Um Sistema de Transações Adaptável para o Ambiente de Computação Móvel

Tarcisio da Rocha, Maria Beatriz Felgar de Toledo

Instituto de Computação – Universidade Estadual de Campinas (UNICAMP)
Caixa Postal 6.176 – 13.083-970 – Campinas – SP – Brazil

{tarcisio.rocha,beatriz}@ic.unicamp.br
http://www.ic.unicamp.br

Abstract. *This article focuses on transaction management for the mobile computing environment revising the traditional transaction model in order to make it more flexible. The proposed model allows applications to choose the required isolation level and the operation mode (remote or local). Another facility provides resource monitoring and notifications to applications when a certain resource is out of the required range. When a notification is received, the transaction may decide the appropriate steps to react to environment changes. Moreover the article presents a prototype developed in Java to validate the model.*

Resumo. *Esse artigo enfoca o gerenciamento de transações no ambiente de computação móvel revisando o modelo tradicional de transações de forma a torná-lo mais flexível. O modelo de transações proposto permite à aplicação escolher o grau de isolamento necessário e o modo de operação (remoto ou local). Outra facilidade oferecida possibilita o monitoramento de recursos e o envio de notificações para a transação quando um determinado recurso estiver fora de um intervalo considerado adequado. Ao receber uma notificação, a transação pode então decidir os passos apropriados para reagir a uma mudança no ambiente. Além disso, o artigo apresenta um protótipo desenvolvido em Java para validar o modelo.*

1. Introdução

Avanços no desenvolvimento de dispositivos de computação portáteis e de tecnologias de comunicação sem fio possibilitaram o surgimento da computação móvel. Assim, dispositivos portáteis, equipados com interfaces de comunicação sem fio, ganharam a capacidade de participar de computações distribuídas mesmo enquanto se movem.

A computação móvel, apesar de atraente, traz consigo uma série de obstáculos [Forman94, Imielinski92, Katz95] como, por exemplo, a baixa largura de banda, o alto custo de comunicação em redes sem fio, a maior susceptibilidade a falhas de comunicação e a baixa capacidade da bateria dos dispositivos portáteis. De forma a possibilitar que aplicações e sistemas distribuídos possam executar neste ambiente instável foi introduzido o conceito de adaptação [Jing97]. A partir deste conceito três estratégias de adaptação foram identificadas. Em um extremo, está a chamada adaptação laissez-faire propondo que cada aplicação deva individualmente prover meios de contornar os problemas do ambiente móvel. Esta estratégia apresenta como problema a falta de uma centralização no gerenciamento nos recursos compartilhados pelas

aplicações podendo assim gerar conflitos. No outro extremo, está a chamada adaptação transparente propondo que os problemas do ambiente móvel devam ser contornados pelo sistema sobre o qual as aplicações executam sem que estas tomem decisões. Esta segunda forma de adaptação traz o seguinte problema: as medidas de adaptação tomadas pelo sistema são refletidas a todas as aplicações sem respeitar suas individualidades. A outra forma de adaptação, chamada adaptação colaborativa, se encontra entre estes dois extremos. Ela propõe que o sistema de base seja responsável por monitorar o ambiente móvel e por prover os mecanismos de adaptação enquanto que as aplicações sejam responsáveis por definir individualmente as suas políticas de adaptação.

Nesse trabalho, enfocaremos o gerenciamento de transações no ambiente de computação móvel revisando o modelo tradicional de transações de forma a adequá-lo a esse tipo de ambiente. O modelo tradicional [Bernstein87] desenvolvido para aplicações de curta duração garante seriabilidade (serializability) geralmente através de trancas (locks). Já no ambiente móvel, é comum a existência de transações com longa duração o que exigiria a retenção de trancas por muito tempo. Além disso, nem toda aplicação necessita de um critério de correção tão restritivo como seriabilidade.

Tendo em vista estas considerações, propomos um modelo flexível de transações para o ambiente de computação móvel capaz de se adaptar às mudanças do ambiente móvel. Esta adaptação utiliza a estratégia de colaboração entre as transações e um sistema de base. O sistema de base é responsável por prover os mecanismos de adaptação, por monitorar o ambiente móvel e por notificar as transações sobre mudanças significativas no mesmo ambiente. Ao receber uma notificação, a transação pode decidir as medidas apropriadas de reação à mudança que deverão ser tomadas. O sistema provê às transações duas medidas de adaptação: a mudança do nível de isolamento e a mudança do modo de operação. Além do nível de seriabilidade, são providos outros três níveis de isolamento menos restritivos permitindo às transações ganhos em desempenho. São providos também três modos de operação (*local*, *remoto* e *local-remoto*) que permite às transações executarem mesmo quando desconectadas do restante da rede.

O restante deste trabalho está organizado como segue. A seção 2 apresenta o modelo de transações. A seção 3 trata dos componentes da arquitetura. A implementação do protótipo é discutida na seção 4. Na seção 5, são apresentados alguns trabalhos relacionados. A seção 6 termina o artigo apresentando as conclusões e os trabalhos futuros.

