- 本文主要是照搬这篇 博客 ,进行了部分勘误,并且本人做了些许补充,使得逻辑更为自洽.

摘要

GFS（Google File System）是由 我们设计并实现的为大规模分布式数据密集型应用程序设计的可伸缩（scalable）的分布式文件系统 。GFS为在廉价商用设备上运行提供了容错能力，并可以在有大量客户端的情况下提供较高的整体性能。

GFS的设计来自于我们对我们的应用负载与技术环境的观察。虽然GFS与过去的分布式文件系统有着共同的目标，但是根据我们的观察，我们的应用负载和技术环境与过去的分布式系统所做的假设有明显的不同。这让我们重新审视了传统的选择并去探索完全不同的设计。

GFS很好地满足了我们的存储需求。GFS在Google被广泛地作为存储平台部署，用于生成、处理我们服务所使用的数据或用于需要大规模数据集的研发工作。到目前为止，最大的GFS集群有上千台机器、上千块磁盘，并提供了上百TB的存储能力。

在本文中，我们介绍了为支持分布式应用程序而设计的文件系统接口的扩展，还从多方面讨论了我们的设计，并给出了小批量的benchmark与在现实场景中的使用表现。

1. 引言

为了满足Google快速增长的数据处理需求，我们设计并实现了GFS。GFS与过去的分布式系统有着很多相同的目标，如**性能（performance）、可伸缩性（scalability）、可靠性（reliability）和可用性（availability）**。但是我们的设计来自于我们对我们的应用负载与技术环境的观察。这些观察反映了与过去的分布式系统所做的假设明显不同的结果。因此，我们重新审视的传统的选择并探索了完全不同的设计。

第一，**我们认为设备故常经常发生**。GFS由成百上千台由廉价设备组成的存储节点组成，并被与其数量相当的客户端访问。设备的数量和质量决定了几乎在任何时间都会有部分设备无法正常工作，甚至部分设备无法从当前故障中分恢复。我们遇到过的问题包括：应用程序bug、操作系统bug、人为错误和硬盘、内存、插头、网络、电源等设备故障。因此，系统必须具有持续监控、错误检测、容错与自动恢复的能力。

第二，**文件比传统标准更大**。数GB大小的文件是十分常见的。每个文件一般包含很多引用程序使用的对象，如Web文档等。因为我们的数据集由数十亿个总计数TB的对象组成，且这个数字还在快速增长，所以管理数十亿个几KG大小的文件是非常不明智的，即使操作系统支持这种操作。因此，我们需要重新考虑像I/O操作和chunk大小等设计和参数。

第三，**大部分文件会以“追加”（append）的方式变更（mutate），而非“覆写”（overwrite）。在实际场景中，几乎不存在对文件的随机写入。**文件一旦被写入，即为只读的，且通常仅被顺序读取。很多数据都有这样的特征。如数据分析程序扫描的大型数据集、流式程序持续生成的数据、归档数据、由一台机器生产并同时或稍后在另一台机器上处理的数据等。鉴于这种对大文件的访问模式，追加成了为了性能优化和原子性保证的重点关注目标，而客户端中对chunk数据的缓存则不再重要。

第四，**同时设计应用程序和文件系统API便于提高整个系统的灵活性**。例如，我们放宽了GFS的一致性协议，从而大幅简化了系统，减少了应用程序的负担。我们还引入了一种在不需要额外同步操作的条件下允许多个客户端并发将数据追加到同一个文件的原子性操作。我们将在后文中讨论更多的细节。

目前，我们部署了多个GFS集群并用于不同的目的。其中最大的集群有超过1000个存储节点、超过300TB的磁盘存储，并被数百台客户端连续不断地访问。

2. 设计概述

2.1 假设

在设计能够满足我们需求的文件系统时，我们提出并遵循了一些挑战与机遇并存的假设。之前我们已经提到了一些，现在我们将更详细地阐述我们的假设。

- 系统有许多可能经常发生故障的廉价的商用设备组成。它必须具有持续监控自身并检测故障、容错、及时从设备故障中恢复的能力。
- 系统存储一定数量的大文件。我们的期望是能够存储几百万个大小为100MB左右或更大的文件。系统中经常有几GB的文件，且这些文件需要被高效管理。系统同样必须支持小文件，但是不需要对其进行优化。
- 系统负载主要来自两种读操作：大规模的流式读取和小规模的随机读取。在大规模的流式读取中，每次读取通常会读几百KB、1MB或更多。来自同一个客户端的连续的读操作通常会连续读文件的一个区域。小规模的随机读取通常会在文件的某个任意偏移位置读几KB。性能敏感的应用程序通常会将排序并批量进行小规模的随机读取，这样可以顺序遍历文件而不是来回遍历。
- 系统负载还来自很多对文件的大规模追加写入。一般来说，写入的规模与读取的规模相似。文件一旦被写入就几乎不会被再次修改。系统同样支持小规模随机写入，但并不需要高效执行。
- **系统必须良好地定义并实现多个客户端并发向同一个文件追加数据的语义**。我们的文件通常在生产者-消费者队列中或多路归并中使用。来自不同机器的数百个生产者会并发地向同一个文件追加写入数据。因此，最小化原子性需要的同步开销是非常重要的。文件在被生产后可能同时或稍后被消费者读取。
- 持续的高吞吐比低延迟更重要。我们的大多数应用程序更重视告诉处理大量数据，而很少有应用程序对单个读写操作有严格的响应时间的需求。

2.2 接口

尽管GFS没有实现像POSIX那样的标准API，但还是提供了大家较为熟悉的文件接口。文件被路径名唯一标识，并在目录中被分层组织。GFS支持如创建（create）、删除（delete）、打开（open）、关闭（close）、读（read）、写（write）文件等常用操作。

此外，**GFS还支持快照（snapshot）和追加记录（record append）操作。快照操作会以最小代价创建一个文件或一个目录树的拷贝。追加记录操作允许多个客户端在保证每个独立的客户端追加操作原子性的同时能够并发地向同一个文件追加数据。**这对实现如多路归并、生产者-消费者队列等多个客户端不需要额外的锁即可同时向同一文件追加数据非常有益。我们发现这类文件对于构建大型分布式应用程序有极高的价值。快照和追加记录的操作将分别在 章节3.4 和 章节3.3 讨论。

2.3 架构

如**图1**所示，**一个GFS集群包括单个master（主服务器）和多个chunkserver（块服务器）**，并被多个client（客户端）访问。**每个节点通常为一个运行着用户级服务进程的Linux主机**。如果资源允许且可以接受不稳定的应用程序代码所带来的低可靠性，那么可以轻松地在一台机器上同时运行chunkserver和client。

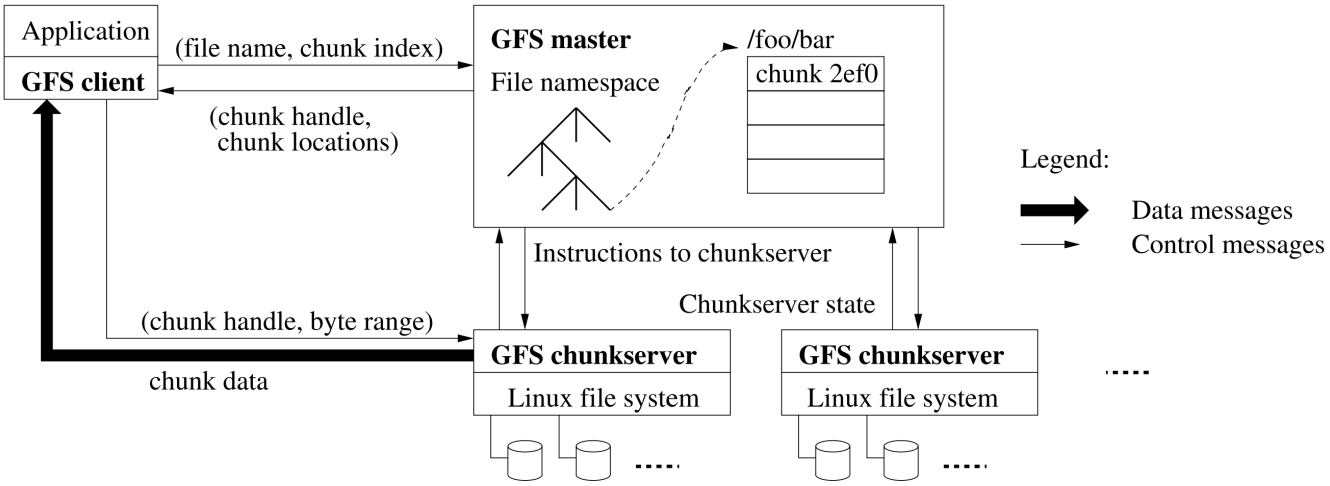


Figure 1: GFS Architecture

文件被划分为若干个固定大小的chunk（块）。**每个chunk被一个不可变的全局唯一的64位`chunk handle`（块标识符）唯一标识**，chunk handle在chunk被创建时由主节点分配。chunkserver将chunk作为Linux文件存储到本地磁盘中，通过chunk handle和byte range（字节范围）来确定需要被读写的chunk和chunk中的数据。**为了可靠性考虑，每个chunk会在多个chunkserver中有副本**。我们默认存储三份副本，用户也可以为不同的命名空间的域指定不同的副本级别。

master维护系统`所有`的元数据。元数据包括命名空间（namespace）、访问控制（access control）信息、文件到chunk的映射和chunk当前的位置。master还控制系统级活动如chunk租约（chunk lease）管理、孤儿chunk垃圾回收（garbage collection of orphaned chunks）和chunkserver间的chunk迁移（migration）。**master周期性地通过心跳（HeartBeat）消息与每个chunkserver通信，向其下达指令并采集其状态信息**

被链接到应用程序中的GFS client的代码实现了文件系统API并与master和chunkserver通信，代表应用程序来读写数据。进行元数据操作时，client与master交互。**而所有的数据（译注：这里指存储的数据，不包括元数据）交互直接由client与chunkserver间进行**。因为GFS不提供POXIS API，因此不会陷入到Linux vnode层。

无论client还是chunkserver都不需要缓存文件数据。在client中，因为大部分应用程序需要流式地处理大文件或者数据集过大以至于无法缓存，所以缓存几乎无用武之地。不使用缓存就消除了缓存一致性问题，简化了client和整个系统。（当然，client需要缓存元数据。）chunkserver中的chunk被作为本地文件存储，Linux系统已经在内存中对经常访问的数据在缓冲区缓存，因此也不需要额外地缓存文件数据。

