计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2018.20160796
55(2):438-446,2018

基于代理的并行文件系统元数据优化与实现

易建亮[1] 陈志广[1] 肖 侬[1,2] 卢宇彤[3]
[1](国防科技大学计算机学院 长沙 410073)
[2](高性能计算国家重点实验室(国防科技大学) 长沙 410073)
[3](中山大学数据科学与计算机学院 广州 510275)
(jianliang.yi@foxmail.com)

Proxy Based Metadata Optimization and Implementation in Parallel Filesystem

Yi Jianliang[1], Chen Zhiguang[1], Xiao Nong[1,2], and Lu Yutong[3]
[1](*College of Computer, National University of Defense Technology, Changsha* 410073)
[2](*State Key Laboratory of High Performance Computing (National University of Defense Technology), Changsha* 410073)
[3](*School of Data and Computer Science, Sun Yat-sen University, Guangzhou* 510275)

Abstract In high-performance computing environment, parallel file system faces a mega client. These clients often issue a large number of concurrent IO request to the system in the same period of time, making the metadata server under a huge pressure. On the other hand, concurrent read and write requests from these clients often relate to the same directory. It makes it difficult to schedule work load across multiple servers. Therefore, we add a proxy server between the client and the metadata server and propose corresponding optimization methods to reduce the work load of the metadata server. In this paper, we realize two aspects of optimization based on proxy server. First of all, since the high-performance computing program often access files concurrently, we consider merging the numerous requests into a big one and then sent it to metadata server. Secondly, concurrent IO from the high-performance computing program often points to the same directory. Traditional metadata load balancing mechanism commonly use sub-tree partitioning method to dispatch work load across multiple server. This method is unable to realize load balancing in the situation where all operations relate to the same directory. The paper realizes fine-grained load balancing by scheduling the operations from the same directory to the plurality of metadata servers.

Key words proxy server; high concurrency; high performance computing; parallel file system; load balancing

摘 要 在高性能计算环境中,并行文件系统面临百万量级的客户端,这些客户端往往在同一时间段内发出大量并发I/O请求,使元数据服务器承载巨大的压力.另一方面,这些客户端发出的并发读写请求往往指向同一目录,导致很难将元数据负载调度到多个服务器上.为此,提出在并行文件系统的客户端和元数据服务器之间增加一级代理(proxy),并给出相应的优化措施降低元数据服务器的负载.在元数据代理上实现2方面的优化:1)由于高性能计算程序往往并发访问大量的文件,可以考虑通过元数据聚合将大量请求合并成1个请求发送到元数据服务器上,降低元数据服务器的负载;2)高性能计算程序的

收稿日期:2016-11-02;**修回日期**:2017-02-21
基金项目:国家重点研发计划(2016YFB1000302);国家自然科学基金项目(U1611261,61433019,61402503);广东省引进创新创业团队项目(2016ZT06D211)
This work was supported by the National Key Research and Development Program (2016YFB1000302), the National Natural Science Foundation of China (U1611261, 61433019, 61402503), and the Introduction of Innovative and Entrepreneurial Team Project of Guangdong Province of China (2016ZT06D211).
通信作者:陈志广(zhiguang.chen@nscc-gz.cn)

并发 I/O 往往指向同一目录,而传统的元数据负载均衡机制一般采用子树划分的方法将元数据负载调度到多个元数据服务器上,无法实现针对同一目录元数据操作的负载均衡,通过代理将针对同一目录的元数据操作调度到多个元数据服务器上,实现细粒度的负载均衡.

关键词 代理服务器;高并发;高性能计算;并行文件系统;负载均衡

中图法分类号 TP311.1

数值计算已成为继理论推理、实验验证之后的第三大科学研究手段,随着人们越来越多地借助计算手段解决科学研究、工业生产中的问题,人类社会对高性能计算系统的需求日益增加,并催生了天河[1]、Titan[2]、红杉[3]等一系列超大规模高性能计算系统.这些高性能计算系统包含数百万计算核心,能够支持数百万进程的并发执行,协同处理 TB 级、甚至 PB 级的数据.在处理这些数据的过程中,计算进程会频繁地执行文件输入/输出(input/out, I/O)操作.为此,高性能计算系统一般配备并行文件系统实现数据的管理和 I/O.例如,Titan 采用 Lustre[4]管理数据和存储资源;天河系列超级计算机在 Lustre 的基础上辅以 H2FS[5]实现异构存储资源的管理;红杉则采用 IBM 研发的 GPFS[6].并行文件系统的所有功能由元数据服务器和数据服务器实现.其中,元数据服务器是整个并行文件系统的核心.早期的高性能计算系统规模较小,并行文件系统元数据服务器承载的压力较低,一般采用单节点的元数据服务器即可满足整个计算系统的 I/O 需求.随着计算系统规模的不断增大,尤其是百万核系统(天河二号包含 312 万个计算核心,Titan 包含 56 万个运算核心)的陆续出现,传统的单节点元数据服务器逐渐不能满足高并发访问需求.因此,各种并行文件系统逐步采用分布式元数据管理方式,提供可扩展的元数据服务.例如,Lustre 从 2.X 版本以后开始提供分布式元数据服务,采用子树划分的方式将元数据分布到多个元数据服务器上.分布式元数据管理能够有效提高元数据服务能力,但是,鉴于计算系统规模不断增大,尤其是 E 级计算系统在不久的将来即将出现,并行文件系统的分布式元数据管理仍然面临以下 2 方面的挑战:

