文章编号: 1000-5641(2018)05-0091-16

一致性协议在分布式数据库系统中的应用

赵春扬, 肖 冰, 郭进伟, 钱卫宁

(华东师范大学 数据科学与工程学院, 上海 200062)

摘要: 近年来分布式数据库产品层出不穷, 但分布式数据库较于单机数据库更复杂, 为了让系统可用, 设计者需要采用一致性协议来保证分布式数据库系统中的可用性和一致性这两个重要特性. 保证一致性需要使用一致性协议为并发的事务更新操作确定一个全局的执行顺序, 并协调局部状态和全局状态不断的达成动态一致; 保证可用性需要一致性协议协调多副本之间的一致来实现主备节点的无缝切换. 因此分布式一致性协议是实现分布式数据库系统的重要基础. 详细介绍了经典的分布式一致性协议以及在目前常见的几种分布式数据库系统中一致性协议的应用, 并从读写操作、节点类型与网络通信等方面进行对比分析.

关键词: 分布式数据库; 分布式一致性协议; 可用性; 一致性

中图分类号: TP392 **文献标志码**: A **DOI:** 10.3969/j.issn.1000-5641.2018.05.008

Application of the consistency protocol in distributed database systems

ZHAO Chun-yang, XIAO Bing, GUO Jin-wei, QIAN Wei-ning

(*School of Data Science and Engineering, East China Normal University, Shanghai* 200062, *China*)

Abstract: In recent years, many distributed database products have emerged in the market; yet, distributed databases are still more complex than centralized databases. In order to make the system useable, designers need to adopt the consistency protocol to ensure two important features of distributed database systems: availability and consistency. The protocol ensures consistency by determining the global execution order of operations for concurrent transactions and by coordinating local and global states to achieve continuous dynamic agreement; The consistency protocol ensures availability by coordinating consistency between multiple copies to achieve seamless switching between master and standby nodes. Hence, the distributed consensus protocol is the fundamental basis for the distributed database system. This paper reviews, in detail, the classic distributed consistency protocol and the application of the consistency protocol to current mature distributed database. The study also provides analysis and a comparison between

收稿日期: 2018-06-27

基金项目: 国家自然科学基金重点项目 (61332006); 国家 863 计划项目 (2015AA015307)

第一作者: 赵春扬, 男, 硕士研究生, 研究方向为数据库系统. E-mail: zhaochunyang@stu.ecnu.edu.cn.

通信作者: 钱卫宁, 男, 教授, 博士生导师, 研究方向为社交媒体数据管理与分析、互联网环境的数据管理与基准评测、基于开放信息的知识图谱构建与利用等. E-mail: wnqian@dase.ecnu.edu.cn.

the two approaches considering factors like read-write operation, node type, and network communication.

Keywords: distributed database; distributed consistency protocol; consistency; availability

0 引 言

分布式系统是若干独立计算机的集合[1], 独立的计算机之间通过网络连接形成一个整体对外服务. 随着数据量的极速增长和互联网应用的繁荣发展, 越来越多的数据库系统通过数据副本或者数据分区的方式提供可扩展的数据存储与服务. 分布式数据库系统在数据的容量、持久性和可用性上提供了很好的支持, 但是也带来了新的问题: 一是物理服务器可能发生故障失效, 二是网络资源是昂贵的而且网络也有可能出现不可靠的问题.

根据分布式系统环境, 可将故障分为两类: 节点故障和网络故障. 针对节点故障, 分布式数据库系统通常采用副本的方式, 将副本存储于不同的节点, 以削弱:

- 单点故障对可靠性/可用性带来的影响.
- 单点过载瓶颈对可扩展带来的影响.
- 网络中的通信延迟与失败对容错性带来的影响.

然而当多个副本分布在不同的网络分区中时, 受到网络故障的影响, 对一个副本的写入可能会无法同步到其他副本, 那么读取不同的副本将会返回不一致的结果. 此外在进行读写操作时, 由于不同的复制协议涉及不同的更新时机、不同的系统体系结构等内容, 多个副本又可能带来数据不一致的问题. 因此副本在提高了系统可用性的同时, 又带来了一致性问题.

著名的CAP定理[2]展示了可用性与分布式一致性之间的关系. 2000年, Eric Brewer提出了该定理, 解释了分布式系统的基本准则. Giler和Lynch则从理论上证明了CAP定理的正确性[3]. CAP定理说明了在一个分布式系统中, 一致性、可用性、分区容错性, 三者不可兼得. CAP定理中的一致性指分布式一致性, 即分布式系统中的所有数据副本, 在任一时刻是否具有相同的值. 一致性模型[4]可分为强一致性、弱一致性和最终一致性. 具体的区别如下.

- 强一致性: 每次读操作都返回之前最后一次写操作写入的内容.
- 弱一致性: 只关注主副本是否更新完成. 一次写入操作W完成后, 系统无法保证之后开始的读取操作都能获得W写入的内容. 在弱一致的前提下, 系统仍然可以采取措施减少不一致所带来的异常. 典型的方案是保证最终一致性.
- 最终一致性: 即便多个副本的状态没有充分同步, 每个副本依然可以处理读写操作.使用最终一致性需要容忍一段时间内可能发生读取数据的不一致, 但在一段时间后所有副本能够被充分同步. 保证了较高的系统可用性.

CAP定理中的可用性指的是, 当分布式系统中的节点发生故障时, 系统是否还能正常响应客户端的读写请求. CAP定理中分区容错性指的是, 系统应确保能在网络异常时仍正常使用, 除非整个网络全部瘫痪.

在分布式系统中, 由于网络硬件可能会出现延迟丢包等问题, 导致分区容忍性是必须要实现的, 所以只能在一致性和可用性之间进行权衡. 为了保证数据强一致性, 副本之间需要时刻保持强同步, 但是当某一副本出现故障时, 可能阻塞系统的正常写服务, 从而影响到系统的可用性. 如果各副本之间不保持强同步, 虽然系统的可用性相对较好, 但是强一致性却

得不到保障, 当某一副本出现故障时, 数据还可能丢失. 因此为了满足不同的应用需求需要在可用性与一致性之间进行权衡[5].

为了保证系统可用, 关键是选取合适的分布式一致性协议, 以更好地权衡可用性与一致性间的关系. 保证一致性需要使用一致性协议为并发的事务更新操作确定一个全局的执行顺序, 并协调局部状态和全局状态不断的达成动态一致; 保证可用性需要一致性协议协调多副本间的一致来实现主备节点的无缝切换. 因此分布式一致性协议是实现分布式数据库系统的重要基础. 根据不同的应用需求选择不同的分布式一致性协议是保证系统可用的关键[6].

综上可见, 分布式一致性协议是实现强一致性和高可用性的重要基础. 本文梳理总结了经典的分布式一致性协议以及其在当前工业应用场景实现过程中的考虑, 并进行了分析和对比. 第 1 节简要介绍了经典的分布式一致性协议; 第 2 节介绍了常见的分布式数据库系统中分布式一致性协议实现过程中的考虑; 第 3 节总结全文, 并对未来的研究工作进行了展望.

