Um Sistema de Gerenciamento de Workflows Cooperativos

Hudo Rodrigues de Almeida

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Um Sistema de Gerenciamento de Workflows Cooperativos

Hudo Rodrigues de Almeida

Dezembro de 2002

Banca Examinadora:

- Dra. Maria Beatriz Felgar de Toledo
  IC/UNICAMP – Universidade Estadual de Campinas (Orientadora)
- Dra. Itana Maria de Souza Gimenes
  Departamento de Informática – Universidade Estadual de Maringá
- Dr. Célio Cardoso Guimarães
  IC/UNICAMP – Universidade Estadual de Campinas
- Dra. Anamaria Gomide (Suplente)
  IC/UNICAMP – Universidade Estadual de Campinas

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Almeida, Hudo Rodrigues de

AL64s Um sistema de gerenciamento de workflows cooperativos / Hudo Rodrigues de Almeida -- Campinas, [S.P. :s.n.], 2002.

Orientadora : Maria Beatriz Felgar de Toledo

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

1. Fluxo de trabalho. 2. Cooperação. 3. Sistemas de transação (Sistemas de computação). 4. Computação tolerante de falhas. I. Toledo, Maria Beatriz Felgar de. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 02 de dezembro de 2002, pela Banca Examinadora composta pelos Professores Doutores:

Profa. Dra. Itana Maria Souza Gimenes
UEM

Prof. Dr. Célio Cardoso Guimarães
IC - UNICAMP

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

Um Sistema de Gerenciamento de Workflows Cooperativos

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Hudo Rodrigues de Almeida e aprovada pela Banca Examinadora.

Campinas, 02 de Dezembro de 2002.

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

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

Resumo

Esta dissertação propõe uma arquitetura para um sistema de gerenciamento de *workflows* que atenda aos requisitos de aplicações para o desenvolvimento de projetos. São considerados modos mais flexíveis para troca de informações entre usuários que trabalham no mesmo projeto, no contexto de uma transação de grupo. Em sistemas de *workflow* tradicionais, a troca de informações é realizada através de parâmetros entre tarefas. O modelo proposto permite a troca de resultados intermediários entre usuários de uma transação de grupo. Além disso, o sistema garante a execução confiável de processos mesmo quando falhas ocorrem. Finalmente, um protótipo foi implementado sobre uma plataforma distribuída baseada em CORBA.

Abstract

This dissertation proposes an architecture for a workflow management system that meets the requirements of project development applications. More flexible modes for information exchange between users that work on the same project are considered, in the context of a group transaction. In traditional workflow systems, the information exchange is achieved by parameter passing between tasks. The proposed model allows the exchange of intermediary results between users within a group transaction. Moreover, the system guarantees the reliable execution of processes even when failures occur. Finally, a prototype has been implemented on a CORBA-based distributed platform.

Agradecimentos

Aos amigos que fiz no Mestrado, que muito me ajudaram durante o período de quase três anos em que estivemos juntos. Ao Felipe e ao Guilherme, pela ajuda nas disciplinas e pelo companheirismo. Ao Fábio, pelas conversas, pelas risadas, pelas idéias compartilhadas, pelas dicas e pela ajuda. Ao Leonardo, com quem tive maior contato durante o Mestrado. A ele, que me ajudou na fase de elaboração e implementação do sistema, que me esclareceu dúvidas (e quantas dúvidas), que foi meu parceiro de projeto e com quem conversei, ri, compartilhei idéias e sonhos!

À minha noiva, Camila, pela pessoa maravilhosa que você é, pelo carinho, pelo amor e pelas palavras de ânimo. Obrigado por estar sempre ao meu lado, por eu poder contar com você, pelas alegrias e dificuldades que compartilhamos, por ser minha companheira e por me fazer sentir tão amado. Te amo de paixão!!!

Aos meus pais Aureliano e Zélia e aos meus irmãos Fernando e Caroline, por estarem sempre presentes e por me apoiarem a todo instante. Agradeço por intercederem por mim em suas orações. Certamente elas foram ouvidas! Obrigado porque, pelo simples fato de existirem, são dádivas de Deus em minha vida. Vocês são muito preciosos! Amo vocês demais!!!

À minha orientadora Beatriz, por toda a ajuda durante a elaboração do projeto, pelos refinamentos realizados durante a escrita da tese e do artigo, pelas críticas sempre construtivas, pelas sugestões e pela compreensão que foi fundamental para o meu crescimento e aprendizado!

A todos os que foram citados, eu afirmo que vocês são parte integrante do meu sucesso hoje! Desejo-lhes muita prosperidade e felicidade ... e que Deus os abençoe em todas as áreas de suas vidas!

Finalmente, a Jesus Cristo, meu Deus e Senhor da minha vida, que esteve comigo incessantemente, que me ajudou, me capacitou, firmou meus passos, renovou minhas forças e me deu vitória em todas as dificuldades. Te dou graças por que eu não seria o que sou sem o Teu favor. Sem Ti eu não sou nada. Eu Te amo mais do que tudo!!!

"Em todas estas coisas, porém, somos mais que vencedores, por meio daquele que nos amou." (Bíblia Sagrada – Romanos 8:37)

Acrônimos

São indicados abaixo, em ordem alfabética, os principais acrônimos utilizados neste trabalho.

ACID – Atomicidade, Consistência, Isolamento e Durabilidade
API – Application Program Interface
BNF – Backus Normal Form
CAD – Computer-Aided Design
CASE – Computer-Aided Software Engineering
CORBA – Common Object Request Broker Architecture
CoAct – Cooperative Activity Model
DM – Definition Manager
GA – Geradas Automaticamente
GIOP – General Inter-ORB Protocol
GTM – Group Transaction Manager
IDL – Interface Definition Language
IIOP – Internet Inter-ORB Protocol
IM – Instance Manager
JDK – Java Development Kit
METEOR – Managing End-To-End OpeRations
OMA – Object Management Architecture
OMG – Object Management Group
ORB – Object Request Broker
PDL – Process Definition Language
PDLI – Process Definition Language Interpreter
PM – Process Manager
RM – Recovery Manager
SGBD – Sistema de Gerenciamento de Bancos de Dados
SGWF – Sistema de Gerenciamento de Workflows
SQL – Simple Query Language

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

A tecnologia de *workflow* tem sido cada vez mais utilizada pelas empresas a fim de automatizar ou semi-automatizar seus processos de negócio. Embora existam muitos produtos comerciais disponíveis tanto para a especificação como para a execução de *workflows*, a maioria desses produtos ainda apresenta muitas limitações [5, 16, 24]. Essas limitações estão associadas com os aspectos abaixo:

Desempenho - Os processos das empresas apresentam restrições quanto ao número de usuários que executam suas atividades de forma concorrente e quanto ao número de processos que estão sendo executados ao mesmo tempo.

Escalabilidade - Os sistemas comerciais geralmente são centralizados, o que facilita o monitoramento e a sincronização das tarefas, porém esses sistemas são mais vulneráveis a falhas e menos escaláveis. Algumas arquiteturas distribuídas começaram a aparecer mais recentemente [2, 8].

Confiabilidade - Os sistemas comerciais centralizados não possuem mecanismos de recuperação eficientes quando ocorrem falhas. Existem propostas que utilizam mecanismos de transações e também replicação e distribuição de servidores [7] para melhorar a confiabilidade.

Disponibilidade - A replicação de bancos de dados e outros componentes mais críticos pode assegurar maior disponibilidade.

Mobilidade - O desenvolvimento de computadores portáteis e a comunicação sem fio possibilitaram a mobilidade de usuários. O projeto Exotica [6] e o projeto definido em [39] já contemplam esse aspecto, permitindo que usuários trabalhem em modo desconectado.

Interoperabilidade - Devido à tendência de se interligar vários sistemas distribuídos e heterogêneos [14] e à existência de um grande número de produtos disponíveis comercialmente, tornou-se necessária a padronização de conceitos e interfaces [26].

Concorrência - Como várias tarefas pertencentes a diferentes processos podem ser executadas concorrentemente, devem existir mecanismos para evitar interferência entre essas tarefas. O modelo *ConTract* [25], por exemplo, utiliza invariantes para sincronização entre tarefas. Em [10], é proposto um mecanismo baseado em uma matriz de compatibilidades.

Cooperação - Os requisitos para a execução de tarefas cooperativas têm se resumido à troca de informações entre tarefas através de parâmetros de entrada e saída. As aplicações para desenvolvimento de projeto necessitam de formas mais flexíveis para troca de informações [23] a fim de facilitar o trabalho cooperativo entre usuários.

No contexto desse trabalho, estaremos tratando, em especial, da flexibilização de formas de cooperação entre usuários e da recuperação de falhas.

1.1 Motivação

Em sistemas de gerenciamento de *workflows* (SGWFs) tradicionais, a troca de informações é realizada através de parâmetros passados entre tarefas. Isso, contudo, não atende os requisitos de aplicações para desenvolvimento de projetos como CAD, CASE, por exemplo. São necessárias formas mais flexíveis de cooperação, como será visto no exemplo a seguir.

Consideremos que uma empresa E que desenvolve sistemas de *software* para outras empresas possui um processo gerenciado por um SGWF, composto por diversas tarefas, dentre elas `ElaborarEspecificacao`. Essa tarefa é executada por um analista de sistemas que, a partir de um documento de requisitos elabora uma especificação do sistema que o cliente deseja adquirir de E, informando depois ao SGWF se obteve sucesso na execução desta atividade. Caso o documento de requisitos não contenha informações suficientes ou estas informações não estejam claras, o analista informa ao SGWF de que não teve sucesso na elaboração da especificação do sistema.

Na elaboração da especificação do sistema, o analista de sistemas que coordena o projeto define quem serão os programadores que participarão da implementação do sistema, bem como quais classes estes programadores implementarão. Por exemplo, a analista de sistemas *Vera* define que a programadora *Maria* implementará a classe *Administracao.java*, que o programador Luís implementará a classe *Estoque.java* e assim por diante.

Após a elaboração da especificação do sistema, que envolve documentos de análise e projeto de *software*, o analista de sistemas informa ao SGWF sobre o sucesso na execução

desta atividade e o SGWF notifica ao coordenador do projeto sobre a possibilidade de começarem a implementar as classes que compõem o sistema a ser implementado utilizando o paradigma de orientação a objetos. Para a execução desta atividade cooperativa, os programadores da empresa que estão envolvidos no projeto participam implementando as classes dos quais foram incumbidos.

Durante a implementação das classes, os programadores podem trocar as classes que estão implementando através do SGWF, a fim de facilitar seus respectivos trabalhos. Conforme os programadores vão terminando a implementação de suas respectivas classes, eles informam ao SGWF sobre o término da sua atividade, bem como o seu resultado. Após a execução de todas as atividades que compõem a tarefa cooperativa, o coordenador do projeto encerra a tarefa cooperativa, informando seu resultado final.

As atividades cooperativas serão consideradas transações neste sistema, necessitando assim de um gerenciamento mais complexo e específico para que sejam obedecidas as restrições e características de transações.

Como um SGWF é uma ferramenta para facilitar o trabalho colaborativo, ele exige um monitoramento constante de seus diversos componentes e das ações dos usuários que o utilizam, garantindo assim maior confiabilidade na execução de tarefas complexas e que envolvem capital e prestígio para a empresa.

1.2 Objetivos

Neste trabalho, será definida uma arquitetura para um sistema de gerenciamento de *workflows* compatível com o Modelo de Referência para *Workflow* da *WfMC* (*Workflow Management Coalition*) [26]. O sistema de *workflow* proposto, **WorkToDo**, permitirá a definição de processos como um conjunto de atividades ou tarefas. As tarefas podem ser de vários tipos, como por exemplo, tarefas manuais, automáticas, semi-automáticas comuns e semi-automáticas cooperativas.

Além disso, o WorkToDo fornecerá ferramentas para interpretar a definição de processos e permitir sua execução. O foco desse trabalho serão as tarefas cooperativas que oferecem formas mais flexíveis e dinâmicas para facilitar a cooperação entre usuários. Entendemos por **Cooperação** a troca controlada de resultados intermédiários de trabalho entre os usuários que participam de um grupo. Os usuários de um grupo poderão participar de uma tarefa cooperativa executando transações dentro de uma transação de grupo e trocando resultados parciais entre eles, conforme a necessidade do trabalho cooperativo. O sistema WorkToDo também oferecerá facilidades para recuperação de falhas, garantindo a execução correta de processos.

Finalmente, será implementado um protótipo, utilizando a camada ORB de CORBA como infra-estrutura de comunicação. Esse protótipo deverá validar o modelo proposto.

1.3 Organização da Dissertação

Esta dissertação está organizada como se segue. O Capítulo 2 apresenta os conceitos básicos utilizados neste trabalho. O Capítulo 3 descreve os trabalhos relacionados ao WorkToDo. O Capítulo 4 apresenta a descrição do modelo e da arquitetura do WorkToDo. O Capítulo 5 apresenta algumas características da implementação do protótipo do sistema. Finalmente, o Capítulo 6 apresenta as conclusões, contribuições e possíveis trabalhos futuros.

Capítulo 2

Conceitos Básicos

Neste Capítulo, são introduzidos os conceitos básicos que envolvem os Sistemas de Gerenciamento de *Workflows* (SGWFs) e o gerenciamento de transações. São também apresentados resumidamente os conceitos da arquitetura CORBA e as características principais do OrbixWeb, que foi a implementação de CORBA utilizada para a construção do protótipo do WorkToDo.

2.1 Sistemas de Gerenciamento de Workflows

Um ***Sistema de Gerenciamento de Workflows*** (***SGWF***) é um sistema que define, gerencia e executa *workflows* de forma automatizada [29]. Isso se dá através do gerenciamento da seqüência de atividades e da invocação de recursos humanos e/ou de Tecnologia da Informação associados com os vários passos da atividade [26].

Os conceitos relacionados a Sistemas de Gerenciamento de *Workflows* são descritos nas Seções seguintes.

2.1.1 Processo de Negócios

Um ***Processo de Negócios*** é um conjunto de atividades com um objetivo comum. O processo de negócios é formado por diversas atividades e pelas especificações de fluxo de dados e de controle entre elas [4]. Os processos de negócios tendem a ser de longa duração, envolvendo muitos usuários e ferramentas sobre ambientes heterogêneos e distribuídos.

Alguns exemplos de processos de negócios são:

1. A reserva de passagens aéreas juntamente com a estadia em hotéis, seguindo um roteiro de viagem;

2. A abertura de uma nova conta bancária;

3. A matrícula de um aluno em um curso superior de uma universidade particular;

4. A instalação de uma rede de computadores em uma empresa;

As atividades citadas acima podem ser divididas em tarefas menores, de tal forma que, ao final da execução de cada tarefa, a atividade como um todo esteja concluída. Por exemplo, o processo 3 pode ser dividido nas seguintes tarefas: verificar histórico de conclusão do segundo grau do aluno, cadastrar os dados do aluno no sistema, conferir a renda do aluno e atribuir ao aluno um registro acadêmico. Após a conclusão destas tarefas, o aluno estará matriculado no curso universitário, ou seja, a atividade 3 estará concluída.

Geralmente, os processos de negócios são de longa duração e envolvem muitas entidades de processamento presentes em diferentes localidades.

2.1.2 Workflow

Um ***Workflow***[1] é a representação computacional de um processo de negócios em que documentos, informações ou tarefas são passados de um participante[2] para outro, de acordo com um conjunto de regras e procedimentos [30].

Um processo de negócios possui a seguinte estrutura, que é definida por um *workflow*:

Tarefa - É o conjunto de ações elementares que devem ser desempenhadas para que um processo seja completado. Um conceito mais detalhado de tarefa é encontrado na Seção 2.1.4.

Responsável - É a entidade incumbida da execução de uma tarefa. As entidades podem ser seres humanos ou aplicativos. Podem haver vários responsáveis por uma única tarefa ou um mesmo responsável para diversas tarefas.

Seqüência de Execução - É a seqüência em que as tarefas devem ser executadas.

Fluxo de dados - É a especificação de como as informações são passadas entre as tarefas.

2.1.3 Instância de Workflow

Uma ***Instância de Workflow***[3] é uma execução particular de um determinado *workflow*, com parâmetros e dados específicos.

[1] ou ***Processo de Workflow*** ou simplesmente ***Processo***
[2] participante = recurso (ser humano ou máquina)
[3] ou ***Instância de Processo***

Em um dado momento, várias instâncias de um mesmo *workflow* podem estar em execução simultaneamente. Por exemplo, para um *workflow* de matrícula em um curso superior, pode existir uma instância para o aluno A, outra para o aluno B e uma terceira para o aluno C.

2.1.4 Tarefa

Uma ***Tarefa*** é uma ação elementar a ser executada para que um processo de negócios seja concluído. Um processo de negócios pode conter diversas tarefas, de variados tipos, com diferentes prioridades e com diferentes tempos de execução. As tarefas podem ser executadas por seres humanos, por sistemas de *software* ou por ambos.

Alguns exemplos de tarefas são listados abaixo:

1. Fazer uma ligação telefônica;
2. Assinar um documento;
3. Realizar uma operação básica sobre um registro em um banco de dados (inclusão/exclusão/alteração/consulta);
4. Preencher um formulário;
5. Imprimir um documento;
6. Escanear uma fotografia;
7. Instalar um dispositivo;
8. Criar cópia (*backup*) de uma pasta de arquivos;
9. Emitir um relatório;
10. Enviar um *e-mail*.

2.1.5 Modelo de Referência para Workflow

A *Workflow Management Coalition* (*WfMC*) é uma organização composta por mais de 200 empresas fabricantes e usuárias de sistemas de *workflow* que se uniram para tentar solucionar o problema de não haver nenhum padrão que possibilite a integração dos produtos de gerenciamento de *workflows* existentes no mercado e seu objetivo é identificar as características comuns entre estes produtos e desenvolver especificações apropriadas para a implementação dos mesmos. O modelo de referência para sistemas de gerenciamento de *workflows* (SGWFs), descrito a seguir, é o resultado desse esforço.

O ***Modelo de Referência para Workflow*** [26] foi desenvolvido a partir da estrutura genérica de sistemas de *workflow* identificando-se as interfaces dentro desta estrutura que capacitam os produtos a interoperarem em uma variedade de níveis. Para se obter interoperabilidade entre os produtos de *workflow*, é necessário um conjunto padronizado de interfaces e formatos de intercâmbio de dados entre os componentes genéricos que interagem num sistema de *workflow*.

O modelo possui três áreas funcionais básicas, mostradas na Figura 2.1 e descritas a seguir:

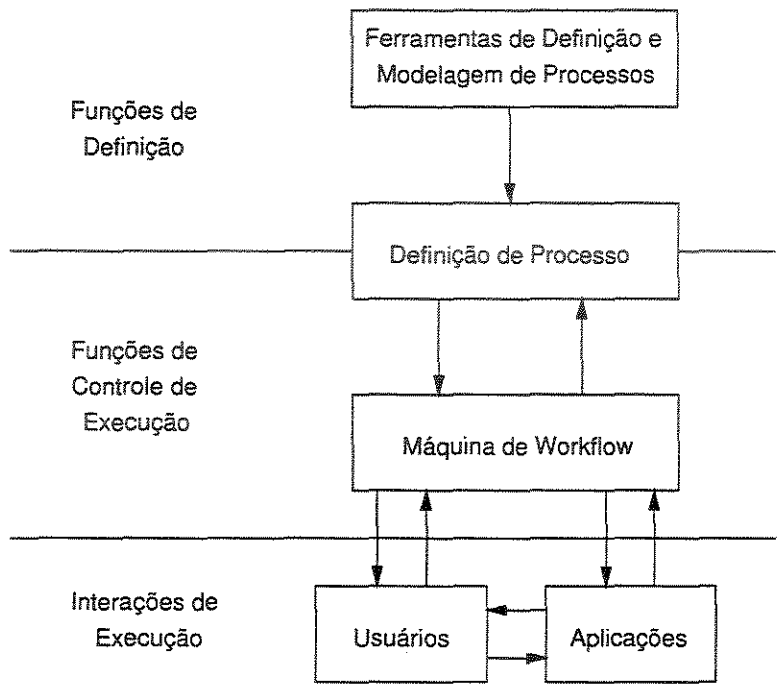

Figura 2.1: Áreas funcionais básicas do Modelo de Referência para *Workflow*

Funções de Definição - São funções de definição e modelagem de um processo e das atividades que fazem parte desse processo. Tais funções são úteis para a conversão de processos do mundo real para um formato computacional, chamado ***Definição de Processo***.

Funções de Controle de Execução - São funções utilizadas na interpretação e execução de processos, bem como para a coordenação e escalonamento da execução das atividades de cada processo. Tais funções são providas por uma ***Máquina de Workflow***.

Interações de Execução - São as interações do SGWF com usuários e também com outras aplicações e são necessárias para o processamento das atividades.

