分类号 TP3 密级

UDC 编号

中国科学院研究生院
硕士学位论文

基于对象的存储系统中元数据管理算法研究

杨林

指导教师 杜晓黎 研究员

中国科学院计算技术研究所

申请学位级别 工学硕士 学科专业名称 计算机应用技术

论文提交日期 2011 年 4 月 论文答辩日期 2011 年 5 月

培养单位 中国科学院计算技术研究所

学位授予单位 中国科学院研究生院

答辩委员会主席

声　　明

我声明本论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，本论文中不包含其他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示了谢意。

作者签名：　　日期：

论文版权使用授权书

本人授权中国科学院计算技术研究所可以保留并向国家有关部门或机构送交本论文的复印件和电子文档，允许本论文被查阅和借阅，可以将本论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编本论文。

（保密论文在解密后适用本授权书。）

作者签名：　　导师签名：　　日期：

摘 要

面对全球范围内迅猛增长的数据存储需求，基于对象的存储技术(OBS)应运而生，相对于 NAS 和 SAN 而言，基于对象的存储系统由于其可伸缩性、低成本、跨平台、易管理等特性，逐渐成为海量存储系统的一个最佳选择。如 Google 公司的 GFS 作为 Google 的核心技术，利用数以万计的普通商用服务器，为 Google 的各种服务提供了强大的数据存储能力；Cluster File System 公司的 Lustre，支持上千个存储节点，上万个客户端，PB 级存储容量，100GB/s 的传输带宽，为高性能计算提供了强有力的支持。

在基于对象的存储系统中，元数据（目录、文件大小、访问时间等信息）与数据存储相分离，文件数据保存在大量的对象存储服务器（OSD）中，元数据则由专门的元数据服务器处理。

元数据请求在文件系统中所占的比例高达 50%-80%，随着文件存储规模从 TB 级别走向 PB 级别甚至 EB 级别，在 GFS 和 Lustre 中所使用的单一的元数据服务器设计所暴露的问题也越来越严重，如单点失效，性能瓶颈，可靠性差，文件系统规模受限等。

本文针对基于对象的存储系统中单一元数据服务器带来的问题，结合联想网盘的实际存储需求，设计实现了一个支持多元数据服务器的分布式文件系统，能够利用元数据服务器集群，避免元数据服务器成为系统单点，通过将文件系统的元数据请求分布到多个元数据服务器，提供更好的元数据处理性能，同时保证元数据的冗余。本文的主要挑战在于：

1）在元数据服务器集群中，如何合理的分布和备份元数据，能够提供较好的性能，较好的扩展性和稳定性。

2）如何均衡系统中所有元数据服务器的负载，在元数据迁移中，如何防止过度迁移导致系统抖动，以及在元数据迁移后及时地更新客户端缓存状态。

本文是为联想网盘下一代存储系统进行的预研项目，利用以上研究成果，我们已经实现了一个文件系统原型，相对于我们的原有系统，其元数据处理能力得到了很大的提高。

关键词：云存储，对象存储技术，元数据服务器集群，动态子树划分，元数据迁移

Research on Metadata Management for Object Based Storage System

Yang Lin (Computer Application Engineering)

Directed By Du Xiaoli

With the increasing storage need, Object Based Storage System is already the best choose of Cloud Storage System due to it's scalability, usability, low cost, cross-platform, and easy to manage features.

An object-based file system splits file metadata (such as the filename, its size and access times) from file data and stores them on metadata servers, the file data is split into so-called objects and stored on object storage servers. The file system client employs metadata and object storage servers to present a full file system abstraction to the users. Operations about the metadata will take 50% to 80% percent of all file system operations, so the metadata management is crucial to the OBS architecture. The storage System is becoming bigger and bigger today, the architecture used in GFS and Lustre which has only one MDS can not afford to the increasing needs for the scalibility and capability. The single metadata server is the single point of failure in whole system, and its performance, scalability and capibility is limited.

Our project focuses on the metadata management of object based storage system, and we will consider our need in Lenovodata online storage system, the key point of this paper is :

1) The method for metadata distribution and backup in many metadata servers
2) How to dynamic balance the load of all metadata servers. We need to avoid too much migration operation, and update the cache information in client when metadata is migrated.

Our project is an research project in Lenovo, we designed and implemented a prototype to distribute metadata over all metadata server in a cluster, its metadata performance is much better then the original system.

Keywords: Object Based Storage，Metadata Cluster，Dynamic Subtree Partition，Metadata Migration

目 录

图目录

表目录

第一章 引 言

在云计算迅速发展的今天，用户更趋向于把数据存储在云端，云端的数据正在以前所未有的速度增长，YouTube 上每天上传 6500 段视频，每个月增加大约 20TB 存储需求；Google 每天处理的数据量超过 20PB(2008 年数据)。由此带来的存储需求极大地挑战着云端的存储架构，据 Cartner 在 2010 年的研究表明，对于大型企业来说，数据增长是其基础架构面临的最大挑战。

面对日益增加的数据量，云端使用的存储系统正在从 TB 级别走向 PB 级别甚至 EB 级别，云存储的强大需求要求存储系统具有大容量，高并发，易扩展，容易管理等特性，现有的网络存储系统比如 NAS 和 SAN 显然无法满足这样的数据容量和访问带宽，也不能及时有效地扩展到 PB 级别。

基于对象的存储系统(Object Based Storage, OBS[1])应运而生，它具有低成本、可扩展、高容量 、高性能、异构，易于管理等特性，基于对象的存储能够集合集群中数以万计的存储服务器，提供 PB 级别的存储容量和很高的性能；由于 OBS 使用商用服务器和 TCP/IP 网络构建，而不需要光纤，磁盘阵列等昂贵的设备，使得它能够保持很低的成本；OBS 设计中充分考虑到商用服务器的异构性，对硬件设备没有严格的要求，任何一台提供对象接口的对象存储服务器，都可以很简单的加入存储集群中，使得存储系统具有很好的扩展性；此外，OBS 中，数据通常以 Raid 或冗余的方式存储，能够保证系统具有很好的容错性，可以保证系统在部分硬件损坏的情况下有效对外提供服务。

这些特性完全契合了云存储的各种需求，因此基于对象的存储系统是新一代集群存储的最佳选择，在本章中，我们将简要介绍一下本课题的研究背景，引出本课题的研究目标和研究内容。

1.1 应用背景

1.1.1 云存储介绍

云存储是在云计算(cloud computing)概念上延伸和发展出来的一个新概念，是指通过集群应用、网格技术或分布式文件系统等技术，将网络中大量同构或异构的存储设备集合起来，共同对外提供数据存储功能的系统。云存储和云计算一样，具有对用户端的设备要求最低、方便共享、按使用付费等一些特征。

用户使用云存储后，终端将不再需要巨大的存储空间，它们将仅仅作为访问云端数据的一个设备。云存储提供了可靠、安全的数据存储中服务，用户不用再担心数据丢失、病毒入侵等麻烦。Google Docs 是云存储最典型的应用，利用 Google Docs，用户文件不再保存在用户的电脑上，而是保存在 Google 的存储云中，无论用户在什么地方，只需要一个浏览器登陆 Google Docs，就可以访问到自己的文件，这不仅可以防止用户电脑意外

丢失导致的数据泄露，还解决了用户在多台电脑上管理各种文件版本的困难，此外，利用云端的搜索技术，用户可以很方便地找到自己曾经记录的文档。

IDC 调查数据显示：到 2013 年，云存储服务的增长率预计将超过所有其他 IT 云服务。在未来四年内，云服务的市场规模将从现在的 174 亿美元增长到 442 亿美元，其中，云存储的市场比例将从目前的 9%增长到 14%，也就是说云存储的市场规模将接近 62 亿美元。

云存储的发展依赖于以下一些技术：

- **集群技术和分布式文件系统**：云存储中最关键的问题就是大量文件的存储问题，目前云存储架构中，在数据中心通常有数以万计的服务器，通过集群方式组合，通过一个在所有存储服务器上分布式运行的分布式文件系统，对外提供统一的、高性能的存储服务，这就依赖于分布式文件系统技术的发展。
- **网络接入技术：**在云存储中，所有数据都存放在云端，对数据的存取只能依赖于用户和服务器之间的网络连接，而现在的ADSL、DDN等宽带接入技术只能提供4Mbps的网络带宽，这对于文档型的云存储已经足够，但是对于多媒体数据则显得力不从心。只有用户端与服务器端都能提供较大的带宽，才能满足用户在云端存放各种数据的需求，使用户真正享受到云存储提供的便捷服务。
- **CDN(Content Delivery Networ)技术或跨机房调度技术：**在当前国内和国际的互联网情况下，各个ISP相对独立，跨ISP或跨国通讯仅能提供非常有限的带宽，云存储提供商只能利用CDN技术或跨机房调度技术，在多个国家，多个ISP都部署相应的存储服务器，才能满足用户快速存取文件的需求，如Google在全球各地都设有大型数据中心，通过在这些数据中心之间的数据调度、备份等，使得用户不管在哪里，都能以较快的速度访问Google提供的服务。
- **数据安全技术：**对于存放在云端的数据，必须保证数据不被人为或者意外的损坏而泄露或更改。云存储服务商可以通过身份认证、入侵检测、数据加密、访问认证、权限控制、数据备份等手段来保证数据的安全性。

1.1.2 云存储产品介绍

云存储目前已有很多产品，面向开发者的有亚马逊提供的 S3、Google 的 Google Storage，面向用户的有 Dropbox、Box.net、联想网盘等。

2006 年 3 月，亚马逊发布了简单存储服务(Simple Storage Service，S3) [2]，它按照每月租金的形式提供服务付费，用户需要为网络存储空间和相应的网络流量进行付费。亚马逊网络服务平台使用 REST (Representational State Transfer)和简单对象访问协议(Simple Object Access Protocol，SOAP)等标准接口，开发者可以通过这些接口访问到相应的存储服务。

Google 在 2010 年 5 月发布的 Google Storage [3] 也是针对开发者推出的一个云存储服务，它的作用类似于 CDN，是一个构建在 Google 的存储和网络设备上的 RESTful 云服务，开发者可以非常容易的使用 REST 风格的 API 存取 Google Storage 中的数

据，这些数据将保存在美国的若干机房，十分快速可靠。Google Storage 支持数百 G 大小的对象。开发者可以通过 web 界面或者 gsulti 这个开源的命令行工具来管理他们的存储内容。

Dropbox[4]是 Dropbox 公司运行的在线存储服务，它的特点是能够实现多个 PC 之间的数据同步和共享。安装 Dropbox 客户端后，把任意文件放入指定文件夹，就会被同步到云端，以及该用户的其他装有 Dropbox 客户端的计算机中。Dropbox 文件夹中的文件可以方便的与其他用户分享。Dropbox 提供免费和收费服务，提供网页访问模式和客户端访问模式。目前 Dropbox 在中国大陆地区已经无法访问。

Box.net[5]是一个网络硬盘服务网站，它利用 AJAX 技术构建的操作界面，清新简单而且非常容易上手，免费注册后就可以得到 1G 的云存储空间。

国内针对开发者的云存储产品还处于空白，针对用户的存储产品有联想网盘，金山公司的金山快盘，华为的 Dbank 等。

1.1.2 联想网盘介绍

自 2007 年开始稳定运行的联想企业网盘，在几近“零宣传”的情况下已经拥有了二十多个行业的数百家企业用户，为他们提供了安全、稳定、快速高效且性价比高的全面解决方案。在前期的稳健创新和研发基础上，联想企业网盘在 2011 年将大规模推向全国市场 [6] 。

联想网盘提供了超大文件传输（单个文件超过 1GB），断点续传，跨数据中心调度，跨国调度，协同办公、外链功能、多重备份等功能，拥有大规模数据分布式存储技术、远程容灾备份和数据同步技术、高负载并发的传输技术。在安全性方面，联想网盘在文件的存储和传输中都使用了加密技术，联想网盘服务已经稳定运营三年，达到 99.99%的安全运营和服务成功率。

联想网盘旗下拥有多个产品：联想企业网盘、联想商务网盘、桌面网盘、备份网盘、云杀毒网盘、手机网盘，乐空间等，有效满足了不同客户的特色需求。企业网盘和商务网盘已经正式接受客户订单；桌面网盘、备份网盘、云查杀网盘、手机网盘处于小规模试用阶段，上线后会扩大联想网盘应用领域并丰富其功能，乐空间已随联想多个型号 ideapad 发布上市。在云计算开始走下云端的 2011 年，作为联想云存储战略基础与核心的联想企业网盘将有突飞猛进的发展。

1.2 本文面对的问题和研究目标

联想网盘的存储规模不断扩大，文件系统负载快速增加，其存储架构经过几次重大变迁：

联想网盘在初期使用 Lustre[7]作为文件系统，但是由于 Lustre 不提供数据冗余机制，Failover 也仅仅实现服务器节点的冗余，如果磁盘发生损坏，整个 Lustre 空间将损

坏（Lustre 官方 manual 也声明，Lustre 的存储必须基于 Raid 阵列等硬件冗余才能保证数据的完整）。

以联想网盘多机房部署为契机，我们决定放弃 Lustre，采用新的文件系统，经过对 NFS、Ceph、mooseFS、HDFS、PVFS 等文件系统的调研，由于 mooseFS[8]部署简单，容错性好，支持动态扩容，并且具有 POSIX 接口，方便原有系统的移植，我们最终选定用 mooseFS 替换 Lustre。

我们原计划将所有原来存储在 Lustre 上的文件都迁移到 mooseFS 中，但由于 mooseFS 使用单一元数据服务器结构，并且将所有文件元数据保存在内存中，文件系统的规模受到极大限制，例如一台 16GB 内存的文件系统，假设所有内存都用于存放元数据的，大约只能存储 5 千万个文件。这只能在短期内解决联想网盘的存储需求。因此，我们急需一个大容量、高性能、能保存大量元数据并且对所有数据有冗余备份的分布式文件系统。

在当前的很多 SAN 文件系统中，都采用元数据服务器集群技术应对大量的元数据请求。元数据集群将使用多台元数据服务器构成集群，通过一定的划分方法，将整个文件系统的元数据负载在多个元数据服务器上均匀地分布，为元数据管理提供较好的扩展性和很高的性能。如中科院计算所研发的 DCFS2、COSMOS，蓝鲸文件系统等。

然而，由于元数据服务器集群实现复杂，在当前各个基于对象文件系统中，GFS、Lustre、HDFS、mooseFS 等都是单一元数据服务器结构或元数据主从备份结构，AFS[9]、Coda[10]、Sprite[11]具有静态元数据集群能力，不能提供有效的扩展性，仅有的具有元数据集群能力的 Ceph[12]文件系统，使用动态子树划分实现元数据的划分和负载均衡，然而 Ceph 目前仍然处于研究阶段，尚不能使用在生产环境中（官方文档中称 Ceph 处于重度开发中，建议仅用于 Benchmark）。

当前，联想网盘存储规模约 300TB，整体使用率约 40%，文件平均大小为 200KB 左右，文件个数约 3 亿，每日新增文件 20GB-100GB，联想网盘应用的特点是：文件数目大(意味着文件元数据多)，文件偏小，此外，由于联想网盘提供的是在线存储业务，必须有数据的冗余备份机制。

我们预期文件系统将会在今后一段时间内成为系统瓶颈， 制约联想网盘的发展，因此，本文的项目背景正是实现一个有较高的元数据处理性能，提供元数据动态扩展和冗余功能的文件系统。在该文件系统中，我们将采用基于对象的存储架构，并实现动态元数据服务器集群。

本文的研究目标：研究基于对象的存储系统中元数据集群化管理算法，在总结前人工作的基础上，实现一种元数据集群化管理系统，该系统应该能够具有动态扩展能力，能够对元数据进行冗余存储和负载均衡。

1.3 本文的贡献

根据[13]的研究，元数据请求在文件系统中所占的比例高达 50%-80%，元数据请求的性能对集群存储系统的影响至关重要。

本文针对基于对象的存储系统中单一元数据服务器带来的问题，结合联想网盘的实际存储需求，设计实现了一个支持多元数据服务器的分布式文件系统，能够利用元数据服务器集群，避免元数据服务器成为系统单点，通过将文件系统的元数据请求负载分布到多个元数据服务器，提供了更好的元数据处理性能，同时保证元数据的冗余，本文的工作主要可以分为以下三个部分：

1. **设计了一种元数据集群动态管理方案：**为了使元数据集群具有较好的扩展性，具有动态扩容能力，就要求系统能够动态地加入或者去除元数据服务器节点，本文中，我们通过基于版本的集群管理实现了元数据节点的动态加入和删除。
2. **元数据动态分布：**元数据在 MDS 集群上的分布策略，是影响元数据服务器集群性能、可靠性和扩展性的主要因素，本文使用动态子树划分算法进行元数据的分布，期望提供较好的负载均衡能力，扩展能力和可靠性。此外，为了保证元数据的安全，应该对元数据使用某种形式的备份策略。
3. **元数据负载均衡：**我们应该将整个文件系统的负载平均分布在元数据服务器集群中，当一个新的 MDS 加入系统后，旧的 MDS 需要把一部分元数据迁移到新 MDS 上，才能让新的 MDS 承担一定的元数据负载。当某个目录子树成为访问热点时，需要将一部分元数据迁出，使得系统中所有 MDS 的负载相对均衡。

利用以上研究成果，我们基于 Fuse(Filesystem in Userspace 用户空间文件系统)实现了一个基本兼容 POSIX 的分布式文件系统，能够支持多元数据服务器集群，系统能够动态加入和删除元数据节点，能够有效地进行元数据负载均衡，从而对外提供理想的元数据处理能力。

1.4 论文的组织

论文从元数据服务器集群的研究背景和意义出发，在第一章中介绍了云存储背景和联想网盘中的应用需求。

在第二章中，介绍了分布式文件系统的发展，当前流行的基于对象存储系统的基本结构和相关系统，并分析了它们的优劣。

第三章是对我们实现的分布式文件系统的一个总体设计，其中描述了基于版本的元数据集群节点管理，包括节点的定位方法，节点加入和离开时的流程。

第四章详细描述了元数据的分布算法和备份策略，通过对目前已有的元数据分布算法的比较分析，我们选择动态子树划分方法来实现元数据在多个 MDS 之间的分布，我们介绍了其具体实现，以及元数据集群中元数据的备份策略。

第五章介绍系统使用的负载均衡策略，包括负载的计算和评估模型，迁移粒度的选择，元数据迁移关键算法的实现以及客户端缓存一致性的维护等问题，这是本文的重点。

第六章利用目前常用的文件系统的 benchmark 工具，对我们实现的文件系统在元数据处理性能，扩展性，负载均衡能力等方面进行了一系列测试。证明本文所使用的方法有效解决了单元数据服务器的种种局限性。

最后，对整个论文进行了总结，并给出了将来的研究方向。

第二章 基于对象的存储系统介绍

本章首先将简要介绍存储系统的历史和发展现状，从 SAN 和 NAS，到基于对象存储系统，介绍了基于对象存储系统的原理和优势所在，最后选取几个有代表性的对象存储系统进行介绍。

2.1 存储系统的历史发展

在基于对象的存储系统出现之前，网络存储主要有 NAS[14]和 SAN[15]两种架构。

NAS 采用 NFS[16]或 CIFS[17]命令集访问数据，他们都是文件级协议，NAS 直接通过 TCP/IP 实现互联，可扩展性好，价格便宜，用户易管理，例如目前在集群计算中应用较多的 NFS 文件系统。但由于 NAS 的协议开销高、带宽低、延迟大，不利于在高性能集群中应用。

SAN 采用 SCSI 的块级 I/O 命令集，而不是文件级命令，通常使用光纤或 GB 以太网互联，具有高带宽、低延迟的优势，主要用于高性能计算，但是 SAN 系统的价格较高，不具有异构性（必须采用一致的硬件），可扩展性较差。

基于对象的存储系统是在 NAS 和 SAN 的基础上发展起来的，结合了二者的设计优点，在 I/O 性能、可用性和可扩展性方面有结构优势。它采用元数据与数据传输分开处理的策略，能提供更大的容量，更高的性能和可靠性，并能方便的括容，代表了文件系统发展的方向，如 Google 研发的 Google File System[18]，Cluster File Systems 公司的 Lustre，Panasas 公司的 ActiveScale[19]文件系统等。

此外，在 Key-Value 数据库流行的今天，一类 Key-Value 存储系统也逐渐发展起来，它们只提供健/值形式的存储，放弃了目录树结构，从而能够获得很高的性能、容量和可扩展性，目前也广泛使用在互联网企业中。例如 Amazon 的 Dynamo[20]、Facebook 的 Cassandra [21]、Redis、Tokyo Tyrant 等，在国内有淘宝的 TFS，豆瓣的 BeansDB，人人网的 Nuclear 等，这些存储系统通常用于存储大量的小文件，如淘宝的 TFS 存储约 300 亿图片。

当前存储技术的另外两个研究热点是存储虚拟化和固态硬盘技术，存储虚拟化把各种不同的网络存储设备集中在一起，提供一组虚拟存储卷供主机使用，通过屏蔽下层的设备异构性，从而使存储系统方便扩展；固态硬盘是对硬盘的极大革新，基于永久性存储器(如闪存)或非永久性存储器（如 SDRAM），与硬盘不同，固态硬盘不再需要机械旋转结构，具有高速、低功耗、无噪音、抗震动、低热量的特点。

2.1.1 NAS （Net Attached Storage）

按照存储网络工业协会（SNIA）的定义：NAS 是可以直联到网络上向用户提供文件级服务的存储设备。NAS 通过远程文件访问协议(NFS、CIFS 等)提供网络间的文件共享服务，可以实现统一的名字空间，服务可以在线扩容，但是它的可靠性和和性能都比较难以控制。其架构如图 2.1 所示：

