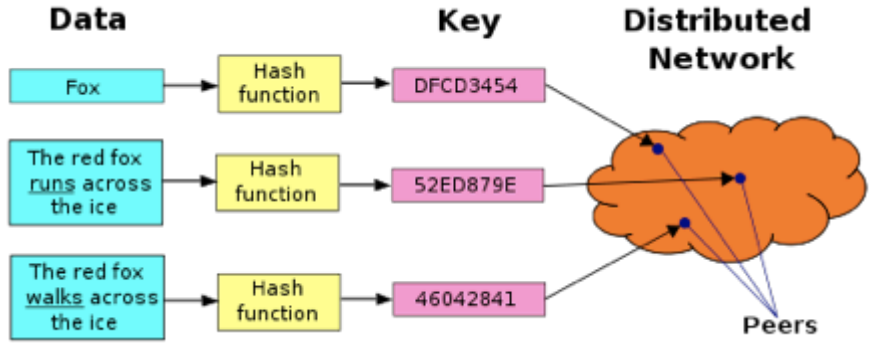
分布式哈希表（**Distributed Hash Table**，**DHT**）

分布式哈希表，即提供 Key/Value 的存储服务。最基础的两个接口就是 PUT(Key,Value)和 GET(Key)，即将数据保存到 DHT 和从 DHT 中读取某个特定的 Key 对应的 Value 值。

分布式哈希表必须具备以下特性：动态可拓展，数据具被分散性，系统自管理。简单地说就是能够动态扩展节点，每个节点上的数据尽量分散，系统能够自动容错等。



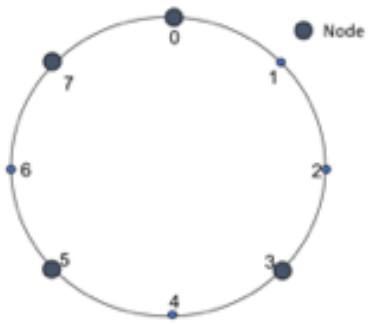
本文先简单介绍 DHT 的基本知识，然后介绍 DHT 中用到的算法，包括环的原子管理算法，路由算法，组通信算法，副本管理算法，然后简单介绍了 DHT 的一些应用，最后进行简单的总结。

1. 基础知识

在查找或插入新数据时候，首先，Hash 算法会将 key 映射到一个键空间（Keyspace），例如，键空间[0, 2^M)，表示 0 ~ 2^M-1。

然后将键空间进行划分成多个分区（Keyspace Partition）。

如下图所示，表示[0,7]的键空间，并划分成 4 个分区，即[0,0],[1,3],[4,5],[6,7]。



在 DHT 中，键空间划分是根据节点进行的，即如上所示，0，3，5，7 表示一个数据节点，而每

个节点将负责某一区间的键值，即节点 0 负责[0,0]，节点 3 负责[1,3] ， 节点 5 负责[4,5]，节点 7 负责[6,7]。

下面将主要介绍 DHT 中使用的算法。

2. 环的原子管理算法

本章节将介绍分布式哈希中的环管理的原子算法。

2.1 环管理中存在的问题

DHT 一般会采用到环型阶段管理的方法，然而环中节点是可能随时变化的，即随时可能有新的节点加入，随时有节点离开环。在 DHT 中节点变更后，每个节点定期进行检查，如果检查到差异，则进行 Stabilization，即下图中的算法。

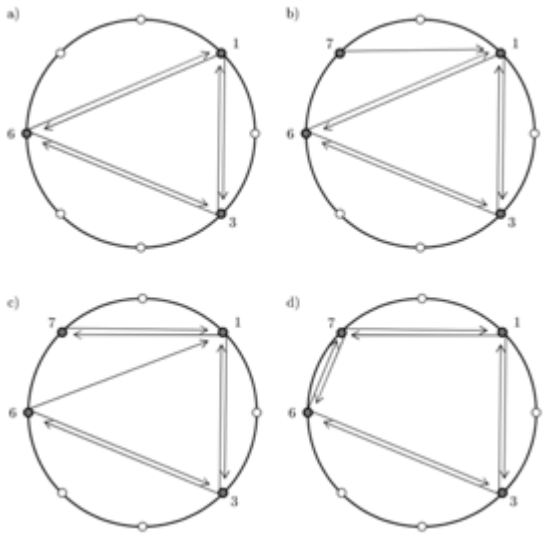
该算法首先拿到 succ（后续节点）的前序节点（pred）p；然后检查 p 是否在 n 和 succ 之间，如果是，则将 succ 减到 p；然后通知 succ 节点，尝试修改 succ 节点的前序节点。

