计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
2025,34(10):101−109 [doi: 10.15888/j.cnki.csa.009958] [CSTR: 32024.14.csa.009958]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

DFS 元数据缓存一致性的轻量级维护机制①

吴雨飞[1], 李　诚[1,2], 李嘉豪[1], 王一多[3]

[1](中国科学技术大学 计算机科学与技术学院, 合肥 230026)
[2](合肥综合性国家科学中心 人工智能研究院, 合肥 230026)
[3](中国电信云计算研究院, 北京 100195)
通信作者: 李　诚, E-mail: chengli7@ustc.edu.cn

摘　要: 分布式文件系统 (DFS) 能够高效管理数据中心的存储资源, 已经成为支撑众多数据密集型应用的核心基础设施. 为了降低执行文件系统操作时路径解析的开销, 分布式文件系统普遍采用了客户端元数据缓存, 同时在客户端之间同步元数据修改以保证元数据缓存的一致性. 但是现有的元数据缓存一致性方案在引入了高昂性能开销的情况下, 仍然未能正确同步全部的目录树修改, 导致正确性问题. 针对这一现状, 本文提出了一种维护元数据缓存一致性的轻量级机制, 通过基于并发广播机制的目录树修改方法以及基于惰性广播和墓碑机制的目录删除方法, 既突破了现有方案的性能瓶颈又解决了正确性缺陷. 实验结果表明, 这种机制使得分布式文件系统中目录树修改操作的延迟降低了 65.8%–66.9%, 吞吐量提升了 2.94–4.53 倍. 此外, 在运行 Spark 作业时, 作业提交的延迟下降了 43.6%.

关键词: 存储系统; 分布式文件系统; 元数据; 缓存; 一致性

引用格式: 吴雨飞,李诚,李嘉豪,王一多.DFS 元数据缓存一致性的轻量级维护机制.计算机系统应用,2025,34(10):101–109. http://www.c-s-a.org.cn/1003-3254/9958.html

Lightweight Mechanism to Maintain Consistency of Metadata Cache in DFS

WU Yu-Fei[1], LI Cheng[1,2], LI Jia-Hao[1], WANG Yi-Duo[3]

[1](School of Computer Science and Technology, University of Science and Technology of China, Hefei 230026, China)
[2](Institute of Artificial Intelligence, Hefei Comprehensive National Science Center, Hefei 230026, China)
[3](China Telecom Cloud Computing Research Institute, Beijing 100195, China)

Abstract: Distributed file system (DFS), capable of efficiently managing storage resources in data centers, has become the cornerstone of data-intensive applications. To reduce the overhead of path resolution during file system operation, distributed file systems adopt client-side metadata cache, while ensuring the consistency of metadata cache by synchronizing metadata modifications. However, the state-of-the-art metadata consistency mechanisms introduce significant performance overhead and still fail to correctly synchronize all metadata modifications. To address these limitations, a lightweight metadata consistency mechanism is proposed. By employing a namespace modification method based on concurrent broadcast and a directory deletion method based on lazy broadcast and tombstones, the proposed mechanism overcomes the performance bottleneck and resolves the correctness issue. The evaluation results show that the proposed design reduces the namespace modification latency by 65.8%–66.9% and achieves throughput speedups ranging from 2.94–4.53 times. For real-world applications, the job submission latency of Spark workloads is reduced by 43.6%.

Key words: storage system; distributed file system (DFS); metadata; cache; consistency

① 基金项目: 国家重点研发计划 (2024YFB4505701)
收稿时间: 2025-02-19; 修改时间: 2025-03-12; 采用时间: 2025-03-28; csa 在线出版时间: 2025-08-26
CNKI 网络首发时间: 2025-08-27

1 引言

分布式文件系统 (distributed file system, DFS) 能够高效管理数据中心的存储资源, 已经成为支撑众多数据密集型应用的核心基础设施. 例如, 谷歌和脸书等公司内部数据仓库的存储层采用了分布式文件系统[1,2], 微软和阿里云等云计算厂商基于分布式文件系统提供对象存储和块存储等云存储服务[3,4], DeepSeek 将分布式文件系统作为大模型训练语料和推理上下文的存储底座[5].

当前, 主流分布式文件系统普遍将元数据 (如层级的目录结构信息) 与数据解耦, 并使用专用的元数据节点管理元数据. 在执行文件系统操作时, 客户端首先需要解析路径, 即逐级从多个元数据节点读取目标路径上各层目录的元数据, 并验证用户权限, 之后才能执行实际的文件系统操作. 路径解析操作处在文件系统操作的关键路径上, 在数据中心内文件平均大小较小[6]而路径平均深度较深[7]的背景下, 解析文件路径的开销甚至高于读写文件数据的开销, 对文件系统的整体性能产生显著影响. 为了降低路径解析延迟, 分布式文件系统普遍在客户端构建元数据缓存[8–10], 使得客户端在解析路径时能够从本地读取缓存中的目录元数据. 缓存机制能够减少元数据的网络传输开销并降低元数据节点的负载, 显著提升了分布式文件系统的整体性能.

当分布式文件系统在客户端构建元数据缓存时, 维护元数据缓存的一致性至关重要. 如果元数据缓存和元数据节点中的元数据不一致, 客户端在使用缓存进行路径解析时, 会导致元数据操作发生在错误的目录下, 严重时可能造成文件系统数据损坏.

