The Google File System

摘要
我们已经设计和实现了 Google File System，一个适用于大规模分布式数据处理相关应用的，可扩展的分布式文件系统。它基于普通的不算昂贵的硬件设备，实现了容错的设计，并且为大量客户端提供极高的聚合处理性能。
我们的设计目标和上一个版本的分布式文件系统有很多相同的地方，我们的设计是依据我们应用的工作量以及技术环境来设计的，包括现在和预期的，都有一部分和早先的文件系统的约定有所不同。这就要求我们重新审视传统的设计选择，以及探索究极的设计要点。
这个文件系统正好与我们的存储要求相匹配。这个文件系统在 Google 内部广泛应用于作为存储平台使用，适用于我们的服务要求产生和处理数据应用，以及我们的研发要求的海量数据的要求。最大的集群通过上千个计算机的数千个硬盘，提供了数百 TB 的存储，并且这些数据被数百个客户端并行同时操作。
在这个论文里，我们展示了用于支持分布式应用的扩展文件系统接口设计，讨论了许多我们设计的方面，并且列出了我们的 micro-benchmarks 以及真实应用性能指标。

1 介绍

我们已经为 Google 迅速增长的数据处理需要而设计和实现了 Google File System(GFS)。GFS 和上一个分布式文件系统有着很多相同的设计目标，比如性能，扩展性，可靠性，以及可用性。不过，他的设计是基于我们应用的工作量和技术环境驱动的，包括现在和预期的，都有部分和上一个版本的约定有点不同。这就要求我们重新审视传统的设计选择，以及探索究极的设计要点。
首先，节点失效将被看成是正常情况，而不再视为异常情况。整个文件系统包含了几百个或者几千个由廉价的普通机器组成的存储机器，而且这些机器是被与之匹配数量的客户端机器访问。这些节点的质量和数量都实际上都确定了在任意给定时间上，一定有一些会处于失效状态，并且某一些并不会从当前失效中恢复回来。这有可能由于程序的 bug，操作系统的 bug，人工操作的失误，以及硬盘坏掉，内存，网络，插板的损坏，电源的坏掉等等。因此，持续监视，错误检测，容错处理，自动恢复必须集成到这个文件系统的设计中来。
其次，按照传统标准来看，文件都是非常巨大的。数个 GB 的文件是常事。每一个文件都包含了很多应用程序对象，比如 web 文档等等。当我们通常操作迅速增长的，由很多 TB 组成的，包含数十亿对象的数据集，我们可不希望管理数十亿个 KB 大小的文件，即使文件系统能支持也不希望。所以，设计约定和设计参数比如 I/O 操作以及 blocksize（块大小），都需要重新审查。
第三，大部分文件都是只会在文件尾新增加数据，而少见修改已有数据的。对一个文件的随机写操作在实际上几乎是不存在的。当一旦写完，文件就是只读的，并且一般都是顺序读取得。多种数据都是有这样的特性的。有些数据可能组成很大的数据仓库，并且数据分析程序从头扫描到尾。有些可能是运行应用而不断的产生的数据流。有些是归档的数据。有些是一个机器为另一个机器产生的中间结果，另一个机器及时或者随后处理这些中间结果。对于这些巨型文件的访问模式来说，增加模式是最重要的，所以我们首要优化性能的以及原子操作保证的就是它，而在客户端 cache 数据块没有什么价值。
第四，与应用一起设计的的文件系统 API 对于增加整个系统的弹性适用性有很大的好处。例如我们不用部署复杂的应用系统就可以把 GFS 应用到大量的简单文件系统基础上。我们也引入了原子的增加操作，这样可以让多个客户端同时操作一个文件，而不需要他们之间有额外的同步操作。这些在本论文的后边章节有描述。
多个 GFS 集群现在是作为不同应用目的部署的。最大的一个有超过 1000 个存储节点，超过 300TB 的硬盘存储，并且负担了持续沉重的上百个在不同机器上的客户端的访问。

2 设计概览

2.1 约定

在为我们的需要设计文件系统得时候，我们需要建立的事先约定同时具有挑战和机遇。我们早先提到的关于观测到的关键要点，现在详细用约定来说明。

- 系统是建立在大量廉价的普通计算机上，这些计算机经常故障。必须对这些计算机持续进行检测，并且在系统的基础上进行：检查，容错，以及从故障中进行恢复。
- 系统存储了大量的超大文件。我们与其有好几百万个文件，每一个超过 100MB。数 GB 的文件经常出现并且应当对大文件进行有效的管理。同时必须支持小型文件，但是我们不必为小型文件进行特别的优化。
- 一般的工作都是由两类读取组成：大的流式读取和小规模的随机读取。在大的流式读取中，每个读操作通常要读取几百 k 的数据，每次读取 1M 或者以上的数据也很常见。对于同一个客户端来说，往往会发起连续的读取操作顺序读取一个文件。小规模的随机读取通常在文件的不同位置，读取几 k 数据。对于性能有过特别考虑的应用通常会作批处理并且对他们读取的内容进行排

序，这样可以使得他们的读取始终是单向顺序读取，而不需要往回读取数据。

- 通常基于 GFS 的操作都有很多超大的，顺序写入的文件操作。通常写入操作的数据量和杜如的数据量相当。一旦完成写入，文件就很少会更改。对于文件的随机小规模写入是要被支持的，但是不需要为此作特别的优化。
- 系统必须非常有效的，明确细节的对多客户端并行添加同一个文件进行支持。我们的文件经常使用生产者/消费者队列模式，或者作为多路合并模式进行操作。好几百个运行在不同机器上的生产者，将会并行增加一个文件。其本质就是最小的原子操作的定义。读取操作可能接着生产者操作进行，消费者会同时读取这个文件。
- 高性能的稳定带宽的网络要比低延时更加重要。我们的目标应用程序一般会大量操作处理比较大块的数据，并且很少有应用要求某个读取或者写入要有一个很短的响应时间。

2.2 接口

GFS 提供了常见的文件系统的接口，虽然他没有实现一些标准的 API 比如 POSIX。文件是通过 pathname 来通过目录进行分层管理的。我们支持的一些常见操作：*create,delete,open,close,read，write* 等文件操作。另外，GFS 有 *snapshot*，*record append* 等操作。snapshort 创建一个文件或者一个目录树的快照，这个快照的耗费比较低。record append 允许很多个客户端同时对一个文件增加数据，同时保证每一个客户端的添加操作的原子操作性。这个对于多路合并操作和多个客户端同时操作的生产者/消费者队列的实现非常有用，它不用额外的加锁处理。这种文件对于构造大型分布式应用来说，是不可或缺的。snapshot 和 record append 在后边的 3.4 和 3.3 节有单独讲述。

2.3 架构

GFS 集群由一个单个的 master 和好多个 chunkserver（块服务器）组成，GFS 集群会有很多客户端 client 访问（图 1）。每一个节点都是一个普通的 Linux 计算机，运行的是一个用户级别（user-level）的服务器进程。只要机器资源允许，并且允许不稳定的应用代码导致的低可靠性，我们就可以运行 chunkserver 和 client 可以运行在同一个机器上。

在 GFS 下，每一个文件都拆成固定大小的 chunk(块)。每一个块都由 master 根据块创建的时间产生一个全局唯一的以后不会改变的 64 位的 chunk handle 标志。chunkservers 在本地磁盘上用 Linux 文件系统保存这些块，并且根据 chunk handle 和字节区间，通过 LInux 文件系统读写这些块的数据。出于可靠性的考虑，每一个块都会在不同的 chunkserver 上保存备份。缺省情况下，我们保存 3 个备份，不过用户对于不同的文件 namespace 区域，指定不同的复制级别。

master 负责管理所有的文件系统的元数据。包括 namespace，访问控制信息，文件到 chunk 的映射关系，当前 chunk 的位置等等信息。master 也同样控制系统级别的活动，比如 chunk 的分配管理，孤点 chunk 的垃圾回收机制，chunkserver 之间的 chunk 镜像管理。master 和这些 chunkserver 之间会有定期的心跳线进行通讯，并且心跳线传递信息和 chunckserver 的状态。

连接到各个应用系统的 GFS 客户端代码包含了文件系统的 API，并且会和 master 和 chunkserver 进行通讯处理，代表应用程序进行读写数据的操作。客户端和 master 进行元数据的操作，但是所有的数据相关的通讯是直接和 chunkserver 进行的。我们并没有提供 POSIX API 并且不需要和 LInux 的 vnode 层相关。

客户端或者 chunkserver 都不会 cache 文件数据。客户端 cache 机制没啥用处，这是因为大部分的应用都是流式访问超大文件或者操作的数据集太大而不能被 chache。不设计 cache 系统使得客户端以及整个系统都大大简化了（少了 cache 的同步机制）(不过客户端 cache 元数据)。chunkserver 不需要 cache 文件数据，因为 chunks 就像本地文件一样的被保存，所以 LInux 的 buffer cache 已经把常用的数据 cache 到了内存里。