2.4 单master

采用单master节点大大简化了我们的设计，且让master可以通过全局的信息做复杂的chunk分配（chunk placement）和副本相关的决策。然而，**我们必须最小化master节点在读写中的参与，以避免其成为系统瓶颈**。client不会直接从master读取文件数据，而是询问master它需要与哪个chunkserver通信。client会在一定时间内缓存信息，并直接与对应的chunkserver通信以完成后续操作。

让我们结合**图1**来解释一个简单地“读”操作:

- 首先，通过固定的chunk大小，client将应用程序指定的文件名和读取地在文件中的偏移量翻译为该文件中的chunk index（块序号）
- 然后，client向master发送一个包含了文件名和chunk index的请求。master会返回其相应的chunk handle和副本所在的位置。client将这个信息以文件名和chunk index为键进行缓存。

- client接着向最有可能为最近的副本所在的chunkserver发送请求。请求中指定了chunk handle和byte range。
- 之后，client再次读取相同的chunk时不再需要与master交互，直到缓存过期或文件被重新打开。

  事实上，client通常会在同一个请求中请求多个chunk，master也可以返回包含多个chunk的响应。这种方式避免了client与master进一步的通信，在几乎不需要额外开销的情况下得到更多的信息。

2.5 chunk大小

chunk大小是关键的设计参数之一。我们选择了64MB，其远大于通常的文件系统的块大小。每个chunk的副本被作为普通的Linux文件存储在chunkserver上，**其仅在需要时扩展**。懒式空间分配（lazy space allocation）避免了内部碎片（internal fragmentation）带来的空间浪费，而内部碎片可能是选择较大的chunk大小所带来的最大的不利因素。

选择较大的chunk大小提供了很多重要的优势。第一，**减少了client与master交互的次数**，因为对一个chunk的读写仅需要与master通信一次以请求其位置信息。因为我们的应用程序通常连续地读写大文件，所以减少了client与master交互的次数是尤为重要的。即使对于小规模的随机读取的情况，client也可以轻松地缓存一个数TB的数据集所有的chunk位置信息。第二，因为chunk较大，client更有可能在一个chunk上执行更多的操作，**这可以通过与chunkserver保持更长时间的TCP连接来减少网络开销**。第三，**减少了master中保存的元数据大小**。我们可以将元数据保存在master的内存中，这样做提供了更多的优势，这些优势将在 章节2.6.1 中讨论。

然而，即使有懒式空间分配，较大的chunk大小也存在着缺点。管理仅有几个chunk的小文件就是其中之一。如果多个client访问同一个文件，那么存储这这些文件的chunkserver会成为hot spot（热点）。在实际情况，因为应用程序大部分都顺序地读取包含很多chunk的大文件，所以hot spot不是主要问题。

然而在GFS首次被批处理队列（batch-queue）系统使用时，确实出现了hot spot问题：一个可执行文件被以单个chunk文件的形式写入了GFS，然后在数百台机器上启动。存储这个可执行程序的几台chunkserver因几百个并发的请求超载。**我们通过提高这种可执行文件的副本数（replication factor）并让批处理队列系统错开应用程序启动时间的方式修复了这个问题**。一个潜在的长期解决方案是在让client在这种场景下**从其他client读取数据**。

2.6 元数据

master主要存储三种元数据：**文件和chunk的命名空间（namespace）、文件到chunk的映射和chunk的每个副本的位置**。所有元数据被存储在master的**内存**中。**前两种类型（文件和块的命名空间、文件到chunk的映射）还通过将变更（mutation）记录到一个操作日志（operation log）的方式持久化存储在master的磁盘上**，并在远程机器上备份。通过日志，我们可以简单、可靠地更新master的状态，即使master故障也没有数据不一致的风险。**master不会持久化存储chunk的位置信息，而是在启动时和当chunkserver加入集群时向chunkserver询问其存储的chunk信息**。

2.6.1 内存数据结构

因为元数据被存储在内存中，master可以快速地对其进行操作。此外，在内存中存储元数据可以使master周期性扫描整个的状态变得简单高效。这种周期性的扫描被用作实现垃圾回收、chunkserver故障时重做副本、chunkserver间为了负载均衡和磁盘空间平衡的chunk迁移。 章节4.3 和 章节4.4 会进一步讨论这些活动。

这种仅使用内存的方法的一个潜在问题是chunk的数量及整个系统的容量受master的内存大小限制。在实际情况中，这并不会成为一个严重的限制。master为每个64MB的chunk维护少于64字节的元数据。因为大多数文件包含多个chunk，所以大部分chunk是满的，仅最后一个chunk被部分填充。并且因为采用了前缀压缩的方式紧凑地存储文件名，每个文件的命名空间数据通常需要少于64字节。

即使当有必要支持更大型的文件系统时，增加额外的内存的成本，远远低于通过内存存储元数据所带来的简单性、可靠性、性能和灵活性。

2.6.2 chunk位置

master不会持久化保存哪台chunkserver含有给定的chunk的副本的记录，而是简单地在启动时从chunkserver获取信息。随后，master就可以保证自己的记录是最新的，因为master控制着所有chunk的分配并通过周期性的心跳消息监控chunkserver状态。

最初我们试图让master持久化保存chunk位置信息，但是后来我们意识到在chunkserver启动时和启动后周期性请求数据要简单的多。这样做消除了当chunkserver加入或离开集群、更改名称、故障、重启等问题时，保持master和chunkserver同步的问题。在有着数百台服务器的集群中，这些事件都会经常发生。

另一种理解这种设计的方法是，chunkserver对其磁盘上有或没有哪些chunk有着最终决定权。因为chunkserver中的错误会导致chunk消失（例如磁盘可能损坏或被禁用）或一个操作者可能重命名一个chunkserver。因此，试图在master上维护一个持久化的快位置信息视图是没有以意义的。

2.6.3 操作日志

操作日志包含重要的元数据变更的历史记录。这是GFS的核心。它不仅是元数据中唯一被持久化的记录，还充当了定义并发操作顺序的逻辑时间线。带有版本号的文件和chunk都在他们被创建时由逻辑时间唯一、永久地确定。

操作日志是GFS至关重要的部分，其必须被可靠存储，且在元数据的变更被持久化前不能让client对变更可见。否则当故障发生时，即使chunk本身没有故障，但是整个文件系统或者client最近的操作会损坏。我们将操作日志备份到多台远程主机上，且只有当当前操作记录条目被本地和远程主机均写入到了磁盘后才能向客户端发出响应。master会在操作记录被写入前批量合并一些操作记录来减少写入和备份操作对整个系统吞吐量的影响。

master通过重放（replay）操作日志来恢复其文件系统的状态。操作日志要尽可能小以减少启动时间。当日志超过一定大小时，master会对其状态创建一个检查点（checkpoint），这样master就可以从磁盘加载最后一个检查点并重放该检查点后的日志来恢复状态。**检查点的结构为一个紧凑的B树（B-tree）**这样它就可以在内存中被直接映射，且在查找命名空间时不需要进行额外的解析。这进一步提高了恢复速度，并增强了系统的可用性。

因为创建一个检查点需要一段时间，所以master被设计为可以在不推迟新到来的变更的情况下创建检查点。创建检查点时，master会切换到一个新的日志文件并在一个独立的线程中创建检查点。这个新的检查点包含了在切换前的所有变更。一个有着几百万个文件的集群可以再一分钟左右创建一个检查点。当检查点被创建完成后，它会被写入master本地和远程主机的磁盘中。

恢复仅需要最后一个完整的检查点和后续的日志文件。旧的检查点和日志文件可以随意删除，不过**我们会保留一段时间以容灾**。创建检查点时发生错误不会影响日志的正确性，因为恢复代码会检测并跳过不完整的检查点。

2.7 一致性模型

GFS宽松的一致性模型可以很好地支持我们的高度分布式应用程序，且实现起来简单高效。我们将讨论GFS提供的保证和其对应用程序的意义。我们也会重点讨论GFS如何维持这些保证，但会将细节留给本论文的其他部分。

2.7.1 GFS提供的保证

文件命名空间的变更（例如创建文件）操作是原子性的。它们仅由master处理。命名空间锁保证了原子性和正确性（章节4.1）；master的操作日志定义了这些操作的全局总顺序（章节2.6.3）。

在数据变更后，无论变更的成功与否，一个文件区域（file region）的状态都取决于变更类型。**表1**总结了变更后文件区域的状态。**如果一个文件区域的 `任意一个副本` 被任何client读取总能得到相同的数据，那么这个文件区域状态为 `consistent` （一致的）**。在一个文件区域的 `数据变更后` ，**如果它是一致(`consistent`)的，且client 总能看到其写入的内容(可以完整的看到任何一次完整的的写入)，那么这个文件区域的状态为 `defined` （确定的）**。文件区域在并发变更执行后的状态为 `consistent but undefined` ：**所有客户端能看到同样的数据，但数据可能并不反映任何一个变更写入的数据**，其通常混杂了多个变更的内容。文件区域在一个失败的变更后状态会变为inconsistent（不一致的）（且undefined）：不同client在不同时刻可能看到不同的数据。下面我将描述我们的应用程序如何区分defined和undefined的区域。应用程序不需要进一步区分不同种的undefined状态。

表1 变更后文件区域状态 Table 1: File Region State After Mutation		
	Write（写入）	Record Append（记录追加）
串行成功 （Secrial success）	defined （确定的）	defined interspersed with inconsistent 确定的，但部分不一致
并发成功 （Concurrent success）	consistent but undefined （一致的但非确定的）	同上
失败（Failure）	inconsistent （不一致的）	同上

数据变更操作可能为write或record append（译注：record append操作与文件的append有所不同，下文中会有对record append的介绍）。write操作会在应用程序指定的文件与偏移处写入数据。**record append会将数据至少一次（at least once）地原子性地写入文件**，即使在record append的同时可能存在并发的变更，但是**record append写入位置是由GFS选择的偏移量**（章节3.3）。（**与常规的append不同**，append仅会在client认为的文件末尾处写入数据。）record append的偏移量会被返回到client，这个偏移量为record append写入的数据的起始位置。除此之外，GFS可能会在记录的中间插入填充（padding）和或重复的记录。它们(padding)占用的区域状态为inconsistent的，通常情况下，它们的数量远少于用户数据。