当前最先进的元数据缓存机制是 InfiniFS[7]提出的乐观访问元数据缓存 (optimistic access metadata cache), 该机制通过以下方式维护元数据缓存的一致性: 客户端执行涉及目录树修改的元数据操作 (目录重命名和目录删除) 时, 需要通过协调节点 (coordinator) 将操作广播到各个元数据节点, 并将操作参数写入元数据节点的失效列表 (invalidation list) 中. 客户端执行路径解析时, 先从本地缓存中读取各级目录的元数据, 再将路径发送至元数据节点, 由元数据节点根据失效列表判断客户端的元数据缓存是否有效.

但是, 这种设计不仅在性能上引入了较高的开销, 也在正确性上存在缺陷: (1) InfiniFS 采用中心化的协调节点串行广播目录树修改到各个元数据节点, 导致目录树修改操作开销高昂. (2) InfiniFS 中未对目录删除操作进行广播, 难以保证元数据缓存的一致性.

鉴于此, 本文提出了一种维护元数据缓存一致性的轻量级机制. 主要贡献如下: (1) 提出了基于并发广播机制的目录树修改方法, 允许并发地记录目录树修改历史, 突破了现有方案中协调节点串行广播造成的性能瓶颈, 提升了目录树修改操作的吞吐量. (2) 设计了基于惰性广播和墓碑机制的目录删除方法, 以尽可能低的性能开销, 解决了现有方案中未广播目录删除造成的正确性缺陷, 保证了元数据缓存的一致性.

实验结果表明, 相比于 InfiniFS, 本文提出的机制能够将分布式文件系统中目录树修改操作的延迟降低 65.8%–66.9%, 峰值吞吐量提升 2.94–4.53 倍. 在运行 Spark[11]作业时, 作业提交的延迟下降了 43.6%. 由此可见, 这种轻量化的缓存机制能够以更低的开销维护客户端元数据缓存的一致性, 在保证分布式文件系统正确性的同时显著提升了目录树修改操作的性能.

本文第 2 节介绍研究背景和相关工作. 第 3 节分析维护客户端元数据缓存一致性所面临的挑战. 第 4 节详细阐述系统设计要点. 第 5 节从多个维度进行性能评估. 第 6 节为总结和展望.

2 研究背景

2.1 元数据管理

早期的分布式文件系统为了简化系统设计, 普遍采用单个元数据节点管理整个文件系统的元数据[1,12]. 近年来, 随着数据规模的日益扩大, 单个节点有限的资源难以应对文件数量的增长, 涌现出一系列将分布式键值数据库作为元数据节点, 管理分布式文件系统元数据的工作[2,7,13–15]. 这些系统普遍采用父目录标识符 (inode) 和名称为主键, 将树状结构映射为键值对, 并将同一父目录下的条目划分到同一个数据库分片上. 分布式文件系统中的各项元数据操作可以通过数据库事务实现.

在这种架构下, 执行文件系统操作时, 客户端需要先从元数据节点逐级读取目标路径中各层目录的元数据, 验证用户权限, 完成路径解析, 最终定位到操作对象. 例如, 当客户端打开路径“/A/B/C”处的文件 C 时, 需要以根目录标识符与名称 A 作为主键, 读取目录 A 的元数据以验证权限; 再以目录 A 标识符与名称 B 作为主键, 读取目录 B 的元数据并检查权限; 最终读取到 C

的元数据, 获知文件 C 中数据块的分布. 这一流程涉及客户端与元数据节点之间的多次远程过程调用, 路径解析的延迟严重影响了分布式文件系统中各项操作的性能[7].

2.2 元数据缓存及其一致性

为了降低路径解析的开销, 客户端普遍利用文件系统访问的时间局部性, 在本地构建元数据缓存进行优化[8–10]. 在路径解析时, 客户端会优先从本地的缓存中读取目录元数据. 只有发生缓存缺失时, 客户端才需要从元数据节点进行读取. 尽管客户端缓存避免了多次读取元数据的网络请求开销, 但会引入新的一致性问题. 当多个客户端缓存了同一目录树中的元数据时, 如果某一个客户端对目录树进行了修改, 若没有额外的一致性维护机制, 其他客户端可能得到错误的路径解析结果. 例如, 客户端 1 和 2 同时缓存了目录树 “/A/B/C”, 当客户端 1 将目录 C 重命名为目录 D 时, 客户端 2 根据本地缓存, 仍然认为目录 C 存在, 并且能够以“/A/B/C”路径在目录 D 下执行操作.

在较早期的分布式文件系统中, 维护元数据缓存一致性的方案是采用租约 (Leases)[16]机制, 例如 Ceph[8]、Lustre[10]、NFSv4[17]、LocoFS[18]和 IndexFS[19]. 在租约机制中, 当客户端缓存元数据时, 需要向元数据节点申请一定时间的租约. 租约期限可以通过续约请求延长. 当某个客户端执行元数据修改操作时, 元数据节点会向其他持有租约的客户端广播缓存失效消息. 待其他客户端的缓存失效后, 元数据节点才能够继续执行当前操作. 由于每个客户端都会缓存靠近根节点的目录, 当客户端数量增加时, 对这些目录的续约请求将导致元数据节点之间的负载不均衡, 最终影响分布式文件系统元数据操作的整体性能. 这一点在 InfiniFS 工作[7]中已通过实验得到验证.

鉴于租约机制的局限性, InfiniFS[7]提出了乐观访问元数据缓存, 通过记录目录树修改的操作历史来维护元数据缓存的一致性, 其基本架构如图 1 所示. 客户端在执行目录重命名 (rename) 和目录权限修改 (chmod) 操作时, 需要将这些操作通过中心化的协调节点依次广播到所有的元数据节点, 并将操作参数写入元数据节点的失效列表中. 客户端执行路径解析时, 先从本地缓存中读取各级目录的元数据, 再将路径发送至任意元数据节点. 该元数据节点会遍历本地的失效列表, 检查路径中目录的元数据是否发生修改. 如果发生修改, 说明客户端的元数据缓存已失效, 需要通过元数据节点读取元数据重新进行路径解析, 并更新元数据缓存.

