Serviços de Transação Abertos para Ambientes Dinâmicos

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

Campinas, 12 de Agosto de 2008.

Maria Beatriz Felgar de Toledo (Orientadora)

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

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Júlia Milani Rodrigues CRB8a / 2116

Rocha, Tarcisio da

R582s Serviços de transação abertos para ambientes dinâmicos / Tarcisio da Rocha -- Campinas, [S.P. :s.n.], 2008.

Orientador : Maria Beatriz Felgar de Toledo

Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Computação móvel. 2. Sistemas de transação (Sistemas de computação). 3. Middleware. 4. Sistemas distribuídos. I. Toledo, Maria Beatriz Felgar de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Open transaction services for dynamic environments.

Palavras-chave em inglês (Keywords): 1. Mobile computing. 2. Transaction systems (Computer systems). 3. Middleware. 4. Distributed systems.

Área de concentração: Sistemas Distribuídos

Titulação: Doutor em Ciência da Computação

Banca examinadora:

Profa. Dra. Maria Beatriz Felgar de Toledo (IC-UNICAMP)
Prof. Dr. Markus Endler (PUC-Rio)
Profa. Dra. Itana Maria de Souza Gimenez (UEM)
Prof. Dr. Edmundo Roberto Mauro Madeira (IC-UNICAMP)
Profa. Dra. Islene Calciolari Garcia (IC-UNICAMP)
Prof. Dr. Marcelo Fantinato (Motorola)
Profa. Dra. Claudia Bauzer Medeiros (IC-UNICAMP)

Data da defesa: 12/08/2008

Programa de pós-graduação: Doutorado em Ciência da Computação

TERMO DE APROVAÇÃO

Tese Defendida e Aprovada em 12 de agosto de 2008, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Markus Endler
DI – PUC-RIO.

Profª. Drª. Itana Maria de Souza Gimenes
DI – UEM.

Prof. Dr. Edmundo Roberto Mauro Madeira
IC – UNICAMP.

Profª. Drª. Islene Calciolari Garcia
IC – UNICAMP.

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

Instituto de Computação
Universidade Estadual de Campinas

Serviços de Transação Abertos para Ambientes Dinâmicos

Tarcisio da Rocha[1]

Agosto de 2008

Banca Examinadora:

- Maria Beatriz Felgar de Toledo (Orientadora)
- Markus Endler
  PUC-RIO
- Itana Maria de Souza Gimenes
  UEM – Universidade Estadual de Maringá
- Edmundo Roberto Mauro Madeira
  IC-UNICAMP – Universidade Estadual de Campinas
- Islene Calciolari Garcia
  IC-UNICAMP – Universidade Estadual de Campinas
- Marcelo Fantinato (Suplente)
  Motorola
- Claudia Bauzer Medeiros (Suplente)
  IC-UNICAMP – Universidade Estadual de Campinas

[1]**Esta tese recebeu apoio financeiro da CAPES e do CNPq.**

Dedicatória

À minha mãe Carolina.

Agradecimentos

À minha esposa Shirley por todo amor e por ter sido a minha fiel companheira em todos os momentos que envolveram esse trabalho. Esta é uma conquista nossa, minha morena! Te amo!

À minha mãe Carolina pelos exemplos de vida, de doação e de incentivo constante aos meus estudos. Devo a ela a minha formação.

À minha orientadora Beatriz por todo o apoio, e experiência o e por ter me conduzido durante todo esse trabalho de maneira humana e acolhedora.

Aos pesquisadores da Universidade de Tromso (Noruega), Randi Karlsen, Anna-Brith Arntsen e Arne Eidsvik pelas discussões e contribuições.

Às amigas Karianne e Juliane pelo apoio e carinho recebidos na Noruega.

Aos meus familiares e irmãos Adriano, Marcos, Suzana, Miguel, Rita, Sérgio, Lucas, Raquel e Raimundo por compartilharem comigo mais esta conquista.

Aos funcionários e professores do Instituto de Computação da UNICAMP, pelo apoio recebido durante o desenvolvimento desta pesquisa.

Aos professores Luiz Brunelli e Sergey Zybin da Universidade Federal de Sergipe pelo incentivo e confiança depositados.

Aos meus amigos de Aracaju, Saulo, Gildo, Marcos, Josivan e Edson pela amizade e companheirismo.

Ao meu sogro Teles e minha sogra Simonete pelo apoio e carinho recebidos.

À minha avó Cantionila e à minha tia Ana pela presença e pela alegria transmitida.

À CAPES e ao CNPq pelo apoio financeiro.

Resumo

Técnicas de processamento de transações têm sido de grande importância no que diz respeito à preservação da correção em diversas áreas da computação. Devido a funções como, garantir a consistência de dados, a recuperação de falhas e o controle de concorrência, transações são consideradas blocos de construção apropriados para a estruturação de sistemas confiáveis.

Contudo, desenvolver técnicas de apoio a transações para ambientes dinâmicos pode ser uma tarefa complexa. O primeiro obstáculo está no próprio dinamismo – a disponibilidade de recursos pode variar inesperadamente. Isso pode causar dois efeitos diretos: altas taxas de cancelamento de transações e grandes atrasos na execução das tarefas transacionais. O segundo obstáculo está na crescente flexibilização do conceito de transação. Isso ocorre porque os requisitos transacionais exigidos pelas aplicações atuais estão se tornando mais variados, indo além das propriedades tradicionalmente definidas para uma transação.

Nesse contexto, esta tese aborda a viabilização de serviços de transações abertos, ou seja, capazes de terem sua estrutura e comportamento configurados pelos programadores de aplicações como um meio de atender a requisitos específicos do domínio de suas aplicações. Como parte desse estudo foi proposto um modelo que abstrai alguns elementos arquiteturais como jumpers, slots e demultiplexadores que podem ser usados na especificação de pontos de configuração em serviços de transação. Esse modelo é implementado como uma camada acima de um modelo de componentes existente. Com isso, desenvolvedores de serviços de transação passam a contar com esses elementos abertos além daqueles disponibilizados por abordagens tradicionais baseadas em componentes.

Para confirmar os benefícios em usabilidade, flexibilidade e extensão, esta tese apresenta dois serviços de transação abertos que foram especificados com base no modelo proposto. O primeiro serviço faz parte de uma plataforma de transações adaptável para ambientes de computação móvel. O segundo serviço faz parte de um sistema que provê adaptação dinâmica de protocolos de efetivação (*commit*) de transações. Segundo os testes realizados, a abordagem apresentada nesta tese trouxe a esses serviços a capacidade de atender requisitos de aplicações de diferentes domínios.

Abstract

Transaction processing techniques are considered important solutions on preserving correctness in several fields of computing. Due their functions such as, failure recovery and concurrency control, transactions are considered appropriated building blocks for structuring reliable systems.

Despite its advantages, to develop transaction systems for dynamic environments is not an easy task. The first problem is the dynamism – the resource availability can vary unexpectedly. This can cause the following side effects: high transaction abort rates and relevant delays of transaction operations. The second problem is the flexibilization of the transaction concept. The transactional requirements are becoming more diversified – they extrapolate the bounds of the traditional transactional properties.

In this context, this thesis approaches the practicability of open transaction services that can be configured by the application programmers for attending specific requirements of different application domains. This thesis includes a model that abstracts some architectural elements (slots, jumpers and demultiplexers) that can be used for specifying configuration points in transaction services.

To confirm its benefits on usability, flexibility and extension, this thesis presents two open transaction services that were specified based on the proposed model. The first service is part of an adaptable transaction platform for mobile computing environments. The second service is part of a system that provides dynamic adaptation of commit protocols. According the accomplished tests, the approach presented in this thesis is able to give to these services the capacity of attending the requirement of applications in different domains.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Esta introdução inicia apresentando as motivações deste trabalho. Em seguida, serão apresentadas as principais contribuições e a estrutura deste documento.

1.1 Motivação

Técnicas de processamento de transações têm sido de grande importância no que diz respeito à preservação da correção[1] em diversas áreas da computação. Devido a funções como, garantir a consistência de dados, a recuperação de falhas e o controle de concorrência, transações são consideradas blocos de construção apropriados para a estruturação de sistemas confiáveis [80].

Dentre as áreas que mais têm influenciado o estudo de novas técnicas de processamento de transações, a computação móvel merece uma posição de destaque. Apesar da sua crescente popularidade por trazer a computadores portáteis a capacidade de se tornarem onipresentes, a computação móvel trouxe consigo uma série de novos desafios ao processamento de transações [102]. Esses desafios são causados principalmente pelo seu alto dinamismo que é refletido na prática por: desconexões inesperadas, erros de transmissão, variações na largura de banda e escassez de recursos nos dispositivos de computação portáteis. Diante desse dinamismo, transações também agregam o importante papel de garantir que as variações ocorridas não comprometam a confiabilidade das aplicações.

Contudo, desenvolver técnicas de apoio a transações para ambientes dinâmicos é uma tarefa complexa. O primeiro obstáculo está no próprio dinamismo dos ambientes – a disponibilidade de recursos pode variar inesperadamente. Isso pode causar dois efeitos diretos no processamento de transações: altas taxas de cancelamento de transações e grandes atrasos na execução das tarefas transacionais. O segundo obstáculo está na at-

[1] Em inglês: correctness

ual flexibilização conceito de transações. Isso ocorre porque os requisitos transacionais exigidos pelas aplicações atuais estão tão variados que extrapolam as propriedades tradicionalmente definidas para uma transação[2].

A complexidade em se projetar serviços de transações para esses ambientes não está somente em encontrar formas de superar esses obstáculos, mas principalmente no fato de que essas formas podem ser dependentes das aplicações que utilizam os serviços. Por exemplo, considerando o caso das desconexões, o interessado em determinar quais serão as medidas adotadas em caso de uma interrupção abrupta causada por uma desconexão inesperada é o desenvolvedor da aplicação e não o desenvolvedor do serviço de transação por si só. O problema é que as medidas para lidar com o dinamismo e as propriedades transacionais requeridas pelas aplicações podem ser tão diferentes que é impossível implementar um serviço de transação que consiga contemplar todas as possíveis alternativas exigidas pelas aplicações.

Uma das conseqüências desse problema é o desenvolvimento de vários serviços de transação diferentes para esses ambientes, cada um adotando soluções específicas para os obstáculos mencionados anteriormente. Isso reduz a aplicabilidade dos serviços e os tornam incompatíveis entre si.

1.2 Contribuições desta Tese

Esta tese aborda a viabilização de serviços de transações abertos, ou seja, capazes de terem sua estrutura e comportamento configurados pelos programadores de aplicações como um meio de atender a requisitos específicos do domínio de suas aplicações. Como parte desse estudo foi proposto um modelo que abstrai alguns elementos arquiteturais que podem ser usados na especificação de pontos de configuração em serviços de transação. Esse modelo foi incorporado a um ambiente de execução implementado como uma camada acima de um modelo de componentes existente. Com isso, desenvolvedores de serviços de transação passam a contar com esses elementos abertos além daqueles disponibilizados por abordagens tradicionais baseadas em componentes. Além disso, esta tese apresenta dois serviços de transação abertos que foram especificados com base no modelo proposto. Apesar do foco principal desta tese ser serviços de transações, suas contribuições também podem ser aplicadas no desenvolvimento de outros serviços de middleware.

Para alcançar suas metas, a elaboração dessa tese contou com experiências obtidas a partir do estudo de diversos serviços de transação para ambientes de computação móvel e de frameworks para a definição de transações. Além disso, foram explorados os conceitos de middlewares abertos e de reflexão computacional para a definição do modelo e

[2] *atomicidade*, *consistência*, *isolamento* e *durabilidade*.

do ambiente de execução. Esses conceitos forneceram bases para permitir que os serviços de transação definidos tivessem também a capacidade de serem reconfigurados dinamicamente. A capacidade de reconfiguração dinâmica é um requisito indispensável em ambientes sujeitos a variações em tempo de execução, como no caso da computação móvel.

A seguir são destacados alguns pontos relevantes do ambiente de execução implementado:

1. permite a definição de serviços de transação abertos capazes de serem configurados estática ou dinamicamente para atender os requisitos das aplicações e para se adaptar às variações ocorridas no ambiente.

2. facilita o processo de configuração de serviços de transação através da separação de interesses entre os desenvolvedores do serviço e dos usuários do serviço (desenvolvedores de aplicações). Enquanto o desenvolvedor do serviço define detalhes de sua arquitetura e comportamento, o usuário enfoca principalmente os seus pontos de configuração.

3. permite a definição e documentação de diferentes tipos de configuração que podem ser aplicados a um mesmo serviço de transação. Cada configuração pode ser armazenada em um arquivo XML para ser selecionada por usuários de acordo com seus interesses.

4. facilita a extensão de serviços de transação. Ao especificar serviços de transações a partir de elementos arquiteturais abertos, os desenvolvedores são impelidos a pensar e a estruturar arquiteturas de forma extensível.

5. promove o reuso de componentes que implementam serviços de transação. Para definir arquiteturas abertas, o desenvolvedor de serviços tem que identificar e separar as partes que serão implementadas como componentes individuais. Essas partes podem ser reusadas em diversas outras configurações do serviço de transação.

6. permite a definição de serviços de transação autoconfiguráveis – que são capazes de acessar seu próprios pontos de configuração e manipulá-los em tempo de execução.

7. permite a proteção de partes críticas de serviços de transação contra a inclusão de código malicioso ou contra o mau uso das suas capacidades de configuração pelos usuários. Os usuários possuem permissão de manipular somente os pontos em abertos que foram previamente definidos pelo desenvolvedor do serviço.

8. permite a definição e configuração de arquiteturas tanto de forma declarativa – através de especificações XML – quanto de forma programática – através do uso

de uma API. Enquanto a forma declarativa provê a vantagem de não precisar de recompilação de código, a forma programática é capaz de oferecer um melhor desempenho.

1.3 Organização deste Documento

Este documento está organizado da seguinte forma:

Capítulo 2. Apresenta os principais conceitos que fundamentam esta tese. Nesse capítulo serão descritos: *(i)* os ambientes de computação móvel ressaltando os aspectos que influenciam o projeto de serviços de transação; *(ii)* o conceito tradicional de transações e os novos requisitos de flexibilização de suas propriedades; *(iii)* os conceitos de reflexão computacional, componentes de software e framework de componentes – que são técnicas usadas para se implementar o ambiente de execução proposto.

Capítulo 3. Apresenta grupos de trabalhos que forneceram experiências ao desenvolvimento desta tese. Primeiramente são discutidos alguns padrões tradicionais para transações distribuídas. Em seguida serão apresentados trabalhos relevantes no campo de transações para ambientes de computação móvel. Na seqüência serão apresentados alguns frameworks que propõem a flexibilização do conceito de transação. Por fim, serão discutidos trabalhos relacionados com middleware aberto.

Capítulo 4. Apresenta o novo modelo proposto nesta tese para permitir a definição de serviços de transação abertos. Serão apresentadas algumas limitações de soluções em middleware aberto e as contribuições desse modelo.

Capítulo 5. Apresenta a plataforma que provê o ambiente de execução para serviços de transação abertos. Essa plataforma implementa o modelo proposto nesta tese.

Capítulo 6. Propõe duas soluções em serviços de transações abertos. Esses serviços foram especificados e implementados utilizando o modelo e a plataforma propostos. Para cada serviço especificado são apresentadas duas opções de configuração: cada uma honra os requisitos de uma aplicação diferente.

Capítulo 7. Apresenta resultados experimentais de testes de desempenho que foram aplicados à plataforma. Esses testes medem o impacto no desempenho que podem sofrer os serviços de transação quando configurados a partir dos elementos abertos definidos pelo modelo proposto.

Capítulo 8. Apresenta as conclusões desta tese e os trabalhos futuros que podem ser realizados a partir dos seus resultados.

Capítulo 2

Fundamentos

2.1 Computação Móvel

A computação móvel é um paradigma que permite que computadores portáteis equipados com interfaces de comunicação sem fio participem de uma computação distribuída independentemente de localização física ou características de movimento. Isto se tornou possível graças aos avanços nas tecnologias de telecomunicação, redes e dispositivos de computação portáteis.

2.1.1 Arquitetura da Computação Móvel

A Figura 2.1 mostra um modelo de arquitetura de apoio à computação móvel normalmente apresentado pela literatura [102, 75, 31]. Esse modelo é composto basicamente por Máquinas Fixas (MF), Estações de Apoio(ES) e Unidades Móveis(UM). Esses elementos podem ser definidos como a seguir.

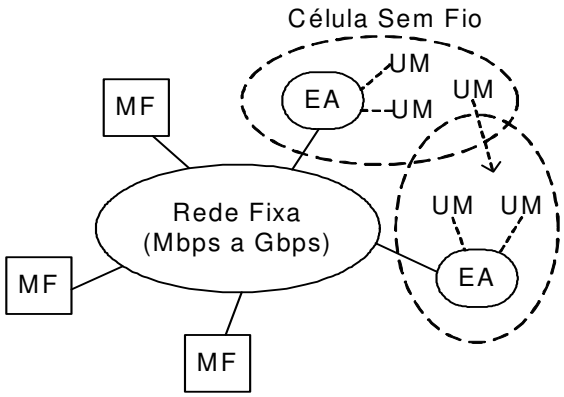

Figura 2.1: Ambiente de Computação Móvel

Unidade Móvel computador portátil capaz de se comunicar com a rede fixa através de uma interface de comunicação sem fio usando a Estação de Apoio como ponto de acesso.

Máquina Fixa computador da rede fixa que não possui interface de comunicação sem fio.

Estação de Apoio computador da rede fixa capaz de estabelecer comunicação com unidades móveis através de um meio de comunicação sem fio.

Cada estação de apoio possui uma área de cobertura chamada *célula sem fio*. Essa é a área por onde uma unidade móvel pode se mover e manter a conexão com a estação de apoio correspondente. Ao se mover, uma unidade móvel poderá sair de uma célula e entrar em uma outra e manter a comunicação com a rede fixa através de outra estação de apoio.

2.1.2 Obstáculos da Computação Móvel

A computação móvel tem exigido várias mudanças no desenvolvimento de soluções no que diz respeito a circuitos integrados, processamento de sinais, projeto de rede e projeto de sistemas computacionais. Essas mudanças são necessárias porque a computação móvel introduziu uma série de novos obstáculos normalmente não existentes nos ambientes tradicionais. Alguns desses obstáculos são citados a seguir [102, 91, 50, 34, 44]:

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

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

Escassez de recursos. Recursos como, energia, espaço em disco e capacidade de processamento podem ser escassos em dispositivos móveis. Em particular, o suprimento de energia dos dispositivos móveis normalmente é feito por baterias que possuem uma capacidade limitada de suprimento. Isto pode exigir técnicas para a redução do consumo de energia por parte dos elementos de hardware e software.

Segurança. A segurança da comunicação sem fio é bem mais vulnerável do que na comunicação com fio, pois o sinal da comunicação sem fio pode ser facilmente interceptado por outros dispositivos de captação intrusos. As soluções adotadas para lidar com

esse problema são a criptografia e a autenticação feitas por software ou hardware especializado.

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

Maior dinamismo. Informações que são consideradas estáticas para um computador estacionário poderão passar a ser dinâmicas para um computador móvel. Um exemplo desse dinamismo pode ser o endereço de rede do computador móvel que poderá mudar cada vez que houver uma mudança no ponto de acesso desse com a rede fixa.

2.1.3 Discussão

Os obstáculos introduzidos pelo dinamismo de ambientes de computação móvel têm exigido novas soluções em diversas áreas que envolvem o processo de desenvolvimento de sistemas. Uma dessas áreas é a de serviços de transação. Serviços de transação tradicionais, quando aplicados em ambientes de computação móvel, apresentam um baixo desempenho e altas taxas de cancelamento causadas pelas desconexões e pelo mau uso dos escassos recursos disponíveis. Por isso, vários serviços de transações projetados especificamente para esses ambientes têm sido propostos [90, 59, 94, 92]. Esses serviços serão descritos com mais detalhes nos capítulos a seguir.

2.2 Transações

Transação é um conceito importante para a simplificação da construção de aplicações confiáveis, especialmente aquelas que requerem acesso concorrente a dados compartilhados. O conceito de transações foi desenvolvido primeiramente junto a aplicações comerciais como um mecanismo de proteção aos dados em bases centralizadas. Posteriormente, o conceito de transações foi estendido para o mais amplo contexto da computação distribuída. Atualmente, há uma grande aceitação de transações como um elemento chave para a construção de aplicações confiáveis em novas áreas como, serviços web, comércio eletrônico e computação móvel.

2.2.1 Propriedades Tradicionais de Transações

Tradicionalmente, uma transação pode ser definida como um conjunto de operações que é executado honrando as propriedades ACID [40]. ACID é um acrônimo para Atomici-

dade, Consistência, Isolamento e Durabilidade. Segue a descrição de cada uma destas propriedades:

Atomicidade. O conjunto de operações de uma transação deverá constituir uma unidade indivisível, ou seja, ou todas operações são executadas ou nenhuma das operações é executada. Isto implica que, se a transação for interrompida por uma falha, todos os seus efeitos sobre os itens manipulados têm que ser desfeitos.

Consistência. Uma transação deve manter as regras de integridade dos dados que ela acessa.

Isolamento. Os resultados de uma transação não deverão sofrer interferências de outras transações concorrentes. Se duas ou mais transações estão executando concorrentemente e de forma isolada, o resultado final obtido será o mesmo se elas fossem executadas uma após a outra em alguma ordem.

Durabilidade. Assim que uma transação for efetivada, seus resultados se tornam permanentes, ou seja, nenhuma falha depois da efetivação poderá desfazer esses resultados.

Devido ao seu conceito simples, rigoroso e eficaz, as propriedades ACID são muito usadas em operações de sistemas de banco de dados. Modelos de transação que seguem estas propriedades com rigor são adequados para aplicações que fazem transformações de estado simples, como realizar uma transferência de valores entre duas contas bancárias. Entretanto, a aplicação rígida das propriedades ACID impõe limitações que tornam o modelo inadequado para outros tipos de aplicações.

2.2.2 Relaxamento das Propriedades Transacionais

Vários outros modelos de transação que relaxam estas propriedades foram propostos na literatura. Esses modelos têm por objetivo melhor atender aos requisitos de aplicações avançadas. Por exemplo, transações aninhadas [71] relaxam a propriedade da atomicidade. Uma contribuição das transações aninhadas é evitar a necessidade de desfazer a transação por completo se algo der errado durante a sua execução. Para isso, uma transação é dividida em subtransações e cada subtransação pode ser desfeita sem que a transação seja desfeita por completo.

Um outro exemplo do relaxamento das propriedades ACID é o uso de níveis de isolamento [40, 39]. Para a definição de níveis de isolamento, graus de interferência entre transações foram definidos e vão desde um isolamento total (sem interferências) até a ausência de isolamento (permite todo tipo de interferência). O uso de níveis de isolamento é muito útil para melhorar o desempenho de aplicações que não requerem uma

garantia total de isolamento. Isto se deve ao fato de que as restrições impostas para se garantir o isolamento total de transações podem acarretar grandes perdas de desempenho [1].

Na literatura também há exemplos de modelos de transações que relaxam as propriedades da consistência e da durabilidade [39, 59]. Uma forma de relaxar a propriedade da consistência é permitir que as aplicações definam regras de consistência personalizadas para o acesso a dados. Uma forma de relaxar a propriedade de durabilidade é tolerar o descarte de parte de atualizações de dados no processo de efetivação de transações.

2.2.3 Discussão

Diante da variedade dos ambientes em que transações são executadas e da variedade dos requisitos transacionais das aplicações destinadas a esses ambientes, pode-se dizer que, ao implementar um serviço de transações, poder-se-ia adotar uma das duas estratégias:

(i) o serviço seria implementado levando em conta somente os mecanismos necessários para atender aos requisitos comuns de aplicações de alguns domínios correlatos;

(ii) o serviço seria implementado com um grande número de mecanismos capazes de atender aos requisitos de aplicações do maior número possível de domínios.

Cada uma dessas duas estratégias possui suas desvantagens. Uma das desvantagens da primeira estratégia é que o serviço poderá se tornar muito específico e incapaz de atender aos requisitos de diferentes tipos de aplicações. Uma das desvantagens da segunda estratégia é que nenhuma aplicação necessitará de todos mecanismos disponibilizados pelo serviço, mas, mesmo assim, pagará por eles (ex.: grande porte, custo monetário alto, baixo desempenho).

Uma resposta para esse problema pode estar na possibilidade de personalização dos serviços de transações. Dessa forma, um serviço poderia ser composto somente com aqueles mecanismos necessários às aplicações interessadas. Nas seções a seguir são apresentados alguns conceitos que podem fornecer as bases para a definição de serviços de transação personalizáveis: reflexão computacional, componentes de software e framework de componentes.

2.3 Reflexão Computacional

Reflexão é um termo vindo do latim que significa “voltar para trás”. Esse significado original serviu como base para o uso desse termo em diversas áreas do conhecimento como, a matemática, a física, a filosofia e a computação. O campo da computação usa

esse termo na definição da chamada reflexão computacional. Nesse contexto, a reflexão pode ser vista como a capacidade que um sistema pode ter de "voltar a si próprio" a sua computação. Especificamente, esse sistema teria a capacidade de conhecer, analisar e alterar a sua própria representação.

Na literatura são apresentadas algumas definições clássicas do conceito de reflexão computacional [99, 64, 100]. Em uma destas definições [64], um sistema reflexivo é aquele que incorpora estruturas que representam aspectos dele mesmo. Esta auto-representação torna o sistema apto a responder questões e a executar ações sobre si mesmo. Além disso, a computação realizada por um sistema reflexivo e a sua representação podem ser fortemente vinculadas. Nesse caso, mudanças ocorridas na computação apareceriam na representação e vice-versa.

A reflexão computacional pode ser diferenciada em dois tipos básicos [33]: a reflexão estrutural e a reflexão comportamental. No geral, a reflexão estrutural está relacionada com a habilidade de refletir sobre os elementos que compõem a estrutura do sistema. Já a reflexão comportamental está relacionada com a habilidade de refletir sobre os elementos que definem o comportamento do sistema [65]. Por exemplo, se aplicada ao paradigma da orientação a objetos, a reflexão estrutural ocorreria em classes enquanto que a comportamental ocorreria em objetos.

Arquiteturas que utilizam reflexão computacional normalmente são organizadas em uma estrutura orientada a objetos de dois níveis. O primeiro nível é chamado de nível base, onde ficam os objetos do domínio da aplicação. O segundo é o chamado metanível, onde ficam metaobjetos ou metaclasses. Quando a reflexão é comportamental, o metanível é composto por metaobjetos que possuem informações que descrevem o comportamento dos objetos do nível base. Quando a reflexão é estrutural, o metanível é composto de metaclasses que possuem informações que descrevem a estrutura do nível base.

A reflexão computacional tem se mostrado uma ferramenta de grande utilidade em diversas áreas da computação. Ela nasceu no contexto de linguagens de programação [99] e se difundiu por outras áreas como, sistemas operacionais [57, 107], sistemas tolerantes a falhas [88]. Nos últimos anos a reflexão computacional tem sido usada com sucesso em projetos de sistemas distribuídos e middlewares abertos dando a esses a capacidade de se reconfigurarem em tempo de execução [52].

Uma das chaves do sucesso da reflexão computacional está na sua capacidade de prover a inspeção e a adaptação de sistemas. Essa capacidade se torna essencial para lidar com elementos cada vez mais dinâmicos. Por um lado, esse dinamismo pode ser observado nas características dos novos ambientes computacionais que surgiram em decorrência dos avanços em sistemas distribuídos e sistemas móveis: variações na disponibilidade de recursos, na conectividade e nas plataformas de hardware e software. Por outro lado, esse dinamismo pode ser observado nos requisitos exigidos por aplicações de diferentes domínios.

Desta forma, a reflexão computacional pode ser usada como ferramenta para dar a um sistema a capacidade de se adaptar às variações dos novos ambientes e à diversidade de requisitos de aplicações que necessitem do seu apoio.

Apesar de atrativa, a reflexão computacional também pode trazer implicações que devem ser observadas [8]:

- **Desempenho** – quando um sistema reflexivo é implementado, código extra é adicionado a ele e, conseqüentemente, um processamento extra é exigido. Esse processamento pode acarretar perdas de desempenho do sistema como um todo.
- **Integridade** – permitir acesso e manipulação dos elementos que compõem um sistema pode ocasionar problemas na integridade do mesmo. Isto exige um cuidado adicional com a manutenção desta integridade, seja com o uso de mecanismos de verificação ou com o uso de restrições ao acesso e manipulação desses elementos.

2.4 Componentes

A dificuldade existente em se desenvolver softwares que atendam aos seus requisitos, que possam ser compreendidos e que possam ter a corretude verificada deu origem à chamada crise do software. Um dos primeiros registros do termo "crise do software"ocorreu em uma conferência de engenharia de software na Alemanha em 1968. Foi a partir desse ano que pesquisadores da área começaram a concordar que o processo de desenvolvimento de softwares de grande porte é uma tarefa muito difícil e que requer uma série de cuidados [28].

Uma das formas de facilitar o processo de desenvolvimento de um software de grande porte é dividi-lo em partes menores. Utilizando essa idéia e fazendo um paralelo com a indústria de componentes de hardware, foi proposta a idéia de produção de componentes de software [66]. Componentes poderiam ser desenvolvidos por indústrias de software que garantissem a alta qualidade, confiabilidade e eficiência dos componentes. Componentes implementariam famílias de rotinas para cada dada tarefa e poderiam ser interconectados para produção de softwares.

Na literatura, vários autores definem componente de software. Em uma destas definições um componente de software é "uma unidade de composição com interfaces contratualmente especificadas com dependências explícitas de contexto" [101]. O elemento chave desta definição é a ênfase na composição ao invés da herança de classes (como ocorre na orientação a objetos).

Para que softwares sejam implementados a partir de componentes, esses devem ser interconectados. A estrutura usada para a interconexão entre as partes de um software é de grande importância para a definição de uma estrutura global clara, concisa e que

possa ter a corretude verificada [26]. É devido a essa importância que para um sistema ser considerado baseado em componentes ele, além de ser composto por componentes, tem que possuir uma noção explícita das interconexões entre eles [97]. Uma outra característica que deve fazer parte dos sistemas baseados em componente é que a troca de um componente por um outro (por exemplo, em uma manutenção) poderia ser feita sem a necessidade de recompilação [25].

Em [97] são destacadas algumas vantagens e desvantagens de sistemas baseados em componentes:

Vantagens:

- A complexidade de sistemas grandes pode ser tão alta que quebrá-lo em partes pode ser a única solução para conseguir gerenciá-la.
- Quando se constrói um sistema a partir de partes, é possível que algumas destas partes já estejam disponíveis, ou seja, elas podem ser reusadas.
- Cada parte pode ser desenvolvida ou alterada em paralelo e por equipes distintas.
- Sistemas baseados em componentes delimitam aspectos relacionados de um sistema e, desta forma, se tornam mais fáceis de entender, explicar e gerenciar.
- Sistemas baseados em componentes são mais flexíveis. Por possuírem interfaces bem definidas e encapsular detalhes de implementação, os sistemas são mais fáceis de serem adaptados, adicionando-se, removendo-se ou substituindo componentes.

Desvantagens:

- Dividir um sistema em partes, ou seja, o processo de decomposição, é uma tarefa difícil. Além disso, existem muitas decomposições possíveis e pode não ser muito claro qual é a mais apropriada segundo os requisitos.
- Compor um sistema a partir de partes também é uma tarefa difícil. As partes disponíveis podem não se encaixar perfeitamente com o restante do sistema e exigir, assim, trabalho extra.
- Sistemas compostos de partes são freqüentemente criticados pelo seu desempenho reduzido.
- Sistemas com um número grande de partes podem se tornar difíceis de entender.

Existem várias plataformas que permitem a implementação de sistemas baseados em componentes. Dentre essas podem ser destacadas o Enterprise JavaBeans [70], o CORBA Component Model [42], o Component Object Model (COM) [23], o .NET [24] e o OpenCOM [19].

2.5 Framework de Componentes

O uso de componentes é considerado uma técnica que traz uma série de benefícios à implementação de sistemas. Porém, para que esse mecanismo possa revelar a sua eficiência e eficácia, aspectos relacionados ao sistema como um todo devem ser observados como, regras de integridade e de interconexão entre os componentes, aspectos não funcionais e a arquitetura do sistema. Framework de componentes é uma tecnologia capaz de lidar com esses aspectos.

Um framework de componentes pode ser visto como uma arquitetura genérica e organizada que pode ser personalizada interconectando-se componentes prontos. A interação dos componentes que fazem parte do framework é regida por um conjunto de regras e interfaces [101, 67].

Na prática, a principal contribuição dos frameworks de componentes é que eles provêem meios de impor as propriedades arquiteturais desejadas controlando as interações entre os componentes de um sistema dentro de um domínio pré-estabelecido. As propriedades a serem mantidas podem ser tanto relacionadas a aspectos funcionais quanto a aspectos não funcionais. Quanto a aspectos funcionais, um framework de componentes pode definir, por exemplo, como as funcionalidades devem ser decompostas entre os componentes. Já quanto a aspectos não funcionais, o framework de componentes pode definir elementos como, o desempenho e os tipos de modificações que podem ser feitos nos componentes. Como benefícios adicionais, frameworks de componentes podem simplificar o processo de desenvolvimento de componentes fornecendo bases para a reutilização, para o desenvolvimento de componentes de pequeno porte e para facilitar a manutenção e a compreensão do sistema [10].

Devido ao seu potencial de impor regras apropriadas relacionadas à interação dos componentes de um sistema, frameworks de componentes têm sido usados em conjunto com o conceito de reflexão computacional em vários projetos de middleware e de aplicações distribuídas [9, 53, 96, 97]. Um dos objetivos desses projetos em reunir estas técnicas é o de permitir reconfiguração dinâmica. Enquanto que a reflexão computacional provê meios de acessar e de configurar os elementos que compõem o sistema, o framework de componentes tem a função de impor regras apropriadas a esse processo.

Capítulo 3

Trabalhos Relacionados

Nesta seção serão discutidos alguns grupos de trabalhos que serviram como base para esta pesquisa. Isso será feito em quatro partes.

Primeira parte. Discutirá padrões tradicionais de transações distribuídas, de onde se terá uma visão geral de como é a arquitetura e o funcionamento dos mesmos.

Segunda parte. Discutirá modelos de transações para ambientes móveis destacando os mecanismos usados para lidar com esses ambientes.

Terceira parte. Apresentará frameworks para transações destacando a tentativa de flexibilizar aspectos relacionados com o processamento de transações.

Quarta parte. Discutirá os trabalhos relacionados em middlewares abertos. Nesses trabalhos identificaremos os mecanismos utilizados para se prover flexibilidade e configurabilidade.

3.1 Padrões Tradicionais de Transações Distribuídas

Padrões de processamento de transações distribuídas surgiram como conseqüência dos esforços mantidos por grandes representantes da indústria de software em aumentar a confiabilidade de aplicações em domínios como, financeiro, bancário e de comércio eletrônico. Entre esses padrões estão X/Open Distributed Transaction Processing, Java Transaction Service, OMG Object Transaction Server e Microsoft Transaction Server que, na prática, visam manter a integridade de dados acessados por aplicações concorrentes em ambientes distribuídos.

Nas seções a seguir, três dos principais padrões serão apresentados ressaltando suas arquiteturas e funcionamento. Na primeira seção, os conceitos básicos do processamento

de transações distribuídas e o padrão X/Open DTP serão discutidos. A segunda seção apresenta o padrão Object Transaction Service. A terceira seção apresenta o padrão Java Transaction Service. Na última seção, resumiremos os padrões descritos fazendo um paralelo entre os componentes envolvidos e discutindo o apoio a ambientes móveis.

3.1.1 X/Open Distributed Transaction Processing

O X/Open DTP (Distributed Transaction Processing) [22] é uma arquitetura proposta pelo Open Group[1] que possibilita o compartilhamento de recursos entre diversas aplicações através da coordenação de transações. O X/Open DTP tem sido adotado como padrão por grandes representantes da indústria de software para o desenvolvimento de sistemas de banco de dados e de processamento de transações.

A Figura 3.1 mostra uma arquitetura simplificada do X/Open DTP. Essa arquitetura é representada por três principais componentes:

Resource Managers (RM). Representam bancos de dados ou sistemas gerenciadores de arquivos que provêem acesso aos recursos compartilhados.

Transaction Manager (TM). É responsável por gerenciar transações globais, associar identificadores a transações, monitorar seu progresso e coordenar os processos de efetivação e recuperação.

Application Programs (AP). Representam aplicações que implementam as funcionalidades desejadas do usuário. Cada aplicação especifica a seqüencia de operações que envolvem recursos compartilhados junto aos RM's. Uma aplicação define o início e o fim de transações globais, acessa recursos dentro do escopo de transações e normalmente toma decisões como, efetivar ou abortar transações.

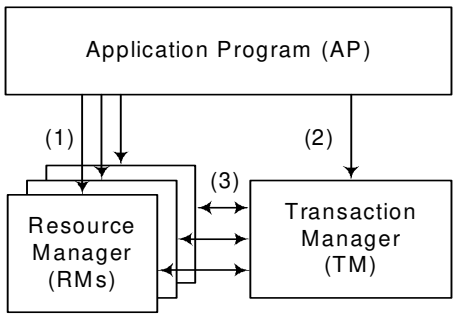

Figura 3.1: Arquitetura do X/Open DTP

[1]Um consórcio entre empresas de desenvolvimento de software

A interação entre os elementos que compõem o padrão X/Open é feita através de um conjunto de interfaces bem definidas. Dentre elas estão as interfaces TX [21], XA e AX [20].

- **Interface TX** – define as operações do TM que são chamadas por aplicações. Algumas das principais operações:
  - `tx_begin` – chamada pela aplicação para iniciar uma nova transação.
  - `tx_commit` – chamada pela aplicação para efetivar a transação.
  - `tx_rollback` – chamada pela aplicação para abortar a transação.
- **Interface XA** – define as operações dos RM's que são chamadas pelo TM. Algumas das principais operações:
  - `xa_prepare` – chamada pelo TM na primeira fase do protocolo de efetivação[2] para pedir um voto ao RM. Nesse voto o RM indica ao TM se a transação deve ser efetivada ou cancelada.
  - `xa_commit` – chamada pelo TM na segunda fase do protocolo de efetivação para efetivar as operações transacionais junto ao RM.
- **Interface AX** - define as operações do TM chamadas pelo RM. Algumas das principais operações:
  - `ax_reg` – usado pelo RM para se registrar junto ao TM.
  - `ax_unreg` – remove o registro do RM junto ao TM

Em geral, o apoio transacional provido pelo padrão X/Open DTP funciona da seguinte maneira.

1. A aplicação chama a operação `tx_begin` do TM para iniciar uma transação.
2. A aplicação executa as suas operações sobre os recursos compartilhados junto a um ou mais RM's.
3. Ao receber operações sobre os recursos, cada RM requisitado se registra junto ao TM chamando a operação `ax_reg` do TM.
4. A aplicação chama a operação `tx_commit` ou `tx_rollback` do TM para finalizar a transação.

[2]Two Phase Commit

5. Ao receber a chamada da operação `tx_commit`, o TM inicia o protocolo de efetivação. Na primeira fase do protocolo, o TM chama a operação `xa_prepare` em cada RM registrado. Se todos os RM's responderem positivamente, o TM inicia a segunda fase do protocolo. Na segunda fase, o TM chama a operação `xa_commit` para cada RM registrado. Depois de executada a segunda fase, a transação é considerada efetivada. No caso de término com `tx_rollback`, a transação é abortada.

3.1.2 Object Transaction Service

O Object Transaction Service (OTS) [41] é um serviço de processamento de transações distribuídas baseado no X/Open DTP que foi especificado pela OMG (Object Management Group). O OTS define um conjunto de interfaces que provêem um serviço de processamento de transações para aplicações que acessam objetos CORBA.

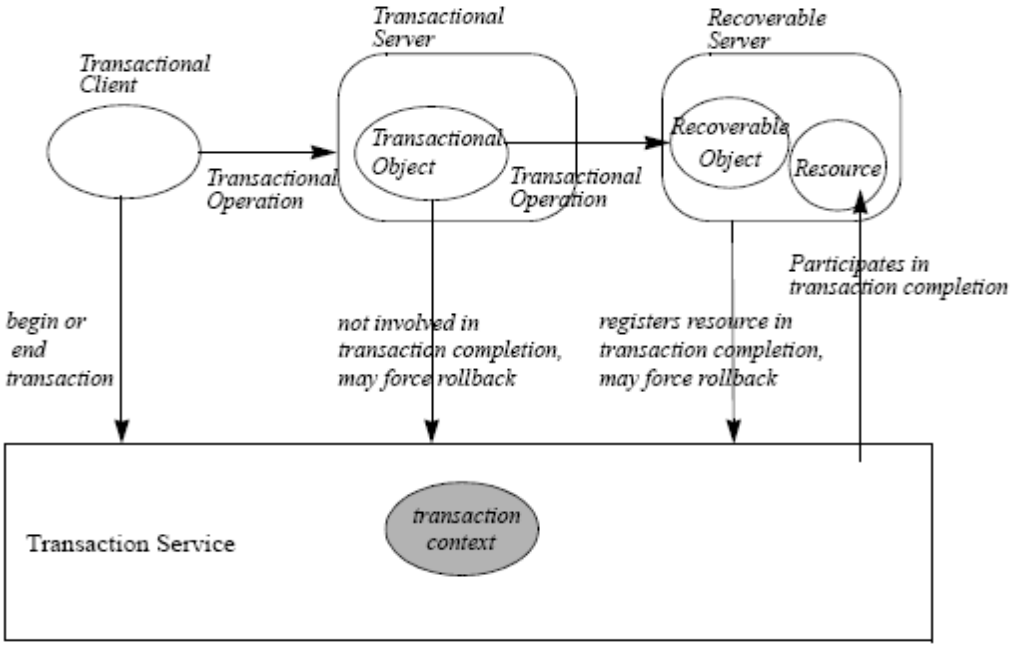

Figura 3.2: Arquitetura do OTS

A arquitetura do OTS é apresentada na Figura 3.2. Uma breve descrição dos seus elementos é apresentada a seguir.

Transactional Client. Representa as aplicações que utilizam o serviço de transações provido pelo OTS para acessar objetos transacionais.

Transactional Object. Representa os objetos transacionais, ou seja, objetos que estão aptos a serem controlados pelo serviço de transação quando acessados dentro do

escopo de uma transação.

Recoverable Object. Representa objetos transacionais que possuem um estado persistente que deve ser mantido pelo serviço de transações.

Resource Objects. Representa objetos que são registrados pelo serviço de transações para que possam fazer parte dos protocolos de efetivação e de recuperação.

Transactional Server. Representa uma coleção de um ou mais objetos transacionais.

Recoverable Server. Representa uma coleção de objetos em que pelo menos um deles é um objeto recuperável. o *Recoverable Server* participa dos protocolos do serviço de transações registrando um ou mais *Resource Objects*.

Transaction Service. Representa o gerenciador de transações que provê operações transacionais (ex: begin, end) para aplicações transacionais (Transactional Client) e gerencia a execução de protocolos de efetivação e de recuperação junto aos participantes da transação (Resource).