1) 高性能计算系统中计算节点数目不断增加,且经常出现大量节点同时产生并发 I/O 请求的情况,传统的分布式元数据管理机制逐渐难以应对超大规模计算系统产生的并发元数据操作;

2) 对于高性能计算系统中运行的一个任务,参与该任务的所有进程往往在同一时间点向同一文件夹发出大量的 I/O 请求,这会导致该文件夹下很重的元数据访问负载,而传统的基于子树划分的负载均衡机制并不能将同一文件夹下的元数据负载分散到多个元数据服务器上.

本文针对以上 2 方面的挑战,结合高性能计算系统中经常出现属于同一任务的大量进程向同一文件夹发出并发元数据操作的负载特点,在并行文件系统中引入代理机制,通过代理上的元数据聚合和动态负载均衡,显著提升元数据操作的性能.实验表明,本文提出的基于代理的元数据优化机制具备优秀的平均文件 I/O 性能及可扩展的元数据管理方案.

1 高性能计算系统的 I/O 特征

在高性能计算系统中运行的任务往往包含大量的进程,这些进程相互协作、不断迭代以完成计算任务.具体地,所有进程每完成一个阶段的计算任务后执行 1 次同步操作、实现交互通信,然后进入下一阶段的执行,每个执行阶段都被称作一个迭代步,一个任务往往要经历成千上万的迭代步才能最终完成.任务执行过程中一旦在某个迭代步出现了进程失效,最直接的应对方法就是重启任务,从第 1 个迭代步重新执行,直至该任务的所有迭代步顺利完成.由于高性能计算系统的规模不断增大,系统中运行的任务所包含的进程不断增多,进程失效已成为经常发生事件.如果每次出现进程失效都从第 1 个迭代步重新执行,这将导致计算任务永远无法完成.为此,研究人员引入检查点(checkpoint)机制,应对计算过程中的频繁进程失效.检查点机制的具体做法是:计算任务每隔若干迭代步就把整个任务的执行状态持久保存在底层并行文件系统中,一旦出现进程失效,任务可以从最近保存的检查点重启,而不必从第 1 个迭代步重新开始执行,这样,计算任务每保存一个检查点即表明计算过程向前推进一步,最终一定能够保证计算任务顺利完成.研究表明,读写

检查点已成为高性能计算系统中最重要的 I/O 行为之一.

高性能计算系统中另一类重要的 I/O 操作是输出中间结果.典型的高性能计算程序包含大量的迭代步,各迭代步的中间结果也具有重要科学价值.为此,研究人员可能选择一些重要迭代步输出其中间结果,并使用数据可视化等手段加以观测、分析.输出中间结果的 I/O 模式与输出检查点类似,但检查点只有在程序因故中断、再次恢复执行时才可能读取,读取的几率较低,而输出的中间结果一般都会再次读入到计算系统中加以分析.

高性能计算程序一般采用 2 种模式将其产生的检查点或中间结果保存在并行文件系统中,即共享模式(shared mode)和独占模式(independent mode).所谓共享模式,就是该应用程序的所有并发进程将输出数据(检查点或中间结果)写到同一个共享的大文件中.相应地,独占模式是指每个进程将输出数据写到各自的文件中.由于高性能计算系统的规模不断增大,能够支撑的并发进程不断增多,以共享模式输出数据将会导致大量进程针对同一个文件的锁竞争,从而降低数据输出效率.因此,共享模式逐渐不适用于超大规模高性能计算系统.然而,同样因为高性能计算系统规模不断变大的原因,以独占模式输出数据会产生大量的文件,给数据管理带来不便.为此,研究人员采用折中方案,将高性能计算程序的所有进程分组,组内的若干进程以共享模式将输出数据写到同一个共享文件中,每个组生成一个独立的文件.这种方法能够显著降低应用程序产生的文件数量,同时能够将针对同一文件的锁竞争限制在可控范围之内.但是,随着计算规模的不断增大,应用程序产生的文件仍然较多.

高性能计算程序在某个迭代步输出数据(中间结果)时,往往将该迭代步产生的所有文件输出到同一个目录下,且该目录下所有文件的文件名由应用程序保证不会发生冲突.因此,高性能计算应用的典型 I/O 模式可简单归纳为:大量进程在同一目录下并发读写大量文件,且这些文件命名不会发生冲突.