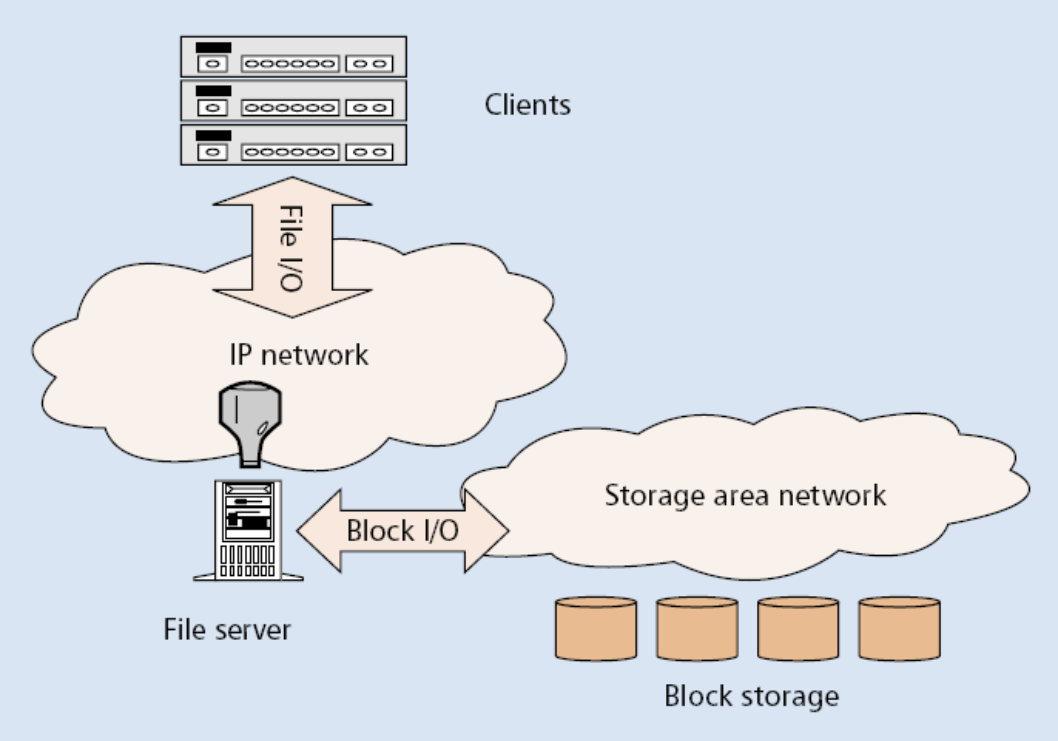


图 2.1 NAS 存储架构（来自[1]）

1985 年 Sun 公司最早在 Solaris 系统中发布了 NFS，并将其协议公开，到目前为止，NFS 已经被移植到了几乎所有主流的操作系统，IETF 为其发布了 RFC1094、RFC 1813、RFC 3010 等一系列协议规范，使 NFS 成为分布式文件系统事实上的标准。

SMB(Server Message Block[22])协议最先由 IBM 公司为 DOS 系统开发，从 windows 95 开始，微软开始支持 SMB 协议，Windows 2000 系统在 TCP 协议之上实现了 SMB 协议，SMB 协议用于 Windows 系统间的文件和打印机的共享，其的开源版本叫做 CIFS 协议(Common Internet File System，CIFS)，可以用于 windows 系统和 Linux 系统之间的文件共享。

NAS 具有如下优点：使用现成的 TCP/IP 网络，成本较低，方便安装和括容，使用简单；能提供异构平台（不同操作系统）下的文件共享，而且 NFS 和 CIFS 都有已经成为标准，使用它们，不需要开发专门的客户端协议。但是 NAS 有一个很大的缺点：由于所有的文件请求都要经过 NAS 服务器，因此当客户端数目或来自客户端的请求较多时，NAS 服务器将成为系统的瓶颈所在。此外，由于 NAS 的协议开销高、带宽低、延迟大，整体性能较差。

2.1.2 SAN （Storage area network ）

按照 SNIA 定义，SAN 是一种利用 FC（Fibre Channel，光纤通道）等高速互联协议连接起来的、可以在服务器和存储系统之间直接传送数据的网络，SAN 基于虚拟的共享磁盘(Shared-disk File System)，是目前集群文件系统的主流，它利用 Fibre Channel（FC SAN），GB 以太网（IP SAN）等协议将磁盘阵列直接暴露成为一个虚拟的磁盘，各个客户端将虚拟磁盘当作块设备操作。SAN 能提供很高的随机访问性能和数据聚合带宽，在高性能计算领域占有一席之地，另外 SAN 还常用于关键数据的存储，比如 Oracle 等关键业务企业级数据库，为了保证高可用性，通常使用主从的双机备份策略，它们共用同一个 SAN 存储，如果主服务器故障，则从服务器可以很快接手主服务器的工作，继续对外提供服务，由于数据存储是一致的，不涉及数据迁移等问题。其架构如图 2.2 所示：

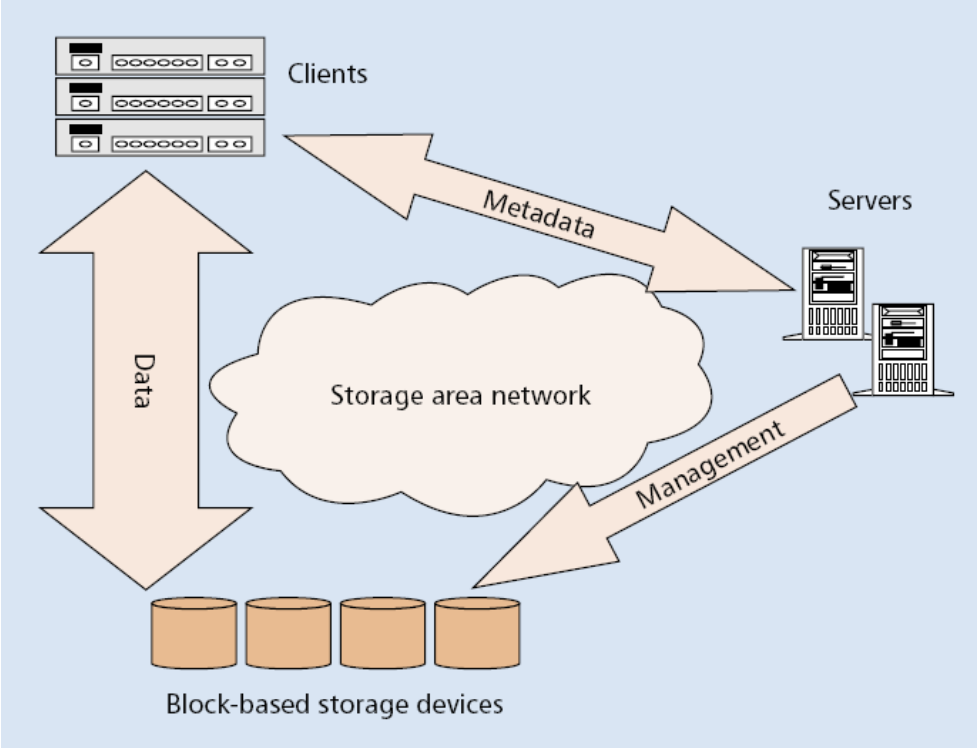


图 2.2 SAN 存储架构（来自[1]）

除了高性能之外，SAN 具有很高的可扩展性和可用性；由于存储设备被所有服务器共享，这些服务器中的任何一个宕机，都不会影响到系统的可用性，这对于数据库类的应用非常重要。

到 2010 年为止，SAN 在存储市场占有相当大的市场份额，如 IBM 公司的 General Parallel File System(GPFS) [23]， Red Hat 公司的 Global File System(GFS) [24]，EMC 公司的 Celerra HighRoad，Oracle 公司的 ACFS，惠普的 HP Cluster File System 等。

GPFS（General Parallel File System）是 IBM 推出的 SAN 架构文件系统，GPFS 从 1993 开始研发，1995 年发布，集群内的所有节点可以并行地访问所有共享磁盘，GPFS

的应用范围非常广泛，特别是在大型的高性能计算机群中，在 2002 年，GPFS 曾经占据 top10 中的 6 台。GPFS 采用扩展哈希(extensible hashing)技术来支持含有大量文件和子目录的大目录，提高文件的查找和检索效率。大量采用分布式锁解决系统中的并发访问和数据同步问题。GPFS 还有效地克服了系统中任意单个节点的失效、网络通信故障、磁盘失效等异常事件。此外，GPFS 支持在线动态添加、减少存储设备，然后在线重新平衡系统中的数据。这些特性在需要连续作业的高端应用中尤为重要。

SAN 的主要缺点是价格昂贵，这主要来自于 FC 设备的成本；SAN 不具备异构性，由于不同的制造商光纤通信协议有所不同，所以其 SAN 产品很难整合；另外，SAN 管理成本很高，需要经验丰富的、接受过专门训练的人员（通常由 EMC，Panasas 等公司提供后续维护工作），这大大增加了构建和维护费用。

2.1.3 IP-SAN

SAN 中使用的光纤网络直接导致其成本居高不下，随着用于千兆(GB)，万兆(10GB)以太网的出现，基于 IP 协议的 IP-SAN 实现方案成为替代传统 SAN 的更佳选择。IP-SAN 是指利用 GB 以太网的 TCP/IP 协议替换 FC 协议实现 SAN 方式的文件共享，它成本更低，而且可以解决 FC 传播距离短(10KM)的问题。IP-SAN 中，通常使用 iSCSI 协议实现存储设备之间的互联，此外还有 FCIP，iFCP 等协议。

蓝鲸分布式文件系统(BWFS)是由中国科学院计算所工程中心研制的网络存储系统，它是 IP-SAN 和 NAS 的结合体，BWFS 利用元数据集群提高系统元数据的处理能力，文件元数据到 MDS 的映射采用了动态映射策略（利用绑定服务器动态绑定），可以实现元数据服务器的负载均衡，易于扩展和管理。

Dawning Cluster File System (DCFS2) 是用在曙光超级服务器集群上的机群文件系统，曙光文件系统经历了 D2K-COSMOS、D3K-COSMOS、DCFS、DCFS2 等版本的发展，DCFS2 机群文件系统由国家智能中心开发，是基于 IP-SAN 的共享存储机群文件系统， 它采用将文件数据与文件系统元数据分开进行处理的策略，而且支持多个元数据服务器，使用一定的映射策略将元数据处理映射到元数据服务器。

2.1.4 对象存储系统

对象存储的思想起源于卡内基梅隆大学并行数据实验室(Parallel Data Lab，PDL)从 1995 年到 1999 年的“Network Attached Secure Disks ”(NASD)项目。该项目的目标是“使用商业存储设备构建高带宽、低延迟、安全、可扩展的存储系统”，图 2.3 是当时的架构图，其思想是吸取 NAS 和 SAN 的优点，一方面通过直接访问数据，提供 SAN 所具有的高性能，另一方面通过较高层次的协议，提供较好的异构性和扩展性。

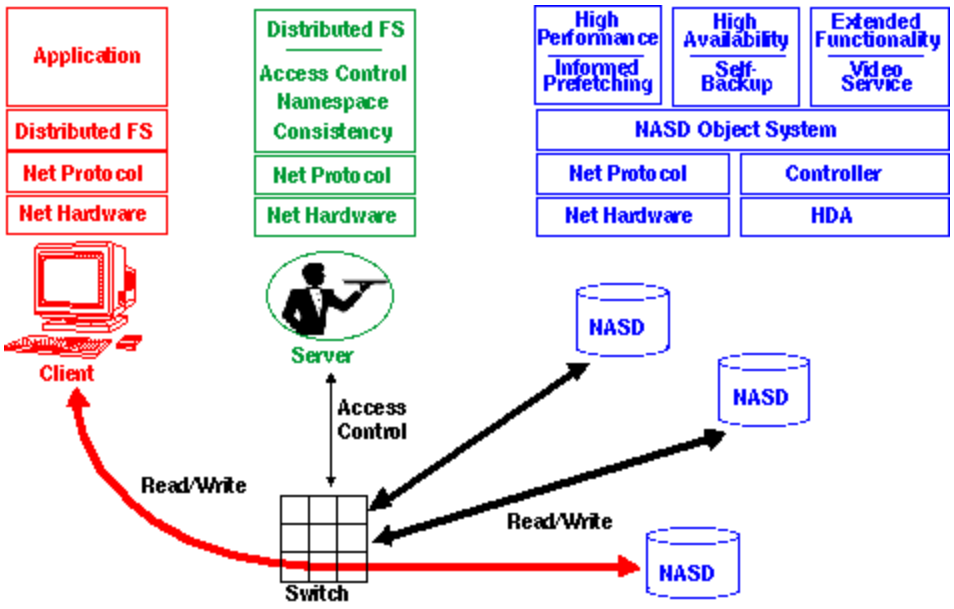


图 2.3 NASD 存储架构 (来自：http://www. pdl. cmu. edu/NASD/)

在对象存储系统中，“对象”是数据存储的基本单位，对象提供和传统的文件类似的访问方法，如 Open、Close、Create、Read、Write 等。传统文件系统中的文件被分解为一系列存储对象，存储于“对象存储设备(Object-based Storage Devices，OSD)”上，OSD 是一个具有独立智能的设备，有自己的处理器、内存、网络接口和磁盘，能够管理存储在自己身上的对象。

在基于对象的存储系统中，文件系统中的元数据管理功能和数据传输分离，通常由元数据服务器，对象存储设备和客户端三部分组成，其基本架构如图 2.4 所示：

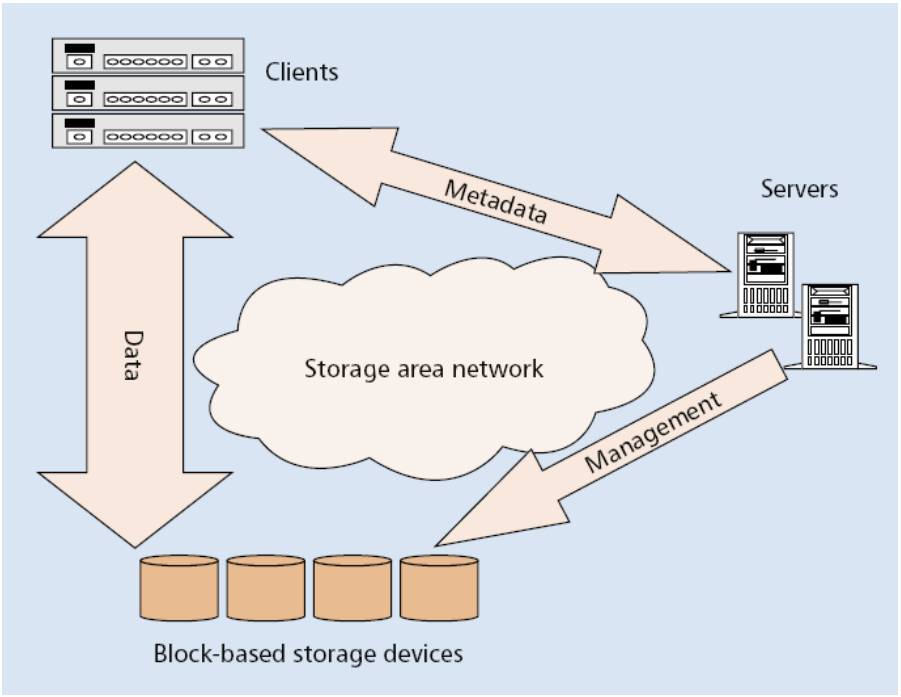


图 2.4 基于对象的存储架构 （来自[1]）

元数据服务器(Metadata Server，MDS)负责管理文件的名字空间，主要是文件和目录的组织关系，每个文件对应的对象 ID，每个对象的存放位置等信息。此外，元数据服务器需要记录文件属性信息，如文件的大小、属组、访问时间等。

对象存储服务器(Object based Storage Device，OSD) 的功能是存储文件的内容，提供磁盘管理。对象存储服务器有一定的智能（通常是一个挂载很多磁盘 Linux 服务器）通过暴露统一的对象存取接口，向客户端提供服务。

客户端(Client)将来自应用层的 I/O 请求包装成对 MDS 和 OSD 的请求，为用户提供访问文件的接口，供用户层应用使用。

对象存储系统有下面优势：

高性能：对象存储系统使用专门的元数据服务器处理元数据请求，不需要大规模使用像 NAS 那样的锁机制来防止元数据被并发写，此外，由于 Client 可以直接与对象存储服务器通信，可以实现很高的文件读写带宽。

可扩展性：在 OBS 中，所有 OSD 可以是异构的，只要它们提供相同的访问接口即可，可以很容易地通过添加对象存储服务器，实现系统括容。

易管理：系统容错性好，很少需要人工维护，括容也比较简单。

基于对象的架构能够提供很好性能和扩展性，极大的存储容量，代表着文件系统的发展方向，目前比较有代表性的基于对象的存储系统有 Panasas 公司的 PanFS，Cluster File Systems 公司的 Lustre，IBM 的 StorageTank，Google File System，MooseFS 等。下面我们将选择几个进行介绍。

2.2 典型的对象的存储系统介绍

2.2.2 Lustre

Cluster File System 公司开发的 Lustre 分布式文件系统，是最经典的对象存储系统，它于 2003 年 12 月发布了 1.0 版，被 Sun 和 Oracle 收购以后，在 2009 年 5 月发布 1.8 版，2010 年 8 月发布 2.0 版。截止到 2010 年 11 月，在全球 top30 高性能计算机中，有一半以上都部署 Lustre，包括比如 Bule Gene，Red Storm，以及部署在国家超算天津中心的天河一号(top 1)，用来进行天气预报的模拟，以及分子动力学模拟等高性能计算领域。

Lustre 支持上千个存储节点，上万个客户端，PB 级存储容量，100GB/s 的传输带宽。Lustre 的设计目标是大型集群中的科学计算，性能和带宽是它的优势，数据的安全性不是其主要目标，所以 Lustre 中数据只存一份，不支持数据的冗余，其数据完整性的保证依赖于下层的 Raid 阵列等硬件冗余技术。

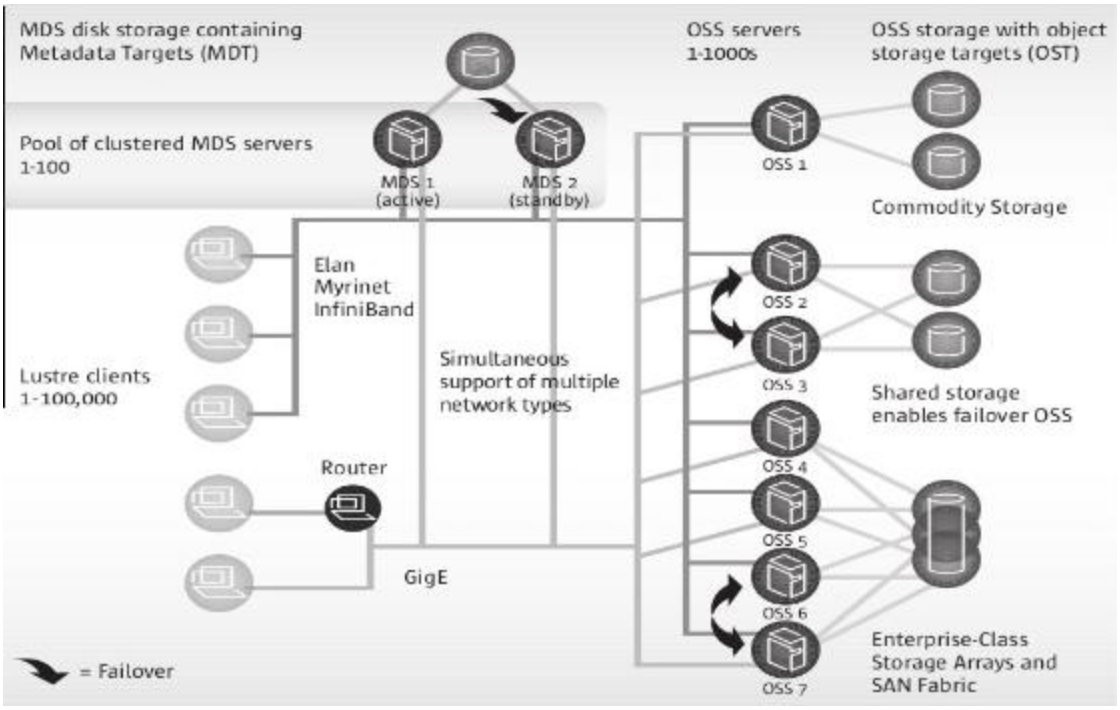


图 2.5 Lustre 系统结构(来自 Oracle 公司)

图 2.5 是 Lustre 的架构图，Lustre 集群组件包含了 MDS（元数据服务器）、MDT（元数据存储节点）、OSS（对象存储服务器）、OST（对象存储节点）、Client（客户端），以及连接这些组件的高速网络。

在元数据管理方面，Lustre 使用 MDT 存储元数据，MDS 提供元数据服务，MDT 只能有 1 个，不同 MDS 之间共享访问同一个 MDT。而 MDS 可以设置两个，它们之间采用了 Active-Standby 的容错机制，当其中一个 MDS 不能正常工作时，备份服务器能接管其服务，确保系统的正常运行。

Lustre 将 OSD 抽象成 OSS 和 OST，其中 OSS 负责提供 I/O 服务，接受并服务来自客户端的请求。一个 OSS 对应到后端的 2-8 个 OST。 OST 上的文件数据是以分条的形式保存的，文件的分条可以在一个 OSS 之中，也可以保存在多个 OSS 中。

Lustre 中 Client 通过 Linux 下的 VFS（虚拟文件系统）机制 mount 在目录层次中，完全兼容 POSIX，用户可以像操作本地文件系统一样操作它。

2.2.3 Google File System 和 HDFS

Google File System(GFS)构建于 2000 年，它使用廉价的通用商业部件构建系统，而不使用昂贵的磁盘阵列，光纤网络；在设计上，它与当时的其它文件系统截然不同，GFS 把出错作为常规情况，而不是当作异常来处理，把容错性作为系统的一个内置特性；由于 Google 最常见的数据处理模式是对大文件的顺序读写，GFS 普遍针对大文件的情况（针对的文件大小为 GB 级别）进行优化，并且不支持文件随机写，这极大地简化了其数据缓存的设计。为了保证系统的可靠性，可用性，通常所有文件都存放 3 份，

