UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

DESENVOLVIMENTO DE UM MECANISMO GERENCIADOR DE TRANSAÇÕES PARA SISTEMAS DISTRIBUÍDOS

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

CHRISTIANO MARCIO BORCHARDT

BLUMENAU, JUNHO/2002

2002/1-16

DESENVOLVIMENTO DE UM MECANISMO GERENCIADOR DE TRANSAÇÕES PARA SISTEMAS DISTRIBUÍDOS

CHRISTIANO MARCIO BORCHARDT

ESTE TRABALHO DE CONCLUSÃO DE CURSO, FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Marcel Hugo — Orientador na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Marcel Hugo

_____

Prof. Paulo Cesar Rodacki Gomes

_____

Prof. Luiz Bianchi

AGRADECIMENTOS

Agradeço primeiramente aos meus pais Tito Borchardt e Aurea Gadotti Borchardt por terem feito o possível e o impossível para me oferecer uma boa educação.

Aos meus amigos Francisco R. Beltrão e Vianei Schmith por terem sempre me dado a maior força nos momentos mais difíceis.

Ao meu chefe e amigo sr. Werner Keske pelas conversas que resultaram no tema e pelos recursos oferecidos que foram indispensáveis para a conclusão deste trabalho.

Ao professor orientador Marcel Hugo, pela dedicação, apoio, competência e pela amizade que ficou consolidada durante a elaboração deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE SIGLAS

2PC	*Two-Phases Commit*
2PL	*Two-Phase locking*
API	*Application Program Interface*
COM	*Component Object Model*
CORBA	*Common Object Request Broker Architecture*
DCE	*Distributed Computing Enviroment*
DCOM	*Distributed Component Object Model*
DLL	*Dynamic Link Library*
IDL	*Interface Definitions Language*
IP	*Internet Protocol*
MIDL	*Microsoft Interface Definitions Language*
OMG	*Object Management Group*
ORB	*Object Request Broker*
SGBD	Sistema Gerenciador de Banco de Dados
TP	T*ransaction Processing*
UUID	*Universally Unique ID*

RESUMO

Este trabalho consiste na pesquisa sobre o funcionamento de sistemas de processamento de transações em ambientes distribuídos e as técnicas para garantir o funcionamento das propriedades ACID de transações. Como resultado obteve-se a implementação de um mecanismo de gerenciamento de transações distribuídas através do protocolo de consolidação de duas fases e do algoritmo de registro de *log* de “refazer”, juntamente com uma classe de gravação de arquivos para serem utilizados por desenvolvedores da ferramenta Visual C++ realizando a comunicação através da especificação de objetos distribuídos DCOM da Microsoft.

ABSTRACT

This work consists on the research about the functionality of transactions processing systems in distributed environments and the techniques to ensure the functionality of the ACID transactions properties. As the result a distributed transaction management system was implemented, using the two-phase commit protocol and "redo" logging algorithm, altogether with a read-write file class that can be used among Visual C++ developers, assisting the comunication through Microsoft's distributed object specification (DCOM).

1 INTRODUÇÃO

Um “sistema distribuído” é qualquer sistema que envolve múltiplas localidades conectadas juntas em uma espécie de rede de comunicações, nas quais o usuário (usuário final ou programador de aplicação) de qualquer localidade pode acessar os dados armazenados em outro local (Date, 1988).

Segundo uma recente publicação de Raynal (2001), sistemas distribuídos são difíceis de modelar e de implementar, por causa da imprevisibilidade e demora de transferência de mensagens, velocidade de execução de processos e tratamento de falhas. Tais variáveis tornam o sistema vulnerável a anormalidades, principalmente quando este estiver executando uma transação envolvendo diversas operações.

Grandes corporações em transportes, finanças, telecomunicações, órgãos governamentais e militares, necessitam que suas aplicações se utilizem de mecanismos de transação para garantir a integridade das tarefas e dados de suas aplicações.

Uma transação de negócios normalmente requer a execução de múltiplas operações. Por exemplo, considera-se a compra de um item em um grande departamento de vendas. Uma operação registra o pagamento do item e outra operação registra a remoção do item do local de estocagem. A transação deve ser feita por completo, ou seja, se uma das operações acima for executada com sucesso, e a outra não, haverá uma grande falha na consistência dos dados, pois a venda do item pode ser registrada e a quantidade em estoque não corresponde a realidade por motivo de alguma falha na baixa.

A finalidade de qualquer banco de dados é executar transações. No entanto, a noção de transação requer definição cuidadosa no ambiente distribuído, porque ela pode envolver a execução de código em múltiplos componentes do sistema (processamento de transações distribuído). Uma transação é definida como sendo uma unidade de recuperação (uma unidade lógica de trabalho) e um agente como sendo a execução do processo por conta de alguma transação em particular, em algum elemento em especial (isto é, um representante daquela transação, naquele componente) (Date, 1991).

Um sistema de processamento de transações é uma coleção de programas de transação designados a fazer as funções necessárias para automatizar uma atividade de negócio.

O primeiro sistema de transação *on-line* foi um sistema de reservas de uma companhia aérea. O sistema foi chamado de SABRE, desenvolvido em meados de 1960, como uma união entre a IBM e a American Airlines. SABRE foi um dos maiores esforços em sistemas computacionais até aquele momento, e certamente é um dos maiores sistemas de processamento de transações no mundo (Bernstein, 1997).

Em um sistema distribuído, uma transação freqüentemente envolve vários elementos como participantes. No final de uma transação, estes participantes precisam de um protocolo para concluir (*commit*) a transação, caso tudo ocorrer normalmente, ou abortar (*abort*) se algo ocorrer errado.

Segundo Bernstein (1997), uma transação possui quatro propriedades que devem ser atendidas (ACID): atomicidade (a transação executa completamente ou não executa nada), consistência (deve preservar a consistência interna da base de dados), isolamento (precisam ser executadas uma de cada vez), e durabilidade (o resultado de uma transação não deve ser perdido por motivo de falha).

Para atender estas propriedades em sistemas distribuídos, devem ser explorados conceitos e mecanismos fundamentais como *time-out, multicast,* problemas de consenso e detecção de falhas.

No gerenciamento de transações em ambientes distribuídos, atualmente é necessário a presença de um banco de dados com estas características avançadas. No entanto, o custo de aquisição e gerenciamento de um banco de dados deste tipo não é acessível para usuários de aplicação de pequeno e médio porte.

O trabalho proposto consiste em estudo dos principais mecanismos de transações existentes e implementação de uma biblioteca de gerenciamento de transações em ambientes distribuídos para uso da ferramenta Visual C++ 6.0. A comunicação entre os componentes distribuídos será feita utilizando-se a tecnologia COM/DCOM da Microsoft, e a gravação dos dados será feita através de sistema de arquivos, sem o uso de banco de dados.

O desenvolvimento será feito utilizando-se a linguagem C++ e a técnica para especificação será a UML (*Unified Modeling Language*).

1.1 MOTIVAÇÃO

A utilização de COM/DCOM para desenvolvimento de sistemas distribuídos tem sido de grande interesse para desenvolvedores de aplicações de pequeno e médio porte, onde a base de dados é mantida através de sistemas de arquivos e sem a utilização de um banco de dados.

Freqüentemente podem ocorrer problemas utilizando-se sistemas distribuídos, por estes estarem em ambientes complexos e passíveis de falhas, que podem fazer com que diversas tarefas de aplicações não sejam executadas por completo e causar inconsistências na base de dados.

Como vários artigos têm sido escritos sobre os problemas de transações nesses ambientes, escolheu-se a área de gerência de transações em sistemas distribuídos para solucionar esse tipo de problema.

1.2 OBJETIVO

O objetivo do trabalho é desenvolver um mecanismo de gerenciamento de transações para uso em sistemas distribuídos, na linguagem C++.

Os objetivos específicos do trabalho são:

a) desenvolvimento de uma biblioteca em C++ de gerenciamento de transações para uso em sistemas distribuídos, utilizando sistema de arquivos;
b) implementação de uma pequena aplicação utilizando a biblioteca desenvolvida para fins de demonstração, cuja integridade dos dados seja mantida, mesmo estes sendo expostos a diversos tipo de situações que podem tornar "anormal" a execução de alguma tarefa.

1.3 ORGANIZAÇÃO DO TEXTO

Este trabalho está organizado da seguinte forma:

O capítulo 1 apresenta, objetivamente, uma introdução ao trabalho, suas motivações, seus objetivos e a organização do texto.

O capítulo 2 apresenta o funcionamento do processamento de transações, suas propriedades, mecanismos de tratamento de falhas e recuperação da base de dados.

O capítulo 3 apresenta como tratar a concorrência em sistemas de transação, e algumas técnicas de bloqueio.

O capítulo 4 apresenta uma definição de sistemas distribuídos, e como funciona a utilização de objetos distribuídos com OMG CORBA e Microsoft DCOM.

O capítulo 5 apresenta como devem ser tratadas as transações em um ambiente distribuído, que é o principal objetivo deste trabalho. Este capítulo está basicamente dividido em algoritmos de bloqueios e protocolos de consolidação distribuídos.

O capítulo 6 apresenta como o protótipo da biblioteca de transações foi desenvolvido e sua especificação através da modelagem UML. Também é apresentado como funciona a aplicação que usa a biblioteca desenvolvida.

O sétimo e último capítulo mostra o resultado do trabalho, suas limitações e possíveis melhoramentos.

2 PROCESSAMENTO DE TRANSAÇÕES

Segundo Date (2000), uma transação, chamada de unidade lógica de trabalho, é fundamental para assegurar a integridade de um banco de dados. A transação usual diária em uma empresa típica, envolve uma seqüência complexa de ações. Nesse ambiente, a habilidade de um produto suportar adequadamente o gerenciamento de transações, é considerado uma função crítica do produto.

Ramakrishnan (2000) também comenta que uma transação é vista como uma série ou lista de ações. As ações que podem ser executadas por uma transação podem ser leitura e gravação de objetos de uma base de dados.

Segundo Bernstein (1997), uma transação *on-line* é a execução de um programa que executa funções administrativas acessando uma base de dados compartilhada.

A maior parte dos programas de transações acessam dados compartilhados, mas nem todos possuem essa característica. Alguns programas executam uma função de comunicação pura, somente enviando uma mensagem de um sistema para outro. Uma aplicação na qual os programas não acessam dados compartilhados não é considerada um processamento de transações verdadeira, porque dessa maneira a aplicação não necessita de todos os recursos e mecanismos especiais que um sistema de processamento de transações pode oferecer.

Date (2000) comenta que um sistema que admite o gerenciamento de transações, deve garantir que as atualizações serão desfeitas caso a transação que executar estas atualizações na base de dados falhar (por qualquer motivo) antes de a transação atingir seu término planejado. Dessa forma, ou a transação será executada integralmente ou será totalmente cancelada (isto é, será como se ela nunca tivesse sido executada). Desse modo, uma sequência de operações que fundamentalmente não é atômica pode parecer atômica de um ponto de vista externo.

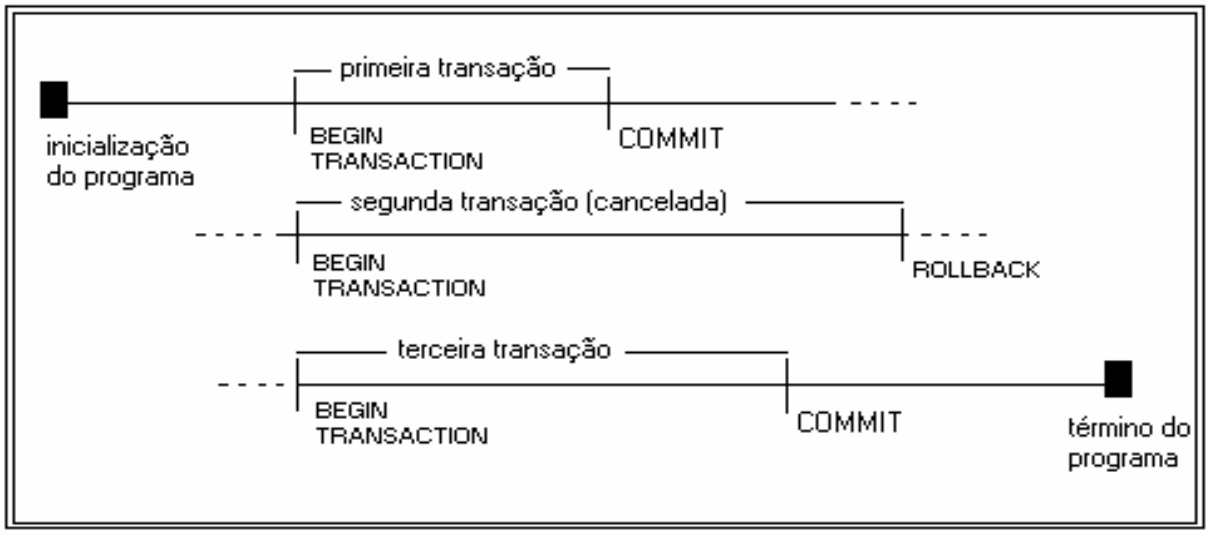
Segundo Date (1988), uma transação consiste na execução de uma seqüência de operações especificadas pela aplicação, começando com uma operação especial BEGIN TRANSACTION, e terminando ou com uma operação COMMIT ou com uma operação ROLLBACK.

O componente do sistema que fornece essa atomicidade é chamado de **gerenciador de transações,** também conhecido como **monitor de processamento de transações** ou **monitor TP (*transaction processing*)**, e as operações COMMIT E ROLLBACK são a chave para se entender o modo como ele funciona:

a) A operação COMMIT indica o término de uma transação bem sucedida. Ela informa ao gerenciador de transações que uma unidade lógica de trabalho foi concluída com sucesso, a base de dados está (ou deveria estar) novamente em um estado consistente e todas as atualizações feitas por essa unidade de trabalho podem agora ser “validadas” ou tornadas permanentes;
b) Em contraste, a operação ROLLBACK assinala o término de uma transação mal sucedida. Ela informa ao gerenciador de transações que algo saiu errado, que a base de dados pode estar em um estado inconsistente, e que todas as atualizações feitas pela unidade lógica de trabalho até agora devem ser “retomadas” ou desfeitas.

Deve-se observar que COMMIT e ROLLBACK terminam a transação, não o programa. Em geral, a execução de um único programa consistirá em uma seqüência de várias transações executadas uma após a outra, como ilustra a figura 1:

FIGURA 1: A EXECUÇÃO DE UMA SEQUÊNCIA DE TRANSAÇÕES



Fonte: adaptado de Date (2000)

Segundo Garcia-Molina (2001), assegurar que as transações serão executadas corretamente é o trabalho de um gerenciador de transação, um subsistema que executa várias funções, inclusive:

a) Emitir sinais para o gerenciador de *log*, de modo que informações necessárias sob a forma de "registros de log" possam ser armazenadas no *log* (registros de *log* serão vistos na seção 2.5);
b) Assegurar que transações em execução concorrente não irão interferir umas com as outras de forma a introduzir erros ("escalonamento" ou "programação de execução").

2.1 PROPRIEDADES DE UMA TRANSAÇÃO

Ramakrishnan (2000) comenta a existência de quatro propriedades das transações que precisam ser asseguradas para manter os dados consistentes frente a concorrência de acesso e falhas de sistema (ACID):

a) A aplicação precisa garantir que a execução de cada transação seja feita de forma atômica (**atomicidade)**, ou seja, todas as ações são feitas por completo ou nenhuma ação será feita. O usuário – desenvolvedor de aplicação - não deverá se preocupar com o efeito de transações incompletas (caso alguma falha de sistema ocorra);
b) Cada transação precisa ser executada sem a execução concorrente de outra transação, preservando a consistência da base de dados. Esta propriedade é chamada **consistência**;
c) Desenvolvedores de uma aplicação precisam conhecer uma transação sem considerar o efeito de outras transações executando concorrentemente. Esta propriedade é chamada de **isolamento**;
d) Uma vez informado que a transação foi bem sucedida, este efeito deverá persistir mesmo se o sistema falhar antes que todas as modificações na base sejam refletidas em disco. Esta propriedade é chamada **durabilidade**.

2.2 FALHAS

Segundo Bernstein (1997), é tarefa de um gerenciador de recursos retornar a base de dados a um estado que inclui todas as transações consolidadas antes de uma falha e exclui todas as transações que abortaram antes ou durante a falha.

Existem poucos casos em que um programador de aplicação ou gerenciador do sistema podem afetar a performance de recuperação de uma transação. No entanto, existem coisas que um gerenciador de sistema pode fazer para afetar a proporção de tolerância a falhas de um sistema, como alterar configurações de *logs* e dispositivos de disco. Discutir sobre performance e implicações de tolerância a falhas de aplicações e modelagem de sistemas, é de uma grande importância para conhecer a base dos principais conceitos de algoritmos de recuperação de base de dados.

2.3 TIPOS DE FALHAS

Segundo Bernstein (1997), a maioria das falhas são devidas a transações programadas incorretamente e erros de entrada de dados que geram parâmetros incorretos para transações. Infelizmente, estas falhas degradam a suposição que a execução de uma transação preserva a consistência da base de dados (“C” em ACID).

Existem dois tipos de falhas que são de maior importância em um sistema TP (*transaction processing*) centralizado: falhas de sistema e falha de mídia.

2.3.1 FALHAS DE SISTEMA

Date (1988) utiliza o termo “falha de sistema” para designar qualquer evento que obrigue o sistema a parar, exigindo um subsequente reinicio.

Garcia-Molina (2001) comenta que as falhas de sistema são problemas que fazem o estado de uma transação ser perdido. As falhas típicas do sistema são quedas de energia e erros de software. Para ver porque problemas como quedas de energia geram perda de estado, observe que, como qualquer programa, as etapas de uma transação ocorrem normalmente na memória principal. Diferente do disco, a memória principal é “volátil”. Isto é, uma falha de energia fará o conteúdo da memória principal desaparecer, enquanto os dados de um disco (não voláteis) permanecem intactos. De modo semelhante, um erro de software pode

sobrescrever parte da memória principal, possivelmente incluindo valores que faziam parte do estado do programa.

Quando a memória principal é perdida, o estado da transação é perdido; isto é, não se pode saber quais partes da transação, inclusive suas modificações na base de dados, foram feitas. Executar a transação novamente pode não corrigir o problema. Por exemplo, se a transação tiver de adicionar 1 a um valor na base de dados, não se saberá se é preciso ou não repetir a adição de 1.

2.3.2 FALHAS DE MÍDIA

Segundo Bernstein (1997), uma falha de mídia ocorre quando qualquer parte do armazenamento estável é destruído. Por exemplo, isso acontece se algum setor de um disco se danificou.

O tratamento deste tipo de falha não será abordado neste trabalho, pois o foco do mesmo se resume apenas em falhas de sistema.

2.4 RECUPERAÇÃO DE FALHAS

Date (2000) afirma que as transações não são apenas uma unidade de trabalho, mas também uma unidade de **recuperação**.

Segundo Bernstein (1997), o processo de transação garante atomicidade através de mecanismos na base de dados, que guardam as operações realizadas. Se a transação falhar por algum motivo antes de completar sua tarefa, o sistema de transações deverá desfazer todos os efeitos de qualquer modificação sobre a base de dados feitas pela transação. Somente se todas as tarefas foram realizadas com sucesso, o sistema de transações permite que as alterações tornem-se partes permanentes da base de dados.

Utilizando-se da propriedade atomicidade, pode-se escrever um programa que emula uma transação de negócios atômica, como uma movimentação de uma conta bancária, reservas aéreas ou controle de estoques. Cada ação de negócios requer múltiplas atualizações de itens de dados. Implementando uma ação de negócios através de uma transação, garante-se que todas as atualizações serão executadas ou nenhuma. A atomicidade garante que a base de

dados seja retornada para um estado conhecido depois de uma falha, reduzindo a intervenção manual durante a reinicialização do sistema (Bernstein, 1987).

Segundo Date (2000), o ponto crítico no que se refere a falhas do sistema é o fato de que o conteúdo da memória principal é perdido (em particular, os *buffers* do banco de dados se perdem). Então, o estado exato de qualquer transação em curso no momento da falha deixa de ser conhecido; desse modo, tal transação não poderá nunca mais ser concluída com sucesso e deverá ser desfeita – isto é, retomada – quando o sistema for reinicializado.

Além disso, também pode ser necessário refazer no momento de reinicialização certas transações concluídas com êxito antes da queda, mas que não conseguiram ter suas atualizações transferidas dos *buffers* do banco de dados para o banco de dados físico.

Surge aqui a questão óbvia: de que maneira o sistema saberá no momento de reinicialização quais transações devem ser desfeitas e quais devem ser refeitas? Segundo Date (2000), a resposta é a seguinte: em certos intervalos predeterminados – em geral, sempre que algum número preestabelecido de entradas é gravado no log – o sistema automaticamente **marca um *checkpoint***. Marcar um *checkpoint* envolve (a) gravar fisicamente ("gravação forçada") o conteúdo dos *buffers* do banco de dados no banco de dados físico e (b) gravar fisicamente um **registro de *checkpoint*** especial no *log* físico. O registro do *checkpoint* fornece uma lista de todas as transações que estavam em andamento no momento em que o *checkpoint* foi marcado.

2.5 REGISTROS DE LOG

Segundo Garcia-Molina (2001), um *log* é uma sequência de registros de *log*, cada um informando algo sobre o que alguma transação fez.

As ações de várias transações podem se "intercalar", de forma que uma etapa de uma transação possa ser executada e seu efeito anotado, e depois o mesmo ocorra para uma etapa de outra transação, em seguida para uma segunda etapa da primeira transação ou uma etapa de uma terceira transação e assim por diante. Essa intercalação de transações complica o registro de *log*; não é suficiente apenas registrar no *log* o histórico inteiro de uma transação depois que essa transação se completa. É crucial que o log reflita exatamente a ordem na qual as operações conflitantes realmente executaram. Isto é, se uma atualização precede e conflita

com outra atualização no *log*, então a atualização deve realmente ser executada nessa ordem. O motivo é que depois de uma falha, o sistema de recuperação repetirá parte da tarefa realizada antes da falha, e ele poderá assumir que a ordem das operações no *log* é a ordem na qual ele repetiria a tarefa. Nota-se que não é necessário que o *log* reflita exatamente a ordem de todas as atualizações, mas somente as conflitantes, as quais a ordem relativa faz diferença.

Se houver uma pane do sistema, o *log* será consultado para se reconstituir o que as transações estavam fazendo quando a pane aconteceu. O *log* também pode ser usado em conjunto com um arquivo de armazenamento, no caso de haver uma falha de mídia de um disco que não armazene o *log*. Em geral, para reparar o efeito da pane, algumas transações terão seu trabalho refeito, e os novos valores que elas gravaram na base de dados serão gravados novamente. Outras transações terão seu trabalho desfeito, e a base de dados será restaurada, de forma que parecerá que elas nunca foram executadas.

