计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2014.20130058
51(9):2046-2057,2014

自稳定的分布式事务内存模型及算法

林 菲[1] 孙 勇[2] 丁 宏[1] 任一支[1]
[1](杭州电子科技大学软件工程学院 杭州 310018)
[2](浙江交通职业技术学院信息学院 杭州 311112)
(linfei@hdu.edu.cn)

Self Stabilizing Distributed Transactional Memory Model and Algorithms

Lin Fei[1], Sun Yong[2], Ding Hong[1], and Ren Yizhi[1]
[1](*School of Software Engineering*, *Hangzhou Dianzi University*, *Hangzhou* 310018)
[2](*Department of Information Technology*, *Zhejiang Institute of Communications*, *Hangzhou* 311112)

Abstract Aiming at the issue of transient fault tolerance in distributed system while taking into account the system's robustness and scalability, a self-stabilizing distributed transactional memory model, called SSDTM, is proposed. Firstly, the model frame is constructed by layered technology and collateral composition theory, which includes spanning tree layer, objects location layer, transaction proxy layer and application layer. Furthermore, the spanning tree algorithm is improved in self-stabilizing way, which can solve deficiencies of being only adaptive to stable environments based on existing methods. Then, data object manipulation algorithms are designed, which utilize data stream paradigm and self-stabilizing theory for locating objects and enhancing data access locality, and ensure mutually exclusive access to objects in distributed system with transient faults. Moreover, after building the transaction service model that defines the basic types of memory transaction states and operations, the concurrency control algorithms based on the improved logic clock are given. Finally, combining theoretical derivation and instance verification, the performance of SSDTM is verified and analyzed through multi-angle large-scale virtual tests based on 4 classical benchmarks in SimJava environment. Experimental results show that the algorithms presented in the paper exhibit good robustness and applicability, and SSDTM has higher system throughput and better fault-tolerance compared with other models under the same conditions.

Key words self stabilization; transactional memory; transient fault; spanning tree; ballistic target; logical clock

摘 要 针对具有瞬时故障的分布式系统,综合考虑系统鲁棒性和可扩展性,提出了一种自稳定的分布式事务内存模型(self-stabilizing distributed transactional memory, SSDTM).首先,利用分层技术和抵押组合理论建立模型框架,并对生成树算法进行了自稳定改进,以克服现有算法只能适应稳定环境的缺点;其次,将数据流技术与自稳定相结合,设计了数据对象操作方法,提高了系统的数据访问局部性;然后,在给出事务服务模型的基础上,提出了基于改进逻辑时钟的SSDTM并发控制算法;最后,结合理论

收稿日期:2013-01-14;**修回日期**:2013-12-23
基金项目:国家自然科学基金项目(61100194);浙江省自然科学基金项目(LY12F02017);浙江省公益性技术应用研究计划项目(2013C31130,2013C33082)
通信作者:林 菲(linfei@hdu.edu.cn)

推导,使用 4 个典型测试用例在 SimJava 环境下对 SSDTM 进行了多角度、大规模的分析和性能测试.结果表明,所提算法具有较强的参数鲁棒性和适用性,与其他模型相比,SSDTM 具有更高的吞吐量和容错性.

关键词　自稳定;事务内存;瞬时故障;生成树;弹道目标;逻辑时钟

中图法分类号　TP316

事务内存是一种通过事务来同步并发线程的编程模型,与基于锁机制的传统模型相比,事务内存具有易用性和高扩展性等特点.近年来,在多核、众核处理器中支持事务内存模型已成为研究热点[1-3],但是针对分布式系统的事务内存研究却很少,具有容错能力的分布式事务内存研究更是尚未起步.

鲁棒性是现代分布式系统中最重要的要求之一,一个实用的分布式系统应当能从进程和通信链路的瞬时(短时存在)故障中恢复.理想情况是一旦检测到故障出现就应当执行恢复操作,且不必依赖于从一个明确定义的状态启动的假设.自稳定是一个系统、组件、进程或对象自我修正的一种能力,近年来的应用研究已在自稳定的文件系统、路由选择、再编程和传感器网络方面取得了成功,说明自稳定原理能用于解决现实系统中轻量级的容错问题.在一个分布式系统(distributed system, DS)中,如果从任一状态开始,系统能在有限时间内自动恢复到特定的合法状态集合,则称 DS 是自稳定的.DS 的起始状态可能是由系统中瞬时故障所引起的一个故障状态,故障可能是被破坏的本地状态、丢失或不正确的消息等.自稳定系统满足了故障下的正确性这样一个很强的概念,如果一个瞬时错误使系统进入到一个不正确或不一致的状态,系统无需外力协助,最终将会稳定到一个正确状态.

本文针对具有瞬时故障的分布式系统,研究如何利用自稳定原理和分层技术高效地构建事务内存模型,首先提出了一种自稳定分布式生成树(self-stabilizing distributed spanning tree, SSDST)构建算法,然后基于 SSDST 提出了一种数据对象定位(ballistic data object location, BDOL)算法,最后实现了编程模型.本文主要贡献包括:1)首次将自稳定性引入事务内存,通过分层法设计了面向事务内存模型的一系列自稳定算法,以实现数据对象的可靠访问,同时解决了系统容错性问题;2)与传统面向单机的事务内存模型相比,本文所有算法均是分布式的,且能通过自稳定自动地实现快速故障恢复,并具有较高的系统性能.

1　理论基础

1.1　分布式计算模型

设分布式系统 DS 由 n 个具有唯一标识(假设为自然数)的异步进程 $P_1, P_2, \cdots, P_n$ 组成,它们之间通过通信网络进行消息传递.不失一般性,假设每个进程都各自运行在不同的处理器上.进程间没有可共享的全局存储,只能通过消息传递来进行交互.令 C_{ij} 表示 P_i 和 P_j 之间双向的通信信道($C_{ij} \Leftrightarrow C_{ji}$),$m_{ij}$ 表示由 P_i 发往 P_j 的消息.DS 的全局状态由各个进程的状态以及所有通信信道的状态组成.进程的状态则由其本地状态所描述并依赖于上下文,通信信道的状态由在该信道中传送的消息集合所描述.根据预定义的谓词 L(布尔表达式),DS 的全局状态又分为合法状态(L 为真)和非法状态(L 为假)两类.分布式算法(distributed algorithm, DA)由一组程序构成,每个程序运行于一个进程上,程序由一组变量和动作组成,变量只能由本地程序修改.本文将动作表述为形式如下的规则:

$$\langle ActionName\rangle :: \langle ActionPredicate\rangle \rightarrow \langle ActionSteps\rangle,$$

其中 *ActionName* 是动作名称;*ActionPredicate* 是决定是否执行动作的布尔型谓词,由进程本地状态决定,如果 *ActionPredicate* 为真,则称该动作被使能,只有被使能的动作才可以执行,存在使能动作的进程称为使能进程;*ActionSteps* 是动作执行体,负责修改本地变量.本文假设所有动作的谓词检测与执行均在一个原子步中完成.

定义 1. 配置.进程在 DA 中的状态由其所有变量决定,DA 的一个配置 c 是指时刻 t 中所有进程状态的集合,令 $c(P_i)$ 表示进程 P_i.

在配置 c 下的状态.一轮计算是指最小动作集合,使得所有使能进程至少完成一个属于该集合的动作.如果 c 可以通过 DA 的一轮计算转换为 c',则记为 $c \vdash\!\!\rightarrow c'$.如果在配置 c 下谓词 L 为真,则称 c 遵守 L,否则称 c 违反 L.

