Um Serviço de Transações Cooperativas Baseado em CORBA

Álvaro Hervella

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Um Serviço de Transações Cooperativas Baseado em CORBA

Álvaro Hervella[1]

Novembro de 2002

Banca Examinadora:

- Profa. Dra. Maria Beatriz Felgar de Toledo
  IC-UNICAMP (Orientadora)
- Prof. Dr. Edmundo Roberto Mauro Madeira
  IC-UNICAMP
- Prof. Dr. Eleri Cardozo
  FEEC-UNICAMP
- Profa. Dra. Anamaria Gomide
  IC-UNICAMP (Suplente)

[1]**Apoio financeiro da FAPESP, processo número 99/03475-0.**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Hervella, Álvaro

H445s Um serviço de transações cooperativas baseado em CORBA / Álvaro Hervella -- Campinas, [S.P. :s.n.], 2003.

Orientador : Maria Beatriz Felgar de Toledo

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Sistema de transação (Sistemas de computação). 2. CORBA (Arquitetura de computador). 3. Java (Linguagem de programação de computador). I. Toledo, Maria Beatriz Felgar de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 03 de dezembro de 2002, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Eleri Cardozo
FEEC - UNICAMP

Prof. Dr. Edmundo Roberto Mauro Madeira
IC - UNICAMP

Profa. Dra. Maria Beatriz Felgar de Toledo
IC - UNICAMP

Um Serviço de Transações Cooperativas Baseado em CORBA

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Álvaro Hervella e aprovada pela Banca Examinadora.

Campinas, 21 de Fevereiro de 2003.

Profa. Dra. Maria Beatriz Felgar de Toledo
IC-UNICAMP

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Agradecimentos

Antes de mais nada, gostaria de agradecer à Profa. Beatriz pela orientação séria em todos esses anos, ajudando-me a desenvolver um trabalho interessante, atual e comprometido com a formação do conhecimento sem mostrar-se desanimada ou impaciente. Muito obrigado.

Agradeço aos meus pais por me incentivarem nos momentos difíceis e por me propiciarem condições para chegar onde agora almejo. Aos meus familiares de modo geral, que estiveram sempre presente.

Especialmente, agradeço a minha namorada Marlene nesses anos de convivência. Sua atenção e carinho sempre me fortificou. Espero que tenha compreendido todos os momentos de privação pelos quais passamos em decorrência de minha dedicação à tese. Te adoro.

Gostaria de agradecer aos colegas do IC-Unicamp e do LSD que direta ou indiretamente colaboraram para a realização deste trabalho.

E claro, a você, leitor, pelo prestígio e interesse demonstrado por este trabalho. Espero que o conteúdo desta dissertação lhe sirva para suas necessidades ou apenas para seu conhecimento. Pois, não é isso que interessa?! Obrigado.

Resumo

O Serviço de Transações Cooperativas desenvolvido nessa tese visa atender os requisitos de aplicações cooperativas. Para isso, oferece novos modos de tranca mais flexíveis que melhoram a visibilidade entre usuários que trabalham em colaboração, permite a estruturação de aplicações em uma hierarquia de transações que modelam a hierarquia de grupos de trabalho e relaxa a propriedade de atomicidade salvando de forma persistente estados das transações.

Esse Serviço é oferecido através de objetos CORBA e foi implementado na linguagem Java utilizando OrbixWeb 3.1 como a plataforma de comunicação.

Abstract

The Cooperative Transaction Service developed in this thesis aims at meeting the requirements of cooperative applications. For that, it provides new lock modes to increase the visibility between users that work in collaboration; it allows application structuring as a hierarchy of transactions which models the hierarchy of work groups; and finally it relaxes the atomicity property saving transaction state persistently.

This Service is provided by CORBA objects and was implemented in Java using OrbixWeb 3.1 as the underlying communication platform.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Apesar de transações serem fundamentais para uma grande variedade de aplicações, não é possível adotar o mesmo modelo de transação para todas elas. Estas aplicações podem ser classificadas de acordo com as características e as demandas impostas pelo gerenciamento de transações, podendo ser divididas em duas classes:

- **Aplicações Tradicionais:** Aplicações comerciais/financeiras como, por exemplo, sistemas bancários e sistemas de reservas de vôos, caracterizando-se pela curta duração, acesso aos dados de modo competitivo e estrutura de dados simples.
- **Aplicações Avançadas:** CAD/CAM[1], ferramentas CASE[2] e groupware são exemplos destas aplicações, tendo longa duração, acesso aos dados de maneira cooperativa e estrutura complexa de dados.

Existem diversos modelos, como em [Elm92, BHG87], que levam em consideração os requisitos das aplicações avançadas. Esses modelos revisam os mecanismos tradicionais de controle de concorrência e recuperação para um ambiente onde as aplicações têm longa duração e os usuários desenvolvem tarefas colaborando entre si. Estes mecanismos consideram que:

1. A longa duração de uma aplicação aumenta o número de conflitos entre as transações envolvidas. A solução destes conflitos usando transações que abortam ou bloqueiam seus dados degrada o desempenho do sistema.
2. O critério de correção baseado em seriabilidade[3] impede interações entre os usuários que trabalham de forma cooperativa.

[1] Computer Aided Design/Computer Aided Manufacturing
[2] Computer Aided Software Engineering
[3] Em inglês *serializability*

3. Desfazer completamente uma transação interrompida por uma falha pode ser inaceitável quando tal transação tiver longa duração.

As seções abaixo discutem os problemas que devem ser tratados dentro do modelo de transação para aplicações avançadas.

Controle de Concorrência

Alguns trabalhos revisam mecanismos tradicionais aumentando o nível de concorrência das transações por permitir a liberação dos recursos de modo antecipado, como em [GMS87, SW90, KP92]. Outros mecanismos são baseados em um critério de correção mais flexível. Em [NRZ92], não somente um aumento de concorrência é alcançado, mas também a visibilidade entre os usuários. Estes mecanismos permitem que cada transação de grupo especifique os padrões de operação e os conflitos presentes. Tal especificação determina as possíveis interações entre os membros de um grupo. Em [NG92], os usuários envolvidos no desenvolvimento de um projeto determinam as atualizações a serem aplicadas em tempo de execução. Um protocolo de comunicação transmite as atualizações propostas, as quais podem ser aprovadas, rejeitadas ou revisadas.

Recuperação de Falhas

Os mecanismos de recuperação para aplicações de longa duração relaxam a propriedade de atomicidade, permitindo o uso de *checkpointing*. Se uma falha interrompe a execução de uma transação com longa duração, o mecanismo de recuperação tem a capacidade de restaurar o estado da transação de acordo com o estado intermediário salvo pelo checkpointing, ao contrário do procedimento tradicional de descartar todas as atualizações desde o início da transação.

Cooperação

Parte dos modelos de transações para aplicações avançadas provê interação entre seus usuários. A comunicação entre os usuários pode ser alcançada, por exemplo, através de notificação. As atualizações podem ser enviadas para um grupo de usuários que pode aprová-las ou rejeitá-las [NG92]. Em [HZ87], as notificações podem ser enviadas para os usuários que possuem uma tranca[4] sobre um objeto quando outra tranca é requisitada para o mesmo objeto ou quando o objeto é atualizado. Depois do recebimento da notificação, o usuário pode tomar as ações corretivas tais como a liberação do objeto requisitado ou a releitura do objeto atualizado.

[4]Em inglês *lock*

Diversas propostas, como em [US92, BKK85, FZ89, NRZ92, RKT+95], modelam hierarquias de grupo através de transações aninhadas. Uma transação de grupo gerencia as interações entre os usuários em um contexto compartilhado. As ações dos usuários devem ser refletidas para este contexto geralmente de maneira assíncrona. A propagação de mudanças de um modo assíncrono é implementado por protocolos check-in/check-out [US92, BKK85]. Os dados podem ser transferidos da área pública para a área privada da transação (check-out) e levados para a área pública quando a transação termina as atualizações necessárias na área privada (check-in). Em outras propostas, estas ações são imediatamente propagadas para todos os usuários no grupo [EGR91]. Neste trabalho, a cooperação baseada em transações foi criticada e substituída pelo conceito de sessões. Neste caso, os usuários podem entrar ou deixar uma sessão em qualquer momento e podem acessar os objetos em um contexto compartilhado.

Outras críticas vêm da área de *workflow* com relação a não adequação de modelos de transação avançada em operar sistemas de trabalho reais [AAA+96, Moh94], afirmando que modelos de workflow podem atender um campo mais vasto de requisitos.

Fluxo de Controle

Tradicionalmente, o fluxo de controle entre os passos de uma aplicação é gerenciado pela própria aplicação. Deste modo, uma computação de longa duração poderia ser estruturada como uma seqüência de transações de curta duração sem suporte à sincronização ou recuperação de falhas englobando os passos envolvidos.

Gradualmente, este suporte apareceu em modelos como [WR92, ZNBB94, DHL90, GMS87]. Sagas [GMS87], por exemplo, provê fluxo de controle e recuperação de falhas baseada em compensação. Outras abordagens mais flexíveis baseadas na especificação de dependências entre as transações são encontradas em [ZNBB94, DHL90]. O modelo Contract [WR92] provê duas noções independentes de suporte de atividades complexas: os passos que são as unidades elementares de trabalho e os *scripts* que descrevem a estrutura de uma atividade em termos de seqüência, desvios, laços e construtores paralelos.

1.1 Objetivos

Este trabalho apresenta um Serviço de Transações Cooperativas que atende os requisitos de aplicações cooperativas através do gerenciamento de transações. Este gerenciamento garante consistência de dados e recuperação de falhas com um benefício adicional de possibilitar a estruturação destas aplicações como transações hierárquicas. A cooperação

entre os usuários é alcançada, na medida em que trabalhos em andamento são transferidos entre eles.

O Serviço possui as seguintes características:

- **consistência:** a manutenção de consistência é garantida pelos mecanismos de controle de concorrência baseado em trancas e recuperação de falhas baseada em *logs*.
- **flexibilidade:** a exigida flexibilidade para modelar grupos hierárquicos trabalhando em cooperação é provida pelo aninhamento de transações. Há dois tipos de transações, a saber: de grupo e de usuário.
- **visibilidade:** os usuários trabalhando de modo cooperativo podem manipular tarefas não finalizadas de outros usuários através da requisição de operações especiais, que permite cópia, empréstimo ou concessão de objetos.
- **distribuição:** o devido suporte às transações distribuídas é alcançado utilizando-se a camada ORB de CORBA.

O Serviço de Transações Cooperativas utiliza a camada ORB da arquitetura CORBA, Java como linguagem de programação e OrbixWeb 3.1 como plataforma de comunicação.

1.2 Organização

O conteúdo desta dissertação está distribuído em seis capítulos:

1. **Introdução.** Descreve a motivação e o contexto no qual o trabalho se encontra. Alguns problemas importantes de transações avançadas são discutidos. O Serviço de Transações Cooperativas é introduzido e suas principais características são descritas.
2. **Modelos de Transação para Aplicações Cooperativas.** Vários modelos de transações avançadas são apresentados. Estes modelos definem mecanismos para o compartilhamento de dados entre transações de longa duração, sem permitir, no entanto, que os dados fiquem inconsistentes. No final, uma breve comparação é realizada entre os modelos levando-se em consideração suas principais características.
3. **Arquitetura Distribuída Orientada a Objetos.** Neste capítulo, o padrão e a arquitetura distribuída de CORBA são analisados. O conceito de um ORB é discutido, bem como, as características principais da linguagem Java. Ambos são descritos como tecnologias que se complementam.

4. **Serviço de Transações Cooperativas.** Apresenta em detalhe o Serviço de Transações Cooperativas. No final, um exemplo interessante de sua utilização é descrito.

5. **Implementação do Serviço de Transações Cooperativas.** Este capítulo explica como a implementação do Serviço de Transações Cooperativas foi realizada. Primeiro, é esclarecido qual o ORB utilizado. Depois, são feitas algumas considerações sobre a implementação. Por fim, é mostrado como os testes foram desenvolvidos e quais foram os resultados alcançados.

6. **Conclusões.** Apresenta um resumo do que foi desenvolvido, ressalta as contribuições do trabalho e a partir deles analisa futuras linhas de pesquisa.

Capítulo 2

Modelos de Transação para Aplicações Cooperativas

2.1 Introdução

Aplicações avançadas necessitam do uso de transações de longa duração e caracterizam-se por desempenhar tarefas complexas num ambiente de trabalho de equipe. A complexidade destas aplicações reside no fato de os usuários operarem sobre um número relativamente grande de objetos por um longo período de tempo e precisarem interagir entre membros do mesmo projeto. Para tanto, é preciso um modelo de transações que forneça ferramentas para o trabalho cooperativo sobre dados compartilhados e que também proteja a equipe de acessos externos.

Em ambientes cooperativos times de usuários executam concorrentemente tarefas diferentes interagindo entre si, e essas interações, principalmente entre os usuários de um mesmo time, devem ser regulamentadas para assegurar que o fluxo de informação não seja incorreto e inconsistente e que o objetivo da colaboração seja alcançado. A cooperação é alcançada pela troca controlada dos conteúdos no espaço de trabalho dos usuários e pela instalação dos resultados das atividades no banco de dados comum do grupo.

Modelos de transações cooperativas geralmente definem um conjunto de participantes cooperativos num grupo e sempre permitem que a manipulação dos dados compartilhados seja feita de modo consistente.

Sistemas de Gerenciamento de Banco de Dados (SGBD), por exemplo, garantem o acesso aos dados de modo consistente fornecendo seriabilidade. Porém, não permitem que grupos de usuários interajam entre si compartilhando os dados. Neste caso, o critério de correção é restritivo para o trabalho cooperativo. Para solucionar tal problema, mecanismos como versão ou controle de concorrência baseado em semântica da aplicação foram

desenvolvidos.

Do ponto de vista de interação, um ambiente de desenvolvimento de projetos deve possibilitar a inclusão e a exclusão de usuários e permitir a colaboração entre usuários que trabalham sobre o mesmo conjunto de objetos.

Alguns modelos de transação mais representativos para aplicações cooperativas são analisados nas seções seguintes. Por fim, uma comparação é feita entre eles, levando-se em consideração alguns aspectos como controle de concorrência e colaboração.

2.2 O Modelo de Klahold

Esse modelo foi desenvolvido por Klahold et al [KSUW85] e baseia-se em conceitos como áreas de grupos de trabalho, versões e alternativas, que juntos aumentam a flexibilidade no gerenciamento de objetos e permitem paralelismo entre as atividades desenvolvidas pelos membros dos grupos. Para isso, diferentes modos de tranca[1] de objetos e transações de grupo e de usuário são introduzidos. A cooperação é alcançada através do modelo de transações e de operações para transferência de objetos entre as áreas de trabalho.

A seguir são apresentados os principais conceitos.

2.2.1 Transações de Grupo e de Usuário

Esse modelo permite estruturar uma aplicação como uma hierarquia de transações de dois tipos: transações de grupo e de usuário. As transações de grupo são associadas a um conjunto de usuários e a uma área de trabalho em comum, enquanto que as transações de usuário são associadas a cada participante do grupo e cada um deles tem uma área particular de trabalho. A área de trabalho comum contém os objetos utilizados pelo grupo e copiados de um banco de dados público, enquanto que a área de trabalho particular contém os objetos que só podem ser acessados por um determinado usuário.

Quando um usuário quer atualizar um objeto que está na área de trabalho comum, faz uma cópia do mesmo para sua área particular (operação de *check-out*). As atualizações sobre os objetos são feitas no banco de dados privado do usuário. Quando o usuário termina de usar um objeto é feita uma transferência do objeto para o banco de dados do grupo (operação de *check-in*). Existem algumas vantagens no mecanismo de check-in/check-out: o objeto inicial localizado na transação de grupo pode ser lido em paralelo pelas transações de usuário, as modificações são feitas só no espaço privado ficando transparentes aos outros usuários e as fases críticas de concorrência só acontecem na execução do check-in e do check-out.

[1] Em inglês *locking modes*

No momento em que é realizada uma operação de cópia (check-out) o modo de tranca deve ser especificado e deve ser igual ou mais fraco do que o modo especificado na cópia do banco de dados público, ou seja, não pode ser mais restritivo. As transações de grupo são necessariamente serializáveis e as de usuário, dependendo da sua intervenção, podem não ser.

A colaboração dentro do grupo é realizada disponibilizando resultados parciais na área do grupo através da operação de Release (que realiza check-in) ou passando estes resultados diretamente para algum usuário específico através das operações Grant ou Pass. Além disso, é possível trabalhar em paralelo produzindo-se soluções alternativas para os mesmos objetos.

2.2.2 Versões

O modelo de versão auxilia as transações no processo de documentação, na medida em que cada objeto tem um histórico de versões associado. Isso permite que um projeto seja modificado a partir de versões anteriores e que projetos alternativos sejam desenvolvidos em paralelo. Para tanto, um objeto é representado como um grafo acíclico, onde os nós são as versões e os arcos são os caminhos de derivação. Este grafo possui um nó raiz que representa a versão inicial.

É possível criar novas versões, modificar ou apagar as existentes. A operação de modificação só muda o nó especificado, enquanto que as outras operações alteram a estrutura do grafo. Quando na criação de uma nova versão a partir de várias versões, a operação de integração[2] deve ser feita pela aplicação ou pelo usuário e não pelo gerenciador de versões.

Os grafos são construídos para cada nível de trabalho: de banco de dados público (grafo global), de grupo e de usuário. O grafo de grupo habilita a comunicação no grupo e documentações de curta duração.

2.3 O Modelo de Fernandez

Esse modelo foi desenvolvido por Fernandez e Zdonik [FZ89] e utiliza o banco de dados orientado a objetos ObServer [HZ87], que estende o conjunto de trancas tradicionais e as operações comuns de banco de dados. A cooperação no ObServer é realizada através do compartilhamento do trabalho em andamento e das notificações geradas entre as transações. O modelo abandona a seriabilidade entre as transações que desenvolvem trabalho cooperativo, mas não entre as demais transações. Neste contexto é que grupos de transação são inseridos.

[2]Em inglês *merge*

Os grupos de transação são organizados hierarquicamente como uma árvore, onde os nós internos são grupos, as folhas representam as aplicações de projeto e a raiz é o banco de dados permanente compartilhado. Os protocolos de entrada de um grupo descrevem a cooperação entre seus membros, quais trancas são permitidas e como os objetos são salvos e recuperados. Já o protocolo de saída determina o tipo de interação do grupo com o nó pai, quando ocorre a propagação das mudanças dos objetos e quais trancas podem ser requisitadas do pai. Os modos de tranca oferecidos pelo ObServer consistem de um par ordenado contendo o modo de tranca e o tipo de comunicação.

A extensão das trancas e os protocolos das transações são analisados a seguir.

2.3.1 O Mecanismo de Tranca Estendido

Esse modelo introduz mais uma dimensão às trancas para permitir a comunicação entre as transações. Assim, a tranca é um par ordenado do tipo *(modo da tranca, modo de comunicação)*. O primeiro parâmetro indica se o objeto é para leitura ou escrita. O modo de comunicação especifica se a transação quer ser notificada quando uma transação atualiza ou requisita uma tranca sobre um objeto. Os modos de tranca e comunicação estão detalhados nas tabelas 2.1 e 2.2:

Modos de Tranca	
NULL	Permite apenas modo de comunicação.
NR-READ	Leitura não-restritiva. Permite que outras transações façam acesso de leitura e escrita não-restritiva no objeto.
R-READ	Leitura restritiva. Proíbe que outras transações executem escrita.
M-WRITE	Escrita múltipla. Permite que múltiplas transações realizem escrita sobre o objeto.
NR-WRITE	Escrita não-restritiva. Proíbe que outras transações obtenham trancas de leitura e escrita restritiva, mas permite leitura no modo NR-READ.
R-WRITE	Escrita restritiva. Provê acesso exclusivo. Outras transações não podem ler ou escrever no objeto.

Tabela 2.1: Modos de tranca.

2.3.2 Protocolos de Cooperação

Um grupo de transações controla um conjunto de transações cooperativas ou outros grupos de transação compondo uma árvore de transações de muitos níveis. Cada grupo tem associado um protocolo de entrada e um protocolo de saída. A interação entre as transações

Modos de Comunicação	
N-NOTIFY	Sem notificação, somente modos de tranca são efetivos.
U-NOTIFY	Informa a transação quando objeto foi atualizado.
R-NOTIFY	Informa o dono da tranca se outra transação requisitou uma tranca do tipo leitura.
W-NOTIFY	Informa o dono da tranca se outra transação requisitou uma tranca do tipo escrita.
RW-NOTIFY	Informa o dono da tranca se outra transação requisitou uma tranca do tipo leitura ou escrita.
UR-NOTIFY	Combinação do U-NOTIFY e R-NOTIFY.
UW-NOTIFY	Combinação do U-NOTIFY e W-NOTIFY.
URW-NOTIFY	Combinação do U-NOTIFY e RW-NOTIFY.

Tabela 2.2: Modos de comunicação.

filhas de um grupo de transações é definida pelo protocolo de entrada. A interação do grupo com seu pai é determinada pelo protocolo de saída do grupo.

O protocolo de entrada de um grupo é definido por um filtro de tranca, que define o conjunto de trancas que o grupo pode oferecer para os seus membros. As requisições das trancas sobre os objetos pelo grupo devem casar com o protocolo. Em um grupo totalmente cooperativo, o filtro pode incluir só trancas com algum tipo de notificação, enquanto que em grupos não cooperativos, só trancas restritivas sem modo de comunicação podem ser definidas. O protocolo também é definido por um filtro de operação, que especifica o conjunto de operações que um membro pode requisitar do grupo. Por exemplo, num grupo cooperativo, mudanças intermediárias de objetos podem ser transferidas para o grupo através de uma operação especial, entretanto, num grupo não cooperativo, esta operação pode violar a seriabilidade e deve ser omitida do filtro de operação.

Os protocolos de saída são responsáveis pela transformação das trancas requisitadas pelos membros para trancas permitidas pelo pai do grupo. Portanto, quando um objeto é requisitado e o grupo não possui uma tranca sobre ele ou a tranca não for suficientemente forte, o grupo deve requisitar o objeto de seu pai e possivelmente transformar a tranca requisitada para uma tranca existente no filtro do pai. O protocolo de saída também determina quando os objetos modificados pelo grupo podem ser propagados para o pai. As regras de propagação permitem que objetos sejam liberados quando todos os membros terminarem, quando um objeto não estiver mais sendo efetivamente usado ou ainda antecipadamente através da operação Register. Por exemplo, uma transação que tem protocolos de entrada cooperativos e quer obter objetos com trancas restritivas converte uma requisição de tranca do tipo NR-READ/U-NOTIFY feita por um membro para uma requisição de tranca R-READ/NULL do pai. O grupo adquire direitos de leitura restritiva

e fornece direitos de leitura não restritiva e notificações de atualização para seus membros.

2.4 O Modelo de Agrawal

O modelo criado por Agrawal et al [ABAK95] soluciona alguns problemas no processo de colaboração apresentando um protocolo de coordenação, que permite a execução de transações colaborativas chamadas co-ações. A cooperação do modelo é alcançada usando-se um canal de colaboração entre os usuários. O protocolo de tranca desenvolvido é uma adaptação do protocolo de duas fases rígido [BHG87], garantindo a execução relativamente serializável das co-ações. Esse protocolo utiliza as especificações resultantes dos canais de colaboração para forçar execuções corretas no sistema.

A seguir apresentamos a noção de atomicidade relativa de acordo com os canais de colaboração entre as co-ações, as propriedades das execuções corretas e o protocolo de tranca.

2.4.1 A Atomicidade Relativa

Os objetos do banco de dados cooperativo são acessados por operações atômicas de leitura e escrita e os usuários interagem com o banco invocando co-ações. Cada co-ação é uma seqüência dessas operações. A operação de leitura (ou escrita) executada pela co-ação C_i no objeto x é denotada por $r_i[x]$ (ou $w_i[x]$).

A execução S sobre $C = C_1, \ldots, C_n$ é uma seqüência entrelaçada de todas as operações das co-ações em C tal que as operações das co-ações C_i aparecem na mesma ordem em S como em C_i, onde $i = 1, \ldots, n$.

Em bancos de dados tradicionais a noção de atomicidade é absoluta, isto é, cada transação aparece como uma unidade atômica em relação às outras transações. Em ambientes de colaboração, a noção de atomicidade é relativa permitindo aos usuários a comunicação através de canais de colaboração entre pares de co-ações.

No modelo de atomicidade relativa uma co-ação tem múltiplas especificações de atomicidade definidas com respeito a todas outras co-ações conforme o tipo de interação que ocorrer entre os usuários através de canais de colaboração.

A unidade atômica de C_i relativa a C_j é uma seqüência de operações de C_i tal que nenhuma operação de C_j pode executar dentro desta seqüência. Entretanto, C_i pode estabelecer um canal de colaboração em relação a C_j e permitir operações de C_j antes e depois de cada unidade atômica. A seqüência ordenada de unidades atômicas de C_i em relação a C_j é denotada por $Atomicity(C_i, C_j)$ e a k-ésima unidade atômica em $Atomicity(C_i, C_j)$ é denotada por $AtomicUnit(k, C_i, C_j)$. O exemplo a seguir mostra que as operações de

C_2, se executadas entrelaçadas com as operações de C_1, só podem ser executadas depois de $r_1[y]$ e antes de $w_1[x]$.

$Atomicity(C_1, C_2) : \boxed{r_1[x]r_1[y]}\ \boxed{w_1[x]}$, onde
$C_1 = r_1[x]r_1[y]w_1[x]$,
$C_2 = r_2[x]r_2[y]w_2[y]$,
$C_3 = r_3[x]r_3[y]$,
x, y são objetos do banco de dados

A operação o de uma co-ação C_j é entrelaçada com uma $AtomicUnit(k, C_i, C_j)$ numa execução S, se há operações o' e o'' de $AtomicUnit(k, C_i, C_j)$ tal que o' precede o, que por sua vez precede o''.

