封面

内封

摘要

Abstract

目录

前言

自从世界上第一台全自动电子计算机“埃尼阿克”诞生以来，计算机技术的发展极其迅速。在移动互联网及大数据的背景下，计算机已经融入到人类生活的方方面面。从衣食住行到工业生产，计算机已经成为人类社会不可或缺的重要组成部分。随着计算机科学与技术的快速发展而来的各种规模的软件工程，常常需要整合人力以及各种软件开发工具，这时版本控制的概念应运而生。

版本控制作为软件工程的一种技巧，需要让项目中的多个开发人员的开发进度保持同步。在软件开发过程中，参与项目的人数多不代表一定可以加快项目的开发速度，因为软件产品的抽象性，人员之间的沟通与管理极有可能占用很多资源，导致整体开发效率下降，而版本控制系统的作用就是协调开发人员的工作，同步不同开发者的进度，尽可能减少版本管理占用的资源。

现代软件开发已经不是“软件=数据结构+算法”的模式，而是要适应各种变化的需求，使软件架构有足够的灵活性，不至于因为需求的变化而推倒重来。现代软件源于互联网的发展，互联网使得软件进入新的时代。互联网深入生产生活的方方面面，因此需要处理一些难以用算法表达的业务逻辑，如银行的金融业务不仅很复杂，很难用算法表示，而且经常调整，导致需求多变。这就需要版本控制适应变化的需求，可以进行同一系统下的分支管理。

一个算法可以从源代码中识别出来，而业务逻辑则很难从代码中看出来。一些企业级软件即使留下源代码，后来者也很难明白其中的业务逻辑。随着老一代程序员的退休，他们也将业务逻辑带走了。以至于后来者不敢轻易重构遗留代码，怕一个误解造成巨大损失。为了解决这样的问题，版本控制系统需要记录下迭代过程中的版本描述，以简化维护人员分析项目的过程。

源代码是开发人员的产品，因此对版本控制系统的安全性及稳定性有较高的要求。软件及硬件上的错误是无法预料的，同时也很难避免，因此对于版本控制系统的文件系统这方面，分布式架构是一个非常好的选择。分布式系统具有较高容错性，将文件操作等较慢的操作分发给多个服务器也有助于提高文件读写效率，采用RAID技术的底层文件系统对文件安全也有很大保障。

1 需求分析

1.1 需求分析引言

1.1.1 项目背景

本系统的名称为基于ASP.NET的源代码版本控制系统。版本控制是一种记录一个或若干文件内容变化，以便将来查阅特定版本修订情况的系统。随着软件系统规模的日益扩大和复杂程度的日益增长，软件工程师或网页设计师可能会需要保存某一系统的源码或页面布局文件的所有修订版本，采用版本控制系统是个明智的选择。有了版本控制系统就可以将某个文件回溯到之前的状态，甚至将整个项目都回退到过去某个时间点的状态。用户可以通过系统比较文件的变化细节，查出最后是团队中的哪个成员修改了哪个地方，从而找出导致问题出现的原因，是谁在何时报告了某个功能缺陷等等。使用版本控制系统通常还意味着，就算对整个项目中的文件进行大幅度改动，也照样可以轻松恢复到原先稳定版本的样子，但额外增加的工作量却微乎其微。

版本管理是软件配置管理的基础，它管理并保护开发者的软件资源。本系统作为软件开发过程中的辅助工具，目的在于减少版本控制及管理过程中使用的人力资源，协调进行不同工作的开发人员的工作，同步不同开发者的进度。系统需要尽可能保存每一阶段的工作成果，尤其是源文件，以保证每个阶段性工作成果的安全，这样任何时候都可以方便的找回原来的工作成果；另一方面系统应能够快速检索工作成果，比如很容易找到某个版本的文件，或者最主要的几个阶段性成果，并且能够很容易预览需要的文件。

很多编程人员有乐观主义，总是相信自己的代码是肯定能运行的。所以在安排项目的进度的时候就会是假设在代码能够在正常运行时应该花费的时间。而现实往往不是乐观，在项目的进展过程中会存在各种不可预知的问题。在这个时候项目经理就会为项目增加人员，然而增加人员反而导致项目进度越来越慢。因为新增加的人员还需要培训，需要时间去了解项目的内容和进展情况。在投入了更多的人力的时候，经理发现项目进度反而更慢他就会投入更多的人力，这种恶行循环导致项目的失败。

1.1.2 项目风险

由于开发能力有限以及时间安排上的任务冲突，可能无法实现开题报告所述的全部功能，但可以保证实现基本的版本控制功能。对于使用者，风险主要来自于程序编写过程产生的漏洞，以及系统的初期版本可能不稳定。此外，因为系统的重点在于版本控制，分布式文件系统可能仅有部分功能可以启用。初期版本系统的部分功能在质量及性能上可能无法达到本需求文档的要求，但后续版本可能会有提升。

1.1.3 本系统的目标用户

本系统的目标用户分为普通用户及系统管理员。普通用户是本系统的最终用户，一般为软件开发人员或软件项目管理人员，这一类用户应具有版本控制的基本知识，可以理解计算机的基本操作及运行原理。使用者应尽可能保存软件开发过程中的各个阶段，以及各版本对应的描述性说明及详细开发文档。对于每一个使用本系统的软件项目，本系统的使用频度应对应于项目的开发速度，可以每次修改源代码就使用本系统，也可以在某开发进程达到某阶段后使用本系统。

系统管理员对本系统进行管理，其具有直接操作后台数据库以及配置本系统的权限（如备份、回滚数据库等），并在系统出现问题时对普通用户进行回应。

1.2 功能需求

1.2.1 系统范围

代码仓库：具有受版本控制的所有文件的完整修订历史的共享数据库。

分支：分支是指目录和文件的现有原始树的副本。分支的生命周期是从某事物的副本开始的，并从此副本处移动，生成自己的历史。通常创建分支以尝试新功能，同时不影响具有编译器错误和小问题的开发的主分支。

普通用户可以创建软件项目对应的代码仓库；代码仓库有且只有一个主分支，该分支无法被删除且至少有一个版本；用户可以在代码仓库下创建分支，分支的根可以不是主分支；用户可以查看选中的代码仓库，也可以查看选中的分支及版本；用户可以向代码库中选中的版本上传文件、下载文件或创建文件夹；系统应记录每一次版本迭代的信息，如描述和时间等;用户可以在代码仓库下提交评论和文件，文件经过仓库拥有者审核后可以合并到对应版本中。

代码仓库、分支及版本应该具有项目描述和创建时间等描述性信息。仓库拥有者可以创建、删除分支，可以将当前分支回滚到之前的某一个版本。仓库拥有者也可以删除指定仓库。

普通用户可以创建账号；系统应对已登录用户和未登录用户的权限进行验证。具有权限的用户可以预览某一版本的文件结构和文件内容，并可以签入或签出选中的文件。如果时间允许，系统应实现不同版本文件之间的差异对比。

分布式文件系统应具有文件操作的基本功能，并且分布式文件服务器应是可拓展的。

1.2.2 系统体系结构



系统功能结构图

版本控制系统下分为九个模块：

用户管理：可以进行注册、登录、记录用户信息等操作。

代码仓库管理：用户登录成功后便可以在自己的工作空间中查看已创建代码仓库的列表，同时也可以在这里创建和删除代码仓库。

分支管理：创建和删除分支，也可以添加描述信息。

版本控制：签入或签出文件，回滚分支到指定版本，推出新版本。

文件系统：由分布式服务器构成的文件系统，独立于web服务器。Web服务器与分布式文件系统之间采用自定义通许协议。

数据库封装：将页面与数据分离，把对数据库的操作封装成库。

权限检查：对于某些操作进行用户身份检查，来确定是否有操作权限。

差异对比：对比不同版本文件的差异。

1.2.3 系统总体流程



系统整体流程图

用户使用系统时需要先注册，注册成功后后台会记录用户的账号、密码、用户名、用户描述和注册时间等信息。用户可以使用注册时使用的账号登录系统，如果登录失败则返回登录界面重新登录，如果登录成功系统会重定向至用户工作空间，在该界面用户可以查看和管理已拥有的代码仓库。

如果用户想查看或管理某个代码仓库，可通过工作空间中对应的链接进入代码仓库详细信息界面，用户可以在这里查看代码仓库的描述信息，如代码仓库创建时间、说明等信息。用户可以在仓库详细信息界面进行版本控制，如管理分支和文件。分布式文件系统独立于web服务，web服务可通过特定的通讯协议对文件系统进行控制，如进行文件读写等操作。

1.2.4 用户界面概述

系统至少要具有如下界面：

主页：访问系统时打开的第一个界面。主页中有系统的介绍，用户可以通过主页跳转到注册和登录界面。

注册页面：用户在注册页面输入用户名、账户和密码等信息，通过后台审核后即可获得账户和工作空间。

登录页面：用户可在登录页面使用注册时提交的账号登录。登录成功后自动跳转至该用户工作空间界面。

用户工作空间页面：在这个页面可以查看用户名、用户描述信息，也可以看到该用户已创建的代码仓库。用户可以通过该页面跳转到代码仓库浏览页面。工作空间的拥有者可以在该页面创建新的代码仓库。

创建代码仓库页面：在该页面，工作空间拥有者可创建新的代码仓库。用户输入仓库名称和描述信息后可创建代码仓库，系统会自动初始化代码仓库的主分支和第一个版本。每个代码仓库有且仅有一个名为master的主分支，且主分支无法被删除。

代码仓库浏览界面：这个页面包含了本系统的核心功能。代码仓库的拥有者可以在这个页面进行版本控制，包括分支管理、版本管理和基本的文件管理。在该页面代码仓库的拥有者可以跳转到创建分支界面和推出新版本界面。该页面也可以进行基本的文件访问操作。

创建分支界面：仓库拥有者可以在该页面向代码仓库提交新的分支，新分支的第一个版本的文件结构由选中的起始版本决定。用户应输入分支名和描述信息。用户不能创建名为master分支，且每个分支至少会包含一个版本。

推出新版本界面：用户在该界面输入新版本名称和描述，服务器审核成功后就可推动选中分支发展。

评论查看界面：代码仓库的拥有者可以在这里查看其他用户对该仓库提交的评论。

用户界面的总体色调以蓝色和白色为主，文件系统在初期版本可以使用控制台界面。界面为级简风格，尽量采用风格相近的颜色搭配，尽量减少颜色种类以突出网页重点内容。

1.2.5 需求分析

软件需求分析是启动一个软件工程的重要工作阶段，本小节将概述性的说明本系统中用户、需求、系统功能单元之间的联系。

系统整体用例图

游客、普通用户和管理员用例图

|  |  |
| --- | --- |
| 用例说明 | 详细信息 |
| 用例名称 | 游客注册账户 |
| 用例标识号 | 1 |
| 简要说明 | 为游客创建新账户。数据库应记录用户名、登录账户、密码、用户描述和账户创建时间。 |
| 前置条件 | 用户已打开注册界面。 |
| 基本事件流 | 1.用户输入用户名。  2.用户输入用于登录的账户。  3.用户输入登录密码。  4.用户输入用户描述。  5.用户单击创建按钮。  6.服务器校验用户输入。如果输入符合规则，系统自动生成新用户的ID并记录账户创建时间；否则要求用户重新输入注册信息。 |
| 其他事件流 | 如果用户关闭注册页面，注册程序终止。 |
| 异常事件流 | 1.用户输入不符合规则或输入为空，要求用户重新输入注册信息。  2.用户输入的账号已存在，要求重新输入数据。 |
| 后置条件 | 1.数据库记录新用户信息。  2.页面跳转到登录界面。 |
| 注释 | 可以有同名用户，但用户ID是唯一的。 |

|  |  |
| --- | --- |
| 用例说明 | 详细信息 |
| 用例名称 | 用户登录 |
| 用例标识号 | 2 |
| 简要说明 | 让拥有帐号的用户登录系统。 |
| 前置条件 | 用户已打开登录界面。 |
| 基本事件流 | 1.用户输入账户及密码。  2.用户单击登录按钮。  3.服务器校验登录信息。如果账户密码不匹配，要求重新输入登录信息。 |
| 其他事件流 | 1.如果关闭页面，登录程序终止。  2.如果用户无账户，可跳转到登录界面。 |
| 异常事件流 | 1.如果输入不符合规则，要求重新输入。  2.如果密码错误，要求重新输入密码。 |
| 后置条件 | 1.用户登录系统。  2.页面跳转到用户工作空间。 |
| 注释 | - |

|  |  |
| --- | --- |
| 用例说明 | 详细信息 |
| 用例名称 | 系统管理员帐户管理 |
| 用例标识号 | 3 |
| 简要说明 | 系统管理员帐户管理 |
| 前置条件 | 1.系统管理员成功登录数据库。  2.系统管理员具备访问用户表的权限。 |
| 基本事件流 | 1.系统管理员选择操作：对用户进行增加、删除、修改或查询操作。  2.提交修改。 |
| 其他事件流 | - |
| 异常事件流 | - |
| 后置条件 | 数据库对用户表进行修改。 |
| 注释 | - |



|  |  |
| --- | --- |
| 用例说明 | 详细信息 |
| 用例名称 | 浏览代码仓库 |
| 用例标识号 | 4 |
| 简要说明 | 所有用户都可以浏览公开的代码仓库。 |
| 前置条件 | 用户处于代码仓库界面。 |
| 基本事件流 | 1.显示版本关系图。  2.用户选择查看关注的版本。  3.预览显示用户选中的文件。 |
| 其他事件流 | 用户可评论代码仓库。 |
| 异常事件流 | 1.非仓库拥有者尝试修改代码仓库，系统提示无权限。  2.版本跳转参数无效提示参数错误。 |
| 后置条件 | - |
| 注释 | - |

|  |  |
| --- | --- |
| 用例说明 | 详细信息 |
| 用例名称 | 查看工作空间 |
| 用例标识号 | 5 |
| 简要说明 | 用户可在工作空间中查看已创建的代码仓库。 |
| 前置条件 | 用户处在工作空间页面。 |
| 基本事件流 | 1.显示该工作空间拥有者的用户名，描述信息。  2.显示工作空间拥有者创建的仓库列表。 |
| 其他事件流 | 1.用户可以通过仓库列表跳转到被选中的浏览代码仓库页面。  2.用户可以通过单击创建按钮跳转到创建仓库页面。 |
| 异常事件流 | 非工作空间拥有者尝试创建仓库是显示无创建权限。 |
| 后置条件 | - |
| 注释 | - |

|  |  |
| --- | --- |
| 用例说明 | 详细信息 |
| 用例名称 | 创建代码仓库 |
| 用例标识号 | 6 |
| 简要说明 | 为已登录用户创建新的代码仓库。 |
| 前置条件 | 1.用户已登录。  2.用户从工作空间页面跳转至该页面。  3.URL参数正确。 |
| 基本事件流 | 1.用户输入代码仓库名称和描述。  2.用户单击创建按钮。  3.服务器校验用户输入。如果输入为空或内容不符合要求，提示重新输入。 |
| 其他事件流 | 如果用户关闭页面，创建程序终止。 |
| 异常事件流 | 1.如果数据库或文件系统的操作中出现异常，则删除新增数据，回滚到未添加新代码仓库的状态。  2.URL参数错误，系统提示参数错误并关闭当前页面。 |
| 后置条件 | 1.系统为新代码仓库分配唯一的代码仓库ID，并记录代码仓库创建时间。  2.系统为新仓库提交第一个分支和第一个版本。新分支与新版本的描述信息与代码仓库的描述信息一致，名称统一为master。  3.设定主分支的版本范围。  4.将代码仓库更新为指向最新版本。  5.在文件系统中创建对应文件。  6.反馈是否创建成功。如果失败进行异常处理。 |
| 注释 | 1.每个代码仓库有且仅有一个主分支，其名称为master，且无法被移除。  2.每个分支（包括主分支）至少包含一个版本。 |



|  |  |
| --- | --- |
| 用例说明 | 详细信息 |
| 用例名称 | 仓库管理 |
| 用例标识号 | 7 |
| 简要说明 | 代码仓库拥有者对代码仓库进行版本控制。 |
| 前置条件 | 1.用户已登录。  2.用户对当前代码仓库具有修改权限。 |
| 基本事件流 | 1.用户可通过单击创建分支按钮跳转到创建分支界面。  2.用户可签入签出文件。  3.用户可以在该页面查看版本关系图，并可跳转到选中版本，也可以回滚到选中版本。  4.用户可以删除分支。  5.用户可以通过页面中的文件浏览器管理文件。 |
| 其他事件流 | 1.如果页面关闭，终止管理程序。 |
| 异常事件流 | 1.如果无操作权限，系统提示无权限。  2.如果未登录，跳转到登录界面。  3.如果数据库或文件系统的操作中出现异常，则删除新增数据，回滚到未添加新代码仓库的状态。 |
| 后置条件 | 反馈操作状态。 |
| 注释 | 主分支无法删除。每个分支至少包含一个版本。 |