Algorithm 1 Chord's periodic stabilization protocol

```
1: procedure n.STABILIZE()
2:     p := succ.GetPredecessor()
3:     if p ∈ (n, succ) then
4:         succ := p
5:     end if
6:     succ.Notify(n)
7: end procedure

8: procedure n.GETPREDECESSOR()
9:     return pred
10: end procedure

11: procedure n.NOTIFY(p)
12:     if p ∈ (pred, n] then
13:         pred := p
14:     end if
15: end procedure
```

举个例子，如下图所示。环中先有 1，3，6 节点（a）；先加入一个新节点 7（b），并将 7 的 succ 节点指向 1；待 7 执行完 Stabilization，得到（c）；待 6 也执行完 Stabilization 后，得到最终的

结果（d），完成节点添加的过程。



以上是一个节点的增加，但是节点变化，数据增删改查等操作同时发生，可能会发生无法预知的错误，为此需要对环中的节点进行保护管理。

如下图所示，假如原来环中有 3 和 9 两个节点，现在往环中加入新的节点 7，即将 7 的 succ 指针指向了 9（a）；在 7 完成 Stabilization 后，得到（b）；此时，节点 5 加入，并将 succ 指向 9（c）；待 5 完成 Stabilization 后，得到 d）。

此时如果一个 key 为 6 的请求发送到节点 3 上，那么 3 最后会路由到节点 9 上；而如果发送到节点 5 上，而会路由到节点 7 上。这时存在不一致，出现问题。

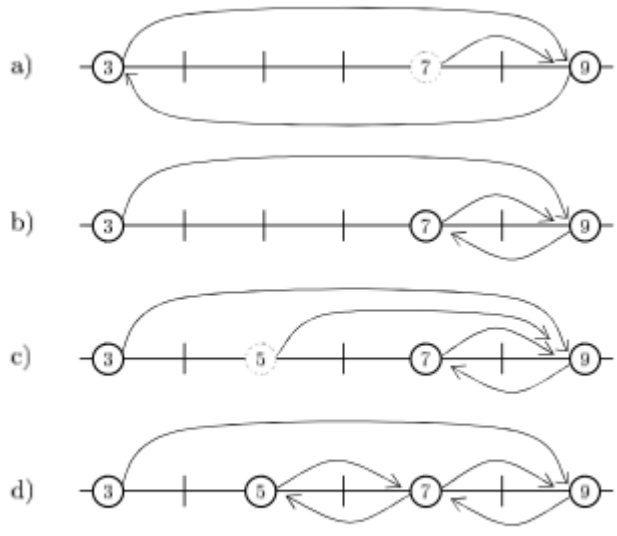


Figure 3.1: Example of inconsistent stabilization.

2.2 环管理并发控制算法

为了解决以上的问题，DHT 需要对环的节点进行保护管理。并在此基础上，修改对数据管理算法。

2.2.1 环管理锁算法

最简单的方法就是在有节点变化时候，将整个环锁住，即只最多允许一个节点在某一个时候加入或离开环。该算法实现相对简单，但是性能并不好。

另外一个方法就是三把锁方法，即在节点加入或者离开的时候，锁住要加入/离开的节点（node），该节点的后续节点（node's succ），该节点的前序节点（node's pred）。该方法相对锁整个环的方法有更好的性能，但是存在很大概率发生死锁/活锁的可能（即类似 *the dining philosophers* 问题）。

还有一个相对简单的方法，即只锁当前节点和该节点的后续节点。该节点相对来说，实现更加简单【注：在论文中划分了大章节证明该方法是安全的】，但是仍是存在死锁/活锁的问题，但是概率相对前面方法较低，即发生的前提是全部节点一起离开或者加入。

为了解决这个问题，DHT 可以采用非对称锁的方法进行解决。这里介绍一种基于锁队列（LockQueue）的方法。

每一个节点会有自己的 LockQueue 队列，当每一个请求需要锁某一个节点时，首先将请求加到该节点中的队列中；然后根据请求中的队列顺序进行锁的分配。

具体的基于锁队列的节点加入离开算法如下所示。

Algorithm 2 Asymmetric locking with forwarding

```
1: procedure n.JOIN(succ)                    ▷ Join the ring with succ as successor
2:     Leaving :=false                                          ▷ Initialize variable
3:     LockQueue.ENQUEUE(n)                  ▷ Enqueue request to local lock
4:     slock :=GETSUCCLOCK()
5:     pred := succ.pred
6:     pred.succ := n
7:     succ.pred := n
8:     LockQueue := succ.LockQueue                ▷ Copy successor's queue
9:     LockQueue.FILTER((pred, n])                ▷ Keep requests in the range
10:    succ.LockQueue.FILTER((n, pred])           ▷ Keep requests in the range
11:    LockQueue.DEQUEUE()                                  ▷ Remove local request
12:    RELEASELOCK(slock)
13: end procedure

14: procedure n.LEAVE()                                          ▷ Leave the ring
15:     if n > succ then                                      ▷ Asymmetric Locking
16:         slock :=GETSUCCLOCK()
17:         Leaving :=  true                                  ▷ Enable forwarding
18:         LockQueue.ENQUEUE(n)              ▷ Enqueue request to local lock
19:     else
20:         Leaving :=  true                                  ▷ Enable forwarding
21:         LockQueue.ENQUEUE(n)              ▷ Enqueue request to local lock
22:         slock :=GETSUCCLOCK()
23:     end if
24:     pred.succ := succ
25:     succ.pred := pred
26:     LockQueue.DEQUEUE()                                  ▷ Remove local requst
27:     RELEASELOCK(slock)
28: end procedure
```

