计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2024,33(7):188−200 [doi: 10.15888/j.cnki.csa.009581]
©中国科学院软件研究所版权所有.
E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

LRCRaft: 支持节点数据快速恢复的共识协议①

袁佳正, 胡晓鹏

(西南交通大学 计算机与人工智能学院, 成都 611756)
通信作者: 胡晓鹏, E-mail: xphu@swjtu.edu.cn

摘　要: 在支持纠删码的分布式存储系统中, 最常用的编码是 RS (Reed-Solomon) 码. 对于一个 $RS(k, m)$ 编码条带, 常见的配置是一个节点仅存储条带中的一个分片, 这导致在节点出现故障的情况下, 对其存储分片的恢复需要跨多个节点读取分片并重新编码生成恢复分片, 容易造成系统网络拥塞. 在需要恢复大量数据的场合, 系统在恢复期间会处于较长时间的脆弱期, 容错能力和吞吐量下降、读写时延升高时有发生. LRCRaft 是一个基于 LRC (local reconstruction code) 的改进 Raft 共识协议, 通过在 Raft 中引入 LRC 码、动态日志增补、状态机删减和分片版本一致性等机制, 降低了 Raft 的读写时延, 缩短了节点故障恢复时间. 实验结果表明, 相较于 Raft, LRCRaft 在不同恢复模式中恢复一个单节点故障数据时, 恢复用时有着 49.25%–74.97% 的减少.

关键词: 分布式存储; Raft 共识协议; 纠删码; 局部重构码 (LRC); 节点数据恢复

引用格式: 袁佳正,胡晓鹏.LRCRaft: 支持节点数据快速恢复的共识协议.计算机系统应用,2024,33(7):188–200. http://www.c-s-a.org.cn/1003-3254/9581.html

LRCRaft: Consensus Protocol with Rapid Node Data Recovery Support

YUAN Jia-Zheng, HU Xiao-Peng

(School of Computing and Artificial Intelligence, Southwest Jiaotong University, Chengdu 611756, China)

Abstract: RS (Reed-Solomon) code is most widely adopted in distributed storage systems that support erasure coding. For an $RS(k,m)$ coding stripe, a common approach to store it is to distribute one fragment to one node. Such an approach could cause network congestion when a node fails since the system needs to read fragments across multiple nodes before it can decode and rebuild the lost data. The system would be in a fragile period for a long time when a great amount of data recovery is taking place. During this period, the system would suffer from lower failure tolerance capability, lower throughput, and higher read/write latency constantly. LRCRaft is an optimized version of Raft based on local reconstruction code (LRC). By introducing LRC, dynamic log replenishment, state machine purge, and fragment version consistency to Raft, LRCRaft can reduce read/write latency and the time consumed for node failure recovery. The results of our experiments indicate that compared to Raft, LRCRaft can reduce the time for a single node recovery by up to 49.25%–74.97% in different recovery modes.

Key words: distributed storage; Raft consensus protocol; erasure coding; local reconstruction code (LRC); node data recovery

分布式存储系统常用多副本来提高存储系统的可靠性. 采用多副本的优势是实现简单、拥有较为可靠的容错能力和相对良好的读写性能. 然而在大数据时代, 用户数据量呈指数级增长, 采用多副本的分布式存

① 基金项目: 河北省自然科学基金 (F2022105033)
收稿时间: 2024-01-23; 修改时间: 2024-02-26; 采用时间: 2024-03-18; csa 在线出版时间: 2024-06-05
CNKI 网络首发时间: 2024-06-08

© 中国科学院软件研究所 http://www.c-s-a.org.cn

储系统面临着空间利用率不高、网络带宽开销较大等问题. 在分布式存储领域中, 网络时延又是存储系统性能的决定因素之一. 存储系统网络带宽的高开销容易引发网络拥塞, 最终导致时延升高, 影响上层应用响应速度和终端用户体验. 为解决上述问题, 一些分布式存储系统的数据冗余策略以数据编码替代传统的数据副本. 纠删码 (erasure code, EC)[1,2]通过对数据进行编码, 添加适当的冗余信息, 具有一定程度的容错和纠错能力. 以最常用的 Reed-Solomon 编码 (RS 码) 为例, $RS(k, m)$ 将原始数据切分成 k 个数据分片 (data fragment), 并根据 k 个数据分片生成 m 个校验分片 (parity fragment). 在 $k+m$ 个分片中, 任取 k 个即可计算出原始数据, 故 $RS(k, m)$ 码可以容忍至多 m 个分片错误. 在实际应用中, 通常 k 大于 m, RS 码作为数据冗余策略要比三副本节省存储空间. 因此, 一些大规模的存储系统 (Tectonic[3]、Windows Azure Storage[4]、HDFS[5]以及 Ceph[6]) 采用 EC 作为冗余策略.

EC 有两个优良特性: (1) 容错能力可以通过参数配置; (2) 存储空间利用率较高. 在分布式存储领域, 引入 EC 冗余策略可以显著优化存储系统空间利用率, 但同时也会带来一些新的问题, 例如: 如何保证跨节点数据一致性? 分布式存储系统的共识协议往往存在一个 Leader 节点来推动各存储节点的共识进程. 采用多副本的系统, Leader 节点向其余节点分发数据副本并使所有节点的数据达到最终一致性; 采用 EC 的系统, Leader 节点分发彼此间完全不同的数据分片或校验分片, 如何检测并修正各节点存储分片的错误, 并保证对原始数据的恢复能力是 EC 与分布式存储结合的关键问题. 如何保证存储系统可用性级别 (liveness) 不下降? 分布式存储系统的一个共识组是若干节点的集合, 这些节点在共识协议的约束下协同工作, 对外表现为一个整体. 可用性级别是一个共识组最多能容忍的故障节点数量. 例如, 一个拥有 $2F+1$ 个节点的 Raft 共识组可用性级别为 F[7,8]. 引入 EC 后, 要如何设计共识协议以使存储系统可用性级别不下降也是一个关键问题. 如何处理更加复杂的节点故障恢复? 多副本冗余策略下, 恢复一个故障节点仅需从 Leader 节点同步故障期间落后的数据, 引入 EC 后则需从多个节点进行同步, 此过程还涉及数据的编码/解码, 进一步增加节点数据恢复的复杂度.

LRC (local reconstruction code)[4,9]码是微软提出的一个基于 RS 码的变种, 通过多次编码、引入局部组和全局组的设计实现了比 RS 码更优良的理论恢复性能. 本研究工作通过将 Raft 协议和 LRC 码相结合, 提出了一种新的共识协议: LRCRaft. LRCRaft 适配了 LRC 码的特性, 在不降低系统可靠性的前提下缩短了共识组中节点故障的处理时间. 此外, LRCRaft 拥有和 Raft 相同的可用性级别, 并且支持动态日志增补、均衡状态机数据、快速节点恢复的特性, 以保证数据写入的低时延和 LRC 冗余策略的可靠性.

本文后续内容将按照以下结构进行安排: 第 1 节概括与 EC 相关的共识协议研究工作. 第 2 节详细阐明 LRCRaft 的协议设计和关键特性. 第 3 节构建了一个支持可插拔协议的分布式存储原型系统, 通过对不同协议的性能进行对比测试, 验证 LRCRaft 的性能指标. 最后对工作进行总结.

1 相关工作

最早将 EC 引入分布式存储领域的共识协议是 RS-Paxos[10]. 该协议对 Paxos[11,12]的同步机制进行了修改, 使共识组每次同步一个值时使用的是分片而不是完整数据, 从而节省了网络带宽和存储开销, 但是带来了共识组可用性级别下降的问题. Paxos 通常将共识组分为两个类别: 读多数派 Q_R 和写多数派 Q_W. 读多数派指的是读取一个数据至少需要 Q_R 个节点返回其本地存储的分片才能恢复完整数据; 写多数派指的是 Q_W 个节点写入成功才认为共识组对某个数据写入成功. 根据容斥原理, Q_R 和 Q_W 需满足[10]:

$$Q_R + Q_W - N \geqslant k \tag{1}$$

其中, N 是共识组的节点数, k 是数据分片的数量. RS-Paxos 只有在至少 $\max\{Q_R, Q_W\}$个节点可用的情况下, 共识组才处于可用状态. 根据式 (1) 可得:

$$\max\{Q_R, Q_W\} \geqslant \frac{Q_R + Q_W}{2} \geqslant \frac{N+k}{2} \tag{2}$$

