基于改进 Kademlia 协议的分布式爬虫①

陶耀东[1], 向中希[1,2]

[1](中国科学院 沈阳计算技术研究所, 沈阳 110168)
[2](中国科学院大学, 北京 100049)

摘 要: 随着互联网信息的爆炸式增长, 搜索引擎和大数据等学科迫切需要一种高效、稳定、可扩展性强的爬虫架构来完成数据的采集和分析. 本文借助于对等网络的思路, 使用分布式哈希表作为节点间的数据交互的载体, 同时针对网络爬虫自身的特点, 对分布式哈希表的一种实现——Kademlia 协议进行改进以满足分布式爬虫的需求. 在此基础上设计并完善了具有可扩展性和容错性的分布式爬虫集群. 在实际试验中, 进行了单机多线程实验和分布式集群的实验, 从系统性能角度和系统负载角度进行分析, 实验结果表明了这种分布式集群方法的有效性.

关键词: 分布式哈希表; P2P; 网络爬虫; Kademlia 协议; 去中心化

Distributed Crawler Based on the Improved Kademlia Protocol

TAO Yao-Dong[1], XIANG Zhong-Xi[1,2]

[1](Shenyang Institute of Computing Technology, Chinese Academy of Sciences, Shenyang 110168, China)
[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: With the explosive growth of Internet information, researches on search engine and big data call for an efficient, stable and scalable crawler architecture to collect and analyze Internet data. Inspired by peer to peer network, we use distributed hash table as a carrier of communication between nodes, while a distributed hash table implementation—Kademlia protocol is modified and improved to meet the needs of the distributed crawler cluster's scalability and fault tolerance. In the experiments, we carried out multi-threaded experiment on single computer and node expansion experiment on distributed cluster. From system performance and system load point of view, the experimental results show the effectiveness of this kind of distributed cluster.

Key words: distributed hash table; peer to peer; network crawler; Kademlia protocol; decentration

随着互联网时代的来临, 网络信息呈指数级增长. 传统的网络爬虫已渐渐不能满足互联网搜索引擎和大数据分析的需要[1], 而基于中心调度的主从式的爬虫也因为网络负载高、扩展相对困难、广域网部署困难[2,3]等原因发展缓慢, 因此全分布式、易扩展的网络爬虫架构[4-6]成为了学术界和工业界的优选方案.

对等网络(Peer-to-Peer Networks, P2P)是一种采用对等策略计算模式的网络, 是近年来较为流行的一种网络架构[7]. 网络的参与者共享他们所拥有的部分硬件资源(CPU、内存、硬盘、带宽等), 这些共享资源能被其他对等结点直接访问而无需经过中间实体. 在此网络中的参与者既是资源(服务和内容)提供者, 又是资源(服务和内容)获取者. 这种网络体系可以满足全分布式架构的需要.

快速高效资源检索是 P2P 网络体系的核心问题. 其主要检索方式经历了中心索引服务器、非结构化覆盖网络、结构化覆盖网络这几个阶段. 结构化覆盖网络基于分布式哈希表(Distributed Hash Table, DHT)的技术, 具有无需中心索引服务器、查找速度快、网络开销小等优点, 在实际的大规模的 P2P 网络环境中

① 基金项目:沈阳市科技计划(F14-056-7-00)
收稿时间:2015-07-21;收到修改稿时间:2015-09-14

© 中国科学院软件研究所 http://www.c-s-a.org.cn

被广泛使用. DHT 的代表实现有 Chord[8], Kademlia [9] 等. 而在实际使用中较广泛的是 Kademlia 协议, 其主要应用有 eMule、Bitcomet.

1 Kademlia 协议

1.1 距离度量

在 Kademlia 中, 每个节点都在初始化时被分配了一个 160 位的节点 ID ,同时 DHT 网络中的所有资源(<Key, Value>键值对)也是用 160 位的标识键(Key)来表示. 一个节点通常存储那些与它距离相近的资源. 每一个资源的键一般通过哈希函数计算生成, 而节点 ID 是节点自身随机生成的. 在 DHT 网络中, 所有距离(节点与节点、节点与资源)都是通过异或运算产生的. 可以表示为:

$$d(x,y) = x \oplus y$$

x, y 可以是一个节点 ID 或者是资源的标识键. 因为距离是基于节点 ID 的, 而节点 ID 是随机生成的, 所以距离相近并不意味物理距离上的相近.

