支持 EB 级存储的元数据服务器集群系统

刘健^{①②*}, 张军伟^①, 邵冰清^①, 董欢庆^①, 刘振军^①, 许鲁^{①*}

① 中国科学院计算技术研究所, 北京 100190

② 中国科学院大学, 北京 100049

* 通信作者. E-mail: liujian@bwestor.com.cn, xulu@bwestor.com.cn

收稿日期: 2014-11-05; 接受日期: 2015-04-06; 网络出版日期: 2015-05-20

中国科学院战略性先导科技专项 (批准号: XDA06010401) 资助

摘要 随着全球数据总量的快速增长, PB 级存储系统已无法满足未来应用对其数据容量的需求, 通用、高扩展且易于部署的 EB 级存储系统的研发势在必行, 其核心技术为元数据服务器集群技术. 本文介绍了在 pNFS 和蓝鲸块设备文件系统 EXFS 基础上研发的支持 EB 级存储的蓝鲸元数据服务器集群技术. 首先分析了支持 EB 级与 PB 级存储的元数据服务器集群系统的区别; 然后介绍了蓝鲸元数据服务器集群原型系统的整体架构及其全局命名空间的分布策略; 之后介绍了基于分布式日志的一致性元数据原子操作协议, 和细粒度、低延时的元数据负载迁移两项关键技术; 最后进行了测试. 结果显示, 集群系统中每台元数据服务器均可提供超过 10000 OPS 的元数据操作支持.

关键词 EB 级 存储 元数据服务器集群 全局命名空间 一致性 负载迁移

1 引言

2014 年 4 月 IDC 发布的白皮书 *The Digital Universe of Opportunities*^[1] 显示, 2013 年全球新增数据总量 4.4 ZB, 并以每年 40% 的速度增长, 预计到 2020 年全球新增数据总量将达到 44 ZB. 现有的企业级存储系统面临极大挑战, PB 级的存储容量无法满足未来应用对存储系统的需求, 研究通用且易于部署的 EB 级存储系统势在必行.

在当前主流的 PB 级存储系统中, 主要采用单一元数据服务器 (metadata server, MDS) 和 MDS 集群两种架构. GFS^[2], HDFS^[3], GPFS-SNC^[4], Lustre^[5], pNFS^[6] 和 BWFS^[7] 等都采用单一 MDS 架构. 这种架构的特点是元数据与数据访问分离, 元数据访问通过专门的 MDS 实现, 数据访问则由客户端直接访问数据服务器或数据存储设备实现. 这种结构能有效减轻 MDS 的访问压力, 通过提供并发数据访问来提升系统聚合 I/O 带宽, 通过横向扩展技术增大系统存储容量^[8]. 随着云计算和大数据应用对底层存储需求的爆炸式增长, 文件数量成倍增加, 元数据规模也随之急速增长. 在大规模客户端对共享分布式文件系统进行高强度并发访问的情况下, 单一 MDS 架构成为整个系统扩展的瓶颈, 无法应对更大规模数据存储需求的挑战.

MDS 集群架构可以通过多个 MDS 并发提供元数据服务提高系统元数据访问能力, 一般采用共享元数据存储的方式, 其组成部分主要包括客户端、并发提供元数据服务的多个元数据服务器、共享

引用格式: 刘健, 张军伟, 邵冰清, 等. 支持 EB 级存储的元数据服务器集群系统. 中国科学: 信息科学, 2015, 45: 721–738, doi: 10.1360/N112014-00330

的存储池以及连接网络. 业内众多顶尖企业和研究机构都将这种架构作为解决性能瓶颈问题的首选方案进行研究和开发. Ceph^[9], GFS II^[10], Ursa Minor^[11], Farsite^[12], IBM Storage Tank^[13] 和 Map-R^[14] 等都是采用的 MDS 集群架构, 同时, Lustre^[15] 的 2.0 版也在进行 MDS 集群架构的开发.

对以上系统架构的研究和分析表明, 在 EB 级的存储系统中, 单台 MDS 无法满足系统对元数据操作的性能需求, 迫切需要采用 MDS 集群架构. 与支持 PB 级存储的单 MDS 系统相比, EB 级的 MDS 集群系统具有其自身特点, 主要包括: (1) 更高的元数据存储容量; (2) 更高的单位时间元数据操作处理数量; (3) 更高的分布式元数据操作处理复杂度; (4) 更高的集群负载管理复杂度等.

针对 EB 级 MDS 集群系统特点, 中国科学院计算技术研究所并行网络文件系统协议 pNFS 和自主研发的蓝鲸块设备文件系统 EXFS^[16] 的基础上, 开发了支持 EB 级存储的 MDS 集群原型系统 BWMMs. 测试结果显示, BWMMs 支持的存储系统数据容量可超过 EB, 索引节点数达到 100 G, 支持文件数量达到 1000 亿, 每台 MDS 每秒元数据操作数超过 10000 OPS, 具备支持 EB 级存储系统对元数据操作性能需求的能力.

本文主要介绍 BWMMs 三个方面的创新工作. 首先介绍了 BWMMs 的整体架构和全局命名空间, 提出一种以目录为粒度的元数据分布策略, 兼顾系统分布性和局部性, 在降低系统管理复杂度的同时, 保持进程访问的顺序性; 然后, 在分布式元数据操作一致性方面, 提出一种基于分布式日志的一致性原子操作协议, 执行过程不需磁盘同步写, 对正常操作延时影响较小, 在宕机重启后, 只需很短时间即可恢复 MDS 集群的元数据一致性; 最后, 在 MDS 负载均衡方面, 提出一种细粒度、低延时的负载迁移机制, 在迁移过程中, 通过判断操作的类型和状态, 选择由迁出端或迁入端进行响应, 无需阻塞用户访问.

2 整体架构与全局命名空间

BWMMs 基于 pNFS 和 EXFS 构建, pNFS 已经进入 Linux 内核并开放源码, 受到业内的广泛认可和支持; EXFS 是 BWFS 中基于块设备的日志文件系统, 性能可靠. 在 BWMMs 中, pNFS 负责客户端与 MDS 集群的交互以及 MDS 之间的交互; BWMMs 利用了 pNFS 的状态服务、协议交互等机制, 在此基础上, 在客户端增加了访问定位 MDS 的机制, 以使得客户端在访问元数据时能够与正确的 MDS 进行交互. EXFS 负责管理 MDS 本地元数据和数据资源, 并提供元数据服务, 在此基础上, 增加了 MDS 之间的元数据关联关系的维护以及保证宕机一致性的分布式日志机制.