在保证 RS-Paxos 和 Raft 的共识组大小一致的前提下, 有 $N=2F+1$, 故 RS-Paxos 的可用性级别 $L_{\text{RS-P}}$:

$$L_{\text{RS-P}} \leqslant N - \frac{N+k}{2} = F - \frac{k-1}{2} < F \tag{3}$$

根据式 (3) 可得 RS-Paxos 的可用性级别低于 Raft, 为了解决存储系统可用性级别下降问题, CRaft[13]协议应运而生. CRaft 是一个将可用性级别作为重要指

© 中国科学院软件研究所 http://www.c-s-a.org.cn

标的协议, 其目标是在 Raft 上引入 EC 冗余策略且不降低系统的可用性级别. Raft 协议采用日志对客户端下发的写请求进行封装, 日志分为两部分: Raft 协议头和载荷数据 (即客户端要存储的原始数据). Raft 协议头包含了共识组保证数据一致性的必要信息, 如请求类型、实际载荷大小、日志编号 (index) 和创建日志的 Leader 节点任期 (term) 等. 通过日志机制, Raft 共识组各节点得以保证数据一致性: 只要所有节点按照相同的顺序将日志应用 (apply) 到本地状态机上即可保证各节点存储的数据是一致的. 在 Raft 中引入 EC 后, 需要对日志中的载荷数据进行编码, 从而形成一个编码条带. 该编码条带上不同的分片通常需要分散存储到共识组中不同的节点中, 因此需要为每个分片加上对应的共识协议头以便共识组节点能够正确存储载荷分片. 一个对所有载荷分片都加上对应协议头的编码条带称为日志条带, 日志条带中载荷分片及其共识协议头被称为日志分片.

在 CRaft 中 N 是共识组的节点数, 且 $N=k+m$. 为了与 Raft 协议进行可用性级别的直观对比, CRaft 还满足 $N=2F+1$, 即控制共识组大小与 Raft 相等. 为了达到和 Raft 相同的可用性级别, CRaft 确立了一个混合冗余策略: 当共识组中可用节点的数量不小于 $F+k$ 时, CRaft 节点之间采用 EC 冗余策略分发日志分片进行数据同步; 当可用节点数量处于区间$[F+1, F+k)$ 时, 则在节点之间分发完整日志, 即退化为多副本方式. CRaft 解决了引入 EC 后系统可用性降级的问题, 但是在可用节点数不够的场景下, 混合冗余策略带来了额外的带宽和存储开销.

受 CRaft 混合冗余策略的启发, HRaft[14]被提出, 其核心思想是要消除 CRaft 的混合冗余策略, 达到在 Raft 中完全使用 EC 冗余策略并且不降低系统可用性级别的目标. HRaft 引入了 Helper 组的概念, Helper 组由共识组中的 F 个可用节点组成, 由 Leader 节点选出. 在 Leader 节点分发一个日志条带时, 当可用节点数量处于区间$[F+1, F+k)$ 时, Leader 节点会把原本应当分发给那些故障节点的日志分片分发给 Helper 组中的每个节点, 该操作被称为增补 (replenishment). 在故障节点恢复后, HRaft 需要对冗余的日志分片进行删除, 具体做法是 Leader 节点从 Helper 组中读出日志分片, 重新分发给自故障中恢复的节点, 分发成功后通知 Helper 组节点删除对应的日志分片.

ECRaft[15]是另一个对 CRaft 改进的协议, 它和 HRaft 有一定的相似之处. ECRaft 衡量多数派的标准和上述几个协议均不同, 其选取的多数派集合大小为 k, 共识组大小为$N = k+m, k > m$. ECRaft 是一个纠删码同步写协议, 在所有节点均可用的情况下, ECRaft 会等待每一个节点写成功日志分片才对上层返回写成功; 当共识组中有$1 \leqslant f \leqslant m$个节点故障时, ECRaft 为剩余节点分发的日志分片数量不再是 1, 而是 $1+f$, 即包含了本该分发给故障节点的所有日志分片, 这个过程和 HRaft 相似. ECRaft 提出了状态机删减 (state machine purge) 来处理节点上存储的额外日志分片: 当节点从故障中恢复时, Leader 节点通过访问其余节点将该节点的数据进行恢复后, 就通知其余节点删除与该节点相关的额外状态机数据. 当所有节点都从故障中恢复后, 可以保证共识组中每个节点状态机存储的数据都是原始数据大小的 $1/k$.

Raft 协议的设计目的是保持简单、易理解的特性, 便于教学和演示, 而在写性能、存储空间利用率等方面有所妥协, 将其运用到实际存储系统中仍有较大的改进空间. CRaft、HRaft 和 ECRaft 均对 Raft 做出了适当优化, 使其性能有了可观的改进. 然而上述协议重点关注 Raft 引入 EC 后如何保持可用性级别, 而没有对数据丢失或新加入共识组的节点进行数据重构时会遇到的性能瓶颈进行讨论. 本文重点关注数据重构时间这一性能指标, 将 LRC 和 Raft 进行结合构建一个新的共识协议——LRCRaft, 并使协议拥有和 Raft 一样的可用性级别、低 I/O 时延、较强的可靠性和快速故障恢复能力.

2 LRCRaft 协议设计

2.1 LRC 码和 LRCRaft 的形式化定义

在分布式存储系统中采用 EC, 故障恢复时需要跨节点读取的分片数量较多, 由于网络拥塞等客观因素的存在, 这样的系统在恢复数据时通常会遇到恢复速度缓慢、读写性能下降的问题. 以 EC 的这个特性为切入点, 微软提出了 LRC 码, 该编码是 EC 家族中的一个编码变种, LRC 编码过程分为两个阶段: 第 1 阶段采用 RS 编码将原始数据编码成一个由数据分片和校验分片构成的 RS 编码条带; 第 2 阶段对数据分片进行分组, 再对每个分组进行一次 RS 编码, 生成新的校验分片. 如图 1 所示, 在第 1 阶段, 原始数据 D 被编码成了

© 中国科学院软件研究所 http://www.c-s-a.org.cn

4 个数据分片{D_1, D_2, D_3, D_4}和 2 个校验分片{P_1, P_2}; 在第 2 阶段, 将数据分片分为{D_1, D_2}和{D_3, D_4}两组, 对两个分组再进行一次 RS 编码, 生成新的校验分片{LP_1, LP_2}. 这两个阶段生成的所有分片组合在一起构成一个 LRC 编码条带. 为便于区分, LRC 码将第 1 阶段生成的校验分片称为全局校验分片 (global parity fragment, GP), 对应{GP_1, GP_2}; 第 2 阶段生成的校验分片称为局部校验分片 (local parity fragment, LP), 对应{LP_1, LP_2}. 这样, 原始数据 D 经过编码后得到了一个由 4 个数据分片、2 个全局校验分片和 2 个局部校验分片构成的 LRC 编码条带. 将 LRC 编码条带中第 1 阶段产生分片构成的集合称为全局组, 即{D_1, D_2, D_3, D_4, GP_1, GP_2}; 第 2 阶段各分组的数据分片, 及其生成的校验分片构成的集合称为局部组, 即{D_1, D_2, LP_1}和{D_3, D_4, LP_2}.

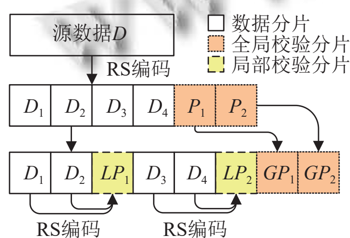

图 1 LRC 编码过程

以局部组{D_1, D_2, LP_1}为例, 假设分片 D_1 损坏, 可以通过对分片 D_2 和 LP_1 做一次 RS 解码进行恢复. 如果没有 LRC 引入的二阶段分组及编码流程, 普通的 RS 码只有第 1 阶段编码过程, 分片 D_1 损坏只能通过对{D_2, D_3, D_4, P_1, P_2}中的 4 个分片进行一次 RS 解码恢复, 恢复代价 (参与恢复的分片数量) 是 LRC 的两倍. 由于 LRC 码分组机制的引入, 其局部组分片的恢复代价总是要小于 k, m 两个参数与之相等的 RS 码, 这个优势在编码越长、局部组越多的情况下越显著.

EC 类编码具有 MDS 性质, 即一个 (k, m) 的编码序列代表原始的 k 个数据分片编码后生成了 m 个校验分片, 且这 $k+m$ 个分片中任取 k 个分片可将原始 k 个数据分片恢复出来. LRC 不具备 MDS 性质, 一个 LRC 编码条带中任取 k 个分片不保证能恢复原始数据.

