不妥协：具有一致性、可用性和性能的分布式事务

Aleksandar Dragojevic´, Dushyanth Narayanan, Edmund B. Nightingale,
Matthew Renzelmann, Alex Shamis, Anirudh Badam, Miguel Castro

微软研究院

SOSP'15，2015 年 10 月 4-7 日，加利福尼亚州蒙特雷

ACM 978-1-4503-3834-9/15/10。
http://dx.doi.org/10.1145/2815400.2815425

摘要

具有强一致性和高可用性的事务处理简化了分布式系统的构建和推理。然而，以前的实现效果很差。这迫使系统设计者完全避免事务，削弱一致性保证，或提供需要程序员分割数据的单机事务。在本文中，我们展示了现代数据中心无需妥协。我们展示了一种名为 FaRM 的主内存分布式计算平台，它可以提供具有严格序列化、高性能、耐用性和高可用性的分布式事务。FaRM 在 90 台机器上实现了每秒 1.4 亿次 TATP 交易的峰值吞吐量，数据库容量为 4.9 TB，故障恢复时间小于 50 毫秒。取得这些成果的关键在于从第一原理出发设计了新的事务、复制和恢复协议，以利用具有 RDMA 功能的商品网络和新的廉价方法来提供非易失性 DRAM。

1. 导言

具有高可用性和严格序列化的事务 [35]，通过提供一个简单而强大的抽象概念，简化了有关分布式系统的编程和推理：一台机器永不故障，每次执行一个事务的顺序与实时一致。然而，之前在分布式系统中实现这一抽象概念的尝试导致性能低下。因此，Dynamo [13] 或 Memcached [1] 等系统通过不支持事务或实施弱一致性保证来提高性能。其他系统（如 [3-

6、9、28]），只有当所有数据都位于一台机器内时才提供事务，这就迫使程序员对数据进行分区，并使正确性推理变得复杂。

本文展示了现代数据中心的新软件可以消除妥协的需要。它描述了主内存分布式计算平台 FaRM [16] 中的事务、复制和恢复协议。FaRM 提供分布式 ACID 事务，具有严格的可重复性、高可用性、高吞吐量和低延迟。这些协议的设计初衷是利用数据中心出现的两种硬件趋势：带有 RDMA 的快速商品网络和提供非易失性 DRAM 的廉价方法。实现非易失性的方法是在电源装置上安装电池，并在断电时将 DRAM 的内容写入固态硬盘。这些趋势消除了存储和网络瓶颈，但也暴露了限制其性能优势的 CPU 瓶颈。FaRM 的协议遵循三个原则来解决这些 CPU 瓶颈问题：*减少报文数量*、*使用单边 RDMA 读写而不是介质*，以及*有效利用并行性*。

FaRM 通过将对象分布在数据中心的各台机器上进行扩展，同时允许事务跨越任意数量的机器。FaRM 不使用 Paxos（例如 [11]）复制协调器和数据分区，而是使用垂直 Paxos [25]、主备份复制以及与主备份直接通信的非复制协调器来减少消息数量。FaRM 使用乐观并发控制和四阶段提交协议（锁定、验证、提交备份和提交主）[16]，但我们改进了原始协议，取消了锁定阶段向备份发送的消息。

FaRM 通过使用单边 RDMA 操作进一步降低了 CPU 开销。单边 RDMA 不使用远程 CPU，可避免大部分本地 CPU 开销。FaRM 事务在事务执行和验证期间使用单边 RDMA 读取。因此，它们在远程只读参与者处不使用 CPU。此外，协调器在将记录记录到事务中修改对象的副本的非易失性超前写日志时，也会使用单边 RDMA。对于

例如，协调器使用单边 RDMA 向远程备份写入提交记录。因此，在备份时，传输不使用前台 CPU。随后在后台懒散地截断日志以就地更新对象时，才会使用 CPU。

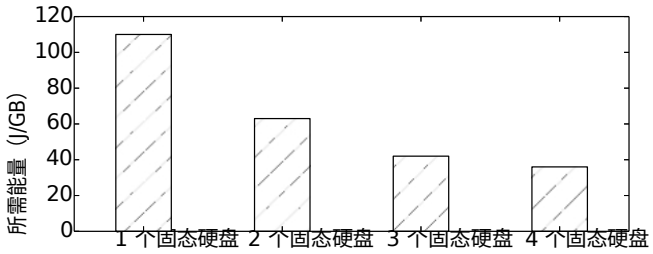
使用单边 RDMA 需要新的故障恢复协议。例如，FaRM 不能依靠服务器在租约[18]到期时拒绝传入的请求，而是要通过一个新的故障恢复协议。
这些请求由网卡提供服务，而网卡则不提供服务。
端口租约。我们通过使用*精确成员*[10]来解决这个问题，以确保机器就当前配置成员达成一致，并只向成员机器发送单边操作。在准备阶段，FaRM 也不能依靠传统机制来确保参与者拥有提交事务所需的再资源，因为事务记录是在不涉及远程 CPU 的情况下写入参与者日志的。相反，FaRM 使用*保留*来确保日志中有空间容纳提交所需的所有记录，并在开始提交前截断事务。

FaRM 的故障恢复协议之所以快速，是因为它有效地利用了并行性。它在整个集群中平均分配每一点状态的恢复，并在每台机器的内核间并行恢复。此外，它还采用了两种优化方法，使事务执行与恢复并行进行。首先，事务会在锁恢复阶段后开始访问受故障影响的数据，该阶段只需几十毫秒即可完成，而无需等待数秒才能完成其余恢复工作。其次，不受故障影响的事务会继续执行而不会阻塞。FaRM 还通过利用快速网络频繁交换心跳来提供快速故障检测，并使用优先级和预分配来避免误报。

我们的实验结果表明，一致性、高可用性和性能是可以兼得的。FaRM 在不到 50 毫秒的时间内就能从单机故障中恢复，其性能优于仅有几台机器的最先进的单机内存事务系统。例如，仅在三台机器上运行时，它的吞吐量就超过了 Hekaton [14, 26]，吞吐量和延迟也超过了 Silo [39, 40]。

2. 硬件趋势

FaRM 的设计灵感来自数据中心机器中大量廉价的



DRAM。典型的数据中心配置是每台双插槽主机拥有 128-512 GB 的 DRAM [29]，而 DRAM 的价格不到 12 美元/GB。[1].这意味着 1 PB 的 DRAM 只需要 2000 台机器，这足以容纳许多有趣应用的数据集。此外，FaRM 还利用两种硬件趋势来消除存储和网络瓶颈：非易失性 DRAM 和使用 RDMA 的快速商品网络。

[1] 16 GB DDR4 DIMM，newegg.com，2015 年 3 月 21 日。

图 1.从 DRAM 复制一 GB 到 SSD 的能耗

2.1 非易失性 DRAM

与使用铅酸电池的传统集中式方法相比，"分布式不间断电源 (UPS) "利用锂离子电池的广泛可用性降低了数据中心 UPS 的成本。例如，Mi-crosoft 的开放云服务器（OCS）规范包括本地能源存储（LES）[30, 36]，它将锂离子电池与机架内每个 24 机箱的电源装置集成在一起。LES 不间断电源的估计成本低于每焦耳 0.005 美元。[2]这种方法比传统 UPS 更可靠：锂离子电池由多个独立电池组成，任何电池故障都只会影响机架的一部分。

分布式不间断电源可有效延长 DRAM 的使用寿命。发生断电时，分布式不间断电源会利用电池能量将内存内容保存到商用固态硬盘中。这不仅避免了同步写入固态硬盘，从而提高了一般情况下的性能，而且只在发生故障时才写入固态硬盘，从而延长了固态硬盘的使用寿命。另一种方法是使用非易失性 DIMM（NVDIMM），其中包含自己的专用闪存、控制器和超级电容器（如 [2]）。遗憾的是，这些设备专业、昂贵且笨重。相比之下，分布式不间断电源使用商品 DIMM 并利用商品 SSD。唯一的额外成本是固态硬盘和 UPS 电池本身的预留容量。

电池配置成本取决于将内存保存到固态硬盘所需的能量。我们在一台标准双插槽机器上测量了未优化的原型。发生故障时，它会关闭硬盘和网卡，将内存数据保存到单个 M.2（PCIe）固态硬盘上，每保存 1 GB 数据消耗 110 焦耳。在保存过程中，大约 90 焦耳用于为机器上的两个 CPU 插座供电。额外的固态硬盘缩短了保存数据的时间，从而降低了能耗（图 1）。优化（如将 CPU 置于低功耗状态）将进一步降低能耗。

在最坏的情况下（单固态硬盘，无优化），以每焦耳 0.005 美元计算，非固态硬盘的能源成本为每焦耳 0.005 美元。

[2] 锂离子不间断电源的成本是传统铅酸不间断电源的 5 倍，传统铅酸不间断电源每 25 兆瓦数据中心的成本为 3100 万美元。一个 25 兆瓦的数据中心可容纳 100,000 台机 器，因此每台机器的锂离子不间断电源成本为 62 美元。一个 24 台机器的机箱有 6 个 PSU，每个 PSU 的 LES 在 5 秒钟内至少可提供 1600 W，在 30 秒钟内至少可提供 1425 W，即每个 PSU 共提供 50 kJ 或每台机器 12.5 kJ，每焦耳的成本为 0.0048 美元。

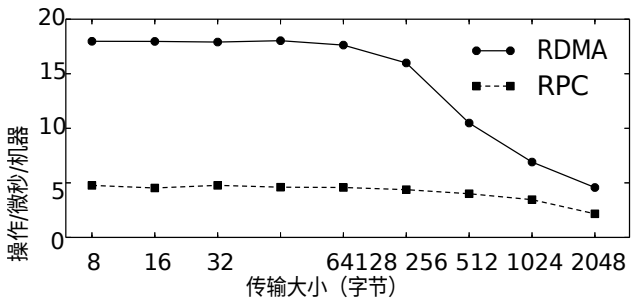


图 2.每台机器的 RDMA 和 RPC 读取性能

波动率为 0.55 美元/GB，预留固态硬盘容量的存储成本为 0.90 美元/GB。[3].综合额外成本不到基本 DRAM 成本的 15%，与成本是 DRAM 3-5 倍的 NVDIMM 相比，这是一个显著的改进。因此，将所有机器内存作为非易失性 RAM (NVRAM) 处理是可行的，也是符合成本效益的。FaRM 将所有数据存储在内存中，并在多个副本中将数据写入 NVRAM 时将其视为持久数据。

2.2 RDMA 网络

FaRM 尽可能使用单边 RDMA 操作，因为它们不使用远程 CPU。我们的这一决定基于我们之前的工作和额外的测量结果。在 [16] 中，我们展示了在 20 台机器的 RoCE [22] 集群上，当所有机器从集群中的其他机器随机读取小对象时，RDMA 读取的性能是可靠 RPC over RDMA 性能的 2 倍。瓶颈在于网卡消息速率，我们的 RPC 实现所需的消息数量是单边读取的两倍。我们在 90 台机器的集群上重复了这一实验，每台机器都有两个 Infiniband FDR（56 Gbps）网卡。与 [16] 相比，每台机器的信息传输率提高了一倍多，并消除了网卡信息传输率的瓶颈。如图 2 所示，RDMA 和 RPC 现在都受 CPU 约束，性能差距扩大到 4 倍。