2 引入元数据代理的并行文件系统架构

在高性能计算环境中,并行文件系统面临百万量级的客户端,这些客户端往往在同一时间段内发出大量并发 I/O 请求,使元数据服务器承载巨大的压力.另一方面,这些客户端发出的并发读写请求往往指向同一目录,导致很难将元数据负载均衡调度到多个服务器上.为此,本文提出在并行文件系统的客户端和元数据服务器之间增加一级代理(proxy),并提出相应的优化措施降低元数据服务器的负载.本文在元数据代理上实现 2 方面的优化:1)由于高性能计算程序往往并发访问大量的文件,可以考虑通过元数据聚合将大量请求合并成一个请求发送到元数据服务器上,降低元数据服务器的负载;2)高性能计算程序的并发 I/O 往往指向同一目录,而传统的元数据负载均衡机制一般采用子树划分的方法将元数据负载调度到多个元数据服务器上[7],无法实现针对同一目录元数据操作的负载均衡,本文通过代理将针对同一目录的元数据操作调度到多个元数据服务器上,实现细粒度的负载均衡.

如图 1 所示,系统由 3 个组件构成:客户端(client)、元数据服务器(meta data server, MDS)、代理服务器(proxy server).客户端实现了部分接口,提供给上层应用程序调用;元数据服务器存储每个文件/目录的关键元数据,主要功能包括管理文件系统名字空间、文件/目录和对象物理存储位置之间的映射[8];代理服务器实现客户端和元数据服务器之间的桥接,接受客户端的 I/O 请求,经适当处理后转发到元数据服务器,代理服务器由多个节点实现,每个节点负责管理元数据服务器集群的部分节点,本文提出的优化措施将集中在代理服务器上实现.

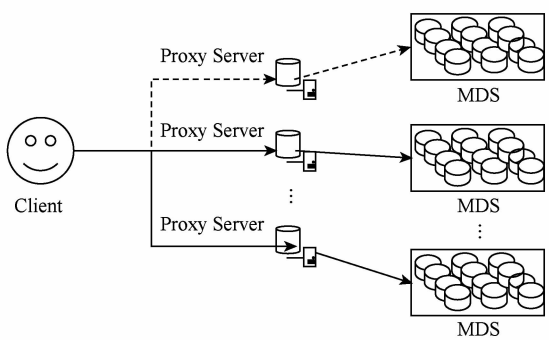

Fig. 1 System architecture

图 1 系统架构图

3 基于代理的元数据聚合

我们认识到,并行文件系统的工作负载本质上是动态的,随着上层应用和数据集的变化,数据块和元数据访问请求也随之发生变化.在用户创建文件/文件夹过程当中,需要依靠文件系统遍历目录项来判断是否已有重名文件/文件夹存在,传统文件系统

采用加锁机制来维护目录项,目录项是一种临界资源,进程之间只能通过互斥来访问,为此,每个进程在进入临界区之前,先对临界资源进行检查,看它是否正被其他进程访问,如果此刻该临界资源未被访问,进程便可以进入临界区对该资源进行访问,并设置它正被访问的标志;如果此刻该临界资源正被某进程访问,则本进程不能进入临界区.面对高性能计算环境大量并发 I/O 请求的场景,这种方式将产生无法接受的响应延迟,严重制约了高性能计算所带来的并发处理能力.针对以上问题,我们提出了基于代理的元数据聚合解决方案.

来自客户端的 I/O 请求首先由代理进行优化处理,再由其负责转发至元数据服务器端.代理服务器维护一个请求队列来接收来自客户端的 I/O 请求,并且为每个 MDS 维护一个转发队列,如 $\{Serv_1, Serv_2, Serv_3, Serv_4\}$,转发队列用于存储优化后的 I/O 请求.优化步骤包括以下 3 个部分:

1) 目录子树分配表.根据第 4 节提出的负载均衡策略,系统会自动生成一张目录子树分配表.为了提升查表效率,目录树分配表采用 Hash 表来保存.代理从请求队列中依次取出 I/O 请求,将请求的访问路径与目录子树分区表中的路径进行比对,从而找到相应的 MDS,并将此请求添加到相应的服务器转发队列中.

2) 最长路径名匹配.当查找/usr/local/cofs/etc/config 这个路径时,会匹配/usr/local/cofs 和/usr/local/cofs/etc 这 2 条路径,因为它们同时符合要求,借鉴 TCP/IP 协议中 IP 路由查找的最长匹配原则[9], /usr/local/cofs/etc 能匹配更长的目录路径,所以选择/usr/local/cofs/etc 作为索引来选择 MDS.当列表中出现多个最长路径名匹配的条目时,选择其中路径名长度最小的条目.

3) 组内聚合操作.经过以上步骤,请求包被分成 $Serv_1, Serv_2, Serv_3, Serv_4$ 四部分,将每个组别的请求聚合成 1 个请求包,将请求包保存到相应转发队列.

4 基于代理的元数据负载均衡

在分布式元数据管理方式中,为了防止热点出现,系统需要一套负载均衡机制,以实时监控节点状况,调整负载分配,达到均衡状态.根据系统运行阶段的不同,分为静态均衡机制和动态均衡机制.静态均衡机制在文件系统启动阶段运行,以使得系统达到一个静态的、粗略的均衡状态;动态均衡机制作为文件系统的子服务,在文件系统运行时执行,以实时监控各节点情况,实现系统动态均衡.