1.2 K 桶

Kademlia 的路由表是通过一些称之为 K 桶的表格构造起来的. 对每一个 0≤ i ≤160, 每个节点都保存有一些和自己距离范围在区间$[2^i, 2^{i+1})$内的一些节点信息, 这些信息由一些(IP 地址,UDP 端口,Node ID) 的数据列表构成(DHT 网络是靠 UDP 协议交换信息的). 每一个这样的列表都称之为一个 K 桶, 并且每个 K 桶内部信息存放位置是根据上次看到的时间顺序排列, 最近看到的放在头部, 最后看到的放在尾部. 每个桶都有不超过 k 个的数据项.

一个节点的全部 K 桶列表如表 1 所示.

表 1 K 桶的结构表

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

当 i 值很小时, K 桶通常是空的(也就是说没有足够多的节点, 比如当 $i=0$ 时, 就最多可能只有 1 项); 而当 i 值很大时, 其对应 K 桶的项数又很可能会超过 k 个(覆盖距离范围越广, 存在较多节点的可能性也就越大), 这里 k 是为平衡系统性能和网络负载而设置的一个常数, 但必须是偶数.

1.3 RPC 操作

Kademlia 定义了 4 种 RPC(Remote Procedure Call)操作, 它们分别是 PING、STORE、FIND_NODE、FIND_VALUE.

① PING 操作允许一个节点来检测另一个节点是否在线. 同时每个 PING 的回复包含了回复者的节点 ID.

② STORE 操作是用来存储资源到 DHT 网络中, 通知一个节点存储一个<Key, Value>对, 以便以后查询需要.

③ FIND_NODE 操作是用来查找另一个节点. 当一个节点需要查找另一个节点的时候, 它会发起一个 FIND_NODE 的请求给与路由表中与目标节点最接近的 k 个邻居节点. 然后每一个邻居节点重复以上操作, 直至没有更好的结果或者目标节点被发现.

④ FIND_VALUE 操作与 FIND_NODE 操作相似, 然而当一个节点包含所需的 key 时, 会返回所需资源而不是离它最接近的 k 个邻居节点, 当请求该 key 的节点获得了资源以后, 执行 STORE 操作把结果存入到最近的 k 个节点中, 以加快下一次执行 FIND_VALUE 的速度.

1.4 路由表查询

DHT 网络的核心问题是快速节点查找. Kademlia 的节点查询通过以下步骤实现:

① 计算节点之间的距离 $d(x,y)=x \oplus y$

② 从 x 的第$\lfloor \log_2 d \rfloor$个 K 桶中取出 α 个节点, 分别对其进行 FIND_NODE 操作, 如果这个 K 桶中的信息少于 α 个, 则从附近多个 K 桶中选择距离最接近 d 的总共 α 个节点.

③ 对于收到 FIND_NODE 请求的每个节点, 如果发现自己就是 y, 则回答自己是最接近 y 的; 否则测量

© 中国科学院软件研究所 http://www.c-s-a.org.cn

自己和 y 的距离, 并从自己对应的 K 桶中选择 α 个节点的信息给 x.

④ x 对新接受到的每个节点都再次执行 FIND_NODE 操作, 此过程不断重复执行, 直到每一个分支都有节点响应自己是最接近 y 的.

⑤ 通过上述查找操作, x 得到了 k 个最接近 y 的节点信息.

这里的α参数是用来控制路由查询的速度的, 当α比较大时, 会加快查找速度但同时会增加节点间的通信量. 这个过程可以用图 1 描述.