2. O Modelo de Transação Adaptável

O modelo de transações proposto permite que transações executem em um ambiente de comunicação sem fio sujeito à escassez de recursos e desconexões. Dentre estes recursos podemos citar: a largura de banda da comunicação entre o dispositivo móvel e a rede fixa; o tempo de autonomia das baterias do dispositivo móvel; e o espaço livre em disco do dispositivo móvel. Cada transação pode especificar os recursos do ambiente móvel que devem ser monitorados durante a sua execução e tomar medidas adequadas para se adaptar caso algum recurso não esteja disponível dentro dos limites adequados. Para isso, o sistema provê um mecanismo de monitoramento de recursos e notificação quando ocorrem variações significativas nos recursos de interesse das

transações. Uma transação, quando notificada, poderá se adaptar executando ações de contingência.

Estas ações de contingência podem consistir no cancelamento da transação ou na mudança de alguns parâmetros de execução da transação como o seu nível de isolamento e o seu modo de operação discutidos a seguir.

2.1 Níveis de Isolamento

O conceito de níveis de isolamento foi inicialmente introduzido por [Gray76] e se tornou base para as definições do padrão ANSI/ISO SQL-92 [ANSI92]. A exemplo de Oracle [Oracle95], tais níveis têm sido comumente usados em bancos de dados comerciais como um mecanismo de melhoria de desempenho. Críticas às definições ANSI e proposições de novos níveis são apresentados em [Bereson95, Adya99]. [Adya99] além de mostrar a definição de níveis de isolamento em função das trancas, propõe uma definição para níveis de isolamento que é independente de implementação adequando-se a outros métodos de controle de concorrência tais como os métodos otimistas [Bernstein87].

Como o método de controle de concorrência usado no nosso modelo é o de bloqueio em duas fases (two-phase locking) [Bernstein87], 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 1). Uma tranca de curta duração (ou tranca curta) sobre um objeto deve ser adquirida no início da execução da operação e liberada logo após o término da mesma. Já a tranca de longa duração (ou tranca longa), só deverá ser liberada no final da transação.

Tabela 1. Níveis de isolamento ANSI baseados em trancas

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

O nível mais baixo de isolamento (nível 0) permite que uma transação interfira na escrita da outra, evento este chamado de perda de atualização. Isto acontece quando duas transações concorrentes recuperam o mesmo valor e tentam atualizá-lo. Se estas transações executam sob o nível 0 de isolamento, a primeira atualização será perdida.

O nível 1 de isolamento evita a perda de atualização, porém permite que a transação leia dados que foram alterados por transações que ainda não foram efetivadas (committed). Este fenômeno é conhecido como leitura inconsistente (dirty read).

O nível 2 evita tanto a perda de atualização quando a leitura inconsistente, porém permite que uma leitura subseqüente ao mesmo dado apresente valores diferentes. Este fenômeno é conhecido como leitura que não pode ser repetida (non-repeatable read).

O nível 3 evita todos os fenômenos dos níveis inferiores e apresenta assim o nível mais alto de isolamento. As transações que executam sob este nível de isolamento são ditas serializáveis.

[Astrahan76] e [Adya99] descrevem com mais detalhes os fenômenos acima em função dos tipos de trancas que devem ser usados para evitá-los.

Apesar de os níveis de isolamento mais baixos oferecerem uma menor garantia de consistência eles dão às transações a vantagem de se envolverem em menos conflitos. Isto pode implicar na redução da comunicação e menores atrasos [Adya99]. Levando-se em conta que nem todas as transações exigem um nível de isolamento tão restritivo quanto a seriabilidade, faz sentido que estas possam optar por níveis de isolamento mais baixos principalmente ao se depararem com obstáculos da computação móvel como as desconexões freqüentes, a baixa largura de banda e provável alto custo da comunicação sem fio. Com isso, é dada às transações que executam no ambiente móvel, a capacidade de trocar consistência por desempenho. Um maior desempenho pode ainda implicar em uma redução de custos na comunicação sem fio, por exemplo.

Uma transação no sistema proposto (representada pela classe `MyAtomicAction`), ao iniciar, poderá especificar o nível de isolamento sob o qual as operações sobre os objetos de seu interesse deverão ser executadas (ver linha 5 do exemplo 1). O sistema também permite que este nível de isolamento seja mudado (para níveis mais baixos de isolamento) em qualquer ponto da execução da transação (ver linha 10 do exemplo 1).

A possibilidade de mudança do nível de isolamento durante a execução de uma transação traz benefícios: permite a uma transação delimitar as operações que deverão ser executadas em um maior ou menor nível de isolamento, e pode ser usada como uma forma de reação e adaptação às mudanças na disponibilidade de recursos do ambiente móvel.

Exemplo 1: Especificação e mudança de nível de isolamento

```
1  MyAtomicAction act; // Definição de uma transação
2  ObjectA objA;
3  ObjectB objB;
4
5  act.Begin(SERIALIZABILITY); // Inicia a transação com o nível 3 de
6                              // isolamento.
7    objA.operation1();
8    objB.operation2();
9    ...
10   act.setIsolationLevel(UNCOMMITTED_READ); // Muda isolamento.
11   objA.operation3();
12 act.End(); // Finaliza a transação
```

2.2 Modos de Operação

Diante da alta variabilidade e escassez de recursos do ambiente móvel, outro parâmetro que pode ser especificado no início da transação e modificado durante sua execução é o modo de operação. Os modos de operação considerados são três: *remoto*, *local-remoto* e *local*.

