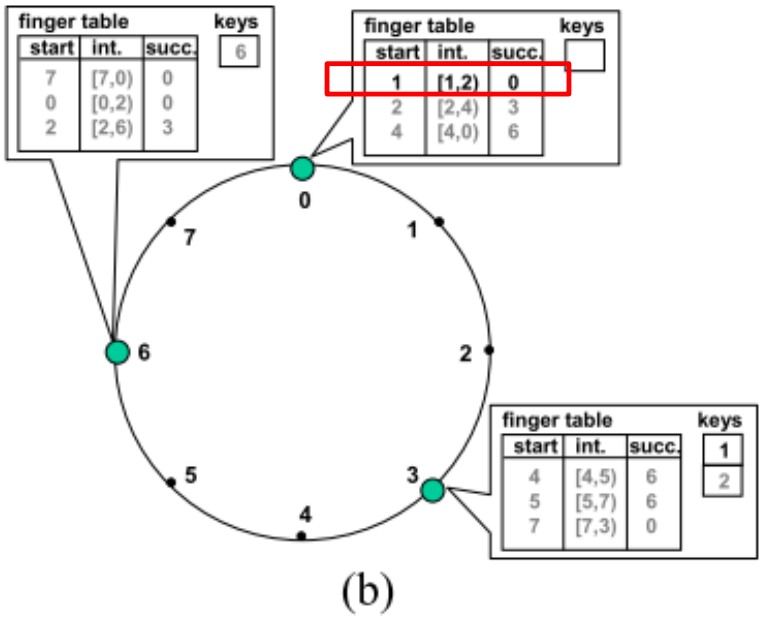
- 论文图5：finger table的变化：

左图Node 6加入每个节点路由表的变化（灰色是不变）

右图Node 1离开每个节点路由表的变化（论文中错误标记为Node 3）

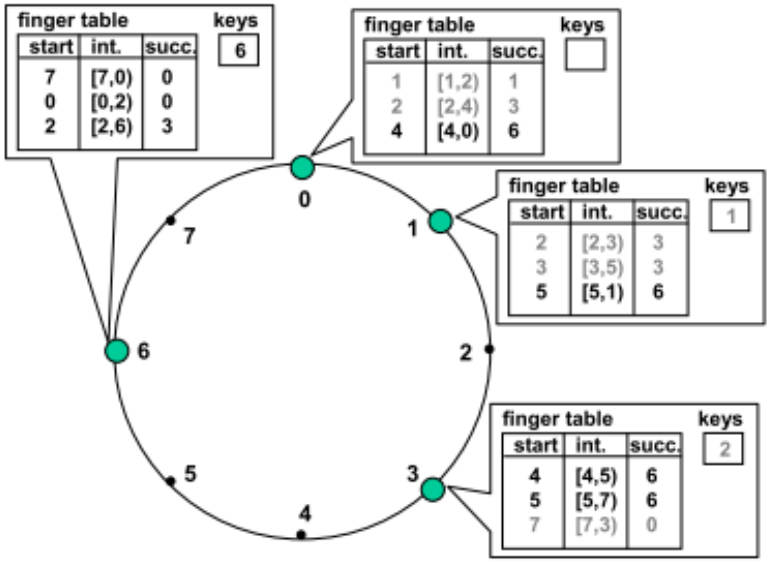
（另外：Node 1离开后，Node 0的finger table第一项应该是3）





Chord的例子

- 左图增加一个节点6，跟右边伪代码对照。
- 右边伪代码，双重循环，外循环m次找可能finger_table[i]=6的节点。
  - 外层循环i=1： find_predecessor(6-1)=3
  - 内层循环： 节点3的finger[1]=6，然后再找前继节点1，6不属于[2,3）停止
  - 外层循环i=2……



(a)

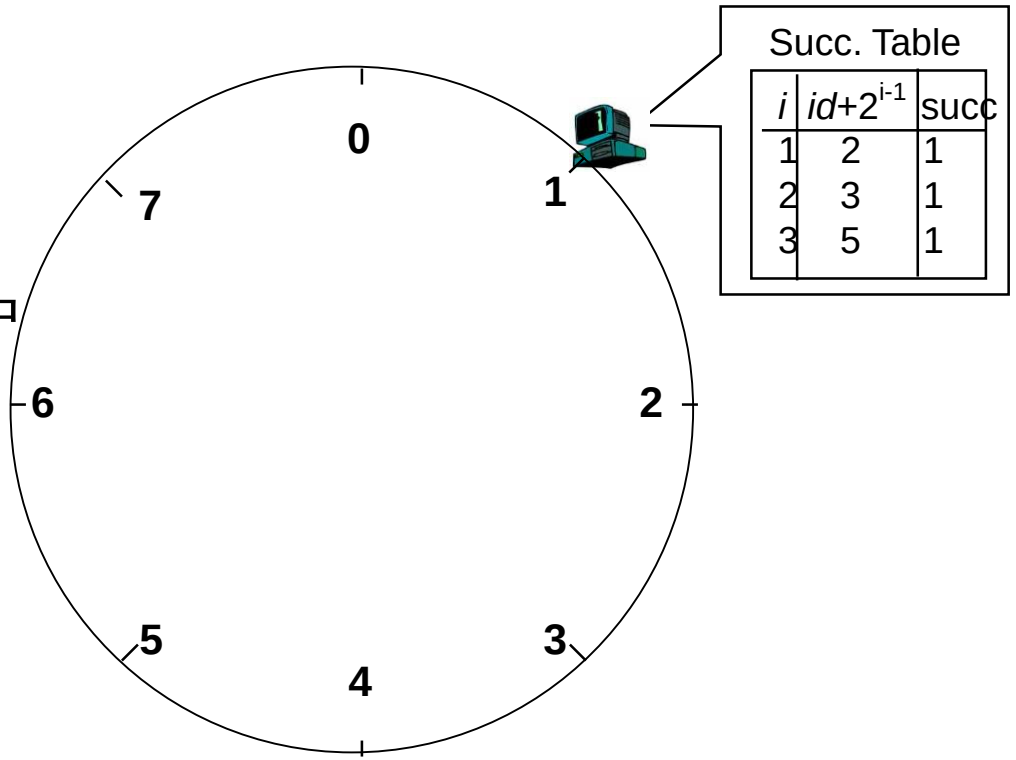
```
// update all nodes whose finger
// tables should refer to n
n.update_others()
  for i = 1 to m
    // find last node p whose i^th finger might be n
    p = find_predecessor(n − 2^(i−1));
    p.update_finger_table(n, i);

// if s is i^th finger of n, update n's finger table with s
n.update_finger_table(s, i)
  if (s ∈ [n, finger[i].node))
    finger[i].node = s;
    p = predecessor; // get first node preceding n
    p.update_finger_table(s, i);
```