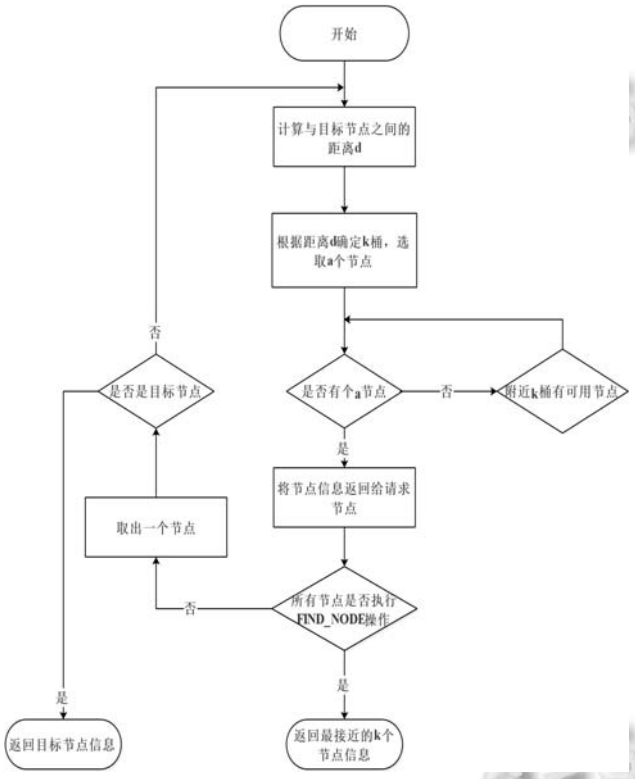

图 1　节点发现流程图

2 分布式爬虫策略

2.1 系统结构

本文设计的基于改进的Kademlia协议的分布式爬虫的结构(单节点)如图 2 所示.

其中, 爬虫模块负责以广度优先的方式爬取互联网信息, 将获取的网页解析并将得到的目标链接交给协议模块处理, 协议模块根据定义好的处理方式分发链接(执行 RPC 操作), 然后爬虫模块继续从任务队列中获取下一个任务, 以同样的方式处理. 节点间的协作完全通过协议模块来控制, 以此实现完全分布式.

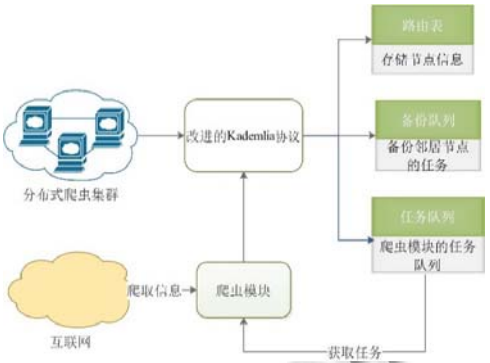

图 2　爬虫节点的系统结构

2.2 协议改进

结合分布式爬虫自身的特点和特定的需求, 本文在 Kademlia 算法原有的基础上提出了以下改进措施:

2.2.1 增加新的存储模块

爬虫模块是以 URL 为单位执行的, 对整个系统而言, URL 就是资源, 此时不能将其存储到原有的<Key, Value>对. 这里将原有协议中的存储分为 4 个部分:

① 原有存储: 用来保存<Key, Value>对信息.

② 任务队列: 用来存储需要执行的 URL 的队列.

③ 备份队列: 用来存储需要备份的邻居节点的 URL 的队列.

④ 处理记录: 用来存储已完成的 URL 的哈希表.

2.2.2 增加新的 RPC 操作

定义了四种新的 RPC 操作: STORE_TASK、STORE_BACKUP、DELETE_BACKUP、REFRESH_TASK. 通过新的 RPC 操作来保证分布式中各个节点的协作.

① STORE_TASK 操作允许一个节点将它发现的资源(URL)进行分发, 根据任务划分策略执行 RPC 操作, 将资源存储到划分的节点让其处理.

② STORE_BACKUP 操作允许一个节点间它发现的资源进行备份, 根据容灾策略执行 RPC 操作, 将资源存储到距离它最为接近的多个邻居节点中.

③DELETE_BACKUP 操作允许一个节点在执行完任务后, 执行 RPC 操作, 将它的邻居节点备份的信息删除掉.

④ REFRESH_TASK 操作允许一个节点通知它的邻居节点重新分发其任务队列中的所有任务.

2.2.3 增加网络结构变化的处理

对于 DHT 网络节点的增加、退出、异常等情况增

© 中国科学院软件研究所 http://www.c-s-a.org.cn

加了新的处理方式, 保证 DHT 网络的可扩展性和容错性.

2.3 任务划分与处理策略

对于整个 DHT 网络而言, 需要将任务进行划分到每一个具体的节点, 同时保证该任务的唯一性(相同的任务每次都会被分配到同一个节点)和整个系统的均衡性(每个节点分配的任务量大致相当)[10]. 这里采用的策略是: 采用与节点资源存储相同的哈希函数, 将 URL 哈希成为与节点 ID 位数一致的 key,根据之前提到的距离定义获得与节点最接近的节点, 将该 URL 存放到所选节点的任务队列中(使用 RPC 操作 STORE_TASK).

因为使用策略的唯一性可以保证相同任务每次都能被发送到同一节点, 所以可以根据已保存的处理 URL 记录进行去重, 如果不对 URL 进行检查, 会导致大量冗余任务产生, 从而降低系统效率.