Quando uma transação opta pelo modo de operação *remoto*, toda operação sobre objetos é executada na máquina onde o objeto se encontra. O acesso a objetos remotos da rede fixa estará sujeito ao método de controle de concorrência de bloqueio em duas fases e obedecerá o nível de isolamento da transação. No final da transação, se o objeto remoto foi modificado seu respectivo número de versão é incrementado. Em caso de desconexão repentina, a transação que executa sobre o modo de operação *remoto* será abortada. As trancas desta transação sobre os objetos remotos estarão sujeitas à liberação automática feita pelos seus respectivos gerenciadores de trancas (classe `LockManager`). Além da liberação das trancas, os gerenciadores de trancas recuperarão

o estado dos mesmos objetos anterior à transação. Depois que a transação é abortada por uma desconexão, ela poderá esperar a conexão ser restabelecida ou reiniciar sua execução modo de operação *local*. Para que isto seja possível, os objetos requeridos pela transação devem estar em cache.

Uma transação também poderá optar pelo modo de operação *local-remoto*. Neste modo, uma cópia do objeto remoto com seu respectivo número de versão são transferidos para a cache da máquina móvel e as operações sobre o objeto são direcionadas somente para a sua cópia local em cache. Para manter a consistência entre a cópia local e a cópia remota na rede fixa, o objeto remoto deverá ser bloqueado. No término da transação, o objeto remoto é atualizado, sua tranca é liberada e tanto o número de versão do objeto remoto quanto o da cópia local são incrementados. Com o intuito de reduzir a comunicação com a rede fixa, neste modo de operação somente trancas longas são atribuídas aos objetos remotos. Assim, estas trancas, que são atribuídas no início da transação, só são liberadas no final da mesma. No caso de desconexão repentina não há a necessidade de abortar a transação, esta deverá mudar seu modo de operação para *local*. Essa mudança deve ser feita porque, assim como no modo *remoto*, as trancas sobre objetos remotos estarão sujeitas à liberação automática e à restauração do estado anterior, operações estas, feitas pelos respectivos gerenciadores de trancas dos objetos remotos.

No modo de operação *local*, uma cópia do objeto remoto com seu respectivo número de versão também são transferidos para a cache da máquina móvel, de forma que as operações sobre o objeto sejam direcionadas para a cópia em cache. Porém, neste modo de operação, o objeto remoto não receberá trancas por parte da transação que optou por este modo de operação. Portanto, esta transação poderá em algum momento estar operando em uma cópia local que não é consistente com a sua correspondente remota. Em caso de desconexão repentina, a transação poderá continuar no modo local e executar normalmente sobre as cópias dos objetos já em cache.

No final da transação que optou pelo modo de operação *local*, a transação irá disparar um processo de validação. Este processo consiste em, para cada objeto participante da transação:

- atribuir tranca de leitura sobre o objeto local e sobre o seu correspondente remoto;
- recuperar o número de versão do objeto local e o do seu correspondente remoto;
- se o número de versão do objeto local for igual ao do seu correspondente remoto, então:
  - objeto validado com sucesso;
- senão (o número de versão do objeto local for diferente ao do seu correspondente remoto), então:
  - abortar a transação e interromper a validação;

Depois do processo de validação, se todos os objetos participantes da transação foram validados com sucesso, cada objeto local terá seu número de versão incrementado, cada objeto remoto será atualizado com o estado do seu correspondente local e terá seu número de versão também incrementado.

O modo de operação de uma transação, assim como seu nível de isolamento, é um parâmetro que pode ser especificado no início de uma transação e mudado durante sua execução (Exemplo 2).

Exemplo 2: Especificação e mudança do modo de operação de uma transação

```
1  MyAtomicAction act; // Definição de uma transação
2  ObjectA objA;
3  ObjectB objB;
4  // Inicia transação com nível de isolamento 1 e modo de operação
5  // REMOTE.
6  act.Begin(UNCOMMITTED_READ, REMOTE);
7    objA.operation1();
8    objB.operation2();
9    ...
10   act.setOperationMode(LOCAL); // Muda modo de operação
11   objA.operation3();
12 act.End(); // Finaliza a transação
```

3 Arquitetura

A arquitetura do sistema para gerenciamento de transações proposto é mostrada na figura 1. Nesta arquitetura, o Monitor de Recursos monitora os recursos de interesse da transação que pode acessar tanto objetos remotos como locais. O Gerenciador de Cache é o responsável por trazer cópias de objetos remotos para a máquina móvel. A recuperação e armazenamento de objetos persistentes ficam a cargo do Repositório de Objetos.

