基于互联网网页的小型分布式文件系统

详细设计报告

王珺 夏昊珺 滕思洁 郑值

2017 年 6 月 25 日星期日

摘要

本报告首先提出了一个新型的基于互联网网页的小型分布式文件系统的设计并与现有的相关工作进行了对比，接着讨论了该分布式文件系统实现的可行性、整体的体系结构、需要使用到的技术、表现出的优势与各模块的实现方案。最后，本文分析了该分布式文件系统的性能特点。

本项目已在 GitHub 上开源，项目网站为
https://github.com/IngramWang/DFS_OSH2017_USTC

目录

一、项目背景

……5

应用环境与设计愿景

……5

相关工作

……6

二、可行性分析

……14

客户端服务程序

……14

服务器程序

……15

web 服务程序

……17

动态网页设计

……20

三、总体设计与技术

……23

系统结构

……23

技术路线

……24

四、创新点与优势

……26

基于 Erasure Code 的备份机制

……26

可用性

……26

充分利用闲置设备

……26

存储容量与可扩放性

……27

兼容性

……27

五、系统组成

……28

六、模块设计与实现

……29

客户端启动模块

……29

客户端文件分块模块

……29

客户端文件夹监控模块

……30

客户端网络链接模块

……30

服务器启动模块

……31

服务器控制链接模块

……31

服务器数据链接模块

……31

服务器数据库模块

……32

客户端-服务器间的网络通信

……35

Tomcat 服务程序

……37

登录注册模块

……38

网页主界面

……39

文件目录展示交互模块

……42

文件下载模块

……43

七、系统性能分析

……46

文件可用概率分析

……46

上传速度分析

……46

下载速度分析

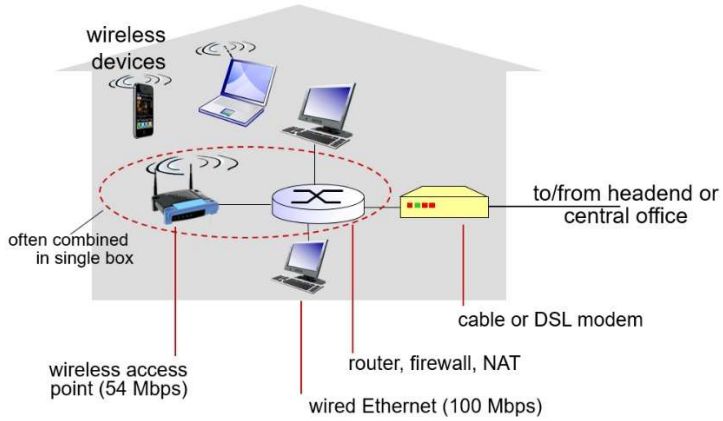
……47

一、项目背景

应用环境与设计愿景

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，即使是传统的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）传统的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）传统的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上；

同时，曾经流行的网盘如今随的监管与盈利变得岌岌可危。2016 年，包括 360 网盘、新浪微云、金山快盘等一大批免费网盘相继关闭，余下的网盘则或限速或收费，这为我们利用网盘进行设备间的文件同步敲响了警钟。

面对这种现状，我们试图自行设计一个新型的家用的分布式文件系统，在上述应用环境中解决文件的分散存储并提供比免费网盘更好的服务。在我们的设计中，我们将利用所有安

装了客户端的机器共享存储空间存储文件碎片，利用一个专用的服务器维持系统的状态并协调各个客户机处理请求，利用互联网网页访问整个文件系统。这样，只要是一个安装了客户端的机器便都可以通过贡献存储空间的方式加入分布式文件系统；只要是一个可以上网、安装了网页浏览器的电脑就都可以访问我们的分布式文件系统。这使得我们的分布式文件系统具备较为便捷的访问方式与利用闲置设备的潜力。

相关工作

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。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），或性能难以达到目标，总之难以进入主流市场。

以下分析集中典型的分布式文件系统并与我们的设计进行对比。

Google File System

Google File System（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。

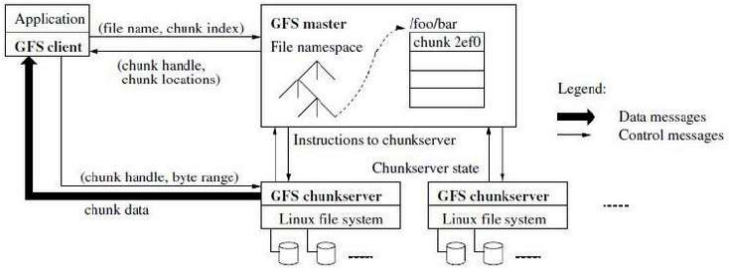


Figure 1: GFS Architecture

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（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。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。

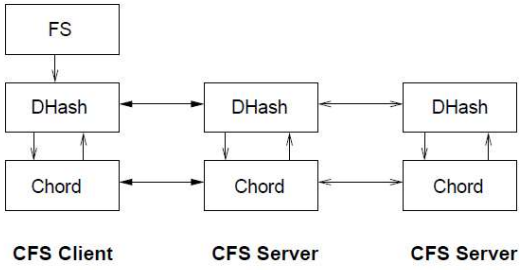
下面介绍CFS的结构与实现。

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

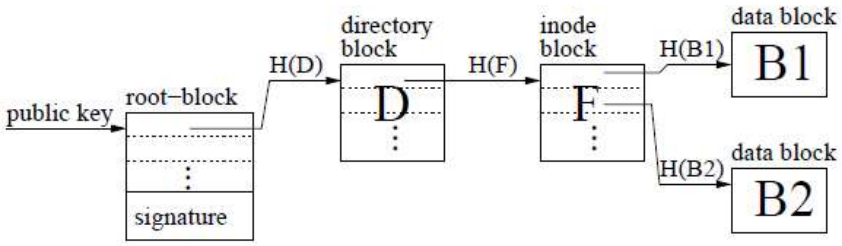
有其他的实现方式可以达到相同目的。
DHash层的功能包括为节点存储并取回非结构化的数据块、将数据块分散到各个节点上并维持保证稳定服务必需的缓存与冗余备份、平衡各个节点的负载。DHash层使用Chord层提供的块位置服务，也提供非同步的预取服务以降低获取块需要的延迟。
Chord层是一个类似与哈希操作的实现，负责根据块的标识符将块映射到具体的存储节点。其会为每个块与每个节点分配一个160bit的标识符，并用一个类似于分布式散列表的方式将各个标识符组织成一个环，这样每个块便可以存储在其最邻近的后继上。在每个节点上，Chord层的实现都会维护一个包含O(log(N))表项的节点查找表。这样，无论是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。

如下图所示，在CFS中，块都是按照SFSRO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：

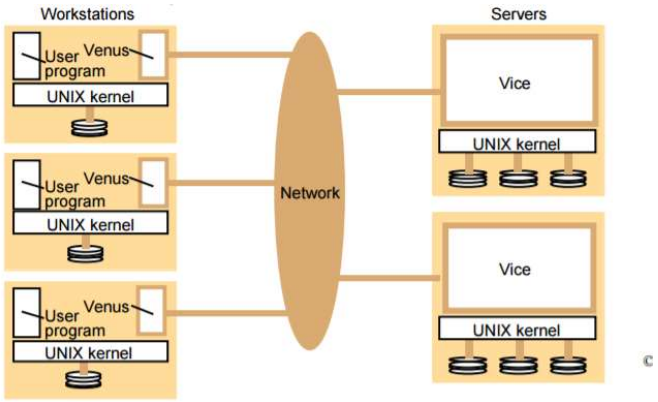
AFS使文件的更新在服务器上可见，并且在关闭更新的文件之后使旧的缓存副本无效。（在同一台机器上的文件更新例外，这种情况下更新立即可见。）当文件在不同的机器上发生更新时，afs选取最后更新的版本。

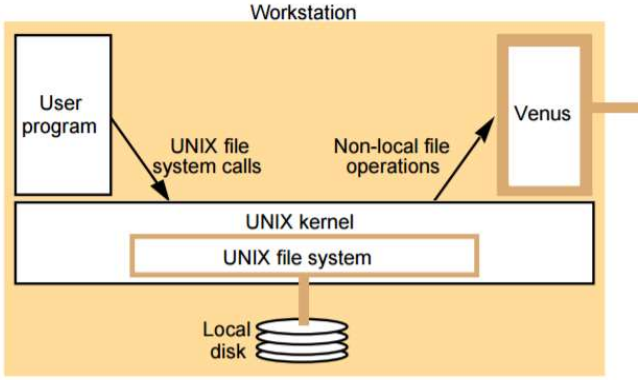
如下表：

P_1	$Client_1$ P_2	Cache	P_3	$Client_2$ Cache	Server Disk	*Comments*
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）设计目的：当前包括 GFS、AFS 在内的大多数商业分布式文件系统均是面向大型企业或团体设计的，强调高性能与高可靠，而我们的分布式文件系统均是面向家庭与小微企业，强调低成本与结构简单。

（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）网络环境：目前，大多数高性能分布式文件系统只支持在局域网上运行，而我们的分布式文件系统可以在互联网上运行。

二、可行性分析

在这一部分中，我们将分别阐述客户端、服务器、网页服务器以及动态网页四者的设计与初步实现并讨论这些实现方案的可行性。

客户端服务程序

本项目的 java 客户端是一个服务程序，与服务器基于 tcp 协议的 socket 通信建立连接。客户端通过配置文件来进行配置。配置文件会选择要进行共享的文件列表，并提供服务器的 IP 接口。

我们采用一般的文本文件作为配置文件。因此，通常的文件读写函数就可以实现配置的读取与更新。

Java 客户端提供以下几种功能：

1. 获取本地目录：

服务器需要知道本地机器参与共享的文件列表
使用 java.io.File 类即可操作获取

```
class FileList
{
    File file;
    String[] fileList;

    public FileList(String fileRoute)
    {
        //通过指定路径，构造 file 对象
        file = new File(fileRoute);
        //获取指定路径下的文件列表，保存至 fileList 中
        fileList = file.list();
    }
}
```