Para estudar os detalhes dos algoritmos de registro de log e outros algoritmos de gerenciamento de transações, precisa-se de uma notação que descreva todas as operações que movem dados entre espaços de endereços. As primitivas usadas por Garcia-Molina (2001) e que serão adotadas são:

1. INPUT(X): Copie o bloco de disco contendo o elemento do banco de dados X para um *buffer* da memória;
2. READ(X, t): Copie o elemento X da base de dados para a variável local *t* da transação. Mais precisamente, se o bloco que contém o elemento da base de dados X não estiver em um *buffer* da memória, execute primeiro INPUT(X). Em seguida, atribua o valor X à variável local *t*;
3. WRITE(X, *t*): Copie o valor da variável local *t* para o elemento da base de dados X em um *buffer* da memória. Mais precisamente, se o bloco que contém o elemento de banco de dados X não estiver em um *buffer* da memória, então execute INPUT(X). Em seguida, copie o valor de *t* para X no *buffer*;
4. OUTPUT(X): Copie o *buffer* que contém X para o disco.

Segundo Date (1988), as operações anteriores fazem sentido, desde que os elementos da base de dados residam dentro de um único bloco de disco, e portanto dentro de um único *buffer*. Esse seria o caso para elementos da base de dados que fossem blocos. Se os elementos

da base de dados ocuparem vários blocos, então deve-se imaginar que cada porção do elemento com o tamanho de um bloco será um elemento em si. O mecanismo de registro de *log* a ser usado irá assegurar que a transação não irá se completar sem que a gravação de X seja atômica; isto é, ou todos os blocos de X serão gravados em disco, ou nenhum. Desse modo, pode-se supor para toda a discussão sobre registro de *log* que um elemento da base de dados não é maior que um único bloco.

Segundo Garcia-Molina (2001), é importante observar que os componentes que emitem esses comandos são diferentes. READ e WRITE são emitidos por transações. INPUT e OUTPUT são emitidos pelo gerenciador de *buffers*, embora OUTPUT também possa ser iniciado pelo gerenciador de *log*, sob certas condições.

Garcia-Molina (2001) e Ramakrishnan (2000), comentam três estilos diferentes de algoritmos de registro de *log*, chamados de “desfazer”, “refazer” e “desfazer/refazer”.

Segundo Garcia-Molina (2001), existem várias formas de registro de log utilizadas com cada um destes tipos de registro de *log*:

1. <START T>: Esse registro indica que a transação T começou;
2. <COMMIT T>: A transação T foi concluída com sucesso e não fará mais nenhuma mudança nos elementos do banco de dados. Quaisquer mudanças no banco de dados feitas por T devem aparecer em disco. Porém, como não se pode controlar quando o gerenciador de buffers optará por copiar blocos de memória para o disco, em geral não se pode ter certeza de que as mudanças já estão no disco quando se vê o registro de *log* <COMMIT T>. Se insistir-se no fato de que as mudanças já estão em disco, esse requisito deverá ser imposto pelo gerenciador de *log* (como no caso do registro de *log* de desfazer);
3. <ABORT t>: A transação T não conseguiu se completar com sucesso. Se a transação T abortar, nenhuma mudança que ela fez poderá ser copiada em disco, e esse é o trabalho do gerenciador de transações para ter certeza de que mais mudanças nunca aparecerão no disco, ou que seu efeito sobre o disco será cancelado se isso ocorrer.

Segundo Garcia-Molina (2001), no caso de um log de desfazer, o único tipo restante de registro de *log* de que se precisa é um registro de atualização, que é uma tripla <T,X,*v*>. O

significado desse registro é: a transação T mudou o elemento do banco de dados X e seu antigo valor era *v*. A mudança refletida por um registro de atualização ocorre normalmente na memória, e não em disco; isto é, o registro de *log* é uma resposta a uma ação WRITE, não a uma opção OUTPUT. Observa-se também que um log de desfazer não registra o novo valor de um elemento do banco de dados, apenas o valor antigo. Caso a recuperação seja necessária em um sistema que use o registro de *log* de desfazer, a única ação do gerenciador de recuperação será cancelar o possível efeito de uma transação em disco, restaurando o valor antigo.

2.5.1 REGISTRO DE LOG DE DESFAZER

Segundo Bernstein (1997), se o algoritmo não estiver absolutamente certo de que os efeitos de uma transação foram completados e armazenados em disco, qualquer alteração no banco de dados que a transação possa ter feito será desfeita, e o estado do banco de dados será restaurado ao que existia antes da transação.

Garcia-Molina (2001) comenta que há duas regras às quais as transações devem obedecer para que o *log* de desfazer permita a recuperação de uma falha do sistema. Essas regras afetam o que o gerenciador de *buffers* pode fazer e também exigem que certas ações sejam executadas sempre que uma transação for consolidada. Estas serão resumidas aqui:

U_1: Se a transação T modificar o elemento X do banco de dados, então o registro de *log* da forma <T, X, *v*> deve ser gravado em disco antes do novo valor de X ser gravado em disco;

U_2: Se uma transação é consolidada, seu registro de log COMMIT deve ser gravado em disco somente depois que todos os elementos do banco de dados alterados pela transação forem gravados em disco, mas tão logo quanto possível depois disso.

Para resumir as regras U_1 e U_2, o material associado com uma transação deve ser gravado em disco na seguinte ordem:

a) Os registros de *log* que indicam os elementos do banco de dados alterados;
b) Os próprios elementos do banco de dados alterados;
c) O registro de log COMMIT.

Segundo Garcia-Molina (2001), para forçar os registros de *log* para o disco, o gerenciador de *log* precisa de um comando esvaziar *log* que informe ao gerenciador de *buffers* para copiar para o disco quaisquer blocos de *log* que não tenham sido copiados antes em disco, ou que tenham sido alterados desde que foram copiados pela última vez. Nas seqüências de ações, será mostrado FLUSH LOG explicitamente. O gerenciador de transações também precisa ter um meio para informar o gerenciador de *buffers* para executar uma ação OUTPUT sobre um elemento de banco de dados.

Garcia-Molina (2001) exemplifica as regras do registro de *log* de desfazer através do quadro 1. O cabeçalho M-A indica “a cópia de A em um *buffer* de memória”, e D-B indica “a cópia de B em disco” e assim por diante.

Na linha (1) do quadro 1, a transação T começa. A primeira ação que ocorre é que o registro <START T> é gravado no *log*. A linha (2) representa a leitura de A por T. A linha (3) é a mudança local de *t*, que não afeta nem a base de dados armazenada em disco nem qualquer porção do banco de dados em um *buffer* da memória. Nem a linha (2) nem a (3) exigem qualquer entrada de *log*, pois elas não tem efeito sobre o banco de dados.

A linha (4) é a gravação do novo valor de A para o *buffer*. Essa modificação para A é refletida pela entrada de *log* <T, A, 8> que informa que A foi alterado por T e seu antigo valor era 8. Nota-se que o novo valor, 16, não é mencionado em um *log* de desfazer.

As linhas (5) a (7) executam as mesmas três etapas com B em lugar de A. Nesse momento, T se completou e deve ser consolidada. Seria preferível que os valores A e B alterados migrassem para o disco mas, a fim de seguir as duas regras do registro de *log* de desfazer, há uma seqüência fixa de eventos que devem ocorrer.

QUADRO 1: AÇÕES E SUAS ENTRADAS DE *LOG* USANDO O REGISTRO DE *LOG* DE DESFAZER

Etapa	Ação	*t*	M-A	M-B	D-A	D-B	Log
1							<START T>
2)	READ(A,t)	8	8		8	8	
3)	T := t * 2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T, A, 8>
5)	READ(B,t)	8	16	8	8	8	
6)	T := t * 2;	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T, B ,8>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)	OUTPUT(B)	16	16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

Fonte: Garcia-Molina (2001)

Primeiro, A e B não podem ser copiados para o disco enquanto os registros de *log* relativos às mudanças não estiverem em disco. Desse modo, na etapa (8) o *log* é esvaziado, assegurando que esses registros aparecerão no disco. O gerenciador de transações solicita essas etapas ao gerenciador de *buffers*, a fim de consolidar T.

Agora, é possível consolidar T, e o registro <COMMIT T> é gravado no *log*, que é a etapa (11). Por fim, devemos esvaziar o *log* novamente na etapa (12) para ter certeza de que o registro <COMMIT T> do *log* aparecerá em disco.

2.5.2 RECUPERAÇÃO USANDO O REGISTRO DE LOG DE DESFAZER

Segundo Garcia-Molina (2001), a primeira tarefa do gerenciador de recuperação é dividir as transações em transações consolidadas e não consolidadas. Se houver um registro de *log* <COMMIT> então, pela regra de desfazer U_2, todas as mudanças feitas pela transação T foram gravadas anteriormente em disco. Desse modo, T não poderia ter deixado o banco de dados em um estado inconsistente quando ocorreu a falha do sistema.

Entretanto, se supostamente for encontrado um registro <START T> no *log*, mas nenhum registro <COMMIT T>. Então, poderia ter havido algumas mudanças no banco de dados feitas por T ou não foram feitas, nem mesmo nos *buffers* da memória principal, ou foram feitas nos *buffers* mas não copiadas em disco. Nesse caso, T é uma transação incompleta e deve ser desfeita. Isto é, qualquer mudança realizada por T deve ser restaurada a seu valor anterior. Felizmente, a regra U1 nos assegura que, se T alterou X em disco antes da pane, então haverá um registro <T, X, *v*> no *log*, e esse registro terá sido copiado em disco antes da pane. Desse modo, durante a recuperação, deve-se gravar o valor *v* para um elemento do banco de dados X. Observe que essa regra admite como verdade que X tinha o valor *v* no banco de dados; nem mesmo deve-se preocupar em verificar esse fato.

Segundo Bernstein (1997) e Garcia-Molina (2001) , tendo em vista que pode haver várias transações não consolidadas que modificaram X, deve-se ser sistemático quanto a ordem que se restaura os valores. Desse modo, o gerenciador de recuperação deve varrer o *log* a partir do final (isto é, a partir do registro gravado mais recentemente até o registro mais antigo gravado). À medida que percorrer o *log*, ele se lembrará de todas as transações T para as quais viu um registro <COMMIT T> ou um registro <ABORT T>. Também à medida que percorrer o *log* de trás para frente, ele verá um registro <T, X, *v*>, e então:

a) Se T for uma transação cujo registro COMMIT foi visto, não fará nada. T foi consolidada e não precisa ser desfeita;
b) Caso contrário, T será uma transação incompleta ou uma transação abortada. O gerenciador de recuperação deverá mudar o valor de X no banco de dados para *v*.

Depois de fazer essas mudanças, o gerenciador de recuperação deve gravar um registro de *log* <ABORT T> para cada transação incompleta T que não tenha sido abortada antes, e depois esvaziar o *log*. Agora, a operação normal do banco de dados pode prosseguir, e novas transações podem começar a ser executadas.

2.5.3 REGISTRO DE LOG DE REFAZER

Segundo Bernstein (1997), o registro de *log* de desfazer tem um problema potencial de não poder consolidar uma transação sem primeiro gravar todos os seus dados alterados em disco. Às vezes, se pode poupar operações de Entrada/Saída de disco, se permitir que as mudanças no banco de dados residam apenas na memória principal durante algum tempo; desde que exista um *log* para fazer correções no caso de uma pane, é seguro fazê-lo.

Segundo Garcia-Molina (2001), o registro de *log* de refazer representa mudanças em elementos do banco de dados por um registro de *log* que fornece o novo valor, em lugar do valor antigo, que o registro de *log* de desfazer utiliza. Esses registros parecem idênticos ao registro de *log* de desfazer: <T, X, *v*>. A diferença é que o significado desse registro é "uma transação T gravou um novo valor *v* para o elemento do banco de dados X". Não há nenhuma indicação do antigo valor de X nesse registro. Toda vez que uma transação T modifica um elemento do banco de dados X, um registro de forma <T, X, *v*> deve ser gravado no *log*.

Além disso, a ordem em que as entradas de dados e de *log* alcançam o disco pode ser descrita por uma única "regra de refazer", chamada de **regra de registro de *log* de gravação antecipada**.

> R_1: Antes de modificar qualquer elemento do banco de dados X em disco, é necessário que todos os registros de *log* que pertencem a essa modificação de X, inclusive tanto o registro de atualização <T, X, *v*> quanto o registro <COMMIT T>, devem aparecer em disco.

Tendo em vista que o registro COMMIT para uma transação só pode ser gravado no *log* quando uma transação se completa, e portanto o registro de consolidação deve seguir todos os registros de *log* de atualização, pode-se resumir o efeito da regra R1 afirmando que, quando o registro de *log* de refazer está em uso, a ordem na qual o material associado a uma transação gravado em disco é:

a) Os registros de *log* que indicam elementos do banco de dados alterados;
b) O registro de *log* COMMIT;
c) Os próprios elementos do banco de dados alterados.

Garcia-Molina (2001) exemplifica as regras do registro de *log* de refazer através do quadro 2. As principais diferenças deste quadro com o quadro 1 são dadas a seguir: Primeiro, deve-se observar as linhas (4) e (7), onde os registros de *log* que refletem as mudanças têm os novos valores de A e B, em vez de terem os valores antigos. Em segundo lugar, é visto que o registro <COMMIT T> surge antes, na etapa (8). Então, o *log* é esvaziado, e assim todos os registros de *log* que envolvem as mudanças da transação T aparecem em disco. Somente então os novos valores de A e B podem ser gravados em disco. É mostrado esses valores de imediato, nas etapas (10) e (11), embora na prática eles possam ocorrer muito mais tarde.

QUADRO 2: AS AÇÕES E SUAS ENTRADAS DE *LOG*, USANDO O REGISTRO DE *LOG* DE REFAZER

Etapa	Ação	*t*	M-A	M-B	D-A	D-B	Log
1							<START T>
2)	READ(A,t)	8	8		8	8	
3)	T := t * 2		16	8	8	8	
4)	WRITE(A,t)	16	16		8	8	<T, A, 16>
5)	READ(B,t)	8	16	8	8	8	
6)	T := t * 2;	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T, B ,16>
8)							<COMMIT T>
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

Fonte: Garcia-Molina (2001)

2.5.4 RECUPERAÇÃO USANDO O REGISTRO DE LOG DE REFAZER

Segundo Garcia-Molina (2001), uma conseqüência importante da regra R1, de refazer é que, a menos que o *log* tenha um registro <COMMIT T>, sabe-se que nenhuma mudança no banco de dados feita pela transação T foi gravada em disco. Desse modo, transações incompletas podem ser tratadas durante a recuperação como se nunca tivessem ocorrido. Porém, as transações consolidadas representam um problema, pois não se sabe quais de suas mudanças do banco de dados foram gravadas em disco. No entanto, o *log* de refazer tem

exatamente as informações de que se precisa: os novos valores, que se pode gravar em disco independente do fato de já estarem lá. Para se recuperar, usando um *log* de refazer, após uma pane do sistema, é feito o seguinte:

a) Identifica-se as transações consolidadas;
b) Deverá ser varrido o *log* desde o início. Para cada registro de *log* <T, X, *v*> encontrado:
   - Se T não for uma transação consolidada, não faça nada;
   - Se T for consolidada, grave o valor *v* para o elemento do banco de dados X.
c) Para cada transação incompleta T, grave um registro <ABORT T> para o *log* e esvazie o *log*.

2.5.5 REGISTRO DE LOG DE DESFAZER/REFAZER

Segundo Garcia-Molina (2001), um *log* de desfazer/refazer tem os mesmos tipos de registros de *log* que as outras espécies de *log*, com uma exceção. O registro de *log* de atualização que é gravado quando um elemento do banco de dados muda de valor tem quatro componentes. O registro <T, X, *v, w*> significa que a transação T mudou o valor do elemento do banco de dados X; seu valor anterior era *v*, e seu novo valor é *w*. As restrições que um registro de *log* de desfazer/refazer deve seguir são resumidas pela seguinte regra:

UR_1 : Antes de modificar qualquer elemento do banco de dados X em disco devido às mudanças efetuadas por alguma transação T, é necessário que o registro de atualização <T, X, *v, w*> apareça em disco.

Segundo Garcia-Molina (2001), a regra UR_1 para o registro de desfazer/refazer impõe apenas as restrições impostas tanto pelo registro de *log* de desfazer quanto pelo registro de *log* de refazer. Em particular, o registro de *log* <COMMIT T> pode preceder ou seguir qualquer das mudanças nos elementos do banco de dados em disco.

Garcia-Molina (2001) exemplifica as regras do registro de *log* de refazer através do quadro 3, que é uma variação na ordem das ações associadas com a transação T que foi visto pela última vez no exemplo do quadro 2. Deve ser observado que os registros de *log* para atualizações têm agora tanto o valor antigo quanto o novo valor de A e B. Nessa sequência, grava-se o registro de *log* <COMMIT T> em meio à saída dos elementos do banco de dados

A e B em disco. A etapa (10) também poderia aparecer antes da etapa (9) ou após a etapa (11).

QUADRO 3: EXEMPLO DE *LOG* DE DESFAZER/REFAZER

Etapa	Ação	*t*	M-A	M-B	D-A	D-B	Log
1							<START T>
2)	READ(A,t)	8	8		8	8	
3)	T := t * 2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T, A, 8, 16>
5)	READ(B,t)	8	16	8	8	8	
6)	T := t * 2;	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T, B , 8, 16>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)							<COMMIT T>
11)	OUTPUT(B)	16	16	16	16	16	

Fonte: Garcia-Molina (2001)

2.5.6 RECUPERAÇÃO COM O REGISTRO DE LOG DE DESFAZER/REFAZER

Segundo Bernstein (1997), a recuperação com um registro de *log* de desfazer/refazer segue as seguintes regras:

a) Refazer todas as transações consolidadas na ordem da mais antiga para a mais recente;
b) Desfazer todas as transações incompletas na ordem da mais recente para a mais antiga.

Note que é necessário executar ambas as ações. Devido a flexibilidade permitida pelo registro de *log* de desfazer/refazer com respeito à ordem relativa em que os registros de *log* de COMMIT e as próprias mudanças no banco de dados são copiados em disco, poder-se-ia ter uma transação consolidada com algumas ou todas as suas mudanças não gravadas em disco, ou uma transação não consolidada com algumas ou todas as suas mudanças em disco.

3 CONCORRÊNCIA

Segundo Date (2000), os tópicos de concorrência e recuperação caminham juntos, ambos fazendo parte do tópico mais geral de gerenciamento de transações. Agora, deverá ser voltada a atenção especificamente para a concorrência. O termo **concorrência** se refere ao fato de que os SGBDs em geral permitem que muitas transações tenham acesso ao mesmo banco de dados ao mesmo tempo – e em um sistema desse tipo, é necessário algum tipo de **mecanismo de controle de concorrência** para assegurar que transações concorrentes não interfiram umas nas outras.

3.1 EXECUÇÕES SERIALIZÁVEIS

Segundo Bernstein (1987), uma maneira de evitar problemas de interferência é não permitir que as transações sejam intercaladas entre si. Uma execução na qual duas transações não são intercaladas entre si é chamada execução serial. Mais precisamente, uma transação é serial se, para cada par de transações, todas as operações de uma transação executem antes do que qualquer operação de outra transação. Do ponto de vista do usuário, uma execução serial é vista como se fosse processada atomicamente. Execuções seriais são corretas porque cada transação individualmente está correta, e execuções que executam serialmente não podem interferir em outra transação.

Segundo Date (2000), dado um conjunto de transações, qualquer execução dessas transações, intercaladas ou não, é chamada **escalonamento**. Um escalonamento serial é a execução das transações uma de cada vez sem intercalação. Um escalonamento não-serial é um escalonamento **intercalado** (ou simplesmente um escalonamento não-serial). Dois escalonamentos são ditos **equivalentes** se com certeza produzem o mesmo resultado, qualquer que seja o estado inicial do banco de dados. Assim, um escalonamento é correto (isto é, serializável) se é equivalente a algum escalonamento serial.

Garcia-Molina (2001) mostra um exemplo de escalonamento de transações que é serializável mas não é serial, através do quadro 4. Nesse escalonamento, T_2 atua sobre A depois de T_1 atuar, mas antes de T_1 atuar sobre B. Todavia, é visto que o efeito das duas transações escalonadas dessa maneira é igual ao de um escalonamento serial.

QUADRO 4: UM ESCALONAMENTO SERIALIZÁVEL, MAS NÃO SERIAL

T_1	T_2	A	B
READ(A,t)		25	25
t := t + 100			
WRITE(A,t)		125	
	READ(A,s)		
	s := s*2		
	WRITE(A,s)	250	
READ(B,t)			
t := t + 100			
WRITE(B,t)			125
	READ(B,s)		
	s := s*2		
	WRITE(B,s)		250

Fonte: adaptado de Garcia-Molina (2001)

3.2 SERIALIZABILIDADE DE CONFLITO

Garcia-Molina (2001) desenvolve uma condição suficiente para garantir que um escalonamento seja serializável. Os escalonadores de sistemas comerciais em geral asseguram essa condição mais forte, que é chamado “serializabilidade de conflito”, quando querem garantir que as transações irão se comportar de uma maneira serializável. Ela se baseia na idéia de um conflito: um par de ações consecutivas em um escalonamento tais que, se sua ordem for trocada, o comportamento de pelo menos uma das transações envolvidas poderá mudar.

3.3 SERIALIZABILIDADE ATRAVÉS DE BLOQUEIOS

Segundo Garcia-Molina (2001), é tarefa do escalonador evitar qualquer ordem de ações que leve a um escalonamento não serializável. Pode ser considerada a arquitetura mais comum para um escalonador, aquela na qual são mantidos “bloqueios” sobre elementos do banco de dados, a fim de evitar um comportamento não serializável. Intuitivamente, uma transação obtém bloqueios sobre os elementos do banco de dados a que ela tem acesso para impedir que outras transações acessem esses elementos aproximadamente no mesmo tempo e, portanto, haja o risco de não serializabilidade.

Date (2000) explica que há três maneiras específicas pelas quais a serializabilidade poderia ser violada caso a transação não manter algum tipo de bloqueio sobre o elemento do

banco de dados: **leitura suja** (*dirty read*), **leitura não repetível** (*nonrepetable read*) e "**fantasmas"**:

a) Leitura suja: suponha que a transação T_1 execute uma atualização em algum registro. Então, a transação T_2 lê esse registro e a transação T_1 termina em seguida, com um ROLLBACK. A transação T_2 leu assim um registro que não existe mais, e que em certo sentido nunca existiu (porque a transação T_1 na verdade nunca foi executada);
b) Leitura não repetível: suponha que a transação T_1 leia um registro, depois a transação T_2 atualize esse registro, e a transação T_1 leia uma vez mais o "mesmo" registro. Agora, a transação T_1 leu duas vezes o "mesmo" registro, mas viu dois valores diferentes para ele;
c) Fantasmas: suponha que a transação T_1 leia um conjunto de todos os registros que satisfazem a alguma condição (por exemplo, todos os registros de fornecedores que satisfazem à condição de que a cidade do fornecedor seja Blumenau). Suponha ainda que a transação T_2 insira em seguida um novo registro que satisfaz a mesma condição. Se T_1 repetir suas solicitações de leitura, verá um registro que não existia antes – um registro **"fantasma**".