3.1.3 Java Transaction Service

O Java Transaction Service (JTS) [69] especifica a implementação de um gerenciador de transações capaz de prover um serviço de transações a aplicações através de uma API chamada JTA [68]. Visando uma maior interoperabilidade e portabilidade, o JTS foi especificado com base nas interfaces definidas pelo OTS.

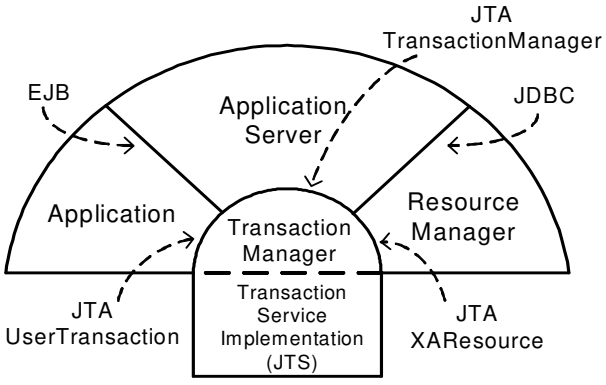

Figura 3.3: Arquitetura do JTS

A arquitetura do JTS é apresentada na Figura 3.3. Uma breve descrição dos seus elementos é apresentada a seguir.

Transaction Manager. Representa o gerenciador de transações que provê operações transacionais de demarcação a aplicações, gerenciamento de recursos, sincronização e propagação do contexto de transações.

Application Server. Representa o servidor de aplicação que provê a infraestrutura necessária para apoiar um ambiente de execução de aplicações. Um exemplo de servidor de aplicação é um servidor EJB.

Resource Manager. Representa o gerenciador de recursos – gerencia os recursos acessados por aplicações durante a execução de transações. Participa dos protocolos de efetivação e de recuperação de transações.

Application. Representa as aplicações que utilizam o serviço de transações para acessar os recursos gerenciados pelo gerenciador de recursos.

3.1.4 Discussão

Dada a grande importância dos serviços de processamento de transações na implementação de aplicações que acessam dados distribuídos, várias plataformas de processamento de transação diferentes foram propostas. Para tentar manter uma interoperabilidade entre as mesmas é que os padrões descritos acima foram especificados. Assim, as plataformas de processamento de transação desenvolvidas em um mesmo padrão poderiam interoperar.

Apesar da vantagem da interoperabilidade, esses padrões foram especificados com base nos requisitos de aplicações tradicionais. Esses requisitos são baseados nas propriedades ACID tradicionais. Dessa forma, esses padrões não são aptos a atender os requisitos de aplicações de novos domínios como, a computação móvel, aplicações multimídia e de tempo real. Esses domínios possuem requisitos como: flexibilização das propriedades ACID, configurabilidade e extensibilidade.

Dados os requisitos dos novos domínios de aplicação, diversos modelos de transação especializados foram propostos. Na seção a seguir, serão apresentados alguns desses modelos especializados que consideram os requisitos de ambientes de computação móvel.

3.2 Modelos de Transação para Computação Móvel

Nesta seção será apresentada uma breve descrição dos principais modelos de transação projetados para o ambiente de computação móvel. Uma descrição mais detalhada desses e outros modelos pode ser encontrada em [83].

3.2.1 Coda-IOT

O Coda [92] é um dos trabalhos pioneiros e relevantes na área da computação móvel. Apesar de não ter sido esse um dos seus objetivos iniciais, o Coda é um sistema que provê acesso a dados a partir de máquinas móveis. Para isto, esse sistema utiliza técnicas como operação desconectada e operação fracamente conectada.

Os primeiros enfoques do Coda foram voltados ao tratamento de falhas de partição que afetam os usuários de sistemas distribuídos. Para contornar esses problemas foram propostas basicamente duas técnicas: a *replicação de servidor* e a *operação desconectada*. A *replicação de servidor* é uma técnica fundamental para a garantia da disponibilidade de dados durante falhas que ocasionam partição da rede. O Coda utiliza um conjunto de servidores que contém réplicas dos dados úteis aos usuários da rede. Assim, quando ocorre a partição da rede, um usuário poderá acessar os dados de um outro servidor na mesma partição em que ele se encontra.

A *operação desconectada* é uma técnica usada principalmente para garantir a disponibilidade em circunstâncias de falhas em que a replicação de servidor não resolve. Um exemplo disso seria a perda de comunicação com todos os servidores da rede. A idéia central da operação desconectada é manter na cache da máquina do usuário uma cópia dos dados úteis ao mesmo. Assim, com a desconexão, o usuário poderia continuar as suas operações sobre a cópia em cache.

Estudos posteriores deram ao Coda a capacidade de operar em ambientes com comunicação sem fio, caracterizados pela largura de banda baixa e intermitente. Para isto, foi utilizada uma técnica chamada operação fracamente conectada [72]. Primeiro, o protocolo de manutenção dos dados em cache foi modificado para permitir uma rápida revalidação de grandes volumes de dados em cache depois de um período de desconexão. Além disso, foi desenvolvido um mecanismo chamado reintegração por partes que utiliza, de forma flexível, a largura de banda da rede na propagação da atualização dos dados em cache para os servidores.

Os modelos iniciais de replicação de dados no Coda só visavam a resolução de conflitos do tipo escrita/escrita. Os conflitos do tipo leitura/escrita eram ignorados visto que a possibilidade de sua ocorrência era tolerada pelos usuários de sistemas de arquivos compartilhados. Porém, com a operação desconectada a ocorrência desses conflitos aumentou de forma considerável. Para lidar com esse problema, o Coda foi estendido com um mecanismo chamado Isolation-Only Transaction (IOT) [61]. Esse mecanismo permite ao sistema admitir somente os conflitos leitura/escrita que satisfaçam certos critérios de serialização. Assim, o IOT busca dar uma melhor garantia de consistência a aplicações que executam no ambiente móvel.

3.2.2 PRO-MOTION

O PRO-MOTION [104] é um sistema de processamento de transações projetado para lidar com os problemas introduzidos pela desconexão e pelos recursos limitados do ambiente móvel. Seu principal bloco de formação é o *compact* que funciona como uma unidade básica de replicação de dados.

Um *compact* é representado como um objeto que encapsula: dados, operações para acesso aos dados, operações para o gerenciamento do compact, informação a respeito do estado corrente do compact, regras de consistência e obrigações. O compact representa um acordo entre o cliente móvel que requisitou o dado e o servidor de banco de dados. Nesse acordo, o servidor de banco de dados delega à unidade móvel o controle de alguns dados que serão usados por transações locais. O cliente móvel, em contrapartida, concorda em honrar as condições específicas do uso dos dados de maneira que sua consistência seja mantida quando as atualizações forem propagadas de volta para o servidor de banco de dados. Cada compact é responsável pelo controle de concorrência e recuperação dos seus dados.

O PRO-MOTION sugere quatro fases de processamento de transações:

1. **Armazenamento**[3] - A unidade móvel está conectada à rede e os compacts estão sendo armazenados localmente na preparação para uma eventual desconexão.

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

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

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

Para aumentar a disponibilidade de dados, o PRO-MOTION permite a efetivação[4] local de transações. Caso uma transação opte pela efetivação local, seus resultados se tornarão visíveis para outras transações da unidade móvel. Se a transação não optar pela efetivação local, ela só será efetivada localmente após a efetivação global realizada no servidor depois da reconexão.

A consistência das operações sobre os dados no PRO-MOTION pode ser caracterizada através de uma escala com dez níveis de isolamento baseados no padrão ANSI-SQL. O nível 9 garante a execução seqüencial de transações e o nível 8 garante uma execução

[3]Hoarding
[4]Commit

serializável[5]. Cada nível inferior representa um menor nível de isolamento. O nível 0 não garante nenhum tipo de consistência. Flexibilizando a propriedade de isolamento, o PROMOTION aumenta a concorrência no acesso a dados, o que implica em uma melhoria de desempenho das transações.

3.2.3 Reporting e Co-Transactions

Reporting Transactions e *Co-Transactions* [16] é um modelo transacional que relaxa as restrições impostas pelas manutenção da propriedade da atomicidade. As transações atômicas tradicionais possuem restrições que impedem o particionamento da computação e o compartilhamento de resultados parciais. Porém, esse modelo considera que o particionamento da computação e o compartilhamento dos resultados parciais das transações no ambiente móvel são mecanismos necessários para lidar com a escassez de recursos nos dispositivos de computação móveis e com outros problemas como, desconexões e baixa largura de banda. Sendo assim, para reduzir os impactos desses problemas, esse modelo propõe a decomposição de uma transação em um conjunto de transações. Uma parte delas pode executar na unidade móvel enquanto que uma outra parte executa na rede fixa.

Nesse modelo, transações que compartilham resultados parciais são chamadas de *Reporting Transaction*. Para isso, essas transações são aptas a delegar resultados parciais a uma outra transação. Essa delegação pode ser feita em qualquer ponto de sua execução. Dessa forma, uma transação executando no computador móvel pode repassar resultados parciais para outra transação que executa na rede fixa.

O particionamento da computação é feito por transações chamadas *Co-Transactions*. *Co-Transactions* podem ser definidas como *Reporting Transactions* que têm um comportamento semelhante a co-rotinas. *Co-Transactions* permitem que o controle seja passado de uma transação para outra no momento do compartilhamento dos resultados parciais. As *Co-Transactions* diferem das *Reporting Transactions* pelo fato de que a execução daquelas são suspensas no momento da delegação e são reiniciadas a partir do ponto que foram previamente suspendidas, enquanto que as *Reporting Transactions* continuam a executar concorrentemente com a transação para a qual ela delegou seus resultados parciais.

3.2.4 Kangaroo

O modelo de transações Kangaroo [32] enfoca a questão da mobilidade de transações, permitindo que elas possam executar mesmo a partir de uma unidade móvel em movimento. Nesse modelo, o local de origem de uma transação pode ser diferente do local de

[5]Em inglês: serializable

finalização da mesma. As operações de uma transação Kangaroo sempre são executadas em máquinas da rede fixa a partir de uma unidade móvel.

A execução de uma transação Kangaroo pode consistir na execução de um conjunto de transações chamadas *Joey*. Uma transação Kangaroo inicia a sua execução através de uma transação *Joey* em uma determinada estação base. À medida que a unidade móvel onde se encontra a transação Kangaroo se move de uma célula para outra, uma transação *Joey* da célula anterior é efetivada e é criada uma nova transação *Joey* na estação base da nova célula. Cada transação *Joey* representa uma parte da transação Kangaroo que foi executada em uma determinada célula.

Uma transação Kangaroo pode ser executada no modo *compensating* ou no modo *split*. Quando uma transação Kangaroo executa no modo *compensating*, se ocorrer falha na transação *Joey* corrente, todas as transações Joey´s que já tiverem sido executadas são desfeitas juntamente com a transação *Joey* que falhou. No modo split, quando ocorre uma falha, a decisão de efetivar ou de abortar as transações Joey passa a ser responsabilidade do DBMS.

Esse modelo foi projetado com base nos conceitos de transações globais e transações split [13].

3.2.5 AMT

AMT [94] é um modelo que tem como objetivo dar às transações a capacidade de se adaptar à alta instabilidade do ambiente de computação móvel caracterizada por largura de banda variável, desconexões e custo de comunicação variável. Para isso, ele provê um mecanismo de monitoramento capaz de informar as transações sobre as variações ocorridas no ambiente.

Ciente do estado do ambiente de execução, uma transação AMT é apta a escolher a melhor forma de execução dentre um conjunto de alternativas previamente programadas. O sucesso de pelo menos uma das alternativas representa a execução correta da transação.

Uma transação AMT pode conter descritores de ambiente que expressam os estados no qual o ambiente deve estar para que cada alternativa seja executada. Quando uma transação é iniciada, o estado do ambiente é verificado e a alternativa de execução apropriada é escolhida. Se não existir nenhuma alternativa para o estado do ambiente, a execução da transação pode ser adiada. Uma alternativa de execução será executada assim que um estado aceitável do ambiente for detectado.

O modelo AMT utiliza um protocolo de efetivação chamado CO2PC que é uma combinação entre o protocolo Two-Phase Commit (de abordagem pessimista) e mecanismos de uma abordagem otimista. Esse protocolo permite que alguns tipos de transações possam ser efetivadas localmente dentro da abordagem otimista enquanto que outros tipos

de transações só sejam efetivadas globalmente juntamente com a efetivação da EA.

3.2.6 Moflex

Moflex [55] é um modelo que visa trazer mais mobilidade, heterogeneidade e flexibilidade para a execução de transações em ambientes de computação móvel. Esse modelo se propõe a permitir que transações possam ser executadas mesmo enquanto a unidade móvel se move entre diferentes células.

Uma transação Moflex é definida como uma coleção de subtransações que podem estar inter-relacionadas por um conjunto de dependências de execução. Podem ser definidos três tipos de dependências entre subtransações: dependências de sucesso (*ds*), dependências de falhas (*df*) e dependências externas (*de*). Uma transação `A` que possui uma *ds* em relação a uma transação `B` só pode ser executada se `B` tiver sido efetivada com sucesso. Se `A` possui uma *df* em relação a `B`, `A` só pode ser executada depois que `B` falhar. Quando uma transação possui uma *de*, ela só pode executar quando um predicado externo (tempo, custo ou localização) for satisfeito.

Uma transação Moflex pode conter regras de mudança de célula. Estas regras definem as políticas de execução de uma subtransação quando há mudança de célula em decorrência da movimentação da unidade móvel. Podem ser definidas as seguintes regras para a mudança de célula de uma transação: `continue(ti)` – a transação `ti` continua a sua execução na nova célula; `restart(ti)` – a transação `ti` é abortada na célula anterior e reiniciada na nova célula; `resume(ti)` – a transação `ti` é dividida em uma subtransação efetivada na célula anterior e uma segunda subtransação que continua a executar na nova célula; `split_restart(ti)` – a transação `ti` é dividida em uma subtransação efetivada na célula anterior e uma segunda subtransação é reiniciada na nova célula.

As transações que executam com as regras `split-resume(ti)` ou `split_restart(ti)`, ou seja, que executam a operação de divisão da transação em duas subtransações, são acompanhadas por uma regra de junção que é usada para verificar se a concatenação das subtransações resultantes de `ti` é computacionalmente equivalente à `ti`.

3.2.7 HiCoMo

HiCoMo [59] é um modelo de transações voltado a aplicações de tomada de decisão que acessam dados do tipo estatístico ou agregados. O ambiente é composto por tabelas base na rede fixa e agregados de dados das tabelas base (*data warehouse*) nas unidades móveis. Esse modelo se propõe a manter altas taxas de efetivação global de transações que acessam esses dados agregados durante os períodos de desconexão. Para isto, o modelo impõe restrições como: *(i)* os dados agregados a serem armazenados nas unidades móveis só devem conter valores como média, soma, mínimo e máximo; *(ii)* uma transação HiCoMo

só pode realizar operações comutativas (adição e subtração) sobre os dados agregados; *(iii)* deve ser especificada uma margem de erro para a execução de uma transação HiCoMo.

O resultado do processamento de transações HiCoMo sobre os dados agregados durante a desconexão é transferido para as tabelas da rede fixa quando ocorre a reconexão. Para isso, transações base são criadas e executadas na rede fixa para gerar sobre as tabelas base o mesmo resultado que foi obtido durante a desconexão. Esse processo é realizado pelo seguinte algoritmo:

- **Passo 1.** Detecção de conflitos: verifica-se se há conflitos entre as transações HiCoMo e as transações base que executaram na rede fixa. Se algum conflito for detectado, a transação HiCoMo é abortada. Esta detecção de conflitos é implementada por um controle de concorrência otimista baseado em *timestamps*.

- **Passo 2.** Criação das transações base: se não existem conflitos, é criada uma transação base para cada tabela base que foi usada na formação dos dados agregados. Uma vez criadas, estas transações base são executadas como subtransações de uma transação aninhada estendida aplicadas sobre as tabelas base.

- **Passo 3.** Cancelamento[6] de subtransações bases: quando submetidas à execução, algumas subtransações geradas podem abortar devido a regras de manutenção de integridade das tabelas base. Se a diferença que as subtransações abortadas criam estiver dentro da margem de erro, pode-se considerar que as subtransações obtiveram sucesso. Senão, novas subtransações podem ser geradas e executadas com uma nova estratégia. Se a transformação não resultar em sucesso em algum ponto, a transação HiCoMo é abortada.

3.2.8 MobileTrans

MobileTrans [90] é um modelo que provê meios de uma transação se adaptar a diferentes cenários de ambientes de computação móvel. Nesse modelo, o desenvolvedor deve especificar uma política de adaptação para cada transação. Uma política de adaptação consiste em associar valores a um conjunto de atributos que determinarão o comportamento da transação. Os atributos que podem ser definidos são os seguintes:

Consistência – é possível especificar regras de consistência para se permitir o uso de versões desatualizadas de objetos se a cópia remota correspondente não estiver disponível.

Busca[7] – é possível selecionar os objetos que devem ser copiados para a cache da unidade

[6] Abort
[7] Fetching

móvel antes da execução da transação; pode-se especificar também se os objetos devem ser copiados para cache previamente ou sob demanda.

Delegação – é possível delegar a responsabilidade de efetivação de uma transação para outra transação.

Atomicidade – é possível especificar se a transação pode ser efetivada mesmo se nem todos os participantes envolvidos estão acessíveis.

Caching – é possível especificar que os objetos copiados para a cache e os objetos modificados por transações locais serão armazenados localmente de forma que possam ser acessados durante os períodos de desconexão.

Tratamento de falhas – é possível determinar o que deve ser feito quando as condições de consistência, de busca e de atomicidade especificadas não puderem ser garantidas devido ao estado da rede. Também é possível mudar a configuração da transação em tempo de execução como um meio de tratamento da falha ocorrida.

3.2.9 Mobisnap

Mobisnap [77] é um modelo de gerenciamento de transações móveis que permite a execução de transações em unidades móveis desconectadas. O ambiente para o qual esse modelo foi desenvolvido é composto por servidores sempre conectados a uma rede fixa que armazena a cópia mestra de todos os itens de dados e unidades móveis que podem armazenar réplicas dos dados da rede fixa.

Neste modelo, uma unidade móvel armazena duas cópias de cada item de dado da rede fixa: uma versão efetivada e uma versão de tentativa. A versão efetivada reflete a última versão que foi copiada da versão da rede fixa. A versão de tentativa é baseada na versão efetivada, porém ela reflete as operações das transações móveis que executam na unidade móvel. Uma transação que executa na unidade móvel pode acessar ambas as versões sendo que a versão de tentativa representa a evolução esperada do item de dado correspondente da rede fixa.

Um aspecto relevante do modelo Mobisnap é que ele disponibiliza um mecanismo que permite a reserva prévia de itens de dados da rede fixa que serão usados por transações que executarão em uma unidade móvel desconectada. O modelo Mobisnap define quatro tipos de reserva:

1. **Escrow** - usado para dividir um recurso que seja divisível. Por exemplo, pode-se dividir um lote de um determinado produto em várias partes sendo que cada parte seria reservada a um vendedor.

2. **Slot** - usado para reservar o direito de inserir um novo registro com valores pré-definidos. Por exemplo, pode-se reservar o direito de agendar um encontro em uma sala em um determinado horário.

3. **Value-change** - usado para reservar o direito de mudar o valor de itens de dados de uma base de dados. Por exemplo, reservar o direito de mudar a descrição de um produto.

4. **Value-use** - usado para reservar o direito de uso de itens de dados. Por exemplo, reserva-se o direito de vender um produto por um determinado preço mesmo que esse preço seja alterado.

Cada reserva de recurso feita na unidade móvel possui um prazo de expiração. Isto permite que o sistema use itens reservados por uma transação desconectada que não entrou em contato com o servidor para efetivar as operações. Enquanto o prazo de uma reserva não expira, nenhuma transação pode utilizar o item de dado reservado. Desta forma, se uma transação só acessa valores que foram previamente reservados, o seu resultado pode ser corretamente estabelecido na própria unidade móvel.

As transações que executaram com sucesso na unidade móvel são posteriormente propagadas para o servidor. Quando o servidor recebe uma transação móvel ele executa o seu programa transacional de forma a efetivar a execução da transação atualizando os dados do servidor. Nesta fase, uma transação móvel poderá ser abortada se houver casos como, detecção de conflito com outras transações sobre itens de dados que não foram previamente reservados ou que tiveram a reserva expirada.

3.2.10 Modelo baseado em Pré-Serialização

Pre-serialization [29, 30] é uma técnica de gerenciamento de transações móveis sobre um sistema de bases de dados múltiplas (SBDM). Esta técnica permite que uma transação móvel estabeleça a sua ordem de serialização[8] antes mesmo de chegar à fase de efetivação. Nesse modelo, uma transação que executa na unidade móvel (chamada transação global) é composta por um conjunto de subtransações compensáveis na rede fixa. Cada subtransação engloba as operações da transação global realizadas sobre uma determinada base de dados do SBDM. Como todas as subtransações locais são compensáveis, elas podem ser efetivadas antes mesmo que a transação global tome a decisão de entrar na fase de efetivação. Isto permite que os recursos alocados para a execução da transação global possam ser liberados à medida que cada subtransação vai sendo finalizada e efetivada.

[8]Segundo a regra da serialização, transações que executaram concorrentemente são consideradas corretas quando os resultados obtidos são equivalentes aos resultados de uma execução serial das mesmas (uma após a outra).

Em cada estação de suporte do cenário para o qual a técnica é proposta existe um coordenador de transações globais (GTC) responsável por submeter as subtransações aos servidores da rede fixa, gerenciar a migração da unidade móvel e a desconexão e verificar as propriedades de isolamento e atomicidade.

Quando uma unidade móvel migra de uma estação de suporte para outra, o GTC da primeira estação se encarrega de enviar à segunda estação as informações necessárias para que a transação global possa continuar a sua execução.

Quando uma unidade móvel é desconectada, a transação global não é totalmente interrompida, as subtransações que já tiverem sido submetidas continuam a executar na rede fixa. Nesse período, todas as respostas geradas pelas transações locais são inseridas em uma lista que é enviada para a transação global quando ocorre a reconexão. Para evitar que os recursos de uma transação global T_x que não voltou a se reconectar fiquem indefinidamente alocados, quando outra transação T_y deseja acessar tais recursos, esses são automaticamente liberados para o uso de T_y e T_x pode ser considerada abortada.

Para verificar a garantia das propriedades de atomicidade e isolamento da transação global, o GTC, em determinados períodos, executa um algoritmo de serialização global em grafos. Para garantir a atomicidade, ou todas subtransações são efetivadas ou todas são abortadas ou compensadas (atomicidade semântica). Para garantir a propriedade de isolamento, as transações globais conflitantes são serializadas na mesma ordem da serialização de suas subtransações que executam em cada servidor.

3.2.11 Modelo baseado em Semântica

Esse trabalho [103] usa informação semântica de objetos para facilitar a execução de transações em unidades móveis sujeitas a desconexões. A informação semântica é usada para prover controle de concorrência e caching com uma menor granularidade permitindo uma manipulação assíncrona dos objetos em cache e efetivação unilateral de transações que executam na unidade móvel.

A idéia básica é dividir objetos maiores em fragmentos menores e do mesmo tipo. Exemplos de objetos considerados fragmentáveis são: itens agregados, pilhas, filas e conjuntos. Uma unidade móvel copia para a sua cache somente os fragmentos de objetos que serão utilizados por transações durante o período de desconexão. Isso minimiza o espaço em disco requerido na unidade móvel.

Uma requisição de caching feita por uma unidade móvel deve incluir dois parâmetros: critério de seleção e condições de consistência. O critério de seleção especifica o objeto requisitado e o tamanho requerido para o fragmento do objeto. As condições de consistência especificam as regras que precisam ser satisfeitas para manter a consistência do objeto como um todo.

Quando um fragmento de um objeto é copiado para a cache de uma unidade móvel, ela ganha o direito exclusivo de operar sobre ele, ou seja, nenhuma outra transação poderá acessar esse mesmo fragmento na rede fixa enquanto ele não for atualizado e liberado pela unidade móvel que o requisitou. Entretanto os demais fragmentos do objeto podem ser usados por outras transações.

Ao se reconectar, a unidade móvel inicia um processo de reconciliação dos fragmentos de objetos em cache com os objetos do servidor da base de dados.

3.2.12 Discussão

Para lidar com obstáculos presentes em ambientes de computação móvel (descritos no Capítulo 2) cada modelo descrito acima introduz um conjunto próprio de mecanismos. A Tabela 3.1 destaca, para cada modelo, os mecanismos utilizados para lidar com os seguintes obstáculos: desconexões, largura de banda baixa ou variável, suprimentos de energia limitados, escassez de espaço em disco e mobilidade (mudança de célula).

Como se pode observar, todos os modelos descritos propõem pelo menos um mecanismo para lidar com o problema da desconexão. Nesse caso, o mecanismo mais utilizado é a realização de caching de dados, que consiste em trazer uma cópia dos dados da rede fixa para a unidade móvel. Com isso, os dados se tornam disponíveis às aplicações nos momentos de desconexão.

Para lidar com os demais obstáculos, podem-se destacar *(i)* os mecanismos do *Coda-IOT*, do *PRO-MOTION* e do *Modelo baseado em semântica* para lidar com a baixa largura de banda – transferir para a cache somente as partes necessárias dos dados; *(ii)* o mecanismo do *Modelo baseado em semântica* para lidar com a escassez de energia – realizar desconexão voluntária para evitar gastos de energia com a transmissão de dados; *(iii)* os mecanismos do *PRO-MOTION* e do *Modelo baseado em semântica* para lidar com a escassez de espaço em disco – transferir para a cache somente as partes necessárias dos dados; *(iv)* os mecanismos do *Kangaroo* e do *Mobisnap* para lidar com a mobilidade – divisão de transações e transferência de informações de uma estação base para a outra.

Nos modelos de transação descritos, notou-se um relaxamento das propriedades ACID. Isso foi feito por pelo menos um dos dois motivos: *(i)* para atender requisitos específicos de aplicações e *(ii)* para lidar com as características dos ambientes de computação móvel. A Tabela 3.3 mostra, para cada modelo, os tipos de relaxamento das propriedades ACID que foram adotados.

Apesar da variedade de mecanismos de adaptação e da variedade das propriedades transacionais providos pelo conjunto dos modelos de transação apresentados, quando visto de forma isolada, cada modelo provê somente um subconjunto limitado desses mecanismos e propriedades. Essa limitação faz com que cada modelo atenda somente aos requisitos

de pequenos grupos de aplicação. Além disso, esses modelos não foram estruturados de forma extensível, pois se uma aplicação possuir requisitos um pouco diferentes daqueles previstos pelo modelo, ele não é capaz de prover.

3.3 Frameworks para Transações

Considere os seguintes dois fatos: *(1)* cada aplicação pode possuir um conjunto de requisitos distintos com relação às propriedades transacionais; *(2)* os modelos de transação propostos normalmente provêem um conjunto fixo e limitado de propriedades transacionais (tomem-se como exemplo os modelos apresentados na seção anterior). Esses fatos podem tornar difícil a tarefa de encontrar um modelo transacional que se enquadre perfeitamente nos requisitos desejados de uma determinada aplicação.

Diante dessa dificuldade, surgiram várias pesquisas que propuseram métodos de flexibilização do apoio transacional. A idéia principal foi, ao invés de propor um apoio transacional com um conjunto fixo de propriedades, a de permitir a personalização do apoio transacional de forma a atender aos requisitos específicos de cada aplicação. Esta seção apresenta alguns trabalhos que foram guiados nesse sentido.

3.3.1 ACTA

O ACTA [17, 18] é dos trabalhos que forneceram bases para a definição de um apoio transacional flexível. Esse trabalho demonstrou como a estrutura e o comportamento de transações estendidas poderiam ser analisados e formalmente especificados. Para a especificação sistemática de modelos de transação estendidos, esse trabalho propôs o framework ACTA. Nesse framework, a especificação de um modelo estendido consiste na definição dos seguintes elementos:

1. **Histórico.** Representa a execução concorrente de um conjunto de transações T. Um histórico define a ordem parcial dos eventos significantes (gerados por operações de controle como, begin, commit e abort) e dos eventos sobre objetos (gerados por operações sobre objetos) invocados por transações em T. Segundo o ACTA, através da imposição de restrições a um histórico, pode-se definir o efeito das transações sobre outras transações e sobre os objetos.

Tabela 3.1: Estratégias dos Modelos de Transações Móveis

Modelo	DESCONEXÃO	L. DE BANDA	ENERGIA	DISCO	MOBILIDADE
Coda-IOT	Caching de dados; efetivação local de transações	Reintegração por partes; validação rápida de dados em cache	Não especificado	Não especificado	Não especificado
PRO-MOTION	Caching de dados; efetivação local de transações	Transmissão das partes dos dados em falta ou desatualizados	Não especificado	Transferência de partes de dados para a cache; otimização do log de recuperação	Não especificado
Reporting	A parte da computação da unidade móvel pode continuar a executar	Não especificado	Divisão da computação entre máquinas fixas e máquinas móveis	Divisão da computação entre máquinas fixas e máquinas móveis	Não especificado
Kangaroo	Transações interrompidas por uma desconexão podem continuam a executar depois da reconexão	Não especificado	Desligar a unidade móvel permitindo a continuação da execução das transações quando esta for religada	Não especificado	Divide a execução da transação para que esta possa continuar a executar na nova célula
AMT	Permite que a transação decida qual alternativa deve ser executada	Permite que a transação decida qual alternativa deve ser executada	Permite que a transação decida qual alternativa deve ser executada	Permite que a transação decida qual alternativa deve ser executada	Não especificado
Moflex	Permite criação de dependências entre transações	Permite criação de dependências entre transações	Permite criação de dependências entre transações	Permite criação de dependências entre transações	Permite criação de dependências entre transações
HiCoMo	Caching de dados; restrições no acesso a dados em cache	Não especificado	Não especificado	Não especificado	Não especificado
MobileTrans	Caching de dados	A transação pode decidir quais serão as medidas de adaptação	A transação pode decidir quais serão as medidas de adaptação	Não especificado	Não especificado
Mobisnap	Caching de dados; reserva de acesso a dados com prazo de expiração	Não especificado	Não especificado	Não especificado	Transferência de informações de uma estação de apoio para outra necessárias para a continuação da transação
Pré-serialização	As operações são sempre realizadas na rede fixa; as subtransações que já tiverem sido submetidas na rede fixa continuam a executar	Não especificado	Não especificado	Não especificado	Não especificado
Baseado em semântica	Caching de dados	Transferir somente os fragmentos de objetos necessários	Operar no modo desconectado para evitar gastos de energia com transmissão de dados	Armazenar na cache somente fragmentos de objetos necessários durante a desconexão	Não especificado

Tabela 3.3: Relaxamento das Propriedades ACID

Modelo	ATOMICIDADE	CONSISTÊNCIA	ISOLAMENTO	DURABILIDADE
Coda-IOT	Sem flexibilidade	Sem flexibilidade	Liberação de resultados parciais antes da efetivação global	Sem garantia de durabilidade
PRO-MOTION	Transações aninhadas	Regras de consistência para acesso a dados	Provê dez níveis de isolamento	Sem garantia de durabilidade
Reporting	Transações aninhadas	Não especificado	Permite liberação de resultados parciais durante a execução	Não especificado
Kangaroo	Divisão de transações no processo de mudança de célula	Não Especificado	Sem garantia de isolamento	Não especificado
AMT	Transações aninhadas	Não especificado	Liberação de resultados parciais antes da efetivação global	Não especificado
Moflex	Divisão de transações no processo de mudança de célula	Regras de junção de subtransações que definem a corretude de uma transação	Sem garantia de isolamento	Não especificado
HiCoMo	Divisão de transações no processo de mudança de célula	Pode-se especificar uma margem de erro para execução de transações	Liberação de resultados parciais antes da efetivação global	Tolera o descarte de atualizações dentro de uma margem de erro
MobileTrans	Permite efetivação de transações mesmo se determinados objetos não puderem ser efetivados	Regras de consistência para acesso a dados	Permite a leitura de dados não atualizados	Tolera a efetivação de transações mesmo se todas as atualizações não puderem ser propagadas para a rede fixa
Mobisnap	Sem flexibilidade	Utiliza informações semânticas dos dados acessados para manter a consistência dos mesmos nos períodos de desconexão	Sem flexibilidade	Não especificado
Pre-serialização	Sem flexibilidade	Sem flexibilidade	Liberação de resultados parciais antes da efetivação global	Sem flexibilidade
Baseado em semântica	Sem flexibilidade	Regras de consistência para acesso a dados; informações semânticas para manter consistência durante a desconexão	Liberação de resultados parciais antes da efetivação global	Sem garantia de durabilidade

2. **Dependências entre transações.** São de dois tipos: dependência de efetivação (*de*)–se uma transação t_i possuísse uma *de* em relação a uma outra t_j, t_i só poderia ser efetivada quando t_j fosse efetivada ou cancelada; dependência de cancelamento (*dc*)–se t_i tivesse uma *dc* em relação a t_j, se t_j fosse cancelada, t_i teria que ser também cancelada.

3. **Visão.** Para cada transação t deve ser definida uma visão. Uma visão determina os objetos e estados de objetos que são visíveis a t em um dado ponto no tempo. Uma visão normalmente é definida como um subconjunto do histórico que satisfaz a um dado predicado.

4. **Conjunto de operações conflitantes.** O conjunto de operações conflitantes de uma transação t (*ConflictSet*) são todas aquelas operações pertencentes ao histórico que podem ser conflitantes em relação a operações invocadas por t. Quando t invoca uma operação, os conflitos desta operação são verificados junto ao *ConflictSet*.

5. **Delegação.** Uma transação t_a pode delegar operações para uma outra transação t_b. Quando t_a delega uma operação *op* para t_b, t_b passa a ser responsável por efetivar ou cancelar *op*.

Esse trabalho demonstrou que através do uso desses cinco blocos, vários modelos de transação estendidos podem ser especificados. Como exemplo, esse trabalho especificou modelos como, transações aninhadas, aninhadas abertas e split/join. Apesar de ter sido um trabalho teórico, esse framework tem sido usado como base para diversos outros trabalhos que implementam frameworks flexíveis de transação.

3.3.2 Transações Bourgogne

Este trabalho [78] apresenta uma extensão para o apoio transacional do Enterprise JavaBeans. As limitações do apoio a transações provido pelo Enterprise JavaBeans foram identificadas e um modelo chamado transações Bourgogne foi proposto para sobrepô-las. As transações Bourgogne visam dar ao Enterprise JavaBeans o apoio a modelos avançados de transações.

Para permitir a definição de transações avançadas, esse trabalho tomou como base a idéia do framework ACTA – de que um modelo de transação arbitrário pode ser especificado usando a definição de eventos significantes da transação (tal como a criação, o inicio, a efetivação e o cancelamento da transação), o estabelecimento de dependências entre transações, delegação e compartilhamento de recursos entre transações. Na prática, o modelo Bourgogne estende o apoio a transações do Enterprise JavaBeans estabelecendo as seguintes premissas:

1. **Delegação de recursos de uma transação para outra.** Uma transação pode mover objetos associados a ela para uma outra transação tal que a transação receptora se torna responsável pelos processos de efetivação e cancelamento dos mesmos.

2. **Compartilhamento de recursos entre transações.** Uma transação pode dar a uma outra o direito de acessar dados reservados para ela. Esta permissão pode ser para escrita ou leitura e pode se restringir a somente um subconjunto das operações de um objeto.

3. **Estabelecimento de dependências entre transações.** Dependências são vínculos condicionais entre transações que podem depender de eventos significantes como a efetivação ou o cancelamento de uma transação. Por exemplo, se uma transação A possui uma dependência de cancelamento em relação a uma transação B, se B for cancelada, A também deve ser cancelada.

3.3.3 Síntese de Plataformas Transacionais

Este trabalho [108] apresenta um método que, dados os requisitos transacionais de uma aplicação, sintetiza uma plataforma que atende a esses requisitos baseado no uso de serviços já existentes. Esse processo de sintetização de plataformas transacionais é realizado em três etapas: especificação dos requisitos transacionais, seleção dos serviços existentes e composição da plataforma.

Para a especificação dos requisitos transacionais deve ser provida uma descrição da arquitetura da aplicação e uma especificação formal das propriedades transacionais usadas pela aplicação. A descrição da arquitetura identifica as interfaces providas e requeridas para cada componente da aplicação assim como as interconexões entre os componentes. Na especificação dos requisitos transacionais, esse método provê a flexibilização das propriedades ACID. Cada propriedade ACID pode ser gradualmente refinada em propriedades mais especializadas. As propriedades de um dado modelo transacional corresponderá à conjunção de um refinamento de cada uma das propriedades ACID. A especificação destas propriedades deve ser feita formalmente através do uso de lógica temporal.

No processo de seleção, dada a especificação formal das propriedades transacionais requeridas, um procedimento verifica e seleciona os serviços de middleware existentes que atendem a cada propriedade especificada. Esse procedimento só é executado com sucesso se, para cada propriedade especificada, existe pelo menos um serviço disponível que a atende.

Depois de selecionados os serviços de middleware, o método usa um procedimento de composição que utiliza esses serviços e a descrição da arquitetura da aplicação para sintetizar a plataforma desejada. Com base nestas informações, esse procedimento gera um

conjunto de *stubs* que intermediará a interconexão entre os componentes da aplicação e os serviços selecionados. Depois de gerados os *stubs*, o método sintetiza a plataforma transacional combinando os serviços selecionados com os componentes funcionais da aplicação.

3.3.4 Monitor de Processamento de Transações Extensível

Este trabalho [5] teve como objetivo permitir a implementação de modelos de transações estendidos a partir de um monitor de processamento de transações (TP-Monitor) comercial existente. No geral, a definição de transações estendidas é feita em uma camada de mais alto nível utilizando operações primitivas providas pelo monitor de processamento de transações.

Nesse trabalho, a reflexão computacional é usada na definição de adaptadores de transações – módulos de software reflexivos que têm a tarefa de revelar aspectos normalmente não visíveis da funcionalidade do TP-Monitor. Cada adaptador corresponde a um aspecto particular do TP-Monitor como, a execução da transação, gerenciamento de trancas, detecção de conflitos e gerenciamento de logs. Um adaptador de transação contém metaobjetos que representam o comportamento desejado para o componente funcional do TP monitor e uma metainterface para controlar o comportamento desse componente. O comportamento do TP-Monitor pode ser mudado (para a definição de modelos de transação estendidos) atualizando-se esses metaobjetos através da metainterface do adaptador. Dessa forma, programadores poderiam implementar modelos de transações alternativos e criar novas interfaces de aplicação de acordo com os requisitos de suas aplicações.

A arquitetura proposta é dividida em dois níveis: o nível base e o metanível. O nível base é onde fica o TP-Monitor e seus componentes funcionais. No metanível ficam os adaptadores de transação. Como o nível base não foi desenvolvido com apoio à reflexão, essa arquitetura foi implementada no metanível com base no uso de eventos propagados pelo nível base. Um evento do nível base é propagado para um dos adaptadores do metanível toda vez que o estado de uma transação está para ser mudado, por exemplo, uma transação está para ser abortada ou efetivada. Ao receber a notificação do evento, o adaptador toma o controle da transação juntamente com as informações que descrevem o evento. A partir destas informações, o adaptador, verifica qual o modelo estendido definido pelos metaobjetos e responde ao evento com chamadas à API do TP-Monitor de acordo com o modelo correspondente.

3.3.5 Transações Estendidas com o Reflective Java

Este trabalho [106] descreve uma arquitetura flexível de apoio a transações baseada no uso do Reflective Java. Nessa arquitetura, serviços de transação podem ser personalizados

para atender necessidades específicas de aplicações de internet. Para se personalizar um serviço de transação, estratégias de implementação são passadas para um *container* na forma de metaobjetos. A troca de um metaobjeto por outro pode representar uma reconfiguração do serviço de transação para atender necessidades específicas das aplicações.

A arquitetura de apoio a transações organiza os metaobjetos em uma estrutura de dois níveis. No primeiro nível ficam os metaobjetos relacionados com a coordenação dos objetos do segundo nível e no segundo nível ficam os metaobjetos responsáveis por tarefas básicas do serviço de transação como, controle de concorrência e gerenciamento da persistência.

Para prover flexibilidade e adaptabilidade de serviços de transação, cada tarefa básica pode possuir diversas implementações representadas através de diferentes metaobjetos. Dessa forma, um serviço de transação pode ser personalizado selecionado-se os metaobjetos mais apropriados para cada tarefa do serviço.

3.3.6 CAGISTrans

CAGISTrans [79] é um framework de apoio a transações que, além de prover flexibilização de serviços de transação, visa a adaptação de transações em relação a aspectos dinâmicos de ambientes cooperativos. Esse trabalho assume que o apoio à adaptação dinâmica é essencial para se obter um maior sucesso na execução de transações em ambientes que podem sofrer mudanças em tempo de execução.

O CAGISTrans divide a especificação de transações em duas partes. A primeira parte é chamada de especificação das características – em que são definidos todos os aspectos fixos da execução de uma transação como, propriedades ACID, critério de corretude, mecanismos e regras que definem o uso dos critérios de corretude. A segunda parte é chamada de especificação da execução – onde são definidos todos os aspectos da transação que poderão sofrer mudança em tempo de execução. A especificação da execução é dividida em três blocos:

1. **Operações de gerenciamento.** Permite a reestruturação dinâmica de transações dando a estas a capacidade de delegar operações entre si.

2. **Operações avançadas.** Permite a definição de operações de mais alto nível baseadas no uso de operações simples de leitura e escrita.

3. **Gerenciamento e controle do comportamento transacional.** Permite a definição de regras de corretude pelo usuário. Basicamente estas regras definem: *(i)* operações que não podem ser executadas concorrentemente; *(ii)* operações que, embora sejam

consideradas conflitantes com as operações definidas em (i), ainda podem ser executadas concorrentemente; *(iii)* seqüências de operações que devem aparecer para se alcançar execuções corretas.

3.3.7 ReflecTS

ReflecTS [46, 4] é uma plataforma reflexiva que provê apoio a diferentes serviços de transação concorrentes. O principal objetivo dessa plataforma é o de atender a diversidade de requisitos das aplicações permitindo que eles escolham o serviço mais adequado.

Através de uma arquitetura chamada TSenvironment [48, 49], o ReflecTS permite que diferentes serviços de transação possam ser instalados, modificados e usados concorrentemente de acordo com as necessidades das aplicações. Cada serviço de transação é considerado um componente que pode ser desenvolvido independentemente e que pode ser composto por componentes menores chamados componentes de serviço. Um componente de serviço normalmente implementa uma tarefa bem definida de um serviço de transação como, controle de concorrência, efetivação ou recuperação.

Quando uma aplicação requisita apoio transacional ao ReflecTS, ela fornece a especificação formal do serviço de transação a ser usado. A partir desta especificação, o ReflecTS realiza as seguintes etapas: *(i)* seleciona o serviço de transação adequado dentre os serviços disponíveis, *(ii)* verifica a compatibilidade do serviço com relação aos gerenciadores de recursos e com relação aos demais serviços de transação correntemente ativos. Se estas etapas forem cumpridas com sucesso, o ReflecTS ativa o serviço de transação requisitado e o disponibiliza à aplicação requisitante.