2.4 单个 master

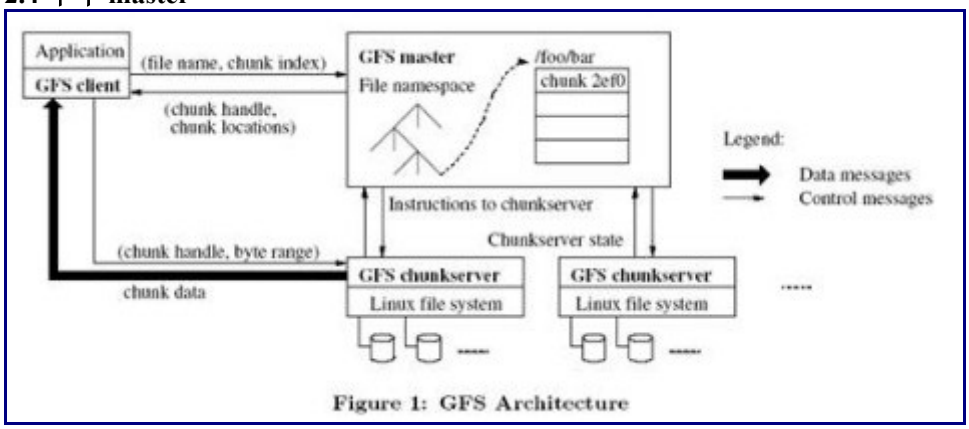

Figure 1: GFS Architecture

引入一个单个 master 的设计可以大大简化我们的设计，并且也让 master 能够基于全局的角度来管理 chunk

的存放和作出复制决定。不过，我们必须尽量减少 master 的读和写操作，以避免它成为瓶颈。客户端永远不会通过 master 来做文件的数据读写。客户端只是问 master 它应当访问那一个 chunkserver 来访问数据。客户端在一定时间内 cache 这个信息，并且在后续的操作中都直接和 chunkserver 进行操作。
这里我们简单介绍一下图 1 中的读取操作。首先，客户端把应用要读取的文件名和偏移量，根据固定的 chunk 大小，转换成为文件的 chunk index。然后向 master 发送这个包含了文件名和 chunkindex 的请求。master 返回相关的 chunk handle 以及对应的位置。客户端 cache 这些信息，把文件名和 chunkindex 作为 cache 的关键索引字。
于是这个客户端就像对应的位置的 chunkserver 发起请求，通常这个会是离这个客户端最近的一个。请求给定了 chunk handle 以及一个在这个 chunk 内需要读取得字节区间。在这个 chunk 内，再次操作数据将不用再通过客户端-master 的交互，除非这个客户端本身的 cache 信息过期了，或者这个文件重新打开了。实际上，客户端通常都会在请求中附加向 master 询问多个 chunk 的信息，master 于是接着会立刻给这个客户端回应这些 chunk 的信息。这个附加信息是通过几个几乎没有任何代价的客户端-master 的交互完成的。

2.5 chunk 块大小

chunk 的大小是一个设计的关键参数。我们选择这个大小为 64M，远远大于典型的文件系统的 block 大小。每一个 chunk 的实例（复制品）都是作为在 chunkserver 上的 Linux 文件格式存放的，并且只有当需要的情况下才会增长。滞后分配空间的机制可以通过文件内部分段来避免空间浪费，对于这样大的 chunksize 来说，（内部分段 fragment）这可能是一个最大的缺陷了。
选择一个很大的 chunk 大小提供了一些重要的好处。首先，它减少了客户端和 master 的交互，因为在同一个 chunk 内的读写操作之需要客户端初始询问一次 master 关于 chunk 位置信息就可以了。这个减少访问量对于我们的系统来说是很显著的，因为我们的应用大部分是顺序读写超大文件的。即使是对小范围的随机读，客户端可以很容易 cache 一个好几个 TB 数据文件的所有的位置信息。其次，由于是使用一个大的 chunk，客户端可以在一个 chunk 上完成更多的操作，它可以通过维持一个到 chunkserver 的 TCP 长连接来减少网络管理量。第三，它减少了元数据在 master 上的大小。这个使得我们可以把元数据保存在内存，这样带来一些其他的好处，详细请见 2.6.1 节。
在另一方面，选择一个大型的 chunk，就算是采用滞后分配空间的模式，也有它的不好的地方。小型文件包含较少树木的 chunk，也许只有一个 chunk。保存这些文件的 chunkserver 就会在大量客户端访问的时候就会成为焦点。在实践中，焦点问题不太重要因为我们的应用大部分都是读取超大的文件，顺序读取超多的 chunk 的文件的。
不过，随着 batch-queue 系统开始使用 GFS 系统的时候，焦点问题就显现出来了：一个可执行的程序在 GFS 上保存成为一个单 chunk 的文件，并且在数百台机器上一起启动的时候就出现焦点问题。只有两三个 chunkserver 保存这个可执行的文件，但是有好几百台机器一起请求加载这个文件导致系统局部过载。我们通过把这样的执行文件保存份数增加，以及错开 batchqueue 系统的各 worker 启动时间来解决这样的问题。一劳永逸的解决方法是让客户端能够互相读取数据，这样才是解决之道。

2.6 元数据

master 节点保存这样三个主要类型的数据：文件和 chunk 的 namespace，文件到 chunks 的映射关系，每一个 chunk 的副本的位置。所有的元数据都是保存在 master 的内存里的。头两个类型（namepspaces 和文件到 chunk 的映射）同时也是由在 master 本地硬盘的记录所有变化信息的 operation log 来持久化保存的，这个记录也会在远端机器上保存副本。通过 log，在 master 宕机的时候，我们可以简单，可靠的恢复 master 的状态。master 并不持久化保存 chunk 位置信息。相反，他在启动地时候以及 chunkserver 加入集群的时候，向每一个 chunkserver 询问他的 chunk 信息。

2.6.1 内存数据结构

因为元数据都是在内存保存的，master 的操作很快。另外 master 也很容易定时后台扫描所有的内部状态。定时内部状态扫描是用于实现 chunk 的垃圾回收机制，当 chunkserver 失效的时候重新复制，以及为了负载均衡和磁盘空间均衡使用的目的做 chunkserver 之间的 chunk 镜像。4.3 和 4.4 节将讨论这些操作的细节。
这种内存保存数据的方式有一个潜在的问题，就是说整个系统的 chunk 数量以及对应的系统容量是受到 master 机器的内存限制的。这个在实际生产中并不是一个很重要的限制。master 为每 64Mchunk 分配的空间不到 64 个字节的元数据。大部分的 chunks 都是装满了的，因为大部分文件都是很大的，包含很多个 chunk，只有文件的最后部分可能是有空间的。类似的，文件的名字空间通常对于每一个文件来说要求少于 64 个字节，因为保存文件名的时候是使用前缀压缩的机制。
如果有需要支持到更大的文件系统，因为我们是采用内存保存元数据的方式，所以我们可以很简单，可靠，高效，灵活的通过增加 master 机器的内存就可以了。

2.6.2 chunk 的位置

master 并不持久化保存 chunkserver 上保存的 chunk 的记录。它只是在启动的时候简单的从 chunkserver 取得这些信息。master 可以在启动之后一直保持自己的这些信息是最新的，因为它控制所有的 chunk 的位置，并且使用普通*心跳信息*监视 chunkserver 的状态。
我们最开始尝试想把 chunk 位置信息持久化保存在 master 上，但是我们后来发现如果再启动时候，以及定期性从 chunkserver 上读取 chunk 位置信息会使得设计简化很多。因为这样可以消除 master 和 chunkserver 之间进行 chunk 信息的同步问题，当 chunkserver 加入和离开集群，更改名字，失效，重新启动等等时候，如果 master 上要求保存 chunk 信息，那么就会存在信息同步的问题。在一个数百台机器的组成的集群中，这样的发生 chunserver 的变动实在是太平常了。
此外，不在 master 上保存 chunk 位置信息的一个重要原因是因为只有 chunkserver 对于 chunk 到底在不在自己机器上有着最后的话语权。另外，在 master 上保存这个信息也是没有必要的，因为有很多原因可以导致 chunserver 可能忽然就丢失了这个 chunk（比如磁盘坏掉了等等），或者 chunkserver 忽然改了名字，那么 master 上保存这个资料啥用处也没有。

2.6.3 操作记录（operation log）

操作记录保存了关键的元数据变化历史记录。它是 GFS 的核心。不仅仅因为这时唯一持久化的元数据记录，而且也是因为操作记录也是作为逻辑时间基线，定义了并行操作的顺序。chunks 以及文件，连同他们的版本（参见 4.5 节），都是用他们创建时刻的逻辑时间基线来作为唯一的并且永远唯一的标志。
由于操作记录是极关键的，我们必须可靠保存之，在元数据改变并且持久化之前，对于客户端来说都是不可见的（也就是说保证原子性）。否则，就算是 chunkserver 完好的情况下，我们也可能会丢失整个文件系统，或者最近的客户端操作。因此，我们把这个文件保存在多个不同的主机上，并且只有当刷新这个相关的操作记录到本地和远程磁盘之后，才会给客户端操作应答。master 可以每次刷新一批日志记录，以减少刷新和复制这个日志导致的系统吞吐量。
master 通过自己的操作记录进行自身文件系统状态的反演。为了减少启动时间，我们必须尽量减少操作日志的大小。master 在日志增长超过某一个大小的时候，执行 checkpoint 动作，卸出自己的状态，这样可以使下次启动的时候从本地硬盘读出这个最新的 checkpoint，然后反演有限记录数。checkpoint 是一个类似 B-树的格式，可以直接映射到内存，而不需要额外的分析。这更进一步加快了恢复的速度，提高了可用性。
因为建立一个 checkpoint 可能会花一点时间，于是我们这样设定 master 的内部状态，就是说新建立的 checkpoint 可以不阻塞新的状态变化。master 切换到一个新的 log 文件，并且在一个独立的线程中创建新的 checkpoint。新的 checkpoint 包含了在切换到新 log 文件之前的状态变化。当这个集群有数百万文件的时候，创建新的 checkpoint 会花上几分钟的时间。当 checkpoint 建立完毕，会写到本地和远程的磁盘。
对于 master 的恢复，只需要最新的 checkpoint 以及后续的 log 文件。旧的 checkpoint 及其 log 文件可以删掉了，虽然我们还是保存几个 checkpoint 以及 log，用来防止比较大的故障产生。在 checkpoint 的时候得故障并不会导致正确性受到影响，因为恢复的代码会检查并且跳过不完整的 checkpoint。

2.7 一致性模型

GFS 是一个松散的一致性检查的模型，通过简单高效的实现，来支持我们的高度分布式计算的应用。我们在这里讨论的 GFS 的可靠性以及对应用的可靠性。我们也强调了 GFS 如何达到这些可靠性，实现细节在本论文的其他部分实现。

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

Table 1: File Region State After Mutation

2.7.1 GFS 的可靠性保证

文件名字空间的改变（比如，文件的创建）是原子操作。他们是由 master 来专门处理的。名字空间的锁定保证了操作的原子性以及正确性（4.1 节）；master 的操作日志定义了这些操作的全局顺序（2.6.3）
什么是文件区，文件区就是在文件中的一小块内容。
不管数据变化成功还是失败，是否是并发的数据变化，一个数据变化导致的一个文件区的状态依赖于这

