并行与分布式计算 - June 26'th, 2018

Data Processing Concepts

https://github.com/rh01

P2P Lookup

P2P(Peer to peer) 系统

- P2P 系统中的每个结点在连接上是互联的，在功能上是平等的，在行为上是自由的
- P2P 系统通常构建有高效的覆盖网 (overlay)，允许结点动态地加入和离开
- P2P 系统的每个结点既是服务使用者，也是服务提供者
- P2P 系统的每个结点通过冗余机制或周期性检测等提供容错性
  - 自治性
  - 分散性：资源的所有权和控制权被分散到网络的每一个节点中
  - 高容错性：系统能适应网络拓扑结构的变化，能避免单点故障
  - 高可伸缩性
  - 高可用性，负载平衡…

P2P Lookup

- Functional requirements:
  - Locate and communicate with any individual resource
  - Add new resources or remove them at will
  - Add hosts or remove them at will
  - Provide simple APIs to store and find data
    - Typical DHT interface: put(key, value), get(key) → value
- Non-functional requirements:
  - Global scalability
  - Load balancing
  - Accommodating to highly dynamic host availability
  - Optimization for local interactions between neighboring peers
  - Security of data in an environment with heterogeneous trust
  - Anonymity, deniability and resistance to censorship

Several P2P Systems/Overlays

- **Napster**: a music file sharing system
- **BitTorrent**: a file sharing system
- **Gnutella**: a file sharing system
- Pastry, Tapestry: DHT, Prefix routing

- **Chord**: DHT, a ring (an artificial one-dimensional space)
- **CAN**: DHT, d-dimensional Cartesian coordinate space
- Kademlia (eMule, BitTorrent without tracker)

Napster 音乐文件共享系统

- 使用方式
  - 用户连接到服务器集群中的一台服务器，把他愿意与其它用户共享的文件信息发送给服务器，服务器根据这些信息和用户的位置，建立索引并加入到原有的索引表中
  - 用户发查询请求 Q 给与其相连的服务器，该服务器收到请求后，与其他服务器协作处理查询消息 Q，回复用户一个表单，这个表单包含了所查到的所有匹配的文件索引
  - 用户收到回复 R 后，选择他想要的文件，根据文件索引中对应的位置与其他用户直接建立连接并下载文件
- 服务器的作用
  - 维护所有用户的共享文件索引
  - 监控用户的状态

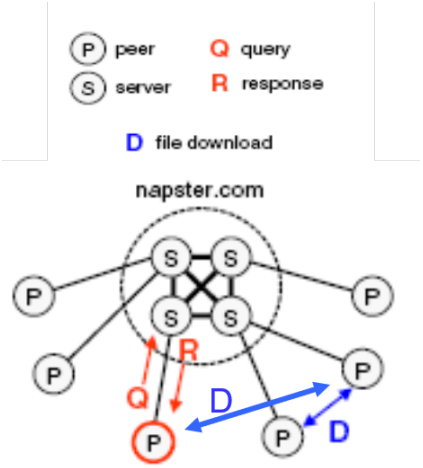

图 1：Napster

- Napster 的设计依据
  - 音乐文件不会被更新
  - 不需要保证单个文件的可用性
- Napster 的缺陷
  - 服务器是系统瓶颈，可能单点失效，可伸缩性差
  - 过于松散的组织管理
  - 版权问题
- 从 Napster 获得的 P2P 系统的设计要点
  - 要考虑节点异构性：节点的网络连接能力差异很大

  - 系统应该有方法鼓励用户报告正确的信息，限制错误的报告
  - 系统必须有方法鼓励上传，能限制或禁止自私节点使用网络
  - 需要更强的匿名技术

BitTorrent

- BitTorrent 由 BT 网站、.torrent 文件服务器、Tracker、BT 用户组成
  - BitTorrent 中，一个文件分割成固定大小的块 (chunk)，对应一个.torrent 文件
    * 文件的名字和长度，下载次数、种子数、上载文件的人
    * 跟踪器 (tracker) 的位置 (用一个 URL 指定)
    * 与每个块相关的校验和
  - BT 网站，供用户搜索.torrent 文件列表
  - Tracker 保存该文件的所有下载器 (downloader) 和 seed 的注册信息，同时，管理多个文件的并发下载
- 流程：用户请求 → BT 网站 → .torrent 文件服务器 → .torrent 文件 → Tracker → 返回下载该文件的场地信息
  - 用户与那些 peer 直接相连，进行文件下载 (barter for chunks of the file)
  - 下载同一个文件的用户围绕 Tracker 形成一个独立的子网
- BitTorrent 将文件分片 (piece)，分片又被划分成子分片，子分片进行流水作业
  - Piece Selection: Rarest First
  - Piece Selection: Endgame Mode
- 一报还一报 (tit-for-tat) 的激励机制
  - BitTorrent 要求 peer 合作，合作，意味着 peer 要上传，不合作的话，就阻塞 peer
  - 提供基于 Pareto-efficient 的阻塞算法：每隔一定周期 (10 秒) 计算下载率，下载率高的 peer 就不阻塞。
  - Optimistic unchoking：每隔一定周期 (30 秒) 随机选择一个用户实施不阻塞