2.4.2 Execuções Corretas

O modelo provê duas noções para detectar como as co-ações podem interferir entre si: conflito e dependência. A noção de conflito tem o mesmo significado de conflito dentro do controle de concorrência tradicional, ou seja, duas operações conflitam se elas acessam o mesmo objeto e pelo menos uma delas é uma operação de escrita. Duas execuções são equivalentes com relação ao conflito se ambas ordenam operações conflitantes da mesma maneira. A noção de dependência é derivada do conceito de conflito e da estrutura interna das co-ações. Ela descreve o fluxo de informação que ocorre entre co-ações como o resultado de uma seqüência de conflitos. Uma operação o_2 é dependente diretamente de o_1 em S se o_1 precede o_2 em S e, ou o_1 e o_2 são operações da mesma co-ação ou o_1 conflita com o_2. A relação "é dependente de" é o fechamento transitivo da relação "é dependente diretamente de".

Se uma operação o_{ij} é a j-ésima operação da co-ação C_i definimos S uma execução *relativamente serial* sobre C para todas co-ações C_i e C_l, se uma operação o_{ij} de C_i é entrelaçada com uma $AtomicUnit(k, C_l, C_i)$ para algum k, então o_{ij} não é dependente de nenhuma operação $o_{lm} \in AtomicUnit(k, C_l, C_i)$ e nenhuma $o_{lm} \in AtomicUnit(k, C_l, C_i)$ é dependente de o_{ij}.

A noção de execução relativamente serial é análoga à noção de execução serial na teoria da seriabilidade. Existem execuções que não são relativamente seriais mas comportam-se como tal. Essas execuções são chamadas *relativamente serializáveis*. Uma execução é relativamente serializável se é equivalente com relação ao conflito a alguma execução relativamente serial.

Se usarmos as co-ações do exemplo anterior e considerando-se que $Atomicity(C_1, C_3) : \boxed{r_1[x]r_1[y]w_1[x]}$, a execução

$$S = r_1[x]r_1[y]r_3[x]r_3[y]w_1[x]$$

não é relativamente serial já que $r_3[x]$ é entrelaçada com a $AtomicUnit(1, C_1, C_3)$ e $w_1[x]$ é dependente de $r_3[x]$. Entretanto, S é relativamente serializável já que é equivalente com relação ao conflito à seguinte execução relativamente serial:

$$S = r_3[x]r_1[x]r_1[y]r_3[y]w_3[z]w_1[x].$$

O modelo de atomicidade relativa é uma generalização do modelo tradicional e se as especificações são tais que cada co-ação é uma unidade atômica única com respeito a todas as outras, então a atomicidade relativa se reduz à atomicidade absoluta.

2.4.3 O Protocolo de Tranca

Uma co-ação obtém uma tranca exclusiva sobre um objeto se e somente se nenhuma outra co-ação tem alguma tranca sobre o mesmo e uma tranca compartilhada somente se todas as outras trancas sobre o objeto são compartilhadas. Todas as trancas adquiridas pelas co-ações são liberadas no fim de suas execuções.

Neste modelo quando uma operação o é entrelaçada numa unidade atômica de uma co-ação C as condições abaixo devem ser obedecidas:

- o não depende de nenhuma operação da unidade atômica.
- Nenhuma operação da unidade atômica depende de o.

Se o' de C foi executada e queremos garantir a primeira condição, o protocolo deve retardar a execução de o se resulta numa dependência de o' para o até que a unidade atômica tenha completado.

Quando o entrelaça em uma unidade atômica e depois o' dessa unidade vai ser executada (o' depende de o), então a execução de o' violará a segunda condição anterior. O protocolo escolhe abortar o. Note que, se atrasarmos a execução de o', a condição dois continuaria não satisfeita.

2.5 CoAct

O modelo de atividade cooperativa desenvolvida por Rusinkiewicz et al [RKT+95], conhecido como CoAct, tem propriedades transacionais para cenários cooperativos. Nas aplicações avançadas, a idéia é não restringir acessos aos recursos compartilhados, mas basear-se na troca semanticamente correta de informação entre as atividades concorrentes. Ao invés de atomicidade, o modelo garante que as atividades executem de acordo com regras de execução definidas pelo projetista da atividade e finalizem somente em estados válidos

previamente especificados. Ao invés de isolamento, o modelo permite que informações sejam trocadas entre usuários concorrentes, mas assegura que os resultados da execução das atividades sejam equivalentes à execução hipotética da atividade por um único usuário.

2.5.1 Atividades Cooperativas

A infra-estrutura apresentada pelo modelo provê atividades cooperativas e atividades de usuários, cada uma com sua área de trabalho correspondente. Uma atividade cooperativa é uma instância de um tipo de atividade cooperativa e consiste de subatividades que podem ser outras atividades cooperativas ou uma atividade elementar. Na especificação do tipo ainda constam um conjunto de pontos de parada[3] que definem os estados onde a execução pode ser interrompida, um conjunto de regras de execução que governam a execução das atividades e um conjunto de regras de entrelaçamento que define as dependências entre as subatividades.

O conjunto de pontos de parada define os estados onde a execução da atividade pode ser interrompida. Uma vez que este ponto é alcançado, as restrições de consistência interna são satisfeitas, correspondendo a um estado intermediário semanticamente consistente.

O conjunto de regras de execução consiste de duas partes: execução para frente, que governa a execução regular de uma atividade cooperativa e execução para trás, que conduz o cancelamento da atividade se ela não foi terminada e precisa ser abortada. Para garantir as regras de execução nas atividades de usuário em tempo de execução, precisa-se de um escalonador de atividades, que recebe as requisições dos usuários e permite que estas executem de acordo com a especificação.

Já o conjunto de regras de entrelaçamento define a existência de dependências, que não estão nas regras de execução, entre as subatividades em função do fluxo de dados. Estas regras são definidas em função de um predicado de compatibilidade $\otimes : A \times A \rightarrow \{True, False\}$ entre pares de subatividades de uma atividade cooperativa. A avaliação do predicado pode depender do estado do sistema e dos valores de parâmetros das subatividades. A compatibilidade é usada para determinar as dependências entre as subatividades executadas por um único usuário e para decidir se duas execuções da mesma subatividade por diferentes usuários podem ambas estar presentes na integração (possível se elas forem compatíveis). Um exemplo das regras de entrelaçamento pode ser visualizado a seguir:

$$\otimes(editar(cap1), editar(cap2)) = (cap1 \neq cap2)$$
$$\otimes(anotar(cap1), anotar(cap2)) = True$$

O primeiro predicado especifica que se dois usuários querem editar um capítulo na mesma atividade, para que a integração seja realizada com sucesso os capítulos devem ser

[3] Em inglês *breakpoints*

diferentes. No segundo caso, os usuários podem anotar qualquer capítulo, já que essas operações são compatíveis.

Usuários dentro de uma atividade cooperativa executam atividades no seu espaço de trabalho e podem integrá-las quando atingem pontos de parada.

O modelo permite a troca de informação direta entre os usuários como também indireta através da área de trabalho comum. A integração das atividades ocorre no momento em que estas informações são trocadas entre os usuários.

CoAct define algumas facilidades de coordenação entre os usuários cooperativos como um conjunto de funções que implementa os protocolos de coordenação pré-definidos. Estas funções estão disponíveis no nível de interface de usuário e são independentes do tipo de atividade cooperativa. Entre as funções disponíveis podemos citar a função Preempt que elimina um usuário da atividade cooperativa e passa seu trabalho para outro, a função RequestCooperation que permite a um usuário participar de uma atividade e a função RequestExclusiveAccess que permite a um usuário assumir as responsabilidades de outro.

2.6 O Modelo de Nodine

Esse modelo foi desenvolvido por Nodine et al [NRZ92] e introduz uma hierarquia de transações cooperativas e grupos de transação, um controle de concorrência baseado em padrões e conflitos e um mecanismo de recuperação baseado em operações.

A hierarquia de transações cooperativas é um conjunto estruturado de transações cooperativas que reflete a decomposição hierárquica das tarefas. Os nós internos da hierarquia são grupos de transação e os nós-folha são transações cooperativas. Um grupo de transação raiz está no topo da hierarquia e sempre existe. Cada grupo contém um grupo de membros que cooperam numa tarefa. O grupo de transação controla ativamente a interação entre os membros que podem ser uma transação cooperativa ou outro grupo.

Cada grupo tem sua própria especificação de correção baseada na noção de padrões e conflitos. Os padrões definem a forma de execução ordenando as operações dos membros, enquanto que os conflitos especificam como as operações dos membros são compatíveis entre si.

O grupo de transações tem um conjunto local de versões de objetos. Pode haver inúmeras versões espalhadas na hierarquia. Para um objeto específico, a versão no grupo raiz é a mais antiga e aquelas abaixo na hierarquia mais recentes. A versão de um objeto do grupo é acessível por qualquer descendente na hierarquia. Além disso, um grupo deve ter sua própria área de armazenamento da qual pode-se recuperar as cópias dos objetos depois de uma falha de sistema.

A versão do objeto é automaticamente copiada para o grupo de transação quando um membro desse grupo inicia uma operação sobre o objeto. Quando alguma subtarefa

está completa, permite-se que o membro do grupo de transação realize um ponto de verificação[4], propagando os efeitos das operações realizadas no grupo para o pai do grupo. O mapeamento de seqüências de operação para seqüências equivalentes é realizado no ponto de verificação e é específico para o tipo de objeto e as operações que ele fornece. É permitido para o membro desfazer as operações individuais que ainda não passaram pelo ponto de verificação.

Quando uma operação é submetida para o grupo, ela deve ser aceita antes de ser executada de acordo com o critério de correção. Posteriormente, o membro pode abortá-la ou efetivá-la usando o ponto de verificação. O ato de se chegar ao ponto de verificação não garante que a operação não possa ser cancelada. Por exemplo, se alguma outra operação for cancelada, a operação que chegou até o ponto de verificação pode ser cancelada se for dependente da primeira.

O controle de concorrência baseado em padrões e conflitos e o mecanismo de recuperação baseado em operações são descritos nas seções que seguem.

2.6.1 Padrões e Conflitos

O critério de correção para um grupo de transação descreve histórias válidas. Uma história é uma seqüência de invocações de operação realizadas pelas transações cooperativas sobre versões de objetos e é mantida no *log* do grupo de transação. A história de um grupo de transação é refletida para a história do pai do grupo e é correta se satisfizer seu próprio critério e se as histórias para cada filho também forem corretas.

Uma especificação de correção para um grupo consiste de um conjunto de gramáticas de padrão e conflito. Este conjunto varia no tempo quando os membros são incluídos ou removidos do grupo e quando os diferentes membros interagem com os objetos.

Os padrões especificam os entrelaçamentos de operações que devem acontecer para que a execução no grupo seja correta. Uma história é correta quando ela segue as especificações de padrão e não contém conflitos.

2.6.2 A Recuperação Baseada em Operações

A recuperação de falhas é baseada em operações, isto é, o mecanismo de recuperação remove os efeitos de operações individuais bem como das operações dependentes. Esse tipo de recuperação mantém as dependências entre as operações no log, além das histórias dos grupos de transação.

Existem alguns tipos de dependências consideradas: dependências de padrão, dependências de escrita-leitura, dependências de leitura e dependências entre pai-filho.

[4]Em inglês *checkpoint*

Para cada padrão definido num grupo de transação, dependências de padrão são formadas entre as operações que participam no padrão.

As dependências de escrita-leitura são mantidas em cada versão de objeto criando-se uma entrada no log quando uma operação alterou a versão.

A leitura de uma versão de objeto do membro M é somente correta se a operação que escreveu sobre a versão também é correta. Se a operação de escrita depois torna-se inválida, então a operação de leitura de M lê informação incorreta e é também inválida. Estas são as dependências de leitura[5].

As dependências entre pai-filho[6] ocorrem entre as operações de níveis adjacentes na hierarquia de transação. Uma leitura de um membro M sobre um objeto o é correta somente se a leitura do grupo foi feita sobre uma versão correta do mesmo. Se a versão posteriormente for invalidada a leitura de M também será. Análogo vale para a operação de escrita.

Quando uma operação é abortada, é necessária a invalidação de outras operações. Isso exige que o log seja percorrido cronologicamente e as operações com as características abaixo sejam invalidadas:

- operação dependente de alguma operação abortada.
- operação que entra em conflito devido às diferenças entre a nova e a velha histórias.

Uma vez que as operações foram invalidadas, todas as versões de objetos escritas pelas operações de escrita inválidas são eliminadas do banco de dados.

No fim da fase de varredura do log cada membro afetado pelo cancelamento recebe uma mensagem de invalidação, notificando quais operações foram invalidadas.

Na fase de recuperação, os membros usam estas mensagens como ponto de partida para o trabalho de recuperação. Neste momento, os membros podem fazer releitura de versões, submetendo operações de compensação. A releitura de versões inválidas pode determinar qual trabalho foi invalidado e o que pode ser preservado. Se o membro escolher abortar quaisquer operações que não passaram pelo ponto de verificação, ele deve iniciar ações de recuperação.

2.7 Comparação

Os modelos descritos acima podem ser comparados em função de alguns aspectos como a estrutura de transações, o controle de concorrência, a recuperação de falhas e as facilidades de cooperação.

[5]Em inglês *reads-from dependencies*
[6]Em inglês *parent-child dependencies*

O controle de concorrência apresentado pelos modelos difere dos existentes para aplicações tradicionais pelo fato de modelarem a execução das transações de acordo com o grau de consistência e a cooperação exigidos pelas aplicações. Tanto mecanismos de tranca estendidos quanto aqueles baseados em semântica de aplicação foram introduzidos.

A recuperação descrita por alguns modelos é realizada por mecanismos que cancelam as transações ou provêem formas de compensações.

Os modelos criaram soluções para o compartilhamento dos dados de aplicações colaborativas usando canais de colaboração, versões, operações especiais para troca de informações e trancas com notificações.

2.7.1 A Estrutura das Transações

Modelos como de Klahold, de Fernandez e de Nodine [KSUW85, FZ89, NRZ92] introduzem a idéia de uma organização hierárquica com áreas de trabalho para as transações. A hierarquia consiste de dois tipos de transações, de grupo e de usuário. Nós internos representam transações de grupo e as folhas transações de usuário. Essa hierarquia pode conter múltiplos níveis [FZ89, NRZ92] ou dois níveis em Klahold.

O modelo de Agrawal define co-ações que são seqüências de operações não apresentando níveis hierárquicos. Para o modelo CoAct, existe uma composição de atividades cooperativas. As transações são atividades cooperativas ou atividades de usuários. Uma atividade cooperativa neste modelo é uma instância de um tipo de atividade cooperativa e consiste de subatividades que podem ser outras atividades cooperativas ou uma atividade elementar. Também possui área de trabalho para cada nível.

2.7.2 Mecanismos de Tranca Estendidos

O modelo de Klahold introduz novos modos de tranca para facilitar a cooperação e possibilitar a manutenção de versões de objetos. São definidas quatro operações sobre as versões: leitura, derivação, modificação e remoção. Para modificar e derivar uma versão, é preciso ler a versão original e portanto o direito de modificar ou derivar inclui o direito de ler. Além disso, o direito de modificar inclui também o direito de derivar versões. Já a remoção exige que trancas temporárias sejam colocadas nos antecessores e sucessores da versão a remover. Isso é feito automaticamente pelo sistema. Assim, cinco modos básicos englobam todas as operações: R-lock, RD-lock, DX-lock, X-lock e DS-lock. Os tipos R-lock e RD-lock permitem leitura para o possuidor da tranca. Os tipos DS-lock e DX-lock permitem leitura e derivação. E, finalmente, o tipo X-lock permite modificação, leitura e derivação, garantindo o acesso exclusivo. Leituras com RD-lock podem ser realizadas em paralelo com derivação (DS-lock ou DX-lock). Derivações também podem ser realizadas em paralelo com DS-lock.

As trancas são obtidas nas operações de *check-out*. Na transação de grupo, as trancas podem ser convertidas para trancas mais fortes na fase de aquisição e enfraquecidos na fase de liberação. As transações de usuário, por sua vez, não precisam ter trancamento em duas fases, permitindo, assim, maior grau de flexibilidade, em detrimento da seriabilidade. Se a tranca requisitada for incompatível, ela é rejeitada. Assim, evita-se o estado de espera, bem como *deadlocks*.

A granularidade no controle de concorrência do modelo de Klahold pode ser tanto uma versão quanto um subgrafo de versões.

Já o modelo de Fernandez estende os mecanismos tradicionais de trancas com a introdução de mais uma dimensão às trancas, para permitir comunicação entre as transações. Além disso, permite a definição de filtros para cada nível de transação, facilitando a cooperação ou garantindo o isolamento.

No modelo de Agrawal, o controle de concorrência pode ser reduzido ao protocolo de duas fases[7] rígido. O protocolo do modelo falha em aceitar algumas execuções que o protocolo de duas fases aceitaria, mas permite o estabelecimento de canais colaborativos entre co-ações. A unidade atômica de C_i relativa a C_j, por exemplo, é uma seqüência de operações de C_i tal que nenhuma operação de C_j pode executar dentro desta seqüência. Entretanto, C_i pode estabelecer um canal de colaboração em relação a C_j e permitir operações de C_j antes e depois da seqüência.

2.7.3 Utilização de Semântica da Aplicação

No domínio das aplicações cooperativas, o critério de correção baseado em seriabilidade não é apropriado. O conhecimento sobre a aplicação e os dados por ela manipulados podem ajudar na definição de critérios mais flexíveis de correção.

CoAct tem um modelo de controle de concorrência baseado na especificação de regras de execução e de regras de entrelaçamento. O uso de pontos de parada para determinar os estados onde a execução pode ser interrompida e a utilização de predicados de compatibilidade entre as subatividades do modelo permitem que a execução da aplicação seja semanticamente correta.

Para Nodine [NRZ92], o usuário deve utilizar o conhecimento semântico sobre a aplicação para definir o critério de correção. A definição da correção de um grupo de transação é feita em termos de gramáticas de padrão e conflito. Assim, as gramáticas de padrão são usadas para definir como as operações são ordenadas para formar sentenças corretas. A transação que submete uma operação é imediatamente informada se a operação é inválida, isto é, se causa conflito. O mecanismo que verifica a correção não se preocupa em só reconhecer histórias corretas, mas também prefixos válidos destas

[7] Em inglês *two-phase locking*

histórias.

2.7.4 Recuperação de Falhas

A propriedade de atomicidade do modelo tradicional não é adequada para transações de longa duração. Alguns modelos de transações precisam disponibilizar seus resultados antecipadamente. Isso pode provocar cancelamentos em cascata. Em [RKT+95], subatividades externalizam seus efeitos, exigindo que o cancelamento de uma atividade seja realizado através de compensações. A especificação de uma atividade cooperativa deve conter além das regras de execução das subatividades, as regras de execução para o cancelamento de atividade através de compensações.

Em [NRZ92] a unidade de recuperação é uma operação e não uma transação. No log são incluídas, além das operações em si, todas as formas de dependências entre elas. Assim, o cancelamento de uma operação exige o cancelamento dos efeitos de todas as operações dependentes. Nesse modelo, o mecanismo de recuperação é mais caro, mas é também mais flexível.

2.7.5 Mecanismos para Troca de Informação

Num ambiente cooperativo, é importante aumentar a visibilidade entre os usuários. No modelo de Klahold, é possível emprestar ou transferir versões de objetos entre eles. O empréstimo da versão **V** para o usuário **U** passa todos os direitos adquiridos pelo usuário original, exceto a possibilidade de transferência para o banco de dados do grupo. Tão logo **U** termine de trabalhar sobre o objeto, passa o controle de **V** de volta para o usuário anterior. Se um usuário não quiser mais manipular uma versão, pode transferi-la para outro, inclusive passando o direito de escrita no banco de dados do grupo. Se uma transação emprestou versões individuais para outras transações e elas não foram devolvidas, o fim de transação será rejeitado. Se a transação pediu emprestadas versões e ainda não as retornou, quando o fim de transação é requisitado, as modificações por ela realizadas são transferidas para o usuário original.

No modelo de Nodine, a interação entre os usuários de um grupo é definida por padrões e conflitos. Qualquer usuário de um grupo pode requisitar operações para o grupo concorrentemente com outros membros do grupo. Essas operações serão aceitas se estiverem de acordo com os padrões especificados e se não causarem conflitos.

Os autores do modelo CoAct introduziram mecanismos de troca de resultados, como Import e Delegate, que permitem a comunicação de usuários que trabalham em cooperação. O mecanismo Import, por exemplo, importa (partes de) uma atividade de usuário para o espaço de trabalho da atividade do usuário corrente, exigindo que a atividade do usuário realize a operação de integração das partes importadas de acordo com as

regras de execução e de entrelaçamento para criar uma versão consistente. Essencialmente, a integração pode ser vista como a incorporação de histórias ordenadas parcialmente. A integração das histórias obedece a ordem parcial de cada história e, se a intersecção delas não é vazia ou as subatividades são incompatíveis, algumas subatividades precisam ser eliminadas conforme escolha do usuário.

O modelo de Fernandez oferece operações especiais sobre os objetos e as trancas com notificações para melhorar a visibilidade. A operação Register, por exemplo, realizada por uma transação que possui uma tranca do tipo escrita sobre o objeto, torna a versão do objeto visível para todas outras transações. Já a operação Commit instala o objeto no banco de dados e libera a tranca sobre ele, mas não garante que a transação valide[8]. Para liberar a tranca sem modificar o objeto, a transação pode abortar as atualizações sobre o objeto. Quando a transação valida ou aborta, todos as trancas são liberadas e os objetos modificados são instalados ou descartados.

As notificações ajudam os usuários a ficar cientes das atividades de outros usuários. Por exemplo, um usuário pode receber uma notificação quando um segundo usuário requisita um objeto por ele alocado. No caso de já ter terminado o trabalho, ele pode liberá-lo permitindo que o outro usuário inicie seu trabalho.

[8]Em inglês *commit*

Capítulo 3

Arquitetura Distribuída Orientada a Objetos

3.1 Introdução

O modelo de transações cooperativas da dissertação é orientado a objetos e baseado na arquitetura cliente/servidor. A orientação a objetos aliada ao modelo cliente/servidor apresentam amplas vantagens discutidas nas seções 3.2 e 3.3. Mais especificamente trataremos na seção 3.4 da arquitetura CORBA desenvolvida pela OMG que constitui padrão para objetos distribuídos. Na seção 3.5, são apresentados os benefícios da linguagem Java em ambientes CORBA. E na seção 3.6, os serviços de objetos de transação em CORBA são analisados.

3.2 Objetos Distribuídos

As metodologias tradicionais de desenvolvimento de software tratam as funções e os dados de uma aplicação de modo independente, enquanto que as metodologias orientadas a objetos não fazem tal distinção. Nessas metodologias, um objeto define tanto seu comportamento quanto seus atributos. Além disso, os objetos permitem o reuso de código, melhorando a produtividade de software.

Objetos definidos em C++, por exemplo, provêem facilidades de reuso de código através dos mecanismos de herança e encapsulamento, porém estes objetos só são reconhecidos pelo compilador da linguagem que os criou. Em contrapartida, um objeto distribuído contém trechos de código independentes que podem ser acessados por clientes remotos via invocações de métodos. O compilador e a linguagem usados para criar aplicações distribuídas são totalmente transparentes para os clientes, que não precisam

saber onde eles residem no sistema distribuído nem em qual sistema operacional eles estão executando.

A tecnologia de objetos distribuídos tornou possível o desenvolvimento de sistemas cliente/servidor robustos com altos níveis de transparência. Esta é a área de maior potencial e crescimento dos objetos distribuídos [OHE96].

3.3 Arquitetura Cliente/Servidor

Com o desenvolvimento das redes de alta velocidade a baixo custo, surgiram aplicações que consistem de transações de negócios internacionais (ex. bancos, lojas de departamento) e que demandam um volume cada vez maior de dados distribuídos.

Para que estas redes de computadores atendam estas novas demandas do mercado, algumas tecnologias são introduzidas [OHE96]:

- **Processamento de Transações Complexas.** Além de transações tradicionais[1], este ambiente cliente/servidor necessita de novos modelos como, por exemplo, transações aninhadas espalhadas em diversas máquinas e transações de longa duração.
- **Agentes Móveis.** Neste ambiente devem executar todos os tipos de agentes eletrônicos que podem vender produtos, colecionar informações para gerenciamento de sistema ou simplesmente coletar estatísticas sobre os usuários na rede.
- **Gerenciamento de Dados Complexos.** Superservidores devem prover repositórios para armazenamento e distribuição de documentos compostos de dados multimídia.
- **Auto-Gerenciamento Inteligente.** É preciso um software distribuído que seja capaz de auto-gerenciar-se e auto-configurar-se.
- **Middleware Inteligente.** O ambiente distribuído deve simular um sistema único sobre múltiplas máquinas híbridas para criar a ilusão de que todos os servidores conectados na rede comportam-se como um único sistema de computador. Usuários podem conectar ou desconectar-se dinamicamente e devem ser capazes de usar as mesmas convenções de nomes para localizar qualquer recurso na rede (por exemplo, aplicativos e impressoras) sem se preocupar com protocolos de comunicação.

Existem quatro paradigmas no desenvolvimento de aplicações com caráter cliente/servidor: Banco de Dados (BD) SQL, Monitores de Processamento de Transações,

[1] Em inglês *flat transactions*

Groupware e Objetos Distribuídos. Todos paradigmas criam aplicações completas, gerenciando processos e dados e introduzindo seu próprio *middleware* para interações cliente/servidor [OHE96]:

- **Cliente/Servidor com BD SQL**

  Os servidores de BD SQL são modelos dominantes na criação de aplicações cliente/servidor. Porém, a linguagem não é adequada para gerenciar dados complexos ou dados armazenados em servidores distribuídos. Para melhorar o desempenho em sistemas distribuídos, pode-se utilizar procedimentos armazenados[2], que encapsulam os comandos SQL num procedimento que reside no mesmo servidor do BD.