个变化的类型。表一列出了这些结果。当所有的客户端都看到的是相同的数据的时候，并且与这些客户端从哪个数据的副本读取无关的时候，一个文件区是***一致性***的。一个文件区是***确定的***，当数据发生变化了，在一致性的基础上，客户端将会看到这个全部的变化。当一个更改操作成功完成，没有并发写冲突，那么受影响的区就是***确定的***了（并且潜在一致性）：所有客户端都可以看到这个变化是什么。并发成功操作使得文件区是***不确定的***，但是是***一致性***的：所有客户端都看到了相同的数据，但是并不能却分到底什么变化发生了。通常，他是由好多个变动混合片断组成。一个失败的改变使得一个文件区***不一致***（因此也***不确定***）：不同的用户可能在不同时间看到不同的数据。我们接下来会描述我们的应用如何能够辨别***确定的***和***不确定的***区块。应用程序并不需要进一步区分不同种类的***不确定***区。

数据更改可能是写一个记录或者是一个记录增加（*writes/record appends*）。写操作会导致一个应用指定的文件位置的数据写入动作。记录增加会导致数据（记录）增加，这个增加即使是在并发操作中也*至少是一个原子操作*，但是在并发 record append 中，GFS 选择一个偏移量（3.3）增加。（与之对应的是，一个”普通”增加操作是类似一个客户端相信是写到当前文件最底部的一个操作）。我们把偏移量返回给客户端，并且标志包含这个纪录的***确定的***区域的开始。另外，GFS 可以在这些记录之间增加填充，或者仅仅是记录的重复。这些***确定***区间之间的填充或者记录的重复是不一致的，并且通常是因为用户记录数据比较小造成的。

在一系列成功的改动之后，改动后的文件区是确保**确定的**，并且包含了最后一个改动所写入的数据。GFS 通过（a）对所有的数据副本，按照相同顺序对 chunk 进行提交数据的改动来保证这样的一致性（3.1 节），并且（b）采用 chunk 的版本号码控制，来检查是否有过期的 chunk 改动，这种通常发生在 chunkserver 宕机的情况下（4.5 节）。过期的副本将不参加到改动或者提交给 master，让 master 通知客户端这个副本 chunk 的位置。他们属于最早需要回收的垃圾 chunk。

另外，由于客户端会 cache 这个 chunk 的位置，他们可能会在信息刷新之前读到这个过期的数据副本。这个故障潜在发生的区间受到 chunk 位置 cache 的有效期限制，并且也受到下次重新打开文件的限制，重新打开文件会把这个文件所有的 chunk 相关的 cache 信息全部丢弃重新设置。此外，由于多数文件都是只是追加数据，过期的数据副本通常返回一个较早的 chunk 尾部（也就是说这种模式下，过期的 chunk 返回的仅仅是说，这个 chunk 它以为是最后一个 chunk，其实不是），而不是说返回一个过期的数据。当一个热 ader 尝试和 master 联系，它会立刻得到最新的 chunk 位置。

在一个成功的数据更改之后，并且过了一段相对较长的时间，元器件的实效当然可以导致数据的损毁。GFS 通过 master 和 chunkserver 的普通握手来标记这些 chunserver 的损坏情况，并且使用 checksum 来检查数据是否损坏（5.2 节）。当发现问题的时候，数据会从一个有效的副本立刻重新恢复过来（4.3 节）。只有当 GFS 不能在几分钟内对于这样的损坏做出响应，并且在这几分钟内全部的副本都失效了，这样的情况下数据才会永远的丢失。就算这种情况下，数据 chunk 也是不可用，而不是损坏：这样是给应用程序一个明确的错误提示，而不是给应用程序一个损坏的数据。

2.7.2 应用的实现。

GFS 的应用程序可以用简单的几个技术来实现相关的一致性，这些技术已经被其他目的而使用了：尽量采用追加方式而不是更改方式，checkpoint，写自效验，自标示记录等等。

实际上几乎我们所有的应用程序都是通过追加方式而不是覆盖方式进行数据的操作。通常都是一个程序创建一个文件，从头写到尾。当所有的数据都写完的时候，才把文件名字更改成为正式的文件名，或者定期 checkpoint 有多少数据已经完成写入了。Checkpoint 可以包括应用级别的 checksum。读取程序只校验和处理包含在最近 checkpoint 内的文件区，这些文件区是确定的状态。不管在一致性方面还是并发的方面，这个已经足够满足我们的应用了。追加方式对于应用程序来说更加有效，并且相对随机写操作来说对应用程序来说更加可靠。Checkpoint 使得写操作者增量的进行写操作并且防止读操作者处理已经成功写入，但是对于应用程序角度看来并未提交的数据。

在另一种常见情况下，很多个写操作者对一个文件并发增加，用来合并结果数据，或者提供一个生产者-消费者的队列。增加记录的 至少增加一次 的机制保护了每一个写入者的输出。读取者需要处理这些非必然的空白填充以及记录的重复。写入者写入的每一个记录都包含额外的信息，比如 checksum 等等，这样可以使得每条记录都能够效验。读取者可以通过这些 checksum 辨别和扔掉额外的填充记录或者记录碎片。如果读取者不能处理这些偶然的重复记录（比如，如果他们触发了一种非等幂操作等等 non-idempotent operations）,他可以通过记录的唯一标志来区分出记录，这些唯一标志常常用来标记相关的应用实体，比如 web 文档等等。这些记录 I/O 的功能（除了移出复制记录），都是放在函数库中的，用于我们的应用程序，并且可应用于 google 里边的其它的文件接口实现。通过这些函数库，相同序列的记录，和一些重复填充，就可以提供给记录的读取者了。

3 系统交互

我们设计的一个原则是尽量在所有操作中减少 master 的交互。在这个基础上，我们现在讨论客户

端，master，chunkserver 如何交互，实现数据的变动，原子的记录增加，以及快照。

3.1 令牌[1]和变化顺序

变动操作是一种改变 chunk 内容或者 chunk 的原数据的操作，比如改写或者增加操作。每一个变动操作都要对所有的 chunk 的副本进行操作。我们用租约的方式来管理在不同副本中的一致的更改顺序。master 首先为副本中的一个 chunk 分配一个令牌，这个副本就是 primary 副本。这个 primary 对所有对 chunk 更改进行序列化。所有的副本都需要根据这个 primary 的序列进行更改。这样，全局的更改顺序就是首先由 master 分配的 chunk 令牌顺序决定的，并且 primary 决定更改的序列。

令牌机制设计用来最小化 master 的管理负载。一个令牌初始化的有效期是 60 秒。不过，随着 chunk 的更改操作的进行，primary 可以请求延期并且一般情况下都会收到 master 的批准。这些延期请求并且批准延期都是通过在 master 和所有 chunkserver 之间的 HeartBeat 心跳消息来承载的。master 有可能会在令牌超时前收回令牌（比如，master 可能会终止正在改名的文件上的修改操作等等）。即使 master 和 primary 丢失了联系，master 也可以很安全的在原始令牌超时后授予另外一个副本一个令牌。

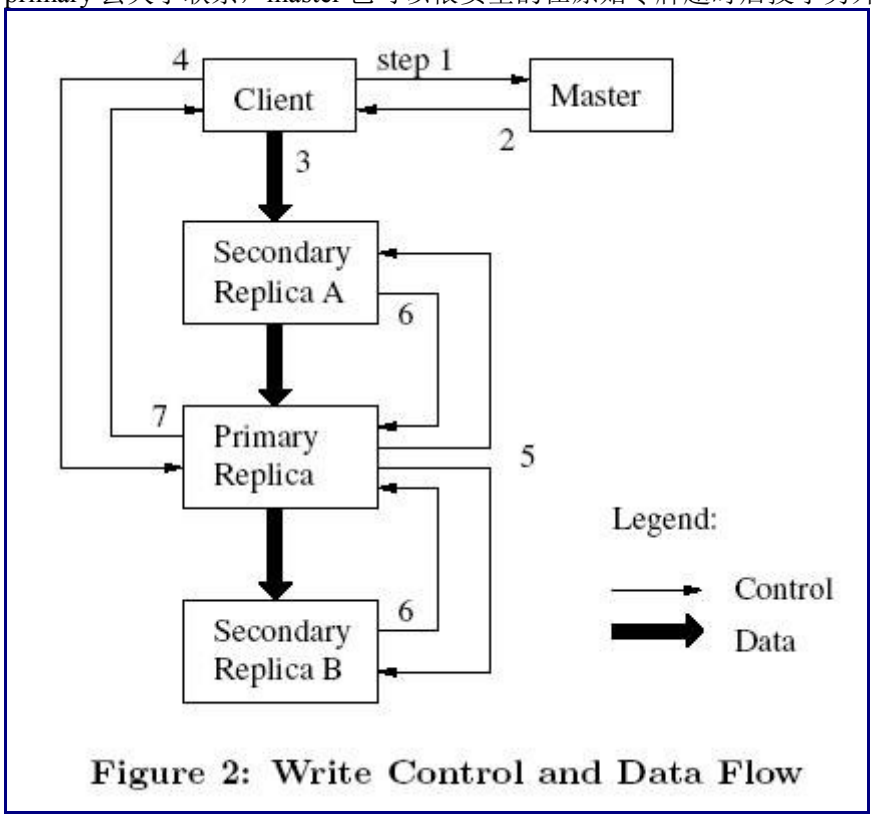

Figure 2: Write Control and Data Flow

图 2 中，我们展示了这个更改的控制流过程：

1. 客户端向 master 请求当前 chunk 的令牌位置以及其他所有副本的位置。如果没有 chunkserver 持有这个 chunk 的令牌，则 master 选择一个 chunk 副本授权一个令牌（在图上没有标出）
2. master 给出应答，包括了 primary 和其他副本位置（secondary）标记。客户端 cache 这些数据，用于以后的变动。只有当 primary 不能访问或者 primary 返回它不再持有令牌的时候，客户端才需要重新联系 master。
3. 客户端把数据发布给每一个副本。客户端可以以任意顺序发布这些数据。每一个 chunkserver 都在内部的 LRU 缓冲中 cache 这些数据，这些数据一旦被提交或者过期就会从缓冲中去掉。通过把数据流和控制流的分离，我们可以不考虑哪个 chunkserver 是 primary，通过仔细调度基于网络传输的代价昂贵的数据流，优化整体的性能。3.2 节进一步讨论了这个。
4. 当所有的副本都确认收到了数据，客户端发起一个写请求给 primary。这个请求标记了早先发给所有副本的数据。primary 分配一系列连续的序列号给所有的收到的变动请求，这个可能是从好多客户端收到的，这提供了必要的序列化。primary 按照这个序列号顺序变动他自身本地的状态。
5. prmary 把写请求发布到所有的 secondary 副本。每一个 secondary 副本都依照和 primary 分配的相同的序列号顺序来进行变动的提交。
6. secondary 副本全部都给 primary 应答，表示他们都已经完成了这个操作。
7. primary 应答给客户端。如果有任何副本报告了任何错误，都需要报告给客户端。在发生错的情况下，写入者会在 primary 成功但是在 secondary 副本的某些机器上失败。（如果在 primary 失败，