定义 2. 执行. DA 的执行记为 exe_{DA},是由一组配置组成的序列 $c_0,c_1,\cdots,c_i\cdots$,使得 $c_{i-1}\vdash c_i,i>0$. 将 c_0 称为起始配置,执行停止的条件是其最后一个配置中不存在使能进程,将该配置称为终止配置.

1.2 瞬时故障下的自稳定性

故障模型说明了系统及其组件可能的故障方式,由于解决任何特定问题的算法因采用故障模型不同而差异很大,所以指定故障模型非常必要. 瞬时故障是临时的(短时存在)且不会持久,它可能是由运行过程中进程状态或通信信道状态损坏所引起的,例如由电磁辐射或宇宙射线等问题造成的处理器状态变化. 一个瞬时故障可能改变系统的全局状态,但不能改变系统的行为. 本文仅考虑瞬时故障下的系统自稳定问题,其他故障类型(如进程或通信信道可以表现出任意行为的拜占庭故障[4])不在本文研究范围之内.

定义 3. 自稳定性. 设 L 和 L' 是两个谓词,exe 是 DA 中的任意一次执行,如果 DA 满足如下两个性质,称 DA 使 L' 稳定到 L,如果 $L'\equiv$true,称 DA 关于 L 是自稳定的:

1) 闭包. 如果 exe 的起始配置遵守 L 且在执行中没有瞬时故障,则 exe 的所有配置均遵守 L. 就是说一旦在 DA 中建立 L,它就不能被改变.

2) 收敛. 如果 exe 的起始配置遵守 L',则通过有限步计算,exe 能达到遵守 L 的一个配置.

自稳定系统无需外力协助,能自发地从一个或多个(把系统带入任意配置)瞬时故障中恢复. 在任意时刻,进程 P_i 正常运行或发生瞬时故障. 设 P_i 和 P_j 为在某时刻 t 下正常运行的两个进程,如果存在通信信道 C_{ij},则将这两个进程称为邻接进程,只有邻接进程才可以互相通信. 所有正常运行的进程(顶点集)及其邻接关系(边集)构成了动态拓扑图 $G=(P,adj,t)$,G 随着时刻 t 的变化而变动(本文假设 G 始终保持连通). 每个进程 P_i 需要实时维护一张记录其邻接进程标识的表 $P_i.adj$,为此 P_i 将定期通过所有 C_{ij} 向 P_j 广播检测消息,并根据响应消息或超时的触发来更新 $P_i.adj$.

本文将在以上理论基础与假设的前提下,设计构建具有自稳定性的分布式事务内存模型.

2 事务内存模型

本文介绍具有自稳定性的分布式事务内存模型(self-stabilizing distributed transactional memory, SSDTM)的建模方法,在 SSDTM 中,每个进程 P_i 均由 4 层构成:$Tree_i$ 通过执行 SSDST 算法维护生成树信息;$Location_i$ 通过执行 BDOL 算法维护数据对象位置信息;$Proxy_i$ 为事务内存代理模块,并为上层提供相应服务;App_i 为用户特定的应用程序,如图 1 所示. 本节首先给出自稳定分布式生成树的设计方法,在此基础上,进一步阐述如何利用生成树设计服务于分布式事务内存的数据定位算法,最后介绍 SSDTM 的服务和并发控制策略.

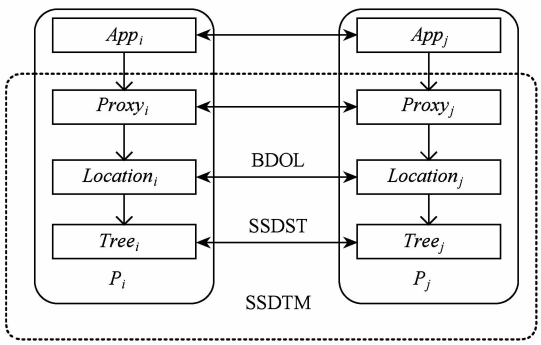

Fig. 1 SSDTM model framework.

图 1 SSDTM 模型框架

为简化算法设计,SSDTM 采用文献[5]中的"抵押组合"理论来实现分层架构. 将算法 DA 和 DB 的抵押组合算法记为 DA⊙DB,其中 DA 和 DB 并发执行,DB 的执行中会使用 DA 的输出结果. 在 SSDTM 中,对组合算法 SSDST⊙BDOL 作出进一步限定,即只有在 SSDST 中不存在使能动作的前提下,BDOL 才被允许执行. 根据文献[5]可得出如下推论.

推论 1. 如果算法 SSDST 和 BDOL 均是自稳定的,则 SSDST⊙BDOL 也是自稳定的.

2.1 SSDST 算法

生成树是许多复杂分布式协议的基础,在时刻 t 下,G 的生成树 $Tree=(P,adj')$ 是一个图,其中 adj' 是 adj 的子集,且 P 中每一对进程间仅存在一条路径. 根据前面介绍可知,G 中的 P 和 adj 都是动态变化的.

SSDST 算法的核心任务是持续地维护动态拓扑图 G 的覆盖子图 T,使其最终成为生成树,同时还需要保证 SSDST 在任意状态下启动时的正确性,以实现自稳定. 在该算法中,每个 $Tree_i$ 维护一个指向 P_i 当前父节点进程的变量 fa_i、一个指向当前根节点进程的变量 r_i 和一个记录 P_i 与 $P_i.r_i$ 之间最短距离的变量 d_i(最大值为 $n-1$). 由于进程可能从任意状态下启动,所以 fa_i 的初始值可以是任

意值,可能导致 T 的初始值是一个森林或存在环.对于 T 初始值是森林的情况,SSDST 用合并方法将森林化为一棵树.在时刻 t,对于两个邻接进程 P_i 和 P_j,如果 $r_i<r_j$(比较两个根节点的进程标识),则将 P_i 的父节点设为 P_j,根节点设为 P_j 的根节点;对于 T 初始值存在环的情况,SSDST 通过计算路径长度的方法来检测并消除环.为检测 T 中是否存在环,$Tree_i$ 周期性地将其 d_i 值设为 $d_{fa}+1$(d_{fa} 为 P_i 父节点的 d 值),如果 T 中存在环,经过几轮计算后必然存在某个 $d_i>n-1$,这与 d_i 最大值为 $n-1$ 产生矛盾,此时 $Tree_i$ 通过将 P_i 设为自身的父节点来消除环.由于初始状态的任意性和瞬时故障的影响,还需要考虑 fa_i,r_i,d_i 3 个变量初始值错误或不一致的情况,根据以上描述,SSDST 算法在每个进程中执行 4 个动作(优先级从 $A1\sim A4$ 降序),见算法 1.

算法 1. 进程 P_i 的 $Tree_i$ 构造算法.

变量:fa_i,r_i,d_i.