Chord的例子

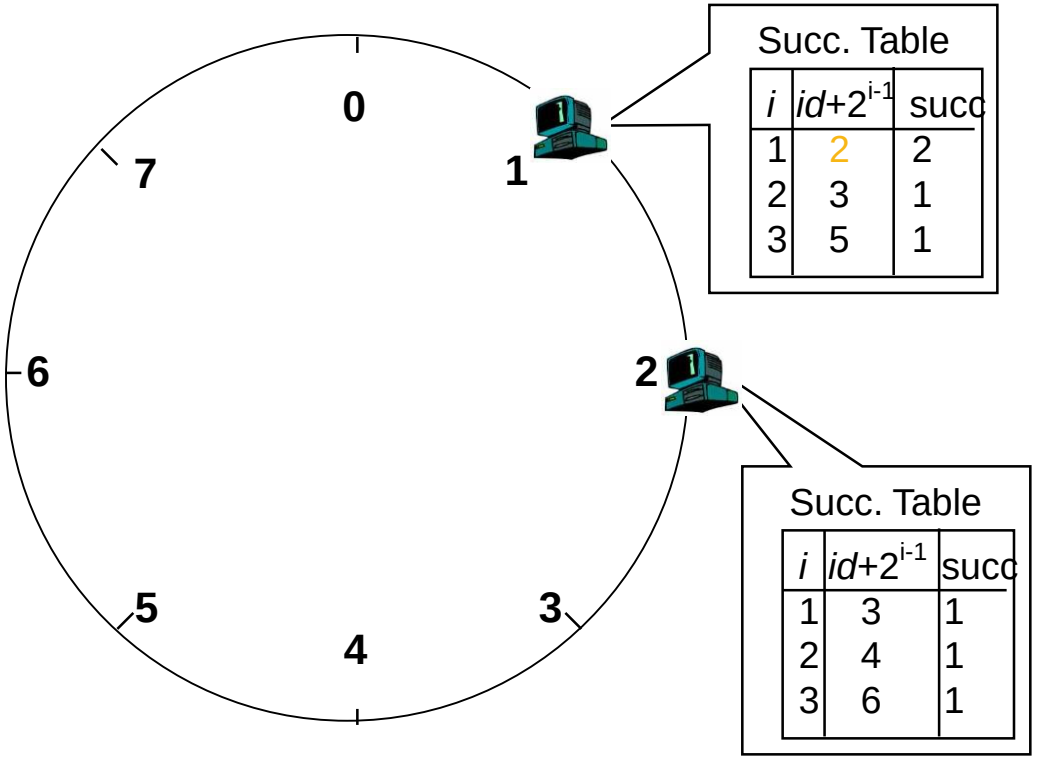
- 假设id空间是 [0..2^m-1]
  - m=3, [0.. 7]; 0≤i≤2
- 节点1加入
  - 即id=1之节点加入

