**Um Sistema de Transações Adaptável para o Ambiente de Comunicação Sem Fio**

*Tarcisio da Rocha*

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

# Um Sistema de Transações Adaptável para o Ambiente de Comunicação Sem Fio

Tarcisio da Rocha

10 de Dezembro de 2003

**Banca Examinadora:**

- Dra. Maria Beatriz Felgar de Toledo (Orientadora)
  IC/ UNICAMP – Universidade Estadual de Campinas
- Dr. Edmundo Roberto Mauro Madeira
  IC/ UNICAMP – Universidade Estadual de Campinas
- Dr. Flávio Morais de Assis Silva
  UFBA – Universidade Federal da Bahia
- Dra. Islene Calciolari Garcia (Suplente)
  IC/ UNICAMP – Universidade Estadual de Campinas

# Um Sistema de Transações Adaptável para o Ambiente de Comunicação Sem Fio

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

Campinas, 10 de Dezembro de 2003.

Dra. Maria Beatriz Felgar de Toledo
IC/UNICAMP – Universidade Estadual de Campinas (Orientadora)

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

# Resumo

Recentes avanços nas tecnologias de telecomunicação e de dispositivos de computação portáteis possibilitaram o surgimento da Computação Móvel. Assim, dispositivos como laptops e PDA's equipados com interfaces de comunicação sem fio ganharam a capacidade de participar de computações distribuídas mesmo estando em movimento ou desconectados. Apesar da computação móvel ser muito atraente, ela impõe uma série de obstáculos aos sistemas distribuídos. Dentre estes obstáculos está a baixa largura de banda das redes de comunicação sem fio, desconexões inesperadas e escassez de recursos do dispositivo de computação móvel (como energia e espaço em disco).

Diante deste contexto, esta dissertação apresenta um sistema de transações que possui a capacidade de adaptação diante dos obstáculos impostos pelo ambiente de comunicação sem fio. Esta adaptação é obtida através da colaboração entre as transações e um sistema de apoio. O sistema de apoio é responsável pelo monitoramento dos recursos do ambiente, por notificar as transações sobre as mudanças ocorridas no ambiente e por prover mecanismos de adaptação às transações. Cada transação pode reagir às mudanças ocorridas no ambiente de acordo com a sua política de adaptação. Basicamente, os mecanismos de adaptação providos pelo sistema são os modos de operação e os níveis de isolamento.

Para a validação do modelo proposto, esta dissertação apresenta também um protótipo que foi implementado usando as tecnologias Java e CORBA.

# Abstract

Recent advances in telecommunication technologies and portable computing devices have made possible the appearance of Mobile Computing. Thus devices like laptops and PDAs equipped with wireless communication interfaces have acquired the capacity to participate in distributed computing while moving or disconnected. Although attractive, mobile computing imposes a set of obstacles to distributed computing. Among these obstacles is the low bandwidth of the wireless communication networks, unexpected disconnections, and lacking of resources in mobile computing devices (such as power and disk space).

In this context, this dissertation presents a transaction system that can adapt in the face of the obstacles imposed by the wireless communication environment. This adaptation is acquired through the collaboration between transactions and the underlying system. The underlying system is responsible for monitoring of resources, notifying transactions about changes in the environment and providing adaptation mechanisms. Each transaction may react to environmental changes according to its own adaptation policy. The provided adaptation mechanisms are operation modes and isolation levels.

Moreover, this dissertation presents a prototype developed for the validation of the proposed model. The prototype was implemented using Java and CORBA technologies.

# Agradecimentos

A Deus pelo dom da vida e do amor, por minha saúde física e mental e pela minha liberdade, requisitos por Ele concedidos e que me torna possível cada passo deste caminhar.

À minha namorada, Shirley, primeiramente por achar que o título de "namorada" seria muito pouco diante da sua diária, alegre e marcante presença na minha vida no decorrer de todo deste curso. Com ela tanto o meu trabalho quanto o meu lazer permeou-se com o mais completo das essências: a do Amor! Essa é a Minha Morena!

À minha orientadora Maria Beatriz por ter compartilhado comigo a sua experiência e sabedoria no cumprimento das etapas deste curso. A sua humana presença foi tranqüilizadora diante das turbulências normalmente vividas no ambiente acadêmico.

A Moacir por todo companheirismo e parcerias em disciplinas do curso e por ter me agüentado como colega de quarto com as minhas bagunças, sem contar que ele suportou comer o macarrão com frango à água e sal que eu preparava todo Domingo. Valeu, meu irmão!

A todos os colegas de turma que eu lembro com alegria, pelos momentos juntos e pelas dificuldades enfrentadas.

A todos os professores e funcionários do Instituto de Computação que trabalharam direta ou indiretamente na minha formação. Em especial quero agradecer a professora Eliane por ter me conduzido no processo inicial de adaptação ao curso.

Ao meu pai José (que já está no céu) e a minha mãe Carolina por não terem medido esforços, pela total doação em busca da minha felicidade. Para mim eles são literalmente um exemplo de vida que quero seguir. Que Deus vos conceda toda bênção e proteção!

A todos os meus familiares, Tia Ana pela preocupação com a minha saúde, Avó Cantionila pela força da oração, a Sr. Teles e D. Simonete por todo o apoio e à minha lista de irmãos dos quais me orgulho muito, Adriano, Marcos, Suzy, Miguel, Rita, Sérgio, Lucas, Raquel e Raimundo por todo apoio e por compartilharem comigo desta alegria multiplicando-a.

A todos meus amigos de Aracaju que sempre torceram por mim.

# Sumário

# Lista de Tabelas

# Lista de Figuras

# Capítulo 1

# Introdução

Neste capítulo será apresentada uma breve introdução do trabalho que será descrito nesta dissertação. Esta introdução será dividida em: Motivação (Seção 1.1); Objetivos (Seção 1.2); e Estrutura da Dissertação (Seção 1.3).

## 1.1 Motivação

Recentes avanços nas tecnologias de telecomunicação e de dispositivos de computação portáteis possibilitaram o surgimento da Computação Móvel. Assim, dispositivos como laptops e PDA's equipados com interfaces de comunicação sem fio ganharam a capacidade de participar da computação distribuída mesmo estando em movimento ou desconectados.

Apesar da computação móvel prover características de mobilidade bastante atraentes, a sua infra-estrutura possui alto grau de dinamismo que não era previsto pelos sistemas distribuídos tradicionais. Variações na disponibilidade de recursos, na conectividade da rede, nas plataformas de hardware e software são características do ambiente móvel que podem inviabilizar a execução de aplicações caso não sejam devidamente consideradas. Mais especificamente, as aplicações que executam no dispositivo móvel terão que enfrentar problemas, como a variável largura de banda da comunicação sem fio, a possibilidade de desconexões freqüentes e inesperadas, conexões com redes heterogêneas, além de problemas inerentes ao dispositivo móvel, como energia e espaço em disco limitados [16, 24, 21, 51]. Estes problemas herdados da computação móvel fizeram com que inúmeros estudos fossem realizados com o objetivo de viabilizar a execução de aplicações neste novo ambiente.

Operações de acesso a dados remotos são muito comuns em sistemas distribuídos. Para evitar que a concorrência e as falhas de rede sejam prejudicais às aplicações, elas podem utilizar mecanismos como transações. Vários modelos de transações vêm sendo desenvolvidos para ambientes distribuídos tradicionais. Porém, devido às restrições do ambiente de computação móvel, estes modelos de transação tradicionais não são adequados ao ambiente de computação móvel. Isto se deve ao fato que modelos de

transação para sistemas distribuídos não levam em conta o alto dinamismo do ambiente móvel.

Para possibilitar que aplicações executem no ambiente de computação móvel, o projeto Odyssey [33] utilizou uma técnica que se mostrou muito eficaz chamada adaptação colaborativa. Nesta técnica, o processo de adaptação de uma aplicação ao dinamismo do meio é feito através de um padrão de colaboração entre a aplicação e um sistema de apoio. O sistema de apoio proveria monitoramento do ambiente móvel e os mecanismos de adaptação enquanto que a aplicação especificaria a sua própria política de adaptação. Desta forma, uma aplicação passa a ter a capacidade de conhecer as mudanças ocorridas no ambiente móvel, bem como de decidir como melhor se adaptar às mudanças utilizando mecanismos providos pelo sistema de base.

Como exemplo de uma aplicação que é apoiada pelo Odyssey, suponhamos um navegador de acesso a páginas web que executa no dispositivo móvel e carrega as páginas em html com imagens através de uma conexão sem fio. Suponhamos também que em um determinado instante esta aplicação é notificada de que a largura de banda da conexão foi reduzida consideravelmente. Para se adaptar a esta mudança, a aplicação poderá pedir ao sistema Odyssey para reduzir a qualidade das imagens (em pixels/cm) da página html antes da transmissão. Páginas com imagens de menor qualidade são menores em bytes sendo então mais adequados para serem transferidas enquanto a largura de banda for baixa.

Projetos como o Odyssey que utilizam técnicas de adaptação colaborativa foram muito empregados em aplicações que operam sobre dados multimídia, como áudio, vídeo, imagens, páginas html. Isto se deve ao fato de que este tipo de dados possui uma certa riqueza de manipulação, eles possuem uma noção clara de qualidade que pode ser manipulada a fim de reduzir o seu tamanho em bytes. Porém, há uma carência de estudos de técnicas que permitam a utilização da adaptação colaborativa em aplicações que não utilizam dados multimídia como, por exemplo, dados numéricos.

## 1.2 Objetivos

Levando-se em conta a importância que têm os sistemas de transações para ambientes distribuídos, o foco desta dissertação está na investigação de mecanismos de adaptação que possibilitem a cada transação executar no ambiente de comunicação sem fio sujeito a falhas de comunicação e escassez de recursos no dispositivo móvel. Estes mecanismos serão concebidos através do uso da técnica de adaptação colaborativa. Esta colaboração será dada entre a transação e um sistema de base onde este fornece serviços de adaptação enquanto que a transação determina a sua política de adaptação. Este trabalho parte do pressuposto de que cada transação, por ser uma unidade independente de execução, deve possuir a sua própria política de adaptação. Esta política seria implementada de modo a configurar a transação para enfrentar as turbulências do ambiente móvel da maneira que lhe fosse mais adequada.

É através do uso deste padrão de colaboração que será dado a uma transação uma maior disponibilidade dos dados distribuídos acessados e uma maior capacidade de

garantia das propriedades ACID[1] mesmo diante da instabilidade do ambiente de computação móvel. Para isso serão utilizadas técnicas de caching, validação e flexibilização do controle de concorrência.

## 1.3 Estrutura da Dissertação

Esta dissertação está estruturada como segue:

- O Capítulo 2 apresenta alguns conceitos básicos que serão utilizados no decorrer da dissertação;
- O Capítulo 3 apresenta uma visão geral do modelo de transações proposto, descreve a arquitetura do sistema de transações adaptável e apresenta os mecanismos de adaptação dos quais uma transação pode se utilizar;
- O Capítulo 4 apresenta aspectos da implementação do protótipo e de como configurar e utilizar transações no protótipo;
- O Capítulo 5 apresenta trabalhos relacionados destacando-se a sua relevância na concepção do modelo apresentado nesta dissertação;
- O Capítulo 6 finaliza a dissertação apresentando as conclusões, contribuições da dissertação e trabalhos futuros.

[1] Acrônimo que significa: Atomicidade, Consistência, Isolamento e Durabilidade.

# Capítulo 2

# Conceitos Básicos

Este Capítulo introduz os conceitos básicos relacionados a Computação Móvel e a Transações. Também será apresentado um breve resumo sobre o padrão CORBA [37] e sua respectiva implementação JavaIDL [22] que foi utilizada no desenvolvimento do protótipo. Por fim, este Capítulo apresentará alguns requisitos de sistemas de transações em ambientes de comunicação sem fio.

## 2.1 Computação Móvel

A computação móvel é um paradigma que permite a computadores portáteis equipados com interfaces de comunicação sem fio participarem da computação distribuída independentemente da sua localização física ou características de 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 [21, 45, 13]. Este modelo é composto por componentes fixos e componentes móveis (unidades móveis). Uma *Unidade Móvel* é um computador móvel capaz de se comunicar com a rede fixa através de um meio de comunicação sem fio. Os componentes fixos da rede são máquinas fixas ou estações de apoio. Estes componentes se comunicam entre si através de uma rede fixa de alta velocidade. Uma *Máquina Fixa* é um computador da rede fixa que não possui interface de comunicação sem fio. Uma *Estação de Apoio* é capaz de se conectar a uma unidade móvel através de um meio de comunicação sem fio. As estações de apoio agem como interfaces entre as unidades móveis e os componentes da rede fixa. Cada estação de apoio possui uma área de cobertura chamada *Célula Sem Fio*. Esta é a área por onde uma unidade móvel pode se mover e manter a conexão com a estação de apoio correspondente. Ao se mover, uma unidade móvel poderá sair de uma célula e entrar em uma outra e assim manter a comunicação com a rede fixa.

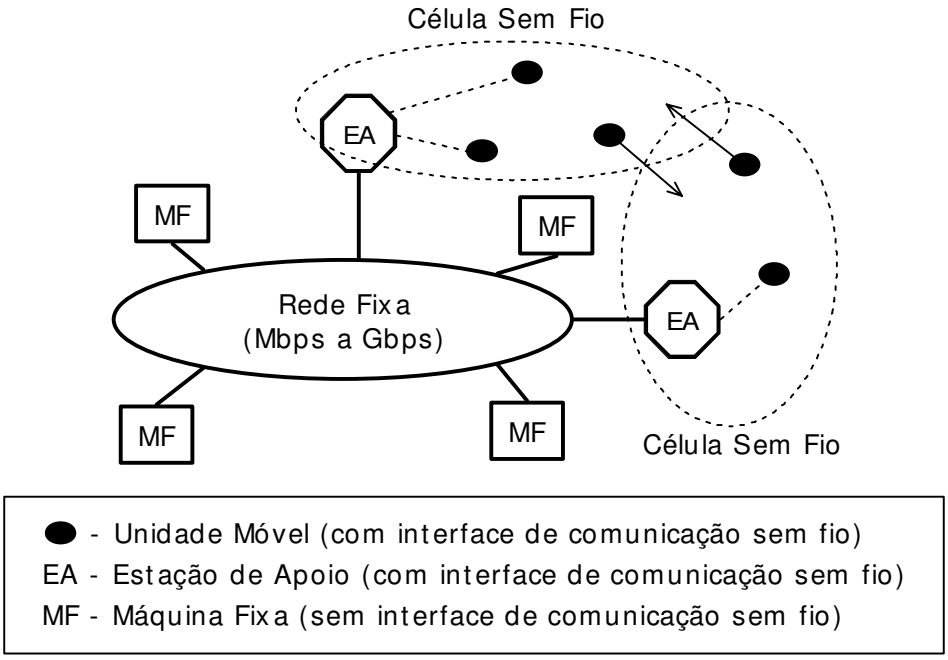



Figura 2.1: Uma arquitetura de apoio à computação móvel.

### 2.1.1 Obstáculos da Computação Móvel

A computação móvel tem exigido várias mudanças no desenvolvimento de soluções no que diz respeito a circuitos integrados, processamento de sinais, projeto de rede e projeto de sistemas computacionais. Essas mudanças são necessárias porque a comunicação móvel sem fio enfrenta muito mais obstáculos do que a comunicação em redes tradicionais "com fios". [16, 24, 21, 51] 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*[2] e *escrita com atraso*[3]. A *busca antecipada* é a ação de fazer uma cópia dos prováveis objetos que

[2] Em inglês: prefetching.

[3] Em inglês: delayed write-back.

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 pode ser o endereço de rede do computador móvel que poderá mudar cada vez que houver uma mudança no ponto de acesso deste com a rede fixa.

### 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 e a rede fixa que poderá variar entre três estados básicos (ver Figura 2.2): a desconexão; a conexão com baixa largura de banda (conexão fraca); e a conexão com alta largura de banda (conexão forte). Normalmente, 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 até mesmo Mbps ou Gbps.

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 tomar vantagens de um recurso em abundância. [23] categoriza os mecanismos de adaptação em três paradigmas:

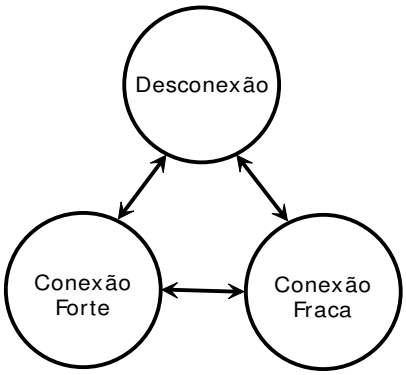



Figura 2.2: Estados da conexão entre a unidade móvel e a rede fixa.

**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.

**Adaptação transparente.** Os obstáculos do ambiente móvel deverão ser contornados por um sistema de base. 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 de base 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 às quais o sistema de base dá apoio possuem diferentes necessidades de adaptação. Exemplos desta abordagem podem ser vistos nos projetos Coda [26, 52] e Little Work [19].

**Adaptação colaborativa.** Os obstáculos da ambiente móvel são contornados através da colaboração entre o sistema de base e as aplicações. Esta abordagem propõe que o sistema de base 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. Exemplos desta abordagem podem ser vistos nos projetos Odyssey [33], Cadmium [6] e Prayer [10].

## 2.2 Transações

Informalmente, uma transação é um conjunto de operações que transforma o estado consistente de uma base de dados em um outro estado consistente. Um processo que utiliza uma transação normalmente contém as seguintes primitivas:

**begin_transaction e end_transaction.** Delimitam o início e o fim de uma transação, respectivamente.

**abort_transaction.** Cancela a transação restaurando o estados dos dados modificados por ela para o estado anterior à sua execução.

**commit_transaction.** Esta operação é chamada no final da transação para indicar que a sua execução foi bem sucedida e que os resultados obtidos devem se tornar permanentes.

As primitivas *begin_transaction* e *end_transaction* são delimitadores de operações do tipo *read* ou *write* que compõem o processo a ser executado. Uma operação é do tipo *read* quando ela não modifica o estado da base de dados. Em contrapartida, uma operação é do tipo *write* quando ela modifica o estado da base de dados.

### 2.2.1 Propriedades de Transações

As propriedades de uma transação atômica podem ser resumidas no acrônimo ACID, que significa Atomicidade, Consistência, Isolamento e Durabilidade. A seguir é dada uma definição para cada um destes elementos:

**Atomicidade.** Ou todas ou nenhuma das operações da transação será executada. Em caso de falha, o efeito de cada operação de uma transação será desfeito.

**Consistência.** Uma transação só possui permissão para ler valores consistentes e também só disponibiliza valores consistentes. Uma transação poderá normalmente manter estados intermediários inconsistentes durante a sua execução, porém estes resultados intermediários não serão visíveis a outras transações.

**Isolamento.** Os resultados de uma transação não deverão sofrer interferências de outras transações concorrentes. Se duas ou mais transações estão executando concorrentemente e de forma isolada, o mesmo resultado final poderá ser obtido executando as mesmas transações de forma serial, uma após a outra em alguma ordem.

**Durabilidade.** Uma vez que uma transação efetue a operação *commit_transaction*, seus resultados serão feitos permanentes. Nenhuma falha depois desta operação poderá desfazer seus resultados.

A partir da propriedade de isolamento de uma transação nasce o conceito de *serialização*[4]. Um conjunto de transações possui uma execução serializável quando o resultado das suas execuções é equivalente a uma execução serial das mesmas. Por sua vez, uma execução é serial quando para cada par de transações, todas as operações de uma transação executam antes de qualquer operação da outra transação. A serialização é um requisito altamente usado em sistemas que requerem um rigoroso grau de garantia de consistência de suas operações. Uma definição formal de serialização é apresentada em [9].

Existem modelos apresentados, por exemplo, em [46, 29, 3] que relaxam algumas das propriedades ACID tornando-os mais flexíveis e adequados às exigências de outros tipos de aplicações.

### 2.2.2 Controle de Concorrência

Quando duas ou mais transações intercalam seus acessos a uma base de dados, o resultado de suas execuções poderá sofrer interferências indesejáveis. O mecanismo que evita estas interferências é chamado *controle de concorrência*.

Um meio de evitar que uma transação interfira na execução da sua concorrente é através da atribuição de trancas[5] de leitura ou de escrita sobre dados acessados pelas transações. Se uma transação *T* precisar acessar um dado *x* para somente leitura, deverá ser atribuída a *x* uma tranca de leitura em favor de *T*. Se *T* precisar acessar um dado *x* para escrita, deverá ser atribuída uma tranca de escrita a *x* em favor de *T*. Quando uma tranca *tr* é atribuída a um dado *x*, uma outra transação *T'* só poderá atribuir uma tranca *tr'* a *x* se *tr'* não for conflitante com *tr*. Caso contrário, a transação *T'* será bloqueada até que as trancas atribuídas a *x* incompatíveis com *tr'* sejam liberadas. A Tabela 2.1 mostra a relação de conflito entre trancas de leitura e de escrita.

| | **LEITURA** | **ESCRITA** |
|---|---|---|
| **LEITURA** | Sem conflito | Conflito |
| **ESCRITA** | Conflito | Conflito |

Tabela 2.1: Relação de conflito entre trancas

Para que a estratégia de atribuição de trancas possa evitar interferências entre duas ou mais transações concorrentes, deve ser usado um algoritmo de controle de concorrência. Um dos algoritmos mais usados para controle de concorrência em transações é o de

[4] Em inglês: serializability.

[5] Em inglês: lock.

*bloqueio em duas fases*[6]. Este algoritmo faz com que uma transação adquira todas as trancas que ela necessita durante a chamada fase de crescimento, e depois os libere no final. [9] prova de maneira simples que transações que executam sob o controle de concorrência bloqueio em duas fases são serializáveis.

O bloqueio em duas fases é considerado um método de controle de concorrência pessimista, ou seja, evita inconsistências antes mesmo que elas aconteçam. Já os métodos de controle de concorrência otimistas [17] permitem que as inconsistências aconteçam para só depois resolvê-las.

## 2.3 CORBA

O CORBA (*Common Object Request Broker Architecture*) [27, 28] é uma arquitetura aberta desenvolvida pela OMG [38] que provê uma interface comum de interoperabilidade entre componentes de software independentemente de linguagem de programação, plataforma, sistema operacional e localização. A OMG (*Object Management Group*), criada em 1989, é hoje formada por cerca de 800 empresas que unem esforços na definição de padrões que possibilitem a interoperabilidade de componentes de software em um ambiente distribuído.

Para que a interoperabilidade proposta pelo CORBA se tornasse possível, a OMG definiu uma linguagem chamada IDL (Interface Definition Language) [37]. Esta linguagem é usada para a definição das interfaces pelas quais um objeto cliente pode requisitar operações de outros objetos que provêem as implementações das mesmas. IDL é considerada uma linguagem neutra, já que interfaces definidas em IDL podem ser compiladas em objetos implementados em qualquer linguagem em que a IDL possa ser mapeada. Atualmente, a IDL pode ser mapeada para as seguintes linguagens: C, C++, Java, Smalltalk, ADA, COBOL, LISP, Python e IDLscript. Este mapeamento é normalmente feito por compiladores específicos para cada linguagem.

Observando a Figura 2.3, a compilação do IDL gera basicamente *stubs* para o lado cliente e *skeletons* para o lado servidor (onde se encontra a implementação da interface). Os *stubs* e os *skeletons* servem como representantes (*proxies*) para clientes e servidores, respectivamente. Os clientes utilizam *stubs* para invocar operações remotas do servidor. Um *stub* é responsável por propagar a invocação recebida para o skeleton correspondente. Este, por sua vez, é encarregado de propagar a invocação para o objeto do servidor que implementa a interface. Para a comunicação entre cliente e servidor foi implementado um protocolo para redes baseadas no protocolo TCP/IP chamado IIOP (*Internet Inter-ORB Protocol*).

A especificação CORBA [37] define a arquitetura de um ORB (*Object Request Broker*). Um ORB é responsável por todo o mecanismo necessário para localizar a implementação do objeto invocado, por preparar a implementação do objeto para receber a

---

[6] Em inglês: two-phase lock.

invocação e por comunicar os dados que compõem os parâmetros da invocação. A interface que o cliente acessa é totalmente independente de onde o objeto está localizado e da linguagem de programação em que ele está implementado.

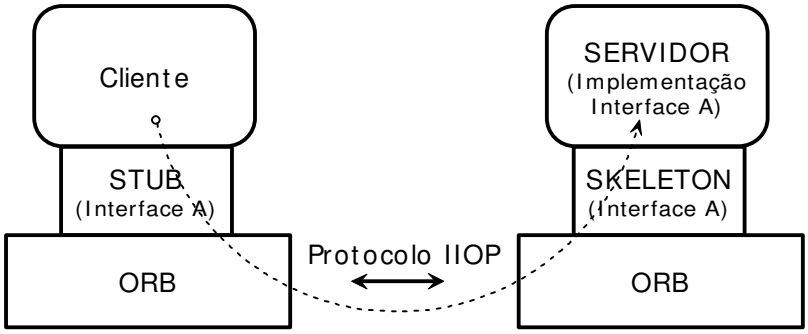



Figura 2.3: Uma invocação passando do cliente para o servidor.

Existem várias implementações da especificação CORBA. Dentre estas está o *OrbixWeb* da IONA [20], o *Java IDL* da Sun [53] e o *VisiBroker* da Borland [11].

### 2.3.1 Java IDL

O Java IDL [22] é uma implementação CORBA da Sun [53] segundo as especificações da OMG. Com ele o ambiente Java ganha o poder da interoperabilidade do CORBA. O Java IDL é parte integrante da plataforma Java *J2SE 1.4 (Java™ 2 Platform, Second Edition)* que oferece as seguintes facilidades:

**ORB.** Object Request Broker baseado em Java com apoio do protocolo IIOP.

**Idlj.** Compilador que gera código Java como *stubs* e *skeletons* a partir de um arquivo IDL.

**Adaptador Portátil de Objetos.** Permite a programadores construírem implementações de objetos que são portáveis entre diferentes ORBs.

**Interceptadores Portáteis.** São ganchos no ORB através dos quais é possível interceptar o fluxo normal de execução de um ORB.

**General Inter-ORB Protocol 1.2.** É um protocolo de transporte que define as bases para o IIOP (Internet Inter-ORB Protocol).

**Object Request Broker Deamon.** É uma ferramenta usada por clientes para localizar e invocar objetos no servidor de forma transparente. A ORBD é composta por

componentes como: Serviço de Nomes Persistente, Serviço de Nomes Transiente e Gerenciador de Servidor.

**Server Tool.** Ferramenta que provê uma interface simples de uso para programadores de aplicação registrar, iniciar e finalizar um servidor.

A implementação do protótipo deste trabalho utilizou o ORB Java IDL por oferecer este conjunto de funcionalidades, por ser obtido gratuitamente no site da Sun incluso na plataforma J2SE e pela disponibilidade de documentação.

## 2.4 Requisitos de Sistemas de Transações em Ambientes de Comunicação sem Fio

Sistemas de transações visam a garantia de um conjunto de propriedades como, por exemplo, as propriedades ACID. Para que um sistema de transações garanta estas propriedades, uma série de medidas é tomada. A implementação destas medidas normalmente é feita levando-se em consideração que o sistema executa em uma rede local caracterizada por uma boa estabilidade, baixa latência e alta largura de banda da comunicação.

Em um ambiente de computação móvel, um sistema de transações não pode mais contar com os recursos providos por uma rede local. Devido às restrições impostas pelo ambiente móvel, o sistema de transações tem que utilizar estratégias que viabilizem a sua execução neste ambiente instável, de baixa largura de banda e sujeito a desconexões freqüentes e inesperadas. Tendo em vista estas considerações, a seguir serão apresentados alguns requisitos necessários para que um sistema de transações possa executar no ambiente de computação móvel:

**Caching de dados.** Uma transação em um dispositivo móvel que acessa dados remotos via comunicação sem fio está sujeita a desconexões inesperadas. Para permitir que esta transação continue a executar enquanto desconectada do restante da rede faz-se necessário copiar para a unidade móvel os dados necessários para execução da transação.