4.1 静态均衡机制

4.1.1 目录树分区

在文件系统启动阶段,为了将名称空间分配给各节点,需要对任务进行划分,达到初始均衡状态.借鉴 CEPH[7]存储系统策略,我们将文件系统元数据组织成一颗多叉树结构,采用目录树分区方式来组织文件系统元数据.具体方式如图 2 所示:

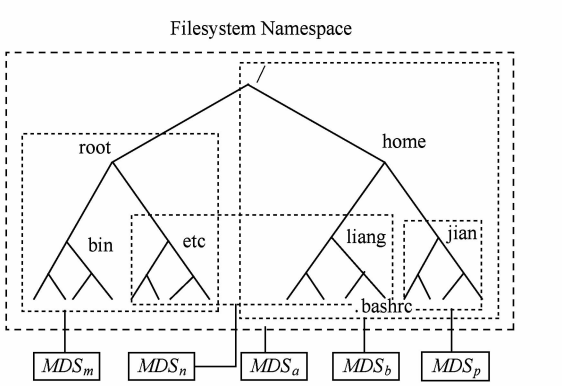

Fig. 2 Partitions of namespace

图 2 名称空间的分区

4.1.2 目录树分配表

为了使系统达到初始均衡状态,我们对目录树进行切分[10],将整个目录树划分成若干个子目录树结构(根据节点数量确定),并根据各节点配置情况,将这些子目录树分摊到相应节点,从而达到静态均衡.我们首先假设所有元数据的负载大小是相同的,例如每个元数据有一次访问.图 1 每个虚线框代表划分的子树,MDS_m, MDS_n, MDS_p, MDS_b 代表分配的节点.访问次数为当前情况下每个元数据的访问请求量,初始情况下假设为 1,即只有一条访问请求访问当前元数据;之后每当有一条新的请求访问某条元数据时,就将对应的访问次数加 1.目录树分配表记录了子树与节点的对应关系,表 1 记录了图 2 分配策略下的对应关系.

Table 1 Partition Table of Directory Subtree

表 1 目录子树分配表

Subtree Path of Directory	MDS	Number of Times
/root/etc	MDS_n	1
/root/bin	MDS_m	1
/home/jian	MDS_p	1
/home/liang	MDS_b	1

4.2 动态均衡机制

随着时间的推移,每个元数据的负载发生了变化(某些元数据同时有多个访问,某些元数据则访问量为0),导致各节点负载发生了变化,服务器集群的均衡状态被破坏(即某些节点请求量超过了它的处理能力,出现服务失效;而某些节点则无请求到来,处在空闲状态,导致资源浪费).此时,需要代理服务器的动态均衡机制进行调整,重新回到均衡状态,如图3所示:

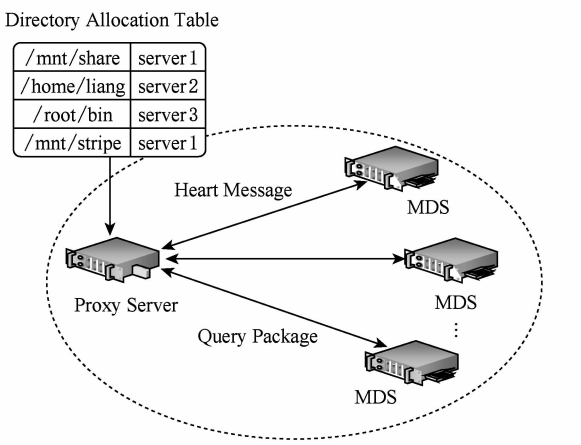

Fig. 3 Graph of load-balance algorithm

图3 负载均衡算法图解

4.2.1 负载计算方法

本文使用访问延迟来衡量某个服务器上的负载情况(元数据的访问请求是小文件I/O操作,每个请求的服务时间的差异性不大,因而使用访问延迟作为衡量指标),对于连续I/O操作,则使用吞吐量比较合理[11].在某个时刻 t,观测服务器 S_i 上的访问延迟为 $Latency_i(t)$.使用一个基于权重的平均来计算访问延迟,避免某个时刻的访问延迟突然变得很大影响综合情况[12].

$$Latency_i(t)=(1-\alpha)\times Latency_i(t)+\alpha\times Latency_i(t-1). \quad (1)$$

设在时刻 $t-1$ 到时刻 t 这段时间内,服务器 S_i 上的平均访问延迟为 $Latency_i(t)$,其中 i 代表第 i 个节点,t 为时刻,α 为权重系数.按照式(1)计算权值后,各节点以心跳的方式将计算结果反馈给代理服务器(如图2).则所有服务器上的加权平均访问延迟为[13]:

$$AvgLatency(t)=\sum_{i=1}^{n} w_i(t-1)\times Latency_i(t), \quad (2)$$

其中 $w_1(t),w_2(t),\cdots,w_n(t)$ 为各节点 t 时间所分配的权值.

