云计算环境下分布式文件系统的负载平衡研究

尹向东 杨 杰 屈长青
(湖南科技学院计算机与通信工程系 永州 425100)

摘 要 在云计算环境下,文件以块的形式分布在文件系统中。然而系统状态的更新,如节点加入和离开,会导致文件块在系统中的分布不均衡,从而对系统执行的任务产生性能上的影响。针对该分布式文件系统的文件分布不均衡问题,提出了一种完全分布式的负载平衡算法,并采用了仿真实验对所提出的算法与集中式和分布式的负载均衡算法进行了对比。结果表明,该算法在解决了集中式算法的单点瓶颈的基础上只增加了少量的额外开销,其性能明显优于分布式的负载均衡算法。

关键词 负载均衡,分布式文件系统,云计算,算法
中图法分类号 TP312 **文献标识码** A

Research on Load Balancing of Distributed File System in Cloud Computing

YIN Xiang-dong YANG Jie QU Chang-qing
(Department of Computer and Communication Engineering, Hunan University of Science and Engineering, Yongzhou 425100, China)

Abstract In cloud computing, the files are divided into chucks, and stored in distributed file system. However, the updates of the system states, such as node joining and leaving, will cause an unbalancing distribution of file chunks in the distributed file system, thereby degrading the system performance a lot. For solving this problem, this paper proposed a distributed load balancing algorithm, and compared it with the centralized and distributed load balancing algorithms. The experiments show that the proposed algorithm increases only a little overhead while solving the single point bottleneck of the centralized algorithm, and it has obviously better performance than the distributed load balancing algorithm.

Keywords Load balancing, Distributed file system, Cloud computing, Algorithm

云计算技术在最近几年得到了广泛的发展,不管是在工业界还是在学术界都对其进行了大量的研究。在云中,云供应商对系统进行部署;而客户端只需要根据其应用的需求对服务端提出需求,从而服务商根据其要求对资源进行分配,所以客户端不需要进行繁杂的系统部署和资源管理的工作。在最近的研究成果中,能够保证云得到广泛应用的主要技术有Google提出的MapReduce编程模型[1]、GFS分布式文件系统[2]、虚拟化技术[3,4]等。这些技术的一个重要的特点是都具有很强的扩展性,只有这样云才能够具有很强的伸缩性,能够在系统中节点出错以及扩展节点等情况下还为用户提供完好的服务。在云计算环境下实现并行计算的最主要方法就是使用MapReduce技术,MapReduce技术的实现主要是依赖于底层的分布式文件系统,其效率的高低主要由分布式文件系统来决定。而在云环境下,文件可以被任意地创建、删除和增加;而且节点随时有可能出现错误、替换以及增加的情况。在这样变化的环境下,对文件系统影响最大的是节点的负载可能会出现不均衡的状况。如果节点的负载出现不均的情况,将会对系统的性能产生重大的影响,因为当系统负载不均时可能会导致大部分的操作在某些节点上处理而降低了处理的并行性,而且此时会出现一些节点处于空闲状态,从而使得大量的资源浪费。因此,文件系统是否能适应这样的变化对于系统的性能来说具有重要的影响。

现有的云环境下的分布式文件系统都是使用一个中心节点来管理文件系统的元数据,而该中心节点将根据系统的这些元数据信息来对系统的存储以及处理的负载进行调整。这种中心式的方法能够简化分布式文件系统的实现。但是在实际的实践中,随着存储节点、文件数量以及访问请求的增加,该中心节点就会成为整个文件系统的性能瓶颈。该中心节点无法对大量的用户访问请求以及MapReduce程序的请求进行及时的响应。因此,这种中心式方法对于文件系统的负载均衡处理并没有特别好的效果,而且这种架构很有可能会出现单点失效的情况。如果该中心节点失效,则整个文件系统将无法为用户提供服务。

为了解决负载不均衡的问题,本文在研究中提出了将负载的均衡任务分配到存储节点上,让这些存储节点自发地来平衡负载。这样就可以消除依赖于中心节点的情况,同时也可以消除单点失效的问题。在本文的研究中,存储节点的网络架构是基于分布式hash表(DHTs)来构建的[5],每个文件块都有一个唯一的标识。在云环境中,节点出错以及增加节点是正常现象,而DHT在这些情况下能够进行自我管理和