O Modelo de Referência para *Workflow* é identificado através de uma arquitetura básica, com seus componentes definidos de acordo com as áreas descritas acima e com suas interfaces. Os componentes e interfaces do modelo são mostrados na Figura 2.2 e serão abordados a seguir:

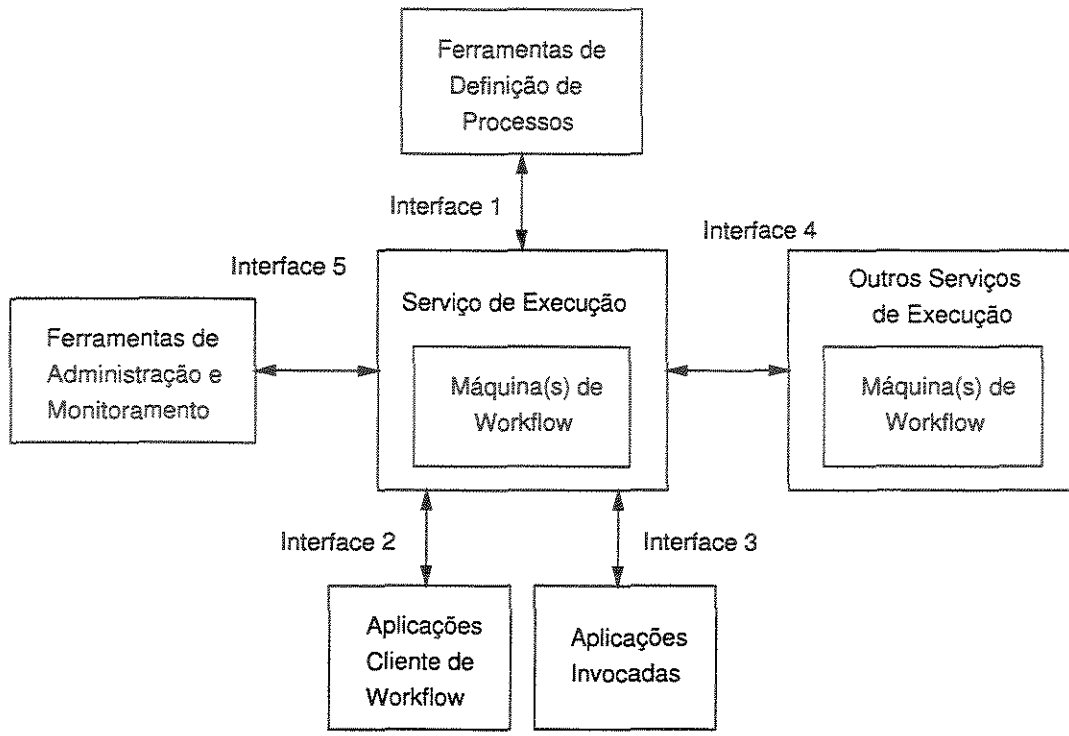

Figura 2.2: Componentes e Interfaces do Modelo de Referência para *Workflow*

1. ***Serviço de Execução*** - Pode consistir de uma ou mais máquinas de *workflow* com o objetivo de criar, gerenciar e executar instâncias de *workflow*. As aplicações podem fazer interface com este serviço via interface de programação de aplicação de *workflow* (WAPI). A máquina de *workflow* fornece o ambiente de execução para uma instância de *workflow*. Este componente possui uma interface específica para cada um dos outros componentes.

2. ***Ferramentas de Definição de Processos*** - São ferramentas que podem ser fornecidas como parte de um produto de *workflow* ou separadamente. Quando um produto de *workflow* é fornecido com sua própria ferramenta de definição de processo, as definições de processo resultantes podem ou não ser acessíveis via interface de programação para leitura e escrita de informações (Interface 1 da Figura 2.2). Quando produtos separados são usados para definir e executar o processo, as definições de processo podem ser transferidas entre os produtos ou podem ser armazenadas

em um repositório separado que seja acessível a ambos os produtos. A saída final desta atividade de modelagem de processo é uma definição de processo que pode ser interpretada em tempo de execução pela(s) máquina(s) dentro do serviço de execução.

3. ***Aplicações Cliente de Workflow*** - Consistem de entidades de *software* que interagem com o usuário final naquelas atividades que requerem e envolvem recursos humanos. No modelo de *workflow*, a interação entre a aplicação do cliente e a máquina de *workflow* se dá através de uma interface bem definida (Interface 2 da Figura 2.2) que abrange o conceito de uma lista de ***Itens de Trabalho*** atribuídos a um usuário particular ou a um grupo de usuários pela máquina de *workflow*. Essa lista é chamada ***Lista de Trabalho***.

4. ***Aplicações Invocadas*** - A máquina de *workflow* deve ser capaz de invocar aplicações, transferir dados e responder a eventos de atividades. Os aplicativos podem ser locais à máquina de *workflow* ou podem exigir transferência de informações entre máquinas heterogêneas.

5. ***Outros Serviços de Execução*** - Uma máquina de *workflow* de um fabricante pode interoperar com máquinas de outros fabricantes através da Interface 4 da Figura 2.2. Com a padronização, é possível que itens de trabalho sejam transferidos de uma máquina para outra, ou seja, diferentes Serviços de Execução podem operar em conjunto para a execução de um mesmo processo.

6. ***Ferramentas de Administração e Monitoramento*** - A padronização de interface para funções de administração visa a permitir que aplicações de gerenciamento de um fabricante interopere com máquinas de *workflow* de outros fabricantes. Essa interface (Interface 5 da Figura 2.2) permite que se tenha uma visão completa do estado de um processo, incluindo também funções relacionadas com segurança, controle e autorização.

Como se observa na Figura 2.2, o componente Serviço de Execução interage com todos os outros componentes do Modelo de Referência para *Workflow* através de cinco interfaces. Em cada interface, são definidas as formas de comunicação com cada componente e a forma como ocorre a transferência de dados e informações entre eles. Isso faz com que diferentes SGWFs que respeitem as definições das interfaces sejam capazes de operar de forma conjunta, independentemente de suas implementações.

2.2 Transações

Uma ***Transação*** é uma unidade de execução que possibilita um processamento concorrente e confiável, envolvendo o acesso a dados compartilhados. Tal garantia é possível devido à sincronização automática e apropriada dos acessos a dados e à recuperação de falhas [20].

As transações foram inicialmente utilizadas no contexto de aplicações de bancos de dados. Com o passar do tempo, surgiu a necessidade de se utilizar outros modelos de transação, para atender os requisitos de novas aplicações, tais como longa duração, necessidade de cooperação entre usuários, por exemplo. A partir daí, surgiram os ***Modelos de Transação Avançados*** [27].

Nesta Seção, são descritos os conceitos básicos de alguns dos principais tipos de transação relevantes no contexto dessa tese.

Transação ACID

O conceito de ***Transações ACID*** [31] surgiu no contexto de sistemas de bancos de dados e possui quatro propriedades principais, que são: Atomicidade, Consistência, Isolamento e Durabilidade[4].

A propriedade de **Atomicidade** garante que todas as operações são executadas com sucesso ou nenhuma delas é executada. A seqüência de ações deve ser totalmente concluída ou abortada. Portanto, a transação não pode ser parcialmente executada, mesmo que ocorram falhas durante sua execução.

A propriedade de **Consistência** garante que a transação leve os recursos de um estado consistente para outro.

A propriedade de **Isolamento** garante que os efeitos de uma transação em execução não são visíveis para outras transações até que a transação seja concluída.

A propriedade de **Durabilidade** garante que as mudanças feitas pela transação concluída são permanentes e devem sobreviver a falhas do sistema.

Estas propriedades são úteis para aplicações como, por exemplo, sistemas de reserva de vôo, sistemas bancários e sistemas de controle de estoque. Essas aplicações têm curta duração e acessam poucos itens de dados.

Transação Aninhada

Um passo importante na evolução dos modelos transacionais básicos foi a extensão da estrutura de transação de um único nível para estruturas multi-níveis.

[4]Essas propriedades formam a sigla ACID

Uma ***Transação Aninhada***[5] [20] é um conjunto de subtransações que podem recursivamente conter outras subtransações, formando assim uma ***Árvore de Transações***. Se uma transação pai é abortada, todos os seus filhos são abortados. Entretanto, quando um filho falha, o pai pode escolher sua própria forma de recuperação. Por exemplo, o pai pode executar outra subtransação que realiza uma ação alternativa (chamada "subtransação de contingência") [27].

As transações aninhadas fornecem isolamento no nível global, mas elas permitem uma modularidade maior, uma granularidade mais fina de tratamento de falhas e um maior grau de concorrência intra-transação do que as transações tradicionais. São necessários, portanto, mecanismos de sincronização e recuperação, descritos a seguir:

Sincronização - Em um sistema de transações atômicas tradicionais, as transações são impedidas de interferir na execução umas das outras, mesmo se elas acessarem dados compartilhados concorrentemente. Um requisito típico é que as transações sejam serializáveis, ou seja, que o processamento concorrente de um grupo de transações seja equivalente a executar as transações uma de cada vez (serialmente), em alguma ordem. Uma forma de sincronizar transações que garante a seriabilidade são as trancas de leitura e escrita[6]. No modelo de transações aninhadas, o mecanismo de trancas tradicional é estendido para permitir que uma transação filha possa obter tranca sobre um item já bloqueado por sua transação pai e uma transação pai herde todas as trancas de seus filhos.

Recuperação - Em um sistema de transações tradicionais, a recuperação envolve assegurar que cada transação seja processada de forma completa ou então, em caso de falha, que não seja processada. As transações raiz têm a propriedade "tudo ou nada" sobre suas transações filhas. Mas, se uma transação filha falha (aborta), então o pai não precisa abortar. Isto permite que os pais tentem sua própria recuperação da falha de um ou mais filhos.

Transação Cooperativa

Uma ***Transação Cooperativa*** é uma transação que envolve atividades de longa duração, em que a competição por recursos é substituída pela necessidade de cooperação. A ênfase, portanto, não é prevenir o acesso a recursos, mas sim realizar a troca correta de informações entre atividades concorrentes de usuários que trabalham cooperativamente.

As propriedades do ambiente tradicional são, portanto, relaxadas. A maior parte dos modelos permite a organização de aplicações como transações hierárquicas que possuem

[5] Do Inglês: *Nested Transaction*
[6] Do Inglês: *read-write locking*

sua própria área de trabalho, onde resultados parciais podem ser armazenados e recuperados por subtransações de níveis inferiores [12].

O acesso controlado a recursos compartilhados pode ser realizado através de trancas com novos modos de acesso ou outros mecanismos baseados em semântica para garantir consistência e, ao mesmo tempo, permitir cooperação entre usuários.

A atomicidade é substituída por pontos de recuperação que salvam resultados parciais de trabalhos desenvolvidos pelo grupo.

2.3 CORBA

O ***OMG*** (***Object Management Group***) [32] é um consórcio criado em 1989, com o objetivo de definir padrões que possibilitassem a interoperabilidade de componentes de *software* em um ambiente distribuído, independentemente de sua localização, plataforma, sistema operacional e linguagem de programação [22, 28]. Como resultado da pesquisa do OMG, formado atualmente por mais de 800 empresas, originou-se o padrão ***CORBA*** (***Common Object Request Broker Architecture***) em 1991. Posteriormente, foi lançado o CORBA 2 [21] em 1996. A última versão do padrão é o CORBA 2.6, lançado em Dezembro de 2001.

A interoperabilidade proposta pelo OMG é implementada através de uma camada de *software* chamada ***Object Request Broker*** (***ORB***). Esta camada é definida pela especificação do CORBA [33] e fornece mecanismos em que objetos clientes podem requisitar serviços de objetos servidores de forma transparente [21]. Para o funcionamento satisfatório de tais mecanismos, o conjunto de serviços oferecidos pelos servidores deve estar definido através de uma linguagem chamada ***IDL*** (***Interface Definition Language***). Com o uso desta interface, o cliente pode fazer requisições junto ao servidor através das operações nela definidas.

A implementação de cada operação descrita na IDL deve ser fornecida pelos servidores, em quaisquer linguagens para as quais a IDL possa ser mapeada. Atualmente, essas linguagens são: C, C++, Java, Smalltalk, ADA, COBOL, LISP e Python. A especificação em IDL é convertida para a linguagem escolhida pelo projetista através de um compilador que gera as estruturas de implementação necessárias para as invocações de serviços, chamadas ***Stubs*** e ***Skeletons***.

Os *stubs* são utilizados pelos clientes para efetuar as invocações remotas para o servidor. O *stub* invoca as operações do *skeleton* que, por sua vez, repassa tais invocações para o servidor. Portanto, os *stubs* e *skeletons* tratam dos aspectos relativos à distribuição e localização dos servidores, tornando a comunicação entre clientes e servidores completamente transparente. A comunicação entre cliente e servidor se dá através de um protocolo padronizado pelo OMG, denominado ***GIOP*** (***General Inter-ORB Proto-***

col), que define um conjunto de primitivas de comunicação. Para redes baseadas nos protocolos TCP/IP, a implementação do GIOP é o protocolo *IIOP* (*Internet Inter-ORB Protocol*).

A requisição de um serviço de um cliente para um servidor utilizando a camada, as estruturas e o protocolo descritos acima é mostrada na Figura 2.3.

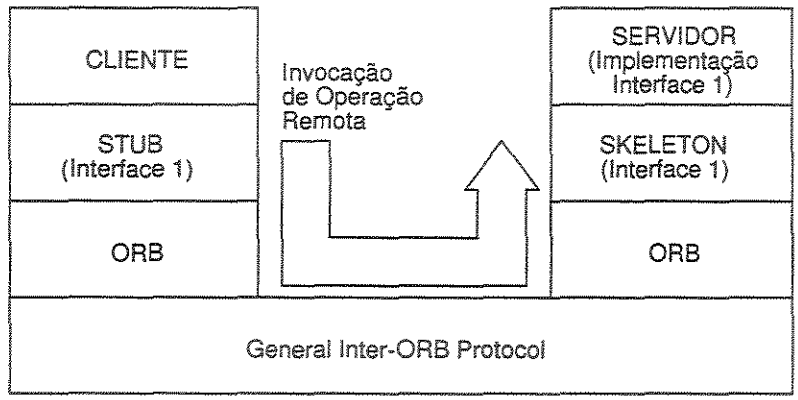

Figura 2.3: Requisição de um serviço através da camada ORB

Com o intuito de facilitar o desenvolvimento de aplicações distribuídas, o OMG desenvolveu também a arquitetura ***OMA*** (***Object Management Architecture***) que, além do ORB, especifica outros quatro componentes, que podem ser vistos na Figura 2.4 e são descritos abaixo.

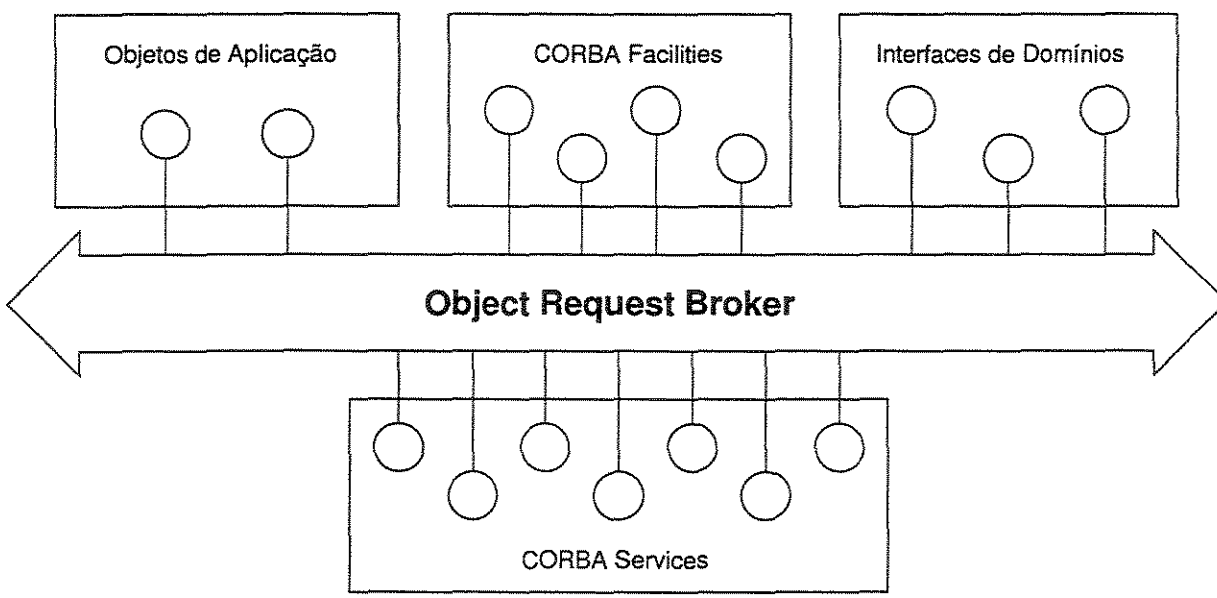

Figura 2.4: Arquitetura OMA com seus componentes

Objetos de Aplicação - São objetos não padronizados, criados por usuários da arquitetura OMA. Estes objetos podem ser escritos em qualquer linguagem e plataforma e podem utilizar os serviços oferecidos pelos demais componentes da arquitetura OMA.

CORBA Facilities - Este componente define um conjunto de serviços de maior uso pelas aplicações, mas que não são necessariamente serviços fundamentais. Estes serviços podem ser aplicados a qualquer domínio, tais como Internacionalização e Agentes Móveis.

Interfaces de Domínios - Definem interfaces IDL para áreas de aplicação específicas, tais como Telecomunicações, Bancos, Indústria e Saúde.

CORBA Services - Este componente define serviços que estendem as funcionalidades básicas do ORB. Tais serviços são, por exemplo, Serviço de Nomes, Serviço de Transações, Serviço de Persistência e Serviço de Concorrência.

A arquitetura OMA é definida em termos de interfaces e funcionalidades, porém não fornece nenhuma implementação. Apenas padrões são definidos e podem ser implementados de múltiplas formas. Uma implementação da arquitetura OMA deve fornecer os serviços definidos no padrão CORBA. Os demais componentes são opcionais.

2.3.1 OrbixWeb

O OrbixWeb 3.1c [34, 35, 36] foi a implementação de CORBA 2.0 da Iona [37] utilizada para a implementação do protótipo do WorkToDo. Essa implementação possui um compilador IDL com mapeamento para Java e a comunicação remota é realizada através do padrão IIOP.

O OrbixWeb possui um componente chamado ***orbixd*** (*OrbixWeb daemon*). Esse componente é um processo que deve ser executado em cada máquina que hospeda um servidor. Esse *daemon* implementa a maioria dos componentes do ORB e é responsável por controlar a ativação e desativação dos servidores. O *orbixd* recebe a requisição a um servidor e verifica se o mesmo está ativo. Caso esteja, ele passa essa requisição ao servidor em questão; caso contrário, ativa o servidor para então repassar a requisição. Daí em diante, o próprio servidor passa a receber as requisições dirigidas a ele. Com isso, não é necessário criar um servidor antes de fazer uma requisição através de suas operações; a criação é realizada automaticamente pelo *orbixd* sempre que houver necessidade.

Através do *orbixd*, pode-se localizar servidores e recuperar suas referências. A diferença entre o Serviço de Nomes padrão do CORBA e o *orbixd* é que este último deve ser executado em cada máquina que hospeda um servidor, pois o *orbixd* localiza e ativa

servidores somente na máquina em que está sendo executado. Para conseguir isso, ele se utiliza da operação `bind()` que é definida pelo OrbixWeb. Essa operação recebe como parâmetro o nome do servidor a ser localizado e o nome da máquina onde se quer localizá-lo.

Além do *orbixd*, pode-se ainda localizar os servidores através de requisições ao Serviço de Nomes do CORBA, oferecido pelo OrbixWeb. Além disso, o OrbixWeb ainda oferece serviços adicionais, estendendo assim as funcionalidades do ORB. Estes serviços são descritos abaixo:

Persistência dos servidores - Oferecido pelo *orbixd*, este serviço permite que o estado de um objeto servidor seja gravado de forma persistente quando o objeto é desativado e recuperado quando o mesmo é ativado. Tal serviço utiliza um ***Loader***, que deve ser associado ao servidor cujo estado se quer manter persistente. Para definir um *loader*, basta criar uma classe que estenda a classe `Loader` definida no OrbixWeb. Sempre que o *orbixd* ativa o servidor correspondente, seu *loader* é acionado e carrega o último estado gravado. Da mesma forma, quando o servidor é desativado, o seu respectivo *loader* é ativado e grava o estado atual do servidor.

Threads* múltiplas nos servidores** - Este serviço também é oferecido pelo *orbixd* e permite que um servidor receba múltiplas requisições de serviços ao mesmo tempo. Este serviço utiliza um ***Filtro, que deve ser associado a um servidor. Para definir um filtro, basta criar uma classe que estenda a classe `Filter` definida no OrbixWeb. Sempre que o servidor receber uma requisição, o filtro correspondente criará uma *thread* (linha de execução) para atender à requisição, permitindo que o servidor fique livre para receber outras requisições de clientes.

Filtros para operações - Além dos filtros que implementam servidores com múltiplas *threads* (descritos no item anterior), pode-se criar também outros tipos de filtros, que são executados no instante em que operações remotas são invocadas. Com isso, é possível invocar operações avançadas, tais como coleta de informações a respeito de invocações, controle de acesso e depuração.

Localização de objetos por nome - Este serviço, implementado pelo *orbixd*, permite localizar objetos específicos através de seu nome (*marker*). Uma lista com os nomes dos objetos servidores, bem como suas respectivas referências é mantida pelo *orbixd* em sua máquina. Desta forma, quando o *orbixd* recebe uma requisição para um objeto específico da sua máquina, ele busca este objeto em sua lista e retorna a referência para este objeto. Tal serviço representa um tipo de "serviço de nomes local" na máquina do *orbixd* em questão. A operação `bind()` recebe o *marker* do referido objeto. Tomando-se como exemplo a chamada `bind("Objeto_o:Servidor_s")`, o

orbixd retornará, como resultado da chamada, uma referência para o objeto do tipo `Servidor_s`, que tem como *marker* `Objeto_o`. Mesmo existindo vários objetos do tipo `Servidor_s` ativos nessa máquina, o *orbixd* irá retornar como resultado da chamada especificamente o objeto requisitado. Se nenhum *marker* for especificado na chamada da operação `bind()`, uma referência a qualquer objeto do tipo `Servidor_s` será retornada pelo *orbixd*.