2.1 EXFS

EXFS 是以 EXT3 为雏形, 经过改进和添加功能而实现的一个完整的分布式块设备文件系统, 主要用于处理 BWFS 对元数据的创建、释放、查询等管理操作. EXFS 磁盘布局类似 EXT3, 不同点是 EXFS 采用元数据与数据分离存储的方式. 每个 EXFS 文件系统是由一个元数据子卷和多个数据卷组成的卷组, 它可以实时添加数据卷. 元数据子卷存储超级块、索引节点、目录项等元数据, 而数据卷则用来存放文件数据. 采用这种存储分离结构和元数据子卷管理方式, 一方面实现了对 pNFS 的带外访问方式的支持, 另一方面提升了 EXFS 文件系统数据容量的扩展性. EXFS 还支持文件条带存储, 即一个大文件的数据可以分散存储到多个独立的数据卷设备上, 从而能够提升大文件的聚合访问性能.

EXFS 实现了与 EXT3 类似的日志技术以保证宕机一致性, 元数据更新记录到系统日志, 其大小可在格式化时设置. 上层应用的元数据更新写入页缓存即返回成功. 然后通过内核调用日志块设备层

图 1 蓝鲸 MDS 集群整体架构图
Figure 1 Architecture of Blue Whale MDS cluster

的 `kjournald` 线程异步地记录缓冲区数据到日志. 最后通过异步下刷线程将页缓存中的脏元数据写入相应的元数据磁盘块. 相对于需要进行大量随机写入的同步写方式, 顺序写入日志寻道时间将大为减小.

2.2 整体架构

BWMMS 的整体架构如图 1 所示, 主要分为 3 个模块, 包括客户端 `nfs` 模块、MDS 的 `nfsd` 模块和 `EXFS` 模块. 每个 MDS 管理一个或者多个 `EXFS` 卷组, 所有 MDS 通过一个确定的根 MDS 导出全局命名空间. 客户端访问时, 系统透明地将元数据分布到不同 MDS 的某个元数据子卷上, 因此, 可由多个 MDS 并发提供元数据服务来提升元数据访问聚合带宽, 并且系统可以通过调整某些 MDS 的元数据子卷管理域实现负载均衡. 共享数据存储池是 BWRAID^[17] 通过 iSCSI^[18] 协议导出的若干逻

表 1 MDS_Map 结构字段
Table 1 Field description of MDS_Map structure

Field name	Field description
Volume_count	The amount of subvolumes
Volume_high_seq	Highest seq.no of subvolume
Mds_table_version	Version for updating mds_table
Vol_entry[]	Subvolume info including addr and state

表 2 Vol_entry 结构字段
Table 2 Field description of vol_entry structure

Field name	Field description
Vol.no	Index of subvolumes
Status	UP, DOWN or MIGRATING
Ipaddr	IP address of MDS
Path	Exported directory of the subvolumes
Rpc_client	Pointer of communication structure for ipaddr

图 2 Ino 结构图
Figure 2 Struct of ino

辑卷, BWRAID 是一种网络 RAID, 比普通 RAID 具有更高的存储空间利用率和高可靠性等优点. 当然, BWMMs 采用其他系统通过 iSCSI 导出的逻辑卷也可以正常运行.

2.3 全局命名空间

根 MDS 在内存中维护一个 MDS_Map 结构, 存储元数据子卷与 MDS 映射关系, 其他 MDS 和客户端在加入系统时从根 MDS 获取一个副本. MDS_Map 结构的主要字段及相关说明如表 1 所示.

每个元数据子卷格式化时指定一个子卷号, 写入超级块中, 作为其在 MDS_Map 结构的 vol_entry 数组索引, 用以快速查找到元数据子卷信息. 每个 vol_entry 结构包含单个元数据子卷所有的信息, 如管理该元数据子卷的 MDS 地址及可用状态等, 该结构的主要字段及相关说明如表 2 所示.

在 BWMMs 中, 索引节点由 ino 标识, 如图 2 所示, ino 前半部分为元数据子卷号, 后半部分为子卷内索引号. 客户端挂载根 MDS 导出的根目录, 首先在 1 号元数据子卷上进行目录或者文件的创建. 当根 MDS 启动某种以目录为粒度的元数据分布策略 (如轮转、哈希等) 后, 客户端在新创建目录时, 其 ino 中元数据子卷号被设置为其他元数据子卷的编号, 其索引节点转而由元数据子卷编号所在 MDS 管理. 客户端根据 ino 的元数据子卷编号查找 MDS_Map 以获取相应 MDS 的地址, 并访问相应的 MDS. 这样, 随后的元数据服务负载被分流到其他的 MDS 和元数据子卷上, 而命名空间中的目录对象也随之分散.

图 3 目录分区方法示意图

Figure 3 Example of directory partition method

2.4 以目录为粒度的元数据分布式策略

元数据的分布问题, 本质上是分布性与局部性的权衡. Linux 文件系统的命名空间结构和元数据操作都具有明显的局部性, 进程的访问模式具有明显的顺序性, 因此局部性应当放在首要考虑的位置. 对于子树的分布粒度, 过于灵活的设置既增加了系统的复杂性和实现难度, 同时也不容易建立明确的算法模型.

BWMMS 对全局命名空间进行分区, 为每个分区规定存储目录的总数, 一旦分区被目录填满, 则创建新目录将进行分布处理. 在目录树中, 目录文件是内部节点, 而普通文件只是叶子节点. 因此, 在对文件系统目录层次进行分布时, 优先考虑目录的分布. 如果将文件与目录分开, 在负载不重的状况下, 会增大访问文件后更新父目录的操作延迟, 不利于进程的顺序访问. 文件系统中目录文件的个数远小于文件个数, 分区数据结构不会占用过多的内存, 分区的计算开销也不会过多影响文件的创建性能. 对一个分区内的所有目录, 其树形结构的生长方式不受任何限制, 这样就更容易适应不同的应用, 并且一旦需要对分区算法做出调整时, 调整的变量只有一个.

如图 3 所示, 当一个目录分区分配满以后, 该分区的新建子目录都将分布到其他元数据子卷上, 现阶段采用 Round-robin 方式, 选择循环分配的第一个非本地元数据子卷, 以此将元数据分散到各个 MDS 上. 这样既做到了分布, 又做到了定位与路径无关. 当有新的元数据子卷和 MDS 注册到系统中, 各个 MDS 会更新 MDS_Map 和元数据子卷个数, 然后使用更新后的信息进行计算, 新建目录就会分流到新添加的元数据子卷和 MDS 中. BWMMS 采用 ino 批量预取机制来加快目录分布操作的速度.