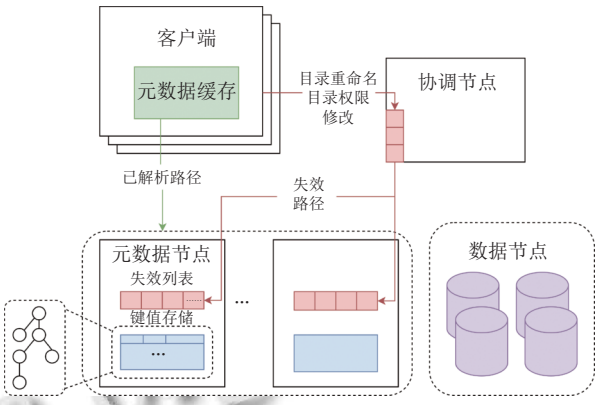

图 1 InfiniFS 的系统架构

和传统的租约机制相比, InfiniFS 中, 客户端无需定期向元数据节点发送续约请求以维持本地元数据缓存的有效性, 降低了元数据节点的负载. 此外, 执行目录树修改操作时, 客户端只需要将操作写入元数据节点的失效列表, 不需要等待元数据节点通知其他客户端缓存失效, 降低了目录树修改操作的延迟.

和租约机制相比, 乐观访问元数据缓存具有显著优势. 因此, 本文在延续乐观缓存思路的基础上, 解决了其在正确性方面的缺陷, 并有效降低了维护缓存一致性的开销.

2.3 其他相关工作

由于维护元数据缓存一致性的复杂性, 部分分布式文件系统仅提供“最终一致性”保证. 例如, NFSv3[9]中, 元数据缓存条目有着固定的存活周期 (time to live, TTL), 超出存活周期后, 需要从元数据节点重新读取元数据条目来更新缓存. 这种方式避免了维护元数据缓存一致性的性能开销, 但是无法在客户端之间及时同步元数据修改, 不能满足 Spark 等应用对文件系统强一致性语义的需求. 与最终一致性元数据缓存相比, 本文提出的机制保证了客户端执行目录树修改操作后, 其他客户端不会读取到过时的元数据, 满足了应用对强一致性语义的需求.

除元数据缓存外, Duplex[20]提出了在系统中增加权限服务器的方案. 这种方案允许客户端将操作对象的完整路径发送至权限服务器, 由权限服务器读取本地缓存的权限信息完成路径解析, 降低了路径解析的延迟. 但在执行目录树修改时, 客户端需要采用两阶段提交等机制同步更新权限服务器与元数据节点, 引入

了高昂的一致性开销, 导致目录树修改操作的性能较低.

3 问题与挑战

本节将通过分析 InfiniFS 存在的问题, 说明通过目录树修改历史来维护元数据缓存一致性所面临的挑战.

挑战 1: 如何降低维护元数据缓存一致性的性能开销. 为了记录目录树修改历史, InfiniFS 为目录树修改操作引入了较高的性能开销. InfiniFS 中, 客户端将目录重命名操作和目录权限修改操作委托给协调节点执行. 协调节点将这些操作广播至所有的元数据节点. 待广播完成后, 协调节点再将操作所对应的元数据修改发送至元数据节点, 以完成元数据操作. 为了确保每个元数据节点的失效列表中, 记录的操作顺序是相同的, 从而使得客户端能够通过任意元数据节点都能检查客户端的元数据缓存是否已经失效. 协调节点需要串行地将目录重命名和权限修改操作广播至元数据节点. 这导致协调节点成为整个分布式文件系统的性能瓶颈. 当多个客户端并发执行目录树修改操作时, 会在协调节点发生排队, 导致元数据操作的延迟升高. 为了验证这一问题, 本文在采用 InfiniFS 方案的分布式文件系统中, 测量了不同数量的客户端并发执行目录重命名操作时, 完成目录重命名操作的延迟, 并和关闭缓存时的延迟进行了对比, 结果如图 2 所示. 实验中各个客户端执行的目录重命名操作相互独立, 元数据修改不存在冲突, 可以并发执行. 从图中可以看出, 随着客户端数量的增加, InfiniFS 中目录重命名操作的延迟持续升高, 这正是由协调节点串行向元数据节点广播所导致的.

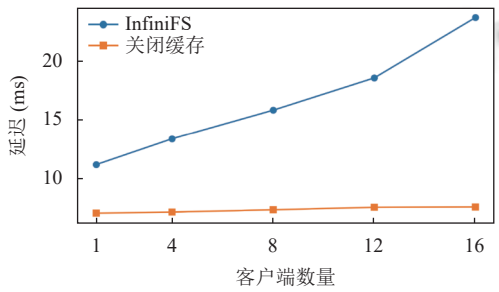

图 2 增加客户端数量时, InfiniFS 执行目录重命名操作的延迟

挑战 2: 如何正确地维护元数据缓存的一致性. InfiniFS 没有通过协调节点将目录删除操作 (rmdir) 广播至元数据节点, 未能正确地同步全部目录元数据修改. 例如, 当多个客户端并发访问分布式文件系统时, 若某一客户端先删除目录再重新创建同名目录 (mkdir) 时, 将导致正确性问题. 如图 3 所示, 客户端 1 和客户端 2 同时缓存了目录 A 的元数据, 客户端 1 删除了目录 A 并重新创建. 随后, 客户端 2 使用本地缓存中旧目录 A 标识符进行访问, 认为目录 A 不存在; 这显然与文件系统中的实际情况不符. 一旦旧目录 A 的标识符被文件系统回收利用, 客户端 2 甚至有可能将其他目录当作目录 A 来访问, 导致更加严重的错误. 通过协调节点对目录删除操作进行广播, 虽然可以解决上述问题, 但是将进一步加剧协调节点的性能瓶颈, 同时导致失效列表中的操作数量增多, 元数据节点读取失效列表的性能下降.

