软件学报ISSN 1000-9825, CODEN RUXUEW E-mail: jos@iscas.ac.cn

*Journal of Software*, [doi: 10.13328/j.cnki.jos.000000] http://www.jos.org.cn

©中国科学院软件研究所版权所有. Tel: +86-10-62562563

分布式系统中的因果一致性综述[[1]](#footnote-1)\*

江雪1, 魏恒峰1,2, 黄宇1

1(南京大学 计算机科学与技术系, 江苏 南京 210023)

2(南京大学 软件学院, 江苏 南京 210023)

通讯作者: 魏恒峰, E-mail: hfwei@nju.edu.cn

摘 要:

关键词: 因果一致性;协议;理论;检测

中图法分类号: TP311

中文引用格式: 江雪,魏恒峰,黄宇.分布式系统中的因果一致性综述.软件学报,2023,32(7). http://www.jos.org.cn/1000-9825/0000.htm

英文引用格式: Jiang X, Wei HF, Huang Y. Survey on causal consistency. Ruan Jian Xue Bao/Journal of Software, 2023 (in Chinese).http://www.jos.org.cn/1000-9825/0000.htm

Survey on Causal Consistency in Distributed Systems

JIANG Xue1, WEI Heng-Feng1,2, HUANG Yu1

1(Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

2(School of Software, Nanjing University, Nanjing 210023, China)

**Abstract**:

**Key words**: causal consistency; protocol; theory; testing

# 引言

使用副本的分布式系统

CAP, PACELC, 弱一致性

因果一致性,非事务和事务

规约,理论,协议,检测几个部分的重要性,每个部分大致包括哪些内容

# 规约框架

在本章中我们将介绍针对弱一致复制数据类型的通用规约框架.该框架是通过扩展 Burckhardt 等人[1,2]提出的针对最终一致的复制数据类型的规约框架获得的.

## 关系和序

一个二元关系

# 非事务因果一致性

因果一致性是在分布式系统中被广泛使用的一种弱一致性模型[3,4].因果一致性的规约中最关键的概念是事件之间的发生顺序(happens-before order/hb)[2,5].直观地说,因果一致性确保如果一个事件发生在事件之前,那么所有会话(session)必须在之前观察到.然而,同时并发事件可能被不同的会话以不同的顺序观察到.

相关文献中定义了几种因果一致性的变体,他们之间有细微的差别[2,6,7].在本节中,我们考虑六种变体,称之为Weak Causal Consistency(WCC)、Weak Causal Convergence(WCCv)、Causal Memory(CM)、Causal Memory Convergence(CMv)、Strong Causal Consistency(SCC)和Strong Causal Convergence(SCCv),如表2所定义.请注意,这些变体在相关工作中可能有不同的名称,如表2所总结的.

就可见性()而言,所有因果一致性变体需要来捕捉事件之间发生的先后关系.在仲裁()方面,它们都要求,用以强制要求当解释事件在其操作上下文中的返回值时,可见的所有事件()执行hb顺序.然而,它们可能在以下两个方面有所不同:

* 用于指定其返回值必须被考虑的可见事件子集的函数有多大?请注意,在因果一致性模型中,对于任何事件(因为).因此,候选.
* 仲裁关系对于解决冲突有多强?给定,根据是否是全序,我们区分两种情况.

表 1 因果一致性变体在((框架中的规约

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Consistency Models | Alternative Names |  |  |  |
| WCC ([6]) | CC ([7]) |  |  |  |
| CM ([3,7]) | CC ([6]) |  |
| SCC |  |  |
| WCCv | CCv ([6,7])  CAUSALCONSISTENCY ([2])  CAUSALITY ([8]) |  |  |
| CMv |  |  |
| SCCv |  |  |

## 规约

### Weak Causal Consistency

在[6]中提出的Weak Causal Consistency(WCC)是我们考虑的最弱的因果一致性变体.非正式地讲,一个抽象执行满足WCC,当且仅当每个事件的返回值)能够被其可见事件集的某种序列化所解释(忽略所有可见事件的返回值).更具体地说,在仲裁方面,WCC对收敛性不做额外的限制,因此并发的事件可能被不同的会话以不同的顺序观察到.就而言,WCC允许每个事件忽略其可见事件的所有返回值.

其可见事件的所有返回值.

1. (Weak Causal Consistency)

### Causal Memory

Causal Memory(CM)最初是由Ahamad等人[3]提出的针对读/写寄存器(read/write register)的定义.最近,Perrin等人[6]将其扩展到任意的复制数据类型.CM比WCC更强,因为在证明每个事件的返回值)的合理性时, CM不仅考虑到了WCC中对可见的事件集的操作调用,还考虑到同一会话中之前的事件集合中事件的返回值.换句话说,与WCC相比,CM要求每个会话与之前提供的返回值相一致[6].

1. (Causal Memory)

### Strong Causal Consistency

我们通过进一步要求每个会话和其他会话提供的返回值保持一致,来加强CM使其成为 SCC(Strong Causal Consistency).形式上,我们有.

1. (Strong Causal Consistency)

### Weak Causal Convergence

WCCv(Weak Causal Convergence)[2,6]是WCC的收敛版本.它通过对执行中的所有事件施加一个全序加强WCC,这为所有会话提供了一个解决并发事件引起的冲突的统一方法.因此,每个事件的返回值在对可见的事件的集合上进行验证时(忽略他们所有的返回值),这些可见事件要按照共同的全序排序.

1. (Weak Causal Convergence)

### Causal Memory Convergence(CMv)

CMv(Causal Memory Convergence)是CM对应的收敛版本.它要求是一个全序.

1. (Causal Memory Convergence)

### Strong Causal Convergence(SCCv)

SCCv(Strong Causal Convergence)是SCC对应的收敛版本.它要求是一个全序.