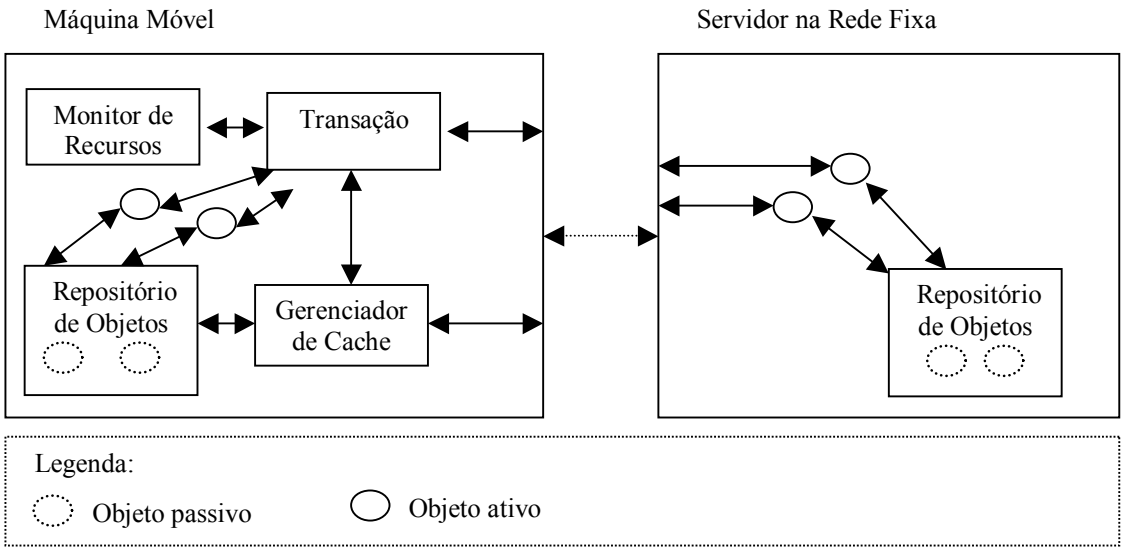

Figura 1. Arquitetura

3.1 Monitor de Recursos

O Monitor de Recursos é responsável pelo monitoramento dos recursos necessários para a execução das transações. Cada transação pode requisitar o monitoramento de um ou mais recursos especificando, também, os limites superior e inferior para cada recurso. O Monitor de Recursos registra, para cada transação, as quantidades desejadas de cada

recurso e notifica a transação quando um determinado recurso estiver fora do intervalo especificado.

Este mecanismo de notificação dá à transação a capacidade de estar ciente das variações significativas nos recursos que lhe interessam possibilitando, assim, a tomada de medidas de adaptação por parte da mesma.

Entre os recursos que podem ser monitorados podemos citar largura de banda, espaço em disco, tempo de autonomia da bateria do dispositivo móvel, por exemplo.

3.2 Gerenciador de Cache

O Gerenciador de Cache é responsável pelo armazenamento de objetos na máquina móvel. Provê operações como: transferência de estado de objetos remotos para a cache; remoção de um objeto da cache; verificação de consistência entre a cópia do objeto em cache e a sua réplica remota; marcação de objetos que não podem ser removidos da cache. Esta última operação pode ser utilizada por aplicações que desejam evitar que determinados objetos úteis na sua execução sejam removidos por outras aplicações.

O Gerenciador de Cache é um dos elementos que viabilizam a não interrupção das transações na máquina móvel quando ocorrem desconexões. Nesses casos, a transação pode continuar operando somente sobre cópias locais e deve sofrer validação posterior. Os objetos necessários para a transação podem ser trazidos sob demanda ou antecipadamente em background.

3.3 Repositório de Objetos

O Repositório de Objetos possui operações para recuperar estados de objetos da memória secundária e salvar objetos em memória secundária. A operação de recuperação é utilizada para ativação de um objeto e a operação para salvar um objeto é necessária na sua desativação.

Dessa forma, podemos então resumir o ciclo de vida básico de um objeto:

- O objeto está inicialmente em sua forma passiva armazenado no Repositório de Objetos.
- Quando uma transação faz uma invocação de operação sobre esse objeto, ele é ativado (seu estado é recuperado do Repositório de Objetos ficando disponível em memória primária) passando da forma passiva para ativa.
- Depois de um período sem que haja invocações sobre o objeto, ele é desativado (seu estado é salvo no Repositório de Objetos) voltando para o estado passivo.

4 Implementação do Protótipo

O protótipo do sistema proposto foi implementado com o uso das tecnologias Java [JKD1.4] (JDK 1.4) e CORBA [OMG96] (Java IDL – implementação CORBA da Sun).

Escolhemos Java por ser multiplataforma permitindo assim que o sistema seja executado em diferentes hardwares e sistemas operacionais, pelas suas API's que nos fornece um conjunto de facilidades de implementação e pelo seu suporte a CORBA.

A seguir descrevemos as classes principais desse sistema com exemplos de utilização.

4.1 Ciclo de Vida

As classes envolvidas no ciclo de vida de um objeto são: `ObjectStore`, `ObjectState` e `Buffer`.

A classe que implementa o Repositório de Objetos é `ObjectStore` (Exemplo 3). Essa classe possui a função de recuperar o estado do objeto da memória secundária e carregá-lo da memória primária e vice-versa. Quando uma operação é invocada sobre um objeto na forma passiva, o seu estado é recuperado através da operação `read_committed`. Esta operação devolve uma instância da classe `ObjectState` (Exemplo 4) contendo o estado do objeto em sua forma linearizada[1]. O estado do objeto é então restaurado através da operação `restore_state`. Quando um objeto é desativado, após um período em que não ocorrem invocações de operações sobre ele, seu estado é linearizado através da operação `save_state` e salvo através da operação `write_committed`. Como podemos ver no Exemplo 5, as operações `save_state` e `restore_state` são implementadas pelo próprio objeto.