3 基于分布式日志的一致性原子操作协议

在 BWMMS 中, 一些元数据操作涉及多个不同的文件或目录, 如果这些文件或目录分布在不同的 MDS 上, 那么这些操作称之为分布式元数据操作. 分布式元数据操作对分布在各个不同的 MDS 的文件或目录的访问需要保持一致, 在某个分布式元数据操作的部分服务器宕机情况下, 也必须保证这个分布式元数据操作的一致性, 从而避免上层应用访问到不一致的元数据.

针对传统的分布式元数据操作一致性保证协议通常需要多次的网络交互和状态强制写, 对元数据

访问性能影响较大的问题, 提出一种非磁盘同步写的基于分布式日志的原子提交协议 Dual-Log ACP (DL-ACP). 在 DL-ACP 中, 一个分布式元数据操作仅涉及两个 MDS, 接受客户端请求的 MDS 被称为协调者 (Coordinator), 另一个 MDS 被称为参与者 (Participant), 对于涉及更多 MDS 的元数据操作采用异步方式分解为只涉及两个 MDS 的操作^[19]. 协调者执行的子操作被称为子操作 1 (Sub-op1), 参与者执行的子操作被称为子操作 2 (Sub-op2). 在分布式操作执行过程中, 协调者和参与者将各自子操作的元数据更新记录异步地写入各自的本地日志, 同时将各自子操作的重做记录传送到对方, 冗余记录在对方的分布式日志中. 在协调者宕机或者参与者宕机的情况下, 都能够依据对方保存的冗余重做记录对自己的子操作进行重做恢复, 从而保证一致性.

3.1 DL-ACP 协议流程

如图 4 所示, DL-ACP 的具体步骤如下:

(1) 协调者首先对第 1 个子操作的可执行性进行预先检查 (Sub-op1 precheck). 如果检查通过, 投赞成票 (Vote Yes); 如果检查不通过, 直接结束, 分布式元数据操作执行失败.

(2) 协调者在对第 1 个子操作的预先检查通过之后, 协调者向参与者发送操作请求消息, 请求参与者执行第 2 个子操作 (Sub-op2), 请求消息中携带了第 1 个子操作的重做日志. 然后协调者进入等待状态.

(3) 参与者收到协调者的请求消息后, 参与者执行第 2 个子操作 Sub-op2. 如果执行不成功, 向协调者发送 ABORT 消息, 直接结束; 如果执行成功, 参与者做出 COMMIT 决策, 向协调者发送 COMMIT 消息, 消息中携带第 2 个子操作的重做日志. 然后参与者结束操作, 进入异步提交阶段, 异步地向日志文件中写入第 2 个子操作的日志记录, 包括第 1 个子操作的重做日志、第 2 个子操作的操作结果日志以及 COMMIT 标志.

(4) 协调者收到参与者的返回消息后, 如果是 ABORT 消息, 直接结束分布式元数据操作的执行; 如果是 COMMIT 消息, 则先把第 2 个子操作的重做日志记录到第 1 个子操作的日志记录中, 然后执行第 1 个子操作 Sub-op1. 第 1 个子操作 Sub-op1 执行完之后, 协调者结束分布式元数据操作, 将操作结果返回给应用程序, 然后进入异步提交阶段, 异步地将第 1 个子操作的日志记录, 包括第 2 个子操作的重做日志、第 1 个子操作的操作结果日志以及 COMMIT 标志写入日志文件.

(5) 协调者将第 1 个子操作的日志记录持久写入日志文件后, 向参与者发送确认消息 ACK(C). 参与者收到 ACK(C) 消息后, 标识日志文件中该分布式元数据操作第 2 个子操作的日志记录可以清除.

(6) 参与者将第 2 个子操作的日志记录持久写入日志文件后, 向协调者发送确认消息 ACK(P). 协调者收到 ACK(P) 消息后, 协调者标识日志文件中该分布式元数据操作第 1 个子操作的日志记录可以清除.

(7) 协调者和参与者分别持久地将分布式元数据操作的子操作对元数据的更新写回磁盘指定的区域后, 将日志记录从日志空间中清除.

3.2 分布式日志的实现

在 BWMMs 中, 每个 MDS 上预留出一定容量的存储空间, 用于记录分布式元数据操作的重做记录 and 状态信息, 称为分布式日志空间. 在分布式日志空间中, 每一个分布式元数据更新操作对应一个记录项. 在分布式操作执行过程中, 协调者和参与者完成各自子操作后, 在分布式日志当中为此子操作分配并且初始化一个分布式日志记录项, 并且将各自子操作的重做记录传送到对方, 冗余记录在对

图 4 DL-ACP 协议流程
Figure 4 Process of the DL-ACP

方的分布式日志记录项当中. BWMMs 将分布式元数据更新子操作对分布式日志的更新和对元数据子卷的更新实现为一个本地事务, 从而保证分布式日志更新和元数据更新的原子性, 确保分布式日志中的状态信息能够准确匹配元数据的更新状态.

在 BWMMs 中, 已完成的本地事务并不需要立即同步提交到本地日志中, 而是采用一种异步的方式, 按照顺序定期批量的持久提交到本地日志中. 在协调者宕机或者参与者宕机的情况下, 协调者或

者参与者首先利用本地日志恢复对分布式日志的更新和对元数据子卷的更新, 然后根据分布式日志保存的状态信息和重做记录对分布式元数据操作进行一致性恢复。

3.3 级联撤销的避免

在 BWMMs 中, 本地日志技术采用了异步提交方式, 在一个元数据更新操作还没有持久提交的时候, 允许依赖于该元数据操作结果的后续操作执行。对于跨 MDS 的分布式元数据操作, 不同 MDS 上子事务独立提交, 在 MDS 之间不是同步的。因此当元数据服务器宕机造成某些事务被撤销后, 也需要在其他 MDS 上级联撤销依赖于该被撤销事务的其他事务。级联撤销会导致文件系统不一致。在 BWMMs 中, 在分布式元数据操作各个子操作对应的本地事务开始执行时, 会强制提交在该分布式元数据操作之前发生的被依赖的其他本地事务, 保证了被依赖的文件系统更新都已经持久到存储设备中, 从而避免了级联撤销。

3.4 减少阻塞对系统性能的影响

在 BWMMs 中, 分布式日志记录项的大小是固定的, 并且是按顺序循环使用的, 未被清除的分布式日志记录项之后的分布式日志空间, 是不可以分配给新建的分布式事务的, 如果某个分布式元数据操作的日志记录一直不能被清除, 就会出现没有分布式日志空间可以被分配给新建的分布式事务的尴尬局面, 也就意味着涉及到该元数据服务器的分布式元数据操作都会被阻塞。分布式元数据操作的日志记录一直不能被清除, 是因为无法收到合作者元数据服务器发送的 ACK 消息, 这主要是由合作者元数据服务器宕机造成的。