将 LRC 码局部组数据恢复代价低的特性应用到分布式存储系统中, 理论上可以减少恢复局部组节点数据所需的节点数量, 减少网络总带宽的消耗从而降低时延, 提升节点数据恢复性能. 基于 Raft 易理解、易实现的特性, 本文选择 Raft 作为基础共识协议, 修改其工作流程以支持 LRC 码. 为适应 LRC 码的分片特性, Raft 共识组中的每个节点存储 LRC 编码条带中的一个分片. 由于 LRC 码将分片分为 3 种类型和两个组别, 因此 Raft 共识组按照节点存储的分片类型, 将节点分为数据节点、全局校验节点和局部校验节点, 并同样分为全局组和局部组两个节点组别. 经过上述对 LRC 码适配的 Raft 协议称为 LRCRaft. 图 2 是图 1 中所示的 LRC 编码条带在一个 LRCRaft 共识组中的分布示例. LRCRaft 将这 8 个节点分为了 1 个全局组和 2 个局部组, 全局组的节点存储全部数据分片和全局校验分片{D_1, D_2, D_3, D_4, GP_1, GP_2}, 两个局部组分别存储部分数据分片及其对应的局部校验分片, 即{D_1, D_2, LP_1}和{D_3, D_4, LP_2}.

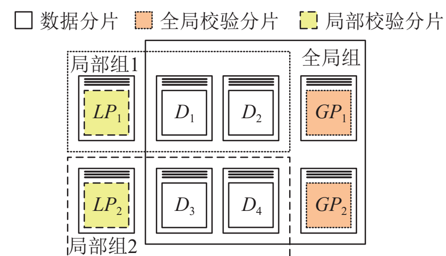

图 2 LRC 编码条带在 LRCRaft 共识组节点上的分布

下面给出 LRCRaft 的形式化定义: LRCRaft 是一个支持 LRC 码的 Raft 共识协议, 其共识组配置用一个四元组 $L(k,m,n,r), k,m,n,r \in N+, k > m,n \mid k$ 来描述, 其中 k 表示数据节点数量, m 表示全局校验节点数量, n 表示局部组数量 (全局组数量为 1), r 表示每个局部组包含的局部校验节点数量. 对应于图 2 中的 LRCRaft 共识组, 其配置描述为 $L(4, 2, 2, 1)$.

2.2 Leader 节点选举和日志提交规则

LRCRaft 的 Leader 节点选举环节限定了只有全局组中的节点才可以参与 Leader 节点选举, 也就是在一个 $L(k, m, n, r)$ 共识组中只有 $k+m$ 个节点可以参选. 在限定条件 $k>m$ 的约束下, 使得这其中任意 k 个节点即可构成多数派 (majority). 故 Candidate 节点发起选举后, 收到至少 $k-1$ 个其余全局组节点的投票才可成为 Leader 节点.

与 ECRaft 一样, LRCRaft 也是一个同步写协议. 因此 LRCRaft 的 Leader 节点在分发日志分片时总是尽可能等待所有可用节点将某个编号的日志分片提交后, 才更新共识组的日志提交编号. 如图 3 所示, 一

© 中国科学院软件研究所 http://www.c-s-a.org.cn

个 $L(4, 2, 2, 1)$ 共识组的所有节点均为可用节点, 此时 Leader 节点的日志提交进度总是与共识组中最慢的节点保持一致. 如果存在故障节点, 则至少需保证全局组中有 k 个节点处于可用状态, LRCRaft 共识组才能继续提交日志, 否则整个共识组将无法正常工作. 存在故障节点时, 为保证数据安全性, LRCRaft 使用动态日志增补机制进行日志提交, 该机制将在第 2.3 节中讨论.

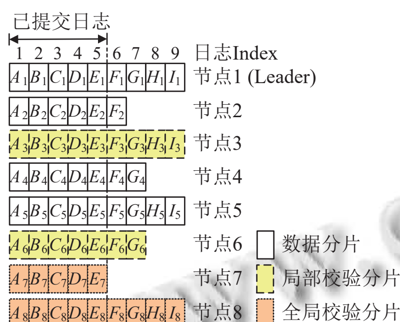

图 3 LRCRaft 日志

2.3 动态日志增补

在一般情况下, Leader 节点会把日志分片分发给其余节点, 等待每个节点都返回写成功后再推进共识组应用日志条带, 最后向上层返回数据写成功. 然而在真实的场景下, 节点故障是不可避免的, 因此 LRCRaft 改进了 ECRaft 的日志增补机制来保证在部分节点故障情况下共识组的可用性.

LRCRaft 的 Leader 节点会根据最近一次的心跳响应来推测共识组各节点当前的状态. 当全局组有 $x\,(x \leqslant m)$ 个节点故障时, Leader 节点会将原本要分发给这 x 个节点的日志分片打包, 作为一个全局增补请求发送给全局组中的那些可用节点上, 这类节点被称为全局增补节点. 全局增补节点在收到全局增补请求后, 按照正常日志分片处理流程处理这些分片, 包括日志分片的存储和应用到状态机的全流程.

局部组的引入是为了减少节点恢复所需时间, 因此局部组中的节点应当尽量与整个共识组的数据处理进度保持同步, 这样当本组的其他节点从故障中恢复后, 局部组节点能够提供足够新的增补数据以供节点快速追赶共识组的整体进度, 加快节点恢复速度. 为实现这一点, Leader 节点在进行日志增补时就需要考虑那些只属于局部组中的节点, 即局部校验节点. 具体做法是当某个局部组有节点故障时, Leader 节点将原先要分发给这些故障节点的日志分片打包, 然后封装成局部增补请求, 发送到该局部组的那些可用节点上, 这些可用节点被称为局部增补节点. 由于 LRCRaft 支持全局增补机制, 故在一个局部组中, 既属于局部组同时也属于全局组的那些节点 (即该局部组中的数据节点) 不再增补彼此的日志分片, 这部分日志分片已经在全局增补流程中完成增补, 不必重复处理.

如图 4 所示, 一个 $L(4, 2, 2, 1)$ 共识组的 8 个节点编号为 1–8.

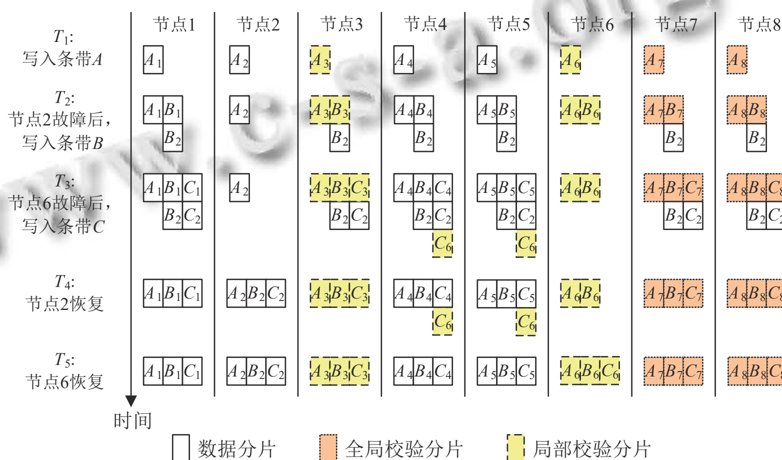

图 4 LRCRaft 节点的全局与局部增补

图 4 中, 节点{1, 2, 4, 5, 7, 8}构成全局组, 其中节点{7, 8}为全局校验节点, 其余为数据节点; 节点{1, 2, 3}构成局部组 1, 节点 3 为局部校验节点, 节点{4, 5, 6}构成局部组 2, 节点 6 为局部校验节点. T_1 时刻所有节点均可用, 日志条带 A 在各节点上对应的日志分片均成功写入; T_2 时刻节点 2 故障, 随后写入日志条带 B 时数

© 中国科学院软件研究所 http://www.c-s-a.org.cn

据日志分片 B_2 被增补到了节点{1, 3, 4, 5, 7, 8}上; T_3 时刻节点 6 故障, 随后写入日志条带 C 时局部校验日志分片 C_6 被增补到了节点{4, 5}上, 数据日志分片 C_2 被增补到了节点{1, 3, 4, 5, 7, 8}上; T_4 时刻节点 2 恢复, Leader 节点通知局部组 1 的节点将增补的数据日志分片{B_2, C_2}复制到节点 2 上, 待复制成功后再通知共识组其余节点删除增补日志分片{B_2, C_2}; T_5 时刻节点 6 恢复, Leader 节点通知局部组 2 的节点将增补的局部校验日志分片 C_6 复制到节点 6 上, 待复制成功后再通知局部组 2 其余节点删除增补日志分片 C_6.