Exemplo 3: Classe ObjectStore

```
1 class ObjectStore{
2    // Recupera o estado de um determinado objeto
3    ObjectState read_committed(String objectId, String objectType);
4
5    // Armazena o estado de um objeto em memória não volátil
6    boolean write_committed(ObjectState os);
7    ...
8 }
```

Para linearizar o estado de um objeto, são necessárias as classes `Buffer` e `ObjectState`. A classe `Buffer` contém operações capazes de armazenar o conteúdo de cada atributo de um objeto (Exemplo 4). A classe `ObjectState` herda da classe Buffer.

Exemplo 4: Classes Buffer e ObjectState

```
1  class Buffer {
2     // Empacota tipos no buffer.
3     void packInt(int data);
4     void packString(String data);
5     ...
6     // Recupera tipos do buffer.
7     int unpackInt();
8     String unpackString();
9     ...
10 }
11
12 class ObjectState extends Buffer{
13    // Atributos que armazenam o identificador e o tipo do objeto
14    String objectId;
15    String objectType;
16    ...
17 }
```

[1] A forma linearizada do estado de um objeto consiste em um array composto pelo conteúdo em bytes dos atributos que constituem o estado do objeto.

Exemplo 5: Implementação dos métodos save_state e restore_state

```
1  class ObjectA{
2     // Atributos que constituem o estado do objeto
3     string variable_1;
4     long variable_2;
5     ...
6     // Empacota o estado do objeto no ObjectState
7     void save_state(ObjectState os){
8        os.packString(variable_1);
9        os.packLong(variable_2);
10    }
11    // Restaura o estado do objeto a partir do ObjectState
12    void restore_state(ObjectState os){
13       variable_1 = os.unpackString();
14       variable_2 = os.unpackLong();
15    }
16 }
```

4.2 Gerenciamento de Cache

A classe `CacheManager` (Exemplo 6) implementa o Gerenciador de Cache. Ela possui a operação `loadState` que copia o estado de um objeto de um repositório remoto para o repositório local (cache). A operação `isConsistent` verifica se o estado de um objeto do repositório local está consistente com o seu estado no repositório remoto. A operação `markState` marca um objeto para indicar que esse objeto não deve ser removido do cache. A operação `unmarkState` indica que o objeto pode ser removido do cache.

Exemplo 6: A classe CacheManager.

```
1  class CacheManager{
2     load loadState (String objectId, String objectType,
3                     String server, String port);
4     boolean isConsistent (String objectId, String objectType,
5                           String server, String port);
6     void markState (String objectId, String objectType);
7     void unmarkState (String objectId, String objectType);
8     ...
9  }
```

4.3 Monitoramento de Recursos

A classe responsável pelo monitoramento de recursos no ambiente móvel é a `ResourceMonitor` (ver Exemplo 8). Esta classe encapsula a implementação de vários monitores de recursos independentes onde cada um destes implementa a interface `MonitorInterface` (ver Exemplo 9). Um novo monitor de recurso independente é adicionado ao `ResourceMonitor` através da operação `addMonitor`. Dessa forma, pode-se implementar e adicionar os diversos monitores de recurso ao `ResourceMonitor`, como por exemplo, um monitor para largura de banda, monitor de energia e monitor de espaço em disco.

Exemplo 8: A classe ResourceMonitor

```
class ResourceMonitor{
   string request(AtomicAction aaRef, ResourceDescriptor descriptor);
   void cancel(string requestId);
   void addMonitor(MonitorInterface monitor); ...
}
```

Exemplo 9: A interface MonitorInterface

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

A classe `ResourceMonitor` possui operações que possibilitam a uma transação requisitar o monitoramento de um determinado recurso (operação `request`) e cancelar a requisição de monitoramento efetuada (operação `cancel`). Uma transação (`AtomicAction`), ao requisitar o monitoramento de recurso, especifica como parâmetro a sua referência e o descritor do recurso – composto pelo identificador do recurso a ser monitorado e pelos limites superior e inferior de tolerância às variações do recurso. Quando uma variação no recurso monitorado extrapolar os limites especificados pela transação, esta receberá uma notificação por parte do monitor de recursos.

4.4 Recuperação e Persistência

A classe `StateManager` é responsável pela ativação e desativação de um objeto (ver operações `activate` e `desactivate` no Exemplo 10). Na ativação de um objeto, seu estado (`ObjectState`) é recuperado do repositório de objetos e carregado na memória primária através da operação `restore_state`. Na desativação de um objeto, seu estado é linearizado (`ObjectState`) através da operação `save_state` e armazenado no repositório de objetos.

`StateManager` é uma classe abstrata que deve ser herdada pelos objetos que deverão ter seus estados armazenados em um `ObjectStore`. Três são os as operações abstratas que devem ser implementadas por tais objetos: `save_state`, `restore_state` e `type`.