  由于只有一个节点，所以路由表中都是自己



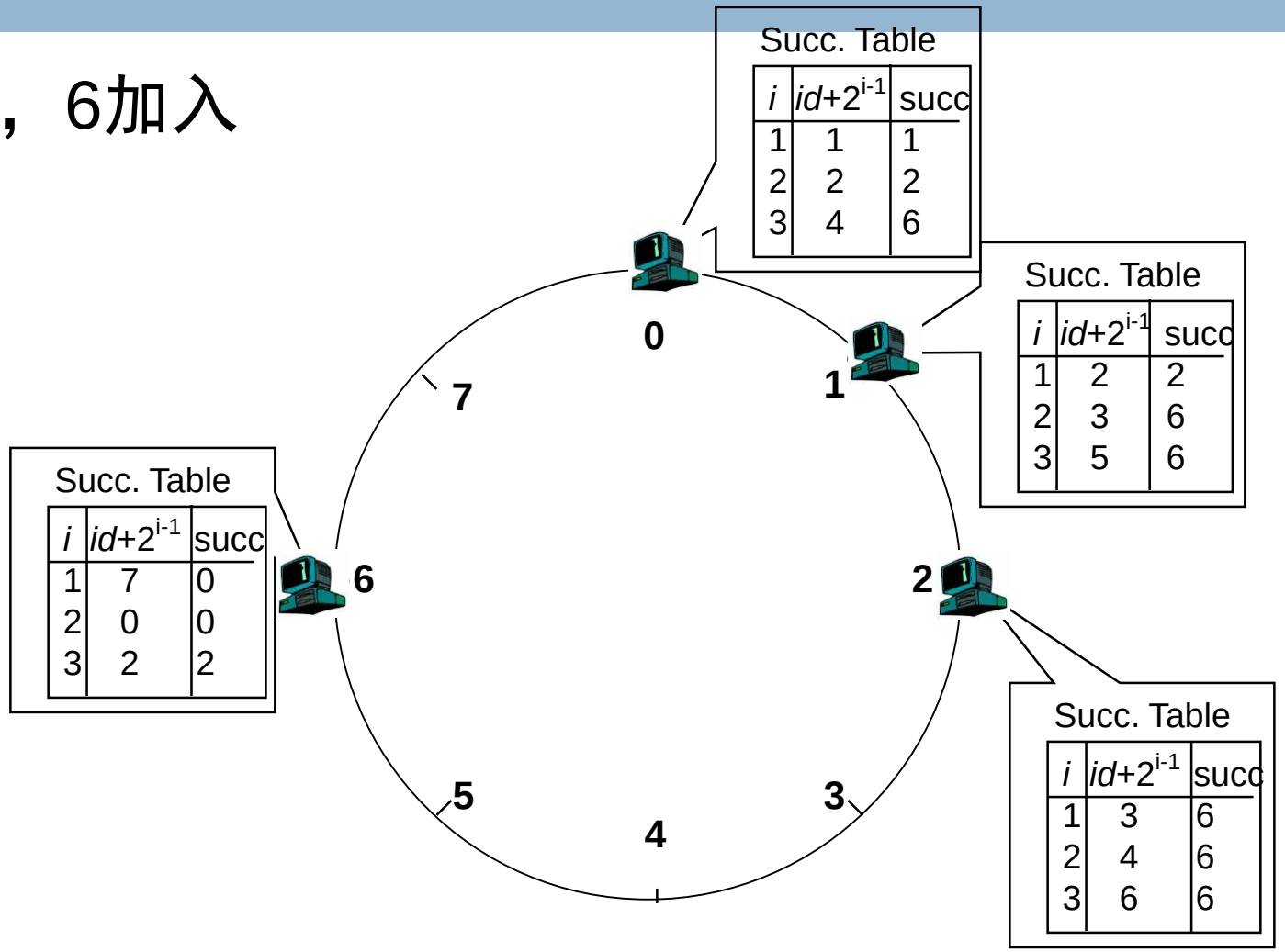
Chord的例子

- 节点2加入



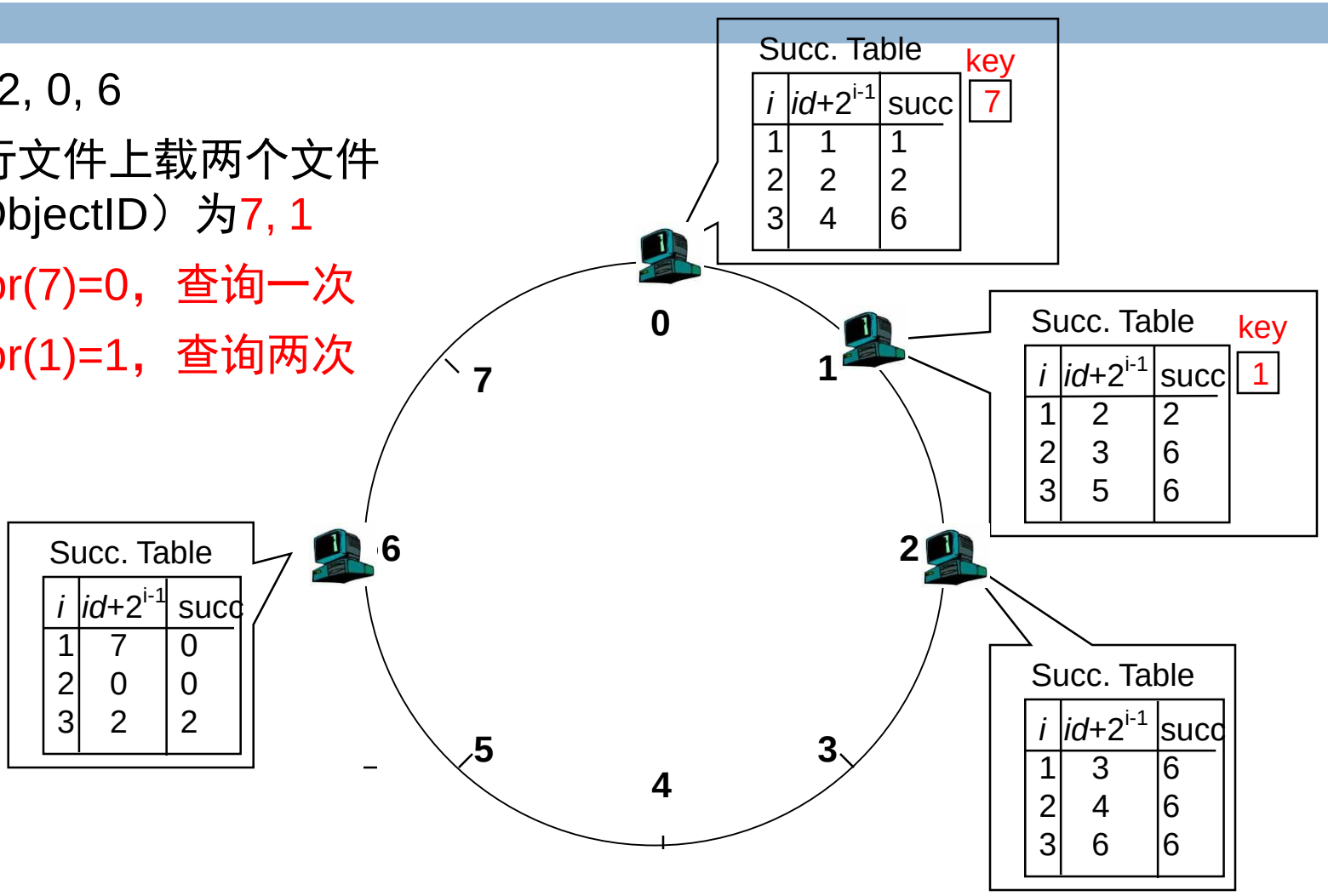
Chord的例子

- 节点0，6加入



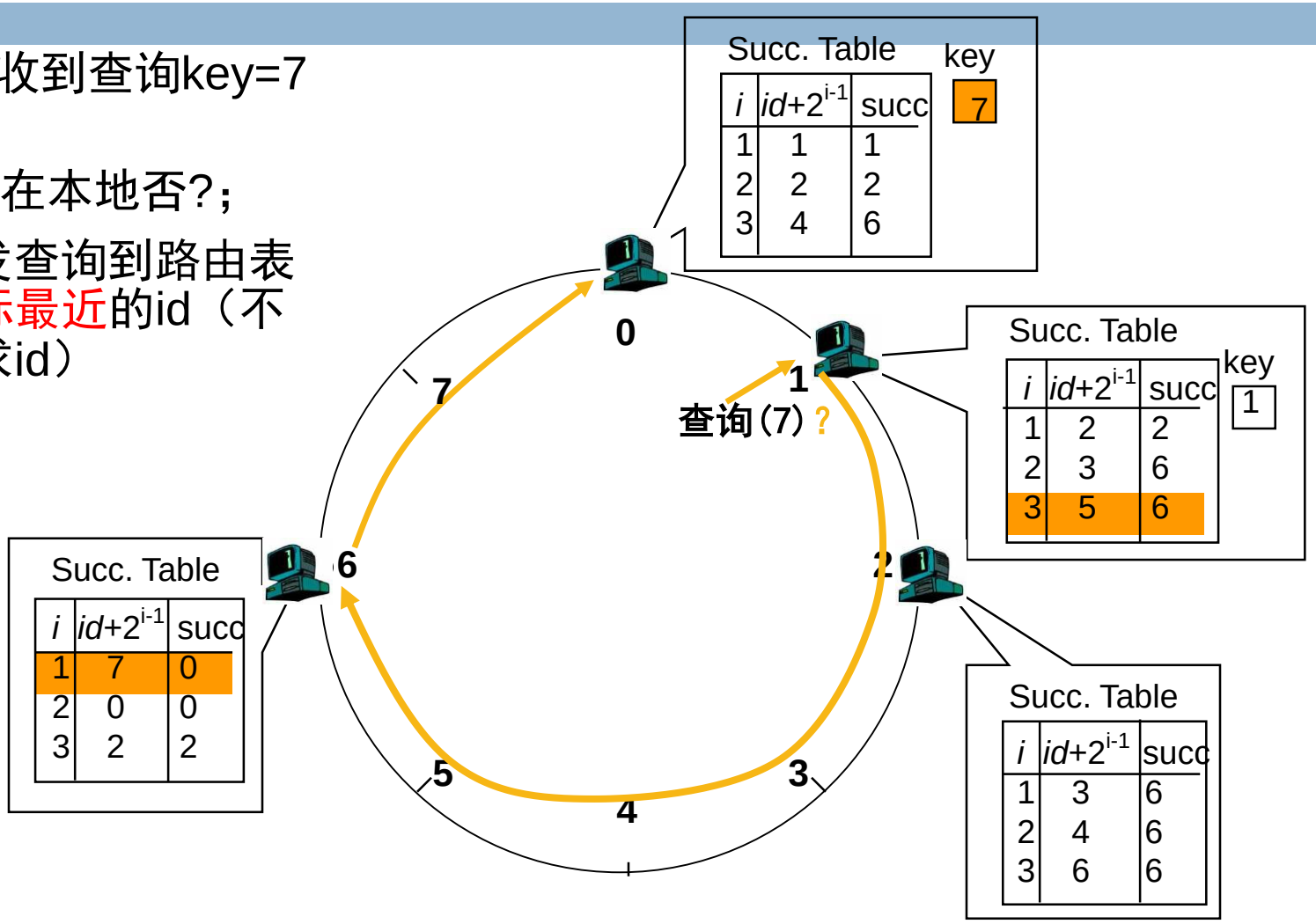
Chord的例子

- 节点：1, 2, 0, 6
- 节点6进行文件上载两个文件，key（ObjectID）为7, 1
- Successor(7)=0，查询一次
- Successor(1)=1，查询两次