在确保节点管理是原子后，还需要对数据的增删改查进行修改，以满足数据管理上能够正常工作。

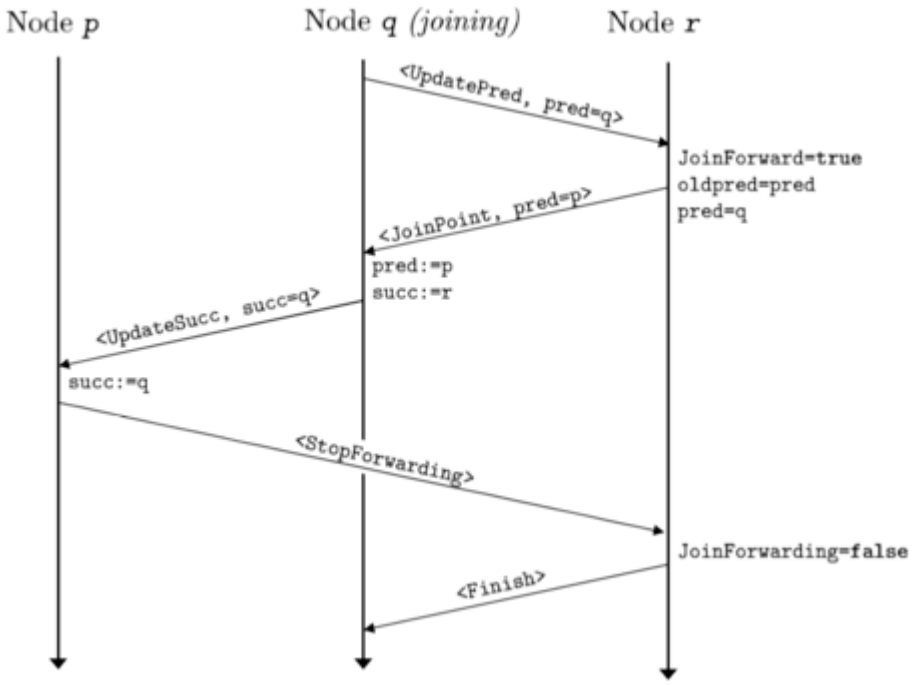
2.2.2 环原子管理的算法

首先介绍在节点加入的修改算法，在节点 n 加入时，促发 succ.UPDATEPRED，即将 succ 节点 JoinForward 置为 true，然后修改节点 n 的前置节点置为 pred 和 succ，然后更新 pre 节点的后续节点为 n，最后将 JoinForward 置为 false。

Algorithm 4 Pointer updates during joins

```
1:  event n.UpdateJoin() from n                    ▷ Assuming succ is correct
2:      sendto succ.UpdatePred()
3:  end event

4:  event n.UpdatePred() from m
5:      JoinForward :=true                         ▷ Forwarding Enabled
6:      sendto m.JoinPoint(pred)                   ▷ Join Point
7:      oldpred := pred
8:      pred := m
9:  end event

10: event n.JoinPoint(p) from m
11:     pred := p
12:     succ := m
13:     sendto pred.UpdateSucc()
14: end event

15: event n.UpdateSucc() from m
16:     sendto succ.StopForwarding()
17:     succ := m
18: end event

19: event n.StopForwarding() from m
20:     JoinForward :=false
21:     sendto pred.Finish()
22: end event
```

节点加入的状态迁移。



类似地，也需要在节点 n 离开环时的算法进行简单修改，如下图。基本过程如下：首先将节点 n 的 LeaveForward 置为 true；然后将 succ 节点的 pred 节点指向 n 的 pred 节点；然后修改 n 的 pred 节点的 succ 直接指向 n 的 succ 节点。

Algorithm 5 Pointer updates during leaves

```
1: event n.UpdateLeave() from n
2:     LeaveForward := true                    ▷ Forwarding Enabled
3:     sendto succ.LeavePoint(pred)
4: end event

5: event n.LeavePoint(p) from m
6:     pred := p
7:     sendto pred.UpdateSucc()
8: end event

9: event n.UpdateSucc() from m
10:    sendto succ.StopForwarding()
11:    succ := m
12: end event

13: event n.StopForwarding() from m
14:    LeaveForward :=false                    ▷ Forwarding Disabled
15: end event
```