4.2.2 负载均衡策略

当 $Latency_i(t)\leqslant AvgLatency_i(t)$ 时,将服务器 S_i 归为集合 A;当 $Latency_i(t)>AvgLatency_i(t)$ 时,将服务器 S_i 归为集合 B.其中,集合 A 包含负载较轻的节点,集合 B 包含负载较重的节点.我们的目标是动态地调整服务器负载,使得集合 B 中服务器将部分负载转移到集合 A 中服务器上,并调整目录分配表条目,达到动态负载均衡的目的.

代理服务器节点从集合 A 中选择响应延迟超出平均值一定值的节点作为迁移源,从集合 B 中选择响应延迟低于平均值一定值的节点作为迁移目的地.

如图4所示,均衡任务由代理服务器检测到后,发起元数据迁移任务,并由代理服务器负责选择迁移源和迁移目的地.源服务器接收到迁移任务后,开始具体执行数据复制,之后源服务器将源数据删除[14],并将操作结果反馈给代理服务器.

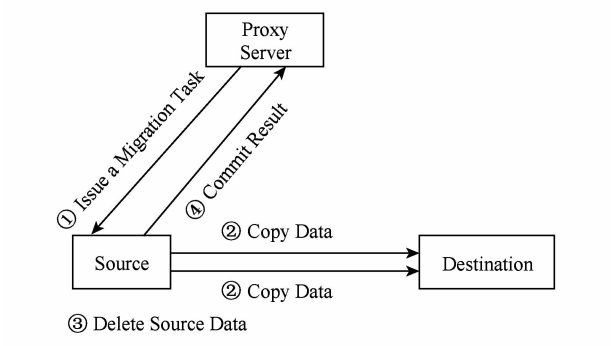

Fig. 4 The schedule of metadata migration

图4 元数据迁移流程

4.2.2.1 副本策略

代理服务器负责监控集群,判断集群是否处于失衡状态,并在失衡状态时触发系统负载均衡机制.均衡器对集合 A 和 B 中的节点按照延迟大小进行排序,如集合 A 排序后为 $\{a,b,c,d\}$,集合 B 排序后为 $\{m,n,o,p\}$.然后在集合 A 中选择延迟比较高的若干节点作为迁移源(如选择 m),在集合 B 中选择同等数据的延迟低的节点作为迁移目的地(如选择 p),将高延迟和低延迟节点配对(如 $m\rightarrow p$).考虑到元数据集群规模很小,不超过10个节点,排序过程并不会对系统性能产生很大影响.将每一组配对中前一个节点上的目录树复制一份到配对后一个节点上,如图5所示.

在选择需要迁移的元数据时,当前区域的代理

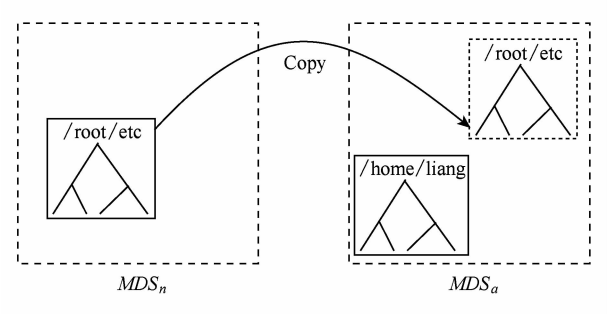

Fig. 5 Copy of directories subtree

图 5 目录子树复制

节点会分别统计配对的 2 个节点(此处为节点 m 和 p)所有元数据的总访问请求数量(这里假设节点 m 上的总访问请求数量为 x,节点 p 上的总访问请求数量为 y),之后节点 m 将$\frac{x-y}{2}$请求数量迁移到节点 p 上.然后在节点 m 上找若干个元数据,这些元数据需要满足的条件:所有元数据的总访问请求数量之和需要约等于$\frac{x-y}{2}$,并将选择的元数据迁移到节点 p 上.

由于元数据规模很小,在 1PB 数据量的文件系统中,理论上元数据量不超过 2 GB,分在每个节点上的元数据较少,所以复制过程并不会对系统性能造成重大影响.

4.2.2.2 一致性问题

副本策略实现了 I/O 分流,使原本属于节点 n 的一部分流量导向节点 a,降低了 n 的压力.然而随着时间的推移,a 和 n 上保存的子目录副本将变得非常不一致.

我们了解到,当且仅当涉及到 create_file(创建文件)、delete_file(删除文件)等操作时,系统才会出现不一致.针对这个特点,这里采用"更新转移"策略:只要涉及创建文件等操作,即在分配表中生成一条新的纪录导向另外的节点.当客户端打开刚刚创建的文件,只需查找目录表,找到这一条目即可.实践证明此方案很好地解决了不一致问题.

4.2.3 修改分配表条目

目录分配表记录的是子目录树与节点的对应关系,每次负载均衡机制执行完成,都要将修改反馈到分配表条目上.执行完目录树复制后,目录分配表中需要添加条目来记录副本与节点的对应关系.表 2 为执行均衡策略后的目录分配表,其中加深的 2 行对应相同的子树,但是对应不同的节点.