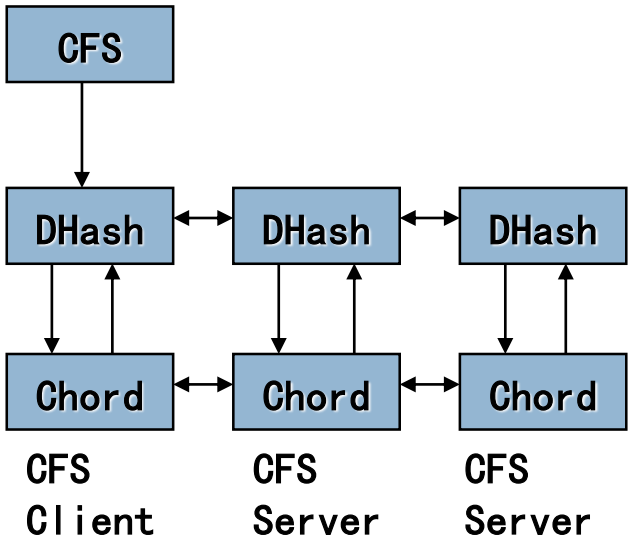
Chord的例子

- 当节点1收到查询key=7请求后：
- key=7存在本地否?；
- 否，转发查询到路由表中与目标最近的id（不超过请求id）



协同文件系统CFS简介

- CFS是以Chord作为基础的P2P只读文件存储系统
  - 依靠Chord作为其分布式HASH表，提供高效、容错和负载均衡的文件存储和获取
  - 系统由文件系统FS、分布式散列表DHash和底层定位散列表Chord三层构成
  - 每个节点既是服务器又是客户机
- 功能说明
  - FS:高层，从DHash层获取数据块，将这些快转换成文件，给更高层文件系统接口
  - DHash：中间层，分布和缓存数据块以负载平衡，复制数据块以容错，通过服务器选择来减少延迟，用Chord定位数据块
  - Chord：底层，维护路由表，定位数据块所在的服务器。每个服务器就是Chord的一个节点



CFS的系统结构

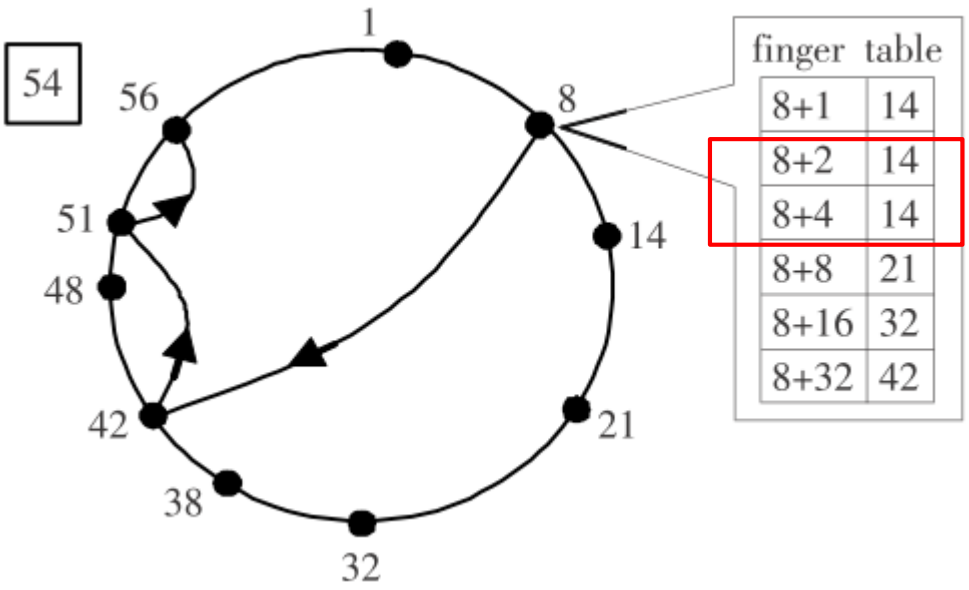
Chord：总结

- 简单、精确，但要求
  - 每个节点的后继节点始终准确
  - 每个对象k的后继节点始终承担k的索引
- 查询与维护代价
  - 查询代价：m项的平均间隔 delta = $[2^0+ 2^1+\ldots +2^{m-1}]/m$;设经J跳命中，覆盖节点数≤2^m ;故 J*delta≤2^m ;故有J≤m = O(log *N*) ；一次查询的路由平均跳数为O(log*N*)
  - 维护代价：节点的进/出，自适应到最新状态需$O(\log^2 N)$
- 缺点：
  - 没有实际使用（只有一个文件共享应用）
  - 不支持非精确查找
  - finger table有冗余
  - 维护finger table（加入/退出）代价比较高

Chord：总结

Chord的问题（1）：finger table有冗余

- 以 2^i 为跨度来寻找路由表中的后继节点，这会在 finger 中产生一定的冗余量，例如下图中第2、3条路由是冗余的。
- 即节点稀疏的时候，路由容易冗余



Chord：总结

Chord的问题（2）：路由维护开销大

- 节点加入：Chord需要环形拓扑中的任意一个节点来协助完成,且加入过程包括新节点本身的Join操作和修改其他节点finger表两个阶段；
- 节点失效的处理：Chord需要周期性对节点的前继节点和后继节点进行探测，并按照节点加入时的算法重建Finger表；
- 对于节点退出的处理：Chord采取了将节点的退出当作为失效来处理的方式。

4.4.6 结构化P2P网络Kademlia

基本概念

- Kademlia和Chord类似，属于常数度P2P网络
  - 路由、定位、自组织方式与前4种区别不大
  - 每个节点的“度”（连接数）是固定的，与规模无关
    - 维护固定路由表项，仍能达到O(logN)跳的指数定位效率
    - 路由表固定导致网络自适应开销减少
- 更容错实用的结构化网络Kademlia
  - 路由方法类似Chord
  - 加入、离开节点更加简单