1. (Strong Causal Convergence)

## 理论

现有的系统采用不同的技术来实现因果一致性,但它们都有相同的关键机制:只有当更新的所有因果依赖都已知在数据中心复制时,更新才会在该数据中心可见.因果一致性的实现中最核心的内容就是追踪事件之间的因果依赖关系(Causal Dependencies).无论利用何种机制来进行追踪,都离不开对事件进行标记(时间戳),通常可以使用以下几种方式.

1. Single Timestamp
2. Lamport Clock[5]

Lamport Clock由计算机科学家Leslie Lamport于1978年提出.Lamport时间戳是一种逻辑时钟,它可以被用于确定事件之间的顺序关系.

Lamport Clock的实现方式是每个进程都维护一个本地的时钟计数器,初始值为0.当进程执行一个事件时,它会将本地时钟计数器的值作为该事件的时间戳,并将时间戳随事件一起发送到其他节点.当接收到其他节点发送的事件时,节点会将本地时钟计数器的值与接收到的事件的时间戳进行比较,然后将本地时钟计数器的值设置为较大的那个值加1,并将更新后的值作为新的时间戳.

使用Lamport Clock,我们可以将事件排序,并确定它们之间的顺序关系.如果事件A的时间戳早于事件B的时间戳,则可以得出结论,事件A在事件B之前发生.但是,Lamport Clock不能用于确定事件的实际发生时间,因为事件的时间戳只是一个逻辑标记,不具有实际的时间单位. 当两个事件的时间戳相等时，无法判断它们之间的因果关系，也无法判断它们是否并发。

Lamport时间戳是分布式系统中一种广泛使用的技术,可以用于并发控制、同步、消息传递等方面.它的优点是实现简单,可扩展性好,缺点是无法反映实际时间,也不能用于测量两个事件之间的时间差.

1. Physical scalar clocks[9]

物理标量时钟(Physical Scalar Clock)是一种基于物理时钟的时钟算法,它基于物理时钟的增长来衡量事件的发生顺序,并提供了全局事件的绝对排序.

物理标量时钟的实现方式是每个进程(节点)都维护一个本地的物理时钟,它可以通过系统

的物理时钟来获取.当进程执行一个事件时,它将物理时钟的值作为该事件的时间戳,并将时间戳随事件一起发送到其他节点.当接收到其他节点发送的事件时,节点会将本地物理时钟的值与

接收到的事件的时间戳进行比较,然后将本地物理时钟的值设置为较大的那个值加1,并将更新后的值作为新的时间戳.

在事件的排序中,根据物理时钟的值可以直接判断事件的发生顺序.较早发生的事件具有较小的物理时钟,较晚发生的事件具有较大的物理时钟值.

物理标量时钟提供了全局事件的绝对排序,因为它是基于物理时钟的增长来确定事件的发生顺序.它能够准确地处理节点之间的时钟偏差和传输延迟等问题,因为它直接使用物理时钟的值.然而,物理标量时钟的一个限制是,如果系统中存在物理时钟不同步或不可靠的情况,那么事件的排序也可能出现错误.

1. Hybrid Logical Physical Clock[10]

Hybrid Logical Physical Clock (HLC)是由Marcin Lis和Roman Dębski在2005年提出的.HLC结合了逻辑时钟和物理时钟的优势,以提供更准确和可比较的时间戳.

HLC的基本思想是将逻辑时钟和物理时钟结合起来,得出一个更准确的时间戳.具体地说,每个进程维护两个计数器：一个物理时钟计数器和一个逻辑时钟计数器.物理时钟计数器递增的速率由物理时钟决定,而逻辑时钟计数器递增的速率由系统时钟决定,通常比物理时钟快得多.

当一个事件发生时,HLC首先使用物理时钟计数器来计算事件的物理时间戳.然后,HLC使用逻辑时钟计数器来计算事件的逻辑时间戳,并将物理时间戳和逻辑时间戳组合成一个Hybrid时间戳.Hybrid时间戳由物理时间戳和逻辑时间戳的加权平均值组成,权重由一个参数控制,通常是一个介于0和1之间的小数.

我们可以通过比较两个事件的HLC时间戳来确定它们之间的因果关系.首先比较两个时间戳的物理时钟部分.较早的物理时间戳被认为在时间上更早.如果两个时间戳的物理时间戳部分相等,那么比较它们的逻辑时钟部分.较大的逻辑时钟值被认为在事件顺序上更晚.如果两个时间戳的物理时间戳和逻辑时间戳都相等,则比较它们的Hybrid时间戳.Hybrid时间戳是物理时间戳和逻辑时间戳的加权平均值.根据加权平均值的结果,较大的Hybrid时间戳被认为在事件顺序上更晚.

1. Vector Clock[11,12]

向量时钟可以用于解决Lamport时间戳无法解决的判断事件是否并发的问题.

1. Logical Vector Clock

逻辑向量时钟的基本思想是,为每个事件分配一个向量时间戳,该向量时间戳由所有进程的本地计数器值组成.每个进程维护一个本地计数器,该计数器在进程执行每个事件时递增,同时将本地计数器值和进程标识符包含在向量时间戳中.

当一个进程向另一个进程发送消息时,它将自己的逻辑向量时间戳附加到消息中,并将其发送给接收进程.接收进程接收到消息后,将自己的本地计数器递增,并使用接收到的逻辑向量时间戳更新自己的逻辑向量时间戳.如果接收到的逻辑向量时间戳中有任何进程的计数器值大于自己的计数器值,则将该进程的计数器值复制到自己的计数器中.

使用逻辑向量时钟,我们可以比较两个事件的时间戳,并确定它们之间的因果关系.如果事件A的时间戳在事件B的时间戳中的所有进程上都小于或等于事件B的时间戳,则可以得出结论,事件A在事件B之前发生.如果事件A的时间戳与事件B的时间戳中的某些进程计数器值相等,而其他进程的计数器值更大,则无法确定两个事件的因果关系,因此它们是并发事件.

