Distributed Hash Table

- Introduction
- DHT原理
- 代表性DHT算法
- 基于DHT的结构化P2P比较
- 基于DHT的P2P应用

1 Introduction

- Chord
- Pastry
- CAN

基于分布式Hash表
(DHT： Distributed Hash Table)

结构化P2P:
直接根据查询内容的关键字定位其索引的存放节点

2 DHT原理

- Hash函数可以根据给定的一段任意长的消息计算出一个固定长度的比特串，通常称为消息摘要（MD：Message Digest），一般用于消息的完整性检验。
- Hash函数有以下特性：
  - 给定 P，易于计算出 MD（P）
  - 只给出 MD（P），几乎无法找出 P
  - 无法找到两条具有同样消息摘要的不同消息
- Hash函数
  - MD5：消息摘要长度固定为128比特
  - SHA-1：消息摘要长度固定为160比特

2.1 Hash函数概述

- Hash函数可以根据给定的一段任意长的消息计算出一个固定长度的比特串，通常称为消息摘要（MD：Message Digest），一般用于消息的完整性检验。
- Hash函数有以下特性：
  - 给定 P，易于计算出 MD（P）
  - 只给出 MD（P），几乎无法找出 P
  - 无法找到两条具有同样消息摘要的不同消息
- Hash函数
  - MD5：消息摘要长度固定为128比特
  - SHA-1：消息摘要长度固定为160比特

2.2 Hash函数应用于P2P的特性

- 惟一性：不同的输入明文，对应着不同的输出摘要
  - 将节点IP地址的摘要作为节点ID，保证了节点ID在P2P环境下的惟一性

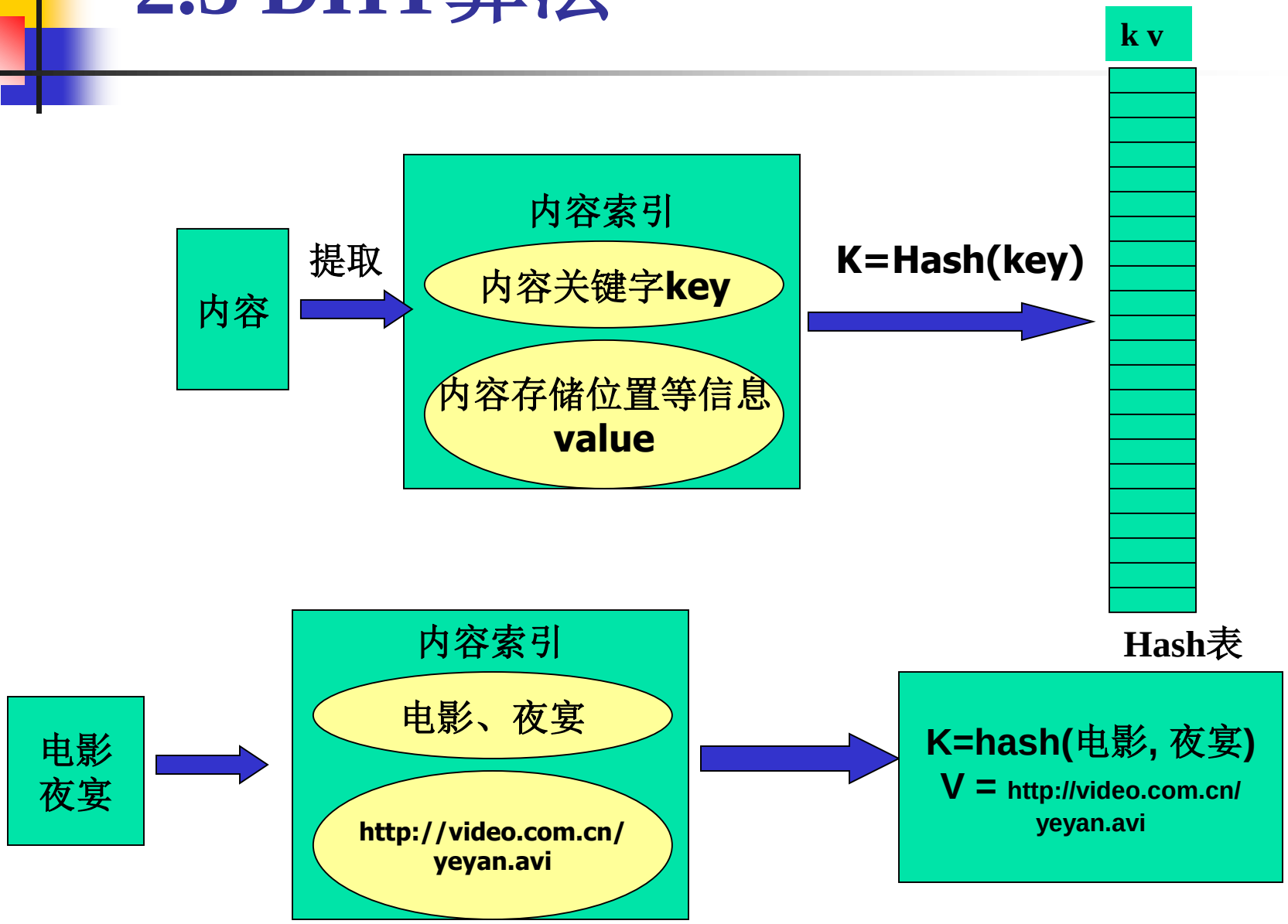
SHA-1(“202.38.64.1”)
=24b92cb1d2b81a47472a93d06af3d85a42e463ea

SHA-1(“202.38.64.2”)
=e1d9b25dee874b0c51db4c4ba7c9ae2b766fbf27

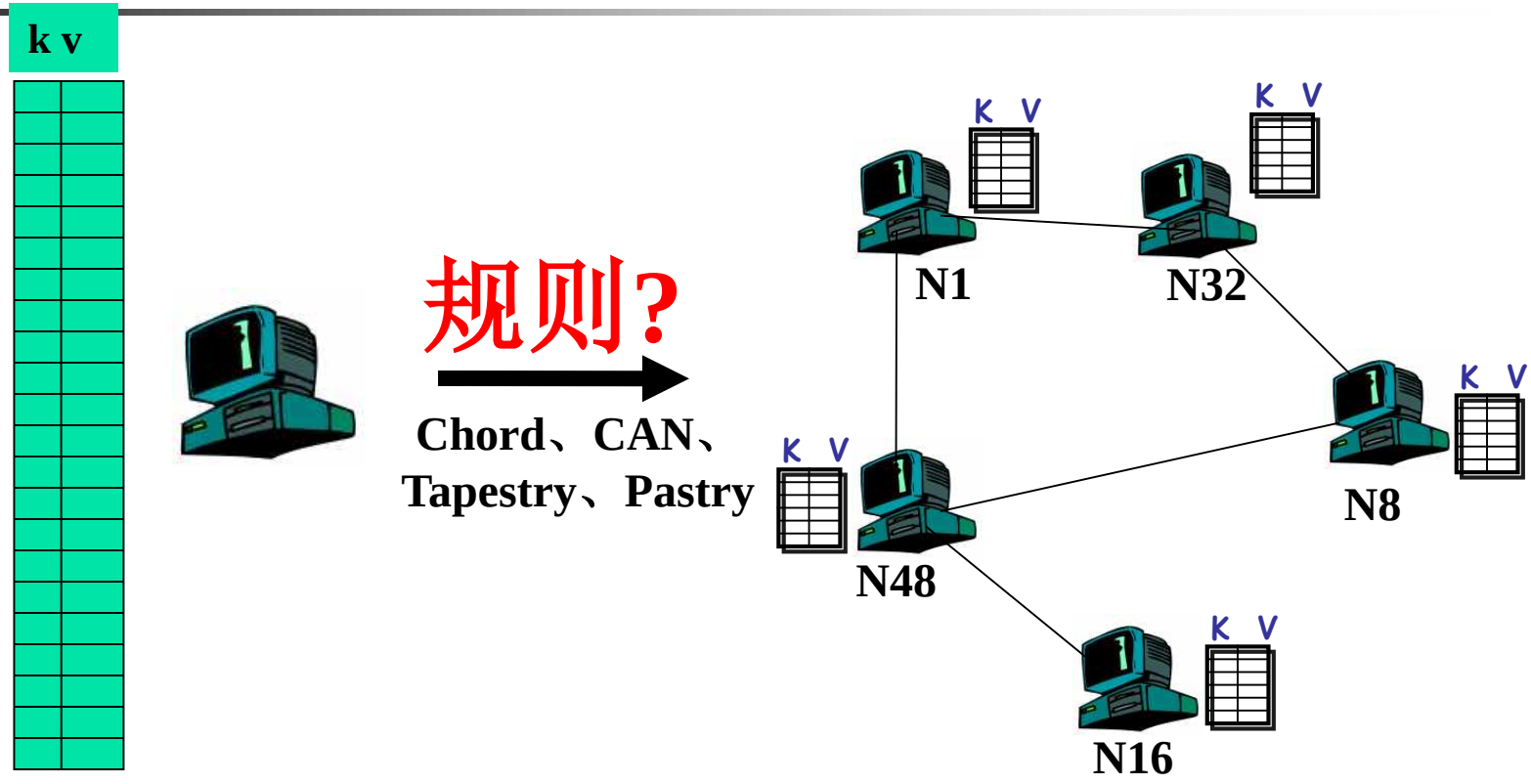
2.3 DHT算法

- 将内容索引抽象为<K, V>对
  - K是内容关键字的Hash摘要：K = Hash(key)
  - V是存放内容的实际位置，例如节点IP地址等
- 所有的<K, V>对组成一张大的Hash表，该表存储了所有内容的信息
- 每个节点都随机生成一个标识(ID)，把Hash表分割成许多小块，按特定规则（即K和节点ID之间的映射关系）分布到网络中去，节点按这个规则在应用层上形成一个结构化的重叠网络
- 给定查询内容的K值，可以根据K和节点ID之间的映射关系在重叠（Overlay）网络上找到相应的V值，从而获得存储文件的节点IP地址

2.3 DHT算法


k v
内容
提取
内容索引
内容关键字key
内容存储位置等信息
value
K=Hash(key)
Hash表
电影
夜宴
内容索引
电影、夜宴
http://video.com.cn/
yeyan.avi
K=hash(电影, 夜宴)
V = http://video.com.cn/
yeyan.avi

2.3 DHT算法



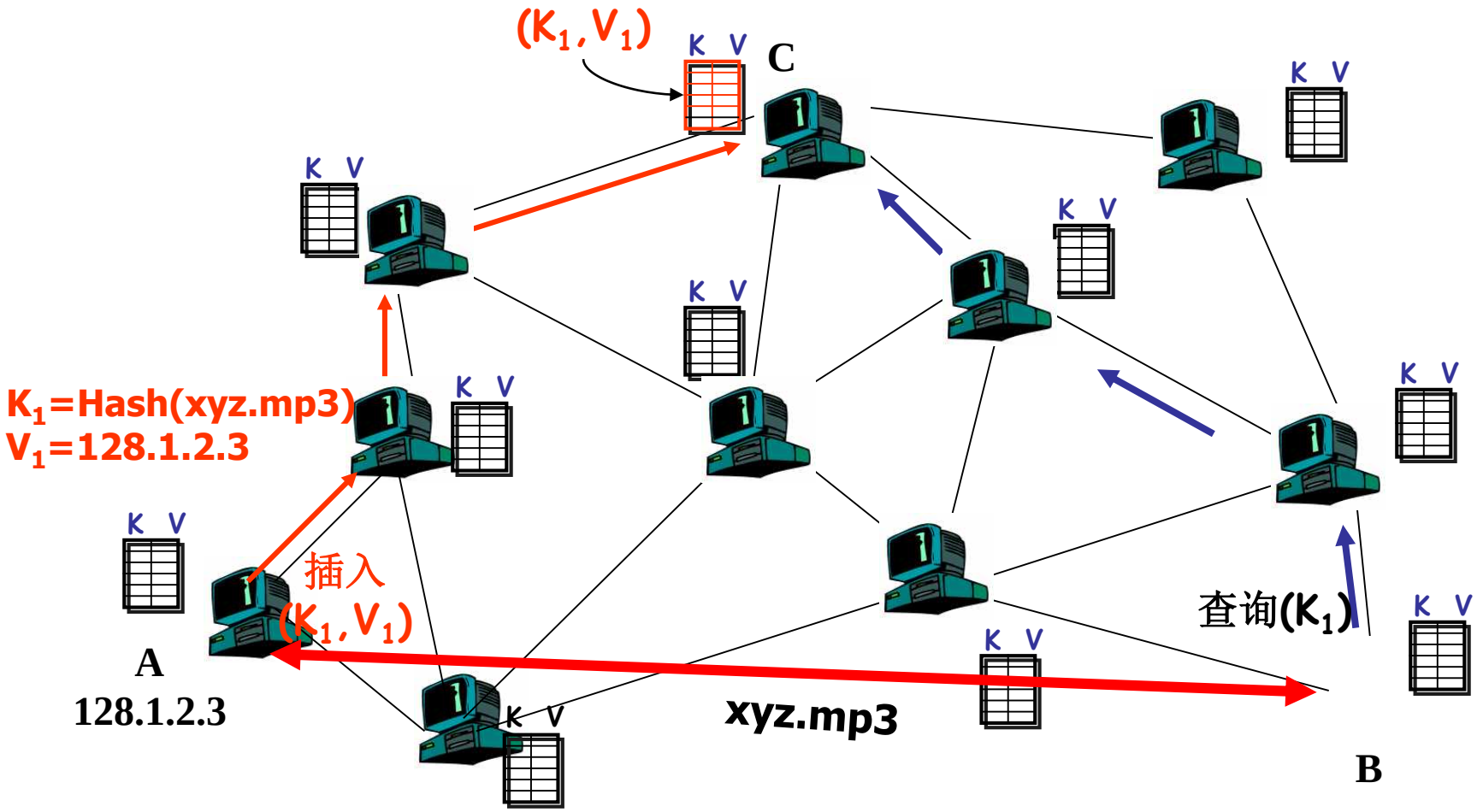
a. Hash表

b. 分布式Hash表

在许多情况下，节点ID为节点IP地址的Hash摘要

2.3 DHT算法

- 索引发布和内容定位



2.3 DHT算法

- 定位(Locating)
  - 节点ID和其存放的<K, V>对中的K存在着映射关系，因此可以由K获得存放该<K, V>对的节点ID
- 路由(Routing)
  - 在覆盖网上根据节点ID进行路由，将查询消息最终发送到目的节点。每个节点需要到其邻近节点的路由信息，包括节点ID、IP等