在一系列变更执行成功后，被变更的文件区域状态被保证为 `defined` 的，且该区域中包含最后一次变更写入的数据。这一点是GFS通过以下方式实现的：（a）对chunk执行变更时，其所有副本按照相同的顺序应用变更（章节3.1）（b）使用chunk版本号（chunk version）来检测因chunkserver宕机而错过了变更的陈旧的chunk副本（章节4.5）。陈旧的chunk副本永远不会在执行变更时被使用，也不会在master返回client请求的chunk的位置时被使用。它们会尽早地被作为垃圾回收。

由于client会缓存chunk的位置，在缓存信息刷新前，client可能会访问陈旧的副本(写后读问题)。这个时间窗口会受缓存过期时间和下一次打开文件限制（下一次打开文件会清除文件的所有chunk位置信息）。除此之外，由于我们大多数文件是仅追加的，陈旧的副本的通常会返回一个版本较早的结束位置处的数据，而不是陈旧的数据（译注：这里陈旧的数据指错过了write变更的数据）。当reader重试并与master通信时，它将立刻获取目前最新的chunk位置。

即使在变更被成功应用的很长时间后，设备故障仍然可以损坏（corrupt）会销毁（destroy）数据。GFS通过master和所有chunkserver周期性握手的方式来确定故障的chunkserver，并通过校验和（checksunmming）的方式检测数据损坏（章节5.2）。一旦出现问题，数据会尽快地从一个合法的副本恢复 章节4.3）。一个chunk只有在GFS作出反应前（通常在几分钟内）失去了所有的副本，chunk才会不可逆地丢失。即使在这种情况下，chunk也仅变得不可用而非损坏，因为应用程序可以收到明确的错误而非损坏的数据。（译注：本节中的“损坏corrupt”指读到错误的数据，“销毁（destory）”指数据丢失。）

2.7.2 对应用程序的影响

GFS应用程序可以通过一些简单的技术来使用其宽松的一致性模型，且这些技术已经用于其他目的，如：依赖append而不是overwrite、检查点、自验证写入（writing self-validating）、自标识记录（self-identifying records）。

在实际使用中，我们所有的应用程序都通过`append`而不是overwrite的方式对文件进行变更。其中一个典型的引用场景是：一个write从头到尾地生成一个文件。它会周期性地为已经写入的文件数据创建检查点，并在所有数据都被写入文件后自动将其重命名为一个永久的文件名。检查点可能包含应用程序级别的校验和。reader会验证文件仅处理跟上最新的检查点的文件区域，这些区域的状态一定的“defined”的。尽管这种方法有一致性和并发问题，它仍很好地满足了我们的需求。append的效率远高于随机写入，且在应用程序故障时更容易恢复。检查点机制允许writer在重启时增量写入，并能够防止reader处理那些虽然已经被成功写入文件但是从应用程序的角度看仍然不完整的文件数据。

另一种典型的用途是，许多write并发地向同一个文件append数据以获得合并后的结果或文件作为生产者-消费者队列使用。record append的“至少一次追加（append-at-least-once）”语义保证了每个write的输出。而reader偶尔需要使用下文所说的方法填充和处理重复的数据。writer为每条记录生成如校验和的额外信息，这样，记录的合法性就可被校验。一个reader通过校验和来识别并丢弃额外的填充和记录。如果rearder无法容忍偶尔发生的重复（如果重复的记录可能触发非幂等（non-idempotent）运算），它可以使用记录中的唯一标识符来进行去重。通常，在命名应用程序相关的实体时（如web文档），总会使用唯一的标识符。数据记录的I/O的充能都在库代码中（除了去重），可以被我们的应用程序使用，且其还适应于Google实现的其他文件接口。通过这些库，带有极少的重复的记录，总会被以相同顺序交付给reader。

3. 系统交互

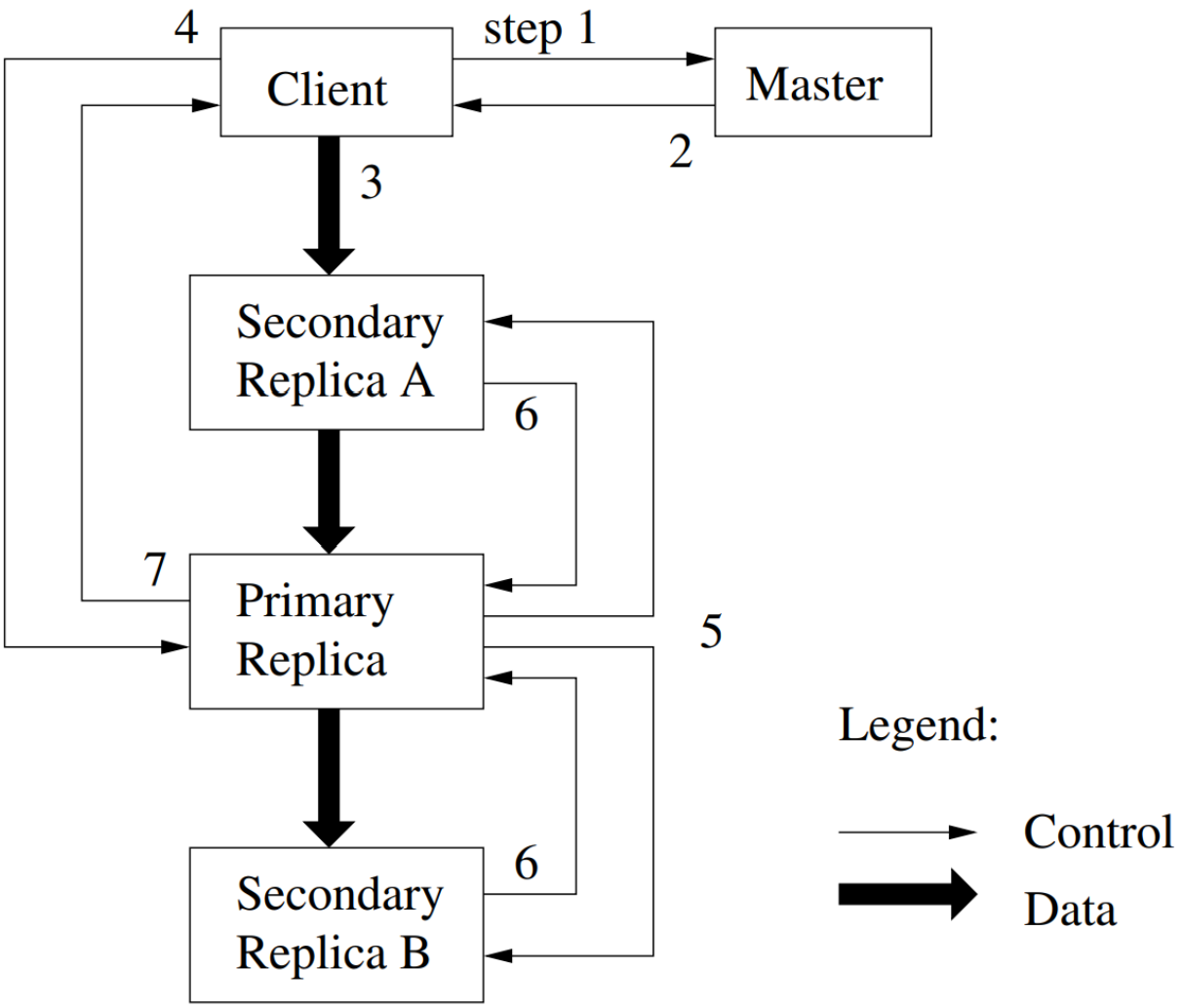
在我们设计系统时，`我们让master尽可能少地参与所有操作`。在此背景下，我们将描述client、master和chunkserver如何交互来实现数据变更、原子地record append和快照操作。

3.1 租约和变更顺序

改变chunk或元数据的操作被称为“变更(mutation)”，如write或append。chunk变更时，其每个副本都会应用变更。**我们使用租约（lease）来维护副本间变更顺序的一致性**。master向其中一份副本授权一个变更的租约，我们称这个副本为primary（译注：有时也可代指primary副本所在的chunkserver）。primary为应用于该chunk的所有变更选取顺序。所有副本都会按照这个顺序来应用变更。因此，全局的变更顺序首先由master选取的租约授权顺序定义，接着在租约内由primary选取的顺序编号定义。

这种租约机制是为了最小化master管理负载而设计的。租约的初始超时时间为60秒。然而，一旦chunk被变更，primary就可以向master请求延长租约时间，或者（通常为）接受来自master的租约时间延长操作。这些租约延长请求和租约授权请求依赖master与chunkserver间周期性地**心跳消息**来实现。有时master可能会在租约过期前试图撤销租约（例如，当master想禁止对正在被重命名的文件进行变更时）。即使master与一个primary的通信丢失，master仍可以在旧租约过期后安全地向另一个副本授权新的租约。

在**图2**中，我们将通过带编号的控制流来讲解一次write的流程。



1. client向master询问哪个chunkserver持有指定chunk的租约及该chunk的其他副本的位置。如果没有chunkserver持有租约，那么master会选择一个副本对其授权（这一步在图中没有展示）。
2. master向client回复primary副本的标识符和其他副本（也称secondary）的位置。client为后续的变更缓存这些信息。client只有当primary不可访问或primary向client回复其不再持有租约时才需要再次与master通信。
3. client将数据推送到所有副本。client可以按**任意顺序**推送。每个chunkserver都会将数据在内部的LRU中缓存，直到数据被使用或缓存老化失效（age out）。通过将数据流和控制流解耦，我们可以使用基于网络拓扑的技术来提高开销高昂的数据流的性能，且与哪台chunkserver是primary无关。章节3.2将对此进一步讨论。
4. 一旦所有副本都确认收到了数据，client会向primary发送一个write请求。这个请求标识了之前推送到所有副本的数据的作用。primary会为其收到的所有的变更（可能来自多个client）分配连续的编号，这一步提供了重要的顺序。primary对在本地按照该顺序应用变更。
5. `primary将write请求继续传递给其他secondary副本`。每个secondary副本都按照primary分配的顺序来应用变更。
6. 所有的secondary副本通知primary其完成了变更操作。
7. primary回复client。任意副本遇到的任何错误都会被报告给client。即使错误发生，write操作可能已经在primary或secondary的任意子集中被成功执行。（如果错误在primary中发生，那么操作将不会被分配顺序，也不会被继续下发到其他副本。）只要错误发生，该请求都会被认为是失败的，且被修改的区域的状态为inconsistent。client中的代码会通过重试失败的变更来处理这种错误。首先它会重试几次步骤（3）到步骤

