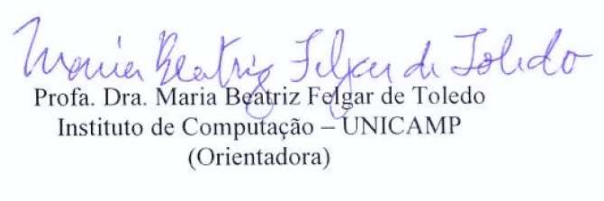
Transações Reconfiguráveis para o Ambiente Móvel

Este exemplar corresponde à redação da Dissertação apresentada para a Banca Examinadora antes da defesa da Dissertação.

Campinas, 18 de Setembro de 2009.



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

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Crisllene Queiroz Custódio – CRB8 / 7966

Pierre, Allyn Grey de Almeida Lima
P614t Transações reconfiguráveis para o ambiente móvel / Allyn Grey de Almeida Lima Pierre -- Campinas, [S.P. : s.n.], 2009.

Orientador : Maria Beatriz Felgar de Toledo
Dissertação (Mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Computação móvel. 2. Sistemas de computação adaptativos. 3. Reconfiguração dinâmica. 4. OpenCOM (Modelo de componentes) - Software. 5. Controle de concorrência. 6. Nível de isolamento. I. Toledo, Maria Beatriz Felgar de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Reconfigurable Transactions for Mobile Environment.

Palavras-chave em inglês (Keywords): 1. Mobile computing. 2. Adaptive computing systems. 3. Dynamic reconfiguration. 4. OpenCOM (component model) - Software. 5. Concurrency control. 6. Isolation level.

Área de concentração: Sistemas Distribuídos

Titulação: Mestre em Ciência da Computação

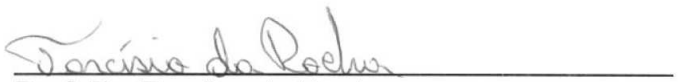
Banca examinadora: Profa. Dra. Maria Beatriz Felgar de Toledo (IC - UNICAMP)
Profa. Dra. Islene Calciolari Garcia (IC - UNICAMP)
Prof. Dr. Tarcisio da Rocha (DCOMP - UFS)

Data da defesa: 18/09/2009

Programa de Pós-Graduação: Mestrado em Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 18 de setembro de 2009, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Tarciso da Rocha
Departamento de Computação / Universidade Federal de Segipe.

Profª. Drª. Islene Calciolari Garcia
IC / UNICAMP.

Instituto de Computação
Universidade Estadual de Campinas

Transações Reconfiguráveis para o Ambiente Móvel

Allyn Grey de Almeida Lima Pierre

Setembro de 2009

Banca Examinadora

- Maria Beatriz Felgar de Toledo (Orientadora)
  Instituto de Computação – UNICAMP
- Islene Calciolari Garcia
  Instituto de Computação – UNICAMP
- Tarcisio da Rocha
  Departamento de Computação - Universidade Federal de Sergipe
- Anamaria Gomide (Suplente interno)
  Instituto de Computação – UNICAMP
- Marcelo Fantinato (Suplente externo)
  Universidade de São Paulo

Resumo

Dentre as tecnologias emergentes, a computação móvel tem a sua posição de destaque. Os dispositivos móveis estão mais presentes na vida das pessoas e contendo aplicações mais sofisticadas e semelhantes às executadas em computadores pessoais. Num mundo globalizado, onde o tempo é escasso e valioso, os dispositivos móveis mantêm as pessoas em contato com informações e atividades que elas desejam enquanto elas estão em movimento. Um exemplo recente é aumento do uso da internet em celulares, permitindo que os usuários acessem diversos tipos de aplicações, tendo grande parte delas interação com bancos de dados.

Apesar de atrativa, a computação móvel traz desafios ao desenvolvedor, pois ele deve considerar os recursos limitados tais como largura de banda, conectividade e o alto custo da obtenção de dados.

Nesse contexto, as transações representam um importante papel de garantir que o dinamismo do ambiente da computação móvel não comprometa a confiabilidade das aplicações. Porém, algumas aplicações não podem ser implementadas considerando o modelo de transações tradicional, pois elas têm um tempo mais longo de duração do que aquelas convencionalmente modeladas. Sendo assim, as configurações de uma transação realizadas no início de sua execução podem deixar de ser adequadas no decorrer da sua execução, devido às mudanças no ambiente.

Diversos modelos de transações têm sido apresentados na literatura para atender a esse ambiente. Apesar de muitas idéias interessantes e relevantes, alguns modelos não permitem que a adaptação diante da variação dos recursos seja realizada durante a execução de uma transação e quando permitem, eles realizam grandes reconfigurações arquiteturais.

Motivada por essas questões, essa dissertação propõe transações reconfiguráveis, isto é, a configuração dinâmica de mecanismos transacionais antes do início da transação e a reconfiguração de propriedades transacionais durante sua execução. Para que a reconfiguração dinâmica fosse realizada, um modelo de componentes chamado OpenCOM foi utilizado na arquitetura proposta, por este ser reflexivo, leve e independente de plataforma. O nível de isolamento é a propriedade transacional que poderá ser reconfigurada durante a transação e o controle de concorrência é o mecanismo que garantirá o isolamento entre as transações e poderá ser configurado antes do início da transação. A configuração do controle de concorrência é uma contribuição inovadora dessa dissertação, pois em muitos trabalhos existentes não é possível a configuração desse mecanismo transacional.

A fim de validar a arquitetura proposta, um protótipo de um sistema de vendas foi desenvolvido. Através dessa implementação foi possível analisar os impactos da reconfiguração durante uma transação.

Abstract

Among the emerging technologies, mobile computing has its position of prominence. Mobile devices are more present in people's lives and with more sophisticated applications similar to those implemented in personal computers. In a globalized world where time is scarce and of great importance, mobile devices keep people in touch with information and activities they want while they are moving. A recent example is the increasing use of the Internet on mobile phones allowing users to access various types of applications and much of them interacting with databases.

Although attractive, the mobile computing brings challenges to the developer because he must consider the limited resources such as bandwidth, connectivity and the high cost of obtaining data.

In this context, the transactions represent an important role to ensure that the dynamic environment of mobile computing does not compromise the reliability of applications. However some applications cannot be implemented given the traditional transactions models because they have a longer duration than those conventionally shaped. Therefore the settings of a transaction carried out before its execution may not be appropriate during the execution due to changes in the environment.

Various transactions models have been reported in the literature to serve this environment. Although having many interesting and relevant ideas, some models do not allow the adaptation in the face of change of resources during the execution of a transaction and when this is allowed, they require many transactional reconfigurations.

Motivated by these issues, this dissertation proposes reconfigurable transactions that are the dynamic configuration of transactional mechanisms before the beginning of the transaction and the reconfiguration of transactional properties during its execution. For dynamic reconfiguration, a component model called OpenCOM has been used in the proposed architecture because it is reflective, lightweight and platform-independent. The isolation level is the property that may be reconfigured during the transaction and the concurrency control is the mechanism that ensures the isolation between the transactions and it can be configured before the beginning of transaction. The configuration of concurrency control is an original contribution of this dissertation because many works do not allow the configuration of this transactional mechanism.

In order to validate the proposed architecture, a prototype of a sales system has been developed. Through this implementation it was possible to analyze the impacts of the reconfiguration during a transaction.

Agradecimentos

Agradeço a Deus, por me dar fé e saúde para vencer mais essa etapa.

À minha orientadora Maria Beatriz Felgar de Toledo, pelos seus valiosos ensinamentos e críticas que tanto contribuíram para o meu crescimento pessoal, profissional e acadêmico. Obrigada por compartilhar toda a sua experiência e sabedoria.

Aos meus queridos pais e ao meu querido irmão, por serem meus mestres durante toda a minha vida. A toda a minha família, pelo apoio e carinhos recebidos.

Ao meu querido marido, meu grande companheiro, por todo o amor, compreensão e incentivo durante esse trabalho.

Ao colega Tarcisio da Rocha, por todas as suas excelentes sugestões. Com certeza você contribuirá para o nosso país através de suas pesquisas e de seus ensinamentos.

Ao colega Anderson de Almeida, por fazer parte da idéia inicial desse projeto quando cursamos a disciplina de Tópicos Distribuídos.

Ao corpo docente do Instituto de Computação, pelos ensinamentos durante as disciplinas que cursei.

Aos membros da banca de qualificação, pelas sugestões e contribuições durante a apresentação.

Aos funcionários do Instituto de Computação e da Diretoria Acadêmica que contribuíram diretamente ou indiretamente na realização dessa dissertação.

A todos meus amigos que 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) e Organização da Dissertação (Seção 1.2).

1.1 Motivação

Dentre as tecnologias emergentes nos próximos anos, a computação móvel tem a sua posição de destaque. Os dispositivos móveis estão mais presentes na vida das pessoas e contendo aplicações mais sofisticadas e semelhantes às executadas em computadores pessoais.

Um dos exemplos mais recentes é o surgimento da plataforma aberta para dispositivos móveis, chamada Android [2], criada pela empresa Google que promete revolucionar o desenvolvimento de aplicações nesse ambiente. Outro exemplo é o uso crescente da internet em celulares, permitindo que os usuários acessem diversos tipos de aplicações, tendo grande parte delas interação com banco de dados. Há um número cada vez maior de novos sistemas móveis que necessitam de acessos a bancos de dados utilizando equipamentos, como PDAs (*Personal Digital Assistants*) e celulares.

Dentre muitos tipos de sistemas que utilizam dispositivos móveis e banco de dados, pode-se citar sistema de controle de vendas e *check-in* pelo celular.

Sendo a mobilidade uma das tendências do futuro, preocupações com integridade e consistência dos dados, rapidez de acesso à informação e segurança, tornam-se cada vez mais comuns em pesquisas e soluções apresentadas para esse ambiente. A computação móvel traz desafios ao desenvolvedor, pois ele deve considerar os recursos limitados tais como largura de banda, conectividade e o alto custo da obtenção de dados.

Nesse contexto, as transações representam um importante papel de garantir que o dinamismo do ambiente da computação móvel não comprometa a confiabilidade das aplicações. Porém, algumas aplicações não podem ser implementadas considerando o modelo de transações tradicional, pois elas normalmente têm um tempo mais longo de duração do que aquelas convencionalmente modeladas.

Diversos modelos de transações têm sido propostos na literatura, cada um analisando um domínio específico e algumas características das propriedades transacionais. Idéias relevantes têm sido propostas em modelos de transação como, por exemplo, AMT [1] e MobileTrans [46]. Esses modelos permitem a adaptação diante da variação dos

recursos disponíveis do ambiente móvel, mas não permitem que essa adaptação seja realizada durante a execução da transação.

Motivada por essas questões e por algumas idéias proposta no modelo SGTA [43], essa dissertação propõe a configuração de mecanismos transacionais antes do início da transação e a reconfiguração de algumas propriedades durante a transação.

O nível de isolamento é a propriedade transacional que poderá ser reconfigurada durante a transação e o controle de concorrência é o mecanismo que garantirá o isolamento entre as transações e poderá ser configurado antes do início da transação.

A configuração do controle de concorrência é uma proposta inovadora em comparação aos modelos transacionais existentes.

Um modelo de componentes denominado OpenCOM [36] foi o escolhido para implementação da arquitetura proposta. Como esse modelo utiliza a técnica de reflexão computacional [49], ele provê uma facilidade de adaptação da arquitetura dos componentes diante de certas mudanças do ambiente.

A fim de verificar a arquitetura proposta, um protótipo de um sistema de vendas foi desenvolvido.

1.2 Organização da Dissertação

Essa dissertação está organizada em seis capítulos descritos abaixo:

- Capítulo 2: apresenta os conceitos que servem como base para o entendimento da dissertação. Os conceitos básicos abordados são: computação móvel, transações com foco em controles de concorrência e níveis de isolamento, reflexão computacional e arquitetura baseada em componentes.
- Capítulo 3: descreve o modelo proposto para transações reconfiguráveis para o ambiente móvel.
- Capítulo 4: discute a implementação do protótipo da arquitetura proposta no capítulo 3 e apresenta alguns estudos casos de uma aplicação de vendas.
- Capítulo 5: contém os trabalhos relacionados com transações em ambientes de computação móvel. Foram selecionados cinco trabalhos: MobileTrans [46], CATE [45], AMT [1], ReflecTS [5] e SGTA [43].
- Capítulo 6: apresenta as conclusões e contribuições dessa dissertação e possíveis trabalhos futuros.

Capítulo 2

Conceitos básicos

Este capítulo introduz os conceitos básicos ao entendimento desta dissertação. Aqui serão apresentadas as definições básicas sobre computação móvel (seção 2.1), transações incluindo controles de concorrência e níveis de isolamento (seção 2.2), reflexão computacional (seção 2.3) e arquitetura baseada em componentes (seção 2.4).

2.1 Computação móvel

Computação móvel é um paradigma que permite que usuários acessem serviços através de pontos diferentes, independentes da sua localização física e mobilidade.

A arquitetura mais comum no ambiente de computação móvel é apresentada na figura 2.1 [33]. Ela é composta por computadores fixos e móveis. Os computadores fixos fazem parte de uma rede fixa de alta velocidade. Os computadores móveis (também chamados de *hosts/*estações móveis) são computadores portáteis que se conectam a uma máquina fixa (estação base) através de uma rede sem fio (podendo ser via satélite, telefonia celular, entre outras).

A comunicação sem fio se faz através de um computador denominado estação base. A estação base está ligada a uma antena, satélite ou outro receptor/transmissor capaz de se comunicar com os computadores móveis. As estações base são conectadas a uma rede fixa com cabos, podendo se comunicar com outros computadores ou servidores. A região de alcance de um receptor sempre é limitada e é chamada "célula". Durante o movimento do computador móvel, pode haver diversas trocas de células. As trocas de célula são conhecidas como "*handoff*". [33, 34].

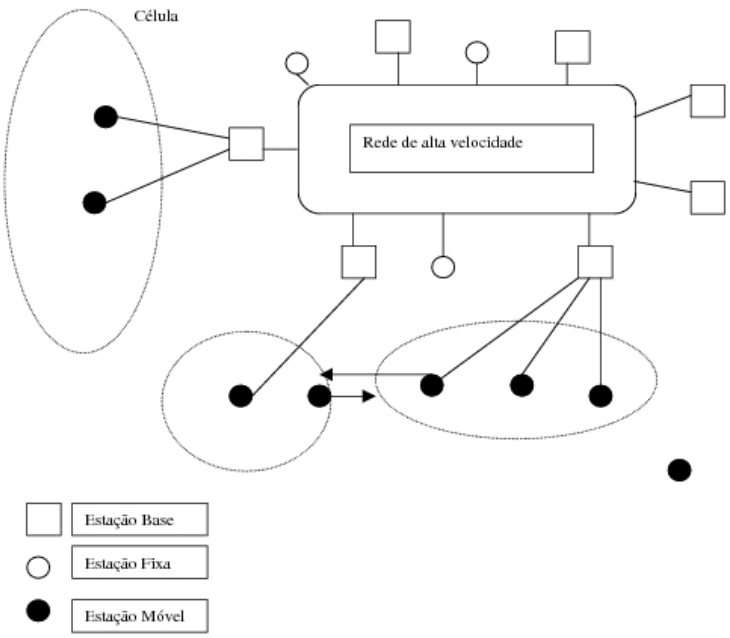


Figura 2.1: Arquitetura da computação móvel

Como a mobilidade é a característica principal da computação móvel, ela ocasiona vários problemas e desafios como os listados a seguir: [33, 34, 43]

- Desconexões: as desconexões são freqüentes neste ambiente e isto requer que os dispositivos móveis continuem a operar enquanto estiverem desconectados.
- Baixa largura de banda: a velocidade de transmissão de dados através de redes sem fio é menor que em redes com fio.

Escassez de energia: os dispositivos móveis utilizam como fonte de energia as baterias que possuem tempo de vida limitado. A transmissão de dados através de meios sem fio é uma operação que consome muita energia.

- Recursos limitados: memória e capacidade de processamento são limitadas.
- Segurança: nas redes sem fio, a segurança é mais vulnerável, pois os dados podem ser decifrados por outros dispositivos caso não haja esquema de autenticação e criptografia.
- Interface com dispositivos móveis: geralmente as telas são pequenas, não existem comumente teclados ou mouse.
- Adaptabilidade: características físicas (largura de banda, memória, energia) e pessoais (informações de localização, preferências pessoais) devem ser consideradas. Por exemplo: algumas aplicações podem desejar mudar características de sua execução para exigir menos de um recurso limitado ou tirar vantagens de um recurso em abundância. Essa é uma das idéias que viabilizam a computação móvel, pois mecanismos de adaptação devem ser projetados para que os recursos disponíveis sejam melhor utilizados.

Para obter mais informações sobre computação móvel e algumas soluções para os obstáculos apresentados, ver referências [15, 19, 22, 27, 33, 34, 37 e 48].

O número de aplicações em dispositivos móveis tem crescido anualmente. Numa pesquisa realizada esse ano pela consultoria comScore [11], o uso de internet móvel cresceu 107% nos Estados Unidos. Sendo assim, os dispositivos móveis executarão cada vez mais aplicações robustas e de diferentes domínios. A tendência agora é mobilidade e isto traz conseqüências como às citadas anteriormente.

Existe um relatório chamado "*Horizon Report*" [26] que identifica anualmente as tecnologias emergentes que devem causar um impacto amplo no ensino e pesquisa nos próximos tempos. Esse relatório é uma publicação de responsabilidade EDUCAUSE [16] (associação americana de mais 2000 universidades e outras organizações educacionais).

No relatório de 2009, um dos itens citados como tendências essenciais são os dispositivos móveis, em particular, os celulares que têm se beneficiado de inovações sem precedentes em função da competição global. Um único dispositivo móvel pode hoje fazer ligações, tirar fotos, gravar áudio e vídeo e interagir com a Internet. Dispositivos móveis mantêm as pessoas em contato com informações e atividades que elas desejam enquanto elas estão em movimento. No relatório são citadas várias aplicações promissoras.

Diante disso, aumenta-se a preocupação em garantir um controle adequado às informações e a adaptação aos recursos limitados. O foco dessa dissertação será no gerenciamento de transações para esse ambiente.

2.2 Transações

O conceito de transação está presente em nosso dia-a-dia. O exemplo mais clássico é a transferência bancária, em que as operações de saque e depósito devem ser realizadas como uma operação única, ou seja, se ocorrer um erro no depósito, o saque deve ser desfeito. Uma transação é definida como um conjunto de operações que deve ser executado completamente (em caso de sucesso), ou cancelado completamente (em caso de falha).

As operações de leituras e escritas (*reads* e *writes*) [38] são controladas através das seguintes primitivas:

- *begin_transaction*: inicia a transação.
- *commit_transaction*: termina a transação efetivando seus resultados.
- *abort_transaction*: cancela a transação descartando seus resultados.

Transações respeitam algumas propriedades conhecidas pela sigla ACID que significa: **A**tomicidade, **C**onsistência, **I**solamento e **D**urabilidade. As quatro propriedades estão descritas abaixo.

- Atomicidade: as operações pertencentes à transação são todas executadas ou nenhuma delas é executada. Se ela é executada até o final, os resultados obtidos são efetivados. Caso o contrário, as alterações executadas até o momento da falha, são desfeitas.
- Consistência: antes do início da transação, as informações devem estar consistentes e após o término da transação devem permanecer consistentes.

- Isolamento: uma transação não visualiza os resultados intermediários de outras transações em execução. Ela deve ser executada sem a interferência de outras transações que eventualmente estejam sendo executadas concorrentemente.
- Durabilidade: após uma transação ter sido efetivada com sucesso, as informações não serão perdidas (os resultados serão permanentes). Nenhuma falha após esta transação poderá desfazer os resultados.

Normalmente, as transações tradicionais são de curta duração e as falhas não são freqüentes. Já no ambiente móvel, as desconexões ocorrem com mais facilidade e as transações são mais longas. Devido a estas características intrínsecas do ambiente, algumas das propriedades ACID podem ser relaxadas [33].

2.2.1 Controles de concorrência pessimista e otimista

É necessário controlar a concorrência para garantir o isolamento entre transações.

As duas principais categorias de controles de concorrência são: pessimista e otimista [7, 12].

O controle pessimista assume que conflitos serão constantes e o ideal é bloquear o acesso aos dados. Já o controle otimista considera que conflitos serão raros e quando eles acontecerem, é possível lidar com eles.

O controle pessimista impede que um dado seja modificado enquanto uma transação o estiver acessando. As transações solicitam o bloqueio[1] para leitura ou escrita antes de acessar um dado e liberam o bloqueio quando este não for mais necessário.

Já no controle otimista, a probabilidade de duas transações conflitarem é baixa. Uma transação prossegue sem limitações de acesso até o seu final. Verifica-se então, os conflitos com outras transações. Se um conflito for detectado, a transação é abortada.

O controle otimista é eficiente quando existem poucos conflitos, pois no caso de abortar, existe a necessidade de repetir todo o trabalho.

Um dos protocolos pessimistas mais utilizados é o bloqueio de duas fases (T*wo-Phase Locking - 2PL*) [7]. A execução da transação consiste de duas fases:

- Fase de crescimento: a transação pode obter bloqueios, mas não pode liberar nenhum bloqueio.
- Fase de encolhimento: a transação pode liberar bloqueios, mas não pode obter nenhum bloqueio.

Os bloqueios são realizados através de trancas de leitura e escrita. A tabela 2.1 mostra a relação de conflitos entre trancas de leitura e escrita.

	Leitura	**Escrita**
Leitura	Sem conflito	Conflito
Escrita	Conflito	Conflito

Tabela 2.1: Conflitos entre trancas

[1] Do inglês lock

Algumas vezes o protocolo *2PL* é confundido com o protocolo de efetivação em duas fases (*Two-Phase Commit – 2PC*). Eles são protocolos que podem ser utilizados em conjunto. Enquanto *2PL* é um protocolo de controle de concorrência, o *2PC* é o protocolo que garante a execução de transações distribuídas.

Já o controle de concorrência otimista utiliza a técnica de validação que possui três fases:

- Fase de leitura: são realizadas cópias locais dos dados originais durante a execução da transação.
- Fase de validação: verifica se a transação X que está terminando obedece a uma das condições abaixo.
    - Outras transações completam sua fase de escrita antes que a transação X inicie sua fase de leitura;
    - A transação X inicia a fase de escrita depois que outras transações concorrentes completam a fase de escrita, e o conjunto de dados lidos pela transação X não tem intersecção com o conjunto de dados escritos por essas outras transações;
    - Os conjuntos de leitura e escrita da transação X não têm itens em comum com o conjunto de escrita de outras transações concorrentes, e essas transações completam sua fase de leitura antes que transação X complete sua fase de leitura.
- Fase de escrita: se a transação for válida (uma das três condições acima ocorrer), um número de versão é atribuído aos dados e as operações são efetivadas no banco de dados.

2.2.2 Níveis de isolamento

Conforme citado anteriormente, o isolamento é uma propriedade transacional que garante que transações concorrentes não serão afetadas por atualizações realizadas por outras transações [23].

Os quatro níveis de isolamento são definidos pelo ANSI SQL-92 [3]:

- Nível 0 (*READ UNCOMMITTED*): uma transação pode ler os dados alterados por outras transações que ainda não foram efetivadas (antes do *commit*). Esse fenômeno é conhecido como leitura inconsistente (*dirty read*).
- Nível 1 (*READ COMMITTED*): uma transação só pode ler os dados alterados por outras transações que foram efetivadas. Não permite leituras de dados não efetivados (*dirty read*), mas permite que leituras subseqüentes retornem valores diferentes. Este fenômeno é conhecido como leitura não repetida (*nonrepeatable read*).
- Nível 2 (*REPEATABLE READ*): uma transação só pode ler os dados alterados por outras transações que foram efetivadas e, além disso, exige que nenhuma outra transação consiga atualizar um dado entre duas leituras feitas por uma transação. Porém, podem ocorrer leituras fantasmas (*phantom reads*). Na leitura fantasma, uma transação lê um conjunto de entradas que satisfaça algum critério

de busca. Outra transação insere uma nova entrada que satisfaça o critério da anterior. Se a primeira transação executar novamente a busca, ela receberá um conjunto diferente de entradas.

- Nível 3 (*SERIALIZABLE*): as operações das transações são executadas de forma concorrente, mas os efeitos de sua execução equivalem à execução dessas transações em forma serial.

A tabela 2.2 resume os fenômenos que podem acontecer quando cada um dos níveis de isolamento é escolhido:

Nível de isolamento	Leituras inconsistentes	Leituras não repetidas	Leituras fantasmas
Read Uncommitted	Sim	Sim	Sim
Read Committed	Não	Sim	Sim
Repeatable Read	Não	Não	Sim
Serializable	Não	Não	Não

Tabela 2.2: Níveis de isolamento

Para entender melhor a diferença entre os níveis, segue um exemplo considerando a tabela 2.3.

Código	Quantidade
1	100
2	50

Tabela 2.3: Estoque de produtos

Na figura 2.2, a transação B alterou a quantidade do produto 1 e antes de a transação B ser efetivada, a transação A já conseguiu ver o novo valor. Porém, a transação B foi cancelada desfazendo essa atualização. Desta forma, a transação A obteve um valor inconsistente (quantidade = 90).

