Agentes no Gerenciamento de Transações Móveis

Giovanni Bogéa Viana

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Agentes no Gerenciamento de Transações Móveis

Giovanni Bogéa Viana

Abril de 2006

Banca Examinadora:

- Dra. Maria Beatriz Felgar de Toledo (Orientadora)
  IC-UNICAMP
- Dr. Ricardo de Oliveira Anido
  IC-UNICAMP
- Dr. Eleri Cardozo
  FEEC-UNICAMP
- Dr. Edmundo Roberto Mauro Madeira (Suplente)
  IC-UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecária: Maria Júlia Milani Rodrigues – CRB8a / 2116

Viana, Giovanni Bogéa

V654a Agentes no gerenciamento de transações móveis / Giovanni Bogéa Viana -- Campinas, [S.P. :s.n.], 2006.

Orientador : Maria Beatriz Felgar de Toledo

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

1. Computação móvel. 2. Agentes móveis (Software). 3. Sistema de transação (Sistemas de computação). I. Toledo, Maria Beatriz Felgar de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Agents in management of mobile transactions

Palavras-chave em inglês (Keywords): 1. Mobile computing. 2. Mobile agents (Computer software). 3. Transaction systems (Computer systems).

Área de concentração: Computação Móvel

Titulação: Mestre em Ciência da Computação

Banca examinadora: Profa. Dra. Maria Beatriz Felgar de Toledo (IC-UNICAMP)
Prof. Dr. Ricardo de Oliveira Anido (IC-UNICAMP)
Prof. Dr. Eleri Cardozo (FEEC-UNICAMP)
Prof. Dr. Edmundo Roberto Mauro Madeira (IC-UNICAMP)

Data da defesa: 28/04/2006

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 28 de abril de 2006, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Eleri Cardozo
FEEC / UNICAMP

Prof. Dr. Ricardo de Oliveira Anido
IC / UNICAMP

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

Agentes no Gerenciamento de Transações Móveis

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Giovanni Bogéa Viana e aprovada pela Banca Examinadora.

Campinas, 28 de abril de 2006.

Dra. Maria Beatriz Felgar de Toledo
(Orientadora)

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

Resumo

Esta dissertação apresenta um modelo de transações para ambientes de computação móvel que leva em conta o dinamismo e interatividade nesse ambiente. Para lidar com o dinamismo, tanto aplicações como o gerenciador da transação e os objetos participantes de uma transação são executados como agentes e podem se mover a critério da aplicação. As responsabilidades de adaptação ao dinamismo do ambiente são divididas entre aplicações e o sistema de apoio. O sistema monitora o ambiente e envia notificações às aplicações sobre variações no ambiente. As aplicações decidem sobre as políticas para se adaptar às mudanças. Para lidar com a interatividade, as operações de uma transação podem ser submetidas passo-a-passo e o usuário pode adotar as estratégias mais adequadas conforme as necessidades da aplicação e as mudanças no ambiente.

Abstract

This dissertation presents a transaction model for mobile computing environments that takes into account their dynamism and interactivity. To deal with dynamism, applications, transaction managers and participating objects are executed as agents which can move as commanded by the application. The responsabilities for adaptation are divided between the applications and the underlying system. The system monitors the environment and sends notifications to applications about variations in the environment. Applications decide about policies to adapt to changes. To deal with interactivity, operations of one transaction may be submitted step-by-step and the user is able to adopt the best strategies considering application requirements and changes in the environment.

Dedicatória

À DEUS.
Aos meus pais e irmão.

Agradecimentos

À minha mãe e irmão, pelo incentivo aos meus estudos.

Ao meu pai (*in memorium*), pela certeza de estar sempre ao meu lado.

À professora Maria Beatriz Felgar de Toledo, pela orientação cuidadosa durante todo o curso de Mestrado, e pelos conselhos e atenção dispensada no desenvolvimento desta dissertação.

Aos professores do Instituto de Computação, por seus ensinamentos.

Ao amigo Tarcísio Rocha, por diversas sugestões, paciência e disponibilidade na adaptação/entendimento do sistema base SGTA.

Sumário

Lista de Tabelas

Lista de Figuras

Lista de Siglas

ACID - Acrônimo para Atomicidade, Consistência, Isolamento e Durabilidade

AMT - Adaptable Mobile Transaction

CORBA - Common Object Request Broker Architecture

CTKT - Continuing Kangaroo Transaction

DII - Dynamic Invocation Interface

EA_k - Execution Alternatives (modelo AMT)

ED - Environment Descriptor (modelo AMT)

Gbps - Gigabits per Second

HOKT - HandOff Kangaroo Transaction

IDL - Interface Definition Language

IIOP - Internet Inter-ORB Protocol

JT - Joey Transaction

Kbps - Kilobits per Second

KT - Kangaroo Transaction

KTID - Kangaroo Transaction Identification

MASIF - Mobile Agent System Interoperability Facility

Mbps - Megabits per Second

OMG - Object Management Group

ORB - Object Request Broker

P2P - Peer-to-Peer

PDA - Personal Digital Assistant

RMI - Java Remote Method Invocation

SGTA - Sistema Gerenciador de Transações Adaptativas

SGTAM - Sistema Gerenciador de Transações para o Ambiente Móvel

SSL - Secure Socket Layer

T_{AMT} - Transação AMT

UCP - Unidade Central de Processamento

XML - Extensible Markup Language

Capítulo 1

Introdução

As seções desse capítulo darão uma introdução ao tema desta dissertação. Inicialmente, a motivação para o seu desenvolvimento será vista na seção 1.1. Logo após, a seção 1.2 mostra os objetivos a serem alcançados. Por último, a seção 1.3 apresenta a estrutura da dissertação.

1.1 Motivação

A presença cada vez maior de dispositivos portáteis com capacidade de comunicação sem fio faz com que os usuários passem a executar aplicações a qualquer momento e em qualquer lugar. No entanto, as dificuldades inerentes às tecnologias desse ambiente [1–4] devem ser enfrentadas. Entre elas estão as restrições de recursos do dispositivo móvel; e a variabilidade de banda, desconexões freqüentes e alta taxa de erros de transmissão do meio de comunicação sem fio. Essas restrições e a variabilidade na disponibilidade de recursos exigem novos enfoques para garantir a consistência de dados às aplicações que executam nesse ambiente.

Uma das primeiras abordagens foi esconder das aplicações esses problemas [5]. Contudo, essa abordagem se mostrou inadequada para atender à diversidade de requisitos de aplicações. Outra alternativa é oferecer mecanismos de adaptação às restrições do ambiente de computação móvel deixando a cargo das aplicações a escolha das políticas de adaptação.

Esse modelo de adaptação em que a aplicação determina suas políticas de adaptação é

chamado de adaptação colaborativa e foi primeiramente proposto no projeto Odyssey [6]. Esse sistema monitora os recursos do ambiente e a aplicação especifica as políticas de adaptação a serem seguidas.

São, portanto, necessários novos modelos de transações que garantam consistência e, também, ofereçam apoio à adaptação colaborativa para atender aos requisitos de diversas aplicações.

1.2 Objetivos

O objetivo dessa dissertação é o desenvolvimento de um modelo de transações que atenda aos requisitos de aplicações no ambiente de computação móvel.

Para isso, é necessário incorporar ao modelo de transações a capacidade de adaptação às restrições de recursos e variabilidade na disponibilidade de recursos. Em especial, esse trabalho enfocará a capacidade de mobilidade de componentes como forma de adaptação.

Além disso, o modelo visa atender ao requisito de interatividade comum em aplicações no ambiente de interesse.

1.3 Estrutura

Esta dissertação está dividida em seis capítulos.

O capítulo 2 descreve os fundamentos da computação móvel, de agentes móveis e de transações necessários ao correto entendimento do trabalho desenvolvido. Ao final do capítulo, modelos de transações relacionados ao modelo proposto são apresentados.

O capítulo seguinte trata do sistema de transações SGTA utilizado como base no desenvolvimento do trabalho. Seus pontos básicos são discutidos assim como os seus mecanismos de adaptação e componentes utilizados na sua implementação.

O capítulo 4, Modelo de Transações Baseado em Agentes Móveis, apresenta uma visão geral do modelo de transações desenvolvido.

O próximo capítulo inicia com uma descrição do ambiente de execução que consiste das plataformas ORB para comunicação e Grasshopper para agentes móveis. Em seguida, discute a implementação do protótipo para o modelo de transações proposto e mostra exemplos de utilização.

Por último, o capítulo 6 apresenta os principais pontos do trabalho, suas contribuições e trabalhos futuros.

Capítulo 2

Fundamentos

Nesse capítulo serão apresentados os fundamentos necessários ao perfeito entendimento do trabalho de dissertação realizado: computação móvel em 2.1, agentes móveis em 2.2 e transações em 2.3.

2.1 Computação Móvel

A computação móvel é um paradigma que permite a computadores portáteis equipados com interfaces de comunicação sem fio participarem de uma computação distribuída mesmo em movimento. Isto se tornou possível graças aos avanços nas tecnologias de telecomunicação, redes e dispositivos de computação portáteis.

A Figura 2.1 mostra um modelo de arquitetura de apoio à computação móvel normalmente apresentado pela literatura [2,7,8]. Este modelo é composto por componentes fixos e componentes móveis (unidades móveis). Os componentes fixos da rede são máquinas fixas ou estações de apoio. Estes componentes se comunicam entre si através de uma rede tradicional de alta velocidade (isto é, com fio). Uma Unidade Móvel é um computador portátil capaz de se comunicar com a rede fixa através de um meio de comunicação sem fio. Uma Máquina Fixa é um computador que se comunica com o resto do sistema através de um ponto de conexão fixo. Uma Estação de Apoio é uma máquina fixa que tem também uma interface sem fio por meio da qual pode se conectar a unidades móveis. As estações de apoio agem como interfaces entre as unidades móveis e a rede fixa. Cada estação de apoio possui uma área de cobertura chamada Célula Sem Fio. Esta é a área

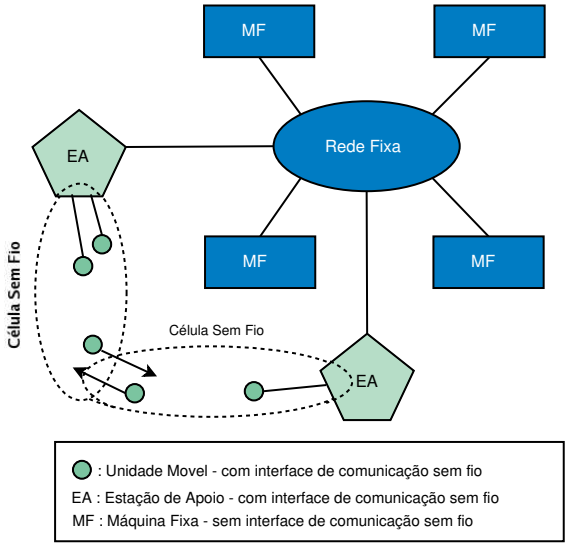

Figura 2.1: Modelo de arquitetura da computação móvel

por onde uma unidade móvel pode se mover mantendo a conexão com a estação de apoio correspondente. Ao se mover, uma unidade móvel poderá sair de uma célula e entrar em uma outra. Nesse caso, a comunicação com a rede fixa poderá ocorrer através de outra estação de apoio.

2.1.1 Problemas da Computação Móvel

A computação móvel exige várias mudanças no desenvolvimento de soluções no que diz respeito a circuitos integrados, processamento de sinais e projeto de sistemas computacionais. Essas mudanças são necessárias porque a comunicação sem fio enfrenta muito mais obstáculos do que a comunicação em redes tradicionais "com fios". Os autores em [1–4] citam alguns destes obstáculos:

Baixa largura de banda. Redes sem fio possuem uma largura de banda bem inferior a das redes com fio. Técnicas como *buffering* e compressão de dados têm sido usadas para reduzir os impactos da baixa largura de banda.

Desconexões freqüentes. A susceptibilidade a desconexões freqüentes das redes sem fio requer dos dispositivos móveis um certo nível de autonomia, ou seja, requer o poder de continuar a operar enquanto desconectados da rede fixa. Para tanto,

são usadas técnicas de operação assíncrona como busca antecipada[1] e escrita com atraso[2]. A busca antecipada é a ação de fazer uma cópia dos prováveis objetos que serão usados por um usuário em seu dispositivo móvel de forma que eles estejam disponíveis localmente quando a desconexão ocorrer. A escrita com atraso é uma técnica utilizada para atualizar os objetos remotos da rede fixa aproveitando os momentos em que haja largura de banda adequada para a transferência.

Escassez de energia. O suprimento de energia dos dispositivos móveis normalmente é feito por baterias que possuem uma capacidade limitada de suprimento. Isto exige técnicas para a redução do consumo de energia por parte dos elementos de hardware e software.

Segurança. A segurança da comunicação sem fio é bem mais vulnerável do que na comunicação com fio, pois o sinal da comunicação sem fio poderá ser interceptado por outros dispositivos de captação intrusos. As soluções adotadas para lidar com este problema são a criptografia e a autenticação feitas por software ou hardware especializado.

Heterogeneidade das redes. Devido à sua mobilidade, o dispositivo de computação móvel poderá se conectar a redes distintas e heterogêneas. Isto poderá implicar em mudanças nas velocidades e nos protocolos de transmissão bem como em mudanças na configuração necessária para se adequar a cada novo ambiente.

Maior dinamismo dos dados. Informações que são consideradas estáticas para um computador estacionário poderão passar a ser dinâmicas para um computador móvel. Um exemplo deste dinamismo é o endereço de rede do computador móvel que muda sempre que o computador se conecta a uma rede diferente.

2.1.2 Adaptação

Além dos obstáculos citados na seção anterior, a computação móvel apresenta uma outra característica marcante: a alta e rápida variabilidade dos recursos do ambiente móvel. Um exemplo que pode ser citado é a largura de banda da conexão entre uma unidade móvel

[1] Do inglês, Prefetching
[2] Do inglês, Delayed Write-Back

e a rede fixa que poderá variar entre três estados básicos: a desconexão; a conexão com baixa largura de banda (conexão fraca); e a conexão com alta largura de banda (conexão forte). Um estado de conexão fraca existe quando a unidade móvel está conectada à rede fixa através de uma interface de comunicação sem fio. Já a conexão forte existe quando a unidade móvel estiver diretamente conectada à rede fixa com fio. Assim, a largura de banda disponível para a comunicação poderia então variar desde Kbps[3] até mesmo Mbps[4] ou Gbps[5].

Diante desta alta variabilidade dos recursos do ambiente móvel, mecanismos de adaptação às mudanças deverão ser providos de forma que os recursos disponíveis possam ser mais bem utilizados. Quando a disponibilidade de recursos muda, as aplicações afetadas podem desejar mudar características da sua execução de forma a exigir menos de um recurso limitado ou aproveitar melhor a abundância de um recurso. São três os paradigmas de adaptação [9]:

Adaptação *laissez-faire*. Cada aplicação na unidade móvel deverá individualmente prover meios de contornar os obstáculos do ambiente móvel. Esta abordagem apresenta como problema a falta de centralização no gerenciamento dos recursos compartilhados por diferentes aplicações podendo assim gerar conflitos entre elas. É encontrada em sistemas como o Eudora [10].

Adaptação transparente. Os obstáculos do ambiente móvel deverão ser contornados por um sistema de apoio. Neste caso, as aplicações não tomam nenhum conhecimento dos obstáculos do ambiente móvel nem mesmo das medidas que são tomadas para superá-los. Esta abordagem possui a seguinte característica: as decisões de adaptação tomadas pelo sistema são iguais para todas as aplicações às quais ele dá apoio sem respeitar as suas individualidades. Esta característica poderá se tornar um problema quando as aplicações possuem diferentes necessidades de adaptação. É exemplificada pelo Coda [5].

Adaptação colaborativa. Os obstáculos da ambiente móvel são contornados através da colaboração entre o sistema de apoio e as aplicações. Esta abordagem propõe

[3] Acrônimo para Kilobits per Second ou 10^3 bits por segundo
[4] Acrônimo para Megabits per Second ou 10^6 bits por segundo
[5] Acrônimo para Gigabits per Second ou 10^9 bits por segundo

que o sistema seja responsável por monitorar o ambiente móvel e por prover os mecanismos de adaptação enquanto que as aplicações ficam livres para especificar individualmente as suas políticas de adaptação. Um exemplo dessa abordagem pode ser visto no sistema Odyssey [6], discutido a seguir.

Odyssey

Odyssey [6] é um sistema de computação móvel que segue o modelo colaborativo. A aplicação é ciente do ambiente de execução e uma parceria entre a mesma e o sistema operacional oferece uma abordagem geral e ao mesmo tempo eficaz para o acesso de informações no ambiente móvel.

O Odyssey monitora recursos como largura de banda, UCP[6] e bateria, e interage com cada aplicação para melhor explorá-los. Por exemplo, se a largura de banda ficar baixa, Odyssey detecta a mudança e informa a aplicação, que poderia, caso fosse uma aplicação de vídeo, adaptar-se descartando alguns quadros de exibição.

Uma propriedade definida, nesse trabalho, é a fidelidade de dados. Fidelidade é o grau de similaridade entre o dado apresentado ao cliente e a cópia original armazenada no servidor. Essa propriedade possui muitas dimensões, entre elas, a dimensão de consistência. Outras dimensões incluem (para vídeo) taxa de exibição de quadros e qualidade das imagens.

Outra característica do Odyssey é que, apesar de alguns dispositivos móveis, como os PDAs tradicionais existentes, não permitirem a execução concorrente de aplicações, essa capacidade não pode ser descartada. Para oferecer o apoio para concorrência, um módulo de monitoramento de recursos é definido e centraliza o controle e coordenação dos recursos na máquina móvel. Recursos como a largura de banda são divididos de maneira justa entre as diversas aplicações.

O Odyssey considera que os tempos de deteção e tratamento de mudanças no ambiente são requisitos chave. Esse parâmetro, conhecido como *agilidade*, leva em conta que as mudanças podem ser grandes e rápidas, e somente um sistema altamente *ágil* pode se comportar de maneira adequada para esses casos. Nesse sentido, a *agilidade* pode ser definida como um parâmetro que determina o grau máximo com o qual o sistema móvel pode trabalhar de maneira adequada.

[6] Acrônimo para Unidade Central de Processamento

Uma aplicação é notificada sempre que ocorrerem mudanças significativas. Cada aplicação responde de maneira independente à notificação e determina a maneira correta de se adaptar. Essa divisão de responsabilidades tem como foco principal permitir a concorrência e diversidade entre as aplicações. A diversidade é atingida permitindo-se que a aplicação determine níveis de fidelidade que sejam adequados aos recursos disponíveis. A concorrência, fazendo com que o sistema mantenha o controle do monitoramento e arbitração dos recursos.

Nesse sistema, é possível ainda explorar um certo conhecimento específico a respeito dos dados de uma aplicação a fim de permitir um apoio adequado aos vários tipos de dados. Esse conhecimento é utilizado para, por exemplo, estimar custos e benefícios entre compactar dados em cache ou enviá-los de volta ao servidor.

O monitoramento de recursos é requisitado através de uma chamada `request` identificando o recurso a ser monitorado e uma janela de tolerância para disponibilidade (o nível de recursos aceitáveis para execução da aplicação). Se o nível atual não estiver dentro da janela especificada, a solicitação é negada e um novo pedido com novas especificações de recursos deve ser enviado.