|  |  |
| --- | --- |
| 用例说明 | 详细信息 |
| 用例名称 | 创建分支 |
| 用例标识号 | 8 |
| 简要说明 | 为代码仓库创建分支。 |
| 前置条件 | 1.用户已登录。  2.用户拥有指定代码仓库。  3.URL参数正确。  4.用户已选择作为分支起点的目标版本。 |
| 基本事件流 | 1.用户输入分支名和分支描述。  2.用户单击创建按钮。 |
| 其他事件流 | 如果用户关闭页面，创建程序终止。 |
| 异常事件流 | 1.输入为空或内容不符合规则，要求重新输入。  2.如果数据库或文件系统的操作中出现异常，则删除新增数据，回滚到未添加新代码仓库的状态。  3.URL参数错误，系统提示参数错误并关闭当前页面。 |
| 后置条件 | 1.以用户输入创建分支，自动生成ID和创建时间戳。  2.创建新分支的第一个版本，其名称与描述对应于分支的名称和描述。  3.设定新分支的版本范围。  4.反馈操作结果。如果失败则进行异常处理。 |
| 注释 | 用户创建的分支不能与主分支同名（master）。 |

|  |  |
| --- | --- |
| 用例说明 | 详细信息 |
| 用例名称 | 文件系统 |
| 用例标识号 | 9 |
| 简要说明 | 封装对文件的基本操作。 |
| 前置条件 | 1. 分布式文件系统服务启动成功。  2.web服务已启动。 |
| 基本事件流 | 根据接收到的指令进行对应的文件操作。 |
| 其他事件流 | 文件读写过程中与web服务器连接中断，文件操作终止。 |
| 异常事件流 | 程序崩溃，文件系统自动重启。 |
| 后置条件 | 反馈操作结果 |
| 注释 | 文件系统可以独立于web服务运行。 |

|  |  |
| --- | --- |
| 用例说明 | 详细信息 |
| 用例名称 | 删除分支 |
| 用例标识号 | 10 |
| 简要说明 | 删除指定代码仓库下选中的分支。 |
| 前置条件 | 1.用户已登录。  2.用户拥有指定代码仓库。  3.URL参数正确。  4.用户已选择分支。 |
| 基本事件流 | 1.从选中分支的起点，迭代删除分支中的版本和对应的文件结构。  2.从数据库清除选中的分支。 |
| 其他事件流 | 如果用户关闭页面，删除程序终止。 |
| 异常事件流 | 1.如果没有选中要删除的分支，系统提示参数错误。  2.如果数据库或文件系统的操作中出现异常，则删除新增数据，回滚到未添加新代码仓库的状态。  3.URL参数错误，系统提示参数错误并关闭当前页面。 |
| 后置条件 | 1.反馈操作结果，如果操作失败则进行异常处理。  2.操作成功后跳转到选定的代码仓库。 |
| 注释 | 主分支无法删除。 |

|  |  |
| --- | --- |
| 用例说明 | 详细信息 |
| 用例名称 | 回滚到选中版本 |
| 用例标识号 | 11 |
| 简要说明 | 把指定代码仓库下某一分支回滚到选中版本。 |
| 前置条件 | 1.用户已登录。  2.用户拥有指定代码仓库。  3.URL参数正确。  4.用户已选择分支和目标版本。 |
| 基本事件流 | 1.从选中版本的后面第一个版本开始，迭代删除分支中的版本和对应的文件结构。  2.在数据库中更新选中的分支。 |
| 其他事件流 | 如果用户关闭页面，则该进程终止。 |
| 异常事件流 | 1.如果没有选中目标分支和版本，系统提示参数错误。  2.如果数据库或文件系统的操作中出现异常，则删除新增数据，回滚到未添加新代码仓库的状态。  3.URL参数错误，系统提示参数错误并关闭当前页面。 |
| 后置条件 | 1.反馈操作结果，如果操作失败则进行异常处理。  2.操作成功后跳转到选定的代码仓库。 |
| 注释 | 每个分支至少有一个版本存在。 |

|  |  |
| --- | --- |
| 用例说明 | 详细信息 |
| 用例名称 | 推进新版本 |
| 用例标识号 | 12 |
| 简要说明 | 用户可以通过此页面手动推出新版本。 |
| 前置条件 | 1.用户已登录。  2.用户拥有指定代码仓库。  3.URL参数正确。  4.用户已选择目标版本。 |
| 基本事件流 | 1.用户输入新版本名称和描述。  2.用户单击创建按钮。 |
| 其他事件流 | 如果用户关闭当前页面，创建进程终止。 |
| 异常事件流 | 1.输入为空或内容不符合规则，要求重新输入。  2.如果数据库或文件系统的操作中出现异常，则删除新增数据，回滚到未添加新代码仓库的状态。  3.URL参数错误，系统提示参数错误并关闭当前页面。 |
| 后置条件 | 1.使用用户的输入信息创建新版本。  2.更新分支起止范围，如果更新的是主版本，则还要更新代码仓库记录信息中的最新主版本。  3.反馈操作结果。如果操作失败则启用异常处理流程。 |
| 注释 | - |

1.3 非功能需求

初期版本Web服务器应至少可承受50人同时在线，数据库服务器应至少可承受每秒500次查询，单个文件服务器在极限情况下应至少能承受每个会话125kb/s的文件传输流量。Web页面的加载时间应小于3秒。

1.4 非功能需求

1.4.1 运行环境规定

版本控制系统运行环境：

处理器：Intel(R) Pentium(R) 4 2.4 GHz 或 AMD(R) Athlon(TM) 64 2800+ 处理器 或任何 1.8Ghz Dual Core处理器。

显卡：NVIDIA(R) Geforce(TM) 6600 以上或 ATI(R) Radeon(R) 9800Pro以上 。

内存：8GB。

硬盘空：500GB。

操作系统：Windows8.1及Windows Server 2012以上的Windows操作系统。

客户端运行环境：

主流浏览器如chrome、firefox，IE等。

1.4.2 支持软件

开发环境使用Visual Studio 2015，开发语言采用C#、Javascript、C++、Html。数据库采用Microsoft SQL Server 2012，数据库控制中心使用Visual Studio 2010。数据库连接使用微软企业库。

2 系统总体设计

2.1 系统设计原则

一个优良的系统设计，强调模块间保持低耦合、高内聚的关系。本系统在设计和实现的过程中应尽可能遵守以下原则：

开闭原则(OCP):一个软件实体应当对扩展开放，对修改关闭。“抽象化”是OCP的关键。

里氏代换原则(LSP)：在一个软件系统中，子类应该可以替换任何基类能够出现的地方，并且经过替换以后，代码还能正常工作。“继承”是LSP的关键。

依赖倒转原则(DIP)：要依赖于抽象,不要依赖于具体。或者说是：要针对接口编程，不要对实现编程。“规范抽象”是DIP的关键。

接口隔离原则(ISP)：使用多个专门的接口比使用单一的总接口要好。也就是说，一个类对另外一个类的依赖性应当是建立在最小的接口上的。“多重继承”是ISP的关键。

组合/聚合复用原则(CARP)：在一个新的对象里面使用一些已有的对象，使之成为新对象的一部分：新的对象通过向这些对象的委派达到复用已有功能的目的。“组合/聚合”是CARP的关键。聚合指的是整体与部分的关系，在定义一个整体类后，再去分析这个整体类的组成结构。从而找出一些组成类，该整体类和组成类之间就形成了聚合关系。组合表示类之间整体和部分的关系，但是组合关系中部分和整体具有统一的生存期，一旦整体对象不存在，部分对象也将不存在。

迪米特法则(LoD)：一个对象应当对其他对象有尽可能少的了解。“传递间接的调用”是LoD的关键。

2.2 系统结构化分析

系统主要采用“用户界面-业务逻辑层-数据访问层”的结构。实体类用于映射数据库的表结构。使用这种设计模式的理由是，用一种业务逻辑、数据访问和界面显示分离的方法来组织代码，将业务逻辑集中到一个部件里面，在改进和个性化定制用户界面及用户交互过程的同时，不需要重新编写业务逻辑。



系统功能逻辑关系图



上下文图

普通用户和游客可以浏览系统中文件，可以进行注册和登录等操作，也可以对代码仓库进行版本控制；系统管理员有对整个系统的完全控制权限，能调整系统的运行状态，修改数据库，更新文件。在正常情况下系统会对普通用户、游客和系统管理员的一切操作进行反馈，提示用户的操作是否成功，显示用户所处的状态。

用户在版本控制系统中选择对文件系统和代码仓库的操作。用户可以选择对代码仓库、分支或版本进行版本控制操作,这些操作由数据库记录，系统会反馈对数据库的操作结果。版本控制系统通过内部协议，对分布式文件系统进行远程过程调用，完成文件操作。分布式文件系统会维持版本控制系统中，用户创建的代码仓库、分支和各版本的对应文件结构。

0层图

用户管理模块通过数据库操作模块完成注册、登录等操作,用户操作结果由控制系统模块直接返回。控制系统通过操作数据库和分布式文件系统来达到目的，数据库操作结果先返回到版本控制模块，将异常信息和错误代码转换为可读性较强的文本，作为普通用户的操作结果返回给用户。系统管理员对整个系统具有完全的控制权限，可以直接操作各个模块，也可以直接修改文件系统中的文件结构和数据库的内容。



用户管理1层图

没有帐号的游客可以通过用户管理模块注册新账户。已拥有账户的用户可以通过用户管理模块登录到系统，登录后对工作空间进行管理，也有修改已拥有代码仓库和账户信息的权限，同时可以向任意代码仓库提交评论。所有操作都经过控制系统，对数据库中的用户表进行操作；操作结果有控制系统反馈回用户。

控制系统1层图

控制系统作为业务逻辑层，整合对数据库实体的操作。Web服务通过该模块修改数据库，该模块将异常、错误代码和运行状态封装为可读性较强的文本返回给用户。

用户表数据字典

|  |  |
| --- | --- |
| 项目 | 内容 |
| 名称 | 用户表 |
| 别名 | user\_table |
| 使用地点 | 用户管理、控制系统。 |
| 使用方法 | 作为储存用户信息的容器。 |
| 描述 | 用户表=用户ID+用户名+用户帐户+用户密码+注册时间+用户描述+用户类型。 |

代码仓库数据字典

|  |  |
| --- | --- |
| 项目 | 内容 |
| 名称 | 代码仓库表 |
| 别名 | warehouse\_table |
| 使用地点 | 版本控制、控制系统 |
| 使用方法 | 记录用户创建的代码仓库的描述性信息。 |
| 描述 | 代码仓库表=代码仓库ID+用户ID+所属组织ID+代码仓库类型+创建时间+代码仓库描述+主分支最新版本ID+代码仓库名称。 |

分支表数据字典

|  |  |
| --- | --- |
| 项目 | 内容 |
| 名称 | 分支表 |
| 别名 | branch\_table |
| 使用地点 | 版本控制、控制系统。 |
| 使用方法 | 记录代码仓库下的所有分支。 |
| 描述 | 分支表=分支ID+代码仓库ID+用户ID+起始版本ID+结束版本ID+时间戳+分支名称+分支描述。 |

版本表数据字典

|  |  |
| --- | --- |
| 项目 | 内容 |
| 名称 | 版本表 |
| 别名 | version\_table |
| 使用地点 | 版本控制、控制系统。 |
| 使用方法 | 记录版本控制系统中所有的版本信息。 |
| 描述 | 版本表=当前版本ID+所属代码仓库ID+创建者ID+前一版本ID+下一版本ID+时间戳+版本名称+版本描述+所属分支ID。 |

2.3 数据库整体设计



版本控制系统实体关系图

2.3.1 数据表设计

版本控制系统中数据库表应遵守的原则：关系中的每个属性都不可再分；数据库表中的每个实例或行必须可以被唯一地区分；一个数据库表中不包含已在其它表中已包含的非主关键字信息。所有字段不可以是null，且必须具有初始值。默认情况下，值类型的字段默认值为0，字符串类型的默认值为长度为0的字符串。

用户表是用于记录用户信息的表。用户表具有一个作为主键的用户ID，用来唯一识别用户，用户名可以不唯一。用户账户必须唯一，因为用户将使用账户与密码登录系统。用户表还应记录账户的创建时间和用户的个人描述，以方便用户之间的理解和交流。

代码仓库表用于记录用户创建的代码仓库的信息。代码仓库表具有一个具有一个作为主键的代码仓库ID，用于唯一识别每个代码仓库。该表应记录是哪个组织的哪个用户在什么时间创建了代码仓库，还要记录代码仓库的名称、类型、描述信息和主分支的最新版本ID。在创建代码仓库时，代码控制系统会自动创建主分支，并添加主分支下的第一个版本。每个代码仓库只能有一个名为master的分支作为主分支。版本控制操作应遵循每个分支至少具有一个版本这一原则。

分支表用于记录代码仓库中出现的分支信息。分支表具有一个作为主键的分支ID，用于唯一标识每个分支；具有记录该分支属于哪个代码仓库的字段，同时具有记录该代码仓库拥有者的字段。分支表记录一个分支的名称、创建时间、描述，还应记录分支下各个版本的起点和终点。代码仓库的主分支无法被删除，用户无法创建名为master的分支，每个分支至少包含一个版本。

版本表用于记录版本控制系统中的所有版本。版本表内的数据可以通过前向和后向指针形成有向图的结构。该表应记录版本是属于哪个代码仓库的哪个分支，记录是由哪个用户在什么时间创建，还要记录版本的名称和描述。版本由主键版本ID唯一标识。

组织表用于记录用户所属的组织机构，该表记录组织机构的名称描述，具有一个组织ID作为主键。评论表记录用户向代码仓库提交的评论的内容。

2.3.2 数据访问层设计

数据访问层即DAL层，也称为是久层，其功能主要是负责数据库的访问。实现对数据表的Select（查询），Insert（插入），Update（更新），Delete（删除）等操作。如果加入ORM的元素，那么也包括对象和数据表之间的映射以及对象实体的持久化。数据库访问层的主要职责是：读取数据和传递数据。

2.4 用户界面总体设计

Index.aspx为进入系统的主页。主页中概述的介绍系统的功能，用户也可以通过此页面跳转到注册新账户或登录页面。

Register.aspx为注册账户页面。该界面上有记录新账户用户名、登录帐号、密码和用户描述信息的文本输入框，此外还有一个创建按钮。用户单击创建按钮后，如果输入的内容满足数据库的存储要求，则新账户创建成功，自动跳转至登录窗口，引导用户登录新账户；如果创建失败，则要求用户重新输入注册信息。

Login.aspx为用户登录界面。该界面上有记录用户帐户和密码的输入框，还有一个登录按钮。用户单击登录按钮后，服务器校验用户输入的账户与密码是否匹配。如果账户和密码正确，服务器记录登录状态，用户浏览器自动跳转到用户工作空间界面；如果出现错误则要求用户重新输入登录凭据。

User\_page.aspx为用户工作空间。该界面的上方显示用户名和用户描述，这里的数据来自于注册时用户填写的数据；通过这里的查看评论按钮，用户空间拥有者可以跳转到查看评论的页面。页面的中部有创建代码仓库的按钮，用户单击后可跳转到创建代码仓库页面。界面的下方显示该用户创建的代码仓库的预览信息，包括跳转链接、创建时间和项目名称，点击跳转链接，用户浏览器可跳转至选中代码仓库的详细信息页面。

Create\_page.aspx为创建代码仓库页面。该页面具有记录新代码仓库名称、项目描述的文本输入框，此外还有一个创建按钮。用户单击创建按钮，服务器检查用户权限和用户提交的数据，如果符合创建规则，服务器在数据库中创建对应数据，在文件系统中创建对应的文件结构。如果用户输入为空或不符合数据库存储条件，提示用户重新输入相关内容。

Warehouse\_page.aspx为代码仓库详细信息页面，也是实现版本控制功能的核心页面之一。该页面的顶端显示当前代码仓库的名称和项目描述，这部分数据来自于代码仓库被创建时用户提交的数据。该页面的中上部分为该代码仓库的版本结构图，该部分显示项目的分支和版本之间的联系。

页面的中下部分为版本控制功能选择区。在这里首先显示用户当前浏览的版本的详细信息、版本所属分支、版本创建时间、版本描述，用户在这里可以选择对当前版本的操作，这里有签出当前版本、推送新版本、创建分支、删除当前分支按钮。用户单击按钮后服务器先验证当前用户是否是当前代码仓库的拥有者，如果是则执行对应操作，并返回最新版本信息；如果不是则提示没有权限。通过单击推送新版本和创建分支按钮，用户可以跳转到对应页面。该部分也会显示用户在版本结构图选中版本的部分信息，如所在分支和版本ID。用户可以通过单击跳转到选中版本按钮重定向至选中版本的详细信息页面。此外用户单击回滚至选中版本按钮将选中分支回滚到指定版本。

该页面的下部为一个简单的文件浏览器，用户可以在此浏览当前版本的文件结构，浏览选中的文件，上传或下载选中的文件。该部分还包含文件列表，文件内容查看组件，返回至版本根目录按钮，返回上级菜单按钮。