Para permitir introspecção e reconfiguração, o ReflecTS disponibiliza um conjunto de metainterfaces. Através dessas metainterfaces podem-se realizar as seguintes ações *(i)* adicionar, remover ou modificar os componentes do ReflecTS, *(ii)* adicionar, remover e reconfigurar serviços de transação e *(iii)* permitir a seleção de serviços de transação dentre os disponíveis. Para evitar que estas reconfigurações comprometam a integridade da plataforma, cada mudança realizada é comparada a um conjunto de arquiteturas válidas pré-definidas.

3.3.8 Discussão

Apesar de os trabalhos descritos acima permitirem a definição de propriedades transacionais personalizadas, eles possuem algumas limitações. O ACTA é um modelo meramente teórico. Os demais trabalhos, com exceção do CAGISTrans, não consideram os requisitos transacionais de ambientes dinâmicos – que requer a capacidade de reconfiguração em tempo de execução. Em especial, o CAGISTrans se limita a atender requisitos específicos de ambientes cooperativos.

Embora esses trabalhos tentem prover um apoio transacional personalizável, ainda há uma lacuna no que diz respeito ao apoio à reconfiguração em tempo de execução necessária em ambientes dinâmicos. A busca de meios de preencher essa lacuna é um dos pontos abordados nesta tese. Um dos elementos que ajudaram na busca de uma solução foi o estudo de middlewares abertos.

3.4 Middlewares Abertos

Middlewares têm emergido como um componente arquitetural importante no apoio a aplicações distribuídas. A função do middleware é apresentar um modelo de programação unificado aos desenvolvedores e mascarar problemas de heterogeneidade e de distribuição [11]. Além disso, middlewares normalmente disponibilizam um conjunto de serviços de alto nível que podem ser usados para facilitar o desenvolvimento de aplicações. Dentre esses serviços está o de transações. A importância do conceito de middleware é refletida na crescente popularidade de seus representantes como, CORBA, DCOM/.NET e Java RMI/J2EE.

Com os recentes avanços em sistemas distribuídos, móveis e onipresentes, surgiram novas infra-estruturas computacionais caracterizadas por um alto dinamismo. Esse dinamismo exige das aplicações a capacidade de se adaptar às mudanças ocorridas no ambiente [52]. O problema é que middlewares convencionais não têm a capacidade de adaptação e reconfiguração necessárias para lidar com os aspectos introduzidos por estas novas infra-estruturas. Uma das soluções propostas para transpor estas limitações foi o uso da chamada implementação aberta [51]. Através dela, aspectos da implementação do middleware seriam revelados dando a esses a capacidade de reconfiguração, tanto para atender aos requisitos das aplicações quanto para melhor se adequar ao ambiente. Esta é a característica principal dos chamados middlewares abertos [54, 11].

São vários os trabalhos que podem ser considerados parte integrante desse novo conceito de middleware, dentre eles estão o Quarterware, o OpenCORBA, o DynamicTAO, o OpenORB, e o ReMMoC. Nas seções a seguir, será apresentada uma breve descrição desses trabalhos.

3.4.1 Quarterware

O Quarterware [98] propõe um framework para a construção de middleware flexível e de alto desempenho. Para permitir apoio a diferentes funcionalidades, o Quarterware abstrai as propriedades comuns presentes em middlewares convencionais permitindo que versões especializadas possam ser implementadas para satisfazer necessidades específicas das aplicações. Para atender os requisitos de desempenho das aplicações o Quarter-

ware permite otimizações com relação ao gerenciamento de memória e a comunicação. O Quarterware também disponibiliza interfaces reflexivas que tornam as aplicações aptas a interagir com o middleware e reconfigurá-lo dinamicamente [97].

3.4.2 OpenCORBA

O OpenCORBA [58] é um ORB que revela aspectos da implementação do CORBA através da reflexão para permitir a reconfiguração dinâmica. Esses aspectos são representados através de metaclasses, cuja mudança dinâmica permite modificar os mecanismos do ORB por elas representados. Os aspectos do ORB que podem ser reconfigurados são: os mecanismos de invocação remota, a verificação de tipos do IDL e o gerenciamento de exceções.

3.4.3 DynamicTAO

O DynamicTAO [87, 53] é um trabalho que utilizou reflexão computacional para permitir a reconfiguração dinâmica a partir de um middleware chamado TAO. O DynamicTAO exporta uma interface reflexiva que pode ser usada por objetos locais ou remotos para instalar, desinstalar ou configurar componentes. O DynamicTAO também propõe mecanismos que visam evitar que estas reconfigurações comprometam a integridade do middleware como um todo.

3.4.4 OpenORB

O OpenORB [9] é um middleware que foi projetado para ser altamente configurável e reconfigurável dinamicamente de forma a prover apoio a aplicações com requisitos dinâmicos. Diferentemente do DynamicTAO e do OpenCORBA que foram projetados com base na implementação de middlewares já existentes, o OpenORB foi inteiramente projetado e implementado independentemente de estruturas de middleware já existentes. Instâncias personalizadas do OpenORB podem ser configuradas selecionando os componentes apropriados. A arquitetura do OpenORB permite que componentes possam ser identificados e reconfigurados em tempo de execução.

A reconfigurabilidade dinâmica do OpenORB é obtida através de um extensivo uso de reflexão com uma clara separação entre o nível base e o metanível. Enquanto o nível base consiste de componentes que implementam os serviços usuais do middleware, o metanível expõe a implementação desses componentes através da reflexão permitindo inspeção e adaptação.

3.4.5 ReMMoC

O ReMMoC [73, 37, 38] é um middleware direcionado a lidar com aspectos do ambiente de computação móvel. O ReMMoC foi projetado para ser configurável, reconfigurável e para dar a clientes móveis a capacidade de interoperar transparentemente com diversos serviços independentemente do protocolo de comunicação. Para permitir configuração e reconfiguração, o ReMMoC foi projetado com base no modelo de componentes OpenCOM, reflexão e framework de componentes. A estrutura do ReMMoC usa dois frameworks de componentes: binding framework – responsável pela interoperabilidade entre os serviços implementados em diferentes paradigmas de middleware (IIOP, SOAP); discovery framework – responsável pela localização de serviços móveis independentemente do protocolo de localização (ex: SLP, UPnP e Jini). Esses frameworks são compostos por grupos de componentes e cada grupo implementa um tipo de protocolo. Outros protocolos também podem ser implementados e adicionados ao framework.

Para permitir que clientes sejam desenvolvidos independentemente do tipo de middleware com o qual eles interagirão, o ReMMoC disponibiliza uma API baseada em uma abstração dos diferentes tipos de serviço. Esta API pode ser usada tanto para a localização de serviços quanto para o acesso a serviços conhecidos. Quando é utilizada uma operação de localização da API, o ReMMoC utiliza transparentemente os diversos protocolos de localização implementados e devolve a lista de serviços encontrados. Ao receber uma operação de acesso a serviços através da API, o ReMMoC mapeia essa chamada para o protocolo do middleware correspondente.

3.4.6 Discussão

Uma característica em comum que pode ser encontrada entre esses middlewares abertos apresentados é a combinação do uso de reflexão, componentes e framework de componentes como técnicas para se obter implementações mais flexíveis e com capacidade de configuração dinâmica. Com isso, a partir de uma única implementação podem ser derivadas várias especializações capazes de atender os requisitos de uma gama maior de aplicações.

Apesar dos avanços alcançados pelo conceito de middleware aberto, algumas limitações ainda precisam ser superadas. As soluções apresentadas restringem-se basicamente aos problemas da heterogeneidade e da distribuição. Soluções para outros serviços importantes que podem estar integradas a plataformas de middleware, como serviços de transações, não são apresentadas.

Outra limitação das soluções correntes em middlewares abertos está na estrutura utilizada para representá-los. No geral, a representação de middlewares abertos conta somente com elementos tradicionais dos modelos de desenvolvimento baseados em compo-

nentes: componentes e conexões. Essa representação é exibida ao usuário do middleware (desenvolvedor da aplicação) através de interfaces reflexivas para que ele possa fazer as requeridas configurações. Porém, possuir uma estrutura baseada em componentes que pode ser manipulada via reflexão não é o suficiente para encorajar os usuários a configurar o middleware aberto. A principal preocupação desse usuário está em utilizar os serviços do middleware para desenvolver aplicações. Se o processo de configuração do middleware for complexo, o usuário poderá perder o interesse em utilizá-lo, já que um dos principais motivos para se adotar o middleware é justamente o de facilitar o processo de implementação de aplicações.

Uma tentativa de facilitar o processo de configuração de middlewares comumente adotada pelos trabalhos em middleware abertos apresentados é a utilização de framework de componentes [38, 46, 10, 9]. Porém, nesses trabalhos, um framework de componentes é simplesmente um invólucro que contém uma sub-arquitetura composta de componentes e conexões. Dessa maneira, essa noção de framework de componentes pode ser resumida na simples noção de componente composto. É certo que componentes compostos podem ajudar na organização da estrutura de um middleware, mas eles não são o suficiente para torná-lo facilmente configurável para o usuário.

Por fim, o estudo de middlewares abertos pode ser um caminho promissor para se chegar a serviços de transações flexíveis e configuráveis o bastante para atender os variáveis requisitos de ambientes dinâmicos, como a computação móvel. Porém, novas formas de representação de middleware devem ser estudadas para que o processo de configuração esteja ao alcance dos interesses dos usuários. Antes de chegar à proposta de serviços de transações abertos, esta tese perpassou por esse problema apresentando novos elementos de representação de middlewares abertos que prometem facilitar o processo de configuração

Capítulo 4

O Modelo Motherboard

No capítulo anterior, foram apresentadas algumas limitações encontradas em trabalhos que apresentam serviços de transação para ambientes dinâmicos. Essas limitações estão relacionadas com deficiências em atender requisitos como, flexibilidade, extensão, adaptação e usabilidade. Na busca de avanços em serviços de transações que atendem esses requisitos, este capítulo propõe o modelo *motherboard*.

4.1 Middleware Tradicional e Limitações

Com o surgimento de ambientes distribuídos, a complexidade de implementar aplicações aumentou consideravelmente. O programador passou a ter que lidar com detalhes complicados de comunicação em rede, chamadas remotas de métodos, serviços de nomes [1], instanciação de serviços, além de outras questões envolvidas como, controle de concorrência, recuperação de falhas e persistência. Como uma solução encontrada para lidar com essa complexidade, surgiu o conceito tradicional de middleware. A idéia seria a de encapsular todos esses detalhes em uma "caixa preta" provendo serviços de alto nível que estariam prontos para serem utilizados pelos programadores de aplicação. Essa foi a idéia inicialmente adotada por middlewares populares como o CORBA, o J2EE e o .NET.

Apesar do sucesso que essa solução representou no intuito de facilitar o desenvolvimento de aplicações distribuídas, algumas limitações foram se tornando evidentes. Como essa solução utilizava a abordagem "caixa preta", ela não permitia que ajustes necessários para algumas aplicações com requisitos específicos fossem realizados. Além disso, o middleware nessa abordagem era fornecido como um bloco único – mesmo que o programador fosse utilizar somente um pequeno subconjunto das funções do middleware, ele teria que arcar com os custos de obter e manter o bloco inteiro.

[1] Em inglês: naming

4.2 Soluções em Middleware Aberto e Limitações

Para contornar as limitações apresentadas na solução de middleware tradicional (fechado), foi proposta a noção de middleware aberto em que os detalhes complicados de ambientes distribuídos seriam encapsulados mas seriam permitidas a inspeção e reconfiguração da estrutura e dos mecanismos internos do middleware. Paradoxalmente, essa solução trouxe de volta o problema da complexidade que o middleware tradicional visou eliminar. Neste novo caso, essa complexidade está relacionada à dificuldade existente em se analisar a estrutura interna do middleware, identificar os pontos onde os ajustes deveriam ser feitos e de efetuar os próprios ajustes. Isso se deve às seguintes características comumente encontradas em projetos de middleware aberto:

(i) Possuem uma estrutura semelhante a dos projetos de middleware fechados;

(ii) Não especificam claramente quais os pontos ou aspectos do middleware são passíveis de reconfiguração;

(iii) Apresentam detalhes complexos e dispensáveis da estrutura do middleware aos usuários (programadores de aplicação);

(iv) Não indicam ao usuário quais os tipos de reconfiguração e como eles devem ser aplicados.

Essas características podem induzir uma série de desvantagens:

- ao se analisar um projeto de middleware fica difícil distinguir se o mesmo é aberto ou fechado;
- pode tornar a tarefa de reconfiguração complexa e enfadonha;
- pode induzir erros de reconfiguração pela má interpretação com relação à estrutura do middleware.

Para reduzir os impactos causados pelas deficiências encontradas em projetos de middleware, este trabalho propõe um novo modelo para a especificação de middleware aberto. A solução adotada no modelo proposto é baseada na idéia de placas mãe em hardware (*motherboard*).

4.3 Placas Mãe: Lidando com a Complexidade em Hardware

Uma placa mãe, também denominada *motherboard* ou *mainboard* [60], é um circuito impresso responsável por fazer a interligação entre os elementos que constituem arquiteturas de hardware. As placas mãe foram propostas como solução ao problema da alta complexidade para se interconectar manualmente esses elementos. A solução encontrada foi a de pré-montar essas interconexões através de um circuito impresso fixo. Para dar à arquitetura uma certa flexibilidade (configurabilidade), foram incluídos elementos abertos (slots, sockets e jumpers) ao circuito impresso. Basicamente uma placa mãe é composta por pelo menos os seguintes elementos:

Componentes. elementos que possuem uma função bem determinada – pode ser simples (transistores, diodos, capacitores, etc.) ou composto (controladores e outros circuitos de hardware).

Slots. são espaços reservados para a conexão de circuitos de hardware (como memória, placas de áudio e vídeo).

Sockets. são um tipo especial de slot onde se pode conectar diferentes processadores de uma mesma família.

Jumpers. são conexões que podem ser abertas/fechadas através da retirada/introdução de um pino.

Conectores. são circuitos impressos que interconectam os demais elementos para a formação da arquitetura desejada.

Atualmente, essa solução tem sido adotada largamente, incluindo desde arquiteturas de computadores pessoais a mainframes. Seguem algumas das razões pelas quais o conceito de placa mãe se tornou tão popular: *(i)* esconde a complexidade das interconexões das arquiteturas de hardware em um circuito impresso; *(ii)* favorece a fácil identificação e manipulação dos elementos abertos à reconfiguração (sockets, slots and jumpers) para a criação de arquiteturas personalizadas; *(iii)* famílias de componentes (processadores, memórias, placas de video e áudio) podem ser desenvolvidas para um mesmo slot/socket; *(iv)* placas-mãe podem ser desenvolvidas com diferentes "níveis de abertura" – podem ser usados mais elementos fixos (componentes *onboard*) e menos elementos abertos (slots, sockets e jumpers) e vice-versa.

4.4 O Modelo Motherboard: lidando com a complexidade em middleware abertos

Inspirado na solução adotada em placas mãe em hardware, foi criado o modelo motherboard [86, 81]. Esse modelo visa lidar com aspectos relativos à complexidade existente em se configurar middlewares abertos. A idéia chave para a elaboração do modelo motherboard está na separação de atribuições entre os principais grupos interessados: o desenvolvedor do middleware e o usuário do middleware (programador de aplicações).

Atribuições do Desenvolvedor

- Definir todos os detalhes quanto à estrutura e funcionalidade do middleware.
- Definir precisamente os aspectos relacionados com a reconfiguração do middleware: quais partes são reconfiguráveis e como fazer a reconfiguração.

Atribuições do Usuário

- Identificar as possibilidades de reconfiguração que o middleware pode assumir.
- Configurar o middleware de forma a atender os requisitos desejados.
- Usar os serviços do middleware no desenvolvimento de aplicações.

Considerando as diferentes atribuições que há entre esses dois grupos de interessados, o modelo motherboard possibilita a definição de middleware aberto em duas diferentes visões: *(i)* a visão do desenvolvedor do middleware (MD-view) e *(ii)* a visão do usuário do middleware (MU-view) descritas a seguir.

4.4.1 Visão do Desenvolvedor de Middleware

A visão do desenvolvedor do middleware (MD-view) é a parte do modelo motherboard usado para se definir as atribuições do desenvolvedor do middleware: definir os detalhes da arquitetura e os pontos de configuração. Em particular, uma MD-view define um nível base de middleware a partir do qual se podem derivar diferentes configurações. Fazendo uma analogia às placas mãe em hardware, o MD-view é a "placa mãe" do middleware.

O modelo motherboard provê os seguintes elementos para a definição do MD-view: componentes, conexões, interfaces, jumpers, slots e demultiplexadores. Os três primeiros são elementos comumente encontrados em definições tradicionais de middleware baseado em componentes. Os demais são elementos introduzidos pelo modelo motherboard para definir os pontos de configurabilidade do middleware. Uma breve definição de cada um desses elementos é dada como segue:

Componente. Um componente é uma unidade de computação que implementa uma tarefa bem definida. Um componente é dito simples quando é monolítico e composto quando possui uma arquitetura interna formada por outros componentes interligados.

Interface. Uma interface define operações a partir das quais componentes interagem. Em um componente, as interfaces podem ser providas ou requeridas. Interfaces providas são aquelas pelas quais um componente provê seus serviços (a outros componentes ou a usuários). Interfaces requeridas são aquelas pelas quais um componente usa serviços de outros componentes.

Conexão. Uma conexão é definida pela interligação entre a interface requerida de um componente e a interface provida de um outro componente. Através de uma conexão, o primeiro componente é capaz de utilizar serviços providos pelo segundo.

Jumper. Um jumper é um tipo especial de conexão que pode ser ativado ou desativado. Um jumper ativado cria uma conexão entre os dois componentes envolvidos. No jumper desativado, essa conexão é desfeita.

Slot. Um slot é um elemento que equivale a um espaço reservado à conexão de componentes prontos. Um slot é capaz de prover um ponto de configurabilidade no qual diferentes variações implementadas como componentes independentes podem ser conectadas.

Um slot pode ser simples ou múltiplo. No primeiro caso, somente um componente pode ser conectado por vez. No segundo caso, mais de um componente podem ser conectados ao mesmo tempo para serem usados em paralelo ou selecionados em tempo de execução.

A definição de um slot pode incluir interfaces providas e requeridas. É o conjunto dessas interfaces que irá definir se um componente é compatível com um slot. Um componente é compatível com um dado slot se o conjunto formado por suas interfaces providas e requeridas é equivalente ao do slot.

Demultiplexador. O demultiplexador é um elemento especial que mapeia uma interface requerida de um componente com à interface provida de dois ou mais componentes. Assim definido, o demultiplexador permite que, uma vez fornecida uma chave de seleção, a interface provida correspondente seja conectada à interface requerida.

Notação Gráfica da MD-View

Para facilitar o projeto da visão do desenvolvedor do middleware, foi definida uma notação gráfica. Essa notação inclui uma representação para cada elemento de uma arquitetura do

tipo MD-View: componente simples, componente composto, interface provida, interface requerida, demultiplexador, slot, jumper ativado e jumper desativado (Figura 4.1). Nessa notação, para representar um slot simples ou múltiplo coloca-se o número indicador "1" ou o símbolo indicador "*" sobrescrito logo após o seu identificador, respectivamente. Se após o identificador do slot não houver nenhum dos indicadores ("1" ou "*") sobrescritos, o slot é considerado simples.

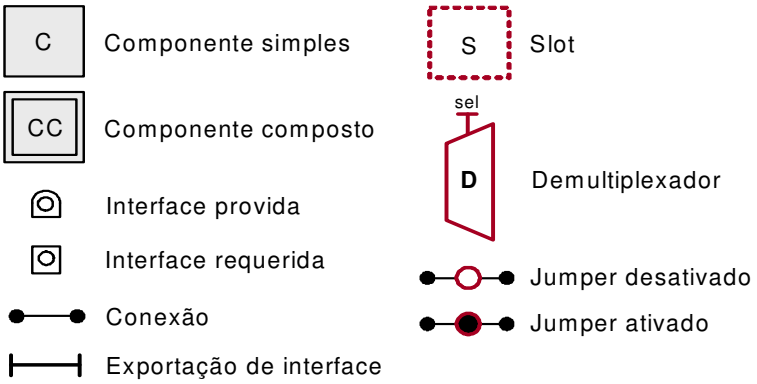

Figura 4.1: Notação Gráfica da MD-View

Para ilustrar o uso da notação gráfica da MD-View, a Figura 4.2(a) apresenta um exemplo simples. Nesse exemplo, uma arquitetura do tipo MD-View é definida pelos seguintes elementos: componentes simples C_1, C_2, C_3 e C_4, componente composto CC_1, slot S_1, jumper J_1 e conexões B_1, B_2, B_3, B_4 e B_5. Como se pode observar, nessa arquitetura foram definidos três elementos abertos a partir dos quais podem ser derivadas diferentes configurações – pode-se conectar um componente no slot S_1, ativar ou desativar o jumper J_1 e selecionar a porta de saída do demultiplexador D_1.

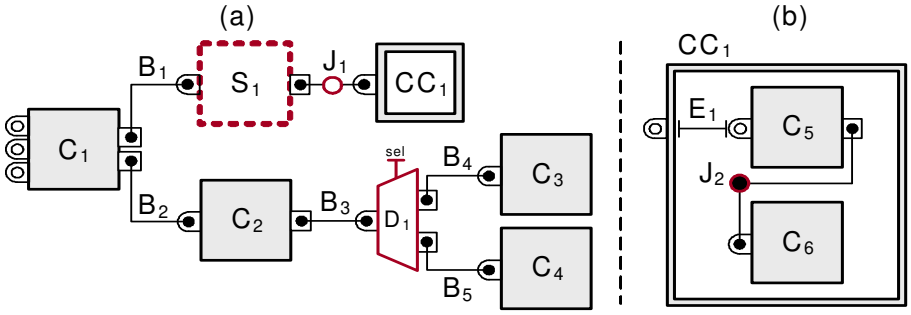

Figura 4.2: Exemplo de Uso da Notação Gráfica da MD-View

A Figura 4.2(b) mostra como a arquitetura interna de um componente composto pode ser representada graficamente. Em particular, foi representada a arquitetura do

componente CC_1 – composto pelos componentes C_5 e C_6 e pelo jumper J_2. Pode-se notar que nessa arquitetura, o componente C_5 exporta uma interface – isso significa que essa interface do componente C_5 passa a ser uma interface do componente composto CC_1.

Notação XML do MD-View

Além da notação gráfica, foi proposta uma notação XML para a representação de uma MD-View. De forma a mostrar a estrutura formal dessa representação XML, a Figura 4.3 apresenta o metamodelo da mesma.

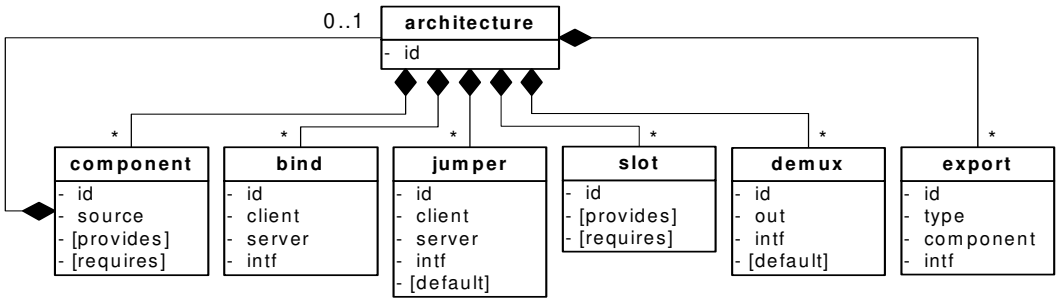

Figura 4.3: Metamodelo da Notação XML da MD-View

Cada entidade representada nesse metamodelo, indica o nome da tag XML e o seu conjunto de atributos. Segue a descrição de cada uma das entidades representadas.

1. **architecture** – representa uma arquitetura MD-View. Pode ser definida por uma composição de componentes, conexões, jumpers, slots, demultiplexadores e interfaces exportadas.

   - *Tag*: `<architecture id=''> ... </architecture>`
   - *Atributos*:
     - **id**: identificador da arquitetura MD-View.

2. **component** – representa um componente da arquitetura MD-View.

   - *Tag*: `<component id='' source='' [provides=''] [requires='']/>`
   - *Atributos*:
     - **id**: identificador do componente.
     - **source**: referência ao componente (ex: nome canônico da classe que o implementa).

  - **provides**: atributo opcional que indica as interfaces providas pelo componente. Se existir mais de uma interface, estas devem ser separadas por vírgulas.
  - **requires**: atributo opcional que indica as interfaces requeridas pelo componente. Se existir mais de uma interface, estas devem ser separadas por vírgulas.

3. **bind** – representa uma conexão entre uma interface requerida e uma interface provida de elementos da arquitetura.

   - *Tag*: `<bind id='' client='' server='' intf=''/>`
   - *Atributos*:
     - **id**: identificador da conexão.
     - **client**: identificador do elemento que possui a interface requerida.
     - **server**: identificador do elemento que possui a interface provida.
     - **intf**: nome da interface utilizada na conexão.

4. **jumper** – representa um jumper entre uma interface requerida e uma interface provida de elementos da arquitetura.

   - *Tag*: `<jumper id='' client='' server='' intf='' [default='']/>`
   - *Atributos*:
     - **id**: identificador do jumper.
     - **client**: identificador do elemento que possui a interface requerida.
     - **server**: identificador do elemento que possui a interface provida.
     - **intf**: nome da interface utilizada na conexão.
     - **default**: atributo opcional que indica o estado "default" do jumper ("on" ou "off").

5. **slot** – representa um slot na arquitetura MD-View.

   - *Tag*: `<slot id='' [provides=''] [requires='']/>`
   - *Atributos*:
     - **id**: identificador do slot.
     - **provides**: atributo opcional que indica as interfaces providas pelo slot. Se existir mais de uma interface, estas devem ser separadas por vírgulas.
     - **requires**: atributo opcional que indica as interfaces requeridas pelo slot. Se existir mais de uma interface, estas devem ser separadas por vírgulas.

6. **demux** – representa um demultiplexador na arquitetura MD-View.
   - *Tag*: `<demux id='' out='' intf='' [default='']/>`
   - *Atributos*:
     - **id**: identificador do demultiplexador.
     - **out**: lista os identificadores das conexões de saída do demultiplexador. Esses identificadores dever ser separados por vírgula.
     - **intf**: interface das conexões envolvidas com o demultiplexador.
     - **default**: atributo opcional que indica a conexão "default" de saída do demultiplexador.
7. **export** – representa a exportação de interface de subcomponentes que fazem parte da arquitetura interna de um componente composto. Uma vez exportada a interface de um subcomponente, esta passa a ser uma interface externa do próprio componente composto.
   - *Tag*: `<export id='' intf='' type='' component=''/>`
   - *Atributos*:
     - **id**: identificador da exportação.
     - **intf**: interface que será exportada.
     - **type**: tipo da interface exportada – "provida" ou "requerida".
     - **componente**: identificador do subcomponente do qual será exportada a interface.

Para exemplificar o uso da notação XML, foi representada abaixo a arquitetura ilustrada na Figura 4.2.

```
<architecture id='arch1'>
   <component id='C1' source='br.C1'/>
   <component id='C2' source='br.C2'/>
   <component id='C3' source='br.C3'/>
   <component id='C4' source='br.C4'/>
   <component id='CC1'>
      <architecture id=arch1_1>
         <component id='C5' source='br.C5'/>
         <component id='C6' source='br.C6'/>
         <jumper id='J2' client='C5' server='C6' intf='Int5' default='on'/>
         <export id='E1' intf='Int4' type='provided' component='C5' />
      </architecture>
```

```
    </component>
    <slot id='S1' provides='Int1' requires='Int4'/>
    <demux id='D1' out='B4, B5' intf='Int3' default='B4'/>
    <bind id='B1' client='C1' server='S1' intf='Int1'/>
    <bind id='B2' client='C1' server='C2' intf='Int2'/>
    <bind id='B3' client='C2' server='D1' intf='Int3'/>
    <bind id='B4' client='D1' server='C3' intf='Int3'/>
    <bind id='B5' client='D1' server='C4' intf='Int3'/>
    <jumper id='J1' client='S1' server='CC1' intf='Int4' default='off'/>
</architecture>
```

4.4.2 Visão do Usuário de Middleware

A visão do usuário do middleware (MU-View) é a parte do modelo motherboard em que o usuário (programador de aplicação) configura o middleware de acordo com os requisitos desejados. Na MU-View, ao invés de ter que lidar com detalhes complexos da estrutura interna do middleware, o usuário só precisa manipular os pontos abertos previamente definidos pelos desenvolvedores do middleware como uma MD-View.

Na prática, a definição de uma MU-View envolve um conjunto das seguintes tarefas:

Instanciar componentes. O processo de definição de uma MU-View pode envolver a instanciação de componentes. Normalmente, antes da tarefa de conectar um novo componente em um determinado slot, esse componente precisa ser instanciado anteriormente.

Conectar componentes em slots. Depois de instanciados, os componentes que implementam os requisitos desejados podem ser conectados nos slots disponíveis.

Ativar ou desativar jumpers. Os jumpers disponíveis podem ser ativados ou desativados de acordo com os requisitos desejados para o middleware.

Selecionar saída de demultiplexadores. Para cada demultiplexador disponível pode-se definir qual a conexão de saída que deverá ser usada na arquitetura. Na prática, selecionar uma conexão de saída, significa conectar uma interface requerida de um determinado componente com a interface provida de um outro componente específico.

Notação Gráfica da MU-View

A notação gráfica da MU-View enfoca um subconjunto da notação da MD-View representado pelos elementos abertos disponíveis (jumpers, slots e demultiplexadores). Na

prática, a notação da MU-View visa representar graficamente o resultado da manipulação dos elementos abertos previamente definidos na MD-View. Essa notação é apresentada na Figura 4.4.

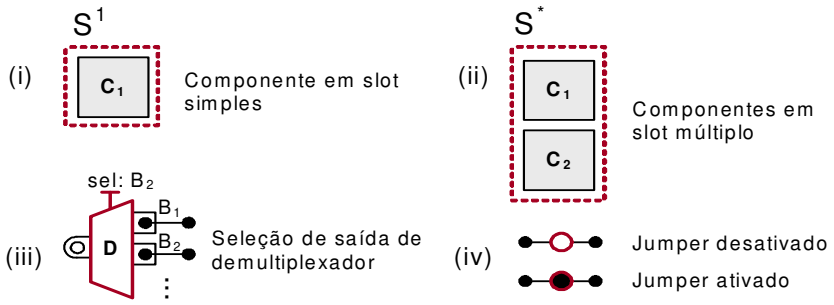

Figura 4.4: Notação Gráfica da MU-View

A notação da MU-View mostra como devem ser representados: *(i)* um componente conectado em um slot simples; *(ii)* mais de um componente conectado em um slot múltiplo; *(iii)* seleção de conexão de saída de um demultiplexador (representada pelo símbolo "sel:" seguido do identificador da conexão selecionada); *(iv)* ativação ou desativação de jumpers.

As Figuras 4.5 e 4.6 apresentam dois exemplos de como a manipulação dos elementos abertos de uma arquitetura pode ser representada graficamente. A MD-View usada como base para esses exemplos é a apresentada na Figura 4.2.

No primeiro exemplo é mostrada uma MU-View com a seguinte configuração: componente C_7 conectado no slot S_1, jumper J_1 ativado e saída B_4 do demultiplexador ativada. Além disso, esse exemplo mostra a configuração da arquitetura interna do componente composto CC_1. Nesse caso, o jumper J_2 do componente CC_1 foi desativado.

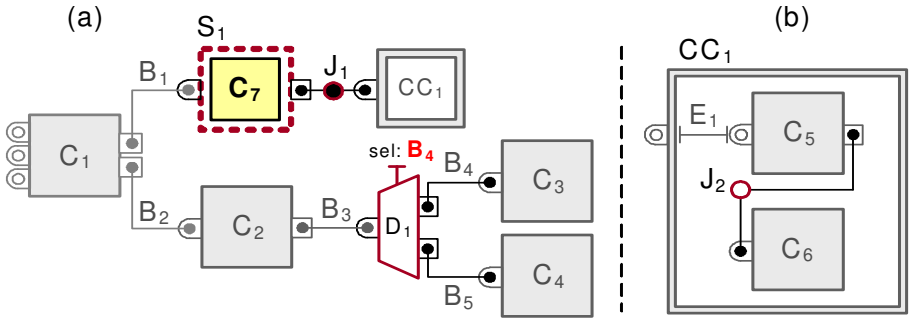

Figura 4.5: Notação Gráfica da MU-View (Primeiro Exemplo)

No segundo exemplo é mostrada uma outra MU-View com a seguinte configuração: componente C_8 conectado no slot S_1, jumper J_1 desativado e saída B_5 do demultiplex-

ador ativada. Nesse exemplo, o jumper J_2 do componente composto CC_1 foi mantido desativado.

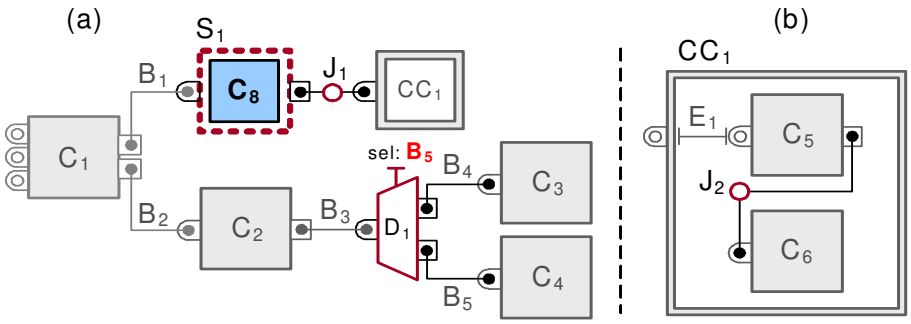

Figura 4.6: Notação Gráfica da MU-View (Segundo Exemplo)

Notação XML da MU-View

Para a representação da MU-View, também foi proposta uma notação XML. De forma a mostrar a estrutura formal dessa representação XML, a Figura 4.7 apresenta o metamodelo da mesma.

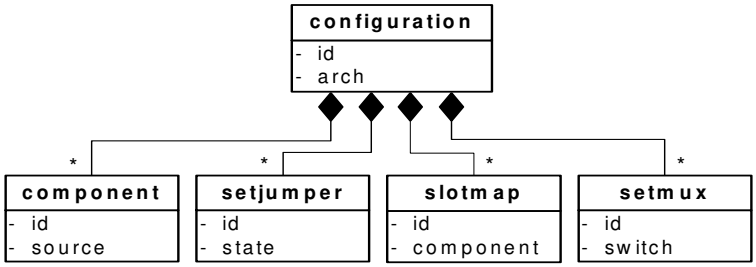

Figura 4.7: Metamodelo da Notação XML da MU-View

Cada entidade representada nesse metamodelo, indica o nome da tag XML e o seu conjunto de atributos. Segue a descrição de cada uma das entidades representadas.

1. **configuration** – representa uma configuração MU-View. Pode ser definida por uma composição de tags de criação de componentes e de manipulação de elementos abertos (jumpers, slots e demultiplexadores).

   - *Tag*: `<configuration id='' arch=''> ... </architecture>`
   - *Atributos*:
     - **id**: identificador da configuração MU-View.

  - **arch**: identificador da arquitetura MD-View à qual a configuração MU-View será aplicada.

2. **component** – representa a instanciação de um componente que é usado na configuração MU-View.
   - *Tag*: `<component id='' source=''/>`
   - *Atributos*:
     - **id**: identificador do componente.
     - **source**: referência ao componente (ex: nome canônico da classe que o implementa).

3. **setjumper** – representa a ativação ou desativação de um determinado jumper.
   - *Tag*: `<setjumper id='' state=''/>`
   - *Atributos*:
     - **id**: identificador do jumper.
     - **state**: define se o estado do jumper será ativado ("on") ou desativado ("off").

4. **slotmap** – representa a ação de conectar um componente em um determinado slot.
   - *Tag*: `<slotmap id='' component=''/>`
   - *Atributos*:
     - **id**: identificador do slot.
     - **component**: identificador do componente a ser conectado no slot.

5. **setmux** – representa a seleção de uma determinada conexão de saída de um demultiplexador.
   - *Tag*: `<setmux id='' switch=''/>`
   - *Atributos*:
     - **id**: identificador do demultiplexador.
     - **switch**: identificador da conexão de saída a ser selecionada.

Para mostrar o uso dessa notação XML, foram montadas três configurações. A primeira configuração mostrada a seguir representa a MU-View apresentada na Figura 4.5(a).

```
<configuration id='conf1' arch='arch1'>
   <component id='C7' source='br.C7'/>
   <slotmap id='S1' component='C7'/>
   <setmux id='D1' switch='B4'/>
   <setjumper id='J1' state='on'/>
</configuration>
```

A segunda configuração mostrada a seguir apresenta a MU-View para a arquitetura interna do componente composto CC1 apresentado na Figura 4.5(b). Pode-se notar que a configuração da arquitetura de um componente composto é representada de forma independente da arquitetura da qual o componente pertence.

```
<configuration id='conf1_1' arch='arch1_1'>
   <setjumper id='J2' state='off/>
</configuration>
```

A terceira configuração mostrada a seguir apresenta a MU-View representada na Figura 4.6(a).

```
<configuration id='conf2' arch='arch1'>
   <component id='C8' source='br.C8'/>
   <slotmap id='S1' component='C8'/>
   <setmux id='D1' switch='B5'/>
   <setjumper id='J1' state='off'/>
</configuration>
```

Capítulo 5

A Plataforma MotherFrame

Com o intuito de verificar e de validar o modelo Motherboard, ele foi implementado em uma plataforma chamada MotherFrame. Na prática, a plataforma MotherFrame é um ambiente de execução no qual o modelo Motherboard pode ser usado para a definição de serviços de middleware como o de transações. Nesta seção, serão discutidas primeiramente algumas decisões que guiaram o projeto e a implementação dessa plataforma. Em seguida será apresentada uma visão geral do seu funcionamento. Por fim, serão apresentados aspectos da sua implementação.

5.1 Decisões de Desenvolvimento

O processo de desenvolvimento da plataforma MotherFrame foi precedido por duas questões fundamentais. A primeira questão é a que segue:

(1) *Dever-se-ia adotar a abordagem clássica de desenvolvimento – pautada no encadeamento sistemático de fases bem definidas (análise, projeto, implementação e testes) – ou utilizar uma abordagem baseada na prototipagem – na qual se definem os requisitos iniciais e dá-se andamento à implementação de um protótipo para a instigação dos demais requisitos?*

Sobre essa questão, foi decidida a adoção da abordagem de prototipagem. As razões que guiaram essa decisão são:

i. São conhecidas diversas limitações da abordagem clássica. Uma dessas limitações consiste justamente na dificuldade em que se há em se seguir sistematicamente as fases propostas. Por exemplo, é comum aparecerem novos requisitos durante as fases implementação e testes do sistema em questão exigindo mudanças nas fases anteriores a essas. Nesse contexto, a prototipagem é uma técnica que possibilita o levantamento

desses "requisitos ocultos" que só são visualizados no momento da implementação e dos testes.

ii. Como a plataforma MotherFrame implementa um modelo novo – o Motherboard – seria esperado o surgimento de novos requisitos durante o curso das fases de projeto, implementação e testes. Dessa forma, a prototipagem poderia entrar como uma técnica de refinamento do próprio modelo Motherboard.

A segunda questão que precedeu o processo de desenvolvimento da plataforma foi:

(2) *A plataforma MotherFrame deveria ser implementada tomando-se como base um ambiente de execução de componentes já existente ou deveria ser implementada como um ambiente de execução totalmente novo?*

Sobre essa questão, foi decidida a implementação da plataforma tomando-se como base um ambiente de execução já existente.

i. Implementar a plataforma tomando-se como base um ambiente de execução de componente já existente traria benefícios como: os sistemas já implementados no ambiente base adotado permaneceriam compatíveis com a plataforma MotherFrame; a plataforma MotherFrame se tornaria fácil de ser utilizada por aqueles que já possuíssem algum conhecimento acerca do ambiente base adotado.

ii. Os elementos arquiteturais providos por ambientes tradicionais de componentes são um subconjunto claro dos elementos incluídos no modelo Motherboard (Figura 5.1). Isso favorece a implementação da plataforma MotherFrame tomando como base um ambiente de componentes já existente. No geral, a tarefa seria a de acrescentar ao ambiente existente os elementos introduzidos pelo modelo Motherboard.

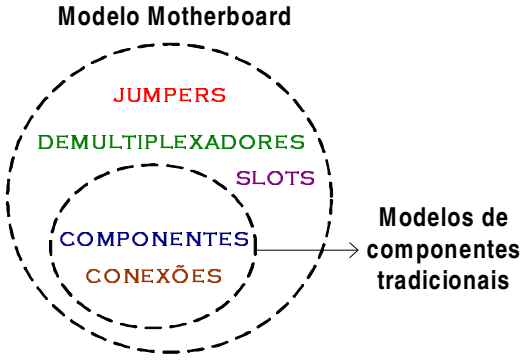

Figura 5.1: Elementos do Modelo Motherboard

5.2 Visão Geral da Plataforma MotherFrame

Considerando as decisões discutidas na seção anterior, a plataforma MotherFrame foi projetada como uma camada de abstração em cima de um ambiente de execução de componentes existente – o OpenCOM [19]. Nesse projeto, enquanto que o modelo OpenCOM provê os elementos arquiteturais básicos como componentes e conexões, a plataforma MotherFrame adiciona os elementos o apoio aos conceitos de jumpers, slots e demultiplexadores.

Em particular, a adoção do OpenCOM como modelo de componentes base foi influenciada pelas seguintes características que ele agrega: foi projetado especialmente para o desenvolvimento de middlewares abertos [9, 19, 38]; possui apoio a reflexão computacional permitindo reconfiguração em tempo de execução; possui um ambiente de execução leve e adequado a dispositivos com recursos limitados.

