Journal of Computer Applications
计算机应用,2013,33(7):1885 - 1889
ISSN 1001-9081
CODEN JYIIDU
2013-07-01
http://www.joca.cn

文章编号:1001-9081(2013)07-1885-05
doi:10.11772/j.issn.1001-9081.2013.07.1885

基于分布式编程语言的 Chord 协议和算法

彭成章,蒋泽军*,蔡小斌,张志珂
(西北工业大学 计算机学院,西安 710129)
(*通信作者电子邮箱 claud@nwpu.edu.cn)

摘　要:P2P 分布式哈希表(DHT)协议本身简洁并且易于理解,但是命令式语言与分布式架构的不匹配使得实现和部署一个拥有全部功能的类似 Chord 的组件相当困难和复杂。针对这些问题,提出一种基于 Bloom 系统来设计 P2P 分布式哈希表协议的方法。首先,阐述了 Bloom 系统的分布式逻辑编程语言要素;其次,设计了一个最小分布式系统;再次,通过定义永久、暂时、异步通信和周期集合,设计了指表维护算法、后继列表算法以及维持稳定算法等,实现一个 Chord 原型系统。实验结果证明,原型系统能完成 Chord 所有功能,并且与传统语言相比,代码量减少 60%。分析表明最终的算法代码和分布式哈希表协议规范高度一致,不仅增强了代码的可读性和重用性,而且加深了对协议本身及其应用的理解。

关键词:P2P;分布式哈希表;逻辑编程;Chord;Bloom
中图分类号:TP311.133.1　　**文献标志码**:A

Chord protocol and algorithm in distributed programming language

PENG Chengzhang, JIANG Zejun*, CAI Xiaobin, ZHANG Zhike
(*School of Computer Science, Northwestern Polytechnical University, Xi'an Shaanxi 710129, China*)

Abstract: The Peer-to-Peer (P2P) Distributed Hash Table (DHT) protocol is concise, and can be understood easily, but implementing and deploying a component like Chord with all functions in practice is very difficult and complicated because of the mismatch between popular imperative language and distributed architecture. To resolve these problems, a P2P DHT protocol based on Bloom system was proposed. Firstly, the distributed logic programming language's key elements of Bloom system were expounded. Secondly, a minimal distributed system was designed. Thirdly, a Chord prototype system was implemented through defining persistent, transient, asynchronous communicating and periodic collections and designing several algorithms for finger table maintaining, successor listing, stabilization preseving and so on. The experimental results show that the prototype system can finish full functions of Chord, and compared to traditional languages, 60% of the code lines can be saved. The analysis indicates such a high degree of uniformity between final code of the algorithm and the DHT protocol specification makes it more readable and reusable, and helpful for further understanding the specific protocol and relative applications.

Key words: Peer-to-Peer (P2P); Distributed Hash Table (DHT); logic programming; Chord; Bloom

0　引言

随着云计算和云存储的发展,快速构建出分布式原型系统来验证新的思想和算法成为热点。对于原型系统,首先需要实现或借用已有的一个或多个分布式协议,P2P 分布式哈希表协议就是其中之一,被广泛地使用在各种云计算和云存储核心组件中,以著名的 Dynamo[1] 和 Cassandra[2] 为例,它们都使用分布式哈希表和基于 Quorum 的备份控制协议。但是,可供研究所用并且独立实现的协议相当少,一般都是作为组件而存在某些系统中,很难将其分解出来,供相关实验所用。

对 Chord 协议及其实现进行分析得出结论,由于传统编程语言(命令式语言)如 C、C++ 和 Java 遵循经典的冯·诺依曼架构[3-4],编写的命令式语言程序按照顺序执行,而分布式计算系统无法严格按照固定的顺序执行,如并发、网络延时和网络通信。为了解决命令式语言和分布式系统的不匹配,针对不同的协议,必须设计特定的事件处理、并发控制、序列化组件[5]。因此这类协议,以 Stoica 等[6] 的原型系统为例,由数千行精心设计的代码组成,阅读和理解这些代码,需要良好的学术和工程背景,更不用说以此为基础验证创新点。