Create\_branch.aspx为创建新分支界面。该界面包含记录新分支名称和描述信息的文本输入框，还有创建按钮。用户单击创建按钮，服务器检查提交内容和用户权限，如果符合创建规则，则记录新分支并创建对应的文件结构，如果不符合创建规则，提示用户重新输入并提交数据。

Create\_version.aspx为推送新版本页面。与创建新分支界面类似，该界面包含记录新版本名称和描述信息的文本输入框，以及创建按钮。用户单击创建按钮，服务器检查提交内容和用户权限，如果符合创建规则，则在对应分支创建新版本并创建对应的文件结构，如果不符合创建规则，提示用户重新输入并提交数据。

UploadFile.aspx为上传文件界面。包含记录文件名的文本输入框和提交按钮。用户应是代码仓库的拥有者。用户单击提交按钮后，如果输入不为空且符合规则，则提示创建成功，之后返回到代码仓库界面。

CreateFolder.aspx为创建文件夹。包含记录文件夹名称的文本输入框和提交按钮。用户应是代码仓库的拥有者。用户单击提交按钮后，如果输入不为空且符合规则，则提示创建成功，之后返回到代码仓库界面。

Comments.aspx为查看评论页面。该页面显示一个，每一项包含其他用户对代码仓库、版本和分支的评论信息。

2.5 文件系统设计



分布式文件系统结构图

分布式文件系统在版本控制系统中，是一个独立于Web服务的组件。将支撑业务的文件系统与Web服务分离，一方面模块化功能加快设计及开发进度，另一方面尽可能增加文件系统的灵活性，满足业务对存储容量、访问速度、文件安全性的需求。



服务器集群结构

服务器集群包含一个主节点，负责负载均衡和记录操作状态；集群中也可以添加多个从节点，拓展主节点的功能，或完全承担业务功能，其功能根据应用可以不同，执行的功能由集群内部通讯协议决定。而在版本控制系统中，文件系统作为一个服务运行于分布式系统中，所有节点执行相同的功能。文件系统服务采用异步架构完成文件操作请求。

一个文件系统应具备组织文件和文件夹的能力，如创建、删除文件或文件夹，遍历目录，获取文件或文件夹属性，读写文件等功能，版本控制系统中的文件操作接口对应于以上这几方面。

分布式文件系统并不是版本控制系统的核心内容，因此重要程度较低，视时间安排尽可能实现功能。版本控制系统中留出文件系统的接口，如果分布式文件系统实现对应的功能，就将其添加到版本控制系统中；而未实现的功能则使用Web服务器本身的文件操作功能。

3 系统详细设计

3.1 数据库定义及初始化

创建用户表：create table user\_table( user\_id int primary key,username nvarchar(64) not null,user\_account nvarchar(64) not null,user\_password nvarchar(64) not null,register\_time nvarchar(32) default '' not null,user\_description nvarchar(256) default '' not null,user\_type int default 0 not null );

创建代码仓库表：Create table warehouse\_table(warehouse\_id int primary key,user\_id int not null,organization\_id int default 0 not null,warehouse\_type int default 0 not null,create\_time varchar(32) default '' not null, warehouse\_description varchar(512) default '' not null, master\_version\_id int default 0 not null, warehouse\_name varchar(64) not null default '');

创建分支表：create table branch\_table(branch\_id int primary key not null, warehouse\_id int not null default 0, user\_id int not null default 0,start\_id int not null default 0, end\_id int not null default 0,timestamp varchar(32) not null default '', branch\_name varchar(64) not null default '', description varchar(1024) not null default '');

创建版本表：create table version\_table(version\_id int primary key, warehouse\_id int not null default 0, user\_id int not null default 0, prev\_id int not null default 0, next\_id int not null default 0,timestamp varchar(32) not null default '',version\_name varchar(64) not null default '', description varchar(512) not null default '', branch\_id int not null default 0);

创建评论表：create table comment\_table(comment\_id int primary key,user\_id int not null default 0,target\_user\_id int not null default 0,warehouse\_id int not null default 0,content nvarchar(1024) not null default'');

初始化各表：

insert into branch\_table values(0,0,0,0,0,'time','master','description');

insert into version\_table values(0,0,0,0,0,'0','0','0',0);

insert into warehouse\_table values(0,0,0,0,'0','0',0,'0');

insert into user\_table values(0,’nemo’,’nemo’,’password’,’0’,’null’,0);

insert into comment\_table(0,0,0,0,’null’);

3.2 数据库实体



用户类

User类用于记录用户信息。用户名（user\_name）、用户帐户（user\_account）和用户密码（user\_password）的长度限制为64字节，注册时间长度限制为32字节，用户描述（user\_description）长度限制为256字节。用户名中不可以出现下划线以外的符号，用户帐户只能由数字、字母和下划线构成，密码和用户描述在长度限制内，可以为任意内容。可以有多个用户使用相同的用户名，但用户帐户是全局唯一的。



代码仓库类

代码仓库类用于存储用户创建的代码仓库的信息，在这里不保存文件结构和内容。关于数据的长度限制，创建时间（create\_time）为32字节，代码仓库描述（warehouse\_description）为512字节，代码仓库名称（warehouse\_name）为64字节。代码仓库名称只能由数字、字母和下划线构成，代码仓库描述对内容无要求。Master\_version\_id记录的是主分支最新版本ID。



版本类

版本类用于记录版本控制过程中产生的版本，版本的文件结构和数据由文件系统保存，这里只保存版本的描述性信息。关于字符串的长度限制，时间戳（timestamp）为32字节，版本名称（verson\_name）为64字节，版本描述（description）为512字节。版本名称只能由数字、字母和下划线构成，版本描述对内容无要求。版本表的每一条记录，都是某个版本图的一部分，其结构类似于双向链表，prev\_id和next\_id分别记录着前一版本和后一版本的ID，如果不存在则为0。分支的起点和重点在分支表中记录。分支ID（branch\_id）记录当前版本属于哪个分支。



分支类

分支类用于记录版本控制过程中产生的分支。start\_id和end\_id分别记录分支起点版本和终点版本ID。每个代码仓库至少有一个名为“master”的主分支，该分支有服务器在创建代码仓库时自动创建，该分支无法被删除，用户无法创建名为“master”的分支。每个分支中至少有一个版本。字符串的长度限制：时间戳（timestamp）为32字节，分支名称（branch\_name）为64字节，分支描述（description）为1024字节。分支名称只能由数字、字母和下划线构成，分支版本描述对内容无要求。



评论类

评论类用于记录用户提交的评论。评论内容（content）的长度限制为1024字节。评论提交者（user\_id）向评论接受者（target\_user\_id）提交评论，评论的代码仓库由warehouse\_id记录。

3.3 数据访问层



UserDAL类

UserDAL用于操作用户类在数据库中的记录。包含对数据库的增删改查操作。



WarehouseDAL类图

WarehouseDAL用于操作代码类在数据库中的记录。



VersionDAL类图

VersionDAL用于操作版本类在数据库中的记录。



BranchDAL类图

BranchDAL用于操作分支类在数据库中的记录。



CommentDAL类图

CommentDAL用于操作评论类在数据库中的记录。

3.4 用户页面功能设计

Index.aspx为进入系统的主页。主页中概述的介绍系统的功能，用户也可以通过此页面跳转到注册新账户或登录页面。

Register.aspx为注册账户页面。用户单击创建按钮后，如果输入的内容满足数据库的存储要求，则新账户创建成功，自动跳转至登录窗口，引导用户登录新账户；如果创建失败，则要求用户重新输入注册信息。

Login.aspx为用户登录界面。。用户单击登录按钮后，服务器校验用户输入的账户与密码是否匹配。如果账户和密码正确，服务器记录登录状态，用户浏览器自动跳转到用户工作空间界面；如果出现错误则要求用户重新输入登录凭据。用户登录成功后，在Session中写入用户信息。

User\_page.aspx为用户工作空间。该界面的上方显示用户名和用户描述，这里的数据来自于注册时用户填写的数据；通过这里的查看评论按钮，用户空间拥有者可以跳转到查看评论的页面。页面的中部有创建代码仓库的按钮，用户单击后可跳转到创建代码仓库页面。界面的下方显示该用户创建的代码仓库的预览信息，包括跳转链接、创建时间和项目名称，点击跳转链接，用户浏览器可跳转至选中代码仓库的详细信息页面。在进入该页面时，URL需要有1个参数wid，分即代码仓库ID，如果参数不合法则提示非法参数。

Create\_page.aspx为创建代码仓库页面。用户单击创建按钮，服务器检查用户权限和用户提交的数据，如果符合创建规则，服务器在数据库中创建对应数据，在文件系统中创建对应的文件结构。如果用户输入为空或不符合数据库存储条件，提示用户重新输入相关内容。在进入该页面时，URL需要有2个参数，vid和wid，分别是版本ID和代码仓库ID，如果参数不合法则提示非法参数。

Warehouse\_page.aspx为代码仓库详细信息页面，也是实现版本控制功能的核心页面之一。在进入该页面时，URL需要有2个参数，vid和wid，分别是版本ID和代码仓库ID，如果参数不合法则提示非法参数。该页面的中上部分为该代码仓库的版本结构图，该部分显示项目的分支和版本之间的联系。用户单击某个按钮后服务器先验证当前用户是否是当前代码仓库的拥有者，如果是则执行对应操作，并返回最新版本信息；如果不是则提示没有权限。通过单击推送新版本和创建分支按钮，用户可以跳转到对应页面。该部分也会显示用户在版本结构图选中版本的部分信息，如所在分支和版本ID。用户可以通过单击跳转到选中版本按钮重定向至选中版本的详细信息页面。此外用户单击回滚至选中版本按钮将选中分支回滚到指定版本。

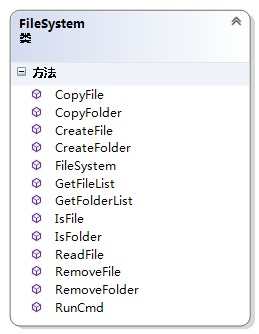
该页面的下部为一个简单的文件浏览器，用户可以在此浏览当前版本的文件结构，浏览选中的文件，上传或下载选中的文件。该部分还包含文件列表，文件内容查看组件，返回至版本根目录按钮，返回上级菜单按钮。

Create\_branch.aspx为创建新分支界面。用户单击创建按钮，服务器检查提交内容和用户权限，如果符合创建规则，则记录新分支并创建对应的文件结构，如果不符合创建规则，提示用户重新输入并提交数据。

Create\_version.aspx为推送新版本页面。用户单击创建按钮，服务器检查提交内容和用户权限，如果符合创建规则，则在对应分支创建新版本并创建对应的文件结构，如果不符合创建规则，提示用户重新输入并提交数据。

Comments.aspx为查看评论页面。该页面显示一个，每一项包含其他用户对代码仓库、版本和分支的评论信息。

3.5 文件系统设计



文件系统类图

文件系统具有如下基本功能，判断一个路径是文件夹还是文件，创建文件或文件夹，移动、复制、删除文件或文件夹，获得某一个路径下文件和文件夹列表，读写文件。

服务器集群内部通讯协议与HTTP类似，协议头部采用键值对组，每一组键值对后有\r\n两个字符。数据段在一个空行之后，如果在通讯中携带数据，则需在头部指定Content-Length字段，即系带数据的长度，服务器协议组件会解析该字段数据并和数据长度对比，如果长度不符，则丢弃数据。协议格式如下：

key1:value1\r\n

key2:value2\r\n

[ContentLength:2333\r\n]

\r\n

[Content]

同步IO：假如对一个文件（socket也同理）进行处理，那么一般的流程就是：定义一个文件流对象，将文件流对应到指定文件，读取文件内容，对读取到的进行操作，最后关闭文件流。

通常情况下，当这个线程运行到读函数时会被阻塞，直到文件读取完成。

在异步情况下：还是上面的流程，我在读函数时通过操作系统或库提供的异步机制，告诉操作系统我想读一个文件，数据读完后执行某个功能；而当前线程在交代完操作系统该做什么工作之后，还可以做些别的事情（线程不必等待文件IO完成）。

线程池：为了避免IO阻塞线程导致程序无响应，完全可以为每一个文件操作创建一个线程，这样就可以同时处理多个文件了。但是创建线程，切换线程，销毁线程也是一笔资源开销，如果想重复使用已有的线程，就可以使用线程池。作为线程池，至少要提供创建线程和提交任务的功能，复杂一点可以智能控制线程池里的线程数量,还应该具有基本的负载均衡功能。这个文件系统中就会使用线程池。

这个模块只使用stl 和boost 两个库。stl主要涉及容器和fstream。boost涉及到智能指针shared\_ptr，线程同步shared\_mutex,lock\_guard, boost::filesystem中的path和一些文件操作，线程操作创建退出等。

关于智能指针。自古以来内存管理都是C/C++中的重头戏，智能指针的功能就是分配出来的内存由库管理，如果某个智能指针指向的内存，通过其他的智能指针也能访问到（即有多个引用），那么该智能指针即时被销毁，指向的内存也不会销毁；只有这块内存没有引用，才会被库释放。

boost::filesystem库提供了一些跨平台文件操作的API，如文件夹遍历，查看属性，删除文件等。path类可以记录跨平台的路径。

数据结构及定义：

AsyncStatus是异步IO中需要实现的功能。像读，写，放弃异步操作，错误处理等。

ErrorCode会出现在回调函数中，表示之前异步读写的结果，如正在处理，出错，EOF等。

FS\_Handle\_ST：这个结构体对应一个文件路径。在系统中每个handle都是唯一的，系统有一个map，通过handle可以找到它对应的路径。

FS\_AsyncHandle\_ST:标识某个handle需要执行的任务，每个AsyncHandle都需要指定status即任务。一个handle可以有多个异步任务，但多个任务在系统中按照队列顺序执行。

回调函数FileSystemIOCallback:定义异步操作完成之后要做什么。使用的时候把功能在派生类里实现，重载虚函数run就可以了。

3.6 软件测试

单元测试是对软件中的最小可测试单元进行检查和验证。对于单元测试中单元的含义，通常要根据实际情况去判定其具体含义，如C语言中单元指一个函数，Java里单元指一个类，图形化的软件中可以指一个窗口或一个菜单等。总的来说，单元就是人为规定的最小的被测功能模块。单元测试是在软件开发过程中要进行的最低级别的测试活动，软件的独立单元通常将在与程序的其他部分相隔离的情况下进行单元测试。

在一种传统的结构化编程语言中，比如C，要进行测试的单元一般是函数或子过程。在像C++和C#这样的面向对象的语言中，要进行测试的基本单元是类。单元测试的基本单元也可以被划分为一个菜单或显示界面。经常与单元测试联系起来的另外一些开发活动包括代码走读，静态分析和动态分析。静态分析就是对软件的源代码进行研读，查找错误或收集一些度量数据，并不需要对代码进行编译和执行。动态分析就是通过观察软件运行时的动作，来提供执行跟踪，时间分析，以及测试覆盖度方面的信息。

总结

这个毕业设计阐述了一个构建版本控制系统的方法。版本管理是软件配置管理的基础，它管理并保护开发者的软件资源。本系统作为软件开发过程中的辅助工具，目的在于减少版本控制及管理过程中使用的人力资源，协调进行不同工作的开发人员的工作，同步不同开发者的进度。系统需要尽可能保存每一阶段的工作成果，尤其是源文件，以保证每个阶段性工作成果的安全，这样任何时候都可以方便的找回原来的工作成果。

随着软件系统规模的日益扩大和复杂程度的日益增长，软件工程师或网页设计师可能会需要保存某一系统的源码或页面布局文件的所有修订版本，采用版本控制系统是个明智的选择。有了版本控制系统就可以将某个文件回溯到之前的状态，甚至将整个项目都回退到过去某个时间点的状态，就算对整个项目中的文件进行大幅度改动，也照样可以轻松恢复到原先稳定版本的样子，但额外增加的工作量却微乎其微。

在软件开发过程中，参与项目的人数多不代表一定可以加快项目的开发速度，因为软件产品的抽象性，人员之间的沟通与管理极有可能占用很多资源，导致整体开发效率下降，而版本控制系统的作用就是协调开发人员的工作，同步不同开发者的进度，尽可能减少版本管理占用的资源。

**致谢**

首先非常感谢软件学院的各位老师，老师们严谨细致、一丝不苟的作风一直是我工作、学习中的榜样；他们循循善诱的教导和精彩绝伦的思路常常给予我启迪。之后我要感谢指导我完成毕业设计的刘亮老师，本毕业设计获得老师的悉心指导和严格要求，从课题选择、方案论证到具体设计和调试，无不凝聚着老师的心血和汗水，在完成毕业设计的过程中，也始终感受着导师的精心指导和无私的关怀，我受益匪浅。在此向刘亮老师表示深深的感谢和崇高的敬意。最后我要感谢我的父母，在我的成长历程里离不开父母的鼓励和支持，是他们辛勤的劳作，无私的付出，为我创造良好的学习条件，我才能顺利完成完成学业，感激他们一直以来对我的抚养与培育。