到稿日期:2013-04-05 返修日期:2013-08-23 本文受湖南省自然科学基金(11JJ6065),湖南省教育厅科研项目(12C0681,10C0732)资助。
尹向东(1976-),男,硕士,副教授,主要研究领域为计算机网络、算法,E-mail:yinxiangdongxz@163.com;杨 杰(1976-),男,硕士,副教授,主要研究领域为计算机网络、人工智能、数据挖掘;屈长青(1963-),男,硕士,副教授,主要研究领域为计算机网络安全、网络检测。

修复，从而可以降低系统对节点的管理难度。

1 相关工作

本节将详细地介绍对于负载均衡的一些相关研究，并对它们进行分析。

在以前的学习中有中心式的负载均衡算法[6]以及分散的负载均衡算法[7]，但是这些算法的研究主要都是在一般的环境下进行的。这些算法的主要目标是为了能够平衡节点的负载来完成计算任务。它们并没有考虑在计算密集型应用中负载迁移的代价；而且这些方法在存储节点上完成计算的大规模数据密集型计算系统（例如 Google 的 GFS[2]、Hadoop[8]的 HDFS）中的作用是很小的；并且针对负载均衡进行的元素迁移实体是文件块，例如在 HDFS 中一个文件被划分成多个不相交的块（典型的块大小为 64MB）。文献[6,7]的工作中假设系统具有一个稳定的计算环境，计算系统中不会出现实体随意的增加或者减少。而与文献[6,7]的假设不同的是，本文主要是面向大规模、动态的数据密集型计算环境，从而提出一个负载重均衡算法来应对分布式文件系统中文件创建、删除、扩展以及节点的增加、删除等情况下系统中各个节点的负载变化。

分布式文件系统的负载均衡算法在文献[9,10]中都有一些相关的介绍。这些方法主要是依靠于中心节点来平衡磁盘的负载。中心服务器对分布在系统中的每个文件块进行全局的统计。基于这些全局的知识，一些文件块将会被进行重分配，从而保证系统中没有一个节点是超负荷的。在以前的一些负载均衡算法中都会出现性能瓶颈这个概念，性能瓶颈使正常的应用程序无法在一个漫长的负载重均衡时期对它们的文件请求进行操作。相反，本文将提出一个完全的分布负载重均衡算法，该算法具有快速的收敛性，并且不会出现性能瓶颈的问题。

很多分布式文件系统（例如 Google 的 GFS[2]、Hadoop 的 HDFS[8]、Lustre[11]、Panasas[12]以及 PVFS[13]等）在一些文献中已经被提出，并且在实际的系统中已经得到了广泛的使用。在最近的一些系统中（例如 IBM 的 GPFS[5]、G-HBA[8]以及 PVFS[13]）都结合了文件分块技术，它们认为对于文件的元数据（例如目录树）也应该进行分布式的管理，从而可以提高系统的扩展能力。虽然有这样的技术，但是新兴的分布式文件系统（Google 的 GFS[2]、Hadoop 的 HDFS[8]）仍然是采用单独的中心节点来调整存储节点的负载。例如，在 GFS 中的 master 节点（或者 HDFS 中的 namenode）主要是用来维护文件块被分配到 chunkservers（HDFS 中的 datanodes）的位置。文件块将会被尽量均匀地分配到 chunkservers（datanodes）中，因此 MapReduce 可以通过大量并行任务的形式来完成任务。使用一个单独的服务器从 master（namenode）节点中获取文件位置的全局信息，如果有 chunkservers（datanodes）出现存储量超过了限制的 blocks，就会根据全局信息对一些文件块进行重分配。

2 负载平衡问题描述

假设在云中的大规模分布式文件系统由一个 chunkservers V 集合组成，其中 V 的基数为 n。在一般的情况下，n 可能是一千、一万，甚至更多。系统中将会有大量的文件被分布地存储在这 n 个 chunkservers 上。首先，假设文件集合用 F 来表示。对于任意的文件 $f\in F$ 将被划分成大量的不相交的文件块，并且文件块的大小固定地用 C_f 来表示。例如，在 Hadoop 中的 HDFS 的每个文件块的大小是一样的，而且都是 64MB。其次，假设 chunkserver 的负载将会与该服务器所存储的 chunks 的数量成比例。第三，在这样的分布式环境下节点出错的情况是正常的；而且 chunkservers 随时有可能升级、替换或者增加新的节点。另外，在 F 中的文件会被任意地创建、删除和增加。由于网络的影响，可能会导致文件块不能被均匀地分到 chunkservers 上。图 1 就是一个负载均衡问题的例子，首先把 $f_i(1\leqslant i\leqslant 6)$ 初始分配到 N_1，N_2 和 N_3 3 个节点上，见图 1(a)；在图 1(b)中，将文件 f_2 和 f_5 删除；在图 1(c)中，将文件 f_6 加入到系统中；在图 1(d)中，新加入节点 N_4。上述图中，图 1(a)系统处于均衡状态，图 1(b)，(c)和(d)都处于不均衡状态。

