# 介绍

Google文件系统（GFS）是一种分布式文件系统，以能够大规模处理大型文件存储而著称，同时具有容错能力，并在一定程度上保持一致性以支持典型用法。

* **为什么要做这个项目？**

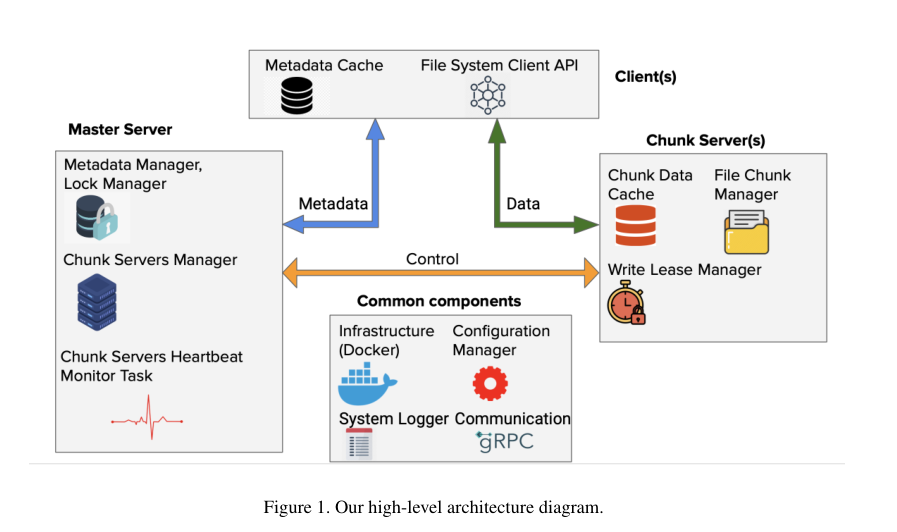
1. GFS是由Google Inc.开发的，并且是最新的专有技术，因此缺乏反映其原始设计并具有足够文档的开源实现，很难从学术角度评估GFS，和测试。
2. 详细描述GFS技术细节的文献较少，例如GFS的高并发性、crash-resistance（容忍碰撞）
3. 随着数据量、网络容量、硬件功能的改善，一个设想是：GFS的多master模式是有利的，因为它提供了更好的容错能力、可用性和吞吐量。（由于时间原因未实现这个版本，但是方便扩展）
4. 使用docker实现

* **实现概要**

1. 该项目实现了一个功能齐全的GFS（具有可配置数量的块服务器的单个主服务器）。
2. 简化了一些功能（例如不支持并发附加和快照）来反映其大部分原始设计。
3. 我们应用了Singleton（单例）和Composite软件设计模式逐步构建了系统的核心组件。
4. 此外，此项目已经进行了严格的测试，代码库的很大一部分（将近30％）是由测试组成的，包括单元测试，模拟测试，集成测试和端到端测试。

# 架构

GFS由三个部分组成：Master server, Chunk servers ， Client library（客户端库）



## Master server

主服务器是文件系统的核心部分。它处理文件元数据，块服务器控制文件系统中的操作。实际数据不是存储在主服务器上，而是存储在块服务器上。它知道文件系统中的所有文件以及文件的所有块，以及每个文件块的存储位置。

主服务器由四个主要组件组成：元数据服务，Lock管理器，块服务器管理器和块服务器心跳监视任务。

### Metadata Service

Metadata Manager是**单例对象**，用于管理元数据上的并发CRUD（创建，读取，更新和删除）操作，包括从文件名到一系列块句柄的映射（由块索引索引），以及从块句柄到块版本，服务器位置和 primary lease holder（主要租赁持有人）。

并发操作是通过**并行哈希图**实现的，并行哈希图由下面的多个子图（默认为16个）组成，因此可以安全地并行化对不同子图的读取和写入访问。 Metadata Manager的一项基本职责是命名空间的管理，即使文件共享同一父目录，也允许并发创建文件。锁管理器可简化此操作，该管理器可管理文件锁的创建和获取。

