ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.20, No.3, March 2009, pp.744−753
doi: 10.3724/SP.J.1001.2009.03475

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种基于补偿代价的长事务调度算法*

朱 锐[1+], 郭长国[1], 王怀民[1,2]

[1](国防科学技术大学 计算机学院,湖南 长沙 410073)

[2](国防科学技术大学 并行与分布处理国家重点实验室,湖南 长沙 410073)

A Scheduling Algorithm for Long Duration Transaction Based on Cost of Compensation

ZHU Rui[1+], GUO Chang-Guo[1], WANG Huai-Min[1,2]

[1](School of Computer, National University of Defense Technology, Changsha 410073, China)

[2](National Laboratory for Parallel and Distributed Processing, National University of Defense Technology, Changsha 410073, China)

+ Corresponding author: E-mail: mruzcce@gmail.com

Zhu R, Guo CG, Wang HM. A scheduling algorithm for long duration transaction based on cost of compensation. *Journal of Software*, 2009,20(3):744−753. http://www.jos.org.cn/1000-9825/3475.htm

Abstract: Transaction of service composition has long-lived feature which a global-transaction is divided into several distributed sub-transactions. Atomicity property is preserved by using compensating transactions, which semantically undo the effects of the completed sub-transactions, in case of global-transaction abort. However, the cost of compensation may be expensive and methods may be complex. To overcome this limitation, a novel scheduling algorithm named STCD (SubTransaction committing delay) is presented based on analysis of compensation. Different from traditional methods, sub-transactions determine the time of committing according to both the cost of compensation and the state of execution dynamically. The correctness of proposed algorithm is proved. Simulations show that STCD algorithm can confine the compensation sphere and reduce the cost of compensation.

Key words: long-lived transaction; delay commit; compensating transaction; cost of compensation; service composition

摘 要: 服务组合应用中的事务具有较长的生命周期,一个事务被划分成多个分布的子事务.补偿机制是保证长事务原子性的一种有效方法,允许在语义上逆转一个已提交子事务的结果.然而,补偿的代价可能是巨大的.针对该问题,在分析并定义事务补偿代价的基础上,提出一种子事务延迟提交算法——STCD(subtransactions committing delay)算法,与传统子事务执行后立即提交的方法相比,STCD 算法允许某个全局事务的子事务在提交前根据自身的补偿代价和全局事务的运行状态动态确定提交时间,使可能的补偿操作变更为回滚操作.算法的正确性得以证明.模拟实验结果表明,在事务失败时,STCD 算法可以有效地减少补偿活动的数目,降低补偿代价.

关键词: 长事务;延迟提交;补偿事务;补偿代价;服务组合

* Supported by the National Natural Science Foundation for Distinguished Young Scholars of China under Grant No.60625203 (国家自然科学基金杰出青年基金); the National Basic Research Program of China under Grant Nos.2005CB321800, 2005CB321804 (国家重点基础研究发展计划(973))

Received 2008-02-21; Accepted 2008-10-07

中图法分类号: TP311 文献标识码: A

目前,随着服务成为开放网络环境下资源封装与抽象的核心概念,通过动态地组合服务实现资源的灵活聚合是面向服务计算的核心技术,成为近年来的研究热点.事务机制是保证服务组合可靠性的重要手段,服务组合的复杂性、动态性、长期运行性以及服务之间的异构性使得传统事务的 ACID 特性必须被放松.

传统的事务处理技术主要面向那些处理时间较短(若干毫秒)的事务活动,而服务组合应用中的事务通常具有较长的生命周期,一个全局事务被划分成多个分布的子事务,对于这种长事务,非全则无的原子性需求过于严格,可能造成时间、资源的大量浪费;传统的锁机制可能引发资源被长期占用,从而导致系统的性能降低,在长事务中放松对隔离性的要求,使长事务的中间结果立即对其他事务可见,是防止资源被长时间锁定、提高事务间并发度的一种手段[1].但是,这种可见性可能会破坏数据的完整性和一致性,因此,人们引入了补偿事务(compensating transaction).文献[2]最早提出了补偿的概念,用于维持长期运行的数据库应用的事务特征.补偿不同于传统事务的回滚.回滚是指在产生失效时,将系统的状态或者数据恢复到执行前的状态,主要是通过原来的数据替换执行中产生的结果[3].补偿并非如此,而是执行原操作的逆操作,在语义上逆转另一个已提交事务的结果[3].如果要对事务进行补偿,那么除了对事务本身进行补偿外,还应对所有直接和间接依赖于该事务的其他各个事务也进行补偿,而且通过补偿不一定可以完全消除事务提交带来的影响.

为了减少一个全局事务失败时补偿活动的数目,降低补偿的代价,本文首先对事务补偿活动进行分析,对补偿操作进行划分并定义事务补偿代价,然后,提出了一种子事务延迟提交(subtransaction committing delay,简称 STCD 算法)算法,与传统子事务执行后立即提交的方法相比,STCD 算法允许子事务在提交前根据自身的补偿代价和全局事务的运行状态动态地确定提交时间.

