计 算 机 研 究 与 发 展
Journal of Computer Research and Development

https://crad.ict.ac.cn
63(3): 585−596, 2026

本期推荐

一种远程直接内存访问网络中的高效分布式锁协议

高 健 舒继武

（清华大学计算机科学与技术系 北京 100084）

（gaoj20@mails.tsinghua.edu.cn）

An Efficient Distributed Lock Protocol in Remote Direct Memory Access Network

Gao Jian and Shu Jiwu

（*Department of Computer Science and Technology*, *Tsinghua University*, *Beijing* 100084）

Abstract Distributed lock is a crucial component in distributed storage systems. The performance of the lock protocol significantly influences the overall performance of the entire system. Remote direct memory access (RDMA) is an emerging data center networking technology that supports one-sided communication verbs and offers low CPU overhead, low latency, and high throughput. It presents new opportunities for designing high-performance distributed lock protocols. However, designing such RDMA based network protocols faces significant challenges, for example scalability and fairness. We address these challenges by proposing FeLock, an RDMA-based distributed lock protocol. FeLock achieves high performance while tackling the above challenges by leveraging different RDMA communication verbs, which enables the clients to communicate directly with both the server for lock acquisition and release, and other clients to hand over lock ownership. Specifically, first, to improve performance, FeLock introduces a per-node lock management mechanism to reduce network roundtrips on the critical paths of the lock protocol. Second, to achieve scalability, FeLock incorporates a round-robin handover mechanism, in which nodes are logically organized into a ring, and clients hand over lock ownership sequentially according to their positions within the ring. Third, to ensure fairness and prevent clients from starvation, FeLock employs a node credit mechanism that limits the number of consecutive lock acquisitions by any single node, thereby preventing clients on any node from being indefinitely blocked by others. Experimental results demonstrate that FeLock achieves performance comparable to or exceeding that of existing one-sided RDMA lock protocols, such as DSLR, while exhibiting better fairness and scalability. With 3 to 120 clients, FeLock achieves throughput 1.01 to 7.51 times of DSLR, with its fairness improved by up to 2.24 times.

Key words distributed system; concurrency control; distributed lock; remote direct memory access (RDMA); fairness

摘 要 分布式锁是分布式存储系统的重要组件，锁协议的性能对系统整体的性能有关键性影响。远程直接内存访问（remote direct memory access, RDMA）是一种新兴的数据中心网络技术，它支持单边网络通信原语，可以降低系统CPU开销，同时具备低延迟、高吞吐的性能特性，为设计高速分布式锁协议提供了新机遇。然而，设计基于RDMA的分布式锁协议面临诸多挑战。着重在保证高性能的前提下解决扩展性和公平性挑战，提出一种RDMA网络中的高性能分布式锁协议FeLock，它利用多种类型的RDMA网络通信原语，使客户端不仅能与服务端通信加解锁，还能与其他客户端直接通信以移交锁所有权，同时实现

收稿日期：2025-05-29；**修回日期**：2025-10-16

基金项目：国家重点研发计划项目（2024YFB4505201）；国家自然科学基金项目（U22B2023）

This work was supported by the National Key Research and Development Program of China (2024YFB4505201) and the National Natural Science Foundation of China (U22B2023).

通信作者：舒继武（shujw@tsinghua.edu.cn）

了高性能、公平性和性能的扩展性。具体地,为保证高性能,FeLock 引入了节点粒度锁管理机制,缩减锁协议在关键路径上的网络往返次数。为实现扩展性,FeLock 引入了轮转移交机制,将所有节点排成 1 个环,客户端按照其在环中的顺序依次移交锁的所有权。为实现公平性和避免客户端饥饿,FeLock 引入了节点信用机制,限制节点连续加锁的次数,避免其他节点上的客户端无法加锁。实验显示,FeLock 相比于现有单边 RDMA 锁协议(如 DSLR)表现出相似或更高的性能,并且具有更好的公平性和扩展性。在 3~120 个客户端的环境下,FeLock 的吞吐量是 DSLR 的 1.01~7.51 倍,公平性提升至多 2.24 倍。

关键词 分布式系统;并发控制;分布式锁;远程直接内存访问;公平性

中图法分类号 TP302.1

DOI: 10.7544/issn1000-1239.202550361 **CSTR:** 32373.14.issn1000-1239.202550361

远程直接内存访问(remote direct memory access, RDMA)技术是一种新兴的数据中心网络技术,它除了支持传统的发送、接收这类双边(two-sided)通信操作以外,还允许单边(one-sided)通信操作,即某一节点可以直接访问另一节点上的内存,期间后者的 CPU 不参与、不感知。RDMA 具备高性能的特点,能提供微秒级别的访问延迟[1]和每秒数亿个网络包级别的处理速率[2]。因此,相较于传统内核 TCP/IP 网络通信操作,使用 RDMA 单边通信操作(以下简称“单边 RDMA”)既可以降低 CPU 资源开销,也能显著提升系统的性能。

并发能力是分布式存储系统的重要能力,锁是实现并发能力的底层机制。在分布式文件系统、键值存储系统、关系型和非关系型数据库系统中,分布式锁均是实现各种类型的分布式事务的基础[3-5],锁协议的性能对系统整体的性能有关键性的影响。RDMA 技术所提供的高性能为改善分布式锁协议的性能提供了新的机遇:受益于 RDMA 技术本身的低延迟和单边通信操作提供的低 CPU 开销,锁协议的加解锁关键路径延迟可得到有效降低,这将缩短分布式存储系统 I/O 操作关键路径长度和降低服务端 CPU 压力。

然而,构建基于单边 RDMA 的分布式锁协议面临诸多挑战。具体地,分布式锁协议难以在保证高性能的同时保证扩展性和公平性。1)扩展性指的是当客户端数目增加时,在冲突不严重的情况下,系统整体的吞吐量应当也能随客户端的数目增加而增加。然而,单边 RDMA 分布式锁机制要求所有客户端各自访问服务端内存进行加解锁;受限于 RDMA 网卡的网络包处理速率上限以及分布式锁协议的复杂性,当客户端数目较多时,大量客户端为执行加锁请求或在加锁失败时执行重试,向服务端发送大量 RDMA 原语,服务端网卡扇入(fan-in)问题严重,性能下降明显。因此,现有单边 RDMA 分布式锁协议的扩展性普遍较差,随着客户端增多,即使其总数目并不大,系统整体性能也将停止增长,甚至反而下降。2)公平性指的是各个客户端所得到的服务水平应当基本相同。例如,若所有客户端均连续执行相同的加解锁操作,则每个客户端的吞吐量也应基本相同。公平性问题并非分布式锁协议独有,单机锁场景也有类似问题,然而由于分布式系统中客户端数目众多,且不同客户端与服务端通信的性能可能不同,在单边 RDMA 分布式锁协议中保证公平性面临比单机锁场景更严峻的挑战。