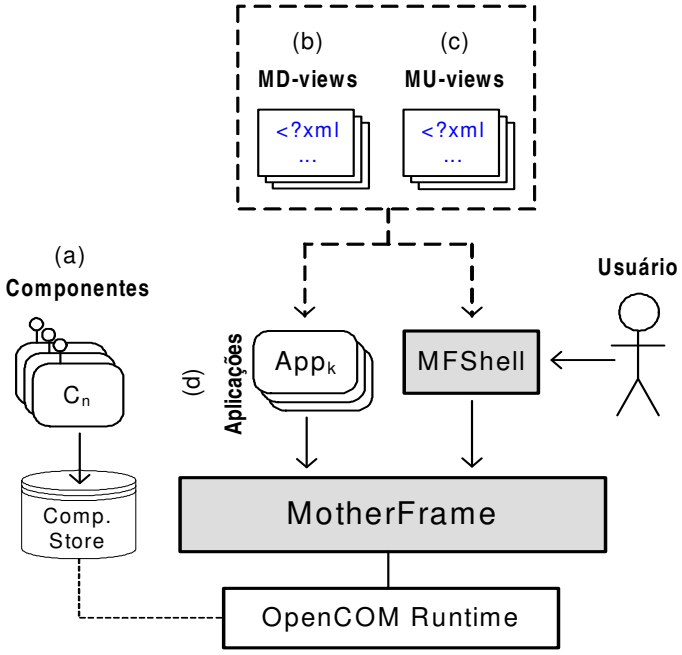

Figura 5.2: Elementos da Plataforma Motherboard

Uma visão geral das interações que envolvem a plataforma MotherFrame é apresentada na Figura 5.2. A plataforma MotherFrame faz um mapeamento de especificações MD-Views (Figura 5.2(b)) e MU-Views (Figura 5.2(c)) de middlewares em código integrado no OpenCOM. A submissão de MD-Views e MU-Views pode ser feita de duas maneiras: (i) através de aplicações que implementam a API da MotherFrame; (ii) através de comandos tipo texto informados pelo usuário para o interpretador de comandos *MFShell*. Cada componente referenciado nessas especificações devem estar previamente implemen-

tados e armazenados no repositório de componentes do OpenCOM. A implementação de componentes deve seguir o modelo de componentes OpenCOM (Figura 5.2(a)). O Apêndice A apresenta uma visão geral do OpenCOM descrevendo como componentes do tipo OpenCOM podem ser implementados.

Os serviços da plataforma MotherFrame são providos através de uma interface chamada `IMotherFrame`. Essa interface provê o conjunto das seguintes operações:

```
void deploy(String xmlFile);
void config(String xmlFile);

void beginConfigTrans(String confId, String archId);
void endConfigTrans();
void abortConfigTrans();

void createComponent(String compId, String source);
void setJumperStatus(String jumperId, String status);
void mapCompToSlot(String compId, String slotId);
void setMux(String muxId, String output);
```

O acesso aos serviços da plataforma MotherFrame pode ser feito de duas maneiras. A primeira é implementar uma aplicação que utiliza as operações da interface `IMotherFrame` (Figura 5.2(d)). A segunda é utilizar o interpretador de comandos `MFShell`. Essas duas formas de acessar a plataforma serão discutidas nas seções a seguir.

5.2.1 Usando a interface `IMotherFrame`

Usando a interface IMotherFrame, uma aplicação pode acessar os dois serviços básicos da plataforma MotherFrame – submissão de MD-Views e submissão de MU-Views

Submissão de MD-Views

Para a submissão de especificações XML do tipo MD-View, uma aplicação pode usar a operação `deploy` da interface `IMotherFrame`. Essa operação recebe como parâmetro o nome do arquivo XML que contém a especificação MD-View desejada.

Quando a operação `deploy` é chamada, a especificação XML fornecida como parâmetro é processada para extrair as informações da arquitetura desejada. A partir dessas informações, a plataforma MotherFrame cria uma representação interna da respectiva MD-View e instancia uma estrutura de mais baixo nível baseada em componentes e conexões no ambiente de execução do modelo OpenCOM. Na prática, a plataforma realiza o mapeamento de MD-Views para o ambiente OpenCOM.

Submissão de MU-Views

Para a submissão de especificações XML do tipo MU-View, uma aplicação pode usar a operação `config` da interface `IMotherFrame`. Essa operação recebe como parâmetro o nome do arquivo XML que contém a especificação MU-View desejada.

Quando a operação `config` é chamada, a especificação XML é processada para extrair as informações da configuração. Uma das informações que é extraída é o identificador da MD-View à qual a MU-View deverá ser aplicada. A partir dessas informações, a plataforma MotherFrame cria uma representação interna da respectiva MU-View associada à MD-View correspondente. Além dessa representação, a plataforma mapeia a MU-View para o nível mais baixo – o do ambiente de execução do modelo OpenCOM.

Além da forma de submissão de MU-Views baseada em especificações XML, a plataforma MotherFrame provê ainda uma forma dinâmica de submissão de uma MU-View. Ao invés de especificar e de submeter um arquivo XML, uma MU-View pode ser definida dinamicamente através da chamada direta de um conjunto de operações de configuração da interface `IMotherFrame`. Basicamente, essas operações acessam e manipulam os elementos abertos previamente definidos em uma MD-View. As operações de configuração são:

`createComponent` – instancia um componente a ser usado na configuração MU-View (parâmetros: o identificador desejado para a instância do componente e o nome canônico da classe que o implementa).

`setJumperStatus` – determina se o jumper deverá estar ativado ou desativado (parâmetros: o identificador do jumper e o estado desejado "ON" ou "OFF").

`mapCompToSlot` – conecta um componente em um determinado slot (parâmetros: o identificador do componente e o identificador do slot).

`setMux` – seleciona uma conexão de saída de um demultiplexador (parâmetros: identificador do demultiplexador e identificador da conexão de saída).

Para definir uma MU-View dinamicamente, essas operações de configuração devem ser chamadas como uma transação. Para isso elas devem ser sempre delimitadas pelas operações transacionais `beginConfigTrans` e `endConfigTrans` da interface IMotherFrame. Como parâmetro da operação `beginConfigTrans` deve ser especificado o identificador da configuração MU-View e o identificador da arquitetura base MD-View (para a qual a configuração MU-View será aplicada).

5.2.2 Usando o Interpretador de Comandos MFShell

Uma alternativa ao uso dos serviços da plataforma MotherFrame é através do interpretador de comandos MFShell. Esse interpretador recebe comandos do tipo texto a partir de

uma linha de comando e os converte a chamadas à interface IMotherFrame. Usando o MFShell, usuários e desenvolvedores de middleware podem acessar os serviços da plataforma sem a necessidade de implementar e compilar uma aplicação que utilize diretamente a interface IMotherFrame.

Para utilizar o MFShell, após o interpretador de comandos `mf>` deve-se digitar a operação desejada seguida pelos parâmetros requeridos pela operação (se houver). Cada uma das operações fornecidas pelo MFShell é descrita abaixo.

`mf> deploy` *`nome_do_arquivo`* – Usado para a submissão de especificações do tipo MD-View no formato XML. Como parâmetro deve ser fornecido o nome do arquivo que contém a especificação XML da MD-View.

`mf> config` *`nome_do_arquivo`* – Usado para a submissão de especificações do tipo MU-View no formato XML. Como parâmetro deve ser fornecido o nome do arquivo que contém a especificação XML da MU-View.

`mf> begin` *`id_da_muview id_da_mdview`* – Inicia uma transação de configuração (processo dinâmico de definição de MU-Views). Como parâmetros devem ser fornecidos o identificador da MU-View que será definida e o identificador da MD-View para a qual a MU-View será aplicada.

`mf> end` – Submete a transação de configuração corrente que foi iniciada pela operação `begin`.

`mf> abort` – Aborta a transação de configuração corrente iniciada anteriormente pela operação `begin`.

`mf> comp` *`id_do_componente classe`* – Instancia um componente dentro de uma transação de configuração. Como parâmetros devem ser fornecidos um identificador para a instância do componente a ser criada e o nome canônico da classe que o implementa.

`mf> comp2slot` *`id_do_componente id_do_slot`* – Conecta um componente em um slot. Como parâmetros devem ser fornecidos o identificador da instância do componente a ser conectada e o identificador do slot.

`mf> jstate` *`id_do_jumper estado_do_jumper`* – Define o estado de um jumper (ativado ou desativado). Como parâmetros devem ser fornecidos o identificador do jumper e o estado desejado (*on* ou *off*).

`mf> setmux` *`id_do_demultiplexador conexão_de_saída`* – Seleciona uma determinada conexão de saída de um demultiplexador. Como parâmetros devem ser fornecidos o identificador do demultiplexador e o identificador da conexão de saída a ser selecionada.

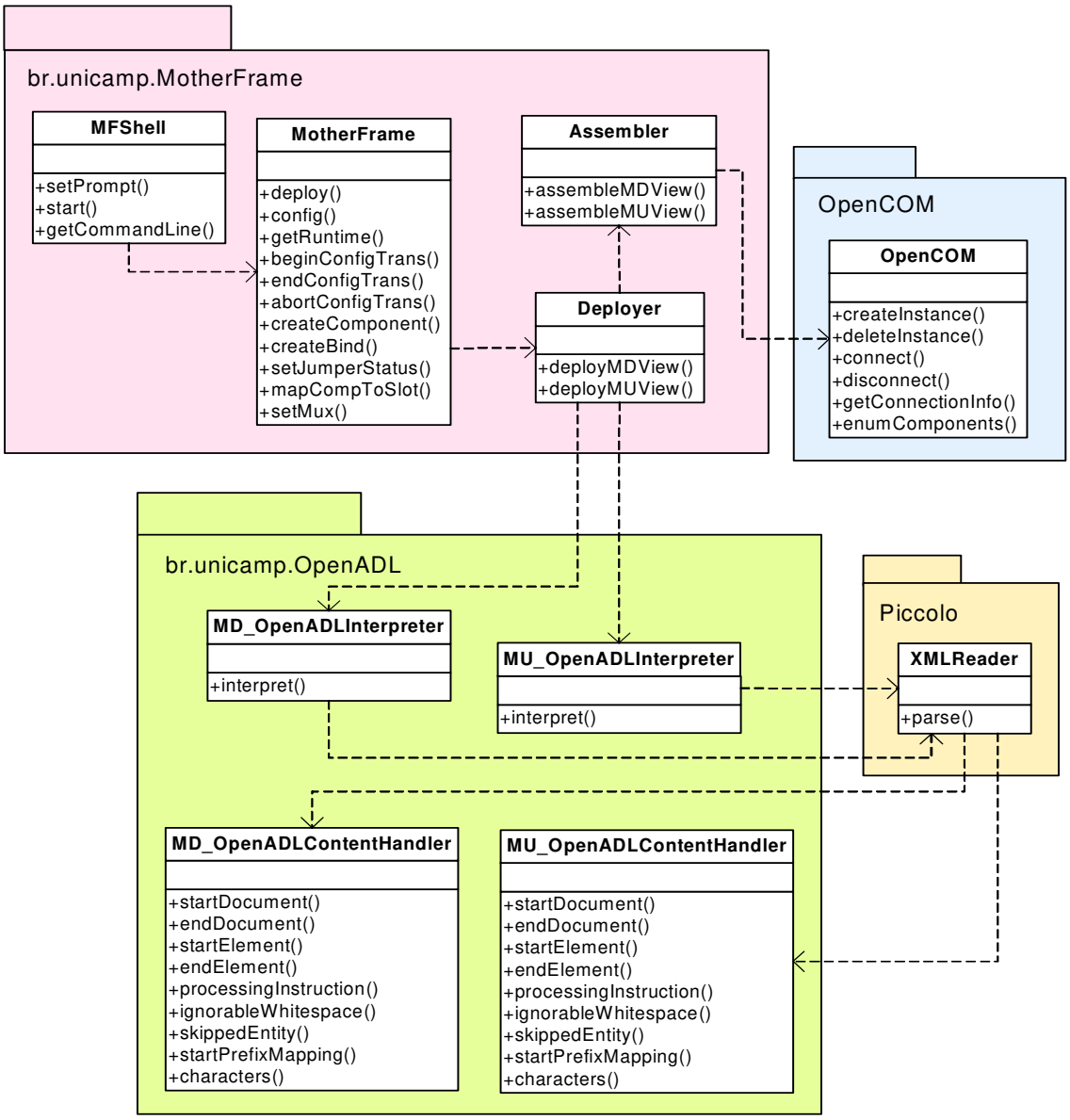

Figura 5.3: Diagrama de Classes da Plataforma MotherFrame

5.3 Implementação da Plataforma MotherFrame

Nesta seção serão apresentados alguns aspectos relacionados à implementação da plataforma MotherFrame. Dentre esses serão apresentados diagramas de classe e seqüência UML da plataforma, o mapeamento do modelo Motherboard para o modelo OpenCOM e detalhes acerca das ferramentas utilizadas na implementação.

5.3.1 Diagrama de Classes

A Figura 5.3 apresenta um diagrama de classes UML simplificado da plataforma MotherFrame. Nele são apresentadas as principais classes de cada um dos quatro pacotes utilizados na implementação da plataforma:

(1) **br.unicamp.MotherFrame** – Pacote que inclui as classes que implementam os principais processos da plataforma MotherFrame.

- `MFShell` – Implementa um interpretador de comandos tipo texto (*shell*) que pode ser usado pelo usuário para interagir com a plataforma MotherFrame.
- `MotherFrame` – Implementa o controlador principal da plataforma MotherFrame.
- `Deployer` – Controla o processo de submissão de MD-Views e MU-Views.
- `Assembler` – Recebe informações de especificações MD-Views e MU-Views e as instancia no ambiente de execução do OpenCOM.

(2) **br.unicamp.OpenADL** – Pacote que inclui as classes responsáveis por extrair as informações das representações XML de MD-Views e MU-Views.

- `MD_OpenADLInterpreter` – Controla o processo de extração de informações de especificações XML do tipo MD-View.
- `MD_OpenADLContentHandler` – Extrai informações de cada um dos elementos que compõem uma MD-View (ex: componentes, conexões, jumpers, etc.).
- `MU_OpenADLInterpreter` – Controla o processo de extração de informações de especificações XML do tipo MU-View.
- `MU_OpenADLContentHandler` – Extrai informações de cada um dos elementos que compõem uma MU-View.

(3) **OpenCOM** – Pacote que implementa o ambiente de execução do modelo de componentes OpenCOM.

- `OpenCOM` – Classe principal do ambiente OpenCOM. Processa requisições como, instanciação de componentes e criação de conexões entre componentes.

(4) **Piccolo** – Pacote que implementa um analisador (*parser*) para arquivos XML.

- `XMLReader` – Controla o processo de leitura e análise de arquivos que contém especificações XML.

Dentre esses pacotes, somente os dois primeiros foram implementados neste trabalho (`br.unicamp.MotherFrame` e `br.unicamp.OpenADL`). O pacote OpenCOM foi implementado pelo grupo de sistemas distribuídos da Universidade de Lancaster (Inglaterra) e o pacote Piccolo [43] foi implementado pela companhia BlueCast.

5.3.2 Diagramas de Seqüência

Nesta seção, serão apresentados três diagramas de seqüência UML que representam as principais atividades da plataforma MotherFrame. Esses diagramas mostram a seqüência de interação das classes mostradas na Figura 5.3.

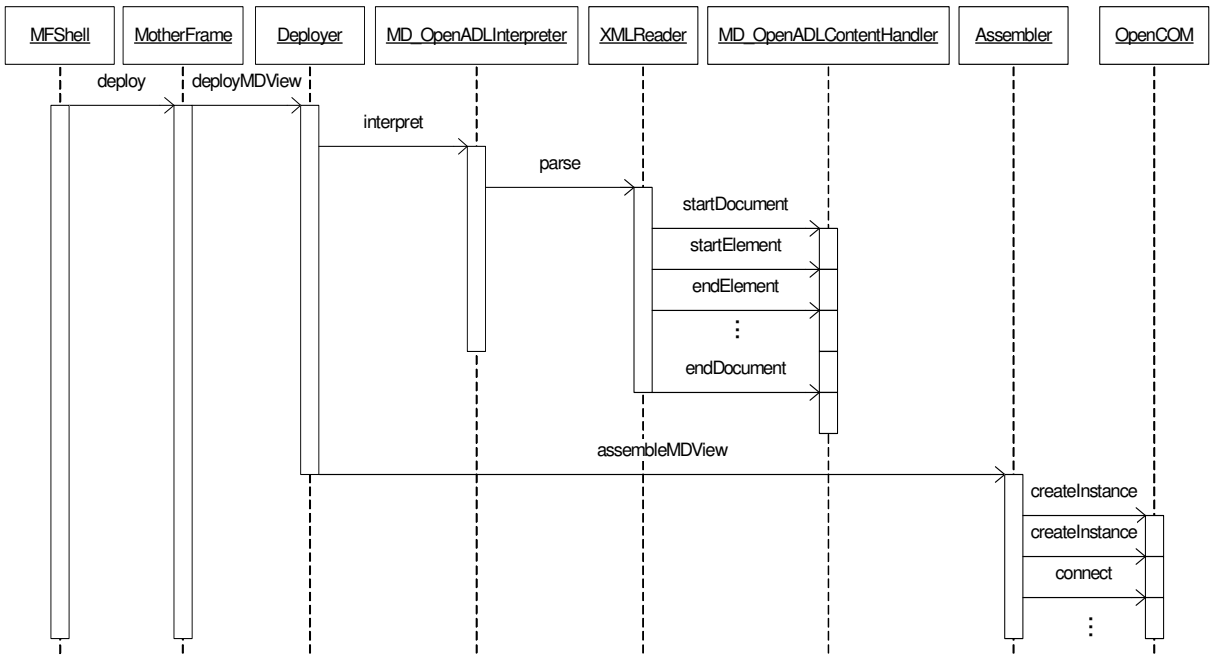

Figura 5.4: Diagrama de Seqüência – Submissão de MD-Views (XML)

O primeiro diagrama (Figuras 5.4) mostra o processo de submissão e processamento de MD-Views. Nele, especificações XML de MD-Views são submetidas à classe MotherFrame através da operação `deploy`.

O segundo e o terceiro diagrama (Figuras 5.5 e 5.6, respectivamente) mostram os dois processos disponíveis para a submissão e o processamento de MU-Views. O primeiro

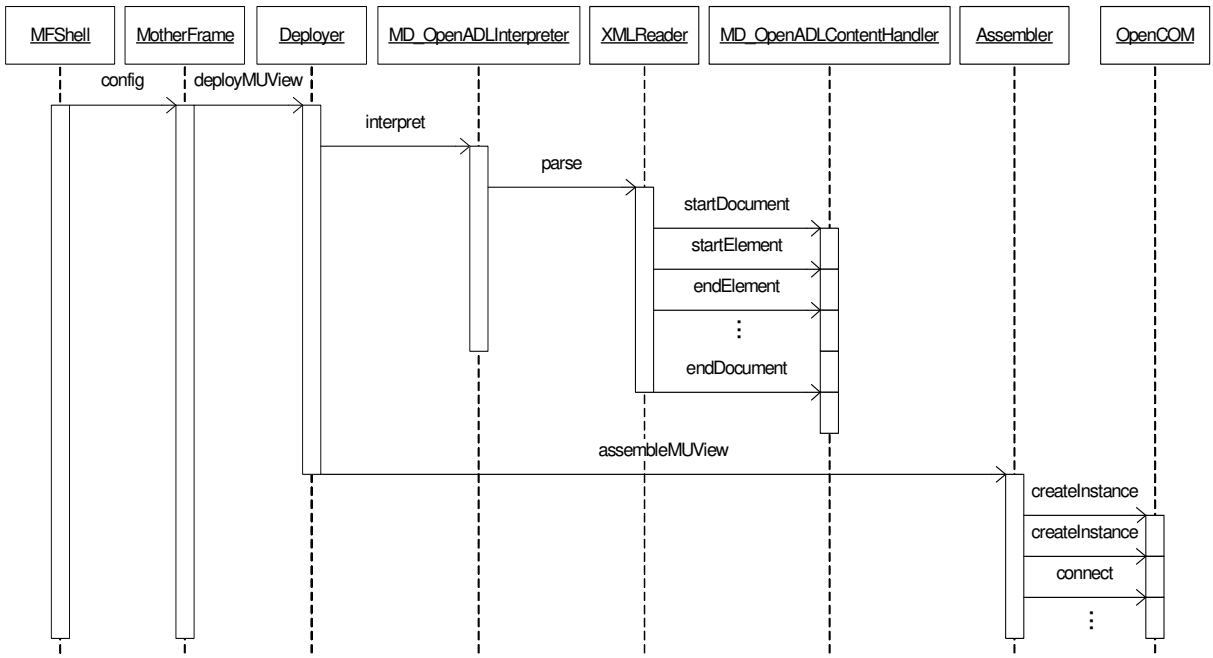

Figura 5.5: Diagrama de Seqüência – Submissão de MU-Views (XML)

mostra a submissão de MD-Views como especificações XML, ao passo que o segundo mostra a submissão como chamadas das operações transacionais providas pela classe MotherFrame (por meio da interface `IMotherFrame`).

5.3.3 Mapeamento *Modelo Motherboard* para *Modelo OpenCOM*

Como já foi discutido anteriormente, a plataforma MotherFrame foi definida como uma camada de abstração em cima do modelo de componentes OpenCOM. É nessa camada de abstração que são representados elementos do modelo Motherboard: componentes, conexões, jumpers, demultiplexadores e slots. É através desses elementos que arquiteturas de middleware são representadas na plataforma MotherFrame.

Além de ser capaz de manter representações de arquiteturas de middleware, a plataforma MotherFrame realiza um mapeamento dessas representações para código integrado junto ao ambiente OpenCOM. É no ambiente OpenCOM que a arquitetura desejada é montada e os serviços do middleware são disponibilizados para o uso. Nesse contexto, enquanto todo o código que implementa o middleware é mantido no ambiente OpenCOM, sua representação no modelo Motherboard é mantida na plataforma MotherFrame. O modelo Motherboard é a forma que usuários e desenvolvedores enxergam e manipulam o middleware. O modelo OpenCOM é a forma como o middleware é realmente estruturado.

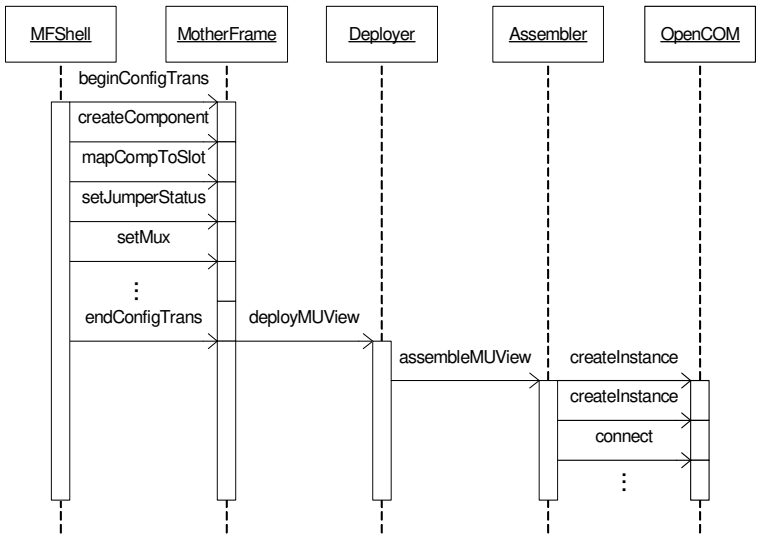

Figura 5.6: Diagrama de Seqüência – Submissão de MU-Views (Operações Transacionais)

Nesse contexto, a função da plataforma MotherFrame é justamente a de fazer um mapeamento entre as representações do modelo Motherboard (MD-Views e MU-Views) e o ambiente OpenCOM. Nesse mapeamento, mudanças ocorridas na representação do middleware são refletidas para o ambiente OpenCOM (onde o middleware se encontra implementado).

Na prática, realizar esse mapeamento consiste em traduzir os cinco elementos disponíveis no modelo Motherboard (componentes, conexões, jumpers, demultiplexadores e slots) para os dois elementos do modelo OpenCOM (componentes e conexões). A forma como essa tradução é feita é ilustrada na Figura 5.7.

Na realização desse mapeamento, a plataforma MotherFrame executa um conjunto de operações disponibilizadas pelo ambiente OpenCOM que podem ser do seguinte tipo: *(i)* instanciar e ativar um componente; *(ii)* excluir a instância de um componente; *(iii)* criar uma conexão entre dois componentes; *(iv)* excluir a conexão entre dois componentes; *(v)* localizar a referência de um componente.

Observando o item 4 da Figura 5.7, será exemplificado como a MotherFrame usa essas operações para fazer reconfigurações nas arquiteturas OpenCOM de acordo com a mudanças no modelo Motherboard. Nesse exemplo, será considerado uma mudança na representação do item 4 de *slot vazio* para *slot com componente*. Segue a seqüência de operações no OpenCOM necessárias para se realizar essa mudança:

1. Localizar a referência do componente C_f.

2. Localizar a referência do componente C_g.

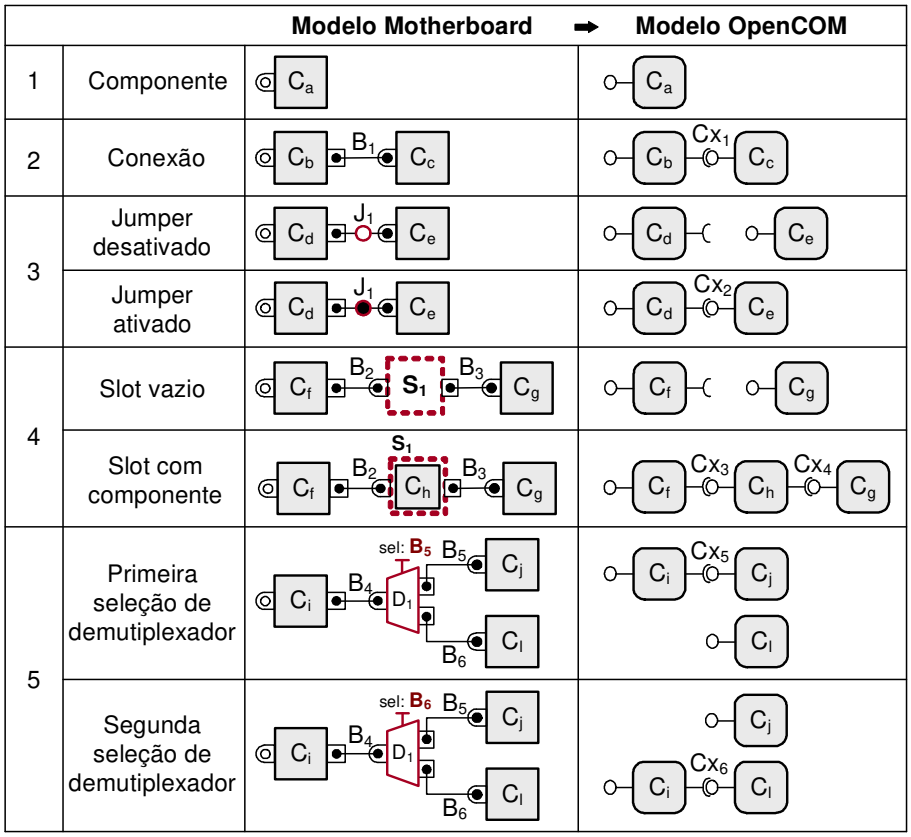

Figura 5.7: Mapeamento

3. Criar uma instância do componente C_h e ativá-la.

4. Criar uma conexão entre uma interface requerida do componente C_f e uma interface provida do componente C_h.

5. Criar uma conexão entre uma interface requerida do componente C_h e uma interface provida do componente C_g.

5.3.4 Linguagem e Ferramentas Utilizadas

Para a implementação da plataforma foi usada a linguagem Java na plataforma J2SE 1.5. Os motivos que levaram à adoção dessa linguagem foram: *(i)* ser amplamente conhecida tanto em comunidades científicas quanto em comerciais; *(ii)* possuir um grande conjunto de bibliotecas disponíveis que facilitam e agilizam o processo de implementação – o que vai ao encontro das idéias defendidas pelo modelo de prototipagem rápida adotado nesse trabalho; *(iii)* ser a mesma linguagem com que foram desenvolvidas algumas versões do OpenCOM.

O ambiente OpenCOM foi utilizado na versão 1.3.5 tipo Java disponibilizada online. Algumas das razões que levaram à adoção desse ambiente foram: *(i)* ter sido proposto como uma solução na implementação de middleware aberto; *(ii)* ter sido utilizado na implementação de vários trabalhos relevantes relacionados com middleware aberto; *(iii)* ser leve e possuir interfaces reflexivas que possibilitam a reconfiguração dinâmica de componentes e conexões.

Apesar de o ambiente OpenCOM ser utilizado como a camada de baixo nível sobre o qual está a plataforma MotherFrame, esta não foi implementada como componentes OpenCOM e sim como classes puramente Java. Isso garante uma independência da plataforma MotherBoard do ambiente OpenCOM facilitando a portabilidade da plataforma para ambientes de outros modelos de componentes como o Fractal [12].

Na implementação dos processos da plataforma MotherFrame, foi utilizado um analisador (*parser*) de arquivos XML chamado Piccolo. A versão utilizada foi a 1.04 tipo Java. Em particular, o pacote Piccolo foi escolhido como o analisador de arquivos XML deste trabalho pelas vantagens de ser pequeno e rápido quando comparado a outros analisadores de XML. Tais vantagens vêm ao encontro de alguns dos requisitos desejados para a plataforma MotherFrame como o de não consumir muitos recursos (memória, disco e processamento).

Capítulo 6

Serviços de Transações Abertos

Nesta seção serão apresentados dois estudos que mostram como serviços de transações podem ser representados no modelo Motherboard. A seção 6.1 apresenta uma plataforma de transações adaptável para o ambiente de computação móvel e a seção 6.2 discute um serviço de transação com adaptação dinâmica de protocolos de efetivação.

6.1 Primeiro Estudo – A ATP

O primeiro estudo de caso é baseado em uma plataforma de transações adaptável para ambientes de computação móvel (ATP[1]) [85]. Essa plataforma foi remodelada no modelo Motherboard adicionando-se a ela a definição personalizada de critérios de corretude.

Primeiramente serão descritos alguns fundamentos necessários para a compreensão do estudo de caso. Em seguida, será descrita a plataforma ATP. Por fim, a plataforma ATP será representada no modelo motherboard.

6.1.1 Critérios de Corretude

Esta seção apresenta o primeiro fundamento necessário para uma mais clara compreensão do primeiro estudo de caso a ser apresentado: a noção de *critérios de corretude*. Além disso, esta seção introduz uma solução apresentada em [84] que permite a definição personalizada de critérios de corretude.

Motivação para o Surgimento de Critérios de Corretude

Tradicionalmente, uma transação só seria considerada correta se ela honrasse com rigor cada uma das propriedades ACID. Apesar de essa definição de *transação correta* ainda

[1] ATP é um acrônimo para Adaptable Transactional Platform

ser adotada por diversos tipos de aplicação, ambientes emergentes (como a computação móvel e ambientes de tempo real) têm requerido uma flexibilização dessas propriedades – tanto para lidar com aspectos específicos do ambiente em questão como para atender requisitos específicos das aplicações.

Para exemplificar essa demanda por flexibilização, pode-se considerar a propriedade tradicional de isolamento. Para garanti-la foi proposto o conceito de serialização [7]. Apesar de ser uma técnica simples e eficaz no que diz respeito à garantia de total isolamento entre transações concorrentes, a serialização impõe impactos consideráveis no desempenho das transações concorrentes [62]. Esses impactos passaram a contrariar os requisitos de desempenho de alguns domínios de aplicação. Para melhorar esse desempenho, foram propostos três níveis de isolamento mais relaxados que a serialização: o `Repeatable Read`, o `Read Committed` e o `Read Uncommitted`. Quanto mais relaxado o nível de isolamento, um maior desempenho se pode obter.

É dessa nova demanda de propriedades ACID mais flexíveis que se pode extrair o conceito de critérios de corretude. Uma transação correta não é mais necessariamente aquela que honra as propriedades ACID tradicionais, mas aquela que honra um conjunto de critérios de corretude previamente especificados. Nesse contexto, cada conjunto de critérios de corretude passa a ser especificado de acordo com os requisitos de cada ambiente ou de cada aplicação envolvida. Por exemplo, considerando-se o contexto de níveis de isolamento, uma transação que aceita executar com o critério de corretude *Read Uncommitted* pode ser considerada correta mesmo que ela faça acesso a dados inconsistentes pelo critério de serialização.

Soluções em Critérios de Corretude

Como resposta à demanda de maior flexibilidade de corretude transacional, foram propostos inúmeros serviços de transação. Somente no ambiente de computação móvel, já se encontram vários exemplos: os serviços de transação PRO-MOTION [104], AMT [94] e PTM [63] relaxam a propriedade de atomicidade através do conceito de transações aninhadas (partes de uma transação pode ser abortada sem que se aborte a transação como um todo); Coda-IOT [92], HiCoMo [59], AMT e MobileTrans [90] relaxam a propriedade de isolamento permitindo que uma transação em execução libere resultados parciais para outras transações; HiCoMo e MobileTrans flexibilizam a durabilidade tolerando o descarte de algumas atualizações realizadas por transações que já foram efetivadas.

Limitações das Soluções Existentes

Apesar de existirem vários serviços de transação que relaxam as propriedades ACID tradicionais, uma limitação comumente presente nesses modelos é que cada serviço provê so-

mente um conjunto pré-determinado de critérios de corretude. Como esse conjunto é fixo, ele normalmente só atende aos requisitos de um domínio ou de um grupo restrito de aplicações. Caso uma aplicação requeira um critério de corretude um pouco diferente do que foi definido, esses modelos não são aptos a provê-lo.

A falta de flexibilidade desses serviços normalmente está relacionada com sua implementação. O conjunto de critérios de corretude provido por um serviço de transação normalmente é parte integrante do mesmo código que implementa os controladores do serviço. Por exemplo, o apoio a níveis de isolamento normalmente é implementado como parte integrante do código do controlador de concorrência do serviço de transação em questão. Com isso, a alteração dos critérios de corretude só pode ser feita através da manipulação do código do serviço de transações.

Critérios de Corretude Personalizados - Uma Nova Proposta

Considerando essa limitação, em [84] foi proposta uma arquitetura de um serviço de transação que permite a inclusão de critérios de corretude personalizados baseados nas propriedades ACID. A estratégia proposta para se permitir essa personalização foi a de separar o código que implementa o controlador de concorrência do código que implementa cada critério de corretude. Assim, novos critérios poderiam ser implementados independentemente do restante do serviço de transação.

A arquitetura que representa essa solução é apresentada na Figura 6.1. A notação adotada utiliza elementos do modelo de componentes OpenCOM: componentes e conexões. Pode-se observar que cada critério de corretude está implementado como um componente individual (*READ_UNCOMMITTED*, *READ_COMMITTED*, *REPEATABLE_READ* e *SERIALIZABLE*). Esses critérios são implementados idependentemente do controlador de concorrência (*ConcurrController*). Além desses componentes, essa arquitetura é composta por um gerenciador de transações – elemento central que provê operações transacionais a aplicações e coordena as atividades de execução das transações – e um gerenciador de recuperação – que gerencia as atividades de recuperação em casos de falhas.

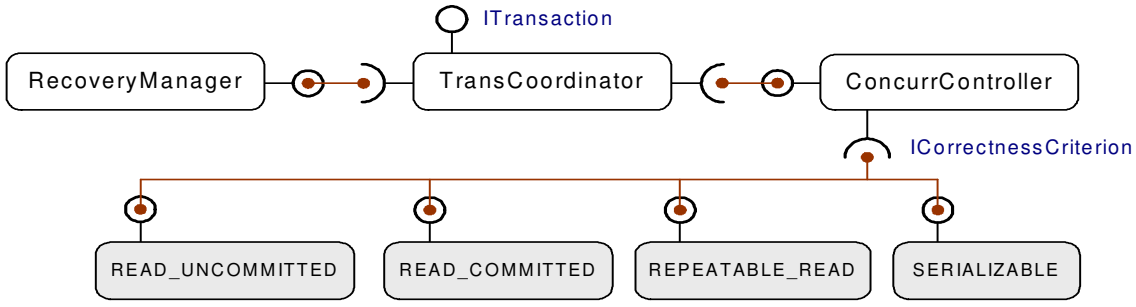

Figura 6.1: Definição Personalizada de Critérios de Corretude

Para se acrescentar um critério de corretude personalizado aos disponibilizados nessa arquitetura, basta implementar um novo componente. Na prática, isso implica em implementar as operações da interface *ICorrectness*. Essa é uma interface requerida do controlador de concorrência *ConcurrController* com as seguintes operações:

```
interface ICorrectness{
  void begin_transaction(ILockManager lm);//Início de transação
  void begin_item(ILockManager lm);//Início de operação simples
  void end_item(ILockManager lm);//Fim de operação simples
  void begin_predicate(ILockManager lm);//Início de oper. composta
  void end_predicate(ILockManager lm);//Fim de operação composta
  void end_local(ILockManager lm);//Fim de transação local
  void end_transaction(ILockManager lm);//Fim de transação global
}
```

A interface *ICorrectness* indica eventos que podem ocorrer no decorrer da execução de uma transação. Considerando que o controlador de concorrência é baseado em trancas[2], implementar essa interface significa indicar em quais desses eventos se devem adquirir ou liberar as trancas necessárias de modo a obter o critério de corretude desejado[3]. Para representar a aquisição ou liberação de trancas, as operações da interface *ILockManager* devem ser utilizadas. Essa interface é fornecida como parâmetro das operações da interface *ICorrectness*. As operações da interface *ILockManager* são representadas pelos seguintes termos:

- LOCK_I – Adquire tranca de leitura ou de escrita sobre um item de dado.
- LOCK_P – Adquire tranca de leitura ou de escrita sobre um grupo de itens de dado.
- unLOCK_I – Libera tranca de leitura ou de escrita sobre um item de dado.
- unLOCK_P – Libera tranca de leitura ou de escrita sobre um grupo de itens de dado.

Para ilustrar o uso dessas operações na definição de critérios de corretude, a Tabela 6.1 mostra como podem ser implementados os níveis de isolamento definidos pelo padrão ANSI/ISO SQL-92.

De modo similar à definição dos critérios da Tabela 6.1, critérios de corretude personalizados podem ser implementados. Em [84] são mostrados outros exemplos de definição de critérios de corretude personalizados baseados na solução apresentada.

[2]Locks, em inglês.

[3]Em controles de concorrência baseados em trancas, critérios de corretude relacionados à propriedade de isolamento são especificados através da indicação dos pontos de aquisição e de liberação de trancas.

	READ_UNCOMMITTED		READ_COMMITTED		REPEATABLE_READ		SERIALIZABLE	
	Read	Write	Read	Write	Read	Write	Read	Write
begin_item		$LOCK_I$	$LOCK_I$	$LOCK_I$	$LOCK_I$	$LOCK_I$	$LOCK_I$	$LOCK_I$
end_item			$unLOCK_I$					
begin_predicate		$LOCK_P$	$LOCK_P$	$LOCK_P$	$LOCK_P$	$LOCK_P$	$LOCK_P$	$LOCK_P$
end_predicate			$unLOCK_P$		$unLOCK_P$			
end_transaction		$unLOCK_I$ $unLOCK_P$		$unLOCK_I$ $unLOCK_P$	$unLOCK_I$	$unLOCK_I$ $unLOCK_P$	$unLOCK_I$ $unLOCK_P$	$unLOCK_I$ $unLOCK_P$

Tabela 6.1: Implementação dos níveis de isolamento

6.1.2 Plataforma ATP

Além da noção de critério de corretude discutida na seção anterior, esta seção apresenta mais um fundamento necessário para a compreensão do primeiro estudo de caso: a descrição da plataforma ATP. Essa plataforma provê um serviço de transações adaptável para ambientes de computação móvel [85].

Motivações para o Desenvolvimento da ATP

O compartilhamento de dados entre computadores que fazem parte de ambientes móveis é uma necessidade clara. Porém, apesar dos recentes avanços nas tecnologias envolvidas, uma série de obstáculos ainda continua a dificultar esse compartilhamento em maior ou menor grau. Entre esses obstáculos, estão: uma maior susceptibilidade a desconexões, variações na largura de banda e no custo da comunicação e escassez de recursos nos dispositivos de computação portáteis (como baterias, espaço em disco e poder de processamento). Características como essas tornaram o ambiente móvel muito mais dinâmico do que os ambientes distribuídos tradicionais.

Para reduzir os impactos causados pelo dinamismo de ambientes móveis no que diz respeito ao acesso a dados, o projeto de serviços de transação passou a considerar um novo requisito: a capacidade de se adaptar às mudanças ocorridas no ambiente. Para satisfazer esse novo requisito, foi proposta a plataforma ATP.

Visão Geral da Plataforma ATP

A ATP é uma plataforma de gerenciamento de transações que provê um serviço de transações capaz de se adaptar a aspectos dinâmicos de ambientes de computação móvel. Essa adaptação é dada através do conjunto de mecanismos de adaptação provido pelo serviço de transações.

O serviço de transações proposto utiliza a abordagem de adaptação colaborativa, dividindo as responsabilidades do processo de adaptação entre a ATP – que provê o serviço

de transações – e as aplicações – que utilizam o serviço provido.

- **ATP**. É responsável por *(i)* prover operações transacionais para aplicações, *(ii)* monitorar aspectos dinâmicos dos ambientes de execução como largura de banda, custo da comunicação, espaço em disco e energia, *(iii)* notificar as aplicações acerca de mudanças significativas nesses aspectos dinâmicos e *(iv)* prover mecanismos capazes de adaptar as transações às variações do ambiente de execução.

- **Aplicações**. São responsáveis por *(i)* requisitar o monitoramento de recursos de acordo com os interesses envolvidos e *(ii)* indicar a política de adaptação que deve ser adotada pela ATP como reação às mudanças do ambiente.

Nesse modelo, enquanto que a ATP provê mecanismos de adaptação, as aplicações são capazes de definir como a adaptação deve ser feita de forma a melhor satisfazer seus interesses. Os mecanismos de adaptação providos pela ATP são: a mudança de níveis de isolamento e de modos de operação. Esses dois mecanismos são descritos a seguir.

Níveis de Isolamento

O isolamento é uma propriedade transacional que visa conter interferências entre transações concorrentes. Uma forma tradicional de evitar essas interferências é através da técnica de serialização. Nela, transações podem executar concorrentemente com a garantia de total isolamento entre elas.

Apesar da sua eficácia na manutenção da propriedade de isolamento, a serialização é uma técnica considerada de baixa eficiência, pois impõe uma série de restrições ao acesso a dados que causam impactos no desempenho das transações em execução. Foi para aliviar esses impactos que o conceito de níveis de isolamento foi proposto. Os níveis de isolamento são: 1-Uncommitted Read; 2-Committed Read; 3-Repeatable Read; 4-Serializable. Nessa escala, quanto menor o nível de isolamento, menor a garantia de acesso consistente a dados e maior o desempenho obtido.

Considerando a possibilidade de ganhos em desempenho obtidos ao se usar níveis de isolamento mais baixos, a ATP provê o apoio a níveis de isolamento como um dos seus mecanismos de adaptação. Usando níveis mais baixos, as aplicações podem aliviar as perdas em desempenho causadas pelo dinamismo de ambientes de computação móvel.

Modos de Operação

O modo de operação é outro mecanismo de adaptação provido pela ATP para viabilizar a execução de transações em ambientes móveis sujeitos a problemas como variações na

largura de banda, variações de latência e desconexões inesperadas. Para isso, são utilizadas basicamente técnicas de caching de dados e de operação desconectada.

Os modos de operação providos são três: remoto (R), local (L) e local-remoto (LR). Basicamente eles definem se os objetos (dados) participantes de uma transação serão acessados remotamente ou localmente (a partir de uma cópia mantida em cache). Segue uma breve descrição de cada modo de operação.

R. Neste modo, as transações executam suas operações diretamente sobre as matrizes dos objetos localizados remotamente em uma rede fixa. O acesso a esses dados fica sujeito a um controle de concorrência pessimista baseado em trancas. Para realizar esse acesso é necessária a utilização do canal de comunicação sem fio.