不会产生一个写入的序列号并且发布序列号）。客户端请求就是由失败的情况，并且修改的区域就有不一致的状态。我们的客户端代码是通过重试改动来处理这些错误。他可能会在从头开始重试前，在第三步到第 7 步尝试好几次。

如果应用的一个写入操作不止一个 chunk 或者是跨 chunk 的操作。GFS 客户端代码把这个写入操作分解成为多个写入操作。每一个写操作都按照上边描述的控制流进行的，这些写操作可能和其他客户端的操作并发进行交叉存取和改写。因此，虽然因为每个操作都是对每个副本相同顺序完成的，对每一个副本都是一致的，但是大家共享操作的文件区块可能最后会包含不同客户端的小块。这就使得文件区虽然一致，但是是不确定的（如同 2.7 节讲述的一样）。

3.2 数据流

我们把数据流和控制流分开，是为了更有效的利用网络资源。当控制流从客户端到 primary，记者到所有的 secondary 副本，数据是通过一个精心选择的 chunkserver 链，在某种程度上像是管道线一样线形推送的。我们的目的是完全利用每一个机器的网络带宽，避免网络瓶颈以及高延时的连接，最小化同步数据的时间。

为了挖掘每一个机器的网络带宽，数据是依据一个 chunkserver 链路进行线形推送的，而不是根据其他的拓扑结构推送的（比如树形）。因此，每一个机器的全部输出带宽都是用于尽可能快地传送数据，而不是在多个接收者之间进行分配。

为了尽可能避免网络的瓶颈和高延时连接（比如 inter-switch 连接通常既是瓶颈延时也高），每一个机器都是把数据发送给在网络拓扑图上”最近”的尚未收到数据的机器。假设客户端把数据从 chunkserver S1 发送到 S4。他首先发送给最近的 chunkserver，假设是 S1。S1 把数据发送给 S2 到 S4 内的最近 chunkserver，假设是 S2。类似的，S2 发送给 S3 或者 S4，看谁更近，以此类推。我们的网络拓扑图是很简单的，所以，”距离”可以直接根据 IP 地址进行推算。

最后，我们通过流水线操作基于 TCP 连接数据传输，来最大限度的减少延时。当一个 chunkserver 接收到一些数据，它就立刻开始转发。因为我们用的是全双工交换网络，所以流水线对于我们特别有用。立刻发送数据并不会降低接收数据的速率。抛开网络阻塞，传输 B 个字节到 R 个副本的理想时间是 B/T+RL，T 是网络吞吐量，L 是两点之间的延时。我们网络连接时 100M（T），L 通常小于 1 毫秒。因此，1M 通常理想情况下发布时间小于 80ms。

3.3 原子纪录增加

GFS 提供了原子增加操作，叫做 record append。在传统写操作中，客户端给定写入数据的偏移量。对同一个区域的并发写操作并没有序列化；这个区域可能会包含多个客户端的分段的数据。在 record append，客户端只是给出数据，GFS 在其指定的一个偏移量上，原子化的保证起码增加一次（也就是说，保证在一个连续的字节序内），并且把这个偏移量返回给客户端。这个很类似当多个写者并发操作的情况下，unix 下没有竞争条件的 O_APPEND 写文件操作。

纪录添加模式大量在我们的应用中是用，在我们的分布式应用情况下，大量的客户端分布在不同的机器上，同时向同一个文件进行追加纪录得操作。客户端需要额外的复杂的代价高昂的同步操作，比如如果按照传统的写操作，就基于一个分布的锁管理。在我们的工作量下，这样的文件经常需要为多生产者/单消费者的队列或者包含从多个客户端合并结果集的操作。

纪录增加也属于一种改动操作，并且遵循 3.1 描述的控制流，它在 primary 上只有一点额外的逻辑操作。客户端把数据分布给文件最后一个 chunk 所在的每个副本。于是，他把请求发送给 primary。primary 检查看看是否这些对当前 chunk 的增加是否会导致 chunk 超过最大大小（64M）。如果超过了，它就把 chunk 填写到最大大小，告诉 secondary 也跟着填写到最大，并且返回给客户端表示这个操作需要在下一个 chunk 重试。（纪录增加操作严格限制在 1/4 最大 chunk 大小，来保证最坏的分段情况下还是可以接受的）。如果纪录可以在最大大小内存放，这也是常见的情况，primary 就增加这个纪录到它自己的副本，告诉其他 secondary 副本也在相同的偏移量开始写，并且最终返回成功给客户端。

如果任意一个副本报告纪录增加失败，客户端就重新尝试这个操作。因此，副本中，对于相同 chunk 可能包含不同的数据，这些不同数据包括了相同纪录的完整或者部分的重复纪录。GFS 并不保证所有的副本都是 byte 级别相等的。它只保证数据时基于原子级别至少写入一次。这个特性是从对操作开始到成功完成的一个简单研究得出的，数据必须写在所有副本的相同 chunk 的相同偏移量写入。此外，所有的副本都必须起码和纪录结束点等长，并且因此即使另外一个副本成了 primary，所有后续的纪录都会被分配在一个较高的偏移量或者在另外一个 chunk 中。在我们一致性的保证中，成功追加纪录操作写入他们的数据的区域是确定的（因此也是一致的），但是追加纪录与纪录之间的区域却是不一致的（因此也使不确定的）。我们的应用可以处理这些不一致的区域，我们在 2.7.2 中有所讨论。

3.4 快照

快照操作在尽量不影响正在发生的变动的情况下，几乎即时产生一个文件或者目录树（”源”）。我们

的用户是用快照来迅速创建举行数据集的一个分支拷贝（经常还有拷贝的拷贝，递归拷贝），或者在提交变动前做一个当前状态的 checkpoint，这样可以使得接下来的 commit 或者回滚容易一点。
如同 AFS[5],我们用标准的 copy-on-write(写时拷贝)的技术来实现快照。当 master 收到一个快照请求，他首先收回所有的相关快照文件中发布出去的 chunk 令牌。这确保了后续的对 chunk 的写入都会产生一个和 master 的交互来寻找令牌持有者。这使得 master 可以在产生写入操作的时候先产生一个 chunk 的副本。
当令牌收回或者已经过期以后，master 把这个操作纪录到磁盘。接着他把这个 log 纪录通过复制源文件或者目录树的元数据的方式，提交到内存状态中。新创建的快照文件指向和源文件相同的 chunk。
等客户端在快照操作后，首次吸入一个 chunk C 的时候，他发送请求给 master 来寻找当前的令牌持有者。master 发现这个 chunk C 的引用次数超过 1。它就推迟应答给客户端，并且找一个新的 chunk 来处理 C’。接着向每一个包含当前 C 副本的 chunk 服务器创建一个新 chunk C’。通过在和原始 chunk 有相同的 chunkserver 上新的 chunk，我们确保数据可以进行本地拷贝，而不是通过网络（本地硬盘速度大概是 100M 网络连接速度的 3 倍）。从这点开始，对于请求的后续处理就和处理其他 chunk 没有什么不同了，master 分配一个令牌给新的 chunk C’，并且回答给客户端，这样客户端可以正常的写操作，不需要知道 chunk 是刚从现存的 chunk 上创建的。

4 master 操作

master 执行所有的 namespace 的操作。另外，他管理系统中所有 chunk 的副本；他决定 chunk 的放置策略，穿件新的 chunk 及其副本，以及相关的多个系统级别的操作来保持 chunk 有好几个副本，平衡所有 chunkserver 之间的附在，回收未使用的存储。我们现在讨论每一个小节。

4.1 namespace 管理及锁定

很多 master 的操作都可能执行比较长的时间；比如，一个快照操作可能要回收快照所覆盖的所有 chunk 所在的 chunkserver 的令牌等等。我们不希望这个操作执行的同时，阻碍其他 master 的操作。因此，我们允许多个操作同时进行，并且使用基于 namespace 的区域的锁来保证必须要得序列化。
不同于很多传统的文件系统，GFS 没有一个 per-directory 数据结果列出了所有在此目录下的文件，也不支持对同一个文件或者目录的别名（比如，unix 系统下的硬连接或者符号连接）。GFS 从逻辑上是通过一个查找路径名到元数据映射表的方式来体现 namespace 的。通过前缀压缩，这个表可以有效地在内存中存放。每一个 namespace 树种的节点（不论是绝对文件名还是绝对目录名），都有一个相关的读写锁。
每一个 master 操作在执行前都要求一组锁的集合。通常，如果它包含/d1/d2/…/dn/leaf，它会要求在/d1,/d1/d2,…/d1/d2/…/dn 上的读锁，并且读锁以及写锁在全文件名的/d1/d2/…/dn/leaf。注意，leaf 可以是这个操作相关的一个文件或者目录名。
我们现在通过一个例子来讲解这个锁机制如何有效工作的。假定我们在创建/home/user 的快照/save/user 的时候，如何防止/home/user/foo 文件的创建。这个快照的操作要求读锁：/home 以及/save，写锁/home/user 和/save/user。文件创建操作要求读锁/home 和/home/user，写锁/home/user/foo。这两个操作就会被正确序列化，因为他们尝试解决锁冲突/home/user。文件创建并不要求一个在父目录的写锁，因为这并没有一个需要保护的”目录”或者 inode-like 的数据结构。在名字上的读锁已经足够来保护父目录不被删除。
这种锁机制带来一个好处就是在同一个目录下允许并发改动。比如，在同一个目录下的多个文件创建可以并行执行；每一个要求一个在目录名上的读锁，并且要求在一个文件名上的写锁。在目录名上的读锁足以防止目录被删除，改名或者快照。在文件名上的写锁序列化对两次同一文件的创建操作。
因为 namespace 可以有很多的节点，读写锁对象是滞后分配以及一旦没有使用就立刻删除的。并且，锁是基于一个相同的总顺序来分配的，这样放置死锁：他们是首先根据 namespace 树种的级别顺序，以及相同级别下根据字典顺序进行分配的。