整个任务的处理流程如下所示:

(1) 本地节点选择需要的分发的 URL, 计算它的哈希值作为 key .

(2) 本地节点执行 RPC 操作 FIND_NODE 寻找与 URL 的 key 距离最接近的节点作为目标节点.

(3) 对目标节点执行 RPC 操作 STORE_TASK .

(4) 对目标节点的邻居节点执行 RPC 操作 STORE_BACKUP, 将 URL 存到邻居节点的备份队列.

(5) 目标节点先在处理记录中进行 URL 去重检查, 如果不重复则将 URL 存到任务队列.

(6) 目标节点的爬虫模块从任务队列中获取任务并执行.

(7) 目标节点执行完成后, 计算 key 与节点 ID 的距离的哈希值, 将任务存放到处理记录对应的哈希表项中.

(8) 目标节点对邻居节点执行 RPC 操作 DELETE _BACKUP, 删除邻居节点备份队列中的任务.

2.4 扩展与容灾策略

2.4.1 节点加入

当一个新的节点需要加入到现有的 DHT 网络中时, 需要知道至少一个节点的信息, 其主要过程为:

(1) 新节点将已有节点插入到合适的 K 桶中, 建立路由表.

(2) 新节点执行 RPC 操作 FIND_NODE, 请求的节点为自身 ID,从而更新 DHT 网络其他节点的路由表信息.

(3) 新节点根据其他节点的返回信息, 更新自身的路由表信息.

(4) 新节点执行 RPC 操作 REFRESH_TASK,向邻居节点请求任务.

(5) 邻居节点对其任务列表中的所有 URL 执行 RPC 操作 STORE_TASK, 将距离新节点较近的任务分发给新节点.

2.4.2 节点正常退出

在 Kademlia 协议中, 节点退出时是不需要发布任何信息的, 只需要每个节点周期性地发布所有的<Key, Value>对信息, 然而改进的 Kademlia 协议中需要对退出节点的备份队列和任务队列进行处理, 主要流程:

(1) 退出节点对其任务队列中所有的任务, 进行重新发布, 找出每个任务最接近的节点, 并执行 RPC 操作 STORE_TASK

(2) 退出节点对其备份队列中所有的任务, 进行重新发布, 找出每个任务最接近的几个节点, 对最接近的节点执行 RPC 操作 STORE_TASK,而对其他节点执行 RPC 操作 STORE_BACKUP

(3) 节点退出

2.4.3 节点异常退出

每个节点周期性地发布其备份队列中的所有的信息并执行 RPC 操作 STORE_TASK, 这样当节点发生异常退出时, 可以保证该任务不会丢失掉.

3 实验与仿真

为了比较爬虫的扩展性, 本文设计了两组试验: 单机试验和集群实验. 其中单机实验使用多线程的方法进行扩展, 而集群实验通过增加节点来进行扩展. 爬虫模块两者相同, 都是根据广度优先的方法进行爬取, 起点设为 http:.www.baidu.com . 主要流程如图 3 所示.

3.1 单机实验

单机实验所使用的环境为 Ubuntu 14.04 LTS 操作系统, 使用 Intel Core i5 处理器. 实验中通过改变系统所用线程数来对爬虫模块的单机性能从以下两个方面进行评估:

© 中国科学院软件研究所 http://www.c-s-a.org.cn

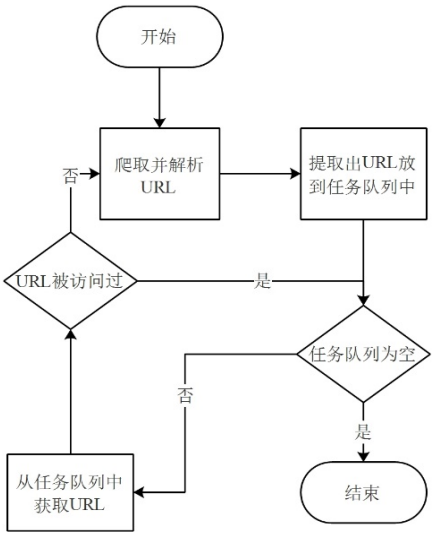

图 3　爬虫模块的流程

3.1.1 系统性能

其中处理速度代表每秒中能够爬取并解析的 URL 数, 生成速度代表每秒钟解析生成的 URL 数. 通过设置爬虫模块的线程数统计系统的处理速度和生成速度, 如图 4 所示, 随着线程数的增加, 系统的处理速度和生成速度有显著的提升, 然而提升的速度会受到实验机器性能的限制而降低最终饱和.

