武汉理工大学

硕士学位论文

# 面向数据存储的分布式文件系统的研究与设计

姓名：张云升

申请学位级别：硕士

专业：计算机应用技术

指导教师：钟珞

201206

# 摘要

自从互联网诞生以来，尤其是网络应用在最近十年的迅猛发展，很多公司的互联网上的业务数据呈现爆炸性增长的态势，这些数据内容涉及了诸多领域，是公司发展必不可少的保证。继互联网热潮之后，数据存储已以成为又一次技术浪潮，网络已进入数据中心时代，数据已经越来越重要，不段增加的用户，越来越多样化的网络服务，迅速发展的多媒体，都使数据处理的难度不段地增大。爆炸性增长的信息资源，对存诸系统的各方面性能提出了越来越高的要求，传统的文件系统满足不了现代应用对高可靠性高可用性易维护性可扩展性高性能以及大容量等要求。

文件系统是常见数据存储的方式，是操作系统在硬盘上存诸，组织和检索数据的方式，硬盘可以是存在于本地的，也可以是存在于网络上的。文件系统总体上分为本地文件系统和分布式文件系统。本地文件系统不需要网络连接即可访问数据，文件系统的数据存放在本地设备上。分布式文件系统管理的存储资源不一定直接在本地设备上，而是通过网络与各个节点相连，它系统中的数据可能来自很多不同只的节点，它管理的数据也有可能存储在网络中不同的节点上。分布式文件系统中的很多设计实现与本地文件系统有很大的差别，分布式文件系统的设计和实现更复杂。

本文就是通过分析对比各种分布式文件系统，吸取各个文件系统的优点，设计出了一个面向海量数据的分布文件体系统。第一章绪论，针对分布式文件件系统的研究现状（包括各种分布式系统的介绍）和研究背景作了详细的介绍，最后给出了本文主要内容。第二章分布式文件系统架构，首先分析了需求并给出了本系统的设计目标，最后出了本文件系的整个架构。第三章元数据系统，先分析元数据系统设计目标，接着给出元数据的分割策略，负载平衡策略以及数据一致性策略。第四章数据副本，介绍了请求的分发策略和副本的放置策略。第五章功能测试，介绍了对文件系统各个模块的测试用例。

关键词：分布式文件系统 元数据 海量数据

# Abstract

Since the birth of the Internet, especially the rapid development of network applications in the last ten years many companies on the Internet, business data showing the trend of explosive growth.These data covering many fields, is essential for the development of the company's guarantee. Users following the Internet boom, data storage has become yet another wave of technology, the network has entered the era of data center, data has become increasingly important, not of increase, more and more diverse network services, the rapid development of multimedia, both increases the difficulty of processing the data. The explosive growth of information resources on all aspects of performance that kept all system requirements, the traditional file system can not meet the modern applications of high reliability, high availability, ease of maintenance, scalability, performance and The large capacity requirements.

File system is a common way of data storage, and various operating system on the hard disk memory organization and retrieval of data.The hard disk can exist at the local or the network. Overall file system into the local file system and distributed file system. Local file system does not require a network connection to access the data, and file system data stored on the local device. Distributed File System management of storage resources on the local device are not necessarily directly, but through the network and each node is connected to the data may come from many different only node in its system, it manages data may also be stored in the network node. Distributed File System Design and Implementation of a great difference to the local file system, distributed file system design and implementation of more complex.

By analyzing and comparing a variety of distributed file system, to learn the advantages of each file system, I devised a distribution for massive data files-body system. The first chapter introduces the research background and the research status of the distributed file system, and gives the main content of this article. The second chapter of the distributed file system architecture, the first analysis of the needs and gives the design goals of the system, and finally the whole structure of this document. Chapter III data system, the first analysis of the metadata system design goals, and then gives the meta-data partitioning strategies, load balancing strategy and data

consistency strategies. The fourth chapter of the data a copy of the request distribution strategy and a copy of the placement strategy. Chapter functional testing, test cases for each module of the file system.

**Key words:** Distributed file system Meta data Huge amounts of data

# 第一章 绪论

## 1.1 研究背景

自从互联网诞生以来，尤其是网络应用在最近十年的迅猛发展，很多公司的互联网上的业务数据呈现爆炸性增长的态势，这些数据内容涉及了诸多领域，是公司发展必不可少的保证。继互联网热潮之后，数据存储已以成为又一次技术浪潮，网络已进入数据中心时代，数据已经越来越重要，不段增加的用户，越来越多样化的网络服务，迅速发展的多媒体，都使数据处理的难度不段地增大。爆炸性增长的信息资源，对存诸系统的各方面性能提出了越来越高的要求，传统的文件系统满足不了现代应用对高可靠性高可用性易维护性可扩展性高性能以及大容量等要求。

文件系统是常见数据存储的方式，是操作系统在硬盘上存诸，组织和检索数据的方式，硬盘可以是存在于本地的，也可以是存在于网络上的。文件系统总体上分为本地文件系统和分布式文件系统。本地文件系统不需要网络连接即可访问数据，文件系统的数据存放在本地设备上。分布式文件系统管理的存储资源不一定直接在本地设备上，而是通过网络与各个节点相连，它系统中的数据可能来自很多不同只的节点，它管理的数据也有可能存储在网络中不同的节点上。分布式文件系统跟本地文件系统在设计与实现的很多方面都存在着巨大的差异，分布式文件系统的设计和实现更复杂。

伴随着互联网的飞速发展，网络、服务器、存储等各方面技术都得到了快速发展。IT 三大基础设施（存储、传输、计算）都得到了很大的发展，但是相对于其它两项，存储技是相对发展较缓慢的，主要是 I/O 子系统没有相应地得到提高，I/O 子系统成为存储系统的瓶颈。因此，提高存储系统的 I/O 子系统有着重要的意义，对于这方面的研究已经得到了学术界以及工业界等各界的广泛关注。分布式文件系统不论是在数据的存储容量可扩展性方面还是在可靠性和可用性等方面都有很大的优势，分布式文件系统可以显著提高存储容量，有效地解决海量数据的存储问题，显著提高 I/O 性能。分布式文件系统可以轻易地提供 PB 级的数据存储，目前普遍被认为是应对海量数据存储以及高效快速访问的有效方法。因此，对分布式分件系统的研究非常具有科研价值和现实意义。目前许多领域都对海量数据存储存在需求，例如宇宙航空航天，医学影

像，地理信息系统，高分子材料，海洋信息，精密农业，石油勘探，三维虚拟人，哈勃望远镜等有有 TB 甚到 PB 级的个子储容量需求。越来越大的存储需求使信息存储从传统的的附属地位独立出来，成为 IT 领域不可或缺的重要组成部分。

分布式文件系统大都采用可扩展体系结构，多台存储服务器分担了待存储的数据量，不仅突破了服务器容量这一性能瓶颈，还可有效地避免服务器间点失所造成的服务可靠性下降。网络技术的发展也为分布式存储系统的建立奠定了良好的基础，分布式存储技术的发展是建军立在网络技术飞速发展的基础之上的。以太网已经从最初的 10Mbps 网络发展到了现如今的千兆以太网万兆以太网，还有其它一些高速网络技术也不段地发展着，例如 ATM[1]，Myrinet[2]，Fiber Channel[3]，InfiniBand[4]等，30Gbps 带宽的网络技术已经出现，网络传输层的各种软件和协议也逐渐发展起来，精简消息通讯协议[8][9][10],用户层通讯协议[5][6][7],VIA[11]等网络接口协议纷纷出台，它们都被设计用来增加网络带宽并且减少络延迟。现在的高速网络的带宽可以与计算机内部总线的带宽相当了，网络传输技术的发展络分布式文件系统发展提供了很好的空间。通过与最新的网络传输技术相结合，可以使网络存储系统的规模扩展得更大，存储能力变得更强，同时数据服务的性能也变得更高。分布式文件系统用来对用户提供统一的虚拟文件接口，存储并管理系统中的信息。

## 1.2 分布式文件系统研究现状

自 1985 年 Sun 公司推出 NFS 文件系统以来，分布式文件系统已经越来越受到大家的重视，在这二十多年的时间里，分布式文件系统在理论、实践中都取得了较大的发展。早期的分布式文件系统都是通过局域网相联的，受到技术水平的限制，人们重点关注它数据的可靠性，数据的可访问性和可扩展性。早期的分布式文件系统包括 NFS[12]，Coda[14]，AFS[13]，Sprite File System[15]等。

20 世纪 90 年代初，由于磁盘存储技术的不断进步，存储成本也随之降低。Windows 操作系统的出现，为微机的普及做出了极大的贡献，伴随着互联网的普及，在网络传输的多媒体及其它类型的数据也逐渐地增加。在大容量应用存储需求不断增加的情况下，加利福尼亚大学（University of alifornia at Berkeley）借鉴了高性能对称多处理器的恩想，设计开发出了 xFS[16]。xFS 突破了之前的分布式文件系统都运行在局域网的局限，也很好地克服了文件系统在

在广域网中的一系列难题，也有效地利用了本地存储空间和本地主机。这时其它的文件系统还有 SFS 和 Frangipani[17]等

20 世纪 90 年代中后期，随着网络技术的进一步发展和普及，网络存储技术也得到了相应的发展，基于 Fiber Channel（光纤通道）的 SAN（存储区域网 Storage Area Storage）和 NAS（网络附连存储 Network Attached Storage）也被广泛地应用。此时，单位存储的成本继续大幅下降，网络技术和计算机技太更是突飞猛进地向前发展，极大地推动了网络存储的发展。多种体系结构都是在这个阶段产生的，它们都对网络技术进行了充分的利用。为了满足对于数据性能、容量和共享等方面的需求，，这时期的分布式文件系统变得更复杂，规模也变得更大，在磁盘布局、物理设备的访问和检索效率等方面也在不断地优化。可扩展性、可靠性，缓存一致性的需求也伴随着规模的扩大随之而来，缓存管理、分布式锁、文件级负载平衡和 Soft Updates 等技术也逐渐地被应用到系统的实现中。

20 世纪 90 年代后期以后，SAN 和 NAS 得到了广泛的应用并逐渐成熟，人们考虑将两种技术结合起来，来充分利用它们的优势。人们对体系结构的认识也随着多种分布式文件系统的发展面不断加深，网格（Grid）的部分研究成果也对分布式文件系统的发展起到了推动作用。这时产生的文件系统有 Sun 的 Lustre[19]，IBM 的 Storage Tank[18]，蓝鲸文件系统（BWFS）[21]，Panasas 的 PanFS[20]等。不段出现的应用对我们的存储系统提出了越来越多的需求：（1）数据量要比以前大得多，而且数据增长速度也在逐渐加快。（2）对数据的访问速度要求更高（3）要保证数据和服务的高可用性（4）现在的系统和应用都处于不断变化之中，我们的系统必须提供良好的扩展性，以适应应用在性能、容量、管理等方面的变化（5）我们的系统随着应用的加深，数据量会变得越来越庞大，存储规模也会飞速地增长，存储系统本身会越发的复杂，系统的管理维护成本会变得很高。（6）存储系统本身要有一定的灵活性，能够按照办需要提供不同的服务。

不断提出的新的需求使人们对体系结构的认识不断地加深，也促进了分布式文件系统在体系结构方面不断向前发展，时至今日已经有很多的分布式文件系统产品了，各个系统在设计上都采用了很多不同的先进技术，在性能方面都各有特点和优势。

### 1.2.1 NFS（网络文件系统）

NFS[22][23][24][25]全称是网络文件系统，它之所以叫这个名字，是因为它允许客户端通过网络连接挂载其他系统的磁盘卷。在这个系统里，为其他机器提代共享磁盘的主机是服务器，挂接服务器磁盘的主机是客户机。客户机可以透明地像对待本地文件系统一样地在服务器提供的共享磁盘上进行诸如创建、删除和修改等操作。它类似于 windows 中的共享文件夹，跟 Samba 也比较地相似，它们都支持系统透明的文件共享，但是不同的是，NFS 被专门的 Solaris 内核支持，，它具 更高的数据吞吐能力。NFS 文件系统是基于远程调用（RPC）的分布式文件系统。NFS 利用远程调用（RPC）技术，可以很方便地实现对服务器系统过程的远程执行请求。目前，RPC 已经可以支持 Solaris，Linux，及 Mirosoft Windows。RPC 可以对通过网络进行过程的的连接方法和细节进行抽象，客户端和服务器就不需要配备专门的网络就可以进行文件共享了。
NFS 是 C/S 结构的分布式文件系统，它使用 Unix 的 VFS 机制，通过该机制，NFS 利用远程过程调用（RPC）和规范的文件访问协议，将客户端的本地请求发送到 NFS 服务器端进行处理；服务器端在收到客户端的请求后，在本地虚拟文件系统之上处理并回得客户端的请求，并将数据存储在服务器本地的文件系统中，从而实现了分布式文件系统全局共享，各用户间共享就是 NFS 文件系统的实质。用户连接好到共享计算机后，就可以像访问本地磁盘一样透明地访问共享计算机上的文件，用户根本感觉不到他们访问的是远程的文件。在 NFS 系统中，拥有实际的物理磁盘，通过 NFS 共享自大己的物理磁盘供其他用户访问的叫做 NFS 文件服务器，通过 NFS 系统访问访问其他用户所共享的磁盘的叫做客户机。一个 NFS 客户机可以同时利用多个 NFS 服务器提供的服务，一个 NFS 服务器也可以同时为多个 NFS 客户机提供化的服务，同时一个共享了部分磁盘的 NFS 服务器还可以当做另一个 NFS 服务器的客户机。原理如图 1.1 所示。

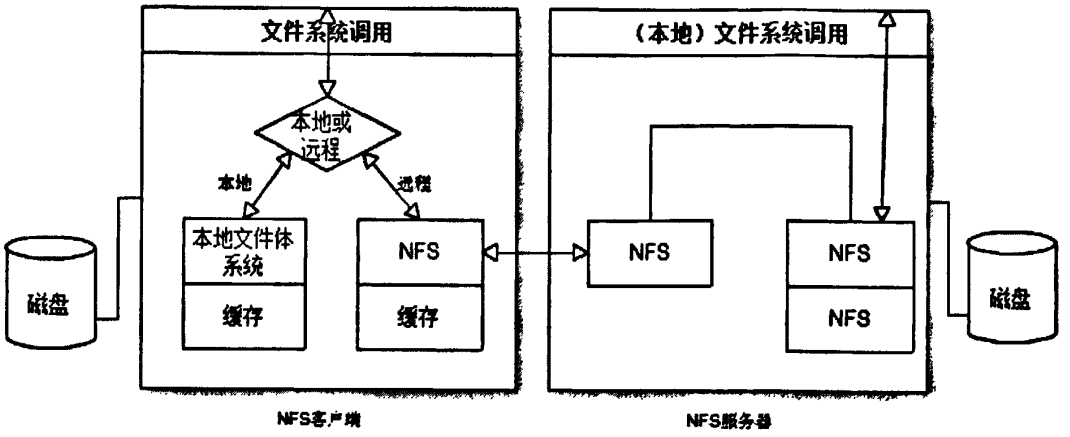



图 1.1 NFS 原理

严格来说，NFS 还并不能算是完整的文件系统，因为它仅仅对文件请求提供了重定向的功能，而 NFS 本身并没有对文件请求做任何的处理工作，而是完全依赖于本地的文件系统完成文件的处理工作。因为 NFS 服务器没有统一的接口，所以对于各个服各器的负载没有控制，很难保证各个服务器之间的负载均衡。NFS 是无状态的文件服务器，及时服务器崩溃也不存在丢失状态信息的问题。但是 NFS 的这种无状态策略会使缓存产生一致性的问题，从而对系统性能产生影响。因为客户使用了哪些缓存，服务器是不知道的，所以服务器无法对缓存进行管理，当某块缓存无效或者过时时，服务器不能给客户相应的通知，这就造成了缓存的不一致。NFS 系统一般都没有服务器或是磁盘的复制功能，当服务器失效时，系统只能停止向客户提供服务。NFS 文件系统在性能，扩展性，管理性以容量方面很难满足现有的应用所提出的要求。

### 1.2.2 AFS

1983 年卡内基梅隆大学（Carnegie Mellon）开发 AFS[26][27]，而后 Transarc 公司接管它的开发，后来 IBM 收购了这个公司，并在 2000 年宣布开放 AFS 的源码。AFS 系统跟 NFS 系统在结构上类似，但是比 NFS 功能能所增强。AFS 解决了很多简单文件系统中的问题，是功能最丰富的分布式文件系统之一。它提供统一的命名空间，客户端增加了检查 Cahe 一致性的功能，共享文件的与位置无关，还增加了通过 Kerberos 进行安全认证的功能。

AFS 的统一命名空间，遵守透明性和位置无关性，不会在文件名中本现出存储数据的位置，并且可以随意地迁移文件而不需要更秘方文件名。AFS 的命名空间是由卷（vloume）组成的，卷将同一用户的文件关联起来，卷可以透明地从一台服器迁移到另一台上，这个迁移的过程对户还是完全透明的，这样系统中每个用户只要一个挂接点就足够了，对于用户量很大的系统来说，这一点是非常重要的。AFS 还提供了良好的扩展功能，它将大量的工作转移到客户端由用户来完成。如图 1.2 中/afs/cpc.com.cn/usr 的挂载点会关联在 usr 卷中。例如用户打开一个文件，文件的内容会全部发送给客户，然后所有的所续操作都是在用户本地的磁盘上完成的，在用户关闭文件时，如果文件做了改变，才将改后的文件发送到服务器进行改新操作。如果用两个用户 1 和 2 先后打开同一个文件，如果用户 1 修改了文件，只用在用户打开文件之前用户 1 已经关闭了文件，文件修改才会生效。AFS 通过运用回调技术，使用户可以不需要访问服务器而打开或关闭同一个文件多次。当用户通过 AFS 打开一个文件的时候会同时收到一个回调来指明该文件的最新版本，回调只在服务器通知版本更新或者用户更新文件时才失效，所以回调如果保持有效就说明本地保存的副本是可以继续使用的。同一个文件会有多个存诸节点保存着文件的备份，对用户是透明的，当一个节点发生故障时，另一个节点会自动地接替它的工作。

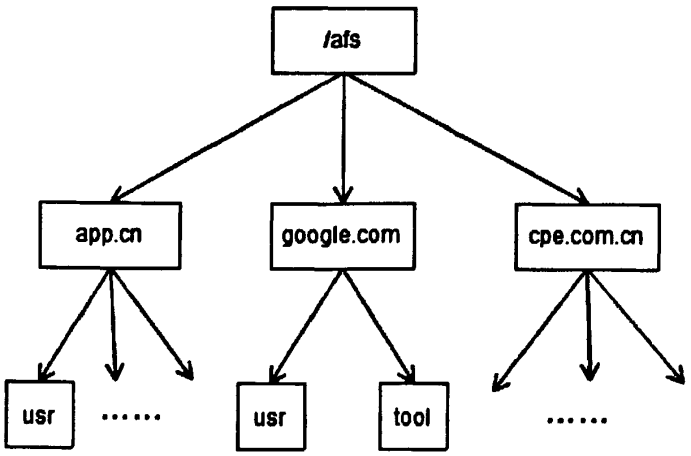



图 1.2 AFS 结构图

### 1.2.3 Coda

1987年卡内基梅隆大学（Carnegie Mellon）以AFS-2作为原型开发出Coda[28][29]文件系统。Coda文件系统继承AFS的部分优点，并对其特性进行一些改进。Coda文件系统可以提供透明的文件接口，各个服务器之间可以复制数据，客户端可以访问到数据的任意一个副本，所以Coda对热点数据的访问性能比较好。Coda能够提供对数据高效、安全的访问，对于企业级的计算机来说，这些需求是很重要的。Coda不仅能直接为用户提供存储服务，这是Coda设计的最初目标，Coda还能为具体的应用充当数据访部层。Coda客户端会缓存服务器上的文件，当无法与服务器连接时，会使用缓存继续进行工作，当重新连接到服务器后，会将副本再写回服务器。在网络恢复后，Coda会检查所有的写冲突并尽可能地解决冲突，当出现无法解决的冲突的时候它会提醒由人工来处理这些冲突。Coda一般是不允许并发现写的，还采用了信号回叫机制，Resolution例程和版本向量机制等多种方法来维护系统缓存后致性。