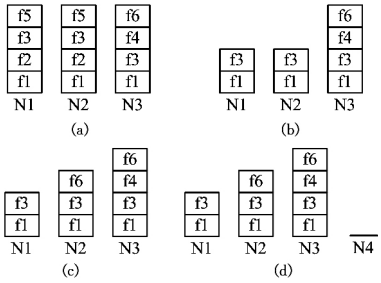

图 1 系统不均衡实例

本文研究的目标是要设计一个负载调整算法来对一些文件块进行重新的分配。从而尽量将文件块均匀地分布到系统中，并且也要降低文件移动的代价（为了保证系统中 chunkservers 负载均衡而需要进行调整的文件块的数量）。因此，负载调整算法的目标是为了最小化每个 chunkserver $i\in V$ 负载不均衡因子：

$$\| L_i - A \| \tag{1}$$

其中，L_i 代表 chunkserver i 的负载（节点 i 所存储的文件块数量），而 $\| \cdot \|$ 代表绝对值函数。而 A 是节点 i 在一个系统范围以及负载均衡状态下理想的文件块的数量，其可以表示为：

$$A=\frac{\sum_{f\in F}|C_f|}{n} \tag{2}$$

负载平衡问题其实是一个典型的 NP 难问题。为了简化对该问题的讨论，在本文的研究中将假设系统的节点是同构的，则在任意两个节点对文件块进行迁移的代价是一样的，而且每个 chunkserver 都将拥有相同的存储容量。在今后的研究中我们将考虑节点异构的情况。

3 完全的负载重均衡策略—设计思想及算法

本节将详细地介绍我们提出的完全负载重均衡方法，首先将介绍适应该策略的分布式文件系统的详细架构，然后再对算法的思想进行详细的介绍。

3.1 系统架构

本文提出的策略使用的文件系统的 chunkservers 是用一个 DHT 网络来组织的；每个 chunkserver 都会实现一个类似于 Chord[14]或者 Pastry[15]的 DHT 协议。一个文件将会被划分成大量具有固定大小的文件块，且每个块都将拥有一个块柄（chunk handle），并用一个全局的 hash 函数来命名。该 hash 函数将根据给定的文件路径串和块索引来返回一个唯一的标识符。每个 chunkserver 也都将拥有一个唯一的 ID。在 V 中，为了描述方便就用 $1,2,\cdots,n$ 来表示 n 个chunkservers。

我们将对chunkserver i 的后继 chunkserver 表示为 $i+1$，而 chunkserver n 的后继 chunkserver 是 1。在一个 DHT 中，一个 chunkserver i 能够存储的文件块的句柄范围在$(\frac{i-1}{n},\frac{i}{n}]$之间，但是对于 chunkserver n 来说其存储的文件块的句柄范围在$(\frac{n}{n},\frac{1}{n}]$之间。

为了能够找到文件块，DHT 需要完成查找的操作。在大部分的 DHTs 中，如果每个 chunkserver i 拥有的邻居数为 $\log_2 n$，在一次的查找中需要访问的节点的平均数量为 $O(\log n)$[14,15]。在这 $\log_2 n$ 个邻居中，$i+2^0$ 是 i 的后继节点。如果要查找一个拥有 n 个块的文件，则需要进行 $O(\log n)$ 次的查找。在本文的算法中使用 DHTs 架构主要有以下几方面的原因：

(1)由于 chunkservers 具有随意到达、离开以及出错的情况，因此在我们的方法中 chunkservers 具有自我配置以及自我修复的能力，从而简化系统的供应和管理。在典型的 DHTs 中都能保证当有一个节点离开时，则该节点存储的文件块将迁移到其后继的节点上存储。如果新加入一个节点，那些文件块的 ID 从其后继节点到该节点的值之间的数据块由该节点来存储。本文的研究主要侧重于当有节点离开或者加入时的文件块的迁移。在接下来的讨论中，假设如果一个节点 i 离开之后又将其加入到其他节点 j 的后继节点，则节点 i 将用 $j+1$ 来表示，而节点 j 原来的后继节点将表示为 $j+2$。节点 j 的原始邻居表示为 $j+3$ 等。