Assim que uma janela de tolerância especificada tem seus limites violados, uma notificação é enviada à aplicação solicitante com a identificação do recurso em questão, o nível atual e o identificador da requisição. A aplicação então ajusta a fidelidade dos dados de acordo com sua política e envia uma nova requisição com os níveis atualizados para monitoramento.

2.2 Agentes Móveis

Agentes móveis são softwares criados com o objetivo de executar uma determinada tarefa e possuem capacidade de sofrer migração entre vários computadores. O agente está ciente do ambiente de execução e age sobre ele ao longo do tempo, com base no seu conhecimento. Possui oito características principais: reatividade (capacidade de reação às mudanças no ambiente), autonomia, orientação a um objetivo, continuidade (está sempre em execução), habilidade para comunicação, aprendizagem (muda seu comportamento de acordo com experiências passadas) e mobilidade [11].

São particularmente interessantes em ambientes de computação em que PDA[7]s e computadores pessoais não podem estar permanentemente ligados à rede, fornecendo uma solução aos problemas de desconexões freqüentes, das baixas taxas de transmissão de dados e das constantes trocas de endereços de rede que esse ambiente impõe. O agente móvel pode, por exemplo, ser utilizado por usuários de máquinas portáteis de onde o agente seria lançado em busca de informações pela rede. O agente pode executar sua tarefa independentemente, evitando troca de mensagens desnecessárias com o equipamento de onde foi lançado e permitindo até mesmo que o usuário se desconecte da rede e obtenha o resultado da sua busca na próxima conexão [12].

Outra vantagem relacionada à utilização de agentes móveis é a facilidade de implementação e testes das aplicações que usam esse tipo de paradigma. Tal facilidade é obtida porque os agentes ocultam dos usuários detalhes dos meios de comunicação e possíveis falhas que possam ocorrer na troca de mensagens pela rede. Os agentes possibilitam ainda que as aplicações tenham uma maior escalabilidade já que podem migrar para qualquer máquina de uma rede onde a execução seja mais apropriada [12].

No paradigma de agentes móveis, o cliente envia um subprograma ao servidor para execução de uma tarefa. Essa característica flexibiliza o rígido modelo cliente/servidor e a comunicação passa a ser semelhante a P2P[8] [13] em que não há uma definição fixa de máquinas clientes e servidoras e uma mesma máquina cliente de uma aplicação pode estar agindo como servidora em outra [12].

Um agente também pode ser utilizado para tirar vantagens da interação local com os objetos de interesse da aplicação. Ele pode percorrer todos os servidores onde os objetos estão localizados migrando de um a outro de maneira a continuar sua execução do ponto onde foi interrompida. Assim, troca de mensagens desnecessárias são evitadas e o acesso é sempre otimizado por se tratar de um acesso local.

Uma das mais sérias restrições à utilização de agentes diz respeito à segurança. Deve haver uma maneira de garantir que um determinado computador que envie um agente para execução seja realmente a máquina que diz ser. Uma máquina servidora não deve permitir que um agente qualquer execute instruções e operações sem algum tipo de controle. Assim, uma máquina servidora deve limitar acessos a recursos conforme a origem dos agentes, sendo esta restrição diretamente ligada à autenticação do usuário [14].

[7] Acrônimo para Personal Digital Assistant
[8] Acrônimo para Peer-to-Peer (em inglês)

Outra restrição ao uso dos agentes móveis diz respeito ao seu comportamento. Mesmo que exista a garantia de que o agente partiu de uma máquina confiável, esta máquina pode estar contaminada por agentes mal intencionados, cujo único objetivo seria a destruição dos dados dos usuários. Idealmente, cada máquina que esteja executando o sistema de agentes deveria ter uma maneira de verificar se as operações executadas pelos agentes têm como objetivo a colaboração benigna com alguma aplicação ou apenas a destruição/violação dos dados do usuário. Entretanto não é possível detetar a intenção de violação e os problemas relacionados à propagação de vírus não estão limitados à utilização de agentes [14].

Apesar de não existirem aplicações que não possam ser executadas sem o uso dos agentes móveis, as vantagens obtidas pela sua utilização são gratificantes. O uso dos agentes móveis tem mostrado que as aplicações executadas com o seu auxílio ganham em desempenho, independente das taxas de transmissão da rede. Essa característica, aliada à possibilidade de continuidade de execução do processo mesmo quando ocorre desconexão da rede, e a maior facilidade no desenvolvimento das aplicações estimulam o uso dos agentes móveis, mesmo quando levadas em consideração as suas restrições [12].

2.2.1 Agentes Móveis na Computação sem Fio

O trabalho de Spyrou et al. [15] descreve como os modelos *cliente/agente/servidor* [16] e *cliente/interceptador/servidor* [17] podem ser incrementados para permitir um arcabouço de configuração dinâmica de aplicações em que nenhuma pré-instalação é necessária. O trabalho mostra como agentes móveis tornam estes modelos mais leves, tolerantes às instabilidades da rede e adaptáveis às mudanças no ambiente [15].

No modelo *cliente/agente/servidor* um agente *estático* está sempre presente no servidor e é responsável pela comunicação entre o dispositivo móvel e o servidor, de maneira a otimizar a troca de dados com o cliente. Spyrou et al. [15] estendem o modelo fazendo com que o agente seja criado dinamicamente no cliente e enviado ao servidor (agente *móvel*) no momento da requisição. O agente é mantido no servidor somente durante o período de execução da aplicação. Como opção, o agente do lado do servidor pode ser *móvel* e acompanhar o cliente, flexibilizando ainda mais o modelo.

O modelo *cliente/interceptador/servidor* utiliza agentes *estáticos* tanto no lado cliente quanto servidor, de maneira a permitir que a comunicação do cliente para o servidor

também possa ser otimizada. Para este modelo, os autores [15] argumentam que ambos os agentes devem ser *móveis* e mantidos apenas durante a execução da aplicação. Ambos são criados no cliente dinamicamente e, igualmente ao modelo *cliente/agente/servidor*, o agente servidor é enviado ao servidor para tratar as requisições do cliente.

Agente móveis criam novos modelos computacionais para a computação móvel que permitem uma alta flexibilidade e incorporam as vantagens das plataformas de agentes. Essas plataformas são escritas na sua maioria em Java e podem ser executadas em sistemas heterogêneos. Além disso, fornecem execução segura de programas e reduzem a quantidade de código compilado. Agentes no servidor podem ainda migrar para componentes fixos seguindo o cliente móvel para manter-se perto do mesmo. De maneira geral, os agentes possuem papel importante ao participar dos arcabouços não somente como uma unidade de computação, mas como um mecanismo eficaz de comunicação sem fio [15].

2.3 Transações

Uma transação é uma unidade de execução que consulta e eventualmente atualiza vários itens de dados. Essas operações são delimitadas pelas operações `begin_transaction`[9] e `commit_transaction`[10] ou `abort_transaction`[11]. Em caso de falha, nenhuma das operações deve ter efeito mas, se for completada com sucesso, todas as operações são efetivadas atomicamente e a base de dados passa a ter um novo estado consistente. Por exemplo, a transferência de um valor X de uma conta corrente para poupança é uma operação única do ponto de vista do cliente e não deve haver débito em um conta sem correspondente crédito na poupança [18].

Para garantir a integridade de dados, quatro propriedades conhecidas como ACID devem ser garantidas:

Atomicidade. Ou todas as operações de uma transação são propriamente refletidas na base de dados ou nenhuma. É garantida por um gerenciador de transações.

Consistência. A execução de uma transação preserva a consistência dos dados. A base de dados sai de um estado consistente e passa para um novo estado consistente após

[9] Em português, Iniciar
[10] Em português, Efetivar
[11] Em português, Abortar ou Cancelar

a execução da transação. É responsabilidade do programador da transação.

Isolamento. Apesar de múltiplas transações poderem executar de maneira concorrente, cada transação não deve ter conhecimento das outras. Resultados intermediários não devem ser aparentes. É responsabilidade do componente de controle de concorrência do gerenciador de transações.

Durabilidade. Após a execução com sucesso de uma transação, as mudanças devem se tornar permanentes, mesmo em caso de falhas do sistema. É responsabilidade do componente de recuperação de falhas do gerenciador de transações.

Tendo como exemplo a transferência bancária, a propriedade de consistência de dados garante que a soma do valor na conta corrente A com a conta poupança B deve parmanecer inalterado após a execução da transação. A propriedade de atomicidade garante que, caso ocorra uma falha no sistema, a conta A não tenha o valor debitado sem que B receba o valor X. Estados intermediários inconsistentes não podem ser visíveis para outras transações em execução. Essa é a propriedade de isolamento. Já a durabilidade, para a transação completada com sucesso, assegura que todas as atualizações (débito e soma de valores) persistirão mesmo se houver uma falha no sistema.

Múltiplas transações podem executar concorrentemente, trazendo vantagens como o aumento de uso da UCP e dos dispositivos de entrada/saída. Outra vantagem é o menor tempo de resposta obtido, já que transações de curta duração não necessitam esperar o término de transações longas.

Quando várias transações são executadas de modo concorrente, a consistência pode ser violada mesmo que cada transação seja executada individualmente com correção. O mecanismo de controle de concorrência impede que uma transação interfira no processamento da outra. A execução serial de transações garantiria a consistência, no entanto, as mesmas devem ser executadas de forma concorrente e ainda assim devem obter o mesmo resultado que seria obtido se fossem executadas seqüencialmente em alguma ordem. Nesse caso as transações são ditas serializáveis[12].

Duas instruções I_i e I_j de transações T_i e T_j respectivamente conflitam se existir algum dado Q acessado por ambas e pelo menos uma delas atualiza Q [18]. Conflitos estabelecem

[12] Do inglês, Serializable

T_1	T_2
read(A)	
$A = A$ - 50	
	read(A)
	$temp = A$ * 0.1
	$A = A$ - $temp$
	write(A)
write(A)	read(B)
read(B)	
$B = B + 50$	
write(B)	
	$B = B +$ `temp`
	write(B)

Tabela 2.1: Execução de Transações - não serializável

uma relação de ordem entre transações. O mecanismo de controle de concorrência deve evitar ciclos nessas relações.

Como exemplo, considere a tabela 2.1 onde a transação T_1 transfere R$50,00 de uma conta A (saldo inicial R$1000,00) para uma conta B (saldo inicial R$2000,00) e T_2 onde 10% do valor da conta A é transferido para B. O resultado final obtido viola a consistência (soma de valores) entre as duas contas (A=R$950,00 e B=R$2100). Nesse exemplo, temos o conflito read/write sobre A estabelecendo a relação de precedência de T_1 sobre T_2 e o conflito read/write sobre B estabelecendo a relação de precedência de T_2 sobre T_1. Esses conflitos resultam em um ciclo na relação de precedência e, portanto, numa execução não serializável que deve ser evitada.

Uma execução concorrente serializável é mostrada na tabela 2.2. Nesse caso, temos o conflito sobre A com relação de precedência T_1 precede T_2 e o conflito sobre B com relação T_1 precede T_2. Portanto, não existe ciclo e essa execução é dita serializável.

2.3.1 Controle de Concorrência

Quando duas ou mais transações executam de maneira concorrente, suas operações são intercaladas o que pode levar à interferência de uma transação sobre outra. A inconsistência resultante dessa interferência independe da codificação correta da transação individual e é gerada unicamente pela concorrência de execução. O controle de concorrência é o res-

T_1	T_2
read(A)	
$A = A$ - 50	
write(A)	
	read(A)
	$temp = A * 0.1$
	$A = A$ - $temp$
read(B)	write(A)
$B = B + 50$	
write(B)	
	read(B)
	$B = B + temp$
	write(B)

Tabela 2.2: Execução de Transações - serializável

T_1	T_2
read(A)	
	read (A)
	write(A)
	commit
write(A)	
commit	

Tabela 2.3: Atualização Perdida

ponsável por evitar essas interferências [19].

Um primeiro tipo de interferência é chamado atualização perdida[13] ou escrita inconsistente. Ocorre sempre que duas transações desejam atualizar um item de dado e ambas lêem o valor anterior do item antes que alguma das duas transações escreva o novo valor. Um exemplo é mostrado na tabela 2.3, onde uma transação T_1 faz com que a atualização de um item de dado A realizada por outra transação T_2 seja perdida.

Um segundo problema, conhecido como leitura inconsistente[14], ocorre quando a transação T_2 consulta um item que foi atualizado pela transação T_1 (na tabela 2.4). Em caso de cancelamento de T_1, a transação T_2 terá lido um valor inconsistente.

O terceiro tipo de interferência, leitura que não pode ser repetida[15], ocorre quando

[13] Do inglês, Lost Update
[14] Do inglês, Dirty Read
[15] Do inglês, Non-repeatable Read

T_1	T_2
read(A)	
write(A)	
	read(A)
abort	

Tabela 2.4: Leitura Inconsistente

T_1	T_2
read(A)	
write(A)	
	read(A)
write(A)	
	read(A)

Tabela 2.5: Leitura que não pode ser repetida

uma transação faz várias consultas sobre um item que é atualizado por outra transação entre as consultas da primeira transação. Um exemplo é mostrado na tabela 2.5.

Um último caso é a leitura fantasma e acontece quando a inserção ou exclusão é executada por T_1 em um item pertencente a um conjunto de dados sendo acessados pela transação T_2. T_2, ao fazer uma nova leitura, acessa um item de dado que não existia.

Para evitar essas interferências, o mecanismo de controle de concorrência deve garantir a serialização de transações concorrentes. Um dos mecanismos mais utilizados é baseado em trancas[16]. Cada transação antes de efetuar acesso aos dados deve solicitar uma tranca sobre o mesmo (em modo de leitura ou escrita). A solicitação é avaliada pelo mecanismo de controle de concorrência que determina se a tranca pode ou não ser concedida.

Uma função de compatibilidade é definida de maneira a identificar quais operações podem ser executadas de maneira concorrente e garantir a serialização das transações. Operações de escrita entram em conflito com quaisquer outras operações, conforme mostrado na tabela 2.6. Um dado pode ser trancado a qualquer momento durante a execução da transação mas as trancas obtidas por uma transação só devem ser liberadas no final (o que evita o problema de cancelamentos em cascata [18]).

A utilização de trancas pode levar a situações indesejadas. Uma delas, chamada impasse[17], acontece, por exemplo, quando uma transação T_1 mantém uma tranca exclusiva

[16] Do inglês, Lock
[17] Do inglês, Deadlock

	Leitura	***Escrita***
Leitura	Pemitido	Negado
Escrita	Negado	Negado

Tabela 2.6: Função de Compatibilidade

(escrita) sobre um item de dado A e T_2 solicita uma tranca em modo compartilhado (leitura) também sobre A. Além disso, T_2 mantém uma tranca exclusiva sobre B e T_1 solicita uma tranca sobre esse item. Enquanto T_1 aguarda o término de T_2 para prosseguir, T_2 também aguarda T_1, gerando uma situação em que nenhuma das transações pode ser completada. Uma das transações terá que ser cancelada.

Outra situação indesejada é conhecida como espera indefinida[18] e acontece quando uma transação fica indefinidamente aguardando a liberação de uma tranca por outra transação. Considere uma transação T_1 que esteja aguardando para atualizar um objeto X que já foi trancado por T_2. Após o término de T_2, uma outra transação T_3 solicita também uma tranca de leitura sobre esse mesmo objeto que é imediatamente concedida (ver tabela 2.6). Outras transações podem indefinidamente obter trancas sobre o objeto, fazendo com que T_1 nunca possa ser executada. A solução é dar maior prioridade para quem está esperando mais tempo [18].

Protocolos de Tranca

O protocolo de tranca que garante a serialização de transações é o protocolo de duas fases. Cada transação emite suas solicitações e liberações de trancas em duas fases distintas. Uma prova de que o protocolo garante serialização pode ser encontrada em [18] e suas fases são definidas a seguir:

Fase de Expansão: Uma transação pode obter trancas, mas não pode liberar nenhuma.

Fase de Diminuição: Uma transação pode liberar trancas, mas não consegue obter novas trancas.

Outros métodos de controle de concorrência existentes são baseados em grafos, marcas de tempo[19], validação (também utilizado nesta dissertação, ver Modelo de Transações Adaptáveis - capítulo 3) e multiversão. São descritos em [18].

[18] Do inglês, Starvation
[19] Do inglês, Timestamp

2.3.2 Recuperação de Falhas

Qualquer sistema de transações está sujeito à ocorrência de falhas que podem levar a uma queda do mesmo. Quando há alguma falha de *hardware* ou *software*, o conteúdo armazenado em memória volátil pode ser perdido. Desde que a falha não afete o armazenamento não volátil, a informação em *logs* (armazenados em disco rígido) é usada para recuperação de falha e garantia de atomicidade. Outra falha grave é a perda do armazenamento não volátil. Nesse caso, o *log* deve ser replicado para garantir a recuperação [18].

2.4 Modelos de Transações

A seguir serão apresentados vários modelos de transações que incluem os diversos aspectos considerados nesta dissertação: mobilidade, adaptação e agentes. Outros modelos de transações para o ambiente de computação móvel são descritos em [20].

2.4.1 Transações e Mobilidade

O modelo Kangaroo [21] captura a idéia de movimento de transações fazendo com que o estado da transação se mova junto com a máquina móvel. É baseado em dois outros modelos de transações: transações globais [22, 23] e transações particionadas[20] [24], e não visa ser puramente ACID. É composto por três camadas: sistema origem, o agente de acesso aos dados (Data Access Agent - DAA) e a própria transação móvel.

O Sistema de Origem corresponde ao conjunto de serviços de informação oferecidos aos usuários. São acessados através do DAA. Cada estação base contém um DAA que encaminha as requisições do cliente para o computador estático contendo os dados necessários à requisição.

A função principal exercida pelo DAA é o gerenciamento da transação móvel e, sempre que o cliente troca de estação base, o DAA no novo destino recebe as informações de processamento da transação vindas da estação antiga. Esse componente do DAA é chamado de gerenciador de transação móvel (Mobile Transaction Manager - MTM) e suas responsabilidades incluem manter informações sobre as transações móveis em processamento pelo DAA, armazenando registros de recuperação e realizando pontos de verificação[21].

[20] Do inglês, Split-transactions
[21] Do inglês, Checkpoints

O DAA executa sobre um Sistema Global de Banco de Dados (SGloBD) e Sistemas de Gerenciamento de Banco de Dados (SGBD) residentes na rede fixa. Um SGloBD assume que os sistemas SGBD locais executem as funções de processamento da transação, incluindo recuperação e concorrência. A visão do DAA sobre o SGloBD é a mesma visão de um usuário em seu terminal na parte fixa do sistema. O SGloBD não tem conhecimento da natureza móvel das transações e o DAA não é responsável pela implementação de detalhes de cada transação.

Para facilitar o reinício de transações interrompidas (comum no ambiente móvel), partes da transação móvel são efetivadas durante a execução da transação. Essas efetivações liberam trancas obtidas e evitam o bloqueio de dados por um longo período de tempo, oferecendo maior disponibilidade de recursos.

O modelo é baseado em transações tradicionais executadas sob o controle de um SGBD. Transações globais executam em sistemas de bancos de dados múltiplos e são compostas por subtransações executadas como transações locais nos SGBD. Uma subtransação pode ser ainda uma outra transação global e as folhas dessa hierarquia são obrigatoriamente transações locais. Uma transação *Kangaroo* (KT)[22] funciona como uma transação global, com a propriedade de mobilidade.