2. 对文件进行分块

当服务器发来需要分块的文件时，客户端需要对文件应用 erasure code 算法进行分块，分块算法已经基于 backblaze 开源实现。

3. 向服务器发送文件碎片、接收服务器传来的碎片

当文件分块完成后，客户端便向服务器发出 post 请求，服务器开通数据传输端口，客户端将文件碎片发送向客户端。若有服务器发出获取碎片请求，客户端则将存在本地的碎片发向客户端。

收发文件已经有很成熟的实现方法，以发送一张照片为例：

```
public class Main {
  public static void main(String[] args) throws IOException {
```

```
    ServerSocket servsock = new ServerSocket(12345);
    File myFile = new File("qq.jpg");
    while (true) {
      Socket sock = servsock.accept();
      byte[] mybytearray = new byte[(int) myFile.length()];
      BufferedInputStream bis = new BufferedInputStream(new
FileInputStream(myFile));
      bis.read(mybytearray, 0, mybytearray.length);
      System.out.println(mybytearray.length);
      OutputStream os = sock.getOutputStream();
      os.write(mybytearray, 0, mybytearray.length);

      os.flush();
      sock.close();
    }
```

4. 响应服务器删除本地文件

```
public void testDelete(){
        try {
            ClientGlobal.init(conf_filename);

            TrackerClient tracker = new TrackerClient();
            TrackerServer trackerServer = tracker.getConnection();
            StorageServer storageServer = null;

            StorageClient storageClient = new StorageClient(trackerServer,
                    storageServer);
            int i = storageClient.delete_file("group1",
"M00/00/00/wKgRcFV_O8OAK_KCAAAA5fm_sy874.conf");
            System.out.println( i==0 ? "删除成功" : "删除失败:"+i);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

服务器程序

服务器程序是分布式文件系统的核心组件，其管理着整个分布式文件系统的元数据并支持客户端服务程序和动态网页与分布式文件系统的交互。

服务器程序的功能主要有：

连接类	数据管理类
接收、回复、转发服务请求与控制信息	维护云共享文件索引
收发数据（文件碎片）	维护各个客户端的状态信息

	记录、处理当前等待响应的文件请求

服务器程序包括两个常开的线程用于监听两个不同的 TCP 端口分别用来回应服务请求和转发数据，对于每个 TCP 链接，服务器进程将创建一个线程进行处理，这将保证服务器可以在处理一个请求时不至于失去响应。

Java 提供了接口 Runable 供多线程编程使用，实现 Runable 的类只需重写 run 方法便可以在类中实例化一个线程（Thread 类）对象，并通过 start 方法将这个线程提交给 JVM 的线程调度队列，形如：

Main.java

```
public class Main{
    public static void main(String[] args){
        new HelloThread("thread #1").start();
        new HelloThread("thread #2").start();
        new HelloThread("thread #3").start();
    }
}
```

HelloThread.java

```
class HelloThread implements Runnable{

    private String name;