本文提出一种可扩展、公平且高效的单边 RDMA 锁协议 FeLock。通过以节点为单位管理锁,并在多个节点间直接移交锁的所有权(即,一个已加锁的客户端可以令另一个客户端成功加锁,同时自身变为解锁状态),FeLock 能够有效降低服务端网卡的高扇入问题,在提升性能的同时,保证公平性和性能的可扩展性。FeLock 提出了 3 个机制:节点粒度锁管理机制、轮转(round-robin)移交机制,以及节点信用(credit)机制。具体地:1)在节点粒度锁管理机制中,即便 1 个节点上有多个线程试图加锁,它们也仅对应 1 次分布式的加锁操作。FeLock 中的每个锁条目均为 1 个位图,位图中的每一位对应集群中的 1 个节点,可维护节点粒度的加解锁信息。某一节点成功加锁后,该节点上的各线程再依次加锁。该方法可减少客户端执行的 RDMA 通信操作次数,保证高性能。2)在轮转移交机制中,FeLock 将系统中的所有节点在逻辑上排列成 1 个环,节点按照其在环中的顺序依次移交锁的所有权,保证公平性。客户端使用 RDMA 单边原子原语读写位图的值,并在解锁时从位图中识别出环上下一个需要加锁的客户端以减少锁移交的开销。3)在节点信用机制中,FeLock 在每个节点上为每把锁维护 1 个信用值;节点获得锁

的所有权时获得一定信用值，其上的客户端线程每成功加 1 次锁则信用值减 1，信用值归 0 时锁的所有权将被移交给下一节点。该方法可避免单个节点上的大量客户端持续加锁导致其他节点上客户端饥饿的情形出现，保证公平性。

本文的主要贡献包括 3 个方面：

1）分析了单边 RDMA 锁协议相较于传统单机锁协议面临的挑战。

2）提出了一种可扩展、公平且高效的单边 RDMA 锁协议 FeLock，并引入了节点粒度锁管理、轮转移交以及节点信用这 3 个机制，在提升加解锁操作性能的同时，实现性能随客户端数目可扩展，同时保证不同客户端间的公平性。

3）通过实验和对结果的分析，揭示出 FeLock 相比于现有单边 RDMA 锁协议具有更高的性能。其中，在 3~120 个客户端的环境下，FeLock 的吞吐量是 DSLR 的 1.01~7.51 倍，公平性提升至多 2.24 倍。

1 背景介绍与研究动机

本节主要介绍 RDMA 技术的基本特性，并分析单边 RDMA 锁的设计范式及其面临的主要挑战。

1.1 RDMA 技术

RDMA 的全称为远程直接内存访问，是一种新兴的数据中心网络技术。RDMA 本身仅仅是一组技术规范[6]，无限带宽（InfiniBand）、基于融合以太网的 RDMA（RDMA over converged Ethernet, RoCE）、互联网广域 RDMA 协议（Internet wide area RDMA protocol, iWARP）是目前主要的 3 种 RDMA 实现方式。这些实现均具有纯用户态数据路径、极低延迟（1 次网络往返仅需 2~3 μs）和极高吞吐量（每秒可处理数亿个网络包）等特点，相较于基于传统以太网的内核 TCP/IP 协议栈具备显著的性能优势。

RDMA 不仅支持传统的数据发送、接收操作，还支持某一节点直接访问另一节点上的内存区域，且后者的 CPU 无需感知。执行发送、接收这类操作需要数据收发方 CPU 共同参与，因此称作双边操作；直接访问另一节点内存的操作则只需要操作发起方的 CPU 参与，因此称作单边操作。RDMA 的全部双边和单边操作统称为它的网络通信原语（verb）。

双边 RDMA 仅包含发送（send）、接收（recv）这 2 类原语，其中 send 用于发送数据，recv 用于接收 send 发送的数据。单边 RDMA 包含读、写和原子操作这 3 类原语，其中原子操作进一步细分为比较并交换（compare and swap, CAS）和获取并自增（fetch and add, FAA）这 2 种原语。CAS 原语接受 2 个参数 *compare* 与 *swap*，它原子地比较目标内存区域中的 64 b 整数值与 *compare* 的值，若两者相等则将目标内存区域中的值替换为 *swap* 的值，否则不做修改并返回这个值。FAA 原语接受一个参数 *add*，它原子地将目标内存区域中的 64 b 整数值与 *add* 相加，结果写入目标内存区域中，并返回其修改前的值。

特别地，RDMA 还支持扩展的单边原子原语。扩展的 CAS 原语（以下记作 ExtCAS）相较于原本的 CAS 原语额外接受 2 个位掩码参数，它执行比较和交换操作时仅考虑各自位掩码中值为 1 的那些位。扩展的 FAA 原语（以下记作 ExtFAA）额外接受 1 个位掩码参数，它执行加法操作时在位掩码中置 1 的那些位不进位，这相当于位掩码把目标内存区域切分为若干区段，加法在每个区段中分别进行，不同区段间的加法不互相影响。相较于上述标准的单边原子原语，扩展的单边原子原语在语义方面提供了更大的灵活性，可用于实现复杂的分布式协议。

1.2 单边 RDMA 锁及其挑战

基于 RDMA 的分布式锁一般采用如下设计范式：服务端分配内存用于存储锁条目（lock entry），并将这段内存空间暴露给客户端，使其能通过 RDMA 直接访问。客户端使用单边 RDMA 原语读写锁条目的内容，执行特定的锁协议，最终使得彼此冲突的客户端之间互斥。由于客户端读写锁条目时使用的是单边 RDMA，因此这一范式通常称为单边 RDMA 锁。

上述设计范式与单机场景下的自旋锁在一定程度上类似。单机场景下，锁条目被初始化后，多个客户端线程或进程使用 CPU 提供的访存指令读写锁条目的内容以实现加解锁。单边 RDMA 锁场景与单机锁场景的区别仅在于 2 点：1）单机锁场景下，执行锁协议的 CPU 与存储锁条目的内存位于同一台服务器内，而单边 RDMA 锁场景下这二者分别位于客户端和服务端侧，中间由 RDMA 网络连接；2）单机锁场景下客户端使用 CPU 指令读写锁条目，而单边 RDMA 锁场景下客户端使用 RDMA 原语读写锁条目。

然而，相较于单机锁场景，单边 RDMA 锁工作在分布式系统中存在 3 方面的独特挑战：

1）扩展性。相较于单机存储系统，分布式存储系统的一个重要优势就是可以通过网络进行水平扩展（即增加系统中的节点数目和客户端数目），从而实现性能提升。然而，现有的基于单边 RDMA 的分布式锁协议在客户端增多到一定程度时，即使客户

端总数并不大,性能也往往停止增长甚至开始下跌。这是因为客户端数目的增加导致服务端网卡收到RDMA请求的数目也随之增加,且客户端发出的请求还可能互相冲突,需要串行执行,最终导致服务端网卡收到的网络包需长时间排队等待处理,整体性能下降。

2)公平性。假设多个客户端需要反复对同一把锁进行加解锁操作。如果有可能存在某个客户端,其加解锁吞吐量明显高于或低于其他客户端,则称锁协议缺少公平性;若能避免这种情况出现,则称锁协议具有公平性。保证公平性的挑战并非单边RDMA锁独有,单机锁场景中也存在类似问题,但单边RDMA锁运作在分布式场景下,其客户端数目比单机场景多,且不同客户端与服务端的通信延迟也可能存在较大差异,因此其面临的挑战更加严峻。

3)高效率。为了能有效解决以上2个挑战,单边RDMA锁协议的设计通常较为复杂。但是,锁协议的复杂性与其运行效率之间存在矛盾,即执行更复杂的锁协议需付出更大的运行时开销(例如需要多次读写锁条目等),反而导致效率降低。