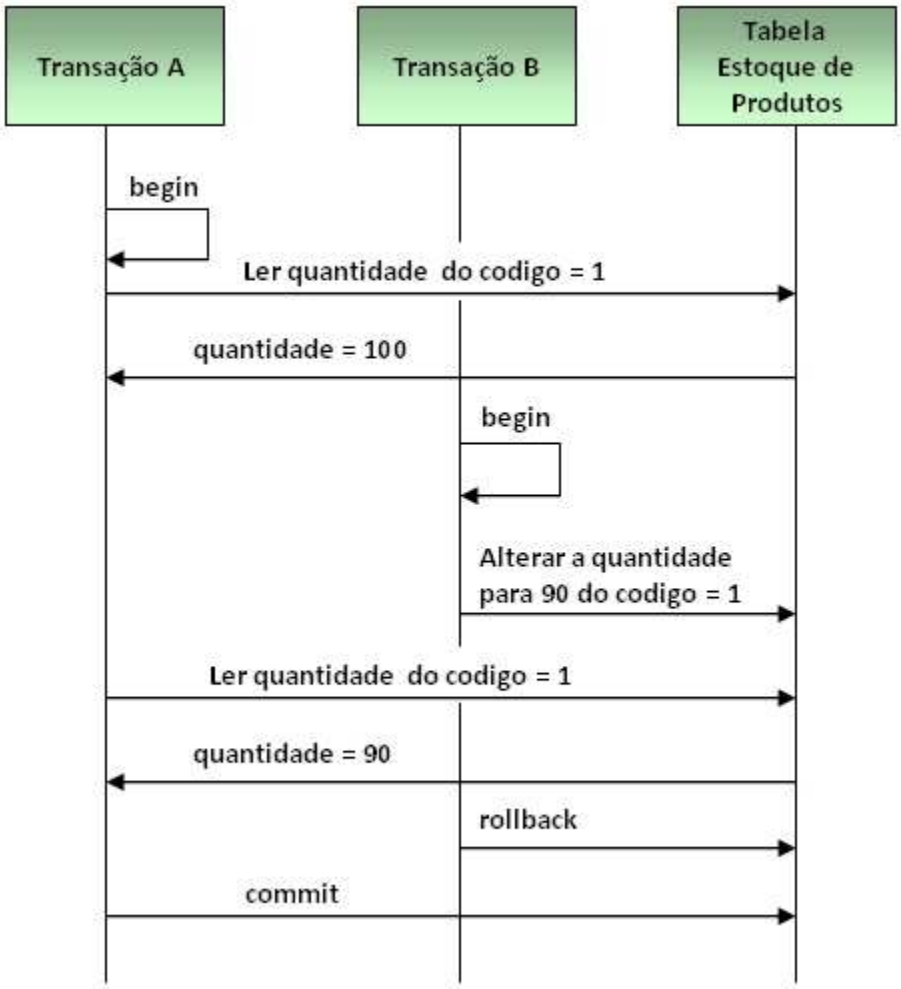


Figura 2.2: Exemplo de nível de isolamento READ UNCOMMITTED

Na figura 2.3, a transação A lê a quantidade do produto 1 mais de uma vez e obtém um estado diferente em cada leitura, porque a transação B alterou a quantidade do produto 1 durante as leituras. As trancas curtas são utilizadas para leitura e trancas longas para escrita. As trancas curtas são adquiridas no início da execução da operação e liberadas logo após o termino desta. Já a tranca de longa duração só é liberada no final da execução da transação.

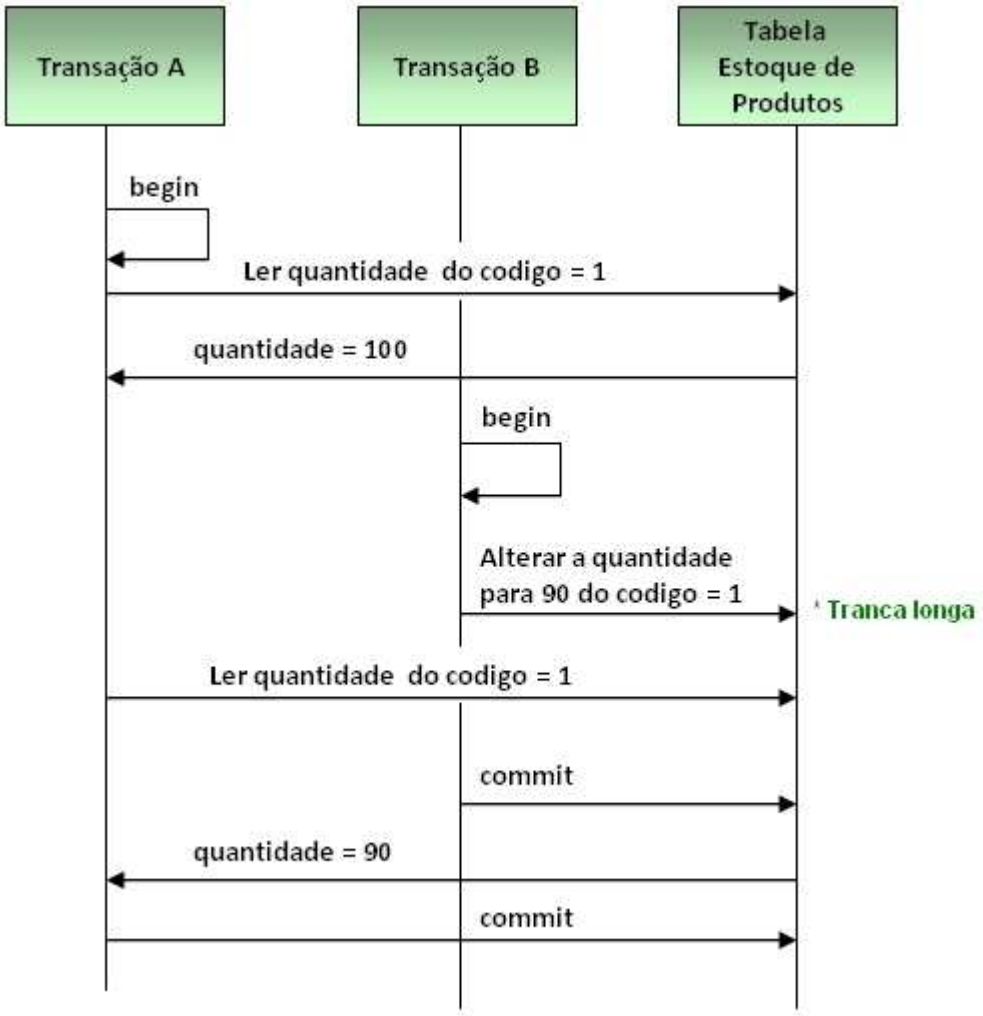


Figura 2.3: Exemplo de nível de isolamento *READ COMMITTED*

Na figura 2.4, a transação A executa uma consulta duas vezes e o segundo resultado incluiu registros que não estavam na primeira consulta, porque a transação B inseriu um novo registro entre as consultas.

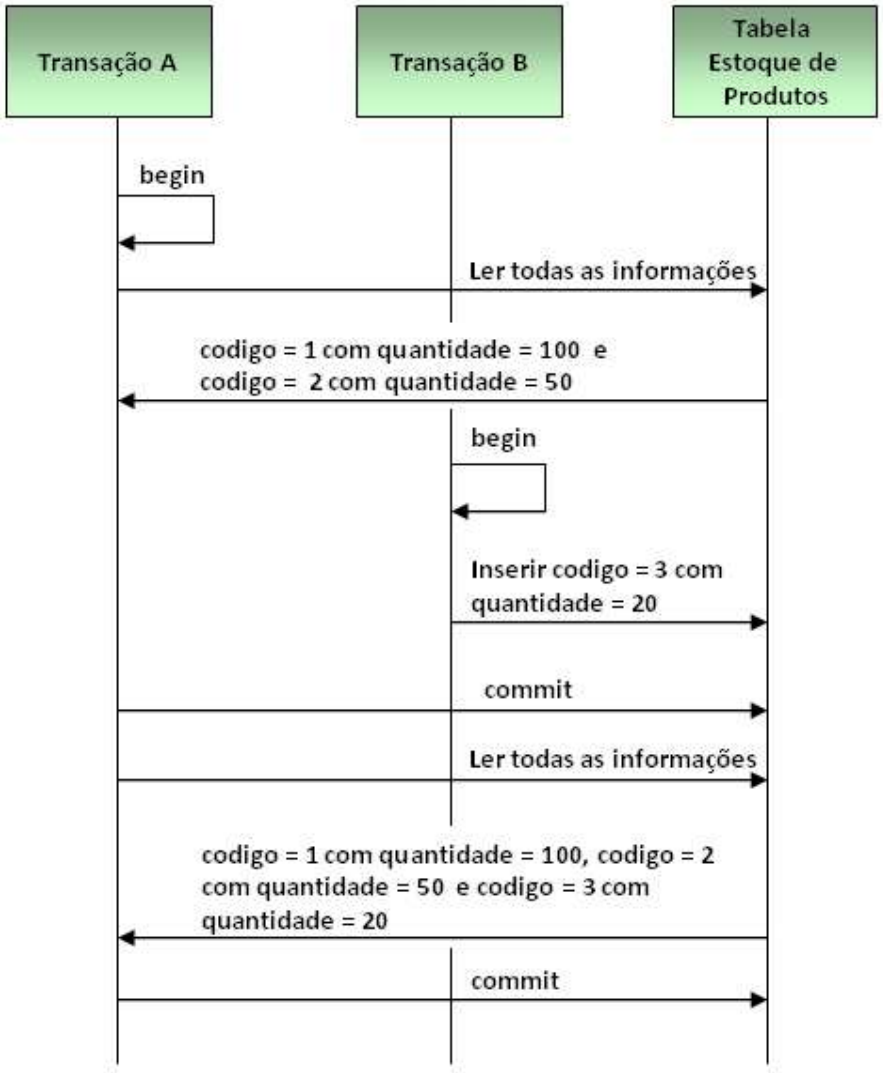


Figura 2. 4: Exemplo de nível de isolamento *REPEATABLE READ*

A escolha do nível de isolamento depende dos requisitos das aplicações em que a aplicação executa. Não existe uma regra que sirva para todas as situações. Isolamento excessivo geralmente não é aceitável, devido ao alto custo (atrasos). Níveis de isolamento muito baixos afetam a consistência da informação. Os níveis de isolamento mais baixos oferecem uma menor garantia de consistência, porém eles dão às transações a vantagem de se envolverem em menos conflitos. Algumas transações não exigem um nível de isolamento tão alto, principalmente se elas são executadas num ambiente de computação móvel. Como nesses ambientes o custo da comunicação é alto e as desconexões são freqüentes, a necessidade de desempenho é maior que a de consistência em muitas situações. Sendo assim, níveis de isolamento mais baixos podem ser considerados.

2.3 Reflexão computacional

Como alguns sistemas computacionais tradicionais permitiam a modificação do seu comportamento apenas através de alteração de código, um novo paradigma surgiu para diminuir a complexidade dessa tarefa. Esse novo paradigma denominado reflexão computacional foi apresentado no início dos anos 80 [49 e 32] e consiste na capacidade de um programa observar ou até mesmo modificar sua estrutura ou comportamento. Um sistema reflexivo é um sistema que incorpora estruturas representando aspectos de si mesmo.

Reflexão implica inspeção e adaptação. O sistema inspeciona o estado corrente e a adaptação faz com que o comportamento do mesmo seja alterado para melhor se ajustar ao ambiente.

A arquitetura usada em um sistema com suporte à reflexão é chamada de arquitetura reflexiva. Consiste em duas partes: um nível base, onde estão os dados e código responsáveis por atender os requisitos para os quais a aplicação foi desenvolvida, e um meta-nível, onde se encontram estruturas e ações que devem ser executadas dinamicamente em resposta às ações do nível base do programa [30].

A reflexão computacional pode ser classificada de acordo com as operações a serem realizadas sobre os objetos do nível base. Ela pode ser classificada em: reflexão estrutural e comportamental [17].

A reflexão estrutural permite obter a informação estrutural do programa, possibilitando inspeção, adição, remoção e modificação de características encapsuladas de entidades no nível base. Já a reflexão comportamental permite a alteração do comportamento do programa, interceptando chamadas e valores devolvidos pelos métodos.

A principal vantagem da reflexão computacional é permitir que o sistema seja alterado em tempo de execução, sendo essas alterações comportamentais ou até mesmo estruturais.

Na computação móvel essa técnica é essencial para prover inspeção e adaptação do sistema diante da variação dos recursos disponíveis. Apesar de poderosa, a reflexão computacional traz algumas preocupações que devem ser consideradas quando essa técnica for escolhida, como por exemplo, o desempenho: um alto nível de flexibilidade exige código a mais no sistema, o que pode afetar o nível de desempenho do mesmo.

2.4 Arquitetura baseada em Componentes

Com o objetivo de suprir a necessidade de desenvolver software de forma mais rápida, surgiu a idéia de dividi-lo em partes menores. Para essa divisão, deu-se o nome de componente de software definido como: uma unidade de composição que pode ser desenvolvida independentemente, com interfaces bem definidas e função específica [51].

Tendo em vista a idéia de reuso, algumas características são importantes na definição de um componente [18]:

- Tamanho, ou seja, a quantidade de funções implementadas.
- Funcionalidades devem ser logicamente relacionadas de forma a compor um conjunto coeso.
- A dependência entre os componentes deve ser pequena.
- A troca de um componente por outro pode ser feita, em algumas situações, sem a necessidade de recompilação [14].

O modelo de componentes se assemelha a um quebra-cabeças. Componentes devem ser encaixados corretamente para que juntos executem a função desejada. Para isso acontecer, há a necessidade de haver um contrato entre eles.

Conforme [6], existe uma diferença entre modelo e framework de componentes. Modelo de componentes é um conjunto de padrões e convenções com as quais os componentes devem estar em conformidade. A infra-estrutura que dá suporte a estes modelos é o framework de componentes. A função do framework é gerenciar os recursos compartilhados pelos componentes e prover um mecanismo que possibilite a comunicação e interação entre eles.

Com isso, o framework e o modelo de componentes podem ser considerados dois conceitos complementares. As definições estabelecidas pelo modelo de componentes devem ser suportadas pelo framework assim como o framework deve respeitar as definições estabelecidas pelo modelo de componentes.

OpenCOM [36] foi o modelo de componentes escolhido para a arquitetura proposta nessa dissertação. Ele é um modelo simples, eficiente e reflexivo projetado para o desenvolvimento de middlewares em dispositivos de computação com limitação de recursos (memória, armazenamento, processamento). Existem outros modelos de componentes como Fractal [20], mas devido à simplicidade, o OpenCOM foi a opção selecionada. Ele reúne as três tecnologias citadas anteriormente: reflexão computacional, componentes e framework de componentes.

O OpenCOM se baseia nas características da tecnologia COM (*Component Object Model*) da Microsoft [9], mas ao contrário dela, as dependências entre os componentes são explícitas, ou seja, pode-se saber em tempo de execução, as dependências entre os componentes. Sem isso, não seria possível implementar adaptação dinâmica.

Os conceitos fundamentais do OpenCOM são interfaces, receptáculos e conexões. Interface é um conjunto de serviços providos pelo componente. Receptáculos são serviços requeridos pelo componente. Uma conexão existe quando um componente provê uma interface para outro componente que depende dela.

Um componente OpenCOM deve estender a classe *OpenCOMComponent* e implementar quatro interfaces: *IUnknown, ILifeCycle, IConnections* (se o componente possuir receptáculos) e *IMetaInterface*.

O OpenCOM possui um ambiente de execução que gerencia a criação e remoção de componentes. As funcionalidades desse ambiente são acessadas através da interface *IOpenCOM*.

- *IUnknown* é equivalente à interface de mesmo nome do modelo COM e é responsável por obter a referência da interface requerida do componente.
- *ILifeCycle* fornece operações denominadas *startup* e *shutdown* que são chamadas quando um componente é criado ou finalizado.
- *IMetaInterface* permite a inspeção das interfaces e receptáculos declarados pelo componente.
- *IConnections* é uma interface opcional que oferece métodos para modificar as interfaces conectadas aos receptáculos do componente.

Como o componente OpenCOM estende essas interfaces, ele deve implementar os métodos declarados por elas como: *startup* e *shutdown* da interface *ILifeCycle*

(responsáveis por controlar o ciclo de vida do componente), *connect* e *disconnect* da interface *IConnections* (responsáveis por conectar e desconectar os componentes).

A reflexão é a técnica utilizada pelo OpenCOM para permitir reconfiguração dinâmica da sua estrutura e comportamento. Ela é fornecida através de três metamodelos distintos: o metamodelo de interface (definido pela interface *IMetaInterface*), o metamodelo de arquitetura (definido pela interface *IMetaArchitecture*) e o metamodelo de comportamento (definido pela interface *IMetaInterception*).

O metamodelo de interface provê acesso à representação externa de um componente OpenCOM no que diz respeito às suas interfaces e receptáculos. Através desse metamodelo é possível: listar todas as interfaces e receptáculos de um componente e recuperar todo o conjunto de meta-informações de uma interface ou receptáculo.

O metamodelo de arquitetura permite acesso a informações de arquitetura Através dele é possível: acessar os identificadores de todas as conexões que envolvem uma dada interface de um componente específico e acessar os identificadores de todas as conexões que envolvem um dado receptáculo de um componente específico.

Já o metamodelo de comportamento permite a alteração no comportamento de operações de uma dada interface de um componente específico.

O OpenCOM permite a criação de framework de componentes. Um framework de componentes é definido no OpenCOM como uma sub-arquitetura de componentes sobre a qual pode ser verificado um conjunto de propriedades arquiteturais desejadas.

O gerenciamento de um framework pode ser realizado através das operações da interface *ICFMetaInterface*. Essas operações podem ser divididas em dois grupos: operações de configuração e operações de inspeção. Entre as principais operações de configuração de um framework estão: a criação de um componente interno e a exposição de interfaces e receptáculos (faz com que a interface/receptáculo de um componente interno passe a ser a interface ou receptáculo do framework).

Capítulo 3

Transações Reconfiguráveis para o Ambiente Móvel

Conforme discutido no capítulo anterior, a computação móvel é a nova tendência e apesar de atrativa, ela traz uma série de obstáculos e preocupações.

Diante do dinamismo do ambiente móvel, as transações representam uma maneira de garantir que as variações ocorridas não comprometam a confiabilidade das aplicações.

Vários modelos de transações foram propostos [41], sendo que alguns deles serão discutidos no Capítulo 5. A maioria desses modelos não provê facilidades que permitam pequenas reconfigurações durante a execução de uma transação.

Para se adequar ao alto dinamismo do ambiente móvel, um modelo de transação deve permitir que propriedades e mecanismos sejam configuráveis antes da execução e reconfiguráveis em tempo de execução. Como geralmente as transações são de longa duração (devido às freqüentes desconexões), a configuração de uma transação realizada no início de sua execução pode deixar de ser adequada no decorrer da sua execução, devido às mudanças no ambiente.

Este capítulo apresenta uma adaptação do modelo de transações SGTA [42 e 43] a ser descrito no capítulo 5, utilizando um modelo de componentes para permitir a reconfiguração de algumas propriedades de uma transação. Esse capítulo será dividido em arquitetura e implementação para um melhor entendimento.

O foco principal dessa dissertação é demonstrar como propriedades transacionais e alguns mecanismos para garanti-las, podem ser configurados e reconfigurados dinamicamente e qual o impacto disso nas aplicações. A propriedade escolhida foi o nível de isolamento e para garantir o isolamento, necessitamos do mecanismo de controle de concorrência. Permitir a configuração do controle de concorrência é algo novo nesse trabalho e permite que cada tipo de controle tire suas vantagens no ambiente móvel.

Reconfigurar é acomodar mudanças sem causar interrupções no provimento de serviços existentes. A reconfiguração dinâmica do apoio transacional não é uma tarefa simples, pois ela pode ser realizada sobre propriedades, estrutura ou comportamento. Porém, realizar grandes mudanças estruturais em tempo de execução, como troca de controle de concorrência, não é muito interessante para as aplicações. As reconfigurações mais prováveis serão de propriedades como, por exemplo, os níveis de isolamento que causam menos mudanças estruturais.

3.1 Arquitetura proposta

A figura 3.1 mostra a arquitetura proposta pelo modelo.

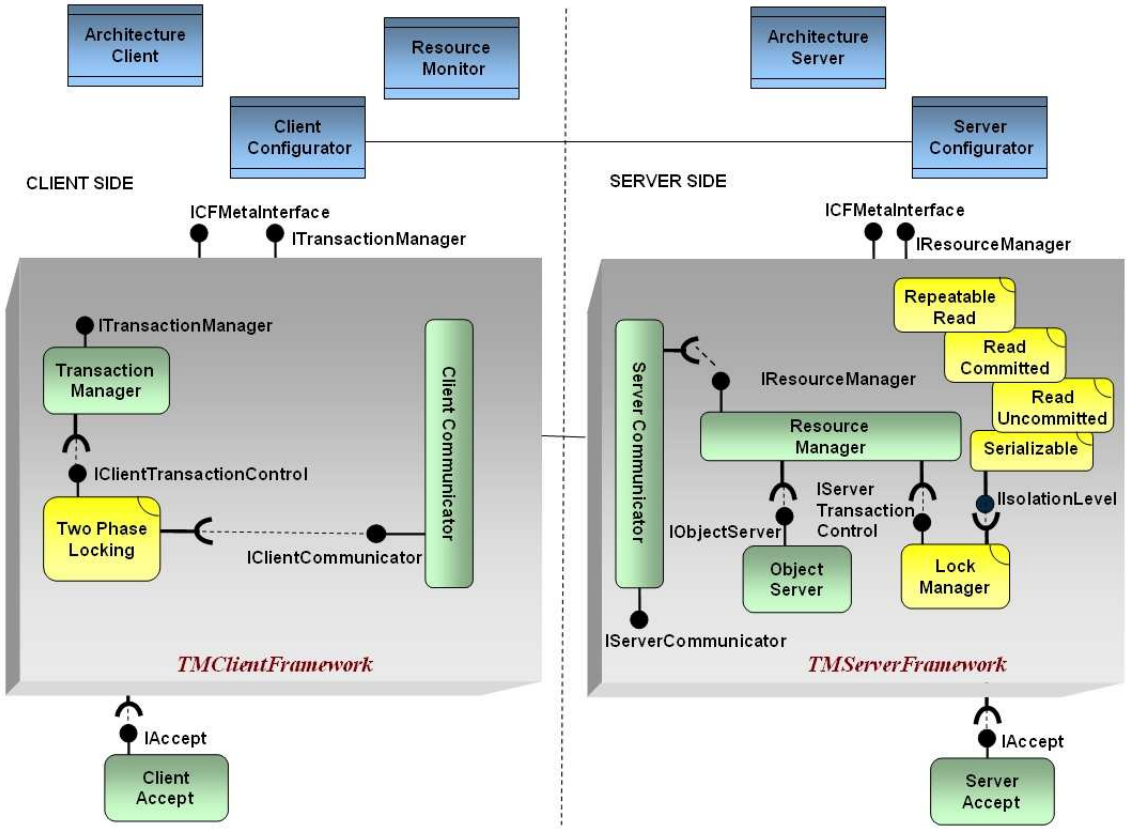


Figura 3.1: Arquitetura proposta – controle pessimista

As convenções utilizadas para representar o desenho arquitetural são descritas a seguir:


representa um framework de componentes que é definido no OpenCOM como uma sub-arquitetura de componentes sobre a qual pode ser verificado um conjunto de propriedades arquiteturais desejadas. Essa sub-arquitetura também pode ser vista como um componente complexo que é composto por outros componentes interconectados.

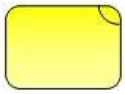
A função do framework de componentes é impor restrições ao processo de configuração e reconfiguração da plataforma de forma que a integridade da mesma possa ser mantida.

 representa uma interface que são serviços expostos. Uma interface estabelece uma espécie de contrato que é obedecido por uma classe. Ela só expõe o que uma classe que a implementa deve fazer. Ela não possui atributos, nem métodos com implementação. Quando uma classe implementa uma interface, ela está comprometida a fornecer o comportamento publicado pela interface.

 representa um receptáculo que são serviços requeridos. Eles podem ser simples ou múltiplos. Um receptáculo simples só aceita a conexão de uma interface por vez. Já o receptáculo múltiplo permite a conexão de mais de uma interface ao mesmo tempo.

 representa uma classe para um conjunto de objetos com características comuns. Ela possui atributos e métodos. É necessário instanciá-la para criar os objetos.

 representa um componente OpenCOM.

 também representa um componente OpenCOM. Esses componentes estão destacados na arquitetura porque podem ser substituídos dinamicamente conforme o tipo de controle de concorrência e nível de isolamento escolhidos.

 representa a conexão entre uma interface e um receptáculo. Para um que componente A utilize serviços de outro componente B, um receptáculo de A precisa ser conectado a uma interface de B.

Na arquitetura proposta foram considerados os controles de concorrência pessimista e otimista explicados no capítulo 2. Os componentes *TwoPhaseLocking*, *LockManager*, *ReadUncommitted, ReadCommitted, Repeatable Read* e *Serializable* fazem parte do controle de concorrência pessimista, conforme demonstrado anteriormente na figura 3.1.

No controle de concorrência otimista, no lado Cliente, o componente "*TwoPhase Locking* é substituído pelo componente *VersionControl*. Já no lado servidor, o componente *LockManager*" é trocado pelo componente *VersionManager* e os componentes *Read Uncommitted, ReadCommitted, Repeatable Read* e *Serializable* não existem, já que o nível de isolamento só é definido para o controle de concorrência pessimista. Os componentes que são alterados para o controle otimista estão destacados na figura 3.2.