  Esse paradigma é adequado para um grupo restrito de aplicações executando preferivelmente em um ambiente de um único servidor.

- **Cliente/Servidor com Monitores de Processamento de Transação**

  Monitores de Processamento de Transação foram criados para gerenciar processos e organizar programas, dividindo as aplicações complexas em trechos de código chamados transações que podem executar independentemente. Os monitores permitem a execução de aplicações que servem milhares de clientes, provendo um ambiente que separa os clientes remotos dos recursos do servidor, gerencia transações, realiza balanceamento de execução e recuperação após falhas.

- **Cliente/Servidor com Groupware**

  Groupware é uma coleção de tecnologias que representa processos complexos que participam de atividades humanas colaborativas. Coleta dados não estruturados (texto, imagens, de correio eletrônico e outros) e organiza-os em documentos distribuídos. Um documento multimídia está para groupware assim como uma tabela está para um BD SQL. Sistemas de *workflow* e correio eletrônico são tecnologias relacionadas.

- **Cliente/Servidor com Objetos Distribuídos**

  Objetos Distribuídos são importantes num modelo cliente/servidor por permitir a divisão de aplicações desenvolvidas monoliticamente em componentes auto-contidos que podem executar conjuntamente em vários pontos e sobre sistemas operacionais diferentes. A orientação a objetos é adequada para a criação de sistemas cliente/servidor flexíveis, pois os dados e a lógica estão encapsulados no próprio

[2] Em inglês *stored procedures*

objeto, permitindo o auto-gerenciamento e a manutenção independente dos componentes de software envolvidos. Outros benefícios destes objetos são os componentes reutilizáveis, a interoperabilidade, a portabilidade e a coexistência com aplicações legadas.

3.4 CORBA

CORBA (Commom Object Request Broker Architecture) [COR02] é uma plataforma distribuída aberta desenvolvida pela OMG (Object Management Group) [OMG] baseada em objetos e estruturada em várias camadas. CORBA possibilita a construção e a integração de aplicações na medida que alcança as propriedades abaixo, presentes em sua especificação e em sua arquitetura:

- Os objetos devem ser tão simples de criar quanto em linguagens orientadas a objetos.
- A interface IDL (Interface Definition Language) de CORBA deve ser simples de ser mapeada e não deve dispender mais esforço de programação do que em linguagens orientadas a objetos.
- O acesso aos objetos deve ser eficiente.
- Os objetos devem ser de qualquer tamanho.
- Os objetos podem ser criados em quaisquer sistemas operacionais comerciais existentes.
- Deve ser possível utilizar várias linguagens de programação.
- Um objeto deve estar acessível a partir de qualquer linguagem de programação e não somente daquela em que foi implementado.
- Os objetos devem estar acessíveis através de uma rede, com a mesma sintaxe de acesso local.
- Um objeto deve estar acessível a partir de qualquer sistema operacional e não apenas daquele onde o objeto está executando.
- Os objetos devem estar acessíveis a partir de clientes baseados na Web, usando chamadas simples e eficientes sem indireção, sem auxílio de outros processos e sem interpretadores de *scripts*.

- O sistema deve ser capaz de trabalhar com sistemas não orientados a objetos como banco de dados relacional.
- O sistema deve interagir com outros sistemas orientados a objetos como SGBDOs, sistemas de análise e projetos orientados a objeto.
- As invocações de operações devem ser seguras.
- Deve ser fácil associar nomes simbólicos de alto nível aos objetos do sistema, obtendo uma referência de objeto para poder invocar operações sobre ele. Esses nomes devem ser hierárquicos para evitar colisões de nomes.
- Os detalhes de comunicação devem ser escondidos como se a conexão fosse direta entre os objetos.
- A verificação de tipo em tempo de execução deve ser oferecida.
- O sistema deve ter grande aceitação industrial em mercados como o financeiro, o petroquímico, por exemplo.
- O sistema deve ser baseado numa padronização não proprietária.
- Deve estar disponível a partir de múltiplos fabricantes, com mais de um fornecedor em cada plataforma.

As seções seguintes descrevem os conceitos de sua padronização, os componentes de sua arquitetura, a camada ORB em detalhe e os serviços de objetos presentes em sua especificação.

3.4.1 Padrão

Muitas empresas, hoje em dia, necessitam de sistemas integrados. Como CORBA é um padrão não proprietário pode solucionar as dificuldades encontradas nos domínios de redes de computadores, linguagens de programação e sistemas operacionais. Assim, CORBA pode ser visto como um ambiente de suporte ao desenvolvimento de novos sistemas ou como um ambiente para integração de aplicações no qual novos sistemas podem ser construídos combinando-se características de sistemas existentes [Bak97].

OMA (Object Management Architecture) provê algumas definições fundamentais de conceitos que ajudam a utilização de CORBA como ambiente padronizado de desenvolvimento de sistemas. São as seguintes [VD98]:

- **Implementações e Referências de Objeto.** As implementações de objeto são o código das operações definidas por uma interface IDL, enquanto as referências são a identificação do objeto usada pelos clientes para invocar suas operações. A implementação é parte do objeto CORBA e usualmente contém algum estado interno, interagindo, por exemplo, com banco de dados, monitores ou elementos de rede de telecomunicações. As referências são apontadores para os objetos e denotam um objeto em particular, podendo ser passadas para os clientes de objetos através de parâmetros ou de resultados de operações.

- **Tipos.** Os tipos são definidos usando a lógica de predicado na especificação CORBA. Os tipos de objeto estão relacionados por uma hierarquia de herança, onde o tipo Object está na raiz. Estão associados aos parâmetros e aos resultados de operações e podem ser usados como componentes em tipos de dados estruturados. Os tipos não objeto, como um inteiro, são usualmente chamados tipos de dados.

- **Interfaces.** Uma interface é uma descrição das operações que são oferecidas por um objeto podendo conter definições de tipos estruturados usados como parâmetros para aquelas operações. As interfaces são especificadas em IDL. Em CORBA, os tipos de interface e de objetos têm um mapeamento um para um.

- **Semântica de Operação.** Há dois tipos de semânticas de execução de operação definidas para as invocações estáticas: At-Most-Once e Best-Effort. O primeiro faz a operação ser executada exatamente uma vez se a execução foi bem sucedida, ou se uma exceção foi levantada será executada no máximo uma vez. Se uma operação é declarada usando a palavra-chave *oneway*, então o requisitante não espera pelo término da operação e a semântica é do tipo Best-Effort.

- **Assinaturas de Operação.** Cada operação tem uma assinatura, expressa em IDL, que contém um identificador de operação (nome de operação), o tipo do valor de retorno, a lista de parâmetros com nome, tipo e indicação de direção (IN, OUT ou INOUT), além de componentes opcionais: cláusula *raises* para levantar exceções, palavra-chave *oneway* para semântica Best-Effort e cláusula *context* para listar valores de ambiente.

- **Atributos.** As interfaces podem conter atributos, que são declarados com o possível modificador *readonly*. Os atributos são logicamente equivalentes a um par de operações, onde a primeira é uma operação de acesso que retorna um valor de um tipo especificado e a segunda é uma operação de modificação que toma um argumento do tipo especificado e ajusta seu valor.

- **Exceções.** Uma exceção é um tipo não objeto especializado. São declaradas com a palavra-chave *exception*, têm um nome e campos opcionais de valores de dados que provêem informação extra sobre a causa do término anormal de uma operação. Podem ser levantadas pelas operações, ou implicitamente pelo ORB ou pela implementação da operação.

3.4.2 Arquitetura

A arquitetura OMA é composta pelo núcleo ORB (Object Request Broker) e por quatro categoria de objetos: Serviços de Objetos (*CORBAservices*), Facilidades Comuns (*CORBAfacilities*), Interfaces de Domínio (*CORBAdomain*) e Objetos de Aplicação.

- **ORB.** Permite que objetos façam e recebam requisições e respostas de modo transparente em um ambiente distribuído. É fundamental para o desenvolvimento de aplicações com objetos distribuídos e para a interoperabilidade entre aplicações que executam em diferentes plataformas [COR02].

- **Serviços de Objetos.** Uma coleção de serviços (interfaces e objetos) que oferece funcionalidades básicas para uso e implementação de objetos. Serviços são necessários para construir qualquer aplicação distribuída e são sempre independentes de domínios de aplicação. Por exemplo, o Serviço de Ciclo de Vida, que define convenções para criar, apagar, copiar e mover objetos.

- **Facilidades Comuns.** Uma coleção de serviços que muitas aplicações podem compartilhar, mas não são fundamentais como o Serviço de Objetos. Por exemplo, gerenciamento de sistemas ou a facilidade de correio eletrônico.

- **Interfaces de Domínio.** Definem interfaces padronizadas em IDL para objetos padronizados usados por qualquer empresa na indústria.

- **Objetos de Aplicação.** Correspondem à noção tradicional de aplicação e não são padronizados pela OMG.

A arquitetura OMA é apresentada na figura 3.1 e o ORB é descrito na seção seguinte.

ORB

Cada implementação do núcleo de CORBA inclui a linguagem IDL e o mapeamento de IDL para algumas linguagens de programação (por exemplo, C++ e Java), o envio de uma requisição de objeto entre um cliente e um objeto alvo, as interface de programação de aplicação CORBA e o Repositório de Interface.

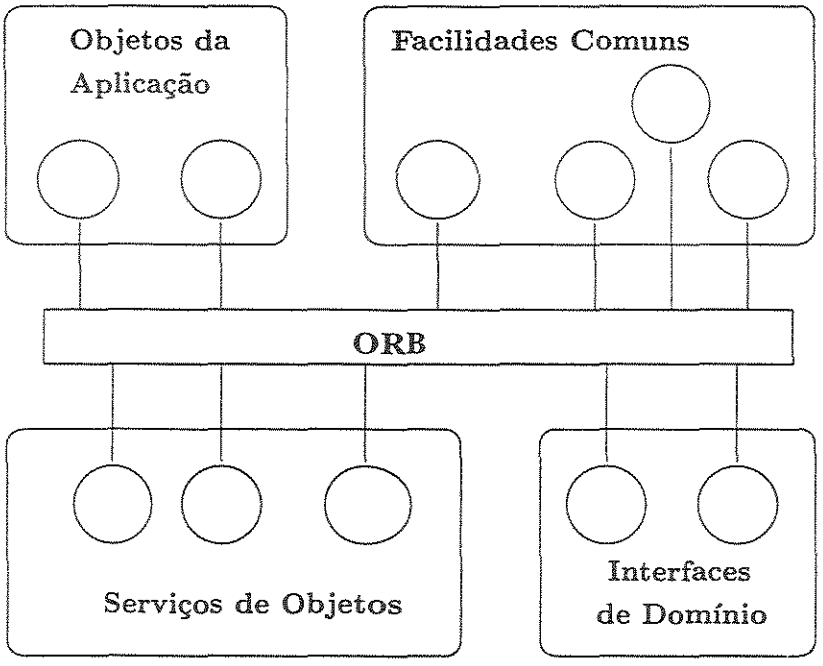


Figura 3.1: Estrutura de camadas de OMA.

O ORB é responsável pelos detalhes que envolvem um pedido de requisição de objeto a partir de um cliente e a resposta da requisição para seu destino [COR02].

ORBs provêem os seguintes benefícios [COR02]:

- **Invocações Estáticas e Dinâmicas de Métodos.** O ORB permite a definição de invocações de métodos em tempo de compilação ou dinamicamente em tempo de execução.
- **Ligações entre Linguagens de Alto Nível.** O ORB permite a invocação de métodos em objetos de servidores usando uma linguagem de alto nível a sua escolha, que pode ser diferente da linguagem em que os servidores foram escritos. CORBA separa interface da implementação provendo tipos de dados neutros independentes de linguagem.
- **Sistema Auto-Descritivo.** CORBA provê metadados para descrever interfaces de servidores conhecidos no próprio sistema, permitindo que o Repositório de Interface seja capaz de informar em tempo real as funções e os parâmetros que os servidores provêem. Tais metadados são gerados automaticamente pelos compiladores de IDL.
- **Transparência de Acesso.** O ORB pode enviar as invocações para os objetos

usando o protocolo IIOP em um único processo, em múltiplos processos executando na mesma máquina ou em múltiplos processos executando entre redes e sistemas operacionais diferentes.

- **Suporte à Segurança e às Transações.** O ORB inclui informação de contexto em suas mensagens para tratar de segurança e de transações.
- **Mensagens Polimórficas.** Em contraste com muitos middlewares, um ORB não invoca simplesmente uma função remota através de RPC, ele invoca uma função num objeto alvo. Isso significa que a mesma chamada de função tem diferentes efeitos dependendo do objeto que a recebeu.
- **Coexistência com Sistemas Legados.** A separação da definição de objetos de sua implementação é perfeita para o encapsulamento de aplicações legadas, permitindo a criação de objetos a partir de códigos implementados em COBOL, por exemplo.

A estrutura do ORB é apresentada na figura 3.2.

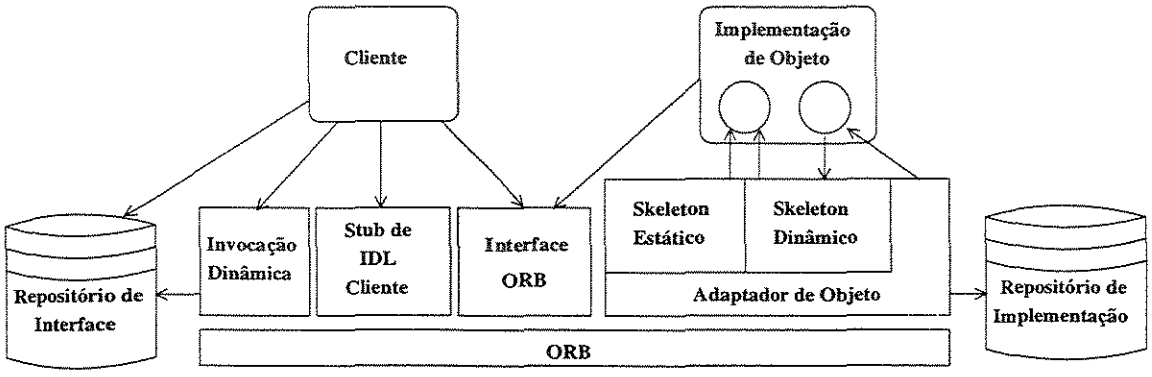


Figura 3.2: Estrutura do ORB.

Os objetos no ORB podem atuar como clientes ou como servidores. Os componentes do lado cliente do ORB são [COR02]:

- **Stubs IDL dos Clientes.** Provêem as interfaces estáticas para os objetos. São pré-compilados e definem como os clientes invocam os serviços nos servidores. Esses serviços são definidos em IDL e ambos *stubs* do cliente e do servidor são gerados pelo compilador de IDL. O stub inclui o código para codificar e decodificar a operação e seus parâmetros em formatos de mensagens que podem ser enviadas para o servidor.
- **Interface de Invocação Dinâmica.** Encontra a definição de métodos em tempo de execução através de metadados. CORBA define APIs que manipulam estes metadados.

- **APIs do Repositório de Interface.** Permitem a obtenção e a modificação das descrições de todas as interfaces de componentes registradas, seus métodos e seus parâmetros. O Repositório de Interface é um banco de dados distribuído que contém as informações sobre as interfaces definidas em IDL. As APIs permitem aos componentes dinamicamente acessar, armazenar e atualizar a informação dos metadados.

- **Interface do ORB.** Consiste de algumas APIs para serviços locais como, por exemplo, converter uma referência de objeto para uma cadeia de caracteres.

Do lado do servidor, o ORB localiza o adaptador de objetos do servidor, transmite os parâmetros e transfere o controle para a implementação de objetos através do stub IDL do servidor. Os elementos do lado servidor são os seguintes [COR02]:

- **Stubs IDL dos Servidores (Skeletons).** Provêem as interfaces estáticas para cada serviço exportado pelo servidor.

- **Interface Skeleton Dinâmica.** Provê um mecanismo de ligação em tempo de execução para os servidores que precisam manipular as chamadas de métodos para os objetos que não têm *skeletons*. O Skeleton Dinâmico identifica os métodos e o objeto alvo na mensagem que está chegando. Equivale à Interface Dinâmica do lado do cliente.

- **Adaptador de Objeto.** Aceita as requisições para os serviços de objetos do servidor. Provê o ambiente em tempo de execução para a instanciação de objetos do servidor, passando requisições para eles e atribuindo-lhes identificadores. Também registra as classes do servidor e suas instâncias no Repositório de Implementação.

- **Repositório de Implementação.** Provê um repositório de informação em tempo de execução sobre as classes de um servidor, os objetos que são instanciados e seus identificadores.

- **Interface ORB.** Consiste de um conjunto de APIs para serviços locais idênticos ao lado do cliente.

3.5 Java com CORBA

Os dois maiores modelos de objetos usados em computação distribuída, Java e CORBA, introduzem diferentes abordagens. Java oferece independência de plataforma e código de alto nível, enquanto CORBA provê uma infra-estrutura que possibilita invocações de operações em objetos localizados em qualquer lugar numa rede como se estivem locais

para a aplicação. As duas abordagens convergem quando um mapeamento é definido da IDL da OMG para Java. Quando combinado com um sistema em tempo de execução que provê esse mapeamento, o resultado é um ORB Java. Os objetos portáveis de Java podem ser executados independente de plataforma, enquanto a infra-estrutura de CORBA provê transparência de distribuição [OH98].

3.5.1 O que Java oferece para CORBA

A principal razão para usar o mapeamento de Java a partir de IDL é explorar a combinação de características únicas da própria linguagem como portabilidade entre plataformas, programação para Internet, linguagem orientada a objetos e modelo de componentes [VD98]:

- **Portabilidade de Aplicações entre Plataformas.** Programas em Java são altamente portáveis por causa da representação *byte-code* padronizada gerada por seus compiladores. Como compiladores e sistemas em tempo de execução estão disponíveis para virtualmente qualquer plataforma de hardware e sistema operacional, isto é uma vantagem relevante em relação a outras linguagens de programação. Conseqüentemente, custos de desenvolvimento e manutenção podem ser significantemente reduzidos.

- **Programação para Internet.** A linguagem Java permite a implementação de clientes CORBA como *applets*, habilitando acesso aos objetos CORBA e, potencialmente, às aplicações legadas usando *browsers* para Web. De fato, estes tipos de browsers estão se tornando uma universal interface gráfica de usuário. Além dos applets serem clientes CORBA, eles podem implementar objetos CORBA também. Apesar de o modelo applet desabilitar o acesso dos recursos na máquina onde o applet está executando, possibilita o uso de interfaces *callback*.

- **Orientação a Objetos.** Os principais mapeamentos oferecidos pelos atuais produtos ORB disponíveis são C++, C e Smalltalk. Java provê uma abordagem mais simples no processo de programação do que C++, com menos gerenciamento de memória, sem ponteiros, sintaxe menos confusa e regras de resolução de métodos mais simples. Além disso, Java possui características exclusivas em relação às linguagens citadas, como coleta de lixo automática, manipulação de exceção e suporte de threads integrado. Tais características são geralmente desejáveis e úteis para a programação em sistemas distribuídos.

- **JavaBeans.** Esse modelo de componentes Java foi adicionado ao núcleo da linguagem permitindo aos programadores combinar a funcionalidade provida por um número de classes Java num componente único.

Em relação a CORBA, [OH98] descreve que Java:

- Complementa os serviços de Ciclo de Vida e de Externalização de CORBA, na medida que possibilita para o ORB identificar o comportamento e o estado dos objetos.
- Integra CORBA com a tecnologia Web, na medida em que as empresas incluem ORBs em seus browsers e servidores, fazendo do protocolo IIOP de CORBA o modelo de objetos distribuídos utilizado tanto para Java quanto para Internet.
- Simplifica a distribuição de código em grandes sistemas CORBA, na medida que seu código pode ser disponibilizado e gerenciado de forma centralizada a partir do servidor. A atualização é feita uma única vez no servidor. Não é preciso que atualize-se cada *desktop* individualmente, pois os clientes receberão as atualizações quando necessitarem.
- Complementa a infra-estrutura de agentes em CORBA.
- Usa o conceito de interfaces, como em CORBA, para separar uma definição de objeto de sua implementação.

[OH98] afirma que sem CORBA, Java na Web é apenas um competidor do HTML dinâmico. Com CORBA, o cliente Java torna-se um objeto complexo numa arquitetura cliente/servidor de três níveis. HTML dinâmico não se interessa em invocar métodos em objetos de servidores.

Uma breve comparação entre Java e CORBA ORB e Java e HTTP-CGI é feita na tabela 3.1.

3.5.2 ORB Java

O ORB Java é um ORB CORBA que usa o protocolo IIOP escrito inteiramente em Java para permitir portabilidade. Além disso, qualquer código gerado pelo compilador IDL deve estar em Java puro e deve-se ser capaz de baixar o código da rede e executá-lo em qualquer máquina que possua um ambiente Java em tempo de execução.

Com um ORB Java, um applet Java pode diretamente invocar métodos em objetos CORBA usando o protocolo IIOP sobre a Internet. Assim, o cliente e o servidor estabelecem uma conexão direta usando o ORB. Deve-se ser capaz de carregar um applet em CORBA em qualquer browser comercial com Java nativo, podendo executá-lo.

As padronizações alcançadas por Java e CORBA cobrem as seguintes áreas [OH98]:

Característica	Java/CORBA ORB	Java/HTTP-CGI
Preservação de estado entre invocações	Sim	Não
IDL e Repositório de Interface	Sim	Não
Invocações Dinâmicas	Sim	Não
Suporte aos Metadados	Sim	Não
Transações	Sim	Não
Segurança	Sim	Sim
Serviços de Objetos	Sim	Não
Callbacks	Sim	Não
Infra-estrutura Servidor/Servidor	Sim	Não
Escalabilidade de Servidor	Sim	Não
Métodos definidos em IDL	Sim	Não

Tabela 3.1: Comparação entre ORBs Java e Java com HTTP/CGI.

- **IDL/Java.** Esse padrão mapeia IDL de CORBA para Java. Define uma interface para stubs e skeletons portáveis, permitindo que seus binários executem em ORBs de diferentes fornecedores.

- **Java/IDL.** Esse é o padrão que converge CORBA e RMI. Permite que especifiquem interfaces remotas usando semântica RMI de Java e não IDL de CORBA. O compilador usa essa semântica para gerar automaticamente IDLs CORBA e códigos stub e skeleton. O subconjunto RMI/IDL permite que programas RMI sejam invocados por clientes CORBA em diversas linguagens usando IIOP e permite que tais programas chamem objetos CORBA escritos em outra linguagem. Este padrão está relacionado com o padrão chamado Objetos-por-Valor, que permite passar o estado de um objeto como parâmetro de uma invocação de método remota.

- **Enterprise JavaBeans.** A especificação EJB inclui um mapeamento CORBA e é baseado no subconjunto RMI/IDL e em transações de CORBA. Mapeia, também, JNDI (Java Naming and Directory Interface) para o Serviço de Nomes de CORBA.

3.6 Serviços de Transação em CORBA

Existem dois serviços de objetos em CORBA que definem interfaces para a manipulação de transações: Transação e Mecanismos de Estruturação de Transação.

O Serviço de Transação controla a validação e o cancelamento de transações que executam em diversos bancos de dados heterogêneos. Usa o protocolo de validação de duas fases entre os servidores de objetos transacionais e os coordenadores de transação [OTS02b].

O Serviço de Mecanismos de Estruturação de Transação provê uma arquitetura de baixo nível para aplicações, por exemplo Workflows, que criam transações avançadas e caracterizam-se por terem estrutura complexa, levarem longo tempo para completar, podendo ter períodos de inatividade e terem relacionamentos com outras aplicações [OTS02a]. É uma extensão do Serviço de Transação.

3.6.1 Serviço de Transação

Conhecido por OTS, Object Transaction Service, este serviço integra o paradigma de transação com o desenvolvimento orientado a objetos. Suporta o conceito de transações com características ACID e define interfaces que permitem múltiplos objetos distribuídos cooperarem provendo atomicidade.

O serviço provê o suporte de dois modelos de transação: tradicionais (flat) e aninhadas. O primeiro define transações que executam no mesmo nível hierárquico e que não possuem transações filhas. O segundo especifica transações em vários níveis hierárquicos com transações filhas (subtransações) associadas. As transações aninhadas validam (executam *commit*) se suas subtransações validaram. As subtransações podem ser recuperadas em caso de falha (executam *rollback*) sem que a transação pai seja abortada também.

Uma transação pode envolver diversos objetos desempenhando múltiplas requisições. O escopo de uma transação é definido por um contexto de transação que é compartilhado pelos objetos envolvidos. Entretanto, o Serviço de Transação não exige que as requisições sejam executadas no escopo da transação. Além disso, não restringe o número de objetos participantes, nem a topologia da aplicação e nem como esta é distribuída pela rede.

Neste serviço são definidas cinco entidades: Cliente Transacional, Objeto Transacional, Objeto Recuperável, Servidor Transacional e Servidor Recuperável.

Cliente Transacional é o programa que invoca operações sobre os objetos transacionais em uma única transação. O programa que inicia uma transação é chamado de originador de transação.

Os Objetos Transacionais são aqueles que tem seus comportamentos afetados por serem requisitados no escopo de uma transação. O Serviço de Transação não exige que as requisições tenham comportamento transacional, mesmo se estiverem no escopo de uma transação. Um objeto pode escolher possuir comportamento transacional ou prover suporte transacional para algumas requisições. Se o objeto não possui comportamento transacional, as mudanças realizadas sobre ele podem não se manter persistentes e podem não ser recuperadas.

Objetos Recuperáveis são objetos cujos dados são afetados pelo processo de validação e recuperação de uma transação. Por definição, um objeto recuperável é um objeto transacional. Tipicamente, estes objetos são envolvidos numa transação pois é necessário

que certas informações sejam armazenadas persistentemente em momentos críticos de seu processamento. Quando um objeto recuperável recomeça depois de uma falha, ele participa da recuperação utilizando os dados armazenados de modo persistente.