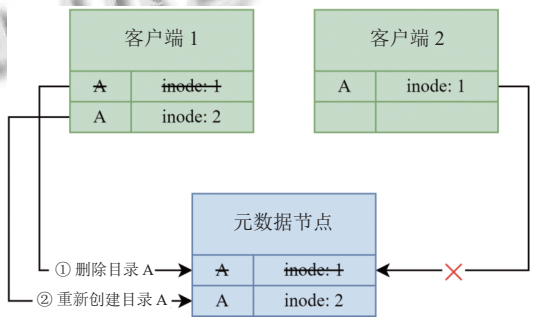

图 3 InfiniFS 未能正确同步目录删除操作

4 系统设计

面对如何降低性能开销和如何正确地维护一致性两方面挑战, 本文的目标是在采用以尽可能低的性能开销, 正确地维护分布式文件系统元数据缓存的一致性. 因此本文提出以下设计.

4.1 基于并发广播机制的目录树修改

针对协调节点串行广播目录树修改操作导致的性能瓶颈, 本文提出了基于并发广播机制的目录树修改. 其基本思路是, 协调节点维护文件系统目录树的版本号, 每当客户端向其发送目录树修改操作时递增, 目录树修改操作发生的顺序可根据操作对应的版本号确定. 协调节点仅需负责为目录树修改操作分发版本号, 由客户端将元数据修改操作和对应的版本号并发广播至元数据节点.

系统执行修改目录树的元数据操作时, 维护元数据缓存一致性的流程如图 4 所示: (1) 客户端向协调节点发送请求, 协调节点版本号递增, 返回版本号给客户端; (2) 客户端向所有元数据节点广播当前操作 (包括版本号和路径参数), 元数据节点将收到的操作写入失效列表, 并按照版本号排序. 不同于 InfiniFS 中广播只

能由协调节点串行进行, 上述流程中失效列表可以并发写入, 从而提升了元数据操作的并发度, 降低了维护元数据缓存一致性的性能开销.

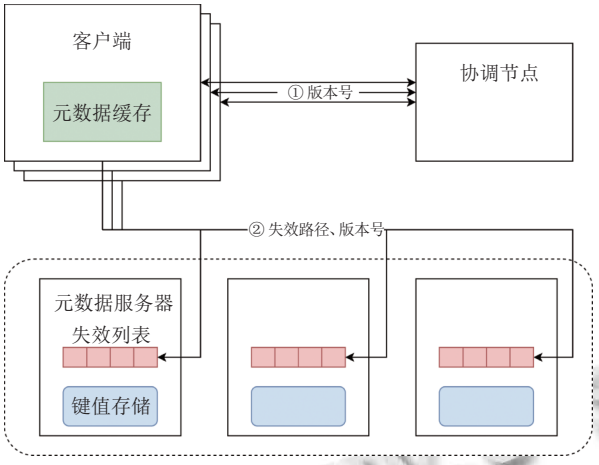

图 4 并发广播目录树修改的流程

由客户端而不是协调节点广播还具有以下优势. 首先, 协调节点自身的硬件资源有限, 发送广播请求的能力有限; 其次, 协调节点广播时需要额外同元数据节点建立连接, 而客户端广播能够复用客户端用于元数据读写的连接.

并发广播机制实现了目录树修改历史的并发记录. 客户端执行路径解析时, 需要根据修改历史同步目录树修改操作, 维护元数据缓存一致性. 为了降低同步目录树修改操作的开销, 客户端需要维护一个缓存版本号, 表示它已经根据这个版本号之前的元数据修改记录更新缓存, 只需要同步这个版本号之后的操作.

具体同步目录流程如下: (1) 客户端通过本地元数据缓存完成路径解析后, 将当前元数据操作的信息 (包括路径和缓存版本号) 发送至任意元数据节点. (2) 该元数据节点会从缓存版本号之后开始遍历本地的失效列表. 如果某个路径参数是当前路径的前缀 (即该操作修改了当前路径上的元数据), 元数据节点为缓存确定一个新版本号并返回失效信息 (包括新版本号和新旧版本号之间的路径). 否则, 元数据节点返回缓存有效. (3) 当元数据节点返回失效信息后, 客户端通过元数据节点重新进行路径解析, 根据失效信息更新缓存及其版本号, 表明它同步了两个版本号之间全部目录树修改操作.

在上述流程中, 新版本号的确定对元数据缓存的一致性至关重要. 并发广播时, 版本号的分配顺序和失效列表的写入顺序可能不完全一致, 导致失效列表中某个版本号暂时空缺. 如果直接将整个失效列表的最大版本号确定为新版本号, 可能会因为未同步空缺版本号对应的操作, 破坏了元数据缓存的一致性. 因此, 元数据节点需要将新版本号确定为版本号连续部分的最大版本号, 而非整个失效列表的最大版本号, 避免因记录空缺导致部分操作未同步, 确保元数据缓存的一致性.

下面举例分析如何确定缓存的新版本号, 图 5 中目录树修改操作 (3, /A/C) 比 (2, /A/B) 先写入失效列表. 如果 t_1 时元数据节点将新版本号 v_{new} 确定为整个失效列表的最大版本号 3, 则 t_2 时刻元数据节点将从版本号 4 开始遍历, 导致客户端未同步 (2, /A/B). 因此, t_1 时元数据节点应当将 v_{new} 确定为失效列表版本号连续部分的最大版本号 1.