4.4.6 结构化P2P网络Kademlia

- Kademlia协议是美国纽约大学的 Petar P. Maymounkov 和David Mazieres 在2002年发布的一项研究结果：

  《Kademlia: A peer-to-peer information system based on the XOR metric》

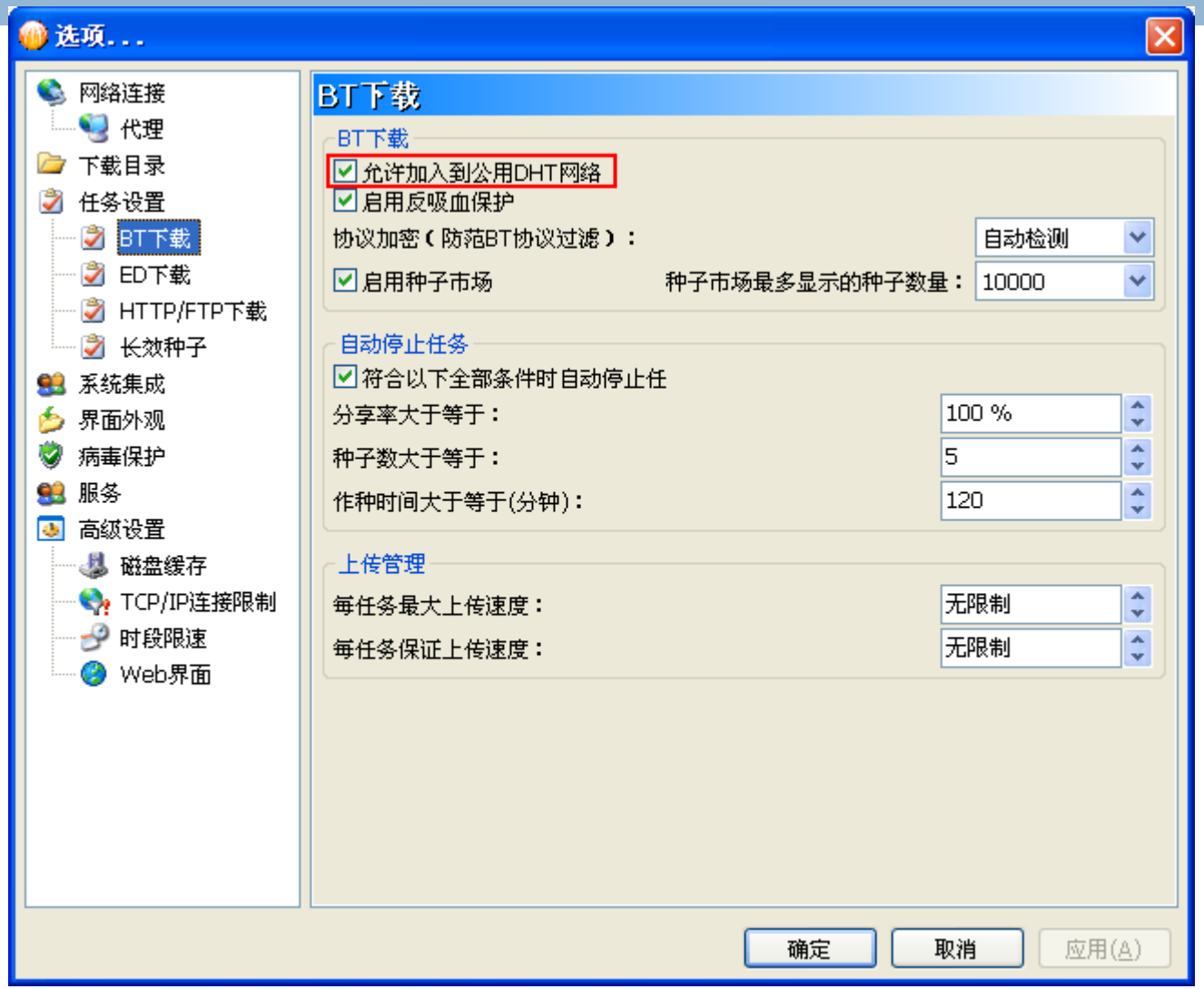
- Kademlia 是一种分布式哈希表（DHT）技术，Kademlia 通过独特的以异或算法（XOR）为距离度量基础，建立了一种全新的DHT 拓扑结构，相比于其他算法，大大提高了路由查询速度

当前应用现状

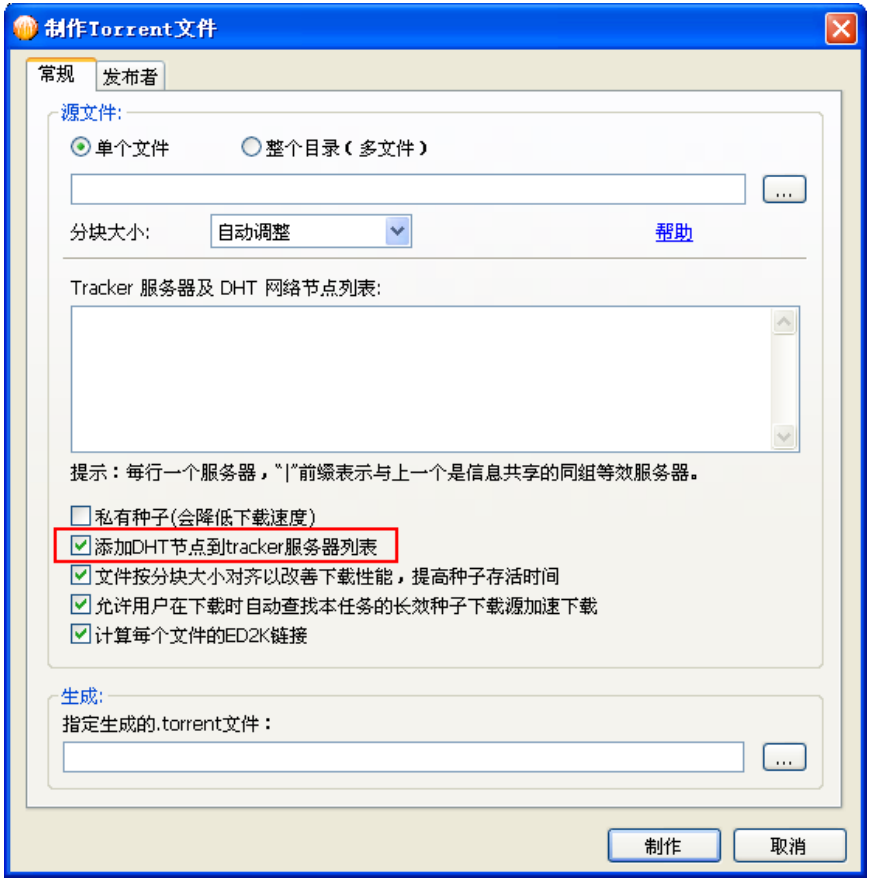
- 在2005年5月著名的BitTorrent在4.1.0版实现基于Kademlia协议的DHT技术后，很快国内的BitComet和BitSpirit也实现了和BitTorrent兼容的DHT 技术，实现trackerless下载方式。

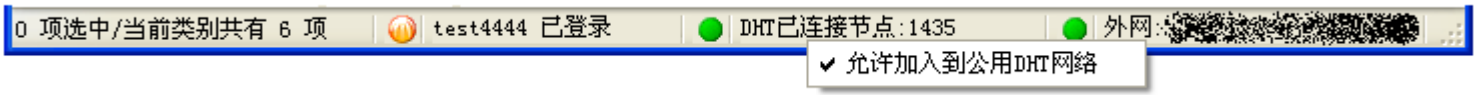
- 另外，emule中也很早就实现了基于Kademlia类似的技术（BT中叫DHT，emule中叫Kad），和BT软件使用的Kad技术的区别在于key、value和node ID的计算方法不同。