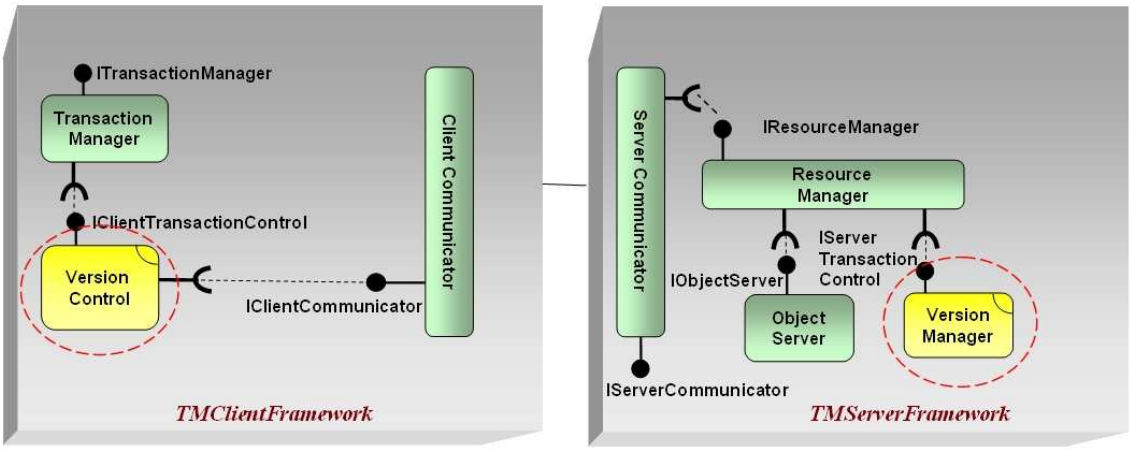


Figura 3.2: Arquitetura proposta – controle otimista

A seguir, cada elemento da arquitetura será detalhado dentro do contexto arquitetural (eles estão listados em ordem alfabética):

- ***ArchitectureClient*** **e** ***ArchitectureServer****:* montam a arquitetura OpenCOM (definindo os controles de concorrência e níveis de isolamento) do lado cliente e servidor, fazendo a ligação entre os componentes. Posteriormente será explicado como as arquiteturas são montadas.

- ***ClientAccept*** e ***ServerAccept:*** componentes que implementam as regras capazes de impor as propriedades arquiteturais desejadas para a arquitetura interna do dos frameworks *TMClientFramework* e *TMServerFramework*. Para evitar que estas configurações e reconfigurações comprometam a integridade da plataforma, cada mudança realizada é comparada a um conjunto de arquiteturas válidas pré-definidas. As operações de inspeção dos frameworks incluem: a recuperação de componentes internos, a recuperação da lista de conexões de um dado componente e a recuperação da lista de interfaces e da lista de receptáculos de componentes internos.

- ***ClientCommunicator*****:** componente responsável pela comunicação com o lado servidor através de mensagens via RMI. Implementa a interface *IClientCommunicator*.

- ***ClientConfigurator*****:** classe que implementa a política de adaptação do lado cliente, como por exemplo, o que acontece com o controle de concorrência e com o nível de isolamento se um determinado recurso do dispositivo móvel está limitado. Ela se comunica com a classe *ServerConfigurator* através da tecnologia RMI para notificar que uma mudança ocorreu na arquitetura do lado cliente e a mesma deve ser realizada no lado servidor.

- ***ServerCommunicator***: responsável pela comunicação com o lado cliente, através de mensagens via RMI. Implementa a interface *IServerCommunicator*.

- ***ServerConfigurator*:** classe que define o controle de concorrência e o nível de isolamento do lado servidor especificados no início da aplicação ou após a notificação da classe *ClientConfigurator*.

- ***TMClientFramework:*** framework OpenCOM do lado cliente.

- ***TMServerFramework:*** framework OpenCOM do lado servidor.

Os componentes e classe que implementam o gerenciamento de transações são descritos a seguir:

- ***ObjectServer*:** componente que representa o repositório de objetos.

- ***ResourceManager***: componente que gerencia o recurso (*ObjectServer*) executando as operações necessárias (inclusão, atualização e consulta).

- ***ResourceMonitor*:** classe responsável por monitorar os recursos do ambiente do dispositivo móvel que possam sofrer variações como por exemplo, bateria e sinal da conexão. Ela também é responsável por notificar as transações das variações do ambiente quando estas solicitam tal monitoramento.

- ***TransactionManager:*** componente que gerencia a transação do lado cliente.

Os componentes relacionados com o controle de concorrência pessimista são descritos a seguir:

- ***LockManager:*** componente pertencente ao controle de concorrência pessimista do lado servidor. Ele mantém uma lista de objetos bloqueados (uma cópia desses objetos) durante uma transação.

- ***ReadCommitted*:** componente que representa o tipo de isolamento *Read Committed*.

- ***ReadUncommitted*:** componente que representa o tipo de isolamento *Read Uncommitted*. É o nível de isolamento menos restrito.

- ***Repeatable Read*:** componente que representa o tipo de isolamento *RepeatableRead*.

- ***Serializable*:** componente que representa um tipo de isolamento *Serializable*. É o nível de isolamento mais restrito.

- ***TwoPhaseLocking:*** componente pertencente ao controle de concorrência pessimista do lado cliente. Implementa o protocolo de bloqueio de 2 fases.

Os componentes relacionados com o controle de concorrência otimista são descritos a seguir:

- ***VersionControl:*** componente que faz parte do controle de concorrência otimista do lado cliente e é responsável por implementar as operações (inclusão, atualização e consulta) sobre os objetos da transação. As alterações são armazenadas em *cache* (memória local de acesso rápido) antes de serem efetivadas.

- ***VersionManager:*** componente que faz parte do controle de concorrência otimista do lado servidor. Ele controla a versão dos objetos utilizados durante uma transação implementando as operações de validação referentes ao controle de concorrência.

3.2 Implementação

Existem algumas classes que não aparecem na figura da arquitetura apresentada, pois elas são parte da implementação do estudo de caso a ser demonstrado no capítulo 4. Elas estão descritas abaixo:

- ***ClientApplication*****:** classe que implementa a aplicação que utilizará o controle de transações.

- ***FileXML:*** classe que lê os dados referentes aos recursos do sistema (como bateria e nível do sinal de conexão) e armazena num objeto.

- ***IRMICommunication:*** interface remota.

- ***Resource:*** classe que representa um recurso do sistema a ser medido (bateria, nível do sinal de conexão, largura de banda, etc).

- ***Resources.xml:*** arquivo XML [54] que simula os recursos do sistema que podem ser medidos como por exemplo, bateria, nível do sinal de conexão, largura de banda, entre outros.

- ***Sales:*** classe que representa uma tabela de vendas, contendo o código do produto e a quantidade vendida.

- ***Stock:*** classe que representa uma tabela de estoque, contendo o código do produto, a quantidade em estoque e o custo unitário.

- ***ThreadClass:*** classe utilizada pela classe *FileXML* para permitir que o arquivo XML seja sempre lido em tempo de execução.

- ***TransInfo:*** classe que armazena os dados de uma transação (identificador da transação e objetos pertencentes a essa transação). Utilizada pelo componente *ResourceManager*.

A comunicação entre o cliente e servidor é feita via RMI (*Remote Method Invocation)* [40]. RMI é uma tecnologia Java que possibilita a invocação de um método remoto como se fosse um método local possibilitando a comunicação entre objetos em diferentes máquinas de forma transparente. Os serviços providos por objetos remotos são descritos por interfaces e o modelo de execução é cliente-servidor. Java RMI libera o programador de tratar de detalhes como endereçamento e codificação e decodificação de mensagens.

Na tecnologia RMI:

- O servidor define os objetos que o cliente pode usar remotamente. Os objetos devem ser registrados num servidor de nomes.
- O mecanismo subjacente de transporte via rede fica escondido dos argumentos dos métodos e valores de retorno.
- O cliente pode invocar métodos desse objeto remoto como se ele estivesse executando localmente. Para isso, ele deve consultar o servidor de nomes pela referência do objeto remoto.

A arquitetura simplificada da RMI é exibida na figura 3.3.

Quando dois objetos em espaços de endereçamento diferentes se comunicam, é necessário existir intermediários que realizem essa comunicação, são eles o s*tub* e sk*eleton*. O cliente não conhece a implementação do objeto no servidor, apenas a interface remota. A interface remota define quais métodos podem ser chamados remotamente. A função do *stub* é, portanto, encapsular uma série de operações necessárias ao acesso a objetos remotos, como iniciar conexão com a máquina remota, preparar os dados que serão transmitidos, aguardar retorno, enviar resultados. Já no lado do servidor, a forma de tratar as chamadas remotas é utilizando um *skeleton*. Este *skeleton,* assim como o *stub,* é responsável por encapsular uma série de atividades necessárias para a decodificação, execução e o envio dos resultados de uma chamada.

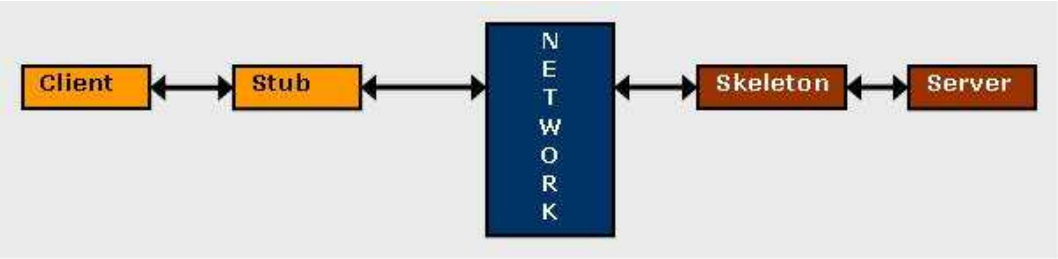


Figura 3.3: A arquitetura RMI

3.3 Montagem das Arquiteturas Cliente e Servidor

Conforme mencionado, as arquiteturas cliente e servidor são montadas respectivamente pelos componentes *ArchitectureClient* e *ArchitectureServer*. Os passos para montar a arquitetura serão explicados abaixo.

Lado servidor:

Na arquitetura utilizando o modelo de componentes OpenCOM, o primeiro componente criado é o framework denominado *TMServerFramework*.

O próximo objeto criado é o *ServerAccept* que conterá as definições arquiteturais. Existindo os dois objetos criados no ambiente OpenCOM, é criada uma conexão entre eles – ligando o receptáculo do componente *TMServerFramework* com a interface *IAccept* implementada pelo componente *ServerAccept*.

A metainterface *ICFMetaInterface* é definida para o *TMServerFramework*. Através dela pode-se gerenciar as funcionalidades do framework.

A partir desse ponto, os componentes dentro do framework são criados. Eles devem estar dentro do escopo de uma transação arquitetural do OpenCOM (no final dessa transação, a arquitetura é validada para verificar se os componentes criados e conectados dentro do framework seguem as regras definidas no componente *ServerAccept*). Essa transação pertence ao OpenCOM e inicia e termina no momento de criação da arquitetura. Não é a transação da aplicação.

Os componentes internos do *TMServerFramework* criados são: *ObjectServer*, *ResourceManager* e *ServerCommunicator*. Conforme a figura 3.1 mostrada anteriormente, o componente *ResourceManager* possui dois receptáculos simples, sendo o primeiro deles conectado à interface *IObjectServer* do componente *ObjectServer*. O receptáculo do componente *ServerCommunicator* é conectado à interface *IResourceManager* implementada pelo componente *ResourceManager*.

No caso do controle de concorrência ser pessimista, os componentes *LockManager*, *Serializable, Read Committed, Repeatable Read* e *ReadUncommitted* são criados.

O segundo receptáculo do componente *ResourceManager* é conectado à interface *IServerTransactionControl* implementada pelo componente *LockManager*. O receptáculo do componente *LockManager* é conectado à interface *IIsolationLevel* implementada pelos componentes *Serializable, Read Committed, Repeatable Read* e *ReadUncommitted*.

Já no controle de concorrência otimista, apenas o componente *VersionManager* é criado. O segundo receptáculo do componente *ResourceManager* é conectado à interface *IServerTransactionControl* implementada pelo componente *VersionManager*.

Para que os métodos expostos pelo framework do servidor possam ser utilizados remotamente pelo lado Cliente, o componente *ServerCommunicator* dever ser registrado no servidor de nomes do RMI e exportado.

Lado cliente:

Assim como no lado servidor, o primeiro componente criado é o framework denominado *TMClientFramework*. O próximo objeto criado é o *ClientAccept* que conterá as definições arquiteturais. Assim como no lado servidor, existindo esses dois objetos criados no ambiente OpenCOM, é criada uma conexão entre eles – ligando o receptáculo do componente *TMClientFramework* com a interface *IAccept* implementada pelo componente *ClientAccept*.

A partir desse ponto, os componentes dentro do framework são criados. Os componentes internos do *TMClientFramework* criados são: *TransactionManager* e *ClientCommunicator*. No controle de concorrência pessimista, o componente *TwoPhaseLocking* é criado. O receptáculo do componente *TransactionManager* conecta à interface *IClienTransactionControl* implementada pelo componente *TwoPhaseLocking* e o receptáculo deste componente se conecta à interface *IClientCommunicator* implementada pelo componente *ClientCommunicator*.

A interface *ITransactionManager* será exposta para fora do framework para que ela seja utilizada pelo *ClientApplication*.

Já no controle de concorrência otimista, o componente *VersionControl* é criado e conectado ao componente *TransactionManager*

3.4 Reconfiguração dos Componentes

A reconfiguração da transação é feita através da troca de componentes: os componentes necessários são conectados e desconectados, não necessitando remontar toda a arquitetura definida.

Como o foco do modelo é o controle de concorrência, a idéia é que o modelo permita a configuração do controle de concorrência e a reconfiguração do nível de isolamento baseado em políticas de adaptação pré-definidas.

O controle de concorrência é configurado no início da transação conforme a situação dos recursos disponíveis no ambiente móvel, como, por exemplo, bateria, largura de banda ou conectividade.

O controle de concorrência definido como padrão é o pessimista com nível de isolamento *Serializable*. Se a situação de um recurso for alterada extrapolando alguns limites pré-estabelecidos pela transação, algumas propriedades podem mudar se: a transação estiver no início, o controle de concorrência poderá ser alterado, mas se ela estiver em execução, apenas o nível de isolamento será alterado.

A transação reage às mudanças do ambiente conforme a política de adaptação definida para ela. A lógica dessa política é definida pelo programador. Ela é escrita utilizando uma linguagem de programação e contém as regras que deverão ser seguidas caso o monitor de recursos verifique que os valores pré-estabelecidos pela aplicação foram extrapolados.

A reconfiguração é feita pelas classes *ArchitectureServer* e *ArchitectureClient*. O monitor de recursos verifica os recursos disponíveis no ambiente do dispositivo móvel cujo monitoramento foi requisitado pela aplicação. Se o limite for atingido, ele notifica a classe *ClientConfigurator* que contém a política de adaptação do lado cliente.

Se uma das regras definidas na política de adaptação for cumprida, a arquitetura deverá ser alterada, portanto, a classe *ArchitectureClient* será chamada para realizar a adaptação dinâmica. A reconfiguração é realizada desconectando e conectando componentes.

Como o cliente e o servidor devem estar sincronizados, ou seja, com o mesmo tipo de controle de concorrência, o cliente notifica a classe *ServerConfigurator* indicando que a sua arquitetura também deve ser alterada já que a arquitetura do cliente foi mudada. O *ServerConfigurator* contém a política de adaptação do lado servidor e caso alguma regra

inserida nele seja cumprida, a classe *ArchitectureServer* é chamada para também fazer a reconfiguração do lado servidor.

A arquitetura apresentada na figura 3.1 contém alguns níveis de isolamento, porém novos níveis de isolamento poderiam ser adicionados. Para isso, seria necessário alteração nas classes *ArchitectureClient* e *ArchitectureServer* para incluir os novos componentes e nas classes *ClientConfigurator* e *ServerConfigurator* para estabelecer as regras indicando sob quais condições esses novos componentes seriam utilizados. Além disso, as classes *ClientAccept* e *ServerAccept* deveriam atualizar as regras arquiteturais.

Pode-se questionar a necessidade da alteração dessas classes cada vez que um novo componente for adicionado à arquitetura. As classes *ArchitectureServer* e *ArchitectureClient* criam e conectam os componentes utilizando o modelo OpenCOM. Uma vez definidas, essas classes não precisarão ser alteradas a cada execução. Porém, o modelo OpenCOM trabalha dessa forma, é através da programação que os componentes são conectados.

Já as classes *ClientConfigurator* e *ServerConfigurator* contêm as políticas de adaptação que devem seguir as regras do domínio da aplicação. Como forma de oferecer um melhor desempenho, as regras são definidas através de programação, porém, poderiam ser definidas através de arquivos XML ou interface gráfica (o que evitaria a recompilação do código).

De qualquer forma, as regras para alterar o controle de concorrência e o nível de isolamento, assim como outras possíveis propriedades transacionais, estão relacionadas ao tipo de aplicação. As propriedades transacionais requeridas pelas aplicações podem ser tão diferentes que é impossível atender todas as possíveis situações exigidas pelas aplicações. Porém, o assunto muda um pouco de foco: de controle de transações reconfiguráveis surge a preocupação com a variedade de requisitos transacionais exigidos por aplicações de diferentes domínios. A partir desse ponto, outros itens são questionados, como a necessidade de uma camada extra mais genérica que exigiria mais recursos (não abundantes nos dispositivos móveis).

Portanto, o foco dessa dissertação é a configuração do controle de concorrência e a reconfiguração do nível de isolamento conforme as regras estabelecidas pela própria aplicação que determina os limites dos recursos do ambiente móvel.

Capítulo 4

Protótipo e Estudo de Casos

Um protótipo foi desenvolvido para comprovar a viabilidade da arquitetura proposta. Ele foi desenvolvido utilizando a versão Java do OpenCOM [52] - versão 1.3.5 e a linguagem Java [24]. O motivo da escolha de Java é por ela ser uma linguagem orientada a objetos e independente de plataforma.

Existe uma versão de Java para aplicações móveis que é chamada de Java ME (*Java Micro Edition*) [25]. Porém, ela não foi utilizada porque não existe uma versão do OpenCOM para Java ME. Até existe uma versão de teste, mas não uma versão oficial disponibilizada para uso [13].

O protótipo evidencia a configuração dos controles de concorrência e a reconfiguração dos níveis de isolamento. Como se trata de um protótipo e não de um sistema completo, apenas dois níveis de isolamento foram implementados: *Serializable* (mais restrito) e *Read Uncommitted* (menos restrito).

Para que o entendimento fosse completo, um estudo de caso na área de vendas foi escolhido.

4.1 Estudo de Caso: Vendas

Nesta seção será apresentado um exemplo de aplicação utilizando a arquitetura proposta no capítulo 3.

Considere uma aplicação de vendas destinada aos gerentes e representantes de vendas de uma distribuidora de medicamentos. Ambos utilizam um PDA (*Personal Digital Assistant*) para acessarem os módulos da aplicação e realizarem as tarefas de sua função. PDA é conhecido como computador de bolso, por ter dimensões reduzidas.

A aplicação possui dois módulos: Pedido de venda de produtos e Relatório da média das vendas do dia.

O representante de vendas efetua o pedido de venda do produto através de um PDA. Ele não necessita utilizar papéis, economizando dinheiro e tempo. Ele comunica-se diretamente com o servidor em tempo real, fazendo seus pedidos através do computador, ganhando dessa forma, agilidade. O pedido do produto, após a venda efetivada, é enviado para o serviço de entrega, que já inicia o seu trabalho. Enquanto a maioria dos representantes está chegando em casa ou na empresa para finalizar seus pedidos, esse representante está descansando.

O gerente consulta a média das vendas realizadas no dia através de um PDA também. Para tomar alguma decisão com maior rapidez, ele precisa ser informado sobre as vendas realizadas no dia e o desempenho de sua equipe de vendedores.

Os dois módulos se conectam a um servidor para efetuar as vendas dos produtos e realizar as consultas necessárias.

Existem vários representantes de vendas em lugares diferentes que utilizarão a aplicação ao mesmo tempo.

Como esses usuários estão utilizando dispositivos móveis, eles encontram restrições de alguns recursos como, por exemplo, bateria e conectividade. Sendo assim, em situações de escassez de energia ou nível de bateria limitado, o gerente necessita visualizar a média das vendas considerando até os produtos que já foram registrados pelos representantes, mas cuja venda ainda não foi concluída. Essa informação o ajudará a tomar decisões como criar promoções e agilizar a entrega de alguns produtos, aumentando a margem de lucro e a satisfação do cliente.

Para esse tipo de aplicação em que existe concorrência, é necessário haver um controle de transações.

O protótipo dessa aplicação de Vendas utilizará o modelo de transações exposto no capítulo 3. Três casos de uso serão demonstrados passo a passo para explicar o funcionamento da arquitetura, a configuração dos controles de concorrência no início da transação e a reconfiguração dos níveis de isolamento durante a transação.

4.1.1 Caso de Uso: Controle de concorrência pessimista

A aplicação de vendas contendo os dois módulos (Pedido de venda de produtos e Relatório de média das vendas do dia) representa o lado Cliente na arquitetura proposta. O lado servidor contém o banco de dados com as informações dos produtos e das vendas.

Para um melhor entendimento da implementação, será mostrada, primeiramente a arquitetura OpenCOM criada e posteriormente a aplicação desenvolvida. Algumas informações sobre a arquitetura que serão detalhadas no decorrer do caso de uso não são

relevantes para o usuário final, porém ajudam no entendimento da implementação do protótipo do estudo de caso.

O servidor aciona a classe *ArchitectureServer* para montar a arquitetura com os componentes OpenCOM do lado servidor.

A figura 4.1 mostra a visão geral da arquitetura montada no lado servidor, contendo os componentes *TMServerFramework* e *ServerAccept*.

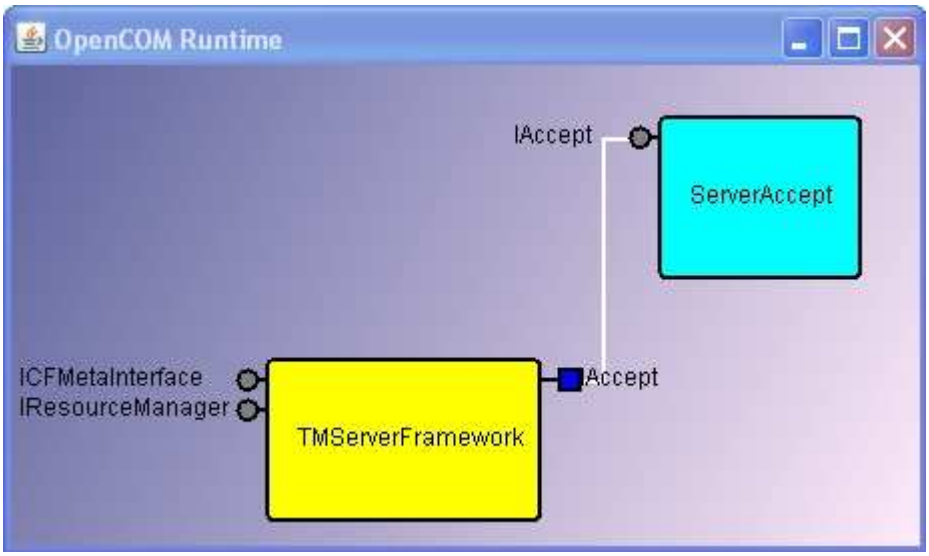


Figura 4.1: Arquitetura do lado servidor

A figura 4.2 expande o componente *TMServerFramework*. Como o controle de concorrência definido como padrão é o pessimista, o componente *LockManager* é conectado ao componente *ResourceManager*. Os componentes referentes ao nível de isolamento são conectados ao *ResourceManager* também. O nível de isolamento padrão é o *Serializable***.** O outro nível isolamento (*ReadUncommitted*) permanece desconectado da arquitetura.

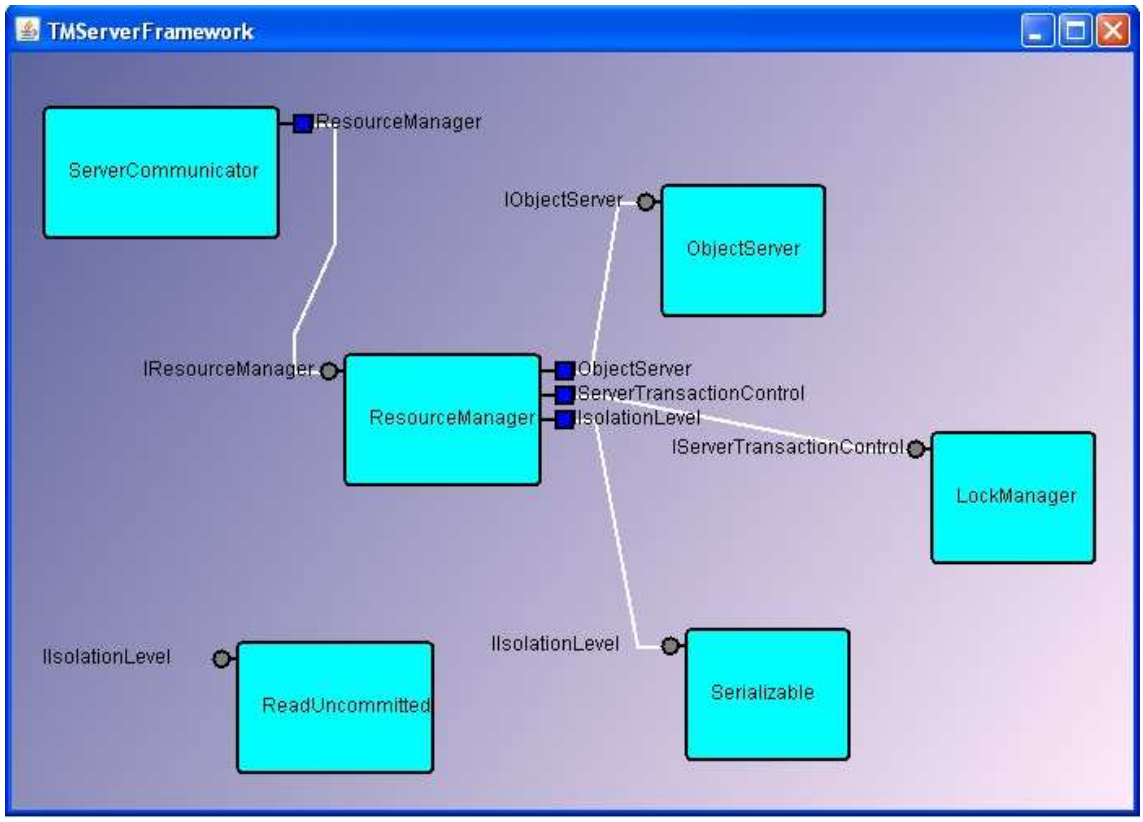


Figura 4.2: Framework do lado servidor

O componente que gerencia os recursos do sistema (*ResourceManager)* exibe, conforme a figura 4.3, o recurso gerenciado (*ObjetServer*), que contém os objetos estoque e vendas chamados respectivamente de *Stock* e *Sales*, e os objetos bloqueados (gerenciados pelo componente *LockManager*).