3. 编程模型和架构

FaRM 为应用程序提供了跨越集群中机器的全局地址空间的抽象。每台机器都运行应用线程，并在地址空间中存储对象。FaRM API [16] 在事务中提供对本地和远程

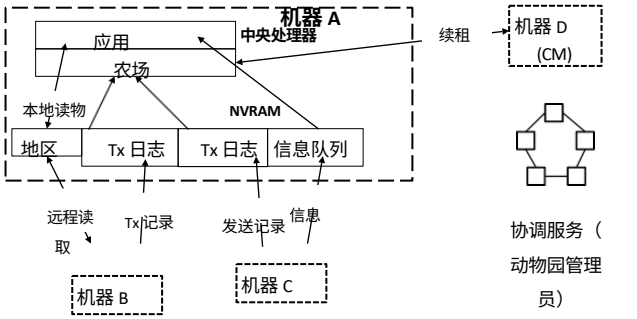


图 3. FaRM 架构

对象的透明访问。应用线程可随时启动事务，并成为事务的协调者。在事务执行期间，线程可以执行任意逻辑以及读取、写入、分配和释放对象。执行结束时，线程调用 FaRM 提交事务。

FaRM 事务使用乐观并发控制。在执行过程中，更新在本地缓冲，并且只在下列情况下进行

[3] Samsung M.2 256 GB MLC，newegg.com，2015 年 3 月 25 日

在成功提交时，其他事务也能看到。提交可能会因为与并发事务冲突或失败而失败。FaRM 为所有成功提交的事务提供严格的可序列化功能 [35]。在事务执行期间，FaRM 保证单个对象的读取是原子性的，它们只读取已提交的数据，对同一对象的连续读取返回相同的数据，对事务写入的对象的读取返回最新写入的值。它并不保证对不同对象的读取都是原子性的，但在这种情况下，它能保证事务不提交，从而确保已提交的事务是严格可序列化的。这样，我们就可以把一致性检查推迟到提交时进行，而不用在每次读取对象时都重新检查一致性。不过，这也增加了编程的复杂性：FaRM 应用程序必须在执行过程中处理这些短暂的不一致性[20]。自动处理这些不一致性是可能的[12]。

FaRM API 还提供无锁读取（经过优化的单对象只读事务）和定位提示（使程序员能够在同一组机器上共同定位相关对象）。如 [16] 所述，应用程序可利用这些功能提高性能。

图 3 显示了一个有四台机器的 FaRM 实例。每台机器都在用户进程中运行 FaRM，每个硬件线程都有一个内核线程。每个内核线程运行一个事件循环，执行应用代码并轮询 RDMA 完成队列。

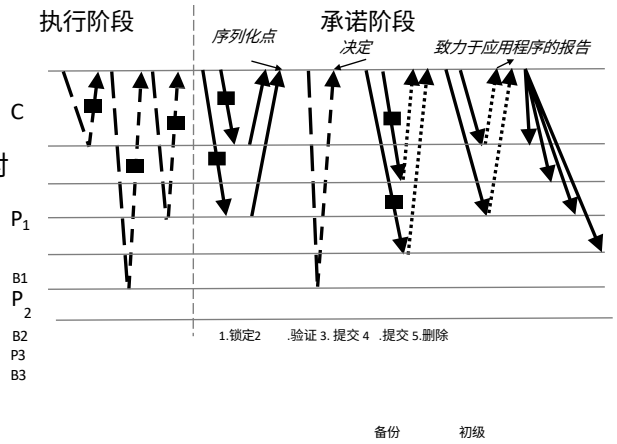
随着时间的推移，随着机器的故障或新机器的添加，FaRM 实例会经历一系列*配置*。配置是一个元组(i, S, F, CM)，其中 i 是唯一的、单调递增的 64 位配置标识符，S *是*配置中的机器集，F 是机器到预计会独立故障的故障域（如不同机架）的映射，$CM \in S$ 是*配置管理器*。FaRM 使用 Zookeeper [21] 协调服务来确保机器就当前配置达成一致，并将其存储起来，如同垂直 Paxos [25]。但它并不像通常那样依赖 Zookeeper 来管理租约、检测故障或协调恢复。CM 通过利用 RDMA 快速恢复的高效实现来完成这些工作。每次更改配置时，CM 都会调用 Zookeeper 更新配置。

FaRM 的全局地址空间由 2 GB 的区域组成，每个区域复制在一个主区域和 f 个备份区域上，其中 f 是所需的容错率。每台机器都存储有 7 这些区域位于非易失性 DRAM 中，其他机器可使用 RDMA 对其进行读取。对象总是从 包含区域的主副本，使用本地内存 如果区域位于本地计算机上，并使用 如果是远程读取，则是单侧 RDMA 读取。每个对象都有一个 64 位版本，用于并发控制和复制。区域标识符与其主用和备用的映射由 CM 维护，并与区域一起复制。 这些映射由其他机器按需获取 并由线程与 RDMA 引用一起缓存，以便向主设备发出单边 RDMA 读取。

机器联系 CM 分配新区域。CM 从单调递增的计数器中分配一个区域标识符，并为该区域选择副本。副本选择会平衡每台机器上存储的区域数量，但必须有足够的容量，每个副本位于不同的故障域中，并且当应用程序指定了 位 置 约束时，区域与目标区域位于同一位置。然后，它会向选定的副本发送包含区域标识符的准备信息。如果所有副本都成功分配了区域，CM 就会向所有副本发送 com- mit 消息。这种两阶段协议可确保映射有效，并在使用前复制到所有区域副本。

与我们之前基于一致散列的方法 [16] 相比，这种集中式方法在满足故障独立性和本地性约束方面具有更大的灵活性。此外，它还能更容易地平衡机器间的负载，并在接近容量的情况下运行。通过 2 GB 的区域，我们预计一台典型机器上最多可有 250 个区域，因此单个 CM 就能处理数千台机器的区域分配。

每台机器还存储有实现先进先出队列的环形缓冲区 [16]。它们可用作事务日志或消息队列。每对发送方-接收方都有自己的日志和消息队列，它们物理上位于接收方。发送方使用单边 RDMA 写入其尾部的方式将记录添加到日志中。这些写入由网卡确认，不涉及接收方的 CPU。接收方定期轮询日志头部以处理记录。当发送方截断日志时，接收方会懒散地更新发送方，



从而允许发送方重复使用环形缓冲区中的空间。

4. 分布式事务和复制

FaRM 整合了事务协议和复制协议，以提高性能。与传统协议相比，它使用的信息更少，并利用单边 RDMA 读取和写入实现 CPU 效率和低延迟。FaRM 在非易失性 DRAM 中对数据和事务日志使用主-备份复制，并使用与主和备份直接通信的非复制事务协调器。它使用乐观并发控制，读取

图 4.有协调器 C 的 FaRM 提交协议，P , P_{12} , P_3 上有主程序，B , B_{12} , B_3 上有备份程序。P_1 和 P_2 为读写程序。P_3 只进行读取。我们使用虚线表示 RDMA 读取，实线表示 RDMA 写入，虚线表示硬件 acks，矩形表示对象数据。

验证，如某些软件事务处理内存系统（如 TL2 [15]）。

图 4 显示了 FaRM 事务的时间轴，表 1 和表 2 列出了事务协议中使用的所有日志记录和消息类型。在执行阶段，事务使用单边 RDMA 读取对象，并在本地缓冲写入。协调器还会记录所有被访问对象的地址和版本。对于与协调器在同一台机器上的主程序和备用程序，对象读取和写入日志使用的是本地内存访问而不是 RDMA。在执行结束时，FaRM 会尝试通过执行以下步骤来完成事务：

*1. 锁定。*协调器会在每台机器上的日志中写入一条 LOCK 记录，该机器是任何已写入对象的主服务器。该记录包含该主节点上所有已写入对象的版本和新值，以及所有已写入对象的区域列表。主控程序在处理这些记录时，会尝试使用比较和交换（compare-and-swap）将对象锁定在指定的版本上，然后发回一条信息，报告是否成功锁定了所有对象。如果对象版本在事务读取后发生变化，或者对象当前已被其他事务锁定，则锁定可能失败。在这种情况下，协调器会中止事务。它会向所有主事务写入中止记录，并向应用程序返回错误信息。

*2. 验证。*协调器执行读取验证时，会从其主对象中读取该事务已读取但未写入的 所有对象的版本。如果有对象发生变化，则验证失败，事务中止。验证默认使用单边 RDMA 读取。对于持有超过 t_r 个对象的主事务，验证通过 RPC 完成。阈值 t_r（目前为 4）反映了 RPC 相对于 RDMA 读取的 CPU 成本。

*3. 提交备份。*每次备份时，协调器都会在非易失性日志中写入 COMMIT- BACKUP 记录，然后等待 NIC 硬件的应答，而不会相互干扰。

日志记录类型	目录
锁定	事务 ID、事务写入对象的所有区域的 ID，以及目的地为主要对象的事务写入的所有对象的地址、版本和值
commit-backup commit-primary abort	内容与锁定记录事务 ID 相同。
删除	未截断交易的低限交易 ID 和要截断的交易 ID

表 1.事务协议中使用的日志记录类型。未被截断的事务标识符的低限和用于截断的事务标识符捎带在每条记录上。

信 息 类 型	目录
LOCK-REPLY	事务 ID，表示锁定是否成功的结果
VALIDATE	从目的地读取的对象的地址和版本（在通过 RDMA 读取进行验证时不发送） 待恢复的配置 ID、区域 ID 和事务 ID（由备份发送给主服务器）
need-recovery fetch-tx-state send-tx-state replicate-tx-state recovery-vote	配置 ID、区域 ID 和状态被请求的事务 ID（由主系统发送给备份） 配置 ID、区域 ID、事务 ID 和获取请求的事务的锁定记录内容 配置 ID、区域 ID、事务 ID 和锁定记录内容（由主系统发送给备份） 配置 ID、区域 ID、事务 ID、事务修改的区域的区域 ID 和投票 配置 ID、事务 ID 和区域 ID
请求--表决	配置 ID 和事务 ID 配置 ID 和事务 ID 配置 ID 和事务 ID
commit-recovery abort-recovery	
截断恢复	

表 2.事务协议中使用的报文类型。除前两种外，其他都只在恢复时使用。

占用备份的 CPU。COMMIT-BACKUP 日志记录的有效负载与 LOCK 记录相同。

*4. 提交主日志。*在所有 COMMIT-BACKUP 写入都被接受后，协调器会在每个主日志中写入 COMMIT- PRIMARY 记录。协调器在收到至少一个针对该记录的硬件回执时，或在本地写入一个记录时，就会向应用程序报告完成。主节点在处理这些记录时，会就地更新对象、增加其版本并解锁，从而暴露事务提交的写入。