    public HelloThread(String name) {
        this.name=name;
    }

    public void run(){
      System.out.println("helloworld from "+name);
    }

    public void start() {
        Thread thread;
        thread = new Thread(this,name);
        thread.start();
    }
}
```

为了避免复杂的线程间同步与内存管理，本报告设计的分布式文件系统采用 MySQL 数据库保存文件索引信息和各客户端的状态信息。MySQL 原本是一个开放源代码的关系数据库管理系统，原开发者为瑞典的 MySQL AB 公司。2008 年 MySQL AB 公司被昇阳微系统收购，2009 年，甲骨文公司收购了昇阳微系统公司，故当前 MySQL 为 Oracle 旗下产品。同其他数据库一样，MySQL 可以通过标准查询语言（SQL）进行查询。

为了避免在数据库上进行漫长的线性查找，MYSQL 支持为数据库添加索引。引擎为 InnoDB 或 MyISAM 的数据库支持基于 B-Tree 的索引，其可以将查询时间减少到 O(log(n))；引擎为 MEMORY 的数据库支持基于 Hash 的索引，其可以将查询时间减少到 O(1))。添加索引可以通过 SQL 命令

alter table *table_name* add index *index_name* using {hash, BTREE} (*column_name*);

执行；具体数据库引擎的选择则还要考虑到数据库的大小（如 MEMORY 引擎的数据库大小不能超过 RAM 大小）和并发性的要求。

MySQL 数据库支持并发操作，为了在并发操作中保持数据的一致性，MySQL 提供锁机制与

事务机制。MySQL 的锁机制包括表级锁（锁定整张表）和行级锁（锁定特定行，并发程度更高）；MySQL 的事务机制保证一系列的 SQL 可以被原子的执行且不被其他事务干扰。不同引擎的表对并发性的支持是不同的，如 InnoDB 实现了行级锁而 MyISAM 存储引擎不支持行级锁。MySQL 中 InnoDB 引擎支持具有事务、回滚和崩溃修复能力的事务安全型表，对并发性有最好的支持。因此，考虑到索引的速度、数据库大小的支持与对并发性的支持，本报告中设计的分布式文件系统采用基于 InnoDB 引擎的数据库。

Java 提供了 Java 数据库连接（JDBC）作为规范客户端程序如何来访问数据库的应用程序接口并提供了诸如查询和更新数据库中数据的方法。只需加载 MySQL 的驱动，就可以在 Java 程序中链接 MySQL 数据库并通过 SQL 查询（或修改）数据库中的内容。

下面介绍服务器程序提供的功能。

接收、回复、转发服务请求与控制信息：服务器与客户端通过非持续的 TCP 链接传递 HTTP 报文来交换信息。

Java 提供了 ServerSocket 类和 Socket 类来进行基于 TCP 链接的网络通信。ServerSocket 类可以用于在服务器端创建一个欢迎套接字，Socket 类可以在客户端或服务器端创建链接套接字。

收发数据（文件碎片）：由于各个客户端之间未必可以直接连接（如客户端可能隐藏在 NAT 后），当有文件请求时服务器需要先收集文件碎片再根据请求的类型进行处理。

TCP 链接实现了可靠数据传输，因此利用 TCP 链接进行文件碎片的传输是十分简单的，只需要同时打开 socket 与文件的输入/输出流并在一侧写入另一侧写出即可。

维护云共享文件索引

云共享文件索引以 MySQL 数据库中的一系列 table 的形式保存。每台共享了文件的电脑均在数据库中对应两个 table，其一记录了这台电脑上共享的每个文件的唯一标识符和其逻辑位置；其二记录了这台电脑上各个文件的碎片的物理位置和其唯一标识符。

维护各个客户端的状态信息

客户端状态信息以 MySQL 数据库中的一个 table 的形式保存。这个 table 中包括了客户端的唯一标识符，在线情况，剩余空间及当前复杂维持与这个客户端的控制链接（TCP 链接）的线程的编号。

记录、处理当前等待响应的文件请求

由于服务器并不时时连接各客户端，必须在 MySQL 数据库中使用一个 table 记录当前网页提出的文件请求。当服务器收到来自客户端的心跳连接时，其将查询文件请求表，如果发现有对客户端上文件的请求，则在回复心跳连接时将文件请求发给客户端并令其（通过服务器）将文件发往请求方。

web服务程序

网页使用 JS：

JavaScript 是 Web 的编程语言，所有现代的 HTML 页面都使用 JavaScript。
使用 JS 可以实现网页背后的逻辑，让网页在用户的点击下产生一系列反应。

使用 JS 时包含 JQuery 库：

jQuery 是一个 JavaScript 库，"写的少，做的多"，极大地简化了 JavaScript 编程。

网页 UI 设计采用 Bootstrap 框架：

Bootstrap，来自 Twitter，是目前最受欢迎的前端框架。Bootstrap 是基于 HTML、CSS、JAVASCRIPT 的，它简洁灵活，使得 Web 开发更加快捷。

1. 考虑到它适用于快速开发+可视化开发+提供可重用组件：进度条等+UI 美观，所以值得使用。
2. 还有一点是，它秉承移动设备优先。Bootstrap 的响应式 CSS 能够自适应于台式机、平板电脑和手机，可以自动适应于屏幕大小调节网页布局。
3. 浏览器支持：所有的主流浏览器都支持 Bootstrap。

网页和服务器的交互采用 AJAX：

AJAX = 异步 JavaScript 和 XML（Asynchronous JavaScript and XML）
AJAX 是与服务器交换数据的技术，它在不重载全部页面的情况下，实现了对部分网页的更新。

jQuery 提供多个与 AJAX 有关的方法。（如果没有 jQuery，AJAX 编程还是有些难度的。编写常规的 AJAX 代码并不容易，因为不同的浏览器对 AJAX 的实现并不相同。这意味着你必须编写额外的代码对浏览器进行测试。不过，jQuery 团队为我们解决了这个难题，我们只需要一行简单的代码，就可以实现 AJAX 功能。）

jQuery AJAX 方法

jQuery AJAX 方法

AJAX 是一种与服务器交换数据的技术，可以在不重新载入整个页面的情况下更新网页的一部分。
下面的表格列出了所有的 jQuery AJAX 方法：

方法	描述
$.ajax()	执行异步 AJAX 请求
$.ajaxPrefilter()	在每个请求发送之前且被 $.ajax() 处理之前，处理自定义 Ajax 选项或修改已存在选项
$.ajaxSetup()	为将来的 AJAX 请求设置默认值
$.ajaxTransport()	创建处理 Ajax 数据实际传送的对象
$.get()	使用 AJAX 的 HTTP GET 请求从服务器加载数据
$.getJSON()	使用 HTTP GET 请求从服务器加载 JSON 编码的数据
$.getScript()	使用 AJAX 的 HTTP GET 请求从服务器加载并执行 JavaScript
$.param()	创建数组或对象的序列化表示形式（可用于 AJAX 请求的 URL 查询字符串）
$.post()	使用 AJAX 的 HTTP POST 请求从服务器加载数据
ajaxComplete()	规定 AJAX 请求完成时运行的函数
ajaxError()	规定 AJAX 请求失败时运行的函数
ajaxSend()	规定 AJAX 请求发送之前运行的函数
ajaxStart()	规定第一个 AJAX 请求开始时运行的函数
ajaxStop()	规定所有的 AJAX 请求完成时运行的函数
ajaxSuccess()	规定 AJAX 请求成功完成时运行的函数
load()	从服务器加载数据，并把返回的数据放置到指定的元素中
serialize()	编码表单元素集为字符串以便提交
serializeArray()	编码表单元素集为 names 和 values 的数组

通讯格式：

网页和后台 通信发送的内容是 JSON 字符串（JavaScript JSON，英文全称 JavaScript Object Notation）：

JSON 是用于存储和传输数据的格式，通常用于服务端向网页传递数据 ，是一种轻量级的数据交换格式。

JSON 是独立的语言，而且易于理解。

相关函数

函数	描述
JSON.parse()	用于将一个 JSON 字符串转换为 JavaScript 对象。
JSON.stringify()	用于将 JavaScript 值转换为 JSON 字符串。

Apache+Tomcat+Servlet+java web 应用

1. Web 服务器采用：Apache

Apache HTTP 服务器是一个模块化的服务器，可以运行在几乎所有广泛使用的计算机平台上。其属于应用服务器。Apache 支持支持模块多，性能稳定，Apache 本身是静态解析，适合静态 HTML、图片等，但可以通过扩展脚本、模块等支持动态页面等。

（Apche 可以支持 PHPcgiperl,但是要使用 Java 的话，你需要 Tomcat 在 Apache 后台支撑，将 Java 请求由 Apache 转发给 Tomcat 处理。）

2. 配合 Apache 使用 Tomcat（应用服务器）

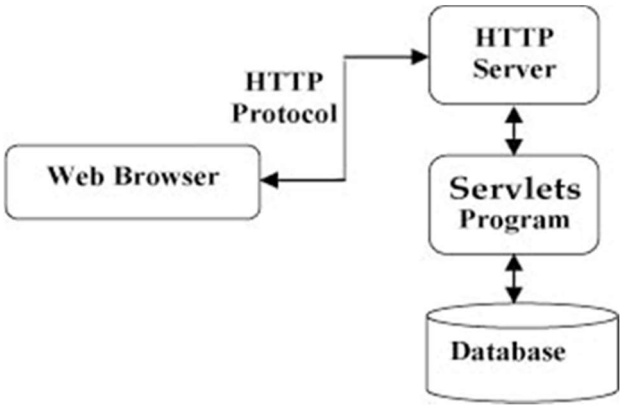
Tomcat 是应用（Java）服务器，它只是一个 Servlet(JSP 也翻译成 Servlet)容器，可以认为是 Apache 的扩展，但是可以独立于 Apache 运行。

3. 使用 Servlat 实现 web 应用程序和其他进程间的通讯

Java Servlet 是运行在 Web 服务器或应用服务器上的程序，它是作为来自 Web 浏览器或其他 HTTP 客户端的请求和 HTTP 服务器上的数据库或应用程序之间的中间层。

使用 Servlet，您可以收集来自网页表单的用户输入，呈现来自数据库或者其他源的记录，还可以动态创建网页。

只需要在我们的 web 应用程序中导入一些 java 库就可以通过库函数实现对 http 端口的请求监听和响应。



Servlet 执行以下主要任务：
读取客户端（浏览器）发送的显式的数据。这包括网页上的 HTML 表单，或者也可以是来自 applet 或自定义的 HTTP 客户端程序的表单。
读取客户端（浏览器）发送的隐式的 HTTP 请求数据。这包括 cookies、媒体类型和浏览器能理解的压缩格式等等。
处理数据并生成结果。这个过程可能需要访问数据库，执行 RMI 或 CORBA 调用，调用 Web 服务，或者直接计算得出对应的响应。
发送显式的数据（即文档）到客户端（浏览器）。该文档的格式可以是多种多样的，包括文本文件（HTML 或 XML）、二进制文件（GIF 图像）、Excel 等。
发送隐式的 HTTP 响应到客户端（浏览器）。这包括告诉浏览器或其他客户端被返回的文档类型（例如 HTML），设置 cookies 和缓存参数，以及其他类似的任务。

4. 服务器端网络服务程序基于 JAVA
我们已有基于 java 实现的一些模块，所以使用 java 可以减少任务量（不用重写模块）；
Java 的网络功能很强大，写起来也方便；
客户端服务程序使用 java，两个互相交互的服务程序实现上尽量保持统一。

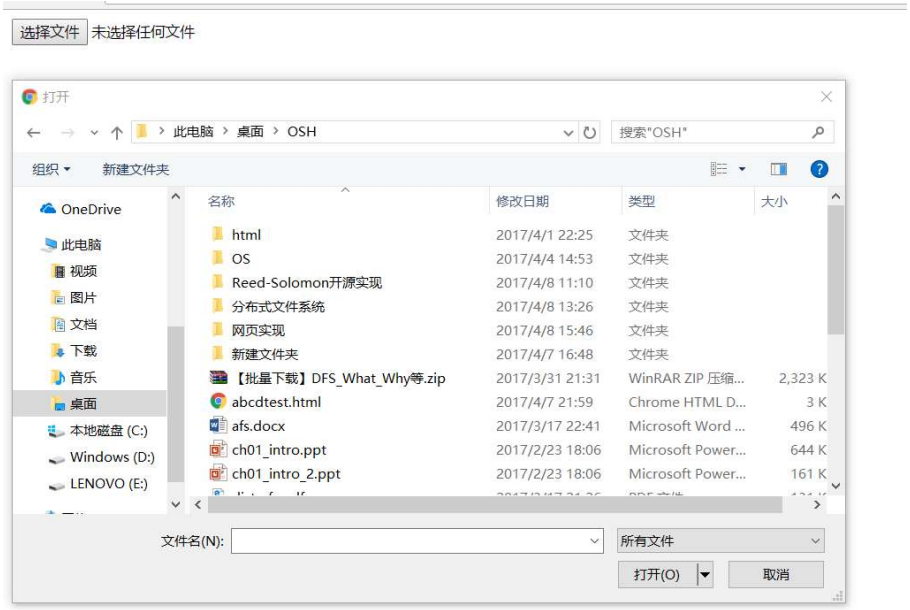
动态网页设计

网页对文件系统的支持：
JavaScript 对本地文件操作的支持：
Js 提供如下文件操作函数：
CopyFile() 复制文件
CopyFolder() 复制目录
CreateFolder() 创建新目录
CreateTextFile() 生成一个文件
DeleteFile() 删除一个文件
DeleteFolder() 删除一个目录
DriveExists() 检验盘符是否存在
Drives 返回盘符的集合
FileExists() 检验文件是否存在
FolderExists 检验一个目录是否存在
GetAbsolutePathName() 取得一个文件的绝对路径
GetBaseName() 取得文件名
GetDrive() 取得盘符名
GetDriveName() 取得盘符名
GetExtensionName() 取得文件的后缀
GetFile() 生成文件对象
GetFileName() 取得文件名
GetFolder() 取得目录对象
GetParentFolderName 取得文件或目录的父目录名
GetSpecialFolder() 取得特殊的目录名
GetTempName() 生成一个临时文件对象
MoveFile() 移动文件

MoveFolder() 移动目录

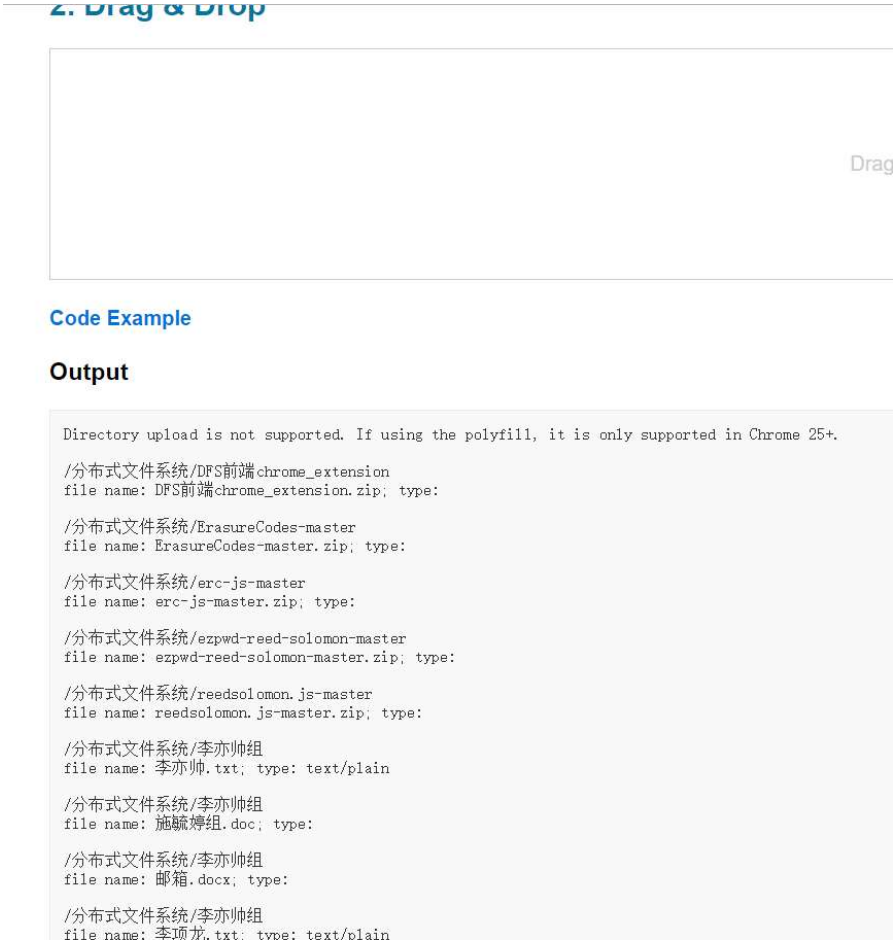
可以利用 js 实现在网页上获取本地文件目录，并上传本地文件。我们尝试了利用 js 选择并打开本地文件：

如图：



可以利用网页获取本地文件目录：

如：



Eursure code 的 Javascript 实现：

Reed-solomom 的开源 js 实现库 erc-js：

###Classes

ReedSolomon

Public-facing interface providing encode/decode functionality

ReedSolomon.Codec

Implements the Reed-Solomon error correction algorithm

ReedSolomon.GaloisField

Implements a Galois field GF(p^n) over p = 2

ReedSolomon.Utils

Implements string and array manipulation methods

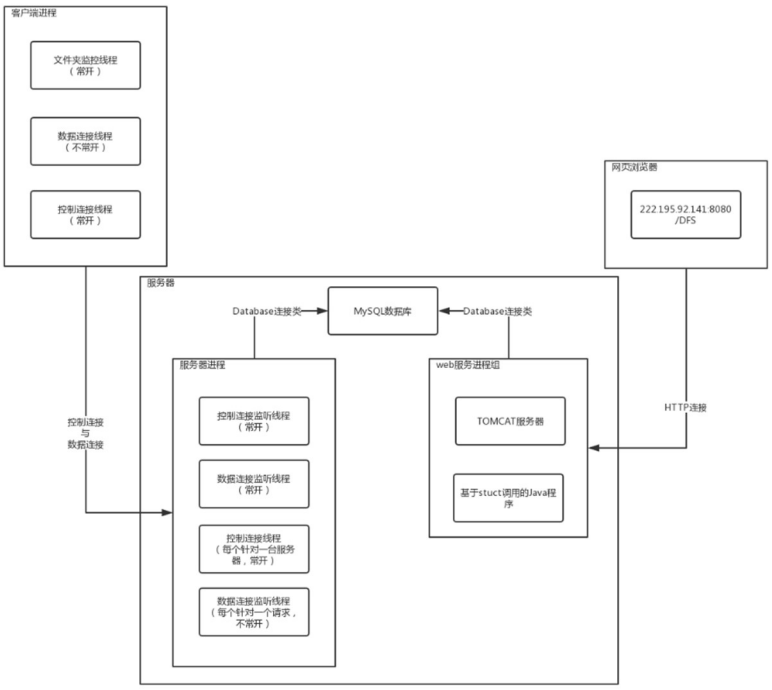
库的使用：