（7），如果还没有成功，再从write请求的初始操作开始重试。

如果应用程序发出的一次write请求过大或跨多个chunk，GFS的client代码会将其拆分成多个write操作。拆分后的write请求都按照上文中的控制流执行，但是可能存在与其他client的并发的请求交叉或被其他client的并发请求覆盖的情况。因此，共享的文件区域最终可能包含来自不同client的片段。但共享的文件区域中的内容最终是相同的，因为每个操作在所有副本上都会以相同的顺序被成功执行。正如 章节2.7 中所述，这会使文件区域变为consistent but undefined状态。

3.2 数据流

为了高效地利用网络，我们对数据流与控制流进行了解耦。在 控制流 从client向primary再向所有secondary推送的同时， 数据流 沿着一条精心挑选的chunkserver链以流水线的方式线性推送。我们的目标是充分利用每台机器的网络带宽，避免网络瓶颈和高延迟的链路，并最小化推送完所有数据的时延。

为了充分利用机器的网络带宽，数据会沿着chunkserver链 线性 地推送，而不是通过其他拓扑结构（如树等）分配发送。因此，每台机器全部的出口带宽都被用来尽可能快地传输数据，而不是非给多个接受者。

为了尽可能地避免网络瓶颈和高延迟的数据链路（例如，交换机间链路（inter-switch）经常同时成为网络瓶颈和高延迟链路），每台机器会将数据传递给在网络拓扑中最近的的且还没有收到数据的机器。假设client正准备将数据推送给S1~S4。client会将数据发送给最近的chunkserver，比如S1。S1会将数据传递给S2~S4中离它最近的chunkserver，比如S2。同样，S2会将数据传递给S3~S4中离它最近的chunkserver，以此类推。由于我们的网络拓扑非常简单，所以可以通过IP地址来准确地估算出网络拓扑中的“距离”。

最后，我们通过流水线的方式通过TCP连接传输数据，以最小化时延。当chunkserver收到一部分数据时，它会立刻开始将数据传递给其他chunkserver。因为我们使用全双工的交换网络，所以流水线可以大幅减少时延。发送数据不会减少接受数据的速度。如果没有网络拥塞，理论上将B个字节传输给R个副本所需的时间为$B/T + RL$(流水线 :传输一次的总时间加上每个节点之间的传输时延和)，其中T是网络的吞吐量，L是两台机器间的传输时延。通常，我们的网络连接吐吞量T为100Mbps，传输时延L远小于1ms。

3.3 原子性record append

GFS提供了一种叫做record append的原子性append操作。在传统的write操作中，client会指定数据写入的偏移量。对同一个文件区域的并发write操作不是串行的，可能会导致该区域中不同段的数据来自多个cllient。然而在record append中，client仅需指定待追加的数据。**GFS会为其选择一个偏移量**，在该偏移量处至少**一次地原子性地将数据作为一个连续的字节序列追加到文件**，并将该偏移量返回给client。这很像Unix系统中，在不存在多writer并发写入带来的竞态条件下，写入以O_APPEND模式打开的文件的情况。

record append被大量应用在我们的有多个来自不同机器的client向同一个文件并发append数据的分布式应用程序中。如果通过传统的write操作，那么client还需要额外的复杂且开销很高的同步操作（例如分布式锁管理）。这种文件在我们的工作环境下常被作为MPSC（multiple-producer/single-consumer，多生产者单消费者）队列使用，或是作为包含了来自多个client的数据合并后的结果被使用。

record append是变更的一种，也遵循 章节3.1 中的控制流，仅在primary端稍有点额外的逻辑。在client将数据推送到所有副本的最后一个chunk之后，client会向primary发送一个请求。primary会检查当新记录追加到该chunk之后，是否会导致该chunk超过最大的chunk大小限制（64MB）。如果会超出chunk大小限制，primary会将该chunk填充到最大的大小，并通知secondary也做相同的操作，再回复客户端，使其在下一个chunk上 重试 该操作。record append操作限制了每次最多写入最大chunk大小的四分之一的数据，以保证在最坏的情况下产生的碎片在可接受的范围内。（译注：过大的请求会被拆分成多个请求，如 章节3.1 中所述。）在一般情况下，记录大小都在最大限制以内，这样primary会向数据追加到它的副本中，并通知secondary在与其追加的偏移量相同的位置处写入数据，并将最终成功操作的结果返回给client。

如果record append操作在任何一个副本中失败，那么client会重试操作。这样会导致同一个chunk的不同副本中可能包含不同的数据，这些数据可能是同一条记录的部分或完整的副本。GFS不保证所有副本在字节级别一致，其只保证record append的数据作为一个单元被原子性地至少写入一次。这一点很容易证明，因为数据必须在某个chunk的所有副本的相同偏移位置处写入。此外，在record append之后，每个副本都至少与最后一条记录一样长。这样，任何未来的新记录都会被分配到一个更高的偏移位置或者一个新chunk(不会覆盖)，即使另一个副本成为了primary也能保证这个性质。这样，被record append操作成功写入的区域在一致性方面都将是defined状态（因此也是consistent的），而这些defined区域间的文件区域是inconsistent的（因此也是undefined的）。我们应用程序会通过 章节2.7.2 中讨论的方式处理inconsistent的区域。

3.4 快照

快照操作几乎会在瞬间对一个文件或一个目录树（被称为源）完成拷贝，同时能够尽可能少地打断正在执行的变更。我们的用户使用快照操作来快速地对一个庞大的数据集的一个分支进行拷贝（或对其拷贝再进行拷贝等等），或者在实验前对当前状态创建检查点，这样就可以在试验后轻松地提交或回滚变更。

我们使用类似AFS[5]的标准的 写入时复制技术 来实现快照。当master收到快照请求的时候，它首先会撤销快照涉及到的文件的chunk上所有未完成的租约。这确保了对这些chunk在后续的写入时都需要与master交互以查找租约的持有者。这会给master优先拷贝这些chunk的机会。

在租约被收回或过期后，master会将快照操作记录到日志中，并写入到磁盘。随后，master会通过在内存中创建一个源文件或源目录树的元数据的副本的方式来进行快照操作。新创建的快照文件与源文件指向相同的chunk。

在快照操作后，首次想要对chunk C 进行write操作的client会向master发送一个请求以找到当前的租约持有者。master若检测到chunk C 的引用数超过1个。master会推迟对client的响应，并选取一个新的chunk handler C'。接着，master请求每个当前持有chunk C 副本的chunkserver去创建一个新chunkC'。通过在与源chunk相同的chunkserver上创建新chunk，可以保证数据 只在本地拷贝 ，而不会通过网络拷贝（我们的磁盘大概比100Mb的以太网连接快3倍左右）。在这之后，请求的处理逻辑就与处理任何其他chunk的请求一样了：master向新chunkC'的一个副本授权租约并将其响应client的请求。这样，client就可以像平常一样对chunk进行write操作，且client并不知道这个chunk是刚刚从一个已有的chunk创建来的。

4. master操作

master 执行所有命名空间操作 。除此之外， `master还管理整个系统中chunk的副本` ：master做chunk分配（placement）决策、创建新chunk与副本、协调各种系统范围的活动以保持chunk副本数饱和、平衡所有chunkserver的负载并回收未使用的存储。现在我们将讨论这些主题。

4.1 命名空间管理与锁

master的很多操作可能消耗很长时间，例如：快照操作必须收回其涉及到的所有chunk所在的chunkserver的租约。当这些操作执行时，我们不希望推迟master的其他操作。因此，我们允许同时存在多个运行中的操作，并对命名空间的不同区域使用 锁 机制来保证操作正确地串行执行。

不像很多传统的文件系统，GFS没有用来记录目录中有哪些文件的数据结构。GFS也不支持对同一个文件或目录起别名（alias）（如Unix系统中的硬 链接 （hard link）或软链接（symbolic link））。GFS在逻辑上用一个完整路径名到元数据的查找表(LUT)来表示命名空间。通过前缀压缩技术，这个查找表可在内存中高效地表示。在命名空间树上的每个节点（既可能是一个文件的绝对路径名，也可能是一个目录的绝对路径名）都有一个与之关联的 读写锁 （read-write lock）。

master的每个操作执行前都会请求一系列的锁。通常，如果master的操作包含命名空间/d1/d2/…/dn/leaf，master会在目录/d1、/d1/d2，…，/d1/d2/…/dn上请求读取锁（read lock），并在完整路径名/d1/d2/…/dn/leaf/上请求读取锁或写入锁。其中，leaf可能是文件或者目录，这取决于执行的操作。

现在，我们将说明锁机制如何在/home/user/正在被快照到/save/user/时，防止/home/user/foo/被创建。快照操作会在/home和/save上请求读取锁、在/home/user和/save/user上请求写入锁。文件创建操作需要在/home和/home/user上请求读取锁，在/home/user/foo上请求写入锁。由于它们试图在/home/user上获取锁时发生冲突，因此这两个操作可以正确地串行执行。因为GFS中没有目录数据结果或像inode一样的数据结构，所以无需在修改时对其进行保护，因此在文件创建操作时不需要获取其父目录的写入锁。其父目录上的读取锁已经足够保护其父目录不会被删除。

这种锁机制提供了一个非常好的性质：允许在同一目录下并发地执行变更。例如，在同一目录下的多个文件创建操作可以并发执行：每个文件创建操作都获取其父目录的读取锁与被创建的文件的写入锁。目录名上的读取锁足够防止其被删除、重命名或快照。文件名上的写入锁可以防止相同同名文件被创建两次。

因为命名空间可能含有很多的结点，所以读写锁对象会在使用时被懒式创建，并一旦其不再被使用就会被删除。此外，为了防止死锁，锁的获取顺序总是一致的：首先按照命名空间树中的层级排序，在同一层级内按照字典顺序排序。

当前锁的状态	读锁请求	写锁请求
无锁	可以	可以
读锁	可以	阻塞
写锁	阻塞	阻塞

4.2 副本分配

GFS集群在多个层级上都高度分布。GFS通常有数百个跨多个机架的chunkserver。这些chunkserver可能会被来自相同或不同机架上的数百个clienet访问。在不同机架上的两台机器的通信可能会跨一个或多个交换机。另外，一个机架的出入带宽可能小于这个机架上所有机器的出入带宽之和。多层级的分布为数据的可伸缩性、可靠性和可用性带来了特有的挑战。

chunk副本分配策略有两个目标：最大化数据可靠性和可用性、最大化网络带宽的利用。对于这两个目标，仅将副本分散在所有机器上是不够的，这样做只保证了容忍磁盘或机器故障且只充分利用了每台机器的网络带宽。我们必须在机架间分散chunk的副本。这样可以保证在一整个机架都被损坏或离线时（例如，由交换机、电源电路等共享资源问题引起的故障），chunk的一些副本仍存在并保持可用状态(可靠性)。除此之外，这样还使对chunk的流量（特别是读流量）能够充分利用多个机架的总带宽。而另一方面，写流量必须流经多个机架，这是我们资源做出的权衡。