(2)访问 $O(\log n)$ 个节点的查找有一个普通的延迟，因为这个文件块的查找可以被并行完成，所以该查找延迟就可以忽略。另外，我们的方法是独立于 DHT 协议的，主要将会采用类似于 Amazon's 的 Dynamo 的方法，其可以只进行一跳的查找延迟。

(3)在完全负载均衡策略中 DHT 网络对于元数据管理是透明的。在我们的方法中当 DHT 网络指定一个特定位置的块时，其能够快速地对文件块进行查找。我们的方法可以与现有的大规模分布式文件系统进行集成。例如，将该方法与 GFS 中的 master 节点进行集成，每个 chunkserver 周期地将其本地的文件块的信息以心跳的形式返回给 master 节点，因此 master 节点可以知道块在系统中的详细位置。

(4) DHT 网络指定了系统中文件块的位置。如果节点和文件块都指定了一个唯一的 ID，就可以保证一个节点的平均最大负载概率与文献[16]所提出的 DHT 网络下的节点文件下载时的负载均衡概率值变化一样，然后对节点的负载平衡到一个固定的程度。但是本文的方法在无唯一的 ID 分布的情况下，节点和文件块也能良好运行。

根据以上的描述，我们知道负载重均衡问题是一个 NP 难的问题，因此在一些技术研究上有很大的挑战。而对于那些元数据管理、文件一致性模型以及副本策略在本文上将不进行相关的讨论，本文将重点研究如何保证系统中节点的负载平衡。

3.2 算法设计

如果一个分布式文件系统中每个节点都存储 A 个块，则该系统处于一种负载均衡的状态。因此，对于一个大规模的分布式文件系统，每个 chunkserver 节点 i 都将首先分析该节点的负载状态，即是负载轻还是负载较重。如果一个节点负载较轻，则该节点所存储的块数量将会比平均阈值 $(1-\Delta_L)A$ 小，其中 Δ_L 是一个系统参数，并且 $0\leqslant\Delta_L\leqslant 1$。相反，一个超负荷节点所管理的文件块数量将大于 $(1+\Delta_L)A$。

本文的算法过程如下，假设任意的节点 $i\in V$，如果节点 i 负载较轻，则从高负载节点中迁移超过 A 个文件块的节点。如果节点 i 是系统中负载最轻的节点，而且 i 即将离开系统，则需要将该节点的文件块迁移到其后继节点 $i+1$ 上。该分配过程如下：

(1)低移动代价：因为在所有的 chunkservers 中 i 是一个负载最轻的节点，所以节点 i 离开系统所需要迁移的块数量是很小的，因此其移动的代价将会很小。

(2)快速收敛比例：i 试图获得负载较少的节点中最大节点 j 的负载，希望系统能够快速地收敛到一个负载均衡的状态。如果负载最重的节点 j 的负载为 $2A$，则节点 i 从节点 j 上获得 A 个文件块；否则就从节点 j 上获得的负载量为 L_j-A。如果 $L_i=A$，则节点 i 的负载会处于平衡状态，而且节点 j 的负载会被立刻降低。

在一些情况下，虽然将 j 中部分的负载迁移到 i 上，但是 j 仍然有可能是负载最重的节点。如果是这样，就使负载轻的节点中负载最轻的节点 k 离开系统，并将该节点重新加入到节点 j 的后继节点上。因此节点 k 将变成节点 $j+1$，而 j 的原始邻居 i 将会变成节点 $j+2$。这样的过程一直迭代处理，直到节点 j 不是负载最重的节点为止。接着，在其他的负载较重的节点上，负载最重的节点将会把负载最轻的节点加入到其后继节点上来减轻该节点的负载。