① $A1::(fa_i=j)\wedge(j\in P_i.adj)\wedge(d_i<n)\wedge((r_i\neq r_j)\vee(d_j+1\neq d_i))\rightarrow r_i:=r_j;d_i:=d_j+1$;

② $A2::((r_i<i)\vee((fa_i=i)\wedge((r_i\neq i)\vee(d_i\neq 0))))\vee(fa_i\notin(P_i.adj\cup\{i\}))\rightarrow r_i:=i;fa_i:=i;d_i:=0$;

③ $A3::(d_i>n-1)\rightarrow r_i:=i;fa_i:=i;d_i:=0$;

④ $A4::((r_i<r_j)\wedge(j\in P_i.adj)\wedge(d_j<n))\vee((r_i=r_j)\wedge(j\in P_i.adj)\wedge(d_j+1<d_i))\rightarrow r_i:=r_j;fa_i:=j;d_i:=d_j+1$.

令算法 SSDST 的检测谓词 $L\equiv T$ 是一棵生成树,接下来证明 SSDST 关于 L 的自稳定性.

定理 1. 对于任意一个配置 c,如果此时 SSDST 中不存在使能动作,则 T 是一棵生成树.

证明. 采用反证法.假设此时 T 不是一棵生成树.则有以下两种情况:

1) T 中存在环.此时必定存在一个长度为 $l(l>2)$的邻接进程标识数组 $ID=[a,b,\cdots,y,z=a]$,使得对于每个进程 $P_{ID[i]}$,$0\leqslant i<l-1$,有 $fa_{ID[i]}=ID[i+1]$,即数组中每个进程的后继进程是其父节点,同时易知 $d_{ID[i]}=d_{ID[i+1]}+1$,$0\leqslant i<l-1$(否则动作 $A1$ 将被使能).通过传递性可得到 $d_a=d_z+l-1$,即 $d_a=d_a+l-1$,等式不成立,矛盾.

2) T 不连通.设 T_1 和 T_2 是 T 的两个子图,且互不连通.因为本文已假设 G(T 覆盖在 G 上)始终保持连通,所以必存在 $P_i\in T_1$,$P_j\in T_2$,使得 P_i 和 P_j 互为邻接进程,即 $j\in P_i.adj$.因为 P_i 和 P_j 位于不同子图,所以 $r_i\neq r_j$(DS 中每个进程标识是唯一的).设 $r_i<r_j$($r_i>r_j$ 的情况同理可证),此时如果 $d_j>n-1$,将使能动作 $A3$;如果 $d_j<n$,将使能动作 $A4$,矛盾.

综上,两种情况均产生矛盾,即假设不成立.

证毕.

定理 2. 从任意一个配置 c 中启动,SSDST 会在有限步计算后使 T 成为一棵生成树.

证明. 首先定义如下关于配置 c 的函数:

$$\phi(c)=\langle wrong(c),cycle(c),forest(c)\rangle,\quad(1)$$

其中,$wrong(c)=|\{i|P_i\text{ 状态值有误}\}|$,$cycle(c)$为 c 中环个数,$forest(c)$为 c 中森林个数.对于任意配置 c_1,c_2,其函数 ϕ 值的关系$\prec$定义如式(2):

$$\langle wrong(c_1),forest(c_1),cycle(c_1)\rangle\prec\langle wrong(c_2),forest(c_2),cycle(c_2)\rangle\equiv\phi(c_1)\text{字典序小于}\phi(c_2),\quad(2)$$

易知,$\prec$是一个良基关系[3],因此以 $x|x\in\phi$ 的最小值为始的降链($x_{\min}\prec\cdots\prec x$)都是有限长的,函数 ϕ 在关系$\prec$下的最小值 $x_{\min}=\langle 0,0,0\rangle$.根据算法 1 中的 4 个动作及其优先级可知,$wrong(c)$和 $cycle(c)$的值在 SSDST 执行中均是非递增的,而 $forest(c)$的值在两种($A2$ 与 $A3$)执行情况下是递增的,但动作 $A2$ 使得 $wrong(c)$递减;动作 $A3$ 使得 $cycle(c)$递减,所以 SSDST 执行中函数 ϕ 的值在关系$\prec$下是逐步递减的.根据良基关系的降链有限性质可知,系统最终将达到 ϕ 的最小值,此时 T 为一棵生成树.

证毕.

现有的各种领导人选举算法与生成树构建算法均基于所有进程从特定状态启动的假设,这对达到系统自稳定目标造成严重阻碍.SSDST 算法取消了这个限制,允许系统从任意状态下启动,以实现自稳定.

2.2　BDOL 算法

根据计算方式不同,分布式事务系统可分为"控制流"和"数据流"两种类型.在控制流系统中,数据对象是静止不动的,计算则在不同节点间以远程过程调用(remote procedure call, RPC)的形式移动,同时需要死锁检测与提交协议来实现同步;在数据流系统中,事务是静止的,数据对象可在节点间移动,只要不存在冲突事务就可以提交.与控制流系统相比,数据流系统具有如下优点:1)易于使用,不需要通过"锁"来实现同步;2)可以通过数据对象的流动提高系统局部性,减少节点间通信,避免"热点"的产生.基于以上考虑,本文的分布式事务内存系统采用了数据流模型.

基于数据流事务系统的核心功能是对数据对象的定位,当进程 P_i 中一个事务访问的对象不在本地时,定位算法需要找到该对象,将其移动至 P_i 所在节点的缓存中,并废止对象所有旧的版本. BDOL 算法的目标是在 SSDST 算法的基础上完成上述工作,同时保证系统的自稳定.

对于每个数据对象 do,需要在每个进程的 $Location_i$ 中维护一条路径信息变量 dir_i,如果 $dir_i=P_i$,说明 do 已经或将要位于 P_i;如果 $dir_i=P_j$,$j\in P_i.adj$,说明 P_j 知道 do 的位置. 易知,如果生成树 T 的最大度为 deg,每个 dir_i 要占用 $\mathrm{lb}(deg)$ 比特的内存空间. 假设每个 $Location_i$ 可以在一个原子步内完成接收消息(输入)、计算处理和发送消息(输出)的操作. BDOL 算法主要包括发布、查找、移动 3 个操作和 1 个自稳定协议,数据对象被创建时或生成树重建后需要调用发布操作,使得其他进程可以定位到该对象,如算法 2 所示;查找操作服务于事务中的数据对象读取,用来获取数据对象最新的只读副本;移动操作服务于事务中的数据对象更新,用来获取数据对象.

算法 2. 发布算法 $P_i.Publish(do)$.

变量: i,j / * 表示 2 个不相等的整型变量 * /.

① $dir_i:=i$; / * dir 为相应 $Location$ 模块中 do 的路径信息 * /

② While ($fa_i\neq i$) / * 将 do 的路径信息在生成树 T 中沿着父节点逐层向上传递,直至根节点 * /

③ $j:=fa_i;dir_j:=i;i:=fa_i$;

④ End While

定义 4. 弹道目标. 数据对象 do 的弹道目标是指相应的路径信息变量指向其自身的进程 $P_i(dir_i=i)$.

查找和移动操作可被任意进程调用,核心任务都是定位 do 的弹道目标,包含"上升"和"下降"两个阶段. 上升阶段中沿着父节点逐层向上查找到根节点直至遇到路径信息非空进程,下降阶段则沿着相应路径信息逐层向下,两个阶段中如果遇到 do 的弹道目标则完成了定位任务,由于该查找过程类似于弹道导弹的运行轨迹,所以将算法命名为弹道数据对象定位算法(BDOL),具体见算法 3.

算法 3. 弹道数据对象定位算法 $P_i.Locate(do,op)$.

