毕业设计[论文]任务书

姓名 夏 江 班号 0803 专业 计算机科学与技术

同组姓名 （无） 指导教师 王 芳

1. 课题名称

基于FUSE的客户端实现

1. 课题内容

针对Cappella并行文件系统，实现其客户端从内核态到用户态的部分功能移植。

1. 课题任务要求
2. 实现客户端与元数据服务器端及设备服务器端间的初始化工作
3. 实现客户端的挂载工作
4. 实现创建目录命令
5. 上述功能建立在FUSE框架之上

四、参考文献

[1] Guan P, Kuhl M, Li Z, et al. A Survey of Distributed File Systems. University of California, San Diego. 2000:1

[2] Yan J, Zhu Y L, Xiong H, et al. A design of metadata server cluster in large distributed object-based storage. Citeseer, 2004:116~124

[3] Mesnier M, Ganger G R, Riedel E. Object-based storage. Communications Magazine, IEEE. 2003, 41(8): 84~90.

[4] Aditya Rajgarhia. Ashish Gehani.Performance and Extension of User Space File Systems. Stanford University.2000.

[5]Jeff Boniwick and Bill Moore.ZFS The Last Word In File Systems.2005.

[6]Sun Celebrates Successful One-Year Anniversary of Open Solaris. Sun Microsystems. June 20, 2006.

[7] S . A . Weil, S . A . Br an d t, E . L . Miller , D. D. E . L on g,an d C . Maltzah n , C ep h : A scalab le, high -p erformance distributed file system, 7th Symposium on Opeating Systems Design and Implementation (OS DI ), 2000.

[8]Striegel, Jason (6 May 2007). WikipediaFS – a Linux MediaWiki file-system. MAKE. Retrieved 2012-02-10.

[9]Peter J. Braam .The Coda Distributed File System School of Computer Science, Carnegie Mellon University, 2007

[10] A. Westerlund, and J. Danielsson, Arla - a free AFS client, 1998 USENIX Technical Conference, Freenix track, 1998.

[11] Solaris Sun. 多线程编程指南. Mccartney, 2002 年. 4:165

[12] 姚念民，舒继武，郑纬民. SAN 中的分布式锁机制. 计算机研究与发展. 2005, 42(002): 338~343.

[13] 林晓东，刘心松. 文件系统中日志技术的研究. 计算机应用. 1998, 18(001): 28~30.

[14] 熊劲，范志华，马捷，等. DCFS2 的元数据一致性策略.计算机研究与发展. 2005, 42(006): 1019~1027.

[15] 黄雨田，陈庆奎. 多机群网格的数据负载均衡模型. 计算机应用. 2008, 28(1): 52~55.

[16] 覃灵军，冯丹，曾令仿，等. 基于对象存储系统的动态负载均衡算法. 计算机科学. 2006, 33(005): 88~91.

[17]科默，史蒂文. 用TCP\_IP进行网际互联第三卷\_客户——服务器编程与应用. 赵刚，林瑶，蒋慧译. 电子工业出版社. 2008-10.

[18] 刘心松, 邱元杰, 杨锋. Linux 下RPC的分析及改进.小型微型计算机系统. 2001年 07期：801~804

[19] 赵小建, 方康玲. 基于流套接字的RPC技术研究与应用. 武汉科技大学学报(自然科学版) 2007年 01期：68~70

[20]DANIEL P. BOVERT & MARCO CESATI. 深入理解LINUX内核.陈莉君,张琼声,张宏伟译. 中国电力出版社.2006-09：455~517

# 摘要

在分布式文件系统中，基于LINUX内核的客户端面临着开发、调试、维护、移植困难的问题。作为一个较成熟的用户态文件系统框架，FUSE为开发用户态文件系统客户端提供了方便。

通信是分布式文件系统的关键部分。通过分析远程过程协议（RPC），探究内核态RPC与用户态RPC间不同点，将RPC通信技术应用到客户端开发，实现了远程数据本地化的功能。针对用户态文件系统（FUSE）框架，对其原理进行了比较详细的阐述和分析，解决了如何将FUSE框架应用到分布式文件系统客户端中的问题。

在客户端初始化过程中，实现了对输入参数、元数据服务器对象、设备服务器对象、超级块等的初始化。在挂载过程中，实现了客户端和FUSE接口间的初始化，客户端和元数据服务器的通信初始化，以及在客户端和元数据服务器端的挂载。在命令流程中，主要完成了创建目录功能，使得用户可以在客户端用创建目录命令与元数据服务器通信并进行相应处理。最后，对客户端功能做展示，表明其挂载根节点、创建目录功能完好。

**关键词：**用户态文件系统框架，远程过程调用协议，分布式文件系统

# Abstract

In distributed file system, the client based on LINUX kernel faced with the problem of hard developing, debugging, maintaining, and transferring. FUSE, as a mature frame in user space file system, is providing convenience to develop a user space file system.

Communication is a key part of a distributed fie system. Applying the Remote Procedure Call technique (RPC) in client developing through RPC study and analysis, to achieve the goal of making remote data localized. Aimed at applying FUSE frame in client of a distributed file system, some research has been done on FUSE theory.

While in the process of client initialization, the function of initializing input parameter, metadata server object, database server object, and superblock has been fulfilled. While in the process of mount, this course has done initialization between FUSE interface and client, and initialized the communication between metadata server and client. At the same time, this paper has fulfilled the function of mounting on both client and metadata server. While in the process of implementing command, has mainly fulfilled the function of making directory, as to make users to apply the command of creating directory to communicate with metadata server and to do some corresponding work. In the end, the showing results indicating the functions of mounting and making directories work well.

**Keywords:** file system in user space, remote procedure call, a distributed file system

[摘要 I](#_Toc326734294)

[Abstract II](#_Toc326734295)

[1 绪论](#_Toc326734296)

[1.1 背景 1](#_Toc326734297)

[1.2 Cappella体系结构与软件框架 1](#_Toc326734298)

[1.3 RPC协议相关概念 6](#_Toc326734299)

[1.4 FUSE文件系统框架相关概念 6](#_Toc326734300)

[1.5 国内外FUSE客户端研究状况 7](#_Toc326734301)

[1.6 研究意义与目的 8](#_Toc326734302)

[2 核态与用户态RPC异同及其实现方式研究](#_Toc326734303)

[2.1 远程过程模型 11](#_Toc326734304)

[2.2 适应远程程序的多个版本 12](#_Toc326734305)

[2.3 核态RPC实现 13](#_Toc326734306)

[2.4 用户态RPC实现 14](#_Toc326734307)

[2.5 核态与用户态RPC异同 15](#_Toc326734308)

[3 FUSE与VFS交互分析及实现方式选择](#_Toc326734309)

[3.1 FUSE模块框架 16](#_Toc326734310)

[3.2 源代码结构及开发库 17](#_Toc326734311)

[3.3 系统工作流程 18](#_Toc326734312)

[3.4 命令调用流程分析 19](#_Toc326734313)

[3.5 FUSE实现方式的选择 21](#_Toc326734314)

[4 客户端实现](#_Toc326734315)

[4.1 整个客户端的流程框架 22](#_Toc326734316)

[4.2 服务器对象初始化模块 24](#_Toc326734317)

[4.3 LIBFUSE的接口模块 27](#_Toc326734318)

[4.4 命令流程模块模块 29](#_Toc326734319)

[4.5 RPC通信模块 32](#_Toc326734320)

[5 结果展示 34](#_Toc326734321)

[6 全文总结 37](#_Toc326734322)

[7 致谢 38](#_Toc326734323)

[8 参考文献 39](#_Toc326734324)

# 绪论

## 背景

在如今这个信息爆炸的时代，数据的存储具有越来越重要的地位，从而使得企业在存储硬件设备和存储管理软件上的花费在其整个运营成本中的比重也越来越大。当前大量地密集型数据的应用，如数字图书馆、网络播放多媒体系统、气象数据处理、医药视频图像处理等对存储系统的性能、可扩展性和可管理性等方面提出了越来越高的需求。巨大的数据量使得传统的以主机为中心的存储结构无法再满足密集型数据存储应用的要求，为了解决传统存储系统的不足，存储技术逐渐从传统的主机为中心的存储结构向网络存储技术发展。

课题来源于华中科技大学信息存储与应用实验室863项目“海量存储系统关键技术”，该项目研究方向关注于海量存储系统架构、数据的组织、处理和管理机制，着重研究海量网络存储系统结构、并行文件系统、元数据服务器集群技术、元数据的组织与快速检索技术、海量数据的布局与动态负载均衡、数据相关性分析与应用研究、系统一致性研究、存储系统接口和广域网数据系统等。其研究目标一方面支持高效能科学计算中数据的处理、分析和共享协作，另一方面支持面向广域网多种应用的数据共享和交流。

结合当前的863项目课题对系统中的客户端进行重新编写，实现从内核态到用户态的迁移。在性能影响可接受的范围内，使程序员在繁杂的内核态开发中解放出来，实现功能更加完备、稳定的分布式文件系统。

## Cappella体系结构与软件框架

分布式文件系统或网络文件系统是允许多主机通过互联网络共享文件访问的一种文件系统[1]。这类文件系统与本地文件系统的访问语义基本一致，如进行读、写、查找文件等各种操作。文件系统客户端节点不能直接访问存储设备，而是通过网络协议进行访问。依靠设计协议，就可以严格限制服务器和客户端以访问控制列表或者权能的方式遵循文件系统访问控制。