BitTorrent 的优化：Merkle tree

- Merkle tree/Hash tree：叶子结点是数据块 (e.g., 文件或者文件的集合) 的 hash 值，非叶节点是先将两个孩子节点的 hash 值合并成一个字符串，然后运算这个字符串的 hash 值。
  - 在 P2P 网络中，Merkle Tree 用来校验数据完整性，确保从其他节点接收的数据块没有损坏且没有被替换，甚至检查其他节点不会欺骗或者发布虚假的块。
  - 利用 Merkle tree 比较两台机器上文件的不同
  - Dynamo：使用 Merkle tree 快速判断副本之间是否一致，可以快速发现不一致的副本并减少副本同步需要传输的数据量

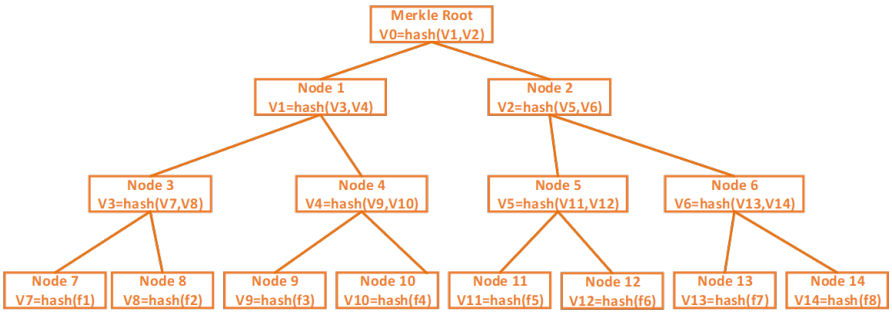

图 2：Merkle tree

Gnutella 协议

- Gnutella 协议包含下列消息：
  - Ping，Pong
  - Query，QueryResponse
    * 文件搜索的洪泛策略：每个节点给它的每个邻居转发请求，这些邻居节点再依次把请求传递给它们的邻居，如此这般，一直到找到匹配的文件为止
    * 每条消息有一个 TTL(time-to-live)，以限制洪泛
    * 每个结点缓存最近路由的消息，以支持汇聚 (沿洪泛的反向路径回传消息) 和阻止不必要的重复广播
  - Get，Put
- 节点接入：通过众所周知的 Gnutella 结点连入 Gnutella 网络
- (改进 1) 引入了 Ultrapeer：超级节点间连接紧密 (每个都有超过 32 个连接)，其他一些对等节点承担叶子节点的角色，这大大减少了进行彻底搜索所需求的最大跳数
- (改进 2) 改进了 QRP(Query Routing Protocol)：节点生成 Query Routing Table，它包含代表节点上的每个文件的哈希值，接着，QRT 发送给所有与它相连的超级节点，超级节点基于所有相连的叶子节点的所有项加上自身包含的文件的项，形成它们自己的 QRT，并与其他相连的超级节点交换 QRT。这样，超级节点能对一个给定的查询决定哪个路径能提供一个有效的路由，从而大大减少了不必要的流量
- Gnutella 大致符合 =2.3 的幂律 (Power-Law) 分布网络，面对随机结点失效的容错性非常高
- Gnutella 大多数结点的连接数都高于某个常数值，故对于恶意攻击的容错性较高

Chord Characteristics

- Simplicity, provable correctness, and provable performance
- Chord origins from the consistent hashing algorithm and addresses following difficult problems:
  - Load balance: Chord adopts a distributed hash function, spreading keys evenly over nodes
  - Decentralization: Chord is fully distributed, no node more important than other, improves robustness
  - Scalability: Chord has a logarithmic growth of lookup costs with number of nodes in network, even very large systems are feasible
  - Availability: Chord automatically adjusts its internal tables to ensure that the node responsible for a key can always be found
  - Flexible naming: There are no constraints on the structure of the keys. Key-space is flat, with the

flexibility in how to map names to Chord keys

Why Consistent Hashing

- Hash 函数是把任意长度的输入变换成固定长度的输出
  - 压缩映射
- 已有 hash 算法的问题不满足单调性
- 令 hash 函数为 x → (ax + b) mod (P)，P 表示全部缓冲的大小
- 一致性哈希应该满足的 4 个条件：
  - 均衡性 (Balance)，指哈希的结果能够尽可能分布到所有的缓存中
  - 单调性 (Monotonicity)，当缓冲区大小变化时，应尽量保护已分配的内容不会被重新映射到新缓冲区
  - 分散性 (Spread)：避免由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致
  - 负载 (Load)：对于一个特定的缓冲区而言，避免被不同的用户映射为不同的内容