为了减少因为分布式日志记录项不能被清除而造成的阻塞对系统性能的影响, BWMMs 在系统中实现了一个分布式日志后备文件, 超过一定时间不能被清除的分布式日志记录项将会被转移到该日志文件中, 从而缓解分布式元数据操作阻塞对系统性能造成的影响。

3.5 减少发送 ACK 消息对系统性能的影响

在 BWMMs 中, 协调者或者参与者子操作对元数据的更新和对分布式日志的更新实现为一个本地事务, 在这个事务提交到本地日志之后, 需要向合作者服务器发送 ACK 消息。对于 ACK 消息的发送, 可以在本地事务提交的过程中同步完成, 但是这样会延长本地事务提交的时间。因为在本地日志系统内, 当一个事务未完成提交时, 系统中处于运行态的事务无法进入提交状态, 即使这个事务已经无法容纳其他的原子操作, 这样就使得系统不能开始一个新的事务, 也就是说会阻塞元数据操作的执行, 造成系统元数据处理能力的降低。

基于以上分析 BWMMs 采用了异步的方式, 在本地事务提交完成之后, 由专门的进程来完成向合作者服务器发送 ACK 消息的工作。另外, 在 BWMMs 中, 如果参与者向协调者或者协调者向参与者发送 ACK 消息时, 发现对方已经处于 COMMIT 状态, 则可以将对方的 ACK 消息捎带回来, 不需要对方再单独发送一个 ACK 消息, 能够有效降低系统内消息传递的数量, 从而提升系统性能。

4 细粒度低延时的负载迁移

在 MDS 集群文件系统中, 由于系统规模大, 客户端访问随机性强, 导致元数据负载在 MDS 集群中分布不均, 主要包括元数据存储和客户端访问两部分在 MDS 间分布不均衡。轻负载 MDS 会使系统

的资源使用率降低, 重负载 MDS 由于处理请求过多导致系统元数据访问性能下降, 单位时间内系统整体的吞吐量也随之大幅减少. 解决 MDS 负载不均一般采用静态分配和动态再均衡两种策略^[20,21]. 静态分配指在系统生成元数据过程中尽量将元数据均衡地分布在不同 MDS 上; 动态再均衡指在元数据访问过程中, 一旦发现负载分布不均, 就根据某种特定策略将重负载 MDS 上部分元数据负载迁移到轻负载的 MDS 上. BWMMs 全局命名空间采用的以目录为粒度的分布式策略可以解决元数据本身分布不均问题, 但无法解决客户端访问不均问题, 同时也无法适应负载的动态变化. 因此, 本节介绍采用元数据负载迁移机制应对上述问题.

元数据负载迁移的本质是通过切换 MDS 的管理域和迁移相应的文件操作状态来实现客户端对元数据访问的转移, 从而实现 MDS 负载的迁移. 元数据负载迁移机制有 3 个基本要求: 一致性、透明性和低延迟. 此外, BWMMs 是提供状态服务来保证更严格的共享一致性、宕机一致性及会话一致性的. 为了保证元数据负载迁移的透明性和一致性, 必须进行状态的迁移.

为了实现 BWMMs 快速的元数据负载迁移, 同时对正常元数据访问的影响较小, 提出一种细粒度的元数据负载迁移方案, 包括两部分: (1) 基于日志读取与迁出端状态恢复的元数据服务迁移机制, 简称迁出端恢复迁移机制, 每个待迁移元数据子卷对应一个单独日志, 迁入端先通过顺序读取该日志恢复迁移元数据的内存结构, 然后通过迁移状态信息来恢复这些内存结构对应的状态, 保证了上层应用元数据访问的一致性和快速性. (2) 细粒度的状态迁移控制机制, 简称细粒度控制迁移机制, 在负载迁移过程中, 以文件为单位响应客户端请求, 对非写请求直接返回结果, 对影响元数据和状态一致性的写相关请求, 根据请求类型确定在迁出端或迁入端进行相应处理, 无需等待整个元数据子卷迁移完毕再响应.

4.1 迁出端恢复迁移机制

迁出端恢复迁移机制分为 3 个阶段: (1) 基于日志读取的元数据迁移; (2) 基于迁出端状态恢复的状态迁移; (3) 客户端请求重定向. 元数据迁移负责切换 MDS 管理域, 状态迁移负责迁移状态保证元数据访问的透明性和一致性, 客户端请求重定向负责将客户端的元数据请求重发到迁入端.

4.1.1 元数据迁移

本机制在保证元数据一致性前提下对现有方案进行了优化, 降低了迁移时间并预热迁入端缓存. 如图 5 所示, 迁移机制主要流程如下: (1) 停止写入访问. 迁出端停止对元数据子卷写入访问, 但允许元数据读访问和状态读写访问; 同时, 主动唤醒 kjournald 线程将脏元数据写入日志. (2) 读取日志. 迁入端读取日志到内存, 并复制内容到磁盘块缓存区; 同时, 通过建立 VFS 层文件系统管理结构获取元数据子卷管理权. (3) 迁出端迁移状态到迁入端. 迁出端逐个扫描元数据子卷的所有状态, 并在迁入端重建, 状态数据迁移的详细过程随后介绍. (4) 标记干净且不做检查点 (Checkpoint) 操作. 迁出端标记所有元数据为干净即返回, 然后通过异步线程释放内存中的状态和元数据缓存结构, 释放 VFS 层文件系统管理结构, 但因为此时日志块对应的缓冲区记录已在迁入端, 所以不做 Checkpoint 操作.