Tempo de ativação dos servidores - Assim que um servidor é criado, pode-se estabelecer o tempo em que esse servidor permanecerá ativo. Quando esse tempo é atingido, o servidor é automaticamente desativado pelo *orbixd*. É possível estabelecer um tempo infinito para a ativação de um servidor, ou seja, o servidor permanecerá ativo enquanto seu processo Java correspondente não for finalizado explicitamente (por exemplo, com a utilização do comando `kill` do sistema operacional Linux).

Modos de ativação dos servidores - É possível especificar diferentes modos de ativação para cada servidor. Dois modos de ativação são considerados: o modo *shared* (compartilhado), em que todas as requisições recebidas por um servidor em uma determinada máquina são tratadas pelo mesmo objeto; e o modo *unshared* (não-compartilhado), em que, para cada requisição recebida por um servidor, é criado um novo objeto, com sua própria máquina virtual Java. Neste último modo, um servidor funciona da mesma forma que um servidor com múltiplas *threads*, porém ao invés de ser criada apenas uma *thread* para cada requisição, um novo processo é criado, o que representa uma desvantagem deste modo de ativação.

Smart Proxies - Possibilita que os *stubs* sejam definidos manualmente, ou seja, o programador pode tanto adicionar funcionalidades, como personalizá-las, conforme desejar. Com isso, as operações do lado do cliente podem ser otimizadas e políticas de balanceamento de carga entre os servidores podem ser implementadas.

Capítulo 3

Trabalhos Relacionados

A seguir, são descritos alguns trabalhos significativos na área.

3.1 Sagas

Uma transação de longa duração é um *saga* se ela puder ser escrita como uma seqüência de transações de curta duração. Sem sacrificar a consistência da base de dados, um saga permite que certas transações de longa duração liberem seus recursos antes de serem completadas, assim permitindo que outras transações bloqueadas prossigam.

Sagas [13] visam também diminuir a taxa de cancelamento de transações. A freqüência de *deadlock* é muito sensível à duração das transações. Portanto, já que transações de longa duração acessam muitos objetos, elas podem causar muitos *deadlocks* e, consequentemente, muitos cancelamentos.

Como exemplo, considere uma aplicação de reserva de uma linha aérea. A base de dados (ou uma coleção de bases de dados de diferentes linhas aéreas) contém reservas de vôos e uma transação T deseja fazer um certo número de reservas. Nesta aplicação pode não ser necessário bloquear todos os recursos até o final. Por exemplo, depois que T reserva uma passagem no vôo F_1, ela pode imediatamente permitir que outras transações reservem passagens no mesmo vôo. Em outras palavras, nós podemos ver T como uma coleção de "subtransações" T_1, T_2, ..., T_n que reservam passagens em vôos distintos.

Contudo, não se deseja submeter T ao Sistema de Gerenciamento de Bancos de Dados (SGBD) simplesmente como uma coleção de transações independentes porque ainda se quer que T seja uma unidade que é completada com sucesso ou não. Não seria satisfatório um SGBD que permitisse que T reservasse alguns lugares e fosse interrompida por uma falha. Por outro lado, seria satisfatório um SGBD que garantisse que T fizesse todas as suas reservas ou cancelasse quaisquer reservas feitas em caso de falha.

A noção de *saga* está relacionada a vários conceitos existentes. Por exemplo, um *saga* é como uma transação aninhada, exceto pelo fato de que:

1. Um *saga* permite somente dois níveis de aninhamento: o *saga* de nível mais alto e transações simples de curta duração e;

2. No nível externo, o isolamento completo não é fornecido. Isto é, os *sagas* podem ver os resultados parciais de outros *sagas*.

Quando uma falha interrompe um *saga*, há duas escolhas: compensar as transações já terminadas, (**recuperação com atraso**[1]) ou executar as transações que estão faltando (**recuperação com avanço**[2]). Para a recuperação com atraso, o sistema precisa de **transações de compensação** e, para recuperação com avanço, precisa de **pontos de recuperação**[3].

3.2 WAMO

O modelo ***WAMO*** (***Workflow Activity Model***) [11] capacita o projetista de *workflow* a modelar facilmente processos de negócios complexos de forma simples e direta. A idéia básica é decompor um processo de negócios complexo em unidades de trabalho menores (atividades) e garantir o fluxo de controle confiável (incluindo tratamento de exceções) através do uso de estruturas de controle e outras construções específicas de transações.

As construções básicas do modelo são atividades e tarefas. As atividades podem recursivamente consistir de outras atividades. As atividades aninhadas formam uma ***Árvore de Atividades***. As arestas entre atividades representam diferentes estruturas de controle e as folhas são tarefas. Tarefas são as unidades de trabalho elementares (por exemplo: transações ACID, um programa de aplicação ou uma interação com seres humanos).

As ***Estruturas de Controle*** são mecanismos que permitem a decomposição de processos de negócios em unidades de trabalho menores e a definição de fluxo de controle entre estas unidades. Há quatro estruturas de controle simples, mas poderosas: ***Seqüência***, ***Escolha por prioridades***, ***Escolha livre*** e ***Estrutura paralela***.

As atividades e tarefas passam por diferentes estados de execução, como por exemplo: `ativa`, `não ativa`, `concluída`, `abortada` ou `compensada`. Todas as atividades e tarefas são executadas sob o controle de um gerenciador de transações avançado.

Para cada tarefa, é possível especificar parâmetros como STORNO-TYPE, FORCE e VITAL, explicados a seguir.

[1] Traduzido do termo em inglês: *backward recovery*
[2] Traduzido do termo em inglês: *forward recovery*
[3] Traduzido do termo em inglês: *save-points*

Os parâmetros STORNO-TYPE e FORCE de uma tarefa são necessários para transações de compensação eventuais. Com o STORNO-TYPE o projetista pode especificar como uma tarefa específica se comporta no caso de compensação. Há quatro possibilidades de comportamento da tarefa:

- `Não compensável`: A tarefa concluída não precisa ser compensada porque não é relevante do ponto de vista de uma aplicação;
- `Compensável sem efeitos`: A tarefa concluída pode ser desfeita pela tarefa de compensação correspondente, sem qualquer efeito colateral. Por exemplo, um cliente faz uma reserva de vôo – mais tarde ele cancela a reserva, sem pagar qualquer taxa de cancelamento;
- `Compensável com efeitos`: A tarefa concluída pode ser semanticamente desfeita pela tarefa de compensação correspondente, mas há efeitos colaterais. Por exemplo, um cliente faz uma reserva de vôo – mais tarde ele cancela a reserva, mas agora ele tem que pagar a taxa de cancelamento;
- `Crítica`: A tarefa não pode ser desfeita ou compensada no caso de falha porque não existe nenhuma tarefa de compensação para desfazer os efeitos já concluídos.

Espera-se que algumas tarefas em situações do mundo real sempre terminem com sucesso. Esta característica natural pode também ser requerida de tarefas neste modelo através do uso do parâmetro FORCE na especificação de uma tarefa (por exemplo: abrir uma conta deveria ser sempre possível). Tais tarefas são repetidas e re-executadas em caso de falha (por exemplo: *deadlock*, recurso indisponível, etc.) até que uma resposta positiva seja recebida.

Podem existir também atividades que são *não essenciais* para a atividade pai terminar com sucesso. Para tais relações pai-filho, introduz-se o parâmetro de transação NON-VITAL. Se uma atividade **não vital** falha, o *workflow* pode prosseguir sem nenhuma ação de compensação. Se uma atividade **vital** falha, então o mecanismo de compensação é ativado. Por exemplo: se uma atividade vital em uma seqüência falha, então a seqüência inteira falha, o que significa que todas atividades e tarefas previamente concluídas na seqüência têm que ser compensadas.

Os requisitos de coordenação do *workflow* (fluxo de dados e controle) entre unidades de trabalho podem ser descritos *formalmente* por *regras de dependência* baseadas em transições de estado válidas. As transições de estado possíveis dependem principalmente de estados e valores de saída de outras atividades ou tarefas.

Para a formalização de um processo de negócio complexo, foi desenvolvida uma linguagem de alto nível simples de usar, chamada ***Linguagem de Descrição da Atividade de Workflow*** (***Workflow Activity Description Language - WADL***).

3.3 CoAct

O *CoAct* (*Cooperative Activity Model*) [23] é um modelo desenvolvido no contexto de atividades cooperativas, cuja ênfase está na *troca semanticamente correta de informações* entre atividades concorrentes de usuários que trabalham de forma cooperativa.

Nesse modelo, são importantes os conceitos abaixo:

- *Tipos de Atividades Cooperativas* que descrevem atividades cooperativas;
- *Atividades Cooperativas* como instâncias concretas de tipos de atividade cooperativa, consistindo de uma composição de atividades de usuários;
- *Atividades de Usuários* como atividades de usuários que participam colaborando em uma atividade cooperativa;
- *Base Comum de Atividades* para armazenar dados comuns de uma atividade cooperativa;
- *Espaços de Trabalho Privados* para armazenar dados de atividades de usuários individuais;
- *Integrador* responsável pela troca correta de informações entre usuários de uma atividade cooperativa e;
- *Facilidades de Coordenação* para usuários em atividades cooperativas.

Um tipo de atividade cooperativa consiste de cinco componentes, que são:

1. *Declaração de Atividade* - A declaração de um tipo de atividade cooperativa inclui um nome único do tipo de atividade cooperativa N, uma definição de parâmetros da atividade P e uma atividade de compensação C.

   A definição dos parâmetros da atividade inclui os parâmetros de entrada E_1, E_2, ..., E_n e a lista de parâmetros de saída S_1, S_2, ..., S_m. Os parâmetros de entrada e saída permitem uma instanciação parametrizada de um tipo de atividade cooperativa. A declaração de uma atividade de compensação C especifica a forma como os resultados de uma atividade cooperativa podem ser semanticamente desfeitos após sua conclusão.

2. *Subatividades Constituintes* - As subatividades constituintes são especificadas como um conjunto de atividades que pode ser uma atividade cooperativa, especificada por um tipo de atividade cooperativa ou uma atividade elementar. Uma atividade elementar, descrita por um tipo de atividade elementar, é um programa que consiste

de uma seqüência de invocações de métodos ou uma chamada para um programa externo.

3. *Estados e Transições de Estado da Atividade* - Estados de uma atividade cooperativa e suas transições de estado são descritos como um conjunto de estados observáveis S e possíveis transições ST: $S \rightarrow S$. Os estados observáveis das atividades incluem não `executada`, `em execução`, `concluída` e `desfeita`[4]. As transições entre os estados da atividade cooperativa podem ser representadas por um autômato de estados finitos e seu grafo de transições.

4. *Regras de Execução* - Um conjunto de regras de execução $\mathcal{E}$ é especificado para comandar a execução das subatividades constituintes. Estas regras podem ser interpretadas como restrições de integridade na ocorrência e precedência temporal de estados observáveis associados com a execução das subatividades. Regras de Execução são definidas em termos de estados observáveis, valores de saída de atividades e variáveis externas.

5. *Regras de Entrelaçamento de Atividades* - As regras de entrelaçamento de atividades definem uma possível existência de dependências entre subatividades constituintes de uma atividade em termos de fluxo de dados. Se duas subatividades dependem uma da outra, isso é relevante para uma integração da atividade com outras atividades. Se a subatividade B depende da subatividade A e deseja-se que B seja parte de uma integração, A tem que estar presente na integração também.

A ênfase do modelo CoAct é permitir que usuários compartilhem e troquem os resultados de seu trabalho concorrente e garantir que a atividade resultante esteja correta, isto é, obedeça todas as restrições especificadas no tipo de atividade cooperativa. Já que cada usuário que participa em uma atividade cooperativa tem seu próprio espaço de trabalho privado, a cooperação é obtida através da troca controlada e da sincronização dos conteúdos dos espaços de trabalho. Os mecanismos de troca permitem que usuários que cooperam entre si se comuniquem diretamente ou através da base comum de atividades. Os mecanismos são os seguintes:

- *Import* (atividade fonte): Para importar uma atividade de usuário no espaço de trabalho da atividade do usuário atual. A base de atividades comum, assim como outras atividades de usuário, pode ser referenciada como atividade fonte.

- *Delegate* (atividade destino): Para delegar a atividade de usuário do espaço de trabalho atual para o espaço de trabalho da atividade destino.

[4]Do inglês: `not executed`, `executing`, `done` e `undone`

- *Save*: Para gravar a atividade de usuário atual na base comum de atividades.

As facilidades de coordenação consistem de um conjunto de funções que implementam protocolos de coordenação pré-definidos. Algumas funções úteis para coordenar o processo de atividades cooperativas são:

- *Função preemptiva*: É executada quando, por alguma razão, a participação de um usuário não é desejada. Ele é eliminado da atividade cooperativa e seu trabalho pode ser assumido por um outro usuário.
- *Pedido de acesso exclusivo*: É executada por um usuário que quer tomar posse da atividade de um co-trabalhador, intercalando-a com sua própria atividade de usuário.
- *Pedido de cooperação*: É executada por um usuário que quer se juntar à atividade cooperativa em andamento ou para convidar uma nova pessoa para contribuir com a atividade cooperativa.
- *Pedido de observação*: Permite que diferentes políticas de notificação possam ser selecionadas. Uma vez que diferentes atividades de usuário estão ativas, a política de notificação determina quais eventos particulares estão sujeitos à notificação.

3.4 ConTracts

A idéia básica do modelo ***ConTract*** [25] é construir aplicações complexas a partir de transações ACID[5] curtas e fornecer um serviço independente de aplicação que controle estas transações. Como contribuição principal, o modelo permite que computações sejam realizadas de forma confiável e correta.

Um ConTract é uma execução consistente e tolerante a falhas de uma seqüência arbitrária de ações pré-definidas (chamadas passos) de acordo com uma descrição de fluxo de controle explicitamente especificada (chamada *script*).

No modelo de programação ConTract, a codificação dos passos é separada da definição de *scripts*. Qualquer linguagem de programação seqüencial seria adequada para a programação de passos. Já a linguagem para *scripts* deve incluir elementos como seqüência, desvio, laço e construtores paralelos que modelem fluxos de controle. A conseqüência, portanto, é que existem pelo menos dois "níveis" de programação. Do ponto de vista do programador de passos, os passos serão executados numa máquina virtual (gerenciador de recursos) que é arbitrariamente confiável e serão executados no modo mono-usuário.

[5]de Atomicidade, Consistência, Isolamento e Durabilidade

Quanto à recuperação de falhas, o modelo garante que todas as computações em execução sejam posteriormente reiniciadas. ConTracts interrompidos por falhas são restaurados para o estado consistente mais recente e continuam a execução desse ponto. Para possibilitar esse tipo de recuperação, as informações que constituem o contexto de um ConTract devem ser recuperáveis. São as seguintes:

Estado local - Constituído de variáveis de programa, janelas, cursores, etc.

Estado da computação - Constituído de informações sobre passos já executados, eventos disparados, etc.

Essas informações são armazenadas num banco de dados privado da computação. Já a manutenção de consistência exige:

1. Cancelamento de ConTracts através de ações compensáveis.

2. Alguma forma de sincronização entre os passos para garantir isolamento.

No caso de cancelamento de um ConTract a pedido de um usuário, os passos já executados e que, portanto, já externalizaram seus resultados, devem ser desfeitos através de ações compensáveis. A execução de ações compensáveis pode, por sua vez, provocar o cancelamento de outros ConTracts e a compensação de outros passos afetados por essa ação de compensação.

Para implementar a sincronização baseada na idéia de invariantes, o *script* de um ConTract deve incluir informação para:

1. Definir a visão de um ConTract depois de um certo passo ter sido executado. A condição que define essa visão é chamada **invariante de saída**. Estabelecer uma pós-condição significa vincular os valores correntes de objetos compartilhados a variáveis numa expressão de predicado.

2. Especificar quais destas invariantes de saída devem ser satisfeitas para um passo posterior ser executado corretamente. Este predicado é chamado de **invariante de entrada** do passo.

No caso de uma invariante não ser satisfeita, ações para tratar conflitos devem ser especificadas. O modelo permite que o passo seja abortado, re-executado ou que sejam especificados passos alternativos.

3.5 METEOR

Um *workflow* em ***METEOR*** (***Managing End-To-End OpeRations***) [18] é formado por múltiplas tarefas heterogêneas, executadas por entidades de processamento heterogêneas. Muitas tarefas não podem ser modificadas ou migradas para ambientes mais modernos devido ao tamanho e complexidade das entidades de processamento existentes. Assim, é necessário fornecer suporte tanto para os ambientes de execução atuais como para novos ambientes. Além disso, esse sistema se preocupa com o reuso das especificações de tarefas individuais, com a possibilidade de incluir as mudanças necessárias para que se obtenha maior funcionalidade e automação.

Um sistema de gerenciamento de *workflows* que utilize aplicações de múltiplos sistemas deve lidar com o suporte à especificação e execução, relacionados a diferentes tipos de tarefas, aos requisitos de coordenação entre tarefas e à troca de dados entre tarefas.

Geralmente, diferentes tipos de tarefas podem ter diferentes estruturas. Cada tarefa possui um estado `inicial`, com transição para o estado `em execução`. Pode haver uma ou mais transições desse estado para um estado de término (tais como `término com sucesso`, `término com falha`, `concluído` e `abortado`).

As três estruturas de tarefas consideradas nesse modelo são: as não transacionais, as transacionais e as de duas fases abertas. As tarefas **não transacionais** atingem um estado de término com falha se ocorre um erro durante sua execução. As tarefas **transacionais** têm um estado final `abortado` ou `concluído` e obedecem as propriedades ACID. O terceiro tipo de tarefa é o de **duas fases abertas**, ou seja, o das transações executadas por SGBDs que fornecem validação em duas fases. As transições de `preparado` para `abortado` ou de `preparado` para `concluído` são controláveis. Além disso, há dois tipos de transições de cancelamento a partir do estado `em execução`: uma que é controlável (usada quando há um *deadlock*) e outra que não é controlável e que é iniciada pela entidade de processamento.

A especificação de *workflow* envolve duas linguagens de alto nível: a ***WFSL*** (***Workflow Specification Language***) para expressar as interações no nível de aplicação entre múltiplas tarefas e a ***TSL*** (***Task Specification Language***) focando as questões relacionadas a tarefas individuais.

A WFSL é uma linguagem declarativa baseada em regras. Cada regra da WFSL tem dois componentes: uma parte de controle e uma parte de transferência de dados, que é opcional. A parte de controle denota as pré-condições para uma transição de tarefas. Sempre que uma tarefa faz uma transição de estado, diz-se que um evento ocorre. As pré-condições podem incluir referências a eventos e/ou saídas de dados de outras tarefas, assim como das variáveis de programa. Já a parte de transferência de dados indica quais saídas de outras tarefas ou variáveis (possivelmente passando por filtros) são entrada para

aquela tarefa. Outra característica da WFSL é que as especificações de tarefas podem ser aninhadas, o que permite que tarefas complexas como um *workflow*, com suas próprias tarefas, sejam especificadas como tarefas compostas por unidades de tarefas menores.

A TSL tem como objetivo principal a reutilização de tarefas existentes. A TSL fornece uma estrutura que descreve a interação com uma interface para uma entidade de processamento e compreende um conjunto de macros que pode estar embutido em uma linguagem hospedeira como C ou C++. A principal funcionalidade das macros da TSL é indicar pontos na execução da tarefa em que o gerenciador de *workflows* pode ser informado sobre o estado atual da tarefa. Como parte da TSL, pode-se prescrever também como novos programas de tarefas deveriam ser escritos. A interface se torna explícita na especificação da tarefa. Isto permite ao programador especificar o tratamento de erros para os erros de interface ou de entidades de processamento do sistema.

Em resumo, a WFSL lida com questões de empresas ou de aplicação. Ela é usada para especificar os *workflows*, incluindo todos os tipos de tarefas e classes em um *workflow*, todas as dependências entre tarefas e questões de recuperação de falhas no nível de aplicação e de tratamento de erros. A TSL é usada para especificar tarefas individuais, incluindo a recuperação de falhas no nível de tarefas e o tratamento de erros que são específicos para interfaces e/ou entidades de processamento.

Existem duas implementações do modelo. A primeira utiliza a camada ORB de CORBA como infra-estrutura de comunicação e tecnologia web para interface [17]. A segunda utiliza somente tecnologia web [19].

3.6 Exotica

O projeto ***Exotica*** [2, 7], desenvolvido pela IBM, explora alguns conceitos de modelos de transação avançados e arquiteturas cliente/servidor, dentro do contexto de gerenciamento de *workflows*.

O Exotica se baseia em um produto comercial da IBM, chamado ***FlowMark***. No FlowMark, um *workflow* é representado como um grafo acíclico dirigido, onde os nós representam atividades e os arcos representam conectores de controle e conectores de dados. As **atividades** podem ser simples ou subgrafos. Uma atividade simples tem associado a ela um programa que é invocado quando a atividade está pronta para execução e um usuário qualificado a inicia. Atividades subgrafos correspondem a blocos ou processos e são usadas para permitir aninhamento e composição no projeto de processos.