并且具有机架感知的功能，GFS 会尽量将数据存放在不同的机架上以在最大程度上避免机柜掉电带来的数据丢失。

在前面假设的情况下，Google 设计了如图 2.8 所示的结构：

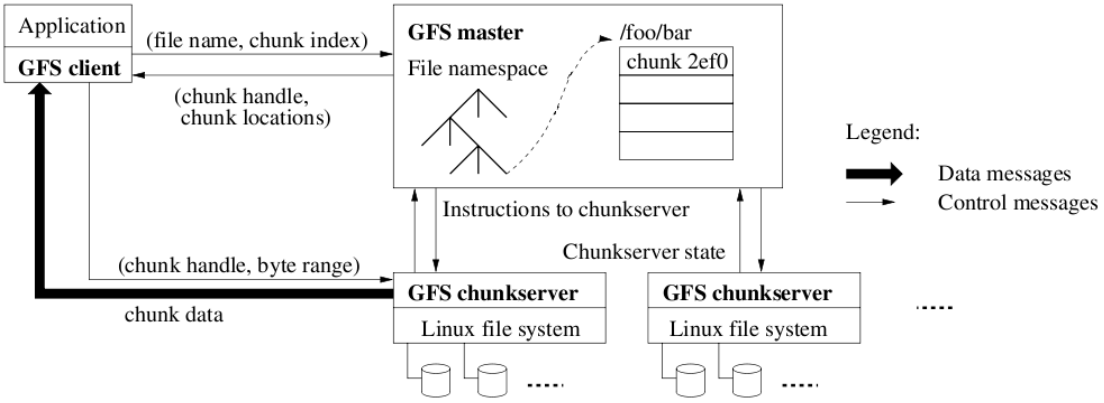


图 2.6 Google File System 系统结构(来自 Google 公司)

上图中，Master 相当于元数据服务器，管理名称空间，访问控制信息，文件到块的映射关系，以及块当前所在的位置等，同时提供垃圾回收，数据块的迁移调度等，GFS 使用的单一 Master 结构简化了设计，其元数据完全放在 Master 的内存中，提高了元数据存取的效率，此外，GFS 的客户端也会缓存一部分元数据，从而降低 Master 的压力。

Hadoop 是 GFS 的开源版本，使用 Java 开发，它起初是为开源搜索引擎 Nutch 开发，为 Nutch 提供大规模的网页存储功能，Hadoop 包括分布式文件系统 HDFS 和分布式计算框架 MapReduce，HDFS 的结构如图 2.7 所示：

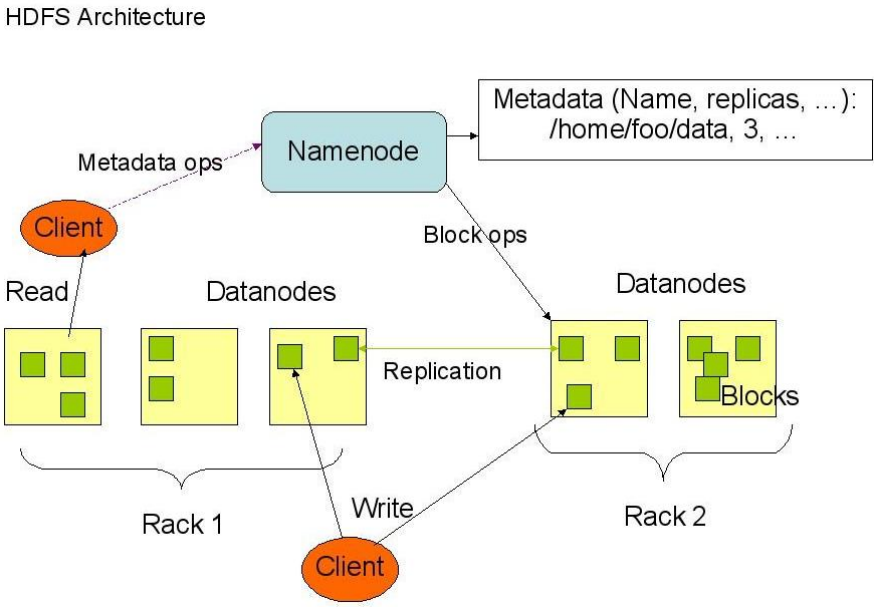


图 2.7 HDFS 系统结构(来自 http://www.apache.org/)

GFS 和 HDFS 都使用单一 Master 结构，在 HDFS 中，每个文件的元数据大小约 150 字节，一亿文件需要占用约 3GB 内存空间，这将是 GFS 和 HDFS 的一个最大限制；GFS 和 HDFS 都不支持通用 POSIX 标准，必须使用专门的 API 来进行文件的读取。此外，GFS 和 HDFS 都是针对大文件顺序读写设计，块大小设计为 64MB，在大量小文件的情况下，其数据读写取效率很低。

2.2.4 Ceph

Ceph 最初是 Sage Weil 在加州大学(University of California，Santa Cruz，UCSC)博士期间的研究课题，自 2007 年毕业之后，Sage 开始全职投入到 Ceph 开 发之中，使其能适用于生产环境。Ceph 的主要目标是设计成基于 POSIX 的没有单点故障的分布式文件系统，使数据能容错和无缝的复制。2010 年 3 月，Linus Torvalds 将 Ceph client 合并到内核 2.6.34 中。说明 Ceph 已经进入分布式文件系统的主流。

Ceph 的架构也是典型的 OBS 架构，如图 2.8 所示。在 Ceph 的对象存储系统设计很有特色，Ceph 设计了 RADOS (Reliable, Autonomic Distributed Object Store)作为 Ceph 的对象存储系统（相当于 OSD），RADOS 能够在动态变化和异构的存储设备机群之上提供一种稳定、可扩展、高性能的单一逻辑对象(Object)存储接口，能够实现节点的自适应和自管理。RADOS 是由大量 OSD 和少量 Monitor 组成，OSD 用于存储数据，Monitor 负责管理集群的 Cluster Map。在 RADOS 中，OSD 被分为一个个 PG (Placement Group)，相同 PG 里的对象会被系统分发到相同的 OSDs 集合中，数据备份在一个 OSD 集合中的各个 OSD 之间进行。

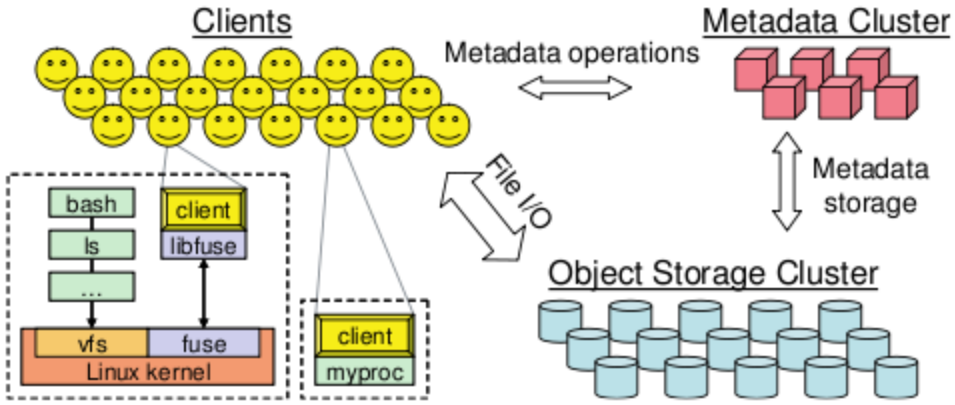


图 2.8 Ceph 架构结构(来自 Ceph)

Ceph 是最先提供动态元数据集群功能的开源文件系统，它使用动态子树划分算法，可以根据元数据服务器的负载动态调整元数据的分布策略，将负载过重的 MDS 上的元数据迁移到其它元数据服务器，此外，Ceph 还可以将访问热点复制多份，分布在多个 MDS 上，为多客户端并发读取同一个元数据的情况提供较好的性能。

早期版本的 Ceph 利用 Fuse 在用户空间实现，在很大程度上简化了其开发难度。但是今天，Ceph 已经被集成到 Linux 主线内核，在内核态下实现文件系统，可以减少

不必要的用户空间切换，使其更快速。然而，Ceph 的数据存储依赖于尚不成熟的 btrfs，其本身也非常不成熟，其官方网站上说明该文件系统只能用于 Benchmark 目的和研究目的，不建议部署到生产环境中。

2.2.5 MooseFS

MooseFS 采用和 Google File System 完全相同的设计，与 Google File System、HDFS 不同的是，它是一个 POSIX 兼容文件系统，可以直接挂载到 Linux 系统，像操作本地文件系统一样操作 MooseFS 中的文件。

MooseFS 使用 FUSE (Filesystem in User Space) 实现文件系统挂载， 能用于所有支持 Fuse 的操作系统（Linux，FreeBSD，Mac OS X 等），它还支持特殊的文件(块设备，字符设备和管道)，支持软链接，快照和回收站等，每个目录的冗余策略可配置，具有一个完善的监控界面。目前在国内互联网公司得到广泛使用，例如豆瓣网使用 MooseFS 存储大量图片，遨游使用 MooseFS 存储用户的收藏夹。

2.3 小结

由于基于对象的存储在 I/O 性能、可用性和可扩展性方面的优势。基于对象存储技术在过去几年中得到了巨大的发展，很多新的集群存储系统都采用基于对象的存储架构，但是，从目前情况看，基于对象的存储技术不太可能在短时间内完全取代传统的网络存储方式，以 NFS 为代表的 NAS 系统在小量文件共享方面以其简单方便而得到广泛使用，而 SAN 依然在高性能计算中占用一席之地。

目前 Google File System、MooseFs 都支持动态增加存储节点，但是由于它们只有一个元数据节点，扩展元数据服务器的时候必须扩充内存，并且需要系统重启，系统动态扩充的能力受到很大限制。

第三章 元数据服务器集群整体架构

本章将介绍我们使用的元数据集群管理方案，包括元数据服务器节点的加入和删除，根据联想网盘应用的特点所做出的一系列设计决策等。

3.1 总体设计

前面已经提到，联想网盘应用的特点是：文件数量大（意味着元数据多），文件偏小，文件的数据存储和元数据必须有冗余机制，整个文件系统的存储规模约300TB。针对目前联想网盘的存储规模及其特点，在我们设计的存储系统中，我们做出如下设计决策：

不使用 Chunk 切分：我们的文件系统中文件平均大小为 200kB，文件大小通常不会超过 1GB，因此，对于文件数据我们不会采用 GFS，mooseFS 中广泛使用的 Chunk 切分算法。

不考虑大目录：根据对联想网盘使用文件系统的模式进行分析，在每个目录下的子目录或文件不会超过 256(2^8)个，文件系统层次深度不多于 8 级（即最深的目录为 8 级）。因此，我们不会考虑针对大目录的优化，不会使用如 GPFS 中采用的扩展哈希(extensible hashing)技术，GIGA+中的目录分区等。

集中式集群管理：我们使用的服务器至少内存为 16GB，并且均支持 epool 机制，服务器之间使用千兆以太网连接，所以这些服务器对简单请求的处理能力大于 2 万次/秒，例如对于集群管理器来说，假设集群中的每个服务器每秒向集群管理器发送一个请求，集群管理器完全可以支撑 2 万个服务器。

对象存储直接使用 HTTP 接口：在联想网盘中，我们使用一个或多个传输服务器(Transfer Server，TS，通常为 nginx 模块)为用户提供上传/下载服务，用户上传一个文件时，传输服务器将该文件写入文件系统，写入通常为顺序写（由于客户端具有分片上传功能，也会有随机写需求，但随机写远少于顺序写）。用户下载文件时，传输服务器读取文件发送到 socket 上，通过 web 端进行下载时，读请求为顺序读，对于迅雷等下载工具下载时，读取为分片读取。

重点优化写操作：

在大多数情况下，对于元数据的写请求多于读请求（用户的主要操作为上传，备份），通常不会出现多个客户端并发读取文件的情况，因此我们的文件系统中将文件写优化作为重点。

3.1.1 存储系统的组成

本文实现了一个分布式文件系统原型，并重点研究其元数据管理模块，在该文件系统中，有集群管理器(Cluster Manager, cmgr), 元数据服务器(MDS), 对象存储服务器

(OSD)，客户端(Client) 四种角色：

集群管理器用于维护集群的配置，更新并发布所有元数据服务器的当前状态，分配元数据服务器的 Machine_ID，分配新建文件的 inode 范围等。

元数据服务器实现元数据属性的管理以及文件系统名字空间的管理，在我们的实现中，每个文件或目录被赋予一个唯一的 ID，用以在该文件的整个生命周期中标识该文件或目录，在具体实现中，我们将对目录和对象使用同一种数据结构，这样可以简化系统的设计。

对象存储服务器提供磁盘管理功能，在我们实现的文件系统中，OSD 是一个 HTTP 服务器，对象的读写通过 http://127.0.0.1/put/id 和 http://127.0.0.1/get/id 来进行，它们的表现和一个正常的 HTTP Server 完全一致，可以通过 HTTP Range 头来指定具体读写的文件范围，来这种方式实现 OSD 模块，简单高效，并且可以将 OSD 直接作为 HTTP 服务器暴露在 web 上，可以很方便的提供互联网下载服务，特别适合于网盘这样的应用。我们还可以通过 Nginx 模块的方式来实现 OSD 接口，可以提供非常高效的对象存储服务。

客户端将利用 Fuse 实现文件系统的大部分 POSIX 接口，将 Fuse 对文件的请求包装成对 MDS 和 OSD 的请求。在我们的实现中 Client 端是最为复杂的一部分，元数据的缓存，容错等机制都是在客户端实现的，它的代码质量和数据结构效率对整个文件系统的影响最大。

四个角色之间的关系如图 3.1 所示：

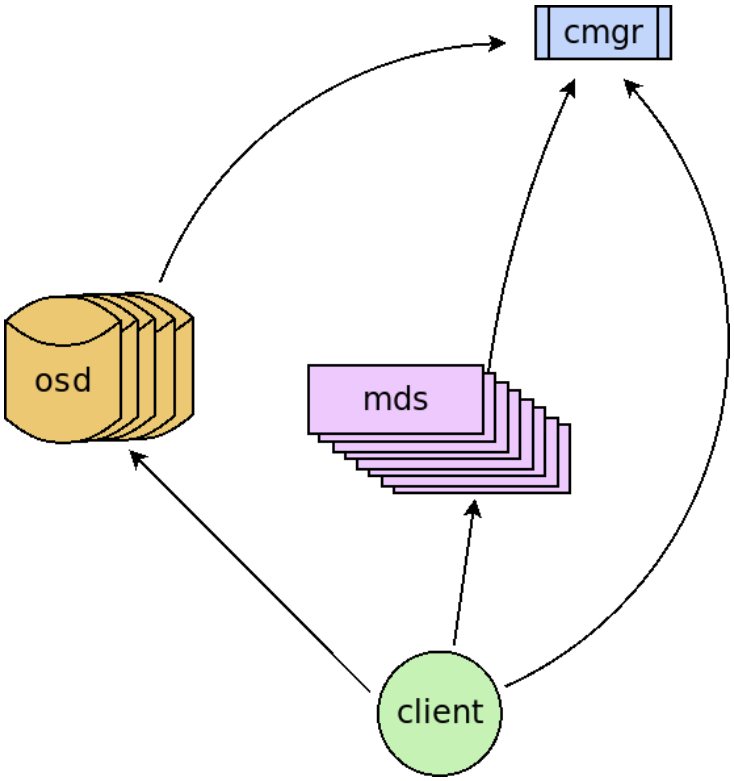


图 3.1 集群中各个角色的关系

3.1.2 文件与对象的映射关系

在我们的原型系统中，文件和对象具有一一对应关系，由唯一 ID 定位，而这个 ID 又和文件元数据 ID 唯一对应，即 inode number、文件元数据 ID、对象 ID 三个 ID 一致，此 ID 由集群管理器唯一分配，Client 在创建文件时，会向集群管理器申请一批 ID（通常为 100 个），然后从中选择一个，作为新创建文件或目录的 ID，下一次创建文件时，就可以从这些未使用 ID 中重新选择一个，这样可以防止多个 client 创建文件时使用相同的 ID，同时避免频繁的向集群管理器申请 ID，而 ID 的唯一性由集群管理器负责。

3.1.3 基于 Fuse 的 POSIX 客户端

Fuse 是一个在用户空间实现文件系统的框架，Fuse 向开发人员提供一组文件系统接口，开发人员只需要在用户态实现这些接口，Fuse 就会调用自己的内核模块处理文件系统请求，从而在用户空间下实现文件系统，其原理如图 3.2 所示。

在 Fuse 出现之前，文件系统通常是 Linux 内核的一部分，开发文件系统必须进入 Linux 内核，需要了解内核编程和内核技术 VFS(Virtual File System)方面的知识。在内核态的开发和调试通常都非常复杂，Fuse 的出现改变了这一局面，从 Linux 内核 2.6.14 开始，Fuse 已经集成进 Linux 内核。

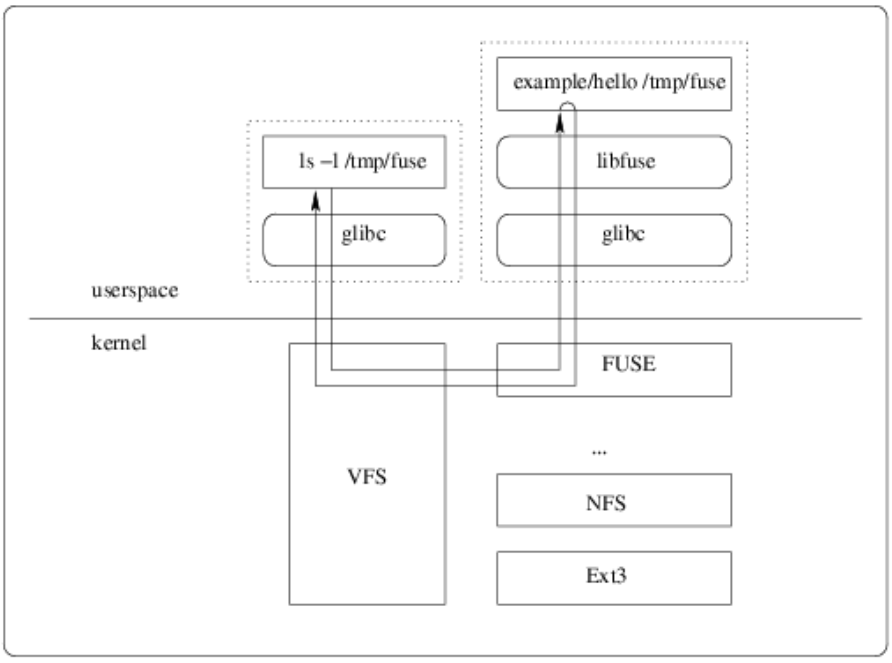


图 3.2 Fuse 原理(来自 http://fuse.sourceforge.net/)

Fuse 能够大幅简化文件系统的实现，简化了为操作系统提供新的文件系统的工作量，特别适用于实现网络文件系统和一些原型系统的开发。目前 GlusterFS、ZFS in Linux、MooseFS、SSHfs 等都是基于 Fuse 实现的。

它的优点包括：

- 库文件简单
- 稳定，使用广泛
- 安装简便，不需要重新编译内核
- 运行不需要 root 权限

但是，使用 Fuse 必然会引入用户态/内核态切换所带来开销，对文件系统的性能有一定影响。

本文中实现的原型存储系统，支持大多数 Fuse API（如表 3.1 所示），作为以数据存储为主要目的的文件系统，setxattr、getxattr、ioctl、 getlk、setlk 等操作都不是必须的，所以我们并没有实现。在目前的版本上，基本的文件系统操作如 grep、copy、 touch、tree 等都能正常完成。

表 3.1 本系统支持的 fuse 操作列表

init	✔		read	✔	
destroy	✔		write	✔	
lookup	✔		flush	✔	
forget		✘	release	✔	
getattr	✔		fsync	✔	
setattr	✔		opendir	✔	
readlink	✔		readdir	✔	
mknod	✔		releasedir	✔	
mkdir	✔		fsyncdir		✘
unlink	✔		statfs		✘
rmdir	✔		access	✔	
symlink	✔		create	✔	
rename		✘	ioctl		✘
link		✘	poll		✘
open	✔				

3.1.4 基于 libevent 的 rpc 框架

在整个文件系统的多个角色之间，消息通讯是至关重要的，客户端收到来自 Fuse 的 create、unlink、lookup 等请求后，都会向 MDS 发送相应的 rpc 请求，MDS 解析请求的参数，修改相应的数据结构后，再返回 Client，在我们的原型系统中，我们将使用 libevent 中提供的 evrpc 框架。

Libevent[25]是一个轻量级的开源高性能网络库，它使用 linux 的 epool 机制，是一个高效的、基于事件的网络 io 处理框架，在 libevent 中提供了一个简单的基于 HTTP 的 rpc 开发框架－evrpc，evrpc 不仅可以提供高效的事件处理机制，还能够自动生成网络数据

包格式，在具体开发中，使用 evrpc 可以省去很多不必要的麻烦，极大的简化了我们的原型系统开发工作，虽然 evrpc 是基于 HTTP 协议，协议开销较大，但是它的性能还是非常优秀，在我们的测试中，evrpc 服务器在一个 4G 内存的服务器上可以支持每秒 2 万次元数据请求。

3.2 版本化的元数据服务器节点管理

在分布式系统中，任何一个节点都可能在任何时刻加入集群，系统运行过程中也经常有节点因为出现故障而退出。我们需要动态维护集群中所有节点的在线状况，并且通知到集群中的每一个服务器中，为此我们设计了基于版本的元数据服务器节点管理算法。下面将首先介绍集群中服务器定位的方法，然后通过一个元数据服务器加入和离开集群的过程介绍该算法。

3.2.1 元数据服务器的定位

集群中的每个元数据服务器都会分配唯一的 Machine_ID，该 Machine_ID 是在元数据服务器第一次启动时候向集群管理器申请的，申请后记入自己的配置文件，无论今后元数据服务器是否重启，该元数据服务器都将一直使用此 Machine_ID。

在集群管理器中维护的 Cluster Map 是一个集群中所有节点的一个描述表，记录 Machine ID 到具体 MDS 服务器(IP，端口)的映射以及服务器的在线状况，Cluster Map 结构如表 3.2 所示：

表 3.2 Cluster Map

服务器类型	IP 端口	服务器 ID	在线状况
MDS	127.0.0.1:10000	10000	在线
MDS	127.0.0.1:10001	10001	在线
OSD	127.0.0.1:6006	6006	在线
OSD	127.0.0.1:6007	6007	在线
Client	127.0.0.1:0	8	在线