**Validação.** Uma transação que executa na unidade móvel normalmente pode vir a acessar dados em cache que na verdade são cópias dos dados localizados em máquinas remotas. Esta pode ser uma boa estratégia para sobrepor problemas da comunicação com o restante da rede. Porém, quando uma transação utiliza dados copiados em cache, ela poderá estar acessando cópias que são em algum momento inconsistentes em relação às suas correspondentes remotas. Como uma transação requer um grau de consistência de suas operações, deverá existir algum mecanismo de validação que permita verificar a consistência das operações realizadas localmente pela mesma.

**Controle de concorrência.** Apesar de existirem diversas formas de controle de concorrência que podem ser empregadas em um sistema de transações, pode-se dizer que grande parte destes métodos é baseada em trancas. Com a utilização de trancas, uma transação poderá bloquear o acesso a um determinado dado de forma que outras transações não consigam acessá-lo até que a transação responsável libere a respectiva tranca. No ambiente de computação móvel poderão existir transações executando em unidades móveis que atribuem trancas a dados remotos. Pode acontecer de, durante a execução destas transações, haver queda de conexão das suas unidades móveis com o restante da rede. Com isto, torna-se possível que aquelas trancas atribuídas aos dados remotos nunca mais sejam liberadas pelas transações responsáveis. Para evitar este tipo de problema, sistemas de transações que operam em ambientes móveis devem prover mecanismos de liberação de trancas perdidas por transações que perderam o contato com o restante da rede. Além disso, são necessários mecanismos de controle de concorrência mais flexíveis para que as transações que executam no ambiente de computação móvel não tenham o desempenho comprometido pela perda de concorrência.

# Capítulo 3

# Sistema de Gerenciamento de Transações Adaptável

Neste capítulo, será apresentado o Sistema de Gerenciamento de Transações Adaptável (SGTA) proposto neste trabalho. Alguns conceitos utilizados serão baseados na orientação a objetos.

Na Seção 3.1 será apresentada uma visão geral do sistema, na Seção 3.2 será mostrada a arquitetura do sistema, a Seção 3.3 apresenta alguns mecanismos de adaptação e, para finalizar, a Seção 3.4 discute políticas de adaptação.

## 3.1 Visão Geral

Como foi apresentado no Capítulo 2, o ambiente de computação móvel possui uma série de problemas que podem afetar aplicações que executam na unidade móvel. Em termos práticos, pode-se citar alguns exemplos de problemas: os suprimentos de energia da unidade móvel podem se esgotar; o espaço em disco da unidade móvel pode se tornar insuficiente; a largura de banda da conexão com a rede fixa poderá sofrer variações bruscas; podem acontecer desconexões inesperadas. O Sistema de Gerenciamento de Transações Adaptável (SGTA) [47, 48, 49] que será apresentado provê a transações que executam em unidades móveis a capacidade de se adaptar a problemas como estes existentes no ambiente de computação móvel (ver Figura 3.1).

O modelo de adaptação empregado no SGTA é a **adaptação colaborativa**. Neste modelo, as responsabilidades do processo de adaptação são divididas entre as transações e um sistema de apoio:

**Sistema de Apoio.** É responsável por monitorar os recursos do ambiente que podem sofrer variações como, por exemplo, a energia da unidade móvel e a largura de banda. O sistema de apoio também é responsável por notificar as transações das variações nos recursos do ambiente móvel quando solicitado e por prover mecanismos de adaptação às transações.

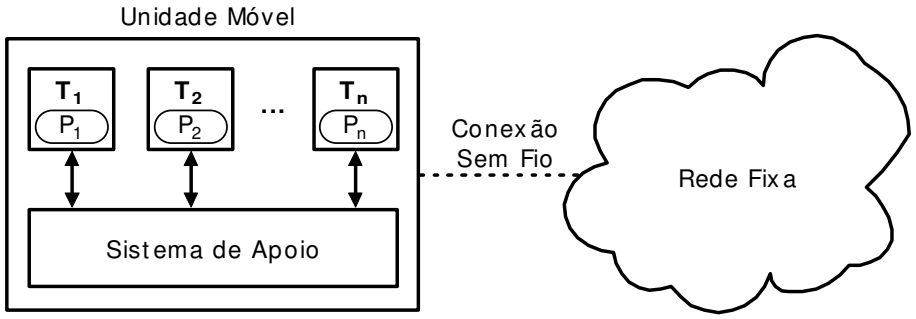



Figura 3.1: Visão geral do SGTA.

**Transações.** São individualmente responsáveis por solicitar ao sistema de base o monitoramento de cada recurso do ambiente móvel de acordo com seus interesses e por definir suas políticas de adaptação a serem adotadas em reação às mudanças na disponibilidade de recursos no ambiente móvel.

Esta estratégia de adaptação segue basicamente a seguinte seqüência de passos:

**Passo 1.** A transação requisita o monitoramento de um ou mais recursos de seu interesse ao Sistema de Apoio. Nesta requisição, a transação especifica os limites superior e inferior toleráveis para cada recurso.

**Passo 2.** O Sistema de Apoio registra a requisição da transação e passa então a monitorar cada recurso requisitado. Se o nível de um determinado recurso requisitado extrapolar os limites especificados pela transação, esta receberá uma notificação por parte do Sistema de Apoio.

**Passo 3.** Ao receber uma notificação do Sistema de Apoio, a transação poderá então tomar medidas reativas de adaptação à mudança de acordo com políticas próprias de adaptação.

**Passo 4.** A transação pode voltar ao Passo 1 e especificar os novos limites toleráveis para cada recurso de seu interesse.

Este modelo de adaptação permite que uma transação possa tomar ciência das variações ocorridas nos recursos de seu interesse possibilitando que elas mesmas possam decidir como melhor se adaptar. Estas decisões de adaptação são definidas na política de adaptação da transação.

## 3.2 Arquitetura do SGTA

Nesta Seção será mostrada a arquitetura distribuída do SGTA que é formada por componentes que fazem parte do Sistema de Apoio e pelas transações. O Sistema de Apoio executa na unidade móvel e é formado pelos seguintes componentes: o Monitor de Recursos, o Gerenciador de Cache, o Repositório de Objetos e a Fila de Efetivações. Máquinas da rede fixa também poderão manter o componente Repositório de Objetos (ver Figura 3.2).

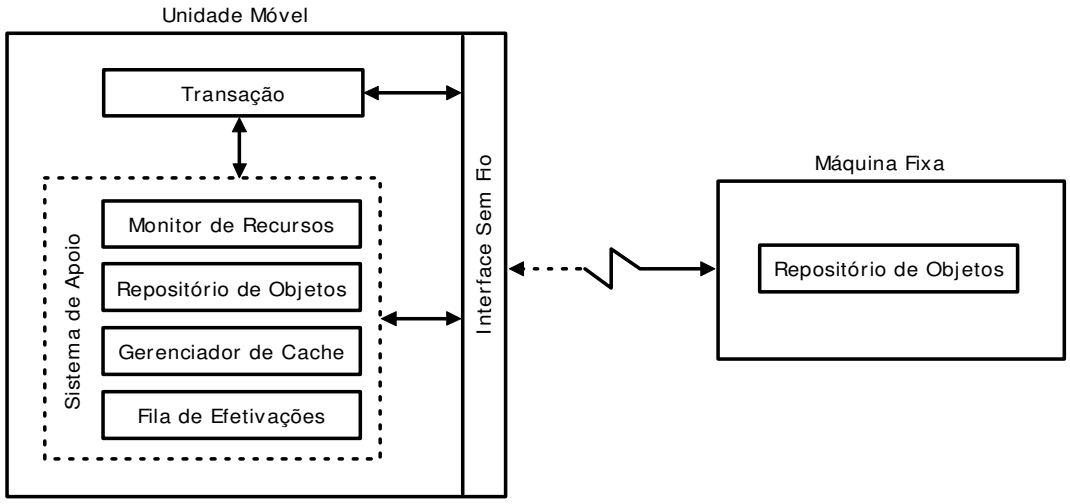



Figura 3.2: Arquitetura do SGTA.

Nas seções a seguir será descrito cada um dos componentes citados.

### 3.2.1 Monitor de Recursos

O Monitor de Recursos (MR) é responsável pelo monitoramento de recursos do ambiente de computação móvel que podem afetar ou influenciar a execução de transações que executam na unidade móvel. Estes recursos poderão dizer respeito a aspectos da comunicação entre a unidade móvel e a rede fixa como a largura de banda, a latência, as taxas de erros e o custo da comunicação, e a aspectos da unidade móvel, como nível de energia das baterias e espaço disponível em disco.

O MR é composto por módulos agregados onde cada um deles é independentemente responsável pelo monitoramento de um único recurso, ou seja, existirá um módulo responsável pelo monitoramento da largura de banda, outro para a latência e assim sucessivamente. Com esta estrutura, pode-se desenvolver e agregar novos módulos ao MR, que, por sua vez, poderão ser responsáveis pelo monitoramento de outros recursos (Figura 3.3).

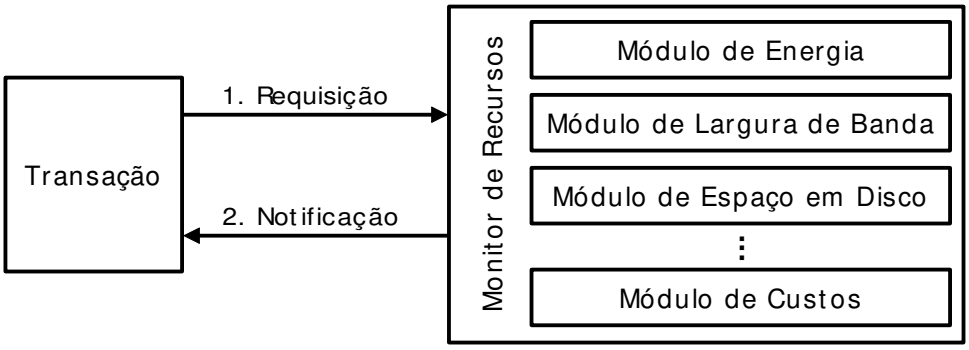



Figura 3.3: O Monitor de Recursos.

Cada transação é responsável por requisitar ao MR o monitoramento de um ou mais recursos de seu interesse. Esta requisição consiste na chamada de uma operação do MR com a passagem dos seguintes parâmetros: referência da transação interessada e o descritor do recurso em questão. Este descritor contém o identificador do recurso a ser monitorado e a janela de tolerância da transação interessada. A janela de tolerância é composta pelos limites superior e inferior aceitáveis pela transação para o recurso requisitado.

Quando o MR recebe uma requisição de monitoramento de recurso por parte de uma transação, ele registrará a requisição e passará a verificar se o nível do recurso requisitado está entre os limites especificados na requisição feita pela transação. Caso o nível do recurso requisitado extrapole um destes limites, o MR enviará uma notificação à transação correspondente (Figura 3.4). Uma vez notificada, a transação poderá tomar medidas de adaptação à mudança.

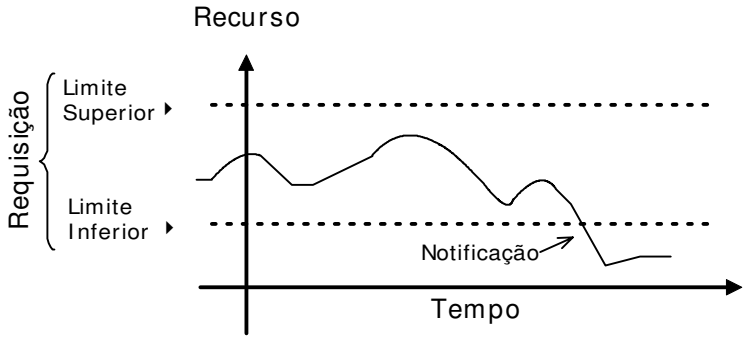



Figura 3.4: Monitoramento de um recurso.

O MR também aceita requisições de monitoramento que ignorem um dos limites (superior ou inferior) a ser monitorado. Podem existir transações que só estejam

interessadas em receber notificações em relação a um único limite especificado. Para que o MR ignore o monitoramento de um dos limites da requisição (`lowerBound` ou `upperBound`), deverá ser passado o valor -1 como o parâmetro do limite que deverá ser ignorado.

Vale ressaltar que, quando a transação recebe uma notificação do MR referente a uma requisição de monitoramento de recurso feita pela mesma, esta requisição é automaticamente excluída do MR. Para que a transação continue a receber notificações referentes a futuras mudanças no mesmo recurso, ela terá que refazer a requisição de monitoramento junto ao MR.

O MR, além de monitorar os recursos do ambiente de comunicação sem fio, assume a função de manter as transações interessadas informadas das variações ocorridas no ambiente. Este é um dos principais pontos do sistema adaptável que permite que cada transação possa tomar medidas reativas às variações do ambiente de comunicação sem fio.

### 3.2.2 Gerenciador de Cache

Transações que executam em uma unidade móvel normalmente necessitam acessar dados remotos localizados em máquinas na rede fixa. Porém, quando a comunicação com a rede fixa é realizada através de uma conexão sem fio, as transações podem encontrar problemas como alta latência, altas taxas de erros, baixa largura de banda, desconexões inesperadas e possivelmente um alto custo da comunicação (por exemplo, via telefone celular). Como um meio de contornar este problema, o SGTA provê técnicas de caching de forma a evitar o tanto quanto possível a comunicação com a rede fixa. A comunicação pode ser evitada copiando os dados necessários para a execução das transações da rede fixa para a cache da unidade móvel. Assim, as transações poderiam acessar as cópias em cache sem a necessidade de acessar a rede fixa durante suas execuções. Somente no final das transações é que as atualizações feitas sobre as cópias locais seriam propagadas para as respectivas cópias remotas.

Diante este contexto, o Gerenciador de Cache (GC) age como o componente da arquitetura do SGTA que provê serviços que possibilitam a cópia de objetos remotos da rede fixa para a cache da unidade móvel, a verificação da consistência entre as cópias dos objetos mantidos na unidade móvel e seus respectivos correspondentes remotos da rede fixa, a verificação da existência de um determinado objeto na cache da unidade móvel, e a atualização dos estados das cópias remotas da rede fixa a partir dos estados em cache atualizados por transações da unidade móvel.

### 3.2.3 Repositório de Objetos

Uma das propriedades ACID das transações é a durabilidade. Esta propriedade garante que depois que uma transação é efetivada com sucesso seus resultados não serão mais perdidos. Para garantir esta propriedade, o SGTA armazena os resultados de uma transação em

memória persistente. O Repositório de Objetos é o componente da arquitetura que provê operações para recuperar estados de objetos da memória persistente para a memória volátil e salvar objetos da memória volátil para a memória persistente.

### 3.2.4 Fila de Efetivações

Para uma transação que executa suas operações sobre cópias dos objetos da rede fixa armazenadas na cache da unidade móvel, geralmente é necessário que, ao final da execução da transação, as cópias locais sejam validadas em relação às suas correspondentes da rede fixa. Se estes objetos forem validados com sucesso, os resultados obtidos localmente sobre os objetos da cache seriam propagados para seus correspondentes da rede fixa. Este processo visa a garantia das propriedades de consistência e durabilidade de transações que operam sobre cópias em cache de objetos da rede fixa. A esse processo chamamos de processo de efetivação[7].

Para que o processo de efetivação de uma transação que opera sobre cópias de objetos da rede fixa possa ser efetuado, a unidade móvel deverá estar conectada com a rede fixa. Caso contrário, a transação em questão delegará seu processo de efetivação à Fila de Efetivação (FE). A FE é um componente da arquitetura que se responsabiliza pelo processo de efetivação de transações desconectadas da rede fixa. A FE se responsabilizará em detectar o restabelecimento da conexão da unidade móvel com a rede fixa e de disparar os processos de efetivação que estão em sua fila. Se o processo de efetivação puder ser aplicado com sucesso, a FE envia uma notificação de sucesso à transação correspondente, senão, a própria FE abortará a transação correspondente enviando uma notificação à mesma. Quando a FE aborta uma transação, todos os resultados da transação sobre os objetos acessados durante a execução da mesma são desfeitos.

As fases do processo de efetivação (validação e propagação dos resultados) serão detalhadas a seguir.

### 3.2.5 Transações

Uma transação é o componente alvo da arquitetura proposta. Os demais componentes foram concebidos como uma infra-estrutura necessária para prover mecanismos de adaptação às transações. Em suma, esta infra-estrutura provê às transações na unidade móvel: ciência das variações ocorridas nos recursos no ambiente móvel e mecanismos de adaptação.

---

[7] Em inglês: commit.

## 3.3 Mecanismos de Adaptação

A seção 3.2 apresentou a infra-estrutura que é dada a uma transação para que ela possa se adaptar às adversidades do ambiente de computação móvel. A seguir, serão apresentados os mecanismos que uma transação dispõe para concretizar esta adaptação.

Na seção 3.3.1 será apresentado o conceito de *nível de isolamento* e como uma transação poderá utilizar estes níveis. A Seção 3.3.2 apresentará um outro mecanismo de adaptação que poderá ser utilizado pelas transações chamado *modo de operação*. E para finalizar, a Seção 3.3.3 apresentará idéias de outros mecanismos de adaptação que poderão ser usados pelas transações.

### 3.3.1 Nível de Isolamento

O *Isolamento* é uma propriedade ACID que visa a garantia de que uma transação execute de forma isolada, ou seja, que não sofra interferências de outras transações concorrentes. Os efeitos destas interferências podem ser altamente indesejáveis para certos tipos de aplicações.

Para exemplificar a importância da propriedade de isolamento, consideremos o exemplo clássico de uma operação bancária concorrente onde as transações $T_1$ e $T_2$ tentam sacar os valores $v_1$ e $v_2$ respectivamente de uma mesma conta corrente. Vamos considerar que o saldo inicial da conta é $S$ onde $S \geq (v_1 + v_2)$. Se a propriedade de isolamento não for empregada, as transações poderão executar da seguinte maneira: **(i)** $T_1$ e $T_2$ lêem o saldo inicial $S$; **(ii)** $T_1$ subtrai de $S$ o valor $v_1$ e $T_2$ subtrai de $S$ o valor $v_2$; **(iii)** $T_1$ finaliza sua execução escrevendo o novo valor do saldo da conta como sendo $S$-$v_1$ e $T_2$ também finaliza sua execução escrevendo o novo valor do saldo como sendo $S$-$v_2$. Assim, $T_2$ sobrescreve o saldo escrito por $T_1$. Neste exemplo, as execuções de T1 e T2 resultaram em uma inconsistência inaceitável para uma aplicação bancária onde o saldo final deveria ser $S$-($v_1$+$v_2$) ao invés de $S$-$v_2$.

A forma mais simples de evitar que uma transação interfira no resultado de outras transações é através da execução serial das mesmas, ou seja, uma transação só inicia a sua execução quando nenhuma outra transação estiver executando. Porém, a execução serial não utiliza a vantagem da multiprogramação, o que pode acarretar grande impacto no desempenho. Como um meio de aliviar estes impactos e mesmo assim garantir a não interferência entre transações, a literatura propõe a técnica de *serialização* [9]. Se duas ou mais transações estão executando concorrentemente e de forma serializável, o resultado final obtido será o mesmo obtido pela execução das mesmas transações de forma serial.

Se as transações do exemplo acima executassem de forma serializável, elas não gerariam inconsistências, pois o resultado seria equivalente à seguinte execução serial: T1 seguido de T2.

Apesar de a serialização evitar interferências entre transações concorrentes com um desempenho melhor do que a execução serial, ela ainda apresenta problemas de desempenho [1, 2]. Estes problemas, basicamente advêm do seguinte fato: os mecanismos

utilizados para a garantia da serialização normalmente impõem restrições que acarretam um aumento no tempo médio necessário para uma transação obter acesso aos dados. Estes problemas podem ficar ainda mais acentuados quando uma transação acessa dados remotos via conexão sem fio caracterizada por baixa largura de banda, alta latência e altas taxas de erros que, por si só, podem acarretar perdas de desempenho.

Para aliviar os impactos no desempenho causados pela serialização, [18] introduziu o conceito de níveis de isolamento (inicialmente chamados de níveis de consistência) e que se tornou base para o padrão ANSI SQL-92 [4]. Este padrão vem sendo utilizado em bancos de dados comerciais, a exemplo do Oracle [41], como um mecanismo de melhoria de desempenho. Os níveis de isolamento foram numerados de 0 a 3, onde o nível 3 é a própria serialização. Quanto menor o nível de isolamento, menor é a garantia de consistência e maior será o desempenho alcançado. Assim, pode-se dizer que a mudança de nível de isolamento é uma troca entre garantia de consistência e desempenho.

Os níveis de isolamento são úteis devido à existência de tipos de aplicações que não requerem um nível tão restritivo quanto a serialização [1]. Podemos citar os casos abaixo:

**Programador da aplicação está ciente de que certos tipos de conflitos não irão ocorrer:** apesar de a aplicação executar com um nível de isolamento menos restritivo que a serialização, ela continuará a ser serializável.

**Programas que lêem informações aproximadas ou não serializáveis:** nesta classe de aplicações, a leitura de um estado aproximado (e inconsistente) de uma base de dados já é suficiente.

**Programas que detectam violação de invariantes:** existem programas que podem ser escritos para detectar invariantes violados e evitar inconsistências sem a necessidade do nível de serialização.

No SGTA proposto neste trabalho, foram utilizados os quatro níveis de isolamento baseados no padrão ANSI SQL-92 para aliviar as perdas de desempenho sofridas por transações que acessam dados remotos via comunicação sem fio caracterizada por baixa largura de banda, alta taxas de erros e alta latência. Assim, transações que suportam níveis de isolamento mais baixos que a serialização poderão melhorar seus desempenhos quando estes são afetados pela escassez de recursos do ambiente móvel.

Os níveis de isolamento podem ser informalmente definidos em função dos tipos de fenômenos geradores de inconsistência que eles evitam. Tais fenômenos são descritos como:

**Escrita Inconsistente[8].** Uma transação $T_1$ modifica um item de dado $x$ e logo depois $T_2$ também modifica $x$ antes que $T_1$ efetue a operação de *commit* ou de *abort*. Se $T_1$ ou $T_2$ efetuar uma operação de *abort*, não ficará claro qual o valor que $x$ deverá

[8] Em inglês: dirty write.

manter. Isto irá caracterizar uma escrita inconsistente onde uma transação poderá interferir na escrita da outra podendo gerar inconsistências na base de dados.

**Leitura Inconsistente[9].** Uma transação $T_1$ modifica um item de dado *x*. Uma transação $T_2$ lê *x* antes que $T_1$ efetue a operação de *commit* ou de *abort*. Se $T_1$ efetuar uma operação de *abort*, $T_2$ terá lido um item de dado que nunca existiu efetivamente caracterizando uma *leitura inconsistente*.

**Leitura Não Repetível[10].** Uma transação $T_1$ lê um item de dado *x* e então $T_2$ modifica ou exclui *x* e efetua a operação de *commit*. Se $T_1$ tentar reler o valor de *x*, ela receberá um valor modificado ou descobrirá que o item de dado foi excluído caracterizando uma *leitura não repetível*.

**Leitura Fantasma[11].** Uma transação $T_1$ lê um conjunto de itens de dados que satisfazem a uma condição de busca. Logo a seguir, uma transação $T_2$ cria novos itens de dados que satisfazem a condição de busca de $T_1$ e efetua a operação de *commit*. Se T1 repetir sua leitura com a mesma condição de busca inicial, ela receberá um conjunto de itens de dados diferente do obtido na primeira leitura caracterizando uma *leitura fantasma*.

O nível de isolamento mais baixo (Nível 0) não evita nenhum dos fenômenos acima citados enquanto que o nível mais alto (Nível 3) evita todos os fenômenos acima. A tabela X mostra os fenômenos de inconsistência que cada nível de isolamento previne.

| Nível de Isolamento | Fenômenos prevenidos |
|---|---|
| Nível 0 | Nenhum |
| Nível 1 = UNCOMMITTED_READ | Escrita Inconsistente |
| Nível 2 = COMMITTED_READ | Escrita Inconsistente, Leitura Inconsistente. |
| Nível 3 = SERIALIZABILITY | Escrita Inconsistente, Leitura Inconsistente, Leitura Não Repetível, Leitura Fantasma. |

Tabela 3.1: Fenômenos prevenidos pelos níveis de isolamento.

Como o método de controle de concorrência usado no SGTA proposto é o de bloqueio em duas fases (two-phase locking) [9], adotamos a definição de níveis de isolamento em função de trancas de curta ou longa duração que devem ser adquiridas para garanti-los (ver Tabela 3.2). Uma tranca de curta duração (ou tranca curta) sobre um objeto deve ser adquirida no início da execução da operação sobre o objeto e liberada logo após o término

[9] Em inglês: dirty read.

[10] Em inglês: non-repeatable read.

[11] Em inglês: phantom read.

da mesma. Já a tranca de longa duração (ou tranca longa), só deverá ser liberada no final da execução da transação.

| Nível de Isolamento | Tranca para Leitura | Tranca para Escrita |
|---|---|---|
| Nível 0 | Nenhum | Trancas curtas para escrita |
| Nível 1 = UNCOMMITTED_READ | Nenhum | Trancas longas para escrita |
| Nível 2 = COMMITTED_READ | Trancas curtas para leitura | Trancas longas para escrita |
| Nível 3 = SERIALIZABILITY | Trancas longas para leitura | Trancas longas para escrita |

Tabela 3.2: Níveis de isolamento ANSI baseados em trancas.

A seguir será detalhado cada um dos níveis de isolamento que pode ser adotado por uma transação assim como o grau de inconsistência que eles podem permitir:

**Nível 0.** Menor nível de isolamento que pode ser adotado por uma transação e não previne nenhum dos fenômenos de inconsistência citados. Este nível de isolamento é o que possui uma menor retenção de trancas e, portanto é o que gera um menor sobrecarga no acesso à base de dados. O nível 0 de isolamento pode ser, por exemplo, inteiramente satisfatório para transações somente de leitura que desejam obter informações estatísticas de uma base de dados quando não é necessária a obtenção de resultados exatos.

**Nível 1.** Previne transações do fenômeno da e*scrita inconsistente*. Este nível de isolamento, assim com os níveis maiores que ele, não gera inconsistências na base de dados. Porém, as transações que optam por este nível estarão sujeitas a fenômenos de inconsistência como *leitura inconsistente*, *leitura não repetível* e *leitura fantasma*. É claro que a execução de uma transação sob o nível 1 estará sujeita ao risco de ler dados que violam regras de integridade, porém poderá ser inteiramente satisfatório, por exemplo, para transações que não necessitem obter valores exatos ou quando o programador da transação está ciente de que certos conflitos nunca irão ocorrer.

**Nível 2.** Previne transações dos fenômenos *escrita inconsistente* e *leitura inconsistente*. Entretanto não evita fenômenos como *leitura não repetível* ou *leitura fantasma*.

**Nível 3.** Garante a execução de transações de forma serializável evitando todos os fenômenos de inconsistência. Este é o nível de isolamento mais alto que poderá ser atribuído a uma transação, porém ele causa a maior sobrecarga e a maior contenção de trancas em relação aos outros níveis. Isto ocorre porque são atribuídas trancas longas tanto para operações de leitura quanto para operações de escrita feitas pela transação.

### 3.3.2 Modo de Operação

O modo de operação é um parâmetro de adaptação que foi definido nesta pesquisa com o objetivo de viabilizar a execução de transações que acessam objetos remotos via comunicação sem fio sujeita a problemas como largura de banda e latência variáveis e sujeita a desconexões inesperadas. Com os modos de operação uma transação é capaz de manter uma política de adaptação de forma a continuar a sua execução em uma unidade móvel sujeita aos problemas da comunicação sem fio.