4.2 副本位置

GFS 集群是在不止一个级别上的多级别的高度分布的系统。通常由好几百台分布在很多机架上的 chunkserver 组成。这些 chunkserver 可能被相同或者不同的机架的几百台客户端轮流访问。两个机架上的两台机器可能会跨越不止一个网络交换机。此外，机架的入口带宽或者出口带宽往往会比在机架内的机器聚合带宽要小。多级别的分布凸现了一个独特的要求，为了可扩展性，可靠性，以及可用性要把数据进行分布。
chunk 复本存放机制有两个目的：最大限度的保证数据的可靠和可用，并且最大限度的利用网络带宽。对于两者来说，仅仅在机器之间进行复本的复制时不足够的，它只保证了磁盘或者机器的实效，以及每一个机器的网络带宽的使用。我们必须也在机架之间进行 chunk 的复制。这就保证了当整个机架都损坏的时候（或者掉线的时候），对于任意一个 chunk 来说，都有一些副本依旧有效（比如，由于共享资源的失效，例如网络交换机或者电源故障，导致机架不可用）。这也意味着网络冲突的减轻，尤其是对于读取操作来说，对于一个 chunk 的读取来说，使用的是每一个机架内部的聚合带宽。换句话说，会增加跨越各个机架的写流量，这个是我们相比较愿意承受的代价。

4.3 创建，重新复制，重新均衡

有三个原因需要创建 chunk 的副本：chunk 的创建，chunk 的重新复制，chunk 的重新均衡。

当 master 创建了一个 chunk，它会选择放置初始化空白副本的位置。它会考虑几个因素：(1)我们希望新副本所在的 chunkserver 有着低于平均水平的磁盘空间利用率。随着时间的推进，chunkserver 上的磁盘利用率会趋于均匀。(2)我们希望限制每一个 chunkserver 上的”最近”创建的数量。虽然创建操作本身的负载很轻，但是它却意味着随后立刻又很重的写操作，因为 chunk 是因为要写东西才会创建，并且在我们的写一次读多次的工作量下，他们通常完成写操作以后，他们实际上就成为只读的了。(3)如同上边讨论得这样，我们希望把副本跨越机架。

当 chunk 的副本数量小于一个用户指定的数量后，master 会立刻尝试重新复制一个 chunk 副本。这有可能由好几种原因引起：比如 chunkserver 失效，或者 chunkserver 报告自己的副本损坏了，或者它的某一个硬盘故障，或者增加了副本数量等等。每一个需要重新复制的 chunk 于是根据几个因素进行优先级分布。一个是与副本指定数量差距决定优先级。例如，我们对于一个差了两个副本的 chunk 给定一个高优先级，而给一个只差一个副本的 chunk 一个较低的优先级。此外，我们倾向于首先重新复制活跃文件的 chunk，而不是复制刚刚被删掉的文件的副本（参见 4.4）。最后，为了减少失效的 chunk 对正在运行应用的影响，我们提高妨碍客户端进程的 chunk 副本的优先级。

master 选择最高优先级的 chunk 并且通过通知一些 chunkserver 从现有副本上直接复制这个 chunk 数据的方式来”克隆”这个 chunk。新的副本是用和他们创建时相同的策略来选择存放位置的：均衡磁盘利用率，在单个 chunkserver 上限制活跃克隆操作，在机架间进行副本的分布。

为了防止克隆导致的带宽消耗大于客户端的带宽消耗，master 限制集群上和每个 chunkserver 上的活跃的克隆操作。此外，每个 chunkserver 通过限制对源 chunkserver 的克隆读取请求来限制克隆操作的带宽开销。最后，master 定期重新进行均衡副本：他检查当前的副本分布情况，并且把副本调整到更好的磁盘和负载分布。并且通过这个步骤，master 逐渐渗透使用一个新的 chunkserver，而不是立刻大量分布新 chunk 给新的 chunkserver 从而导致新 chunkserver 过载。选定副本分布的策略和上边讨论的策略类似。此外，master 必须决定哪个副本需要移动。总的来说，它会倾向于移动分布在低于平均磁盘剩余空间 chunkserver 上的 chunk，这样会平衡磁盘空间使用。

4.4 垃圾回收

当文件被删除以后，GFS 并不立刻要求归还可用的物理存储。它是通过滞后的基于文件和 chunk 级别的普通垃圾回收机制来完成的。我们发现这样可以使得系统更加简单和可靠。

4.4.1 机制

当应用删除了一个文件，master 像记录其他变动一样，立刻记录这个删除操作。不过不同于立刻回收资源，这个文件仅仅是改名称为一个隐藏的名字，并且包含了一个删除时戳。在 master 的常规的文件系统 namespace 的检查中，他会移出这些超过 3 天隐藏删除文件（这个 3 天是可以配置的）。直到此时，文件依旧可以通过新的，特别的名字读取，并且可以通过改名来恢复成为未删除状态。当隐藏文件从 namespace 删除后，它在内存的元数据也随之删除。这个会影响他所指向的各个 chunk。

在对 chunk 的 namespace 的常规扫描中，master 识别 chunk 孤点（就是说，没有被任何文件所指向）并且删除这些孤点的元数据。在 chunkserver 和 master 的常规心跳消息中，每一个 chunkserver 都报告自己的 chunk 集合，并且 master 回复在 master 的元数据中已经不存在的 chunk 标记。chunkserver 随即释放和删除这些 chunk 的副本。

4.4.2 讨论

虽然分布式的垃圾回收是一个艰巨的问题，在程序设计的时候需要复杂的解决，但是在我们的系统中却是比较简单的。我们可以轻易辨别出对一个 chunk 的全部引用：它们都唯一保存在 master 的文件-chunk 影射表中。我们也可以容易辨别所有的 chunk 副本：它们是在各个 chunkserver 上的指定目录下的 linux 文件。所有不被 master 知道的副本就是”垃圾”。

垃圾回收比较类似提供几种便利删除机制的存储回收。首先，他在一个由非可靠节点组成的超大分布式系统中可靠而简单的运行。chunk 的创建在某些 chunkserver 可以成功但是在某些 chunkserver 却会失败，这样导致 master 不知道的某些副本。副本删除消息可能会丢失，master 被迫记住并且重发这些失败的副本删除消息，不仅仅是它自己的副本删除消息，也包含 chunkserver 的。垃圾回收机制提供了一个统一的可靠的方式来清除未知的副本，这种机制非常有用。其次，他把存储的回收合并到常规的 master 后台操作，如同常规的对 namespace 的扫描，以及对 chunkserver 的握手。因此，它是作为批处理的，处理开销也是分批地。另外，这仅仅是当 master 的负载相对较轻的时候进行的。master 可以更快的相应客户的请求，不过这需要花时间来关注客户的请求（所以对垃圾的回收时在负载较轻的情况下执行的）。第三，在存储回收的延后回收也提供了某种程度的由于网络故障导致的不可回收的删除的恢复。

在我们的经验中，主要的缺点就是当存储紧张的时候，滞后删除会导致阻碍我们尝试调整磁盘使用情况

的效果。应用程序重复创建和删除临时文件可能不会正确重用存储。我们通过如果一个已经删除了的文件再次删除的操作，来加速存储回收，部分解决这个问题。我们也允许用户对于不同的 namespace 部分，使用不同的复制和回收策略。例如，用户可以指定所有的在某目录树下的文件 chunk 的保存没有副本，任何删除的文件立刻并且不可撤销的从文件系统状态中删除。

4.5 过期副本删除

chunk 副本可能会因为 chunkserver 失效期间丢失了对 chunk 的变动而导致过期。对于每一个 chunk，master 保持一个 chunk 的版本号码来区分最新的和过期的副本。

无论何时 master 为一个 chunk 颁发一个令牌，他都增加 chunk 的版本号码并且通知最新的副本。master 和这些副本在他们的持久化状态中都记录最新的版本号码。这在任何客户端被通知前发生，并且因此在开始对这个 chunk 写之前发生。如果另一个副本当前不在线，那么他的 chunk 的版本号码就不会改变。当这个 chunkserver 重新启动，并且向 master 报告它的 chunk 以及相关版本号码的时候，master 会根据版本号码来检查出这个 chunkserver 有一个过期的副本。如果 master 发现一个更高版本号码的 chunk，master 会认为他在颁布令牌的时候失败了，于是会取较高的版本号来作为最新的 chunk。

master 通过正常的垃圾回收机制来删除过期的副本。在删除之前，它需要确认在它给所有客户端的 chunk 信息请求的应答中，都没有包含这个过期的副本。作为另外一个安全控制，master 采用了 chunk 版本号，当它告诉客户端那个 chunkserver 有这个 chunk 的令牌，或者指使 chunkserver 从另一个 chunkserver 去克隆一个 chunk 的副本的时候，都会采用版本号来控制。客户端或者 chunkserver 会在执行操作前检查版本号，确保操作的数据都是最新的数据。

5 容错和故障诊断

我们遇到的最大难题就是设计的系统会有经常性的节点故障。节点数量和质量导致了这个问题还不是意外情况：我们不能完全相信机器，也不能相信磁盘。节点失效可以使得系统不可用，或者更糟糕的是会损坏数据。这里我们讨论我们怎样解决这样的难题，以及我们构建在系统内部的工具用来侦测系统不可避免的会存在的问题。

