软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(8):1952−1967 [doi: 10.13328/j.cnki.jos.005199]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种自适应文件系统元数据服务负载均衡策略*

佘楚玉[1,2,3], 温武少[1,2], 肖 扬[1], 刘育擘[1], 贾 殷[1]

[1](中山大学 数据科学与计算机学院,广东 广州 510006)

[2](中山大学-卡内基梅隆大学国际联合研究院,广东 顺德 528300)

[3](广东财经大学 数学与统计学院,广东 广州 510320)

通讯作者: 温武少, E-mail: wenwsh@mail.sysu.edu.cn

摘 要: 随着大数据时代的到来,全球信息存储量呈现爆发式的增长,传统的存储系统在存储性能、存储容量、数据可靠性和成本等方面存在诸多不足.近年来,以云计算平台为依托的存储技术得到了飞速发展,成为处理海量数据的重要工具.针对分布式文件系统元数据管理的问题,提出了一种自适应元数据服务负载均衡策略.该策略主要包括以下 3 点内容:介绍了一种实时的元数据服务器的性能评价模型;提出了一种基于服务器负载变化的检测周期自适应调整机制;提出了一种基于元数据服务器性能指标的自适应负载均衡算法.实验结果证明了该方法的可行性、有效性和稳定性.

关键词: 分布式文件系统;元数据管理;自适应;负载均衡;性能评价模型

中图法分类号: TP316

中文引用格式: 佘楚玉,温武少,肖扬,刘育擘,贾殷.一种自适应文件系统元数据服务负载均衡策略.软件学报,2017,28(8):1952−1967. http://www.jos.org.cn/1000-9825/5199.htm

英文引用格式: She CY, Wen WS, Xiao Y, Liu YB, Jia Y. Adaptive load balancing strategy for file-system metadata service. Ruan Jian Xue Bao/Journal of Software, 2017,28(8):1952−1967 (in Chinese). http://www.jos.org.cn/1000-9825/5199.htm

Adaptive Load Balancing Strategy for File-System Metadata Service

SHE Chu-Yu[1,2,3], WEN Wu-Shao[1,2], XIAO Yang[1], LIU Yu-Bo[1], JIA Yin[1]

[1](School of Data and Computer Science, Sun Yat-Sen University, Guangzhou 510006, China)

[2](SYSU-CMU Shunde International Joint Research Institute, Shunde 528300, China)

[3](School of Mathematics and Statistics, Guangdong University of Finance and Economics, Guangzhou 510320, China)

Abstract: With the advent of big data era, global storage is experiencing an explosive growth. Traditional storage systems have several drawbacks in storage performance, storage capacity, data security and device cost. In order to handle large amount of data, the storage technology for cloud computing platform has undergone rapid development in the recent years, and become an important tool to deal with big data. This paper analyzes the shortcomings of metadata management of certain distributed file system and proposes an adaptive metadata load balancing mechanism. First, a real-time server performance evaluation model is introduced. Next, a period adaptive mechanism based on change of server load is built. Finally, an adaptive load balancing algorithm based on server performance is proposed. Experimental results demonstrate the practicability, availability and stability of the new mechanism.

Key words: distributed file system; metadata management; adaptive; load balancing; performance evaluation model

随着云计算技术的逐步成熟,存储系统作为云计算中重要的组成部分,同样需要满足当下新的需求.在云计

* 基金项目: 广东省科技计划(2014B010114002, 2015B010108004)
Foundation item: Science and Technology Project of Guangdong Province (2014B010114002, 2015B010108004)
收稿时间: 2016-06-17; 修改时间: 2016-09-21, 2016-11-11; 采用时间: 2016-12-01; jos 在线出版时间: 2017-01-12
CNKI 网络优先出版: 2017-01-12 10:35:59, http://www.cnki.net/kcms/detail/11.2560.TP.20170112.1035.002.html

算环境下,存储系统通常要求支持 PB 级的数据存储,同时需要满足高可用性、高扩展性、高性能等多种需求.文件系统作为存储系统中重要的一种类型,被业界广泛地应用.不同于传统环境,在云计算环境下,对文件系统性能等方面要求更为苛刻.目前,随着云计算的发展,对存储 I/O、吞吐量、读写延迟都有比较高的要求,例如在云桌面[1]的应用场景中,特别是在大量云桌面同时启动时,需要存储系统达到很高的 IOPS 性能.

在文件系统中,其 IOPS 主要是由文件元数据处理的速度和后端文件存储的速度决定.在文件存储方面,当前一些文件系统提供了较好的性能,如 Ceph[2],GFS[3],HDFS[4]等.然而在元数据管理上,当前的文件系统的元数据管理存在以下两点不足.

- 首先,目前主流的分布式文件系统很多都是采用集中式的元数据服务模型,而不是采用一个元数据服务器集群来处理元数据操作,如 HDFS,GFS 等.但随着云计算的广泛应用,存储的服务器硬盘上的数据量急剧增加,元数据处理的请求也不断增加,单一的元数据服务器在处理能力和系统可用性方面都无法满足当下的需求.而采用元数据服务集群的文件系统,如 CephFS[5],虽然使得元数据服务能力可以线性扩容,但在元数据服务的负载均衡上的表现并不是很好,从而也影响到整个文件系统的性能.
- 其次,由于硬件的迭代,目前企业的服务器集群存在性能异构等现状,传统的负载均衡策略没有考虑到服务器间的性能的差异.目前,主流的分布式文件系统在执行元数据负载均衡时,只是将元数据访问的热点作为单一热点评价标准,并没有考虑到服务器存在异构的情况.实际上,服务器异构的情况是普遍存在的,主要体现在下面几点.
  (1) 服务器本身的性能存在差异,例如 CPU、内存、硬盘读写性能存在差异,这使得不同服务器应对负载时的承受能力存在差异.
  (2) 服务器在不同时期的性能有所不同,即使可以评估出服务器硬件上的性能指标,但是如果遇到像网络状况发生变化、系统中其他应用占据了 CPU 或内存资源等情况,都会使得同一台服务器在不同的时间点表现出不同的性能特性.
  (3) 服务器可能搭建在云平台上,即服务器本身可以是虚拟机.在云计算时代,很多服务器的功能都被迁移到了云端,不同的云平台对于虚拟机资源一般是使用弹性机制,所以要确定一台虚拟机的性能也比较困难.

根据上述总结的两点不足可知,当前制约文件系统效率的一个重要因素是文件系统元数据的处理效率不高.一些文件系统使用元数据服务集群来获得元数据服务的线性扩展特性,从而提高文件系统元数据处理的效率.但是当前,许多文件系统对元数据服务集群的负载均衡策略不是很灵活,而且没有考虑到服务器的性能存在差异,这会导致负载均衡的效果不佳.例如,集群中高性能服务器负载虽然超过了集群负载的阈值,但服务器本身其实没有超载,而文件系统依然会启用负载均衡机制将负载迁移到了集群中性能较弱的服务器上,这会导致一些不必要的资源开销,甚至反而影响这个文件系统元数据服务的效率.

本文提出了一种自适应的文件系统元数据服务负载均衡策略来提高文件系统在元数据服务集群的负载均衡的效率.本文的主要贡献包括以下几点.

(1) 介绍了一个灵活的服务器性能评估模型,充分考虑了集群中服务器的性能差异现状,为元数据服务集群的负载均衡提供了可靠的依据.
(2) 提出了一种自适应元数据服务负载均衡算法.该算法可以根据元数据服务集群的总体请求压力,动态修改负载均衡策略,使得集群的负载均衡更加高效.
(3) 在 Ceph 上使用本文提出的元数据负载均衡策略,同时与原生的 Ceph 进行实验对比,验证了该策略的可行性与有效性.

本文第 1 节介绍本文的相关工作.第 2 节介绍文件系统中的自适应元数据服务负载均衡策略,其中包括对服务器性能评估模型的介绍、自适应元数据服务负载均衡算法的介绍以及目录子树迁移过程的描述.第 3 节对该负载均衡策略进行实验验证,同时对实验结果进行对比分析.第 4 节对全文进行总结.