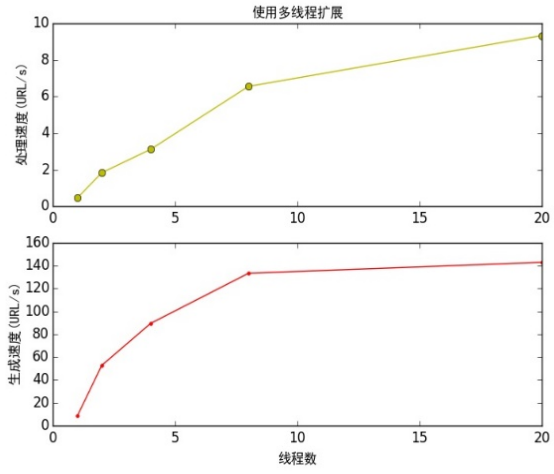

图 4　多线程的系统性能

3.1.2 系统负载

网络带宽占用和 CPU 占用是爬虫运行过程中的平均负载. 由图 5 可知, 随着线程数的增加, 系统的网络负载和 CPU 负载会逐渐增加, 很容易达到饱和.

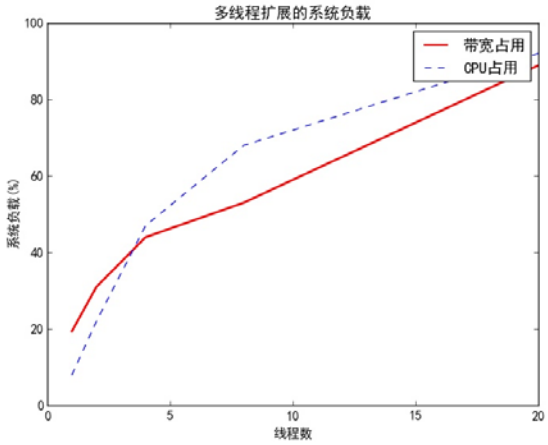

图 5　多线程的系统负载

3.2 集群实验

集群实验是通过云主机来进行测试的, 集群中每一个节点所使用的环境均为 Ubuntu 14.04 LTS 操作系统, 使用单核处理器. 实验中通过增加集群中的节点来对整个分布式集群实验. 节点 IP 分布如表 2 如示.

表 2　分布式集群节点分布

序号	地理位置	节点 IP
1	纽约	104.236.31.1xx
2	纽约	45.55.243.1xx
3	阿姆斯特丹	188.166.124.3x
4	新加坡	167.170.200.2xx
5	伦敦	178.62.64.x
6	法兰克福	46.101.191.5x

设置爬虫模块每 5 s 调用一次, 每隔 1 min 增加一个节点, 将采样的周期定为 20 s, 采样时间 8 min,记录下 URL 处理速度和生成速度如图 6 所示.

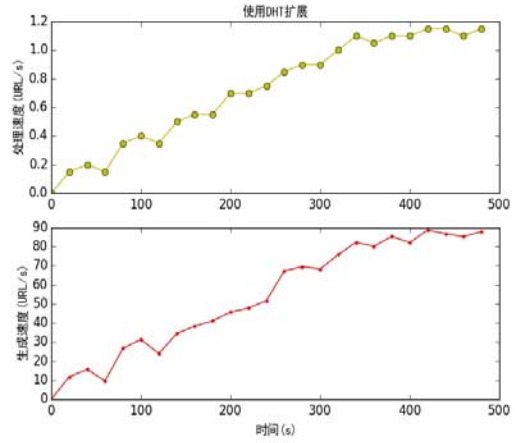

图 6　分布式集群采样图

© 中国科学院软件研究所 http://www.c-s-a.org.cn

3.2.1 系统性能

将采样的结果按照不同的集群规模进行分类并计算在不同规模下集群的平均处理速度. 通过最小二乘法进行一元线性拟合, 所得结果如图 7 所示.