```
// n is the length of a codeword
var rs = new ReedSolomon(n);
var enc = rs.encode('hello world');
var msg = rs.decode(enc);
```

三、总体设计与技术

系统结构

我们设计的分布式文件系统的整体结构下图所示：



该分布式文件系统的程序主要包括客户端程序、服务器程序与web服务程序三部分，其中客户端程序运行在客户主机上，服务器程序与web服务程序都运行在服务器上。访问该分布式文件系统可以利用网页浏览器通过访问互联网网页的方式进行，故不需要特殊程序的支持。

我们设计的客户端有三个主要的功能。首先，其必须要定时联系服务器并处理服务器发给它的控制报文，这样服务器才能了解各个客户端的状态并且保持与各个客户端的连接。其次，客户端要能扫描本地共享文件夹并上传其中的新文件。第三，客户端要能根据服务器指示保存、传输、删除保持在本地文件块，这样服务器才能响应网页的文件请求。为了达成这些功能，客户端进程包括两个常开的线程：文件夹监控线程（用于监控指定的文件夹并上传其中的数据）与控制链接线程（用于与服务器交换控制报文）。此外，在需要时，客户端还会开启数据链接线程以与服务器交换数据。

我们的服务器程序也有三个主要功能。首先，它要能联系并管理各个客户端。其次，其需要根据客户端的上传请求与网页的下载请求与各个客户端合作完成文件碎片的分发与收集。最后，服务器将要维持整个分别式文件系统的状态信息从而保证服务的可用性。为了完成这个目的，服务器进程使用两个线程分别监听数据连接与控制链接的欢迎套接字。当有新连接时，服务器进程还将相应的创建新线程处理这个连接。

为了保持兼容性，我们设计的客户端程序与服务器程序都使用Java语言编写。

Web服务提供了基本的基于http请求响应服务，同时还部署了一系列配合该系统的java服务器应用程序。
采用的主要布局如下：

- Tomcat WEB应用服务器
- Struts2 动态网站网站应用调度框架
- BOOTSTRAP网页主题
- JQery+AJAX异步C/S通讯+JSON+MySql

在本系统中，服务器进程与web服务进程组、服务器进程的各个线程之间均通过MySQL数据库交换数据。

技术路线

本次工程主要需要完成的任务是客户端java程序 + 服务器JAVA程序 + web服务 + web服务程序

我们主要分成了两组人员，第一组主要负责实现客户端java和服务器java，并实现两者间的通讯和文件传输；
另一组实现了web服务和网页的实现，搭建了一个网站。

两组之间的融合靠的是mysql数据库，二者通过数据库传达各个子系统的状态信息和请求。当整个系统出错时，我们通过数据库这个中间层的信息就可以初步判断错误出现在哪一组。

我们的分布式文件系统主要使用到了以下技术：

网络数据传输：为了使分布在不同地点的各种设备可以共同维护、使用我们设计的分布式文件系统，基于网络的数据交换是不可避免的。目前在世界范围内应用最为广泛的网络是因特网，其使用TCP/IP协议互联了世界上数以亿计的各类设备。因此，我们设计的分布式文件系统将使用因特网进行数据的传输。

JVM：我们设计的分布式文件系统使用Java编程，因此其将运行在Java虚拟机（JVM）上。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。JVM是Java语言实现与平台的无关性的关键：一般的高级语言如果要在不同的平台上运行需要编译成不同的目标代码，而JVM屏蔽了与具体平台相关的信息，使得Java语言只要编译为字节码就可以在多种平台上不加修改地运行。

Erasure Code：我们设计的分布式文件系统使用Erasure Code来实现副本管理。Erasure Code（又称纠删码）是一种编码技术，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。

JQery：″写的少，做的多″，极大地简化了 JavaScript 编程

AJAX：不重载全部页面，实现对部分网页的更新

JSON：轻量级的数据交换格式

- BOOTSTRAP 网页主题：基于 HTML、CSS、JAVASCRIPT
- 适用于快速开发+可视化开发+提供可重用组件
- 响应式 CSS 能够自适应于不同尺寸电脑和手机
- 所有的主流浏览器都支持 Bootstrap

Struts2 动态网站：

- Tomcat 基础上
- 采用 MVC 设计模式
- 控制器从视图读取数据，控制用户输入，向模型发送数据
- 管理复杂的应用程序，可以在一个时间内专门关注一个方面
- 简化了分组开发，不同开发人员可同时开发视图、控制器逻辑和业务逻辑

Tomcat WEB 应用服务器

- 本身支持一般的 http 服务
- 提供 JSP 的 JAVA 运行环境和 JavaBean 的运行环境
- 响应浏览器请求，调用服务器端 java 函数
- 在以上基础上，搭建动态网站

四、创新点与优势

在这一部分中，我们会从备份机制、可用性、充分利用闲置设备、存储容量与备份机制和兼容性五个方面将本报告中设计的分布式文件系统与目前已有的分布式文件系统进行对比并分析其具有的优势与创新。

备份机制：Erasure code

为了提供容错功能并保证在数据节点（无论是传统的数据服务器还是P2P机制中的对等方）崩溃或离线时数据的可用性，大多数分布式文件系统都实现了备份机制，即通过保存文件的多个副本保证文件的安全性。

目前包括GFS、HDFS在内的主流分布式文件系统往往采用了完全复制副本，即将一个文件（文件块）完整地复制为多份完全相同的副本并保存在不同的位置。这种方式无疑能提供文件的安全性，同时其也允许客户端获取离其最近的副本以加快访问速度，但完全复制副本的缺点也是明显的：其消耗了过多的存储空间，以大多数分布式文件系统默认的3备份为例，这意味着系统只能提供其物理存储空间1/3容量的存储服务。

Erasure Code（中文名为抹除码或纠删码）是一种编码技术，它可以将n个原始数据块编码为n+m个数据块，并能通过在n+m个数据块任取n个进行解码还原出原始数据。一般的，记编码效率$\mathrm{r}=\frac{n+m}{n}$,表示编码后总占用空间与原始数据大小的比。对于常见的应用，一般把m设置为$\frac{n}{2}$左右，即r约为1.5，显然这个结果要远好于完全复制副本的实现方式。

我们将使用 Erasure Code 作为我们的数据备份方式。目前已经有了一些基于 Erasure Code 的存储系统设计，但正如前文所述，主流的分布式文件系统大多现在仍使用完全复制副本的方式。

可用性：结合对等方主机与网盘服务

除了备份机制，为了进一步提高文件的可用性，在未来，我们将会把目前已十分成熟的网盘资源整合进本文设计的分布式文件系统，这种实现方式在目前常见的分布式文件系统中尚未出现。

由于网盘往往由专业的数据存储企业运营、维护，其本身也使用了一套数据备份机制，故其几乎可以保证总是可用。因此，网盘的加入能够大大地提高分布式文件系统中文件的可用性。同时，与单纯依靠网盘进行存储相比，我们的设计只需在网盘中存储文件的部分块（而非整个文件），这意味着我们的设计可以利用相同空间的网盘资源存储更多的文件，进而减少了租赁网盘空间的成本。

充分利用闲置设备

随着网络技术的进步与电子产品的迭代，目前我们有大量空闲的带宽与闲置的主机，它们可以存储海量的文件却难以得到应用。在我们的分布式文件系统中，只需安装客户端，客户机就可以分享存储空间且几乎不需要人工的维护。因此，我们的设备有能力利用闲置设备的存储空间。

存储容量与可扩放性

通过海量客户机的加入，我们的分布式文件系统有能力以较低的成本提供巨大的存储容量。此外，我们的分布式文件系统在运行过程中允许主机可以动态地加入或退出，因此其良好的可扩放性。

兼容性：基于 Java 的客户端与基于网页的访问

以GFS为代表的主流分布式文件系统往往面向专业用户或程序设计者，其提供的访问接口往往以库函数或POSIX标准的API为主。由于缺少专业的技术人员，一般的家庭、非IT领域的小微企业应用这类分布式文件系统是非常困难的。

我们设计的分布式文件系统提供了包括传统的API、客户端软件在内的多种访问方式。我们创新性的提供了网页访问方式，允许用户使用任何一台连接互联网的主机如同打开商业网盘一样打开我们的分布式文件系统。

为了尽可能追求效率，包括GFS、NFS、FastDFS在内的大多数分布式文件系统往往选择使用C/C++进行实现，有的甚至还会要求修改操作系统以实现更好的支持，这意味着这些分布式文件系统难以在运行不同不同的操作系统或指令集的主机集群中使用。

随着JIT技术的发展，目前Java的执行效率已得到了巨大的提升，同时，Java基于JVM的运行模式使其可以兼容各种体系结构的主机。我们的客户端完全使用Java编写，这使得任何可以允许Java虚拟机的主机理论上都可以运行我们的客户端。

同时，我们的分布式文件系统支持网页访问，这使得几乎任何能上网的设备都可以访问系统中的数据。

五、系统组成

本分布式文件系统的模块组成如下表示：

客户端程序	服务器程序	web 服务进程组
启动与异常处理模块	启动模块	Tomcat 服务程序
文件分块模块	控制连接模块	登录注册模块
服务器连接模块	数据连接模块	网页主界面
文件夹监控模块	数据库模块	文件目录展示交互模块
		文件下载模块

六、模块设计与实现

客户端启动模块

客户端启动模块是客户端的主模块，其在启动后将先读取配置文件，然后根据配置文件的设置启动网络链接模块中的控制链接线程与文件夹监控线程，最后其调用 wait 函数进入等待状态。当有线程出错时，其将被唤醒并结束所有线程的执行。

客户端的配置文件就是一般的文件文件，其格式如下：

{服务器 IP}
{服务器控制链接端口}
{服务器数据链接端口}
{客户端 ID}
{碎片文件夹路径（用于保存服务器分配来的碎片）}
{临时文件夹路径（用于在上传过程中保存本地文件的碎片，上传完成后将被清空）}
{需要监控的上传文件夹数量}
{上传文件夹 1 路径}
{上传文件夹 1 中的文件在分布式文件系统中的 path}
{上传文件夹 2 路径}
{上传文件夹 2 中的文件在分布式文件系统中的 path}
。。。

易于发现，只需要用 Scanner 类处理这个文件并每次使用 nextline 函数读取下一行进行相应处理即可。

值得一提的是启动模块监控其他线程是否出错并在发生异常时终止所有线程的机制，这里涉及到线程间的同步。启动模块利用 SYMITEM 类完成了这个任务，SYMITEM 类使用了 Java 的 synchronized 保留字与 notifyAll、wait 函数。启动模块首先创建一个 SYMITEM 类的实例并将其传给所有其创建的线程。当启动模块完成了其工作后，其调用 SYMITEM 类的 waitChange 函数进入等待状态。waitChange 函数是一个被 synchronized 关键字修饰的函数，其将进一步调用 wait 函数并在被唤醒时检查 SYMITEM 类的私有变量 status 是否与调用时传入的 status 相同，如相同，重新调用 wait 函数进入等待状态。当一个线程出错时，其将调用 setStatus 函数设置 status 的值，这个函数也将进一步调用 notifyAll 函数唤醒所以在 waitChange 函数中等待的线程。

客户端文件分块模块

客户端文件分块模块对文件应用 erasure code 算法进行分块，其整体采用了 backblaze 公司提供的开源实现，只是根据 SampleDecoder 类与 SampleEncoder 类重写了 Decoder 类与 Encoder 类以实现文件的还原与分块。

Decoder 类只包括一个静态函数 decode 用以实现将文件碎片还原为文件的工作，decode 包括 4 个参数，分别为 File 类型的 shardsFolder 表示碎片所在的文件夹、File 类型的

decodedFile 用来表示碎片要生产的还原文件、int 类型的 fid 表示文件的 ID、int 类型的 noa 表示文件碎片的数量。如果 Decoder 函数执行成功，其将生成还原文件到 decodedFile 指定的位置并返回真；否则返回假。

与 Decoder 类类似，Encoder 类也只包括一个静态函数 encode 用以实现将文件分块的工作，encode 函数接受三个参数，分别是 File 类型的 inputFile 用来表示要被分块的文件，File 类型的 shardsFolder 用来表示碎片产生到的文件夹，int 类型的 fid 用来表示文件的 ID。如果分块成功，encode 函数将返回真并在 shardsFolder 指定的文件夹下保存所有的文件碎片（文件名就是碎片的 ID）；否则 encode 函数将返回假。

目前，分块的大小固定为 512kB、数据分块与冗余分块的比例固定为 1：1，这些内容均在客户端文件分块模块的代码中被确定。

客户端文件夹监控模块

这个模块将根据配置文件的设置监控一系列文件夹并在发现新文件时发起数据连接将其上传到分布式文件系统中配置文件指定的逻辑位置并在本地删除之。为了实现这个目的，该模块包含了 4 个类，分别是 FolderScanner 类（继承了 Thread 类，可以在新线程中执行）用于监控文件夹、FileUploader 类用于发起到服务器的数据链接并上传文件、FileUtil 类用于提供一些文件遍历、文件删除等方法、FileAttr 是定义了文件各属性的数据结构。

FolderScanner 类由启动模块调用执行，在执行的一开始，其将调用 FileUploader 类发送检查并新建文件夹报文确保各被监控文件夹对应的逻辑位置存在。之后，其将每隔 1 分钟确认一次是否有文件夹中被放入了新文件。如有，其首先获取这个文件的所有属性，然后调用 FileUploader 类向服务器注册这个文件。如注册成功，其将调用文件分块模块将文件分块并再次调用 FileUploader 类向服务器上传各个文件碎片。

FileUploader 类用来发起到服务器的数据链接发送上传文件相关的报文。其先建立到服务器的 TCP 链接，然后根据调用函数的不同发送不同的报文并根据服务器的回复返回不同的值表示状态。

FileUtil 类是一个工具方法类，其提供了基于广度优先搜索的文件文件夹遍历方法和基于深度优先搜索的文件夹清空方法。

客户端网络链接模块

客户端网络链接模块中包括 3 个类，分别是数据链接类 FragmentManager、控制连接类 ServerConnecter 与文件传送类 FileTransporter。

ServerConnecter 类继承了 Thread 类，将被启动模块调用并在一个新线程中执行。ServerConnecter 类每隔 5 秒钟向服务器发送一条客户端状态报文并根据服务器的回复判断是否有待处理的请求。如果有，其将再发送处理请求报文并创建一个 FragmentManager 类的实例处理服务器回复的请求。

FragmentManager 类是数据链接类，负责处理数据链接中的客户机碎片上传报文、客户机碎片下载报文与客户机碎片删除报文。就性能而言，其本应继承 Thread 类并在新线程中并发执行，然而目前为了简化系统设计避免同步错误其还不能在新线程中执行。调用 submit 函数时，首先将建立到服务器的数据链接，然后根据 FragmentManager 类的成员变量发送不同的报文并在需要时调用 FileTransporter 类的静态函数进行文件碎片的传输。

FileTransporter 类定义了两个静态函数分别用于文件碎片的发送与接收。其执行原理非常简单，只需要在 socket 与文件分别打开一个输入流和一个输出流并在一侧读出另一侧写入即可。由于文件发送完成后不能立刻断开链接，在发送文件之前需要先发送文件的大小以供接收方判断文件何时结束。

服务器启动模块

服务器启动模块的工作非常简单，其只需要启动控制链接模块与数据链接模块的链接监听线程（即调用两个包中 ServerThread 类的 start 方法）即可。

服务器控制链接模块

服务器控制链接模块包括类 ClientThread 与 ServerThread，它们都使用 java.net.Socket 类实现了网络通信。

ServerThread 的实现是平凡的，其只需要监听控制链接端口并在有新连接时创建一个 ClientThread 类的实例并执行 start 函数在一个新线程中处理之即可。

每个 ClientThread 类的实例负责与一台客户机的控制报文通信，其不断从 socket 的输出流读取客户端发来的报文并判断报文的种类（控制链接报文的具体种类与格式见本章客户端-服务器间的网络通信一节）：
（一）若是客户端状态报文，其根据报文内容在数据库中更新状态，然后查询等待这个客户端处理的请求数量并装在回复报文中发送给客户端；
（二）若是处理请求报文，其在数据库中查找等待该客户端处理的第一条请求并返回给客户端；
（三）若是中断连接报文，其立即中断链接并在数据库中记录该客户端已经下线；

此外，在任何时刻，如果服务器无法理解报文的内容或链接发生了超时，服务器将主动中断与客户端链接并在数据库中记录该客户端已经下线；如果客户端的报文内容与系统的状态矛盾（如收到了处理请求报文但并没有等待该客户端处理的请求），服务器将返回 Error!。

服务器数据链接模块

服务器数据链接模块包括类 ClientThread、ServerThread、FileTransporter，其中 FileTransporter 类与客户端网络链接模块中的 FileTransporter 类是相同的。

与之前一样，本模块使用 java.net.Socket 类实现了网络通信。

ServerThread 的实现是平凡的，其只需要监听数据链接端口并在有新连接时创建一个 ClientThread 类的实例并执行 start 函数在一个新线程中处理之即可。

ClientThread 类在开始执行后将会先读取客户端传送的第一行内容，之后根据报文的类型（数据链接报文的具体种类与格式见本章客户端-服务器间的网络通信一节）调用不同的函数来处理。具体的处理过程是琐碎的，大体上都是根据报文内容查询（修改）数据库、根据系统的状态回复客户端报文、在需要时调用 FileTransporter 类实现文件碎片的传输。以下结合使用场景例举几个报文的处理过程：

（1）服务器请求文件碎片：这个工作将包括以下 4 步，其中第 3、4 步为客户机碎片上传报文的处理过程。
1.在数据库的 REQUEST 表中写入请求
2.客户端在下次心跳连接时将收到请求
3.客户端将新建一条与服务器的数据连接并将碎片发往服务器,发送后客户端上的碎片将不会被删除.
4.服务器收到碎片后将会把碎片保存到下载文件夹并删除数据库中的请求

（2）文件上传：这个工作将包括以下 4 步，其中第 2、3 步为文件上传报文的处理过程；第 4、5 步为文件碎片上传报文的处理过程；第 7、8 步为客户机碎片下载报文的处理过程。
1.客户端需要向服务器发送上传文件报文
2.服务器收到报文后将会在数据库 FILE 中插入文件项并返回文件的 ID.为了表示文件仍不可用,文件项的 NOA 字段将被设置为实际 NOA 的相反数
3.客户端接下来向服务器发送文件的各个碎片,其中最后一个碎片一定要最后一个发送,之前的碎片可以乱序发送.
4.服务器收到碎片后将会把碎片保存在本地临时文件夹并数据库 FRAGMENT 中插入碎片项.由于碎片未被分配到文件夹,此时 PATH 均为'-1'
5.服务器收到最后一个碎片后检测是否碎片已被全部上传,如是,将碎片分配到各个客户端并将数据库中文件项的 NOA 字段改为实际的碎片数量.分配的方法是在数据库 REQUEST 中写入请求等待客户端取走碎片
6.被分配到的客户端在下次心跳连接时将收到请求
7.客户端将新建一条与服务器的数据连接并接收碎片,发送完碎片后服务器将暂存的碎片删除,同时将数据库中碎片项的 PATH 改为设备的 ID
8.客户端收到碎片后将会把碎片保存到碎片文件夹,服务器完成传输后删除数据库中的请求

（3）碎片删除：这个工作将包括以下 4 步，其中第 3、4 步为客户机碎片删除报文的处理过程。
1.在数据库 REQUEST 中写入请求
2.客户端在下次心跳连接时将收到请求
3.客户端将删除这一碎片并新建一条与服务器的数据连接以通知服务器请求完成.
4.服务器通知后删除数据库中的请求

服务器数据库模块

服务器数据库模块负责分布式文件系统的数据库访问，包括封装了数据库访问方法的Query类与用于定义数据结构的FileItem、DeviceItem、RequestItem类。

本分布式文件系统使用数据库维护所有的元数据，数据库中具体包括表FILE用于存储文件的逻辑位置与属性、表FRAGMENT用于存储碎片的物理位置、表REQUEST用于存储服务器对客户端的碎片请求、表DEVICE用于存储系统中客户端的信息、表USER用于存储网页的注册用户。各个表的定义如下：