我的论文和作品不是很成熟，还有很多不足之处，但这里面的每一段代码，每一段文字，都有我的劳动。这次做论文的经历也会使我终身受益，我感受到做论文是要真真正正用心去做的一件事情，是真正的自己学习的过程和研究的过程，没有学习就不可能有研究的能力，没有自己的研究，就不会有所突破。希望这次的经历能让我在以后学习中激励我继续进步。

**参考文献**

**附录A 译文**

我们设计并实现了谷歌文件系统(Google File System – GFS)，一个面向大规模数据密集型应用的、可伸缩的分布式文件系统。GFS虽然运行在廉价的普遍硬件设备上，但是它依然了提供灾难冗余的能力，为大量客户机提供了高性能的服务。

虽然GFS的设计目标与许多传统的分布式文件系统有很多相同之处，但是，我们的设计还是以我们对自己的应用的负载情况和技术环境的分析为基础 的，不管现在还是将来，GFS和早期的分布式文件系统的设想都有明显的不同。所以我们重新审视了传统文件系统在设计上的折衷选择，衍生出了完全不同的设计 思路。

GFS完全满足了我们对存储的需求。GFS作为存储平台已经被广泛的部署在Google内部，存储我们的服务产生和处理的数据，同时还用于那些 需要大规模数据集的研究和开发工作。目前为止，最大的一个集群利用数千台机器的数千个硬盘，提供了数百TB的存储空间，同时为数百个客户机服务。

在本论文中，我们展示了能够支持分布式应用的文件系统接口的扩展，讨论我们设计的许多方面，最后列出了小规模性能测试以及真实生产系统中性能相关数据。

为了满足Google迅速增长的数据处理需求，我们设计并实现了Google文件系统。GFS与传统的分布式文件系统有着很多相同的设计目标，比如，性能、可伸缩性、可靠性以及可用性。但是，我们的设计还基于我们对我们自己的应用 的负载情况和技术环境的观察的影响，不管现在还是将来，GFS和早期文件系统的假设都有明显的不同。所以我们重新审视了传统文件系统在设计上的折衷选择，衍生出了完全不同的设计思路。

首先，组件失效被认为是常态事件，而不是意外事件。GFS包括几百甚至几千台普通的廉价设备组装的存储机器，同时被相当数量的客户机访问。GFS组件的数量和质量导致在事实上，任何给定时间内都有可能发生某些组件无法工作，某些组件无法从它们目前的失效状态中恢复。我们遇到过各种各样的问题，比如应用程序bug、操作系统的bug、人为失误，甚至还有硬盘、内存、连接器、网络以及电源失效等造成的问题。所以，持续的监控、错误侦测、灾难冗余以及自动恢复的机制必须集成在GFS中。

其次，以通常的标准衡量，我们的文件非常巨大。数GB的文件非常普遍。每个文件通常都包含许多应用程序对象，比如web文档。当我们经常需要处理快速增长的、并且由数亿个对象构成的、数以TB的数据集时，采用管理数亿个KB大小的小文件的方式是非常不明智的，尽管有些文件系统支持这样的管理方 式。因此，设计的假设条件和参数，比如I/O操作和Block的尺寸都需要重新考虑。

第三，绝大部分文件的修改是采用在文件尾部追加数据，而不是覆盖原有数据的方式。对文件的随机写入操作在实际中几乎不存在。一旦写完之后，对文 件的操作就只有读，而且通常是按顺序读。大量的数据符合这些特性，比如：数据分析程序扫描的超大的数据集；正在运行的应用程序生成的连续的数据流；存档的 数据；由一台机器生成、另外一台机器处理的中间数据，这些中间数据的处理可能是同时进行的、也可能是后续才处理的。对于这种针对海量文件的访问模式，客户 端对数据块缓存是没有意义的，数据的追加操作是性能优化和原子性保证的主要考量因素。

第四，应用程序和文件系统API的协同设计提高了整个系统的灵活性。比如，我们放松了对GFS一致性模型的要求，这样就减轻了文件系统对应用程 序的苛刻要求，大大简化了GFS的设计。我们引入了原子性的记录追加操作，从而保证多个客户端能够同时进行追加操作，不需要额外的同步操作来保证数据的一致性。本文后面还有对这些问题的细节的详细讨论。

Google已经针对不同的应用部署了多套GFS集群。最大的一个集群拥有超过1000个存储节点，超过300TB的硬盘空间，被不同机器上的数百个客户端连续不断的频繁访问。

在设计满足我们需求的文件系统时候，我们的设计目标既有机会、又有挑战。之前我们已经提到了一些需要关注的关键点，这里我们将设计的预期目标的细节展开讨论。

系统由许多廉价的普通组件组成，组件失效是一种常态。系统必须持续监控自身的状态，它必须将组件失效作为一种常态，能够迅速地侦测、冗余并恢复失效的组件。

系统存储一定数量的大文件。我们预期会有几百万文件，文件的大小通常在100MB或者以上。数个GB大小的文件也是普遍存在，并且要能够被有效的管理。系统也必须支持小文件，但是不需要针对小文件做专门的优化。

系统的工作负载主要由两种读操作组成：大规模的流式读取和小规模的随机读取。大规模的流式读取通常一次读取数百KB的数据，更常见的是一次读取1MB甚至更多的数据。来自同一个客户机的连续操作通常是读取同一个文件中连续的一个区域。小规模的随机读取通常是在文件某个随机的位置读取几个KB数据。如果应用程序对性能非常关注，通常的做法是把小规模的随机读取操作合并并排序，之后按顺序批量读取，这样就避免了在文件中前后来回的移动读取位置。

系统的工作负载还包括许多大规模的、顺序的、数据追加方式的写操作。一般情况下，每次写入的数据的大小和大规模读类似。数据一旦被写入后，文件就很少会被修改了。系统支持小规模的随机位置写入操作，但是可能效率不彰。

系统必须高效的、行为定义明确的实现多客户端并行追加数据到同一个文件里的语意。我们的文件通常被用于“生产者-消费者”队列，或者其它多路文件合并操作。通常会有数百个生产者，每个生产者进 程运行在一台机器上，同时对一个文件进行追加操作。使用最小的同步开销来实现的原子的多路追加数据操作是必不可少的。文件可以在稍后读取，或者是消费者在 追加的操作的同时读取文件。

高性能的稳定网络带宽远比低延迟重要。我们的目标程序绝大部分要求能够高速率的、大批量的处理数据，极少有程序对单一的读写操作有严格的响应时间要求。

GFS提供了一套类似传统文件系统的API接口函数，虽然并不是严格按照POSIX等标准API的形式实现的。文件以分层目录的形式组织，用路径名来标识。我们支持常用的操作，如创建新文件、删除文件、打开文件、关闭文件、读和写文件。

另外，GFS提供了快照和记录追加操作。快照以很低的成本创建一个文件或者目录树的拷贝。记录追加操作允许多个客户端同时对一个文件进行数据追 加操作，同时保证每个客户端的追加操作都是原子性的。这对于实现多路结果合并，以及”生产者-消费者”队列非常有用，多个客户端可以在不需要额外的同步锁 定的情况下，同时对一个文件追加数据。我们发现这些类型的文件对于构建大型分布应用是非常重要的。快照和记录追加操作将在3.4和3.3节分别讨论。

一个GFS集群包含一个单独的Master节点、多台 Chunk服务器，并且同时被多个客户端访问，如图1所示。所有的这些机器通常都是普通的Linux机器，运行着用户级别(user-level)的服务 进程。我们可以很容易的把Chunk服务器和客户端都放在同一台机器上，前提是机器资源允许，并且我们能够接受不可靠的应用程序代码带来的稳定性降低的风险。

GFS存储的文件都被分割成固定大小的Chunk。在Chunk创建的时候，Master服务器会给每个Chunk分配一个不变的、全球唯一的 64位的Chunk标识。Chunk服务器把Chunk以linux文件的形式保存在本地硬盘上，并且根据指定的Chunk标识和字节范围来读写块数据。出于可靠性的考虑，每个块都会复制到多个块服务器上。缺省情况下，我们使用3个存储复制节点，不过用户可以为不同的文件命名空间设定不同的复制级别。

Master节点管理所有的文件系统元数据。这些元数据包括名字空间、访问控制信息、文件和Chunk的映射信息、以及当前Chunk的位置信息。Master节点还管理着系统范围内的活动，比如，Chunk租用管理、孤儿Chunk的回收、以及Chunk在Chunk服务器之间的迁移。Master节点使用心跳信息周期地和每个Chunk服务器通讯，发送指令到各个Chunk服务器并接收Chunk服务器的状态信息。

GFS客户端代码以库的形式被链接到客户程序里。客户端代码实现了GFS文件系统的API接口函数、应用程序与Master节点和Chunk服 务器通讯、以及对数据进行读写操作。客户端和Master节点的通信只获取元数据，所有的数据操作都是由客户端直接和Chunk服务器进行交互的。我们不 提供POSIX标准的API的功能，因此，GFS API调用不需要深入到Linux vnode级别。

无论是客户端还是Chunk服务器都不需要缓存文件数据。客户端缓存数据几乎没有什么用处，因为大部分程序要么以流的方式读取一个巨大文件，要么工作集太大根本无法被缓存。无需考虑缓存相关的问题也简化了客户端和整个系统的设计和实现。（不过，客户端会缓存元数据。）Chunk服务器不需要缓存文件数据的原因是，Chunk以本地文件的方式保存，Linux操作系统的文件系统缓存会把经常访问的数据缓存在内存中。

单一的Master节点的策略大大简化了我们的设计。单一的Master节点可以通过全局的信息精确定位Chunk的位置以及进行复制决策。另外，我们必须减少对Master节点的读写，避免Master节点成为系统的瓶颈。客户端并不通过Master节点读写文件数据。反之，客户端向Master节点询问它应该联系的Chunk服务器。客户端将这些元数据信息缓存一段时间，后续的操作将直接和Chunk服务器进行数据读写操作。

我们利用图1解释一下一次简单读取的流程。首先，客户端把文件名和程序指定的字节偏移，根据固定的Chunk大小，转换成文件的Chunk索 引。然后，它把文件名和Chunk索引发送给Master节点。Master节点将相应的Chunk标识和副本的位置信息发还给客户端。客户端用文件名和 Chunk索引作为key缓存这些信息。

之后客户端发送请求到其中的一个副本处，一般会选择最近的。请求信息包含了Chunk的标识和字节范围。在对这个Chunk的后续读取操作中，客户端不必再和Master节点通讯了，除非缓存的元数据信息过期或者文件被重新打开。实际上，客户端通常会在一次请求中查询多个Chunk信 息，Master节点的回应也可能包含了紧跟着这些被请求的Chunk后面的Chunk的信息。在实际应用中，这些额外的信息在没有任何代价的情况下，避 免了客户端和Master节点未来可能会发生的几次通讯。

Chunk的大小是关键的设计参数之一。我们选择了64MB，这个尺寸远远大于一般文件系统的Block size。每个Chunk的副本都以普通Linux文件的形式保存在Chunk服务器上，只有在需要的时候才扩大。惰性空间分配策略避免了因内部碎片造成 的空间浪费，内部碎片或许是对选择这么大的Chunk尺寸最具争议一点。

选择较大的Chunk尺寸有几个重要的优点。首先，它减少了客户端和Master节点通讯的需求，因为只需要一次和Mater节点的通信就可以 获取Chunk的位置信息，之后就可以对同一个Chunk进行多次的读写操作。这种方式对降低我们的工作负载来说效果显著，因为我们的应用程序通常是连续读写大文件。即使是小规模的随机读取，采用较大的Chunk尺寸也带来明显的好处，客户端可以轻松的缓存一个数TB的工作数据集所有的Chunk位置信 息。其次，采用较大的Chunk尺寸，客户端能够对一个块进行多次操作，这样就可以通过与Chunk服务器保持较长时间的TCP连接来减少网络负载。第三，选用较大的Chunk尺寸减少了Master节点需要保存的元数据的数量。这就允许我们把元数据全部放在内存中，在后面我们会讨论元数据全部 放在内存中带来的额外的好处。

另一方面，即使配合惰性空间分配，采用较大的Chunk尺寸也有其缺陷。小文件包含较少的Chunk，甚至只有一个Chunk。当有许多的客户端对同一个小文件进行多次的访问时，存储这些Chunk的Chunk服务器就会变成热点。在实际应用中，由于我们的程序通常是连续的读取包含多个 Chunk的大文件，热点还不是主要的问题。

然而，当我们第一次把GFS用于批处理队列系统的时候，热点的问题还是产生了：一个可执行文件在GFS上保存为single-chunk文件，之后这个可执行文件在数百台机器上同时启动。存放这个可执行文件的几个Chunk服务器被数百个客户端的并发请求访问导致系统局部过载。我们通过使用更大 的复制参数来保存可执行文件，以及错开批处理队列系统程序的启动时间的方法解决了这个问题。一个可能的长效解决方案是，在这种的情况下，允许客户端从其它 客户端读取数据。

Master服务器存储3种主要类型的元数据，包括：文件和Chunk的命名空间、文件和Chunk的对应关系、每个Chunk副本的存放地点。所有的元数据都保存在 Master服务器的内存中。前两种类型的元数据（命名空间、文件和Chunk的对应关系）同时也会以记录变更日志的方式记录在操作系统的系统日志文件 中，日志文件存储在本地磁盘上，同时日志会被复制到其它的远程Master服务器上。采用保存变更日志的方式，我们能够简单可靠的更新Master服务器 的状态，并且不用担心Master服务器崩溃导致数据不一致的风险。Master服务器不会持久保存Chunk位置信息。Master服务器在启动时，或 者有新的Chunk服务器加入时，向各个Chunk服务器轮询它们所存储的Chunk的信息。

因为元数据保存在内存中，所以Master服务器的操作速度非常快。并且，Master服务器可以在后台简单而高效的周期性扫描自己保存的全部 状态信息。这种周期性的状态扫描也用于实现Chunk垃圾收集、在Chunk服务器失效的时重新复制数据、通过Chunk的迁移实现跨Chunk服务器的 负载均衡以及磁盘使用状况统计等功能。

将元数据全部保存在内存中的方法有潜在问题：Chunk的数量以及整个系统的承载能力都受限于Master服务器所拥有的内存大小。但是在实际应用中，这并不是一个严重的问题。Master服务器只需要不到64个字节的元数据就能够管理一个64MB的Chunk。由于大多数文件都包含多个 Chunk，因此绝大多数Chunk都是满的，除了文件的最后一个Chunk是部分填充的。同样的，每个文件的在命名空间中的数据大小通常在64字节以 下，因为保存的文件名是用前缀压缩算法压缩过的。

即便是需要支持更大的文件系统，为Master服务器增加额外内存的费用是很少的，而通过增加有限的费用，我们就能够把元数据全部保存在内存里，增强了系统的简洁性、可靠性、高性能和灵活性。

Master服务器并不保存持久化保存哪个Chunk服务器存有指定Chunk的副本的信息。Master服务器只是在启动的时候轮询Chunk服 务器以获取这些信息。Master服务器能够保证它持有的信息始终是最新的，因为它控制了所有的Chunk位置的分配，而且通过周期性的心跳信息监控 Chunk服务器的状态。

最初设计时，我们试图把Chunk的位置信息持久的保存在Master服务器上，但是后来我们发现在启动的时候轮询Chunk服务器，之后定期轮询 更新的方式更简单。这种设计简化了在有Chunk服务器加入集群、离开集群、更名、失效、以及重启的时候，Master服务器和Chunk服务器数据同步 的问题。在一个拥有数百台服务器的集群中，这类事件会频繁的发生。

可以从另外一个角度去理解这个设计决策：只有Chunk服务器才能最终确定一个Chunk是否在它的硬盘上。我们从没有考虑过在Master服务器 上维护一个这些信息的全局视图，因为Chunk服务器的错误可能会导致Chunk自动消失(比如，硬盘损坏了或者无法访问了)，亦或者操作人员可能会重命 名一个Chunk服务器。

操作日志包含了关键的元数据变更历史记录。这对GFS非常重要。这不仅仅是因为操作日志是元数据唯一的持久化存储记录，它也作为判断同步操作顺序的逻辑时间基线。文件和Chunk，连同它们的版本，都由它们创建的逻辑时间唯一的、永久的标识。

操作日志非常重要，我们必须确保日志文件的完整，确保只有在元数据的变化被持久化后，日志才对客户端是可见的。否则，即使Chunk本身没有出现任 何问题，我们仍有可能丢失整个文件系统，或者丢失客户端最近的操作。所以，我们会把日志复制到多台远程机器，并且只有把相应的日志记录写入到本地以及远程 机器的硬盘后，才会响应客户端的操作请求。Master服务器会收集多个日志记录后批量处理，以减少写入磁盘和复制对系统整体性能的影响。

Master服务器在灾难恢复时，通过重演操作日志把文件系统恢复到最近的状态。为了缩短Master启动的时间，我们必须使日志足够小。Master服务器在日志增长到一定量时对系统状态做一次Checkpoint，将所有的状态数据写入一个Checkpoint文件。 在灾难恢复的时候，Master服务器就通过从磁盘上读取这个Checkpoint文件，以及重演Checkpoint之后的有限个日志文件就能够恢复系统。Checkpoint文件以压缩B-树形式的数据结构存储，可以直接映射到内存，在用于命名空间查询时无需额外的解析。这大大提高了恢复速度，增强了可用性。

由于创建一个Checkpoint文件需要一定的时间，所以Master服务器的内部状态被组织为一种格式，这种格式要确保在Checkpoint 过程中不会阻塞正在进行的修改操作。Master服务器使用独立的线程切换到新的日志文件和创建新的Checkpoint文件。新的Checkpoint 文件包括切换前所有的修改。对于一个包含数百万个文件的集群，创建一个Checkpoint文件需要1分钟左右的时间。创建完成后，Checkpoint 文件会被写入在本地和远程的硬盘里。

Master服务器恢复只需要最新的Checkpoint文件和后续的日志文件。旧的Checkpoint文件和日志文件可以被删除，但是为了应对灾难性的故障，我们通常会多保存一些历史文件。Checkpoint失败不会对正确性产生任何影响，因为恢复功能的代码可以检测并跳过没有完成的Checkpoint文件。

GFS支持一个宽松的一致性模型，这个模型能够很好的支撑我们的高度分布的应用，同时还保持了相对简单且容易实现的优点。本节我们讨论GFS的一致 性的保障机制，以及对应用程序的意义。我们也着重描述了GFS如何管理这些一致性保障机制，但是实现的细节将在本论文的其它部分讨论。

文件命名空间的修改（例如，文件创建）是原子性的。它们仅由Master节点的控制：命名空间锁提供了原子性和正确性的保障；Master节点的操作日志定义了这些操作在全局的顺序。

**附录B 原文**

**The Google File Systema**

**Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung**

**Google**

**ABSTRACT**

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

**Categories and Subject Descriptors**

D [**4**]: 3—*Distributed file systems*

**General Terms**

Design, reliability, performance, measurement

**Keywords**

Fault tolerance, scalability, data storage, clustered storage



∗The authors can be reached at the following addresses: {*sanjay,hgobioff,shuntak*}*@google.com*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*SOSP’03,* October 19–22, 2003, Bolton Landing, New York, USA.