每个元数据服务器在启动的时候都会向集群管理器发起 Ping 消息，并且在自己的整个生命周期中都会定期向集群管理器发送 Ping 消息，Ping 消息将携带自己当前持有的 Cluster Map 版本，自己的 IP、端口等信息，集群管理器利用此消息维护 Cluster Map。

这种设计，可以避免元数据服务器重启后 IP 变化导致无法确定它是一个新加入的元数据服务器，或是一个元数据服务器短暂消失后重新回到集群中。

3.2.2 版本化的节点管理

版本化的节点管理，是指集群管理器在每次 Cluster Map 的改变的时候，都会赋予修改后的 Cluster Map 一个自增的版本号，每个节点在获取某个版本的 Cluster Map 后，

以后的 Ping 消息就会携带此版本信息，集群管理器在接到 Ping 消息后，首先将比较发起方的 Cluster Version 和自己当前持有的 Cluster Version，如果自己的 Cluster 版本较新，则表示集群有更新，会将当前 Cluster Map 返回，否则证明发起方已经具有最新的集群信息，集群管理器直接告诉发起方 Cluster Map 没有更新。

以上机制能保证集群中的每个元数据服务器尽快获得最新的 Cluster Map，此外，如果一个客户端访问某 MDS 失败，就会认为该 MDS 离线，它会将此消息报告给集群管理器，集群管理器确认后，将该 MDS 标记为离线。

元数据服务器和集群管理器之间的 Ping 消息结构如下：

```
struct ping {
    int version;
    string self_ip;
    int self_port;
    int self_type;
    int mid; //Machine_ID
}
```

集群管理器返回的 Cluster Map 结构如下：

```
struct pong {
     int version;
     optional int mid;
     array struct[machine] machines;
}
```

其中，machine 结构如下：

```
struct machine {
     int mid;
     string ip;
     int port;
     int type;
     int status;
}
```

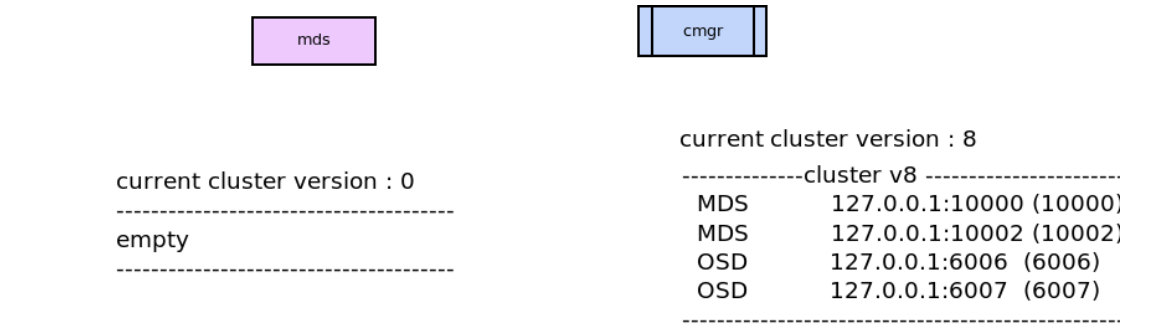
集群变更有元数据服务器加入和离开两种情况，下面以一个 MDS 加入集群和离开的过程分析，详细描述集群变更的过程：

3.2.3 节点加入

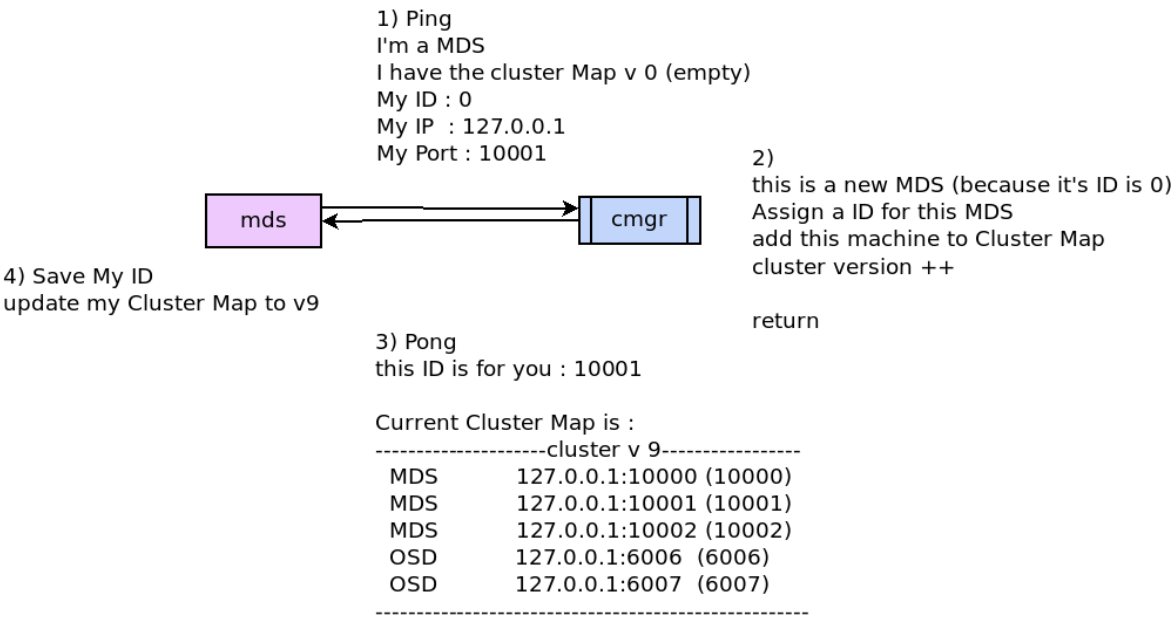
假设一个全新的 MDS 需要加入集群，如图 3.3 所示，集群管理器上已经有当前的 Cluster Map，版本号为 8，其中包含两个 MDS(10000，10002)和两个 OSD(6006，6007)。

MDS 启动后，根据配置文件，得到集群管理器的 IP/端口，向集群管理器发起 Ping

消息(如图 3.3(b)所示)，由于该 MDS 是第一次启动，Cluster Map 版本为 0，并且没有 Machine_ID，集群管理器收到该请求后，为此 MDS 分配一个 Machine_ID，将此 MDS 加入 Cluster Map，自己的 Cluster Map 版本从 8 变为 9， 同时将分配的 Machine_ID 和版本号为 9 的 Cluster Map 返回，MDS 就更新到当前最新的 Cluster Map。经过一段时间（2s）后，集群中的其它 MDS 和 Client 就会得知一个新的 MDS 加入，将会迁移一些元数据到此 MDS，它就开始提供元数据服务。



(a) 加入前



(b) MDS(10001) 加入过程

图 3.3 元数据服务器加入流程

3.2.4 节点离开

假设MDS(10001)在运行一段时间后宕机，它将首先被Client发现，某个Client在尝试向 MDS(10001)发起请求的过程中出错，Client 就会认为此 MDS 离线，并且将此消息

告诉集群管理器，集群管理器修改自己的 Cluster Map，将 MDS(10001)的状态标记为离线，版本变为 v10（图 3.4）。经过一段时间后，集群中所有元数据服务器都将获得 v10 版本的 Cluster Map，Client 将不再向该 MDS 发起元数据请求。

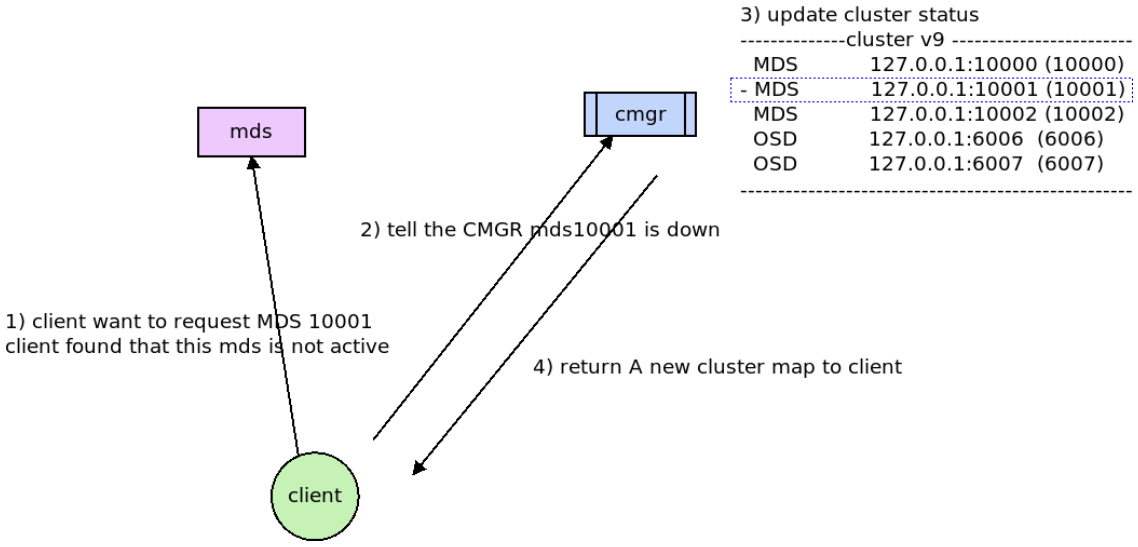


图 3.4 元数据服务器离开流程

3.2.5 节点恢复

如果 MDS(10001)经过维护人员的修理，很快得以恢复，它重启后，同样将首先向集群管理器发送 Ping 消息，其 Machine ID 已经固定为 10001，Cluster Map 为 v0，此消息将使得集群管理器重新将 MDS(10001)加入 Cluster Map，版本增加为 v10(如图 3.5 所示)。

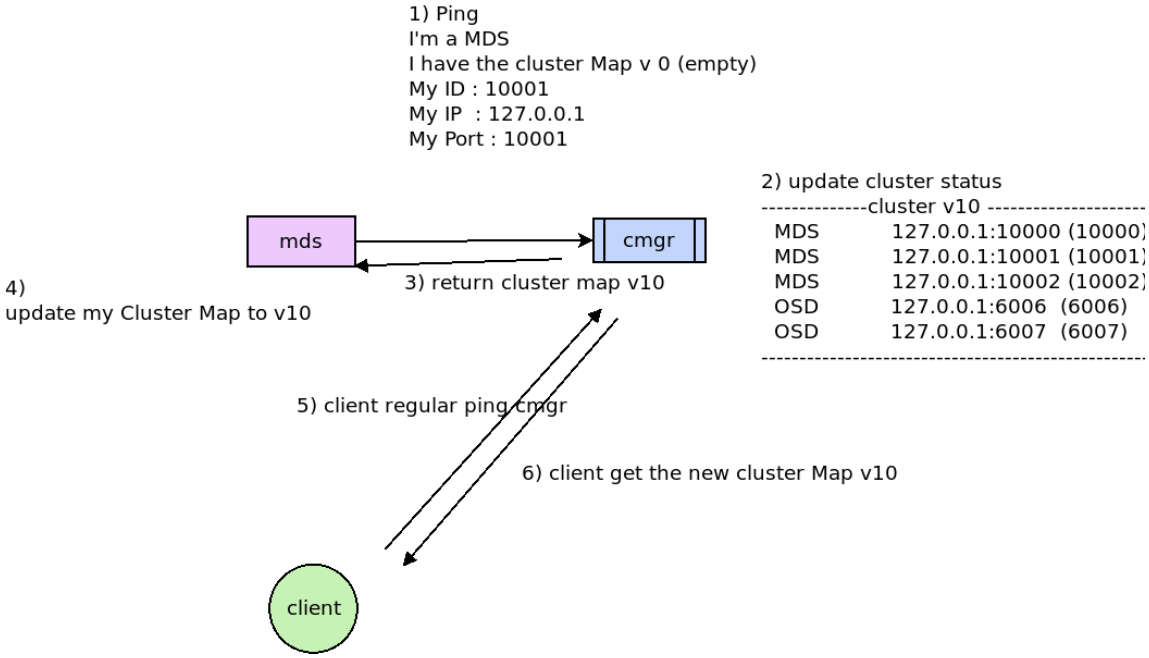


图 3.5 元数据服务器恢复流程

第四章 元数据分配策略

设计初期的对象存储系统为了降低设计的复杂度，一个文件系统中通常只有单个元数据服务器，为了能提供元数据的高速存取，很多系统都把元数据全部放在元数据服务器内存中（GoogleFS, mooseFS, HDFS 等），这样，系统的元数据个数就受到 MDS 内存的限制，最多可以支撑几亿文件的元数据，例如 MooseFS 的每个文件元数据需要占据大约 300 字节空间，这样在 32G 内存的元数据服务器上可以支持大约一亿个文件），在这样的硬限制下，系统管理员通常会限制文件系统集群的规模，几百台存储服务器作为一个集群，把它们构成一个类似于“卷”的概念，每个卷作为一个单独的名字空间，限制文件系统用户只能在单一的卷内操作，卷之间的操作会比较麻烦。

除了上面提到的名字空间受限的问题外，单 MDS 还有单点失效(single point of failure，SPOF)的问题，单一 MDS 在系统高负载，大并发的时候也不能很好的应付整个系统带来的负载，成为系统的性能瓶颈。此外，元数据服务器重启过程中，需要将整个名字空间从硬盘加载到内存，这是非常耗时的，经常需要十分钟甚至更长的时间。

Google File System 和 HDFS 目前也面临单一元数据服务器带来种种限制，在 HDFS 邮件列表中，元数据服务器集群化已经成为大家关注的热点，有人尝试利用 Ceph 为 HDFS 提供可扩展的元数据服务[26]。将 MDS 扩展为集群方式是必由之路，然而，目前在开源系统中只有 Ceph 可以支持基于动态子树划分的元数据管理。

本章中，我们将利用现有的一些元数据集群化分布方法，实现文件系统元数据的分布式管理，将整个文件系统的元数据负载在集群中合理地分布，为对象存储系统的元数据管理提供较好的扩展性和很高的性能。

4.1 元数据服务器集群技术

在学术界，针对对象文件系统元数据集群的讨论已经有由来已久，常见的策略有 Hash 切分，静态子树切分，动态子树切分，动态绑定等几种方法，下面将逐一详细讨论：

4.1.1 静态子树划分

静态子树划分是最简单的元数据分布策略，分布策略预先配置到客户端中，客户端在作元数据请求的时候，就直接联系相应的 MDS。

下面是一个典型的静态子树划分配置：

表 4.1 静态子树划分

目录	提供服务的 MDS
/usr/bin	MDS0
/usr/include	MDS1
/usr/lib	MDS2
/usr/share	MDS3

这个配置对应于如图 4.1 所示分割策略，每个元数据服务器独立处理一个子树下的元数据请求。

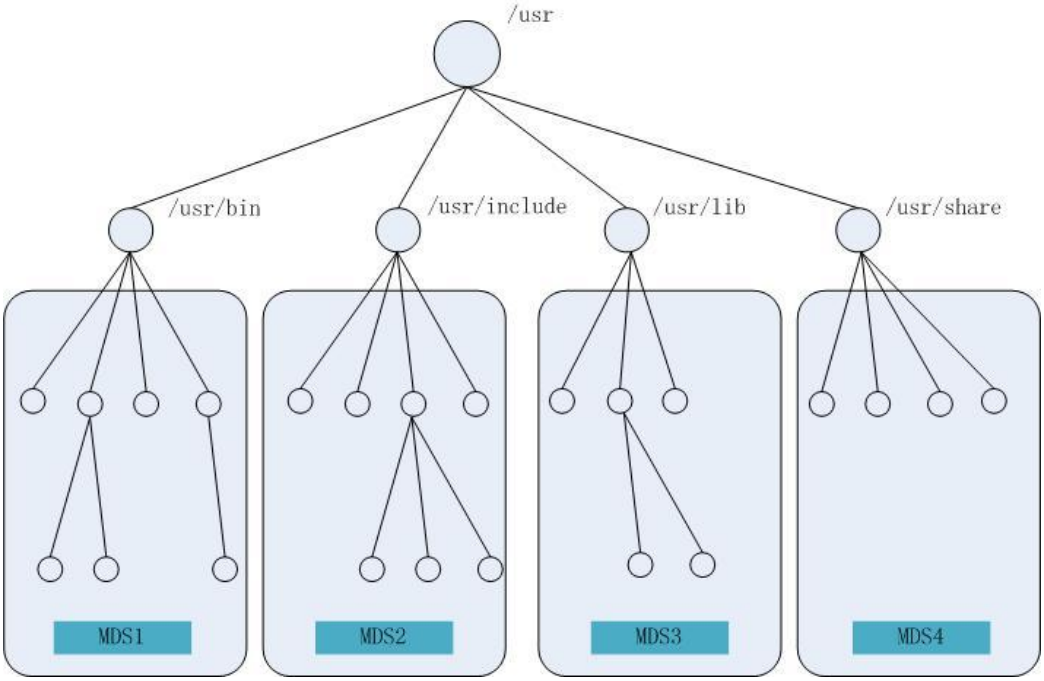


图 4.1 静态子树划分示意图

使用静态子树划分的好处是：Client 无须预先联系一个中心服务器，可以直接联系对应的 MDS 服务器就可以进行文件元数据操作，坏处是：如果某个文件或目录被多个 Client 集中访问（成为热点），这个目录所在的 MDS 的性能就会急剧下降。另外，如果某个子树的文件个数迅速增多，使得该子树所在的元数据服务器负载上升，元数据服务器集群的处理能力受到限制，这时就需要手动调整元数据的划分策略，例如/usr/include 下的/usr/include/mysql 目录迅速增大，可以增加一个 MDS 服务器，并做如表 4.2 所示调整，但是，这样的调整必须重启系统，不能实现动态扩展。

表 4.2 静态子树划分调整结果

目录	提供服务的 MDS
/usr/include/mysql	MDS5
/usr/bin	MDS1
/usr/include	MDS2
/usr/lib	MDS3
/usr/share	MDS4

例如 NFS、AFS、Coda 和 Sprite 都是使用静态子树划分策略，中科院为曙光开发的 DCFS1 是一个 SAN 文件系统，其元数据管理也是使用静态子树划分策略。

由于该方法的静态性，进行以上调整必须暂停服务，而不能动态调整，使用该方法的文件系统不具有扩展性，在本文中我们将不予考虑。

4.1.2 Hash 方法

Hash 方法将文件 id 或者路径计算 Hash 值，再把 Hash 的结果映射到某个 MDS 服务器，这种方法实现的分布策略简单，负载能够较好的在多个 MDS 之间均衡，而且实现也比较容易。

例如，有 4 个 MDS：MDS0，MDS1，MDS2，MDS3，采用如下 Hash 函数时：

Hash(id) = id%4;

可能的元数据的分布情况如下表所示：

表 4.3 Hash 划分方法示意

目录	id	Hash(id)	MDS 服务器
/usr	1	1	MDS1
/usr/include	3	3	MDS3
/usr/bin	4.	0	MDS0
/usr/lib	7	3	MDS3
/usr/share	9	1	MDS1
/usr/include/mysql	11	3	MDS3
/usr/include/err.h	14.	2	MDS2
/usr/include/memory.h	16	0	MDS0
/usr/include/regex.h	19	3	MDS3

相对于静态子树划分，请求被均衡地分布到每个服务器上，能够更好地做到负载均衡，不会因为每个子树的膨胀而需要手动修改划分策略。

但是该方法的主要问题在于当元数据服务器集群需要增加元数据服务器时，Hash函数会变化，映射关系也就发生变化，原来映射到MDS1的元数据现在很可能有大部分被映射到MDS2，此时会导致大量元数据发生迁移，由于迁移期间系统无法对外提供服务，这回降低系统的可用性。

该方法的另外一个缺点是不能利用文件系统使用中的局部性，比如对于bash的ls命令，首先需要readdir系统调用读取目录中的子目录或文件列表，然后需要对每个文件做stat系统调用，如果使用用基于子树的划分方法，可以在一个网络连接上读取所有这些文件的stat信息，而基于hash的切分方案，则需要针对每一个子目录或文件，分别对每个元数据服务器发起请求。

具体来说，Hash有两种形式，比较常用的是以文件的路径作为键计算Hash值，另一种是用文件或目录的inode id作为键。Vesta和Mezzo就是使用全路径Hash作为键进行Hash，使用全路径Hash的问题在于在对目录进行rename，remove等操作的时候，该目录下的所有子目录的Hash值都会随之发生变化，如果更改一个较高层次的目录名字，将涉及大量Hash值的更新，由此带来的效率问题非常严重。

4.1.3 Lazy Hybrid 方法

Lazy Hybrid[27]方法和Hash类似，但是它并不直接将Hash值映射到一个MDS服务器，而是利用一个动态的元数据查找表（Metadata Look Table，MLT)结构，将Hash后的结果在MLT表中查找，可以找到存放该元数据的MDS。一个典型的MLT如下：

表4.4 LH方法的MLT

hash范围	对应的MDS服务器
0x0000-0x2FFF	MDS0
0x3000-0x8FFF	MDS1
0x9000-0xCFFF	MDS2
0xD000-0xFFFF	MDS3

注意以上Hash值范围并非均匀分布，而是根据MDS的负载能力而有所不同，例如MDS1硬件配置高，就会多分配更大的MDS范围给它。这种方法避免了Hash的静态映射模式，而是使用动态映射，这样在增加MDS服务器的时候，之需要少量的迁移，并且修改MLT即可，而且可以动态调整MLT来实现负载均衡，假如在系统运行过程中发现MDS2负载过大，可以增加一个MDS服务器MDS4，并调整MLT如下：

表 4.5 更新后的 MLT

hash 范围	对应的 MDS 服务器
0x0000-0x2FFF	MDS0
0x3000-0x8FFF	MDS1
0x9000-0xAFFF	MDS2
0xB000-0xCFFF	MDS4.
0xD000-0xFFFF	MDS3

这样，只需要把MDS2上的一部分数据迁移到MDS4.，就可以对系统实现扩容，该方法有效解决了静态 Hash 中的扩展性问题。

Lazy Hybirid 也是使用文件的全路径计算 Hash 值，所以，当对目录进行 rename，remove 等操作的时候，该目录下的所有子目录的 Hash 值都会随之发生变化，带来大量的元数据迁移。