Cada atividade possui uma condição de início e uma condição de término, que especificam as condições para que a atividade seja iniciada e terminada, respectivamente. A ordem de execução das atividades é especificada por meio de **conectores de controle**. Cada atividade possui como entrada e saída um ou mais conectores. Assim que todos os

conectores de entrada de uma atividade são avaliados, sua condição de início é testada. Assim que uma atividade é terminada, sua condição de término é testada.

O fluxo de dados entre as atividades ocorre através dos **conectores de dados** e dos **recipientes de dados de entrada e de saída**. Cada atividade possui um recipiente de entrada e um recipiente de saída. Os conectores de dados mapeiam valores de um recipiente de saída de uma atividade para os valores de um recipiente de entrada de uma outra atividade. Pode haver conectores de dados de uma atividade para ela mesma, permitindo que o processo interaja com a mesma atividade.

O FlowMark oferece **recuperação com avanço**, ou seja, depois de uma falha durante a execução de um processo, o FlowMark continua a execução a partir do ponto de falha. No caso de atividades que se encontram em execução no momento da falha, há vários casos a considerar. Se os clientes que executam as atividades não falharam, eles irão manter a informação sobre a execução até que o servidor se recupere e então enviam a informação normalmente. Se, durante o intervalo enquanto o servidor se recupera, o cliente também falha, a informação a respeito da execução das atividades é perdida. Os clientes do FlowMark não usam nenhum armazenamento estável para guardar os resultados da execução de uma atividade. Em tais casos, é responsabilidade do usuário resolver a situação terminando ou reiniciando as atividades. Por outro lado, o servidor continuará a achar que elas são atividades em progresso.

Como melhoria futura, o FlowMark oferecerá **recuperação com atraso**. Isto dará ao usuário a capacidade de pedir ao FlowMark que compense (ou seja, logicamente desfaça) as atividades já terminadas de um processo em execução.

O componente que gerencia a execução dos processos é o **Servidor FlowMark**. Originalmente, este servidor acessa um único banco de dados centralizado que armazena todas as informações a respeito dos processos. Com isso, tem-se um "gargalo" na performance do sistema, pois pode haver muitos processos em execução ao mesmo tempo, causando sobrecarga no banco de dados. Se o banco de dados falha, a execução dos processos no FlowMark é interrompida. O Exotica propõe duas soluções para resolver estes problemas. Nelas, o banco de dados centralizado é substituído por bancos de dados espalhados em diferentes máquinas.

1. O banco de dados centralizado é substituído por bancos de dados autônomos [3], que podem se comunicar entre si e trocar informações a respeito da execução dos processos e atividades. Nesta solução, a falha de um banco de dados não provoca a falha dos processos que utilizavam outros bancos de dados, ou seja, estes processos podem continuar sua execução normalmente;

2. Diversos grupos de servidores FlowMark são formados [7] de modo que todos os servidores pertencentes a um determinado grupo estão conectados a um mesmo

banco de dados. A falha em um banco de dados interrompe somente a execução dos processos pertencentes ao grupo correspondente. Os processos de outros grupos continuam sendo executados normalmente.

Estas soluções adotadas no projeto Exotica aumentam a escalabilidade do FlowMark.

Capítulo 4

O Sistema WorkToDo

O ***WorkToDo*** [1] é um Sistema de Gerenciamento de *Workflows* (SGWF) que inclui facilidades para a definição de processos e controle da execução de processos. A ênfase desse trabalho está no suporte a tarefas cooperativas e na execução confiável de processos mesmo na ocorrência de falhas.

O modelo de *workflow* é descrito na Seção 4.1. A linguagem de especificação é mostrada na Seção 4.2. A arquitetura do WorkToDo é apresentada na Seção 4.3. Uma comparação entre o WorkToDo e o Modelo de Referência para *Workflow* é apresentada na Seção 4.4. A Seção 4.5 mostra os aspectos relativos ao funcionamento do sistema. O gerenciamento de transações é apresentado na Seção 4.6. Finalmente, a Seção 4.7 apresenta o tratamento de falhas para os diversos componentes da arquitetura do WorkToDo.

4.1 Modelo de Workflow

Um ***Modelo de Workflow*** é utilizado para modelar um processo (*workflow*) do mundo real em termos computacionais. A modelagem de um processo consiste em definir os seguintes aspectos: o que deve ser executado, em qual momento, por quem e com quais dados. Esses aspectos são chamados, respectivamente, de funcional, comportamental, organizacional e informacional [15].

Aspecto Funcional - Especifica quais ações devem ser executadas pelos processos. O processo pode ser decomposto em atividades, que por sua vez podem ser decompostas em outras atividades até que as decomposições resultem apenas em ações elementares (tarefas).

Aspecto Comportamental - Especifica quando cada atividade deve ser executada, através de uma lista de interdependências entre as diversas atividades.

Aspecto Organizacional - Especifica quem é o responsável pela execução de cada atividade. O responsável, denominado entidade processadora, pode ser uma aplicação ou um ser humano (usuário).

Aspecto Informacional - Especifica os dados utilizados pelas atividades, assim como o fluxo de dados entre essas atividades. Uma atividade pode utilizar dados de entrada e gerar dados de saída, que por sua vez podem servir como dados de entrada para outras atividades, e assim sucessivamente.

A Figura 4.1 mostra uma representação do modelo utilizado pelo WorkToDo, em notação UML (*Unified Modeling Language*) [9]. Um processo é formado por um conjunto de atividades e dependências entre elas. Cada atividade possui um conjunto de dados de entrada e saída e consiste de uma ou mais tarefas. Cada tarefa, por sua vez, possui uma entidade processadora responsável por sua execução.

O WorkToDo utiliza o conceito de ***Tipos de Processo***. Quando se modela um processo, na verdade um tipo de processo é definido, podendo este ser instanciado diversas vezes e criando-se assim execuções particulares desse tipo.

A Tabela 4.1 expressa a associação existente entre a nomenclatura adotada nesta dissertação e a nomenclatura definida pela *Workflow Management Coalition* (*WfMC*) no Modelo de Referência para *Workflow* [26]. A nomenclatura adotada corresponde à encontrada em outros trabalhos [18].

WorkToDo	WfMC
Tipo de Processo	Definição de Processo
Instância de Processo	Instância de Processo
Atividade	Atividade
Tarefa	Atividade Atômica
Entidade Processadora	Participante de Processo

Tabela 4.1: Nomenclatura adotada no WorkToDo

4.1.1 Atividades

As atividades são as unidades que compõem os processos. A execução de um processo consiste na execução das atividades das quais é formado, definindo-se assim o aspecto funcional de um processo.

As atividades consistem de tarefas ou outras atividades (sub-atividades). As tarefas são ações elementares de um processo.

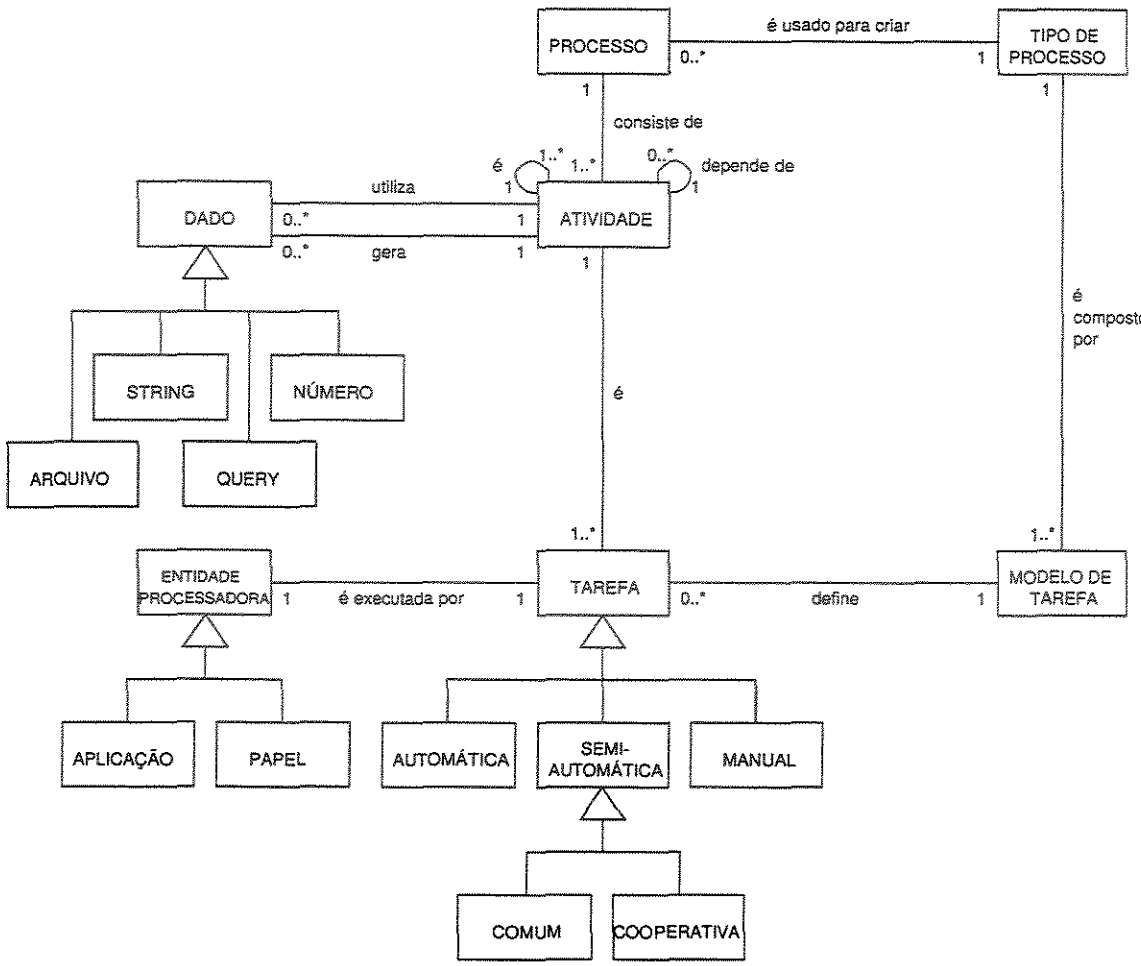

Figura 4.1: Meta-modelo de processo

Há um maior grau de reusabilidade no modelo pelo fato de uma atividade poder ser constituída de outras atividades, já que se torna desnecessária a redefinição das atividades que se enquadram nesse caso.

Estados das Atividades

Neste modelo, são definidos cinco estados possíveis de uma atividade:

`NOT_READY` - A atividade assume este estado quando não está pronta para ser executada, pois as condições necessárias para sua execução ainda não foram satisfeitas.

`READY` - A atividade assume este estado quando está pronta para ser executada, pois as condições necessárias para sua execução já foram satisfeitas. Essas condições são avaliadas usando-se uma árvore de dependências, explicada em 4.1.3.

`RUNNING` - Se a atividade for automática, ela assume este estado quando está em execução. Se ela for semi-automática ou manual, ela assume este estado quando foi selecionada por algum usuário do SGWF.

`SUCCEEDED` - A atividade assume este estado quando foi executada com sucesso.

`FAILED` - A atividade assume este estado quando ocorreu alguma falha durante sua execução.

A forma como se dá a transição entre os estados de uma atividade é mostrada na Figura 4.2. O estado inicial de uma atividade é o estado `NOT_READY`. Já os estados finais que uma atividade pode assumir são: `SUCCEEDED` e `FAILED`.

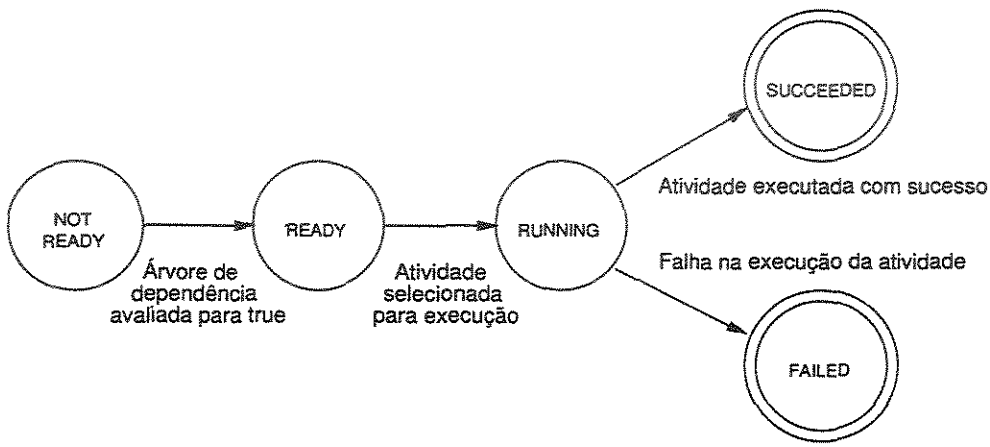

Figura 4.2: Diagrama de transição de estados de uma atividade

As atividades, em qualquer instante, assumem um e somente um destes estados.

No contexto desse trabalho, utilizaremos somente o conceito de **Tarefa**, definido a seguir.

4.1.2 Tarefas

As tarefas são as atividades que constituem ações elementares de um processo. As tarefas assumem um comportamento bem definido e explicíto, como por exemplo a composição de um arquivo texto ou o envio de um *e-mail*.

Para a definição de uma tarefa, é necessária a utilização de um ***Modelo de Tarefa***. O Modelo de Tarefa contém a especificação das características principais de uma tarefa, tais como o tipo da tarefa, sua prioridade e o papel de quem a executa. Através desse modelo, o processo pode instanciar as tarefas que serão executadas.

A reutilização do Modelo de Tarefa é possível, já que o mesmo pode ser utilizado em diversos processos distintos, assim como diversas vezes em um mesmo processo.

As tarefas podem utilizar e gerar diferentes tipos de dados, por isso os dados são especificados na definição do processo, permitindo que diferentes tipos de dados possam ser utilizados para cada instanciação de um modelo de tarefa.

Tipos de Tarefas

O modelo apresentado possui diferentes tipos de tarefas, que diferem umas das outras quanto à maneira como são executadas. O sistema trata as tarefas conforme o seu tipo, tornando necessária a classificação dessas tarefas nos seguintes tipos:

Automáticas - São as tarefas que são executadas por alguma entidade ou sistema de *software* que é invocado automaticamente pelo SGWF. Alguns exemplos de tarefas automáticas são: a ativação de um programa que atualiza um registro num banco de dados, a invocação de um *script* que copia arquivos para um diretório especificado ou a execução de um programa que realiza um cálculo matemático.

As tarefas automáticas se subdividem em:

Transacionais - São as tarefas que são executadas como transações, considerando-se as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Alguns exemplos de tarefas transacionais são: a atualização de um registro em um banco de dados ou uma operação de saque feita em uma conta corrente bancária.

Não-transacionais - São as tarefas que são executadas sem satisfazer as propriedades ACID. Alguns exemplos de tarefas não transacionais são: a cópia de um arquivo ou a listagem de um diretório.

Semi-automáticas - São as tarefas que são executadas por um ser humano auxiliado por uma entidade ou sistema de *software*. As ações e decisões referentes à tarefa são tomadas pelo usuário, que utiliza um *software* como ferramenta. Alguns exemplos de tarefas semi-automáticas são: o envio de um *e-mail*, a edição de um texto utilizando um editor de texto ou a criação de um gráfico utilizando uma planilha eletrônica.

Dentre as tarefas semi-automáticas estão as **Tarefas Cooperativas** executadas por um grupo de usuários. Um desses usuários é o coordenador da tarefa, encarregado de iniciar uma transação cooperativa dentro da qual os outros usuários do grupo podem criar subtransações.

Manuais - São as tarefas que são executadas por um ser humano, sem a utilização de uma entidade ou sistema de *software*. Alguns exemplos de tarefas manuais são: fazer um telefonema, preencher um formulário ou assinar um documento.

As tarefas automáticas são executadas pelo SGWF de forma direta, sem a interferência de um usuário, o que não ocorre com as tarefas semi-automáticas e manuais, que se caracterizam pela participação de usuários em sua execução.

4.1.3 Regras de Dependência

No WorkToDo, o aspecto comportamental do modelo de *workflow* é modelado através de regras de dependência.

Uma ***Regra de Dependência*** consiste em um conjunto de restrições que devem ser satisfeitas para a execução de uma tarefa. Tais restrições são descritas em termos de transições de tarefas para estados. Cada regra de dependência é composta por um ou mais termos. Cada termo é composto por uma identificação (nome) de tarefa T e por um estado E, representando assim a transição de T para E. Os termos podem ser relacionados através dos operadores booleanos `and` e `or`.

Um exemplo de regra de dependência é mostrado abaixo:

$$\texttt{and}\ (T_1 \rightarrow \texttt{SUCCEEDED}, T_2 \rightarrow \texttt{FAILED})$$

Neste exemplo, a tarefa a que se refere a regra de dependência será executada se e somente se a tarefa T_1 alcançar o estado `SUCCEEDED` e a tarefa T_2 alcançar o estado `FAILED`. Esta regra de dependência é composta pelos termos $T_1 \rightarrow$ `SUCCEEDED` e $T_2 \rightarrow$ `FAILED`, relacionados através do operador booleano `and`.

As regras de dependência podem ser construídas de forma mais complexa, como mostrado no exemplo abaixo:

$$\texttt{or}\ (\texttt{and}\ (T_1 \rightarrow \texttt{SUCCEEDED}, T_2 \rightarrow \texttt{FAILED}), T_3 \rightarrow \texttt{RUNNING})$$

Neste exemplo, a tarefa a que se refere a regra será executada se a tarefa T_1 alcançar o estado `SUCCEEDED` e a tarefa T_2 alcançar o estado `FAILED`, ou se a tarefa T_3 alcançar o estado `RUNNING`.

Uma regra de dependência é associada a uma tarefa e esta somente é executada quando a sua respectiva regra é avaliada para verdadeiro (`true`). Quando a regra de dependência é vazia, significa que a tarefa pode ser executada assim que a instância do processo que contém a tarefa em questão for criada.

Os estados válidos nos termos de uma regra são os mesmos descritos na Seção 4.1.1, ou seja, são os mesmos estados válidos para as atividades.

Através da regra de dependência, pode-se construir uma *Árvore de Dependência*. Nessa árvore, os nós intermediários constituem os operadores booleanos and e or e os nós folhas constituem os nomes das tarefas. Além do nome da tarefa e dos operadores, cada nó ainda possui um campo `valor`, contendo um booleano que inicialmente é `false`. Os nós folhas possuem também um campo `nome-de-estado`, contendo um nome de estado E associado à tarefa.

Ao ocorrer a transição de uma tarefa T para um estado E, o campo `valor` do nó folha correspondente é atualizado para `true`, indicando que a dependência deste nó (nó da tarefa T) foi satisfeita. O nó pai deste nó folha é então notificado desta atualização e pode avaliar sua expressão booleana (o nó pai é sempre um operador and ou or), levando em consideração os seus nós filhos. Caso a expressão seja avaliada para `true`, o nó pai altera seu próprio campo `valor` e notifica seu nó pai, que por sua vez repete o processo. Caso contrário, o campo `valor` do nó pai permanece da mesma forma (igual a `false`). Quando o nó raiz é notificado e altera o seu campo `valor` para `true`, a tarefa em questão está pronta para ser executada. É importante ressaltar que o campo `valor` de todos os nós da árvore contém inicialmente o valor `false`.

A Figura 4.3 mostra a árvore de dependências referente à primeira regra de dependência mostrada. Na regra, quando a tarefa T_1 alcança o estado `SUCCEEDED`, seu nó folha correspondente assume o valor `true`, notificando também o seu nó pai (nó and). O nó pai então verifica sua expressão equivalente (operação and aplicada aos dois termos contidos nos nós filhos), mas a expressão ainda não é verdadeira, já que o nó filho do lado direito da árvore ainda contém o valor `false`.

Caso a regra de dependência seja vazia, a árvore de dependência é sempre avaliada para verdadeiro.

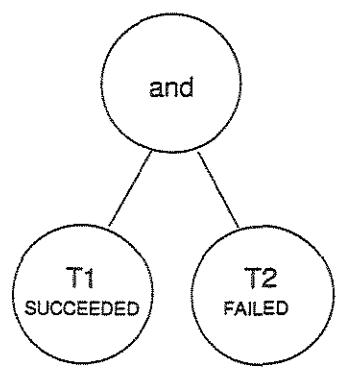

Figura 4.3: Exemplo de Árvore de Dependência

4.1.4 Entidades Processadoras

Uma ***Entidade Processadora*** é responsável pela execução de uma tarefa em algum momento.

O aspecto organizacional de um processo é modelado através das entidades processadoras, que são definidas nos modelos de tarefas. Uma entidade processadora pode estar presente em diversas localidades diferentes e pode ter zero ou mais tarefas atribuídas a ela, num dado momento.

O WorkToDo aborda dois tipos de entidades processadoras: ***Aplicações*** e ***Usuários***.

Aplicações

As aplicações são as entidades processadoras responsáveis pela execução das tarefas automáticas, ou seja, para executar uma tarefa automática, basta executar a aplicação associada a ela. Para isso, deve-se fornecer os dados de entrada necessários. Cada tarefa automática é composta por apenas uma aplicação.

As aplicações são definidas independentemente, assim como as tarefas. Por isso, quando um modelo de tarefa deseja utilizar uma aplicação, basta referenciá-la, sem ser necessário redefiní-la. A definição de uma aplicação contém as seguintes informações: nome do arquivo executável da aplicação, tamanho total desse arquivo, sistema operacional em que a aplicação será executada, entre outras informações.