Table 2 Partition Table of Directory Tree

表 2 目录树分配表

Subtree Path of Directory	MDS
/root/bin	MDS_m
/root/etc	$\boldsymbol{MDS_n}$
/home/jian	MDS_p
/root/etc	$\boldsymbol{MDS_a}$
/home/liang	MDS_b

5 元数据代理的系统实现

系统由 3 个组件构成:客户端、元数据服务器和代理服务器.元数据服务器(MDS)存储每个文件/目录的关键元数据,主要功能包括管理文件系统名字空间等;每个代理服务器(proxy)维护整个文件系统的部分目录分配表,实现元数据服务器集群动态负载均衡.

代理服务器由多个节点构成,从 1 开始编号,我们假设共有 N 个节点.首先将所有元数据服务器节点平均分成 N 个组,编号分别对应 1,2,…,N,每个组里面的元数据服务器分别由对应编号的代理节点负责管理.代理节点与元数据服务器之间的对应关系如图 6 所示:

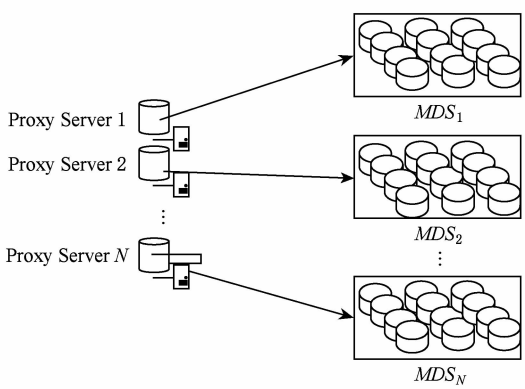

Fig. 6 The architecture of system

图 6 系统实现结构

5.1 代理与客户端的映射

我们对文件名进行了扩展.文件名用一个 40 b 的存储空间保存,如图 7 所示,其中前 8 b 用于标识文件所对应的代理节点路径,通过对文件名采用逻辑运算的方法获取前 8 b 的值,即获取到文件所对应的代理节点编号;后 32 b 保存文件的目录路径,用于保存文件的"真实"名称.

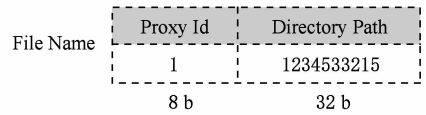

Fig. 7 Construction of file name

图 7 文件名组成

图 8 所示为文件操作的总体流程. 首先由客户端通过逻辑操作,获取文件名的前 8 b 的值(即为代理节点的编号);之后将此文件操作命令转发给对应编号的代理服务器,代理服务器再根据逻辑操作获取文件名的后 32 b 作为目录路径. 根据目录路径查找自己的目录表,然后根据目录表条目通过 Hash 的方法找到相对应的元数据服务器,转发到相对应的元数据服务器上进行操作.

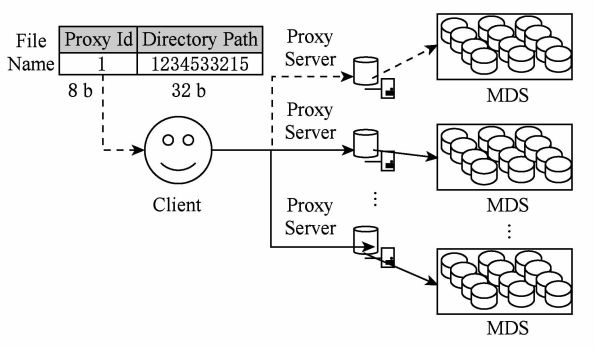

Fig. 8 Mapping relations

图 8 映射关系

5.2 代理服务器操作流程

Client↔Proxy:代理采用线程池技术处理来自客户端的请求. 每个代理服务器有一个连接队列,用来维护来自客户端的请求包. 线程池中的线程相互竞争,到连接队列里取包进行处理. 引入线程池机制,能极大提高系统处理并发的能力.

Proxy↔MDS:代理为每个 MDS 维护一个转发队列,用来保存分组结果;代理服务器的转发器实时监控每个转发队列,一旦发现某个队列填满,就将此队列中的请求打包,并发给相应 MDS.

Proxy:负载均衡器采用后台运行模式,实时接收来自 MDS 端的心跳信息,定时运行负载均衡机制,实现动态负载均衡.

具体包含如下步骤:

1) 应用程序调用客户端 API 接口,发出文件 open 操作请求;

2) 请求通过网络传送给对应编号的代理服务器端;

3) 代理服务器负责接收来自客户端的文件请求,并将收集到的请求根据目录分配表进行分组,并将每个装满的分组打包成请求包,对应元数据服务器的处理包;

4) 代理服务器上的转发器选择将各请求包转发给对应元数据服务器进行处理;

5) 元数据服务器将请求包"解包"成相应文件请求,对每个请求执行相应的操作;

6) 元数据服务器将处理结果分发给相应客户端.

系统采用 I/O 多路复用技术,利用 epoll 库实现异步事件处理机制. 客户端实现了 write_file 和 read_file 接口,应用程序调用这 2 个接口,和文件系统交互. 其中 write_file 实现将 buffer 缓冲区的数据写入到系统中;read_file 实现将文件系统数据写入到 buffer 缓冲区中.

