ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*, Vol.19, No.2, February 2008, pp.432−445
DOI: 10.3724/SP.J.1001.2008.00432

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

# 支持复杂事务模式的Web应用服务器复制机制*

左 林[1,2,3+], 刘绍华[1], 冯玉琳[1,2], 魏 峻[1], 李 洋[1,2,3]

[1](中国科学院 软件研究所 软件工程技术中心,北京 100080)

[2](中国科学院 软件研究所 计算机科学重点实验室,北京 100080)

[3](中国科学院 研究生院,北京 100049)

## A Web Application Server Replication Scheme for Complex Transaction Patterns

ZUO Lin[1,2,3+], LIU Shao-Hua[1], FENG Yu-Lin[1,2], WEI Jun[1], LI Yang[1,2,3]

[1](Technology Center of Software Engineering, Institute of Software, The Chinese Academy of Sciences, Beijing 100080, China)

[2](Key Laboratory of Computer Science, Institute of Software, The Chinese Academy of Sciences, Beijing 100080, China)

[3](Graduate University, The Chinese Academy of Sciences, Beijing 100049, China)

+ Corresponding author: Phn: +86-10-62630989 ext 215, Fax: +86-10-62562538, E-mail: martin_zl@otcaix.iscas.ac.cn

**Zuo L, Liu SH, Feng YL, Wei J, Li Y. A Web application server replication scheme for complex transaction patterns. *Journal of Software*, 2008,19(2):432−445.** http://www.jos.org.cn/1000-9825/19/432.htm

**Abstract**: The support of reliability as adopted in conventional replication or transaction processing techniques is not enough due to their distinct objectives: Replication guarantees the liveness of computational operations by using forward error recovery, while transaction processing guarantees the safety of application data by using backward error recovery. Combining the two mechanisms for stronger reliability is a challenging task. Current solutions, however, are typically on the assumption of simple transaction pattern where only a server transaction exists at the middle-tier application server, and seldom think about some complex patterns, such as client transaction or nested transaction. To address this problem, four typical transaction patterns in J2EE application are recognized first. Then a Web application server replication scheme based state synchronization point concept, RSCTP (replication scheme for complex transaction pattern), is presented to uniformly provide exactly-once semantic reliability support for these complex transaction patterns. In this scheme, EJB components are replicated to endow business logics with high availability. In addition, by replicating transaction coordinator, the blocking problem of 2PC protocol during distributed transactions processing is eliminated. Different transaction scenarios are also discussed to illustrate the effectivity of this scheme. This scheme has been implemented, and it has been integrated into J2EE compatible application server, OnceAS, and the performance evaluation shows that its overhead is acceptable.

**Key words**: replication; end-to-end reliability; transaction pattern; J2EE

**摘 要**: 当前普遍采用的复制技术和事务处理技术都无法满足应用的 End-to-End 可靠性需求,前者通过前向错误

* Supported by the National Natural Science Foundation of China under Grant No.60573126 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant Nos.2003AA115440, 2005AA112030 (国家高技术研究发展计划(863)); the National Basic Research Program of China under Grant No.2002CB312005 (国家重点基础研究发展计划(973))

Received 2006-10-08; Accepted 2006-12-06

恢复来保证应用操作的存活性,后者通过后向错误恢复来保证应用数据的安全性.如何融合这两种技术以实现 End-to-End 可靠性保证,成为目前研究的热点问题.然而,已有的方法都是基于简单事务模式的假设,即只有中间层应用服务器上的容器发起事务,而很少考虑应用中普遍存在的复杂事务模式,如客户事务和嵌套事务.为了解决这个问题,首先识别出了几种典型的事务模式.针对这些事务模式,基于状态同步点概念提出了一种能够统一提供 End-to-End 可靠性保证的 Web 应用服务器复制机制 RSCTP(replication scheme for complex transaction pattern).RSCTP 机制采取 primary-backup 方式来复制 EJB 组件以保证业务逻辑的高可用性,同时采取 primary-backup 方式复制事务协调者来消除分布式事务处理中两阶段提交协议可能出现的阻塞问题.通过在不同事务模式下的失效分析,说明了该机制的有效性.已经实现了 RSCTP 机制并集成到了遵循 J2EE 规范的 Web 应用服务器 OnceAS 中.性能评价显示,该机制带来的系统开销较小.

**关键词:** 复制;End-to-End 可靠性;事务模式;J2EE

**中图法分类号:** TP311 **文献标识码:** A

J2EE 已经成为分布式计算环境下的主流中间件平台,其核心是位于中间层的 Web 应用服务器,它为创建、部署、运行、集成和管理 Web 应用提供一系列运行时服务[1].随着 Web 应用服务器在关键任务领域应用程度的加深,高可靠性支持逐渐成为其必须向上层应用提供的基础服务.

当前的 Web 应用服务器主要采取复制技术和事务处理技术来提供可靠性支持.复制技术[2]采取实体冗余的方式来保护部署在 Web 应用服务器上的业务逻辑.当主服务器有错误发生时,由备份服务器接管并重新执行当前的操作.事务处理技术侧重于保护位于数据库层的数据的 ACID 特性(原子性、一致性、隔离性和持久性).当有错误发生时,所有当前事务都回滚到最新的提交状态.然而在多层架构环境下,需要融合复制技术和事务处理技术来实现 End-to-End 的可靠性保障[3],已有许多研究者提出了解决办法[3−5],然而,这些方法主要适用于简单的事务模式,如图 1(a)中的 NCT-NNT(none client transaction-none nested transaction)模式,而没有为多种复杂事务模式提供统一的支持,包括客户层发起的事务(如图 1(b)所示)以及嵌套事务(如图 1(c)和图 1(d)所示).

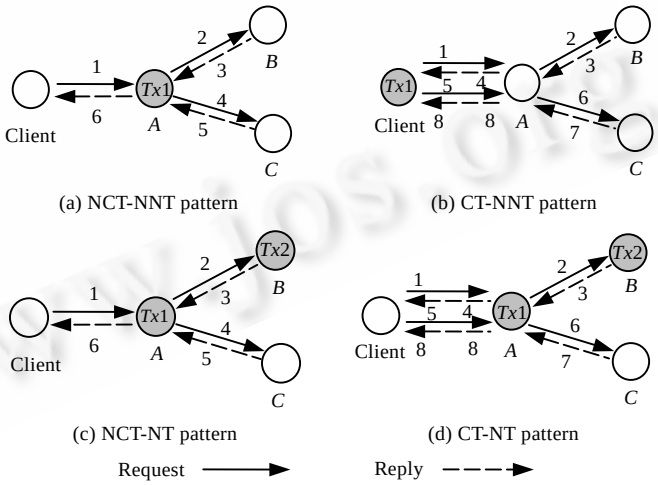


Fig.1 Complex transaction patterns

图 1 复杂事务模式

为了满足基于 J2EE 多层架构的企业应用对 End-to-End 可靠性的需求,本文结合复制技术和事务处理技术,提出了一种支持多种复杂事务模式的 Web 应用服务器复制机制 RSCTP(replication scheme for complex transaction pattern).RSCTP 采用 Primary-Backup[6]方式复制 EJB 组件来保证中间层业务逻辑的存活性.任意时刻,只有主 Web 应用服务器上的组件副本执行业务处理,并根据状态同步点(state synchronization point,简称

SSP)及时地与备份 Web 应用服务器上的组件副本进行状态同步.同时,RSCTP 采用 Primary-Backup 方式复制事务协调者以避免在分布式事务处理的两阶段提交过程中由于协调者失效而导致的阻塞问题.针对不同的事务模式场景的失效分析说明了该机制的有效性.此外,我们在遵循 J2EE 规范的 Web 应用服务器 OnceAS[7]上实现了该机制,实验结果显示,该方法性能良好.