Usuários

Os usuários são as entidades processadoras responsáveis pela execução das tarefas semi-automáticas ou manuais. Um sistema de papéis é utilizado para a atribuição de tarefas a usuários. Um ***Papel*** define um grupo de usuários com determinadas características, requisitos e/ou habilidades. Um usuário pode pertencer a um ou mais papéis e um papel pode conter um ou mais usuários. Um papel R, responsável pela execução de uma tarefa, é indicado na especificação da mesma. Com isso, qualquer usuário pertencente ao papel R pode executar a tarefa. Tal fato representa uma melhoria na eficiência do sistema, já que a tarefa pode ser executada tão logo esteja pronta para ser executada e exista algum usuário do papel R conectado ao sistema.

Apesar de as tarefas semi-automáticas serem caracterizadas pela utilização de uma aplicação por parte dos usuários para a execução de uma tarefa, as aplicações não são consideradas entidades processadoras no caso desse tipo de tarefa e sim, os usuários.

4.1.5 Dados

Os dados representam o aspecto informacional dos processos. As tarefas podem utilizar dados de entrada e gerar dados de saída. No WorkToDo, os dados podem ser de quatro tipos:

Arquivos - São dados armazenados em disco. Um arquivo possui um nome e um tamanho associados a ele.

Queries - São expressões em SQL (*Simple Query Language*) que definem consultas a bancos de dados e que, possivelmente, retornam como resultado um conjunto de dados (registros). Uma *query* possui um banco de dados e uma expressão em SQL associados a ela.

Strings - São dados compostos por valores alfanuméricos.

Números - São dados compostos por valores inteiros ou reais.

Quando um processo está em execução, ele armazena seus dados em uma área exclusiva, chamada ***Contexto do Processo***. Essa área é acessada quando uma tarefa precisa armazenar ou recuperar dados. O contexto de uma instância de processo P é acessível somente às tarefas associadas a P, garantindo-se assim que a execução de um processo não interfira na execução de um outro, pois os dados de cada instância de processo estão isolados.

4.2 Linguagem de Especificação

A especificação de um processo no WorkToDo se dá através de uma ***Linguagem de Definição de Processo*** (***Process Definition Language - PDL***). Com o uso desta linguagem, pode-se especificar tipos de processos, modelos de tarefas e aplicações. A PDL é descrita nas próximas seções e a gramática utilizada na linguagem é mostrada no Apêndice A.

4.2.1 Tipos de Processo

A definição de um tipo de processo é composta por duas estruturas, que são:

- Um bloco contendo a definição dos dados que serão utilizados na execução das tarefas e;

- Um bloco contendo a definição das tarefas que serão desempenhadas durante a execução do processo. Para cada tarefa, são descritos os dados que ela utiliza (seus dados de entrada e saída), as condições necessárias para a sua execução (suas dependências) e a entidade que deverá executá-la (sua entidade processadora).

A especificação de um tipo de processo é mostrada abaixo:

```
WORKFLOW <workflow-id> {
    FILE <arquivo-id> {
        NAME <nome-do-arquivo>;
    }
    :
    QUERY <query-id> {
        DATABASE <nome-do-BD>;
        EXPRESSION <expressão-SQL>;
    }
    :
    STRING <string-id> {
        VALUE <valor>;
    }
    :
    NUMBER <número-id> {
        VALUE <valor>;
    }
    :
    TASK <tarefa-id>:<modelo-de-tarefa> {
        DEPENDS <regra-de-dependência>;
        IN_CONTEXT <lista-de-dados>;
        OUT_CONTEXT <lista-de-dados>;
    }
    :
}
```

O `workflow-id` é a identificação do tipo de processo e será parte integrante do nome de uma instância de processo. As cláusulas `FILE`, `QUERY`, `STRING` e `NUMBER` possibilitam a definição dos dados de entrada e saída das tarefas. Um tipo de processo pode conter zero ou mais dessas cláusulas.

A cláusula FILE define um arquivo identificado pelo arquivo-id. As referências a um arquivo utilizado por uma tarefa devem ser feitas através desse identificador. O arquivo físico, bem como o caminho onde se encontra, é indicado por nome-do-arquivo.

A cláusula QUERY define uma operação a ser realizada em um banco de dados e é identificada pelo query-id. Na cláusula DATABASE é identificado o nome do banco de dados e na cláusula EXPRESSION especifica-se a expressão SQL correspondente à consulta.

A cláusula STRING define uma variável que armazena valores alfanuméricos e é identificada pelo string-id.

A cláusula NUMBER define uma variável que armazena valores numéricos, que podem ser inteiros ou reais e é identificada pelo número-id.

O tarefa-id é a identificação de uma tarefa que está contida em um tipo de processo e está vinculado à cláusula TASK. Este identificador é utilizado nas regras de dependência que referenciam a tarefa correspondente. O modelo-de-tarefa faz menção a um modelo de tarefa existente. Nesse modelo, estão detalhadas as informações referentes ao modelo de tarefa em questão. O operador ":" estabelece a instanciação de um modelo de tarefa a ser utilizado na execução do processo.

As dependências entre tarefas são definidas na cláusula DEPENDS. A regra-de-dependência é descrita na Seção 4.1.3.

As cláusulas IN_CONTEXT e OUT_CONTEXT contêm, respectivamente, os dados utilizados como entrada para uma tarefa e os dados que essa tarefa gera como saída. Em ambas as cláusulas, lista-de-dados é uma lista de arquivo-id's, query-id's, string-id's e/ou número-id's. Esses identificadores são separados por vírgulas. As cláusulas IN_CONTEXT e OUT_CONTEXT podem não existir. Isso significa que a tarefa não utiliza dados de entrada ou não gera dados de saída. Essas cláusulas também especificam o fluxo de dados entre as tarefas, já que identificam os dados de entrada e saída de cada tarefa.

4.2.2 Modelos de Tarefa

Um modelo de tarefa especifica as características de um tipo de tarefa. Assim sendo, ao instanciar-se um modelo de tarefa, as principais características de uma tarefa já estão definidas, sem que seja necessário repetir a especificação da tarefa.

```
TASK <modelo-de-tarefa-id> {
     TYPE <nome-do-tipo>;
     ROLE <nome-do-papel>;
     DESCRIPTION <descrição-textual>;
     APPLICATION <aplicação-id>;
     PRIORITY <número-de-prioridade>;
     DEADLINE <prazo> <tipo-de-prazo>;
```

```
        DISCONNECTED_OPERATION <permissão>;
        RETRIES <número-de-tentativas>;
        USERS <lista-de-usuários>;
}
```

O `modelo-de-tarefa-id` é a identificação do modelo de tarefa e as definições de processo que instanciam esse modelo o referenciam por esse identificador.

A cláusula TYPE contém a definição do tipo de tarefa, conforme descrito na Seção 4.1.2. A execução da tarefa será gerenciada conforme a definição do tipo da tarefa.

A cláusula ROLE identifica, através do `nome-do-papel`, o papel responsável pela execução da tarefa. Tal identificação restringe a execução da tarefa aos usuários pertencentes a esse papel. No caso de tarefas automáticas, a cláusula ROLE é desconsiderada, ou seja, o `nome-do-papel` é vazio.

Para descrever textualmente a tarefa, utiliza-se a cláusula DESCRIPTION. Nela, pode-se informar a finalidade da tarefa, sua forma de execução, enfim, quaisquer informações que forem consideradas relevantes. A `descrição-textual` pode ser vazia.

A cláusula APPLICATION contém a identificação da aplicação que será invocada no momento da execução da tarefa. A cada tarefa está relacionada uma e somente uma aplicação. Caso a tarefa seja do tipo manual, nenhuma aplicação deve ser especificada.

A cláusula PRIORITY contém, em `número-de-prioridade`, a definição da prioridade da tarefa. No WorkToDo, as tarefas automáticas com maior prioridade são executadas antes das tarefas com menor prioridade.

A cláusula DEADLINE indica o prazo máximo para a execução de uma tarefa. O `tipo-de-prazo` pode ser especificado em `dias` ou `horas`.

A cláusula DISCONNECTED_OPERATION indica se a tarefa pode (`permissão` é igual a `true`) ou não (`permissão` é igual a `false`) ser executada em modo desconectado. Caso a tarefa seja automática, esta cláusula não assume nenhum valor.

A cláusula RETRIES indica o número de vezes que uma tarefa automática deve ser re-executada, caso ela falhe. O `número-de-tentativas` é decrementado a cada re-execução e quando atinge o valor zero, a execução da tarefa é dada como falha. O usuário responsável pelo *workflow* é notificado em sua lista de mensagens de que a tarefa em questão falhou, mesmo após um número arbitrário de re-execuções. Com isso, este usuário pode tomar as providências cabíveis em relação a este resultado.

Por fim, a cláusula USERS, válida somente para tarefas semi-automáticas cooperativas, contém a `lista-de-usuários` que farão parte do grupo que irá colaborar na execução da tarefa. Caso a tarefa não seja semi-automática cooperativa, a `lista-de-usuários` deve ser vazia.

4.2.3 Aplicações

A definição de uma aplicação é composta pelas características de uma aplicação. Tal definição é utilizada por um modelo de tarefa, sem que seja necessário repetir a definição da aplicação.

```
APPLICATION <aplicação-id> {
    FILENAME <nome-do-arquivo>;
    SIZE <tamanho-da-aplicação>;
    OS <sistema-operacional>;
    CPU <processador>;
    INSTALLER <nome-do-instalador>;
    HOSTS <lista-de-hosts>;
}
```

A cláusula `APPLICATION` contém o nome da aplicação em `aplicação-id`, que é o nome pelo qual os modelos de tarefa referenciam a aplicação.

A cláusula `FILENAME` contém o nome do arquivo executável da aplicação, bem como o seu caminho completo, em `nome-do-arquivo`.

A cláusula `SIZE` contém a informação do `tamanho-da-aplicação`. Tal informação corresponde ao tamanho total de todos os arquivos utilizados pela aplicação.

A cláusula `OS` especifica o `sistema-operacional` em que a aplicação deve ser executada.

A cláusula `CPU` indica o tipo de `processador` em que a aplicação deve ser executada. Por exemplo: Pentium MMX.

A cláusula `INSTALLER` contém o nome do arquivo de instalação da aplicação, caso exista, em `nome-do-instalador`. É necessário especificar o caminho completo deste arquivo. O instalador fornece todos os arquivos necessários à execução da aplicação. Caso o instalador seja omitido, assume-se que o arquivo executável da aplicação seja suficiente para a execução da mesma.

A cláusula `HOSTS` contém uma lista de nomes de máquinas (*hosts*) em que a aplicação pode ser executada. É necessário especificar pelo menos um nome de máquina. A especificação de mais de um nome de máquina é uma forma de garantir a execução da aplicação, mesmo que uma máquina falhe.

4.3 Arquitetura do WorkToDo

O WorkToDo possui uma arquitetura distribuída, como é mostrado na Figura 4.4, e os componentes pelos quais é formada são descritos detalhadamente a seguir.

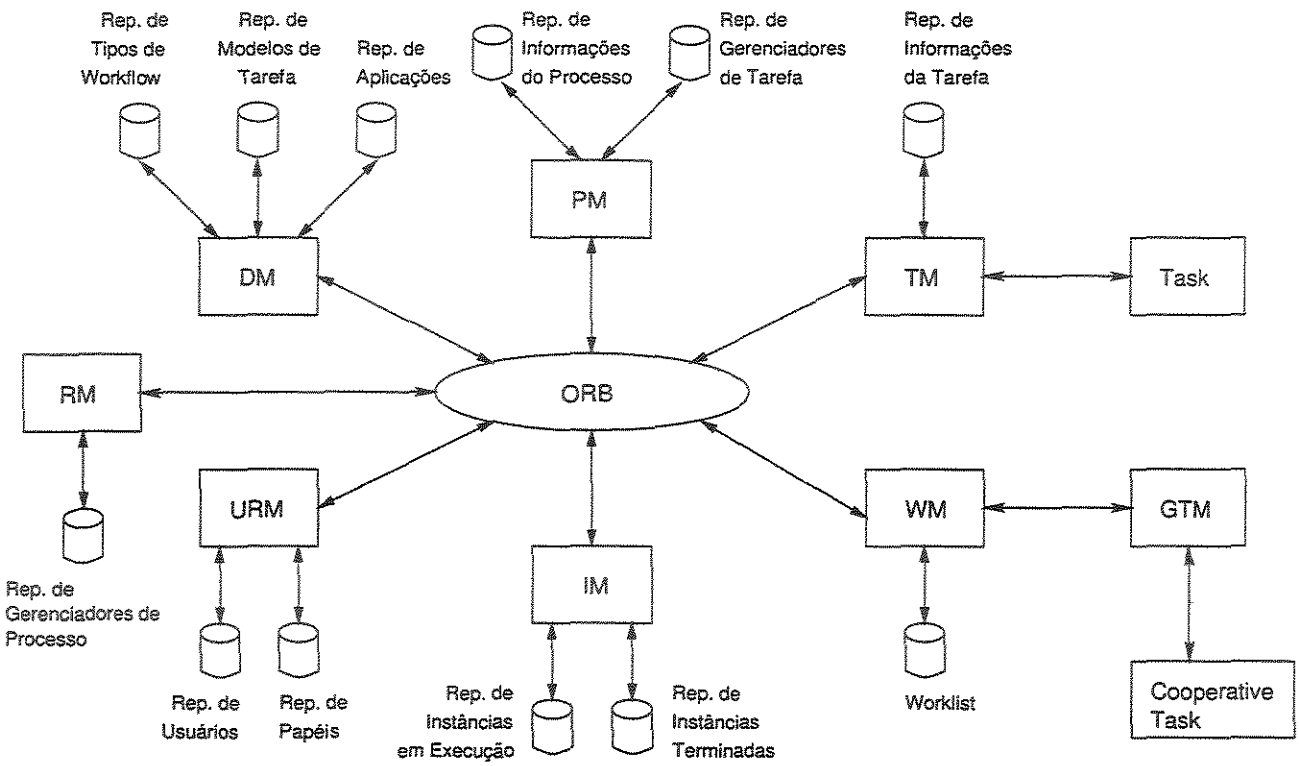

Figura 4.4: Arquitetura do WorkToDo

Cada componente da arquitetura do WorkToDo desempenha uma função primordial, conforme descrito abaixo:

- O Gerenciador de Usuários e Papéis (*User and Role Manager - URM*) gerencia os usuários e papéis existentes, bem como a lista de usuários pertencentes a cada papel;
- O Gerenciador de Definições (*Definition Manager - DM*) cuida do acesso às definições de processos, tarefas e aplicações;
- O Gerenciador de Instâncias (*Instance Manager - IM*) trata das informações referentes às instâncias de processo que estão em execução e que já terminaram de executar;
- O Gerenciador de Processo (*Process Manager - PM*) coordena a execução de uma determinada instância de processo;
- O Gerenciador de Tarefa (*Task Manager - TM*) controla a execução de uma tarefa automática;

- O Gerenciador de Lista de Trabalho (*Worklist Manager - WM*) controla uma lista com as tarefas que podem ser executadas por um determinado usuário. Ele controla também a execução das tarefas semi-automáticas e manuais;
- O Gerenciador de Recuperação (*Recovery Manager - RM*) monitora os gerenciadores de processo e recupera os que não estão ativos e;
- O Gerenciador de Tarefas Cooperativas (*Group Transaction Manager - GTM*) gerencia a execução de uma tarefa cooperativa.

Cada componente pode ser executado em uma máquina diferente, portanto é necessária uma infra-estrutura que realize a comunicação remota entre tais componentes, como *Sockets*, RPC (*Remote Procedure Call*), CORBA (*Common Object Request Broker Architecture*) ou RMI (*Remote Method Invocation*). A plataforma utilizada no WorkToDo é baseada em CORBA [21], utilizando a camada ORB para comunicação entre os componentes distribuídos.

A escolha de uma arquitetura distribuída tem o propósito principal de impedir a sobrecarga de processamento em uma determinada máquina. Além disso, tal escolha propicia maior tolerância a falhas, já que a falha de um determinada máquina não leva à queda de todo o sistema, que continua funcionando normalmente. Somente os componentes executados na máquina que falhou se tornam inacessíveis pelo sistema.

4.3.1 Gerenciador de Usuários e Papéis

O Gerenciador de Usuários e Papéis (*User and Role Manager - URM*) é o componente responsável pelo gerenciamento dos usuários e papéis do sistema. O URM é o componente que efetua a adição e remoção de usuários e papéis dentro do sistema, verifica a quais papéis um determinado usuário pertence e verifica quais usuários pertencem a um determinado papel. Para realizar estas operações, ele dispõe de dois repositórios, que são:

Repositório de Usuários - Este repositório armazena todas as informações relevantes a respeito dos usuários do sistema: nome, senha, situação (conectado ou não ao SGWF), endereço IP de sua máquina, qual o tempo em que está conectado ao SGWF e quando foi a sua última conexão ao SGWF. Além destas informações, este repositório também armazena uma lista de mensagens para cada usuário do SGWF, como será mostrado na Seção 4.5.4.

Repositório de Papéis - Este repositório armazena as informações referentes aos papéis existentes no SGWF. Além destas informações, armazena também a lista de usuários pertencentes a um determinado papel. Um usuário pode pertencer a zero ou mais papéis dentro do SGWF.

Sempre que um componente do sistema precisa acessar ou modificar as informações referentes a algum usuário ou papel do SGWF, ele o faz através de uma requisição ao URM, que por sua vez consulta seus repositórios, conforme a requisição. O URM contém operações básicas sobre seus repositórios, tais como `add()`, `remove()` e `list()`. A relação das operações disponíveis no URM pode ser encontrada no Apêndice B.

4.3.2 Gerenciador de Definições

O Gerenciador de Definições (*Definition Manager - DM*) é o componente responsável pelo gerenciamento das definições de processos, tarefas e aplicações. O DM é o componente que efetua a adição e remoção de definições e as recupera quando necessário. Para realizar estas operações, ele dispõe de três repositórios que contêm informações sobre os tipos de processos (*workflows*), os modelos de tarefas e as definições de aplicações. O DM contém operações do tipo `add()`, `remove()` e `list()` sobre seus repositórios. A relação das operações disponíveis no DM pode ser encontrada no Apêndice B.

Quando um tipo de processo é inserido no SGWF, o DM requisita o nome de um papel, que é chamado ***Papel Criador*** do tipo de processo. Esta informação é muito importante, pois somente usuários deste papel poderão criar instâncias deste tipo de processo. Portanto, a cada tipo de processo está associado um papel criador.

4.3.3 Gerenciador de Instâncias

O Gerenciador de Instâncias (*Instance Manager - IM*) é o componente responsável por manter um registro das instâncias de processo que estão em execução e das que já terminaram de executar. O IM também é responsável por criar um Gerenciador de Processo (descrito na Seção 4.3.4) para cada instância criada.

O IM não controla a execução das instâncias de processo. Ele transfere para o Gerenciador de Processo uma cópia da definição do processo, cuja instância foi criada.

Para realizar estas operações, o IM dispõe de dois repositórios, que são:

Repositório de Instâncias em Execução - Este repositório armazena informações a respeito das instâncias em execução, tais como o nome, o tipo e a máquina onde a instância está sendo executada.

Repositório de Instâncias Terminadas - Este repositório armazena informações a respeito das instâncias que já terminaram a sua execução. Além destas informações, armazena também o estado final do processo, a data e a hora de início e término da execução e o estado final de cada tarefa vinculada ao processo.

Sempre que um componente do sistema precisa recuperar ou atualizar as informações de uma instância de processo, ele consulta o IM para saber a máquina (*host*) a partir da qual a instância está sendo gerenciada. Pode-se verificar também, através do IM, quais são as instâncias de processo em execução e qual o estado das instâncias de processo terminadas. Para atender a tais requisições, o IM consulta seus repositórios correspondentes. O IM contém operações do tipo `add()`, `remove()` e `list()` sobre seus repositórios. A relação das operações disponíveis no IM pode ser encontrada no Apêndice B.

O usuário que cria uma instância de processo é o ***Usuário Responsável*** pelo *workflow*. No caso de uma tarefa cooperativa, este mesmo usuário é o ***Coordenador*** da transação de grupo.

4.3.4 Gerenciador de Processo

O Gerenciador de Processo (*Process Manager - PM*) é o componente responsável por coordenar a execução de um processo. O PM ainda verifica quais tarefas estão prontas para serem executadas, inicia sua execução e obtém seus resultados, verificando se tais resultados disparam a execução de novas tarefas. Conforme mencionado anteriormente, existe um PM para cada instância em execução.

O PM mantém uma lista de tarefas que compõem o processo, denominada ***Task List***. A *task list* contém as informações necessárias à execução das tarefas. Tais informações são: nome, tipo, entidade processadora, estado atual e dados utilizados como entrada e saída. Este conjunto de informações é denominado ***Task Definition***. Uma *task list* é composta, portanto, por uma lista de *task definitions*. As estruturas que constituem uma *task list* e uma *task definition* são mostradas na Figura 4.5.

A *task definition* de uma tarefa T possui, além dos campos contidos nos tipos de tarefa (tipo, prioridade, prazo), os seguintes campos:

Contexto de Entrada - Consiste de uma lista dos dados que são utilizados por T. O contexto de entrada é definido na cláusula `IN_CONTEXT` da definição do tipo de processo.

Contexto de Saída - Consiste de uma lista dos dados que são gerados como resultado da execução de T. O contexto de saída é definido na cláusula `OUT_CONTEXT` da definição do tipo de processo.

Árvore de Dependência - Consiste da árvore de dependência de T. Esta árvore é descrita na Seção 4.1.3.