本文第 1 节通过实例分析对要解决的问题进行阐述.第 2 节给出子事务延迟提交算法 STCD 的设计,并证明算法的正确性.第 3 节对 STCD 算法进行实验和评价.第 4 节比较相关研究工作.第 5 节对全文进行总结.

1 问题描述

我们首先列举出一个用户外出旅游的经典例子,然后讨论其可能引起的高额补偿代价问题.为了实现用户外出旅游的目的,中间可能需要实现一些必需的目标,比如:对用户行程进行规划(customer requirements specification,简称 CRS);为了到达另外的城市,需要购买飞机票(flight booking,简称 FB),还需要预订当地宾馆(hotel booking,简称 HB)和为了出行方便租赁当地出租车(taxi booking,简称 TB);在预定完成后,需要进行网上支付(online payment,简称 OP);最后用户可以任意选择一家快递公司把票送至家中(有两家快递公司可供选择:tickets delivery with FedEx,简称 TDFE;tickets delivery with UPS,简称 TDU).图 1 给出了用户外出旅游的服务组合示意图.

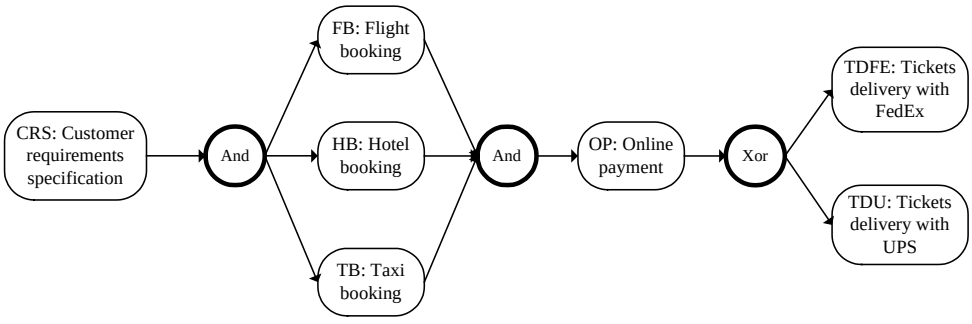

Fig.1 Composite service for online travel arrangement

图 1 用户外出旅行的服务组合

从图 1 可以看出,服务间执行的先后关系已经确定:首先要对用户的行程进行规划(CRS),之后同步进行购买机票(FB)、预订宾馆(HB)和预租出租车(TB)的活动,待 3 个任务都成功完成后,用户可以进行网上支付(OP),支付成功后选择 FedEx 或 UPS 快递公司送票到家.如果其中某项任务不能成功完成,则将取消整个旅行预约,即整个旅游活动作为一个整体完成,具有原子性要求.

假设网上支付服务为不可靠服务,其执行成功的概率为 30%.如前所述,在执行到网上支付时,已成功地完成了购买机票(FB)、预订宾馆(HB)和预租出租车(TB)的任务.那么,网上支付服务执行失败将导致补偿操作的执行,例如:退掉机票、退订宾馆和出租车、交纳补偿金和违约金、个人信誉降低等.本文要解决的问题是,在服务间执行的先后关系已经确定的情况下,找寻减少补偿活动数目、降低补偿活动代价的方法,同时,还要保证事务间的并发度以及防止资源被长期锁定造成的资源浪费.

2 STCD:一种长事务中子事务延迟提交算法

STCD 算法需要用到两张表:事务补偿代价表和全局事务运行状态表.事务补偿代价表用于记录所有事务的补偿代价,全局事务运行状态表则用于记录全局事务执行成功率.下面,首先介绍如何建立事务补偿代价表和全局事务运行状态表,然后详细介绍 STCD 子事务延迟提交算法.

为便于后文的讨论和分析,现定义如下记号和概念:

定义 1(事务 QoS 指标). 关于事务的 QoS 尚未有统一的定义[4],但一般认为,事务 QoS 指标应包括执行费用、执行时间和事务执行成功率等.事务 QoS 指标的定义如下:

(1) 事务 t_i 的执行费用,记为 $q_{pay}(t_i)$.

(2) 事务 t_i 的执行时间,记为 $q_{time}(t_i)$.

(3) 事务 t_i 的执行成功率,记为 $Q_{rel}(t_i)\in[0,1]$.

本文并不讨论事务 QoS 的具体执行机制,如 QoS 规约的执行、协商、反馈等,仅定义基本的 QoS 指标来量化事务的执行代价和补偿代价.

为了统一不同质量度量的量纲,在计算事务执行代价之前需要对事务的质量向量作标准化处理,如下式所示:

$$r_x(t_i)=\frac{q_x(t_i)-q_x^{\min}}{q_x^{\max}-q_x^{\min}}(x\in\{pay,time\}),$$

其中 $q_x^{\max}$, $q_x^{\min}$ 分别为在所有事务中度量 x(x 为执行费用或者执行时间)上的最大值和最小值.根据上式得到的标准化质量向量具有如下特点:$r_x(t_i)\in[0,1]$且 $r_x(t_i)$的值越小意味着事务 t_i 在度量 x 上越优.基于上述质量向量标准化方法,基于用户偏好的事务执行代价定义如下.