```
CREATE TABLE `DEVICE`(
   `ID` int NOT NULL AUTO_INCREMENT,
   `IP` char(20) NOT NULL DEFAULT '',
   `PORT` int NOT NULL DEFAULT 0,
   `ISONLINE` boolean NOT NULL,
   `RS` int NULL DEFAULT 0 ,
   PRIMARY KEY (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `FRAGMENT`(
   `ID` int NOT NULL,
   `PATH` char(20) NOT NULL DEFAULT '',
   PRIMARY KEY (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `FILE`(
   `ID` int NOT NULL AUTO_INCREMENT,
   `NAME` char(20) NOT NULL DEFAULT '',
   `PATH` char(60) NOT NULL DEFAULT '',
   `ATTRIBUTE` char(10) NOT NULL DEFAULT '',
   `TIME` char(10) NOT NULL DEFAULT '',
   `NOA` int NOT NULL DEFAULT 1,
   `ISFOLDER` boolean NOT NULL DEFAULT false,
   PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `REQUEST`(
   `ID` int NOT NULL AUTO_INCREMENT,
   `TYPE` int NOT NULL DEFAULT 0,
   `FRAGMENTID` int NOT NULL DEFAULT 0,
   `DEVICEID` int NOT NULL DEFAULT 0,
   PRIMARY KEY (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `USER`(
   `ID` int NOT NULL AUTO_INCREMENT,
   `NAME` char(20) NOT NULL UNIQUE DEFAULT '',
```