1　分布式一致性协议概述

为了解决分布式环境下的一致性问题, 前人研究设计了许多策略, 比如 Quorum[7]、Raft[8]、Zab[9]、Basci Paxo[10-11] 以及基于 Basci Paxos 衍生的变种算法 Cheap Paxos[11-12]、Fast Paxos[13]、Vertical Paxos[14] 等. 其中 Zab 协议主要用于构建一个高可用的主备分布式数据库系统, 而 Paxos 主要用于构建一个分布式的一致性状态机系统, 本文主要介绍 Quorum、Raft 和 Basci Paxos. 这些分布式一致性协议如今已经在工业界得到广泛应用, 并衍生出许多变种的策略, 以更好地适应不同的应用场景, 如阿里巴巴的 XPaxos、微信的 PaxosStore 等. 本节会详细介绍几种经典的分布式一次性协议, 并在下节介绍分布式一致性协议在几种分布式数据库中的应用. 分布式一致性协议的发展历程见图 1.

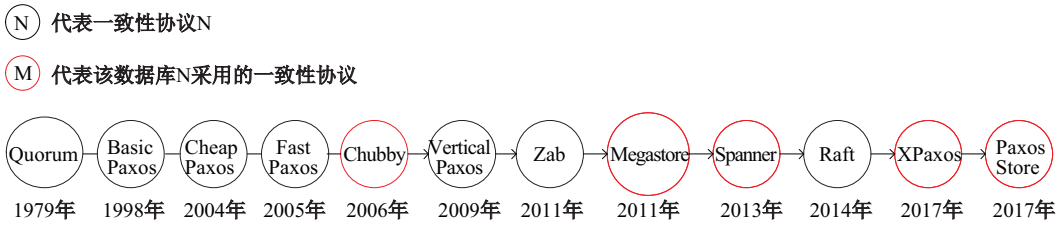

图 1　分布式一致性协议

Fig. 1　Distributed consistency protocol

1.1　Quorum

Quorum 算法由 Gifford 于 1979 年提出, 可实现不同级别的一致性. 其主要思想来源于“抽屉原理”. 抽屉原理的一般含义为, 如果每个抽屉代表一个集合, 每一个苹果就可以代表一个元素, 假如有 n+1 个元素放到 n 个集合中去, 其中必定有一个集合里至少有两个元素.

1.1.1　Quorum 算法

Quorum 算法是“抽屉原理”的一个应用, 其主要思想为多数派思想, 所谓多数即超过半数. 现考虑这样一个场景: 假设系统中某数据有 N 个副本, 对于写操作, 要求至少要在 W 个副本上更新成功后, 才认为此次写操作成功; 对于读操作, 至少读取 R 个副本才认为此次读操作成功. 因此为了保证读操作每次都能读取到最新写操作成功的数据, Quorum 要求 $W+R>N$, 即写入 W 个副本与读取 R 个副本之间有重叠的副本. W 和 R 的关系见图 2.

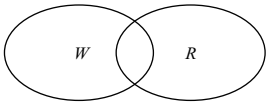

图2 W 和 R 的关系

Fig. 2 The relationship between W and R

1.1.2 Quorum 算法应用

现考虑如下应用场景: 假设系统中某数据副本数 N=5, W=3, R=3, 令 V_n 表示第 n 次写操作写入的数据. 初始状态数据 5 个副本分别为 (V_0, V_0, V_0, V_0, V_0), 版本号为 0. 现在第一次写操作在 3 个副本上更新成功, 数据变为 (V_1, V_1, V_1, V_0, V_0), 版本号为 1, 即认为写操作成功. 因此接下来读操作读取 3 个副本的数据时, 一定可以读取到第一次写操作写入的数据 V_1. 写操作成功后, 可继续将 V_1 同步到剩余的 V_0, 而不需要让客户端知道.

然而仅仅依靠上述 Quorum 算法并不能保证正常的读写服务. 如当第一次写操作只在两个副本上更新成功, 数据变为 (V_1, V_1, V_0, V_0, V_0) 时, 写操作失败. 接下来的读操作可能读取到 (V_1, V_1, V_0) 或者 (V_1, V_0, V_0), 依旧可以读取到 V_1. 即仅仅通过一次读操作可能读取到第 n 次写操作写入的数据 V_n, 但不能确保 V_n 已经提交. 为了确保 V_n 是已经提交的数据, 可以继续读其他副本, 直到读到的 V_n 副本数为 W.

基于 Quorum 机制选取主节点时, 可首先读取 R 个副本, 选择 R 个副本中版本号最高的副本作为主节点, 新选出的主节点不能立即提供服务, 还需要至少与 W 个副本完成同步后, 才能提供服务.

1.2 Paxos

Paxos 是一种基于消息传递模型且具有高度容错特性的一致性算法, 同时也是目前公认的解决分布式一致性问题最有效的算法之一, 可保证强一致性. 最早的 Basic Paxos 由 Leslie Lamport 于 1998 年提出. 目前 Paxos 及其衍生算法广泛应用于工业界中, 典型的有 Google 的 Megastore、Spanner、微信的 PaxosStore、阿里的 XPaxos 等. 本节主要介绍 Basic Paxos 的内容.

1.2.1 问题场景

在分布式系统中, 针对同一个数据, 不同的进程可能会提出不同的值. 而常见的分布式系统难以避免诸如机器宕机或网络异常等问题. Paxos 需要解决如何在可能发生上述异常的分布式系统中, 快速且正确地对系统内某个数据的值达成一致. Paxos 不考虑拜占庭问题[15].

1.2.2 Basic Paxos 过程

Basic Paxos 根据算法过程中进程的功能将进程分为以下 3 种角色.

• proposer: 提出议案. 议案包含一个议案编号和一个数据值, 该值需要各进程之间最终达成一致. proposer 可以有多个, 不同的 proposer 可以提出不同的议案, 不同议案的编号一定不同, 但数据的值可以相同.

• acceptor: 接收议案并决定是否批准. acceptor 可以批准多个议案, 但要求这些议案具有相同的值. acceptor 需要有多个.

• learner: 将目前已被选中的数据值同步给其他尚未批准该值的 acceptor.

具体实现中, 同一时刻一个进程可能扮演上述多种角色. 一轮 Basic Paxos 流程可分为 PrepareRequest 和 AcceptRequest 两个阶段, 具体如下, 为方便描述现定义几个符号.

• $P = (n, v)$, P 代表议案, n 代表议案编号, v 代表议案中数据值.

• MaxN, acceptor 已经回复过的所有 PrepareRequest 中的最大编号.

• MaxP, acceptor 已经批准过的所有议案中编号最高的议案.

(1) PrepareRequest 阶段

a) 当一个 proposer 准备提出议案时, proposer 首先会向超过半数的 acceptor 发送一个新的编号为 n 的 PrepareRequest.

b) acceptor 收到编号为 n 的 PrepareRequest 时, 如果 n 大于MaxN, 则该 acceptor 将不再接收任何编号小于 n 的请求, 并回复 MaxP (如果有的话). 如果 n 小于 MaxN, 则 acceptor 会通知对应的 proposer 放弃本次提议.