在以上基础上，可以实现数据的管理算法。如下图所示，即根据节点的状态进行处理，

- 当 JoinForward 为 true 且消息来源为旧的 pred 节点时，将请求转发到 pred 节点；
- 当 LeaveForward 为 true 时，将请求转发到后续节点中；
- 如果 pred 节点存在，且请求归属于当前节点，则在当前节点查找；
- 其他情况将转发到 succ 节点。

Algorithm 6 Lookup algorithm

```
1: event n.LOOKUP(id, src) from m
2:     if JoinForward = true and m = oldpred then
3:         sendto pred.LOOKUP(id, src)                ▷ Redirect Message
4:     else if LeaveForward = true then
5:         sendto succ.LOOKUP(id, src)                ▷ Redirect Message
6:     else if pred ≠ nil and id ∈ (pred, n] then
7:         sendto src.LOOKUPDONE(n)
8:     else
9:         sendto succ.LOOKUP(id, src)
10:    end if
11: end event
```

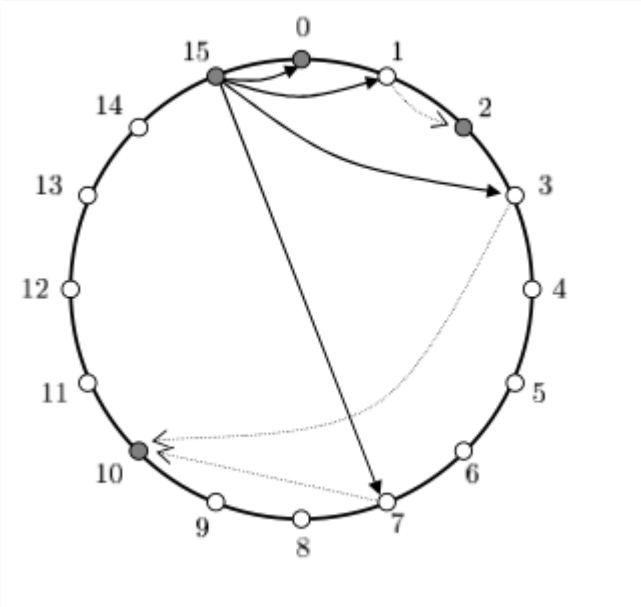
以上是环的原子管理算法，更多内容可以参考原论文。论文中介绍了一些其他优化点和容错方法。

3. 路由算法

DHT 中任意阶段均对外提供服务，即数据的增删改查请求可能落在任意节点上，为了将数据正确管理起来，DHT 需要提供一个正确有效的路由算法。

在环中，如果不做其他拓展，数据的管理性能是很差的，时间复杂度为 O(n)，n 为节点的个数。为了提高性能，DHT 可以对节点指针进行拓展。方法如下：在每一个节点中，除了 pred 和 succ 节点外，还增加了 successor-list，记录一些后续节点。successor-list 里头的 ID 由以下公式计算得到，其中 p 为当前节点，L=log(N)，N 为 Keyspace 空间中 ID 的总个数，如果计算出来的不是一个节点，那么将该 ID 的下一个节点作为 successor-list 的节点。
例如，对于下图中的环，其中 0，2，10，15 是节点，对于 15 节点来说，其 successor-list 由公式直接计算得到的值为{0,1,3,7}，但是 1, 3 和 7 不是节点，所以 15 的 successor-list 为{0, 2, 10}。

$$p \oplus 2^{L-1},$$



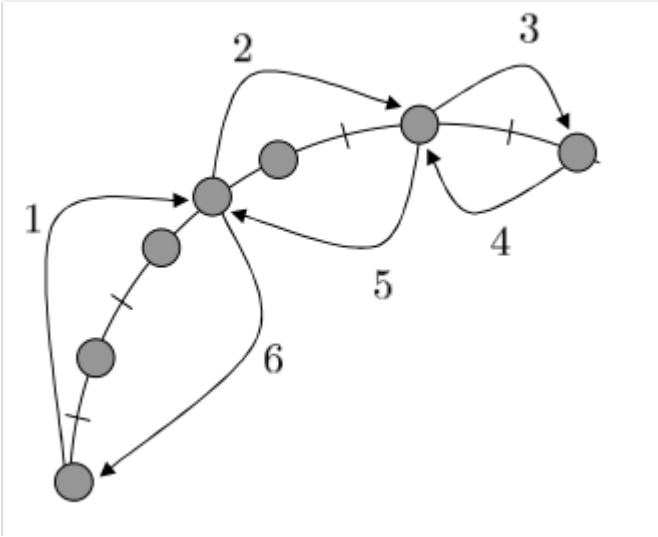
在以上的拓展后，路由算法可以采用图论中的一些经典算法，即递归算法，迭代算法，传递算法，贪心算法。下面将简单对这几个算法进行介绍。