图1.1 Cappella应用拓扑结构图

如图1.1所示是毕设课题来源Cappella系统的拓扑结构图。

cappella海量存储系统是一种基于对象的并行海量存储系统[2]，主要由四部分构成：客户端文件系统（client），元数据服务器[3]（metadata server，MDS），对象存储设备[4]（object storage device，OSD）以及互联网络。客户端为用户提供文件访问服务，符合Posix语义，与Linux虚拟文件系统框架兼容；元数据服务器具有海量元数据组h织及管理能力，具有高可靠性和高可扩展性；对象存储设备支持对象存储接口iSCSI，智能地对文件数据进行存储、布局和管理，具有自治性；互联网络支持多种异构网络协议，对上层网络请求提供协议透明的传输。

### 元数据服务器

元数据服务器（MDS）维护整个文件系统树状结构视图，存储、管理文件系统元数据，协调客户机与OSD之间的交互。其主要功能模块包括

1. 名字空间的组织及元数据管理

其中元数据服务器管理的数据包含文件属性数据、文件布局信息的元数据、文件系统属性、用户信息等。

1. 元数据并发控制

Cappella作为一个支持多客户端并发访问的分布式文件系统，主要保证多线程环境下元数据操作的正确性[5]。使用锁机制实现并发控制，实现资源的读共享及写排他访问。使用轻量级的锁模型保证了元数据服务性能，避免了分布式锁服务[6]的复杂性。

1. 副本机制

为了防止元数据服务器的单点失效和瓶颈效应，必须采用元数据集群向外提供服务，而对于关键元数据，单一服务器存储将可能带来服务器故障时的数据丢失或者服务暂时不可用，对于客户端应用来说，这可能是灾难。因此元数据需要提供备份机制以支持故障恢复。

1. 元数据日志

元数据日志记录每个元数据访问原子操作及其相关信息，系统故障时利用日志[7]将系统恢复到故障前最近的一致性状态，保证系统高可靠性。

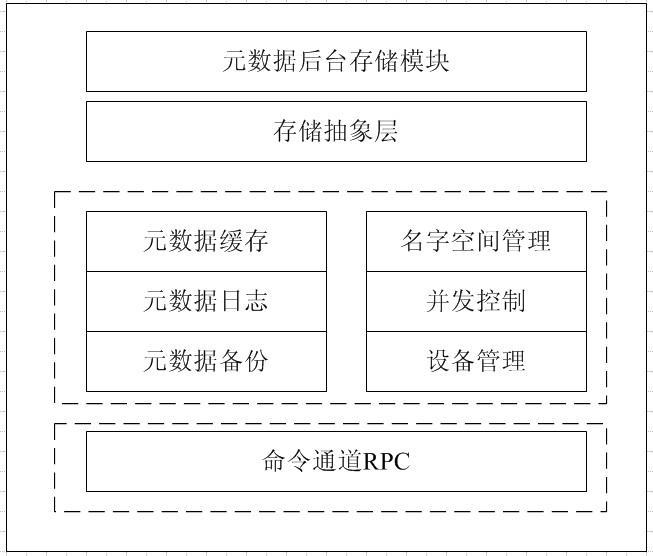


图1.2元数据服务器模块示意图

### 对象存储设备

对象存储设备（OSD）是一种智能的设备，包含磁盘，处理器，RAM内存和网络接口。对象访问接口允许OSD去管理本地对象的存储，并自治地服务和存储来自网络的数据，其模块结构图如所示。OSD是对象存储体系结构的基础，本节重点介绍其主要模块功能。其模块示意图见图1.3所示。

（1）RPC 收发模块：通过封装底层的各种网络协议（TCP，UDP，RDMA等），为上层提供统一简洁的调用接口。

（2）RPC 解析模块：向客户端或者MDS提供实际的处理接口，即客户端或者MDS的远程过程调用的实际执行者。

（3）OSD 标准命令解析模块：解析或封装OSD命令。

（4）OSD 数据缓存模块：缓存客户端经常访问的数据信息，数据一致性保证使用锁机制，。定期将脏数据更新到后端磁盘。

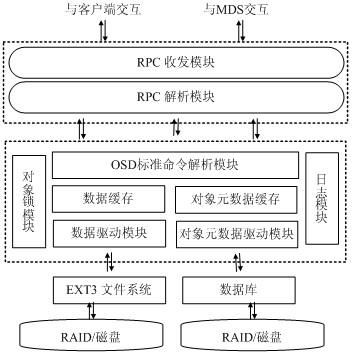


图1.3 OSD模块示意图

（5）OSD 对象元数据缓存：缓存经常访问的对象元数据信息，通过数据库自带的缓存机制完成。目前，系统暂时没有对于对象元数据的应用和访问，OSD只预留部分接口。本文中的对象元数据主要指对象句柄。

（6）对象锁模块：OSD端能并发的对外提供I/O请求，实现了多线程RPC和多线程OSD服务，所以同时会有多个线程访问文件。为了保证多线程访问的一致性，使用了线程锁机制。

（7）日志模块：通过日志来记录客户端或者MDS发来的请求，处理成功之后即从日志删除，否则保留在日志中，这样可以保证在OSD 宕机之后也能保证系统的一致性。

### 客户端文件系统

客户端为用户提供文件访问服务，符合Posix[8]语义，与Linux虚拟文件系统框架兼容，使得分布式文件系统对用户透明，同时又提供高性能的文件访问服务。其模块示意图见图1.4。本节介绍其主要模块功能。



图1.4cappella客户端模块示意图

1. 元数据与数据逻辑抽象

实现虚拟文件系统的各种接口，包括文件操作接口，数据页操作接口，inode对象操作接口，目录项操作接口等。实现诸如创建目录文件操作，删除目录文件操作，重命名操作，查找目录或文件的操作，硬软连接操作，文件系统信息和配额查看操作等逻辑。屏蔽底层远程操作和本地缓存操作的差异，实现抽象无差别的处理控制流。

1. 元数据缓存与数据缓存

为改善文件系统性能与用户体验，实现元数据缓存与数据缓存，完成基于缓存的查找、替换等操作。

1. 一致性控制

为了保证多客户端名字空间视图的一致性[9]，以及元数据能够正确描述数据，客户端提供一致性控制逻辑。使得本地缓存命中时可用；多客户端一致性需要协调时接受MDS请求对本地缓存进行更新或淘汰；元数据更新能及时反映异步的数据更新。

1. 对象数据布局驱动

实现对象布局算法以提高文件系统并行性和数据读写吞吐率。智能选择设备，以避免系统热点和负载不均发生[10, 11]。

### 互连网络

互联网络支持IP，IB，iSCSI协议等主流协议，为了实现协议无关性方便上层应用使用，在主流协议上实现有RPC层通信协议，并加入线程池以更好地利用下层网络的优势。

本课题涉及到RPC协议、FUSE文件系统框架，下面作简要介绍。

## RPC协议相关概念

RPC协议(Remote Procedure Call Protocol)——远程过程调用协议[12]。它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。RPC使得开发包括网络分布式多程序在内的应用程序更加容易。程序可使用这种协议向网络中的另一台计算机上的程序请求服务。由于使用 RPC 的程序不必了解支持通信的网络协议的情况，因此 RPC 提高了程序的互操作性。在RPC 中，发出请求的程序是客户程序，而提供服务的程序是服务器。

RPC协议模型是以我们所熟悉的常规程序的概念模型作为基础设计的（详见2.1.1节）。一般的分布式应用设计有两种方法[13]：一是面向通信的设计，设计报文格式和语法；二是面向应用的设计，设计常规的应用程序来解决问题，将程序分为多个程序块，加入通信协议以允许每个程序块的通信。RPC协议模型是采用了面向应用的设计方法。面向应用的设计方法强调的是所要解决的问题而不是所需要的通信。较面向通信的设计方法而言，避免了程序员过于关注通信协议而错过了应用协议设计的一些重要细节以及容易被忽略的功能需求，使RPC通信对程序员透明。程序员不必去了解通信的细节，可以把精力集中在应用设计上面。

## FUSE文件系统框架相关概念

Fuse[14]是File system in User space，一个用户空间的文件系统框架，允许非特权用户建立功能完备的文件系统，而不需要重新编译内核。fuse模块仅仅提供内核模块的入口，而本身的主要实现代码位于用户空间中。对于读写虚拟文件系统来讲，fuse是个很好的选择。fuse包含一个内核模块和一个用户空间守护进程，将大部分的VFS调用都委托一个专用的守护进程来处理。

Fuse用户空间文件系统与真实的文件系统不同，它的supper block, indoe, dentry等都是由内存虚拟而来，具体在物理磁盘上存储的真实文件结构是什么，它不关心，且对真实数据的请求通过驱动和接口一层层传递到用户空间的用户编写的具体实现程序里来，这样就为用户开发自己的文件系统提供了便利，这也就是所谓的“用户空间文件系统”的基本工作理念。

在用户空间实现文件系统能够大幅提高生产率，简化了为操作系统提供新的文件系统的工作量，特别适用于各种虚拟文件系统和网络文件系统。但是，在用户态实现文件系统必然会引入额外的内核态/用户态切换带来的开销，对性能会产生一定影响。

## 国内外FUSE客户端研究状况

### ZFS

在用户态实现，不受内核权限限制。

ZFS[15]文件系统的英文名称为Zettabyte File System,也叫动态文件系统（Dynamic File System）,是第一个128位文件系统。最初是由Sun公司为Solaris 10操作系统开发的文件系统。作为OpenSolaris开源计划的一部分，ZFS于2005年11月发布，被Sun称为是终极文件系统。ZFS-FUSE[16]是SUN公司通过FUSE实现的一个文件系统。由于ZFS-FUSE是在用户态运行的，所以它不受到内核许可的限制。