1 相关工作

目前,大多数文件系统的元数据服务设计有 3 种主要模式.

- 第 1 种是单元数据服务节点,如 GFS[3],HDFS[4],BWFS[6]和 pNFS[7]等.单节点模式使得元数据服务节点容易成为文件系统瓶颈,同时有单点故障的问题.
- 第 2 种是双元数据服务节点,如 Lustre[8,9]等.这种模式通常是一个元数据服务器处理请求,另一个服务器作为其备份.这种方法虽然解决了系统的单点故障问题,但是性能较差,扩展性也较弱.
- 第 3 种是使用元数据服务集群,如 CephFS[5]等.这种模式解决了以上两种模式的不足.

目前,文件系统元数据研究热点主要有以下几个方面:一是跨数据中心的元数据管理,如 CalvinFS[10]采用 OLLP[11]机制实现高效的元数据管理;二是针对 NVM(non volatile memory)广泛应用的环境下元数据管理的优化,如 BetrFS[12,13]和 TokuFS[14]利用 TokuDB 较高的插入性能以及较高的压缩比特性实现一个 VFS 到 Disk 的中间层来优化元数据管理;三是元数据服务的负载均衡问题.

目前,关于元数据服务的负载均衡研究主要集中在两个方面:一个是元数据分区,另一个是服务器热点评价.元数据分区研究的是如何将元数据按照一定的规则分配给元数据服务集群中的服务器,使得它能够实现系统整体性能的均衡以及更充分利用系统资源.目前,大多数分布式文件系统都采用系统控制流和数据流分离的策略,这样可以获得更好的系统扩展性和读写性能.元数据分区模型非常重要,因为它影响了整个系统的性能——稳定性、可靠性和扩展性.目前,主要有两种分区方案:目录子树分区法[15]和哈希分区法[16].

- 目录子树分区法的主要思想是:将命名空间划分为不同目录子树,同一个目录子树的元数据由同一个节点管理.目录子树分区分为静态和动态两种[17],静态子树分区是一种自然的负载均衡方法,动态子树分区是一种基于静态子树分区的改进方法.目录子树分区方法虽然保留了文件系统的层次结构,有利于提高文件系统读写效率,但由于无法有效处理请求密集的目录,所以无论是静态还是动态子树分区,都不能有效地平衡元数据负载.
- 哈希分区法的主要思想是:使用文件标志符或文件系统路径,通过哈希函数将元数据映射到元数据服务集群中的服务器上,由该服务器负责管理,如 zFS[18].这种方法虽然不保持文件系统的结构,但是由于可以方便感知和管理访问频繁地目录,可以将负载更均匀地分布到各个元数据服务器上.

服务器热点评价一般是用在动态负载均衡的策略中,用于及时地获取服务器的状态,作为下一步操作的依据.元数据服务热点评测主要解决的问题是使用一种方法来对当前的负载量作一个估算,为负载均衡提供依据,主要分为以下 3 类.

(1) 静态模型.主要根据 CPU、内存、操作系统和网络状况等因素,按照一定的影响因子,综合确定服务器的性能峰值.这种方法需要大量的实验测试才能得出结论,而且通用性差.

(2) 计数模型.该模型主要是按照记录历史访问的信息,依据这些数据,通过数学模型以及当前服务器的负载情况计算预测出下一阶段的负载.该模型的优点是在负载稳定的情况下能够得出比较准确的负载预测,但是其缺乏对服务器性能的感知.

(3) 实施探测.主要思路是:通过测试获取服务器的负载情况,例如服务器的响应时间、数据吞吐量和请求队列长度等,这些因素可以用来反映当前服务器的负载情况.该方式为动态模型提供了支持,但是缺点是会占用一定的系统开销.而且此方法的准确度是基于该服务器只作为元数据服务器使用的,如果服务器上存在其他服务组件时,其他组件在运行时同样会消耗一定的系统资源,那么此时的性能评估值就不准确了.正因为以上原因,我们亟需一种能够实时、准确地评估系统性能的估算方式.

目前,主流的文件系统是将集群中每个服务器上的条目的访问频度作为负载均衡的依据,根据每个服务器的负载情况来进行热点迁移,从而达到负载均衡的目的[19],然而没有考虑到由于硬件的迭代升级等现实带来的机器间性能的差异.本文根据上述不足,提出了一个新的自适应文件系统元数据负载均衡策略.该策略包括一种服务器性能评价模型和一种自适应元数据服务负载均衡算法.

2 自适应元数据服务负载均衡策略

元数据管理没有考虑异构而只是考虑访问热点的情况,容易造成系统抖动、延长负载均衡的时间,或者造成资源的浪费.另外,元数据服务集群策略不灵活,无法及时应对元数据请求暴增的情况.本节针对这些问题,提出了一种自适应元数据服务负载均衡策略,主要包括以下内容:1) 介绍一种实时服务器性能评估模型,该模型不仅能够对异构的服务器进行准确的性能评估,而且能够实时的估算服务器的性能情况;2) 提出了一种服务器负载评估方法,此方法适用于系统中存在其他占用系统资源的程序时,能够准确评估元数据服务的负载情况;同时,提出了一种自适应调整性能评估周期以达到快速负载均衡的方法;3) 提出了一种基于实时性能指标的子树迁移策略.

2.1 服务器性能评估模型

分布式文件系统负载均衡的一个关键问题是要准确地估算出这台服务器的性能情况,并给它分配相应的负载量,以达到最大限度地利用系统资源的目的.在异构的情况下,集群中的元数据服务器各不相同,即使是硬件条件一致的元数据服务器,也可能在不同时刻有着不同的性能值.传统的方法主要有两种.

- 第 1 种,根据 CPU、内存、硬盘的读写性能等,按照一定的权重给整个系统的性能做出估算.这种方法需要大量的实验,而且通用性差.
- 第 2 种,通过使用压力测试的方式,测试服务器的响应时间、速率、数据吞吐量等指标来反映服务器的整体性能.这些方法在复杂的私有云环境下无法达到实时更新服务器性能指标的目的.

所以,本文通过一种基于时延的性能估算方法来实现实时、准确地评估系统的性能.这里,时延是指数据包在客户端与服务器之间的传输时间与数据包在服务器端的总处理时间之和.服务器端的总处理时间是指该数据包的排队等待时间和处理时间之和.

文献[20]提到,在不考虑其他因素的情况下,服务器在单位时间内的平均时延随着该单位时间内并行请求数量的增加而线性增长.于是,用 Req 表示单位时间内元数据请求数量,用 Lag 表示该单位时间元数据请求的时延,根据线性关系,可得公式(1):

$$Lag=Req\times Slope+n \tag{1}$$

其中,$Slope$ 表示斜率;n 表示在没有请求的情况下,服务器本身各种因素导致的时延.我们假设两个时间点的单位时间内元数据请求数量分别为 $Req1$ 和 $Req2$,而这两个时间点对应的单位时间的请求时延分别为 $Lag1$ 和 $Lag2$,我们把数据代入公式(1),可推导出公式(2)和公式(3):

$$Lag1-Lag2=(Req1-Req2)\times Slope \tag{2}$$

$$Slope=\frac{Lag1-Lag2}{Req1-Req2} \tag{3}$$

$$P=\frac{\partial}{Slope} \tag{4}$$

在一个性能稳定的服务器中,两个时间点单位时间内元数据请求时延之差,除以两个时间点单位时间内的请求数量之差,是一个相对稳定的值 $Slope$.我们设 P 为评估元数据服务器的元数据处理能力的一个性能值.从公式(4)可以看出,P 与 $Slope$ 成反比,是 $Slope$ 的倒数的∂倍,其中,∂是常数.$Slope$ 是表示延迟的增量与请求的增量之比,它反映了随着请求的增加,延迟增加的快慢程度.也就是说,这个比值越大,反映出随着请求量的增大,延迟增加的速度越快,说明此时服务器的处理能力越差,P 值也就越小.由此可见,P 值可以实时、准确地反映元数据服务器的元数据处理能力,它是一个二维的综合性指标.而传统的评价指标(如速率、时延等)虽然比较简单、直观,但是这些指标是分散的,它们之间有一定的联系却没有系统地综合起来.性能值 P 正是把传统的指标综合起来的一个更系统、更有效的评价指标.