当前应用现状（BitComet）



当前应用现状（BitComet）





Kademlia协议的应用		
所在网络		具体应用说明
Overnet网络	Overnet	已被整合到eDonkey2000中
	eDonkeyHybrid	混合式eDonkey软件
	mlDonkey	运行于多平台，多网络的eDonkey扩展版软件，支持多协议，如BitTorrent, eDonkey, DirentConnet, FastTrack, Gnutella2, FTP/HTTP, Kad, Cvernet, OpenNap, SoulSeek
Kad网络	eMule	开始于0.40版
	mlDonkey	开始于2.5—28版
	aMule	即all-platform eMule，是eMule的扩展版，支持很多OS，开始于2.10
RevConnect		基于DirentConnet协议的P2P文件共享系统，以Kademlia为分布式Hash，能多源下载，部分文件共享，安全用户认证，自动资源回收、增强的自动搜索等功能，开始于.404版
KadC		用以在Overnet网络中发布、获取信息的C语言库
Azureus		开始于2.3.0.0版，使用Kademlia网络作为BitTorrent Trackers失效是的替代定位方法
BitTorrent		其Beta 4.1.0版拥有一个Kademlia网络，为无Trackers的Torrents服务
BitSpitit		基于BitTorrent协议的一个客户端，开始于3.0版
eXeem		基于BitTorrent网络的一个P2P文件共享软件，目的是取代BitTorrent中原有的Trackers，开始于Beta 0.25版

节点间的异或距离

- 定义两个节点x,y（ID值表示）的“异或”距离（非欧距离）
  - 节点与数据对象都用SHA-1分配160 Bits ID，
  - 对象索引由与objectID最接近的nodeID负责，“最接近”由XOR距离度量
  - d(x,y)=x⊕y，如1011⊕0010=1001=9 ＞ 1011⊕1010=0001=1
- 异或距离的性质：
  - 合理性： d(x,x)＝ 0
  - 非负性： d(x,y)≥ 0
  - 对称性： d(x,y) ＝ d(y,x) ，对任何x,y,且 x≠y. （Chord不具备）
  - 三角不等式： d(x,y)+d(y,z)≥ d(x,z)
    - 因为d(x,z) ＝ d(x,y)⊕d(y,z)
    - 对任意 a≥0, b≥0 , a+b≥ a⊕b
  - 单向性：任意节点x和距离d，系统中仅有唯一节点一y满足d(x,y)=d
    - 单向性保证相同数据的定位最终将会汇聚于相同的路径
  - 传递性：显然 if d1≥d2 and d2≥d3 ,then d1≥d3 成立

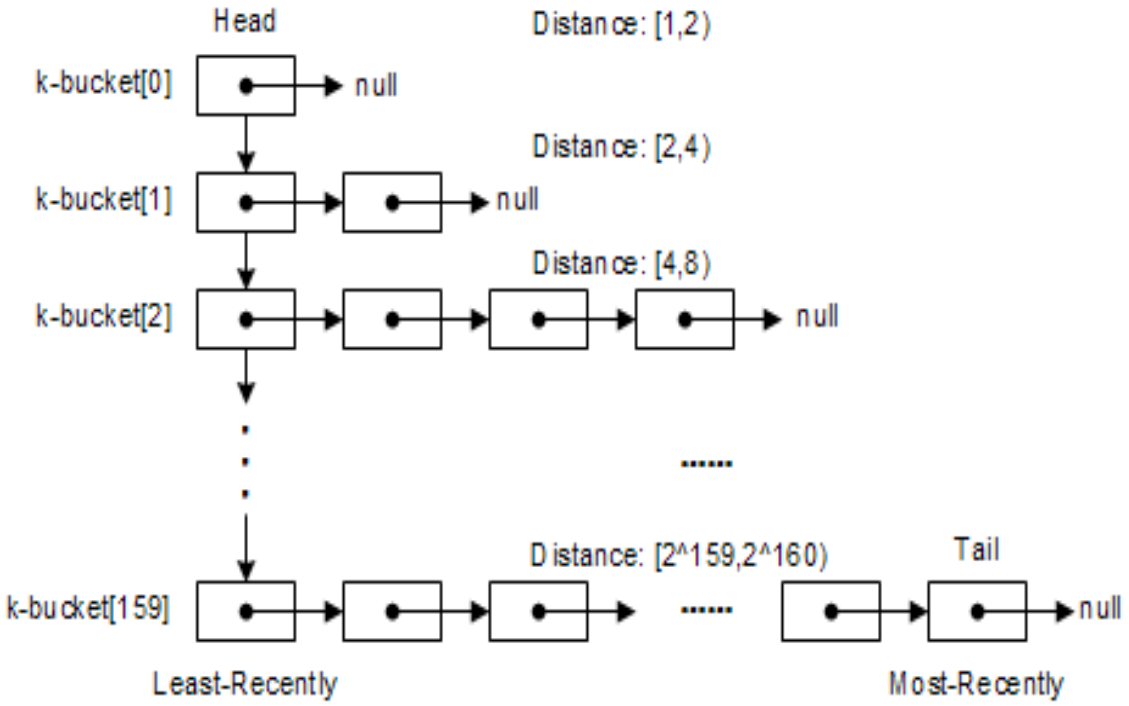
- 异或距离结构性好处
  - 按当前节点ID与所有其它节点ID间XOR距离大小排队，知道目标结点ID后，就很容易计算出目标节点在这条队列中的位置；
  - 如果给定一个异或距离，你也能很容易从这条队列里找出与该距离最接近的那些结点
  - 与自己前缀相似度越高的结点离自己越近（异或之后高位为0）
- 异或求距离

```
      010101
XOR   110001
-------------
      100100 = 32＋4＝36。显然，高bit更容易影响距离
```

K-桶路由表

- K-buckts：每节点维护一个路由表，它由logN（N为最大节点数）个k桶构成
  - 实质是一个链表集，每节点 0≤i＜160个桶；每桶内装Max个（10或20）记录=（序号i，到自己的异或距离[2^i—2^{i+1})，节点信息）
  - 节点信息 = <IP，端口，节点ID >三元组
  - 表项以访问时间排序，最久（least-recently）访问的放在尾部，最近（most-recently）访问的放在头部