```
  `PASSWD` char(20) NOT NULL DEFAULT '',
  PRIMARY KEY (`ID`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

对应于表 FILE、DEVICE、REQUEST，数据库模块中定义了 FileItem、DeviceItem、RequestItem 类来表示相应的数据结构。FileItem 类包括成员变量

```
private int id;
private String name;
private String path;
private String attribute;
private String time;
private int noa;
private boolean isFolder;
```

分别用于表示文件 ID、文件名、文件逻辑路径、文件属性、修改时间、碎片数量、文件 or 文件夹。

DeviceItem 类包括成员变量

```
private int id;
private String ip;
private int port;
private boolean isOnline;
private int rs;
```

分别用于表示设备 ID、IP 地址、端口号、在线情况、剩余空间。

RequestItem 类包括成员变量

```
private int id;
private int type;
private int fragmentId;
private int deviceId;
```

分别用于表示请求 ID、请求类型（1 服务器取分块；2 服务器将分块发送给客户端；3 删除客户端上的分块）、被请求的分块的 id、被请求的分块所在的设备的 id。

Query 类定义了对上述五个表查询、修改、删除、新增条目的函数，其通过 JDBC 接口实现了对数据的访问，访问的流程为：

（一）在构造函数中使用 DriverManager.getConnection 函数创建到数据库的连接（一个 Connection 类实例）；

（二）通过 Connection 类实例的 createStatement 函数创建一个 Statement 类实例；

（三）通过 Statement 类实例的 executeQuery 函数执行 SQL，SQL 的内容可以使用格式化字符串根据函数的参数填入不同的内容，该函数将返回一个 ResultSet 类实例；

（四）对 ResultSet 类实例，使用 next 函数与 getInt、getBoolean、getString 等函数遍历查询的每个结果；

（五）对 ResultSet 类实例与 Statement 类实例，执行 close 函数关闭连接；

（六）在 closeConnection 函数中，调用 Connection 类实例 close 函数关闭连接。

客户端-服务器间的网络通信

这一部分将单独介绍客户端与服务器间的通信方式。

考虑到目前国内网络的现状，在网络通信中，我们假设在我们的系统中客户端都隐藏在 NAT 之后。NAT 全称网络地址转换，是一种在 IP 封包通过路由器或防火墙时根据 NAT 表重写來源 IP 地址及端口或目的 IP 地址及端口的技术。这种技术被普遍使用在有多台主机但只通过一个公有 IP 地址访问因特网的私有网络中，在一定程度上解决了 IPv4 地址不足的问题。

然而，由于在一个具有 NAT 功能的路由器下的主机并没有建立真正的 IP 位址，并且不能参与一些因特网协议，一些需要初始化从外部网络建立的 TCP 连接是无法实现的，这意味着当服务器需要客户端响应请求时，服务器可能无法主动建立与客户端的 TCP 连接。

为了解决这个问题，我们采用了分离的数据连接与控制连接：服务器与客户端的控制连接从客户端被执行开始变被建立并保持长期连接，这样服务器就可以从客户端的心跳连接中检测客户端的在线状态并在需要客户端响应请求时通过回复心跳连接通知客户端。

当客户端发现服务器上有等待其处理的请求时，客户端将主动建立一条与服务器的数据连接并处理这个请求。

在我们的控制连接中目前定义了三种报文，分别为：

（一）客户端状态报文
此报文为客户端的心跳报文，每隔 5 秒发送一次，格式为：
客户端发送：1 {设备 ID} {客户端剩余空间}
服务器回复:received with {等待客户端处理的请求数量} unread request!

（二）处理请求报文：
此报文当客户端发现自己有未处理的请求时发送，服务器将回复等待其处理的请求的具体内容。其格式为：
客户端发送：3 {设备 ID}
服务器回复:{请求 ID} {碎片 ID} {请求类型}

（三）中断连接报文
此报文用来在发生异常时立即中断连接，格式为
客户端或服务器发送：exit
----链接中断----

在我们的数据连接中目前定义了六种报文，分别为：

（一）客户机碎片上传报文
客户端发送:1 {请求 ID} {碎片 ID}
服务器回复:received!