Quando uma requisição de transação é feita pela unidade móvel à estação base associada, o DAA inicia uma transação móvel KT para processar o pedido e cria uma identificação da transação (KTID)[23]. Cada subtransação de KT representa uma unidade de execução da estação base e é chamada transação *Joey* (JT)[24]. A estação base inicial é responsável por criar a primeira JT para execução da transação. Quando a máquina móvel salta de uma célula para outra, o controle da KT muda para o DAA da nova estação base. O DAA da nova estação base cria uma outra JT e a transação original é particionada. A JT antiga, com a parte da transação já processada, é efetivada independentemente da nova JT. Uma falha em alguma JT pode fazer com que toda KT seja desfeita, desde que transações compensatórias[25] estejam disponíveis para serem aplicadas.

Existem dois modelos de processamento para transações Kangaroo:

- modo de compensação: nesse modo, a falha de uma JT provoca seu cancelamento

[22] Do inglês, Kangaroo Transaction - KT
[23] Do inglês, Kangaroo Transaction IDentification - KTID
[24] Do inglês, Joey Transaction - JT
[25] Transações que desfazem semanticamente uma transação já efetivada

e a compensação de outras JT's já efetivadas. Operar nesse modo requer que o usuário forneça transações de compensação a serem executadas em caso de falha.

- modo de particionamento: quando uma JT falha nenhuma nova subtransação é submetida para a KT correspondente. A decisão por efetivação de subtransações em execução são deixadas a cargo do SGBD. Subtransações já efetivadas não são desfeitas.

Em cada DAA, o MTM mantém, em um *log*, os registros para processamento de recuperação. A maioria dos registros estão relacionados ao estado da transação. Durante o processo de troca de célula, informações referentes ao log são forçadas para memória persistente. Um registro de troca de célula é armazenado (HOKT)[26] na estação origem e um outro de continuação da KT (CTKT)[27] é escrito na estação destino. Registros de início e término das transações *Joey* assim como das subtransações dentro de uma JT também são salvos. O registro de uma subtransação contém a indicação da transação compensatória (no modo compensatório), em caso da mesma necessitar ser desfeita.

2.4.2 Transações e Adaptação

Modelo de Transações Móveis Adaptáveis

O Modelo de Transações Móveis Adaptáveis (AMT)[28] [25] permite a definição de transações aninhadas abertas com diversas alternativas de execução associadas a um contexto específico. As propriedades de atomicidade e isolamento são relaxadas, e o modelo considera que aplicações necessitam estar cientes do ambiente de execução a fim de superar a variabilidade de infra-estruturas e disponibilidade de recursos comuns no ambiente móvel (ver seção 2.1.2).

A idéia geral é definir uma transação móvel adaptável (T_{AMT}) com diversas alternativas de execução associadas a um ambiente móvel particular. Para cada transação móvel, o usuário programador é responsável por criar alternativas de execução compatíveis com o ambiente.

[26] Do inglês, HandOff KT - HOKT
[27] Do inglês, Continuing KT - CTKT
[28] Do inglês, Adaptable Mobile Transaction - AMT

Uma T_{AMT} é composta por no mínimo uma alternativa de execução e alternativas de execução são todas semanticamente equivalentes. O sucesso de execução de uma delas representa a execução correta de uma T_{AMT}. Uma T_{AMT} contém ainda descritores de ambiente (ED)[29] que expressam o estado do ambiente móvel necessário para execução de cada alternativa. Quando a T_{AMT} é iniciada, o ambiente de execução é verificado e a alternativa de execução apropriada escolhida.

O conjunto de dimensões relevantes é específico de cada aplicação. Entre as dimensões já definidas podemos citar: estado da conexão (conectado ou não), largura de banda (alta, média ou baixa), custos de comunicação, bateria disponível, disponibilidade de cache, disponibilidade de memória persistente, capacidade de processamento (alta, média ou baixa) e tempo estimado de conexão (hh:mm:ss). Além disso, outras dimensões definidas pelo usuário podem ser criadas.

Cada alternativa de execução (EA_k)[30] é associada com um descritor de ambiente ED_k e pode ser executada de quatro maneiras diferentes: transação móvel criada por uma unidade móvel e inteiramente executada em computadores estáticos; transações iniciadas por uma máquina móvel ou fixa e inteiramente executada na máquina móvel; execução distribuída entre máquinas estáticas e móveis; e execuções distribuídas entre várias máquinas móveis.

Cada EA_k contém um conjunto de *transações componentes* t_{ki} que respeitam as propriedades ACID e podem ser tradicionais, distribuídas ou aninhadas. Transações compensatórias podem estar associadas às transações componentes. EAs e T_{AMT} são apenas unidades de coordenação, enquanto o acesso a dados é realizado pelas *transações componentes*. Em um sistema múltiplo de base de dados, *transações componentes* correspondem a transações locais participantes de transações globais.

Considerando as restrições do ambiente móvel, o modelo AMT relaxa a atomicidade em uma EA_k adotando a chamada *atomicidade semântica*: todas t_{ki}s definidas por uma EA_k são efetivadas se a EA_k é efetivada; todas t_{ki}s de uma EA_k são compensadas ou canceladas se a EA_k for cancelada. Uma T_{AMT} é corretamente executada se uma de suas EA_ks obtiver sucesso (semi-atomicidade [26]): a efetivação de uma T_{AMT} implica na efetivação de apenas uma EA_k e o cancelamento ou compensação de todas as outras *transações componentes* de outras EA_i; o cancelamento de uma T_{AMT} implica no cancelamento ou

[29] Do inglês, Environment Descriptor - ED
[30] Do inglês, Execution Alternatives - EA_k

compensação de todas as *transações componentes* da EA_k em execução. O isolamento é relaxado por se tratar de transações aninhadas abertas.

MobileTrans

MobileTrans [27] é um sistema de transações móveis que permite a especificação de políticas de transações. Tais políticas são separadas do código da aplicação e determinam como os dados são acessados e atualizados, e graus de consistência e atomicidade. Alternativas ao cancelamento da transação, como realizar uma nova tentativa de execução ou suspender a execução até a ocorrência de algum evento, são permitidas.

Nesse modelo, um nó da rede é capaz de acessar objetos disponibilizados por qualquer outro nó e pode ter papel de servidor (disponibiliza objetos a outros nós) ou cliente (acessa objetos de outros nós). O acesso aos objetos acontece somente através de transações, executadas através de um controle otimista de concorrência [28].

Para implementar uma transação, além de seu código, o desenvolvedor deve especificar uma política de transação, isto é, as condições em que a transação deve executar e os procedimentos a serem tomados caso essas condições não possam ser satisfeitas.

São pré-definidos seis atributos como políticas de transação:

- consistência: especifica regras de consistência que permitam utilizar versões desatualizadas de objetos;
- busca de dados: especifica se objetos devem ser pré-transferidos ou transferidos sob demanda;
- delegação: permite ou não a transferência de responsabilidade pela efetivação da transação a outros nós;
- atomicidade: permite que a transação efetive mesmo se alguns nós participantes não estiverem disponíveis;
- *caching*: especifica se versões alteradas dos objetos devem ser mantidas em *cache*;
- tratamento de falhas: especifica se a transação deve abortar ou se devem ser tomadas medidas alternativas em caso de falha.

O modelo é extensível permitindo a inclusão de novos atributos não previstos inicialmente.

Todo atributo possui um valor a ser especificado. Com relação a consistência, objetos são definidos como necessários, réplicas ou dispensáveis e, de acordo com a política da transação (consistência alta, média ou baixa), devem ser obrigatoriamente buscados na matriz, em réplicas (possivelmente desatualizadas) ou em *cache*. Para busca de réplicas e delegação de responsabilidade, nós aleatórios ou específicos podem ser definidos. A atomicidade pode ser definida como alta ou baixa e objetos como obrigatórios ou opcionais. No caso de nível alto de atomicidade, todos os objetos devem estar disponíveis no momento da efetivação. No caso de nível baixo, só objetos definidos como obrigatórios devem estar disponíveis. O atributo *caching* determina se os objetos devem ser armazenados localmente ou não. Com relação ao atributo falha devem ser especificados o tipo de falha (consistência, busca de objetos, delegação, atomicidade) e a medida a ser tomada (cancelamento da transação, nova tentativa de execução ou suspensão).

Para cada objeto existe um par de *proxies* (*proxy-in* e *proxy-out*). Quando um objeto ainda não replicado no cliente é solicitado pela primeira vez, o *proxy-out* é o responsável por interagir com a contraparte do objeto (*proxy-in*) para iniciar a replicação. Uma vez que os objetos estão replicados, o acesso passa a ser de maneira direta, sem redirecionamento à matriz do mesmo. Um pré-compilador adiciona códigos especiais para manipulação de réplicas de objetos.

As políticas são implementadas como um conjunto de regras. O desenvolvedor da aplicação é responsável por criar e iniciar a política para configurar uma transação. A especificação pode ocorrer de maneira declarativa (através da linguagem XML) ou programática (através da linguagem C# ou Java).

2.4.3 Transações e Agentes

Agentes no Processsamento de Transações Confiáveis

A confiabilidade de execução é obtida pela replicação dos agentes e uma execução dividida em uma seqüência de estágios. Em cada estágio, cópias dos agentes são realizadas em várias agências e um novo estágio é iniciado sempre que ocorrer um pedido de migração de agentes. Em cada estágio, um único agente é escolhido como líder e é responsável pela

execução da tarefa.

Através de um processo de monitoramento de agentes, sempre que ocorrer falha na execução do agente líder, um protocolo para eleição de um novo líder é iniciado e o agente com falha substituído. O novo líder consulta o último ponto de verificação e reinicia a execução. Quando a execução de um estágio termina, cópias do agente com o estado atual são enviadas para as agências no estágio seguinte.

Um repositório distribuído de informação existe para cada estágio de processamento e é formado por repositórios locais de cada agência. O repositório é utilizado para armazenar pontos de verificação[31] durante a execução de um estágio e também durante a migração para um nova máquina. Em caso de falhas, o último ponto armazenado pelo agente anteriormente em execução ou o ponto armazenado na migração é utilizado e o processamento restabelecido a partir do mesmo.

O modelo permite que um grupo de agentes crie novos grupos (filhos) para execução de subatividades de maneira assíncrona ao grupo pai. No momento da criação, um ponto de verificação do grupo pai é estabelecido, e, caso ocorram falhas do grupo pai, o grupo filho pode continuar sua execução com a garantia que o pai irá se recuperar. No entanto, uma mudança de estágio só é permitida quando todas as operações do grupo filho tiverem sido completadas.

Durante a execução de um estágio por um grupo de agentes, vários eventos podem ocorrer e devem ser tratados apropriadamente:

- a eleição de um novo líder (*leader_ elected*): o último ponto de verificação é lido e a cópia local do agente eleito reinicia a execução como o novo líder;
- a ocorrência de falhas curtas (*short_ term_ failure*): o líder anterior se recupera de uma falha e continua sua execução;
- falhas longas (*long_ term_ failure*): o agente que se recupera após uma falha descobre que um novo líder o substituiu e desfaz sua execução ao ponto de verificação anterior;
- falha na recuperação (*processing_ long_ failure*): um agente falha quando está cancelando suas operações e deve continuar o cancelamento no próximo reinício.

[31] Do inglês, Checkpoint

O modelo de transações desse trabalho [29] é o modelo aninhado aberto. Uma transação é uma transação aninhada aberta composta por um conjunto de subtransações que são transações tradicionais ACID ou transações aninhadas abertas. Cada transação filha (ACID) deve executar inteiramente em uma agência. Um grupo de agentes inicia a transação raiz e executa as transações ACID que são subtransações da raiz. Para uma subtransação aberta um grupo de agentes filho é criado para executá-la. Portanto, uma transação raiz será executada por uma árvore de grupos de agentes em que cada grupo é responsável por uma transação aberta.

Agentes Móveis no Processamento de Transações

O modelo de transações em [30] considera o agente como um programa que executa métodos sobre objetos locais podendo migrar ao longo da transação. Neste modelo, antes de um agente iniciar o seu processamento, a disponibilidade de recursos como memória e poder computacional são verificados e possíveis incompatibilidades identificadas.

Após manipular objetos em um servidor D_i, um agente A se move para outro servidor D_j com intenção de manipular outros objetos. Se houver múltiplas cópias de um mesmo objeto, a quantidade de cópias acessadas depende da política especificada: para operações de leitura somente um objeto é acessado ou um quorum de leitura Q_r; para operações de escrita, todas as cópias ou um quorum de escrita Q_w, sendo $Q_r \cap Q_w \neq \emptyset$ e $Q_r \cup Q_w =$ Cand(A). Cand(A) é o conjunto de destinos possíveis do agente A.

Um agente pode ser composto por múltiplos módulos A_1, ..., A_m (m $>$ 1) que podem executar em diferentes objetos concorrentemente. Após manipular objetos nos servidores, os subagentes são novamente agrupados em um único agente.

Se um agente A executar com sucesso em um servidor D_i, um agente representante[32] A', clone do agente A, é criado. A fica responsável por manipular as trancas existentes nos objetos até que a transação complete, enquanto que A' parte para os outros servidores. Se não existir outros destinos, A inicia o protocolo de efetivação de duas fases que verifica a condição de efetivação para o agente A.

São quatro as condições de efetivação:

- efetivação atômica: o agente executa com sucesso em todos os computadores.

[32] Do inglês, Surrogate

- efetivação majoritária: o agente executa com sucesso na maioria dos computadores.
- efetivação pelo menos um: o agente executa com sucesso em pelo menos um dos computadores.
- efetivação $\begin{pmatrix} n \\ r \end{pmatrix}$: o agente executa com sucesso em mais de r dos n computadores.

Quando um agente A se move para um computador X onde já existe um agente (ou representante) B em conflito uma das quatro opções abaixo pode ser tomada:

- espera: A espera até B liberar o objeto (similar ao método de trancas).
- fuga: A procura outro computador onde possa manipular outros objetos antes de manipular os objetos no computador X.
- negociação: A negocia com B (B pode liberar objetos ou abortar).
- cancelamento: A aborta.

Para escolher uma dessas opções, os agentes levam em consideração sua política de efetivação:

- atômica: A espera B terminar.
- pelo menos um: se B sabe que algum dos seus representantes pode ser efetivado, libera objetos e aborta.
- maioria: se B sabe que a maioria dos seus representantes pode ser efetivada, libera objetos e aborta.
- $\begin{pmatrix} n \\ r \end{pmatrix}$: Se B sabe que pelo menos r podem ser efetivados, libera objetos e aborta.

Capítulo 3

Modelo de Transações Adaptáveis

Esse capítulo descreve o Sistema Gerenciador de Transações Adaptativas (SGTA) [31] que foi utilizado como base para o modelo de transações proposto nessa dissertação.

O SGTA segue o modelo de adaptação colaborativa. Neste modelo, um sistema de apoio é responsável por monitorar os recursos do ambiente (bateria, rede, entre outros) e notificar as transações sobre mudanças significativas, enquanto a aplicação é responsável por informar os recursos que deseja monitorar e decidir as medidas apropriadas de reação.

Dois parâmetros podem ser especificados no início da transação e modificados durante a execução da mesma como forma de adaptação: nível de isolamento e modo de operação. O conceito de níveis de isolamento foi inicialmente introduzido por [32] e tornou-se base para as definições do padrão ANSI/ISO SQL-92 [33]. Além do nível de seriabilidade, são providos outros três níveis de isolamento menos restritivos permitindo às transações ganhos em desempenho. O segundo parâmetro, modo de operação, permite que a transação opere sobre objetos remotos ou, então, sobre cópias locais.

Esse capítulo contém uma primeira seção que apresenta a arquitetura do SGTA. Em seguida, o diagrama de classes é mostrado em 3.2 e um exemplo de aplicação em 3.3. As seções seguintes tratam dos mecanismos de adaptação, políticas de adaptação, persistência dos objetos, replicação dos objetos no *cache* da máquina móvel, gerenciamento de transações e monitoramento de recursos. A seção 3.10 finaliza o capítulo apontando algumas diferenças entre o SGTA e o SGTAM desenvolvido nessa dissertação.

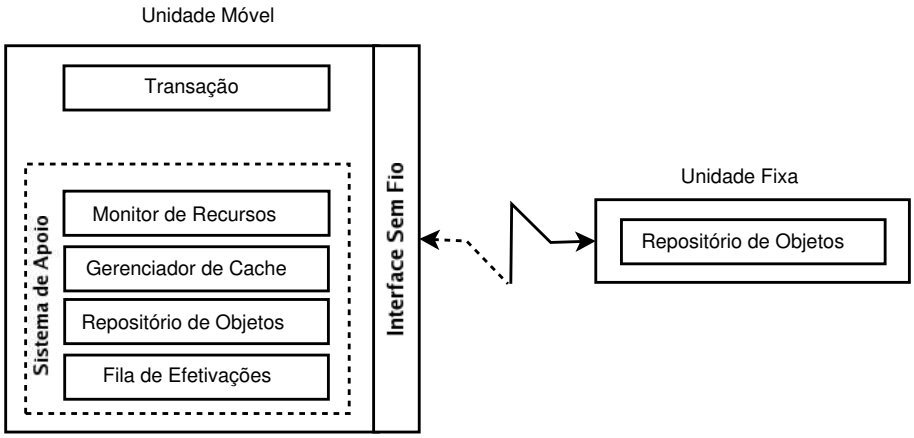

Figura 3.1: Arquitetura do SGTA

3.1 Arquitetura

O Sistema de Transações Adaptativas é composto basicamente de cinco módulos: Monitor de Recursos, Gerenciador de Cache, Repositório de Objetos (cliente e servidor), Fila de Efetivações e Transações. Destes, os quatro primeiros fazem parte do chamado sistema de apoio, enquanto o último trata da transação propriamente dita. A figura 3.1 mostra os módulos.

O Monitor de Recursos é a base do sistema de apoio. É ele que verifica a disponibilidade dos recursos físicos, recebe pedidos de monitoramento de aplicações e notifica aplicações a respeito de mudanças no ambiente. Numa requisição para monitoramento de recurso deve ser especificada uma janela de tolerância, em moldes semelhantes ao Odyssey (ver seção 2.1.2), com os valores mínimos e máximos aceitáveis pela aplicação. Caso estes sejam ultrapassados, o monitor envia a notificação para a transação que requisitou o monitoramento.

O Gerenciador de Cache é responsável por manter localmente as cópias dos objetos recentemente utilizados e por garantir sua consistência de acordo com as políticas estabelecidas.

O Repositório de Objetos é utilizado para armazenar em memória persistente os estados dos objetos.