(2) AcceptRequest 阶段

a) 如果 proposer 收到超过半数 acceptor 的回复, 会给超过半数 acceptor 发送一个包含议案 P=(n, v) 的 AcceptRequest. 如果这些回复中包含议案, 则 v 是所有回复过来的议案中编号最高议案的数据值; 如果回复中没有议案, 则 v 是 proposer 原本准备赋予数据的值. 如果 proposer 未收到超过半数 acceptor 的回复, 则放弃本次提议.

b) acceptor 收到编号为 n 的 AcceptRequest 时, 如果 n 大于 MaxN, 则 acceptor 批准该 AcceptRequest 中的议案. 如果 n 小于 MaxN, 则 acceptor 会通知对应的 proposer 放弃本次提议.

当 proposer 在 AcceptRequest 中的议案被超过半数 acceptor 批准时, 该议案中的值即为达成一致的数据的值. 此后 Basic Paxos 使用 learner 来找到被超过半数 acceptor 批准的议案, 并将该议案同步给其他尚未批准该议案的 acceptor 批准该议案, 到此一轮 Basic Paxos 结束.

1.2.3　基于 leader 的 Multi Paxos

由于 Basic Paxos 存在多 proposer 发起议案导致复杂情况、网络 I/O 多、延迟高等问题, 为了便于工业实现, 提高性能, 后人提出了基于 leader 的 Multi Paxos, 具体如下.

集群初始状态没有 leader, 经过一轮 Basic Paxos 后, 获得超过半数 acceptor 支持的 proposer 成为唯一的 leader, 所有议案都只能由 leader 发起. 由于没有了并发冲突, leader 在发起提案时, 不必每次都进入 PrepareRequest 阶段, 只需直接执行 AcceptRequest 阶段.

在 Multi Paxos 中, 可将最初选主过程中的 PrepareRequest 阶段视为对 leader 任期内将要发起的所有议案的一次性 prepare 操作, 在 leader 任期内发出的所有议案都携带相同的编号. Basic Paxos 与基于 leader 的 Multi Paxos 的区别参见表 1.

表 1　Basic Paxos 与基于 leader 的 Multi Paxos 的区别

Tab. 1　Differences between Basic Paxos and Multi Paxos based on leader

Basic Paxos	Multi Paxos
达成一致性的流程可分为 Prepare Request 和 Accept Request 两阶段.	选主运行一次 Basic Paxos. leader 任期内达成一致性只需执行 AcceptRequest 阶段即可.
无 leader, 所有proposer 都可以发起议案, 容易造成复杂情况, 性能低.	只有 leader 才能发起议案, 逻辑简单, 性能高, 存在单点故障问题.
proposer 只有在收到多数派 PrepareRequest 回复成功的消息时, 才能确定议案中的值.	Leader 可以直接可以确定议案中的值.
达成一致性至少需要两轮网络 I/O, 延迟高	需要网络 I/O 较少, 延迟低

大量的理论已经证明了基于 leader 的 Multi Paxos, 性能好于 Basic Paxos. 当前大部分的基于 Paxos 的分布式数据库系统, 如 Chubby、Spanner、XPaxos 等都采用这种方法.

1.3 Raft

Raft 是一种基于Paxos的用于管理重复日志的分布式一致性协议, 由Diego Ongaro和John Ousterhout于2014年提出, 可保证强一致性. 相较于Paxos, Raft更加容易理解和工程实现[16]. 本节将对Raft算法进行简要描述, 并分析指出Raft应用过程中可能遇到的问题.

1.3.1 问题场景

Raft将分布式数据库系统中的节点抽象化为复制状态机. 初始状态假设各个状态机状态相同, 在系统运行期间, 只要每个状态机按照相同顺序执行同样操作, 那么最终状态就是一致的. 因此保证操作和执行顺序的一致性, 就能保证状态机节点数据的一致性. 本节分leader选举和日志复制两部分介绍Raft算法.

1.3.2 leader选举

Raft引入"任期(Term)"作为判断时间顺序的标识, 每个节点存储一个任期值. 如图3所示, Raft将时间分为任意长度的任期, 每次选举开始代表进入一个新的任期, 任期编号随之递增. 任期具有传递性, 节点会拒绝具有更小任期节点的消息, 而当节点收到具有更大任期节点的消息时, 会将自己的任期更新为这个更大的任期.

Raft保证任期内当选的leader存储着已经提交的最新的日志. Raft使用日志号唯一标识一条日志, 日志号越大说明日志越新. 一个节点在一个指定的任期内最多只能投一票且先来先得. 当节点接收到投票请求后, 除了判断任期编号以及自己是否已经投过票, 还会比较自己和请求投票者的日志号大小, 只有当前者不大于后者时才会投票, 否则拒绝投票.

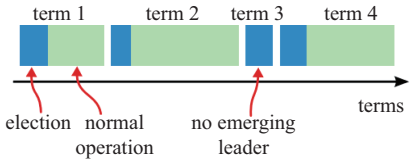

图3 任期

Fig. 3 Term

在Raft算法中根据功能将节点分为3种角色.

- Leader: 在一个集群中, leader负责处理所有的客户端读写请求, 并执行日志复制.
- Follower: 不主动发出任何信息, 只被动接受来自leader或candidate的信息并回复.
- Candidate: follower若想成为leader, 必须首先变为candidate, candidate发起选举.

每个节点在同一时刻只能担任一种角色, 这点与Basic Paxos不同, 节点角色在算法运行过程中可以切换. 3种角色之间的转换见图4.

每个follower都有一个随机定时器, 当follower收到来自leader或者candidate的有效消息时, 会重置定时器. 如果定时器超时, follower会变为candidate, 自增任期, 重置定时器, 并向集群中所有节点请求投票, 然后根据投票结果决定将要转变的角色.

选主成功后, 在没有日志信息的情况下, 新的leader会向所有的follower周期性地发送消息以重置follower的定时器, 防止出现选举. 只要leader没有发生故障宕机或者没有发现具有更高

任期编号的节点, 会一直工作下去.

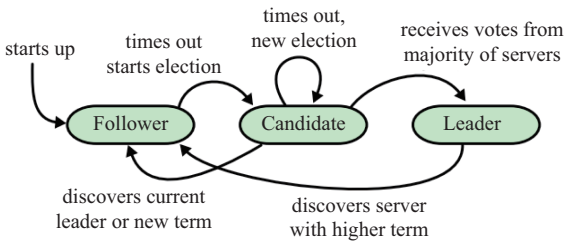

图 4　Raft 角色转换

Fig. 4　The transition of roles in Raft

1.3.3　日志复制

当客户端发出读写请求时, leader 生成相应的日志, 然后向集群中 follower 发送携带该日志的消息, follower 收到后复制该日志. 当超过半数 follower 复制了该日志后, leader 便将日志应用到自己的状态机上并将执行结果返回给客户端, 认为日志已提交. Raft 保证所有已提交的日志最终都会被所有可行复制状态机执行.

