LEC 3 GFS

为什么我们阅读这个论文?

1.Map/reduce 使用这种文件系统

2.关于课程 6.284 的全部主题都会在这个论文中。包括用一致性换取简单和性能(trading consistency for simplicity and performance)以及后续设计的动机(motivation for subsequent designs)。

3.好的系统论文--从 apps 到网络都有详细地说明，包括性能、容错、一致性。

4.影响力:许多其他的系统使用 GFS(e.g.,Bigtable,Spanner) ,HDFS(hadoop Distribution File System)也是基于 GFS 的。

一致性是什么?

1.它是系统保持正确性的必要条件(a correctness condition)。

2.当数据需要被拷贝或者存在副本时一致性是非常重要但又很难被实现，尤其是当应用程序并发访问数据时。

3.如果一个应用程序进行写操作，那么之后的读操作可以观察到什么？如果这个读操作来自其他应用程序又会看到什么？在写操作完成之前，如果没有加锁读操作应该看到的是不完整的新数据,如果有加锁看到的应该是旧数据，这个读操作来自其他应用程序要考虑延时性问题，其余和之前是一样的。

弱一致性:

read()可能返回不新鲜的数据(脏数据)——不是最近写操作的结果。

强一致性:

read()返回的结果数据总是最近一次的写操作结果。

弱一致性和强一致性之间的关系、权衡:

强一致性对程序的写操作（application writers）表现不错。

强一致性将会影响性能(不好的影响,耗费性能)。

弱一致性有好的性能并且容易扩展到更多的服务器。

弱一致性很复杂很难推理。

两者之间更多的权衡会导致不同的正确性条件，这被称为“consistency models”(一致性模型)。这是今天我们第一次看到这个名词，在之后的每篇论文中几乎都会出现这个术语。

理想的一致性

让我们回到单机的情况。

一个存在副本的文件表现的跟单一文件系统一样,就像很多客户端访问存在同一个机器的单一磁盘的文件,我们就说这个系统表现的很好。

如果一个应用程序写入,之后的读取将看到该写入。

那么如果两个应用程序并发的写入同一文件又如何?Question:在单个机器上将会发生什么?

答:在文件系统中这种行为经常是未定义的 —— 文件也许会混合两个写操作的内容。

如果两个应用程序并发写同一个目录会怎么样? Question:在单个机器上会发生什么?

答:先写一个,完成后再写另一个也就是一个一个顺序执行(使用锁)。

(译者自问:以上的单个机器是指应用程序并发写入一台服务器吗?)

实现理想一致性的挑战

1.并发,正如我们所看到的,在真实环境中存在很多磁盘。

2.机器故障,任何操作都可能无法完成(机器很多)。

3.脑裂(Network partitions),存在访问不到另一机器/磁盘的情况。

4.难以有效的使用网络。

5.性能:很多并发的读写操作,map/reduce 工作会从 GFS 中读取数据,然后保存最后的结果

为什么这些挑战难以被克服?

因为多个客户端和多个服务器之间需要通信,这可能会影响性能;通信协议可能变得复杂(见下周)而变得难以正确实施系统;在 6.824 中的许多系统不能提供理想的一致性模型,GFS 就是一个例子。

GFS 的目标

1.因为有很多机器,所以发送错误是很常见的情况,GFS 系统必须有容错性。假设一台机器每年出现一次故障,如果有 1000 台机器,每天大约有 3 台会出现故障。

2.高性能,因为存在很多并发的读操作和写操作。Map/Reduce 任务读取并将最终结果存储在 GFS 中。注意:不是临时的中间文件。

3.有效的利用网络:节省带宽。

然而这些挑战很难与“理想”的一致性结合起来。

High-level desgin/Reads

定义目录、文件、命名(names)、open/read/write,但是都不服从 posix 标准。

成百上千个带有硬盘的 linux 块服务器(linux chunk servers):

- 存储 64MB 的块(an ordinary Linux file for each chunk)。
- 每个块在三个服务器上都有备份。

Question:除了数据可用，三备份方案给我们带来了什么?

读取热文件(hot files)时可以做负载均衡。

Question:为什么不把每一份文件存储在 RAID 硬盘?

RAID 不常用，我们想给整台机器做容错，而不是仅仅针对存储系统。

Question: 为什么 chunk 这么大?

分摊管理的代价，减少主机中的状态数。

GFS 的 master 服务器知道目录层次

对于目录而言，知道里面有哪些文件。

对于文件而言，知道哪些数据块服务器存储了相关的 64MB 大小数据块

master 服务器在内存中保存状态信息，每个 chunk 在 master 服务器上面只保存 64bytes 大小的元数据

master 服务器有为元数据准备的可回收数据库，操作日志刷新到磁盘,偶尔异步压缩信息检查点,可以从断电故障后快速恢复。

同时存在备份的 master 服务器(shadow master,影子服务器)，数据略比主控服务器服务器延迟，可以被提升为 master 服务器

客户端读操作顺序如下:

1.客户端向 master 发送文件名(file name)和块索引(chunk index)。

2.主服务器回复具有该组块的服务器集,并且响应信息中包括块的版本号，然后客户端会缓存该信息。

3.客户端根据服务器返回的块服务器集去访问最近的块服务器，然后检查版本号，如果版本号错误，需要重新联系 master 服务器。

Writes

客户端随机写入已存在的文件:

1.client 先向 master 询问 chunk 的位置以及 chunk server 中 primary(块服务器中的主服务器)的位置和版本号。

2.primary 有 60s 的租约(有租约的就是 primary),client 根据网络拓扑计算副本链(chain of replicas)。

3.client 发送数据到第一个副本,该副本转发到网络使用的其他管道(分配负载,或者说负载均衡)。这里的意思是,副本转发 client 写入的数据的管道和 client 写入的不同,不占用同一个带宽,提高效率。

4.副本确认数据接收

5.client 告诉 primary 写入:

primary 分配序列号并写入,然后告诉其他副本写入,完成后发送 ack 给 client。

如果有另一个并发客户端在同一个地方写东西怎么办?client2 在 client1 之后按顺序进行写操作会覆盖 client1 的写入数据。现在 client2 再次写入,这一次先写入的 client1 因为写入速度很慢所以会覆盖 client2 的写入数据，这就导致所有所有的副本有同样的数据(=一致,consisent),但混合了来自 client1 和 client2 的数据(=不等于 defined)。

client append(not record append):

同样的操作可能会将 client 1 和 client 2 的数据按任意顺序排列。

一致,但没有定义(consisent,but not defined)。

或者,如果只有一个客户端写,那就没有问题啦,既一致又明确。

在 unix 上并发写也会出现乱七八糟的东西。

Record append

client record append(GFS 支持原子操作,至少保证一次 append)。

primary 服务器选择记录需要添加到的文件位置，然后发送给其他副本。如果和一个副本的联系失败，那么 primary 会告诉客户端重试，如果重试成功，有些副本会出现追加两次的情况(因为这个副本追加成功两次)。当 GFS 要去填塞 chunk 的边缘时，如果追加操作跨越 chunk 的边缘，那么文件也可能存在空洞。

1.client 向 master 询问 chunk 的位置。

2.client 将数据推送到副本,但不指定偏移量。

3.当数据在所有 chunk server 上时,client 联系 primary。

3.1.primary 分配序列号。

3.2.primary 检查 append 是否适合 chunk,如果不合适,填充到块边界(pad until chunk boundary)。

3.3.primary 为 append 操作指定偏移量。

3.4.primary 应用本地更改。

3.5.primary 将请求转发给副本。

这里讨论一个错误的情况:R3 在 write 操作的过程中失败了,primary 检测到了错误,并告诉客户端再次尝试。

4.client 在重新连接 master 后重试。

4.1.master 可能同时出现 R4(或者 R3 又回来了)。

4.2.一个副本在一个字节序列中可能有一个 gap,所以不能只是 append。

4.3.填充到所有副本的下一个可用偏移量。

4.4.primary 和 secondaried 执行 write 操作

4.5.从所有副本接收 ack 应答后,primary 对 client 做出响应。

Chunk 数据的持久化

有些数据因为错过了更新，所以过时了。

通过 chunk 的版本号判断数据是否不新鲜的，在发生租约前，增加 chunk 版本号码，将数据发送到主数据块服务器，同时在其他数据块服务器中备份，主服务器和数据块服务器长久的存储版本信息。

发送版本号给客户端。

版本号帮助主控服务器和客户端判断备份是否不新鲜。

Housekeeping(内务处理)

如果 master 不更新租约,master 可以指定新的 primary。

如果副本数量低于某个值,master 将复制 chunk。

master 会重新平衡副本。

如果一个副本没有回复，那么客户端会重试。

Failures

chunk server 易于替换,如果替换失败可能会导致一些客户端重试(或者重复记录)。

master:可能出现宕机导致 GFS 不可用，此时影子 master 可以提供只读操作,这可能会返回过时的数据(master 恢复很快的,如果 master 不能恢复,然后 master 又重新启动了,系统必须避免出现两台 master)。

问题:为什么不能提供 write 操作呢?

会出现脑裂的情况,详细见下一章节讨论。

GFS 是否达到了"理想"的一致性

分为两种情况:目录和文件。针对目录而言是达到了强一致性(仅有一个),但是 master 不总是可用的,况且其伸缩度有限。针对文件而言不总是强一致性的。我们分以下 f1,f2 情况讨论

f1.原子 append 的突变:记录可以在两个偏移量处重复,而其他副本可能在一个偏移处有孔(hole)

f2.非原子 append 的突变:几个 client 的数据可能混合在一起。如果你愿意可以使用原子 append 或临时文件+原子级重命名的操作。

一个"不幸"的 client 可以在短时间内读取过时的数据。因为突发的失败会导致 chunks 之间数据的不一致,例如,primary chunk server 试图更新 chunk,但后来失败了,同时复制品也过时了。当客户端刷新到租约时它会了解最新的版本,此时 client 可以读取最新的块。

作者声称弱一致性对应用程序来说不是大问题，因为:

1.大多数文件更新仅是 append 更新,应用程序可以在 append record 中使用 UUID 来检测重复项,并且应用程序可能仅是读取少量的数据(不是失效的数据)。

2.应用程序可以使用临时文件和原子级的重命名操作。

GFS 的设计者为了最求更好的性能和更简单的设计而放弃理想的一致性模型

性能

巨大的读操作总吞吐量(3 个副本)达到 125 MB/sec，接近网络饱和状态。

写入不同的文件低于可能的最大值:作者怪网络堆栈并且 chunk 直接的复制操作会引起延迟。

并发追加同一份文件则被服务器存在的最新的 chunk 所限制。

15 年来数字和说明发生了很大的变化。

总结

GFS 中工作很好的?