3.4 BLOQUEIOS

Segundo Date (2000), a idéia básica de bloqueio é simples: quando uma transação necessita de uma garantia de que um objeto no qual está interessada – em geral, um elemento do banco de dados – não mudará de algum modo enquanto ela estiver ativa, a transação adquire um bloqueio sobre esse objeto. O efeito do bloqueio é "impedir que outras transações atuem" sobre o objeto em questão e portanto, em particular, impedir que elas alterem o objeto. Desse modo, a primeira transação é capaz de executar seu processamento tendo a certeza de que o objeto em questão permanecerá em um estado estável durante o tempo que essa transação desejar.

Date (2000) explica mais detalhadamente como funciona o bloqueio:

1. Primeiro, suponha que o sistema admita duas espécies de bloqueios, os **bloqueios exclusivos** (bloqueios X) e os **bloqueios compartilhados** (bloqueios C), definidos da maneira indicada nos dois parágrafos seguintes;

2. Se a transação A mantiver um bloqueio exclusivo (X) sobre o elemento do banco de dados *t*, então uma solicitação feita por uma transação distinta B de um bloqueio de qualquer tipo sobre *t* será negada;
3. Se a transação A mantiver um bloqueio compartilhado (C) sobre o elemento do banco de dados *t*, então:
    - Uma solicitação de uma transação distinta B de um bloqueio X sobre *t* será negada;
    - Uma solicitação de alguma transação distinta B de um bloqueio C sobre *t* será concedida (ou seja, agora B também manterá um bloqueio C sobre *t*).

Garcia-Molina (2001), estende a notação para ações, a fim de incluir ações de leitura, gravação, bloqueio e desbloqueio:

R_i (X): A transação T_i lê o elemento do banco de dados X;
w_i (X): A transação T_i grava o elemento do banco de dados X;
l_i (X): A transação T_i solicita um bloqueio sobre o elemento do banco de dados X;
u_i(X): A transação T_i libera seu bloqueio ("desbloqueia") o elemento do banco de dados X;

Desse modo, a condição de consistência para transações pode ser declarada como: "Sempre que uma transação T_i tiver uma ação r_i (X) ou w_i (X), haverá uma ação anterior l_i (X) sem qualquer ação interveniente u_i (X) e haverá uma u_i (X) subseqüente". A validade dos escalonamentos é declarada: "Se houver ações l_i(X) seguidas por l_j (X) em um escalonamento, então em algum lugar entre essas ações tem de haver uma ação u_i (X)".

3.4.1 BLOQUEIO DE DUAS FASES

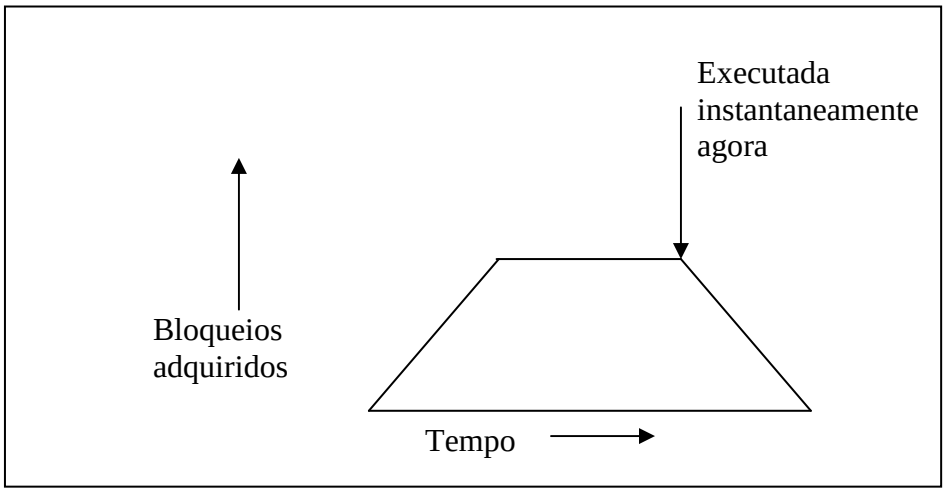
Segundo Garcia-Molina (2001), existe uma condição, sob a qual se pode garantir que um escalonamento válido de transações consistentes é serializável de conflito. Essa condição, que é amplamente seguida nos sistemas de bloqueio comerciais, é chamada **bloqueio de duas fases** ou **2PL** (*two-phase locking*). A condição 2PL é:

- Em toda transação, todas as solicitações de bloqueio precedem todas as solicitações de desbloqueio.

A expressão “duas fases” indicada por 2PL é, portanto, a primeira fase na qual os bloqueios são obtidos, e a segunda fase, em que os bloqueios são liberados. O bloqueio de duas fases é uma condição, como a consistência, sobre a ordem das ações sobre uma transação. Uma transação que obedece à condição 2PL é chamada uma **transação bloqueada em duas fases**, ou **transação 2PL**.

Garcia-Molina (2001) comenta que intuitivamente, pode-se imaginar que cada transação bloqueada em duas fases é executada em sua totalidade no momento em que emite sua primeira solicitação de desbloqueio, como sugere a figura 2. O escalonamento serial a conflito para um escalonamento S de transações 2PL é aquele no qual as transações são desbloqueadas na mesma ordem de seus bloqueios iniciais.

FIGURA 2: TRANSAÇÃO BLOQUEADA EM DUAS FASES

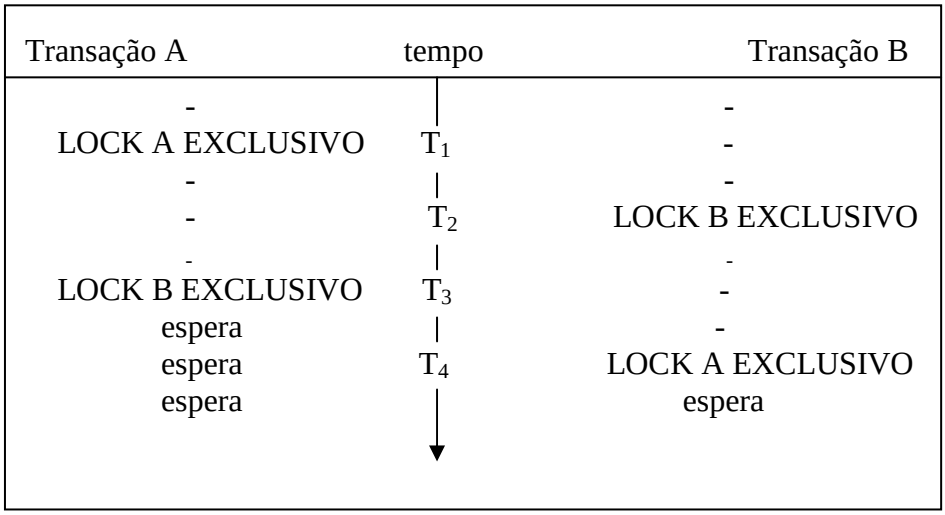


Fonte: Garcia-Molina (2001)

3.4.2 DEADLOCK

Segundo Date (2000), o *deadlock* é uma situação na qual duas ou mais transações estão em estado de espera simultânea, cada uma esperando que uma das outras libere um bloqueio antes de poder prosseguir (exemplo na figura 3).

FIGURA 3: UM EXEMPLO DE *DEADLOCK*



Fonte: Date (2000)

3.4.2.1 DETECÇÃO DE *DEADLOCK* PELO TEMPO LIMITE

Segundo Bernstein (1987), o escalonador precisa de uma estratégia para detectar *deadlocks*, para que as transações não fiquem bloqueadas para sempre. Uma estratégia é chamada ***timeout***. Se o escalonador achar que uma transação estiver um longo período de tempo esperando para obter um recurso, então ele simplesmente supõe que pode haver um *deadlock* envolvendo essa transação e consequentemente deverá abortá-la. No entanto, o escalonador pode estar supondo que a transação esteja envolvida em um *dedlock*, e cometer algum engano. Ele abortará a transação que não faz realmente parte de um *deadlock*, mas está esperando por um bloqueio adquirido por outra transação que está levando bastante tempo para terminar.

Uma maneira de evitar esse tipo de engano é usar um *timeout* com um grande período de tempo. Quanto maior for o período de tempo do *timeout*, maior são as chances do escalonador abortar as transações que estão realmente envolvidas em *deadlocks*. De qualquer modo, um longo período de *timeout* requer uma grande responsabilidade do escalonador, pois

ele não notifica que uma transação está em *deadlock* até o período de *timeout* se esgotar e conseqüentemente, esse período deve ser um parâmetro que precisa ser bem estudado.

3.4.2.2 O GRAFO DE ESPERAR POR

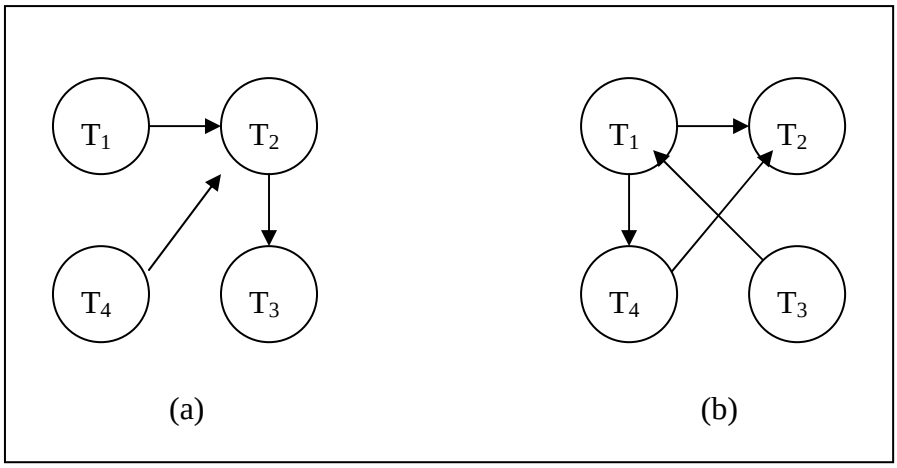
Segundo Garcia-Molina (2001), outra maneira de tratar os impasses causados por transações que esperam por bloqueios mantidos por outras podem ser resolvidos por um grafo de “esperar por”, indicando quais transações estão esperando por bloqueios mantidos por outra transação. Esse grafo pode ser usado para detectar impasses depois deles terem se formado, ou para evitar que esses impasses se formem. Deve-se supor esse último caso, que exige a manutenção do grafo de “esperar por” a toda hora, recusando-se a permitir uma ação que crie um ciclo no grafo.

Segundo Bernstein (1997), uma tabela de bloqueio mantém para cada elemento do banco de dados X uma lista das transações que estão esperando por bloqueios sobre X, bem como transações que retêm atualmente bloqueios sobre X. O grafo de “esperar por” tem um nó para cada transação que mantém um bloqueio ou está esperando por um. Há um arco do nó (a transação) T para o nó U se existe algum elemento do bando de dados A, tal que:

1. U mantém um bloqueio sobre A;
2. T está esperando por um bloqueio sobre A;
3. T não pode obter um bloqueio sobre A em seu modo desejado, a menos que U primeiro libere seu bloqueio sobre A.

A figura 4 mostra a situação do grafo de “esperar por” antes e depois da ocorrência do *deadlock.*

FIGURA 4: GRAFO DE ESPERAR POR ANTES E DEPOIS DO DEADLOCK



Fonte: adaptado de Garcia-Molina (2001)

Segundo Garcia-Molina (2001), se não houver nenhum ciclo no grafo de “esperar por”, então cada transação poderá eventualmente se completar. Haverá pelo menos uma transação que não espera por nenhuma outra transação, e essa transação sem dúvida poderá se completar. Nesse momento, haverá pelo menos uma outra transação que não estará esperando, a qual poderá se completar e assim por diante.

Porém, se houver um ciclo, então nenhuma transação no ciclo poderá prosseguir, e assim haverá um impasse. Desse modo, uma estratégia para evitar impasses é fazer a reversão (ROLLBACK) de qualquer transação que faça uma solicitação que poderia gerar um ciclo no grafo de “esperar por”.

4 SISTEMAS DISTRIBUÍDOS

Segundo Lampson (1988), a maior parte dos sistemas de computação são constituídos dos seguintes ingredientes: hardware, software de sistema, dados do sistema, software de usuário e dados. Pode-se então classificar como sistema distribuído todo sistema no qual algum desses ingredientes esteja distribuído.

Segundo Ramakrishnam (2000), para gerenciar um sistema de base de dados distribuído será necessário utilizar software distribuído (cada *site* é tipicamente gerenciado por um SGBD), hardware e dados distribuídos (os dados podem ser armazenados em diversos *sites* na rede).

Neste capítulo será estudado uma arquitetura especial de sistemas distribuídos: arquitetura **cliente-servidor**.

4.1 SISTEMAS CLIENTE/SERVIDOR

Segundo Date (2000), os sistemas cliente/servidor podem ser considerados um caso particular de sistemas distribuídos. Mais precisamente, um sistema cliente/servidor é um sistema distribuído em que (a) alguns *sites* são clientes e outros são servidores, (b) todos os dados residem nos *sites* servidores, (c) todas as aplicações são executadas nos *sites* clientes e (d) a operação não é uniforme (não é fornecida completa independência de local).

Segundo Date (2000), a expressão “cliente/servidor” se refere principalmente a uma arquitetura, ou divisão lógica de responsabilidades; o **cliente** é a aplicação (também conhecida como *front-end*) e o servidor é o SGBD (também chamado *back-end*). Porém, é precisamente pelo fato de que o sistema geral pode ser dividido de modo tão simples em duas partes que surge a responsabilidade de executá-las em duas máquinas diferentes. E essa última possibilidade é tão atraente que a expressão “cliente-servidor” veio a ser aplicada quase exclusivamente ao caso em que o cliente e o servidor estão de fato em máquinas diferentes.

Contudo ainda são possíveis diversas variações sobre o tema básico:

a) Vários clientes poderiam compartilhar o mesmo servidor (de fato, esse é o caso normal);
b) Um único cliente poderia ser capaz de ter acesso a vários servidores. Esse caso por sua vez se subdivide em dois outros:
    - O cliente está limitado a obter acesso a um único servidor de cada vez – isto é, cada solicitação individual de banco de dados deve ser dirigida a um só servidor; não é possível, dentro de uma única solicitação, combinar dados de dois ou mais servidores. Além disso, o usuário tem de saber qual servidor em particular armazena quais fragmentos de dados;
    - O cliente pode obter acesso a muitos servidores simultaneamente – isto é, uma única solicitação de banco de dados pode combinar dados de vários servidores, o que significa que os servidores aparentam para o cliente serem um único servidor e o usuário não precisa saber qual servidor armazena quais fragmentos de dados.

Contudo, esse último caso é um verdadeiro banco de dados distribuído (a operação é uniforme); ele não é aquilo que se costuma entender pela expressão “cliente/servidor”.

4.2 OBJETOS DISTRIBUÍDOS

Segundo Grimes (1997), desenvolvedores muitas vezes precisam simplificar a apresentação dos dados utilizando um objeto para encapsular o código e os dados necessários para acessar uma janela ou um terminal. De uma forma semelhante, eles podem usar um objeto no cliente para abstrair o acesso à base de dados. Para o cliente, todos os acessos à base de dados podem ser feitos através deste objeto. Isto significa que um objeto local representa a base de dados, embora os objetos possam ser usados no servidor (objetos remotos), e o cliente não precisa conhecer nada sobre o funcionamento deste objeto.

Comunicações entre programas clientes e a base de dados são bem estabelecidas, e existe um grande número de protocolos de rede e APIs (*Application Program Interface*) que permitem dois programas se comunicarem entre si. Assim, se um programa implementa alguma funcionalidade, é possível para outro programa usar tal funcionalidade. Por isso, visto

que os métodos do objeto são a própria funcionalidade deste objeto, é possível acessá-los através da rede.

Para fazer isso, o objeto servidor tem que se tornar público, para que dessa forma qualquer objeto cliente possa acessá-lo; o objeto servidor além disso deve atender requisições de clientes no próprio objeto, e despachar as chamadas para o objeto. Além do mais, o cliente deve fazer uma chamada na rede para se conectar ao objeto, e precisa passar o método requisitado de uma maneira que o objeto servidor reconheça.

Do ponto de vista do cliente, ele precisa saber como criar o objeto remoto, como exatamente comunicar para o objeto chamar seus métodos, e como extrair o resultado da resposta enviada pelo objeto. Isto requer uma grande quantidade de chamadas de API de rede, e não é o tipo de programação que o desenvolvedor do cliente precisa fazer: ele está somente preocupado em pegar os dados e apresentá-los. Desta forma, a programação seria confusa e não muito intuitiva.

Segundo Grimes (1997), para facilitar este problema, o desenvolvedor do objeto deverá encapsular todas estas chamadas de rede em um objeto "sombra" (ou *proxy*) que enxerga como o objeto servidor, em termos de métodos implementados. De qualquer forma, essa implementação dos métodos do objeto deverá ser passada fora das chamadas de rede para transferi-las para um objeto remoto. O desenvolvedor do objeto poderá então escrever o código necessário no programa servidor para aceitar as chamadas e passá-las para o objeto.

Agora, o desenvolvedor do cliente pode criar um objeto e usá-lo sem se preocupar com qualquer detalhe de programação referente à comunicação na rede. O desenvolvedor somente cria o objeto e chama seus métodos. O objeto é mais do que somente remoto. Ele é distribuído.

Objetos distribuídos são usados por vários motivos: eles poderiam, por exemplo, executar cálculos complexos que necessitam de uma alta especificação de máquina; ou talvez, por motivos de autorização, somente uma cópia do código pode estar executando. Um objeto remoto poderia também facilmente tornar-se o cliente de outro objeto, e estes objetos remotos (e a funcionalidade que eles fornecem) podem ser distribuídos através da rede.

4.2.1 AMBIENTE DE COMPUTAÇÃO DISTRIBUÍDA (RPC)

Segundo Grimes (1997), a *Open Software Foundation* (OSF) definiu um ambiente de computação distribuída (DCE), que fornece muitas facilidades, como segurança, serviços de diretório, *time services*, *threads*, e chamadas de procedimentos remotos (RPC). O DCE RPC definiu uma API para chamadas de funções remotas de um cliente para um servidor executando eu uma máquina remota. Os programadores devem definir quais funções devem ser chamadas através do mecanismo RPC usando o DCE *interface definitions language* (IDL). A IDL agrupa todas as funções dentro da interface, e o programa servidor na máquina servidora implementa estas funções. Através da IDL, tanto o cliente como o servidor conhecem quais funções estão na interface, e quais são os parâmetros destas funções.

Quando um cliente deseja usar uma interface de um servidor, ele precisa identificar qual interface usar. O DCE definiu *Universally Unique* IDs (UUIDs), um número de 128 bits, para identificar interfaces. Como um UUID é único, quando um cliente requisita um servidor que implementa uma interface com um UUID específico, o cliente pode garantir que o servidor apresenta exatamente a interface que ele espera, e este meio que a interface implementou no servidor terá os métodos e parâmetros esperado pelos clientes.

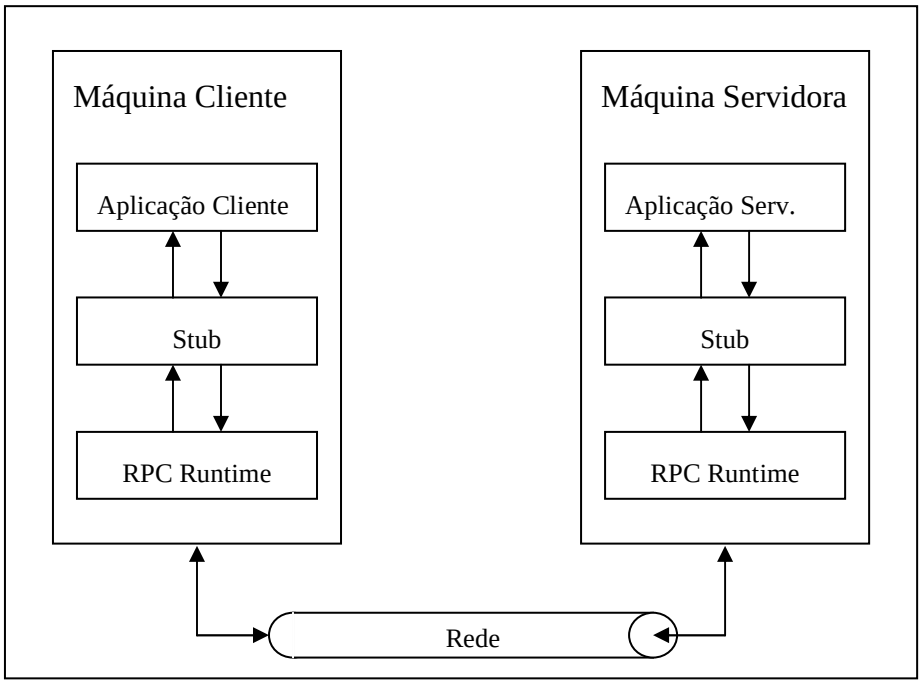
4.2.2 O MECANISMO RPC

Segundo Grimes (1997), as atuais chamadas de funções remotas são mandadas pra fora via seções de código chamadas ***code stub*** qua são criadas pelo compilador IDL. O *stub code* pode ser ligado estaticamente dentro de executáveis ou, em um sistema operacional como Windows, o *stub code* pode ser compilado dentro de uma DLL separada. De qualquer maneira, as funções que são implementadas no *stub code* no lado do cliente enxerga como as funções que estão implementadas no servidor. O cliente está totalmente inconsciente que o código está implementado em outra máquina. O *stub* do lado do cliente empacota a função requisitada e os parâmetros do método e, através do RPC *runtime library*, envia a requisição pela rede para a máquina servidora. Na máquina servidora, o servidor RPC já está executando e esperando por conexões de clientes. Se o servidor não estiver executando, o cliente não pode chamar seus métodos, já que não existe o mecanismo de ativação remota em RPC. Uma vez que a conexão está feita, o RPC *runtime* chama o *stub* do lado do servidor, o qual desempacota a requisição do cliente e chama o método no servidor. Qualquer resposta do

código do servidor é empacotada e enviada para o cliente. Durante este tempo, a *thread* do cliente que faz a chamada é efetivamente bloqueada: portanto as chamadas RPC são síncronas.

A figura 5, mostra como funciona a estrutura de um mecanismo RPC:

FIGURA 5: MECANISMO RPC



Fonte: adaptado de Grimes (1997)

4.2.3 MICROSOFT RPC E DCE

Segundo Grimes (1997), DCE é implementado por muitas plataformas. A Microsoft pegou DCE RPC e implementou em seu próprio sistema operacional. Microsoft RPC é compatível com DCE RPC, mas embora a Microsoft RPC forneça (quase) todas as facilidades de DCE RPC, é necessário utilizar alguns softwares adicionais para permitir que programas Microsoft RPC se comuniquem com programas DCE RPC. Isto porque a Microsoft tem seus próprios caminhos para lidar com questões de segurança e serviços de nomeação.