### NTFS-3G

基于FUSE实现，在系统间移植方便。

一个增长的趋势是在自己的机器上面安装上微软WINDOW系统和UNIX类的系统。但是由于文件系统的不兼容，不能在两类操作系统上对同一文件进行有效的操作。NTFS-3G[17]是WINDOW NTFS文件系统的一个开源的实现。它提供读写支持。由于NTFS-3G是通过FUSE实现，所以它可以不修改而在很多操作上运行。

### GLUSTERFS

GlusterFS基于可堆叠的用户空间设计。

可堆叠的文件系统，允许人们不断在其中增加新的功能。

使用这样一种描述，一个代码产生器可以为LINUX，FREEBSD及SOOLARIS提供代码模块而不对内核作任何改变。GlusterFS是一款开源的分布式存储系统，它基于fuse设计，具有强大的横向扩展能力，通过扩展能够支持数PB存储容量和处理数千客户端。

### 其他

Ceph[18]: 最初是一项关于存储系统的PHD研究项目，可轻松扩展到数PB容量；支持多种工作负载的高性能；高可靠。

ExpanDrive: 商业文件系统，实现了SFTP/FTP/FTPS协议。

SSHFS: 通过SSH协议访问远程文件系统。

GmailFS: 通过文件系统方式访问Gmail。

EncFS: 加密的虚拟文件系统。

WikipediaFS[19]: 支持通过文件系统接口访问Wikipedia上的文章。

升阳公司的Lustre: 和GlusterFS类似但更早的一个集群文件系统。

HDFS: Hadoop提供的分布式文件系统。HDFS可以通过一系列命令访问，并不一定经过Linux FUSE。

Coda: Coda[20]和传统的NFS不同之处，在于Coda将档案分散在很多服务器上，并维持适当的冗余备份。也因为如此，可以动态的增加容量。和Coda相近的是从集档案系统。

Arla: Arla[21]是由一个内核模块和用户态守护进程组成的文件系统客户端。

和以前的一些努力不同，FUSE在LINUX2.6.14版本以后成了内核的一部分，也可以运用到Mac OS X, Op en Solaris, FreeBSD和NetBSD[22]系统。

目前国内外对FUSE的研究正处于起步阶段 有研究者已经成功的将EXT2等文件系统移植到FUSE框架上也出现了一些利用FUSE结合FTP、HTTP协议实现的新型用户级文件系统。关于FUSE更多的研究工作正在展开。

## 研究意义与目的

### 内核态文件系统与系统紧密耦合，难以进行功能扩展

目前，计算机日常处理的信息量正在以几何级的速度爆炸性增长，这对文件系统的设计带来了挑战。

一方面要求文件系统具有海量存储的能力；另一方面要求它提供强大的功能支持，如数据安全性保护、快速检索、数据共享等。

网络文件系统在这些能力上有很大提高。但是这些文件系统并没有改变运行在系统内核态的特点。传统的文件系统运行在内核态目的是获得更高的运行速度。然而，当数据分散存放后，网络延时成为影响运行速度的首要因素。内核态文件系统与系统紧密耦合难以进行功能扩展的缺点显得越来越突出。

### UNIX上开发核态文件系统是一个挑战性的工作

这种途径需要程序员懂得并熟悉运用核态代码及数据结构，使得新代码倾向产生各种漏洞。并且，对于很多应用程序者，学习核态代码是一个突兀的过程。比如，核态代码缺少内存保护，需要对同步原语小心运用，只能用C写，并且不能用现成的C代码库。调试核态代码也是冗长乏味的，并且错误有可能需要重启系统。

### 核态文件系统挂载必须要有超级权利特点影响了某些功能的应用。

一个核态文件系统只有在拥有了超级用户权利后才能被挂载。这对于在中心管理机器的文件系统开发和使用带来了障碍，比如学校和公司。

比如，WikipeidaFS允许用户查看和编辑维基文件就好像他们是本地文件一样。

### 核态文件系统移植困难

从一个型号的UNIX版本移植文件系统到另外一个版本需要修改文件系统的很多设计和实现，尽管类似的文件系统接口（比如VFS层）使得这个任务容易了些。

与核态开发相比较，在用户态[23]编程减缓甚至消除了上面所说的一些问题。

* 用户态文件系统功能丰富；
* 通过在用户态开发系统，程序员可以很自由的用到各种各样的语言，第三方工具和库，并且不用担心纷繁复杂的核态编程；使用FUSE开发文件系统，开发者不需要与VFS打交道，因此也不需要他了解LINUX文件系统的架构、文件系统内核模块开发的流程以及相关背景知识。开发者只需要关注FUSE的接口定义，甚至连FUSE的内部机制也不必了解。
* 它允许非特权用户在用户态开发文件系统，比如，WikipeidaFS允许用户查看和编辑维基文件就好像他们是本地文件一样；
* 开发一个FUSE文件系统不需要重编译和重新启动内核，因此带来了方便。突破了内核程序的限制，并且隔开了应用程序与系统内核的直接对话，有效地保证了系统的安全性和稳定性。
* 近来FUSE使得其效率也在好起来，使我们的系统真正在应用上跑起来！

### 目的及要解决的问题

整个课题主要要作三方面的工作。

* 实现与FUSE文件系统的接口

在核态文件系统开发中，客户端直接与VFS交互，获取上层用户请求再进行相应的处理。在本课题中，则需要实现与FUSE文件系统的接口，而由FUSE的内核模块与VFS进行交互。

* 实现网络RPC传输交互

在Cappllea系统客户端中，是通过核态RPC与元数据服务器集群和数据服务器集群进行交互。相应的，在本课题中，通过用户态RPC完成通信工作。

* 实现客户端系统初始化和命令流程

一个文件系统的实现，必然涉及挂载和命令的支持。对于本课题，需要实现客户端对目录的挂载；对命令获取并将该命令封装与元数据服务器进行通信。总之，就是要实现一个完整的用户态客户端。

# 核态与用户态RPC异同及其实现方式研究

分布式文件系统必然少不了通信。在本课题中，我们运用以上层协议远程过程模型（RPC）通信。在原来的内核态客户端中是采用的核态RPC实现功能，而实现用户态的客户端则只能利用用户态RPC。因而，实现核态RPC到用户态RPC的迁移需要了解二者的异同及其实现方式。

## 远程过程模型

到目前为止，我们在描述客户和服务器的各个构建的结构。然而，在程序员构建客户和服务器软件时，他们不能一次只关注与其中的一个构建。他们必须要考虑整个系统是如何工作的，以及这两个构建将如何交互。远程过程模型是通过客户端-服务器方式实现的[24]。

为帮助程序员设计和理解客户和服务器的交互，研究人员已为构建分布式程序设计出一套概念性框架。改框架称为远程过程调用模型或RPC模型，它把我们所熟悉的来自常规程序的概念作为设计分布式应用的基础。

远程过程调用模型使用了和常规程序一样的过程抽象，只是它允许一个过程的边界跨越两台计算机，图2.1展示了过程调用模式如何将一个程序划分成两片，每片在一台单独的计算机上执行。当然，常规的过程调用不能由一台计算机传递到另一台上。在程序可以使用远程过程调用之前，必须加入允许程序与远程过程通信的协议软件。

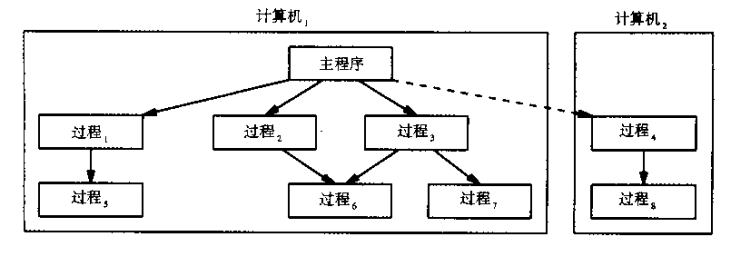


图2.1 分布式程序

通过思考常规的计算机程序，程序员所使用的过程调用的执行模型可以帮助我们理解分布式系统中程序师如何进行的。我们不是考虑客户程序和服务器程序交换消息，而是想象每个服务器实现了一个（远程的）过程，而且，客户和服务器之间的交互对应于过程的调用和返回。由客户发送给服务器的请求对应于对远程过程的调用，而由服务器发回给客户的响应对应于返回指令的执行。图2.2展示了这种比喻

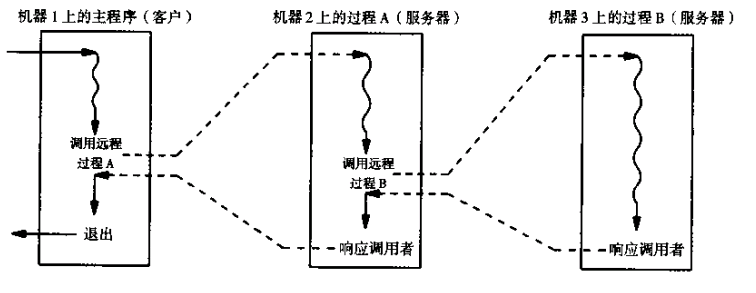


图2.2 程序执行概念模型

ONC RPC通过定义远程执行环境而扩展了远程过程调用模型。它定义了一个远程程序，把它作为在远程机器上执行的软件的基本单元。每个远程程序对应于我们所设想的服务器，它包括一组远程过程以及全局数据。在远程程序中的所有过程都能共享它的全局数据。因此，一组密切合作的远程过程可以共享状态信息。例如，要想实现远程数据库，可以构造远程程序。这个程序包含装在共享信息的数据结构，还有三个维护这些信息的远程过程：insert、delete和lookup。如图2.3所示，远程程序中的所有过程可以共享访问一个数据库。