在服务器领域，"rack"（机架）和"machine"（机器）是常用的术语。

机架（Rack）：机架是一种用于存放服务器和其他网络设备的框架结构。它通常是一个独立的金属架构，具有标准尺寸和规范，能够容纳多个设备并进行组织和管理。一个机架通常有固定数量的垂直U（单位）空间，每个U的高度为1.75英寸（约为4.45厘米），用于安装服务器、交换机、存储设备等。

机器（Machine）：在服务器环境中，"机器"通常指的是一个独立的物理服务器。这是一个实体设备，通常是一个框架式结构，包含处理器、内存、存储设备和其他必要的组件，用于运行应用程序、存储数据和提供服务。一台机器可以是一个独立的服务器，也可以是一个服务器机架中的一个单独单元，其中包含多个机器。

在一个典型的服务器数据中心中，机架被用于组织和安装多个机器。每个机架可以容纳多个机器，每个机器都可以独立运行和管理，但它们可能 共享 一些资源和网络连接。通过将机器放置在机架中，可以实现更高效的空间利用、易于管理和维护以及更好的冷却和电源管理。

4.3 chunk创建、重做副本、重均衡

chunk副本的创建可能由三个原因引起：chunk创建、重做副本（re-replication）和重均衡（rebalance）。

当master创建一个chunk的时候，它会选择初始化空副本的位置。位置的选择会参考很多因素：（1）`我们希望在磁盘利用率低于平均值的chunkserver上放置副本`。随着时间推移，这样将平衡chunkserver间的磁盘利用率（2）我们希望`限制每台chunkserver上最近创建的chunk的数量`。尽管创建chunk本身开销很小，但是由于chunk写入时创建的，且在我们的一次追加多次读取（append-once-read-many）的负载下chunk在写入完成后经常是只读的，所以master还要会可靠的预测即将到来的大量的写入流量。（3）对于以上讨论的因素，我们希望将chunk的副本跨机架分散。

`当chunk可用的副本数少于用户设定的目标值时，master会重做副本副本`。chunk副本数减少可能有很多种原因，比如：chunkserver可能变得不可用、chunkserver报告其副本被损坏、chunkserver的磁盘因为错误变得不可用、或者目标副本数增加。每个需要重做副本的chunk会参考一些因素按照优先级排序。其中之一是当前chunk副本数与目标副本数之差。例如，我们给失去两个副本的chunk比仅失去一个副本的chunk更高的优先级。另外，我们更倾向于优先为还存在(alive)的文件的chunk重做副本，而不是优先为最近被删除的文件（见 **章节4.4**）重做。最后，为了最小化故障对正在运行的应用程序的影响，我们`提高了所有正在阻塞client进程的chunk的优先级`。

`master选取优先级最高的chunk，并通过命令若干chunkserver，直接从一个存在且合法的副本来克隆这个chunk`。新副本位置的选取与创建新chunk时位置选取的目标类似：均衡磁盘空间利用率、限制在单个chunkserver上活动的克隆操作数、在机架间分散副本。为了防止克隆操作的流量远高于client流量的情况发生，master需要对整个集群中活动的克隆操作数和每个chunkserver上活动的克隆操作数进行限制。除此之外，在克隆操作中，每个chunkserver还会限制对源chunkserver的读请求，以`限制每个克隆操作占用的总带宽`。

最后，`每隔一段时间master会对副本进行重均衡`：master会检测当前的副本分布并移动副本位置，使磁盘空间和负载更加均衡。同样，在这个过程中，master会`逐渐填充`一个新的chunkserver，而不会立刻让来自新chunk的高负荷的写入流量压垮新的chunkserver。新副本放置位置的选择方法与我们上文中讨论过的类似。此外，master必须删除一个已有副本。通常，master会选择删除空闲磁盘剩余空间低于平均的chunkserver上的副本，以均衡磁盘空间的使用。

4.4 垃圾回收

在文件被删除后，GFS不会立刻回收可用的物理存储空间。master仅在周期性执行 `懒式垃圾回收` 时回收物理存储空间，其中垃圾回收分为 `文件级垃圾回收和chunk级垃圾回收` 。我们发现这种方法可以让系统更为简单可靠。

4.4.1 垃圾回收机制

当一个文件被应用程序删除时， `master会像执行其他操作时一样立刻将删除操作写入日志` 。但是master不会立刻对资源进行回收，而是 `将待删除的文件重命名为一个带有删除时间戳的隐藏文件名` 。当master周期性地扫描文件系统命名空间时， `它会删除已经存在超过三天（用户可以配置这个间隔时间）的这种隐藏文件` 。在文件被彻底删除之前，仍可通过该文件被重命名后的特殊的新文件名对其进行访问，也可以通过将其重命名为正常文件的方式撤销删除。当隐藏文件被从命名空间中移除时，其在内存中的元数据也会被删除。这种方式可以有效地切断文件和其对应的chunk的链接。

和上文介绍的文件级垃圾回收类似，在进行chunk级垃圾回收时，master会周期性扫描chunk命名空间，并找出孤儿chunk（ `orphaned chunk` ）（例如哪些无法被任何文件访问到的chunk）并删除这些chunk的 `元数据` 。在chunkserver周期性地与master进行心跳消息交换时，chunkserver会报告其拥有的chunk的子集，而master会回复这些chunk中元数据已经不存在的chunk的标识。chunkserver可以自由地删除这些元数据已经不存在的chunk的副本。

4.4.2 关于垃圾回收的讨论

分布式系统垃圾回收通常是一个很困难的问题，其往往需要在编程时使用复杂的解决方案。但是在我们的场景下它非常简单。因为文件到chunk的映射由master专门管理，所以我们可以轻松地识别所有chunk的引用。同样，因为chunk的副本在每个chunkserver上都是Linux系统中指定目录下的文件，所以我们也可以轻松地识别所有chunk的副本。所有master中没有记录的副本都会被视为垃圾。

这种暂存待回收文件的垃圾回收方法相比饿汉式回收有很多优势。第一， `这种方法在设备经常出现故障的大规模可伸缩分布式系统中非常简单可靠` 。chunk的创建可能仅在部分chunkserver上成功而在其他chunkserver上失败，这样会导致系统中出现master不知道的副本。且副本删除消息可能会丢失，这样master在其自身和chunkserver故障时都必须重新发送该消息。垃圾回收机制为清理那些不知道是否有用的副本提供了一个统一且可靠的方法。第二，垃圾回收机制将对存储空间的回收操作合并为master的后台活动，如周期性扫描命名空间和周期性地与chunkserver握手。因此，垃圾回收机制可以 `分批回收存储空间并平摊回收的开销` 。另外，垃圾回收仅在master相对空闲时执行。这样，master可以更迅速的相应需要及时响应的来自client的请求。第三，延迟回收存储空间可以 `防止意外的不可逆删除操作` 。

根据我们的实际经验，延迟回收的主要缺点是：当用户存储空间紧张时， `延迟回收会让用户难以释放存储空间` 。快速创建并删除临时文件的应用程序可能无法立刻重用存储空间。为了解决这个问题，我们在用户再次显示删除已删除文件时，加快了对存储空间的回收。同时，我们允许用户对不同的命名空间应用不同的副本与回收策略。例如，用户可以指定某个目录树下的所有文件都不需要副本，且当这个目录树下的文件被删除时立刻且无法撤销地将其从文件系统中移除。

4.5 陈旧副本检测

如果chunkserver因故障离线时错过了对其中的chunk的变更，那么该chunkserver中chunk的副本会变为陈旧的副本。 `master会为每一个chunk维护一个chunk版本号（chunk version number），用来区分最新的和陈旧的副本` 。

master每当为一个chunk授权新租约时，都会增加chunk的版本号并同时其最新的副本。master和这些副本都持久化保存这个新版本号。这一步发生在master响应任何client前，即在chunk可以被写入前。如果一个副本当前不可用，那么这个副本的chunk版本号不会增长。这样，当这个chunkserver重启时并向master报告其包含的chunk和chunk对应的版本号时，master会检测出这个chunkserver中的副本是陈旧的。如果master收到了比它的记录中更高的chunk版本号，master会认为其授权租约失败，并将更高的版本号视为最新的版本号。

master在周期性垃圾回收时会删除陈旧的副本。即使在master回收陈旧副本之前，当client向master请求该副本的chunk时，master仍会认为该陈旧的副本不存在。另一种保护措施是，当master通知client哪个chunkserver持有指定chunk的租约时，和当master在克隆操作中命令一个chunkserver从另一个chunkserver读取chunk时，其请求中需要带有chunk的版本号。client或者chunkserver会在执行操作时验证版本号以确保其始终在操作最新的数据。

5. 错误容忍与诊断

在我们设计GFS时，最大的挑战之一就是处理经常发生的设备故障。设备的质量和数量让故障发生不再是异常事件，而是经常发生的事。我们既无法完全信任机器，也无法完全信任磁盘。设备故障可能导致系统不可用，甚至会导致数据损坏。我们将讨论我们是如何应对这些挑战的，以及系统内建的用来诊断系统中发生的不可避免的问题的工具。

5.1 高可用

在由数百台服务器组成的GFS集群中，在任意时间总会有一些服务器不可用。我们通过两个简单但有效的策略保证整个系统高可用：快速恢复和副本。

5.1.1 快速恢复

在master和chunkserver的设计中，它们都会保存各自的状态，且无论它们以哪种方式终止运行，都可以在数秒内重新启动。事实上，我们并不区分正常终止和非正常的终止。通常，服务会直接被通过杀死进程的方式终止。当client和其他服务器的请求超时时，它们会在发生一个时间很短的故障，并随后重新连接到重启后的服务器并重试该请求。章节6.2.2 中有启动时间相关的报告。

5.1.2 chunk副本

正如之前讨论的，每个chunk会在不同机架的多个chunkserver上存有副本。用户可以为不同命名空间的文件制定不同的副本级别。副本级别默认为3。当有chunkserver脱机或通过校验和（见 章节5.2）检测到损坏的副本时，master根据需求克隆现有的副本以保证每个chunk的副本数都是饱和的。尽管副本策略可以很好地满足我们的需求，我们还是探索了其他形式的跨服务器的冗余策略以满足我们日益增长的只读数据存储需求，如：奇偶校验码（parity code）或擦除码（erasure code）。因为我们的流量主要来自append和读操作，而不是小规模的随机写操作，所以我们希望在松散耦合的系统中，既有挑战性又要可管理地去实现这些复杂的冗余策略。