*5. 截断。*备份和初选程序将记录保留在日志中，直到被截断。协调器会在收到所有主日志的请求后，懒散地截断主日志和备份日志。为此，协调器会在其他日志记录中搭载被截断事务的标识符。备份会在截断时间将更新应用到其对象副本中。

正确性已提交的读写事务可在获取所有写锁时序列化，已提交的只读事务可在最后一次读取时序列化。这是因为所有读写对象在序列化点的版本与执行过程中的版本相同。锁定可确保写入对象的版本一致，而验证可确保只读对象的版本一致。在没有故障的情况下，这相当于在序列化点以原子方式执行和提交整个事务。FaRM 中的序列化也很*严格*：序列化点总是在开始执行与向应用程序报告完成之间。

为了确保跨故障的可序列化，在写入 COMMIT-PRIMARY 之前，有必要等待所有备份的硬件应答。假设协调器没有收到某个备份 b 对区域 r 的确认，那么主存可能会暴露事务修改，随后与协调器和 r 的其他副本一起失效，而备份 b 从未收到 COMMIT-BACKUP 记录。这将导致 r 的更新丢失。

由于读取集仅存储在协调器中，因此如果协调器失效，且没有网络记录证明验证成功，事务就会中止。因此，协调器在向应用程序报告成功提交之前，有必要等待其中一个主节点的成功提交。这样就能确保重新提交到应用程序的事务中至少有一条提交记录能在失败后*存活*下来。否则，在写入任何 COMMIT-PRIMARY 记录之前，如果协调器和所有备份都失败了，那么这样的事务仍会中止，因为只有 LOCK 记录会存活下来，而不会有验证成功的记录。

在传统的两阶段提交协议中，参与者可以在处理准备消息时为提交事务预留资源，或者在没有足够资源的情况下拒绝准备事务。但是，由于我们的协议避免在提交过程中使用备份的 CPU，因此协调者必须为所有参与者预留日志空间，以保证进度。协调者在开始提交前为所有提交协议记录预留空间，包括主日志和备份日志中的截断记录。日志预留是协调器的本地操作。

因为协调器会在每个参与者处向其拥有的日志写入记录。在写入相应记录时，保留会被释放。如果截断被其他报文捎带，截断记录保留也会被释放。如果日志已满，协调器会使用保留来写入明确的截断记录，以释放日志空间。这种情况很少见，但为了确保及时性，必须这样做。

性能对于我们的目标硬件而言，该协议与传统的分布式提交协议相比具有多项优势。考虑一下带有复制功能的两阶段提交协议，比如 Spanner 的协议[11]。Spanner 使用 Paxos [24] 来复制事务协调器及其参与者（即存储事务读写数据的机器）。每个 Paxos 状态机在传统的两阶段提交协议[19]中扮演单个机器的角色。由于每个状态机操作至少需要 $2f+1$ 个往返消息，因此需要 $4P(2f+1)$ 个消息（其中 P 是事务参与者的数量）。

FaRM 使用主备份复制取代 Paxos 状态机复制。这将数据副本的数量减少到 $f+1$，同时也减少了事务中传输的信息数量。协调器状态不进行复制，协调器直接与主备和备份进行通信，从而进一步减少了延迟和信息数量。FaRM 因复制而产生的开销很小：只需向每台远程机器写入一次 RDMA，就能备份任何已写入的对象。该协议完全不涉及只读参与者的备份。此外，通过 RDMA 进行的读验证可确保只读参与者的主程序不做 CPU 工作，而 COMMIT-PRIMARY 和 COMMIT-BACKUP 记录的单向 RDMA 写入可减少对远程 CPU 的等待，并允许远程 CPU 懒惰地分批工作。

FaRM 提交阶段使用 $P_w(f+3)$ 单边 RDMA 写入（其中 P_w 是事务写入对象的主控机器数量）和 P_r 单边 RDMA 读取（其中 P_r 是从远程主控机器读取但未写入的对象数量）。读验证会在关键路径上增加两个单边 RDMA 延迟，但这是一个很好的权衡：增加的延迟在无负载时只有几个微秒，而 CPU 开销的减少会带来更高的吞吐量和更低的负载延迟。

5. 故障恢复

FaRM 利用恢复功能提供耐用性和高可用性。我们假设机器会因崩溃而失效，但可以在不丢失非易失性 DRAM 内容的情况下恢复。我们依靠有界时钟漂移来保证安全性，依靠最终有界消息延迟来保证有效性。

即使整个集群发生故障或断电，我们也能保证所有已提交事务的持久性：所有已提交状态都能从存储在非易失性 DRAM 中的区域和日志中恢复。即使每个集群中最多只有 f 个副本，我们也能确保持久性。

对象丢失非易失性 DRAM 的内容。FaRM 还能在出现故障和网络分区的情况下保持可用性，前提是存在一个分区，其中包含大部分保持相互连接的机器和 Zookeeper 服务中的大部分副本，并且该分区包含每个对象的至少一个副本。

FaRM 的故障恢复分为五个阶段：故障检测、重新配置、事务状态恢复、批量数据恢复和分配器状态恢复。

5.1 故障检测

FaRM 使用租约[18]来检测故障。每台机器（CM 除外）在 CM 处都有一个租约，而 CM 在其他每台机器处都有一个租约。任何租约到期都会触发故障恢复。租约通过 3 路握手授予。每台机器向 CM 发送租用请求，CM 则回复一条信息，该信息既是对机器的租用授予，也是 CM 的租用请求。然后，机器向 CM 回复租约授予。

FaRM 租期极短，这是高可用性的关键所在。在重负载情况下，FaRM 可以对一个 90 台机器的集群使用 5 毫秒的租期，而不会出现误报。更大的集群可能需要两级层次结构，在最坏的情况下，故障检测时间将增加一倍。

要在负载情况下实现较短的租用时间，需要精心实施。FaRM 为租期使用专用队列对，以避免租期信息在共享队列中被其他信息类型延迟。如果使用可靠的传输方式，则需要在 CM 中为每个网络增加一个队列对。由于网卡队列对缓存中的容量缺失，这将导致性能低下[16]。相反，租约管理器使用 Infiniband 发送和接收动词与无连接不可靠数据报传输，这只需要网卡上一个额外队列对的空间。默认情况下，每隔 1/5 个租期到期日就会尝试续租一次，以考虑潜在的报文丢失。

租约续订也必须在 CPU 上及时安排。FaRM 使用专门的租约管理器线程，该线程以最高的用户空间优先级（Windows 为 31）运行。租赁管理器线程不固定在任何硬件线程上，它使用中断而不是轮询，以避免启动必须在每个硬件线程上定期运行的关键操作系统任务。这会增加几微秒的信息延迟，但这对租约来说不成问题。

此外，我们没有将 FaRM 线程分配给每台机器上的两个硬件线程，而是留给了租赁管理器。我们的测量结果表明，租赁管理器通常在这些硬件线程上运行，不会影响其他 FaRM 线程，但有时会被优先级更高的任务抢占，导致租赁管理器在其他硬件线程上运行。因此，在使用短租期时，将租期管理器固定在硬件线程上很可能会导致误报。

最后，我们在初始化过程中预先分配了租约管理器使用的所有内存，并将其使用的所有代码分页并销钉，以避免因内存管理而造成的延迟。

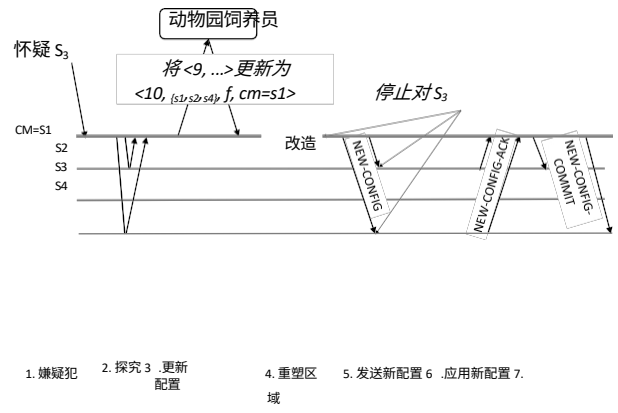


图 5.重新配置

5.2 重新配置

重新配置协议将 FaRM 实例从一种配置移动到另一种配置。使用单边 RDMA 操作对于实现良好性能非常重要，但它对重新配置协议提出了新的要求。例如，实现一致性的常用技术是使用租约[18]：服务器在回复访问对象的请求之前，会检查自己是否持有该对象的租约。如果服务器被逐出配置，系统会保证其存储的对象在租约到期前不会被更改（例如，[7]）。当外部客户端使用消息与系统通信时，FaRM 在为其请求提供服务时使用了这种技术。但由于 FaRM 配置中的机器使用 RDMA 读取方式读取对象，而不进入远程 CPU，因此服务器的 CPU 无法检查自己是否持有租期。目前的 NIC 硬件不支持租约，也不清楚将来是否会支持。

我们通过实施*精确成员制* [10] *来*解决这个问题。故障发生后，新配置中的所有机器必须就其成员资格达成一致，然后才允许对象突变。这样，FaRM 就能在客户端而不是服务器上执行检查。配置中的机器不会向不在配置中的机器发出 RDMA 请求，不再在配置中的机器对 RDMA 读取的回复和对 RDMA 写入的请求也会被忽略。

图 5 显示了一个重新配置时间表示例，包括以下步骤：

*1. 怀疑。*当一台机器的租约在 CM 处到期时，它就怀疑。通过这些*读取探针，只需进行*一次重新配置，就能处理影响多台机器的相关故障，如电源和交换机故障。新的 CM 只有在获得大多数探测的响应后，才会继续进行重新配置。这样可以确保在网络分区时，CM 不会位于较小的分区中。

*3.更新配置。*收到探测回复后，新 CM 会尝试更新配置

存储在 Zookeeper 中的数据为 $\langle c+1, S, \mathsf{F}, \mathrm{CM_{id}} \rangle$，其中 c 是 当前配置标识符，S 是机器集

会怀疑这台机器出现故障，并开始重新配置。此时，它会开始阻止所有外部客户端请求。如果非 CM 机器怀疑 CM 因租约到期而失效，它会首先请求少数 "备份 CM " 之一启动重新配置（CM 的 k *个*后继者使用一致的哈希算法）。如果超时后配置仍未改变，它就会自己尝试重新配置。这种设计避免了在 CM 失败的情况下同时进行大量的重新配置尝试。在所有情况下，启动重新配置的机器都会尝试成为新的 CM，作为重新配置的一部分。

*2. 探测。*新 CM 会向配置中除可疑机器外的所有机器发出 RDMA 读取。读取失败的机器也会被

F 是机器到故障域的映射，CM_{id} 是其自身的标识符。我们使用 Zookeeper znode 序列号来实现原子比较和交换，只有在当前配置仍为 *c* 时才会成功。这确保了即使多台机器同时尝试从标识符为 c 的配置更改配置，也只有一台机器能成功地将系统移动到标识符为 *c+1* 的配置（并成为 CM）。