4.1.2 状态数据迁移

在状态数据迁移过程中, 迁入端无需等待所有客户端节点恢复状态, 只需等待迁出端一个节点传输状态结束即开始恢复对迁移元数据子卷的元数据服务. 基于迁出端状态进行迁移的总体思路是将迁

图 5 基于日志读取的元数据迁移机制

Figure 5 Metadata migration mechanism based on read journal

出端元数据子卷相关的所有状态结构在迁入端进行重建,并在客户端继续维护相应的状态和 `state_id`,然后由迁入端重建 `state_id` 与相应状态结构的映射关系。

迁出端恢复状态的主要流程如下. 对于每一个状态的迁移, 首先将 RPC 操作号及相关参数编码到 XDR 数据流中, 然后发送 RPC 请求, 其中迁移 Open, Lock, Delegation, Layout 状态的 RPC 操作号分别为 MIGOPEN, MIGLOCK, MIGDELEG 和 MIGLAYOUT. 相应的, 迁入端要在内核守护线程 `nfsd` 收到恢复状态请求后, 根据状态参数, 建立相应状态及 `state_id`. `pNFS` 中 `state_id` 在节点内唯一, 但迁出端待迁移的 `state_id` 可能与迁入端已有 `state_id` 冲突. 因此, 在格式化 EXFS 文件系统时指定一个全局唯一的元数据子卷号, 写入其超级块中. MDS 在响应客户端请求生成相应的状态时, 在原方案生成的 `state_id` 前加上所访问元数据子卷的元数据子卷号, 使 `state_id` 全局唯一。

迁出端恢复迁移机制最简单的方式是逐个迁移所有状态, 但当迁移的状态规模较大时, 会带来过多的网络交互, 使迁移时间与状态的规模正相关. 采取 `compound` 和多线程两种方式进行优化, 减小状态规模对迁移延迟的影响。

4.1.3 客户端请求重定向

在迁移过程中, 客户端发送到迁出端的请求要重定向到迁入端, 由迁入端提供已迁移元数据子卷的元数据服务, 主要分为 3 个步骤:

(1) 元数据子卷映射表更新. 迁出端和迁入端在迁移完成后分别修改本地元数据子卷与 MDS 的映射表, 然后向根 MDS 发送更新映射表的请求, 最后由根 MDS 异步广播映射表更新信息。

(2) 客户端更新缓存的映射表. 客户端在访问迁出端的时候不断重试, 直到迁出端标识迁移完成, 客户端主动请求获取映射表。

(3) 客户端请求转发. 客户端根据更新后的映射表, 找到元数据子卷所在 MDS 的地址, 与迁入端建立连接或复用已有连接, 然后发送请求到迁入端。

迁出端恢复迁移机制为了保证迁出端和迁入端状态和元数据的一致性, 采取元数据子卷粒度的加

表 3 pNFS 请求分类
Table 3 Operation types and requests of pNFS

Operation type	Operation request
META_READ	Fsinfo getattr lookup lookup_root server_caps statefs readlink readdir getacl get_lease_time
META_WRITE	Commit rename link pnfs_laoutcommit create setacl pnfs_write pnfs_commit
STATE_READ	Lockt
STATE_WRITE	Delegreturn pnfs_layoutget pnfs_layoutreturn lock locku close
METASTATE_MIX	Read write setattr delegreturn
OPEN_TYPE	Open open_confirm open_downgrade

锁方法来控制元数据的读写, 在整个迁移阶段, 禁止对元数据子卷的所有元数据和状态的写, 只允许读. 但当需迁移的状态规模增大时, 元数据写及状态写延迟也会增大, 因此, 需要对迁移过程进行细粒度控制, 解决迁移过程中元数据迁移规模正相关的问题.

4.2 细粒度控制迁移机制

经过分析, 在以元数据子卷为粒度的负载迁移过程中, 只要实现元数据读写在迁入端响应, 客户端读到的元数据就一定是最新的, 即可保证元数据一致性. 对于状态, 只要保证未迁移的状态在迁出端访问, 已迁移的状态在迁入端访问, 即可保证状态写入的一致性. 因此, 提出细粒度的状态控制迁移机制, 使元数据和状态的写入延迟与迁移的状态规模无关, 并降低了迁移对最正常操作的影响.

pNFS 的元数据写入有两种类型, 一种是只进行元数据写入, 如 Mkdir 操作; 另外一种是先访问相关的状态, 获取元数据写入权限后才能进行元数据写入, 如 Setattr 操作. 对于两种元数据写入类型, 在保证元数据写入一致性的前提下, 本机制通过两种方式降低元数据写入延迟: (1) 在元数据迁移阶段禁止写入访问. (2) 在状态迁移阶段, 单纯元数据写入请求直接由迁入端进行响应, 混合元数据写入则是在迁出端立即迁移相关状态后由迁入端进行响应. 这使得元数据写入延迟最多只是元数据迁移时间、一个状态迁移时间、两次网络通信时间和元数据操作固有时间之和, 与迁移状态的规模无关.

细粒度控制迁移机制在迁移过程中要响应其他类型的客户端请求, 元数据和状态的迁移仍采用迁出端恢复迁移机制, 对客户端的请求则采用如下处理流程: (1) 判断客户端请求类型, 不同操作类型采用不同的处理方式, 保证访问一致性. (2) 迁出端根据迁移过程所处阶段决定客户端请求由迁出端还是迁入端处理. (3) 客户端在收到迁出端告知到迁入端进行响应的信息后, 进行请求重定向.

4.2.1 请求类型判断

根据对 pNFS 现有元数据操作的统计, 客户端元数据请求分为 6 种: 元数据读 (META_READ), 仅进行元数据的读操作; 元数据写 (META_WRITE), 仅进行元数据的写入操作; 状态读 (STATE_READ), 仅进行状态的读操作; 状态写 (STATE_WRITE), 更新 state_id 或者状态结构; 混合类型操作 (METASTATE_MIX), 这种类型的操作一般会先访问元数据相关的状态, 查看是否有访问权限, 然后再访问相应的元数据; 打开类型 (OPEN_TYPE), open 操作需要先访问索引节点, 然后创建相应的状态结构, 由于与之前的混合类型操作的处理方式不一样, 所以单独划分一个类型. 具体地划分结果如表 3 所示. 每一个操作在 RPC 调用过程中都会对应一个操作号 (op_num), 以 op_num 和表格中的分类作为依据进行操作类型的判断.

4.2.2 迁出端请求处理

迁出端获得客户端请求类型后, 还需要知道元数据子卷所处迁移阶段和被访问文件状态的迁移阶段, 才能判断是否立即响应请求, 不响应则返回错误码. 元数据子卷有 4 种状态: LOCAL, 元数据子卷在 MDS 本地, 所有访问正常进行; META_MIGRATING, 元数据子卷处于元数据迁移阶段, 允许元数据读和状态数据的读写, 禁止元数据写; STATE_MIGRATING, 元数据子卷处于状态迁移阶段, 所有的元数据访问转到迁入端进行, 其他类型请求依据所访问的状态是否已经迁移来进行具体的响应; ABSENT, 元数据子卷不在 MDS 本地, 如客户端请求访问该元数据子卷, 则更新 MDS_Map, 获取该元数据子卷所在 MDS 的地址, 转发请求到目标 MDS.