2 FeLock架构设计

本文提出一种可扩展、公平且高效的基于单边RDMA的分布式锁协议FeLock。图1描述FeLock的总体架构。FeLock向客户端线程提供加锁、解锁2个接口。二者均接受一个64 b非负整数*id*作为参数,*id*表示需要获取或释放的锁的编号。

FeLock将分布式存储系统中的所有节点在逻辑上分为服务端与客户端2部分。服务端是系统中的一个节点,其上开辟一段内存空间,用于存储若干个锁条目,这些锁条目均是对齐的64 b字段,且在内存中按一定间隔排列(如果锁条目连续排列,则间隔为0),并且用非负整数依次编号为0, 1, …。服务端一方面将这段内存空间注册到其上的RDMA网卡上,另一方面通知客户端其地址和长度,使客户端能通过RDMA单边原语直接访问这段内存空间。客户端是系统中的任意节点,应用程序运行在这些节点的CPU上,应用程序的线程如果执行加锁和解锁操作,则称作客户端线程。为了区分不同的节点,所有节点也以非负整数编号为0, 1, …, $n-1$(其中n是系统中的节点数目),它们在逻辑上构成有向的环状结构:1号节点排在0号节点后、2号节点排在1号节点后、…、$n-1$号节点排在$n-2$号节点后、0号节点排在$n-1$号节点后。FeLock提供加解锁程序库,所有客户端线程均可通过调用该程序库提供的加锁、解锁这2个接口函数完成加解锁操作。

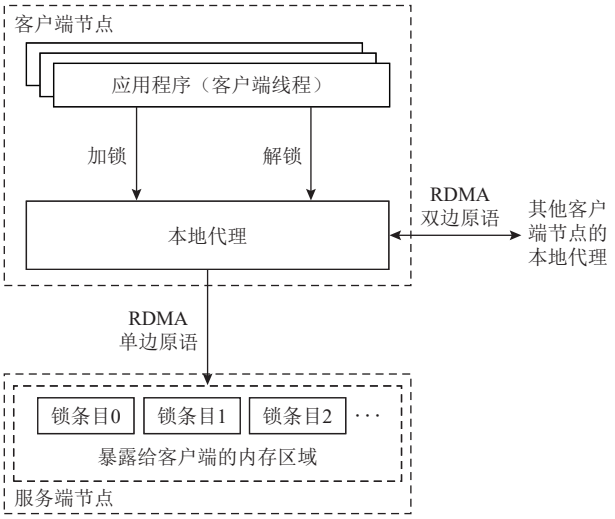


Fig. 1 Overall architecture of FeLock

图1 FeLock总体架构

FeLock在每个节点上分别运行1个本地代理模块,节点上的客户端线程执行加解锁操作时,用户库接口函数进一步调用本地代理模块;本地代理模块负责真正通过RDMA单边原语读写服务端内存中的锁条目,以及通过RDMA双边原语与其他节点上的本地代理模块通信以移交锁的所有权。由于锁条目在服务端内存中等间隔排列,因此本地代理欲访问某一锁条目时,可根据服务端开辟出的内存空间的地址和锁的编号直接计算出对应锁条目的地址,然后即可通过RDMA单边原语读写其内容,无需服务端CPU介入。

为解决研究动机中提到的3个挑战,FeLock引入了节点粒度锁管理、轮转移交、节点信用这3个机制。轮转移交机制涉及锁的所有权在节点间的移动,因此由本地代理负责实现;节点粒度锁管理机制和节点信用机制涉及单个节点内的多个客户端线程,因此由本地代理和程序库中的接口函数共同实现。

3 关键技术

本节介绍FeLock的3个关键技术,包括节点粒度锁管理机制、轮转移交机制以及节点信用机制。

3.1 节点粒度锁管理机制

节点粒度锁管理机制的目标是充分利用分布式存储系统中客户端分布于多个节点上、每个节点可能运行多个客户端线程的架构特点,缩减锁协议在

关键路径上所需的 RDMA 网络往返次数。

具体地，FeLock 基于分层的思想管理锁的所有权，节点为第 1 层，节点中应用程序的客户端线程为第 2 层。对于每把锁而言，服务端上存储的锁条目称作全局锁。每个节点中最多只能有 1 个客户端获取全局锁；一旦 1 个客户端线程成功获取全局锁，则其所在节点内的所有客户端线程均视为成功获取全局锁。在已获取全局锁的情况下，在每个节点内部，各个客户端线程进一步基于 MCS 协议[7] 进行协调。

假设客户端线程 *A* 需要对一把锁执行加锁操作，*A* 首先根据锁的编号，从哈希表中查找当前试图加这把锁的其他客户端构成的链表的尾指针。每个节点上维护一个哈希表，其索引为锁的编号，其值为 1 个指针，指向该节点上试图获取对应编号的锁的客户端构成的链表的尾部（即该指针是该链表的尾指针）。如果这一步未能找到，或者找到的值为空指针，则意味 *A* 所在的节点上当前不存在其他客户端试图加同一把锁。这时，*A* 向哈希表中插入 1 条记录，索引为锁的编号，值为指向 *A* 的指针，由此，当前节点内的加锁操作成功。然后，*A* 调用当前节点上的本地代理提供的加锁接口函数，代表当前节点对全局锁进行加锁，成功后，*A* 的整个加锁操作结束。

如果哈希表中找到了值，且其非空，则意味着除了 *A* 之外，当前节点上还有其他客户端试图获取同一把锁。这时，*A* 根据尾指针信息将自身加入客户端链表的尾部，然后自旋等待链表中排在其前面的客户端全部解锁。等待结束后，整个加锁操作也结束。

成功加锁一段时间后，*A* 将会对这把锁执行解锁操作。解锁时，*A* 检查其所在的客户端链表中是否有排在它后面的其他客户端。如果有，记链表中 *A* 的后继为 *B*；*A* 根据指向 *B* 的指针通过修改 *B* 的本地内存的方式通知其可以加锁，然后 *A* 的整个解锁流程结束。由于 *A*，*B* 位于同一节点上，因此 *A* 在这时无需通过 RDMA 释放全局锁，而是保持着当前节点获取全局锁的状态，直接通知 *B* 获取了当前节点上的锁。相比每个客户端线程独立与服务端通信的做法，这减少了加解锁流程所需的 RDMA 网络往返次数[1,8]。如果 *A* 处于链表的末尾位置，则 *A* 先调用本地代理模块中的解锁函数，使当前节点释放全局锁，然后哈希表中锁的编号所对应的值设为空指针，表示当前节点上已无客户端持有这把锁。