定义 2(事务执行代价). 设 $w^{(p)},w^{(t)}$分别为执行费用、执行时间的偏好权值,$0<w^{(p)},w^{(t)}<1$ 且 $w^{(p)}+w^{(t)}=1$,由用户指定.事务 t_i 执行代价 $Cost_{exec}(t_i)=w^{(p)}\times r_{pay}(t_i)+w^{(t)}\times r_{time}(t_i)$.

定义 3(事务无影响). 设$\delta=\langle t_1,t_2,\ldots,t_k\rangle$是一个事务执行序列,如果对任意的事务执行序列 u 和 v,$\langle u,v\rangle$与$\langle u,\delta,v\rangle$的执行效果相同,则称事务执行序列$\delta$是无影响的.

2.1 事务补偿代价

如果某个全局事务的执行成功率很低,应延后提交补偿代价较高的子事务,从而在该全局事务失败时,使尽可能多的补偿操作变更为回滚操作.这就要求我们准确计算出已执行事务的补偿代价,文献[5]对补偿操作进行了划分,本文以此为基础并进行扩充.下面,首先对补偿操作进行划分,然后根据不同的补偿操作类型定义补偿操作的代价.

本文将补偿操作划分为如下 5 类:

(1) 不需要补偿(needless compensable,简称 NLC):对于事务 t_i,若$\delta=\langle t_i\rangle$无影响,则事务 t_i 是不需要补偿的,记作 $comp_{type}(t_i)$=NLC.

(2) 完全可补偿(fully compensable,简称 FUC):对于事务 t_i,若存在事务 t_j,使得 $\delta=\langle t_i,t_j\rangle$ 无影响且 $Cost_{exec}(t_i)=Cost_{exec}(t_j)$,则事务 t_i 是完全可补偿的,记作 $comp_{type}(t_i)$=FUC.t_j 称为 t_i 的完全可补偿事务,记作 t_i^{-1}.

(3) 有条件可补偿(conditionally compensable,简称 CDC):对于事务 t_i,若存在事务 t_j,使得 $\delta=\langle t_i,t_j\rangle\wedge cond_j$ ($cond_j\neq\{\lambda\}$)无影响,其中 $cond_j$ 是为进行补偿所附加的条件,则事务 t_i 是有条件可补偿的,记作 $comp_{type}(t_i)$=CDC.如果 $cond_j=\{\lambda\}$,则事务 t_i 是完全可补偿的.以用户购买电脑为例:用户完成付款收到电脑后,决定退掉电脑,此时,电脑卖主不退款而是要求用户选择另外一种同等价值的商品.因此,对于有条件可补偿操作来说,服务提供者必须明确 3 点补偿条件[5]:

a) 补偿有效期限(valid period):补偿操作必须在一定时间内完成,记作 $cond_{time}(t_i)$.通常,$cond_{time}(t_i)\geq q_{time}(t_i)$.例如,从退掉电脑后一个月内.

b) 补偿规则(valid against):补偿操作的规则,一般认为,补偿规则主要是指用来替换的项目的执行费用,记为 $cond_{pay}(t_i)$.通常 $cond_{pay}(t_i)\geq q_{pay}(t_i)$.例如,用户选择另外商品的价值不能小于电脑的价值.

c) 补偿有效地点(valid at):补偿操作在指定的地点执行才是有效的,记作 $cond_{location}(t_i)$.例如,必须在原电脑提供商处购买其他商品.

(4) 部分可补偿(partially compensable,简称 PAC):对于事务 t_i,若存在事务 t_j,使得 $\delta=\langle t_i,t_j,t_{addition}^{-1}\rangle$ 无影响,其中 $t_{addition}$ 是额外的补偿事务且 $Cost_{exec}(t_i)=Cost_{exec}(t_j)$,则事务 t_i 是部分可补偿的,记作 $comp_{type}(t_i)$=PAC.

(5) 不可补偿(non compensable,简称 NOC):对于事务 t_i,若不存在事务 t_j,使得 $\delta=\langle t_i,t_j\rangle\wedge cond_j$ 无影响,则事务 t_i 是不可补偿的,记作 $comp_{type}(t_i)$=NOC.

定义 3(事务补偿代价). 一个事务 t_i 的补偿代价记为 $Cost_{comp}(t_i)\in[0,2]$:

$$Cost_{comp}(t_i)=\begin{cases}0, & comp_{type}(t_i)=\text{NLC}\\ Cost_{exec}(t_i), & comp_{type}(t_i)=\text{FUC}\\ Cost_{exec}(t_i)\times(1+\alpha), & comp_{type}(t_i)=\text{CDC},\\ Cost_{exec}(t_i)+Cost_{exec}(t_{addition}), & comp_{type}(t_i)=\text{PAC}\\ 2, & comp_{type}(t_i)=\text{NOC}\end{cases}$$