Hellerstein[7] 提出用扩展的逻辑语言 Datalog((声明式逻辑编程语言))构建各种覆盖网络协议和分布式系统,并产生了最具技术创新性[8] 的 Bloom 系统来支持基于逻辑规则的编程,Chord 及其类似的协议是高级抽象的,可以总结为状态发现、状态更新、信息交换和路由表状态维护等,它们非常适合使用 Bloom 系统所提供的逻辑声明语言来表达。在此之前,Alvaro 等[4] 用分布式编程语言 OverLog 实现了一致性协议 Paxos,Borkar 等[9] 重新实现大数据分析平台 MapReduce,文献[10]构建了 Chord 协议的静态环。

本文首先简单介绍了 Bloom 系统,为了对基于 Bloom 的

收稿日期:2013-01-31;**修回日期**:2013-03-13。
基金项目:航空科学基金资助项目(2010ZD53042,2012ZC53040);陕西省自然科学基金资助项目(2010JM8023)。
作者简介:彭成章(1977 -),男,湖南岳阳人,博士研究生,CCF 会员,主要研究方向:云计算、分布式数据流处理;　蒋泽军(1964 -),男,陕西西安人,教授,主要研究方向:云计算、云存储、网络与信息安全;　蔡小斌(1957 -),男,江西黎川人,教授,博士生导师,主要研究方向:虚拟实验;　张志珂(1984 -),男,河南巩义人,博士研究生,主要研究方向:重复数据删除。

分布式系统有整体的认识,设计并实现了最简单的 ping-pong 程序;然后讨论了使用 Bloom 系统如何精确地实现一个完整的 Chord 协议,并选择了集合定义、指表维护算法、后继列表算法和维持稳定算法四个重要的组成部分进行了详细的描述;接着讨论了可视化数据流分析工具;最后对全文进行了总结。

1 Bloom 系统

Bloom[6,11]是一个让程序员以分布式逻辑编程语言编写应用程序的系统,在其内部,主要包含扩展的 Datalog[12]引擎和快速的事件处理框架。前者提供关于时间而不是分布式系统空间的编程语言,因此程序员只要将注意力集中在数据、不变性和随着时间会产生变化的内容,不需要关心空间上的布局(当然,也不需要理解 Datalog 解析器/编译器的行为和操作),因此也称 Bloom 系统为 Bloom 语言;后者提供大规模的事件驱动和异步消息,以及网络监控。这两个特征使得 Bloom 特别适合设计各种分布式协议或分布式系统。

1.1 Bloom 程序

从语言的角度,Bloom 可以称为基于 Datalog 的语言,Datalog 是一种真正的声明式逻辑编程语言(可以简称为逻辑语言或声明式语言),综合来说,它是 Prolog 的一个子集,常常用作推演数据库的查询语言,在数据集成、信息获取、网络、编程分析、安全和云计算中都有越来越多的应用。Datalog 程序将所有计算的内容表示为规则,根据规则主体(右边)的关系,产生规则头部(左边)的关系,头部的关系是一条规则的最终结果,下面是一个计算图的传递闭包的 Datalog 的例子:

1) TransitiveClosure(x,y) :-Graph(x,y);

2) TransitiveClosure(x,y) :-Graph(x,z),Transitive(z,y)。

其中":-"意为推导,","表示连接。可以看出,规则的定义是递归的,因此所定义的关系可以出现在规则的左边,也可以出现在规则的右边。这样,一个 Datalog 程序,可以解释为:只有出现在规则主体的关系实例,能映射成出现在规则头部的关系实例。显然,Datalog 程序的语句可以定义成不动点等式的特定方法,使其总是迭代一个规则求值,一直到该迭代达到了特定的不动点为止。用不动点方法来讨论上面的例子中的规则 1),从关系 Graph 开始,通过替换规则 1)中右边关系的第一个元素来产生规则头部关系中的相应元素,直到所有元素被迭代过一次为止(称为自底向上方法)。这样的规则集合总是表达了如下限制:基本事实和传递结果在不动的位置肯定是一致的。与 Datalog 类似,Bloom 程序的语句由多个事实构成的集合(相当于 Datalog 的关系)以及对它们的归并操作和关系操作构成,每个语句的形式为:

< 目标集合 > < 归并操作 > < 集合表达式 >

每个时间步,Bloom 系统对右边的集合表达式进行求值,其结果根据语句的归并操作类型,在当前时间步或下一个时间步的开始,归并到左边的目标集合中。对 Bloom 系统的集合、操作和集合表达式中所用的关系操作解释如下。

1)集合。所有的集合都存于本地节点中,集合构成 Bloom 的数据模型,借用 Ruby 的类型系统完成。所有的集合都声明在 state 语句块中,每个集合由 schema 所定义的无序事实组成,即规定事实的列名,某个列的子集合,形成决定其他列或者整个集合特征的键。Bloom 提供了 5 种集合类型,见表 1。

表 1 Bloom 的集合类型

类型名称	意义
table	永久存储,直到程序被中断
scratch	每个时间步暂存
channel	节点之间异步通信,在 schema 中第一列名字前加"@"字符,表示通信目的地
periodic	定义周期时间,定时触发时间事件
temp	语句中临时定义

2)归并操作。将右边计算结果在确定的时间或不确定时间内归并到左边。分别由表 2 提供的四种操作完成。

表 2 Bloom 的操作

操作符	操作名称	意义
<=	即刻归并	当前时间步内,将右边集合表达式的结果即刻归并到左边的集合中
<	异步归并	在本地未来的某个时间步内,将右边集合表达式的结果,异步归并到左边集合中的位置标识符所代表的节点中
<+	延迟归并	在下一个时间步开始时,将右边集合表达式的结果归并到左边的集合中
<-	延迟删除	在下一个时间步开始时,从左边的集合中删除右边集合表达式的结果
<+-	延迟归并和删除	<+ 和 <- 的组合

3)关系操作。对集合的逻辑运算通过关系操作来完成,包括选择、投影、聚集(求和、最大值、最小值和分组等)和联合,同时支持 Ruby 语句块来达到精确计算。

因此对 Bloom 程序的计算,即对所有本地节点中基于集合的操作进行求值。每个节点中的 Bloom 程序称为一个 Bloom 实例,各个实例之间使用异步消息进行通信。

Bloom 程序的运行由一系列的称为原子的"时间步"来定义,每个时间步由连续的三个阶段 setup、logic 和 tansition 组成。在 setup 阶段,事件到达如与某些集合相关的定时或网络消息;在 logic 阶段,基于当前状态和新到达的事件,对所有满足条件的语句进行求值,生成临时的事实,有的需要修改本地节点状态或发送消息给远端(称为"副作用"),这些操作将延迟到下一个时间步开始时来完成;在 transition 阶段,将 logic 阶段生成的事实存储到节点所定义的集合中。

1.2 最小应用举例

本节讨论一个常见的 ping-pong 分布式小程序。首先,构建一个只有两个节点的分布式小系统,假设 ping 程序位于节点 1,启动 ping 后,它会不停地向指定的另一节点发送 ping 消息;pong 程序位于节点 2,启动 pong 后,它等待 ping 消息,一旦收到 ping 消息,会向 ping 所在节点发送 pong 消息。在这之后,将讨论 Bloom 系统如何能将前面的程序扩展称为一个主节点和多个从节点的分布式系统。

算法 1 所示为 ping 节点的主要程序,程序中 Bloom 系统是使用 Ruby 的元编程实现的,所有 Bloom 程序都由 state 语句块和 bloom 语句块组成,并嵌入到 Ruby 中,所有集合使用之前必须在 state 定义(temp 类型集合除外),所有语句都在 bloom 中定义(回调方法除外)。第 4) ~ 6)行定义了 pingMsg、ping 和 pong 集合,第 9)行语句的意义为:当 pingMsg 在某一个时间步收到有新元组或说元组更新时,其新元组将