1. Physical Vector Clock

和逻辑向量时钟不同,物理向量时钟该由所有进程的本地物理时钟的值组成.每个进程有一个本地物理时钟,在进程执行每个事件时,将本地当前物理时钟的值更新到向量时间戳中的对应位置.

它的实现方式和逻辑向量时钟相同,当一个进程向另一个进程发送消息时,它将自己的向量时间戳附加到消息中,并将其发送给接收进程.接收进程接收到消息后,进程要更新本地向量时钟.具体而言,将接收到的向量时钟与本地的向量时钟进行逐元素比较,对于每个元素,将取两个向量对应位置的较大值作为更新后的值.

它的比较方式和逻辑向量时钟是类似的.于逻辑向量时钟不同,物理向量时钟依赖物理时钟,所以会受到时钟同步的影响.如果进程之间的物理时钟不同步,会影响事件之间的因果关系的正确性.假设进程之间的物理时钟时同步的,相比逻辑向量时钟,物理向量时钟可以更精确地判断事件之间的发生顺序.

和标量时钟相比,向量时钟需要更多的存储空间来维护.向量的长度等于进程的数量，因此存储空间的需求随着进程数量的增加而增加。

1. Dependency List[4]

Dependency List(依赖列表)的基本思想是为每个事件维护一个列表,列表中包含该操作所因果依赖的其他操作.

当一个操作要执行之前,它会先检查依赖列表中是否存在其他操作的标识符,如果存在则需要等待这些操作执行完成.操作只有在其所依赖的操作全部执行完成后才能执行.这种方式确保了操作的执行顺序满足其所依赖的其他操作的要求.

要确定因果依赖关系，可以检查事件的依赖列表。如果事件A在事件B的依赖列表中，表示事件B在因果上依赖于事件A。

依赖列表适用于具有相对较少依赖关系的分布式系统.然而,依赖列表也存在一些限制.当依赖关系非常复杂时,列表的长度可能会变得很大,导致存储和传输开销增加.

1. Dependency Matrices[13]

Dependency Matrix(依赖矩阵)的基本思想是每个进程维护一个矩阵(假设有个进程,矩阵的大小是).对进程来说,表示进程当前最新的本地逻辑时钟;表示进程对进程的本地逻辑时钟的最新了解情况;表示进程对进程关于进程的逻辑时钟的最新知识的了解情况.矩阵表示进程对全局逻辑时钟的了解.

当进程在执行一个操作前,先将本地的增加1,将其作为该操作的时间戳.当一个进程向另一个进程发送消息时,它将自己的依赖矩阵附加到消息中,并将其发送给接收进程.当进程接收到来自进程的消息(包含矩阵)后,会进行如下更新.以及.再将更新完之后的增加1.

依赖矩阵只适合规模较小的分布式系统,其存储空间的需求随着进程数量的增加而增加.

1. Version Vector

使用Version Vector的最大好处是,因为它们为每个数据中心存储一个时间戳,所以它们允许以更细的粒度追踪更新的可见性.这种方法的缺点是,可扩展性差,因为元数据的大小与数据中心的数量呈线性增长.这就产生了存储和通信的开销,因为每个更新都是和它相应的Version Vector一起存储和复制的.可扩展性问题随着Dependency Matrices的出现进一步加剧了.相反,Single Scalar(e.g, Single Timestamp)是用于追踪依赖关系的可扩展性最好的方法.然而,它将所有的依赖关系压缩在一个单一的时间戳中,导致由最慢的连接(与最远的数据中心的连接)决定的最差可见性延迟.这大大影响了本地更新的可见性,而且往往是不必要的.

版本向量在添加分区以支持太大的数据集而无法在单台服务器上安装时不能扩展。它们仍然将所有分区视为一个逻辑复制，并要求在各分区之间有一个单一的序列化点进行复制，这限制了复制的吞吐量。

Version Vector和Dependency Matrics通常用来描述服务器执行操作的状态,即执行了多少操作(假设搭配每次增量都为1的计数器),利用他们,可以帮助服务器了解远程服务器的状态,以便在服务器之间同步操作的时候知道哪些操作已经是被所有服务器执行,可以进一步利用这些信息来丢弃一些过时的信息,缓解服务器的存储压力.

每个事件从以上列出的时间戳中选择一种作为标识后,可以利用以下的方法来实现因果一致性.这些标志会跟随操作作为系统中的元数据,以识别更新并确保正确的交付.元数据被附加在更新信息上也增加了通信的成本.

### sequencer-based, 利用日志

日志是代表系统中事件的一连串记录.日志中的每条记录对应于一个事件,通常会用一个唯一的标识符和一个时间戳表示.利用日志来实现因果一致性的方案是基于日志复制的方案,其基本思想是在系统中维护多个副本,并通过日志记录和复制来保证数据的因果一致性.

具体来说,系统中的每个副本都维护一个日志,用于记录所有的写操作.在写入数据时,需要将该操作写入本地日志,成为日志中的一条记录,该记录还包含它与其他记录的因果依赖关系的信息(例如用时间戳表示).副本还需要将该操作的在本地日志中的记录复制到其他副本的日志中.由于每个操作都包含了其与其他操作的因果依赖关系,因此在读取数据时,可以比较该数据项在不同副本上的版本信息,以保证因果一致性.具体来说,读取操作会要求读取特定版本的数据,系统会比较该版本的标识符和副本上对该版本对应数据的已经执行的操作的标识符,只有当该版本的所有因果依赖(Causal Dependencies)都已经执行完成时,才返回数据给客户端.

通过利用日志来实现因果一致性,系统可以保证在分布式环境下对数据的操作是有序的,并且不会发生竞争条件.同时,日志的复制也可以保证数据的可靠性,即使某个副本出现故障,系统也可以通过其他副本的日志来进行数据恢复.