前文已提出,服务器由于各种原因,在不同的时间点可能出现不一样的性能情况.那么,我们每隔一段时间估算服务器的元数据处理能力,近似地给出实时的性能值.我们假设时间段 t_1(默认为 1 分钟)为估算周期.在时间

t_1 内,我们采集 n 次数据分别作为 x 轴的请求量 $Req=[x_1,x_2,x_3,x_4,\ldots,x_n]$和 y 轴的时延量 $Lag=[y_1,y_2,y_3,y_4,\ldots,y_n]$,从理论上来说,在一个性能稳定的系统里,收集到的数据会落在一条直线上.但是由于系统的测量误差和性能抖动等因素,实际上收集到的数据会离散地分布于直线的两边,呈现线性增长的趋势.为了确定这条直线从而确定 *Slope* 的大小,我们使用最小二乘法[20]来实现离散节点的线性拟合,见公式(5)~公式(7):

$$A=\frac{1}{C}\sum_{k=1}^{N}(x_k-\overline{x})(y_k-\overline{y}) \tag{5}$$

$$B=\overline{y}-A\overline{x} \tag{6}$$

$$C=\sum_{k=1}^{N}(x_k-\overline{x})^2 \tag{7}$$

其中,A 为斜率 *Slope*.

2.2 自适应元数据服务负载均衡算法

本节主要描述一种自适应元数据服务负载均衡算法.该算法的主要思路是:

- 首先,通过每一时间段 t_2($t_2<t_1$,假设 t_2 为 1s)收集系统各个硬件的负载,判断元数据服务器是否出现元数据处理请求的暴增:如果是,则启动自适应机制,缩短负载均衡的周期;否则,系统以 t_1 作为周期,根据一种新的估算元数据服务器热度的方法,计算出每个元数据服务器的负载量 *Load*.
- 同时,在元数据服务器集群中用 UDP 广播包广播自己的性能值、负载量、负载信息和时延量,选择负载水平最低的元数据服务器作为决策元数据服务器.
- 决策元数据服务器选出其中负载水平大于目标负载水平的节点,向小于目标负载水平的节点进行目录子树迁移.这样一方面可以避免集群中性能较弱的节点因为过载而成为集群瓶颈,另一方面,性能较强的节点也可以得到充分的利用.

(1) 自适应周期调整

在云计算环境下,元数据服务器的元数据处理请求暴增是有可能出现的.例如,当云桌面同时启动大量虚拟机时,同一个镜像源的请求就会在瞬间暴增,从而使得所有虚拟机的启动都比较缓慢.这样的读写请求暴增的情况发生在较短的时间内,如果我们默认的负载均衡调度周期较长,则有可能无法对暴增的元数据请求做出及时的处理,那么这样的现象就会严重影响到用户的体验.一般采取的解决方法是缩短负载均衡调度的周期,例如,将 1 分钟的周期缩短为 10s.但是,元数据服务器在进行一次负载均衡决策的时候需要收集信息、发送广播、计算决策,最后执行动态子树迁移.如果长期使用较短的周期,系统消耗会明显增加,这是很不明智的.

鉴于以上考虑,本文采用自适应的周期调整方法,即元数据服务器在较短的时间间隔内收集自身的硬件负载,检测元数据服务器短时间内负载的增加是否超过了设定的阈值,如果超过阈值,则触发自适应机制,缩短负载均衡监测机制的周期,主动向元数据服务器集群广播,触发负载均衡调度机制.

a) 自适应机制触发阈值

在云计算环境下,资源的消耗主要体现在 CPU 和内存上,所以本文采用单位时间内 CPU 和内存的使用率的变化来判断该元数据服务器的负载是否在短时间内暴增.然而,考虑到元数据服务器的 CPU 和内存的性能可能无法确定,同样的元数据请求造成的 CPU 和内存的负载增加可能相差比较大,两者增长的百分比也可能相差很大,因此难以按照一个统一的阈值来界定是否触发自适应机制.所以为了避免这种情况,本文采用一种新的方法来确定元数据服务器的硬件负载增量.我们使用元数据测试工具 mdtest 对元数据服务器进行测试:首先,记录在负载的情况下,CPU 的使用率 RC_1 和内存的使用率 RM_1;接着,使用 mdtest 发送一组元数据处理请求,并逐渐增加并行的元数据请求的数量,直到 CPU 或内存的使用率达到一定的百分比 r(默认为 50%),则停止增加并行元数据请求的发送,记下此时 CPU 的使用率 RC_2 以及内存的使用率 RM_2;然后计算两者使用率的变化,见公式(8)和公式(9).

$$\Delta RC=RC_2-RC_1 \tag{8}$$

$$\Delta RM=RM_2-RM_1 \tag{9}$$

其中,ΔRC 表示 CPU 使用率的变化,ΔRM 表示内存使用率的变化.

比较ΔRC与ΔRM,如果$\Delta RC>\Delta RM$,则表示 CPU 相对于内存来说性能较差,对负载增加的反应比较明显;如果$\Delta RC<\Delta RM$,则表示内存相对于 CPU 来说性能较差,对负载增加的反应比较明显.服务器整体性能在某一硬件存在明显瓶颈的情况下,会使得其他硬件无法发挥自身应有的性能,所以根据木桶原理,在考察服务器硬件的负载变化时,首先应该关注其硬件反应较为敏感的部分.所以我们定义:如果$\Delta RC>\Delta RM$,则 R 表示 CPU 在单位时间内使用率的变化;如果$\Delta RC<\Delta RM$,则 R 表示内存在单位时间内使用率的变化.接下来,我们设定一个负载变化的阈值 $R_{\text{thredhold}}$:如果在当次检测 $R>R_{\text{thredhold}}$,则触发自适应机制,调小负载均衡检测的周期,在元数据服务器集群中发出广播,唤醒集群中的负载均衡机制.

关于触发阈值 $R_{\text{thredhold}}$的设定,可以根据实际应用场景来设置,不同的应用场景对于自适应调整的反应速度要求不一样.本文主要考虑在云桌面启动的应用场景中,如何应对大量云桌面同时启动带来的启动风暴问题.所以本文的阈值以云桌面用户得到良好体验为目标而设定的.每一个云桌面在启动时会带来 H(假设 H=40)个 IOPS(即每秒 I/O 请求次数).假设我们的目标是让 K(假设 K=100)个云桌面用户得到较好的体验,那么需要在 KH(KH=4000)个 IOPS 时就能触发阈值,那么我们可以通过实验得出增加 KH 个 IOPS 时元数据服务器的 CPU 或者内存增加的负载,而该负载的增幅就是该元数据服务器以 K 个云桌面用户带来良好体验的目标下,启动自适应机制的阈值 $R_{\text{thredhold}}$.

b) 使用 UDP 广播的通信开销和时延分析

当启动自适应机制或者到达系统设置的周期 t_1 时,所有元数据服务器用 UDP 广播包广播自己的性能值、负载量、负载信息和时延量,选择负载水平最低的元数据服务器作为决策元数据服务器.那么使用 UDP 广播造成的通信开销和时延分析如下.