在正常的运作中, leader 和 follower 的日志始终保持强一致. 但在分布式环境下一旦出现节点故障、网络包延迟或丢失等情况, 日志的强一致性难以保持. leader 和 follower 的日志不一致的现象及可能的导致该现象的部分原因分析参见表2.

表 2　节点日志不一致现象及原因分析

Tab. 2　Occurances and reasons for inconsistencies of logs between nodes

日志不一致现象	部分原因
follower 没有 leader 的某些日志.	旧 leader 还未将全部日志复制到该 follower 就宕机, 随后迅速重启又当选为新 leader; 或者 follower 宕机后重启, leader 正常运行.
leader 没有 follower 的某些日志.	leader 在前几个任期一直处于宕机状态, 在当前任期重启后当选为 leader, 而 follower 一直正常运行.
follower没有 leader 的某些日志, leader 又没有 follower 的某些日志.	leader 在还没有将自己的全部日志复制到该 follower 就宕机, 随后一直处于宕机状态, 经过几个任期后重启并当选为新的 leader.

Raft 利用 leader 与 follower 的日志匹配来检测日志不一致现象. 一旦出现日志不一致, Raft 会找到两条日志序列最后相同的位置, 然后将该位置后的 leader 日志序列覆盖到 follower 日志序列上.

1.3.4　Raft 问题分析

在 Raft 中, 如果出现网络分区, 很有可能导致“双主问题”和“频繁选举”两个问题[17].

双主问题是指集群中同时存在两个或更多主节点. Raft 算法虽然不能避免双主问题, 但是也不会受到该问题的影响, 当网络分区恢复后, Raft 能自动修复双主问题.

频繁选举是指集群中不断有节点发起选举. 频繁选举可能导致主节点切换, 对系统性能会造成影响. 当网络分区恢复后, Raft 能自动修复频繁选举问题.

1.4　分布式一致性算法分析

本文结合日志同步应用将上述 3 种分布式一致性算法进行对比分析. 具体参见表 3.

2 分布式一致性协议在分布式数据库中的应用

分布式一致性协议在分布式数据库中得到了广泛应用, 不同的数据库在实现一致性过程之中, 会结合自身架构、数据类型、故障处理等具体需求对理论中的分布式一致性协议做出修改以满足应用. 本节将主要介绍Cassandra[18]、PaxosStore[19]、XPaxos[20]、Chubby[21]、Meagstore[22]和Spanner[23]6种分布式数据库中分布式一致性协议的实现, 并作出对比分析.

2.1 Cassandra

Apache Cassandra是一套开源分布式NoSQL数据库系统, 最初由Facebook开发, 用于储存收件箱等简单格式数据. 这是一个开源的、分布式、无中心、支持水平扩展、高可用的KEY-VALUE类型的NOSQL数据库.

表 3 分布式一致性算法对比分析

Tab. 3 Comparsion and analysis of distributed consistency algorithms

	Quorum	Basic Paxos	Raft
主要思想	多数派思想	多数派思想	多数派思想
选主阶段	可以有多个leader	可以有多个leader, leader不需要拥有全部已提交的日志.	强调leader的地位(唯一性), leader拥有全部已提交的日志.
日志复制阶段		允许日志空洞	没有日志空洞, 保持日志的连续性.
优点	可根据实际需求配置不同的NWR参数实现不同级别的一致性.	可实现强一致性, 能容忍少于半数节点故障, 可用性较高.	可实现强一致性, 逻辑简单, 可用性较高, 工程实现难度较低.
缺点	不能保证多副本之间更新操作执行顺序的一致性.	多个proposer发起议案可能导致复制情况较多, 工程实现难度较大.	受网络分区影响可能出现“频繁选举”和“双主”问题.
适用场景	对一致性要求不高的场景.	对一致性要求较高, 多节点间网络通信良好的场景.	无网络分区的场景.

2.1.1 Cassandra架构

Cassandra采用无中心的P2P架构, 集群中所有节点都是平等的. 所有节点构成一个环, 节点间通过点对点通讯协议gossip每秒交换一次数据, 每个节点都拥有其他节点的信息, 如位置、状态等. 客户端可以连接集群中任一节点, 和客户端建立连接的节点称为协作者, 负责决定将本次请求发送到实际拥有请求所需数据的节点.

Cassandra通过一致性的有序哈希算法, 动态地在节点之间分割数据以实现规模扩容. 在系统中, 每个节点都会被随机分配一个值用来标定其在环中的位置.

2.1.2 Cassandra中的一致性

Cassandra采用最终一致性模型, 可以让用户指定每个读/写操作的一致性级别. 写操作一致性决定在向客户端回复写操作成功前, 必须成功写入几个节点; 读操作一致性决定在将结果返回到客户端前, 必须有几个节点返回结果. Cassandra目前支持以下几种一致性级别.

• ZERO: 只对写操作有意义. 协作者节点把该修改发送给所有的备份节点, 但是不会等待任何一个节点回复确认, 因此不能保证任何的一致性.

• ONE: 对于写操作, 协作者节点保证已经写到一个节点中; 对于读操作, 执行节点在获得一个节点上的数据后立即返回结果. 提供弱一致性.

• QUORUM: 令数据的备份节点数目为n. 对于写操作, 保证至少写到$n/2+1$个节点上; 对于读操作, 向$n/2+1$个节点查询, 返回时间戳最新的数据. 提供最终一致性.

• ALL: 对于写操作, 协作者节点保证所有节点写成功后才向客户端返回成功确认消息, 任何一个节点没有成功, 认为写失败; 对于读操作, 会向所有节点查询, 返回时间戳最新的数据, 如果某个节点没有返回数据, 则认为读失败. 提供强一致性.

Cassandra默认的读写模式W(QUORUM)/R(QUORUM), 只要保证$W+R>N$(W是写节点数目, R是读节点数目, N为副本数), 即写的节点和读的节点重叠, 则是强一致性. 如果$W+R<=N$, 则是弱一致性. 如果在读和写操作的时候都选择QUORUM级别, 那么就能保证每次读操作都能得到最新的更改. Cassandra通过以下技术来维护数据的最终一致性.

• 逆熵: 这是一种备份之间的同步机制. 节点之间定期互相检查数据对象的一致性.

• 读修复: 客户端读取某个数据对象的时候, 触发对节点上数据对象的一致性检测, 如果发现有不一致, 则进行一致性修复.

■ 如果读一致性级别为ONE, 会立即返回离客户端最近的一份数据副本. 然后会在后台执行读修复. 这意味着第一次读取到的数据可能不是最新的数据;

■ 如果读一致性级别为QUORUM, 则会在读取超过半数的一致性的副本后返回一份副本给客户端, 剩余节点的一致性检查和修复则在后台执行;

■ 如果读一致性级别为ALL, 则只有读修复完成后才能返回一致性的一份数据副本给客户端. 可见该机制有利于减少最终一致的时间窗口.

Cassandra可以实现跨数据中心的数据库高效访问, 其方法是把延时、吞吐与一致性的权衡交给用户抉择. Cassandra提供了如下两种访问级别.