### Sequencers[14]

### Explicit Dependency Checking[4,15]

Explicit Dependency Checking利用消息传递来实现因果一致性的方案是在消息中包含操作的因果关系信息,并在接收到消息后,根据这些信息来维护因果关系.

具体来说,对于写操作,可以在消息中包含该操作之前(限定在因果关系上)的操作的标识符,即先决条件(Precondition).在接收到写操作的消息时,系统会检查该操作的先决条件是否已经满足,只有在先决条件满足的情况下才执行该写操作.这样可以保证操作的因果关系是有序的,并且不会发生竞争条件.对于读操作,需要在消息中包含读取数据的版本信息(标识符,通常是一个向量时钟(Vector Clock)或时间戳(Timestamp),用于标识该读操作因果依赖的所有先前操作).在接收到读操作的消息时,系统会比较该版本的标识符和接收到的其他操作的标识符,只有当该版本的所有因果依赖(Causal Dependencies)都已经执行完成时,才返回数据给客户端.

Explicit Dependency Checking因为利用消息传递,所以面临一个潜在问题是消息延迟或丢失的可能性.如果一个消息被延迟或丢失,如果依赖该消息的其他消息已经被处理,就会导致违反因果一致性.为了减轻这种风险,消息传递协议通常包括检测和处理消息延迟或丢失的机制,如超时机制、重试和重复检测.消息传递协议的另一个问题是可能出现时钟同步问题.为了让Lamport时间戳和向量时钟正常工作,系统中的每个节点必须有一致的时间概念.如果时钟同步不当,就会导致不正确的因果排序,违反因果一致性.为了解决这个问题,可以使用时钟同步算法来确保所有节点的时钟同步到一定的精度内.此外,消息传递协议的性能会受到网络延迟、消息量和协议实现的复杂性等因素的影响.在某些情况下,与维护和处理时间戳有关的开销会影响系统性能和可扩展性.总之,虽然使用消息传递来保证因果一致性,如果实施和使用得当是安全的,但在分布式系统中设计和部署消息传递协议时,必须考虑潜在的问题,如消息延迟或损失、时钟同步问题和性能开销.

### Stablization

### Tree Dissemination

## 协议

系统类文章的协议的正确性证明

假设提供更新和查询两类操作

### Sequencer-based

Sequencer-based技术的依赖于集中式组件(Sequencers),负责定义更新操作的执行顺序.这种方法可以避免元数据规模爆炸的问题.但是,限制了节点处理请求的速度,因为在处理请求之前必须与先同Sequencer联系,这将节点处理请求的速度限制在Sequencer能够在每单位时间内处理的请求数量上.

#### ISIS toolkit[16]

ISIS toolkit[16]是一个分布式编程环境,它引入了基于虚拟同步(Virtually Synchronous)进程组和组通信(Group Communication)的协议.其协议基于多播源语(Multicast Primitive).因果广播原语(Causal Broadcast,CBCAST)可用于实现因果一致的地理复制数据存储(Geo-replicated Datastores).它的系统模型假设在全副本(Full Replication)下的单机复制,即不考虑进行数据分区.

假设系统里有一组进程,任意两个进程之间都可以通信. CBCAST协议使用一种逻辑时钟,即向量时钟(Vector Clock)来追踪因果关系,每个进程对应矢量时钟的一个条目.其基本思想是给每条消息贴上一个时间戳,,表明在之前进程已经发送出去的Multicast的数量.当收到消息后,要等到收到从进程发送的个消息后才会进行处理.该协议的具体流程如下:

1. 在发送前,进程令增加1,并将其作为的时间戳(即).进程的初始为所有条目为0的向量().
2. 当收到(Receive)发送的时间戳是的消息时,进程(非)延迟发送直到满足:

进程不需要延迟从自身发出的信息.延迟发送的消息被保存在一个队列中,即CBCAST延迟队列.这个队列里的消息是按照他们的时间戳排序的,两个并发的消息是按照收到他们的时间进行排序.

1. 当进程能够交付(Deliver)由进程发送的消息(时间戳是)时,修改本地的向量时钟为

#### Lazy Replication[17]

Lazy Replication方法向客户端提供了保证因果关系的更新和查询操作.Lazy Replication使用Logical Vetor Clock()时间戳来追踪因果关系,其中是副本的数量,每个是一个非负整数计数器,的初始值是每个为0.不同的时间戳的比较参照3.2.7(logical vector clock).

Lazy Replication可用于由在不同节点上运行的固定数量的多个副本组成的分布式系统,系统中的副本之间可以互相通信.不同节点上的时钟采用松散的同步机制,例如NTP.系统向客户端提供两种操作:查询和更新.

客户端维护一个Logical Vector Clock,用表示,用于追踪因果依赖关系.每当向副本节点发起请求时,都会将和操作一同发送给副本节点.

副本节点维护一个本地的日志,记录收到的从客户端和其他节点上发送过来的更新操作.每个副本节点还维护一个本地时间戳(Logical Vector clock),用于标识其日志中的记录集合来表示其已经知道的更新.副本节点的表示它已处理的更新次数,的其他部分的值表示在副本上处理的其他副本节点复制(通过Gossip协议)到的更新数量.此外,副本节点还维护一个本地时间戳用来表示最新执行的操作.下面将详细介绍Lazy Replication的处理流程.

1. 客户端:客户端将查询或更新操作和客户端维护的一起发送给副本节点.当更新操作请求收到副本节点的回复(包含一个时间戳)时,将更新为收到的.当查询操作收到副本节点的回复(包含一个值和一个时间戳)时,更新为.
2. 副本节点: 当副本节点收到一个客户端发来的查询时,将和进行比较.如果满足,则查询本地数据并将结果和一起回复给客户端,否则等到条件满足再执行操作. (如果副本没有必要的信息,它可以等待来自其他副本的复制消息,也可以向另一个副本发送请求以获取缺失的信息,可以通过比较和来确定缺失哪些副本的消息.)