Copyright 2003 ACM 1-58113-757-5/03/0010 ...$5.00.

**INTRODUCTION**

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google’s data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large repositories that data analysis programs scan through. Some may be data streams continuously generated by running applications. Some may be archival data. Some may be intermediate results produced on one machine and processed on another, whether simultaneously or later in time. Given this access pattern on huge files, appending becomes the focus of performance optimization and atomicity guarantees, while caching data blocks in the client loses its appeal.

Fourth, co-designing the applications and the file system API benefits the overall system by increasing our flexibility.

For example, we have relaxed GFS’s consistency model to vastly simplify the file system without imposing an onerous burden on the applications. We have also introduced an atomic append operation so that multiple clients can append concurrently to a file without extra synchronization between them. These will be discussed in more details later in the paper.

Multiple GFS clusters are currently deployed for different purposes. The largest ones have over 1000 storage nodes, over 300 TB of disk storage, and are heavily accessed by hundreds of clients on distinct machines on a continuous basis.

**DESIGN OVERVIEW**

**Assumptions**

In designing a file system for our needs, we have been guided by assumptions that offer both challenges and opportunities. We alluded to some key observations earlier and now lay out our assumptions in more details.

* The system is built from many inexpensive commodity components that often fail. It must constantly monitor itself and detect, tolerate, and recover promptly from component failures on a routine basis.
* The system stores a modest number of large files. We expect a few million files, each typically 100 MB or larger in size. Multi-GB files are the common case and should be managed efficiently. Small files must be supported, but we need not optimize for them.
* The workloads primarily consist of two kinds of reads: large streaming reads and small random reads. In large streaming reads, individual operations typically read hundreds of KBs, more commonly 1 MB or more. Successive operations from the same client often read through a contiguous region of a file. A small random read typically reads a few KBs at some arbitrary offset. Performance-conscious applications often batch and sort their small reads to advance steadily through the file rather than go back and forth.
* The workloads also have many large, sequential writes that append data to files. Typical operation sizes are similar to those for reads. Once written, files are seldom modified again. Small writes at arbitrary positions in a file are supported but do not have to be efficient.
* The system must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file. Our files are often used as producerconsumer queues or for many-way merging. Hundreds of producers, running one per machine, will concurrently append to a file. Atomicity with minimal synchronization overhead is essential. The file may be read later, or a consumer may be reading through the file simultaneously.
* High sustained bandwidth is more important than low latency. Most of our target applications place a premium on processing data in bulk at a high rate, while few have stringent response time requirements for an individual read or write.

**Interface**

GFS provides a familiar file system interface, though it does not implement a standard API such as POSIX. Files are organized hierarchically in directories and identified by pathnames. We support the usual operations to *create*, *delete*, *open*, *close*, *read*, and *write* files.

Moreover, GFS has *snapshot* and *record append* operations. Snapshot creates a copy of a file or a directory tree at low cost. Record append allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individual client’s append. It is useful for implementing multi-way merge results and producerconsumer queues that many clients can simultaneously append to without additional locking. We have found these types of files to be invaluable in building large distributed applications. Snapshot and record append are discussed further in Sections 3.4 and 3.3 respectively.

**Architecture**

A GFS cluster consists of a single *master* and multiple *chunkservers* and is accessed by multiple *clients*, as shown in Figure 1. Each of these is typically a commodity Linux machine running a user-level server process. It is easy to run both a chunkserver and a client on the same machine, as long as machine resources permit and the lower reliability caused by running possibly flaky application code is acceptable.

Files are divided into fixed-size *chunks*. Each chunk is identified by an immutable and globally unique 64 bit *chunk handle* assigned by the master at the time of chunk creation. Chunkservers store chunks on local disks as Linux files and read or write chunk data specified by a chunk handle and byte range. For reliability, each chunk is replicated on multiple chunkservers. By default, we store three replicas, though users can designate different replication levels for different regions of the file namespace.

The master maintains all file system metadata. This includes the namespace, access control information, the mapping from files to chunks, and the current locations of chunks. It also controls system-wide activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers. The master periodically communicates with each chunkserver in *HeartBeat* messages to give it instructions and collect its state.

GFS client code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the application. Clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunkservers. We do not provide the POSIX API and therefore need not hook into the Linux vnode layer.

Neither the client nor the chunkserver caches file data. Client caches offer little benefit because most applications stream through huge files or have working sets too large to be cached. Not having them simplifies the client and the overall system by eliminating cache coherence issues. (Clients do cache metadata, however.) Chunkservers need not cache file data because chunks are stored as local files and so Linux’s buffer cache already keeps frequently accessed data in memory.

|  |
| --- |
| Legend:  Data messages  Control messages  Application  (  file name, chunk index  )  (  chunk handle,  chunk locations)  **GFS master**  File namespace  /foo/bar  Instructions to chunkserver  Chunkserver state  **GFS chunkserver**  **GFS chunkserver**  (  chunk handle, byte range  )  chunk data  chunk 2ef0  Linux file system  Linux file system  **GFS client**  **Figure 1: GFS Architecture** |

**Single Master**

Having a single master vastly simplifies our design and enables the master to make sophisticated chunk placement and replication decisions using global knowledge. However, we must minimize its involvement in reads and writes so that it does not become a bottleneck. Clients never read and write file data through the master. Instead, a client asks the master which chunkservers it should contact. It caches this information for a limited time and interacts with the chunkservers directly for many subsequent operations.

Let us explain the interactions for a simple read with reference to Figure 1. First, using the fixed chunk size, the client translates the file name and byte offset specified by the application into a chunk index within the file. Then, it sends the master a request containing the file name and chunk index. The master replies with the corresponding chunk handle and locations of the replicas. The client caches this information using the file name and chunk index as the key. The client then sends a request to one of the replicas, most likely the closest one. The request specifies the chunk handle and a byte range within that chunk. Further reads of the same chunk require no more client-master interaction until the cached information expires or the file is reopened. In fact, the client typically asks for multiple chunks in the same request and the master can also include the information for chunks immediately following those requested. This extra information sidesteps several future client-master interactions at practically no extra cost.

**Chunk Size**

Chunk size is one of the key design parameters. We have chosen 64 MB, which is much larger than typical file system block sizes. Each chunk replica is stored as a plain Linux file on a chunkserver and is extended only as needed. Lazy space allocation avoids wasting space due to internal fragmentation, perhaps the greatest objection against such a large chunk size.

A large chunk size offers several important advantages. First, it reduces clients’ need to interact with the master because reads and writes on the same chunk require only one initial request to the master for chunk location information. The reduction is especially significant for our workloads because applications mostly read and write large files sequentially. Even for small random reads, the client can comfortably cache all the chunk location information for a multi-TB working set. Second, since on a large chunk, a client is more likely to perform many operations on a given chunk, it can reduce network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time. Third, it reduces the size of the metadata stored on the master. This allows us to keep the metadata in memory, which in turn brings other advantages that we will discuss in Section 2.6.1.

On the other hand, a large chunk size, even with lazy space allocation, has its disadvantages. A small file consists of a small number of chunks, perhaps just one. The chunkservers storing those chunks may become hot spots if many clients are accessing the same file. In practice, hot spots have not been a major issue because our applications mostly read large multi-chunk files sequentially.

However, hot spots did develop when GFS was first used by a batch-queue system: an executable was written to GFS as a single-chunk file and then started on hundreds of machines at the same time. The few chunkservers storing this executable were overloaded by hundreds of simultaneous requests. We fixed this problem by storing such executables with a higher replication factor and by making the batchqueue system stagger application start times. A potential long-term solution is to allow clients to read data from other clients in such situations.

**Metadata**

The master stores three major types of metadata: the file and chunk namespaces, the mapping from files to chunks, and the locations of each chunk’s replicas. All metadata is kept in the master’s memory. The first two types (namespaces and file-to-chunk mapping) are also kept persistent by logging mutations to an *operation log* stored on the master’s local disk and replicated on remote machines. Using a log allows us to update the master state simply, reliably, and without risking inconsistencies in the event of a master crash. The master does not store chunk location information persistently. Instead, it asks each chunkserver about its chunks at master startup and whenever a chunkserver joins the cluster.

*In-Memory Data Structures*

Since metadata is stored in memory, master operations are fast. Furthermore, it is easy and efficient for the master to periodically scan through its entire state in the background. This periodic scanning is used to implement chunk garbage collection, re-replication in the presence of chunkserver failures, and chunk migration to balance load and disk space usage across chunkservers. Sections 4.3 and 4.4 will discuss these activities further.

One potential concern for this memory-only approach is that the number of chunks and hence the capacity of the whole system is limited by how much memory the master has. This is not a serious limitation in practice. The master maintains less than 64 bytes of metadata for each 64 MB chunk. Most chunks are full because most files contain many chunks, only the last of which may be partially filled. Similarly, the file namespace data typically requires less then 64 bytes per file because it stores file names compactly using prefix compression.

If necessary to support even larger file systems, the cost of adding extra memory to the master is a small price to pay for the simplicity, reliability, performance, and flexibility we gain by storing the metadata in memory.

*Chunk Locations*

The master does not keep a persistent record of which chunkservers have a replica of a given chunk. It simply polls chunkservers for that information at startup. The master can keep itself up-to-date thereafter because it controls all chunk placement and monitors chunkserver status with regular *HeartBeat* messages.

We initially attempted to keep chunk location information persistently at the master, but we decided that it was much simpler to request the data from chunkservers at startup, and periodically thereafter. This eliminated the problem of keeping the master and chunkservers in sync as chunkservers join and leave the cluster, change names, fail, restart, and so on. In a cluster with hundreds of servers, these events happen all too often.

Another way to understand this design decision is to realize that a chunkserver has the final word over what chunks it does or does not have on its own disks. There is no point in trying to maintain a consistent view of this information on the master because errors on a chunkserver may cause chunks to vanish spontaneously (e.g., a disk may go bad and be disabled) or an operator may rename a chunkserver.

*Operation Log*

The operation log contains a historical record of critical metadata changes. It is central to GFS. Not only is it the only persistent record of metadata, but it also serves as a logical time line that defines the order of concurrent operations. Files and chunks, as well as their versions (see Section 4.5), are all uniquely and eternally identified by the logical times at which they were created.

Since the operation log is critical, we must store it reliably and not make changes visible to clients until metadata changes are made persistent. Otherwise, we effectively lose the whole file system or recent client operations even if the chunks themselves survive. Therefore, we replicate it on multiple remote machines and respond to a client operation only after flushing the corresponding log record to disk both locally and remotely. The master batches several log records together before flushing thereby reducing the impact of flushing and replication on overall system throughput.

|  |  |  |
| --- | --- | --- |
|  | Write | Record Append |
| Serial | *defined* | *defined* |
| success |  | interspersed with *inconsistent* |
| Concurrent | *consistent* |
| successes | but *undefined* |  |
| Failure | *inconsistent* | |

**Table 1: File Region State After Mutation**

The master recovers its file system state by replaying the operation log. To minimize startup time, we must keep the log small. The master checkpoints its state whenever the log grows beyond a certain size so that it can recover by loading the latest checkpoint from local disk and replaying only the limited number of log records after that. The checkpoint is in a compact B-tree like form that can be directly mapped into memory and used for namespace lookup without extra parsing. This further speeds up recovery and improves availability.

Because building a checkpoint can take a while, the master’s internal state is structured in such a way that a new checkpoint can be created without delaying incoming mutations. The master switches to a new log file and creates the new checkpoint in a separate thread. The new checkpoint includes all mutations before the switch. It can be created in a minute or so for a cluster with a few million files. When completed, it is written to disk both locally and remotely.

Recovery needs only the latest complete checkpoint and subsequent log files. Older checkpoints and log files can be freely deleted, though we keep a few around to guard against catastrophes. A failure during checkpointing does not affect correctness because the recovery code detects and skips incomplete checkpoints.

**Consistency Model**

GFS has a relaxed consistency model that supports our highly distributed applications well but remains relatively simple and efficient to implement. We now discuss GFS’s guarantees and what they mean to applications. We also highlight how GFS maintains these guarantees but leave the details to other parts of the paper.

*Guarantees by GFS*

File namespace mutations (e.g., file creation) are atomic. They are handled exclusively by the master: namespace locking guarantees atomicity and correctness (Section 4.1) ; the master’s operation log defines a global total order of these operations (Section 2.6.3).

The state of a file region after a data mutation depends on the type of mutation, whether it succeeds or fails, and whether there are concurrent mutations. Table 1 summarizes the result. A file region is *consistent* if all clients will always see the same data, regardless of which replicas they read from. A region is *defined* after a file data mutation if it is consistent and clients will see what the mutation writes in its entirety. When a mutation succeeds without interference from concurrent writers, the affected region is defined ( and by implication consistent): all clients will always see what the mutation has written. Concurrent successful mutations leave the region undefined but consistent: all clients see the same data, but it may not reflect what any one mutation has written. Typically, it consists of mingled fragments from multiple mutations. A failed mutation makes the region inconsistent (hence also undefined): different clients may see different data at different times. We describe below how our applications can distinguish defined regions from undefined regions. The applications do not need to further distinguish between different kinds of undefined regions.

Data mutations may be *writes* or *record appends*. A write causes data to be written at an application-specified file offset. A record append causes data (the “record”) to be appended *atomically at least once* even in the presence of concurrent mutations, but at an offset of GFS’s choosing (Section 3.3). (In contrast, a “regular” append is merely a write at an offset that the client believes to be the current end of file.) The offset is returned to the client and marks the beginning of a defined region that contains the record. In addition, GFS may insert padding or record duplicates in between. They occupy regions considered to be inconsistent and are typically dwarfed by the amount of user data.

After a sequence of successful mutations, the mutated file region is guaranteed to be defined and contain the data written by the last mutation. GFS achieves this by (a) applying mutations to a chunk in the same order on all its replicas (Section 3.1), and (b) using chunk version numbers to detect any replica that has become stale because it has missed mutations while its chunkserver was down (Section 4.5). Stale replicas will never be involved in a mutation or given to clients asking the master for chunk locations. They are garbage collected at the earliest opportunity.

Since clients cache chunk locations, they may read from a stale replica before that information is refreshed. This window is limited by the cache entry’s timeout and the next open of the file, which purges from the cache all chunk information for that file. Moreover, as most of our files are append-only, a stale replica usually returns a premature end of chunk rather than outdated data. When a reader retries and contacts the master, it will immediately get current chunk locations.

Long after a successful mutation, component failures can of course still corrupt or destroy data. GFS identifies failed chunkservers by regular handshakes between master and all chunkservers and detects data corruption by checksumming (Section 5.2). Once a problem surfaces, the data is restored from valid replicas as soon as possible (Section 4.3). A chunk is lost irreversibly only if all its replicas are lost before GFS can react, typically within minutes. Even in this case, it becomes unavailable, not corrupted: applications receive clear errors rather than corrupt data.

*Implications for Applications*

GFS applications can accommodate the relaxed consistency model with a few simple techniques already needed for other purposes: relying on appends rather than overwrites, checkpointing, and writing self-validating, self-identifying records.