综上所述，如图 2 所示，当某个节点上的客户端 *A* 经由本地代理获取全局锁时，获取的锁实际上对该节点上所有的客户端有效；在 *A* 解锁后，该节点上除

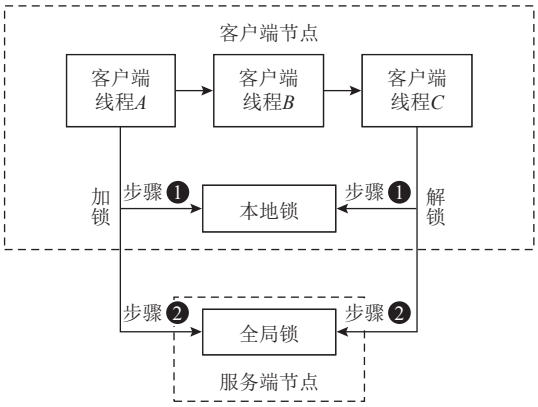


Fig. 2 Demonstration of per-node lock management mechanism

图 2 节点粒度锁管理机制示意

A 以外的客户端可以在不释放全局锁的情况下继续加锁。因此，可以得出 FeLock 中全局锁是以节点粒度管理的结论。

3.2 轮转移交机制

轮转移交机制的目标是将所有客户端在逻辑上排列成一个环，并在有多个试图获取同一把锁的节点时，使其按环所规定的顺序依次移交全局锁的所有权，从而以这些节点间的环状通信模式取代客户端与服务端间的“*n* 对 1”通信模式，降低服务端网卡的扇入压力，提升系统的扩展性。

具体地，每个全局锁均为一个 64 b 的位图，其中的每一位代表系统中的 1 个节点。例如，假设 0 号节点上有客户端线程试图获取 1 把锁，则其锁条目中的最低位将被设为“1”，反之最低位则将被设为“0”。

轮转移交机制由每个节点上的本地代理模块负责实现。首先，当节点上的本地代理模块被客户端线程所调用，需要对 1 把全局锁进行加锁时，本地代理首先根据本文第 2 节所述的方式计算出全局锁的地址，然后向服务端发送一个 RDMA 单边 ExtCAS 原语，将全局锁中当前节点所对应的那一位置为“1”。通过指定合适的位掩码，上述 ExtCAS 原语可以只比较和修改当前节点所对应的那一位，而不关心全局锁中其他位的值；另一方面，由于当前节点在加锁前必然不持有全局锁，因此全局锁中当前节点对应的位的值必为 0；只要将参数 *compare* 设为 0，则上述 ExtCAS 原语必定直接执行成功，而不会出现比较时发现不相等而无法修改全局锁的值的情况。

当 ExtCAS 原语返回全局锁的原始值时，本地代理检查这一原始值是否为 0。若是，则意味着没有其他节点持有全局锁的所有权。因此，加全局锁的操作在这时已经成功，本地代理通知调用方客户端线

程加锁成功;否则,本地代理在1个私有的哈希表中插入1条记录(该哈希表与3.1节中所述的用于管理节点内加解锁的哈希表不同),其索引为锁的编号,其值为调用方客户端线程的地址。然后,本地代理通知调用方客户端线程自旋等待直到其被通知可以加锁为止。

当本地代理被客户端线程调用需要对1把全局锁进行解锁时,本地代理首先计算出全局锁的地址,然后向服务端发送一个RDMA单边ExtCAS原语,将其中当前节点所对应的那一位置为"0"。与加锁时类似,这一ExtCAS原语也必定直接执行成功。然后,本地代理检查返回值,确认除了当前节点所对应的那一位的其他位是否均为0。若是,则解锁操作完成;否则,本地代理从返回值中当前节点所对应的那一位开始,依次向更高位方向查找(遇最高位后返回最低位继续),直到找到第1个值为"1"的位为止。随后,本地代理使用RDMA双边send原语,向这一位所对应的节点发送1条消息,消息内容为锁的编号。这条消息的含义即为向目标节点移交全局锁的所有权。

本地代理同时运行1个后台线程,通过RDMA的recv原语接收其他节点发来的消息。当其接收到的内容为*id*的消息时,意味着其所在的节点收到了其他节点移交来的编号为*id*的全局锁的所有权。这时,本地代理查询其私有哈希表,找到调用了本地代理执行加锁的客户端线程的地址,该客户端即为当前节点上第1个尝试加编号为*id*的锁且尚未成功的客户端。根据3.1节可知,该客户端这时已经成功获取本地锁,因此本地代理可以根据该客户端的地址直接修改其内存,通知其可以加锁。

3.3 节点信用机制

节点信用机制的目标是限制单个节点上客户端线程连续加锁的次数,避免某一节点出现无法加锁的饥饿情况,保证公平性。

当本地代理模块获取1把全局锁的所有权时,其自动获得m个单位的信用。FeLock默认$m=16$,其原因将在3.4节详细讨论。该节点上每有1个客户端线程获取这把锁,信用值就减去1。当信用值减为0时,即便当前节点仍有客户端线程等待加这把锁,其也需执行释放全局锁的操作。如果这种情况出现,则客户端线程在释放全局锁后需要立即重新尝试加全局锁。为减少这一过程的RDMA网络开销,本地代理模块提供一个特殊的"解锁后重新加锁"接口,用于应对这种情况。"解锁后重新加锁"接口的实现方式如下:假设当前等待加锁的客户端链表的头部为客户端线程C。本地代理首先计算出全局锁的地址,然后向服务端发送1个RDMA单边读原语,读取全局锁的当前值,并检查读到的值中除了当前节点所对应的那一位,是否还有其他位为"1"。若没有,则"解锁后重新加锁"操作立即成功,C成功加锁,节点获得m个单位的信用;否则,本地代理向其私有哈希表中插入1条记录(索引为锁的编号、值为指向C的指针),接着按3.2节所述的方式向下一个在等待的节点移交全局锁所有权,最后通知C自旋等待。

如果信用值尚未减为0时节点上就已经没有等待加锁的客户端,则未用完的信用值清0,不累积至下一次。这是因为如果累积,那么如下场景将产生显著的不公平性:一段时间内,节点1上无客户端线程尝试加锁,节点2上仅有少量客户端线程以较低频率尝试加锁;一段时间后节点2将累积大量信用,而节点1根本不会收到其他节点移交的全局锁所有权,无法累积任何信用。后续若2个节点均开始频繁尝试加锁,则节点2的吞吐量将显著高于节点1。

3.4 讨 论

本节依次讨论FeLock的设计和实现中的3个问题,包括支持的节点数目、客户端或服务端故障的处理方式,以及节点信用机制中参数m的取值。

在节点粒度锁管理机制中,由于每个锁条目的字段大小为64 b,因此系统中也最多包含64个节点,否则无法做到令锁条目中每一位对应系统中的1个节点。为了解决这一问题以及容纳更多的节点,可以采用分层的思想。例如,可将整个存储系统中所有节点划分为不超过64个子集群,然后令服务端的锁条目中的每一位代表1个子集群;同时,每个子集群中也选出1个节点来管理该子集群上的锁。当某一节点上的客户端线程需要加锁时,它先获取当前节点上的锁,然后获取其所在的子集群上的锁,最后获取全局锁。

FeLock工作在分布式环境中,若有必要,它将处理客户端或服务端故障的情况。下面先讨论仅有客户端故障的情况。这时,需要使用租约机制,即设置一个最长租期T以表示每个客户端线程每次从加锁到解锁的最长时间。为避免网络传输时延和操作系统调度等因素的影响,T应当远大于RDMA网络往返时间和操作系统的时间片长度,一般至少为数十毫秒。轮转移交机制中,当某一节点上的本地代理试图加全局锁,但发现全局锁的值非0时,其可计算出全局锁移交至当前节点上所需的最多移交次数k。如果本地代理等待足够长时间后仍未收到这把全局