整个哈希值空间组织成一个虚拟的圆环

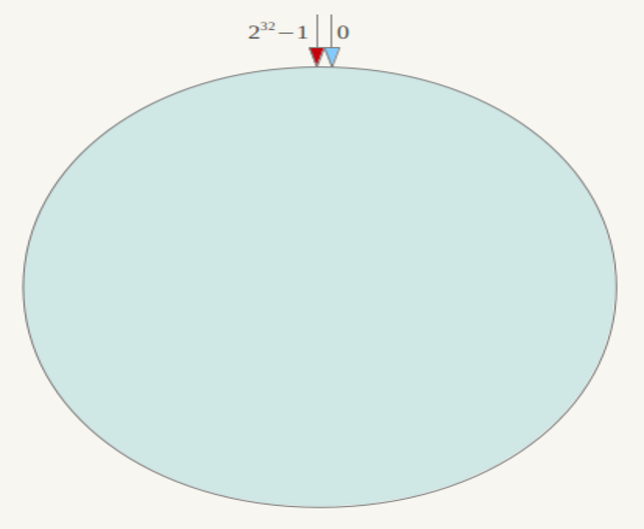

图 3：整个哈希值空间组织成一个虚拟的圆环

4 个数据对象经过 hash 计算后在环空间上的位置

Ex. 假设 Node C 不幸宕机，可以看到此时对象 A、B、D 不会受到影响，只有 C 对象被重定位到 Node D。一般的，在一致性哈希算法中，如果一台服务器不可用，则受影响的数据仅仅是此服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它不会受到影响。

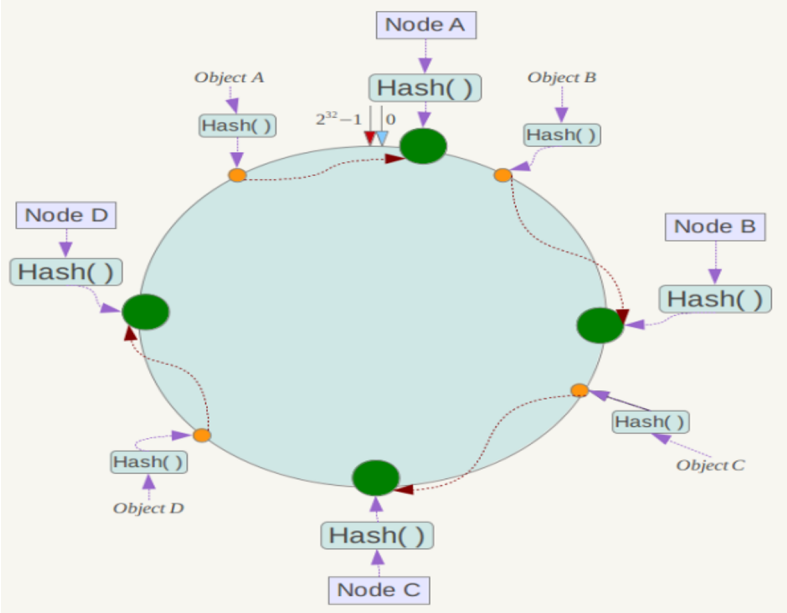

图 4：4 个数据对象经过 hash 计算后在环空间上的位置

在系统中增加一台服务器 Node X

Ex. 此时对象 Object A、B、D 不受影响，只有对象 C 需要重定位到新的 Node X 。一般的，在一致性哈希算法中，如果增加一台服务器，则受影响的数据仅仅是新服务器到其环空间中前一台服务器（即沿着逆时针方向行走遇到的第一台服务器）之间数据，其它数据也不会受到影响。