不论是全局增补还是局部增补, 增补日志分片最后都要应用到各节点的本地状态机以确保存储系统正常运作. 对于客户端的新写入的数据, LRCRaft 都会将其封装成若干个日志条带, 并为每个日志条带生成一个唯一的逻辑地址 (logical address), 然后将逻辑地址写入日志条带中每个日志分片的共识协议头中. 当这些日志条带中的分片成功应用到各节点状态机后, 存储系统将在元数据仓库中记录这些逻辑地址, 最后向客户端返回这些逻辑地址, 客户端收到后就可以通过逻辑地址来访问存储系统中对应的数据. LRCRaft 共识组中的每个节点维护有一张逻辑地址-物理地址映射表 (简称逻辑地址映射表), 用以将存储系统的逻辑地址映射到其本地存储介质中实际的块地址. 在这张映射表上, 每个条目至少包含 3 个字段: 逻辑地址、偏移 (offset) 和物理块地址 (physical block address). 偏移指的是逻辑地址对应的分片在原日志条带中的编号, 该字段用来判定分片是否为增补分片.

因此对于一个分片, 通过查找节点的逻辑地址映射表, 即可找到与其相关的所有分片.

图 5 是一个 $L(4, 2, 2, 1)$ 共识组节点 4 的状态机数据, 该节点是数据节点, 存储 LRC 条带中编号为 4 的数据分片. 图 5 表示在共识组写入日志条带 A、B 和 C 时, 节点 4 对故障节点状态机数据的增补. 对于日志条带 A, 节点 4 状态机中只有数据分片 A_4, 说明写入该条带时未有增补发生或增补数据已被删减. 对于日志条带 B, 节点 4 状态机除了数据分片 B_4, 还增补了数据分片 B_2、全局校验分片 B_7 和局部校验分片 B_6. 对于日志条带 C, 节点 4 则增补了数据分片 C_2 和 C_5、局部校验分片 C_6. 图 5 中箭头表示通过在逻辑地址映射表中查找与 B_4 或 C_4 逻辑地址相同的分片, 即可找到日志条带 B 和 C 在节点 4 状态机上所有的对应增补分片.

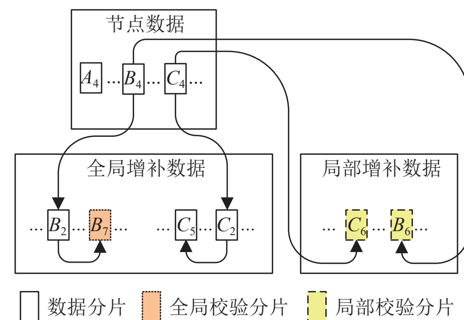

图 5 LRCRaft 节点状态机数据与增补数据

2.4 状态机删减

为实现最优的存储空间利用率, LRCRaft 要在故障节点恢复或者被新节点替换后, 对增补节点的额外分片进行删减, 删减对象包括节点存储介质中的增补日志分片和状态机数据. LRCRaft 的 Leader 节点利用心跳机制, 通过心跳包中的 commitIndex 和 lastApplied 字段, 可以很方便地追踪整个共识组各节点日志处理进度. 当一个日志条带被所有节点都确认写入完成并应用到状态机之后, 就可以开始删除各节点中与之相关的增补日志分片和状态机数据. 根据日志顺序提交规则, 当编号为 i 的日志分片被某个节点应用到状态机, 则编号小于 i 的所有日志分片也一定被应用完成. 因此一旦 Leader 节点确认某个日志条带已经被整个共识组应用成功, 则可以删减节点中所有编号小于等于该日志条带编号的增补日志分片和状态机数据. LRCRaft 通过扩展心跳机制实现对增补数据的删减, 由 Leader 节点发起一轮心跳 (AppendEntries 请求), 其中附带一个 lastSync 字段, 表示共识组最新的日志同步进度. 心跳包中 lastSync 和 lastApplied 作用不同: lastApplied 旨在于让节点知悉共识组日志应用进度, 以此决定是否要应用一些日志分片到本地状态机上; 而 lastSync 则表示共识组中被所有节点都应用完成的最新日志条带编号, 节点根据该字段检查本地存储介质中是否有编号在此之前的增补日志分片和状态机数据, 若有则进行删减. 通过状态机删减操作, 整个共识组的存储负载最终可以达到最小均衡——每个节点只存储本节点负责的数据而没有冗余数据, 各节点存储的数据量大小为客户端原始数据的 $1/k$.

2.5 分片版本一致性保障

动态日志增补机制的引入使 LRCRaft 成为一个延迟同步写协议, 即已向客户端回复写成功的数据因为节点故障、网络分区等情况存在, 并非总是在每个节

© 中国科学院软件研究所 http://www.c-s-a.org.cn

点上都被同步写入, 但增补机制保证数据最终在每个节点上都会被成功写入. 该特性使得 LRCRaft 面临一个不可避免的问题: 新旧数据共存. LRCRaft 可能会遇到这样的一个情景: 在某个节点故障期间, 客户端对一些数据进行了更新, 本应存储在该节点的日志分片被其余节点增补存储. 随后该节点从故障中恢复, 暂未收到其余节点发送的对应增补日志分片. 若此时客户端下发一个读请求, 读取先前更新的部分数据, 则该节点就会将状态机中目标分片发送给 Leader 节点进行响应. Leader 节点将收到部分过时的分片, 用这部分分片参与解码以恢复客户端数据将会产生错误, 从而白费计算资源和时间. 发生这种情况 Leader 节点也无法判定是哪个分片有问题, 只能向共识组中更多的节点请求对应分片, 以保证最终有足够多的正确分片完成解码. 这个问题不止在从故障中恢复的节点上存在, 实际测试中发现在慢节点上也存在. 这是对网络带宽的一种浪费, 更优的策略是保证从共识组中读取的分片都是可信的、最新的分片.

LRCRaft 引入分片版本管理机制以解决此问题, 该机制的核心在于为每个分片保存一个版本号. Leader 节点请求某个分片时除了逻辑地址, 还需附上版本号, 其余节点收到请求后, 通过比对版本号即可知晓自身存储的分片是否过时, 从而决定向 Leader 节点返回什么样的内容. 对于已过时的分片, 节点对读请求的回应将不带有任何数据, 并将回复消息中的 outdated 标识设为真. Leader 节点收到空的请求回应后即可重新安排读取请求的目标节点, 如此操作既可以避免浪费系统网络带宽, 同时也让 Leader 节点无需额外处理过时分片, 减轻了 Leader 节点的职责, 能有效提高存储系统的数据处理速度. LRCRaft 将版本号和逻辑地址一同写入每个日志分片的共识协议头中, 最后写入到节点的逻辑地址映射表中. 对于客户端新写入的数据, 其对应的日志条带中每个日志分片版本号从 1 开始; 对于被更新的数据, 版本号则是 Leader 节点对应分片的本地版本号加 1, 读操作不改变分片的版本号.

至此 LRCRaft 有了逻辑地址映射、分片版本追踪等特性. 采用逻辑地址的好处是将软件层面日志条带处理逻辑和硬件层面的数据存储逻辑解耦, 软件层面使用统一的逻辑地址管理分片, 不受实际物理介质、配置结构的影响. 各节点能更灵活地配置其存储后端, 每个节点存储池用多少种、多少个存储介质, 每个存储介质容量大小都可根据实际情况进行调整, 只要保证节点能够正确寻址即可. 实现这一切的基础就是节点的逻辑地址映射表, 该表的关键属性设计如图 6 所示.

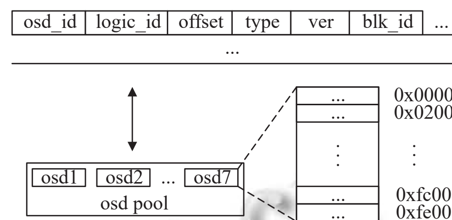

图 6 节点逻辑地址映射表部分属性

图 6 中, osd_id 表示分片存储在哪个 OSD (object-based storage device) 上; logic_id 表示分片所处日志条带的逻辑地址, 同一日志条带上的不同分片共用一个逻辑地址; offset 表示分片在日志条带中的编号; type 表示分片类型, 包括数据分片、全局校验分片和局部校验分片 3 种; ver 表示该分片的版本; blk_id 表示该分片存储在 OSD 中的物理块地址.

2.6 数据读取请求处理

LRCRaft 不经过日志机制处理客户端的数据读取请求, 而是通过对状态机数据进行直接读取、校验和封装的方式返回给客户端. 基于多副本的 Raft 模型中, 由于每个节点存储的都是完整数据, 因此数据读取请求只需通过 Leader 节点即可完成. LRCRaft 中每个节点的状态机是非完整状态机, 因此读取数据需要至少 k 个节点的参与, 这个过程称为恢复读 (recovery read). LRCRaft 读取数据可以分为两类情景: 对完备数据的读取和对增补数据的读取.