变量: i,j,dir,top / * i,j,dir,top 表示 4 个整型变量 * /

① If ($dir_i=i$) Return P_i;End If / * do 已经或将要位于 P_i * /

② $dir:=dir_i;top:=i$;

③ If ($op=$移动) $dir_i:=i$;End If / * 对于移动操作,P_i 将成为 do 新的弹道目标 * /

④ While (($fa_i\neq i$) ∧ ($dir_i=$null)) / * 上升阶段 * /

⑤ $j:=fa_i;dir:=dir_j;top:=j$;

⑥ If ($op=$移动) $dir_j:=i$;End If

⑦ If ($dir=j$) Return P_j;End If

⑧ If ($dir\neq$null)Break;End If / * 进入下降阶段 * /

⑨ $i:=fa_i$;

⑩ End While

⑪ While ($dir\neq top$) / * 下降阶段 * /

⑫ $i:=top;top:=dir;dir:=dir_{top}$;

⑬ If ($op=$移动) $dir_{top}:=i$;End If

⑭ End While

⑮ Return P_{dir}.

在移动操作的定位过程中,进程 P_i 在向下一个进程发出定位请求前要将 dir_i 指向其上一个进程,以更新路径信息,并将数据对象移动请求通过底层网络直接发送至弹道目标. 接下来讨论 BDOL 算法的自稳定性.

定义 5. 弹道状态. 弹道状态包括稳定、合法和非法 3 种情况. 如果从任一节点出发,沿着 dir 中的路径信息均可到达唯一的弹道目标,并且任一通信信道中不存在定位消息,则称该状态是稳定的;以一个稳定状态为起点,经过有限次定位操作后可能达到的状态为合法状态;既不稳定又不合法的状态为非法状态. 可知,稳定状态也是合法状态.

以图 2 中的状态图为例,用箭头标明了 dir 的

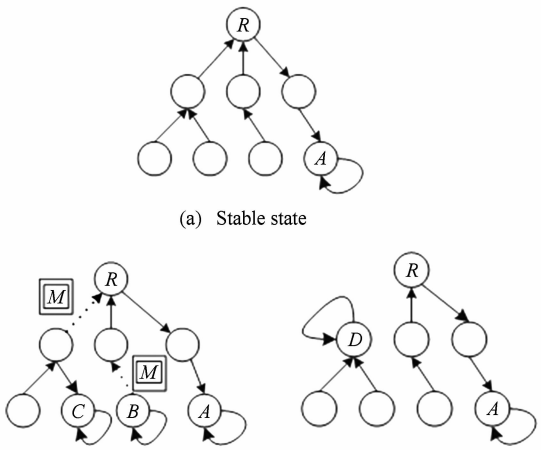

Fig. 2 Instances of three ballistic states.

图 2 3 种弹道状态实例

方向,虚线边表示通信信道中存在定位消息.图 2(a)显示了一个稳定的弹道状态,其中 R 为生成树 T 的根节点,A 为弹道目标;图 2(b)显示了一个合法的弹道状态,其中 M 为传输中的定位消息,节点 B 或 C 将成为新的弹道目标;图 2(c)显示了一个非法的弹道状态,因为其中存在着 2 个弹道目标(A 和 D).

算法 BDOL 自稳定的目的是从任意弹道状态出发,经过有限次计算,使系统进入合法状态.令算法 BDOL 的自稳定检测谓词 $L\equiv T$ 中的弹道状态是合法的,可以看出 L 是一个全局谓词,与所有 dir 的值和通信信道中的定位消息相关,接下来证明可将对 L 的检测转化为对本地谓词的检测,以简化自稳定工作.

设通信信道 C_{ij} 为生成树 T 中连接节点 P_i 和 P_j 的边,对于一个稳定状态,易知此时 $dir_i=j$,或 $dir_j=i$,但两者不能同时成立(因为只存在一个弹道目标);对于一个非稳定状态,如果 C_{ij} 中存在从 $P_i\sim P_j$ 的定位消息 M,可知 P_i 在发送 M 之前 $dir_i=j$ 成立,但算法 BDOL 会在 M 发送后将 dir_i 指向其他节点,使得此时 P_i 和 P_j 均未指向对方.设 $LMN(C_{ij})$ 表示 C_{ij} 中定位消息的数量,用 $DIR(i,C_{ij})=1$ 表示 $dir_i=j$,否则 $DIR(i,C_{ij})=0$.关于 C_{ij} 的函数定义如下:

$$\phi(C_{ij})=LMN(C_{ij})+DIR(i,C_{ij})+DIR(j,C_{ij}). \tag{3}$$

定义 6. 合法通信信道.通信信道 C_{ij} 是合法的当且仅当 $\phi(C_{ij})=1$,此时以下 3 个条件有且仅有一个成立:1)$LMN(C_{ij})=1$;2)$dir_i=j$;3)$dir_j=i$.

定理 3. 生成树 T 中的弹道状态 S 是合法的,当且仅当 T 中的每条边都是合法通信信道.

证明. 首先证明 S 是合法的$\Rightarrow T$ 中所有边都是合法通信信道,考虑以下 2 种情况:

1) S 是稳定的.对于任意的通信信道 C_{ij},可知此时 C_{ij} 中不存在定位消息,即 $LMN(C_{ij})=0$,且节点 P_i 和 P_j 不会同时指向对方(因为稳定状态中只存在一个弹道目标).假设 P_i 和 P_j 均未指向对方,根据定义 5 可知,存在着从 P_i 出发的路径 $Path_i$ 和从 P_j 出发的路径 $Path_j$,且路径 $Path_i$ 和 $Path_j$ 的终点均为弹道目标 Tar,因此在生成树 T 中构成了一个由 C_{ij},$Path_i$,$Path_j$ 和 Tar 组成的环,这与 T 是一棵树矛盾,假设不成立,即 P_i 和 P_j 中有且仅有一个节点指向对方.所以对任意通信信道 C_{ij},$\phi(C_{ij})=1$.

2) S 是合法但不稳定的.对于任意节点 P_i,此时可能存在以下 3 种情况:

① P_i 接收到自身发送来的定位消息,它将该消息转发到 dir_i,并将 dir_i 设置为 P_i;

② P_i 接收到 P_j 发送来的定位消息且 $dir_i\neq i$,它将该消息转发到 dir_i,并将 dir_i 设置为 P_j;

③ P_i 接收到 P_j 发送来的定位消息且 $dir_i=i$,它将该定位消息插入请求消息队列,并将 dir_i 设置为 P_j.

易知,在以上每种情况下,$\phi(C_{ij})=1$ 均成立.接下来证明 T 中所有边都是合法通信信道$\Rightarrow S$ 是合法的.假设 DG 为 T 中所有边都是合法通信信道时由所有节点及其 dir 组成的有向图,易知 DG 每个顶点的出度为 1,且只有弹道目标节点上存在一个指向其自身的有向环,因为如果存在其他有向环将在生成树 T 中也构成环.设 k 为 DG 所有边上定位消息的数量,本文将采用数学归纳法来证明对于任意 k,存在着稳定状态 S' 和一个有限定位操作序列 $lseq$,使 S' 能够到达 S.

