基于 Erasure Code 与 P2P 的分布式文件系统

前期调研报告

王珺 夏昊珺 滕思洁 郑值

2017 年 3 月 18 日星期六

摘要

本文先分析了在当前家庭和小微企业中计算机存储系统的典型使用环境及传统的集中式文件系统和客户端-服务器的体系分布式文件系统在这种环境中应用的困难，接着介绍了近几年来发展迅猛的 JVM、P2P 和 Erasure code 等技术并提出了基于这些技术构建一个分布式文件系统的构想。最后，本文介绍了现今流行的几个分布式文件系统的实例并与我们构想的分布式文件系统进行了对比。

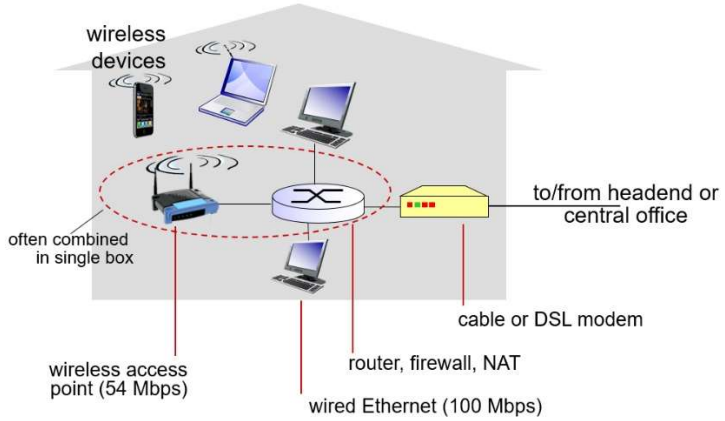
目录

一、项目背景
……4
JVM
……5
Erasure code 原理
……6
P2P 网络传输
……7
分布式文件系统
……9
二、立项依据与目的
……10
三、项目前瞻性与重要性分析
……11
四、相关工作
……12
Google File System
……12
Cooperative File System
……15
Andrew File System
……17
分布式文件系统间的对比
……19
参考文献
……20

一、项目背景

随着社会经济的发展与信息化进程的继续，台式计算机、膝上电脑、智能手机、平板电脑和更多的智能可穿戴设备正疯狂涌入当前的家庭和小微企业。这些设备极大地提高了企业的办公效率、丰富了家庭的娱乐需求，但如何高效地利用分散在这些不同的设备上的存储空间如今正越发成为大家关注的问题：运用好这些分散的存储空间不仅可以方便多人合作办公更可以避免资源的浪费。

下图[0]是当前家庭和小微企业中典型环境的示意图，其中存储设备包括图中的无线设备、膝上电脑和台式计算机上，这种环境有以下特点：



（1）存储资源小而分散，每个设备的存储容量通常不超过 1TB；
（2）设备通常只有在使用时才会在线联网，否则处于关闭状态；
（3）很多设备的位置随时间而变化，故它们常常并不总在其归属网络上；
（4）和专用的服务器相比，这些设备的性能较低；
（5）设备没有统一的体系结构、指令集和操作系统；
（6）连接设备的网络环境较差，往往通过一般的局域网或互联网相连接。

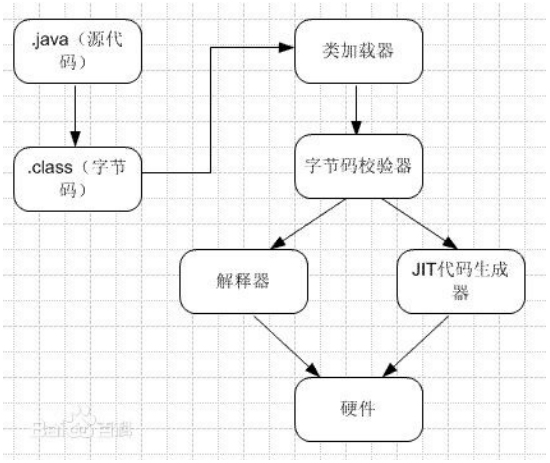
面对这些特点，很难用一个集中式的文件系统组织分散在这些不同的设备上的存储空间。然而，即使是客户端-服务器的体系分布式文件系统想在这种环境中应用也是十分困难的，这体现在
（1）客户端-服务器的体系分布式文件系统往往要求高性能、稳定的服务器，而上述环境中的机器不但性能不足，更不常在线；
（2）客户端-服务器的体系分布式文件系统往往要求服务器具有相同的操作系统甚至是定制的操作系统以方便管理，而上述环境中的机器运行不同的操作系统且很多设备难以定制操作系统；
（3）客户端-服务器的体系分布式文件系统往往通过高速的局域网保障设备间通讯的速度，而上述环境中的网络难以满足要求。

上述问题的出现使得设计一种新型的分布式文件系统变得十分必要，而近些年逐渐成熟并得到应用的 JVM、Erasure code 和 P2P 网络传输等技术为这一分布式文件系统的出现提供了可能。下面先介绍 JVM、Erasure code、P2P 网络传输三种技术，再简单地回顾分布式文件系统的意义。

JVM

JVM是Java Virtual Machine（Java虚拟机）的缩写，JVM是一种用于计算设备的规范，它是一个虚构出来的计算机，是通过在实际的计算机上仿真模拟各种计算机功能来实现的。Java语言的一个非常重要的特点就是与平台的无关性。而使用Java虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的目标代码。而引入Java语言虚拟机后，Java语言在不同平台上运行时不需要重新编译。Java语言使用Java虚拟机屏蔽了与具体平台相关的信息，使得Java语言编译程序只需生成在Java虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。Java虚拟机在执行字节码时，把字节码解释成具体平台上的机器指令执行。这就是Java的能够"一次编译，到处运行"的原因。

JVM是java的核心和基础，在java编译器和os平台之间的虚拟处理器。它是一种基于下层的操作系统和硬件平台并利用软件方法来实现的抽象的计算机，可以在上面执行java的字节码程序。java编译器只需面向JVM，生成JVM能理解的代码或字节码文件。Java源文件经编译器，编译成字节码程序，通过JVM将每一条指令翻译成不同平台机器码，通过特定平台运行。Jvm运行原理如下图