A Fila de Efetivação executa o processo de validação e efetivação (se a transação for

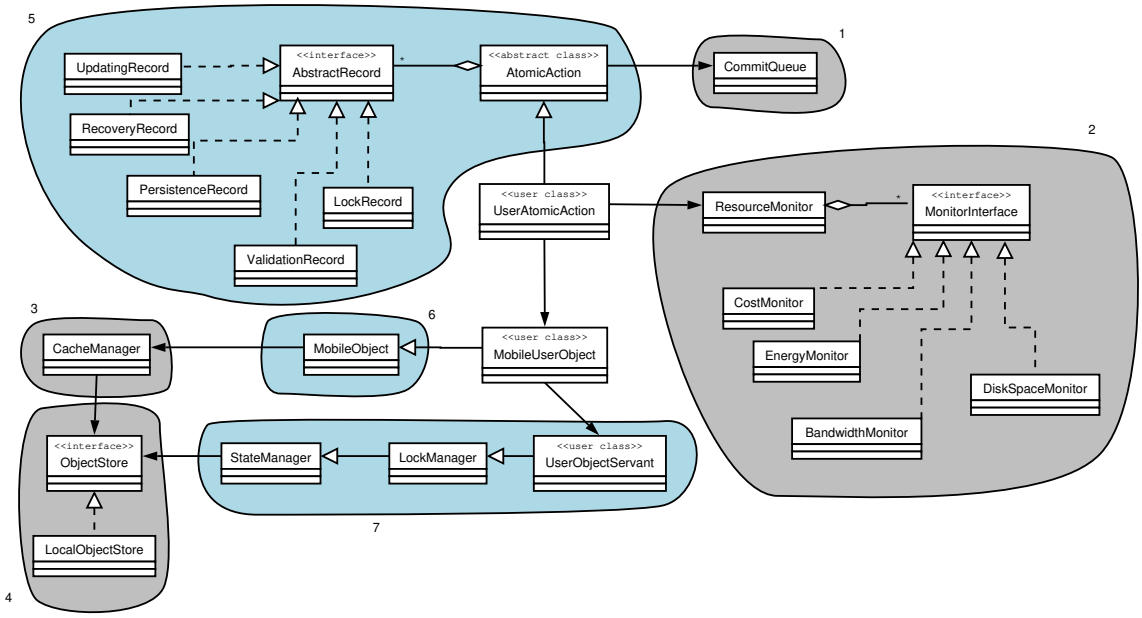

Figura 3.2: Diagrama de Classes do SGTA - unidade móvel

válida) para cada transação na fila.

3.2 Diagrama de Classes

O diagrama de classes do SGTA pode ser visto na figura 3.2 (módulos que executam na unidade móvel) e 3.3 (módulos que executam em unidades fixas). As classes relevantes são discutidas ao longo do capítulo. A cor cinza indica classe participante do sistema de apoio e a cor azul componente participante da infra-estrutura.

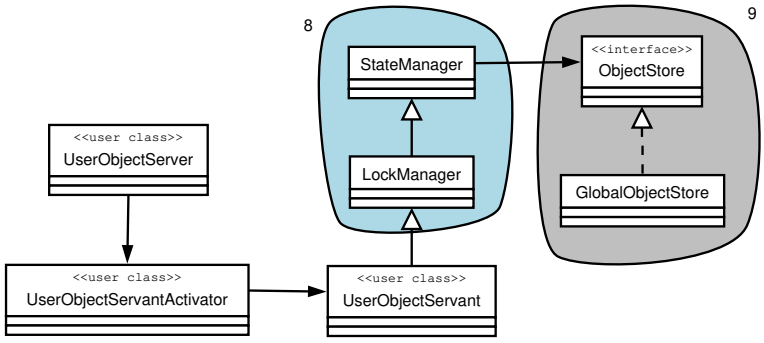

Figura 3.3: Diagrama de Classes do SGTA - unidade fixa

Os blocos nas figuras representam:

- bloco 1: Fila de Efetivações
- bloco 2: Monitor de Recursos
- bloco 3: Gerenciador de Cache
- bloco 4: Repositório de Objetos de unidades móveis
- bloco 5: Gerenciador de Transações
- bloco 6: MobileObject (superclasse de objetos definidos pelo usuário)
- bloco 7 e 8: Controle de Concorrência e Gerenciamento de Estados
- bloco 9: Repositório de Objetos de unidades fixas

3.3 Desenvolvendo uma Aplicação

Para implementar uma transação, o desenvolvedor deve definir uma nova classe que herda da classe abstrata `AtomicAction` (na figura 3.4). A classe `AtomicAction` contém as operações para gerenciamento de transações (`Begin`, `Commit`, `Abort`), as operações relacionadas com adaptação (setIsolationLevel, setOperationMode) e a operação abstrata `_notify`.

Cada nova transação deve ser definida por uma classe que herda da classe `AtomicAction` e implementa a operação abstrata `_notify`. Essa operação define uma política de adaptação específica da transação. A figura 3.5 mostra um exemplo de uma classe que pode ser usada para instanciar transações.

A figura 3.6 mostra um exemplo de uma aplicação desenvolvida como uma transação.

3.4 Mecanismos de Adaptação

O Sistema de Transações possui dois parâmetros de adaptação: nível de isolamento que controla as interferências entre transações concorrentes e modo de operação que determina se o acesso aos objetos será local ou remoto. Por se tratar de um sistema aberto, novas políticas de adaptação podem ser implementadas. Esses parâmetros serão discutidos a seguir.

```
1  abstract class AtomicAction{
2     int Begin();
3     int Begin(int isolationLevel, int operationMode);
4     int Commit();
5     int Abort();
6     ...
8     int setIsolationLevel(intisolationLevel)
9     int setOperationMode(int operationMode;)
10    ...
12    abstract void _notify (String requestId, String monitorId,
      int resourceLevel);
13 }
```

Figura 3.4: A classe abstrata AtomicAction

```
1  class MyAtomicAction extends AtomicAction {
2     ...
4     void _notify (String requestId, String monitorId, int resourceLevel)
      {
5       //Política de adaptação
6       if (monitorId == "BandwidthMonitor")
7         if (resourceLevel < 10000 ){
8           setOperationMode(LOCAL);
9         }
10      }
11    }
12    ...
14 }
```

Figura 3.5: Definição de uma transação

```
1  MyAtomicAction act; //Definição de uma transação
2  ObjectA objA;
3  ObjectB objB;
4  ...
6  act.Begin(SERIALIZABILITY, REMOTE);
7    resourceMonitor.request(act, resourceDescriptor);
8    objA.operation1();
9    objB.operation2();
10   ...
12   objA.operation3();
13 act.Commit(); // Finaliza a transação
```

Figura 3.6: Exemplo de aplicação

3.4.1 Nível de Isolamento

O método utilizado pelo SGTA para controle de concorrência é o de trancas e os níveis de isolamento são determinados através de trancas de curta ou longa duração. Uma tranca de curta duração deve ser adquirida no início da operação sobre o objeto e liberada após o término da mesma. Uma tranca longa só é liberada no final da transação.

Segundo o padrão ANSI SQL-92 [33], existem quatro níveis de isolamento desde nenhum isolamento entre transações até a execução serializável de transações. Esta característica garante uma grande flexibilidade aos usuários do sistema, que, de acordo com as necessidades da sua aplicação, especificam no início da transação um nível de isolamento adequado.

O primeiro nível de isolamento fornecido não evita as interferências discutidas na seção 2.3.1. Apenas trancas curtas são estabelecidas e somente na escrita de dados. O segundo nível impede a escrita inconsistente, aplicando trancas de longas duração na escrita. Ambos podem ser úteis para, por exemplo, obter índices estatísticos sobre a base de dados. O terceiro nível, COMMITTED_READ, impede a escrita e leitura inconsistentes. São aplicadas trancas de curta duração para leitura e de longa duração para escrita. O último nível é o único que garante a execução de transações de forma serializável. Trancas de longa duração devem ser obtidas tanto para escrita quanto para leitura. Esse nível tem como desvantagem a diminuição de concorrência entre transações.

3.4.2 Modo de Operação

O SGTA permite três modos de operação: remoto, local-remoto e local que definem se os objetos devem ser acessados no servidor remoto ou localmente no *cache* da máquina móvel. A escolha do modo de operação é também feita no início da transação e pode ser alterada a qualquer momento, a fim de permitir adaptação da transação. Uma classe `MobileObject` é responsável por garantir mudanças transparentes entre os modos.

No modo remoto, todas as operações sobre os objetos participantes da transação são redirecionadas para o respectivo objeto localizado remotamente na rede fixa. Este modo permite que os objetos acessados sempre estejam consistentes, uma vez que estarão sujeitos a um controle de concorrência global.

O modo remoto-local permite que as cópias dos objetos armazenadas em *cache* possam ser utilizadas pelas transações. Neste modo, trancas sobre os objetos remotos são obtidas, mantendo assim a consistência desses objetos com as cópias locais, e, ao fim da transação, as atualizações são propagadas para o servidor. Esse modo minimiza a comunicação com a rede fixa mas restringe a concorrência entre transações.

No modo local, são feitas cópias dos objetos na máquina móvel permitindo que transações locais sejam executadas mesmo desconectadas. Não há trancas sobre os objetos remotos. Assim, uma transação T_2 pode modificar o estado de um objeto remoto enquanto a transação T_1, executando em uma máquina móvel, modifica cópias locais. Ao final da transação T_1, um processo de validação verifica se houve também modificações nos objetos remotos. Se a transação passar na validação, as suas atualizações são propagadas para as cópias remotas. Caso contrário, será abortada.

3.5 Políticas de Adaptação

A política de adaptação é de total responsabilidade do programador. Assim, ao receber uma notificação sobre mudanças no ambiente, a aplicação pode tomar as medidas de adaptação mais adequadas aos seus requisitos. Uma política é especificada através da implementação de uma operação especial, `_notify`, definida na classe abstrata `AtomicAction` e implementada na transação. Esta operação é chamada automaticamente pelo Monitor de Recursos sempre que algum dos recursos monitorados ultrapassarem os limites definidos.

A notificação enviada indica qual o recurso teve o limite violado e o nível atual do

recurso. De acordo com o que o programador especificar na operação, um novo modo de operação ou um nível de isolamento diferente poderá ser solicitado. Na figura 3.5 temos um exemplo de definição de política de adaptação. A transação (na figura 3.6) inicia no modo de operação remoto acessando objetos remotos. Mas, ao ser notificada de que a largura de banda disponível está inferior ao limite tolerado, passa a executar sob o modo de operação local (a operação setOperationMode altera o modo para LOCAL na figura 3.5). A partir desse momento, são realizadas cópias dos objetos na máquina móvel e os acessos são locais.

3.6 Persistência de Objetos

Um objeto utilizado por transações está obrigatoriamente em um dos seguintes estados: ativo ou passivo. Um objeto ativo é a instância de uma classe cuja representação está em memória volátil. O objeto nesse estado está pronto para receber invocações de aplicações. Por sua vez, um objeto passivo mantém seu estado através da classe `ObjectState` que permite o armazenamento do estado do objeto em memória persistente.

Um repositório de objetos, instância da classe `ObjectStore`, é responsável por manter em memória não volátil o estado de objetos persistentes, armazenando e recuperando instâncias da classe `ObjectState`. Para cada objeto podem ser armazenadas duas versões: a *efetiva* e a *não efetiva*. A versão *efetiva* contém o estado resultante da última transação efetivada que atualizou o objeto. A versão *não efetiva* contém as atualizações de uma transação em execução. A versão *não efetiva* é armazenada na primeira fase do protocolo de efetivação. Na segunda fase, a versão *não efetiva* sobrepõe a versão *efetiva*.

3.7 Replicação

Com o intuito de aumentar a autonomia da máquina móvel, o SGTA realiza replicação de objetos no *cache* local. Políticas otimistas[1] e pessimistas[2] de controle de concorrência, adequando-se ao estado atual da rede, podem ser adotadas.

[1] Cópias desatualizadas podem ser acessadas. Os problemas de consistência são resolvidos no processo de validação.

[2] O objeto fica bloqueado no servidor à espera de liberação pela transação que tem tranca sobre ele.

No modo de operação local, o controle de réplicas é otimista e a verificação de consistência é realizada ao final da transação. A transação pode ser efetivada se os objetos participantes dessa transação não tiverem sido atualizados por outras transações. As cópias das atualizações são propagadas para a rede fixa se a transação passar na validação. O controle pessimista é utilizado nos modos remoto e local-remoto. Para o modo local-remoto, a tranca sobre objetos remotos garante a consistência dos objetos em *cache*.

Um gerenciador de *cache*, executando na máquina móvel, é uma instância da classe `CacheManager`. Ele é responsável por transferir estados de objetos de máquinas fixas para a máquina móvel (operação `loadState`) e vice-versa (operação `updateRemoteState`).

3.8 Gerenciamento de Transações

Transações no SGTA são instâncias de uma classe definida pelo programador. Essa classe deve herdar da classe `AtomicAction` que fornece as operações básicas de gerenciamento de transações (`Begin`, `Commit` e `Abort`), as operações para definir parâmetros de nível de isolamento e modo de operação (`setIsolationLevel` e `setOperationMode` respectivamente), e a operação abstrata `_notify`. Essa operação deve ser implementada na subclasse que herda de `AtomicAction`. A operação `_notify` define ações de adaptação a mudanças no ambiente e é invocada quando um recurso monitorado ultrapassar os limites especificados.

Objetos participantes de uma transação herdam da classe `LockManager` que é a responsável pelo controle de concorrência destes objetos. O método utilizado é o de trancas em duas fases [34]. A classe `StateManager`, indiretamente herdada, controla os aspectos de recuperação e persistência.

As trancas são realizadas sobre objetos remotos (nos modos de operação *Remoto* e *Local-Remoto*) e objetos locais (nos modos *Local-Remoto* e *Local*). Uma transação executando no modo *Local* deverá passar por uma validação após o seu término. Somente se passar neste processo, as versões locais dos objetos modificados pela transação serão propagadas e os objetos remotos atualizados. A validação consiste na comparação do *número de versão* da cópia local com o *número de versão* da cópia matriz. Se cada objeto participante da transação tiver o mesmo número na cópia local e matriz, a transação pode ser validada e um novo número de versão é gerado.

A seguir são descritos o mecanismo de cancelamento parcial e o protocolo de efetivação.

3.8.1 Cancelamento Parcial

Em se tratando de ambiente móvel, uma transação pode obter *trancas* sobre objetos remotos e a seguir perder a conexão com a rede fixa por um longo período. Neste cenário, um objeto bloqueado por essa transação ficaria indisponível indefinitivamente, impedindo outras transações de prosseguir.

A fim de evitar o problema, a classe `LockManager` realiza um *Cancelamento Parcial* que libera trancas e restaura estados de objetos remotos. Após um período de tempo determinado, o gerenciador de trancas (instância da classe `LockManager`) envia uma mensagem à transação. Caso a resposta seja obtida, a tranca é mantida. Caso contrário o cancelamento parcial será executado.

3.8.2 Efetivação

A classe `AtomicAction` utiliza diversas classes auxiliares a fim de possibilitar o gerenciamento da transação: `LockRecord`, responsável por gerenciar a tranca de um objeto; `RecoveryRecord`, para gerenciamento de recuperação de um objeto; `PersistenceRecord`, para persistência; `UpdatingRecord`, para atualização da cópia remota e `ValidationRecord` que controla o processo de validação.

Durante uma transação, são criadas instâncias dessas classes para cada objeto envolvido nessa transação. Essas instâncias são armazenadas em uma lista chamada `pendingList`. Assim que a transação é encerrada, o processo de efetivação inicia a fase um para cada objeto nessa lista. Se a fase um for executada com sucesso para todos os objetos, a transação tem sucesso e será efetivada. Caso contrário, será cancelada.

Se a máquina móvel estiver desconectada do resto do sistema no final da transação, essa transação passa para o estado pendente e é inserida numa fila de efetivação. Quando a conexão for restabelecida, uma instância da classe `CommitQueue`, responsável pela fila, realizará o processo de efetivação para cada transação pendente.

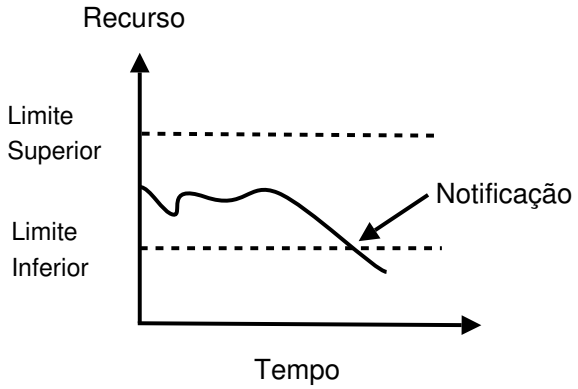

Figura 3.7: Monitoramento de um recurso

3.9 Monitor de Recursos

O monitor de recursos, executando na máquina móvel, monitora os recursos de interesse para uma transação. Para cada recurso, deve ser invocada a operação `request` que tem dois parâmetros: a referência da transação requisitante e o recurso a ser monitorado, com seus limites superior e inferior de tolerância. A operação retorna uma identificação da requisição para futuras solicitações de cancelamento do monitoramento ou de consulta sobre o nível atual do recurso.

Se um dos limites for violado, a operação `_notify` é invocada. Uma ação de adaptação pode, então, ser realizada (ver figura 3.7).

Módulos para o monitoramento de largura de banda, energia, espaço em disco e custo de comunicação foram desenvolvidos através de simulação devido à complexidade incompatível com o tempo disponível para o desenvolvimento do protótipo. Novos recursos podem ser adicionados ao monitor.

3.10 Relações com o Modelo SGTAM

Os módulos de Monitoramento de Recursos, Repositório de Objetos e Transações foram aproveitados na implementação do Modelo de Transações com Agentes Móveis (SGTAM) discutido nos capítulos 4 e 5. Contudo, o Monitor de Recursos foi incrementado para oferecer monitoramento de mudança de células (ver seção 5.7).

O gerenciador de transação passou a ser um agente móvel e pode executar em qualquer

máquina da rede. Aplicações e objetos de seu interesse também são executados como agentes móveis.

Foi necessário acrescentar uma nova classe `AtomicClient` para auxiliar no desenvolvimento da aplicação.

Os parâmetros nível de isolamento e modo de operação não foram incluídos no modelo SGTAM que só oferece como forma de adaptação a mobilidade de aplicação, gerenciador de transação e objetos participantes da transação. Esses parâmetros do SGTA não foram considerados porque sua integração com o parâmetro mobilidade seria complexa e excederia o tempo previsto para essa dissertação de mestrado.

Capítulo 4

Modelo de Transações Baseado em Agentes Móveis

O modelo de transações proposto nessa dissertação é baseado em uma versão simplificada do modelo descrito no capítulo 3 e utiliza as classes `AtomicAction` e classes auxiliares para o controle de concorrência, protocolo de efetivação em duas fases e os outros serviços correlatos. A alteração principal diz respeito à introdução de mobilidade em alguns módulos do sistema.

A mobilidade oferecida pelo Sistema Gerenciador de Transações para o Ambiente de Computação Móvel (SGTAM) [35] é obtida através da utilização de agentes: tanto o gerenciador de transações, quanto a aplicação e os objetos envolvidos são executados como agentes móveis e podem se mover de maneira independente uns dos outros.

Outro aspecto também considerado nesse modelo é o de interatividade de aplicações no ambiente móvel. Para lidar com esse aspecto, o usuário pode submeter operações passo-a-passo através de uma interface gráfica. A adaptação a mudanças no ambiente também é possível através da mobilidade de gerenciador de transação e objetos participantes.

Esse capítulo contém uma seção que descreve a arquitetura do sistema em 4.1, o apoio à adaptação em 4.2, o apoio à interatividade em 4.3 seguido da seção com o diagrama de classes do sistema desenvolvido.