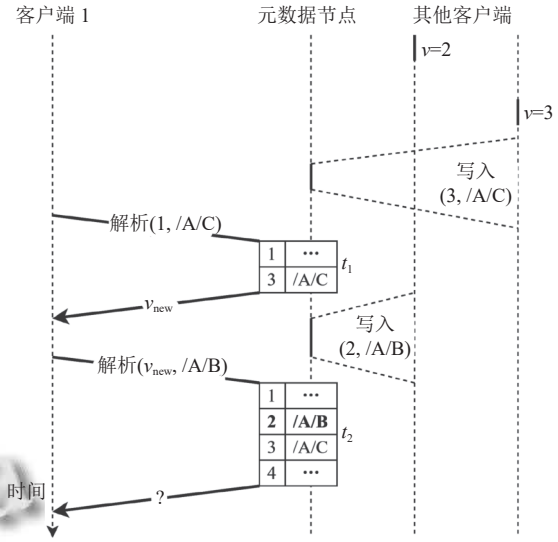

图 5 缓存的新版本号确定过程举例

具体算法如算法 1.

算法 1. 目录树修改操作同步算法

1. 元数据节点收到客户端缓存版本号 v_c, 路径参数 p_c;
2. 元数据节点返回的新版本号 v_{new};
3. 失效列表按照版本号索引并排序;
1) 从 v_c+1 开始, 遍历失效列表中的元素, 对其中的每个元素 (v_i,p_i): 如果 p_i 是 p_c 的前缀, 客户端缓存失效, 执行 2). 如果不存在 p_i 是 p_c 的前缀, 客户端缓存有效;
2) 从 v_c+1 开始, 找到失效列表版本号连续部分的最大版本号 v_{new}, 将 v_c+1 和 v_{new} 之间的元素返回给客户端.

为了防止元数据节点的失效列表不断增长, 占用过多的系统资源, 元数据节点需要为失效列表长度设置上限. 到达上限时, 元数据节点截断失效列表, 版本

号更低的客户端失效元数据缓存.

4.2 基于惰性广播和墓碑机制的目录删除

对于 InfiniFS 中目录树修改同步存在的问题, 一种直接的解决方案是通过协调节点对目录删除操作也进行广播. 但是, 由于实际应用负载中目录删除操作远多于目录重命名和目录权限修改, 这种方案将进一步加剧协调节点的性能瓶颈. 同时, 这会导致元数据节点中失效列表中的操作数量增多, 路径解析时读取失效列表的性能下降, 从而影响各类文件系统操作的性能. 为了以尽可能低的性能开销, 正确同步所有修改目录树的元数据操作, 本文提出了基于惰性广播和墓碑机制的目录删除.

墓碑机制的基本思路是, 当客户端执行目录删除操作时, 如图 6 中的①所示向元数据节点的键值存储中写入一个类型为墓碑 (tombstone) 的元数据条目, 表示该目录已被客户端删除. 墓碑条目的编码方式和普通条目一致, 以被删除目录的父目录标识符和名称作为键, 但其类型为墓碑, 其余项为空. 通过墓碑条目, 可以在客户端之间同步目录删除和创建操作.

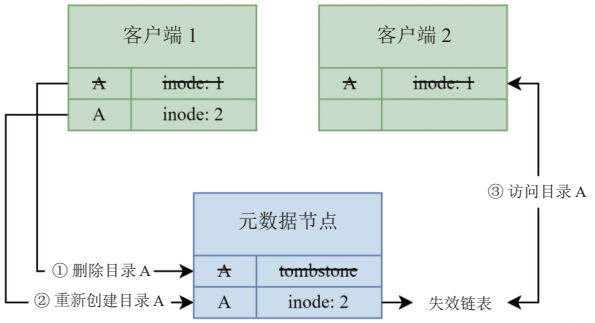

图 6 目录删除墓碑的工作原理

惰性广播指的是, 只有当客户端在相同路径下创建相同名称的新目录时 (如图 6 中的②所示), 才需要采用第 4.1 节中的并发广播机制, 将当前路径写入元数据节点的失效列表, 随后删除墓碑并创建新目录. 这样一来, 当其他客户端使用已被删除的旧目录的元数据缓存进行路径解析时, 元数据节点能够通过失效列表发现缓存已经失效, 客户端能够用新目录的元数据替换旧目录的元数据, 以便之后能够正常访问新目录.

还需要考虑其他元数据操作如何处理目录删除墓碑. 执行其他元数据操作时, 若客户端读取到了类型为墓碑的元数据条目, 则说明该目录已经被删除, 客户端只需按照目录不存在时的流程继续执行元数据操作. 若该目录元数据仍然在客户端的元数据缓存中, 则需从缓存中删除.

在目录删除墓碑机制中, 若没有客户端在删除目录后重新创建, 则无需对目录删除操作进行广播. 与客户端在删除目录后直接进行广播相比, 这种方式能够按需同步目录树修改, 避免不必要的广播, 从而进一步缓解降低协调节点负载. 为了防止目录删除墓碑在元数据节点积累, 元数据节点需在截断失效列表时清除墓碑.

5 性能评估

5.1 实验配置

首先介绍实验的硬件配置. 实验中设置 3 台服务器作为元数据节点, 每台服务器配备了 2 个 12 核 2.2 GHz 的 Intel Xeon E5-2650 CPU 和 64 GB 内存. 此外, 实验中使用一台性能更强的服务器作为客户端节点, 该机器配备了 4 个 20 核 2.5 GHz 的 Intel Xeon Gold 6248 CPU. 机器之间采用 10 GB/s 的以太网互联. 由于实验评估对象为元数据缓存设计, 实验中不涉及实际文件读写.