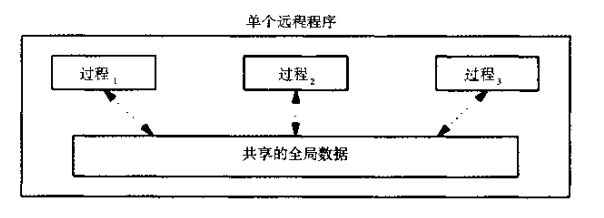


图2.3 一个远程程序中三个远程过程的概念性组织

## 适应远程程序的多个版本

除了程序号之外，ONC RPC还包括针对每个远程程序的整数型版本号[25]。程序的一个版本往往分配个版本号1.以后的每一个版本都收到一个惟一的版本号。

版本号提供了一种能力，它可以使程序不必获得新的程序号就能改变某个远程过程的细节。在实际当中，每个RPC报文通过一个三元组来标识某台给定计算机上的所期待的接收者，这个三元组是：

（prog，vers，proc）

在此，prog标识远程程序，vers指明报文所要发往的程序的版本号，proc指明该远程程序中的某个远程过程。RPC的规约允许计算机同时运行远程程序的多个版本，它就可以运行改进程序过程时进行从容的迁移。这个想法可以概括如下：

因为所有ONC RPC报文要标识远程程序，该程序的版本号以及程序中的某个远程过程，这可能使程序中的一个版本从容地迁移到另一个版本上去，而且可以再过程的旧版本继续运行的情况下测试服务器的新版本。

RPC同时支持底层TCP和UDP传输协议[26]。UDP和TCP传输协议使用16比特的协议端口号来标识通信端点。服务器创建被动的套接字，并将这个套接字绑定到某个熟知协议端口上，等待客户与之联系。为使客户和服务器的这种会合成为可能，我们假设每个服务都被指派了一个唯一的端口号，并且这些指派是熟知的。因此，服务器和客户都认可这个协议端口，服务器在这个端口上接受请求，因为它们都参与了公共的端口分配表。

如果RPC程序没有使用保留的，熟知的协议端口，客户就不能直接与之联系。为了解其中氮肥原因，考虑一下客户和服务器的一些构建。当服务器开始执行时，它要求操作系统分配，一个未使用的协议端口号。服务器使用这个新分配的协议段考进行所有的通信。对服务器每次启动，系统可能选择不同的协议端口。

## 核态RPC实现

LINUX内核的RPC模块实现涉及了大致三个小模块：一是RPC与用户层的接口；二是RPC的逻辑控制框架；三是RPC的通信框架。在这三个小模块里，RPC协议细节贯穿前后，毕竟就是由协议规范来规定具体行为的。这三个模块中除了第二个是逻辑控制必要的之外，另外两个都是可插拔可替换的，逻辑控制模块实际上你自己可以有更好的实现，它无非就是在RPC协议规范下将数据完成XDR二进制编码然后发送到RPC的通信子模块，那么这个通信子模块就是很随意的，只要是可以传输二进制数据的网络协议都可以作为RPC的通信协议，当然用的最多的还是TCP/IP了，表现出来就是INET的socket；至于第一个子模块的可替换就更好理解，NFS就是一个RPC应用，因此NFS可以作为一个用户接口子模块，当然另外一个应用，只要基于RPC的，都可以作为一个RPC用户接口子模块。

struct rpc\_task\_setup task\_setup\_data = {

.rpc\_client = NFS\_CLIENT(inode), /\*属性参数\*/

.rpc\_message = &msg, /\*数据参数，很重要\*/

.callback\_ops = &nfs\_read\_direct\_ops,/\*控制参数，执行过程的回调函数\*/

.workqueue = nfsiod\_workqueue,

.flags = RPC\_TASK\_ASYNC,

};

以上过程中rpc\_task\_setup结构体的作用就是将数据从VFS模块传递到RPC模块，这个结构体封装了很多信息，这些信息都是在执行RPC的过程中要用到的或者初始化任务的时候要用到的。rpc\_execute是一个很重要的函数，它将RPC的执行过程表示为一个状态机，RPC的不同执行过程表示不同的状态，另外rpc\_task结构体封装了一个RPC任务，里面有表示RPC任务当前状态的字段，控制字段，一些维护管理模块要用到的链表，总之在linux中这样的结构很多，比如task\_struct等等，这些结构体均封装了一个实体，也就是一个对象，理解了OO就好理解这些了。另外还有一类结构体是专门为了在不同的模块之间传递参数的，比如上面的rpc\_task\_setup等等，在LINUX内核中，有些这样的结构可以在不同的模块或者层次之间传递，上述的实体结构只在一个模块内被使用，这二类结构体的意义并不同。

在LINUX内核实现的RPC中，异步RPC是靠工作队列来完成的，在老版本中是靠一个内核线程完成的，在新的内核中，工作队列担当了一个很重要的角色，在新内核中工作队列实现了异步IO和异步RPC，不用再像以前那样为每一个特殊内核任务都创建一个独立的内核线程了，统一用工作队列完成，2.6内核就是。具体说来，如果当前的RPC传输任务没有完成，那么直接返回到\_\_rpc\_execute函数，然后判断后返回，但是这个时候RPC还没有完成，具体的完成工作就要工作队列完成了，大致过程和AIO一样，就是不睡眠而是直接返回，待到该任务被wakeup“唤醒”（加上引号是因为根本没有真正睡眠何谈真正唤醒）的时候将任务加入到工作队列中去，工作队列会调度任务的执行的。

在实际传输之前要用XDR规范将数据进行编码，这是RPC的约定。看一下xprt\_transmit(此函数传输实际的数据)就会发现，底层的RPC使用socket将数据传给服务器的，当然也可以用别的机制，比如任何底层链路协议，只要能进行网络传输的就可以，在socket实现的RPC中，socket结构是怎样传递给RPC的xprt\_transmit的呢？还记得上面说的LINUX内核的结构类型吧，实际上rpc\_task内就包含了足够的信息，而rpc\_task在初始化的时候，从vfs层传递而来的数据结构已经将数据参数赋给了rpc\_task了，而这些参数是在open的时候被创建的，这样的话，从一个rpc\_task很容易的得到了需要的数据，比如socket。LINUX内核中的数据结构耦合性彼此都很小，并且数据结构本身大多数也都是小型的，这种特性使得不同模块的数据结构之间的协作相当容易，也正因为如此，一个数据结构才得以在不同的模块穿梭，方便得传递参数。

## 用户态RPC实现

用户态RPC程序的实现可以依靠RPCGEN工具，亦可自己实现。

用RPCGEN工具实现应采取以下八个步骤：

* 构建解决问题的常规应用程序。
* 选择一组过程，以便将这些过程转移到远程机器中，通过这种方法将程序分隔。
* 为远程程序编写RPC规约，包括远程过程的名字及其编号，还有对其参数的声明。选择远程程序号和版本号。
* 运行RPCGEN检查该规约，如果合法，便生成四个源代码文件，这些文件将在客户和服务器程序中使用。
* 为客户端和服务器编写stub接口例程。
* 编译并链接客户程序。它由四个主要文件构成：最初的应用程序（远程过程中被删除了的那个）、客户端的stub、客户端的接口stub以及XDR过程。当所有这些文件都被编译和链接到一起后，最终的可执行程序就是客户。
* 编译并链接服务器程序。它由四个主要文件构成：由最初的应用程序得来的过程，它们现在构成了远程程序；服务器端的stub；服务器端的接口stub以及XDR过程。当所有这些文件被编译和链接到一起后，最终的可执行程序就是服务器。

自己实现则遵循以下规则。程序需要在发出调用请求前线创建一个客户端句柄，或是在侦听请求前先建立一个服务器句柄。程序再在该层可以自由的将自己的应用绑在所有的传输端口上。在函数调用的过程中，需要自己编写编码解码函数，指定需要调用的过程流程等。总之就是要了解整个RPC的通信过程，并运用其提供的功能函数实现通信。

实现方式的选择如下。在此课题中，元数据服务器端的通信代码已经存在，故不能使用代码生成工具来为两方生成代码。而只得自己了解通信流程，通过RPC提供的函数实现通信。也就是选择自己实现方式。

## 核态与用户态RPC异同

RPC协议有多个版本，分为内核态RPC和用户态RPC两类。开发人员根据设计需求选用合适的RPC版本，很多情况下会设计跨内核态和用户态的通信。

用户态RPC不如内核态RPC完善，缺少多线程机制、RDMA（Remote Direct Memory Access）机制等。只有了解了内核态与用户态RPC的异同，才能清晰地了解内核RPC所实现的功能，从而完成内核态RPC到用户态RPC的移植。

# FUSE与VFS交互分析及实现方式选择

过去人们在用户态文件系统做了很多努力。但很多努力都没有对产品系统造成重大影响。然而，近来用户态文件系统复活了。FUSE是一个容许人们在用户态发展文件系统的框架，其便捷使用方式和稳定性使得它很受欢迎。

FUSE是国外一个开源项目这个项目的目的是提供一个完整的接口让使用者可以轻松地创建一个属于他自己的用户级文件系统。目前，很好地实现了与各种Unix操作系统的相结合。自Linux2.6.9后的内核版本都默认地安装FUSE内核模块，正朝着支持多平台多语言的方向发展。

## FUSE模块框架

FUSE分为三大模块。

* FUSE内核模块（内核态）
* LibFUSE模块（用户态）
* 用户程序模块（用户态）

用户程序模块在用户空间实现LibFUSE库封装的文件系统操作。

LibFUS模块实现文件系统主要框架、对“用户实现的文件系统操作代码“的封装、mount管理、通过字符设备/dev/fuse与内核模块通信。

FUSE内核模块实现VFS接口（实现fuse文件驱动模块的注册、fuse 的(虚拟)设备驱动、提供supper block、dentry、inode的维护[27]），接收来至后者的请求,传递给LibFUSE，LibFUSE再传递给我们用户程序的接口进行实现操作。