Coda采用了自己专门设计Venus这种缓存管理器。当VFS发送请求到Venus后，Venus先查看本地硬盘缓存区，如果找到了此文件，就直接使用这个文件，如果在硬盘缓冲区没有找到此文件，则向服务器发送请求，定位到这个文件，并把它放到缓冲区中，这时再用本地硬盘缓冲的方式使用这个文件。Venus为保证重启后客户还能正常的操作，它要求缓存数据的持久性。

大多数的分布式文件系统的服务器端是一个允许被远程客户端调用的文件结构，可以被挂载到远程的客户端。这样的文件系统都是挂载一个远程服务器上的卷，关心的是服务器的目录、分区以及共享。而Coda从本质来说是不一样的，Coda系统的文件并不是存在一般意义的文件系统之上，Coda系统将服务器及工作站分区上的可用部分按照卷的方式来存储管理文件。

### 1.2.4 Sprite 文件系统

Sprite[30]文件系统为分布式计算环境提供了一个全局访问的功能，是Sprite操作系统的一个重要组成部分。与NFS文件系统相比，在客户端和服务器端都设置了缓存，系统性能有了很大的提升。Sprote文件系统采用了读写锁机制保障数据的一致性，还可支持应用程序通过系统接口访问服务器上的设备。为了提高性能，Sprite在服务器中对部分状态信息作了分布式缓存，并且利用相应

的协议来保证缓存的一致性。Sprite采用了多种数据快速恢复的技术，客户端将自己的状态信息传送到重启服务器，重启服务器的恢复工作就是根据收到的信息做的；重启服务器主动发起状态恢复，利用其他服务器来恢复本机的状态信息；服务器将状态信息写入本地硬盘中，恢复时利用本地信息进行恢复。磁盘阵列和分布式状态在服务器因为硬件失效而不能重启时将提供恢复的可能性。如果服务器在崩溃前没有将状态信息写到磁盘中，那么用户最新写入的数据将会丢失，也说是说Sprite系统没有提供不停机的可用性，服务器在失效时，将没用备用的服务器以供使用。当人们不关注不停机的可用性和可靠性时，可以采用这种技术进行快速的数据恢复。

### 1.2.5 xFS文件系统

九十年代初期，美国加得福尼亚大学伯克力分校设计了xFS[31]基于广域网的分布式文件系统。xFS采用多层结构来充分利用文件访问中局部性的特点，还采用了无效写回策略促使客户端主动缓存，这些技术都是为了减少广域网上的流量。xFS采用的是无集中服务器的对称模型（如图1.3），它将数据存储和元数据管理都分布在多个不同的服务器上以获得良好的可恢复性、可扩展性并且可以简化存储的管理开销。不同区域之间存在着协调服务器，该服务器负责来维户数据的分布信息，该服务器并不是传统意义的服务器，它只负责文件信息的传递工作，而并不处理信息，所有的信息都是在相应文件的客户端完成的，每个用户服务器既充当系统的系统服务器也充当客户端。

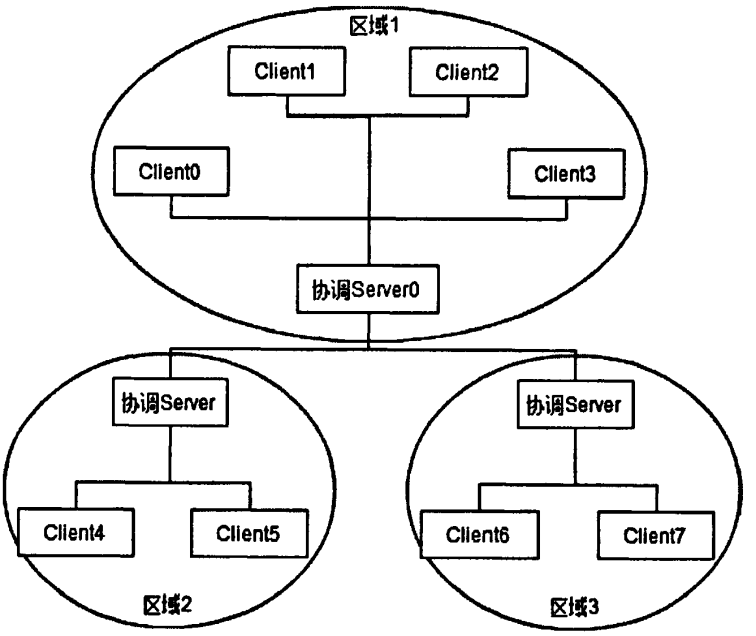



图 1.3 xFS 模型图

由于 xFS 文件系统采用了日志功能，所在在服务器失效时，能够利用日志来恢复磁盘数据，保证了数据的安全性；xFS 文件系统一个全 64 位的系统，能够支持几百万 T 存储空间的数据，还能技持特大文件的存储，还利用 B+树结构来保证对文件的快还检索和空间的快速分配功能；对单个文件的读写操作，xFS 系统的吞吐量高 4GB/s。

xFS 系统也存在着不少的缺点，首它必须维护着文件级粒度的缓存一致性，否则可能出现共享错误。同时 xFS 还没有实现在线扩展功能，这一局限性使得它仅仅停留在实验阶段而并没有投入实际应用。

### 1.2.6 Global File System

Minnesota 大学基于 SNA 研发出了 Global File System[32]共享存储机群文件系统，尔后被 Sistina 公司产品化。Red Hat 公司收购 Sistina 后，公开了 GFS 的源代码，现在 GFS 的全名是 Red Hat Clobal File System。Global File System 是全对称的机群文件系统，根据对称多处理器系统设计，没有传统意义的服务器，所以也就不存在单点故障问题，基本结构如图 1.4 所示。GFS 遵循 UNIX 文件共享语义，在存储设备上缓存文件，并且通过使用设备锁来实现不同客户对文件访问的同步。

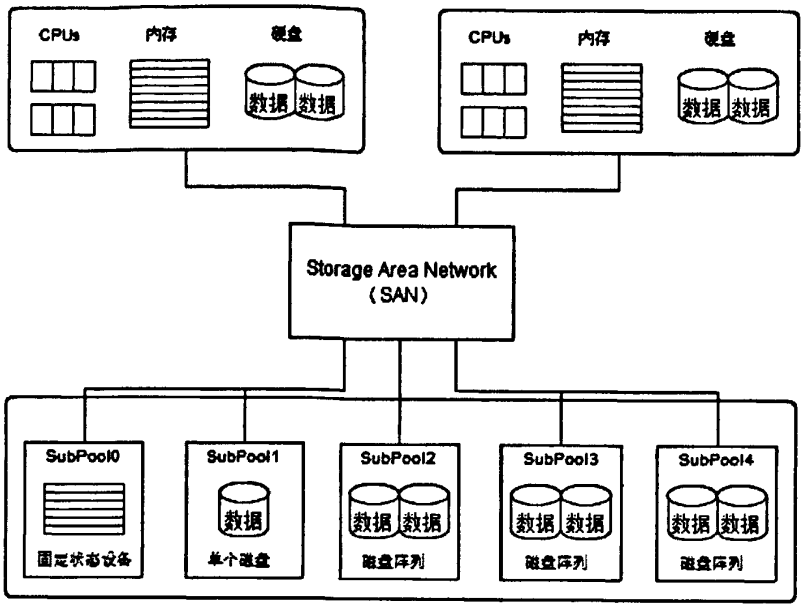



图 1.4 Global File System

通过 GFS 多台服务器可以用同一个文件系统存储文件，信息既可以存在服务器上，也可以存在一个存储局域网上。GFS 将文人年数据缓存在本地存储设备上而不是在内存中。GFS 用事务来对改变文件系统状态的操作进行描述，日志模块从事物模块接收数据，写入磁盘，每个日志为每一个它进行锁保护的元数据都设一个相关的锁。各个客户端有独立的日志空间，所有日志对每个客户端都是可见的。GFS 使用了止前很少设备上使用的 SCSI 锁来进行同步，GFS 通过唯一的锁服务器来进行同步，即使没有设备锁也是如此，锁服务器就有可能会成为其性能瓶颈。

### 1.2.7 General Parallel File System

在 IBM SP 系统应用的虚拟共享磁盘技术的基础之上，IBM 公司推出的第一个共享文件系统是 General Parallel File System（GPFS）[33]，GPFS 在结构上与 Global File System 是似的，但是 GPFS 是并行的文件系统。GPFS 依靠不同粒度的分布式锁来解决并发访问和数据同步问题，依靠扩展哈希技术支持了大量文件和大目录问题。GPFS 依靠日志技术实现对系统故障进进在线恢复，节点日志都在共享磁盘中记录，当某个节点失效时，其他节点通过共享磁盘中的日志来对元数据进行恢复。GPFS 还可以支持热拔插，添加或减少设备后，系

统会对数据进行在线的重新平衡。基本结构如图 1.5 所示，GPFS 文件设备，磁盘及共享网络磁盘组成了整个 GPFS 文件系统，客户机用 GPFS 文件设备将文件系统目不挂载到各自客户机上 iSCS 与网络中的磁盘连接，也可通过通和网络来进行连接。

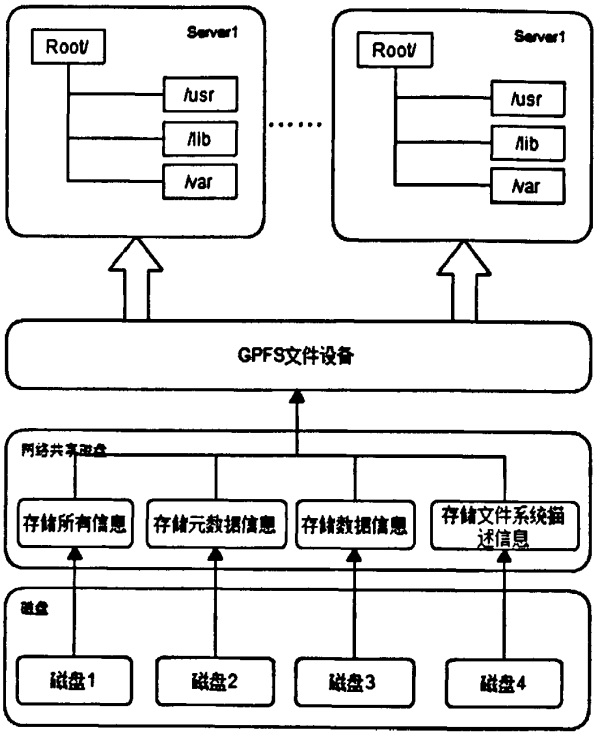



图 1.5 GPFS 系统结构

### 1.2.8 Lustre

Cluster File System 联合 Intel，HP 及美国能源部开发出了 Lustre[35]。Lustre 的名称是由 Linux 和 Cluster 组成的一个混合词，根据 GNU GPL，该项目提供一个数万集群节点和 PB 级容量的高性能的文件系统。从小型的工作组集群到大规模，多站点集群计算机集群都可使用 Lustre 文件系统，世界上排名前 30 位的超级计算机都使用 Lustre 文件系统。Lustre 文件系统可支持上万的客户端系统，几十 PB 的存储以及 GB/s 的数据吞吐量。1999 年在卡耐基梅隆大学的高级系统科学家 Peter Braam 将 Lustre 作为一个研究项目进行研究，成立了集群文件系统公司，并在 2003 年发布了 Lustre1.0。2007 年 Sun 收购了集群文件系统公司，将自己的各种技术带入了 Lustre 中。2008 年 11 月，Braam 离开了 Sun 到

另一个文件系统工作，2010 年 Oracle 公司收购了 Sun，开始管理和发布 Lustre。

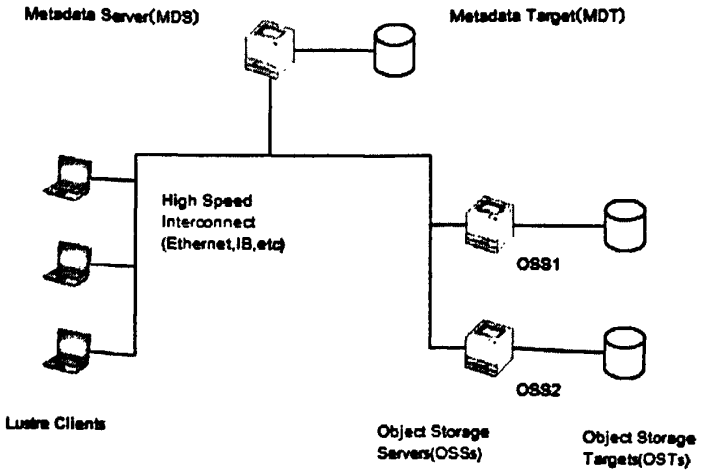



图 1.6 Lustre 结构

Lustre 由 Metadata Server（MDS）、Metadata Target（MDT）、Object Storage Servers（OSS）、Object Storage Target（OST）、Clietn 五闻分组成，如图 1.6 所示。

### 1.2.9 Google File System

Google File System[36] 是针对 Google 公司内部需求设计的分布式文件系统，运用廉价的 pc 机，提供对大文件及大量数据的有效访问。它专门为页面搜索这个核心数据的存储进行了优化。其中使用在至 G 级的文件持续存储，而极少对这些文件进行修改或删除；一般只进行读取或添加操作。它也针对于 Google 的计算机集群进行了优化和设计，这些节点由廉价的 PC 机组成，这就意味着设计上必须防止单个节点失效带来的效率及数据损失。还有其它的设计理念比如数据的高吞吐率，这甚至带来了存取反应期的变差。

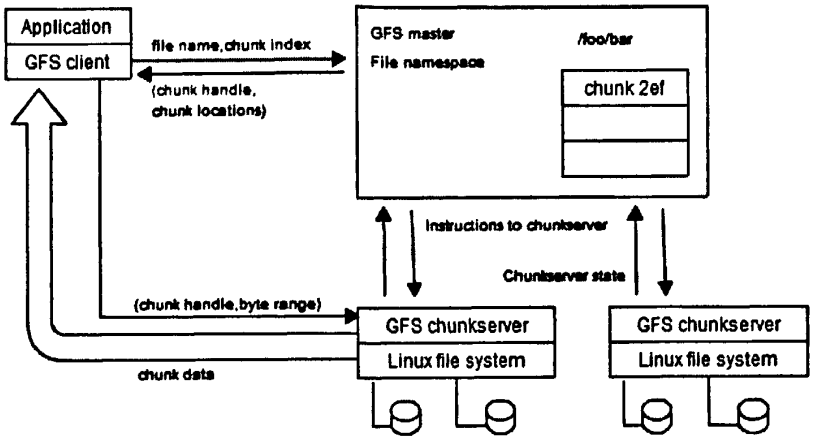



图 1.7 Google File System 结构图

Google File System 集群由一个 mater 和 chunkserver 所组成，并且由很多客户端进行访问，如图 1.7 所示。master 作为文件系统的无数据服务器，chunkserver 是文件系统的数据服务器。Master 通常不存储实际的大块数据，而是维护系统中访问控制信息、命名空间、块当前位置及映射表格这些元数据。映射表格将 64 位的 ID 映射到数据块及其副本的位置和哪些进程正在读写特定的大数据块或追踪复制大块数据的“快照”（一般在 master 的激发下，当由于节点失效时，一个数据块的副本数降到设定数目之下）。所有这些元数据通过 master 周期性地从每个 chunkserver 中收集过来的更新（“心跳”）保持最新状态。Chunkserver 存储数据文件，单个的文件象通常的文件系统中的簇或都扇区一样被分成了许多固定大小的数据块。每个数据都在系统中有一定数量的备分，缺省是 3 个，对于常用文件备分个数会比较多。

Google File System 通过限时的、倒计时系统来处理操作的允许授权，主服务器可能授楹一个进程在一定的时间内访问数据块，这段时间内主服务器再不会给任何其它的进程授权。被更改的 chunkserver 总是主要的数据块存储器，然后再将所做的更改，复制到其它的 chunkserver 上。这些变化直到所有的 chunkserer 做了确认之后才会存储起来，这就保证了所做操作的完整性和自动性。当客户端要访问大的数据快时，着先查询 master 取得数据块的位置，如果数据块没有进行操作，主服务器反回客户端数据块的位置，然后客户端直接与 chunkserver 进行联系并接收数据。

### 1.2.10 HDFS

Hadoop Distributed File System,简称 HDFS[37]，是由 Apache 技持的开源的分布式文件系统,是为以流的方式存取大文件设计的，但是对于大量小文件、低延时数据访问、任意修改或同时写，它的效率就不是特别理想了。HDFS 主要是由元数据节点 Namenode（master）、从元数据节点（Secondary namenode）和一系列的 Datanode（workers）构成，如图 1.8 所示。元数据节点（Namenode）负责管理 HDFS 的命名空间和元数据，它将所有文件夹和文件的元数据存储在一个目录树中，这些信自成在硬盘上保存为命名空间镜像（namespace）和修改日志（edit blog）文件。其中还保存着一个文件由哪些块所组成，以这些块的分布，但是这些信息并不直接存在硬盘上，而是在系统启动时从各个节点收集起来的。数据节点（Datanode）是系统中真正存储数据的地方，客户端（client）或元数据节点（Namenode）可以向数据节点请求读取或写入数据，它隔一定的时间就向 Namenode 汇报一次它存储的数据信息。元数据节点（Secondary namenode）并不是 namenode 出现问题时的备用节点，而是负责完全不同的事情，其主要的工作就是周期性地将元数据节点的数据文件跟修改日志合并，以防止日志文件过大 。为了在元数据节点失败时可以恢复，也将合并的命名空间镜像文件在从元数据节点中保存一份。HDFS 是以 block-sized chunk 组织它存储的数据的，block 默认大小是 64MB，对于不中一个 block 的文件，也会占到一个 block，但是却不必占用 64MB 的物理空间，block 可以理解为 HDFS 在文件系统之上的一个中间层。block 之所以被设置成 64MB 这么大，因为 block-sized 可以很方便地进行文件定位，同时采用大文件，这样传输时间会远远大于文件寻找的时间，这样就最大化地减少了文件定位时间在总的文件获取时间中所占的比例。

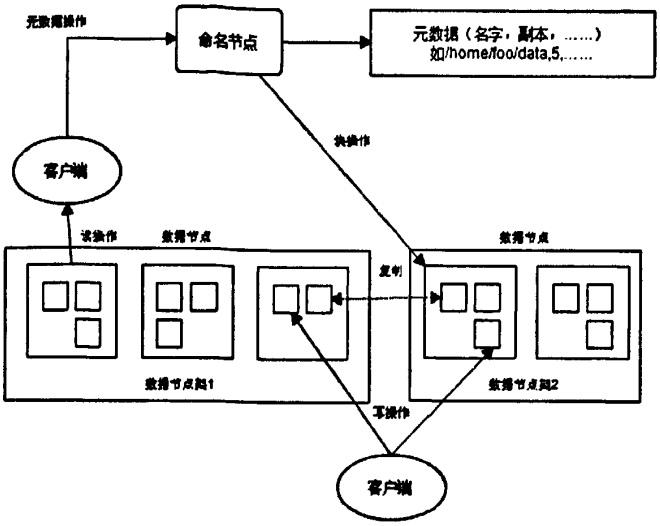



图 1.8 HDFS 系统结构

当客户端（client）在文件系统中进行写操作时，系统会首先将客户端的操作记录在修改日志（edit log）中。元数据节点在人存中保存文件的元数据信息，在记录修改日志之扣，元数节点就会修改内存中的数据结构，并且在每次写操作成功之前，系统都会对修改日志进行同步。命名空间镜像文件（fsimage）是一种序列化的列式，并不能够直接在硬盘上进行修改，它是内存中元数据在硬盘上 checkpoint。如果元数据节点失效，那么就从 fsimage 中加载最新 checkpoint 的元数据信息到内存中，然后按照修改日志重新执行一次操作。从元数据节点（Secondary namenode）的一个主要作用就是用来将内存中的无数据 checkpoint 到硬盘上。从元数据节点首先通知数据节点生成新日志文，以后的修改操作就都记录在新的日志文件中，从元数据节点从数据节点中取得 fsimage 文件和旧的日志文件，并将 fsimage 加载到内存中，然后按照日志文件中的操作一步步地执行，生成新的 fsimage 文件，然后将新的 fsimage 文件传送回元数据节点。元数据节点用新的 fsimage 文件和新的日志文件（开始生成的）替换旧的 fsimage 文件和旧的日志文件，接着将本次 checkpoint 的时间更新到 fstime 文件中。这样元数据节点的 fsimage 就保存着最新 checkpoint 的元数据信息，日志文件也会重新记录，过程如图 1.9 所
示。