被全局组中的每个节点都应用到其自身状态机的日志分片对应的数据被称作完备数据. 对该类数据的读取不需要进行额外处理, 只需要 Leader 节点将读取分片的逻辑地址和版本号分发给全局组中的节点即可, 这些节点从状态机读取对应分片时会进行一次版本校验, 确认无误后再向 Leader 节点返回分片. 当 Leader 节点收到 k 个 (含 Leader 节点本地的 1 个) 分片后会对这些分片进行二次校验, 确保在网络传输过程中没有出现问题, 最后经过解码还原出完整的数据返回给客户端.

与完备数据对应的是增补数据, 并非每个全局组中节点的状态机上都有要读取的目标分片, 此种情况一般是节点故障导致, 此时读取数据要涉及其余节点的增补数据. Leader 节点首先对全局组中的所有节点

© 中国科学院软件研究所 http://www.c-s-a.org.cn

发送一轮读取请求, 各节点收到请求后查找本地状态机, 此时可能会出现 3 种情况: A 类节点成功找到目标分片, 且没有找到增补分片; B 类节点成功找到目标分片及一系列增补分片; C 类节点未找到目标分片. 此时 A 和 B 类节点将向 Leader 节点返回目标分片, 且 B 类节点附带其存储的增补分片在日志条带中的编号一同返回, C 类节点则返回目标分片未找到. 经过一轮应答, Leader 节点对收到的目标分片进行排序, 当收到的目标分片小于 k 个, 无法进行数据还原时, Leader 节点就会向 B 类节点发送增补读取请求, 附带目标分片的逻辑地址、版本号和缺少的分片编号. B 类节点收到增补读取请求后, 立即向 Leader 节点返回所有符合条件的增补分片, Leader 节点收到回复后使用增补分片和上一轮读取的结果解码出完整的数据并返回给客户端. 理论上 Leader 节点只需要收到一个 B 类节点的增补数据就可以完成对数据的还原, 因此为了降低时延、减少网络拥塞, Leader 节点会根据前一轮的读取请求应答的先后顺序, 选择响应速度最快的那个 B 类节点发送增补读取请求. 若该 B 类节点响应超时, 则依次选择后面的 B 类节点重试.

2.7 快速数据恢复

节点故障是分布式存储系统无法避免的一类问题, LRCRaft 采用 LRC 码正是为了减少故障节点恢复所需的时间, 在 LRCRaft 中, 节点故障分为两类情况: 瞬时性故障和永久性故障.

节点瞬时性故障指的是由于某些原因 (如网络故障、节点重启等) 造成的节点短时间不可用, 这类故障不会导致该节点存储的历史数据丢失, 故障持续时间通常在数分钟到数小时之间, 且故障原因消除后节点可自行恢复正常工作状态. 当全局组中有一个节点 X 恢复后, Leader 节点会根据心跳回应确认它的落后程度, 然后向全局组和节点 X 所属局部组的其余节点发送一个数据恢复请求. 在收到来自 Leader 节点的数据恢复请求后, 这些节点会查找本地的日志, 将与节点 X 相关的增补日志分片编号打包返回给 Leader 节点. Leader 节点根据各节点的回复以及节点响应时间等要素, 动态生成一个计划恢复序列, 该序列规定由哪个节点向节点 X 发送哪些增补日志分片, 并将这个序列发送给这些节点, 这些节点称为辅助节点 (assistant node). 收到恢复序列的辅助节点会按照计划有序地向节点 X 发送增补日志分片, 节点 X 收到来自不同辅助节点的增补日志分片后要进行重排序、应用到状态机等操作. 同时节点 X 和辅助节点通过响应心跳定期向 Leader 节点汇报恢复进度. 在数据恢复过程中, 如果发生了 Leader 节点切换, 新的 Leader 节点可以根据心跳响应获悉之前的恢复进度, 还可以与辅助节点通信获取上一任 Leader 安排的恢复序列. 如果恢复期间辅助节点发生了故障, Leader 节点可以根据数据恢复的进度动态调整恢复序列.

节点永久性故障通常是由存储介质故障导致的, 当某个节点的存储介质发生不可逆损毁后, 该节点处于不可用状态且无法自行恢复, 一段时间后系统巡检流程会发现节点存储介质故障并进行通报. 此类故障的标准应对措施是管理员更换存储介质并在存储系统上做好配置, 让系统识别并使用新的存储介质. 由于更换后的存储介质上没有历史数据, 因此需要将原来丢失的数据恢复到新存储介质上. 在 Raft 中, Leader 节点将完整日志快照发送到要恢复数据的节点上, 由恢复节点将快照应用到新的本地状态机即可, 整个过程仅涉及这一对节点. 引入 EC 后, 由于每个节点没有完整的日志, 也没有完整的日志快照, 因此恢复数据需要若干个节点的参与. 以 $RS(k, m)$ 为例, 恢复一个节点丢失的历史数据, 需要其余 k 个节点的全程参与, 较大的历史数据量会导致恢复过程相当耗时, 且参与节点过多容易导致存储系统网络拥塞, 使系统处于一个较为脆弱的状态: 容错率将会降低、读写时延将会升高.

LRCRaft 在发生节点永久性故障后, 依靠增补日志分片恢复机制已无法恢复数据, 因为无法保证待每个目标日志分片在其他节点上都有增补, 从而无法保证数据完整性. 为了降低恢复时长, LRCRaft 优先利用局部组进行数据恢复. 假设现有一个节点 X 发生了永久性故障, 损坏的存储介质已被替换, 该节点通过心跳告知 Leader 节点其已可以正常接收数据, 同时报告了丢失的数据详情. Leader 节点收到后, 立即向节点 X 所在的局部组发起一轮数据恢复请求, 将节点 X 需要的数据告知该局部组其余节点, 这些节点收到消息后立即向节点 X 发送存储在本节点的历史日志分片、快照或增补日志分片等数据. 假设该局部组由 k_1 个数据节点和 m_1 个局部校验节点构成, 则节点 X 收到 k_1 个日志分片即可计算出完整日志条带, 从而提取出丢失的日志分片. 整个数据恢复过程由局部组节点相互通信完成, 无需其余节点参与, 局部组节点只需在心跳响应

© 中国科学院软件研究所 http://www.c-s-a.org.cn

中定期向 Leader 节点返回恢复进度即可.

利用局部组可以减少恢复一个节点数据所需要参与的节点数量, 尽可能降低对系统的影响, 并且更少的辅助节点数量意味着更短的恢复时间和更低的带宽占用. 然而 LRCRaft 也不得不面临一个问题: 当某个局部组的节点故障数量超过其数据恢复能力的阈值时, 系统只能降级为普通的 MDS 恢复过程, 即通过全局组中的 k 个节点来恢复故障节点的数据, 该过程与目前主流的 EC+Raft 的数据恢复方案一致. 为使系统尽快度过这一段降级恢复的脆弱时期, LRCRaft 采用了 PPR (partial parallel repair)[16]方式优化恢复过程的数据流. PPR 无法降低总体网络带宽消耗, 但通过降低瞬时带宽峰值, 可以起到减少网络拥塞、加速恢复的作用. 一旦有部分全局组节点恢复完成, 且达到让其所在的局部组可以独立进行数据恢复的条件后, 系统就会使该局部组内的其余故障节点转入局部组恢复过程, 从而使整个系统尽快脱离降级恢复状态, 降低整体性风险.

2.8 可用性级别

由于 LRCRaft 引入了局部组的概念, 故对其可用性级别的衡量与其他的共识协议有些许差别. 在一个 $L(k, m, n, r)$ 共识组中, 理论上可以容忍的节点故障数量为: $m+n\cdot r$, 即 m 个全局组节点和所有的局部校验节点. 如果按照 Raft 的可用性衡量标准, 对齐 LRCRaft 和 Raft 的共识组大小和可用性, 则有:

$$2F+1=k+m+n\cdot r \tag{4}$$

$$F=m+n\cdot r \tag{5}$$

根据式 (4)、式 (5) 可得, 当 $k=m+n\cdot r+1$ 时, LRCRaft 拥有和 Raft 相同的可用性级别.

LRCRaft 的编码配置参数可以灵活调整, 结合很多实际系统的 EC 配置, 使用 LRCRaft 实现和 Raft 相同的可用性级别相当容易. 例如 Facebook 在 Tectonic 中采用的 $RS(10, 4)$ 或 $RS(9, 6)$ 编码配置, 通过适配可以转换为 $L(10, 4, 5, 1)$ 或 $L(9, 6, 3, 1)$. 表 1 列出了上述配置的 LRCRaft 可用性级别与同共识组大小的 Raft 协议可用性级别, 可见 LRCRaft 的可用性级别已和 Raft 持平, 因此可以认为 LRCRaft 是一个可用性级别与 Raft 相当的共识协议.