L. No modo local, é realizada uma cópia dos dados remotos para a unidade móvel. Nesse modo, transações podem executar sobre as cópias locais mesmo na ocorrência de desconexões. Essa execução é feita de forma otimista, ou seja, não é tomada nenhuma medida prévia para se garantir a consistência entre as cópias locais e as suas correspondentes remotas. Essa consistência só é verificada no final da execução de cada transação através da execução de um protocolo de validação.

LR. Este modo de operação é semelhante ao modo local no que diz respeito à manutenção da cópia dos dados remotos na unidade móvel. Porém, ao invés de trabalhar de maneira otimista como o modo local, o modo local-remoto toma medidas prévias para se manter a garantia da consistência entre os dados remotos e suas cópias locais. Essas medidas consistem em colocar trancas sobre os dados remotos que foram copiados para a unidade móvel.

Em suma, o modo de operação R oferece a forma de acesso a dados tradicional – provendo acesso remoto aos dados localizados na rede fixa. Esse modo tradicional pode sofrer as conseqüências das desconexões inesperadas e das variações na largura de banda. Como opção, são providos os modos L e LR que realizam caching de dados para que se tornem disponíveis durante os períodos de desconexão.

Limitações da Plataforma ATP

Como foi apresentado, a ATP é uma plataforma que provê um serviço de transações capaz de se adaptar ao dinamismo de ambientes de computação móvel. Para isso, esse serviço fornece basicamente dois mecanismos de adaptação: níveis de isolamento – provê diferentes graus de desempenho na execução de transações; e modos de operação – permite execução de transações em diferentes estados da comunicação remota (fortemente conectado, fracamente conectado ou desconectado).

Apesar dessa flexibilidade, a ATP possui a seguinte limitação: os mecanismos de adaptação propostos não são extensíveis. Isso implica que a ATP só é apta a satisfazer os requisitos de um conjunto restrito de aplicações – aquelas que possuem requisitos que foram previstos pelos mecanismos de adaptação propostos. Aplicações que possuem requisitos diferentes dos que foram previstos pela ATP não podem se beneficiar da mesma. Considerando que, no geral, as aplicações que fazem uso de serviços de transação possuem requisitos que tendem a ficar cada vez mais variados, a capacidade de ser extensível é uma característica que tem sido considerada indispensável para a nova geração de serviços de transação.

6.1.3 Reestruturando a ATP no Modelo Motherboard

Considerando as limitações da ATP no quesito extensibilidade, esta seção apresenta como o modelo Motherboard pode ser usado para corrigir essas limitações. Para isso, essa plataforma foi remodelada no modelo Motherboard. Essa remodelagem, que será chamada de ATPm, trouxe para a ATP as seguintes novas características:

- O apoio à personalização de critérios de corretude. Critérios de corretude que satisfazem os requisitos específicos de aplicações de diferentes domínios passam a poder ser definidos na ATP.

- Fácil extensibilidade dos serviços de monitoramento de ambiente – novos monitores de ambiente (ou de contexto) podem ser facilmente adicionados à plataforma. Dessa forma, requisitos específicos com relação ao monitoramento de ambiente podem ser atendidos pela plataforma.

- Inclusão de políticas de adaptação personalizadas que considera tanto adaptação estrutural quanto comportamental do serviço de transação provido.

- Uma maior usabilidade da plataforma do ponto de vista do usuário (programador de aplicação que se utiliza dos serviços da plataforma). Através de elementos do modelo motherboard, os pontos de configuração (personalização) da plataforma se tornaram mais claros de localizar e mais fáceis de manipular.

A Figura 6.2 apresenta a representação gráfica da MD-View da ATP depois da remodelagem definida no modelo Motherboard. Essa remodelagem foi dividida em duas partes. Enquanto a primeira parte (Figura 6.2(a)) representa elementos do serviço de transações da plataforma ATPm, a segunda parte (Figura 6.2(b)) representa os elementos do serviço de monitoramento de ambiente e de adaptação do serviço de transações.

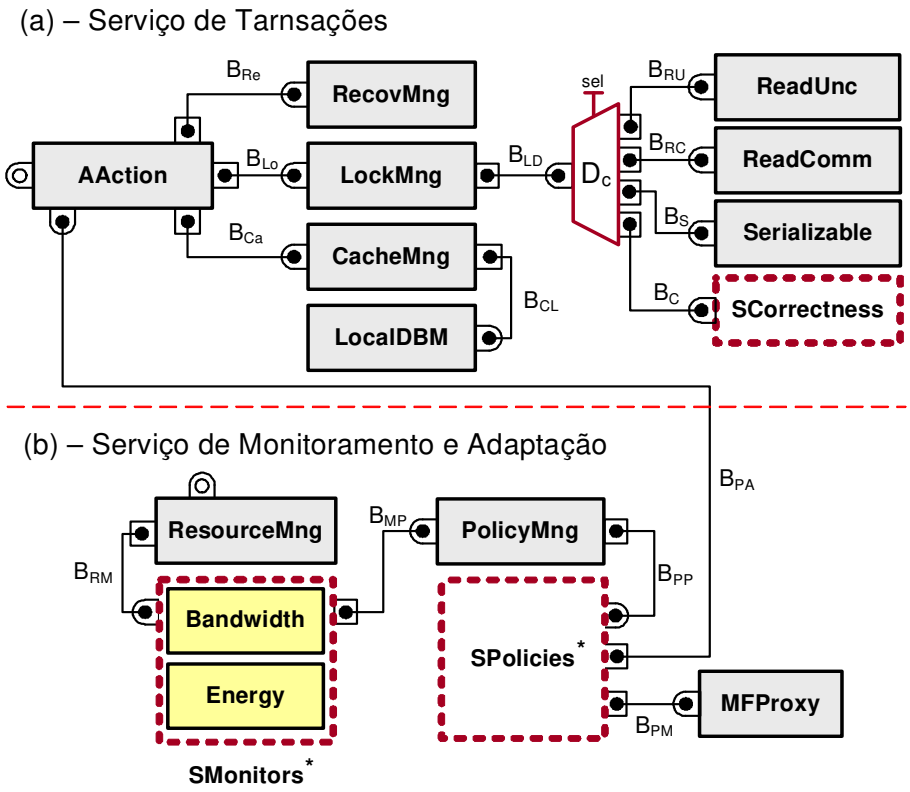

Figura 6.2: A Plataforma ATP no Modelo Motherboard

O Serviço de Transações

Na remodelagem da ATP, o serviço de transações foi estruturado da seguinte forma. O coordenador de transações, o gerenciador de recuperação, o gerenciador de locks, o gerenciador de cache e o gerenciador de base de dados local (*AtomicAction*, *RecovMng*, *LockMng*, *CacheMng* e *LocalDBM*, respectivamente) foram representados como componentes individuais interconectados.

Para dar a capacidade de personalização/configuração de critérios de corretude ao serviço de transação, a implementação do gerenciador de locks (*LockMng*) foi quebrada em uma estrutura que define cada critério de corretude como um componente individual da plataforma. É dessa individualização que nasce a extensibilidade dos critérios de corretude – novos critérios podem ser implementados de forma independente do gerenciador de locks. Para permitir a inclusão desses novos critérios, o modelo proposto define o slot *SCorrectness*. É nesse slot que componentes que implementam novos critérios de corretude são conectados. Para que um componente seja compatível com esse slot, ele deve implementar a sua interface provida *ICorrectness*.

Além do critério de corretude personalizado que pode ser conectado no slot `SCorrectness`,

essa modelagem inclui três critérios de corretude pré-implementados – os critérios *read uncommitted*, *read committed* e *serializable* do padrão ANSI SQL-92 (componentes *ReadUnc*, *ReadComm* e *Serializable*, respectivamente). Para permitir a seleção do critério de corretude desejado, o modelo inclui o demultiplexador D_C. Esse demultiplexador força o gerenciador de locks a utilizar o componente que implementa o critério selecionado.

```
<?xml version='1.0' encoding='UTF-8'?>
<architecture id='ATP'>
  <component id='AAction' source='ATP.AtomicAction'/>
  <component id='RecovMng' source='ATP.RecoveryManager'/>
  <component id='CacheMng' source='ATP.CacheManager'/>
  <component id='LocalDBM' source='ATS.LocalDBM'/>
  <bind id='BLo' client='AAction'
                 server='LockMng' intf='ATP.ITrnEvents'/>
  <bind id='BCa' client='AAction'
                 server='CacheMng' intf='ATP.ICacheManager'/>
  <bind id='BRe' client='AAction'
                 server='RecovMng' intf='ATP.IRecoveryManager'/>
  <bind id='BCL' client='CacheMng'
                 server='LocalDBM' intf='ATP.IDBM'/>
  <component id='LockMng' source='ATP.LockManager'/>
  <demux id='DC' out='BS, BRC, BRU, BC'/>
  <component id='Serializable' source='ATP.CrSerializable'/>
  <component id='ReadComm' source='ATP.CrReadCommitted'/>
  <component id='ReadUnc' source='ATP.CrReadUncommitted'/>
  <slot id='SCorrectness' cardinality="1"/>
  <bind id='BLD' client='LockMng'
                 server='DC' intf='ATP.ICorrectness'/>
  <bind id='BS' client='DC'
                server='Serializable' intf='ATP.ICorrectness'/>
  <bind id='BRC' client='DC'
                 server='ReadComm' intf='ATP.ICorrectness'/>
  <bind id='BRU' client='DC'
                 server='ReadUnc' intf='ATP.ICorrectness'/>
  <bind id='BC' client='DC'
                server='SCorrectness' intf='ATP.ICorrectness'/>
  ...
</architecture>
```

Figura 6.3: MD-View do Serviço de Transações da ATP

A Figura 6.3 mostra a parte da ATP correspondente ao serviço de transações representada na notação XML para MD-Views. Essa representação mostra detalhes das interconexões entre elementos da arquitetura apresentada na Figura 6.2(a).

O Serviço de Monitoramento e Adaptação

Nesta seção, será discutida a segunda parte da ATPm: o serviço de monitoramento de ambiente e de adaptação (Figura 6.2(b)). Esse serviço pode ser utilizado por aplicações para definir as medidas de adaptação que devem ser aplicadas ao serviço de transação como resposta às variações nos recursos do ambiente. Esse serviço é implementado basicamente por dois componentes de controle – o gerenciador de recursos (*ResourceMng*) e o gerenciador de políticas de adaptação (*PolicyMng*).

O gerenciador de recursos recebe requisições de monitoramento de recursos através da sua interface provida *IResourceManager*. Uma requisição de monitoramento é realizada com os seguintes parâmetros:

(i) O identificador do recurso a ser monitorado.

(ii) A janela de tolerância para a variação – consiste em dois valores fornecidos (um de mínimo e um máximo) entre os quais o recurso monitorado pode variar sem a necessidade de medidas de adaptação. Caso o valor medido do recurso seja maior que o valor máximo tolerado ou menor que o valor mínimo tolerado, uma notificação é enviada para o gerenciador de políticas de adaptação.

(iii) O identificador da política de adaptação que deve ser executada caso seja extrapolada a janela de tolerância informada.

O gerenciador de recursos foi estruturado para trabalhar em conjunto com um slot múltiplo de expansão – o *SMonitors*. Nesse slot podem ser conectados componentes que implementam monitores de ambiente (por exemplo, largura de banda, energia) ou de algum outro parâmetro de contexto desejado (por exemplo, taxa de efetivação/cancelamento de transações). Um monitor de ambiente compatível com o slot *SMonitors* deve implementar a sua interface provida (*IResourceRequest*) e sua interface requerida (*IPolicyManager*). A interface provida do slot *SMonitors* é utilizada pelo gerenciador de recursos para requisitar o monitoramento de recursos aos monitores conectados. Já a interface requerida, é utilizada pelos monitores para notificar o gerenciador de política sobre a ocorrência de mudanças nos recursos monitorados.

Similarmente, o gerenciador de políticas foi também estruturado para trabalhar em conjunto com um slot múltiplo de expansão – o *SPolicies*. É nesse slot que políticas de adaptação personalizadas podem ser conectadas. Para cada aplicação que utiliza a

ATP pode ser definida uma política de adaptação. Essa política de adaptação se torna responsável por reconfigurar a ATP de modo a honrar os requisitos da aplicação.

Definir uma nova política de adaptação implica em desenvolver um componente que implementa a interface provida *IPolicy* e as interfaces requeridas *IControl* e *IMother-Frame*. Através da interface *IPolicy*, os componentes de política recebem a autorização para executar a política implementada. Na sua execução, uma política de adaptação utiliza as interfaces requeridas *IControl* e *IMotherFrame* para fazer modificações na estrutura ou no comportamento do serviço de transações. *IControl* é uma interface provida pelo gerenciador de transações (*AAction*) que pode ser usada para requisitar mudanças no comportamento do serviço de transações. Essas mudanças podem consistir na alteração do nível de isolamento ou no modo de operação utilizados. *IMotherFrame* é uma interface provida pelo componente *MFProxy* que pode ser usada para realizar adaptações na estrutura do serviço de transações da ATP.

Nessa arquitetura, a função do componente MFProxy é fornecer os serviços da plataforma MotherFrame para a ATP que, por sua vez, encontra-se representada dentro da própria plataforma MotherFrame. É através dessa estrutura reflexiva que a ATPm consegue acessar e fazer alterações em sua própria estrutura (por exemplo, selecionar diferentes componentes que implementam critérios de corretude através do demultiplexador D_C).

A Figura 6.4 mostra a parte da ATPm correspondente ao serviço monitoramento e adaptação representada na notação XML para MD-Views do modelo motherboard. A união dessa representação com a da Figura 6.3 representa a MD-View completa da ATPm.

6.1.4 Primeiro Cenário de Configuração da ATPm

Nesta seção, será discutido o primeiro cenário de configuração da plataforma ATPm: uma aplicação do conceito de moeda digital. Primeiramente serão apresentados aspectos do ambiente no qual essa aplicação está inserida. Na seqüência, a aplicação de moeda digital será apresentada e seus requisitos serão descritos. Por fim, será apresentado como a plataforma ATPm foi configurada para atender os requisitos dessa aplicação.

O Ambiente de Comunicação Via NFC

Nos últimos anos, tecnologias de comunicação sem fio para curtas distâncias têm obtido grande popularidade entre usuários de dispositivos de computação móveis. Dentre essas tecnologias podem ser destacados o Bluetooth [27] e o Wi-Fi [74] que limitam a comunicação sem fio a um alcance de poucas dezenas ou centenas de metros. Seguindo uma nova demanda para dispositivos de comunicação com alcance ainda mais reduzido, a Sony e a Philips montaram uma parceria da qual foi desenvolvido o NFC (Comunicação em Campo de Curta Distância) [45, 47] – uma nova tecnologia de comunicação com um al-

```
<?xml version='1.0' encoding='UTF-8'?>
<architecture id='ATP'>
   ...
  <component id='ResourceMng' source='ATP.ResourceManager'/>
  <component id='PolicyMng' source='ATP.PolicyManager'/>
  <component id='MFProxy' source='ATP.MotherFrameProxy'/>
  <slot id='SMonitors' cardinality="*"/>
  <slot id='SPolicies' cardinality="*"/>
  <bind id='BRM' client='ResourceMng'
                  server='SMonitors' intf='ATP.IResourceRequest'/>
  <bind id='BMP' client='SMonitors'
                  server='PolicyMng' intf='ATP.IPolicyManager'/>
  <bind id='BPP' client='PolicyMng'
                  server='SPolicies' intf='ATP.IPolicy'/>
  <bind id='BPM' client='SPolicies'
                  server='MFProxy' intf='MotherFrame.IMotherFrame'/>
  <bind id='BPA' client='SPolicies'
                 server='AAction' intf='MotherFrame.IControl'/>
</architecture>
```

Figura 6.4: MD-View dos Serviços de Controle da ATP

cance de menos de vinte centímetros. As vantagens de limitar esse alcance à casa de poucos centímetros estão em questões como: (i) segurança – é necessária a proximidade física para se estabelecer uma comunicação e se executar transações; (ii) consumo de energia – devido à curta distância, a energia necessária para o circuito NFC estabelecer a comunicação passa a ser mínima[4].

A combinação entre a tecnologia de comunicação NFC e o dispositivo móvel mais popular do mundo – o telefone celular – tem fornecido bases promissoras para a real implantação de diversos novos tipos de aplicação. Através dessa combinação, o aparelho celular pode passar a funcionar como diferentes tipos de dispositivos: (i) chave eletrônica para controle de acesso em ambientes físicos, (ii) cartão de passagem para transportes públicos, (iii) cartão de pagamentos (restaurantes, lojas, teatros) e de transferência de dinheiro ponto a ponto[6]. Para executar essas funções bastaria aproximar o celular de um leitor habilitado e confirmar a operação.

Esse novo ambiente computacional (celular + NFC) tem lançado novos desafios em diversas áreas da computação, dentre as quais estão as de serviços de transação e de criptografia. Em especial, com relação aos serviços de transação, foram levantados alguns

[4]Um circuito NFC pode também trabalhar de forma passiva, ou seja, captando energia do outro dispositivo com o qual ele está se comunicando.

requisitos necessários para esse tipo de ambiente:

1. **Flexibilidade**. Como diferentes tipos de aplicações podem fazer uso desse ambiente, é esperado que os requisitos dessas aplicações sejam também diferentes. Sendo assim, para que a aplicabilidade do serviço de transação seja mais ampla, ele deve prover propriedades flexíveis para atender à diversidade de requisitos.

2. **Personalização**. Mesmo provendo propriedades transacionais flexíveis, existem aplicações que possuem requisitos tão específicos que vão além do limite dessa flexibilidade. Para atender esses requisitos, o serviço de transações deve prover pontos de extensibilidade que permitam ao desenvolvedor da aplicação implementar os novos requisitos como parte integrante do serviço de transação.

3. **Baixo peso**. Como telefones celulares normalmente possuem recursos de memória e armazenamento limitados, o serviço de transação deve possuir um tamanho adequado a esses recursos.

4. **Adaptabilidade**. Como esse ambiente pode apresentar um alto dinamismo às aplicações – desconexões inesperadas (por sair do campo de alcance), desligamento do aparelho (por falta de energia nas baterias), e variações na largura de banda (ao se mudar de interface de comunicação – Wi-Fi, Bluetooth) – mecanismos de adaptação podem ser necessários para evitar inconsistências, perdas de desempenho e altas taxas de cancelamento de transações.

A Aplicação de Moeda Digital

Esta seção apresenta uma aplicação promissora e que se tornou possível com a combinação de telefone celular com NFC: a moeda digital. O termo moeda digital se refere à moeda (dinheiro) que é mantida e repassada na forma digital. Moeda digital é um tema que há mais de duas décadas vem sendo estudado [15]. Nesses últimos anos, o surgimento de tecnologias como o NFC tem impulsionado ainda mais esses estudos [6, 14, 105]. Isso tem ocorrido porque as idéias que envolvem o NFC fornecem fortes bases para a sua real implantação.

Com base na combinação celular-NFC, a utilização prática da moeda digital poderia ser dada da seguinte maneira. Considere uma estrutura em que os usuários (potenciais compradores de produtos e serviços) estão equipados com celulares-NFCs e os vendedores (ex: lojas e restaurantes) estão equipados com leitores[5] NFC. O usuário sacaria uma quantia em dinheiro de sua conta bancária e a armazenaria na forma digital no seu celular.

[5]Apesar de serem chamados de leitores, esses dispositivos dos vendedores são aptos a se comunicar em duas vias – tanto enviam quanto recebem informações

Feito isso, o celular poderia ser utilizado para efetuar pagamentos – o comprador transferiria o dinheiro digital para o vendedor através dessa comunicação entre o seu celular e o leitor NFC do vendedor. Passagens de transporte público, entradas em teatros, compras em supermercado são outros exemplos de pagamentos que poderiam ser feitos através do celular.

Requisitos Específicos da Aplicação da Moeda Digital

A seguir são listados alguns requisitos que uma aplicação de moeda digital deverá satisfazer:

(i) **Critério de corretude**. Como essa é uma aplicação que manipula dados monetários, é exigida uma total consistência dos dados nas transações executadas. Para isso, o critério de corretude mais adequado para esse tipo de aplicação seria o da serialização. Esse é o critério de corretude adotado como padrão em aplicações que realizam transações bancárias.

(ii) **Diferentes vias de pagamento**. A aplicação de moeda digital deve fornecer ao usuário diferentes vias de pagamento para que o usuário possa escolher a mais adequada. Seguem alguns dos possíveis tipos de pagamento que poderiam ser disponibilizados:

- **Pagamento direto**. Essa é a forma de pagamento que foi discutida anteriormente. O usuário transfere previamente uma quantia de dinheiro digital para seu celular e se torna apto a realizar pagamentos transferindo dinheiro diretamente do seu celular para os leitores NFCs dos vendedores.
  - **Vantagens**: maior disponibilidade do dinheiro no celular para uso *offline* – sem a necessidade de se conectar ao banco no momento da compra;
  - **Desvantagens**: em caso de perda ou roubo do aparelho celular, o dinheiro digital pré-armazenado seria perdido.
- **Pagamento indireto**. Essa forma de pagamento é feita do banco do comprador para o banco do vendedor. Primeiramente, dados bancários do vendedor seriam passados do leitor NFC do mesmo para o celular do comprador. Em seguida, seria estabelecida uma conexão remota entre o celular do comprador e o seu banco para passagem dos dados de transferência (através de outra interface de comunicação). O banco do comprador transfere o valor para o banco do vendedor que, por sua vez, notifica o vendedor do sucesso da transação. Todas essas etapas seriam feitas de forma transparente para o vendedor e o comprador.

  - **Vantagens**: garantia contra perda do dinheiro em conseqüência de perda, roubo ou inutilização do celular;
  - **Desvantagens**: é necessária a disponibilidade de conexão com o banco para realizar a transferência para o celular da quantia necessária para a compra. Essa comunicação passa a depender da disponibilidade de uma segunda interface de comunicação.

- **Pagamento com crédito**. Essa via de pagamento funciona de forma similar à de cartões de crédito. Dados de crédito são passados do usuário do celular para o leitor de NFC do vendedor. A partir desses dados, o valor da compra é transferido do banco (ou operadora de cartão de crédito) do comprador para o banco do vendedor.
  - **Vantagens**: garantia contra perda do dinheiro em conseqüência de perda, roubo ou inutilização do celular; não é necessária a disponibilidade de conexão entre o celular e o banco do cliente
  - **Desvantagens**: risco de captação ilegal dos dados de créditos do comprador que são passados para o leitor NFC do vendedor.

(iii) **Mecanismos de adaptação**. Como cada tipo de pagamento tem suas limitações, um mecanismo de adaptação permite a seleção do tipo de pagamento mais adequado a depender do estado ou da situação em que o ambiente se encontra. Um algoritmo de adaptação que poderia ser utilizado é o seguinte:

```
SE houver alguma conexão gratuita de internet disponível[6] ENTÃO
    Utilizar pagamento indireto
SENÃO
    SE tiver dinheiro em cash disponível ENTÃO
        Utilizar pagamento direto
    SENÃO
        Utilizar pagamento com crédito
    FIM SE
FIM SE
```

Esse algoritmo de adaptação tenta priorizar as formas mais seguras de pagamento. A primeira tentativa é o pagamento direto em que, apesar de depender da disponibilidade de conexão com o banco, não há envio de dados confidenciais do cliente para o leitor NFC do vendedor. Se a primeira tentativa falhar (ex: não há disponibilidade

[6] Por exemplo, uma conexão aberta do tipo Wi-Fi

de conexão), tentar-se-ia o pagamento direto repassando o valor da compra para o vendedor na forma de moeda digital. Se não houver no celular dinheiro digital suficiente para o pagamento, tentar-se-ia o pagamento com crédito que seria a forma de pagamento menos segura.

Configuração da ATPm para a Aplicação de Moeda Digital

Até esse ponto já foram discutidos aspectos do ambiente envolvido (celular + NFC), uma aplicação para esse ambiente (a moeda digital) e os requisitos específicos dessa aplicação. Agora será discutido como a ATP – remodelada no modelo motherboard – estaria apta a prestar seus serviços a essa aplicação honrando os requisitos descritos.

Para satisfazer os requisitos da aplicação da moeda digital apresentados na seção anterior, foram tomadas uma série de decisões de configuração junto à ATPm. O resultado dessas decisões foi representado em uma especificação XML do tipo MU-View (mostrado na Figura 6.5). O resultado da aplicação dessa MU-View na ATPm é mostrado na Figura 6.6. Essa MU-View foi aplicada à MD-View da ATPm apresentada anteriormente na Figura 6.2. A seguir, para cada requisito da aplicação de moeda digital, são descritas as decisões de configuração da ATPm adotadas:

```
<configuration id='MoedaDigital' arch='ATPm'>
   <setmux id='DC' switch='BS'/>
   <component id='Connection' source='br.MD.Connection'/>
   <slotmap id='SMonitors' component='Connection'/>
   <component id='MOPolicy' source='br.MD.MOPolicy'/>
   <slotmap id='SPolicies' component='MOPolicy'/>
</configuration>
```

Figura 6.5: MU-View de Configuração da ATPm para a Aplicação de Moeda Digital

- **Requisito "*(i)*":** *implementar o critério de serialização*

  Como a plataforma ATPm disponibiliza três critérios de corretude integrados dentre os quais está o de serialização, bastou selecionar esse critério. Para isso, foi selecionada a saída B_S do demultiplexador D_C da plataforma. Selecionar essa saída significa forçar o gerenciador de locks da ATPm a utilizar o critério de serialização (Implementado pelo componente *Serializable*).

- **Requisito "*(ii)*":** *viabilizar as três vias de pagamento*

  Para a viabilização das três vias de pagamento requeridas, foram utilizados dois modos de operação providos pela ATPm: o modo local e o modo remoto. O modo

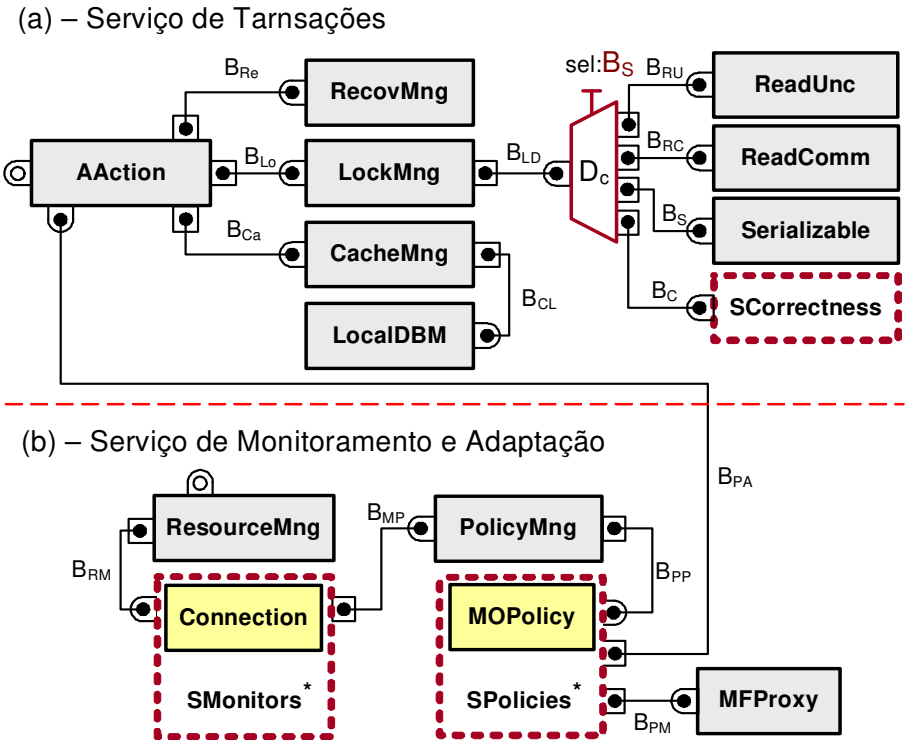

Figura 6.6: Configuração da ATP para o Cenário da Moeda Digital

de operação local foi usado tanto na via de pagamento direta quanto na via de pagamento com crédito. Em ambos os casos, as transações de pagamento executam sobre dados locais armazenados na memória do celular. No primeiro caso, a transação é composta basicamente da seguinte seqüência de operações: leitura do valor em moeda digital disponível no celular; subtração do pagamento do valor total disponível; comunicação com o leitor do vendedor para enviar o valor da compra; escrita do novo valor disponível na memória do celular. A via de pagamento do segundo caso consiste em uma transação local do tipo somente leitura que lê os dados de crédito do usuário e os envia para o leitor do vendedor.

O modo de operação remoto foi usado para realizar a via de pagamento indireta. Nesse caso a transação de pagamento executa diretamente sobre os dados remotos da conta bancária do usuário. A operação dessa transação consiste justamente em uma operação de transferência bancária solicitada ao sistema do banco.

- **Requisito "*(iii)*":** *algoritmo de adaptação*

  Para satisfazer esse requisito, foi implementado o algoritmo de adaptação descrito na seção anterior. Na prática, implementar esse algoritmo implicou em configurar o

serviço de monitoramento e de adaptação da plataforma ATPm da seguinte maneira: implementando um monitor de ambiente e uma política de adaptação. O monitor de ambiente ficou responsável por detectar as conexões disponíveis dentro do alcance do celular. Esse monitor foi implementado pelo componente *Connection* que foi conectado no slot $SMonitors$. A política de adaptação ficou responsável por decidir o tipo de pagamento a ser utilizado, baseado em informações que incluem o estado do ambiente fornecido pelo monitor de ambiente implementado. Essa política foi implementada pelo componente *MOPolicy* que foi conectado no slot $SPolicies$.

Para escolher o tipo de pagamento que deverá ser usado, o componente *MOPolicy* utiliza a sua interface requerida *IControl* para alterar o modo de operação com o qual o serviço de transações deverá trabalhar. Na Figura 6.6 essa interface é usada para conectar o slot *SPolicies* com o componente *AAction* (através do bind B_{PA}).

6.1.5 Segundo Cenário de Configuração da ATPm

Nesta seção será apresentado o segundo cenário de configuração da ATPm – uma aplicação para o serviço de resgates médicos de emergência. A próxima seção descreve o ambiente no qual essa aplicação está inserida. Em seguida será descrita a aplicação e, por fim, será apresentada uma configuração da ATPm para atender os requisitos dessa aplicação.

O Ambiente de Resgates Médicos de Emergência

Considere o cenário comum de sistemas de atendimento de saúde de cidades modernas. Uma rede de hospitais mantém serviços móveis de resgate de cidadãos em situação de emergência. Na equipe de resgate estão incluídos médicos e enfermeiros capazes de prestar os socorros de emergência. Para isso, esse serviço conta com ambulâncias e helicópteros capazes de levar a equipe de resgate até a cena do incidente.

As várias experiências reais com esse tipo de cenário têm confirmado que o rápido acesso a certas informações pode fazer a diferença entre a vida e a morte de vítimas envolvidas em incidentes [89]. Essas informações podem incluir:

(i) *dados do paciente* – tipo sanguíneo, históricos de doenças e alergias a medicamentos;

(ii) *dados do sistema de atendimento de saúde* – hospitais especializados mais próximos e equipes médicas disponíveis para o pronto-atendimento;

(iii) *outros dados relevantes* – o trajeto mais rápido entre o local do acidente e o hospital desejado.

Em vista da importância dessas informações, novas tecnologias têm sido empregadas para permitir o seu pronto-acesso durante o socorro às vítimas [76, 93, 89, 36] que incluem dispositivos de computação portáteis equipados com interface de comunicação sem fio. Através desses dispositivos, a equipe poderia acessar as informações necessárias junto a sistemas mantidos remotamente (por exemplo, sistemas de informações de saúde da rede de hospitais e sistemas de informações de tráfego).

Apesar da existência da infra-estrutura necessária, o acesso a essas informações não é uma tarefa fácil de garantir. Estudos de casos mostram falhas que têm comprometido esse acesso em situações reais de emergência [89]. Essas falhas incluem: perda da conexão causada pela interferência de obstáculos físicos como montanhas, túneis e edificações; esgotamento da energia e do espaço em disco disponíveis nos dispositivos portáteis; atrasos no recebimento de informações causados pela baixa largura de banda disponível.

É em busca de soluções para obstáculos como esses que projetos como o MIDAS [36] e o Ad-hoc InfoWare [76] têm reunido instituições de pesquisa de diversos países. Em geral, a busca da solução tem sido no desenvolvimento de serviços de middleware capazes de suportar as características específicas desse ambiente. É com base nesses serviços que as aplicações de emergência adequadas ao ambiente poderiam ser desenvolvidas.

Um dos serviços de middleware que precisam ser reestruturados para se adequar ao ambiente em questão é o de transações. Em serviços de transação tradicionais de acesso a dados distribuídos, uma meta a ser atingida é a consistência do dado acessado. Uma razão pela qual essa estratégia não pode ser adotada nesse tipo de ambiente é a seguinte: ao se tentar manter a total consistência de acesso a dados, poderiam ocorrer atrasos altamente consideráveis no acesso como efeito colateral – o que contraria o requisito de rápida resposta do cenário de resgates de emergência.

Uma outra questão que precisa ser considerada são as possíveis desconexões. A desconexão é fato sempre presente nesse ambiente. Ela pode ser tanto voluntária – para economizar as baterias da unidade móvel[7] – quanto involuntária – causada por perda de sinal e interferências. Para lidar com esse fator, os serviços de transação devem viabilizar meios de dar às aplicações de emergência a capacidade de continuarem a executar na unidade móvel mesmo desconectadas dos sistemas remotos. Para isso, esses serviços devem utilizar técnicas de caching de dados de modo que eles se tornem disponíveis nos momentos de desconexão.

Uma Aplicação para o Serviço de Resgate

Nesta seção, será descrita uma aplicação específica projetada para o cenário do serviço móvel de atendimento de emergência. O propósito dessa aplicação é executar nos disposi-

[7]Sabe-se que a comunicação sem fio é um dos elementos que mais consome a energia das baterias de dispositivos móveis.

tivos móveis das equipes de resgate fornecendo informações necessárias a um atendimento mais eficaz às vítimas de incidentes. Nesse sentido, estes seriam alguns dos requisitos funcionais dessa aplicação:

(1) Permitir à equipe de emergência acesso aos dados médicos da vítima. Cada cidadão possui seus dados (tipo sanguíneo, históricos de doenças e alergias a medicamentos) armazenados nas bases do sistema da rede de hospitais.

(2) Permitir à equipe de emergência o acesso a dados de disponibilidade da rede de hospitais no tratamento do caso em questão. Esses dados informariam, por exemplo, quais hospitais especializados no caso estariam com profissionais disponíveis para o pronto-atendimento da vítima.

(3) Permitir o envio prévio de dados que descrevem o estado da vítima para o hospital escolhido. Isso permite ao hospital preparar a equipe (ex: médicos e enfermeiros) e os recursos médicos (ex: centro cirúrgico, medicamentos) antes mesmo da chegada da vítima.

Pode-se notar que para atender esses requisitos, deve haver um fluxo de dados em duas vias: **(i)** do sistema da rede de hospitais para a aplicação do dispositivo móvel da equipe de resgate – requisitos "(1)" e "(2)" – e **(ii)** da aplicação do dispositivo móvel da equipe de resgate para o sistema da rede de hospitais – requisito "(3)".

Além dos requisitos funcionais, o projeto dessa aplicação possui os seguintes requisitos não funcionais:

(A) O acesso aos dados da vítima junto à base da rede de hospitais deve ser imediato – mesmo que sua consistência não seja garantida de imediato.

(B) A equipe do hospital para onde o paciente está sendo levado deve ter acesso imediato às informações sobre o estado da vítima informadas pela equipe de resgate que atua no local do incidente.

(C) Os recursos de energia das baterias do dispositivo móvel devem ser utilizados de forma racional – economizando-se quando as fontes de recarregamento forem inexistentes.

(D) Considerada a possível instabilidade do canal de comunicação sem fio do dispositivo móvel, deve-se prover meios de viabilizar a execução da aplicação mesmo se ela estiver desconectada. Na medida do possível, a aplicação deve continuar a receber e a fornecer dados à equipe de resgate mesmo sem comunicação com o sistema da rede de hospitais.

Configuração da ATPm para a Aplicação do Serviço de Resgate

Esta seção apresenta uma configuração para a ATPm de modo a atender os requisitos da aplicação do serviço móvel de emergência descritos na seção anterior. A representação gráfica da MU-View dessa configuração é representada na Figura 6.7.

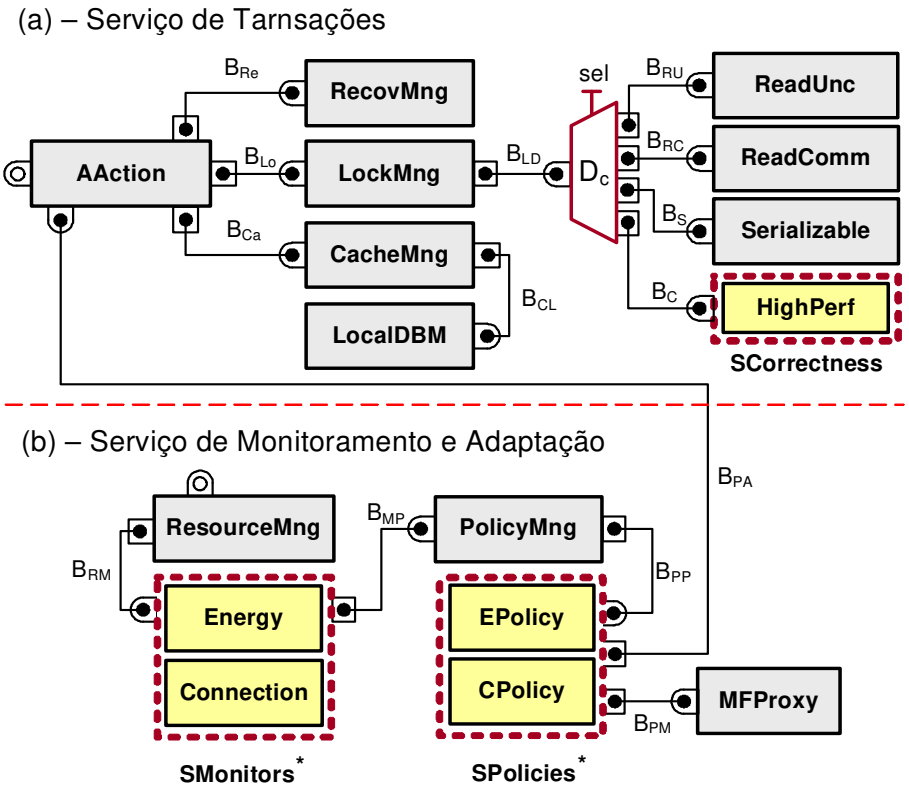

Figura 6.7: Configuração da ATP para o Cenário do Serviço de Saúde

Considerando os requisitos da aplicação descritos na seção anterior, serão mostradas as ações de configuração que foram tomadas junto à ATPm para atendê-los.

- **Requisitos "*(A)*" e "*(B)*":** *acesso imediato a dados*

  Para que o serviço de transações da ATPm pudesse atender os requisitos "*(A)*" e "*(B)*", foi projetado um critério de corretude personalizado de alto desempenho chamado *HighPerf*. A implementação desse critério de corretude é mostrada na Tabela 6.2.

  Para permitir acesso imediato aos dados da vítima por parte da equipe de resgate (requisito "*(A)*"), esse critério de corretude permite leitura de dados sem a necessidade de aquisição de trancas de leitura. Assim, a transação tem acesso livre aos dados requisitados junto ao controlador de concorrência.

Para permitir acesso imediato aos dados que compõem o estado da vítima por parte da equipe do hospital (requisito "*(B)*"), esse critério atribui trancas curtas de escrita aos dados da vítima que vão sendo atualizados pela equipe de resgate que atua no local do incidente. Isso significa que, logo após a atualização de cada item de dado da vítima por parte da equipe de resgate, a equipe do hospital já se torna apta a acessá-lo tomando as medidas necessárias em preparação à chegada da vítima.

	HighPerf	
	Read	Write
begin_item		$LOCK_I$
end_item		$unLOCK_I$
begin_predicate		$LOCK_P$
end_predicate		$unLOCK_P$

Tabela 6.2: Implementação do Critério de Corretude HighPerf

Para que o critério de corretude *HighPerf* pudesse ser incluído no serviço de transação da ATPm, ele foi implementado como um componente individual para ser conectado ao slot $SCorrectness$ (Figura 6.7). Para utilizar esse critério de corretude, a saída B_C do demultiplexador D_C foi selecionada.

- **Requisito "*(C)*":** *fazer uso racional da energia disponível*

  Usar as reservas de energia do dispositivo de forma racional visa garantir que ela esteja disponível toda vez que o dispositivo móvel precisar ser utilizado. Para isso, a ATPm foi configurada para tomar algumas medidas de racionamento de energia quando necessárias. Nesse sentido, a configuração da ATPm incluiu: *(i)* a implementação de um monitor de energia capaz de fornecer informações acerca da sua disponibilidade; e *(ii)* uma política de adaptação que força o uso racional das reservas de energia do dispositivo.

  O monitor de energia foi definido como um componente individual ($Energy$) projetado para ser conectado no slot $SMonitors$ do serviço de monitoramento e adaptação da ATPm. Esse monitor é mostrado na Figura 6.7(b).

  Além disso, foi definida uma política de adaptação que utiliza as informações do monitor de energia para tomar medidas relacionadas à utilização racional das reservas disponíveis. Essa política foi definida como um componente individual ($EPolicy$) que foi projetado para ser conectado no slot $SPolicies$ do serviço de monitoramento e adaptação da ATPm (Figura 6.7(b)).

  A política de adaptação definida pelo componente $EPolicy$ é basicamente a que segue. *(i)* O *EPolicy* recebe periodicamente a informação acerca da disponibilidade

de energia por parte do monitor *Energy*. Essa informação indica se a fonte de energia é inesgotável (fornecida fonte externa confiável de energia) ou esgotável (fornecimento limitado pela carga disponível das baterias internas). Quando a fonte é esgotável, também é recebida a informação da previsão de duração da carga de energia armazenada. *(ii)* Se a fonte for inesgotável, o dispositivo pode ser mantido conectado com o sistema da rede de hospitais através da conexão sem fio. *(iii)* Se a fonte for esgotável, a conexão sem fio é encerrada – depois de confirmado que os dados necessários ao atendimento da vítima já estão disponíveis para o uso *offline* na unidade móvel. *(iv)* A conexão poderá ser restabelecida em duas situações: quando é necessário o acesso a novos dados ou quando a fonte de energia passa a ser do tipo inesgotável.

- **Requisito "*(D)*":** *viabilizar operação desconectada*

  Para permitir que a aplicação no dispositivo móvel continue a fornecer informações à equipe de resgate mesmo na ocorrência de desconexões, foram tomadas as seguintes medidas de configuração da ATPm: *(i)* definição de um monitor de conexão; *(ii)* definição de uma outra política de adaptação.

  Assim como o monitor de energia, o monitor de conexão foi projetado para ser conectado ao slot *SMonitors*. Esse monitor foi definido como um componente individual (*Connection*) capaz de fornecer informações acerca da disponibilidade de conexão entre o dispositivo móvel da equipe de resgate e o sistema da rede de hospitais.