对于状态的迁移, 一次 compound RPC 传输恢复一个文件的所有状态. 每个 nfs4_file 文件定义 3 种状态: FILE_LOCAL, 对应的文件状态集合尚未迁移, 在初始化时设置; FILE_MIGRATING, 对应的文件状态集合处于迁移过程中, 在启动文件状态迁移前设置; FILE_ABSENT, 对应的文件状态集合迁移已经完成, 在文件状态迁移完成后设置. 与元数据子卷粒度状态迁移方式相比, 细粒度控制迁移机制在迁移每个文件的状态前先判断该文件是否已被迁移, 如果处于 FILE_LOCAL 状态就对该文件进行迁移, 否则跳过处理下一个文件.

迁出端收到客户端请求后, 首先判断客户端请求类型, 同时根据此时元数据子卷状态和相应文件迁移状态判断是否直接响应请求, 若不能响应, 则返回相应错误码. 客户端根据收到的错误码可以进行相应处理. 错误码有 5 种: (1) METAMIGRATING, 正在进行元数据迁移无法响应请求; (2) METAABSENT, 元数据已迁出无法响应请求; (3) STATEMIGRATING, 被访问状态处于迁移过程中无法响应请求; (4) STATEABSENT, 被访问状态已迁出无法响应请求; (5) MIGRATECOMPLETE, 所有迁移已经完成无法响应请求.

4.2.3 客户端请求重定向

元数据子卷粒度的迁出端恢复迁移机制中, 某一特定时刻对元数据子卷的访问只能在一个 MDS 上进行. 通过细粒度控制迁移机制的处理, 对元数据子卷的访问既可能在迁出端进行, 也可能在迁入端进行. 在元数据迁移阶段或者状态迁移阶段, 客户端根据迁出端返回的错误码, 选择延迟重试或将请求重定向到迁入端. 客户端在收到重定向消息后, 根据迁出端返回的迁入端地址, 与迁入端建立连接, 并重新发送请求. 具体的处理方式如图 6 所示.

客户端访问的元数据子卷由 nfs_server 结构标识, 对管理子卷的 MDS 的 RPC 访问通过 rpc_client 结构进行. nfs_server 结构中有一个 rpc_client 结构指针 client 标识管理元数据子卷的 MDS 的 RPC 通信结构, 所有相关 RPC 请求都通过该结构发送. 细粒度控制迁移机制在 nfs_server 结构中添加一个 rpc_client 指针 forward_client 标识迁入端的 RPC 通信结构, 这样对元数据子卷的访问既能在迁出端进行, 也可以将对元数据子卷的请求重发到迁入端. 具体的流程如下:

(1) 客户端设置 forward 标志. 客户端收到 METAABSENT 和 STATEABSENT 时, 在 RPC 出错处理路径的上下文中设置一个 forward 标志.

(2) 客户端 RPC 请求重发. 先检查该 forward 标志是否置位, 如果置位那么通过 forward_client 指针发送 RPC 请求到迁入端, 否则通过 client 指针发送到迁出端.

(3) 迁移结束. 在收到 MIGRATECOMPLETE 时, client 指针赋值为 forward_client, forward_client 指针置空, 保证后续所有请求都被发送到迁入端.

图 6 客户端请求重定向

Figure 6 Redirect of client request

5 测试与结果分析

BWMMS 采用 MDS 集群技术支持 EB 级存储系统对元数据操作的性能和扩展性需求, 其客户端为通用的 pNFS 客户端, 提供标准的 POSIX 接口, 与 HDFS 等系统相比, 对上层应用的支持更为广泛. 现阶段, BWMMS 实现了 4 个节点的 MDS 集群, 在千兆以太网环境下, 数据存储容量可超过 1000 PB, 索引节点总数达到 100 G, 单位时间每台 MDS 元数据操作数超过 10000 OPS. 本节首先介绍测试平台的系统配置; 然后分别介绍系统正常运行时的综合性能与各个元数据操作延时的测试结果, 并进行分析; 最后讨论负载迁移对元数据操作的影响.

5.1 MDS 集群系统测试环境配置

MDS 集群系统测试环境由 8 台物理服务器组成, 其中 4 台 Dell R710 组成 MDS 集群, CPU 型号为 Intel Xeon E5620, 内存 24 G; 4 台 Dell 1950 作为客户端, CPU 型号为 Intel Xeon L5320, 内存 4 G, MDS 和客户端的网卡均为 BCM5709 Gigabit Ethernet. 客户端及 MDS 除了系统盘是本地磁盘之外, 其他磁盘均是通过磁盘服务器采用 iSCSI 协议导出的虚拟网络磁盘设备. 这些虚拟磁盘被所有的 MDS 和客户端共享访问, 但系统设置客户端无法访问元数据子卷. 元数据子卷大小设为经验值 2 G, 数据卷大小不影响迁移时间, 这里设置为 100 G, 一个元数据子卷管理一个数据卷, 构成一个卷组.

5.2 综合性能测试

为了测试 MDS 的元数据操作的性能, 在 4 台客户端上分别安装元数据基准测试工具 mdtest, 同时运行测试命令, 只进行元数据测试, 不进行数据的读写. 针对每台 MDS 进行测试 100 次, 任一台

表 4 Compound 函数及其子操作的平均执行时间
Table 4 Average running time of compound function and its suboperation

Operation	Mkdir	Rmdir	Touch	Chmod	Ln	Unlink	Mv	Rm
Function	op_create	op_remove	op_open	op_setattr	op_link	op_remove	op_rename	op_remove
Compound (μ s)	102.501	40.455	85.911	18.933	31.087	24.063	37.143	30.513
Suboperation (μ s)	91.671	29.683	36.641	7.542	16.998	16.409	24.673	20.812

MDS 单位时间最低元数据操作数均超过 10000 OPS. 为了测试分布式日志恢复性能, 对 10000 个未完成的分布式元数据操作的恢复时间进行了测试, 重复 100 次, 在分布式日志的大小为 4 MB 的情况下, 其恢复时间最多不超过 1 s.

5.3 元数据操作性能测试

一个元数据的执行过程, 主要包括: (1) 客户端脚本执行和 RPC 消息准备阶段, 这段时间比较固定, 应用层所占时间较多, 不因类型元数据操作的 RPC 消息准备对整个操作时间的影响较小; (2) 网络交互阶段, 由于系统采用的网络类型不同, 千兆网与万兆网对操作时间的影响比较大, 但各类型的网络延时一般也是固定的; (3) MDS 端元数据操作的执行阶段, 不同类型元数据操作在此阶段的执行时间都是不同的. 因此, 为了更精确的测试系统元数据操作的性能, 采用在第 3 个阶段打印 10000 次同种操作的 compound 函数执行时间及其核心子操作执行时间的方式, 进行数据收集和统计, 结果如表 4 所示.