## 1 事务模式

J2EE 通过 JTA[8]和 JTS[9]规范来提供分布式事务的支持,事务处理使用隐式编程模型,即事务上下文隐式地与请求线程关联.事务的发起者可以是客户或者 EJB 容器.前者通过 JTA 的 UserTransaction 接口发起一个客户事务(client transaction,简称 CT),该事务包含多个客户请求(client request,简称 CR);后者通过 TransactionManager 接口发起一个事务,该事务包含一个内部请求(internal request,简称 IR).此外,如果 EJB 方法在部署描述文件中的事务属性指定为“RequiresNew”,则会出现嵌套事务(nested transaction,简称 NT).其中,被嵌套的事务称为子事务(child),嵌套的事务称为父事务(parent),它们的提交是独立的,这不同于扩展事务模型中的嵌套事务模型.嵌套事务模式常常用来避免数据库资源的长时间阻塞.根据是否出现 CT 和 NT,我们识别出了 NCT-NNT,CT-NNT,NCT-NT 和 CT-NT 这 4 种事务模式,如图 1 中例子所示.

(1) NCT-NNT 模式.在此模式中,只存在容器发起的事务,没有 CT 和 NT.在图 1(a)所示的例子中,当一个有状态会话 Bean(stateful session bean,简称 SFSB) *A* 收到来自客户的请求 1 时,*A* 的容器为其向事务协调者 *T* 请求一个事务 *Tx*1.*A* 在 *Tx*1 的执行过程中调用实体 Bean(entity bean,简称 EB)*B* 和 *C*.由于 *B* 和 *C* 被持久化到不同的数据库中,因此,*Tx*1 是分布式事务.在 *A* 执行完请求时,容器提交 *Tx*1 后将结果返回给客户.

(2) CT-NNT 模式.在此模式中,只存在一个 CT,没有 NT.在图 1(b)所示的例子中,客户首先向事务协调者 *T* 发送 begin 请求来开始一个 CT *Tx*1,然后,客户向 SFSB *A* 发送一组 CR.最后,客户向 *T* 发送 commit 请求来提交 *Tx*1.

(3) NCT-NT 模式.在此模式中,只存在 NT 没有 CT.在图 1(c)所示的例子中,当 SFSB *A* 收到来自客户的请求 1 时,容器为该请求发起事务 *Tx*1.当 EB *B* 收到来自 *A* 的子调用 2 时,容器将 *Tx*1 挂起,然后为 2 开始一个新的事务 *Tx*2.在 *B* 执行结束即将返回时,容器提交 *Tx*2 后,将 *Tx*1 由挂起状态变为活动状态,然后继续 *A* 的执行.在这个例子中,*Tx*1 是 parent,*Tx*2 是 child,两个事务独立提交,并且 child 总是先于 parent 结束.这种模式常常用于解决由于并发而导致的资源的长时间封锁.

(4) CT-NT 模式.在此模式中,CT 和 NT 同时存在.在图 1(d)的例子中,客户在客户事务 *Tx*1 中向 SFSB *A* 发送一组请求,在 *A* 的执行过程中可能由于发起新的事务而导致若干 NT 的出现,而 *Tx*1 是所有 NT 的祖先.

## 2 系统模型

RSCTP 复制机制采取 primary-backup 复制方式,其系统模型如图 2 所示.系统将通信协议和复制协议完全分离,由两层组成:容错通信层(fault-tolerant communication layer,简称 FCL)和复制管理层(replication management layer,简称 RML).RML 层提供复制逻辑的实现,由位于客户端的 EJB 组件 Stub 中的复制拦截器 RI 和服务器端的复制管理器 RM 组成.FCL 向 RML 层提供通信服务,包括可靠的点到点通信和原子组播通信.此外,系统还通过拦截器(interceptor)在请求的处理流程中插入复制相关的逻辑.

**定义 1(Web 应用服务器组)**. 我们将部署了一个 EJB 组件的所有副本的 Web 应用服务器 *S* 的集合 $SG=\{S1,S2,...,Sn\}$称为 Web 应用服务器组.在任意时刻,组内只存在 1 个主服务器进程能够处理来自客户的请求,将其记为 *PS*,$PS \in SG$.其余的充当备份进程,与主服务器进程协作完成进程间的状态同步,记为 *BS*.

同一个组件被部署到 *SG* 中的每个 Web 应用服务器上,这些部署的组件称为组件副本.如图 2 所示,*C*1,*C*2,…,*Cn* 为组件 *C* 的副本.系统为每个组件副本分配一个寻址标识,即组件引用 *ejb_ref*=⟨*was_ref*,*ejb_id*⟩,其中,*was_ref* 标识副本所在 Web 应用服务器的引用,*com_id* 标识组件的 ID,同一组件的所有副本拥有相同的 *ejb_id*.系统将一个组件的所有副本看作一个逻辑实体,为其提供 EJB 组件组引用 *group_ref*=⟨*primary*,

*ejb_ref_set*⟩,其中,*ejb_ref_set* 表示所有组件副本的 *ejb_ref* 的集合,指针 *primary* 指向位于当前 *PS* 上的组件副本的 *ejb_ref*.当客户通过名字服务获取组件引用时,实际上得到的是 *group_ref*.当 *RI* 截获客户请求时,从 *group_ref* 中得到当前 *PS* 上的副本引用,并向其转发请求消息.值得提出的是,Web 应用服务器为组件副本分配独立的容器,不同组件的副本之间通过容器进行交互.

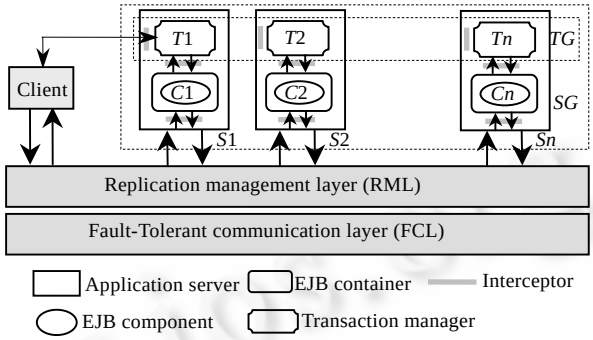


Fig.2 System model

图 2 系统模型

**定义 2(事务协调者组)**. 每个 Web 应用服务器都与一个事务协调者 *T* 关联,所有的协调者组成的集合 *TG*={*T*1,*T*2,...,*Tn*}定义为事务协调者组,*T* 向 RML 提供接口以查询事务的执行状态.*TG* 中与 *SG* 中的 *PS* 关联的协调者定义为主协调者,记为 *PT*.在任意时刻,只有 *PT* 能够为 *PS* 上的应用提供事务的相关操作,并将其记录到日志中,其他协调者则作为备份,记为 *BT*.当 *PT* 失效时,从 *BT* 中选举出的新 *PT* 根据事务日志进行恢复,继续未完成事务的提交操作.

**定义 3(失效检测)**. 在失效检测方面,系统使用一个经典的非可靠失效检测器◇*S*[10].◇*S* 满足两种属性:(1) 严格完全性(strong completeness),存在一个时刻 *t*,保证时刻 *t* 之后,任何无效的 *SG* 成员都可以被其他有效的成员检测到;(2) 弱精确性(weak accuracy),存在一个时刻 *t*,保证时刻 *t* 之后,任何有效的 *SG* 成员都不会被其他有效的成员怀疑为无效.在我们的系统中,该检测器作为容错基础设施实现.

由于在实际的实现中,Web 应用服务器往往与 *T* 驻留在相同的进程空间中,我们认为两者之间存在失效依赖关系,即当 Web 应用服务器失效后,与它关联的 *T* 也随之失效,同时,部署在其上的 EJB 组件副本也停止提供服务.此外,假定 Web 应用服务器的失效是 fail-stop 类型的,并且通信链路故障均为临时性故障.

为了保证失效时新的 *PS* 能够恢复到一致的状态,*PS* 通过 RML 在适当的时机与 *BS* 之间进行状态同步,下面给出状态同步的相关定义.

**定义 4(请求调用树)**. 一个客户请求的执行过程形成一棵请求调用树(invocation tree,简称 IT),如图 3 所示.IT 中的节点表示客户端应用或者请求访问过的组件,根节点为客户端应用;有向边表示节点之间存在调用关系,调用节点称为父节点,被调用节点称为子节点.我们假定不存在回路,因为 J2EE 规范不推荐使用循环调用[11].

**定义 5(状态同步点)**. 在请求调用树中进行状态同步的节点定义为状态同步点.SSP 分为两类:操作型同步点(operational synchronization point,简称 OSP)和事务型同步点(transactional synchronization point,简称 TSP).