  Para atender esse requisito, uma política de adaptação foi definida. Ela é responsável por reconfigurar o serviço de transação da ATPm com base nas informações obtidas a partir do monitor de conexão. Na prática, essa reconfiguração consiste em selecionar o *modo de operação* do serviço de transação mais adequado ao estado da conexão.

  Na ATPm, essa política de adaptação foi definida pelo componente $CPolicy$ – projetado para ser conectado ao slot $SPolicies$ (Figura 6.7(b)). Na prática, essa política é definida como segue: (i) Se a conexão estiver ativa, pode-se utilizar o modo de operação remoto do serviço de transações – os dados são acessados a partir das bases do sistema da rede de hospitais. Paralelamente são copiados para a cache do dispositivo móvel, os principais dados relativos à vítima atendida. (ii) Caso haja desconexão – causada pela perda involuntária de sinal ou pela desconexão voluntária guiada pela política de economia de energia – o serviço de transações é reconfigurado para o modo de operação local. No modo local, os dados passam a ser acessados a partir da cache do dispositivo móvel da equipe. Durante o período de desconexão, a equipe de resgate pode realizar atualizações nos dados da vítima. Essas atualizações são armazenadas temporariamente na cache da unidade móvel.

(iii) Quando a conexão for restabelecida, o serviço de transação poderá voltar a operar no modo remoto e acessar demais dados a respeito da vítima nas bases remotas da rede de hospitais. Se houver atualizações dos dados da vítima em cache, essas são propagadas para as bases do hospital.

A Figura 6.8 mostra a representação XML da configuração MU-View para a aplicação do serviço de resgate descrito. Na seqüência, essa representação *(i)* define os componentes utilizados na configuração ($HighPerf$, $Connection$, $Energy$, $EPolicy$ e $CPolicy$), *(ii)* seleciona a saída B_C do demultiplexador D_C, *(iii)* mapeia os componentes $Connection$ e $Energy$ para o slot $SMonitors$, e *(iv)* mapeia os componente $EPolicy$ e $CPolicy$ para o slot $SPolicies$.

```
<configuration id='ServiçoRegate' arch='ATPm'>
   <component id='HighPerf' source='br.SR.HighPerformance'/>
   <component id='Connection' source='br.SR.ConnectionMonitor'/>
   <component id='Energy' source='br.SR.EnergyMonitor'/>
   <component id='EPolicy' source='br.SR.EnergySavingPolicy'/>
   <component id='CPolicy' source='br.SR.ConnectionPolicy'/>
   <setmux id='DC' switch='BC'/>
   <slotmap id='SMonitors' component='Connection'/>
   <slotmap id='SMonitors' component='Energy'/>
   <slotmap id='SPolicies' component='EPolicy'/>
   <slotmap id='SPolicies' component='CPolicy'/>
</configuration>
```

Figura 6.8: MU-View da Configuração da ATPm para a Aplicação do Serviço de Resgate

6.2 Segundo Estudo – O ACOM

O segundo estudo de caso é baseado em um serviço de transação que provê adaptação dinâmica de protocolos de efetivação - o ACOM (self-Adaptive Component-based cOmmit Management). O ACOM foi remodelado no modelo motherboard para superar algumas de suas limitações.

A apresentação deste estudo está estruturada como segue. Na próxima seção será introduzido o conceito de protocolos de efetivação. Em seguida o ACOM será discutido destacando-se os seus benefícios e limitações. Por fim será mostrado como as limitações do ACOM podem ser superadas através da sua reestruturação no modelo motherboard.

6.2.1 Protocolos de Efetivação

O protocolo de efetivação é um mecanismo essencial na garantia da propriedade da atomicidade de transações que executam operações em bases de dados distribuídas. Ele visa garantir que ou todas as operações sejam efetivadas (confirmadas como permanentes) ou todas as operações sejam abortadas (desfeitas).

A execução do protocolo de efetivação é uma das fases da transação que mais influenciam no seu desempenho. Isso ocorre porque são requeridos dois tipos de operações consideradas de alto tempo de resposta: (i) operações de comunicação remota – necessárias para a comunicação entre o coordenador e os participantes; (ii) operações de escrita em disco – necessárias para o processo de logging realizado pelo protocolo.

O protocolo de efetivação mais utilizado é o 2PC (*two-phase commit*[8]). Apesar do 2PC ser um protocolo considerado eficaz na garantia da atomicidade de transações distribuídas, ele é considerado um protocolo de baixo desempenho. Para superar essa limitação, várias otimizações do 2PC têm sido propostas. Os protocolos PrC e PrA estão entre as suas otimizações mais comuns [2].

O protocolo 2PC e seus variantes são projetados para executar em um ambiente distribuído composto por um coordenador e vários participantes. O coordenador é o ponto onde a aplicação transacional está localizada e os participantes são os pontos distribuídos onde estão localizadas as bases de dados acessadas pela aplicação.

Para garantir a tolerância às falhas ocorridas no momento da execução, o progresso do protocolo normalmente é armazenado em logs mantidos tanto pelo coordenador quanto pelos participantes. As operações de escrita nesses logs podem ser de dois tipos: *force* ou *non-force*. Operações do tipo *force* são escritas imediatamente no log, o que gera um acesso a disco. Operações do tipo *non-force* só são escritas no log eventualmente. Apesar de demandarem menos acesso a disco, escritas do tipo *non-force* ficam vulneráveis a perdas até que elas sejam escritas em disco [95].

As seções a seguir apresentam uma breve introdução do protocolo 2PC e dos variantes 2PrC e 2PrA.

Two Phase Commit (2PC)

Como o seu nome sugere, o protocolo 2PC é composto por duas fases: a fase de votação e a fase de decisão. Na fase de votação, o coordenador envia um pedido de voto (*prepare*) para cada um dos participantes envolvidos. Como resposta, cada participante pode decidir se a transação deve ser efetivada ou cancelada. Depois de recebidos todos os votos, o protocolo passa para a fase de decisão. Se todos os participantes concordarem em efetivar a transação, o coordenador envia a decisão de efetivação para os participantes. Se ao

[8]Protocolo de efetivação em duas fases

menos um participante votar para abortar a transação, o coordenador envia a decisão de cancelamento para os participantes. Ao receber a decisão cada participante responde ao coordenador com uma mensagem *acknowledge*[9].

No protocolo 2PC, o coordenador realiza uma escrita ao log do tipo *force* e uma do tipo *non-force* – a primeira para registrar a decisão e a segunda para registrar o fim da execução do protocolo. Além disso, cada participante realiza duas escritas ao log do tipo *non-force* – para registrar o seu voto e a decisão do coordenador.

Presumed Commit (PrC)

O protocolo PrC é uma otimização do protocolo 2PC que reduz o custo de transações efetivadas. O PrC interpreta a falta de informação como uma decisão de efetivação. Para isso, o coordenador escreve um registro de inicialização do tipo *force* antes de enviar o pedido de voto (*prepare*) aos participantes. Quando o coordenador decide efetivar a transação, ele escreve um registro de efetivação do tipo *force* e, em seguida, envia a decisão aos participantes. Cada participante registra como *non-force* a decisão de efetivação e libera os recursos transacionais sem enviar mensagem de *acknowledge* ao coordenador.

No protocolo PrC, o caso de cancelamento (*abort*) é tratado de maneira diferente. Quando o coordenador decide cancelar uma transação, ele envia a decisão de cancelamento para os participantes e espera pelas mensagens de *acknowledge*. Essa decisão não é registrada pelo coordenador. Cada participante escreve como *force* a decisão de cancelamento e envia a mensagem de *acknowledge* ao coordenador. Ao receber as mensagens de *acknowledge*, o coordenador registra como *non-force* o fim do protocolo.

Em comparação ao 2PC, o PrC economiza uma mensagem de acknowledge de cada participante e uma escrita ao log do tipo force para o caso de efetivação. Em contrapartida, o PrC inclui uma escrita extra do tipo *force* no coordenador para registrar a início do protocolo. No total, o custo de efetivar uma transação com o PrC é de $2 + p$ escritas do tipo *force* e de $3p$ mensagens[10].

Presumed Abort (PrA)

O protocolo PrA é uma otimização do PrC que, ao contrário do PrC, reduz o custo de transações abortadas. No PrA, quando o coordenador decide abortar uma transação, ele envia mensagens de cancelamento para todos os participantes sem gravar a decisão no log. Cada participante escreve como *non-force* um registro de cancelamento sem a necessidade de enviar mensagem de *acknowledge* ao coordenador. No caso de efetivação, o PrA funciona da mesma forma que o 2PC.

[9] Mensagem onde o participante confirma estar ciente da decisão.
[10] A variável p indica o número de participantes.

Em comparação ao 2PC, o PrA economiza uma escrita do tipo *force* no coordenador e em cada participante, além de economizar uma mensagem de *acknowledge* em cada participante no caso de abort. No total, o custo para se cancelar uma transação no PrA é de $3p$ mensagens e de p escritas do tipo *force*. O custo de efetivar uma transação no PrA é o mesmo do 2PC.

6.2.2 ACOM – Gerenciamento Auto-Adaptável de Protocolos de Efetivação

Ao analisar os custos de efetivar e de cancelar uma transação nos protocolos PrC e PrA, pode-se notar que o PrC possui o menor custo para transações efetivadas, enquanto que o PrA possui o menor custo para transações canceladas. Considerando essa característica, em [95] foi proposto o gerenciador de transações ACOM[11]. O ACOM é capaz de alterar dinamicamente o protocolo de efetivação utilizado com base nas taxas de efetivação e de cancelamento. Com isso o ACOM visa obter sempre o melhor custo dentre os protocolos de efetivação utilizados.

Os Protocolos de Efetivação

O ACOM generaliza a definição dos protocolos de efetivação 2PC, PrC e PrA. Seu objetivo é prover uma implementação desses protocolos como um conjunto de componentes interconectados. Cada protocolo seria implementado pelo mesmo conjunto de componentes, sendo que suas principais diferenças seriam expressas somente através de mudanças nas suas interconexões.

A Figura 6.9 mostra como os protocolos 2PC, PrC e PrA podem ser definidos no ACOM através de mudanças nas conexões dos componentes. O *CommitManager* atua como o coordenador, enviando mensagens de *prepare*, *commit* e *abort* [12] a cada participante – representado pelo *ResourceManager* – e registrando os passos do protocolo no log. O componente *CommunicationManager*, implementado como um barramento de comunicação, permite ao coordenador enviar mensagens síncronas e assíncronas (com *acknowledge* e sem *acknowledge*, respectivamente) aos participantes. Para o coordenador e para cada participante, existe um componente *LogManager* – responsável por prover escritas ao log do tipo *force* e do tipo *non-force*.

No ACOM, os protocolos de efetivação são implementados como segue:

2PC. Para implementar esse protocolo (Figura 6.9(a)), o *CommitManager* envia as mensagens *prepare*, *commit* e *abort* de forma síncrona através da interface *sync* do

[11] ACOM é um acrônimo para o termo "self-Adaptive Component-based cOmmit Management"

[12] *prepare* representa o pedido de voto, *commit* representa a decisão de efetivação e *abort* representa a decisão de cancelamento

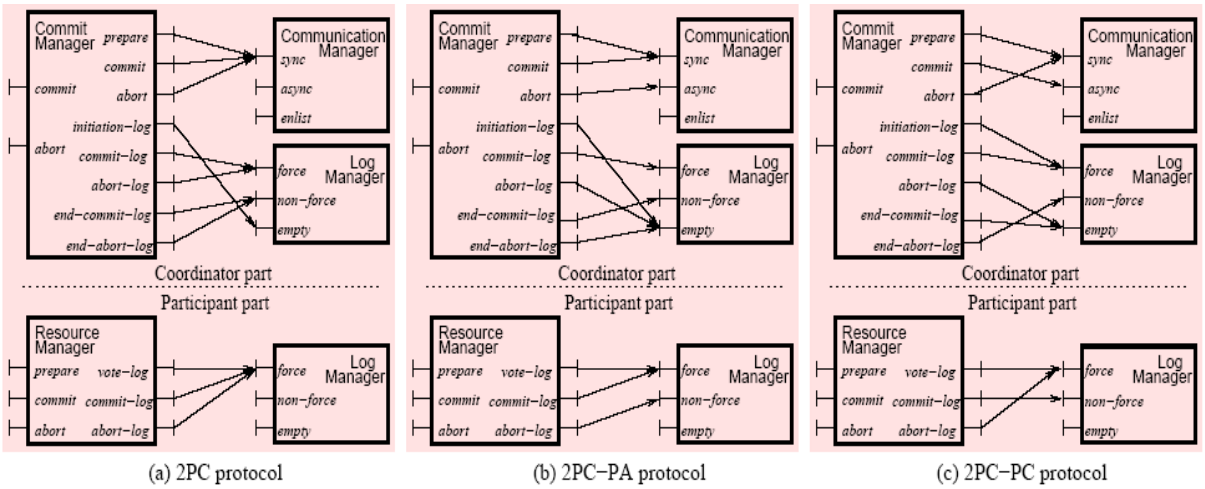

Figura 6.9: Arquitetura dos Protocolos de Efetivação no ACOM

CommunicationManager. As operações de decisão de efetivação e de cancelamento são escritas como *force* no *log* através da interface *force* do LogManager. No final do protocolo, o resultado da transação é escrita no *log* como *non-force* através da interface *non-force* do *LogManager*. O *ResourceManager*, ao receber as mensagens de *prepare*, *commit* e *abort* do *CommitManager* realiza registros correspondentes no log do tipo *force*.

PrA. Nessa implementação (Figura 6.9(b)), a mensagem *prepare* e a decisão de efetivação são enviadas pelo *CommitManager* de forma síncrona através da interface *sync* do *CommunicationManager*. Ao contrário do 2PC, a decisão de cancelamento é enviada de forma assíncrona através da mensagem *asyn*. No PrA, o *CommitManager* registra a decisão de efetivação como *force* e registra a finalização de transações efetivadas como *non-force*. O *ResourceManager* registra o voto e a decisão de efetivação como *force* e a decisão de cancelamento como *non-force*.

PrC. Nessa implementação (Figura 6.9(c)), antes de enviar a mensagem *prepare*, o *CommitManager* utiliza a interface *initiation-log* para registrar a inicialização do protocolo como *force* junto ao *LogManager*. No PrC, a mensagem *prepare* e a decisão de cancelamento são enviadas pelo *CommitManager* de forma síncrona através da interface *sync* do *CommunicationManager*. Por outro lado, a decisão de efetivação é enviada de forma assíncrona através da mensagem *asyn*. A decisão de efetivação é registrada no log como *force* e o final de transações abortadas são registradas como *non-force*. O *ResourceManager* registra o seu voto e a decisão de cancelamento como *force* e a decisão de efetivação como *non-force*.

O Gerenciador de Comportamento

Para que o ACOM possa decidir e configurar dinamicamente o protocolo de efetivação de menor custo, ele deve conhecer as taxas de efetivação e de cancelamento das transações que vão sendo executadas. Para isso ele define um gerenciador de comportamento capaz de medir essas taxas e de aplicar a política de adaptação. Essa política é do tipo ECA (*Evento/Condição/Ação*), onde o *Evento* é a taxa de efetivação/cancelamento, a *Condição* especifica quando o protocolo deve ser mudado e a *Ação* é a reconfiguração do mecanismo de efetivação para implementar o protocolo escolhido.

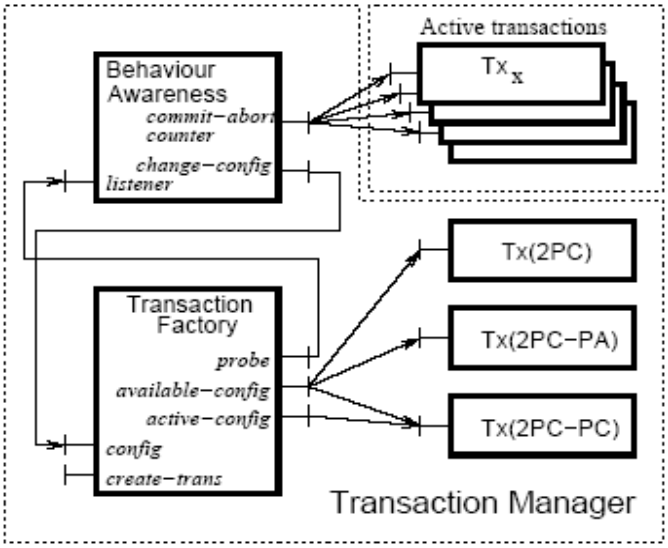

Figura 6.10: Arquitetura do Gerenciador de Comportamento no ACOM

O componente *BehaviourAwareness* implementa a política de adaptação. Ele monitora o número de transações efetivadas e canceladas e decide quando o protocolo deve ser alterado. A alteração do protocolo é determinada pela regra ECA implementada pelo *BehaviourAwareness*. Por exemplo, uma regra ECA poderia ser: se taxa–de–abort $<$ 10% então use 2PC.

Quando o *BehaviourAwareness* decide alterar o protocolo, ele utiliza a sua interface *change-config* que está conectada à interface *config* do componente *TransactionFactory*. Ao receber a decisão, o *TransactionFactory* conecta sua interface *active-config* à configuração apropriada. Dessa forma, as próximas transações são criadas usando a nova configuração. O conjunto de configurações disponíveis é listado pela interface *available-config* do *TransactionFactory*. Cada configuração disponível é definida individualmente por um componente. *Tx(2PC)*, *Tx(2PC-PC)* e *Tx(2PC-PA)* são os componentes que definem as configurações dos protocolos de efetivação 2PC, PrC e PrA, respectivamente.

6.2.3 Limitações do ACOM

Apesar de ACOM ser capaz de se reconfigurar dinamicamente para obter o melhor custo dentre os protocolos de efetivação 2PC, PrC e PrA, sua estrutura dificulta possíveis extensões (como se verá a seguir). Dada a variedade de requisitos das aplicações transacionais, outras otimizações do protocolo 2PC têm sido propostas na literatura, como o 1PC [3] e o NPrC [56].

Estender o ACOM para incluir novos protocolos de efetivação é uma tarefa possível, porém complexa. Por exemplo, para incluir um novo protocolo ao ACOM (ex:NPrC), o programador teria que realizar as seguintes tarefas:

(i) Identificar todos os componentes envolvidos na implementação do mecanismo de efetivação do ACOM.

(ii) Identificar as interfaces envolvidas nas interconexões entre os componentes que implementam o mecanismo de efetivação.

(iii) Conhecer cada conexão a ser criada e cada conexão a ser excluída para implementar o novo protocolo.

(iv) Implementar e compilar o componente de configuração do novo protocolo a ser incluído no ACOM (ex: Tx(NPrC)).

(v) Analisar e alterar a implementação do componente *BehaviourAwareness* para incluir a política de adaptação que considera o uso do novo protocolo.

(vi) Incluir o componente de configuração (Tx(NPrC)) na lista de configurações disponíveis (*available-config*) da *TransactionFactory*.

Para realizar essas tarefas são necessárias a análise detalhada do código fonte do ACOM e a modificação do código que implementa um dos seus componentes – o *BehaviourAwareness*. Além de difícil e enfadonho, essa alteração de código fonte pode levar a erros de implementação do novo protocolo pondo em risco a estabilidade e a precisão do serviço de transações. Isso reforça a premissa de que possuir uma arquitetura baseada em componentes não é garantia absoluta de facilidade de extensões [35, 97].

6.2.4 Remodelando o ACOM no Modelo Motherboard

Esta seção apresenta uma solução para facilitar as extensões do ACOM. Ela reestrutura a arquitetura do ACOM incluindo elementos abertos de mais alto nível propostos pelo modelo motherboard. Essa reestruturação do ACOM será chamada de ACOMm. Ao

invés de ter que analisar e manipular o código fonte (classes, componentes, interfaces e conexões), o usuário do ACOMm passa a manipular elementos de mais alto nível do modelo motherboard, como demultiplexadores, jumpers e slots. Por exemplo, no ACOMm, para definir a decisão de efetivação como síncrona o usuário precisa somente fornecer um parâmetro que diz "a decisão de commit é síncrona" ao invés de ter que realizar a tarefa de baixo nível "conectar a interface *commit* do componente *CommitManager* à interface *sync* do componente *CommunicationManager*".

O Mecanismo de Efetivação

O mecanismo de efetivação do ACOMm é mostrado na Figura 6.11. Ele reutiliza os componentes do ACOM incluindo um conjunto de elementos abertos do modelo motherboard (jumpers e slots). No ACOMm, a definição de diferentes tipos de protocolos de efetivação passa a ser possível somente através da manipulação desses elementos abertos.

Para definir protocolos de efetivação, são definidos elementos abertos tanto na arquitetura do coordenador como na do participante. Os elementos abertos do coordenador são (Figura 6.11(a)):

J_{SP} . Jumper que pode ser ativado para que o pedido de voto (*prepare*) do coordenador seja realizado de forma síncrona ou desativado quando o protocolo não possui a fase de pedido de voto (como no protocolo 1PC)[13].

D_C . Demultiplexador que permite que a decisão de efetivação seja configurada como síncrona ou assíncrona. Para síncrona, o parâmetro de seleção tem o valor "BCS" e, para assíncrona, o valor "BCA".

D_A . Demultiplexador que permite que a decisão de cancelamento seja configurada como síncrona ou assíncrona. Para síncrona, o parâmetro de seleção tem o valor "BAS" e, para assíncrona, o valor "BAA".

D_{IL} . Demultiplexador que permite que informações de inicialização do protocolo sejam registrados no log como *force* ou como *non-force*. Para *force*, o parâmetro de seleção tem o valor "BIF" e, para *non-force*, o valor "BIN". Caso não haja registro de inicialização, deve-se fornecer o valor "NULL".

J_{CF} . Jumper que pode ser ativado para que a decisão de efetivação seja registrada no log como *force* ou desativado quando não há registro da decisão de efetivação.

[13]Como se pode notar, a arquitetura não disponibiliza a opção de pedido de voto assíncrono. Isso se baseia no fato de que quando existe pedido de voto, ele será sempre síncrono por ele exigir a resposta (voto) dos participantes.

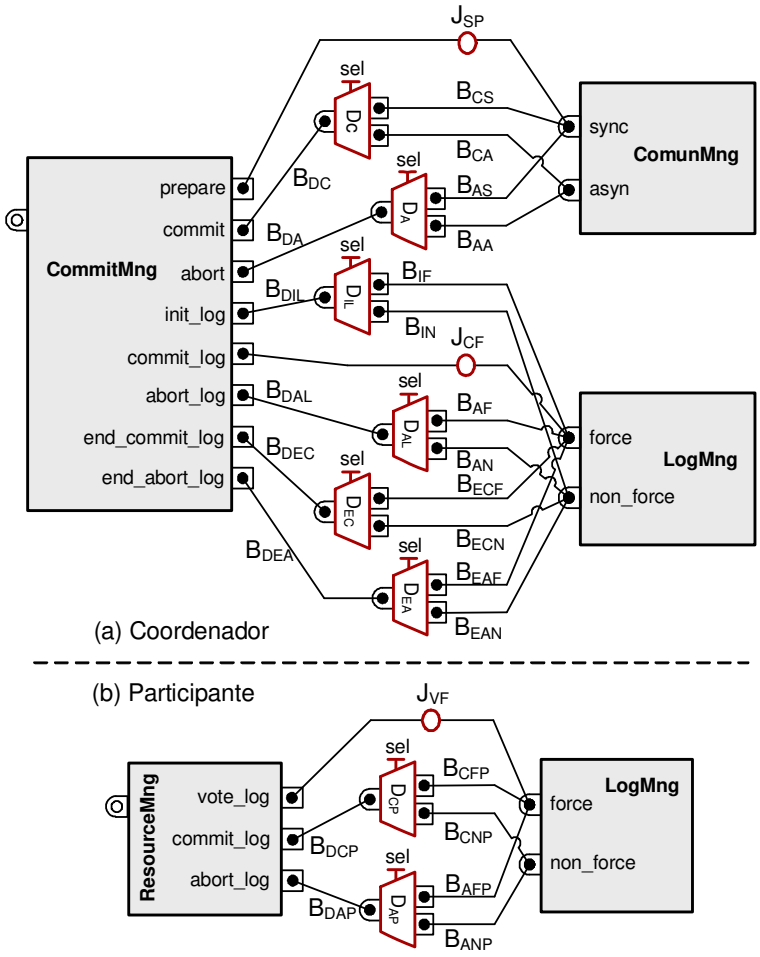

Figura 6.11: O Mecanismo de Efetivação do ACOM no Modelo Motherboard

D_{AL} . Demultiplexador que permite que a decisão de cancelamento do protocolo seja registrada no log como *force* ou como *non-force*. Para *force*, o parâmetro de seleção tem o valor "BAF" e, para *non-force*, o valor "BAN". Caso não haja registro de inicialização, deve-se fornecer o valor "NULL".

D_{EC} . Demultiplexador que permite que o final do protocolo de transações efetivadas seja registrada no log como *force* ou como *non-force*. Para *force*, o parâmetro de seleção tem o valor "BECF" e, para *non-force*, o valor "BECN". Caso não haja registro de inicialização, deve-se fornecer o valor "NULL".

D_{EA} . Demultiplexador que permite que o final do protocolo de transações canceladas seja registrada no log como *force* ou como *non-force*. Para *force*, o parâmetro de seleção tem o valor "BEAF" e, para *non-force*, o valor "BEAN". Caso não haja

registro de inicialização, deve-se fornecer o valor "NULL".

Os elementos abertos do participante são (Figura 6.11(b)):

J_{VF} . Jumper que pode ser ativado para que o voto do participante seja registrado no log como *force* ou desativado quando não há registro do voto.

D_{CP} . Demultiplexador que permite que a decisão de efetivação do protocolo seja registrada no log como *force* ou como *non-force*. Para *force*, o parâmetro de seleção tem o valor "BCFP" e, para *non-force*, o valor "BCNP". Caso não haja registro de inicialização, deve-se fornecer o valor "NULL".

D_{AP} . Demultiplexador que permite que a decisão de cancelamento do protocolo seja registrada no log como *force* ou como *non-force*. Para *force*, o parâmetro de seleção tem o valor "BAFP" e, para *non-force*, o valor "BANP". Caso não haja registro de inicialização, deve-se fornecer o valor "NULL".

A representação XML da MD-View do mecanismo de efetivação do ACOMm é mostrada na Figura 6.12 e na Figura 6.13. A primeira figura apresenta a parte do coordenador e a segunda figura apresenta a parte do participante.

```
<?xml version='1.0' encoding='UTF-8'?>
<architecture id='ACOM'>
 <component id='CommitMng' source='ACOM.CommitManager'/>
 <component id='ComunMng' source='ACOM.CommunicationManager'/>
 <component id='LogMng' source='ACOM.LogManager'/>
 <jumper id='JSP' client='CommitMng'
                  server='ComunMng' intf='ACOM.ISyncPrepare'/>

 <demux id='DC' out='BCS, BCA'/>
 <bind id='BDC' client='CommitMng' server='DC' intf='ACOM.IComm'/>
 <bind id='BCS' client='DC' server='ComunMng' intf='ACOM.ISync'/>
 <bind id='BCA' client='DC' server='ComunMng' intf='ACOM.IAsyn'/>

 <demux id='DA' out='BAS, BAA'/>
 <bind id='BDA' client='CommitMng' server='DA' intf='ACOM.IAbort'/>
 <bind id='BAS' client='DA' server='ComunMng' intf='ACOM.ISync'/>
 <bind id='BAA' client='DA' server='ComunMng' intf='ACOM.IAsyn'/>

 <demux id='DIL' out='BIF, BIN'/>
 <bind id='BDIL' client='CommitMng' server='DIL' intf='ACOM.IILog'/>
 <bind id='BIF' client='DIL' server='LogMng' intf='ACOM.IForce'/>
 <bind id='BIN' client='DIL' server='LogMng' intf='ACOM.INonForce'/>

 <jumper id='JCF' client='CommitMng'
                  server='LogMng' intf='ACOM.ICommitForce'/>

 <demux id='DAL' out='BAF, BAN'/>
 <bind id='BDAL' client='CommitMng' server='DAL' intf='ACOM.IAbLog'/>
 <bind id='BAF' client='DAL' server='LogMng' intf='ACOM.IForce'/>
 <bind id='BAN' client='DAL' server='LogMng' intf='ACOM.INonForce'/>

 <demux id='DEC' out='BECF, BECN'/>
 <bind id='BDEC' client='CommitMng' server='DEC' intf='ACOM.IECLog'/>
 <bind id='BECF' client='DEC' server='LogMng' intf='ACOM.IForce'/>
 <bind id='BECN' client='DEC' server='LogMng' intf='ACOM.INonForce'/>

 <demux id='DEA' out='BEAF, BEAN'/>
 <bind id='BDEA' client='CommitMng' server='DEA' intf='ACOM.IEALog'/>
 <bind id='BEAF' client='DEA' server='LogMng' intf='ACOM.IForce'/>
 <bind id='BEAN' client='DEA' server='LogMng' intf='ACOM.INonForce'/>
</architecture>
```

Figura 6.12: MD-View do Mecanismo de Efetivação do ACOM (Coordenador)

```
<?xml version='1.0' encoding='UTF-8'?>
<architecture id='ACOM'>
 <component id='ResourceMng' source='ACOM.ResourceManager'/>
 <component id='LogMng' source='ACOM.CommitManager'/>
 <jumper id='JVF' client='ResourceMng'
                  server='LogMng' intf='ACOM.ICommitForce'/>

 <demux id='DCP' out='BCFP, BCNP'/>
 <bind id='BDCP' client='ResourceMng' server='DCP' intf='ACOM.ICLog'/>
 <bind id='BCFP' client='DCP' server='LogMng' intf='ACOM.IForce'/>
 <bind id='BCNP' client='DCP' server='LogMng' intf='ACOM.INonForce'/>

 <demux id='DAP' out='BAFP, BANP'/>
 <bind id='BDAP' client='ResourceMng' server='DA' intf='ACOM.IAbLog'/>
 <bind id='BAFP' client='DAP' server='LogMng' intf='ACOM.IForce'/>
 <bind id='BANP' client='DAP' server='LogMng' intf='ACOM.INonForce'/>
</architecture>
```

Figura 6.13: MD-View do Mecanismo de Efetivação do ACOM (Participante)

O Gerenciador de Comportamento

Além do mecanismo de efetivação, o gerenciador de comportamento do ACOM também foi reestruturado no modelo motherboard. Essa reestruturação teve o objetivo de tornar o ACOM extensível no que diz respeito à inclusão de novas políticas de adaptação e de novos configuradores de protocolo de efetivação. A principal medida nesse sentido foi a de realizar uma separação entre o código que implementa as políticas de adaptação e o código que implementa o componente *BehaviourAwareness*. Dessa forma, novas políticas de adaptação poderiam ser incluídas sem a necessidade de alterar o código do *BehaviourAwareness*.

A reestruturação do gerenciador de comportamento é mostrada na Figura 6.14. Para obter maior facilidade de extensão, ele conta com a inclusão de dois elementos abertos do modelo motherboard: o slot *SPolicies* e o slot *SConfig*. Esses slots são descritos a seguir:

SPolicies. Esse slot múltiplo foi a solução adotada para se separar as políticas de adaptação do componente *BehaviourAwareness*. Nessa estrutura, cada política de adaptação passa a ser implementada como um componente individual que pode ser conectado ao slot *SPolicies*. Para se definir um componente de política, deve-se implementar a interface provida *IRateEvent* – através da qual o *BehaviourAwareness* passa informações sobre as taxas de efetivação e de cancelamento – e a interface requerida

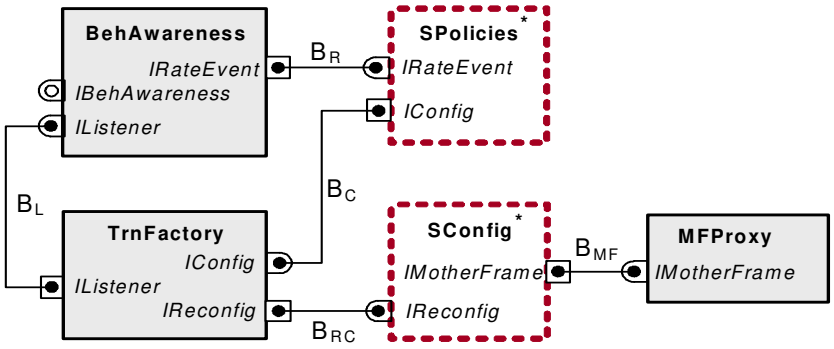

Figura 6.14: O Gerenciador de Comportamento do ACOM no Modelo Motherboard

IConfig – através da qual o componente de política dispara a ação de reconfiguração do protocolo de efetivação.

SConfig. Nesse slot devem ser conectados os configuradores de protocolos de efetivação. Cada configurador é responsável por manipular os elementos abertos do mecanismo de efetivação do ACOMm de forma que ele passe a implementar o protocolo de efetivação desejado. Um componente configurador deve implementar a interface provida *IReconfig* – a partir da qual ele recebe a autorização para reconfigurar o mecanismo de efetivação – e a interface requerida *IMotherFrame* – a partir da qual o configurador acessa e manipula os elementos abertos do mecanismo de efetivação (demultiplexadores e jumpers) de forma a configurá-lo de acordo com o protocolo correspondente.

A representação XML da MD-View do gerenciador de comportamento é mostrada na Figura 6.15.

Nas seções a seguir será mostrado como o ACOMm possui a flexibilidade necessária para ser configurado de acordo com requisitos de diferentes cenários.

6.2.5 Primeiro Cenário de Configuração do ACOMm

Este primeiro cenário de configuração mostra como o ACOMm pode ser configurado para permitir adaptação dinâmica entre três protocolos de efetivação: 2PC, PrC e PrA. Esse cenário é o mesmo para o qual foi proposto o ACOM [95], com a exceção de que no ACOMm essa configuração pode ser feita de maneira mais clara e modular – sem a necessidade de alterar o código fonte do ACOMm.

A representação gráfica da MU-View para essa configuração é mostrada na Figura 6.16. Ela mostra os novos componentes que foram definidos para atingir essa configuração: *Policy1* – define a política de adaptação do mecanismo de efetivação do ACOMm; *Tx2PC*,

```
<?xml version='1.0' encoding='UTF-8'?>
<architecture id='ACOM'>
 <component id='BehAwareness' source='ACOM.BehaviourAwareness'/>
 <component id='TrnFactory' source='ACOM.TransactionFactory'/>
 <component id='MFProxy' source='ACOM.MotherFrameProxy'/>
 <slot id='S_Config' cardinality="*"/>
 <slot id='S_Policies' cardinality="*"/>
 <bind id='BL' client='TrnFactory'
               server='BehAwareness' intf='ACOM.IListener'/>
 <bind id='BR' client='BehAwareness'
               server='S_Policies' intf='CaseStudy1.IRateEvent'/>
 <bind id='BC' client='S_Policies'
               server='TrnFactory' intf='ACOM.IConfig'/>
 <bind id='BRC' client='TrnFactory'
                server='S_Config' intf='ACOM.IReconf'/>
 <bind id='BMF' client='S_Config'
                server='MFProxy' intf='IMotherFrame'/>
</architecture>
```

Figura 6.15: MD-View do Gerenciador de Comportamento do ACOM

TxPrC e *TxPrA* – definem como o mecanismo de efetivação deve ser configurado para implementar os protocolos 2PC, PrC e PrA, respectivamente.

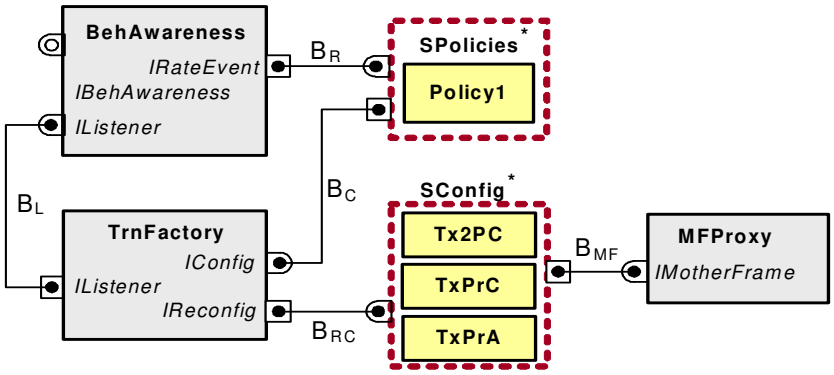

Figura 6.16: Primeira Configuração do Gerenciador de Comportamento do ACOMm

O componente v*Policy1* define uma regra do tipo Evento/Condição/Ação. O evento é a taxa de efetivação que é enviada periodicamente pelo *BehAwareness* para o *Policy1* através da conexão B_R. A condição adotada foi a seguinte: se a taxa de efetivação for menor que 54% solicitar a configuração PrA, senão, se a taxa de efetivação for maior ou igual a 54% solicitar a configuração PrC. Essas solicitações de configuração são justamente

as ações da regra ECA. O componente solicita essas ações ao *TrnFactory* através da conexão B_C.

Os componentes *Tx2PC*, *TxPrC* e *TxPrA* recebem o pedido de reconfiguração enviado pelo *TrnFactory* através da conexão B_{RC}. Para realizar a configuração desejada do mecanismo de efetivação, esses componentes acessam a interface *IMotherFrame* do *MF-Proxy* através da conexão B_{MF}. É através de operações dessa interface que os elementos abertos do mecanismo de efetivação do ACOM são manipulados.

Configuração do 2PC

Para configurar o mecanismo de efetivação do ACOMm com o protocolo 2PC, o componente *Tx2PC* aplica ao coordenador a MU-View representada na Figura 6.17 e aplica a cada participante a MU-View representada na Figura 6.18.

```
<?xml version='1.0' encoding='UTF-8'?>
<configuration id='2PC' architecture='ACOM'>
 <jumper id='JSP' state='ON'/>
 <setmux id='DC' switch='BCS'/>
 <setmux id='DA' switch='BAS'/>
 <jumper id='JCF' state='ON'/>
 <setmux id='DAL' switch='BAF'/>
 <setmux id='DIL' switch='NULL'/>
 <setmux id='DEC' switch='BECN'/>
 <setmux id='DEA' switch='BEAN'/>
</configuration>
```

Figura 6.17: MD-View do Mecanismo de Efetivação para o 2PC (Coordenador)

```
<?xml version='1.0' encoding='UTF-8'?>
<configuration id='2PC' architecture='ACOM'>
 <jumper id='JVF' state='ON'/>
 <setmux id='DCP' switch='BCFP'/>
 <setmux id='DAP' switch='BAFP'/>
</configuration>
```

Figura 6.18: MD-View do Mecanismo de Efetivação para o 2PC (Participantes)

Configuração do PrC

Para configurar o mecanismo de efetivação do ACOMm com o protocolo PrC, o componente *TxPrC* aplica ao coordenador a MU-View representada na Figura 6.19 e aplica a cada participante a MU-View representada na Figura 6.20.

```
<?xml version='1.0' encoding='UTF-8'?>
<configuration id='PrC' architecture='ACOM'>
 <jumper id='JSP' state='ON'/>
 <setmux id='DC' switch='BCA'/>
 <setmux id='DA' switch='BAS'/>
 <jumper id='JCF' state='ON'/>
 <setmux id='DAL' switch='NULL'/>
 <setmux id='DIL' switch='BIF'/>
 <setmux id='DEC' switch='NULL'/>
 <setmux id='DEA' switch='BEAN'/>
</configuration>
```

Figura 6.19: MD-View do Mecanismo de Efetivação para o PrC (Coordenador)

```
<?xml version='1.0' encoding='UTF-8'?>
<configuration id='PrC' architecture='ACOM'>
 <jumper id='JVF' state='ON'/>
 <setmux id='DCP' switch='BCNP'/>
 <setmux id='DAP' switch='BAFP'/>
</configuration>
```

Figura 6.20: MD-View do Mecanismo de Efetivação para o PrC (Participantes)

Configuração do PrA

Para configurar o mecanismo de efetivação do ACOMm com o protocolo PrA, o componente *TxPrA* aplica ao coordenador a MU-View representada na Figura 6.21 e aplica a cada participante a MU-View representada na Figura 6.22.

```
<?xml version='1.0' encoding='UTF-8'?>
<configuration id='PrA' architecture='ACOM'>
 <jumper id='JSP' state='ON'/>
 <setmux id='DC' switch='BCS'/>
 <setmux id='DA' switch='BAA'/>
 <jumper id='JCF' state='ON'/>
 <setmux id='DAL' switch='NULL'/>
 <setmux id='DIL' switch='NULL'/>
 <setmux id='DEC' switch='BECN'/>
 <setmux id='DEA' switch='NULL'/>
</configuration>
```

Figura 6.21: MD-View do Mecanismo de Efetivação para o PrA (Coordenador)

```
<?xml version='1.0' encoding='UTF-8'?>
<configuration id='PrA' architecture='ACOM'>
 <jumper id='JVF' state='ON'/>
 <setmux id='DCP' switch='BCFP'/>
 <setmux id='DAP' switch='BANP'/>
</configuration>
```

Figura 6.22: MD-View do Mecanismo de Efetivação para o PrA (Participantes)

6.2.6 Segundo Cenário de Configuração do ACOMm

No primeiro cenário, foi apresentada uma configuração do ACOMm para permitir a escolha dinâmica do protocolo de adaptação mais adequado dentre o 2PC, o PrC e o PrA. Neste segundo cenário, será apresentada uma outra configuração para o ACOMm que considera a escolha dinâmica entre dois protocolos: o NPrC e PrA. Esse segundo cenário mostra como o ACOMm é extensível à configuração de outros protocolos de efetivação baseados no 2PC além daqueles definidos no ACOM.

O NPrC[14] [56] é uma otimização do protocolo PrC que não realiza a escrita de inicial-

[14]New Presumed Commit Protocol

ização do tipo *force* no log. No PrC, essa escrita é realizada para armazenar informações sobre os participantes da transação no log. Em caso de falhas, essas informações são acessadas para se realizar o contato com os participantes envolvidos no processo de recuperação. Ao invés de escrever essas informações de inicialização no log, o protocolo NPrC realiza algumas escritas do tipo *non-force* no log que, da mesma forma que o PrC permitirá a recuperação de informações a respeito das transações envolvidas. Na prática, o protocolo NPrC troca a escrita do tipo *force* por eventuais escritas do tipo *non-force*.