O Servidor Transacional é uma coleção de um ou mais objetos cujos comportamentos são afetados pela transação e que não possuem estados recuperáveis. Mesmo assim, implementa mudanças transacionais a partir de objetos recuperáveis. Um servidor transacional não participa do término de uma transação, mas pode força-lá a executar rollback.

Servidores Recuperáveis são coleções de objetos, nas quais pelo menos um deles é recuperável.

3.6.2 Serviço de Mecanismos de Estruturação de Transação

Este serviço de objetos define o Modelo de Serviço de Atividade. Atende aos mais diversos modelos de transação, como as transações definidas no Serviço de Transação, transações aninhadas, Sagas com Compensação, Transações Flexíveis e Esquemas de Versionamento. Nele existem três camadas: a Infra-Estrutura de Aplicação, o próprio Serviço de Atividade e a Camada Básica de Suporte.

A camada de Infra-Estrutura de Aplicação é responsável pelo gerenciamento de workflow e de transações de longa duração e pelo suporte aos Componentes de Aplicação, que utilizam as Atividades especificadas pelo Serviço. Utiliza as outras duas camadas.

A camada do Serviço de Atividade especifica as entidades do modelo: Atividade, Ação e Sinal. E divide-se em duas partes: Interfaces e Implementação.

Uma Atividade é uma unidade de trabalho, eventualmente distribuída, que pode ter comportamento transacional. Toda entidade pode fazer parte de uma Atividade, embora uma Atividade não necessite ser composta por outras Atividades. As Atividades são criadas, executadas e terminadas. Podem ser suspensas e, então, reiniciadas. O resultado de uma Atividade pode ser usado para determinar o fluxo de controle de outras Atividades.

Uma Atividade interage com outra Atividade através de Sinais e Ações. Uma Atividade pode transmitir seus dados (Sinais) para outras Atividades em qualquer momento de seu ciclo de vida, por exemplo quando termina.

Para permitir que uma Atividade seja independente de outra Atividade e permitir coordenação e pontos de controle que estão fora do domínio da Atividade, os Sinais são enviados para as Ações ao invés das Atividades. A Ação utiliza as informações do Sinal de modo conveniente para a aplicação. Uma Ação pode ser considerada como um ponto de entrada/saída para/de uma Atividade.

As Interfaces do Serviço de Atividade são usadas pela camada de Infra-Estrutura de Aplicação e a Implementação utiliza a Camada Básica de Suporte.

A Camada Básica de Suporte provê pelo menos o Serviço de Transação, o Serviço

de Objeto Persistente, o Serviço de Logging, para substituir o Serviço de Objeto Persistente, e o ORB. Esta camada é responsável pelas funcionalidades essenciais do Serviço de Atividade.

Uma Atividade durante sua existência pode utilizar transações e subtransações. A figura 3.3 exemplifica o relacionamento entre as Atividades e as Transações em uma aplicação. Os círculos com linha cheia representam as Transações e os círculos com linha tracejada representam as Atividades. A1 é uma Atividade composta por duas Transações subseqüentes. A2 não é transacional. A3 possui uma Transação, que possui a Atividade A3', que por sua vez possui outra Transação. A4 e A5 são Atividades que não utilizam Transações.

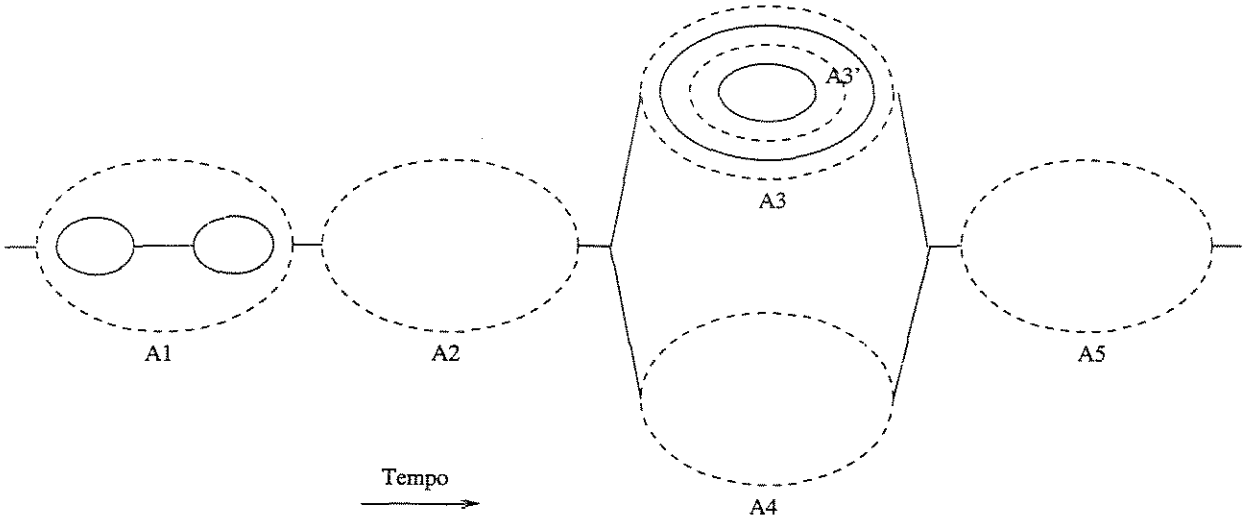


Figura 3.3: Relacionamento entre Atividade e Transação.

Capítulo 4

Serviço de Transações Cooperativas

4.1 Introdução

Nesse capítulo é apresentado o Serviço de Transações Cooperativas para atender os requisitos das aplicações baseadas em trabalho de grupo. São amplamente conhecidos os benefícios de transações voltadas para aquisição de consistência [BHG87]. Entretanto, no serviço proposto os mecanismos de consistência tradicionais foram substituídos com o intuito de oferecer maior visibilidade dos dados entre os usuários trabalhando em cooperação. Assim, esse serviço ao mesmo tempo que flexibiliza o controle de concorrência, provê consistência de dados através de transações.

Além disso, as aplicações podem ser estruturadas como transações hierárquicas modelando grupos de usuários que trabalham em cooperação. Cada nível da hierarquia tem sua área de trabalho onde os objetos podem ser copiados de áreas de níveis ascendentes. O trabalho em andamento dos membros do grupo pode ser copiado ou transferido entre suas áreas de trabalho, aumentando a visibilidade entre eles.

A arquitetura para o serviço de transações é apresentada na seção sequinte. O modo como é feito o gerenciamento das transações é mostrado na seção 4.3. Os objetos transacionais são descritos na seção 4.4. A persistência dos dados e a recuperação de falhas são analisados nas duas últimas seções do capítulo. A seção 4.7 exemplifica o modelo. 4.8 finaliza o capítulo com uma breve comparação com outros modelos de transação.

4.2 Arquitetura

O Serviço de Transações Cooperativas pode ser dividido em três níveis (figura 4.1). O inferior é responsável pelo gerenciamento dos objetos distribuídos e consiste de um ORB [COR02]. O nível intermediário dá suporte aos objetos transacionais e o nível superior ao

gerenciamento de transação e às facilidades de interação às aplicações cooperativas.

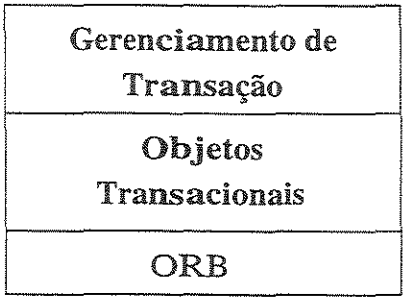


Figura 4.1: Arquitetura do Serviço de Transações Cooperativas.

O modelo de classes para o Serviço é apresentado na figura 4.2 de acordo com a notação UML [GRJ99]. As classes Cooperative Transaction, Group Transaction e User Transaction provêem o gerenciamento de transação. A classe Transactional Object constitui a camada intermediária da arquitetura.

O nível inferior tem como intuito gerenciar a comunicação distribuída entre os objetos do Serviço. Provê mecanismos transparentes de comunicação entre os objetos das camadas superiores. Os detalhes da camada ORB são mostrados no capítulo 3.

A camada de objetos transacionais consiste de objetos que incluem controle de concorrência e operações para implementação do protocolo de validação em duas fases[1] e checkpointing. Define os tipos tradicionais de tranca de leitura e escrita e novos tipos que permitem cooperação.

A camada de gerenciamento de transações preocupa-se com a criação e a manipulação da hierarquia de transações, utilizando os objetos transacionais e provendo mecanismos para trabalho em grupo.

4.3 O Gerenciamento de Transação

Existem dois tipos de transações cooperativas definidos no Serviço de Transações Cooperativas: grupo e usuário. As transações de grupo são aquelas que organizam as transações em grupos de trabalho. As transações de usuário são transações que representam os participantes do grupo. O usuário que cria uma transação de grupo torna-se o coordenador do grupo e responsável pelo gerenciamento dos participantes daquele grupo (inserção ou remoção de membros). Cada transação cooperativa tem uma lista de objetos alocados por ela. As transações de grupo, em particular, têm uma lista de transações que fazem parte do grupo e uma lista de usuários inscritos. Os usuários são inscritos para poderem participar do grupo de trabalho.

[1] Em inglês *two-phase commit*

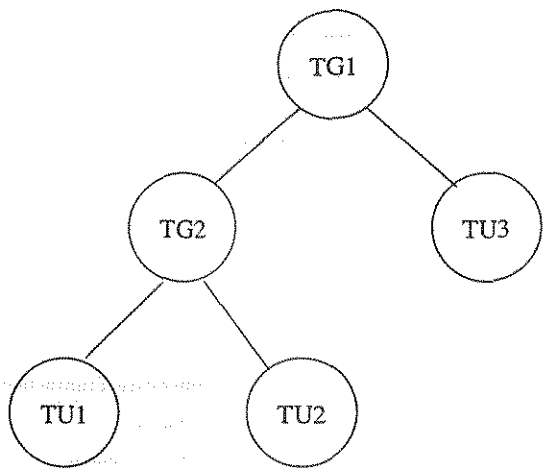


Figura 4.3: Estrutura de uma transação cooperativa.

raiz da hierarquia, requisita os objetos a partir da área pública, caso contrário requisita os objetos do grupo a que pertence.

A transferência dos objetos entre os níveis da hierarquia utiliza o mecanismo de *check-in/check-out*, que encarrega-se do controle de concorrência. O check-out é responsável pela cópia de objetos da área de trabalho de uma transação de grupo ou da área pública para uma área de trabalho descendente e pela obtenção de trancas sobre eles na área da transação de grupo ou na área pública (figura 4.4). O check-in (figura 4.5) preocupa-se com a atualização dos objetos na área da transação de grupo ou na área pública e a liberação de trancas obtidas. Desta forma, podem existir várias versões de um mesmo objeto, uma para cada nível na hierarquia.

Na figura 4.4, o objeto X com valor 1 presente na área pública sofre ckeck-out pela transação de grupo TG1, que copia o objeto para sua área com o nome de X1. A transação de usuário TU1 faz check-out do objeto X1 da área de TG1 e copia o objeto para sua área privada com o nome de X2. TU1 altera o valor de X2 para 2.

Na figura 4.5, TU1 executa check-in do objeto X2. O objeto X1 na TG1 é atualizado com o valor de X2, que é removido da área de TU1. De modo análogo, TG1 faz check-in do objeto X1 para a área pública de objetos.

4.3.1 Facilidades para Cooperação

A cooperação entre os usuários é alcançada através de operações especiais para transferência de objetos. Foram definidas três operações de cooperação: cópia, empréstimo e concessão. Para cada uma delas foi definida uma tranca cooperativa. Estas operações transferem os objetos entre as áreas de trabalho das transações de usuário de um mesmo grupo e relaxam a seriabilidade do controle de concorrência tradicional baseado em tran-

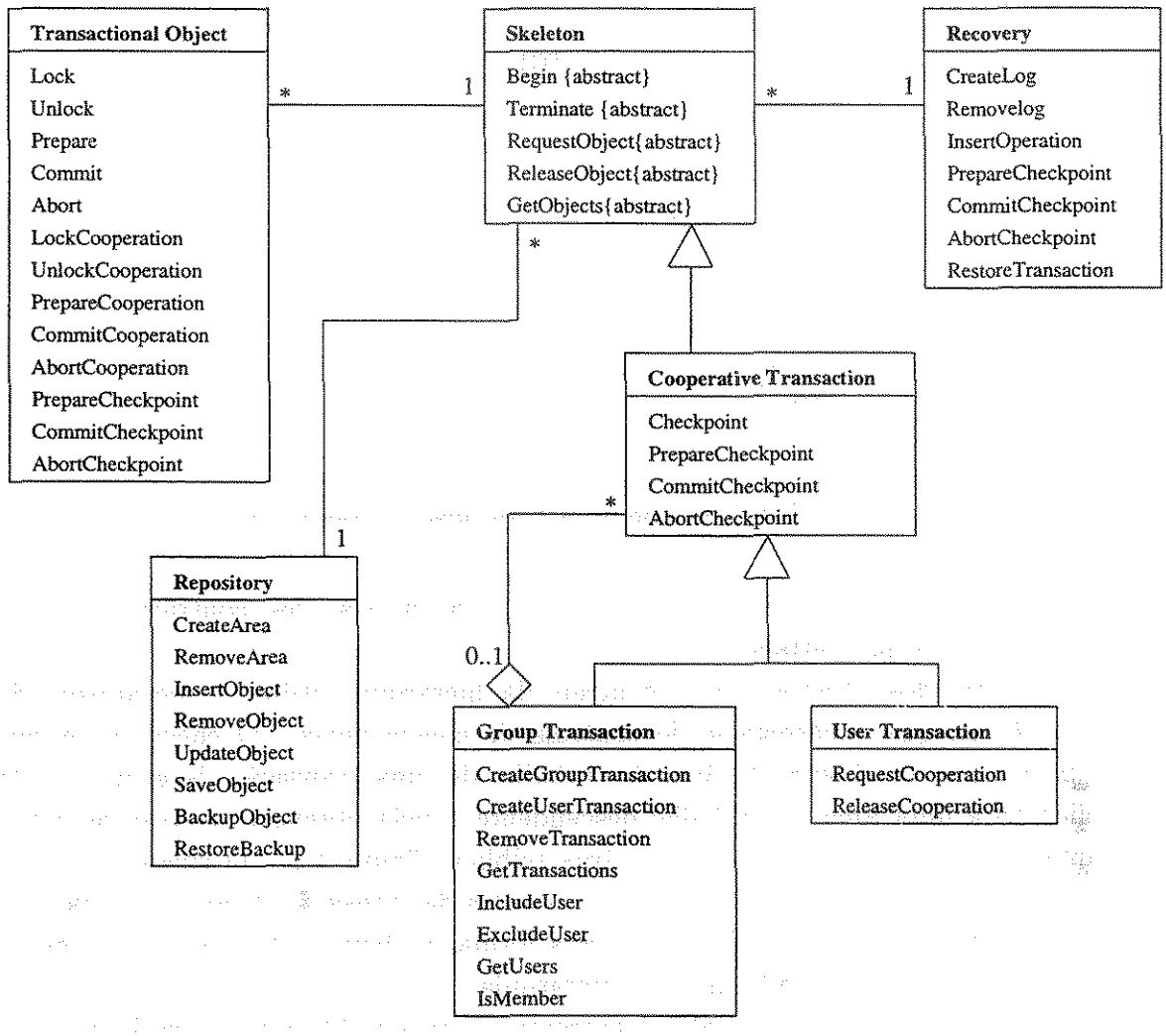


Figura 4.2: Estrutura de classes do Serviço de Transações Cooperativas.

O gerenciamento de transação estrutura as transações de grupo e de usuário como uma árvore. Os nós internos da hierarquia representam as transações de grupo e as folhas representam as transações de usuário. A figura 4.3 mostra uma transação de grupo TG1 que possui duas subtransações - uma transação de grupo TG2 e uma transação de usuário TU3. A transação de grupo TG2 criou duas subtransações de usuário TU1 e TU2.

Quando as transações são criadas na hierarquia, o usuário ou o grupo deve dizer se ela é relevante ou pode ser descartada em caso de falha. Para tanto, deve definir se ela é do tipo vital. O cancelamento de uma transação vital provoca o cancelamento de sua transação de grupo.

Cada transação possui sua própria área de trabalho, ou seja, uma vez criada a transação uma área vazia específica de manipulação de objetos é criada para ela. Se ela estiver na

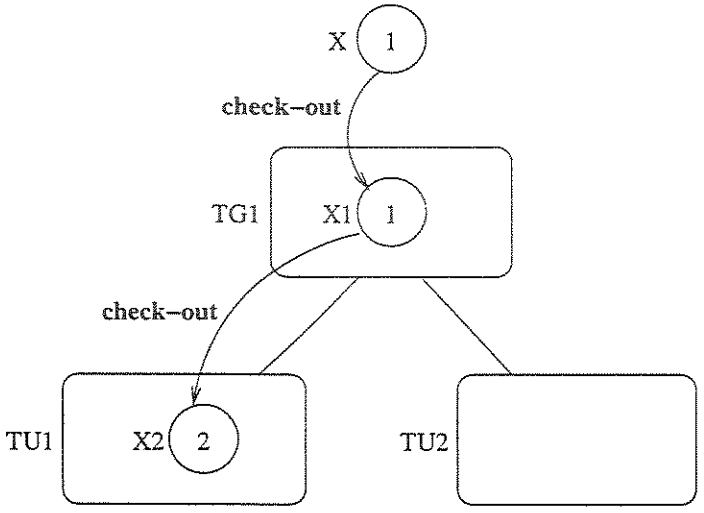


Figura 4.4: Check-out de objetos.

cas.

Assim, os objetos podem ser copiados de uma área de trabalho para outra, emprestados de uma transação e mais tarde devolvidos para a transação original ou concedidos para uma segunda transação que adquire os direitos da transação original.

Uma transação de usuário ao requisitar um objeto, deve especificar um dos tipos de tranca de cooperação disponíveis descritos na seção 4.4. Outra transação, desta maneira, pode executar uma das operações de cooperação sobre o objeto. São elas: cópia, empréstimo e concessão descritas em 4.3.5.

Uma transação de usuário requisita para o grupo um objeto de seu interesse através de uma operação de cooperação. Se a tranca sobre o objeto permitir a operação de cooperação, a transação de grupo encarrega-se de transferir o objeto da transação original para aquela que fez a requisição.

O mecanismo das operações é exemplicado do seguinte modo: quando um usuário precisa manipular determinado objeto, requisita uma operação de cooperação sobre ele. A transação do usuário delega a requisição para a transação de grupo e esta repassa a requisição para a transação de usuário que possui o objeto. O objeto é, então, alocado e disponibilizado para a transação requisitante. Uma vantagem deste mecanismo é que ele utiliza a estrutura de transações existente para coordenar o compartilhamento de objetos.

4.3.2 A Classe Skeleton

A classe Skeleton é a raiz da hierarquia de classes (figura 4.2) e define as operações básicas do Serviço. As operações definidas na classe são:

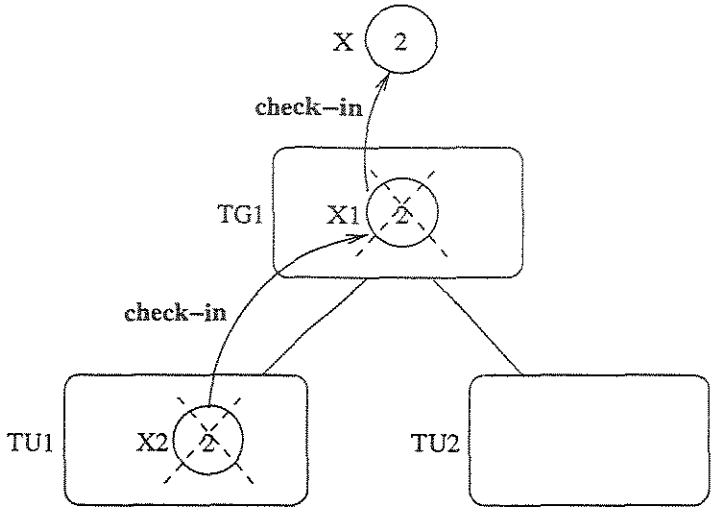


Figura 4.5: Check-in de objetos.

1. **Begin**
2. **Terminate**
3. **RequestObject**
4. **ReleaseObject**
5. **GetObjects**

Estas operações são abstratas e implementadas pela subclasse Cooperative Transaction.

4.3.3 A Classe Cooperative Transaction

Esta classe herda as operações da classe Skeleton e define as operações de uma transação cooperativa comuns às transações de grupo e de usuário. Introduz operações relacionadas com o mecanismo de checkpoint. Não é instanciada pelo Serviço. Esta classe têm duas especializações: Group Transaction e User Transaction.

As operações implementadas na classe são:

1. **Begin.** Inicia uma transação cooperativa criando uma nova área de trabalho para ela. A área de trabalho contém os objetos da transação e um arquivo de *log* para armazenar o estado da transação. A seção 4.5 explica como é feita a manipulação da área de trabalho.

2. **Terminate.** Finaliza a transação cooperativa de acordo com o tipo de término escolhido pelo usuário da transação e descarta sua área de trabalho. Se a transação for de usuário o término pode ser com falha, cancelando a transação e descartando seus objetos, ou pode ser com sucesso, atualizando os objetos da transação na área da transação de grupo ou na área pública, caso a transação esteja na raiz da hierarquia. Para a transação de grupo, o término está condicionado ao término de suas subtransações. A transação de grupo pode terminar com sucesso se todas as subtransações vitais terminaram com sucesso, caso contrário ela é cancelada. A transação pode terminar com falha mesmo se as subtransações vitais terminaram com sucesso. Na hora do término, os objetos da transação são transferidos para a área de trabalho do grupo ou para a área pública (se a transação for raiz) e liberados. Depois, remove a área de trabalho da transação. Os tipos de término especificados são os seguintes:

   - **COMMITTING.** Valida os resultados da transação e propaga as modificações sobre os objetos.
   - **ABORTING.** Cancela os resultados da transação e descarta as modificações sobre os objetos.

3. **RequestObject.** Aloca um objeto presente na área da transação de grupo ou na área pública, caso a transação esteja na raiz da hierarquia, de acordo com o tipo de tranca requisitado e o copia para a área de trabalho da transação requisitante. Verifica se a tranca requisitada é compatível com as trancas existentes sobre o objeto. É responsável pelo mecanismo de check-out. Esta operação pode ser bloqueante, através da qual a transação cooperativa espera pela liberação do objeto se ele foi requisitado por outra transação cooperativa anteriormente em modo incompatível.

   Um conjunto de trancas foi especificado e dividido de acordo com a natureza das trancas:

   - Tradicionais
     - **READ.** Leitura compartilhada.
     - **WRITE.** Escrita exclusiva.
   - Cooperativas
     - **W-COPY.** Escrita e cooperação somente para cópia.
     - **W-LOAN.** Escrita e cooperação para cópia ou empréstimo.
     - **W-CONCESSION.** Escrita e cooperação para cópia, empréstimo ou concessão.

As trancas que permitem escrita são incompatíveis entre si. Se um objeto estiver alocado com tranca de READ para uma transação, outra transação pode alocá-lo também com tranca de READ.

4. **ReleaseObject.** Libera um objeto. Propaga ou descarta as modificações realizadas sobre o objeto de acordo com o tipo de liberação. Se o objeto estiver alocado com tranca que permite escrita e se o tipo de liberação for com sucesso, o objeto é atualizado na área de trabalho da transação de grupo ou na área pública, caso a transação esteja na raiz da hierarquia. Caso o tipo de liberação seja com falha ou se o objeto estiver alocado com tranca de READ, a cópia local do objeto é descartada. Implementa o mecanismo de check-in. Os tipos de liberação de objetos estão a seguir:

   - **COMMITTING.** Propaga as modificações do objeto.
   - **ABORTING.** Descarta as modificações do objeto.

5. **GetObjects.** Retorna a lista de objetos alocados pela transação cooperativa.

6. **Checkpoint.** Armazena o estado da transação cooperativa e o estado de seus objetos na área persistente da transação (seção 4.6). Para o checkpoint ser realizado com sucesso, é preciso que o checkpoint tenha tido sucesso em cada uma das subtransações e dos objetos da transação.

7. **PrepareCheckpoint.** Salva o estado da transação de forma persistente.

8. **CommitCheckpoint.** Efetiva o checkpoint.

9. **AbortCheckpoint.** Cancela o checkpoint.

4.3.4 A Classe Group Transaction

A classe Group Transaction define o comportamento das transações de grupo. Ela herda as operações da classe Cooperative Transaction e adiciona operações para construção da hierarquia de transações.

As operações definidas na classe são:

1. **CreateGroupTransaction.** Cria uma subtransação de grupo e a inclui em sua lista de transações.

2. **CreateUserTransaction.** Cria uma subtransação de usuário e a inclui em sua lista de transações.

3. **RemoveTransaction.** Remove uma subtransação da transação de grupo, excluindo-a de sua lista de transações.

4. **GetTransactions.** Retorna as transações que fazem parte do grupo.

5. **IncludeUser.** Inscreve um usuário no grupo. O nome do usuário é inserido na lista de usuários do grupo.

6. **ExcludeUser.** Remove um usuário do grupo. O nome do usuário é removido da lista de usuários do grupo.

7. **IsMember.** Verifica se um determinado usuário pertence ao grupo.

8. **GetUsers.** Retorna os usuários que participam do grupo.

4.3.5 A Classe User Transaction

A classe User Transaction define o comportamento das transações de usuário. Herda as operações da classe Cooperative Transaction adicionando operações de cooperação.

As operações definidas nesta classe provêem colaboração entre os usuários, relaxando a seriabilidade e melhorando a visibilidade entre os mesmos.