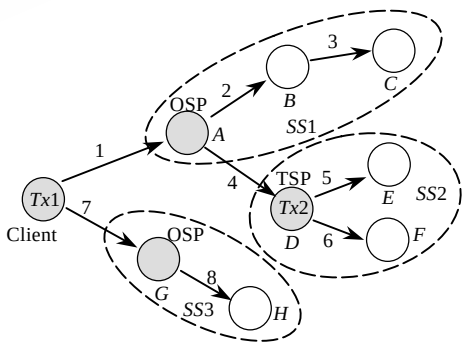


Fig.3 An example of SSP

图 3 状态同步点示例

**定义 6(操作型同步点)**. 在一棵 IT 中,如果一个节点的父节点是根节点并且根节点发起了一个客户事务 CT,则称该节点为操作型同步点 OSP.对于一个 OSP,状态同步发生在该节点发出的调用返回后,并且在调用结果返回父节点之前.图 3 中的组件 *A* 和 *G* 均为 OSP.

**定义 7(事务型同步点)**. 在一棵 IT 中,如果一个节点不是根节点,并且该节点的容器发起了一个事务,则称该节点为事务型同步点 TSP.对于一个 TSP,状态同步发生在该节点发出的调用返回后,并且在对应的事务提交之前.图 3 中的组件 *D* 为 TSP.

**定义 8(状态同步域)**. 一个 SSP 执行状态同步时所涵盖的组件集合定义为该 SSP 的状态同步域(synchronization scope,简称 SS).在 IT 中,一个节点 $P_i$ 的所有子节点表示为 $Subtree(P_i)$,$P_i$ 所包含的 SSP 节点为 $\{SSP_j|SSP_j\in Subtree(P_i)\}$,$j$=1,…,$n$,$n$ 为 SSP 的个数,则 $P_i$ 的状态同步域 $SS_i$ 为

$$SS_i = Subtree(P_i) - \bigcup_{j=1}^{n} Subtree(SSP_j).$$

在图 3 的例子中,OSP 同步点 *A* 和 *G* 的 *SS* 分别为 *SS*1 和 *SS*2,TSP 同步点 *D* 的 *SS* 为 *SS*2.

**定义 9(消息类型)**. *PS* 与 *BS* 的 RML 之间使用两类消息进行通信:

(1) 状态同步消息(*SSM*)用于 *PS* 将 *SSP* 的 *SS* 传递到 *BS*,*SSM*=(*TYPE*,*RID*,*TxID*,*STATES*,*REPLY*),其中,*TYPE* 表示 *SSM* 的类型,包括两种类型:TSP 表示是从 TSP 同步点发出的 *SSM*,OSP 表示是从 OSP 同步点发出的 *SSM*;*RID* 表示同步点所对应请求(IT 中同步点的输入边)的 ID;*TxID* 表示同步点对应的事务 ID,*STATES* 表示同步点对应同步域内的组件状态,对于 SFSB,*STATES* 包含它的整个状态,对于 EB,如果是 TSP 同步点,则包含它的 ID,如果 OSP 同步点,则包含它的整个状态;*REPLY* 表示同步点对应请求的应答.

(2) 事务同步消息(*TSM*)用于 *PS* 向 *BS* 传递事务状态,*TSM*=(*TYPE*,*TxID*,*OUTCOME*),其中,*TYPE* 表示 *TSM* 的类型,包括两种类型:TSP 表示是从 TSP 同步点发出的 *TSM*,OSP 表示是从 OSP 同步点发出的 *TSM*;*TxID* 表示同步点对应事务的 ID;*OUTCOME* 表示事物的执行状态:COMMIT 表示提交成功;ABORT 表示提交失败.

## 3 RSCTP复制机制

RSCTP 复制机制由 Client,Primary_Server,Backup_Server 和 Recovery 这 4 种算法组成.当位于客户端 Stub 中的复制拦截器截取到了客户请求时,Client 算法被调用;在 *PS* 处理客户请求的过程中,RML 调用 Priamry_Server 算法;当 *BS* 收到来自 *PS* 的同步消息时,RML 调用 Backup_Server 算法;当 *PS* 失效时,RML 调用 Recovery 算法.本文假设存在请求标识产生机制,能够为客户和 EJB 组件发出的请求产生唯一的标识.下面对这 4 种算法分别进行介绍.

### 3.1 Client算法

RI 拦截到客户请求 *R* 后,调用 Client 算法,其执行过程如下:

(1) 为 *R* 产生唯一的标识 *RID*;

(2) 如果 *R* 与一个客户事务关联,则获取事务标识 *TxID*;

(3) 创建一个调用上下文对象 *IC*,并将 *RID* 和 *TxID* 加入 *IC* 中;

(4) 从 Stub 中得到 *group_ref*,然后根据 *primary* 指针得到部署在 *PS* 上的组件引用 *ejb_ref*;

(5) 以 *R* 和 *IC* 为参数向 *ejb_ref* 所指向的 EJB 组件副本发起调用;

(6) 如果收到一个 OK 应答,则表示 *R* 调用成功,将结果返回给客户应用;

(7) 如果收到一个 TIMEOUT 异常,则说明所请求的 EJB 组件副本所在的 Web 应用服务器已经失效(假设链路故障是临时的).然后,顺序地从 *group_ref* 中获取下一个 *ejb_ref*,执行步骤(5);如果其中没有可用的 *ejb_ref*,则将一个 No_Valid_Member 异常返回给客户应用;

(8) 如果收到一个 NEW_GROUP_REF 消息,则说明调用的 Web 应用服务器不是当前的 *PS*,并且 *group_ref* 已经过时.以消息中返回的新 *group_ref* 替换当前 *group_ref*,然后根据 *primary* 指针得到部署在当前 *PS* 上的组件引用 *ejb_ref*,执行步骤(5);

(9) 如果收到一个 Exception 信息(不包括 TIMEOUT 异常),则将异常直接返回给客户应用.

注意,如果拦截到对事务协调者的请求,如 Begin 和 Commit,则不需要执行上述算法中的步骤(1)~步骤(3),并且,此时会向 *ejb_ref* 中的 *was_ref* 转发请求,*PS* 收到请求后会将其转发给 *PT*.

### 3.2 Primary_Server算法

*PS* 的 RML 在请求的处理流程中通过拦截器插入复制相关的处理逻辑,包括 3 个插入点:(1) 客户请求到达 *PS* 但还未进入容器前;(2) 容器或客户请求开始一个事务时;(3) 容器或客户提交一个事务时.下面分别针对 TSP 和 OSP 同步点对 Primary_Server 算法的执行过程进行介绍.

#### 3.2.1 对TSP同步点的处理

为了清楚地描述算法,首先介绍在 TSP 同步点时,请求的执行流程(如图 4 所示):① Web 应用服务器收到来自客户或 EJB 组件的请求 *R* 后将其传递给 *R* 所请求的 EJB 组件所在的容器;② 容器为 *R* 向事务协调者 *T* 请求一个新事务 *Tx*;③ 在得到 *Tx* 后,容器将 *R* 转发到相应的 EJB 组件处理(如果调用 EB,则容器通知持久化管理器 PM 从数据库中加载相应的状态到容器中);④ EJB 组件处理 *R* 的过程中可能调用其他组件,服务器将这些调用转发到对应组件所在的容器(相似的处理流程).在 *R* 的处理结束后,EJB 组件向容器返回结果;⑤ 容器要求 *T* 对 *Tx* 进行提交;⑥ 提交结束后,容器将应答返回给 *PS*,*PS* 将应答传递给调用者.

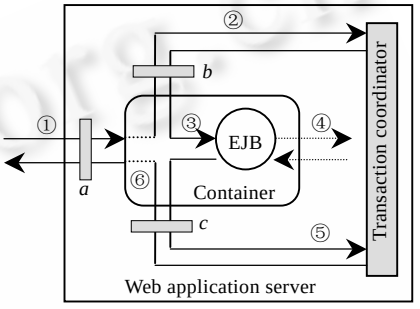


Fig.4 Request process procedure of TSP

图 4 TSP 同步点的请求处理过程

Primary_Server 算法的处理逻辑包括 3 个子过程:On_Invoke, On_Begin 和 On_Commit.RML 在图 4 中的 *a* 拦截点执行 On_Invoke,在 *b* 拦截点执行 On_Begin,在 *c* 拦截点执行 On_Commit.算法需要的数据结构包括:(1) 请求-应答映射 *RR*=⟨*RID*,*REPLY*⟩,其中,*RID* 表示请求的唯一 ID,*REPLY* 表示对应的应答;(2) 系统为每个请求关联一个执行上下文 *EC*=⟨*RID*,*REPLY*,*TxID*,*SS*⟩,其中,*RID* 表示所请求的 ID,*REPLY* 表示对应的应答,*TxID* 表示请求所关联的事务,*SS* 表示当前同步点的同步域.下面分别介绍这 3 种方法.