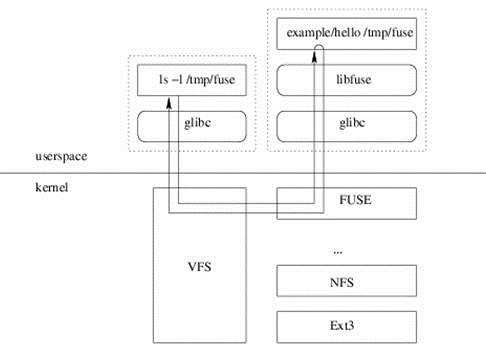


图3.1 FUSE调用路径

在图2.1中显示了FUSE的调用路径。如图，假定目录/tmp/fuse被选为FUSE的挂载点。用户发出文件操作命令，如“ls –l /tmp/fuse”；命令通过用户glibc库传输到VFS层，在VFS调用FUSE内核模块程序。如果发现该请求的数据在页面缓存里，它立即返回。否则，FUSE内核模块，通过一个字符设备/DEV/FUSE将请求命令发送到用户态的LIBFUSE模块；LIBFUSE模块即在此处理程序员在用户态实现的程序，如“example/hello”。实现完毕后，依次沿LIBFUSE、FUSE、VFS返回到用户的命令操作。

## 源代码结构及开发库

FUSE源代码结构及开发库如下表所示，通过这些表格可以了解到用户态文件系统的功能框架。

表3.1 FUSE内核模块

|  |  |
| --- | --- |
| kernel/inode.c | 主要完成fuse文件驱动模块的注册，提供对supper block的维护函数以及其它(驱动的组织开始文件) |
| kernel/dev.c | fuse 的(虚拟)设备驱动 |
| kernel/control.c | 提供对于dentry的维护及其它 |
| kernel/dir.c | 主要提供对于目录inode索引节点的维护 |
| kernel/file.c | 主要提供对于文件inode索引节点的维护 |

表3.2 LibFUSE模块

|  |  |
| --- | --- |
| lib/helper.c | “fuse\_main()”调用的主入口 |
| lib/fuse\_kern\_chan.c | 主要实现fuse应用层访问(读写)fuse driver的功能 |
| lib/mount\_util.c | 提供mount的基础函数调用 |
| lib/mount.c | 主要实现设备的“mount”、“umount”等挂接操作 |
| lib/mount\_bsd.c | “Free bsd”下的“mount”、“umount”实现 |
| lib/fuse\_mt.c | fuse 的mount管理 |
| lib/fuse.c | lib库主框架文件，实现了主要框架及对“用户实现的文件系统操作代码“的封装 |
| lib/fuse\_lowlevel.c | 实现比较底层的函数封装，供fuse.c等使用 |
| lib/fuse\_loop.c | fuse lib循环监视“fuse driver”的通信缓存 |
| lib/fuse\_loop\_mt.c | fuse会话管理 |
| lib/fuse\_session.c | fuse会话管理 |

表3.3 开发库

|  |  |
| --- | --- |
| include/fuse.h | the library interface of FUSE （High Level） |
| include/fuse\_common.h | common |
| include/fuse\_lowlevel.h | Lowlevel API |
| include/fuse\_opt.h | option parsing interface of FUSE |

## 系统工作流程

如下图是FUSE的系统工作流程[28]。fuse\_main为fuse用户空间主函数，用户程序调用它时，该函数解析相关参数（如挂载点,多线程数）,并调用fuse\_mount函数。调用fuse\_new()函数，为FUSE文件系统数据分配存储空间。调用fuse\_loop函数实现会话的接受与处理。

fuse\_mount创建UNIX本地套接口，创建并运行子进程.并返回fuse模块文件fd给fuse\_main函数；fusemount确保fuse模块已经加载，通过UNIX套接口返回fuse模块的文件fd给fuse\_mount()函数；fusermount提供一系列挂载选项，如跳过页缓存，允许root访问挂载的文件系统，允许其他用户访问挂载的文件系统；fuse\_new为fuse创建数据结构空间，用来存储文件系统数据。

fuse\_loop从/dev/fuse读取文件系统调用，调用fuse操作结构中的处理函数，返回调用结果给/dev/fuse.

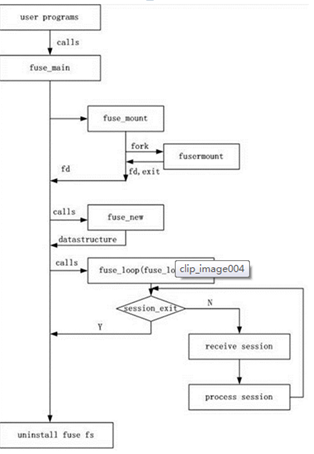


图3.2 FUSE系统工作过程

## 命令调用流程分析

fuse处理请求的整个流程如图三所示，以unlink操作为例进行说明。其中“>”表示调用，”<”表示返回，[]表示调用中所做的工作。

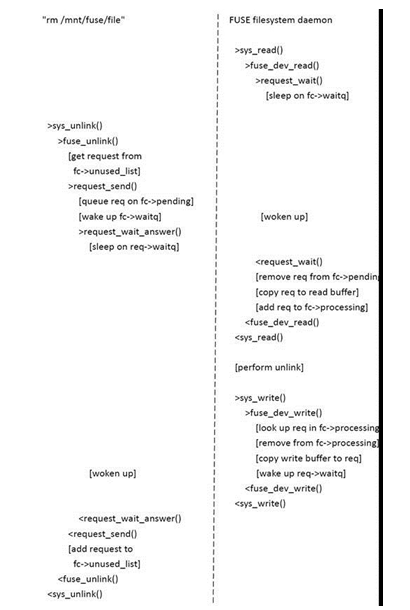


图3.3 FUSE命令调用过程

上图说明了FUSE内核模块与FUSE用户态守护进程之间通信的过程。左方是FUSE内核模块执行”rm /mnt/fusefile”命令时执行的命令流程；右方部分则是FUSE用户态守护进程响应的执行流程。

FUSE文件系统用户守护进程开启后便调用SYSREAD函数一直在/DEV/FUSE字符设备上循环监听命令请求，然后调用REQUEST\_WAIT处于休眠状态以等待FUSE内核模块发出的用户命令的唤醒。

当处理unlink操作时，FUSE内核调用响应的FUSE内核命令fuse\_unlink，然后模块调用request\_send唤醒正在等待的用户LIBFUSE进程，接着进入休眠状态以等待用户守护进程的响应。

FUSE用户守护进程LIBFUSE收到内核发送来的请求，则调用用户实现的unlink接口，处理具体实现的功能。

用户进程处理操作完毕后，需要返回结果给FUSE内核模块。用户守护进程LIBFUSE调用FUSE\_DEV\_WRITE将得到的结果写入到字符设备/DEV/FUSE，这样内核模块就可以从这里取得需要的结果了。此时，FUSE内核模块正好在等待用户守护进程的响应。用户进程发送消息唤醒了正在内核等待的函数。最后FUSE内核模块通过SYS\_UNLINK将结果通过VFS返回给上层用户。

## FUSE实现方式的选择

FUSE发展提供选择两种不同的API以开发用户空间文件系统。低级别的API与 VFS接口紧密联系，高级别用户空间文件系统必须管理inode和执行路径的翻译（和任何相关的翻译）。此外，文件系统必须手动填写重新使用的数据结构以响应内核。此接口是非常有用的文件系统因为它是从无到有的一个创造过程，如ZFS-FUSE，它对这些功能添加到现有的本地文件系统。

另一方面，高层次的API需要的文件只处理与路径，而不是处理Inode相关的东西，因此其系统调用VFS接口类似。 libfuse是负责执行及路径名翻译，填补和发送答复数据结构以恢复内核。高级别API对于大多数用途是足够的，因此使用基于FUSE的文件系统是绝大多数的。

在本课题中，由于涉及实现的功能比较复杂，故采取比较底层的API实现客户端系统。

# 客户端实现

通过对RPC及FUSE的探究，了解到整个客户端的框架。下面探究客户端的具体实现。客户端采取了模块化的设计方法。下面依模块来做设计和探究。

## 整个客户端的流程框架

（1）对输入的各种参数进行初始化，得到符合FUSE框架的输入数据结构框架。动态申请数据结构内存空间，然后利用FUSE的参数解析函数将输入的参数填充到主函数对应的数据结构中，如果出错则退出。在输入的参数中，主要包括：挂载点，读数据的块大小，写数据的块大小，用户的ID等。

（2）对得到的输入参数进行有效的初始化。也就是，对用户未指定初始化的参数进行默认的参数化，或者根据用户输入的参数推导出其他有效的参数。这些参数包括：元数据服务器设备服务器的地址、端口、传输协议和其服务器ID；设备服务器的地址、端口、传输协议和其服务器ID；客户端的元数据服务器端口、设备服务器端口及传输的时延参数；设备服务器的数据分布情况，即数据条带的宽度深度及镜像及RAID情况；还有就是在元数据服务器和设备服务器上的挂载点的名字和长度等参数等。

参看解析参数的数据结构如下：

struct capfs\_parsed\_mount\_data {

int flags;

int rsize, wsize; /\*读写数据块大小\*/

int retrans;

int timeo;

int uid;

unsigned int bsize;

char \*mountpoint; /\*挂载点\*/

int mountpoint\_namlen; /\*挂载点长度\*/

unsigned short clnt\_ds\_port;

unsigned short clnt\_mds\_port;

struct capfs\_layout\_option layout\_option;

struct capfs\_addr server\_addr; /\*元数据服务器地址\*/

};