O estoque contém dois produtos:

- O produto com código igual a 1, quantidade igual a 200 unidades e custo de 10 dólares a unidade.
- O produto com código igual a 2, 100 unidades em estoque e custo unitário de 100 dólares.

Figura 4.3: Gerenciador de Recursos

O servidor está pronto aguardando a conexão de algum cliente.

Um representante de vendas chamado José inicia o seu dia de trabalho. Ele está nas ruas, utilizando seu PDA. Ele representa o cliente da aplicação. Internamente, o cliente aciona a classe *ArchitectureClient* para montar a arquitetura com os componentes OpenCOM do lado cliente.

A figura 4.4 mostra a visão geral da arquitetura montada no lado cliente, contendo os componentes *TMClientFramework* e *ClientAccept*. Note que a interface *ITransactionManager* foi exposta pelo framework.

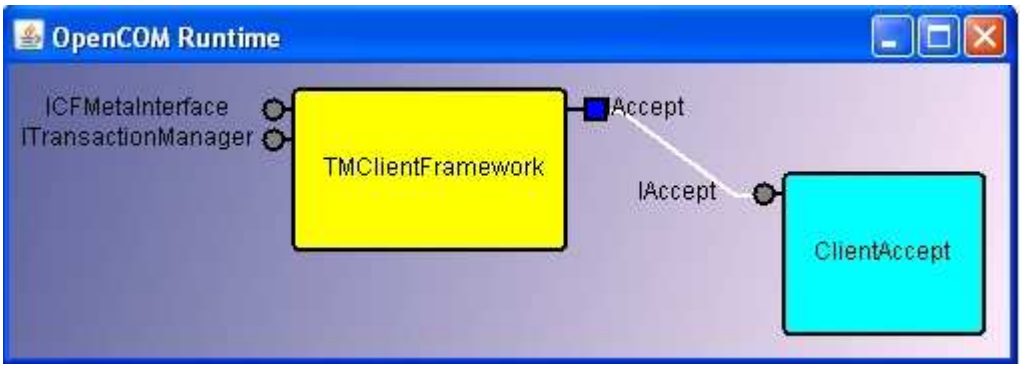


Figura 4.4: Arquitetura do lado cliente

A figura 4.5 expande o componente *TMClientFramework*. Como o controle de concorrência é o pessimista, o componente *TwoPhaseLocking* é conectado ao componente *TransactionManager*. O tipo de controle de concorrência é exibido na figura 4.6.

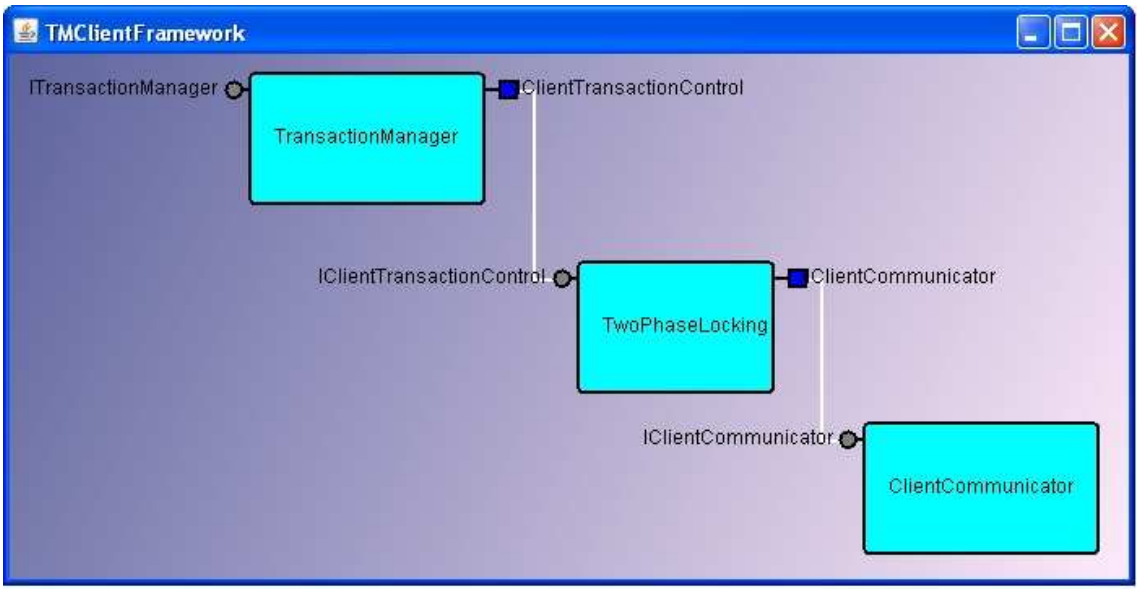


Figura 4.5: Framework do lado cliente

Figura 4.6: Tipo de controle de concorrência

A seguir será mostrada a aplicação do usuário final. O representante de vendas visita uma farmácia, verifica os medicamentos que necessitam ser solicitados à distribuidora e começa a registrar a solicitação na aplicação de vendas em seu PDA. A aplicação é exibida na figura 4.7.

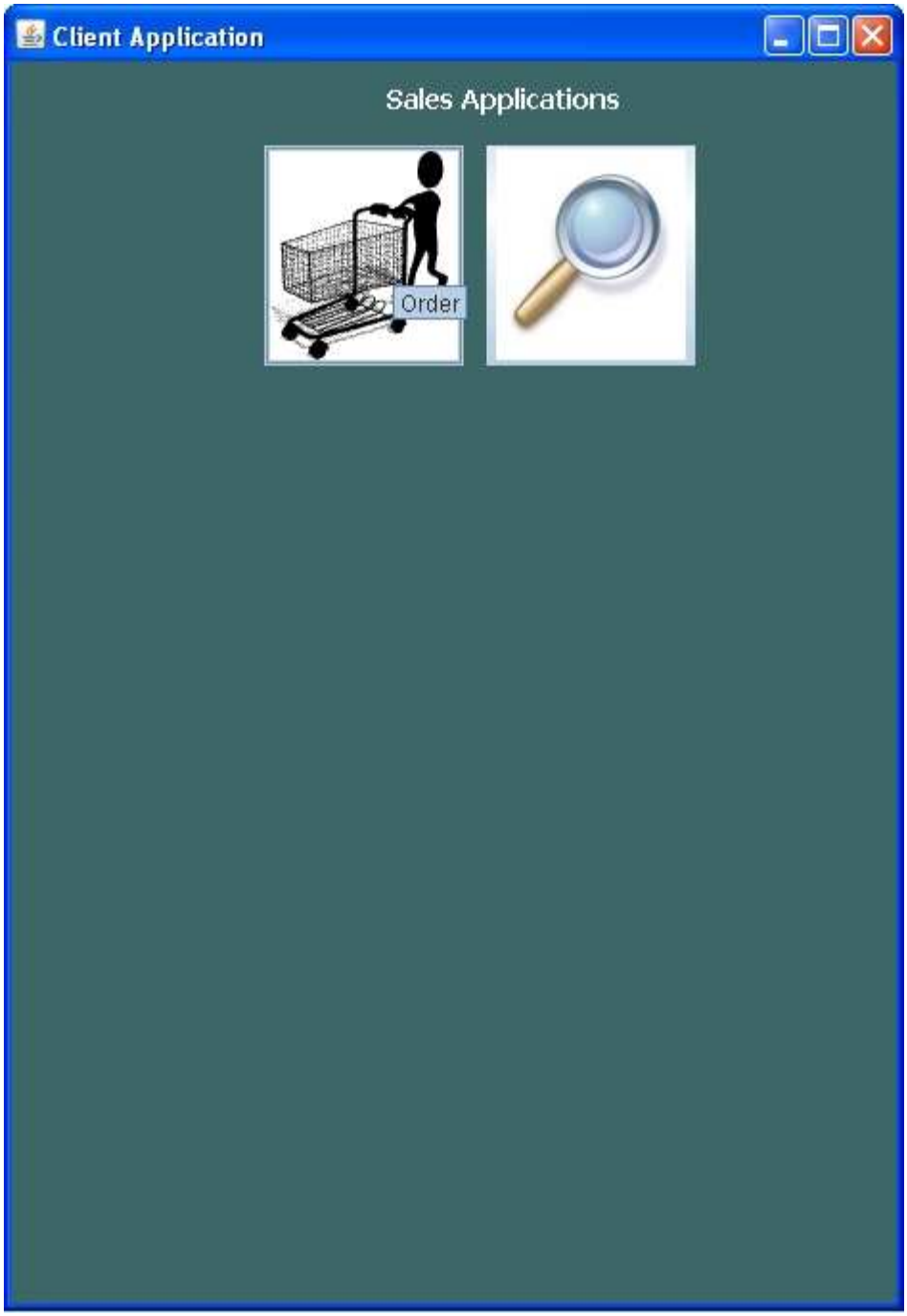


Figura 4.7: Aplicação cliente

O representante de vendas seleciona a opção "*Order*". Em seguida, ele escolhe o produto "p1", digita quantidade igual a 2 e seleciona a opção "*Confirm*", conforme a figura 4.8.

Figura 4.8: Produto e quantidade escolhidos pelo representante de vendas José

Para verificar como o protótipo implementa a arquitetura proposta, será mostrado passo a passo a montagem da arquitetura após o representante de vendas selecionar a opção "*Confirm*". As mensagens da figura 4.9 não serão exibidas ao representante de vendas. Conforme mostrado anteriormente, o controle de concorrência configurado nesse momento é o pessimista.

Como a interface exposta pelo framework do cliente é a *ITransactionManager*, todas as operações pertencentes a ela podem ser utilizadas pelo cliente da aplicação. Seguindo o modelo de transação, as operações chamadas são: *begin, execute*, *commit* e *rollback*.

A primeira operação chamada do componente *TransactionManager* é *begin* que indica o início da transação. Seguindo a arquitetura, a operação chama os componentes

TwoPhaseLocking e *ClientCommunicator***.** No *ClientCommunicator*, o servidor é procurado no servidor de nomes RMI, realizando dessa forma a conexão cliente-servidor.

Do lado servidor, o componente *ServerCommunicator* recebe a requisição do cliente e chama o componente *ResourceManager*. Na operação de *begin* dentro desse componente, a transação é registrada no componente *TransInfo* e uma cópia dos objetos a serem utilizados pela transação (*Sales* e *Stock*) é inserida na lista de objetos bloqueados. O responsável por esse controle é o componente *LockManager*.

A próxima operação da transação é o *execute*. Nesse exemplo, duas operações de execução são realizadas: inserir o pedido solicitado na tabela de vendas (representada pelo objeto *Sales*) e atualizar a tabela de estoque (representada pelo objeto *Stock*).

As duas operações são executadas de forma atômica, se o produto for inserido na tabela de vendas, a tabela de estoque deve ser atualizada. Se uma das operações falhar, a outra deve ser desfeita.

Na execução das operações, o componente *ResourceManager* pede para o componente *LockManager* verificar se o objeto está bloqueado por outra transação. Caso exista outra transação utilizando o mesmo objeto, a transação corrente não pode continuar (isso será mostrado posteriormente, quando existir outro representante de vendas tentando solicitar o mesmo produto). Outra possibilidade seria bloquear a transação corrente, mas só a primeira opção foi implementada. Em seguida, se a requisição de bloqueio teve sucesso, o componente inclui o pedido na tabela de vendas e atualiza o estoque. O sistema verifica se existe o produto no estoque antes da realização da venda, pois mesmo que a lista de produtos exibidos na aplicação cliente exiba apenas os produtos disponíveis no estoque, até o pedido chegar ao servidor, o estoque pode ter sido alterado.

Para efetivar a transação, a operação *commit* precisa ser executada. Porém, antes de a transação do representante de vendas José ser efetivada, outro representante de vendas chamado Manuel estava executando, ao mesmo tempo, a aplicação em outra farmácia. Ele solicitou o mesmo produto "p1" com quantidade igual a 3. Manuel não consegue realizar a compra do produto porque a transação do representante de vendas José está bloqueando o objeto *Sales*. Como o controle de concorrência é pessimista, a transação do representante de vendas Manuel segue apenas até a primeira operação de inserir o pedido na tabela de vendas. A mensagem de bloqueio é exibida conforme a figura 4.10.

Figura 4.9: Mensagens durante a transação do representante José

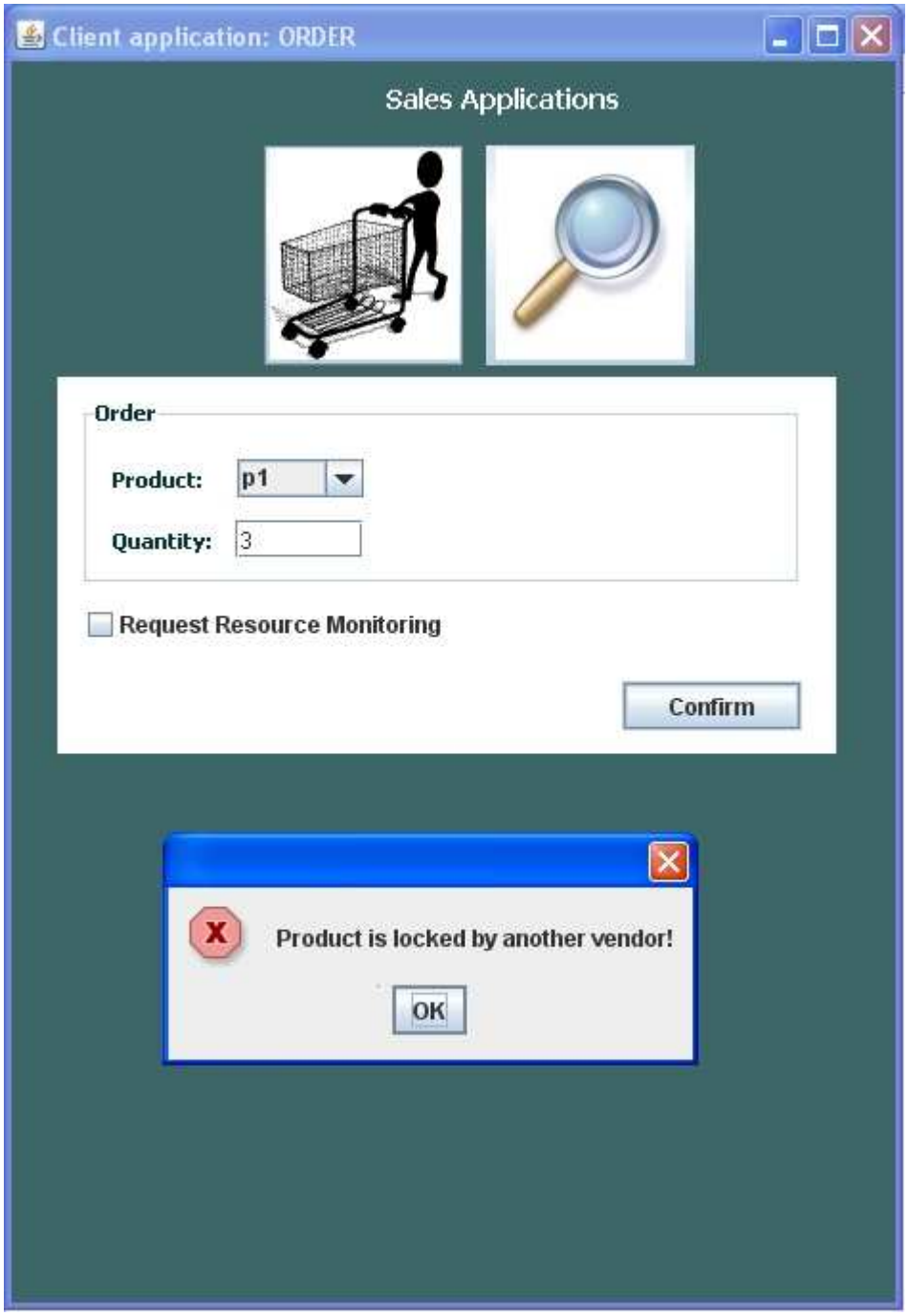


Figura 4.10: Objeto bloqueado

O representante de vendas José decide cancelar o pedido do produto “p1” porque a farmacêutica descobriu que havia ainda dois medicamentos “p1” no estoque, porém eles estavam armazenados no lugar incorreto. José seleciona a opção “*Cancel*” e com isso a operação *rollback* é chamada. As operações de inserção do produto na tabela de vendas e a atualização da quantidade na tabela de estoque são desfeitas.

Portanto, nenhum produto foi vendido. O representante de vendas Manuel poderia ter efetivado a sua compra, mas devido ao controle de concorrência ser pessimista, ele perdeu a venda.

Diante disso, nunca deveria ser utilizado o controle de concorrência pessimista? A resposta seria: depende. O controle de concorrência de pessimista é necessário quando existe a grande probabilidade de conflitos e no ambiente de computação móvel, quando os recursos como conexão e bateria não estão limitados. Mas como saber se os recursos estão limitados? Para isso, a arquitetura contém o componente *ResourceMonitor.* Na figura 4.8, o representante de vendas poderia optar por monitorar os recursos do dispositivo móvel, como por exemplo, o sinal da conexão com o servidor.

Como os representantes usam dispositivos móveis, o sinal de conexão poderia estar ruim e a conexão com o servidor cair. Sendo assim, os objetos ficariam bloqueados até a conexão voltar, impedindo novas vendas de um determinado produto. Diante desse contexto, seria apropriado utilizar o controle de concorrência otimista conforme o próximo caso de uso.

Abaixo serão exibidos alguns diagramas de sequência para ajudar a compressão da arquitetura sob o ponto de vista de programação.

As figuras 4.11 e 4.12 mostram os principais componentes e métodos utilizados no controle de concorrência pessimista. O método *execute* é chamado duas vezes, uma vez para executar a operação de inserir na tabela de vendas e a outra vez para executar a operação de atualização na tabela de estoque.

Por uma questão de simplificação, todos os objetos a serem utilizados pela transação são bloqueados no início da transação, não durante a execução de cada operação.

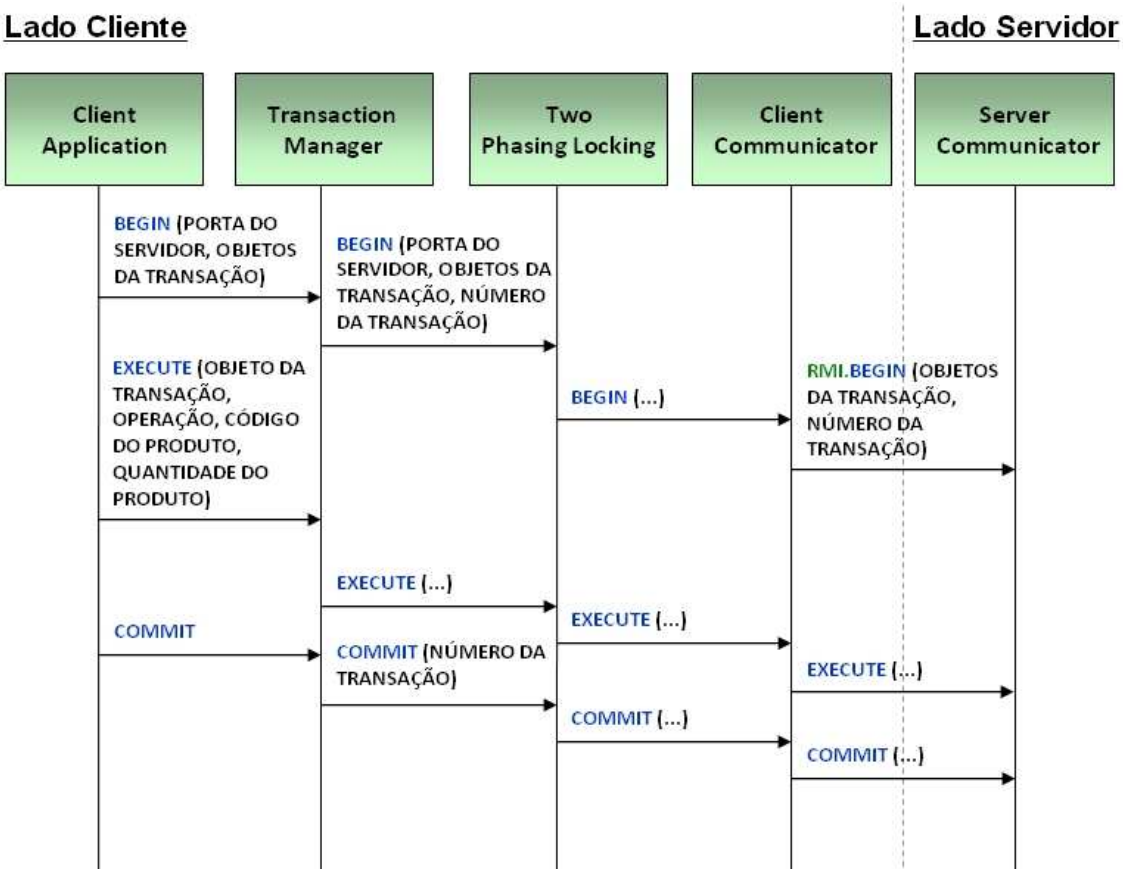


Figura 4.11: Diagrama de sequência do controle pessimista – lado Cliente

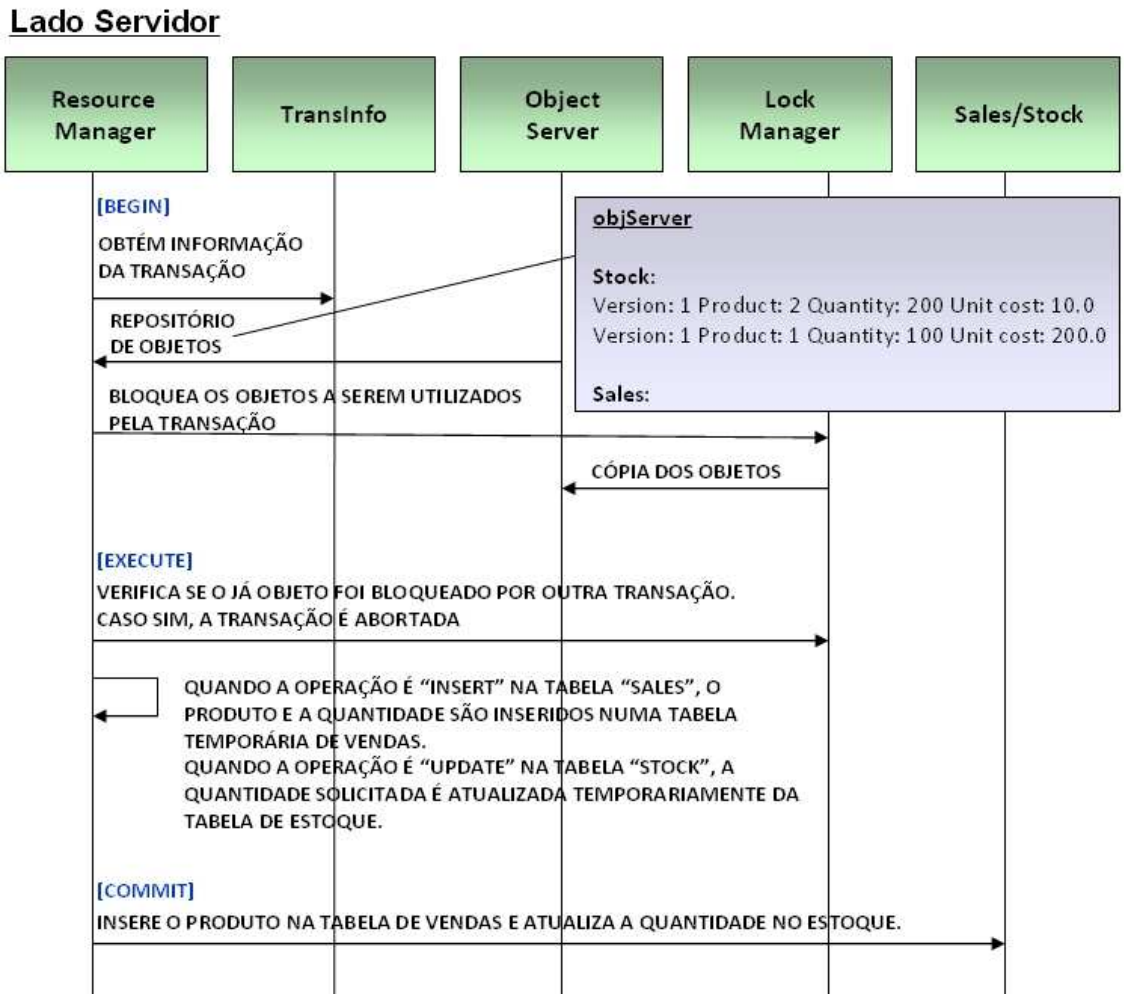


Figura 4.12: Diagrama de sequência do controle pessimista – lado Servidor

4.1.2 Caso de Uso: Configuração do controle de concorrência

Suponha que o representante de vendas José tenha optado por monitorar os recursos do dispositivo móvel no primeiro caso de uso. Conforme a figura 4.13, ele pode solicitar ao monitor de recursos que verifique o nível do sinal de conexão ou da bateria. Nesse caso, ele solicitou monitorar o nível do sinal de conexão selecionando a opção "*Request Resource Monitoring*" e "*Signal Strength*". O limite mínimo deve ser escolhido também. Entre as opções, existem: "*No Connectivity*", "*Limited*", "*Good*", "*Very Good*" e "*Excellent*", ou seja, opções indicando desde nenhum sinal até um nível de conexão excelente. Esse limite indica qual o nível mínimo do recurso monitorado será suportado pelo dispositivo móvel.