5.1 高可用性

在一个 GFS 集群里，会有好几百台服务器，在任何时候都可能会有机器不可用。我们用两条很简单有效的策略来保证整体上的系统高可用性：迅速的恢复机制以及副本机制。

5.1.1 快速恢复机制

master 和 chunkserver 都设计成为本地保存状态，并且无论他们正常或者异常退出都可以在几秒钟之内启动。实际上，我们不用讨论正常和异常退出；服务器通常是直接杀掉进程来关机的。客户端和其他服务器在他们正在发起的请求上会获得一个超时，感觉一点颠簸，他们会重新尝试连接这个重新启动的服务器，并且重试这个请求。6.2.2 节讲述了启动过程。

5.1.2 chunk 副本机制

如同早先讨论过的，每一个 chunk 都在不同的机架上的不同 chunkserver 上有副本。用户可以对不同的文件 namespace 指定不同的复制级别。缺省是 3 个。master 根据需要克隆现有的副本，并且维持当 chunkserver 掉线的时候，每一个 chunk 都有完善的副本，以及 master 通过 checksum 检查来删除损坏的副本（5.2 节）。虽然复制机制对我们来说已经很有效了，我们依旧在探索其他的跨机器的冗余机制，比如对于我们日益增长的只读存储需要，设计奇偶或者其他标记代码。因为我们的网络负载受控于增加性质的修改以及大量的读取，很少有随机的写，所以我们可以预期我们可以克服挑战，在我们的很松散的系统上，设计和实现更复杂的冗余机制。

5.1.3 master 的复制

master 的状态基于可靠性的理由也要做复制。master 的所有操作都是基于 log 和 checkpoint 的，这些 log 和 checkpoint 将被复制到多个机器上。一个对 master 状态的修改操作只有当所有远程 master 副本提交成功，并且也提交到了本地磁盘的时候，才认为是已经提交了。简单说来，一个 master 的进程负责所有的改动，包括后台的服务，比如垃圾回收等等内部活动。当它失效的时候，几乎可以立刻启动。如果机器或者磁盘失效，GFS 外部的监控机制在别的地方包括操作副本 log 的机器上启动一个新的 master 进程。客户端只用规范的名字来访问 master（比如 gfs-test）,这个市一个 DNS 的别名，可以由于 master 改动到别的机器上执行而更改实际地点。

进一步说，master 的”影子进程”，提供了对文件系统的只读操作，即使当当前的 master 失效的时候也是只读的。他们是影子进程，并非镜像进程，他们可能会比 primary master 稍慢一拍，通常是不到一秒钟。这些进程增强了对于那些并不是很活跃修改的文件的读取能力，或者对那些读取脏数据也无所谓的应用来说，提高了读取性能。实际上，因为文件内容是从 chunkserver 上读取的，应永不不能发现文件内容的过期。什么原因会导致在一个很小的时间窗内文件的元数据会过期呢，目录内容或者访问控制信息的变动会导致小时间窗内元数据过期。

为了保证 master 的影子进程能维持最新状态，它从当前的操作 log 的副本中读取，并且根据与 primary 完全相同顺序来更改内部的数据结构。如同 primary 一样，他在启动的时候从 chunkserver 拉数据（并且启动以后定期拉），这些数据包括了 chunk 的副本位置信息，并且也会和 chunkserver 进行握手来确定他们的状态。它从 primary master 上只处理因为 primary 决定创建或者删除副本导致的副本位置更新结果。

5.2 数据完整性

每一个 chunkserver 都是用 checksum 来检查保存数据的完整性。通常一个 GFS 集群都有好几百台机器以及几千块硬盘，磁盘损坏是很经常的事情，在数据的读写中经常出现数据损坏（7 节讲了一种原因）。我们可以通过别的 chunk 副本来解决这个问题，但是如果跨越 chunkserver 比较这个 chunk 的内容来决定是否损坏就很不实际。进一步说，允许不同副本的存在;在 GFS 更改操作的语义上，特别是早先讨论过的原子纪录增加的情况下，并不保证 byte 级别的副本相同。因此，每一个 chunkserver 上必须独立的效验自己的副本的完整性，并且自己管理 checksum。

我们把一个 chunk 分成 64k 的块。每一个都有相对应的 32 位的 checksum。就像其他的元数据一样，checksum 是在内存中保存的，并且通过分别记录用户数据来持久化保存。

对于读操作来说，在给客户端或者 chunkserver 读者返回数据之前，chunkserver 效验要被读取的数据所在块的 checksum。因此 chunkserver 不会把错误带给其他设备。如果一个块的 checksum 不正确，chunkserver 会给请求者一个错误，并且告知 master 这个错误。收到这个应答之后，请求者应当从其他副本读取这个数据，master 也会安排从其他副本来做克隆。当一个新的副本就绪后，master 会指挥刚才报错的 chunkserver 删掉它刚才错误的副本。

checksum 对于读取性能来说，在几方面有点影响。因为大部分的读取操作都分布在好几个 block 上，我们只需要额外的多读取一小部分相关数据进行 checksum 检查。GFS 客户端代码通过每次把读取操作都对齐在 block 边界上，来进一步减少了这些额外的读取。此外，在 chunkserver 上的 chunksum 的查找和比较不需要附加的 I/O，checksum 的计算可以和 I/O 操作同时进行。

checksum 的计算是针对添加到 chunk 尾部的写入操作作了高强度的优化（和改写现有数据不同），因为它们显然占据主要工作任务。我们增量更新关于最后 block 的 checksum 部分，并且计算添加操作导致的新的 checksum block 部分。即使是某一个 checksum 块已经损坏了，但是我们在写得时候并不立刻检查，新的 checksum 值也不会和已有数据吻合，下次对这个 block 的读取的时候，会检查出这个损坏的 block。另一方面，如果写操作基于一个已有的 chunk（而不是在最后追加），我们必须读取和效验被写得第一个和最后一个 block，然后再作写操作，最后计算和写入新的 checksum。如果我们不效验第一个和最后一个被写得 block，那么新的 checksum 可能会隐藏没有改写区域的损坏部分。

在 chunkserver 空闲的时候，他扫描和效验每一个非活动的 chunk 的内容。这使得我们能够检查不常用的 chunk 块的完整性。一旦发现这样的块有损坏，master 可以创建一个新的正确的副本，然后把这个损坏的副本删除。这个机制防止了非活动的块损坏的时候，master 还以为这些非活动块都已经有了足够多的副本。

5.3 诊断工具

详细的扩展诊断日志对于问题的发现和调试解决，以及性能分析来说，有着不可估量的作用，并且记录详细的扩展诊断日志只需要一点小小的开销。如果没有日志，很难理解偶发的，不能重现的机器间的交互作用。GFS 服务器产生诊断日志，记录下很多关键时间（比如 chunkserver 启动和停止等等），以及所有的 RPC 请求和应答。这些诊断日志可以在不影响系统正确性的前提下删除。不过，如果磁盘空间允许，我们希望尽量保留这些日志信息。

RPC 日志包括了在网络上传输的除了被操作的文件数据以外的请求和应答细节。通过匹配和比较不同机器上的 RPC 请求和应答，我们可以重构整个的交互历史，这样可以诊断问题。这些日志同样用于追踪负载测试以及性能分析。

记录日志对于性能的影响来说是比较小的（相比较而言产生的效果却是巨大的），因为日志是顺序的并且是异步的。最新的时间是在内存中保留，并且可以作持续的在线监控。

6 度量

在本节，我们通过几个小型的 bechmark 来演示 GFS 架构和实现上的性能瓶颈，并且展示了一些再 GOOGLE 内使用的真实地集群的性能数据。

6.1 小型 benchmark

我们在一个有一个 master，两个 master 副本，16 个 chunkserver，16 个客户端的 GFS 集群上进行的性能测试。这个配置是用于便于测试使用的。实际上的集群通常有好几百台 chunkserver 和几百台客户端组成。所有的机器都是有双 1.4G PIII 处理器，2GB 内存，两个 80GB 5400 转硬盘，一个 100M 全双工网卡（连接到 HP2524 交换机）组成。全部 19 台 GFS 服务器都是连接到一个交换机上的，16 台客户端连接到另一个交换机。两台交换机之间是用的 1G 链路连接的。

6.1.1 读取

N 个客户端并行从文件系统读取。每一个客户端从 320GB 文件集合中，读取随机选取的 4M 区域。重复读取 256 次，每一个客户端最终读取 1GB 数据。chunkserver 总共有 32GB 内存，这样我们预期有最多 10%的 Linux buffer cache 命中率。我们的结果应当和冷 cache 的结果一致。

图三（a）展示了 Nge 客户端合计读取速度，以及它的理论上限。合计的理论上限是两个交换机之间的 1GB 链路饱和的情况下达到，就是 125MB/s 的速度，或者当客户端的 100M 网卡饱和的情况下的每客户端 12.5MB/s 的速度。当只有一个客户端读取的时候，观测到的读取速度是 10MB/s，或者 80%客户端的限制。16 个客户端的合计读取速度达到了 94MB/s，大约是 75%的 125MB/s 的理论限制。由 80%降低到 75%的原因是由于读取者的增多，导致多个读取者同时从一个相同 chunkserver 读取得可能性增加，导致的读取性能下降。

6.1.2 写入

N 个客户端同时向 N 个不同的文件写入。每一个客户端每次写入 1MB，总共写入 1GB 的数据到一个新的文件。合计写入速度以及它的理论上限在图三(b)中展示。理论限制是 67MB/s，是因为我们需要把每一个字节写入 3 个 chunkserver，每个都有 12.5MB/s 的输入连接。

每一个客户端的写入速度是 6.3MB/s,差不多是一般的限制主要原因是我们的网络协议栈。它和我们使用的推送数据到 chunk 副本的流水线机制的交互并不是很好。再把数据从一个副本传输到另一个副本的延时导致了整个写入速度的降低。16 个客户端的合计写入速度差不多是 35MB/s（或者 2.2MB/s 每客户端），差不多是理论极限的一般。和读取情况比较类似，这样的情况多半发生于多个客户端同步写入同一个 chunkserver 时导致的性能下降。此外，由于 16 个写者要比 16 个读者更容易产生冲突，这是因为每一个写入要写三份副本的原因。