2.3 DHT算法

- 网络拓扑
  - 拓扑结构由节点ID和其存放的<K, V>对中的K之间的映射关系决定
  - 拓扑动态变化，需要处理节点加入/退出/失效的情况

在重叠网上节点始终由节点ID标识，并且根据ID进行路由

3 代表性DHT算法

- **Chord**
- **Pastry**
- **CAN**

基于分布式Hash表

(DHT： Distributed Hash Table)

结构化P2P:

直接根据查询内容的关键字定位其索引的存放节点

3.1 Chord：概述

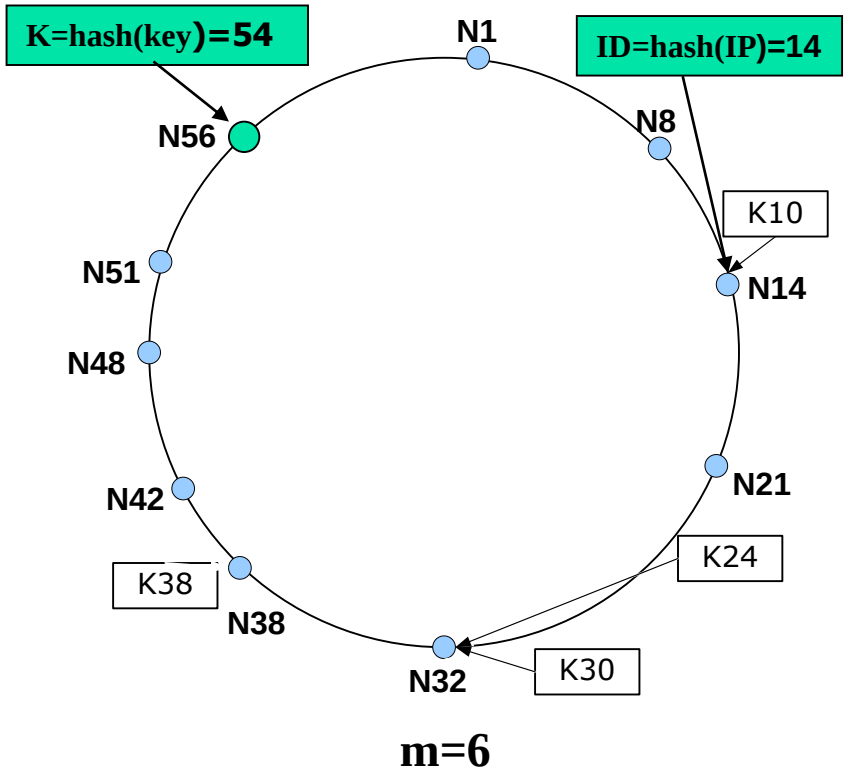
- UC Berkeley和MIT共同提出
- 采用环形拓扑(Chord环)
- 其核心思想就是要解决在P2P应用中遇到的基本问题：如何在P2P网络中找到存有特定数据的节点
- Chord使用一致性哈希作为哈希算法，在Chord协议中将其规定为SHA-1。

3.1 Chord：概述

- 应用程序接口
  - **Insert(K, V)：将<K, V>对存在放到节点ID为Successor(K)上**
  - **Lookup(K)：根据K查询相应的V**
  - **Update(K, new_V)：根据K更新相应的V**
  - **Join(NID)：节点加入**
  - **Leave()：节点主动退出**

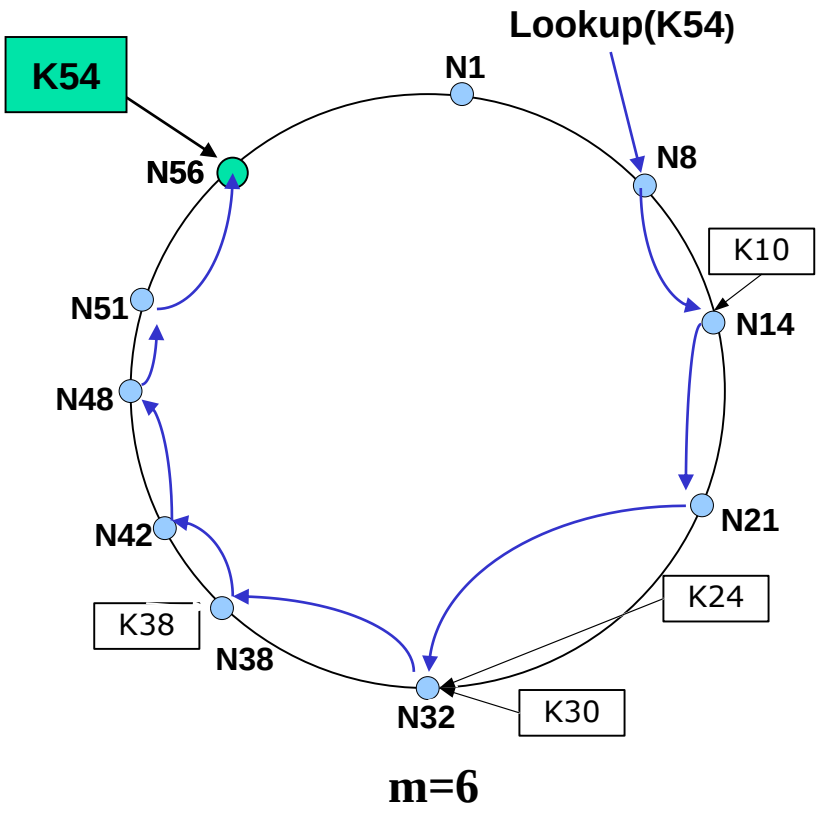
(1) Chord：Hash表分布规则

- Hash算法：SHA-1
- Hash节点IP地址－>m位节点ID(表示为NID)
- Hash内容关键字－>m位K(表示为KID)
- 节点按ID从小到大顺序排列在一个逻辑环上
- <K, V>存储在后继节点上
  - Successor(K)：从K开始顺时针方向距离K最近的节点

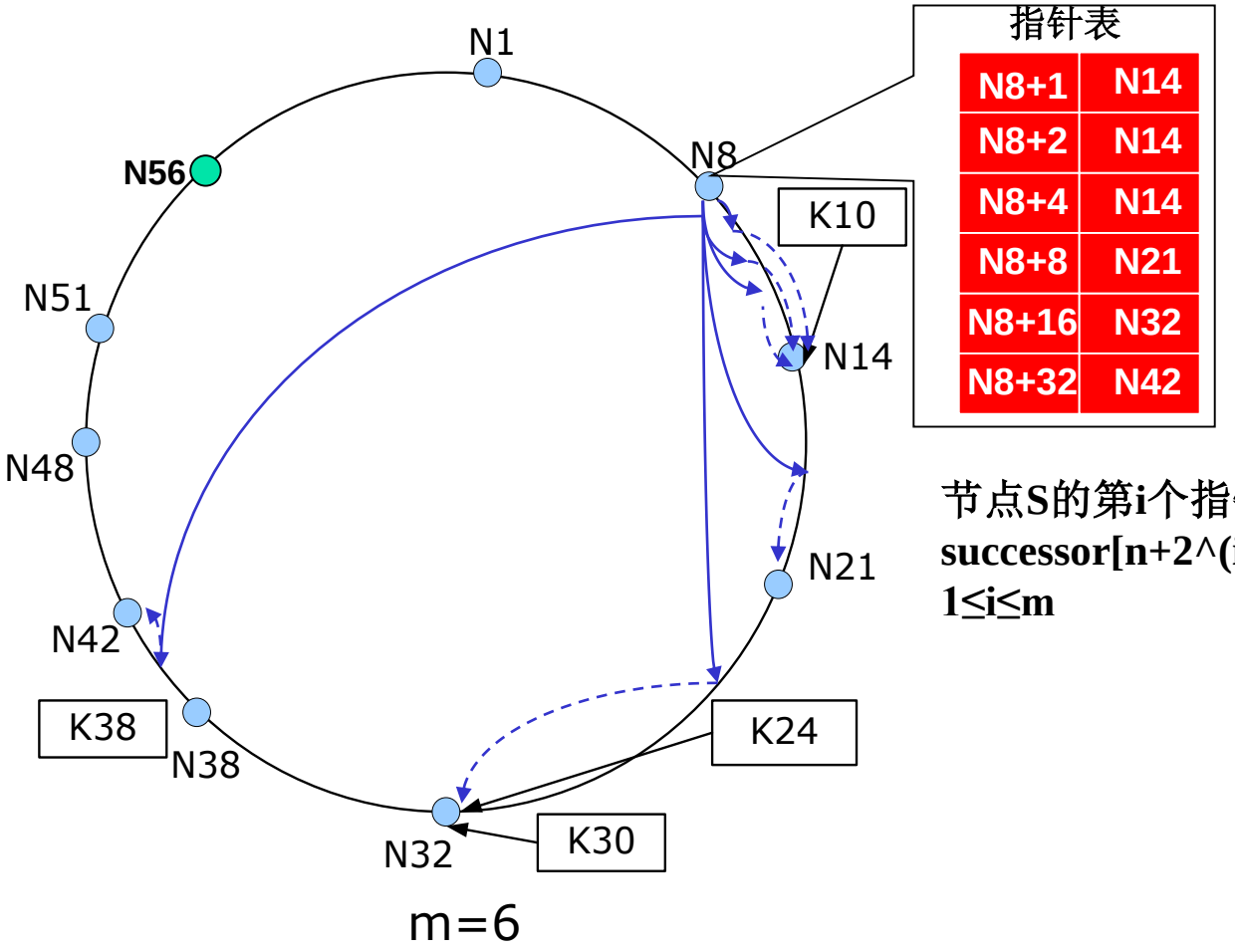


(2) Chord：简单查询过程

- 每个节点仅维护其后继节点ID、IP地址等信息
- 查询消息通过后继节点指针在圆环上传递
- 直到查询消息中包含的K落在某节点ID和它的后继节点ID之间
- 速度太慢 O(N)，N为网络中节点数