Segundo Grimes (1997), as funções DCE RPC implementadas pela Microsoft foram renomeadas para usar o padrão de capitalização da Microsoft. Contudo, existe um mapeamento de um-para-um. Você pode compilar programas DCE padrão em plataformas

Microsoft usando um arquivo *header* para mapear nomes de funções DCE para nomes Microsoft.

Note também, que com poucas exceções (atributos como **[*async*]**), o compilador de IDL da Microsoft, **MIDL**, suporta todas as funcionalidades do compilador DCE IDL. MIDL compila a IDL para produzir o *stub code* do lado do cliente e do lado do servidor, bem como um *header* que descreve a interface.

4.2.4 OBJETOS E INTERFACES

Segundo Grimes (1997), caso se deseje escrever programas clientes e servidores, por exemplo, em C++, e agregar sua interface para os métodos de uma classe, pode-se fazê-lo. A Microsoft RPC dá suporte aos princípios de orientação a objetos como encapsulamento e polimorfismo, mas a DCE RPC, e conseqüentemente a Microsoft RPC, não dá suporte a herança.

4.2.5 ORB – OBJECT REQUEST BROKER

Segundo Geraghty (1999), *object request brokers* fornece a funcionalidade da interface da aplicação e um transporte que permite a transmissão de sequências de requisições e respostas entre objetos. Geralmente uma IDL suporta operações de interface de aplicação, embora exista uma segunda opção usando um mecanismo de interfaces dinâmicas.

Em termos de transporte, ORB fornece um mais alto nível protocolo condutor acima do protocolo de rede.

Geralmente, os dois principais ORBs usados em sistemas comerciais são: **CORBA (*Common Object Request Broker Architecture*)**, da OMG (*Object Management Group*) e **COM (*Component Object Model*),** da Microsoft.

4.3 CORBA

Segundo Grimes (1997), a OMG fornece uma especificação para um sitema de objetos distribuídos através do CORBA, uma implementação comercial robusta que está disponível e bastante em uso.

4.3.1 ARQUITETURA CORBA

Segundo Geraghty (1999), para conectar um cliente a um objeto que executa em um espaço diferente de endereço, CORBA usa um mecanismo ORB. Sendo que a interação do objeto ocorre através do mecanismo ORB, ele é muitas vezes referenciado como sendo um encaminhador de objetos, pois este fornece um software de encaminhamento que conecta objetos. Portanto, CORBA é uma especificação, e define uma interface que especifica como clientes podem manipular objetos via ORB, mas não define uma implementação. Dessa forma, os fornecedores estão livres para implementar seus ORBs da forma que eles desejarem. Um ORB pode ser implementado a partir de DCE RPC, por exemplo; de acordo com a especificação, o objeto cliente e servidor são obviamente feitos dessa forma.

Em relação a programação, CORBA trabalha de uma forma semelhante para DCE RPC. O programador define a interface do objeto usando a linguagem de descrição de interfaces CORBA (IDL), e então é compilado com o compilador de IDL do fornecedor. Note que a IDL de CORBA é diferente para DCE ou Microsoft IDL.

Geraghty (1999) mostra um simples exemplo da IDL da OMG através do quadro 5:

QUADRO 5: IDL DA OMG

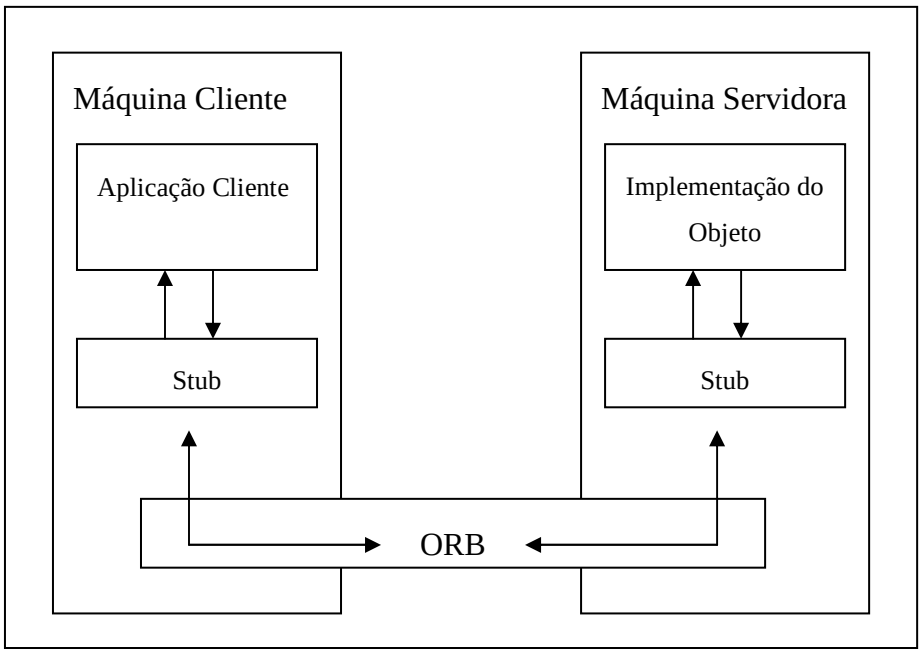
```
// OMG IDL
typedef long HRESULT;
exception COM_ERROR
{
        long hresult;
};
exception COM_ERROREX
{
        long hresult;
        Any info;
};
interface Isimple
{
        HRESULT op1(in float f, inout float io, out float o)
                raises (COMM_ERROR, COMM_ERROREX);
        attribute float f;
        readonly attribute long l;
}
```

Fonte: Grimes (1997)

O compilador de IDL gera um código ***stub***, o qual fica na máquina cliente, e um código ***skeleton***, o qual fica na máquina servidora. Dependendo do fornecedor e do sistema operacional, *stubs* e *skeletons* podem ser em C, C++, Smalltalk, ou quase sempre Java.

A figura 6, apresentada por Grimes (1997), mostra como funciona a estrutura do mecanismo de RPC CORBA:

FIGURA 6: MECANISMO DE RPC CORBA



Fonte: adaptado de Grimes (1997)

Segundo Grimes (1997), o cliente usa a API CORBA para requisitar o objeto através de um processo conhecido como ***binding*** (ou ligação). Essa ligação mapeia a interface do objeto através de uma linguagem que é usada no cliente. Pode ser uma classe em C++, java, ou um conjunto de funções em C.

Segundo Grimes (1997), essa linguagem de mapeamento precisa fornecer um mecanismo para o cliente especificar o objeto específico em uma máquina específica. Isto é feito através da **referência de objetos**. A referência do objeto é um ID que identifica como único o objeto através da rede. É responsabilidade do ORB fornecer a referência do objeto, e isso é assegurado pela implementação do *stub*, para ser passado de volta para o ORB sempre que o objeto é acessado.

4.4 *DISTRIBUTED COMPONENT OBJECT MODEL (DCOM)*

Segundo Geraghty (1999), em COM existe uma separação definida entre interface e implementação – dado um componente o qual disponibiliza um conjunto de interfaces, a perspectiva dos usuários do componente é que não precisa conhecer como estas interfaces são implementadas; a interface tem dessa forma um significado de encapsulamento forçado em COM, como em CORBA.

Segundo Grimes (1997), DCOM é uma especificação para objetos distribuídos proprietária da Microsoft e está disponível somente para seus sistemas operacionais, a partir do Windows NT 4.0 para servidores, e Windows 95 para estações. Diferente disso, CORBA é uma especificação normativa e está disponível para diversos sistemas operacionais.

Uma outra grande diferença, contudo, é que COM é um padrão binário para interoperabilidade de componente, e o *layout* binário da interface em COM é crucial, diferente de CORBA – uma prova é o fato de que *Interface Repository* em CORBA não precisa reter informações sobre a ordem dos métodos. O motivo é que CORBA realiza sua interoperabilidade através das linguagens ligando-as à IDL. Por exemplo, para escrever programas CORBA em C++ é preciso ter um compilador IDL o qual gera código em C++ mediante o mapeamento IDL-to-C++. O mesmo se aplica para escrever programas CORBA em Java, C, etc. COM, contudo, confia na imutabilidade do *layout* da interface (que é sua *v-table*) para permitir clientes escreverem em diversas linguagens para usar o dado componente. Dessa forma, duas interfaces que são idênticas em todos outros aspectos (até agora como número de operações, seus nomes, etc.) mas tem seus métodos em uma ordem diferente são, para COM, interfaces distintas. O motivo disso é que um dado deslocamento em *v-table* denotará um diferente método em cada caso. Por esta razão, para identificar não somente uma dada interface mas também uma versão específica de uma interface, COM ordena que um UUID distinto seja associado com cada interface e com qualquer revisão desta interface. Isto permite clientes requisitarem um ponteiro de uma versão específica para uma interface de conhecimento que atualiza o componente para estar inversamente compatível e ser refletido em uma nova interface com seu próprio UUID distinto. Em COM, diferente de CORBA, existe um mecanismo padrão pelo qual um objeto pode ter múltiplas interfaces (sem qualquer relacionamento de herança entre estas interfaces) e pelo qual clientes possam navegar de uma interface suportada para outra. A navegação é suportada pelo fato de que todo

objeto COM suporta uma interface chamada **IUnknown**. A definição desta interface pode ser visulizada no quadro 6.

QUADRO 6: DEFINIÇÃO DA INTERFACE IUNKNOWN

```
// MIDL
interface IUnknown
{
        typedef [unique] IUnknown *LPUNKNOWN;
        HRESULT QueryInterface ([in] REFIID riid,
                [out, iid_is(riid)] void** ppvObject);
        ULONG AddRef();
        ULONG Release();
}
```

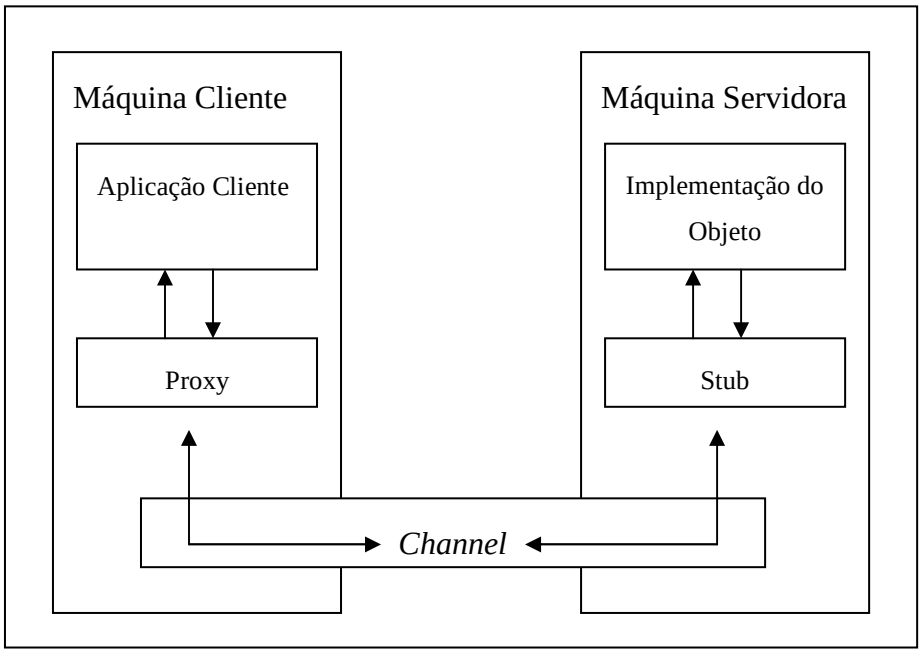
Fonte: adaptado de Grimes (1997)

Segundo Gerahgty (1999), todas interfaces COM finalmente derivam de IUnknown, dessa forma permitindo clientes determinar, em tempo de execução, que interfaces um objeto suporta.

Segundo Geraghty (1999), o cliente acessa esta interface através de um trecho de código chamado **proxy** (stub, na terminologia RPC). O proxy é apresentado para o cliente como a interface que ele requisitou, mas o proxy pode ser código gerado pelo compilador Microsoft IDL para despachar a requisição para o servidor. O próprio servidor pode ou não estar na mesma máquina que o cliente.

As chamadas do método da interface são enviados, via ***channel*** (ou canal), para o código **stub** na máquina servidora. O canal pode ser mensagens do Windows, se o servidor estiver na máquina local, ou pode ser algum mecanismo RPC, como Microsoft RPC. O stub recebe a requisição, e invoca o método na interface do objeto servidor. A estrutura do mecanismo de RPC COM/DCOM é demonstrada na figura 7.

FIGURA 7: MECANISMO RPC EM COM/DCOM



Fonte: adaptado de Grimes (1997)

Segundo Grimes (1997), para habilitar os clientes a achar os objetos requeridos por eles, todos os objetos são registrados no sistema de registro local. Como no DCE, interfaces e classes COM têm UUIDs que são chamados IIDs (*interface Ids*) e CLSIDs (*class Ids*) respectivamente, GUIDs em geral, que tornam classes do objeto serem unicamente identificados. O sistema de registros permite que informações sobre o código do executável que implementa um objeto sejam salvas, onde elas residem, e qual código implementa os *proxies* para as interfaces. Esta construção flexível, permite por exemplo, que um objeto possa ser implementado como um objeto *in-process*, ou como um objeto remoto em outra máquina. O cliente pode explicitamente escolher uma destas implementações, ou ele pode deixar com que o próprio COM escolha.

4.4.1 CONNECTION POINTS

Segundo Grimes (1997), quando é feita uma chamada de um método em um objeto, a *thread* do cliente faz com que a chamada seja bloqueada. O bloqueio ocorre no método de manipulação no *proxy*, o qual fica esperando o *stub* retornar. A *thread* do cliente pode somente continuar executando quando o método do objeto retornar. Se o método do objeto levar um longo tempo para se completar, o cliente deverá aceitar isso, implicando na sua performance.

Grimes (1997) cita dois caminhos para aliviar este problema. Um caminho seria o cliente criar uma *thread* separada para fazer as chamadas dos métodos. Outro caminho é o servidor criar uma *thread* separada para manipular a tarefa e então imediatamente retornar a chamada do método. No segundo caso, o cliente deveria achar uma maneira de saber quando o servidor terminou sua tarefa. Ele poderia ou ficar questionando o servidor, o qual seria dispendioso em chamadas de rede, ou passar para o servidor um ponteiro para um ***callback*** que o servidor pode chamar quando ele terminar sua tarefa.

Connection Points funcionam de uma maneira semelhante: Quando um evento em particular ocorre, o servidor chamará um *callback* no cliente. Em se tratando de COM, diz-se que o servidor possui uma interface ***outgoing*** ou **de partida** . Uma interface deste tipo é definida na IDL do servidor, mas é implementada no cliente.

Para o MIDL criar este código de comunicação, o servidor IDL declara que ele usará essa interface *outgoing* usando o atributo **[source].** Assim o servidor chamará esta interface para indicar ao cliente que alguma coisa ocorreu, e desse modo o servidor é um **código** de notificações.

As outras interfaces na IDL que não são marcadas como [source] são conhecidas como interfaces ***incoming*** ou **de chegada**, pelo fato de que esta interface será chamada pelo cliente. Estas são as interfaces que o servidor implementa. No contexto de *connection points*, a interface de chegada no cliente é chamada de ***sink***.

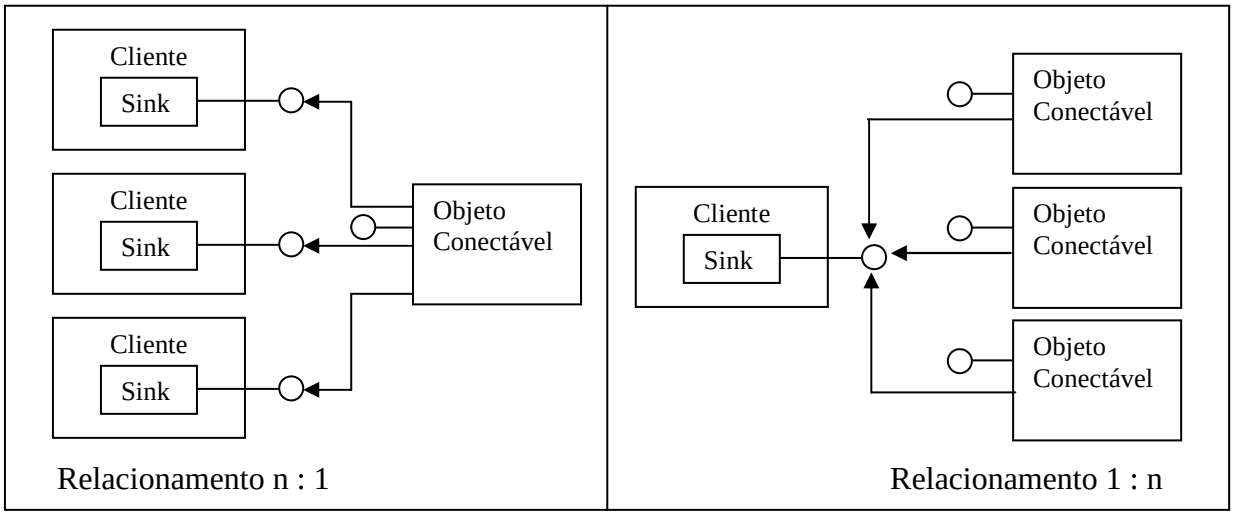
Segundo Grimes (1997), para fazer uma conexão, um cliente implementa uma interface *sink* a qual permitirá ele se conectar a um específico tipo de objeto. Quando esse cliente precisar fazer uma conexão para um objeto conectável, ele precisa verificar que o

objeto estará apto a chamar a interface *sink* do cliente. Uma vez feito isso, o cliente pode ganhar uma conexão ao objeto e fornecer a ele um ponteiro para a interface *sink* do cliente.

Uma vez feita a conexão, o cliente pode fazer uma chamada do método na interface de chegada do objeto, o qual iniciará algum processamento, em uma *thread* separada. Com o processamento iniciado, o objeto pode retornar da chamada do objeto, o qual desbloqueia a *thread* no cliente que chamou o objeto. Quando o processamento se completar, o objeto pode chamar um método na interface *sink* do cliente, de modo a indicar que existem dados disponíveis ou, naturalmente, fornecer o próprio dado.

Segundo Grimes (1997), esta arquitetura é muito flexível: um cliente pode ter multiplas conexões em muitos objetos conectáveis, e um objeto conectável pode ter conexões com muitos clientes, como mostra a figura 8:

FIGURA 8: ARQUITETURA DE *CONNECTION POINTS*



Fonte: adaptado de Grimes (1997)

5 TRANSAÇÕES DISTRIBUÍDAS

Segundo Date (2000), há dois aspectos principais do gerenciamento de transações, o controle de recuperação e o controle de concorrência, cada um deles exigindo um extenso tratamento no ambiente distribuído. Para explicar esse extenso tratamento, é preciso introduzir um novo termo **agente**. Em um sistema distribuído, uma única transação pode envolver a execução de código de vários *sites*; em particular, pode envolver atualizações em muitos *sites*; Diz-se então que cada transação consiste em vários **agentes**, onde um agente é o processo executado em virtude de uma determinada transação em um *site* específico. E o sistema precisa saber quando dois agentes são ambos parte da mesma transação – por exemplo, dois agentes que fazem parte da mesma transação obviamente não podem estar em conflito.

Segundo Garcia-Molina (2001), o modelo do que é uma transação pode mudar. Uma transação não é mais um fragmento de código executado por um único processador que se comunica com um único escalonador e um único gerenciador de *log* em um único local. Em vez disso, uma transação consiste em componentes de transações que se comunicam, cada qual em um local diferente, comunicando-se com o escalonador e gerenciador de *log* locais.

5.1 PROTOCOLOS DE CONSOLIDAÇÃO

5.1.1 CONSOLIDAÇÃO DE DUAS FASES

Segundo Date (2000), a consolidação de duas fases é importante sempre que uma determinada transação pode interagir com vários “gerenciadores de recursos” independentes, cada um gerenciando seu próprio conjunto de recursos recuperáveis e mantendo seu próprio *log* de recuperação.

Segundo Bernstein (1997), quando uma transação atualiza dados em dois ou mais sistemas em um ambiente distribuído, há que se assegurar a propriedade atomicidade, isto é, ou ambos os sistemas consolidam suas atualizações, ou nenhum. A solução é um protocolo especial chamado *two-phase commit* (2PC), que é executado por um módulo especial chamado gerenciador de transação.

O ponto principal do problema é que uma transação pode consolidar suas atualizações em um sistema, mas o segundo sistema falha antes que a transação as consolide também.

Neste caso, quando o sistema que falhou se recuperar, ele deve estar apto a consolidar a transação.

Segundo Garcia-Molina (2001), seguem-se vários pontos de destaque sobre o protocolo de consolidação de duas fases:

- Em uma consolidação de duas fases, supões-se que cada local anota as ações desse local, mas não há nenhum *log* global
- Também supõe-se que um local, também chamado de coordenador, desempenha um papel especial no ato de decidir se a transação distribuída pode ou não ser consolidada.
- O protocolo de consolidação de duas fases envolve o envio de certas mensagens entre o coordenador e os outros locais. À medida que cada mensagem é enviada, ela é anotada no local de envio, para ajudar na recuperação, caso seja necessário.

Com esses pontos em mente, pode-se descrever as duas fases em termos das mensagens enviadas entre os locais.

Fase I

Segundo Garcia-Molina (2001), na primeira etapa da consolidação de duas fases, o coordenador para uma transação distribuída T decide quando deve tentar consolidar T. Supõe-se que a tentativa de consolidação ocorre após o componente de T no local do coordenador estar pronto para consolidar mas, em princípio, as etapas devem ser executadas ainda que o componente do coordenador queira abortar (mas com simplificações óbvias, como será visto). O coordenedor consulta todos os locais com componentes de transação T, a fim de determinar sua opinião em relação à decisão de consolidar/abortar:

1. O coordenador insere um registro de *log* <*Prepare T*> no *log* de seu local;
2. O coordenador envia para cada local do componente (em princípio, incluindo a si mesmo) a mensagem *prepare T;*
3. Cada local que recebe a mensagem *prepare T* decide se deve consolidar ou abortar seu componente de T. O local pode se atrasar, se o componente ainda não tiver completado sua atividade, mas eventualmente deve enviar sua resposta;

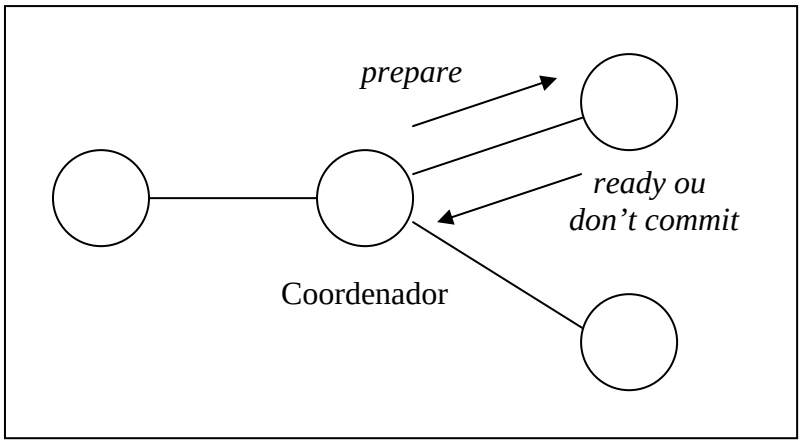
4. Se um local quiser consolidar seu componente, deve entrar em um estado chamado **pré-consolidado**. Uma vez no estado de pré-consolidado, o local não poderá abortar seu componente de T sem uma orientação para fazê-lo do coordenador. As etapas a seguir são executadas de forma que o componente se torne pré-consolidado:
    - Executar toda as etapas necessárias para assegurar que o componente local de T não terá de abortar, ainda que exista uma falha do sistema seguida pela recuperação no local. Desse modo, não apenas todas as ações associadas com a T local serão executadas, mas as ações apropriadas relativas ao *log* devem ser executadas de forma que T seja refeita, em vez de ser desfeita em uma recuperação. As ações dependem do método de registro de *log*, mas seguramente os registros de *log* associados com as ações de T local terão de ser esvaziados para o disco.
    - Inserir o registro <*Ready T*> no *log* e esvaziar o *log* para o disco.
    - Enviar ao coordenador a mensagem *ready T.*

Porém, o local não consolida seu componente de T nesse momento; ele deve esperar pela fase 2.