Practically all our applications mutate files by appending rather than overwriting. In one typical use, a writer generates a file from beginning to end. It atomically renames the file to a permanent name after writing all the data, or periodically checkpoints how much has been successfully written. Checkpoints may also include application-level checksums. Readers verify and process only the file region up to the last checkpoint, which is known to be in the defined state. Regardless of consistency and concurrency issues, this approach has served us well. Appending is far more efficient and more resilient to application failures than random writes. Checkpointing allows writers to restart incrementally and keeps readers from processing successfully written file data that is still incomplete from the application’s perspective.

In the other typical use, many writers concurrently append to a file for merged results or as a producer-consumer queue. Record append’s append-at-least-once semantics preserves each writer’s output. Readers deal with the occasional padding and duplicates as follows. Each record prepared by the writer contains extra information like checksums so that its validity can be verified. A reader can identify and discard extra padding and record fragments using the checksums. If it cannot tolerate the occasional duplicates (e.g., if they would trigger non-idempotent operations), it can filter them out using unique identifiers in the records, which are often needed anyway to name corresponding application entities such as web documents. These functionalities for record I/O (except duplicate removal) are in library code shared by our applications and applicable to other file interface implementations at Google. With that, the same sequence of records, plus rare duplicates, is always delivered to the record reader.

**SYSTEM INTERACTIONS**

We designed the system to minimize the master’s involvement in all operations. With that background, we now describe how the client, master, and chunkservers interact to implement data mutations, atomic record append, and snapshot.

**Leases and Mutation Order**

A mutation is an operation that changes the contents or metadata of a chunk such as a write or an append operation. Each mutation is performed at all the chunk’s replicas. We use leases to maintain a consistent mutation order across replicas. The master grants a chunk lease to one of the replicas, which we call the *primary*. The primary picks a serial order for all mutations to the chunk. All replicas follow this order when applying mutations. Thus, the global mutation order is defined first by the lease grant order chosen by the master, and within a lease by the serial numbers assigned by the primary.

The lease mechanism is designed to minimize management overhead at the master. A lease has an initial timeout of 60 seconds. However, as long as the chunk is being mutated, the primary can request and typically receive extensions from the master indefinitely. These extension requests and grants are piggybacked on the *HeartBeat* messages regularly exchanged between the master and all chunkservers. The master may sometimes try to revoke a lease before it expires (e.g., when the master wants to disable mutations on a file that is being renamed). Even if the master loses communication with a primary, it can safely grant a new lease to another replica after the old lease expires.

Primary

Replica

Secondary

Replica B

Secondary

Replica A

Master

Legend:

Control

Data

3

Client

2

step 1

4

5

6

6

7

**Figure 2: Write Control and Data Flow**

In Figure 2, we illustrate this process by following the control flow of a write through these numbered steps.

1. The client asks the master which chunkserver holds the current lease for the chunk and the locations of the other replicas. If no one has a lease, the master grants one to a replica it chooses (not shown).
2. The master replies with the identity of the primary and the locations of the other (*secondary*) replicas. The client caches this data for future mutations. It needs to contact the master again only when the primary

becomes unreachable or replies that it no longer holds a lease.

1. The client pushes the data to all the replicas. A client can do so in any order. Each chunkserver will store the data in an internal LRU buffer cache until the data is used or aged out. By decoupling the data flow from the control flow, we can improve performance by scheduling the expensive data flow based on the network topology regardless of which chunkserver is the primary. Section 3.2 discusses this further.
2. Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary. The request identifies the data pushed earlier to all of the replicas. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serialization. It applies the mutation to its own local state in serial number order.
3. The primary forwards the write request to all secondary replicas. Each secondary replica applies mutations in the same serial number order assigned by the primary.
4. The secondaries all reply to the primary indicating that they have completed the operation.
5. The primary replies to the client. Any errors encountered at any of the replicas are reported to the client. In case of errors, the write may have succeeded at the primary and an arbitrary subset of the secondary replicas. (If it had failed at the primary, it would not have been assigned a serial number and forwarded.) The client request is considered to have failed, and the modified region is left in an inconsistent state. Our client code handles such errors by retrying the failed mutation. It will make a few attempts at steps (3) through (7) before falling back to a retry from the beginning of the write.

If a write by the application is large or straddles a chunk boundary, GFS client code breaks it down into multiple write operations. They all follow the control flow described above but may be interleaved with and overwritten by concurrent operations from other clients. Therefore, the shared file region may end up containing fragments from different clients, although the replicas will be identical because the individual operations are completed successfully in the same order on all replicas. This leaves the file region in consistent but undefined state as noted in Section 2.7.

**Data Flow**

We decouple the flow of data from the flow of control to use the network efficiently. While control flows from the client to the primary and then to all secondaries, data is pushed linearly along a carefully picked chain of chunkservers in a pipelined fashion. Our goals are to fully utilize each machine’s network bandwidth, avoid network bottlenecks and high-latency links, and minimize the latency to push through all the data.

To fully utilize each machine’s network bandwidth, the data is pushed linearly along a chain of chunkservers rather than distributed in some other topology (e.g., tree). Thus, each machine’s full outbound bandwidth is used to transfer the data as fast as possible rather than divided among multiple recipients.

To avoid network bottlenecks and high-latency links ( e.g., inter-switch links are often both) as much as possible, each machine forwards the data to the “closest” machine in the network topology that has not received it. Suppose the client is pushing data to chunkservers S1 through S4. It sends the data to the closest chunkserver, say S1. S1 forwards it to the closest chunkserver S2 through S4 closest to S1, say S2. Similarly, S2 forwards it to S3 or S4, whichever is closer to S2, and so on. Our network topology is simple enough that “distances” can be accurately estimated from IP addresses.

Finally, we minimize latency by pipelining the data transfer over TCP connections. Once a chunkserver receives some data, it starts forwarding immediately. Pipelining is especially helpful to us because we use a switched network with full-duplex links. Sending the data immediately does not reduce the receive rate. Without network congestion, the ideal elapsed time for transferring *B* bytes to *R* replicas is *B/T* + *RL* where *T* is the network throughput and *L* is latency to transfer bytes between two machines. Our network links are typically 100 Mbps (*T*), and *L* is far below 1 ms.

Therefore, 1 MB can ideally be distributed in about 80 ms.

**Atomic Record Appends**

GFS provides an atomic append operation called *record append*. In a traditional write, the client specifies the offset at which data is to be written. Concurrent writes to the same region are not serializable: the region may end up containing data fragments from multiple clients. In a record append, however, the client specifies only the data. GFS appends it to the file at least once atomically (i.e., as one continuous sequence of bytes) at an offset of GFS’s choosing and returns that offset to the client. This is similar to writing to a file opened in O APPEND mode in Unix without the race conditions when multiple writers do so concurrently.

Record append is heavily used by our distributed applications in which many clients on different machines append to the same file concurrently. Clients would need additional complicated and expensive synchronization, for example through a distributed lock manager, if they do so with traditional writes. In our workloads, such files often serve as multiple-producer/single-consumer queues or contain merged results from many different clients.

Record append is a kind of mutation and follows the control flow in Section 3.1 with only a little extra logic at the primary. The client pushes the data to all replicas of the last chunk of the file Then, it sends its request to the primary. The primary checks to see if appending the record to the current chunk would cause the chunk to exceed the maximum size (64 MB). If so, it pads the chunk to the maximum size, tells secondaries to do the same, and replies to the client indicating that the operation should be retried on the next chunk. (Record append is restricted to be at most one-fourth of the maximum chunk size to keep worstcase fragmentation at an acceptable level.) If the record fits within the maximum size, which is the common case, the primary appends the data to its replica, tells the secondaries to write the data at the exact offset where it has, and finally replies success to the client.

If a record append fails at any replica, the client retries the operation. As a result, replicas of the same chunk may contain different data possibly including duplicates of the same record in whole or in part. GFS does not guarantee that all replicas are bytewise identical. It only guarantees that the data is written at least once as an atomic unit. This property follows readily from the simple observation that for the operation to report success, the data must have been written at the same offset on all replicas of some chunk. Furthermore, after this, all replicas are at least as long as the end of record and therefore any future record will be assigned a higher offset or a different chunk even if a different replica later becomes the primary. In terms of our consistency guarantees, the regions in which successful record append operations have written their data are defined (hence consistent), whereas intervening regions are inconsistent (hence undefined). Our applications can deal with inconsistent regions as we discussed in Section 2.7.2.

**Snapshot**

The snapshot operation makes a copy of a file or a directory tree (the “source”) almost instantaneously, while minimizing any interruptions of ongoing mutations. Our users use it to quickly create branch copies of huge data sets (and often copies of those copies, recursively), or to checkpoint the current state before experimenting with changes that can later be committed or rolled back easily.

Like AFS [5], we use standard copy-on-write techniques to implement snapshots. When the master receives a snapshot request, it first revokes any outstanding leases on the chunks in the files it is about to snapshot. This ensures that any subsequent writes to these chunks will require an interaction with the master to find the lease holder. This will give the master an opportunity to create a new copy of the chunk first.

After the leases have been revoked or have expired, the master logs the operation to disk. It then applies this log record to its in-memory state by duplicating the metadata for the source file or directory tree. The newly created snapshot files point to the same chunks as the source files.

The first time a client wants to write to a chunk C after the snapshot operation, it sends a request to the master to find the current lease holder. The master notices that the reference count for chunk C is greater than one. It defers replying to the client request and instead picks a new chunk handle C’. It then asks each chunkserver that has a current replica of C to create a new chunk called C’. By creating the new chunk on the same chunkservers as the original, we ensure that the data can be copied locally, not over the network (our disks are about three times as fast as our 100 Mb Ethernet links). From this point, request handling is no different from that for any chunk: the master grants one of the replicas a lease on the new chunk C’ and replies to the client, which can write the chunk normally, not knowing that it has just been created from an existing chunk.

**MASTER OPERATION**

The master executes all namespace operations. In addition, it manages chunk replicas throughout the system: it makes placement decisions, creates new chunks and hence replicas, and coordinates various system-wide activities to keep chunks fully replicated, to balance load across all the chunkservers, and to reclaim unused storage. We now discuss each of these topics.

**Namespace Management and Locking**

Many master operations can take a long time: for example, a snapshot operation has to revoke chunkserver leases on all chunks covered by the snapshot. We do not want to delay other master operations while they are running. Therefore, we allow multiple operations to be active and use locks over regions of the namespace to ensure proper serialization.

Unlike many traditional file systems, GFS does not have a per-directory data structure that lists all the files in that directory. Nor does it support aliases for the same file or directory (i.e, hard or symbolic links in Unix terms). GFS logically represents its namespace as a lookup table mapping full pathnames to metadata. With prefix compression, this table can be efficiently represented in memory. Each node in the namespace tree (either an absolute file name or an absolute directory name) has an associated read-write lock. Each master operation acquires a set of locks before it runs. Typically, if it involves /d1/d2/.../dn/leaf, it will acquire read-locks on the directory names /d1, /d1/d2, ..., /d1/d2/.../dn, and either a read lock or a write lock on the full pathname /d1/d2/.../dn/leaf. Note that leaf may be a file or directory depending on the operation.

We now illustrate how this locking mechanism can prevent a file /home/user/foo from being created while /home/user is being snapshotted to /save/user. The snapshot operation acquires read locks on /home and /save, and write locks on /home/user and /save/user. The file creation acquires read locks on /home and /home/user, and a write lock on /home/user/foo. The two operations will be serialized properly because they try to obtain conflicting locks on /home/user. File creation does not require a write lock on the parent directory because there is no “directory”, or *inode*-like, data structure to be protected from modification. The read lock on the name is sufficient to protect the parent directory from deletion.

One nice property of this locking scheme is that it allows concurrent mutations in the same directory. For example, multiple file creations can be executed concurrently in the same directory: each acquires a read lock on the directory name and a write lock on the file name. The read lock on the directory name suffices to prevent the directory from being deleted, renamed, or snapshotted. The write locks on file names serialize attempts to create a file with the same name twice.

Since the namespace can have many nodes, read-write lock objects are allocated lazily and deleted once they are not in use. Also, locks are acquired in a consistent total order to prevent deadlock: they are first ordered by level in the namespace tree and lexicographically within the same level.

**Replica Placement**

A GFS cluster is highly distributed at more levels than one. It typically has hundreds of chunkservers spread across many machine racks. These chunkservers in turn may be accessed from hundreds of clients from the same or different racks. Communication between two machines on different racks may cross one or more network switches. Additionally, bandwidth into or out of a rack may be less than the aggregate bandwidth of all the machines within the rack. Multi-level distribution presents a unique challenge to distribute data for scalability, reliability, and availability.

The chunk replica placement policy serves two purposes: maximize data reliability and availability, and maximize network bandwidth utilization. For both, it is not enough to spread replicas across machines, which only guards against disk or machine failures and fully utilizes each machine’s network bandwidth. We must also spread chunk replicas across racks. This ensures that some replicas of a chunk will survive and remain available even if an entire rack is damaged or offline (for example, due to failure of a shared resource like a network switch or power circuit). It also means that traffic, especially reads, for a chunk can exploit the aggregate bandwidth of multiple racks. On the other hand, write traffic has to flow through multiple racks, a tradeoff we make willingly.

**Creation, Re-replication, Rebalancing**

Chunk replicas are created for three reasons: chunk creation, re-replication, and rebalancing.

When the master *creates* a chunk, it chooses where to place the initially empty replicas. It considers several factors. (1) We want to place new replicas on chunkservers with below-average disk space utilization. Over time this will equalize disk utilization across chunkservers. (2) We want to limit the number of “recent” creations on each chunkserver. Although creation itself is cheap, it reliably predicts imminent heavy write traffic because chunks are created when demanded by writes, and in our append-once-read-many workload they typically become practically read-only once they have been completely written. (3) As discussed above, we want to spread replicas of a chunk across racks.

The master *re-replicates* a chunk as soon as the number of available replicas falls below a user-specified goal. This could happen for various reasons: a chunkserver becomes unavailable, it reports that its replica may be corrupted, one of its disks is disabled because of errors, or the replication goal is increased. Each chunk that needs to be re-replicated is prioritized based on several factors. One is how far it is from its replication goal. For example, we give higher priority to a chunk that has lost two replicas than to a chunk that has lost only one. In addition, we prefer to first re-replicate chunks for live files as opposed to chunks that belong to recently deleted files (see Section 4.4). Finally, to minimize the impact of failures on running applications, we boost the priority of any chunk that is blocking client progress.

The master picks the highest priority chunk and “clones” it by instructing some chunkserver to copy the chunk data directly from an existing valid replica. The new replica is placed with goals similar to those for creation: equalizing disk space utilization, limiting active clone operations on any single chunkserver, and spreading replicas across racks. To keep cloning traffic from overwhelming client traffic, the master limits the numbers of active clone operations both for the cluster and for each chunkserver. Additionally, each chunkserver limits the amount of bandwidth it spends on each clone operation by throttling its read requests to the source chunkserver.

Finally, the master *rebalances* replicas periodically: it examines the current replica distribution and moves replicas for better disk space and load balancing. Also through this process, the master gradually fills up a new chunkserver rather than instantly swamps it with new chunks and the heavy write traffic that comes with them. The placement criteria for the new replica are similar to those discussed above. In addition, the master must also choose which existing replica to remove. In general, it prefers to remove those on chunkservers with below-average free space so as to equalize disk space usage.

**Garbage Collection**

After a file is deleted, GFS does not immediately reclaim the available physical storage. It does so only lazily during regular garbage collection at both the file and chunk levels. We find that this approach makes the system much simpler and more reliable.

*Mechanism*

When a file is deleted by the application, the master logs the deletion immediately just like other changes. However instead of reclaiming resources immediately, the file is just renamed to a hidden name that includes the deletion timestamp. During the master’s regular scan of the file system namespace, it removes any such hidden files if they have existed for more than three days (the interval is configurable). Until then, the file can still be read under the new, special name and can be undeleted by renaming it back to normal. When the hidden file is removed from the namespace, its inmemory metadata is erased. This effectively severs its links to all its chunks.

In a similar regular scan of the chunk namespace, the master identifies orphaned chunks (i.e., those not reachable from any file) and erases the metadata for those chunks. In a *HeartBeat* message regularly exchanged with the master, each chunkserver reports a subset of the chunks it has, and the master replies with the identity of all chunks that are no longer present in the master’s metadata. The chunkserver is free to delete its replicas of such chunks.

*Discussion*

Although distributed garbage collection is a hard problem that demands complicated solutions in the context of programming languages, it is quite simple in our case. We can easily identify all references to chunks: they are in the fileto-chunk mappings maintained exclusively by the master. We can also easily identify all the chunk replicas: they are Linux files under designated directories on each chunkserver.