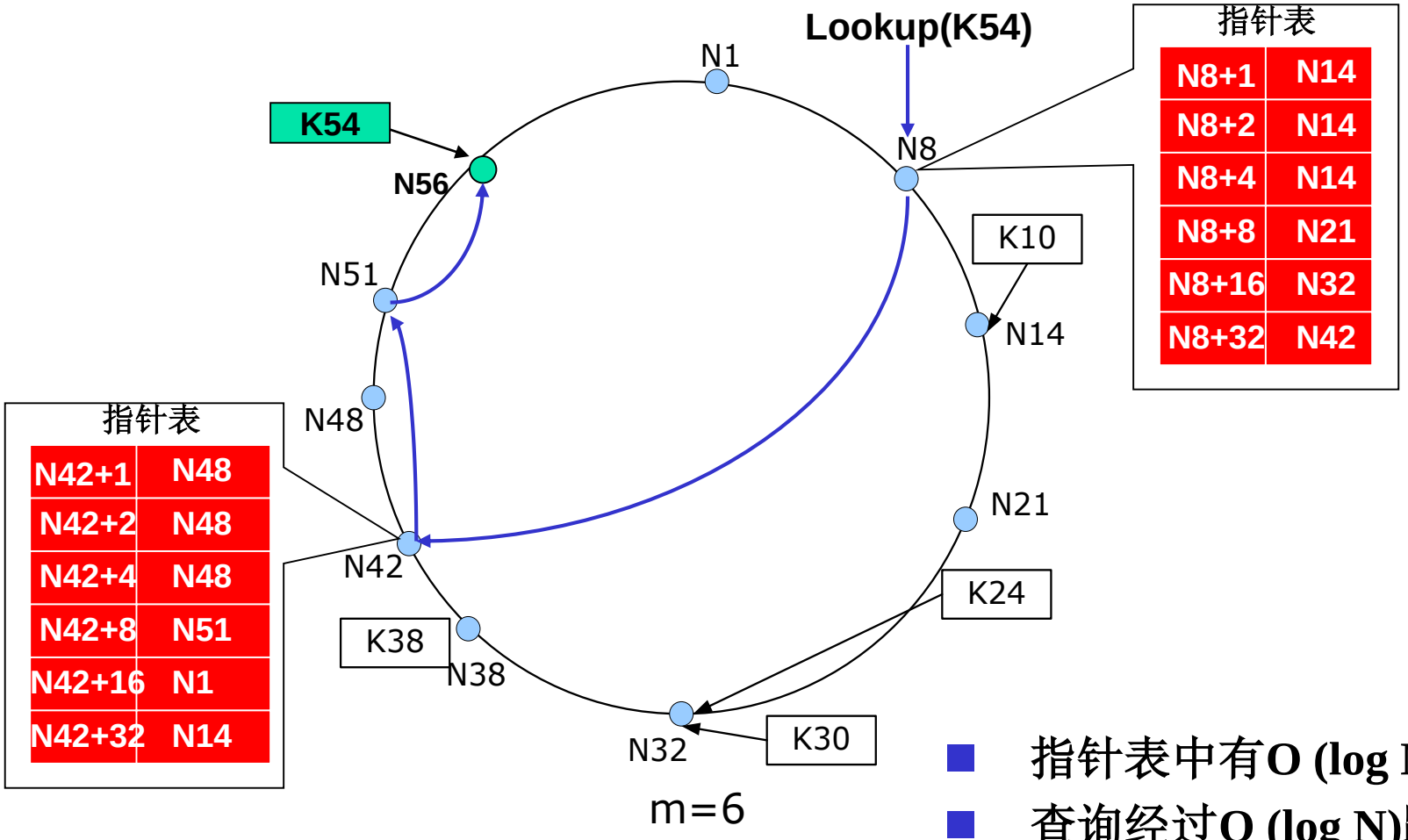


(3) Chord：指针表



节点S的第i个指针
successor[n+2^(i-1)], 1≤i≤m

(4) Chord：基于指针表的扩展查找过程



- 指针表中有O (log N)个节点
- 查询经过O (log N)跳

(5) Chord：网络波动(Churn)

- **Churn**由节点的加入、退出或者失效所引起
- 每个节点都周期性地运行探测协议来检测新加入节点或退出/失效节点，从而更新自己的指针表和指向后继节点的指针

(6) Chord：节点加入

- 新节点N事先知道某个或者某些节点，并且通过这些节点初始化自己的指针表，也就是说，新节点N将要求已知的系统中某节点为它查找指针表中的各个表项
- 在其他节点运行探测协议后，新节点N将被反映到相关节点的指针表和后继节点指针中
- 新结点N的第一个后继结点将其维护的小于N节点的ID的所有K交给该节点维护

(7) Chord：节点退出/失效

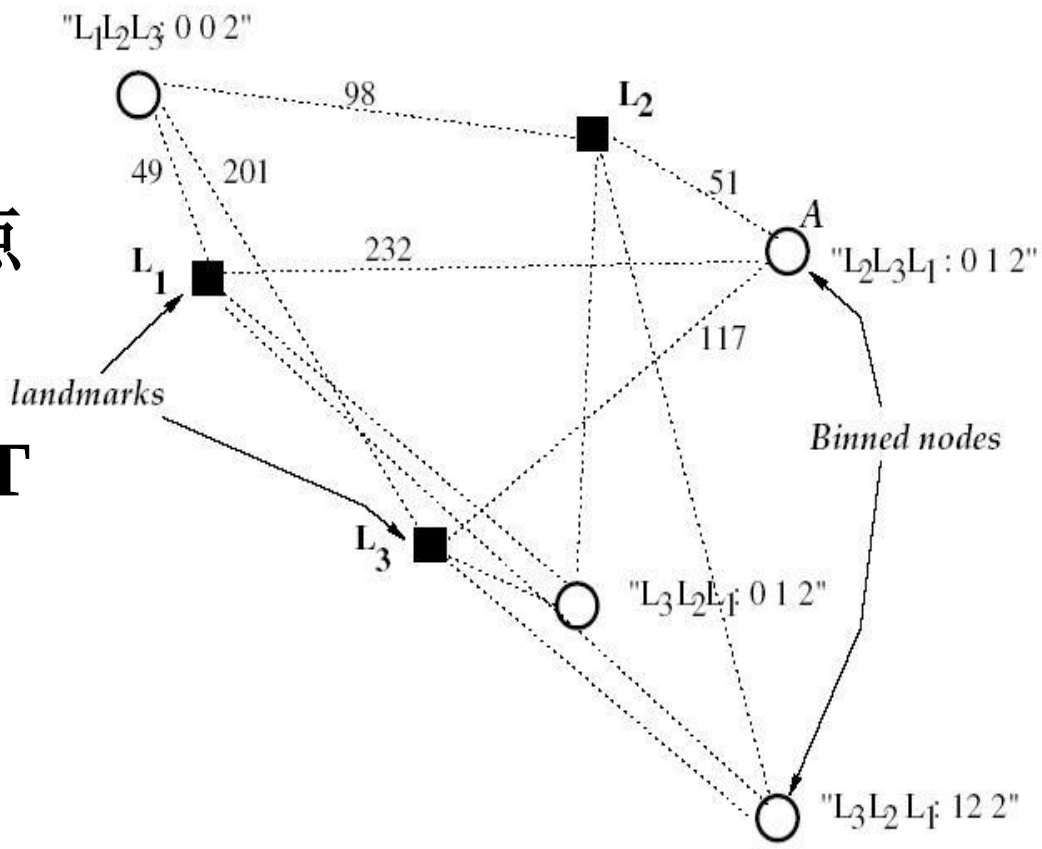
- 当Chord中某个节点M退出/失效时，所有在指针表中包含M的节点将相应指针指向大于M节点ID的第一个有效节点，即节点M的后继节点
- 为了保证节点M的退出/失效不影响系统中正在进行的查询过程，每个Chord节点都维护一张包括r个最近后继节点的后继列表。如果某个节点注意到它的后继节点失效了，它就用其后继列表中第一个正常节点替换失效节点

(8) Chord：拓扑失配问题

- O(LogN)逻辑跳数，但是每一逻辑跳可能跨越多个自治域，甚至是多个国家的网络
- 覆盖网络与物理网络脱节
- 实际的寻路时延较大

(8) Chord：拓扑失配问题

- 提取物理网络的拓扑信息改造Chord
  - 存在w个界标站点
  - 每个节点测量它到w个界标的RTT
  - 将RTT递增排列
  - 具有相同RTT序列的节点在物理网络上临近



(9) Chord：小结

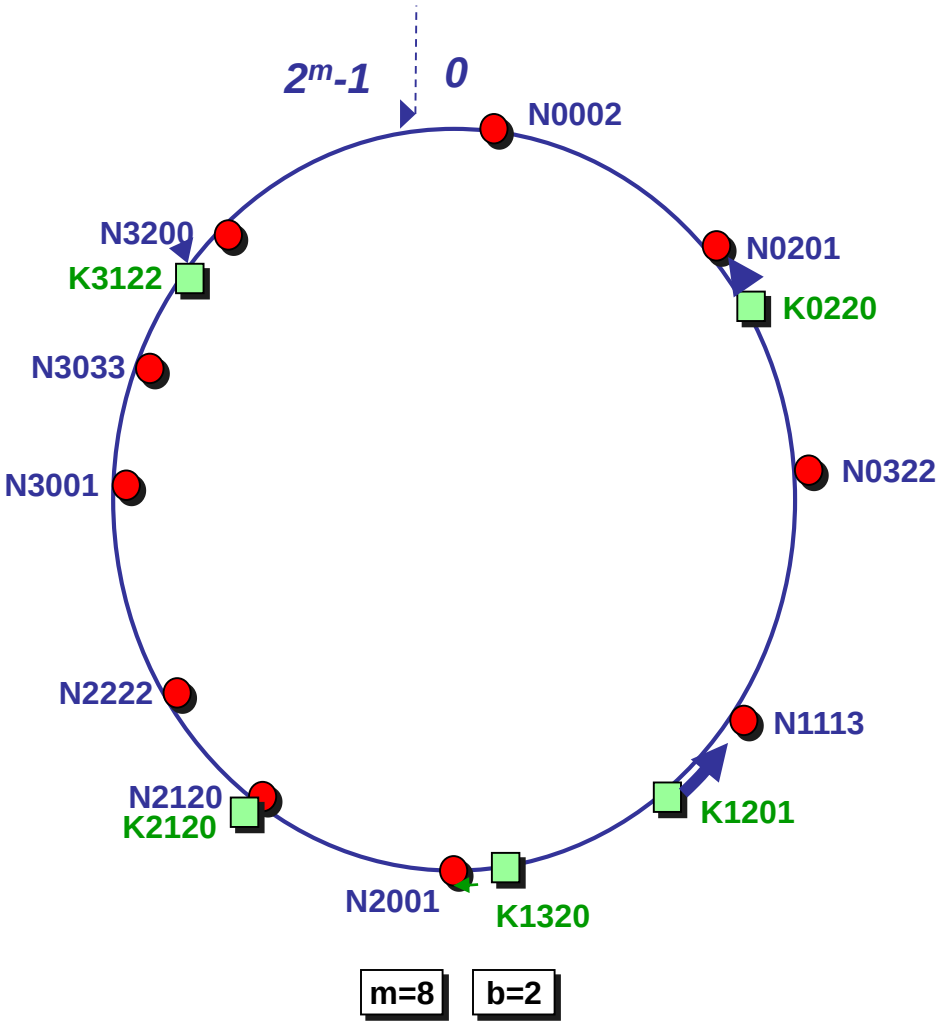
- 算法简单
- 负载平衡：所有的节点以同等的概率分担系统负荷，从而避免某些节点负载过大
- 可扩展：查询过程的通信开销和节点维护的状态随着系统总节点数增加成对数关系(O (log N)数量级)
- 可用性：要求节点根据网络变化动态更新查询表，能够及时恢复路由关系，使得查询可靠地进行。
- 拓扑失配问题

3.2 Pastry：概述

- Microsoft研究院和Rice大学共同提出
- 考虑网络的本地性，解决物理网络和逻辑网络的拓扑失配问题
  - 基于应用层定义的邻近性度量，例如IP路由跳数、地理距离、往返延时等
- 节点ID分布采用环形结构

(1) Pastry： Hash表分布规则

- Hash算法：SHA-1
- Hash节点IP地址－>m位节点ID(表示为NID)
- Hash内容关键字－>m位K(表示为KID)
- NID和KID是以2^b为基的数，共有m/b个数位
  - m/b是一个配置参数，一般为4
- 节点按ID从小到大顺序排列在一个逻辑环上
- <K, V>存储在NID与KID数值最接近的节点上



(2) Pastry：节点维护状态表

- 每个节点维护一个状态表
  - 路由表
  - 邻居节点集
  - 叶子节点集

(2) Pastry：节点维护状态表

- 路由表R
  - 包括 m/b行，每行包括2^b 个表项
  - 第n行与节点ID的前n-1个数位相同，第n个数位不同，取值从0到2^b -1，也称n-1数位前缀相同
  - 表中的每项包含节点ID，即IP地址等
  - 根据邻近性度量选择距离本节点近的节点
  - b过大，节点要维护的路由表大，但存储的邻居节点多，在转发时更为精确，b的选择反映了路由表大小和路由效率之间的折衷