5. Se, em vez disso, o local quiser abortar seu componente de T, então ele anotará o registro <*Don't Commit T*> e enviará a mensagem *don't commit T* ao coordenador. É seguro abortar o componente nesse momento, pois T sem dúvida abortará se até mesmo um componente quiser abortar.

A figura 9 mostra o fluxo de mensagens da fase I.

FIGURA 9: MENSAGENS DA FASE I



Fonte: Garcia-Molina (2001)

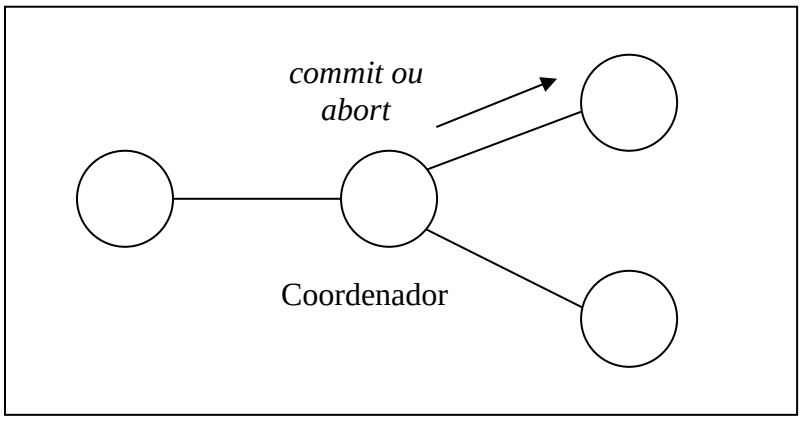
Fase II

Segundo Garcia-Molina (2001), a segunda fase começa quando as respostas *ready* ou *don't commit* são recebidas de cada local pelo coordenador. Porém, é possível que algum local deixe de responder; ele pode estar fora de serviço ou pode ter sido desconectado pela rede. Nesse caso, depois de um espaço de tempo limite satisfatório, o coordenador tratará o local como se ele tivesse enviado um *don't commit*.

1. Se o coordenador tiver recebido *ready T* de todos os componente de T, então ele decidirá consolidar T. O coordenador:
    - Anota <C*ommit T*> em seu local, e
    - Envia a mensagem *commit T* a todos os locais envolvidos em T.
2. Se tiver recebido *don't commit T* de um ou mais locais, o coordenador:
    - Anotará <A*bort T*> em seu local e,
    - Enviará mensagens *abort T* a todos os locais envolvidos em T.
3. Se um local receber uma mensagem *commit T*, ele consolidará o componente de T nesse local, anotando <*Commit T*> ao fazê-lo;
4. Se um local receber a mensagem *abort T*, ele abortará T e gravará o registro de *log* <*Abort T*>.

As mensagens da fase 2 estão resumidas na figura 10:

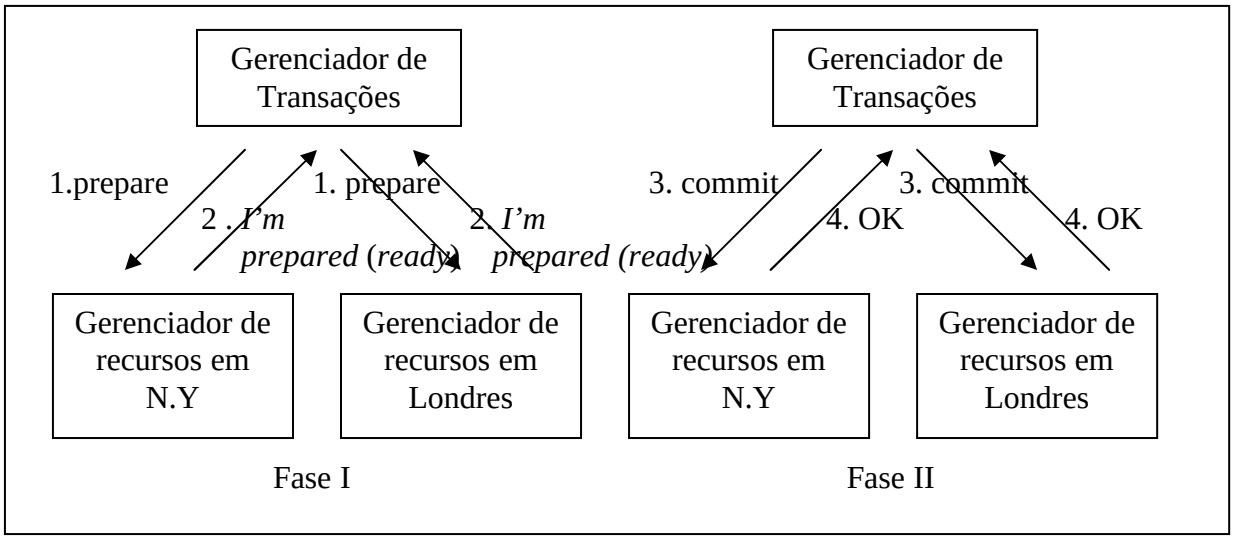
FIGURA 10: MENSAGENS DA FASE II



Fonte: Garcia-Molina (2000)

As mensagens da fase 1 e fase 2 estão resumidas na figura 11:

FIGURA 11: MENSAGENS DAS FASES I E II



Fonte: adaptado de Bernstein (1997)

5.2 RECUPERAÇÃO DISTRIBUÍDA

Segundo Ramakrishnam (2000), recuperação em um SGBD distribuído é mais complicado do que em um SGBD centralizado pelos seguintes motivos:

a) Novas espécies de falhas podem surgir, isto é, falhas de *links* de comunicação e falhas de um *site* remoto na qual uma subtransação está executando;

b) Ou todas as subtransações de uma dada transação devem consolidar, ou nenhuma consolidará, e essa propriedade deve ser garantida desprezando qualquer

combinação de falhas de *site* ou *link*. Esta garantia é alcançada usando um protocolo de consolidação.

Segundo Date (2000), o controle de recuperação em sistemas distribuídos em geral se baseia no protocolo de consolidação em duas fases (ou alguma variante dele). O COMMIT em duas fases é exigido em qualquer ambiente em que uma única transação pode interagir com vários gerenciadores de recursos autônomos; porém, é particularmente importante em um sistema distribuído, porque os gerenciadores de recursos em questão – isto é – os SGBD's locais estão esperando em *sites* distintos e, portanto, são muito autônomos.

Garcia-Molina (2001) comenta que é preciso ter certeza de que o que acontecerá quando o local se recuperar será coerente com a decisão global tomada sobre uma transação distribuída T. Segundo ele, há vários casos a considerar, dependendo da última entrada de *log* para T:

a) Se o último registro de *log* para T foi <*Commit T*>, então T deve ter sido consolidado pelo coordenador. Dependendo do método de *log* usado, pode ser necessário refazer o componente de T no local da recuperação.
b) Se o último registro de log foi <*Abort T*>, então de modo semelhante sabe-se que a decisão global foi a de abortar T. Se o método de *log* exigir, irá desfazer o componente de T no local da recuperação.
c) Se o último registro de *log* foi <*Don't Commit T*>, então o local sabe que a decisão global deve ter sido a de abortar T. Se necessário, os efeitos de T sobre o banco de dados local serão desfeitos.
d) O caso complicado ocorre quando o último registro de *log* para T é <*Ready T*>. Agora, o local da recuperação não sabe se a decisão global foi a de consolidar ou a de abortar T. Esse local deve se comunicar com pelo menos outro local para descobrir qual foi a decisão global quanto à T. Se o coordenador estiver ativo, o local poderá perguntar ao coordenador. Se o coordenador não estiver ativo no momento, algum outro local poderá ser solicitado a consultar seu *log* para descobrir o que aconteceu a T. No pior caso, nenhum outro local poderá ser contactado, e o componente local de T deverá ser mantido ativo até ser determinada a decisão de consolidar/abortar.

e) Também pode surgir o caso do *log* local não ter nenhum registro sobre T que vieram das ações do protocolo de consolidação de duas fases. Nesse caso, o local da recuperação poderá tomar uma decisão unilateral de abortar seu componente de T, o que é coerente com todos os métodos de registro de *log*. É possível que o coordenador já tenha detectado um tempo limite do local com falha e tenha decidido abortar T. Se a falha foi breve, T ainda poderá estar ativa em outros locais, mas ela nunca será inconsistente, se o local da recuperação decide abortar seu componente de T e responder com *don't commit T* se for consultado mais tarde na fase 1.

A análise anterior de Garcia-Molina (2001), pressupõe que o local onde houve a falha não é o coordenador. Quando o coordenador falha durante uma consolidação de duas fases, surgem novos problemas. Primeiro, os locais participantes sobreviventes devem esperar que o coordenador se recupere ou eleger um novo coordenador. Como o coordenador pode estar inativo por um período indefinido, há uma boa motivação para eleger um novo líder, pelo menos após um breve período de espera para ver se o coordenador volta à atividade.

Segundo Garcia-Molina (2001), a questão da eleição de um líder é por si própria um problema complexo de sistemas distribuídos. Porém, um método simples funcionará na maioria das situações. Por exemplo, pode-se supor que todos os locais participantes terão números de identificação exclusivos; endereços de IP funcionarão em muitas situações. Cada participante enviará mensagens anunciando sua disponibilidade como líder a todos os outros locais, fornecendo seu número de identificação. Após um período adequado, cada participante confirma como o novo coordenador o local de numeração mais baixa dentre as que ouviu, e envia mensagens informando esse fato a todos ou outros locais. Se todos os locais receberem mensagens consistentes, haverá uma única escolha como novo coordenador, e todos a conhecerão. Se houver inconsistência, ou um local sobrevivente deixar de responder, isso também será conhecido universalmente, e a eleição será reiniciada.

5.3 CONTROLE DE CONCORRÊNCIA DISTRIBUÍDO

Segundo Date (2000), na maioria dos sistemas distribuídos o controle de concorrência se baseia no bloqueio, exatamente como em sistemas não distribuídos. Porém, em um sistema distribuído, solicitações para testar, impor e liberar bloqueios se tornam mensagens (supondo-se que o objeto em consideração seja um *site* remoto), e mensagens significam sobrecarga.

5.3.1 SISTEMAS DE BLOQUEIO CENTRALIZADO

Segundo Garcia-Molina (2001), talvez a abordagem mais simples de bloqueio seja designar um local, o local de bloqueio, para manter uma tabela de bloqueio para elementos lógicos, quer eles tenham ou não cópias nesse local. Quando uma transação quiser um bloqueio sobre o elemento lógico X, ela enviará uma solicitação de bloqueio ao local, que concederá ou negará o bloqueio, conforme apropriado. Obter um bloqueio global sobre X é o mesmo que obter um bloqueio local sobre X no local de bloqueio, e então pode-se assegurar que os bloqueios globais se comportarão corretamente, desde que o local de bloqueio administre os bloqueios de forma convencional. O custo usual é de três mensagens por bloqueio (solicitação, concessão e liberação), a menos que a transação esteja sendo executada no local de bloqueio.

5.3.2 BLOQUEIO DE DUAS FASES DISTRIBUÍDO

Segundo Bernstein (1987), o bloqueio de duas fases também pode ser usado em um sistema distribuído. Como visto anteriormente, um sistema distribuído consiste em uma coleção de *sites* de comunicação, onde cada qual possui uma base de dados centralizada. Cada item de dados é armazenado em exatamente um *site*. Foi comentado que o escalonador gerencia os itens de dados armazenados neste *site*. Isto significa que o escalonador é responsável pelo controle de acesso a estes (e somente estes) itens.

Uma transação submete suas operações para o gerenciador de transações. O gerenciador de transações então transfere cada operação READ(X) ou WRITE(X) de uma transação para o escalonador que gerencia X. Quando (e se) um escalonador decidir processar READ(X) ou WRITE(X), ele envia as operações para seu local de gerenciamento de dados, o qual pode acessar X e retornar seu valor (para um READ) ou atualizá-lo (para um WRITE). A

operação de COMMIT ou ABORT é enviada para todos os *sites* onde a transação acessou os itens de dados.

O escalonador em todos os *sites*, mantidos ao mesmo tempo, constitui um escalonador distribuído. A tarefa do escalonador distribuído é processar as operações submetidas pelo gerenciador das transações em uma maneira (globalmente) serializável e recuperável.

Segundo Bernstein (1987), pode-se construir um escalonador baseado em 2PL. Cada escalonador mantém o bloqueio dos itens de dados armazenados em seu site e gerencia então de acordo com as regras de 2PL. Em 2PL, um READ(X) ou WRITE(X) é processado quando o bloqueio apropriado em X pode ser obtido, o qual depende somente de quais outros bloqueios de X foram requeridos. Por esta razão, cada escalonador local 2PL tem todas as informações que precisa para decidir quando processar uma operação, sem se comunicar com outros *sites*. Um pouco mais problemático é a emissão de quando liberar um bloqueio. Para forçar a regra de duas fases, um escalonador não pode liberar uma transação de Ti bloqueada até saber que Ti não submeterá qualquer outra operação a ele ou qualquer outro escalonador. Desse modo, um escalonador poderia liberar o bloqueio de Ti, e algum tempo depois outro escalonador poderia bloquear Ti, desse modo violando as regras de bloqueio de duas fases.

Acontece então que forçar a regra de duas fases requer comunicação entre os escalonadores em diferentes *sites*. Todavia, se os escalonadores usam estritamente 2PL, então eles podem evitar dessa maneira muita comunicação. Aqui está o porquê. O gerenciador das transações, ou coordenador, que gerencia a transação Ti envia COMMIT para todas os *sites* onde Ti acessou itens de dados. Na hora que o gerenciador decidir enviar COMMIT Ti para todos estes *sites*, ele precisa ter recebido as confirmações para todas as operações de Ti. Por esta razão, Ti terá certamente obtido todos os bloqueios que ele precisa. Desta maneira, se um escalonador liberar o bloqueio de Ti depois que foi processado COMMIT Ti, ele sabe que nenhum escalonador subseqüentemente exigirá qualquer bloqueio para Ti.

5.3.3 UM MODELO DE CUSTO PARA ALGORITMOS DE BLOQUEIO DISTRIBUÍDO

Garcia-Molina (2001) sugere o seguinte modelo de custo para algoritmos de sistemas distribuídos:

Suponha que cada elemento de dados exista em exatamente um local (isto é, não há nenhuma replicação de dados), e que o gerenciador de bloqueios em cada local armazena bloqueios e solicitações de bloqueio para os elementos em seu local. As transações podem ser distribuídas, e cada transação consiste em componentes de um ou mais locais.

Embora existam vários custos associados ao gerenciamento de bloqueios, muitos deles são fixos, independente do modo como as transações solicitam bloqueios em uma rede. O único fator de custo sobre qual tem-se controle é o número de mensagens enviadas entre locais quando uma transação obtém e libera seus bloqueios. Desse modo, conta-se o número de mensagens exigidas pelos diversos esquemas de bloqueio, considerando-se a hipótese de que todos os bloqueios sejam concedidos ao serem solicitados. É claro que uma solicitação de bloqueio pode ser negada, resultando em uma mensagem adicional para negar a solicitação e uma mensagem posterior quando o bloqueio for concedido. Porém, como não se pode prever a taxa de negações de bloqueios e, de qualquer modo, essa taxa não é algo que se pode controlar, será ignorado esse requisito adicional para as mensagens nas comparações ora efetuadas.

Segundo Garcia-Molina (2001), no método de bloqueio central, a solicitação de bloqueio típica usa três mensagens, uma para solicitar o bloqueio, uma do local central para conceder o bloqueio e uma terceira para liberar o bloqueio. As exceções são:

a) As mensagens são desnecessárias quando o local da solicitação for o local de bloqueio central, e
b) Mensagens adicionais devem ser enviadas quando a solicitação inicial não puder ser atendida.

Porém, suponha que ambas as situações sejam relativamente raras; isto é, a maioria das solicitações de bloqueio vem de locais diferentes do local de bloqueio central, e a maioria das solicitações de bloqueio podem ser concedidas. Desse modo, três mensagens por bloqueio é uma boa estimativa do custo do método de bloqueio centralizado.

Agora, considere uma transação mais flexível que a de bloqueio central, na qual cada elemento do banco de dados X pode manter seus bloqueios em seu próprio local (2PL). Pode parecer que, considerando-se uma transação que desejar bloquear X terá um componente no local de X, não haverá necessidade de mensagens entre locais. O componente local simplesmente negocia com o gerenciador de bloqueio nesse local para o bloqueio sobre X. Entretanto, se a transação distribuída precisa de bloqueios sobre vários elementos, por exemplo X, Y e Z, então a transação não poderá completar seus cálculos até obter o bloqueio sobre todos os três elementos. Se X, Y e Z estiverem em locais diferentes, então os componentes das transações nesse local deverão pelo menos trocar mensagens de sincronização, a fim de evitar que a transação “chegue antes de si mesma”.

Em lugar de lidar com todas as variações possíveis, adota-se um modelo simples de como as transações conseguem bloqueios. Suponha que um componente de cada transação, o coordenador de bloqueio (escalonador), tem a responsabilidade de juntar todos os bloqueios que todos os componentes da transação exigem. O coordenador de bloqueio impõe bloqueios sobre elementos de seu próprio local sem mensagens, mas bloquear um elemento X em qualquer outro local exige três mensagens:

a) Uma mensagem para o local X solicitando o bloqueio;
b) Uma mensagem de resposta concedendo o bloqueio (lembre que se supõe que todos os bloqueios são concedidos de imediato; se não forem, uma mensagem de negação seguida por uma mensagem de concessão posterior será enviada);
c) Uma mensagem ao local de X liberando o bloqueio.

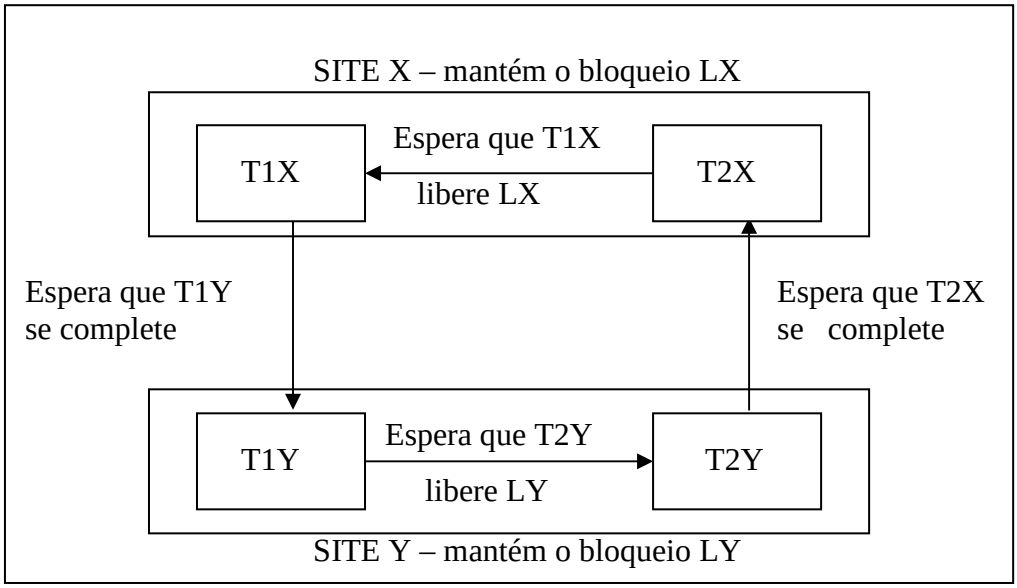
Como se deseja apenas comparar protocolos de bloqueio distribuído, em vez de dar valores absolutos para seu número médio de mensagens, essa simplificação servirá aos propósitos.

Se for escolhido como coordenador de bloqueio o local em que a maioria dos bloqueios é necessária para a transação, será minimizado o requisito de mensagens. O número de mensagens exigidas é três vezes o número de elementos do banco de dados nos outros locais (Garcia-Molina, 2000).

5.3.4 DEADLOCK DISTRIBUÍDO

Segundo Date (2000), outro problema com o bloqueio distribuído é que ele pode levar ao ***deadlock* global**. Um *deadlock* global é um *deadlock* envolvendo dois ou mais *sites*. Por exemplo, observe a figura 12:

FIGURA 12: EXEMPLO DE *DEADLOCK* DISTRIBUÍDO

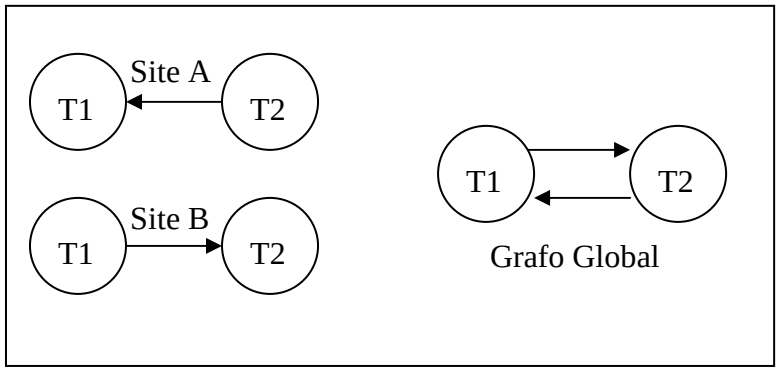


Fonte: Garcia-Molina (2001)

1. O agente da transação T2 no *site* X está esperando que o agente da transação T1 no *site* X libere um bloqueio.
2. O agente da transação T1 no *site* X está esperando que o agente da transação T1 no *site* Y se complete.
3. O agente de transação T1 no *site* Y está esperando que o agente de transação T2 no *site* Y libere o bloqueio.
4. O agente da transação T2 no *site* Y está esperando que o agente da transação T2 no *site* X se complete: *DEADLOCK!*

O problema com um *deadlock* como esse é que nenhum dos *sites* pode detectá-lo usando somente informações internas a esse *site*. Em outras palavras, não há ciclos nos grafos de “esperar por” locais, mas aparecerá um ciclo se esses dois grafos locais se combinarem para formar um grafo global. Segue-se que a detecção de *deadlock* global acarreta mais sobrecarga de comunicação, porque exige que grafos individuais locais sejam reunidos de alguma maneira, como demonstra a figura 13.