Exemplo 10: A classe abstrata StateManager.

```
1  abstract class StateManager{
2     boolean activate();
3     boolean deactivate();
4     ...
5     // Metodos a ser implementados pelo objeto a ser gerenciado
6     abstract boolean restore_state(ObjectState objState,
7                                    int objectType);
8     abstract boolean save_state(ObjectState objState,
9                                 int objectType);
10    abstract String type();
11    ...
12 }
```

4.5 Controle de Concorrência

O controle de concorrência de objetos participantes de uma transação é feita pela classe `LockManager` (Exemplo 11) que utiliza o método de bloqueio em duas fases. Para que um objeto possua controle de concorrência, sua classe deverá herdar de `LockManager` e implementar as suas operações abstratas `restore_state`, `save_state` e `type`. Esta última operação devolve o tipo da classe que o está implementando.

`LockManager` é uma classe abstrata que herda da classe `StateManager`. Portanto, as classes que herdarem de `LockManager` possuirão tanto operações de controle de concorrência quanto operações de recuperação e persistência.

Exemplo 11: A classe abstrata LockManager

```
1  abstract class LockManager extends StateManager{
2     int setLock(Lock lock);
3     int releaseLock(String lockId);
4     ...
5     // Operações abstratas herdadas
6     abstract boolean restore_state(ObjectState objState,
7                                    int objectType);
8     abstract boolean save_state(ObjectState objState,
9                                 int objectType);
10    abstract String type();
11    ...
12 }
```

A classe `LockManager` possui as operações `setLock` e `releaseLock`. Quando uma classe `C` herda de `LockManager`, a operação `setLock` deverá ser chamada antes da execução de qualquer operação de `C`. No Exemplo 12 mostramos uma classe `C` que herda de `LockManager` e possui a operação de escrita operation_1. Antes da execução da operação, uma tranca de escrita deve ser obtida (linha 4).

A liberação de uma tranca é feita automaticamente se a operação de aquisição de tranca foi feita dentro do escopo de uma transação (`AtomicAction`). Se a aquisição de uma tranca for feita fora do escopo de uma transação, a sua liberação é de responsabilidade do programador através da chamada da operação `release_lock`.

Exemplo 12: A operação setLock

```
1  class C extends LockManager{
2     ...
3     void operation_1{
4        if (setLock(new Lock(WRITE) == GRANTED)){
5           // Executa a operação de escrita
6        }
7     }
8  }
```

O parâmetro do método `setLock` é uma instância da classe `Lock` (ver Exemplo 13). Ao criar uma nova instância da classe `Lock`, deve ser especificado o modo da tranca como parâmetro. O nosso sistema implementa modos básicos de tranca: o de leitura e o de escrita. A tabela 2 mostra a relação de conflito entre estas trancas. A operação `conflictWith` verifica se uma tranca é conflitante com outra e a operação `modifiesObject` verifica se a tranca é para modificação ou não.

Exemplo 13: A classe Lock

```
1  class Lock{
2     // Construtor
3     Lock(short lockMode);
4     // Operações
5     boolean conflictsWith(Lock otherLock);
6     boolean modifiesObject();
7     ...
8  }
```

Tabela 2: Relação de conflito entre trancas

	READ	WRITE
READ	Sem conflito	Conflito
WRITE	Conflito	Conflito

No modelo proposto neste artigo, tanto a cópia local quanto a sua correspondente remota estarão sujeitas a controles de concorrência independentes. Assim, transações que acessam uma mesma cópia local estarão sujeitas ao controle de concorrência local que garantirá a sua consistência local. O modelo proposto neste artigo permite que só exista uma cópia de cada objeto remoto no cache de cada máquina móvel. Sendo assim, quando a cópia remota tiver que ser atualizada com a cópia em cache validada, esta certamente será uma cópia localmente consistente que passará a ser uma cópia globalmente consistente.

4.6 Gerenciamento de Transação

A implementação das operações de uma transação é definida na classe `AtomicAction` (ver exemplo 14). As operações `Begin` e `End` delimitam uma transação. Ao iniciar, uma transação poderá especificar parâmetros de execução como o nível de isolamento e o modo de operação. Se estes parâmetros não forem especificados, a transação executará com nível 3 de isolamento (`SERIALIZABILITY`) e modo de operação remoto (`REMOTE`) como padrão.

A operação `Abort` deverá ser chamada quando for necessário abortar a execução da transação. As operações `setIsolationLevel` e `setOperationMode` servem para mudar o nível de isolamento e o modo de operação da transação no decorrer de sua execução.

Exemplo 14: A classe abstrata AtomicAction

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

A classe `AtomicAction` possui uma operação abstrata `_notify` que deve ser implementada pelo programador. Esta operação é chamada pelo Monitor de Recursos quando ocorre uma variação significativa na disponibilidade de um determinado recurso que está sendo monitorado a pedido da transação. Os parâmetros passados na chamada da operação são: o identificador da requisição de monitoramento feita pela transação ao Monitor de Recursos, o identificador do Monitor de Recurso que chamou a operação e nível atual do recurso monitorado.

A implementação da operação `_notify` normalmente será uma medida de reação à mudança ocorrida no ambiente móvel. Esta reação poderá consistir em uma chamada de `abort` à transação, na execução de um conjunto de operações de

recuperação, bem como na mudança do seu nível de isolamento e do seu modo de operação.