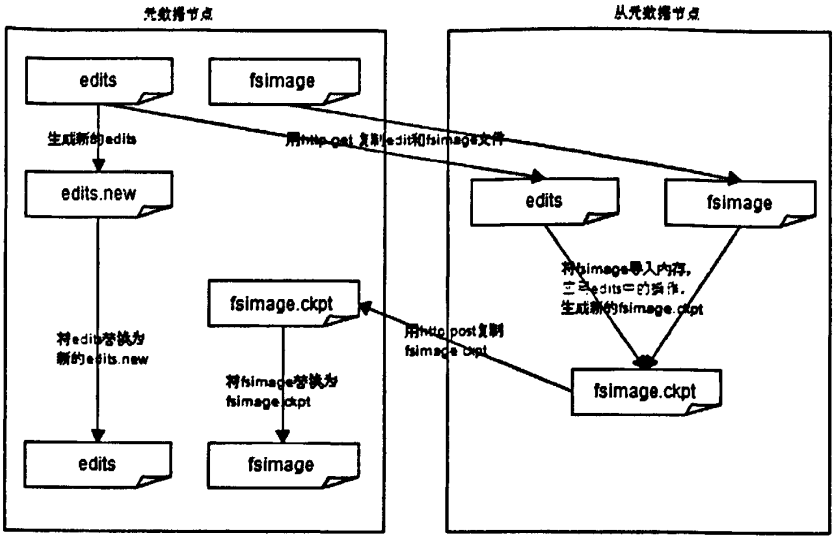



图 1.9 HDFS

### 1.2.11 蓝鲸分布式文件系统

BWFS(BlueWhale File System)[38][39]是中科院计算机所研发设计的网络存储系统，它支持异构的多操作系统平台的跨平台的文件共享，目前支持windows、Linux 和 Mac，它可以许多多台服务器并发地访问磁盘而不必去关心各自的文件系统。针对不同的操作系统 BWFS 都会有不同的客户端，这些客户端能提供对 BWFS 文件系统的访问，并且保证，在不同的系统中的表现是致的。BWFS 中提供了元数据控制器 MDC（MetaData Controller）用来防止当多服务器并发访问相同文件系统时，两台服务器同时写到同一个磁盘位置，或是当某台服务器在读取文件时因为有其他服务器在更新文件造成文件的不一致。MDC 负责协调服务器对 BWFS 文件系统的访问，它在文件数据的读写路径之外。客户端通过独立的链接跟 MDC 通信，获取文件的数据块分配信息和位置信息，然后通过 SAN 网络直接读写入磁盘，这种结构在专业术语中叫“带外（out-of-band）”

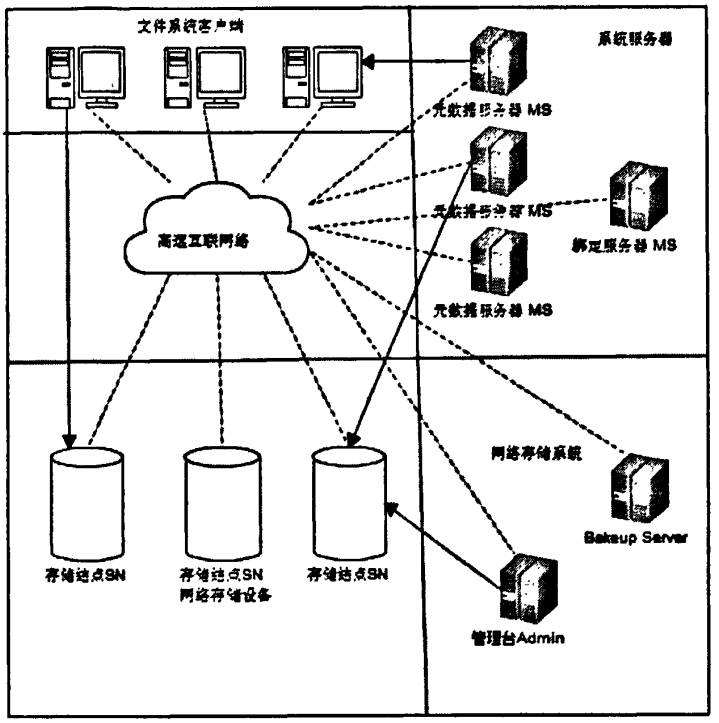



图 1.10 BWFS 系统结构

BWFS 是个可以管理上百个存储节点的大规模集群文件系统，可以向数以千计的客户端提供文件访问服务。其结构如图 1.10 所示，AS 代表应用服务器（application server），MS 代表元数据服务器（metadata server），AD 代表管理服务器（administrator），BS 代表绑定服务器（binding server），SN 代表存储节点（sotrage nose）。

## 1.3 本文主要内容

本文设计并实现了一个分布式文件体系统，通过网络将分散的服务器统一起来为用户提供一个统一的访问平台。分布式文件体系统除了要为用户提供一个方便的访问接口，还要处理负载均衡、节点失效等一系列的问题，还要提供良好的可用性和可扩展性。

## 1.4 本文组织

第一章绪论，针对分布式文件件系统的研究现状（包括各种分布式系统的介绍）和研究背景作了详细的介绍，最后给出了本文主要内容。

第二章分布式文件系统架构，首先分析了需求并给出了本系统的设计目标，最后出了本文件系的整个架构。

第三章元数据系统，先分析元数据系统设计目标，接着给出元数据的分割策略，负载平衡策略以及数据一致性策略。

第四章数据副本，介绍了请求的分发策略和副本的放置策略。

第五章功能测试，介绍了对文件系统各个模块的测试用例。

# 第二章 分布式文件系统架构

## 2.1 系统需求分析

### 2.1.1 设计目标

我们设计的 LFS 文件系统运行在由普通计算机所组成的计算机集群上而不是专门的高性能服务器上，LFS 文件系统存储的文件通常是几百兆字节，上 G 级别的文件十分常见，而且 LFS 中对文件的主要操作是顺序读取，客户通常连续读几百 kb 的数据，同一客户端通常读取文件的相邻位置。分布式文件系统的设计目标大概有透明性、并发控制、可伸缩性、容错以及安全需求等这几个。

#### 2.1.1.1 透明性

访问透明性，用户能通过相同的操作来访问本地文件和远程文件资源。用户可以不需要做任何修改，而同时将本地文件和远程文件以一般的方式处理。

位置透明性，统一使用文件命名空间，文件或者文件集合可以不必改变路径名重新定位。LFS 中采用专门的节点来负责文件系统命名空间的管理，文件的 block 可以被重新分布复制，block 可以增加或减少副本，副本可以跨机架存储，这一切对用户来说都是透明的。

移动透明性，这一点与文件透明性类似，文件系统中经常由于节点的失效增加或者 replication 因子的改变或都重新均衡等进行着复制或都移动，而客户端并不需要改变什么，Namenode 的 edits 日志文件记录着这些改变。

伸缩透明性，LFS 的目标就是构建在大规模廉价机器上的分布式文件系统集群，可伸缩性是无庸置疑的。

#### 2.1.1.2 并发控制

客户端对文件的读写不应该影响其他客户端对同一文件的读写。要想实现近似原生文件系统的单个文件拷贝语义，分布式文件系统需要做出复杂的交互，例如采用时间戳，或者类似回调承诺（类似服务器到客户端的 RPC 回调，在文件更新的时候；回调有两种状态：有效或都取消。客户端通过检查回调承诺的状态，来判断服务器的的文件是否被更新过）。我们的 LFS 并没有这样

做，它的机制非常简单，任何时间都只允许一个写的客户端，文件经创建并写入之后再改变，它的模型是 write-one-read-many，一次写，多次读。这与它的应用场合是一致的，LFS 文件系统中的文件大小通常是 M 至 T 级的，这些数据不会经常修改，最经常的是被顺序读并处理，随机读很少。LFS 文件系统中的文件大小也决定了它的客户端不能像某些分布式文件系统一样缓存常用的几百个文件。

#### 2.1.1.3 文件复制功能

一个文件可以表示为其内容在不同位置的多个拷贝。这样做带来了两个好处：访问同个文件时可以从多个服务器中获取而改善服务的伸缩性，另外就是提高了容错能力，某个副本损坏了，仍然可以从其他服务器节点获取该文件。LFS 文件的 block 为了容错都将被备份，根据配置的 replication 因子来，默认是 3.副本的存放策略也是很有讲究的，一个放在本地机架的节点，一个放在同一机架的另一个节点，另一个放在其他机架上。这样可以最大限度地防止因故障导致的副本丢失问题。不仅如此，LFS 读文件的时候也将优先选择从同一机架乃至同一数据中心的节点上读取 block。

#### 2.1.1.4 硬件和操作系统的异构性

系统必须能够在不同的硬件及操作系统平台上运行。

#### 2.1.1.5 容错能力

在分布式文件系统中，尽量保证文件服务在客户端或都服务端出现问题的时候能正常使用是非常重要的。文件系统的容错性通过这么几个手段：

在 nodeserver 通过向 schedule master 报心跳，当由于网络故障之类的原因，导致 nodeserver 发出的心跳没有被正确地收到时，系统就不会把任何新的 IO 操作派发给那个 nodeserver，该 nodeserver 上的数据被认为是无效的，因此系统就会检测是否有文件 block 的副本数小于设置的值，如果小于就自动开始复制新的副本并分发到其他的 nodeserver 节点上。

检测文件 block 的完整性，LFS 会记录每个新创建的文件所有 block 的校验和。当以后检索这些文件的时候，从某个节点获取 block，会首先确认校验和是否一致，如果不一致，会从其他的 nodeserver 节点点上获取 block 的副本。

集群的负载均衡，由于节点的失效或者增加，可以导致数据分布的不均匀，当某个 nodeserver 的空闲空间大于一个临界值时，LFS 就会从其他节点迁

移数据过来。当某个节点的负载大于一个临界值时，就会将该节点上的数据迁移到其他节点上。

nodeserver 上的 fsimage 和 edits 日志文件是 LFS 的核心数据结构，如果这些文件损坏了，LFS 将失效。因而，fsimage 和 editlog 要保持有多个拷贝，任何对 fsimage 或者 editlog 的修改，都将同步到它们的副本上。它总是选取最近的一致的 fsimage 和 editlog 使用。scheduler master 是单节点存在的，如果 scheduler master 出错，人工干预是必须的。

文件的删除，删除并不是马上从 nodeserver 中移出 namespace，而是将文件重命名，可以恢复，直到起过设置的时间才被正式删除。

### 2.1.2 功能性需求

功能性需求就是要实现文件系统的各种基本操作。

要向外提供文件系统的基本操作，包括：文件的创建，文件的读取，文件的删除，文件的写入，文件的拷贝，目录的创建和删除。

## 2.2 系统架构的设计

### 2.2.1 逻辑结构的分析与设计

对于分布式文件系统服务端的架构一般采用两种设计模式[40]：一种是主从模式，一种是对等模式。在主从模式中，系统采用一个或一组服务器存储文件的元数据并且提供对元数据访问服务，元数据指的是某个文件的文件位置，文件标识符，文件副本数，文件访问权限等属性信息，其他服务器用来实际存储用户数据并对外提供对数据的访问服务。主从模式也有所区分，分为单服务器模式和多服务器模式，单服务器模式（图 2.1）下，系统中只有一个元数据报务器，而多服务器模式下，由多台服务器共同承担无数据的存储及服务。多个元数据服务器模式的优势是可扩展性好，但同时结构很复杂而且也比较容易造成数据不一致。单服务器模式中的单一的元数据服务器很容易就成为系统的瓶颈。

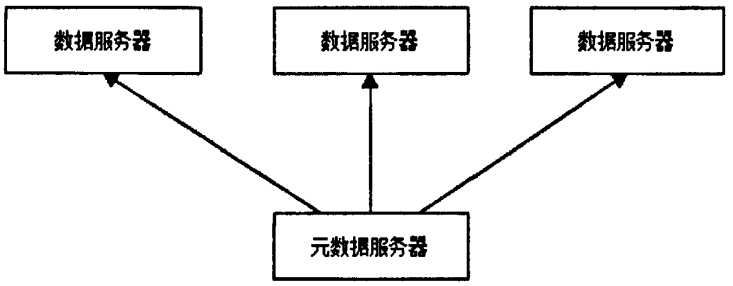



图 2.1 单服务器主从模式

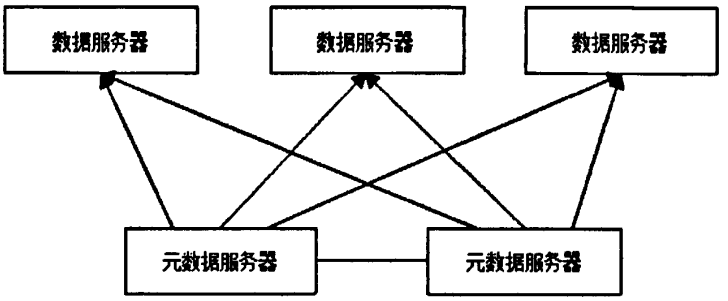



图 2.2 多服务器主从模式

对等模式（图 2.3）中所有的服务器不仅对客户提供数据访问服务，而且还存储了元数据，并向系统内其他服务器提供元数据服务。在对等模式下系统的管理变得很复杂，并且数据更加难以保持一致，而且系统的复杂度也大大地增加了。

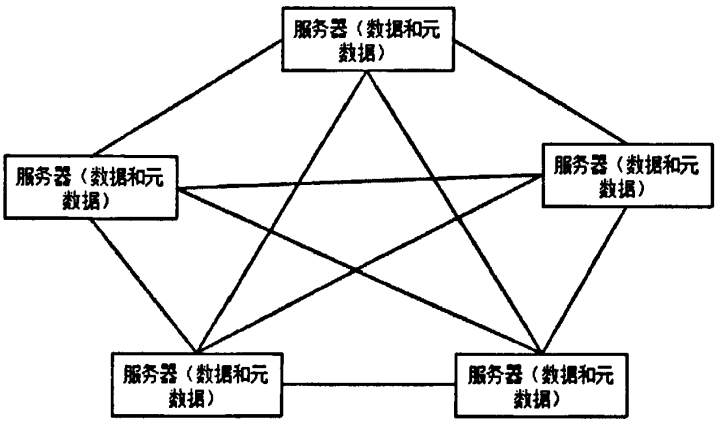



图 2.3 对等模式

主从模式下文件系统内数据一致可以比较好地得到解决但是它的伸缩性和可扩展性就没有对等模式好了。而在对等模式下对元数据进行管理比较复杂，并且数据的一致性也不容易实现。我们采用的是独特设计的分层的多服务器的主从模式，我们的元数据服务器融合了单服器主从模式和多服务器主从模式的

特点，设计了一个独特的多服务器结构，使得我们的元数据服务器获得了良好的扩充性，同时也不容易出现数据不一致的情况。

### 2.2.2 访问数据方式的分析与设计

现在数据访问的方式主要有两种[41]。一种是元数据服务器在收到用户的数据请求后，元数据服务器通过查询元数据取得数据所在的数据服务器的位置，再通过元数据服务器把数据从数据服务器传送客户。这种方式存在着一个很大的缺陷，就是客户的元数据操作和数据操作都经过了元数据服务器，这会对元数据服务器的 I/O 造成很大的压力，使它成为系统的瓶颈。第二种方式中，首先向元数据服务器请求元数据信息，元数据服务器返回元数据，然后客户端根据返回的元信息直接跟数据服务器交互取得数据。这个过程中，客户端还会将元数据缓存起来，在元数据过期之前如果还需再次访问同一数据时，就不需要再次向元数据服务器发送请求，而是根据缓存数据直接访问数据服务器，这样可以大大地提高整个系统的性能。我们设计的系统中使用第二套方案。

### 2.2.3 数据存储体系结构

我们的数据存储系统的结构可以看做两层：有结构的存储网和广域对等存储网，其拓扑结构如图 2.4 所示。有结构的存储网分为两层，第一层存储结构是 Peterson 存储网络，即基本存储网络，第二层结构是由基本网络形成的广域网环。这两层结构就形成了有结构的存储网络。

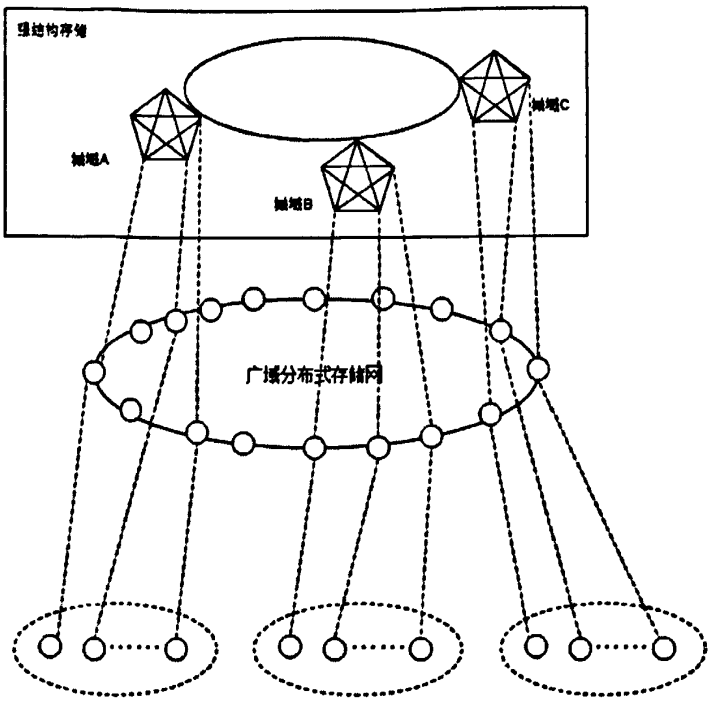
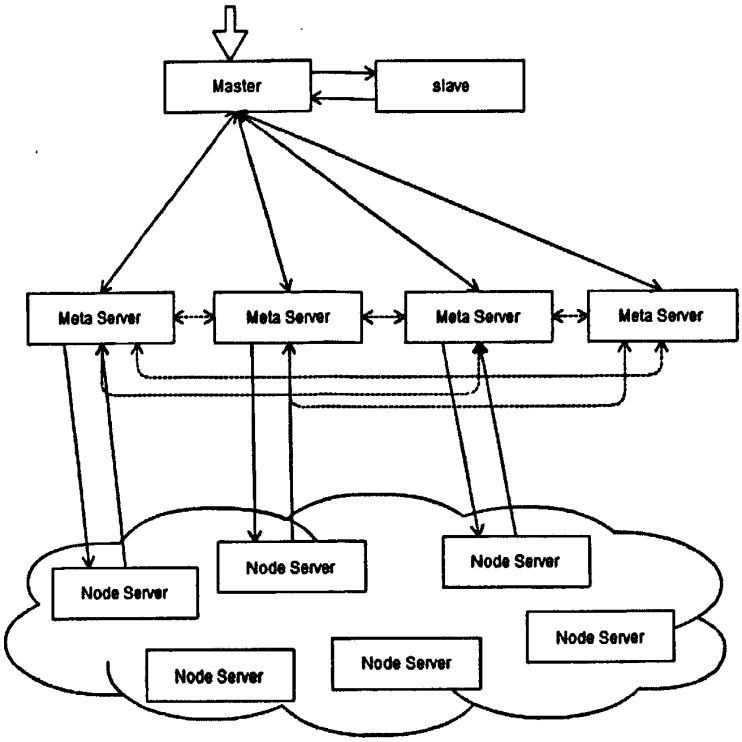