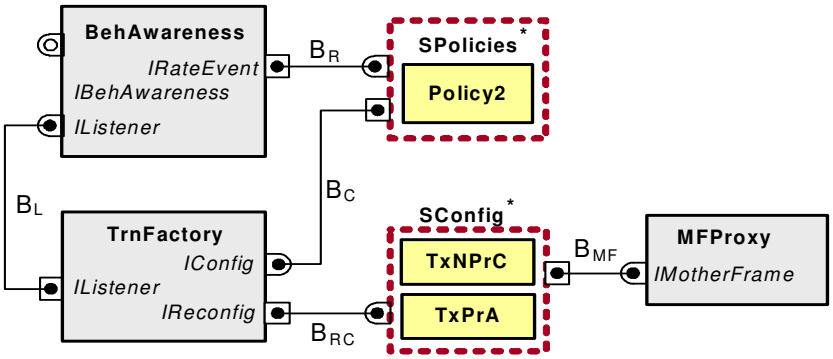

Figura 6.23: Segunda Configuração do Gerenciador de Comportamento do ACOMm

A configuração do ACOMm para esse cenário envolveu a definição de três componentes (Figura 6.23): *Policy2* – define a política de adaptação capaz de definir o protocolo a ser utilizado (NPrC ou PrA), *TxNPrC* e *TxPrA* – definem as configurações do ACOMm para os protocolos NPrC e PrA, respectivamente. Do mesmo modo que o componente *Policy1* do primeiro cenário, *Policy2* define a política de adaptação seguindo uma regra do tipo ECA. Ao receber a taxa de efetivação/cancelamento do *BehAwareness*, *Policy2* analisa uma condição e executa as ações correspondentes como segue: se a taxa de efetivação for menor que 54%, solicitar a configuração PrA, senão, se a taxa de efetivação for maior que 54%, solicitar a configuração NPrC.

Ao receber o comando de reconfiguração, os componentes *TxNPrC* e *TxNPrA* realizam o processo de configuração do protocolo correspondente através das operações fornecidas pelo componente *MFProxy*. A seção a seguir apresenta como o *TxNPrC* manipula os elementos abertos do mecanismo de efetivação do ACOMm para implementar o protocolo NPrC. Como o componente *TxPrA* foi reutilizado do primeiro cenário, a sua descrição é a mesma da seção anterior.

Configuração do NPrC

Para configurar o mecanismo de efetivação do ACOMm com o protocolo NPrC, o componente *TxNPrC* aplica ao coordenador a MU-View representada na Figura 6.24 e aplica

a cada participante a MU-View representada na Figura 6.25. A única diferença entre a configuração do PrC e a do NPrC é que nessa não é realizado registro de inicialização, ou seja, o demultiplexador D_{IL} do coordenador é chaveado para o valor "NULL".

```
<?xml version='1.0' encoding='UTF-8'?>
<configuration id='NPrC' architecture='ACOM'>
 <jumper id='JSP' state='ON'/>
 <setmux id='DC' switch='BCA'/>
 <setmux id='DA' switch='BAS'/>
 <jumper id='JCF' state='ON'/>
 <setmux id='DAL' switch='NULL'/>
 <setmux id='DIL' switch='NULL'/>
 <setmux id='DEC' switch='NULL'/>
 <setmux id='DEA' switch='BEAN'/>
</configuration>
```

Figura 6.24: MD-View do Mecanismo de Efetivação para o NPrC (Coordenador)

```
<?xml version='1.0' encoding='UTF-8'?>
<configuration id='NPrC' architecture='ACOM'>
 <jumper id='JVF' state='ON'/>
 <setmux id='DCP' switch='BCNP'/>
 <setmux id='DAP' switch='BAFP'/>
</configuration>
```

Figura 6.25: MD-View do Mecanismo de Efetivação para o NPrC (Participantes)

Capítulo 7

Resultados Experimentais

Este capítulo apresenta os resultados experimentais realizados na plataforma MotherFrame em comparação com o ambiente de execução do OpenCOM. A próxima seção apresenta a motivação e os objetivos dos experimentos. Em seguida será descrito o ambiente no qual foram realizados os experimentos. Na seqüência serão apresentados os experimentos realizados e os resultados obtidos. Por fim, será apresentada uma discussão dos experimentos realizados.

7.1 Motivação e Objetivos

Como já foi discutido nesta tese, a plataforma MotherFrame foi implementada como uma camada de abstração em cima do ambiente do modelo de componentes OpenCOM. No geral, quando nova camada de abstração é adicionada sobre uma já existente ela induz a uma perda de desempenho. Essa perda é causada pelo processamento extra que a nova camada normalmente demanda.

Este capítulo se propõe a apresentar medições que indicam qual é a perda de desempenho introduzida pela plataforma MotherFrame com relação ao OpenCOM. Para isso, um conjunto de experimentos de configuração implementados na plataforma MotherFrame foram confrontadas com um outro conjunto de experimentos equivalentes implementadas diretamente no ambiente OpenCOM. Esses experimentos medem o tempo necessário para que as seguintes ações de configuração sejam realizadas: instanciação de componentes, conexão de componentes em slots, alteração do estado de jumpers e seleção da saída de demultiplexadores.

Os experimentos realizados na plataforma MotherFrame abrangerão as duas formas de configuração providas pela mesma: a submissão de uma especificação XML do tipo MU-View e o uso das operações da API provida pela MotherFrame através da interface IMotherFrame. Essas duas formas foram comparadas com as realizadas diretamente no

OpenCOM.

Para exemplificar como um mesmo experimento pode ser realizado de forma equivalente na MotherFrame (API e XML) e no OpenCOM, pode-se observar um experimento simples que instancia um componente chamado *CompEx*:

1. No OpenCOM essa instanciação seria feita através da chamadas das seguintes operações:

```
IUnknown cExUnk = (IUnknown) openCOM.createInstance("br.unicamp.CompEx", "CompEx");
ILifeCycle cExLife = (ILifeCycle) cExUnk.QueryInterface("OpenCOM.ILifeCycle");
adderLife.startup(openCOM);
```

2. Na MotherFrame API essa instanciação seria feita através da seguinte operação da interface IMotherFrame:

```
mf.createComponent("CompEx", "br.unicamp.CompEx");
```

3. Na MotherFrame XML essa instanciação seria feita através da submissão da seguinte especificação na plataforma MotherFrame:

```
<configuration id='ConfEx' arch='ArchEx'>
   <component id='CompEx' source='br.unicamp.CompEx'/>
</configuration>
```

Para medir as operações de configuração que não existem nativamente no ambiente OpenCOM, foi utilizada a estratégia da emulação. Por exemplo, uma das operações que forma emuladas no OpenCOM é a ativação/desativação de jumpers. Para ativar um jumper, foi usada a operação de criação de conexão disponibilizada pelo OpenCOM. Para desativar, foi usada a operação de exclusão de conexão. De maneira similar, outros elementos como slots e demultiplexadores foram emulados no OpenCOM.

7.2 O Ambiente dos Experimentos

7.2.1 Descrição

O ambiente no qual os experimentos foram realizados contou com os seguintes elementos:

- **Máquina** – Laptop Toshiba com processador Intel Celeron de 1.10 GHz e memória RAM de 512 MB.

- **Sistema Operacional** – Microsoft Windows XP Professional, Versão 2002, Service Pack 2.

- **Ferramentas** – Plataforma Java J2SE Development Kit 5.0 Update 6. Ambiente de execução do OpenCOM na versão 1.3.5 para Java.

7.2.2 Interferências do Ambiente nos Experimentos

A análise dos resultados dos experimentos que serão apresentados deve levar em conta as interferências do ambiente no qual eles foram realizados. Estas incluem:

- **Interferências do Sistema Operacional** – Assim como qualquer sistema operacional, o Microsoft Windows possui uma série de processos internos que também utilizam os recursos da máquina. Com isso, a depender dos processos ativos do Windows, um mesmo experimento pode apresentar variações em seus resultados quando executado em momentos diferentes.

- **Interferências das Aplicações Concorrentes** – Como o Microsoft Windows XP é um sistema operacional considerado de propósito geral que se à execução dos mais diversos tipos de aplicação, processos de aplicações instaladas podem também concorrer aos recursos do ambiente. Essa concorrência também pode causar alteração nos resultados dos experimentos.

Considerando essas possíveis interferências, foi tomado um conjunto de medidas para minimizá-las:

- Alguns processos dispensáveis do windows foram desabilitados para reduzir a concorrência com os experimentos.

- Processos de aplicações que estavam programadas para executar na inicialização do sistema foram desabilitados. Por exemplo, processos de inicialização rápida de aplicativos, processos de verificação de atualizações automáticas, processos de inicialização de antivirus e de outros utilitários.

- Cada conjunto de experimentos foi executado três vezes, em momentos diferentes. Dos resultados obtidos foi calculada uma média aritmética da qual foi obtido o resultado final. O objetivo dessa medida foi a de dar mais precisão aos resultados.

7.3 Os Experimentos

Nesta seção serão apresentados os cinco experimentos que foram realizados na plataforma MotherFrame em comparação com o OpenCOM. Esses experimentos serão identificados pelas siglas $E1$, $E2$, $E3$, $E4$ e $E5$ e explicados a seguir:

- ***E1*** – Mede o tempo de instanciação de componentes de tamanho fixo (10 KBytes).
- ***E2*** – Mede o tempo de instanciação de componentes de tamanho variável (de 1 KBytes a 100 KBytes).
- ***E3*** – Mede o tempo para se conectar componentes em slots.
- ***E4*** – Mede o tempo para se ativar ou desativar jumpers.
- ***E5*** – Mede o tempo para se selecionar as conexões de saída de demultiplexadores.

Cada um desses experimentos foi aplicado à MotherFrame e ao OpenCOM. Os experimentos aplicados à MotherFrame foram subdivididos em dois – um para a sua API e outra a forma de configuração via XML. Mais detalhes são descritos abaixo:

MotherFrame API. Os experimentos são realizados utilizando as operações de configuração da API provida pela plataforma MotherFrame através da interface IMotherFrame. Essas operações são: *createComponent*, *setJumperStatus*, *mapCompToSlot* e *setMux*.

MotherFrame XML. Os experimentos são realizados através da submissão de especificações XML que representam MU-Views do modelo motherboard.

OpenCOM. Os experimentos são executados diretamente no OpenCOM. As operações que não existem nativamente no OpenCOM (envolvendo jumpers, slots e demultiplexadores) foram emuladas.

Para a execução de cada experimento, um programa em linguagem Java foi implementado.

7.3.1 E1: Instanciação de Componentes de Tamanho Fixo

Este experimento compara o tempo necessário para se instanciar componentes de 10 KBytes no OpenCOM, no MotherFrame API e no MotherFrame XML.

Preparação do experimento

Para realizar este experimento no OpenCOM foi utilizada a seguinte seqüência de operações:

```
IUnknown cExUnk = (IUnknown) openCOM.createInstance("br.unicamp.CompEx1", "CompEx1");
ILifeCycle cExLife = (ILifeCycle) cExUnk.QueryInterface("OpenCOM.ILifeCycle");
adderLife.startup(openCOM);
...
```

No MotherFrame API foi utilizada a seguinte operação da interface IMotherFrame:

```
mf.createComponent("CompEx1", "br.unicamp.CompEx1");
...
```

No MotherFrame XML foram submetidas especificações XML do seguinte tipo:

```
<configuration id='ConfE1' arch='ArchE1'>
   <component id='CompEx1' source='br.unicamp.CompEx1'/>
   ...
</configuration>
```

Apesar de as operações acima especificarem somente a instanciação de um único componente (CompEx1), os experimentos foram realizados diversas vezes variando o número de componentes instanciados de 1 até 100.

Resultados

O resultado das medições realizadas são mostradas nas Figuras 7.1 e 7.2. Enquanto a segunda figura mostra uma visão mais global dos resultados (de 1 a 100 componentes) a primeira mostra com mais detalhes o custo de instanciar de 1 a 10 componentes.

Primeiramente pode-se observar a diferença entre o OpenCOM e a MotherFrame API na Figura 7.1. Para instanciar um único componente, o MotherFrame API leva o dobro do tempo do que leva o OpenCOM. Porém, essa diferença cai à medida que o número de componentes instanciados aumenta. Isso ocorre porque a instanciação do primeiro componente na MotherFrame API demanda um processamento extra para a criação de um conjunto de estruturas de dados internas. Depois de criadas essas estruturas, o processo de instanciação de componentes se torna mais rápido e mais próximo do OpenCOM.

Ainda na Figura 7.1 pode-se observar que o desempenho da MotherFrame XML é muito menor do que o da MotherFrame API. Tomando-se como exemplo a instanciação de um único componente, o MotherFrame XML leva em média um tempo três vezes maior que o MotherFrame API e seis vezes maior que o OpenCOM. Observa-se também que essa diferença cai quando o número de componentes instanciados aumenta. Essa discrepância

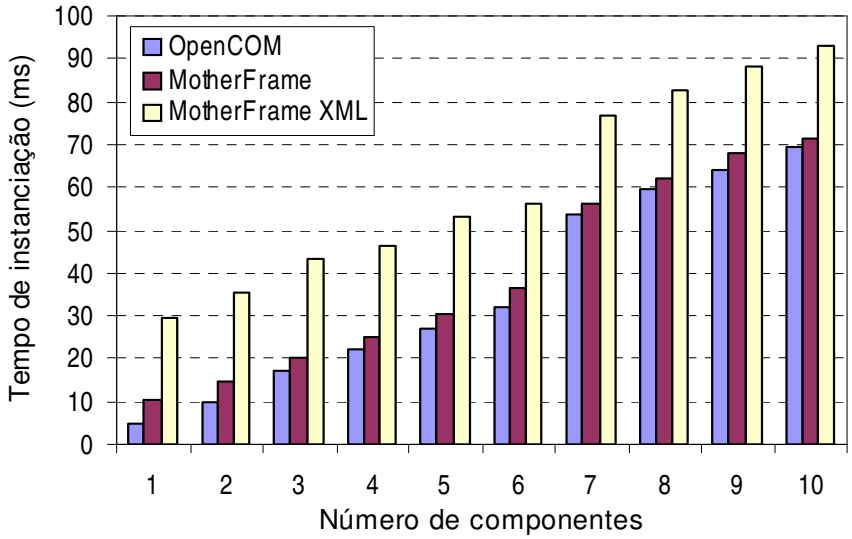

Figura 7.1: Resultados do Experimento E1 (A)

entre a MotherFrame XML e os demais ocorre porque ela exige uma operação de leitura em disco do arquivo XML que contém a especificação dos componentes a serem instanciados. Sabe-se que operação de leitura em disco é um grande elemento impactante no desempenho de sistemas em geral.

A Figura 7.2 mostra o comportamento do MotherFrame quando o número de componentes instanciados vai de 1 a 100. Apesar de, teoricamente, o OpenCOM ter sempre desempenho melhor que o MotherFrame, já que esse requer processamento extra, observa-se que as linhas de desempenho total entre o MotherFrame API e o OpenCOM se tornam cada vez mais próximas ao ponto de entrelaçarem. Isso ocorre por causa dos pequenos erros de medição causados pelas inevitáveis interferências do ambiente descritas anteriormente neste capítulo.

Pode-se observar que, em geral, a diferença entre o desempenho do MotherFrame XML e a do MotherFrame API cresce lentamente. Esse lento crescimento se dá pelo aumento progressivo do tamanho do arquivo XML à medida que o número de componentes a serem instanciados aumenta.

7.3.2 E2: Instanciação de Componentes de Tamanho Variável

Este experimento compara o tempo necessário para se instanciar componentes de diferentes tamanhos que vão de 1 KByte até 100 KBytes.

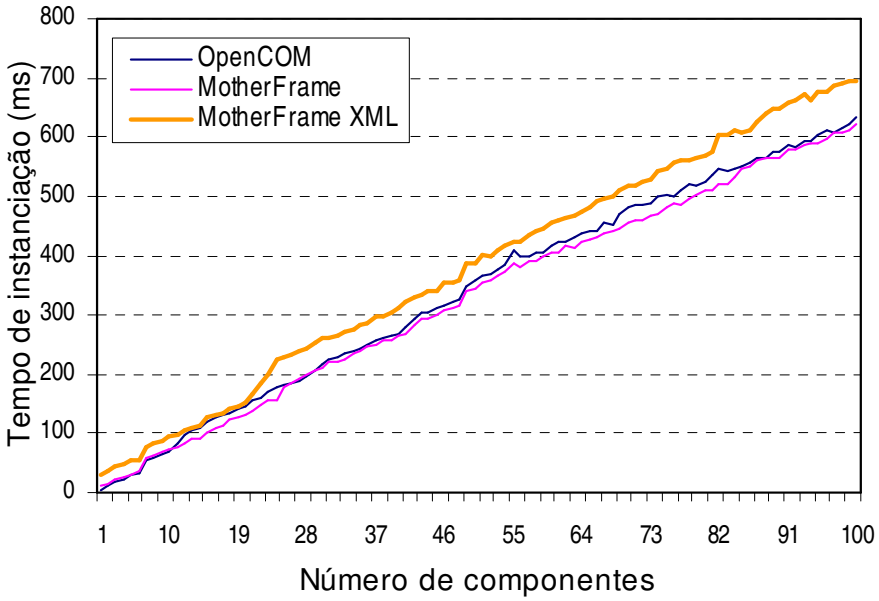

Figura 7.2: Resultados do Experimento E1 (B)

Preparação do experimento

Para realizar este experimento no OpenCOM foi utilizada a seguinte seqüência de operações:

```
IUnknown cExUnk = (IUnknown) openCOM.createInstance("br.unicamp.CompEx1K", "CompEx1K");
ILifeCycle cExLife = (ILifeCycle) cExUnk.QueryInterface("OpenCOM.ILifeCycle");
adderLife.startup(openCOM);
```

No MotherFrame API foi utilizada a seguinte operação da interface *IMotherFrame*:

```
mf.createComponent("CompEx1K", "br.unicamp.CompEx1K");
```

No MotherFrame XML foram submetidas as especificações XML do seguinte tipo:

```
<configuration id='ConfE2' arch='ArchE2'>
   <component id='CompEx1K' source='br.unicamp.CompEx1K'/>
</configuration>
```

Os comandos mostrados acima para o OpenCOM e para o MotherFrame (API e XML) foram executados não só para componentes de 1 KByte (representado por *CompEx1K*) como para componentes de 10, 20, ..., 100 KBytes (os quais seriam representados pelos componentes *CompEx10K*, *CompEx20K*, ..., *CompEx100K*).

Resultados

Os resultados deste experimento são apresentados no gráfico da Figura 7.3. Observa-se que à medida que o tamanho do componente instanciado aumenta, a diferença absoluta entre o OpenCOM, o MotherFrame API e o MotherFrame XML se torna cada vez menor. Isso ocorre porque o tempo que o OpenCOM e a MotherFrame levam para processar um componente x maior do que um componente y só varia porque y demanda mais tempo de leitura em disco do que x.

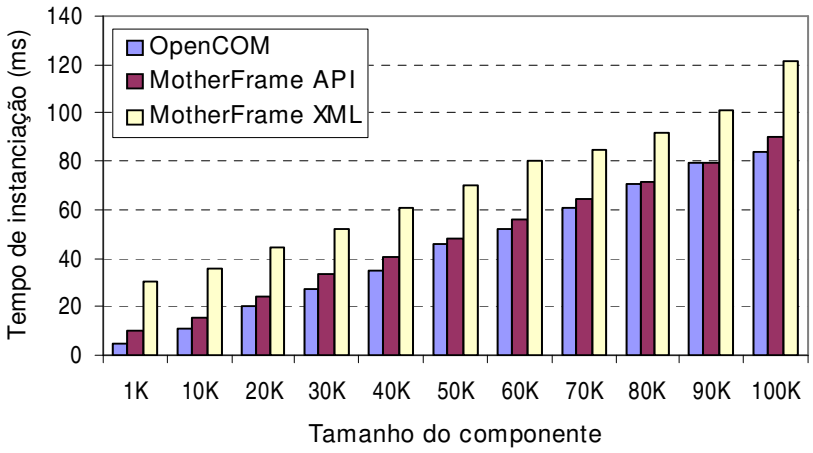

Figura 7.3: Resultados do Experimento E2

7.3.3 E3: Conexão de Componentes em Slots

Este experimento mediu o tempo para se conectar componentes em slots. O número de conexões variou de 1 a 100.

Preparação do experimento

Para realizar este experimento, foi especificada uma arquitetura base do tipo MD-View que contém um slot. É a partir desse slot que foi medido o tempo de conexão de componentes.

A arquitetura base é mostrada na Figura 7.4. Ela representa uma calculadora simplificada que provê operações de multiplicação através do componente $Calc_n$. O componente $Calc_n$ se conecta ao slot $SMult_n$ que, por sua vez, conecta-se ao componente $Adder_n$ (provê operações de adição). No slot $SMult_n$ podem ser conectados componentes que implementam multiplicadores incrementais. Um multiplicador incremental é aquele que

calcula a operação de multiplicação através de operações de soma ($\sum_{i=1}^{x} x$). A operação de soma usada pelo multiplicador é provida pelo componente $Adder_n$.

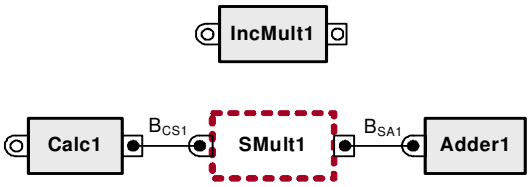

Figura 7.4: Arquitetura Base para o Experimento E3

Essa arquitetura conta com um multiplicador incremental previamente instanciado (representado pelo componente $IncMult_n$). É esse componente que foi conectado ao slot $SMult_n$ durante a execução do experimento.

A conexão do componente $IncMult_n$ ao slot $SMult_n$ foi emulada no OpenCOM através da seguinte seqüência de operações:

```
IUnknown calc1 = openCOM.getComponentPIUnknown("Calc1");
IUnknown adder1 = openCOM.getComponentPIUnknown("Adder1");
IUnknown incMult1 = openCOM.getComponentPIUnknown("IncMult1");
openCOM.connect(calc1, incMult1, "IMultiplier");
openCOM.connect(mult1, adder1, "IAdder");
...
```

No MotherFrame API foi utilizada a seguinte operação da interface IMotherFrame:

```
mf.mapCompToSlot("IncMult1", "SMult1");
...
```

No MotherFrame XML foram submetida especificações XML do seguinte tipo:

```
<configuration id='ConfE3' arch='ArchE3'>
   <slotmap id='SMult1' component='IncMult1'/>
   ...
</configuration>
```

Resultados

Os resultados deste experimento são mostrados nas Figuras 7.5 e 7.6. A primeira mostra os resultados dos testes até 10 conexões em slot e a segunda os estende até 100 conexões.

Pode-se observar que para uma conexão em slot (Figura 7.5), o MotherFrame API leva um tempo que é três vezes superior ao que leva o OpenCOM em uma operação equivalente

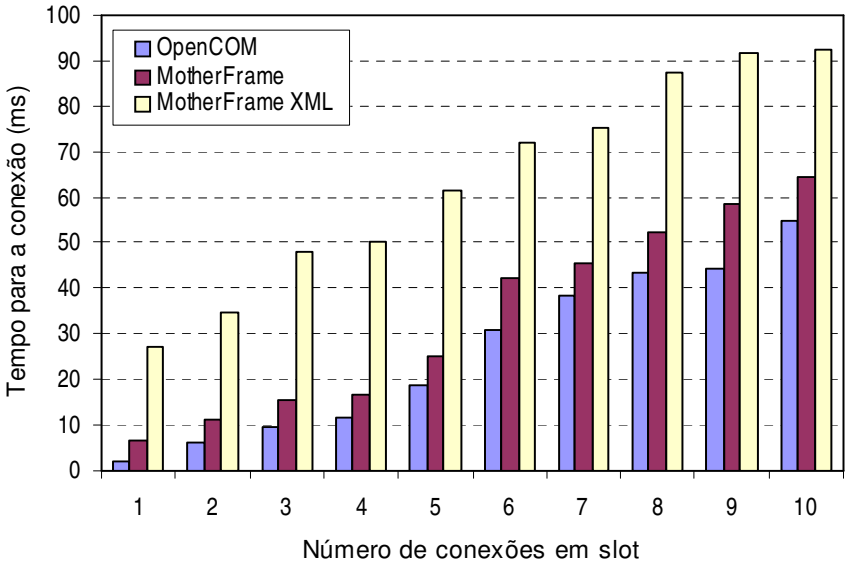

Figura 7.5: Resultados do Experimento E3 (A)

(emulada). Porém, essa diferença diminui à medida que o número de conexões realizadas aumenta.

Observando-se o gráfico da Figura 7.6, as linhas que indicam o desempenho se entrelaçam à medida que o número de conexões aumentam. Isso ocorre porque, apesar de o OpenCOM ter desempenho melhor que o MotherFrame API, as interferências inerentes do ambiente de testes introduzem uma margem de erro aos resultados.

7.3.4 E4: Ativação/Desativação de Jumpers

Este experimento mediu o tempo para se ativar jumpers. O número de ativações variou de 1 a 100.

Preparação do experimento

Para se realizar este experimento, foi especificada uma arquitetura base do tipo MD-View que contém um jumper. É a partir desse jumper que foi medido o tempo de ativação.

A arquitetura base é mostrada na Figura 7.7. Ela representa uma calculadora simplificada que provê operações de multiplicação através do componente $Calc_n$. Para prover essas operações, o componente $Calc_n$ pode utilizar os serviços do componente multiplicador $SimpleMult_n$. Os serviços desse multiplicador só podem ser utilizados quando o jumper $JMult_n$ estiver ativado.

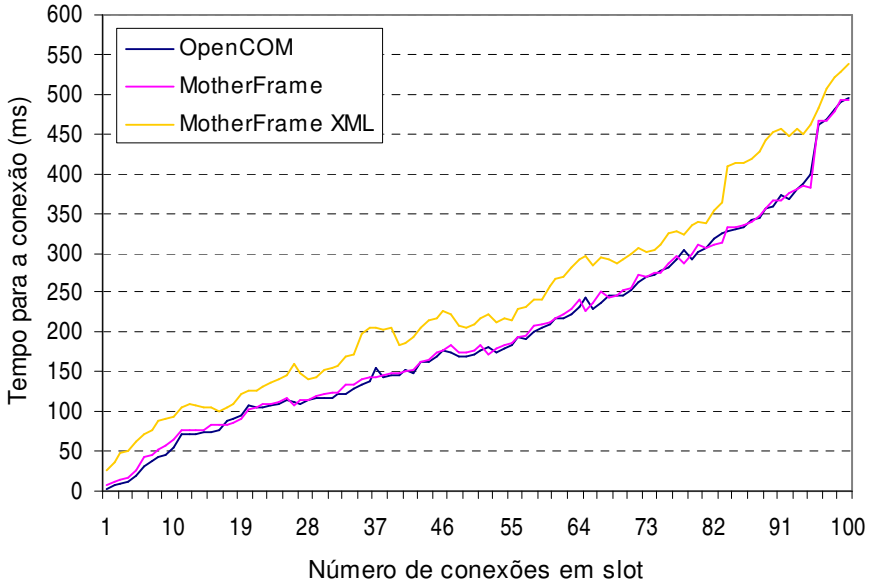

Figura 7.6: Resultados do Experimento E3 (B)

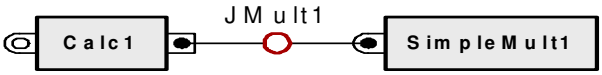

Figura 7.7: Arquitetura Base para o Experimento E4

O processo de ativação do jumper $JMult_n$ foi emulado através da seguinte seqüência de operações:

```
IUnknown calc1 = openCOM.getComponentPIUnknown("Calc1");
IUnknown mult1 = openCOM.getComponentPIUnknown("SimpleMult1");
openCOM.connect(calc1, mult1, "IMultiplier");
...
```

No MotherFrame API foi utilizada a seguinte operação da interface IMotherFrame:

```
mf.setJumperStatus("JMult1", EJumperStatus.ON);
...
```

No MotherFrame XML foram submetidas as especificações XML do seguinte tipo:

```
<configuration id='ConfE4' arch='ArchE4'>
   <jumper id='JMult1' state='ON'/>
   ...
</configuration>
```

Resultados

Os resultados deste experimento são mostrados nas Figuras 7.8 e 7.9. A primeira mostra os resultados dos testes até 10 ativações de jumpers e a segunda os estende até 100 ativações.

Os resultados deste experimento mostram o mesmo padrão daquele obtido nos experimentos anteriores. A diferença entre o OpenCOM e o MotherFrame API se apresenta alta para uma única ativação (Figura 7.8) e reduz à medida que o número de ativações cresce. Por causa da necessária operação de leitura em disco, o MotherFrame XML possui um desempenho bem menor que os demais.

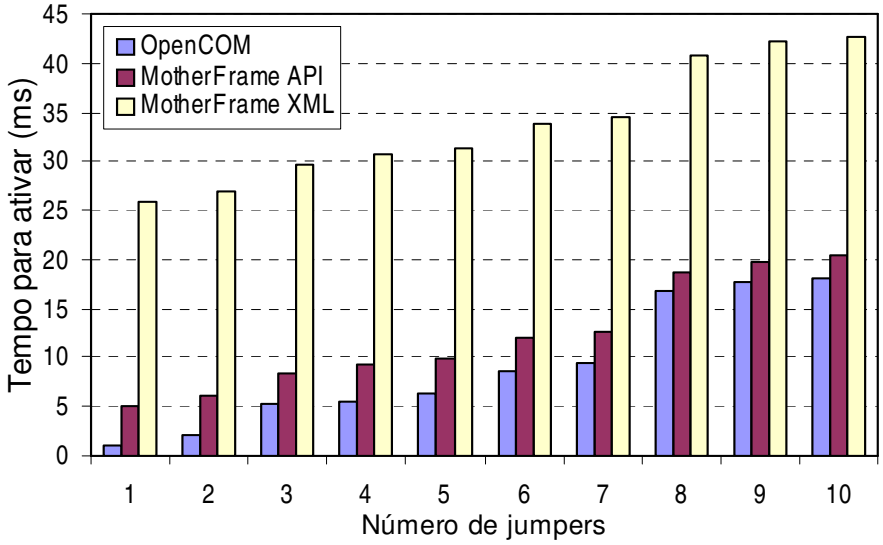

Figura 7.8: Resultados do Experimento E4 (A)

7.3.5 E5: Seleção de Saída de Demultiplexadores

Este experimento mediu o tempo para se selecionar a conexão de saída de demultiplexadores. O número de seleções variou de 1 até 100.

Preparação do experimento

Para se realizar este experimento, foi especificada uma arquitetura base do tipo MD-View que contém um demultiplexador. É a partir desse demultiplexador que foi medido o tempo de seleção da conexão de saída.

A arquitetura base é mostrada na Figura 7.10. Ela representa uma calculadora simplificada que provê operações de multiplicação. Para prover essas operações, o componente

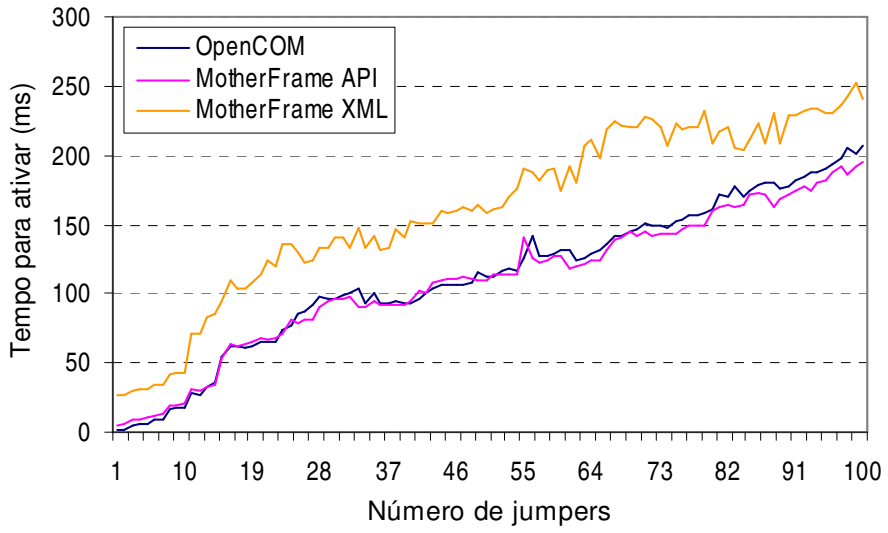

Figura 7.9: Resultados do Experimento E4 (B)

$Calc_n$ pode utilizar os serviços do componente multiplicador simples $SimpleMult_n$ ou do componente multiplicador incremental $IncMult_n$. Para se alternar entre essas duas opções de multiplicação, pode-se usar o demultiplexador D_M. Ao selecionar a saída B_{I1} o multiplicador incremental é utilizado. Ao selecionar a saída B_{S1} o multiplicador simples é utilizado.

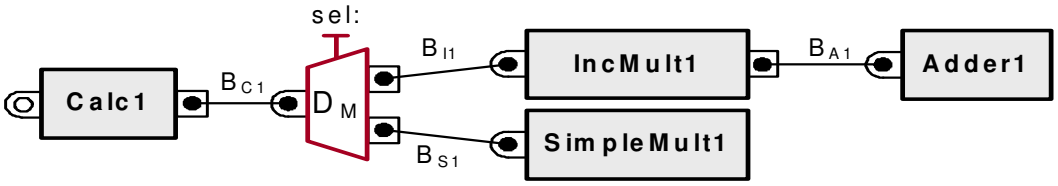

Figura 7.10: Arquitetura Base para o Experimento E5

No OpenCOM, a manipulação do demultiplexador D_M para a seleção da saída B_{I1} foi emulada através da seguinte seqüência de operações:

```
IUnknown calc1 = openCOM.getComponentPIUnknown("Calc1");
IUnknown inc1 = openCOM.getComponentPIUnknown("IncMult1");

Vector<Long> recepConnecIdList = new Vector<Long>();
IMetaArchitecture metaArch = (IMetaArchitecture)
                  openCOM.QueryInterface("OpenCOM.IMetaArchitecture");
metaArch.enumConnsFromRecp(calc1, "IMultiplier", recepConnecIdList);
Long recepConnecId = recepConnecIdList.get(0);
```

```
openCOM.disconnect(recepConnecId);
openCOM.connect(calc1, inc1, "IMultiplier");
...
```

No MotherFrame API foi utilizada a seguinte operação da interface IMotherFrame:

```
mf.setMux("DM", "BI1");
...
```

No MotherFrame XML foram submetida especificações XML do seguinte tipo:

```
<configuration id='ConfE5' architecture='ArchE5'>
   <setmux id='DM' switch='BI1'/>
   ...
</configuration>
```

Os comandos mostrados acima para o OpenCOM e para o MotherFrame (API e XML) foram executados de uma a cem vezes.

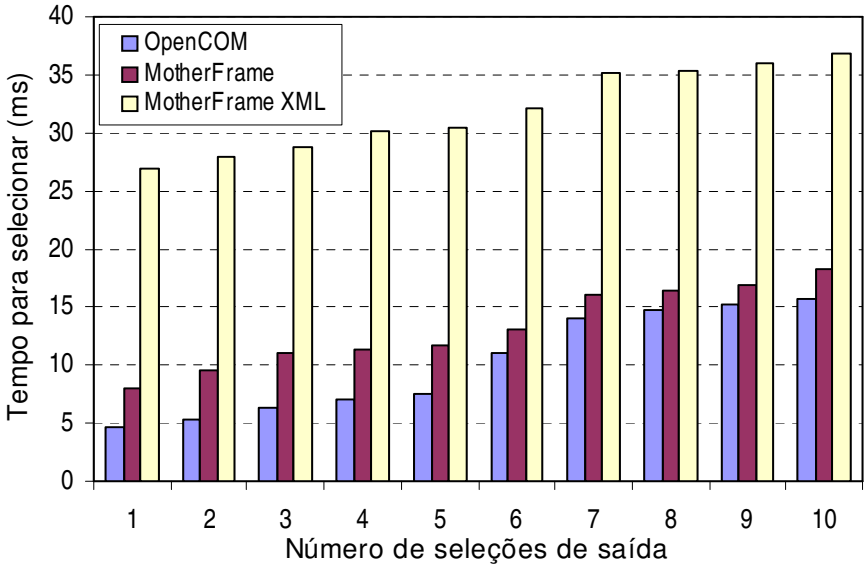

Figura 7.11: Resultados do Experimento E5 (A)

Resultados

Os resultados deste experimento são mostrados nas Figuras 7.11 e 7.12. A primeira mostra os resultados dos testes até 10 seleções de saídas de demultiplexadores e a segunda a estende até 100 seleções.

Os resultados desse experimento se comportaram de maneira similar aos dos demais experimentos. A diferença entre o desempenho total a MotherFrame e do OpenCOM é maior para um menor número de seleções e, à medida que esse número aumenta, essa diferença se torna cada vez menor. Um outro aspecto que pode ser observado neste experimento é que seus resultados se apresentaram levemente inclinados a uma linha exponencial (Figura 7.12).

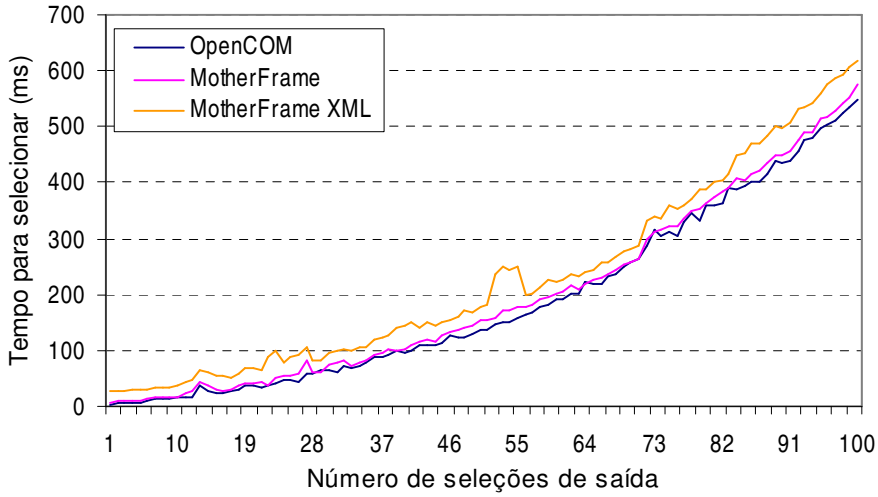

Figura 7.12: Resultados do Experimento E5 (B)

7.3.6 Discussão

Diante dos resultados obtidos com os cinco experimentos apresentados acima, pode-se enfatizar os seguintes pontos:

- O processo de reconfiguração de sistemas especificados na plataforma MotherFrame possui menor desempenho do que emulações do mesmo processo usando somente o OpenCOM. Isso já era um resultado esperado – como a MotherFrame foi implementada como uma nova camada de abstração em cima do OpenCOM, é esperado que aquela demande processamento extra.

- O desempenho da MotherFrame API pode ser considerado bom por estar na casa dos poucos milisegundos. Além disso, a MotherFrame API respondeu bem aos testes de escalabilidade mantendo-se sempre próximo do desempenho do OpenCOM.

- O desempenho do MotherFrame XML, apesar de ser muito inferior ao do OpenCOM e ao da MotherFrame API, é totalmente aceitável para a maioria dos domínios de aplicação, com exceção somente para aqueles domínios nos quais o desempenho das reconfigurações é altamente restritivo e incompatível com a resposta da MotherFrame XML. Essa diferença entre o desempenho da MotherFrame API e da MotherFrame XML é esperado. As especificações XML exigem processamento extra de leitura em disco e extração de informações.

Por causa do seu melhor desempenho, não se pode concluir que a MotherFrame API deve ser sempre preferível à MotherFrame XML. Cada uma possui suas vantagens e limitações e o desempenho é só um dos pontos de comparação. Por exemplo, a MotherFrame XML provê a capacidade de especificar, modificar e submeter configurações XML sem a necessidade de compilar o código.

Capítulo 8

Conclusões

Desenvolver serviços de transações para ambientes dinâmicos pode ser uma tarefa complexa. Nesse processo de desenvolvimento estão envolvidos pelo menos dois grupos de obstáculos: *(i)* lidar com o próprio dinamismo do ambiente e *(ii)* lidar com a grande variedade de requisitos transacionais que podem ser exigidos pelas aplicações interessadas nos serviços. Um agravante nesse processo é que as soluções que devem ser adotadas para esses obstáculos são do domínio das aplicações interessadas. Portanto, elas não podem ser totalmente previstas pelos desenvolvedores dos serviços.

Neste trabalho, essa questão da complexidade é abordada através de um estudo que propõe serviços de transações abertos. A meta principal é a de possibilitar a estruturação de serviços de transações aptos a serem configurados estática ou dinamicamente para atender os requisitos dos domínios das aplicações. Para atingir a meta traçada, foi proposto o modelo *motherboard* que disponibiliza um conjunto de elementos arquiteturais que podem ser usados para especificar pontos de configuração em aberto em serviços de transações. Com base nesse modelo, foi implementada a plataforma MotherFrame que disponibiliza a desenvolvedores e usuários de serviços de transação a capacidade de especificar e configurar seus elementos.

Um aspecto relevante deste trabalho foram as estratégias adotadas para facilitar o processo de configuração de serviços de transação por parte dos seus usuários. Primeiramente os interesses/responsabilidades dos desenvolvedores dos serviços e dos usuários dos serviços foram separados. Enquanto o desenvolvedor define os detalhes da estrutura e do funcionamento do serviço, o usuário se preocupa somente com a configuração dos pontos em aberto definidos pelo desenvolvedor. Essa estratégia poupa o usuário de ter que lidar com detalhes complexos das estruturas dos serviços no processo de configuração. A segunda estratégia foi prover um meio declarativo de configuração via XML. Isso livra o usuário da necessidade de escrever e compilar código em linguagem de programação toda vez que uma nova configuração tiver que ser especificada (como é feito no Open-

COM). Além disso, arquivos de configuração XML possuem melhor legibilidade do que configurações especificadas em linguagem de programação. Outros benefícios podem ser obtidos a partir dessas estratégias: *(i)* para fazer uso dos elementos arquiteturais abertos do modelo motherboard, os desenvolvedores são impelidos a pensar em questões como extensão, configuração e reuso; *(ii)* desenvolvedores de serviços podem documentar em arquivos XML um conjunto de configurações prontas para serem usadas ou modificadas pelos usuários de acordo com os requisitos de suas aplicações.

As principais contribuições desta tese não se limitam ao modelo definido e à plataforma implementada, mas incluem também os serviços de transação abertos apresentados no Capítulo 6. Além de serem elementos de validação da plataforma MotherFrame, esses serviços são contribuições significativas à área de transações para ambientes dinâmicos, que é o enfoque desta tese. O primeiro serviço de transação aberto foi definido a partir da ATP – que foi remodelada para incluir pontos de configuração para critérios de correção, políticas de adaptação e monitoramento de recursos. O segundo serviço de transação aberto foi definido a partir do ACOM – que foi remodelado com pontos de configuração para a definição de protocolos de efetivação personalizados e de novas políticas de adaptação. Para confirmar a capacidade de configuração e de usabilidade desses serviços, eles foram configurados para atender os requisitos reais de duas aplicações diferentes.

Apesar de essa tese ter explorado os requisitos de flexibilização e usabilidade de serviços de transações em ambientes dinâmicos, suas contribuições podem ser aplicadas ao âmbito global de middlewares abertos. Além de serviços de transações, outros serviços abertos de middleware poderiam se beneficiar da abordagem apresentada como, serviços de nomes, serviços de comunicação, serviços de segurança e serviços de persistência.

Por fim, uma das principais metas dessa tese foi a de viabilizar a especificação de serviços de transação capazes de incorporar os requisitos de uma gama maior de aplicações. Ao invés de propor novos serviços de transação para cada novo grupo de requisitos que surgem, a abordagem dessa tese possibilita a especificação de serviços abertos com capacidades claras de configuração, extensão e usabilidade. Com essas capacidades, serviços de transação poderiam ser configurados pelos próprios usuários para atender novos requisitos.