*4. 重新映射区域。*然后，新的 CM 会重新分配之前映射给故障机器的区域，将副本数量恢复到 $f+1$。在容量和故障独立性的限制下，它会尝试平衡负载并满足应用程序指定的位置提示。对于失败的主服务器，它会通过其他方式将幸存的备份提升为新的主服务器，以减少恢复时间。如果检测到区域丢失了所有副本或没有空间重新复制区域，就会发出错误信号。

*5. 发送新配置。*重新映射区域后，CM 会向配置中的所有机器发送 NEW-CONFIG 消息，其中包含配置标识符、自身标识符、配置中其他机器的标识符以及区域到机器的所有新映射。如果 CM 已更改，NEW-CONFIG 还将重置租用协议：它将作为新 CM 向每台机器发出的租用请求。如果 CM 未变，则在重新配置期间继续交换租约，以便快速检测其他故障。

*6. 应用新配置。*当一台机器收到一个配置标识符大于其自身配置标识符的 NEW-CONFIG 时，它会更新其当前配置标识符和区域映射的缓存副本，并分配空间来容纳分配给它的任何新区域副本。从这时起，它不会向不在配置中的机器发出新请求，并拒绝来自这些机器的读取响应和写入请求。它还会开始阻止来自外部客户端的请求。机器通过 NEW-CONFIG-ACK 消息回复 CM。如果 CM 已更改，它既会向 CM 授予租约，也会请求租约。

*7. 提交新配置。*一旦 CM 收到来自配置中所有机器的 NEW-CONFIG-ACK 消息，它就会等待，以确保先前配置中授予不再是配置成员的机器的租约已过期。然后，CM 会向所有配置成员发送 NEW-CONFIG-COMMIT（新配置确认）信息，该信息同时也作为新配置确认信息。

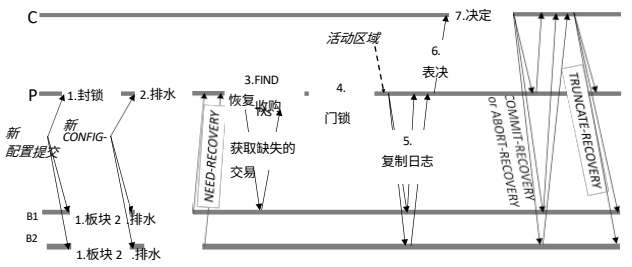


图 6.显示协调器的事务状态恢复

C、主 *P* 和两个备份 B_1 和 B_2

租赁许可。现在，所有成员都能解除阻止先前阻止的外部客户端请求，并启动事务恢复。

5.3 事务状态恢复

FaRM 利用分布在事务修改对象副本中的日志，在配置更改后恢复事务状态。这涉及在被事务修改的对象副本中恢复状态，以及在协调器中恢复状态，以决定事务的结果。图 6 显示了一个事务恢复时间线示例。FaRM 通过将工作分配给集群中的线程和机器来实现快速恢复。所有消息日志的排空（第 2 步）都是并行进行的。第 1 步和第 3-5 步并行针对所有区域执行。第 6-7 步并行针对所有恢复 。

*1. 阻止访问正在恢复的区域。*当一个区域的主区域发生故障时，其中一个备份会在重新配置过程中被提升为 新的主区域。在更新区域的所有事务在新主区域得到反映之前，我们不允许访问该区域。为此，我们会阻止对该区域的本地指针和 RDMA 引用请求，直到第 4 步，即所有更新该区域的恢复事务的所有写锁都被获取 。

*2. 耗尽日志。*单边 RDMA 写入也会影响跨操作恢复。实现跨配置一致性的一般方法是拒绝来自旧配置的消息。FaRM 无法使用这种方法，因为 NIC 会确认写入事务日志的 COMMIT-BACKUP 和 COMMIT-PRIMARY 记录，而不管这些记录是在什么配置下发出的。由于协调程序只等待这些确认，然后才会公开更新并向应用程序报告成功，因此机器在处理来自旧配置的记录时，不可能总是将其拒之门外。我们通过*消耗日志*来解决这个问题，以确保在恢复过程中处理所有相关记录：所有机器在收到 NEW-CONFIG- COMMIT 消息时都会处理日志中的所有记录。处理完成后，它们会在变量 *LastDrained* 中记录配置标识符。

FaRM 事务具有唯一标识符⟨c, *m, t,* l⟩，这些标识符在提交开始时签名，编码了开始提交的配置 *c*、协调器的机器标识符 *m* 和协调器的线程标识符 *t*、

和线程本地唯一标识符 *l*。配置标识符小于或等于 *LastDrained* 的跨操作日志记录将被拒绝。

*3. 查找正在恢复的事务。*恢复事务是指提交阶段跨越配置更改的事务，其中写入对象的某些副本、读取对象的某些主副本或协调器因重新配置而发生了更改。在日志排空过程中，会检查每个日志记录中的事务标识符和更新区域标识符列表，以确定正在恢复的事务集。只有恢复中的事务才会在主事务和备份事务中进行事务恢复，协调器只拒绝恢复中事务的硬件请求。

所有机器必须就给定事务是否为恢复事务达成一致。为此，我们在重新配置阶段为通信添加了一些额外的元数据。作为探测读取的一部分，CM 会读取每台机器上的 *LastDrained* 变量。对于自 *LastDrained* 之后映射发生变化的每个区域 *r*，CM 会在 NEW-CONFIG 消息中向该机器发送两个配置标识符。这两个标识符分别是 *LastPrimaryChange*[*r*] 和 *LastReplicaChange*[*r*]，前者是 *r* 的主配置发生变化时的最后一个配置标识符，后者是 *r* 的任何副本发生变化时的最后一个配置标识符。在配置 *c* - 1 中开始提交的事务会在配置 *c* 中恢复，*除非*：对于包含被事务 *LastReplicaChange*[*r*] < *c* *所*修改对象的所有区域 *r*，对于包含被事务 *LastPrimaryChange*[*r*′] < *c* *所*读取对象的所有区域 *r*′，且协调器未从配置 *c* *中*移除。

恢复事务的 记录可能分布在不同的主事务日志和事务更新的备份中。区域的每个备份都会向主服务器发送一条 "需要恢复"（NEED- RECOVERY）消息，其中包含配置标识符、区域标识符以及更新区域的恢复 ing 事务的标识符。

*4. 锁定恢复。*每个区域的主线程在本地机器日志耗尽和从每个备份收到 NEED- RECOVERY 消息之前，等待建立影响该区域的完整恢复事务集。与此同时，主线程从备份中获取尚未存储在本地的事务日志记录，然后锁定被恢复事务修改的对象。

当一个区域的锁恢复完成时，该区域处于*活动状态*，本地和远程协调器可以获得本地指针和 RDMA 引用，从而可以读取对象并提交对该区域的更新，与 后续恢复步骤并行。

*5. 复制日志记录。*主线程通过向备份发送 REPLICATE-TX- STATE 消息来复制日志记录。

该信息包含区域标识符、当前 con- figuration 标识符以及与 LOCK 记录相同的数据。

*6. 投票。*正在恢复的事务的协调人根据该事务更新的每个区域的投票情况，决定是提交还是中止该事务。这些投票由每个区域的初选者发出。FaRM 使用一致的哈希算法来确定事务的协调者，确保所有初选者都能独立地就恢复事务协调者的身份达成一致。如果协调器所运行的机器仍在配置中，则协调器不会改变，但当协调器失效时，协调其恢复事务的责任会分散到集群中的所有机器上。

主线程会向协调器中的对等线程发送 RECOVERY-VOTE 消息，内容涉及修改区域的每个再覆盖事务。如果有副本看到 COMMIT-PRIMARY 或 COMMIT-RECOVERY，则投票为 *commit*-primary。否则，如果有副本看到 COMMIT-BACKUP，但没有看到 ABORT-RECOVERY，则投票决*定提交*-备份。否则，如果有副本看到 LOCK 记录且没有 ABORT-RECOVERY，它就投 *lock 票*。否则，它会投票*中止*。投票信息包括配置标识符、区域标识符、事务标识符和被事务修改的区域标识符列表。

一些初选者可能不会启动事务投票，因为他们要么从未收到过该事务的日志记录，要么已经截断了该事务的日志记录。协调器会向超时（设定为 250 μs）内尚未投票的初选者发送明确的投票请求。REQUEST-VOTE 消息包括配置标识符、区域标识符和跨行动标识符。有事务日志记录的优先级在等待该事务日志复制完成后，会像之前一样进行投票。

如果事务已被截断，则没有该事务任何日志记录的主线程会投票*截断*；如果事务未被截断，则主线程会投票*未知*。为了确定事务是否已被截断，每个线程都会维护一组事务的标识符，这些事务的记录已从其日志中被截断。通过使用未被截断事务标识符的下限来保持该集合的紧凑性。下限根据每个协调器的下限进行更新，这些下限在协调器消息和重新配置过程中捎带更新。

*7. 决定。*如果协调器收到任何区域的提交-*主要*投票，它就会决定提交事务。否则，协调器等待所有区域投票，如果至少有一个区域投了提交*备份票*，且所有其他被事务修改的区域都投了*锁定票*、*提交备份票*或*截断票*，*协调器才会提交*。否则，它将决定中止。然后，它会向所有参与的副本发送 COMMIT-RECOVERY 或 ABORT-RECOVERY。这两条信息都包括配置标识符和事务标识符。COMMIT-RECOVERY 的处理方法与 COMMIT-PRIMARY 类似，如果是在一个

如果在备份中收到，则处理为 COMMIT-BACKUP。ABORT-RECOVERY 的处理与 ABORT 类似。协调器从所有主节点和备份节点接收到回跳后，会发送 TRUNCATE-RECOVERY 消息。

正确性接下来，我们将介绍事务恢复的不同步骤如何确保严格的序列化能力。关键在于，事务恢复要保留先前提交或中止的事务的结果。当主事务公开事务修改或协调器通知应用程序事务已*提交时*，我们就认为事务已提交。当协调器发送中止消息或通知应用程序事务已中止时，事务即被中止。对于结果尚未确定的事务，恢复可能会提交或中止事务，但会确保从其他故障中恢复时保留结果。未恢复事务的结果（步骤 3）是通过正常情况协议决定的（Sec. 第 4 节）。因此，我们不再进一步讨论。

已提交的恢复事务的日志记录保证在日志耗尽之前或期间（步骤 2）被处理和接受。这是因为主事务只有在处理 COMMIT- PRIMARY 记录后才会暴露修改。如果协调器通知了应用程序，那么它必须已收到所有 COMMIT- BACKUP 记录和至少一个 COMMIT-PRIMARY 记录的硬件应答。