- 占用带宽分析

假设有 n 台服务器,每台服务器发包的周期为 t(这里的 t 可以是系统原来设置的周期 t_1,也可以是触发自适应的时刻),包的大小为 s.当 n 台服务器发出 UDP 广播时,假设每个广播包会在每条链路传输一次,那么每条链路的负载见公式(10).

$$D_n = s \times \frac{1}{t} \times n \tag{10}$$

在发包过程中,因为广播包只广播服务器自己的性能值、负载量、负载信息和时延量,所以实验中包的大小 s 为 94 个字节,即 752b.另外,服务器在较短的时间间隔 t_2 内收集自身的系统负载,在实验中,我们设置这个时间间隔 t_2 是 1s,那么如果触发自适应机制,则 t 是 1s 的整数倍;如果没有触发自适应机制,那么 t 就是系统原来设置的周期 t_1.这里,我们假设第 1s 就触发自适应机制,则 t 为 1s.

在小集群中,假设 n 为 10,那么 D_{10} 为 7.52kbps.在大集群中,假设 n 为 100,因为在实际应用中,100 台元数据服务器集群已经是非常大的集群,这时,每条链路的负载 D_{100} 为 75.2kbps.因为目前的集群环境基本都采用 1 000M 带宽的高速网络,所以即使在 100 台元数据服务器集群中,系统占用的带宽也很小,所以带宽方面的开销可以忽略.

- 时延分析

时延包括处理时延 T_{proc}、排队时延 T_{queue}、传输时延 T_{trans} 和传播时延 T_{prop}.因为由上文已知带宽占用率比较低,广播包没有造成网络拥塞,所以排队时延可以忽略;而且目前集群采用的是千兆以太网交换机,其处理的速率为 1.488 百万包每秒,所以处理时延也可以忽略.假设源服务器和目的服务器之间有 m–1 台交换机,那么源服务器发包到目标服务器的时延 $T_{\text{end-end}}$ 的计算见公式(11).

$$T_{\text{end-end}}=m(T_{\text{trans}}+T_{\text{proc}}) \tag{11}$$

假设包的大小为 s,带宽为 d,则传输时延 T_{trans} 由公式(12)可计算:

$$T_{\text{trans}}=s/d \tag{12}$$

假设每条链路长度为 h,光在链路中的速度为 c,则传播时延 T_{prop} 由公式(13)可计算:

$$T_{prop}=h/c \tag{13}$$

假设在 100 台元数据服务器集群中,带宽 d 为 1 000Mbps,包大小 s 为 752b,每条链路长度 h 为 20m,光在光纤中的传播速度 c 为 2.0×10^8m/s,源服务器和目标服务器之间有 3 台交换机.那么时延 $T_{end\text{-}end}$ 为 3.4ms.

另外,每台元数据服务器在接收到 $n-1$ 个广播包后,分析并选择决策元数据服务器这个过程的时间复杂度为 $O(n)$,它与集群内发送广播包的服务器的增长是成线性关系的,在小规模的集群实验中表明,这种延迟是可以忽略不计的,那么集群的增大带来的 UDP 广播线性增长导致的延迟事实上也是可以忽略的.

(2) 负载均衡机制

启动负载均衡机制时,元数据服务器首先会计算出自己的负载量 $Load$,作为判断是否触发目录子树迁移的一个条件.传统的负载均衡机制在估算元数据服务器负载的时候是采用响应时间、元数据热度等参数作为估算标准的,本文在此基础上加上对元数据处理请求等待队列长度的考虑,因为等待队列的长度从一个侧面可以体现元数据服务器在接下来一段时间的热点增长趋势,所以每个元数据服务器的负载量包括整个目录树的热度值与等待处理的元数据请求队列长度两部分.

元数据热度的统计是指每一个元数据都设有一个计数器,记录该元数据收到元数据操作(read,open 等)时,为对应的元数据的计数器加 1.同时,每次访问不仅增加该点的热值,而且该热值还会沿着目录树向上造成影响,即对其祖先节点造成影响.另外,每一个元数据的热度会随着一定时间段内访问量的减少而衰减,见公式(14)和公式(15).

$$H_{new}=H_{old}\times g(\Delta t)+1 \tag{14}$$

$$H_{ancient\_new}=H_{ancient\_old}\times g(\Delta t)+\frac{1}{2^n} \tag{15}$$

其中,H_{old} 表示受访问节点的初始的热度,H_{new} 表示受访问节点更新后的热度,$H_{ancient\_old}$ 表示受访问节点的祖先节点的初始热度,$H_{ancient\_new}$ 表示受访问节点的祖先节点更新后的热度.Δt 表示本次负载统计与上一次统计的时间间隔,g 是关于时间间隔的递减函数,n 则是表示受访问的节点和当前计算的祖先节点之间的层次数[21].在不考虑时间间隔递减函数 g 的情况下,热度值为 8 的文件在接受一个处理请求之后的热点反应如图 1 所示,颜色越深,表明增加的热点值越高.该请求的影响向上传递,而且逐层递减.

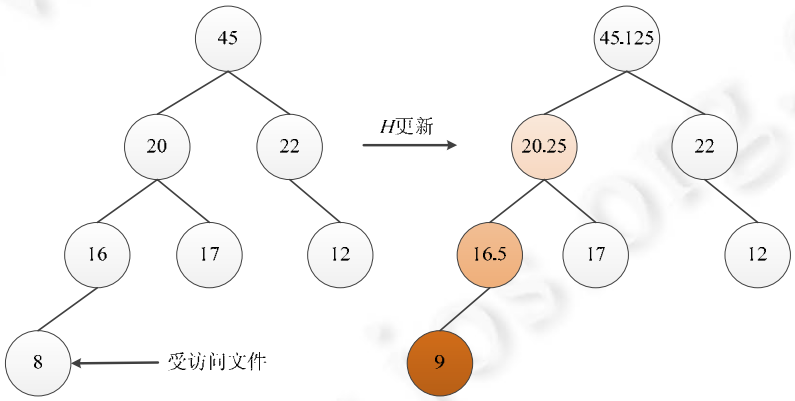

Fig.1 Heat transmission of metadata directory subtrees

图 1 元数据目录子树热度传播图

本文在估算元数据服务器的负载水平时,在热度的基础上加上对元数据处理请求等待队列长度的考虑,所以得到负载量 $Load$ 的计算公式(16):

$$Load=H+w\times L \tag{16}$$

其中,$Load$ 表示元数据服务器的负载量,L 表示等待处理的元数据请求队列的长度,w 是加权系数.

关于目录子树迁移的触发条件,传统的负载均衡机制一般只把元数据热度作为目录子树迁移的触发条件,

而不考虑服务器本身的性能差异.这样就可能出现高性能服务器负载虽然超过了集群负载的阈值,但服务器本身其实没有超载,而文件系统依然会启用负载均衡机制将负载迁移到了集群中性能较弱的服务器上,这会导致一些不必要的资源开销,而且很有可能出现性能较弱的服务器在完成子树迁移之后负载过高,服务器压力过大,又要再次进行迁移,此时就会造成系统抖动,极大地延长了负载均衡的时间.所以为了避免这种情况,本文充分考虑到了元数据服务器的性能差异.我们把从上一节获得的性能值 P 用于表示该元数据服务器处理元数据请求的能力.每个元数据服务器在元数据服务器集群中用 UDP 广播包广播自己的性能值、负载量、负载信息和时延量,并由负载水平最低的元数据服务器作为决策元数据服务器.决策元数据服务器记录发送的时间、负载均衡的周期等信息.决策元数据服务器收集集群内所有服务器的性能值 $P=[p_1,p_2,p_3,\ldots,p_n]$以及所有服务器的负载量 $Load=[load_1,load_2,load_3,\ldots,load_n]$,然后计算出集群的总性能值 P_{total} 和总负载量 $Load_{\text{total}}$,见公式(17)、公式(18).

$$P_{\text{total}}=\sum_{k=1}^{n}P_k \tag{17}$$

$$Load_{\text{total}}=\sum_{k=1}^{n}Load_k \tag{18}$$

在异构的元数据服务器中,服务器的性能越强,其应该负担的负载量越大,这样才可以充分利用集群的系统资源,又可以平衡各个元数据服务器的处理效率.通过上述的总性能值和总负载值,我们可以得到第 i 号元数据服务器在负载均衡的情况下的目标负载量 $Load_{target(i)}$,见公式(19).

$$Load_{target(i)}=\frac{P_i}{P_{\text{total}}}\times Load_{\text{total}} \tag{19}$$

接下来可以计算出第 i 号元数据服务器的现有负载值与其目标负载值的差距$\Delta Load$,见公式(20).

$$\Delta Load=Load_i-Load_{target(i)} \tag{20}$$

如果$\Delta Load>0$,表示该元数据服务器的负载值相对过高,需要向其他元数据服务器迁移目录子树;如果$\Delta Load<0$,表示该元数据服务器的负载值相对过低,可以接收其他元数据服务器迁移过来的目录子树.同时,我们设置一个触发阈值$\Delta Load_{\text{thredhold}}$,如果该周期内有元数据服务器的$|\Delta Load|>\Delta Load_{\text{thredhold}}$,则触发目录子树迁移;否则不触发.设置阈值的目的是防止迁移过于频繁而导致的系统性能抖动或消耗增加.