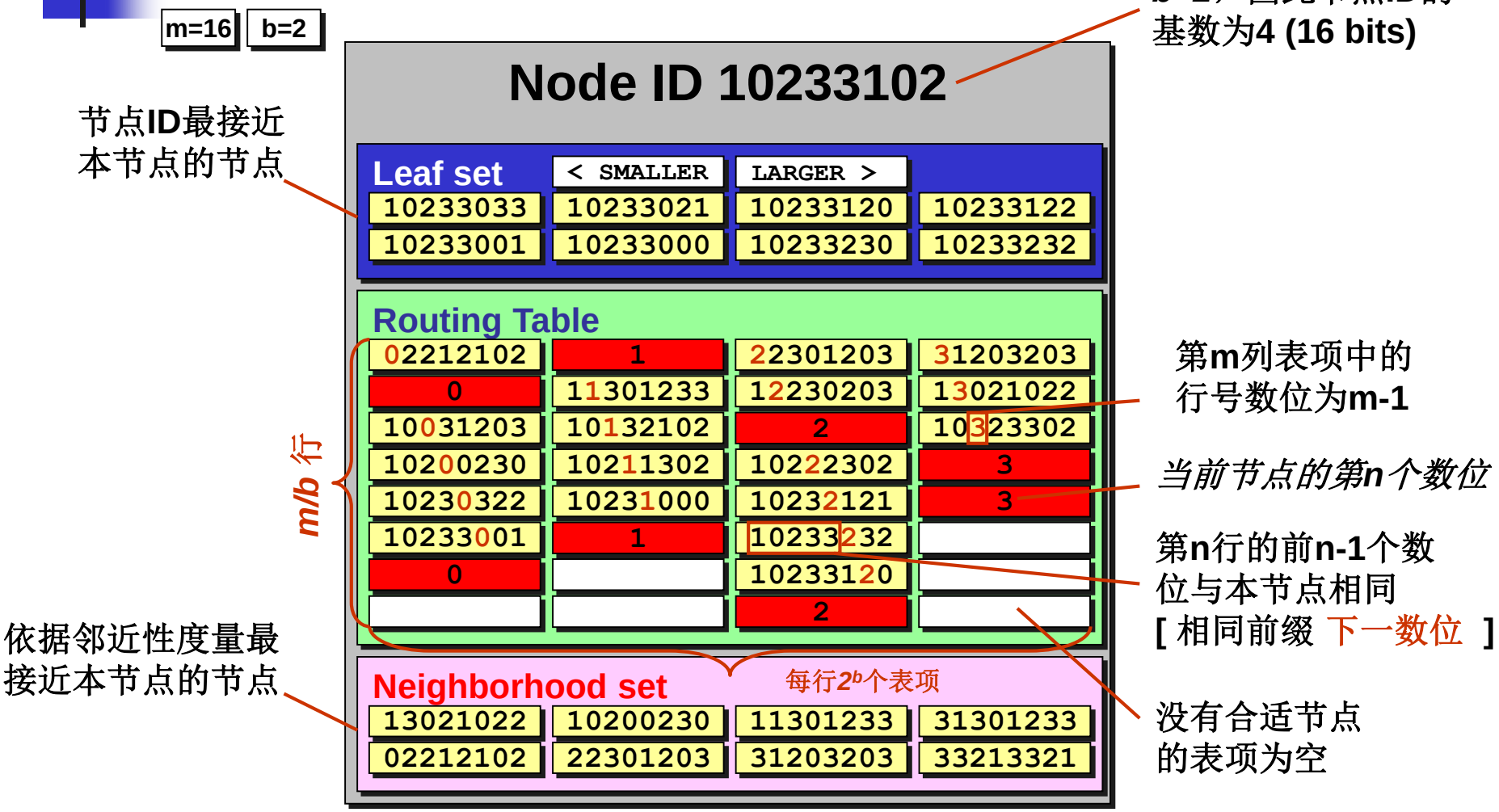
(2) Pastry：节点维护状态表

- 邻居节点集M
  - 存放在真实网络中与当前节点“距离”最近的|M|个节点的信息
  - “距离” 类似IP路由协议中的距离， 考虑转发跳数、传输路径带宽、QoS等综合因素后所得的转发开销
  - |M|的典型值为2^b或者2×2^b
  - 邻居节点集通常不用于路由查询消息，而是用来维护本地性

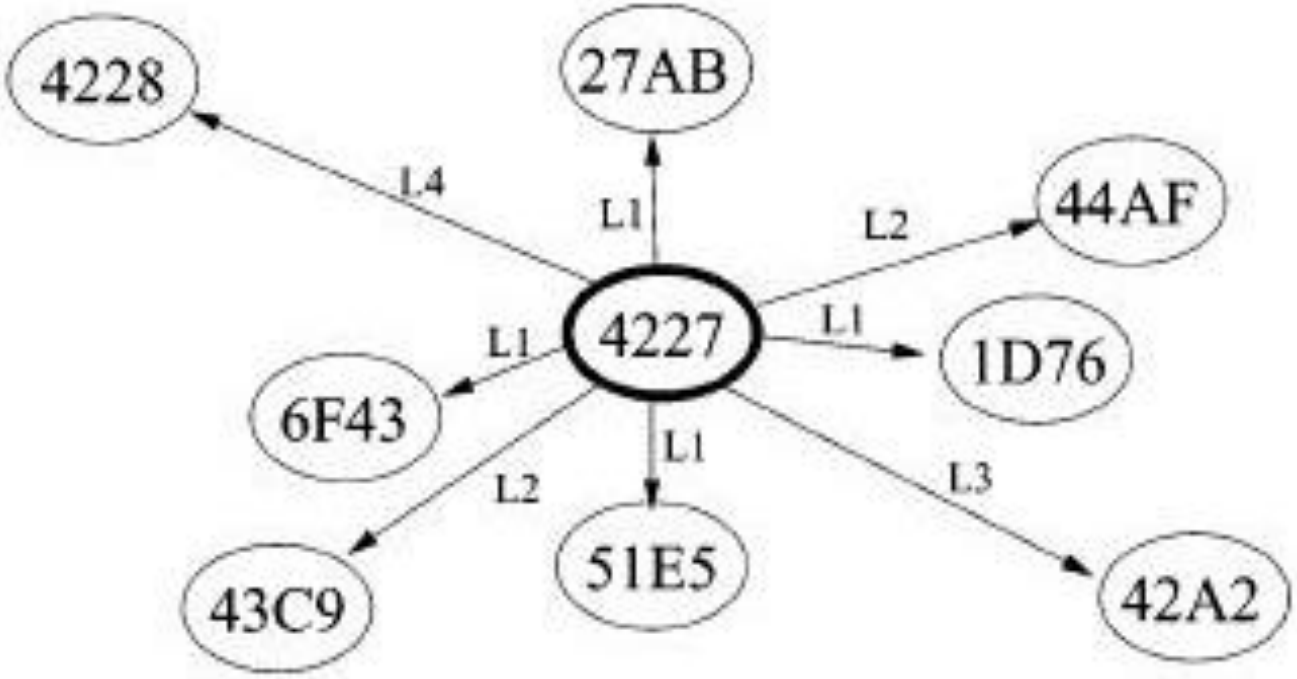
(2) Pastry：节点维护状态表

- 叶子节点集L
  - 存放在键值空间中与当前节点距离最近的|L|个节点的信息，其中各有一半的节点标识大于或小于当前节点
  - |L|的典型值为2^b或者2×2^b
  - 路由时，首先检查叶子节点集

(2) Pastry：节点维护状态表



(2) Pastry：节点维护状态表



(3) Pastry：查询过程

- 当一个K为D的查询消息到达节点A

1. 节点A首先看D是否在当前节点的叶子节点集中，如果是，则查询消息直接被转发到目的节点，也就是叶子节点集中节点ID与D数值最接近的那个节点（有可能就是当前节点），否则进行下一步

(3) Pastry：查询过程

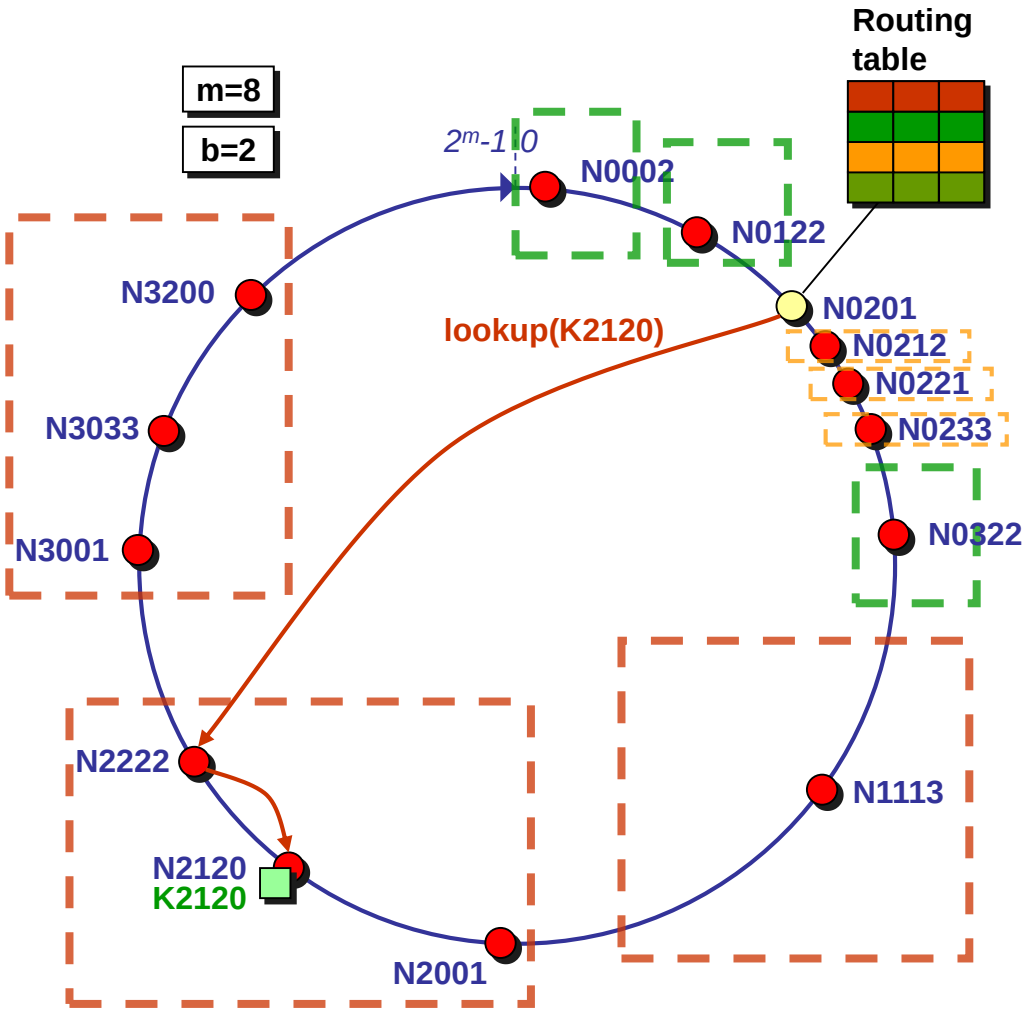
2. 在路由表中根据最长前缀优先的原则选择一个节点作为路由目标，转发路由消息，如果该表项不为空，则将查询消息直接转发到该节点，否则进行下一步

3. 如果不存在这样的节点，当前节点将会从其维护的所有邻居节点集合中选择一个距离消息键值最近的节点作为转发目标。

路由查询消息的逻辑跳数：$O(\log_{2^b} N)$

(4) Pastry：节点状态表和查询

- 节点路由表R中的每项与本节点具有相同的n数位长度前缀，但是下一个数位不同
- 例如，对于节点N0201：
  - N-: N1???, N2???, N3???
  - N0: N00??, N01??, N03??
  - N02: N021?, N022?, N023?
  - N020: N0200, N0202, N0203
- 当有多个节点时，根据邻近性度量选择最近的节点
  - 维持了较好的本地性



(5) Pastry：节点加入

- 初始化状态表
  1. 新节点开始时知道一个根据邻近性度量接近自己的节点A
     - 节点A可以通过使用扩展环IP组播等机制自动定位，或由系统管理员通过其他手段获得
  2. 新节点通过运行SHA-1算法计算自己的IP地址(或者public key)的摘要得到节点ID为X