3.1 深度递归算法

深度递归算法就是逐步查找每一个节点，是否是找到对应的数据，如下所示。如果请求不存在于该节点上，则将该请求转发到下一个节点，逐一逐一查找；待找到后，则将结果按照查找路径进行返回。

Algorithm 12 Recursive lookup algorithm

```
1: procedure n.LOOKUP(i, OP)
2:     if TERMINATE(i) then
3:         p :=NEXT_HOP(i)
4:         res := p.OP(i)                    ▷ OP could carry parameters
5:         return res
6:     else
7:         m :=NEXT_HOP(i)
8:         return m.LOOKUP(i, OP)
9:     end if
10: end procedure
```

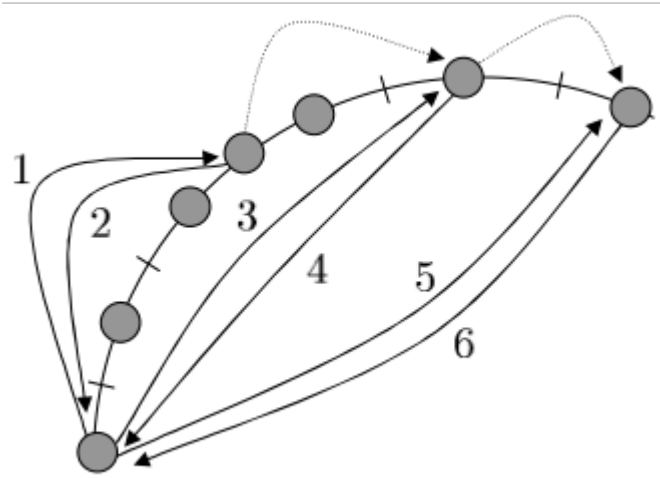


3.2 广度迭代算法

广度迭代算法是迭代地访问所有 successor-list 中的节点，如下图所示。

Algorithm 13 Iterative lookup algorithm

```
1: procedure n.LOOKUP(i, OP)
2:     m := n
3:     while not m.TERMINATE(i) do
4:         m := m.NEXT_HOP(i)
5:     end while
6:     p := m.NEXT_HOP(i)
7:     return p.OP(i)
8: end procedure
```

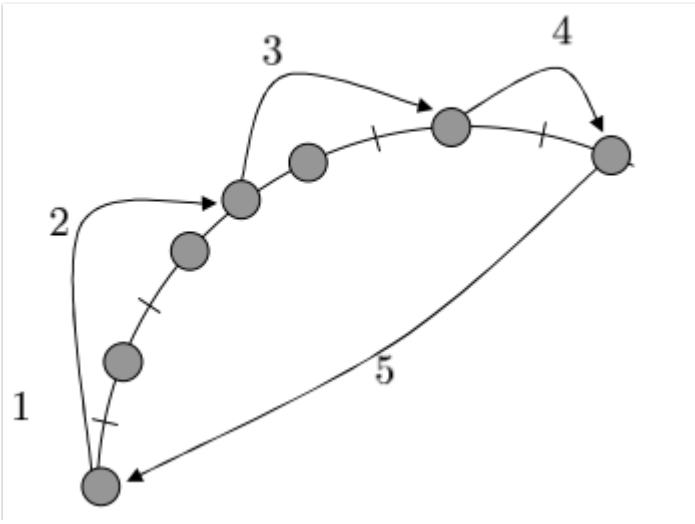


3.3 传递算法

深度递归算法和广度迭代算法两者的消息传输比较多，即 2n，n 为访问节点的个数。为了降低消息量，可以对递归算法进行修改，即在每一个请求中附带请求来源的节点信息，待查找到数据后，即将数据直接发送回请求来源的节点。

Algorithm 14 Transitive lookup algorithm

```
1:  procedure n.LOOKUP(i, OP)
2:      sendto n.LOOKUP_AUX(n, i, OP)
3:      receive LOOKUP_RES(r) from q
4:      return r
5:  end procedure

6:  event n.LOOKUP_AUX(q, i, OP) from m
7:      if TERMINATE(i) then
8:          p := NEXT_HOP(i)
9:          sendto p.LOOKUP_FIN(q, i, OP)
10:     else
11:         p :=NEXT_HOP(i)
12:         sendto p.LOOKUP_AUX(q, i, OP)
13:     end if
14: end event

15: event n.LOOKUP_FIN(q, i, OP) from m
16:     r := OP(i)
17:     sendto q.LOOKUP_RES(r)
18: end event
```