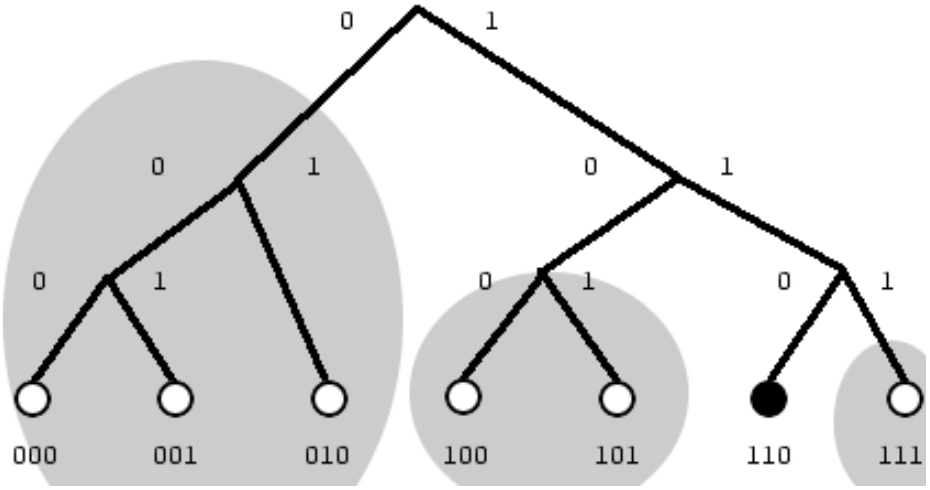
I	距离	邻居
0	$[2^0,2^1)$	$(IP\ address,UDP\ port,Node\ ID)_{0\text{-}1}$ ..... $(IP\ address,UDP\ port,Node\ ID)_{0\text{-}k}$
1	$[2^1,2^2)$	$(IP\ address,UDP\ port,Node\ ID)_{1\text{-}1}$ ..... $(IP\ address,UDP\ port,Node\ ID)_{1\text{-}k}$
2	$[2^2,2^3)$	$(IP\ address,UDP\ port,Node\ ID)_{2\text{-}1}$ ..... $(IP\ address,UDP\ port,Node\ ID)_{2\text{-}k}$
.....		
i	$[2^i,2^{i+1})$	$(IP\ address,UDP\ port,Node\ ID)_{i\text{-}1}$ ..... $(IP\ address,UDP\ port,Node\ ID)_{i\text{-}k}$
.....		



- 每个桶容量大小
  - 很小的i的桶，包含与自己很近的节点
  - 很大的i的桶，包含与自己很远的节点
  - 每个桶容纳节点上限为max，一般偶数，emule的max=10，OverNet的max=20,BT的max=8
- Kad路由表实际是一颗二叉树
  - 叶节点为K桶，存放有长度为i的相同前缀的节点信息
  - 这个前缀就是该K桶在二叉树中的位置
  - 每个K桶都覆盖了ID 空间的一部分，全部K桶的信息加起来就覆盖了整个160bit的ID空间，且不重叠

K桶可以看作一颗二叉树

- 对于一个节点ID为110，那么其K桶覆盖范围可以整个ID空间分解为一系列连续的，不包含自己的子树：如下图
  - 在最高层子树，左边是不包含自己的子树（000，001，010）
  - 下一层子树由剩下部分不包含自己的一半组成（100，101）
  - 依此类推，直到分割完整颗树（111）
  - 下图中每个阴影可以看做一个K桶，由于每个K桶最多Max个节点，所以离110越近的K桶，可能的覆盖率越高



节点的行为（初始化）

节点U初始化

- Kademlia读取本地配置文件
  - src_index.dat,key_index.dat load_index.dat（索引）
  - nodes.dat（上次程序启动时连接上的节点 RoutingZone.cpp）
  - preferencesKad.dat (上次程序启动时本地节点IP、ID、Port)
- 没有ID，生成ID
  - 根据特定信息Hash或随机生成160位ID
- 构造本地结点二叉树
  - 所有上次连接到的节点都根据其ID值加入到不同的K桶中

结点的行为（加入）

- U在自己的K桶中找到一个距离自己最小的节点W；
- U向W发出一个对自己id的FIND_NODE操作；W会返回已知的最接近id的节点，U根据收到的节点信息更新自己的K桶内容
- 按照上面过程，U对自己邻近节点由近及远的逐步查询刷新所有的 k-bucket；（刷新算法）
- 在刷新的同时，也把自己的信息发布到其他节点的K桶中（其他节点收到FIND_NODE也会更新K桶）

节点间的交互行为

- Kademlia协议包括四种远程RPC操作
  - PING：探测一节点，判断其是否仍然在线。并在回应中携带网络地址
  - STORE：指示一节点存储一个<key, value>对，以便查找
    - key 是对象的hash值，即ObjectID
    - Value是真正的数据对象或其索引
  - FIND_NODE：以160bit ID 作为参数。本操作的接受者返回它所知离目标ID最近的a个节点的(IP address ,UDP port , Node ID)三元组信息。（如果a=1，类似于chord）
  - FIND_VALUE：以key为参数寻找key对应的value.
    - 同FIND_NODE过程
    - 若接受者已收到同一key的STORE消息，则直接返回存储的value 值。（cache：加速）
    - 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。（cache：查找成功范围扩大）

节点间的交互行为

- Piggy back确认捎带
  - PING 消息可以在其他RPC响应里使用捎带确认机制来获得发送者网络地址，避免多余的PONG
- 防止消息伪造
  - 所有RPC都带一160 bit随机 RPC ID，由发送者产生，
  - 接收者返回消息里面必需拷贝此 RPC ID。目的是防止伪造回复消息

节点更新自己的K-BUCKET

- x节点收到y节点消息，则更新与d(x,y)= x⊕y 对应的K桶
  - 若y已在于该K 桶中，把对应项移到该K桶的头部
  - 若y不在该K 桶中
    - 若该K桶不满，把y的三元组插入队列头部
    - 若该K桶满了，向尾节点z（最老）发RPC_PING
      - 如果z没有响应，则从K 桶中移除z 的信息，并把y的信息插入队列头部
      - 如果z有响应，则把z 的信息移到队列头部，同时忽略y的信息
  - 若x在过去的1小时在某个桶覆盖范围内没有发生查询操作
    - x节点删掉该桶中的所有节点，默认这些节点都下线或者都没有保存资源
    - x节点就随机FIND_NODE一个在该桶覆盖范围内的节点id,即希望刷新该桶中的节点