表 1 共识组大小相同的 Raft 和 LRCRaft 协议可用性级别

LRCRaft配置	LRCRaft可用性	Raft可用性
$L(10, 4, 5, 1)$	9	9
$L(9, 6, 3, 1)$	9	9

3 性能测试实验与结果分析

3.1 实验环境搭建

为了测试 LRCRaft 的实际性能表现, 本文用 C 实现了一个以块为最小单位的分布式存储原型系统, 该系统支持不同的共识协议以方便进行比较. 实验以相同的可用性为前提设定不同的协议节点数量配置, 考虑到 Raft 与 ECRaft 无局部组的概念, 故仅将 LRCRaft 的全局组配置作为可用性的衡量标准 (LRCRaft 的全局组、Raft 和 ECRaft 的整个共识组均可容忍两个节点故障), 各协议采用的存储节点配置如表 2 所示.

表 2 不同共识协议下存储节点配置

共识协议	数据节点数量	校验节点数量	节点配置
Raft	5	0	$Raft(3, 2)$
ECRaft	4	2	$EC(4, 2)$
LRCRaft	4	4	$L(4, 2, 2, 1)$

网络带宽和硬件配置方面, 每个节点配有 2.5 Gb/s 的网络带宽、8 个 CPU 核心 (8×5.10 GHz)、8 GB 内存和单块容量为 2 TB 的 Pcie 4.0 SSD (4 KB 随机读 IOPS: 1 200k、4 KB 随机写 IOPS: 1 100k、7 300 MB/s 顺序读取和 6 600 MB/s 顺序写入), 节点运行的操作系统是 Ubuntu Server 22.04 LTS.

软件设计方面, 存储节点采用 SPDK[17]的 bdev 块设备作为底层存储后端驱动, 以实现对裸块设备进行直接读写. EC 编码则是采用 SPDK 的 ISA-L 子模块实现, 节点间通信的 RPC 采用 Libuv[18]实现.

系统软件架构如图 7 所示, 架构分层包括客户端层、管理层、分布式协议层、持久化层、存储协议层、通用块设备层和物理介质层. 客户端层使用 FIO[19]作为压测工具, 对存储系统进行性能测试. 分布式协议层运行着不同共识协议管理的存储节点集群, 采用可插拔的设计, 与上下两层通过通用接口进行对接, 实验中仅需对该层提供不同的实现即可完成对 Raft 协议、ECRaft 协议和 LRCRaft 协议的对比测试. 管理层运行着管理节点, 起到维护存储节点集群正常运行的作用. 管理节点负责实时监控各存储节点状态并生成存储节点集群视图 (用来描述存储节点集群每个节点的状态), 一旦有视图变更 (节点故障、恢复事件均会触发变更) 就会立即通知存储节点集群中的 Leader 节点, 使其节点能够及时调整与其他存储节点的通信策略. 此外管理节点还负责接受客户端的请求, 解析并转发给 Leader 节点处理, Leader 节点返回给客户端的消息也

© 中国科学院软件研究所 http://www.c-s-a.org.cn

由管理节点转发. 实验主要关注几个性能指标: 不同协议下存储系统读、写操作的平均时延、吞吐量以及故障节点恢复所需时间.

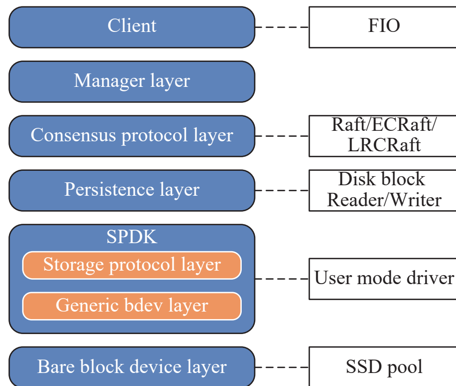

图 7 分布式存储原型系统的软件架构设计

3.2 读写时延测试

实验评估了各协议在不同块大小下随机写入 100 个块的平均时延. 图 8 表示在没有节点故障的情况下, 各协议的平均写时延. 结果表明随着块大小的提升, ECRaft 和 LRCRaft 比 Raft 分别有着 20.89%–67.57% 和 20.91%–54.66% 的写时延降低. 在块大小较小时, 影响系统时延的主要因素是硬盘 IOPS, 由于数据量过小, LRCRaft 和 ECRaft 在时延方面的优势无法在高速存储介质体现出来. 随着块大小的增加, 系统占用网络带宽随之增加, 此时网络时延和硬盘吞吐量共同影响着存储系统的写时延. 在块大小超过 128 KB 以后, ECRaft 和 LRCRaft 在写时延方面对 Raft 有着显著优势, 该优势随着块大小的增大而进一步扩大. 在块大小超过 512 KB 后, ECRaft 有着比 LRCRaft 更加低的写时延. 虽然 ECRaft 和 LRCRaft 都是基于纠删码的同步写协议, 但 LRCRaft 引入了局部组, 写一个块需要进行一次全局编码和两次局部编码, 且经过网络发送的分片也更多, 因此在块大小增加到一定程度后写时延要低于 ECRaft.

图 9 表示在单节点故障时, ECRaft 和 LRCRaft 比 Raft 分别有着平均 35% 和 30% 左右的写时延降低. 在单节点故障时各协议在不同块大小下的写时延的整体变化趋势和无节点故障时相同, Raft 依然是平均写时延最高的协议, 但和无节点故障情况对比可以发现 Raft 和其他两个协议写时延的绝对差距明显减小, 且 Raft 的写时延也相对有了提高. 出现该现象的原因是在系统管理节点发现有节点故障后将告知 Leader 节点最新的视图, Leader 节点根据最新的视图来优化共识组日志同步: Leader 节点不再尝试将日志发送给故障节点, 直到获取到新视图表明故障节点已经恢复为止. 该机制的引入使得 Raft 得以减少在网络中发送的数据量, 最终降低写时延. ECRaft 和 LRCRaft 虽采用同样的机制, 但由于这两个协议增补机制的存在, 写入一条日志时 ECRaft 需要在共识组中写入 9 个分片 (无节点故障时只需写 6 个分片), LRCRaft 需要在共识组中写入个 13 分片 (故障节点为全局数据节点, 无节点故障时只需写 8 个分片). 增补机制增加了网络中发送的数据量, 从而降低了 ECRaft 和 LRCRaft 的写时延.

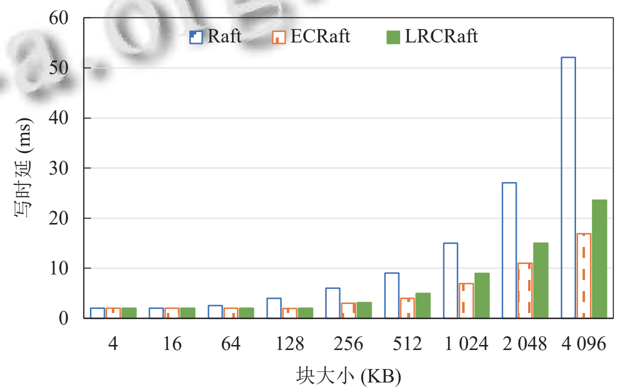

图 8 无节点故障时各协议在不同块大小下的写时延

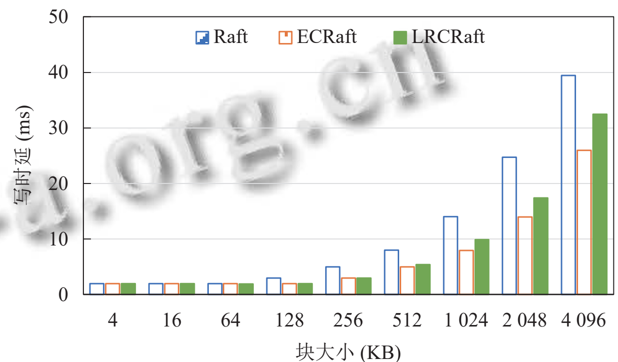

图 9 单节点故障时各协议在不同块大小下的写时延

图 10 表示在两节点故障时, ECRaft 和 LRCRaft 比 Raft 分别有着最多 12.20% 和 36.67% 的写时延上升. 在两节点故障下, ECRaft 和 LRCRaft 的写时延在块大小超过 256 KB 后反而不如 Raft. 实验中 LRCRaft 故障的节点均为全局数据节点, 此场景下由于增补的分片数量最多, LRCRaft 写入一条日志需要写入 16 个分片, 此时写性能将达到理论最低点. ECRaft 写入一条日志需要写入 12 个分片, 故这两个协议的网络带宽开销进一步增加. Raft 则由于仅剩 3 个可用节点, 相较于

© 中国科学院软件研究所 http://www.c-s-a.org.cn

4、5 个可用节点的情况反而降低了同步日志的网络带宽开销. 此消彼长之下, Raft 在块大小持续增大的情况下反而成为写时延最低的协议.