1. On_Invoke 执行过程

(1.1) 判断当前服务器是否为 *PS*,如果不是,则将最新的 *group_ref* 和一个 NEW_GROUP_REF 应答返回给客户端;如果是,则继续执行;

(1.2) 根据调用参数 *IC* 中的 *RID* 检查 *RR* 中是否存在相应的记录,如果有,则将应答及一个 OK 消息返回给客户端;如果没有,则继续;

(1.3) 将传入的请求 *R* 传递给容器以调用相应的 EJB 组件,并等待其返回应答 *Result*;

(1.4) 将 *RID* 和 *Result* 放入 *RR* 中;

(1.5) 删除对应的 *EC*;

(1.6) 如果 *Result* 为异常,则直接将其返回;否则,将 *Result* 随同 OK 消息返回给客户端.

2. On_Begin 执行过程

(2.1) 创建一个 *SS*,并将 *R* 所调用的 EJB 组件 ID 加入 *SS*,然后创建一个新 *EC*,将 *RID* 和 *SS* 加入 *EC* 中;

(2.2) 将 *EC* 与 *R* 关联;

(2.3) 调用主事务协调者 *PT* 的 *begin*()方法;

(2.4) 拦截到 *begin*()方法的返回结果,将其中的事务 ID 加入到 *EC* 中,然后将结果返回.

3. On_Commit 执行过程

(3.1) 通知持久化管理器 PM 将请求所修改过的 EB 的状态写回到数据库中;

(3.2) 创建一个 *SSM* 消息(“*TSP*”,*RID*,*TxID*,*STATES*,*REPLY*),其中,*STATES* 为 *SS* 中所有 EJB 组件的状态.然后,通过 FCL 层提供的可靠多播通信服务发送到所有的 *BS*;

(3.3) 调用 *PT* 的 *commit*()方法提交事务;

(3.4) 如果提交成功,则创建一个 *TSM* 消息(“*TSP*”,*TxID*,“*COMMIT*”);如果提交失败,则创建一个 *TSM* 消息(“*TSP*”,*TxID*,“*ABORT*”),然后,将该消息通过可靠组播通信服务发送到所有的 *BS*;

(3.5) 返回.

3.2.2 对OSP同步点的处理

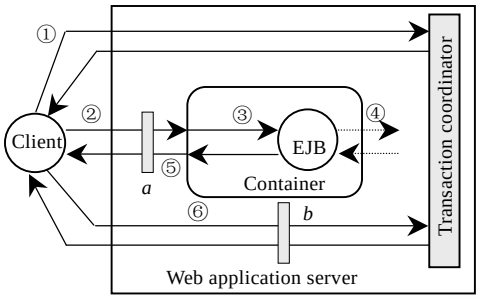


Fig.5 Request process procedure of OSP

图 5 OSP 同步点的请求处理过程

在 OSP 同步点时,请求的执行流程(如图 5 所示):① 客户向事务协调者 *T* 请求开始一个事务 *Tx*;② 客户向 Web 应用服务器发出一个请求 *R*(多个请求处理方式相同);③ Web 应用服务器将 *R* 传递给容器,容器将 *R* 转发到相应的 EJB 组件处理(如果调用 EB,则容器通知持久化管理器 PM 从数据库中加载相应的状态到容器中);④ EJB 组件处理 *R* 的过程中可能调用其他组件,服务器将这些调用转发到对应组件所在的容器处理;⑤ 容器将结果返回给客户端;⑥ 客户得到结果后要求 *T* 提交 *Tx*.

Primary_Server 算法包括两个子过程:On_Invoke 和 On_Commit,分别对应图 5 中的 *a* 和 *b*.下面给出其处理逻辑.

1. On_Invoke 执行过程

(1.1) 判断当前服务器是否为 *PS*,如果不是,则将最新的 *group_ref* 和一个 NEW_GROUP_REF 应答返回给客户端;如果是,则继续执行;

(1.2) 根据调用参数 *IC* 中的 *RID* 检查 *RR* 中是否存在相应的记录,如果有,则将应答及一个 OK 消息返回给客户端;如果没有,则继续执行;

(1.3) 创建一个 *SS*,并将传入请求 *R* 所调用的 EJB 组件 ID 加入 *SS*.然后创建一个新 *EC*,将 *RID*,*IC* 中的 *TxID* 和 *SS* 加入 *EC* 中;

(1.4) 将 *EC* 与 *R* 关联;

(1.5) 将传入的请求 *R* 传递给容器以调用相应的 EJB 组件,并等待其返回应答 *Result*;

(1.6) 创建一个 *SSM* 消息(“*OSP*”,*RID*,*TxID*,*STATES*,*REPLY*),其中,*STATES* 为 *SS* 中所有 EJB 组件的状态.然后,通过 FCL 层提供的可靠组播通信服务发送到所有的 *BS* 上;

(1.7) 将 *RID* 和 *Result* 放入 *RR* 中;

(1.8) 删除对应的 *EC*;

(1.9) 如果 *Result* 为异常,则直接将其返回;否则,将 *Result* 随同 OK 消息返回给客户端.

2. On_Commit 执行过程

(2.1) 判断当前服务器是否为 *PS*,如果不是,则将最新的 *group_ref* 和一个 NEW_GROUP_REF 应答返回给客户端;如果是,则继续执行;

(2.2) 根据 *TxID* 向主事务协调者 *PT* 询问当事务执行状态 *Tx_Status*;

(2.3) 如果 *Tx_Status* 表示事务已经提交,则将提交结果返回给客户端;如果还未提交,则首先通知持久化管理器 PM 将 *SS* 中修改过的 EB 状态写回到数据库中,然后让 *PT* 进行事务提交;

(2.4) 如果提交成功,则创建一个 *TSM* 消息(“*OSP*”,*TxID*,“*COMMIT*”);如果提交失败,则创建一个 *TSM* 消息(“*OSP*”,*TxID*,“*ABORT*”),然后,将该消息通过可靠多播通信服务发送到所有的 *BS*;

(2.5) 将提交结果返回给客户端.

### 3.3 Backup_Server算法

所有 *BS* 的 RML 在收到来自 *PS* 上同步点传来的消息时,执行 Backup_Server 算法,它包括 3 个子过程:On_Received_SSM,On_Received_TSM 和 IncarnateEJBs.当收到 *SSM* 消息时,RML 执行 On_Received_SSM 过程;当收到 *TSM* 消息时,RML 执行 On_Received_TSM 过程;IncarnateEJBs 作为独立执行的过程,根据收到的组件状态创建 EJB 组件实例.下面分 TSP 和 OSP 两种情况介绍 Backup_Server 算法的处理逻辑.

1. On_Received_SSM 执行过程

- 当收到一个 TSP 同步点的 *SSM* 消息时,RML 将其放入 FIFO 队列 *ITSM* 依次处理,对于每个 *msg*:

(1.1) 从 *msg* 中提取 *TxID*,然后将 *TxID* 和 *msg* 记录到临时状态集 *ISS* 中,*ISS*=⟨*txid*,*message*⟩,其中,*txid* 为 *SSM* 消息对应的事务 ID,*message* 为 *SSM* 消息体;

(1.2) 返回;

- 当收到一个 OSP 同步点的 *SSM* 消息时,RML 将其放入 FIFO 队列 *IOSM* 依次处理,对于每个 *msg*:

(1.3) 从 *msg* 中提取 *TxID*,然后将 *TxID* 和 *msg* 记录到临时状态集 *ISS* 中;

(1.4) 从 *msg* 中提取 *STATES*,根据 *STATES* 中的 EJB 组件 ID 在 *BS* 的容器中找到对应的组件实例,然后将其状态序列化后移入备份状态集 *BackupSet* 中,*BackupSet*=⟨*txid*,*ejb_set*,*message*⟩,其中,*txid* 为 *SSM* 消息对应的事务 ID,*ejb_set* 为备份的 EJB 组件状态集,*message* 表示收到的 *SSM* 消息,如果此时 *txid* 在 *BackupSet* 中已经存在对应的 *ejb_set*,则只需将与(*STATES*-*ejb_set*)对应的组件状态加入到 *ejb_set* 中;

(1.5) 将 *msg* 放入 *BackupSet* 中;

(1.6) 将 *STATES* 移入 *VSS* 中,由 IncarnateEJBs 子过程进行实例化操作;

(1.7) 将 *msg* 中的 *RID* 及 *REPLY* 记录到 *RR* 中,*RR*=⟨*RID*,*REPLY*⟩,其中,*RID* 表示请求的唯一 ID,*REPLY* 表对应的应答;

(1.8) 返回.