Os modos de operação que foram definidos são três: *remoto*, *local* e *local-remoto*. Basicamente eles definem se os objetos remotos participantes de uma transação na unidade móvel serão acessados remotamente ou localmente a partir de uma cópia mantida em cache. A seguir discutiremos brevemente a função de cada um deles:

**Remoto.** É o modo de operação mais comum quando se trata de uma transação que acessa objetos remotos: as operações da transação que executa sob o modo remoto são direcionadas para os seus respectivos objetos remotos localizados na rede fixa. Com este modo de operação, uma transação acessa objetos remotos consistentes já que cada objeto estará sujeito a um controle de concorrência. Porém, esta noção de consistência dependerá do nível de isolamento escolhido pela transação.

**Local.** As operações da transação que executa sob o modo de operação local são direcionadas para as cópias locais dos respectivos objetos mantidos na cache da unidade móvel. Com isso, no momento da execução da transação a comunicação com a rede fixa poderá ser evitada já que cada objeto remoto acessado pela transação possui uma cópia local em cache. Este modo de operação possibilita, por exemplo, que transações possam executar quando desconectadas da rede fixa ou quando a qualidade da comunicação com a rede fixa se tornar inviável devido a problemas relacionados com o custo da comunicação, com a largura de banda, com a latência ou taxas de erros, por exemplo. Com este modo de operação, uma transação acessa cópias locais que podem estar inconsistentes em relação às suas correspondentes remotas. Para lidar com este problema, no final da execução da transação que operou sob o modo de operação local, é disparado um processo de validação que verificará se os objetos locais acessados pela transação estão consistentes em relação aos seus correspondestes remotos. Se todos os objetos acessados estiverem consistentes, a transação é considerada válida e as atualizações feitas sobre as cópias locais são propagadas para as correspondentes remotas.

**Local-remoto.** Este modo de operação é semelhante ao modo de operação local, ou seja, as operações da transação que executa sob o modo de operação local-remoto são direcionadas para as cópias dos respectivos objetos mantidos em cache. Porém, no modo de operação local-remoto são atribuídas trancas aos respectivos objetos remotos da rede fixa de forma a manter a consistência destes com suas respectivas cópias locais da cache da unidade móvel. Assim, não será necessário o processo de

validação da transação como é feito no modo local. Apesar de garantir a consistência das cópias em cache acessadas pela transação, este modo de operação reduz a concorrência no acesso às cópias remotas na rede fixa.

### 3.3.3 Outros Mecanismos

Além dos níveis de isolamento e dos modos de operação que podem ser usados por transações como mecanismos de adaptação, uma transação também poderá utilizar outros mecanismos de adaptação de acordo com as suas necessidades. Como a política de adaptação é um trecho de código implementado pelo usuário da transação (como se verá a seguir), esta implementação poderá utilizar outros mecanismos de adaptação concebidos ou idealizados pelo próprio usuário de maneira a atender a outras necessidades de uma transação em específico.

Para exemplificar como outros mecanismos de adaptação podem ser usados imaginemos um computador portátil de um gerente de uma empresa bancária equipado com duas interfaces de comunicação: uma via cabo e outra via telefone celular. Este computador portátil possui uma aplicação bancária que utiliza o SGTA para efetuar algumas transações de longa duração. Na comunicação feita por estas transações com a rede local da empresa são passadas informações altamente confidenciais que devem ser protegidas. Enquanto o gerente está na empresa, ele conecta o seu computador portátil à rede local através da interface de comunicação via cabo. Quando o gerente precisa fazer alguma viagem de negócios ele leva o seu computador portátil e continua a utilizar a aplicação bancária, que, por sua vez, comunica-se com a rede local da empresa através da interface de comunicação via celular. Para proteger as informações que trafegam entre o computador pessoal do gerente e a rede local da empresa, a aplicação bancária utiliza um algoritmo de criptografia de alto nível de segurança que, apesar de proteger estas informações, torna a aplicação do gerente bem mais lenta devido às suas exigências de processamento. Tendo em vista estes problemas, pode-se adotar uma política de adaptação que só utiliza níveis de segurança mais altos enquanto o computador pessoal está conectado à empresa via celular. Quando a conexão é via cabo (indicando que o computador se encontra na empresa), o nível de segurança seria reduzido de forma a reduzir a carga de processamento do computador pessoal e, consequentemente, reduzir o seu tempo de resposta.

Apesar de este exemplo ser meramente ilustrativo, ele serviu para mostrar que outros mecanismos de adaptação assim como “níveis de criptografia” podem ser concebidos e implementados para serem usados por uma transação.

## 3.4 Políticas de Adaptação

Como foi dito nas seções anteriores, uma transação do SGTA poderá utilizar os mecanismos de adaptação em uma política de adaptação independente que deverá tratar aspectos de seu interesse.

Antes mesmo que uma transação do SGTA possa ser utilizada, a sua política de adaptação tem que ser definida. Esta definição é feita através da implementação de uma operação da transação chamada `notify`. Esta implementação deverá utilizar os mecanismos de adaptação (como os modos de operação e os níveis de isolamento) para adequar a transação ao dinamismo do ambiente móvel.

A operação `notify` será chamada automaticamente pelo Monitor de Recursos quando algum recurso de interesse da transação sofrer uma variação significativa, ou seja, extrapole os limites especificados pela mesma transação. Para que uma transação possa ser notificada através da chamada da operação `notify`, esta transação deverá efetuar as devidas requisições de monitoramento ao Monitor de Recursos, especificando cada recurso a ser monitorado bem como os limites toleráveis pela mesma.

Quando o monitor de recursos envia uma notificação a uma transação através da operação `notify`, ele passa dois parâmetros que serão fundamentais na definição de uma política de adaptação consistente. Estes parâmetros são o identificador e o nível atual do recurso que sofreu variação. A partir destes parâmetros a política de adaptação saberá qual foi o recurso que variou e qual a sua disponibilidade atual e assim poderá tomar as medidas adequadas para tal mudança. A seguir é apresentado um exemplo simples de uma política de adaptação:

```
void notify(resourceId,  resourceLevel){
   if (resourceId = "bandwidth"){
      if (resourceLevel < 10000){
         setOperationMode(LOCAL)
      }
   }
}
```

A política de adaptação apresentada no pseudocódigo acima é a seguinte: "ao receber uma notificação referente a uma mudança na largura de banda, se o nível da mesma for menor que 10000bps então mude o modo de operação da transação para local".

Apesar de o exemplo acima ser bastante simples, uma política de adaptação poderá utilizar qualquer conjunto de recursos monitorado pelo Monitor de Recursos, poderá utilizar mudanças tanto no modo de operação, no nível de isolamento, bem como em outros mecanismos de adaptação. Uma política de adaptação poderá também fazer novas requisições ao Monitor de Recursos. Tudo isto dependerá da lógica da política de adaptação criada pelo programador da transação.

## 3.5 Diagramas de Classes

Nesta seção, serão apresentados diagramas de classes simplificados que fornecerão uma visão geral do modelo utilizado na implementação do SGTA. A notação utilizada nestes diagramas foi a da UML[57]. O Apêndice B descreve os elementos desta notação UML que foram utilizados nos diagramas desta seção. O Apêndice A descreve os principais atributos e as operações de cada classe.

O modelo de classes do SGTA foi dividido em duas partes, uma que representa a parte do SGTA que se localiza em unidades móveis (Figura 3.5) e outra que representa a parte do SGTA que fica em máquinas da rede fixa (Figura 3.6).

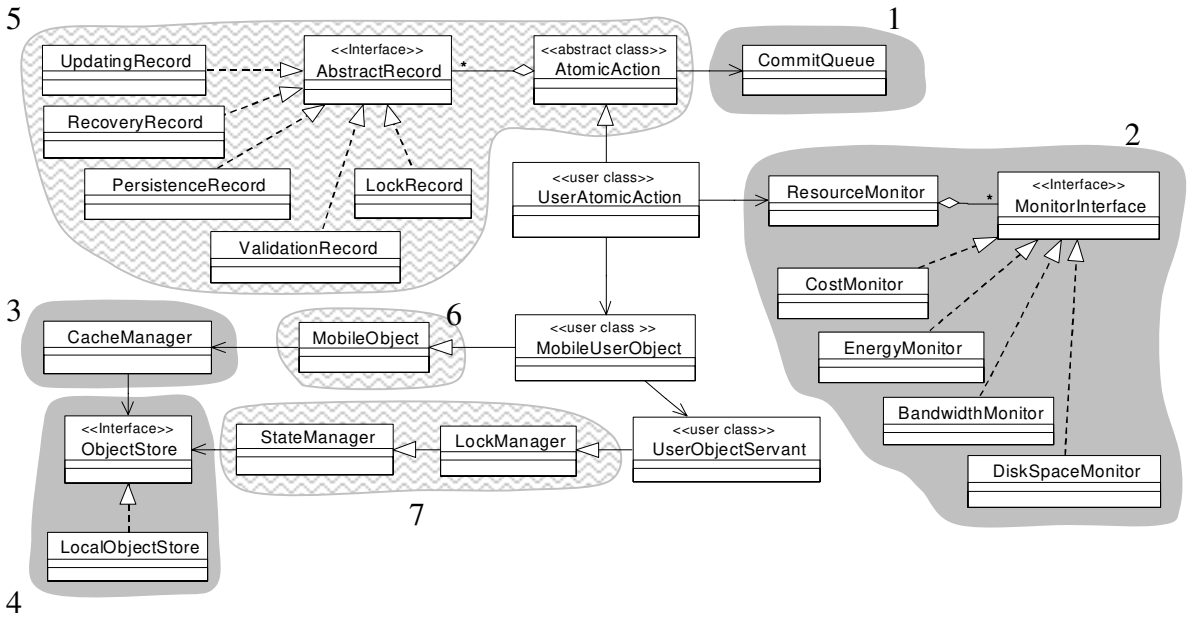



Figura 3.5: Diagrama de classes – componentes do SGTA na unidade móvel.

Para facilitar a compreensão dos diagramas de classes apresentados, estes foram divididos em blocos numerados. Na Seção 4 será descrita com mais detalhes cada parte constitui estes diagramas. Nesta seção será apresentada apenas uma breve descrição de cada bloco:

Considerações iniciais:

(i) a coloração indica que o componente faz parte do Sistema de Apoio;

(ii) a coloração indica outros componentes que fazem parte da infra-estrutura do SGTA;

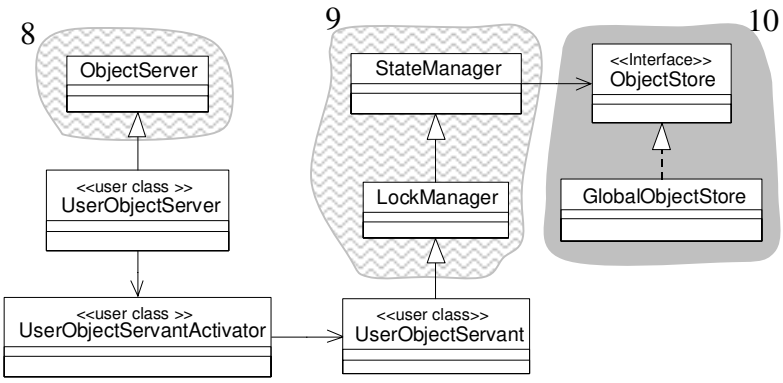



Figura 3.6: Diagrama de classes – componentes do SGTA na rede fixa.

(iii) classes marcadas com o estereótipo `<<user class>>` são classes que seriam desenvolvidas por programadores de transações que usam a infra-estrutura do SGTA;

Blocos que compõem os diagramas de classes:

**Bloco** 1 – implementa a Fila de Efetivações;

- **Relacionamentos:** (i) é usado pelo gerenciador de transações para a delegação do processo de efetivação.

**Bloco** 2 – implementa o Monitor de Recursos;

- **Relacionamentos:** (i) é usado pela transação do usuário para requisitar o monitoramento de recursos.

**Bloco** 3 – implementa o Gerenciador de Cache;

- **Relacionamentos:** (i) é usado pela classe `MobileObject` para transferência de objetos da rede fixa para a unidade móvel (e vice-versa) e para verificação da consistência de objetos; (ii) usa o Repositório de Objetos Local para transferir objetos da memória volátil para a não volátil e vice-versa.

**Bloco** 4 – implementa o Repositório de Objetos de unidades móveis;

- **Relacionamentos:** (i) é usado pelo Gerenciador de Cache para transferir estados de objetos da memória volátil para a não volátil e vice-versa. (ii) é usado pelo Gerenciador de Estados de objetos para transferir seus estados da memória volátil para a não volátil e vice-versa.

**Bloco** 5 – implementa o gerenciador de transações;

- **Relacionamentos:** (i) é herdado pela transação do usuário para que esta se torne um gerenciador de transações do SGTA; (ii) usa a Fila de Efetivações para a delegar o seu processo de efetivação.

**Bloco** 6 – implementa operações da classe MobileObject

- **Relacionamentos:** (i) é herdado por wrappers de objetos do usuário para que estes possuam operações que permitam a uma transação mudar o seu modo de operação de maneira transparente ao programador da transação.

**Bloco** 7 – implementa o Controle de Concorrência e Gerenciamento de Estados;

- **Relacionamentos:** (i) é herdado por objetos do usuário localizados na unidade móvel para que estes possuam operações de controle de concorrência e de gerenciamento de persistência.

**Bloco** 8 – implementa operações de um servidor de objetos do usuário;

- **Relacionamentos:** (i) é herdado pelo servidor de objetos do usuário para facilitar o seu desenvolvimento.

**Bloco** 9 – implementa o Controle de Concorrência e Gerenciamento de Estados;

- **Relacionamentos:** (i) é herdado por objetos do usuário localizados em máquinas da rede fixa para que estes possuam operações de controle de concorrência e de gerenciamento de persistência.

**Bloco** 10 – implementa o Repositório de Objetos de máquinas da rede fixa;

- **Relacionamentos:** (i) é usado pelo Gerenciador de Estados de objetos para transferir seus estados da memória volátil para a não volátil e vice-versa.

A seguir serão definidas as classes que não se encontram nos blocos descritos acima. Estas classes (com o estereótipo `<<user class>>`) são não fazem parte da infra-estrutura do SGTA. Ao invés disso, estas são classes que seriam definidas pelos programadores de transações que desejam utilizar a infra-estrutura do SGTA no processo de adaptação. Seguem as classes:

**`UserObjectServant`** – classe que implementa um objeto do usuário, ou seja, implementa um objeto que poderá ser usado por uma transação no escopo de suas operações (entre o *begin-transaction* e o *end-transaction*). Por exemplo, esta classe poderia se chamar `CounterServant` e implementar as operações de um contador (inc – incrementar contador; dec – decrementar contador). Este contador seria, por sua vez, usado por uma transação como parte suas operações.

**`MobileUserObject`** – classe que herda da classe `MobileObject` e que assume a função de invólucro (wrapper) de um objeto do usuário. No SGTA, todo objeto do usuário só deve ser usado por uma transação através do seu invólucro. O invólucro é responsável por receber mensagens (operações) de transações e redirecionar estas mensagens para o objeto do usuário correspondente (em sua instância remota ou

local). Por exemplo, o invólucro do objeto do usuário `CounterServant` seria o `MobileCounter`. Esta classe seria responsável por receber mensagens de transações e redirecioná-las para o objeto `CounterServant` instanciado localmente (na unidade móvel) ou remotamente (na rede fixa) dependendo do modo de operação da transação requisitante.

**`UserAtomicAction`** – classe que implementa a transação do usuário. Ela herda da classe `AtomicAction`, implementa a política de adaptação para a transação e implementa o conjunto de operações da transação que será executado. Por exemplo, poderia ser definida uma classe chamada `TransferAtomicAction` que herda `AtomicAction` e que é responsável por transferir um determinado valor entre duas contas correntes de um banco.

**`UserObjectServer`** – classe que implementa um servidor de objetos do usuário. Para que objetos da rede fixa possam ser usados remotamente por transações localizadas em unidades móveis é necessário que um servidor de objetos o disponibilize de alguma maneira. Por exemplo, para disponibilizar o acesso remoto a objetos do usuário CounterServant localizados na rede fixa, um servidor de objetos chamado CounterServer poderia ser desenvolvido.

**`UserObjectServantActivator`** – classe que implementa um ativador de servants sob demanda. Esta classe é responsável por instanciar objetos do usuário somente quando estes são requisitados por uma transação. Por exemplo, um ativador de servants para a classe `CounterServant` poderia ser chamada `CounterServantActivator` e seria responsável por instanciar contadores somente quando estes são requisitados por transações.

Nesta seção foi apresentada uma breve descrição do modelo do SGTA. A Seção 4 apresentará descrições mais detalhadas dos elementos que compõem o SGTA.

# Capítulo 4

# Implementação

Como uma maneira de levantar e consolidar requisitos do SGTA, foi utilizada a técnica de engenharia de software chamada prototipação rápida [55]. Esta técnica é baseada no desenvolvimento de um protótipo rápido dando ênfase aos seus aspectos funcionais.

Para implementação do protótipo utilizamos a linguagem de programação Java e a tecnologia CORBA (JavaIDL) ambos através da plataforma J2SE 1.4 da Sun. Esta plataforma foi escolhida devido ao seu conjunto de API's disponíveis que ajudam no desenvolvimento rápido do protótipo. Neste Capítulo, serão apresentados aspectos desta implementação.

## 4.1 O Ciclo de Vida dos Objetos do SGTA

Primeiramente, um objeto é dito persistente quando o seu estado não é perdido quando os processos que o acessam são finalizados. Um meio de implementar objetos persistentes é manter o seu estado em memória não volátil.

Para garantir a propriedade de durabilidade sobre os objetos persistentes acessados por uma transação, foi implementado um modelo de persistência de objetos composto de duas representações:

**Ativa.** É o próprio objeto composto por variáveis de instância que mantém o seu estado e por operações que podem acessar este mesmo estado. Esta representação ativa é uma instância da classe que define o objeto. O estado de um objeto em sua representação ativa está pronto para ser acessado ou manipulado por aplicações.

**Passiva.** É uma instância da classe `ObjectState` com a finalidade de encapsular todo o estado de um objeto. Para isso, a classe `ObjectState` contém um buffer que armazena o valor de cada variável de instância que compõe o estado do objeto em questão. A classe `ObjectState` também provê um conjunto de operações que permitem a conversão de um objeto ativo em uma instância de `ObjectState` e vice-versa. Basicamente, a representação passiva é útil para que o estado de um

objeto possa ser armazenado em um dispositivo de memória não volátil para fins de persistência.

A partir da noção de representação ativa e passiva de um objeto será definido o conceito de ciclo de vida de um objeto persistente [42] (Figura 4.1).

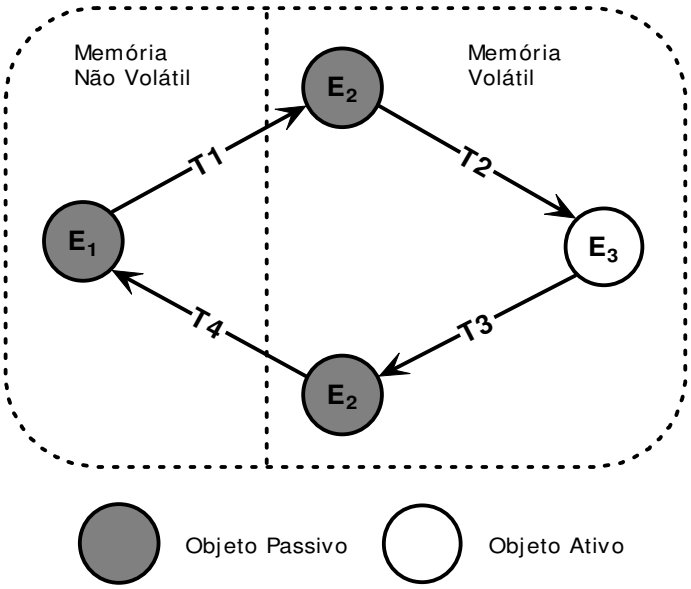



Figura 4.1: Ciclo de vida de um objeto persistente.

O ciclo de vida de um objeto persistente é composto pelos estados E1, E2 e E3 e pelas transições T1, T2, T3 e T4. Segue a definição de cada um dos estados:

**E1** - Objeto passivo em memória não volátil. Encontra-se armazenado no repositório de objetos como uma instância da classe `ObjectState`.

**E2** - O objeto passivo na memória volátil. É uma instância da classe `ObjectState` na memória volátil.

**E3** - O objeto ativo na memória volátil. Está pronto para ser utilizado por uma aplicação.

Considerando que o estado inicial do ciclo é o E1, as transições do ciclo de vida sempre acontecerão na ordem T1, T2, T3 e T4. Segue a definição de cada uma destas transições:

**T1** - O objeto é requisitado por uma aplicação. Assim, a instância da classe `ObjectState` armazenada no repositório de objetos é recuperada para a memória volátil.

**T2** - A instância de `ObjectState` é convertida na representação ativa do objeto.

**T3** - A aplicação que utiliza o objeto ativo é finalizada. O objeto ativo será convertido de volta para a forma passiva como uma instância de `ObjectState`.

**T4** - A instância de `ObjectState` é armazenado novamente no repositório de objetos.

## 4.2 Gerenciamento de Estados

### 4.2.1 Armazenando o Estado de Objetos

O estado de um objeto utilizado por transações no SGTA deverá ser armazenado com os seguintes propósitos: recuperação, persistência e distribuição. A seguir é apresentado cada um destes propósitos mostrando a necessidade de armazenamento do estado de um objeto:

**Recuperação.** Garante que um objeto poderá ter o seu estado anterior consistente recuperado caso ocorra falha em alguma aplicação que o manipule. A recuperação é utilizada no SGTA para garantir a propriedade da atomicidade das operações de uma transação. Quando uma transação é abortada, todas as atualizações sobre os objetos manipulados por ela devem ser desfeitas através da restauração do estado anterior consistente armazenado de cada objeto.

**Persistência.** Os resultados obtidos na manipulação de um objeto por uma aplicação não serão perdidos quando a aplicação que o manipula é finalizada. Para garantir a persistência de um objeto, o seu estado deverá ser armazenado em um dispositivo de memória não volátil. A persistência é utilizada no SGTA para garantir a propriedade da durabilidade de transações.

**Distribuição.** Permite que cópias de um mesmo objeto possam estar localizadas em diferentes pontos de uma rede. O SGTA utiliza a distribuição de objetos da seguinte maneira: um objeto possui uma única cópia de referência (ou matriz de referência) normalmente localizada em uma máquina da rede fixa e as demais cópias deste mesmo objeto podem estar distribuídas nas caches das unidades móveis. Para efetuar a cópia de um objeto matriz para uma unidade móvel, o SGTA transfere e armazena na cache da unidade móvel somente o estado do objeto. As regras de consistência deste ambiente são: o estado do objeto matriz é considerado consistente; os estados das cópias desta matriz localizadas em unidade móveis diferentes poderão divergir entre si; o estado de uma cópia de um objeto só será consistente se ele for consistente em relação ao estado de sua matriz.

Para qualquer um destes três propósitos citados, o mecanismo utilizado para armazenar o estado de um objeto é o mesmo: através das classes `ObjectState` e `Buffer`. A classe `Buffer` mantém um array que é capaz de armazenar seqüencialmente instâncias de tipos primitivos do Java. Este armazenamento é feito pelas operações que possuem em

seu nome o termo “pack”. Os tipos primitivos armazenados no array da classe `Buffer` também podem ser recuperados seqüencialmente pelas operações com o termo “unpack”. Segue o resumo da interface desta classe:

```
class Buffer
{
   /* CONSTRUTORES */
   Buffer(String bufferId);
   Buffer();

   /* Operações que armazenam (pack) tipos primitivos Java no Buffer */
   boolean packByte(byte data);
   boolean packInt(int data);
   boolean packBoolean(boolean data);
   boolean packString(String data);
   ... /* Existem outras operações */

   /* Operações que recuperam (unpack) do Buffer tipos primitivos Java */
   byte unpackByte();
   int unpackInt();
   boolean unpackBoolean();
   String unpackString();
   ... /* Existem outras operações */
}
```

A classe `ObjectState` herda toda a funcionalidade da classe `Buffer` adicionando basicamente dois atributos: identificador (`id`) e tipo (`type`). Uma instância da classe `ObjectState` possui a função de armazenar o estado de um determinado objeto identificado unicamente pelos atributos `id` e `type`. Este armazenamento é feito através do uso das operações “pack” herdadas da classe `Buffer` e que são capazes de armazenar cada item de dado que compõe o estado de um objeto. Segue o resumo da interface da classe `ObjectState`:

```
class ObjectState extends Buffer
{
   /* ATRIBUTOS PRIVADOS */
   String id;
   String type;

   /* CONSTRUTORES */
   ObjectState(String id, String type);
   ObjectState();

   /* OPERAÇÕES PÚBLICAS */
   String getId();
   String getType();
}
```

### 4.2.2 O Repositório de Objetos

Como foi mostrado na seção 3.2, o repositório de objetos é o componente da arquitetura do SGTA responsável por manter em memória não volátil o estado de objetos persistentes. O repositório de objetos foi concebido através da implementação da interface `ObjectStore`.

O repositório de objetos só armazena e recupera instâncias da classe `ObjectState`. Portanto, para que ele possa armazenar o estado de um objeto, dever ser criada uma instância da classe `ObjectState` que encapsule este estado. O repositório de objetos poderá armazenar duas versões do estado de um mesmo objeto: uma versão chamada *não efetivada*[12] e uma chamada *efetivada*[13]. O armazenamento destas duas versões é feito através de arquivos em disco distintos nomeados com o valor dos atributos `id` e `type` do `ObjectState`. A versão efetivada pode ser entendida como a versão de referência do estado do objeto armazenado. A versão não efetivada pode ser entendida como uma versão temporária do estado do objeto. A versão não efetivada normalmente é uma versão candidata a versão efetivada.

A seguir serão apresentadas as principais operações da interface do repositório de objetos:

```
interface ObjectStore
{
   /* OPERAÇÕES PÚBLICAS */
   boolean commit_state(String id, String type);
   boolean write_committed(ObjectState state);
   boolean write_uncommitted(ObjectState state);
   ObjectState read_committed(String id, String type);
   ObjectState read_uncommitted(String id, String type);
   boolean remove_committed(String id, String type);
   boolean remove_uncommitted(String id, String type);
   String type();
   ...
}
```

A operação `commit_state` transforma uma versão não efetivada de um determinado objeto identificado pelos atributos `id` e `type` em versão efetivada. Caso já exista uma versão efetivada para o objeto em questão, esta será sobrescrita.

As operações responsáveis por armazenar estados de objetos no repositório são o `write_committed` e o `write_uncommitted`. A primeira operação armazena uma instância da classe `ObjectState` como a versão efetivada do objeto. A segunda operação armazena uma instância de `ObjectState` como a versão não efetivada do objeto. Para cada uma

---

[12] Em inglês: uncommitted.

[13] Em inglês: committed.

destas operações, se já existir uma versão anterior correspondente armazenada para o objeto em questão, esta versão será sobrescrita.

Para recuperar estados de objetos do repositório são usadas as operações `read_committed` e `read_uncommitted`. Como o próprio nome diz, a primeira operação é responsável pela leitura do estado efetivado de um objeto identificado pelos atributos `id` e `type`. Por sua vez, a segunda operação é responsável pela leitura do estado não efetivado do objeto. Ambas as operações devolvem uma instância da classe `ObjectState`.