6 性能测试及扩展性评估

我们通过一系列测试数据来评估原型系统,以展现其在性能、可靠性以及扩展性方面的特征. 原型系统包含客户端、MDS 以及代理服务器 3 个部分. MDS 和代理服务器有自己单独的硬件运行环境. 我们选择了一些不涉及数据块的文件操作,这些操作在运行过程中会产生大量元数据相关操作,而不会有很多数据块相关操作,从而模拟实际生产环境中大规模 I/O 请求的场景,来验证系统承受压力的情况.

6.1 硬件环境

测试所使用的服务器配置情况如表 3 所示:

Table 3 Server Configuration

表 3 软件环境配置

Usage	Kernel Version	Compiler
Client	Linux kernel 3.10	gcc (GCC) 4.8.5
Data Server	Linux kernel 3.10	gcc (GCC) 4.8.5
Proxy Server	Linux kernel 3.10	gcc (GCC) 4.8.5

主机的型号与配置情况如表 4 所示:

Table 4 Hardware Configuration

表 4 硬件环境描述

Usage	CPU Type	Memory Size	Network Interface Card	Hark Disk
Client	Intel®			Seagate
MDS	Xeon® CPU	64 GB	1 GB	2TB 7200 r/s
Proxy Server	E5 2620 V3			Cache:128 MB
Network Configuration	Gigabit Network Switches			

6.2 元数据更新延迟

首先考虑元数据更新所带来的延迟问题.客户端在某一时刻产生大量文件和目录,这些请求提交给代理,由代理进行预处理后转发给相应元数据服务器.图 9 为随备份副本数量的变化,元数据更新操作延迟的变化曲线.

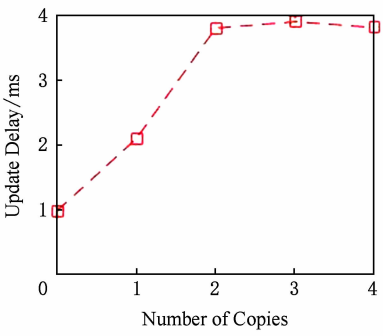

Fig. 9 Delay of metadata update

图 9 元数据更新延迟

元数据读请求比较复杂一些,图 10 为客户端嵌套访问 10000 个目录,对每个目录执行 readdir 操作和 stat 操作[15].我们统计了在每个目录下分别包含 10 个文件和 1 个文件时,这 2 种操作所消耗的累积时间.对实验结果分析可知,随着目录变大,MDS 消耗更多时间来处理 ls 命令操作,其中 stat 操作消耗的累积时间加速增长,readdir 则基本维持不变,甚至出现时间减少的情况.

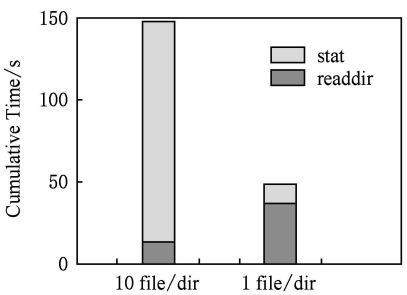

Fig. 10 Cumulative time of ls command

图 10 ls 命令累积时间

然而,增加代理会延长 I/O 路径,会对元数据访问带来额外的开销.

6.3 扩展性评估

我们使用包含 24 个节点的 Linux 集群来评估元数据服务器的可扩展性能.图 11 表示随着集群规模的扩展,单节点 IOPS 的变化情况曲线图.以单节点时候的吞吐率为基准画一条水平线,这条线就代表了理想情况下的线性扩展,即随着集群规模的变大,单节点吞吐量保持不变.

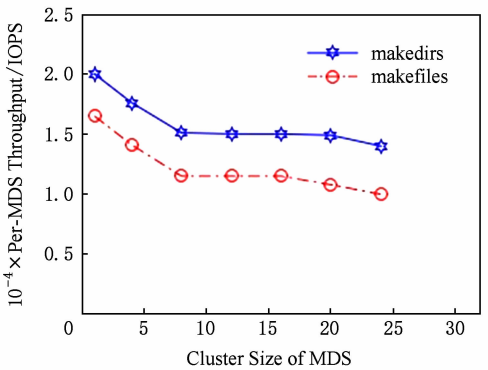

Fig. 11 Per-MDS IOPS along with cluster's scale

图 11 随集群规模扩展单节点 IOPS 的变化情况

如图 11 所示,MDS 集群节点平均吞吐量从单个节点时的 20 000 IOPS 降到了 24 个节点时的 13 800 IOPS.当节点规模 24 时,单节点下降到理想情况的 69%,以上结果验证了系统具有较好的扩展性.

7 总 结

分布式存储系统可以提供海量信息的存储以及设备级的安全性,广泛应用于科学计算以及数据密集型应用.访问数据之前必须要访问元数据,元数据访问很容易成为性能瓶颈[16].因而,合理的元数据管理,对于提高整个系统的 I/O 性能具有重要的意义.如何在元数据服务器之间均衡地分布元数据负载,是元数据管理的关键技术之一.本文设计了一种分布式的元数据负载均衡机制,可以按照服务器的性能来分布负载,不仅能够自适应负载的变化,而且能够自适应服务器规模的变化.在服务器增加和删除时,动态地均衡负载.