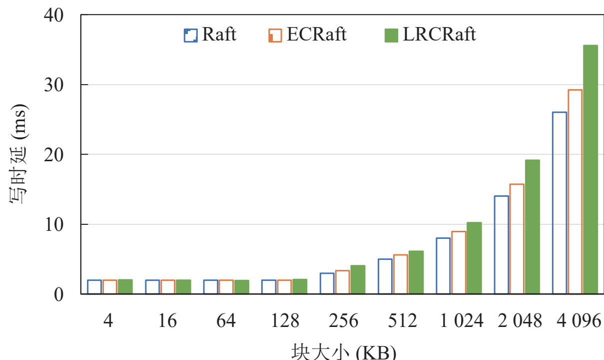

图 10　两节点故障时各协议在不同块大小下的写时延

图 11 是各协议在不同块大小下随机读取 100 个块的平均读时延测试结果. 结果表明在不同块大小下, ECRaft 和 LRCRaft 的读时延均高于 Raft, 且 LRCRaft 和 ECRaft 的读时延几乎相等, 平均读时延是 Raft 的 130% 左右. 此结果符合各协议读取数据的设计: Raft 无需经过其余节点, 直接由 Leader 节点向客户端返回数据, 故在网络中带宽消耗最少, 读取时延最低; ECRaft 和 LRCRaft 都需要至少通过 k 个节点将数据 (或校验) 分片从共识组中读取出来, 再经 Leader 节点恢复原始数据后方可向客户端返回, 此过程需要消耗额外的带宽, 故时延高于 Raft. LRCRaft 读取数据仅从全局组中进行读取, 其配置与 ECRaft 并无差异, 读时延亦不存在明显差异.

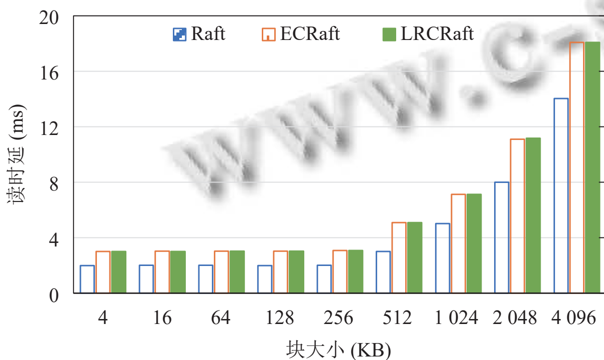

图 11　各协议在不同块大小下的读时延

3.3　吞吐量测试

实验评估了各协议在不同块大小下连续写入 100 个块的平均写吞吐量. 图 12 表示无节点故障时, ECRaft 和 LRCRaft 比 Raft 分别有着最多 208.31% 和 120.55% 的写吞吐量提升; 图 13 表示单节点故障时, ECRaft 和 LRCRaft 比 Raft 分别有着最多 69.29% 和 68.77% 的写吞吐量提升; 图 14 表示两节点故障时, ECRaft 和 LRCRaft 比 Raft 分别有着最多 10.88% 和 26.83% 的写吞吐量下降. 随着块大小的提升, 各协议写吞吐量也在提升. 无节点故障时, Raft 写吞吐量随块大小增大的收敛速度最快, 这是因为 Raft 的 Leader 节点并发向其余节点发送的日志数据量最大, 块大小的增加最先对 Leader 节点出口端造成网络拥塞. 其余两个协议则由于暂时没有触碰到节点网络带宽上限, 因而写吞吐量得以保持上升态势.

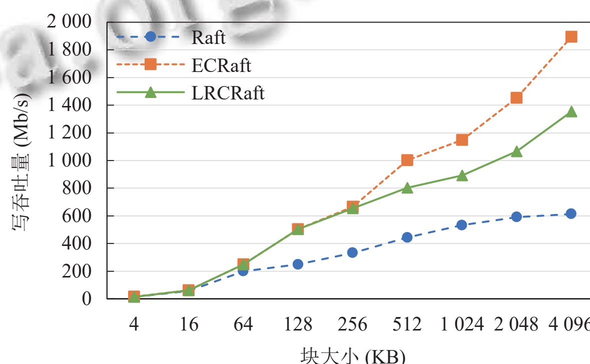

图 12　无节点故障时各协议在不同块大小下的写吞吐量

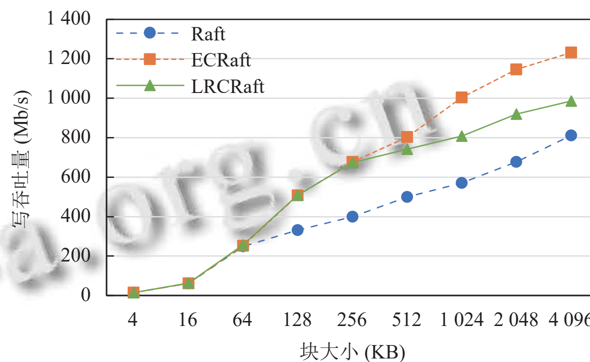

图 13　单节点故障时各协议在不同块大小下的写吞吐量

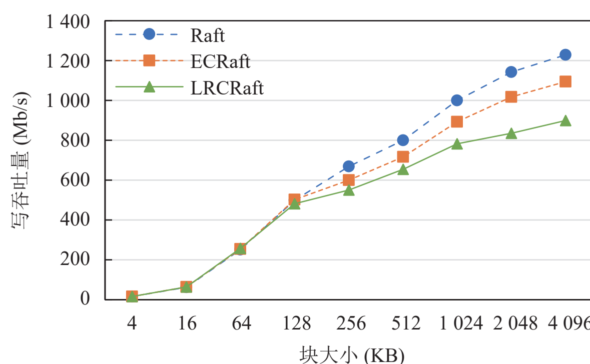

图 14　两节点故障时各协议在不同块大小下的写吞吐量

© 中国科学院软件研究所 http://www.c-s-a.org.cn

对比无节点故障情况, Raft 在单节点故障和两节点故障时的写吞吐量均有所上升, 且在两节点故障时写吞吐量超过 ECRaft 和 LRCRaft. 其原因和第 3.2 节中测试写时延时相同: 处理节点故障的增补机制加重了 ECRaft 和 LRCRaft 的网络负担, 较少的可用节点反而使得 Raft 的网络开销有所降低.

图 15 是各协议在不同块大小下连续读取 100 个块的平均读吞吐量测试结果, 该结果与第 3.2 节的各协议读时延结果类似. 受恢复读机制的影响, ECRaft 和 LRCRaft 读吞吐量均比 Raft 低, 且 ECRaft 和 LRCRaft 读吞吐量不存在差异, 对比 Raft 均有最多 40.91% 的读吞吐量下降.

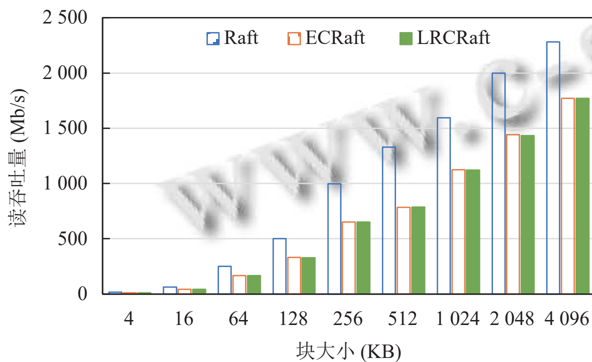

图 15 各协议在不同块大小下的读吞吐量

3.4 故障恢复时间测试

为了完整测试各个协议对于节点故障恢复的能力, 本实验针对 ECRaft 和 LRCRaft 协议设定了两个场景: 有增补恢复和无增补恢复, 分别对应前文提及的节点瞬时性故障和永久性故障. 有增补恢复指的是某个节点从故障中恢复后, 共识组中一些节点拥有该节点故障期间应存储的全部数据, 无增补恢复则是共识组节点没有增补数据. 对于 Raft 的节点恢复, 可以看作有增补恢复的一种, Leader 节点保证拥有恢复的节点故障期间的完整数据, 只不过这些数据不是经编码过的分片数据.