接下来介绍实验的软件配置. 我们使用 TiKV 8.1[21] 作为元数据节点, Intel DAOS[22]作为数据节点, 实现了一套分布式文件系统. 在此基础上, 本研究分别实现了: (1) InfiniFS 元数据缓存设计, 并在其中加入目录删除广播以保证正确性, 以下简称 InfiniFS; (2) 第 3 节所述的元数据缓存设计, 以下简称本系统. TiKV 服务器部署在 3 个元数据节点, 采用 RAMDisk 存储元数据. 多个文件系统客户端和协调节点同时部署在客户端节点.

5.2 整体性能

为了评估 InfiniFS 和本系统中各项元数据操作的整体性能, 本研究采用 IOR mdtest 基准测试, 测量了系统中常见目录树修改操作的性能. 实验中设置目录平均深度为 10 层, 与文献[7]保持一致.

首先, 本研究采用单个客户端执行元数据操作, 测量了操作的延迟, 实验结果如图 7 所示. 实验结果表明, 与 InfiniFS 相比, 本系统中目录重命名 (rename) 操作的延迟下降了 65.8%. 这是因为 InfiniFS 中将目录重命名操作委托给协调节点执行, 协调节点需要从元数据节点逐层读取元数据进行路径解析. 而本系统中, 客户端直接执行目录重命名操作, 路径解析时可以读取本地的元数据缓存. 本系统中目录删除 (rmdir) 操作的延迟下降了 66.9%, 这是因为采用墓碑机制后, 客户端无

需立即向元数据节点广播目录删除, 能够进一步降低目录删除操作的延迟. 本系统中目录创建 (mkdir) 操作的延迟和 InfiniFS 持平, 这说明墓碑机制对目录创建操作的延迟几乎没有影响. 这是因为在执行目录创建操作时, 文件系统本就需要判断同名目录是否已经存在, 检查是否存在同名墓碑并不需要额外的开销.

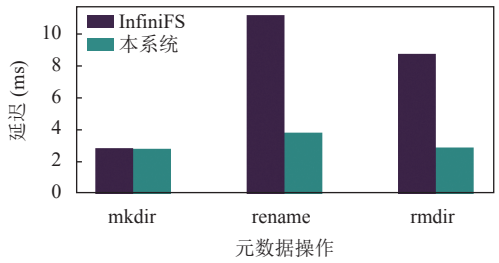

图 7 分布式文件系统中目录树修改操作的延迟

随后, 本研究采用 16 个客户端并发执行元数据操作, 测量了目录树修改操作的吞吐量, 实验结果如图 8 所示.

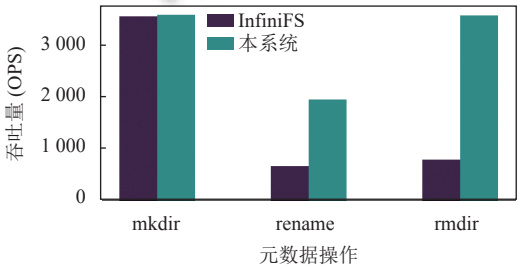

图 8 分布式文件系统中目录树修改操作的吞吐量

实验结果表明, 本系统中目录重命名操作的吞吐量是 InfiniFS 的 2.94 倍. 这是因为在 InfiniFS 中目录重命名操作都需要由协调节点串行广播到各个元数据节点, 协调节点限制了分布式文件系统处理目录重命名操作的吞吐量. 而本系统中, 客户端在通过协调节点获取版本号之后, 可以并发地广播到各个元数据节点. 本系统中目录删除操作的吞吐量是 InfiniFS 的 4.53 倍, 这是由于采用墓碑机制之后, 客户端惰性向元数据节点广播目录删除, 能够提升目录删除操作的吞吐量. 本系统中目录创建操作的吞吐量和 InfiniFS 持平, 这是因为如前所述, 执行目录创建操作并不需要额外的开销.

5.3 可扩展性

为了评估 InfiniFS 和本系统处理元数据操作的可扩展性, 本研究采用 IOR mdtest 基准测试测量了不同客户端并发执行元数据操作时, 系统处理目录重命名和目录删除操作的吞吐量.

图 9 展示了不同客户端数量时, 系统处理目录重命名操作的吞吐量. 实验结果表明, 当客户端数量从 12 个增加到 16 个时, InfiniFS 的吞吐量仅升高 8.26%. 并且 16 个客户端时的吞吐量只有 4 个客户端时的 2.22 倍. 这是因为 InfiniFS 中协调节点串行广播目录重命名操作到各个元数据节点, 导致协调节点成为性能瓶颈, 无法处理更多客户端的目录重命名操作. 而本系统在客户端由 1 个增加到 16 个时, 吞吐量达到了 7.55 倍. 当客户端由 1 个增加到 4 个时, 实现了接近线性的提升. 协调节点分发版本号不会限制整个分布式文件系统的可扩展性. 随后由于失效列表并发写入存在冲突, 以及元数据节点处理目录重命名需要采用分布式事务, 导致吞吐量增长放缓, 但仍然维持了稳定的增长趋势.

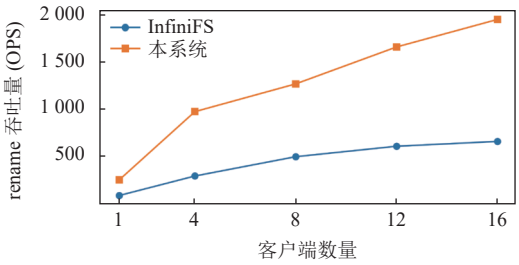

图 9 不同客户端数量时, 目录重命名操作的吞吐量

图 10 展示了不同客户端数量时, 系统处理目录删除操作的吞吐量.