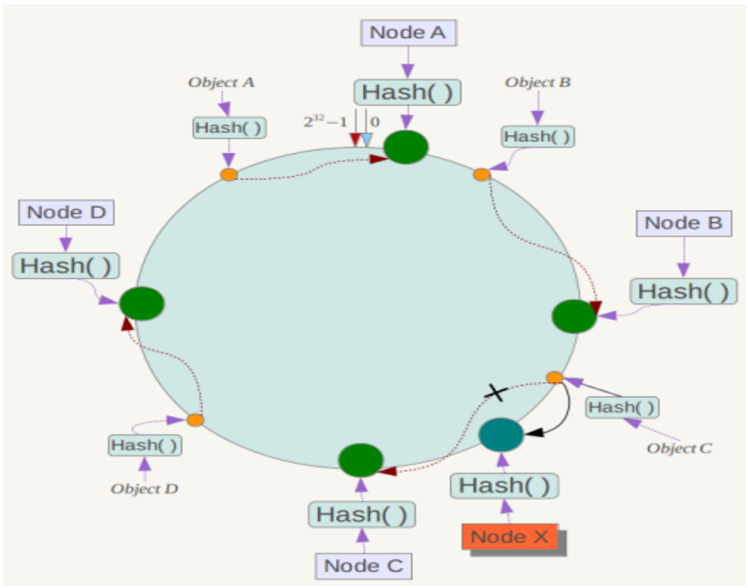

图 5：在系统中增加一台服务器 Node X

虚拟节点机制：解决数据倾斜

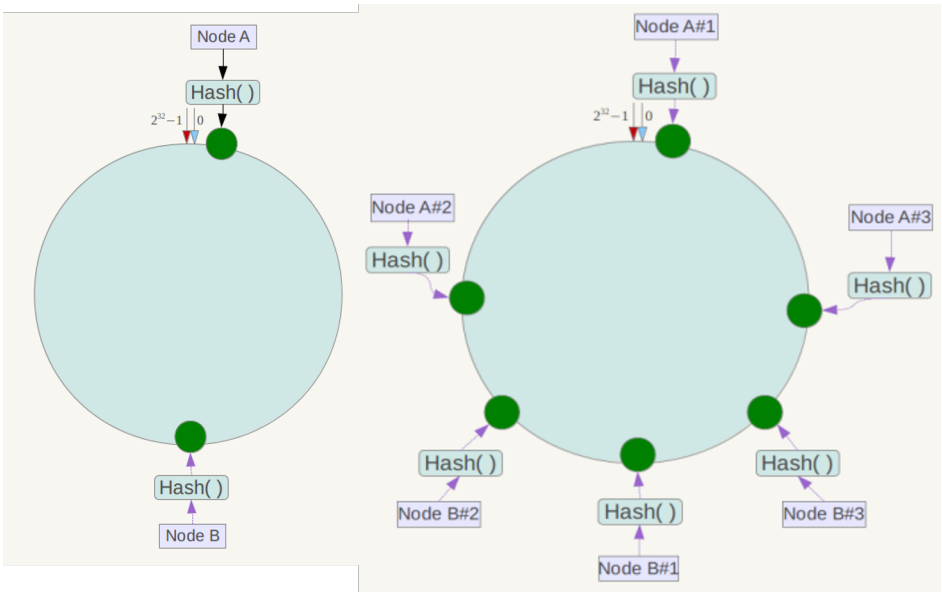

图 6：虚拟节点机制

Chord Ring

- A node’ s m-bit identifier is chosen by hashing the node’ s IP address, using SHA-1 as a base hash function
- Nodes are ordered on an identifier circle modulo 2^m (Chord ring)
- Key k is assigned to the first node whose identifier is equal to or follows (the identifier of) k in the identifier ring. This node is called the successor node of key k, denoted by successor(k)