JVM指令系统同其他计算机的指令系统极其相似。Java指令也是由操作码和操作数两部分组成。操作码为8位二进制数，操作数紧随在操作码的后面，其长度根据需要而不同。JVM只设置了4个最为常用的寄存器。它们是：

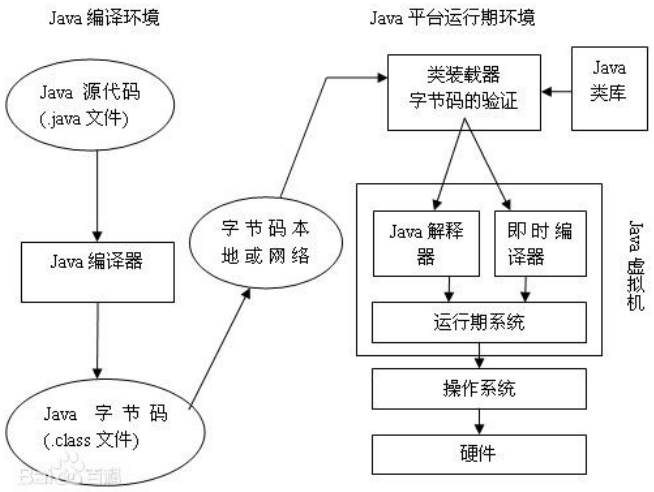
pc程序计数器、optop操作数栈顶指针、frame当前执行环境指针、vars指向当前执行环境中第一个局部变量的指针

所有寄存器均为32位。pc用于记录程序的执行。optop,frame和vars用于记录指向Java栈区的指针。

Java栈是JVM存储信息的主要方法。当JVM得到一个Java字节码应用程序后，便为该代码中一个类的每一个方法创建一个栈框架，以保存该方法的状态信息。每个栈框架包括以下三类信息：

局部变量、执行环境、操作数栈

JVM体系结构如下图



Erasure code 原理

erasure code是一种技术，它可以将n份原始数据，增加m份数据，并能通过n+m份中的任意n份数据，还原为原始数据。RS code是最基本的一种。RS 类纠删码由生成矩阵不同主要分为两类，范德蒙码和柯西码。

存储系统中的符号约定

k:数据块的个数

m:校验块的个数（就是code）

n:k+m，也就是数据块和校验块的个数总和。

编码效率：r = k/m

以下介绍范德蒙码。其基本思想很简单，采用范德蒙矩阵作为生成矩阵，得到校验数据。现假设输入数据为D1~Dn，生成的校验数据为C1~Cm，那么输入数据和校验数据之间的关系可以描述为：

$$\begin{bmatrix} f_{1,1} & f_{1,2} & \cdots & f_{1,n} \\ f_{2,1} & f_{2,2} & \cdots & f_{2,n} \\ f_{3,1} & f_{3,2} & \cdots & f_{3,n} \\ \vdots & \vdots & \vdots & \vdots \\ f_{m,1} & f_{m,2} & \cdots & f_{m,n} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix} =$$

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & n \\ 1 & 4 & \cdots & n^2 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 2^{m-1} & \cdots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_m \end{bmatrix}$$

其中,

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 2 & \cdots & n \\ 1 & 4 & \cdots & n^2 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 2^{m-1} & \cdots & n^{m-1} \end{bmatrix}$$

为范德蒙矩阵，该矩阵为编码矩阵。所以，为了得到校验数据，主要的任务是将输入数据

和编码矩阵相乘，得到的输出结果就是编码值。为了能够更好的表示磁盘上存储的数据，通常将编码矩阵方程表示如下：

$$\begin{bmatrix} 1 & 0 & 0\cdots & 0 \\ 0 & 1 & 0\cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0\cdots & 1 \\ 1 & 1 & 1\cdots & 1 \\ 1 & 2 & 3\cdots & n \\ \vdots & \vdots & \vdots & \vdots \\ 1 & 2^{m-1} & 3^{m-1}\cdots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \\ c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

可以发现这个生成矩阵（A）就是单元矩阵和范德蒙矩阵的组合。输入数据（D）和生成矩阵（A）的乘积就是编码之后的存储数据（E）。

如何进行解码操作呢？当存储的数据 d1~dn,c1~cm 中有些数据无法读取时，如何进行数据恢复呢？从数学的原理来看，只要将读取的有效数据和生成矩阵的逆矩阵相乘就可以恢复丢失的数据。假设 m 个数据块丢失，则可以将 m 个数据块对应的矩阵 A 和 E 中的行删掉，得到新的 n*n 阶生成矩阵 A2 和 1*n 阶结果矩阵 E2 。由于生成矩阵是有范德蒙矩阵和单元矩阵的组合，所以，矩阵 A 的任意 n 行子集都可以保证线性无关。因此，需要恢复的数据可以通过 A2 和 E2 的逆矩阵乘积得到，即 D= 逆（A2）*逆（E2）。

基于范德蒙矩阵的 Erasure Code 编解码原理比较简单，。采用这种方法的算法复杂度还是比较高的，编码复杂度为 O（mn），其中 m 为校验数据个数， n 为输入数据个数。解码复杂度为 O（n^3），解码具有较高的计算复杂度。

P2P 网络传输

对等网络，即对等计算机网络，是一种在对等者（Peer）之间分配任务和工作负载的分布式应用架构，是对等计算模型在应用层形成的一种组网或网络形式。网络的参与者共享他们所拥有的一部分硬件资源（处理能力、存储能力、网络连接能力、打印机等），这些共享资源通过网络提供服务和内容，能被其它对等节点（Peer）直接访问而无需经过中间实体。在此网络中的参与者既是资源、服务和内容的提供者（Server），又是资源、服务和内容的获取者（Client）