5.1.3 master副本

为了保证可靠性，master的状态同样有副本。master的操作日志和检查点被在多台机器上复制。只有当变更在被日志记录并被写入master本地和所有master副本的磁盘中后，这个变更才被认为是已提交的。为了简单起见，一个master进程既要负责处理所有变更又要负责处理后台活动，如垃圾回收等从内部改变系统的活动。当master故障时，其几乎可以立刻重启。如果运行master进程的机器故障或其磁盘故障，在GFS之外的负责监控的基础架构会在其它持有master的操作日志副本的机器上启动一个新的master进程。client仅通过一个规范的命名来访问master结点（例如gfs-test），这个规范的命名是一个DNS别名，其可以在master重新被分配到另一台机器时被修改为目标机

器。

此外，“影子”master节点（“shadow” master）可以提供只读的文件系统访问，即使在主master结点脱机时它们也可以提供服务。因为这些服务器可能稍稍滞后于主master服务器（通常滞后一秒不到），所以这些服务器是影子服务器而非镜像服务器。这些影子master服务器增强了那些非正在被变更的文件和不介意读到稍旧数据的应用程序的可用性。实际上，由于文件内容是从chunkserver上读取的，所以应用程序不会读取到陈旧的文件内容。能够在一个很短的时间窗口内被读取到的陈旧的数据只有文件元数据，如目录内容和访问控制信息。

为了让自己的元数据跟随主master变化，影子master服务器会持续读取不断增长的操作日志副本，并像主master一样按照相同的顺序对其数据结构应用变更。像主master一样，影子master服务器也会在启动时从chunkserver拉取数据来获取chunk副本的位置（启动后便很少拉取数据），并频繁地与chunkserver交换握手信息来监控它们的状态。只有因主master决定创建或删除副本时，影子master服务器上的副本位置才取决于主master服务器。

5.2 数据完整性

每个chunkserver都使用校验和来检测存储的数据是否损坏。由于GFS集群通常在数百台机器上有数千chunk磁盘，所以集群中经常会出现磁盘故障，从而导致数据损坏或丢失（第七章 中介绍了一个诱因）。我们可以通过chunk的其他副本来修复损坏的chunk，但不能通过比较chunkserver间的副本来检测chunk是否损坏。除此之外，即使内容不同的副本中的数据也可能都是合法的：GFS中变更的语义（特别是前文中讨论过的record append）不会保证副本完全相同。因此，每个chunkserver必须能够通过维护校验和的方式独立的验证副本中数据的完整性。

一个chunk被划分为64KB的block。每个block有其对应的32位校验和。就像其他元数据一样，校验和也在内存中保存且会被通过日志的方式持久化存储。校验和与用户数据是分开存储的。

对于读取操作，无论请求来自client还是其他chunkserver，chunkserver都会在返回任何数据前校验所有包含待读取数据的block的校验和。因此，chunkserver不会将损坏的数据传给其他机器。如果一个block中数据和记录中的校验和不匹配，那么chunkserver会给请求者返回一个错误，并向master报告校验和不匹配。随后，请求者会从其他副本读取数据，而master会从该chunk的其他副本克隆这个chunk。当该chunk新的合法的副本被安置后，master会通知报告了校验和不匹配的chunkserver删除那份损坏的副本。

校验和对读取性能的影响很小。因为我们的大部分读操作至少会读跨几个block的内容，我们只需要读取并校验相对少量的额外数据。GFS客户端代码通过尝试将读取的数据与需要校验的block边界对其的方式，进一步地减小了校验开销。除此之外，chunkserver上校验和的查找与比较不需要I/O操作，且校验和计算操作经常与其他操作在I/O上重叠，因此几乎不存在额外的I/O开销。

因为向chunk末尾append数据的操作在我们的工作负载中占主要地位，所以我们对这种写入场景的校验和计算做了大量优化。在append操作时，我们仅增量更新上一个block剩余部分的校验和，并为append的新block计算新校验和。即使最后一个block已经损坏且目前没被检测到，增量更新后的该block的新校验和也不会与block中存储的数据匹配。在下一次读取该block时，GFS会像往常一样检测到数据损坏。

相反，如果write操作覆盖了一个chunk已存在的范围，那么我们必须读取并验证这个范围的头一个和最后一个block，再执行write操作，最后计算并记录新的校验和。如果我们没有在写入前校验头一个和最后一个block，新的校验和可能会掩盖这两个block中没被覆写的区域中存在的数据损坏问题。

chunkserver可以在空闲期间扫描并验证非活动的chunk的内容。这样可以让我们检测到很少被读取的chunk中的数据损坏。一旦检测到数据损坏，master可以创建一个新的未损坏的副本并删除损坏的副本。这样可以防止master将chunk的非活动的但是已损坏的副本识别成数据合法的副本。

5.3 诊断工具

全面且详细的诊断日志以极小的开销为问题定位、调试和性能分析提供了很大的帮助。如果没有日志，理解机器间短暂且不重复的交互将变得非常困难。GFS服务器会生成用来记录重要事件（如chunkserver上线或离线）和所有RPC请求与响应的诊断日志。这些诊断日志可以随意删除，不会影响到系统正确性。不过，如果磁盘空间允许，我们将尽可能地保持这些日志。

RPC日志包括通过网络收发的请求和响应中除读写的文件数据之外的详细内容。在诊断问题时，我们可以通过整合不同机器中的日志并将请求与响应匹配的方式，重建整个交互历史。同样，这些日志也可用来跟踪压力测试、性能分析等情况。

因为日志是顺序且异步写入的，因此日志对性能的影响非常小，并带来了很大的好处。其中最近的事件也会在内存中保存，以便在持续的在线监控中使用。

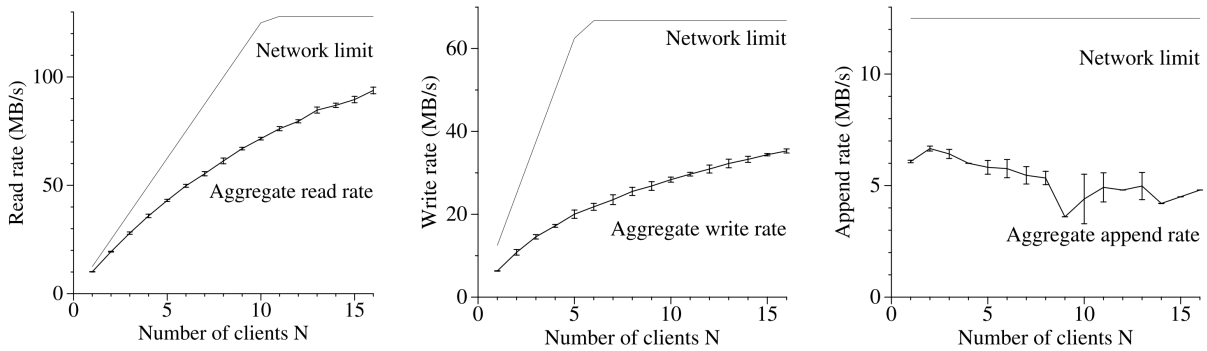
6. 性能测试

在本章中，我们将展示一些小批量的benchmark，以说明在GFS架构和实现中的瓶颈。我们还将展示一些Google在真是集群中使用时的一些指标。

6.1 小批量benchmark

我们在一个由1个master、2个master副本、16个chunkserver和16个client组成的GFS集群中测量性能表现。该配置的选择仅为了便于测试。通常一个GFS集群会由数百个chunkserver和数百个client组成。

所有的机器都采用双核1.4GHz的奔腾III处理器、2GB内存、两块5400转的80GB磁盘和100Mbpc全双工以太网，并连接到一台HP2524交换机。其中所有的19台GFS服务器都连接到同一台交换机，所有的16台client机器都连接到另一台交换机。这两个交换机之间通过1Gbps连接。



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

Figure 3: Aggregate Throughputs. Top curves show theoretical limits imposed by our network topology. Bottom curves show measured throughputs. They have error bars that show 95% confidence intervals, which are illegible in some cases because of low variance in measurements.

图3 总吞吐量（上面的曲线表示在网络拓扑中的 理论极限 。下面的曲线表示 测量到的吞吐量 。测量结果曲线显示了95%置信区间的误差柱，在一些情况下，由于测量值的方差很低，置信区间在图中难以辨认。

6.1.1 read操作

N个client同时从GFS读取数据。每个client从320GB的数据集中随机选取4MB的区域读取。读操作将重复256次，即每个client最终将读取1GB的数据。chunkserver总计有32GB内存，因此我们预测读操作中最多10%命中Linux缓冲区缓存。我们的测试结果应该接近冷缓存的结果。

图3(a)展示了N个client的总读取速率和理论速率上限。整体的理论速率在125MB/s时达到峰值，此时两个交换机间的1Gbps的链路达到饱和；或者每个client的理论速率在12.5MB/s时达到峰值，此时它的100Mbps的网络接口达到饱和。当仅有一台client在读取时，我们观测到其读取速率为10MB/s，在每台client理论上限的80%。当16个client一起读取时，总读取速率达到了94MB/s，大致达到了理论上限125MB/s的75%，平均每个client的读取速率为6MB/s。因为reader的数量增加导致多个reader从同一个chunkserver读取的概率增加，所以读取速率从理论值的80%下降到了75%。

6.1.2 write操作

N个client同时向NNN个不同的文件写入。每个client通过一系列的1MB的写操作向一个新文件写入总计1GB数据。**图3(b)**展示了整体的写入速率和理论速率上限。因为我们需要将每个字节写入16个chunkserver中的三个，每个chunkserver的连接输入速率为12.5MB/s，所以整体的理论写入速率上限为67MB/s。

> 译注：67MB/s的理论写入速率上限的计算方式为如下。因为集群中总计有16个chunkserver，每个chunkserver的100Mbps全双工连接为输入速率为12.5MB/s。数据有3份副本。因此，当所有chunkserver的连接输入全部饱和时，写入的速率为12.5MB/s×16÷3=67MB/s。根据在 **章节3.2** 中对数据流的介绍可知，在数据写入时，client仅与chunkserver中的primary副本间有一次完整的数据传输，其他secondary副本数据均通过chunkserver递交。因此在本实验的集群中，每个chunkserver的连接输入饱和时，两个交换机建的数据传输速率为67MB/s，即数据写入的速率。小于交换机间的最大传输速率1Gbps，因此不会因交换机间的连接产生瓶颈。