2. On_Received_TSM 执行过程

- 当收到一个 TSP 同步点 *TSM* 消息时,RML 将其放入 FIFO 队列 *ITTM* 依次处理,对于每个 *msg*:

(2.1) 从 *msg* 中提取 *TxID* 和事务提交结果 *OUTCOME*;

(2.2) 如果 *OUTCOME* 为 COMMIT,则说明与当前同步点关联的事务已提交成功,根据 *TxID* 从 *ISS* 中提取相应记录⟨*TxID*,*m*⟩,然后将 *m* 中的 *RID* 及 *REPLY* 记录到 *RR* 中,接着将 *m* 中的 *STATES* 移到有效状态集合 *VSS* 中,并删除⟨*TxID*,*m*⟩,IncarnateEJBs 子过程随即根据 *VSS* 中的状态创建 EJB 组件实例.

(2.3) 如果 *OUTCOME* 为 ABORT,则说明与当前同步点关联的事务提交失败,根据 *TxID* 从 *ISS* 中提取相应记录⟨*TxID*,*m*⟩,然后将 *m* 中 *RID* 和一个 TxAbortException 异常记录到 *RR* 中,最后将⟨*TxID*,*m*⟩从 *ISS* 中删除;

(2.4) 返回.

- 当收到一个 OSP 同步点 *TSM* 消息时,RML 将其放入 FIFO 队列 *IOTM* 依次处理,对于每个 *msg*:

(2.5) 从 *msg* 中提取 *TxID* 和事务提交结果 *OUTCOME*;

(2.6) 如果 *OUTCOME* 为 COMMIT,则说明与当前同步点关联的事务已提交成功,根据 *TxID* 删除 *BackupSet* 中相应的 *ejb_set*,然后从 *BackupSet* 提取相应的 *SSM* 消息 *m*,在将 *m* 中的 *RID* 和 *REPLY* 记录到 *RR* 后,删除 *m*.

(2.7) 如果 *OUTCOME* 为 ABORT,则说明事务提交失败,将 *BackupSet* 中对应的 *ejb_set* 放入 *VSS* 中进行实例化操作,然后从 *BackupSet* 提取相应的 *SSM* 消息 *m*,在将 *m* 中的 *RID* 和一个 TxAbortException 异常记录到 *RR* 后,删除 *m*.

3. IncarnateEJBs 执行过程

对 *VSS* 中的元素依次处理,如果是组件状态,则通过反序列化操作创建相应的实例,并将其放入对应容器的组件实例池中;如果是 EB 组件 ID,则通过 PM 从数据库中加载其状态,在创建实例后放入组件实例池中.

### 3.4 Recovery算法

当 RML 收到失效探测器的失效通知时,执行 Recovery 算法进行失效恢复,其处理过程如下:

(1) 处于监听状态,直到收到从 FaultDector 传来的失效通知为止.

(2) 如果是 *BS* 失效,则将其从 *SG* 中删除(为了避免 *PS* 失效时客户端的 RI 向已经失效的 *BS* 重发请求而延长响应时间,可以在 BS 失效时更新 *group_ref*,并将其附加在 *PS* 返回的应答中,这样,*RI* 就能及时地更新

持的 *group_ref*.这里没有将这种方法体现在算法中,但并不会影响算法的有效性),然后返回到(1).

如果是 *PS* 失效,则执行下面的步骤:

(3) 将失效的 *PS* 从 *SG* 中删除,然后从 *SG* 中当前有效的 *BS* 中选举一个作为新的 *PS*.

(4) 新 *PS* 所关联的事物协调者根据日志进行恢复,并继续未完成的事务,然后成为 *PT*.

(5) 执行状态恢复过程,将新 *PS* 恢复到正确的状态,包括下面的过程:

(5.1) 对 *ISS* 中的每个记录⟨*txid*,*message*⟩逐一处理:

(5.1.1) 通过 *PT* 得到 *txid* 的执行状态 *Tx_Status*.

(5.1.2) 如果 *Tx_Status* 显示 *txid* 对应的事务已经提交成功,则将 *message* 中的 *RID* 及 *REPLY* 记录到 *RR* 中,然后将其中的 *STATES* 组件状态集移入到 *VSS* 中.如果 *Tx_Status* 显示事务提交失败,则将 *message* 中 *RID* 和一个 TxAbortException 异常记录到 *RR* 中.

(5.1.3) 删除该记录.

(5.2) 对 BackupSet 中的每个记录⟨*txid*,*ejb_set*,*message*⟩逐一处理:

(5.2.1) 通过 *PT* 得到 *txid* 的执行状态 *Tx_Status*.

(5.2.2) 如果 *Tx_Status* 显示 *txid* 对应的事务已经提交成功,则将 *message* 中的 *RID* 和 *REPLY* 记录到 *RR* 后;如果 *Tx_Status* 显示事务提交失败,则将 *ejb_set* 放入 *VSS* 中进行实例化操作,然后将 *message* 中的 *RID* 和一个 TxAbortException 异常记录到 *RR* 后.

(5.2.3) 删除该记录.

(6) 更新 *group_ref*,并返回到(1).

### 3.5 处理非确定性问题

当请求执行路径上访问的所有 EJB 组件都是确定性(deterministic)组件(即组件在相同的初始状态下,接受相同的请求并按照相同的顺序处理请求,则会到达相同的结束状态)时,RSCTP 复制机制能够很好地保证事务的 exactly-once 执行语义和状态的一致性.如果出现非确定性组件,则可能产生状态不一致的问题,如图 6 所示:在客户请求 1 的执行过程中,组件 *E*1 和 *E*3 分别发起事务 *Tx*1 和 *Tx*2,这时嵌套事务出现,*Tx*1 为父事务,*Tx*2 为子事务.此外,只有 *E*2 为非确定性组件.根据前面的介绍可知,*E*1 和 *E*3 都为 TSP 同步点.当请求 1 沿着 *E*1-*E*2-*E*3-*E*4-*E*3-*E*2 的执行路径到达 *E*2 时,*PS* 发生了失效,根据前面的算法可知,此时,同步点 *E*3 执行 *Tx*2 后的状态已经在所有 *BS* 上生效,而同步点 *E*1 还没有向 *BS* 传递任何信息.客户端 *C* 重传请求 1 到新的 *PS*.新 *PS* 将其作为新的请求重新处理,由于 *E*1 是确定的,因此一定会向 *E*2 发出请求 2,但 *E*2 是非确定性的,则可能不会发出请求 3,而是向 *E*5 发出请求 5.结果,从 *C* 和新 *PS* 的角度来看,*Tx*2 的执行结果成为不一致的状态.

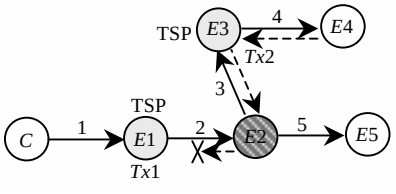


Fig.6 An example of non-determinism

图 6 非确定性示例

**定义 10(非确定同步点序列)**. 在一个请求执行过程中,由于非确定组件的存在而可能产生不一致状态的同步点序列 $NDSSP=(nd_1,nd_2,\ldots,nd_n)$,其中,$nd_i=\langle ssp_i,co_j\rangle$,$ssp_i$ 表示可能产生不一致状态的同步点(由于 OSP 同步点只会收到客户端的请求,故 $ssp_i$ 只可能是 TSP 同步点),$co_j$ 表示对应的补偿操作(每个 TSP 同步点的组件都为其事务性操作提供了补偿操作),$nd_i$ 按照调用的先后顺序排列.