FIGURA 13: GRAFO GLOBAL DE DETECÇÃO DE *DEADLOCK*



Fonte: adaptado de Ramakrishnam (2000)

Segundo Ramakrishnam (2000), para detectar *deadlocks*, um algoritmo de detecção de *deadlock* distribuído precisa ser usado, e cita três algoritmos.

O primeiro algoritmo, que é centralizado, consiste em mandar periodicamente todos os grafos “esperar por” locais para algum *site* que é responsável pela detecção de um deadlock global. Neste *site*, o grafo “esperar por” global é gerado combinando todos os grafos locais; o conjunto de nodos é a união de nodos no grafo local, e há uma conexão de um nodo a outro se existir dessa forma uma extremidade em qualquer dos grafos locais.

O segundo algoritmo, o hierárquico, agrupa *sites* através de uma hierarquia. Por exemplo, os *sites* podem ser agrupados por estado, então por países, e finalmente em um grupo que contém todos os *sites*. Todo nodo nesta hierarquia constrói um grafo de “esperar por” que detecta *deadlocks* envolvendo somente *sites* contidos neste grupo de nodos. Desse modo, todos os *sites* periodicamente (por exemplo, a cada 10 segundos) mandam seus grafos locais de “esperar por” para o *site* responsável pela construção do grafo para seu estado. Os *sites* que estiverem construindo grafos no nível de estado periodicamente (por exemplo, a cada minuto) mandam seus grafos de “esperar por” do estado para o *site* responsável pela

construção do grafo do país, e cada um desses, manda periodicamente seu grafo para o site responsável pela construção do grafo global.

O terceiro algoritmo é simples: se uma transação ficar esperando por mais tempo do que o tempo escolhido para *time-out*, ela é abortada. Apesar de que esse algoritmo pode causar muitos reinicios desnecessários, o custo da detecção do *deadlock* é obviamente baixo, e em uma base de dados distribuída heterogênea, se os *sites* participantes não puderem cooperar com a extensão de compartilhamento de seus grafos de “esperar por”, esta pode ser uma única saída.

6 DESENVOLVIMENTO DA BIBLIOTECA

Neste capítulo são apresentados os requisitos do problema a ser trabalhado, a especificação da biblioteca de transações distribuídas, juntamente com suas limitações e apresentação de um aplicativo exemplo para fins de demonstração.

6.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

Os requisitos identificados para este trabalho foram:

a) a biblioteca não deve permitir que mais de uma transação execute ao mesmo tempo (isolamento, pela serialização da execução);
b) a biblioteca, se utilizada corretamente, deve garantir que a consistência da base de dados seja mantida mesmo com a ocorrência de falhas (consistência);
c) deve ser garantido que ou uma transação executa todas as suas operações com sucesso ou nenhuma operação terá efeito (atomicidade);
d) deve ser garantido que depois que uma transação terminar suas tarefas com sucesso, seus resultados nunca serão perdidos (durabilidade).

6.2 ESPECIFICAÇÃO

Para a especificação da biblioteca de transações distribuídas foi utilizada a UML, através do diagrama de casos de uso, diagrama de classes e diagrama de seqüência. A ferramenta utilizada para a especificação foi o *Rational Rose/C++ Demo* versão 4.0.3. Estes diagramas serão apresentados a seguir.

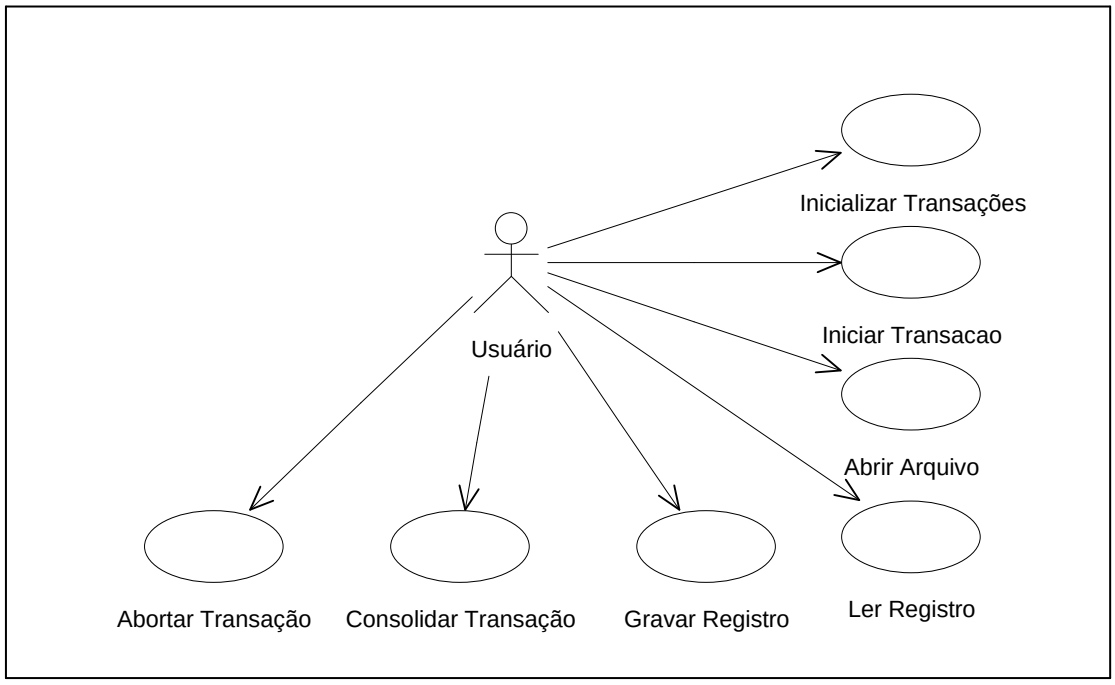
6.2.1 DIAGRAMA DE CASO DE USO

Na modelagem desta biblioteca foram observados sete casos de uso (fig. 14), que são descritos a seguir:

a) inicializar transações: o usuário (neste trabalho, este é o desenvolvedor da aplicação) inicializa a sua aplicação para o uso da biblioteca de transações;
b) iniciar transação: o usuário inicia uma transação onde ele realiza todas as operações que envolvem essa transação;

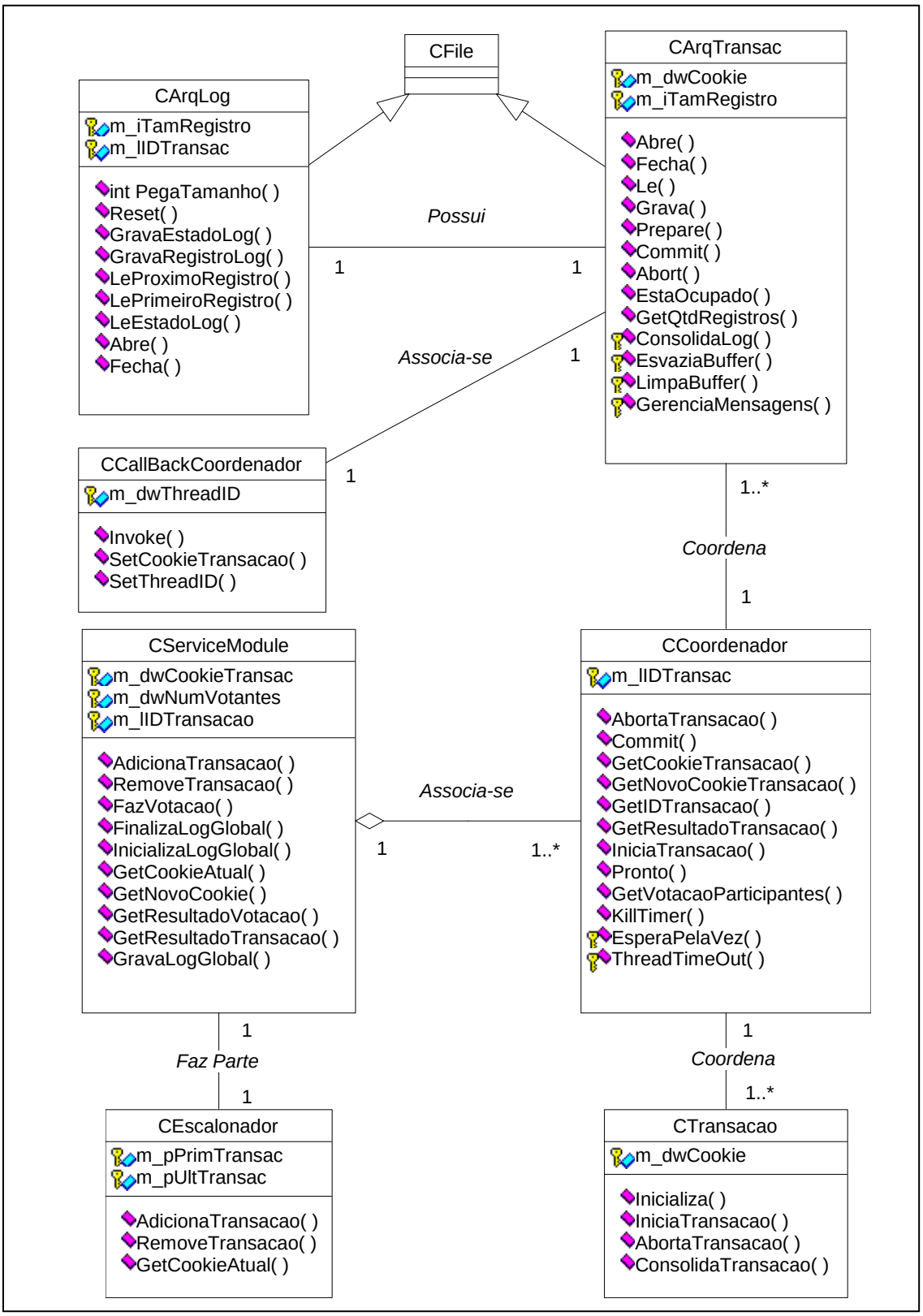
c) abrir arquivo: o usuário abre cada arquivo onde serão realizadas as operações da transação atual;
d) ler registro: o usuário executa uma operação de leitura de um registro de um determinado arquivo, o qual foi aberto anteriormente e faz parte da transação atual;
e) gravar registro: o usuário executa uma operação de gravação em um registro de um determinado arquivo, o qual foi aberto anteriormente e faz parte da transação atual;
f) consolidar transação: o usuário consolida todas as operações realizadas desde que ele iniciou essa transação;
g) abortar transação: o usuário aborta, ou seja, cancela todas as operações realizadas desde que ele iniciou essa transação.

FIGURA 14: DIAGRAMA DE CASO DE USO



6.2.2 DIAGRAMA DE CLASSES

FIGURA 15: DIAGRAMA DE CLASSES



As classes do diagrama da figura 15 são descritas a seguir:

a) CFile: é uma classe da biblioteca MFC (*Microsoft Foundation Class*) e é utilizada como classe base porque possui as operações primitivas de acesso, leitura e gravação de arquivos;
b) CArqLog: esta classe é derivada da classe CFile e possui as operações básicas de leitura e gravação de *log*, bem como do próprio estado do *log* (deve-se destacar, para o melhor entendimento do mecanismo, que o método de *log* utilizado é o de "Refazer") ;
c) CArqTransac: esta classe é derivada de CFile e é usada diretamente pelo usuário para realizar todas as operações da transação. Esta classe também encapsula toda a comunicação com o coordenador e faz o gerenciamento do *buffer* de operações e do *log* através da classe CArqLog (sua definição pode ser vista no ANEXO 3);
d) CCallBackCoordenador: esta classe é utilizada pela classe CArqTransac e serve para fazer as operações de *CallBack* entre ela e o coordenador. Portanto, ela é responsável pelo recebimento e tratamento de todas as notificações vindas do coordenador através do mecanismo de *Connection Points;*
e) CCoordenador: esta classe implementa a *interface* de comunicação do componente que coordena as transações, e é responsável por todo o gerenciamento das transações, como: Escalonamento das transações, controle de *time out*, protocolo de consolidação de duas fases (2PC), e gerenciamento do *log* global;
f) CServiceModule: é uma classe gerada pela ferramenta Visual C++ 6.0 através do ATL COM *AppWizard*, e é responsável pelo serviço de registro e comunicação do componente. Ela é utilizada pela classe CCoordenador na gerência de transações, pelo fato de ser criada em uma única instância, o que facilita no controle das diversas transações. Esta classe também possui mecanismos de bloqueio que permitem apenas uma transação acessar os dados de cada vez;
g) CEscalonador: utilizada pela classe CServiceModule, implementa os métodos necessários para gerenciar uma fila de requisições de início de transação. Ela garante que apenas uma transação execute de cada vez, e faz com que as transações adquiram seus recursos na mesma ordem que foram feitas as requisições;
h) CTransação: esta classe é utilizada diretamente pelo usuário, e é responsável pelas chamadas de início de transação, cancelamento e consolidação das mesmas, como

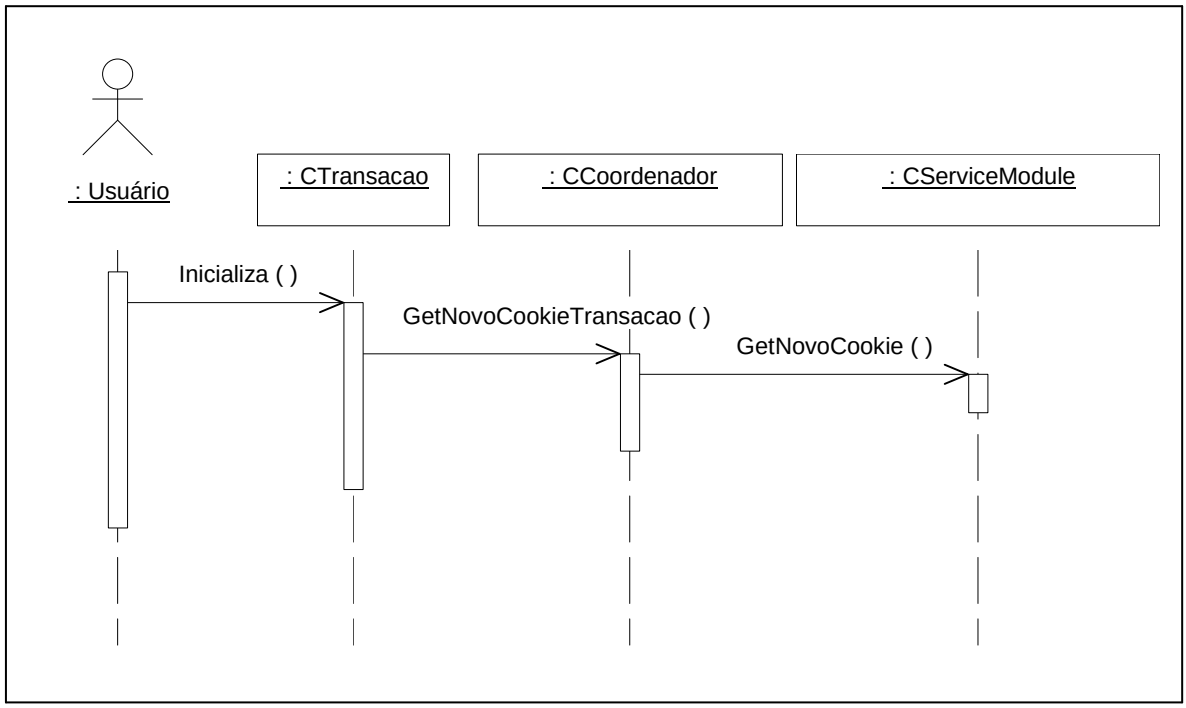
é visto no ANEXO 2. Ela também encapsula a sua comunicação com o componente coordenador.

6.2.3 DIAGRAMA DE SEQÜÊNCIA

Para cada caso de uso definido na figura 14, foi gerado um diagrama de seqüência correspondente, o que significa que foram desenvolvidos sete diagramas de seqüência, e que serão devidamente analisados.

O primeiro diagrama de seqüência identificado é o diagrama “inicializar transação” que será demonstrado na figura 16. Este diagrama é iniciado pela chamada do método “inicializa” do objeto da classe CTransação. Este cria uma instância do componente coordenador e solicita um identificador temporário da transação (*Cookie*) para identificar qual componente participante estará requisitando algo ao coordenador em um determinado instante. O coordenador, por sua vez requisita que o objeto da classe CServiceModule gere esse código e o retorne. É necessário que a geração deste identificador seja na classe CServiceModule porque esta classe possui uma única instância do objeto no componente, e não há perigo de ser gerado um identificador repetido para mais de uma transação ao mesmo tempo.

FIGURA 16: DIAGRAMA DE SEQÜÊNCIA – INICIALIZAR TRANSAÇÃO



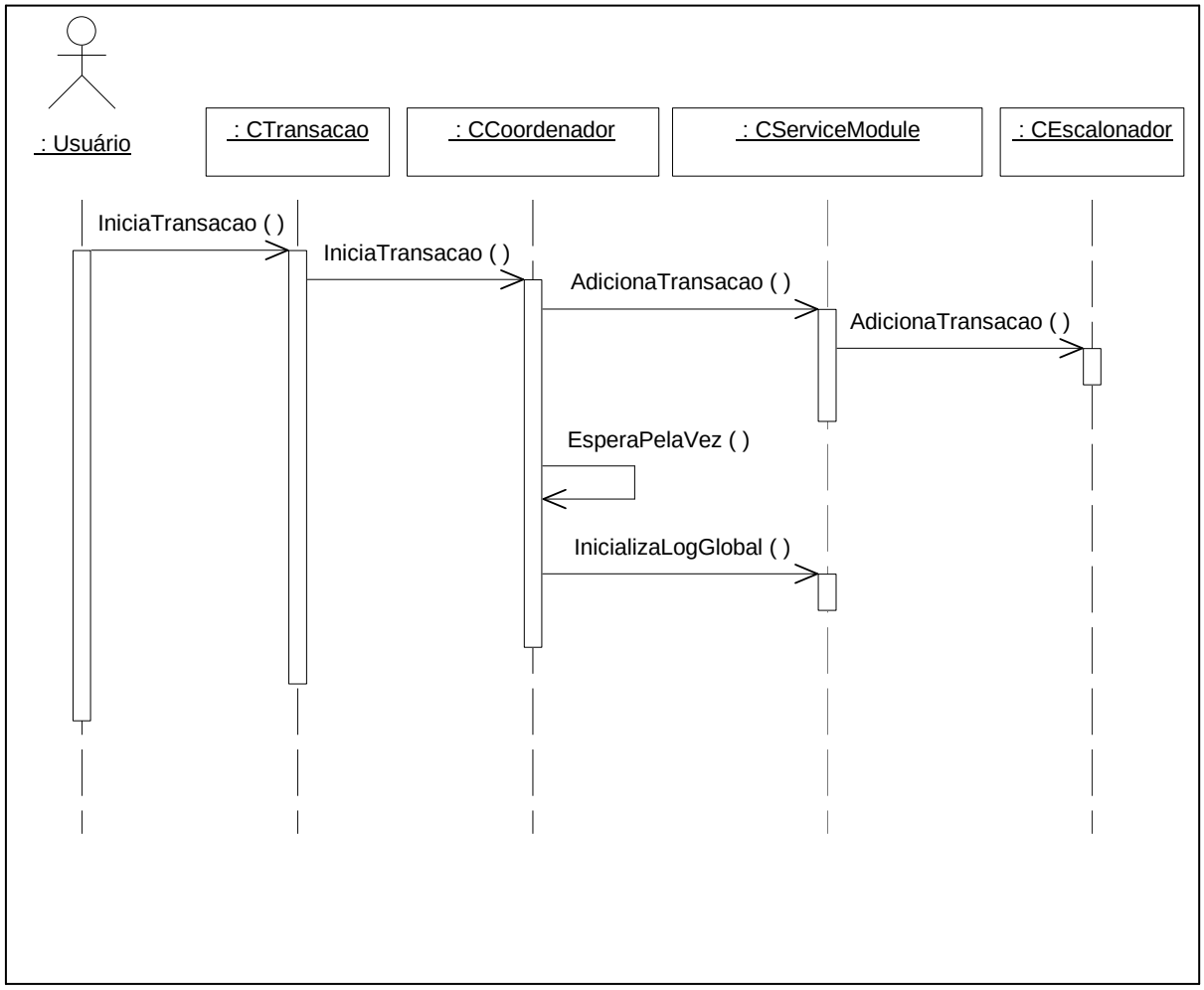
O segundo diagrama é o “iniciar transação” identificado na figura 17. Este processo faz com que a partir daí todas as operações realizadas utilizando-se da classe CArqTransac pertençam a essa transação. Depois de chamado o método “IniciaTransacao” do objeto da classe CTransacao, o mesmo notifica ao coordenador o pedido de início da transação, passando o *cookie* gerado na inicialização. O coordenador por sua vez requisita ao objeto da classe CServiceModule para adicionar esse pedido de início ao escalonador, que colocará o *cookie* em uma fila de espera.

Depois disso o coordenador não permite que a transação continue executando, a menos que o seu *cookie* seja o primeiro da fila. Caso não seja o primeiro da fila, a transação continua estática até que as transações anteriores terminem suas tarefas e retirem seus *cookies* da fila.

Quando a transação ganha a vez para executar, o componente coordenador gera um outro identificador para a transação através do método “InicializaLogGlobal” do objeto da classe CServiceModule, só que este nunca deverá ser repetido por outras transações, pois ele é gravado em todos os arquivos de *log* utilizados pela transação atual para fins de consulta posterior. Logo após, o coordenador solicita a inicialização do *log* global para

CServiceModule que gera um arquivo com a denominação do identificador gerado. Esse arquivo conterá qual foi a decisão global da transação, caso algum componente não consiga se comunicar com o coordenador (a implementação do método “IniciaTransação” do coordendor pode ser vista no ANEXO 4).

FIGURA 17: DIAGRAMA DE SEQÜÊNCIA – INICIAR TRANSAÇÃO



Depois de iniciada a transação, todas as operações a serem feitas se utilizarão da classe CArqTransac, e a primeira coisa a ser feita é a abertura do arquivo. O diagrama de seqüência “abrir arquivo” é ilustrado na figura 18.