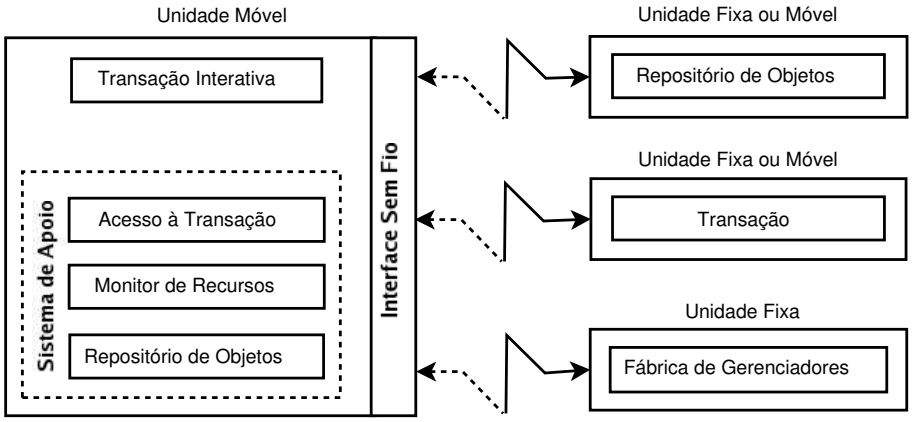

Figura 4.1: Arquitetura do SGTAM

4.1 Arquitetura do SGTAM

A arquitetura do SGTAM é composta por seis módulos que constituem o Sistema de Apoio e o Sistema de Gerenciamento de Transação. O Sistema de Apoio executa na máquina móvel e é responsável por monitorar os recursos do ambiente e também disponibilizar os mecanismos de adaptação. O Sistema de Gerenciamento de Transação executa em máquina fixa ou móvel, e é responsável por controlar a execução da transação.

O Sistema de Apoio é constituído pelos módulos Acesso à Transação, Monitor de Recursos e Repositório de Objetos. O Sistema de Gerenciamento de Transação é constituído pelos módulos Transação, Fábrica de Gerenciadores e Transações Interativas. A arquitetura do sistema é mostrada na figura 4.1. A seguir cada um dos módulos será discutido. O módulo de Transações Interativas será discutido em Apoio à Interatividade (seção 4.3).

4.1.1 Acesso à Transação

O módulo Acesso à Transação (classe `AtomicClient` na seção 4.4) é responsável por esconder do programador os detalhes de comunicação envolvendo a aplicação e o gerenciador de transação, como, por exemplo, localização. Todos os pedidos de operações para o módulo Transação são solicitados via o módulo de Acesso.

4.1.2 Fábrica de Gerenciadores

A Fábrica de Gerenciadores é responsável pela criação de gerenciadores de transação (instância da classe `AtomicAction`). A Fábrica é um objeto CORBA acessado através do serviço de nomes. Interage com a plataforma de agentes móveis para a criação de agentes gerenciadores. Detalhes de implementação são mostrados na seção 5.4.1.

4.1.3 Monitor de Recursos

O Monitor de Recursos segue o modelo descrito na seção 3.9 e é responsável por monitorar recursos que podem influenciar na decisão por migração de componentes. Atualmente, monitoramento da largura de banda e de mudança de célula são oferecidos.

O monitor é composto por módulos independentes e uma *interface* padrão onde as operações para requisição de monitoramento, cancelamento de monitoramento e nível atual do recurso são especificadas. Cada módulo é responsável pelo monitoramento de apenas um recurso e novos módulos podem ser adicionados. A aplicação é responsável por solicitar o monitoramento de recursos.

Ao receber o pedido de monitoramento, o monitor registra a requisição em uma lista de aplicações para cada tipo de recurso. A referência da aplicação é armazenada pelo monitor juntamente com os limites superior e inferior especificados. Em intervalos de tempo pré-definidos, os recursos são verificados e a aplicação notificada em caso de violação dos limites. Detalhes de implementação são mostrados na seção 5.7.

4.1.4 Repositório de Objetos

Uma propriedade fundamental dos objetos da transação é a de persistência. Para isso, o resultado de alterações sobre objetos do sistema são imediatamente salvos em disco após a transação ser efetivada. O repositório provê o mecanismo para transferência de estado dos objetos entre memória persistente e memória volátil e vice-versa. Segue o modelo descrito na seção 3.6.

4.1.5 Transação

A transação é o componente alvo do sistema oferecendo controle de concorrência baseado em trancas, recuperação de falhas e efetivação em duas fases.

4.2 Apoio à Adaptação

Enquanto o SGTA trata de mecanismos de adaptação relacionados com nível de isolamento e modo de operação, o modelo aqui proposto considera três tipos de mobilidade como forma de adaptação:

- mobilidade da aplicação: para permitir que uma aplicação migre de uma máquina para outra, a aplicação pode ser executada como um agente móvel. Assim, uma dada aplicação iniciada em uma máquina móvel pode migrar a qualquer momento para uma máquina fixa evitando as instabilidades da comunicação sem fio e poupando recursos do dispositivo móvel.

- mobilidade do gerenciador de transações: no sistema proposto, o gerenciador de transações é uma instância da classe `AtomicAction` (figura 3.4) que também é executado como um agente. Assim, por exemplo, o gerenciador pode migrar da máquina móvel para uma máquina fixa a fim de executar uma operação cara como o protocolo de efetivação em duas fases aproveitando a largura de banda da rede com fio.

- mobilidade dos objetos envolvidos em transações: objetos de interesse de uma transação executando em uma máquina móvel podem migrar de uma máquina fixa para outra mais próxima da máquina móvel, ou ainda, para a própria máquina local. Essa decisão pode ser tomada pela aplicação a fim de minimizar custos de comunicação.

Uma transação ao ser iniciada deve determinar qual desses aspectos de mobilidade melhor se adequa a seus objetivos. Este é um parâmetro que pode ser alterado durante a execução da transação.

4.3 Apoio à Interatividade

A interatividade do ambiente de computação móvel é tratada no SGTAM permitindo que usuários criem transações e decidam as operações a serem executadas em tempo de execução. Este modo de transação recebe o nome de Transação Passo-a-Passo e oferece uma interface gráfica para o usuário através da qual operações dentro do escopo de uma transação são submetidas.

Ao ser iniciado, o módulo de transações passo-a-passo se encarrega de automaticamente criar e iniciar uma nova transação. Após a criação, o usuário se torna apto a solicitar operações sobre objetos.

Um usuário do módulo passo-a-passo também pode receber notificações sobre variações no ambiente e tomar decisões sobre formas de adaptação a essas variações, como, por exemplo, migrar o gerenciador de transações ou os objetos participantes. Detalhes de implementação são mostrados na seção 5.8.

4.4 Diagrama de Classes

O diagrama de classes SGTAM é mostrado na figura 4.2 de maneira simplificada e é dividido em três partes: o sistema de apoio (cinza), sistema de transação (azul) e classe `MobileAgent` (amarelo) a ser utilizada para implementação de agentes móveis (ver seção 5.1.2). Classes não coloridas são definidas pelo usuário.

As classes `ObjectStore` e sua interface (contém operações para efetivação e recuperação de estado de um objeto) compõem o módulo Repositório de Objetos e são responsáveis por armazenar o estado de objetos em memória não volátil (seção 3.6). As classes `ResourceMonitor`, `BandwidthMonitor`, `CellMonitor` e `MonitorInterface` (define operações básicas para requisição e cancelamento de monitoramento, entre outras) representam o módulo Monitor de Recursos.

As classes `StateManager` e `LockManager` implementam o controle de recuperação, concorrência e persistência de objetos definidos no SGTA (ver seção 3.8).

A classe `StrongMigration` contém operações que possibilitam a migração forte de aplicação (seção 5.1.2). Um pré-compilador (seção 5.3.2, classe `PreCompiler`) transforma uma classe base desenvolvida pelo programador em uma aplicação que possa ser executada como agente móvel com capacidade de migração forte. Fazem parte do módulo Transação.

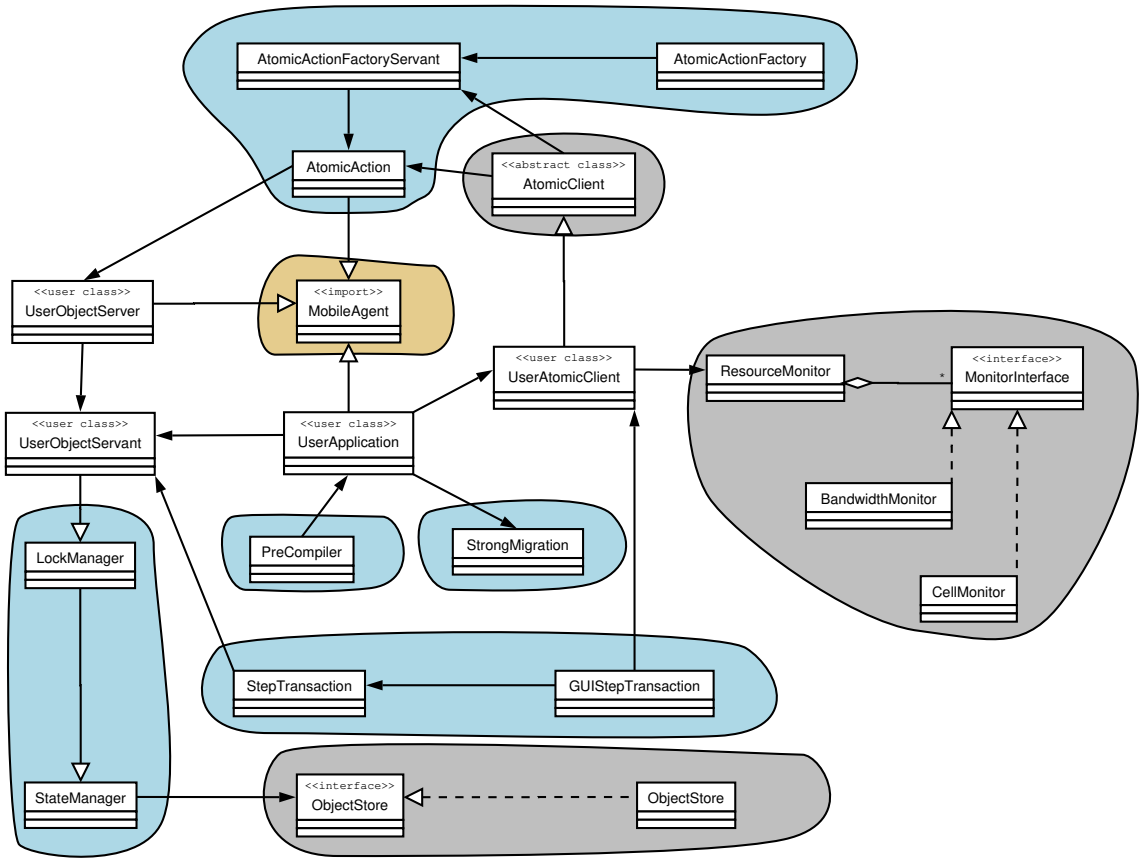

Figura 4.2: Diagrama de Classes do SGTAM

As classes `AtomicActionFactoryServant` (implementação de um objeto CORBA), `AtomicActionFactory` e `AtomicAction` fazem parte do módulo Fábrica de Gerenciadores e são responsáveis por criar novos gerenciadores para as aplicações.

O módulo Acesso à Transação é implementado pela classe `AtomicClient` e o módulo Transações Interativas pelas classes `StepTransaction` e sua interface gráfica `GUIStep Transaction`.

As classes restantes são definidas pelo usuário e representam os objetos utilizados por aplicações (`UserObject` e `UserObjectServant`) e a aplicação (`UserApplication` e `UserAtomicClient`).

Todas as classes `CellMonitor`, `StrongMigration`, `PreCompiler`, `AtomicActionFacto ryServant`, `AtomicActionFactory`, `AtomicClient`, `StepTransaction` e `GUIStepTrans action` foram desenvolvidas para o SGTAM como parte dessa dissertação. Testes para as classes `UserApplication` e `UserAtomicClient` foram desenvolvidos. A classe `Mobile Agent` é implementada pela plataforma Grasshopper. O restante das classes - `ObjectStore`, `ResourceMonitor`, `BandwidthMonitor`, `MonitorInterface`, `StateManager`, `LockManager`, `AtomicAction`, `UserObject`, `UserObjectServant` - foram aproveitadas do protótipo do SGTA, algumas com modificações para fornecer migração de componentes (`LockManager`, `UserObject`, `UserObjectServant`).

Capítulo 5

Implementação de Um Protótipo de Transações Baseadas em Agentes

O protótipo do SGTAM utiliza a camada ORB de CORBA (na seção 5.1.1) como plataforma de comunicação. Foi utilizada JacORB [36] que é livre e possui todas as características necessárias para a implementação do SGTAM inclusive interceptadores e invocação dinâmica. Os agentes foram desenvolvidos através da plataforma Grasshopper (na seção 5.1.2) que também tinha uma versão livre.

Esse capítulo apresenta o ambiente de execução em 5.1, o desenvolvimento das classes na implementação de uma aplicação em 5.2, detalhes sobre a mobilidade da aplicação em 5.3, mobilidade do gerenciador de transações em 5.4 e mobilidade de objetos em 5.5. A utilização de interceptadores na transmissão de referências é discutida na seção 5.6. As mudanças necessárias no Monitor de Recursos são apresentadas em 5.7. A seção 5.8 discute transações interativas. O capítulo termina com a seção que apresenta exemplos de aplicação que é também uma motivação para a utilização do modelo de transações proposto nessa dissertação.

5.1 Ambiente de Execução

A seguir será dada uma visão geral do ambiente sobre o qual foi desenvolvido o protótipo do SGTAM: a plataforma de comunicação descrita em 5.1.1 e a plataforma de agentes móveis Grasshopper em 5.1.2.

5.1.1 ORB

CORBA[1] [37] é uma arquitetura aberta, de várias camadas, desenvolvida pela OMG[2] com o objetivo principal de prover interoperabilidade entre objetos em sistemas heterogêneos [38].

O ORB[3] é a camada responsável pela comunicação entre objetos distribuídos. Um objeto CORBA é uma entidade encapsulada com uma identidade única e imutável cujos serviços são acessados através de interfaces. A implementação e localização de cada objeto não precisa ser do conhecimento do cliente.

O ORB entrega requisições dos clientes para objetos e devolve a resposta aos clientes. Sua principal característica é a transparência de comunicação. O cliente não sabe onde o objeto alvo reside, não sabe como o objeto é implementado (linguagem de programação, sistema operacional ou máquina em utilização) e nem mesmo se o objeto está ativo.

Um serviço de nomes é utilizado para mapear nomes a objetos. Cada nome é único no ambiente e deve ser registrado no serviço através de um método específico *bind*. A partir de então, qualquer objeto com um nome associado pode ser encontrado utilizando-se o método *resolve* que retorna sua referência e permite que seus métodos sejam invocados [39].

A especificação das operações de um objeto e tipos de dados utilizados é realizada na linguagem IDL[4]. É uma linguagem declarativa que permite que objetos sejam implementados em diferentes linguagens de programação e ainda se comunicarem uns com os outros [38]. A especificação IDL é compilada para a linguagem de programação da aplicação. Trechos de código chamados *stubs* e *skeletons* são gerados pelo compilador para, respectivamente, criar requisições para o lado do cliente e recebê-las pelo objeto alvo.

Contudo, nem sempre é possível conhecer com antecedência a interface dos objetos a serem utilizados. Para resolver esse problema, CORBA fornece uma Interface de Invocação Dinâmica[5] (DII) pela qual um cliente pode requisitar operações sobre um objeto sem ter qualquer conhecimento de sua interface em tempo de compilação. Um pseudo-objeto *Request* é utilizado e, antes de a requisição ser efetuada, o pseudo-objeto deve receber

[1] Acrônimo para Common Object Request Broker Architecture
[2] Acrônimo para Object Management Group
[3] Acrônimo para Object Request Broker
[4] Do inglês, Interface Definition Language
[5] Do inglês, Dynamic Invocation Interface

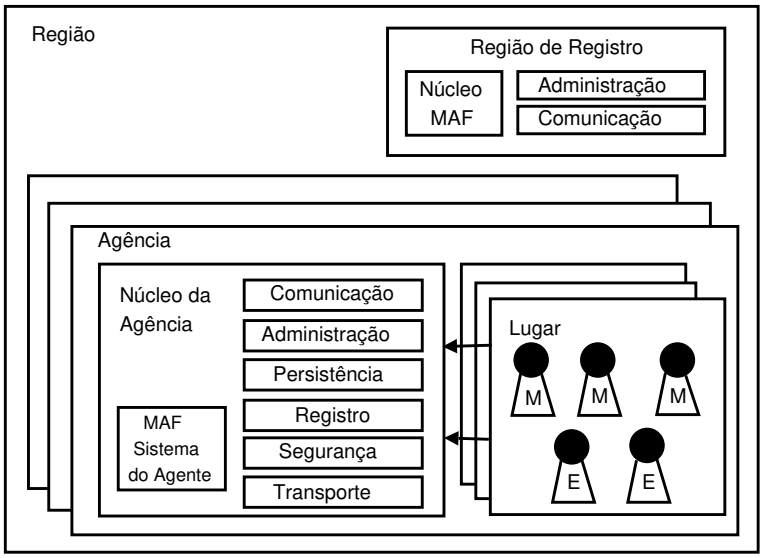

Figura 5.1: Estrutura da plataforma Grasshopper

a operação a ser invocada e seus argumentos. Analogamente ao *skeleton* da invocação estática, o servidor possui um skeleton dinâmico.

Um Repositório de Interfaces é utilizado para armazenar as IDLs dos objetos no ambiente CORBA. É também um objeto CORBA. Qualquer aplicação pode percorrer o repositório e obter os métodos disponíveis de um objeto e seus argumentos. Em especial, todo objeto CORBA tem um método `get_interface_def` pelo qual é possível consultar operações e seus parâmetros. Nem toda implementação CORBA possui um Repositório. Portanto, um dos critérios usados para escolher uma implementação ORB foi o oferecimento desse repositório e apoio a invocações dinâmicas.

5.1.2 Grasshopper

A plataforma Grasshopper foi criada pela empresa IKV++ e está logicamente dividida em regiões. Uma região contém internamente uma ou várias "agências" e uma única "Região de Registro". Estas agências também são subdividas, contendo cada uma delas exatamente um núcleo e um ou mais "lugares", como mostra a figura 5.1 [40].

As agências são os locais onde os agentes executam e para onde eles devem migrar quando solicitado, sendo que, em uma mesma máquina, pode existir mais de uma agência. Cada agência é subdivida em "lugares". Nestes "lugares", os agentes podem trocar

informações entre si, agrupando-se de acordo com as funcionalidades de cada um.

A subdivisão "núcleo" da agência é a área responsável pelo gerenciamento da agência. É ela que efetivamente oferece o apoio necessário aos agentes para a sua execução. Cada núcleo de agência é responsável por oferecer os seguintes serviços que serão utilizados pelos agentes durante o seu ciclo de vida:

- Serviço de Comunicação: responsável pela comunicação entre os agentes e suas agências. Pode-se escolher um protocolo de comunicação entre, por exemplo, os protocolos CORBA IIOP[6] e Java RMI[7].
- Serviço de Registro: notificado cada vez que um agente chega a uma agência. Sua intenção é manter um controle de todos os agentes que atualmente residem em cada agência.
- Serviço de Administração: voltado para interação com o mundo real. Através dele, realizam-se ações como criação e remoção de agentes.
- Serviço de Transporte: fornece as ferramentas necessárias para a migração dos agentes.
- Serviço de Segurança: responsável por isolar as agências e regiões umas das outras. São utilizados os protocolos SSL[8] e X.509[9].

A "Região de Registro" é o módulo responsável por abrigar informações de todas as agências que estão atualmente em execução no ambiente distribuído. Esse módulo não é obrigatório e não foi incluído na implementação do protótipo a fim de aumentar a compatibilidade com agentes desenvolvidos sobre outras plataformas (como, por exemplo, Telescript, Mole, TACOMA, Aglets) e minimizar os recursos necessários à execução da plataforma.