锁的所有权，即可断定某一其他客户端故障，全局锁当前无人持有，因此它可以直接获取这把全局锁的所有权。具体地，这一等待时长可设置为$(m+1)kT$，原因如下：每个节点获取全局锁所有权时，其上的客户端线程最多连续加锁m次，然后就需要将全局锁移交给下一节点。因此，每个节点从收到全局锁所有权到将其移交给下一节点，至多耗费m个租约时间和1个RDMA网络往返延迟（远小于T），其时间总和不应超过$(m+1)T$。此外，全局锁所有权在抵达当前节点前最多被移交k次。综上，若没有任何节点故障，则当前节点最多经过$(m+1)kT$时长就能获取锁的所有权；换言之，若等待超时，则一定有节点故障。

服务端故障的处理流程与仅有客户端故障时的处理流程类似，区别仅在于全局锁条目无法访问。不妨设节点1, 2, 3, …是在环上依次排列的若干节点。若节点1需要在无法访问服务端的情况下加全局锁，则应直接等待$(m+1)nT$时长，原理同上；等待结束或等待期间收到其他节点移交的锁的所有权均意味着成功加锁。节点1释放全局锁时直接将其移交给环中的节点2，无论节点2上是否有客户端线程正等待加锁；若节点2上没有等待加锁的客户端，则其继续将所有权移交给它的节点3即可。

综合FeLock的设计和以上讨论可知，节点信用参数m的取值既不能过大也不能过小。具体地，其取值应大致匹配或略小于实际负载下单个节点内竞争同一把锁的最大客户端数目。若在此基础上增大m值，由于客户端间竞争程度有上限，增加的部分不仅没有作用，而且还增大了客户端获取全局锁所有权所需的时间，即相当于增大了节点饥饿的最大时长，影响短期内的公平性。反过来，若在此基础上减小m值，则在客户端间竞争最激烈时，部分客户端线程可能需要等待其所在节点多次获取全局锁所有权后才能成功加锁，这会显著影响FeLock的尾延迟；但另一方面，这种情况出现的几率较低，因此m的取值可以略微减小以提升公平性。数据中心中常见的高性能计算服务器中，单个CPU一般拥有不超过48个核心；实际负载中最热的数据的访问频率可达40%~50%[8]，因此在竞争最强烈的情况下，单个节点上可能有19~24个CPU核心竞争同一把锁。FeLock所选取的m=16是最接近且略小于这一值的2的幂次。

4 整体流程

为清晰起见，本节给出FeLock加解锁操作的伪代码，如算法1、算法2所示。其中，*LH*表示3.1节所述的管理节点内加解锁的哈希表。

算法1. FeLock加锁操作。

输入：锁的编号*id*；

输出：结构体*lock*（表示已获取的锁）。

① *lock* ← {*next*:NULL, *credit*:0};

② *prev* ← *Exchange*(*LH*[*id*], &*lock*);

③ if *prev* = NULL then

④ 调用算法2;

⑤ else

⑥ *prev.next* = &*lock*;

⑦ 自旋等待，直至*lock.credit* ≠ 0;

⑧ end if

算法2. FeLock本地代理模块加全局锁操作。

输入：锁的编号*id*，算法1中分配的结构体*lock*的地址，节点编号*node*；

输出：无。

① *addr* ← 编号为*id*的锁的地址；

② *mask* ← 1 << *node*;

③ *t* ← *ExtCAS*(*addr*, 0, *mask*, *mask*, *mask*);

④ if *t* = 0 then

⑤ *lock.credit* ← *m*;

⑥ else

⑦ 在私有哈希表中插入记录⟨*id*, *lock*⟩;

⑧ end if

解锁操作流程如算法3和算法4所示，其中*LH*的含义同上。

算法3. FeLock解锁操作。

输入：锁的编号*id*，结构体*lock*；

输出：无。

① if *lock.next* = NULL then

② 以解锁模式调用算法4;

③ *t* ← *LH*[*id*].*CAS*(&*lock*, NULL);

④ 若*t* ≠ &*lock*，则以*t*作为*lock*调用算法2;

⑤ else if *lock.credit* = 1 then

⑥ 以解锁后重新加锁模式调用算法4;

⑦ else

⑧ *lock.next.credit* ← *lock.credit* − 1;

⑨ end if

算法4. FeLock本地代理模块解全局锁操作。

输入：锁的编号*id*，结构体*lock*；

输出：无。

① *addr* ← 编号为*id*的锁的地址；

② *mask* ← 1<<*node*;

③ if 是解锁模式 then
④ $t \leftarrow ExtCAS(addr, mask, 0, mask, mask)$;
⑤ else
⑥ $t \leftarrow Read(addr)$;
⑦ end if
⑧ 若 $t \neq mask$，则找到后继节点并移交锁 id;
⑨ 若 $t = mask$ 且为解锁后重新加锁模式，则执行 $lock.credit \leftarrow m$。

5 实 验

本节实验将在不同的客户端线程数下，从 3 个方面对 FeLock 进行测试，并分析其与现有的基于单边 RDMA 的分布式锁协议的差异：1)对比锁协议的吞吐量；2)对比锁协议的延迟，包括中位延迟和 99 分位尾延迟；3)对比锁协议的公平性。

5.1 实验平台

本实验使用的实验平台如表 1 所示。本实验使用 4 台服务器(即 4 个节点)组成的集群进行测试，其中 1 个节点用作服务端，其余 3 个节点用作客户端，运行客户端线程执行加解锁操作。每个节点上运行至多 40 个客户端线程，因此总客户端线程数至多为 120。每个客户端线程运行于 1 个独立的 CPU 核心上，全体客户端线程均匀分布在 3 个用作客户端的节点上。

Table 1 Experimental Platform Configuration Information
表 1 实验平台配置信息

配置	参数
节点数	4
处理器	Intel® Xeon® Gold 6240M×2
内存	Samsung 32 GB×6
RDMA 网卡	Mellanox ConnectX-5 100 Gbps×1
RDMA 交换机	Mellanox QM8790
操作系统	Ubuntu 18.04，Linux 4.15.0

本实验将 FeLock 与 CAS 和 DSLR 进行性能对比，后二者均是基于单边 RDMA 的锁协议。CAS 是一种经典的锁协议，其锁条目的可能取值仅有“0”和“1”这 2 种，其中“0”代表解锁状态，“1”代表加锁状态。客户端在加锁时，使用 RDMA 单边 CAS 原语将锁条目的值由“0”修改为“1”，若失败则重试此操作直至成功为止。解锁时，客户端用 RDMA 单边写原语将锁条目的值改回“0”。DSLR 是一种基于面包房算法[9]的锁协议，它使用 RDMA 单边扩展 FAA 原语和读原语访问锁条目，其核心思想是将等待加锁的客户端排成一个虚拟队列，相比 CAS 协议而言更能够保证公平性。

实验中共使用 1 000 000 个锁条目。每个客户端线程对锁条目的访问均服从偏斜参数为 0.99 的齐普夫(Zipfian)分布。为了避免无关因素影响测出的延迟和吞吐量等结果，客户端线程在成功加锁后将会立即解锁。