（3）对超级块进行初始化。主要是对超级块中元数据服务器链表的初始化。默认最多有20个元数据服务器，并用哈希链表将这些元数据服务器对象链接在一起。

参看超级块的数据结构如下：

struct capfs\_sb\_info{

struct mds\_server \*s\_rootmds; /\* 挂载MDS\*/

struct mds\_server \*s\_master\_mds; /\*主MDS\*/

struct mds\_server \*s\_currentmds;

struct list\_head s\_server\_list; /\*服务器链表\*/

struct mds\_hlist mds\_hash\_list;

struct inode\_hlist \*inode\_list;

struct osd\_dev \*s\_dev;

int s\_timeout; /\* OSD操作的时延值 \*/

uint64\_t s\_nextid;

uint32\_t s\_numfiles;

spinlock\_t s\_next\_gen\_lock;

u32 s\_next\_generation;

atomic\_t s\_curr\_pending;

};

（4）对服务器对象进行初始化。通过有效化的初始化参数及初始化了超级块来对服务器进行初始化以得到主元数据服务器句柄。在这个过程中，程序先对元数据服务器进行初始化并将其加入到元数据服务器的链表中；然后针对每个元数据服务器初始化与之对应的相关设备服务器并将其加入到设备服务器队列链表中。具体的服务器初始化情况参看下面的服务器对象初始化模块。

（5）与元数据服务器进行交互得到元数据的根节点。程序首先查看元数据服务器对应的目录路径是否存在。如果不存在则报错，否则，则通过元数据服务器返回的数据来填充客户端的INODE节点。接下来则正式调用GETROOT得到更多与根节点的相关信息并更新客户端INODE节点。

（6）进入FUSE循环以接受用户的命令。与LIBFUSE的字符设备/DEV/FUSE建立联系，并创建与VFS层对应的各个FUSE函数接口；接下来并建立用户态的FUSE会话，在会话中，FUSE不断从字符设备读取上层传来的命令，并调用相应在用户态实现的函数以实现的对应的功能。具体的情况参看LIBFSUE的接口模块。

整个客户端的流程图框架如下图所示：

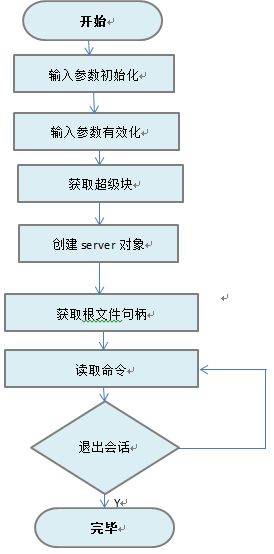


图4.1 客户端的流程框架

## 服务器对象初始化模块

（1）初始化主元数据服务器。申请元数据服务器对象数据结构的内存空间，若申请句柄为空则退出，否则继续；初始化元数据对象数据结构中的客户端元数据服务器的通信句柄，申请通信句柄空间，失败则退出。然后根据对象是元数据服务器或者设备服务器，初始化对应的RPC操作函数，实现句柄的通信操作接口；将初始化的通信句柄添加到元数据对象数据结构中；通过获得的用户输入参数初始化元数据对象数据结构中的一些成员，包括元数据服务器ID、读写块大小；最后将初始化了的主元数据服务器添加到元数据服务器链表中。

元数据服务器对象数据结构如下：

struct mds\_server{

struct capfs\_client \*capfs\_client; /\* 共享客户端 \*/

struct list\_head client\_link; /\*共享一个客户端句柄的MDS链表指针\*/

struct hlist\_node hash\_node; /\*设备哈希节点\*/

int flags;

u16 mdsid;

unsigned long mount\_time; /\* 文件系统挂载时间 \*/

atomic\_t active;

atomic\_t ref\_count;

u32 capfs\_fs\_ltype;

struct capfs\_mount\_type \*capfs\_mountid;

u32 attr\_bitmask[2];

unsigned int dtsize;

unsigned int bsize; /\* 服务器块大小 \*/

size\_t rsize;

size\_t wsize;

unsigned int acregmin;

unsigned int acregmax;

unsigned int acdirmin;

unsigned int acdirmax;

unsigned int namelen;

unsigned int options;

struct mds\_server\* (\*getcurrent)(const struct inode\*inode,const int);

/\*找到当前文件或当前文件父亲所在的服务器\*/

};

（2）初始化与主元数据服务器相关的设备服务器。首先查看设备服务器链表，如果查看到需要初始化的设备服务器已在设备链表中则返回，否则申请设备服务器对象数据结构，通过输入参数获取设备服务器的名字和地址。然后初始化设备对象数据结构中的客户端设备服务器的通信句柄，申请通信句柄空间，失败则退出。然后根据对象是元数据服务器或者设备服务器，初始化对应的RPC操作函数，实现句柄的通信操作接口；将初始化的通信句柄添加到设备对象数据结构中；通过获得的用户输入参数初始化元数据对象数据结构中的一些成员，包括设备服务器ID、读写块大小；初始化设备服务器相关的数据布局情况，并初始化与设备服务器请求相关的队列请求结构；最后将初始化了的设备服务器添加到设备服务器链表中。

（3）循环初始化与主元数据服务器相关的其他设备服务器，初始化过程与（2）类似，并将其加入到元数据服务器对象数据结构所指的设备服务器链表中。

（4）将主元数据服务器句柄添加到超级块对象数据结构的成员指针中，并将其添加到超级块对象数据结构的元数据服务器链表中。

（5）循环初始化其他元数据服务器。初始化过程与（1）类似，并将其加入到超级块对象数据结构中的元数据服务器链表中。如果初始化失败，则依次清除元数据服务器链表、设备服务器链表，并释放与其相应的内存空间。

服务器对象初始化的流程框架如下图4.2所示。

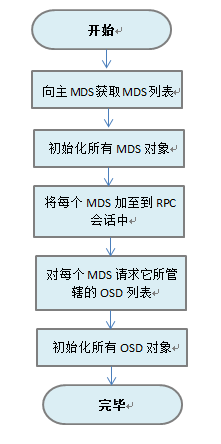


图4.2 服务器对象初始化框架

初始化完服务器对象这后，得到的链表结构如下图4.3所示。

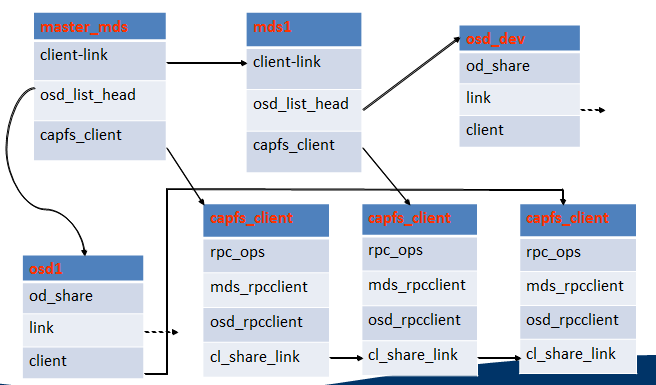


图4.3服务器初始化后的链表结构图

## LIBFUSE的接口模块

LIBFUSE是与FUSE内核模块打交道的模块。从而实现与VFS及用户的间接沟通。他的实现流程如下。

（1）通过客户端的挂载点及输入的其他参数创建FUSE与内核FUSE的通道句柄。通道句柄通过/DEV/FUSE字符设备获取上层传来各种命令，并将得到的结果返回给用户。挂载后，针对挂载点的各种操作便会传输到用户实现的程序里从而实现相应的功能。

（2）通过输入参数及实现的各命令函数命令接口创建FUSE的底层会话句柄[29]。如此，用户的命令，通过VFS，即会调用用户实现的各接口来实现相应的功能。如果创建会话句柄失败，则卸载挂载点并退出，否则继续。

命令接口数据结构如下：

static struct fuse\_lowlevel\_ops capfs\_meta\_oper = {

.init = capfs\_fsinit,

.statfs = capfs\_meta\_statfs,

.lookup = capfs\_meta\_lookup,

.getattr = capfs\_meta\_getattr,

.setattr = capfs\_meta\_setattr,

.unlink = capfs\_meta\_unlink,

.rename = capfs\_meta\_rename,

.mkdir =capfs\_meta\_mkdir,

.opendir = capfs\_meta\_opendir,

.readdir = capfs\_meta\_readdir,

.releasedir = capfs\_meta\_releasedir,

.open = capfs\_meta\_open,

.release = capfs\_meta\_release,

.read = capfs\_meta\_read,

.write = capfs\_meta\_write,

// .access = mfs\_meta\_access

};

在各接口中，一方面程序通过会话句柄、INODE节点、及文件数据结构等信息传入给程序一些必要的参数；另一方面，程序通过FUSE\_REPLY\_ENTRY、FUSE\_REPLY\_ERR[30]等函数将程序处理得到的结果返回给用户上层。而在这些程序中的核心目标就是去填充这些函数要返回给上层的数据结构。

（3）将（1）、（2）中得到的通道句柄和会话句柄联系起来。并针对会话设置退出环境。在运行过程当中，若程序遇到KILL、INT（当用户键<Control-c>时由终端驱动程序发送的信号）、TERM、HUP等信号，程序退出，并删除通道句柄，摧毁会话句柄，卸载挂载点。

（4）以会话句柄作为参数，创建会话循环。根据多线程变量选择进入多线程会话循环或者单线程循环。在循环中，LIBFUSE循环监听/DEV/FUSE字符设备中传来得命令，并进行相应的处理，通过FUSE通道返回结果到上层。