Outras operações do repositório são o `remove_committed` e o `remove_uncommitted`, que são utilizadas para remover a versão efetivada e a versão não efetivada de um objeto do repositório, respectivamente. Por fim, a operação `type` devolve o tipo do repositório. Este tipo pode significar, por exemplo, qual o meio de armazenamento que o repositório usa para manter os estados armazenados.

## 4.3 Replicação

A replicação de dados é uma técnica fundamental para garantir a disponibilidade de dados em redes sujeitas a falhas. Basicamente, a replicação de dados consiste em manter cópias dos dados em mais de um ponto da rede de maneira que, se um ponto da rede não estiver disponível, pode-se acessar os dados em um outro ponto que contenha a cópia dos dados.

### 4.3.1 Mecanismo de Replicação

Os mecanismos de replicação podem ser classificados em: replicação nos servidores ou replicação nos clientes [26]. Os mecanismos de replicação nos servidores disponibilizam cópias dos mesmos dados em mais de um servidor de dados da rede. Os mecanismos de replicação em clientes são também conhecidos como *caching* e possuem a característica de manter cópias dos dados dos servidores da rede na cache do cliente. Algumas vantagens da replicação nos servidores são uma maior segurança, completude e precisão dos dados. Porém, em caso de desconexão do cliente, os dados ficam totalmente indisponíveis ao mesmo. Já a replicação em clientes, apesar de ser inferior com relação a estes aspectos de segurança, completude e precisão, oferece um maior desempenho no acesso a dados e maior disponibilidade em casos de desconexão. Porém, este mecanismo requer uma revalidação periódica dos dados em cache. Também existem soluções que buscam tirar proveito de ambos os mecanismos ao mesmo tempo como, por exemplo, o projeto Coda [32].

Apesar de a replicação em servidores apresentar uma série de benefícios ao acesso a dados de uma rede, o SGTA, até então, utiliza um mecanismo que se restringe à replicação em clientes. Esta restrição foi feita com o intuito de concentrar esforços na resolução de problemas que se acentuam quando se trata da comunicação sem fio. Estes problemas podem ser visivelmente diluídos através da utilização da replicação em clientes.

### 4.3.2 Controle de Réplicas

Quando se utilizam mecanismos de replicação de dados faz-se necessária a existência de um controle de réplicas. A função deste algoritmo consiste basicamente em coordenar a leitura, a cópia e a atualização das réplicas. Os controles de réplicas são classificados em duas famílias: os pessimistas e os otimistas. Um controle de réplica pessimista só permite a uma aplicação acessar uma réplica quando o mesmo tem certeza prévia de que o acesso é consistente. Para garantir este acesso consistente, o controle pessimista normalmente dá a esta aplicação o acesso exclusivo. Embora estes sejam proibitivos e possam assim acarretar baixos desempenhos no acesso a dados concorrentes, seu uso é essencial para certos tipos de aplicações que exigem respostas corretas a todo instante, tais como as aplicações bancárias.

Em contrapartida, um controle de réplicas otimista permite a uma aplicação o acesso imediato a uma réplica sem a certeza prévia de que o acesso é consistente. Os riscos da utilização deste controle de réplicas são a leitura de dados desatualizados e conflitos na operação de escrita. A estratégia utilizada por este controle de réplicas é tentar detectar e resolver estes problemas depois que eles ocorrem. A vantagem da utilização deste controle de réplicas é que eles se adequam bem a redes lentas e com baixa confiabilidade.

Como foi apresentado na Seção 2.1.2, o ambiente de computação pode gerar três estados básicos de conexão entre a unidade móvel e a rede fixa: conexão forte, conexão fraca e desconexão. Levando-se em conta as características apresentadas de cada estado, observamos que, tanto algoritmos otimistas quanto pessimistas poderiam se tornar adequados a depender do estado de conexão e dos requisitos de cada aplicação. Portanto, decidimos prover às transações do SGTA a possibilidade de escolha entre o controle de réplicas otimista ou o pessimista. Ambos os controles serão mecanismos que coordenarão de maneiras diferentes a leitura e a atualização entre os objetos na rede fixa e suas réplicas na cache da unidade móvel.

O meio de uma transação optar pelo controle de réplicas otimista é utilizar o modo de operação local. Neste modo, uma transação acessa a cópia em cache sem se estar previamente ciente de que a cópia em cache está consistente em relação a sua correspondente da rede fixa. A verificação da consistência dos dados acessados pela transação e a propagação das atualizações para a rede fixa só serão feitas no final da execução da transação, na chamada fase de validação.

Uma transação estará optando pelo controle de réplicas pessimista quando ela utilizar o modo de operação local-remoto. Neste modo de operação, são atribuídas trancas aos objetos remotos da rede fixa de forma a garantir previamente a consistência destes com suas respectivas cópias locais da cache da unidade móvel. Como outras transações só terão acesso às respectivas réplicas da rede fixa para operações de leitura, a consistência da transação que optou pelo modo local-remoto estará garantida.

O modo de operação remoto não utiliza controle de réplicas, pois considera que a única réplica dos objetos acessados é aquela que está na rede fixa.

### 4.3.3 Gerenciador de Cache

O gerenciador de cache é o componente da arquitetura responsável pelos processos de replicação, atualização de réplicas, verificação de consistência de réplicas dentre outras funções. A classe que implementa o gerenciador de cache é o `CacheManager`. Segue o resumo da interface desta classe:

```
class CacheManager
{
   /*CONSTRUTORES*/
   CacheManager();
   /*OPERAÇÕES PÚBLICAS*/
   boolean loadState(String id, String type, String server, String port);
   boolean updateRemoteState(String id, String type, String server,
                             String port);
   boolean isPresent(String id, String type);
   boolean isActive(String id, String type);
   boolean activate(String id, String type);
   boolean isConsistent(String id, String type, String server,
                        String port);
   ...
}
```

As operações do gerenciador de cache que fazem transferência de estados de objetos entre a rede fixa e a unidade móvel são o `loadState` e o `updateRemoteState`. A operação `loadState` é responsável por copiar para a unidade móvel o estado de um objeto localizado na máquina da rede fixa e carregar este estado em um objeto ativo. A operação `updatRemoteState` tem a função de atualizar o estado do objeto localizado na máquina da rede fixa a partir do estado do seu correspondente na unidade móvel. Nestas operações, os parâmetros `id` e `type` identificam o objeto que será manipulado, o parâmetro `server` identifica a máquina da rede fixa onde o objeto matriz se encontra e o parâmetro `port` indica a porta da máquina da rede fixa onde se encontra o serviço de nomes do qual pode ser obtida uma referência para o objeto matriz.

Além das operações de transferência de estados, o gerenciador de cache possui outras operações auxiliares, como o `isActive`, que indica se um determinado objeto da unidade móvel, identificado pelos parâmetros `id` e `type`, encontra-se no estado ativo ou não. A operação `isPresent` verifica se o estado do objeto identificado pelos parâmetros `id` e `type` se encontra armazenado no repositório de objetos da unidade móvel. A operação `activate` faz com que um objeto que se encontra no estado passivo armazenado no repositório de objetos se torne ativo. E por fim, a operação `isConsistent` verifica se um determinado objeto na cache da unidade móvel é consistente em relação a sua matriz na máquina da rede fixa. Nesta operação, o objeto em questão é identificado pelos parâmetros `id` e `type`, a máquina da rede fixa é identificada pelo parâmetro `server`, o parâmetro `port` indica a

porta da máquina fixa onde se encontra o serviço de nomes do qual pode ser obtida uma referência para o objeto matriz na máquina fixa.

## 4.4 Gerenciamento de Transações

### 4.4.1 Visão Geral

Transações no SGTA são implementadas pela classe `AtomicAction`. Esta classe provê as operações básicas que são úteis da definição de transações. Segue um resumo da classe `AtomicAction`:

```
abstract class AtomicAction
{
   /* OPERAÇÕES DELIMITADORAS */
   int Begin();
   int Begin(int isolationLevel, int operationMode);
   int End();
   int Abort();
   /* OPERAÇÕES DE ADAPTAÇÃO */
   int setOperationMode(int operationMode);
   int setIsolationLevel(int isolationLevel);
   /* OPERAÇÃO ABSTRATA PARA A POLÍTICA DE ADAPTAÇÃO */
   abstract void notify(String resourceId, int resourceLevel);
}
```

Todo conjunto de operações que fazem parte de uma transação tem que estar entre as operações delimitadoras `Begin` e `End/Abort`. Estas operações marcam o início e o fim de uma transação, respectivamente. A operação `Begin()` (sem parâmetros) considera que a transação iniciará sob o modo de operação remoto e sob o nível de isolamento 3. A operação `Begin` também poderá conter os parâmetros `isolationLevel` e `operationMode` que indicarão, respectivamente, o nível de isolamento e o modo de operação sob os quais a transação irá executar. A operação `Abort`, quando chamada, desfaz todo o efeito da transação sobre os objetos que ela manipulou. A operação `End` delimita o fim de uma transação dando início, assim, à fase de efetivação da mesma.

Além destas operações, a classe `AtomicAction` provê operações de adaptação como o `setIsolationLevel` e `setOperationMode`. Estas operações permitem, respectivamente, que a transação mude o seu nível de isolamento e o seu modo de operação durante a execução de suas operações. Estas operações são chamadas operações de adaptação porque podem ser usadas com o propósito de adaptação às mudanças dos recursos do ambiente computacional onde a transação executa.

Para que cada transação possa ter a sua política de adaptação, a classe `AtomicAction` provê uma operação abstrata chamada `notify`. É através da implementação desta operação

que deverá ser concebida a política de adaptação que possibilitará à transação reagir e se adaptar às variações nos recursos do ambiente móvel. Esta operação contém dois parâmetros, o `resourceId` e o `resourceLevel`. Estes parâmetros significam, respectivamente, o identificador do recurso que sofreu variação significativa e o nível atual do mesmo recurso. Através destes parâmetros é possível saber qual o recurso que mudou e qual foi a mudança.

O mecanismo provido pela linguagem Java para implementar uma operação abstrata de uma classe é através da herança. A seguir é apresentado um exemplo simples de uma classe chamada `MyAtomicAction` que implementa a operação `notify`. Esta implementação definirá a política de adaptação: "se a largura de banda sofreu variação e se encontra em menos que 10000bps então mude o modo de operação da transação para o modo local".

```
class MyAtomicAction extends AtomicAction
{
   void notify(String resourceId, int resourceLevel){
      /* POLÍTICA DE ADAPTAÇÃO */
      if (resourceId.equals("bandwidth")){
         if (resourceLevel < 10000 ){
             this.setOperationMode(LOCAL);
         }
      }
   }
}
```

Como foi apresentado na seção 3.2.1, para que uma transação possa executar sua política de adaptação definida na operação `notify`, ela deve requisitar o monitoramento de recursos ao monitor de recursos do SGTA. Nesta requisição, a transação informa: o recurso a ser monitorado, e a janela de tolerância em relação às variações do recurso. Esta janela é composta por um limite inferior e um limite superior. Uma vez efetuada a requisição de monitoramento, o monitor de recursos registrará a requisição e ficará responsável por notificar a transação requisitante caso o recurso em questão extrapole um dos limites especificados. Esta notificação será justamente uma chamada à operação `notify` da transação requisitante. Isto fará com que a política de adaptação da transação seja disparada.

Acompanhando o exemplo a seguir pode-se observar uma maneira simples de como a transação adaptativa `MyAtomicAction` , que foi definida anteriormente, poderá ser usada.

```
/* DEFINIÇÃO DE UMA TRANSAÇÃO */
MyAtomicAction trn;
/* OBJETOS PARTICIPANTES DA TRANSAÇÃO */
ObjectA objA;
ObjectB objB;

/* INICIANDO A TRANSAÇÃO COM O NÍVEL 3 DE ISOLAMENTO
   E COMO O MODO DE OPERAÇÃO REMOTO */
trn.Begin(SERIALIZABILITY, REMOTE);
   /* A TRANSAÇÃO REQUISITA O MONITORAMENTO DE LARGURA DE BANDA
      ENTRE OS LIMITES 10000bps E 90000bps*/
   resourceMonitor.request(trn, ("bandwith", 10000, 90000));
   /* OPERAÇÕES SOBRE OS OBJETOS PERTICIPENTES DA TRANSAÇÃO *.
   objA.operation1();
   objB.operation2();
   ...
   objA.operation3();
trn.End(); /* FIM DA TRANSAÇÃO */
```

Este exemplo mostra que a transação requisita o monitoramento de largura de banda ao monitor de recursos. Esta requisição especifica limites de tolerância entre 10000bps e 90000bps de largura de banda. Se a largura de banda extrapolar estes limites especificados, o monitor de recursos chamará a operação `notify` da transação definida pela classe `MyAtomicAction`. Em suma, estes exemplos mostram que no SGTA uma transação tem a capacidade de: (i) estar ciente das variações ocorridas nos recursos do ambiente; (ii) possuir uma política própria de adaptação que atenda os seus interesses individuais; (iii) ter mecanismos de adaptação adequados para sobrepor os obstáculos do ambiente.

Os objetos participantes de uma transação são instanciados a partir de classes definidas pelo usuário e que herdam da classe `LockManager`. Esta classe é responsável pelo controle de concorrência do objeto. A classe `LockManager` herda da classe `StateManager`, que, por sua vez, é responsável por gerenciar o estado do objeto no que diz respeito a aspectos de recuperação e persistência. Desta forma, o objeto do usuário que herda da classe `LockManager` possuirá primitivas necessárias para o controle de concorrência, recuperação e persistência do objeto. Estas primitivas são utilizadas pela classe `AtomicAction` para gerenciar os objetos participantes de uma transação.

### 4.4.2 Recuperação e Persistência

Os objetos participantes de transações do SGTA possuem as propriedades de persistência e de recuperação. A persistência garante que o estado dos objetos participante não será perdidos depois que a transação é efetivada. É através do uso da persistência de objetos que

uma instância de `AtomicAction` garante a propriedade da *durabilidade* de transações. A recuperação garante que os objetos participantes terão seu estado anterior recuperado caso a transação seja abortada. A recuperação é usada como mecanismo para garantia da propriedade de *atomicidade* de transações. Como foi mencionado na seção anterior, a classe `StateManager` é responsável por gerenciar a recuperação e a persistência de um objeto. Segue o resumo da classe `StateManager`:

```
abstract class StateManager
{
   /* CONSTRUTORES */
   StateManager(String objectId);
   ...
   /* OPERAÇÕES PÚBLICAS */
   boolean activate();
   boolean deactivate(boolean committed);
   String getId();
   ...
   /* OPERAÇÕES ABSTRATAS */
   abstract boolean save_state(ObjectState objState);
   abstract boolean restore_state(ObjectState objState);
   abstract String type();
}
```

Segundo o ciclo de vida de objetos persistentes no SGTA, um objeto poderá estar no estado ativo ou passivo. Considerando-se que um objeto encontra-se inicialmente no estado passivo armazenado no repositório de objetos, a operação `activate` do `StateManager` tem a função de recuperar este estado passivo e transformá-lo no estado ativo. Um objeto no estado ativo estará pronto para ser manipulado por uma ou mais transações. Para que o estado ativo de um objeto volte ao estado passivo e seja novamente armazenado no repositório de objetos, a operação `deactivate` deverá ser chamada. Esta operação possui o parâmetro `committed` que indica se o estado passivo que será armazenado diz respeito à versão efetivada ou à versão não efetivada do objeto.

A classe `StateManager` possui três operações abstratas que deverão ser implementadas pelas classes que definem objetos do usuário. As operações são: `save_state`, `restore_state` e `type`. A operação `save_state` deverá armazenar em uma instância da classe `ObjectState` o conteúdo de todos os atributos que compõem o estado do objeto definido pela classe do usuário. Por exemplo, a seguir será implementada a operação `save_state` para uma classe chamada `Pessoa` que encapsula os atributos `nome` e `idade`. A classe `Pessoa` será então composta por estes dois atributos. Então a operação `save_state` para esta classe simplesmente armazenará o conteúdo destes atributos no `ObjectState`:

```
class Pessoa ...
{
   String nome;
   int idade;
   ...
   boolean save_state(ObjectState state){
      state.packString(nome);
      state.packInt(idade);
      return true;
   }
   ...
}
```

A operação `restore_state` deverá ser implementada de forma inversa à implementação do `save_state`, ou seja, deve-se recuperar cada valor que está armazenado no `ObjectState` e armazená-los nos respectivos atributos que compõem o estado dos objetos definidos pela classe em questão. Deve-se salientar que os valores deverão ser recuperados por operações do tipo "unpack" na mesma ordem que eles foram armazenados por operações do tipo "pack". Voltando ao exemplo da classe `Pessoa`, a implementação da operação `restore_state` seria:

```
boolean restore_state(ObjectState state){
   this.nome = state.unpackString();
   this.idade = state.unpackInt();
   return true;
}
```

As operações `save_state` e `restore_state` devolvem um tipo boolean. Este boolean indica se as respectivas operações foram efetuadas com sucesso. Os exemplos das operações `save_state` e `restore_state` acima retornam sempre `true` de forma a simplificar o exemplo.

A última operação abstrata que deverá ser implementada é a operação `type`. Ela deverá ser implementada para devolver o tipo da classe que está implementando a operação. Segue o exemplo para a classe `Pessoa`:

```
String type(){
   return "Pessoa";
}
```

As operações `save_state`, `restore_state` e `type` serão utilizadas pela classe `StateManager` no processo de ativação e desativação dos objetos persistentes.

### 4.4.3 Controle de Concorrência

O controle de concorrência do SGTA provê a garantia das propriedades de *isolamento* e *consistência* das transações. Tradicionalmente, a propriedade de isolamento busca a garantia da serialização, porém o controle de concorrência proposto não só provê o nível de serialização como provê níveis de isolamento mais flexíveis.

A consistência normalmente pode ser vista em termos de *estados* do sistema. Porém, existe uma primeira noção de consistência que pode não depender unicamente do sistema de transações e sim de predicados a serem definidos pelas aplicações. Por exemplo, considere o estado de uma conta bancária que se encontra com saldo negativo. Este estado poderia ser considerado inconsistente quando a conta é comum, ou seja, não aceita saldos negativos. Entretanto este estado também poderia ser considerado consistente se a conta é especial, ou seja, aceita saldos negativos. Nestes casos, para que uma transação que manipula o saldo de uma conta garantisse esta noção de consistência, ela dependeria de um predicado fornecido pela aplicação com regras específicas de consistência. São por estas razões que freqüentemente surgem discussões de que a consistência não é "por si só" uma propriedade de transações.

Existe uma segunda noção de consistência que pode depender unicamente do sistema de transações. Quando uma transação está em execução é comum que ela possua operações que tornem o estado do sistema temporariamente inconsistente ou indefinido. Porém, estas indefinições e inconsistências intermediárias só duram no máximo até a fase de efetivação da transação onde são todas resolvidas. O problema que poderá ocorrer é quando este estado intermediário é acessado por outras transações concorrentes. Este acesso poderá levar estas transações ao encontro com fenômenos de inconsistência como *escrita inconsistente*, *leitura inconsistente*, *leitura não repetível* e *leitura fantasma* que foram apresentados na Seção 3.3.1. Estes fenômenos de inconsistência podem ser evitados através de regras de serialização. Transações que executam sob as regras da serialização possuem suas propriedades de isolamento garantidas.

Diante destas duas noções de consistência apresentadas acima, o controle de concorrência do SGTA proverá mecanismos para garantir somente esta segunda noção de consistência. A primeira noção de consistência apresentada ficará aos cuidados dos usuários do sistema. O método utilizado para o controle de concorrência do SGTA será uma variante do método de *bloqueio em duas fases*. Basicamente, a única diferença entre o bloqueio em duas fases tradicional [9] e o que será utilizado no SGTA é que o tradicional provê somente um nível de isolamento (a serialização), enquanto que esta variante poderá utilizar quatro níveis de isolamento diferentes. Para garantir estes níveis de isolamento, o controle de concorrência utilizará trancas de longa e curta duração, como foi apresentado na Seção 3.3.1.

O controle de concorrência do SGTA é centrado nos objetos do usuário de modo que cada objeto possui operações para o seu controle de concorrência. Para que isto se torne possível, as classes que implementam os objetos do usuário deverão herdar a classe `LockManager` que provê um conjunto de operações para este controle de concorrência. Por sua vez, a classe `LockManager` herda da classe `StateManager`, portanto a classe do usuário

que herda da classe `LockManager` possuirá operações para o controle de concorrência, manutenção da recuperação e persistência de suas instâncias.

```
abstract class LockManager extends StateManager
{
   /* CONSTANTES */
   static final short GRANTED = 0, REFUSED = 1, RELEASED = 2;
   /* CONSTRUTOR */
   LockManager(String objectId);
   /* OPERAÇÕES PÚBLICAS */
   int setLock(Lock lock);
   int releaseLock(String lockId);
   /* OPERAÇÕES ABSTRATAS HERDADAS DE StateManager */
   abstract boolean restore_state(ObjectState objState);
   abstract boolean save_state(ObjectState objState);
   abstract String type();
}
```

O controle de concorrência do SGTA define a seguinte regra: antes que uma transação execute qualquer operação de leitura ou escrita sobre um objeto do usuário, ela deverá obter a permissão do `LockManager` responsável pelo objeto. Quando uma transação faz um pedido de acesso a um objeto, o `LockManager` poderá responder de três maneiras: (i) permitir o acesso de maneira imediata; (ii) bloquear a transação até que o permissão possa ser concedida; (iii) rejeitar a permissão de acesso. A operação `setLock` da classe `LockManager` possui a função de efetuar este pedido.

O programador das classes que definem objetos do usuário é o responsável por colocar a operação `setLock` no início do corpo de cada operação pública destas classes. Desta forma, os pedidos de acesso serão feitos de forma transparente à medida que a transação vai sendo executada. Um pedido de acesso pode ser de dois tipos: de leitura ou de escrita. Os pedidos de leitura devem ser feitos para operações que somente lêem o estado do objeto enquanto que os pedidos de escrita devem se feitos para operações que modificam o estado do objeto. O parâmetro `lock` da operação `setLock` é quem define se o pedido é de leitura ou de escrita.

O exemplo que segue mostra o uso da operação `setLock` na operação `setNome` da classe `Pessoa`. Como a operação `setNome` é uma operação de escrita, a operação `setLock` tentará obter do `LockManager` a permissão de escrita para executar a operação `setNome`. Se a permissão for concedida, o restante do código será executado, se a permissão não for concedida será emitida uma exceção do tipo `LockRefusedException`. Como a execução da operação `setNome` só será realizada quando a operação `setLock` terminar de executar, enquanto esta não der uma resposta (de concessão ou de rejeição do pedido de acesso), a transação que chamou a operação `setNome` ficará bloqueada, ou seja, impedida de continuar.

```
class Pessoa extends LockManager
{
   String nome;
   ...
   void setNome(String novoNome) throws LockRefusedException{
      this.setLock(new Lock(WRITE));
      this.nome = novoNome;
   }
   ...
}
```

O parâmetro da operação `setLock` é uma instância da classe `Lock`. Esta instância provê operações que serão usadas pelo `LockManager` no processo de controle de concorrência da seguinte forma: a operação `conflictsWith` informa se o respectivo pedido de acesso é conflitante com um outro pedido de acesso concedido anteriormente a outra transação; a operação `modifiesObject` informa se o pedido de acesso visa modificar ou não o objeto.

```
class Lock
{
   /* MODOS DE TRANCAS */
   static short READ = 0, WRITE = 1;
   /* CONSTRUTOR */
   Lock(int lockMode);
   /* OPERAÇÕES PÚBLICAS */
   boolean conflictsWith(Lock otherLock);
   boolean modifiesObject();
   ...
}
```

O construtor da classe `Lock` possui o parâmetro `lockMode` que indica se a instância a ser criada será associada a uma operação de escrita ou de leitura. Para operações de escrita pode-se utilizar como parâmetro a constante `WRITE` e para operações de leitura pode-se utilizar a constante `READ`. Estas constantes são definidas pela própria classe `Lock`.

Para mostrar como o `LockManager` efetua o controle de concorrência utilizando trancas, observemos o exemplo genérico abaixo:

```
class ObjetoA extends LockManager{
   operation1(){
      this.setLock(lockMode);
      /* EXECUTA CORPO DE operation1 */
   }
}
```

```
AtomicActionImp trn;
ObjetoA objA;
trn.Begin(...);
   objA.operation1();
   ...
trn.End();
```

Em relação a este exemplo observemos que:

1. `trn` é uma transação que efetuará operações sobre o objeto do usuário `objA`.
2. A classe `ObjetoA` herda de `LockManager`.
3. A operação `operation1` possui no início do seu corpo um pedido de permissão através da chamada ao método `setLock`.
4. O parâmetro `lockMode` da operação `setLock` em `operation1` indica se `operation1` é de leitura ou escrita.

Como o controle de concorrência é baseado em trancas, o `LockManager` tentará atribuir uma tranca adequada sobre o objeto que será acessado pela transação. Um objeto poderá conter várias trancas de diferentes transações que o acessam, porém estas trancas têm que ser compatíveis umas com as outras. Observemos a tabela de compatibilidade entre trancas:

| | **Leitura** | **Escrita** |
|---|---|---|
| **Leitura** | Compatível | Conflito |
| **Escrita** | Conflito | Conflito |

Tabela 4.1: Compatibilidade entre trancas.

A atribuição de trancas realizada pelo `LockManager` terá como base as seguintes informações: o parâmetro da operação `setLock` (que indica se a operação é de leitura ou escrita) e o nível de isolamento da transação requisitante (`trn`). Com isso, o `LockManager` tentará atribuir ao objeto `objA` uma tranca adequada à transação `trn`. A tabela 4.2 mostra os níveis de isolamento e as trancas respectivas.

| **Nível de Isolamento da Transação** | **Tipo operação a ser realizada pela transação** | |
|---|---|---|
| | **Leitura** | **Escrita** |
| Nível 0 | Não requer tranca | Requer tranca curta de escrita |
| Nível 1 | Não requer tranca | Requer tranca longa de escrita |
| Nível 2 | Requer tranca curta de leitura | Requer tranca longa de escrita |
| Nível 3 | Requer tranca longa de leitura | Requer tranca longa de escrita |

Tabela 4.2: Atribuição de Trancas.

Tendo em vista estas considerações, quando `operation1` é chamada, a operação `setLock` também será chamada. A partir de então o `LockManager` se baseará na tabela 4.2 para decidir se a execução de `operation1` pode ser efetuada ou não pela transação. Se o tipo de tranca requerido pela operação `operation1` for compatível com as eventuais trancas que já existam sobre `objA`, esta tranca será atribuída ao `objA` e a transação `trn` terá permissão para executar `operation1`. Se a tranca requerida para executar `operation1` não for compatível com outras trancas que possam existir sobre `objA`, o `LockManager` bloqueará a transação `trn` até que o acesso possa ser concedido.

Pode-se observar na Tabela 4.2 que transações no nível 0 ou 1 de isolamento não requerem nenhum tipo de tranca para operações de leitura sobre objetos. Nestes casos, o `LockManager` não impõe nenhuma restrição de acesso a um objeto acessado.

Todas as trancas atribuídas sobre objetos em nome de transações que os acessam são liberadas automaticamente. Se uma tranca for do tipo curta ela será liberada logo após o término da operação para a qual ela foi atribuída. Se uma tranca for do tipo longa ela só será liberada na fase de efetivação da transação para a qual ela foi atribuída. A operação utilizada na liberação de trancas é o `releaseLock` da classe `LockManager`.

Os objetos do usuário acessados pelas transações do SGTA possuem um controle de concorrência local e um controle de concorrência global. A diferença entre o local e o global é que este age sobre os objetos matrizes da rede fixa enquanto que aquele age sobre as cópias dos objetos da rede fixa mantidos na cache de uma unidade móvel. Se uma transação acessa a matriz de um objeto localizado na rede fixa, ela estará sujeita ao seu controle de concorrência global. Caso a transação acesse a cópia em cache de um objeto matriz, ela estará sujeita ao controle de concorrência local. Transações que executam sob o controle de concorrência local precisam ser submetidas a um processo de validação para que elas possam se tornar globalmente consistentes.