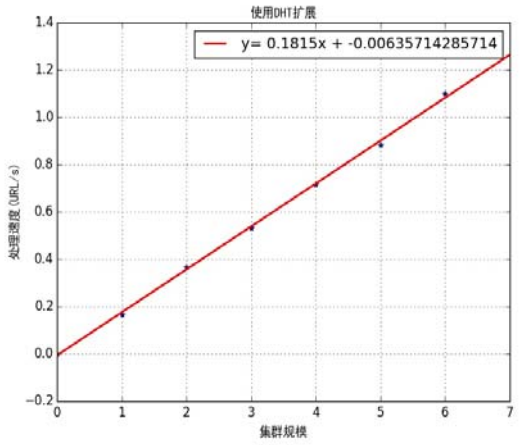

图 7 分布式集群的系统性能

随着集群规模的增大, 整个集群的处理能力增强, 且相对稳定, 处理速度大致呈线性增长. 当需要提升集群的处理能力时, 仅需要增加额外的节点即可, 具有较好的拓展性.

3.2.2 系统负载

由于云主机的带宽较大, 系统使用的带宽非常少, 因此未做测量. 统计每个节点在不同规模下的 CPU 负载如图 8 所示.

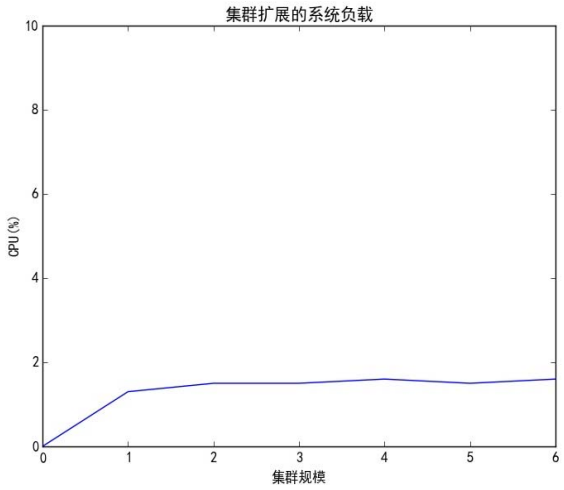

图 8 分布式集群的系统负载

由图 8 可知, 随着集群规模的增大, 每个节点的平均 CPU 占用基本不发生改变. 只有在节点间通信和爬虫模块会消耗一些 CPU 资源, 这意味着系统的拓展性良好, 节点负载基本不会受集群规模扩大的影响.

4 结语

本文提出了一种基于改进的 Kademlia 协议的完全分布式的网络爬虫, 同时阐述了针对 Kademlia 协议的改进和整个系统的设计与结构. 通过实际实验, 有效地验证了这种架构的可行性. 对比单机多线程方式, 能够避免单个节点资源耗尽的问题, 具有良好的扩展性和容错性. 同时本文的分布式爬虫能够部署到广域网范围, 不受局域网的制约.

参考文献

1 许笑.分布式 Web 信息采集关键技术研究[博士学位论文]. 哈尔滨:哈尔滨工业大学,2011.

2 许笑,张伟哲,张宏利,方滨兴. 广域网分布式 Web 爬虫.软件学报,2010,21(5): 1067–1082.

3 黄志敏,曾学文,陈君.一种基于 Kademlia 的全分布式爬虫集群方法.计算机科学,2014,41(3):124–128.

4 Boldi, Paolo, et al. Ubicrawler: A scalable fully distributed web crawler. Software: Practice and Experience, 2004, 34(8): 711–726.

5 金凡,顾进广.一种改进的 T-Spider 分布式爬虫.电子学与计算机,2011,28(8):102–104.

6 周模,张建宇,代亚非.可扩展的 DHT 网络爬虫设计和优化. 中国科学(信息科学),2010,40(9):1211–1222.

7 Singh A, et al. Apoidea: A decentralized peer-to-peer architecture for crawling the world wide web. Distributed Multimedia Information Retrieval. Springer Berlin Heidelberg. 2004. 126–142.

8 Stoica I, et al. Chord: A scalable peer-to-peer lookup service for internet applications. ACM SIGCOMM Computer Communication Review, 2001, 31(4): 149–160.

9 Maymounkov P, Mazieres D. Kademlia: A peer-to-peer information system based on the xor metric. Peer-to-Peer Systems. Springer Berlin Heidelberg. 2002. 53–65.

10 Li GL, Zhang HB. Design of a distributed spiders system based on Web service. Second Pacific-Asia Conference on Web Mining and Web-based Application, 2009. WMWA'09. IEEE, 2009.

© 中国科学院软件研究所 http://www.c-s-a.org.cn