Dependentes - Consiste de uma lista das tarefas que dependem de T. A lista possui a seguinte estrutura: para cada estado que T pode assumir, é mantida uma lista

contendo as tarefas do *workflow* que dependem da transição de T para aquele estado. Exemplificando, a tarefa T possui, em seu estado `READY`, todas as tarefas que contêm em sua árvore de dependência a transição de T para o estado `READY`. Essa lista de dependentes é utilizada para que, quando T atingir um estado E da lista, as tarefas que dependem da transição de T para E sejam notificadas de tal transição.

Estado - Consiste do estado corrente da tarefa T. Os possíveis estados de uma tarefa estão descritos na Seção 4.1.1.

Usuário - Consiste do nome do usuário que selecionou a tarefa T para execução, caso T seja semi-automática ou manual.

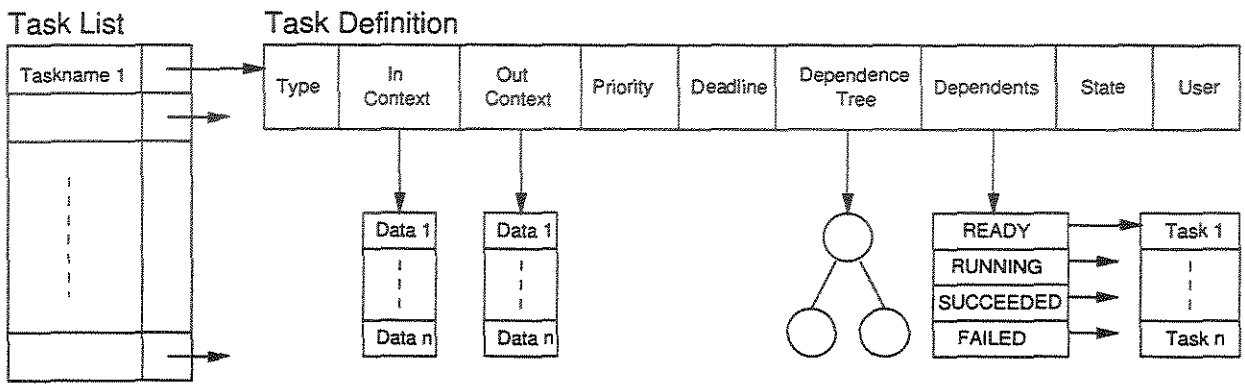

Figura 4.5: Representação da *Task List* e da *Task Definition*

Pode-se deduzir que, quando não há mais nenhuma tarefa nos estados `READY` ou `RUNNING`, o processo já terminou de ser executado. Para que o processo seja concluído, não é necessário que todas as suas tarefas tenham sido executadas, pois uma tarefa pode nunca ter as condições para sua execução satisfeitas, ou seja, nunca sua árvore de dependência é avaliada para verdadeiro. Assim que um processo termina de ser executado, seu estado final é armazenado no Repositório de Instâncias Terminadas do Gerenciador de Instâncias.

Ao ser criado o PM, a definição do tipo de processo é interpretada e é construída uma *task list*. Para interpretar a definição do tipo de processo, o PM utiliza um interpretador da PDL (descrita na Seção 4.2), denominado ***Interpretador da Linguagem de Definição de Processo*** (***Process Definition Language Interpreter - PDLI***) que constrói a *task list* da instância de processo a partir do tipo de processo e das tarefas e aplicações que o compõem. O PDLI constrói ainda uma *task definition* para cada tarefa e estrutura a *task list* do processo a partir de todas as *task definitions* das tarefas desse processo.

O PM mantém um repositório que armazena o estado da instância de processo (*workflow*), onde estão contidas informações tais como: o nome do processo, o horário de início

e término de sua execução, e sua *task list* correspondente. Com isso, pode-se consultar tanto o estado do processo, como o estado de cada tarefa que compõe o processo individualmente.

A relação das operações disponíveis no PM pode ser encontrada no Apêndice B.

O PM possui dois subcomponentes que o auxiliam no gerenciamento das instâncias de processo. São eles o ***Escalonador*** e o ***Despachante*** e são descritos a seguir.

Escalonador

O Escalonador (*Scheduler*) avalia as condições para a execução das tarefas através das árvores de dependência dessas tarefas. Portanto, ele decide quais tarefas podem ser executadas num determinado instante.

Sempre que uma tarefa T sofre uma transição para um estado E, o Escalonador reavalia a árvore de dependência das tarefas que dependem da transição $T \rightarrow E$, ou seja, das tarefas que faziam parte da lista de Dependentes de T. Quando a árvore de dependência de T é avaliada para `true`, o estado de T é alterado para `READY`, fazendo com que a tarefa T esteja pronta para ser executada.

Quando o Escalonador altera o estado de uma tarefa para `READY`, os dados necessários para a execução da tarefa são copiados para a área do processo, caso existam. Com isso, a tarefa pode acessar esses dados durante sua execução.

Assim que uma instância de processo é criada, o Escalonador altera o estado de todas as tarefas cuja árvore de dependência é vazia para `READY`.

Despachante

O Despachante (*Dispatcher*) prepara as tarefas para execução. Apesar disso, não é ele quem executa as tarefas; ele apenas provê as condições necessárias para a execução das mesmas.

O Despachante realiza uma verificação periódica na *task list* a fim de determinar quais tarefas estão prontas para executar. Ao encontrar tais tarefas, executa uma ação conforme o tipo da tarefa. As ações executadas são as seguintes:

1. Se a tarefa for **automática**, ele cria um Gerenciador de Tarefa para controlar sua execução. O Gerenciador de Tarefa, descrito na próxima Seção, é criado em um dos *hosts* indicados na definição da aplicação referente à tarefa.

2. Caso a tarefa seja **semi-automática** ou **manual**, ele notifica os usuários pertencentes ao papel da tarefa a respeito de sua disponibilidade para execução. A escolha dos usuários a serem notificados se dá de acordo com uma ***Política de Notificação***

que pode ser, por exemplo, todos os usuários do papel, ou os usuários com menos tarefas selecionadas.

Quando a tarefa é **cooperativa**, a escolha dos usuários a serem notificados se dá de acordo com o grupo de usuários pertencentes à tarefa cooperativa. Estes usuários são notificados sobre o início da tarefa cooperativa no instante em que o usuário coordenador do grupo, que é o usuário responsável pelo processo, seleciona a tarefa cooperativa para execução. O usuário responsável pelo processo é aquele que cria a instância de processo. Uma descrição mais detalhada sobre este tipo de usuário é dada na Seção 4.5.1.

4.3.5 Gerenciador de Tarefa

O Gerenciador de Tarefa (*Task Manager - TM*) é o componente responsável por controlar a execução de uma tarefa automática.

O TM é criado por um PM e inicializado com a *task definition*, a partir da qual invoca a aplicação responsável pela execução da tarefa, passando os dados referentes à tarefa. Após o início da execução da tarefa, o TM a monitora. Quando a execução é terminada, o TM notifica o PM a que está vinculado informando o tipo de término da tarefa, ou seja, se foi com sucesso ou com falha.

Caso a execução de uma tarefa termine com falha, o TM pode re-executá-la um número arbitrário de vezes (definido na especificação da tarefa, na cláusula `RETRIES`) até terminar com sucesso. Se após as tentativas, a execução ainda falhar, o usuário responsável pela instância do processo é notificado em sua lista de mensagens (descrita na Seção 4.5.4) sobre o término com falha da tarefa, para que tome as devidas providências.

A relação das operações disponíveis no TM pode ser encontrada no Apêndice B.

4.3.6 Gerenciador de Lista de Trabalho

O Gerenciador de Lista de Trabalho (*Worklist Manager - WM*) é o componente responsável por controlar e manter uma lista com as tarefas que podem ser executadas por um determinado usuário. Essa lista recebe o nome de ***Worklist*** (*Lista de Trabalho*) e as tarefas são denominadas ***Workitems*** (*Itens de Trabalho*).

Cada *workitem* corresponde a uma tarefa de um processo. Assim sendo, sempre que um *workitem* sofre alteração, o estado da tarefa correspondente também é atualizado. Tal associação garante que as alterações efetuadas pelos usuários sobre os *workitems* se reflitam nos processos. Os *workitems* correspondem sempre a tarefas semi-automáticas ou manuais, já que as automáticas são tratadas de forma distinta (através de TMs) e não são inseridas na *worklist*.

As estruturas de uma *worklist* e de um *workitem* são mostradas na Figura 4.6. Essas estruturas são similares à *task list* e à *task definition*, respectivamente. A diferença está no fato de que os primeiros se referem às tarefas de um usuário e os últimos às tarefas de uma instância de processo.

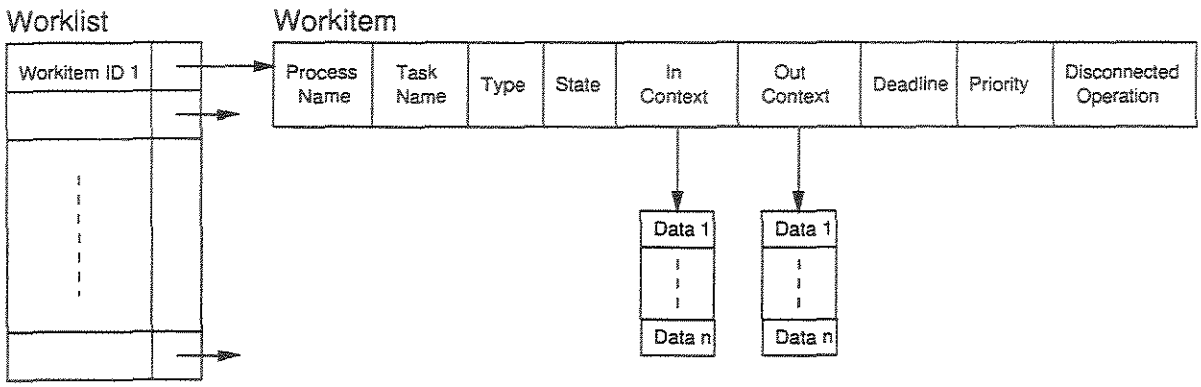

Figura 4.6: Representação da *Worklist* e do *Workitem*

Os campos `Task Name` e `Process Name` indicam, respectivamente, o nome da tarefa correspondente ao *workitem* e o nome da instância de processo ao qual a tarefa pertence. Os demais campos são similares aos existentes na *task definition* e estão definidos na Seção 4.3.4.

A *worklist* é armazenada no URM, mais especificamente no Repositório de Usuários. Sempre que um WM é ativado, ele restaura uma cópia da *worklist* do seu respectivo usuário através de uma requisição ao URM. Assim sendo, antes de o usuário se desconectar do SGWF, a *worklist* é atualizada no URM, pois seus campos podem ter sido alterados. Isso garante que, mesmo que um usuário se conecte ao SGWF através de diferentes máquinas, sua *worklist* sempre esteja disponível e atualizada.

Para cada usuário do sistema existe um e somente um WM. O WM é executado sempre na máquina onde o seu usuário correspondente está.

4.3.7 Gerenciador de Recuperação

O Gerenciador de Recuperação (*Recovery Manager - RM*) é o componente responsável por testar periodicamente se os PMs estão ativos. Para isso, cada PM criado é registrado no RM. Caso algum PM falhe, o RM o reinicializa em outra máquina com o último estado consistente antes de a falha ocorrer. Quando o processo termina sua execução, o registro do PM correspondente é retirado da lista do RM.

Para realizar tais operações, o RM utiliza um ***Repositório de Gerenciadores de Processo***, que armazena as informações relevantes do processo de forma persistente, tais como nome, máquina onde foi criado e endereço IP dessa máquina.

4.3.8 Gerenciador de Tarefas Cooperativas

Uma tarefa cooperativa é realizada por um grupo de usuários especificado na definição da tarefa. Esse tipo de tarefa é estruturada como uma hierarquia de transações, formando uma árvore na qual o nó raiz e os nós internos são transações de grupo e os nós-folha são transações de usuário.

A tarefa cooperativa é iniciada pelo usuário coordenador. Nesse momento, é criado o Gerenciador de Transação de Grupo (*Group Transaction Manager - GTM*) e os outros usuários do grupo recebem uma notificação através de suas listas de trabalho sobre o início da tarefa cooperativa podendo, então, criar subtransações dentro dessa transação de grupo.

A cada transação pertencente à hierarquia está associado um espaço de trabalho para armazenamento e manipulação de objetos. Portanto, paralelamente à hierarquia de transações existe uma hierarquia de áreas de trabalho para onde os objetos bloqueados[1] pela transação são copiados. Assim sendo, podem existir várias cópias de um objeto para cada nível de transação que requisitou tranca sobre o objeto.

No espaço de trabalho, os objetos podem ser copiados a partir de áreas de trabalho pertencentes a níveis mais altos da hierarquia. Embora os usuários tenham seus espaços de trabalho privados, onde o trabalho individual pode ser desenvolvido, os resultados intermediários podem ser observados e trocados entre os usuários do grupo facilitando, assim, o desenvolvimento da tarefa cooperativa [38]. O serviço para gerenciamento de transações [38] permite a cooperação através da invocação de operações para a obtenção de resultados intermediários de um usuário e da liberação de resultados para o espaço de trabalho do grupo antes do término da transação.

4.4 Comparação entre o WorkToDo e o Modelo de Referência para Workflow

O Modelo WorkToDo, assim como o Modelo de Referência para *Workflow* [26] possui funções que dizem respeito à definição e modelagem de um processo (através do **Tipo de Processo**, comum a ambos os modelos), bem como das tarefas que o compõem (através do **Modelo de Tarefa**).

O WorkToDo também se preocupa com a interpretação e execução de processos, assim como com a coordenação e escalonamento da execução das tarefas de um processo, através do componente Gerenciador de Processo (Ver Seção 4.3.4). No Modelo de Referência para *Workflow* (Ver Seção 2.1.5), estas funções são providas pela **Máquina de *Workflow***.

[1] Do termo em inglês *locked*

O SGWF WorkToDo ainda interage com usuários e também com outras aplicações, para efetuar o processamento das tarefas através do Gerenciador de Lista de Trabalho (WM) e do Gerenciador de Tarefa (TM), respectivamente. No Modelo de Referência para *Workflow*, essas características estão contidas nas **Aplicações Cliente de *Workflow*** e **Aplicações Invocadas**.

Além disso, o modelo WorkToDo também define uma arquitetura básica, identificando seus componentes principais e suas respectivas funcionalidades. Um conjunto de interfaces entre os componentes também é definido no WorkToDo.

4.5 Funcionalidades Básicas

Nesta Seção, são descritas as funcionalidades básicas do sistema WorkToDo. No final da Seção, são apresentados alguns diagramas de seqüência referentes a algumas operações realizadas no WorkToDo.

Usuários

No WorkToDo, os usuários podem estar ***ativos*** ou ***inativos***. Um usuário está ativo quando está conectado ao SGWF, podendo receber notificações de tarefas prontas para executar, selecionar tarefas para execução, criar processos, consultar os estados dos processos, entre outras operações. Um usuário está inativo quando não está conectado ao SGWF, portanto ele não é capaz de interagir com os componentes do SGWF que exigem chamadas remotas [39].

Os usuários do SGWF podem ainda assumir alguns papéis, que são:

1. **Administrador** – É o papel assumido por um ou mais usuários, que permite que se tenha alguns privilégios com relação aos demais usuários.

   O usuário administrador, que possui privilégios administrativos e gerenciais, é capaz de:

   - Incluir, remover e consultar usuários e papéis;
   - Incluir/remover usuários em/de papéis;
   - Verificar quais são os usuários pertencentes a cada papel;
   - Verificar quais são os papéis que um determinado usuário assume no SGWF;
   - Verificar quais são os usuários ativos num determinado momento;
   - Verificar a quantidade de itens atribuídos e selecionados por um usuário;

- Incluir, remover, consultar e modificar tipos de processo, modelos de tarefas e aplicações;
- Consultar o estado de todos os processos. Para isso, cada processo pode estar tanto em execução, como já ter sido terminado.

2. **Participante** – É o papel assumido pelos usuários que participam da execução de um *workflow.*

3. **Criador ou Responsável pelo Processo** – É o usuário que cria uma instância de processo. O SGWF assume automaticamente o usuário criador da instância como sendo o usuário responsável pelo processo. Esse usuário deve acompanhar o andamento do processo e tomar certas decisões quando for necessário. O sistema notifica esse usuário sobre os prazos para a execução das tarefas, sobre as tarefas que falharam em sua execução e sobre o início e término da execução de um processo. Esse usuário é responsável por tomar as medidas adequadas quanto a essas informações.

   O usuário responsável pelo processo também é capaz de tomar decisões a respeito das tarefas cuja execução falhou e cuja especificação do processo não define uma ação a ser tomada nesse caso. Assim sendo, ele pode, por exemplo, cancelar o processo. O cancelamento implica em terminar de executar somente as tarefas que estão sendo executadas, ou seja, que estão no estado `RUNNING` no ato do cancelamento. Qualquer outra tarefa não será executada, uma vez que o Gerenciador de Processo será terminado.

   Além disso, este tipo de usuário é quem inicia a execução de uma transação de grupo, assim que seleciona uma tarefa cooperativa para execução. Assim que a transação de grupo termina, ele informa ao SGWF qual o tipo de término da tarefa cooperativa (`SUCCEEDED` ou `FAILED`).

4.5.1 Interação com os Usuários

A interação dos usuários com o SGWF se dá através de uma interface, que fornece mecanismos de comunicação com os componentes do sistema. A interação de cada usuário com o SGWF ocorre principalmente por meio de sua *worklist*, através da qual o usuário pode realizar as seguintes ações:

1. **Visualizar *workitems*** - A *worklist* exibe todos os *workitems* pertencentes ao usuário. Tais *workitems* são mostrados de forma ordenada, seguindo-se um dos quatro critérios de ordenação a seguir: por ordem de chegada, por prioridade, por prazo

de finalização e por tamanho dos dados pertencentes ao *workitem*. A *worklist* também possibilita que o usuário mova um *workitem* para qualquer posição na lista dos *workitems*.

2. **Selecionar *workitems*** - Sempre que um usuário deseja executar um *workitem*, é necessário que ele selecione este *workitem* para execução. Ao efetuar tal ação, caso a tarefa correspondente ao *workitem* seja manual ou semi-automática comum, o PM correspondente é notificado e remove o *workitem* da *worklist* dos demais usuários daquele papel, avisando seus respectivos WMs. O WM do usuário recebe então uma confirmação e, a partir daí, o usuário pode executar o *workitem*. Esse processo de seleção evita que dois usuários executem um mesmo *workitem*. Caso a tarefa correspondente ao *workitem* seja semi-automática cooperativa, o PM é notificado e o WM do usuário recebe uma confirmação, indicando que o usuário pode executar o *workitem*.

3. **Executar *workitems*** - Após a ação de selecionar um *workitem*, o usuário o executa. Com isso, a aplicação referente ao *workitem* é invocada automaticamente pelo SGWF (caso o *workitem* possua uma aplicação associada a ele).

A interface do WorkToDo também permite ao usuário:

- Verificar quais são os processos em execução;
- Consultar o estado dos processos dos quais participa;
- Criar instâncias de um determinado tipo de processo, desde que o usuário pertença ao papel criador daquele tipo.

4.5.2 Notificação de Tarefas

Quando o Despachante detecta que existem tarefas semi-automáticas ou manuais para serem executadas, ele notifica um grupo de usuários através de seus respectivos Gerenciadores de Lista de Trabalho, de acordo com a política de notificação adotada. Os WMs recebem a notificação do Despachante e criam *workitems* nas *worklists* dos usuários.

Para que os usuários sejam notificados, é necessário que estes estejam ativos, pois o Despachante não é capaz de se comunicar com os WMs de usuários inativos.

No caso das tarefas cooperativas, os usuários que pertencem ao grupo definido no modelo de tarefa em questão devem estar ativos. Caso algum usuário não esteja conectado ao SGWF, o Despachante envia uma mensagem alertando o usuário responsável pelo processo de que existem usuários pertencentes ao grupo que estão inativos. Apesar disso,

o Despachante notifica os WMs correspondentes aos usuários ativos e um *workitem* referente à tarefa cooperativa é adicionado na *worklist* de cada um destes usuários. Como periodicamente o Despachante verifica se existe algum novo usuário conectado ao SGWF que deveria ser notificado sobre a disponibilidade da tarefa, novas notificações podem ser enviadas. Já no caso das tarefas semi-automáticas comuns, os usuários que se conectam ao SGWF podem receber notificações de tarefas ainda não selecionadas.

4.5.3 Prazos para Tarefas

Cada tarefa possui um prazo para ser executada pelo usuário. Esse prazo é especificado na definição do modelo de tarefa e representa o tempo máximo para que uma tarefa seja executada (*deadline*), conforme descrito na Seção 4.2.2. O valor desse prazo é decrementado com o passar do tempo e, quando este valor chega a zero, o prazo máximo foi atingido, ou seja, o prazo para execução da tarefa está esgotado. Quando isso acontece, o responsável pelo processo decide se a execução do processo continua ou se deve ser interrompida.

O prazo para execução das tarefas é definido em horas, através de um conjunto de valores $V = \{1, 6, 12, 24, 48\}$, de forma que, quando o prazo expresso é igual a um dos valores do conjunto V, uma mensagem de prazo de tarefa é criada e enviada para a lista de mensagens do usuário, conforme descrito na próxima Seção. Dentre os valores de V, é selecionado também um valor denominado ***Valor de Transferência***. Quando esse valor é igual ao valor do prazo, a execução da tarefa é atribuída automaticamente ao responsável pelo processo, ainda que algum usuário tenha selecionado a tarefa em questão. Tal medida serve para que nenhuma tarefa deixe de ser executada no prazo estipulado porque não foi selecionada por usuário algum ou porque o usuário que a selecionou não a executa. O valor de transferência é único para todos os processos e seu valor (um valor típico é 24) pode ser alterado arbitrariamente pelo administrador.