(5) Pastry：节点加入

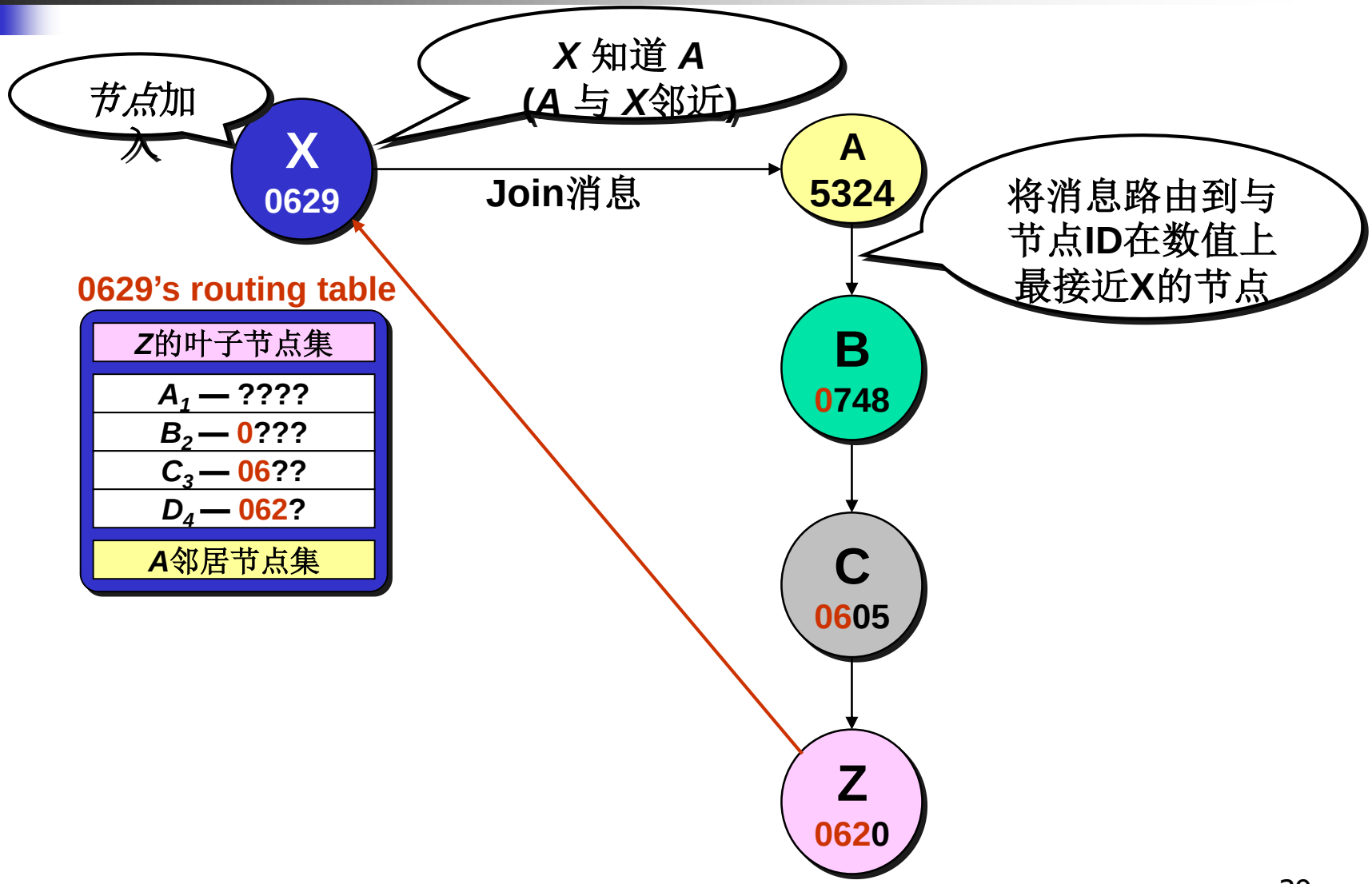
3. 节点X向节点A发送K为X的Join消息，Pastry将该消息路由到节点ID在数值上最接近X的节点Z

- 接收到Join消息的节点，包括A、Z，以及A到Z路径上所有的节点将发送它们的状态表给X，X检查这些信息，并且可能从其他的节点请求状态，然后节点根据下面的过程初始化状态表:
  - 由于A与X在邻近性度量上接近，所以使用A的邻居节点集来初始化X的邻居节点集
  - 由于Z的节点ID与X最相近，因此使用Z的叶子节点集来初始化X的叶子节点集
  - X将Join消息经过的第i个节点的路由表的第i行作为自己路由表的第i行，因为Join消息经过的第i个节点与X的前i个数位相同

(5) Pastry：节点加入

- 向其他相关节点通告自己的到来
  - 新节点向邻居节点集、叶子节点集和路由表中的每个节点发送自己的状态，以更新这些节点的状态表

(5) Pastry：节点加入



(6) Pastry：节点退出/失效

- 叶子节点集L中的节点失效：联系L中失效节点一边具有最大索引的存活节点（即节点ID最小或者最大的存活节点），并且请求该节点的叶子节点集
  - 除非|L|/2个节点同时失效，否恢复过程始终是有效的
  - 失效检测：和叶子节点集中的节点周期性交换存活消息

(6) Pastry：节点退出/失效

- 邻居节点表M中的节点失效：向M中的其它节点请求邻居节点表，检查每个新发现节点的距离（根据邻近性度量），然后更新自己的邻居节点表
  - 失效检测：和邻居节点表中的每个节点周期性的联系，以确认节点存活

(6) Pastry：节点退出/失效

- 路由表R中的节点失效：如果失效节点对应的表项为R^d_l(第l行第d列)，则联系同一行中的R^i_l，$i \neq d$所指向的存活节点并且获取该节点的R^d_l表项，如果l行中没有存活节点，则从下一行选择一个节点
  - 失效检测：和路由表中的节点联系(例如发送查询消息)如果无反应则检测到节点失效

(7) Pastry：路由本地性

- 根据邻近性度量为消息选择一条好的路由
  - 邻近性度量包括IP路由跳数、地理距离、往返延时等
  - 应用层为每个节点提供了根据邻近性度量确定到其他节点距离的功能，例如traceroute等

(7) Pastry：路由本地性

- 新节点加入过程保持了本地性
  - 首先：A必定与X相接近
  - A路由表中第1行表项为A1，而A与X接近，因此A1可作为X路由表中第一行表项X1
  - 由于节点路由表中的下一行节点的可选择集合指数递减，因此B2中的节点到B的距离要比A到B的距离大得多(B在A1中)，因此B2可作为X2，依此类推，C3可作为X3
  - X向路由表和邻居节点集中的每个节点请求状态，并且使用更近的节点来更新自己的状态
    - 由于邻居节点集根据邻近性度量而不是节点ID前缀来维护节点信息，因此在此过程中发挥重要作用

(7) Pastry：路由本地性

- 实践中Pastry保持了良好的本地性
  - 伸展度(Stretch)大约为2~3

伸展度 = 逻辑网络的路由延时/IP网络的单播路由延时

(8) Pastry：总结

- 逻辑网络路由跳数$O(log_{2^b} N)$
- 路由表开销$log_{2^b} N * 2^b$
- 路由本地性：状态表（路由表、邻居节点集、叶子节点集）中的表项选择在邻近性度量上与本节点相近的节点
- 稳健性：只有在|L|/2个叶子节点完全失效时才会路由失败

3.3 CAN(Content Addressable Network)

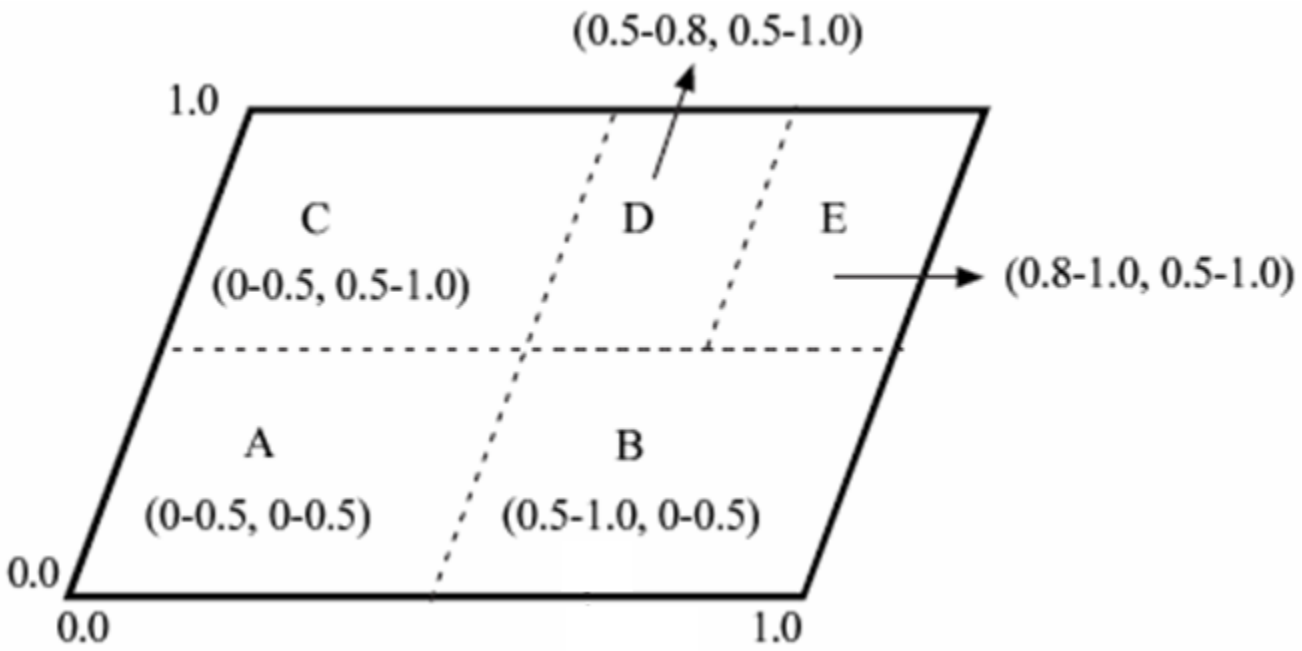
- 内容寻址网络，UC Berkeley提出
- 实现了文件索引和存放位置的有效映射，不需要任何形式的中央控制点
- 节点只需要维护少量的控制状态而且状态数量独立于系统中的节点数量
- 具有完全自组织和分布式的结构，并且有良好的可扩展性和容错性。

3.3 CAN(Content Addressable Network)

- 节点ID分布在d维笛卡尔坐标空间，d是一个由系统规模决定的常量。坐标空间完全是逻辑的，和任何物理坐标没有任何关系
- 在任何时候，整个坐标空间动态地分配给系统中的所有节点，每个节点负责维护独立的互不相交的一块区域
- 每个节点要了解并维护相邻区域中节点的IP地址，用这些邻居信息构成自身的坐标路由表。有了这张表，CAN可以在坐标空间中任意两点间进行寻路。

(1) CAN：Hash表分布规则

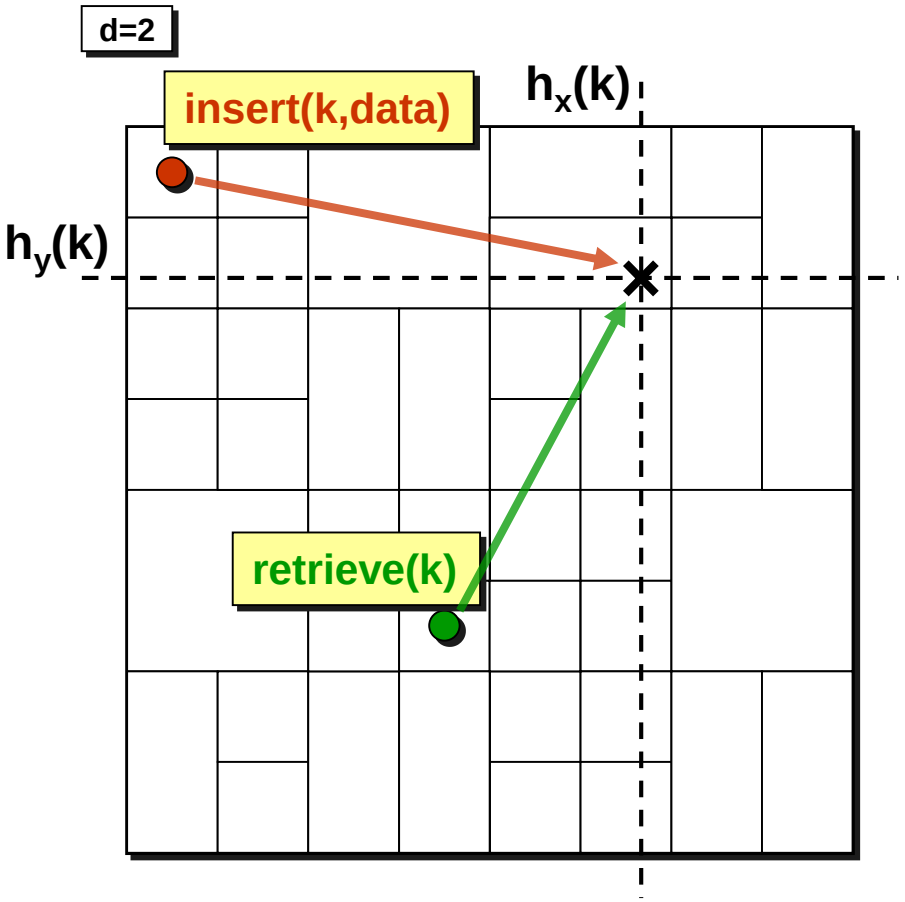
两个节点互为邻居：在d维坐标空间中，两个节点维护的区域在d-1维的坐标上有重叠而在剩下的一维坐标上相互邻接，邻居D和E，D与A不是邻居