实验评估了有增补和无增补两种恢复模式下, 各协议恢复一个故障节点数据所用时间, 针对不同的块大小, 统一控制故障节点落后于共识组进度 1 024 个块, 随后对其进行数据恢复. 图 16 表示在有增补恢复模式下, ECRaft 和 LRCRaft 恢复用时几乎相等, 比 Raft 有着最多 74.97% 的恢复用时减少; 图 17 表示在无增补恢复模式下, ECRaft 和 LRCRaft 比 Raft 分别有着最多 15.62% 和 49.25% 的恢复用时减少.

在有增补数据时, ECRaft 和 LRCRaft 恢复故障节点数据用时要显著低于 Raft, 前两者整个网络中传输的数据量大小是 Raft 协议的 1/4, Raft 恢复数据需要由 Leader 节点将完整的数据发送给目标节点, 因而用时较长. 而在无增补数据时, ECRaft 和 LRCRaft 的恢复用时相较有增补数据时均有上升, 其中 ECRaft 恢复用时受影响更大. 此情景下 ECRaft 恢复数据需要进行重编码, Leader 节点将从共识组的其余 3 个节点中读取分片, 重新编码生成目标分片再发给目标节点. 此时 ECRaft 在整个网络中传输的总数据量已经等于 Raft, 只是由于读取数据时采用并发读取方式, 故恢复时间略少于 Raft, 但也已经较为接近. LRCRaft 则是进行局部组恢复, 局部组中的两个节点同时向目标节点发送分片, 无需 Leader 节点作为中间节点传输数据, 且在网络中传输的数据总量只有 Raft 的一半, 保证了在 3 个协议中最少的数据恢复时间.

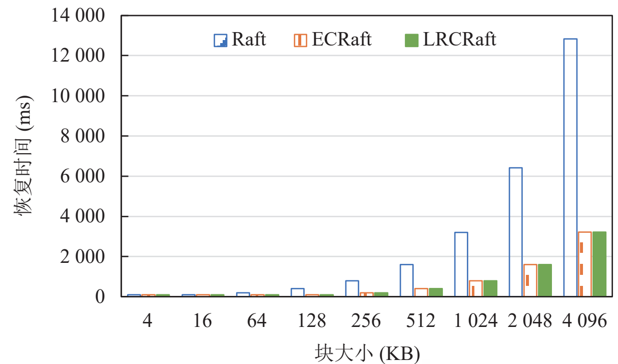

图 16 各协议在不同块大小下的有增补恢复用时

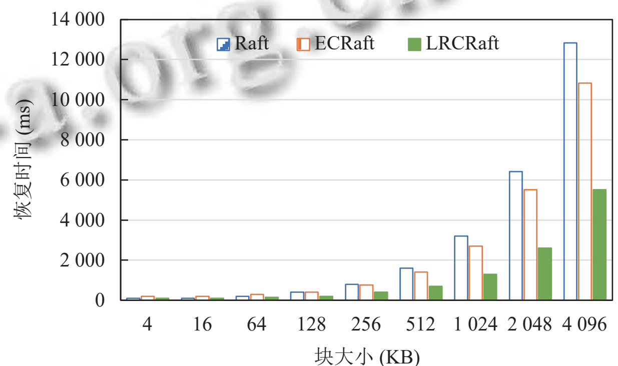

图 17 各协议在不同块大小下的无增补恢复用时

3.5 总结

LRCRaft 在写时延、写吞吐量等性能指标上对比 Raft 有明显提升, 对比 ECRaft 稍逊一筹. 在单节点故障下, LRCRaft 比 Raft 有着更加优秀的性能表现; 在两节点故障情况下, 就测试使用的节点配置 $L(4, 2, 2, 1)$ 而言, LRCRaft 的读写性能表现不如 Raft. 但可以预测的是, 采用更长 LRC 编码配置的 LRCRaft 共识组将拥

© 中国科学院软件研究所 http://www.c-s-a.org.cn

有更强的节点容错能力, 为一个节点进行增补的成本也会更低 (分片越多, 单分片体积越小), 在 LRCRaft 共识组中采用长编码会更有优势. 另外相较于其他共识协议, LRCRaft 拥有更优秀的节点恢复性能, 部署到真实的分布式存储系统中可以很好地解决大量数据恢复耗时长的痛点.

4 结束语

本文提出了一个将 Raft 和 LRC 码相结合的共识协议——LRCRaft. LRCRaft 设计的意图在于提供较高的空间利用率、较低的读写时延以及缩短节点故障恢复时间. 实验结果表明, 在无节点故障和单节点故障的情况下, LRCRaft 的写时延和写吞吐量均优于 Raft, 虽整体写性能上稍逊于 ECRaft, 但在故障恢复时间这一性能指标上 LRCRaft 对比 Raft 和 ECRaft 有着巨大优势, 这也是 LRCRaft 设计的侧重点所在. 后续研究可以考虑进一步优化 LRCRaft 的写性能, 一个可行思路是结合 EC 条带增量更新的特性. 对 EC 更新机制的相关研究表明, 利用 EC 计算的数学特征, 可以采取不同的方式对编码条带实现增量更新[20]. 因为 LRC 编码过程要进行多次 EC 编码, 故该成果也可以适配到 LRC 上来, 使 LRCRaft 支持编码条带增量更新的特性, 理论上可以进一步降低特定场合下的写时延.

参考文献

1 Reed IS, Solomon G. Polynomial codes over certain finite fields. Journal of the Society for Industrial and Applied Mathematics, 1960, 8(2): 300–304. [doi: 10.1137/0108018]

2 Lin WK, Chiu DM, Lee YB. Erasure code replication revisited. Proceedings. Proceedings of the 4th International Conference on Peer-to-peer Computing, 2004. Zurich: IEEE, 2004. 90–97.

3 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 (FAST 21). 2021. 217–231.

4 Huang C, Simitci H, Xu YK, *et al*. Erasure coding in windows azure storage. Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC 12). Boston: USENIX Association, 2012. 15–26.

5 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 (MSST). Incline Village: IEEE, 2010. 1–10.

6 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.

7 Ongaro D, Ousterhout JK. In search of an understandable consensus algorithm. Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 2014). Philadelphia: USENIX Association, 2014. 305–319.

8 Ongaro D. Consensus: Bridging theory and practice [Ph.D. Thesis]. Stanford: Stanford University, 2014.

9 Guruswami V, Jin LF, Xing CP. Constructions of maximally recoverable local reconstruction codes via function fields. IEEE Transactions on Information Theory, 2020, 66(10): 6133–6143. [doi: 10.1109/TIT.2020.2988459]

10 Mu S, Chen K, Wu YW, *et al*. When Paxos meets erasure code: Reduce network and storage cost in state machine replication. Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing. Vancouver: ACM, 2014. 61–72.

11 Lamport L. The part-time parliament. In: Malkhi D, ed. Concurrency: The Works of Leslie Lamport. New York: ACM, 2019. 277–317.

12 Lamport L. Paxos made simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001), 2001. 51–58.

13 Wang ZZ, Li TL, Wang HX, *et al*. CRaft: An erasure-coding-supported version of raft for reducing storage cost and network cost. Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST 20). Santa Clara: USENIX Association, 2020. 297–308.

14 Jia YL, Xu GP, Sung CW, *et al*. HRaft: Adaptive erasure coded data maintenance for consensus in distributed networks. Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS). Lyon: IEEE, 2022. 1316–1326.

15 Xu MW, Zhou Y, Qiao YY, *et al*. ECRaft: A raft based consensus protocol for highly available and reliable erasure-coded storage systems. Proceedings of the 27th IEEE International Conference on Parallel and Distributed Systems (ICPADS). Beijing: IEEE, 2021. 707–714.

16 Mitra S, Panta R, Ra MR, *et al*. Partial-parallel-repair (PPR): A distributed technique for repairing erasure coded storage. Proceedings of the 11th European Conference on Computer Systems. London: ACM, 2016. 30.

17 The storage performance development kit (SPDK). https://github.com/spdk/spdk. [2024-01-10].

18 Libuv is a multi-platform support library with a focus on asynchronous I/O. https://github.com/libuv/libuv. [2024-01-10].

19 fio. https://github.com/axboe/fio. [2024-01-10].

20 Zhang FH, Huang JZ, Xie CS. Two efficient partial-updating schemes for erasure-coded storage clusters. Proceedings of the 7th IEEE International Conference on Networking, Architecture, and Storage. Xiamen: IEEE, 2012. 21–30.

(校对责编: 张重毅)

© 中国科学院软件研究所 http://www.c-s-a.org.cn