当 $k=0$,即通信信道中没有定位消息时,若从任一节点出发,沿着 dir 将到达唯一弹道目标,则 S 就是稳定的,即 $S'=S$,$lseq=$ null.使用反证法,假设 S 中存在两个弹道目标 Tar_1 和 Tar_2,设 $Path$ 为 T 中连接 Tar_1 和 Tar_2 的路径,因为 Tar_1 和 Tar_2 均指向其自身且每个顶点的出度为 1,由此可知 $Path$ 中必存在一条通信信道 C_{ij},使得 P_i 和 P_j 均未指向对方,即 $\phi(C_{ij})=0$,矛盾.因此 $k=0$ 时 S 是稳定的.

假设当 $k<l$ 时归纳假设成立,考虑 $k=l$ 时 S 的合法性.不妨设 $k=l$ 时 C_{ij} 中存在一个定位消息 M,根据已知条件 $\phi(C_{ij})=1$,所以 P_i 和 P_j 均未指向对方,同时根据上文可知以 i 为起点,可沿唯一路径 $Path$ 到达弹道目标 i_0,且 $Path$ 中所有通信信道中无消息(否则 $\phi>1$),如图 3(a)所示.接着考虑图 3

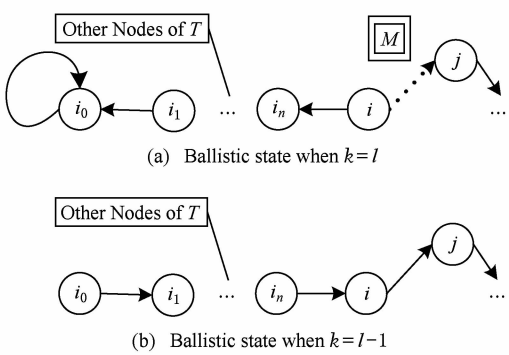

Fig. 3　Legality analysis diagram of ballistic state.

图 3　弹道状态合法性分析示意图

(b)中的弹道状态 S',将 $Path$ 中的所有箭头反转,并去除 i_0 的自指向,且 $dir_i=j$,其余部分不变,根据归纳假设易知 S' 中 $k=l-1$,并且是稳定的.

从 i_0 发出定位消息,构造定位操作序列 $lseq=i_0\rightarrow i_1\rightarrow\cdots\rightarrow i_n\rightarrow i\rightarrow j$,可以使 S' 到达 S,所以 S 是合法的. 证毕.

结合定义 6 和定理 3,只需使生成树 T 的每条边为合法通信信道,即可保证算法 BDOL 的自稳定.令 F_{ij} 表示通信信道 C_{ij} 的父节点(2 个节点中距离 T 的根节点较近的 1 个),该节点是固定的,与路径方向无关.自稳定协议的核心思想是每个父节点周期性地检测其通信信道并计算 ϕ,如果发现 $\phi\neq1$,则执行以下修正动作,见算法 4.

算法 4. 自稳定算法 $F_{ij}.SelfStabilize()$.

变量:fa,ch. /* fa 表示通信信道 C_{ij} 的父节点,ch 为 fa 的子节点 */

/* 向通信信道 C_{ij} 中发送一条定位消息,使 ϕ 的值加 1 */

① $A5::(\phi(C_{ij})=0)\rightarrow P_{fa}.Locate(do,op)$;

/* 使 ϕ 的值减 1 */

② $A6::(\phi(C_{ij})>1)\wedge(dir_{fa}=ch)\rightarrow dir_{fa}=fa$;

/* 此时 C_{ij} 中必定存在定位消息,当其最终达到 fa 时,ϕ 的值将自动下降,所以无需处理 */

③ $A7::(\phi(C_{ij})>1)\wedge(dir_{fa}\neq ch)\rightarrow nop$.

2.3 SSDTM 服务建模

SSDTM 是以文献[4]为基础,并将其中的定义扩展至分布式系统;同时以上文中的分层抵押组合算法 SSDST⊙BDOL 为支撑,确保系统的鲁棒性.

在 SSDTM 中,具有唯一 id 的事务在一组共享数据对象上执行读写操作,令 $DO:=\{do_1,do_2,\cdots\}$ 表示全体数据对象的集合,$TX:=\{tx_1,tx_2,\cdots\}$ 表示全体事务的集合.事务 tx 被进程 P_i 中的 $Proxy_i$ 调用执行并向 App_i 提供服务(如图 1 所示),tx 只能处于 live(活动)、aborted(撤销)、committed(提交)这 3 种状态中的 1 种,将该属性记为 $tx.status$.如果一个已撤销事务被重新调度执行,系统则将其视为一个新事务(id 值变化).事务执行过程中还需维护 $readSet$(读对象集合)和 $writeSet$(写对象集合).

SSDTM 的执行模式采用"数据流"方式,即事务是静止的(运行于单一节点内),数据对象则可在节点间移动.$Proxy_i$ 作为事务代理对外提供 2 种基本服务:read,write,分别用于数据对象的读写操作,具体执行过程描述如下.

当事务 tx_i 发出一个对数据对象 do 的 read 或 write 服务请求时,将触发一个服务请求事件 $sinv_i(do,op,arg)$,其中 op 为服务类型(read 或 write),arg 为传递给服务的参数(read 操作时为空值⊥,write 操作时为写入值 v).事件 $sinv_i$ 将被转发给相应的 $Proxy_i$,并触发一个数据对象定位事件 $dol_i(do,op)$,以获得 do 的只读或可写副本.$Proxy_i$ 首先检查 do 是否在本地,如果没有则调用 $Location_i$ 中的 BDOL 算法在网络中定位 do.每个 dol_i 事件应与一个响应事件 $dor_i(do,op)$ 匹配,对于 read 操作,dor_i 返回 do 的只读副本;对于 write 操作,BDOL 算法将把位于远程节点的 do 移动至 tx_i.当 $Proxy_i$ 接收到 dor_i 事件后,服务响应事件 $sres_i(do,op,v)$ 将被触发,其中 v 为返回值(read 操作时为 do 值,write 操作时为 ok).事件 $sres_i$ 与事件 $sinv_i$ 只有在被同一个事务调用且使用同一个操作访问同一个数据对象的情况下才是匹配的;事件 dol_i 与事件 dor_i 只有在被同一个 $Proxy_i$ 调用且使用同一个操作访问同一个数据对象的情况下才是匹配的.

事务 tx_i 在结束执行前还可能触发两个事件:co_i 事件和 ab_i 事件.co_i 事件表示 tx_i 请求提交,ab_i 事件表示 tx_i 被系统强制撤销.撤销事件 ab_i 也可能作为 $sinv_i$ 或 dol_i 的响应消息被 tx_i 接收到,此时系统也会撤销 tx_i.本文中,SSDTM 服务都是顺序执行的,即事务 tx_i 只有在接收到上一个操作的响应后才会发起新的服务请求,或者一个新的 $sinv_i$ 事件只能在 $sres_i$ 事件、co_i 事件或 ab_i 事件之后发生.

2.4 SSDTM 的并发控制

对任何分布式系统,并发控制都是一个重要且困难的问题.针对并发访问分布式数据对象的问题,目前存在两种解决方案:原子广播法[6]和版本控制法[7].原子广播法的缺点是扩展性不佳,因为广播消息数是以节点数平方的形式增长;版本控制法则需要为异步执行的分布式事务维护一个全局时钟,这将加重系统通信负担[8],同时,由于在分布式系统中不可能有全局物理时间,实际的全局时钟只可能是一个近似值.针对这个问题,出现了以跳跃方式行进的逻辑时钟[9]的概念,它能充分捕获由分布式系统中的因果关系导致的单调性.在一个逻辑时钟系统中,每个进程都有一个按规则运行的逻辑时钟(或本地时钟).每个事件都被赋予一个时间戳,事件之间的因果关系一般从它们的时间戳推导出来.赋予事件的时间戳遵守基本单调性,也就是说,如果事件 a 的因果性影响事件 b,那么 a 的时间戳小于 b 的时间戳.