Nesse exemplo, a opção escolhida foi "*Very Good*". Isto significa que, se o nível do sinal de conexão do dispositivo estiver abaixo do limite "muito bom", o monitor de recursos notificará a transação que o limite está baixo do solicitado e a política de adaptação definida será aplicada.

Após selecionar a opção "*Confirm*", antes de iniciar a transação, o sistema verificará que a opção de monitoramento de recursos foi selecionada. O componente *ResourceMonitor* é chamado para monitorar o nível de sinal de conexão. Para simular os recursos do dispositivo móvel, foi criado um arquivo XML contendo a estrutura apresentada na figura 4.14. O monitor de recursos lê esse arquivo a cada meio segundo.

Dentro do arquivo XML, o elemento "s*ignalStrength*" contém o valor 4 indicando que o nível do sinal de conexão do dispositivo móvel está excelente e portanto dentro do limite estabelecido pela aplicação cliente (limite de sinal igual a "muito bom").

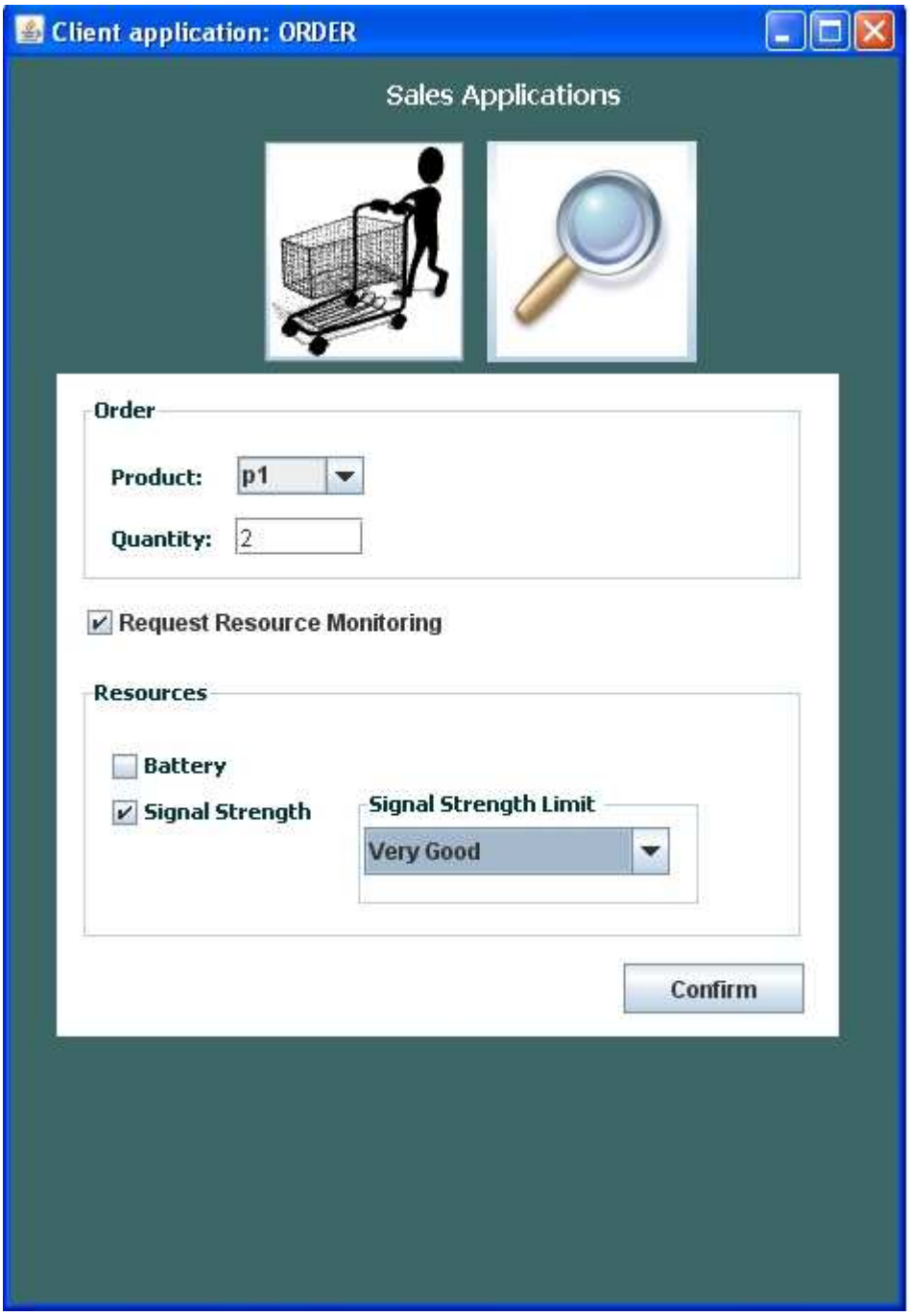


Figura 4.13: Solicitação de monitoramento de recursos do dispositivo móvel

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">

<properties>

    <comment>
        Battery:
            0 a 100%
        Signal Strength:
            0 - No Connectivity
            1 - Limited
            2 - Good
            3 - Very Good
            4 - Excellent
    </comment>

    <entry key="battery">
       <![CDATA[100]]>
    </entry>

    <entry key="signalStrength">
        <![CDATA[4]]>
    </entry>
</properties>
```

Figura 4.14: Resources.xml

A transação será iniciada no dispositivo móvel do José. Como no caso de uso anterior, o controle de concorrência seria o pessimista. Porém, como ele solicitou o monitoramento do sinal de conexão, o monitor de recursos constantemente verifica se o sinal está dentro do limite pré-estabelecido. A conexão com o servidor estava excelente, porém de repente o sinal ficou limitado, antes do início da transação.

O outro representante de vendas Manuel, ao mesmo tempo, executa a aplicação cliente também solicitando o monitoramento de recursos do seu dispositivo móvel. O recurso escolhido também é o sinal de conexão e o limite tem que ser no mínimo "muito bom". O dispositivo móvel do Manuel está com uma conexão excelente com o servidor.

A transação da aplicação do Manuel inicia chamando a operação *begin*. Antes da execução da operação *begin*, a conexão com o servidor é alterada e se torna limitada. Para representar esse cenário, o valor do elemento "*signalStrength*" do arquivo Resources.xml é alterado de 4 para 1, conforme a figura 4.15.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">

<properties>

    <comment>
        Battery:
            0 a 100%
        Signal Strength:
            0 - No Connectivity
            1 - Limited
            2 - Good
            3 - Very Good
            4 - Excellent
    </comment>

    <entry key="battery">
       <![CDATA[100]]>
    </entry>

    <entry key="signalStrength">
        <![CDATA[1]]>
    </entry>
</properties>
```

Figura 4.15: Resources.xml – nível do sinal alterado

Neste momento, o monitor de recursos, após ler o arquivo XML, percebe que o um dos recursos foi alterado. Ele compara o nível do sinal do arquivo com o limite estabelecido pelo cliente da aplicação. Como o limite do recurso monitorado é menor que o limite esperado, o monitor de recursos notifica a transação. A transação então avalia as regras da política de adaptação para tomar alguma decisão diante da mudança.

Uma das regras definidas na política de adaptação é que, se um dos recursos monitorado estiver abaixo do limite estabelecido pelo cliente da aplicação antes do início da transação (antes da operação *begin*), o controle de concorrência deve ser alterado de pessimista para otimista.

Para que o controle de concorrência seja alterado dentro da arquitetura proposta, o monitor de recursos notifica a classe *ClientConfigurator* que contém a política de adaptação. Essa classe solicita para o componente *ArchitectureClient* que ele altere o controle de concorrência de pessimista para otimista. Esse componente, por sua vez, desconecta o componente *TwoPhaseLocking* do componente *TransactionManager* e conecta o componente *VersionControl*. A nova arquitetura do lado cliente é exposta na figura 4.16.

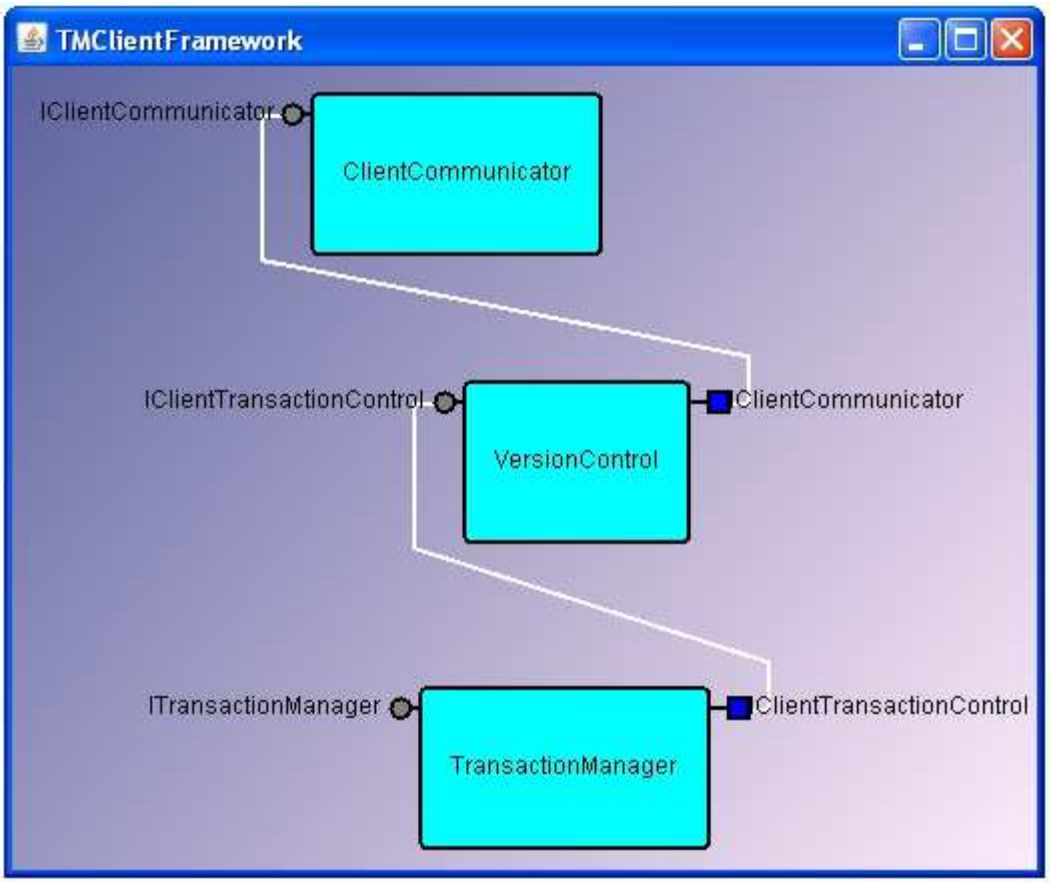


Figura 4.16: Arquitetura do lado cliente com controle de concorrência otimista

Até esse momento, o controle de concorrência foi alterado apenas do lado cliente. O mesmo deve acontecer do lado servidor. Para isso, a classe *ClientConfigurator* após solicitar a alteração do controle de concorrência do lado cliente, manda uma notificação para a classe *ServerConfigurator* através da tecnologia RMI. A classe *ServerConfigurator* ao receber a notificação, solicita ao componente *ArchitectureServer* a alteração do controle de concorrência de pessimista para otimista. Esse componente, por sua vez, desconecta o componente *LockManager* do componente *ResourceManager* e conecta o componente *VersionManager*. A nova arquitetura do lado servidor é exposta na figura 4.17.

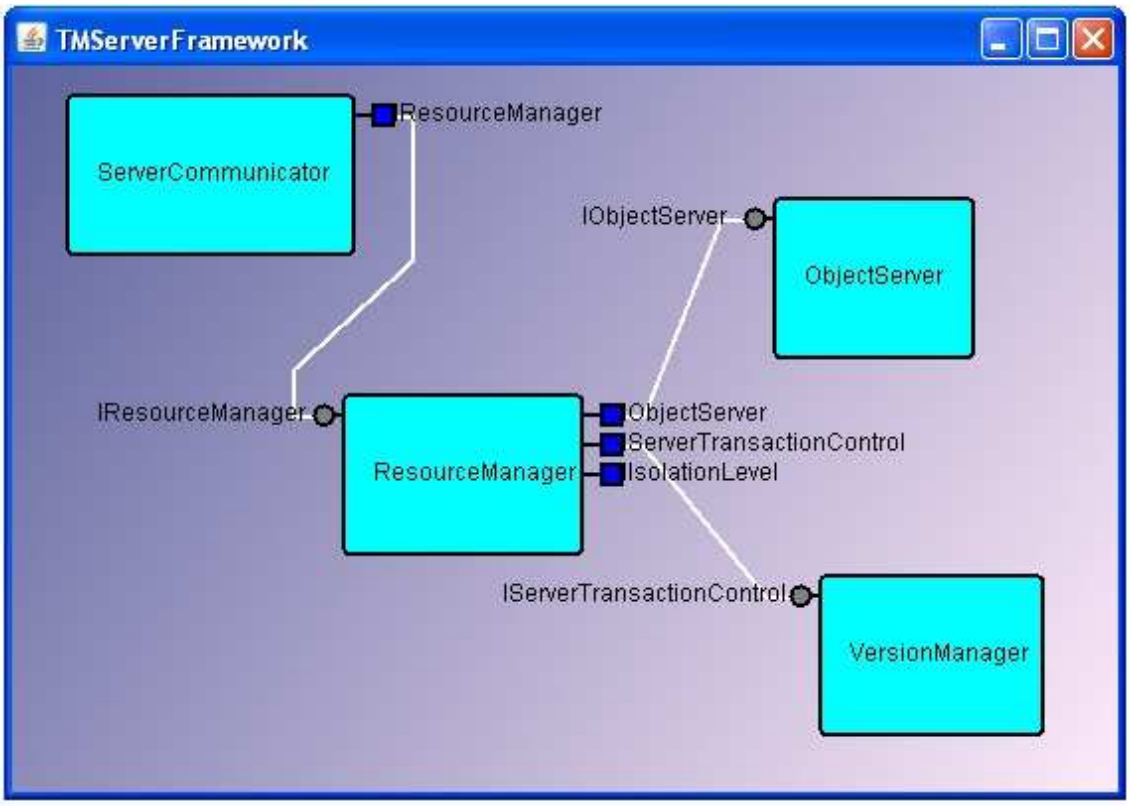


Figura 4.17: Arquitetura do lado servidor com controle de concorrência otimista

No controle de concorrência otimista, os objetos não são bloqueados. Os objetos da transação são copiados para a máquina local do cliente (mantidos no *cache*) e todas as operações são efetuadas sobre a cópia local. Portanto, se a conexão com o servidor cair, a transação pode continuar.

Nesse exemplo, os objetos *Stock* e *Sales* são copiados para o lado cliente e as operações de inserção e atualização são efetuadas do lado cliente. Essa é a fase de preparação. O responsável por esse processo é o componente *VersionControl*. Apenas no final da transação, nas fases de validação e escrita, o cliente necessita se conectar ao servidor.

O dispositivo móvel do representante de vendas Manuel estava com a conexão limitada. Prevendo uma possível desconexão, os objetos da transação foram copiados e as operações de inserção na tabela de vendas e de atualização na tabela de estoque foram executadas localmente no dispositivo móvel. No final da transação, as alterações são validadas para verificar se a versão de cada objeto foi alterada. As versões dos objetos de vendas e estoque permaneceram a mesma, portanto a venda do produto foi efetivada e o estoque atualizado, como se pode ver nas figuras 4.18 e 4.19.

O controle de concorrência do dispositivo móvel do José também foi alterado para otimista já que o nível do sinal estava limitado e ele havia estabelecido um limite de nível de conexão “muito bom”. Ele conseguiu executar localmente as operações sobre os objetos. Porém, quando ele foi efetivar a transação do lado servidor, como o representante de vendas Manuel já havia alterado os objetos de vendas e estoque, as versões estavam

diferentes (a versão dos objetos *Sales* e *Stock* do repositório com a versão desses objetos do *cache* do lado cliente). Dessa forma, José não conseguiu efetivar o seu pedido.

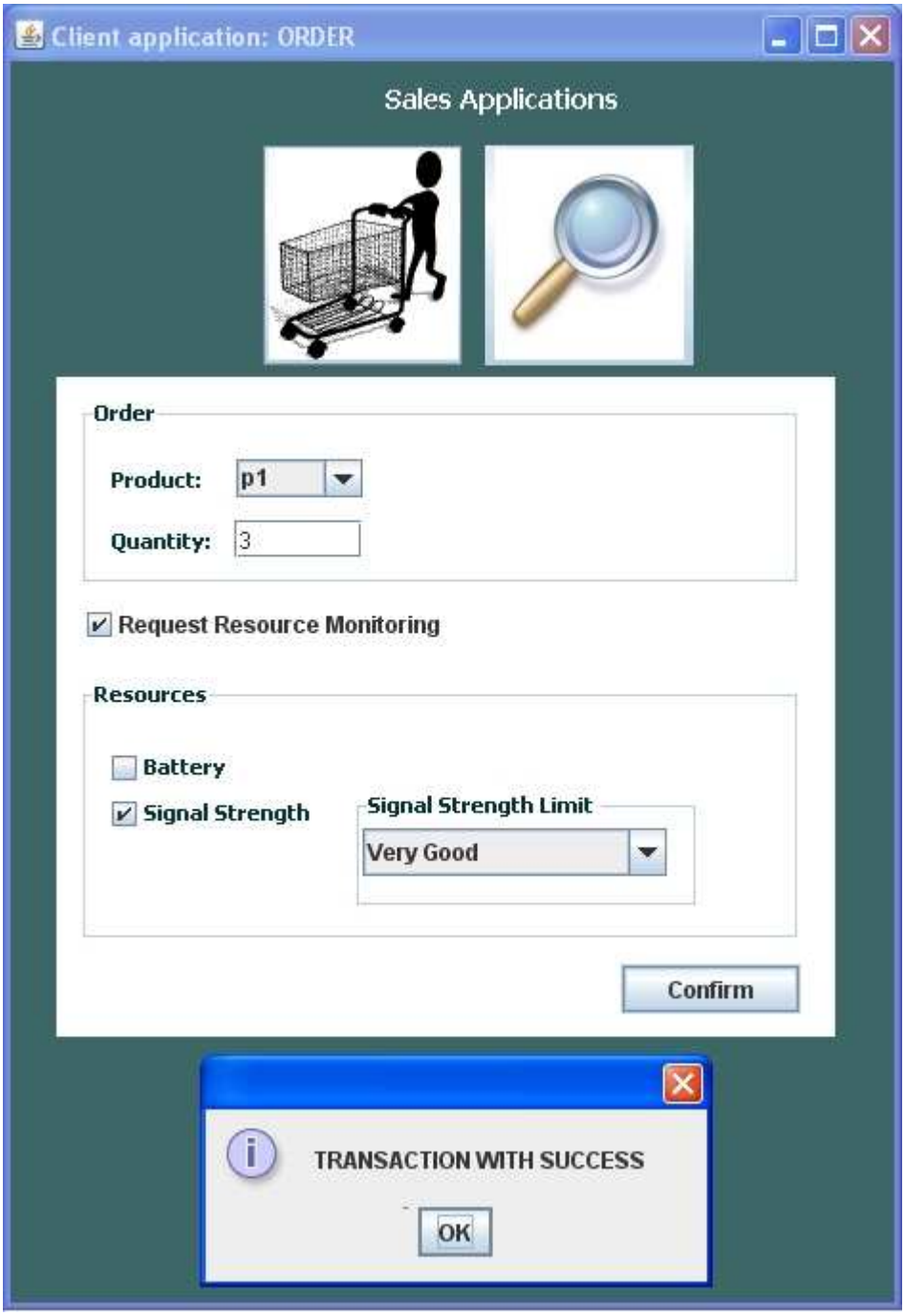


Figura 4. 18: Transação do Manuel foi efetivada com sucesso

Figura 4.19: Estoque atualizado

No caso de uso acima, os dois clientes iniciaram a transação utilizando o controle de concorrência otimista, devido ao sinal de conexão limitado. Supondo que a transação do dispositivo do José inicie utilizando o controle pessimista (o sinal de conexão não está limitado) e que antes de a transação ser efetivada, o Manuel também solicita o mesmo produto. O nível de conexão do dispositivo móvel dele está limitado e, portanto, o controle é alterado para otimista do lado cliente. Quando o servidor receber uma solicitação para alterar o controle de concorrência, ele deve verificar se existe alguma transação iniciada com algum cliente em outro modo de concorrência. Se existir, ele mantém o modo de concorrência e recusa a nova solicitação. Nesse caso, José conseguiria efetivar a transação, Manuel não conseguiria mudar o modo de controle de concorrência e teria que prosseguir no modo pessimista ou abortar a transação. O motivo disso é evitar que cliente e servidor tenham tipos de controles de concorrência diferentes.

Da mesma forma que o caso de uso anterior, as figuras 4.20 e 4.21 mostrarão respectivamente os componentes e a seqüência de métodos utilizados no tipo de controle de concorrência otimista.