P2P网络技术的特点体现在以下几个方面：
非中心化：网络中的资源和服务分散在所有节点上，信息的传输和服务的实现都直接在节点之间进行，可以无需中间环节和服务器的介入，避免了可能的瓶颈。P2P的非中心化基本特点，带来了其在可扩展性、健壮性等方面的优势。
可扩展性：在P2P网络中，随着用户的加入，不仅服务的需求增加了，系统整体的资源和服务能力也在同步地扩充，始终能比较容易地满足用户的需要。理论上其可扩展性几乎可以认为是无限的。例如：在传统的通过FTP的文件下载方式中，当下载用户增加之后，下载速度会变得越来越慢，然而P2P网络正好相反，加入的用户越多，P2P网络中提供的资源就越多，下载的速度反而越快。
健壮性：P2P架构天生具有耐攻击、高容错的优点。由于服务是分散在各个节点之间进行

的，部分节点或网络遭到破坏对其它部分的影响很小。P2P网络一般在部分节点失效时能够自动调整整体拓扑，保持其它节点的连通性。P2P网络通常都是以自组织的方式建立起来的，并允许节点自由地加入和离开。

高性价比：性能优势是P2P被广泛关注的一个重要原因。随着硬件技术的发展，个人计算机的计算和存储能力以及网络带宽等性能依照摩尔定理高速增长。采用P2P架构可以有效地利用互联网中散布的大量普通结点，将计算任务或存储资料分布到所有节点上。利用其中闲置的计算能力或存储空间，达到高性能计算和海量存储的目的。目前，P2P在这方面的应用多在学术研究方面，一旦技术成熟，能够在工业领域推广，则可以为许多企业节省购买大型服务器的成本。

隐私保护：在P2P网络中，由于信息的传输分散在各节点之间进行而无需经过某个集中环节，用户的隐私信息被窃听和泄漏的可能性大大缩小。此外，目前解决Internet隐私问题主要采用中继转发的技术方法，从而将通信的参与者隐藏在众多的网络实体之中。在传统的一些匿名通信系统中，实现这一机制依赖于某些中继服务器节点。而在P2P中，所有参与者都可以提供中继转发的功能，因而大大提高了匿名通讯的灵活性和可靠性，能够为用户提供更好的隐私保护。

负载均衡：P2P网络环境下由于每个节点既是服务器又是客户机，减少了对传统C/S结构服务器计算能力、存储能力的要求，同时因为资源分布在多个节点，更好的实现了整个网络的负载均衡。

由于对等网络不需要专门的服务器来做网络支持，也不需要其他的组件来提高网络的性能，因而组网成本较低，适用于人员少、组网简单的场景，故常用于网络较小的中小型企业或家庭中。

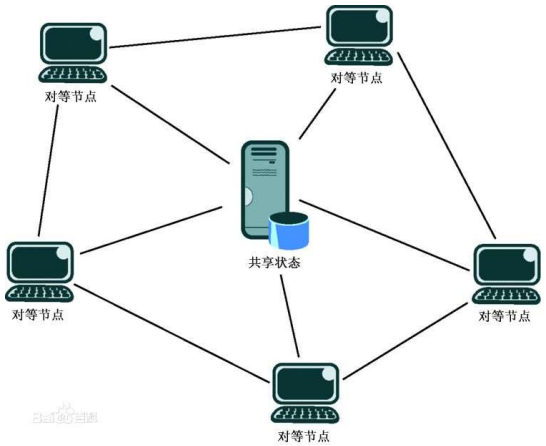
与客户端/服务器网络相比，对等网络具有下列优势：

1、可在网络的中央及边缘区域共享内容和资源。在客户端/服务器网络中，通常只能在网络的中央区域共享内容和资源。

2、由对等方组成的网络易于扩展，而且比单台服务器更加可靠。单台服务器会受制于单点故障，或者会在网络使用率偏高时，形为瓶颈。

3、由对等方组成的网络可共享处理器，整合计算资源以执行分布式计算任务，而不只是单纯依赖一台计算机，如一台超级计算机。

4、用户可直接访问对等计算机上的共享资源。网络中的对等方可直接在本地存储器上共享文件，而不必在中央服务器上进行共享。



分布式文件系统

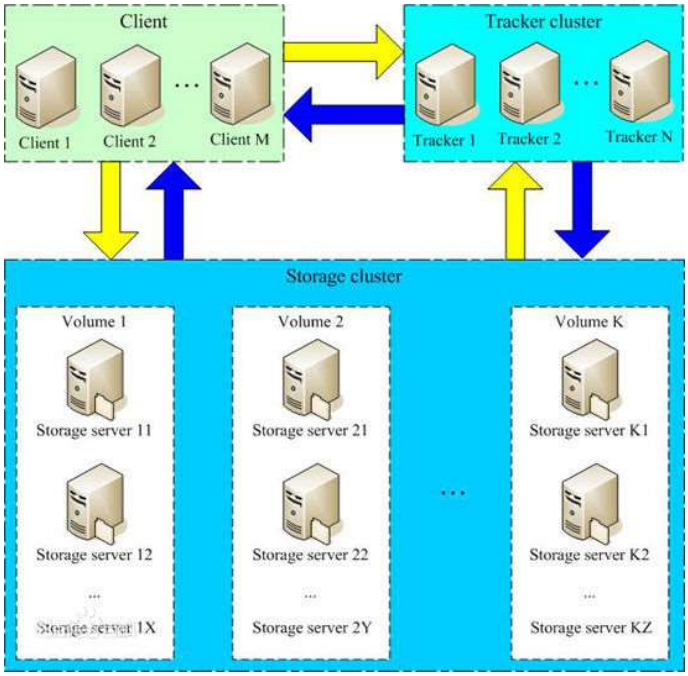
分布式文件系统（Distributed File System）是指文件系统管理的物理存储资源不一定直接连接在本地节点上，而是通过计算机网络与节点相连。分布式文件系统的设计基于客户机/服务器模式。一个典型的网络可能包括多个供多用户访问的服务器。另外，对等特性允许一些系统扮演客户机和服务器的双重角色。例如，用户可以“发表”一个允许其他客户机访问的目录，一旦被访问，这个目录对客户机来说就像使用本地驱动器一样。