记录（因为它会忽略更改配置后的 acks）。因此，由于新配置包括每个区域的至少一个副本，因此至少一个区域的至少一个副本将处理 COMMIT-PRIMARY 或 COMMIT-BACKUP 记录，而每个其他区域的至少一个副本将处理 COMMIT-PRIMARY、COMMIT-BACKUP 或 LOCK 记录。

第 3 步和第 4 步确保被事务修改的区域的初选者能看到这些记录（除非被截断）。它们会将这些记录复制到备份中（步骤 5），以确保即使出现后续故障，投票也能产生相同的结果。然后，初选者根据它们看到的记录向协调者发送投票（步骤 6）。

决策步骤保证协调器决定提交之前已提交的任何事务。如果有副本截断了事务记录，所有主副本都将投票决定*提交-主*副本、*提交-备份*或*截断*。至少有一个主副本会投 *Truncated 以外的*票，否则事务将无法恢复。如果没有副本截断事务记录，则至少有一个主副本将投票决定*提交主*副本或*提交备份，*其他副本将投票决定*提交主*副本、提交*备份*或*锁定*。同样，如果先前中止过事务，协调器将决定中止，因为在这种情况下，要么没有*提交主*记录或*提交备份*记录，要么所有副本都收到 ABORT-RECOVERY。

阻止对恢复区域的访问（第 1 步）和锁恢复（第 4 步）可确保在恢复传输提交或中止之前，任何其他操作都无法访问它所修改的对象。

性能FaRM 采用了多项优化措施来实现快速故障恢复。识别正在恢复的事务将其恢复工作限制在仅受重新配置影响的事务和区域，当大型集群中的单台机器发生故障时，这可能只是总数的一小部分。我们的结果表明，这可以将需要恢复的事务数量减少一个数量级。恢复工作本身是跨区域、机器和线程并行进行的。在锁恢复后立即提供可用区域可提高前台性能，因为访问这些区域的新事务不会长时间阻塞。具体来说，它们无需等待这些区域的新副本被更新，这就需要在网络上大量移动数据。

5.4 恢复数据

FaRM 必须在一个区域的新备份中恢复（重新复制）数据，以确保它在未来能够容忍复制失败。数据恢复并非恢复正常运行所必需，因此我们将其推迟到所有区域都处于活动状态时进行，以尽量减少对延迟关键锁恢复的影响。每台机器都会在其作为主区域的所有区域都处于活动状态时向 CM 发送 REGIONS-ACTIVE 消息。收到所有 REGIONS-ACTIVE 消息后，CM 会向配置中的所有机器发送 ALL-REGIONS-ACTIVE 消息。此时，FaRM 开始与前台操作并行恢复新备份的数据。

一个区域的新备份最初会有一个新分配和清零的本地区域副本。它将区域划分给并行恢复的工作线程。每个线程执行单边 RDMA 操作，每次从主区域读取一个数据块。我们目前使用 8 KB 的数据块，这个数据块足够大，可以有效利用网络，但又足够小，不会影响正常情况下的运行。为减少对前台性能的影响，我们将下一次读取安排在上一次读取开始后的一个随机间隔内（设置为 4ms）开始，以加快恢复速度。

每个恢复的对象在复制到备份之前都必须经过检查。如果对象的版本大于本地版本，备份会通过比较和交换锁定本地版本，更新对象状态，然后解除锁定。否则，对象已经或正在被一个事务更新，而该事务创建的版本大于或等于恢复的版本，则不会应用恢复的状态。

5.5 恢复分配器状态

FaRM 分配器将区域划分成块（1 MB），用作分配小对象的板块。分配器保存两份元数据：包含对象大小的块头和板块空闲列表。块头被复制

在分配新数据块时，将其分配给备份。这可确保故障发生后，它们在新主设备上可用。由于数据块标头用于数据恢复，因此新主设备会在收到 NEW- CONFIG-COMMIT 后立即将其发送给所有备份。这就避免了旧主设备在复制数据块标头时发生故障而导致的任何不一致。

板块空闲列表只保留在主节点，以减少对象分配的开销。每个对象的标头中都有一个位，在事务执行过程中，该位被分配置位，被释放清零。如第 4 章所述，对象状态的这种变化会在事务提交时复制。故障发生后，通过扫描区域中的对象，在新的主节点上恢复空闲列表。为尽量减少对事务锁恢复的影响，分配恢复在收到 ALL-REGIONS-ACTIVE 后开始，为尽量减少对前台工作的影响，每次扫描 100 个对象，每 100 µs 一次。在板块的空闲列表恢复之前，对象的取消分配都要排队等待。

6. 评估

6.1 设置

我们的实验平台由 90 台用于 FaRM 集群的机器和 5 台用于复制 Zookeeper 实例的机器组成。每台机器都有 256 GB DRAM 和两个运行 Windows Server 2012 R2 的 8 核英特尔 E5-2650 CPU。我们启用了超线程技术，前 30 个线程用于前台工作，其余 2 个线程用于租赁管理器。机器有两个 Mellanox ConnectX-3 56 Gbps Infiniband 网卡，每个网卡由不同插座上的线程使用，并由一个具有全双向带宽的 Mellanox SX6512 交换机连接。FaRM 被配置为使用 3 路复制（一个主路和两个备份），租用时间为 10 毫秒。

6.2 基准

我们使用两个事务基准来衡量 FaRM 的性能。我们针对 FaRM API 用 C++ 实现了这两个基准测试。由于 FaRM 使用对称模型来利用局部性，因此每台机器都同时运行基准代码并存储数据。每台机器都在同一进程中运行与 FaRM 代码链接的基准代码。今后，我们将使用 SQL 等安全语言编译应用程序，以防止应用程序漏洞破坏数据。

电信应用事务处理（Telecommunication Application Transaction Processing，TATP）[32] 是高性能主内存数据库的基准。每个数据库表都是以 FaRM 哈希表[16]的形式实现的。TATP 以读取为主。70% 的操作都是单行查找，使用 FaRM 的无锁读取 [16]。这些操作通常只需一次 RDMA 读取即可完成，不需要提交阶段。10% 的 操作会读取 2-4 行，需要在提交阶段进行验证。其余 20% 的操作是更新，需要完整的提交协议。由于 70% 的

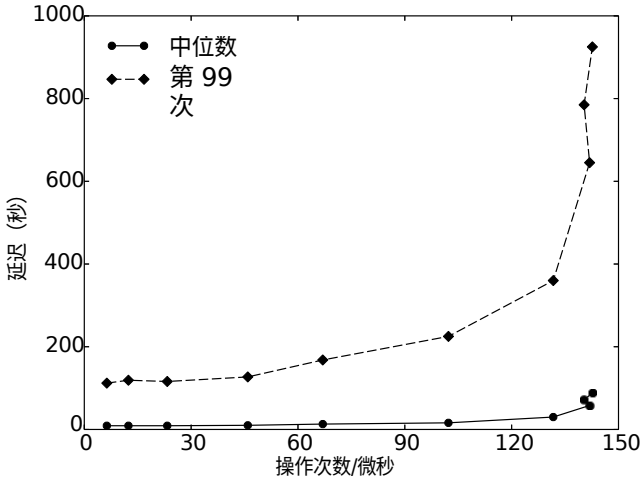


图 7.TATP 性能

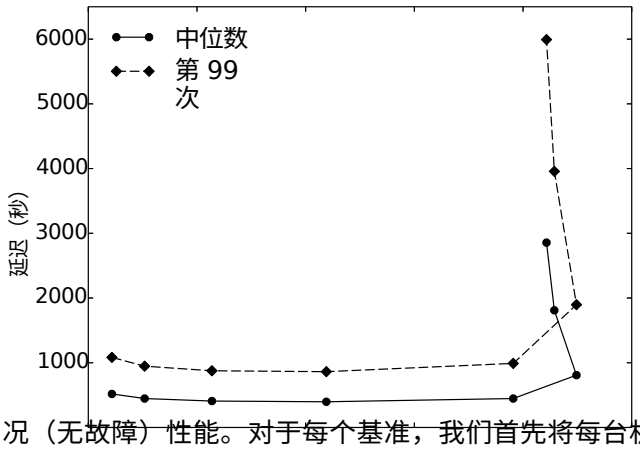
由于更新只修改了一个对象字段，因此我们将这些功能优化为对象的主功能。我们使用了一个拥有 92 亿用户的数据库（注释除外）。TATP 是可分区的，但我们没有对其进行分区，因此大多数操作都是访问远程机器上的数据。

TPC-C [38] 是一个著名的数据库基准，其中包含访问数百行的复杂事务。我们的实施方案使用了具有 16 个索引的模式。其中 12 个索引只需要无序（点）查询和更新，并以 FaRM 哈希表的形式实现。其中四个索引还需要范围查询。这些都是使用 FaRM B 树实现的。B 树会在每台机器上缓存内部节点，因此在普通情况下，查询只需要一次 FaRM RDMA 读取。我们为每台机器预留了 8 GB 的缓存空间。我们使用栅栏密钥 [17, 27] 来确保遍历的一致性，类似于 Minuet [37]。由于篇幅原因，我们省略了对 B 树的详细描述。

我们使用的数据库有 21600 个仓库。我们按仓库对大部分哈希表索引和客户端进行了联合分区，这意味着约有 10% 的事务访问远程数据。根据基准要求，"新订单 "交易占交易组合的 45%。我们运行了全部的交易组合，但以成功提交的 "新订单 "数量来报告性能。

6.3 正常情况下的性能

我们以吞吐量-延迟曲线的形式展示了 FaRM 的正常情况（无故障）性能。对于每个基准，我们首先将每台机器的活动线程数从 2 个增加到 30 个，然后增加每个线程的并发量，直至吞吐量达到饱和。请注意，每个曲线图的左端仍显示了大量并发，因此吞吐量也很大。它并不显示 FaRM 可以达到的最小延迟。



TATP。图 7 显示，FaRM 每秒可执行 1.4 亿次 TATP 事务，中位延迟为 58 微秒，而 TATP 事务的中位延迟则为 60 微秒。

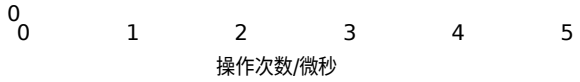


图 8.TPC-C 性能

645 µs 99th 百分位数延迟。在图表左侧，中位延迟仅为 9 微秒，99th 百分位延迟降至 112 微秒，FaRM 每秒执行 200 万次操作。TATP 使用的多对象分布式事务的提交时间为几十微秒，最低吞吐量时的平均提交延迟为 19 微秒，最高吞吐量时为 138 微秒。

FaRM 的性能比 Heka- ton [14, 26]（一种单机内存事务处理系统）公布的 TATP 结果高出 33 倍。Hekaton 的结果是在不同硬件上获得的，但我们预计，在我们的一台测试平台计算机上运行 Hekaton 时，性能将提高 20 倍。在规模较小的实验中，FaRM 的性能超过了仅使用三台机器的 Hekaton。此外，FaRM 可以支持更大的数据集，因为它可以扩展，而且与单机系统不同，它可以提供高可用性。