3.4 贪心算法

前面几种算法均需要遍历，没有利用 successor-list 中的路由信息。为了提升性能，这里介绍贪心算法，即在查找下一跳的时候利用路由信息。在查找下一跳时候，判断请求数据是否在 succ 节点，如果是则直接返回 succ，否则对 successor-list 进行查找。因为 keyspace 是分区的，可以根据 route table 中的 ID 是否在当前节点 n 和数据节点之间，如果在，则将该路由信息返回。

Algorithm 15 Greedy lookup

```
1: procedure n.TERMINATE(i)
2:     return i ∈ (n, succ]
3: end procedure

1: procedure n.NEXT HOP(i)
2:     if TERMINATE(i) then
3:         return succ
4:     else
5:         r := succ
6:         for j := 1 to K do
7:             if rt(j) ∈ (n, i) then
8:                 r := rt(j)
9:             end if
10:        end for
11:        return r
12:    end if
13: end procedure
```

贪心算法能够降低消息的传递量，并且也能够提升性能。

4. 组通信算法

对于一些批量操作/范围查找，例如，模糊查找等，需要遍历所有（一批）节点才能够找到所有请求的数据。为满足该需求，DHT 还可以提供组管理的算法。

组通信算法必须要满足以下要求：

- 可结束行，算法是可结束的；
- 覆盖性，所有被指派的节点都是能够访问到的，并且能够接收到请求的；
- 无重复性，即一个节点不会接收到多于一次相同的消息；

下面将简单介绍组通信算法，即广播算法（Broadcast）和批量操作算法（Bulk）。

4.1 广播算法

广播算法是指将请求转发到所有节点的，如下图所示。为了将请求发送到所有节点，在传递时，将从大到小地遍历 successor-list 列表中的节点，即首先将请求发送到[u(M), n)中，然后[u(M-1), u(M))，然后[u(M-2), u(M-1))，最后是[n+1, u(1))。可以知道，该算法是可结束行，并且覆盖到了所有节点，且一个节点不会接收到多次请求。

Algorithm 19 Simple broadcast algorithm

```
1: event n.StartSimpleBcast(msg) from app
2:     sendto n.SimpleBcast(msg, n)              ▷ Local message to itself
3: end event

1: event n.SimpleBcast(msg, limit) from m
2:     Deliver(msg)                              ▷ Deliver msg to application
3:     for i := M downto 1 do                    ▷ Node has M unique pointers
4:         if u(i) ∈ (n, limit) then
5:             sendto u(i).SimpleBcast(msg, limit)
6:             limit := u(i)
7:         end if
8:     end for
9: end event
```

有时，在广播算法中，期望每一个节点中能够返回一些数据。这时候可以对上面算法做简单修改，然后返回每个节点对应的数据。如下所示，Ack 表示期望返回的节点集合，par 表示消息来源的上一节点。基本思想就是对于最后的子节点，则直接返回数据，否则需要需要等待 Ack 中所有节点均返回结果，然后当前节点才能够返回数据。

Algorithm 20 Simple broadcast with feedback algorithm

```
1: event n.StartBcast(msg) from app
2:     sendto n.Bcast(msg, n)                    ▷ Local message to itself
3: end event

1: event n.Bcast(msg, limit) from m
2:     FB := Deliver(msg)              ▷ Deliver msg and get set of feedback
3:     par := n
4:     Ack := ∅
5:     for i := M downto 1 do                ▷ Node has M unique pointers
6:         if u(i) ∈ (n, limit) then
7:             sendto u(i).Bcast(msg, limit)
8:             Ack := Ack ∪ {u(i)}
9:             limit := u(i)
10:        end if
11:    end for
12:    if Ack = ∅ then
13:        sendto par.BcastResp(FB)
14:    end if
15: end event

1: event n.BcastResp(F) from m
2:     if m = n then
3:         sendto app.BcastTerm(FB)
4:     else
5:         Ack := Ack − {m}
6:         FB := FB ∪ F
7:         if Ack = ∅ then
8:             sendto par.BcastResp(FB)
9:         end if
10:    end if
11: end event
```

4.2 批量操作算法

批量操作算法是指将请求只发送到部分节点中的算法，最简单的算法可以在广播算法基础上加一个判断，即简单该节点是否操作集合中，如果存在则执行对应的操作；否则忽略该消息。但这存