• Local_Quorum: 本地数据中心内超过半数的副本执行成功, 才返回用户操作成功.

• Each_Quorum: 每个数据中心内超过半数的副本执行成功, 才返回用户操作成功.

其中Each_Quorum需要跨数据中心访问, 延时较大且带宽、吞吐量也会降低, 但可以保证全局数据的强一致性; 而Local_Quorum则无法保证全局数据强一致性但提供了较高可用性.

Cassandra通过节点间的交换信息确定节点是否可用, 避免客户端请求被发送到一个不可用的节点. 节点中断并不意味着这个节点永久不可用, 因此不会永久地从网络环中去除, 其他节点会定期探测该节点是否恢复正常. 若想永久去除节点, 需要人工手动删除. 当节点中断时, 所错过的写入数据由其他节点暂为保存, 待该节点恢复后, 从其他节点恢复数据. 若节点中断时间超过默认时间, 这部分数据就会被丢弃. 此后节点恢复需要使用节点修复工具手动在所有节点上执行数据修复, 以保证数据最终一致性.

2.1.3　总结

Cassandra提供了高性能的数据写入、读取功能, 采用去中心化的环形结构, 没有主从节点之分, 各节点地位平等, 互相保存其他节点的状态信息. Cassandra在处理读写请求时首先将请求发给协作者节点, 由协作者节点根据一致性级别和其他节点的信息决定将请求发送给哪些节点. Cassandra采用最终一致性模型, 用户可以制定数据的读写一致性以满足用户的不同需求, 同时采用逆熵、读修复等方法维护数据的最终一致性.

2.2　PaxosStore

PaxosStore是微信研发的用于支持微信复杂业务的高可用存储系统. 微信的业务由其后端许多不同功能的组件所支持, 为避免存储系统的多样性对系统维护难度和可扩展性的影响, 微信使用PaxosStore作为一个通用的存储系统来支持微信的复杂业务.

2.2.1　PaxosStore架构

PaxosStore的整体架构见图5. PaxosStore可分为Programming Model、Consensus Layer、Storage Layer 3层. 其中Programming Model提供与外部应用程序相交互的多种数据结构; Storage Layer由多个基于不同存储模型实现的存储引擎组成; 而Consensus Layer实现

了基于 Multi Paxos 的分布式一致性协议. 接下来主要介绍 Consensus Layer 中的一致性协议.

2.2.2 Consensus Layer 中的 Paxos

PaxosStore 考虑到 Multi Paxos 中复杂的节点状态以及节点间多种类型消息会增加系统开发的难度, 降低系统性能, 因此 PaxosStore 采用一种半对称消息传递的方式实现节点间通信, 各节点上会存储数据在所有节点上副本的相关信息. 为了便于描述, 现定义几个符号.

- P=(n, v), P 代表议案, n 代表议案编号, v 代表议案中数据值.
- N_X 代表 id 为 X 的节点;
- r, 代表数据. r_X 代表数据 r 在节点 N_X 上的副本.
- $S_X^Y = (m, P)$, 表示节点 N_Y 上存储的关于 r_X 的相关信息, 其中 m 表示 r_X 可批准的议案最小编号, P 表示 r_X 已经批准的最新议案.
- $S_X^X = (m, P)$, 表示节点 N_X 上存储的关于 r_X 的相关信息.
- S^Y, 表示 N_Y 上关于数据 r 的副本的全部信息, 即全部的 S_X^Y, X 的范围是所有节点.
- $M_{X\to Y}$=(S_X^X, S_Y^X), 表示 N_X 发给 N_Y 的消息包括 S_X^X 和 S_Y^X.

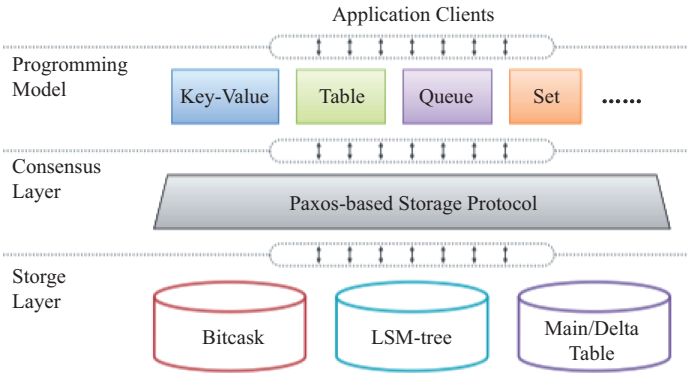

图 5 PaxosStore 架构

Fig. 5 The PaxosStore framework

一个数据副本的 S_X^X 和 S_X^Y 可能不同, 需要在算法运行时不断实现同步. 在 PaxosStore 中, 数据副本通过节点间通信不断互相交换 S_X^X 和 S_X^Y 来实现节点间数据副本的一致. 不同于Basic Paxos 中多种类型的 message (如 PrepareRequest, AcceptRequest), PaxosStore 在整个 Paxos 实现中节点间通信只使用一种统一类型的消息, 即 $M_{X\to Y}$. 当节点 N_Y 收到节点 N_X 发来的消息 $M_{X\to Y}$ 后, 首先会采用一个统一的消息处理函数来更新自己的 S_X^Y 和 S_Y^Y, 然后判断 S_Y^Y 是否已经改变, 如果改变则将新的 S_Y^Y 写入到 r_Y 的 PaxosLog 中, 并判断 S^Y 中是否存在已经达到多数派的议案 P, 如果存在则将该议案 P 提交. 最后 N_Y 会返回给 N_X 相同类型的消息 $M_{Y\to X}$.

PaxosStore 将采用三副本模式, 其中两个副本提供读写功能, 另一个副本只提供复制功能, 不对客户端提供读服务. 同时 PaxosStore 了实现多主多写, 当主节点故障或网络异常时客户端自动选择可用机器进行跳转, 不会有主备切换的时间差延迟.

2.2.3 总结

综上所述, 相对于 lamport 提出的 Basic Paxos 理论, PaxosStore 对 Paxos 的实现放弃了复杂的节点状态和节点间多种类型消息, PaxosStore 认为过多的节点状态和复杂的状态转变不仅在工程上难以实现而且会导致性能的下降. 因此 PaxosStore 采用一种半对称消息传递的方式, 在整个 Paxos 实现中只使用一种统一格式消息和统一的消息处理函数实现, 省去了状态机的维护, 简化了过程. 同时 PaxosStore 实现多主多写, 避免单点故障对系统性能的影响.

2.3 XPaxos

XPaxos是阿里巴巴数据库团队面向高性能、全球部署以及阿里业务特征等需求, 实现的一个高性能分布式强一致的Paxos库.

2.3.1 XPaxos架构

XPaxos的整体架构可分为网络层、服务层、算法模块和日志模块4个部分. 其中网络层负责网络传输; 服务层为Paxos提供事件驱动等核心运行功能; 算法模块是一致性协议的主要部分; 日志模块则是从算法模块中独立出来, 可以结合已有的日志系统, 对接日志模块接口, 以获取更高的性能和更低的成本. 本节主要介绍算法模块中的Paxos.