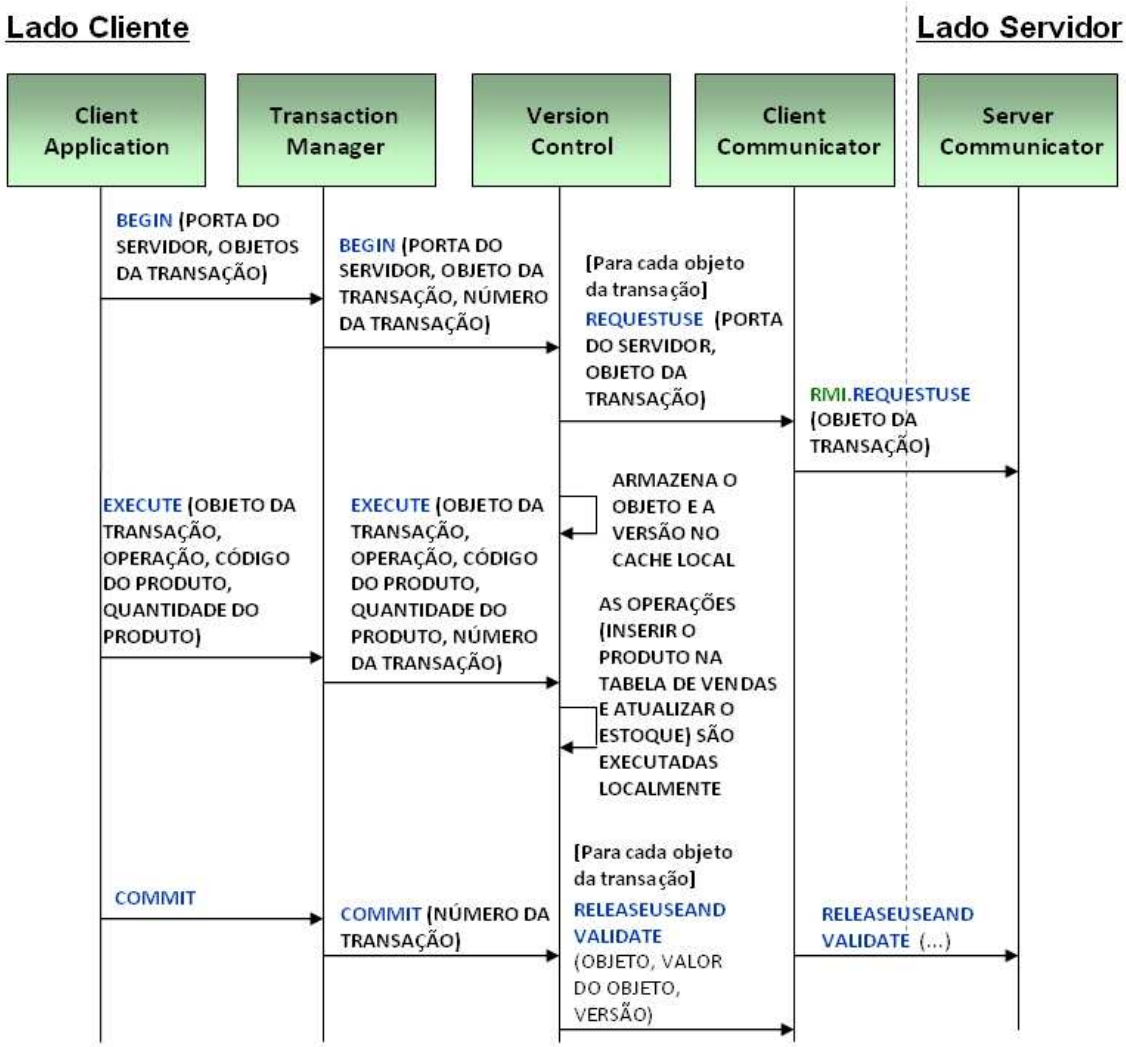


Figura 4.20: Diagrama de sequência do controle otimista – lado Cliente

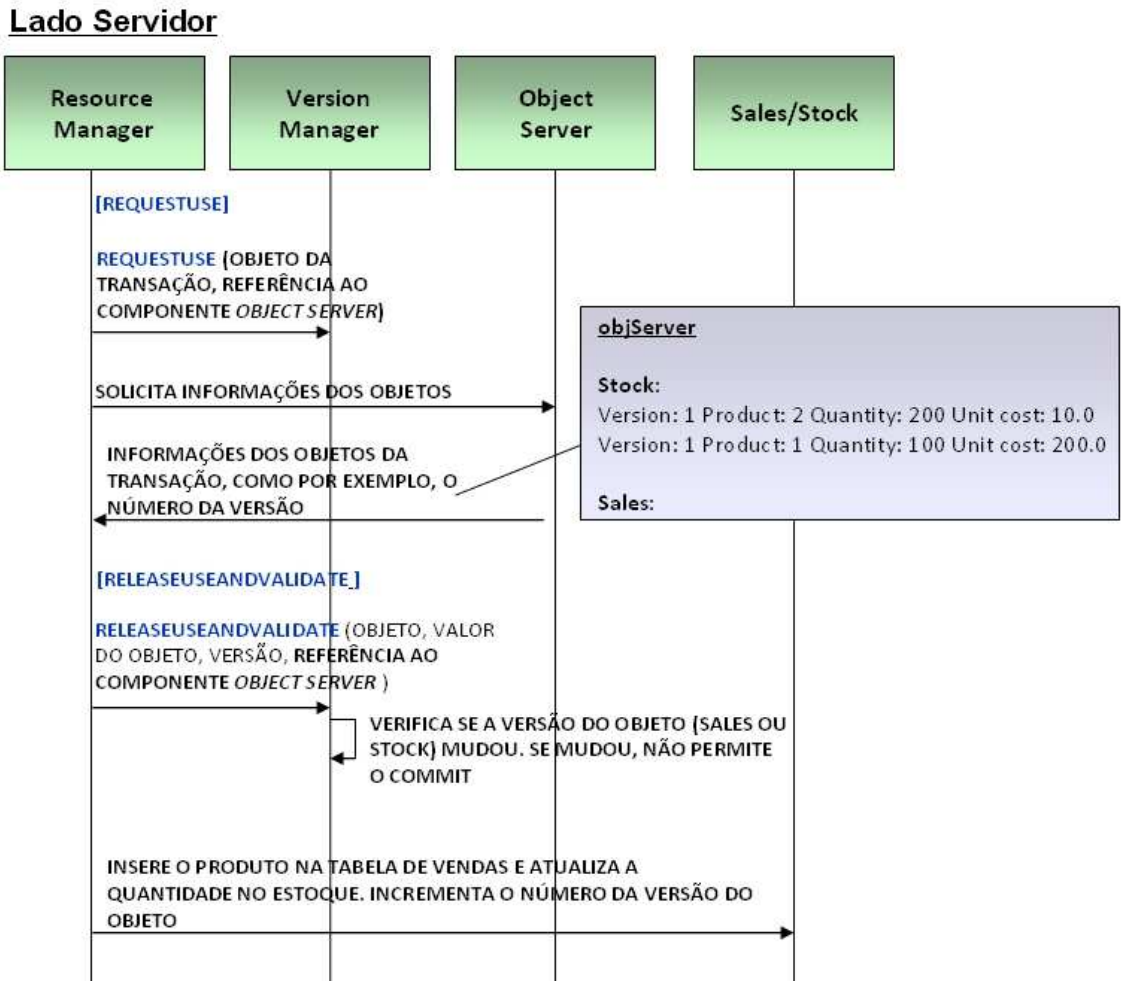


Figura 4.21: Diagrama de sequência do controle otimista – lado Servidor

4.1.3 Caso de Uso: Reconfiguração do nível de isolamento

Os casos de uso anteriores demonstraram a configuração do controle de concorrência antes do início da transação. Esse caso de uso demonstrará a reconfiguração do nível de isolamento durante uma transação.

Suponha que os seguintes pedidos já tenham sido efetivados:

Produto = “p2” com quantidade = 1
Produto = “p2” com quantidade = 2
Produto = “p2” com quantidade = 3

Porém, um outro pedido é solicitado, mas não é efetivado (as operações de inserir na tabela de vendas e atualizar o estoque são executadas, mas a operação *commit* não é realizada ainda). Esse pedido é descrito abaixo:

Produto = “p1” com quantidade = 50

Ao mesmo tempo, o gerente da equipe de vendas, para tomar uma decisão, necessita consultar a média das vendas realizadas no dia. Como ele está fora do escritório, ele utiliza o seu PDA para obter esse resultado. Ele seleciona a opção “*Sales report*” e solicita o monitoramento de recursos do dispositivo móvel. O recurso a ser monitorado será a bateria. A aplicação é exibida na figura 4.22.

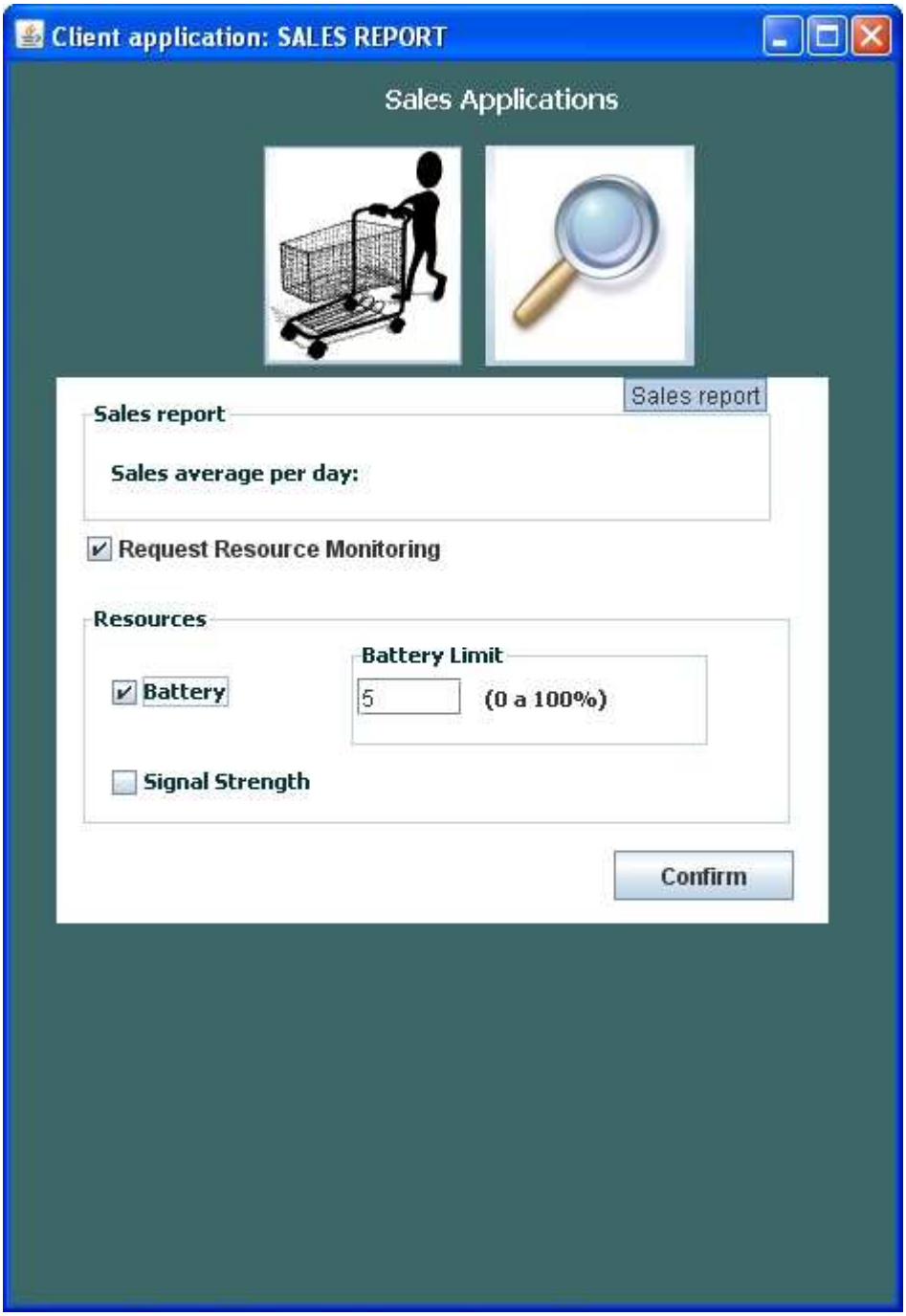


Figura 4.22: Opção selecionada: relatório das vendas do dia

Supondo que o nível da bateria do dispositivo móvel esteja 100%, a aplicação exibirá apenas a média das vendas efetivadas (após *commit*). Portanto, no exemplo acima, a média é igual a 10. Segue o cálculo:

Produto = “p2” com quantidade = 1 (1 x 10.00) = 10.00
Produto = “p2” com quantidade = 2 (2 x 10.00) = 20.00
Produto = “p2” com quantidade = 3 (3 x 10.00) = 30.00

Total: 10.00 + 20.00 + 30.00 = 60.00
Quantidade total de produtos: 1 + 2 + 3 = 6
Média: 60.00 / 6 = **10.00**

No cálculo acima, o produto “p1”, com quantidade = 50, não foi considerado porque o controle de concorrência definido é o pessimista e por padrão o nível de isolamento é *Serializable*. Esse nível não permite que os resultados de transações não efetivadas sejam visualizados.

Porém, conforme o gerente utiliza o dispositivo, ele percebe que o nível da bateria está diminuindo rapidamente. Como ele optou por um limite de 5%, se o nível da bateria estiver abaixo, o monitor de recursos notificará a transação e a política de adaptação será executada. A transação iniciada, conforme demonstrado na figura 4.23, tem a política de adaptação de alterar o nível de isolamento de *Serializable* para *Read Uncommitted* se o nível de bateria estiver abaixo do limite estabelecido, conforme a figura 4.24.

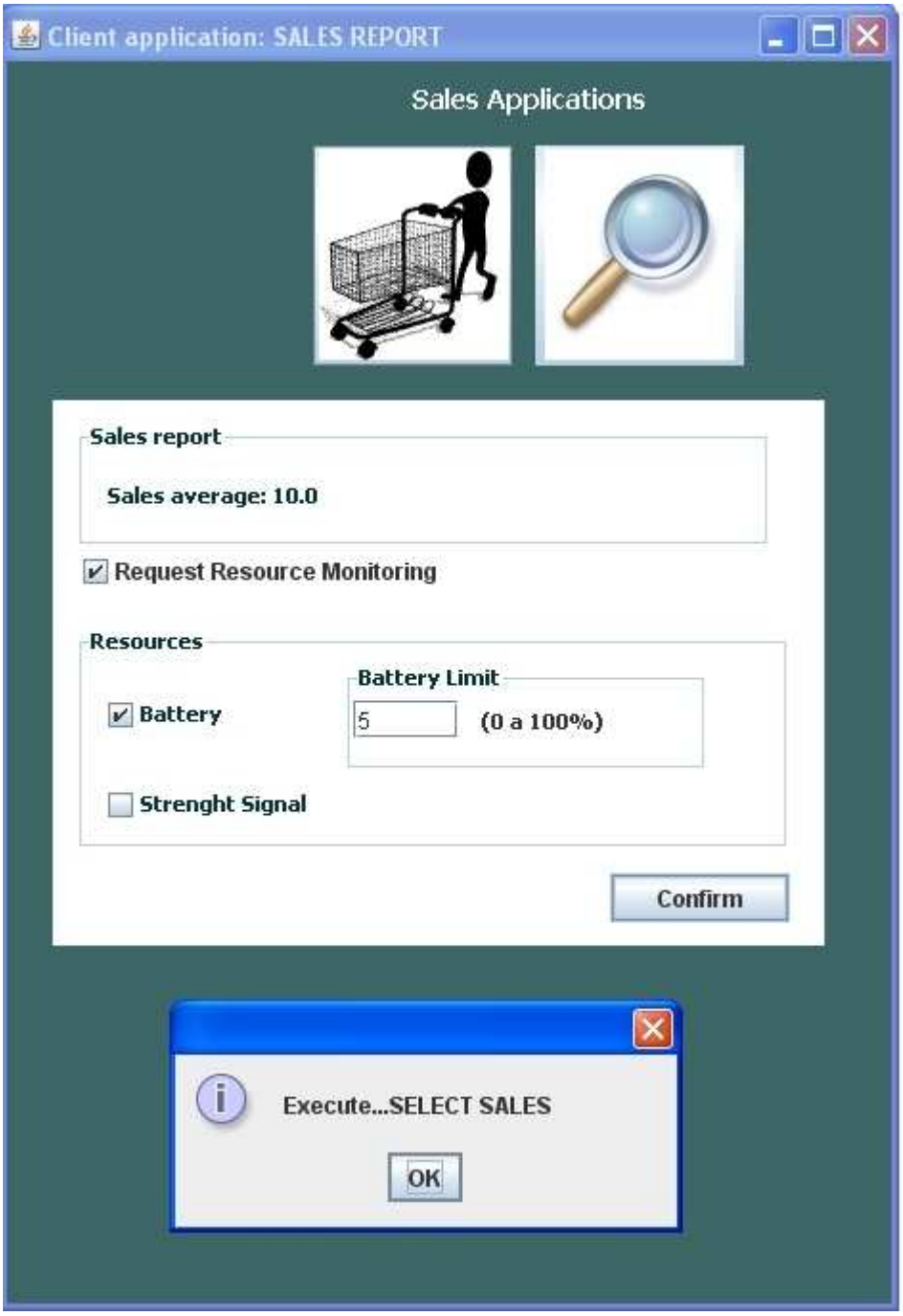


Figura 4.23: Transação iniciada

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">

<properties>

    <comment>
        Battery:
            0 a 100%
        Signal Strength:
            0 - No Connectivity
            1 - Limited
            2 - Good
            3 - Very Good
            4 - Excellent
    </comment>