As operações definidas na classe são as seguintes:

1. **RequestCooperation.** Requisita um objeto do grupo de acordo com o tipo de tranca de cooperação. O usuário pode requisitar o objeto para simples cópia, para empréstimo ou para concessão. Esta operação é válida somente entre transações de usuário associadas à mesma transação de grupo. Esta operação tem sucesso se a tranca sobre o objeto for compatível com o tipo da tranca de cooperação requisitada. A transação de grupo coordena a transferência do objeto entre as áreas das transações envolvidas. Os tipos de tranca de cooperação do mais restritivo para o mais flexível são:

   - **COPY.** Se a requisição for para cópia, transfere uma cópia do objeto para a área da transação requisitante, que pode invocar qualquer operação sobre ela, mas não pode fazer sua devolução para a transação original e não tem o direito de executar o check-in. No final da transação, as cópias são descartadas.
   - **LOAN.** Se o objeto for requisitado para empréstimo, transfere uma cópia do objeto para a área da transação requisitante, que pode executar qualquer operação sobre ele e deve devolvê-lo mais atualizado para a transação original. Porém, não tem o direito de executar o check-in. No final da transação, os objetos que foram emprestados são devolvidos para a transação original.

- **CONCESSION.** Se a requisição do objeto for para concessão, a transação original fornece uma cópia do objeto e concede todos os privilégios sobre o objeto para a transação requisitante. Esta, inclusive, executa o mecanismo de check-in no lugar da transação original.

2. **ReleaseCooperation.** Libera o objeto requisitado para cooperação. O objeto se copiado é descartado da área da transação. O objeto se emprestado é devolvido para a transação original com as possíveis modificações. Se o objeto foi concedido, ele sofre check-in. Os tipos de liberação de objetos estão a seguir:

   - **COMMITTING.** Propaga as modificações do objeto.
   - **ABORTING.** Descarta as modificações do objeto.

4.3.6 Os Estados de Execução da Transação Cooperativa

Os estados de execução da transação cooperativa encontram-se no diagrama 4.6.

Figura 4.6: Diagrama de estados de execução da transação cooperativa.

4.4 Objetos Transacionais

Os objetos transacionais, chamados apenas de objetos, são as entidades definidas no Serviço que sofrem manipulação das transações cooperativas. Garantem consistência de dados ao proverem operações relacionadas ao controle de concorrência. Além disso, utiliza o protocolo de validação em duas fases e especifica um mecanismo de *checkpointing*.

Quando um objeto é requisitado por uma transação o tipo de tranca especificado é comparado com as trancas existentes sobre o objeto. Se as trancas forem compatíveis, o objeto é alocado para a transação com a tranca requisitada. Quando o objeto é liberado, a tranca é removida do objeto.

Os objetos manipulados por uma transação podem propagar suas alterações, através do mecanismo de check-in. Este mecanismo utiliza o protocolo de validação em duas fases. A primeira fase armazena o estado do objeto no diretório da transação. Se esta fase executou com sucesso, a segunda fase transfere os resultados do objeto da área local de trabalho para a área de trabalho do grupo ou área pública. Caso contrário, a segunda fase cancela o processo de check-in e o objeto permanece inalterado na área superior da hierarquia.

O mecanismo de recuperação de falhas adotado para os objetos utiliza checkpoints. No caso do Serviço de Transações Cooperativas, os checkpoints são pontos de recuperação que armazenam o estado de uma transação e os estados dos objetos associados a ela. No armazenamento dos objetos, a primeira fase faz uma cópia do objeto na área persistente da transação. A segunda fase pode ter dois comportamentos: atualizar o estado do objeto na área persistente ou restaurá-lo a partir da cópia criada.

A seguir é analisado o mecanismo de controle de concorrência. Na seção posterior, as operações da classe Transactional Object são detalhadas.

4.4.1 O Controle de Concorrência

A tabela 4.1, mostra os tipos de tranca definidos nesse trabalho. A primeira coluna define os nomes das trancas. A segunda descreve o comportamento da tranca para o usuário que a requisitou, enquanto a terceira define qual o comportamento da tranca se outros usuários requisitassem o mesmo objeto.

Tranca	O que permite a quem requisita	O que permite a outros
READ	leitura	leitura
WRITE	escrita	nada
W-COPY	escrita	cópia
W-LOAN	escrita	cópia ou empréstimo
W-CONCESSION	escrita	cópia, empréstimo ou concessão
COPY	leitura	cópia
LOAN	escrita	nada
CONCESSION	escrita	nada

Tabela 4.1: Tipos de tranca.

A tabela 4.2 define a compatibilidade entre as trancas. As colunas da tabela representam as trancas existentes sobre um objeto e as linhas da tabela representam novas trancas requisitadas.

As trancas READ e WRITE são tradicionais, enquanto as outras foram definidas para permitir cooperação. A tranca READ permite acesso compartilhado para leitura e WRITE acesso exclusivo para atualização.

Se o objeto estiver alocado com a tranca W-COPY, a transação tem o direito de escrita sobre ele. Outras transações podem requisitá-lo com a tranca COPY. Elas copiam o objeto para sua área e passam a ter o direito de leitura sobre ele.

	READ	WRITE	W-COPY	W-LOAN	W-CONCESSION	COPY	LOAN/ CONCESSION
READ	X	-	-	-	-	-	-
WRITE	-	-	-	-	-	-	-
W-COPY/ W-LOAN/ W-CONCESSION	-	-	-	-	-	-	-
COPY	-	-	X	X	X	X	-
LOAN	-	-	-	X	X	-	-
CONCESSION	-	-	-	-	X	-	-

Tabela 4.2: Compatibilidade de trancas.

Se o objeto estiver alocado com a tranca W-LOAN, a transação tem o direito de escrita sobre ele. Outras transações podem requisitá-lo com a tranca COPY, porém apenas uma delas com a tranca LOAN. Neste último caso, a transação tem o direito de escrita. A transação original perde sua cópia temporariamente até que a outra transação o devolva.

Se o objeto estiver alocado com a tranca W-CONCESSION, a transação tem o direito de escrita sobre ele. Outras transações podem requisitá-lo com a tranca COPY e apenas uma delas com a tranca LOAN ou CONCESSION. Se a tranca requisitada for CONCESSION, a transação original cede os direitos de escrita sobre o objeto definitivamente para a transação que o requisitou. Esta transação deve, inclusive, executar o check-in para o objeto.

4.4.2 A Classe Transactional Object

A classe Transactional Object define o comportamento dos objetos transacionais. As operações de Transactional Object são descritas a seguir:

1. **Lock.** É responsável pela obtenção de trancas para os objetos. Verifica a compatibilidade da tranca requisitada com as trancas já existentes. As trancas disponíveis são READ, WRITE, W-COPY, W-LOAN e W-CONCESSION. Esta operação pode ser bloqueante, através da qual a transação requisitante fica bloqueada até que a tranca atual, em modo incompatível com a tranca requisitada, seja liberada. Esta operação é utilizada no mecanismo de check-out.

2. **Unlock.** Libera a tranca do objeto para uma transação. Esta operação é utilizada no mecanismo de check-in.

3. **LockCooperation.** Requisita trancas de cooperação para os objetos. Verifica a compatibilidade da tranca cooperativa requisitada com as trancas já existentes. As trancas de cooperação disponíveis são COPY, LOAN e CONCESSION.

4. **UnlockCooperation.** Libera a tranca cooperativa para o objeto.

5. **Prepare.** Armazena o estado do objeto na área persistente da transação.

6. **Commit.** Atualiza o objeto na área de trabalho do grupo ou na área pública, se for o caso.

7. **Abort.** Rejeita as modificações realizadas sobre o objeto. O objeto não é atualizado na área de trabalho do grupo ou na área pública.

8. **PrepareCooperation.** Armazena o estado do objeto requisitado para cooperação, na área persistente da transação de usuário.

9. **CommitCooperation.** Atualiza o objeto na área de trabalho da transação de usuário original.

10. **AbortCooperation.** Rejeita as atualizações sobre o objeto. O objeto não é atualizado na área de trabalho da transação de usuário original.

11. **PrepareCheckpoint.** Salva o novo estado do objeto na área persistente da transação.

12. **CommitCheckpoint.** Efetiva o checkpoint.

13. **AbortCheckpoint.** Restaura o estado antigo do objeto.

4.4.3 Os Estados do Objeto Transacional

Os estados do objeto transacional estão representados nos diagramas 4.7, 4.8, 4.9 e 4.10.

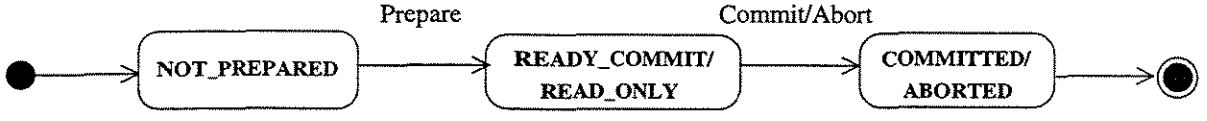


Figura 4.7: Diagrama de estados do objeto transacional em relação ao protocolo de validação.

4.5 O Repositório de Objetos

A persistência dos dados no Serviço de Transações proposto é obtida através de um repositório de objetos.

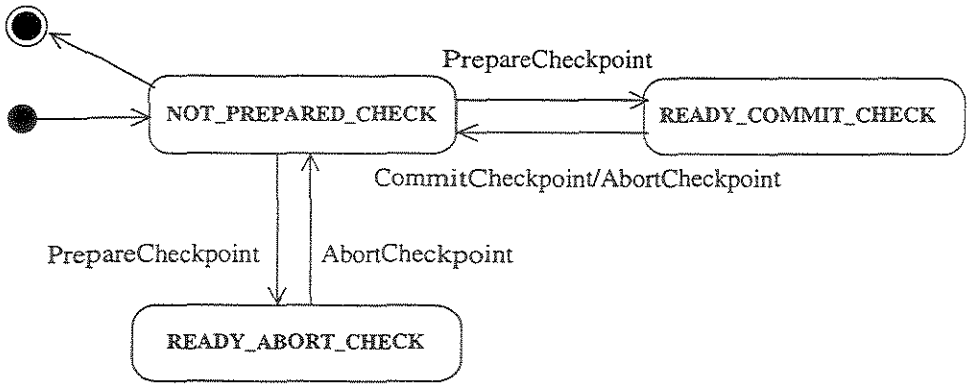


Figura 4.8: Diagrama de estados do objeto transacional em relação ao mecanismo de checkpointing.

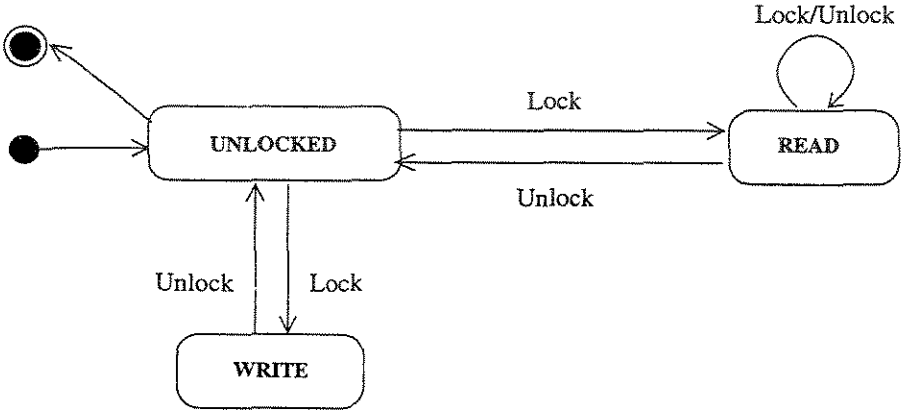


Figura 4.9: Diagrama de estados do objeto transacional em relação ao controle de concorrência tradicional.

À medida que transações são criadas, são criadas áreas de transações no repositório e à medida que objetos são requisitados são criados novos objetos na área das transações requisitantes. As áreas de transação são representadas por diretórios no sistema de arquivos do UNIX e os objetos por arquivos.

Inicialmente, um diretório raiz é criado como referência para qualquer operação no repositório. Este diretório representa a área pública de objetos. A cada nova transação é criado um subdiretório do diretório raiz. O repositório mapeia a árvore de transações para a estrutura de diretórios.

O mecanismo de controle de concorrência utiliza o repositório quando executa o check-in e o check-out. No processo de check-out, o repositório cria um arquivo contendo a cópia do objeto no diretório da transação que o requisitou. Quando ocorre o check-in, o repositório atualiza o arquivo associado ao objeto no diretório da transação

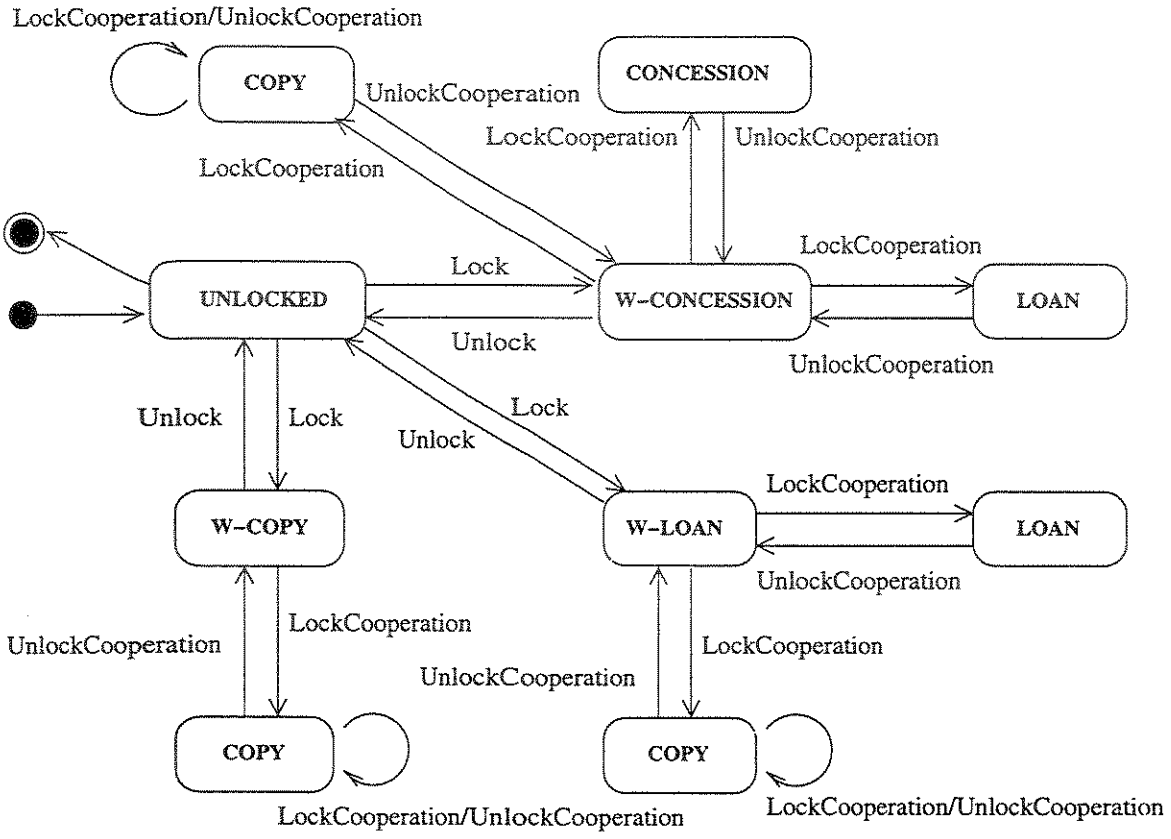


Figura 4.10: Diagrama de estados do objeto transacional em relação à cooperação.

de grupo ou na área pública com o novo estado do objeto.

O repositório abstrai a manipulação da estrutura de diretórios criando operações de alto nível como, por exemplo, **CreateArea**, que cria uma área de transação. Por isto, o impacto de se trocar o sistema de arquivos do repositório por um banco de dados com consultas SQL é reduzido.

A próxima seção descreve as operações da classe Repository.

4.5.1 A Classe Repository

A classe Repository do modelo é responsável pelo mecanismo de persistência dos objetos. Suas operações estão listadas a seguir:

1. **CreateArea.** Cria uma nova área de transação, inserindo um diretório no repositório com o nome da transação. Este diretório encontra-se dentro do diretório da transação de grupo ou do diretório raiz.

2. **RemoveArea.** Remove a área de uma transação, excluindo seu diretório do repositório.

3. **InsertObject.** Insere um objeto na área de uma transação, inserindo um arquivo contendo o estado do objeto no diretório da transação.

4. **RemoveObject.** Remove um objeto da área de uma transação, excluindo o arquivo correspondente do diretório da transação.

5. **UpdateObject.** Atualiza um objeto pertencente a área de uma transação, substituindo o conteúdo de seu arquivo pelo conteúdo do arquivo de um outro objeto.

6. **SaveObject.** Armazena o estado de um objeto em arquivo. Usa o mecanismo de loader do OrbixWeb.

7. **BackupObject.** Cria uma cópia do objeto no diretório da transação.

8. **RestoreBackup.** Restaura o objeto no diretório da transação a partir da cópia do objeto disponível.

4.6 O Mecanismo de Recuperação

A recuperação de falhas adotada no modelo de transações consiste no uso de logs e no mecanismo de *checkpointing*. De modo geral, os logs são arquivos que guardam informação sobre o estado de uma aplicação; no caso do Serviço de Transações proposto, guarda informações sobre o estado das transações. O estado dos objetos é gerenciado pelo repositório (seção 4.5). Os checkpoints são mecanismos utilizados normalmente para sincronizar estados de uma aplicação e, especificamente nesse contexto, servem para armazenar o estado das transações e seus objetos em memória persistente.

O checkpoint pode atuar de forma síncrona ou assíncrona. A forma síncrona atualiza o estado de toda a aplicação. A forma assíncrona, por sua vez, pode atualizar parcialmente o estado da aplicação, delimitando os pontos de atuação do checkpointing. A abordagem síncrona é adotada no Serviço de Transações, onde o checkpoint na transação raiz dispara o checkpoint nas subtransações. Essa abordagem foi escolhida pela sua simplicidade se comparada com outras abordagens.

O log consiste de uma área de estado da transação e uma área de operações. Estas áreas são atualizadas quando o checkpointing é executado. A área de operações representa a seqüência de operações realizadas pela transação depois do último checkpoint. Para cada operação são armazenados o código da operação e quais foram os parâmetros

utilizados. Operações de requisição e liberação de objetos e de inicialização e término de subtransações são as operações armazenadas no log.

A operação definida de checkpoint salva o estado da transação na memória persistente. Quando ocorre uma falha no sistema, o estado da transação gravado pelo último checkpoint é recuperado, as operações no log (executadas após o checkpoint) são analisadas para serem desfeitas e o estado dos objetos é restaurado.

As etapas do mecanismo de checkpointing são as seguintes:

1. Armazena o estado da transação no log.
2. Armazena o estado de seus objetos no repositório (**PrepareCheckpoint**).
3. Armazena o estado das subtransações (**PrepareCheckpoint**).
4. Se houver sucesso em 2 e 3, efetiva checkpoint nos objetos (**CommitCheckpoint**) e nas subtransações (**CommitCheckpoint**). Caso contrário, cancela o checkpoint (**AbortCheckpoint** para os objetos e **AbortCheckpoint** para as subtransações).
5. Marca o checkpoint como terminado.
6. Limpa a área de operações no log.

Os passos do algoritmo de recuperação seguem:

1. Restaura o estado da transação contido do log (estado salvo no último checkpoint).
2. Recupera o estado dos objetos da transação.
3. Percorre a parte de operações no log para desfazer as operações após o checkpoint.

4.6.1 A Classe Recovery

A classe Recovery é responsável pela recuperação de falhas. Esta classe contém as operações necessárias para a manipulação do log e para a execução do mecanismo de checkpointing.

A operação **Checkpoint** da classe Cooperative Transaction utiliza-se das três operações do checkpointing definidas nesta classe. A saber: **PrepareCheckpoint**, responsável pela primeira fase do mecanismo; **CommitCheckpoint**, responsável pela segunda fase do mecanismo e **AbortCheckpoint**, responsável pelo cancelamento da segunda fase. Se **PrepareCheckpoint** executa com sucesso, **CommitCheckpoint** é executado, caso contrário a operação **AbortCheckpoint** é executada.

Após a identificação de uma falha, a operação de recuperação **RestoreTransaction** restaura o último estado disponível da transação no log, restaura os objetos da transação e reexecuta as operações no log da transação.

As operações da classe estão a seguir:

1. **PrepareCheckpoint.** Prepara os objetos da transação para o checkpointing.
2. **CommitCheckpoint.** Valida o checkpoint para cada um dos objetos da transação. Limpa a área de operações do log.
3. **AbortCheckpoint.** Cancela o checkpoint para cada um dos objetos da transação.

A operação de checkpointing deve ser explicitamente invocada para uma transação raiz que inicia checkpoint nas suas subtransações.

Para garantir a recuperação de uma transação é preciso manter um log de operações da transação. Desta maneira, a cada chamada de uma operação da transação seus detalhes são registrados automaticamente no log. Quando uma das operações: requisição e liberação de objetos e criação e remoção de transações é executada o mecanismo de recuperação armazena os dados referentes à operação, utilizando a operação **InsertOperation**. Esta operação insere o nome, os tipos e os valores de entrada da operação na área de operações do log.

A operação de recuperação que preocupa-se com a restauração da transação é chamada de **RestoreTransaction**. Assim como **Checkpoint**, ela não é executada de forma automática. Quando há uma falha de comunicação na rede ou quando o servidor de transações não está acessível, a aplicação pode decidir restaurar uma ou mais transações invocando o método **RestoreTransaction**. Seu algoritmo é definido a seguir:

1. Restaura o estado mais recente da transação. Para tanto, cria uma nova transação com os dados do log.
2. Recupera o estado dos objetos utilizando-se do repositório e os aloca para a transação restaurada.
3. Para cada operação gravada no log, o mecanismo de recuperação verifica qual é seu tipo. Se a operação for do tipo de criação de uma subtransação, o mecanismo executa a operação inversa de remoção de transação. Se a operação for do tipo de requisição de objetos, executa a operação de liberação de objetos. Antes de processar a operação é verificado se existe no log alguma operação que desfaz a operação que está sendo tratada. Por exemplo, se a operação **CreateGroupTransaction** estiver sendo tratada e existir a operação **RemoveTransaction** para a mesma subtransação, a operação é desconsiderada.

4. Remove a área de operações do log.

5. Retorna para a aplicação a transação recuperada.

Existem outras operações que complementam o mecanismo de recuperação: **CreateLog** e **RemoveLog**. A primeira operação cria um arquivo de log para a transação e é executada automaticamente quando se cria a área da transação usando o repositório. A segunda remove o log da transação quando a transação é finalizada.

4.7 Exemplo de Aplicação

Uma aplicação cooperativa típica é o desenvolvimento de um sistema de software por um time de programadores (figura 4.11). Por exemplo, o programador João é responsável pelo desenvolvimento do módulo A, mas precisa ver as mudanças no módulo B desenvolvido por Maria. Enquanto desenvolve B, Maria também necessita ver o módulo A. Isto poderia ser feito criando-se uma instância da classe Group Transaction (transação de grupo Tg na figura 4.11) e então cada programador cria uma instância da classe User Transaction a ser incluída nesta transação de grupo (transações de usuários Tj e Tm para João e Maria, respectivamente). João copia o módulo A para sua área privada enquanto Maria copiará o módulo B.

A colaboração entre João e Maria é alcançada através de requisições de trabalho não finalizado de cada um. Os objetos podem ser copiados, transferidos temporariamente entre áreas de trabalho ou concedidos de uma transação de usuário para outra.

Os objetos do exemplo acima são apresentados na figura 4.12.

Na seção seguinte, alguns cenários importantes são abordados utilizando-se diagramas de seqüência em UML.

4.7.1 Diagramas de Seqüência

Os diagramas de seqüência em UML mostram como os objetos do Serviço de Transações Cooperativas comunicam-se entre si em determinados cenários. UML é uma metodologia de desenvolvimento de sistemas orientada a objetos unificada e padronizada [GRJ99], e tem como uma de suas etapas o desenvolvimento de diagramas de seqüência que esclarecem determinadas situações do sistema.

Estes diagramas retratam a interação entre os objetos em função do tempo. O tempo aumenta conforme as interações ocorrem de cima para baixo. Métodos com as iniciais em letra minúscula representam operações auxiliares do Serviço e são listadas no apêndice A.