会归并到 ping 集合中(因为 ping 为 channel 类型,在下一个时间步开始,ping 的元组将会发送到 ping 中的 dest 所标识的节点)。

算法 1　Bloom 的 Ping 程序。

```
1)    class Ping
2)      include Bud
3)      state do
4)        table : pingMsg, [ : dest, : src, : eventNo]
5)        channel : ping, [ : @ dest, : src, : eventNo]
6)        channel : pong, [ : @ dest, : src, : eventNo]
7)      end
8)      bloom : ping do
9)        ping <~ pingMsg { |p| [ p. dest, p. srcAddr,
            p. eventNo] }
10)     end
11)   end
```

从算法 1 可以看出,必须给 pingMsg 集合新数据,否则这个语句就永远不执行,这可以通过在 bloom 语句块之外使用回调方法完成,下面是在 Ruby 程序中更新 pingMsg 集合的方法:

```
1)    pingIns = Ping. new
2)    pingIns. sync_do {
3)    pingIns. pingMsg <= [[ destIP, src, rand( ) ] ] }
```

pong 程序与 ping 程序类似,channel 类型的集合需要通信,所定义的 schema 必须一致,因此在 pong 的 state 块中需要定义算法 1 中的 ping 和 pong 集合,bloom 块只需要下面一句:

```
pong < ping p [ p. srcAddr, p. dest, p. eventNo ]
```

表示接收到 ping 消息时,将 ping 集合的 src 作为目的地址,原样返回给 ping 节点。

运行这个系统,并且将 pong 程序作为监听程序(也称服务器程序),ping 程序为发起程序(也称客户端程序)。在 Bloom 系统中,提供了 run_fg 方法来让其处在监听状态,使用 run_bg 方法来让其处在发起状态。

为了更清晰认识 Bloom 系统,将其与传统语言构成的 ping-pong 系统作个简要比较。以 C 或 Java 为例,对于创建单客户端,并且服务器端在同一时间只处理单个客户端来说,Bloom 程序所表示的逻辑除了不需要关心 socket 和创建进程等低级处理和逻辑更加清晰易懂外,优势并不明显。但是,如果需要构建一个能同时处理多个客户端的 pong 服务器的话,那么传统的方法将变得非常复杂。下面是构建一个能处理多个客户端请求的一般步骤:每次监听到一个客户端的请求时,创建一个子进程来处理连接该客户端,或者先进一点的,采用多线程方法,然后用一些方法来处理相关的资源问题。这很容易引起分布式系统典型的并发问题,但在 Bloom 系统中,使用目前最先进的事件驱动架构 EventMachine[13],因此设计者不需要处理这些细节,当 pong 端收到来自不同源地址(多个客户端)的 ping 时,不需要对上面的 Bloom 程序做任何修改就自动扩展成为典型的单主节点和多从节点的分布式系统。

2　Chord 协议和算法的 Bloom 实现

Chord[6]是四个最经典的 P2P 分布式哈希表协议和算法之一,其原理直观易懂,特点为去中心化控制、负载均衡、规模易扩展、可用性高和节点命名灵活等,其主要思想可以描述如下。

1)使用一致性哈希算法,将每台参与该系统的计算机(每个机器称为节点)IP 地址作为键,得到一个对应值,这些值按照一定规则构成 Chord 环,使用节点、后继和前驱记录自己和相邻节点,以备路由查询。

2)当环中的任何一个节点收到查询请求时,通过节点和它的后继来判断,所请求的数据是否存在后继中,如果没有找到,就将请求提交给最合适的后继,用指表来提高查询速度及容忍单点失败。