2.3.2 算法模块中的Paxos

XPaxos算法模块中的Paxos在标准基于leader的Multi Paxos的基础上, 支持在线添加/删除多种角色的节点, 支持在线快速将主节点转移到其他节点 (有主选举).

由于阿里巴巴集团很多应用在多地有部署且只有一个中心, 因此应用要求在没有发生城市级容灾的情况下, 仅在中心写入数据库. 当发生城市级容灾时, 可以迅速将写入点切换到非中心城市, 保证完全不丢失数据. 但标准基于leader的Multi Paxos中, 多数派强同步以后即可完成提交, 而多数派是非特定的, 并不能保证某个或某些节点一定能得到完整的数据. 在实际实现中, 往往是地理位置靠近中心的节点会拥有强一致的数据, 而地理位置较远的节点, 往往会一直处于非强一致状态. 在发送城市级容灾时, 地理位置较远的节点永远无法成为主节点.

XPaxos在协议中实现了"策略化多数派"和"权重化选主"两种方法来解决上述问题. 所谓策略化多数派, 即用户可以通过动态配置, 指定某个或某些节点必须保有强一致的数据, 以保证在出现城市级容灾时, 可以激活为主节点. 而权重化选主, 指用户可以指定各个节点的选主权重, 只有在高权重的节点全部不可用的时候, 才会激活低权重的节点.

在标准基于leader的Multi Paxos中, 一般每个节点都包含Proposer/Accepter/Learner3种角色功能, 但是在有些情况下并不需要所有节点都拥有3种角色的功能. 例如在集群三节点时, 可以令其中一个节点不保留数据, 只存储日志, 并可在同步过程中作为多数派计算, 此时去该除节点的proposer角色功能, 保留accepter和learner角色功能后, 依旧可以使另外两个节点上的数据保持一致性. XPaxos通过对节点角色的定制化组合, 可以实现具有定制功能的节点, 以满足不同的需求, 提高系统性能, 避免代码冗余.

基于leader的Multi Paxos在处理多个实例时, 针对每个实例都会发生通信, 造成一次网络I/O, 而XPaxos将多个实例对应的信息合并成单个消息进行发送, 有效地降低了消息粒度带了的额外损耗, 提升吞吐. 同时为避免过低的消息粒度造成单次请求延迟过大, 导致并发请求次数过高, XPaxos允许在上一个消息返回结果之前, 并发的发送下一个消息到对应的节点. 因此XPaxos相较于标准的Multi Paxos, 在一定程度上降低了网络延迟, 提高了系统性能.

基于leader的Multi Paxos存在一个问题就是系统的性能受主节点影响较大. 当主节点负载较高, 系统性能会降低, 并存在单点故障问题. XPaxos保证系统在稳定运行时会感知各个节点间网络延迟, 并形成级联拓扑, 从而降低主节点负载和网络通信负载. 当有节点异常时, 各个节点会自动重组拓扑, 保证各个存活节点正常运行. 因此XPaxos相较于标准的Multi Paxos, 在性能瓶颈和故障处理方面得到提升.

2.3.3 总结

XPaxos在标准的Multi Paxos基础上实现了节点上下线, 有主选举, 同时采用"策略化多数派"和"权重化选主"来提高系统对城市级容灾的故障处理能力. 为提高系统性能、避免代码冗余, XPaxos根据不同需求实现具有定制功能的节点, 并针对高延迟网络做出协议优化. 同时XPaxos通过级联拓扑来降低主节点负载, 降低单点瓶颈和单点故障对系统性能的影响.

2.4 Chubby

Chubby 是一个面向松耦合分布式系统的锁服务, 提供粗粒度的分布式锁服务. GFS 和 Big Table 等大型系统都用其来解决分布式协作、元数据存储和 Master 选举等一系列与分布式锁服务相关的问题. Chubby 的底层一致性实现就是以 Multi Paxos 算法为基础.

2.4.1 Chubby 架构

Chubby 服务端的基本架构见图 6. 可分为容错日志层、容错数据库和服务层. 其中容错数据库负责存储数据. 服务层对外提供分布式锁服务和小文件存储服务. 容错日志层通过 Multi Paxos 算法来保证集群所有机器上的日志完全一致, 同时具备较好的容错性. 本节主要介绍容错日志层中的 Paxos.

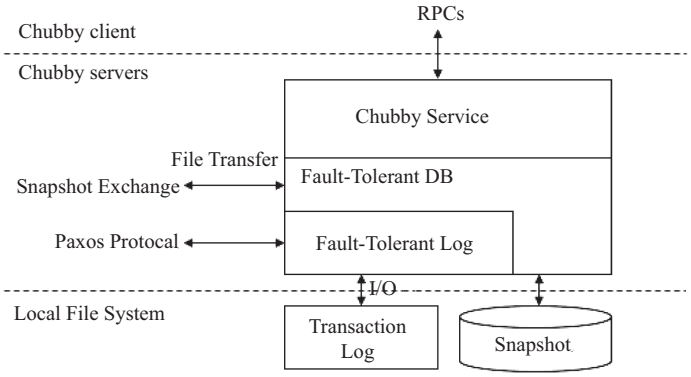

图 6 Chubby 服务端基本架构

Fig. 6 The Chubby framework

2.4.2 容错日志层中的 Paxos

容错日志层中的 Paxos 是采用基于 leader 的 Multi Paxos 方法实现的. Chubby 事务日志中的每一个 Value 对应 Multi Paxos 算法中的一个实例, 由于 Chubby 需要对外提供不断的服务, 因此事务日志会无限增长, 于是在整个 Chubby 运行过程中, 会存在多个Paxos 实例, 同时 Chubby 会为每个 Paxos 实例都按序分配一个全局唯一的实例编号, 并将其按顺序写入到事务日志中去. 在多 Paxos 实例的模式下, 为了提升算法执行的性能, Chubby 选取一个节点作为主节点, 以避免每个节点都提出议案而造成复杂情况.

2.4.3 总结

Chubby 采用基于 leader 的 Multi Paxos 算法实现了日志的一致性. 同时 Chubby 会保证在 leader 重启或出现故障而进行切换的时候, 允许出现短暂的多个 leader 共存而不影响副本之间的一致性.

2.5 Megastore

Megastore 是一个为满足当今交互式在线服务需求而开发的存储系统, 该系统成功地将关系型数据库和 NoSQL 的特点与优势进行了融合, 被广泛应用于 Google 产品中.

2.5.1 Megastore 架构

Megastore 的基本架构见图 7, Megastore 的部署需要通过一个客户端函数库和若干的服务器. 应用程序连接到这个客户端函数库, 这个函数库执行基于 Multi Paxos 的一致性算法. 最底层的数据是存储在 Bigtable 中的. 本节主要介绍客户端函数库中的 Paxos.