设置 $EXSET$,$INSET$ 分别为迁出子树元数据服务器和迁入子树元数据服务器的集合,如果元数据服务器的$\Delta Load>0$,且$|\Delta Load|>\Delta Load_{\text{thredhold}}$,则将其加入 $EXSET$;如果$\Delta Load<0$,且$|\Delta Load|>\Delta Load_{\text{thredhold}}$,则加入 $INSET$.自适应负载均衡算法如算法 1 所示.

算法 **1**. 自适应负载均衡算法.

Task T1: A self-adaption process judgement

1. **while** (true)
2. $R_{\text{change}}=R_{\text{before}}-R$; $R_{\text{before}}=R$;
3. **if** $R_{\text{change}}>R_{\text{thredhold}}$ **then**
4. **break**;
5. **else**
6. sleep 5 seconds;
7. **endif**
8. **endwhile**

Task T2: A self-adaption process initiation

9. $Req=getReq()$;
10. $Lag=gerLag()$;
11. $Queue.push(\langle Req,Lag\rangle)$; $Queue.pop()$;

12. using data in *Queue* to calculate *Slope*;
13. $P=a/Slope$;
14. $Load=H+w\times L$;
15. *broadcast*(*Load*,*P*);

Task T3: The MDS start to make load-balancing

16. $Load_{total}=0$; $P_{total}=0$;
17. **for each** MDS *i* **do**
18. $calarry.add(\langle Load_i,p_i\rangle)$;
19. $Load_{total}=Load_{total}+Load_i$;
20. $P_{total}=P_{total}+P_i$;
21. **endfor**
22. **for each** MDS *i* **do**
23. $Load_{target(i)}=P_i/P_{total}\times Load_{total}$;
24. **if** $Load_i-Load_{target(i)}>Load_{thredhold}$ **then**
25. put MDS_i into *EXSET*;
26. **endif**
27. **if** $Load_{target(i)}-Load_i>Load_{thredhold}$ **then**
28. put MDS_i into *INSET*;
29. **endif**
30. **endfor**

2.3 目录子树迁移过程

当决策元数据服务器选出负载水平较大的节点和负载水平较小的节点后,需要进行目录子树迁移.目录子树迁移[22]主要包括以下 3 个过程:首先进行一次负载信息收集工作,由决策元数据服务器匹配迁出子树的元数据服务器和迁入子树的元数据服务器及其迁移量;接下来,根据迁移量的大小,在迁出子树的元数据服务器上执行拆分子树的操作;最后,将拆分完成的子树迁移到迁入子树的元数据服务器中.

(1) 元数据服务器匹配

首先,将集合 *EXSET* 和 *INSET* 中的元数据服务器按照其$|\Delta Load|$大小做降序排列;接下来,将 *EXSET* 中的第 *i* 位与 *INSET* 中的第 *i* 位做匹配,*i* 从 1 开始,其中,*EXSET* 的第 *i* 位作为目录子树的迁出方,*INSET* 的第 *i* 位作为目录子树的接收方,直到两个集合中有一个集合的所有元数据服务器都匹配完成为止.其中,*EXSET* 的第 *i* 位向 *INSET* 的第 *i* 位迁移时,以两者的$|\Delta Load|$的较小值作为迁入迁出的负载量 $Load_m$.

(2) 子树拆分

在确定了外迁的负载量 $Load_m$ 之后,需要迁出目录子树的元数据服务器需要将负载值总和为 $Load_m$ 的目录子树拆分出来,为迁移做准备.

首先,设 *S* 和 *M* 分别为放置负载量小于 $Load_m$ 的目录子树的集合以及准备进行迁移的目录子树的集合;同时,设置一个阈值 $Load_t$ 来表示目录负载与 $Load_m$ 的误差允许范围;接下来,设 *T* 用于放置从目录树根节点出发的所有目录子树.在 *T* 中选取一个目录子树 $Tree_j$,计算其负载量 $Load_j$,其中,$Load_j$ 是指子树的所有节点负载的和.用目录子树 $Tree_j$ 的负载量 $Load_j$ 与迁入迁出的目标负载量 $Load_m$ 对比:如果它们的差在误差 $Load_t$ 范围内,则将该目录子树添加进 *M*,准备外迁;如果它们的差超出了误差 $Load_t$ 范围,而且 $Load_j$ 小于 $Load_m$,则将该目录子树添加进 *S*,然后在 *T* 中继续选取目录子树.子树拆分算法如算法 2 所示.

算法 **2**. 子树拆分算法.

1. **initial** *T*, *S* and *M*;
2. **for** *i* in the size of *T* **do**

3. calculate $Load_j$ of $Tree_j$;
4. remove $Tree_j$ from T;
5. **if** $|Load_j-Load_m|<Load_t$ **then**
6. put $Tree_j$ into M;
7. goto line (30);
8. **else if** $Load_j-Load_m<0$ **then**
9. Put $Tree_j$ into S;
10. **endif**
11. **if** T is empty **then**
12. **break**;
13. **endif**
14. **endfor**
15. Sort S in descending order and set q=0;
16. **for** q in size of S **do**
17. **if** $Load_m\tilde{}Load_q>0$ **then**
18. put S_q into M;
19. $Load_m=Load_m\tilde{}Load_q$;
20. continues;
21. **endif**
22. **if** $Load_m<Load_t$ **then**
23. **break**;
24. **endif**
25. **if** $Load_m\tilde{}Load_q<0$ and $|Load_m-Load_q|<Load_t$ **then**
26. put S_q into M;
27. **break**;
28. **endif**
29. **endfor**
30. choose M as the export set

(3) 目录子树外迁

在确定好要外迁的目录子树之后,迁出节点对该目录子树进行加锁操作,保证在迁移期间中断所有对该目录子树的元数据服务.同时,向集群中所有与该目录子树有逻辑关系的元数据服务器广播通知该目录子树即将被迁移.接下来,对该目录子树进行序列化编码,并发送到迁入节点.迁入节点在接受到目录子树的编码后进行解码,更新到自己的目录上,并且告知迁出节点已经收到迁移的目录子树.同时,广播通知集群中与该目录子树有逻辑关系的元数据服务器迁移完成.最后,迁出节点对该目录子树进行解锁操作,并删除其信息.所有与该目录子树有逻辑关系的元数据服务器更新目录信息,以保证能够正确工作.

3 实 验

本节对新提出的自适应元数据服务负载均衡策略进行验证,实验证明了该策略的可行性、有效性和稳定性.同时,把该策略与 Ceph 负载均衡策略(ceph load balancing strategy,简称 CLBS)作对比,证明了该策略更适用于异构环境,能够更有效地减少集群系统抖动,确保操作时延相对均衡.

3.1 实验环境

本实验在局域网中搭建了一个 Ceph 集群,包括 5 台异构 MDS(元数据服务器)用于文件系统的元数据管

理,1 台 MON(Monitor 服务器)用于检测集群运行情况,还有 1 台 OSD(对象存储服务器)用于文件数据存储.此外,还包括 10 台客户端机组成的一个用户集群.实验环境的配置见表 1.