图 2.4 数据存储网

城域存储结构是由 Peterson 存储节点组成的。广域存储体系结构是由每个城域结构以双环拓扑连接构成的。

### 2.2.4 整个文件系统的结构

由下图中可以看出本文件系统主要由以下几个部分组成：master、metaserver、nodeserver、scheduler master。其中 master 是本文件接口对系统中的文件请求首先回来到 master 然后由 master 再对请求进行分配，slave 是 master 备分，master 宕机时会有 slave 接替 master 的工作。metaserver 是文件系统的元数据服务器，来对外提供元数据服务，但是 metaserver 并不存储元数据，而是在启动 metaserver 时，从各个 nodeserver 收集起自己的元数据。nodeserver 是真正存储数据的地方，各个数据块最终都会在 nodeserver 中存储。




2.5 文件系统整体结构

## 2.3 系统的 I/O 操作流程

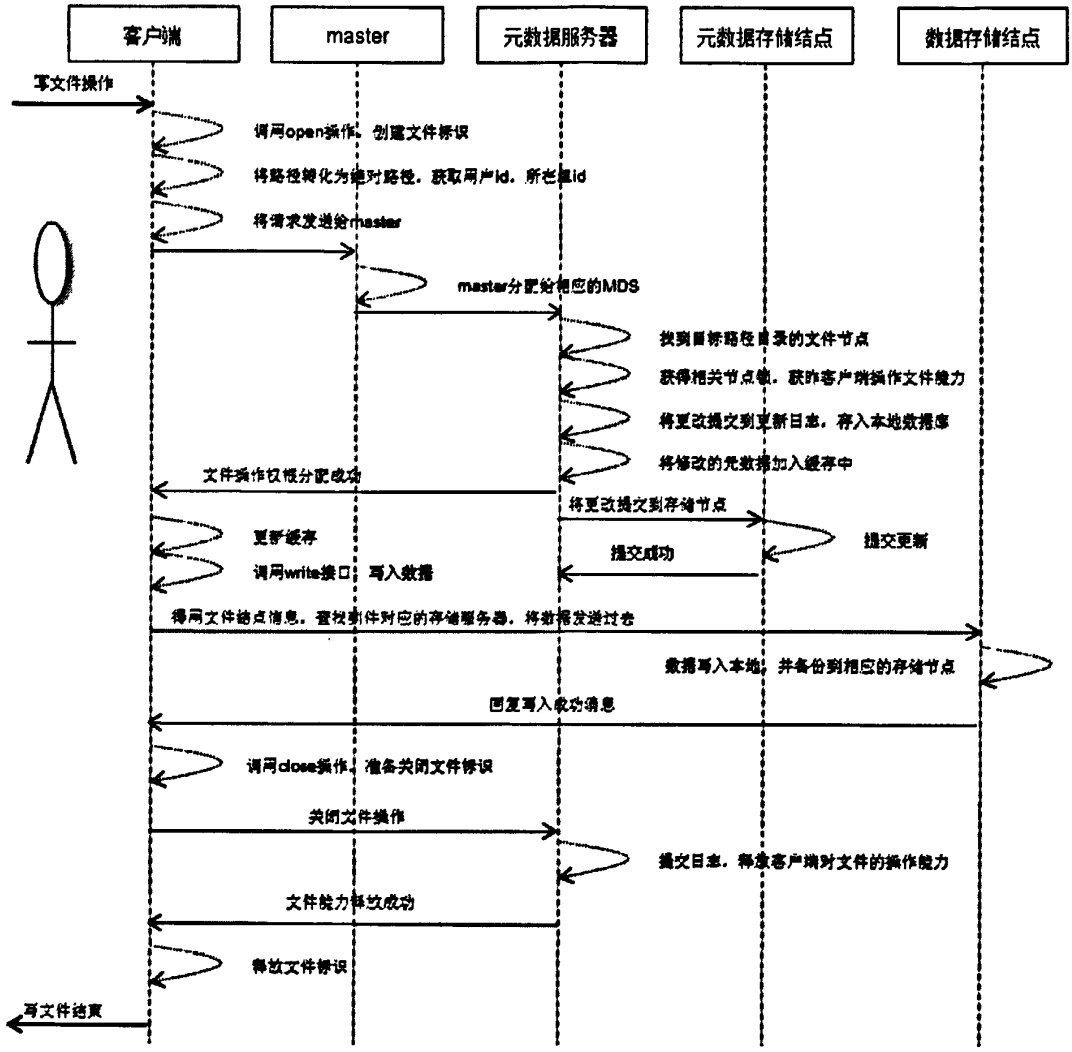



图 2.6 写文件操作流程图

具体步骤如下所示：

1.客户端使用 open 接口打开一个文件，并发送一个 open 请求到 master。

2.在发送请求时，客户端首先检查本地缓存，查看在本地是否已经存在要操作的目录或文件的元数据信息。如果找到缓存数据那就找到该目录或文件所属的主 MDS，并直接将请求发送给这个 MDS 来处理，如果不存在则由 master 分配给相应的 MDS。

3.MDS 收到请求后，若该 MDS 就是主 MDS 则直接对文件进行处理，若不是主 MDS 则查找并转交该请求由主 MDS 处理。

4.请求转到主 MDS 之后，如果缓存中的元数据不完整就需要从服务器中读取元数据，只有缓存中的元数据完整地存在时才能对该目标进行对应的操作。

5.在对元数据进行更新之前，需要对相关数据节点加锁，加锁的同时还进行用户访问控制权限判断工作。
6.根据用户修改产生新的元数据，生成相应的日志记录，并将日志写入到数据库中，然后将修改之后的元数据加入到缓存之中。
7.对元数据的修改完成后回复客户端成功信息，并清除处理过程产生的无用记录，最后启动副本更新流程保证主从副的一致性。
8.如果元数据的修改日志达到了一定的规模，就将在缓存中进行的修改写到硬盘中并清除相应的日志。
9.客户端收到返回的消息后，将相关元数据放入缓存，并设置对应的文件描述符。
10.客户端调用 write 接口对文件进行修改，将会首先判断用户是否有 write 权限，如果没有 write 权限则返回失败，如果有则调用 write 接口。
11.write 接口被成功调用之后，查找到数据所存储服务器。
12.找到数据所存储的服务器后，通过网络将数据传给服务器。
13.服务器收到数据后，调用存储模块对做修改的部分进行修改，并在其他的几个备分中修改数据，保证一致，最将操作的结果返回给客户端。
14.客户端收到返回的消息后，数据更新已经完成，调用 close 接口关闭文件。
15.客户端将关闭请求发送给元数据服务器请求关闭文件。
16.元数据服务器收到关闭请求后，释放相应的锁，
LandFile 元数据服务器收到关闭操作之后，释放文件节点元数据的操作能力并更新文件的大小信息，并回复客户端结束消息。
17. 客户端收到回复消息后，释放文件标识符。写文件的全过程结束。

## 2.4 本章小结

本章首先分析了我们分布式文件系统的需求，然后根据我们的需求设计整个分布式文件系统的结构，分别元数据管理系统和数据存储系统组成，元数据管理系统负责对元数据进行维护工作，而数据存储系统则负责数据的直接存储工作，二者互相配合共同完成了整个系统的功能。最后介绍了文件系统写文件时的操作流程，通过这个流程可以对这个分布式文件系统的结构和功能做个更清楚的介绍。

# 第三章 元数据系统

我们的文件系统中，用元数据描述文件系统中文件的一些特征信息，包括目录属性信息和文件属性信息。目录属性信息包括访问权限、目录名称、该目录下的子目录及文件等，文件属性信息包括访问权限、文件名称、文件用户、文件的大小、创建时间、访问时间、修改时间等。分布系统中的文件数据是分散在不同的节点上的，我们每次访问系统中的数据时首先要访问元数据服务器进行定位，随着数据信息的爆炸性增长，元数据的信息量也变得越来越大，设计一个合适的元数据系统也变得越来越困难。虽然元数据信息在整个文件系统的数据中所占比例不足 10%，但是根据统计，对元数据的访问可以占到全部访问的 50%到 80%，对一个文件系统的访问往往都是以元数据开始访问，最终又以元数据结束访问，一个好的元数据系统会对整个文件系统的提升有至关重要的作用，元数据可以关系到整个系统的可用性、一致性、正确性、扩展性等，因此元数据系统在整个系统中至关重要。

## 3.1 元数据系统的设计目标

（1）高性能

当用户发出请求时，MDS 机群必须对用户的请求进行及时的响应，对元数据提供快速的访问服务。不同的系统中管理元数据的方式是不同的，对请求的处理过程也不一样，但是请求最终都还是要通过直接存储元数据的 MDS 取得，因此，元数据系统中 MDS 必须能最大限地支持对元数据的并发请求；MDS 必须能充分得用内存，努力提高缓存数据的命中率；MDS 最好能够对机群支持负载的动态调整工作；MDS 必须对大目录进行有效的管理。

（2）灵活

当目录的属性被修改时，系统应当尽量避免对其子目录及其目录下的文件的影响。例如当我们修改一个目录的字时，我们应当尽量避免因 hash 的改变而使其下的子目录及文件迁移到其他的 MDS 上；当我们修改了一个目录的访问权限时，就要避免对其子目录及文件造成影响。

（3）一致性

在我们文件系统的MDS中会对元数据同时存储多个副本，而各个副本都分布在不同的MDS上，为了使所有的用户看到相同的命空间，我们必须对元数据的各个副本进行维护，保持它们的同步。

（4） 可扩展性

文件系统在使用过程中，经常会发生服务器宕机或临时增加元数据服务器的情况，这两种情况下都会发生大规模的元数据的迁移，我们系统的策略就要使尽可能多的数据存储位置不发生变化。

## 3.2 元数据的分割策略

元数据如何在服务器上划分和组织，对于多元数据服务器架构的元数据服务来说是关系到其性能和可扩展性的重要因素。

（1）无分割策略

无分割的元数据管理策略并不分割命名空间，而是把整个的元数据和命名空间都放到一台MDS机器中，或都在多台MDS中存放整个命名空间及元数据的多个副本。著名的GFS文件系统就是采用这种策略的。

在这种策略中，用户对一个元数据发出请求时，请求首先被发送到保存着元数据信息的MDS机器上，MDS接收到请求后，会对请求的路径中各级目录的权限进行验证，获得许可后就会将元数据返回给用户。这种策略将元数据跟命名空间存储在同一台计算机上，元数据的一致性比较容易恢复，MDS在处理用户的请求时，不需要同其他的MDS进行交互，也可以大大地减轻网络负载。当元数据规模不太大时，内存中就可以容纳大部分的元数据，缓存的命中率和利用率会很高，可以大大减少硬盘I/O次数，提高存取速度。修改父目录的属性不会对其包含的目录和文件造成影响。但是由于没有对命名空间进行分割，MDS要处理所有的用户请求，容易造成MDS负载过重。访问元数据时需要对目录层次进行遍历，代价比较高。当对元数据进行并发的修改时，同步锁粒度为所在的目录的父目录，并发程度比较低。元数据保存在一台机器上，不能很方便地通过增加服务器来进行扩展。

（2） 静态子树划分策略

在静态子树分割这种策略中，元数据分为两种，文件元数据和目录元数据。这种管理策略通过在MDS启动时进行配置，把文件系统用目录树的结构进行组织，将全局命名空间分成若干棵子树，每一个MDS保存一棵或都若干

棵子树，配置完成后除非进行人工干与，否则不会改变。著名的分布式文件系统 NFS、AFS、Coda 就是采用静态子树分割策略的。

这种方法实现比较简单，对元数据的管理进行了简化，客户也可以很容易地就查找到所需要的 MDS。用户向服务器发出元数据请求时，请求会被发送到相应的 MDS 服务器上，MDS 服务器在经过一系列的权限认证后，就会将用户请求的元数据返回给请求的用户。在这种策略下，每个 MDS 将元数据以子树的形式进行组织，这就具有非常好的存储局部性，而且被请求的 MDS 服务器在处理用户请求时，不需要跟其他的 MDS 进行交互，使整个系统的网络负载很低，并且也很容易对元数据的一致性维护。修改某一个目录的属性，不会对其子目录或文件造成影响。但是整个的命名空间被分割分子树粒度，粒度较粗，而且也不能保证负载均匀地分布在各个 MDS 服务器上。访问元数据时需要对目录层次进行遍历，代价比较高。当同时对一个 MDS 服务器进行请求的用户比较多时，就要频繁替换缓存中的数据，导致硬盘 I/O 的增加，影响效率。同步锁粒度较粗，为所在的子树，当用户并发对元数据进行修改时，并发度较低。我们是根据配置将系统中元数据在 MDS 中进行分布的，如果有 MDS 宕机或需增加 MDS 时，就要重新对系统进行配置，元数据也要重新分布这就会导致元数据的大规模迁移，不易进行护展。随着我们系统的使用，不同的子树的规模也会差别较大，伴随着子树的收缩和扩张有的 MDS 可能会空载而另一些 MDS 则有可能因超载而成为系统瓶颈，这时必须进行手动干预，但是也需要迁移大量的数据，我们难以对其负载进行控制。

（3）动态子树划分策略

动态子树划分策略是 CEPH 文件系统所采取的策略，该系统由 UCSC 存储系统研究中心开发。在动态子树分割这种策略下，也存在着两种元数据，文件元数据和目录元数据。动态子树划分策略在共享的存储系统中保存元数据，MDS 缓存目录子树并且对相应的元数据操作进行处理，这种动态的元数据管理策略克服了静态子树分割的一些缺点，它将全局命名空间的不同子树分布到不同的 MDS 上，并且可以根据各个 MDS 和负载情况进行动态的迁移，从过度负载的 MDS 中将数据迁称到负载较轻的 MDS 上。客户端缓存了已知的元数据与 MDS 的对应关系，这就解决了动态迁移导致的 MDS 查询超时间题，这样根据局布性原理，后续操作就可以直接使用缓存的 MDS，大大提高了命中率。元数据的存储是存共享的存储系统中，而我们进行迁移的只是缓存，这就会大大降低所迁移的数据里，可以平滑地对系统进行扩展，系统可伸缩性良好。

在系统中创建目录或都文件时，系统采用轮转策略选择相应的 MDS，例如第一次创建时选择了 MDS1，下次再创建就选择 MDS2；而在创建比较深的目录或文件时，就不再使用这种轮转策略，而是直接将元数据保存在父目录的 MDS 上。对元数据的访问过程与静态分割策略中类似。在这种策略下，每个 MDS 将元数据以子树的形式进行组织，这就具有非常好的存储局部性，而且被请求的 MDS 服务器在处理用户请求时，不需要跟其他的 MDS 进行交互，使整个系统的网络负载很低，并且也很容易对元数据的一致性维护。修改某一个目录的属性，不会对其子目录或文件造成影响。当系统中 MDS 服务器的负载严重不均衡时，可以将负载较高的 MDS 上的热点数据迁移到负载较低的 MDS 上，来动态地维持负载的均衡。但是整个的命名空间被分割分子树粒度，粒度较粗，而且也不能保证负载均匀地分布在各个 MDS 服务器上。访问元数据时需要对目录层次进行遍历，代价比较高。当同时对一个 MDS 服务器进行请求的用户比较多时，就要频繁替换缓存中的数据，导致硬盘 I/O 的增加，影响效率。同步锁粒度较粗，为所在的子树，当用户并发对元数据进行修改时，并发度较低。当子树的规模扩大时，会造成对应的 MDS 过度负载，并且对数据的迁移也会造成较高的网络负载。

（4）静态哈希策略

在静态哈希这种策略中，通过 hash 函数来完成元数据的定位和分配，也存在两种元数据，文件元数据和目录元数据。这种管理元数据的策略通过对文件的某种标志（路径、名称或其他）进行计算得到 hash 值，将元数据分布到各个 MDS 中。当我们请请一个文件时，先根据其标志算出其标志的 hash 值，就可以确定目标 MDS，然后客户端直接向相应的 MDS 服务器发出请求。相应的 MDS 收到用户请求后就会对用户权限进行认证，然后将用户所请求的元数据返回给用户。使用 hash 函数可以快速地进行元数据的定位，设计较精良的 hash 函数也可以让元数据分布得比较均匀。Lustre、Intermezzo、zFS 等一些著名的文件系统就采取这种策略。

如果确定文件的标志为文件所在的目录的话，则采用以目录为粒度进行分割，而如果采用的标志是自身的全路径的话，则以单个文件进行分割。采用静态哈希这种策略，系统采用目录或者文件作为分割粒度，分布比较均匀随机，整体来看对每个 MDS 上的请求数量基本差不太多，各个 MDS 负载比较均衡。目标 MDS 是通过 hash 值的计算来确定，速度比较快。用户对文件进行并发的修改时，同步锁以文件为粒度，并发程度比较高。但元数据在 MDS 中都是以

目录或者文件为粒度存储的，同一目录下的文件的分布被打乱了，没有利用存储的局部性，分布很随机，不容易对元数据的一致性进行维护，目标在向 MDS 请求元数据的过程中，需要频繁同其他的 MDS 服务器进行交互，也造成了较重的网络负载。虽然我们可以把元数据均匀地分布在各个 MDS 上，但是由于不同的元数据访问热度是不同的，依然可能出对热点文件访问较高而导致负载不均衡时，无法通过迁移热点数据来调整各个 MDS 的负载。进行权限认证时需要遍历整个目录层次，代价比较高，而且不同层次的目录是分布在不同的的 MDS 服务器上的，因此在进行权限认证的过程需要多次发送和接收消息。对于每个 MDS 服务器的请求是随机的，使得 MDS 缓存的命中率比较低，与硬盘进行 I/O 的次数较多。文件的某个目录重命名后，会使该目录下的子目录和文件的 hash 值都发生改变，都需要根据改变后的 hash 值进行相应的迁移；同时在系统中增加或移除某一个服务器时，也会造成 hash 函数改变，直接影响了所有的文件和目录，迁移代价非常大，不易进行扩展。

针对于静态 hash 算法的缺点，华中科技大学的冯丹提出了 DOIDFH[16]（Directory Object IDentifier and Filename Hashing）算法，对文件名和目录都进行 hash，根据文件名和上层目录 id 的 hash 值来确定 MDS。这种方法还使用访问控制表对目录和文件和访问权限进行控制，也能避免目录重命名所造成的在规模数据迁移，但是这种方法仍然没有利用元数据的局布性，热点数据也仍然会造成各 MDS 负载不均衡，增加或删除 MDS 时还是会引起在规模的数据迁移。

（5） Lazy Hybrid 策略

在 Lazy Hybrid 这种策略中，不仅保存了文件元数据和目录元数据，同时还保存着目录的层次结构，用来向用户提供对系统的语义操作。Lazy Hybrid 元数据管理策略是通过文件的全路径计算该文件的 hash 值的，并根据这个值将元数据分布到所有的 MDS 上。当用户对某一元数据发出请求时，客户端会先计算出这个文件的 hash 值，再通过 Meta Lookup Table 来确定相应的 MDS，发送出请求，收到请求后目标 MDS 通过向 Access Control List 进行权限验证，之后将用户请求的元数所返回给用户。