其中,$a=w^{(t)}\times\left(1-\dfrac{cond_{time}(t_i)-q_{time}(t_i)}{cond_{time}(t_i)}\right)+w^{(p)}\times\dfrac{cond_{pay}(t_i)-q_{pay}(t_i)}{cond_{pay}(t_i)}$ 为条件因子(condition factor),是补偿条件引起的补偿代价的增值.记 $m=\dfrac{cond_{time}(t_i)-q_{time}(t_i)}{cond_{time}(t_i)}\in[0,1]$,$n=\dfrac{cond_{pay}(t_i)-q_{pay}(t_i)}{cond_{pay}(t_i)}\in[0,1]$.从条件因子可以看出,若 $cond_{time}(t_i)\to\infty$,则 m=1,那么 $w^{(t)}\times(1-m)$=0;若 $cond_{time}(t_i)=q_{time}(t_i)$,则 m=0,那么 $w^{(t)}\times(1-m)=w^{(t)}$.说明 $cond_{time}(t_i)$越大,则补偿的代价越小,反之亦然.若 $cond_{pay}(t_i)\to\infty$,则 n=1,那么 $w^{(p)}\times n=w^{(p)}$;若 $cond_{pay}(t_i)=q_{pay}(t_i)$,则 n=0,那么 $w^{(p)}\times n$=0.说明 $cond_{pay}(t_i)$越小,补偿的代价越小,反之亦然.

至此,可以根据每个事务所属的补偿操作类型和相应的事务 QoS 建立起事务补偿代价表.但是,在实际的应用中,本文所提出的补偿操作划分并不一定可以准确地覆盖所有补偿操作.如何更为精确地对补偿操作进行划分,进而更为准确地刻画补偿事务的执行代价是我们将来要进一步研究的问题.

2.2 全局事务运行状态

服务组合可以集成现有的简单的服务,按照顺序、并行、选择和循环等组合模式形成复杂服务,快速而又灵活地构建功能强大的新应用.QoS 度量的内在物理含义决定了它们在不同的组合模式下具有不同的叠加意义,即 QoS 度量聚合规则具有组合模式相关性.表 1 给出了事务执行成功率关于顺序、并行、选择和循环 4 种组合模式下的聚合规则(单个事务被看作一种特殊的顺序模式),其中 $t_1,\ldots,t_n$ 分别为某个全局事务 T 的子事务,$p(t_i)$表示在选择模式中子事务 t_i 的相对执行率,$n(t_i)$表示在循环模式中子事务 t_i 循环执行的次数.

Table 1 Rule of QoS aggregation
表 1 质量聚合规则

Composition pattern	Rule of QoS aggregation
Sequence (SEQ)	$Q_{rel}(T)=\prod_1^n Q_{rel}(t_i)$
Parallel (PAR)	$Q_{rel}(T)=\prod_1^n Q_{rel}(t_i)$
Option (OPT)	$Q_{rel}(T)=\sum_1^n (p(t_i)\times Q_{rel}(t_i))$
Cycle (CYC)	$Q_{rel}(t_i)=Q_{rel}(t_i)^{n(ti)}, i\in\{1,2,\ldots,n\}$

从模式集成的角度来看,一个全局事务是由其子事务通过不同的组合模式按照顺序结构相连接而形成的,不妨将全局事务 T 表示为 $T=\langle T_1,T_2,\ldots,T_m\rangle$为事务执行序列,其中 m 是该全局事务 T 中组合模式的个数,$T_1,T_2,\ldots,T_m$ 是顺序、并行、选择和循环 4 种组合模式中的一种,设 $Pat(T_i)\in\{SEQ,PAR,OPT,CYC\}$,为事务所属的组合模式类型.

下面,基于这样的前提给出全局事务执行成功率(success ration)的定义,并引入全局事务相对执行成功率(comparative ration)的概念以支持 STCD 算法.

定义 4(全局事务执行成功率). 设全局事务 $T=\langle T_1,T_2,\ldots,T_m\rangle$为一个事务执行序列,并且 $Pat(T_i)\in\{SEQ,PAR,OPT,CYC\}$,则全局事务执行成功率 $Q_{rel}(T)=\prod_1^m Q_{rel}(T_i)$,且 $Q_{rel}(T)\in[0,1]$.若 $T=\varnothing$,则 $Q_{rel}(T)=1$.

全局事务执行成功率(success ratio)表示全局事务最终成功执行的可能性,即反映了全局事务的静态特征;而全局事务相对执行成功率(comparative ratio)表达的是全局事务在执行过程中其未完成的事务执行序列 $T'=\langle T_i,T_{i+1},\ldots,T_m\rangle$的执行成功率,反映了全局事务的动态特征.通过全局事务执行成功率的定义可知 $Q_{rel}(T)\leq Q_{rel}(T')$,且 $Q_{rel}(T')=\prod_i^m Q_{rel}(T_i)$随着全局事务的运行是递增的.

2.3 STCD子事务延迟提交算法

在长事务中,存在事务的执行流(execution flow)和提交流(commit flow),那么,各个事务之间存在着执行依赖性和提交依赖性[6],下面给出一些相关的定义.

定义 5(执行依赖性). 事务 t_i 执行依赖于 t_j,记为 $depExec(t_i,t_j)=t_j.complete()\rightarrow t_i.activate()$.所有执行依赖于事务 t 的事务组成的集合称为事务 t 的执行依赖集(execution dependency set),记为 $EDS(t)=\{t_i|depExec(t_i,t)\}$.