Table 1 Hardware configuration

表 1 硬件配置

服务器	中央处理器	内存	硬盘	操作系统	带宽
MON	4×Xeon 5580, 2.27Ghz	4G	500G	Ubuntu12.04/kernel3.5.0	100M
OSD	4×Xeon 5580, 2.27Ghz	4G	500G	Ubuntu12.04/kernel3.5.0	100M
MDS1	8×Xeon 5580, 2.67Ghz	8G	500G	Ubuntu12.04/kernel3.5.0	100M
MDS2	4×Xeon 5580, 2.67Ghz	12G	500G	Ubuntu12.04/kernel3.5.0	100M
MDS3	4×Xeon 5650, 2.27Ghz	8G	500G	Ubuntu12.04/kernel3.5.0	100M
MDS4	2×Xeon 5650, 2.27Ghz	8G	500G	Ubuntu12.04/kernel3.5.0	100M
MDS5	2×Xeon 5650, 2.27Ghz	4G	500G	Ubuntu12.04/kernel3.5.0	100M

3.2 实验分析

根据前文提到的性能估算方法,我们对 5 个 MDS 分别进行实验.在保证 MDS 零负载和关闭负载均衡功能的情况下,使用 mdtest 工具分别对每个 MDS 进行压力测试.实验对每个 MDS 逐步线性增加元数据请求量,并记录下每个 MDS 在 8 个标记点的的时延值,记录见表 2,其中,1X 表示 1 倍的元数据请求,依此类推,8X 表示 8 倍的元数据请求.

Table 2 Delay of MDS

表 2 元数据服务器的延迟量

	1X	2X	3X	4X	5X	6X	7X	8X
MDS1	111.4	115.4	118.6	123.2	127	130.2	133.6	140
MDS2	117.2	121.5	125.2	129.8	134.1	139.1	142.5	148.3
MDS3	124.5	130.2	135	139.5	144.1	151.3	155.1	162
MDS4	135.1	141.7	148.2	154.4	161.1	167.7	174.1	180.5
MDS5	141	148.9	167.2	165.1	173	180.8	188.8	197

我们以 MDS1,MDS5 为例,画出元数据请求数量变化与延迟量的关系图,如图 2 所示.

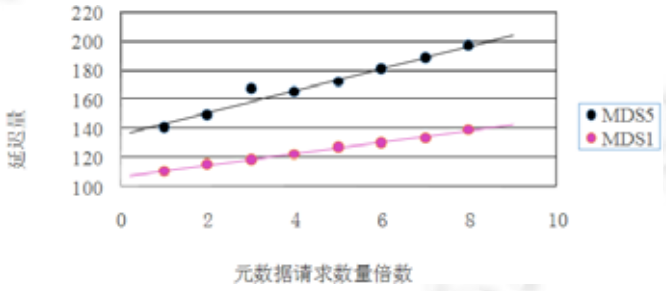

Fig.2 Relation of metadata request quantity and delay

图 2 元数据请求数量变化与延迟量的关系图

从图 2 可以看出,离散点分布明显趋于线性增长的趋势,也就证明了前文在性能估算模型中的结论,在排除其他因素的情况下,元数据请求量与该 MDS 的延迟量成线性关系;同时,我们利用最小二乘法,分别得到这些离散点的线性拟合直线的斜率,然后计算我们的性能估算值 P.我们通过计算这 5 个 MDS 的性能值,从而得到它们的性能比例为 8:6.5:5:4:3.5.

为了验证本文提出的负载均衡策略的优越性,我们采用 3 组实验,分别为关闭负载均衡机制、使用 Ceph 自带的 CLBS 负载均衡策略以及使用本文提出的负载均衡策略这 3 种方法.每组实验运行 10 小时,利用 mdtest 发送元数据请求,每 15 分钟记录下每一台 MDS i 的 CPU 使用率 $Percent_{cpu(i)}$和内存使用率 $Percent_{memory(i)}$.我们用公式(21)来表示每一台 MDS i 的负载情况 L_i:

$$L_i=0.6\times Percent_{cpu(i)}+0.4\times Percent_{memory(i)} \quad (21)$$

我们观察上述 3 组实验中每一台 MDS i 的负载情况 L_i.图 3 为关闭负载均衡机制的情况下,每台 MDS 的负载随时间的变化情况.图 4 为开启 Ceph 自带的负载均衡策略 CLBS 的情况下,每台 MDS 的负载随时间的变化情况.图 5 为使用本文提出的负载均衡策略的情况下,每台 MDS 的负载随时间的变化情况.

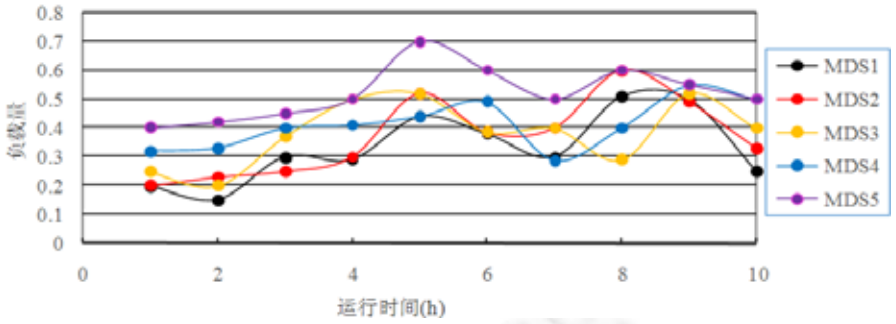

Fig.3 Close the load balancing mechanism

图 3 关闭负载均衡机制的情况

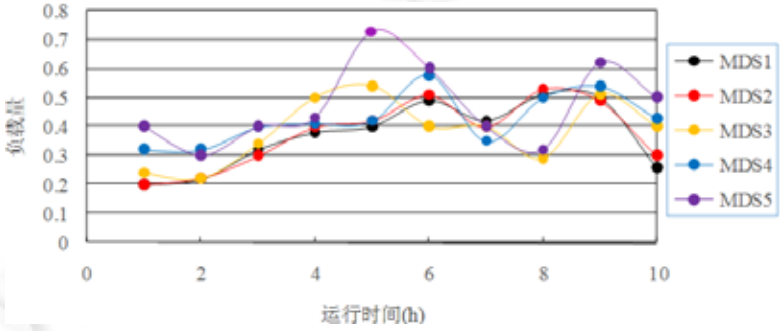

Fig.4 Use the Ceph load balancing strategy

图 4 开启 Ceph 负载均衡策略的情况

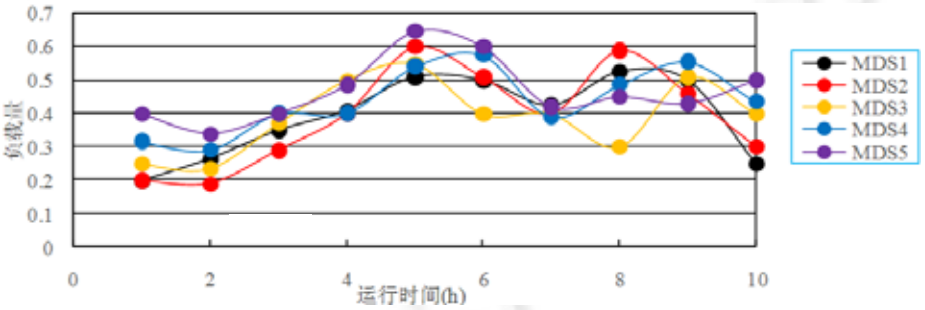

Fig.5 Use load balancing strategy presented in this paper

图 5 开启本文负载均衡策略的情况

从图 3 可以看出,由于访问的随机性,在关闭负载均衡机制的情况下,每一台服务器的负载情况差异都比较大.从图 3 可以看出,MDS5 的性能比较弱,所以它的负载值一直处于偏高状态;而 MDS1 的性能比较强,负载量一直较低,整个硬件使用率也较低.