本文设计了一种基于逻辑时钟的事务控制(logical clock transaction control, LCTC)算法,通过推迟不冲突事务的起始时间 st(事务启动时的本地时钟)和调整本地时钟 lc,可以使冲突事务提前撤销并解决了物理时钟同步问题.在 LCTC 算法中,每个节点的 lc 值被附加到所有消息内,通过节点间消息传递完成同步,以建立用于事务控制的 happens-before 关系[10],保证算法的活性和公平性;每个数据对象需维护一个版本锁 vl(第 1 位是锁标记,其余位是版本号),事务成功提交时将更新所有相关 vl.

节点 P_i 中的事务 tx_i 访问本地数据对象时,如果数据对象版本号大于等于 $tx_i.st$,则将 tx_i 撤销.但在访问远程数据对象时,由于节点间时钟的不同无法直接判断,所以如何在不影响系统性能前提下实现时钟同步以完成事务并发控制是 LCTC 算法的主要挑战.为此,设远程数据对象所在的节点为 P_j,当 P_j 接收到数据访问请求时,首先将数据对象与本地时钟 lc_j 一同返回,然后将 lc_j 与从请求消息中提取的时钟 lc_i 进行比对,如果 $lc_j<lc_i$,则将 lc_j 置为 lc_i 以调整本地时钟.当 P_i 接收到 P_j 的应答消息后,需作如下验证:如果 $lc_j\leqslant tx_i.st$,则数据对象可被安全访问;否则,将 lc_i 和 $tx_i.st$ 调整至 lc_j,然后检查 $tx_i.readSet$,如果其中某个数据对象版本号大于 lc_i,则撤销 tx_i.

事务 tx 完成计算后需对其全体读写集进行验证,以保证数据一致性,见算法 5.

算法 5. 提交算法 $tx.Commit()$.

变量:P,do /* P 为 tx 所在的节点进程;do 为数据对象 */

① Foreach do in $tx.writeSet$
② $do.acquireVersionLock()$;
/* 如果 do 是远程对象,则调用 BDOL 算法获取版本锁 */
③ End Foreach
④ Foreach do in $tx.readSet$
/* 再次验证 $readSet$ 以保证数据一致性 */
⑤ If ($do.version>tx.st$) $tx.Abort()$; End If
⑥ End Foreach
⑦ $P.lc$++; /* 调整本地时钟 */
⑧ Foreach do in $tx.writeSet$
⑨ $do.commitValue()$; /* 如果 do 是远程对象,则执行 BDOL 算法中的移动操作 */
⑩ $do.setVersion(P.lc)$;
/* 调整 do 的版本号 */
⑪ $do.releaseVersionLock()$; /* 将 $do.vl$ 中的锁标记位清零 */
⑫ End Foreach

3 实　　验

3.1 实验方法和环境

为了验证 SSDTM 相关算法的有效性和系统性能,本文基于 Java 语言实现了一个原型系统,系统架构如图 1 所示.在原型系统上,本文设计了 2 个实验分别对 SSDST 算法和 SSDTM 模型进行验证,并采用了英国爱丁堡大学研发的分布式离散事件模拟软件 SimJava[11] 进行环境仿真与参数测量,选取了 4 种用于模拟分布式系统的连通图作为测试用例进行实验:

1) random graph 是一种任意两个节点的连接(两点之间存在边)概率为 p 的随机图,可通过增加边保持图的连通性;

2) bipartite graph 是一种将所有节点划分成两个不相交集 U 和 V 的二分图,使得图中每条边的两个顶点分别位于 U 和 V 中;

3) spiral graph 是一种形似螺旋结构的图,即对于图 $SG=(V,E)$,$V=\{v_1,\cdots,v_n\}\mid v_1<\cdots<v_n$,且 $E=\{(v_1,v_n),(v_n,v_2),(v_2,v_{n-1}),(v_{n-1},v_3),\cdots,(v_{\lfloor n/2\rfloor},v_{\lfloor n/2\rfloor+1})\}$;

4) k-local graph 是一种限定了边长(最长为 k)的图,即$(v_i,v_j)\in E\Leftrightarrow|i-j|\leqslant k$.

对 SSDTM 系统性能的影响主要体现在节点个数、节点本身处理能力及安全级别、节点间链路带宽及安全可靠性、节点和链路瞬时故障率的差异上,因此设定了如下实验参数:

1) n 为网络中全体节点的个数.

2) vf 为节点处理能力因子($0<vf\leqslant 1$).由于分布式系统中大部分节点单位时间内处理数据量的大小各异,但会在一定的范围内变化,因此可以通过 $vmax\times vf$($vmax$ 为最大值)来表示每个节点的处理能力.

3) ef 为链路带宽因子($0<ef\leqslant 1$).用来表示每条通信信道的连接带宽,与 vf 类似,这里不再赘述.

4) tfr 表示网络中全体节点和链路的瞬时故障发生率.

5) ccr 为计算通信比.定位为系统中计算量和通信量的比值,ccr 越大,说明任务类型更接近计算密集型;反之,说明任务间交互及数据通信量越大,

任务类型更接近通信密集型.

对本文所进行的实验设定如表 1 所示的参数,并通过 SimJava 控制这些参数值的变化就可以模拟出不同的分布式环境及任务.

Table 1 Parameter Setting of Simulation Experiments

表 1 仿真实验参数设定

Parameter Name	Parameter Value
n	50,100,150,200,250,300
vf & ef	0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1
tfr	0.01,0.02,0.05
ccr	0.1,1,10

本文实验平台的软硬件配置:CPU 为 Intel Corei5(4 核) 2.53 GHz;内存为 4 GB;操作系统为 Windows 7 Ultimate 64 b;编程语言及工具为 Eclipse Java 3.5.2;分布式系统仿真软件为 SimJava 1.2.

3.2 实验结果

本节首先从收敛时间和执行效率等指标对本文提出的自稳定算法 SSDST 进行实验分析,然后从系统吞吐量和任务类型两个目标对不同的分布式事务内存系统进行了仿真实验及综合分析.

实验 1. SSDST 算法的收敛时间.