在大量的数据报冗余。比较好的方法是根据路由信息确定是否将请求包发送到对应的节点中。

批量操作算法与广播算法类似，主要差别有：如果该节点数据操作集合中，则将捕获该数据包；

在转发时进行相关判断，即判断 successor-list 是否保护请求集合中，如果不存在则直接不转发该数据包。

Algorithm 21 Bulk operation algorithm

```
1:  event n.BULK(I, msg) from m
2:      if n ∈ I then
3:          Deliver(msg)                          ▷ Deliver msg to application
4:      end if
5:      limit := n
6:      for i := M downto 1 do                    ▷ Node has M unique pointers
7:          J := [u(i), limit)
8:          if I ∩ J ≠ ∅ then
9:              sendto u(i).BULK(I ∩ J, msg)
10:             I := I − J                        ▷ Same as I := I − (I ∩ J)
11:             limit := u(i)
12:         end if
13:     end for
14: end event
```

4.3 其他

在组通信中，因为一般周期较长，当中间存在错误时，一般采用超时来检测节点的失败等错误。此外，为了提升组通信的效率，可以在上一层引入通信组的概念，即将一批节点有效管理起来；另外，还可以将组通信包与 IP 广播包进行结合。

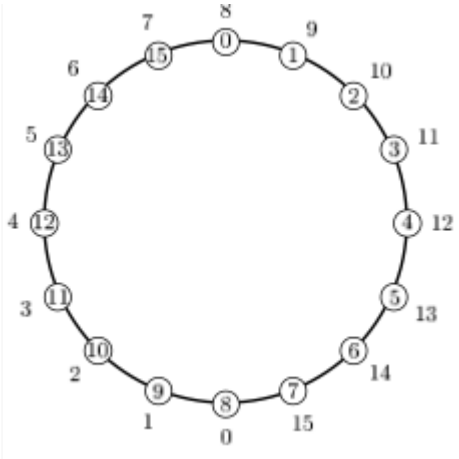
5. 副本管理算法

如果数据保存在一个节点上，那么一旦该节点发生错误或者宕机了，那么数据将永久丢失，因此，还需要对数据进行备份。这里简单介绍几种备份策略。

5.1 对称备份策略

对称备份是指将 Keyspace 划分成 N/f 个等价类的（f 为备份个数）。在同一个等价类内的所有 Key 都是相互关联的。也就是说，同一个等价类中的 key 都是同时出现的。如下图所示，0 和 8，1 和

9...，等都是等价类，这些是同时出现。对称备份的好处是，当一个节点加入或者，离开时，只需要搬迁邻居节点上的数据。



$$r(i,x) = i \oplus (x-1)\frac{N}{f}$$

$$1 \le x \le f, \quad f = 2.$$

$$1 \equiv 9 (\mathrm{mod}\ 8).$$

下面将介绍基于对称备份策略的节点加入/离开算法和数据管理算法。

5.1.1 节点管理算法

节点加入/离开算法主要涉及到节点中数据的搬迁。节点 n 的加入是指将 succ 中从 pred 到 n 的数据搬到新节点 n；节点 n 的离开是需要将 pred 到 n 的数据搬迁到 succ 中。

Algorithm 25 Symmetric replication for joins and leaves

```
1: event n.JoinReplication() from m
2:     sendto succ.RetrieveItems(pred, n, n)
3: end event

4: event n.LeaveReplication() from m
5:     sendto n.RetrieveItems(pred, n, succ)
6: end event
```

```
7: event n.RetrieveItems(start, end, p) from m
8:     for r := 1 to f do
9:         items[r] := ∅
10:        i := start
11:        while i ≠ end do
12:            i := i ⊕ 1
13:            items[r][i] := localHashTable[r][i]
14:        end while
15:    end for
16:    sendto p.Replicate(items, start, end)
17: end event

18: event n.Replicate(items, start, end) from m
19:    for r := 1 to f do
20:        i := start
21:        while i ≠ end do
22:            i := i ⊕ 1
23:            localHashTable[r][i] := items[r][i]
24:        end while
25:    end for
26: end event
```

5.1.2 数据管理算法

数据管理的相关算法需要针对数据备份进行简单修改。

在数据插入时，需要同时向对应的等价类中的所有节点插入对应的数据；

数据查找时候可以从等价类的中的所有节点查找；

为了提升性能，这里可以并发执行。

Algorithm 26 Lookup and item insertion for symmetric replication

```
1: event n.InsertItem(key, value) from app
2:     for r := 1 to f do
3:         replicaKey := key ⊕ (r − 1)N/f
4:         n.Lookup(replicaKey,AddItem(replicaKey, value, r))
5:     end for
6: end event

7: procedure n.AddItem(key, value, r)
8:     localHashTable[key][r] := value
9: end procedure

10: event n.LookupItem(key, r) from app
11:     replicaKey := key ⊕ (r − 1)N/f
12:     Lookup(replicaKey,GetItem(replicaKey, r))
13: end event

14: procedure n.GetItem(key, r)
15:     return localHashTable[r][key]
16: end procedure
```