3)在环中的每个节点都需要定时去检查自己的后继节点是否活跃,如果后继为死节点,则需要更新相关数据结构的内容。

按照 Chord 协议规范以及 Bloom 系统的特点,定义对应的集合;为构建环、维护环以及接受环外查询,需要提供完整的查询算法;准备参与 Chord 环中每个节点,除了第一个节点外,首先需要与环中的某个节点进行连接;在每个节点接入环中后,新节点以及相邻节点的前驱和后继都将发生变化,通过维持稳定算法来完成;新的后继加入后,某些节点的直接后继将必须发生变化;为了以后更好地实现维护和路由查询的目的,添加指表维护算法;系统运行过程中,单点故障随时发生,容忍失败算法提供监听和失败处理。本章将讨论 Chord 协议和算法中定义的集合以及几个重要算法的 Bloom 实现。

2.1　定义集合

讨论所有定义集合超出了本文的范围,只挑选一些主要的集合进行讨论。构建 Chord 环的过程,可以采用任何环外客户端对环进行访问时的处理流程(以达到代码重用),即查询请求和结果响应,lookupReq 和 lookupResults 就被定为 channel 集合;在 Chord 环中的节点需要与它的前驱和后继列表中的节点进行信息交互。从 1.1 节知,Bloom 系统中,所有数据都是本地而不是整个分布式系统中共享,因此需要定义 channel 集合 prec_ask、prec_ans、succ_ask 和 succ_ans 来获取相邻的前驱和后继;任何时候,节点必须知道自己的直接前驱和后继,分别为 bestSucc 和 precursor;Chord 协议中使用指表来避免查找时不需要路由所有的节点就可以得到结果,用如下几个集合:finger、nextFingerFix、eagerFinger 和 uniquefinger 等;处理由于单点失败而导致环无法路由,使用 successors 来保存 4 个后继节点信息;除了这些 channel 集合和永久集合外,在 Chord 环运行的每个时间步,由于定时器时间到,或其他原因,需要定义大量的 scratch(临时)集合,如定时发起接入环中的事件集合 joinEvent,当有新的后继节点时,触发 newSuccEvent,有节点失败时触发 nodeFailure 等;最后需要定义定时器集合,如 joinPeriod、stabPeriod 和 pingPeriod。

2.2　指表维护算法

节点的 Chord 程序运行后,每隔 10 s 生成一个指表维护事件。fFixEvent 包括 nextFingerFix 提供索引号(范围是 0 ~ 159,160 项的目的是为了能适用于大规模的分布式环境)和随机产生的事件号,事件号用于跟踪与其他节点的交互。初始时,根据指表原理,nextFingerFix 的索引号为 0,因此形成一个如下 2^0 + node. id 的查询值,继而生成一个向本节点的查询请求 lookupReq,如算法 2 第 1) ~4)行,找到该查询值所在的后继节点,把这个后继添加到指表 finger 中。如果

```
2^i + node. id; i < 160
```

所得到的值在节点和找到的后继之间,说明它们的后继都是一样的,应该将它们也添加到指表中。因此每次得到一个 lookupResults 后,将 i 进行加 1 操作,重新判断是否位于节点和该后继之间(如算法 2 第 5) ~12)行),如果为真,则添加到指表中,否则形成新的查询。为了完成这个操作,引入了

eagerFinger 表,该表的第一个作用是用于暂存符合添加到 finger 的内容,第二个作用是当 i 的累加导致上式生成的值在找到的后继之后,那么这个新的值将插入到 nextFingerFix,它会在下一个 10 s,插入到 fFixEvent 中,进而生成一个新的查询请求来找到符合要求的后继(如算法 2 中第 13) ~ 19)行)。依此类推,直到 i 的值为 159。

算法 2 指表维护算法。

```
1)  lookupReq <~(fFixEvent * identifier).pairs do |f, i|
2)    kk = (0x1 << f.index) + i.myID
3)    [f.myAddr, kk, i.myAddr, f.eventNo]
4)  end
5)  eagerFinger <+- (identifier * eagerFinger).pairs do |i, e|
6)    if e.start < MAXFINGER
7)      ii = e.start + 1; kk = (0x1 << ii) + i.myID
8)      if rangeOO(kk, i.myID, e.btwnID)
9)        [i.myAddr, ii, e.btwnID, e.btwnAddr]
10)     end
11)   end
12) end
13) nextFingerFix <+-(identifier * eagerFinger).pairs do |i, e|
14)     ii = e.start + 1
15)     kk = (0x1 << ii) + i.myID
16)     if rangeOO(kk, e.btwnID, i.myID)
17)       [i.myAddr, ii]
18)     end
19) end
```