当副本节点收到一个客户端发来的更新时,首先它会将增加1,将更新为,并将更新后的作为的时间戳.然后创建一条新的记录并将其添加到节点的本地日志中.如果依赖的操作都已经被节点执行了,即满足条件,则执行,否则等到满足条件满足再执行.执行完之后,更新为.最后,将回复给客户端.

当副本节点收到来自另一个副本节点的复制消息(包含和)时,将本地和合并(将中满足的记录,即节点还为知道的操作,添加到本地).更新的时间戳的每个部分为.然后将准备好执行,即中满足条件的更新操作的记录都添加到集合中.只要非空,则查找其中满足(即的所有因果依赖的操作在节点上都已经执行过了, 保证按照因果顺序执行中的记录)的记录并执行,执行完后还要更新为.

Lazy Replication不如ISIS toolkit提供的组播机制通用,但更适合于提供不同于客户端的复制服 (Replicated Service). 与组播机制相比,该技术需要的消息更少,而且消息也更小,因此它的性能更好.

#### Bayou[18]

Bayou是一个因果一致的复制数据库系统,允许客户端在与所连接的副本节点断开连接的情况下也能对数据进行读取和更新.Bayou的系统架构中有个副本节点,每个副本节点上保存数据的全副本,节点之间通过Pair-wise的连接通信.Bayou使用反熵协议[19](Anti-entropy Protocol)在副本节点之间通过日志来传播更新操作.

每个副本节点维护一个单调增加的Lamport Clock()和一个保存更新操作的日志WriteLog.另外,每个副本节点还维护两个版本向量(Version Vector),分别用于表示已经提交(Commited)和是还未执行(Tentative)的最新更新操作.本地副本维护两个视图(View),分别表示执行了所有已知操作(Full View,FV)和只执行了已提交操作(Commited View,CV).

为了保证最终所有副本节点上的更新操作之间存在一个全序,Bayou在维护同一个数据集的所有副本节点中选取一个成为主节点(Primary Node).主节点在收到每个更新操作时,使用本地的单调递增Lamport Clock为其打上一个Commit时间戳(Commit Sequence Number,CSN).主节点的日志通过反熵协议发送到其他节点.其他节点根据收到的主节点的日志,和本地的日志,将其中包含的所有操作进行排序,保证所有Commit的更新操作排在所有Tentative的更新操作之前,并且尊重因果顺序.

当副本节点(节点有Unique的ID,)收到客户端发来的新的更新操作时,会将本地的Lamport Clock增加1,将其作为的时间戳,并将添加到本地日志WriteLog的尾部. 当副本节点收到其他节点复制过来的更新操作时,如果是一个Tentative操作,根据的时间戳将其插入到的WriteLog中的Tentative部分(保证WriteLog中Commit操作排在所有Tentative操作之前,每类操作保证因果顺序),并按照新的WriteLog中的操作对本地数据副本内容(只对Full View)进行更新; 如果是一个Commited操作,根据的时间戳将其插入到的WriteLog中的Commit部分(反熵协议保证添加到已有Commit操作的后面),并按照新的WriteLog中的操作对本地数据副本内容(只对Commit View)进行更新.最后将本地的更新为. 当收到主节点复制过来的更新操作时,类似收到其他节点复制过来Commit操作的处理流程.

当副本节点收到来自客户端的读取请求时,只有是一直从Commit View查询才能保证系统的执行保证因果一致性,且应该是满足SCCv.考虑下图的例子,和两个节点分别执行了两个来自客户端的更新操作,时间戳是和,时间戳是,并且分别复制给了节点.和在节点的WriteLog中按照他们的时间戳进行排序,此时他们都是Tentative的.在主节点收到和之后,给他们附上了CSN,并且将他们发送给了,,和节点.因为的CSN比的CSN大,所以在的WriteLog中将和重新进行排序.假设节点还未收到主节点的同步消息时,即左图所示,假设此时执行一个查询的操作,如果从Full View读取,则应该读到2. 如果节点收到主节点的同步消息后,即右图所示,假设此时执行一个查询的操作,如果从Full View读取,则应该读到1.该执行不满足因果一致性.但如果分别在左右两个子图对应的情况下从Commit View读取,则左边只能读取的初始值,右边读到2,保证了因果顺序.

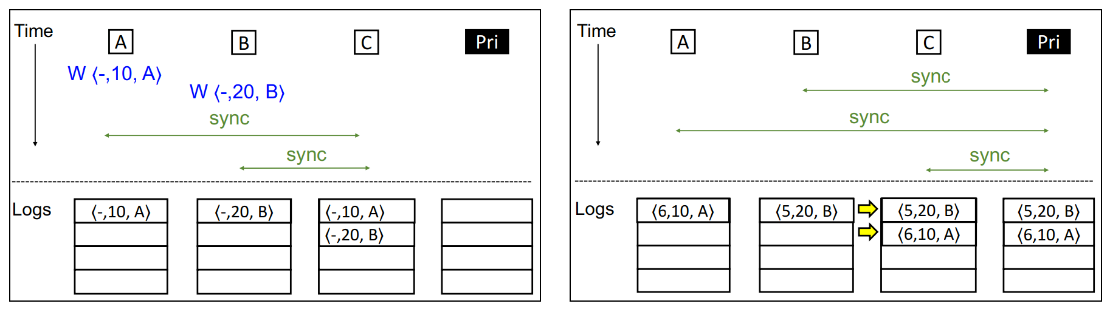


图 1

#### PRACTI[20] (dependency vector)

PRACTI的系统模型是副本节点之间可以互相通信(Topology Independence).它支持Partial Replication,即每个副本节点在本地维护所有数据的部分内容,并将他们分为多个Interest Sets(IS).