5.2 吞吐量测试

图 3 展示了 FeLock 及其对比方法在 3~120 个客户端线程下的吞吐量。从图 3 可知，在客户端数目为 15 个及以下时，FeLock 与 CAS 和 DSLR 的吞吐量几乎完全相同，且 FeLock 的吞吐量近似随客户端线程数目线性增长。这是因为此时所有客户端产生的加解锁流量不大，不同客户端线程间的冲突不严重，且服务端网卡的扇入压力也较小。当客户端线程数目进一步增加时，CAS 和 DSLR 的吞吐量停止增长，并且反而下降，表现出较差的扩展性。其原因与 1.2 节所述的相同：由于客户端数目增大且客户端间存在锁冲突，服务端 RDMA 网卡面临网络包处理速率瓶颈，RDMA 请求排队等待网卡处理，且队列长度随客户端数目的增加而增加，最终造成系统整体的吞吐量下降。具体地，在 120 个客户端线程时，CAS 和 DSLR 的吞吐量分别相比其最大吞吐量降低了 84.0% 和 50.8%。

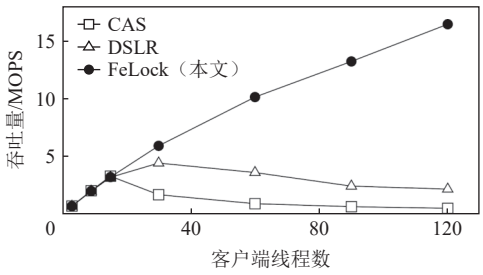


Fig. 3 Throughput with varying client numbers
图 3 不同客户端数目下的吞吐量

FeLock 受益于节点粒度锁管理机制，相较于 CAS 和 DSLR 减少了客户端与服务端间的 RDMA 通信次数，缓解了服务端 RDMA 网卡面临的网络包处理速率瓶颈；其吞吐量能随客户端数目的增加而增加，表现出高性能和良好的性能扩展性。在 3~120 个客户端线程下，FeLock 的吞吐量是 CAS 的 1.00~31.10 倍，是 DSLR 的 1.01~7.51 倍。

5.3 延迟测试

图 4 展示了 FeLock 及其对比方法在 3~120 个客户端线程下加锁操作的中位延迟。图 5 展示了相同

场景下加锁操作的 99 分位延迟(该图为清晰起见，纵轴采用了对数标度)。

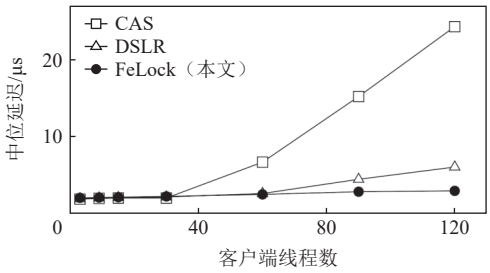


Fig. 4 Median lock acquisition latency with varying client numbers

图 4 不同客户端数目下的加锁中位延迟

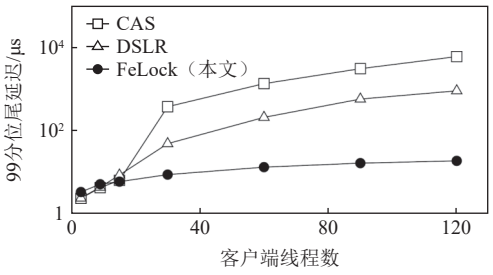


Fig. 5 99th-percentile lock acquisition latency with varying client numbers

图 5 不同客户端数目下的加锁 99 分位延迟

中位延迟反映了冲突不太严重的加锁请求的性能，根据 CAS, DSLR, FeLock 的设计可知，这一延迟的主导因素是 RDMA 单边网络原语的延迟。从图 4 可知，在不同的客户端线程数下，FeLock 加锁操作的中位延迟基本稳定，最大不超过 3.01 μs，这表明服务端网卡的扇入问题不严重，网络包排队等待处理的时间较短。相较之下，CAS 和 DSLR 的服务端网卡扇入问题则更加严重，导致其加锁操作中位延迟在客户端线程数增加时显著增加，其中，CAS 的中位延迟在 120 个客户端线程时比在 15 个客户端线程时增加了 11.62 倍，DSLR 增加了 2.84 倍。

99 分位延迟反映了冲突较严重的加锁请求的性能。从图 5 可知，在 30~120 个客户端线程下，FeLock 加锁操作的 99 分位延迟比 DSLR 低了 1~2 个数量级，比 CAS 则低了 2 个数量级以上。这一性能优势主要来源于 FeLock 的轮转移交机制，其使得每个节点均能在不与服务端通信的情况下快速获得锁的所有权，显著缩短了客户端线程的等待时间。

5.4 公平性测试

图 6 展示了 FeLock 及其对比方法在 3~120 个客户端线程下，所有客户端线程吞吐量的变异系数(coefficient of variation, CV)。一组数据的变异系数指的是其标准差与平均值的比值，刻画了数据的离散程度。由于所有客户端线程的实现均相同，因此吞吐量的变异系数越低，则锁协议的公平性越高，反之亦然(该图为清晰起见，纵轴也采用了对数标度)。

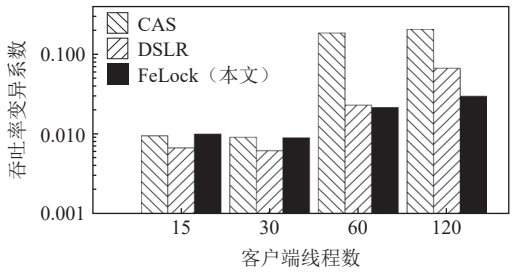


Fig. 6 CV of throughput with varying client numbers

图 6 不同客户端数目下吞吐量的变异系数

在客户端线程数不超过 30 时，被测试的 3 个锁协议的吞吐量变异系数均低于 0.01，各个客户端线程的吞吐量均和平均值接近，锁协议的公平性较好。这一场景下，FeLock 吞吐量的变异系数相较于变异系数最低的 DSLR 高出 0.003 左右，即，FeLock 的客户端吞吐量相对于平均值的偏移量比 DSLR 多 0.3% 左右。造成这一微小的差异的主要原因是在 DSLR 中，所有客户端排成一个虚拟队列依次加锁，严格遵循先到先服务(first-come-first-served, FCFS)原则。然而在 FeLock 中，由于存在节点信用机制，当某个客户端线程在解锁后需要立即重新加锁时，若其所在的节点上这把锁的信用值尚未耗尽，则该客户端线程可能在全局锁所有权不离开当前节点的情况下再次成功加锁；其他节点上这时可能已经有其他客户端线程在等待获取这把锁，由此可知 FeLock 不严格遵循 FCFS 原则，能够导致不同节点间的客户端间产生不公平性。但是，这种情况出现的概率很小，因此其产生的影响也较为有限。

当客户端线程数超过 60 后，CAS 的公平性急剧降低，其吞吐量变异系数超过 0.20，不同客户端的最高吞吐量和最低吞吐量之间相差了 2.07 倍。DSLR 由于采用了虚拟队列机制，公平性比 CAS 好，但其吞吐量变异系数也达到了 0.07，不同客户端的最高吞吐量和最低吞吐量之间相差了 1.27 倍。这一场景下，由于单个节点上客户端线程数较多，FeLock 的客户端线程难以在全局锁所有权不离开其所在节点的情况下重复成功加同一把锁。因此，前述问题不再出现，FeLock 始终能保证 FCFS 原则，能够获得较好的公平性。