从图 4 可以看出,使用了 Ceph 自带的 CLBS 负载均衡策略之后,整个负载情况比较接近,差异并不是很大.

但是从图 4 可以看到,在 4 时,MDS1 的负载量逐渐增高后又慢慢下降;而同时,MDS5 的负载却快速升高,然后又回落,所以可以判定这里出现了一次从 MDS1 向 MDS5 的目录子树迁移;然后,MDS5 又将部分目录子树再次迁出,导致系统性能抖动.发生这种情况的主要原因是:CLBS 只考虑到每个 MDS 的热点值对比而没有考虑服务器异构的问题,所以在没考虑到 MDS5 性能较弱的情况下向其迁入热度过多的目录子树,从而导致 MDS5 负载暴增,所以不得已再次进行迁移,本来一次可以实现的子树迁移过程,CLBS 用了两步才完成,这就是我们前文提到的 Ceph 负载均衡策略在异构服务器上的缺点.

从图 5 可以看出,在开启本文提出的负载均衡策略的情况下,每台 MDS 负载量相差不远,波动相对于 CLBS 来讲也小了很多,整个集群表现比 CLBS 更好.实验结果表明,本文提出的方法相对于 Ceph 自带的 CLBS 负载均衡策略来说,在处理异构服务器问题上表现更好.

为了表示整个集群的负载均衡程度,我们计算集群中某一个时刻 MDS 负载量之间的差异性,差异值用 D_L 表示,见公式(22).

$$D_L=\sqrt{\sum_{i=1}^{n}(L_i-\bar{L})^2} \tag{22}$$

用某一个时刻的 MDS 负载量之间的差异值 D_L 来表示该时刻集群的负载均衡程度.接下来,将 10 个时刻的 MDS 负载量的差异值 D_L 相加作为该实验的 MDS 集群的负载均衡程度 D_{total}.D_{total} 的数值越小,表示 MDS 集群的负载程度越好.

图 6 为未开启负载均衡机制的情况下、开启 Ceph 自带的负载均衡策略 CLBS 的情况下和使用本文提出的负载均衡策略的情况下,整个集群的负载差异值的变化.

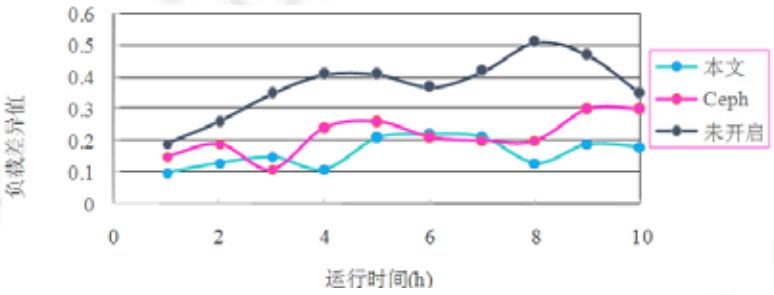

Fig.6 Comparison of variance of three kinds of scheme

图 6 3 种方案的负载差异值对比

从图 6 可以看出,在这 10 个小时中,整个集群的访问量有两次高潮.未开启负载均衡策略的实验中,负载均衡度基本随着访问量的变化而变化;而开启 Ceph 自带的负载均衡策略 CLBS 和使用本文提出的负载均衡策略两种方法都能收敛到一个较好的负载均衡水平.特别是本文的方法,收敛得更快,波动更小.通过计算,可以得到未开启负载均衡机制的实验中,MDS 集群的负载均衡程度 D_{total} 值为 3.2;而开启 Ceph 负载均衡策略 CLBS 的实验中,MDS 集群的负载均衡程度 D_{total} 值为 1.98.相对于第 1 组实验来讲,MDS 集群的负载均衡程度有了明显的改善.使用本文的负载均衡策略的实验中,MDS 集群的负载均衡程度 D_{total} 值为 1.53,相对于 CLBS 方案的实验,MDS 集群的负载均衡程度也有了进一步的改善.

接下来,我们通过实验验证本文提出的方案中的自适应机制的有效性和可用性.本文描述的自适应机制是指系统监控元数据服务器的负载情况,如果负载在短时间内发生较大的变化,则启动自适应机制,不再等待周期结束,而是即刻启动负载均衡检测机制查看是否出现过载现象:如果是,则进行目录子树的迁移.实验使用 mdtest 针对 MDS1 进行 5 分钟的元数据测试,实验期间出现两次元数据请求激增,分别在启用自适应机制和未启用自适应机制的情况下观察 MDS 负载的变化,如图 7 所示.

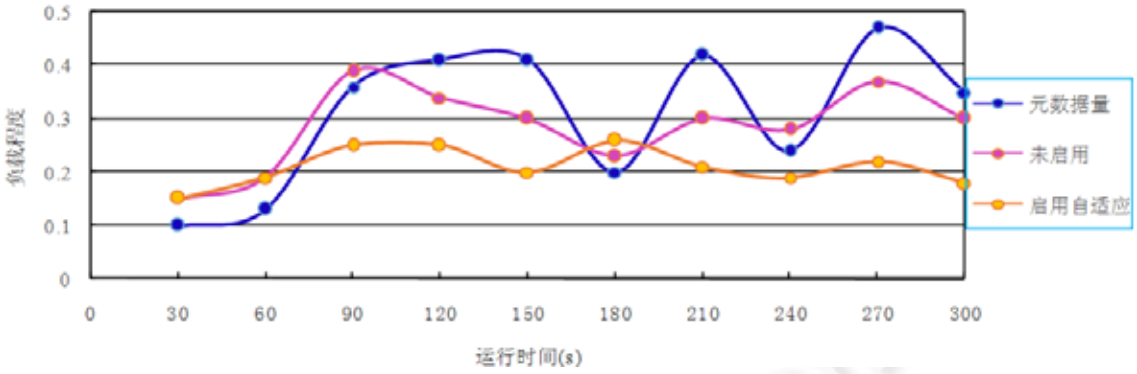

Fig.7 Experiment of adaptive mechanism

图 7 自适应机制实验

从图 7 可以看出,采用自适应机制的 MDS,在元数据请求激增后的一小段时间内就开启负载均衡机制,迁移了部分目录子树,从而降低了系统的负载;而未启用自适应机制的 MDS 则反应比较慢,无法快速转移过高的元数据热度,因此所有访问该 MDS 元数据服务的客户端的请求都会受到一定的影响.实验结果表明,采用自适应机制可以快速应变 MDS 负载的短时间激增.

为了验证本文提出的负载均衡策略在减小系统性能抖动方面的优势,我们对 Ceph 的负载均衡策略和本文的负载均衡策略进行比较,分别统计两种方法的子树迁移次数,我们可以从图 8 清晰地看出两种方法在不同时间段内进行目录子树迁移的次数的不同.

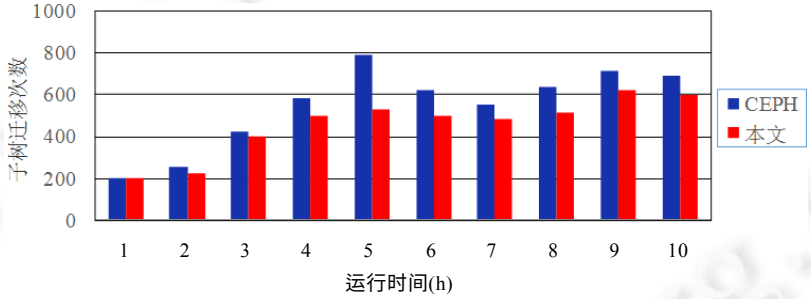

Fig.8 Migration number of directory subtrees migration

图 8 目录子树迁移次数

从图 8 可以看出,相对于 Ceph 的元数据负载均衡策略而言,本文提出的算法在同样的元数据请求下需要的子树迁移次数更少,这种方法不仅纠正了严重的系统性能抖动,而且减少了子树迁移中所消耗的 CPU、内存、I/O 资源和网络流量.