元数据服务处理客户端的请求，并通过元数据管理器处理元数据。它为每个客户端的请求提供完整的CRUD支持，并根据协议安排一系列操作。例如，打开一个未创建的文件请求首先会调用Metadata Manager来创建文件元数据，并调用块服务器管理器来分配第一个块。写请求涉及更多的交互，因为元数据服务需要将请求发送到块服务器以**授予租约并推进块版本**。元数据服务以及其他服务通过gRPC服务调用实现。

### Lock Manager

Lock manager负责管理需要上锁的资源。**该单例对象是必需的，因为服务器在创建文件之前不会创建锁。**它通过并行哈希图提供并发创建和获取服务。此外，它采用了Abseil库提供的absl :: Mutex对象，并支持读取器和写入器锁定模式。**并行哈希图，读取器/写入器锁，锁管理器和元数据管理器的使用为主服务器提供了提供并发文件CRUD操作的完整功能。**

### Chunk Server Manager

块服务器manager管理集群中的所有块服务器。它知道集群中所有可用的块服务器。它负责分配用于存储块的块服务器，因此它知道每个块的存储位置。将新的块服务器添加到集群后，在启动过程中，它将向运行在主服务器上的块服务器管理器报告自身，并且块服务器管理器对其进行注册，随后可以选择该块服务器进行块分配。随后，块服务器定期向管理器发送相关信息。发送的信息包括可用磁盘空间，存储的块句柄列表，管理器将回复现在已经过期的可以被删除的块句柄列表。

#### 存储文件

当创建新的块时，元数据管理器要求块服务器管理器分配一定数量的块服务器以存储该块。块服务器管理器按其可用磁盘大小的顺序维护块服务器的排序列表，列表中具有最大可用磁盘的块服务器位于列表顶部，因此它选择具有最大可用磁盘的N个块服务器。这有助于在块服务器之间实现负载平衡和更好的磁盘使用。它跟踪为该块选择的N个块服务器，并且元数据管理器在读/写请求期间将询问块的位置

#### 注销块服务器

当块服务器不可用/无响应时可以将其从管理器中注销。然后管理器将不会再知道块服务器的存在，也不会使用块服务器。

#### 数据丢失

由于块服务器定期向管理器报告自身，并且它们将存储的块句柄报告为管理器的一部分，因此管理器能够更新有关此块服务器的信息，例如可用磁盘和存储的块。块服务器可能先前被分配用于存储某个块，但是由于写操作期间崩溃或者数据损坏，该块服务器可能不再具有该块。通过报告信息，管理器可以知道该块不在该块服务器中，并停止包括此块服务器作为该块存储位置的一部分。

#### 块服务器心跳监控任务

这是一个后台线程，用于监视块服务器管理器中所有已注册块服务器的心跳。它会定期将心跳消息发送到块服务器，并在3次（可配置）尝试中获得响应时，将服务器声明为可用。如果尝试后仍未收到响应，则会将块服务器声明为不可用，并请求块服务器管理器注销该块服务器，以便该管理器停止为该块服务器分配块。如果块服务器恢复可用，因为块服务器在启动时会定期向管理器报告自身，所以块服务器管理器将知道此服务器重新开始服务并重新注册。

## Chunk Servers

块服务器处理数据操作。将文件划分为可配置的固定大小的文件块（默认为64MB），并且根据客户端的复制需求，每个文件块都存储在块服务器的子集中。默认情况下，将在3个块服务器之间复制块。如上所述，要存储在块服务器上的块由主块服务器管理器确定。块服务器处理将这些块数据持久化到磁盘。客户端仅与块服务器交互以读取或写入实际数据。

块服务器由5个主要组件组成：文件块管理器，块服务器文件服务，块数据缓存，块服务器租赁服务和块服务器报告任务。

### 文件块管理器

**文件块管理器负责将块持久化到磁盘。**使用LevelDB 实现了此功能，这是Google构建的快速持久键值存储。键和值被视为存储的任意字节。文件块管理器使用levelDB存储块，**将块句柄用作键，并将块数据用作值**。这使我们能够在系统中享受这种高性能存储引擎的好处。在数据读写期间，我们无需将并发安全性转移到LevelDB，此时无需在文件块管理器内进行显式同步。