TPC-C我们运行了 60 秒的 TPC-C，并在图 8 中报告了在此期间的延迟和平均吞吐量。FaRM 每秒可处理 450 万个 TPC-C "新订单 "事务，中位延迟为 808 µs，99th 百分位延迟为 1.9 ms。延迟时间可缩短一半，但对吞吐量的影响仅为 10%。据我们所知，已公布的最佳 TPC-C 性能来自 Silo [39, 40]，这是一个单机内存系统，日志记录在 FusionIO SSD 上。FaRM 的吞吐量是不带日志记录的 Silo 的 17 倍，在此吞吐量水平下的延迟是带日志记录的 Silo 的 128 倍。[4]

读取性能。虽然本文的重点是事务性能和故障恢复，但与 [16] 相比，我们也提高了只读性能。我们使用 16 字节键和 32 字节值以及统一的访问模式运行了一个只进行键值查找的工作负载。我们的吞吐量达到 7.9 亿次/秒，中位延迟为 0.5 秒。

[4] Silo 报告总交易次数，我们将其乘以 45%，得出 "新订单 "次数。

为 23 微秒，99th 百分位数延迟为 73 微秒。这比之前报告的同一基准的单机吞吐量提高了 20% [16]。尽管网卡数量增加了一倍，但我们的性能并未翻倍，因为该基准会受到 CPU 的限制。

6.4 失败

为了评估故障时的性能，我们运行了相同的基准，并在实验开始 35 秒后杀死了其中一台机器上的 FaRM 进程。我们以 1 毫秒的时间间隔显示了 89 台幸存机器的吞吐量。时间线在实验开始时使用 RDMA 消息传递进行同步。

图 9 和图 10 显示了每个基准在不同时间尺度上的典型运行情况。两者均以实线显示吞吐量。达到满吞吐量的时间 "是故障周围的放大视图。它显示了故障机器在 CM 上的租约到期时间（"suspect"）；所有读取探针完成时间（"probe"）；CM 成功更新 Zookeeper 的时间（"zookeeper"）；新配置在所有幸存机器上提交的时间（"config-commit"）；所有区域都处于活动状态的时间（"all-active"）；以及后台数据恢复开始的时间（"data-rec-start"）。完全数据恢复时间 "显示的是放大视图，其中包括备份时恢复所有数据的时间（"完成"）。虚线显示了数据恢复随着时间推移恢复的备份区域的累计数量。

TATP。 典型 TATP 运行的时间轴如图 9 所示。我们将其配置为最大吞吐量：每台机器运行 30 个线程，每个线程并发 8 个事务。图 9(a) 显示，吞吐量在故障发生时急剧下降，但又迅速恢复。系统在不到 40 毫秒的时间内恢复到峰值吞吐量。所有区域在 39 毫秒内均处于活动状态。图 9(b) 显示，有节奏的数据恢复不会影响前台吞吐量。故障机器托管了 84 个 2 GB 区域。每个线程每 2 毫秒获取 8 KB 块，这意味着在单台机器上恢复一个 2 GB 的区域大约需要 17 秒。各台机器以大致相同的速度一次并行恢复一个区域，因此恢复的区域数量呈大步幅移动。整个集群的恢复负载（即每台机器上在故障机器上有副本的区域数量）非常均衡：64 台机器恢复一个区域，10 台机器恢复两个区域。这就解释了为什么大多数区域的重新复制在 17 秒左右完成，所有区域的完全重新复制在 35 秒内完成。这就是为什么某些区域的重新复制会在 17 秒内完成。

该图还显示，即使在没有故障的情况下，TATP 的吞吐量也会有所下降。我们认为，这是因为基准中的访问存在偏差。

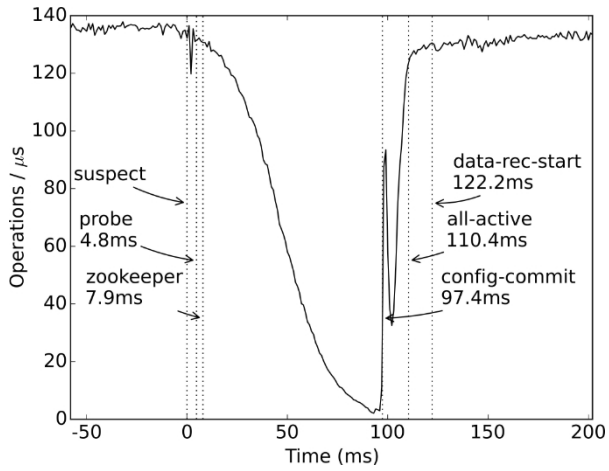


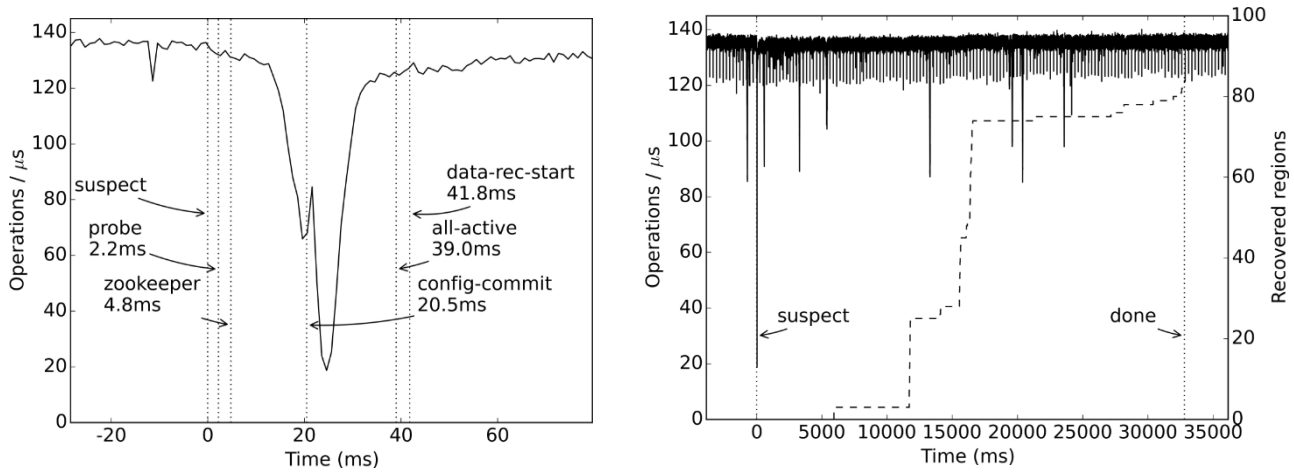
图 11.出现 CM 故障时的 TATP 性能时间表

当许多事务发生冲突并同时关闭热键时，吞吐量就会下降。

TPC-C.图 10 显示了 TPC-C 的时间线。图 10(a)显示，系统在不到 50 毫秒的时间内就恢复了大部分直通，之后不久所有区域都处于活动状态。由于 TPC-C 有更复杂的事务，系统重新覆盖事务锁所需的时间比 TATP 稍长。主要区别在于，尽管 TPC-C 在实验中只恢复了 63 个区域，但数据的重新恢复需要更长的时间（图 10(b)）。这是因为 TPC-C 为利用局部性和提高性能而对哈希表进行了联合分区，这导致恢复时间缩短，因为多个区域被复制到同一组机器上，以满足应用程序指定的局部性限制。在实验中，两台机器各恢复 17 个区域，数据恢复时间超过 4 分钟。请注意，在图 10(b) 中，TPC-C 吞吐量会随着时间的推移逐渐降低，因为数据库的大小会迅速增加。

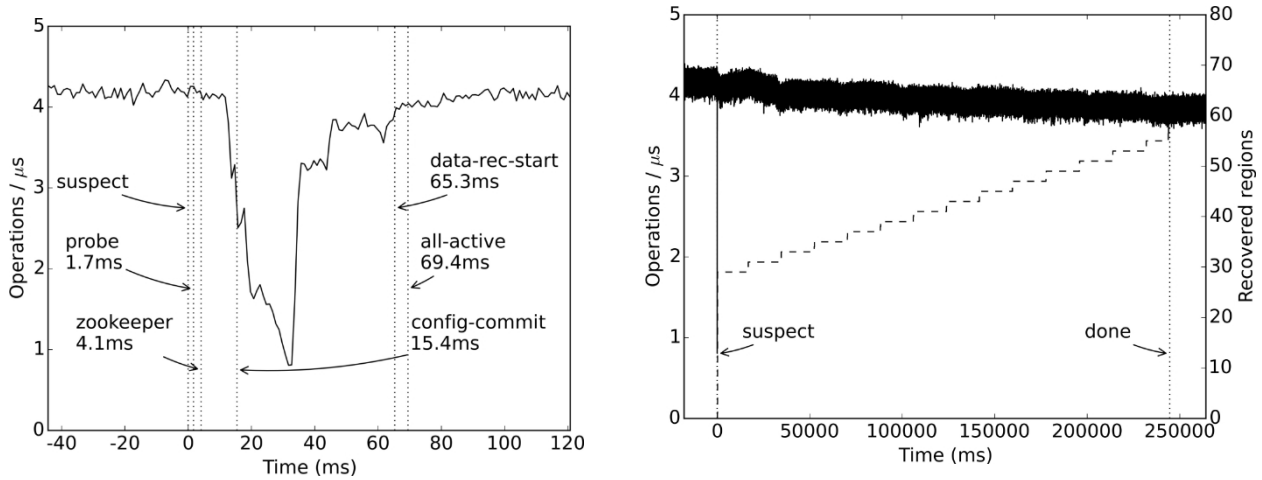
CM 故障。图 11 显示了当 CM 进程发生故障时，TATP 吞吐量随时间变化的情况。恢复速度比非 CM 进程发生故障时慢。吞吐量需要大约 110 毫秒才能恢复到故障前的水平。恢复时间增加的主要原因是重新配置时间的增加：从图 9(a) 中的 20 毫秒增加到 97 毫秒。其中大部分时间是新的 CM 构建仅在 CM 维护的数据结构所花费的。通过让所有机器在从 CM 学习区域映射的过程中逐步维护这些数据结构，应该可以消除这一延迟。

恢复时间的分布。我们重复了 40 次 TATP 恢复实验（无 CM 故障），以获得恢复时间的分布。为了缩短实验时间，我们使用了一个较小的数据集（35 亿用户）进行实验，但我们证实，恢复时间的分布是合理的。



(a) 达到全部吞吐量所需时间　　(b) 全部数据恢复所需时间

图 9.出现故障的 TATP 性能时间表



(a) 达到全部吞吐量所需时间　　(b) 全部数据恢复所需时间

图 10.出现故障时的 TPC-C 性能时间轴

故障后的吞吐量与较大数据集的吞吐量相同。这是因为这段时间主要用于恢复事务状态，而且两种数据集大小的并发执行事务数量相同。图 12 显示了恢复时间的分布。我们测量了从故障机器被 CM 检测到到吞吐量恢复到故障前平均吞吐量的 80% 的恢复时间。恢复时间的中位数约为 50 毫秒，在 70% 以上的执行中，恢复时间小于 100 毫秒。在其余情况下，恢复时间超过 100 毫秒，但总是少于 200 毫秒。