2维的[0, 1]×[0, 1]的笛卡儿坐标空间划分成五个节点区域

(1) CAN：Hash表分布规则

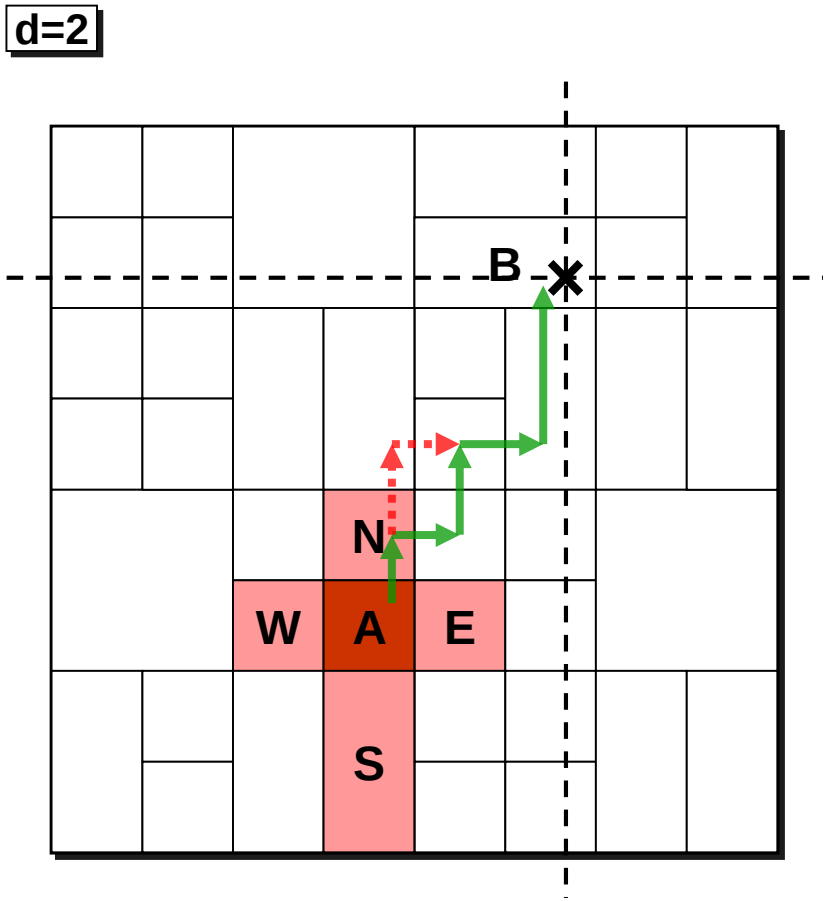
- 每个节点都维护d维笛卡尔坐标空间中的一块区域
- 新加入节点时划分坐标空间
- 对于每个<K, V>，通过Hash函数将K映射到坐标空间中的某点P，存储在维护该点所在区域的节点上
  - 在每一维都对k进行hash运算



(2) CAN：查询过程

- 节点在坐标路由表中只维护直接邻居信息，包括邻居的IP地址及其维护的区域
- 查询消息沿着坐标空间从发起请求的点到目的点之间的一条路径转发
  - 查询消息路由到能够减少坐标空间距离的邻居节点
  - 有多条路径可以选择，在路由时能够绕开失效节点

(2) CAN：查询过程



(3) CAN：节点加入

- 因为整个CAN空间要分配给系统中现有的全部节点，当一个新的节点加入网络时必须得到自己的一块坐标空间。
- CAN通过分割现有的节点区域实现这一过程。
- 它把某个现有节点的区域分裂成同样大小的两块，自己保留其中的一块而另一块分给新加入的节点。

(3) CAN：节点加入

整个过程分为以下三步：

1. 新加入节点在CAN中查找已经存在的节点，即bootstrap节点
2. 找到一个区域将要被分割的节点，进行空间划分
   - bootstrap节点提供系统中有效的并且可以划分区域的节点A，新节点向节点A发送一个加入消息，该消息经过CAN的路由机制发送到A

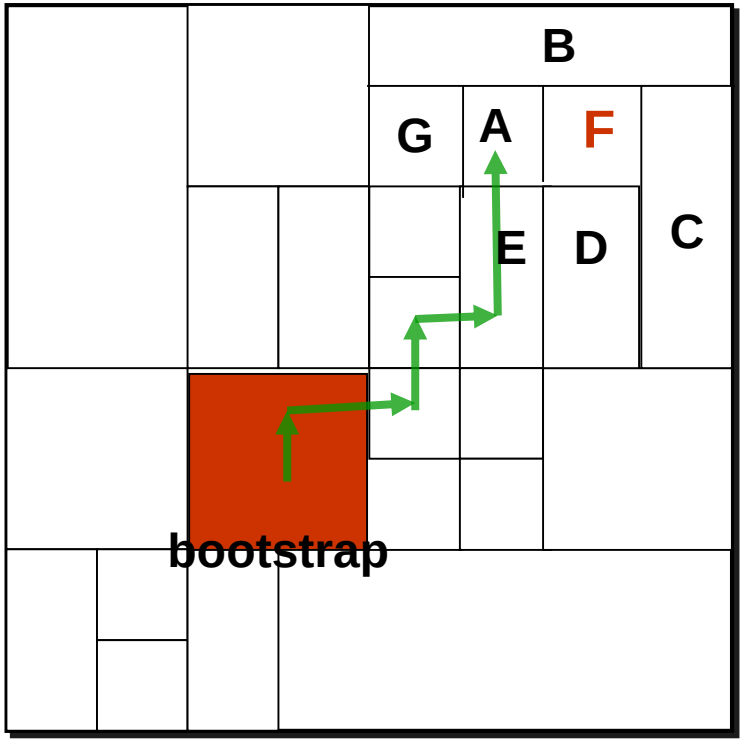
(3) CAN：节点加入

3. 通知被划分节点的邻居节点更新路由表

- 新节点F的邻居表是由节点A的邻居节点的子集合，再加上节点A构成
- 节点A刷新它的邻居节点空间，以删除那些现在已不是邻居节点的节点
- 新节点F发送刷新信息更新邻居节点的坐标路由表

(3) CAN：节点加入

d=2



(4) CAN：节点退出/失效

- 失效检测
  - 每个节点周期性向邻居节点发送更新消息，如果消息中包括自身的区域范围、它的邻居列表以及这些邻居节点负责的区域范围。
  - 如果多次没有接收到某个邻居的更新消息，那么节点就认为这个邻居失效了，这时将启动接管机制。

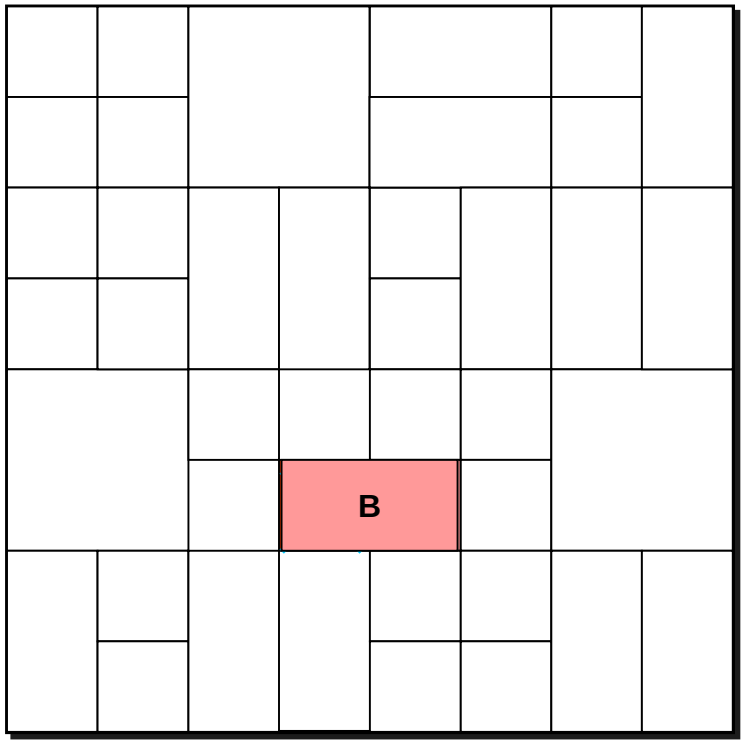
(4) CAN：节点退出/失效

- 接管机制
  - 当节点离开CAN时，必须保证它的区域被系统中剩余的节点接管，即分配给其他仍然在系统中的节点。一般是由某个邻居节点来接管这个区域和所有的索引数据（K，V）对。
  - 接管邻居节点的选择
    - 如果某个邻居节点负责的区域可以和离开节点负责的区域合并形成一个大的区域，那么将由这个邻居节点执行合并操作

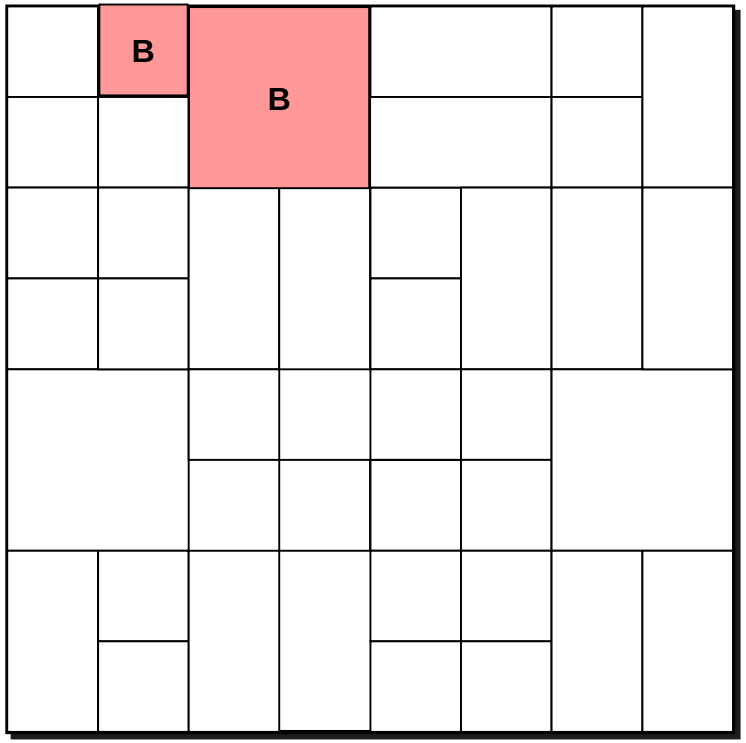
(4) CAN：节点退出/失效

- 否则由邻居节点中区域最小的节点负责该区域
  - 失效节点的每个邻居独立地启动一个时钟，每个时间长短和相应节点负责的区域面积成比例
  - 如果时钟超时，节点将向失效节点的所有邻居节点发送接管消息，该消息中包括它自己的区域面积信息
  - 当某个节点接收到接管消息后，如果它的区域面积比发出消息的节点大，那么它将取消接管操作。否则它将发出自己的取代消息

(4) CAN：节点退出/失效



节点失效后的区间合并



节点失效后由面积最小的邻居节点接管

(5) CAN：负载均衡问题

- 负载均衡
  - 节点负担的面积越大，负载就越重
- 假设整个坐标空间的面积是S，整个空间中共有n个节点，那么理想情况的均衡划分的结果应该是每个节点的面积都是V=S/n
- 采用原始CAN节点加入划分机制，只有43%的节点面积为V

(5) CAN：负载均衡问题

- 解决方案
  - 组播法寻找重负荷节点
    - 新加入系统的节点首先通过引导节点在全网范围内泛洪查找面积最大的节点，对其空间进行划分
  - 逻辑结构自适应调整法
    - 通过目的节点向所有四周邻居进行泛洪，获取面积最大的节点，划分此节点空间
- 缺点：需要泛洪消息，网络开销太大

(6) CAN：总结

- 可扩展性：如d维坐标空间划分成N个相等区域，则
  - 每个节点维护2d个邻居节点的信息
  - 平均路由跳数$(dN^{1/d})/4$，增加维数可减少在逻辑上的路由跳数，但增加了节点维护的状态信息
  - 节点增加时，节点维护的状态信息不变，而路由长度只以$O(N^{1/d})$的数量级增长，可扩展性好
- 稳健性：一个节点的一个或几个邻居节点失效时，它依然可以沿着有效的邻居节点进行寻路
- 负载均衡问题
- 拓扑失配问题

4 基于DHT的结构化P2P比较

	Chord	Pastry	CAN
拓扑结构	节点ID分布在单向环形空间	节点ID分布在单向环形空间，并且表示为以2^b为基的数	节点分布在d维笛卡尔坐标空间
<K, V>存储	储存在K的后继节点即节点ID大于K的第一个节点上	存储在节点ID与K最接近的节点上	通过在每一维对K进行Hash运算将K映射到坐标空间中的一点，存储在维护该点所在区域的节点上
路由查询消息	通过后继节点指针或者指针表找到K的后继节点	比较K和节点ID的前缀，下一跳节点的ID具有更长的前缀或者在数值上更接近K	下一跳节点在坐标空间上距离目标节点更近
节点维护状态	后继节点指针或者指针表：O(logN)	叶子节点集、邻居节点集：2^b 或者$2*2^b$ 路由表：$\log_{2^b}N* 2^b$	坐标路由表：平均2d
路由性能	O(logN)	$O(\log_{2^b}N)$	$(dN^{1/d})/4$
稳健性	维护r个最近的后继节点	只有在|L|/2个叶子节点完全失效时才会路由失败	邻居节点失效时沿其它有效邻居节点路由
路由本地性		状态表(路由表、邻居节点集、叶子节点集)中表项选择在邻近性度量上与本节点相近的节点	