2.3 后继列表算法

当后继节点插入到后继列表中,或因为后继列表中某个节点失败而从列表中删除时,产生一个后继更新事件 newSuccEvent,继而确定直接后继并更新指表的第一项。为了找到这个直接后继,根据 Chord 协议对后继节点的定义和 Bloom 系统的特点,定义一个距离集合 succDist,将后继列表中每个节点 ID 与节点本身的 ID 相减,并再减 1,得到的结果称为后继距离,所有距离都插入距离集合 succDist,分两种情况来找到直接后继 bestSucc:

1)如果 succDist 中所有元组为正数或负数,则直接后继为距离最小的后继;

2)如果 succDist 中所有元组既有正数,也有负数,则直接后继为正数中最小的后继。

表 3 后继距离算法示例

节点	successors				succDist			
100	100	200	300	400	-1	100	200	300
200	200	300	400	100	-1	100	200	-100
300	300	400	100	200	-1	100	-200	-100
400	400	100	200	300	-1	-300	-200	-100

举个例子说明,假设有 4 个节点,每个节点的 ID 分别为 100,200,300 和 400,并后继列表中只存储 4 个后继,根据前面对后继及后继距离的定义,则每个节点的后继距离表 3 所示。根据以上算法,在 Bloom 中,显然需要用 succDistP 和 succDistM 分别表示正数和负数距离表,并且这两个集合是在特定条件下才需要定义,因此它们的类型定义为 temp。如果正数距离表不为空,则不需要考虑负数距离表,直接求出直接后继 bestSucc;如果正数距离表为空,则从负数距离表中求出直接后继 bestSucc;用 Bloom 提供的 none? 方法。

2.4 维持稳定算法

当 Chord 环中只有一个节点时,前面讨论过它会自己构成一个环,并且在环已经存在时,新加入的节点所做的第一件事情就是找到自己的后继。除此之外,环中的某些节点因为新加入的节点而受到影响,需要重新调整自己的前驱和后继列表。为此,定义一个 5 s 的表,来触发维持稳定的事件 stabilizeEvent。

先来考虑环中只有一个节点,ID 为 100,现在有一个新的节点,ID 为 200,向环中唯一的节点发送接入请求 joinReq,环中的节点收到请求后,从自己的直接后继 bestSucc 开始查找该值 200 的后继,由于只有一个节点,bestSucc 为自己,因此环中唯一的节点成为请求节点 200 的后继。

算法 3 所示为维持稳定算法代码,在下一个 stabilizeEent 到来后,如算法 3 第 1) ~ 5)行,节点 200 向节点 100 发送一个 prec_ask 请求,希望得到节点 100 的前驱,节点 100 此时前驱为空,同时根据刚才的接入请求 joinReq,节点 100 为节点 200 的后继,因此可以确定节点 200 为节点 100 的前驱(算法 3 中 6) ~ 14)行)。在下一个时间步开始时,节点 100 向节点 200 发送 prec_ask 请求,也得到前驱为空的结果,同样的推理,最后确定节点 100 为节点 200 的前驱。

算法 3 维持稳定算法。

```
1)  prec_ask <~ (stabilizeEvent * successor).pairs {|st, s|
2)    [s.succAddr, ip_port] }
3)  prec_ans <~ (prec_ask * precursor).pairs do |a, p|
4)    [a.srcAddr, p.precID, p.precAddr, p.myAddr,
        a.asker]
5)  end
6)  precSend <~ (prec_ans * successor * \ identifier).
      combos do |p, s, i|
7)  if ((p.precAddr == "NIL") ||
8)      (rangeOO(i.myID, p.precID, s.succID))) and
9)     (i.myAddr != s.succAddr) and
10)    (s.succAddr == p.ansAddr)
11)   [s.succAddr, i.myID, i.myAddr, "precSend"]
12)   end
13) end
14) precursor <+- precSend do |p|
15)   [p.dest, p.precID, p.precAddr]
16) end
```