实验观测到的每个client的写入速率为6.3MB/s，大概是理论上限的一半。网络栈是造成这一现象的罪魁祸首。在我们使用流水线的方式将数据推送到chunk副本时，网络栈的表现不是很好。数据从一个副本传递给另一个副本的时延降低了整体的写入速率。

16个client的整体写入速率达到了35MB/s，大概是理论上限的一半。与读取相同，当client的数量增加时，更有可能出现多个client并发地向同一个chunkserver写入的情况。此外，因为write操作需要向3份不同的副本写入，所以16个writer比16个reader更有可能出现碰撞的情况。write操作比我们预想的要慢。但是在实际环境中，这并不是主要问题。即使它增加了单个client的时延，但是在有大量client的情况下它并没有显著影响系统的整体写入带宽。

6.1.3 record append操作

图3(c)展示了record append操作的性能表现。NNN个client同时向同一个文件append数据。其性能受存储该文件最后一个chunk的chunkserver的网络带宽限制，与client的数量无关。当仅有1个client时，record append的速率为6.0MB/s，当client的数量增加到16个时，速率下降到4.8MB/s。网络拥塞和不同client的网络传输速率不同是造成record append速率下降的主要原因。

在实际环境中，我们的应用程序往往会并发地向多个这样的文件追加数据。换句话说，即N个client同时地向M个共享的文件append数据，其中N与M均为数十或数百。因此，实验中出现的chunkserver的网络拥塞在实际环境中并不是大问题，因个client可以在chunkserver忙着处理一个文件时向另一个文件写入数据。

6.2 现实中的集群

现在我们来考察在Google中使用的两个集群，它们代表了其他类似的集群。集群A是数百个工程师常用来研究或开发的集群。其中典型的任务由人启动并运行几个小时。这些任务会读几MB到几TB的数据，对其分析处理，并将结果写回到集群中。集群B主要用于生产数据的处理。其中的任务持续时间更长，会不断地生成数TB的数据集，且偶尔才会有人工干预。在这两种情况中，每个任务都有许多过程进程组成，这些进程包括许多机器对许多文件同时的读写操作。

6.2.1 存储

正如**表2**所示，两个集群都有数百个chunkserver，有数TB的磁盘存储空间，且大部分存储空间都被使用，但还没满。其中“已使用空间”包括所有chunk的副本占用的空间。几乎所有文件都以3份副本存储。因此，集群分别存储了18TB和52TB的数据。

这两个集群中的文件数相似，但集群B中停用文件（dead file）比例更大。停用文件即为被删除或被新副本替换后还未被回收其存储空间的文件。同样，集群B中chunk数量更多，因为其中文件一般更大。

表2 两个GFS集群的特征		
集群	A	B
Chunkserver数量	342	227
可用磁盘空间 已使用空间	72 TB 55 TB	180 TB 155 TB
文件数 停用文件数 chunk数	735 k 22 k 992 k	737 k 232 k 1550 k
chunkserver元数据大小 master元数据大小	13 GB 48 MB	21 GB 60 MB

6.2.2 元数据

在chunkserver中，总共存储了数十GB的元数据，其中大部分是用户数据的每64KB大小的block的校验和。除此之外，chunkserver中的保存元数据只有 **章节4.5** 中讨论的chunk版本号。大部分的文件元数据是文件名，我们对其采用前缀压缩的形式存储。其他的文件元数据包括文件所有权和权限、文件到chunk的映射、每个chunk当前的版本号。除此之外，我们还存储了chunk当前的副本位置和chunk的引用计数（以实现写入时拷贝等）。

无论是chunkserver还是master，每个服务器中仅有50MB到100MB元数据。因此，服务器恢复的速度很快。服务器只需要几秒钟的时间从磁盘读取元数据，随后就能应答查询请求。然而，master的恢复稍微有些慢，其通常需要30到60秒才能恢复，因为master需要从所有的chunkserver获取chunk的位置信息。

6.2.3 读写速率

表3展示了不同时间段的读写速率。两个集群在测量开始后均运行了大概一周的时间。（集群最近已因升级到新版本的GFS而重启过。）

从重启后，集群的平均写入速率小于30MB/s30MB/s30MB/s。当我们测量时，集群B正在执行以大概100MB/s写入生成的数据的活动，因为需要将数据传递给三份副本，该活动造成了300MB/s的网络负载。

读操作的速率比写操作的速率要高得多。正如我们假设的那样，整体负载主要有读操作组成而非写操作。在测量时两个集群都在执行高负荷的读操作。实际上，集群A已经维持580MB/s的读操作一周了。集群A的网络配置能够支持750MB/s的读操作，所以集群A在高效利用其资源。集群B能够支持峰值在1300MB/s的读操作，但集群B的应用程序仅使用了380MB/s。

表3 两个GFS集群的性能指标		
集群	A	B
读速率（过去一分钟） 读速率（过去一小时） 读速率（重启后至今）	583 MB/s 562 MB/s 589 MB/s	380 MB/s 384 MB/s 49 MB/s
写速率（过去一分钟） 写速率（过去一小时） 写速率（重启后至今）	1 MB/s 2 MB/s 25 MB/s	101 MB/s 117 MB/s 13 MB/s
master操作数（过去一分钟） master操作数（过去一小时） master操作数（重启后至今）	325 Ops/s 381 Ops/s 202 Ops/s	533 Ops/s 518 Ops/s 347 Ops/s

6.2.4 master的负载

表3中还展示了向master发送操作指令的速率，该速率大概在美妙200到500次左右。master可以在该速率下轻松地工作，因此这不会成为负载的瓶颈。

GFS在早期版本中，在某些负载场景下，master偶尔会成为瓶颈。当时master会消耗大量的时间来扫描包含成百上千个文件的目录以寻找指定文件。在那之后，我们修改了master中的数据结构，允许其通过二分查找的方式高效地搜索命名空间。目前，master已经可以轻松地支持每秒上千次的文件访问。如果有必要，我们还可以通过在命名空间数据结构前放置名称缓存的方式进一步加快速度。

6.2.5 恢复时间

当chunkserver故障后，一些chunk的副本数会变得不饱和，系统必须克隆这些块的副本以使副本数重新饱和。恢复所有chunk需要的时间取决于资源的数量。在一次实验中，我们杀掉集群B中的一个chunkserver。该chunkserver上有大概15000个chunk，总计约600GB的数据。为了限制重分配副本对正在运行的应用程序的影响并提供更灵活的调度策略，我们的默认参数限制了集群中只能有91个并发的克隆操作（该值为集群中chunkserver数量的40%）。其中，每个克隆操作的速率上限为6.25MB/s（50Mbps）。所有的chunk在23.2分钟内完成恢复，有效地复制速率为440MB/s。

在另一个实验中，我们杀掉了两台均包含16000个chunk和660GB数据的chunkserver。这两个chunkserver的故障导致了266个chunk仅剩一分副本。这266个块在克隆时有着更高的优先级，在2分钟内即恢复到至少两份副本的状态，此时可以保证集群中即使再有一台chunkserver故障也不会发生数据丢失。

6.3 负载分解

在本节中，我们将详细介绍两个GFS集群中的工作负载。这两个集群与 **章节6.2** 中的类似但并不完全相同。集群X用来研究和开发，集群Y用来处理生产数据。

6.3.1 方法和注意事项

这些实验结果仅包含来自client的请求，因此结果反映了我们的应用程序为整个文件系统带来的负载情况。结果中不包括用来处理client请求的内部请求和内部的后台活动，如chunkserver间传递write数据和副本重分配等。

I/O操作的统计数据来源于GFS通过RPC请求日志启发式重建得到的信息。例如，GFS的client代码可能将一个read操作分解为多个RPC请求以提高并行性，通过日志启发式重建后，我们可以推断出原read操作。因为我们的访问模式是高度一致化的，所以我们期望的错误都在数据噪声中。应用程序中显式的日志可能会提供更加准确的数据，但是重新编译并重启上千个正在运行的client是现实的，且从上千台机器上采集数据结果也非常困难。

需要注意的一点是，不要过度地推广我们的负载情况。因为Google对GFS和它的应用程序具有绝对的控制权，所以应用程序会面向GFS优化，而GFS也正是为这些应用程序设计的。虽然这种应用程序与文件系统间的互相影响在一般情况下也存在，但是这种影响在我们的例子中可能会更明显。

6.3.2 chunkserver的负载

表4展示了各种大小的操作占比。读操作的大小呈双峰分布。64KB以下的小规模read来自client从大文件查找小片数据的seek密集操作。超过512KB的大规模read来自读取整个文件的线性读取。

在集群Y中，大量的read没有返回任何数据。在我们的应用程序中（特别是生产系统中的应用程序），经常将文件作为生产者-消费者队列使用。在多个生产者并发地向同一个文件支架数据的同时，会有一个消费者读末尾的数据。偶尔当消费者超过生产者时，read即不会返回数据。集群X中这种情况出现的较少，因为在集群X中的应用程序通常为短期运行的数据分析程序，而非长期运行的分布式应用程序。

write也呈同样的双峰分布。超过256KB的大规模write操作通常是由writer中的大量的缓冲区造成的。小于64KB的小规模写操作通常来自于那些缓冲区小、创建检查点操作或者同步操作更频繁、或者是仅生成少量数据的writer。

对于record append操作，集群Y中大规模的record append操作比集群X中要高很多。因为我们的生产系统使用了集群Y，生产系统的应用程序会更激进地面向GFS优化。

表4 各种大小的操作占比（%） 对于read操作，数据大小为实际读取和传输的数据大小，而非请求读取的总大小。						
操作类型	read	write	record append			
集群	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

表5中展示了不同大小的操作中传输数据的总量的占比。对于所有类型的操作，超过256KB的大规模操作通常都是字节传输导致的。小于64KB的小规模read操作通常来自seek操作，这些读操作传输了很小但很重要的数据。

6.3.3 append vs write

record append操作在我们的系统中被大量使用，尤其是我们的生产系统。在集群X中，write操作和record append操作的操作次数比例为8:1，字节传输比例为108:1。在集群Y中，这二者的比例分别为2.5:1和3.7:1。这些数据显示了对于两个集群来说，record append操作的规模通常比write打。然而。在集群X中，测量期间record append的使用量非常的少。因此。这个测量结果可能受一两个有特定缓冲区大小的应用程序影响较大。