为了匹配负载最轻和最重的节点，我们通过 $top-k_1$ 个低负载节点和 $top-k_2$ 个高负载节点来形式化我们的想法。不失一般性，用 U 来表示 $top-k_1$ 个低负载节点的集合，用 O 来表示 $top-k_2$ 个高负载节点的集合。然后，我们试图去发现并匹配 U 中第 k_1' 个低负载节点和 O 中第 k_2' 个高负载节点：

$$k_1'=\left\lceil\frac{\sum_{L_i\in O}^{k_2'}(L_i-A)}{A}\right\rceil \tag{3}$$

其中，$k_1'\leqslant k_1$ 而且 $k_2'\leqslant k_2$。在式(3)中 $\sum_{L_i\in O}^{k_2'}(L_i-A)$ 表示在 $top-k_2$ 个高负载节点中超出的负载量。而使有 $top-k_1$ 个轻负载的节点离开系统并重新加入到这 $top-k_2$ 个高负载节点的后继节点上。通过式(3)，我们试图尽量减少移动的代价来快速地平衡它们的负载。算法的细节如算法 1 和算法 2 所示。

算法 1 Seek(ν,Δ_L,Δ_U) //轻负载节点 i 寻找重负载节点 j

输入：包含多个节点的采样向量 ν，Δ_L 和 Δ_U

输出：重负载节点 j

1. $\widetilde{A}_i\leftarrow$利用采样向量 ν 对节点 i 的 A 进行估计；
2. If $L_i<(1-\Delta_L)\widetilde{A}_i$ then
3. $\quad\nu=\nu\cup\{i\}$；
4. $\quad$对 ν 进行排序，使得 $L_i<L_j(i<j)$；
5. $\quad k\leftarrow i$ 在 ν 中的位置；
6. $\quad$寻找满足下列条件的最小子集 $P\subset\nu$

$\quad(1)L_j>(1+\Delta_U)\widetilde{A}_j,\forall j\in P$，

$\quad(2)\sum_{j\in P}(L_j-\widetilde{A}_j)\geqslant k\widetilde{A}_i$；

7. $\quad j\leftarrow P$ 中负载最轻的节点；
8. Return j；

算法 2 Migrate(i,j) //重负载节点 j 迁移数据到轻负载 i

输入：重负载节点 j，轻负载节点 i

1. If $L_j>(1+\Delta_U)\widetilde{A}_j$ 并且 j 可以迁移数据 then

2. i 将局部块迁移到 i+1；
3. i 离开系统；
4. i 作为 j 的后继节点加入系统，i←j+1；
5. $t \leftarrow \widetilde{A}_i$；
6. If $t > (L_j - (1+\Delta_U)\widetilde{A}_i)$ then
7. $t \leftarrow (L_j - (1+\Delta_U)\widetilde{A}_i)$
8. i 分配 t 个连续空间；
9. 将 j 的 t 个超出负载迁移到 i；

4 仿真实验

在本文中，我们对算法进行了仿真实验。本文应用 java 的多线程技术，基于 Chord 的 DHT[14] 和 gossip 协议[12,13] 实现了算法的模拟器。算法模拟了 1000 个节点，10000 个文件块，这些文件块在初始的时候以几何分布的形式分布在各个节点上。由于文件块的分布符合几何分布，因此在系统初始化的时候，有少量节点分布着大量的文件块。

我们将本文提出的算法(Ours)与分布式的负载均衡算法(Distributed)和集中式的负载均衡算法(Centralized)进行了对比。图 2 为 3 种算法在应用了相应的负载均衡算法后的累积分布函数(cumulative distribution function，CDF)。在本实验中，假设节点是同构的，即存储能力相同，每个节点的理想块分布为 10。由图中可以看出，在收集到了系统的全局信息后，集中式的负载均衡算法具有很好的性能；分布式的负载均衡算法的性能最差；本文提出的算法不需要收集系统的全局信息，并且性能接近集中式的负载均衡算法。

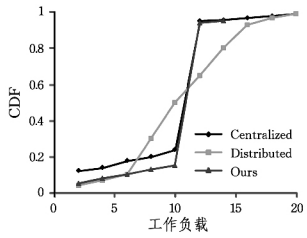

图 2 工作负载分布图

我们对比了本文提出的算法与其它两种算法的迁移开销，分别观察了 3 种算法在 4 个工作任务(A，B，C，D)下的迁移开销，结果见图 3。在实验结果汇中，对实验结果即 3 种算法的迁移开销与集中式算法的比值进行了规范化。从图中可以发现本文的算法与集中式的负载均衡算法的迁移开销相当，并且明显低于分布式的负载均衡算法。