4 总 结

文件系统作为存储系统中重要的一个分支,其性能表现一直是业界、学界研究的热点.而在文件系统中,文件元数据的处理性能对整个文件系统性能有重要的影响.我们可以使用元数据服务集群去实现文件系统元数据服务性能的可扩展性,同时解决单点故障问题.与此同时,一个高效的文件系统元数据负载均衡策略使得元数据服务集群可以最大化地利用其物理资源,避免不必要的开销.本文提出了一个新的文件系统元数据负载均衡策略,其中包括一种基于服务器负载变化的检测周期自适应调整机制和一种新的自适应元数据服务负载均衡算法.最后,通过对比实验证明了该策略的可实施性、有效性和稳定性.

References:

[1] Jiang T, Hou R, Zhang LX, Zhang K, Chen LC, Chen MY, Sun N. Micro-Architectural characterization of desktop cloud workloads. In: Proc. of the 2012 IEEE Int'l Symp. on Workload Characterization (IISWC). 2012. 131−140. [doi: 10.1109/IISWC.2012.6402917]

[2] Wang F, Nelson M, Oral S, Atchley S, Weil S, Settlemyer BW, Caldwell B, Hill J. Performance and scalability evaluation of the Ceph parallel file system. In: Proc. of the 8th Parallel Data Storage Workshop. New York, 2013. 14−19. [doi: 10.1145/2538542.2538562]

[3] Ghemawat S, Gobioff H, Leung ST. The Google file system. In: Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP 2003). New York, 2003. 29−43.

[4] Shvachko K, Kuang H, Radia S, Chansler R. The hadoop distributed file system. In: Proc. of 2010 IEEE the 26th Symp. on Mass Storage Systems & Technologies. 2010(11):1−10. [doi: 10.1109/MSST.2010.5496972]

[5] Weil SA, Brandt SA, Miller EL, Long DDE, Maltzahn C. Ceph: A scalable, high-performance distributed file system. In: Proc. of the 7th Symp. on Operating Systems Design and Implementation. Washington, 2006. 307−320.

[6] Liu JL, Zhang YL, Yang L, Guo MY, Liu ZJ, Xu L. SAC: Exploiting stable set model to enhance cache files. Journal of Computer Science & Technology, 2014,29(2):293−302. [doi: 10.1007/s11390-014-1431-z]

[7] Kim T, Noh SH. pNFS for everyone: An empirical study of a low-cost, highly scalable networked storage. Int'l Journal of Computer Science & Network Security, 2014,14(3):52−59.

[8] Sun Microsystems, Inc. Lustre file system: High-Performance storage architecture and scalable cluster file system. 2008. https://www.sun.com/offers/docs/LustreFileSystem.pdf

[9] Rogers GL, Hanley J, Mohr R. Data management practices on large-scale lustre scratch file systems. In: Proc. of the 14th Conf. on Extreme Science and Engineering Discovery Environment (XSEDE 2014). Atlanta, 2014. 1−6. [doi: 10.1145/2616498.2616545]

[10] Thomson A, Abadi DJ. CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems. In: Proc. of the 13th USENIX Conf. on File and Storage Technologies (FAST 2015). Santa Clara, 2015. 1−14.

[11] Thomson A, Diamond T, Weng SC, Ren K, Shao P, Abadi DJ. Calvin: Fast distributed transactions for partitioned database systems. In: Proc. of ACM SIGMOD Int'l Conf. on Management of Data (SIGMOD 2012). Scottsdale, 2012. 1−12. [doi: 10.1145/2213836.2213838]

[12] Yuan J, Zhan Y, Jannen W, Pandey P, Akshintala A, Chandnani K, Deo P, Kasheff Z, Walsh L, Bender MA, Farach-Colton M, Johnson R, Kuszmaul BC, Porter DE. Optimizing every operation in a write-optimized file system. In: Proc. of the 14th USENIX Conf. on File and Storage Technologies (FAST 2016). Santa Clara, 2016. 1−14.

[13] Jannen W, Yuan J, Zhan Y, Akshintala A, Esmet J, Jiao Y, Mittal A, Pandey P, Reddy P, Walsh L, Bender M, Farach-Colton M, Johnson R, Kuszmaul BC, Porter DE. BetrFS: A right-optimized write-optimized file system. In: Proc. of the 13th USENIX Conf. on File and Storage Technologies (FAST 2015). Santa Clara, 2015. 301−315.

[14] Esmet J, Bender MA, Farach-Colton M, Kuszmaul BC. The TokuFS streaming file system. In: Proc. of the USENIX Conf. on Hot Topics in Storage & File Systems. 2012. 1−5.

[15] Liu J, Zhang JW, Shao BQ, Dong HQ, Liu ZJ, Xu L. Metadata server clustering system for EB-scale storage. Zhong Guo Ke Xue/Science China, 2015,45(6):721−738 (in Chinese with English abstract). [doi: 10.1360/N112014-00330]

[16] Liu Z, Zhou XM. A metadata management method based on directory path. Ruan Jian Xue Bao/Journal of Software, 2007,18(2):236−245 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/18/236.htm [doi: 10.1360/jos180236]

[17] Chen T, Xiao N, Liu F. Adaptive metadata load balancing for object storage systems. Ruan Jian Xue Bao/Journal of Software, 2013, 24(2):331−342 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4177.htm [doi: 10.3724/SP.J.1001.2013.04177]

[18] Rodeh O, Teperman A. zFS-A scalable distributed file system using object disks. In: Proc. of the IEEE Nasa Goddard Conf. on Mass Storage Systems & Technologies. 2003. 207−218.

[19] Zhang X, Li L, Wang S, Yang F. Improved selective randomized load balancing in mesh networks. ETRI Journal, 2007,29(2):255−257. [doi: 10.4218/etrij.07.0206.0215]

[20] Zhang JL, Qian W, Xu XH, Wan J, Yin YY, Ren YJ. WLBS: A weight-based metadata server cluster load balancing strategy. Int'l Journal of Advancements in Computing Technology, 2012,4(1):77−85. [doi: 10.4156/ijact.vol4.issue1.9]

[21] Hua Y, Zhu Y, Jiang H, Feng D, Tian L. Supporting scalable and adaptive metadata management in ultralarge-scale file systems. IEEE Trans. on Parallel and Distributed Systems, 2011,22(4):580−593. [doi: 10.1109/TPDS.2010.116]

[22] Li B, He Y, Xu K. Distributed metadata management scheme in cloud computing. In: Proc. of the 6th Int'l Conf. on IEEE Pervasive Computing and Applications (ICPCA). 2011. 32−38. [doi: 10.1109/ICPCA.2011.6106475]

附中文参考文献:

[15] 刘健,张军伟,邵冰清,董欢庆,刘振军,许鲁.支持 EB 级存储的元数据服务器集群系统.中国科学:信息科学,2015,45(6):721−738. [doi: 10.1360/N112014-00330]

[16] 刘仲,周兴铭.基于目录路径的元数据管理方法.软件学报,2007,18(2):236−245. http://www.jos.org.cn/1000-9825/18/236.htm [doi: 10.1360/jos180236]

[17] 陈涛,肖侬,刘芳.对象存储系统中自适应的元数据负载均衡机制.软件学报,2013,24(2):331−342. http://www.jos.org.cn/1000-9825/4177.htm [doi: 10.3724/SP.J.1001.2013.04177]

佘楚玉(1980－),女,广东潮州人,博士,讲师,主要研究领域为云存储,云安全.

刘育擘(1991－),男,博士生,CCF 学生会员,主要研究领域为高性能存储,高性能计算,大数据.

温武少(1969－),男,博士,教授,博士生导师,CCF 专业会员,主要研究领域为云计算,计算机网络,计算系统安全,大数据.

贾殷(1991－),男,硕士,主要研究领域为CDN,SDN,分布式存储.

肖扬(1989－),男,硕士,主要研究领域为云计算,存储.