FeLock 的公平性在客户端线程数较大时显著优于 CAS 和 DSLR，这一优势主要来源于节点信用机制，

其能够避免单个节点长时间占用某一把锁的所有权，保证不同节点上客户端之间的公平性。FeLock 的吞吐量变异系数在 120 个客户端线程下仅为 0.03，不同客户端的最高吞吐量和最低吞吐量之间相差仅 1.11 倍。

5.5 客户端节点数目对性能的影响

由 5.2~5.4 节的实验结果可以看出，客户端配置对 FeLock 的性能表现有显著的影响。具体地，客户端配置共包含 3 个参数：客户端线程的总数N_{cli}、运行客户端的节点数目n、每个节点上运行的客户端线程(平均)数目$N_{\text{cli_per_node}}$。这 3 项参数之间存在如下关系：

$$N_{\text{cli}} = n \times N_{\text{cli_per_node}} 。$$

因此实际上仅需分析N_{cli}和n这 2 项参数的影响即可。前述实验已经全面地分析了参数N_{cli}的影响，本节则讨论节点数目 n 的影响。

固定客户端线程总数N_{cli}，增加节点数目n时，若n较小，则系统整体性能变化不明显。这是因为，一方面，若节点数目增加，则相同数目的客户端线程依次加锁，期间所需的节点间锁移交次数期望更大。由于节点间移交全局锁所有权所需的时间比节点内移交锁所需的时间更长，因此 FeLock 软件侧的性能将会下降，表现为延迟增加和吞吐量下降。另一方面，增加客户端节点也将客户端的网络流量平均分配到了更多的网卡上，单块网卡压力减小，硬件侧的性能提升。二者效果相抵消，最终导致整体性能变化不明显。本节对于$N_{\text{cli}} \leqslant 40$的数据点重复了前述实验(如 5.1 节所述，单个节点上至多运行 40 个客户端)，结果无明显变化(此处不重复列出)，验证了上述结论。然而，若n进一步增大，节点内的锁移交逐渐不再发挥作用，FeLock 的性能将明显下降。

固定每个节点上运行的客户端线程(平均)数目$N_{\text{cli_per_node}}$，增加节点数目n时，若n较小，则系统的吞吐量和延迟都将增加，这是因为客户端线程的总数变大。若n进一步增大，服务端网卡达到吞吐量上限，则只有延迟增加，吞吐量不变或减少。以上变化趋势符合一般的分布式系统中客户端数目增加时的性能变化规律。

6 相关工作

本节从分布式锁和所有权移交这 2 方面介绍相关研究工作。

1)分布式锁管理。根据依赖的硬件能力不同，目前的分布式锁管理机制可分为不依赖任何特殊硬件的机制、依赖 RDMA 的机制，以及依赖可编程网络硬件的机制 3 类。

在不依赖任何特殊硬件的机制这一类中，2006 年提出的 Chubby[10] 设计了一种面向松耦合分布式系统的锁服务，基于 Paxos[11] 算法可靠地维护分布式锁。2014 年提出的 RedLock[12] 基于 Redis 存储系统，通过在多个 Redis 实例上同时获取锁来提升锁服务的可用性和安全性。相较于这类工作，FeLock 的主要不同点在于其使用了 RDMA 这一特殊的网络技术及相应的特殊硬件，利用硬件能力实现了远高于这类工作的性能；但这也使 FeLock 无法兼容一般的互联网环境，只能工作于数据中心内部的高性能网络中。

在依赖 RDMA 的机制这一类中，2007 年提出的 N-CoSED[13] 混合使用 RDMA 单边 CAS 原语和 FAA 原语，采用与单机读写锁类似的实现，通过写锁和读者计数器共同实现了读写锁协议。2015 年提出的 DrTM[14] 仅使用 RDMA 单边 CAS 原语，使用租约的思想维护读者，实现了读写锁协议。2018 年提出的 DSLR[1] 使用 RDMA 单边 FAA 原语实现了面包房算法，该算法是一个无饥饿的分布式锁协议；本文第 5 节也测试了这一协议的性能。2024 年提出的 SMART[15] 使用 RDMA 单边 CAS 原语，在加锁失败时执行指数回退以降低冲突，同时为回退时间设置动态可调的上限以缩短回退时间来提升性能。相较于这类工作，FeLock 的主要不同点在于其利用了扩展的 RDMA 原子原语，能够在锁条目中编码等待锁的节点信息；而前述工作仅使用了普通的 RDMA 单边原语，锁条目中包含的信息有限，使其无法对锁的所有权进行复杂的控制，性能难以达到和 FeLock 同一水平。

在依赖可编程网络硬件的机制这一类中，2020 年提出的 NetLock[16] 使用可编程交换机维护锁条目，缩短了加解锁操作关键路径，同时充分利用了交换机的网络包处理能力提升系统整体吞吐。2024 年提出的 FissLock[17] 同样使用了可编程交换机，将锁当前的状态(加锁/解锁)与等待加锁的客户端信息分开维护，前者在可编程交换机中维护，后者在客户端节点上维护，节省了交换机上的内存空间占用。相较于这类工作，FeLock 的主要不同点在于其仅使用商用 RDMA 网卡，无需使用可编程网络硬件。可编程网络硬件虽然能带来更高的性能，但其购买、部署和维护成本也远高于硬件逻辑固化、无需手动编程的商用 RDMA 硬件，因此 FeLock 的通用性比上述机制更高。

2)所有权移交。2016 年提出的 D-MCS[18] 面向使用 MPI 的分布式集群，通过 MPI 接口在多个客户端

之间通信，实现了分布式的 MCS 锁协议。相较于 D-MCS，FeLock 仅在节点内使用 MCS 协议，在节点间则使用基于轮转的方式移交全局锁所有权，这使它获得了 D-MCS 不具备的容忍客户端或服务端故障的能力。2022 年提出的 Sherman[19] 采用了与 FeLock 类似的双层级锁管理机制，允许单个节点内的客户端在获取全局锁的情况下彼此移交锁所有权，使用完毕后再释放全局锁。相较于 Sherman，FeLock 一方面引入了节点信用机制，能够防止单个节点上的客户端线程长时间独占锁所有权，造成公平性下降；另一方面采用轮转移交机制而非 RDMA 单边 CAS 原语抢占的方式管理全局锁所有权，有更好的可扩展性。2025 年提出的 ShiftLock[20] 采用了单层的锁管理机制，利用了 RDMA 的动态连接机制中所有客户端之间都能直接移交锁，不管彼此是否在同一节点上。相较于 ShiftLock，FeLock 不仅实现了双层锁管理机制、锁移交时的局部性更优，且其仅使用了扩展的 RDMA 单边原子原语，不依赖 RDMA 动态连接这一高级特性。由于 RDMA 原子原语通常由网卡执行，对 CPU 无需表现原子性[21-22]，因此网卡实现扩展原子原语的开销较低；而动态连接相比传统连接方式需要多个网络包接收方和发送方共同进行复杂的连接状态控制，实现难度大且开销高。综上，FeLock 运用于多种型号的 RDMA 网卡上的潜力更大，相较于 ShiftLock 有更好的通用性。