RML 为每个客户请求 *R* 维护一个 *NDSSP*,并对 *R* 的执行路径进行跟踪,根据部署信息对当前调用组件的确定性进行检查,当发现一个非确定性组件时:

(1) 如果该组件不是 TSP 同步点,则随后遇到的所有 TSP 同步点都按照访问顺序加入到 *NDSSP* 中;

(2) 如果该组件是 TSP 同步点,则将其加入 *NDSSP*,而且随后遇到的所有 TSP 同步点都顺序加入 *NDSSP*.

当 *R* 执行结束并且对应的响应返回给客户端后,RML 删除与 *R* 对应的 *NDSSP*.当 *PS* 发生失效时,在失效恢复阶段,RML 会依次执行 *NDSSP* 序列中的同步点所对应的补偿操作来消除不一致的状态.

## 4 算法分析

本节针对不同的事务模式,讨论 RSCTP 复制机制的有效性,即满足请求的 exactly-once 执行和状态一致性.

(1) 当为 NCT-NNT 模式时,讨论以下几种 *PS* 失效的情况:

A. 在客户请求 *R* 到达 *PS* 之后,并且在容器开始一个事务之前.这时,*R* 还没有造成组件状态的任何改变,并且没有产生任何事务.当客户端向新的 *PS* 重发 *R*(client 算法(5)~算法(7))时,新 *PS* 将把它当作一个新的请求进行处理(primary_server 算法 TSP 对应的 On_invoke 子过程),不会产生状态的不一致.

B. 在容器发起事务 *Tx* 之后,并且在 TSP 的 *SSM* 消息发出之前.这时,*R* 的执行造成了 *PS* 上组件状态的改变,但是这些改变还没有传递到任何 *BS*.新的 *PS* 不会包含这些改变的状态.当从客户端收到重传的 *R* 后,新 *PS* 将其作为新的请求重新执行,容器会为其发起新的 *Tx*,并且原有的 *Tx* 最终会被新 *PT* 终止.这时不会出现状态不一致,保证了 *R* 的 exactly-once 执行.

C. 在事务的两阶段提交过程中.这时,*R* 的执行所产生的状态变化和处理结果已经可靠组播到了所有的 *BS*,新的 *PS* 在失效恢复过程中继续执行还未完全提交的事务(recovery 算法(4)),并将组件恢复到正确的状态(recovery 算法(5)).对于提交成功的事务,其对应的状态变化立即生效;对于提交失败的事务,将撤销其对应的状态,并且 *RR* 中记录了相应的应答.当新的 *PS* 收到重传的 *R* 时,不会再次执行 *R*,只是从 *RR* 映射表中获取对应的处理结果(primary_server 算法 TSP 对应的 on_invoke 子过程),保证了状态的一致.

D. 在 TSP 的 *TSM* 消息发出之后,并且在 *R* 的应答返回给客户之前.此时,由于所有的 *BS* 都已知道了 *Tx* 的提交结果,并且对 *R* 所引起的状态变化作了相应的处理(生效或撤销),并把相应的应答记录到了 *RR* 中.当新 *PS* 收到重传的 *R* 时,只需将 *RR* 中对应的应答返回即可,不会导致 *R* 的重复处理,保持了状态的一致.

(2) 当为 CT-NNT 模式时,讨论以下几种 *PS* 失效的情况:

A. 在客户发送 Begin 请求开始一个事务之后,并在对应的应答返回客户之前.这时,还没有任何请求到达 *PS*,也没有发生任何组件状态改变.客户端会向新 *PS* 重新发送 Begin 请求,没有产生状态的不一致.

B. 在客户开始了事务 *Tx* 之后,并且在客户提交 *Tx* 之前.这时,可能出现 3 种情况:

(a) 在客户发出请求 *R* 之后,且在 *R* 到达 *PS* 上的 OSP 之前.此时,*R* 还没有被 *PS* 处理,故没有引起组件任何状态的变化.在失效恢复过程中,新 *PT* 会根据日志恢复 *Tx*(recovery 算法(4)).当新 *PS* 接到客户端的 RI 重发的 *R* 后,会将其作为新的请求处理,没有状态不一致出现.

(b) 在 *R* 到达 OSP 之后,且在 OSP 的 *SSM* 消息发出之前.此时,由于 *R* 的执行改变了 *PS* 上的组件状态,但是这些改变并没有传递到 *BS*.在失效恢复过程中,新 *PT* 会根据日志恢复 *Tx*.当新 *PS* 在接到 RI 重发的 *R* 时,其状态与执行 *R* 之前是一致的,新 *PS* 会再次处理 *R*.从客户和应用的角度来看,*R* 其实只被执行了一次,不会出现状态不一致.

(c) 在 OSP 的 *SSM* 消息发出之后,且在应答返回客户之前.此时,请求 *R* 引起的状态变化已经通过可靠组播传递到了所有的 *BS*(primary_server 算法 OSP 对应的 on_invoke 子过程(1.6)),并且 *BS* 将应答记录到了 *RR* 中(backup_server 算法(1.7)).在失效恢复过程中,新的 *PT* 会根据日志恢复 *Tx*.故新的 *PS* 收到重传的 *R* 时,只是从 *RR* 中取出应答并返回给客户,不会重复处理 *R*,状态保持了一致.

C. 在客户发出 Commit 请求以提交 *Tx* 之后,并且在应答返回给客户之前.如果在 *PT* 提交 *Tx* 之前 *PS* 失效,则新的 *PT* 在收到重传的 Commit 后会提交 *Tx*;如果在 *PT* 提交过程中 *PS* 失效,则在新 *PS* 的失效恢复过程中会继续 *Tx* 的提交(recovery 算法(4));如果在 *PT* 完成 *Tx* 提交后 *PS* 失效,则在收到重传的 Commit 请求后,RML 通过新 *PT* 得到提交结果并返回.因此,Commit 请求不会重复执行.

(3) 对于 NCT-NT 模式,结合图 1(c)中的例子进行讨论.此例中存在两个 TSP 同步点:组件 *A* 和 *B*,它们对应的事务为 *Tx*1 和 *Tx*2,其中 *Tx*1 是父事务,*Tx*2 是子事务.存在以下 3 种 *PS* 失效的情况:

A. 在 *Tx*2 提交之前.此时,*Tx*1 和 *Tx*2 在 *PS* 上引起的组件状态变化都没有传递到 *BS*,当 *PS* 失效后,*Tx*1 和

*Tx*2 最终会被新 *PT* 终止.当新的 *PS* 收到重传的请求 1 时,将其作为新的请求重新执行,容器会发起新的 *Tx*1 和 *Tx*2.由于新 *PS* 的状态与未执行请求 1 时相同,故实际上请求 1 只执行了一次,并且状态保持一致.

B. 在 *Tx*2 提交之后,且在 *Tx*1 提交之前.此时,根据 *A* 是否为确定性组件分两种情况进行讨论:

(a) *A* 是确定性组件.此时,*Tx*2 引起的状态变化已经在所有的 *BS* 上生效,而 *Tx*1 的状态变化还没有传递到 *BS*.当新的 *PS* 收到重传的请求 1 时,将重新执行它,容器会重新发起新的 *Tx*1,原来的 *Tx*1 最终会被 *PT* 终止.由于 *A* 是确定性的,故一定会重新发出请求 2.因为请求 2 的应答已经记录到了 *RR* 中,所以不会重新执行它.可以看出,这时保证了请求的 exactly-once 执行语义和状态的一致性.

(b) *A* 是非确定性组件.此时,RML 将 *B* 组件加入到了 *NDSSP* 中.当 *PS* 失效时,在失效恢复过程中,RML 会调用 *B* 的补偿操作消除 *Tx*2 引起的状态变化.当恢复结束后,新 *PS* 的状态与未执行请求 1 时相同.重传的请求 1 将作为新的请求来处理,并且容器会发起新的 *Tx*1 和 *Tx*2,原来的 *Tx*1 最终会被 *PT* 终止,因此不会出现状态不一致和请求的重复执行.