事务之间的执行依赖性描述了事务执行的先后关系.如果事务 t_i 执行依赖于 t_j,那么 t_i 只有在 t_j 执行完成后(不一定提交)才能被激活.以图 1 为例,定义了 FB,HB,TB 与 OP 的执行依赖关系,只有当完成购买机票、预订宾馆和预租出租车后才能激活网上的付款事务,也就是说,$depExec(OP,FB\wedge HB\wedge TB)$,且 $EDS(OP)=\{FB,HB,TB\}$.

定义 6(提交依赖性). 事务 t_i 提交依赖于 t_j,记为 $depCommit(t_i,t_j)=t_j.commit()\rightarrow t_i.commit()$.所有提交依赖于事务 t 的事务组成的集合称为事务 t 的提交依赖集(commit dependency set),记为 $CDS(t)=\{t_i|depCommit(t_i,t)\}$.

定义 7(事务补偿风险). 令 CR_t 为事务 t 的补偿风险,设 CR_0 为全局事务中允许执行完毕后的事务进行提交的阈值,如果事务 t 的 $CR_t\leq CR_0$,那么该事务是允许提交的.

事务之间的提交依赖性定义了事务提交的先后关系,也可认为是定义了事务之间的提交约束关系.但通常认为,事务之间是不存在提交约束关系的,只要一个事务的补偿风险低于风险阈值就可以提交.

子事务补偿代价和全局事务相对执行成功率为确定子事务的补偿风险(compensation risk)提供了依据.下面,我们提出一种长事务中子事务延迟提交算法——STCD 算法.该算法的主要思想是在子事务执行完成后,根据子事务的补偿代价和全局事务相对执行成功率采用数值计算来度量子事务的补偿风险,当前补偿风险低于 CR_0 的子事务允许提交,反之,子事务被延迟提交,直到子事务的补偿风险低于 CR_0 为止,是一种顺序执行、乱序提交的过程.

完整的 STCD 子事务延迟提交算法的伪码描述如下,其中,$t_1,\ldots,t_n$ 为某个全局事务 T 的子事务,$T=\langle T_1,T_2,\ldots,T_m\rangle$为全局事务执行序列,$m$ 是 T 中组合模式的个数,$T_1,T_2,\ldots,T_m$ 是顺序、并行、选择和循环 4 种组合模式中的一种.T'是已执行完成但未提交的子事务集合,T''是已经提交的子事务集合.

1. 计算每一个子事务的补偿代价

Forall $t_i\in\{t_1,\ldots,t_n\}$ **Do**

$Cost_{comp}(t_i)=get\_comp\_cost(t_i)$; //计算每一个子事务的补偿代价

2. 计算全局事务执行成功率,并限定风险阈值的最大值

$Global\_trans\_reli=get\_trans\_reli(T)$; //计算全局事务执行成功率

$CR_{\max}=(1-Global\_trans\_reli)\times\max(Cost_{comp}(t_i))$; //定义风险阈值的最大值

3. 计算已执行完子事务的补偿风险 CR_t,补偿风险低于风险阈值 CR_0 的子事务被提交,高于风险阈值的子事务延后提交

$T'=\varnothing$; $T''=\varnothing$;

While ($T\neq\varnothing||T'\neq\varnothing$)

Forall $t_i\in T$ **Do**

If $EDS(t_i)\subseteq T'$ **Then**

Begin

执行子事务 t_i,且把 t_i 加入到 T' 中,并把 t_i 从 T 中删除;

$Compar\_trans\_reli=get\_trans\_reli(T-T')$; //计算全局事务相对执行成功率

$CR_{ti}=(1-Compar\_trans\_reli)\times Cost_{comp}(t_i)$; //计算 t_i 的补偿风险

选取风险阈值 CR_0,$CR_0\in[0,CR_{\max}]$;

If ($CR_{ti}\leq CR_0$ && $CDS(t_i)\subseteq T''$) **Then** //若子事务补偿风险低于阈值,且满足约束,则提交子事务 t_i,且把 t_i 加入到 T'' 中,并把 t_i 从 T' 中删除

End

一旦有子事务执行完成,那么全局事务相对执行成功率也会相应地发生变化,导致子事务的补偿风险也在不停地变化,整个系统呈一种动态性.风险阈值可以根据系统的性能灵活地设定,如果事务的并发度和吞吐率低于预期的指标,则可以提高阈值,加快子事务提交的速度,但是当发生失效时,用户可能要付出更多的补偿代价;如果用户注重降低系统的补偿活动数目和补偿的代价,可以降低阈值,意味着子事务提交的约束更难满足.因此,通过合理地选取阈值,可以在保证尽量不影响全局事务执行效率的前提下兼具较低的补偿活动数目和补偿代价.

2.4 STCD算法的正确性

证明 STCD 算法的正确性,即要证明它能够保证所有子事务最后都能提交.

定理 1. STCD 算法产生的调度可以保证所有子事务最终都能提交.

证明:在不考虑风险阈值的情况下,假设存在子事务 $t_i\in T'$ 执行后没有被提交,那么必然有$\{t_j\}\prec T''$,其中$\{t_j\}\subseteq CDS(t_i)$;由于$\{t_j\}\prec T''$,那么 $t_j\in T'$,如此循环下去,将推导出所有子事务都将是未提交的,这与实际情况不符,因为总存在一个子事务不提交依赖于任何子事务,矛盾.