我们还利用了LevelDB的压缩功能，该功能有助于压缩块数据并减少磁盘消耗，并使我们的IO更快，从而降低了CPU成本。 LevelDB压缩非常快，并且对于不可压缩的数据会自动禁用。作为存储在磁盘上的数据的一部分，我们还存储了块版本，该块版本在读/写期间使用。我们不需要缓存块数据，因为LevelDB将数据作为文件写入文件系统中，因此我们可以利用缓冲区缓存。

文件块管理器为块提供Create，Read，Write，Delete，GetVersion，UpdateVersion操作。读取和写入操作要求指定版本，如果请求的版本与块的存储版本不匹配，则返回错误代码。对于create，我们仅将块句柄存储为键，将create version（1）存储在具有空数据的值中。

### 块服务器文件服务

这是在客户端向其发送块数据请求的块服务器上运行的gRPC服务。这将处理来自客户端的读取数据，发送数据，写入请求，在块创建期间来自主服务器的创建请求以及在写入过程中来自主块服务器的数据变更请求。**该服务与文件块管理器交互以处理客户端请求。**

* **创建请求**

对于块创建请求，它要求文件块管理器创建一个版本等于1的新块。如果该文件块已经存在，则文件块管理器将返回错误代码，否则，它将成功，并且服务将成功响应客户端。

* **读取请求**

它要求文件块管理器指定（偏移量、版本、指定长度）。如果文件块不存在或指定的版本与存储的版本不匹配，或者偏移量大于文件块数据长度，则文件块管理器将返回错误代码。否则，服务会将读取的数据返回给客户端。

* **写请求**

1. 客户端首先将数据发送到块服务器文件服务，然后该服务计算已发送数据的校验和以确保它与已发送校验和匹配，然后使用数据校验和作为key将数据存储在块数据高速缓存中。
2. 客户端随后将写入请求发送到Primary server，该服务器首先检查其在块上是否具有有效的租约。如果有，则要求文件块管理器从指定的偏移量开始将指定长度的数据写入指定版本的块。
3. 如果文件块不存在或指定的版本与存储的版本不匹配，或者偏移量大于文件块数据长度，则文件文件块管理器将返回错误代码。否则，文件块管理器将返回写入的字节数。
4. Primary server将此写请求并行发送到所有其他副本，以应用于其它块副本。副本检查写请求的校验和，并从块数据缓存中读取该校验和的数据，并使用文件块管理器执行相同的写操作。并将结果返回到主块服务器。主数据库将其写入结果以及其他副本的结果发送给客户端。

* **删除操作**

当块服务器向主服务器执行其定期报告时，主服务器返回要删除的块列表，并要求文件块管理器删除它们。

### 块数据缓存

这是块服务器上的缓存，用于临时存储客户端发送的文件块数据。作为写工作流程的一部分，客户端首先将数据推送到块服务器，然后再向主块服务器发出写请求，然后由主服务器指示其他副本服务器使用该数据应用变更。对于我们的用例，缓存将数据校验和映射到数据。通过提供数据的校验和作为键从缓存中获取数据，如果存在，则将其返回。尽管我们将缓存构建为键值结构，但可以用来存储任何内容。为了避免拥有非常大的缓存，我们通过在不再需要数据时删除数据来简化缓存的实现。这通常是在客户端写请求成功完成之后。

### 块服务器租赁服务

这是在块服务器上运行的gRPC服务，主服务器用于将租约授予块的主块服务器。主服务器将租约信息发送到块服务器，块服务器跟踪已被授予的活动，未到期的租约。这在写入过程中使用，在该过程中，主块服务器在继续之前检查其是否具有有效的租用租约。

**块服务器报告任务**

这是在块服务器上运行的后台线程，该线程定期将块服务器信息报告给主服务器。块服务器最初在启动时向主服务器报告其自身，然后继续进行定期报告。这样可以使主服务器保持有关该块服务器的最新信息。诸如可用磁盘空间和存储的块之类的信息会报告给主服务器，而主服务器会用块服务器可以删除的陈旧块列表进行答复。