Any such replica not known to the master is “garbage.” The garbage collection approach to storage reclamation offers several advantages over eager deletion. First, it is simple and reliable in a large-scale distributed system where component failures are common. Chunk creation may succeed on some chunkservers but not others, leaving replicas that the master does not know exist. Replica deletion messages may be lost, and the master has to remember to resend them across failures, both its own and the chunkserver’s. Garbage collection provides a uniform and dependable way to clean up any replicas not known to be useful. Second, it merges storage reclamation into the regular background activities of the master, such as the regular scans of namespaces and handshakes with chunkservers. Thus, it is done in batches and the cost is amortized. Moreover, it is done only when the master is relatively free. The master can respond more promptly to client requests that demand timely attention. Third, the delay in reclaiming storage provides a safety net against accidental, irreversible deletion.

In our experience, the main disadvantage is that the delay sometimes hinders user effort to fine tune usage when storage is tight. Applications that repeatedly create and delete temporary files may not be able to reuse the storage right away. We address these issues by expediting storage reclamation if a deleted file is explicitly deleted again. We also allow users to apply different replication and reclamation policies to different parts of the namespace. For example, users can specify that all the chunks in the files within some directory tree are to be stored without replication, and any deleted files are immediately and irrevocably removed from the file system state.

**Stale Replica Detection**

Chunk replicas may become stale if a chunkserver fails and misses mutations to the chunk while it is down. For each chunk, the master maintains a *chunk version number* to distinguish between up-to-date and stale replicas.

Whenever the master grants a new lease on a chunk, it increases the chunk version number and informs the up-todate replicas. The master and these replicas all record the new version number in their persistent state. This occurs before any client is notified and therefore before it can start writing to the chunk. If another replica is currently unavailable, its chunk version number will not be advanced. The master will detect that this chunkserver has a stale replica when the chunkserver restarts and reports its set of chunks and their associated version numbers. If the master sees a version number greater than the one in its records, the master assumes that it failed when granting the lease and so takes the higher version to be up-to-date.

The master removes stale replicas in its regular garbage collection. Before that, it effectively considers a stale replica not to exist at all when it replies to client requests for chunk information. As another safeguard, the master includes the chunk version number when it informs clients which chunkserver holds a lease on a chunk or when it instructs a chunkserver to read the chunk from another chunkserver in a cloning operation. The client or the chunkserver verifies the version number when it performs the operation so that it is always accessing up-to-date data.

**FAULT TOLERANCE AND DIAGNOSIS**

One of our greatest challenges in designing the system is dealing with frequent component failures. The quality and quantity of components together make these problems more the norm than the exception: we cannot completely trust the machines, nor can we completely trust the disks. Component failures can result in an unavailable system or, worse, corrupted data. We discuss how we meet these challenges and the tools we have built into the system to diagnose problems when they inevitably occur.

**High Availability**

Among hundreds of servers in a GFS cluster, some are bound to be unavailable at any given time. We keep the overall system highly available with two simple yet effective strategies: fast recovery and replication.

*Fast Recovery*

Both the master and the chunkserver are designed to restore their state and start in seconds no matter how they terminated. In fact, we do not distinguish between normal and abnormal termination; servers are routinely shut down just by killing the process. Clients and other servers experience a minor hiccup as they time out on their outstanding requests, reconnect to the restarted server, and retry. Section 6.2.2 reports observed startup times.

*Chunk Replication*

As discussed earlier, each chunk is replicated on multiple chunkservers on different racks. Users can specify different replication levels for different parts of the file namespace. The default is three. The master clones existing replicas as needed to keep each chunk fully replicated as chunkservers go offline or detect corrupted replicas through checksum verification (see Section 5.2). Although replication has served us well, we are exploring other forms of cross-server redundancy such as parity or erasure codes for our increasing readonly storage requirements. We expect that it is challenging but manageable to implement these more complicated redundancy schemes in our very loosely coupled system because our traffic is dominated by appends and reads rather than small random writes.

*Master Replication*

The master state is replicated for reliability. Its operation log and checkpoints are replicated on multiple machines. A mutation to the state is considered committed only after its log record has been flushed to disk locally and on all master replicas. For simplicity, one master process remains in charge of all mutations as well as background activities such as garbage collection that change the system internally. When it fails, it can restart almost instantly. If its machine or disk fails, monitoring infrastructure outside GFS starts a new master process elsewhere with the replicated operation log. Clients use only the canonical name of the master ( e.g. gfs-test), which is a DNS alias that can be changed if the master is relocated to another machine.

Moreover, “shadow” masters provide read-only access to the file system even when the primary master is down. They are shadows, not mirrors, in that they may lag the primary slightly, typically fractions of a second. They enhance read availability for files that are not being actively mutated or applications that do not mind getting slightly stale results. In fact, since file content is read from chunkservers, applications do not observe stale file content. What could be stale within short windows is file metadata, like directory contents or access control information.

To keep itself informed, a shadow master reads a replica of the growing operation log and applies the same sequence of changes to its data structures exactly as the primary does. Like the primary, it polls chunkservers at startup (and infrequently thereafter) to locate chunk replicas and exchanges frequent handshake messages with them to monitor their status. It depends on the primary master only for replica location updates resulting from the primary’s decisions to create and delete replicas.

**Data Integrity**

Each chunkserver uses checksumming to detect corruption of stored data. Given that a GFS cluster often has thousands of disks on hundreds of machines, it regularly experiences disk failures that cause data corruption or loss on both the read and write paths. (See Section 7 for one cause.) We can recover from corruption using other chunk replicas, but it would be impractical to detect corruption by comparing replicas across chunkservers. Moreover, divergent replicas may be legal: the semantics of GFS mutations, in particular atomic record append as discussed earlier, does not guarantee identical replicas. Therefore, each chunkserver must independently verify the integrity of its own copy by maintaining checksums.

A chunk is broken up into 64 KB blocks. Each has a corresponding 32 bit checksum. Like other metadata, checksums are kept in memory and stored persistently with logging, separate from user data.

For reads, the chunkserver verifies the checksum of data blocks that overlap the read range before returning any data to the requester, whether a client or another chunkserver. Therefore chunkservers will not propagate corruptions to other machines. If a block does not match the recorded checksum, the chunkserver returns an error to the requestor and reports the mismatch to the master. In response, the requestor will read from other replicas, while the master will clone the chunk from another replica. After a valid new replica is in place, the master instructs the chunkserver that reported the mismatch to delete its replica.

Checksumming has little effect on read performance for several reasons. Since most of our reads span at least a few blocks, we need to read and checksum only a relatively small amount of extra data for verification. GFS client code further reduces this overhead by trying to align reads at checksum block boundaries. Moreover, checksum lookups and comparison on the chunkserver are done without any I/O, and checksum calculation can often be overlapped with I/Os.

Checksum computation is heavily optimized for writes that append to the end of a chunk (as opposed to writes that overwrite existing data) because they are dominant in our workloads. We just incrementally update the checksum for the last partial checksum block, and compute new checksums for any brand new checksum blocks filled by the append. Even if the last partial checksum block is already corrupted and we fail to detect it now, the new checksum value will not match the stored data, and the corruption will be detected as usual when the block is next read.

In contrast, if a write overwrites an existing range of the chunk, we must read and verify the first and last blocks of the range being overwritten, then perform the write, and finally compute and record the new checksums. If we do not verify the first and last blocks before overwriting them partially, the new checksums may hide corruption that exists in the regions not being overwritten.

During idle periods, chunkservers can scan and verify the contents of inactive chunks. This allows us to detect corruption in chunks that are rarely read. Once the corruption is detected, the master can create a new uncorrupted replica and delete the corrupted replica. This prevents an inactive but corrupted chunk replica from fooling the master into thinking that it has enough valid replicas of a chunk.

**Diagnostic Tools**

Extensive and detailed diagnostic logging has helped immeasurably in problem isolation, debugging, and performance analysis, while incurring only a minimal cost. Without logs, it is hard to understand transient, non-repeatable interactions between machines. GFS servers generate diagnostic logs that record many significant events (such as chunkservers going up and down) and all RPC requests and replies. These diagnostic logs can be freely deleted without affecting the correctness of the system. However, we try to keep these logs around as far as space permits.

The RPC logs include the exact requests and responses sent on the wire, except for the file data being read or written. By matching requests with replies and collating RPC records on different machines, we can reconstruct the entire interaction history to diagnose a problem. The logs also serve as traces for load testing and performance analysis.

The performance impact of logging is minimal (and far outweighed by the benefits) because these logs are written sequentially and asynchronously. The most recent events are also kept in memory and available for continuous online monitoring.

**MEASUREMENTS**

In this section we present a few micro-benchmarks to illustrate the bottlenecks inherent in the GFS architecture and implementation, and also some numbers from real clusters in use at Google.

**Micro-benchmarks**

We measured performance on a GFS cluster consisting of one master, two master replicas, 16 chunkservers, and 16 clients. Note that this configuration was set up for ease of testing. Typical clusters have hundreds of chunkservers and hundreds of clients.

All the machines are configured with dual 1.4 GHz PIII processors, 2 GB of memory, two 80 GB 5400 rpm disks, and a 100 Mbps full-duplex Ethernet connection to an HP 2524 switch. All 19 GFS server machines are connected to one switch, and all 16 client machines to the other. The two switches are connected with a 1 Gbps link.

*Reads*

*N* clients read simultaneously from the file system. Each client reads a randomly selected 4 MB region from a 320 GB file set. This is repeated 256 times so that each client ends up reading 1 GB of data. The chunkservers taken together have only 32 GB of memory, so we expect at most a 10% hit rate in the Linux buffer cache. Our results should be close to cold cache results.

Figure 3(a) shows the aggregate read rate for *N* clients and its theoretical limit. The limit peaks at an aggregate of 125 MB/s when the 1 Gbps link between the two switches is saturated, or 12.5 MB/s per client when its 100 Mbps network interface gets saturated, whichever applies. The observed read rate is 10 MB/s, or 80% of the per-client limit, when just one client is reading. The aggregate read rate reaches 94 MB/s, about 75% of the 125 MB/s link limit, for 16 readers, or 6 MB/s per client. The efficiency drops from 80% to 75% because as the number of readers increases, so does the probability that multiple readers simultaneously read from the same chunkserver.

*Writes*

*N* clients write simultaneously to *N* distinct files. Each client writes 1 GB of data to a new file in a series of 1 MB writes. The aggregate write rate and its theoretical limit are shown in Figure 3(b). The limit plateaus at 67 MB/s because we need to write each byte to 3 of the 16 chunkservers, each with a 12.5 MB/s input connection.

The write rate for one client is 6.3 MB/s, about half of the limit. The main culprit for this is our network stack. It does not interact very well with the pipelining scheme we use for pushing data to chunk replicas. Delays in propagating data from one replica to another reduce the overall write rate.

Aggregate write rate reaches 35 MB/s for 16 clients (or 2.2 MB/s per client), about half the theoretical limit. As in the case of reads, it becomes more likely that multiple clients write concurrently to the same chunkserver as the number of clients increases. Moreover, collision is more likely for 16 writers than for 16 readers because each write involves three different replicas.

Writes are slower than we would like. In practice this has not been a major problem because even though it increases the latencies as seen by individual clients, it does not significantly affect the aggregate write bandwidth delivered by the system to a large number of clients.

*Record Appends*

Figure 3(c) shows record append performance. *N* clients append simultaneously to a single file. Performance is limited by the network bandwidth of the chunkservers that store the last chunk of the file, independent of the number of clients. It starts at 6.0 MB/s for one client and drops to 4.8 MB/s for 16 clients, mostly due to congestion and variances in network transfer rates seen by different clients. Our applications tend to produce multiple such files concurrently. In other words, *N* clients append to *M* shared files simultaneously where both *N* and *M* are in the dozens or hundreds. Therefore, the chunkserver network congestion in our experiment is not a significant issue in practice because a client can make progress on writing one file while the chunkservers for another file are busy.

**Real World Clusters**

We now examine two clusters in use within Google that are representative of several others like them. Cluster A is used regularly for research and development by over a hundred engineers. A typical task is initiated by a human user and runs up to several hours. It reads through a few MBs to a few TBs of data, transforms or analyzes the data, and writes the results back to the cluster. Cluster B is primarily used for production data processing. The tasks last much longer and continuously generate and process multi-TB data sets with only occasional human intervention. In both cases, a single “task” consists of many processes on many machines reading and writing many files simultaneously.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Cluster | A | | B | |
| Chunkservers | 342 |  | 227 |  |
| Available disk space | 72 | TB | 180 | TB |
| Used disk space | 55 | TB | 155 | TB |
| Number of Files | 735 | k | 737 | k |
| Number of Dead files | 22 | k | 232 | k |
| Number of Chunks | 992 | k | 1550 | k |
| Metadata at chunkservers | 13 GB | | 21 GB | |
| Metadata at master | 48 MB | | 60 MB | |

**Table 2: Characteristics of two GFS clusters**

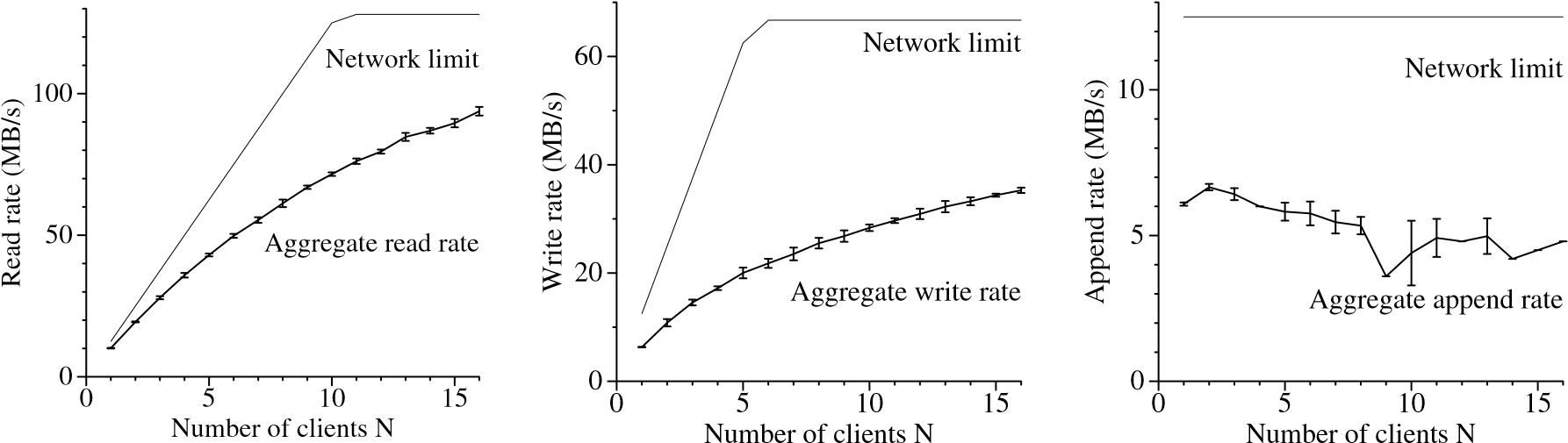
*Storage*

As shown by the first five entries in the table, both clusters have hundreds of chunkservers, support many TBs of disk space, and are fairly but not completely full. “Used space” includes all chunk replicas. Virtually all files are replicated three times. Therefore, the clusters store 18 TB and 52 TB of file data respectively.

The two clusters have similar numbers of files, though B has a larger proportion of dead files, namely files which were deleted or replaced by a new version but whose storage have not yet been reclaimed. It also has more chunks because its files tend to be larger.

*Metadata*

The chunkservers in aggregate store tens of GBs of metadata, mostly the checksums for 64 KB blocks of user data. The only other metadata kept at the chunkservers is the chunk version number discussed in Section 4.5.



(a) Reads (b) Writes (c) Record appends

**Figure 3: Aggregate Throughputs.**

The metadata kept at the master is much smaller, only tens of MBs, or about 100 bytes per file on average. This agrees with our assumption that the size of the master’s memory does not limit the system’s capacity in practice. Most of the per-file metadata is the file names stored in a prefix-compressed form. Other metadata includes file ownership and permissions, mapping from files to chunks, and each chunk’s current version. In addition, for each chunk we store the current replica locations and a reference count for implementing copy-on-write.

Each individual server, both chunkservers and the master, has only 50 to 100 MB of metadata. Therefore recovery is fast: it takes only a few seconds to read this metadata from disk before the server is able to answer queries. However, the master is somewhat hobbled for a period – typically 30 to 60 seconds – until it has fetched chunk location information from all chunkservers.

*Read and Write Rates*

Table 3 shows read and write rates for various time periods. Both clusters had been up for about one week when these measurements were taken. (The clusters had been restarted recently to upgrade to a new version of GFS.)