[6] Do inglês, Internet Inter-ORB Protocol
[7] Do inglês, Java Remote Method Invocation
[8] Do inglês, Secure Socket Layer. Protocolo desenvolvido pela Netscape para transmissão de documentos criptografados pela Internet
[9] Um padrão para utilização de certificados de chaves públicas

Método	*Descrição*
`beforeMove()`	Chamado automaticamente *antes* da migração de um agente.
`afterMove()`	Chamado automaticamente *após* a migração de um agente.
`move()`	Principal método disponibilizado, pois é ele que possibilita a migração do agente de uma agência a outra.

Tabela 5.1: Métodos dos agentes móveis

Especificação dos Agentes

Os agentes desenvolvidos com a plataforma Grasshopper são implementados em Java como uma classe que herda de uma das classes abaixo:

- MobileAgent
- PersistentMobileAgent
- PersistentStationaryAgent
- StationaryAgent

A superclasse `MobileAgent` foi a utilizada pelo SGTAM e provê ao agente as características e métodos necessários para que ele possa se locomover entre diversas agências. Seus métodos são explicados na tabela 5.1. A superclasse `PersistentMobileAgent` provê todas as funcionalidades da superclasse `MobileAgent` e ainda algumas outras necessárias a um serviço de pesistência de dados. `StationaryAgent` e `PersistentStationaryAgent` são as classes para agentes estacionários.

Agentes não podem ser inicializados como classes comuns em Java. Isso se deve a todo um processamento que é realizado imediatamente antes da criação do agente, como a criação de uma identificação própria e o registro do agente na agência. Sendo assim, torna-se necessária uma nova maneira para inicialização dos agentes. Isso é feito pelo método `init()` que funciona como um construtor do agente. Os métodos de migração e cópia de agentes não devem ser chamados durante a sua execução.

O método `live()` é o método mais importante responsável por executar toda a funcionalidade do agente. Ao ser criado, o agente passa a executar de maneira independente dos outros, sendo que cada agente executa em uma *thread* independente. Após a execução

do método `live()`, o agente passa a ser considerado como um agente passivo, ou seja, um agente que não está executando tarefas, mas ainda está apto a executá-las quando necessário.

Existem duas possibilidades para remoção de um agente: via interface com o usuário ou através do próprio agente. A remoção, no código do agente, acontece através do método `remove()`. A remoção via interface é realizada por entidades externas. Se durante o seu ciclo de vida, o agente houver solicitado a criação de mais *threads*, todas elas serão destruídas.

Um agente pode criar outros agentes ou até mesmo fazer cópias dele mesmo.

Migração e Cópia dos Agentes

Durante a migração, os agentes precisam levar consigo a informação de onde a sua classe base está armazenada. Essa informação é armazenada na classe que reune informações de um agente, *AgentInfo*. Como alternativa, há a opção de armazenamento das classes em um servidor HTTP.

Deve existir uma forma de garantir acesso às classes base por todos os agentes. Uma primeira solução é fazer uma cópia de todas as classes em todos os lugares em que os agentes podem executar. Essa solução simples, no entanto, contém todos os problemas inerentes à cópia indiscriminada de arquivos como, por exemplo, a manutenção de consistência entre as cópias. Uma outra solução é manter as classes unicamente no local de criação do agente. Cada vez que o agente migrar deve-se consultar a agência origem. Nessa abordagem, temos problemas inerentes à centralização de informações. O que fazer quando o servidor sair do ar? E se ele ficar sobrecarregado? Os agentes ficarão indefinidamente esperando? A terceira opção é manter o código no último local de execução do agente. Essa escolha possibilita uma maior distribuição dos dados permitindo assim que vários agentes possam ser criados numa mesma máquina sem problemas de desempenho posteriores.

A fim de minimizar as falhas de migração dos agentes, a plataforma Grasshopper realiza a busca da classe do agente em todas as fontes acima descritas, na seguinte ordem: local (CLASSPATH local), agência anterior, servidor HTTP e máquina de origem.

Caso seja necessário, uma cópia de um determinado agente pode ser criada a fim de duplicar a sua tarefa de execução. Todos os dados permanecem como no agente original.

Só o indicador de número de cópia na identificação do agente é alterado.

Migração Forte x Migração Fraca

Para possibilitar a migração, um padrão de comunicação entre os agentes e as suas plataformas é necessário. Na implementação do SGTAM, todos os agentes executam na mesma plataforma, a plataforma Grasshopper. No entanto, é desejável que um agente móvel possa executar em diferentes plataformas. Com esse objetivo, a plataforma Grasshopper foi a primeira a oferecer apoio ao padrão MASIF[10].

Na migração, o agente deve se mover entre diferentes plataformas de maneira absolutamente transparente. Não deve ser visível a suspensão de um agente e seu reinício em outra agência. O que interessa é tão somente o resultado do processamento, independente de qual máquina gerou o resultado [41].

Existem dois tipos de migração:

- migração forte: em que todas as informações de execução dos agentes, assim como os valores de suas estruturas de dados, devem ser levados em consideração na hora de migrar.

- migração fraca: em que são levadas em consideração apenas as estruturas de dados do agente.

Só a migração forte é totalmente transparente. Para alcançá-la, uma série de informações como código, objetos, pilha de execução, contadores de programa, referências a objetos e recursos do sistema, e *threads* criadas devem ser restabelecidos no destino do agente. Somente desta maneira, um agente pode continuar sua execução do ponto onde foi interrompida.

Para implementação da migração forte no SGTAM foram utilizadas a arquitetura JPDA[11] [42, 43] e a biblioteca para manipulação de *bytecode* BCEL [44]. A primeira permite obter o endereço da última instrução executada. A segunda permite inserir um salto para a última instrução executada [41].

[10]Do inglês, Mobile Agent System Interoperability Facility. Desenvolvido a fim de possibilitar a interação entre agentes móveis desenvolvidos em diferentes plataformas

[11]Do inglês, Java Platform Debugger Architecture

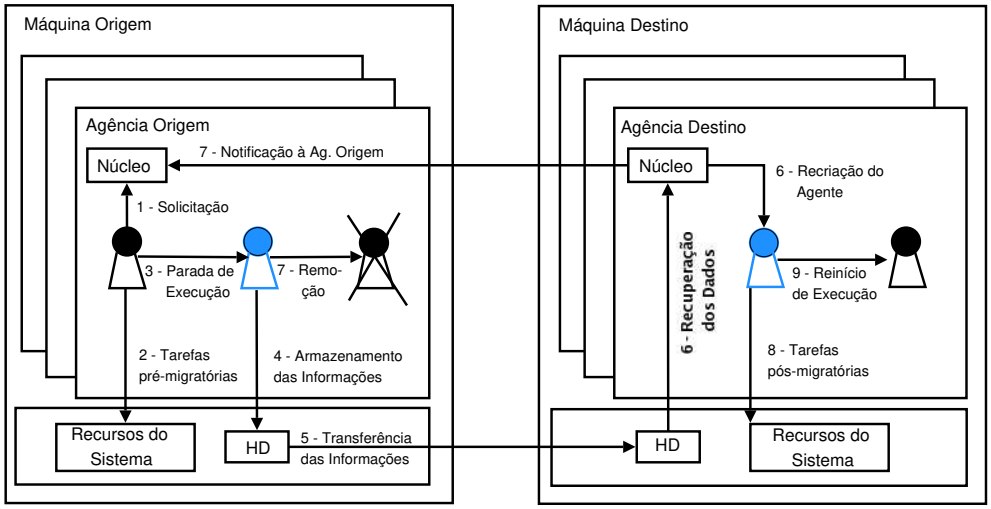

Figura 5.2: Migração de agente

Processo Migratório

Há três maneiras de um determinado agente migrar:

- quando o agente recebe uma indicação de que deve deixar o seu local atual de execução. Por exemplo, o administrador do sistema pode resolver que o agente deve migrar da máquina atual.
- através de interação do agente com o ambiente. Por exemplo, um agente pode receber a ordem de migração através de um outro agente. Isso faz parte da comunicação entre os agentes e mostra também o quanto se deve ter cuidado em permitir a interação de um novo agente com os agentes já existentes. Sempre há o perigo de agentes mal intencionados.
- o próprio agente é que toma a iniciativa de migração.

O processo total de migração do agente envolve nove passos (ver figura 5.2):

- Passos 1 e 2: Solicitção da migração à agência origem e verificação de possíveis tarefas a serem realizadas antes da migração (`beforeMove`).
- Passo 3: Parada de execução do agente, ou seja, finalização da *thread* do agente e de todas as *threads* que tiverem sido criadas pelo mesmo.

- Passos 4 e 5: Armazenamento das informações do agente na sua origem e transferência destas informações para a agência destino.
- Passo 6: Recriação do agente na agência destino. Todos os dados transferidos são recuperados para possibilitar a recriação do agente.
- Passo 7: Remoção do agente na agência origem. Caso o processo de migração ocorra com sucesso, a agência origem recebe esta informação da agência destino e remove o agente da sua agência.
- Passo 8: O método `afterMove()` é chamado na agência destino para realizar possíveis tarefas antes do reinício do agente na nova agência.
- Passo 9: O agente reinicia sua execução na máquina destino. É só neste ponto que a *thread* de execução é reiniciada e o agente continua a sua execução.

O custo de uma migração é tanto maior quanto mais variáveis não transientes[12] houver no agente. Assim, se o valor da variável na agência anterior não for necessária à execução do agente na agência destino, a mesma deve ser declarada como do tipo transiente.

Agentes no SGTAM

Os agentes no SGTAM são do tipo móvel (herdam da classe `MobileAgent`) para possibilitar a migração entre os diversos computadores do ambiente. Seus serviços de persistência foram desabilitados para obter um ganho de desempenho.

Ao serem iniciados, os agentes devem receber três parâmetros: a localização onde iniciar a execução; a máquina onde o ORB está em execução; e a política de adaptação a ser seguida (aplicação, gerenciador de transação ou objetos participantes). Caso alguns dos parâmetros não sejam informados, valores padrão são usados (esses valores são a máquina onde a solicitação foi efetuada e política de adaptação mobilidade de gerenciador).

Para facilitar o acompanhamento do ciclo de vida dos agentes, assim como a troca de mensagens entre os componentes do sistema, toda a comunicação relevante entre os agentes são registradas em uma console da plataforma Grasshopper.

[12] Objetos Transientes não são inclusos no processo de migração.

```
1  class MyAtomicClient extends AtomicClient {
2   void _notify (String requestId, String monitorId, int resourceLevel){
3    //política de adaptação
4   }
5  }
```

Figura 5.3: Definição da classe MyAtomicClient

5.2 Desenvolvendo uma Aplicação

Para se desenvolver uma aplicação no SGTAM, o programador deve definir uma subclasse `MyAtomicClient` (na figura 5.3) da classe abstrata `AtomicClient` (ver interface na figura 5.4). O construtor dessa subclasse cria um gerenciador de transações do tipo `AtomicAction` através de uma fábrica. As operações de `Begin`, `Commit` e `Abort` de `AtomicClient` realizam invocações nas operações correspondentes do gerenciador de transação. Além disso, a operação `setAtomicActionRef` é usada para receber notificações sobre a localização do gerenciador de transações quando o gerenciador é criado ou quando migra. A recuperação de referência é mostrada na figura 5.5. Na subclasse `MyAtomicClient`, o programador deve definir a operação abstrata `_notify` que implementa uma política de adaptação.

Além disso, o programador deve implementar uma classe para a aplicação propriamente dita (figura 5.6) que herda da classe Grasshopper `MobileAgent`. Nessa classe é criado um objeto do tipo `MyAtomicClient` e, em seguida, a transação pode ser iniciada. Toda operação de gerenciamento de transação é realizada indiretamente através de `MyAtomicClient`. A transação é iniciada pela operação `Begin` (linha 9 na figura 5.6) em que a política de adaptação é especificada (mobilidade de aplicação). Essa política pode ser modificada durante a transação através da operação de `SetMigPolicy` da classe `AtomicClient`. A aplicação, ao solicitar monitoramento de recursos do sistema (linha 10 na figura 5.6), deve fornecer sua referência a fim de poder ser contactada pelo Monitor de Recursos. Qualquer chamada ao gerenciador de transações é realizada através de `MyAtomicClient` que mantém a referência do gerenciador atualizada e torna aplicação livre de detalhes sobre a atualização de referência do gerenciador. Ao receber uma notificação sobre mudanças no ambiente, a aplicação, na operação `_notify`, pode mudar a política ou requisitar migração de algum módulo (conforme especificado pela política de

```
1  interface AtomicClient{
2   //Informa referência do Gerenciador à Aplicação
3   void setAtomicActionRef(in string atomicARef);
4   //Devolve o identificação da Transação
5   string getId();
6   //Retorna o estado da Transação
7   long status();
8   //Solicita o cancelamento da transação ao Gerenciador
9   long Abort();
10  //Solicita o início da transação ao Gerenciador
11  long Begin();
12  //Solicita o fim da transação ao Gerenciador
13  long Commit();
14  //Solicita migração do Gerenciador à AtomicAction
15  long migrateManager(in string strDestiny);
16  //Solicita migração dos Objetos à AtomicAction
17  long migrateObjects(in string strDestiny);
18  //Solicita migração da Aplicação
19  long migrateApplication(in string strDestiny);
20  //Solicita migração de acordo com a Política de Adaptação
21  long migrate(in string strDestiny);
22  //Determina política de migração a a ser seguida
23  void setMigPolicy(in string strDestiny);
24  //Usado pelo monitor de recursos para notificar a aplicação
25  //que ocorreram mudanças significativas no nível do recurso
26  //monitorado.
27  void _notify(in string requestId, in long resourceLevel);
28 }
```

Figura 5.4: Interface IDL de `AtomicClient`

```
1  AtomicAction action = AtomicActionHelper.narrow(ORB.init(...).
   string_to_object(atomicActionRef));
```

Figura 5.5: Recuperação de referência do Gerenciador

```
1  public class ApplicationAgent extends de.ikv.grasshopper.agent.
   MobileAgent implements ICORBAReferences {
2    ...
4    public void live ( ) {
5      MyAtomicClient atomicClient;
6      ResourceMonitor resourceMonitor;
7      ...
9      atomicClient.Begin(Migration.APPLICATION);
10       resourceMonitor.request(clientRef, new ResourceDescriptor
         (“CellMonitor”, 0, 0));
11       op1;
12       op2;
13       ...;
15     atomicClient.Commit();
16     ...
18   }
19 }
```

Figura 5.6: Exemplo de uma classe de aplicação

adaptação).

Quando a aplicação migra, todas as referências dos objetos acessados se tornam inválidas. Para recuperação dessas referências, toda aplicação SGTAM define ainda a interface Java `ICORBAReferences`. A interface contém o método `getObjectsReference`, em que todos os objetos acessados devem recuperar sua referência CORBA (método *resolve*, seção 5.1.1). O método é chamado automaticamente pelo sistema através do método `afterMove`, definido na superclasse do agente, após o agente migrar de uma agência a outra.

5.3 Mobilidade da Aplicação

A mobilidade da aplicação é um processo complexo envolvendo modificação de *bytecode* da aplicação, e armazenamento e recuperação de variáveis locais. No entanto, esse modo permite que a aplicação se mova para uma máquina com mais recursos que a máquina móvel e pode permitir que a aplicação execute junto aos objetos de seu interesse diminuindo custos de comunicação.

A seguir são descritos os detalhes para obtenção da migração forte em 5.3.1 e os passos

executados no pré-compilador em 5.3.2.

5.3.1 Obtendo Migração Forte

A plataforma Grasshopper não oferece apoio à migração forte de agentes. Para obter esse tipo de migração foi necessário utilizar um depurador[13] [43] que é chamado antes da migração da aplicação a fim de capturar as últimas instruções executadas pelos métodos empilhados.

O armazenamento de variáveis locais é feito através das instruções `try/catch` do Java. Esse código, assim como o código para a recuperação desses valores é inserido por um pré-compilador discutido em 5.3.2.

Os passos para a migração forte são descritos a seguir.

Armazenamento de Contadores de Programa

O depurador funciona como uma aplicação cliente que se conecta a um servidor e faz com que esse passe a executar em modo de depuração. No caso do SGTAM, uma porta de comunicação é aberta pela máquina virtual para que o depurador obtenha os contadores de programa dos métodos empilhados pela aplicação. A aplicação inicia a *thread* do depurador assim que recebe a ordem de migração. Na criação dessa thread, a aplicação passa o nome da sua classe e a identificação do agente Grasshopper para que o depurador possa identificar a *thread* do agente da aplicação. Após criar a *thread* do depurador, a aplicação fica bloqueada até a sua finalização. A classe `FrameState` implementa o depurador.

O canal de comunicação entre o depurador e a máquina virtual consiste de um conector que pode ser do tipo *listening* (o depurador aguarda requisições de uma outra máquina virtual), *attaching* (o depurador se conecta a uma máquina virtual em execução) ou *launching* (o depurador se conecta a uma outra máquina virtual); e um mecanismo de transporte, podendo esse ser do tipo *socket*, linhas seriais ou memória compartilhada. Na implementação do protótipo foram escolhidos conector do tipo *attaching* e *socket* como mecanismo de transporte.

[13] Do inglês, Debugger

```
1  VirtualMachine vm = launchTarget(vmPort);
2  List threads = vm.allThreads ( );
3  ...
5  ThreadReference thread = (ThreadReference) threads.get(i);
6  thread.suspend( );
7  List frames = thread.frames( );
8  for ( int i = 0 ; i < frames.size ( ) ; i ++ ) {
9    ...
11   StackFrame frame = (StackFrame) frames.get(i);
12   long pc = frame.location( ).codeIndex( );
13   ...
14   framesTable.put (methodID , pc);
15 }
```

Figura 5.7: Passos para obtenção dos Contadores de Programa

O processo para obtenção de um contador de programa (ver figura 5.7) envolve um total de cinco passos descritos a seguir:

1. o estabelecimento da conexão do depurador com a máquina virtual (linha 1) é realizado através de um conector do tipo *attaching*.

2. as *threads* em execução na máquina virtual são obtidas. Todas as *threads* são listadas e o depurador busca pela *thread* de execução do agente. A *thread* encontrada é suspensa (linhas 2 a 6).

3. o nome da classe da aplicação é utilizado para identificação dos métodos (*frames* na arquitetura JPDA) empilhados pela aplicação[14] (linhas 7 a 11).

4. o último *bytecode* executado por cada método é obtido (linha 12), realizando a chamada `codeIndex` (arquitetura JPDA).

5. Finalmente, o valor é salvo em uma variável de instância (*framesTable*), que pode ser acessada diretamente e é levada junto com o agente em migração.

Recuperação de Contadores de Programa

A biblioteca BCEL foi utilizada para reconstruir a pilha de execução do programa. Em cada método anteriormente empilhado pela aplicação, uma instrução `goto` é inserida fa-

[14] Outros métodos internos à execução da plataforma *Grasshopper* também estão empilhados

zendo com que a seqüência de execução salte para a última linha de código executada. A classe `ByteCodeTransform` implementa tal funcionalidade e recebe como parâmetros o local e nome da classe base do agente onde inserir o desvio; e a tabela de métodos/contadores de programa (*framesTable* na figura 5.7) obtida na fase anterior. Os passos para recuperação são os seguintes:

1. O identificador de um método especial `doNothing` é procurado na tabela (o método `doNothing` é inserido pelo pré-compilador e indica a localização no código da aplicação onde deve ser inserido o salto) de constantes da classe Java. Essa tabela contém todas as referências de classe, métodos e variáveis acessadas pela aplicação.

2. Todos os métodos da classe são pesquisados em busca da identificação dos métodos anteriormente empilhados (tabela de métodos *framesTable*).