文件系统最初设计时，仅仅是为局域网内的本地数据服务的。而分布式文件系统将服务范围扩展到了整个网络。不仅改变了数据的存储和管理方式，也拥有了本地文件系统所无法具备的数据备份、数据安全等优点。判断一个分布式文件系统是否优秀，取决于以下三个因素：

数据的存储方式：例如有1000万个数据文件，可以在一个节点存储全部数据文件，在其他N个节点上每个节点存储1000/N万个数据文件作为备份；或者平均分配到N个节点上存储，每个节点上存储1000/N万个数据文件。无论采取何种存储方式，目的都是为了保证数据的存储安全和方便获取。

数据的读取速率：包括响应用户读取数据文件的请求、定位数据文件所在的节点、读取实际硬盘中数据文件的时间、不同节点间的数据传输时间以及一部分处理器的处理时间等。各种因素决定了分布式文件系统的用户体验。即分布式文件系统中数据的读取速率不能与本地文件系统中数据的读取速率相差太大，否则在本地文件系统中打开一个文件需要2秒，而在分布式文件系统中各种因素的影响下用时超过10秒，就会严重影响用户的使用体验。

数据的安全机制：由于数据分散在各个节点中，必须要采取冗余、备份、镜像等方式保证节点出现故障的情况下，能够进行数据的恢复，确保数据安全。



二、立项依据与目的

我们的项目最终目的是实现一个跨平台的，可提供移动式文件访问的分布式文件系统。通过类似 windows 文件管理器的 GUI 接口进行分布式的文件管理和访问。

该文件系统提供两种共享服务：

- 云共享：

将本地的一些文件目录设置为云共享，组内成员可以像访问自己的文件一样进行文件操作；甚至当该节点本身不在线时，依旧可以通过拼凑整个系统内的文件碎片来获取该离线机的云共享文件。

- 本地共享：

将本地一些文件目录设置为本地共享，分布式文件系统相同组内成员节点可以像访问自己的文件一样进行文件操作。

该文件系统的使用者可能是基于个人目的或者是小组目的。

个人用途：

- 利用闲置的硬盘充当个人云盘
- 实现移动化的文件访问

将个人家庭的，办公室的，学校的，甚至是智能手机，只要是能连接上 Internet 的设备，连接在一起，实现轻松地一体化的文件访问和管理。

- 实现文件备份，只要放到云共享目录，即使原副本机器损坏，还是可以通过其他碎片恢复源文件

小组用途：

- 实现对小组成员基于广域网对共享资源协作管理和使用

比如小组内共享的相关资料既需要共享访问，又需要共享添加和修改

- 实现局域网内的文件快捷共享（将室友电脑里的游戏拷到自己电脑里就像将游戏从自己的一个盘拷贝到另一个盘那么简单，只要对方将该目录加入本地共享目录）

三、项目前瞻性与重要性分析

面对项目背景中提出的典型使用环境，我们的项目具备如下前瞻性的设计：

基于 JAVA 虚拟机实现跨平台分布式移动访问 实现便捷设备互联

- 随着移动互联网的进一步发展，设备互联的趋势不断得到发展，跨平台的概念符合当前的发展趋势。

基于 JAVA 虚拟机可以便捷的实现在 windows、linux、mac、ios 和 android 系统上的文件共享。

- 当前人们的设备越来越多，文件越来越杂乱，需要这种可进行统一管理的文件服务
- 人们的设备越来越分散，可能家里有电脑，办公室有，学校有，需要移动式的文件访问服务。
- 同时，现在正是该系统能投入实用并且具有优势的时期：
  - 移动网络不断发展，4G 网络带宽和流量的剧增使得手机等智能设备也加入了共享的设备之列
  - 网络带宽获得了显著的提升，尽管分布式文件系统需要传送很多数据，但是带宽已经不再是太大问题
  - 当前个人存储设备（如硬盘）容量越来越大，完全可以承担存储冗余备份的责任
  - 当前的云盘的可靠性和安全性常被质疑

基于 P2P 原理的网络实现共享

- 现在很多文件备份和个人资料的整合都是通过云盘实现的，人们将电脑上的照片和手机上的照片都放到云盘里，基本上还是基于 server-client 机制
- 而我们的系统参考了 P2P 原理，在保留名字服务器的同时使用 P2P 原理在各个对等方间进行数据传输，这意味着
  - 减少服务器依赖---一个分布式文件系统中名字服务器往往只有 1～2 台，故省去数据服务器可以避免对系统中绝大多数服务器的依赖
  - 直接---直接对目标机器进行操作，不用本地和云盘反复的信息更新和整合
  - 高效---允许多个发送方同时向一个接受方发送数据，减少带宽瓶颈的影响

基于 Essure Code 实现高效冗余备份

- 这种技术很高效，在较低空间开销下实现了高可用性

四、相关工作

这一节中我们将先简要介绍现有的几种分布式文件系统，并在最后将它们与本次实验中拟设计的分布式文件系统进行对比。对于实验中需要的其他技术（如Erasure Code或P2P），请见第一节的项目背景部分。

分布式文件系统的早期雏形可以追溯到1976年Digital Equipment Corporation设计出的File Access Listener（FAL）。其实现了Data Access Protocol，是第一个被广为使用的网络文件系统[1]。到了1980年代，以由Carnegie Mellon University 开发的Andrew File System（AFS）与由Sun Microsystems开发的Network File System（NFS）为首，第一代分布式文件系统开始出现、发展并逐渐应用到了各个领域。