The average write rate was less than 30 MB/s since the restart. When we took these measurements, B was in the middle of a burst of write activity generating about 100 MB/s of data, which produced a 300 MB/s network load because writes are propagated to three replicas.

because of low variance in measurements.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Cluster | A | | B | |
| Read rate (last minute) | 583 | MB/s | 380 | MB/s |
| Read rate (last hour) | 562 | MB/s | 384 | MB/s |
| Read rate (since restart) | 589 | MB/s | 49 | MB/s |
| Write rate (last minute) | 1 | MB/s | 101 | MB/s |
| Write rate (last hour) | 2 | MB/s | 117 | MB/s |
| Write rate (since restart) | 25 | MB/s | 13 | MB/s |
| Master ops (last minute) | 325 Ops/s | | 533 Ops/s | |
| Master ops (last hour) | 381 Ops/s | | 518 Ops/s | |
| Master ops (since restart) | 202 Ops/s | | 347 Ops/s | |

**Table 3: Performance Metrics for Two GFS Clusters**

The read rates were much higher than the write rates. The total workload consists of more reads than writes as we have assumed. Both clusters were in the middle of heavy read activity. In particular, A had been sustaining a read rate of 580 MB/s for the preceding week. Its network configuration can support 750 MB/s, so it was using its resources efficiently. Cluster B can support peak read rates of 1300 MB/s, but its applications were using just 380 MB/s.

*Master Load*

Table 3 also shows that the rate of operations sent to the master was around 200 to 500 operations per second. The master can easily keep up with this rate, and therefore is not a bottleneck for these workloads.

In an earlier version of GFS, the master was occasionally a bottleneck for some workloads. It spent most of its time sequentially scanning through large directories (which contained hundreds of thousands of files) looking for particular files. We have since changed the master data structures to allow efficient binary searches through the namespace. It can now easily support many thousands of file accesses per second. If necessary, we could speed it up further by placing name lookup caches in front of the namespace data structures.

*Recovery Time*

After a chunkserver fails, some chunks will become underreplicated and must be cloned to restore their replication levels. The time it takes to restore all such chunks depends on the amount of resources. In one experiment, we killed a single chunkserver in cluster B. The chunkserver had about 15,000 chunks containing 600 GB of data. To limit the impact on running applications and provide leeway for scheduling decisions, our default parameters limit this cluster to 91 concurrent clonings (40% of the number of chunkservers) where each clone operation is allowed to consume at most 6.25 MB/s (50 Mbps). All chunks were restored in 23.2 minutes, at an effective replication rate of 440 MB/s.

In another experiment, we killed two chunkservers each with roughly 16,000 chunks and 660 GB of data. This double failure reduced 266 chunks to having a single replica. These 266 chunks were cloned at a higher priority, and were all restored to at least 2x replication within 2 minutes, thus putting the cluster in a state where it could tolerate another chunkserver failure without data loss.

**Workload Breakdown**

In this section, we present a detailed breakdown of the workloads on two GFS clusters comparable but not identical to those in Section 6.2. Cluster X is for research and development while cluster Y is for production data processing.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Read | Write | Record Append |
| Cluster | X Y | X Y | X Y |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0K | 0.4 2.6 | | 0 0 | 0 | 0 |
| 1B..1K | 0.1 4.1 | | 6.6 4.9 | 0.2 | 9.2 |
| 1K..8K | 65.2 38.5 | | 0.4 1.0 | 18.9 | 15.2 |
| 8K..64K | 29.9 45.1 | | 17.8 43.0 | 78.0 | 2.8 |
| 64K..128K | 0.1 | 0.7 | 2.3 1.9 | *<* .1 | 4.3 |
| 128K..256K | 0.2 | 0.3 | 31.6 0.4 | *<* .1 | 10.6 |
| 256K..512K | 0.1 | 0.1 | 4.2 7.7 | *<* .1 | 31.2 |
| 512K..1M | 3.9 | 6.9 | 35.5 28.7 | 2.2 | 25.5 |
| 1M..inf | 0.1 | 1.8 | 1.5 12.3 | 0.7 | 2.2 |

**Table 4: Operations Breakdown by Size (%).**

*Methodology and Caveats*

These results include only client originated requests so that they reflect the workload generated by our applications for the file system as a whole. They do not include interserver requests to carry out client requests or internal background activities, such as forwarded writes or rebalancing.

Statistics on I/O operations are based on information heuristically reconstructed from actual RPC requests logged by GFS servers. For example, GFS client code may break a read into multiple RPCs to increase parallelism, from which we infer the original read. Since our access patterns are highly stylized, we expect any error to be in the noise. Explicit logging by applications might have provided slightly more accurate data, but it is logistically impossible to recompile and restart thousands of running clients to do so and cumbersome to collect the results from as many machines.

One should be careful not to overly generalize from our workload. Since Google completely controls both GFS and its applications, the applications tend to be tuned for GFS, and conversely GFS is designed for these applications. Such mutual influence may also exist between general applications

For reads, the size is the amount of data actually read and transferred, rather than the amount requested.

and file systems, but the effect is likely more pronounced in our case.

*Chunkserver Workload*

Table 4 shows the distribution of operations by size. Read sizes exhibit a bimodal distribution. The small reads (under 64 KB) come from seek-intensive clients that look up small pieces of data within huge files. The large reads ( over 512 KB) come from long sequential reads through entire files.

A significant number of reads return no data at all in cluster Y. Our applications, especially those in the production systems, often use files as producer-consumer queues. Producers append concurrently to a file while a consumer reads the end of file. Occasionally, no data is returned when the consumer outpaces the producers. Cluster X shows this less often because it is usually used for short-lived data analysis tasks rather than long-lived distributed applications.

Write sizes also exhibit a bimodal distribution. The large writes (over 256 KB) typically result from significant buffering within the writers. Writers that buffer less data, checkpoint or synchronize more often, or simply generate less data account for the smaller writes (under 64 KB).

As for record appends, cluster Y sees a much higher percentage of large record appends than cluster X does because our production systems, which use cluster Y, are more aggressively tuned for GFS.

Table 5 shows the total amount of data transferred in operations of various sizes. For all kinds of operations, the larger operations (over 256 KB) generally account for most of the bytes transferred. Small reads (under 64 KB) do transfer a small but significant portion of the read data because of the random seek workload.

*Appends versus Writes*

Record appends are heavily used especially in our production systems. For cluster X, the ratio of writes to record appends is 108:1 by bytes transferred and 8:1 by operation counts. For cluster Y, used by the production systems, the ratios are 3.7:1 and 2.5:1 respectively. Moreover, these ratios suggest that for both clusters record appends tend to be larger than writes. For cluster X, however, the overall usage of record append during the measured period is fairly low and so the results are likely skewed by one or two applications with particular buffer size choices.

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | Read | Write | Record Append |
| Cluster | X Y | X Y | X Y |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1B..1K | *<* .1 *<* .1 | | *<* .1 *<* .1 | *<* .1 | *<* .1 |
| 1K..8K | 13.8 3.9 | | *<* .1 *<* .1 | *<* .1 | 0.1 |
| 8K..64K | 11.4 | 9.3 | 2.4 5.9 | 2.3 | 0.3 |
| 64K..128K | 0.3 | 0.7 | 0.3 0.3 | 22.7 | 1.2 |
| 128K..256K | 0.8 | 0.6 | 16.5 0.2 | *<* .1 | 5.8 |
| 256K..512K | 1.4 | 0.3 | 3.4 7.7 | *<* .1 | 38.4 |
| 512K..1M | 65.9 | 55.1 | 74.1 58.0 | .1 | 46.8 |
| 1M..inf | 6.4 | 30.1 | 3.3 28.0 | 53.9 | 7.4 |

**Table 5: Bytes Transferred Breakdown by Operation Size (%).**

As expected, our data mutation workload is dominated by appending rather than overwriting. We measured the amount of data overwritten on primary replicas. This apFor reads, the size is the amount of data actually read and transferred, rather than the amount requested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads.

|  |  |  |
| --- | --- | --- |
| Cluster | X | Y |
| Open | 26.1 16.3 | |
| Delete | 0.7 1.5 | |
| FindLocation | 64.3 65.8 | |
| FindLeaseHolder | 7.8 13.4 | |
| FindMatchingFiles | 0.6 2.2 | |
| All other combined | 0.5 0.8 | |

**Table 6: Master Requests Breakdown by Type ( % )**

proximates the case where a client deliberately overwrites previous written data rather than appends new data. For cluster X, overwriting accounts for under 0.0001% of bytes mutated and under 0.0003% of mutation operations. For cluster Y, the ratios are both 0.05%. Although this is minute, it is still higher than we expected. It turns out that most of these overwrites came from client retries due to errors or timeouts. They are not part of the workload *per se* but a consequence of the retry mechanism.

*Master Workload*

Table 6 shows the breakdown by type of requests to the master. Most requests ask for chunk locations (*FindLocation*) for reads and lease holder information (*FindLeaseLocker*) for data mutations.

Clusters X and Y see significantly different numbers of *Delete* requests because cluster Y stores production data sets that are regularly regenerated and replaced with newer versions. Some of this difference is further hidden in the difference in *Open* requests because an old version of a file may be implicitly deleted by being opened for write from scratch (mode “w” in Unix open terminology).

*FindMatchingFiles* is a pattern matching request that supports “ls” and similar file system operations. Unlike other requests for the master, it may process a large part of the namespace and so may be expensive. Cluster Y sees it much more often because automated data processing tasks tend to examine parts of the file system to understand global application state. In contrast, cluster X’s applications are under more explicit user control and usually know the names of all needed files in advance.

**EXPERIENCES**

In the process of building and deploying GFS, we have experienced a variety of issues, some operational and some technical.

Initially, GFS was conceived as the backend file system for our production systems. Over time, the usage evolved to include research and development tasks. It started with little support for things like permissions and quotas but now includes rudimentary forms of these. While production systems are well disciplined and controlled, users sometimes are not. More infrastructure is required to keep users from interfering with one another.

Some of our biggest problems were disk and Linux related. Many of our disks claimed to the Linux driver that they supported a range of IDE protocol versions but in fact responded reliably only to the more recent ones. Since the protocol versions are very similar, these drives mostly worked, but occasionally the mismatches would cause the drive and the kernel to disagree about the drive’s state. This would corrupt data silently due to problems in the kernel. This problem motivated our use of checksums to detect data corruption, while concurrently we modified the kernel to handle these protocol mismatches.

Earlier we had some problems with Linux 2.2 kernels due to the cost of fsync(). Its cost is proportional to the size of the file rather than the size of the modified portion. This was a problem for our large operation logs especially before we implemented checkpointing. We worked around this for a time by using synchronous writes and eventually migrated to Linux 2.4.

Another Linux problem was a single reader-writer lock which any thread in an address space must hold when it pages in from disk (reader lock) or modifies the address space in an mmap() call (writer lock). We saw transient timeouts in our system under light load and looked hard for resource bottlenecks or sporadic hardware failures. Eventually, we found that this single lock blocked the primary network thread from mapping new data into memory while the disk threads were paging in previously mapped data. Since we are mainly limited by the network interface rather than by memory copy bandwidth, we worked around this by replacing mmap() with pread() at the cost of an extra copy.

Despite occasional problems, the availability of Linux code has helped us time and again to explore and understand system behavior. When appropriate, we improve the kernel and share the changes with the open source community.

**RELATED WORK**

Like other large distributed file systems such as AFS [5] , GFS provides a location independent namespace which enables data to be moved transparently for load balance or fault tolerance. Unlike AFS, GFS spreads a file’s data across storage servers in a way more akin to xFS [1] and Swift [3] in order to deliver aggregate performance and increased fault tolerance.

As disks are relatively cheap and replication is simpler than more sophisticated RAID [9] approaches, GFS currently uses only replication for redundancy and so consumes more raw storage than xFS or Swift.

In contrast to systems like AFS, xFS, Frangipani [12], and Intermezzo [6], GFS does not provide any caching below the file system interface. Our target workloads have little reuse within a single application run because they either stream through a large data set or randomly seek within it and read small amounts of data each time.

Some distributed file systems like Frangipani, xFS, Minnesota’s GFS[11] and GPFS [10] remove the centralized server and rely on distributed algorithms for consistency and management. We opt for the centralized approach in order to simplify the design, increase its reliability, and gain flexibility. In particular, a centralized master makes it much easier to implement sophisticated chunk placement and replication policies since the master already has most of the relevant information and controls how it changes. We address fault tolerance by keeping the master state small and fully replicated on other machines. Scalability and high availability (for reads) are currently provided by our shadow master mechanism. Updates to the master state are made persistent by appending to a write-ahead log. Therefore we could adapt a primary-copy scheme like the one in Harp [7] to provide high availability with stronger consistency guarantees than our current scheme.

We are addressing a problem similar to Lustre [8] in terms of delivering aggregate performance to a large number of clients. However, we have simplified the problem significantly by focusing on the needs of our applications rather than building a POSIX-compliant file system. Additionally, GFS assumes large number of unreliable components and so fault tolerance is central to our design.

GFS most closely resembles the NASD architecture [4]. While the NASD architecture is based on network-attached disk drives, GFS uses commodity machines as chunkservers, as done in the NASD prototype. Unlike the NASD work, our chunkservers use lazily allocated fixed-size chunks rather than variable-length objects. Additionally, GFS implements features such as rebalancing, replication, and recovery that are required in a production environment.

Unlike Minnesota’s GFS and NASD, we do not seek to alter the model of the storage device. We focus on addressing day-to-day data processing needs for complicated distributed systems with existing commodity components.

The producer-consumer queues enabled by atomic record appends address a similar problem as the distributed queues in River [2]. While River uses memory-based queues distributed across machines and careful data flow control, GFS uses a persistent file that can be appended to concurrently by many producers. The River model supports m-to-n distributed queues but lacks the fault tolerance that comes with persistent storage, while GFS only supports m-to-1 queues efficiently. Multiple consumers can read the same file, but they must coordinate to partition the incoming load.

**CONCLUSIONS**

The Google File System demonstrates the qualities essential for supporting large-scale data processing workloads on commodity hardware. While some design decisions are specific to our unique setting, many may apply to data processing tasks of a similar magnitude and cost consciousness. We started by reexamining traditional file system assumptions in light of our current and anticipated application workloads and technological environment. Our observations have led to radically different points in the design space. We treat component failures as the norm rather than the exception, optimize for huge files that are mostly appended to (perhaps concurrently) and then read (usually sequentially), and both extend and relax the standard file system interface to improve the overall system.

Our system provides fault tolerance by constant monitoring, replicating crucial data, and fast and automatic recovery. Chunk replication allows us to tolerate chunkserver failures. The frequency of these failures motivated a novel online repair mechanism that regularly and transparently repairs the damage and compensates for lost replicas as soon as possible. Additionally, we use checksumming to detect data corruption at the disk or IDE subsystem level, which becomes all too common given the number of disks in the system.

Our design delivers high aggregate throughput to many concurrent readers and writers performing a variety of tasks. We achieve this by separating file system control, which passes through the master, from data transfer, which passes directly between chunkservers and clients. Master involvement in common operations is minimized by a large chunk size and by chunk leases, which delegates authority to primary replicas in data mutations. This makes possible a simple, centralized master that does not become a bottleneck. We believe that improvements in our networking stack will lift the current limitation on the write throughput seen by an individual client.

GFS has successfully met our storage needs and is widely used within Google as the storage platform for research and development as well as production data processing. It is an important tool that enables us to continue to innovate and attack problems on the scale of the entire web.

**ACKNOWLEDGMENTS**

We wish to thank the following people for their contributions to the system or the paper. Brain Bershad (our shepherd) and the anonymous reviewers gave us valuable comments and suggestions. Anurag Acharya, Jeff Dean, and David desJardins contributed to the early design. Fay Chang worked on comparison of replicas across chunkservers. Guy Edjlali worked on storage quota. Markus Gutschke worked on a testing framework and security enhancements. David Kramer worked on performance enhancements. Fay Chang, Urs Hoelzle, Max Ibel, Sharon Perl, Rob Pike, and Debby Wallach commented on earlier drafts of the paper. Many of our colleagues at Google bravely trusted their data to a new file system and gave us useful feedback. Yoshka helped with early testing.

**REFERENCES**

1. Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang.

Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

1. Remzi H. Arpaci-Dusseau, Eric Anderson, Noah

Treuhaft, David E. Culler, Joseph M. Hellerstein,

David Patterson, and Kathy Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS ’99)*, pages 10 –22, Atlanta, Georgia, May 1999.

1. Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed disk striping to provide high I/O data rates. *Computer Systems*, 4(4):405–436, 1991.
2. Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff

Butler, Fay W. Chang, Howard Gobioff, Charles

Hardin, Erik Riedel, David Rochberg, and Jim

Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the 8th Architectural Support for Programming Languages and Operating Systems*, pages 92–103, San Jose, California, October 1998.

1. John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81 , February 1988.
2. InterMezzo. http://www.inter-mezzo.org, 2003.
3. Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams.

Replication in the Harp file system. In *13 th*

*Symposium on Operating System Principles*, pages 226–238, Pacific Grove, CA, October 1991.

1. Lustre. http://www.lustreorg, 2003.
2. David A. Patterson, Garth A. Gibson, and Randy H.

Katz. A case for redundant arrays of inexpensive disks

(RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, September 1988.

1. Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies*, pages 231–244, Monterey, California, January 2002.
2. Steven R. Soltis, Thomas M. Ruwart, and Matthew T. O’Keefe. The Gobal File System. In *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, College Park, Maryland, September 1996.
3. Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating System Principles*, pages 224–237 , Saint-Malo, France, October 1997.