相关故障。有些故障会同时影响多台机器，如电源或交换机故障。为了处理此类协调故障，FaRM 允许为每台机器指定一个故障域，CM 会将一个区域的每个副本置于不同的故障域中。我们将集群中的机器分为五个故障域，每个故障域有 18 台机器。这相当于我们交换机中每个叶模块的端口数。我们同时让其中一个故障域中的所有进程失效，以模拟机架顶部交换机的故障。

图 13 显示了 72 台未发生故障的机器的 TATP 吞吐量随时间变化的情况。TATP 被配置为在每台机器上使用约 55 个区域（69 亿用户）。

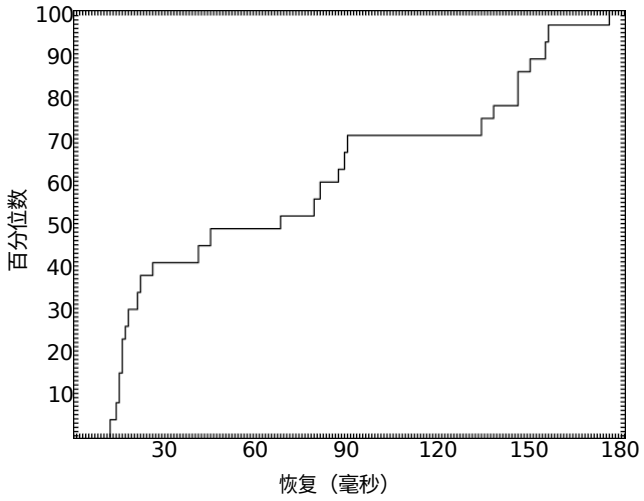


图 12.TATP 的恢复时间分布

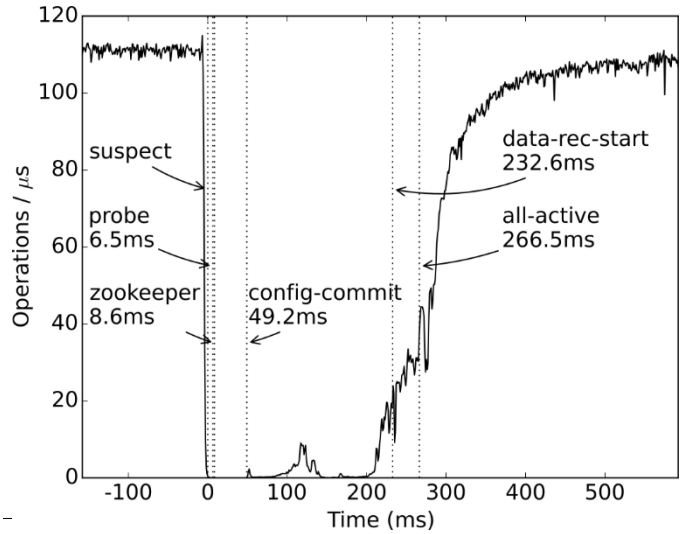
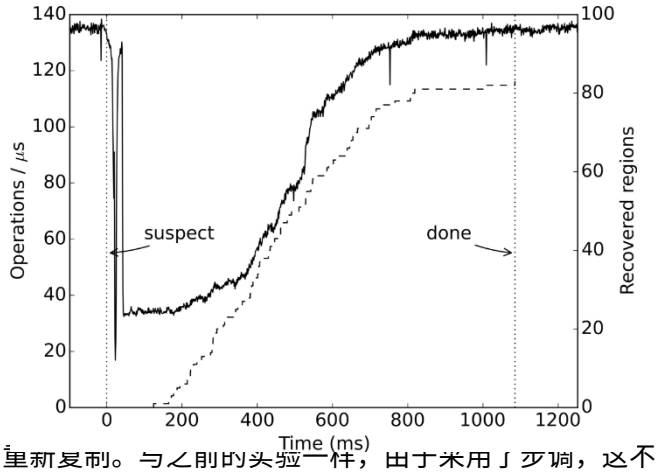


图 13.当 90 台机器中的 18 台同时出现故障时的 TATP 吞吐量

整个集群），以便有足够的空间在故障后重新复制故障区域。故障发生后不到 400 毫秒，FaRM 恢复峰值通量。我们重复了 20 次实验，这个时间是所有实验的中位数。其中大部分时间用于恢复事务。我们需要恢复所有飞行中的事务，这些事务需要修改故障机器中带有副本的任何区域，读取故障机器中带有主区域的区域，或者在故障机器之一上有协调者。这将导致大约 130,000 个事务需要恢复，而 单个故障仅需 7500 个。数据的重新复制需要 4 分钟，因为有 1025 个区域需要



重新复制。与之前的实验一样，由于采用了步调，这不会影响恢复期间的吞吐量。需要注意的是，在这段时间内，每个区域仍有两个可用副本，因此不需要更积极地重新复制。

图 14.优化再复制延迟时的 TATP 吞吐量

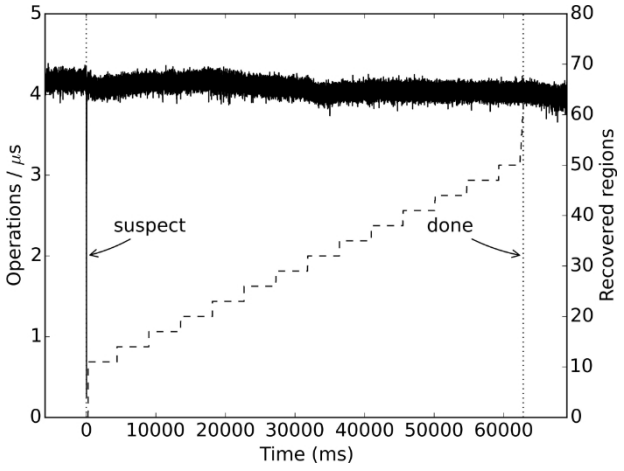


图 15.采用更积极数据恢复的 TPC-C 吞吐量

数据恢复步调。FaRM 对数据恢复进行调整，以减少其对吞吐量的影响。这增加了在新备份中完成区域复制的时间。图 14 显示了 TATP 在数据恢复非常积极的情况下，吞吐量随时间变化的情况：每个线程当前获取 4 个 32 KB 的数据块。系统只有在故障发生 800 毫秒后重新复制大部分区域后，才能恢复峰值吞吐量。然而，数据恢复的速度要快得多：恢复 83 个区域副本（166 GB）仅需 1.1 秒。与 RAMCloud [33]（在 80 台机器上恢复 35 GB 仅需 1.6 秒）相比，这种积极的恢复速度要快得多。

与 TATP 相比，TPC-C 对后台恢复流量的 干扰不那么敏感，因为只有一小部分访问是对远程机器上的对象进行的。这意味着，在可以针对特定应用进行调整的情况下，我们可以更积极地重新复制数据，而无需

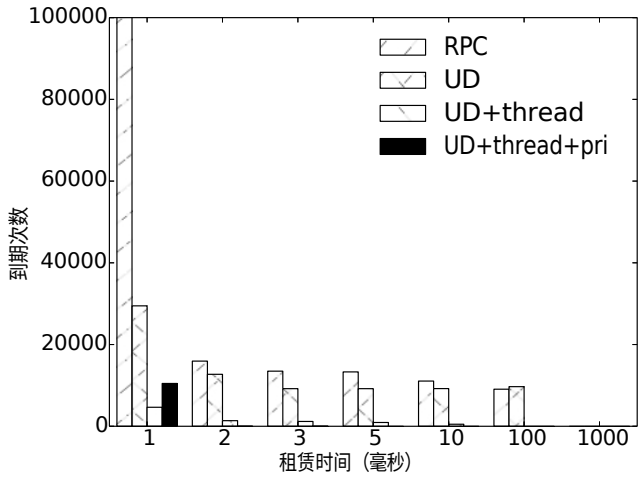


图 16.不同租赁管理器的误报率

影响性能。图 15 显示了线程每 2 毫秒获取 32 KB 数据块时，恢复过程中 TPC-C 吞吐量随时间变化的情况。重新复制在 65 秒内完成，比默认设置快四倍，且未对吞吐量造成任何影响。

6.5 租赁时间

为了评估租赁管理器的优化效果（第 5.1 节），我们进行了一项实验，让所有机器上的所有线程在 10 分钟内反复向 CM 发送 RDMA 读取。我们禁用了恢复功能，并针对不同的租约管理器实现和不同的租约持续时间，统计了整个集群中（假阳性）租约到期事件的数量。这个基准是一个很好的压力测试，因为它在 CM 上产生的流量比我们描述的任何基准都要大。

图 16 比较了四种租赁管理器的实现。第一种使用 FaRM 的 RPC（RPC）。其他几种则使用不可靠数据报：共享线程（UD）、正常优先级的专用线程（UD+thread），以及高优先级、in- terrupts 和无 pinning 的线程（UD+thread+pri）。

结果表明，所有的优化都是必要的，这样才能使用 10 毫秒或更短的租用时间，而不会出现误报。在共享

在所有实验中，我们都保守地将间隔时间设定为 10 毫秒，并且在执行过程中未发现任何误报。

7. 相关工作

据我们所知，FaRM 是第一个同时提供高可用性、高吞吐量、低延迟和严格序列化的系统。在之前的工作[16]中，我们概述了FaRM的早期版本，该版本将日志记录到固态硬盘上，以提高耐用性和可用性，但我们没有介绍故障恢复。本文介绍了一种新的快速恢复协议，以及一种优化的事务和复制协议，该协议发送的信息量大大减少，并利用 NVRAM 避免记录到固态硬盘。优化后的协议 发送的信息量比事务和复制协议少 44%。

队列对的情况下，即使 100 毫秒的租用时间也会经常出现。通过使用不可靠的数据报，误报的数量有所减少，但由于对 CPU 的争夺，误报并没有消除。使用专用线程可让我们在使用 100 毫秒租期时不会出现误报，但由于在 FaRM 机器上运行的后台程序会造成 CPU 竞争，10 毫秒租期仍会过期。如果中断驱动的租期管理器以高优先级运行，我们就可以在 10 分钟内使用 5 毫秒的租期，而不会出现误报。如果租期较短，有时仍会出现误报。我们受到网络往返时间和系统定时器分辨率的限制，网络往返时间在负载情况下高达 1 毫秒，系统定时器分辨率为 0.5 毫秒。系统定时器的分辨率有限，这就解释了为什么中断驱动的租约管理器比基于轮询的 1 毫秒租约管理器有更多的误报。

16] 中描述的协议，并在验证阶段用单边 RDMA 读取代替消息。文献[16]中的研究仅使用 YCSB 基准评估了无故障情况下的单键事务处理性能。在这里，我们使用 TATP 和 TPC-C 基准评估了有故障和无故障情况下的事务性能。