时至今日，在30余年的发展中，伴随着计算机软硬件的不断进步，分布式文件系统在体系结构、系统规模、性能等诸多方面也经历了较大的变化。最初的分布式文件系统（如上面提到的AFS和NFS），受限于当时网络环境、本地磁盘、处理器速度等方面的限制，它们一般以通过标准接口提供远程文件访问为目的，更多地关注访问的性能和数据的可靠性；之后，随着互联网的出现和网络中传输实时多媒体数据的需求的逐渐流行，Frangipani和Slice File system等分布式文件系统开始采用包括多级缓存策略、资源管理优化和更优的调度算法等多种方式提高性能；再然后，伴随着网络技术的发展、普及和总线带宽、磁盘速度逐渐成为了计算机系统发展的瓶颈，Global File System、General Parallel File System等分布式文件系统管理的系统规模变得更大，对物理设备的直接访问、磁盘布局和检索效率的优化、元数据的集中管理也进一步提高了性能和容量；近年来，分布式文件系统的体系结构研究已逐渐成熟，不同文件系统的体系结构、设计策略也趋于一致。然而，在细节层面，大多数分布式文件系统的设计都采用了很多特有的技术，取得了不错的性能和扩展性。

目前，包括google的GFS和Hadoop的HPFS在内的大多数分布式文件系统都采用了服务器-客户机的体系结构，其中服务器分为名字服务器（Name Server或Master）和数据服务器（Data Server或Chunk Server）两类，分别提供目录、管理服务和具体的数据存储功能。这些分布式文件系统大多使用大规模、高性能的集群服务器，为分布式的并行计算或高频的文件请求提供高速稳定的文件服务支持。同时，也出现过一些基于P2P机制的分布式文件系统，如CFS和Fragipani等。但是这些分布式文件系统的实现往往非常复杂，这使得它们或仅仅停留在理论层面，或对使用场景有较强的限制（如CFS仅提供读取服务[2]），或性能难以达到目标，总之难以进入主流市场。

Google File System

Google File System[3]（以下简称为GFS）是由Google设计并实现的一个面向大规模数据密集型应用的分布式文件系统，其可以运行在由廉价的硬件设备组成的机群上，为大量客户机提供高性能的服务。

GFS的设计目标针对Google公司特殊的使用环境进行了优化，与传统的分布式文件系统相比，这主要体现在以下几处不同：
（1）面向经常失效的组件而设计：GFS由成百上千台廉价设备组成，鉴于这些设备的数量

之多与质量之普通，事实上，设备的失效将是一个常态化的问题而非偶发问题。设备失效可能由很多原因造成，如程序的bug、人为失误或设备本身（包括硬盘、内存、网络连接器等）的损坏。无论具体的原因是什么，这意味着GFS必须通过错误侦测、灾难冗余及自动恢复等诸多机制才能提供一个稳定而高性能的服务。

（2）面向大量的巨型文件而设计：GFS中存储的文件往往非常巨大（约在数GB的量级）且数量众多，这意味着设计时的I/O操作和Block的大小都需要对其进行特别的优化。同时，必要的情况下牺牲处理小文件时的性能也是可以考虑的。

（3）面向顺序读写而设计：在GFS中，文件的修改基本上都是以在尾部追加数据的方式进行，随机写入的操作几乎不存在；同时，文件写完之后的操作通常只有顺序读取。大量的数据符合这些特性，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；或由一台机器生成、另外一台机器处理的中间数据等。这个特性意味着客户端对数据缓存是没有意义的、对追加操作的性能优化是提高写入性能的主要手段。

（4）面向协同开发的应用程序而设计：鉴于使用GFS的应用程序也是Google开发的，对于一些在分布式文件系统的设计中比较棘手的问题，可以通过与上层应用的协作来解决。例如，GFS并没有使用严格的一致性模型，而是把这部分的工作交给了具体的应用来解决，这提高了整个系统的灵活性、简化了GFS的设计、减轻了对应用程序的苛刻要求。

（5）面向大吞吐量应用而设计：GFS的目标程序绝大部分要求能高速率、大批量地处理数据，极少对单一的读写操作有严格的响应时间要求。因此，在GFS的设计中，高速稳定的网络带宽比低延迟更重要。

下面介绍GFS的结构与实现。

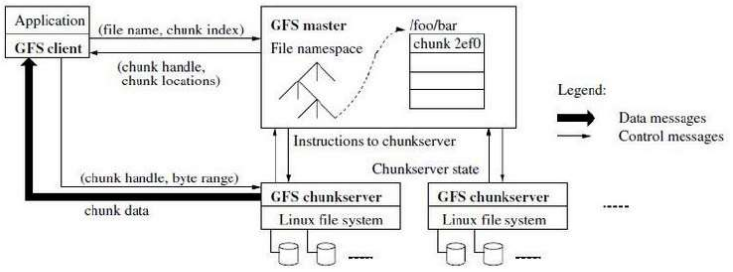


Figure 1: GFS Architecture

GFS的架构如上图[3]所示，其由一个Master节点（包括一主一备至少两个Master服务器）和多个Chunk服务器组成，并且同时被多个客户端访问。在上述的所有机器中，GFS都以用户级别的服务进程的形式存在。

GFS中的Master节点管理分布式文件系统中所有的元数据和系统范围内的活动，如，Chunk的租用、孤儿Chunk的回收、Chunk在服务器间的迁移等；Chunk服务器则承担具体数据的存储任务（以一个个Chunk的形式）。Master节点和Chunk服务器间使用心跳信息的方式进行周期性地通讯，内容包括Master节点发送指令和Chunk服务器发送其状态信息。

GFS中有3种类型的元数据，分别是文件和Chunk的命名空间、文件和Chunk的对应关系和每个Chunk副本的存放地点，它们都保存在Master服务器的内存中。其中前两种类型的元数据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并保存在本地磁盘和其它的远程Master服务器（注意到Master节点包括一主一备至少两个Master服务器）上，这

可以避免因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的位置信息，其将在启动时或新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。

GFS存储的文件都被分割成固定大小的Chunk（鉴于GFS以大文件的存取为主，Chunk的大小一般被定为64MB，使用较大的Chunk可以减少Master节点与客户端的通讯需求并降低Master节点需要保存的元数据的数量）。Master服务器会在每个Chunk被创建的时候为其分配一个不变且唯一的标识。Chunk通常被Chunk服务器以Linux文件的形式保存在本地硬盘上，并根据指定的标识和字节范围来读写块数据。

出于可靠性的考虑，每个Chunk都会被复制到多个Chunk服务器上。默认情况下，GFS采用三备份的机制，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。