3. Em cada método a ser alterado, a lista de instruções é percorrida até o indicador `doNothing` para inserção do salto `goto`.

4. A lista de instruções continua a ser percorrida até a última instrução executada pelo método.

5. Para o método no topo da pilha (último executado), o `goto` a ser inserido é para a instrução seguinte (ao endereço contido na pilha). Para os outros métodos na pilha, o `goto` deve ser para a chamada do método imediatamente superior na pilha.

6. Com o local para inserção da instrução e o local para onde o salto deve ser dirigido, a inserção é realizada.

O código do agente, através da chamada `beforeMove`, é o responsável por iniciar a captura dos contadores de programa e alteração do *bytecode* (ver figura 5.8). A inserção da instrução `goto` é mostrada na figura 5.9.

Armazenamento de Variáveis Locais

O valor das variáveis locais tanto do último método executado quanto dos outros métodos empilhados necessitam ser salvos e recuperados. Isso é feito movendo as declarações de variáveis locais para o início do método, colocando as instruções do método dentro do

```
1  public void beforeMove ( ) {
2    ...
3    //Thread para salvamento dos contadores de programa.
4    FrameState frameState = new FrameState (...);
5    frameState.start ( ) ; //Thread iniciada...
6    frameState.join ( ); //Aguarda salvamento...
7    //Alterações de bytecode.
8    ByteCodeTransform byteCodeTransform = new ByteCodeTransform (...)  ;
9    byteCodeTransform.modify() ;
10   ...
13 }
```

Figura 5.8: Tarefas executadas antes da migração do agente da aplicação

```
1  instructionList.insert(insertPosition, new GOTO(nextInstructionPositi
   on));
```

Figura 5.9: Inclusão do salto no *bytecode* da aplicação

bloco `try` e salvando as variáveis no bloco `catch` correspondente a `Error` (figura 5.10). O pré-compilador altera o código da aplicação para inserir essas instruções.

Na migração, um erro do tipo `Error` é lançado. Todos os métodos na pilha de execução são percorridos para o armazenamento das variáveis locais: após o armazenamento das variáveis de um método, o erro é novamente lançado (no bloco `catch`, linha 10 da figura 5.11) e o método seguinte da pilha de execução processado [45].

A aplicação, antes da migração, é responsável por lançar o erro. A classe `SaveStateToDest` estende a classe `java.lang.Error` a fim de criar o erro pretendido. Todas as variáveis definidas em cada método (deslocadas para fora do bloco `try/catch`) são salvas. O último método na pilha é sempre o método `live` (que é onde a execução de um agente inicia - ver seção 5.1.2). Os valores são salvos em uma variável de instância *varTable*, de maneira similar ao armazenamento dos contadores de programa.

```
1  public class SaveStateToDest extends Error {
2    public String strDestination;
3    public SaveStateToDest(String strDestination) {
4      this.strDestination = strDestination;
5    }
6  }
```

Figura 5.10: Classe que define um `Error` para armazenamento de variáveis locais

```
1  public void method1( ){
2    int a$int = 0;
3    String b$String = “”;
4    try {
5      ...
7    }catch (SaveStateToDest sd) {
8      varTable.put(“method1:-a$int”, aInt);
9      varTable.put(“method1:-b$String”, b$String);
10     new ThrowSaveStateToDest ( sd.strDestination ) ;
11   } catch (Exception e) {
12     System.err.println ( “Exception:  ”  + e ) ;
14   }
15 }
```

Figura 5.11: Armazenamento de variáveis locais

A classe que estende o `Error` é mostrada em 5.10 e carrega consigo o destino da migração do agente. O lançamento do erro ocorre através de uma outra classe, `ThrowSaveState ToDest`, responsável por instanciar `SaveStateToDest`. Um exemplo é mostrado na figura 5.11 em que as linhas 8 e 9 correspondem ao armazenamento de variáveis e a linha 10 ao lançamento de erro.

Recuperação de Variáveis Locais

Após cada declaração de variável local, o pré-compilador verifica se a variável de instância *varTable* tem um valor correspondente a ser recuperado. Caso exista, o valor é restaurado.

Na figura 5.12 temos um exemplo de recuperação de valores locais. Na linha 2, a variável a é declarada e inicializada com valor padrão 0. Da linha 3 a 9, verifica-se em *varTable* se os valores a e b devem ser recuperados. A partir da linha 10, a aplicação continua seu curso de execução. A explicação para mudança nos nomes das variáveis locais será dada na seção 5.3.2.

5.3.2 Pré-Compilador

O analisador de palavras JavaCC [46] é utilizado pelo pré-compilador para processamento do código da aplicação. Um total de oito passos são necessários para preparar o arquivo de uma aplicação Java que será executada como um agente móvel. Os passos para se

```
1  public void method1( ){
2    int a$int = 0;
3    if ( varTable.containsKey ( ‘‘method1:-a$int’’ ) ) {
4     a$int = ( Integer ) varTable.get ( ‘‘method1:-a$int’’ ) ;
5    }
6    String b$String = ‘‘’’;
7    if ( varTable.containsKey ( ‘‘method1:-b$String’’ ) ) {
8     b$String = ( String ) varTable.get ( ‘‘method1:-b$String’’ ) ;
9    }
10   ...
12 }
```

Figura 5.12: Recuperação de variáveis locais

obter a migração forte são os seguintes:

1. Identificação da classe do agente e verificação se a *interface* `ICORBAReferences` está definida. O método `main` é redefinido como o método `live` do agente.

2. Instanciações na classe do agente são excluídas e as chamadas dos métodos respectivos efetuadas com o comando `this`[15].

3. Variáveis definidas para tratamento de erros são excluídas da lista de variáveis a serem salvas para migração. Não fazem parte da pilha de execução da aplicação.

4. Variáveis de instância e locais são identificadas através da biblioteca BCEL. Variáveis de instância têm os nomes alterados para não coincidir com nomes de variáveis locais que tiverem o ponto de declaração alterado.

5. Nomes de variáveis locais são alterados para evitar conflitos com declarações de tipos distintos para um mesmo nome de variável. As definições das variáveis locais ao longo de cada método são retiradas.

6. O método `live` é alterado para disparar a migração do agente após o processamento do mecanismo de erro. Outros métodos são alterados para propagar a exceção `Error` e o código para armazenamento de variáveis locais é inserido.

7. Definições das variáveis locais são deslocadas para o início de cada método e valores padrão são atribuídos. O código para recuperação de valores é inserido.

[15] Agentes não podem ser instanciados diretamente. Ver seção 5.1.2

```
1  public void method1( ){
2    try {
3      int a = 0;
4      String b = “”;
5      ...
8    } catch (Exception e) {
9      ...
11   }
12 }
```

Figura 5.13: Método antes da pré-compilação

8. O método `init` de `MobileAgent` é manipulado para receber os argumentos originários de `main`. Os métodos `beforeMove` e `afterMove` são definidos para liberação e recuperação de referências CORBA obtidos pela aplicação. A aplicação estende a classe `MobileAgent` da plataforma *Grasshopper* para utilização de agentes. Variáveis de instância que carregam os contadores de programa e valores das variáveis locais são definidas. O método especial `doNothing` é inserido ao término da declaração das variáveis locais e restauração de seus valores.

Na figura 5.13 um método de uma aplicação é mostrado antes da alteração pelo pré-compilador. Duas variáveis locais são definidas no método (linhas 3 e 4) e qualquer exceção Java é tratada pela cláusula `catch`. A figura 5.14 mostra o método resultante do processamento do pré-compilador. Todas as variáveis passam a ser declaradas (linhas 2 e 6) do lado de fora do bloco de tratamento de erro a fim de serem acessíveis pela cláusula `catch`. As variáveis são inicializadas com valores padrão nas linhas 2 e 6 e testadas para recuperação de valores nas linhas 3 a 5 e 7 a 9. Após a codificação do método, a cláusula de tratamento de `Error` específica para o erro de migração é mostrada na linha 13. As linhas seguintes prosseguem com outros tratamentos de erros existentes.

5.4 Mobilidade de Gerenciador de Transação

O Gerenciador de Transação é o módulo responsável por garantir a atomicidade da transação e manter a lista de objetos envolvidos na mesma e, assim, poder realizar o protocolo

```
1  public void method1( ){
2    int a$int = 0;
3    if ( varTable.containsKey ( ‘‘method1:-a$int’’ ) ) {
4      a$int = ( Integer ) varTable.get ( ‘‘method1:-a$int’’ ) ;
5    }
6    String b$String = ‘‘’’;
7    if ( varTable.containsKey ( ‘‘method1:-b$String’’ ) ) {
8      b$String = ( String ) varTable.get ( ‘‘method1:-b$String’’ ) ;
9    }
10   try {
11     a$int++;
12     b$String = ‘‘UNICAMP’’;
13   } catch (SaveStateToDest sd) {
14     varTable.put(‘‘method1:-a$int’’, aInt);
15     varTable.put(‘‘method1:-b$String’’, b$String);
16     new ThrowSaveStateToDest ( sd.strDestination ) ;
17   } catch (Exception e) {
18     System.err.println ( ‘‘Exception:  ’’  + e ) ;
19   }
20 }
```

Figura 5.14: Método após pré-compilação

```
1  interface Action{
2   //Devolve o identificador da AtomicAction
3   string getId();
4   //Adiciona um registro na lista de elementos pendentes
5   long add(in AbstractRecord ar);
6   //Remove um registro na lista de elementos pendentes
7   long remove(in AbstractRecord ar);
8   //Solicita Migração do Gerenciador
9   long migrateManager(in string strDestiny);
10  //Solicita Migração dos Objetos participantes.
11  long migrateObjects(in string strDestiny);
12  //Informa referência da Aplicação ao Gerenciador.
13  long setAtomicClientRef(in string atomicCRef);
14  //Retorna o estado da AtomicAction.
15  long status();
16  //Aborta a AtomicAction.
17  long Abort();
18  //Inicia a AtomicAction.
19  long Begin();
20  //Encerra a AtomicAction.
21  long Commit();
22 }
```

Figura 5.15: Interface IDL do Gerenciador de Transação

de efetivação em duas fases. São mantidas três listas para o gerenciamento de uma transação: uma lista de objetos envolvidos na transação e duas outras listas para o processo de efetivação (uma para objetos que forem somente lidos e outra de objetos modificados). A interface IDL do gerenciador é mostrada na figura 5.15.

O gerenciador pode acompanhar a aplicação e estar sempre próximo, ou até mesmo na mesma máquina, para minimizar custos de comunicação. Uma outra vantagem da mobilidade do gerenciador é permitir que operações caras sobre objetos, como o protocolo de efetivação, possam ser executadas na rede fixa.

O gerenciador é um objeto CORBA que atua como um agente móvel *Grasshopper* estendendo a classe base `MobileAgent`. Ao ser criado, o gerenciador se registra no ambiente CORBA. A referência necessária à sua localização é informada pelo próprio gerenciador aos outros componentes do sistema.

Novos métodos para comunicação entre gerenciador e aplicação foram desenvolvidos.

Em especial, o gerenciador ao ser criado recebe a referência da aplicação (criada por `AtomicClient`) e tem como primeira tarefa transmitir sua própria identificação a essa aplicação. Através dessa identificação, a aplicação solicita as operações de gerenciamento da transação (na classe `AtomicClient`). A referência do gerenciador deve ser atualizada em caso de migração do gerenciador através do método `setAtomicActionRef` (figura 5.4).

Enquanto o gerenciador não informar sua referência à aplicação, esta fica bloqueada aguardando que o gerenciador seja criado e inicializado (ver seção 5.4.1). A aplicação aguarda um tempo pré-definido até que a referência seja enviada. Caso não haja notificação, novas tentativas são realizadas.

Como os objetos participantes da transação podem se mover, a lista de participantes da transação precisa ser atualizada quando ocorrer a migração. Durante a migração de objetos, o gerenciador fica bloqueado, até que o processo se complete e os objetos possam, então, enviar suas novas referências.

A migração do gerenciador é mais simples que a migração da aplicação. Não há variáveis locais e contadores de programa a serem salvos. O gerenciador foi projetado de maneira que suas operações pendentes sejam inteiramente processadas antes de a migração ser iniciada. Apenas sua lista de objetos participantes e listas de objetos para efetivação são salvas e recuperadas após a migração para a máquina destino.

O gerenciador possui um papel também na migração de objetos. A aplicação solicita a migração de objetos (na classe `AtomicClient`) ao gerenciador. O gerenciador monta uma lista com os objetos a migrar e requisita migração para cada objeto. A decisão final pela migração cabe aos próprios objetos (ver seção 5.5).

5.4.1 Fábrica de Gerenciadores

A fábrica de gerenciadores é um objeto CORBA. Pode residir em qualquer máquina e está disponível para criar novos gerenciadores desde que a plataforma Grasshopper esteja em execução na máquina alvo que hospedará um gerenciador criado. O serviço de nomes CORBA é utilizado para registrar a fábrica e um *resolve* ao nome *AtomicActionFactory* recupera sua referência. Sua *interface* IDL é mostrada na figura 5.16.

Na figura 5.17 é mostrada a criação de um gerenciador pela fábrica. Na linha 3, o destino do gerenciador é passado no formato de um endereço *Grasshopper*. O destino inicial do gerenciador é *strDestiny* e o mecanismo de transporte utilizado é do tipo *socket*.

```
1  interface AtomicActionFactory {
2   void newAtomicAction();
3  };
```

Figura 5.16: Interface IDL da Fábrica de Gerenciadores

```
1  void newAtomicAction() {
2   try {
3    GrasshopperAddress agencyAddress = new GrasshopperAddress
     (strDestiny‘‘socket://’’+strDestiny+‘‘:7000/Agency)’’;
4    IAgentSystem agencyProxy = ( IAgentSystem ) ProxyGenerator.
     newInstance(IAgentSystem.class , agencyAddress.generateAgent
     SystemId(), agencyAddress);
5    java.lang.Object args[] = {AAFactoryORBInitializer.getRef()};
6    agencyProxy.createAgent (‘‘AtomicAction’’ , agentCodeBase , ‘‘’’ ,
     args);
7   } catch (Exception e) {
8    System.err.println(‘‘[newAtomicAction]Exception:’’  + e);
9   }
10 }
```

Figura 5.17: Criação de um Gerenciador pela Fábrica

Em 4, um *proxy* para a agência destino é criado. Em 5, a referência da aplicação é obtida através de interceptadores (ver seção 5.6). Em 6, o agente é criado na máquina destino. Da linha 7 em diante, o tratamento de erros é realizado.

5.5 Mobilidade de Objetos

Ao contrário da migração da aplicação, a migração de um objeto é realizada sem modificações no *bytecode*. O próprio Grasshopper trata da transferência do estado do objeto.

A migração de objetos é uma alternativa à utilização de cópias em *cache* da máquina móvel (uma das opções de adaptação no modelo SGTA discutido no capítulo 3). Se, ao invés de cópia, for realizada a migração do objeto para a máquina móvel, o custo de comunicação seria minimizado sem a necessidade de validação.

Outra opção é a migração do objeto para uma máquina próxima da aplicação. Se a aplicação se mover os objetos poderiam se mover também para manter a proximidade e diminuir o custo de comunicação.

```
1  boolean askMigration(String transId) {
2    ...
4    for (int i = 0; i < lockList.size(); i++) {
5      if (!this.lockList.get(i).getTransId().equals(transId)){
6        return false;
7      }
8    }
9    ...
11 }
```

Figura 5.18: Solicitação de migração de objetos

A classe `LockManager` de todo objeto possui um método chamado `askMigration` que deve ser invocado pelo gerenciador quando a política de adaptação for mobilidade de objetos. Ao ser solicitado, o método varre a lista de trancas existentes sobre o objeto verificando se há trancas de outras transações. Caso haja, a opção adotada é barrar a migração desse objeto. A figura 5.18 mostra um exemplo em que uma transação está solicitando a migração de objetos.

Na migração de um objeto, a referência do objeto é retirada do gerenciador para ser recriada na máquina destino. Ao chegar no destino, a lista `RecordList` criada por `LockManager` para manipular aspectos de recuperação e persistência de objetos é percorrida e as referências dos objetos que sofreram migração são repassadas ao gerenciador. Um outro enfoque existente seria fornecer referências transparentes de objetos, conforme o trabalho desenvolvido em [47].

Para que o custo da migração de objetos seja menor, novas referências dos objetos não são automaticamente enviadas para a aplicação. Ao invés disso, a própria aplicação é responsável por invocar *resolve* ao servidor de nomes CORBA e obter a referência atual do objeto sempre que ocorrer falhas de comunicação. Para isso, uma invocação sobre um objeto deve ser realizada dentro de um laço. Após um número fixo de tentativas, se a nova referência não puder ser obtida, o objeto é considerado indisponível.

Na figura 5.19 a referência de um objeto *Counter* é obtida (linha 3). Na linha 7, uma operação solicitando o valor atual do contador é efetuada. Caso ocorra falha, o laço da figura 5.20 (linhas 9 a 24) é executado até cinco vezes na tentativa de se restabelecer a conexão. Intervalos de um segundo são inseridos entre as tentativas.

```
1  public void live() {
2    ...
3    Counter objectRef = connectToObject();
4    while (objectRef != null) {
5     try{
6       // Invoca operação no objeto
7       int result = objectRef.getValue();
8       Thread.currentThread().sleep(1000);
9     } catch(Exception e) {
10      // A exceção pode ter ocorrido devido a uma migração de objetos.
11      // Tentar reconexão.
12      objectRef = connectToObject();
13    }
14   }
15 }
```

Figura 5.19: Obtenção de referências de objetos

5.6 Interceptadores

Sempre que uma aplicação faz uma invocação sobre um objeto, a referência do gerenciador é transmitida a esse objeto. O objeto utiliza essa referência para notificar o gerenciador sobre sua entrada na transação e eventuais migrações. De maneira similar, na criação de um gerenciador de transações, a referência da aplicação é passada para que o gerenciador envie à mesma sua referência.

Essas passagens de referências são feitas através de interceptadores de mensagens CORBA [48]. Os interceptadores interceptam o fluxo de um requisição ORB tanto no momento de envio quanto na recepção da mensagem. Através deles as referências da aplicação e do gerenciador podem ser transmitidas. Os interceptadores são usados nas classes `AtomicClient`, Objetos e Fábrica de Gerenciadores, fazendo com que a aplicação/programador não precise se preocupar com a transmissão das referências.

No momento em que o gerenciador da transação informa sua referência à aplicação, o ORB através do qual as requisições são enviadas é inicializado com a propriedade `ORBInitializerClass.AAClientORBInitializer`. Com isso, a classe `AAClientInterceptor` se torna o interceptador de todas as mensagens enviadas. A partir de então, a referência da aplicação ou do gerenciador é enviada em cada mensagem.

Do lado do objeto acontece o processo inverso. O ORB é inicializado com a pro-

```
1  public Counter connectToObject() {
2    short numberOfRetries;
3    Counter ref;
4    // Gera o nome CORBA do objeto Counter
5    NameComponent nc = new NameComponent(“Counter”, “”);
6    NameComponent path[] = nc;
7    numberOfRetries = 0;
8    ref = null;
9    while (numberOfRetries < 5) {
10     try {
11       // Recupera referência do objeto
12       ref = CounterHelper.narrow(...);
13       numberOfRetries = 5;
14     } catch (Exception e) {
15       // A exceção pode ter ocorrido devido a uma migração de objetos.
16       // Testa 5 vezes a reconexão
17       try {
18         Thread.currentThread().sleep(1000);
19       } catch (InterruptedException i) {
20         System.err.println(“Exception:”, e);
21       }
22       numberOfRetries++;
23     }
24   }
25   return ref;
26 }
```

Figura 5.20: Laço para obtenção de nova referência de objeto

priedade `ORBInitializerClass.AAServerORBInitializer` e todas as mensagens que chegam são interceptadas (classe `AAServerInterceptor`) na busca da identificação do gerenciador da transação. Um interceptador para retirada da identificação da aplicação, `AAFactoryInterceptor`, é utilizado de maneira similar na Fábrica de Gerenciadores.