两个节点的前驱节点确定以后,现在的状态为:ID 为 100 的节点的后继是自己,前驱为节点 200,而节点 200 的后继是 100,前驱节点也为 100。另外,根据前面的讨论得知,节点 200 的后继列表 successors 中的后继节点为自己和节点 100,而节点 100 的后继列表 successors 中只有节点 100。在下一个 stabilizeEvent,节点 100 向节点 200 发送请求 succ_ask,来获得节点 200 的所有后继,从 succ_an 返回的结果,将全部插入 successors 中。于是就构成了一个拥有两个节点的环,以此类推。

3 实验分析

算法代码量分析:与最初 Chord 原系统的 C ++ 语言实现使用 3 000 多行代码[14]相比,基于 Bloom 系统的实现为不超过 1 000 行。

Bloom 系统的数据流分析:在设计、实现和调试分布式系统协议的时候,构建或使用工具生成数据流图是非常有帮助的。Bloom 系统提供了将程序生成数据流图的方法,2.3 节的后继列表算法经过工具编译后所得到的数据流图如图 1 所示。

4　结语

本文所讨论的基于分布式逻辑编程语言来构建覆盖网络协议具有理论和现实意义:一方面,逻辑编程语言从早期应用于数据库,发展到现在不断成功地在数据集成、信息获取、网络、编程分析、安全等领域得到广泛应用;另一方面构建新的编程模型或者称为领域专用语言[15]来解决新的问题一直是研究的热点,例如 SQL、NoSQL 和 MapReduce 等。

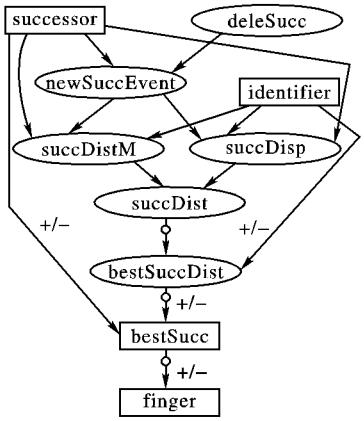

图 1　后继列表算法数据流图

通过分析 Chord 以及类似的 P2P 分布式哈希表协议在实现和研究过程中所面临的问题及其原因,对 Bloom 系统以及基于 Bloom 构建的 Chord 协议和算法进行研究和实验,获得对 Chord 协议精确的实现。实现的原型系统在概念上能与最初的协议规范更接近,为深刻理解分布式逻辑编程语言非常适合构建分布式系统奠定良好的基础。另外这里所实现的 Chord 协议已经成为一个分布式键值存储原型系统的组件,下一步将研究 Bloom 系统在网络存储系统中的应用。

参考文献:

[1] DECANDIA G, HASTORUN D, JAMPANI M, *et al*. Dynamo: amazon' s highly available key-value store [J]. ACM SIGOPS Operating Systems Review, 2007, 41(6): 205 – 220.

[2] LAKSHMAN A, MALIK P. Cassandra: structured storage system on a P2P network [C]// PODC'09: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing. New York: ACM Press, 2009: 5 – 5.

[3] MARCZAK W R, ZOOK D, ZHOU W, *et al*. Declarative reconfigurable trust management [C]// CIDR'09 : Proceedings of the Fourth Biennal Conference on Innovative Data Systems Research. Asilomar: [s. n.], 2009: 21 – 29.

[4] ALVARO P, CONDIE T, CONWAY N, *et al*. I do declare: consensus in a logic language [J]. ACM SIGOPS Operating Systems Review, 2010, 43(4): 25 – 30.

[5] WELSH M, CULLER D, BREWER E. SEDA: an architecture for well-conditioned, scalable Internet services [J]. ACM SIGOPS Operating Systems Review, 2001, 35(5): 230 – 243.