本实验的目的是为了验证 SSDST 算法的自稳定效率,实验启动时,每个 vf 和 ef 的值从表 1 中随机获得,然后通过变化 n 的值来比较在 3 种瞬时故障率和 2 种计算通信比的情况下算法的性能差异.当所有节点不再发送消息时表明系统已稳定,本次实验结束,将一次实验所用的时间称为收敛时间,对于每个 n 值分别用 4 种测试用例图进行 10 次测试,取平均值作为收敛时间.实验结果如图 4 所示:

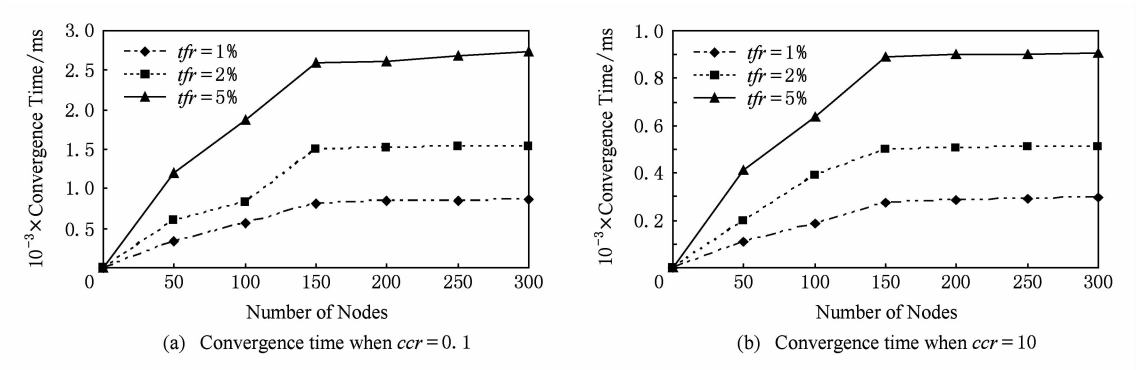

Fig. 4 Comparison results of SSDST algorithm convergence time.

图 4 SSDST 算法收敛时间及结果比较图

从图 4 可以看出,当节点数 n 较少时(小于 150),SSDST 构建生成树所花的收敛时间都呈线性增长;随着网络规模的扩大,收敛时间则以对数形式($\mathrm{lb}(n)$)增长,变化幅度不大,优势更加明显.这主要是因为本文提出的 SSDST 算法可根据局部信息完成生成树的构建,具有高度并发性,在一定网络规模下,收敛时间只与生成树的高度相关,从而使其具有较高的执行效率.同时,对比图 4(a)和图 4(b)可以看出,SSDST 更适合计算密集型应用.

实验 2. 工作负载变化情况下的性能比较.

本实验的目的是为了对 SSDTM 的系统吞吐量进行评估,并与另一种典型分布式事务内存系统 GenRSTM[12] 进行性能分析比对.与 SSDTM 的 LCTC 算法不同的是,GenRSTM 采用组通信和原子广播的形式进行事务并发控制.本次实验选用的事务程序(作为 App 层嵌入 SSDTM 中)和数据来自斯坦福大学开发的 STAMP[13] 中的银行借贷基准测试套件.实验中,每个节点运行 8 个线程,每个线程执行 50～100 个事务,取 10 次测试的平均值,在实验参数 $tfr=0.01$,$ccr=1$,vf 和 ef 为随机值时的结果如图 5 所示.

从图 5 可以看出,当节点数 n 较少时,GenRSTM 的性能优于 SSDTM,但随着节点数的增多,SSDTM 在读写任务的各种比例情况下均表现出优于 GenRSTM 的性能.在以写任务为主的环境下 SSDTM 的优势更为明显,这是因为 GenRSTM 需要通过原子广播将数据变动信息传送给所有副本,增加了开销.同时,实验发现 GenRSTM 在节点较多的情况下(300 以上)将出现较高的崩溃率,而由于 SSDTM 是构建在自稳定平台上,所以不存在系统崩溃的问题.

从上述实验结果可以看出:在存在瞬时故障的情况下,SSDTM 可在各种典型网络环境下较快地以自稳定的方式实现故障恢复;在无故障(或故障间

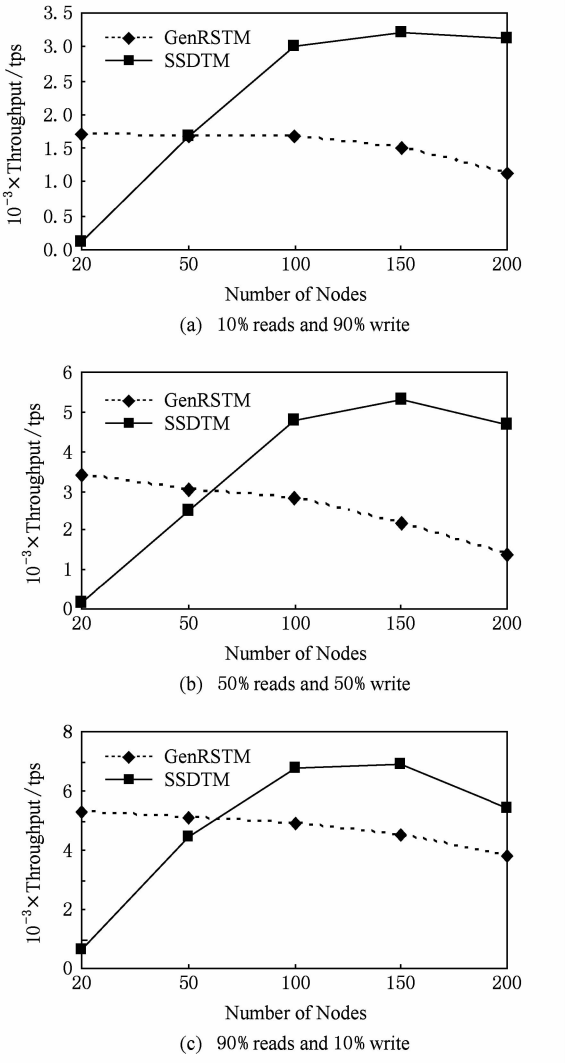

Fig. 5　Comparison of throughput.

图 5　吞吐量比较

隙期)情况下,SSDTM 较于经典的基于原子广播技术的分布式事务内存系统也有很大的性能优势.

4　相关工作

自稳定思想是 Dijkstra 在 1974 年首次提出的[14],自稳定概念的原始应用范围非常窄,但随着应用研究的深入,自稳定已应用于许多领域并且其研究领域也在不断扩大,应用于中央和分布式守护程序、一致和非一致网络的算法已被开发出来;利用自稳定对传统通信协议(例如滑动窗口协议、双路握手协议和交替位协议等)进行扩展是另一个重要研究领域.自稳定系统的主要缺点是非法的初始配置,系统必须很快收敛以减少非法配置的严重性,因为如果系统不能容忍这段未知的初始时间,那么自稳定将无济于事.目前这些相关研究工作集中的领域是互斥,因为原始自稳定模型中定义了一个合法状态,在这种状态下,系统仅存在一个特权进程.将自稳定与分布式编程相结合的研究尚未出现,本文基于现有工作基础并针对事务内存模型,综合考虑了系统鲁棒性和平台易用性,采用自稳定生成树构建策略和分布式数据定位算法实现了高效稳定的自稳定平台.

解决共享内存并发访问的传统方法是"锁"[15],但由于容易引起死锁、护航、优先级反转等问题,使其开销高且不易于使用.作为锁的替代方案,Herlihy 等人提出的事务内存 TM[16]技术是一种更加安全易用的编程模型,很多研究者针对该题目作了大量广泛的研究工作,提出了各种基于软件、硬件以及混合的事务内存系统的设计实现方案.彭林等人对事务内存系统的起源、概念、编程接口和执行模型进行了分析比较,对事务内存系统所涉及的主要内容进行了综述[17].

与事务内存系统类似,分布式事务内存 DTM 是一种替代基于锁的分布式并发控制的解决方案,由于其在分布式系统中的应用潜力,同样受到了众多研究者的关注.Romano 等人描述了一个面向 Web 服务的 DTM 架构,通过在分布式系统节点间复制应用程序状态来保证原子性、隔离性和一致性[18].根据内存事务和数据对象流动方式的不同,可将 DTM 分成"控制流和"数据流"两类,控制流 DTM 使用相关调用机制(例如 RMI)完成对远程对象的访问;数据流 DTM 则允许数据对象在内存事务中移动,本文模型选用了数据流方式,并通过 BDOL 算法来保证数据一致性.