除了上述这些直接移交锁本身的所有权的工作外，2020 年提出的 SCL[23] 实现了一个全局调度器，在某一客户端解锁时决策出下一个能加锁的客户端，借助调度器实现了间接的锁所有权移交。2023 年提出的 TCLocks[24] 在客户端间直接移交临界区：若客户端 1 需要执行一段程序，但需要加的锁被客户端 2 占用，则客户端 1 将这段程序发给客户端 2，客户端 2 直接在持锁状态下代为执行，并将结果返回给客户端 1。SCL 和 TCLocks 仅能运行在单节点环境下，SCL 需要系统中存在至少 1 个中心化的调度器，而 TCLocks 需要客户端间能够高效且不受隐私问题干扰地互相发送程序片段及其运行上下文，上述要求在分布式系统中难以达到。相较于这二者，FeLock 仅管理分布式锁的所有权，在系统不满足上述要求时也能实现高效的并发控制。

7 结　论

基于单边 RDMA 构建高效的分布式锁协议面临诸多挑战。本文提出一种公平、可扩展且高性能的分布式锁协议 FeLock，通过引入节点粒度锁管理、轮转移交以及节点信用机制，FeLock 有效地缓解了服务端网卡导致的性能和扩展性瓶颈，且保证了不同客户端得到公平的服务。实验显示，相比于现有基于单边 RDMA 的分布式锁协议，FeLock 在客户端线程数较多时具有更良好的吞吐量和公平性。FeLock 目前未考虑读写锁场景，这是未来值得探索的方向。

作者贡献声明：高健负责算法整体设计、实验评估及论文撰写；舒继武提出指导意见并修改论文。

参　考　文　献

[1] Yoon D Y, Chowdhury M, Mozafari B. Distributed lock management with RDMA: Decentralization without starvation [C] // Proc of the 27th Int Conf on Management of Data. New York: ACM, 2018: 1571−1586

[2] NVIDIA. NVIDIA ConnectX-7 InfiniBand in-network computing adapter [EB/OL]. [2025-05-09]. https://www.nvidia.com/content/dam/en-zz/Solutions/networking/infiniband-adapters/infiniband-connectx7-data-sheet.pdf

[3] Kim J, Jang I, Reda W, et al. LineFS: Efficient smart-NIC offload of a distributed file system with pipeline parallelism [C] // Proc of the 28th Symp on Operating Systems Principles. New York: ACM, 2021: 756−771

[4] Dragojević A, Narayanan D, Hodson O, et al. FaRM: Fast remote memory [C] // Proc of the 11th USENIX Symp on Networked Systems Design and Implementation. Berkeley, CA: USENIX Association, 2014: 401−414

[5] Ren Kun, Thomson A, Abadi D J. VLL: A lock manager redesign for main memory database systems[J]. The VLDB Journal, 2015, 24(5): 681−705

[6] Recio R, Metzler B, Culley P, et al. A remote direct memory access protocol specification [EB/OL]. 2007[2025-05-09]. https://www.rfc-editor.org/rfc/rfc5040

[7] Mellor-Crummey J M, Scott M L. Algorithms for scalable synchronization on shared-memory multiprocessors[J]. ACM Transactions on Computer Systems, 1991, 9(1): 21−65

[8] Wang Qing, Lu Youyou, Li Junru, et al. Nap: A black-box approach to NUMA-aware persistent memory indexes [C] // Proc of the 15th USENIX Symp on Operating System Design and Implementation. Berkeley, CA: USENIX Association, 2021: 93−111

[9] Lamport L. A new solution of Dijkstra's concurrent programming problem[J]. Communications of the ACM, 1974, 17(8): 453−455

[10] Burrows M. The Chubby lock service for loosely-coupled distributed systems [C] // Proc of the 7th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2006: 335−350

[11] Lamport L. The part-time parliament[J]. ACM Transactions on Computer Systems, 1998, 16(2): 133−169

[12] Sanfilippo S. A proposal for more reliable locks using Redis [EB/OL]. (2014-05-09)[2025-05-09]. https://antirez.com/news/77

[13] Narravula S, Marnidala A, Vishnu A, et al. High performance distributed lock management services using network-based remote atomic operations [C] // Proc of the 7th IEEE Int Symp on Cluster Computing and the Grid. Los Alamitos, CA: IEEE Computer Society, 2007: 583−590

[14] Wei Xingda, Shi Jiaxin, Chen Yanzhe, et al. Fast in-memory transaction processing using RDMA and HTM [C] // Proc of the 25th Symp on Operating Systems Principles. New York: ACM, 2015: 87−104

[15] Ren Feng, Zhang Mingxing, Chen Kang, et al. Scaling up memory disaggregated applications with SMART [C] // Proc of the 29th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2024, 351−367

[16] Yu Zhuolong, Zhang Yiwen, Braverman V, et al. NetLock: Fast, centralized lock management using programmable switches [C] // Proc of the 34th Annual Conf of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication. New York: ACM, 2020: 126−138

[17] Zhang Hanze, Cheng Ke, Chen Rong, et al. Fast and scalable in-network lock management using lock fission [C] // Proc of the 18th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2024: 251−268

[18] Schmid P, Besta M, Hoefler T. High-performance distributed RMA locks [C] // Proc of the 25th ACM Int Symp on High-Performance Parallel and Distributed Computing. New York: ACM, 2016: 19−30

[19] Wang Qing, Lu Youyou, Shu Jiwu. Sherman: A write-optimized distributed B+Tree index on disaggregated memory [C] // Proc of the 31st Int Conf on Management of Data. New York: ACM, 2022: 1033−1048

[20] Gao Jian, Wang Qing, Shu Jiwu. ShiftLock: Mitigate one-sided RDMA lock contention via handover [C] // Proc of the 23rd USENIX Conf on File and Storage Technologies. Berkeley, CA: USENIX Association, 2025: 355−372

[21] Baran A, Nelson-Silvon J, Tseng L, et al. ALock: Asymmetric lock primitive for RDMA systems [C] // Proc of the 36th ACM Symp on Parallelism in Algorithms and Architectures. New York: ACM, 2024: 15−26

[22] Shu Jiwu. Data Storage Architectures and Technologies[M]. Berlin: Springer, 2024

[23] Patel Y, Yang L, Arulraj L, et al. Avoiding scheduler subversion using scheduler-cooperative locks [C] // Proc of the 15th European Conf on Computer Systems. New York: ACM, 2020: 1−17

[24] Gupta V, Dwivedi K K, Kothari Y, et al. Ship your critical section, not your data: Enabling transparent delegation with TCLocks [C] // Proc of the 17th USENIX Symp on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2023: 1−16



Gao Jian, born in 1998. PhD. His main research interests include distributed storage systems and memory systems.

高　健, 1998 年生。博士。主要研究方向为分布式存储系统、内存系统。



Shu Jiwu, born in 1968. PhD, professor, PhD supervisor. Fellow of CCF. His main research interests include non-volatile memory systems and technologies, network (cloud/bigdata) storage systems, storage security and reliability, and parallel and distributed computing.

舒继武, 1968 年生。博士，教授，博士生导师。CCF 会士。主要研究方向为非易失内存系统与技术、网络(云/大数据)存储系统、存储安全与可靠性、并行与分布式计算。