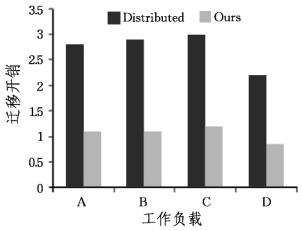

图 3 迁移开销对比图

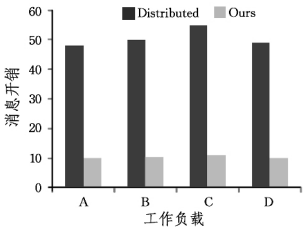

图 4 通信消息开销对比图

此外，我们对比了 3 种算法在执行过程中的通信消息开销，实验结果见图 4。该实验仍然采用 4 个工作任务，实验结果仍然采用对集中式算法的消息开销的规范化结果。从实验结果可以看出，本文提出的算法虽然在消息开销上大于集中式的负载均衡算法，但是远远小于分布式的负载均衡算法。

从上述实验结果可以看出，本文提出的负载均衡算法在文件块的迁移开销和系统的通信消息开销上都明显优于分布式的负载均衡算法。此外，本文提出的算法虽然在通信消息开销上高于集中式的负载均衡算法，但是它可以实现系统的可靠运行，消除了集中式算法的控制中心的瓶颈和可靠性问题。

结束语 分布式文件系统的负载均衡问题是分布式系统的重要研究内容。在云计算环境下，文件以块的形式存放于分布式系统中。然而，随着系统的持续运行，必将导致文件块分布不均衡，于是负载过重的节点就成为了分布式计算任务的瓶颈。为了维持系统内文件块的持续均衡分布，本文提出了一种完全分布式的负载平衡算法，并采用了仿真实验对本文提出的算法与集中式和分布式的负载均衡算法进行了对比。结果表明，该算法在解决了集中式算法的单点瓶颈的基础上只增加了少量的额外开销，其性能明显优于分布式的负载均衡算法。

参 考 文 献

[1] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113

[2] Ghemawat S, Gobioff H, Leung S T. The Google file system[C]// ACM SIGOPS Operating Systems Review. ACM, 2003, 37(5): 29-43

[3] VMware[OL]. http://www.vmware.com/, 2013

[4] Xen[OL]. http://www.xen.org/, 2013

[5] Hsiao H, Chung H, Shen H, et al. Load Rebalancing for Distributed File Systems in Clouds[J]. Appplied Soft Computing, 2012, 21: 102-105

[6] Vu Q H, Ooi B C, Rinard M, et al. Histogram-based global load balancing in structured peer-to-peer systems[J]. IEEE Transactions on Knowledge and Data Engineering, 2009, 21(4): 595-608

[7] Hsiao H C, Liao H, Chen S T, et al. Load Balance with Imperfect Information in Structured Peer-to-Peer Systems[J]. IEEE Transactions on Parallel and Distributed Systems, 2011, 22(4): 634-649

[8] Jones P, Eastlake D E. US secure hash algorithm 1 (SHA1) [J]. Networking, IEEE/ACM Transactions on, 2001: 902-910

[9] Raab M, Steger A. "Balls into Bins"—A Simple and Tight Analysis[M]. Randomization and Approximation Techniques in Computer Science, Springer Berlin Heidelberg, 1998: 159-170

[10] Jelasity M, Montresor A, Babaoglu O. Gossip-based aggregation in large dynamic networks[J]. ACM Transactions on Computer Systems (TOCS), 2005, 23(3): 219-252

[11] Jelasity M, Voulgaris S, Guerraoui R, et al. Gossip-based peer sampling [J]. ACM Transactions on Computer Systems (TOCS), 2007, 25(3): 8

[12] Stoica I, Morris R, Liben-Nowell D, et al. Chord: a scalable peer-to-peer lookup protocol for internet applications [J]. IEEE/ACM Transactions on Networking, 2012, 11(1): 17-32

[13] Rowstron A, Druschel P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems[C]// Middleware 2001. Springer Berlin Heidelberg, 2011: 329-350

[14] Rao A, Lakshminarayanan K, Surana S, et al. Load balancing in structured P2P systems[M]. Peer-to-Peer Systems II, Springer Berlin Heidelberg, 2003: 68-79

[15] Jin X, Chan S-H G, Wong W-C, et al. A Distributed Protocol to Serve Dynamic Groups for Peer-to-Peer Streaming[J]. IEEE Transactions on Parallel and Distributed Systems, 2010, 21: 216-228

[16] Bryhni H. A comparison of load balancing techniques for scalable Web servers[J]. IEEE Network, 2000, 14(4): 58-64