João cria uma transação de grupo (GT) usando a *factory* de transações do Serviço. Depois, insere os usuários Maria e Pedro como participantes do grupo. Maria e Pedro

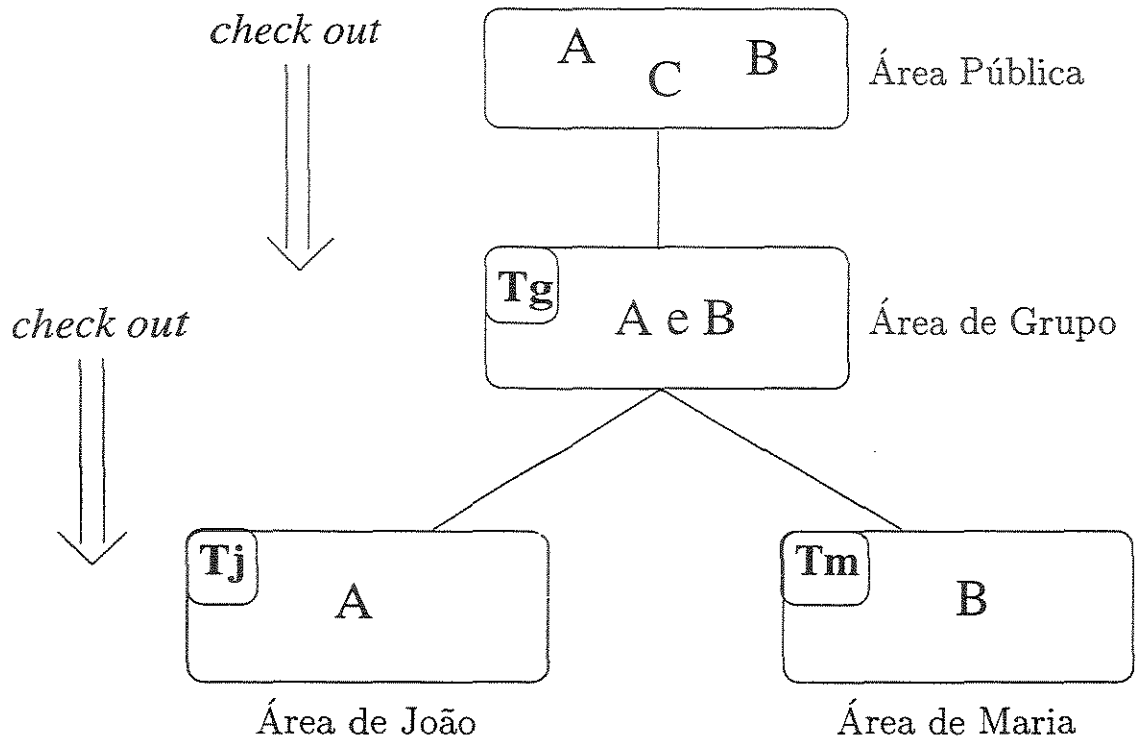


Figura 4.11: Exemplo de Colaboração.

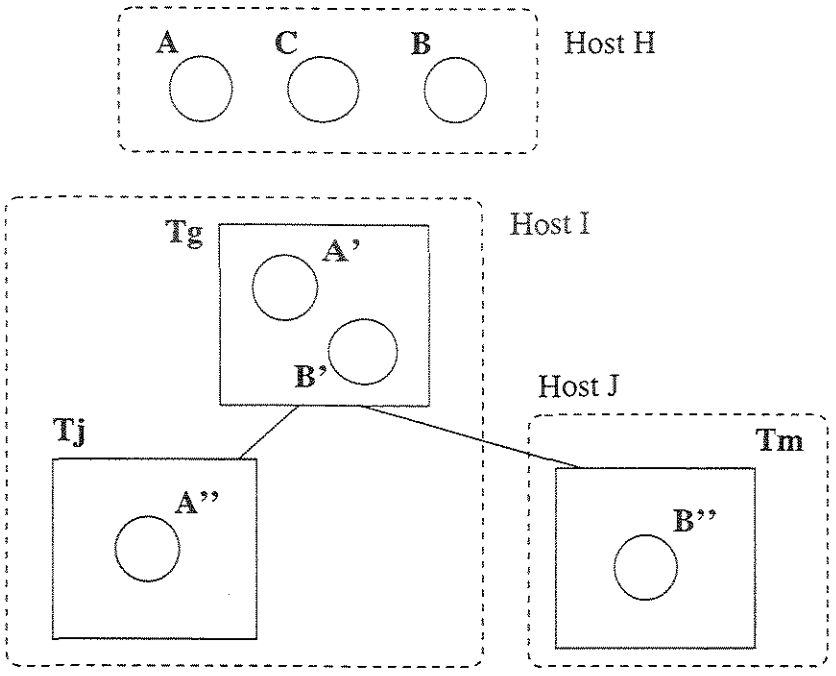
criam transações de usuário (UT1 e UT2, respectivamente). Este cenário é ilustrado pelo diagrama 4.13.

A usuária Maria requisita o objeto X do grupo com tranca W-CONCESSION. UT1 aloca X. A factory de objetos cria e devolve uma cópia do objeto (X1) para UT1. A transação de Maria salva o estado de X1 no repositório. UT1, além disso, armazena os dados da operação de requisição de X em seu log através do mecanismo de recuperação. Na figura 4.14 o diagrama de seqüência mostra este cenário.

O usuário Pedro requisita o objeto X do grupo com tranca de empréstimo. UT2 verifica qual transação (UT1) possui cópia de X (X1) com tranca de cooperação. UT2 aloca X1 para empréstimo. A factory de objetos cria e devolve uma cópia do objeto (X2) para Pedro. A figura 4.15 mostra o cenário de cooperação de objetos.

João executa o checkpointing para sua transação. Automaticamente, GT dispara o checkpointing de UT2, executando as duas fases do mecanismo. A primeira fase prepara os objetos da transação. A segunda fase valida os objetos. O diagrama de seqüência da figura 4.16 exemplifica este cenário.

Após uma falha, o usuário João executa a recuperação para sua transação. O meca-



A, B e C - objetos na área pública
A' e B' - objetos na área da transação Tg
A" - objeto na área da transação Tj
B" - objeto na área da transação Tm
Tg - transação de grupo
Tj - transação de João
Tm - transação de Maria

Figura 4.12: Objetos da Aplicação.

nismo de recuperação cria uma nova transação GT com o estado da antiga transação GT. Além disso, cria um novo objeto X com o estado do antigo objeto X. Aloca o objeto X e insere as subtransações na lista de subtransações de GT, que permanece no estado exatamente anterior à falha. Libera X com tranca WRITE para a transação GT e remove as subtransações UT1 e UT2. Então, devolve GT restaurada para João. O cenário da figura 4.17 ilustra o cenário de recuperação de uma transação.

O coordenador do grupo, João, decide pelo término da transação de grupo, após verificar a ausência de participantes. Este cenário está no diagrama 4.18. Além disso, determina que os objetos do grupo sejam atualizados na área pública, definindo o tipo de término como COMMITTING. O objeto Y, hipoteticamente alocado com tranca de leitura, é descartado. O objeto X, alocado anteriormente com tranca W-CONCESSION,

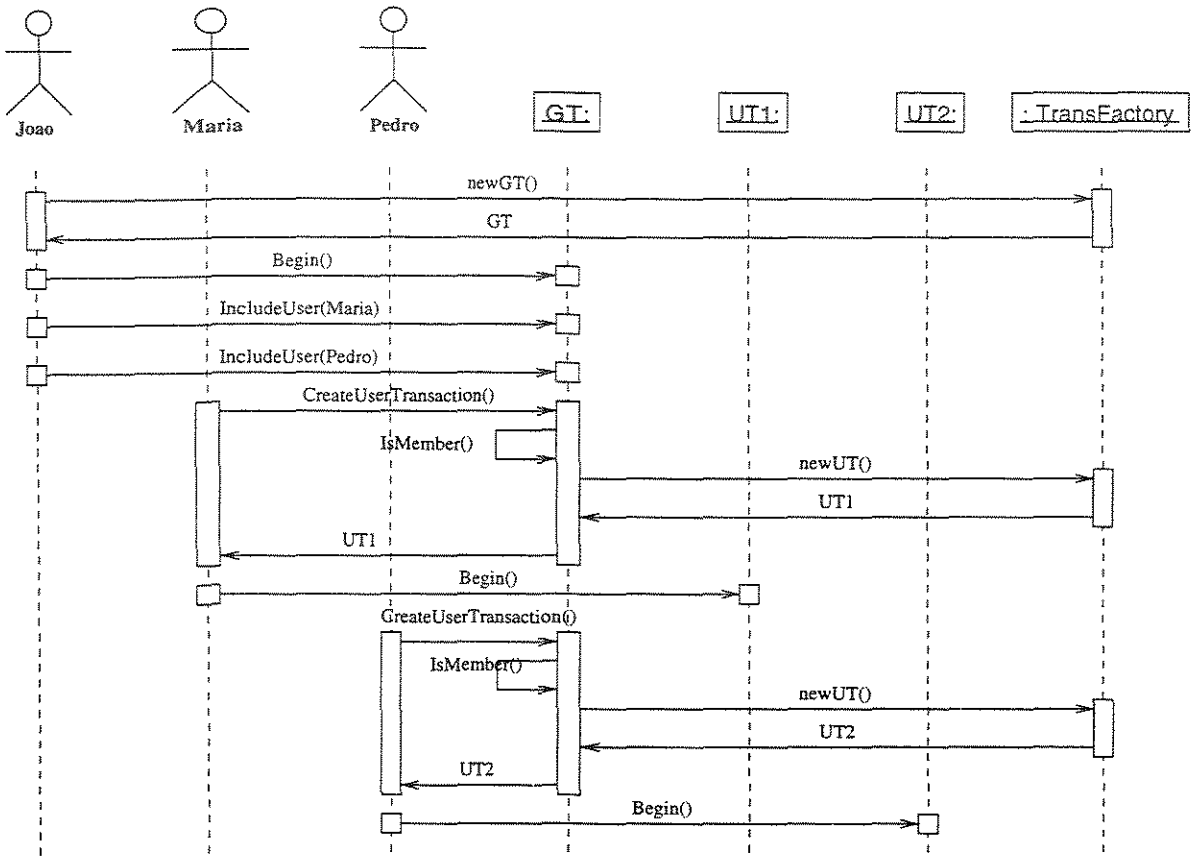


Figura 4.13: Diagrama de criação de uma transação de grupo com duas transações de usuário.

é salvo na área local da transação (primeira fase do protocolo de validação) e, posteriormente, atualizado na área ascendente (segunda fase do protocolo de validação). A transação remove seu log e sua área de trabalho.

4.8 Comparação com Outros Modelos

A estrutura hierárquica do Serviço de Transações Cooperativas assemelha-se com a estrutura de transações dos modelos de Klahold, de Fernandez e de Nodine [KSUW85, FZ89, NRZ92], introduzindo uma organização hierárquica de áreas de trabalho para as transações.

O Serviço define trancas estendidas de maneira parecida como ocorre nos modelos de Klahold e de Fernandez.

E possui facilidades de cooperação, na medida que cria operações especiais para troca de informação, como nos modelos de Klahold, CoAct [RKT+95] e de Fernandez.

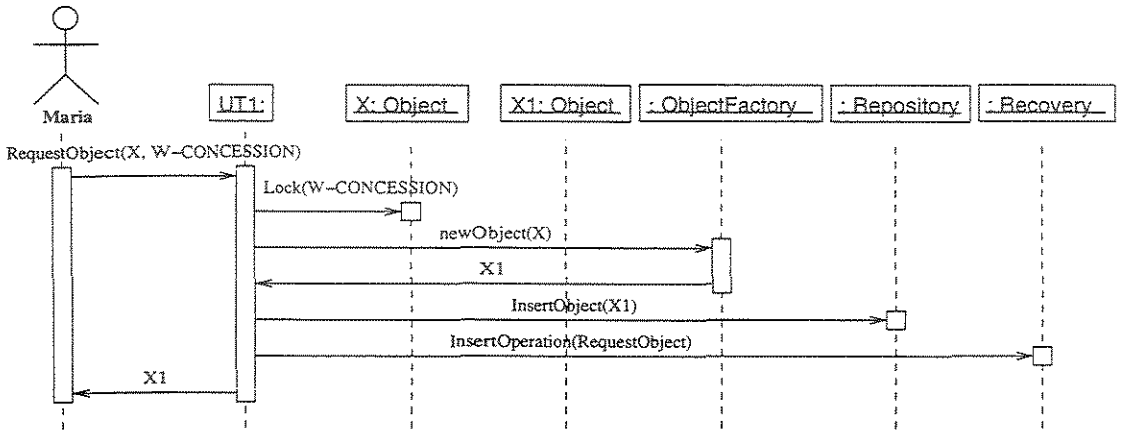


Figura 4.14: Diagrama de alocação de um objeto.

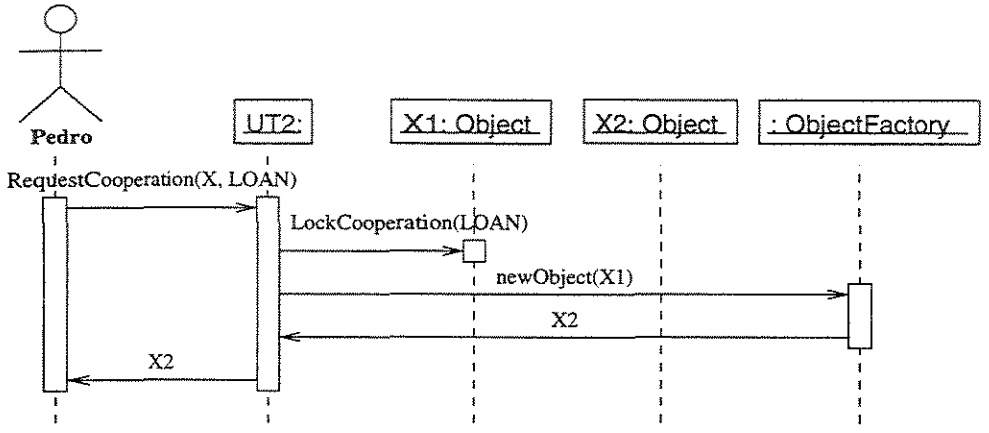


Figura 4.15: Diagrama de cooperação entre duas transações de usuário.

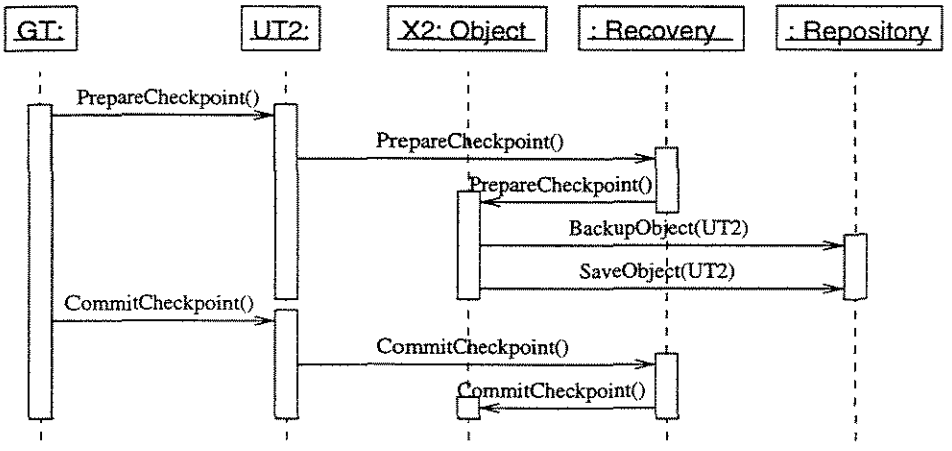


Figura 4.16: Diagrama de checkpointing de uma subtransação.

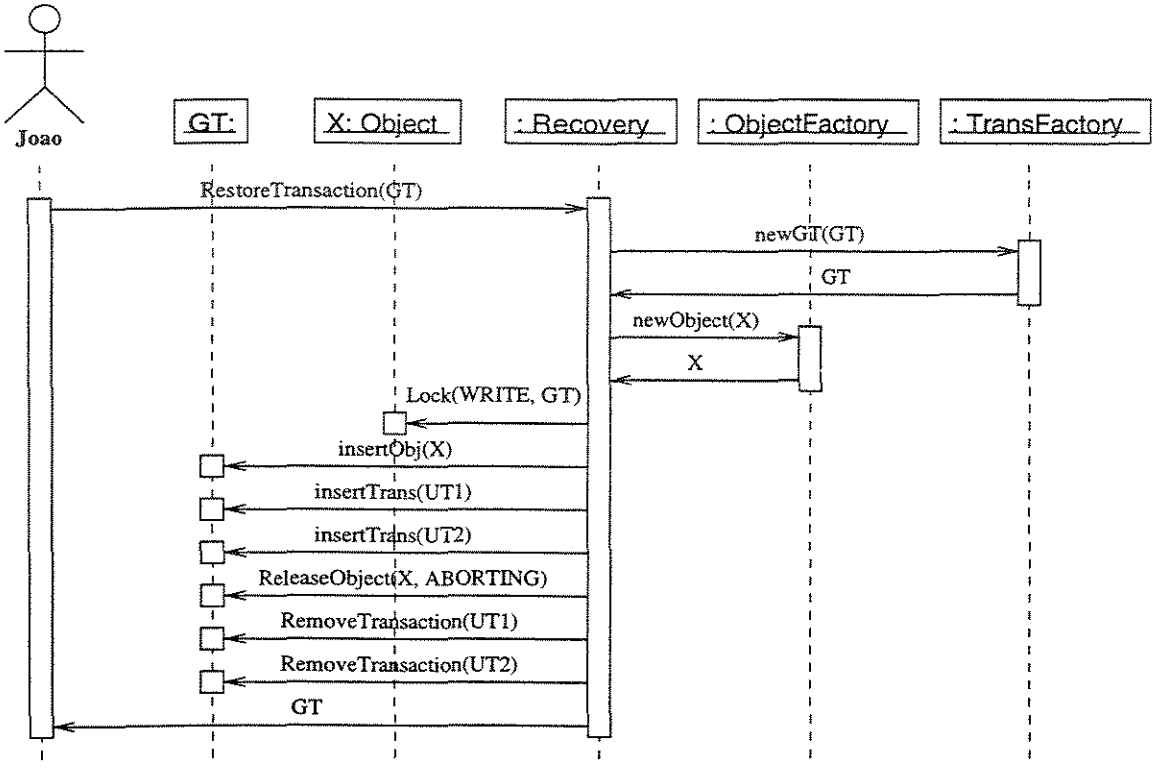


Figura 4.17: Diagrama de recuperação de uma transação.

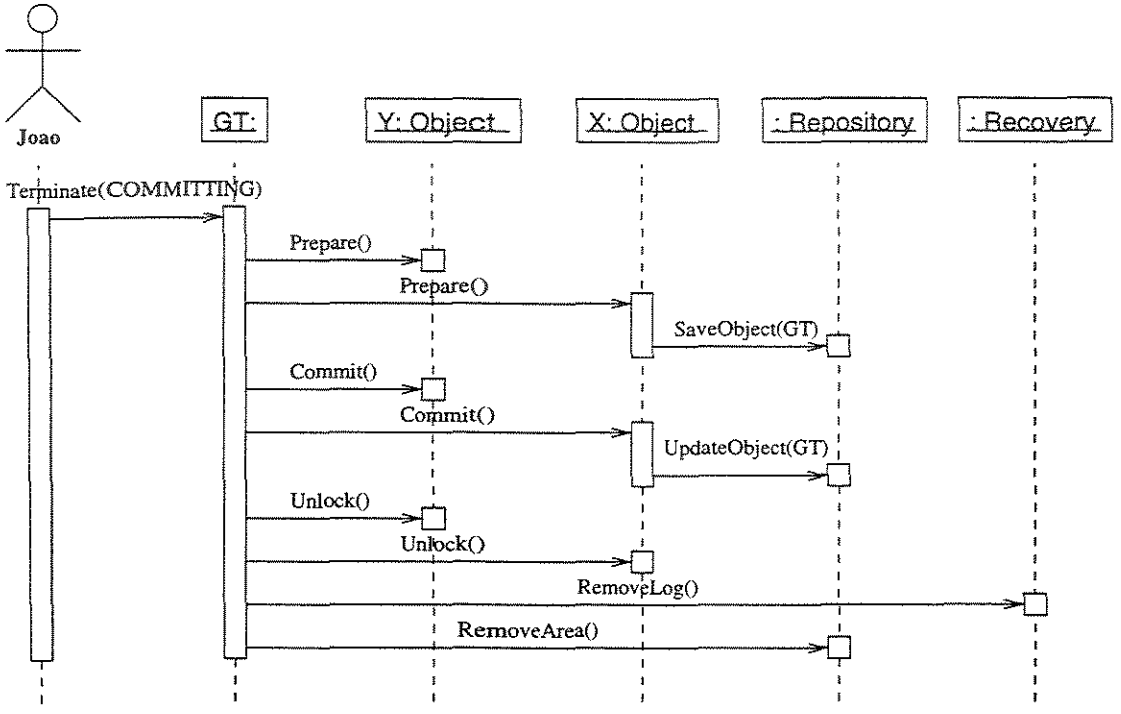


Figura 4.18: Diagrama de término de uma transação.

Capítulo 5

Implementação do Serviço de Transações Cooperativas

5.1 Introdução

Os objetivos deste capítulo são introduzir o ORB OrbixWeb utilizado, esclarecer detalhes de implementação e apresentar uma seqüência de testes sobre a implementação, gerando resultados da manipulação.

Na seção seguinte, é apresentado o OrbixWeb. Na seção 5.3, considerações sobre a implementação são realizadas. Na última seção, são apresentados os testes com os resultados alcançados.

5.2 OrbixWeb

O software escolhido para implementar o Serviço de Transações Cooperativas foi o OrbixWeb da empresa Iona. Foi escolhido dentre alguns importantes como o Java IDL da empresa JavaSoft e o Visibroker da empresa Borland/Visigenic. Iona é líder de produtos com tecnologia CORBA. Seu ORB C++, chamado Orbix, é executado em mais de 20 sistemas operacionais. ORBs da Iona permitem tanto protocolo IIOP como protocolo proprietário. Em julho de 1996, Iona apresentou seu novo produto OrbixWeb V1.0, uma implementação Java do lado do cliente do Orbix. No final de 1996, lançou o lado servidor em Java na implementação da versão 2.0. Em dezembro de 1997, Iona lançou o OrbixWeb 3.0 [OH98].

Esta última versão implementa completamente o mapeamento IDL para Java. O servidor oferece várias políticas de múltiplas *threads* e ativação dinâmica entre várias máquinas virtuais Java. O produto provê segurança baseado em SSL (Secure Socket

Layer). Applets podem chamar servidores IIOP através de *firewalls* e servidores podem fazer *callback* em applets. Além disso, provê diversos serviços CORBA como de Nome, Negociação e Transações. Provê também extensões do padrão CORBA para a persistência automática de objetos (Loader).

A versão utilizada do software nos laboratórios do Instituto de Computação da UNICAMP foi a versão 3.1.

5.3 Considerações

A metodologia adotada para o desenvolvimento do Serviço de Transações Cooperativas define que a implementação do modelo de classes deve usar a definição de interfaces em IDL. O apêndice A mostra as interfaces usadas no Serviço.

5.3.1 Objetos de Aplicação

Novos tipos de objeto podem estar disponíveis no Serviço de Transações Cooperativas. Para isto, os desenvolvedores da aplicação devem projetar classes que herdam da interface `ObjectManager`. Estas classes devem implementar métodos auxiliares que não estão definidos na interface, mas na classe de implementação desta interface. Estes métodos têm comportamento específico por aplicação, como, por exemplo, o modo como os atributos do objeto são serializados, e estão fora do escopo do Serviço de Transações Cooperativas.

Como parte do desenvolvimento de novos tipos de objeto, devem-se definir factories. Para tanto, estas factories devem herdar da interface `ObjectFactory` e implementar o método que cria objetos (`newObject`).

Outra particularidade do Serviço está na manipulação dos objetos, que podem ser criados pelas transações e, não necessariamente, alocados a partir de um repositório de objetos inicial. Se os objetos forem criados pelas transações de usuário, eles não sofrem cooperação e estão disponíveis somente para a transação que os criou. Eles estarão disponíveis para as outras transações quando estiverem em uma transação de grupo ou no próprio repositório de objetos. Tanto o controle de concorrência tradicional quanto a visibilidade destes novos objetos entre as transações da hierarquia são consistentes do ponto de vista do Serviço. De fato, não existem objetos disponíveis no repositório de objetos de uma aplicação quando o Serviço de Transações Cooperativas é executado pela primeira vez. Cabe à aplicação cooperativa criar um conjunto adequado de objetos e ao Serviço, em última instância, armazenar estes objetos na área persistente e pública da aplicação.

5.3.2 Distribuição das Transações

A arquitetura do Serviço de Transações Cooperativas define servidores distribuídos de transações no ambiente colaborativo. Cada servidor cria e destrói transações independentemente dos outros servidores. A definição do local de execução do servidor da transação deve ser especificado em tempo de execução e as informações relativas às transações e aos objetos devem manter-se consistentes e disponíveis pela rede. Para tanto, os marcadores das transações e dos objetos são armazenados no repositório, que é consultado toda vez que uma requisição de criação de transação ou objeto no Serviço acontece.

A transparência de distribuição das transações mais o suporte à recuperação implica em que os servidores de transação executam independentemente entre si e permitem que a recuperação das transações seja realizada por servidor. Assim, uma transação executando em um servidor dedicado pode falhar e ser recuperada sem afetar a execução das outras transações.

Uma transação de grupo, por exemplo, lista quais são suas subtransações. Caso alguma delas não responda, a transação de grupo recebe a mensagem com a falha de execução. Mesmo assim, ela pode continuar executando outras operações, como requisitar novos objetos. Futuramente, listará suas subtransações com sucesso se a subtransação mencionada tiver sido recuperada.

Caso alguma transação tiver problema de execução, o usuário pode requisitar ao servidor algum tipo de correção, até mesmo sua reinicialização. Neste caso, todas as transações do servidor devem ser restauradas ou ficarão inacessíveis para as outras transações do grupo.

5.3.3 Detalhes de Codificação

A implementação do Serviço de Transações Cooperativas pode ser avaliada em função do esforço de codificação realizado. As tabelas 5.1, 5.2 e 5.3 mostram a quantidade de linhas de código parcial e total e o número de classes implementadas parcial e total, inclusive, das classes utilitárias, auxiliares e das interfaces gráficas desenvolvidas para os testes. As colunas das tabelas com o título "Geradas" correspondem às linhas e às classes geradas automaticamente pelo compilador IDL do OrbixWeb.

5.4 Testes e Resultados

Para a fase de testes do Serviço de Transações Cooperativas foi criado um conjunto de interfaces gráficas de usuário (GUI) para acessar as transações cooperativas. Uma GUI em formato de janela foi desenvolvida para a transação de grupo e outra para a transação

	Geradas	Escritas	Total
Com comentários	13858	6943	20801
Sem comentários	13384	5616	19000
Número de classes	182	17	199

Tabela 5.1: Linhas de código do Serviço de Transações Cooperativas.