    <entry key="battery">
       <![CDATA[4]]>
    </entry>

    <entry key="signalStrength">
        <![CDATA[3]]>
    </entry>
</properties>
```

Figura 4.24: Nível da bateria em 4%

O monitor de recursos notifica a classe *ClientConfigurator* que verifica se a transação já começou. Como a transação já foi iniciada, apenas o nível de isolamento pode ser alterado. Se alguma transação estiver em andamento com outro nível de isolamento, ela não será afetada. O servidor permite a execução de diferentes níveis de isolamento em paralelo. Por uma questão de simplificação, isso não foi implementado nesse protótipo, mas a arquitetura permite essa simultaneidade devido à componentização. Como do lado cliente não existe alteração, a classe *ClientConfigurator* notifica a classe *ServerConfigurator* através da tecnologia RMI. A classe *ServerConfigurator* solicita para o componente *ArchitectureServer* a alteração do nível de isolamento. Esse componente, por sua vez, desconecta o componente *Serializable* do componente *ResourceManager* e conecta o componente *Read Uncommitted*. A arquitetura alterada é mostrada na figura 4.25.

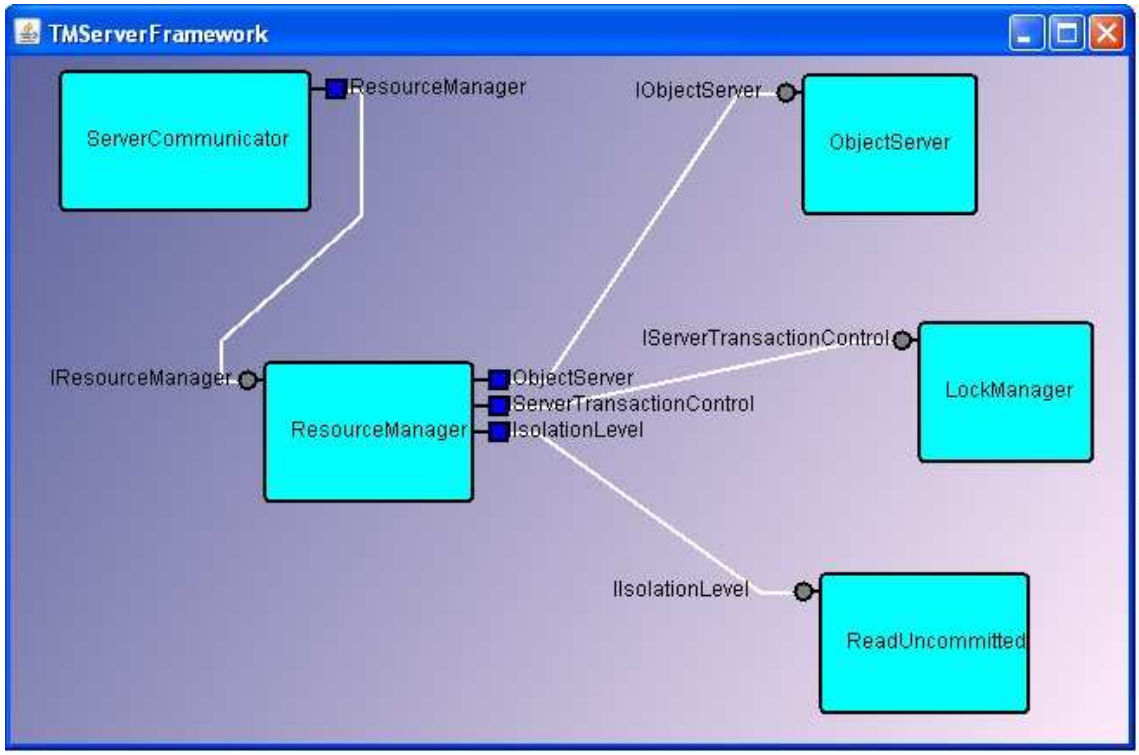


Figura 4.25: Nível de isolamento alterado durante a transação

Enquanto isso, a transação do gerente de vendas continua e como o nível de isolamento é menos restrito, as vendas não efetivadas serão visualizadas pelo gerente. Portanto, a última venda não concluída será considerada no cálculo, logo o resultado mostrado ao gerente não será mais 10.00 e sim 179.64 (figura 4.26). Foram consideradas as seguintes vendas:

Produto = "p2" com quantidade = 1 (1 x 10.00) = 10.00
Produto = "p2" com quantidade = 2 (2 x 10.00) = 20.00
Produto = "p2" com quantidade = 3 (3 x 10.00) = 30.00
Produto = "p1" com quantidade = 50 (50 x 200.00) =10000.00

Total: 10.00 + 20.00 + 30.00 + 10000.00 = 10060.00
Quantidade total de produtos: 1 + 2 + 3 + 50 = 56
Média: 10060.00 / 56 = **179.64**

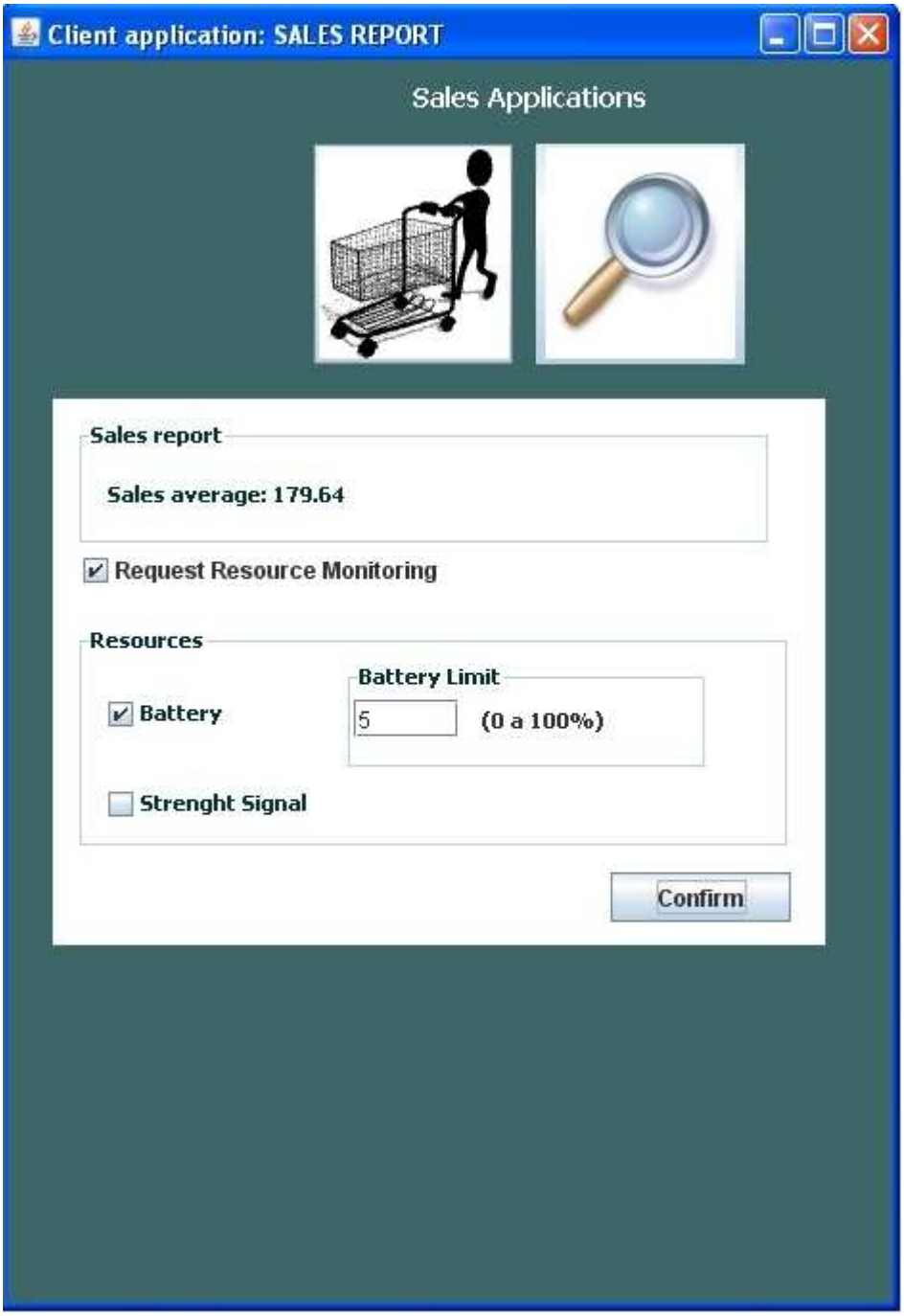


Figura 4.26: Novo resultado da média das vendas realizadas no dia

Se a última venda for concluída pelo representante de vendas, a média de vendas realizadas exibida para o gerente estará correta. Porém, o representante de vendas desistiu do produto, ou seja, a última venda não foi efetivada. Sendo assim, a média das vendas seria 10.00 e não 179.64. Mas o gerente tinha necessidade de ver a informação rapidamente, pois o nível da bateria estava diminuindo e logo o dispositivo móvel deixaria de funcionar. Além disso, o resultado exibe uma média geral. A preocupação nesse tipo de aplicação não é a consistência dos dados e sim o desempenho. O gerente sabe que algumas

vendas podem não ser efetivadas, mas diante da situação do ambiente, ele opta por considerar essas informações para tomar uma decisão.

4.2 Outros Estudos de Casos

Existem outras aplicações que podem necessitar a visualização de dados intermediários atualizados por transações em execução.

Algumas empresas aéreas em parcerias com operadoras permitem que passageiros previamente cadastrados, com apenas bagagem de mão, façam *check-in* e escolham os assentos pelo celular. Um dia antes do vôo, o passageiro recebe um SMS (*Short Message Service*) com os dados do vôo para que ele possa fazer o *check-in*.

Um mapa do avião é exibido e o passageiro marca o assento desejado. Porém, ao mesmo tempo, outro passageiro pode escolher o mesmo assento, pois consta no sistema que este está liberado. Se o controle de concorrência é pessimista, o segundo passageiro não conseguirá escolher esse assento até o primeiro passageiro terminar a transação.

Como ele está usando um dispositivo móvel, os recursos são limitados. O processo de *check-in* pode ser demorado e o passageiro pode não conseguir fazê-lo (a bateria pode acabar, a conexão cair ou o custo da comunicação pode extrapolar o valor permitido). Sendo assim, durante a transação, se algum recurso estiver limitado, uma alternativa é permitir que os possíveis assentos ocupados sejam exibidos ao passageiro.

Considere duas passageiras: Maria e Joana. Ambas estão fazendo o *check-in* pelo celular ao mesmo tempo e escolhem o mesmo assento. Maria escolhe o assento A2, mas antes de efetivar a reserva do assento, Joana está vendo o mapa de assentos. Ela visualizará o assento A2 como possível ocupado (o assento não estará marcado como ocupado e sim como um assento que poderá ser ocupado logo). Joana então pensa se ela deseja continuar a transação assegurando esse assento mesmo assim, ou escolher outro. O que pode acontecer:

- Se ela decidir continuar com o assento A2, no final da transação, pode haver conflito, pois realmente Maria efetivou a reserva do assento A2. Mas se Maria desistiu desse assento, Joana conseguirá reservá-lo.
- Se ela escolher outro assento, no final da transação, pode ser que Maria tenha desistido do assento A2, mas de qualquer forma Joana conseguiu reservar um outro assento.

Seja qual for a situação, numa situação de concorrência e recursos limitados, é interessante permitir a alteração de uma das propriedades transacionais, como o nível de isolamento. O nível de isolamento pode ser alterado durante a transação caso um recurso passe a ficar limitado. Se os recursos estão disponíveis não haveria necessidade de visualizar os resultados intermediários, pois o passageiro poderia tentar várias vezes a reserva de um determinado assento.

Outra aplicação que pode se beneficiar da arquitetura proposta é a inscrição em palestras ou cursos gratuitos através de um site especializado. Empresas e universidades interessadas cadastram neste site as palestras e os cursos gratuitos que serão divulgados aos usuários previamente cadastrados no sistema. As inscrições podem ser realizadas pela internet através de um computador pessoal. Porém, como os cursos são gratuitos e as vagas são limitadas, a concorrência por uma delas é bem grande.

Como as vagas são preenchidas rapidamente, muitos interessados perdem a possibilidade de participar dos cursos por não terem um computador disponível no

momento desejado. Seguindo a tendência da mobilidade, o site especializado envia mensagens SMS de cursos e palestras que atendam ao perfil dos usuários. Através do celular, o usuário pode se inscrever no curso ou palestra desejados. Como as vagas são limitadas e não há custo, a concorrência é freqüente, sendo portanto, necessário haver um controle para evitar conflitos.

O controle de concorrência padrão seria o pessimista, mas devido às características do ambiente móvel, o controle de concorrência poderia ser alterado para otimista caso algum recurso do celular (bateria ou sinal da conexão) estivesse limitado no início da transação.

A transação englobaria duas operações: registro da inscrição no curso selecionado e envio de mensagem de confirmação ao usuário. As operações da transação devem ser executadas de forma atômica e caso a mensagem não seja enviada com sucesso, a inscrição no curso não é efetivada.

Ao utilizar o controle de concorrência otimista, apenas no final da transação é que poderia haver algum conflito: mais de um usuário tentando se inscrever em uma última vaga de um mesmo curso e então o sistema permitiria a inscrição de apenas um dos usuários, o que chegou ao final da transação primeiro.

Capítulo 5

Trabalhos relacionados

Esse capítulo apresenta um resumo de alguns trabalhos publicados sobre modelos de transações para ambientes móveis, detalhando principalmente os trabalhos sobre MobileTrans [46], CATE [45] , AMT [1], ReflecTS [4 e 5] e SGTA [42 e 43]. Esses trabalhos foram usados como base para o desenvolvimento da dissertação.

5.1 MobileTrans

É um modelo de transações [46] baseado em objetos distribuídos e provê adaptabilidade a cenários específicos de ambientes de computação móvel. Os programadores configuram uma política de adaptação, parametrizando-a através de atributos, como por exemplo, modo de execução e requisitos mínimos.

A arquitetura do MobileTrans é do tipo cliente-servidor. Os objetos são armazenados em repositórios gerenciados pelos servidores. O nó que contém a versão consistente do objeto é chamado de *home node.* Os clientes possuem um *cache* contendo réplicas de objetos.

Mesmo quando uma conexão falha, deixando um ou mais nós indisponíveis, os cancelamentos de transações podem ser evitados. Isto pode ser feito de duas formas: as transações são adiadas até que os nós se tornem novamente disponíveis (aumentando o tempo de execução da transação) e/ou através do relaxamento das propriedades ACID.

Alguns indicadores são definidos: tempo de tolerância da transação (t_T) e nível de consistência da transação (c_T). O tempo de tolerância define o tempo máximo para a execução da transação. Se a conexão está estável durante a transação, $t_T = 0s$.

Os níveis de consistência são definidos na tabela abaixo:

c	Busca	Efetivação
5	Réplicas consistentes completas	Nenhuma atualização pode ser descartada
4	Réplicas consistentes completas	Atualizações podem ser descartadas
3	Réplicas inconsistentes	Nenhuma atualização pode ser descartada
2	Réplicas inconsistentes	Atualizações podem ser descartadas
1	Réplicas nulas	Nenhuma atualização pode ser descartada
0	Réplicas nulas	Atualizações podem ser descartadas

Tabela 5.1: Nível de consistência

Com relação à busca de objetos é possível:

a) Obter uma réplica consistente completa do objeto requerido, sendo, portanto, necessário que o nó específico esteja disponível (*home node*).
b) Obter qualquer réplica (até inconsistente) do objeto. Neste caso, a réplica pode ser obtida de qualquer nó que contenha uma réplica em *cache*.
c) Obter uma réplica nula quando nenhum nó está disponível.

Com relação à efetivação da transação, é possível:

a) Validar todas as atualizações da transação, sendo necessário que todos os nós participantes estejam disponíveis.
b) Descartar atualizações cujos nós não estejam disponíveis durante a fase de efetivação.

Para minimizar os cancelamentos devido a desconexões, um procedimento de recuperação (f_t) incrementa o tempo de tolerância da transação ou decrementa o nível de consistência. Isto dependerá dos valores dos atributos modo de execução e requisitos mínimos especificados pelo programador.

Existem dois tipos de modo de execução: *min-time* e *max-consistency*. Para definir qual modo escolher, o programador considera as propriedades da transação. No modo *min-time*, a transação é executada no menor tempo possível mesmo com menor consistência dos dados. Já o *max-consistency* privilegia a alta consistência dos dados, podendo causar um atraso no tempo de execução.

No modo *min-time,* em caso de desconexão, a transação é adiada até o c_{min} (nível de consistência mínimo) ser alcançado. Se o t_{max} (tempo de tolerância máximo) for excedido e o c_{min} não for atingido, a transação é cancelada. Por exemplo: $c_{min} = 3$ e $t_{max} = 15$ s.

Se a transação precisar de um objeto e o *home node* não estiver disponível, uma réplica inconsistente é procurada no cache local ou no cache do nó mais próximo. Se a réplica for encontrada, o c_{min} é atingido e a transação prossegue, senão, a transação é adiada até 15s. Se o tempo ultrapassar t_{max}, a transação é cancelada.

No modo *max-consistency*, em caso de desconexão, a transação é adiada até que o nível de consistência máximo ($c_{max} = 5$) seja obtido. Enquanto este nível não for alcançado, a transação é adiada até o t_{max}.

Se o t_{max} expirar e se o c_T (nível de consistência da transação) atingido até o momento é maior que c_{min}, a transação prossegue com o nível de consistência c_T, senão, a transação é cancelada.

As propriedades das transações nesse modelo [47] são:

- Consistência**:** é possível especificar regras de consistência permitindo usar versões antigas dos objetos, caso o *home node* não estiver mais acessível.
- Transferência de objetos**:** a política descreve se um objeto deve ser solicitado antes do início da transação ou sob demanda durante a execução da transação.

- Delegação**:** quando uma transação está prestes a finalizar, é possível delegar para outro nó a função de efetivação da transação.
- Atomicidade**:** essa política especifica se a transação pode ser finalizada com sucesso mesmo se nem todos os nós estiverem acessíveis.
- *Caching**:*** enquanto é executada a transação, é possível armazenar os objetos em *cache*. Isso é essencial para garantir disponibilidade em caso de desconexão.
- Tratamento de falhas: política responsável por determinar como reagir perante falhas e permitir a mudança de configuração da transação em tempo de execução para tratamento de falhas.

A política de transação é especificada pelo implementador e pode ser definida através de um arquivo XML ou programação (Java/C++). Os atributos que podem ser especificados são definidos na tabela abaixo [47].

Attributes				
Name		**Value**		**Arguments**
consistency	.object	*required*		–
		replica		–
		dispensable		–
	.degree	*high*		–
		medium		–
		low		–
fetching	.object	*random*		depth
		node		depth, node
		randset		depth,{node}
	.mode	*ondemand*		–
		prefetch		{obj}
delegation	.coordinator	*random*		–
		node		node
		randset		{node}
	.responsibility	*local*		–
		foreign		–
atomicity	.object	*mandatory*		–
		tentative		–
	.degree	*high*		–
		low		–
caching	.read	*yes*		–
		no		–
	.write	*yes*		–
		no		–
failure	.consistency .fetching .delegation .atomicity .user.*	*abort*		–
		retry		attribute
		suspend	*.timeout*	time, attribute
			.reshape	attribute
			.user	attribute

Tabela 5.2: Atributos das transações

5.2 Transações Cientes de Contexto

CATE (***C**ontext **A**ware **T**ransaction s**E**rvice*) [45] é uma arquitetura baseada em componentes que utiliza o protocolo *Two-Phase Commit* (2PC). Essa arquitetura é sensível ao contexto e permite reconfiguração de componentes relacionados com o término da transação. São considerados vários tipos de protocolos de efetivação: 2PC, 2PC-PA (*2PC*

Presumed Abort que reduz o custo das transações canceladas) e 2PC-PC (*2PC Presumed Commit* que reduz o custo das transações efetivadas).

Para ser capaz de mudar o protocolo de efetivação no momento certo, é necessário monitorar as taxas de efetivação e cancelamento das transações. A política de adaptação é definida por regras ECA (*Event, Condition, Action*). Evento (do inglês *Event)* é a taxa de efetivação/cancelamento, condição (do inglês *Condition)* especifica quando o protocolo deve ser alterado e ação (do inglês *Action)* é a mudança de protocolo.

A figura 5.1 apresenta a arquitetura CATE composta de:

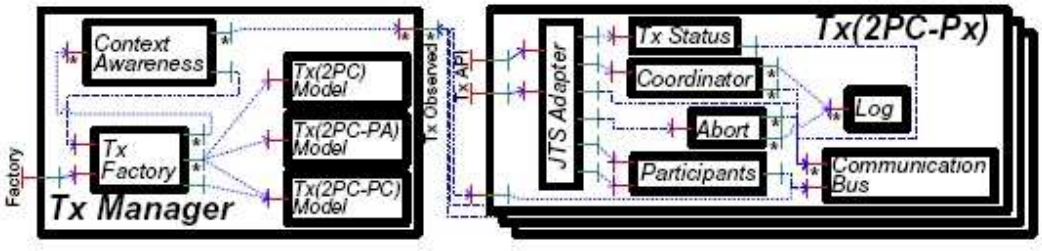


Figura 5.1: Arquitetura CATE

- *TxManager*: gerenciador de transação.
- *Context Awareness*: componente que implementa a arquitetura de adaptação. Monitora o número de transações efetivadas e canceladas e decide se o protocolo ativo deve ser mudado. Isto é possível devido às regras ECA pré-definidas. Exemplo: *if(abort_rate < 10%) then use 2PC*.
- O *Communication Bus* da transação ativa notifica o componente *Context Awareness* quando cada Coordenador (componente *Coordinator*) envia mensagens de efetivação ou cancelamento para os participantes (componentes *Participants*).
- Fornece apoio para JTS (*Java Transation Service*).
- O *Tx-2PC-Px* representa uma transação JTS (*Java Transaction Service*) implementando o protocolo 2PC-Px.
- *Coordinator*: concretiza o protocolo de efetivação.
- *Abort*: concretiza o protocolo de cancelamento.
- A reconfiguração do protocolo de efetivação é feita através do atributo *configuration*. Ele é lido pelo componente *Tx-Factory* quando novas transações são criadas. Dependendo do valor do atributo, o *Coordinator* implementa 2PC, 2PC-PA ou 2PC-PC. A reconfiguração consiste em alterar o valor deste atributo dependendo das condições pré-definidas.

Quando o componente *Context Awareness* decide mudar do protocolo, ele chama a interface *change-config* do *Tx-Factory* que conecta esta interface na configuração apropriada listada na interface *available-config*. Portanto, somente as futuras transações podem ser criadas com esta nova configuração.

Essa arquitetura permite ainda que novos protocolos sejam incluídos.

5.3 ReflecTS

ReflecTS [4 e 5] é uma plataforma de processamento de transações altamente adaptável e flexível. Esta flexibilidade vem do fato de permitir a configuração e seleção de vários serviços transacionais de acordo com as necessidades das aplicações. Além disso, permite que vários serviços de transação executem concorrentemente.

Baseia-se no OpenCOM [36] e no framework de componentes ReMMoc [39] para alcançar configurabilidade e reconfigurabilidade.

As principais características do ReflecTS são:

- Vários *Transactions Managers* (TMs) com o objetivo de apoio transacional para atender aos diferentes requisitos transacionais.
- Composição dinâmica do *Transaction Service* (TS) através da seleção de um *Transaction Manager* (TM) e dos *Resource Managers* (RMs) requeridos.

A arquitetura do ReflecTS [4], na Figura 4, é composta pelos componentes descritos abaixo:

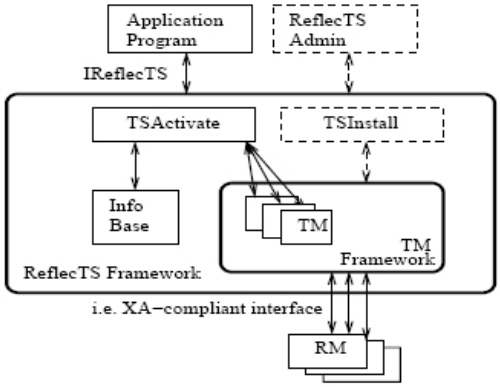


Figura 5.2: Arquitetura geral do ReflecTS

- Interface IReflecTS (apresentada na figura 5.3): define a interação entre programas de aplicação e ReflecTS. Os parâmetros do método TransBegin são: um documento XML que descreve os requisitos transacionais e os identificadores dos *Resource Managers*.

interface IReflecTS:IUnknown
HRESULT Trans_Begin(char* XMLDOC)
HRESULT Trans_Commit()
HRESULT Trans_Rollback()
HRESULT Trans_Info()

Figura 5.3: Interface IReflecTS

- *TMFramework*: hospeda as implementações do *Transaction Manager* (TM).
- *Resource Managers* (RM): são os pontos de acesso para o banco de dados.

- Um serviço de transação (*Transaction Service* – TS) é composto por um *Transaction Manager* (TM) e por vários *Resource Managers* (RM) (um para cada recurso envolvido). O objetivo do TS é coordenar a execução de uma transação assegurando os requisitos transacionais requeridos. Ao compor um TS, a compatibilidade entre TM e RMs é avaliada.
- *ReflecTS Framework*: contém os componentes que executam tarefas como configuração, reconfiguração, seleção e ativação do TM.
- *TSActivate*: recebe requisições para iniciar a transação, seleciona o TM, compõe o TS e ativa a transação. Para selecionar o TM, baseia-se nos requisitos transacionais representados na especificação TX_RQ e nos descritores *TM_Descriptor* e *RM_Descriptor* disponíveis.
- *TSInstall*: trata os pedidos de configuração e reconfiguração do gerenciador de transação.
- *InfoBase* – armazena os descritores TM e RM e as informações de compatibilidade.

Quando uma aplicação requisita apoio transacional, ela tem que informar as propriedades (requisitos) transacionais desejadas e os RM's que ela deseja utilizar. O ReflecTS utiliza a especificação TX_RQ para selecionar o TM que satisfaça os requisitos (o TX_RQ é comparado com os *TM_Descriptors* armazenados). Após a seleção, o ReflecTS verifica a compatibilidade dele com os RM's especificados. Um TM é controlado junto com os RMs dentro de um serviço de transação (TS).

Programadores devem especificar descritores de transações que podem ter propriedades ACID restritas ou mais flexíveis. Esses descritores são armazenados pelo ReflecTS para serem usados no processo de seleção de TMs e na composição do TS, quando uma aplicação requisita apoio transacional. Existem três tipos de especificações:

- *TX_RQ*: descreve os requisitos transacionais (estrutura da transação).
- *TM_Descriptor*: contém informações sobre as propriedades transacionais (ACID ou beyond-ACID), mecanismos de controle de concorrência, interface de comunicação entre TM e RM.
- *RM_Descriptor*: contém informações sobre os *Resource Managers* disponíveis.

5.4 AMT

AMT (***A**daptable **M**obile **T**ransaction*) [1] permite definir transações móveis (T_{AMT}) com várias alternativas de execução (*Execution Alternative* - EA) associadas a um descritor de ambiente (*Environment Descriptor* – ED).

O descritor do ambiente contém um conjunto de dimensões com seus respectivos estados em um determinado instante. A tabela 5.3 contém o conjunto mínimo de dimensões num descritor de ambiente. WN refere-se a redes sem fios (*Wireless Networks*) e MH a hospedeiros móveis (*Mobile Hosts*).

	Dimension	States	Unit
WN	connection-state	*connected, disconnected*	
	bandwidth-rate	*high, medium, low*	kbytes/s
	communication-price	*free, cheap, expensive*	Euros/time
MH	available-battery	*full, half, low*	hh:mm:ss
	available-cache	*full, half, low*	kbytes
	available-persistent-memory	*full, half, low*	kbytes
	processing-capacity	*high, medium, low*	mhz/s
	estimated-connection-time	*t*	hh:mm:ss
	location		

Tabela 5.3: Descritor de ambiente

Os programadores da aplicação são os responsáveis por configurar os descritores e as alternativas de execução, pois eles conhecem as características da aplicação e os requisitos de qualidade de serviço.

Cada alternativa de execução contém um conjunto de transações componentes que consideram as propriedades ACID. Podem existir transações de compensação que serão executadas em casos de falha. Uma alternativa tem um plano de execução (*Execution Plan* - EP) associado ao descritor de ambiente. Um plano de execução é um conjunto de triplas: transação componente com uma transação de compensação correspondente e a máquina onde será executada. Uma alternativa pode ser executada em uma das seguintes formas:

- A transação móvel é iniciada por uma máquina móvel e é inteiramente executada em máquinas fixas.
- A transação móvel é iniciada por uma máquina móvel/fixa e é inteiramente executada em uma máquina móvel.
- A execução é distribuída em várias máquinas móveis e fixas.
- A execução é distribuída em várias máquinas móveis.

Quando uma T_{AMT} é iniciada, o estado do ambiente móvel é verificado e a alternativa de execução apropriada é selecionada. Se nenhuma alternativa for apropriada, a execução pode ser adiada.

O trabalho contém um exemplo de uma aplicação de compra eletrônica. A tabela 5.4 mostra um modelo de transação AMT para essa aplicação contendo três alternativas de execução e seus respectivos descritores de ambiente e plano de execução. As prioridades entre as alternativas de execução são determinadas pelo custo da comunicação.

A dimensão *catalog-state* corresponde à presença ou não do catálogo de produtos (*missing* indica que o catálogo não está disponível na máquina móvel; *present* indica que a máquina móvel tem uma versão provavelmente desatualizada ou incompleta; *uptodate* indica que a máquina móvel tem uma versão atualizada). As dimensões que não aparecem não são relevantes para a aplicação. Devem também ser definidas transações de compensação para o cancelamento de pedido ou a devolução de valores cobrados.

EA_k	ED_k	EP_k
k=1	{catalog-state= *uptodate*}	{(SelectItems, MH), (Order-Pay, PurchaseS)}
k=2	{connection-state=*connected*, bandwidth-rate = *high, medium*, communication-price=*cheap*, catalog-state=*present, missing*}	{(GetCatalog, CatalogS) (SelectItems, MH), (Order-Pay, PurchaseS)}
k=3	{connection-state=*connected*, bandwidth-rate=*low*, catalog-state=*missing*}	{(GetCatalog, CatalogS), (Select-AutoPay, MH), (Order, PurchaseS)}

Tabela 5.4: Modelo AMT para aplicação de compra eletrônica

Como as T_{AMT} contêm um conjunto de transações cuja execução pode ser distribuída em máquinas fixas e móveis, as propriedades ACID são relaxadas. As transações móveis T_{AMT} têm as seguintes propriedades:

- Atomicidade semântica: ou todas as transações numa alternativa terminam com sucesso ou todas são canceladas (ou compensadas). Transações de compensação são executadas em casos de falhas desfazendo semanticamente as transações efetivadas de uma alternativa.
- Isolamento: As transações que podem ser compensadas liberam seus resultados ao terminar. As transações que não podem ser compensadas esperam a efetivação global para liberar seus resultados.

Para validar a proposta, foi desenvolvido um middleware denonimado TransMobi que gerencia a percepção do ambiente a implementa o modelo AMT com protocolos apropriados. Entre esses protocolos, foi proposto o CO2PC (uma combinação de 2PC com uma abordagem otimista). Este protocolo permite que transações componentes de compensação que executam em uma unidade móvel possam ser efetivadas localmente dentro da abordagem otimista enquanto que as transações que não são de compensação são efetivadas globalmente juntamente com a efetivação da EA. O protocolo 2PC não é utilizado na coordenação global, porque ele requer um grande número de mensagens e bloqueia a transação global inteira até decisão global ter sido tomada.

A percepção do estado do ambiente é implementada por eventos. TransMobi usa serviços de eventos para obter informações como mudanças de preço de comunicação, largura de banda, conexão, entre outros.

Para garantir a atomicidade semântica, foi proposto o protocolo CO2PC que combina 2PC (pessimista) com a abordagem otimista. Este protocolo permite que transações componentes compensáveis que executam em uma unidade móvel possam ser efetivadas localmente dentro da abordagem otimista enquanto que as transações não compensáveis são efetivadas globalmente juntamente com a efetivação da EA.

Para garantir a serialização global, foi escolhido OTM (*Optimistic Ticket Method*) [21].

5.5 SGTA

Os trabalhos [42 e 43] apresentam o modelo de transações SGTA (Sistema de Gerenciamento de Transações Adaptável) que possui a capacidade de adaptação diante dos obstáculos da computação móvel. Esta adaptação é obtida através de um monitor de recursos que notifica as transações sobre as mudanças ocorridas no ambiente móvel.

Cada transação pode reagir às mudanças ocorridas de acordo com a sua política de adaptação. A política de adaptação é definida através de um método implementado pelo usuário da transação.

A figura 5.4 [43] mostra uma visão geral do modelo.

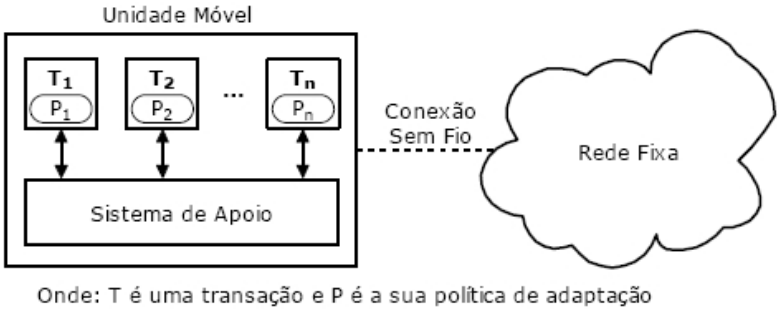


Figura 5.4: Visão geral do SGTA

Existem dois parâmetros que podem ser especificados no início da transação e modificados durante sua execução de adaptação: níveis de isolamento e modos de operação.

Os níveis de isolamento propostos são: nível 0, nível 1 (*UNCOMMITTED_READ*), nível 2 (*COMMITTED_READ*), nível 3 (*SERIALIZABILITY*). Os três modos de operação sugeridos são: local, remoto, local-remoto que permitem que as transações executem mesmo quando as máquinas estão desconectadas da rede.

O método de controle de concorrência utilizado no SGTA é o bloqueio de duas fases e por isso os níveis de isolamentos são em função de trancas de curta ou longa duração. Um tranca de curta duração é adquirida no início de execução da operação sobre o objeto e liberada logo após o término da mesma. Já a tranca de longa duração somente é liberada no final da transação.

Cada transação pode especificar os recursos do ambiente móvel que devem ser monitorados durante a sua execução (bateria, largura de banda, entre outros). Caso algum recurso não esteja disponível dentro dos limites adequados, a transação é notificada e pode ou ser cancelada, ou alterar o nível de isolamento e modo de operação.

A figura 5.5 [43] mostra a arquitetura do SGTA:

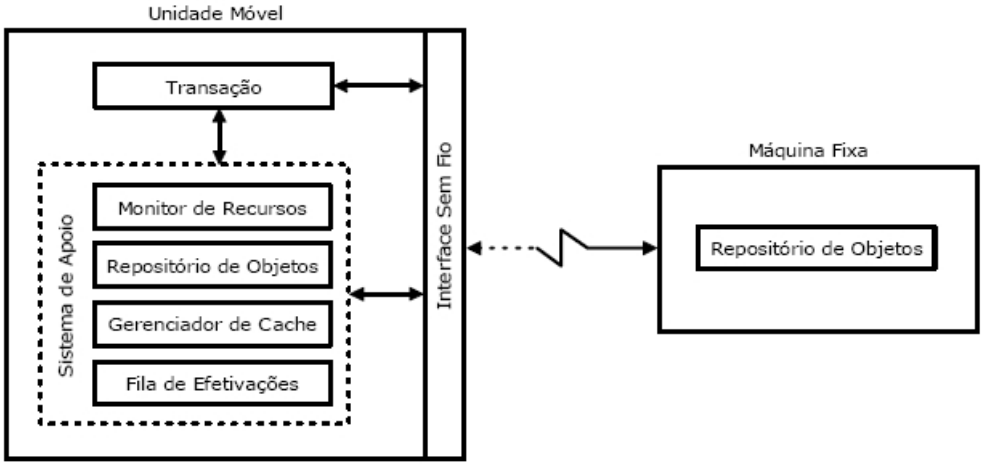


Figura 5.5: Arquitetura do SGTA

Segundo [43], a estratégia de adaptação segue os seguintes passos:

- 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.
- 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.
- 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 (por exemplo, nível de isolamento e modo de operação).
- A transação pode voltar ao primeiro passo e especificar os novos limites toleráveis para cada recurso de seu interesse.

Para validar o modelo, foi desenvolvido um protótipo em Java e CORBA [10] e foram simulados alguns casos de testes.

5.6 Comentários sobre os trabalhos relacionados

Conforme citado, existem vários modelos de transações para computação móvel e cada um deles se especializa em algum domínio. Alguns deles podem ser citados: MobileTrans [46], CATE [45] , AMT [1], ReflecTS [5] e SGTA [43], HiCoMo [29], Pro-motion [53] e Prewrite [31]. Alguns desses modelos foram apresentados no capítulo 5 e após uma análise sobre cada um deles, podemos concluir que a maioria deles não provê reconfiguração dinâmica durante a transação.

O modelo MobileTrans permite que o programador especifique uma política de adaptação, porém ela é definida estaticamente, tornando-se uma tarefa complexa. Por sua vez, CATE é baseada em uma arquitetura de componentes. Ela permite a reconfiguração de componentes com relação ao protocolo de efetivação e permite a inclusão de novos protocolos, sendo, portanto, extensível. Porém, cada transação começa e termina com o mesmo protocolo. Não é permitido mudá-lo no meio da transação.

Já o ReflecTS não é um modelo e sim um framework de transações. Ele baseia a sua reconfiguração dinâmica somente na troca de componentes do nível base dificultando assim a realização de pequenas reconfigurações. Não é voltado diretamente para a computação móvel, ou seja, ele não aborda mecanismos dinâmicos de adaptação com relação a ambiente móveis. O que o ReflecTS aborda é a variedade de requisitos transacionais que são exigidos por aplicações de diferentes domínios.

Uma alternativa de execução é definida no AMT no início da execução da transação e a alternativa é cancelada se ocorrerem mudanças durante a execução da transação. Portanto, após a avaliação e escolha da melhor alternativa, não é possível alterá-la diante de mudanças ocorridas em tempo de execução. Um aspecto relevante desse modelo são os descritores de ambientes associados às alternativas de execução, uma idéia que foi adaptada como monitoramento de recursos nessa dissertação.

O último modelo analisado no capítulo 5 foi o SGTA que serviu como base principal dessa dissertação. Ele permite a reconfiguração dinâmica, mas como o próprio trabalho menciona, foi utilizada a tecnologia CORBA [10] para a implementação do protótipo. Apesar de ser uma tecnologia utilizada em ambientes distribuídos, ela não provê facilidade para atender os requisitos dinâmicos das aplicações que executam em um ambiente de computação móvel. Além disso, ela exige uma grande capacidade de memória, um recurso limitado em dispositivos como PDAs.

Capítulo 6

Conclusões

Com o crescente aumento na utilização de dispositivos móveis, aplicações mais sofisticadas vêm sendo desenvolvidas. É cada vez mais comum a existência de aplicações semelhantes às existentes em computadores pessoais. Como grande parte dessas aplicações se conecta a um banco de dados, questões relacionadas ao controle de concorrência e a consistência dos dados se tornam constantes e novos modelos de transações são apresentados na literatura para atender a esse novo ambiente.

Não é uma tarefa fácil propor um modelo de transação genérico que atenda a todos os requisitos transacionais de aplicações para o ambiente móvel, pois existe a necessidade de lidar com o próprio dinamismo do ambiente. Portanto, o propósito dessa dissertação foi demonstrar como algumas propriedades transacionais e alguns mecanismos para garanti-las, podem ser configurados e reconfigurados dinamicamente utilizando um modelo de componentes. Além disso, a configuração dinâmica do controle de concorrência foi uma proposta inovadora em comparação aos modelos de transações existentes. O modelo proposto baseou-se na idéia do modelo SGTA [43], porém utilizando o modelo de componentes OpenCOM por este utilizar a reflexão computacional como técnica de reconfiguração de componentes e utilizando o controle de concorrência como foco principal da arquitetura. Além disso, foi implementado um protótipo para analisar os impactos da reconfiguração durante uma transação.

A reconfiguração de componentes foi uma maneira de permitir a adaptação do modelo diante da variação dos recursos disponíveis, como bateria e conexão. Como no ambiente de computação móvel os recursos são limitados, as configurações de uma transação realizadas no início de sua execução podem deixar de ser adequadas no decorrer da sua execução, devido às mudanças no ambiente.

O capítulo 3 mostrou a arquitetura baseada em componentes do modelo proposto. Duas escolhas foram feitas para mostrar o que pode ser adaptado diante do dinamismo do ambiente. O motivo do nível de isolamento ter sido escolhido é devido ao ganho de desempenho. Quanto mais baixo o nível de isolamento, maior o ganho de desempenho. Como esse ganho pode muitas vezes ocasionar perda de consistência, apenas em algumas situações isso seria desejável. Portanto, essa é uma boa propriedade para ser alterada durante a execução da transação. Além disso, como mostrado num dos casos de uso do capítulo 4, em algumas aplicações e em situações de recursos limitados, pode-se optar por melhor desempenho ao invés de nível de consistência restrito como seriabilidade.

O mecanismo para controle de concorrência também pode ser adaptado diante das mudanças ocorridas no ambiente, porém ele só pôde ser mudado antes de a transação começar. A razão disso é porque quando a transação já iniciou com um determinado controle de concorrência, algumas ações já foram executadas tanto no cliente quanto no servidor. Por exemplo, se o cliente inicia a transação utilizando o controle pessimista, os objetos utilizados na transação ficam bloqueados até o final da mesma. Já o controle otimista exige que sejam feitas cópias locais dos objetos envolvidos na transação. Se durante a transação o controle de concorrência for alterado para otimista, uma cópia dos objetos será feito no cliente e o processo de validação será realizado como determinado

nesse tipo de controle, porém os objetos não serão desbloqueados. Nessa situação, de pessimista para otimista, poderia até ser avaliado como desbloquear os objetos caso o controle de concorrência fosse alterado. Mas se a alteração for de otimista para pessimista, o bloqueio inicial dos objetos não teria sido realizado e uma cópia dos dados desnecessária poderia ter sido feita (lembrando que nesse ambiente largura de banda e custo de comunicação são recursos preocupantes).

Portanto, até seria possível trocar o controle de concorrência durante a transação, mas o controle da arquitetura para garantir que a aplicação não tenha sido prejudicada, seria algo mais complexo que poderia não valer o esforço. Não haveria um ganho de desempenho ou flexibilidade obtidos na alteração do nível de isolamento.

Diante dessa situação, uns dos itens importantes na arquitetura são os componentes que validam a arquitetura para garantir a integridade da aplicação.

Por fim, o monitoramento de recursos foi um item de apoio importante para avisar as transações sobre as mudanças ocorridas no ambiente e permitir que a política definida fosse executada.

6.1 Contribuições

Essa dissertação permitiu verificar que é possível realizar a configuração dinâmica do controle de concorrência. Alterar o controle de concorrência não é algo simples e exige algumas verificações arquiteturais antes de realizar a mudança para que ela não comprometa a aplicação. Além disso, foi possível verificar que durante a transação, é indicado realizar pequenas reconfigurações, como por exemplo, o nível de isolamento.

Uma contribuição significativa foi a implementação do protótipo para a verificação da arquitetura proposta utilizando o caso de uso de vendas. Através dessa implementação foi possível analisar os impactos da reconfiguração antes e durante uma transação.

Concluiu-se que durante a transação, é recomendável realizar apenas pequenas adaptações dinâmicas para evitar instabilidade e não comprometer a corretude da aplicação. Além disso, há a necessidade de um mecanismo de validação arquitetural para garantir que as mudanças foram corretas e não resultarão em erros posteriormente.

OpenCOM é um modelo de componentes adequado para esse tipo de ambiente, pois ele é reflexivo, leve e independente de plataforma. Apesar da necessidade de programação para a conexão dos componentes e para a validação da arquitetura, a versão na linguagem Java simplifica a programação. Uma versão para dispositivos móveis tem sido proposta e ela exclui a dependência com o modelo COM da Microsoft. Essa nova versão será comentada na seção de Trabalhos Futuros.

O nível de isolamento escolhido como propriedade transacional para ser alterada durante a transação foi uma escolha adequada, pois apenas pequenas reconfigurações foram necessárias na arquitetura. Além disso, em algumas aplicações como as citadas no capítulo 4, a leitura inconsistente[2] nem sempre é indesejável no ambiente de computação móvel onde os recursos são críticos.

Notou-se que cada controle de concorrência tem a sua vantagem. Em determinadas situações, como por exemplo, quando a conexão está limitada, o controle otimista pode ser

[2] dirty read

mais adequado, pois o cliente pode trabalhar com os dados localmente. Já em situações em que a largura de banda nem a conexão são tão limitados, o controle pessimista pode ser utilizado, pois evita a cópia de dados o que acarretaria em largura de banda e custo de comunicação excessivos.

6.2 Trabalhos Futuros

Segue uma relação de alguns trabalhos futuros que podem ser desenvolvidos a partir desta tese:

- **Versão do OpenCOM para dispositivos móveis**: desenvolver um protótipo utilizando a versão 2 do OpenCOM [13]. OpenCOM v2 fornece verdadeiramente um modelo de componentes reflexivo independente da plataforma. Essa versão está em fase experimental e remove a dependência com a tecnologia COM da Microsoft. A aplicação de Vendas apresentada no capítulo 4 e o exemplo de *check-in* pelo celular podem ser utilizados como casos de usos para essa nova versão do OpenCOM.

- **Realizar testes em dispositivos com recursos limitados**: realizar testes de desconexão, limitação de bateria e custo de comunicação para comprovar o ganho em alterar o controle de concorrência e nível de isolamento. Verificar o tempo gasto para a reconfiguração de componentes em tempo de execução em dispositivos com capacidades reduzidas, tais como PDAs e celulares.

- **Permitir outros níveis de isolamento**: considerar outros níveis de isolamento como *Read Committed* e *Repeatable Read*. Demonstrar casos de uso que utilizariam tais níveis de isolamento.

- **Nível de isolamento para controle de concorrência otimista:** alguns bancos de dados comerciais trabalham com um nível de isolamento conhecido como *Snapshot* [50]. Esse nível de isolamento é comparável ao *Read Uncommitted*, porém para o controle otimista já que ele trabalha sob a cópia dos dados.

- **Melhorar o mecanismo de validação arquitetural:** para evitar que as reconfigurações comprometam a integridade da plataforma, cada mudança realizada é comparada a um conjunto de arquiteturas válidas pré-definidas. No modelo apresentado, o programador deve definir as regras da arquitetura nos componentes *ClientAccept* e *ServerAccept*, sendo necessário a cada alteração no modelo arquitetural, recompilar o código fonte. Por exemplo, se adicionarmos um novo componente de nível de isolamento, o programador teria que acrescentar as regras de validação da arquitetura no componente *ServerAccept*. Verificar uma forma de validação da arquitetura sem haver a necessidade de recompilação do código, como por exemplo, utilizar um arquivo XML.

- **Política de adaptação**: deve ser um arquivo XML ou uma interface gráfica para facilitar o programador. Na arquitetura proposta no capítulo 3, ela é definida dentro do código fonte, sendo necessário a recompilação do código toda vez que a política de adaptação for alterada.

- **Expandir a arquitetura para protocolos de efetivação**: permitir a adaptação dinâmica de protocolos de efetivação. O protocolo de efetivação mais conhecido é o 2PC, porém em algumas situações poderia ser necessário utilizar algumas de suas otimizações como os protocolos PrC e PrA [28] ou outros protocolos que se adaptem às variações do ambiente móvel [8 e 35]. Os impactos devem ser analisados e a necessidade de reconfigurar os protocolos no início e durante a transação.

- **Testar a arquitetura apresentada na plataforma Motherframe:** essa plataforma foi implementada com base no modelo Motherboard, proposto no trabalho [44]. Esse modelo é implementado como uma camada acima do OpenCOM e permite a adaptação conforme a variedade de requisitos transacionais de aplicações de diferentes domínios. Uma sugestão é testar a reconfiguração de controles de concorrência e níveis de isolamento utilizando a plataforma Motherframe, para verificar a facilidade de configuração e extensão.

Referências Bibliográficas

[1] P. S. Alvarado, C. Roncancio, M. Adiba and C. Labbé. Adaptable Mobile Transactions and Enviromnet Awareness, 2003.

[2] Android. http://www.android.com. Acessado em 05/2009.

[3] ANSI SQL-92. http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt. Acessado em 04/2009.

[4] A. Arntsen and R. Karlsen. Dynamic Transaction Service Composition. University of Tromsoe, 2007.

[5] A. Arntsen and R. Karlsen. ReflecTS; A Flexible Transaction Service Framework. University of Tromsoe, 2005.

[6] F. Backmann et. al. Volume II: Technical Concepts of Component-Based Software Engineering, 2ª edição, 2002.

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

[8] C. Bobineau, P. Pucheral and M. Abdallah. A unilateral commit protocol for mobile and disconnected computing. Proc. of the 12th int. on parallel and distributed computing Systems (PDCS), Las Vegas, USA.

[9] COM: Component Object Model Architectures. Microsoft Corporation. http://www.microsoft.com/com/default.mspx. Acessado em 04/2009.

[10] CORBA - Common Object Request Broker Architecture. http://www.corba.org. Acessado em 05/2009.

[11] comScore Inc. Mobile Internet Becoming A Daily Activity For Many. Pesquisa realizada em Março de 2009.
http://www.comscore.com/Press_Events/Press_Releases/2009/3/Daily_Mobile_Internet_Usage_Grows. Acessado em 03/2009.

[12] G. Coulouris, J. Dollimore and T. Kindberg. Distributed Systems: Concepts and Design. Third edition. Addison-Wesley, 2001.

[13] G. Coulson, G.S. Blair, P. Grace, A. Joolia, K. Lee and J. Ueyama. OpenCOM v2: A Component Model for Building Systems Software. Proceedings of IASTED Software Engineering and Applications, Cambridge, MA, ESA, 2004.

[14] I. Crnkovic. Building Reliable Component-Based Software Systems. Artech House, Inc., Norwood, MA, USA, 2002.

[15] M. Dunham e A. Helal. Mobile Computing and Databases: Anything New?. ACM SIGMOD Record, Vol. 24, No. 4, Dezembro de 1995.

[16] EDUCAUSE. http://www.educause.edu/. Acessado em 04/2009.

[17] J. Ferber. Computational reflection in class based object-oriented languages. In OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications. New York, USA, 1989. ACM Press.

[18] M. Flatt. Programming Languages for Reusable Software Components. PhD thesis, Rice University, Houston, EUA, 1999.

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

[20] Fractal. http://fractal.ow2.org/documentation.html. Acessado em 05/2009.

[21] D. Georgakopoulos, M. Rusinkiewicz and A. Sheth. Using Tickets to Enforce the Serializability of Multidatabase Transactions. 1993.

[22] T. Imielinski e B. R. Badrinath. Mobile Wireless Computing: Challenges in Data Management. Communications of ACM, Vol. 37, No. 10, Outubro de 1994.

[23] Isolation (database systems). Wikipedia. The Free Encyclopedia. http://en.wikipedia.org/wiki/Isolation_(database_systems). Acessado em 04/2009.

[24] Java Sun Microsystems. http://java.sun.com/. Acessado em 04/2009

[25] Java Micro Edition. Java Sun Microsystems. Acessado em 04/2009. http://java.sun.com/javame/index.jsp

[26] L. Johnson, A. Levine and R. Smith. The 2009 Horizon Report. Austin, Texas: The New Media Consortium, 2009.

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

[28] B. Lampson and D. Lomet. A New Presumed Commit Optimization for Two Phase Commit. Proc. of 14th Intl. Conf. on Very Large Databases, 1993.

[29] M. Lee and S. Helal. Hicomo: High commit mobile transactions. Kluwer Academic Publishers Distributed and Parallel Databases, 2002.

[30] M. Lisbôa. Arquiteturas de meta-nível. Tutorial XI Simpósio Brasileiro de Engenharia de Software. Fortaleza, 1997.

[31] S. Madria and B. Bhargava. A transaction model to improving data availability in mobile computing. Proceeding Distributed and Parallel Databases, 2001.

[32] P. Maes. Concepts and experiments in computational reflection. In OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications. New York, USA, 1987. ACM Press.

[33] E. A. Nassu. Consultas sobre "aqui" em Sistemas de Bancos de Dados em Ambientes de Computação Nômade. Tese de Doutorado - IME-USP, Orientador: Prof. Dr. Marcelo Finger, Junho de 2003.

[34] E. A. Nassu and M. Finger. O Significado de "Aqui" em Sistemas Transacionais Móveis. IME-USP, 2001.

[35] N. Nouali, A. Doucet and H. Drias. A Two-Phase Commit Protocol for Mobile Wireless Environment. Proceedings of the sixteenth Australasian database conference, Australia, 2005.

[36] OpenCOM. Computer Departament of Lancaster University. http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/opencom.php. Acessado em 04/2009.

[37] E. Pitoura and G. Samaras. Data Management for Mobile Computing. Summer School on Mobile Computing, Jyvaskyla, 1998.

[38] R. Ramakrisnan. and J. Gehrke. Database Management Systems. McGraw Hill, 3ª edição, 2003.

[39] ReMMoC. Computer Departament of Lancaster University. http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/remmoc.php. Acessado em 04/2009.

[40] Remote Method Invocation Home. Java Sun Microsystems. Acessado em 04/2009. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp

[41] T. Rocha and M. B. F. Toledo. Estudo de Modelos de Transação para o Ambiente de Computação Móvel. Relatório Técnico. IC-05-025. Instituto de Computação – UNICAMP. Setembro de 2005.

[42] T. Rocha and M. B. F. Toledo. Um Sistema de Transações Adaptável para o Ambiente de Computação Móvel. Simpósio Brasileiro de Redes de Computadores, 2003.

[43] T. Rocha. Um Sistema de Transações Adaptável para o Ambiente de Comunicação Sem Fio. Tese de Mestrado, Instituto de Computação da UNICAMP, Orientadora: Maria Beatriz Felgar de Toledo, 2004.

[44] T. Rocha. Serviços de Transações Abertos para Ambientes Dinâmicos. Tese de Doutorado, Instituto de Computação da UNICAMP, Orientadora: Maria Beatriz Felgar de Toledo, 2008.

[45] R. Rouvoy , P. S. Alvarado and M. Philippe. Towards Context-Aware Transaction Services, 2006.

[46] N. Santos and P. Ferreira. Making Distributed Transactions Resilient to Intermittent Network Connections. Instituto Superior Técnico de Lisboa, 2006.

[47] N. Santos, L. Veiga and P. Ferreira. Transaction Policies for Mobile Networks, Instituto Superior Técnico de Lisboa, 2004.

[48] M. Satyanarayanan. Fundamental Challenges in Mobile Computing. Fifteenth ACM Symposium on Principles of Distributed Computing, Philadelphia, PA, Maio de 1996.

[49] B. Smith. Reflection and Semantics in a Procedural Language. PhD thesis, Cambridge, Massachusetts, 1982.

[50] Snapshot Isolation. Wikipedia. The Free Encyclopedia. Acessado em 05/2009. http://en.wikipedia.org/wiki/Snapshot_isolation

[51] C. Szyperski. Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 1998.

[52] The OpenCOMJ Handbook. Lancaster Univerty. Febreary, 2007. http://www.mirrorservice.org/sites/download.sourceforge.net/pub/sourceforge/g/gr/gridkit/OpenCOMJHandbook.pdf

[53] G. Walborn and P. Chrysanthis. Transaction processing in pro-motion. Proceedings of the 1999 ACM symposium on Applied computing. New York, 1999.

[54] XML Tutorial. W3Schools. Acessado em 04/2009. http://www.w3schools.com/xml/default.asp

Apêndice A

Montagem das Arquiteturas

A classe *ArchitectureServer* contém um método construtor (método chamado automaticamente no momento da criação do objeto da classe) que cria a arquitetura básica do lado servidor, tanto para o controle pessimista quanto para o otimista. O código fonte abaixo mostra como os componentes OpenCOM são criados e conectados.