C. 在 *Tx*1 提交之后.此时,*Tx*1 和 *Tx*2 引起的状态变化都传递到了所有的 *BS*.当新的 *PS* 收到重传的请求 1 时,直接从 *RR* 中取出对应的应答返回给客户端,不会重复执行请求 1,并且状态是一致的.

(4) 对于 NCT-NT 模式,结合图 1(d)中的例子进行讨论.此例中组件 *A* 为 OSP,组件 B 为 TSP,它们对应的事务为 *Tx*1 和 *Tx*2,其中,*Tx*1 是父事务,*Tx*2 是子事务.讨论以下两种 *PS* 失效的情况:

A. 在 *Tx*2 提交之前.此时,*Tx*1 和 *Tx*2 在 *PS* 上引起的组件状态变化没有传递到 *BS*.在失效恢复过程中,新 PT 会根据日志恢复 *Tx*1(recovery 算法(4)).当新的 *PS* 收到重传的请求 1 时,将其作为新的请求重新执行,组件 *B* 的容器会发起新的 *Tx*2,原来的 *Tx*2 最终会被 *PT* 终止.由于新 *PS* 的状态与未执行请求 1 时相同,故实际上请求 1 只执行了 1 次,并且不会产生不一致的情况.

B. 在 *Tx*2 提交之后,且在 *Tx*1 提交之前.假设组件 *A* 还没有从 *B* 收到请求 2 的应答.这时,根据 *A* 是否为确定性组件分两种情况加以讨论:

(a) *A* 是确定性组件.此时 *Tx*2 在 *PS* 上引起的状态变化已经在所有的 *BS* 上生效,而 *Tx*1 的状态变化还没有传递到 *BS*.在失效恢复过程中,新的 *PT* 会根据日志恢复 *Tx*1.当新的 *PS* 收到重传的请求 1 时,将重新执行它.由于 *A* 是确定性的,故一定会重新发出请求 2.因为请求 2 的应答已经记录到了 *RR* 中(backup_server 算法(2.2)),所以不会重新执行它.可以看出,这时保证了请求的 exactly-once 执行语义和状态的一致性.

(b) *A* 是非确定性组件.此时,RML 已将 *B* 组件加入到 *NDSSP* 中.当 *PS* 失效时,失效恢复过程中,RML 会调用 *B* 的补偿操作消除 *Tx*2 引起的状态变化.当恢复结束后,*Tx*1 被恢复,而且新 *PS* 的状态与未执行请求 1 时相同.重传的请求 1 将作为新的请求来处理,因此不会出现状态不一致和请求的重复执行.

## 5 性能评价

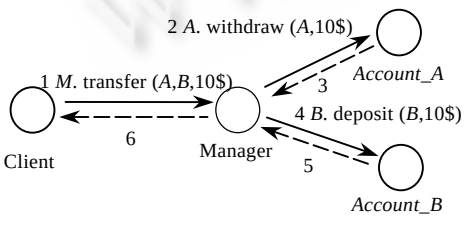


Fig.7 A bank transfer application

图 7 银行转账应用

我们已经在遵循 J2EE 规范的 Web 应用服务器 OnceAS 上实现了 RSCTP 复制机制,并通过实验对该方法的性能进行了测试.实验中实现了一个银行转账的应用,如图 7 所示,包括一个客户端(client),一个管理员组件 Manager(由 SFSB 实现),两个账户组件 *Account_A* 和 *Account_B*(均由 EB 实现).Client 调用 Manager 的 Transfer 接口发起一次转账任务,Manager 首先调用 *Account_A* 的 Withdraw 接口执行取款操作,然后调用 *Account_B* 的 Deposit 接口执行存款操作.为了模拟分布式事务,*Account_A* 和 *Account_B* 分别持久化到两个数据库.

我们设计了 4 组实验:(1) 测试 NCT-NNT 模式——在一次转账任务中,Manager 的容器发起一个事务;

(2) 测试 CT-NNT 模式——只有 Client 发起一个事务;(3) 测试 NCT-NT 模式——Manager 和 *Account_A* 的容器各发起一个事务;(4) 测试 CT-NT 模式——Client 发起一个事务, *Account_A* 的容器发起一个事务.

每组实验都分两种情况进行对比测试:(1) 不采用 RSCTP 复制机制(使用 1 台 OnceAS);(2) 采用 RSCTP 复制机制(使用 4 台 OnceAS,1 个为 *PS*,3 个为 *BS*).由于 RSCTP 机制采取被动复制,任意时刻只有 1 台 OnceAS 执行客户请求,对客户来说,与不使用 RSCTP 机制时所使用的计算资源是相同的.对于每种情况,测试在并发客户不同时,转账任务的平均响应时间.客户机上的模拟器创建多个并发线程,模拟多个并发客户,每个客户连续地向 Manager 组件发送转账任务(当一个任务返回后立即发送下一个任务),并记录每个任务的响应时间.测试阶段分为初始期 *Trampup*=100s,测试期 *Tsteady*=5000s 和结束期 *Trampdown*=100s.在初始期,模拟器只发送请求而不记录响应时间,只在 *Tsteady* 内进行记录.最后,将所有客户记录的响应时间的平均值作为本次测试的结果,同一测试执行 3 次,其平均值作为本组实验的测试结果.

实验环境包括 7 台 PC 机,其中:1 台 PC 机运行客户模拟器,其配置为 Dell OPTIPLEX GX260,CPU 2.66G,512M 内存;4 台 PC 机各自运行 OnceAS 应用服务器,其配置为 Dell PowerEdge 2600,Dual CPU 3.04G,2G 内存;两台 PC 机各自运行 Oracle 8i 数据库,其配置为 IBM Xseries 235,Single CPU 2.66G,2G 内存.所有的机器通过 1000M Ethernet 组成一个独立的局域网.

各组实验的测试结果如图 8 所示.从图中可以看出,当不使用 RSCTP 机制时,在并发客户数相同的情况下,NCT-NNT 事务模式的平均响应时间最小,而 CT-NT 的最大.这是因为后者相对于前者多执行了一个事务.

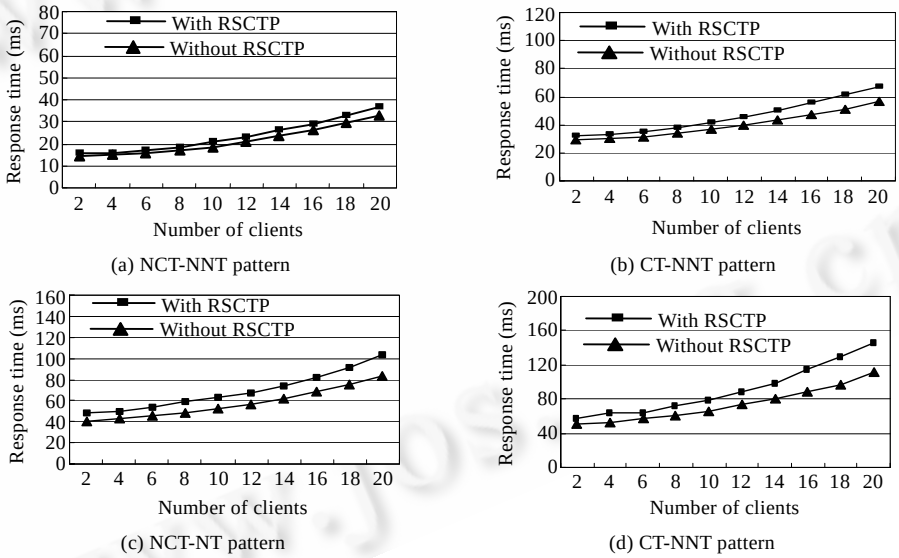


Fig.8 Test results of response time

图 8 响应时间测试结果

从图 8(a)可以看出,在 NCT-NNT 模式下,引入 RSCTP 复制机制导致平均响应时间增加了大约 9%,这些开销主要是由于 *PS* 与 *BS* 之间发送 *SSM* 和 *TSM* 消息引起的.在 CT-NNT 模式下(如图 8(b)所示),平均响应时间大约增加了 13%,大于 NCT-NNT 模式,这是由于复制机制在 *SSM* 的传递过程中对实体 Bean 组件的序列化和反序列化操作引起的.从图 8(c)可以看出,在 NCT-NT 模式下,RSCTP 机制引起的平均响应时间增加要大于 NCT-NNT 和 CT-NNT 模式,大约为 20%.这是因为此时比前两种模式多组播了一个 *SSM* 消息和一个 *TSM* 消息.在 CT-NT 模式下(如图 8(d)所示),RSCTP 使得平均响应时间增加了大约 25%,高于 NCT-NT 模式,这主要也是由于组件的序列化和反序列化操作引起的.因此,RSCTP 机制在 NCT-NNT 模式下引入的开销最小,而在 CT-NT 模式下最大.