在表 4 中, 客户端依次执行以下 8 种元数据操作: mkdir, rmdir, touch, chmod, ln, unlink, mv, rm, 每种操作执行 10000 个操作对象. 创建目录和创建文件操作的延时最长, 其次为删除目录和重命名操作, 然后为链接和删除文件操作, 最后, 删除链接和修改属性操作的延时最短. 以上 8 种最重要的元数据操作的 compound 函数和核心子操作的执行时间处于 10~100 μ s 左右, 当前系统千兆网的网络交互时间约为 200 μ s, 两者相比, 网络交互时间占主要部分, 为了提高系统性能, 未来可采用万兆网或 IB 网络进行 MDS 之间的消息通讯, 而客户端仍采用千兆网与 MDS 进行交互.

5.4 负载迁移对元数据操作影响的测试

5.4.1 快速迁移测试

采用 LTP^[22] 测试套件的文件系统压力测试提供元数据访问负载, 首先在待迁移目录下运行压力测试, 然后对待迁移文件进行各种类型的元数据操作. 测试时间持续 24 h, 每隔 1 min 进行一次元数据子卷迁移. 测试结果显示, 在整个 LTP 压力测试期间, 元数据访问无异常, 证明元数据的迁移没有影响到应用对元数据访问的一致性.

在负载迁移过程中, BWMMMS 采用了 compound 和多线程技术, 加速了迁移速度. 在快速迁移测试中, 采用 3 种方式进行比较分析, 分别为: sequence, 即一次只迁移一个文件; compound, 即采用 RPC compound 方式; multithread, 即同时采用 compound 和多线程技术. 每个文件的状态数量设置为 10, 由一个 Open 状态加上 9 个非阻塞 Lock 状态组成, 分别测试 1, 10, 100 和 1000 个打开文件应用场景下进行元数据服务迁移的迁移时间.

如图 7 所示, 迁移状态规模较小时, compound 和 multithread 对于迁移时间的优化效果不明显. 在状态规模较大时, compound 和 multithread 对于迁移时间的优化则非常明显. 在文件数量为 1000 的迁移中, compound 的迁移时间相对于 sequence 降低了 80%, 而 multithread 相对于 sequence 降低

图 7 迁出端恢复迁移机制迁移时间

Figure 7 Migration time of emmigration recovery mechanism

了 86%. 相对于 pNFS 系统客户端恢复状态迁移机制的 90 s 迁移时间, 这里的迁移时间都有极大地减小. 从图 7 中同时可以看到, 状态规模越大, 迁移时间越长, 而 compound 和 multithread 降低了状态规模对迁移时间的影响, 证明了迁出端恢复迁移机制的有效性.

5.4.2 细粒度控制迁移机制元数据访问延迟

在不同状态规模下测试细粒度控制迁移机制进行元数据负载迁移时的元数据和状态写入延迟, 并与迁出端恢复迁移机制和正常操作进行比较, 3 种机制分别用 FCE, ERE 和 NOP 表示. 对于迁出端恢复迁移机制, 其操作延迟约等于其迁移时间加上一个正常操作延迟. 对于细粒度控制迁移机制, 由于迁移时间较短, 无法精确控制元数据操作与元数据迁移阶段的对应关系, 采用客户端操作延迟来估计元数据操作最大延迟. 客户端采用 Open 和 Close 进行状态数据写入操作, 采用 Mkdir 进行元数据写入操作, 采用 Setattr 进行状态与元数据写入混合操作, 每个操作前后记录时间戳, 从而能够等价测出元数据写入或者状态写入操作的最大延迟.

如图 8(a) 所示, 迁出端恢复迁移机制时间较长, 并且随状态规增大而增加, 细粒度控制迁移机制中, 发生迁移与没发生迁移时的 Open 操作延迟都较小且稳定在 30 ms 左右. 这是因为对 Open 操作采取了特殊处理方法, 先在迁出端进行 Open 后即返回成功给客户端, 然后在后续的元数据访问中访问到文件状态时再进行文件状态的迁移.

如图 8(b) 所示, 在细粒度状态迁移机制中, Mkdir 操作的延迟一直稳定在 173 ms 左右, 与迁移的状态规模无关. 迁出端恢复迁移机制的 Mkdir 延迟在 10000 个状态时达到了 711 ms, 采用细粒度控制迁移机制 Mkdir 延迟降低了 75%.

如图 8(c) 所示, 细粒度状态迁移机制的 Setattr 操作延迟一直稳定在 200 ms 左右. 在 10000 状态规模下, 比迁出端恢复迁移机制的 700 ms 延迟降低了 71%. 虽然对于 EXFS 文件系统来说 Setattr 操作要比 Mkdir 操作耗时少, 但在细粒度状态迁移机制中 Setattr 操作延迟要比 Mkdir 操作大. 这是因为对于 Setattr 操作, 需要先访问状态才能进行属性信息写入, 因而需要先迁移文件状态到迁入端, 再到迁入端设置属性, 这个时间开销大概是 30 ms.

图 8 元数据操作的方向延迟

Figure 8 Operation delay. (a) Open operation; (b) mkdir operation; (c) setattr operation; (d) close operation

如图 8(d) 所示, 迁出端恢复迁移机制的 Close 操作延迟较大, 细粒度状态迁移机制的 Close 操作延迟一直稳定在 2 ms 左右. 由于 Close 操作仅仅只是涉及状态结构的释放, 以及 VFS 层结构引用计数的递减, 并不涉及磁盘操作, 因而延迟很小. 同时, 细粒度状态迁移机制中的 Close 操作只是删除迁入端的文件状态, 而对于迁出端的状态则是在迁移完成之后异步地删除. 采用这种方式, 细粒度迁移方法中 Close 操作相对于正常 Close 操作只多了一次网络延迟.

综合上述 4 种元数据操作的访问延迟可以看出, 细粒度状态迁移机制能够在大规模状态迁移场景中有效降低元数据或状态数据的写入延迟, 使元数据或者状态数据写入延迟与迁移状态规模无关.

6 结论

本文介绍了中国科学院计算技术研究所为了支持 EB 级存储而开发的 MDS 集群技术, 此技术是在现有的 pNFS 和蓝鲸块设备文件系统 EXFS 的基础上, 通过对整体架构、交互协议和操作流程进行调整和重新设计而完成的, 支持通用的 POSIX 接口, 易于部署且应用前景广泛. 蓝鲸 MDS 集群原型系统 BWMMs 整体架构简单清晰, 实现了满足存储局部性原理的全局命名空间分布; DL-ACP 协议避免了在分布式操作过程中宕机引起的元数据访问不一致问题; 在对正常元数据操作影响较小的前提

下, 细粒度低延时的负载迁移技术实现了 MDS 集群系统的快速动态负载再均衡. 测试和结果分析表明, BWMS 系统采用的 MDS 集群技术能够较好的满足 EB 级存储系统对元数据操作的性能需求.