4.7 Exemplo de Uso

Nesta seção, mostraremos um exemplo simples de como o modelo proposto neste artigo pode ser usado. O exemplo consiste em uma transação que executa na máquina móvel definida pela classe `MyAtomicAction` (Exemplo 16). A política de adaptação definida em nosso exemplo muda o modo de operação e o nível de isolamento da transação para `LOCAL` e `UNCOMMITTED_READ`, respectivamente (linhas 8 e 9 do Exemplo 15).

Exemplo 15: Definição da política de adaptação

```
1  class MyAtomicAction extends AtomicAction{
2
3     void _notify (String requestId, String monitorId,
4                   int resourceLevel){
5        // Política de adaptação
6        if (monitorId == "BandwidthMonitor"){
7           if (resourceLevel < 10000 ){
8              setOperationMode(LOCAL);
9              setIsolationLevel(UNCOMMITTED_READ)
10          }
11       }
12    }
13 }
```

Tendo definido a política de adaptação da transação, mostraremos no Exemplo 17 como a mesma pode ser utilizada. As operações `Begin` e `End` delimitam o escopo da transação. A transação inicia com o nível de isolamento `SERIALIZABILITY` e modo de operação `REMOTE`. Na linha 6 do mesmo exemplo, a transação requisita o monitoramento da largura de banda ao respectivo monitor de recurso. Da linha 7 à linha 11, está o conjunto de operações sobre objetos. Como a transação iniciou com o modo de operação `REMOTE`, as operações sobre os objetos `objA` e `objB` serão direcionadas para a sua correspondente remota localizada no servidor da rede fixa.

Exemplo 16: Utilizando uma transação adaptativa

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

Se durante a execução dessas operações, a transação receber uma notificação do monitor de recursos, o método `_notify` (Exemplo 15) será executado. A execução deste método fará com que o modo de operação e o nível de isolamento da transação sejam mudados para `LOCAL` e `UNCOMMITTED_READ` respectivamente.

5. Trabalhos Relacionados

Um dos primeiros trabalhos relevantes para o modelo apresentado neste artigo foi o Coda [Kistler92] é um sistema de arquivos distribuídos que provê adaptação transparente às aplicações diante das falhas de rede. O Coda introduziu o conceito de operação desconectada permitindo assim a uma máquina portátil desconectada da rede a possibilidade de continuação das suas atividades sobre a réplica local dos dados remotos. O conceito de operação desconectada foi aplicado a transações no presente artigo através do chamado modo de operação *local*. Assim como no Coda, o modo de operação *local* usa a replicação otimista, visto que não evita conflitos entre os dados remotos e suas réplicas locais. Estes conflitos são detectados posteriormente através de um processo de validação. Porém, o modelo deste artigo, através do modo de operação *local-remoto*, também usa a replicação pessimista para evitar antecipadamente estes conflitos. Um modelo de transações para o Coda foi apresentado posteriormente em [Lu94]. Diferentemente do modelo transacional proposto neste artigo que visa a garantia das propriedades ACID[Bernstein87], o Coda só visa a garantia de uma estas propriedades: o isolamento.

Estudos posteriores ao Coda deram origem ao Odyssey [Noble98]. Ao contrário do Coda que oferece mecanismos de adaptação transparentes, Odyssey provê adaptação colaborativa entre aplicações e sistema operacional. A adaptação colaborativa trouxe para cada aplicação a possibilidade de definir individualmente a sua política de adaptação. Porém, os experimentos realizados no Odyssey foram focados em aplicações que operam com dados multimídia como, áudio, vídeo e imagens. Esses dados são passíveis de manipulação, por exemplo, uma imagem pode ter o seu tamanho em bytes reduzido através da manipulação da sua resolução. Isso seria uma forma de se adaptar a uma rede com baixa largura de banda. No presente artigo utilizamos a estratégia de colaboração em aplicações que operam em dados numéricos que não possuem a capacidade de manipulação dos dados multimídia. Para tanto, foi dada a possibilidade de relaxamento das regras de consistência destes dados como uma das formas de adaptação.

Vários são os modelos transacionais citados na literatura voltados à computação móvel. Alguns destes, como em [Chrysanthis93] e [Dunham97], 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. [Chrysanthis93] 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 [Dunham97] captura a idéia de movimento criando novas subtransações à medida que a máquina portátil muda de uma célula para outra. O presente artigo propõe mecanismos de continuação da execução das transações através de mudança nos parâmetros como o modo de operação sem que haja subdivisão das mesmas.

Outros modelos de transação são apresentados em [Walborn99] e [Pitoura95]. [Walborn99] é baseado em *compacts* que encapsulam dados e informação sobre seu gerenciamento. Transações podem ser executadas em modo desconectado sobre dados locais armazenados sob a forma de *compacts*. As reações de adaptação no Pro-Motion

são tomadas pelos próprios *compacts* que por sua vez são refletidas a todas às transações que os utilizam. Já no modelo proposto neste artigo, a reação de adaptação é de decisão das próprias transações. [Pitoura95] define dois tipos de transações: transações fracas que podem ser executadas sobre dados com certo grau de inconsistência e transações fortes que executam sobre dados consistentes. Porém, não é dada a uma transação a capacidade de decisão de mudança da consistência das operações durante a sua execução.