RAMCloud [33, 34] 是一种键值存储，它在内存中存储单个数据副本，并使用分布式日志来提高耐用性。它不支持多对象事务。发生故障时，它会在多台机器上并行恢复，在此期间（可能需要几秒钟），故障机器上的数据不可用。FaRM 支持事务，可在故障发生后几十毫秒内提供数据，而且每台机器的吞吐量要高出一个数量级。

第 4 节讨论了 Spanner [11]。它提供了严格的序列化能力，但没有针对 RDMA 性能进行优化。与 FaRM 的 $f + 1$ 相比，它使用了 $2f + 1$ 个副本，而且发送的提交消息比 FaRM 多。Sinfonia[8]提供了一个共享地址空间，其可序列化事务是通过 2 阶段提交和在特殊情况下将读取捎带到 2 阶段提交来实现的。FaRM 提供一般的分布式事务，并对 RDMA 进行了优化。

HERD [23] 是一种基于 RDMA 的内存键值存储，在客户端与服务器运行在不同机器上的异构环境中，每台服务器都能提供高性能。它使用 RDMA 写入和发送/接收动词进行消息传递，但不使用 RDMA 读取。文献[23]的作者指出，在非对称环境下，单边 RDMA 读取的性能比没有可靠性的专门 RPC 实现要差。我们的研究结果在对称环境中使用了可靠的通信，在对称环境中，每台机器既是客户端也是服务器。这使我们能够利用本地性，这一点非常重要，因为访问本地 DRAM 的速度明显快于使用 RDMA 访问远程 DRAM [16]。Pi- laf [31] 是一种使用 RDMA 读取的键值存储。Pilaf 和 HERD 都不支持事务。HERD 不支持容错，而 Pilaf 则通过将日志记录到本地磁盘来获得耐久性，但不支持可用性。

Silo [39, 40] 是一种单机主内存数据库，它通过将日志记录到持久性存储来实现持久性。它将已提交的事务分批写入存储，以实现高吞吐量。故障恢复包括从存储中读取检查点和日志记录。Silo 中的存储是本地的，因此当机器发生故障时，可用性就会丧失。相比之下，FaRM 是分布式的，并使用 NVRAM 中的复制来实现耐用性和高可用性。对于规模更大的数据库，FaRM 在故障后恢复峰值吞吐量的速度比 Silo 快两个数量级以上。通过扩展和使用 NVRAM 中的复制，FaRM 还能实现比 Silo 更高的吞吐量和更低的延迟。Hekaton [14, 26] 也是一种单机主内存数据库，不支持扩展或分布式传输。使用 3 台机器的 FaRM 与 Hekaton 的性能相当，而使用 90 台机器的 FaRM 的吞吐量是 Hekaton 的 33 倍。

8. 结论

事务使分布式系统的编程变得更容易，但许多系统为了提高可用性和性能，避免使用事务或削弱事务的一致性。FaRM 是面向现代数据中心的分布式主存储器计算平台，可提供严格的可序列化事务，具有高吞吐量、低延迟和高可用性的特点。实现这一目标的关键是新的事务、复制和恢复方案，这些方案的设计初衷是利用具有 RDMA 功能的商品网络，以及一种提供非易失性 DRAM 的新型廉价方法。实验结果表明，与最先进的内存数据库相比，FaRM 的吞吐量明显更高，延迟时间更短。FaRM 还能在不到 50 毫秒的时间内从机器故障中恢复到提供峰值吞吐量，使故障对应用程序透明。

致谢

我们要感谢聂杰森（Jason Nieh）、我们的牧羊人以及匿名审稿人提出的意见。We would also like to thank Richard Black for his help in performance debugging, Andy Slowey and Oleg Losinets for keeping the test cluster running, and Chiranjeeb Buragohain, Sam Chandrashekar, Arlie Davis, Orion Hodson, Flavio Jun- queira,Richie Khanna、James Lingard、Samantha Lu¨ber、Knut Magne Risvik、Tim Tan、Ming Wu、Ming-Chuan Wu、Fan Yang 和 Lidong Zhou 进行了大量讨论，并让我们长时间使用整个集群来运行最终实验。

参考资料

[1] http://memcached.org。

[2] 海盗 技术。 http://www.vikingtechnology.com/.

[3] Apache Cassandra .http://cassandra.apache.org/, 2015.

[4] MySQL。http://www.mysql.com/，2015 年。

[5] neo4j. http://neo4j.com/, 2015.

[6] redis。http://redis.io/，2015 年。

[7] ADYA, A., DUNAGAN, J., AND WOLMAN, A. Centrifuge：云服务的综合租赁管理和分区。 *第七届 USENIX 网络系统设计与实施研讨会*（2010 年）*论文集*，NSDI'10。

[8] aguilera, M. K., merchant, A., shah, M., veitch、A., AND KARAMANOLIS, C. Sinfonia：构建可扩展分布式系统的新范式。In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP'07.

[9] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes、A., AND GRUBER, R. E. Bigtable：结构化数据的分布式存储系统。*第 6 届 USENIX 操作系统设计与实现研讨会*（2006）*论文集*，OSDI'06。

[10] CHOCKLER, G. V., KEIDAR, I., AND VITENBERG, R. Group 通信规范：一项综合研究》。*ACM Computing Surveys (CSUR) 33*, 4 (2001).

[11] Corbett, J . C., Dean, J., Epstein, M., Fikes, A., frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W. C., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura、D., NAGLE, D., QINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner：谷歌的全球分布式数据库。*第 10 届 USENIX 操作系统设计与实现研讨会论文集*（2012 年），OSDI'12。

[12] DALESSANDRO, L., AND SCOTT, M. L. Sandboxing transactional memory.*第 21 届 ACM 并行架构与编译技术国际会议论文集*（2012 年），PACT'12。

[13] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo：Amazon的高可用键值存储。*第 21 届 ACM 操作系统原理研讨会*（2007 年）*论文集*，SOSP'07。

[14] diaconu, c., freedman, c., ismert, e., larson, p.- a˚ ., mittal, p., stonecipher, r., verma, n., and ZWILLING，M. Hekaton：SQL Server 的内存优化 OLTP 引擎。 In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2013), SIGMOD'13.

[15] DICE, D., SHALEV, O., AND SHAVIT, N. Transactional locking II.第 *20 届分布式计算国际研讨会论文集*（2006 年），DISC'06。

[16] Dragojevic´ , A., Narayanan, D., HODSON, O., AND CASTRO, M. FaRM：快速远程内存。*第 11 届 USENIX 网络系统设计与实现会议*（2014 年）*论文集*，NSDI'14。

[17] GRAEFE, G. Write-optimized B-trees.*第 30 届超大型数据库国际会议论文集*（2004 年），VLDB'04。

[18] GRAY, C., AND CHERITON, D. Leases：分布式文件缓存一致性的高效容错机制。*SIGOPS Operating Systems Review (OSR) 23*, 5 (1989).

[19] GRAY, J., AND REUTER, A. *Transaction Processing：Concepts and Techniques*.1992.

[20] GUERRAOUI, R., AND KAPALKA, M. On the correctness of transactional memory.In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (2008), PPoPP'08.

[21] HUNT, P., KONAR, M., JUNKEIRA, F. P., AND REED, B. Zookeeper：互联网规模系统的无等待协调。*2010 USENIX 年度技术大会*（2010）*论文集*，USENIX ATC'10。

[22] Infiniband 贸易协会。Infini-Band 架构规范第 1 卷第 1.2.2 版第 A16 节的补充：聚合以太网（RoCE）上的 RDMA，2010 年。

[23] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using 针对键值服务的高效 RDMA。在 *2014 年计算机通信应用、技术、架构和协议会议*（2014 年）*论文集* SIGCOMM'14 中。

[24] LAMPORT, L. The part-time parliament.*ACM Transactions on Computer Systems 16*, 2.

[25] LAMPORT, L., MALKHI, D., AND ZHOU, L. 垂直 Paxos 和主备复制。*第 28 届 ACM 分布式计算原理研讨会论文集*（2009 年），PODC'09。

[26] Larson, P.-A° ., Blanas, S., Diaconu, C., freedman、C., PATEL, J. M., AND ZWILLING, M. High-performance concurrency control mechanisms for main-memory databases.*PVLDB 5*, 4 (2011).

[27] LEHMAN, P. L., AND YAO, S. B. Efficient locking for concurrent operations on B-trees.*ACM Transactions on Database Systems 6*, 4 (Dec. 1981).

[28] 微软。 扩展 SQL Server：//www.microsoft.com/en-us/server-cloud/solutions/high-availability.aspx 。

[29] 微软。开放云服务器 OCS V2 规范：Blade, 2014.

[30] 微软。 OCS Open CloudServer power sup- ply v2.0。http://www.opencompute.org/wiki/Server/SpecsAndDesigns，2015。

[31] MITCHELL, C., YIFENG, G., AND JINYANG, L. Using one-侧 RDMA 读取来构建快速、CPU 效率高的键值存储。*2013 USENIX 年度技术会议*（2013）*论文集*，USENIX ATC'13。

[32] Neuvonen, S ., Wolski, A . , manner, M ., and RAATIKKA, V. Telecom Application Transaction Pro- cessing benchmark .http://tatpbenchmark.sourceforge.net/.

[33] Ongaro, D., Rumble, S. M., Stutsman, R., Ouster-HOUT, J., AND ROSENBLUM, M. Fast crash recovery in RAMCloud.*第 23 届 ACM 操作系统原理研讨会论文集*（2011 年），SOSP'11。

[34] Rumble, S. M., Kejriwal, A., and Ousterhout, J. 基于 DRAM 存储的日志结构存储器。*第 12 届 USENIX 文件与存储技术会议*（2014 年）*论文集*，FAST'14。

[35] SETHI, R. 无用的行动会带来不同的结果：数据库更新的严格序列化能力。*JACM 29*, 2 (1982).

[36] SHAUN HARRIS. 微软通过新的开放计算项目规范重塑了数据中心电源备份。http://blogs.msdn.com/b/windowsazure/archive/2012/11/13/windows-azure-benchmarks-show-top-performance-for-.big-compute.aspx，2015 年。

[37] SOWELL, B., GOLAB, W. M., AND SHAH, M. A. Minuet：A 可扩展分布式多版本 B 树。*PVLDB 5*, 9 (2012).

[38] 事务处理性能理事会 (TPC）。TPC 基准 C：标准规范：//www.tpc.org.

[39] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. 多核内存数据库中的快速事务处理。*第 24 届操作系统原理研讨会论文集*（2013 年），SOSP'13。

[40] ZHENG, W., TU, S., KOHLER, E., AND LISKOV, B. Fast 通过多核并行实现快速持久性和恢复的数据库。*第 11 届 USENIX 操作系统设计与实现研讨会*（2014 年）*论文集*，OSDI'14。