Successor Nodes

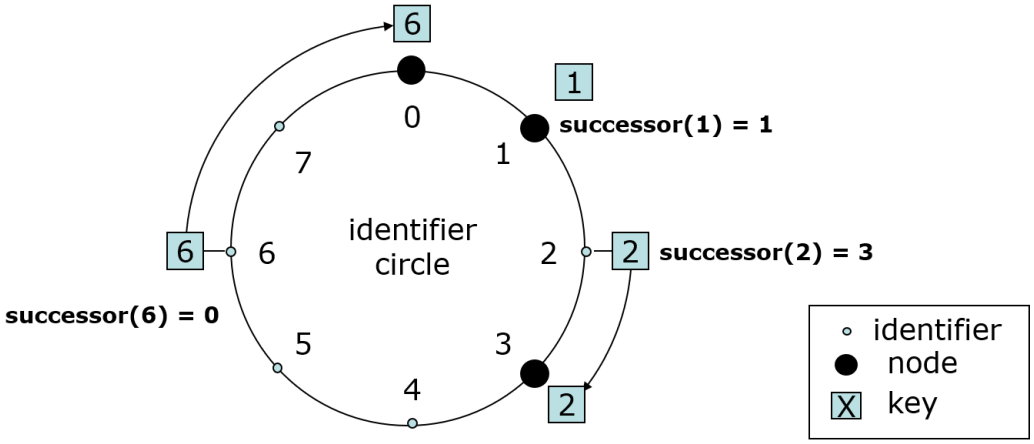

图 7：Successor Nodes

Node Joins and Departures

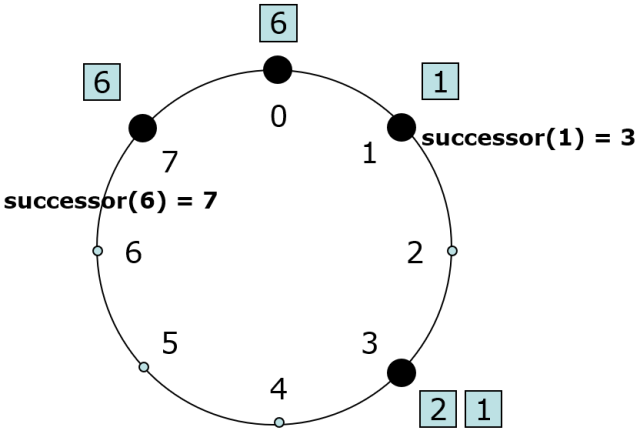

图 8：Node Joins and Departures

Simple key location

- A very small amount of routing information suffices to implement lookup in a distributed environment
- Each node need only be aware of its successor node on the circle
- Queries for a given identifier can be passed around the circle via these successor pointers
- Resolution scheme correct, BUT inefficient: it may require traversing all N nodes!

Algorithm 1 Simple key location

```
1: // ask node n to find the
2: // successor of id
3: n.find_successor(id)
4: {
5:     if (id ∈ (n; successor])
6:         return successor;
7:     else
8:         // forward the query around the circle
9:         return successor:find_ successor(id);
10: }
```

Scalable key location

- Each node n maintains a finger table with up to m entries:
- the i-th entry (n.finger[i]) in the table at node n contains the identity of the first node s that succeeds n by at least 2^{i-1} on the ring, i.e., s = successor(n+2^{i-1}), where 1<=i<=m
- Finger[i]: first node on circle that succeeds (n+2^{i-1}) mod 2m, 1<=i<=m
- Successor: the next node on the identifier circle
- Predecessor: the previous node on the identifier circle

Finger Table

Algorithm 2 Simple key location

```
1: // ask node n to find the
2: // successor of id
3: n.find_successor(id)
4: {
5:     if (id ∈ (n; successor])
6:         return successor;
7:     else
8:         // forward the query around the circle
9:         return successor:find_ successor(id);
10: }
11: //ask node n to find the successor of id
12: n.find_successor(id)
13: {
14:     if (id ∈ (n, successor])
15:         return successor;
16:     else
17:         n' = closest_preceding_node(id);
18:     return n' .find_successor(id);
19: }
20: // search the local table for the
21: // highest predecessor of id
22: n.closest_ preceding_ node(id)
23: {
24:     for i = m downto 1
25:         if (finger[i] ∈ (n, id))
26:             return finger[i];
27:     return n;
28: }
```