副本节点之间通过发送Causally Ordered Streams Of Invalidations和Bodies两种信息同步更新操作.其中,Invalidation消息可以是Precise(每条Invalidation包含写操作的对象的ID和逻辑时间戳)或者Imprecise的(总结一组更新操作的对象在某一时间间隔内的一系列更新操作.和是Vector Vector,每个节点都有一个条目).每条Body消息除了包含写操作的对象的ID,逻辑时间戳,还包含写操作的内容. PRACTI利用Imprecise Invalidations来提供Partial Replication.具体而言,节点仅会接收本地的中的对象的Precies Invalidation,其他对象仅会接收Imprecise Invalidation,并且也仅为中的所有对象维护Per-object State.

每个副本节点维护一个本地日志,保存收到的所有Invalidation.每个副本节点还本地维护一个根据对象ID检索的Checkpoint,保存收到的Bodies.每个副本节点一个版本向量和为每个IS 维护一个,和分别表示知道的源节点是的更新操作的最大逻辑时钟,和源节点是的时间戳小于等于的该中的对象已经都是Precise的(即已经收到了Precise Invalidation)了.

当副本节点收到客户端发送过来的更新请求时,会将本地的逻辑时钟增加1,和节点的ID构成该更新操作的逻辑时间戳,并根据该逻辑时间戳将其插入本地的日志中,其状态标记为Precise,并更新Checkpoint中的对应项,并将其状态标记为Valid,更新和该对象所对应的的为节点当前的逻辑时钟. 最后,将更新操作发送给订阅了该对象更新的所有其他节点.当副本节点收到客户端发送过来的读请求时,可能会阻塞该请求,直到要读取的对象所在的的和节点的相等(表示该是的的了),和该对象在Checkpoint中的状态是Valid这两个条件都满足才执行.

当副本节点收到节点发送过来的Causally Ordered Streams Of Invalidations,1)如果收到的是Precise Invalidations,并将该Invalidation的状态标记为Imprecise, 根据每条Invalidation中的逻辑时间将其插入到本地的日志中,并在Checkpoint中根据对象的ID寻找对应的项,如果Checkpoint中对应项的时间戳比收到的Invalidation中的时间戳小,则更新其时间戳和写操作的内容,并将该项标记为Invalid,否则对Checkpoint什么都不做.另外,还需要更新中对应于该Invalidation的源节点的项(对应的项小于Invalidation中的Lamport Clock,则更新).最后,将本地的Lamport Clock更新(原来的值和所收到的日志中的所有Invalidation的Lamport Clock的最大值).2)如果收到的是Imprecise Invalidation,则根据Imprecise Invalidation中的时间更新(两者中的较大值),将其插入到日志中,并在Checkpoint中根据其涵盖的对象找到对应项(没有该项则跳过,表示该对象不在节点的中),将其状态更新为Invalid.

当副本节点收到节点发送过来的Bodies,根据其中的每条Precise Invalidation中的对象ID和时间戳找到对应日志中的条目,将其状态修改为Precise.并在Checkpoint中根据对象ID找到对应条目,查看其时间戳是否和Invalidation中的相等,如果相等则更新其内容,并将对应项标记为Valid.最后要修改该项所在的的 中对应此Precise Invalidation的更新操作的源节点的项进行更新.

#### TACT[21]

TACT的系统模型是副本节点之间可以进行Pair-wise的通信,每个副本节点维护数据的一个全副本.

每个副本节点维护一个本地的日志,记录对副本进行的所有更新.节点还本地维护一个递增的Lamport Clock.另外,节点维护一个Logical Vector Clock(), ]表示节点已看到节点收到的Lamport Clock小于等于]的所有更新操作.

节点之间通过Pair-wise anti-entropy来传播本地的更新.假设节点向节点传播更新.首先节点先获取的,然后节点根据的内容扫描本地的日志并将没有覆盖到的更新发送给.每收到来自复制来的一条更新,根据更新的时间戳将其插入到日志中的恰当位置(保证所有更新按照时间戳由小到大排序),并且将本地的Lamport Clock更新为其和更新操作的Lamport Clock中的较大值.当收到所有来自的更新后,将最新的发送给,将更新为和中的较大值(Pair-wise Maximum).

和Bayou采用Primary节点决定Committed Order不同,TACT让每个副本节点决定何时Commit日志中的更新.假设节点的中最小项的值为,则表示节点已经看到了来自所有节点的Lamport Clock小于等于的更新,那么这些更新就是可以Commit的了.

当副本节点收到来自客户端的更新请求,将自己当前的Lamport Clock增加1,和自己的ID一起作为该更新操作的时间戳,并将其插入到日志中的恰当位置(保证所有更新按照时间戳由小到大排序),更新为当前的Lamport Clock.

SwiftCloud[22]

SwiftCloud数据库是第一个通过由云支持的因果一致的客户端本地缓存提供快速读取和写入.

每个副本节点本地维护一个版本向量,表示本地已经执行的更新操作的数量.

### Explicit Checking

#### COPS

#### (dependency list)

Eiger

#### Bolt-on causal consistency

Orbe

Dependency Matrices

ChainReaction

COPS-SNOW

Sequencer

### Stabilization

与Sequencer-based方法不同,Stablization方法允许分散化(Decentralization).它确保一个远程操作只有在被所有其他数据中心认为是稳定(Stable)的情况下才会在本地可见.为了实现这一技术,必须有一个定期(Periodically)运行的算法,以了解每个数据中心的状态,并根据该状态决定操作是否可以在本地执行(Applied).