GFS的客户端并没有按照POSIX等标准API的形式实现，而是以库的形式被链接到了客户程序里。尽管如此，GFS还是提供了一套类似传统文件系统的API接口函数。GFS中的文件以分层目录的形式组织，用路径名来标识；GFS也支持通常的文件系统中第常用操作，如创建文件、删除文件、打开文件、关闭文件、读文件和写文件等。此外，GFS还根据其使用特点提供了快照和记录追加操作，前者以很低的成本创建一个文件或者目录树的拷贝后者允许多个客户端同时对一个文件进行数据追加并保证每个客户端的追加操作都是原子性的。

在GFS中，客户端和Master节点的通信只限于获取元数据以寻找其应该联系的Chunk服务器，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。其通信的流程具体为：首先，客户端根据固定的Chunk大小把文件名和字节偏移转换成文件的Chunk索引；之后，其将文件名和上述Chunk索引发送给Master节点；再之后，Master节点将相应的Chunk标识和副本的位置信息发还给客户端；最后，客户端发送请求（包含了Chunk的标识和字节范围）到其中的一个（最近的）副本处。

鉴于GFS的使用方式往往是顺序读写大文件，无论是其客户端还是Chunk服务器都不需要缓存文件数据，这简化了客户端和整个系统的设计。然而，为了减轻Master节点的压力，GFS的客户端会将从其获取的元数据信息缓存一段时间，以便后续操作时可以直接和Chunk服务器进行数据读写操作。

在GFS中，文件命名空间的修改（如文件创建）都是原子性的，它们仅由Master节点控制并被操作系统日志定义了其在全局中的顺序；与之相反，文件内容的一致性保障则相对宽松，其在被修改后的状态如下表[3]所示：

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

其中*consistent*指无论从哪个副本读取，读到的数据都一样；*defined*指*consistent*且客户端能够看到写入操作全部的内容。可以看出，在并行修改的情况下，即使操作成功，GFS也不保证文件处于*defined*状态，这意味着客户端此时读到的通常不是任何一次写入操作写入的数据，而来自多个修改操作的、混杂的数据片段。此外，也可以发现，GFS面向其使用环

境更好地支持了追加操作：在并行追加的情况下，GFS可以保证写入的数据至少原子性的被追加到文件中一次，但此时GFS可能会在文件中间插入填充数据或者重复记录。对于这些问题的处理，GFS交给了使用其的程序去完成。

Cooperative File System

Cooperative File System[4] （以下简称CFS）是一个基于P2P机制的分布式文件系统，最早在MIT的一篇硕士论文中提出。CFS采用了完全的去中心化设计，可以在由多达数百万节点组成的系统中提供只读文件服务。

作为一个基于P2P机制的分布式文件系统，CFS设计时的要点和面对的主要问题与传统的服务器-客户端式的分布式文件系统有很大不同，这主要体现在：
（1）节点的对称与负载的均衡：相较于专用的服务器，P2P结构中的对等节点处理负载的能力（与意愿）无疑更低，因此，必须更加小心地根据各个节点的状态平衡负载以避免由于少数节点的崩溃影响系统整体的性能。同时，由于没有中心服务器统一收集、处理各个节点的状态，这个工作无疑更加困难。
（2）节点十分不可靠：在前面的介绍中提到过GFS是*面向经常失效的组件而设计*的，但相比于主要由于硬件损坏导致组件失效的GFS，CFS的节点更迭的速度无疑更快。研究显示，在基于P2P的分布式文件系统中，50%的节点不会连续服务超过60分钟，甚至有25%的节点不会连续服务超过10分钟[5]。因此，CFS必须付出更大的努力解决节点经常失效带来的状态维护及文件丢失等问题。
（3）可观的通讯次数与延迟：在服务器-客户端体系结构的分布式文件系统中，节点要获取一个文件通常要与名字服务器、数据服务器各进行一次通讯；但CFS没有名字服务器统一地处理请求，因此通常必须与其它的节点直接通讯，在一些结构中，通讯的次数可能和节点的个数在相同量级。当系统中节点数量较多时，这是无法忍受的，因此必须加以优化。
（4）安全性：在基于P2P机制构建的分布式文件系统中，各个存储节点都来自不同用户的自愿加入，这些用户并不来自同一个组织，也不受共同的管理。因此，这些系统除了要小心地避免由于非同步的写操作导致的不一致（由于CFS是只读的，其不必面对这个问题）外，更要通过加密等手段保证数据不被一些节点恶意地篡改。

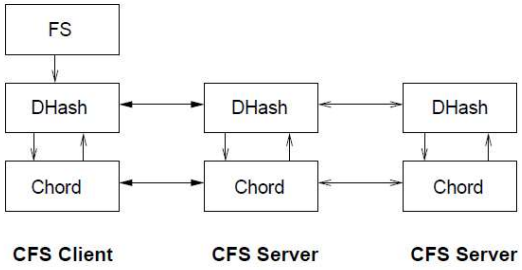
下面介绍CFS的结构与实现。

Layer	Responsibility
FS	Interprets blocks as files; presents a file system interface to applications.
DHash	Stores unstructured data blocks reliably.
Chord	Maintains routing tables used to find blocks.

CFS也使用了文件分块存储的机制，其实现由上图[4]所示，具体由三层组成：
FS层由根据DHash分配到各个节点的块的集合构成。CFS的节点将存储在其上的块理解为文件系统数据和元数据并提供了通常的文件系统只读接口以供其它应用使用。在最初的实现中，CFS的节点使用了NFS loop back server[6]将其存储的文件映射到本地的命名空间，但也有其他的实现方式可以达到相同目的[4]。
DHash层的功能包括为节点存储并取回非结构化的数据块、将数据块分散到各个节点上并维持保证稳定服务必需的缓存与冗余备份、平衡各个节点的负载。DHash层使用Chord层提供