从各种事务模式下的实验结果可以看出,RSCTP 复制机制引入了较小的系统开销.

## 6 相关工作

已有许多研究致力于提高基于 CORBA 的分布式系统的可靠性.文献[12]采取集成的方法将复制和组通信机制集成到 ORB 中来实现容错.文献[13]采取服务的方法定义了一组对象和接口,作为 CORBA 服务来为应用提供容错.文献[14]使用拦截的方法对 IIOP 消息进行拦截,然后映射到组通信系统中,提供透明的容错,同时支持 Primary-Backup 和 Active 复制.这些工作主要侧重于通过复制来提高中间层对象的可靠性,忽略了复制与事务的协作问题.文献[3]探索了融合复制和事务两种技术来提高 CORBA 应用 End-to-End 可靠性的问题.文献[5]基于 Eternal[12]提出了一种容错架构,该架构将 FT-CORBA[15]和 OTS[16]融合在一起,为基于三层架构的应用提供更高的可靠性支持.这些工作主要基于简单的事务模式,没有提出能够同时支持多种事务模式的有效办法.

对于 J2EE 三层架构,也有许多研究工作为提高应用的可靠性提出了解决办法.JBoss[2]基于 JGroups[17]组通信系统实现了对 EJB 组件状态、会话状态等的复制,但是复制算法采取基于方法的状态同步方式,而没有考虑对事务的支持,因此会导致状态不一致情况的发生.文献[18]提出了一种复制框架来简化各种复制算法的集成工作.文献[4]基于类似于文献[19]中的标记机制(marker mechanism),提出了一种 J2EE 复制算法,但是只考虑了容器发起事务的情况,即 NCT-NNT 模式,并且不支持分布式事务.这些工作都没有考虑多种事务模式并存的情况.文献[20]为不同的事务模式分别提出了相应解决办法.当存在客户事务时,客户端需要记录当前事务中已经完成的请求-应答对,并且需要复杂的并发控制机制避免不一致的发生.此外,没有考虑分布式事务的情况.与之相比,RSCTP 基于状态同步点,为各种事务模式提供了统一的支持,客户端实现简单,并且支持分布式事务.

## 7 总 结

提高基于 J2EE 三层架构分布式应用的可靠性已经成为研究的热点.针对实际应用中存在的多种事务模式,提出了一种基于状态同步点的 Web 应用服务器复制机制 RSCTP.它结合复制和事务处理技术,能够统一地为多种事务模式提供可靠性保障,支持分布式事务并且对应用完全透明.针对不同事务模式场景的失效分析说明了该机制的有效性.我们已经在 Web 应用服务器 OnceAS 中实现了 RSCTP 机制.实验结果表明,该方法引入了较小的系统开销.

进一步的工作包括两个方面:(1) 扩展 RSCTP 复制机制,使其能够处理客户层和数据层的失效;(2) 研究如何保证业务逻辑层中复制的 Web 应用服务器与数据层中复制的数据库之间的正确交互.

**References**:

[1] Fan GC, Zhong H, Huang T, Feng YL. A survey on Web application servers. Journal of Software, 2003,14(10):1728−1739 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/14/1728.htm

[2] Labourey S, Burke B. JBoss Clustering v2.0. The JBoss Group, 2002.

[3] Felber P, Narasimhan P. Reconciling replication and transactions for the end-to-end reliability of CORBA applications. In: Meersman R, Tari Z, eds. Proc. of the CoopIS/DOA/ODBASE. LNCS 2519, Heidelberg: Springer-Verlag, 2002. 737−754.

[4] Wu HG, Kemme B. Eager replication for stateful J2EE servers. In: Meersman R, Tari Z, eds. Proc. of the CoopIS/DOA/ODBASE. LNCS 3291, Heidelberg: Springer-Verlag, 2004. 1376−1394.

[5] Zhao WB, Moser LE, Melliar-Smith PM. Unification of transactions and replication in three-tier architectures based on CORBA. IEEE Trans. on Dependable and Secure Computing, 2005,2(1):20−33.

[6] Budhiraja N, Maraullo K, Schneider FB, Toueg S. The primary-backup approach. In: Mullender S, ed. Proc. of the Distributed Systems. New York: ACM Press, 1993. 199−216.

[7] Huang T, Chen NJ, Wei J, Zhang WB, Zhang Y. OnceAS/Q: A QoS-enabled Web application server. Journal of Software, 2004, 15(12):1787−1799 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/15/1787.htm

[8] Cheung S, Matena V. Java Transaction API Specification v1.0. Sun Microsystems Inc., 2002.

[9] Cheung S. Java Transaction Service Specification v1.0. Sun Microsystems Inc., 1999.

[10] Chandra TD, Toueg S. Unreliable failure detectors for reliable distributed systems. Journal of the ACM, 1996,43(2):225−267.

[11] DeMichiel LG. Enterprise JavaBeans Specification 2.1. Sun Microsystems Inc., 2003.

[12] Maffeis S. Electra-Making distributed programs object-oriented. In: Proc. of the USENIX Symp. on Experiences with Distributed and Muhiprocessor System IV. Berkeley: USENIX Association, 1993. 143−156.

[13] Felber P. The CORBA object group service: A service approach to object groups in CORBA [Ph.D. Thesis]. Lausanne: Swiss Federal Institute of Technology, 1998.

[14] Narasimhan P, Moser LE, Melliar-Smith PM. Eternal—A component-based framework for transparent fault-tolerant CORBA. Software-Practice and Experience, 2002,32(8):771−788.

[15] Object Management Group. Fault tolerant CORBA. Technical Committee Document formal/01-12-29, 2001.

[16] Object Management Group. Object transaction service specification. Technical Committee Document formal/03-09-02, 2003.

[17] Abdellatif T, Cecchet E, Lachaize R. Evaluation of a group communication middleware for clustered J2EE application servers. In: Meersman R, Tari Z, eds. Proc. of the CoopIS/DOA/ODBASE. LNCS 3291, Heidelberg: Springer-Verlag, 2004. 1571−1589.

[18] Bartoli A, Maverick V, Patarin S, Wu H. A framework for prototyping J2EE replication algorithms. In: Meersman R, Tari Z, eds. Proc. of the CoopIS/DOA/ODBASE. LNCS 3291, Heidelberg: Springer-Verlag, 2004. 1413−1426.

[19] Frølund S, Guerraoui R. A pragmatic implementation of E-transactions. In: Proc. of the Int'l Symp. on Reliable Distributed Systems. Nürnberg: IEEE Computer Society, 2000. 186−195.

[20] Wu HG, Kemme B. Fault-Tolerance for stateful application servers in the presence of advanced transactions patterns. In: Proc. of the Int'l Symp. on Reliable Distributed Systems. Orlando: IEEE Computer Society, 2005. 95−108.

## 附中文参考文献:

[1] 范国闯,钟华,黄涛,冯玉琳.Web 应用服务器研究综述.软件学报,2003,14(10):1728−1739. http://www.jos.org.cn/1000-9825/14/1728.htm

[7] 黄涛,陈宁江,魏峻,张文博,张勇.OnceAS/Q:一个面向 QoS 的 Web 应用服务器.软件学报,2004,15(12):1787−1799. http://www.jos.org.cn/1000-9825/15/1787.htm

**左林**(1975－),男,四川成都人,博士,主要研究领域为网络分布计算,软件工程.

**刘绍华**(1976－),男,博士,助理研究员,主要研究领域为网络分布计算,工作流技术,服务协作理论与中间件.

**冯玉琳**(1942－),男,博士,研究员,博士生导师,CCF 高级会员,主要研究领域为软件工程,形式化方法,分布对象计算.

**魏峻**(1970－),男,博士,研究员,CCF 高级会员,主要研究领域为软件工程,软件理论,网络分布计算.

**李洋**(1979－),男,博士生,主要研究领域为网络分布计算,软件工程.