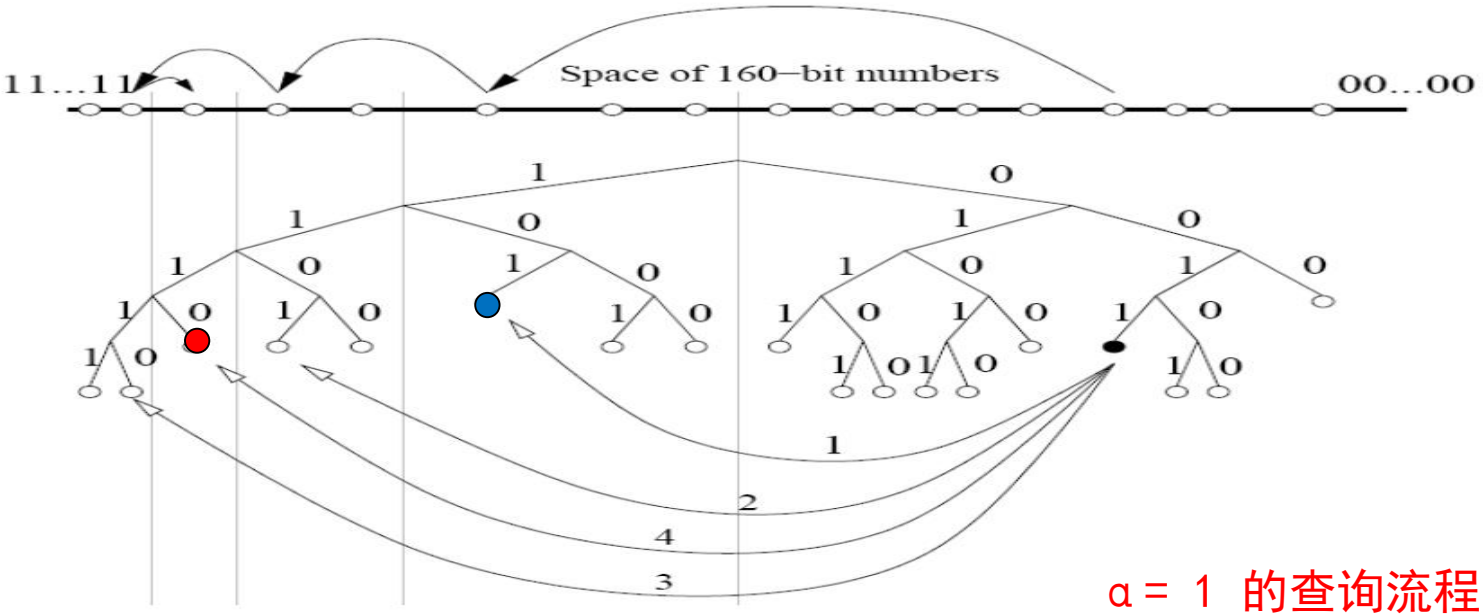
节点更新自己的K-BUCKET

- 特点
  1. 离自己越近的节点越容易放在K桶中，越远越欠了解
  2. 在线时间长的节点具有较高的可性继续保留在K桶中
  3. 可以保持系统稳定和减少节点进出的路由维护代价
  4. 若某节点长时间在线，则网络中有很多节点连接到该节点，其负载会随着在线时间的增大而增大，有可能导致负载极不平衡（所以需要限制节点并发数）

节点查找（找目的节点的三元组）

- node lookup:
  - 找到给定ID距离最近的a个node，a为并发参数，比如3
  - Step1:从k-bucket里面取出 a 个离ID最近的 node（不一定在同一个桶内），然后向这a个node并行发送异步的 FIND_NODE RPC
  - Step2:发送FIND_NODE RPC给前一步返回的node,不必等a个 PRC都返回后才开始
    - 从响应者已知的最接近目标的nodes当中，取出a个还没有查询的node，向这a个node发送FIND_NODE RPC
      - 没有迅速响应的node将考虑排除，直到其响应。
    - 若一轮FIND_NODE RPC没有返回一个比已知的所有node更接近目标ID，发送者将再向a个最接近目标的、还没有查询的node 发送FIND_NODE RPC。
  - Step3:查找结束的条件：发送者经过多轮查询得到距离ID共a个最近的node（不一定在自己的K桶内），并且每个node都有响应。

- 节点0011查询节点1110（注意图中长度不够的ID后面补0）
  - 计算距离 d = 0011⊕1110 = 1101 = 13
  - 找到自举节点1010（蓝色,一启动就存在的邻居节点），它计算1010x⊕1110 = 0100 =4，查对应4的k桶内容有→1101节点；
  - 1101节点计算1101⊕1110 = 0011 = 3，查对应3的k桶内容有→11110
  - 11110节点计算11110⊕1110x = 0010 = 2，查对应2的k桶命中→1110
  - 所以过程是收敛的



α = 1 的查询流程

数据的存储

- 存储<Key, Value>的步骤：
  - 用node lookup算法，找到距离Key最近的k个nodes
  - 向这k个nodes发送STORE RPC
  - 每隔一定周期（一个小时）,每个node重新发布(re-publish) <Key,Value>
- 发布与搜寻的一致性
  - 当节点w发现新节点u比自己更接近key，则w把自己的<key,value>对数据复制到u上，但是并不会从w上删除

节点数据的有效性保障

- 节点有效性
  - 利用经过自己的节点查询操作，持续更新对应的K桶信息
  - 对过去一个小时内还没收到任何节点查询操作的某个桶（buket）执行刷新操作---BT协议实现规定为15 分钟
- 数据有效性
  - 特点：节点离开网络不发布任何信息（弹性网络特点或目标）
  - 要求：每个Kad 节点必须周期性的发布（一个小时）本节点存放的全部<key , value>数据对，并把这些数据缓存在自己的k 个最近邻居处
  - 使失效节点上数据会被很快更新到其他新节点上。

4.4.7 结构化网络总结

- Chord/CAN/Tapestry/Pastry/Kademlia

- 目标相同
  - 减少路由到指定文件的P2P跳数
  - 减少每个Peer必须保持的路由状态

- 算法异同
  - 节点与对象Hash映射到同一空间，走“最接近”路由
  - 都保证算法的跳数与Peer群组的大小相关
  - 方法上的差别很小

DHT存在的问题和研究重点

- 解决P2P系统中固有的Churn高的现象，每个节点上下线都要O(LogN）的修复操作，研究方向：增大邻接表，增加发布及搜索冗余。
- 模糊查询的支持
  - OverNet的查询存在的问题：Key不能太多，切词准确性
  - 让DHT支持复杂查询，不按照NodeID组织DHT，而按照关键字来组织其他办法2：用DHT维护拓扑，而资源的定位仍然采用广播方式查询;分级的索引
- 文件的本地存储特性的消失
  - 目录的结构信息 (本地相同目录下两个相关文件可能被Hash到不同别的节点)

DHT存在的问题和研究重点

- DHT这种架构相对2代P2P更容易受到攻击 (研究重点)
  - 伪装节点
  - 拒绝转发消息等

- 不能符合系统中节点的异质性(可能弱节点被分配到热门关键字)
  - 对热门关键字作Cache
  - 采用分层DHT结构如Kelips，包括以物理位置为依据的分层，以兴趣为依据的分层等

P2P网络路由方式总结

- 服务器路由(Napster)；
- 洪泛路由(Gnutella)；
- 双层路由(KaZaA)；
- 数值邻近路由(Chord)；
- XOR邻近路由(Kademlia)；
- 逐位匹配路由(Tapestry)；
- 位置邻近路由(CAN),；
- 层次路由(Viceroy)；
- 混合式路由(Pastry)…

P2P网络结论

- The key challenge of building wide area P2P systems is a scalable and robust location service
- Solutions covered in this course
  - Naptser: centralized location service
  - Gnutella: broadcast-based decentralized location service
  - CAN, Chord, Tapestry, Pastry，Kademlia: intelligent-routing decentralized solution
    - Guarantee correctness
    - Tapestry (Pastry) provide efficient routing, but more complex