4.1.4. 动态子树分割

前面提到的动态 Hash 方法不能利用文件系统使用中的局部性，而在动态子树划分方法中，元数据在 MDS 上被组织为树状，保持文件系统的目录结构，这样，在大多数情况下，对元数据的 readdir 操作就可以在一个请求中完成。另外，对文件系统的 rename 操作，不会引起任何元数据迁移。

Ceph 是第一个使用动态子树划分算法的文件系统，它可以根据元数据服务器的负载动态调整元数据的分布策略，将负载过重的MDS上的元数据迁移到其它元数据服务器，此外，Ceph 还可以将访问热点复制多份，分布在多个 MDS 上，为多客户端并发读取同一个元数据的情况提供较好的性能。和 Lustre 一样，Ceph 在 MDS 之间使用共享存储保存所有文件的元数据。Ceph 利用动态子树划分算法，通过 128 个 MDS 可以支持每秒 100000 次元数据操作[12]。

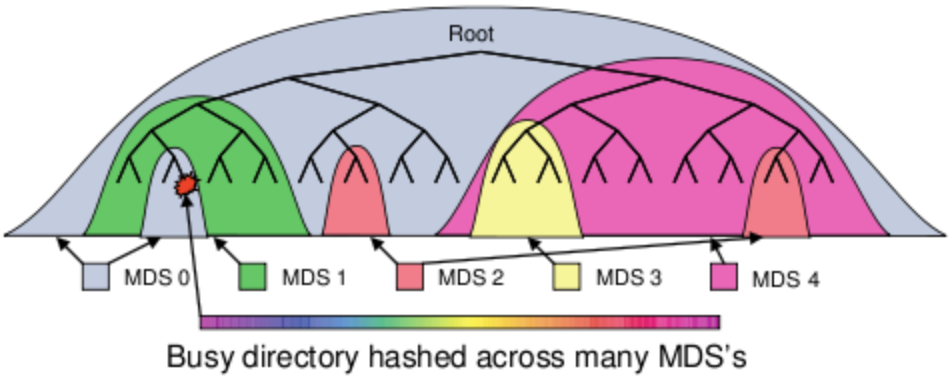


图 4.2 Ceph 的动态子树划分（来自 Ceph）

动态子树划分方法利用目录层次结构中固有的联系，能够克服 Hash 类算法中 readdir 操作的开销，也能避免 rename 操作引起的大量迁移，然而，也正是因为树形结构中，父子的联系太过紧密，动态子树划分中，元数据迁移算法的实现会比较复杂。当前，除了 Ceph 以外，中科院计算所开发的 DCFS2 也支持动态子树分割的概念（ DCFS2 是一个 IP-SAN 文件系统）。

4.1.5 其它方法

蓝鲸文件系统（BWFS ）[28] [29] [30] [31]是一个支持 MDS 集群的 SAN 文件系统，BWFS 使用了一种动态绑定方法，该方法中，元数据集中存储在共享的磁盘中，系统中有一个绑定服务器（Binding Server，BS），第一次访问一个文件/目录时，由绑定服务器根据当前的负载情况决定由哪个 MDS 来管理这个文件/目录的元数据，保证每个元数据有唯一的 MDS 负责。BS 还可以检测 MDS 的负载情况，通过改变映射关系，实现负载均衡。但是 BWFS 的绑定是以单个文件作为绑定单位，绑定服务器上保存每个文件到 MDS 服务器的映射关系，每当第一次对某文件发起元数据请求时，都需要先向 BS 查询 MDS 信息，绑定服务器会成为一个单一故障点，并且随着系统规模的扩大，也将会是系统中的一个瓶颈所在。

华中科技大学的 Handy[32] [33]系统，使用 Dhash 和 Chord[34] [35]算法实现元数据和数据的完全分布，系统利用逻辑矢量环实现动态扩展和资源的动态管理，元数据存储服务器和数据存储服务器分别构成两个逻辑矢量环，如图 4.3 所示。Chord 原本是为 P2P 系统设计，固有动态添加节点的特性，可以方便的添加和删除元数据服务器和存储服务器，具有很好的扩展性。但是，Handy 的元数据是存放在硬盘上，而不是内存中，不能提供太高的效率（大约 100 op/s）。另外，Chord 设计为支持大量节点的 P2P 系统，对于小规模的元数据服务器集群，使用 Chord 算法是不合适的。

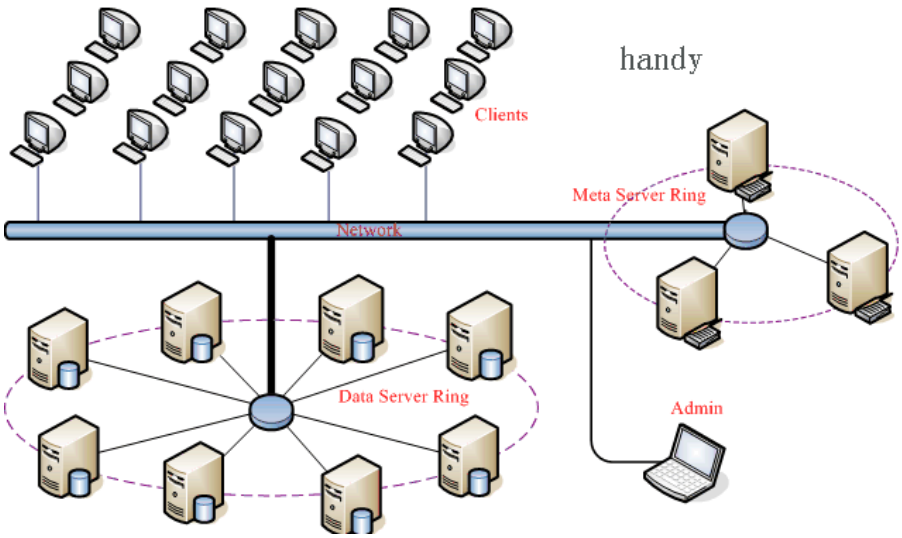


图 4.3 Handy 原理(Chord 环)

4.2 基于动态子树划分的元数据分配算法

通过上面的描述，可以发现元数据在 MDS 集群上的分布策略，是影响元数据服务器集群性能、可靠性和扩展性的主要因素，前面介绍了 Hash 算法，静态子树分割算法，一致性 Hash 和动态子树分割几种算法，并分析了其优劣。

Hash 算法和静态子树划分算法由于其不具有扩展性，不能适应系统的动态扩容需求，Lazy Hybrid 算法和一致性 Hash 算法都是基于 Hash 的算法，虽然它们能有有效扩展，并且能提供较好的负载均衡能力，负载迁移的实现相对简单，但是它们将每个目录中的子节点分布到多台服务器，对文件系统某个目录的 readdir 操作需要逐一对所有包含这些子节点的每个元数据服务器逐一发起请求，影响系统性能。目录的 rename、move 操作带来的大量迁移也会对系统性能产生不良影响。

蓝鲸文件系统中的动态绑定方法是基于元数据共享存储的，在大量元数据请求时，绑定服务器容易成为系统瓶颈。Handy 中使用的 Chord 算法，本文认为它适用于较大规模的集群（集群规模应该在万台以上），因此不符合本文的需求。

表 4.6 元数据分布算法比较

算法名称	扩展性	负载均衡	局部性	主要问题
静态子树分割	✗	✗	✓	不能有效分布负载
静态 Hash 分割	✗	✓	✗	扩充服务器或 rename 操作需要重算 Hash
Lazy Hybrid	✓	✓	✗	rename 操作带来大量迁移
动态子树分割	✓	✓	✓	负载均衡较难以实现
动态绑定(蓝鲸)	✓	✓	✗	绑定服务器成为瓶颈

经过以上分析，动态子树划分方法能够提供很好的扩展性和较高的性能，能利用文件系统访问的局部性，因此，在我们的原型系统中，我们将使用了动态子树划分算法实现元数据的分布。

4.2.1 元数据与元数据服务器的映射关系

如何根据一个元数据 ID 定位到它所在的元数据服务器，是对象存储系统的一个核心问题，对于大量的元数据，我们无法维护一个集中的全局的映射表，因为其开销与文件系统中的 inode 个数成正比。

在我们的原型系统中，我们将每一个文件元数据与元数据服务器的映射关系分布式的存放在每一个元数据节点上（如图 4.4、图 4.5 所示），该映射信息在客户端每次 readdir、stat、lookup、create、mkdir、symlink 操作后返回给客户端，并且由客户端缓存。

为了描述元数据的分布和备份策略，元数据的迁移等，文中将使用如图 4.4 所示图元表示一个文件系统 inode：

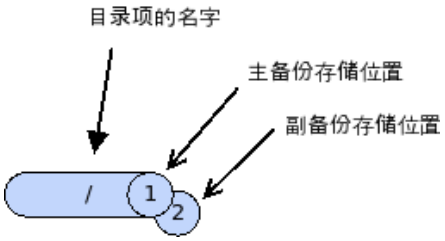


图 4.4 每个 inode 存储示意

在具体实现中，我们使用如图 4.5 的结构来保存每个文件元数据的属性及其与 MDS 服务器的映射关系。对于目录而言，图中 mds_position_1 和 mds_position_2 表示元数据存放的位置（MDS 的编号），对于文件来说，它们表示存放文件数据的两个 OSD 编号。

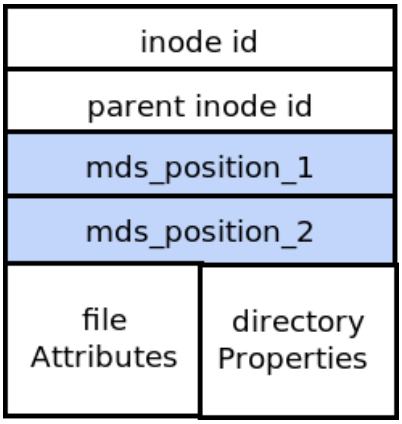


图 4.5 元数据的存储结构

在客户端看来，文件系统的元数据空间是一个完整的树形结构，如图 4.6 所示，客户端对每个元数据的请求，都会根据缓存中该元数据所存放的 MDS 信息，向相应的服务器发起请求。

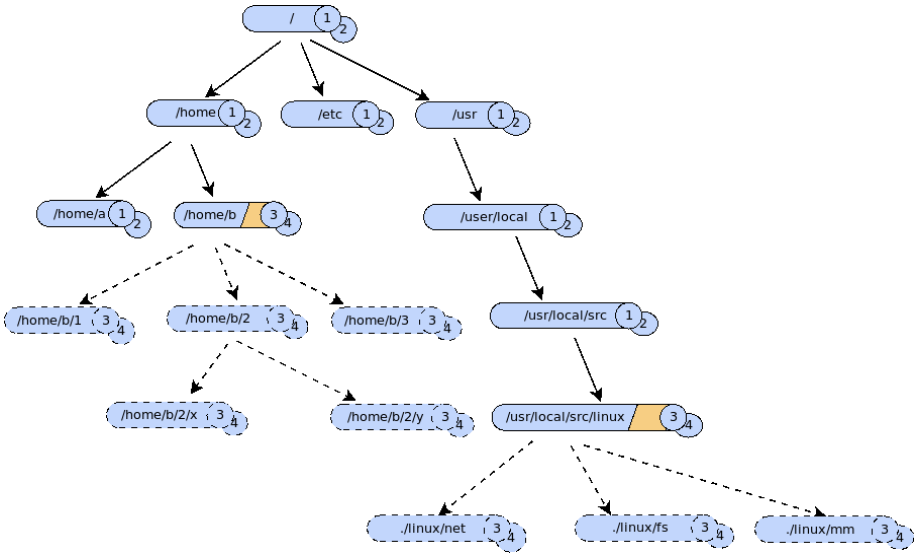
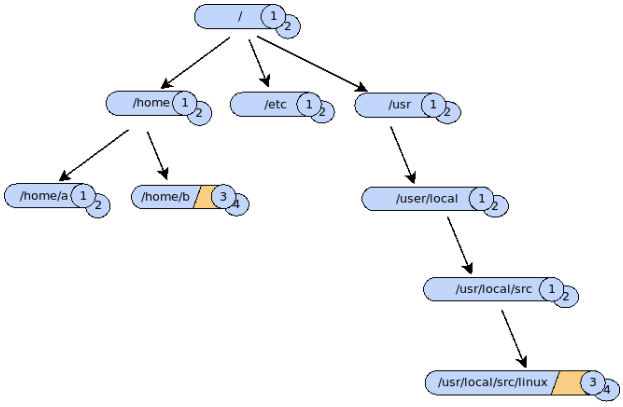
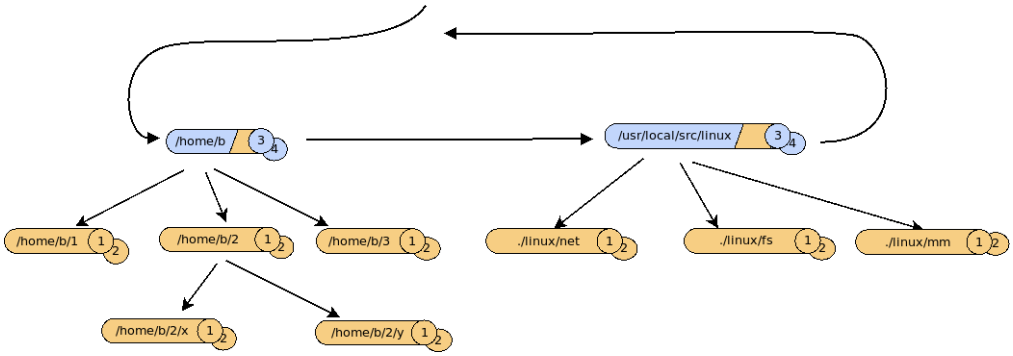


图 4.6 动态子树划分的整体名字空间视图

图 4.6 是一个客户端看到的文件系统的整体结构，实际上，这棵树上的所有元数据被存放在两组元数据服务器上，它们分别为 MDS1/MDS2 和 MDS3/MDS4，在这两组服务器上，存放的数据结构如图 4.7 所示，



a) 在 MDS1/MDS2 上的数据结构



b) 在 MDS3/MDS4 上的数据结构

图 4.7 动态子树划分在各个元数据服务器上的存储结构

客户端通过向 MDS1/MDS2 或 MDS3/MDS4 发起合适的元数据请求，就可以对外提供一个统一的元数据空间，例如，客户端访问/home/b/2/x 的过程中，发起的元数据请求是这样的：

MDS1 : lookup(1,'home') => 返回 {ino:2, name:'home', type: dir, pos:[1, 2]}

MDS1 : lookup(2,'b') => 返回 {ino:22, name:'b', type: dir, pos:[3, 4]}

MDS3: lookup(22,'2') => 返回 {ino:315, name:'2', type: dir, pos:[3, 4]}

MDS3: lookup(315,'x') => 返回 {ino:9527, name:'x', type: file, pos:[601, 602]}

其中对于目录(type=dir)，pos 表示两份元数据所存放的 MDS 编号，而对于文件(type=file)，pos 表示存放数据的两个 OSD 的编号。

4.2.2 分裂点

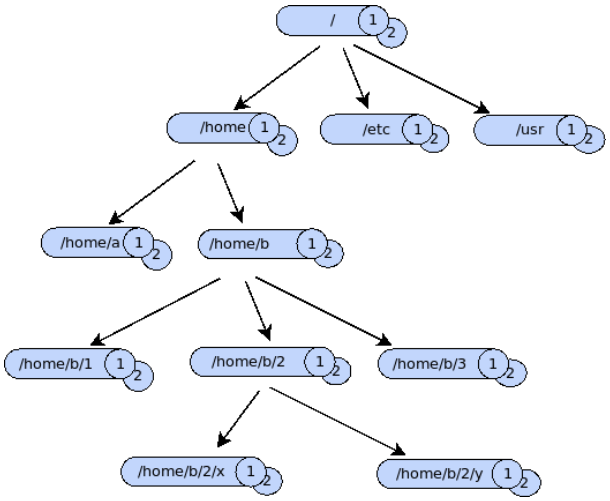
对于图 4.6、图 4.7 中/home/b 和/usr/local/src/linux 这样的 inode，我们称之为分裂点（熊劲的论文[36]中称为分支点），分裂点的元数据被分为两部分，分别保存在两个 MDS 上，对于这些 inode 的修改必须涉及多个 MDS，破坏了文件系统名字空间的层次结构关系，给元数据的管理带来复杂性，但这又是分布式元数据处理所必须的。

以/home/b 为例，在 MDS1 上/home 对应的 inode 下， 有/home/b 对应的 inode，但是这个 inode 只保存/home/b 的 name 属性（用于 lookup、unlink）和 inode number，对于该 inode 进行的 lookup('/home', 'b')、rmdir('/home/', 'b')这样的请求将发送到 MDS1 进行，而对于 setattr, getattr, readdir 之类的请求，Client 将发送到 MDS3 进行处理。

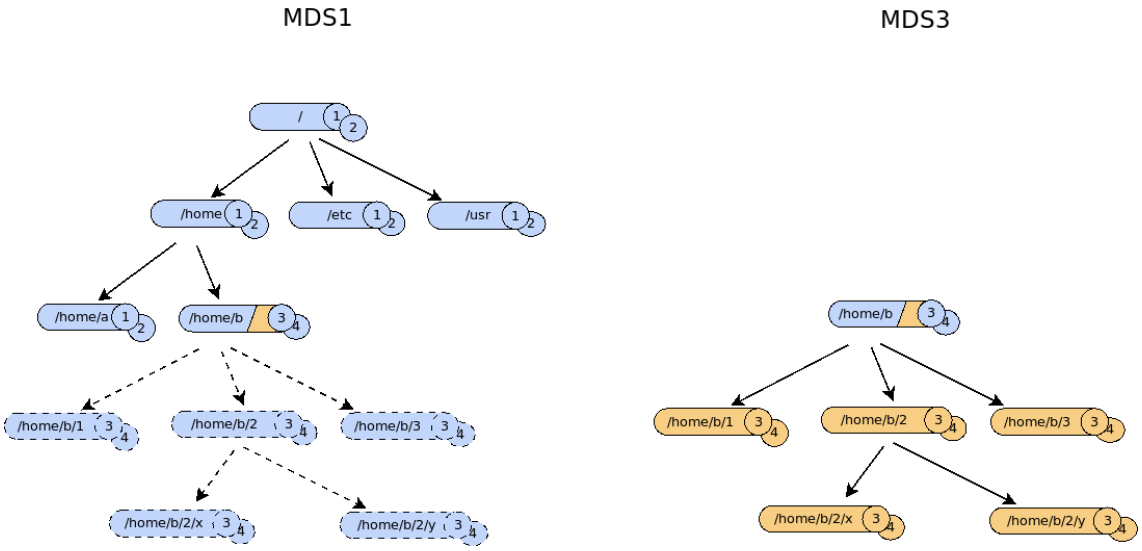
由于对分裂点的操作需要多个 MDS 的协调，这不仅增加了每个操作的处理延迟，而且增加了网络负载，大大影响元数据的性能，所以，我们需要在尽量减少分裂点的同时，保证元数据的尽量平均分布。 在中科院计算所曙光文件系统 DCFS2 上，使用基于粒度的元数据分布策略来达到此目的，当在一个目录树中创建一个新对象（文件/目录）时，如果新加入对象后这个目录树没有超过分布粒度的大小，则就在那个服务器新建元数据，否则重新选择一个元数据服务器来新建元数据。

本文并没有采用基于粒度的分布算法，本文中，分裂不是发生在 mkdir 的时候，而是在系统运行中，由元数据迁移形成的。因为对于一个新建的目录项，其元数据和一个新建的文件是没有差别的，如果今后这个目录下没有文件或文件个数很少，这一次分裂就会反而导致系统性能降低。在 mkdir 的时候，目录的元数据默认在它们的父目录所在 MDS 上创建，只有当它下面创建的文件越来越多，才会考虑将它作为一个子树迁移出来，进行一次“分裂”。

图 4.8 表示一次分裂过程，在分裂前/home/b/元数据上记录的映射关系为 MDS1/MDS2，迁移后，其映射到 MDS3/MDS4。



(a) 分裂前



(b) 分裂后

图 4.8 元数据节点分裂示意图

经过进一步的思考，对于文件而言，它的一次分裂虽然有利于元数据更均匀的分布，但是在对这个文件元数据的访问却需要两次请求，这是得不偿失的。所以，本文中分裂只会发生在目录类型的 inode 上，对于文件类型的 inode，它们的元数据一定存放在父节点所在的 MDS 上，客户端只需要根据其父目录的映射关系就可以得知自己所在的元数据服务器。

这样做的好处是，系统中只有较少的分裂点，所以，对于文件系统中的绝大多数请求都能在一个元数据服务器上完成。

4.2.3 定位根节点

Client 启动后，首先需要连接到元数据集群，获得文件系统根节点的信息（Fuse 文件系统根节点 id 为 1，这里需要对 inode 为 1 的元数据发 stat 请求）由于元数据分布在多个 MDS 上，而所有 MDS 都是功能等价的，client 无法知道根节点存储的位置。

为此，所以在我们的原型系统中，Client 启动后将首先向集群管理器获取最新的 Cluster Map，然后以同步轮询的方式向所有 MDS 发起 stat(1)请求（如图 4.9 所示），由于元数据服务器的数量通常较少(少于 100 台)，我们才可以使用轮询方式获得根节点。一旦发现某两个 MDS 上还有根节点，Client 就会把这一信息记录到缓存中，以后的请求就会根据此映射关系发出。在启动过程中，如果少于两个 MDS 上含有根节点 inode，Client 将会拒绝启动。