```
// Declaração das variáveis
private OpenCOM openCOM;
private IUnknown serverFramework;
private ICFMetaInterface serverMetaInterface;
private int concurrencyControl = 0;
private long bindIObjectServer = 0;
private long bindIServerTrans = 0;
private long bindIResourceManager = 0;
private long bindIIsolationLevel = 0;
private IUnknown objectServer = null;
private IUnknown lockManager = null;
private IUnknown versionManager = null;
private IUnknown resourceManager = null;
private IUnknown serverCommunicator = null;
private IUnknown serializable = null;
private IUnknown readUncommitted = null;
private final int READ_UNCOMMITTED = 0;
private final int SERIALIZABLE = 1;

public ArchitectureServer(int port) {

        // Cria ambiente de execução do OpenCOM e expõe a interface IOpenCOM
        openCOM = new OpenCOM();
        IOpenCOM iOpenCOM = (IOpenCOM)
                openCOM.QueryInterface("OpenCOM.IOpenCOM");

        // Cria framework do servidor
        serverFramework = (IUnknown)
        iOpenCOM.createInstance("server.TMServerFramework", "TMServerFramework");
```

```
// Cria interface ILifeCycle que provê operações para criar e destruir um componente
ILifeCycle iLifeCycle = (ILifeCycle)
        serverFramework.QueryInterface("OpenCOM.ILifeCycle");
iLifeCycle.startup(iOpenCOM);  // Cria e inicia o componente OpenCOM

// Cria componente ServerAccept responsável pela validação da arquitetura
IUnknown serverAccept = (IUnknown)
        iOpenCOM.createInstance("server.ServerAccept", "ServerAccept");
iLifeCycle = (ILifeCycle) serverAccept.QueryInterface("OpenCOM.ILifeCycle");
iLifeCycle.startup(iOpenCOM);

// Conecta os componentes ServerFramework e ServerAccept
long connID1 =
        iOpenCOM.connect(serverFramework, serverAccept, "OpenCOM.IAccept");

// Cria interface ICFMetaInterface que permite a reflexão
serverMetaInterface = (ICFMetaInterface)
        serverFramework.QueryInterface("OpenCOM.ICFMetaInterface");

serverMetaInterface.init_arch_transaction(); //  Inicia a transação do OpenCOM

// Cria componente ObjectServer
objectServer =
serverMetaInterface.create_component("server.ObjectServer", "ObjectServer");

// Cria componente Resource Manager
resourceManager =
        serverMetaInterface.create_component("server.ResourceManager",
        "ResourceManager");

// Cria componente  ServerCommunicator
serverCommunicator =
        serverMetaInterface.create_component("server.ServerCommunicator",
        "ServerCommunicator");


ServerCommunicator robj = new ServerCommunicator(serverCommunicator, false);

 /* UnicastRemoteObject exporta o objeto para que o mesmo possa ser utilizado pelo
cliente */
IRMICommunication stub =
         (IRMICommunication) UnicastRemoteObject.exportObject(robj, port);

// Registra e recupera referências a objetos através do nome
Registry registry = LocateRegistry.createRegistry(port);
registry.rebind("IRMICommunication", stub);

// Estabelece conexão entre os componentes ResourceManager e ObjectServer
```

```
        bindIObjectServer = serverMetaInterface.local_bind(resourceManager, objectServer,
                                                        "server.IObjectServer");

        // Estabelece conexão entre os componentes ResourceManager e ObjectServer

        bindIResourceManager = serverMetaInterface.local_bind(serverCommunicator,
                                    resourceManager, "server.IResourceManager");

        // Expõe interface IResourceManager
        boolean exposeResult =
                serverMetaInterface.expose_interface("server.IResourceManager",
                                                        resourceManager);

        // Final da transação OpenCOM
         boolean success = serverMetaInterface.commit_arch_transaction();

         if (!success) {
                throw new RuntimeException("Controle Pessimista –
                                            Arquitetura do Servidor inválida!");
        }

        // Ferramenta de depuracao da arquitetura montada
         IDebug pIDebug = (IDebug) openCOM.QueryInterface("OpenCOM.IDebug");

        // Exibe através de uma interface gráfica os componentes e suas conexões
         pIDebug.visualise();
}
```

O método *pessimisticConcurrencyControl* é chamado dentro do componente *ArchitectureServer* para montar a arquitetura pessimista do lado Servidor. Segue um trecho do código fonte demonstrando como a programação foi realizada utilizando os componentes do OpenCOM.

```
private void pessimisticConcurrencyControl() {

        serverMetaInterface.init_arch_transaction(); // Início da transação OpenCOM

        // Se componente LockManager não existir, ele deve ser criado
        if (lockManager == null) {
                lockManager = serverMetaInterface.create_component
                                    ("server.LockManager", "LockManager");
        }

        // Remove a conexão entre os componentes ResourceManager e VersionManager
         serverMetaInterface.break_local_bind(bindIServerTrans);
```

```
        // Se componente VersionManager (controle otimista) existir, ele deve ser removido
        if (versionManager != null) {
                serverMetaInterface.delete_component(versionManager);
        }

        // Estabelece a conexão entre os componentes ResourceManager e LockManager
        bindIServerTrans = serverMetaInterface.local_bind(resourceManager, lockManager,
                                                        "server.IServerTransactionControl");

        /* Se os componentes do nível de isolamento não existirem, eles devem ser criados e um
        deles (Serializable) conectado ao ResourceManager */
        if (serializable == null) {
                serializable = serverMetaInterface.create_component
                                        ("server.Serializable", "Serializable");
                readUncommitted =serverMetaInterface.create_component
                                        ("server.ReadUncommitted", "ReadUncommitted");
                bindIIsolationLevel = serverMetaInterface.local_bind(resourceManager,
                                        serializable, "server.IIsolationLevel");
        }

        // Final da transação OpenCOM
        boolean success = serverMetaInterface.commit_arch_transaction();

        if (!success) {     // Problema ao criar a arquitetura do lado servidor
                throw new RuntimeException("Tipo Pessimista - Arquitetura do servidor está
                                                                        inválida!");
        }
}
```

O lado cliente do controle pessimista segue a mesma idéia de montagem de arquitetura do lado servidor. Já para o controle de concorrência otimista, a lógica também é semelhante, porém com alguns componentes diferentes que somente representam esse tipo de controle. Além disso, os componentes relacionados ao nível de isolamento não são conectados, já que nessa arquitetura só existe nível de isolamento para o controle de concorrência pessimista.