### 4.4.4 Cancelamento Parcial

Como foi apresentado nos capítulos iniciais, uma transação que executa na unidade móvel poderá acessar objetos localizados em máquinas da rede fixa. Como este acesso normalmente é feito via comunicação sem fio, as transações estarão sujeitas a interrupções abruptas em decorrência das desconexões inesperadas. As conseqüências destas interrupções podem ser: a transação perde o contato com a rede fixa e não terá como restaurar o estado dos objetos remotos da rede fixa nem terá como liberar as trancas que foram mantidas sobre os mesmos; outras transações poderão ser impedidas de utilizar os objetos remotos devido à existência de trancas da transação que perdeu a comunicação com a rede fixa.

Para lidar com os problemas descritos acima, foi implementado no SGTA um mecanismo chamado *cancelamento parcial*. Este mecanismo é responsável por liberar trancas e por restaurar estados de objetos da rede fixa que estavam sendo manipulados por transações que perderam o contato com a rede fixa. Este mecanismo é chamado cancelamento parcial porque é o mesmo que desfazer a parte das operações da transação que foram realizadas sobre objetos da rede fixa. O cancelamento parcial é implementado por cada `LockManager` responsável pelos objetos da rede fixa. A seguir é apresentada uma seqüência de passos que podem levar um `LockManager` a executar o cancelamento parcial de uma transação:

Considerações iniciais:

(i) `obj` é um objeto da rede fixa.

(ii) $T_1$ é uma transação da unidade móvel que mantém uma tranca $L_{obj}$ sobre `obj`.

(iii) $T_2$ é uma outra transação que deseja acessar obj mas é impedido pela tranca $L_{obj}$ de $T_1$.

Passos:

1. T2 pede ao LockManager permissão para executar uma operação em obj;
2. O LockManager verifica que $T_2$ terá que esperar até que $L_{obj}$ seja liberada;
3. Se $T_1$ não liberar $L_{obj}$ dentro de um tempo determinado então LockManager envia uma mensagem a $T_1$ e aguarda resposta. Se $T_1$ responder, $L_{obj}$ será mantido. Se $T_1$ não responder, o LockManager executará o cancelamento parcial ao considerar que $T_1$ está incomunicável. Operações do abort parcial:
    - o estado de obj será restaurado para o estado anterior à execução de $T_1$;
    - $L_{obj}$ será liberado;

O cancelamento parcial é, então, o mecanismo do SGTA que evita que os objetos da rede fixa sejam infinitamente trancados por transações desconectadas. Desta forma, as trancas de transações que executam sob o modo de operação remoto e que sofreram desconexão repentina estarão sujeitas à liberação automática pelos LockManager's responsáveis permitindo assim que outras transações continuem a acessar os objetos da rede fixa.

Quando uma transação é cancelada parcialmente pelo LockManager, ela terá que ser cancelada[14] por completo. Este cancelamento total da transação poderá ser feito quando ela retoma a conexão com a rede fixa e é notificada pelo LockManager de que ela foi cancelada parcialmente.

### 4.4.5 Validação e Atualização

Foi apresentado anteriormente que a única forma de replicação oferecida pelo SGTA é o caching, ou seja, os objetos matrizes estão na rede fixa enquanto que as caches das unidades móveis poderão conter cópias destes objetos matrizes. Diante deste modelo de replicação, foi apresentado que uma transação poderá acessar estes objetos sob três modos de operação: local, remoto e local-remoto. Quando uma transação executa sob o modo de operação remoto, ela acessa os objetos matrizes diretamente da rede fixa. Como cada objeto matriz possui um controle de concorrência implementado pela classe LockManager, a transação terá a propriedade de consistência garantida, sendo que esta noção de consistência dependerá do nível de isolamento escolhido pela transação.

Uma transação que executa sob o modo de operação local-remoto, apesar de acessar cópias locais dos objetos matrizes da rede fixa, também terá a sua propriedade de consistência garantida. Isto ocorre porque neste modo são atribuídas trancas aos objetos matrizes correspondentes aos que estão sendo acessados em cache. No modo local-remoto,

[14] O mesmo que abortar uma transação.

quando uma transação termina, seus resultados obtidos sobre as cópias locais têm que ser propagadas para suas respectivas matrizes remotas. Este processo é chamado *atualização*.

Uma transação que executa sob o modo de operação local, da mesma forma que o modo local-remoto, acessa cópias locais de objetos matrizes da rede fixa, a única diferença é que, ao contrário que no modo local-remoto, no modo local não são atribuídas trancas às matrizes remotas, portanto a transação em questão poderá vir a acessar cópias locais que estejam inconsistentes em relação a suas matrizes. Para lidar com este problema, no final da execução da transação que está sob o modo local é disparado um processo chamado *validação* da transação. Se a transação passar neste processo, as matrizes remotas serão também atualizadas da mesma maneira como é feito no modo local-remoto.

Antes de mostrar o protocolo de validação de transações que executam sob o modo de operação local, será apresentado o mecanismo utilizado para detectar se a cópia de um determinado objeto em cache é ou não consistente com sua matriz da rede fixa. Este mecanismo é baseado nas seguintes premissas:

1- todo objeto matriz na rede fixa possui um número de versão que é armazenado em um atributo do `StateManager` responsável pelo objeto;
2- quando o estado deste mesmo objeto é armazenado no Repositório de Objetos, seu número de versão é automaticamente armazenado juntamente com este estado;
3- o número de versão de um objeto só é incrementado no final da fase de efetivação de uma transação quando esta transação modificou o estado do objeto;
4- quando um objeto matriz é copiado para a cache de uma unidade móvel, seu número de versão também é copiado.

Protocolo de Validação

`T` – transação que será validada;
$O_i$ – objeto matriz da rede fixa;
$O_i'$ – cópia do objeto $O_i$ em cache;

**PARA** cada $O_i'$ acessado por `T` **FAÇA**
    Obtenha trancas de escrita sobre $O_i'$ e seu correspondente $O_i$ em nome de `T`;
    **SE** o número de versão de $O_i$ é igual ao número de versão de $O_i'$ **ENTÃO**
        As operações de `T` sobre $O_i'$ são válidas!
    **SENÃO**
        As operações de `T` sobre $O_i'$ são inválidas!
    **FIM (SE)**
**FIM (PARA)**

**SE** todas as operações de T forem válidas **ENTÃO**
    T é válida!
**SENÃO**
    T é inválida!
**FIM (SE)**

Se uma transação passar pelo processo de validação com sucesso, ela entrará na fase de atualização remota dos objetos. Senão, a transação será considerada inválida e será abortada. Os processos de validação e atualização são coordenados pela classe `AtomicAction`.

### 4.4.6 Processo de Efetivação

As seções anteriores apresentaram vários aspectos relacionados aos objetos manipulados por transações, dentre estes aspectos está a persistência, a recuperação, a gerência de trancas, a validação e a atualização remota. Para que a `AtomicAction` gerencie estes aspectos para cada um dos objetos participantes de uma transação, ela utiliza um conjunto de classes auxiliares que implementam a interface chamada `AbstractRecord`. Esta interface é definida como segue:

```
interface AbstractRecord
{
  /* OPERAÇÕES DE GERENCIAMENTO */
  int prepare();
  boolean commit();
  boolean abort();
  /* OPERAÇÕES DE IDENTIFICAÇÃO */
  String getId ();
  String getType ();
  ...
}
```

Esta interface possui operações como `prepare`, `commit` e `abort` que serão utilizadas pela `AtomicAction` no processo de efetivação ou cancelamento de uma transação. As operações `getId` e `getType` devolvem o identificador e o tipo do objeto participante que está sendo gerenciado. As classes auxiliares da `AtomicAction` implementam a interface `AbstractRecord` são:

1. `LockRecord` – gerencia a tranca atribuída a um determinado objeto.
2. `RecoveryRecord` – gerencia a recuperação de um determinado objeto.
3. `PersistenceRecord` – gerencia a persistência de um determinado objeto.
4. `UpdatingRecord` – gerencia a atualização remota de um determinado objeto.
5. `ValidationRecord` – gerencia o processo de validação de um determinado objeto.

Uma `AtomicAction` mantém instâncias destas classes para cada objeto participante da transação sendo que cada instância cuidará de um aspecto relacionado a um determinado objeto. Durante a execução da transação estas instâncias são inseridas em uma lista de `AbstractRecord's` chamada `pendingList`. Por exemplo, quando uma transação adquire uma tranca sobre um objeto ela adiciona em sua `pendingList` uma instância da classe

`LockRecord` que gerenciará a respectiva tranca, ou quando a transação modifica o estado de um objeto ela adiciona em sua `pendingList` uma instância da classe `RecoveryRecord` que gerenciará aspectos relacionados à recuperação do mesmo objeto. A seguir é mostrado um esboço da classe `AtomicAction` com algumas das estruturas utilizadas no seu gerenciamento:

```
abstract class AtomicAction
{
   /* LISTAS PRIVADAS DE AbstractRecord's */
   RecordList pendingList;
   RecordList preparedList;
   RecordList readonlyList;
   ...
   /* OPERAÇÕES PÚBLICAS */
   int Begin();
   int Abort();
   int End();
   int add(AbstractRecord ar);
   ...
   /* OPERAÇÕES PROTEGIDAS */
   int prepare();
   void phase2Commit();
   void phase2Abort();
   ...
}
```

Levando-se em conta esta estrutura apresentada acima, serão descritos como são realizados os processos de efetivação e de cancelamento de uma transação. Sabe-se que uma transação é delimitada pelas operações `Begin` e `End`. Quando uma transação termina, ou seja, quando a operação `End` é executada, o processo de efetivação é disparado. Este processo determinará se a transação poderá efetivar suas operações com sucesso ou se deverá ser abortada. O protocolo de efetivação implementado pela classe `AtomicAction` é o *Efetivação em Duas Fases*[15] [9]. A primeira fase deste processo é chamar a operação `prepare` (da interface `AbstractRecord`) para cada registro contido na `pendingList`. À medida que cada registro vai sendo processado ele é movido para a `preparedList` ou para a `readonlyList` dependendo se o registro precisa ou não fazer parte da segunda fase do protocolo de efetivação. Cada chamada à operação `prepare` retorna um valor que indica se a mesma obteve sucesso ou não. Se alguma das operações `prepare` que foram chamadas na `pendingList` não obtiver sucesso, a transação será abortada na segunda fase do protocolo.

Uma vez que a fase de prepare tenha terminado, a `AtomicAction` chamará a operação `phase2Commit` ou `phase2Abort` a depender do seu resultado. Se a fase de prepare for executada com sucesso, a operação `phase2Commit` será chamada, senão a operação

[15] Em inglês: Two-Phase Commit.

`phase2Abort` será chamada. A operação `phase2Commit` possui a ação de chamar a operação `commit` para cada elemento da `preparedList`. Já a operação `phase2Abort` irá chamar a operação `abort` para cada elemento da `preparedList` e da `readonlyList`.

Uma transação que termina de executar o protocolo de *Efetivação em Duas Fases* poderá estar em um dos dois estados: *efetivada* ou *cancelada*. Uma transação efetivada terminou sua execução com sucesso e teve o seu resultado feitos permanente. Uma transação cancelada não obteve sucesso e os seus resultados foram desfeitos.

## 4.5 Fila de Efetivações

O SGTA provê mecanismos para que uma transação na unidade móvel possa executar evitando ao máximo a comunicação com a rede fixa. Isto é possibilitado através do uso dos modos de operação *local* ou *local-remoto* que permitem até que a transação execute mesmo quando desconectada da rede fixa. Porém, para que as propriedades de consistência e durabilidade da transação possam ser garantidas, pode ser necessário que na fase de efetivação da transação ela precise da comunicação com a rede fixa para realizar processos de validação e atualização dos objetos da rede fixa. Se na fase de efetivação a conexão com a rede fixa estiver estabelecida, a própria `AtomicAction` se responsabilizará pelo processo de efetivação. Entretanto, se na fase de efetivação da transação a unidade móvel estiver desconectada da rede fixa, a transação delegará o seu processo de efetivação a uma fila de efetivações e entrará no estado *pendente*.

A fila de efetivações é implementada pela classe `CommitQueue`. Cada unidade móvel contém uma fila de efetivações que poderá manter vários processos de efetivação de diferentes transações pendentes. Quando uma transação da unidade móvel delega o seu processo de efetivação para a `CommitQueue` ela transfere cada registro da sua `pendingList` para a `CommitQueue`. Uma vez realizada esta delegação, a `CommitQueue` se responsabilizará por realizar o processo de efetivação da transação assim que a conexão com a rede fixa for restabelecida. Quando a `CommitQueue` recebe o processo de efetivação de uma transação ela se responsabilizará por realizar basicamente os seguintes passos:

(i) detectar o restabelecimento da conexão;
(ii) executar o processo de efetivação para cada transação da fila;
(iii) se o processo de efetivação de uma transação resultar em sucesso, enviar uma notificação de sucesso à transação para que ela passe para o estado *efetivada*;
(iv) se o processo de efetivação de uma transação falhar, abortar a transação e enviar uma notificação de falha para a transação de modo que esta passe para o estado *cancelada*.

O processo de efetivação executado pela `CommitQueue` no item (ii) é o mesmo que o processo de efetivação de uma transação que foi apresentado anteriormente. Sendo assim, este processo é baseado em chamadas às operações `prepare` e `commit` aos elementos da `pendingList` da transação que foram delegados para a `CommitQueue`.

## 4.6 Monitoramento de Recursos

Como foi apresentado na Seção 3.2.1, o monitor de recursos é um componente da arquitetura responsável por prover serviços de monitoramento dos recursos no ambiente de computação móvel e por prover serviços de notificação a transações. O monitor de recursos é implementado pela classe `ResourceMonitor` que agrega módulos independentes, sendo que cada módulo é responsável pelo monitoramento de um recurso. A classe `ResourceMonitor` é definida como segue:

```
class ResourceMonitor
{
   /* CONSTRUTOR */
   ResourceMonitor();
   /* OPERAÇÕES VISÍVEIS ÀS TRANSAÇÕES */
   String request(AtomicAction action, ResourceDescriptor descriptor);
   int cancel(String requestId);
   int getLevel(String monitorId);
   /* OPERAÇÕES VISÍVEIS AO ADMINISTRADOR */
   int addMonitor(MonitorInterface monitor);
   MonitorInterface getMonitor(String resourceId);
   int run();
   int stop();
   ...
}
```

A operação `request` pode ser chamada por uma transação para requisitar o monitoramento de um determinado recurso. O parâmetro `action` é a referência para a transação que requisitou o monitoramento. O parâmetro `descriptor` descreve o tipo de monitoramento que deve ser realizado. Este parâmetro é definido pela classe `ResourceDescriptor` que encapsula os atributos `resourceId`, `lowerBound` e `upperBound`. O atributo `resourceId` identifica o tipo de recurso a ser monitorado enquanto que os atributos `lowerBound` e `upperBound` definem os limites inferior e superior do recurso que são toleráveis pela transação requisitante.

Quando uma transação chama a operação `request` do monitor de recursos, ela recebe como retorno o identificador da requisição. Este mesmo identificador poderá ser usado posteriormente para cancelar a requisição de monitoramento, caso seja preciso. Para efetuar este cancelamento deve ser chamada a operação `cancel`. O parâmetro que deve ser passado nesta operação é justamente o identificador da requisição (`requestId`).

Outra operação do monitor de recursos que poderá ser chamada por uma transação é o `getLevel`. Esta operação devolve o nível atual de um determinado recurso identificado pelo parâmetro `resourceId`.

As demais operações do monitor de recursos são operações que só são visíveis ao administrador do sistema através de uma interface específica. A operação `addMonitor` tem a função de adicionar novos módulos de monitoramento de recursos. A operação

`getMonitor` devolve um módulo de monitoramento de recurso específico. As operações `run` e `stop` têm a função de executar e parar a execução do monitor de recursos, respectivamente.

Para que um novo módulo de monitoramento possa ser adicionado ao monitor de recursos, este deverá implementar a interface `MonitorInterface`. Esta interface possui as operações básicas que todo módulo de monitoramento deve conter. As operações de um módulo de monitoramento serão usadas pelo monitor de recursos para gerenciá-lo. Segue a interface:

```
interface MonitorInterface
{
   long request(AtomicAction action, int lowerBound, int upperBound );
   int cancel(long requestId);
   String getId();
   int getLevel();
   int _run();
   int _stop();
}
```

Um módulo de monitoramento deverá receber requisições do monitor de recursos através da operação `request` que contém como parâmetros: a referência da transação requisitante (`action`) e os limites inferior (`lowerBound`) e superior (`upperBound`) toleráveis pela mesma transação. A chamada da operação request devolve o identificador da requisição (`requestId`) que poderá ser útil no cancelamento da requisição. Caso o recurso monitorado extrapole um dos limites especificados, o módulo de monitoramento deverá notificar a transação correspondente. Para que seja efetuada esta notificação o módulo de monitoramento deverá chamar a operação `notify` da transação provida através do parâmetro `action`. A partir desta notificação, a transação poderá tomar as medidas necessárias de adaptação à mudança.

As demais operações que deverão ser implementadas por um módulo de monitoramento de recurso são: `cancel` - serve para cancelar uma determinada requisição a partir do identificador da requisição (`requestId`); `getId` - devolve o identificador do módulo de monitoramento de recursos; `getLevel` - devolve o nível atual do recurso monitorado pelo módulo; as operações `_run` e `_stop` têm a função de executar e cancelar a execução do módulo de monitoramento, respectivamente.

No protótipo do SGTA, foram implementados os seguintes módulos de monitoramento:

- Largura de banda – responsável pelo monitoramento da largura de banda entre a unidade móvel e a rede fixa; pode ser referenciado na operação `request` do monitor de recursos através do identificador “bandwidth”; a unidade de medida utilizada é bps.
- Energia – responsável pelo monitoramento da energia mantida pelas baterias da unidade móvel; pode ser referenciado na operação `request` do monitor de recursos

através do identificador “energy”; a energia armazenada pela unidade móvel é medida através do tempo estimado (em segundos) para o seu esgotamento total.

- Espaço em disco – responsável pelo monitoramento do espaço em disco da unidade móvel; pode ser referenciado na operação `request` do monitor de recursos através do identificador “disk”; a unidade de medida utilizada é bytes.
- Custo da comunicação – responsável pelo monitoramento do custo da comunicação entre a unidade móvel e a rede fixa; pode ser referenciado na operação `request` do monitor de recursos através do identificador “cost”; a unidade de medida utilizada é R$/minuto.

Ao reconhecer que a implementação de módulos de monitoramento de recursos é uma atividade que requer um grande volume de tempo que não podemos desprender na implementação do protótipo, estes quatro módulos de monitoramento foram implementados como simuladores que geram valores que simulam a disponibilidade de cada recurso em um ambiente de computação móvel. Desta forma, pôde-se testar a capacidade de reação das transações do SGTA diante das variações do ambiente.

## 4.7 Objetos do Usuário

Os objetos do usuário são os elementos alvo de qualquer transação do SGTA, ou seja, cada operação de leitura ou escrita realizada por uma transação é feita através de operações providas pelos objetos do usuário. Segue um exemplo simples:

```
MyAtomicAction trn;

ObjectA objA;
ObjectB objB;        } Objetos do usuário;

trn.Begin();
   objA.operation1();
   objB.operation2();
   objA.operation2();
   objA.operation3();
trn.End();
```

*Operações (leitura e/ou escrita) da transação sobre os objetos do usuário;*

Além de operações de leitura e escrita sobre objetos do usuário que são usadas pelas transações no decorrer de sua execução, existe uma série de outras operações, mostradas nas seções anteriores, que auxiliam o gerenciamento de propriedades das transações, bem como o controle de concorrência, o gerenciamento da persistência e da recuperação. Existe um outro conjunto de operações auxiliares sobre os objetos do usuário que apóiam o processo de adaptação no que diz respeito ao modo de operação. Este conjunto de operações auxiliar é implementadas pela classe `MobileObject`. Basicamente, a classe `MobileObject` possui operações que manipulam objetos do usuário de maneira que estes possam atender as exigências de cada modo de operação de uma transação. Por exemplo, se uma transação que acessa o `objA` opta pelo modo de operação local, o

MobileObject tratará de copiar o objA para cache da unidade móvel (caso ele não esteja em cache) para que o mesmo objeto possa ser acessado localmente pela transação.

Nas seções que seguem serão apresentados aspectos da implementação da classe MobileObject e serão mostrados os passos para adicionar novos objetos do usuário ao sistema.

### 4.7.1 A Classe MobileObject

A classe MobileObject é a responsável por manipular objetos do usuário de maneira que eles possam atender as exigências de cada um dos modos de operação. As operações desta classe podem ser enquadradas em dois grupos:

**Grupo 1.** Operações que configuram o objeto do usuário para inicialmente atender às exigências de um determinado modo de operação.

**Grupo 2.** Operações que configuram o objeto do usuário para atender a mudanças entre os modos de operação.

Cada instância da classe MobileObject é responsável por manipular um único objeto do usuário. Para isto são passados no seu construtor informações que identificam o objeto que será manipulado, bem como identificadores da unidade móvel e da sua localização remota. Além destas informações, o construtor de MobileObject também possuirá uma referência à transação (AtomicAction). Desta forma, a classe MobileObject poderá conhecer o modo de operação que deverá ser atendido para a transação. A seguir tem-se um esboço da classe MobileObject:

```
class MobileObject
{
   /* CONSTRUTOR */
   MobileObject(String objectId, String objectType,
                String localServer, String localPort,
                String remoteServer, String remotePort,
                AtomicAction atomicAction);
   /* OPERAÇÕES DO GRUPO 1 */
   org.omg.CORBA.Object firstOperationLOCAL();
   org.omg.CORBA.Object firstOperationLOCAL_REMOTE();
   org.omg.CORBA.Object firstOperationREMOTE();
   /* OPERAÇÕES DO GRUPO 2 */
   org.omg.CORBA.Object changeLOCALtoREMOTE();
   org.omg.CORBA.Object changeLOCALtoLOCAL_REMOTE();
   org.omg.CORBA.Object changeREMOTEtoLOCAL();
   org.omg.CORBA.Object changeREMOTEtoLOCAL_REMOTE();
   org.omg.CORBA.Object changeLOCAL_REMOTEtoLOCAL();
   org.omg.CORBA.Object changeLOCAL_REMOTEtoREMOTE();
   ...
   org.omg.CORBA.Object getReference();
}
```

A seguir será dada uma visão geral das tarefas que são realizadas pelas operações dos grupos 1 e 2 da classe `MobileObject`:

- `firstOperationLOCAL` – Usada para configurar um objeto para ser usado pela primeira vez por uma transação que se encontra sob o modo de operação local. Passos:

  (i) Utiliza o gerenciador de cache para verificar se existe cópia do objeto na cache da unidade móvel. Caso não exista, pede ao gerenciador para efetuar uma cópia do objeto matriz da rede fixa para a cache.
  (ii) Para gerenciar os aspectos que dizem respeito a uma possível futura validação do objeto, é adicionado um `ValidationRecord` à `pendingList` da `AtomicAction` responsável.
  (iii) Recupera a referência do objeto local em cache e devolve como resultado da operação.

- `firstOperationLOCAL_REMOTE` – Usada para configurar um objeto para ser usado pela primeira vez por uma transação que se encontra sob o modo de operação local-remoto. Passos:

  (i) Recupera a referência do objeto matriz da rede fixa.
  (ii) Atribui uma tranca de leitura ao objeto matriz da rede fixa. Para gerenciar esta tranca, é adicionado à `pendingList` da `AtomicAction` responsável um `LockRecord`.
  (iii) Através do gerenciador de cache verifica se existe cópia do objeto em cache e se ela está consistente em relação a sua matriz da rede fixa. Caso contrário, o objeto em cache será atualizado ou copiado a partir de sua matriz na rede fixa.
  (iv) Recupera a referência do objeto em cache e devolve como resultado da operação.

- `firstOperationREMOTE` – Usada para configurar um objeto para ser usado pela primeira vez por uma transação que se encontra sob o modo de operação remoto. Passos:

  (i) Recupera a referência do objeto matriz na rede fixa e devolve como resultado da operação.

- `changeLOCALtoREMOTE` – Usada para configurar um objeto quando a transação que o manipula muda seu modo de operação de local para remoto. Passos:

  (i) Realiza o processo de validação entre a cópia local e a sua matriz remota.
  (ii) Se o estado do objeto em cache foi modificado pela transação, a sua matriz na rede fixa será atualizada com este novo estado. Antes que esta atualização seja realizada é atribuída uma tranca de escrita à matriz na rede fixa. Para gerenciar

esta tranca, é adicionado à `pendingList` da `AtomicAction` responsável um `LockRecord`.

(iii) Libera todas as trancas da `AtomicAction` mantidas sobre o objeto em cache e recupera o estado do mesmo para o estado anterior à execução da transação.

(iv) Devolve como resultado da operação a referência do objeto matriz na rede fixa.

- `changeLOCALtoLOCAL_REMOTE` – Usada para configurar um objeto quando a transação que o manipula muda seu modo de operação de local para local-remoto. Passos:

  (i) Realiza o processo de validação entre a cópia local e a sua matriz remota.

  (ii) Se o estado objeto em cache foi modificado pela transação, a sua matriz na rede fixa será atualizada com este novo estado. Antes que esta atualização seja realizada é atribuída uma tranca de escrita à matriz na rede fixa. Para gerenciar esta tranca, é adicionado à `pendingList` da `AtomicAction` responsável um `LockRecord`.

  (iii) Devolve como resultado da operação a referência do objeto na cache da unidade móvel.

- `changeREMOTEtoLOCAL` – Usada para configurar um objeto quando a transação que o manipula muda seu modo de operação de remoto para local. Passos:

  (i) Através do gerenciador de cache verifica se existe cópia do objeto em cache e se ela está consistente em relação a sua matriz da rede fixa. Caso contrário, o objeto em cache será atualizado ou copiado a partir de sua matriz na rede fixa.

  (ii) Para gerenciar os aspectos que dizem respeito a uma possível futura validação do objeto, é adicionado um `ValidationRecord` à `pendingList` da `AtomicAction` responsável.

  (iii) Libera todas as trancas da `AtomicAction` mantidas sobre o objeto matriz na rede fixa e recupera o estado do mesmo para o estado anterior à execução da transação.

  (iv) Recupera a referência do objeto local em cache e devolve como o resultado da operação.

- `changeREMOTEtoLOCAL-REMOTE` – Usada para configurar um objeto quando a transação que o manipula muda seu modo de operação de remoto para local-remoto. Passos:

  (i) Através do gerenciador de cache verifica se existe cópia do objeto em cache e se ela está consistente em relação a sua matriz da rede fixa. Caso contrário, o objeto em cache será atualizado ou copiado a partir de sua matriz na rede fixa.

  (ii) Atribui uma tranca de leitura ao objeto matriz da rede fixa. Para gerenciar esta tranca, é adicionado à `pendingList` da `AtomicAction` responsável um `LockRecord`.

(iii) Recupera a referência do objeto local em cache e devolve como o resultado da operação.

- `changeLOCAL_REMOTEtoLOCAL` – Usada para configurar um objeto quando a transação que o manipula muda seu modo de operação de local-remoto para local. Passos:

  (i) Libera todas as trancas da `AtomicAction` mantidas sobre o objeto matriz na rede fixa e recupera o estado do mesmo para o estado anterior à execução da transação.
  (ii) Para gerenciar os aspectos que dizem respeito a uma possível futura validação do objeto, é adicionado um `ValidationRecord` à `pendingList` da `AtomicAction` responsável.
  (iii) Devolve como o resultado da operação a referência do objeto local em cache.