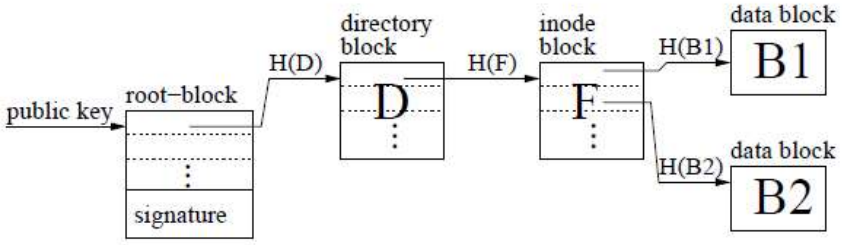
的块位置服务，也提供非同步的预取服务以降低获取块需要的延迟。
Chord层是一个类似与哈希操作的实现，负责根据块的标识符将块映射到具体的存储节点。其会为每个块与每个节点分配一个160bit的标识符，并用一个类似于分布式散列表的方式将各个标识符组织成一个环，这样每个块便可以存储在其最邻近的后继上。在每个节点上，Chord层的实现都会维护一个包含O(log(N))表项的节点查找表。这样，无论是块查找还是节点的新增或退出都可以在O(log(N))的时间内完成（其中N是节点总数）。

在CFS中，上述三层的组织方式如下图[4]所示：



与通常的定义一致的，虽然节点间彼此对等，我们仍将请求的发出方作为客户，将请求的接收方作为服务器。在CFS中，每个客户端都有完成对三层实现而服务器只需要两层实现。在客户端，DHash层被用来获取数据块，Chord层被用来定位数据库所在的具体服务器位置；与之相对的，在服务器上，DHash层被用来保存数据块（及获取其时凭借对标识符）并保证其有足够的冗余备份，Chord层则被用来与DHash层一道检查数据块的各个备份。在上图中，横向的箭头表示RPC API，纵向的箭头表示本地API。

如下图[4]所示，在CFS中，块都是按照SFSRO[7]的格式组织的，这种方式十分类似于一般的LINUX/UNIX的文件系统中块的组织方式。与GFS不同，此时块组成的不仅仅是一个个具体文件，而是整个文件系统（包括文件及其目录结构）。为了便于负载均衡并减少节点的存储压力，在CFS中，块的大小被定在了10KB的量级。



在CFS中，节点根据其当前的动作可以分为发布者（数据的提供者）和客户（数据的消费者）两种。CFS允许加入其的任何节点作为一个发布者，但为了防止恶意节点造成的损害，CFS限制了每个发布者提供的数据总量。当一个节点成为发布者时，会将其文件系统对应的块（们）插入到CFS中并以它（们）的哈希值作为相应的块标识符。然后，发布者还会上传其root块，但与其他块不同的是，root块需要用私钥签名并以公钥作为标识符。这样，一个客户可以用公钥查找一个文件系统并校验其root块的完整性；接着，便可以根据上一级块中保存的下一级块的哈希值查找下一级块并校验下一级块的完整性。

CFS是只读的，这意味着其上面的文件无法被修改。然而，发布者通过重新向CFS重新插入拥有相同公钥的新的root块并使其指向新的数据块还是可以修改CFS上的文件系统。上述操

作会带来两个问题，一是如何分辨新旧root块，二是如何回收未被指向的垃圾块。对于前面一个问题，CFS引入了时间戳的机制；对于后一个问题，CFS为每个块设置了一个有限的失效时间（发布者可以不断延长这一时间），当失效时间到达时，这个块会被保存其的节点会将删除。

Andrew File System

Andrew[8]是由美国卡耐基-梅隆（Carnegie Mellon）大学和IBM公司联合开发的一种分布式计算环境。它的主要功能是用于管理分布在网络不同节点上的文件，其能够使来自任何通过这个国家的AFS机器能够在文件一经在本地存储就能访问。Andrew文件系统的目标是要支持至少7000个工作站，同时为用户、应用程序和系统管理提供一种合适的共享文件系统。其设计的主要目标是扩大规模，即使服务器支持尽可能多的客户端。

AFS被认为是特性最丰富的非实验性DFS。它的特点体现在统一的名称空间、位置独立的文件共享、Cache一致性的客户端缓存技术和通过Kerberos的安全认证。

AFS中几个重要概念：

回调：当一个客户机存储一个文件或目录时，服务器更新用于此存储的状态信息。

单元（Cell）：是AFS一个独立的维护站点，通常代表一个组织的计算资源。一个存储节点在同一时间内只能属于一个站点；而一个单元可以管理数个存储节点。

卷（Volumes）：是一个AFS目录的逻辑存储单元，我们可以把它理解为AFS的cell之下的一个文件目录，。AFS系统负责维护一个单元中存储的各个节点上的卷内容保持一致。

挂载点（Mount Points）：关联目录和卷的机制，挂载点<---> 卷。

复制（Replication）：隐藏在一个单元之后的卷可能在多个存储节点上维护着备份，但是他们对用户是不可见的。当一个存储节点出现故障是，另一个备份卷会接替工作。

缓存一致性：

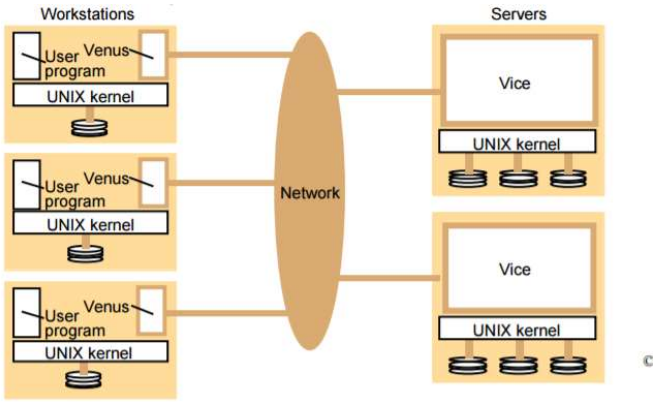
AFS使文件的更新在服务器上可见，并且在关闭更新的文件之后使旧的缓存副本无效。（在同一台机器上的文件更新例外，这种情况下更新立即可见。）当文件在不同的机器上发生更新时，afs选取最后更新的版本。