GentleRain

CausalSpartan

Contrarian

Cure

AV

MongoDB

C3

Xiang, Vaidya

Wren

PaRis

### Lazy Resolution

OCCULT

POCC

## 检测

# 事务因果一致性

## 规约

## 理论

### Client-Assistant Lazy Dependency Resolution(CALDeR,in OCC[23])

OCC为了最大限度地减少元数据的开销和读取响应时间,采取了一种经济有效的方式暴露不稳定的项目版本,并且能够保证系统的因果一致性.这正是通过Client-Assistant Lazy Dependency Resolution(CALDeR)协议来实现的.CALDeR协议的主要思想是通过将追踪因果依赖关系的责任转移到客户端,从而将数据存储从追踪因果依赖关系的责任中解脱出来.CALDeR协议消除了分区之间的同步需求,这隐含地消除了由速度较慢的分区滞后引起的问题.下面将详细介绍该协议的做法.

客户端以客户端依赖元数据(Client-side Dependency Metadata)的形式存储执行读取时建立的因果关系的信息.如果客户端*c*读取了*Y*,并且存在*X*,使得*XY*,*c*会将*X*和*Y*之间的依赖关系记录到它的客户端依赖元数据中.当*c*执行以后的操作时,就可以通过客户端依赖元数据来读取所需的因果依赖关系.

当客户端*c*发起一个读操作(GET或RO-TX)请求时,服务器*p*需要客户端的依赖元数据来让确定它自己的状态是否与*c*的历史一致.考虑上一段中的例子,如果*c*想在读完*Y*之后再读*X*,通过*c*提供的依赖元数据和*p*本地存储的状态信息, *p*检查它是否已经收到*X*.如果*p*已经收到了*X*,那么*p*存储的最新的本地版本的*X*与*c*的历史兼容,它被返回给*c*.如果*p*还没有收到*X*,*p*必须在为*c*的请求服务之前收到它.在这种情况下,*p*会拖延*c*的请求,直到它收到*X*.

当客户端*c*发起一个写操作(PUT操请求时,需要客户端的依赖元数据来知道新创建的项目依赖于什么.我们仍然考虑前面的例子,当在服务器*p*上写入*Y*时,*p*必须已经交付了*Y*因果依赖的所有版本的*y*.也就是说,在OCC中,*p*必须等到它收到发行客户端所依赖的最新的Y的版本才能执行该写请求.

### Universal Stable Time(UST,in PaRis[24])

UST是PaRiS[10]实现的协议,以可扩展的方式识别稳定的快照.每个分区都维护着一个版本向量,表示最新应用的事务的时间戳,包括分区本身执行的事务和从远程replicas收到的事务.通过gossiping协议,同一DC内和跨DC的分区定期交换其版本向量中的最小时间戳.交换值的总体最小值确定了一个时间戳,即每个DC的所有分区都应用了时间戳小于等于该最小值的所有事务.UST使用单一时间戳来保证因果一致性,这减少了分区与分区之间以及客户与分区之间的通信开销,实现了高可扩展性和提高了效率.

仅利用UST并不能保证因果关系.事实上,通过保证分配给客户端*c*发出的事务提交时间戳(Commit Timestamp)大于分配给*T*的稳定快照(Stable Snapshot),使得提交时间戳能够反映因果关系.注意,*T*的提交时间戳可能大于 *c*请求的下一个事务所分配的快照.在这种情况下,这种快照将不包括*c*在*T*中进行的修改,这可能导致违反因果一致性所要求的read-your-writes性质.

### Binary Dependency Time(BDT,in Wren[25])

Wren实现了Binary Dependency Time(BDT)协议用来追踪数据项之间的因果依赖关系.BDT的关键特征是每个数据项只通过两个标量时间戳(Scalar Timestamp)来追踪因果依赖关系,而不管系统的规模如何.一个标量时间戳追踪本地项目(Local Items)的依赖关系,另一个标量时间戳总结远程项目(Remote Items)的依赖关系.与其他设计相比,只使用两个时间戳可以实现更高的效率和可扩展性.

Wren依靠BDT来实现Binary Stable Time(BiST),这是一个高效的稳定协议(Stablization Protocol),以确定更新操作何时可以包括在回复给DC内的客户提出的快照(即更新操作何时在DC内可见).BiST允许起源于一个DC的更新在该DC中立即变得可见,而不需要等待收到远程项目.相反,当一个远程更新*u*是稳定的之后才在DC内可见,这需要等到它的所有因果依赖项全部被DC接收.BiST计算出两个截止值(Cut-of Value)来分别表示那些本地和远程项目可以在一个DC内的事务中可见.BiST计算的本地部分是本地稳定时间(Local Stable Time,LST).远程分量是远程稳定时间(RST),与LST类似,表示本地DC内每个节点已经安装的远程快照的下限.通过解耦本地和远程项目,与使用单一标量时间戳进行依赖性跟踪的系统不同,BiST允许事务确定本地项目的可见性,而无需与远程DC同步.这种解耦在地理复制的情况下也能实现可用性和无阻塞的读取,因为对事务可见的快照只包括本地DC已经收到的远程项目.

通过BiST,DC内的分区定期交换他们所应用的最新本地和远程交易的提交时间戳(Commit Timestamp)后,每个分区计算LST(resp,RST)作为收到的本地(resp,远程)事务对应的时间戳的最小值.因此,LST和RST反映了本地和远程快照,这些快照已经被DC中的所有分区安装,并且事务从这些快照中读取而不被阻塞.

## 协议

### Read-only Transactions

GentleRain

Orbe

Contrarian

COPS-SNOW

### Read-only and Write-only Transactions

ISIS

TACT

Eiger

### Generic

Cure

Wren

Occult

SwiftCloud

系统类文章的协议的正确性证明

考虑抽象类似的几个协议；参考 chapar 的证明

## 检测

# 总结与展望

References:

[1] Burckhardt S, Gotsman A, Yang H, Zawirski M. Replicated data types: specification, verification, optimality. Proceedings of the 41st ACM Symposium on Principles of Programming Languages (POPL’2014). : 271–284.