- `changeLOCAL-REMOTEtoREMOTE` – Usada para configurar um objeto quando a transação que o manipula muda seu modo de operação de local-remoto para remoto. Passos:

  (i) Se o estado objeto em cache foi modificado pela transação, a sua matriz na rede fixa será atualizada com este novo estado. Antes que esta atualização seja realizada é atribuída uma tranca de escrita à matriz na rede fixa. Para gerenciar esta tranca, é adicionado à `pendingList` da `AtomicAction` responsável um `LockRecord`.
  (ii) Libera todas as trancas da `AtomicAction` mantidas sobre o objeto em cache e recupera o estado do mesmo para o estado anterior à execução da transação.
  (iii) Devolve como resultado da operação a referência do objeto matriz na rede fixa.

As operações do grupo 1 e 2 são todas utilizadas pela operação `getReference`. Esta operação possui a função de reconhecer o modo de operação de uma transação e de chamar a devida operação (do grupo 1 ou 2) que configure o objeto do usuário de maneira que ele possa atender respectivo modo de operação.

### 4.7.2 Criação de Objetos do Usuário

Nesta seção será apresentado o processo necessário para a criação e instalação de objetos do usuário no SGTA. Este processo será dividido nas seguintes etapas: definição da interface remota; compilação da interface remota; implementação do objeto remoto; implementação do wrapper. Cada uma destas etapas será definida a seguir. Em paralelo à definição destas etapas será mostrado um exemplo que consiste na definição de um contador simples com três operações: `inc` – incrementar o contador; `set` – atribuir um valor ao contador; `get` – receber o valor do contador.

### Definição da Interface Remota

A interface remota de um objeto do usuário pode ser entendida, em particular, como o conjunto de operações que poderão ser executadas remotamente por uma transação que se encontra na unidade móvel. Esta interface deverá ser escrita com a linguagem neutra IDL (Interface Definition Language) que possibilitará a definição de cada operação que o objeto do usuário deverá conter bem como os parâmetros de cada operação.

Exemplo: a seguir é mostrada a interface `Counter` escrita em IDL:

```
#include "LockManager.idl"
interface Counter:LockManagerInterface
{
   void inc() raises(LockRefusedException);
   void set(in long value) raises(LockRefusedException);
   long get() raises(LockRefusedException);
};
```

Observando a definição da interface `Counter` acima serão destacados alguns padrões que toda interface IDL que define objetos do usuário no SGTA deverá conter:

(i) a interface IDL deverá ser escrita em um arquivo texto com a extensão "`.idl`". No exemplo acima a interface `Counter` será armazenada em um arquivo nomeado "`Counter.idl`";

(ii) a interface IDL deverá conter a diretiva `#include` que faz referência ao arquivo "`LockManager.idl`". O arquivo "`LockManager.idl`" possui uma interface predefinida na implementação do SGTA que define operações do gerenciador de locks.

(iii) a interface IDL deverá herdar da interface `LockManagerInterface`. No exemplo acima, pode-se observar que o símbolo de herança é "`:`". Desta forma, `Counter:LockManagerInterface` significa que a interface `Counter` herda da interface `LockManagerInterface`.

(iv) cada operação da interface IDL deverá propagar a exceção `LockRefusedException`. Como pode ser visto no exemplo acima, isto é feito através do símbolo `raises`.

A interface `LockManagerInterface` e a exceção `LockRefusedException` que são referenciadas na definição da interface de um objeto do usuário estão definidas no arquivo "`LockManager.idl`". Daí vem a necessidade da existência da referência para este arquivo na diretiva `#include`.

### Compilação da Interface Remota

Tendo o arquivo com extensão "`.idl`" que contém a interface que define o objeto do usuário, agora se deve compilar esta interface com o compilador `idlj`. Quando o `idlj` é

usado para compilar uma interface IDL, ele gera a versão Java da interface bem como as classes que representam stubs e skeletons necessários para a interação do objeto com o ORB. Com base no exemplo `Counter` citado anteriormente será mostrado como um objeto do usuário pose ser compilado com o `idlj`:

(i) executa-se o compilador `idlj` com a opção de linha de comando `-fallTie` para o arquivo IDL que contém a definição da interface. Para o exemplo da interface seria usado na linha de comando: `idlj – fallTie Counter.idl`.
(ii) a compilação do arquivo "`Counter.idl`" gerará os seguintes arquivos:
- `CounterPOA.java` – classe abstrata que representa o skeleton do servidor da rede fixa e provê funcionalidades CORBA para este servidor;
- `_CounterStub.java` – classe que representa o stub do cliente na unidade móvel e que provê funcionalidades CORBA para este cliente;
- `Counter.java` – interface que contém a versão Java da interface IDL que foi compilada;
- `CounterHelper.java` – classe que provê funcionalidade auxiliar como o *typecasting* de objetos CORBA para tipo `Counter`;
- `CounterHolder.java` – classe final que mantêm uma instância da classe `Counter` e algumas operações que a manipula;
- `CounterOperations.java` – interface que contém os métodos definidos na interface IDL;
- `CounterPOATie.java` – classe que permite o uso do modelo Tie [22] para representação do skeleton do servidor;

Quando estes arquivos forem gerados pelo `idlj`, o processo de compilação da interface IDL estará concluído. Nos itens a seguir será mostrado de maneira prática qual a utilidade destes arquivos na definição de um objeto do usuário.

**Implementação do Objeto do Usuário**

Agora será apresentado como as operações de um objeto do usuário são implementadas. Toda implementação de objetos do usuário no SGTA deverá herdar da classe `LockManager` e implementar a interface `*Operations`[16] gerada pelo compilador `idlj`. Voltando ao exemplo anterior, serão implementadas as operações do Contador que foi definido na interface IDL. Este contador é implementado pela classe `CounterServant` como é mostrado a seguir:

[16] "*" é o nome da interface idl que foi compilada.

```
class CounterServant extends LockManager implements CounterOperations
{
   private int cont = 0;

   /* CONSTRUTOR */
   public CounterServant(String objectId){
      super(objectId);
   }

   /* OPERÇÕES QUE IMPLEMENTAM A INTERFACE CounterOperations */
   public void inc() throws LockRefusedException{
      this.setLock(new Lock(WRITE));
      this.cont++;
   }

   public int get() throws LockRefusedException{
      this.setLock(new Lock(READ));
      return this.cont;
   }

   void set(int value) throws LockRefusedException{
      this.setLock(new Lock(WRITE));
      this.cont = value;
   }

   /* IMPLEMENTAÇÃO DAS OPERAÇÕES ABSTRATAS HERDADAS DE LockManager */
   public boolean save_state(ObjectState objState){
      return objState.packInt(this.cont);
   }

   public boolean restore_state(ObjectState objState){
      this.cont = objState.unpackInt();
      return true;
   }

   public String type(){
      return "Counter";
   }
}
```

Pode-se observar que a classe `CounterServant` herda da classe `LockManager` e implementa a interface `CounterOperations`. A classe `LockManager` provê operações para o controle de concorrência das transações que por ventura acessarão as operações do contador. Para que este controle de concorrência de um objeto possa ser realizado pelo `LockManager`, cada operação que implementa a interface `*Operations` deverá conter no seu início uma chamada à operação `setLock` herdada do próprio `LockManager` com o parâmetro adequado que indica se a operação é de leitura ou de escrita.

Como mostrado no exemplo da classe `CounterServant` acima, toda classe que define objetos do usuário deverá implementar as três operações abstratas herdadas da classe `LockManager`. Estas operações serão utilizadas pelo `StateManager` para garantir as propriedades de persistência e recuperação dos objetos do usuário. A operação `save_state` deve utilizar as operações do `ObjectState` passado como parâmetro para

salvar cada variável de instância que compõe o estado do objeto. Para o exemplo da classe `CounterServant`, a variável de instância que deverá ser armazenada é o atributo `cont`. A operação `restore_state` deve fazer exatamente o contrário da operação `save_state`, a partir do `ObjectState` passado como parâmetro deve-se recuperar cada item que compõe o estado do objeto e atribuir à respectiva variável de instância. A operação `type` deverá devolver como `String` o tipo do objeto.

### Implementação do Ativador de Objetos

Depois de concluir a implementação da classe que define objetos do usuário, agora deverá ser definida a classe responsável pelo processo de ativação de objetos desta classe sob demanda. Esta classe será chamada de `ServantActivator` e ela permite que os objetos do usuário só sejam carregados para a memória principal quando alguma aplicação requisite operações sobre os mesmos.

Deve ser implementado um `ServantActivator` para cada classe que define objetos do usuário. A classe que implementa um `ServantActivator` deverá herdar da classe `LocalObject` e implementar a interface `ServantActivator` dos pacotes `org.omg.CORBA` e `org.omg.PortableServer` respectivamente que estão disponíveis na plataforma J2SE 1.4. A seguir é mostrado como seria a implementação de um `ServantActivator` para o exemplo do contador que foi implementado.

```
import org.omg.PortableServer.*;
import org.omg.CORBA.*;

class CounterServantActivator extends LocalObject
                              implements ServantActivator
{
   /* CONSTRUTOR */
   public CounterServantActivator(){}

   /* OPERAÇÕES DA INTERFACE ServantActivator */
   public Servant incarnate(byte[] oid, POA adapter)
   {
      CounterServant servant = new CounterServant(new String(oid));  (i)
      servant.activate();                                            (ii)
      CounterPOATie tie = new CounterPOATie(servant, adapter);       (iii)
      return tie;                                                    (iv)
   }

   public void etherealize(byte[] oid, POA adapter, Servant servant,
                           boolean cleanup_in_progress,
                           boolean remaining_activations)
   {
   }
}
```

Como pode ser observado no exemplo acima, um ativador de objetos deve implementar as operações `incarnate` e `etherealize`. A operação `incarnate` deverá ser

implementada de forma a ativar um objeto de uma determinada classe identificado pelo parâmetro `oid`. A operação `etherealize` deverá ser implementada para desativar o objeto identificado pelo parâmetro `oid`. A execução destas operações é comandada pela arquitetura CORBA implementada no JavaIDL. Mais detalhes destas operações são descritos na documentação do J2SE 1.4 [22].

No SGTA, em particular, o ativador de objetos do usuário deverá implementar a operação `incarnate` da seguinte maneira: (i) instanciar o objeto do usuário identificado pelo parâmetro `oid` a partir da classe que o implementa; (ii) chamar a operação `activate` do objeto do usuário que foi instanciado; instanciar a classe `*POATie` gerada pelo compilador `idlj` passando como parâmetro o objeto do usuário e o POA do parâmetro da operação `incarnate`; (iv) devolver a instância da classe `*POATie` como retorno da operação `incarnate`. Cada um destes itens pode ser identificado no exemplo do ativador de objetos `CounterServantActivator` acima.

Pode-se observar que no exemplo da classe `CounterServantActivator` a operação `etherealize` não possui nenhuma implementação em específico. A função da operação `etherealize` seria desativar o objeto do usuário, porém, no SGTA, o processo de desativação de um objeto fica a cargo do próprio SGTA. Portanto, nenhuma implementação específica precisa ser feita para esta operação.

### Implementação do Servidor de Objetos

Depois de ter desenvolvido o ativador de objetos do usuário de uma determinada classe, deve-se implementar o servidor de objetos. Um servidor de objetos tem a função de disponibilizar objetos do usuário de modo que possam ser acessados remotamente através de um ORB. Continuando com o exemplo do contador, será mostrado como deve ser implementado um servidor de objetos do usuário.

```
public class CounterServer extends ObjectServer
{
   public CounterServer(){ }

   public static void main(String args[]){
      /* (i) INSTANCIA E INICIA O SERVIDOR DE CONTADORES */
      CounterServer objectServer = new CounterServer();
      counterServer.initORB(args);
      /* (ii) INSTANCIA O ATIVADOR DE OBJETOS CONTADOR */
      CounterServantActivator csa = new CounterServantActivator();
      counterServer.getPOA().set_servant_manager(csa);

      /* (iii) REGISTRA OBJETOS DO TIPO CONTADOR NO SERVIÇO DE NOMES */
      String objectType = "Counter";
      registerObject("c1", objectType);
      registerObject("c2", objectType);
      registerObject("c3", objectType);
   }
}
```

Para facilitar o desenvolvimento de um servidor de objetos simples, o SGTA provê a classe `ObjectServer`. Esta classe possui as seguintes operações: `initORB`, `getPOA` e `registerObject`. A operação `initORB` inicia e configura um ORB para receber requisições remotas. A operação `getPOA` devolve o adaptador portátil onde estarão conectados os objetos do usuário que serão disponibilizados. A operação `registerObject` serve para registrar no serviço de nomes cada um dos objetos identificados pelo seu `id` e `type`.

Como pode ser acompanhado no exemplo da classe `CounterServer`, um servidor de objetos deverá herdar a classe `ObjectServer` disponível no SGTA, implementar o método `main` da seguinte forma:

(i) instanciar o servidor de objetos e iniciar o ORB;
(ii) instanciar o ativador de objetos e passá-lo como parâmetro da operação `set_servant_manager` do POA;
(iii) registrar no serviço de nomes cada um dos objetos do usuário no serviço de nomes de modo que suas referências possam ser recuperadas remotamente.

No exemplo acima, se o método `main` fosse executado, três contadores seriam registrados no serviço de nomes e estariam prontos para serem acessados remotamente. Os contadores seriam `c1`, `c2` e `c3`. Apesar destes três contadores estarem registrados no serviço de nomes, inicialmente eles ainda não estão instanciados. A instanciação de um destes contadores só ocorrerá quando alguma aplicação recuperar a sua referência registrada no serviço de nomes e requisitar alguma de suas operações. Quando alguma operação for requisitada, o ativador de objetos `CounterServantActivator` tratará de instanciar o objeto e carregar o seu estado passivo em um estado ativo.

### Implementação do Wrapper

A implementação do wrapper é a última etapa da criação e disponibilização de um objeto de forma que este possua mecanismos necessários para serem manipulados por uma transação do SGTA. Foram apresentados vários exemplos como este em que uma transação chama operações de um objeto do usuário:

```
ObjectA objA;
trn.Begin();
   objA.operation1();
   ...
trn.End();
```

No SGTA, para que um objeto do usuário atenda as exigências dos diversos modos de operação, ele deve ser acessado indiretamente pela transação através de um wrapper. Sendo assim, neste exemplo acima se considera que `objA` seria um wrapper que controla um objeto do usuário ao invés do objeto do usuário em si. Por exemplo, este wrapper

determina se `operation1` será executada sobre a cópia em cache ou sobre a matriz da rede fixa a depender do modo de operação da transação.

Para cada classe que define um objeto do usuário deve ser implementado um wrapper. Um elemento fundamental desta implementação é a classe `MobileObject` que foi apresentada anteriormente. Esta classe assume todo o processo de configuração de um objeto de forma que ele se torne disponível às exigências de cada modo de operação da transação que o acessa. O uso da classe `MobileObject` na definição de um wrapper torna esta um tarefa fácil. A seguir é mostrado um exemplo da implementação de um wrapper para o exemplo do contador que está sendo acompanhado:

```
public class MobileCounter extends MobileObject
                           implements CounterOperations
{
   /* CONSTRUTOR */
   public MobileCounter(String objectId, String objectType,
                        String localServer, String localPort,
                        String remoteServer, String remotePort,
                        AtomicAction atomicAction)
   {
      super(objectId, objectType, localServer, localPort,
            remoteServer, remotePort, atomicAction);
   }
   /* OPERAÇÕES QUE IMPLEMENTAM A INTERFACE CounterOperations */
   public void inc() throws LockRefusedException{
      Counter c = CounterHelper.narrow(this.getReference());
      c.inc();
   }
   public int get() throws LockRefusedException{
      Counter c = CounterHelper.narrow(this.getReference());
      return c.get();
   }
   public void set(int value) throws LockRefusedException{
      Counter c = CounterHelper.narrow(this.getReference());
      c.set(value);
   }
}
```

Observando-se a implementação do wrapper `MobileCounter` acima serão levantadas algumas considerações quanto à implementação de wrappers para objetos do usuário em geral:

(i) A classe que define o wrapper deverá herdar da classe `MobileObject` e implementar a interface `*Operations` que é gerada pelo compilador `idlj`;

(ii) O construtor deverá conter os seguintes parâmetros: `objectId` – identifica um objeto do usuário; `objectType` – indica o tipo do objeto do usuário; `localServer` – nome da máquina local (unidade móvel); `localPort` – porta

da máquina local onde se encontra o serviço de nomes local; `remoteServer` – nome da máquina remota (da rede fixa) onde se encontra a matriz remota do objeto do usuário; `remotePort` – porta da máquina remota onde se encontra o serviço de nomes da máquina remota; `atomicAction` – referência da `AtomicAction` que acessará o objeto do usuário. Todos estes parâmetros devem ser passados para o construtor da classe `MobileObject` através do símbolo `super`.

(iii) Cada operação da interface `*Operations` deverá ser implementada pelo wrapper. Esta implementação deverá chamar a operação `getReference` herdada da classe `MobileObject` que devolverá a referência do objeto do usuário como o tipo `Object` do CORBA. Esta referência deverá passar por um typecasting utilizando-se a operação `narrow` da classe `*Helper` gerada pelo compilador `idlj`. O resultado da operação `narrow` deverá ser atribuído à interface do objeto do usuário também gerada pelo compilador `idlj`. O último passo da implementação de um a operação é chamar a mesma operação que está sendo implementada sobre a referência do objeto do usuário que foi obtida pela operação `narrow`.

Vale ressaltar que a operação `getReference` da classe `MobileObject` irá tomar todas as medidas necessárias para que a referência para o objeto do usuário devolvida por ela atenda as exigência do modo de operação da transação definida pela `AtomicAction` que o manipula. Através do uso de wrappers, todas as medidas necessárias para garantia de um modo de operação se tornam transparentes aos usuários de transações do SGTA.

## 4.8 Exemplo de Uso

Nesta seção será apresentado um exemplo simples e completo da utilização de uma transação do SGTA. O exemplo mostra uma transação chamada `SimpleAtomicAction` que executará na unidade móvel e que acessará operações sobre objetos do tipo Contador localizados na rede fixa. O primeiro passo na definição desta transação do SGTA é a implementação da sua política de adaptação. A seguir é mostrada uma implementação da política de adaptação para a transação `SimpleAtomicAction`:

```
class SimpleAtomicAction extends AtomicAction
{
   void notify (String resourceId, int resourceLevel){
      if (resourceId == "bandwidth"){
         if (resourceLevel < 10000 ){
            this.setOperationMode(LOCAL);
         }
      }
   }
}
```

Esta é uma política de adaptação que muda o modo de operação para local caso a largura da comunicação com a rede fixa seja menor que 10000bps. Apesar desta política de adaptação ser muito simples, podem existir políticas de adaptação mais elaboradas relacionadas a outros aspectos como monitoramento de energia, de custos ou de espaço em disco, políticas de adaptação que fazem novas requisições ao monitor de recursos de acordo com as necessidades de cada transação.

Uma vez que foi definida a política de adaptação, a transação `SimpleAtomicAction` poderá ser usada para controlar um conjunto de operações sobre objetos do usuário na rede fixa. Uma transação no SGTA poderá acessar vários objetos do usuário de diferentes tipos e localizados em diferentes máquinas da rede fixa, porém, para que o exemplo não seja muito longo, vamos considerar que a transação executa um conjunto de operações sobre um único objeto do usuário do tipo contador que possui a sua matriz remota localizada na máquina `Aracaju` da rede fixa. Este objeto será acessado através do wrapper `MobileCounter` que foi definido nas seções anteriores. Segue o exemplo:

```
public class TrnA{
   public static void main(String args[]){
      /* INSTANCIANDO A TRANSAÇÃO */
      SimpleAtomicAction trn = new SimpleAtomicAction();
      /* DEFININDO O OBJETO DO USUÁRIO QUE SERÁ ACESSADO */
      MobileCounter c1 = new MobileCounter("c1", "Counter",
                                           "MyLaptop", "2121",
                                           "Aracaju", "6688", trn);
      /* DESCREVENDO O RECURSO A SER MONITORADO PELO MR */
      ResourceDescriptor dscr = new ResourceDescriptor("bandwidth",
                                                       10000,
                                                       -1);
      try{
         trn.Begin();
            ResourceMonitor.request(trn, dscr);
            c1.inc();
            c1.get();
            ...
         trn.End();
      }catch (LockRefusedException e) {
         trn.Abort();
      }
   }
}
```

Este exemplo mostra que as operações da transação são delimitadas pelo `Begin` e o `End`. Antes de executar as operações sobre o objeto `c1`, a transação faz uma chamada ao monitor de recursos requisitando o monitoramento de largura de banda. Nesta requisição, a transação pede para ser notificada quando a disponibilidade da largura de banda seja menor que 10000bps. Uma vez realizada esta requisição, o monitor de recursos chamará a

operação `notify` da transação caso a largura de banda de comunicação com a rede fixa se torne menor que 10000bps.

Pode-se observar que a transação `trn` não especifica como parâmetros da operação `Begin` o nível de isolamento e o modo de operação que ela irá executar. Sendo assim, `trn` executará sob o nível de isolamento e modo de operação padrões que é o nível de serialização e o modo remoto.

O exemplo também mostra que o escopo da transação possui um tratamento de exceções do tipo `LockRefusedException`. Esta é uma exceção que poderá ser emitida pelas operações do objeto do usuário caso o `LockManager` que controla o objeto rejeite o pedido de acesso feito de maneira implícita pela transação. Neste exemplo, quando um pedido de acesso a uma operação é rejeitado, a transação é abortada através da chamada da operação `Abort`.

## 4.9 Principais Aspectos do JavaIDL no Protótipo

Na implementação do protótipo foram utilizados vários recursos do JavaIDL providos pela plataforma J2SE. A seguir serão apresentados os principais recursos que foram utilizados:

**Servidores Persistentes.** Permite que servidores de objetos em inatividade tenham os seus estados armazenados em memória persistente de forma que possam ser automaticamente reativados para atender as requisições de aplicações clientes. Este recurso foi utilizado no protótipo nos servidores remotos de objetos do usuário.

**Interceptadores Portáteis.** Permite a interceptação do fluxo normal da execução de um ORB permitindo o envio e o resgate de informações extras. Este recurso foi utilizado no protótipo da seguinte maneira: quando uma transação chama uma operação remota sobre um objeto da rede fixa, o interceptador detecta a chamada e envia para o gerenciador de trancas do mesmo objeto a informação que identifica a transação que está realizando a chamada. A partir desta informação, passada transparentemente com os dados da chamada remota, o gerenciador de trancas realizará o controle de concorrência sobre o mesmo objeto.

**Ativador de Servants.** Permite que cada objeto do usuário remoto seja ativado sob demanda, ou seja, um objeto remoto do usuário localizado na rede fixa só seria ativado logo depois que alguma aplicação requisitar uma de suas operações. Desta forma, o ativador de servants provê uma melhor utilização da memória das máquinas servidoras de objetos da rede fixa.

**Callbacks.** Os mecanismos de callbacks providos pelo JavaIDL permitem que uma aplicação tipo cliente receba uma notificação de um servidor. Esta notificação consiste na chamada de uma operação do cliente realizada pelo servidor. Callbacks são utilizados no protótipo na interação entre uma transação e o monitor de

recursos, onde a transação é cliente e o monitor de recursos é o servidor que fornece serviços a este cliente. Quando a transação envia uma requisição ao monitor de recursos, este poderá retornar um callback à transação para notificar a mudança em um determinado recurso.

## 4.10 Linhas de Código do Protótipo

O número de linhas de código e o número de classes do protótipo implementado estão descritos na tabela abaixo:

| | **Escritas Manualmente** | **Geradas Automaticamente** | **Total** |
|---|---|---|---|
| **N° de linhas com comentários** | 7712 | 5920 | 13632 |
| **N° de linhas sem comentários** | 5875 | 4055 | 9930 |
| **Número de classes** | 41 | 53 | 94 |

## 4.11 Resultados

Para efetuar os testes do protótipo do SGTA implementado, foi montado um cenário de rede composto por uma unidade móvel e por uma rede fixa. A rede fixa era uma rede local de 10 Mb composta por estações de trabalho com o sistema operacional Windows NT e a unidade móvel era um laptop também com o sistema operacional Windows NT. Para que o código Java do protótipo pudesse ser executado, foi instalado no laptop e em algumas estações de trabalho da rede fixa o interpretador `java` da plataforma J2SE 1.4.

Em uma estação de trabalho da rede fixa chamada `cantionila` foi instalado o repositório de objetos. Neste repositório foram armazenados alguns objetos do usuário que serviriam como matrizes remotas. Já no laptop, foram instalados os componentes do sistema que dará apoio ao processo de adaptação de transações: o gerenciador de cache, o monitor de recursos, o repositório de objetos (local) e a fila de efetivações.

O repositório de objetos instalado na estação de trabalho `cantionila` é um servidor CORBA que provê um conjunto de operações que podem ser acessadas remotamente por outros componentes do protótipo. Cada um dos componentes instalados na unidade móvel (laptop) também é um servidor CORBA que provêem serviços a uma ou mais transações que estejam executando na mesma unidade móvel.

### 4.11.1 Casos de Teste

Tendo o cenário de rede montado e configurado com os seus devidos componentes, foi selecionado um conjunto restrito de casos de testes que consistiam basicamente em confrontar transações que executavam na unidade móvel com obstáculos da computação móvel como, variações na largura de banda, desconexões freqüentes, escassez de energia, escassez de espaço em disco e alto custo da comunicação sem fio. Como não dispúnhamos de uma interface de comunicação sem fio entre a unidade móvel e a rede fixa, todos os testes de adaptação relacionados com a comunicação sem fio foram tratados através de simulação. A seguir são mostradas algumas estratégias que foram utilizadas nas simulações:

**Desconexão.** Foi simulado através da desativação da placa de rede do laptop que representou a unidade móvel. Esta desativação foi feita em tempo de execução através do software controlador da placa de rede do laptop.

**Reconexão.** Foi simulado através da reativação da placa de rede do laptop. A reativação também foi feita em tempo de execução pelo software controlador da placa de rede do laptop.

**Variações nos recursos do ambiente.** Simulado pelo Monitor de Recursos através de geração de valores aleatórios que representavam cada recurso monitorado.

Para iniciar o conjunto de testes foi feita uma cópia dos objetos do usuário mantidos na máquina `cantionila` para a cache do laptop. Esta cópia é necessária para que as transações que executam no laptop possam manter a sua execução mesmo quando desconectadas da rede fixa. A seguir será descrito de forma sucinta cada grupo de teste que foi aplicado ao protótipo do SGTA implementado:

**Grupo A: Testes com Desconexão.** Os testes com desconexão tiveram o objetivo de validar o comportamento de transações que executavam na unidade móvel quando confrontadas com desconexões repentinas. Foram utilizados basicamente três casos de teste com desconexão:

**Caso a.1:** Com transações que executavam sob o modo de operação local.

**Resultados:** Com os testes de desconexão observou-se basicamente que as transações que executavam sob o modo de operação local não sofreram nenhum efeito imediato da desconexão repentina. Isto se deu ao fato de que uma transação que opera sob o modo local acessa somente cópias locais sem a necessidade imediata da comunicação com a rede fixa. No final da execução destas transações, caso a conexão não tenha sido restabelecida, os seus processos de efetivação são delegados à fila de efetivação.

**Caso a.2:** Com transações que executavam sob o modo de operação remoto.

**Resultados:** As transações que executavam sob o modo de operação remoto foram interrompidas abruptamente com a desconexão repentina. Essa interrupção abrupta exigiu que estas transações fossem abortadas. Foi observada também a possibilidade de reiniciar a execução destas transações abortadas sobre o modo de operação local. Desta forma, as transações poderiam acessar as cópias dos objetos remotos previamente mantidas em cache.