4 几种结构化P2P总结

- 完全分布式，不存在任何中心节点
- 直接根据查询内容的关键字定位其索引的存放节点，查找具有确定性
- 节点失效时表现出很好的健壮性
- 可扩展性好，系统开销小
- 自动配置，不需要手工干预就可自动把加入新节点
- 几个需要研究的问题
  - 模糊查找问题
  - 网络波动(Churn)问题
  - 路由本地性问题
  - 负载均衡问题
  - 安全问题
  - …

5 基于DHT的P2P应用

- DHT接口API
- DHT层次体系结构
- OpenDHT
- Indirect Internet Infrastructure (i3)

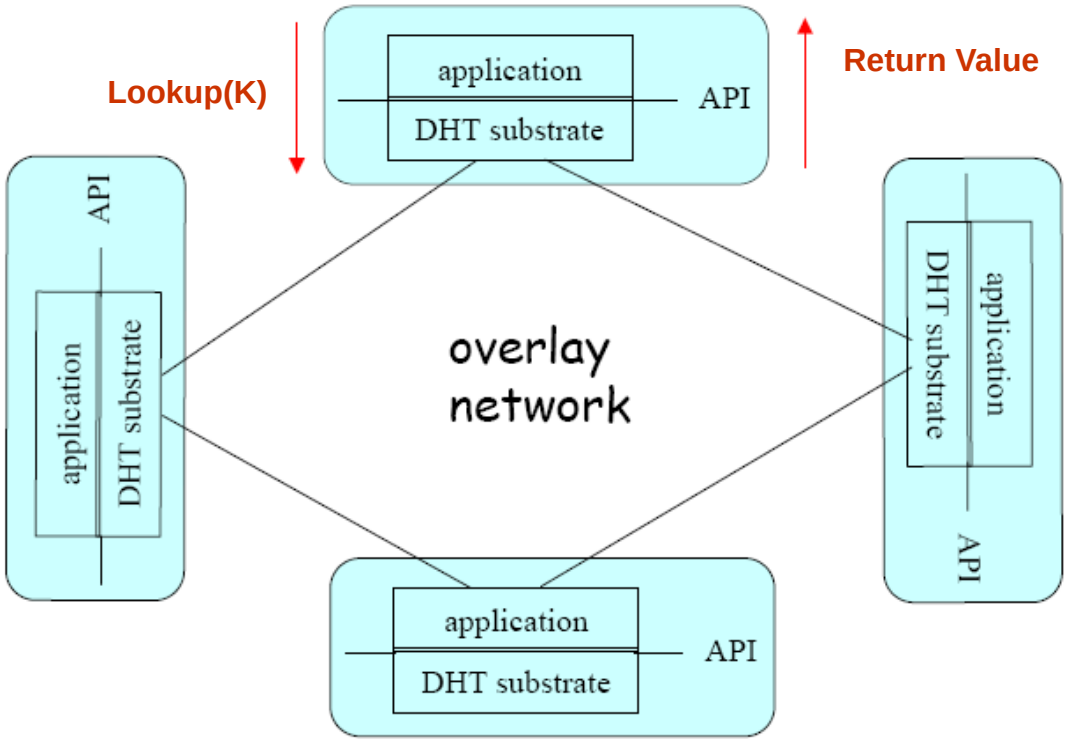
5.1 DHT接口API

- 必须的接口

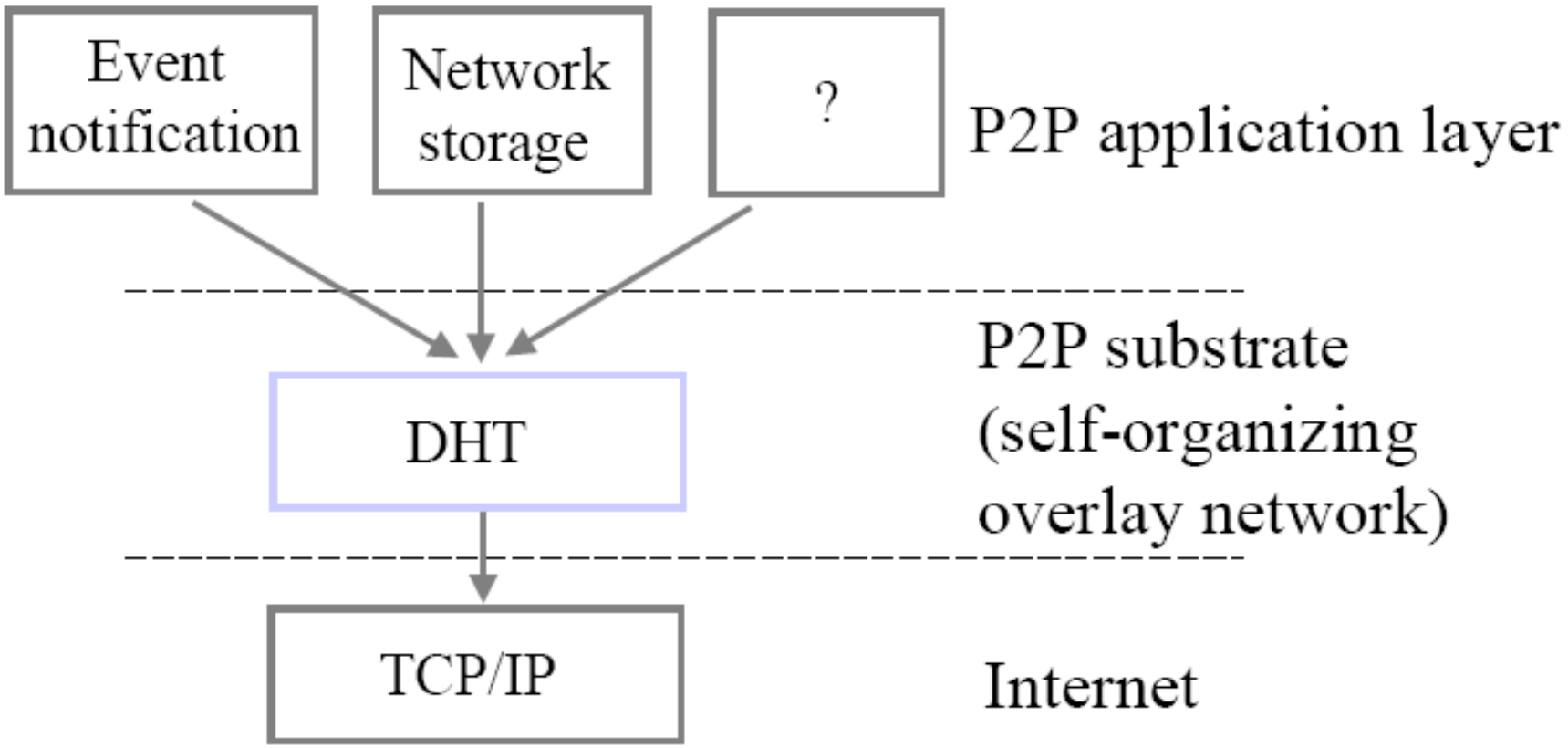
  Inset(K, V)：将<K, V>存储到合适的节点上

  Lookup(K)：获取K所对应的V

- 支持各种应用
  - **K不需要任何语义上的意义**
  - **V与应用相关**
- **具体的API在底层的DHT重叠网络中实现**



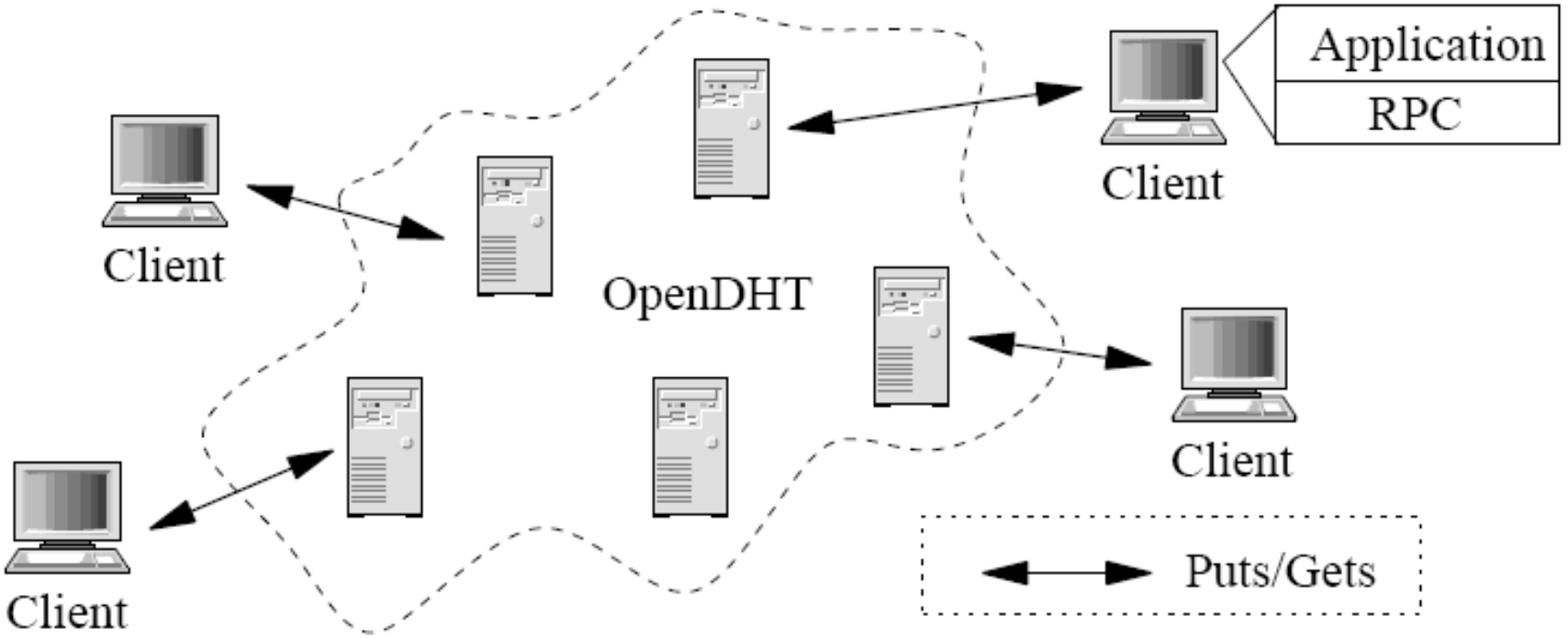
5.2 DHT层次体系结构



5.3 OpenDHT：概述

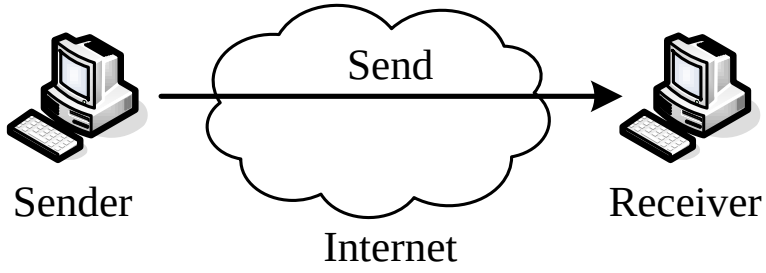
- OpenDHT公共DHT服务平台(http://opendht.org)
  - 基于Bamboo DHT，改写自Pastry
  - 设计原则：易于部署，易于使用
  - 客户端不需要运行DHT软件，而是通过OpenDHT服务平台获取所需的服务，从而实现基于DHT的P2P应用
- 客户端接口API
  - Put(K, V, t)：将<K, V>发布到OpenDHT网络，t为有效期
  - Get(K)：根据K从OpenDHT网络获取<K, V>对

OpenDHT：体系结构

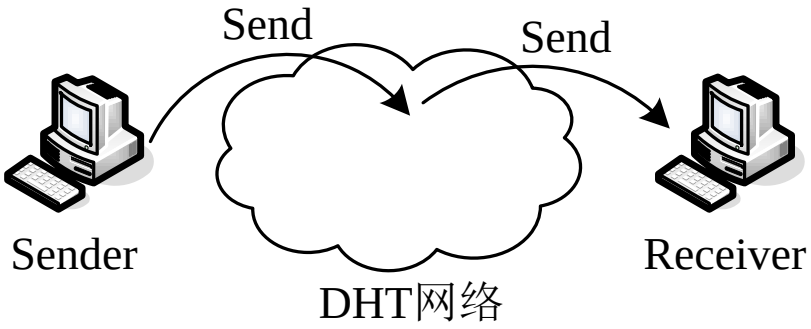


5.4 Indirect Internet Infrastructure (i3)

- 传统的Internet提供端到端通信模型
  - 通信在固定主机间进行
  - 发送主机知道接收主机的IP地址，将分组直接发送给接收主机



- 间接通信模型(i3)
  - 接收主机位置不固定，可能移动
    - **Mobility**
  - 发送主机不知道接收主机的标识
    - **Multicast, Anycast**