（5）跳出会话循环，移除通信退出句柄，去除通道句柄，释放会话句柄，卸载文件系统。

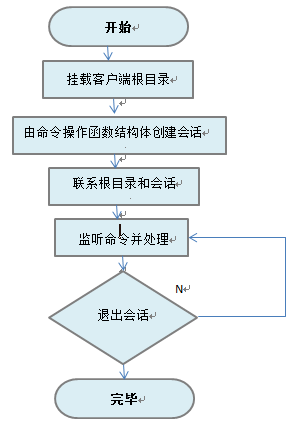


图4.4 LIBFUSE接口模块

## 命令流程模块模块

### 获取根节点命令流程

（1）通过超级块获取主元数据服务器句柄，通过输入参数获取根目录的文件句柄（包括名字、名字长度、及其父目录的全局ID）；由以上获得的两个数据结构作为参数，与元数据服务器进行通信，查看在元数据服务器端是否存在文件句柄所对应的目录路径是否存在。若不存在即返回，存在则返回对应的目录的各属性及查看状态。

（2）通过查看目录路径返回的属性获取根目录的全局ID，将全局ID转化为客户端的INODE序号，由INODE序号以查看相应的INODE节点在哈希链表中是否存在。若存在即返回，若不存在则由返回的属性填写相应的INODE节点数据结构，包括创建时间、修改时间、读取执行时间、引用次数、大小、用户ID、用户组ID及INODE节点序号，并根据节点是目录或者普通文件填写相应INODE操作函数，最后根据INODE序号将将INODE加入到INODE哈希链表中以方便以后的INODE查找。

INODE数据结构如下：

struct capfs\_i\_info {

unsigned long i\_flags;

struct inode vfs\_inode{

struct hlist\_node i\_hash;/\*节点的哈希指针\*/

struct capfs\_sb\_info \*i\_sb;

pthread\_mutex\_t i\_lock;

unsigned long i\_ino;

unsigned int i\_nlink;

uid\_t i\_uid;

gid\_t i\_gid;

loff\_t i\_size;

struct timespec i\_atime;

struct timespec i\_mtime;

struct timespec i\_ctime;

umode\_t i\_mode;

const struct inode\_operations \*i\_op;

const struct file\_operations \*i\_fop;

unsigned long i\_state;

};

/\* cappella 扩展部分 \*/

struct global\_id\_t i\_cap\_id; /\*全局文件ID\*/

struct capfs\_fh i\_fh; /\*标识文件的句柄\*/

unsigned int i\_attr\_validity;

u64 i\_child; /\*目录下的文件数目\*/

unsigned long flags;

unsigned long cache\_validity;

};

（3）由（2）中得到INODE查找对应节点所在的元数据服务器的句柄，并以此句柄初始化超级块的根元数据服务器对象指针。查找思想如下：由主元数据服务器会返回根目录所在的全局序号；而全局序号由元数据服务器序号、区号和本地序号组成。依据返回的元数据服务器序号在客户端做哈希链表查找即可得到根目录所在的元数据服务器。

（4）向（3）中得到的元数据服务器发送GETROOT命令，企图更新与根节点相关的一些INODE信息。填写RPC发送结构体的相关数据结构并与元数据服务器进行通信。如果通信成功即将返回的结果拷贝到客户端的数据结构中，以更新一些INODE信息。不成功则根据返回的错误状态作相应的处理。

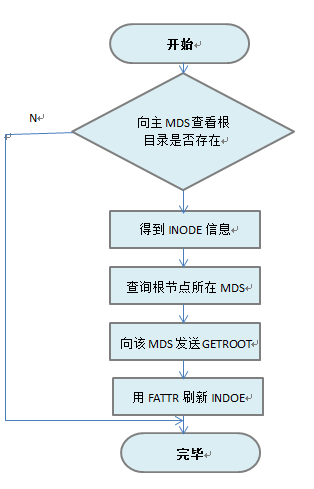


图4.5 获取根节点流程

### 创建目录命令流程

（1）由FUSE请求句柄得到进程上下文环境，此环境包括用户ID、用户组ID、进程ID；由FUSE提供的将要新建的目录的父亲目录的INODE序号转化得到其全局ID（INODE序号与其全局ID是可以相互转化的：全局ID通过移位复合即得到INODE序号、INODE的序号通过分解再移位便得到全局ID的各部分），联合得到的父目录全局ID和上层传输而来的目录名字则组成了要创建的目录的文件句柄；

（2）由父母录的INODE序号通过哈希查找得到父目录INODE，再有父目录INODE找到父目录所在的元数据服务器（查找思路在获取根节点的步骤3中有叙述）。

（3）由（1）和（2）中处理的得到的各数据组成与元数据服务器进行RPC通信的参数数据结构，进行RPC通信，并调用元数据服务器端的MKDIR命令。若通信成功，则将所创建目录的各属性和通信状态返回并填充响应的返回数据结构，否则直接返回并根据返回的错误码进行相应的处理。

（4）通过（3）中返回的属性获取所创建的新目录的全局ID，将全局ID转化为客户端的INODE序号，由INODE序号以查看相应的INODE节点在哈希链表中是否存在。若存在即返回，若不存在则由返回的属性填写相应的INODE节点数据结构，包括创建时间、修改时间、读取执行时间、引用次数、大小、用户ID、用户组ID及INODE节点序号，并根据节点是目录或者普通文件填写相应INODE操作函数，最后根据INODE序号将将INODE加入到INODE哈希链表中以方便以后的INODE查找。更新与本文件系统相关的特许INODE信息，包括文件的全局ID、文件句柄等。

（5）将（4）中得到的INODE信息转化为符合FUSE的数据结构，并通过FUSE\_REPLY\_ENTRY函数将结果返回给上层。

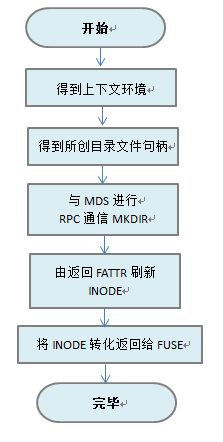


图4.6 创建目录模块

## RPC通信模块

（1）RPC规约。从本质上说，RPC规约文件包含了对远程的声明以及它所使用的数据结构。该规约文件包含常量、类型定义记忆对客户和服务器程序的声明。更明确地说就是，这个规约文件包含：

声明在客户或服务器中所使用的常量。

声明所使用的数据类型。

声明远程程序、每个程序中所包含的过程以及它们的参数的类型。

RPC使用一些数字来标识远程程序以及在这些程序中的远程过程。在规约文件中的程序声明定义了诸如程序的RPC号、版本号以及分配给程序中的过程的编号等细节。一个RPC规约文件并没有包括在最初的程序中所能找到的所有声明。它仅仅定义了哪些在客户和服务器之间要共享的常量和数据类型，或者是哪些需要指明的参数。

（2）RPC对数据类型的XDR转换。RPC对远程程序中例程进行调用，这些例程必须执行XDR转换，并且此XDR转换是针对远程程序中所声明的所有数据类型的。在客户端程序对进行远程调用的参数进行XDR编码，使之转换成能够被网络识别的XDR序列；在服务器端，远程程序接收客户端发送过来的XDR序列，并运用相应的XDR解码函数对其进行解码。同样的，服务端返回的结果首先通过XDR序列转换成网络能识别数据，进行网络传输传送回到客户端，其后，客户端通过相应的XDR转码函数对返回的结果进行解码，得到符合C的数据结构。由此，在一个RPC调用的过程中进行了两次的编码及解码。

（3）客户端远程调用的RPC框架。以元数据服务器地址、远程程序的程序号及版本号作为参数通过CLNT\_CREATE创建通信的RPC句柄。以通信RPC句柄、远程程序的过程号、客户端的编码函数、远程程序的参数、客户端的解码函数、远程程序返回的结果以及超时参数作为输入参数，通过CLNT\_CALL调用相应的远程过程；若返回RPC\_SUCCESS即调用成功，否则调用失败返回。

（4）服务器端接收远程调用的RPC框架。以RPC的通信端口为参数，根据选择底层传输为TCP或者UDP传输，选择不同的函数创建服务端的通信句柄。以服务器的通信句柄、远程程序的程序号、版本号、映射函数及是否选择端口映射参数为输入参数，通过SVC\_REGISTER注册相关程序，从而远程程序的调用与上述的映射函数会取得联系。通过SVC\_RUN启动服务端的程序RPC服务，在这个过程中，服务器循环在映射端口监听从远程发来的调用命令，并传入映射程序进行合理的编码、解码以作相应的处理，最后将处理的结果返回给远程程序。