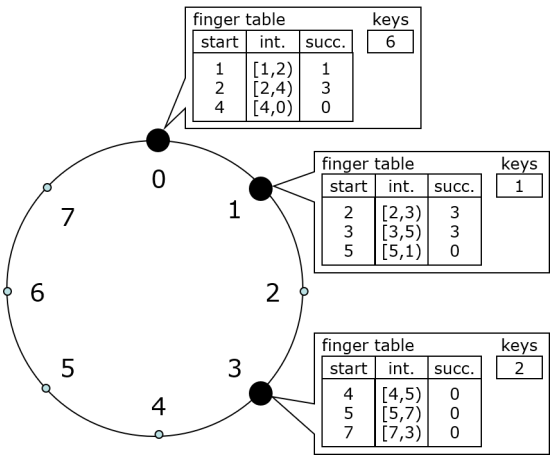

图 9：Finger Table

Node Joins -with Finger Tables

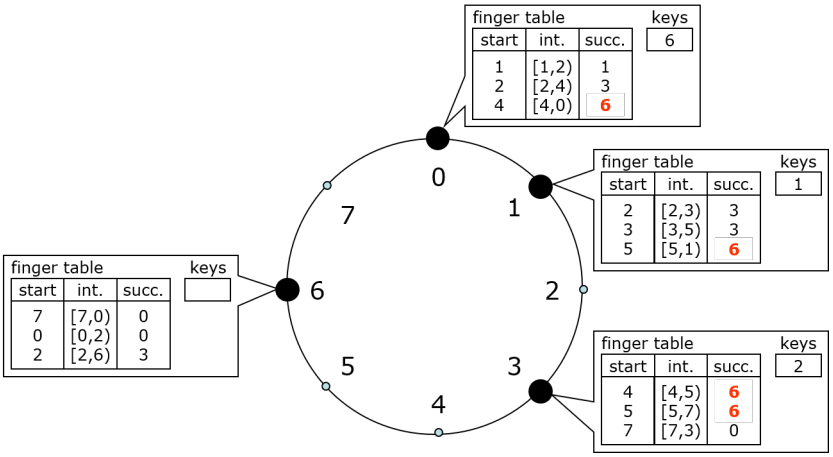

图 10：Node Joins

Node Departures -with Finger Tables

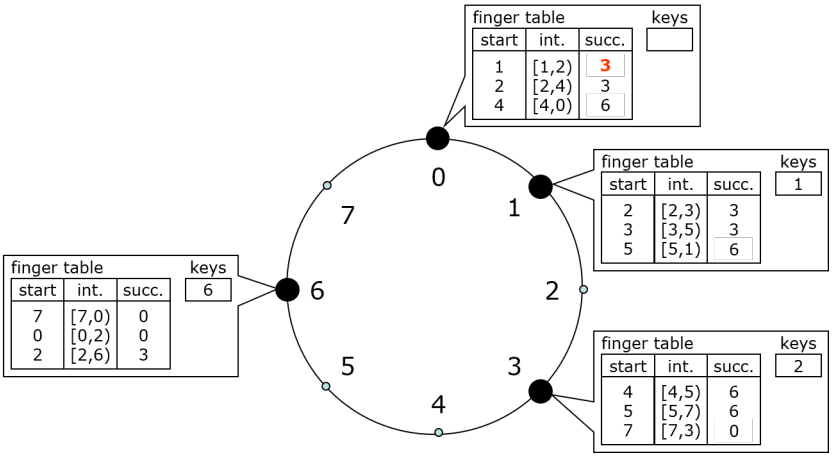

图 11：Node Departures

Stabilization after Join

- n joins
  - predecessor = nil
  - n acquires n_s as successor via some n'
  - n notifies n_s being the new predecessor
  - n_s acquires n as its predecessor
- n_p runs stabilize
  - n_p asks ns for its predecessor (now n)
  - n_p acquires n as its successor
  - n_p notifies n
  - n will acquire n_p as its predecessor
- all predecessor and successor pointers are now correct

- fingers still need to be fixed, but old fingers will still work

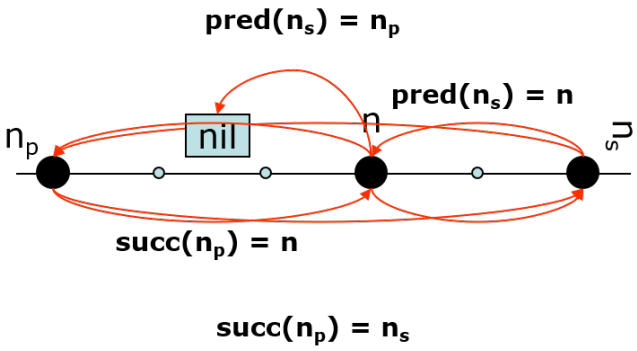

图 12：Stabilization after Join

Failure Recovery

- Key step in failure recovery is maintaining correct successor pointers
- To help achieve this, each node maintains a successor-list of its r nearest successors on the ring
- If node n notices that its successor has failed, it replaces it with the first live entry in the list
- stabilize will correct finger table entries and successor-list entries pointing to failed node
- Performance is sensitive to the frequency of node joins and leaves versus the frequency at which the stabilization protocol is invoked

Theoretical Analysis

- Given N nodes and K keys, every node is responsible for about K/N keys
- When a node joins or leaves an N-node network, only O(K/N) keys change hands (and only to and from joining or leaving node)
- Lookups need O(log N) messages
- To reestablish routing invariants and finger tables after node joining or leaving, only O(log2N) messages are required

Compared with the other P2P protocols

- Napster, it uses a central index, resulting in a single point of failure
- Gnutella, it floods each query over the whole system, so its communication and processing costs are high in large systems
- FreeNet, it cannot guarantee retrieval of existing documents or provide low bounds on retrieval costs
- Pastry, it needs a more elaborated join protocol, which initializes the routing table of the new node by using the information from nodes along the path traversed by the join message.
- Tapestry, it is complicated
- CAN, in CAN, each node maintains O(d) state, and the lookup cost is O(dN1/d), its lookup cost increases faster than logN