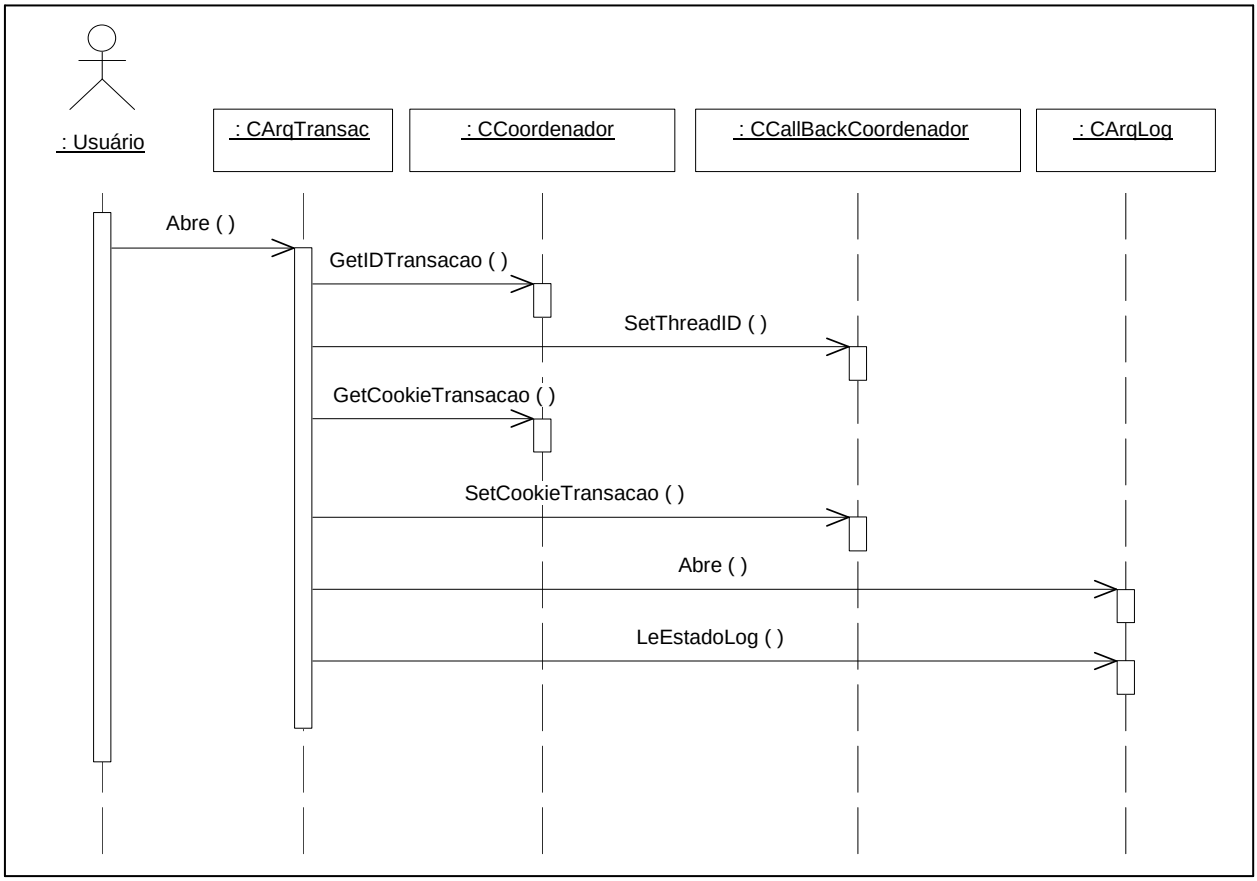
Depois do usuário ter solicitado a abertura do arquivo, é criada uma referência para a instância do coordenador e uma instância da classe CCallBackCoordenador no objeto da classe CArqTransac. Este requisita ao coordenador o identificador global da transação, para identificar a transação que está sendo executada.

Uma *thread* é criada na classe de arquivo de transação que fica monitorando as mensagens que chegam do objeto da classe de *CallBack*, e o identificador da *thread* deve ser passado para este objeto para que ele possa enviar as mensagens.

O *cookie* da transação atual é requisitado ao coordenador para que este seja passado como parâmetro nos métodos de notificação, e posteriormente o coordenador saiba de que transação partiu esta chamada. Este *cookie* também é passado para o objeto da classe de *CallBack* para que ele saiba se as notificações pertencem realmente a mesma transação (este *cookie* somente é necessário aqui se alguma transação foi abortada anteriormente por *timeout* mas ainda continuou executando e enviou uma resposta mais tarde).

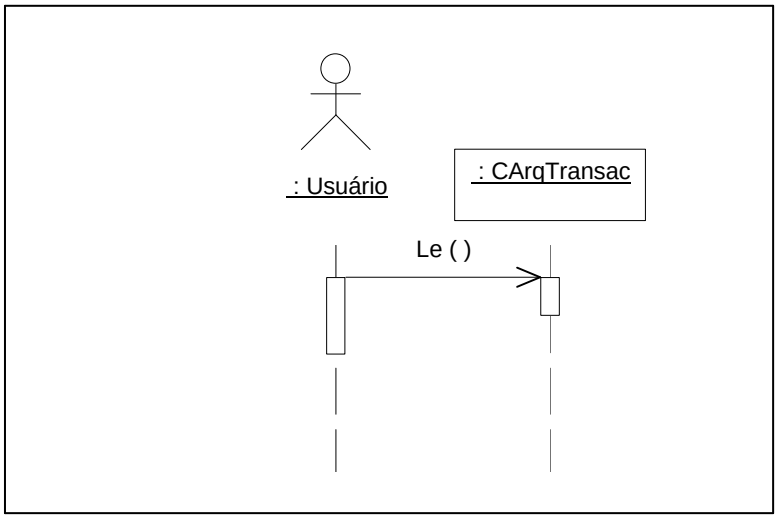
Depois de inicializar a *thread*, o arquivo de *log* correspondente a este arquivo também é aberto, para que possa ser utilizado no decorrer da transação. Após isto, é verificado o estado do arquivo de *log* para ver se existia algo pendente de alguma transação anterior e talvez deva ser consolidado.

FIGURA 18: DIAGRAMA DE SEQÜÊNCIA – ABRIR ARQUIVO



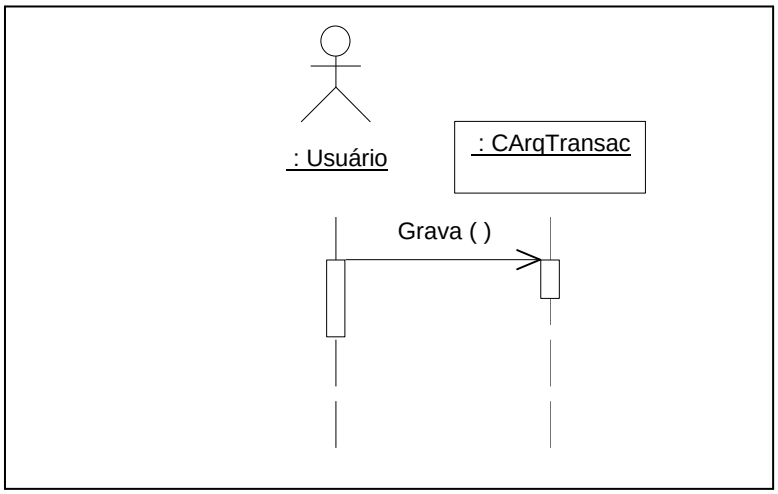
O quarto diagrama, chamado “ler registro”, é muito simples e somente faz a leitura de um registro especificado pelo usuário, como ilustra a figura 19. Não há a necessidade da execução de outros métodos ou da solicitação de bloqueios, pois a execução de uma transação é única e sem concorrência no coordenador de transações.

FIGURA 19: DIAGRAMA DE SEQÜÊNCIA – LER REGISTRO



O diagrama “gravar registro” também é muito simples e somente faz a gravação de um registro especificado pelo usuário como ilustra a figura 20. É importante ser observado que este registro não é gravado imediatamente em disco. Estes registros gravados ficam em um *buffer* de operações e permanecem lá até que o objeto da classe receba autorização do coordenador para consolidar esses dados em meio físico.

FIGURA 20: DIAGRAMA DE SEQÜÊNCIA – GRAVAR REGISTRO



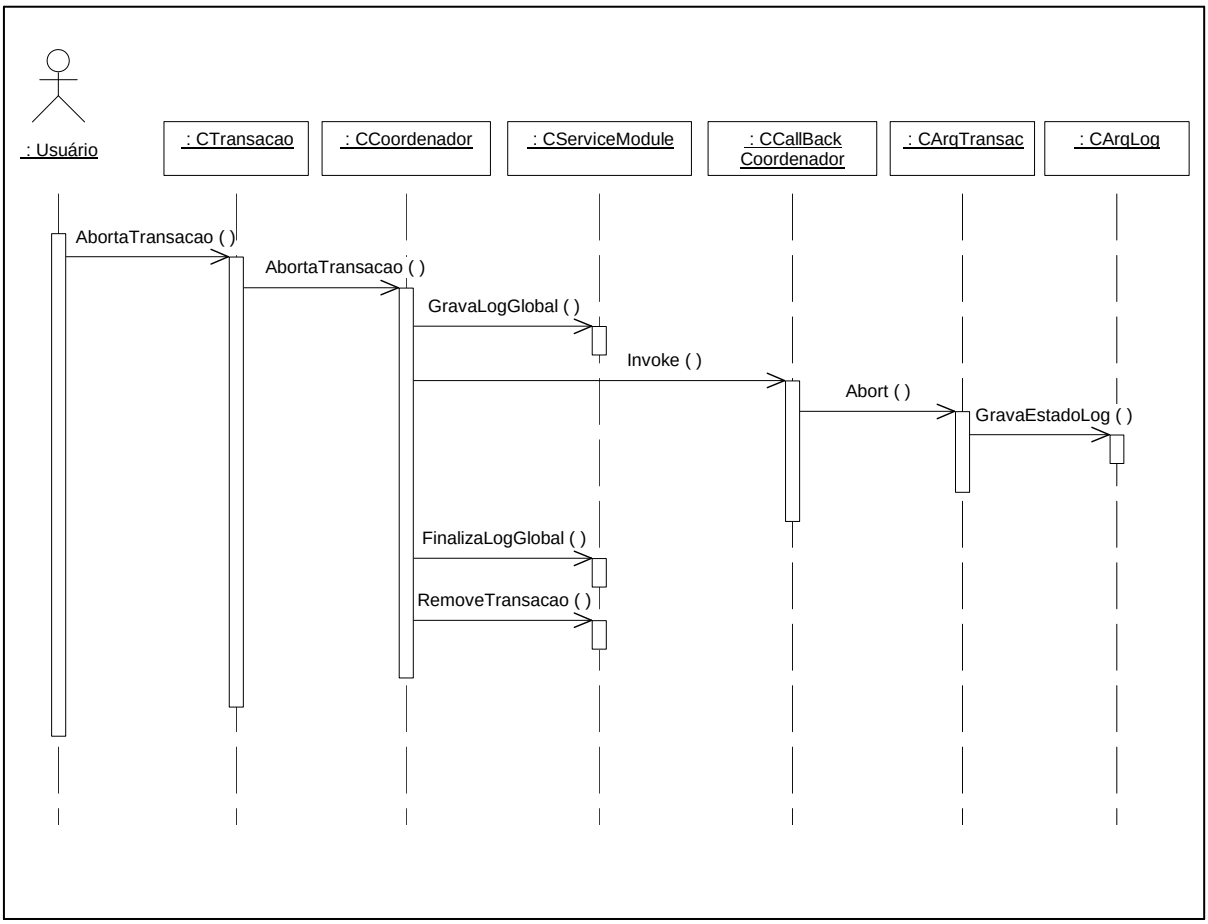
O sexto diagrama é “abortar transação” identificado na figura 21. Este processo é responsável pelo cancelamento de uma transação pelo usuário.

O usuário solicita a classe CTransacao para que esta aborte a transação corrente notificando o coordenador.

Quando o coordenador receber uma notificação de cancelamento, ele primeiramente grava o *log* global com o *status* de abortado, para que se acontecer algum problema de comunicação com os componentes participantes, eles possam futuramente verificar qual foi a decisão global. Após isso, o coordenador dispara uma notificação aos participantes informando que a transação foi abortada. A classe de *callback* no componente participante recebe esta notificação e dispara uma mensagem para o objeto da classe de arquivos, que a recebe através da *thread* receptora de mensagens. Após a *thread* identificar que a mensagem foi a de abortar, ela chama o método correspondente no objeto da classe que faz a gravação do estado do *log* atual e limpa o *buffer* de operações.

Depois do coordenador fazer a notificação aos participantes ele finaliza o arquivo de *log* global desta transação, e em seguida remove o *cookie* da transação da fila, para que a próxima transação que está esperando possa continuar sua execução (a implementação do método “AbortaTransação” da interface do coordenador é vista no ANEXO 6).

FIGURA 21: DIAGRAMA DE SEQÜÊNCIA – ABORTAR TRANSAÇÃO



O último diagrama é “consolidar transação”, que é ilustrado na figura 22. Sem dúvida este é o diagrama mais complexo e mais importante pois trata do processo mais importante de um gerenciador de transações, que é o processo de consolidação distribuída de transações.

O processo se inicia após o usuário terminar todos os procedimentos de sua transação e decidir consolidar efetivamente a transação.

Quando o coordenador recebe a decisão de consolidar a transação, ele inicia o processo de consolidação de duas fases (2PC) distribuído. Na primeira fase do protocolo o coordenador anota em seu *log* global o *status* de PREPARE. Após feito isso, ele envia a mensagem PREPARE para todos os componentes participantes. A partir desse momento, o coordenador fica na espera de resposta de todos os participantes, dentro de um tempo limite.

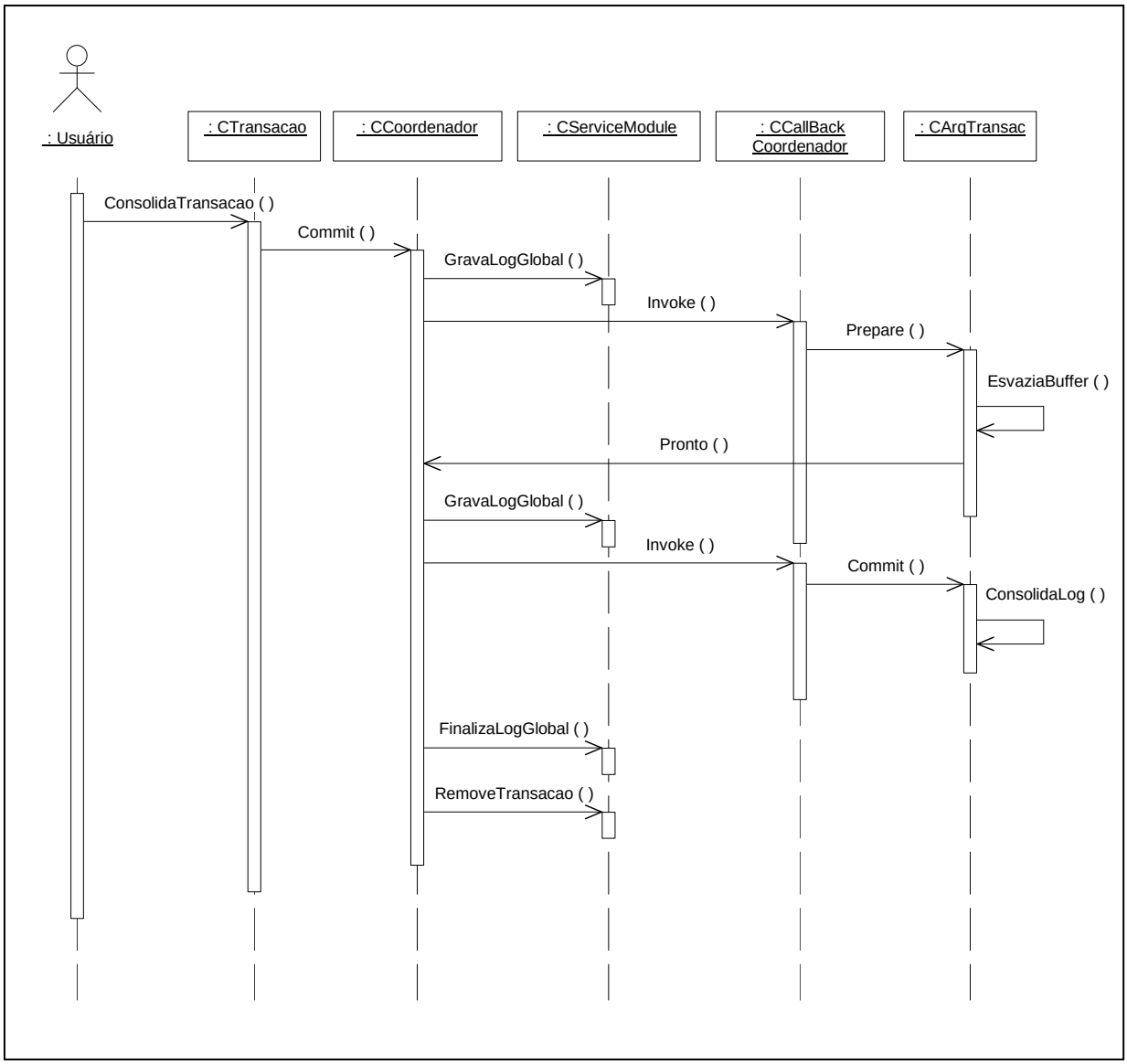
Quando um participante recebe a mensagem PREPARE ele esvazia o *buffer*, ou seja, grava todas as operações realizadas que estão no *buffer* de operações em *log*, e se tudo ocorrer bem ele responde ao coordenador através do método PRONTO.

Caso algum dos participantes não responder ou responder que não pode consolidar, os critérios adotados são os mesmos do diagrama da figura 21 (abortar transação).

Na segunda das duas fases do 2PL, caso tudo ocorra normalmente e todos respondam PRONTO, então o coordenador anota em seu *log* global o *status* de COMMIT e notifica todos os participantes enviando a mensagem COMMIT (a implementação do método "Commit" da interface do coordenador pode ser vista no ANEXO 5).

Quando um participante recebe a mensagem COMMIT, ele consolida o seu *log* varrendo todo o arquivo de *log* e fazendo suas alterações no arquivo de dados. Depois disso é gravado no *log* local o *status* COMMIT para que se tenha certeza de que a transação esteja efetivamente consolidada.

FIGURA 22: DIAGRAMA DE SEQÜÊNCIA – CONSOLIDAR TRANSAÇÃO



6.3 IMPLEMENTAÇÃO DA BIBLIOTECA

A implementação da biblioteca foi realizada através do ambiente de programação Visual C++ 6.0, onde foram utilizados conceitos de orientação a objetos para desenvolver as classes de gerenciamento de arquivo de dados e de *log*, bem como do coordenador de transações. Para desenvolver o componente coordenador das transações, foi necessário também utilizar o conceito de objetos distribuídos da Microsoft (COM/DCOM) para estabelecer a comunicação entre ele e os participantes das transações.

Para a gravação dos arquivos de dados e de *log*, foi utilizado uma classe base da biblioteca MFC (*Microsoft Foundation Class*) da Microsoft , chamada CFile, que permite efetuar operações básicas de manipulação de arquivos.

6.4 CONFIGURAÇÃO DE USO DA BIBLIOTECA

Para utilizar a biblioteca de transações distribuídas é necessário configurar em todos os computadores que farão parte das transações o local onde vai estar o componente coordenador, através do aplicativo DCOMCNFG da Microsoft. No computador onde ficará localizado o coordenador, registrar o componente EXE e a DLL para que o sistema de registros do sistema operacional desta máquina possa reconhecê-los.

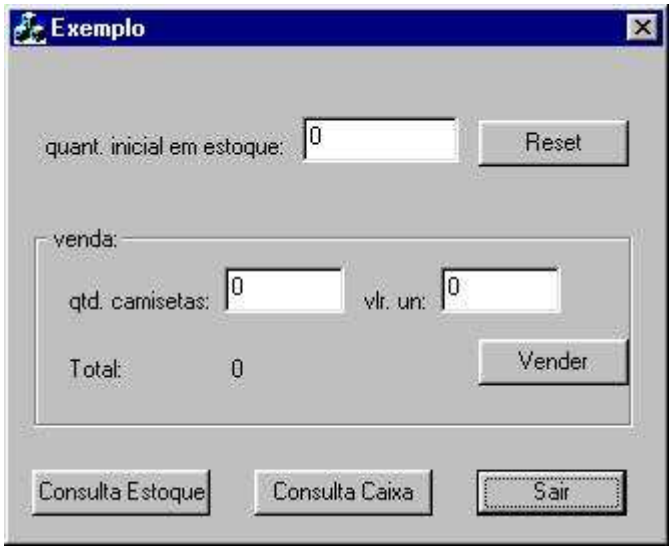
6.5 APLICATIVO EXEMPLO UTILIZANDO A BIBLIOTECA DE TRANSAÇÕES DISTRIBUÍDAS

6.5.1 ESTUDO DE CASO

Para fins de demonstração, foi elaborado um sistema fictício muito simples, que contém todos os passos necessários para se compreender o funcionamento da biblioteca de transações distribuídas.

Este sistema fictício consiste em controlar as vendas de uma loja de camisetas , onde é feito a retirada da mercadoria do estoque e a entrada de valores no caixa da empresa. O usuário deve informar a quantidade de camisetas vendidas e o valor unitário, e então efetuar a venda da mercadoria. Deverá ser informada a quantidade inicial de camisetas em estoque e o usuário poderá também fazer uma consulta do saldo do caixa e da quantidade de mercadorias em estoque, como mostra a figura 23.

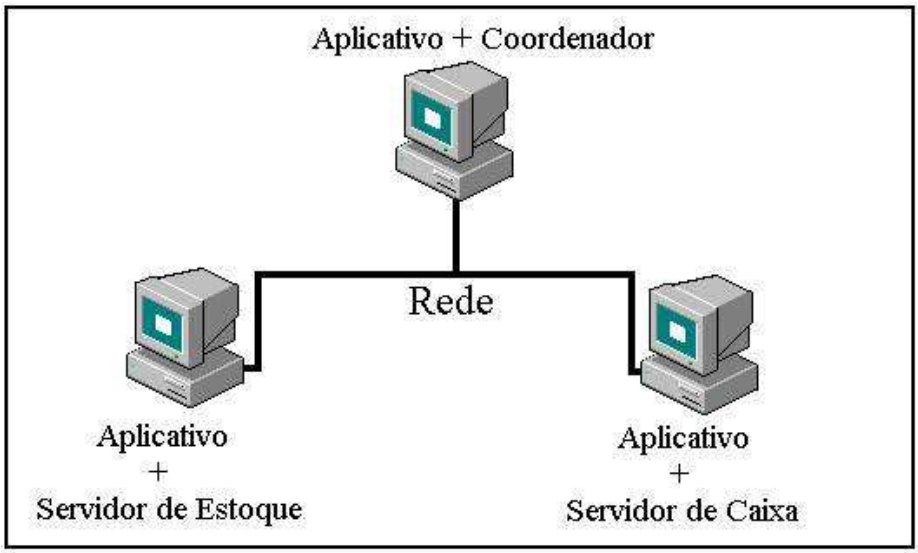
FIGURA 23: TELA DO APLICATIVO DO ESTUDO DE CASO



Deve ser lembrado que o estudo de caso existe somente para fins de demonstração da biblioteca de transações e não pode ser levado em consideração a forma de se resolver o problema por ele proposto.