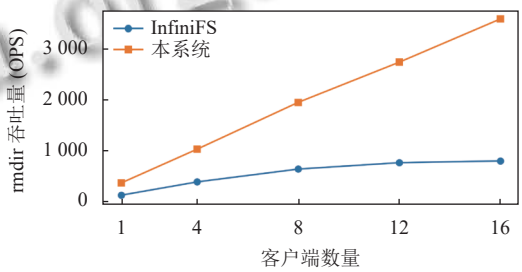

图 10 不同客户端数量时, 目录删除操作的吞吐量

实验结果表明, 当客户端数量从 12 个增加到 16 个时, InfiniFS 的吞吐量仅升高 4.56%. 并且 16 个客户端时的吞吐量只有 4 个客户端时的 2.09 倍. 这是因为, 如前所述, InfiniFS 中协调节点串行广播目录删除操作到各个元数据节点, 导致协调节点成为性能瓶颈, 无法处理更多客户端的目录删除操作. 而本系统在客户端由 1 个增加到 16 个时, 吞吐量达到了 10.0 倍, 并且维持了吞吐量的稳定增长. 这是因为本系统中目录删除

操作只涉及元数据条目的读写, TiKV 这样的分布式数据库保证了元数据节点良好的可扩展性.

5.4 真实应用

Spark[11]是大数据分析领域广泛使用的分布式计算引擎. 为了分析真实应用负载在 InfiniFS 和本系统上的运行时的性能表现, 本文实现了 Spark 作业 (job) 的提交协议, 测试了不同参数下, InfiniFS 和本系统中 Spark 作业提交的延迟.

Spark 作业提交协议包含一系列元数据操作, 主要用于为 Spark 作业结果的写入提供事务保证. 每个 Spark 作业由一个 driver 进程分发给若干 executor 进程, 由于 executor 进程可能发生故障, 为了保证作业结果写入的原子性和持久性, Spark 采用以下流程完成任务提交: (1) 每个 executor 并发执行, 将自己负责任务的结果写在临时目录下, 提交时将临时目录重命名; (2) driver 将每个 executor 的结果重命名到指定位置; (3) driver 将空目录删除, 并创建“_SUCCESS”文件, 表示作业成功提交.

图 11 展示了不同 executor 数量时, 相同任务量的 Spark 作业提交所需的延迟. 当 executor 数量为 4、8 和 16 时, 本系统中任务提交延迟与 InfiniFS 相比分别下降 40.0%、41.1% 和 43.6%. 这是因为随着 executor 数量的增加, 提交 Spark 作业时, 第 (1) 步时并发执行目录重命名操作的客户端随之增多, 第 (3) 步时 driver 所需要执行的目录删除操作也随之增多. 而本系统在并发执行元数据操作的客户端数量增多时, 相较于 InfiniFS 展现出更加显著的性能优势.

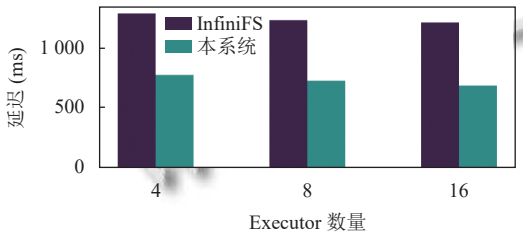

图 11 不同 executor 数量时, Spark 作业提交的延迟

5.5 消融实验

为了分析本系统两大设计要点对分布式文件系统元数据操作性能提升产生的贡献, 本研究首先在系统中增加了基于并发广播机制的目录删除 (此时的系统称为“+并发广播”), 随后在此基础上增加了基于惰性广播和墓碑机制的目录删除 (最终的系统称为“+目录墓碑”) . 使用 IOR mdtest 基准测试测量了 InfiniFS、“+并发广播”和“+录墓碑”这 3 个系统中目录删除操作的吞吐量. 实验中设置 16 个客户端.

表 1 展示了逐步增加系统优化后, 客户端并发执行目录删除操作的吞吐量. 增加基于并发广播机制的目录删除后, 分布式文件系统处理目录删除操作的吞吐量是 InfiniFS 的 2.43 倍, 延迟下降 55.6%. 在此基础上增加基于惰性广播和墓碑机制的目录删除后, 进一步提升到了 InfiniFS 的 4.53 倍, 延迟进一步下降 46.1%. 两个设计要点对吞吐量提升的贡献占比分别为 40.4% 和 59.6%. 这说明在增加目录树修改并发广播后, 失效列表的并发写入性能仍然在制约文件系统中目录删除操作的性能, 需要采用目录删除墓碑进一步优化.

表 1 增加系统优化后, 目录删除操作的延迟和吞吐量

系统	延迟 (ms)	吞吐量 (OPS)
InfiniFS	18.785	787
+并发广播	8.340	1911
+目录墓碑	4.495	3567

6 总结与展望

现有的维护元数据缓存一致性的方案引入了较高的性能开销, 却仍然存在正确性缺陷, 从而无法完全保证元数据缓存的一致性. 针对这些问题, 本文提出了一种维护元数据缓存一致性的轻量级机制, 包括: (1) 基于并发广播机制的目录树修改方法; (2) 基于惰性广播和墓碑机制的目录删除方法, 既突破了现有方案的性能瓶颈又解决了正确性缺陷.