客户端回复：{碎片内容}
服务器回复：received!
----链接中断----

（二）客户机碎片下载报文
此报文用来从服务器下载其指定的碎片，格式为
客户端发送：2 {请求 ID} {碎片 ID}
服务器回复：{碎片内容}
客户端回复：received!
----链接中断----

（三）客户机碎片删除报文
此报文用来通知服务器客户端已经删除了其指定的碎片，格式为
客户端发送：3 {请求 ID} {碎片 ID}
服务器回复：received!
----链接中断----

（四）上传文件报文
此报文用来申请向分布式文件系统中上传文件，格式为
客户端发送：4 {设备 ID} {文件名} {路径} {属性} {碎片数量} {是不是文件夹}
服务器回复：FileId: {文件 ID}
----链接中断----

（五）文件碎片上传报文
在客户机发送了上传文件报文并收到了服务器回复的文件 ID 后，需要使用此报文上传该文件的全部碎片。此报文的格式为

(最后一个碎片一定要最后上传,这时服务器有可能回复 UPLOADFAIL 表示整个文件上传失败)

客户端发送:5 {文件 ID} {碎片序号} {碎片总数}
服务器回复：received!
客户端回复：{碎片内容}
服务器回复：received!
----链接中断----

（六）检查并新建文件夹报文
当客户端启动时，其需要使用此报文保证被监控的文件夹对应的逻辑位置在服务器上有记录。当服务器收到此报文后，如发现该逻辑位置在服务器上没有记录，先新建一条对应的记录。

客户端发送：6 {设备 ID} {文件夹数量}
{文件夹路径} {文件夹名称} （每行一个文件夹信息）
服务器回复：received!（如已有/能建立这些文件夹）或 ERROR!（如某个文件夹与一个文

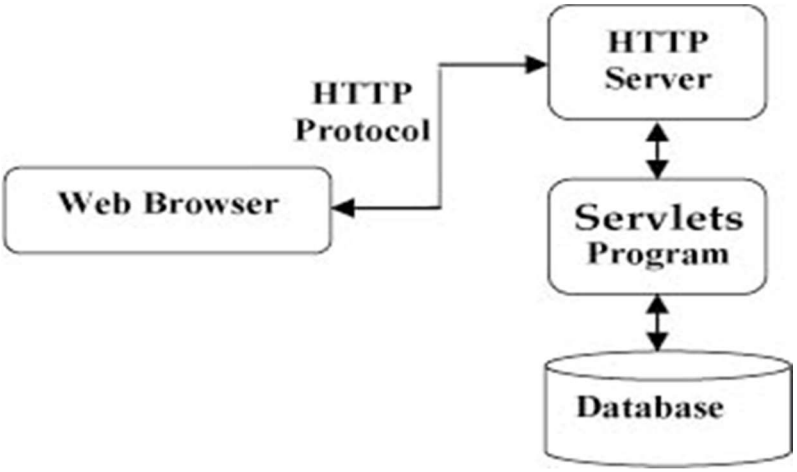
件重名）
----链接中断----

Tomcat 服务程序

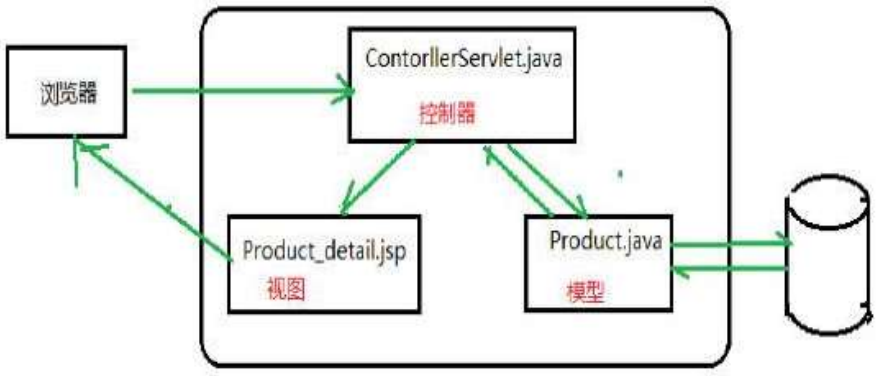
Tomcat 是 Apache 公司开发的 java 应用服务器，提供了 web 请求调用服务器端 java 应用程序的服务。
因此，这是我们 web 服务这一块最重要的一个模块。

我们是直接使用的 tomcat 安装文件，按照官网的指示进行了初期配置，再结合自身需求进行了其他的定制化配置。

其配置我将在服务器部署文件中详细介绍。



在其基础上我们使用了 struts2 框架------当前 jee 主流 SSH 框架的一部分。



虽然也是现成的框架，但是也有一些定制化的配置。比如：
ｓｔｒｕｃｔ动态方法调用默认关闭　需要
<constant name="struts.enable.DynamicMethodInvocation" value="true"></constant>开启

有了 struts，我们可以把所有的 http 对于 action 的请求都转发给 struts 的主控，由 struts 负责应用程度的调用，同时 struts 也提供了很友好的网页到 javabean 的传参数和 javabean 的返回值（很友好的很好的支持返回 jason 格式；同时也便捷的提供了网页的跳转关系的实现）。

登录注册模块

我们的系统允许用户用浏览器通过互联网访问文件系统，因此浏览器端的注册和登录功能室必须的。

登录注册模块主要是包括了：

index.html------提供了一个友好的网页让网页端用户进行注册和登录。

主要采用了 bootstrap 主题，采用了其提供的垂直表单元素；
同时注册和登录表单是可以单击进行切换的，使用了 bootstrap 切换卡元素。

index_ajax.js------与之配套的，让网页具有和用户和服务器交互的动态能力。

主要是为两个 button（注册和登录）提供了对应的 js 动作。
调用异步的 ajax 将表单信息提交给服务器并调用下面两个 java 程序之一进行服务。
（很重要的一点是我并没有用 submit 提交表单而是 ajax，否则网页会进行刷新，这不符合我的设计预期，我希望反馈信息打印在网页控制台部分，而不是刷新网页）

下面是 ajax 的举例：采用了 form 格式的 var 来进行 data 的传送，采用 post 方法，在回调函数中更改网页 html，输出服务器反馈信息

```
$("#regSubmitButton").click(function(){
    var    userName=$("#inputUsername_reg").val();
    var    userPasswd=$("#inputPassword_reg").val();
    var    form=new FormData();
    form.append("userName",userName);
    form.append("userPasswd",userPasswd);
    $.ajax({
  url:"UserReg.action",
  type:"POST",
  data:form,
  dataType:"text",
  processData:false,
      contentType:false,
  success:function(databack){
      var obj = $.parseJSON(databack);
      var feedback = obj.result;
```

```
            //alert(userName);
            $("#statusFeedback").text(feedback);
        }
       });
    });
```

UserLogin.java------接受来自网页的登录请求，查询数据库进行身份核实和反馈核实结果。

通过继承 extends ActionSupport，使得该 java 成为 tomcat 的可调用程序。
只要帮 class 内部的变量提供了设置（set）和返回（get）函数，该变量就会自动被返回到 js，通过 obj.X 的格式访问该变量，X 为其在类中的名字。
如下所示：

```
private  String  result;

public   void setResult(String result)
{
 this.result = result;
}

public   String  getResult()
{
 return this.result;
}
```

登录 java 程序需要 import database.*;
这是我们实现的对于服务器端 database 访问的封装好的接口。
我们从 js 接收用户名，根据用户名查询服务器客户端，获得正确密码，再和用户提交的密码对比，一致则登陆成功，跳转到主界面，否则登陆失败，返回失败信息到网页。

对于 java 程序返回值的格式，需要在 struts 中配置：

```
<action name ="UserLogin" class = "userManagement.UserLogin">
            <result type="json" name="success"></result>
</action >
```

这样返回值就是 json 对象，而不是跳转的网页 html。

UserReg.java------接收来自网页的用户注册请求，将请求插入数据库等待管理员审核。

基本和 UserLogin 类似。

网页主界面

网页主界面是用户登录后的主界面，在这个界面用户可以看到我们项目的介绍图片，标题，介绍，最重要的是可以访问系统的文件目录，可以进入某个文件夹或者返回上层目录，还可以进行文件的下载上传重命名等访问和管理。

网页主界面模块主要是包括了；

majorPage.jsp------包含了界面的主要的 html 代码。之所以还采用了 jsp 在服务器端动态生成 html 代码，是因为第一次打开该网页就会展示文件系统根目录文件夹信息，这些信息是动态的。因此我动态的查询数据库并返回 html 代码。
以下方式可以在 jsp 中插入 java 代码并动态执行：

```
<%
        int i;
        Query query = new Query();
        FileItem[] files = query.queryFile("/");
        query.closeConnection();

        if(files==null)
            return;
        else
        {
            for(i=0;i<files.length;i++)
            {
                out.println("<tr class='file_list_go'>");
                out.println("<td></td>");
```

```
                    if(files[i].isFolder()==false)
                        out.println("<td> <label><input
type=\"checkbox\"></label>     <span class=\"glyphicon glyphicon-
file\"></span>  " + files[i].getName()+"</td>");
                    else
                        out.println("<td> <label><input
type=\"checkbox\"></label>     <span class=\"glyphicon glyphicon-folder-
open\"></span>  " + files[i].getName()+"</td>");

    out.println("<td>"+files[i].getAttribute()+"</td>");
                    out.println("<td>"+files[i].getTime()+"</td>");
                    out.println("</tr>");
                }
            }
        %>
```

该主界面有明显的几个组成部分：（他们之间的布局采用了 bootstrap 的网格布局系统）

1. 网页大标题和副标题

采用 bootstrap 标题元素。

2. 网页宣传动态图片

采用了 bootstrap 的幻灯片元素，自动换图片展示我们的网站。

3. 当前访问位置 面包屑式导航栏

采用了 bootstrap 的面包屑导航元素，可以让用户清晰的直到当前访问的文件系统的位置（全路径）。

4. 文件目录列表

用列表的形式展示当前访问路径下的全部文件和文件夹。
列表包含勾选框，文件夹或者文件图标，文件或者文件夹名称，读写权限和最后修改时间。

5. 文件操作 button 组

四个 button，分别提供下载上传重命名和删除操作。

6. 任务进度条

用来反馈用户正在下载的文件的文件名和该文件所需要的碎片在服务器端的收集进度。
当收集进度到达 100%后，该进度条可点击，点击可下载该文件，

7. 控制台

使用了 bootstrap 的提示栏元素。
用来反馈给用户操作的状态信息。

8. 网站声明

用来声明网站的所有人和网站负责人的联系方式（邮箱链接可直接点击，自动调用打开用户端 email 程序直接填写我的邮件地址）

majorPage_ajax.js------包含了主界面全部的用户交互的代码.