采用静态哈希这种策略，系统采用目录或者文件作为分割粒度，分布比较均匀随机，整体来看对每个 MDS 上的请求数量基本差不太多，各个 MDS 负载比较均衡。目标 MDS 是通过 hash 值的计算来确定，速度比较快。用户对文件进行并发的修改时，同步锁以文件为粒度，并发程度比较高。 系统通过 ACL

直接对用户进行权限验证，减少了 MDS 间的交互，降低了网络负载。当系统中 MDS 服务器的负载严重不均衡时，可以将负载较高的 MDS 上的热点数据迁移到负载较低的 MDS 上，来动态地维持负载的均衡。但元数据在 MDS 中都是以目录或者文件为粒度存储的，没有存储的局部性，分布很随机，不容易对元数据的一致性进行维护。对于每个 MDS 服务器的请求是随机的，使得 MDS 缓存的命中率比较低，与硬盘进行 I/O 的次数较多。文件的某个目录重命名后，会使该目录下的子目录和文件的 hash 值都发生改变，都需要根据改变后的 hash 值进行相应的迁移；同时在系统中增加或移除某一个服务器时，也会造成 hash 函数改变，直接影响了所有的文件和目录，迁移代价非常大，不易进行扩展。

本系统中采用的是静态 hash 方法，以文件作为分割粒度，可以使分布比较均匀，访问元数据的速度比较快。对文件交发修改时是以文件为同步锁粒度的，并发程度比较高。这种策略也有缺点，虽然我们可以使数据均匀地分布在各个 MDS 上，但是不同元数据的访问热度是不同的，再热点数据跟普通数据的访问量差别是非常大的，对点数据的高访问量依然会造成各个 MDS 负载的严重不均衡，我们无法通过迁移热点数据来调整各个 MDS 的负载。针对于这个情况我们的系统在负载均衡策略中有专门的解决方法。

## 3.3 元数据服务器负载均衡策略

网络中任务到达随机性很强，各个的文件的访问频率是处在不断地快速的变化之中的，数据的访问也往往是不可预测的。在多 MDS 体系架构下，各个节点在处理性能方面是有差异的，是非常容易出现 MDS 负载不均的现象的，某些节点上任务过多，称为超载，另一些节点上的任务过少，系统处于空闲，造成资源浪费。而且在超载的节点上，由于过多的任务形成忙等待，导致任务慢和迟延，影响整个服务的质量。怎样才能充分利用系统资源，突破系统瓶颈，提高系统性能和用户体验，一直以来都是人们重点研究的对象。MDS 的负载均衡性对整个文件系统都起着至关重要的作用。本节主要针对负载均衡的一些问题研究常用的解决方案，根据负载均衡的评价指标及目标讨论系统负载均衡的关键技术。

### 3.3.1 MDS 集群负载均衡问题

根据负载均衡决策时机的不同可将负载均衡分为两种方法：一种是在任务到达时将其分配给负载较低的节点上；一种是将部分任务由超载节点上迁移到负载低的节点上。在实际的应用中常常通过调度请求的目标来实现各个节点间的负载调整，从而达到负载均衡的目的。

根据任务行为的预知性可将负载均衡分为动态和静态两种负载均衡策略。

静态负载均衡是完全根据先验知识做出决策而并不考虑实时的系统负载情况，也就是说一个任务的分配跟服务器的负载情况是无关的，一般用在任务比较确定的情况下。静态负载均衡是通过任务的划分和任务分配（排队法、概率法、随机法、图论法）来达到调节负载的目的。静态负载均衡实现起来比较简单，但是这种策略不考虑当前系统中的负载状况，具有一定的盲目性所以决策准确度比较低。此外，决定静态负载效率的一个重要因素就是对系统和任务的特征如任务通信和执行代价等是否了解充分，如果对这些先验知识没有充分了解或者有些偏差，那这种策略的效率会更低。

动态负载均是根据系统实时的负载情况动态地将任务在各处理机之间调整，以达到调节系统负载的目的，一般用于任务不确定的情况。系统中任务是动态到达各个服务器的，所以服务器上的负载有时会突发地减少或者增加。在务务器超载时，应将其部分任务迁移到负载较轻的服务器上，或者由负载较轻的服务器主动提出申请将超载服务器部分任务转移过去。动态负载均衡策略就具有更大的针对性和灵活性，可根据系统实时的负载状况进行有针对性的调整，对系统负载和任务执行进行动态的调整。尽管在收集分析系统负载时会有一定的开销，但是针对性的调整之后系统性能将取得更大地改善。根据动态负载均衡控制方式的不同可分为集中式和分布式两种负载均衡。

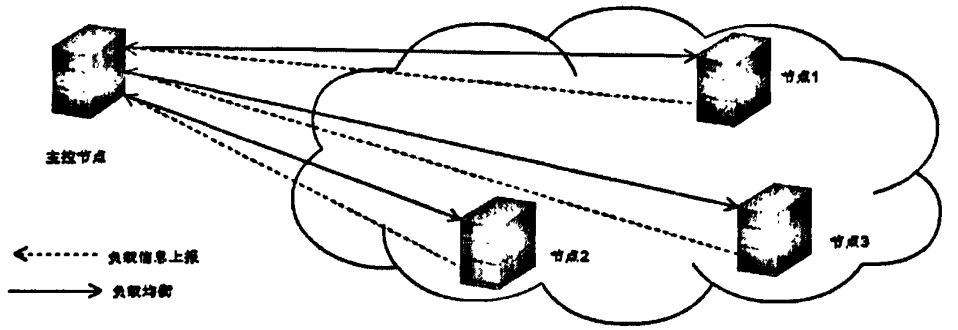



图 3.1 集中式负载均衡

集中式负载均衡将负载控制信息放在一个指定的节点上，所有节点的控制信息都报给这个节点，该节点根据汇总的信息对负载做出调整决策。主控节点通过任务分配或请求重定向来调节各个节点的负载，所以主控节点又是系统中的任务调度器。在这种负载均衡策略中，系统中的节点不知道其它节点的负载情况，通过将负载信息汇总到主控节点，然后由主控节点负责各个节点负载的控制。集中式负载控制策略对系统的负载信息了解比较充分，做出负载调整决策时判断准确，而且也比较的容易。但是主控节点信息流量很大，很容易就造成了单点故障影响系统的稳定性。从图 3.1 和图 3.2 的比较可以看出，分布式负载均衡控制是自组织的负载管理方式所有节点都参与了负载控制，每个节点只需要知道局部的负载信息就可以通过逐步渗透的方式对整个系统的负载进行调节，没有单点故障的问题，可扩展性好。

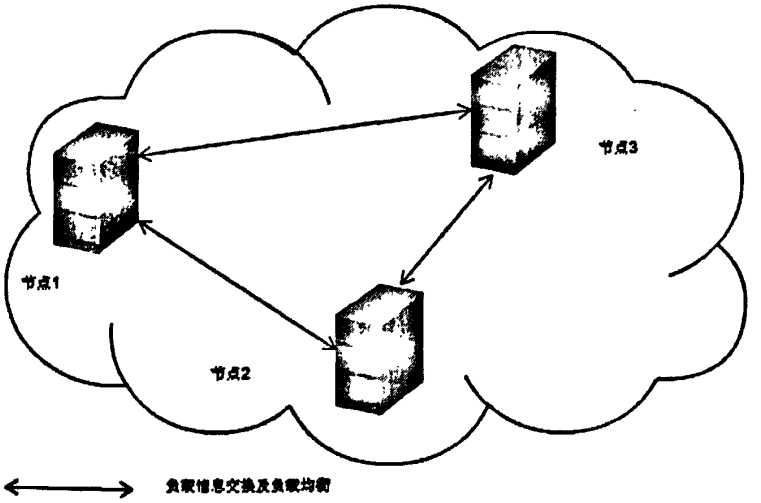



图 3.2 分布式负载均衡

分布式负载均衡策略是依靠任务的复制或者迁移来达到调节系统负载的目的。分布式负均衡策略中没有调度器，系统的负载均衡更多地是要系统内的节点能过自织的方式调节。对于元数据服务系统，服务器的负载来自于对元数据的请求，所以就要通过复制或迁移元数据来调节系统内各服务器的负载。

根据引起负载不均的原因，我们将元数据系统的负载不元的问题分成两类：一是由于元数据分布不均所引起的负载不均；二是由于访问频率不均所引起的负载不均。

元数据分布不均所引起的负载不均是由于不同的 MDS 的元数据的量不同，元数据里较大的 MDS 就很容易过载，而数据量较少的 MDS 就有可能空

载，引起系统负载的不均。在静态子树划分策略下会比较容易发生这种问题。目录子树在分配到 MDS 上之后，随着用户的使用而使有的目录收缩有的目录扩张，过一段时间后各个 MDS 上的元数据量差别会比较大，从而引起各个 MDS 负载不均。

元数据的访问频率不均引起的负载不均：不同的文件的访问频率差别可能很大，并且访问频率还会随着时间不断地变化，在实际应用中，热点文件的访问频率一般是普通文件的上万倍，因此，即使是在元数据分布均匀情况下，各个 MDS 负载的差别仍然可能会很大。

### 3.3.2 负载均衡解决方案

对于由于元数据分布不均引起的负载均衡问题，一般采用两种解决方案，一种是使用静态 hash 方法使元数据重新均匀分布，但这种方法对于由于数据访问频率不同所造成的负载不均没有什么效果；还有另一种方法就是采用动态迁移方法，根据各个 MDS 的负载情况动态地调整其元数据。

对于由于访问频率不同所造成的负载不均问题，根据访问类型一般可分为两种情况：一是由于大量的写操作引起的负载不均，另一种是由于大量的读操作引起的负载不均。在实际的网络应用中，会经常发生突然出现热门数据，比如新闻、图片或者视频，对于这类数据，一般都是大量的读操作，而很少对其进行更新。因此就可以采用副本创建的方式，在多个不同的 MDS 上复制热点数据，从而达到负载分流的目的。虽然可以用复制的方式来解决热门数据的问题，但是这样会使数据的一致性会更加难以维护。

经常会在科学计算中碰到在同一个目录下反复地创建和删除文件的操作，几乎所有的分布式文件系统都不支持并行的写操作，所以这种对目录或文件的更新只能集中在一台 MDS 上进行，所以我们难以对负载进行均衡。针对这种情况，CEPH 文件系统提出了对目录进行分片，使数据迁移的力度细化，使这一问题得到了较好的解决。CEPH 文件系统增辑一个介于目录和文件之间的一个大小的粒度，称作目录分片，目录分片是一个目录下的部分项的集合，目录内仅保留目录分片的信息，单个目录项的增加删除都是在目录分片内进行的，目录项的权限检查也是由目录分片负责的。并且将目录分片作为基本单位进行数据迁移，当某个目录下由于出现了大量的增加删除分在操作引起负载过高时，就可以迁移或复制该目录下的部分目录分片到另外的 MDS 上。当单个目

录分片内包含的内容过多时，就将这个目录分片分成多个；单个目录分片包含的内容较少时就将其与其他的目录分片进行合并。

### 3.3.3 负载均衡评价指标

从服务器角度来看，MDS 集负载分布均匀，元数据均匀分布，MDS 集群吞吐量大是评价 MDS 负载均衡的一个常见指标；而从客户的方面来说，元数据操作响应时间和 MDS 集群可承载的用户是常见的指标。我们根据元数据的访问量和 MDS 集群的负载，选择适合的元数据进行复制或迁移。

我们主要从几个方面对一个负载均衡算法的效果进行评价：MDS 负载均衡的效果，对资源的消耗量和负载的稳定性。根据负载均衡的效果来评价一个负载均衡的算法是效的，那么在各种情况下，负载均衡算法都应该能对 MDS 集群的负载进行调整，使得调整后的系统的性能得到提升，当系统整体负载较高时，应能将负载平均分配到各个节点中去。负载均衡算法不应该出现抖动现象，而应该是稳定的。当 MDS 集群中的机器性能差别比较大时，如果从负载高性能强的机器中迁移部分元数据到负载低但是性能较差的机器上时，很容易就使性能差的机器过载，从而再次引起迁移。像这样元数据在各个 MDS 机器中反复迁移就是抖动现象。要防止产生抖动现象，负载均衡算法应该能预测迁移之后的负载状况，根据各个 MDS 的性能，选择适合的 MDS 进行迁移。我们在进行数据迁移的时候要消耗 CPU、内存、带宽等，在进行元数据迁移的时候，会对元数据产生影响，使其响应时间增加，我们的负载均衡算法应当能够降低这种消耗。在效果相近的情况下，一般尽量迁移较少的元数据，而在 MDS 的选择中，尽量选择距离近的来达到降低消耗的目的。

### 3.3.4 元素访问热度

文件系统中的每一个目录或者文件都有一个自己对应的访问热度来表示相应的节点访问数量的多少，初始时每个节点访问热度值定为 0，此后每当收到一个用户请求，相应的节点访问热度就增加 1。我们定义一个 T1 作为元数据访问热度的阈值，当一个节点的访问热度超过此阈值时它就成为热点数据。对于一个元数据的请求，不同时间内的访问热度对这个元数据现在的访问热度的反

映是不同的，间隔时间越长与这个元数据现在的访问热度关联越小，因此各个节点访问热度是随着时间进行持续衰减的，我们定义访问热度衰减函数为：

$$popularity=popularity-f(popularity,T) \quad （公式 3-1）$$

### 3.3.5 服务器资源消耗模型

我们从 CPU 占用率、内存占用率、网络带宽占用率来评价一个服务器的资源消耗。元数据服务器要对外提供对元数据的操作，需要消耗 CPU 资源来对元数据目录树进行定位，缓存元数据需要占用内存资源，需消耗网络资源来接受请求向用户返 回结果，根据元数据操作的这些特点我们选取了这三个作为负载指标。将内存占用率、CPU 占用率、网络带宽占用率进行加权求和就是元数据服务器的负载：

$L_i = W_1 \times L_{cpu} + W_2 \times L_{mem} + W_3 \times L_{network}$，其中 $W_1+W_2+W_3=1$，且 $0= L_i=1$ （公式 3-2）

$L_i$越大就表示当前这个服务器的工作负载就越大；若越小就表示当前这个服务器的工作负载就越小。在元数据服务器上有工作负载息信采集模块专门负责服务器工作负载的采集，它周期性地采集服务器上的负载指标，并按照权重计算出工作负载并报给负载均衡管理模块进行负载决策。

我们使用负载均衡度来衡量元数据服务系统的负载均衡的情况，负载均衡度定义如下：

$$L_i = \frac{1}{n}\sum_{i}\frac{|L_i - \bar{L}|}{\bar{L}}，其中 \bar{L} = \frac{1}{n}\sum_{i=1}^{n} L_i，且 0 \le L_i \le 1 \quad （公式 3-3）$$

当系统中各个节点的负载相同时 Ls 最小值为 0，系统的负载是完全均衡的；而系统中的负载越不均衡 Ls 的值就越大最大值为 1.负载均衡度阈值 T 表示系统负载均衡的状态已经处在不可接受的状态，MDS 的负载已经严重不均衡必须要对其进行调整。

有多种因素会导致在负载更新的过程中使负载同现大幅度波动，系统引入了平滑函数使曲线变得相对平滑以降低采集负载的误差：

$$L_{new}=(1-d)L_{last}+d\times L_{now} \quad 0<d<1 \quad （公式 3-4）$$

其中，$L_{last}$为前一次进行平滑计算后的负载值，$L_{now}$为当前采集的负载值，d 为平滑因子，$L_{new}$就是经过计算后的平滑的负载值。

### 3.3.6 组内负载均衡策略

LFS 的元数据服务系把元数据分成了很多个组，每个组内的服务器负责同一组元数据的服务，每个组内服务器的数量是不一定的，而且是动态调整的。我们负载均衡的这第一个策略是在任务被分配出去之前实施的，在一个请求还在索引服务器中的时候，索引服务器计算出该请求数据所在的组。该组中有多台服务器，索引服务器中有所有元数据服务器的负载信息，索引服务器就根据元数据服务器的负载信息，选取负载最低的服务器处理该请求。

### 3.3.7 组间负载均衡策略

元数据服务器负载均衡问题实际上就是元数据在服务器中的分配问题，要调整各个服务器的负载就是要将元数据在各服务器间进行合理的分配。一般的，如果元数据分割策略采用的是子树划分策略的话，就可以采用动态子树迁移或者动态子树复制策略来调整服务器负载。但是我们对元数据的分割是采用静态哈希策略，不能通过迁移或者复制目录子树的方法来调整服务器负载，我们是通过在组间迁移服务器来动态调整各服务器负载的。

组间服务器迁移策略：在组间进行负载调整，通过将负载较低的一个服务器迁移到负载较高的组内来调节点各个组间的负载均衡。这种策略跟通常的迁移目录子树的策略思路刚好相反，迁移目录子树策略将整个系统的处理能力当作整体，将任务在各个节点间分配。而组间服务器迁移策略将网络任务看作一个整体，将各个服务器处理能力在任务间进行分配，同样可以达到调整节点工作负载的效果。

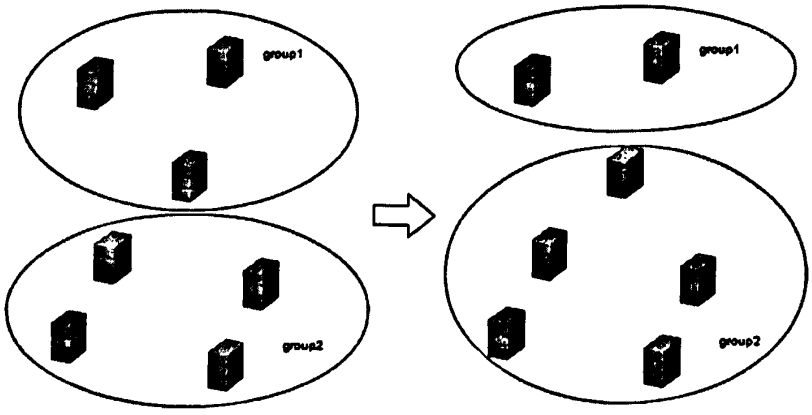



图 3.3 组间服务器迁移

组间服务器共享策略：这种策略跟组间服务器迁移策略本质上是一样的，只是有些许的差别。它是一样是通过将负载较低的内的一个服务器迁移到负载较高组内，但是并没有将这个服务器从原来的组内迁出，这样就形成了一个服务器同时在两个组内服务。这种组内服务器共享的策略通对将元数据的复制可以达到调节点负载的目的；而且通过在异地服务器间进行元数据的复制也可以降低用户请求异地数据时的开销。但是这种策略是一把双刃剑，因为它只从负载角度考虑问题，有可能会造成服务器元数据分布不均，也会由于承担过多的任务使服务器内存命中率较低。

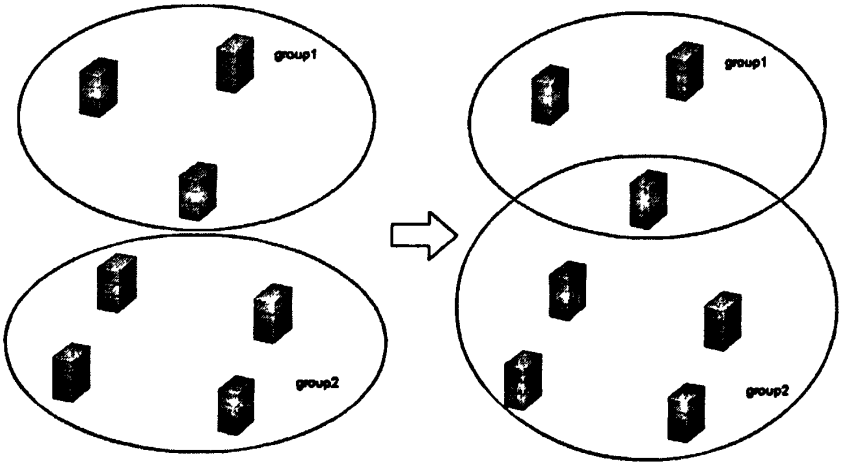



图 3.4 组间服务器共享

3.3.7.1 组间服务器迁移策略

索引节点会对各个组的负载进行实时统计，并计算出各个组间的负载均衡度，如果超过了规定的阈值，则启动组间服务器迁移。迁移服务器是在负载最低和负载最高的服务器间进行的，迁移策略必须能够保证在服务器被迁移后迁入和迁出的组内服务器节点的负载，防止产生抖动现象；服务器迁移后应该能够降低整个系统的负载均衡度而不会引发新的迁移。