(1) i3原理

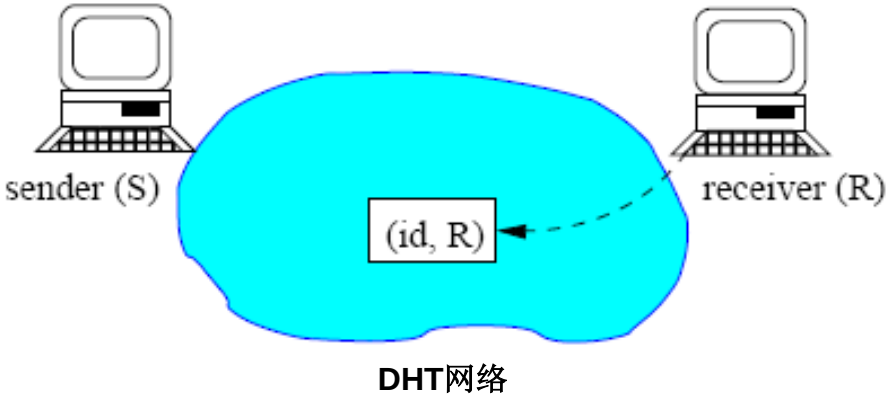
- http://i3.cs.berkeley.edu
- 利用DHT网络提供的索引发布和查询，具有
  - 稳健性
  - 高效性
  - 可扩展性
- DHT网络实现可以使Chord、Pastry、CAN、Tapestry等
  - DHT网络中的每一个节点都为i3服务器

(1) i3原理

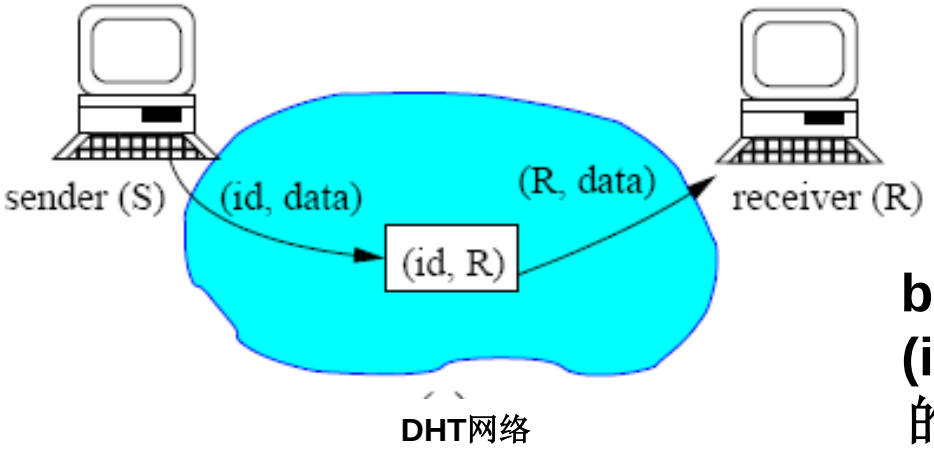
- i3中节点通信过程

1. 每个分组都带有一个标识，接收主机根据标识来获取相应的分组
2. 接收主机在DHT网络中插入trigger (id, R), 其中id为希望接收的分组的标识，R是接收主机的IP地址， (id, R)根据id存储到DHT网络中相应的服务器节点（例如对于chord，(id, R)存储在id的后继节点即节点ID大于id的第一个节点上）
3. 发送主机发送标识为id的分组，该分组以id为k，根据DHT路由查询算法在网络中转发，如果接收到该分组的服务器注册了标识为id的trigger，则将分组发送给trigger中指定的接收主机IP

(1) i3原理



a. 接收主机**R**在**DHT**网络中插入**trigger(id, R)**



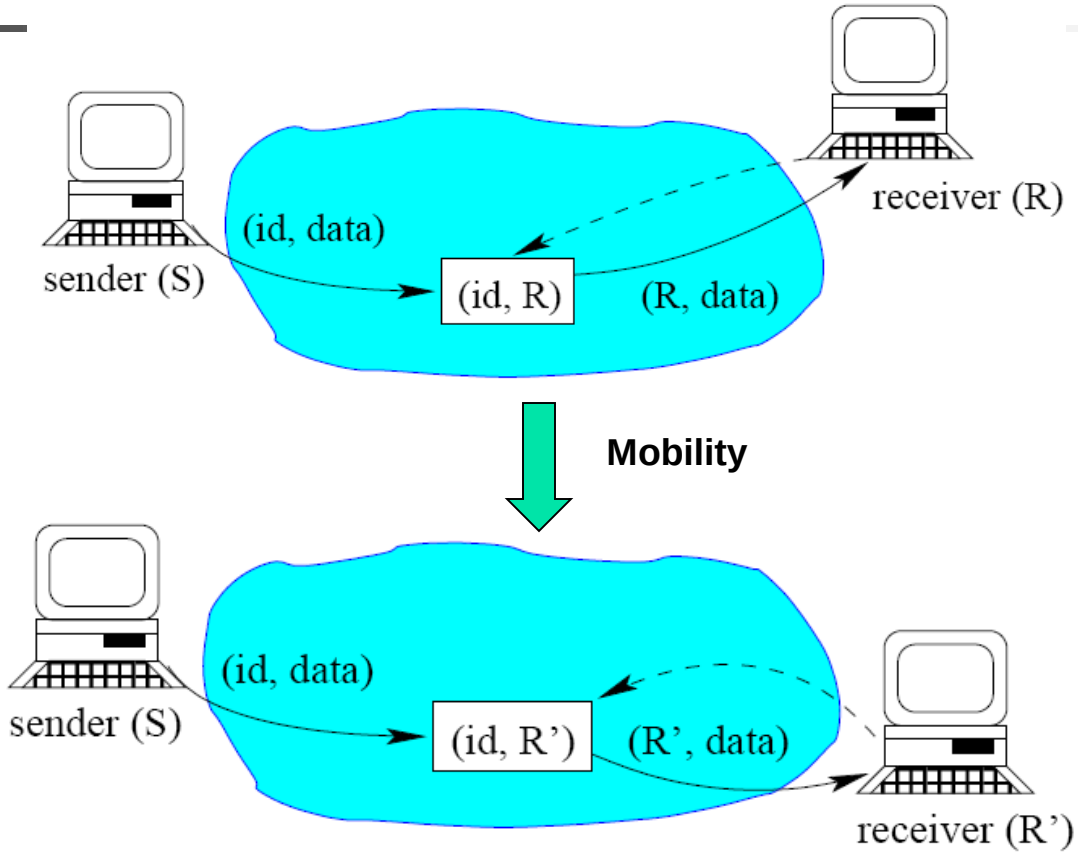
b. 发送主机向**DHT**网络发送分组**(id, data)**，该分组被**DHT**网络中的服务器转发给相应的接收主机

(1) i3原理

- 应用程序接口API
  - sendPacket(p)：发送分组
  - insertTrigger(t)：插入trigger
  - removeTrigger(t)：删除trigger
- 基于OpenDHT的实现
  - 扩展put/get接口以实现服务器转发功能

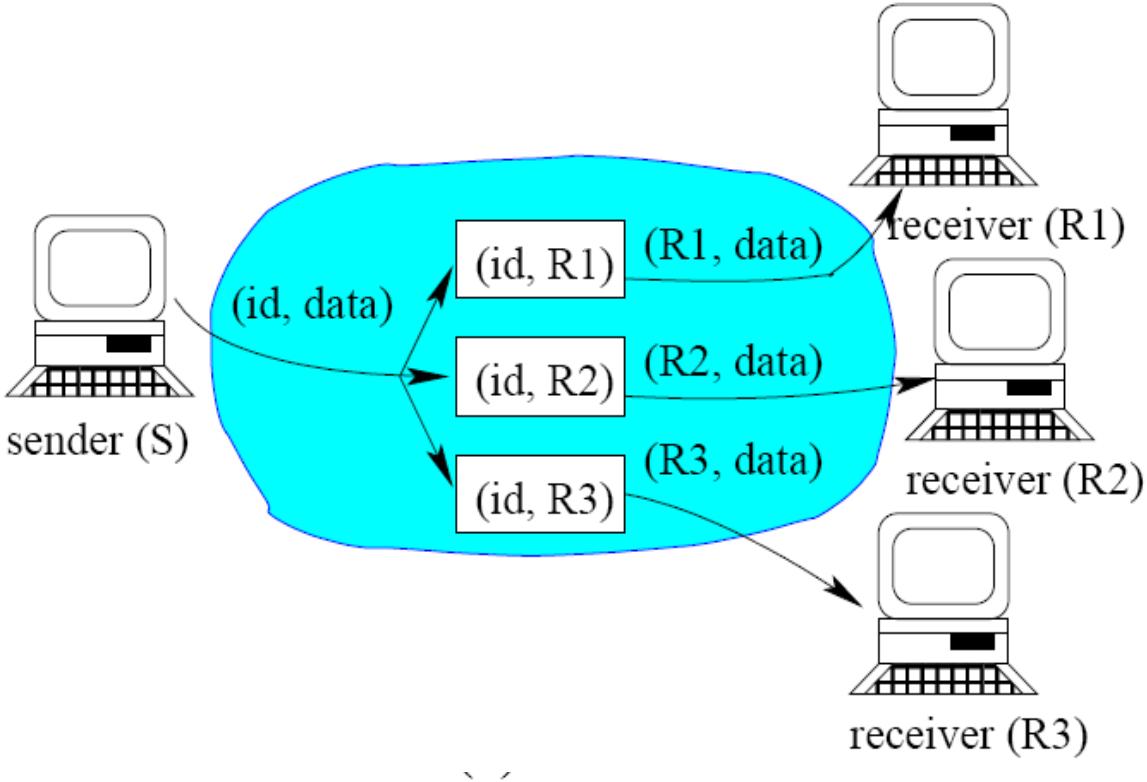
Sean Rhea, Brighten Godfrey, et al., OpenDHT: A Public DHT Service and Its Uses, *Proceedings of ACM SIGCOMM 2005*, August 2005

(2) i3应用：移动



移动对于发送节点完全透明，接收节点只需更新相应的trigger

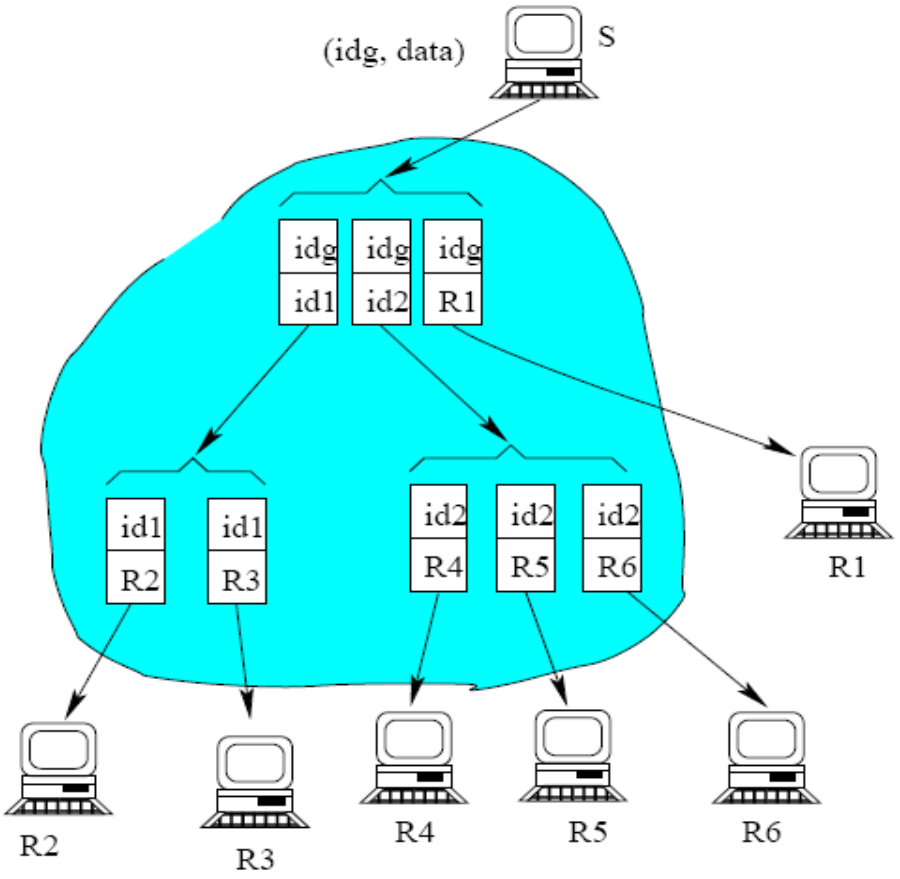
(2) i3应用：组播



组播接收节点向DHT网络插入具有相同标识的trigger

(2) i3应用：大规模组播

- 使用层次触发器构造组播树，以减轻服务器的负担
- 具有相同标识的trigger的数量代表在服务器上分组的复制因子
- 需要分布式算法来构造和维护trigger的层次



LAKSHMINARAYANAN, K., RAO, et al.. Flexible and robust large scale multicast using i3. Tech. Rep. CS-02-1187, University of California-Berkeley, 2002.