**Caso a.3:** Com transações que executavam sob o modo de operação local-remoto.

**Resultados:** As transações que executavam sob o modo de local-remoto, apesar de perderem o contato com a rede fixa devido à desconexão, tiveram a opção de continuar a executar normalmente contanto que mudassem o seu modo de operação para o modo local. A única conseqüência de uma transação que muda o seu modo de operação para o modo local é que esta terá que passar por um processo de validação no final de sua execução.

**Grupo B: Testes com Reconexão.** Os testes com reconexão tiveram o objetivo de verificar se as transações que executam em uma unidade móvel desconectada eram capazes de concluir as suas atividades depois da reconexão. Basicamente foram aplicados os seguintes casos de teste:

**Caso b.1:** Verificar se transações que executam na unidade móvel desconectada sob o modo de operação local estão aptas a efetuar os seus protocolos de efetivação depois de uma reconexão com a rede fixa.

**Resultados:** Foi verificado que transações que executaram na unidade móvel desconectada sob o modo de operação local efetuaram o processo de validação e atualização dos objetos da rede fixa com sucesso.

**Caso b.2:** Verificar se a fila de efetivações consegue detectar a reconexão e assim executar cada processo de efetivação de transações que foi delegado a ela.

**Resultados:** A fila de efetivações detectou a reconexão e executou corretamente cada protocolo de efetivação de transações na fila.

**Grupo C: Testes com variação da disponibilidade de recursos.** Estes testes tiveram o objetivo de verificar a capacidade de reação de transações que são confrontadas com as variações na disponibilidade de recursos do ambiente. Basicamente, esses foram os casos de teste utilizados:

**Caso c.1:** Variação na largura de banda da comunicação com a rede fixa.

> **Resultados:** Foram verificados dois pontos: (i) transações que acessam objetos remotos da rede fixa podem utilizar o modo de operação local ou local-remoto para se livrar dos atrasos gerados quando a largura de banda da comunicação com a rede fixa é baixa; (ii) quando a largura de banda é alta, uma transação poderá obter uma maior garantia de consistência acessando diretamente os objetos matrizes através do modo de operação remoto.

**Caso c.2:** Variação na energia da unidade móvel.

> **Resultados:** Levando-se em consideração que a comunicação sem fio é um dos elementos que mais consomem energia do dispositivo de computação móvel, uma estratégia utilizada para reduzir este consumo é a utilização do modo de operação local. Com o modo de operação local ou local-remoto, pode-se reduzir a comunicação com a rede fixa durante toda a execução da transação. A comunicação com a rede fixa só seria necessária no final da execução da mesma transação, ou seja, para realizar o processo de validação da transação e atualização das cópias remotas se for o caso.

**Caso c.3:** Variação no custo da comunicação sem fio.

> **Resultados:** A depender do meio utilizado para a comunicação sem fio entre a unidade móvel e a rede fixa, esta comunicação pode se tornar bastante cara (com por exemplo a comunicação via celular). Uma estratégia de adaptação que pode ser utilizada para reduzir os custos desta comunicação é a utilização do modo de operação local. Com o modo de operação local a unidade móvel poderia ser desconectada da rede fixa com o intuito de reduzir os custos da comunicação sem fio. Neste caso, a comunicação só seria necessária no final da execução da transação para efetuar os processos de validação e atualização das cópias remotas.

**Caso c.4:** Variação no espaço em disco disponível na unidade móvel.

> **Resultados:** Os mecanismos que foram observados para que uma transação possa se adaptar à insuficiência de espaço em disco são: (i) enviar uma notificação ao usuário pedindo que este libere espaço em disco para dar continuidade a execução; (ii) a transação poderá também excluir objetos da cache que não estão sendo utilizados; (iii) a transação poderá utilizar o modo de operação remoto como uma forma de reduzir a necessidade de uso do disco da unidade móvel.

**Grupo D : Testes com níveis de isolamento.** Estes testes possuem o objetivo de verificar a flexibilidade que os níveis de isolamento podem prover a uma transação. Este grupo é composto por quatro casos de teste que mostram informalmente um paralelo entre o tempo

de resposta e os fenômenos de inconsistência (descritos na Seção 3.3.1) de cada nível de isolamento. Seguem os casos:

**Caso d.1:** Transações que utilizam o nível de isolamento 0.

**Resultados:** Foi observado que a utilização do nível de isolamento 0, como nível que menos retém trancas sobre os objetos acessados, provê um maior grau de concorrência às transações que o utilizam. Como transações sob o nível de isolamento 0 podem efetuar operações de leitura sobre objetos sem nenhuma restrição, ou seja, sem a necessidade de atribuição de trancas de leitura, o tempo de resposta para operações de leitura se torna o mínimo possível e pode ser utilizado como um elemento compensador dos atrasos inerentes à comunicação sem fio. Em relação às operações de escrita, este nível de isolamento impõe uma pequena restrição: tranca de escrita de curta duração. O nível de isolamento 0, apesar de poder prover um dos mais alto tempos de respostas para a transação que o utiliza, é o nível que deixa a transação mais vulnerável aos fenômenos de inconsistência.

**Caso d.2:** Transações que utilizam o nível de isolamento 1.

**Resultados:** Foi observado que o nível de isolamento 1, assim como o nível 0, também provê um tempo resposta mínimo para operações de leitura de transações. Este tempo mínimo é conseqüência da falta de restrições impostas às operações de leitura sobre objetos. Já em relação às operações de escrita, este nível é mais rígido quando comparado ao nível 0, ou seja, o nível 1 exige trancas longas de escrita reduzindo assim a vulnerabilidade de transações aos fenômenos de inconsistência. A utilização do nível de isolamento 1 também pode ser um fator compensador dos atrasos impostos pelo ambiente de comunicação sem fio. É evidente que uma transação só poderá utilizar este fator compensador se ela suportar os fenômenos de inconsistência aos quais estará sujeita ao utilizar este nível.

**Caso d.3:** Transações que utilizam o nível de isolamento 2.

**Resultados**: O nível de isolamento 2, por atribuir trancas curtas para operações de leitura, é mais restritivo que o nível 1. Foi observado que esta nova restrição aumentou consideravelmente o tempo de resposta das operações de leitura de transações que executam sob o nível 2. Apesar deste ser um nível mais restritivo que os anteriores, ele evita ainda mais o contato de transações com os fenômenos de inconsistência.

**Caso d.3:** Utilização do nível de isolamento 3.

**Resultados:** Foi observado que as transações que executam sob o nível de isolamento 3 são as que mais sofrem restrições de acesso a objetos, ou seja, tanto para operações de leitura quanto de escrita se faz necessária a aquisição de trancas de longa duração. Devido a estas restrições, as operações destas transações são as que possuem maior tempo de resposta. Apesar disso, a vantagem da utilização do nível de isolamento 3 é que a respectiva transação executará de forma serializável, evitando assim o contato com fenômenos de inconsistência. Portanto, este nível torna-se fortemente recomendado para transações que exigem a manutenção de regras de consistência mais rígidas.

### 4.11.2 Avaliação do CORBA

O CORBA, em implementações convencionais como JavaIDL e OrbixWeb, apesar de ser uma tecnologia de middleware que facilita o desenvolvimento de sistemas distribuídos escondendo detalhes complexos da comunicação na rede e provendo serviços de invocação remota de métodos, ele exige uma memória relativamente grande. Um dos motivos desta exigência é que nestas implementações convencionais é provido um conjunto grande de funcionalidades a fim de atender as exigências de aplicações dos mais variados escopos. Entretanto, na maioria dos casos, as aplicações só usam um pequeno subconjunto destas funcionalidades.

Em particular, nos testes realizados com o protótipo não tivemos nenhum problema quanto ao espaço de memória exigido pelo JavaIDL, isto porque os testes foram realizados em uma unidade móvel que consistia de um laptop com 120Mb de memória RAM e 20Gb de espaço em disco e um processador Intel Celeron 700MHz. Porém, quando se considera a capacidade de memória de unidades móveis bem menores como alguns palmtops e PDA's, torna-se inviável a utilização de implementações CORBA convencionais devido à alta quantidade de memória requerida.

Uma das soluções adotadas em relação às limitações dos pequenos dispositivos de computação é o uso de tecnologias de middleware que provêem algum tipo de suporte à personalização de componentes. Desta forma, o programador da aplicação poderia reduzir bastante o tamanho requerido pelo middleware ao selecionar apenas os componentes do middleware que fossem úteis na implementação do negócio. Um exemplo de middleware que provê personalização de componentes é o OpenORB [39] – resultado de estudos com middlewares reflexivos da Universidade de Lancaster-UK.

Uma outra solução que pode ser adotada é a utilização do minimumCORBA [40] que é uma implementação CORBA com um conjunto restrito de funcionalidades de forma a reduzir os seus requerimentos de memória e se tornar assim mais adequado a dispositivos de computação menores e com recursos limitados.

### 4.11.3 Requisitos Identificados com a Implementação do Protótipo

A análise, projeto e a implementação de um sistema de transações para o ambiente de computação normalmente devem iniciar a partir de um conjunto de requisitos. Quanto mais bem definidos são estes requisitos, mais sucesso poderá ser obtido em alcançar um produto que atende a objetivos bem determinados. Foi com o objetivo de ajudar a descobrir estes requisitos que se optou pelo desenvolvimento de um protótipo. A partir do processo de desenvolvimento e testes do protótipo requisitos não identificados foram notados. A seguir serão descritos alguns dos principais requisitos necessários em um sistema de transações para o ambiente móvel que foram identificados a partir da implementação do protótipo. Devido às restrições de tempo, somente alguns dos requisitos identificados foram implementados no próprio protótipo. Seguem os requisitos:

- **Mecanismos de caching.** Aplicações que utilizam acesso a dados remotos normalmente são altamente dependentes do processo de comunicação remota. Como este processo é instável e não confiável, faz-se necessário um mecanismo de caching que aumente a disponibilidade dos dados necessários para que as aplicações possam ter a sua dependência em relação ao meio de comunicação reduzida. No protótipo o mecanismo de caching é implementado pelo Gerenciador de Cache.
- **Mecanismos que aproveitem os eventuais estados de abundância de recursos.** Embora o ambiente de comunicação móvel seja caracterizado pela escassez de recursos, claramente podem existir momentos em que determinados recursos se tornem abundantes. Por exemplo, consideremos uma unidade móvel de um usuário que possui uma interface de comunicação sem fio e outra com fio. Enquanto o usuário está fora do seu escritório, a sua unidade móvel utiliza a interface sem fio caracterizada por baixa largura de banda. Porém, quando o usuário chega ao escritório, ele utiliza a interface de comunicação com fio caracterizada por alta largura de banda. Tendo em vista estes momentos de abundância de recursos, o sistema de transações deve possuir mecanismos para aproveitar estes momentos para realizar tarefas que requerem um alto grau de disponibilidade de recursos, como, por exemplo, transferir para a cache da unidade móvel grande quantidade de dados. No protótipo, a alta largura de banda pode ser aproveitada para a transmissão da fila de efetivação.
- **Maior autonomia da unidade móvel.** Um computador desktop integrante de uma rede local normalmente é altamente depende do acesso de dados e serviços fornecidos pelo restante da rede. Se uma unidade móvel integrante da mesma rede também tivesse esta alta relação de dependência com o restante da rede, suas atividades computacionais seriam freqüentemente inviabilizadas pelos constantes problemas impostos pela escassez de recursos. Portanto, faz-se necessário que o dispositivo de computação móvel possua uma certa autonomia em relação aos demais recursos da rede. Um exemplo da busca por essa autonomia implementado no protótipo foi o processo de caching e utilização do modo de operação local. Além disso, a efetivação de transações de forma assíncrona através da fila de efetivações também aumenta a autonomia.

- **Estratégias de adaptação.** Devido ao dinamismo do ambiente de computação móvel, faz-se necessário que o sistema de transações também proveja estratégias de adaptação que venham a acompanhar este dinamismo. No protótipo implementado, estas estratégias foram concebidas através dos modos de operação e níveis de isolamento de modo que as transações podem utilizar para se adaptar ao dinamismo do meio.
- **Ter ciência do que está ocorrendo no ambiente.** Claramente, para que o sistema de transações possa tomar qualquer medida de adaptação ao meio ele deve antes estar ciente de tais mudanças. Esta ciência foi implementada no protótipo através do Monitor de Recursos.
- **Políticas de adaptação independentes.** Quando se consideram transações que possuem interesses distintos executando em um mesmo dispositivo de computação móvel, deve-se levar em conta que uma decisão de adaptação adequada para uma transação pode não ser adequada para uma outra transação. Daí vem a necessidade de que transações com interesses distintos devem possuir políticas de adaptação independentes que venham em busca dos seus interesses individuais. No protótipo implementado, cada transação pode ter a sua própria política de adaptação através da implementação da operação abstrata `notify`.
- **Participação do usuário no processo de adaptação.** Com a implementação do protótipo foi observado que a participação do usuário no processo de adaptação pode ser muito importante para uma adaptação mais previsível, consistente e com menos impactos às aplicações. Essa importância vem do fato de que o usuário pode possuir o controle de vários aspectos relacionados à disponibilidade de recursos do ambiente. Por exemplo, é o usuário que, ao precisar sair do seu escritório, muda a interface de comunicação com fio para sem fio causando grandes reduções na largura de banda disponível. Levando-se em conta este fato, seria fortemente recomendado que o sistema de transações permitisse que o usuário o informasse com antecedência acerca de futuras mudanças na disponibilidade de recursos. Com isso as transações teriam tempo suficiente para tomar medidas de adequação às mudanças. Devido às restrições de tempo, não foram implementados mecanismos deste tipo no protótipo.
- **Mecanismos de cooperação no processo de adaptação.** Existem aspectos relacionados ao ambiente móvel que dependem de um acordo feito entre as aplicações como um todo. Para exemplificar, consideremos uma unidade móvel que se comunica com a rede fixa via telefone celular. Agora vamos considerar que o usuário desta unidade móvel programou uma aplicação para operar localmente em modo desconectado de forma a reduzir os custos impostos pela comunicação via celular. Esta estratégia de adaptação adotada pela mesma aplicação não teria nenhum sentido se não fosse realizada em conjunto com todas as outras aplicações que utilizam o meio de comunicação de maneira que todas elas adotassem a medida de operar em modo desconectado. Mecanismos como este não foram implementados no protótipo.
- **Mecanismos de segurança.** A interface de comunicação sem fio possui outras vulnerabilidades em relação à comunicação com fio como, por exemplo, o sinal de transmissão que pode ser interceptado por um dispositivo de captação intruso. Assim,

dentre os mecanismos de segurança que devem ser providos pelo sistema de transações para o ambiente móvel estão os mecanismos de segurança das informações que trafegam através da rede sem fio. Por restrições de tempo não foram implementados mecanismos de segurança no protótipo.

- **Tamanho do sistema.** O tamanho do sistema de transações também é um aspecto que deve ser considerado quando se utilizam dispositivos de computação pequenos como PDA's e palmtops. O projeto de um sistema de transações deve ser implementado visando este aspecto de maneira que possa ser adequado às capacidades destes dispositivos.
- **Mecanismos de tolerância a falhas.** Um sistema de transação visa a garantia de um conjunto de propriedades no que diz respeito à execução de transações. Como o ambiente de computação móvel é instável e sujeito a falhas repentinas, o sistema de transações deve possuir mecanismos de tolerância a falhas para que as mesmas não corrompam as propriedades a serem garantidas. Os mecanismos de tolerância a falhas implementados no protótipo são os que dizem respeito à garantia da propriedade de durabilidade e também tolerância a desconexões repentinas durante a execução de transações.

# Capítulo 5

# Trabalhos Relacionados

## 5.1 Coda

O Coda [52] é um dos trabalhos pioneiros em sua relevância à computação móvel. Apesar de não ter sido este um dos seus objetivos iniciais, atualmente o Coda é um sistema que provê acesso a dados a partir de máquinas móveis. Para isto, esse sistema utiliza técnicas como operação desconectada e operação fracamente conectada.

Os primeiros enfoques do Coda foram voltados ao tratamento de falhas de partição que afetam os usuários de sistemas distribuídos. Para contornar estes problemas foram propostas basicamente duas técnicas: a *replicação de servidor* e a *operação desconectada*. A *replicação de servidor* é uma técnica fundamental para a garantia da disponibilidade de dados durante falhas que ocasionam partição da rede. O Coda utiliza um conjunto de servidores que contém réplicas dos dados úteis aos usuários da rede. Assim, quando ocorre a partição da rede, um usuário poderá acessar os dados de um outro servidor na mesma partição em que ele se encontra. A *operação desconectada* é uma técnica usada principalmente para garantir a disponibilidade em circunstâncias de falhas em que a replicação de servidor não resolve. Um exemplo disso seria a perda de comunicação com os servidores da rede. A idéia central da operação desconectada é manter na cache da máquina do usuário uma cópia dos dados úteis ao mesmo. Assim, com a desconexão, o usuário poderia continuar as suas operações sobre a cópia em cache.

Estudos posteriores deram ao Coda a capacidade de operar em ambientes com comunicação sem fio, caracterizados pela largura de banda baixa e intermitente. Para isto, foi utilizada uma técnica chamada *operação fracamente conectada* [32] que foi apresentada como um meio de aliviar as limitações da operação desconectada. A operação desconectada apresentou problemas como: a falta de dados em cache comprometendo o andamento das atividades do cliente; a não visibilidade das atualizações sobre dados em cache para outros clientes; conflitos de atualização dos dados nos servidores se tornando mais evidentes quanto maior é a duração da operação desconectada. Para superar estes problemas, a operação fracamente conectada trouxe modificações ao Coda. Primeiro, o

protocolo de manutenção dos dados em cache foi modificado para permitir uma rápida revalidação de grandes volumes de dados em cache depois de um período de desconexão. Depois, foi desenvolvido um mecanismo chamado *reintegração por partes*[17] que utiliza, de forma econômica, a largura de banda da rede na propagação da atualização dos dados em cache para os servidores.

Os modelos iniciais de replicação de dados no Coda só visavam a resolução de conflitos do tipo escrita/escrita. Os conflitos do tipo leitura/escrita eram ignorados visto que a possibilidade de sua ocorrência era tolerada pelos usuários de sistemas de arquivos compartilhados. Porém, com a operação desconectada a ocorrência destes conflitos aumentou de forma considerável. Para lidar com este problema, o Coda foi estendido com um mecanismo chamado *Isolation-Only Transaction* (IOT) [29]. Este mecanismo permite ao sistema admitir somente os conflitos leitura/escrita que satisfaçam certos critérios de serialização. Assim, o IOT busca dar uma melhor garantia de consistência a aplicações que executam no ambiente móvel.

No decorrer de anos de experiência com o Coda, observou-se que um mecanismo de cache automático totalmente transparente ao usuário não era suficiente para garantir a disponibilidade de dados na operação desconectada. A solução adotada foi o mecanismo chamado *caching translúcido*. Este mecanismo faz um balanço entre a total transparência e o total controle manual do processo de caching de dados. Com isso, o usuário passa a fazer parte do processo de seleção dos dados que serão carregados em cache enquanto que o sistema cuida dos detalhes de gerenciamento.

Dentre os conceitos utilizados para conceber o SGTA está o da operação desconectada que foi introduzido pelo Coda. O SGTA provê o serviço de operação desconectada através do modo de operação *local*. Com isto, uma transação pode continuar a executar mesmo estando desconectada da rede fixa. Porém, o SGTA, diferentemente do Coda, permite que a própria transação escolha, em tempo de execução, se vai utilizar e quando vai utilizar o modo de operação *local*.

Uma outra característica existente no SGTA é que este é capaz de prover a uma transação a garantia de cada propriedade ACID. Já o Coda, por não ser um sistema de transações, somente provê a suas aplicações a propriedade de *isolamento* (IOT).

## 5.2 Odyssey

O Odyssey [33, 34, 35, 36] é uma plataforma que permite acessar dados no ambiente de computação móvel e que faz uso do paradigma de adaptação colaborativa. Neste paradigma, o sistema e suas aplicações colaboram entre si para contornar os obstáculos do ambiente de computação móvel. Assim, o sistema assume a responsabilidade de gerenciar os recursos do ambiente móvel e de prover os mecanismos de adaptação, enquanto que as aplicações ficam livres para definir as suas políticas de adaptação.

---

[17] Do inglês trickle reintegration.

Na plataforma Odyssey, o gerenciamento dos recursos é feito por um componente chamado *Viceroy*. Este componente exporta uma operação chamada *request* que poderá ser usada por uma aplicação para requisitar o gerenciamento de recursos. Como parâmetros desta operação a aplicação informa: o recurso que deverá ser gerenciado e os limites superior e inferior do recurso que são toleráveis pela mesma aplicação. Uma vez feita a requisição, o Viceroy enviará uma notificação à aplicação correspondente ao notar que o recurso disponível está fora dos limites toleráveis pela mesma. Com este mecanismo de monitoramento e notificação, as aplicações passam a ter ciência das variações ocorridas na disponibilidade dos recursos do ambiente móvel podendo então reagir com estratégias de adaptação.

A estratégia de adaptação do Odyssey é ajustar a qualidade do dado acessado em relação à disponibilidade dos recursos necessários para acessá-lo. Por exemplo, uma aplicação de vídeo, ao notar uma escassez na largura de banda necessária para a transmissão dos quadros, poderá optar por reduzir a resolução dos quadros ou a taxa de atualização em que os quadros são acessados.

Para quantificar esta noção de qualidade, o Odyssey define o conceito de *fidelidade*. Para cada item de dado passam a existir uma cópia de referência de maior qualidade e outras cópias de qualidade inferior. A fidelidade então seria o grau de degradação do dado que é apresentado ao cliente em relação à cópia de referência. A fidelidade é uma noção dependente do tipo de dado que se está manipulando e possui também outras dimensões como, por exemplo, a consistência do dado. O nível de degradação do dado, em alguns casos, pode ser visto como o nível de garantia de consistência que é aplicado a ele.

Em suma, o processo de adaptação do Odyssey forma um ciclo: a aplicação especifica os limites de tolerância sobre um determinado recurso do ambiente; o sistema monitora o recurso e notifica a aplicação quando sua disponibilidade está fora dos limites toleráveis pela mesma; a aplicação reage à mudança informando novos níveis de fidelidade dos dados que ela acessa; e o processo volta ao início do ciclo.

Assim como o Odyssey, o SGTA também usa o paradigma da adaptação colaborativa (sistema e aplicações colaborando para sobrepor os obstáculos do meio). Com este paradigma, enquanto que o sistema de apoio gerencia os recursos compartilhados, monitora o ambiente e provê outros serviços de apoio a adaptação, cada aplicação fica livre para escolher a melhor forma de adaptação. Entretanto, o Odyssey, apesar de ter obtido bons resultados para aplicações que acessam dados multimídia, não provê suporte a transações.

## 5.3 ProMotion

O Pro-Motion [54] é um sistema de processamento de transações projetado para lidar com problemas introduzidos pela desconexão e pelos limitados recursos do ambiente móvel. Seu principal bloco de formação é o *compact* que funciona como uma unidade básica de replicação de dados.

Um compact é representado como um objeto que encapsula: dados, métodos para acesso aos dados, métodos para o gerenciamento do compact, informação a respeito do estado corrente do compact, regras de consistência e obrigações. O compact representa um acordo entre o cliente móvel que requisitou o dado e o servidor de banco de dados. Neste acordo, o servidor de banco de dados delega o controle de alguns dados à unidade móvel para ser usado no processamento de transações locais. O cliente móvel, em contrapartida, concorda em honrar com as condições específicas do uso dos dados de maneira que sua consistência seja mantida quando as atualizações forem propagadas de volta para o servidor de banco de dados. Cada compact é responsável pelo controle de concorrência e recuperação dos seus dados.

O Pro-Motion, implicitamente sugere quatro atividades de processamento de transações:

**Armazenamento**[18]**.** A unidade móvel está conectada à rede e os compacts estão sendo armazenados localmente na preparação para uma eventual desconexão.

**Processamento conectado.** A estação móvel está conectada à rede e as transações estão acessando dados diretamente do servidor.

**Processamento desconectado.** A estação móvel está desconectada da rede e as transações estão acessando dados locais contidos nos compacts.

**Ressincronização.** A estação móvel se reconectou à rede e as atualizações feitas durante a desconexão estão sendo reconciliadas com o banco de dados no servidor.

O Pro-Motion permite *efetivação local*[19] às suas transações. Caso uma transação opte pela efetivação local, seus resultados serão feitos visíveis para outras transações da unidade móvel, aceitando assim eventuais falhas ao efetuar a efetivação no servidor. Se a transação não iniciar com a opção LOCAL, ela só efetuará a efetivação local após a efetivação no servidor.

A consistência das operações sobre os dados no Pro-Motion pode ser caracterizado através de uma escala com dez níveis de isolamento baseados no padrão ANSI-SQL. O nível 9 garante uma total execução serial de transações e o nível 8 garante uma execução serializável. Cada nível inferior representa um menor nível de isolamento. O nível 0 não garante nenhum tipo de consistência.

Cada método em um compact pode ser associado a um nível de isolamento e a um modo que indica se a operação é de leitura ou de escrita. Assim, um compact poderá conter métodos que possuem níveis de consistência diferentes. Uma transação também poderá especificar o seu nível mínimo de isolamento para operações de leitura e o seu nível

---

[18] Em inglês: hoarding.

[19] O mesmo que local commit.

mínimo para operações de escrita de forma que nenhuma das operações chamadas sobre os dados de um compact seja menor que estes níveis especificados.

Diferentemente do ProMotion, o SGTA, considerando que os recursos do ambiente de computação móvel podem variar de modo imprevisível, provê a cada transação a capacidade de se adaptar a qualquer momento durante sua execução. Esta adaptação pode incluir mudança no modo de operação, mudança no nível de isolamento ou em outro mecanismo de adaptação.

## 5.4 Transações Fortes/Fracas

O estudo apresentado em [43, 44] propõe um esquema para a manutenção da consistência de dados replicados sujeito às restrições impostas pela natureza da computação móvel. Neste esquema, dados podem ser agrupados em diferentes *clusters.* Enquanto que os dados de um mesmo cluster são mantidos sobre total consistência, níveis de consistência mais fracos são mantidos entre dados de diferentes clusters. O nível de consistência poderá variar a depender da disponibilidade da largura de banda da comunicação entre os clusters. Clusters possuem configuração dinâmica, ou seja, um cluster pode ser dividido em vários outros, da mesma forma que vários clusters podem se unir na formação de um único cluster. Para que dois ou mais clusters possam formar um único cluster, todos os problemas de inconsistências entre dados de diferentes clusters devem ser resolvidos.

Uma das formas de se aumentar a disponibilidade de dados em um ambiente móvel sujeito a desconexões é a replicação local de dados. Dessa forma, as operações realizadas em uma unidade móvel podem continuar sobre a réplica dos dados em cache quando ocorre uma desconexão. [44] aponta que dados em cache localmente consistentes podem constituir um cluster. Da mesma forma, os dados globalmente consistentes podem constituir um outro cluster.

A partir destes conceitos, [44] introduziu o conceito de transações fracas e transações fortes. Uma transação fraca é aquela que interage com dados localmente consistentes, enquanto que uma transação forte é aquela que interage somente com dados globalmente consistentes. Enquanto uma transação forte só possui um ponto de efetivação – a *efetivação global*[20], as transações fracas possuem dois pontos de efetivação – a efetivação local no cluster associado e a efetivação global implícita. Esta efetivação global só é concluída quando o cluster localmente consistente relacionado se une ao cluster globalmente consistente formando uma unidade globalmente consistente.

Atualizações feitas por uma transação fraca que efetuou efetivação local só são reveladas a outras transações fracas no mesmo cluster. Estas atualizações só são reveladas a transações fortes depois da união do cluster localmente consistente correspondente com o cluster globalmente consistente, isto é, quando a transação fraca efetuar a efetivação global implícita.