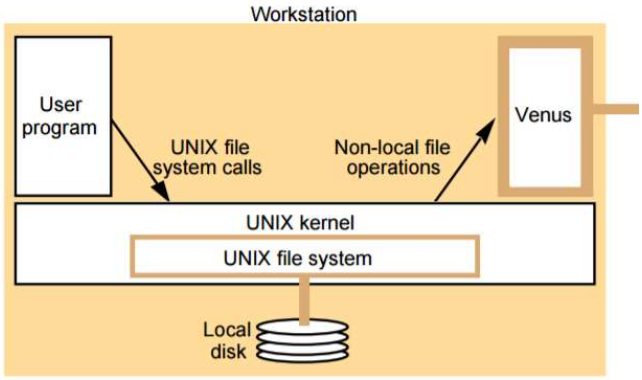
如下表：

Client$_1$			Client$_2$		Server	*Comments*
P_1	P_2	Cache	P_3	Cache	Disk	
open(F)		-		-	-	File created
write(A)		A		-	-	
close()		A		-	A	
	open(F)	A		-	A	
	read() → A	A		-	A	
	close()	A		-	A	
open(F)		A		-	A	
write(B)		B		-	A	
	open(F)	B		-	A	Local processes
	read() → B	B		-	A	see writes immediately
	close()	B		-	A	
		B	open(F)	A	A	Remote processes
		B	read() → A	A	A	do not see writes...
		B	close()	A	A	
close()		B		A	B	... until close()
		B	open(F)	B	B	has taken place
		B	read() → B	B	B	
		B	close()	B	B	
		B	open(F)	B	B	
open(F)		B		B	B	
write(D)		D		B	B	
		D	write(C)	C	B	
		D	close()	C	C	
close()		D		C	D	
		D	open(F)	D	D	Unfortunately for P_3
		D	read() → D	D	D	the last writer wins
		D	close()	D	D	

文件A在完成写操作并关闭后才建立起缓存。
对文件B,其在关闭前就可以被同在一个客户机上的p2查看。
对于C,和D。由于D关闭的较晚，因此最后服务器上保存的是D。

AFS结构：





各部分功能示例：

用户进程	UNIX kernel	Venus	Vice
打开文件	如果打开的是共享空间中的文件，传递请求到Venus。 如果打开本地文件返回文件描述符。	检查本地缓存中的文件列表。 如果不存在或没有有效的回调，将请求发送到包含文件的Vice。 将文件的副本放在本地文件系统中，在本地缓存列表中输入其本地名称，并将本地名称返回到UNIX。	将文件和回调工作站。 记录回调。
读文件	对本地文件执行正常的读操作		

写文件	对本地文件执行正常的写操作。		
关闭文件	关闭本地文件并向venus传递文件被关闭。	如果文件被修改，将其传到文件目录。	替换文件内容并向其他客户机发送回调

AFS的规模和性能：

每个服务器可以支持约50个客户端（而不是只有20个）。而且客户端性能往往与本地性能非常接近，因为在通常情况下，所有文件访问都是本地的;文件读取通常到本地磁盘缓存（可能是本地内存）。只有当客户端创建一个新文件或写入一个现有的文件需要发送存储消息到服务器，并且因此用新内容更新文件。

我们分析了对于不同大小的文件典型读写模式。小文件有Ns块;中文件Nm块;大文件有NL块。下图是其与NFS比较结果：

Workload	NFS	AFS	AFS/NFS
1. Small file, sequential read	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
2. Small file, sequential re-read	$N_s \cdot L_{mem}$	$N_s \cdot L_{mem}$	1
3. Medium file, sequential read	$N_m \cdot L_{net}$	$N_m \cdot L_{net}$	1
4. Medium file, sequential re-read	$N_m \cdot L_{mem}$	$N_m \cdot L_{mem}$	1
5. Large file, sequential read	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
6. Large file, sequential re-read	$N_L \cdot L_{net}$	$N_L \cdot L_{disk}$	$\frac{L_{disk}}{L_{net}}$
7. Large file, single read	L_{net}	$N_L \cdot L_{net}$	N_L
8. Small file, sequential write	$N_s \cdot L_{net}$	$N_s \cdot L_{net}$	1
9. Large file, sequential write	$N_L \cdot L_{net}$	$N_L \cdot L_{net}$	1
10. Large file, sequential overwrite	$N_L \cdot L_{net}$	$2 \cdot N_L \cdot L_{net}$	2
11. Large file, single write	L_{net}	$2 \cdot N_L \cdot L_{net}$	$2 \cdot N_L$

分布式文件系统间的对比

最后来简要对比我们设想的分布式文件系统与上述现有的分布式文件系统的主要异同：

（1）体系结构：我们采用了混合的客户端-服务器的体系结构和P2P体系结构，这即避免了对固定服务器的大多数依赖，又避开了CFS中复杂的状态维护、块查找等问题；

（2）实现层次：与现有的大多数分布式文件系统相同，我们在客户进程的层次提供服务，和AFS相比，我们的分布式文件系统不需要修改操作系统；

（3）备份方式：我们基于Erasure code技术实现数据的备份，这在设备经常离线的系统中可以较GFS采用的文件完整的方式大大节省存储空间；

（4）运行环境：我们的分布式文件系统在JVM虚拟机上运行，这可以使其能够运行在大多数运行不同操作系统的设备上，而GFS、AFS、CFS基本只能运行在UNIX/LINUX上。

参考文献

[0] James F. Kurose, Keith W. Ross. *Computer Networking A Top-Down Approach（Sixth Edition）*

[1] Wikipedia(in English). *DECnet*

[2] Peter Druschel, Antony Rowstron. *PAST: A large-scale, persistent peer-to-peer storage utility*

[3] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. *The Google File System*

[4] Frank Dabek. *A cooperative File System*

[5] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, Hari Balakrishnan. *A measurement study of peer-to-peer file sharing systems*

[6] David Mazieres. *A toolkit for user-level file systems*

[7] Kevin Fu, M. Frans Kaashoek, David Mazieres. *Fast and secure distributed read-only file system*

[8]Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *The Andrew File System (AFS)*