正如我们预期的那样，我们的数据变更负载主要来自于append而非overwrite。我们测量了primary副本上overwrite的数据总量。测量值很接近client故意overwrite数据而不append的情况。对于集群X，overwrite的操作总量低于变更操作的0.0003%，字节数占比低于总量的0.0001%。对于集群Y，这两个数据均为0.05%。尽管这个比例已经很小了，但仍比我们预期的要高。大部分的overwrite都是由client因错误或超时而重试造成的。这本质上是由重试机制造成的而非工作负载。

表5 各种大小的操作字节传输量占比（%） 对于read操作，数据大小为实际读取和传输的数据大小，而非请求读取的总大小。二者的区别为，读取请求可能会试图读取超过文件末尾的内容。在我们的设计中，这不是常见的负载。

操作类型	read		write		record append	
集群	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

6.3.4 master的负载

表6展示了对master的各种类型的请求占比。其中，大部分请求来自read操作询问chunk位置的请求（FindLocation）和数据变更操作询问租约持有者（FindLeaseLocker）。

集群X与集群Y中Delete请求量差异很大，因为集群Y存储被经常重新生成或者移动的生产数据。一些Delete请求量的差异还隐藏在Open请求中，因为打开并从头写入文件时（Unix中以“w”模式打开文件），会隐式地删除旧版本的文件。

FindMatchingFiles是用来支持“ls”或类似文件系统操作的模式匹配请求。不像给master的其他请求，FindMatchingFiles请求可能处理很大一部分命名空间，因此这种请求开销很高。在集群Y中，这种请求更加频繁，因为自动化的数据处理任务常通过检查部分文件系统的方式来了解应用程序的全局状态。相反，使用集群X的应用程序会被用户更明确地控制，通常会提交知道所需的文件名。

表6 master请求类型占比（%）		
集群	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 otder combined	0.5	0.8

7. 开发经历

在构建和部署GFS的过程中，我们经历了很多问题。其中，有些是操作问题，有些是技术问题。

最初，GFS的构思是将其作为我们生产系统的后端文件系统。随着时间推移，GFS的用途演变为包括了研究和开发任务。GFS开始时几乎不支持权限、配额之类的功能，但现在这些功能都变为GFS包含的基本功能。虽然生产系统有着良好的纪律并被良好地控制着，但用户有时却没有。因此，其需要更多的基础设施来防止用户互相干扰。

我们最大的一些问题是磁盘问题和Linux相关问题。我们的许多磁盘都想Linux驱动程序声称它们支持很多版本的IDE（译注：本文IDE指集成设备电路Intergated Drive Electronics）协议，但事实上，它们可能只能可靠地响应最近几个版本的协议。因为这些协议都非常相似，所以大部分时间驱动器都能正常工作。但协议版本偶尔不匹配就会导致驱动器和内核中所认为的驱动器的状态不一致。由于内核中的问题，数据会无法察觉地损坏。这个问题驱动我们通过校验和的方式检测数据是否损坏，同时我们修改了内核去处理协议不匹配的问题。

早些时候，由于 *fsync()* 的开销，我们在Linux2.2内核中遇到了一些问题。这个函数的开销和文件成正比，而不是和修改的部分大小成正比。这对我们使用较大的操作日志造成了问题（特别是在我们实现检查点机制以前）。我们曾经通过同步写入的方式来解决这个问题，直到迁移到Linux2.4。

另一个Linux的问题是一个读写锁。当任何地址空间的线程从磁盘换入页（读锁）或者通过 *mmap()* 函数修改地址空间（写锁）时，都必须持有这一个读写锁。我们发现系统在轻负载下的一个瞬间会出现超时问题，所以我们努力地去寻找资源瓶颈和零星的硬件故障。最终，我们发现在磁盘线程正在换入之前映射的文件时，这个读写锁阻塞了网络主线程，导致其无法将新数据映射到内存。因为我们主要受网络接口限制而非受内存拷贝带宽限制，所以我们用 *pread()* 替换了 *mmap()* 函数，其代价是多了一次额外的拷贝操作。

尽管偶尔会有问题发生，Linux代码的可用性还是帮助了我们一次又一次地探索和理解系统行为。当时机合适时，我们会改进内核并将这些改进与开源社区分享。

8. 相关工作

就像其他大型的分布式文件系统一样（如AFS[5]），GFS提供了与位置无关的命名空间，这可以允许数据为了负载均衡和容错地移动，这一操作对client是透明的。但与AFS不同，GFS将文件数据通过类似xFS[1]和Swtift[3]的方式分散到了存储服务器上，以释放集群整体性能并提高容错能力。

因为磁盘相对廉价且副本的方式比复杂的RAID[9]的方式简单很多，所以GFS仅通过副本的方式作为冗余，因此GFS会比xFS或Swift消耗更多的原始存储空间。与类似AFS、xFS、Frangipani[12]和Intermezzo[6]的文件系统不同，GFS在系文件系统接口下没有提供任何的缓存。在我们的目标工作负载中，一个应用程序几乎不会重用数据，因为其或者流式地处理一个大型数据集，或者每次仅在大型数据及中随机地seek并读取很小一部分的数据。

一些分布式文件系统移除了集中式的服务器，并依赖分布式算法来实现一致性和管理，如Frangipani、xFS、Minnesota's GFS[11]和GPFS[10]。我们选择了集中式的方法来简化设计、增强可靠性，同时还获得了灵活性。集中式的master还大大简化了复杂的chunk分配操作和重分配副本的策略，因为master已经有了大部分相关信息，且由master来控制如何变化。我们通过保持master的状态大小很小并在其他机器上有充足的副本的方式来提高容错能力。可伸缩性和高可用性（对于read操作来说）目前通过影子master服务器机制提供。master状态的变化会通过追加到预写日志的方式进行持久化。因此我们可以通过适配像Harp[7]中的主拷贝模式（primary-copy scheme）的方法，来提供比当前的一致性有更强保证的高可用性。

我们遇到了一个类似Lustre[8]的问题，即为大量client提供整体的性能。然而，我们通过将重点放在我们的应用程序的需求而不是构架一个兼容POSIX文件系统的方式，大幅简化了这个问题。除此之外，GFS假设大量的设备是不可靠的，因此容错是我们设计中的中心问题。

GFS非常接近NASD架构[4]。NASD架构基于通过网络连接的磁盘驱动器，而GFS使用一般的商用机器作为chunkserver，就像NASD的原型那样。与NASD不同是，我们的chunkserver懒式分配固定大小的chunk，而不是可变长的对象。另外，GFS实现了如负载均衡、副本重分配和恢复等在生产环境中需要的特性。

与Minnesota's GFS或NASD不同，我们不希望改变存储设备的模型。我们着重解决由已有的商用设备组成的复杂的分布式系统的日常数据处理问题。

对生产者-消费者队列的原子性record append操作解决了类似于River的分布式队列问题。River[2]使用了分布在不同机器上的基于内存的队列和谨慎的数据流控制，而GFS采用了可以被多个生产者并发追加的持久化文件。River的模型支持M:N的分布式队列，但缺少持久化存储带来的容错能力。而GFS仅支持M:1的高效的队列。多个消费者可一个读相同的文件，但必须相互协调载入的分区。

9. 结论

Google File System论证了在产品级硬件上支持大规模数据处理负载的必要特性。虽然很多设计是为我们特殊的场景定制的，但很多设计可能适用于规模和预算相似的数据处理任务。

我们根据我们当前和预期的应用程序负载和技术环境，重新考察了传统文件系统设计中的假设。我们的考察结果指向了完全不同的设计。我们视设备故障为平常事件而非异常事件。我们优化了大部分操作为追加（可能是并发追加）和读取（通常为顺序读取）的大文件。我们还扩展并放宽了标准文件系统接口来改进整个系统。

我们的系统通过持续的监控、备份关键数据、自动且快速的恢复来提供容错能力。chunk副本让我们能够容忍chunkserver故障。这些故障的频率让我们设计了一种新的在线修复机制：周期性地对client不可见的修复损坏数据，并尽快补充丢失的副本。另外，我们通过校验和的方式来检测磁盘或IDE子系统级别的数据损坏，因为GFS系统中磁盘数量很多，这类问题是非常普遍的。

我们的设计为并发执行多种任务的reader和writer提供了很高的整体吞吐量。为了实现这一点，我们将通过master进行的文件系统控制和通过chunkserver、client的数据传输分离开来。我们还通过选取较大的chunk大小和chunk租约（将数据变更授权给primary副本）的方式最小化了master对一般操作的参与度。这种方式让master变得简单，且中心化的master不会成为系统瓶颈。我们相信，通过改进网络栈，会减少当前对单个client的写入吞吐量的限制。

GFS成功地满足了我们的存储需求，并已经在Google内部作为研究、开发和生产数据处理的存储平台使用。GFS是让我们能够进一步创新并攻克web规模问题的重要工具。

致谢

感谢以下对本系统或本论文做出了贡献的人。Brain Bershad（我们的指导者）和给我我们珍贵的评论和建议的匿名评审员。Anurag Acharya、Jeff Dean和David Desjardins为系统的早期设计做出了贡献。Fay Chang致力于chunkserver间副本比较的研究。Guy Edjlali致力于存储配额的研究。Markus Gutschke致力于测试框架与安全性增强的研究。Fay Chang、Urs Hoelzle、Max Ibel、Sharon Perl、Rob Pike和Debby Wallach对本论文早期的草稿做出了评论。我们在Google的许多勇敢的同事，他们信任这个新文件系统并给我们提出了很多很有用的反馈。Yoshka在早期的测试中提供了帮助。

参考文献

[1] Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless networkfile systems. In Proceedings of the 15th ACM Symposium on Operating System Principles, pages 109–126, Copper Mountain Resort, Colorado, December 1995.

[2] 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.

[3] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using distributed diskstriping to provide high I/O data rates. Computer Systems, 4(4):405–436, 1991.

[4] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, ErikRiedel, 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.

[5] 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.

[6] InterMezzo. http://www.inter-mezzo.org , 2003.

[7] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp file system. In 13th Symposium on Operating System Principles, pages 226–238, Pacific Grove, CA, October 1991.

[8] Lustre. http://www.lustreorg , 2003.

[9] 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.

[10] FrankSchmuckand Roger Haskin. GPFS: A shared-diskfile system for large computing clusters. In Proceedings of the First USENIX Conference on File and Storage Technologies, pages 231–244, Monterey, California, January 2002.

[11] 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.

[12] 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.