这在主机和块服务器之间存在临时网络分区的情况下也很有用，这样主机就无法到达块服务器进行心跳检测。在这种情况下，在经过主服务器几次尝试之后，它会假定块服务器已关闭并取消注册块服务器，这会阻止客户端和主服务器与块服务器进行通信。由于块服务器会定期报告自身，即使在网络分区期间未成功发送报告，在修复了网络分区之后，该报告也将成功提交给主服务器，然后由主服务器重新发现该块服务器并重新注册。现在，主服务器再次开始使用块服务器进行块分配。

## Client Library

客户端库提供线程安全的调用：打开，读取，写入和删除。每个客户端调用都通过线程本地ClientImpl对象进行转换，该对象管理相关资源，例如缓存的文件块元数据，配置管理器以及客户端用来与服务器联系的gRPC端点。

缓存的元数据存储在名为gfs :: client :: CacheManager的per-thread数据结构中，该结构提供从（文件名，块索引）到块句柄以及从块句柄到块元数据（版本，主位置）的映射作为块服务器位置。 CacheManager在创建块元数据条目时存储时间戳，如果过期则在访问请求时将返回错误。

* 读取操作

1. 客户端遍历所有块，并向块服务器发出请求以获取内容（如果发现主机确实有一个或已过期，它将与主服务器联系以获取元数据）。
2. 它连接返回的块字节并将结果返回给用户。由于块是复制的，因此只要读取请求之一成功，客户端调用就会成功返回，也就是说，客户端的读取可以容忍块服务器停机。
3. GFS2.0支持处理EOF上的读取，即，如果请求的读取超出了当前EOF，则我们将其视为有效情况，并返回EOF上读取的字节。

* 写操作

1. 写入实现与GFS论文略有不同，因为块数据直接从客户端并行推送到所有块服务器。由于客户端仅需要处理自身与块服务器之间的错误，因此这大大简化了工程工作。
2. 此外，并行推送数据产生的延迟可能比GFS论文中描述的另一种方式要少，尽管后者可能会更好地利用每台机器的带宽。我们在写库调用中实现了**重试策略**，该策略对用户是透明的，以处理任何暂时性故障，例如租约到期和gRPC调用超时。这样可以提高此函数调用的稳定性。

## System Common Components

我们使用Docker以容器化的方式部署文件系统服务器。每个组件都有其配置管理器，这使我们能够动态更改系统不同组件的配置。配置管理器使用YAML文件来存储和解析配置。 YAML文件已预先加载了一些初始配置，例如主服务器地址/端口，租约超时，最大块大小等。我们还有一个系统记录器，该记录器包装了Google glog库以进行记录。在文件系统的所有组件中使用它进行日志记录。

# Performance and Benchmarks

* **读取**

我们使用单个客户端操作对读取进行基准测试，读取读取长度在1KB至100MB之间的相同文件块，同时确保每个请求都不会重用元数据缓存，而是从主服务器重新获取它以获得清晰的图片非缓存读取性能。

* **并发读**

我们还基准化了文件读取延迟如何随并发读取客户端数量的增加而变化。事实是并发客户端读取同一文件越多，延迟就越好，并且如果我们读取同一块，则延迟似乎保持不变。乍一看似乎违反直觉，但是由于我们正在对同一文件的并发读取进行基准测试，因此可以预期。**由于我们在服务器端也同时处理文件读取，同时共享一个文件管理器，因此来自一个客户端读取请求的数据缓存已经支付了从磁盘获取数据的代价，因此其他并发客户端读取请求可以利用在服务器内存中的缓存，并获得更好的读取延迟。**

* **创建文件**
* **写操作**
* **创建+写操作**

# Discussion

根据当前的发展可以进行几个扩展。

1. 添加元数据状态的操作日志和检查点。
2. 基于当前体系结构开发多主服务器GFS。（算了算了）

为了实现此目标，必须采取几个步骤。首先，我们需要实现共识算法（例如Paxos，Raft）以同步存储在主服务器上的状态。其次，我们需要DNS服务来解析名称，以便可以以透明方式将服务器地址转换为客户端（例如，当客户端切换到其他主服务器时，故障转移应该透明地进行）。最后，主服务器的DNS服务需要负载平衡服务。这些发展需要大量的工程工作，但在将来有待探索。