	Geradas	Escritas	Total
Com comentários	-	2453	2453
Sem comentários	-	2110	2110
Número de classes	-	7	7

Tabela 5.2: Linhas de código das Interfaces Gráficas.

de usuário. A seguir, são detalhadas as interfaces gráficas desenvolvidas para os testes e é exemplificada a utilização do Serviço.

5.4.1 As Interfaces Gráficas

A janela que representa uma transação de grupo encontra-se na figura 5.1. Possui quatro regiões distintas de operações: usuários, subtransações, objetos e transação. A região de usuários possui botões referentes às operações de inclusão e remoção pelo coordenador de membros do grupo. A região de subtransações contém botões relativos às operações de criação de uma transação de grupo ou de usuário sob sua responsabilidade e de listagem de suas subtransações. A região de objetos contém as operações que manipulam os objetos. E, finalmente, a região da própria transação contém as operações de inicialização e término da transação e as operações de armazenamento e restauração do estado da transação.

Além destas regiões, existe uma caixa de texto no topo da janela através da qual o usuário acompanha o estado de sua transação.

	Geradas	Escritas	Total
Com comentários	13858	9396	23254
Sem comentários	13384	7726	21110
Número de classes	182	24	206

Tabela 5.3: Totalização das linhas de código.

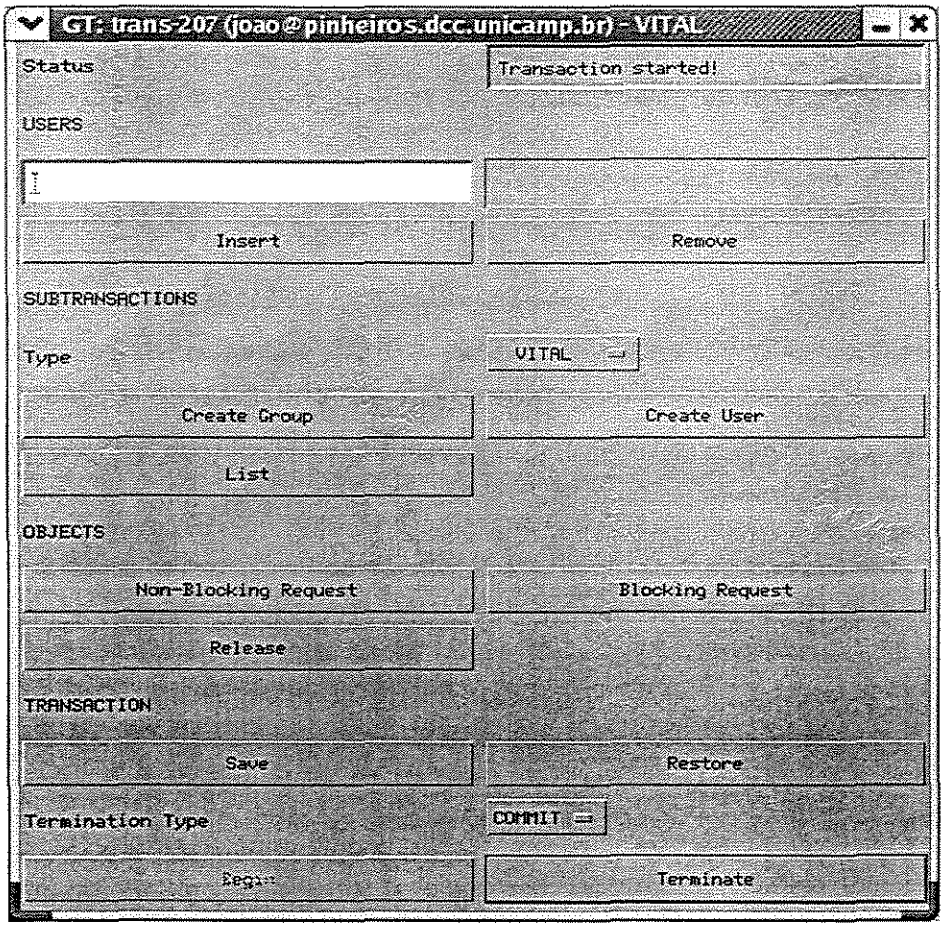


Figura 5.1: GUI para a manipulação de uma transação de grupo.

A janela que representa uma transação de usuário é apresentada na figura 5.2. Como a transação de usuário não possui subtransações, sua janela contém somente as regiões de operações que manipulam os objetos e a própria transação.

A região de usuários da janela de transação de grupo possibilita que um usuário participe do grupo. Para tanto, o coordenador do grupo registra o usuário no grupo através do botão "Insert". Além disto, existe uma lista de usuários do grupo de onde os usuários poderão ser excluídos através do botão "Remove".

As operações relacionadas com as subtransações estão separadas numa região específica da janela da transação de grupo. Nela, existem três operações: criação de transação de usuário, criação de transação de grupo e listagem das subtransações. A primeira é representada pelo botão "Create User" e tem como finalidade criar uma janela para a nova subtransação de usuário. A segunda é representada pelo botão "Create Group" e cria uma janela para a nova subtransação de grupo cujo coordenador é o próprio coordenador da transação de grupo que a criou. A listagem das subtransações é obtida quando o

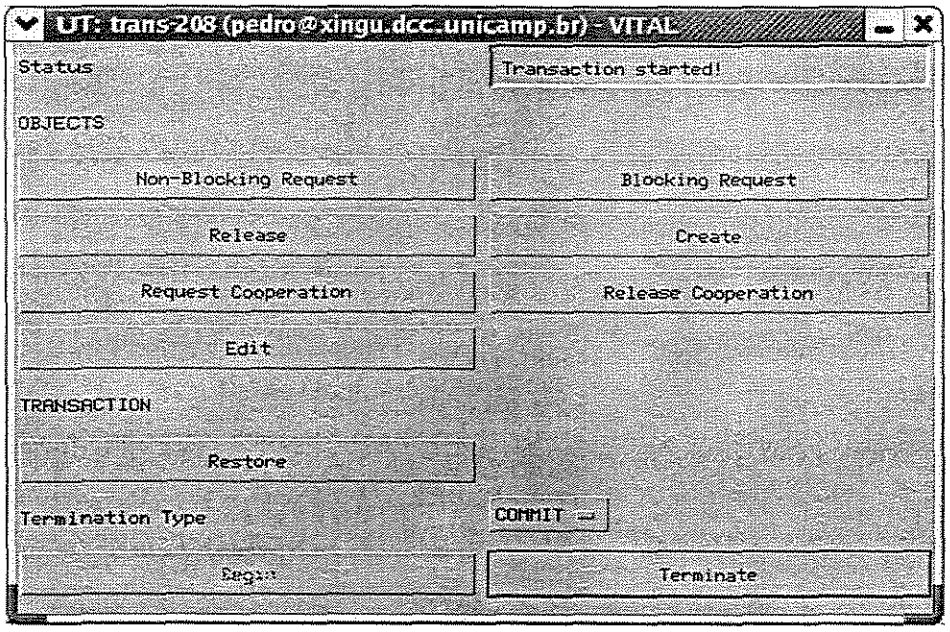


Figura 5.2: GUI para a manipulação de uma transação de usuário.

botão "List" é usado. A janela que mostra as subtransações presentes em uma transação de grupo está na figura 5.3.

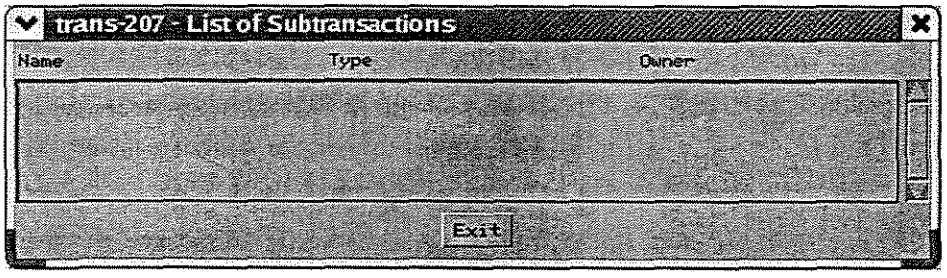


Figura 5.3: GUI para a listagem de subtransações das transações de grupo.

Se um usuário deseja criar um objeto e inserí-lo na área de trabalho de sua transação, deve utilizar o botão "Create" da região da janela destinada à manipulação dos objetos. A janela da criação de objetos é mostrada na figura 5.4. Nela, o usuário escolhe qual o tipo de *factory* de objetos. Se o usuário não prover a factory apropriada, objetos do tipo Counter são criados que foram os objetos utilizados nos testes. A funcionalidade de criação de objetos está apenas disponível para as transações de usuários.

Quando um usuário deseja requisitar um objeto, utiliza os botões "Non-Blocking Request" ou "Blocking Request". A janela responsável pela requisição de objetos é apresentada na figura 5.5. O usuário escolhe um objeto disponível na área da transação de

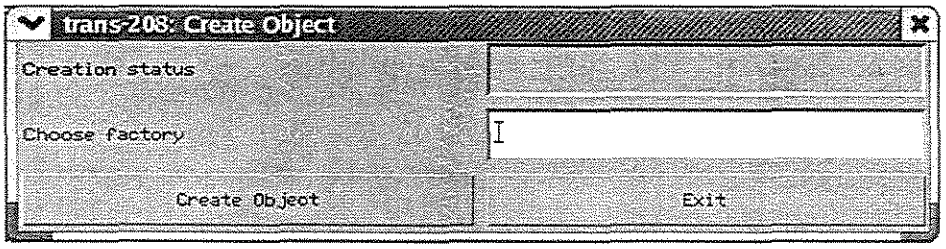


Figura 5.4: GUI para a criação de objetos.

grupo ou na área pública, caso o usuário pertença a uma transação raiz, e o aloca para si com o tipo de tranca preferido através da referência "Request Type". O usuário pode verificar quais são as trancas sobre os objetos.

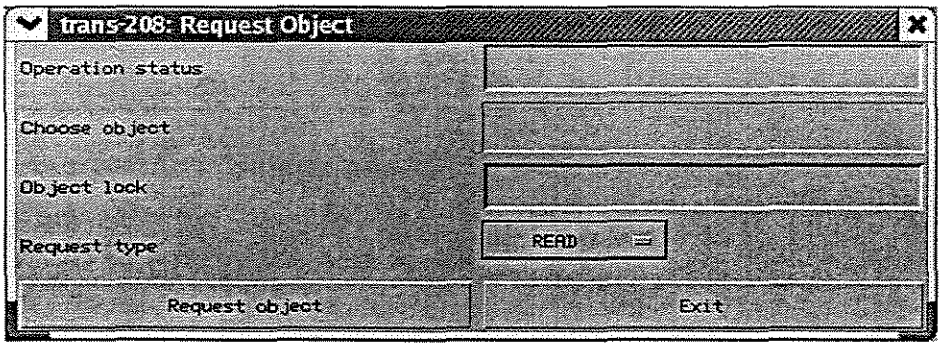


Figura 5.5: GUI para a requisição de objetos.

Se um usuário deseja requisitar um objeto para cooperação, utiliza o botão "Request Cooperation". A janela responsável pela requisição de objetos para cooperação é mostrada na figura 5.6. O usuário verifica qual o objeto na área da transação de grupo que permite cooperação e o aloca para si com o tipo de tranca cooperativa preferido através da referência "Request Type". Esta opção está apenas presente nas transações de usuário (figura 5.2).

Depois de requisitar um determinado objeto, o usuário pode editá-lo pressionando o botão "Edit" da região de objetos da janela. Na nova janela de edição de objetos o usuário pode utilizar as operações do objeto para modificar seu estado. Somente os objetos que não estão alocados para leitura e cópia podem ser editados pela transação. No caso da figura 5.7 o tipo de objeto presente na janela é Counter. A funcionalidade de edição de objetos está apenas disponível para as transações de usuário.

A janela para a liberação de objetos encontra-se na figura 5.8. Nela, é possível escolher os objetos alocados pela transação que serão liberados para a transação de grupo ou para a área pública se a transação estiver na raiz da hierarquia de transações. O botão "Release" na região de objetos da janela da transação é o responsável pela criação desta

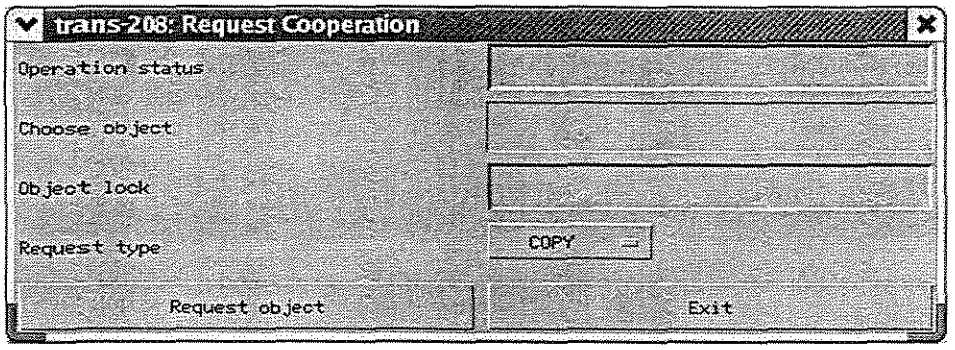


Figura 5.6: GUI para a requisição de objetos para cooperação.

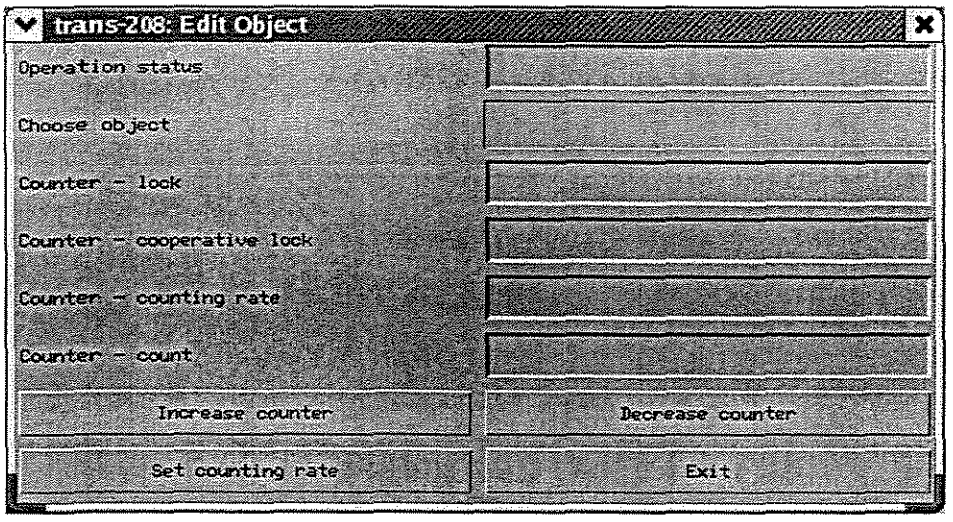


Figura 5.7: GUI para a edição de objetos.

janela. O tipo de término dos objetos é definido pelo parâmetro "Release Type" com os valores COMMIT ou ABORT. A liberação de objetos é executada com sucesso para as transações de usuário que não têm seus objetos adquiridos para cooperação (cópia, empréstimo e concessão) ou para as transações de grupo que não têm os objetos alocados por suas subtransações.

Se uma transação deseja liberar um objeto requisitado para cooperação deve acionar o botão "Release Cooperation" (figura 5.9) . Note que esta operação só está disponível entre transações de usuário no mesmo grupo. As trancas que são denominadas "Object lock" são as trancas cooperativas que estão no objeto que ofereceu a cooperação. "Object cooperative lock" representa as trancas que estão no objeto a ser liberado.

As operações específicas de uma transação estão na região inferior da janela da transação. A primeira delas diz respeito à inicialização da transação (botão "Begin"). A segunda operação finaliza a transação (botão "Terminate"). O tipo de terminação que o usuário

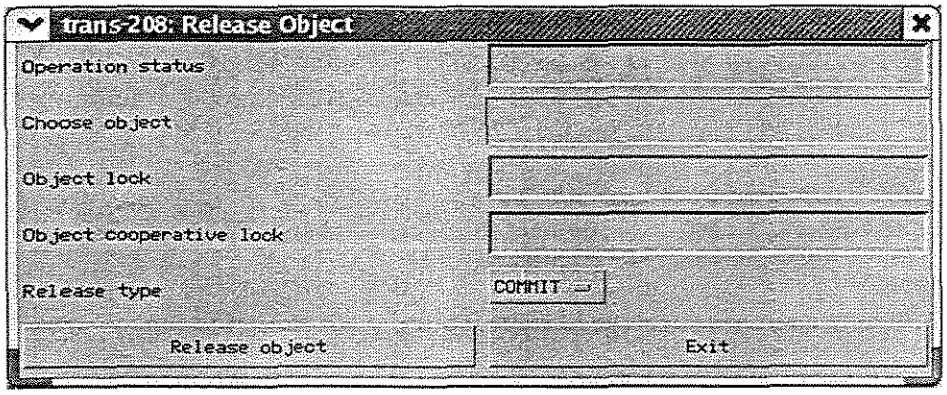


Figura 5.8: GUI para a liberação de objetos.

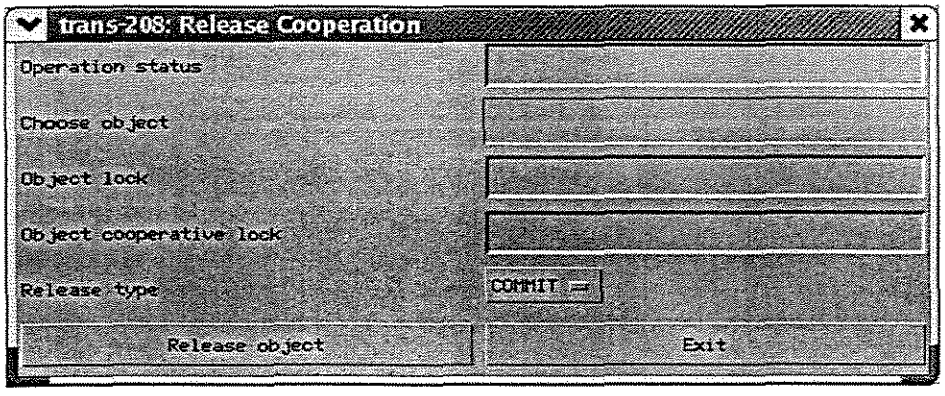


Figura 5.9: GUI para a liberação de objetos adquiridos para cooperação.

pode escolher para sua transação depende do parâmetro escolhido em "Termination Type" (COMMIT ou ABORT). Existem dois outros botões nesta região da janela. Um deles é "Save" que armazena o estado da transação em memória persistente. Ao acionar o outro botão ("Restore"), o Serviço restaura a transação recuperando-a de qualquer falha no sistema. Neste caso, a falha representa qualquer tipo de erro de comunicação com o ORB, supondo desta maneira que o servidor não esteja executando. Na prática, o usuário pode salvar a transação quantos vezes forem necessários e restaurá-la apenas quando o servidor for reinicializado.

5.4.2 Exemplos de Utilização do Serviço de Transações Cooperativas

Os testes realizados foram divididos em diversos cenários, que correspondem às funcionalidades oferecidas pelas interfaces gráficas do Serviço de Transações Cooperativas. Os cenários representam um conjunto de tarefas em um ambiente colaborativo com transações

cooperativas.

Uma breve introdução sobre como o Serviço é utilizado está na seção abaixo. Os cenários são apresentados posteriormente.

Introdução

O Serviço de Transações Cooperativas é executado usando-se a aplicação cliente chamada "application", que inicializa o servidor de objetos correspondente. Esta aplicação provê a criação de transações de grupo e de usuário. Para tanto, existe um conjunto de opções de parâmetros que deve ser considerado na chamada da aplicação:

- `-u <user_name>` (obrigatório): define o usuário da transação.
- `-GT` (obrigatório): cria uma transação de grupo.
- `-UT` (obrigatório): cria uma transação de usuário.
- `<group_transaction_name>@<group_transaction_host_name>` (opcional): define o nome da transação de grupo pai e o nome da máquina na qual está executando.
- `-v` (opcional): define a transação como não vital. Se omitido o valor corresponde a vital.
- `-h <host_name>` (obrigatório): define em qual máquina do sistema a transação executará.

Como conseqüência das opções de inicialização, é possível criar uma transação de usuário filha de uma transação de grupo ou na raiz da hierarquia de transações.

Cenários

Os testes foram divididos em cenários. Cada cenário é descrito em função de passos executados cronologicamente. O primeiro cenário corresponde à criação de um conjunto de transações hierarquicamente distribuídas. O segundo cenário apresenta a manipulação dos objetos presentes na aplicação. O terceiro cenário exemplifica o uso da recuperação.

Estes cenários estão a seguir:

CRIAÇÃO DE TRANSAÇÕES

1. **Inicialização de uma transação de grupo.** Quando o usuário "joao" decide pela criação de uma transação de grupo, executa na sua área de trabalho:

```
>application -u joao -GT -h pinheiros
```

"aplication" é o nome do programa cliente do Serviço de Transações Cooperativas, o parâmetro "-u joao" identifica o nome do usuário da transação, o parâmetro "-GT" especifica a criação de um transação de grupo e o parâmetro "-h pinheiros" especifica a execução da transação na máquina "pinheiros". Desta forma, o ORB ativa o servidor do Serviço na máquina especificada e devolve uma janela de manipulação da nova transação de grupo para o usuário. Como o parâmetro "-v" é omitido da chamada, a transação é criada como vital.

O cabeçalho da janela das transações identifica o nome e o tipo da transação, o usuário e o nome da máquina na qual a transação está executando. Além disso, mostra se a transação é vital.

Antes da manipulação da transação, "joao" a inicializa.

2. **Inclusão de membros no grupo.** O coordenador da transação, "joao", inclui como participantes do grupo os usuários "maria" e "pedro". A GUI que exemplifica este passo do cenário está na figura 5.10.

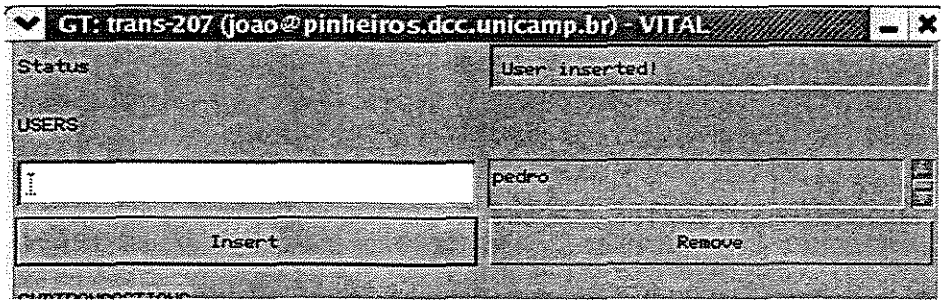


Figura 5.10: Inclusão de membros no grupo.

3. **Inicialização de transações de usuário.** Quando o usuário "pedro" decide participar da transação do "joao", cria uma transação de usuário usando o comando:

```
>application -u pedro -UT trans-209@pinheiros -h xingu
```

O parâmetro "-u pedro" determina o nome do usuário da transação, o parâmetro "-UT" especifica a criação de uma transação de usuário. O parâmetro seguinte diz que a transação de usuário participa da transação "trans-209" que executa na máquina "pinheiros". Após a verificação se a transação do "pedro" pode participar da transação de grupo, é ativado o servidor na máquina "xingu", que gera uma janela para a nova transação de usuário.

A usuária "maria" decide participar da transação do "joao", criando uma transação de usuário na mesma máquina da transação de grupo com o sequinte comando:

```
>application -u maria -UT trans-209@pinheiros -h pinheiros
```

4. **Verificação de subtransações.** Em certo momento "joao" deseja informar-se sobre quais as subtransações que estão participando no grupo. Para tanto, lista as subtransações que pertencem a transação de grupo. A GUI que exemplifica este passo do cenário está na figura 5.11.

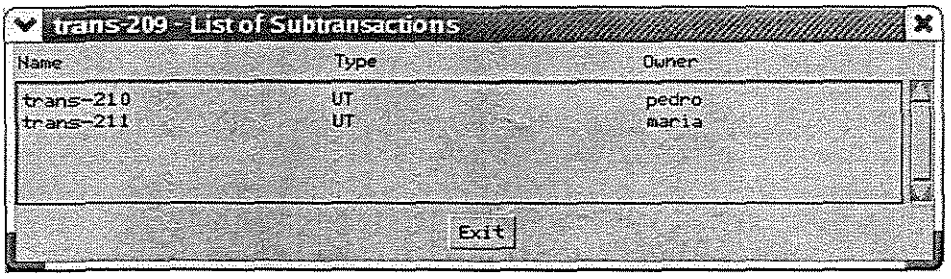


Figura 5.11: A lista das subtransações do grupo.

MANIPULAÇÃO DOS OBJETOS

1. **Requisição de objetos pela transação de grupo.** O coordenador do grupo resolve requisitar o objeto "counter-108" com a intenção de deixá-lo disponível na área do grupo. Desta forma, o objeto é requisitado para escrita. A cópia do objeto, "counter-184" é transferida para a área privada do grupo.
2. **Requisição de objetos pela transação de usuário.** O usuário "pedro" decide alocar do grupo o objeto "counter-184" com tranca que permite empréstimo. A cópia do objeto, "counter-185", é criada na área da transação do usuário.
3. **Edição de um objeto.** "pedro" edita seu objeto de modo a ter seus atributos modificados. Passa os valores do objeto de 1 para 43 e de 11 para 140. As figuras 5.12 e 5.13 mostram o estado do objeto antes e depois da operação.
4. **Cooperação de objetos.** "maria" decide pedir emprestado o objeto "counter-184" do grupo. De modo transparente, o grupo requisita à transação de "pedro" o objeto "counter-185" e retorna o objeto "counter-186" para a transação de "maria". A usuária modifica os valores iniciais do objeto de 140 para 226. As figuras 5.14 e 5.15 mostram os resultados das operações de cooperação e edição.
5. **Liberação de objetos adquiridos para cooperação.** Depois que as mudanças foram realizadas sobre o objeto, "maria" devolve-o para a transação de "pedro". A figura 5.16 ilustra o processo de liberação de um objeto com tranca cooperativa.