6. Conclusões e Trabalhos Futuros

O modelo de transações proposto nesse artigo busca integrar a estratégia de adaptação colaborativa com o gerenciamento de transações com o objetivo de oferecer a flexibilidade exigida pelo ambiente de computação móvel. Dessa forma, cada transação pode requisitar o monitoramento de recursos de seu interesse e, posteriormente, ser notificada sobre variações na disponibilidade desses recursos. Ao receber uma notificação, a transação pode executar individualmente ações de contingência como, por exemplo, abortar ou alterar seus parâmetros de execução.

No início da transação, dois parâmetros podem ser especificados: nível de isolamento e modo de operação. Ambos os parâmetros podem ser mudados durante a execução da transação. A mudança no nível de isolamento de uma transação permite que esta possa trocar consistência por desempenho e a mudança no modo de operação permite que uma transação continue a executar sobre dados locais mesmo quando desconectado da rede fixa.

Entre as contribuições principais desse trabalho, podemos citar a utilização da estratégia de adaptação colaborativa no contexto de gerenciamento de transações e a verificação do modelo de transações proposto através da implementação de um protótipo utilizando Java e CORBA.

Como trabalhos futuros, serão mostrados os resultados obtidos com o uso do protótipo comparando-os com resultados de outros trabalhos relacionados. Serão usadas algumas medidas de desempenho que irão avaliar os impactos no tráfego da comunicação sem fio causados pela transferência de estados entre a rede fixa e a máquina móvel e vice-versa. Será avaliado o ganho em desempenho de transações que optam por níveis de isolamento mais baixos em função da disponibilidade de recursos do ambiente móvel como a largura de banda. Será também avaliada a experiência de utilização do CORBA em um ambiente móvel.

Referências

[Adya99] Adya, A. (1999) “Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions”, Massachusetts Institute of Technology, PhD Thesis, March.

[ANSI92] ANSI X3.135-1992 (1992) “American National Standard for Information Systems – Database Languages – SQL”, November.

[Astrahan76] Astrahan, M. M. et al. (1976) “System R: Relational Approach to Database Management”, ACM Transactions on Database Systems, 97-137, June.

[Bereson95] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E. e O'Neil, P. (1995) "A Critique of ANSI SQL Isolation Levels", Proceedings of SIGMOD, San Jose, CA, May.

[Bernstein87] Bernstein, P. A., Hadzilacos, V. e Goodman, N. (1987) "Concurrence Control and Recovery in Database Systems", Addison Wesley.

[Chrysanthis93] Chrysanthis, P. K. (1993) "Transaction Processing in a Mobile Computing Environment", Workshop on Advances in Parallel and Distributed Systems, 77-82.

[Dunham97] Dunham, M. H. e Helal, A. (1997) "A Mobile Transaction Model that Captures Both the Data and the Movement Behavior", ACM-Baltzer Journal on Mobile Networks and Applications, 149-161.

[Forman94] Forman, G. H. e Zahorjan, J., (1994) "The Challenges of Mobile Computing", University of Washington, IEEE, April.

[Gray76] Gray, J., Lorie, R., Putzolu, G. e Traiger, I. (1976) "Granularity of Locks and Degrees of Consistency in a Shared Database", Modeling in Database Management Systems, Amsterdam: Elsevier North-Holland.

[Imielinski92] Imielinski, T. e Badrinath, B. R. (1992) "Mobile Wireless Computing: Challenges in Data Management", Rutgers University, Department of Computer Science.

[JDK1.4] "Java2 Platform, Standard Edition" http://java.sun.com/j2se/1.4.1/index.html

[Jing97] Jing, J., Helal, A. e Elmagarmid, A. (1997) "Client-Server Computing in Mobile Environments", ACM Transactions on Database Systems.

[Katz95] Katz, R. H. (1995) "Adaptation and Mobility in Wireless Information Systems", University of California.

[Kistler92] Kistler, J. J. e Satyanarayanan, M. (1992) "Disconnected Operation in Coda File System", Carnegie Mellon University, ACM.

[Lu94] Lu, Q. e Satyanaranyanan M. (1994) "Isolation-only transactions for mobile computing", ACM Operating Systems Review, 28(3).

[Mummert95] Mummert, L. B., Ebling, M. R. e Satyanarayanan, M. (1995) "Exploring Weak Connectivity for Mobile File Access", Carnegie Mellon University, ACM.

[Noble98] Noble, B. D. (1998) "Mobile Data Access", Carnegie Mellon University, PhD Thesis.

[OMG96] OMG (1996) "The Common Object Request Broker: Architecture and Specification", Technical Report.

[Oracle95] Oracle Corporation (1995) "Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7", July.

[Pitoura95] Pitoura, E. e Bhargava, B. (1995) "Maintaining Consistency of Data in Mobile Distributed Environment", 15th Int. Conference on Distributed Computer Systems.

[Walborn99] Walborn, G. D. e Chrysanthis, P. K. (1999) "Transaction Processing in PRO-MOTION", 14th ACM Annual Symposium on Applied Computing.