---

[20] O mesmo que commit global.

## 5.5 Prayer

O Prayer [10, 15] é um sistema de arquivos projetado para o ambiente de computação móvel. Como no projeto Odyssey, o Prayer busca contornar os obstáculos deste ambiente através da estratégia de adaptação colaborativa. O modelo de adaptação do Prayer consiste de três camadas. A camada inferior é a de *recursos* como, largura de banda, espaço em disco, energia da bateria. A camada intermediária é a de *gerenciamento de recursos* e fornece apoio à adaptação à camada superior de *aplicação*. A idéia principal deste modelo é que a camada de gerenciamento de recursos proveja os mecanismos de gerenciamento, monitoramento e reação às mudanças na disponibilidade dos recursos, enquanto que a camada de aplicação proveja a política de adaptação à mudança.

No modelo de adaptação do Prayer, um programa é dividido em blocos de adaptação, onde cada bloco é vinculado a uma classe de QoS[21] negociada. Uma classe de QoS é definida pela especificação dos limites superior e inferior de um determinado recurso. Desta forma, um bloco adaptativo é executado enquanto a disponibilidade do recurso atende a classe de QoS associada. Se o ambiente móvel não estiver apto a atender a QoS negociada, a camada de gerenciamento de recursos notifica a aplicação.

Quando uma aplicação recebe uma notificação de não atendimento de uma classe de QoS negociada, ela poderá reagir das seguintes formas: *best effort* – ignora a notificação e continua a tarefa; *block* – suspende a tarefa até que a QoS requerida volte a ser atendida; *abort* – aborta o restante do bloco de adaptação correspondente e passa a executar o próximo bloco de adaptação; *rollback* – aborta o restante da tarefa e reinicia a renegociação de QoS para a mesmo bloco de adaptação.

O Prayer também provê um mecanismo que permite a manutenção da consistência de partes de um arquivo como uma forma de se adequar à conectividade fraca. Por exemplo, quando a largura de banda da conexão é baixa, a aplicação possui a flexibilidade de manter a consistência de determinados campos críticos ou registros enquanto que a consistência do restante do arquivo não é garantida.

Além da flexibilidade de manter a consistência de partes de um arquivo, o Prayer também provê consistência explícita de operações de leitura e escrita. Os tipos de leitura oferecidos são: *leitura local* – simplesmente lê a cópia local; *leitura consistente* – verifica a consistência entre a cópia local e a cópia do servidor remoto e recupera a cópia do servidor remoto caso a cópia local esteja inconsistente. São oferecidos três tipos de escrita: *escrita local* – atualiza somente a cópia local; *escrita atrasada* – escreve a atualização em arquivos de processamento em lotes; *escrita consistente* – atualiza o servidor.

O Prayer, assim com o SGTA, utiliza a adaptação colaborativa para aplicações de acesso a dados, porém não provê suporte a transações.

---

[21] Qualidade de Serviço

## 5.6 Outros Modelos Transacionais

Vários são os modelos transacionais voltados à computação móvel citados na literatura. Estudos, como em [3], [50] e [31], fazem uma síntese dos principais modelos existentes apontando as particularidades de cada um deles. Alguns destes modelos, como em [12] e [14], são adaptações de modelos transacionais aninhados abertos como uma forma de atender requisitos como a longa duração de transações. Estes modelos utilizam a idéia da divisão de uma transação em subtransações que podem executar uma parte na rede fixa e outra parte na máquina móvel como forma de reagir a problemas do ambiente móvel como as desconexões.

[12] apresenta dois novos tipos de subtransações chamadas *reporting* e *co-transactions* para facilitar a colaboração entre a máquina móvel e a estação base.

O modelo Kangaroo [14] captura a idéia de movimento criando novas subtransações à medida que a máquina portátil muda de uma célula para outra.

Estes modelos, apesar de serem voltados à computação móvel, não abordam problemas básicos da mesma como as desconexões inesperadas e a baixa largura de banda da comunicação sem fio.

# Capítulo 6

# Conclusões

Esta dissertação apresentou um sistema de transações que provê mecanismos de adaptação que buscam viabilizar a execução de transações no ambiente de comunicação sem fio sujeitas à instabilidade da comunicação sem fio e escassez de recursos no dispositivo móvel. Estes mecanismos foram concebidos através do uso de um padrão de colaboração entre a transação e um sistema de base onde este fornece serviços de monitoramento do ambiente computacional e mecanismos de adaptação enquanto que a transação determina a sua política de adaptação.

Um mecanismo de adaptação que pode ser utilizado pelas transações para enfrentar as adversidades do ambiente é o uso dos modos de operação. Os modos de operação são três: local, remoto e local-remoto. Através do modo local, as operações da transação sobre objetos da rede fixa são redirecionadas para as cópias dos objetos armazenados na cache da unidade móvel. Com o uso deste modo de operação uma transação pode evitar todo o tipo de comunicação com a rede fixa durante a sua execução. O modo de operação local-remoto, assim como o modo local, redireciona as operações da transação para as cópias dos objetos armazenados na cache da unidade móvel, porém os respectivos objetos da rede fixa são bloqueados com trancas que visam garantir a consistência das cópias em cache em relação as suas correspondentes remotas. Já com o modo de operação remoto, uma transação acessa os objetos da rede fixa diretamente através de invocações remotas.

Outro mecanismo de adaptação provido pelo sistema é o uso de níveis de isolamento. Os níveis de isolamento, além de flexibilizar o controle de concorrência para atender as necessidades de consistência de cada transação em individual, eles provêm uma maneira clara de aumentar o desempenho de transações que executam no ambiente de comunicação sem fio. Embora este ganho no desempenho seja em detrimento da garantia das regras de consistência, existem grupos de transações que podem executar sob níveis de consistência mais baixos.

Além dos mecanismos implementados como modo de operação e nível de isolamento, o usuário do sistema de transações proposto poderá implementar outros mecanismos que atendam às necessidades específicas de sua aplicação em particular.

É através do uso deste padrão de colaboração entre sistema de apoio e transações que é dada a uma transação uma maior disponibilidade dos dados distribuídos acessados, uma

maior capacidade de adaptação e uma maior capacidade de flexibilização das propriedades ACID diante das turbulências do ambiente de comunicação sem fio.

## 6.1 Contribuições

- Foi proposto um modelo de transações que utiliza o padrão colaborativo de adaptação entre as transações e o sistema de apoio como meio eficaz de superar obstáculos impostos pelo ambiente de computação móvel. Neste padrão, cada transação individual pode definir a sua política de adaptação enquanto que o sistema de apoio provê mecanismos de adaptação, monitoramento de recursos e outros serviços de apoio à transação.
- Foram propostos os modos de operação e mecanismos que permitem que uma transação na unidade móvel execute mesmo quando desconectada ou fracamente conectada com o restante da rede.
- Implementação de um protótipo que pode ser usado como ferramenta para o levantamento de outros requisitos de sistemas de transações em ambientes de computação móvel.

## 6.2 Trabalhos Futuros

- Efetuar um conjunto de testes de desempenho que venham avaliar os impactos causados na mudança de modo de operação de uma transação, ou seja, na transferência de estados de objetos entre máquinas da rede fixa e a unidade móvel e vice-versa.
- Apesar de tecnologias de middleware como o CORBA, J2EE e .NET facilitarem o desenvolvimento de aplicações distribuídas, elas não provêm apoio apropriado para atender os requisitos dinâmicos das aplicações que executam em um ambiente de computação móvel. Porém, existem estudos que propõem middlewares como o OpenORB e dynamicTAO [39] que visam justamente atender a estes requisitos dinâmicos através do uso de técnicas de reflexão computacional. Um dos trabalhos futuros seria então investigar técnicas de desenvolvimento de transações para o ambiente móvel que utilizam serviços de middlewares como o OpenORB e o dynamicTAO.
- Fazer um levantamento dos principais grupos de dispositivos computacionais móveis como palmtops, PDA's e laptops de acordo com as suas capacidades de memória, poder de processamento e de transmissão de dados de forma a levantar os principais requisitos de hardware que sistemas de transação têm que se adequar.

# Referências Bibliográficas


[1] Adya, A. "Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions", Massachusetts Institute of Technology, PhD Thesis, March, 1999.

[2] Adya, A., Liskov, B. and O'Neil, P. "Generalized Isolation Level Definitions", In Proceedings of the IEEE International Conference on Data Engineering, San Diego, CA, 2000.

[3] Alvarado, P. S. and Roncancio, M. A. "Mobile Transaction Supports for DBMS: An Overview", Raport de Recherche, RR 1039-1-LSR 16, Laboratorie LSR-IMAG, Grenoble, 2001.

[4] ANSI X3.135-1992 "American National Standard for Information Systems – Database Languages – SQL", November, 1992.

[5] Astrahan, M. M. et al. "System R: Relational Approach to Database Management", ACM Transactions on Database Systems, pages 97-137, June, 1976.

[6] Baggio, A. "Design and Early Implementation of the Cadmium Mobile and Disconnectable Middleware Support", Rapport de Recherche, nº 3515, INRIA, October, 1998.

[7] Basu, J. and Keller, A. M. "Degrees of Transaction Isolation in SQL*Cache: A Predicate-based Client-side Caching System", Technical report, Stanford University, Computer Science Department, Palo Alto, CA, 1996.

[8] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. and O'Neil, P. "A Critique of ANSI SQL Isolation Levels", In Proceedings of ACM SIGMOD International Conference on Management of Data, San Jose, CA, May, 1995.

[9] Bernstein, P. A., Hadzilacos, V. and Goodman, N. "Concurrency Control and Recovery in Database Systems", Addison Wesley, 1987.

[10] Bharghavan, V. and Gupta, V. "A Framework for Application Adaptation in Mobile Computing Environments", In Proceedings of IEEE Compsoc'97, November, 1997.

[11] Borland. http://www.borland.com, 2003.

[12] Chrysanthis, P. K. “Transaction Processing in a Mobile Computing Environment”, Workshop on Advances in Parallel and Distributed Systems, pages 77-82, 1993.

[13] Dunham, M. H. and Helal, A. “Mobile Computing and Databases: Anything New?”, ACM SIGMOD Record, Vol. 24, No. 4, December, 1995.

[14] Dunham, M. H. and Helal, A. “A Mobile Transaction Model that Captures Both the Data and the Movement Behavior”, ACM-Baltzer Journal on Special Topics in Mobile Networks and Applications, Vol. 2, pages 149-162, 1997.

[15] Dwyer, D. and Bharghavan, V. “A Mobility-Aware File System for Partially Connected Operation”, In Operating Systems Review, Vol. 31, No. 1, 1997.

[16] Forman, G. H. and Zahorjan, J. “The Challenges of Mobile Computing”, IEEE Computer, Vol. 27, April, 1994.

[17] Franklin, M. J., Carey, M. J. and Livny, M. “Transactional Client-Server Cache Consistency: Alternatives and Performance”, Technical Report, CS-TR-3511, University of Maryland College Park, 1995.

[18] Gray, J., Lorie, R., Putzolu, G. and Traiger, I. “Granularity of Locks and Degrees of Consistency in a Shared Database”, Modeling in Database Management Systems, Amsterdam: Elsevier North-Holland, 1976

[19] Huston, L. B. and Honeyman, P. “Partially Connected Operation”, In Proceedings of the Second USENIX Symposium on Mobile and Location-Independent Computing, April, 1995.

[20] IONA Technologies. http://www.iona.com, 2003.

[21] Imielinski, T. and Badrinath, B. R. “Mobile Wireless Computing: Challenges in Data Management”, Communications of ACM, Vol. 37, No. 10, October, 1994.

[22] Java2 Platform, Standard Edition. http://java.sun.com/j2se/1.4.1/index.html

[23] Jing, J., Helal, A. and Elmagarmid, A. “Client-Server Computing in Mobile Environments”, ACM Computing Surveys (CSUR), Vol. 31, No. 2, June, 1999.

[24] Katz, R. H. “Adaptation and Mobility in Wireless Information Systems”, IEEE Personal Communications, Vol. 1, No. 1, 1995.

[25] Kayan, E. and Ulusoy, Ö. “An Evaluation of Real-Time Transaction Management Issues in Mobile Database Systems”, The Computer Journal (Special Issue on Mobile Computing), Vol. 42, No. 6, 1999.

[26] Kistler, J. J. and Satyanarayanan, M. “Disconnected Operation in Coda File System”, In ACM Symposium on Operating Systems and Principles, Vol. 10, No. 1, February, 1992.

[27] Orfali, R. and Harkey, D. “Client/Server Programming with Java and CORBA”, Wiley Computing Publishing, Second Edition, 1998.

[28] Vogel, A. and Duddy, K. "Java Programming with CORBA. Advanced Techniques for Building Distributed Applications", Second Edition, Wiley Computing Publishing, 1998.

[29] Lu, Q. and Satyanaranyanan, M. "Isolation-only transactions for mobile computing", In ACM Operating Systems Review, Vol. 28, No. 3, 1994.

[30] Madria, S. K. and Bhargava, B. "On the Correctness of a Transaction Model for Mobile Computing", In Proceedings of the International Conference and Workshop Database and Expert Systems Applications, pages 573-583, 1997.

[31] Madria, S. K. "Transaction Models for Mobile Computing", In Proceedings of 6th IEEE Singapore International Conference on Network, World Scientific, Singapore, July, 1998.

[32] Mummert, L. B., Ebling, M. R. and Satyanarayanan, M. "Exploring Weak Connectivity for Mobile File Access", In Proceedings of the 15th ACM Symposium on Operation Systems Principles, Colorado, December, 1995.

[33] Noble, B., Satyanarayanan, M., Narayanan, D., Tilton, J. E., Flinn, J. and Walker, K. R. "Agile Application-Aware Adaptation for Mobility", In Proceedings of the 16th ACM Symposium on Operating System Principles, France, October, 1997.

[34] Noble, B. "Mobile Data Access", Carnegie Mellon University, PhD Thesis, 1998.

[35] Noble, B. and Satyanarayanan, M. "Experience with Adaptive Mobile Applications in Odyssey", In Mobile Networks and Applications, Vol. 4, 1999.

[36] Noble, B. "System Support for Mobile Adaptive Applications", In IEEE Personal Communications, Vol. 7, No. 1, February, 2000.

[37] Common Object Request Broker Architecture. http://www.corba.org, 2003.

[38] Object Management Group. http://www.omg.org, 2003.

[39] Kon, F., Costa, F., Blair, G. and Campbell, R. H. "The Case for Reflective Middleware", Communications of ACM, Vol. 45, No. 6, June, 2002.

[40] CORBA™/IIOP™ Specification. http:// www.omg.org/technology/documents/formal/corba_iiop.htm, 2003.

[41] Oracle Corporation. "Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7", Oracle Write Paper, Part No. A33745, July, 1995.

[42] Parrington, G., Shrivastava, S. K., Stuart, M. W. and Little, M. C. "The Design and Implementation of Arjuna", USENIX Computing Systems Journal, Vol. 8, No. 3, 1995.

[43] Pitoura, E. and Bhargava, B. "Revising Transaction Concepts for Mobile Computing", In Proceedings of the IEEE Workshop on Mobile Systems and Applications, Santa Cruz, CA, 1994.

[44] Pitoura, E. and Bhargava, B. “Maintaining Consistency of Data in Mobile Distributed Environment”, In Proceedings of the 15th Int. Conference on Distributed Computer Systems, June, 1995.

[45] Pitoura, E. “Data Management for Mobile Computing”, Summer School on Mobile Computing, Jyvaskyla, 1998.

[46] Pu, C., Kaiser, G. E. and Hutchinson, N. “Split-Transactions for Open-Ended Activities”, In Proceedings of the 14th VLDB Conference, Los Angeles, California., 1988.

[47] Rocha, T. and Toledo, M. B. F. “Um Sistema de Transações Adaptável para o Ambiente de Computação Móvel”, In Proceedings of 21th Simpósio Brasileiro de Redes de Computadores, pages 731-746, Natal-RN, Brasil, 2003.

[48] Rocha, T. and Toledo, M. B. F. “A CORBA-based Transaction System for the Wireless Communication Environment”, In Proceedings of Confederated International Conferences: CoopIS, DOA, and ODBASE, Catania, Sicily, Italy, 2003.

[49] Rocha, T. and Toledo, M. B. F. “An Adaptable Transaction System for the Mobile Computing Environment”, In Proceedings of ACM/IFIP/USENIX International Middleware Conference, Poster Session, Rio de Janeiro-RJ, Brasil, 2003.

[50] Seydim, A. Y. “An Overview of Transaction Models in Mobile Environments”, Paper prepared for Distributed Operating Systems course, Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX, 1999.

[51] Satyanarayanan, M. “Fundamental Challenges in Mobile Computing”, Fifteenth ACM Symposium on Principles of Distributed Computing, Philadelphia, PA, May, 1996.

[52] Satyanarayanan, M. “The Evolution of Coda”, In ACM Transactions on Computer Systems, Vol. 20, No. 2, May, 2002.

[53] Sun Microsystems. http://www.sun.com, 2003.

[54] Walborn, G. D. and Chrysanthis, P. K. “Transaction Processing in PRO-MOTION”, In Proceedings of the 4th ACM Annual Symposium on Applied Computing, 1999.

[55] Pressman, R. S. “Engenharia de Software”, Makron Books, Terceira Edição, 1995.

[56] UML – Unified Modelling Language. http://www.uml.org/

# Apêndice A

# Interface das Principais Classes do SGTA

Este apêndice apresentará as interfaces das principais classes que implementam o SGTA. Algumas das interfaces apresentadas foram inspiradas no projeto do sistema de transações Arjuna [42].

## A.1. Buffer

```
public class Buffer
{
   /* CONSTRUTORES */
   public Buffer(String bufferId);
   public Buffer();

   /* OPERAÇÕES PARA EMPACOTAMENTO DE DADOS */
   public boolean packVersion(long data);
   public boolean packByte(byte data);
   public boolean packShort(short data);
   public boolean packInt(int data);
   public boolean packLong(long data);
   public boolean packBoolean(boolean data);
   public boolean packString(String data);
   public boolean packDouble(double data);
   public boolean packFloat(float data);

   /* OPERAÇÕES PARA DESEMPACOTAMENTO DE DADOS */
   public long unpackVersion()
   public byte unpackByte();
   public short unpackShort();
   public int unpackInt();
   public long unpackLong();
   public boolean unpackBoolean();
   public String unpackString();
   public double unpackDouble();
   public float unpackFloat();
}
```

## A.2. ObjectState

```
public class ObjectState extends Buffer{
   /* ATRIBUTOS */
   private String stateId;
   private String imageType;
   /* CONSTRUTORES */
   public ObjectState(String stateId, String objType);
   public ObjectState();
   /* OPERAÇÕES PÚBILCAS */
   public String getStateId();
   public String getType();
}
```

## A.3. ObjectStore

```
public interface ObjectStore{
   /* OPERAÇÕES */
   public boolean commit_state(String stateId, String objType);
   public boolean hide_state(String stateId, String objType);
   public boolean reveal_state(String stateId, String objType);
   public ObjectState read_committed(String stateId, String objType);
   public ObjectState read_uncommitted(String stateId, String objType);
   public boolean write_committed(ObjectState os);
   public boolean write_uncommitted(ObjectState os);
   public boolean remove_committed(String stateId, String objType);
   public boolean remove_uncommitted(String stateId, String objType);
   public boolean isPresent(String stateId, String objType);
   public String type();
}
```

## A.4. CacheManager

```
public class CacheManager
{
  /* CONSTRUTOR */
  public CacheManager();
  /* OPERAÇÕES PÚBLICAS */
  public boolean loadState (String objectId, String objectType,
                            String remoteServer, String remotePort);
  public boolean updateRemoteState(String objectId, String objectType,
                                   String remoteServer, String remotePort);
  public boolean isActive (String objectId, String objectType);
  public boolean activate (String objectId, String objectType);
  public boolean isPresent (String objectId, String objectType);
  public boolean isConsistent (String objectId, String objectType,
                 String remoteServer, String remotePort);
  public int markState (String objectId, String objectType);
  public int unmarkState (String objectId, String objectType);
  public boolean isMarked (String objectId, String objectType);
}
```

## A.5. AtomicAction

```
public abstract class AtomicAction{

   /* CONSTANTES */
   public static final short RUNNING = 0, PREPARING = 1, ABORTING = 2,
                             ABORTED = 3, PREPARED = 4, COMMITTING = 5,
                             COMMITTED = 6, CREATED = 7, INVALID = 8;
   private static final short ADDED = 0, REJECTED = 1;

   /* ATRIBUTOS PRIVADOS */
   private RecordList pendingList;
   private RecordList preparedList;
   private RecordList readonlyList;

   /* ATRIBUTOS PROTEGIDOS */
   protected int status;
   protected String id;
   protected int isolationLevel;
   protected int operationMode;

   /* CONSTRUTOR */
   public AtomicAction();

   /* OPERAÇÕES PÚBLICAS */
   public int setOperationMode(int operationMode);
   public int getOperationMode();

   public int add(AbstractRecord ar);
   public void addI(int index, AbstractRecord record);

   public AbstractRecord get(int index);
   public AbstractRecord remove(int index);
   public int size();
   public int status();
   public int Abort();
   public int Begin();
   public int Begin(int isolationLevel, int operationMode);
   public int Begin(int isolationLevel);
   public int setIsolationLevel(int isolationLevel);
   public int getIsolationLevel();
   public int End();

   /* OPERAÇÕES PROTEGIDAS */
   protected void phase2Commit();
   protected void phase2Abort();
   protected int prepare();

   /* OPERAÇÃO ABSTRATA */
   abstract void notify (String requestId, int resourceLevel);
}
```

## A.6. StateManager

```
public abstract class StateManager
{
   /* CONSTANTES QUE IDNICAM O ESTADO DE UM OBJETO */
   public static final short PASSIVE = 0, PASSIVE_NEW = 1,
                             ACTIVE = 2,  ACTIVE_NEW = 3;

   /* ATRIBUTOS */
   protected String objectId;
   protected int objectStatus;
   protected long version;

   /* CONSTRUTOR */
   public StateManager(String objectId);

   /* OPERAÇÕES PÚBLICAS */
   public boolean activate();
   public boolean deactivate(boolean commit);
   public long getVersion();
   public void setVersion(long version);
   public long incVersion();
   public int status();
   public void modified();
   public String getId();

   /* OPERAÇÕES ABSTRATAS */
   public abstract boolean restore_state(ObjectState objState);
   public abstract boolean save_state(ObjectState objState);
   public abstract String type();
}
```

## A.7. LockManager

```
public abstract class LockManager extends StateManager
{
   /* CONSTANTES */
   public static final short GRANTED = 0, REFUSED = 1, RELEASED = 2;
   public static final short CONFLICT = 0, COMPATIBLE = 1, PRESENT = 2;

   /* ATRIBUTOS PRIVADOS */
   private LockList lockList;
   private RecordList recordList;

   /* CONSTRUTOR */
   public LockManager(String objectId);

   /* OPERAÇÕES PÚBILCAS */
   public int setLock(Lock lock) throws LockRefusedException;
   public synchronized int releaseLock(String lockId);

   /* OPERAÇÃO PROTEGIDA */
   protected synchronized AtomicAction getCurrentAtomicAction();

   /* OPERAÇÕES ABSTRATAS HERDADAS DA CLASSE StateManager */
   public abstract boolean restore_state(ObjectState objState);
   public abstract boolean save_state(ObjectState objState);
   public abstract String type();
}
```

## A.8. Lock

```
public class Lock
{
   /* Constantes que representam os modos de lock. */
   public static short READ = 0, WRITE = 1;

   /* Identificador do lock. */
   protected String lockId;

   /* Modo do lock: READ, WRITE... */
   protected int lockMode;

   /* CONSTRUTOR; lockMode: READ, WRITE... */
   public Lock(int lockMode);

   /* Devolve o identificador do lock. */
   public String getId();

   /* Devolve o modo do lock: READ, WRITE... */
   public int lockMode();

   /* Verifica se o lock é conflitante com outro lock informado. */
   public boolean conflictsWith(Lock otherLock);

   /* Se o modo de lock leva em conta modificações no objeto ou nao. */
   public boolean modifiesObject();
}
```

## A.9. ResourceMonitor

```
public class ResourceMonitor{
   /* Constantes que indicam o estado do monitor */
   public static final int RUNNING = 0, STOPPED = 1, INACTIVE = 2;
   /* CONSTRUTOR */
   public ResourceMonitor();
   /* Operação de requisição de monitoramento do recurso */
   public String request(AtomicAction atomicAction,
                         ResourceDescriptor descriptor);
   /* Cancela o monitoramento de recurso antes requisitado. */
   public int cancel(String requestId);
   /* Adiciona um monitor que monitorará um recurso em específico. */
   public int addMonitor(MonitorInterface monitor);
   /* Devolve o monitor de recursos a partir do seu identificador. */
   public MonitorInterface getMonitor(String monitorId);
   /* Devolve o nível atual do recurso especificado. */
   public int getLevel(String monitorId);
   /* Coloca o monitor de recursos em estado de execução. */
   public int run();
   /* Desativa o monitor de recursos. */
   public int stop();
}
```

## A.10. MonitorInterface

```
interface MonitorInterface{
   /* Operação de requisição de monitoramento do recurso. */
   public long request(AtomicAction atomicAction,
                       int lowerBound,
                       int upperBound );
   /* Cancela o monitoramento de recurso. */
   public int cancel(long requestId);
   /* Devolve o identificador do monitor do recurso. */
   public String getId();
   /* Devolve o nível atual do recurso que está sendo monitorado. */
   public int getLevel();
   /* Coloca o monitor de recurso em execução. */
   public int _run();
   /* Desativa o monitoramento do recurso. */
   public int _stop();
}
```

## A.11. MobileObject

```
public class MobileObject{
  /**
   * Instancia um novo <code>MobileObject</code>.
   *
   * @param objectId Identificador do objeto.
   * @param objectType Tipo do objeto.
   * @param localServer Nome do servidor local.
   * @param localPort Porta do servidor local.
   * @param remoteServer Nome do servidor remoto.
   * @param remotePort Porta do servidor remoto.
   * @param atomicAction AtomicAction que acessará o objeto.
   */
  public MobileObject(String objectId,
                      String objectType,
                      String localServer,
                      String localPort,
                      String remoteServer,
                      String remotePort,
                      AtomicAction atomicAction);


  protected org.omg.CORBA.Object getReference();
}
```

## A.12. ObjectServer

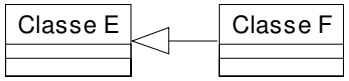
```
public class ObjectServer{
   /* ORB onde sera iniciado o servidor de objetos. */
   private ORB orb;
   /* ORB onde sera iniciado o servidor de objetos. */
   private POA poa;
   /* Instancia o servidor de objetos. */
   public ObjectServer();
   /* Inicia o servidor de objetos. */
   public void initORB(String args[]);
   /* Registra um objeto no serviço de nomes. */
   public void registerObject(String objectId, String objectType);
   /* Devolve o POA. */
   public POA getPOA();
}
```

# Apêndice B

# Relações entre Classes

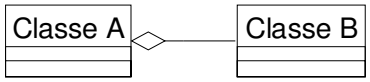
Esta Seção identificará notação utilizada para representar as relações entre classes nos diagramas de classes que foram apresentados neste trabalho.

## B.1. Herança



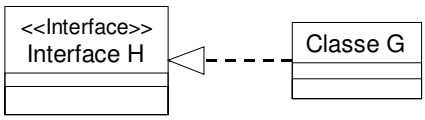

A Classe F herda a classe E.

## B.2. Agregação



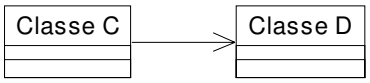

A classe A agrega a classe B.

## B.3. Implementação




A classe G implementa as operações da interface H.

## B.4. Uso




A classe C usa operações da classe D.