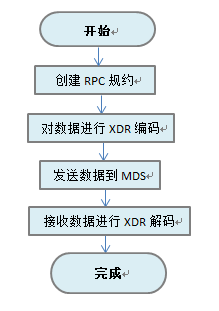


图4.7 RPC通信模块

# 结果展示

功能展示的过程如下所示。

1.运行元数据服务器的代码，启动元数据服务器服务。

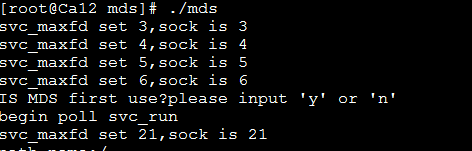


图5.1 启动元数据服务器服务

在图5.1中，默认元数据服务器是第一次启动。从图中可以看到服务器进入了SVC\_RUN服务循环，以监听客户端的RPC请求。

2.加载FUSE内核模块，然后运行客户端代码。

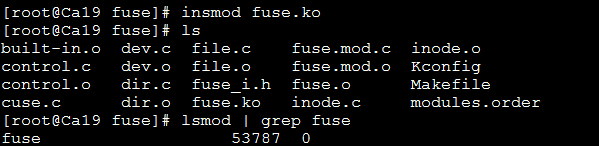


图5.2 加载FUSE内核模块

在上图中可以看到FUSE内核模块加载成功。随后运行客户端代码如下。

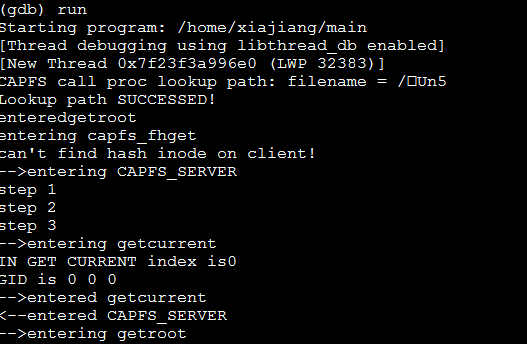


图5.3 客户端运行代码

在图5.2中，客户端启动成功，并可以看到客户端的很多打印信息。从打印信息最后一行可以看出程序进入了获取根节点流程。

3.获取根节点信息

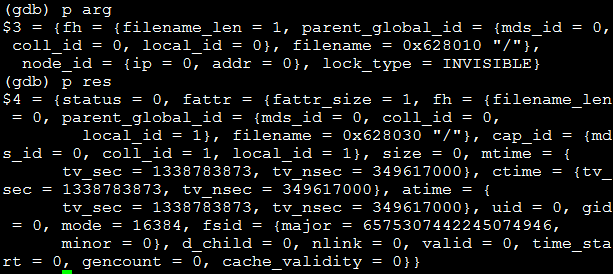


图5.4 客户端发送给服务器的参数和返回结果

在图5.3中，ARG是传输给服务器的参数，RES则是返回的属性结果。从上可以看到获取根节点成功。

4.挂载并获取根节点的全局ID

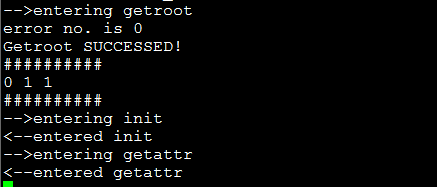


图5.5 获取根节点全局ID

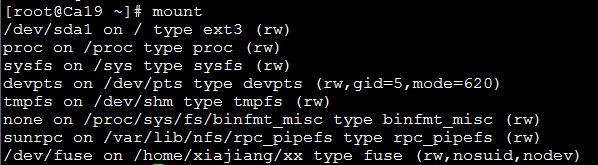


图5.6 mount命令查看结果

在图5.5中可以获取根节点成功，并且获取全局ID为0、1、1。从图5.6用mount命令从其最后一行可以看出字符设备/DEV/FUSE在挂载点/HOME/XIAJIANG/XX上挂载成功。

5．针对挂载点创建目录



图5.7 针对挂载点创建目录dd

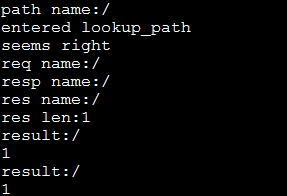


图5.8 元数据服务器端的打印信息

# 全文总结

总体来说，本课题完成了cappella系统客户端从内核态到用户态的部分功能迁移。

首先完成了FUSE框架的搭建。原来的客户端直接与VFS层接口，用户的命令直接从VFS传输到客户端系统代码，并作相应的处理。完成FUSE框架后，客户端与FUSE接口，而由FUSE去完成与VFS的接口。

然后完成了用户态RPC的通信。原来的客户端运用内核态RPC与元数据服务器和设备服务器通信，涉及到多线程和队列管理，非常繁琐。改为用户态RPC，设计相应的编码，解码函数，从而完成相同的通信功能。

其次完成了挂载根目录的功能。这里涉及到两次挂载：客户端根目录挂载及元数据服务器根目录挂载。挂载目录后，建立了挂载点和客户端代码的联系，也建立了挂载点和元数据服务器代码的联系。即针对挂载点的操作会传入到相应的程序进行操作。

最后还完成了创建目录的功能。体现了命令从用户输入，到VFS，再传到内核FUSE，再到LIBFUSE，到客户端代码，再经网络传输到元数据服务器端具体创建目录；然后沿路返回到用户的过程。

# 致谢

值在论文截稿之际，谨向在毕设期间悉心指导过我的老师、学长、学姐，所有关心我的朋友，和帮助过我的人们致以崇高的敬意和深深的感谢！

衷心感谢我的导师王芳教授。在我的研究学习期间，王老师给予了我很多关心和帮助。在做毕设的进程报告时，她为我的课题进行了多次分析和讨论，总是能够指出关键性的问题，并提出可行的解决方案或者思路。王老师对学习、对科研、对生活的很多经验都值得我们学习。她总是孜孜不倦地教导我们，用自己开阔的思维和渊博的知识为我们解开前进道路上一个个迷。

感谢孙海峰学长，他时刻关心着我的研究和工作的进度和困难，帮助我解决了许多课题中遇到的困难。

感谢师兄吴森、肖飞、周俊、杜鑫，感谢学姐苏楠、陈碧砚、聂凤，在学习和研究方面，他们毫无保留地向我传授经验，让我少走了很多歪路。

感谢朱挺炜、祖文强、谢燕文、田昊、刘芬、刘玉、赵玉京，与你们在一起学习、合作和交流，丰富了我的知识，开拓了我的思维，在技术上也给我提供了许多支持，同样也带给了我很多快乐。

感谢我的家人，他们永远是我坚强的后盾。父母给了我生命，养育我长大。他们是我的第一任老师，教会我做人的道理，而且一直鼓励我要专心致志，学有所称。无论天涯海角，总有他们的牵挂陪伴着我，祝福着我。

再次衷心感谢所有关心、支持和帮助过我的人们，谨以此文献给他们！

# 参考文献

[1] Guan P, Kuhl M, Li Z, et al. A Survey of Distributed File Systems. University of California, San Diego. 2000:1

[2]吕文若 分布式文件系统客户端元数据访问行为优化研究:[硕士学业论文]. 保存地点：华中科技大学

[3] Yan J, Zhu Y L, Xiong H, et al. A design of metadata server cluster in large distributed object-based storage. Citeseer, 2004:116~124

[4] Mesnier M, Ganger G R, Riedel E. Object-based storage. Communications Magazine, IEEE. 2003, 41(8): 84~90.

[5] Solaris Sun. 多线程编程指南. Mccartney, 2002 年. 4:165

[6] 姚念民，舒继武，郑纬民. SAN 中的分布式锁机制. 计算机研究与发展. 2005, 42(002): 338~343.

[7] 林晓东，刘心松. 文件系统中日志技术的研究. 计算机应用. 1998, 18(001): 28~30.

[8] Gallmeister B O. POSIX. 4. O'Reilly & Associates, 1995:2~28

[9] 熊劲，范志华，马捷，等. DCFS2 的元数据一致性策略.计算机研究与发展. 2005, 42(006): 1019~1027.

[10] 黄雨田，陈庆奎. 多机群网格的数据负载均衡模型. 计算机应用. 2008, 28(1): 52~55.

[11] 覃灵军，冯丹，曾令仿，等. 基于对象存储系统的动态负载均衡算法. 计算机科学. 2006, 33(005): 88~91.

[12]周俊 RPC协议分析及其优化研究:[本科学业论文].保存地点：华中科技大学

[13]科默，史蒂文. 用TCP\_IP进行网际互联第三卷\_客户——服务器编程与应用. 赵刚，林瑶，蒋慧译. 电子工业出版社. 2008-10.

[14]Aditya Rajgarhia. Ashish Gehani. Performance and Extension of User Space File Systems. Stanford University.2000.

[15]Jeff Boniwick and Bill Moore.ZFS The Last Word In File Systems.2005.

[16]Sun Celebrates Successful One-Year Anniversary of Open Solaris. Sun Microsystems. June 20, 2006.

[17]梁金千，张跃. NTFS文件系统的主要数据结构.计算机工程与应用.2004,39(8):42~46

[18] S.A. Weil, S.A. Br an d t, E.L. Miller, D. D. E. L on g, and C. Maltzahn, Ceph : A scalable, high-performance distributed file system, 7th Symposium on Opeating Systems Design and Implementation (OS DI ), 2000.

[19]Striegel, Jason (6 May 2007). WikipediaFS – a Linux MediaWiki file-system. MAKE. Retrieved 2012-02-10.

[20]Peter J. Braam .The Coda Distributed File System School of Computer Science, Carnegie Mellon University,2007

[21] A. Westerlund, and J. Danielsson , Arla - a free AFS client, 1998 U S E N I X Technical Conference, Freenix track, 1998.

[22] A . Kantee, ReFUSE : User space FUSE reimplementation using puffs , EuroBSDCon , 2007.

[23] R . P. S pillane, C . P. Wr ight, G. S ivathanu , an d E .Z ad ok, Rap id file system development using ptrace, ACM Workshop on Experimental Computer Science(E X P C S ), 2007.

[24] Andrew D. Birrell, Bruce Jay Nelso. Implementing Remote Procedure Calls, ACM TOCS Vo.2 ,February 1984:39~59

[25] 刘心松, 邱元杰, 杨锋. Linux 下RPC的分析及改进.小型微型计算机系统. 2001年 07期：801~804

[26] 赵小建, 方康玲. 基于流套接字的RPC技术研究与应用. 武汉科技大学学报(自然科学版) 2007年 01期：68~70

[27]DANIEL P. BOVERT & MARCO CESATI. 深入理解LINUX内核.陈莉君,张琼声,张宏伟译. 中国电力出版社.2006-09：455~517

[28]URL:http://my.debugman.net/program/fuse-180.html

[29]http://www.cnblogs.com/JeffreySun/archive/2011/04/03/2004259.html

[30]http://fuse.sourceforge.net/doxygen/annotated.html