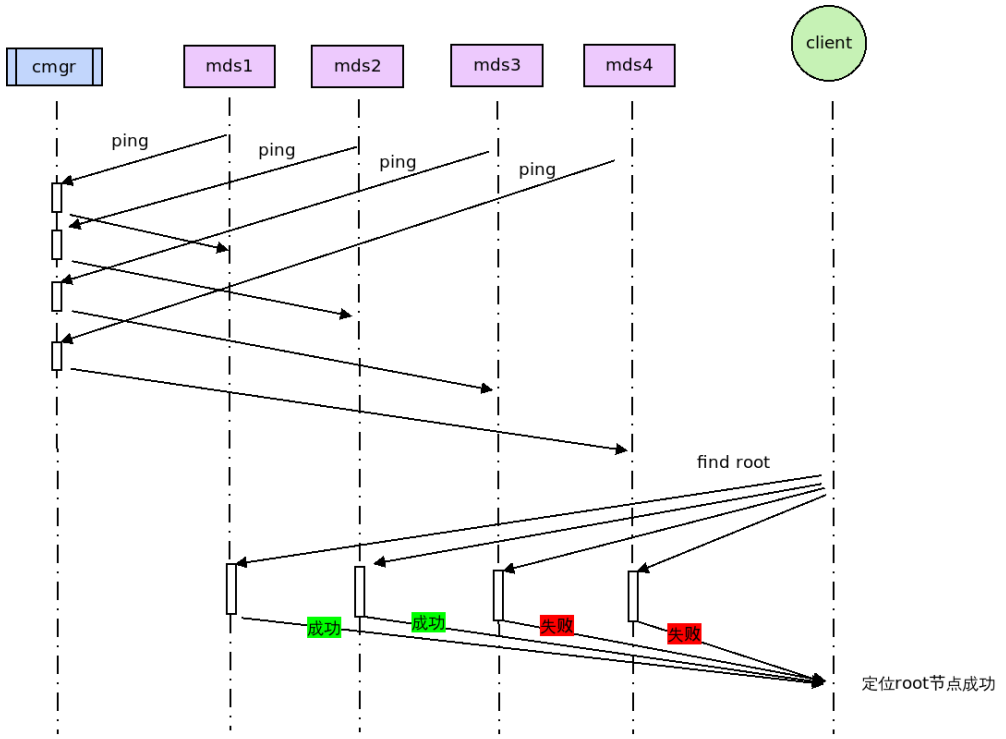


图 4.9 文件系统启动时定位根节点的过程

4.3 元数据备份策略

为了保证元数据的安全，集群中需要使用某种形式的备份策略，目前常用的方法有冗余备份和日志/检查点等。其中冗余备份方式将每个文件的元数据存 2 份或 3 份，在主备份丢失后，可以马上启用备份，冗余备份可以通过一个专用的备份服务器实现，Google File System 和 Moose File System 都是使用这种策略。日志/检查点方式通常配合两段锁协议，需要对每个元数据操作都记录日志，并定期合并日志文件，生成检查点存入磁盘，这种方式实现较难。

在当前我们的实现中，每个 inode 的元数据会保存两份（主备份/副备份），每个文件的数据也会保存两份，分别存在两个 OSD 上，元数据操作和数据操作都是直接由 client 分别写在两个 MDS/OSD 上，其存放位置保存在 MDS 中的 inode 结构上（如图 4.4 所示），如果一个 inode 的宿主 MDS 因故障而宕机，元数据的存取将直接重定向到另一个 MDS 上。

本文将元数据的操作根据读写不同分为两类：

元数据写：

包括：write,、flush、create、mkdir,、symlink、setattr、unlink

元数据读：

包括 read、get_attr、lookup、readdir、open、readlink、statfs、access

对于写类型的元数据操作，客户端将并行地向两个 MDS 发起请求，只有两个请求

都成功返回，客户端才会认为文件元数据写保存成功，如图 4.10 所示：

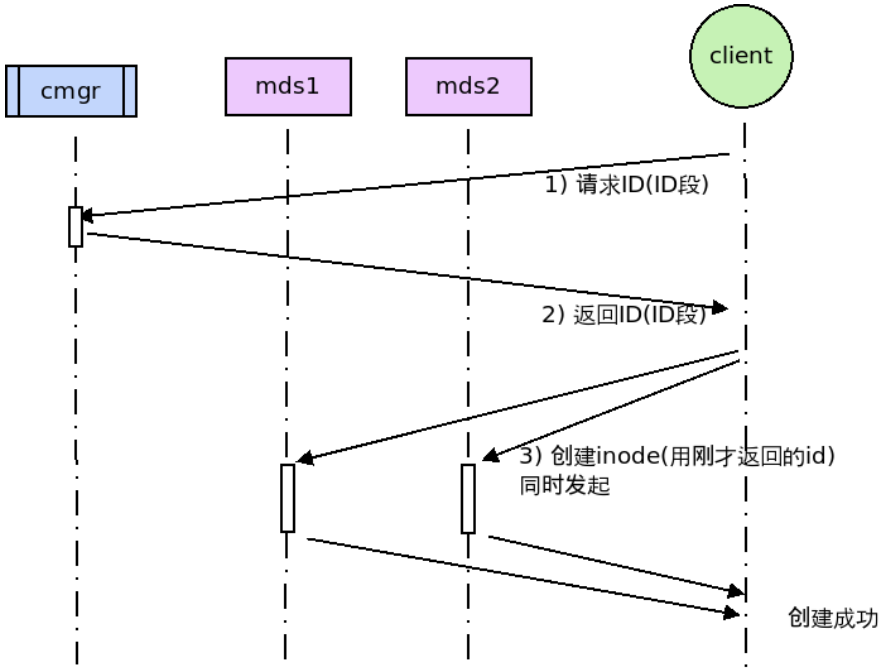


图 4.10 创建新文件时请求示意

对于读类型的元数据操作，客户端将首先尝试从主备份获取文件元数据，如果成功获取则返回，如果主备份出错，客户端将尝试从副备份获取数据，如图 4.11 所示：

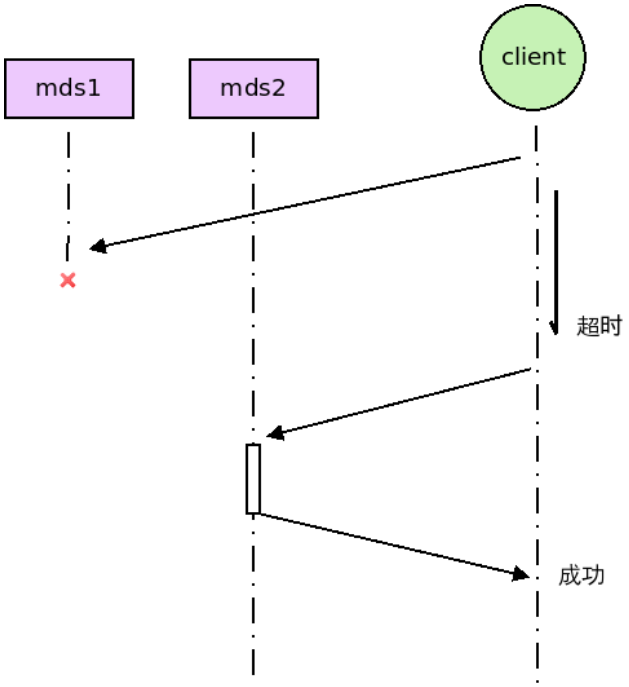


图 4.11 故障发生时，获取文件元数据(stat 操作)

在上面提到的元数据备份策略中，如果一个 MDS 宕机，读取元数据的操作还能够正常进行，而对元数据的写操作，却会因为其中一个 MDS 失效，而无法完成，系统可

用性降低。

为此，我们在对某个文件元数据的写操作中发现一个 MDS 失效后，应该向含有该元数据的另一个 MDS 发起请求，要求它将该元数据所在的子树复制到另一个存活的 MDS，同时更新映射关系，之后才能向含有新副本的 MDS 发起请求。

4.4 小结

在本章中，我们对目前已有的元数据分布算法做了详细介绍分析，通过比较分析，我们选择动态子树划分方法来实现元数据在多个 MDS 之间的分布，我们通过“分裂点”来实现动态子树划分算法，使用“双写”的方式实现元数据冗余备份。

与静态子树划分方法和静态 Hash 方法相比，动态子树划分能够具有很好的扩展性，与动态绑定法相比，不存在单一故障点和瓶颈问题；与 LH 方法相比，它能避免 rename 的巨大开销，在 readdir 操作中也具有较好的效率。

第五章 元数据负载均衡

在元数据集群中，当一个新的元数据服务器加入系统后，旧的元数据服务器需要把一部分元数据迁移到新元数据服务器上，才能让新的元数据服务器承担一定的元数据负载，提高整个元数据服务器集群的处理能力。另外，当文件系统中某个子树成为系统热点，该目录所在的元数据服务器性能就会急剧下降，这时候需要把发生热点的子树迁移到其他负载较轻的元数据服务器，保证所有元数据服务器负载相对均衡，提高元数据集群的整体处理能力。元数据负载迁移需要考虑以下几点：

- 有效性：负载迁移后，系统整体性能应该提高
- 一致性：元数据迁移期间，该子树不应该出现 create、delete 等写操作，防止元数据在不同的 MDS 之间不一致。负载迁移后，Client 需要能获悉该元数据迁移信息，能够到新的元数据服务器上请求服务
- 稳定性：元数据迁移后，短时间内所迁移的元数据不应该发生再迁移，以免系统将大部分时间消耗在负载迁移上，导致系统抖动，影响对外处理能力。

因此，我们需要谨慎的选择需要迁移触发机制、负载信息的统计、迁移元数据范围、迁移的目的地等，还要考虑元数据迁移后 Client 端 Cache 的更新机制等。

5.1 负载统计

元数据迁移发生在负载重的 MDS 和负载轻的 MDS 之间，所以，每个 MDS 的负载统计是负载迁移的重要依据，负载统计主要涉及两个问题，一是负载的计算方式，包括 CPU、内存利用率、网络负载等；另外一个问题是负载的收集方式，是集中收集还是分布式搜集。一个 MDS 必须知道系统中所有 MDS 的负载情况，才能做出合理的迁移计划。

5.1.1 决策形式

在负载均衡系统中，常见的决策形式有集中控制和分布式决策之分，集中控制方式是指系统的所有负载信息都将汇总到一个主控节点，主控节点汇总了所有节点的负载信息，并依此做出决策，它的优点是在进行决策时信息充分，可以得到较好的决策，但是主控节点是单一失效点，容易成为系统的瓶颈。

而在分布式决策方式中，各个节点独立地收集负载，它们可能向自己的邻居发送请求，获得周边节点的负载信息，并依此做出决策，由于它们只能获得有限的信息，做出的决策不如集中控制有效。

在本文设计的系统中，MDS 服务器的规模通常少于 100 台，它们的负载信息不会对集群管理器造成太大的负载，我们完全可以使用集中式的方式来收集系统负载信息。同时，集群中的每个元数据服务器都能在报告自己负载的同时获得整个集群中所有 MDS 的负载情况，并依此做出决策。因此，我们使用的是一种“集中统计，分布式决策”的

方法，决策由各个 MDS 自己做出，当一个 MDS 发现自己的负载过重，就会尝试发起一次元数据迁移，降低自己的负载。这里，MDS 认为自己负载过重的条件是：

1）MDS 当前负载超过自己历史最大负载的 75%

2）存在比自己负载小得多的 MDS 节点。

5.1.2 负载计算

对于负载的计算方法，传统系统一般使用系统的 CPU 使用率和内存使用率来计算系统当前负载，这种方法适应性不强，比如一个 MDS 根本没有元数据请求负载，而它的 CPU 正好被其它进程占用，这时候的元数据迁移不但不能均衡系统整体负载，反而会加重自己所在元数据服务器的负载。

我们希望只有当一个元数据服务器不堪元数据请求带来的负载时，才发生迁移，因此，本文中，我们将结合当前系统资源的占用情况和该元数据服务器的每秒请求次数来计算 MDS 的当前负载：

在我们的原型系统中，MDS 会对每一个元数据请求计数，然后定期收集这些计数信息，假设此计数为 req_count，其算法如下，

- 对于每个请求，req_count += 1
- 每隔固定时间(2s)，获取 req_count，计算当前负载，再将 req_count 重置为 0

在每次计算当前负载的时候，将会综合 req_count 和 CPU 使用率，我们使用公式

$$load\_current = load\_cpu * req\_count$$

来计算元数据服务器的负载，使用该公式，只有在一个元数据服务器收到太多请求，导致自身 CPU 占用率上升时，才有可能发生迁移。

由于集群中的各个元数据服务器的异构性，不同元数据服务器的性能、处理能力等都各不相同，为了使系统具有一定的自适应能力，我们不能使用负载的绝对值作为元数据迁移的唯一标准，而是使用一个“相对负载”来衡量系统的当前负载，相对负载的计算方法如下：

我们将从该 MDS 启动以来所承受的最大负载,作为元数据服务器的处理能力的一个衡量标准，叫做“最大负载”，然后将当前负载与最大负载的比值称为相对负载，作为判断一个 MDS 是否“过载”的一个标准。在我们的原型系统中，当 MDS 当前负载与最大负载的比值大于 0.75 时，MDS 将认为自己过载。

5.1.3 负载累计

在我们的系统中，还必须要考虑 MDS 承受的突发负载，例如某个 MDS 在一段时间内集中收到一些突发请求，它的负载会急剧上升，如果这种请求只是暂时的，系统负载在很短时间内又会回到低点，我们就不能仅仅根据这个突发负载就做出迁移的决定，这样迁移带来的负载可能比这个突发负载更重，将是得不偿失的。

为此，我们将会根据当前负载，使用 AIMD(加性增、乘性减)公式对当前负载进行累

计（此过程类似于积分），当系统负载增加时，我们使用公式：

load_avg = load_old*0.9 + load_current * 0.1

对负载进行加权累计，其中 load_curent 表示系统当前负载，load_old 为上一时刻的负载，load_avg 为经过加权后的负载，这样，当系统负载 load_curent 增加时，load_avg 将会缓慢增加。当系统负载降低时，我们使用公式：

load_avg = load_old*0.5 + load_current * 0.5

进行负载加权，使得负载降低时 load_avg 会更快的衰减。

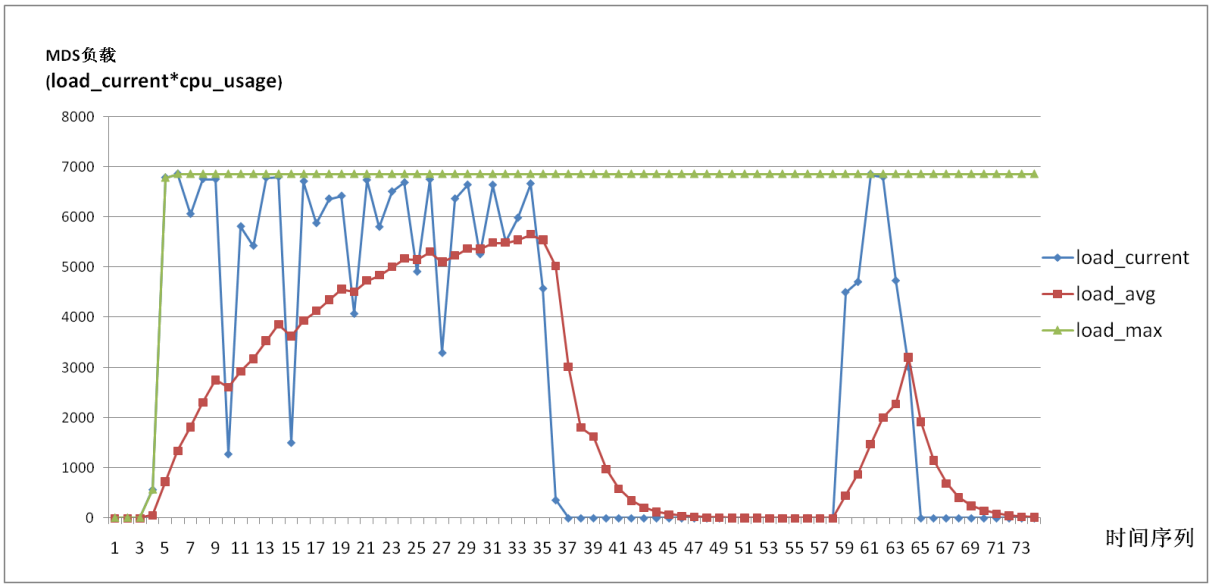


图 5.1 负载评估模型

如图 5.1 中，在时间点 35 之前，我们用 fileop 向我们实现的元数据服务器发起持续的 create 请求，load_current 快速增加，而 load_avg 的增加延迟于 load_current，从图中可以看出，使用此负载累计方法，在时间序列 60 时发生的突发负载，虽然导致 load_current 达到系统最大负载值，但是持续时间很短，经过加权累计后，load_avg 并没有到达持续负载所带来的负载峰值。

图中，load_max 为 MDS 服务器启用以来的最大负载，代表了该 MDS 的处理能力，我们可以调整 load_avg 与 load_max 的比值阈值，使得负载均衡更具有适应性。例如，对于该图所在的 MDS，我们可以使用 load_avg/load_max>0.75 作为元数据迁移的触发条件，时间序列 60 处的突发负载就不会触发元数据迁移。

5.2. 迁移粒度选择

前面文中已经说明，一个目录的所有子结点都直接保存在该目录所在的 MDS 上，对于发生分裂的目录，它会在父目录所在 MDS 和自己所在 MDS 都保存元数据，所以在我们设计的迁移算法中，迁移必须以一棵子树为单位进行，如图 5.3 所示。

此外，迁移的粒度不能太大，如果每次迁移一个很大的子树，一来需要很大的传输时间，由于需要在迁移过程中锁住整个子树，将导致文件系统元数据可用性降低；二来

迁移以后的子树还极有可能成为新的热点，很可能再次引发迁移。

为了选择一个合理的子树作为迁移单位，我们定义访问频度和子树大小两个指标。

5.2.1 访问频度

我们定义文件系统中每一个节点（文件，目录，符号链接）都有一个访问频度，表示该节点的热度，文件系统最初加载后，每个节点的访问频度都为 0，对于新建的节点，访问频度也为 0。

当文件系统中一个节点被访问时，该节点的访问频度就会被更新，同时，由该节点上溯到根节点路径上的每个节点都会被更新，每个节点的访问频度都被加 1。

如图 5.2 所示：

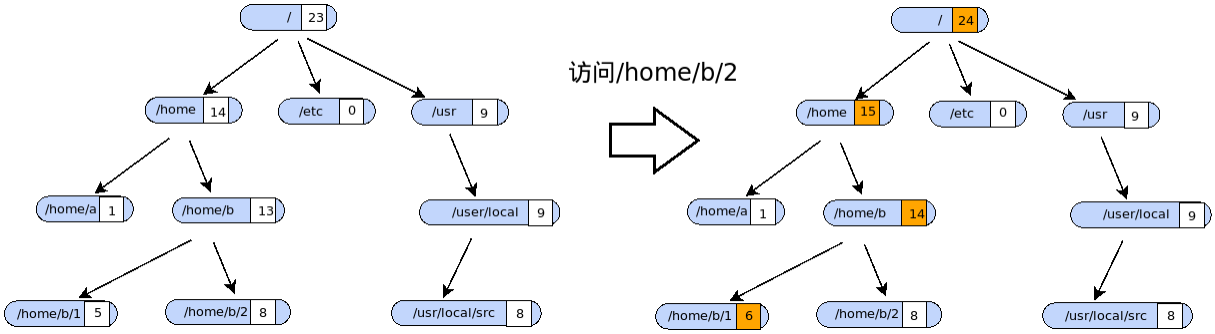


图 5.2 访问频度更新

当迁移发生时，我们将根据此访问频度，定位系统中访问较为频繁的子树，迁移这样的子树，能够较好的将自己的负载转移给其它元数据服务器。

5.2.2 子树大小

在本文述及的文件系统中，我们将会为每一个节点记录自己子树的大小，即自己所有后代的个数，从而为选取迁移单位提供依据。

当在一个目录下做 create、mkdir 或 symlink 操作后，都会在该节点的祖先节点上增加相应的大小，做 unlink，rmdir 的时候都会在该节点的祖先节点上减少该属性，使得该属性能够及时反映出该子树中含有的 inode 个数。注意该统计仅仅是为了在负载迁移的时候用于确定迁移粒度，是表示“某子树在本 MDS 上的大小” ，所以不会跨 MDS。

5.2.3 迁移子树选择

前文已经提及，迁移必须以一个子树为单位，迁移粒度不能太大也不能太小。在有的系统中，为了使系统具有一定的自适应能力，迁移的粒度并不是绝对的，而是一个相对值，通常相对于当前 MDS 中所有节点个数，例如在迁移时选择本 MDS 上节点个数约为总结点个数 5%的元数据作为迁移单位。

在我们的系统中，并没有使用如上面所诉的自适应策略，因为当每个元数据服务器上的元数据个数大于 1 亿后，即便是 1%的元数据，数据也将达到一百万，这么大量的

元数据作为一个迁移单位，迁移的时间和带来的负载都很大，根据我们对服务器性能的评估，我们选择 50000 作为一个迁移单位。当一个 MDS 发现自己负载过重，决定启动一次迁移时，它会从文件系统的根结点开始，寻找一棵负载较大而 inode 个数小于 50000 的子树作为迁移单位，具体算法如下：

```
function locate_hot_sub_tree(fsnode root){
  fsnode hot_son;
  for (son in root) {
    if (son.access_count>hot_son.access_count)
      hot_son = son;
    }
  return hot_son;
}

function select_migration_tree(){
  n = root_of_filesystem;
  while(1){
    n = locate_hot_sub_tree(n);
    if (n->tree_cnt < 50000) {
      migrate_tree(n);
    }
  }
}
```

5.3.迁移算法

将元数据从一个 MDS 迁移到另一个 MDS 的过程中，要注意保证元数据的两个备份以及 Client 端的一致性。为此，我们将使用锁机制和缓存的被动更新机制来实现元数据的迁移。

图 5.3、5.4、5.5 以一个子树的迁移为例，说明了我们选用的迁移算法，图 5.3 为迁移前的初始状态，此时 MDS1 和 MDS2 保存有相同的子树结构，MDS3 上只有一个之前迁移产生的节点，MDS1 由于负载加重，即将启动一次迁移。