2.5.2 Megastore 函数库中的 Paxos

为了实现一个同步的、容错的、适合远距离传输的复制机制, Megastore 在 Multi Paxos 的基础上做出一定改进以满足远距离分布式一致性的要求. 同 XPaxos 一样, Megastore 也通过对

节点角色的定制化组合, 实现具有定制功能的节点. Megastore 节点类型可分为全功能节点, 见证者节点和只读节点, 其中全功能节点依旧存储完整的日志和数据;见证者节点在 Paxos 算法执行过程中产生一个决议时可参与投票, 但只存储全部日志而不存储数据; 只读节点在产生一个决议时不参与投票, 不存储日志但存储数据的所有快照, 其作用只是读取到最近过去某一个时间点的一致性数据.

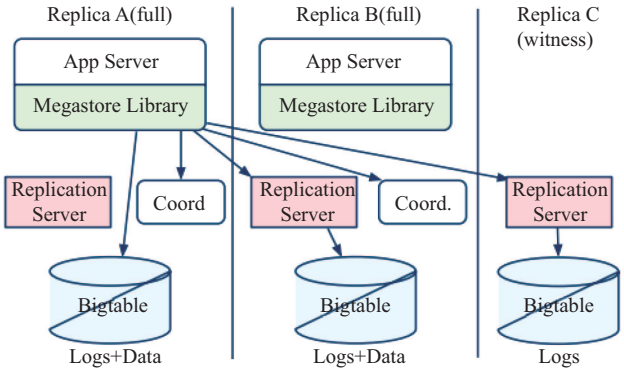

图 7　Megastore 基本架构

Fig. 7　The Meagstore framework

标准 Multi Paxos 使用唯一的主节点, 即 leader 来处理所有的读写服务, 因此主节点上的数据总是最新的, 主节点可以不需要任何网络通信就可以提供读服务. 但太多的读写请求会加大主节点的负载, 影响性能. Megastore 认为集群中其他与主节点保持强一致的节点同样有最新的数据, 可以提供读写服务, 在标准 Multi Paxos 中这些节点的资源被浪费.

为了充分利用这些节点上的资源, Megastore 要求集群所有节点都可以提供本地读的功能, 而不再只由主节点提供读服务. 为此 Megastore 使用了一个协调者服务进程来跟踪已经具有最新数据的节点副本, 协调者的状态要由应用层在写操作时负责同步更新. 发起当前读时, 节点首先通过协调者服务进程查看本地副本数据是否为最新, 若是最新数据则直接读取本地数据, 即本地读; 若不是最新数据节点会使用 Paxos 发起一个多数派读操作, 找到其他节点中的最新数据, 并更新自己的本地数据. 本地读可以在一定程度上避免较高的网络开销, 降低延迟, 实现快读, 同时节省系统资源, 减少读故障, 降低开发难度.

为了达到快速的单次交互的写操作, Megastore 调整了 Multi Paxos 的主节点提交策略, Megastore 不再使用专门的 leader 提供写服务, 而是针对每个写操作指定一个特定的 proposer 执行, 每次写操作的特定 proposer 是与上一次写操作的值一起被选出来, 因此集群中任何节点都可能发起写操作, 只要当选为特定 proposer. 由于大部分应用总是重复提交来自同一区域的写入, Megastore 选取该特定 proposer 的策略是选择距离写入最近的节点.

2.5.3　总结

Meagstore 在 Multi Paxos 的基础上做出一定改进来保证数据副本的读写一致性, 如对节点角色功能进行定制化组合, 以提高系统性能、避免代码冗余、减少数据存储空间; 同时 Meagstore 为避免单点瓶颈和单点故障对系统的影响, 放弃了 Multi Paxos 中主节点处理所有读写操作的方法, 在读操作上, Meagstore 利用协调者进程实现本地读, 以减少网络通信, 提高读操作性能; 在写操作上, Meagstore 针对每个写操作指定一个特定的 proposer 来执行. 各节点没有主从节点之分, 均有可能发起读写操作.

2.6　Spanner

Spanner 是谷歌公司研发的、可扩展的、多版本、全球分布式、同步复制数据库, 是第一

个把数据分布在全球范围内的系统, 并且支持外部一致性的分布式事务. 作为一个可扩展的、全球分布式的数据库, Spanner 把数据分片存储在许多 Paxos 状态机上, 这些机器位于遍布全球的数据中心内, Spanner 的主要工作就是管理跨越多个数据中心的数据副本.

2.6.1 Spanner 架构

Spanner 的架构见图 8. 一个 zone 包括一个zonemaster, 和一百至几千个 spanserver. Zonemaster 把数据分配给 spanserver, spanserver 把数据提供给客户端. 客户端使用每个 zone 上面的 location proxy 来定位可以为自己提供数据的 spanserver. Universe master 和 placement driver, 当前都只有一个. Universe master 主要是一个控制台, 其显示了关于 zone 的各种状态信息, 可以用于相互之间的调试. Placement driver 会周期性地与 spanserver 进行交互, 来发现那些需要被转移的数据, 或者是为了满足新的副本约束条件, 或者是为了进行负载均衡. 本节主要介绍 spanserver 中的 Paxos.

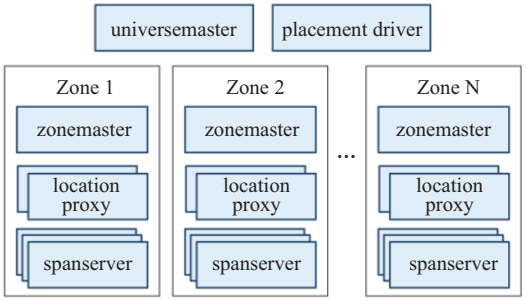

图 8 Spanner 基本架构

Fig. 8 The Spanner framework

2.6.2 spanserver 中的 Paxos

在数据的副本配置方面, Spanner 可以在一个很细的粒度上进行动态控制, 如基于 Spanner 的应用可以规定哪些数据中心包含哪些数据、不同数据副本间距离等, 从而平衡不同数据中心内资源的使用. 在底部, 每个 spanserver 负责管理 100~1 000 个称为 tablet 的数据结构的实例. 为了支持复制, spanner 会在每个 tablet 上实现一个单个的 Paxos 状态机, 每个状态机都会在相应的 tablet 上保存自己的元数据和日志.

Spanserver 中的 Paxos 同样选择一个 leader 负责提供写服务. 且支持长寿命的 leader, 为每个 leader 引入基于时间的租约. 时间通常在 0~10 s 之间. 在当前的 Spanner 实现中. 会对每个 Paxos 写操作进行两次记录: 一次是写入到 tablet 日志中. 一次是写入到 Paxos 日志中. Paxos 会把写操作按顺序执行. Spanner 中读操作可以从底层的任何副本的 tablet 中访问, 只要这个副本足够新.

2.6.3 总结

Spanner 采用基于 leader 的 Multi Paxos 方法. 且 Leader 支持长寿命, 负责处理写操作, 其他节点则可以在保证自身数据最新的情况下处理读操作.

2.7 总结

本文介绍了采用分布式一致性协议的 6 种分布式数据库系统, 并对一致性协议其中的应用进行探讨. 现对采用一致性协议的这 6 种分布式数据库进行对比分析, 具体参见表 4.