在不考虑提交依赖性的情况下,假设存在子事务 $t\in T'$ 最终没有被提交,也就是说,此时 $CR_t>CR_0$.当前 $T=\varnothing$,由定义 4 可知,全局事务相对执行成功率为 1,那么 $CR_t=(1-Compar_trans_reli)\times Cost_{comp}(t_i)$为 0.同时,$CR_0\in[0,CR_{\max}]$大于 0,所以 $CR_t\leq CR_0$,与假设矛盾.

故证明了 STCD 算法能够保证所有子事务都被提交.证毕. □

3 性能评价

3.1 实验的建立

本节通过模拟,对 STCD 与现有的子事务执行完成后立即提交(STC)两种调度算法进行比较,同时探讨了STCD算法中风险阈值的选取对调度结果的影响.我们将一个长事务中子事务的总数定为10~90个,实验基于随机产生的场景,每个场景中的服务组合执行流程结构和子事务提交约束均随机生成,此外,子事务的补偿代价以

及执行成功率分别随机分布在[0,2]和[0.9,1.0]内.实验中,我们认为在子事务执行成功后不一定保证可以成功提交,所以我们假设子事务的提交成功率分布在[0.95,1.0]内,且子事务越晚提交,则提交成功率越低.

实验环境为 PC 机,硬件为 Pentium4 2.0G、内存 1 024M RAM;软件环境为 Windows 2000 Professional Service Pack 4.我们采用如下两项评价指标:

① 长事务失败补偿代价:是指对于每个失败的长事务,应该对多少个子事务进行补偿,总的补偿代价为多少.该指标反映了系统在长事务恢复机制方面的效率.

② 长事务完成时间:是指所有子事务成功完成的时间.该指标反映了并发处理长事务的能力.

3.2 调度算法对比

我们模拟了不同子事务数目的并发执行,并记录长事务的完成时间以及失效时的补偿代价.该实验用于测试算法的有效性.实验场景的基本子事务数目为 10,并以步长 10 递增到 80,风险阈值为 0.3.实验随机产生 1 000 个服务组合场景,对每个场景分别运行子事务立即提交算法(STC)和 STCD 算法.实验结果如图 2 和图 3 所示.

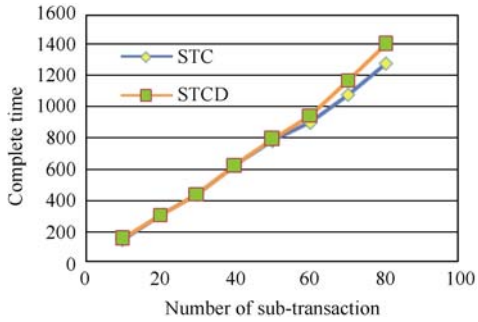

Fig.2 Complete time comparison
图 2 事务完成时间对比

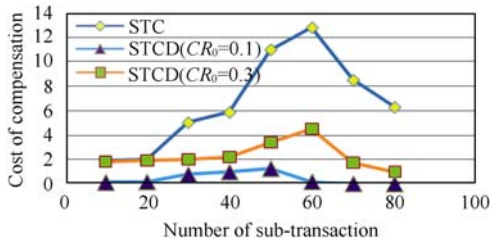

Fig.3 Compensation costs comparison
图 3 补偿代价对比

从图 2 可以看出,随着子事务数目的增加,两种算法在全局事务完成时间上的差别逐渐增大.主要由于随着子事务数量的增加,全局事务相对执行成功率将降低,从而导致更多的子事务被延迟提交,这与算法的实际情况相吻合.

从图 3 中的数据分析可知:STCD 算法在降低失效时补偿代价有着明显的优势,且这种效果随着 CR_0 的减小明显增加.例如:当 CR_0=0.3,子事务数量为 60 时,算法能够减少约 65%的补偿代价;当 CR_0 减小到 0.1 时,约能减少 98%的补偿代价.但是必须注意到,当子事务数量处于 10~20 之间、CR_0=0.3 时,算法的效果并不明显,因此有必要研究风险阈值取值大小对算法的影响.

3.3 风险阈值调整

图 3 显示,风险阈值 CR_0 的选取会显著影响算法的效果.下面的实验用于测试 STCD 算法在不同 CR_0 下的效果.图 4 给出了子事务数量 n 从 20~80 的全局事务在风险阈值 CR_0 从 0~1 的补偿代价的曲线图.