本文采用 TiKV 作为元数据节点, 实现了分布式文件系统中的各类操作. 由于部署 TiKV 需要专门的 PD (placement driver) 节点, 若将维护分布式文件系统元数据缓存所需的协调节点相关功能实现在 PD 节点中, 可以进一步简化系统的部署, 这将是系统建设的下一步方向.

参考文献

1 Ghemawat S, Gobioff H, Leung ST. The Google file system. Proceedings of the 19th ACM Symposium on Operating Systems Principles. Bolton Landing: ACM, 2003. 29–43. [doi: 10.1145/945445.945450]

2 Pan S, Stavrinos T, Zhang YQ, *et al*. Facebook’s tectonic filesystem: Efficiency from exascale. Proceedings of the 19th USENIX Conference on File and Storage Technologies. USENIX Association, 2021. 217–231.

3 Calder B, Wang J, Ogus A, *et al*. Windows Azure storage: A

highly available cloud storage service with strong consistency. Proceedings of the 23rd ACM Symposium on Operating Systems Principles. Cascais: ACM, 2011. 143–157. [doi: 10.1145/2043556.2043571]

4 Li Q, Xiang Q, Wang YX, *et al*. More than capacity: Performance-oriented evolution of Pangu in Alibaba. Proceedings of the 21st USENIX Conference on File and Storage Technologies. Santa Clara: USENIX Association, 2023. 21.

5 An W, Bi X, Chen GT, *et al*. Fire-flyer AI-HPC: A cost-effective software-hardware co-design for deep learning. Proceedings of the 2024 International Conference for High Performance Computing, Networking, Storage and Analysis. Atlanta: IEEE, 2024. 1–23. [doi: 10.1109/SC41406.2024.00089]

6 Harter T, Borthakur D, Dong SY, *et al*. Analysis of HDFS under HBase: A facebook messages case study. Proceedings of the 12th USENIX Conference on File and Storage Technologies. Santa Clara: USENIX Association, 2014. 199–212.

7 Lv WH, Lu YY, Zhang YM, *et al*. InfiniFS: An efficient metadata service for large-scale distributed filesystems. Proceedings of the 20th USENIX Conference on File and Storage Technologies. Santa Clara: USENIX Association, 2022. 313–328.

8 Weil SA, Brandt SA, Miller EL, *et al*. Ceph: A scalable, high-performance distributed file system. Proceedings of the 7th Symposium on Operating Systems Design and Implementation. Seattle: USENIX Association, 2006. 307–320.

9 Pawlowski B, Juszczak C, Staubach P, *et al*. NFS version 3: Design and implementation. Proceedings of the USENIX Summer 1994 Technical Conference. Boston: USENIX Association, 1994. 137–152.

10 Qian YJ, Li X, Ihara S, *et al*. LPCC: Hierarchical persistent client caching for lustre. Proceedings of the 2019 International Conference for High Performance Computing, Networking, Storage and Analysis. Denver: ACM, 2019. 88. [doi: 10.1145/3295500.3356139]

11 Zaharia M, Chowdhury M, Das T, *et al*. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. San Jose: USENIX Association, 2012. 15–28.

12 Shvachko K, Kuang HR, Radia S, *et al*. The hadoop distributed file system. Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies. Incline Village: IEEE, 2010. 1–10. [doi: 10.1109/MSST.2010.5496972]

13 Niazi S, Ismail M, Haridi S, *et al*. HopsFS: Scaling hierarchical file system metadata using NewSQL databases. Proceedings of the 15th USENIX Conference on File and Storage Technologies. Santa Clara: USENIX Association, 2017. 89–103.

14 Thomson A, Abadi DJ. CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems. Proceedings of the 13th USENIX Conference on File and Storage Technologies. Santa Clara: USENIX Association, 2015. 1–14.

15 Wang YD, Wu YF, Li C, *et al*. CFS: Scaling metadata service for distributed file system via pruned scope of critical sections. Proceedings of the 18th European Conference on Computer Systems. Rome: ACM, 2023. 331–346. [doi: 10.1145/3552326.3587443]

16 Gray C, Cheriton D. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. Proceedings of the 12h ACM Symposium on Operating Systems Principles. The Wigwam, Litchfield Park: ACM, 1989. 202–210. [doi: 10.1145/74850.74870]

17 Pawlowski B, Noveck D, Robinson D, *et al*. The NFS version 4 protocol. Proceedings of the 2nd International System Administration and Networking Conference. 2000.

18 Li SY, Lu YY, Shu JW, *et al*. LocoFS: A loosely-coupled metadata service for distributed file systems. Proceedings of the 2017 International Conference for High Performance Computing, Networking, Storage and Analysis. Denver: ACM, 2017. 4. [doi: 10.1145/3126908.3126928]

19 Ren K, Zheng Q, Patil S, *et al*. IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion. Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis. New Orleans: IEEE, 2014. 237–248. [doi: 10.1109/SC.2014.25]

20 Dong C, Wang F, Yang YX, *et al*. Low-latency and scalable full-path indexing metadata service for distributed file systems. Proceedings of the 41st IEEE International Conference on Computer Design. Washington: IEEE, 2023. 283–290. [doi: 10.1109/ICCD58817.2023.00051]

21 Huang DX, Liu Q, Cui Q, *et al*. TiDB: A raft-based HTAP database. Proceedings of the VLDB Endowment, 2020, 13(12): 3072–3084. [doi: 10.14778/3415478.3415535]

22 Liang Z, Lombardi J, Chaarawi M, *et al*. DAOS: A scale-out high performance storage stack for storage class memory. Proceedings of the 6th Asian Conference on Supercomputing Frontiers. Singapore: Springer, 2020. 40–54. [doi: 10.1007/978-3-030-48842-0_3]

(校对责编: 张重毅)