5.7 Monitor de Recursos

O monitoramento de recursos tem como base o monitor desenvolvido no SGTA (ver seção 3.9). No entanto, um módulo para monitoramento de migração de célula foi adicionado. A chamada `_notify` foi alterada, uma vez que quem agora deve receber a notificação do monitor é a aplicação (através da subclasse de `AtomicClient`) e não o gerenciador de transações.

O monitoramento de células é utilizado pela aplicação sempre que desejar ser notificada sobre mudanças de célula da máquina móvel. Baseada em notificações sobre mudanças no ambiente (mudança de célula ou variações nos recursos), a aplicação decide sobre a política de migração adequada aos seus requisitos.

A política de adaptação é escolhida no início da transação (ver seção 5.2). Uma chamada a `setMigPolicy` em `AtomicClient` altera essa opção. No método `_notify`, uma chamada à operação `migrate` solicita a migração do módulo adequado (aplicação, gerenciador de transações ou objetos participantes). Um exemplo é mostrado na figura 5.6 no qual a política de adaptação escolhida é migração da aplicação.

5.8 Transações Interativas

No modo de submissão passo-a-passo uma interface gráfica é apresentada ao usuário. Através dessa interface, ele indica quais os objetos que deseja adicionar a uma transação e pode invocar operações sobre eles. A interface também possui um campo para mostrar o resultado da operação.

O monitoramento de recursos também pode ser utilizado por uma transação interativa. Ao ser iniciado, o módulo passo-a-passo executa a instanciação de `AtomicClient` através de um subclasse específica `StepAtomicClient` em que a operação `_notify` não é previamente definida. Ao invés da definição da política, a operação `_notify` simples-

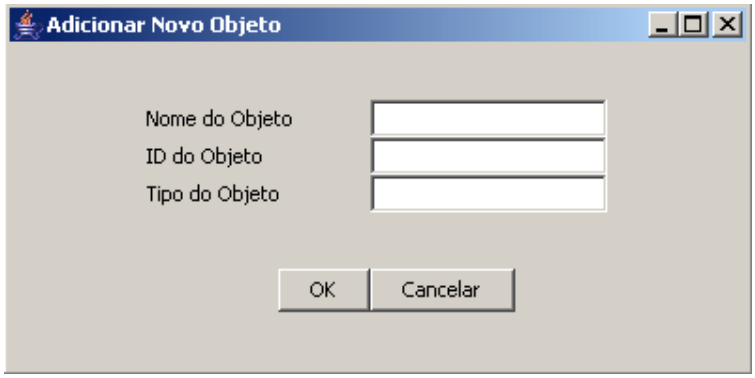

Figura 5.21: Inclusão de novos Objetos Participantes

mente avisa graficamente ao usuário que um recurso ultrapassou o limite especificado. Nesse caso, o usuário pode optar por migrar componentes do sistema.

Para solicitação de um novo objeto participante (ver figura 5.21), o nome do objeto, seu tipo e identificação devem ser especificados. Cada novo objeto é inserido em uma lista de referências para permitir acessos futuros. Essa lista de objetos fica disponível para o usuário que pode selecionar objetos através dela. Após selecionar o objeto, sua lista de operações fica visível e o usuário pode então selecionar uma operação (ver figura 5.22).

Para realizar invocações sobre objetos na transação interativa foi utilizada a interface de invocação dinâmica CORBA [48]. Para obtenção da *interface* de um objeto, uma chamada a `_get_interface_def` definida na *interface* de todos os objetos CORBA é realizada. A chamada retorna um objeto `InterfaceDef` que contém operações para percorrer o Repositório de Interfaces e consultar todos os métodos disponíveis. A implementação JacORB foi utilizada por oferecer essa funcionalidade.

5.9 Exemplos de Aplicação

Considere um hospital com rede sem fio onde médicos podem acessar a base de dados de pacientes através de seus *notebooks*. Os seguintes cenários seriam possíveis:

1. Um médico está em movimento fora do hospital conectado ao sistema através de interface sem fio. Se a largura de banda for baixa ou existir possibilidade de desconexões, uma transação para atualizar prontuários de pacientes poderia mover os dados dos mesmos para a máquina local garantindo, assim, a execução da transação

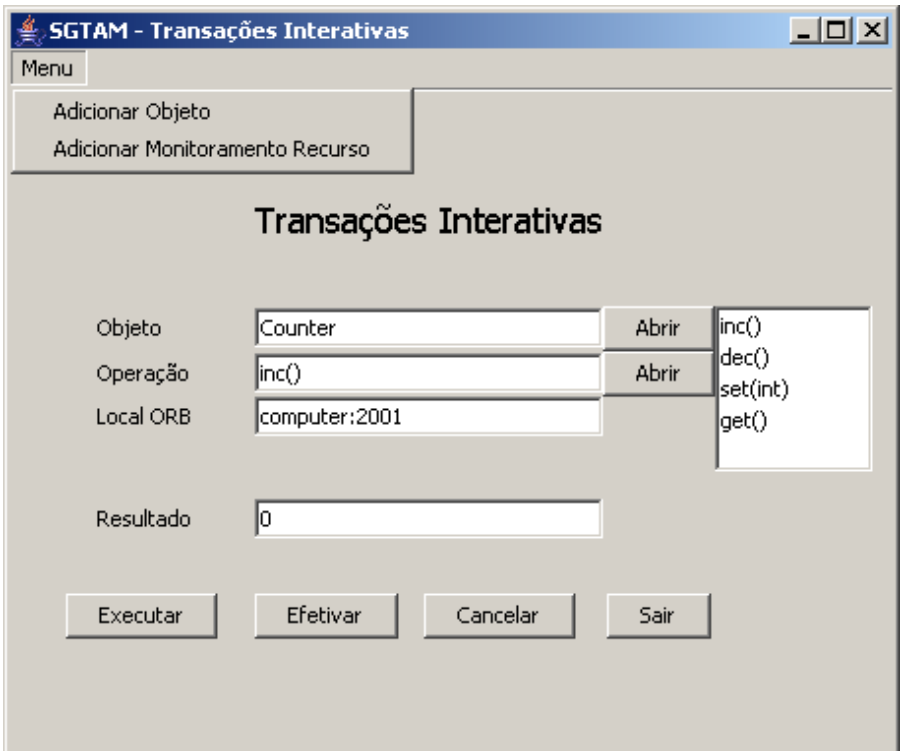

Figura 5.22: Interface gráfica para submissão de Transações Interativas

com sucesso mesmo se ocorrer desconexão. No final da transação (e após restabelecimento de conexão), os dados atualizados seriam movidos de volta ao local de origem.

2. Um médico está no hospital conectado através de rede sem fio mas com melhor qualidade de comunicação e largura de banda. Uma transação para atualizar prontuários de pacientes poderia ser realizada sobre dados remotos através do *notebook*. No final da transação, o gerenciador de transações poderia ser movido para uma máquina da rede fixa onde o protocolo de efetivação fosse executado de forma mais eficiente.

A mobilidade de componentes também é interessante em caso de falta de energia da máquina portátil e ocorrência de invasões (segurança) na máquina onde o gerenciador de transações está em execução. Nesses casos, os componentes podem ser movidos para máquinas disponíveis e seguras.

Outras aplicações que também se beneficiam desse modelo de transações são transações de longa duração envolvendo interatividade e migração de objetos. Entre elas, podemos citar aplicações de turismo e vendedores ambulantes.

Capítulo 6

Conclusões

O modelo de transações proposto nessa dissertação visa atender aos requisitos de aplicações no ambiente de computação móvel. Para oferecer apoio à adaptação ao dinamismo do ambiente, foi acrescentado à transação o parâmetro de mobilidade. Além disso, o requisito de interatividade também foi contemplado através do desenvolvimento de um módulo de transações interativas.

O SGTAM é um modelo de transação em que tanto o gerenciador da transação, como os objetos envolvidos e a própria aplicação são executados como agentes móveis e podem se mover a critério desta.

O primeiro tipo de migração, migração do gerenciador de transação, visa minimizar o custo para realização de operações caras como o protocolo de efetivação em duas fases. No momento da efetivação, a aplicação pode solicitar que o gerenciador migre para uma máquina de rede fixa e proceda com a efetivação da transação nessa máquina.

O segundo tipo, migração de objetos, visa permitir que os mesmos acompanhem a mobilidade da aplicação ficando mais próximos dela a fim de minimizar custos de comunicação. Até mesmo uma migração de objetos da rede fixa para a máquina móvel pode ser solicitada. Nesse caso, o acesso passa a ser local.

O último tipo de migração, migração da aplicação, visa reduzir custos de comunicação entre aplicação e objetos, poupar recursos do dispositivo móvel ou, ainda, permitir que a aplicação execute em uma máquina com algum recurso especial.

Outro ponto considerado é a interatividade das aplicações no ambiente de computação móvel. Para lidar com esse aspecto, foi desenvolvida uma *interface* gráfica para submissão

de operações de uma transação.

6.1 Contribuições

- desenvolvimento de um modelo de transações que incorpora mecanismos de adaptação. Foi proposta uma parceria entre um sistema de apoio responsável por prover mecanismos de adaptação e a aplicação responsável por definir suas políticas de adaptação de acordo com seus requisitos. O mecanismo de adaptação oferecido envolve: mobilidade do gerenciador de transação, dos objetos participantes e da própria aplicação. Não foi encontrado nenhum outro trabalho que utilizasse mobilidade de componentes como forma de adaptação.

- implementação de um protótipo para esse modelo utilizando a linguagem Java; ORB para prover comunicação entre objetos, invocação dinâmica e interceptadores; e a plataforma Grasshopper para possibilitar a criação e execução de agentes móveis.

- implementação de migração forte na linguagem Java.

6.2 Trabalhos Futuros

- a integração entre os mecanismos de mobilidade aqui propostos e os mecanismos de adaptação providos pelo SGTA (modo de operação e nível de isolamento).

- replicação de objetos em diversas máquinas e gerenciamento de consistência entre as réplicas.

- desenvolvimento de um mecanismo para armazenamento de pontos de verificação ao longo da execução da aplicação.

- implementação de tratamento de impasses.

- tratamento de transações que falham: liberação de trancas depois de um intervalo de tempo.

Referências Bibliográficas

[1] George H. Forman e John Zahorjan. The Challenges of Mobile Computing. *IEEE Computer*, 27(4):38–47, Abril 1994.

[2] Tomasz Imielinski e B. R. Badrinath. Mobile Wireless Computing: Challenges in Data Managemen. *Communications of ACM*, 37(10):18–28, 1994.

[3] Randy H. Katz. Adaptation and Mobility in Wireless Information Systems. *IEEE Personal Communications*, 1(1):6–17, 1995.

[4] M. Satyanarayanan. Fundamental Challenges in Mobile Computing. *Fifteenth ACM Symposium on Principles of Distributed Computing*, páginas 1–7, 1996. Philadelphia, PA, EUA.

[5] J. J. Kistler e M. Satyanarayanan. Disconnected Operation in the Coda File System. *Proceedings of the thirteenth ACM symposium on Operating systems principles*, páginas 213–225, 1991. Pacific Grove, California, EUA.

[6] Brian D. Noble, M. Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, e Kevin R. Walker. Agile Application-Aware Adaptation for Mobility. *Sixteen ACM Symposium on Operating Systems Principles*, páginas 276–287, 1997. Saint Malo, França.

[7] Evaggelia Pitoura. Data Management for Mobile Computing. *Summer School on Data Management for Mobile Computing*, 1998. Jyvaskyla, Finlândia.

[8] Margaret H. Dunham e Abdelsalam Helal. Mobile Computing and Databases: Anything New? *ACM SIGMOD Record*, 24(4):5–9, Dezembro 1995.

[9] Jin Jing, Abdelsalam Helal, e Ahmed Elmagarmid. Client-Server Computing in Mobile Environments. *ACM Computing Surveys*, 31(2):117–157, 1999.

[10] Qualcomm. *Eudora Macintosh User Manual*, 1993.

[11] Stan Franklin e Art Graesser. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. *Lecture Notes In Computer Science*, 1193:21–35, 1996.

[12] Robert Gray, David Kotz, Saurab Nog, Daniela Rus, e George Cybenko. Mobile Agents for Mobile Computing. *Dartmouth PCS-TR96-285*, 1996. Department of Computer Science, Dartmouth College, Hanover, NH.

[13] Dejan S. Milojicic, Vana Kalogeraki, Rajan Lukose, Kiran Nagaraja, Jim Pruyne, Bruno Richard, Sami Rollins, e Zhichen Xu. Peer-to-Peer Computing. *HP Survey*, HPL-2002-57R1, 2002.

[14] D. Chess, C. Harrison, e A. Kershenbaum. Mobile Agents: Are they a good idea? *IBM Research Division*, (RC 19887), 1997.

[15] Constantinos Spyrou, George Samaras, Evaggelia Pitoura, e Paraskevas Evtipidou. Mobile Agents for Wireless Computing: The Convergence of Wireless Computational Models with Mobile-Agent Technologies. *Mobile Networks and Applications (MONET)*, 9(5):517–528, 2004. Hingham, MA, EUA.

[16] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, e Gary J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, Janeiro 1997.

[17] G. Samaras e A. Pitsillides. Client/Intercept: a Computational Model for Wireless Environments. *Proceedings of the 4th International Conference on Telecommunications*, Abril 1997. Melbourne, Austrália.

[18] Abraham Silberschatz, Henry F. Korth, e S. Sudarshan. *Database System Concepts*. McGraw-Hill, 5ª edição, 2005.

[19] Philip A. Bernstein, Vassos Hadzilacos, e Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[20] Patricia Serrano-Alvarado, Claudia Roncancio, e Michel Adiba. A Survey of Mobile Transactions. *Distributed and Parallel Databases*, 16(2):193–230, Setembro 2004.

[21] M. H. Dunham, A. Helal, e S. Balakrishnan. A Mobile Transaction Model that Captures Both the Data and the Movement Behavior. *Mobile Networks and Applications*, 2(2):149–162, 1997.

[22] Gerhard Weikum e Hans-J. Schek. *Database Transaction Models for Advanced Applications, capítulo 13 - Concepts and Applications of MultiLevel Transactions and Open Nested Transactions.* Morgan Kaufmann Publishers Inc, 1ª edição, 1992.

[23] Yuri Breitbart, Abraham Silberschatz, e Glenn R. Thompson. Transaction Management Issues in a Failure-Prone Multidatabase System Environment. *VLDB Journal: Very Large Data Bases*, 1(1):1–39, 1992.

[24] C. Pu e G. E. Kaiser. Split-Transactions for Open-Ended Activities. *Proceedings of the 14th Conference on Very Large Databases*, 1988. Los Angeles, CA, EUA.

[25] Patricia Serrano-Alvarado, Claudia L. Roncancio, Michel Adiba, e Cyril Labbé. An Adaptable Mobile Transaction Model for Mobile Environments. *Int. Journal of Computer Systems Science and Engineering (IJCSSE), Special Issue on Mobile Databases*, 20(3), Abril 2005.

[26] A. Zhang, M. Nodine, B. Bhargava, e O. Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, páginas 67–78, 1994. Minneapolis, Minnesota, EUA.

[27] Nuno Santos, Luis Veiga, e Paulo Ferreira. Transaction Policies for Mobile Networks. *Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, 2004. IBM Thomas J Watson Research Center, Yorktown Heights, Nova York, NY, EUA.

[28] B. Bhargava. Concurrency Control in Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):3–16, Janeiro 1999.

[29] F. M. Assis Silva e R. Popescu-Zeletin. Mobile Agent-based Transactions in Open Environments. *EICE Transactions on Communications*, E83-B(5):973–987, Maio 2000.

[30] Takao Komiya, Tomoya Enokido, e Makoto Takizawa. Mobile Agent Model for Transaction Processing on Distributed Objects. *Information Sciences - Informatics and Computer Science*, 154(1-2):23–38, 2003.

[31] Tarcisio da Rocha. Um Sistema de Transações Adaptável para o Ambiente de Comunicação Sem Fio. Tese de Mestrado, Instituto de Computação, UNICAMP, 2004.

[32] J. Gray, R. Lorie, F. Putzolu, e I. Traiger. *Granularity of Locks and Degrees of Consistency in a Shared Database, Readings in Database Systems, páginas 175-193.* Morgan Kaufmann Publishers Inc, 3ª edição, 1998.

[33] American National Standards Institute ANST. *American National Standard for Information Systems: Database Language - SQL: ANSI X3.135-1992.* American National Standards Institute, 1992.

[34] C. J. Date. *Introdução a Sistemas de Bancos de Dados.* Campus, 8ª edição, 2004.

[35] Giovanni Bogéa Viana e Maria Beatriz Felgar de Toledo. Agentes no Gerenciamento de Transações Móveis. *Fourth International Information and Telecommunication Technologies Symposium (I2TS)*, Dezembro 2005. Florianópolis, SC, Brasil.

[36] Software Engineering and Systems Software Group SESSG e Xtradyne Technologies AG Xtradyne. Jacorb - the Free Java implementation of the OMG's CORBA standard. http://www.jacorb.org, acessado em 30/09/05.

[37] Object Management Group Inc (OMG). CORBA. http://www.corba.org, acessado em 21/11/05.

[38] Steve Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 14(2), Fevereiro 1997. Washington, DC, EUA.

[39] Dan Harkey e Robert Orfali. *Client/Server Programming with Java and CORBA.* John Wiley & Sons, 2ª edição, 1998.

[40] IKV++ GmbH Informations, Bernburger Strasse 24-25, 10963 Berlim, Alemanha. *Grasshopper Basics and Concepts*, 1ª edição, 2001.

[41] Torsten Illmann e Frank Kargl. Migration of Mobile Agents in Java: Problems, Classification and Solutions. *International Symposium on Multi-Agent and Mobile Agents in Virtual Organizations and E-Commerce (MAMA)*, Junho 2000. Wollongong, Austrália.

[42] Sun Microsystems Sun. Java Platform Debugger Architecture. http://java.sun.com/products/jpda/index.jsp, acessado em 30/09/05.

[43] Torsten Illmann, Tilman Krueger, Frank Kargl, e Michael Weber. Transparent Migration of Mobile Agents using the Java Platform Debugger Architecture. *Lecture Notes in Computer Science*, 2240:198, Janeiro 2001.

[44] M. Damm. Byte Code Engineering. *Proceedings of JIT'99*, 1999. Duesseldorf, Alemanha.

[45] Stefan Fünfrocken. Transparent Migration of Java-based Mobile Agents. *Lecture Notes in Computer Science*, (1477):26–37, 1998.

[46] Sreeni Viswanadha. Java Compiler Compiler. https://javacc.dev.java.net/, acessado em 19/10/05.

[47] Michi Henning. Binding, migration, and scalability in corba. *Communications of the ACM*, 41(10):62–71, Outubro 1998.

[48] Gerald Brose, Andreas Vogel, e Keith Duddy. *JAVA Programming with CORBA*. Wiley, 3ª edição, 2001.