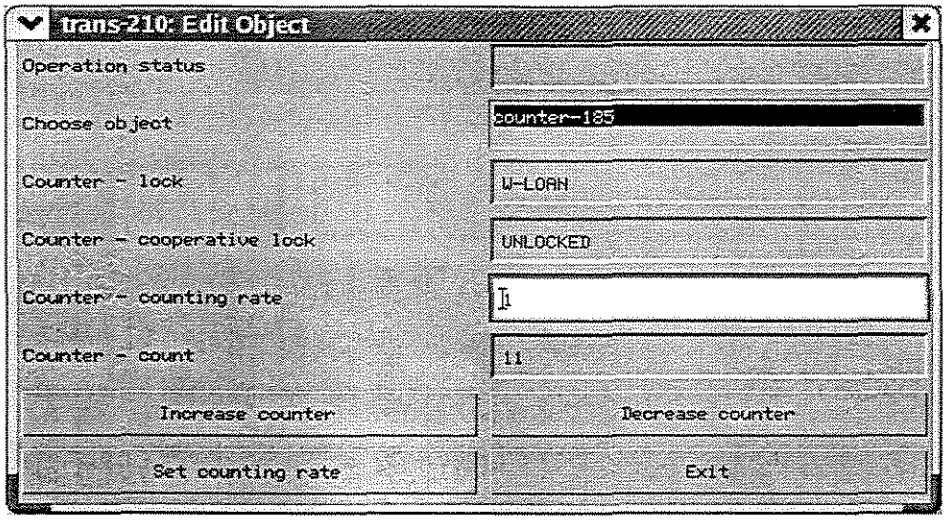


Figura 5.12: O objeto com o estado inicial.

6. **Verificação de um objeto devolvido.** O usuário "pedro" deseja verificar quais são os novos valores do objeto que foi devolvido após o empréstimo. Os novos valores aparecem na janela da figura 5.17.

RECUPERAÇÃO DAS TRANSAÇÕES

1. **Falha na execução da transação de grupo.** Em determinado momento, o servidor de transações, no qual a transação de grupo do usuário "joao" está executando, falha. Neste ponto, a aplicação perde a conexão com o servidor e também falha. O usuário tenta listar as subtransações que pertencem ao grupo, mas não obtém sucesso. Qualquer operação futura sobre a transação levantará uma exceção. Sem a descoberta da causa da falha no sistema, o ORB é reinicializado.

2. **Recuperação da transação de grupo.** "joao" verifica que o servidor de transações está executando normalmente e faz a recuperação de sua transação. Logo após, lista as subtransações com sucesso.

3. **Armazenamento do estado das transações.** O usuário da transação de grupo resolve salvar o estado da sua transação, das suas subtransações e os do objetos pertencentes a ela para que as mudanças sobre eles não sejam perdidas caso ocorra uma falha no sistema.

4. **Falha na execução da transação de usuário.** O servidor de transações, no qual a transação de "pedro" está executando, falha. Neste momento, a

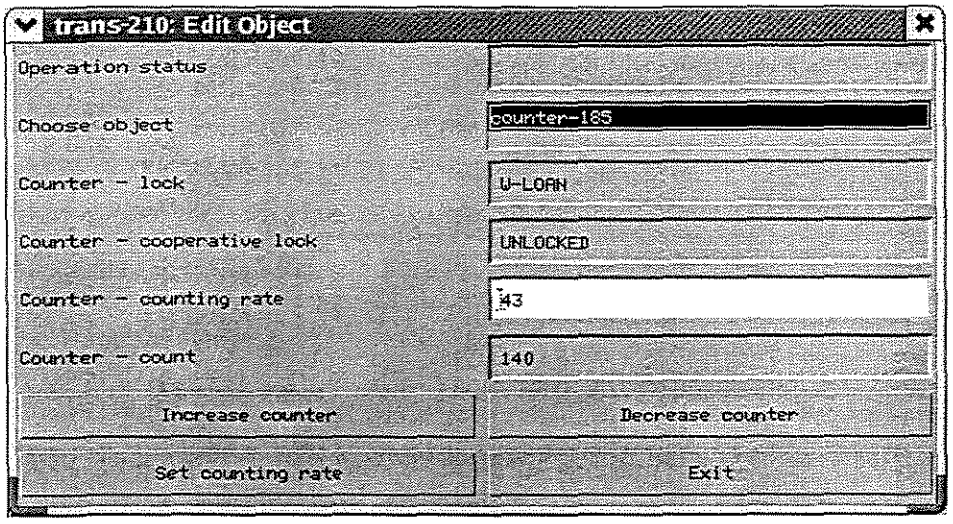


Figura 5.13: O objeto com o estado modificado.

aplicação perde a conexão com a transação. "pedro" requisita que o responsável pelo ORB reinicialize-o.

5. **Recuperação da transação de usuário.** Após certo tempo, o usuário "pedro" restaura a transação e tenta editar um de seus objetos. Ele verifica que o estado dele se manteve após a falha, graças à persistência da transação.

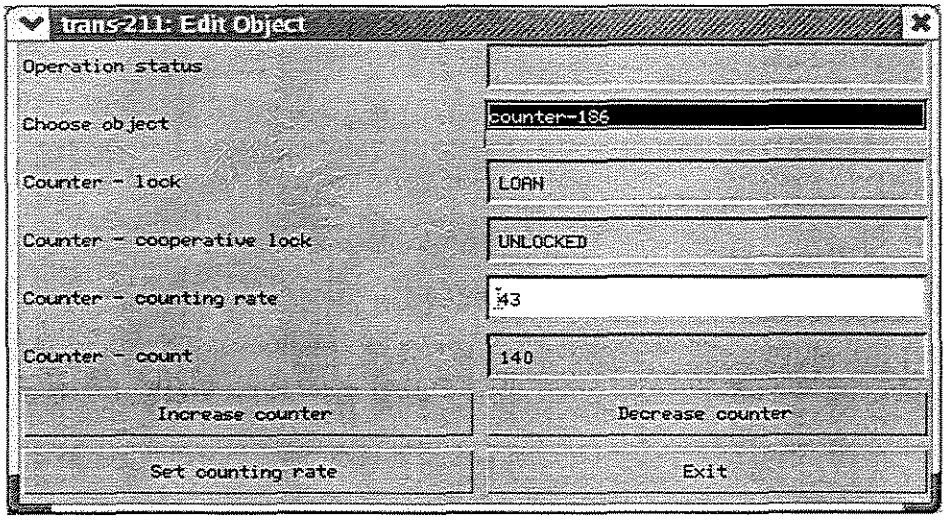


Figura 5.14: O objeto pedido emprestado.

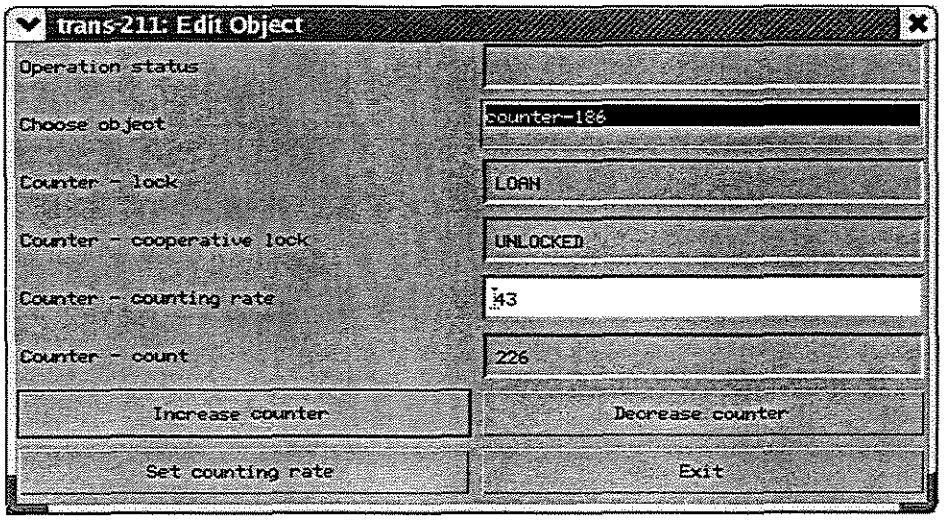


Figura 5.15: O objeto modificado após o empréstimo.

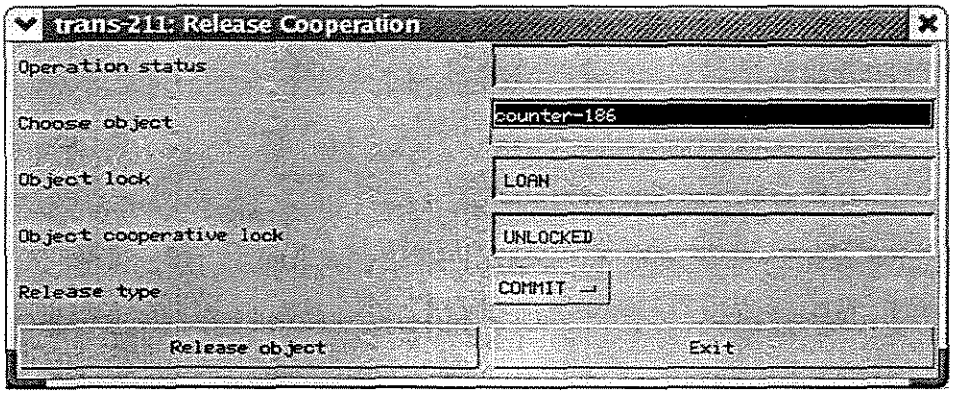


Figura 5.16: A lista dos objetos do grupo.

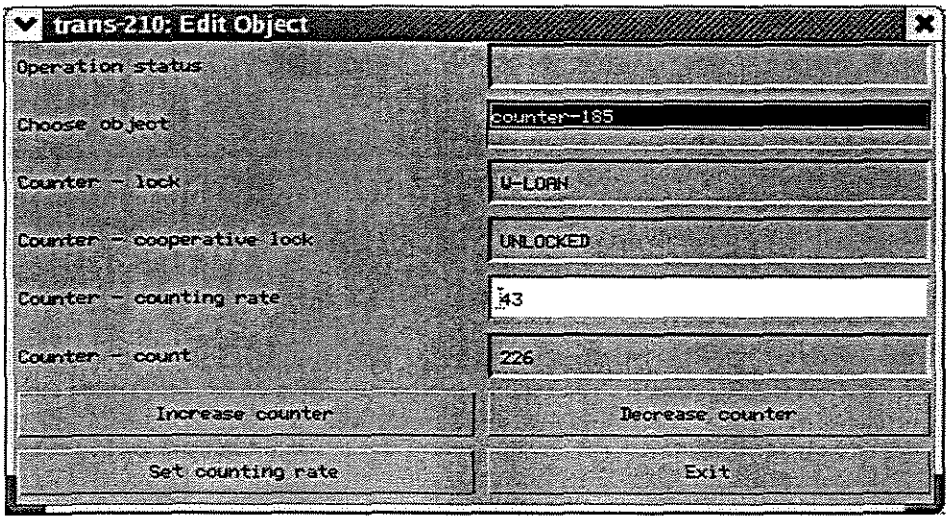


Figura 5.17: O objeto devolvido modificado.

Capítulo 6

Conclusão

O Serviço de Transações Cooperativas proposto nessa dissertação estrutura as transações em uma hierarquia. Os diversos níveis da hierarquia são organizados de acordo com os grupos de transação criados. Os nós internos da hieraquia representam as transações de grupo e os nós do tipo folha representam as transações de usuário.

Se os objetos são transferidos entre dois níveis da hierarquia de transações (mecanismo de check-in/check-out), o controle de concorrência é tradicional com trancas do tipo leitura e escrita. Se os objetos são transferidos entre transações de usuário de um mesmo grupo, as trancas utilizadas estendem as trancas tradicionais para permitir cooperação.

Em certo momento da utilização do Serviço, é natural que um determinado objeto possua várias cópias distribuídas pela hierarquia de transações. Pela própria estrutura em árvore da hierarquia, a cópia mais atualizada é aquela presente no nível mais descendente ou nas transações do tipo folha. Porém, a cópia que é válida para o Serviço é aquela presente no nível mais ascendente ou na raiz da hierarquia, isto porque o sentido de atualização do objeto é do nível mais descendente para o nível mais ascendente. Assim, pode-se dizer que a cópia do objeto mais consistente é aquela presente no nível mais ascendente. De fato, as outras cópias do objeto são apenas estágios intermediários de trabalho de grupo e podem ser descartadas.

O mecanismo de controle de concorrência oferece uma abordagem flexível na manipulação dos objetos na transação. Adiciona trancas cooperativas e disponibiliza operações especiais que garantem colaboração entre os usuários.

O mecanismo de recuperação utiliza um protocolo de checkpointing que sincroniza o estado de todas as transações pertencentes a uma transação raiz. O processo de recuperação utiliza-se do log para restaurar o estado da transação salvo no último checkpoint e cancelar as operações que a transação executou depois do último checkpoint.

Em relação às transações, há alguns anos atrás CORBA definiu um padrão para transações tradicionais. Recentemente, com o Serviço de Mecanismos de Estruturação

de Transações, tem especificado um serviço de objetos (CORBAServices) que permite a criação de transações aninhadas e compensatórias, como ocorre em Sagas [GMS87]. Nessa direção, o Serviço de Transações Cooperativas pode ser visto como uma possível extensão aos serviços de objetos de transações existentes no CORBA. A seguir são apresentados as contribuições e trabalhos futuros.

6.1 Contribuições

O gerenciamento de transação no Serviço é similar a outros modelos [US92, FZ89] em relação ao suporte de transações hierárquicas para aplicações de desenvolvimento de projeto. Entretanto, possui novas características para transações abertas com trabalho cooperativo, como a inclusão de trancas cooperativas que permitem a transferência de trabalho não finalizado entre os usuários.

Outra contribuição consiste na inclusão desse Serviço de Transações na arquitetura CORBA e sua validação através de uma implementação que usou OrbixWeb como plataforma de comunicação.

6.2 Trabalhos Futuros

A integração com um mecanismo síncrono de interação adicionaria ao Serviço uma forma alternativa de cooperação, a qual favorece a total interação entre usuários que trabalham em cooperação. Uma destas formas poderia ser o mecanismo de sessão, onde o coordenador da sessão definiria os objetos no contexto da sessão e receberia pedidos de inclusão e remoção de usuários, que, por sua vez, manipulariam os objetos em turnos.

Além disso, mais facilidades de cooperação poderiam ser tratadas, como a utilização de canais de comunicação para melhorar a interação entre as transações que participam da colaboração, a extensão do mecanismo de cooperação assíncrono para todas as transações ou mesmo a criação de mais operações especiais de cooperação, inclusive com novas trancas.

A implementação do Serviço pode ser modificada para prover maior portabilidade entre implementações de ORB. Do ponto de vista da criação e localização dos objetos, o conceito de marcadores pode ser substituído pelo serviço de objetos existente em CORBA chamado Serviço de Nome. Em relação à persistência dos objetos, a utilização do mecanismo de loader do OrbixWeb pode ser substituído pelo Serviço de Estado Persistente de CORBA.

Poderíamos utilizar, também, um mecanismo de checkpointing mais eficiente.

Os testes realizados sobre o Serviço de Transações Cooperativas demonstraram como os aspectos funcionais das transações foram validados, porém nenhum estudo de desempenho

foi realizado.

O Serviço de Transações Cooperativas poderia atender a uma gama maior de aplicações se fosse integrado a um sistema de workflow que incluiria noções como fluxo de controle e outros tipos de atividades além de transações.

O compartilhamento dos objetos pode provocar deadlocks entre as transações que participam de um grupo, ora pela requisição dos objetos do grupo, ora pela cooperação dos objetos entre as transações de usuário. Este problema é foco de trabalhos futuros.

Bibliografia

[AAA+96] G. Alonso, D. Agrawal, E. Abbadi, M. Kamath, R. Gunthor, and C. Mohan. Advanced Transaction Models in Workflow Contexts. *Proc. of 12nd Int. Conf. Data Engineering*, February 1996.

[ABAK95] D. Agrawal, J. Bruno, El Abbadi, and V. Krishnaswamy. Managing concurrent activities in collaborative environments. In *CoopIS*, 1995.

[Bak97] S. Baker. *CORBA Distributed Objects Using Orbix.* Addison-Wesley, 1997.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, 1987.

[BKK85] F. Bancilhon, W. Kin, and H. F. Korth. A Model of CAD Transactions. *Proc. of 11th VLBD, Stockholm*, pages 25–33, 1985.

[COR02] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 2002. OMG Document Number 02.06.33, Version 3.0.

[DHL90] U. Dahal, M. Hsu, and R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. *ACM SIGMOD*, 1990.

[EGR91] C. A. Ellis, S. J. Gibbs, and G. L. Rein. Groupware: Some Issues and Experiences. *Communication of the ACM*, 1(34):39–58, January 1991.

[Elm92] A. K. Elmagarmid. *Transaction Models for Advanced Database Application.* Morgan Kaufmann, 1992.

[FZ89] M. Fernandez and S. Zdonik. Transaction Groups: A Model for Controlling Cooperative Transactions. *3rd Int. Workshop On Persistent Object Systems*, January 1989.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. *ACM SIGMOD*, pages 249–259, 1987.

[GRJ99] B. Grady, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[HZ87] M. F. Hornik and S. B. Zdonik. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transaction on Office Information Systems*, 5(1):70–95, January 1987.

[ION97] IONA Technologies PLC. *OrbixWeb Programmer's Guide*, November 1997.

[KP92] G. E. Kaiser and C. Pu. Dynamic Restructuring of Transactions. *No editor A. K. Elmagarmid, Database Transaction Models for Advanced Aplications*, 1992.

[KSUW85] P. Klahold, G. Schlageter, R. Unland, and W. Wilkes. A Transaction Model Supporting Complex Applications in Integrated Information Systems. *ACM*, pages 388–401, 1985.

[Moh94] C. Mohan. Advanced Transaction Models - Survey and Critique. *ACM SIGMOD*, 1994.

[NG92] K. Narayanaswamy and N. Goldman. Lazy Consistent: A Basis for Cooperative Software Development. *CSCW Proceedings*, pages 257–264, November 1992.

[NRZ92] M. H. Nodine, S. Rasmaswamy, and S. B. Zdonik. A Cooperative Transaction Model for Design Databases. *No editor A. K. Elmagarmid, Database Transaction Models for Advanced Aplications*, pages 53–85, 1992.

[OH98] Robert Orfali and Dan Harkey. *Client/Server Programming with Java and CORBA.* John Wiley and Sons, Inc., 1998.

[OHE96] R. Orfali, D. HarKey, and J. Edwards. *The Essential Distributed Objects Survival Guide.* John Wiley and Sons, Inc., 1996.

[OMG] OMG Home. www.omg.org. Internet home page.

[OTS02a] Object Management Group. *Additional Structuring Mechanisms for the OTS*, September 2002. OMG Document Number 02.09.03, Version 1.0.

[OTS02b] Object Management Group. *Transaction Service Specification*, September 2002. OMG Document Number 02.08.07, Version 1.3.

[RKT+95] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wäsch, and P. Muth. Towards a Cooperative Transaction Model - The Cooperative Activity Model. *Proc. of 21st Conf. VLDB, Zurich*, 1995.

[SW90] S. K. Shrivastava and S. M. Wheater. Implementing Fault-Tolerant Distributed Aplications Using Objects and Multi-Coloured Actions. *ICDCS-10*, pages 1–9, June 1990.

[US92] R. Uland and G. Schlageter. A Transaction Manager Development Facility for Non Standard Database Systems . *No editor A. K. Elmagarmid, Database Transaction Models for Advanced Aplications*, 1992.

[VD98] A. Vogel and K. Duddy. *Java Programming With CORBA*. John Wiley and Sons, Inc., 1998.

[WR92] H. Watcher and A. Reuter. The ConTract Model. *No editor A. K. Elmagarmid, Database Transaction Models for Advanced Aplications*, 1992.

[ZNBB94] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhers. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. *ACM SIGMOD*, 1994.

Apêndice A

Interfaces do Serviço de Transações Cooperativas

As interfaces definidas neste apêndice correspondem às principais interfaces em IDL para o Serviço de Transações Cooperativas.

```
interface Skeleton {
  // Atributos

  // Operações
  boolean begin(in string name);
  boolean terminate(in short type);
  ObjectManager bckRequestObj(in long transId, in ObjectManager object, in
short lock);
  ObjectManager nonBckRequestObj(in long transId, in ObjectManager object, in
short lock);
  boolean releaseObject(in string objName, in short termType);
  boolean releaseAll(in short termType);
};
```

Figura A.1: IDL da classe Skeleton.

```
interface TransManager: Skeleton {
  // Atributos
  readonly attribute TransType type;
  readonly attribute long transId;
  readonly attribute short status;
  attribute GroupTrans parentTrans;
  readonly attribute string coordinator;
  readonly attribute boolean vital;
  attribute short groupStatus;
  readonly attribute StringList users;
  readonly attribute string hostName;
  attribute Repository repository;
  attribute Recovery recovery;

  // Operações
  boolean checkpoint();
  boolean prepareCheck();
  boolean commitCheck();
  boolean abortCheck();
  StringList getTrans();
  StringList getObjs();
  ObjectManager getObj(in string obj);
  ObjectManagerList getObjList();
  ObjectManager getObjByParent(in ObjectManager obj);
  void includeObj(in ObjectManager obj);
  boolean hasObj(in ObjectManager obj);
  boolean isSubTrans(in long transId);
};
```

Figura A.2: IDL da classe Cooperative Transaction.

```
interface GroupTrans: TransManager {
  // Atributos

  // Operações
  GroupTrans createGT(in string user, in string hostName, in GroupTrans
parentTrans, in boolean vital);
  UserTrans createUT(in string user, in string hostName, in GroupTrans
parentTrans, in boolean vital);
  boolean removeTrans(in TransManager subTrans);
  void includeTrans(in TransManager trans);
  boolean excludeTrans(in TransManager trans);
  TransManagerList getTransactions(in string name);
  void includeUser(in string user);
  void excludeUser(in string user);
  boolean isMember(in string user);
};
```

Figura A.3: IDL da classe Group Transaction.

```
interface UserTrans: TransManager {
  // Atributos

  // Operações
  ObjectManager reqCooperation(in string objName, in short typeReq);
  boolean releaseCoopObj(in string objName, in short termType);
};
```

Figura A.4: IDL da classe User Transaction.

```
interface ObjectManager {
  // Atributos
  readonly attribute short lock;
  readonly attribute short status;
  attribute short checkStatus;
  attribute ObjectManager parentObj;
  readonly attribute ObjectFactory factory;
  readonly attribute string hostName;

  // Operações
  string factoryName();
  boolean hasOwner();
  long isOwner (in long transId);
  LongList getOwner();
  LongList getWaitOwner();
  boolean hasWait();
  short prepare(in long idTrans);
  short prepareCoop(in long transId);
  short prepareCheck(in long transId);
  short abort(in long idTrans);
  short abortCoop(in long transId);
  short abortCheck(in long transId);
  short commit(in long transId);
  short commitCoop(in long transId);
  short commitCheck(in long transId);
  void update(in ObjectManager obj);
  boolean nonBlockingLock(in long idTrans, in short requestedLock);
  void blockingLock(in long idTrans, in short requestedLock);
  boolean cooperateLock(in long transId, in short coopLock);
  boolean unLock(in long idTrans);
  boolean unlockCoop(in long transId);
};
```

Figura A.5: IDL da classe Transactional Object.

```
interface Repository {
    // Atributos
    readonly attribute string root;

    // Operações
    boolean createArea(in long idTrans, in string dirName);
    boolean removeArea(in long idTrans);
    boolean insertObj(in long idTrans, in ObjectManager obj);
    boolean removeObj(in long idTrans, in ObjectManager obj);
    boolean updateObj(in long transId, in ObjectManager obj);
    boolean saveObj(in long idTrans, in ObjectManager obj);
    boolean restoreBackup(in long transId, in string objName);
    boolean backupObj(in long transId, in ObjectManager obj);
    string getDirectory(in string filename);
    string getDirectoryBy(in long transId);
    boolean hasArea(in long transId);
    string getPath(in long transId);
    string getPathBy(in string filename);
    string getTransName(in long transId);
    boolean deleteFile(in string fileName);
    StringList readFileNames(in string pathName);
};
```

Figura A.6: IDL da classe Repository.

```
interface Recovery {
  // Atributos

  // Operações
  boolean createLog(in long transId);
  boolean removeLog(in long transId);
  boolean insertOp(in Operation op, in long transId);
  boolean prepareCheck(in long transId);
  boolean commitCheck(in long transId);
  boolean abortCheck(in long transId);
  TransManager restore(in string transName);
};
```

Figura A.7: IDL da classe Recovery.

```
interface TransFactory {
  // Atributos
  readonly attribute string transMarker;

  // Operações
  GroupTrans newGT(in string user, in string transName, in boolean vital, in
GroupTrans gt);
  UserTrans newUT(in string user, in string transName, in boolean vital, in
GroupTrans gt);
  void freeTrans(in TransManager trans);
};
```

Figura A.8: IDL auxiliar da factory de transações.

```
interface ObjectFactory {
  // Atributos
  readonly attribute string objMarker;

  // Operações
  ObjectManager newObject(in string marker, in ObjectManager parentObject);
  void freeObject(in ObjectManager obj);
};
```

Figura A.9: IDL auxiliar da factory de objetos.

```
interface Counter: ObjectManager {
  // Atributos
  attribute long parameter;
  attribute long count;

  // Operações
  void increase();
  void decrease();
};

interface CounterFactory: ObjectFactory {

};
```

Figura A.10: IDLs do objeto da aplicação e da factory do objeto da aplicação.