5.3.1 迁移步骤

1. 源 MDS 确定迁移对象和迁移目标，一个超载的元数据服务器将利用上文提到的算法选择一个子树。然后从集群管理器获得当前最新的元数据集群负载信息，

选择其中负载较小的一个作为迁移目标。如图 5.3 所示。

2. 利用分布式锁锁定所选择的子树。
3. 压缩发送元数据，源元数据服务器将该子树中所有的文件/目录元数据压缩，包括文件名字、大小、映射关系等都被压缩到数据流中，然后打开一个到目的 MDS 的连接，将压缩后的数据发出。
4. 目的 MDS 收到一个子树的元数据后，逐一将它们恢复，重建树形结构，如果一切顺利，将向源 MDS 报告迁移完成，如图 5.4 所示。
5. 源 MDS 收到报告后，首先通知存有该子树的另一个副本的元数据服务器，更新该子树的映射关系，然后删除源 MDS 上该子树下的所有子孙节点，并将该子树根节点中对应的映射关系修改，如图 5.5 所示。
6. 解锁该子树。

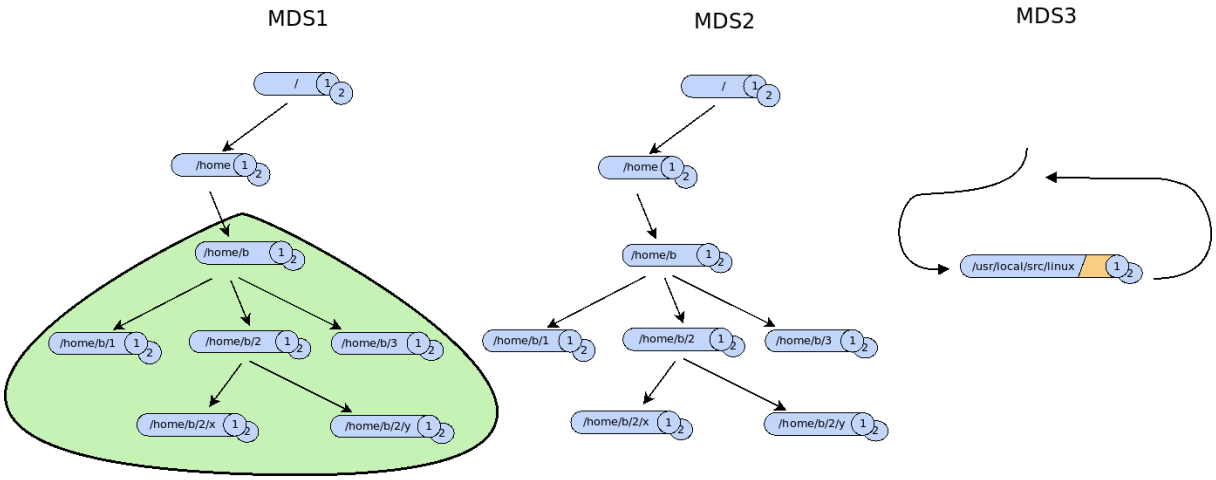


图 5.3 元数据迁移前初始状态，选择迁移对象

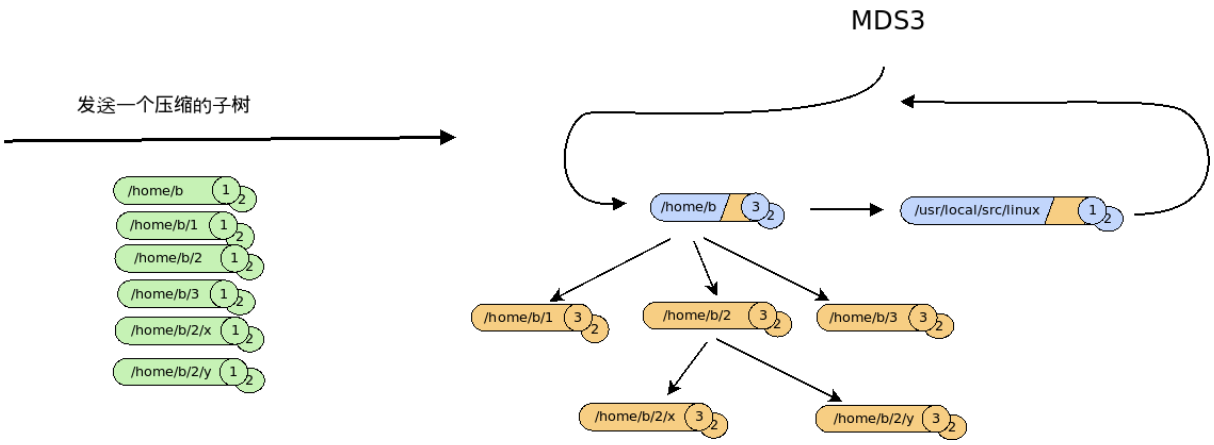


图 5.4 发送压缩子树

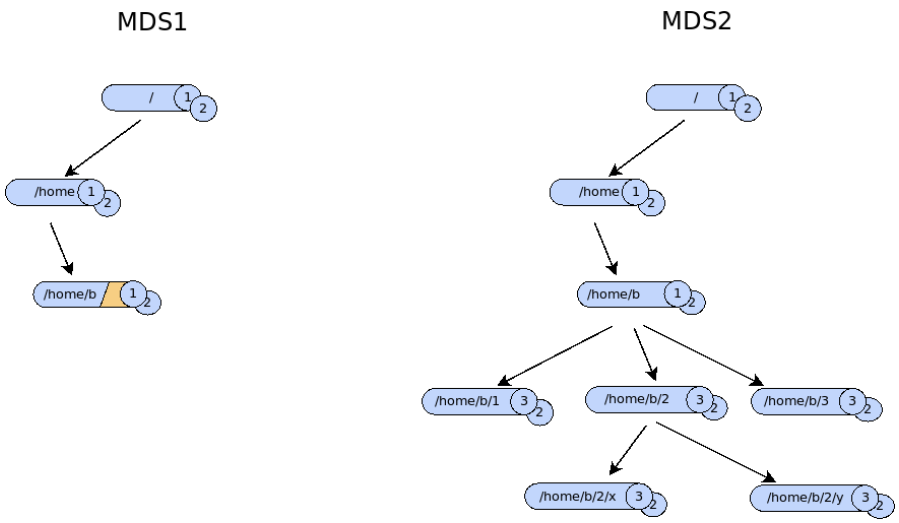


图 5.5 更新相关状态

5.3.2 Client 缓存被动更新

发生了元数据的动态迁移以后，元数据在元数据服务器之间的映射关系改变了，而客户端缓存的元数据的映射关系却没有得到更新。MDS 不可能逐一通知客户端更新自己的缓存，我们将在客户端下一次对该文件的请求中发现自己的 Cache 失效时更新客户端缓存。

此时如果客户端按照旧的映射关系向旧的 MDS 发起请求，该 MDS 会返回一个错误码，要求 Client 重新获取映射关系，Client 收到此信息后，会发起一系列定位请求。下面以假设客户端发起 stat(/home/b/1) 请求为例进行说明，Client 会首先根据 Cache 中/home/b/1 的映射关系，向 MDS1 发送请求，得到的响应为 NOT_FOUND，然后根据其父节点所在的 MDS（仍然为 MDS1），向 MDS1 发起 stat(/home/b)请求。这一次，由于如图 5.6 所示，MDS1 中/home/b 的映射关系已经变为[MDS3,MDS2]，客户端更新自己的 Cache，恢复刚才的查询，向 MDS3 发送 stat(/home/b/1)请求，就可以获得/home/b/1 的元数据，同时完成缓存的更新。

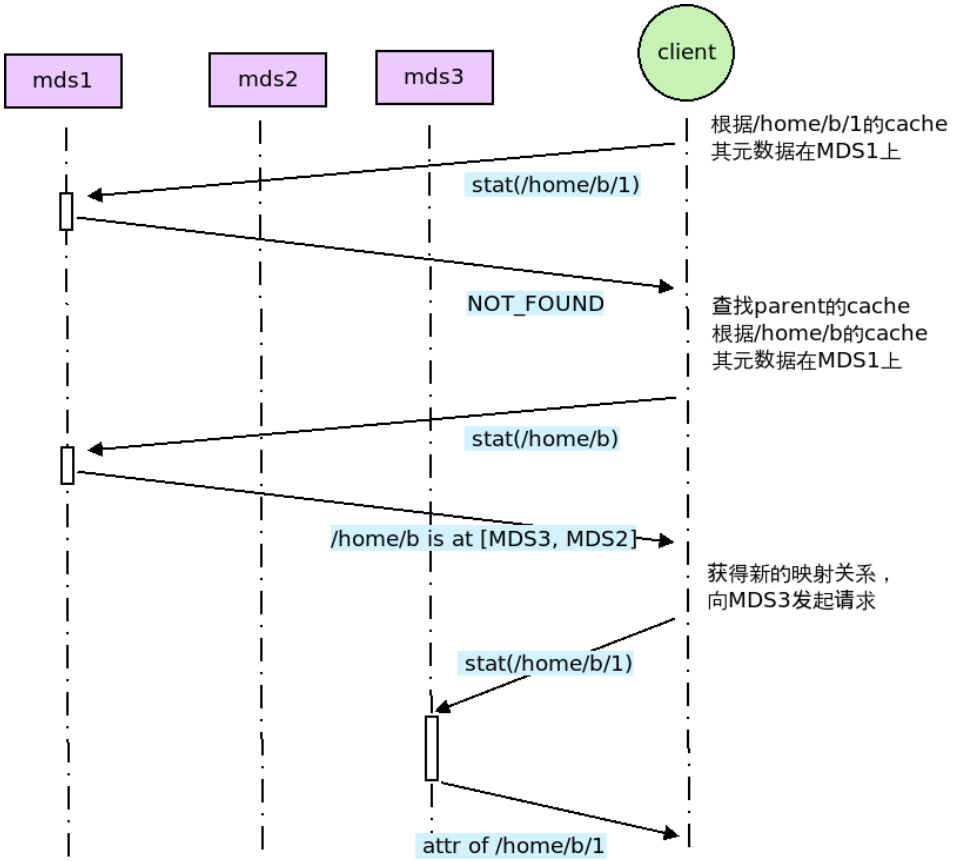


图 5.6 被动 cache 更新策略

5.4 小结

负载均衡在元数据服务器集群中是非常重要的，影响系统性能的重要因素，在本章中，我们介绍了系统使用的负载均衡策略，包括负载的统计，决策的形式，迁移粒度的选择，迁移算法的实现等，我们通过负载的累计来避免突发负载带来的影响，在元数据迁移过程中，我们通过选择合适大小的子树进行迁移，防止过度迁移导致系统抖动，另外，我们通过客户端的被动缓存更新，达到 MDS 和 Client 的缓存一致性。

第六章 性能分析评价

本章将利用现有的文件系统 benchmark 工具，评估我们实现的原型系统中，元数据服务器集群的性能、负载均衡能力，扩展性等。

本章组织如下：第一节介绍几个通用的文件系统测试程序；第二节测试单一元数据服务器处理请求的能力；第三节测试元数据服务器集群的负载均衡能力；第四节测试系统中元数据服务器集群的性能。

6.1 测试工具和测试环境

6.1.1 文件系统 benchmark 工具介绍

针对文件系统 benchmark 的工具有很多，它们的侧重点、测试方法、结果表示都各有不同，下面将重点介绍 iozone 和 Postmark。

Iozone:

Iozone[37]主要用于测试文件系统的读写效率，缓存效率，吞吐率等，可以用来模拟多种不同的硬盘访问模式。它的配置非常丰富，可以比较准确的模拟特定负载。可以配置使用不同的块大小，使用同步/异步方式读写文件等。

使用 iozone 进行性能测试，需要注意 cache 对文件系统性能的影响，对 ext3 文件系统进行测试的时候，可以看到文件系统读写速度达 6GB/s，这显然是不可能的，实际上数据并没有写到硬盘，而是写到了系统缓存，所以在测试过程中，要用大于系统内存的测试文件，才能体现真正的文件系统性能。

Fileop:

Fileop 是 iozone 的一个实用测试工具，主要用于测试文件系统的元数据性能。它会创建很多小文件，然后对这些小文件进行 chdir、rmdir、open、read、write、close、stat access、chmod、readdir、delete 等操作，最后输出每种操作的平均/最好/最差性能（单位为 op/s）。本文将使用 Fileop 的部分功能测试文件系统创建文件的性能。

Postmark:

Postmark[38]是由著名的 NAS 提供商 NetApp 开发，用来测试其产品的后端存储性能。主要用于测试文件系统在邮件系统或电子商务系统中性能，这类应用的特点是：需要频繁、大量地存取小文件。

Postmark 首先将创建很多文件，文件的数量和最大、最小长度可以设定，然后对这些文件进行一系列的事务（transaction）操作，比如读/写/append/setattr 等，在某些情况下，文件系统的缓存策略可能对测试结果造成影响，Postmark 可以通过调整 create/delete/ read/write/append 操作的顺序的比例来抵消这种影响，事务操作进行完毕后，Postmark 删除所有文件，输出结果。输出结果中比较重要的输出数据包括测试总时

间、每秒钟平均完成的事务数、在事务处理中平均每秒创建和删除的文件数，以及读、写的平均传输速度。

6.1.2 测试环境介绍

在本文中，我们共使用了 4 台 Linux 服务器进行测试（在每台服务器上，我们会启动多个 MDS 进程，每个 MDS 进程代表一个 MDS 服务器），服务器的配置如下：

CPU：Intel(R) Xeon(R) CPU E5410 @ 2.33GHz（8 核）

内存：4G

操作系统： Linux version 2.6.32-25-generic

网络：千兆以太网

6.2 单一元数据处理性能测试

本节重点测试单一元数据服务器处理请求的能力,在单一元数据服务器性能测试中，我们关闭 MDS 负载均衡，用多个 Client 对单一 MDS 发起请求。测试环境如下：

系统部署两个元数据服务器，元数据的两个备份会分别存放在这两个 MDS 上，分别用 1、2、4、8 个客户端对 MDS 服务器发起持续请求。

测试所用的命令为：

fileop -l1 -u22 -s0

（注意，这里是用的 fileop 经过我们修改，不再进行 stat 等操作，只进行 mkdir 和 create 操作，便于我们观测 MDS 的性能）

该命令将不断创建文件，并统计每秒创建文件个数，下表是测试结果，表中每个数字表示平均每个客户端观察到的每秒创建文件数。

表 6.1 多客户端测试中每个客户端观测到的平均性能

请求个数	1 个 Client	2 个 client	4 个 client	8 个 client
1000	3108	2433	1133	529
1331	3074	2176	1087	534
1728	3074	2590	1105	526
2197	2748	2282	1096	538
2744	3006	1866	1048	534
3375	3100	2548	1008	529
4096	3058	2531	979	533
4913	3108	2430	1111	521
5832	3079	2390	1132	524
6859	3034	2381	1008	527
8000	3080	2310	1002	526
9261	2938	2411	1022	518
10648	3047	2524	1012	521

上表中，该 MDS 实际承担的负载为所有客户端观测到的负载之和，所以，该 MDS 实际承受的负载如下表所示：

表 6.2 在多客户端测试中，所有客户端观测到的性能之和

请求个数	1 Client	2 个 Client 负载之和	4 个 Client 负载之和	8 个 Client 负载之和
1000	3108	4812	4528	4273
1331	3074	4405	4380	4318
1728	3074	5252	4510	4239
2197	2748	4599	4404	4320
2744	3006	4315	4299	4303
3375	3100	5078	4084	4272
4096	3058	5087	3943	4290
4913	3108	4956	4464	4205
5832	3079	4754	4568	4222
6859	3034	4815	4024	4247
8000	3080	4650	4028	4212
9261	2938	4880	4092	4188
10648	3047	5073	4044	4211

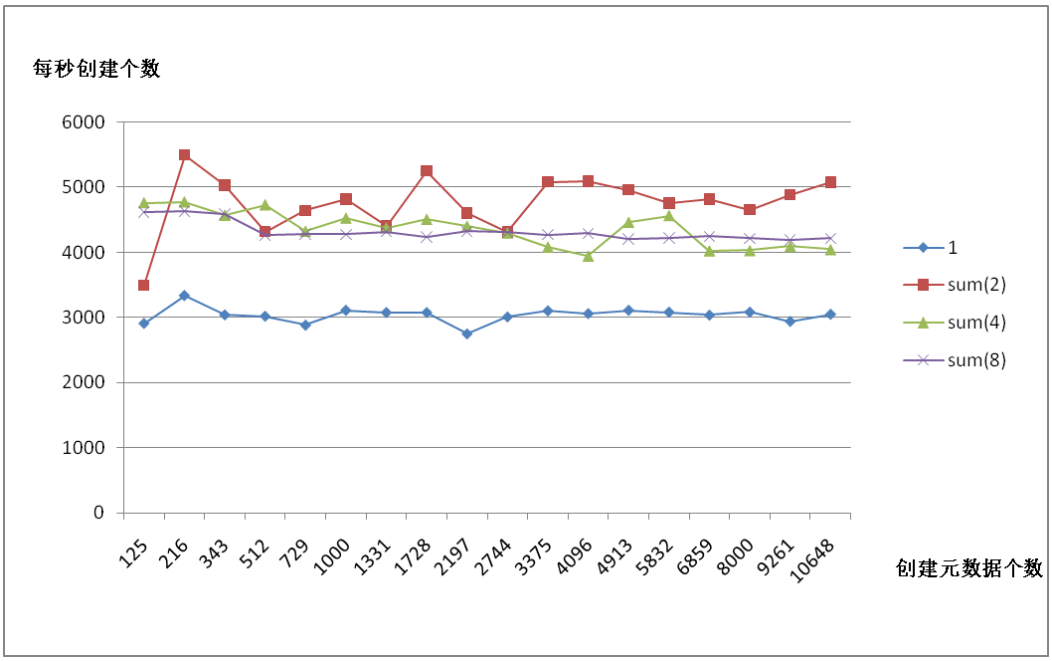


图 6.1 单一元数据服务器性能测试分析

当只有一个 Clinet 时，单一客户端能获得 3000 op/s 的性能，当有两个客户端时，每个客户端能获得 2500op/s 的性能，当有 4 个客户端时，每个客户端获得的性能为 1100op/s，当有 8 个客户端时，每个客户端获得 520op/s，随着 client 数目的增多，虽然每个 Client 获得的性能有所下降，但是将所有客户端的负载聚合后，该 MDS 的处理能力并没有下降，而是相对稳定，峰值达到 5000op/s，如图 6.2 所示，由该图可以看出，单 MDS 的性能不会随着客户端的增多而降低。

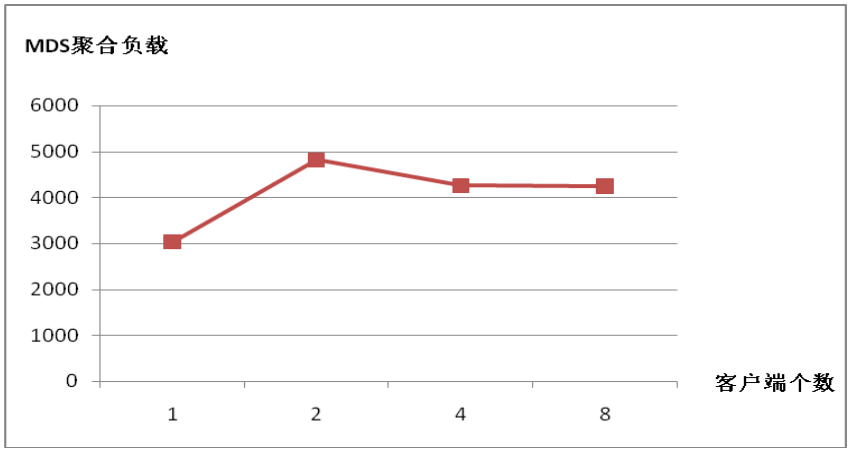


图 6.2 单一元数据服务器对多客户端性能

6.3 元数据服务器负载均衡测试

为了测试元数据服务器集群的负载均衡能力，我们将使用三个元数据服务器，MDS1、MDS2、MDS3，系统初始化时时刻，文件系统的根在 MDS1 和 MDS2 上。然后我们启动一个客户端，使用命令 fileop -l15 -u25 -s0 持续的创建文件和目录。

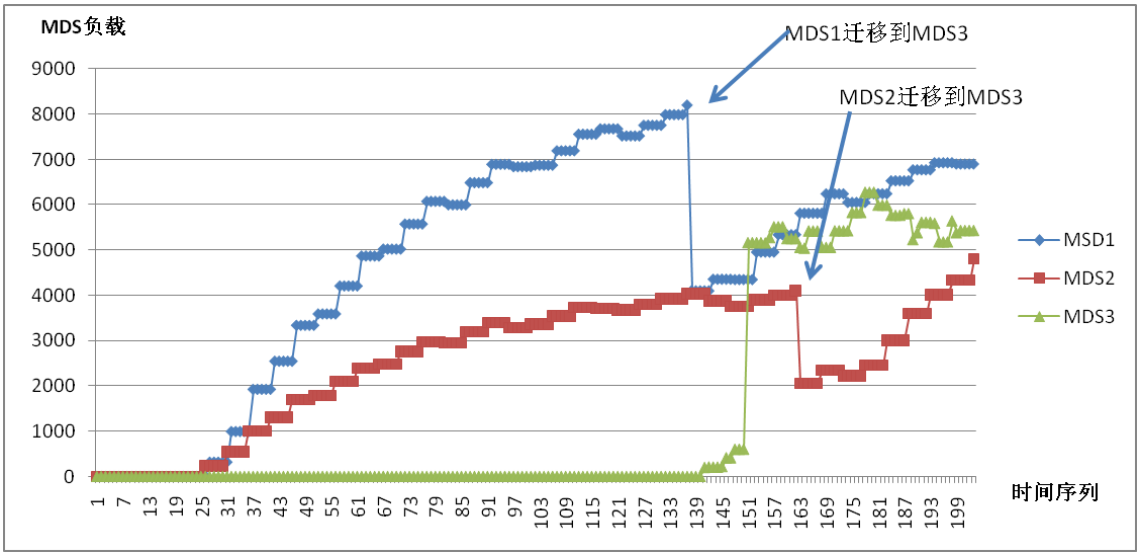


图 6.3 负载均衡测试结果

测试结果如图 6.3 所示，系统刚开始时所有的负载都集中在 MDS1 和 MDS2 上，经过一段时间后，MDS1 的负载到达阈值，开始启动一次迁移，迁移将选择目前负载最轻的 MDS3 为目标，从图中可以看到，经过这次迁移，MDS1 的负载有所下降，同时 MDS3 的负载急剧上升，开始承担文件创建和访问的元数据操作。最终，系统中每一个 MDS 的负载趋向于平均，说明我们的负载均衡是有效的。