参 考 文 献

[1] TOP500. org. Tianhe-2 (milkyway-2) [EB/OL]. 2015 [2016-05-27]. http://www. top500. org/system/177999

[2] TOP500. org. Introducing TITAN: Advancing the era of accelerated computing [EB/OL]. 2012 [2016-05-27]. https://www. olcf. ornl. gov/titan/

[3] TOP500. org. Sequoia-blue gene [EB/OL]. 2015 [2016-05-27]. http://www. top500. org/system/177556

[4] Cluster File System, Inc. Lustre: A scalable, high-performance file system [R]. New York: Cluster File System, Inc, 2012

[5] Zhou Enqiang, Zhang Wei, Dong Yong, et al. Research on optimization towards hybrid and hierarchy storage architecture [J]. Journal of National University of Defense Technology, 2015, 37(1): 47-52 (in Chinese)

(周恩强, 张伟, 董勇, 等. 面向分层混合存储架构的协同式突发缓冲技术[J]. 国防科技大学学报, 2015, 37(1): 47-52)

[6] Wang Rong. Build high availability and high performance GPFS cluster [EB/OL]. [2016-05-27]. https://www.ibm.com/developerworks/cn/aix/library/au-gpfsplan/ (in Chinese)
(王荣. 构建高可用、高性能的 GPFS 集群[EB/OL]. [2016-05-27]. https://www.ibm.com/developerworks/cn/aix/library/au-gpfsplan/)

[7] Weil S A, Brandt S A, Miller E L, et al. Crush: Controlled, scalable, decentralized placement of replicated data [C] // Proc of the 2006 ACM/IEEE Conf on Supercomputing (SC'06). New York: ACM, 2006: 1232-1241

[8] Ghemawat S, Gobioff H, Leung S T. The Google file system [C] //Proc of the 19th ACM Conf on Operating Systems Principles (SOSP'03). New York: ACM, 2003: 32-43

[9] Information Sciences Institute. Internet protocol DARPA Internet program protocol specification [S]. Reston, Virginia: Internet Society (ISOC), 1981

[10] Liu Zhong, Zhou Xingming. A metadata management method based on directory path [J]. Journal of Software, 2007, 18(2): 236-245 (in Chinese)
(刘仲, 周兴铭. 基于目录路径的元数据管理方法[J]. 软件学报, 2007, 18(2): 236-245)

[11] Wu Changxun, Burns R. Handling heterogeneity in shared-disk file systems [C] //Proc of the 2003 ACM/IEEE Conf on Super Computing (SC 2003). Washington: IEEE Computer Society, 2003: 45-55

[12] Chen Tao, Xiao Nong, Liu Fang. Adaptive metadata load balancing for object storage systems [J]. Journal of Software, 2013, 27(2): 331-342 (in Chinese)
(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[J]. 软件学报, 2013, 27(2): 331-342)

[13] Chen Tao, Xiao Nong, Liu Fang, et al. Clustering-based and consistent hashing-aware data placement algorithm [J]. Journal of Software, 2010, 21(12): 3175-3185 (in Chinese)
(陈涛, 肖侬, 刘芳, 等. 基于聚类和一致 Hash 的数据布局算法[J]. 软件学报, 2010, 21(12): 3175-3185)

[14] Dong Yong, Chen Juan, Tang Tao. Power measurements and analyses of massive object storage system [C] //Proc of IEEE Conf on Computer and Information Technology. Washington: IEEE Computer Society, 2010: 21-24

[15] Weil S A, Brandt S A, Miller E L, et al. Ceph: A scalable, high-performance distributed file system [C] //Proc of the 7th Conf on Operating Systems Design and Implementation (OSDI'06). Berkeley, CA: USENIX, 2006: 307-320

[16] Guo Chengcheng, Yan Puliu. A dynamic load-balancing algorithm for heterogeneous Web server cluster [J]. Chinese Journal of Computers, 2005, 28(2): 179-184 (in Chinese)
(郭成城, 晏蒲柳. 一种异构 Web 服务器集群动态负载均衡算法[J]. 计算机学报, 2005, 28(2): 179-184)

Yi Jianliang, born in 1990. Received his bachelor degree of software engineering from Northeastern University in 2014 and master degree in computer science from the National University of Defense Technology in 2017. His current research interests include distributed file system.

Chen Zhiguang, born in 1984. PhD candidate in the National University of Defense Technology. His current research interests include file system and solid state storage system (zhiguang. chen@nscc-gz. cn).

Xiao Nong, born in 1969. PhD, professor and PhD supervisor. Senior member of CCF. His main research interests include high-performance computing and massive storage (xiao-n@vip. sina. com).

Lu Yutong, born in 1969. PhD and professor. Her main research interests include parallel and distributed computing, fault tolerance in HPC (ytlu@nudt. edu. cn).