8.1 Trabalhos Futuros

A seguir são relacionados alguns trabalhos futuros que podem ser desenvolvidos a partir dos resultados desta tese:

1. Uma linguagem para a especificação de configurações válidas pode ser criada em XML. O objetivo dessa linguagem seria a de permitir aos desenvolvedores de serviços especificar os tipos de configurações que são compatíveis com os mesmos. Para cada

arquitetura base do tipo MD-View, um conjunto de regras de validação seriam definidas. Toda vez que uma configuração MU-View fosse aplicada a uma arquitetura base MD-View, a compatibilidade da configuração com a arquitetura base seria verificada. O estudo que visa a criação dessa linguagem já está em andamento [82].

2. Uma interface gráfica para plataforma MotherFrame poderia ser implementada. Através dessa interface, a especificação e configuração de arquiteturas abertas poderia ser feita graficamente utilizando a notação gráfica definida na seção 4.4.2. A essa interface poderiam ser agregadas funções como a conversão de especificações XML (MD-View e MU-View) em notação gráfica e vice-versa. O objetivo dessa interface seria a de facilitar o processo de definição e configuração de serviços.

3. Na versão corrente da MotherFrame o processo de configuração é realizado localmente e de modo descentralizado. Para utilizar essa versão em arquiteturas distribuídas, uma instância da plataforma tem que ser instalada em cada ponto distribuído e o processo de configuração tem que ser realizado localmente em cada ponto envolvido. Essa limitação da versão corrente da plataforma tem que ser superada através da implementação de mecanismos de configuração para arquiteturas distribuídas. Com isso, arquiteturas distribuídas poderiam ser configuradas de modo centralizado a partir de um único ponto da rede.

4. Outros serviços de transação para ambientes móveis poderiam ser remodelados na plataforma MotherFrame. Isso daria a esses serviços características que incluem a facilidade de configuração e de extensão. Dentre os serviços que poderiam ser remodelados estão o AMT [90], o HiCoMo [59], o AMT [94] e o ReflecTS [4].

5. Como a plataforma MotherFrame possibilita mudanças na estrutura de serviços em tempo de execução, essas mudanças podem gerar erros ou instabilidade se não forem realizadas com as devidas cautelas. Por exemplo, desativar um jumper entre dois componentes que se encontram no meio de um processo de comunicação pode gerar um erro em tempo de execução. Na versão atual da plataforma, não foram implementados mecanismos que visam garantir a estabilidade dos serviços. A responsabilidade de verificar as condições ideais para se realizar as mudanças em tempo de execução é do desenvolvedor ou do usuário do serviço. Porém, estes mecanismos podem vir a ser implementados na plataforma como um trabalho futuro. Um mecanismo que pode ser implementado seria: antes de realizar uma configuração, todos os componentes envolvidos responderiam a uma mensagem indicando se estão ou não prontos para o processo de configuração.

6. Projetar e implementar uma versão da plataforma MotherFrame para dispositivos

com capacidade de computação reduzida como telefones celulares. Explorar as capacidades computacionais dos celulares tem sido um dos novos desafios de grupos de pesquisa de diversos países.

8.2 Lista de Publicações

A seguir são listadas algumas publicações que foram realizadas como partes dos estudos realizados nesta tese:

(A) T. Rocha and M. Toledo. Estudo de modelos de transação para o ambiente de computação móvel. Relatório Técnico, IC-05-025, Instituto de Computação–UNICAMP, Setembro 2005.

(B) T. Rocha, A. Arntsen, R. Karlsen, and M.B.F. Toledo. Defnition and evolution of transaction systems in emergent environments. In *IADIS International Conference – Wireless Applications and Computing*, Lisboa, Portugal, 2007.

(C) T. Rocha and M.B.F. Toledo. Customizing correctness criteria for transactions in mobile computing environments. In *IADIS International Conference – Wireless Applications and Computing*, Lisboa, Portugal, 2007.

(D) T. Rocha and M.B.F. Toledo. Mecanismos de adaptação para transações em ambientes de computação móvel. *Revista IEEE América Latina*, 5, 2007.

(E) T. Rocha, A. Arntsen, A. Eidsvik, M.B.F. Toledo and R. Karlsen. Promoting Levels of Openness on Component-Based Adaptable Middleware. In *Proceedings of the 6th Workshop on Adaptive and Reflective Middleware*, ACM, Newport Beach, United States, 2007.

(F) T. Rocha, A. Arntsen, R. Karlsen and M.B.F. Toledo. OpenADL – A Simple and Extensible Architecture Description Language, University of Tromso–Norway, Technical Report, 2007.

(G) T. Rocha, M.B.F. Toledo. Managing Complexity in Open Adaptive Middleware. The *2008 International Symposium on Scientific and Engineering Computing*, IEEE, São Paulo, Brazil, 2008.

Referências Bibliográficas

[1] A. Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1999.

[2] Y. Al-Houmaily and P. Chrysanthis. An atomic commit protocol for gigabit-networked distributed database systems. *Journal of Systems Architecture*, 46(9):809–833, 2000.

[3] Y. Al-Houmaily and P. Chrysanthis. 1-2pc: the one-two phase atomic commit protocol. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 684–691, New York, NY, USA, 2004. ACM.

[4] A. Arntsen and R. Karlsen. Dynamic tarnsaction service composition. In *Proceedings of the 2007 IADIS International Conference on Applied Computing*, 2007.

[5] R. Barga and C. Pu. Reflection on a legacy transaction processing monitor. In *Proceedings of Reflection '96*, San Francisco, CA, USA, April 1996.

[6] B. Benyó, A. Vilmos, K. Kovács, and L. Kutor. The design of nfc based applications. In *Proceedings of the 11th International Conference on Intelligent Engineering Systems*, pages 277–281, July 2007.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, Boston, MA, USA, 1987.

[8] G. Blair and G. Coulson. The case for reflective middleware. Technical report, Distributed Multimedia Research Group - Lancaster University, 1998.

[9] G. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, F. Costa, H. Duran-Limon, T. Fitzpatrick, L. Johnston, R. Moreira, N. Parlavantzas, and K. Saikoski. The design and implementation of openorb v2. *IEEE DS Online - Special Issue on Reflective Middleware*, 2(6), 2001.

[10] G. Blair, G. Coulson, L. Blair, H. Duran-Limon, P. Grace, R. Moreira, and N. Parlavantzas. Reflection, self-awareness and self-healing in openorb. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 9–14, New York, NY, USA, 2002. ACM Press.

[11] G. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *Proceedings of the International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer-Verlag, 1998.

[12] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softwware Practice and Expererience*, 36(11-12):1257–1284, 2006.

[13] G. Kaiser C. Pu and N. Hutchinson. Split-transactions for open-ended activities. In *Proceedings of the 14th VLDB Conference*, 1988.

[14] J. Camenisch, A. Lysyanskaya, and M. Meyerovich. Endorsed e-cash. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 101–115, Washington, DC, USA, 2007. IEEE Computer Society.

[15] D. Chaum. Blind signaturs for untraceable payments. In *Proceedings of Crypto 82*, pages 199–203. New York: Plenum Press, 1983.

[16] P. Chrysanthis. Transaction processing in mobile computing environment. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 77–83, Princeton, New Jersey, 1993.

[17] P. Chrysanthis and K. Ramamritham. Acta: a framework for specifying and reasoning about transaction structure and behavior. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 194–203, New York, NY, USA, 1990. ACM Press.

[18] P. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using acta. *ACM Trans. Database Syst.*, 19(3):450–491, 1994.

[19] M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Middleware '01: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 160–178, London, UK, 2001. Springer-Verlag.

[20] X/Open Company. Distributed transaction specification: The xa specification, December 1991.

[21] X/Open Company. Distributed transaction specification: The tx (transaction demarcation) specification, April 1995.

[22] X/Open Company. Distributed transaction specification: Reference model, version 3, February 1996.

[23] Microsoft Corporation. The component object model specification - version 0.9. Technical report, October 1995.

[24] Microsoft Corporation. The .net framework. Technical report, 2000.

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

[26] F. DeRemer and H. Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, New York, NY, USA, 1975. ACM Press.

[27] M. Dideles. Bluetooth: a technical overview. *Crossroads*, 9(4):11–18, 2003.

[28] E. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[29] R. Dirckze and L. Gruenwald. A toggle transaction management technique for mobile multidatabases. In *In Proceedings of the Seventh International Conference on Information and Knowledge Management*, pages 371–377, 1998.

[30] R. Dirckze and L. Gruenwald. A pre-serialization transaction management technique for mobile multidatabases. *MONET–Mobile Networks and Applications*, 5:311–321, 2000.

[31] M. Dunham and A. Helal. Mobile computing and databases: anything new? *SIGMOD Rec.*, 24(4):5–9, 1995.

[32] M. Dunham, A. Helal, and S. Balakrishnan. A mobile transaction model that captures both the data and movement behavior. *Mob. Netw. Appl.*, 2(2):149–162, 1997.

[33] J. Ferber. Computational reflection in class based object-oriented languages. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 317–326, New York, NY, USA, 1989. ACM Press.

[34] G. Forman and J. Zahorjan. The challenges of mobile computing. *Computer*, 27(4):38–47, 1994.

[35] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of the 17th international conference on Software engineering*, pages 179–185, New York, NY, USA, 1995. ACM.

[36] J. Gorman. The midas project: Interworking and data sharing. In *Proceedings of the Interworking 2006*, 2007.

[37] P. Grace, G. Blair, and S. Samuel. A marriage of web services and reflective middleware to solve the problem of mobile client interoperability. In *ISICT '03: Proceedings of the 1st international symposium on Information and communication technologies*, pages 506–511. Trinity College Dublin, 2003.

[38] P. Grace, G. Blair, and S. Samuel. A reflective framework for discovery and interaction in heterogeneous mobile environments. *SIGMOBILE Mob. Comput. Commun. Rev.*, 9(1):2–14, 2005.

[39] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP Working Conference on Modelling in Data Base Management Systems*, pages 365–394, 1976.

[40] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[41] Object Management Group. Transaction service specification - version 1.4, September 2003.

[42] Object Management Group. Corba component model - v30, 2004.

[43] M. Head and M. Govindaraju. Approaching a parallelized xml parser optimized for multi-coreprocessors. In *SOCP '07: Proceedings of the 2007 workshop on Service-oriented computing performance: aspects, issues, and approaches*, pages 17–22, New York, NY, USA, 2007. ACM.

[44] A. Helal, D. Woelk, B. Haskell, J. Carter, R. Brice, and Rusin. *Any Time, Anywhere Computing: Mobile Computing Concepts and Technology*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[45] H. Huomo. Emerging solutions technology and business views for the ubiquitous communication. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 678–678, San Jose, CA, USA, 2007. EDA Consortium.

[46] A. Jakobsen and R. Karlsen. Reflects: a flexible transaction service framework. In *ARM '05: Proceedings of the 4th workshop on Reflective and adaptive middleware systems*, New York, NY, USA, 2005. ACM Press.

[47] Sixto Ortiz Jr. Is near-field communication close to success? *Computer*, 39(3):18–20, 2006.

[48] R. Karlsen. An adaptive transactional system - framework and service synchronization. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA)*, Catania, Sicily, November 2003.

[49] R. Karlsen and A. Jakobsen. Transaction service management - an approach towards a reflective transactional service. In *Proceedings of the 2nd Workshop on Reflective and Adaptive Middleware*, Rio de Janeiro, Brazil, 2003.

[50] R. Katz. Adaptation and mobility in wireless information systems. *IEEE Personal Communications*, 1:6–17, 1995.

[51] G. Kiczales, J. Lamping, C. Lopes, C. Maeda, A. Mendhekar, and G. Murphy. Open implementation design guidelines. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 481–490, New York, NY, USA, 1997. ACM Press.

[52] F. Kon, F. Costa, G. Blair, and R. Campbell. The case for reflective middleware. *Commun. ACM*, 45(6):33–38, 2002.

[53] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. Magalhães, and R. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.

[54] F. Kordon and L. Pautet. Toward nex-generation middleware? *IEEE Distributed Systems Online*, 6(3):2, 2005.

[55] K. Ku and Y. Kim. Moflex transaction model for mobile heterogeneous multidatabase systems. In *In IEEE Workshop on Research Issues in Data Engineering*, San Diego, USA, 2000.

[56] B. Lampson and D. Lomet. A new presumed commit optimization for two phase commit. In *Proceedings of the 19th International Conference on Very Large Data Bases*, pages 630–640, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[57] R. Lea, Y. Yokote, and J. Itoh. Adaptive operating system design using reflection. In *HOTOS '95: Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, page 95, Washington, DC, USA, 1995. IEEE Computer Society.

[58] T. Ledoux. Opencorba: A reflective open broker. *Proceedings of the 2nd International Conference on Reflection '99*, 1616:197–??, 1999.

[59] M. Lee and S. Helal. Hicomo: High commit mobile transactions. *Distrib. Parallel Databases*, 11(1):73–92, 2002.

[60] M. Lotia and P. Nair. *All About Motherboard: Complete Introduction and Troubleshooting*. BPB Publications, 2002.

[61] Q. Lu and M. Satyanaranyanan. Isolation-only transactions for mobile computing. *SIGOPS Oper. Syst. Rev.*, 28(2):81–87, 1994.

[62] S. Lu, A. Bernstein, and P. Lewis. Correct execution of transactions at different isolation levels. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1070–1081, 2004.

[63] S. Madria and B. Bhargava. A transaction model to improve data availability in mobile computing. *Distributed Parallel Databases*, 10(2):127–160, 2001.

[64] P. Maes. Concepts and experiments in computational reflection. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 147–155, New York, NY, USA, 1987. ACM Press.

[65] J. Malenfant, M. Jacques, and F. Demers. A tutorial on behavioral reflection and its implementation. In *Proceedings of the Reflection '96 Conference*, San Francisco, CA, April 1996.

[66] M. McIlroy. Mass produced software components. In *Proceedings of NATO Software Engineering Conference*, volume 1, pages 138–150, Garmisch, Germany, October 1968.

[67] T. Meijler and O. Nierstrasz. Beyond objects: Components. In *In WCOP'98 Proceedings of the Third International Workshop on Component-Oriented Programming*, 1998.

[68] Sun Microsystems. Java transaction api (jta) version 1.0.1, April 1999.

[69] Sun Microsystems. Java transaction service (jts) version 1.0, December 1999.

[70] Sun Microsystems. Enterprise javabeans specification version 2.0, 2001.

[71] J. Moss. *Nested transactions: an approach to reliable distributed computing*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1985.

[72] L. Mummert, M. Ebling, and M. Satyanarayanan. Exploring weak connec-tivity for mobile file access. In *In Proceedings of the 15th ACM Symposium on Opera-tion Systems Principles*, Colorado, 1998.

[73] G. S. Blair P. Grace and S. Samuel. Remmoc: A reflective middleware to support mobile client interoperability. In *Proceedings of the Int. Symposium on Distributed Objects and Applications (DOA 2003)*, 2003.

[74] S. Y. Pike and P. Osborne. Wi-fi and handhelds: perfect synergy. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 1004–1018, New York, NY, USA, 2004. ACM.

[75] E. Pitoura and G. Samaras. *Data Management for Mobile Computing*, volume 10. Kluwer Academic Publishers, 1998.

[76] T. Plagemann, V. Goebel, C. Griwodz, and P. Halvorsen. Towards middleware services for mobile ad-hoc network applications. In *Proceedings of the The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems*, page 249, Washington, DC, USA, 2003. IEEE Computer Society.

[77] N. Preguiça, C. Baquero, F. Moura, J. Martins, R. Oliveira, H. Domingos, J. Pereira, and S. Duarte. Mobile transaction management in mobisnap. In *ADBIS-DASFAA '00: Proceedings of the East-European Conference on Advances in Databases and Information Systems Held Jointly with International Conference on Database Systems for Advanced Applications*, pages 379–386, London, UK, 2000. Springer-Verlag.

[78] M. Prochazka. Advanced transactions in enterprise javabeans. In *EDO '00: Revised Papers from the Second International Workshop on Engineering Distributed Objects*, pages 215–230, London, UK, 2001. Springer-Verlag.

[79] H. Ramampiaro and M. Nygård. Cagistrans: Providing adaptable transactional support for cooperative work - an extended treatment. *Inf. Tech. and Management*, 5(1-2):23–64, 2004.

[80] K. Ramamritham and P. Chrysanthis. *Executive Briefing: Advances in Concurrency Control and Transaction Processing*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[81] T. Rocha, A. Arntsen, A. Eidsvik, M.B.F. Toledo, and R. Karlsen. Promoting levels of openness on component-based adaptable middleware. In *Proceedings of the 6th Workshop on Adaptive and Reflective Middleware*, Newport Beach, United States, 2007.

[82] T. Rocha, A. Arntsen, R. Karlsen, and M.B.F. Toledo. Definition and evolution of transaction systems in emergent environments. In *IADIS International Conference – Wireless Applications and Computing*, Lisboa, Portugal, 2007.

[83] T. Rocha and M. Toledo. Estudo de modelos de transação para o ambiente de computação móvel. Technical Report IC-05-025, Instituto de Computação - UNICAMP, Setembro 2005.

[84] T. Rocha and M.B.F. Toledo. Customizing correctness criteria for transactions in mobile computing environments. In *IADIS International COnference – Wireless Applications and Computing*, Lisboa, Portugal, 2007.

[85] T. Rocha and M.B.F. Toledo. Mecanismos de adaptação para transações em ambientes de computação móvel. *Revista IEEE América Latina*, 5, 2007.

[86] T. Rocha and M.B.F. Toledo. Managing complexity in open adaptive middleware. In *Proceedings of the 2008 International Symposium on Scientific and Engineering Computing*, São Paulo, Brazil, 2008.

[87] M. Roman, F. Kon, and R. Campbell. Design and implementation of runtime reflection in communication middleware: the dynamicTAO case, May 1999. In Workshop on Middleware, ICDCS'99, Austin, Texas.

[88] J. Ruiz, M. Killijian, J. Fabre, and P. Thévenod-Fosse. Reflective fault-tolerant systems: From experience to challenges. *IEEE Trans. Comput.*, 52(2):237–254, 2003.

[89] N. Sanderson, K. Skjelsvik, O. Drugan, M. Puzar, V. Goebel, E. Munthe-Kaas, and T. Plagemann. Developing mobile middleware – an analysis of rescue and emergency operations. Technical Report Research Report no.358, Department of informatics, University of Oslo, Norway, June 2007.

[90] N. Santos, L. Veiga, and P. Ferreira. Transaction policies for mobile networks. In *POLICY '04: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04)*, page 55, Washington, DC, USA, 2004. IEEE Computer Society.

[91] M. Satyanarayanan. Fundamental challenges in mobile computing. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 1–7, New York, NY, USA, 1996. ACM Press.

[92] M. Satyanarayanan. The evolution of coda. *ACM Trans. Comput. Syst.*, 20(2):85–124, 2002.

[93] B. Schooley, M. Marich, and T. Horan. Devising an architecture for time-critical information services: inter-organizational performance data components for emergency medical service (ems). In *Proceedings of the 8th Annual International Conference on Digital Government Research*, pages 164–172. Digital Government Research Center, 2007.

[94] P. Serrano-Alvarado, C. Roncancio, M. Adiba, and C. Labbé. Adaptable mobile transactions. In *Proceedings of the 29th VLDB Conference*, Berlin, Germany, 2003.

[95] P. Serrano-Alvarado, R. Rouvoy, and P. Merle. Self-adaptive component-based transaction commit management. In *Proceedings of the 4th Workshop on Reflective and Adaptive Middleware*, New York, NY, USA, 2005. ACM.

[96] F. Silva, M. Endler, and F. Kon. A framework for building adaptive distributed applications. In *Proceedings of Middleware 2003 - Workshop on Reflective and Adaptive Middleware Systems*, Rio de Janeiro, Brazil, June 2003.

[97] A. Singhai. *Quarterware: A Middleware Tollkit of Software Risc Components*. PhD thesis, University of Illinois at Urbana-Champaign, 1999.

[98] A. Singhai, A. Sane, and R. Campbell. Quarterware for middleware. In *ICDCS '98: Proceedings of the The 18th International Conference on Distributed Computing Systems*, page 192, Washington, DC, USA, 1998. IEEE Computer Society.

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

[100] B. Smith. Reflection and semantics in lisp. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35, New York, NY, USA, 1984. ACM Press.

[101] C. Szyperski. *Component Software*. ACM Press, 1998.

[102] S. Venkatraman. Mobile computing models - are they meeting the mobile computing challenges? *Association for Computing Machinery New Zealand Bulletin*, 1(1), 2005.

[103] G. Walborn and P. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Symposium on Reliable Distributed Systems*, pages 31–40, 1995.

[104] G. Walborn and P. Chrysanthis. Transaction processing in pro-motion. In *SAC '99: Proceedings of the 1999 ACM symposium on Applied computing*, pages 389–398, New York, NY, USA, 1999. ACM Press.

[105] L. Wen-yuan, L. Yong-an, S. Ya-li, W. Bao-wen, and L. Feng. An off-line divisible e-cash scheme based on smart card. *The Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, 01:799–804, 2007.

[106] Z. Wu. Reflective java and a reflective component-based transaction architecture". In *Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in Java and C++*, October 1998.

[107] Y. Yokote. The apertos reflective operating system: the concept and its implementation. In *OOPSLA '92: conference proceedings on Object-oriented programming systems, languages, and applications*, pages 414–434, New York, NY, USA, 1992. ACM Press.

[108] A. Zarras and V. Issarny. A framework for systematic synthesis of transactional middleware. In *Proceedings of MIDDLEWARE'98 - IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 257–272, 1998.

.

Apêndice A

O Modelo OpenCOM

O OpenCOM é um modelo de componentes de baixo peso projetado para o desenvolvimento de middlewares em dispositivos de computação com poucos recursos (processamento, memória, armazenamento). Além de ser um modelo de baixo peso, o OpenCOM provê a capacidade de reconfiguração dinâmica de middlewares tanto no domínio estrutural quanto no comportamental. Para isso, o OpenCOM foi projetado com base nos seguintes elementos: componentes, reflexão computacional e frameworks de componentes.

A.1 Elementos Básicos do Modelo

O modelo OpenCOM permite a especificação da estrutura de sistemas através do uso de componentes e conexões entre componentes (Figura A.1). Cada componente pode possuir um conjunto de interfaces (ou *interfaces providas*) – através das quais o componente exporta seus serviços – e um conjunto de receptáculos (ou interfaces requeridas) – através dos quais o componente requisita serviços de outros componentes. Para que um componente X utilize os serviços de um outro componente Y, um receptáculo de X deve ser conectado a uma interface de Y.

A.2 O Ambiente de Execução

O ambiente de execução do OpenCOM permite a execução de sistemas baseados em componentes. Esse ambiente (Figura A.2) é composto pelo *OpenCOM Runtime* – que processa e armazena informações sobre sistemas implementados com componentes OpenCOM – e o *System Graph* – onde os componentes de sistemas são mantidos e executados.

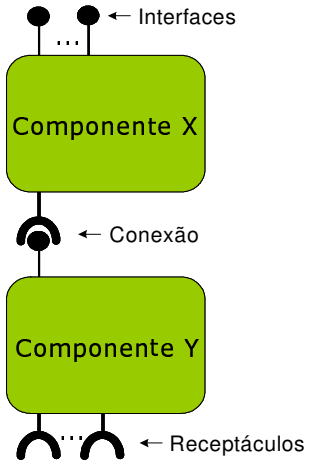

Figura A.1: Elementos Básicos do Modelo OpenCOM

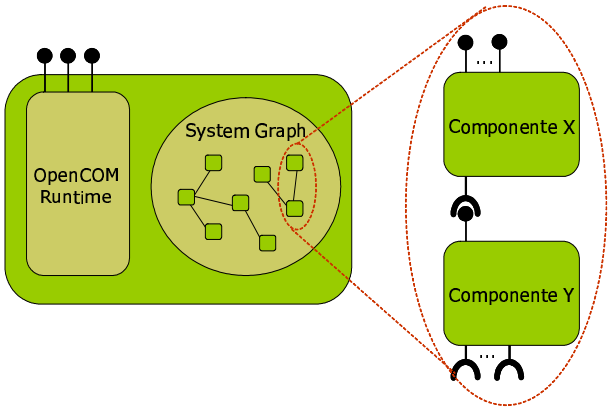

Figura A.2: O Ambiente de Execução do OpenCOM

A.3 Os Metamodelos

A reflexão computacional no OpenCOM é dada através de três metamodelos distintos (Figura A.3): (i) o metamodelo de interface – definido pela interface *IMetaInterface*, (ii) o metamodelo de arquitetura – definido pela interface *IMetaArchitecture* e (iii) o metamodelo de comportamento – definido pela interface *IMetaInterception*.

A.3.1 Metamodelo de Interface

O metamodelo de interface, representado pela interface *IMetaInterface*, provê acesso à representação externa de um componente OpenCOM no que diz respeito às suas interfaces e receptáculos. Através desse metamodelo, é possível: (i) listar todas as interfaces de um componente – operação *enumIntfs*; (ii) listar todos os receptáculos de um componente

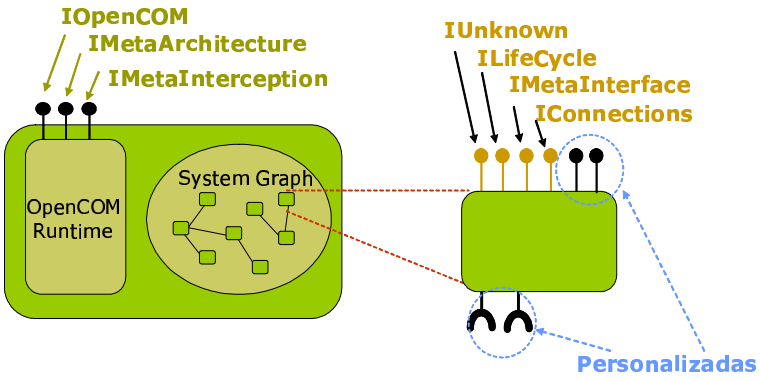

Figura A.3: Interfaces dos Metamodelos do OpenCOM

– operação *enumRecps*; (iii) atribuir metainformações a uma interface/receptáculos – operação *SetAttributeValue*; (iv) acessar metainformações de uma interface/receptáculo – operação *GetAttributeValue*; (v) recuperar todo o conjunto de metainformações de uma interface/receptáculo.

```
public interface IMetaInterface {
  public int enumRecps(Vector<OCM_RecpMetaInfo_t> ppRecpMetaInfo);
  public int enumIntfs(Vector<Class> ppIntf);
  public boolean SetAttributeValue(String iid, String Kind,
                                   String Name, String Type,
                                   Object Value);
  public TypedAttribute GetAttributeValue(String iid,
                                          String Kind,
                                          String Name);
  public Hashtable<String, TypedAttribute> GetAllValues(String Kind,
                                                        String iid);
}
```

A.3.2 Metamodelo de Arquitetura

O metamodelo de arquitetura, representado pela interface *IMetaArchitecture*, permite acesso a informações de arquiteturas mantidas no *System Graph* do OpenCOM. Através dela é possível: (i) acessar os identificadores de todas as conexões que envolvem uma dada interface de um componente específico – operação *enumConnsToIntf*; (ii) acessar os identificadores de todas as conexões que envolvem um dado receptáculo de um componente específico – operação *enumConnsFromRecp*.

```
public interface IMetaArchitecture {
  public int enumConnsToIntf(IUnknown pIUnknown,
                             String riid,
                             Vector<Long> ppConnsToIntf);
  public int enumConnsFromRecp(IUnknown pIUnknown,
                               String riid,
                               Vector<Long> ppConnsFromRecp);
}
```

A.3.3 Metamodelo de Comportamento

O metamodelo de comportamento, representado pelas interfaces *IMetaInterception* e *IDelegator*, permite alteração no comportamento de operações de uma dada interface de um componente específico. Para isso, deve-se primeiramente obter uma referência do tipo *IDelegator* da interface desejada de um componente através da operação *GetDelegator*.

```
public interface IMetaInterception {
  IDelegator GetDelegator(IUnknown pIUnkParent, String riid);

}
```

A partir da interface *IDelegator* podem-se realizar operações como: (i) adicionar um método de pré-interceptação (operação *addPreMethod*) – todas as operações da interface interceptada passam pelo método adicionado antes de serem executadas; (ii) adicionar um método de pós-interceptação (operação *addPostMethod*) – todas as operações da interface interceptada passam pelo método adicionado depois de serem executadas. Os métodos de pré e pós-interceptação também podem ser removidos ou visualizados através das operações *delPreMethod*, *delPostMethod*, *viewPreMethods* e *viewPostMethods*, respectivamente.

```
public interface IDelegator {
  public boolean addPreMethod(Object methodHost,
                              String methodName);
  public boolean delPreMethod(String methodName);
  public boolean addPostMethod(Object methodHost,
                               String methodName);
  public boolean delPostMethod(String methodName);
  public long viewPreMethods(String[] methodNames);
  public long viewPostMethods(String[] methodNames);
}
```

A.4 Frameworks de Componentes

O OpenCOM também permite a criação de frameworks de componentes. Um framework de componentes é definido no OpenCOM como uma sub-arquitetura de componentes sobre a qual pode ser checado um conjunto de propriedades arquiteturais desejadas. Essa sub-arquitetura também pode ser vista como um componente complexo que é composto de outros subcomponentes interconectados.

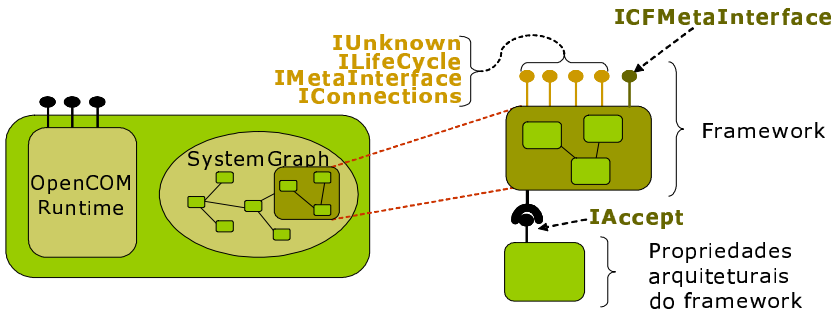

Figura A.4: Frameworks de Componentes

A representação gráfica de um framework de componentes no OpenCOM é mostrada na Figura A.4. Pode-se notar que um framework possui um receptáculo conectado a um componente. É esse componente que implementa as regras capazes de impor as propriedades arquiteturais desejadas para a arquitetura interna do framework.

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: (i) a criação de um componente interno (operação *create_component*); (ii) a exposição de interfaces e receptáculos (operações *expose_interface* e *expose_receptacle*) – faz com que a interface/receptáculo de um componente interno passe a ser a interface/receptáculo do framework. As operações de configuração de um framework sempre devem ser chamadas dentro do escopo de uma transação, isto é, devem ser chamadas entre as operações transacionais *init_arch_transaction* e *commit_arch_transaction*.

As operações de inspeção de um framework incluem: (i) a recuperação de componentes internos (operação *get_internal_components*); (ii) a recuperação da lista de conexões de um dado componente (operação *get_bound_components*); (iii) a recuperação da lista de interfaces e da lista de receptáculos de componentes internos (operação *get_exposed_interfaces*, *get_exposed_receptacles*).

```
public interface ICFMetaInterface{
  // Operações para configuração
  long local_bind( IUnknown pIUnkSource, IUnknown pIUnkSink,
                   String InterfaceType);
  boolean break_local_bind( long connID);
  IUnknown create_component(String className, String componentName);
  boolean insert_component(IUnknown pCompReference);
  boolean delete_component(IUnknown pIUnknown);
  boolean expose_interface(String rintf, IUnknown pComp);
  boolean unexpose_interface(String rintf,  IUnknown pComp);
  boolean unexpose_all_interfaces();
  boolean expose_receptacle(String rintf,  IUnknown pComp,
                            String recpType);
  boolean unexpose_receptacle(String rintf,  IUnknown pComp);
  boolean unexpose_all_receptacles();
  boolean init_arch_transaction();
  boolean commit_arch_transaction();
  boolean rollback_arch_transaction();

  // Operações para inspeção
  boolean find_component(IUnknown pComp);
  int get_internal_components(Vector<IUnknown> ppComps);
  IUnknown getComponentPIUnknown(String CompName);
  String getComponentName(IUnknown CompIUnk);
  int get_bound_components(IUnknown comp,
          Vector<CFMetaInterface.ConnectedComponent> ppConnections);
  int get_internal_bindings(Vector<Long> pConnIDS);
  int get_exposed_interfaces(Vector<String> ppIntfs);
  int get_exposed_receptacles(
                Vector<CFMetaInterface.ExposedReceptacle> ppComps);
}
```

A.5 Tipos de Receptáculos

Um receptáculo de um dado componente OpenCOM pode ser enquadrado em duas principais categorias: receptáculo simples ou receptáculo múltiplo.

(1) **Receptáculo Simples.** Aquele que só aceita a conexão de uma interface de cada vez (Figura A.5(a)).

(2) **Receptáculo Múltiplo.** Aquele que aceita a conexão de mais de uma interface ao mesmo tempo (Figura A.5(b)). Um receptáculo múltiplo pode ser de três tipos:

Sem contexto. As interfaces conectadas no receptáculo são tratadas sem distinção. O componente que possui o receptáculo sem contexto visualiza as interfaces conectadas como um vetor que vai de 0 a $n-1$ (onde n é o número de interfaces conectadas).

Com contexto. As interfaces conectadas no receptáculo podem ser diferenciadas através de informações de contexto. Somente a interface que contém o contexto desejado é chamada.

Paralelo. Todas as interfaces conectadas no receptáculo são chamadas em paralelo toda vez que uma operação é requisitada no receptáculo.

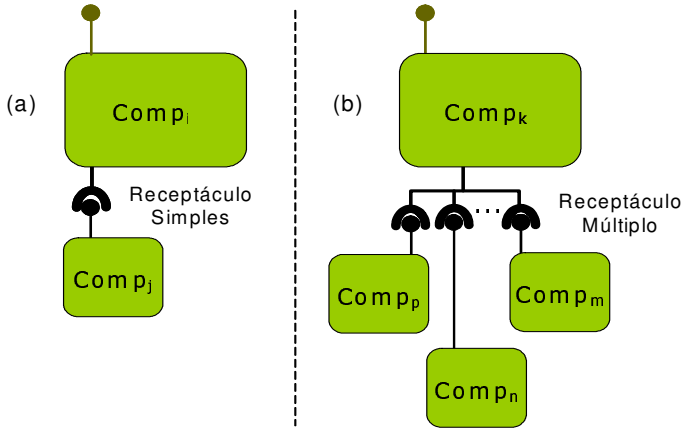

Figura A.5: Tipos de Receptáculos

A.6 O Ambiente de Execução do OpenCOM

As funcionalidades do ambiente de execução do OpenCOM são acessadas através da interface *IOpenCOM*. As principais operações da interface *IOpenCOM* são apresentadas abaixo. Os comentários das operações descreve suas funcionalidades e seus parâmetros.

```
public interface IOpenCOM extends IUnknown{

  /**
   * Create a new instance of a component and insert it into the
   * OpenCOM runtime.
   * @param componentType the string describing the component type
```

```
     *          i.e. the Java class of the component.
     * @param componentName the string represting the unique (user
     *          defined) name of the component.
     * @return an Object that is the reference to the created
     *           component. Null indicates failure.
     **/
    Object createInstance(String componentType, String componentName);

    /**
     * Deletes a component instance which has been previously created.
     * @param pComponentIUnknown an IUnknown reference of the component
     *          to delete.
     * @return Indication of success or failure of operation.
     **/
    boolean deleteInstance(IUnknown pComponentIUnknown);

    /**
     * Connects a receptacle on the Source component to an interface
     * on the Sink component.
     * @param pSourceComponentIUnk Reference to component with the
     *          receptacle.
     * @param pSinkComponentIUnk Reference to component with the
     *          interface.
     * @param InterfaceType a string describing the interface type
     *          of the connection.
     * @return a long describing the unique connection identifier
     *           generated by the run-time.
     **/
    long connect(IUnknown pSourceComponentIUnk,
                 IUnknown ComponentIUnk, String InterfaceType);

    /**
     * Disconnects a receptacle from an interface.
     * @param connID a long describing the unique identifier of
     *          the connection to destroy.
     * @return Indication of success or failure of operation.
     */
    boolean disconnect(long connID);

    /**
     * Fills the given Vector with the complete set of components
```

```
    * currently instantiated in the runtime.
    * @param ppComps Vector to fill with IUnknown pointers of
    *        current components.
    * @return an integer describing the number of components
    *         currently in the run-time.
    */
   int enumComponents(Vector<IUnknown> ppComps);
   ...
}
```

Segue abaixo exemplos práticos de uso de algumas das operações da interface *IOpenCOM*.

Instanciação do OpenCOM

```
IOpenCOM openCOM = new OpenCOM();
```

Criação e Ativação de Componentes

```
IUnknown compRef = (IUnknown) openCOM.createInstance("InterfaceName", "compId");
ILifeCycle compLife = (ILifeCycle) compRef.QueryInterface("OpenCOM.ILifeCycle");
compLife.startup(openCOM);
```

Exclusão de Instâncias de Componentes

```
openCOM.deleteInstance(compRef);
```

Criação de Conexão entre Dois Componentes

```
long connectionId = openCOM.connect(comp1Ref, comp2Ref, "InterfaceName");
```

Destruir a Conexão entre Dois Componentes

```
openCOM.disconnect(connectionId);
```

Localizar Referências de Componentes

```
IUnknown compRef = openCOM.getComponentPIUnknown("compId");
```

A.7 Exemplos de Componentes OpenCOM

Nesta seção será apresentado um exemplo simples de implementação e uso de componentes OpenCOM. Este exemplo será baseado em dois componentes que implementam as funcionalidades de um multiplicador incremental.

Para este exemplo serão implementados dois componentes: um multiplicador de dois valores e um somador de dois valores. O multiplicador é do tipo incremental, ou seja, ele utiliza as operações do somador para realizar multiplicações (ex: $4 * 3 = 4 + 4 + 4 = 12$).

A.7.1 Implementação do Somador

O somador é um componente sem receptáculo que provê sua operação de adição de valores através da interface *IAdder*.

```
public interface IAdder {
   public int add(int x, int y);
}
```

A implementação do somador (classe *Adder*) é mostrada abaixo. Como ele é um componente OpenCOM, a classe *Adder* deve herdar a classe *OpenCOMComponent*. Além disso, as interfaces *IUnknown*, *ILifeCycle* e *IMetaInterface* (que fazem parte do ambiente OpenCOM) devem ser implementadas. Da interface *ILifeCycle*, duas operações devem ser implementadas. A primeira (*startup*), é chamada pelo OpenCOM quando o componente é ativado e a segunda (*shutdown*) é chamada quando o componente é desativado.

```
public class Adder extends OpenCOMComponent
                   implements IAdder, IUnknown,
                                      ILifeCycle, IMetaInterface{

    public Adder(IUnknown binder) {
       super(binder);
    }

    public int add(int x, int y) {
        return (x+y);
    }

    public boolean startup(Object data) {
            return true;
    }

    public boolean shutdown() {
        return true;
    }
}
```

A.7.2 Implementação do Multiplicador

O multiplicador é um componente com receptáculo que provê operação de multiplicação de dois valores através da interface *IMultiplier*.

```
public interface IMultiplier {
   public int multiply(int x, int y);
}
```

O multiplicador é implementado pela classe *IncMultiplier* apresentada abaixo. Como essa classe possui receptáculo, além dela implementar as mesmas interfaces do pacote OpenCOM que o somador implementou, ela deve implementar a interface *IConnections*. Essa interface possui duas operações que devem ser implementadas. A primeira (*connect*) deve implementar a conexão de uma dada interface ao receptáculo do multiplicador. A segunda (*disconnect*) desconecta uma interface do receptáculo do multiplicador. O receptáculo através do qual o multiplicador se conecta ao somador é definido através de um atributo do tipo *OCM_SingleReceptacle*, por se tratar de um receptáculo simples. Pode-se notar que a operação de multiplicação (*multiply*) é implementada usando as operações de um somador acessado através do receptáculo do multiplicador.

```
public class IncMultiplier extends OpenCOMComponent
                           implements IMultiplier, IUnknown,
                                      ILifeCycle, IMetaInterface,
                                      IConnections {

  // Definição do receptáculo simples
  public OCM_SingleReceptacle<IAdder> rcAdder;

  public IncMultiplier(IUnknown binder) {
    super(binder);
    this.rcAdder =
         new OCM_SingleReceptacle<IAdder>(IAdder.class);
  }

  public int multiply(int x, int y) {
    int signal = 1;
    int abs_x = Math.abs(x);
    int abs_y = Math.abs(y);
    int result = abs_y;
    if (((x>=0)&&(y<0))||((x<0)&&(y>=0))) {
       signal = -1;
    }
```

```
    for (int i=1;i<abs_x;i++){
        result = this.rcAdder.m_pIntf.add(result, abs_y);
    }
    return (result/signal);
  }

  public boolean connect(IUnknown compToConnect,
                         String riid, long provConnID) {
    if (riid.equals("IAdder")){
        return this.rcAdder.connectToRecp(compToConnect,
                                          riid, provConnID);
    }
    return false;
  }

  public boolean disconnect(String riid, long connID) {
    if (riid.equals("IAdder")){
        return this.rcAdder.disconnectFromRecp(connID);
    }
    return false;
  }

  public boolean startup(Object data) {return true;}
  public boolean shutdown() {return true;}
}
```

A.7.3 Execução do Exemplo no OpenCOM

A seqüencia de operações necessárias para se executar o exemplo apresentado acima é listada a seguir.

```
public static void main(String[] args){
  OpenCOM ocm = new OpenCOM();
  IOpenCOM ocm = (IOpenCOM) ocm.QueryInterface("OpenCOM.IOpenCOM");

  // Criação da instância MyAdder do componente Adder
  IUnknown adderUnk = (IUnknown) ocm.createInstance("Adder",
                                                    "MyAdder");
  // Ativação da instância MyAdder
  ILifeCycle adderLife =
       (ILifeCycle) adderUnk.QueryInterface("OpenCOM.ILifeCycle");
```

```
    adderLife.startup(ocm);

    // Criação da instância MyIncMultiplier do componente IncMultiplier
    IUnknown multUnk = (IUnknown) ocm.createInstance("IncMultiplier",
                                                     "MyIncMultiplier");

    // Ativação da instância MyIncMultiplier
    ILifeCycle multLife =
         (ILifeCycle) multUnk.QueryInterface("OpenCOM.ILifeCycle");
    multLife.startup(ocm);

    // Conexão do receptáculo de MyIncMultiplier com a interface
    // IAdder de MyAdder
    long connID_Mul_Add = ocm.connect(multUnk, adderUnk, "IAdder");

    // Recuperação da interface do multiplicador
    IMultiplier mult =
         (IMultiplier) multUnk.QueryInterface("IMultiplier");

    // Execução de uma operação de multiplicação
    System.out.println("Operacao 15*2 = "+mult.multiply(15,2));
}
```