1 准备阶段

1）根据各个组的负载情况，作出决策选择负载较高的组和负载较低的组

2）从决定迁出的组中选择出要迁出的服务器

2 迁出阶段

1）待迁出的服务器向数据服务器请迁入组的元数据信息

2）数据服务器收到请求后就会向请求的元数据服务器发送所请求的元数据

3）新的元数据信息接收完成后，对该元数据服务器加锁，使该服务器不能再接受新的请求

4）所有的请求都结束后，服务器切换到新的组中，关闭同步锁正式对外提供服务

#### 3.3.7.2 组间服务器共享策略

索引节点会对各个组的负载进行实时统计，并计算出各个组间的负载均衡度，如果超过了规定的阈值，则启动组间服务器共享。共享服务器是在负载最低和负载最高的服务器间进行的，共享策略必须能够保证共享服务器的负载，防止产生抖动现象；服务器迁移后应该能够降低整个系统的负载均衡度而不会引发新的迁移。

1 准备阶段

1）根据各个组的负载情况，作出决策选择负载较高的组和负载较低的组

2）从负载低的组中选择出要共享的服务器

2 数据迁移

1）共享的服务器向数据服务器请迁入组的元数据信息

2）数据服务器收到请求后就会向请求的元数据服务器发送所请求的元数据

3）新的元数据信息接收完成后，服务器切换到新的组中，对两个组的信息提供服务

### 3.3.8 热点数据服务器

在元数据服务器中有一个特殊的组叫做热点数据服务器组，在该组中的服务器中保存的数据都是热点数据，热点数据并不是迁移到这些服务器上。热点服务器上保存的数据只是原来数据的副本，这些数据由它原来的数据服务器和热点数据服务器共同对外提供服务。当一个对于热点数据的请求到系统中时它首先会分配给它原来所在的服务器，如里这个服务器的负载在规定的阀值内那么它就自行处理这个请求，如果服务器的负载超过规定的阀值，它就会将请求转交给热点服务器由热点服务器来处理。

## 3.4 元数据的一致性

### 3.4.1 元数据一致性分析

分布式文件系统在三个地方存在元数据，元数据服务器存储设备、MDS 服务器缓存和客户端缓存。为了减少网络通信提高性能，系统会在客户端对服务器信息和元数据信息进行缓存。在对 MDS 负载进行均衡的过程中，为降低热门数据所在服务器的负载，系统会在多个 MDS 中对热门的元数据进行缓存。我们要保证的一致性就是要保证在元数据服务器存储设备中的数据与客户端缓存中的数据和 MDS 服务器中的数据一致。

我们保证缓存致性一般用两种方法，一种是存缓存信息有更新时，通知所有缓存了该信息节点对其进行更新，由于很多的缓存可能以后并不会用到，所以这种策略会产生大量不必要的更新，代价很大；还有另外一种方式就是在有缓存的信息更新后，将所有对该息信的缓存失效，在节点需要使用该缓存时再重新获取信息，这种方式效率比较高，可以避免不必要的更新，减少不必要的开销，目前被大量采用。

### 3.4.2 元数据的主从缓存架构

在 LFS 文件系统中，我们引入元数据副本来降低热点数据造成的服务器性能的影响，以提高元数据服务器的性能。为保证分布在不同服务器上的元数据的副本不会在同一时间被更新，文件系统就要求对元数据进行序列化的操作，以保证元数据操作的一致性。为达到保持副本一致性这个目的，LFS 文件系统中采用了基于主从结构的元数据缓存架构。即不论该元数据有多少副本，仅有一台服务器处理这个数据的更新操作，这个副本所在的元数据服务器就是这个元数据的主 MDS，这个数据节点就是主副本节点，除主节点以外的节点副本对该元数据只有读的权限而不能对元数据进行更新，它们是该元数据的从副本节点。

LFS 在其元数据的副本中不保存副本的分布信息，而是在需要分布信息时，向索引服务器 Master 发出请求。当主 MDS 对元数据进行更新时，它首先向索引服务器 Master 发出请求取得该数据的所有从副本的分布信息，然后对数

据的副本进行更新操作，保持数据的一致性。而当元数据的从副本主动要求新版本的时候，也首先从索引服务器 Master 取得该元数据的主 MDS，然后向主管理 MDS 发出请求取得最新的数据信息。

### 3.4.3 元数据状态锁

当元数据信息有更新时，更新首先会在主 MDS 上进行，然后主 MDS 向索引服务器 Master 请求到该元数据的从副本分布信息，接着向从副本所在的 MDS 发出失效信息。从 MDS 在收到数据失效的信息后，将该元数据副本的状态置为失效，后面如果该从副本所在的服务器收这个元数据请求，就会主动向主 MDS 发出请求，更新该元数据的副本。

为了在元数据更新时维持其一致性，当主元素的数据更新时，需要对元数据加锁，锁有三种状态：

（1）SYNC 状态时，该数据的所有节点都可以对元素做读操作，但是不可以做写操作。

（2）LOCK 状态时，主节点可以对元素做写操作，从节点不可以对数据做任何操作。

（3）GLOCKR 状态时，SYNC 正在向 LOCK 状态过渡。

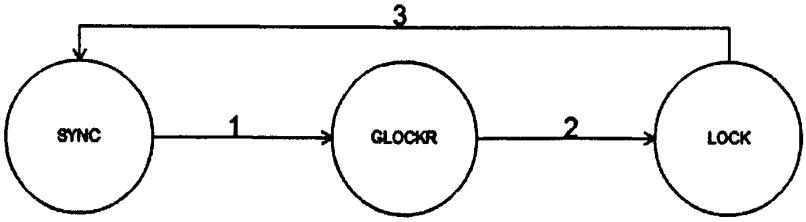



图 3.5 元数据锁状态转移图

过程 1：这个过程正在申请 LOCK 锁，但此时可能有的副本上的数据正在做读操作，加了读锁，所以要等待其他副本上的读操作完成，但是必须要阴止增加新的读操作，因此就将该元素的锁状态设置为 GLOCKR 以阻止其他从副本上读锁的申请，因此申请 LOCK 锁需要两个阶段的协议来完成，要先到 GLOCKR 状态。

过程 2：将元素状态锁的状态设没 GLOCKR 后，直到所有从副本上的读锁都结束之后，将所有从副本上的锁状态都设置为 LOCK 状态，并且通知主副本，所有的从副本都回复了之后将元素的锁状态改为 LOCK。

过程 3：这个过程是在对元素的写操作完成后发生的，这时如果没有其他等待写锁的操作，就将元素锁的状态由 LOCK 状态变为 SYNC 状态，并对所有从副本发出通知，使它们的锁状态也由 LOCK 状态转换到 SYNC 状态。

### 3.4.4 分布式日志

像对元数据进行修改这样的操作可以在一台 MDS 上完成时，使用元数据状态锁就可以了，而有的操作在并不能在只一台 MDS 上完成，像 rename、link、unlink 这种操作，可能都会涉及到两台不同的 MDS。这时如果一台服务器上的修改已经提交到硬盘了，但是另一台数据服务器发生故障使数据没有提交成功，就会造成系统数据的不一致。所以我们采用分布式日志的记录协议来解决这一问题：

1. 当对元数据进行分布式的更新时，首先有元数据的发起都更新缓存中的数据，记录操作日志，开始操作，接着向协作的服务器发出请求；

2. 若协作的元数据服务器无法执行操作，就向发起者返回一个操作失败；若操作成功，就记录操作日志，向操作的发起者返回一个操作成功；

3. 若发起者收到协作者的操作失败，则取消已经做的更改，并且使操作失败；否则，若收到协作者发出的操作成功，则记录日志，并通知协作者操作成功；

4. 协作服务器收到操作成功的消息后，记录操作日志，整个更新操作就成功了。

## 3.5 元数据的可靠性策略

### 3.5.1 元数据可靠性分析

元数据服务器的量会随着文件系统容量的增加而增加，MDS 服务器及各种服务器节失效率也会随着服务器数量的增加而增加。我们主要是从三个方面来维持元数据的可靠性，一是元数据服务器节点失效检查机制，二是元数据服务器故障恢复与排除机制，以及评价元数据服务器可靠性的方法。

检测元数据服务器是否失效一般采用心跳技术，由元数据服务器定期向指定节点报心跳信息，来说明自己在这一段时间内工作正常。

恢复处理元数据服务器故障一般采用两种方法，一是采用备分机制，通常情况下，由元数据服务器的主服务器来处理对元数据的请求，在数据有更新时也是由主服务器来对各个备分进行同步；如果主服务器失效，则由主服务器的从数据服务器接替主服务器的工作。在多元数据服务器架构中，如果让每一台元数据服务器都有专门的从备分服务器，那就使大量的服务器并没有对外提供服务，使大量的服务器资源都浪费了。所以实际应用中一般采取让所有系统中的元数据服务器既充当主服务器也充当另一台服务器的从备分服务器，当某台主服务器故障时，则由它的从服务器暂时接替其工作，以保证服务不中断。采用这种方式，不必让每台数据服务器都有专门的从服务器，大大地提高了服务器的利用率，但是如果一台服务器发生故障会造成其从备分服务器负载过重，需尽快进行人工恢复。

备分机制保证了数据的可靠性，但是如果要使元数据服务器内容保证可靠性，一般采用日志技术。日志技术是用日志来记录对元数据的操作，如果无数据服务器发生了故障，就用日志来对元数据服务器上的数据进恢复。日志一般有两种使用方式，一种是回滚日志另一种就是写前日志。回滚日志是指在系统发生故障时通过之前操作记录的日志回滚元数据到一个一致的状态；写前日志要求系统在对元数据修改之前，必须先将操作写进日志中，可以不直接修改元数据，而是在需要时才按照日志修改元数据。采用写前日志可极大地降低数据的通信量，使得整个系统的性能得到极大地提升。本地文件系统中常用到日志系统，但是在分布试文件系统中，对元数据作的一个操作可能多个地方都会用得到，采用日志系统就可以用效地解决这个问题。

本节所介绍的维持元数据可靠性的方法有 MDS 节点管理及故障检测技术和基于操作日志的故障恢复技术。提出能够及时检测出节点故障的管理元数据的有效方法，根据操作日志可以使节点失效时可以在保证元数据服务不中断的条件下快速地恢复元数据。

### 3.5.2 分区域节点管理策略

元数据服务器集群采取分区域的管理方式，将所有的服务器分成若干个区域，每个区域内的节点都保存了一分相邻节点的列表，记录了这个区域内节点的信息，每人区域内的服务器节点就根据这个邻居节点列表得到整个区域的情

况。每个服务器节点负责监视邻居节点的状态，也会定期把自己的状态报告给邻居节点。

一台新的 MDS 启动时，一台 MDS 会被指定作为其邻居节点，这台启动的 MDS 会复制其被指定的邻居节点列表来当作他自己的邻居节点，这台 MDS 就负责监视列表中服务器的状态。它接着会向自己的邻居列表中的服务器报告自己的状态。其他的服务器在收到消息后，就会将这台服务器加入到自己的邻居列表中。

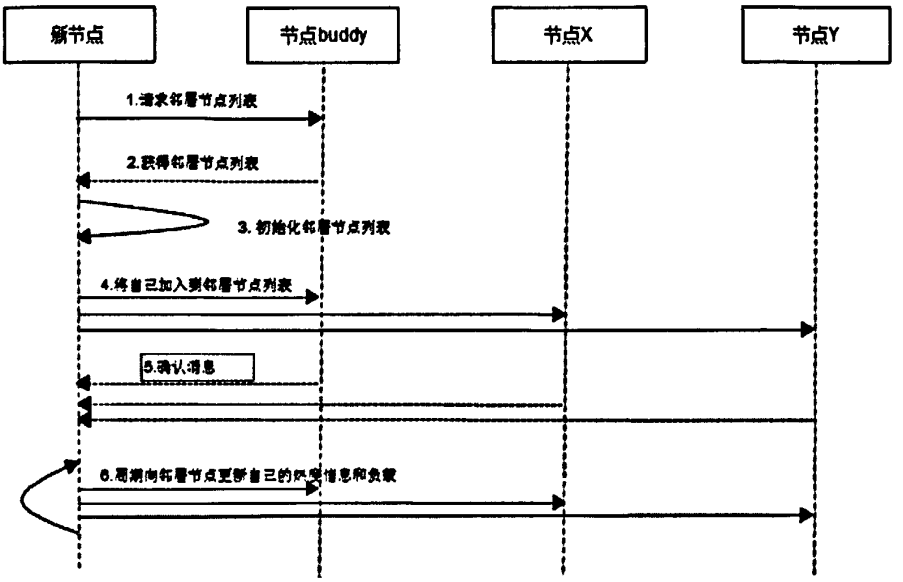



图 3.6 节点加入

当某台 MDS 退出时，也会向它邻居列表中的服务器发出消息，告知其自己已经退出，在收到邻居服务器发出的确认信息后，该服务器正常退出。邻居节点在收到通知后，将发出通信的服务器从自己的邻居列表中删除。

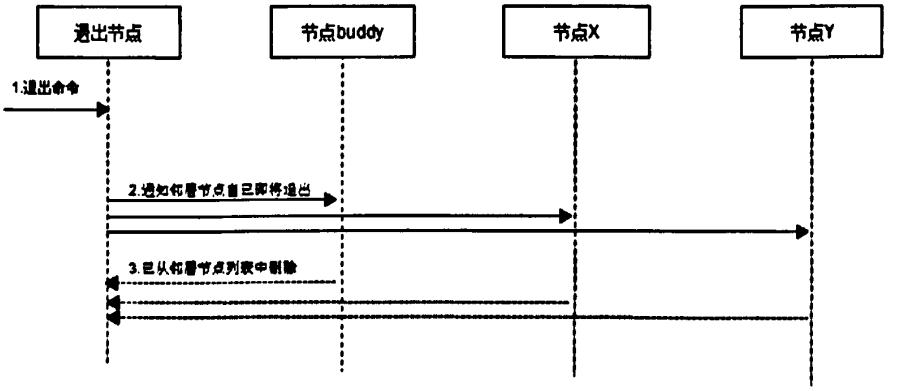



图 3.7 节点正常退出

当某台 MDS 异常退出时，它就不会向自己的邻居节点更新自己的状态，如果是这样，它的邻居节点就向该节点发出控测消息，如果还没有收到回复，那么就会报告服务器故障。

此外，在每个区域都会选择一个节点做为班长节点，各个区域的通信是通过各自的班长节点间的联系来进行的，如果有请求的元数据无法在本地找到，这个区域的班长节点就会向其他区域的班长节点查询该元数据信息。在一个区域内除了有一个班长节点，还会有一个副班节点，用来保证班长节点的可靠性，副班长节点平时是作为班长节点的备份的，当班长节点发生故障时，副班长节点才接替班长的工作处理请求，同时发起新一轮的选举，选出新的班长和副班长，之后由新选举的节点接替各自的工作。一般情况下选择负载最低剩余性能最多的节点作为班长节点，次之的为副班长节点。

### 3.5.3 元数据服务器的故障恢复

系统检测到系统中有机器发生故障后，就会开启报警机制，然后由人工决定是重启原来的 MDS 还是启用新的服务器来替代原服务器的功能。如果超出了一定的时间而没有做出人工选择，那么系统就会在已有的服务器中选择一个启动一个进程接替原 MDS 的工作。一般情况下，我们在系统中设置几个备分服务器来保证系统性能减少故障恢复时间，备分服务器平时不接收请求，只有在有服务器出现故障时，才会接替该服务器的请求处理，并进行故障恢复的工作。如果故障 MDS 排除故障后重新加入系统，它可以重新接管其原来负责的请求，也可为当作备分服务器。

元数据管理是基于共享存储的，由可靠的存储系统来负责保存元数据，存储系统也负责维护元数据的可靠性。MDS 并没有实际地存储元数据，所以也不需要恢复它之前所管理的元数据。MDS 采用日志技术记录对元数据所进行的操作，对元数据许多的修改并没有实时地记录到硬盘中。所以我们必须要读取故障 MDS 的修改日志，根据日志将元数据更新到磁盘上。

MDS 记录元数据的修改采用的是写前日志，我们在将数据写入到磁盘前必须先记录日志操作，日志完成后便返回给用户更新操作成功，所做的更新可以不必立即更新到磁盘中，而是在有对该数据的请求时，再根据日志将所做的修改写入到实际磁盘中。写前日志减少了操作的等待时间和数据的通信量，不必每作一次修改就写一次数据，可以在多次更新后一次性地提交，对同一元数据

作过多次修改后，只要提交最后一次的修改，这样就大大提高了整个系统的性能。

MDS 操作日志在存储系统中保存着，所以如果 MDS 发生故障并不会影响到日志，日志采用分页结构，一页写满后马上将该页的更新写入硬盘，再重新开一页新的日志，提交过写入磁盘的日志可以删除。记录日志时是按照 MDS 操作顺序记录的，提交的时候就不必按照顺序全部提交了，在日志中都记录了每个操作后数据的状态，所以相同数据我们只需提交最后一次修改即可。

采用写前日志可以保证对数据做的更新可以被写进磁盘，我们要维持命名空间的一致性，也需要恢复元数据服务器缓存中的信息。数据量比较大的许多状态数据并没有被记录到日志中以维持日志文件的大小在一定的范围之内，还有很多需要同其他元数据服务器协调完成的涉及到多台数据服务器的操作，所以我们的故障恢复协议如下：

（1）恢复 MDS 从操作日志中读取修改了但未提交的内容，将其加入到缓存中。日志中记录的某些操作可能没有成对出现，这时需要向别的 MDS 查询该操作是否已完成。

（2）恢复 MDS 向其他的 MDS 查询该分布式操作的状态。若 MDS 返回该操作的状态是成功的，那么就正常的提交恢复该操作；若返回该操作的状态是不成功的，则恢复 MDS 根据其操作的日志来决定该操作成功与否，如果该操作在日志中状态为成功的，则通知相关的 MDS 该操作已经成功；否则就失败。

（3）恢复 MDS 向客户端查询打开文件的锁状态和文件的状态，根据收到的信息重新与客户端建立会话。

（4）恢复 MDS 向其他的 MDS 声明对主元数据的所有权，之后，其他 MDS 将自己的相关副本的状态返回给恢复 MDS。根据所收到的回复恢复 MDS 决定对主元数据的上锁状态。决定完成之后故障恢复就算完成了，恢复 MDS 可以重新开始处理请求了。

以上的故障恢复协议保证恢复 MDS 缓存的内容。在这个 MDS 处理故障过程中，客户端如果连不上这个 MDS 便会尝试其他的 MDS，一台 MDS 的缓存故障并不会对其他 MDS 产生影响，而只决定了这一台 MDS 的故障。故障 MDS 负责的元数据操作可以由备分服务器接替也使这一块的操作顺利进展。