[2] Burckhardt S. Principles of eventual consistency. Found. Trends Program. Lang., Now Publishers Inc., 2014, 1(1–2): 1–150.

[3] Ahamad M, Neiger G, Burns JE, Kohli P, Hutto PW. Causal memory: definitions, implementation, and programming. Distributed Computing, 1995, 9(1): 37–49.

[4] Lloyd W, Freedman MJ, Kaminsky M, Andersen DG. Don’t settle for eventual: scalable causal consistency for wide-area storage with cops. Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP’2011). : 401–416.

[5] Lamport L. Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 1978, 21(7): 558–565.

[6] Perrin M, Mostefaoui A, Jard C. Causal consistency: beyond memory. Proceedings of the 21st ACM Symposium on Principles and Practice of Parallel Programming (PPoPP’2016). .

[7] Bouajjani A, Enea C, Guerraoui R, Hamza J. On verifying causal consistency. Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL’2017). : 626–638.

[8] Viotti P, Vukolić M. Consistency in non-transactional distributed storage systems. ACM Comput. Surv., 2016, 49(1): 19:1-19:34.

[9] Du J, Iorgulescu C, Roy A, Zwaenepoel W. GentleRain: cheap and scalable causal consistency with physical clocks. Proceedings of the ACM Symposium on Cloud Computing. New York, NY, USA: Association for Computing Machinery, 2014: 1–13.

[10] Kulkarni SS, Demirbas M, Madappa D, Avva B, Leone M. Logical physical clocks. International Conference on Principles of Distributed Systems. 2014: 17–32.

[11] Fidge CJ. Timestamps in message-passing systems that preserve the partial ordering. Proceedings of the 11th Australian Computer Science Conference, Citeseer, 1987: 56–66.

[12] Mattern F, Others. Virtual time and global states of distributed systems. Parallel and Distributed Algorithms, Univ., Department of Computer Science, 1988.

[13] Du J, Elnikety S, Roy A, Zwaenepoel W. Orbe: scalable causal consistency using dependency matrices and physical clocks. Proceedings of the 4th Annual Symposium on Cloud Computing. New York, NY, USA: Association for Computing Machinery, 2013.

[14] Bravo M, Rodrigues L, Van Roy P. Saturn: a distributed metadata service for causal consistency. Proceedings of the Twelfth European Conference on Computer Systems. New York, NY, USA: Association for Computing Machinery, 2017: 111–126.

[15] Lloyd W, Freedman MJ, Kaminsky M, Andersen DG. Stronger semantics for low-latency geo-replicated storage. Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation. USA: USENIX Association, 2013: 313–328.

[16] Birman K, Schiper A, Stephenson P. Lightweight causal and atomic group multicast. ACM Trans. Comput. Syst., New York, NY, USA: Association for Computing Machinery, 1991, 9(3): 272–314.

[17] Ladin R, Liskov B, Shrira L, Ghemawat S. Providing high availability using lazy replication. ACM Trans. Comput. Syst., New York, NY, USA: Association for Computing Machinery, 1992, 10(4): 360–391.

[18] Terry DB, Theimer MM, Petersen K, Demers AJ, Spreitzer MJ, Hauser CH. Managing update conflicts in bayou, a weakly connected replicated storage system. Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP’1995). : 172–182.

[19] Demers A, Greene D, Hauser C, Irish W, Larson J, Shenker S, Sturgis H, Swinehart D, Terry D. Epidemic algorithms for replicated database maintenance. Proceedings of the sixth annual ACM Symposium on Principles of distributed computing. 1987: 1–12.

[20] Belaramani N, Dahlin M, Gao L, Nayate A, Venkataramani A, Yalagandula P, Zheng J. PRACTI replication. Proceedings of the 3rd Conference on Networked Systems Design &amp; Implementation - Volume 3. USA: USENIX Association, 2006: 5.

[21] Yu H. Design and evaluation of a continuous consistency model for replicated services. Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000). San Diego, CA: USENIX Association, 2000.

[22] Zawirski M, Preguiça N, Duarte S, Bieniusa A, Balegas V, Shapiro M. Write fast, read in the past: causal consistency for client-side applications. Proceedings of the 16th Annual Middleware Conference. New York, NY, USA: Association for Computing Machinery, 2015: 75–87.

[23] Spirovska K, Didona D, Zwaenepoel W. Optimistic causal consistency for geo-replicated key-value stores. IEEE Transactions on Parallel and Distributed Systems, 2021, 32(3): 527–542.

[24] Spirovska K, Didona D, Zwaenepoel W. PaRiS: causally consistent transactions with non-blocking reads and partial replication. 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS). 2019: 304–316.

[25] Spirovska K, Didona D, Zwaenepoel W. Wren: nonblocking reads in a partitioned transactional causally consistent data store. 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). 2018: 1–12.

附中文参考文献:

[2] 陈翔,顾庆,刘望舒,刘树龙,倪超.静态软件缺陷预测方法研究.软件学报,2016,27(1):1−25. http://www.jos.org.cn/1000-9825/4923. htm [doi: 10.13328/j.cnki.jos.004923]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 江雪(1990－),女,博士生,主要研究领域为分布式计算和形式化方法. |  |  | 黄宇(1981－),男,博士,教授,CCF专业会员,主要研究领域为分布式算法和形式化方法. |
|  | 魏恒峰(1986－),男,博士,主要研究领域为分布式计算(尤其是分布式数据一致性 问题(和形式化方法. |  |  |  |

1. \* 基金项目: 国家自然科学基金(61906090, U20B2064, 61773208); 江苏省自然科学基金(BK20191287, BK20170809); 中央高校基本科研业务费专项资金(30920021131); 中国博士后科学基金(2018M632304)

   收稿时间: 2020-04-13; 修改时间: 2020-10-26; 采用时间: 2020-12-14 [↑](#footnote-ref-1)