Kademlia

- 网络中的每个节点根据其 IP 地址及端口分配一个唯一的、随机的 nodeID (160 bit 的整数)
- 节点之间的距离：如果节点的 nodeID 是 x，y，那么它们之间的距离 dist(x,y) = x XOR y
  - dist(x,x) =0; dist(x,y)>0, if x!=y
  - 对称性：对于任何 x、y，有：dist(x,y) = dist(y,x)
  - 三角属性：dist(x,y) + dist(y,z) >= dist(x,z)
  - 单向性：对于任意点 x 和距离 d>0，有且仅有唯一一点 y，满足 dist(x,y) = d
  - 对相同数据对象的定位最终将收敛于相同的路径，所以，用“沿路径缓存”能提高查找效率、缓解热点
- 每个对象分配一个 key，也是一个 160bit 的整数
- <key, value> 对形式的数据被存放在 nodeID 最接近 Key 值的节点上

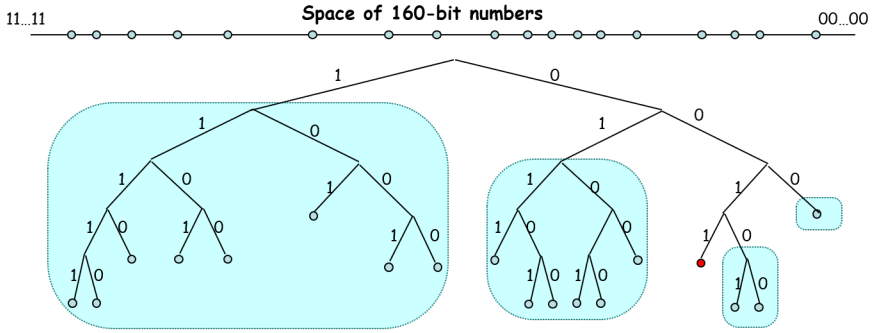

图 13：节点 0011 的子树划分

- 对于任意一个节点，都可以把这颗二叉树分解为一系列连续的、不包含自己的子树
- 每个节点至少知道子树中的一个节点
- 方框部分就是各子树，由上到下各层的前缀分别为 1，01，000，0010
- 每个节点上的路由表共 160 项，每一项是一个列表，称为 K 桶，存放和自己距离在 $[2^i, 2^{i+1}]$ 内的节点信息，总共存放 k 个，并按照最近访问时间排序
- 全部 K 桶的信息加起来就覆盖了整个 160bit 的 nodeID 空间，而且没有重叠

路由表

	距离	邻居 (最多 K 个)
0	[20,21)	(IP 地址, UDP 端口, nodeID)⋯ (IP 地址, UDP 端口, nodeID)
1	[21,22)	(IP 地址, UDP 端口, nodeID)⋯ (IP 地址, UDP 端口, nodeID)
2	[22,23)	(IP 地址, UDP 端口, nodeID)⋯ (IP 地址, UDP 端口, nodeID)
⋯		
i	[2i,2i+1)	(IP 地址, UDP 端口, nodeID)⋯ (IP 地址, UDP 端口, nodeID)
⋯		
159	[2159,2160)	(IP 地址, UDP 端口, nodeID)⋯ (IP 地址, UDP 端口, nodeID)

表 1: 路由表

- 每个节点的路由表都表示为一颗二叉树，叶子节点为 K 桶，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，而且没有重叠
- Kad 路由表确保每个节点知道其各子树的至少一个节点，只要这些子树非空
- 路由查询过程是收敛
- 对于一个有 N 个节点的 Kad 网络，最多只需要经过 logN 步查询，就可以准确定位到目标节点

路由表的维护：稍带更新方式

- 每次收到其他节点的信息，用那些节点的 nodeID 来更新路由表相应行的列表 (即，相应的 K 桶)。在相应行中，
  - 如果在 K 桶中已有此 nodeID，那么，把这项移到列表的尾部；
  - 如果没有，而且，列表长度不到 K ，那么，把这项加到列表的尾部；
  - 如果没有，而且，列表长度为 K ，那么，先联系表头节点，
  - 如果能联系到，则将表头节点移到表尾，其他项不变；
  - 如果联系不到，则删除表头节点，把这项加到列表的尾部 s
- 好处
  - 以较小的开销获得很高的自适应性、容错性
  - 防止 DoS(服务拒绝) 攻击

Kademlia 协议：4 种 RPC 操作

- Ping：用于检测一个节点是否在线
- Store(key, value)：存储 (key, value)，key 为对象散列值，value 为数据对象
- Find_node(ID)：返回离目的 ID 最近的 k 个节点 <IP 地址，UDP 端口，结点 ID>
- Find_value(key)：寻找与 key 对应的 value, 如果该消息的接收者已收到过相应 key 的 Store 消息，那么返回对应 key 的 value，同时，将 (key,value) 对存储到它所知的离 key 最近、但没有返回 value 的节点中