## 3.6 本章小结

本章主要分析在元数据管理方面的各项技术，并从其他的系统中借鉴了它们在元数据管理方面的优点，来设计出我们分布式文件件系统的策略。首先分析了元数据系统的设计目前标，元数据系统应该具有高性能、灵活性、一致性和可扩展性。我们接着又分析了元数据分割的策略，元数据分割策略一般分为元分割策略、静态子树分割策略、动态子树划分策略、静态哈希策略等，本系统中采用的是静态 hash 方法，以文件作为分割粒度，可以使分布比较均匀，访问元数据的速度比较快。对文件交发修改时是以文件为同步锁粒度的，并发程度比较高。又分析了元数据服务器的负载均衡策略，在一个元数据组内采用轮询的策略，始终将请求交由负载最低的服务器去处理，在各个组间的平衡采取组间服务器迁移策略和服务器共享策略来调节。接着分析了元数据的一致性策略和可靠性策略，元数据的一致性采用主从缓存架构和元数据状态锁保证，可靠性采取节点区域自治和服务器故障恢复策略保证。

# 第四章 数据副本

分布式文件系中的节点在性能及可靠性等方面各不相同差别很大，文件系统要合理协调系统中的节点使他们组成一个可靠的高性能的整体。所以在分布式文件系统中副本的创建和管理就是系统设计的主要目标之一，有了资源的副本后我们就可在不同的机器上同时访问同一文使系统的响应时音大大降低，即使节系统中的某些节点出现故障系统还是可以正常对外提供服务，增加了系统的可靠性，有了副本后就不需要专门去对系统的数据做备分了。但是采用副本策略也是会带来一些问题的，创建副本必然会消耗大量的磁盘资源，不过现在磁盘已经相当廉价了，其次我们要去管理这些副本，副本建立后需记录副本信息以准确地找到这些副本，副本创建得越多修改这些文件时同步的开销就越大，副本也会使系统的复杂度大大增加。副本创策略是副本管理的最关键技术，什么时候创建副本，在哪个节点上创建副本以及副本的一致性问题等是副本管理策略的重要内容。好的副本管理策略可以使整个文件系统的容错能力、性能大大提高。

## 4.1 请求分发策略

### 4.1.1 调动方式

（1）地址转换

地址转换是一种类似防火墙的结构，服务器结点被内部 IP 地址同外部网络隔开，客户端跟服务器之间无法直接通信。请求数据和应答数据都要经过负载均衡器对其 IP 进行处理，负载均衡器改写 IP 包的地址信息使请求数据重新指向内部的主机，内部的应答数据同样经过负载均衡器修改后发送给请求者。

（2）IP 隧道

IP 隧道采用开放网络的结构，服务器具有外部网络的 IP 地址，可以跟客户端直接通信，通过路由器真接将数据包返回给客户端。负载均衡器只对系统中的请求数据包进行处理而对于返回的数据包则不再处理。

负载均衡器通过 IPIP 协议将客户的请求包重新处理，将原来的数据包封装在新的包中形成选定目标 IP 的新的 IP 包。服务器将收到的 IPIP 包解开，根据客户端源地址直接将结果伟回给客户端。

（3）直接路由

直接路由跟 IP 通道比较相似，服务器也拥有合法的 IP 地址可以跟客户端直接通信，应答数据也是直接返回给客户端。不同的是在直接路由模式中，负载均衡器改写服务器的 MAC 地址来实现调度，服务器跟负载均衡器必须位于同一个网段中。

经过经较分析，以上的调度方式都不适合应用在本系统中。本系统的调度是在应用层的，通过特定的应用程序来调度平衡，这样的方式对服务器管理员的要求不高，设置也比较简单，具有较强的易用性。

### 4.1.2 平衡算法

（1）轮转法

轮转算法最最容易实现的调度方法，同一个网络的所有服务器具有相同的地位，轮转算法各个服务器间轮转选择依次将收到的请求分发给各个服务器，每个服务器的地位都是相等的。这种算法在系统的的结点性能处理能力等方面差别不大时比较适用，在系统各节点性能有差别时就不太理想了，而且也没有考虑服务器当前状态也可能使平衡结果不理想。

（2）随机法

随机法是根据由伪随机数算法产生的值，选择一个服务器来处理当前请求。跟轮转算法相似，随机法也是将各个结点视作平等的而进行无差别的选择，也适用于系统中各个结点性能处理能力等方面差别不大的情况。

（3）散列法

散列法是根据提前设定的 hash 函数将请求映射到系统中的结点中。散列法法最关键的部分就在于 hash 函数的设定，这直接关系到负载平衡的最终结果。

（4）最少连接法

最少连接法将系统中服务器实时的连接数作为指标，将请求发送到连接数最少的服务器。系统中服务器所允许的连接数目各不相同，每个连接所占用的带宽也不一样的，只凭最少连接数并不能准确地反映服务器的负载。

（5）最小流量法

最小流量法是根据服务器当前所占用的带宽，将请求发送给占宽占用最低的服务器。同最少连接法存在同样的问题，单凭带宽的占用量也不能准确地反映服务器的负载。

（6）最快响应法

这种方法记录到每个服务器的响应时间，将新请求分配给响应时间最小的服务器。这种算法应用在广域网中效果很明显，而如果是应用在局域网中则效果不大。

### 4.1.3 基于最快响应的分发策略

当客户对系统中的文件发出写请求时，由于分布式文件系统中存在多个同一文件的拷贝，分件系统就要为客户的请求选择一个拷贝来响应用户的请求。请求分发的关键点在于既要考虑到整个文件系统的负载平衡也要为用户选择一个距离相隔最近的服务器为其服务。例如在某个文件在北京和武汉两台机器上都有拷贝，如果我们完全按照轮转的方式来选择服务器那么就会经常舍近求远选择到不适合的服务器。如果我们完全选择距离近的服务器那么可能会造成服务器负载不均衡，没有使所有的拷贝都发挥出作用。

分法算法是在元数据服务器上运行的，而元数据服务器任务一般都比较重，所以我们的分发算法必须足够简单。其次我们要优先使客户请求分发到距离近的存储结点上，但随着服务器负载的增加要使部分的用户请求被分配到其他结点上，以均衡整个系统的负载。我们设计的副本选择副本算法如下：

```
ChooseReplica(c1,x1)
{//c1 表示发出请求的客户，x1 表示 c1 所要访问的文件
p=minDistance(D1,c1)      //D1 表示有 x1 副本的结点的集合，p 表示 D1 中与 c1 距离最近的结点
q=minLoad(D1)      //q 是 D1 中负载最低的结点
if(Load(p)>2*Load(q))      //Load(p)表示结点 p 的负载
  return q;
else
  return p;
}
```

Load(p)各节点的负载信息是根据服务器的 CPU 利用率、带宽占用率以及内存使用率加权取得的：$Load = W_1 \times L_{cpu} + W_2 \times L_{mem} + W_3 \times L_{network}$，其中

$W_1+W_2+W_3=1$，且 $0=L_i=1$.其中，$L_{last}$ 为前一次进行平滑计算后的负载值，$L_{now}$ 为当前采集的负载值，d 为平滑因子，$L_{new}$ 就是经过计算后的平滑的负载值。

当客户 c1 发出一个请求时，首先找到与 c1 距离最近的结点和负载最低的结点，然后比较两者的负载差值是否是在 2 倍以内（可根据实际情选择不同的值），若是选择距离近的结点，否则选择负载最低的。

根据这个算法，如果我们系统中有两个副本分别在北京和武汉，多个武汉的客户来对这个文件进行访问时，首先会将请求都转交由武汉的服务器来处理，但是当武汉的服务器的负载过高，超过北京服务器的两倍时，这表示系统中的负载严重不均衡，这时再有从武汉发出的请求就会交由北京的服务器来处理。该算法也存在一个问题，如果武汉的服务一开始就因为处理其他的请求而使负载过高时，此时我们从武汉再访问这个文件时，所有的请求都会由北京的服务器来处理。

## 4.2 副本放置策略

系统采用的副本放置策略是一种主动调整系统负载的策略，它的出发点是在创建副本是就已经开始考虑副系统负载问题，主动选择最佳的位置放置副本，一般是选择负载最轻的节点，有利于调整整个系统的负载降低负载不平衡的潜在风险。这处策略在放置副本时避免了随意放置而是根据节点的负载情况和磁盘占用量来选择合适的位置。

### 4.2.1 副本放置位置

在写一个文件到系统中时默认会产生三个副本，包括两个缺省副本和一个主副本，其中会在本地机架上保存一个缺省副本和主副本而另外再选择一个除本地机架以外的机架放置另外一个缺省副本。

机器的选择是看两个指标，磁盘占用率和系统的负载。设第 i 台机器磁盘点用率为 Ni，系统负载为 Load(i)（计算方法跟上一节相同），则变量

P=(k1Ni)/(k2Load(i)),其中 k1，k2 为常数　　（公式 4-1）

首先计算本地机架节点的 P 值，在 P 值最小的两台机器上分别创建主副本和缺省副本，然后计算远程机架节点的 P 值，在值最小的机器上创建缺省副本。当

然选择机器时要首选跳过已存在这个数据副本的机器和空间不足以存放这个数据的机器，整个过程如下图所示：

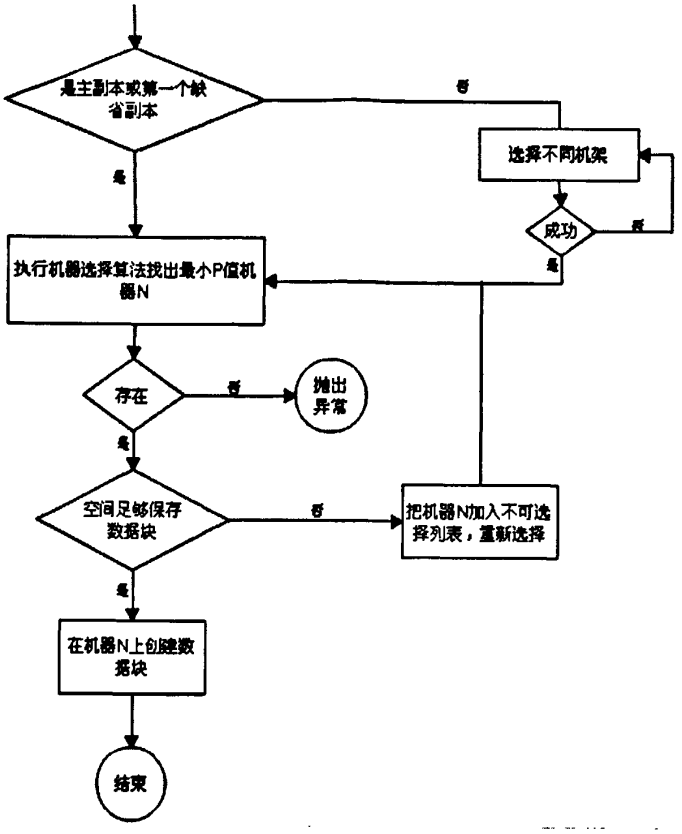



图 4.1 副本创建流程

### 4.2.2 动态副本创建策略

我们采用一种称作负载水位的技术来维持存储系统的稳定性，高负载水位是根据存储结点的性能决定的，结点的工作负载一旦超过了高负载水位性能就会急剧下降。低负载水位也是由节点的性能决定的，而且值必须低于高负载水位。还有两个阀值决定副本放置选项，如果一个副本访问量低于 u，那么它就有可能会被删除，如果访问量高于 m 该副本就有可能被复制到其他节点。

当文件在系统中创建好后，在对外提供服务的过程中会记录文件的实时访问情况，当访问次数超过规定的阀值后，系统会为数据创建额外的副本。由于数据访问的延续性如果某个集群内数据访问量很大，那么在未来对此数据进行访问的可能性也很大，所在在创建副本时首先找到用户访问量最高的机架，然后选取这个机架中最适合的一台机器（根据上一小节的策略）。这种策略将数

据复制到访问量最高的机架，有效地分担了系统中的服务压力。这种选择出最佳机架的策略需要维护一张历史访问的记录表，该表中记录着在一段时间内每个文件副本被访问的次数。当机器的负载高于高负载水位时就会检查访问次数是否超过了指定的阈值，如果有这样的文件就先找出最合适的机架，然后从这个机架中选出负载最轻的服务器，在这个服务器中创建这个文件的副本，并重新开始统计该文件的访问记录。整个过程如下图所示：

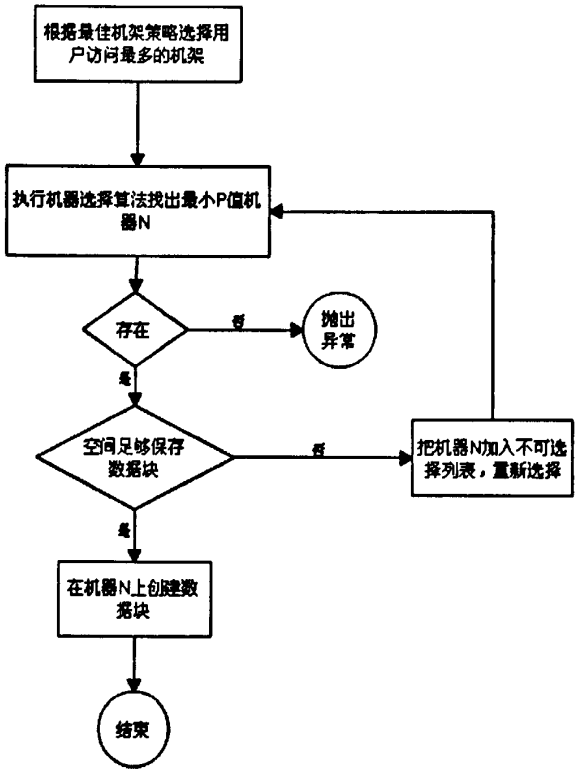



图 4.2 动态副本创建流程

副本的删除也是文件系统要考虑的重要因素，副本的动态创建会使一个数据块存在多个副本，如某数据在某一时刻访问量很大会使系统中增加了许多这个文件的副本。这些副本在数据访问量降下来之后并不能起到提高系统性能的作用，反而会是长是时间不用大量浪费存储空间而且还会增加系统的管理代价，因些删除多余的副本是必要的。删除也是根据副本的访问量动态地删除副本，在一定的时间内，副本的访问次数如果低于一个设定的阀值，那么就要删除这个副本。

```
Placement()
{
```

```
if (Load(s)>hW)   offloading=YES;//工作负载已经高过负载水位
if (Load(s)<lw)   offloading=NO;
for each x
{
 if (popularity(x)<u)
{
if (x 副本>=3)
  reduce(x);  //
}
else if (offloading==YES && popularity(x)>m)
  replica(x);  //多产生一个 x 的副本
}
}
```

由以上算法可以看出当一个节点的负载高于高负载水位时，它就会额外再产生新的副本来分担它的压力。系统采取这种分流用户请求的方式而不是迁移数据到其他的服务器，可以避免迁移数据导致的负载波动而接着引发新一轮的迁移。

## 4.3 本章小结

分布式文件系中的节点在性能及可靠性等方面各不相同差别很大，文件系统要合理协调系统中的节点使他们组成一个可靠的高性能的整体。所以在分布式文件系统中副本的创建和管理就是系统设计的主要目标之一，有了资源的副本后我们就可在不同的机器上同时访问同一文使系统的响应时音大大降低，即使节系统中的某些节点出现故障系统还是可以正常对外提供服务，增加了系统的可靠性，有了副本后就不需要专门去对系统的数据做备分了。

本章首先分析了请求分发策划略，采用了一种称作基于最快响应的分发策略，在存在多个副本本的时候系统首先会考虑将请求分发给响应最快的服务器，当响应最快的服务器负载过高时，将请求分发给负载最低的服务器。接着分析了副本的放置策略，在写一个文件到系统中时默认会产生三个副本，包括两个缺省副本和一个主副本，其中会在本地机架上保存一个缺省副本和主副本而另外再选择一个除本地机架以外的机架放置另外一个缺省副本。当文件在系统中创

建好后，在对外提供服务的过程中会记录文件的实时访问情况，当访问次数超过规定的阀值后，系统会为数据创建额外的副本。

# 第五章 系统功能测试

## 5.1 测试环境

我们文件系统中的各个模块都只是一个进程，由于受实验条件所限，我们将所有的模块都布署在一台服务器中，来测试我们文件系统的功能，具体配置如下：

表 5-1

| | |
|---|---|
| CPU | Intel Xeon X7560 2.266GHz×4 |
| 内存 | 16GB(4×4GB) |
| 硬盘 | 4×146GB SAS |
| 网卡 | 2×千兆接口 |
| 交换机 | 千兆交换机 |
| 操作系统 | SUSE Linux Enterprise Server |

## 5.2 测试内容

### 5.2.1 普通接口测试

（1）上传下载

1. 以流式方式从本地文件 A 读数据写入到 xfs 文件 B，完成文件上传
2. 以流式方式从 xfs 文件 B 读数据写入到本地文件 C
3. 比较本地文件 A 和 C 的 md5
4.本地文件 A 分别为 4M,64M,2G,10G 的文件构成 4 个数据驱动型 case

（2）文件备份因子测试

1. 设置文件备份因子 backup_factor
2. 以流式方式从本地文件 A 读数据写入到 xfs 文件 B，完成文件上传
3. 以流式方式从 xfs 文件 B 读数据写入到本地文件 C

4. 比较本地文件 A 和 C 的 md5

5. 获取 xfs 文件 B 的每一个 chunk 的备份因子，与步骤 1 中 backup_factor 对比

6.backup_factor 分别设置为 1、2、3、（NodeserverNum-1）*2 构成 4 个不同的 case

（3）流式小速率上传下载

1. 随机设置每次读写字节数 stream_bytes（4~10B）
2. 按照设置的 stream_bytes，从一个大小为 4K 的本地文件 A 读数据写入到 xfs 文件 B，完成文件上传
3. 按照设置的 stream_bytes，从 xfs 文件 B 读数据写入到本地文件 C 比较本地文件 A 和 C 的 md5

（4）批量写大块文件

1. 随机设置文件上传下载次数 multi_time（5~10）、每次读写字节数为 stream_bytes 为 16MB
2. 按照设置的 stream_bytes，从一个大小为 64M 的本地文件 A 读数据写入到 xfs 文件 B，完成文件上传
3. 按照设置的 stream_bytes，从 xfs 文件 B 读数据写入到本地文件 C
4. 比较本地文件 A 和 C 的 md5 重复步骤 1~4 multi_time 次

（5）Fileid 反查 filenam

1. 上传一个文件，记录该文件的文件名
2. 获取该文件的 fileid
3. 触发主 master 的 checkpoint，等待 checkpoint 结束
4. 通过主 master 的 search file id 反查该 fileid 的 filename
5. 检查反查到的 filename 与实际的 filename 是否一致
6. 删除该文件
7. 触发主 master 的 check point，等待 checkpoint 结束
8. 通过主 master 的 search file id 反查该 fileid 的 filename 检查是否查询不到 filename

### 5.2.2 备份功能测试

（1）单磁盘故障备份，立即启动备份

1.上传一个文件，k 备份

2. 获取其中某一 chunk 文件（如第 n 块 chunkn）所在的 nodeserver 列表，及最新的 chunk 文件生成时间

3.获取其中某一台 nodeserver 上的对应的该 chunk 文件所在的磁盘

4.设置该 chunk 文件所在的盘故障，启动备份

5.检查该 chunk 文件是否是 k+1 备份，且该 chunk 文块仅在该盘只有一个备份，最新的 chunk 文件生成时间新于原有的最新生成时间

6.上传批量文件，该盘没有新写的 chunk 文件

7.下载文件 md5 正常

（2）nodeserver 异常时（停掉），5 分钟后会启动备份

注：NS 个数是 3，文件备份数也是 3，停掉一台 NS 后找不到对应的机器来备份，因为每一个 chunk 只会在一台机器上只能有一个备份。可以采用 2 个备份或者币 Nsnum-1 个备份。备份时必须保证 chunk 文件可以从可用的 NS 上获取