1.巨大的顺序读写操作

2.追加

3.巨大的吞吐量

4.数据之间的容错(3 个副本)

GFS 中做的不怎么好的?

1.master 服务器的容错。

2.小文件（master 服务器的瓶颈）。

3.客户端可能会看到过时的数据。

4.append 可能重复。

GFS 常见问题

问：为什么原子记录 append 至少一次附加一次，而不是一次附加一次?

很难一次精确地执行 append 操作，因为 primary 随后需要保持状态以执行重复检测。该状态必须在服务器之间复制，这样，如果主服务器发生故障，该信息就不会丢失。您将仅在 lab3 中实现一次，但是使用 GFS 使用的更复杂的协议。

问：应用程序如何知道块的哪些部分由填充和重复记录组成?

为了检测填充，应用程序可以在有效记录的开头放置一个可预测的魔幻数(magic number)，或包括一个仅在记录有效时才有效的校验和。该应用程序可以通过在记录中包含唯一 ID 来检测重复项。然后，如果它读取的 ID 与先前的记录具有相同的 ID，则它知道它们是彼此重复的。GFS 为处理这些情况的应用程序提供了一个库。

问：考虑到原子记录追加将其写入文件中不可预测的偏移量，客户端如何找到其数据?

append（通常是 GFS）主要用于读取整个文件的应用程序。这样的应用程序将查找每条记录（请参阅上一个问题），因此它们不需要事先知道记录的位置。例如，该文件可能包含一组并发 Web 爬虫遇到的一组 URL。任何给定 URL 的文件偏移量都无关紧要；读者只希望能够阅读整个 URL 集。

问：论文中提到参考计数(reference counts)-它们是什么？

它们是快照 copy-on-write(写时复制)实现的一部分，当 GFS 创建快照时，它不会复制数据块，而是增加每个数据块的引用计数器。这使得创建快照的成本较低。如果客户端写入了一个块，并且 master 注意到引用计数大于 1，则 master 会首先创建一个副本，以便客户端可以更新副本（而不是快照的一部分）。您可以将其视为延迟复制，直到绝对必要。希望不是所有的块都会被修改，并且可以避免制作一些副本。

问：如果应用程序使用标准的 POSIX 文件 API，是否需要进行修改才能使用 GFS?

是的，但是 GFS 不适用于现有的应用程序。它是为新编写的应用程序（例如 MapReduce 程序）设计的。

问：GFS 如何确定最近的副本的位置？

本文暗示GFS会根据存储可用副本的服务器的IP地址来执行此操作。在2003年，Google 必须以这样一种方式分配 IP 地址：如果两个 IP 地址在 IP 地址空间中彼此靠近，则它们在机房中也彼此靠近。

问：Google 仍然使用 GFS 吗?

Google 仍在使用 GFS，它是 BigTable 等其他存储系统的后端。自从工作量增加和技术变化以来，多年来，GFS 的设计无疑已经进行了调整，但是我不知道细节。HDFS 是 GFS 设计的 public-domain,许多公司都在使用。

问：Won't the master be a performance bottleneck?

它确实具有这种潜力，并且 GFS 设计师为避免此问题进行了麻烦。例如，主机将其状态保存在内存中，以便可以快速响应。评估表明，对于大文件/读取 (GFS 定位的工作负载)，主服务器不是瓶颈。对于小文件操作或目录操作，主机可以跟上（请参阅 6.2.4）。

问：GFS 以正确性换取性能和简单性的接受程度如何?

这是分布式系统中经常出现的主题。高度的一致性通常需要复杂的协议，并且需要在机器之间进行闲聊（正如我们在接下来的几堂课中将会看到的）。通过利用特定应用程序类可以容忍宽松一致性的方式，人们可以设计出具有良好性能和足够一致性的系统。例如，GFS 针对 MapReduce 应用程序进行了优化，该应用程序需要对大型文件具有较高的读取性能，并且可以在文件中留有空洞，记录显示多次以及读取不一致。另一方面，GFS 不适合银行存储帐户余额的场景。

问：如果主机发生故障怎么办?

有些 replica master 具有 master 状态的完整副本。如果当前的主服务器发生故障，则未指定的机制将切换到其中一个副本（第 5.1.3 节）。人们可能必须干预才

能指定新 master。无论如何，几乎可以肯定，这里潜伏着单个故障点，从理论上讲，这将阻止从主故障中自动恢复。我们将在以后的讲座中看到如何使用 Raft 制作容错主机。