图 6.4 是一个较长期的负载均衡监控结果，测试用例是先通过 fileop –l15 -u22 -s0 创建文件，任务完成后，经过一段时间，再次发起 fileop –l20 -u25 -s0 ，图中刚开始一段

时间负载集中在 MDS1 和 MDS2 上，并且经常发生迁移，但是迁移后，负载并没有转移到 MDS3 上，这是由于 fileop 的测试过程分为两个阶段：

1) 创建阶段，是采用深度优先的方式创建目录和文件。

2) 删除阶段，对刚才创建的目录逐一删除

在创建阶段，一个目录子树很可能因为过多的创建操作发生迁移，而迁移后，fileop 已经移到另一个目录区创建，因此迁移后的目录可能在短时间内都没能得到使用，而是在删除阶段被集中访问（图 6.4 中时间序列为 481 的点正是删除阶段的表现），在第二次发起的 fileop 中，MDS1 首先承受大量负载，然后把一个较大的目录迁移到了 MDS3（把处于创建阶段较高层次的一个目录迁移到 MDS3），之后元数据负载在 3 个 MDS 上相对均衡。

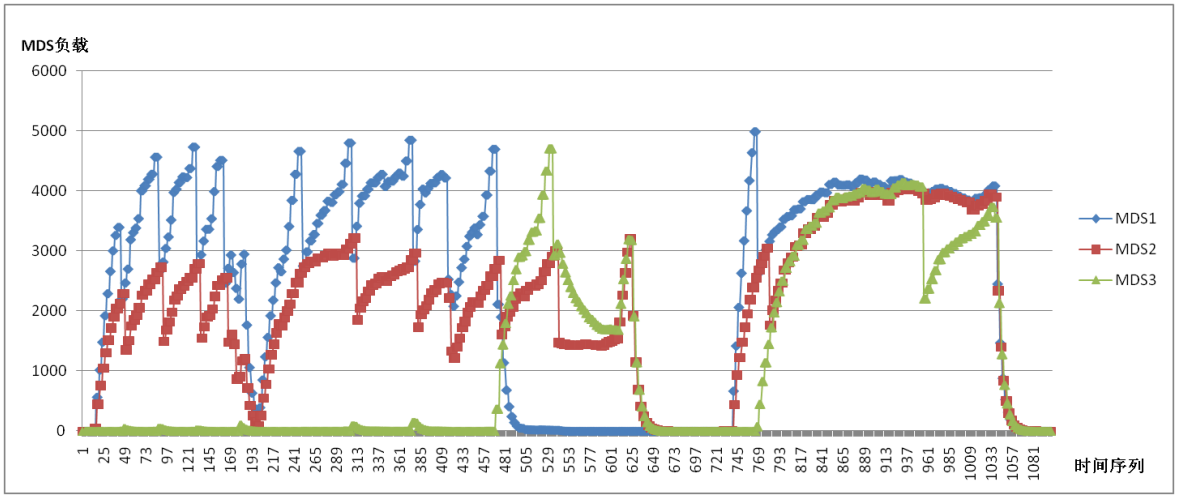


图 6.4 负载均衡测试结果-长期

图 6.5 是对含有 5 个元数据服务器的元数据集群进行负载监控的结果，从图中可以看出，在不同时刻，不同的元数据服务器承担负载的情况不同，总体来说，每个元数据服务器都有负载重的时刻和负载轻的时刻，负载在所有 MDS 服务器上分布较为均匀。

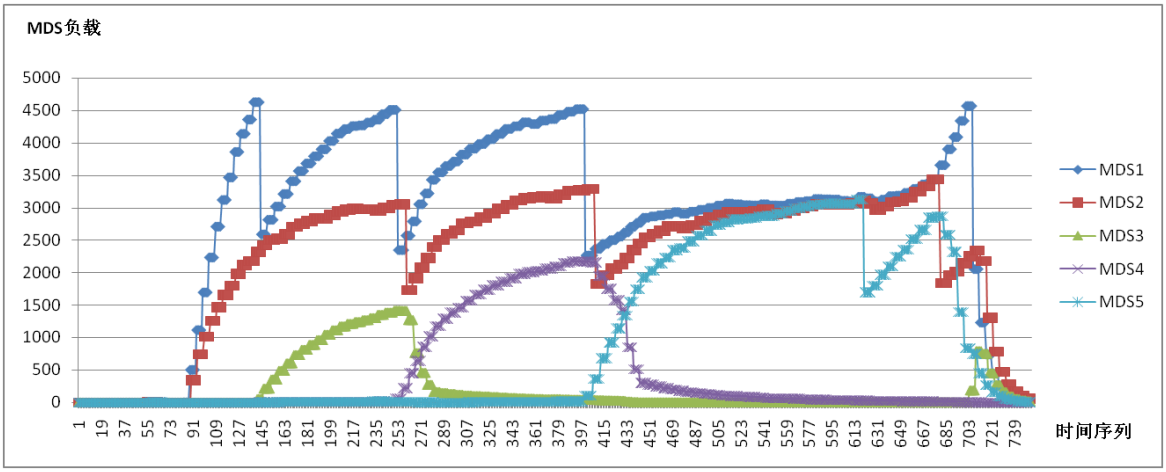


图 6.5 5 个 MDS 之间负载均衡测试

由于条件限制，上面测试均是在系统负载较轻的情况下测得，客户端数量为 1~8 个，

如果系统负载较重，有很多客户端的话，每个元数据服务器的性能应该更加均衡。

通过对元数据服务器集群的负载均衡能力测试，说明我们的系统具有很好的元数据负载均衡能力。

6.4 元数据集群整体性能测试

本节中，我们将测试系统在多 MDS 环境下能够提供的元数据处理能力，我们开启 MDS 负载均衡功能，用多 Client 对多个 MDS 发起请求。

我们将部署 2、4、8 个元数据服务器，然后分别用 2、4、8 个客户端对 MDS 服务器集群发起持续请求，初始情况下，root 的元数据存放在 MDS1/MDS2，最终统计每个客户端观察到的系统每秒创建文件个数。

当使用 2 个 MDS，2 个客户端时，测试结果如下表所示，表中每个数字表示平均每个客户端观察到的每秒创建文件数：

表 6.3　2MDS/2Client 时每个客户端观察到的性能

client1	client2	sum
2263	2060	4323
2262	2117	4379
2259	2160	4419
2253	2060	4313
2253	2119	4372
2251	2070	4321
2251	2115	4366
2163	2047	4210
2091	2124	4215
2083	2063	4146
2073	2017	4090
2072	2024	4096
2035	2078	4113
2030	2038	4068
2031	2101	4132
2129	2338	4467
	聚合性能：	4251.875

当使用 4 个 MDS，4 个客户端时：

表 6.4　4MDS/4Client 时每个客户端观察到的性能

client1	client2	client3	client4	sum
1581	1475	1262	1719	6037
1688	1656	1221	1548	6113
1638	1496	1191	1506	5831
1598	1518	1243	1707	6066
1593	1550	1203	1667	6013
1676	1518	1223	1678	6095
1665	1509	1252	1650	6076
1661	1472	1184	1697	6014
1597	1475	1299	1608	5979
1617	1482	1198	1579	5876
1610	1458	1307	1680	6055
1603	1493	1206	1670	5972
1498	1338	1467	1676	5979
1525	1356	1366	1678	5925
1516	1414	1509	1669	6108
1548	1408	1996	1628	6580
			聚合：	6044.94

当使用 8 个 MDS，8 个客户端时：

表 6.5 8MDS/8Client 时每个客户端观察到的性能

client1	client2	client3	client4	client5	client6	client7	client8	SUM
1414	993	1093	963	891	1325	1163	1203	9045
1372	946	1080	968	1194	1375	1141	1191	9267
1459	1091	1157	955	1196	1387	1105	1242	9592
1351	1027	1067	1165	1206	1299	1100	1284	9499
1422	1016	1118	1167	1253	1343	971	1270	9560
1461	1046	1086	941	1256	1492	970	1264	9516
1390	999	548	1033	1204	1487	957	1267	8885
1399	1006	1060	900	980	1345	973	1275	8938
1370	1027	1045	607	972	1358	972	1245	8596
1380	998	647	1029	930	1322	956	1244	8506
1326	1004	995	1034	927	1319	960	1266	8831
1330	1040	931	985	958	1323	919	1261	8747
1334	1013	1125	949	967	1382	927	1273	8970
1311	994	1114	954	951	1391	898	1311	8924
1293	980	915	964	930	1286	813	1396	8577
1146	976	1042	950	938	1307	882	1252	8493
							聚合	8996.63

综合以上测试，我们得出系统在 2、4、8 个 MDS 的情况下能够提供的聚合性能如下图所示：

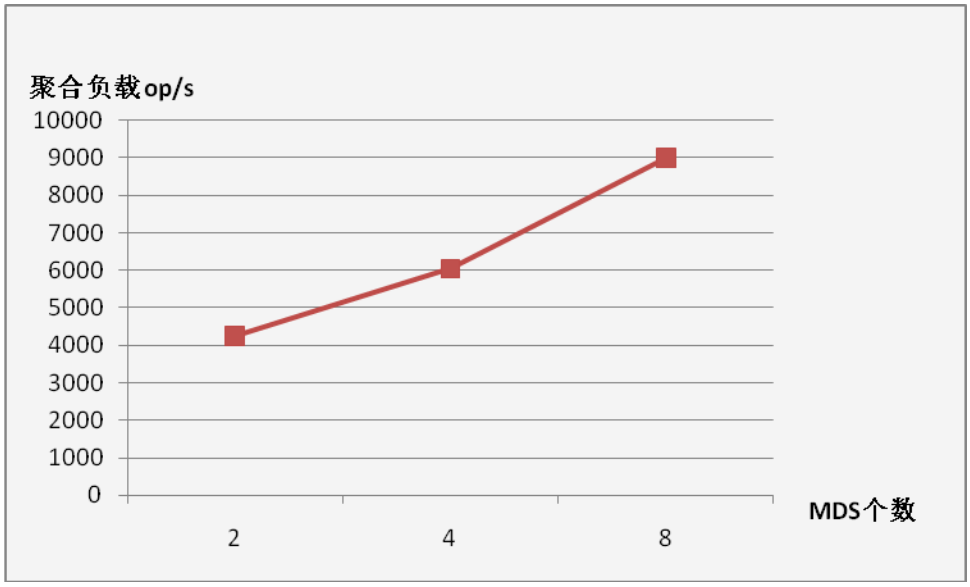


图 6.6 多 MDS 的元数据的聚合性能

由上图中可以看出，使用更多的元数据服务器，能够提供更好的元数据处理性能。但是当元数据服务器个数增多时，平均每个元数据服务器能够提供的处理能力下降，如下图所示，这主要是由于负载统计和元数据迁移带来的开销。

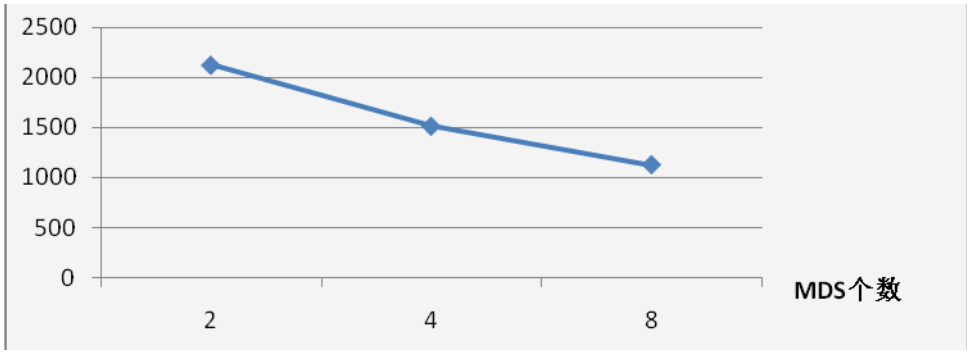


图 6.7 多 MDS 情况下单个 MDS 提供的性能

6.5 小结

本章从单一元数据服务器性能、负载平衡能力和元数据集群性性能三个方面对我们的原型系统进行了测试，表明我们的元数据集群能够有效提供更好的性能和较好的扩展性，能有效地解决文件系统中的不确定性和突发性。

第七章 结束语

基于对象的存储系统作为当前云存储的主流体系结构，已经取得了广泛的应用。它一方面通过元数据与数据的分离，提供高性能，高容量的存储系统，另一方面，通过基于对象的设计，使系统具有较好的扩展性。如何应对对象存储系统中大量元数据的管理问题，是当前对象存储系统的巨大挑战，本文所做的工作就是围绕这个问题展开的。

7.1 本文工作总结

本文结合联想网盘这样的云存储服务的需求，针对基于对象的存储系统中单一元数据服务器带来的问题，设计实现了一个支持元数据服务器集群的分布式文件系统原型，能够利用元数据服务器集群，避免元数据服务器成为系统单点，通过将文件系统的元数据请求负载分布到多个元数据服务器，提供更好的元数据处理性能，同时保证元数据的冗余。

本文取得的主要研究成果和创新表现在以下几个方面：

1) 设计实现了一种元数据集群动态管理方案，利用心跳机制和基于版本的集群管理技术，实现了元数据节点的动态加入和删除。

2) 利用动态子树划分算法实现了元数据动态分布，获得了较好的负载均衡能力，扩展能力和可靠性。

3) 提出了一种元数据迁移粒度的选择算法，通过选择合适的迁移子树，防止过度迁移导致系统抖动，并基于该迁移算法实现了元数据集群的负载均衡。

此外，在具体实现中，我们通过负载的累计来避免突发负载带来的影响，利用客户端的被动缓存更新，实现 MDS 和 Client 的缓存一致性。

利用以上研究成果，我们已经实现了一个基于对象的存储系统原型，相对也我们的原有系统，其元数据处理能力和扩展性得到了很大的提高。

7.2 下一步研究方向

在我们的原型系统中，我们实现了元数据的分布式管理，初步评估结果表明，使用动态子树划分算法能够较好的实现元数据的动态分布，达到了预期的目的，但是本研究工作主要关注元数据的分布算法和负载均衡算法，主要集中在元数据的分布方法、元数据集群的维护、元数据的动态迁移，负载均衡等方面，而在系统容错性，稳定性等方面并没有做深入的研究和测试，因此，我们还需要做很多工作。

1．容错性：我们的工作主要考虑元数据在集群中分布的方式和负载均衡的算法，而没有考虑当一个 MDS 宕机后，它上面所存放的元数据的恢复机制，此时，我们应该启

动一个恢复过程，将该 MDS 上对应的那些元数据复制到新的 MDS 上，和元数据迁移一样，我们需要注意一致性问题，要保证元数据的两个备份和 Client 端的映射关系一致。

2．Client 端优化：在我们的测试中，系统的瓶颈不是元数据服务器的处理能力，而是客户端的性能，我们目前对此还没有太多的关注，今后需要利用更好的数据结构，通过多线程等机制，来提高 Client 的效率。

3．大规模下的测试：在我们的原型系统实现过程中，我们仅对较少 MDS，较少 OSD 的情况做了测试，在未来的工作中，还需要从理论和实际环境两方面验证系统在大规模系统中的负载均衡能力，性能，扩展性等，并根据结果对系统进行优化。

参考文献

[1] M Mesnier, GR Ganger. 2003 "Object-based storage". Communications Magazine, IEEE Volume: 41 Issue:8 On page(s): 84 - 90

[2] "Amazon. Simple storage service". https://s3.amazonaws.com/

[3] "Google Storage". http://code.google.com/apis/storage/

[4] "Dropbox". http://www.dropbox.com/

[5] "Box.net". http://box.net/

[6] "联想网盘". http://www.vips100.com

[7] P. J. Braam. The Lustre storage architecture. Cluster File Systems. Inc. Aug. 2004.

[8] "MooseFS". http://www.moosefs.org/

[9] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith. Andrew. A distributed personal computing environment. Communications of the ACM, 29(3):184–201. Mar. 1986

[10] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda. A highly available file system for a distributed workstation environment. IEEE Transactions on Computers, 39(4):447–459, 1990

[11] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The Sprite network operating system. IEEE Computer, 21(2):23–36, Feb. 1988

[12] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph. A scalable, high-performance distributed file system. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI), Seattle, WA, Nov. 2006. USENIX

[13] J. Ousterhout, Da.Costa, H. Harrison, J. Kunze, M. Kupfer and J. Thompson., "A Trace-Driven Analysis of the Unix 4.2 BSD File System," In Proceedings of the 10th Symposium on Operating Systems Principles, December 1985

[14] "Network-attached storage". http://en.wikipedia.org/wiki/Network-attached_storage

[15] "Storage area network". http://en.wikipedia.org/wiki/Storage_area_network

[16] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. "NFS version 4 protocol". RFC 3010, Network Working Group, December 2000

[17] P. Leach and D. Perr. CIFS: A common internet file system. Microsoft Interactive Developer, Nov 1996

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. the 19th ACM Symposium on Operating Systems Principles (SOSP '03), Bolton Landing, NY, Oct. 2003. ACM

[19] NAGLE.D., SERENYI.D, MATTHEWS.A. The Panasas ActiveScale storage cluster—delivering scal-able high bandwidth storage. In Proceedings of the 2004 ACM/IEEE Conference on Supercomputing. 2004

[20] DECANDIA.G, HASTORUN.D, JAMPANI .etc. Dynamo: Amazon's highly available key-value store. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (2007), ACM Press New York, NY, USA, pp. 205–220

[21] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev., 44(2):35–40, 2010

[22] "Server Message Block" http://en.wikipedia.org/wiki/Server_Message_Block

[23] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In Proceedings of the First USENIX Conference on File and Storage Technologies, pages 231–244, Monterey, California, January 2002

[24] S. R. Soltis, T. M. Ruwart, G. M. Erickson, K. W. Preslan, and M. T. O'Keefe. The Global File System. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, High Performance Mass Storage and Parallel I/O: Technologies and Applications, pages 10–15. IEEE Computer Society Press and John Wiley & Sons, 2001

[25] libevent http://monkey.org/~provos/libevent/

[26] Esteban Molina-Estolano, Carlos Maltzahn. Haceph:Scalable Metadata Management for Hadoop using Ceph

[27] S. A. Brandt, L. Xue, E. L. Miller, and D. D. E. Long. Efficient metadata management in large distributed file systems. In Proceedings of the 20th IEEE / 11th NASA Godard Conference on Mass Storage Systems and Technologies, pages 290–298, Apr. 2003.

[28] 黄华. 蓝鲸分布式文件系统的资源管理[D]. 中国科学院研究生院（计算技术研究所）. 2005 年

[29] 杨德志，黄华，张建刚，许鲁. 大容量、高性能、高扩展能力的蓝鲸分布式文件系统. 计算机研究与发展. 第 42 卷 第 6 期, 2005:1028-1033

[30] 黄华,张建刚,许鲁;蓝鲸分布式文件系统的分布式分层资源管理模型[J];计算机研究与发展;2005 年 06 期

[31] 杨德志.分布式文件系统可扩展元数据服务关键问题研究[D]. 中国科学院研究生院（计算技术研究所）, 2008 年

[32] "Handy". http://grid.hust.edu.cn/handy/

[33] 李胜利，陈谦，程斌等. 一种集群文件系统元数据管理技术. 计算机工程与科学.2006

[34] "Virtual file system". http://en.wikipedia.org/wiki/Virtual_file_system

[35]S. A. Weil, K. T. Pollack, S. A. Brandt, and E. L. Miller. Dynamic metadata management for petabyte-scale file systems. In Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04). ACM, Nov. 2004.

[36] 熊劲. 大规模机群文件系统的关键技术研究 [D]. 中国科学院研究生院（计算技术研究所）, 2006

[37] Iozone filesystem benchmark. http://www.iozone.org/

[38] J Katcher. Postmark: A new file system benchmark. communities-staging.netapp.com. 1997

致 谢

非常荣幸自己能进入联想研究院学习，成为联想研究院数据管理技术部的一员，在联想研究院将近一年半的学习和生活中，自己的学术能力和技术水平有了很大的提高，能够参加联想网盘相关的一系列研发，使自己得到在学校所不能获得的宝贵工业界开发经验，对我来讲更是一笔宝贵的人生经历。

首先要深深地感谢我的导师杜晓黎研究员，他敏锐的洞察力、渊博的知识面、工作的执着和热情使我受益终生。是他把云存储这个热点的问题带给整个团队，让大家有机会接触并深入到世界前沿的领域中，杜老师为为我们提供了积极的研究氛围，他待人和蔼真诚，尽力为学生提供最好的发展空间，这种难得的品质和能力使我们对他充满敬佩和感激。

感谢联想研究院的侯紫峰老师、韦卫两位老师。他们在我毕业论文的写作过程中，从最初选题，到关键点研究，再到最后的写作，都给了我很多指导，提出很多有益的建议。

感谢联想研究院数据管理技术部的每一位成员，你们是我的导师，生活的伴侣，学习的目标。

感谢一起在联想研究院学习的彭湃，孙羽，感谢他们平时对我的鼓励、关心和支持。有了他们，我的学习生活才一会丰富多彩，他们是我研究工作中灵感的源泉。

感谢我的家人，他们虽然身在远方，却时刻关心着我的生活、我的健康。他们无私的爱是我工作中的无限动力。

最后，谨向所有关心和帮助过我的人致以衷心的谢意！

作者简介

姓名：杨林　　性别：男　　出生日期：1986.05.15　　籍贯：云南曲靖

2008.9 – 2011.7　　中科院计算所计算机系统应用专业硕士
2004.9 – 2008.7　　华中科技大学计算机专业本科生

【攻读硕士学位期间发表的论文】

无

【攻读硕士学位期间参加的科研项目】

[1] 联想手机网盘，2009 年 8 月~2009 年 12 月
[2] 联想网盘多点部署项目，2010 年 1 月~2010 年 10 月
[3] 联想网盘服务监控平台，2011 年 1 月~2011 年 3 月

【攻读硕士学位期间的获奖情况】

无