É importante ressaltar que, quando o valor de transferência é atingido e o *workitem* correspondente à tarefa está selecionado por algum usuário, o *workitem* é removido da *worklist* desse usuário. Quando isso acontece, o usuário não tem mais controle sobre a execução da tarefa, pois o usuário responsável pelo processo assume completamente a responsabilidade pela execução dessa tarefa.

É possível que o usuário responsável pelo processo não esteja conectado ao SGWF no momento em que o valor de transferência de um *workitem* for atingido. Nesse caso, o usuário responsável é notificado posteriormente a respeito desse *workitem*, assim que se reconectar ao SGWF.

O controle dos prazos das tarefas é realizado pelos PMs aos quais essas tarefas pertencem. Portanto, para que o usuário receba as mensagens de prazo de tarefa enviadas pelo PM, é necessário que o mesmo esteja conectado ao SGWF. Os PMs também efetuam a

transferência de tarefas para o usuário responsável pelo processo.

4.5.4 Mensagens

O Gerenciador de Usuários e Papéis (URM) mantém uma lista de mensagens para cada usuário do SGWF (conforme descrito na Seção 4.3.1), contendo notificações relevantes para cada ocorrência de eventos. Quatro tipos de mensagens são identificados no WorkToDo. São eles:

1. **Início de processo** - Esse tipo de mensagem serve para notificar o usuário responsável pelo processo de que este foi iniciado, tendo a execução de suas tarefas sido iniciada também.

2. **Término de processo** - Esse tipo de mensagem serve para notificar o usuário responsável pelo processo de que este teve sua execução terminada.

3. **Falha de tarefa** - Esse tipo de mensagem serve para notificar o usuário responsável pelo processo de que a execução de uma determinada tarefa falhou. O responsável pode então identificar qual o problema que causou a falha e tomar as medidas que julgar convenientes (podendo, por exemplo, re-executar a tarefa ou cancelar todo o processo).

4. **Prazo de tarefa** - Esse tipo de mensagem serve para notificar o usuário responsável pelo processo, bem como o usuário que está com a tarefa selecionada (caso houver algum) de que o prazo para execução da tarefa está se esgotando.

Nota-se que os três primeiros tipos de mensagem são direcionados para o usuário responsável pelo processo. Já o último tipo é direcionado tanto para o usuário responsável pelo processo, como para o usuário que selecionou a tarefa em questão.

Ainda que um usuário esteja inativo, suas mensagens são armazenadas em sua lista de mensagens. Nesse caso, quando o usuário se reconecta ao SGWF, recebe todas as mensagens enviadas para ele. Caso o usuário esteja ativo, ele recebe todas as mensagens no instante em que são enviadas.

A Figura 4.7 mostra a estrutura de uma mensagem. O campo `Type` identifica o tipo da mensagem enviada, conforme descrito acima. O campo `Sender` indica o remetente da mensagem e contém o nome de uma instância em execução. O campo `Time` indica a data e a hora em que a mensagem foi enviada. Finalmente, o campo `Text` contém o texto da mensagem.

Figura 4.7: Estrutura das mensagens enviadas aos usuários do SGWF

4.5.5 Diagramas de Seqüência

Alguns diagramas de seqüência das principais operações do WorkToDo são apresentados nesta Seção em notação UML [9], para uma melhor compreensão do funcionamento do sistema e para que se possam esclarecer alguns detalhes importantes.

Quaisquer operações no sistema são iniciadas através da requisição do usuário à interface do sistema, que por sua vez transfere esta requisição ao componente apropriado.

Criação de uma Instância de Processo

A Figura 4.8 mostra o procedimento para a criação de uma instância de processo. Os passos para a execução desse procedimento são detalhados a seguir.

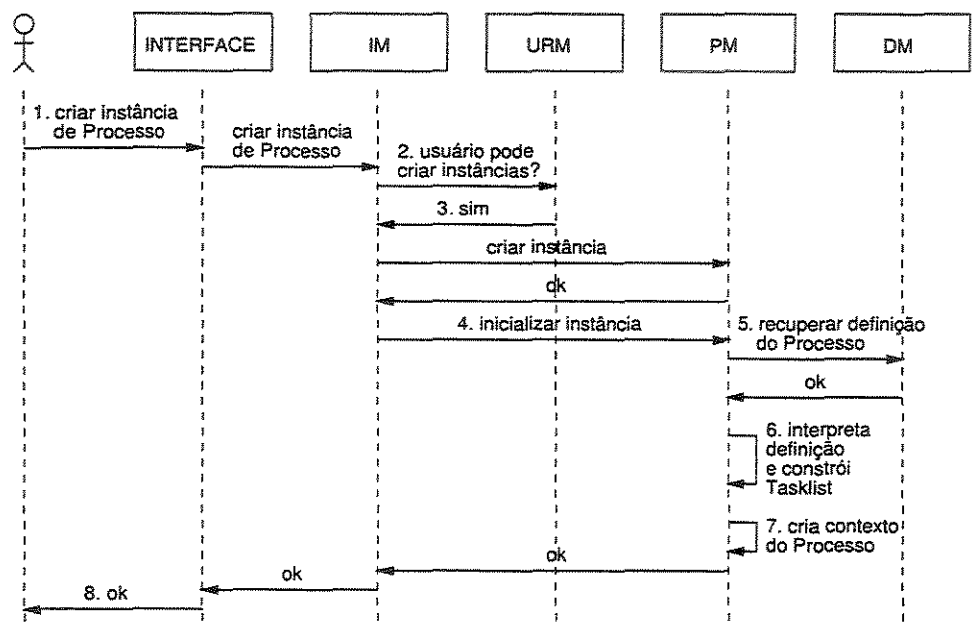

Figura 4.8: Diagrama de seqüência para a criação de uma instância de processo

1. O usuário utiliza a interface do sistema para requisitar ao IM que uma nova instância de processo seja criada. Para isso, ele informa qual o tipo de processo que deseja criar, bem como o nome da máquina a partir da qual deseja que a instância seja gerenciada;

2. O IM, por sua vez, verifica junto ao URM se o usuário que fez a requisição pertence ao papel criador daquele tipo de processo;
3. Caso o IM receba resposta afirmativa para a verificação feita junto ao URM, ele cria um novo PM na máquina informada pelo usuário;
4. O IM inicializa a instância de processo, informando ao PM recém-criado qual o tipo de processo que irá gerenciar;
5. O PM faz uma requisição ao DM para obter a definição do tipo de processo em questão;
6. O PM ativa o PDLI, que interpreta a definição do tipo de processo da seguinte forma: para cada tarefa contida na definição, o PDLI interpreta a definição do modelo de tarefa correspondente e constrói uma *task definition* para a tarefa. Após interpretar todas as definições dos modelos de tarefas referentes ao processo, todas as *task definitions* são inseridas em uma lista, formando a *task list* do processo;
7. O PM cria o contexto do processo na máquina onde está sendo executado. Nessa máquina, ele cria um diretório com o nome da instância de processo, onde são armazenadas as informações referentes à instância. Nesse diretório, são copiados os dados a serem utilizados pelas tarefas conforme a necessidade;
8. A interface informa ao usuário que a instância foi criada com sucesso e que sua execução já foi iniciada.

Em caso de falha em qualquer um dos passos entre 2 e 7, a criação da instância é cancelada automaticamente pelo sistema e a interface informa ao usuário sobre a falha na operação. As ações efetuadas (caso tenham ocorrido) durante a execução desse procedimento, tais como a construção da *task list* e do contexto, são desfeitas pelo sistema.

Execução de uma Tarefa Automática

O procedimento executado pelo sistema para a execução de uma tarefa automática, conforme mostrado na Figura 4.9, é detalhado a seguir:

1. O PM prepara a tarefa para ser executada, através de seu subcomponente chamado Despachante, conforme visto na Seção 4.3.4;
2. Quando a tarefa está pronta para ser executada, o PM cria um TM para essa tarefa e inicia a execução da mesma;
3. O TM passa a gerenciar a tarefa e aguarda um retorno da execução da mesma;
4. A tarefa informa ao seu respectivo TM qual foi o resultado de sua execução. O TM, por sua vez, repassa esse resultado para o PM cujo processo contém a tarefa em questão. Esse resultado fica armazenado no processo e pode ser consultado pelos usuários do sistema;

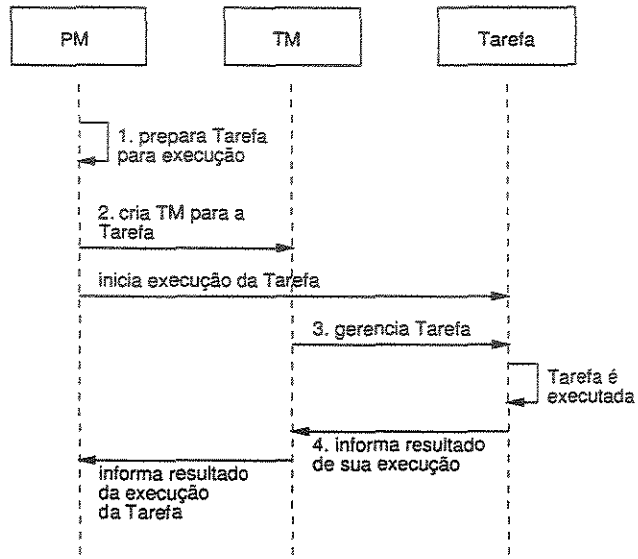

Figura 4.9: Diagrama de seqüência para a execução de uma tarefa automática

Seleção de um Workitem de Tarefa Semi-automática Comum

O procedimento para seleção de um *workitem*, mostrado na Figura 4.10, é detalhado a seguir:

1. Através da interface do sistema, o usuário requisita ao WM a seleção de um determinado *workitem*;
2. O WM verifica qual a tarefa correspondente ao *workitem* e envia uma requisição ao PM ao qual a tarefa pertence solicitando a seleção da tarefa para execução;
3. O PM verifica se a tarefa ainda está disponível (algum outro usuário pode tê-la selecionado);
4. O PM remove os *workitems* correspondentes à tarefa das *worklists* dos demais usuários que haviam sido notificados da disponibilidade da tarefa;
5. O PM altera o estado da tarefa para SELECTED, indicando que ela foi selecionada por um usuário. Além disso, armazena na *task definition* o nome do usuário que efetuou a seleção;
6. O WM altera o estado do *workitem* para SELECTED;
7. A interface notifica o usuário de que o *workitem* foi selecionado com sucesso e está pronto para ser executado.

Se um dos itens entre 2 e 3 falha, a seleção do *workitem* é automaticamente cancelada e a interface informa o usuário a respeito da falha na operação.

É importante observar que um usuário somente é capaz de selecionar *workitems* quando está ativo, caso contrário o PM correspondente não poderá ser notificado da seleção.

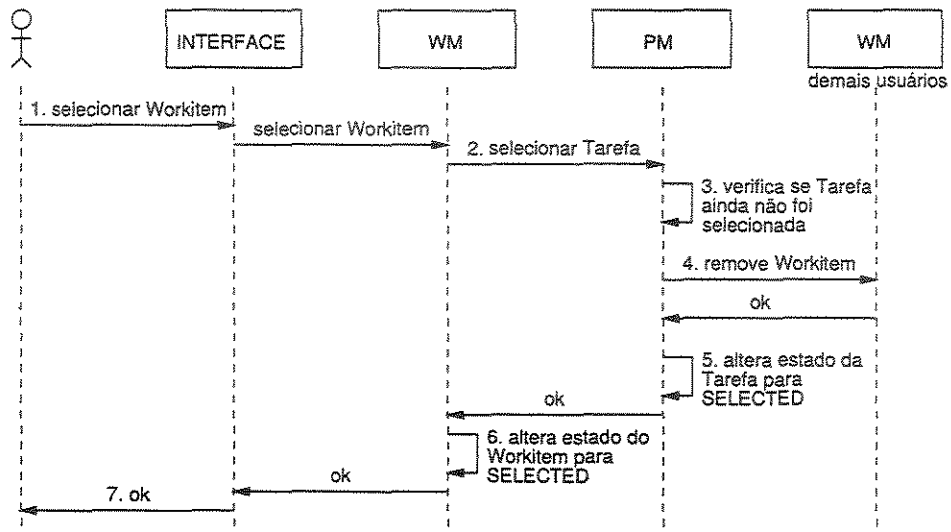

Figura 4.10: Diagrama de seqüência para a seleção de um *workitem* de tarefa semi-automática comum

4.6 Gerenciamento de Transações

No WorkToDo, para que as tarefas semi-automáticas cooperativas sejam executadas, é necessária a utilização de um mecanismo de gerenciamento de transações [38]. Esse mecanismo cria uma estrutura de transações chamada **Transação Cooperativa** (descrita na Seção 4.6.1), composta tanto por **Transações de Grupo** (descritas na Seção 4.6.2) como por **Transações de Usuário** (descritas na Seção 4.6.3).

A estrutura da transação cooperativa é composta por transações na forma de uma árvore que pode conter vários níveis, constituindo uma ***hierarquia de transações***. Os nós internos da árvore representam as transações de grupo e os nós folhas representam as transações de usuário, conforme mostrado na Figura 4.11. Os nós descendentes de uma transação de grupo são chamados ***subtransações*** dessa transação de grupo.

Uma transação de grupo possui um conjunto de transações (de usuário e/ou de grupo) associado a ela. Portanto, somente os usuários pertencentes ao grupo em questão podem criar transações vinculadas a esta transação de grupo. É importante ressaltar que cada transação está vinculada a uma e somente uma transação de grupo.

O usuário que cria uma transação de grupo é chamado de Usuário Coordenador do grupo, conforme visto na Seção 4.3.8. Esse usuário é responsável pelo gerenciamento dos outros usuários que participam do seu grupo (inserção/remoção de usuários no/do grupo).

Cada transação possui sua própria área de trabalho para manipulação de objetos.

A transferência dos objetos entre os níveis da hierarquia se dá através de um mecanismo de *check-in/check-out*, que fornece um controle de concorrência baseado em trancas. Esse

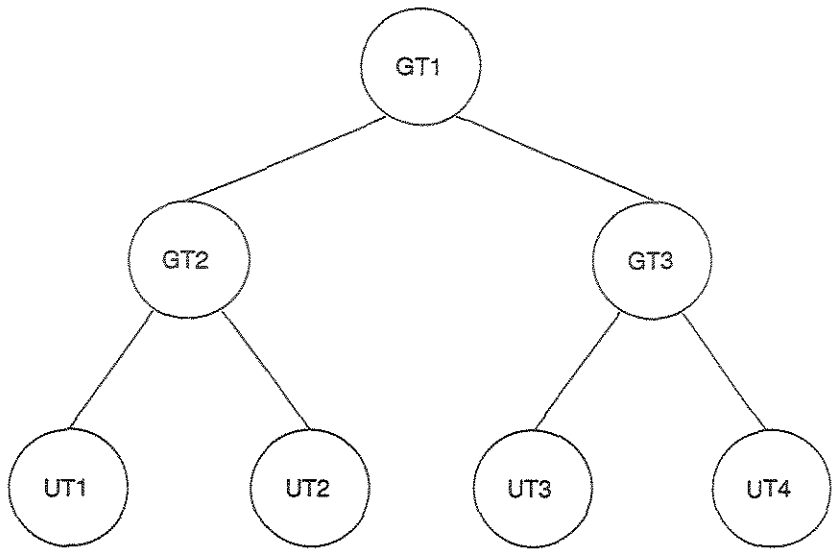

Figura 4.11: Hierarquia de transações de uma tarefa cooperativa

mecanismo é detalhado, como se segue:

Check-out - Este mecanismo é responsável:

- pela transferência de objetos da área de trabalho de uma transação de grupo para uma área de trabalho descendente (transação de grupo ou transação de usuário) e;
- pelo controle de concorrência através de trancas[2] aos objetos requisitados pelas subtransações.

Check-in - Este mecanismo é responsável:

- pela atualização dos objetos na área da transação de grupo;
- pela remoção dos objetos da área da transação requisitante e;
- pela liberação de trancas.

4.6.1 Transação Cooperativa

Conforme visto na Seção 4.3.8, uma transação cooperativa é composta por uma estrutura hierárquica de transações, onde os nós internos são transações de grupo e os nós folhas são transações de usuário.

[2]Do termo em inglês: *locks*

Quando uma transação cooperativa é iniciada, uma nova área de trabalho é criada para ela. Essa área contém os objetos utilizados pela transação cooperativa e suas subtransações. Além disso, essa área mantém um arquivo de *log* para o mecanismo de recuperação, conforme será visto na Seção 4.7.2.

Para que uma transação T possa manipular um objeto presente na transação de grupo, é necessário que T requisite esse objeto para leitura ou para escrita. Esse objeto é então bloqueado[3] e copiado da área de trabalho do grupo para a área de trabalho de T. Ao terminar de utilizar esse objeto, a transação T deve desbloquear o objeto em questão. Caso esse objeto tenha sido bloqueado para leitura, a cópia do objeto é removida da área de trabalho de T e o estado desse objeto é mantido na área de trabalho do grupo. Caso o objeto tenha sido bloqueado para escrita, ele é atualizado na área de trabalho do grupo.

Uma transação, dentro de uma transação cooperativa, pode ser **vital** (o seu cancelamento implica no cancelamento de sua transação pai) ou **não vital** (o seu cancelamento não implica no cancelamento de sua transação pai).

Quando a transação cooperativa é finalizada, o usuário coordenador da transação deve selecionar o tipo de terminação da transação, que pode ser:

ABORTED - tipo de término que equivale ao tipo `FAILED` para término de tarefas ou;

COMMITTED - tipo de término que equivale ao tipo `SUCCEEDED` para término de tarefas.

Esses tipos de terminação são válidos tanto para transações de grupo, como para transações de usuário, mas diferem quanto ao seu significado, dependendo do tipo de transação. Caso a transação seja de usuário, a terminação pode ser do tipo `ABORTED`, cancelando a transação de usuário e suas atualizações ou pode ser do tipo `COMMITTED`, validando os objetos dessa transação na área da transação de grupo ao qual pertence. Caso a transação seja de grupo, a terminação está condicionada ao tipo de término de suas subtransações. Para que a transação de grupo termine com sucesso, é necessário que todas as suas subtransações vitais tenham terminado com sucesso, ou seja, se alguma subtransação vital não tiver terminado com sucesso, a transação de grupo será cancelada. Apesar disso, a transação de grupo pode terminar com falha, independentemente do término com sucesso de todas as suas subtransações. Isso se dá devido ao fato de o usuário coordenador do grupo ter liberdade de determinar o tipo de término da transação de grupo.

É importante lembrar que o estado da transação cooperativa, bem como o estado de seus objetos são armazenados de forma persistente no sistema.

[3]Do termo em inglês *locked*

4.6.2 Transação de Grupo

Dentro de uma transação de grupo pode-se criar outras transações de grupo ou transações de usuário, que são chamadas subtransações dessa transação de grupo. O coordenador de uma transação de grupo é o criador dessa transação.

A transação de grupo pode criar transações de usuário para usuários que pertençam ao grupo, ou seja, que estão definidos no modelo de tarefa, descrito na Seção 4.2.2.

É importante ressaltar que pode-se criar subtransações somente dentro de transações de grupo.

4.6.3 Transação de Usuário

Quando uma transação de usuário é criada, ela pode requisitar um objeto para outra transação de usuário dentro do grupo à qual pertence. Tal requisição pode ser de três tipos:

Cópia (Copy) - Esse tipo de requisição provoca a cópia do objeto na área da transação requisitante e permite que a transação requisitante invoque qualquer operação sobre o objeto requisitado, porém não permite que essa transação devolva o objeto para sua transação de origem, nem atualize esse objeto na área de trabalho do grupo.

Empréstimo (Loan) - Esse tipo de requisição provoca a cópia do objeto na área da transação requisitante e permite que a transação requisitante execute todas as operações do objeto e devolva uma cópia atualizada do mesmo para a transação original. Porém, não permite que a transação atualize o objeto na área da transação de grupo.

Concessão (Concession) - Nesse tipo de requisição, a transação original fornece para a transação requisitante uma cópia do objeto e concede a ela todos os privilégios sobre essa cópia. A transação requisitante também passa a executar o mecanismo de *check-in* no lugar da transação original.

É importante ressaltar que as requisições mencionadas acima são válidas somente entre as transações de usuário pertencentes a um mesmo grupo. Além disso, as transferências dos objetos entre as áreas das transações envolvidas são coordenadas pela transação de grupo.

Quando uma transação deixa de utilizar um objeto, ela pode liberá-lo para que outras transações possam utilizá-lo. Quando a transação libera o objeto, caso este tenha sido copiado, ele é simplesmente descartado. Caso tenha sido emprestado, ele é devolvido para a sua transação de origem com as possíveis modificações que sofreu. Caso tenha sido concedido, a transação que o possui executa o mecanismo de *check-in*.

No instante em que uma transação de usuário é terminada, os objetos emprestados são devolvidos para sua transação original, as cópias são descartadas e os outros objetos são transferidos para a área de trabalho da transação pai.

4.6.4 Diagramas de Seqüência

Alguns diagramas de seqüência das operações sobre tarefas cooperativas são apresentados nesta Seção, em notação UML [9].