写入速度比我们预期的要慢一点。在实际情况下，这并不是一个大问题，因为即使在单个客户端上能够感受到延时，他也不会对大量客户端的情况下，对整个写入带宽造成明显的影响。

6.1.3 记录增加

图三（c）展示了记录增加的性能。N 各客户端同时对一个文件进行增加记录。性能是受到保存文件的最后一个 chunk 的 chunkserver 的带宽限制，而与客户端的数量无关。他从单个客户端的 6.0MB/s 降低到 16 个客户端的 4.8MB/s,主要是由于不同客户端的网络拥塞以及网络传输速度的不同导致的。

我们的应用多属于同步产生多个这样的文件。换句话说，N 个客户端对 M 个共享文件同步增加，N 和 M 都是数十或者数百。因此，在我们实验中出现的 chunkserver 网络拥塞就在实际的情况下就不是一个问题，因为一个客户端可以在 chunkserver 对另一个文件忙得时候进行写入文件的操作。

6.2 实际的集群

我们现在通过 Google 的两个有代表性的集群来检查一下。集群 A 用于上百个工程师的研发。通常这个集群上的任务都是由人工发起的，运行好几个小时的任务。读取范围从好几 M 到好几 TB 数据，分析和处理数据，并且向集群写回结果。集群 B 是当前的生产数据处理集群。它上面的应用通常需要持续处理，并且产生和处理上 TB 的数据集，很少需要人工干预。在两个集群下，单个“task”意味着包含在很多机器上进行的很多读取和写入多个文件的并发进程。

Cluster	A	B
Chunkservers	342	227
可用硬盘空间 已用硬盘空间	72TB 55TB	180TB 155TB
文件数量 死文件数量 Chunk 数量	735k 22k 992k	737k 232k 1550k
chunkserver 元数据 master 元数据	13GB 48MB	21GB 60MB

表 2：两个 GFS 集群特性

6.2.1 存储

如同上表中描述的，两个集群都有上百台 chunkserver，支持很多 TB 的硬盘空间，几乎不会满。”已用空间”包含了所有的 chunk 副本。一般来说所有的文件都有三个副本。因此，集群实际上各存储了 18TB 和 52TB 的文件数据。

两个集群都有相近的文件数目，虽然集群 B 有着很大的死文件数量，也就是说文件被删掉了或者用新版

本代替了，而集群还没有来得及清理。并且应为文件比较大，集群 B 也有比较多的 chunk 数目。

6.2.2 元数据

chunkserver 一共保存了十几个 GB 的元数据，主要是用户数据的 64KB block 的 checksum 数据。其他在 chunkserver 上保存的元数据是 chunk 的版本号，在 4.5 节有讲述。

在 master 上保存的元数据就小多了，只有数十 MB，或者说平均每个文件 100 字节。这和我们的预期的一样，master 的内存在实际上并不会是系统容量的限制瓶颈。大部分的文件的元数据都是文件名，而且是用的前缀压缩模式保存的。其他的元数据包括了文件的所有者和权限，以及文件到 chunk 的应设关系，以及每一个 chunk 的当前版本号。此外，每一个 chunk 我们都保存当前的副本位置以及对其的引用数字，用于实现写时拷贝[copy-on-write，3.4]。

每一个单独的服务器，chunkserver 或者 master，都只有 50 到 100MB 的元数据。因此恢复时很快：只需要几秒钟时间从磁盘读取这些数据就可以了响应请求了。不过，master 会慢一点-通常 30-60 秒-它需要从所有的 chunkserver 获取当前 chunk 位置信息。

6.2.3 读写速率

表三显示了不同时段的读写速率。两个集群都采样了一周的时间。（集群最近因为升级新版本的 GFS 而重新启动了）。重新启动后，平均写入速率小雨 30MB/s。当我们在采样期间，集群 B 在峰值写入速度达到了 100MB/s，并且因为产生三个副本的原因，它产生了 300MB/s 的网络负载。

图三：合计吞吐量：上边的曲线显示了我们网络拓扑下的合计理论吞吐量上限。下边的曲线显示了观测到的吞吐量。这个曲线有着 95%的可靠性，因为有时候测量会不够精确。

集群	A	B
读速率（最后一分钟） 读取率（最后一小时） 读速率（自从重启动）	583MB/s 562MB/s 589MB/s	380MB/s 384MB/s 49MB/s
写速率（最后一分钟） 写取率（最后一小时） 写速率（自从重启动）	1MB/s 1MB/s 25MB/s	101MB/s 117MB/s 13MB/s
master 操作（最后一分钟） master 操作（最后一小时） master 操作（自从重启动）	325Ops/s 381Ops/s 202Ops/s	533Ops/s 518Ops/s 347Ops/s

表三：两个 GFS 集群的性能表

读取速率要比写入速率高很多。总共的负载包含的读取要远大于包含的写入，这符合我们的预期。两个集群都处于中等的读取压力。特别是，集群 1 在一周内都维持了 580MB/s 的读取速度。他的网络支持 750MB/s，因此它很有效的利用了资源。集群 B 可以支持峰值读取速度是 1300MB/s，它的应用只用到了 380MB/s

6.2.4 master 的负载

表三也包括了对 master 的操作率，大概是每秒 200 到 500 个操作。master 可以轻松应付这样的访问速率，因此这不是主要的负载瓶颈。

在早期版本的 GFS，master 偶然会成为负载瓶颈。它花费了大量的时间顺序扫描大型目录（包含了数十万文件）来查找某一个文件。因此我们改了一下 master 的数据结构，这样支持高效的二进制的 namespace 检索。现在它可以支持每秒很多个上千个文件的访问。如果需要，我们还可以使用名字查找 cache 在 namespace 的数据结构来进一步提高性能。

6.2.5 恢复时间

在某一个 chunkserver 失效以后，其上的 chunk 可能会低于副本数量，这样于是就需要把这些 chunk 进行复制来达到相应的副本级别。这个操作所花的时间取决于资源的数量。在我们一个试验中，我们把集群 B 上编的一个 chunkserver 删掉。这个 chunkserver 上有大概 15000 个 chunk，共计 600GB 数据。为了减少对正在运行的应用程序的影响，以及修正调度决策，我们缺省参数是限制这个集群中的并发克隆数量是 91 个（总 chunkserver 数量的 40%），每一个克隆操作可以展到 6.25MB/s（50mbps）的带宽。所有的 chunk 在 23.2 分钟内恢复了，大约有效复制速率是 440MB/s。

在另外一个是严重，我们杀掉了两个 chunkserver，每个 chunkserver 大概有 16000 个 chunk 并且包含大概 660GB 数据。这种双重故障导致了 266 个 chunk 只有单个副本。这些 266 个 chunk 于是克隆操作就提升到比较高的优先级，并且至少要在 2 分钟内恢复到起码两个副本，这样就可以能让系统恢复到容错另一个

chunkserver 失效的情况。

6.3 处理能力细目

在本节中，我们对比展示了两个 GFS 集群的工作量的细目情况。这两个集群和 6.2 中应用的集群不同。集群 X 是用于研发部分的，集群 Y 是生产数据处理的。

6.3.1 方法论和注意事项

我们考虑得处理能力只包含客户端发起的原始请求，这些请求代表了整个我们应用程序产生的对文件系统的工作量。处理能力中不包括处理客户请求所引起的中间请求，以及内部的后台应用，比如转发写请求或者重新均衡 chunk 分布等等。

I/O 操作的统计是基于 GFS 服务器的实际 RPC 请求 log 重构的。例如，GFS 客户端代码可能把一个读操作分解成为多个 RPC 调用，这样来增加并发度，我们从这些 RPC 来推断原始的读取请求。因为我们的操作模式是高度程式化的，所以我们可以把任何不符合的数据认为是误差。应用程序记录的日志可能会稍微对数据的精确性有点帮助，但是基本上不可能在上千个客户端上重新编译和重新运行应用程序，并且从这些机器上获得这些日志结果也是很麻烦的事情。

我们应当避免从我们的负载中进行过度归纳和推广。因为 Google 完全控制 GFS 和其上的应用程序，并且应用程序也是因为 GFS 而特意做调优，并且反过来说，GFS 也是特意为这些应用而设计的。这些相互影响也可能在通用的应用和文件系统中，但是在我们的例子中更显著而已。

6.3.2 chunkserver 的负载

表 4 显示了操作大小的分布。读取操作的大小体现了一个双峰分布。小型的读取请求（小于 64KB），是从面向查找的客户端发起的，用于在巨量文件中查找小块数据的。大型的读取请求（大于 512KB）是来自于对整个文件的序列读取。

在集群 Y 上，没有读到任何数据的请求占了相当的一部分。在我们的应用中，尤其是在生产系统中，经常使用文件作为生产者-消费者队列。生产者并行添加到文件，同时消费者从文件读取。某些情况下，消费者超过了生产者的速度的时候，就会出现没有读到任何数据的情况。集群 X 这样的情况出现的比较少见，这是因为通常集群 X 用于短期的数据分析任务，而不是长期的分布是应用。

写请求的大小也同样体现了双峰分布。大型写入（超过 256KB）通常由写入者的重要缓存操作产生。写入者缓存较小的数据，checkpoint 或者同步点，或者小型写入产生的小数据（小于 64KB）。至于记录增加操作来说，集群 Y 看起来再大型记录增加方面要比集群 X 占百分比多，这是因为我们的实用集群 Y 的生产系统，是为 GFS 做了更极端的调优。

表 5 显示了不同大小的操作请求中的数据传输总数。多于所有操作来说，较大的操作（超过 256KB）占据了主要的传输量，较小的读取（小于 64KB）占据的传输量比较小，但是却是读取操作的相当一部分，这是因为随即寻找的工作量导致的。

操作	读		写		纪录增加	
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 4：操作大小百分比细目（%）。对于读操作，大小是实际读取的数据和传送的数据，而不是请求的数据量。