参考文献

- 1 Vernon T, John F G, David R, et al. The Digital Universe of Opportunities. IDC_1672. 2014
- 2 Ghemawat S, Gobioff H, Leung S. The Google file system. In: Proceedings of SOSP'03, Bolton Landing, 2003. 29–43
- 3 Makoto S, Hiroki K, Shoji K, et al. Performance evaluation of scale-out NAS for HDFS. In: Proceedings of IMMM'13, Lisbon, 2013. 32–35
- 4 Jain R, Sarkar P, Subhraveti D. GPFS-SNC: an enterprise cluster file system for big data. IBM J Res Dev, 2013, 57: 5–14
- 5 Gary L R, Jesse H, Rick M. Data management practices on large-scale lustre scratch file systems. In: Proceedings of XSEDE'14, New York, 2014. 1–6
- 6 Taejin K, Sam H. pNFS for everyone: an empirical study of a low-cost, highly scalable networked storage. Int J Comput Sci Netw Secur, 2014, 14: 52–59
- 7 Liu J L, Zhang Y L, Yang L, et al. SAC: exploiting stable set model to enhance CacheFiles. J Comput Sci Technol, 2014, 29: 293–302
- 8 Alex D, Alessandro O. Scale out with GlusterFS. Linux J, 2013, 235: 72–82
- 9 Feiyi W, Mark N, Sarp O, et al. Performance and scalability evaluation of the Ceph parallel file system. In: Proceedings of the 8th Parallel Data Storage Workshop, New York, 2013. 14–19
- 10 Kirk M, Sean Q. GFS: evolution on fast-forward. Commun ACM, 2010, 53: 42–49
- 11 Shafeeq S, Raja R, James H, et al. A transparently-scalable metadata service for the Ursa Minor storage system. In: Proceedings of USENIXATC'10, Berkeley, 2010. 13–26
- 12 Alexander T. Storage research in industry and universities. ACM SIGARCH Comput Architect News, 2010, 38: 1–48
- 13 Menon J, Pease D, Rees R, et al. IBM storage Tank—a heterogeneous scalable SAN file system. IBM Syst J, 2003, 42: 250–267
- 14 Carl W O. Achieving the Real-Time Enterprise with MapR. IDC_250694. 2014
- 15 Josephine P, Robert B, Kevin S. Kerberized Lustre 2.0 over the WAN. In: Proceedings of TG '10, New York, 2010. 1–5
- 16 Ma H Y, Liu Z J, Zhang H, et al. Experiences with hierarchical storage management support in Blue Whale file system. In: Proceedings of PDCAT'10, Washington, 2010. 369–374
- 17 Guo T, Shen Y L, Liu Z J, et al. BW-FILERAIID: a kind of file based distributed RAID system and optimization. Appl Mech Mater, 2011, 80: 1208–1216
- 18 Shaikh M A, Salahuddin M, Fayshal A, et al. Introducing iSCSI protocol on online based MapReduce mechanism. Lect Notes Comput Sci, 2014, 8583: 691–706
- 19 Shao B Q, Zhang J W, Zheng C P, et al. A non-forced-write atomic commit protocol for cluster file systems. J Comput Sci Technol, 2014, 29: 303–315
- 20 Xu Q Q, Rajesh A, Khai L Y, et al. Efficient and scalable metadata management in EB-scale file systems. IEEE Trans Parall Distr Syst, 2014, 25: 2840–2850
- 21 Xu Q Q, Rajesh A, Khai L Y, et al. DROP: facilitating distributed metadata management in EB-scale storage systems. In: Proceedings of MSST'13, Washington, 2013. 1–10
- 22 Chen Y, Wu F G, Yu K L, et al. Instant bug testing service for Linux kernel. In: Proceedings of HPCC'13, Washington, 2013. 1860–1865

Metadata server clustering system for EB-scale storage

LIU Jian^{1,2*}, ZHANG JunWei¹, SHAO BingQing¹, DONG HuanQing¹, LIU ZhenJun¹ & XU Lu^{1*}

¹ Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China;

² University of Chinese Academy of Sciences, Beijing 100049, China

*E-mail: liujian@bwstor.com.cn, xulu@bwstor.com.cn

Abstract With data volumes rapidly growing on a global scale, PB-scale storage systems will not meet the data-storage requirement capacity of applications in the future. It is time for the research and development of EB-scale storage systems that are general purpose, highly scalable, and easily deployed. Metadata server clustering is the core technology for constructing EB-scale storage systems. Based on pNFS and the Blue Whale block device file system EXFS, Blue Whale metadata server clustering technology for EB-scale storage is presented. To begin, the differences between EB-scale metadata server systems and PB-scale are analyzed. Then, the prototype architecture of a metadata server cluster and the distribution strategy of a global namespace are introduced. Following this, the atomic operation protocol for consistency based on a distributed log is presented. Finally, fine-grained metadata migration technology with low latency is proposed. Measurement results confirm that each metadata server can offer more than 10,000 OPS to EB-scale storage systems simultaneously.

Keywords EB-scale, storage, metadata server clustering, global namespace, consistency, load migration

LIU Jian was born in 1981. He received his M.Sc. degree from Yanshan University in 2008. He is currently a Ph.D. candidate at the University of the Chinese Academy of Sciences. His research interests include load balancing and high availability of cluster file systems.

ZHANG JunWei was born in 1977. He received his Ph.D. degree in computer architecture from the Institute of Computing Technology, Chinese Academy of Sciences (ICT, CAS) in 2010. He is currently an associate professor at the Data Storage and Management Technology Research Center, ICT, CAS. His research interests include cluster file systems and network storage.

LIU ZhenJun was born in 1976. He received his Ph.D. degree in computer architecture from the Institute of Computing Technology, Chinese Academy of Sciences (ICT, CAS) in 2006. He is currently an associate professor at the Data Storage and Management Technology Research Center, ICT, CAS. His research interests include WAN storage, distributed file systems, and raid management.

XU Lu was born in 1962. He received his Ph.D. degree in Computer Systems Software from Purdue University, USA, in 1995. He is currently a professor at the Data Storage and Management Technology Research Center, the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include computer architecture, high performance network storage, and computer system software.