文件目录展示交互模块

该模块属于主界面，不过有其独立性和重要性。该模块让用户可以像使用通用操作系统的文件浏览器一样的访问我们的文件系统。
用户可以单击进入子目录或者返回上层目录，同时当前访问路径导航栏随之刷新。

文件目录展示交互模块主要包括了：

GetFileList.java------根据输入：查询的全路径；输出该路径下的全部列表项的 html 代码。
该类使用了数据库访问包：

```
import database.*;
```

majorPage_ajax.js------包含该模块主要代码

全局变量

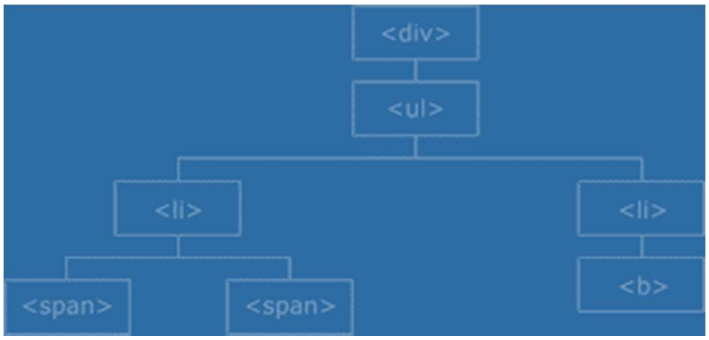
```
var curr_path_array = new Array();
curr_path_array[0] = "/";
```

存储了当前访问路径，数组 index=0 处永远是‘/’，也就是根目录。

导航栏的信息根据操作和该全局变量内容很容易实现。

复杂的是文件列表，其实现都是许多复杂的小细节，不多赘述网页相关细节，只着重介绍两大挑战：

1. 列表是动态生成的，而我们访问网页的元素的常用方法是通过 id 或者 name 或者 class 等等属性进行选择，如果还沿用这种方法，我们需要一套复杂的机制为列表项生成一个 id，而且还不能重复，这基本是不可实现的。
   偶然的机会我突然发现我之前觉得用不到的网页的 dom，对象访问方法恰好可用。



通过 jQuery 遍历，您能够从被选（当前的）元素开始，轻松地在家族树中向上移动（祖先），向下移动（子孙），水平移动（同胞）。这种移动被称为对 DOM 进行遍历。

这样的话我通过访问列表的 children，就相当于访问了所有生成的列表项。

2. $("#button_download").click()这种格式是一种动作函数的静态绑定，所以新加载的列表项如何获得动作能力一开始让我绞尽脑汁。
   后来多方调研知道：on（）函数是一种动态绑定的方法，可以将动作委托给列表，也就是列表项的父亲，这样的话由该父节点将绑定效果传递下去。

点击列表项之后，首先判断是点击的文件夹------从列表项获取对应的文件夹名称；
将该文件夹加入全路径；
刷新导航栏；
通过 ajax 向服务器查询新路径下文件列表。

还是文件------控制台显示，点击的是文件，不可进入；不采取其他措施

还是返回上一层------ 将全路径最里层元素去掉；
刷新导航栏；
通过 ajax 向服务器查询新路径下文件列表。

查询通过 ajax 实现，调用 GetFileList.action

```
                $.ajax({
                        url:"GetFileList.action",
                        type:"POST",
                        data:form,
                        dataType:"text",
                        processData:false,
                        contentType:false,
                        success:function(databack){
                            var obj = $.parseJSON(databack);
                            var new_file_list = obj.html;
                            //alert(new_file_list);
                            $("#file_list_body").html(new_file_list);
                        }
                });
                $("#statusFeedback").text("成功进入该目录！");
            }
    );
```

文件下载模块

该模块提供了用户选中单个文件并进行下载的全套服务。
用户选中单个文件，点击下载，服务器开始收集碎片，实时反馈进度，网页进度条实时更新，进度 100%后可单击进度条下载该文件。

FileDownloader.java------包含了三个功能函数

1. downloadRegister()

将一条下载请求插入数据库，这样的话服务器将知道要从各个客户端收集该指定的碎片。
该函数调用了数据库访问函数包：import database.*

2. progressCheck()

服务器查询特定本地临时碎片数目，计算出碎片收集进度并且返回给网页。
该函数调用了本地文件访问接口。

3. decodeFile()

服务器调用 erasurecode 开源解码程序，将特定文件复原，等待用户通过 http 请求下载。

majorPage_ajax.js------包含该模块的主要代码

当用户点击下载后，遍历列表，对于每一个勾选项进行下载操作。
下载操作就是：
获取要下载文件全路径和名称；
利用 ajax 调用动态方法 FileDownloader!downloadRegister，请求服务器收集碎片；
问该文件任务添加进度条；
定时通过 ajax 调用动态方法 FileDownloader!progressCheck 检测收集进度，并刷新进度条，如果进度到达 100%，利用 ajax 调用动态方法 FileDownloader!decodeFile 进行碎片远程拼接，为进度条添加下载属性，链接到生成的要下载的文件。

注意此时，我们需要 ajax 返回的结果反馈到控制台，我在此处遇到了一个问题，很重要。

我之前采用异步 ajax，导致返回值经常是 undefined，调查了一会明白了，此处应当用同步 ajax，否则 ajax 未完成，控制台已经开始读取返回值了。同步通讯格式如下：

```
$.ajax({
        url:"FileDownloader!decodeFile.action",
        type:"POST",
        data:form,
        dataType:"text",
        processData:false,
        contentType:false,
        async: false,                                    //此处采用同步查询进度
        success:function(databack){
            var obj = $.parseJSON(databack);
            var result = obj.result;
            if(result == "Error")
                $("#statusFeedback").text("解码拼接出错！ ");
            else
```

```
                                $("#statusFeedback").text("解码拼接文件成功！");

                        }
                });
```

定时查询通过

```
//设置进度刷新间隔
    window.setInterval(function(){refresh_progress();},3000);
```

实现

七、系统性能分析

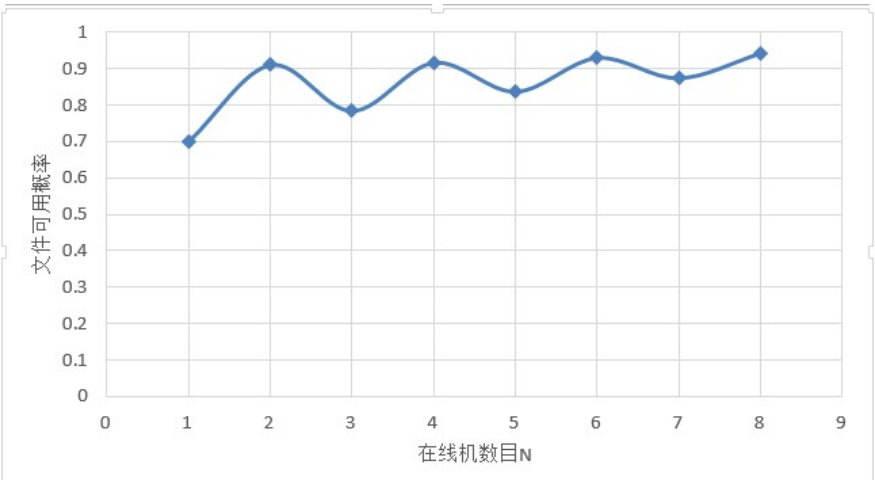
文件可用概率分析

我们的碎片分配方法是将碎片平均分给所有的在线客户机。对于大小为X kb的文件，分块的数量将为2*(X//500+1)。获得所有碎片的一半即可下载文件。

对于有m个客户机的系统，可简单假定机器在线概率为p，不同客户机是否在线相互独立，即客户机在线数量服从参数为m，p的二项分布。当一半储存碎片的客户机在线时，文件可用。

如果上传文件时在线客户机数目为n，文件可用的概率即n个客户机中至少n/2个客户机在线的概率，为$\sum_{i=n/2}^{n} C_n^i p^i (1-p)^{n-i}$。

下为p=70%时文件可用概率与n的关系图：



明显可以看出随着在线机数目增多，文件可用概率呈明显的上升趋势。因此我们的系统呈现出用户越多可用性越高的特点。

当p<50%时，文件可用概率显然会随客户机增加而降低。但是考虑到实际工作中大家使用文件的时间几乎与工作时间重叠，于是在使用文件时客户机应有很大概率上线，p<50%的情况不容易出现。

而即使面对p比较小的情况，我们也可以很容易的进行优化来提高文件可用率，比如我们可以设置少量长期在线客户端，或者增加碎片数量。

上传速度分析

上传速度取决于在线客户端的数量和客户端与服务器数据传输速度。当多个客户机在线时，客户机可以同时与服务器进行传输，从而摆脱用户网络上传速度的限制。

由于测试环境问题，目前只测了一个客户端开启，并且用校园网连接时的数据上传速度。这种环境下，平均上传速度为 2.74M/s。

下载速度分析

下载速度取决于所用网络与服务器之间连接速度。并且由于通过网络下载时需要先收集碎片，碎片上传到服务器的速度也会影响到实际下载速度。

在单客户端，并且校园网连接的条件下，测得下载速度（预下载过程不计）大于 10M/s。