[6] STOICA I, MORRIS R, KARGER D, *et al*. Chord: a scalable peer-to-peer lookup service for Internet applications [C]// SIGCOMM' 01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications. New York: ACM Press, 2001: 149 – 160.

[7] HELLERSTEIN J M. Datalog redux: experience and conjecture [C]// PODS'10: Proceedings of the Twenty-ninth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. New York: ACM Press, 2010: 1 – 2.

[8] HELLERSTEIN J. TR10: cloud programming [EB/OL]. [2010 – 04 – 20]. http://www2. technologyreview. com/article/418545/tr10 – cloud – programming.

[9] BORKAR V, CAREY M, GROVER R, *et al*. Hyracks: a flexible and extensible foundation for data-intensive computing [C]// ICDE' 11: Proceedings of the 2011 IEEE 27th International Conference on Data Engineering. Piscataway: IEEE Press, 2011: 1151 – 1162.

[10] PENG C Z, JIANG Z J, CAI X B, *et al*. Implementing chord with declarative networking language [C]// ICMLC'12: Proceedings of the 2012 International Conference on Machine Learning and Cybernetics. Piscataway: IEEE Press, 2012: 586 – 590.

[11] HELLERSTEIN J. Bloom programming language [EB/OL]. [2011 – 03 – 21]. http://www. bloom-lang. org.

[12] ABITEBOUL S, HULL R, VIANU V. Foundations of databases: the logical level [M]. Boston: Addison-Wesley, 1994.

[13] GRIGORIK I. Ruby event machine—the speed demon [EB/OL]. [2008 – 05 – 27]. http://www. igvita. com/2008/05/27/ruby-eventmachine-the-speed-demon.

[14] FRANK D, KAASHOEK M F, KARGER D, *et al*. Wide-area cooperative storage with CFS [C]// SOSP'01: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles. New York: ACM Press, 2012: 202 – 215.

[15] CONWAY N, MARCZAK W R, ALVARO P, *et al*. Logic and lattices for distributed programming [C]// SoCC'12: Proceedings of the Third ACM Symposium on Cloud Computing. New York: ACM Press, 2012: 1 – 14.

(上接第 1869 页)

[4] KAPOTAS S K, SKODRAS A N. Real time data hiding by exploiting the IPCM macroblocks in H. 264/AVC streams [J]. Journal of Real-Time Image Processing, 2009, 4(1): 33 – 41.

[5] ZOU D, BLOOM J A. H. 264 stream replacement water-marking with CABAC encoding [C]// Proceedings of the 2010 IEEE International Conference on Multimedia and Expo. Piscataway: IEEE Press, 2010: 117 – 121.

[6] 毕厚杰, 王健. 新一代视频压缩编码标准——H. 264/AVC[M]. 2 版. 北京: 人民邮电出版社, 2009.

[7] 吕明洲. 基于 DM6467 DSP 处理器的多通道 H. 264 编码软件设计[D]. 杭州: 浙江大学, 2012.

[8] GONZALEZ R C, WOOD R E. Digital image processing [M]. 2nd ed. 阮秋琦, 阮宇智, 译. 北京: 电子工业出版社, 2007.

[9] NOORKAMI M, MERSEREAU R M. A framework for robust watermarking of H. 264 encoded video with controllable detection performance [J]. IEEE Transactions on Information Forensics and Security, 2007, 2(1): 14 – 23.

[10] 张维纬, 张茹, 刘建毅, 等. 基于纹理特征的 H. 264/AVC 顽健视频水印算法[J]. 通信学报, 2012, 33(3): 82 – 89.

[11] KIM D W, CHOI Y G, KIM H S, *et al*. The problems in digital watermarking into intra-frames of H. 264/AVC [J]. Image and Vision Computing, 2010, 28(8): 1220 – 1228.

[12] XU D W, WANG R D, WANG J C. A novel watermarking scheme for H. 264/AVC video authentication [J]. Image Communication, 2011, 26(6): 267 – 279.