受到 Pedone 等人提出的基于状态机的数据库副本管理策略[19]的启发,Couceiro 等人提出了一个名为 D^{2}STM 的模型[20],将事务复制到不同节点中,并通过一种提交时分布式认证策略来保证事务的强一致性.在数据对象副本数小于 8 个的情况下,D^{2}STM 表现出良好的系统性能,但由于其采用原子广播技术进行并发控制,严重影响了系统可扩展性.本文提出的 LCTC 算法则是一种完全分布式的对象级别的并发控制策略,无需中央控制节点和全局时钟,弃用了原子广播从而减少了通信流量.

此外,宋伟等人提出的 FRTR 故障恢复方法解决了传统容错技术不能针对性的利用事务存储系统自身的容错特性,并利用事务内存系统的版本管理机制避免了额外的检查点保存开销,实现了容错事

务内存系统高效的故障恢复.但对于在分布式系统中的扩展及应用方法未作描述[21].

上述这些相关工作都没有关注分布式事务内存系统的自稳定性以及通过自稳定实现系统鲁棒性的方法.本文挖掘了分布式事务内存系统的自稳定性,提出了基于自稳定平台和分层技术的分布式事务内存系统,并证明了相关算法对于系统自稳定的正确性.

5 结束语

本文通过对分布式系统的自稳定性进行综合分析,并将其作为容错性的一个解法,考虑了系统鲁棒性和可扩展性,分别设计了一个快速收敛的分布式生成树构建策略和一个分布式数据对象定位及并发访问控制算法,从而建立了一个高效的分布式数据访问控制平台,并运用相关知识证明了平台具备自稳定性.以此为基础提出了基于数据流动型的分布式事务内存模型.为了求解该模型,结合事务内存并发控制问题的具体特点设计了基于逻辑时钟的事务控制算法,通过利用本地信息获取内存事务间的依赖关系来降低系统整体管理成本和通信负载,最终提高执行效率.

为验证本文提出算法的性能和新颖性,在SimJava模拟环境和不同的实验参数下进行了大量的仿真实验,其结果表明,当存在瞬时故障时,系统能够以较快的收敛速度和执行效率达到自稳定.与同类方法相比,本文的SSDTM模型具有更高的系统吞吐量和容错性.下一步工作将进一步考虑事务冲突预化解率、事务调度长度等多性能指标,以及在其他故障模型下的系统自稳定性及相关算法研究工作.

参 考 文 献

[1] Harris T, Cristal A, Unsal O S, et al. Transactional memory: An overview [J]. IEEE Micro, 2007, 27(3): 8-29

[2] Larus J, Kozyrakis C. Transactional memory [J]. Communications of the ACM, 2008, 51(7): 80-88

[3] Wikipedia. Well-founded relation [EB/OL]. [2012-11-15]. http://en.wikipedia.org/wiki/Well-founded_relation

[4] Guerraoui R, Kapalka M. On the correctness of transactional memory [C] //Proc of the 13th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2008: 175-184

[5] Tel G. Introduction to Distributed Algorithms [M]. Cambridge: Cambridge University Press, 1994: 39-98

[6] Kotselidis C, et al. DiSTM: A software transactional memory framework for clusters [C] //Proc of the 37th Int Conf on Parallel Processing. Piscataway, NJ: IEEE, 2008: 51-58

[7] Manassiev K, Mihailescu M, Amza C. Exploiting distributed version concurrency in a transactional memory cluster [C] //Proc of the 11th ACM SIGPLAN Symp on Principles and Practice of Parallel Programming. New York: ACM, 2006: 198-208

[8] Raynal M. About logical clocks for distributed systems [J]. ACM SIGOPS Operating Systems Review, 1992, 26(1): 41-48

[9] Lamport L. Time, clocks, and the ordering of events in a distributed system [J]. Communications of the ACM, 1978, 21(7): 558-565

[10] Sundararaman B, Buy U, Kshemkalyani A D, et al. Clock synchronization for wireless sensor networks: A survey [J]. Ad Hoc Networks, 2005, 3(3): 281-323

[11] University of Edinburgh, Institute for Computing Systems Architecture. SimJava [EB/OL]. [2011-11-15]. http://www.dcs.ed.ac.uk/home/hase/simjava

[12] Carvalho N, et al. A generic framework for replicated software transactional memories [C] //Proc of the 10th IEEE Int Symp on Networking Computing and Applications. Piscataway, NJ: IEEE, 2011: 271-274

[13] Minh C C, Chung J, Kozyrakis C, et al. STAMP: Stanford transactional applications for multi-processing [C] //Proc of the IEEE Int Symp on Workload Characterication. Piscataway, NJ: IEEE, 2008: 35-46

[14] Dijkstra E W. Self-stabilizing systems in spite of distributed control [J]. Communications of the ACM, 1974, 17(11): 643-644

[15] Johnson T. Characterizing the performance of algorithms for lock-Free objects [J]. IEEE Trans on Compute, 1995, 44(10): 1194-1207

[16] Herlihy M, Eliot J, Moss B. Transactional memory: Architectural support for lock-free data structures [J]. SIGARCH Computer Architecture News, 1993, 21(2): 289-300

[17] Peng Lin, Xie Lunguo, Zhang Xiaoqiang. Transactional memory system [J]. Journal of Computer Research and Development, 2009, 46(8): 1386-1398 (in Chinese)
(彭林, 谢伦国, 张小强. 事务存储系统[J]. 计算机研究与发展, 2009, 46(8): 1386-1398)

[18] Romano P, Carvalho N, Couceiro M, et al. Towards the integration of distributed transactional memories in application servers' clusters [C] //Proc of the 6th Int ICST Conf on Heterogeneous Networking for Quality, Reliability, Security and Robustness. Berlin: Springer, 2009: 755-769

[19] Pedone F, Guerraoui R, Schiper A, et al. The database state machine approach [J]. Distributed Parallel Databases, 2003, 14(1): 71-98

[20] Couceiro M, Romano P, Carvalho N, et al. D²STM: Dependable distributed software transactional memory [C] // Proc of the 15th IEEE Pacific Rim Int Symp on Dependable Computing. Piscataway, NJ: IEEE, 2009: 307-313

[21] Song Wei, Yang Xuejun. Fault recovery based on transaction rollback in transactional memory [J]. Journal of Software, 2011, 22(9): 2248-2262 (in Chinese)
(宋伟, 杨学军. 基于事务回退的事务存储系统的故障恢复 [J]. 软件学报, 2011, 22(9): 2248-2262)

Lin Fei, born in 1977. Associate professor in software engineering. Member of China Computer Federation. Her main research interests include distributed processing, cloud computing.

Sun Yong, born in 1977. Associate professor. His main research interests include distributed processing, cloud computing, and software transactional memory (sy@zjvtit.edu.cn).

Ding Hong, born in 1963. Professor. His main research interests include network and information security, intelligent information system (dinghong@hdu.edu.cn).

Ren Yizhi, born in 1981. PhD. Member of China Computer Federation. His main research interests include network security, social computing and evolutionary game theory (renyz@hdu.edu.cn).