节点查询：递归过程查找 DesID

- 查询发起者 x 计算自己到 DesID 的距离 d=dist(x,DesID)
- 从 x 的第 [log d] 个 K 桶 (第 [log d] 行) 中取出　个节点的信息，然后执行 FIND_NODE 操作。如果这个 K 桶中的信息少于　个，则从附近多个桶中选择距离最接近 d 的总共　个节点
- 对接收到查询操作的每个节点，如果发现自己就是 DesID，则回答自己是最接近 DesID 的；否则测量自己和 DesID 的距离，并从自己对应的 K 桶中选择　个节点的信息给 x
- x 如果没有收到某个节点的回复，就在 k 桶中删除那个节点
- x 对新接收到的每个节点都再次执行 FIND_NODE 操作，此过程不断重复执行，直到无法获得比查询发起者当前已知的 k 个节点更接近 DesID 的活动节点为止

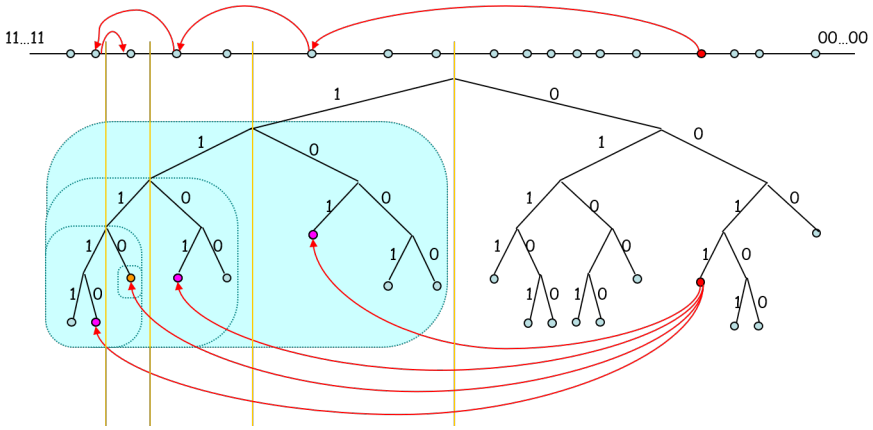

图 14：节点 0011 通过连续查询来找到节点 1110

- 只有第一步查询的节点 101，是节点 0011 已经知道的，后面各步查询的节点，都是由上一步查询返回的更接近目标的节点，这是一个递归操作的过程

存储 (key, value) 对：nodeID 邻近复制策略

- 先用 Find_node(key) 找到离 key 最近的 k 个节点，然后发送 Store(key, value)，在这些节点上存储 value
- 上述 K 个节点每过 1 个小时，重新发布 (key, value) 对，以保证数据可用
- 最初的 (key, value) 对，发布者每过 24 小时重新发布它，否则所有 (key, value) 对会在 24 小时后过期
- 当任何一个节点 w 发现，存在另一个节点 u 离保存在 w 上的 (key, value) 对更近时，w 会复制这个 (key, value) 对到 u，但并不删除自己保存的 (key, value) 对

自适应性

- 如果某行所有节点在 1 个小时中未被查询,那么,就从中任选一个节点 nodeID 对它做节点查询 Find_node(key)，这里 key=nodeID，以刷新该行内的节点
- 如果新节点 u 要加入 Kad 网络，它先联系到一个网络现存节点 w，将 w 加入自己的 k 桶，然后，通过 w 做一次以 u 为目的地的节点查询，从而初始化自己的 k 桶，然后，将自己的信息告诉其他节点，以更新他们的状态
- 节点离开 Kad 网络不需要发布任何信息

DHT Approach

- DHT: Distributed Hash Table
  - Object' s key/GUID (Globally Unique IDentifiers) is calculated by the (part of) states of object using a Hash function such as SHA-1
- Give each node a unique ID, and arrange nodes in an ID-space
  - Examples: 1D line/ring, 2D square, Tree based on bits, hypercube
- Have a rule for assigning keys to nodes based on node/key ID
  - e.g. key X goes on node with nearest ID to X
- Build routing tables to allow ID-space navigation
  - Each node knows about ID-space neighbors, i.e., knows neighbors' IDs and IP addresses
  - Perhaps each node knows a few farther-away nodes, in order to move long distances quickly
- For any set of N nodes and K keys, with high probability for load balance:
  1. Each node is responsible for at most (1+O(logN))K/N keys
  2. When an (N + 1)st node joins or leaves the network, responsibility for O(K/N) keys changes hands (and only to or from the joining or leaving node)