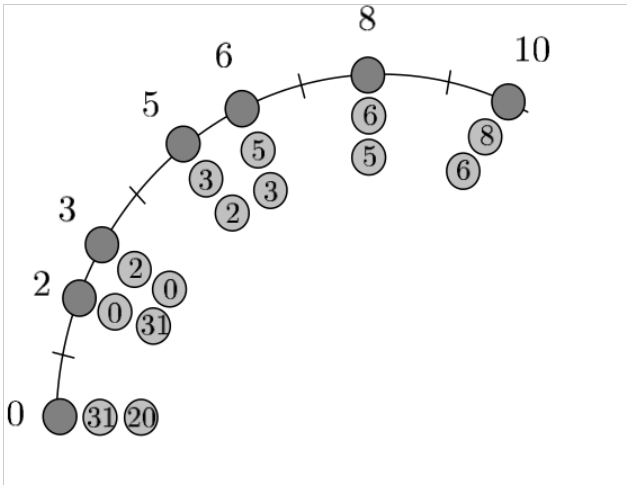
5.2 多哈希备份策略

是指利用 f 个哈希函数对 key 进行计算对应的 hash 值，然后将 f 个 hash 计算对应的节点，并将数据备份到这些节点上。该方法存在的问题是，需要 hash 函数的逆函数。

举个例子，如果 f 为 2，存在 H1 和 H2 两个哈希函数，一个节点 10 负责管理[5,10]的 key，如果一个 key 为 course 的得到 H1(course) = 7，这是由节点 10 负责的，假如节点 10 宕机了，节点 12 负责[5,12]，节点 12 需要从其他备份中 fetch 键为 course 的数据。然后节点 12 需要找到 course，这样 H2(course)才能够计算得到，找到对应的数据。

5.3 后续列表和叶集合

后续列表是指将数据直接备份到最近的 f 个后续节点上，如下图所示。



叶集合是指将数据备份到最近的 f/2 的前序节点和 f/2 的后续节点上（分别向下取整和向上取整）；

6. 分布式哈希表的应用

分布式哈希表在很多领域都有较广的应用。

6.1 底层存储

例如在存储系统中作为底层存储，PAST；在云文件系统 Cloud File System 中用来存在目录数据，在对象存储中保存 key 和数据。

A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Pro-ceedings of the 18th ACM Symposium on Operating Systems Princi-ples (SOSP’01)*, Chateau Lake Louise, Banff, Canada, October 2001. ACM Press. F. Dabek, M. F. Kaashoek, D. R. Karger, R. Morris, and I. Stoica.Wide-area cooperative storage with CFS. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP’01), pages202–215, Chateau Lake Louise, Banff, Canada, October 2001. ACM Press.

6.2 主机发现和移动

在移动环境中，将动态 IP 和地理位置信息作为 K/B 存储到 DHT 中。

P2PSIP. http://www.p2psip.org, 2006.

Host Identity Payload. http://www.ietf.org/html.charters/hip-charter.html, 2006.

I. Stoica, D. Adkins, S. Ratnasamy, S. Shenker, S. Surana, and S. Zhuang. Internet Indirection Infrastructure. In Proceedings of the First Interational Workshop on Peer-to-Peer Systems (IPTPS’ 02), Lecture Notes in Computer Science (LNCS), pages 191–202, London, UK, 2002. Springer-Verlag.

L. Zhou and R. van Renesse. P6P: A Peer-to-Peer Approach to Internet Infrastructure. In Proceedings of the 3rd Interational Workshop on Peer-to-Peer Systems (IPTPS’ 04), volume 3279 of Lecture Notes in Computer Science (LNCS), pages 75–86. Springer-Verlag, 2004.

6.3 网页缓存

在网页缓存中存储一些静态数据。当第一次请求时候，将数据缓存在服务端，避免了重复访问，提升性能。

S. Iyer, A. Rowstron, and P. Druschel. Squirrel: a decentralized peer-to-peer web cache. In Proceedings of the 21st Annual ACM Sym-BIBLIOGRAPHY 177posium on Principles of Distributed Computing (PODC’ 02), pages 213–222, New York, NY, USA, 2002. ACM Press.

J. Jernberg, V. Vlassov, A. Ghodsi, and S. Haridi. DOH: A Content Delivery Peer-to-Peer Network. In Proceedings of the 12th European Conference on Parallel Computing

(EUROPAR’ 06). Springer-Verlag, 2006.

6.4 其他应用场景

DHT 还可以运用到 Publish/Subscribe 系统和 P2P 系统中，例如，BitTorrent，Azureus，eMule 等系统中。

7 总结

本文简单地介绍了分布式哈希表，然后介绍了分布式哈希表中常用的算法，包括环原子管理算法，路由算法，组通信算法和副本管理算法，然后简单介绍了分布式哈希表的一些应用场景。