1.上传一个文件，k 备份（采用）

2. 获取其中某一 chunk 文件（如第 n 块 chunkn）所在的 nodeserver 列表，及最新的 chunk 文件生成时间

4.stop 该 chunk 文件所在的一台 nodeserver，启动备份

5.检查该 chunk 文件是否是 k+1 备份，且该 chunk 文块仅在该 nodeserver 只有一个备份（onebox 无法检测），最新的 chunk 文件生成时间新于原有的最新生成时间（onebox 下最新的 chunk 文件生成时间和原有最新时间可能相同，可以在停掉 nodeserver 之前 sleep 1 秒钟）

6.下载文件对比 md5 正常

（3）停备份因子 m-1 台 nodeserver 时的备份：(m<=(n+1)/2)

1.上传一个 128M 文件，获取该文件的 fileid，备份数为(n+1)/2,n 为 Node Server 的个数

2.获取该 fileid 的某一块 chunk 文件所在的(n+1)/2 台 nodeserver 的 ip

3.从这(n+1)/2 台 nodeserver 中停掉其中(n+1)/2-1（即 m-1）台 nodeserver

4.备份结束后，重复步骤 2（排除被停掉的两台 nodeserver）

5.检查步骤 2 和步骤 4 的 fileid 对应的 chunk 文件是否为(n+1)/2，meta 信息已更新为新的 nodeserver 且下载的 md5 正常

6.重启被停掉的 m-1 台 nodeserver,恢复环境

（4）文件保存后某一个 chunk 文件仅一个 chunk 文件未被删除，再次读写时的备份

1.上传一个文件，获取该文件的 fileid，备份数为(n+1)/2, n 为 Node Server 的个数

2.获取该 fileid 的某一块 chunk 文件所在的(n+1)/2 台 nodeserver 的 ip

3.获取其对应的 meta 信息，从系统中删除掉其中(n+1)/2-1 个 chunk 备份

4.追加写这个文件，1.5 分钟后触发备份流程，备份结束后，重复步骤 2（排除被停掉的两台 nodeserver）

5.检查该 chunk 文件所在的 nodeserver 是否更新为正常的一台和新的两台 nodeserver（没有停 NodeServer，是否需要判断？？），检查该 chunk 文件备份数为(n+1)/2，且下载该文件 md5 校验正常

### 5.2.3 回收站功能测试

（1）非强制删除文件时，等价于 mv 到一个以固定的时间戳格式命名的回收站文件（~[filename].[ts]~，[filename]是原文件名，[ts]是将要被删除的时间戳，以秒为单位。如 a.txt 变成了~a.txt.20110401143000~形式的文件，即在 2011 年 4 月 1 日 14 点 30 分 00 秒，a.txt 文件将会被删除）

1.获取 master 设置的 recycle_file_expire_interval 回收站延时，单位 s

2.上传一个文件，记录文件名 filename，并记录其 fileid

3.非强制删除该文件,记录删除该文件的时间点

4.调用接口查看回收站文件，检查是否有一个~[filename].[ts]~，其中 ts 为步骤 3 中删除操作时间点基础上延时 recycle_file_expire_interval(可能有时间误差，需考虑到)，且文件 fileid 与 filename 的 fileid 一致

5.触发 schedulermaster 回收孤儿节点

6.原文件名不存在

（2）强制删除，即直接删除文件并进行回收，chunk 文件也同时被回收

1.获取 master 设置的 recycle_file_expire_interval 回收站延时，单位 s

2.上传一个文件，记录文件名 filename，并记录其 fileid

3.强制删除该文件,记录删除该文件的时间点

4.调用接口查看回收站文件，检查是否不存在一个~[filename].[ts]~，其中 ts 为步骤 3 中删除操作时间点基础上延时 recycle_file_expire_interval(可能有时间误差，需考虑到)

5.触发 schedulermaster 回收孤儿节点

6.原文件名不存在，且 chunk 文件被回收

（3）重命名回收站文件到另外一个文件是允许的

1.上传一个文件，记录文件名 filename，并记录其 fileid

2.非强制删除该文件,记录删除该文件的时间点

3.调用接口查看回收站文件，检查是否不存在一个~[filename].[ts]~，其中 ts 为步骤 2 中删除操作时间点基础上延时 recycle_file_expire_interval(可能有时间误差，需考虑到)

4.mv 该~[filename].[ts]~文件到新的~[filename].[newts]~文件，检查是否可以成功

5.检查新文件的 fileid 与 filename 的 fileid 一致

（4）创建一个带时间戳标记的文件或目录是被允许的，并按照时间戳标记的时间进行回收

1.上传一个文件，记录文件名~[filename].[ts]~，ts 为当前时间延后 30 秒，并记录其 fileid

2.等待 30 秒，连续触发 checkpoint 三次，每次需上一次完成后再执行

3.触发 schedulermaster 回收孤儿节点

4.查看回收站文件接口检查该回收站文件~[filename].[ts]~是否不存在，chunk 文件被回收

（5）同一秒中非强制删除 100 个以内的相同文件，会以时间戳加序号命名回收站文件，并到期后可以进行回收

1.同一秒内上传多个文件到同一个文件名 filename，并同时进行删除

2.查看回收站文件爱你接口检查是否生成~[filename].[ts].[seq]~文件，其中 seq 为 0-99 之间（当一秒内超过 100 个同文件名的删除时，则会拒绝）

### 5.2.4 节点回收功能测试

（1）正常删除时的节点回收

1.上传一个文件 2G，备份数为 2*（nodeservernum-1），记录 fileid

2.根据 fileid 从 nodeserver 上获取 chunk 文件分布
3.根据 fileid 获取 metaserver 的 groupid
4.rm 删除该文件到回收站，
5.触发 dump master 然后在 scheduler master 页面触发节点回收(或重启 metasever)
6.待回收完成后,重复步骤 2，检查是否所有的 chunk 文件均被回收
（2）孤儿节点回收
1.上传一个文件 2G，备份数为 2*（nodeservernum-1），记录 fileid
2.根据 fileid 从 nodeserver 上获取 chunk 文件分布
3.根据 fileid 获取 metaserver 的 groupid
4.停止 schedulermaster，从 master 和 metasever 删除该文件的相关信息，制造孤儿节点
5.从 metaserver 页面触发孤儿节点回收（页面已经不提供该链接，可以通过重启 metaserver 来回收孤儿节点，后期可能需要修改为重启该文件 meta 信息所在 metaserver 组的整组 metaserver，先停主，再停备;先起备再起主）
6.待回收完成后,重复步骤 2，检查是否所有的 chunk 文件均被回收
（3）批量回收后的已用节点统计
1.上传一个文件 4G，记录 fileid
2.根据 fileid 从 nodeserver 上获取 chunk 文件分布（需要获取该文件所有 chunk 块所在的 NS ip 列表，所以 case 中直接扫描了所有有效的 NS）
3.触发回收
4.记录可以记录整个集群的已用节点统计数。（或者被分配的 nodeserver 已用节点统计数。空余节点会受其他进程写磁盘影响而动态变化，而已用节点则是记录 NS 实际使用节点，不会受其他用户影响）
5.删除该文件并再次触发回收
6.回收完成后，检测集群总已用节点统计是否减少被回收的 chunk 文件数，且总节点数减去总空闲节点数等于总已用节点数
（4）存储节点全部被占用是的回收
1 批量上传文件，记录 fileid 列表
2.根据 fileid 从 nodeserver 上获取 chunk 文件分布
3.触发回收
4.记录整个集群的总已用节点统计数
5.删除上传的所有文件并再次触发回收

6.回收完成后，检查集群总已用节点统计是否减少被回收的 chunk 文件数，且总节点数减去总空闲节点数等于总已用节点数

### 5.2.5 chunk 文件校验测试

（1）头部检验

1.上传一个 1G 的文件，备份数为 3

2.根据 fileid 获取其 chunk 文件分布

3.修改第 0 块的两个 chunk 文件的头部的一个字节，保持另外一块不变

4.下载该文件，校验 md5 是否正确

（2）Checksum 校验

1.上传一个 1G 的文件，备份数为 3

2.根据 fileid 获取其 chunk 文件分布

3.修改第 0 块的两个 chunk 文件的 checksum 的一个字节，保持另外一块不变

4.下载该文件，校验 md5 是否正确

（3）Data 校验

1.上传一个 1G 的文件，备份数为 3

2.根据 fileid 获取其 chunk 文件分布

3.修改第 0 块的两个 chunk 文件的 data 的一个字节，保持另外一块不变

4.下载该文件，校验 md5 是否正确

（4）整个 chunk 文件替换为其他 chunk 文件

1.上传一个 1G 的文件，备份数为 3

2.根据 fileid 获取其 chunk 文件分布

3.修改第 0 块的两个 chunk 文件为其他 fileid 的 chunk 文件，保持另外一块不变

4.下载该文件，校验 md5 是否正确

### 5.2.6 破坏测试

（1）停掉备份数-1 个 nodeserver 时可以正常读写文件

1.上传一个 128M 的文件，备份数为 2

2.根据 fileid 获取其 chunk 文件分布

3.停掉第 1 块的两个 chunk 文件的 nodeserver

4.下载该文件，校验 md5 是否正确

5.追加写该文件

6.下载该文件，校验 md5 是否为追加后的文件 md5

（2）停掉一组 metaserver 时，其他 metaserver 上的文件可以正常读写

1.上传两个文件，fileid1 和 fileid2,分布在 metaserver1 和 metaserver2 上（metaserver1 和 metaserver2 为不同组）

2.下载 file1 和 file2，检查两个文件的 md5 是否正确

3.停掉 metaserver1 所在的组的所有 metaserver

4.检查 file1 是否下载失败，file2 下载正常且 md5 正确

（3）备 master 的文件数据与主 master 的一致

1.检查集群是否有 1 个主 master 至少一个备 master

2.上传一个文件名为 filename，获取 fileid

3.触发 dumpmaster

4.在备 master 通过 search file id 获取该 fileid 的 filename_secondary

5.对比 filename 与 filename_secondary 是否一致

（4）Master 主备切换，新主 master 的 quota 数据与老主 master 一致

1.检查集群是否有一个主 master 和 2 个 second master

2.获取主 master 上的某一个 role 的配额和已使用的 filenum、dirnum、chunknum quota

3.重启主 master，检查是否由新的 master 升级为主 master

4.获取新的主 master 上的某一个 role 的配额和已使用的 filenum、dirnum、chunknum quota

5.对比前后两次获取的 quota 是否一致

## 5.3 本章小结

本章从文件系统的各方面功能对我们的文件系统作了详细的测试，本系统已经具备了良好的读写性能。

# 结束语

在信息化的今天各种信息都爆炸式增涨，在各个领域都有大量的数据，不断膨胀的信息使得信息的存储越发的重要。网络存储是一种降低存储成本提高存储资源利用率的有效途径，网络存储还可以提供更高的可靠性和安全性。因此针对网络存储这一方面和研究，已经被各大互联网公司所重视，纷纷提出自己的分布式文件体系统。

本文主要提出了一种分布式文件系统的架构模型，并对其中的设计策略做了详细介绍。本系统的设计主要是针对搜索引擎的，应用在搜索引擎上这种需要存储海量数据的应用场景中。本系统的元数据集合单元数据服务器和多元数据服务器的优点，既避免了元数据服务器成为系统瓶颈，也比较容易对副本进行维护。元数据虽然只是分布式文件体系统中的一个模块，但是它的地位确是至关重要的，所有的操作首先都要经过元数据服务器，元数据设计的好坏可以决定很大一部分整个文件系统的性能。分布式文件系统负载均衡策略对整个系统的性能至关重要，本系统中采用了动态负载均衡的策略动态调整个各服务器的负载。为防止某些服务器故障系统默认会为数据设置多个副本，并会在多个副本间维持负载的平衡。

本文就是通过分析对比各种分布式文件系统，吸取各个文件系统的优点，设计出了一个面向海量数据的分布文件体系统。第一章绪论，介绍了本论文的研究背景以及分布式文件系统的研究现状，并给出了本文主要内容。第二章分布式文件系统架构，首先分析了需求并给出了本系统的设计目标，最后出了本文件系的整个架构。第三章元数据系统，先分析元数据系统设计目标，接着给出元数据的分割策略，负载平衡策略以及数据一致性策略。第四章数据副本，介绍了请求的分发策略和副本的放置策略。第五章功能测试，介绍了对文件系统各个模块的测试用例。

# 致 谢

本论文的研究和撰写，是在导师钟珞教授的悉心指导和帮助下完成的，在研究生三年时间里，钟老师给了我很大的帮助。他严谨的治学态度、渊博的专业知识以及忘我的工作热情对我的研究生生涯有着非常深远的影响，特别是钟老师深厚的专业背景和精湛的技术能力一直激励我不断努力学习和工作，向着更高的人生目标奋斗!在此，对钟老师辛勤的培育与教导表示衷心的感谢!

三年的求学生涯中，与宿舍的同学周方云一起度过了最难忘的日子，谢谢你一直以来对我的支持与鼓励。同时，感谢刘豪、余尧、蒋幸以及师兄王清波在隧道项目中给予我的帮助。还要感谢许壮、徐山，我们共同探讨问题，互相勉励，一起度过难忘的研究生生活。

另外，要特别感谢我的亲人，谢谢你们一直对我的关爱和支持，正因为有你们，才有了让我一直前行的动力。

最后，感谢各位评委老师百忙中审阅我的论文并提出宝贵意见。

# 参考文献

[1]NanetteJ.Boden，DannyCohen，Robert E.Felderman，et al.Myrinet:AGigabit-Per-Seeond Loeal Area Network.IEEEMicroarchive，1995，15(l):29~36

[2]DavidGinsburg.ATM Solutions for Enterprise Intemetworking (2ndEdition). Addison-Wesley Professional,1999

[3]JurgensC.，Ancor Commun. Fibre Channel: a cornnection to the future. Computer，1995(28):8，88~90

[4]D.Pendery，J Euniee.InfiniBand ArehiteetUre:Bridge Over Troubled Waters，Note， InfiniBand Trade Assoeiation，www.infinibandta.com，2000

[5]GM:the low-level message-passing system for Myrinet networks，http://www.myricom.com/scs/index.html

[6]Soiehiro Araki，Angelos Bilas，Cezary Dubnieki， et al.User-Space Communieation: A Quantitative Study.In Proeeedings of The 1998 SC98 conferenee(SC98)，Oriando，Florida，1998

[7]VonEicken，T.Basu，A.Buch， et al.U-Net: A User-Level Network Interface for Paralleland Distributed Computing，In Proeeedings of SOSP，1995

[8]David E. Culler，Lok T. Liu，Richard P.Martin，et al.Assessing Fast Network Interfaces. IEEE Micro，1996，16(1):35~43

[9]S. H.Rodrigues，T.E.Anderson，D. E. Culler.High-Performance Local-Area Connrnunieation Using Fast Sockets. In Proeeedings of the 1997 USENIX Conferenee，Anaheim，CA，1997.257~274

[10]刘炜，郑纬民，申俊等.底层通信协议中内存映射机制的设计与实现.软件学报，1999，10(l):24~28

[11]ComPaq Computer Corp.，Intel Corporation，Mierosoft Corporation. Virtual Interface Aichiteeture Spceifieation.Version1.0.www.viarch.org，1997

[12] R. Sandberg. Summer 1987. The Sun Network File System: Design, Implementation and Experience[C]. IN Proceedings of USENIX Summer Conference: 300-313.

[13] Sidebotham. B. 1986. Volumes: The Andrew File System Data Structuring Primitive. Technical Report CMU-ITC-053[S], USA: Carnegie Melon University.

[14] J.J. Kistler, M. Satyanarayanan. 1992. Disconnected Operation in the Coda File System [J]. ACM Transactions on Computer Systems: 3-25.

[15] J. K. Ousterhout, A. R. Cherenson, F. Douglis, et al. 1988. The Sprite Network Operating System [J]. IEEE Computer: 23-36.

[16] Randolph Y. Wang, Thomas E. Anderson. xFS: A Wide Area Mass Storage File System. Technical Report, University of California at Berkeley, 1993

[17] Chandramohan A. Thekkath, PTimothy Mann, Edward K. Lee. Frangipani: a scalable distributed file system. ACM SIGOPS Operating Systems Review, Dec. 1997, 31: 224-237

[18] 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博士学位论文.2005

[19] Meth K Z,Satran J.Design of the iSCSI protocol.In:Titsworth F,ed.Proceedingsof the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies.Piscataway[J]:IEEE,2003.116~122

[20] IBM Storage Tank,A Distributed Storage System.White Paper[S],IBM Corporation,January 24,2002

[21] Lu Y P,David H C D,Ruwart T.QoS Provisioning Framework for an OSD-based Storage System.in:Anon,ed.Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies[S],2005.28～35

[22] R. Sandberg. Summer 1987. The Sun Network File System: Design, Implementation and Experience[C]. IN Proceedings of USENIX Summer Conference: 300-313.

[23] Sun Microsystems, Inc. March 1989. NFS: Network File System Protocol Specification[S]. USA: RFC.

[24] B. Callaghan, B. Pawlowski, P. Staubach, Sun Microsystems, Inc. June 1995. NFS Version 3 Protocol Specification[S]. USA: RFC.

[25] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, Sun Microsystems, Inc. C. Beame, Hummingbird Ltd. M. Eisler, Zambeel, Inc. D. Noveck, Network Appliance, Inc. December 2000. NFS Version 4 Protocol Specification[S]. USA: RFC.

[26] Sidebotham. B. 1986. Volumes: The Andrew File System Data Structuring Primitive. Technical Report CMU-ITC-053[S], USA: Carnegie Melon University.

[27] AIX Systems Support Center. 1998. AFS Distributed File System FAQ[S]. UK: IBM.

[28] J.J. Kistler, M. Satyanarayanan. 1992. Disconnected Operation in the Coda File System [J]. ACM Transactions on Computer Systems: 3-25.

[29] Brama, P. J. June 1998. The Coda Distributed File system [J], LINUX Journal.

[30] J. K. Ousterhout, A. R. Cherenson, F. Douglis, et al. 1988. The Sprite Network Operating System [J]. IEEE Computer: 23-36.

[30] Randolph Y. Wang and Thomas E. Anderson. 1993. xFS: A Wide Area Mass Storage File System[C]. In Proceedings of the Fourth Workshop on Workstation Operation Systems: 71-78.

[32] Steve Soltis, Grant Erickson, Ken Preslan, Matthew O'Keefe, and Tom Ruwart. 1998. The Design and Performance of a Shared Disk File System for IRIX [C]. Fifteenth IEEE Symposium on Mass Storage Systems.

[33] Frank Schmuck and Roger Haskin. January 2002. GPFS: A Shared-Disk File System for Large Computing Clusters [C]. Proceedings of the Conference on File and Storage Technologies (FAST'02): 231-244.

[34] PANASAS WHITE PAPER, 2003. Object Storage Architecture, http://www.panasas.com.

[35] Peter J. Braam, RumiZahir. July 29, 2001. Lustre Technical Project Summary [S], Version 2.

[36] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. October 19-22, 2003. The Google File System [C], Proceedings of the nineteenth ACM symposium on Operating systems principles.

[37] K. Shvachko, H. Huang, S. Radia, and R. Chansler. May 2010. The Hadoop Distributed File System[C]. In 26th IEEE(MSST2010) Symposium on Massive Storage Systems and Technologies.

[38] 田颖. 2003. 分布式文件系统中的负载平衡技术研究[D]: [硕士学位论文]. 北京: 中国科学院计算技术研究所.

[39] 杨德志，许鲁，张建刚. 2008. 蓝鲸分布式文件系统元数据服务[J].计算机工程: 4-9.

[40] Hyeran Lim, Vikram Kapoor, Chirag Wighe et al. Active Disk File System: A Distributed, Scalable File System. Proceedings of the Eighteenth IEEE Symposium, Apr. 2001: 101-115

[41] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar. Coda: A Highly Available File System for a Distributed Workstation Environment. IEEE Transactions on Computers, Apr. 1990: 447-459

# 面向数据存储的分布式文件系统的研究与设计



作者： 张云升
学位授予单位： 武汉理工大学