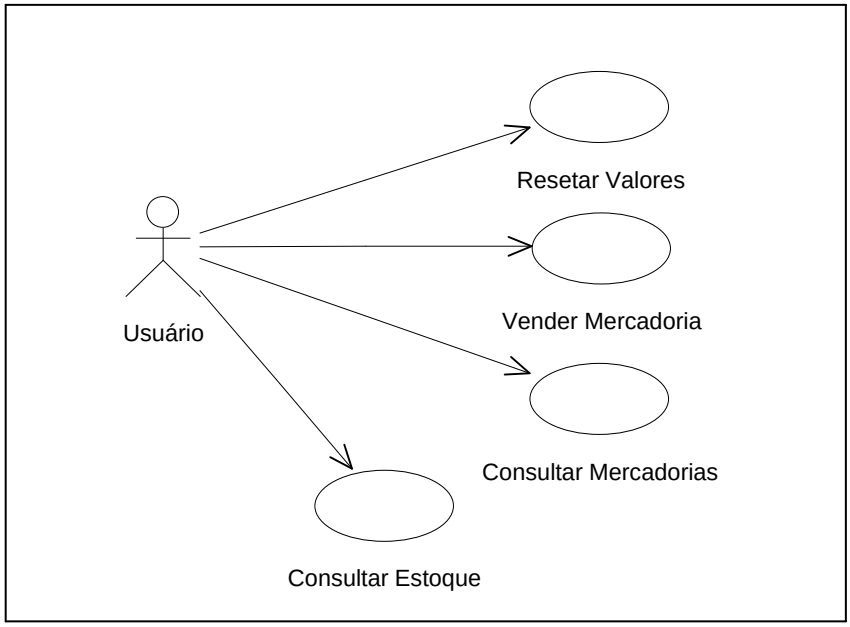
A figura 24 ilustra um exemplo de como a aplicação do estudo de caso pode ser utilizada em um ambiente distribuído.

FIGURA 24: EXEMPLO DA APLICAÇÃO DISTRIBUÍDA



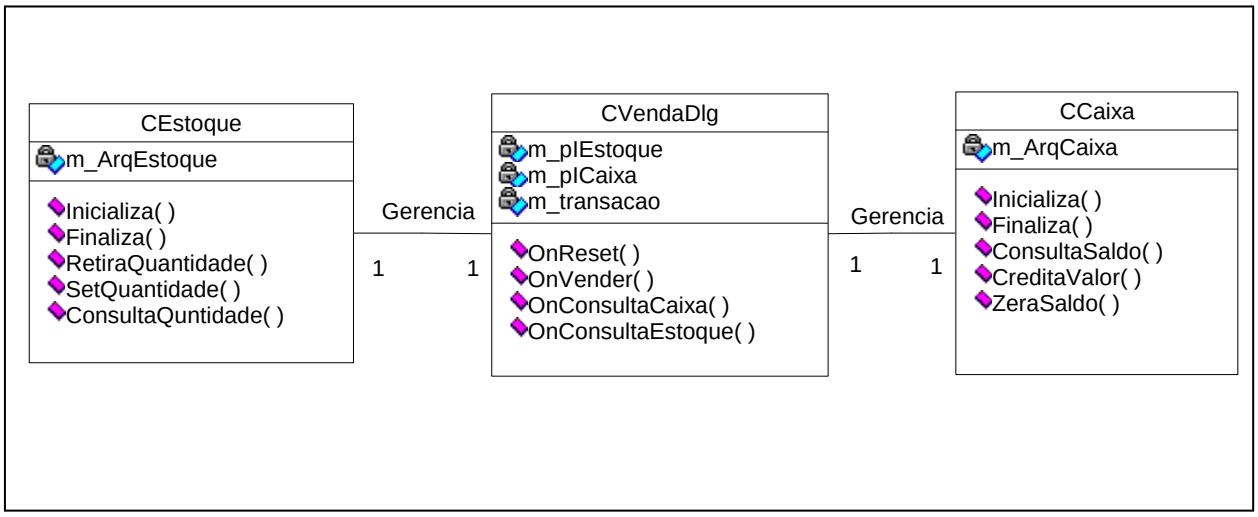
Na figura 25 é mostrado o diagrama de casos de uso modelado na ferramenta CASE *Rational Rose* através da especificação UML, para melhor apresentar o problema.

FIGURA 25: DIAGRAMA DE CASO DE USO DO ESTUDO DE CASO



Na figura 26 é mostrado o diagrama de classes, onde são encontradas as classes que se deseja implementar no sistema de estudo de caso, contendo somente as entidades pertinentes à transação de negócio.

FIGURA 26: DIAGRAMA DE CLASSES DO ESTUDO DE CASO



6.5.2 BIBLIOTECAS A SEREM INCLUÍDAS

Para o desenvolvedor da aplicação poder utilizar sem problemas os recursos do mecanismo de transações, ele deverá efetuar os seguintes procedimentos:

- Na aplicação principal, deverá ser incluído o arquivo *header* “transacao.h” para que possa ser utilizada a classe CTransacao;
- Nos componentes servidores participantes da transação, deverão ser incluídos os arquivos *header* “LibTransac.h” e a *static library* “LibTransac.lib” , para que possa ser utilizada a classe de manipulação de arquivo CArqTransac.

6.5.3 COMO UTILIZAR OS RECURSOS DE TRANSAÇÕES

A primeira coisa a ser feita é instanciar um objeto da classe CTransacao e em seguida inicializá-lo. Feito isso, deve-se instanciar todos os componentes servidores que farão parte da transação, como mostra o quadro 7. Estes componentes estarão conectados aos servidores remotos.

QUADRO 7: INICIALIZAÇÃO DOS COMPONENTES

```
CTransacao     m_transacao;

HRESULT hr = m_transacao.Inicializa();
if(FAILED(hr))
{
        MessageBox("sistema de transacoes não inicializou com sucesso.");
}

hr = m_pICaixa.CoCreateInstance(CLSID_Caixa);
if(FAILED(hr))
{
        MessageBox("falha ao inicializar servidor de caixa");
}

hr = m_pIEstoque.CoCreateInstance(CLSID_Estoque);
if(FAILED(hr))
{
        MessageBox("falha ao inicializar servidor de estoque");
}
```

Depois de feitas as inicializações, caso tudo ocorra sem erros, pode-se efetuar as operações das transações, como ilustra o quadro 8, que mostra a implementação do método OnVender, utilizando dois componentes servidores como sendo participantes da transação atual.

A inicialização dos componentes participantes deve ocorrer obrigatoriamente depois de iniciada a transação, pois é na inicialização do componente participante onde os arquivos serão abertos e estes arquivos já devem pertencer a alguma transação. Caso algum componente participante seja inicializado antes do inicio da transação, este componente estará apto a receber todas as mensagens de outras transações que poderão estar executando naquele momento.

Uma estrutura para a implementação de um componente servidor para usar o mecanismo de transações é demonstrada no quadro 9, que descreve as funções principais do servidor de estoque que foi utilizada na aplicação de demonstração. Portanto a maneira em que o arquivo é aberto tem influência no comportamento da transação e provavelmente fará com que a transação não funcione corretamente.

Foram realizados vários testes do aplicativo exemplo, interrompendo as operações em várias etapas diferentes da transação, e constatou-se que o mecanismo comportou-se conforme o esperado.

QUADRO 8: UTILIZANDO UMA TRANSAÇÃO

```
m_transacao.IniciaTransacao();
if(FAILED(hr))
    return;

hr = m_pICaixa->Inicializa();
if(FAILED(hr))
{
    m_transacao.AbortaTransacao();
    return;
}

hr = m_pIEstoque->Inicializa();
if(FAILED(hr))
{
    m_transacao.AbortaTransacao();
    return;
}

hr = m_pIEstoque->RetiraQuantidade(m_iQtdVendida);
if(FAILED(hr))
{
    m_transacao.AbortaTransacao();
    return;
}

hr = m_pICaixa->CreditaValor(lValorCaixa);
if(FAILED(hr))
{
    m_transacao.AbortaTransacao();
    return;
}

hr = m_transacao.ConsolidaTransacao();
if(SUCCEEDED(hr))
    MessageBox("Transação executada com sucesso");

m_pIEstoque->Finaliza();
m_pICaixa->Finaliza();
```

QUADRO 9: ESTRUTURA DE UM PARTICIPANTE DE UMA TRANSAÇÃO

```
STDMETHODIMP CEstoque::ConsultaQuantidade(long *plQuantidade)
{
    ...

    return S_OK;
}

STDMETHODIMP CEstoque::RetiraQuantidade(long lQuantidade)
{
     ...

    return S_OK;
}

STDMETHODIMP CEstoque::SetQuantidade(long lQuantidade)
{
    ...

    return S_OK;
}

STDMETHODIMP CEstoque::Inicializa()
{
    ...

    iRet  =  m_ArqEstoque.Abre("c:\\dadostcc\\estoque.dat", sizeof(REG_ESTOQUE));
    if(iRet)
         return E_FAIL;

    return S_OK;
}

STDMETHODIMP CEstoque::Finaliza()
{
    m_ArqEstoque.Fecha();
    return S_OK;
}
```

7 CONCLUSÕES

Através do desenvolvimento deste trabalho foi possível adquirir conhecimentos sobre o funcionamento de sistemas de processamento de transações e constatou-se que muito ainda pode ser explorado, pois o número de aplicações utilizando transações é muito grande fazendo com que a sua utilidade seja indispensável.

Para gerenciar transações em um ambiente distribuído, o sistema gera um considerável número de troca de mensagens, e nota-se que o uso excessivo ou impróprio de transações pode fazer com que a performance do sistema seja seriamente afetada, pois existem muitos casos onde não é realmente necessário utilizar processamento de transações.

A utilização de modelos de objetos para gerenciar transações distribuídas é extremamente viável, principalmente pelo baixo custo de desenvolvimento, considerando que o desenvolvedor não precisa se preocupar com a operacionalidade da comunicação dos processos através da rede.

O modelo de objetos COM da Microsoft nos fornece uma arquitetura que atende a maior parte das necessidades no desenvolvimento de um mecanismo de transações distribuídas, porém, pelo fato de ser um padrão proprietário e binário, faz com que a performance do mecanismo fique diretamente relacionada com a performance da arquitetura implementada pela Microsoft.

Os objetivos desejados neste trabalho foram alcançados, e constatou-se através de testes, que a funcionalidade do mecanismo de transações atende suas propriedades básicas, e fornece ao usuário uma maior segurança na execução de seus processos. Conseguiu-se também dar um passo inicial, para o estudo do comportamento de objetos distribuídos no uso de transações, que é de grande importância.

7.1 LIMITAÇÕES

Somente pode ser executada uma transação de cada vez, mesmo que estas se utilizem de recursos independentes.

As transações somente poderão ser executadas se o coordenador estiver ativo, não elegendo algum outro local como coordenador.

Este mecanismo é dependente dos sistemas operacionais da Microsoft e do ambiente de programação Visual C++.

7.2 EXTENSÕES

Como sugestão para continuação deste trabalho inclui-se:

a) implementar um mecanismo que permita executar mais de uma transação ao mesmo tempo, caso estas se utilizem de recursos independentes;
b) implementar um mecanismo de transações deste tipo utilizando-se do padrão CORBA de desenvolvimento de objetos distribuídos, e desenvolvê-lo de tal forma que possa ser utilizado independente de plataforma e linguagem de programação;
c) pesquisar e desenvolver algoritmos avançados para resolução de impasses como tempo limite de transações e *deadlock.*

ANEXO 1 – ARQUIVO IDL DO COORDENADOR

```
import "oaidl.idl";
import "ocidl.idl";
	[
		object,
		uuid(1D3E861F-3628-4002-B839-D054062AB80F),

		helpstring("ICoordenador Interface"),
		pointer_default(unique)
	]

	interface ICoordenador : IUnknown
	{
		[helpstring("method IniciaTransacao")] HRESULT IniciaTransacao([in] DWORD dwCookie);

		[helpstring("method Commit")] HRESULT Commit([in] DWORD dwCookie);

		[helpstring("method AbortaTransacao")] HRESULT AbortaTransacao([in] DWORD dwCookie);

		[helpstring("method GetNovoCookieTransacao")] HRESULT GetNovoCookieTransacao([out] DWORD* pdwCookie);

		[helpstring("method Pronto")] HRESULT Pronto([in] boolean bPronto, [in] DWORD dwCookie);

		[helpstring("method GetIDTransacao")] HRESULT GetIDTransacao([out] unsigned long*  plIdTransac);

		[helpstring("method GetResultadoTransacao")] HRESULT GetResultadoTransacao([in] unsigned long  lIDTransac, [out] small* pResultado);

		[helpstring("method GetCookieTransacao")] HRESULT GetCookieTransacao([out] DWORD* pdwCookie);
	};

[
	uuid(1DC787B9-1824-4F3F-A406-1AFD4A38E7A0),
	version(1.0),
	helpstring("MTransac 1.0 Type Library")
]
```

```
library MTRANSACLib
{
	importlib("stdole32.tlb");
	importlib("stdole2.tlb");

	[
		uuid(581431E8-396B-403D-A2B9-0B477A4DE2AB),
		helpstring("_ICoordenadorEvents Interface")
	]
	dispinterface _ICoordenadorEvents
	{
		properties:
		methods:
		[id(1), helpstring("method Commit")] HRESULT Commit([in] int dwCookie);
		[id(2), helpstring("method Abortar")] HRESULT Abortar([in] int dwCookie);
		[id(3), helpstring("method Prepare")] HRESULT Prepare([in] int dwCookie);
	};

	[
		uuid(E0C4FF87-F115-4082-9CE4-D3D451190C19),
		helpstring("Coordenador Class")
	]
	coclass Coordenador
	{
		[default] interface ICoordenador;
		[default, source] dispinterface _ICoordenadorEvents;
	};
};
```

ANEXO 2 – IMPLEMENTAÇÃO DA CLASSE CTRANSACAO

```
class CTransacao
{
public:
	CTransacao()
	{
		m_dwCookie = 0;
		m_pICoordenador = NULL;
	}

	virtual ~CTransacao()
	{
		if(m_pICoordenador)
		{
			m_pICoordenador.Release();
		}
	}

	HRESULT Inicializa()
	{
		HRESULT hr = m_pICoordenador.CoCreateInstance(CLSID_Coordenador);
		if(FAILED(hr))
			return hr;

		hr = m_pICoordenador->GetNovoCookieTransacao(&m_dwCookie);
		if(FAILED(hr))
			return hr;
		return S_OK;
	}

	HRESULT IniciaTransacao()
	{
		HRESULT hr = m_pICoordenador->IniciaTransacao(m_dwCookie);
		if(FAILED(hr))
			return hr;
		return S_OK;
	}
```

```
        HRESULT AbortaTransacao()
        {
                HRESULT hr = m_pICoordenador->AbortaTransacao(m_dwCookie);
                if(FAILED(hr))
                        return hr;
                return S_OK;
        }

        HRESULT ConsolidaTransacao()
        {
                HRESULT hr = m_pICoordenador->Commit(m_dwCookie);
                if(FAILED(hr))
                        return hr;
                return S_OK;
        }

protected:
        CComPtr<ICoordenador> m_pICoordenador;
        DWORD m_dwCookie;

};
```

ANEXO 3 – DEFINIÇÃO DA CLASSE CARQTRANSAC

```
class CArqTransacImpl : public CFile
{
public:
        // metodo chamado pela thread para saber se alguma
        // tarefa ainda esta em andamento
        bool EstaOcupado();
        // pega a quantidade de registros do arquivo
        USHORT GetQtdRegistros();
        // metodo chamado na primeira fase da consolidacao 2PC
        int Prepare();
        // fecha o arquivo e finaliza os processos
        void Fecha();
        // le um registro do arquivo
        int Le(long lPos, void* pRegRetorno);
        // grava um registro no arquivo
        int Grava(long lPos, void* pRegistro);
        // abre o arquivo e faz todas as inicializações com o coordenador
        // bem como a consolidacao de alguma transacao que ficou pendente
        int Abre(CString sNomeArquivo, USHORT iTamanhoRegistro);
        // metodo chamado na segunda fase da consolidacao 2PC
        void Commit();
        // metodo chamado para abortar a transacao atual
        void Abort();

        CArqTransacImpl();
        virtual ~CArqTransacImpl();
protected:
        // limpa o buffer de operacoes realizadas
        void LimpaBuffer();
        // grava as operacoes que estao em buffer para o
        // arquivo de log
        int EsvaziaBuffer();
        // atualiza as operacoes que estao no log
        // para o arquivo de dados
        int ConsolidaLog();
```

```
        // ponteiro para a interface do coordenador
        CComPtr<ICoordenador> m_pICoordenador;
        // ponteiro para a interface da classe de CallBack
        CComObject<CCallBackCoordenador>* m_pICallback;

        // arquivo de log associado com o arquivo de dados
        CArqLog m_arqLog;
        // cookie da conexao - utilizado para fazer a desconexao
                DWORD  m_dwCookieConexao;
        // identificador da thread de monitoracao de mensagens
        DWORD  m_dwIDThread;

        // guarda o nome do arquivo que esta aberto
        CString m_sNomeArquivo;
        // flag para uso interno
        bool m_bAberto;
        // flag para uso interno
        bool m_bOcupado;

        // metodo estatico que gerencia as mensagens
        // eh chamado por uma thread
        static DWORD _stdcall GerenciaMensagens(void *p);

        // contem as operacoes realizadas ate o momento
        // antes da consolidacao
        CList<REG_OPERACAO, REG_OPERACAO&> m_pilhaOperacoes;
};
```

ANEXO 4 – IMPLEMENTAÇÃO DO MÉTODO INICIA TRANSAÇÃO DA INTERFACE DO COORDENADOR

```
STDMETHODIMP CCoordenador::IniciaTransacao(DWORD dwCookie)
{
	_Module.AdicionaTransacao(dwCookie);


	//////// fica esperando ate que a transacao anterior termine //////////
	//////////////////////
	EsperaPelaVez(dwCookie);
	//////////////////////
	////////////////////////////////////////////////////////////////////

	/// gera um identificador unico da transacao ///

	m_lIDTransac = 0;

	SYSTEMTIME systemtime;
	GetLocalTime(&systemtime);

	WORD wAno2d = systemtime.wYear - 2000;

	m_lIDTransac =  systemtime.wDay      << 27 | systemtime.wMonth << 23 |
			wAno2d                       << 17 | systemtime.wHour   << 12 |
			systemtime.wMinute <<  6  | systemtime.wSecond;

	int iRet = _Module.IncializaLogGlobal(m_lIDTransac);
	if(iRet)
		return E_FAIL;

	return S_OK;
}
```

ANEXO 5 – IMPLEMENTAÇÃO DO MÉTODO COMMIT DA INTERFACE DO COORDENADOR

```
STDMETHODIMP CCoordenador::Commit(DWORD dwCookie)
{
	/// notificar os participantes para consolidar
	if(_Module.GetCookieAtual() == dwCookie)
	{
		m_bKillTimer = true;

		// inserir registro de prepare T global
		if(_Module.GravaLogGlobal(ID_PREPARE))
		{
			// nao conseguiu gravar log global
			AbortaTransacao(dwCookie);
			return E_FAIL;
		}

		// envia a mensagem de prepare para todos
		HRESULT hr = Fire_Prepare(dwCookie);
		if(FAILED(hr))
		{
			/// nao conseguiu notificar os participantes
			AbortaTransacao(dwCookie);
			return E_FAIL;
		}

		if(GetVotacaoParticipantes() == true)
		{
			// todos responderam pronto...
			if(_Module.GravaLogGlobal(ID_COMMIT))
			{
				/// nao conseguiu gravar log global
				AbortaTransacao(dwCookie);
				return E_FAIL;
			}
```

```
                    Fire_Commit(dwCookie);
                }
                else
                {
                    AbortaTransacao(dwCookie);
                    return E_FAIL;
                }
            }
            else
            {
                return E_FAIL;
            }

            _Module.FinalizaLogGlobal();
            _Module.RemoveTransacao();

            return S_OK;
}
```

ANEXO 6 – IMPLEMENTAÇÃO DO MÉTODO ABORTA TRANSAÇÃO DA INTERFACE DO COORDENADOR

```
STDMETHODIMP CCoordenador::AbortaTransacao(DWORD dwCookie)
{
	/// notificar os participantes para Abortar
	if(_Module.GetCookieAtual() == dwCookie)
	{
		m_bKillTimer = true;

		_Module.GravaLogGlobal(ID_ABORT);

		/// nao importa o resultado de Fire_Abortar:
		/// a transacao ja foi abortada

		Fire_Abortar(dwCookie);

		_Module.FinalizaLogGlobal();
		_Module.RemoveTransacao();
	}
	else
	{
		return E_FAIL;
	}

	return S_OK;
}
```

REFERÊNCIAS BIBLIOGRÁFICAS

BERNSTEIN, Philip A.; HADZILACOS, Vassos; GOODMAN, Nathan.**Concurrency control and recovery in database systems.** Massachusetts: Addison-Wesley, 1987.

BERNSTEIN, Philip A.; NEWCOMER, Eric. **Principles of transaction processing.** Reading: Morgan Kaufmann Publishers, 1997.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML : guia do usuario** : o mais avançado tutorial sobre Unified Modeling Language (UML). Tradução Fábio Freitas. Rio de Janeiro: Campus, 2000.

DATE, C.J. **Banco de dados :** tópicos avançados. Tradução Newton Dias de Vasconcellos. Rio de Janeiro: Campus, 1988.

DATE, C. J. **Introdução a sistemas de banco de dados.** Tradução de Contexto Traduções. Rio de Janeiro: Campus, 1991.

DATE, C.J. **Introdução a sistemas de banco de dados.** Tradução Vandenberg Dantas de Souza. Rio de Janeiro: Campus, 2000.

GARCIA-MOLINA, Hector; ULLMAN, J.D; WIDOM, Jennifer. **Implementação de sistemas de banco de dados**. Tradução Vandenberg Dantas de Souza. Rio de Janeiro: Campus, 2001.

GERAGHTY, Ronan; JOYCE, Sean; MORIARTY, Tom; NOONE, Gary. **COM-CORBA interoperability**. New Jersey: A Simon & Schuster Company, 1999.

GRIMES, Dr. Richard. **Professional DCOM programming**. Canadá: Wrox Press, 1997.

LAMPSON, B.W; PAUL, M.; SIEGERT, H. J. **Distributed systems:** Architecture and implementtion. Berlin: Springer-Verlag, 1988.

RAMAKRISHNAM, Raghu, GEHRKE, Johannes. **Database management systems:** second edition. New York: McGraw-Hill, 2000.

RAYNAL, Michel; SINGHAL, Mukesh. Mastering agreement problems in distributed systems. **IEEE Software**, New York, v. 18, n. 4, p. 40-47, jul/ago. 2001.