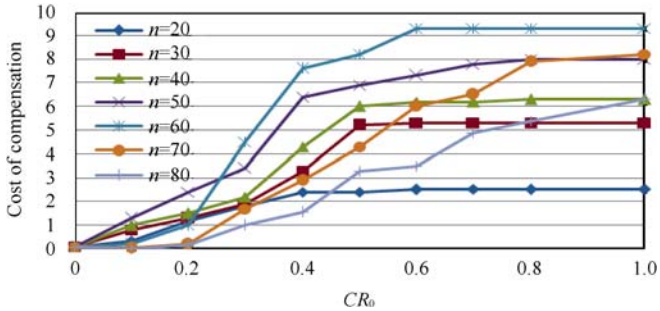

Fig.4 Different CR_0 comparsion
图 4 不同风险阈值的对比

从图中可以很明显地看出,不同 n 值的曲线形态十分类似,这说明随着风险阈值的增大,全局事务的补偿代价也呈增加趋势.全局事务补偿代价的多少与 CR_0 的大小相关.从图上还能看出,当 CR_0 超过一定值时,例如:当 CR_0=0.7 时,增加 CR_0 而带来的降低补偿代价的效果并不是总那么明显.这是因为,CR_0 已经接近了风险阈值的最大值 CR_{max}.同理,当子事务数量较少,例如 n<20 时,CR_{max} 的值也较小,导致 CR_0 的选取对补偿代价的影响也较小,因此,当我们处理子事务数量较少的全局事务时,为达到一定的补偿效果,必须选取较小的 CR_0 以达到较好的算法效果.上述模拟实验说明,在实际应用中,应根据子事务数量和应用场景的不同将 CR_0 设置为一个恰当的大小以获得良好的效果,同时将全局事务的延迟控制在较低的水平.

4 相关工作

补偿的概念最早出现在 Saga 模型[2]中,长事务中每个子事务都被提供一个属于它自己的补偿块(compensation handler),如果发生失效,失败的那个子事务按照常规的方法进行回滚,然后,Saga 执行每一个先前已经完成了的那些子事务的补偿块,它按照子事务被提交的逆序调用对应的补偿块.文献[7,8]扩展了事务补偿的概念,以期为企业事务(business transactions)提供更为强大、安全和灵活的建模机制.文献[9]刻画了补偿操作之间的依赖关系,把依赖类型分为需要(requirement)、排他(exclusive)和隐藏(hint).此外,在文献[10−12]中对长事务的补偿技术也进行了相关的研究.在上述研究工作中,虽然都对补偿操作进行了深入的探讨,但补偿操作代价的刻画是二值的,即补偿与不补偿.而在互联网环境中,补偿操作的代价由于个体差异而有着很大不同,二值的补偿操作代价不利于优化调度算法,另外,也不能刻画补偿操作代价随时间的变化.文献[5]认为,刻画补偿操作的代价可以优化补偿操作的选择,但没有就补偿操作的分类和代价进行精确的定义,缺乏形式化的描述.

由于平坦事务模型不能够很好地满足长事务的应用需求,研究人员考虑到具体应用语义和依赖关系提出了各种扩展事务模型(extended transaction model,简称 ETM),采用补偿和等价替换等手段,适当调整并放松事务的 ACID 要求.常见的 ETM 包括嵌套(nested)事务模型、Sagas[2]、分支/汇合(split-join)事务模型、Flexible 事务模型和 ACTA 模型等.以上这些扩展事务模型为了尽早释放所占用资源,放松了对隔离性的要求,在子事务完成后立即或者及早提交,提高了事务间的并发度.但这些模型均未考虑子事务的补偿代价,并且在子事务提交前未对全局事务的运行状态进行评估,很可能产生子事务刚提交就要被补偿的情况,导致不必要的开销.

对事务工作流管理系统而言,补偿是一种重要的失效恢复手段.ConTracts,FlowMark,WAMO,OPERA,WIDE 等事务工作流系统借鉴 ETM 思想,采用基于平坦流图的事务工作流失效恢复机制,即采用部分补偿或者完全补偿的方法静态确定补偿的范围[13].完全补偿的基本思想是从失效节点开始,逆向补偿回滚至全局事务的起始点,即在产生失效时,需要补偿所有已经提交的子事务.部分补偿则是逆向回溯至预先定义的一个一致点(safepoint),即根据需要部分撤销执行结果,意味着只补偿部分已提交子事务.部分补偿、完全补偿的思想与 STCD 算法最主要的区别在于:STCD 算法在尽量不影响全局事务执行效率的基础性上,延迟子事务提交的时间,使可能的补偿操作变更为回滚操作,是一种主动防御的思想,而部分补偿与完全补偿是一种被动防御的思想,无法从根本上减少补偿活动的数量和降低补偿代价.

资源的准占用模,以 THP(tentative hold protocol)[14]为代表的重大革新,通过一种尝试性的、非阻塞的方式预定事务资源,通过由传统的资源完全锁定方式转变为共享锁定方式,放松对隔离性的要求.该协议允许多个事务参与者同时持有同一个资源,如果其中一个服务修改了资源,那么其他持有该资源的服务都将收到消息告知所持有的资源不再是有效的,从而将补偿的可能性降到最低[15].文献[16]通过修改 THP 协议中两个重要参数:资源预定上限(overhold size)和资源预定时效(hold duration)来优化 THP 性能.资源的准占用模式虽然可以在一定程度上降低补偿活动的数目,但是并未考虑补偿活动本身的代价.

5 结束语

本文研究在服务组合应用中如何解决长事务失效时补偿代价过高的问题.我们面临两个问题:(1) 如何对事务的补偿代价进行刻画;(2) 怎样针对事务的补偿代价为全局事务提供一种高效的调度算法.