综上所述,分布式数据库系统在实现一致性过程中, 考虑到自身的节点分布、网络通信、系统容灾、应用负载和实现成本等实际需求, 会对传统的分布式一致性协议作出一定的改进, 以适

应系统应用, 提升系统性能.

表 4　采用一致性协议的分布式数据库对比分析

Tab. 4　Comparsion and analysis of consistency algorithms in distributed database

	Chubby	XPaxos	Meagstore	Spanner	PaxosStore	Cassandra
设计目标	提供分布式锁服务.	高性能分布式强一致的 Paxos 库.	满足交互式在线服务需求存储系统.	多版本、全球分布式同步复制数据库.	支持混合存储的通用存储系统.	KEY-VALUE 类型的 NOSQL 数据库
读写操作	由唯一主节点处理读写操作, 如 Chubby. 由唯一主节点处理写操作, 其他节点可以处理读操作. 如 Spanner、XPaxos. 放弃唯一主节点处理读写操作的方法, 任意节点均可能处理读写操作. 的 NOSQL 数据库					
节点类型	用状态机维护节点状态, 节点为全功能节点, 如 Chubby. 用状态机维护节点状态, 节点可以是定制功能节点, 如 XPaxos、Meagstore、Spanner. 放弃使用状态机描述节点, 各节点保存数据在所有节点上副本的相关信息, 如 PaxosStore、Cassandra					
节点通信	节点间通信包括 PrepareRequest、AcceptRequest 等多种类型的消息以及多种 消息处理模块. 状态描述较复杂. 如 Chubby、XPaxos、Meagstore、Spanner. 采用统一的通信消息类型, 即 $M_{X\to Y}$=(SX X, SX Y), 统一的消息处理模块. 简化了 信息传递与信息处理的过程, 避免复杂的状态描述. 如 PaxosStore. 采用点对点通讯协议 gossip 实现节点间通信, 如 Cassandra.					

3　总结与展望

本文首先介绍了经典的分布式一致性协议 Quorum、Paxos、Raft, 并分析其特点及应用场景, 之后又介绍了工业界中 6 种分布式数据库系统 Cassandra.、Chubby、XPaxos、Meagstore、Spanner 和 PaxosStore 的基本架构, 以及在实现一致性过程中结合自身应用需求对经典分布式一致性协议作出的改进. 最后对这 6 种采用一致性协议的分布式数据库进行对比分析. 随着互联网的快速发展, 数据量呈爆发式增长, 研发性能更高、更稳定的分布式数据库系统已经成为必然趋势. 因此在目前学业界和工业界已有的分布式一致性理论和成熟的分布式数据库系统基础上, 探究更高效、更稳定的方案来解决分布式数据库系统中的一致性问题已经成为未来的研究工作之一. 本文是一篇综述文章, 希望能够为当今的分布式数据库系统工作者提供参考. 由于分布式一致性协议及其对应的产品很多, 笔者没有能力将所有的一致性协议和分布式数据库相关技术全部归纳到文章中, 希望读者在阅读时发现错误可以多加指正.

[参　考　文　献]

[1] TANENBAUM, MAARTEN VAN STEEN. Distributed Systems Principles and Paradigms [M]. 2nd ed. USA: Pearson, 2001: 1-10.

[2] BREWER E A. Towards robust distributed systems (abstract) [C]// Nineteenth ACM Symposium on Principles of Distributed Computing. New York: ACM, 2000: 7.

[3] GILBERT S, LYNCH N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services [J]. Acm Sigact News, 2002, 33(2): 51-59.

[4] TANENBAUM A S, STEEN M V. Distributed Systems: Principles and Paradigms [M]. Beijing: Tsinghua University Press, 2002.

[5] 朱涛, 郭进伟, 周欢, 等. 分布式数据库中一致性与可用性的关系 [J]. 软件学报, 2018(1): 131-149.

[6] 储佳佳, 郭进伟, 刘柏众, 等. 高可用数据库系统中的分布式一致性协议 [J]. 华东师范大学学报 (自然科学版), 2016, 2016(5): 1-9.

[7] GIFFORD D K. Weighted voting for replicated data [C]// Acm Symposium on Operating Systems Principles. New York: ACM, 1979: 150-162.

[8] ONGARO D, OUSTERHOUT J K. In search of an understandable consensus algorithm [C]//USENIX Annual Technical Conference. New York: ACM, 2014: 305-319.

[9] JUNQUEIRA F P, REED B C, SERAFINI M. Zab: High-performance broadcast for primary-backup systems [C]//International Conference on Dependable Systems & Networks. New York: IEEE, 2011: 245-256.

[10] LAMPORT L. Paxos made simple [J]. ACM Sigact News, 2001, 32(4): 18-25.

[11] CHANDRA T D, GRIESEMER R, REDSTONE J. Paxos made live: an engineering perspective [C]//Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing. New York: ACM, 2007: 398-407.

[12] LAMPORT L B, MASSA M T. Cheap paxos: U.S. Patent 7, 249, 280 [P]. 2007-07-24.

[13] LAMPORT L. Fast paxos [J]. Distributed Computing, 2006, 19(2): 79-103.

[14] LAMPORT L, MALKHI D, ZHOU L. Vertical paxos and primary-backup replication [C]//Proceedings of the 28th ACM symposium on Principles of distributed computing. New York: ACM, 2009: 312-313.

[15] LAMPORT L, SHOSTAK R, PEASE M. The Byzantine generals problem [J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1982, 4(3): 382-401.

[16] 张晨东, 郭进伟, 刘柏众, 等. 基于 Raft 一致性协议的高可用性实现 [J]. 华东师范大学学报 (自然科学版), 2015(5): 172-184.

[17] 庞天泽. 可扩展数据管理系统中的高可用实现 [D]. 上海: 华东师范大学, 2016.

[18] FACEBOOK. The Apache Software foundation: Apache Cassandra Documentation v4.0 [EB/OL]. (2016-09-01)[2018-04-10] http://cassandra.apache.org/.

[19] ZHENG J, LIN Q, XU J, et al. PaxosStore: High-availability storage made practical in WeChat [J]. Proceedings of the VLDB Endowment, 2017, 10(12): 1730-1741.

[20] 江疑. X-Paxos: 阿里巴巴的高性能分布式强一致 Paxos 独立基础库 [EB/OL]. [2017-08-07]. http://developer.51cto.com/art/201708/547380.htm.

[21] BURROWS M. The Chubby lock service for loosely-coupled distributed systems [C]// USENIX Association . Proceedings of the 7th symposium on Operating systems design and implementation. New York: ACM, 2006: 335-350.

[22] BAKER J, BOND C, CORBETT J C, et al. Megastore: Providing scalable, highly available storage for interactive services [C]//Biennial Conference on Innovative Data Systems Research. USA: Online Proceedings, 2011(11): 223-234.

[23] CORBETT J C, DEAN J, EPSTEIN M, et al. Spanner: Google's globally distributed database [J]. ACM Transactions on Computer Systems (TOCS), 2013, 31(3): 8.

(责任编辑: 张 晶)