6.3.3 增加纪录与写操作

纪录增加是在我们的应用系统中大量使用的。对于集群 X 来说，增加纪录写操作和普通写操作的比例按照字节比试 108:1,按照操作比试 8:1。对于集群 Y 来说，我们的生产系统的比例是 3.7:1 和 2.5:1。进一步说，这个比例说明在我们的两个集群上，纪录增加都比写操作要大。对于集群 X 来说，在测量过程中，整体纪录增加占据的比率相对算小的，因此结果受到一个或者两个应用的某些特定的 buffer 大小的影响。如同我们预期的，我们数据操作负载是受到纪录增加的影像而不是是改写的影响。我们测量第一个副本的数据改写情况。这近似于一个客户端故意覆盖刚刚写入的数据情况，而不是增加新数据的情况。对于集群 X 来说，改写操作在所占据字节上小于 0.0001%，并且在所占据操作上小于 0.0003%。对于集群 Y，这个比率都是 0.05%。虽然这只是某一片断的情况，依旧是高于我们的预期。这是由于大部分这些覆盖写，是因为客户端发生错误或者超时以后重试的情况。这本质上不算作负载量，而是重试机制的一个结果。

操作	读		写		纪录增加	
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	0.1	46.8
1M…inf	6.4	30.1	3.3	28.0	53.9	7.4

表 5：操作大小字节传输细目表（%）。对于读取来说，这个大小是实际读取的并且传输的，而不是请求读取的数量。两个不同点是如果读取尝试读超过文件结束的大小，那么实际读取大小就不是试图读取的大小，尝试读取超过文件结束的大小这样的操作在我们的负载中并不常见。

6.3.4 master 的负载

表 6 显示了 master 上的请求类型区分的明细表。大部分的请求都是询问 chunk 位置的（FindLocation）和对读取和颁布令牌信息（FindLease-Locker）以及对数据更新的。

集群 X 和 Y 在删除请求上有着显著的不同，因为集群 Y 存储了生产数据集，会定义重新产生和用新的版本替换。这些不同点也在 Open 请求中有所体现，因为一个文件的旧版本可能随着新打开改写而默认删除（如同 unix 的 open 操作中的”w”模式）。

FindMatchingFiles 是一个模式匹配请求，支持”ls”以及类似文件系统的操作。它不同于其他 master 上的请求，他可能会检索 namespace 的大部分内容，因此可能会非常耗时。集群 Y 这种操作要多一些音位自动化数据处理进程需要尝试检查文件系统的各个部分来试图理解整体的进程状态。与之不同的是，集群 X 的应用程序更加倾向于单独的用户控制，通常预先知道自己所需要使用的全部文件名。

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
其他	0.5	0.8

表 6：master 请求类型明细（%）

7 经验

在建造和部署 GFS 的过程中，我们经历了不少问题，某些是操作上的，某些是技术上的

起初，我们构思 GFS 用于我们生产系统的后端文件系统。随着时间推移，增加了对研发任务的支持。于是开始增加一些小的功能比如权限和配额，到现在全面支持了这些内容。因为我们生产系统是严格受控的，但是用户层却不是这样的。于是我们研发了更多的架构支持用于防止用户间的干扰。

我们最大的问题是磁盘和 linux 相关的问题。很多磁盘都声称他们支持只要是 IDE 的 linux 驱动都可以，但是实际上却不是，实际上他们只支持最新的驱动。因为协议版本很接近，所以大部分磁盘都可以用，但是偶尔也会由于这些不匹配导致驱动和核心对于驱动器的状态误判。这会导致 kernel 无知觉的丢失数据。这些问题促使我们是用 checksum 来检查数据的损坏，同时我们也修改了核心来处理这些协议上的不匹配。

早先情况下，我们在 linux2.2 核心上有些问题，出在 fsync（）的效率问题。它的效率与文件的大小有关而不是和文件修改部分的大小有关。这在我们的操作 log 过长的时候就会有问题，尤其是在我们尚未实现 checkpoint 的时候。我们费了很大的力气用同步写操作解决这个问题，最后移植到 linux2.4 核心上。

另一个 linux 问题时单个写者-读者锁，就是说在某一个地址空间的任意一个线程都必须在从磁盘载入（page in 读者锁）的时候先 hold 住,或者在 mmap()调用（写者锁）的时候改写地址空间。我们发现在我们系统负载很轻的情况下有偶尔超时的情况，并且费了好大的力气去寻找资源的瓶颈或者硬件的问题。终于，我们发现这是单个 lock，在磁盘线程交换以前的映射数据到磁盘的时候，锁住了当前的网络线程把新数据 map 到内存。因为我们主要受限于网络接口，而不是内存 copy 的带宽，所以，我们用 pread()替换掉 mmap，用了一个额外的 copy 动作来解决这个问题。

尽管偶尔还是有问题，linux 的源代码还是使得沃恩能够迅速定位并且理解系统的行为。在适当的时候，我们会改进内核并且和公开源码组织共享这些改动。

8 相关工作

如同其他大型分布式文件系统比如 AFS[5]一样，GFS 提供了一个与位置无关的 namespace，使得数据可以根据负载或者容错的原因在不同位置进行移动。但是不同于 AFS 的是，GFS 把文件的数据分布到存储服务器上的方式，更类似 Xfs[1]和 Swift[3]，这是为了提高整体性能以及增强容错能力。

由于磁盘相对来说比较便宜，以及副本方式比传统的 RAID[9]简单许多，GFS 目前只使用副本方式来进行冗余，因此要比 xFS 或者 Swift 花费更多的存储。

与 AFS，xFS,Frangipani[12]，以及 Intermezzo[6]不同的是，GFS 并没有在文件系统层面提供 cache 机制。我们的主要工作量在单个应用执行的时候几乎不会有重用的可能，因为他们无论是流式读取还是随机寻找，都是对大型数据集的操作。

某些分布式文件系统比如 Frangipani,xFS,Minnesota’s GFS[11],GPFS[10]，去掉了中心服务器，并且依赖于分布式算法来保证一致性和管理性。我们选择了采用中心服务器的做法是为了简化设计，增加可靠性和扩展能力。尤其是，一个中心 master，由于它已经有了几乎所有的相关 chunk 信息以及控制 chunk 信息的能力，它可以非常简化实现传统的 chunk 分布和副本机制。我们通过减少 master 的状态量大小（元数据大小）以及全面分布 master 的状态到不同的机器上来保证容错能力。扩展能力和高可用性（对于读取）目前是通过我们的影子 master 机制来保证的。对 master 的状态更改是通过增加到一个往前写的 log 中持久化。因此我们可以在使用如同在 Harp[7]中使用的 primary-copy 机制来保证高可用性以及对我们机制的一个强有效的一致性保证。

我们用类似 Lustre[8]的方法来提供针对大量客户端的整体性能。不过，我们通过面向我们的应用的方式简化了问题的解决方法，而不是面向提供兼容 POSIX 文件系统的方式。此外，GFS 假设我们大量使用了非可靠节点，所以容错处理是我们设计的核心。

GFS 很类似 NASD 架构[4]。NASD 架构是基于网络磁盘的。GFS 用的是普通计算机作为 chunkserver，如同 NASD 原形中一样。所不同的是，我们的 chunkserver 是滞后分配固定大小的 chunk 而不是用一个变长的结构。此外，GFS 实现了诸如重新负载均衡，副本，恢复机制等等在生产环境中需要的特性。

与 Minnesota’s GFS 和 NASD 不同，我们并不改变存储设备的访问方式。我们集中于解决用普通计算机来解决日常的分布式系统所需要的数据处理。生产者消费者队列是通过原子的纪录增加来解决的，类似在 River[2]中的分布式队列。River 使用的是分布在机器组的基于内存的队列，并且仔细控制数据流。GFS 用的是持久化文件，可以由很多生产者并发增加的方式。River 模式支持 m-到-n 的分布式队列但是缺少用持久化存储的容错机制，GFS 只支持 m-到-1 的队列。多个消费者可以同时读取一个文件，但是他们输入流的区间必须是对齐的。

9 结束语

Google 文件系统展示了使用常规硬件支持大规模数据处理的质量属性。虽然很多设计要素都是针对我们的特殊需要而定制的，但是很多都可以适用于基于成本考虑的类似规模的数据处理任务。首先我们依据

现在和预期的应用负载和技术环境来审视传统的文件系统约定。我们的审视引导我们使用设计领域完全不同的思路来设计。我们在设计上认为部件失效属于常态而不是异常，通过面向巨型文件的主要追加模式进行优化（并发优化）并且读取优化（通常序列化读取），以及扩展和放松了标准文件系统接口来改进整个系统。

我们系统使用持续监控，复制关键数据，快速自动恢复来提供了容错处理。chunk 复制使得我们可以对 chunkserver 的失效进行容错。这些失效的经常发生使得在线修复机制变得很有必要，使得需要尽快修复和补偿丢失的副本。此外，我们使用 checksum 来检查磁盘或者 IDE 系统级别的数据损坏，因为在这样大的一个系统中，磁盘数量惊人，所以损坏率也很高。

我们的设计要求对不同任务的大并发读取和写入有一个很高的合计吞吐量。我们通过分离文件系统控制和文件数据传输的设计来实现，我们通过 master 来进行文件系统的控制，通过 chunkserver 和客户端来进行文件数据的传输。master 包括了常用的操作，并且通过大块的 chunk 大小进行元数据的缩小，以及通过 chunklease 令牌减少数据量，chunk 令牌是授权给第一个副本以数据变更操作。这些使得一个简单，中心 master 不再成为一个瓶颈。我们相信我们对网络协议栈的优化可以提升对于每个客户端的当前的写入吞吐量限制。

GFS 成功的实现了我们的存储要求以及在 Google 内部广泛应用于研发部门和生产数据处理的存储平台。它是我们持续创新和解决整个 web 范围的一个有力工具。

参考资料

[1] Thomas Anderson, Michael Dahlin, Jeanna Neefe,David Patterson, Drew Roselli, and Randolph Wang. Serverless networkfil e 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 disks triping 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, ErikR iedel, 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, 203.

[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] FrankS chmuck and Roger Haskin. GPFS: A shared-diskfi le 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.