本文的贡献是针对上述两个问题的解决:首先对补偿操作进行分类,把补偿操作分为 5 类;其次,针对每类补偿操作,基于用户的偏好定义事务补偿操作的代价;最后,提出 STCD 子事务延迟提交算法,算法的主要思想是一个全局事务中一旦某个子事务完成,就根据该子事务的补偿代价和全局事务相对执行成功率采用数值计算来度量子事务的补偿风险,当前补偿风险低于风险阈值的子事务允许提交,反之,子事务被延迟提交,直到子事务的补偿风险低于风险阈值为止,是一种顺序执行、乱序提交的过程.我们证明了调度算法是正确的.同时,模拟实验结果表明,STCD 算法与子事务执行完立即提交算法相比可以显著降低全局事务的补偿代价.

References:

[1] Bocchi L, Laneve C, Zavattaro G. A calculus for long-running transactions. In: Proc. of the 6th IFIP Int'l Conf. on Formal Methods for Open-Object Based Distributed Systems. 2003. 124−138. http://www.springerlink.com/content/d5tkkhdegw62m9g0/

[2] Garcia-Molina H, Salem K. Sagas. ACM SIGMOD Record, 1987,16(3):249−259.

[3] Gray J, Reuter A. Transaction Processing: Concepts and Techniques. San Francisco: Morgan Kaufmann Publishers, 1993.

[4] Xie WX, Navathe SB, Prasad SK. Supporting QoS-aware transaction in system on a mobile device (SyD). In: Proc. of the 23rd Int'l Conf. on Distributed Computing Systems Workshops (ICDCSW2003). Providence: IEEE Computer Society Press, 2003. 498−502.

[5] Biswas D. Compensation in the World of Web services composition. In: Proc. of the Semantic Web Services and Web Process Composition (SWSWPC). LNCS 3387, 2005. 69−80.

[6] Liu YS, Liao GQ, Li GH, Xia JL. Commitment of mobile distributed real-time nested transaction. Journal of Software, 2003,14(1):139−145 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/14/139.htm

[7] Bruni R, Melgratti H, Montanari U. Theoretical foundations for compensations in flow composition languages. In: Proc. of the 32nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL 2005). ACM Press, 2005. 209−220. http://portal.acm.org/citation.cfm?id=1040305.1040323

[8] Butler M, Chessell M, Ferreira C, Griffin C, Henderson P, Vines D. Extending the concept of transaction compensation. IBM Systems Journal, 2002,41(4):743−758.

[9] Lin LL, Liu FF. Compensation with dependency in Web services composition. In: Proc. of the Next Generation Web Services Practices (NweSP 2005). 2005. 183−188. http://ieeexplore.ieee.org/iel5/10610/33519/01592426.pdf

[10] Strandenas T, Karlsen R. Trans. compensation in Web services. In: Proc. of the NIK 2002, the Norwegian Computer Science Conf., Buskerud College, 2002. http://folk.uio.no/nik/2002/Strandenas.pdf

[11] Sahai A, Ouyang J, Machiraju, V. An approach to optimistic commit and transparent compensation for E-Service transactions. In: Proc. of the 14th Int'l Conf. on Parallel and Distributed Computing Systems (PDCS 2001). 2001. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.7611

[12] Ghafoor MA, Yin JW, Dong JX, Mujeeb-u-Rehman M. π_{RBT}-Calculus compensation and exception handling protocol. In: Proc. of the 14th Euromicro Int'l Conf. on Parallel, Distributed, and Network-Based Processing (PDP 2006). 2006. 39−47. http://www2.computer.org/portal/web/csdl/doi/10.1109/PDP.2006.1

[13] Grefen P, Vonk J, Apers P. Global transaction support for workflow management systems: From formal specification to practical implementation. The VLDB Journal, 2001,10(4):316−333.

[14] Roberts J, Srinivasan K. Tentative Hold Protocol Part 1. White Paper, W3C Note, http://www.w3.org/TR/tenthold-1

[15] Limthanmaphon B, Zhang YC. Web service composition transaction management. In: Proc. of the 15th Australasian database Conf. (ADC 2004). 2004. 171−179. http://crpit.com/confpapers/CRPITV27Limthanmaphon.pdf

[16] Xu W, Cheng WQ, Li B. taTHP: An improved transaction model based on THP. Journal of Chinese Computer Systems, 2007,28(1):97−101 (in Chinese with English abstract).

附中文参考文献:

[6] 刘云生,廖国琼,李国徽,夏家莉.移动分布式实时嵌套事务提交.软件学报,2003,14(1):139−145. http://www.jos.org.cn/1000-9825/14/139.htm

[16] 许炜,程文青,李冰.taTHP:一种 THP 的改进事务模型.小型微型计算机系统,2007,28(1):97−101.

朱锐(1980－),男,北京人,博士生,CCF 学生会员,主要研究领域为分布计算,可信计算.

郭长国(1973－),男,博士,副教授,CCF 会员,主要研究领域为分布计算,数据库技术.

王怀民(1962－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为分布计算中间件,软件 Agent,网络与信息安全.