Seleção de um Workitem de Tarefa Cooperativa

A Figura 4.12 mostra o procedimento para a seleção de um *workitem*. Os passos para a execução desse procedimento são detalhados a seguir.

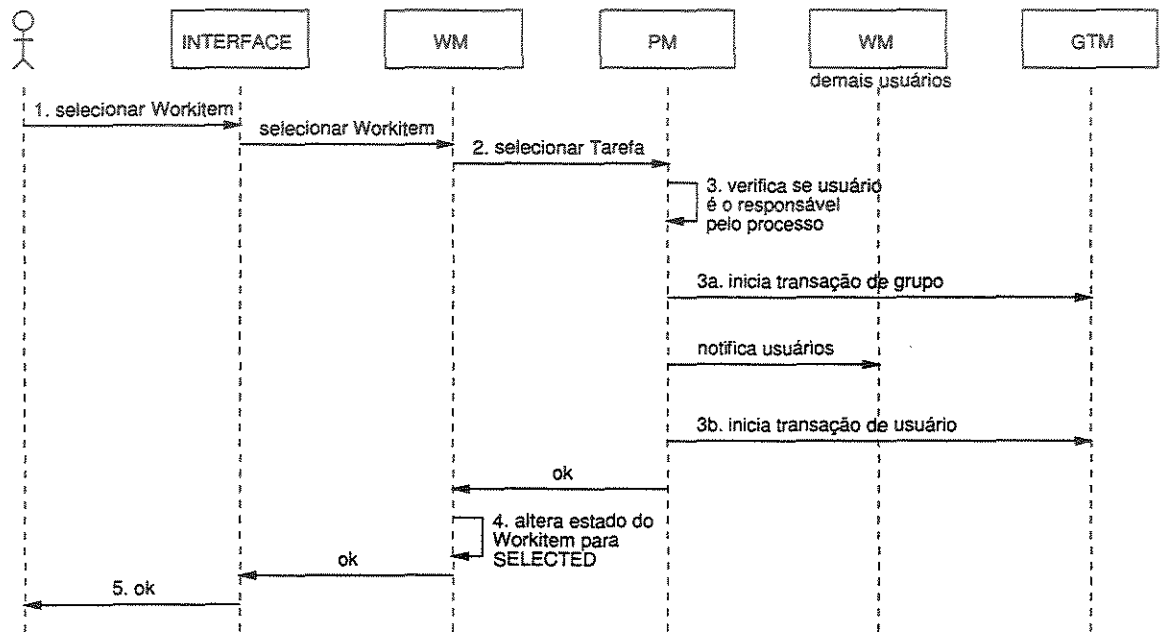

Figura 4.12: Diagrama de seqüência para a seleção de um *workitem* de tarefa cooperativa

1. O usuário utiliza a interface do sistema para requisitar ao WM a seleção de um *workitem*;
2. O WM faz a associação entre o *workitem* e a tarefa a que se refere e envia uma requisição ao PM que contém essa tarefa, solicitando a seleção da mesma para execução;
3. O PM verifica se o usuário em questão é o responsável pelo processo;

a. Caso o usuário seja o responsável pelo processo, a aplicação referente à **transação de grupo** é iniciada pelo PM, através de uma requisição ao GTM e os demais usuários do grupo são notificados sobre a disponibilidade da tarefa cooperativa para execução;
b. Caso o usuário não seja o responsável pelo processo (ou seja, um outro usuário pertencente ao grupo), a aplicação referente à **transação de usuário** é iniciada pelo PM, através de uma requisição ao GTM;
4. O WM altera o estado do *workitem* para SELECTED;
5. A interface notifica o usuário de que o *workitem* foi selecionado com sucesso e está pronto para ser executado.

Se houver falha no passo 2, a seleção do *workitem* é automaticamente cancelada e a interface informa ao usuário que houve falha na operação. No passo 3, o sistema verifica se o usuário é o responsável pelo processo. Mesmo que algum dos demais usuários do grupo não puder ser contactado (por estar desconectado do SGWF, por exemplo), a seleção do *workitem* é efetuada normalmente pelo usuário responsável e, posteriormente, pelos usuários do grupo que estão ativos no SGWF. Os demais usuários inativos do grupo são contactados assim que se conectam ao SGWF, conforme descrito na Seção 4.5.2.

O usuário é capaz de selecionar *workitems* somente quando está ativo. Caso contrário, o PM correspondente não poderá ser notificado da seleção.

Finalização de uma Tarefa Cooperativa

A Figura 4.13 mostra o procedimento para a finalização de uma tarefa cooperativa. Os passos para a execução desse procedimento são detalhados a seguir.

1. O usuário utiliza a interface do sistema para informar ao WM o resultado da tarefa cooperativa;
2. O WM faz a associação entre o *workitem* e a tarefa a que se refere e envia uma requisição ao PM que contém essa tarefa, solicitando a verificação do tipo de usuário;
3. O PM verifica se o usuário em questão é o responsável pelo processo;
4. Caso o usuário seja o responsável pelo processo, o resultado da tarefa cooperativa é o resultado esperado, pois só o usuário responsável pelo processo pode informar o resultado final da tarefa cooperativa, já que ele é o coordenador do grupo. Esse resultado é repassado tanto para a *worklist* do usuário responsável pelo processo, como para as *worklists* dos demais usuários do grupo;
5. O resultado da tarefa cooperativa já está disponível na *worklist* do usuário e pode ser consultado por ele.

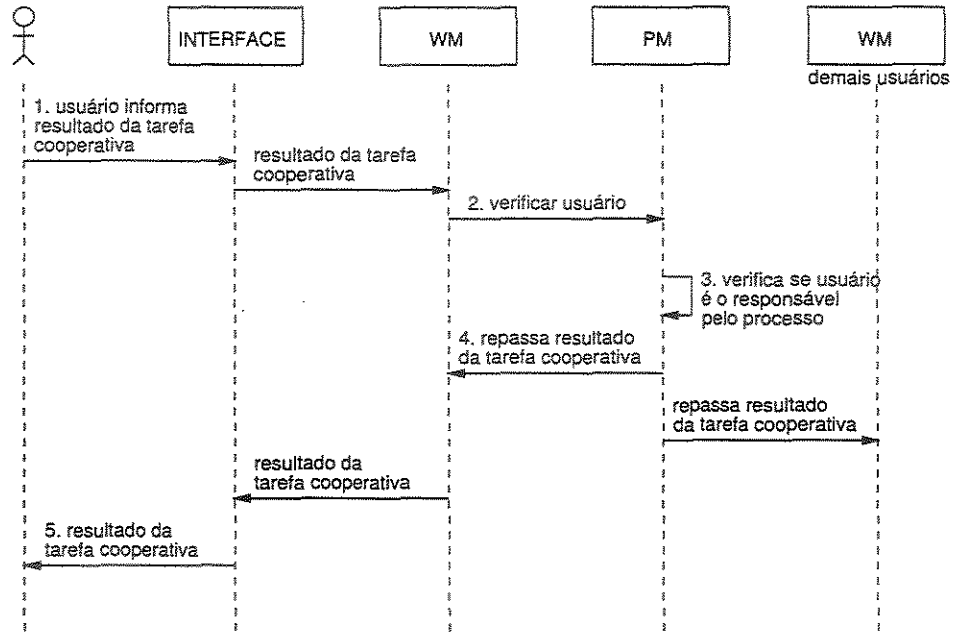

Figura 4.13: Diagrama de seqüência para a finalização de uma tarefa cooperativa

4.7 Tratamento de Falhas

O WorkToDo fornece tratamento para dois tipos de falha: **falha de tarefa** e **falha de componente**. Estes tipos de falha são descritos a seguir. No final da Seção, são apresentados alguns diagramas de seqüência para tratamento de falhas nos componentes da arquitetura do WorkToDo.

4.7.1 Falha de Tarefa

As tarefas podem terminar no estado com sucesso (`SUCCEEDED`) ou ainda com falha (`FAILED`). A recuperação de processos com alguma tarefa que falhou pode ser feita através de **recuperação com avanço**[4] ou **recuperação com atraso**[5].

A utilização de regras de dependência que definem o fluxo de controle de um processo permite a recuperação desses processos. No caso de recuperação com atraso, tarefas de compensação são executadas para desfazer tarefas já executadas antes da falha. No caso de recuperação com avanço, um caminho alternativo é executado. Por exemplo, no caso de recuperação com atraso, para um processo que já teve a tarefa t_1 executada com sucesso e a tarefa t_2 com falha, basta executar a tarefa t_3, que é compensação de t_1, para desfazer o processo. Para isso, deve ser especificada a regra $t_2 \rightarrow$ `FAILED` como condição para início

[4]Do Inglês: *forward recovery*
[5]Do Inglês: *backward recovery*

da tarefa t_3. No caso de recuperação com avanço, para um processo que já teve a tarefa t_1 executada com sucesso, caso t_2 tenha sido executada com sucesso, a tarefa t_3 é executada. Nesse caso, temos o caminho t_1, t_2 e t_3. Caso a tarefa t_2 falhe, a tarefa t_4 é executada, caracterizando-se assim, um caminho alternativo de execução, dado por t_1, t_2 e t_4.

4.7.2 Falha de Componente

O tratamento de falhas para os diversos componentes é o seguinte:

Gerenciador de Processo

O RM trata das falhas que ocorrerem nos PMs, conforme já descrito na Seção 4.3.7.

Gerenciador de Tarefa

O PM trata das falhas que ocorrerem nos TMs. Para isso, o PM verifica periodicamente se os TMs vinculados a ele estão ou não ativos. Caso ocorra uma falha, o PM pode criar um novo TM na mesma, ou em outra máquina, inicializando-o com o último estado consistente do TM que falhou.

Gerenciador de Lista de Trabalho

O estado do WM é persistente e pode ser recuperado no reinício após uma falha. Um usuário pode opcionalmente salvar o estado do WM junto ao URM e recuperar esse estado após uma reconexão, a partir da mesma máquina ou de uma outra máquina.

Gerenciador de Tarefas Cooperativas

Uma tarefa cooperativa é uma transação de grupo estruturada como uma árvore de transações. Cada transação nessa árvore possui um estado persistente que pode ser restaurado em caso de falha. O estado da transação é salvo em memória persistente quando o usuário invoca uma operação de ***checkpoint*** ou quando o gerenciador de transações (GTM) é desativado pelo *daemon* do OrbixWeb (*orbixd*). Na recuperação de uma falha, esse estado é restaurado e as operações subseqüentes à operação de *checkpoint*, armazenadas no ***log*** da transação, são executadas de forma a obter o estado da transação no momento em que ocorreu a falha [38].

4.7.3 Diagramas de Seqüência para Tratamento de Falhas

Os diagramas de seqüência dos tratamentos de falha do WorkToDo são apresentados nesta Seção em notação UML [9], exceto o Diagrama para Tratamento de Falha do Gerenciador

de Tarefas Cooperativas, pois este está melhor definido em [38].

Tratamento de Falha do Gerenciador de Processo

A Figura 4.14 mostra o procedimento para o tratamento de falha do Gerenciador de Processo. Os passos para a execução desse procedimento são detalhados a seguir.

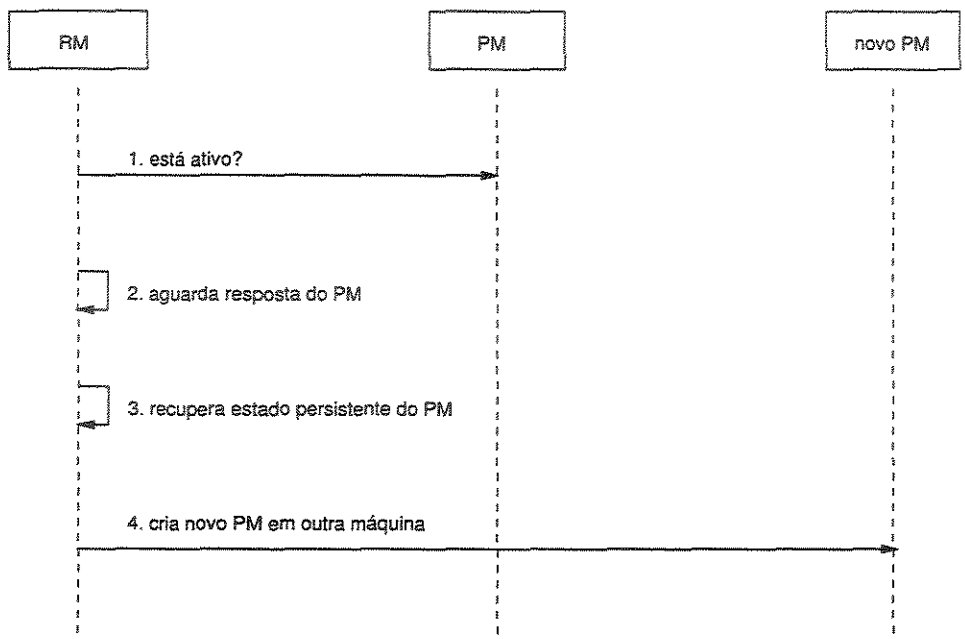

Figura 4.14: Diagrama de seqüência para o tratamento de falha do Gerenciador de Processo

1. O RM verifica periodicamente se o PM está ativo;
2. O RM aguarda uma resposta do PM por um tempo pré-definido;
3. Caso o PM não responda, ou seja, não esteja ativo, o RM recupera o último estado consistente desse PM (mantido em memória persistente) a partir do seu Repositório de Gerenciadores de Processo;
4. Com o estado persistente do PM que falhou, o RM cria um novo PM em outra máquina.

Tratamento de Falha do Gerenciador de Tarefa

A Figura 4.15 mostra o procedimento para o tratamento de falha do Gerenciador de Tarefa. Os passos para a execução desse procedimento são detalhados a seguir.

1. O PM verifica periodicamente se o TM está ativo;

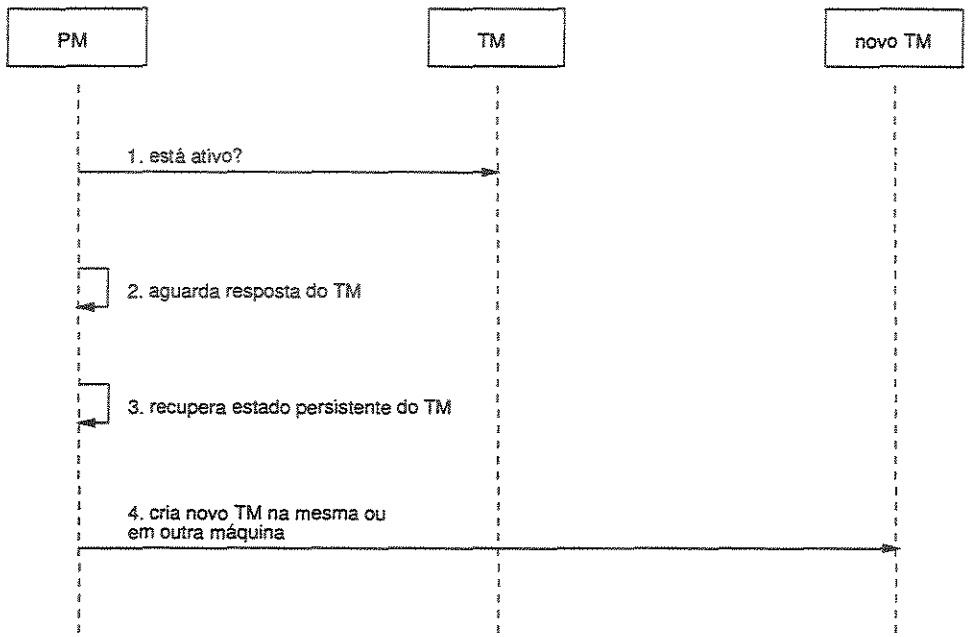

Figura 4.15: Diagrama de seqüência para o tratamento de falha do Gerenciador de Tarefa

2. O PM aguarda uma resposta do TM por um tempo pré-definido;
3. Caso o TM não responda, ou seja, não esteja ativo, o PM recupera o último estado consistente desse TM (mantido em memória persistente) a partir do seu Repositório de Gerenciadores de Tarefa;
4. Com o estado persistente do TM que falhou, o PM cria um novo TM na mesma ou em outra máquina.

Tratamento de Falha do Gerenciador de Lista de Trabalho

A Figura 4.16 mostra o procedimento para o tratamento de falha do Gerenciador de Lista de Trabalho. Os passos para a execução desse procedimento são detalhados a seguir.

1. O usuário pode, opcionalmente, utilizar a interface do sistema para salvar o estado do WM;
2. O WM salva seu estado junto ao URM;
3. O URM salva o estado do WM em seu Repositório de Usuários, que contém as informações relevantes de cada usuário, conforme visto na Seção 4.3.1;
4. A interface notifica o usuário de que o estado do WM foi gravado com sucesso;
5. O usuário pode também utilizar a interface do sistema para recuperar o estado desse WM após se reconectar ao SGWF;

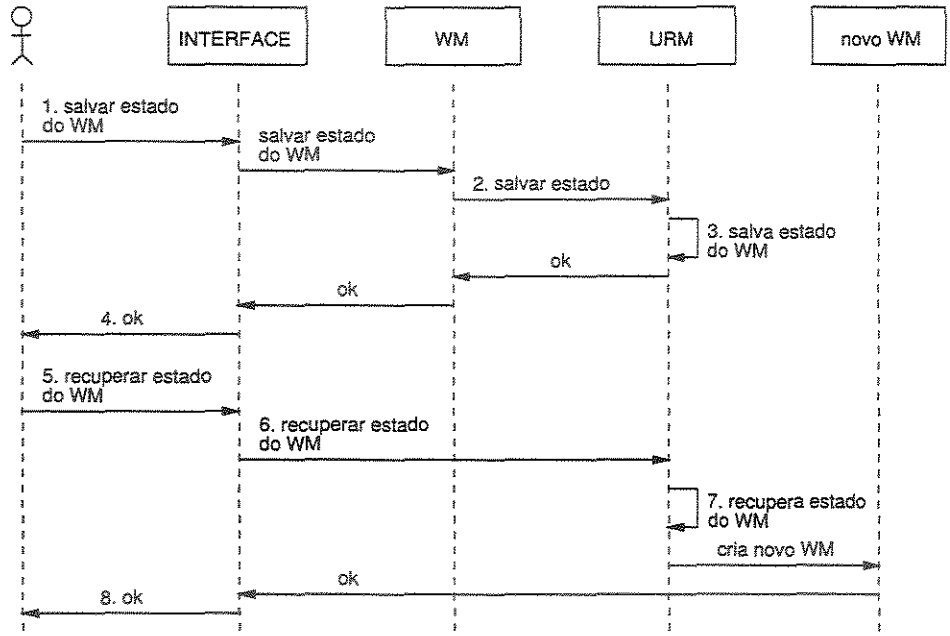

Figura 4.16: Diagrama de seqüência para o tratamento de falha do Gerenciador de Lista de Trabalho

6. A interface requisita a recuperação do estado do WM em questão junto ao URM;
7. O URM recupera o estado do WM e cria um novo WM com esse estado;
8. O usuário já pode utilizar o WM normalmente.

Capítulo 5

Implementação do Protótipo

Neste capítulo, são descritas as principais características do protótipo WorkToDo, que foi implementado em Java - JDK 1.1.8. Como infra-estrutura de suporte à distribuição, utilizou-se o OrbixWeb 3.1c.

Neste capítulo também é apresentado um exemplo de execução de um *workflow* e os resultados obtidos a partir dos testes realizados no protótipo.

5.1 Java e CORBA

O WorkToDo foi desenvolvido de tal forma que possa ser implementado utilizando qualquer linguagem de programação e qualquer infra-estrutura de comunicação entre objetos. Na implementação do protótipo descrito, escolhemos Java como linguagem de programação e CORBA como infra-estrutura de comunicação remota. A seguir, são apresentadas algumas razões pelas quais se optou por essas tecnologias.

Java foi escolhida como linguagem de programação por ser uma linguagem multiplataforma. Tal decisão de projeto permite que o SGWF possa ser executado em diferentes sistemas operacionais e diferentes plataformas de *hardware*. Esta decisão foi embasada no fato de que, num sistema de *workflow*, múltiplos usuários participam na execução das tarefas de um processo e, para isso, utilizam diferentes computadores em diferentes ambientes. A versão de Java utilizada na implementação do protótipo foi a JDK 1.1.8 [40]. Versões mais recentes de Java, tais como a versão 1.2 ou 1.3 poderiam ter sido utilizadas mas, pelo fato de que o mapeamento de IDL para Java oferecido pelo OrbixWeb gerava código na versão 1.1.8, não foi possível utilizar estas versões. Além disso, todas as classes do OrbixWeb também estão implementadas nesta versão de Java (1.1.8). Mesmo assim, tentou-se executar o protótipo do sistema compilado em uma versão mais recente de Java (versão 1.2), mas o OrbixWeb gerou erros de execução.

CORBA foi escolhido como infra-estrutura de comunicação remota por conter um

amplo conjunto de funcionalidades que fornecem transparência de acesso e localização e simplificam a implementação de aplicações distribuídas, conforme descrito na Seção 2.3. A implementação de CORBA utilizada foi o OrbixWeb 3.1c da Iona [37], que está descrito na Seção 2.3.1. O OrbixWeb foi escolhido por fornecer o mapeamento de IDL para Java e por estar disponível no Instituto de Computação da Universidade Estadual de Campinas (IC/Unicamp). Quando o projeto do WorkToDo foi iniciado, as implementações gratuitas tais como a da SUN, que acompanham o próprio Java, se limitavam a implementar apenas as funcionalidades básicas do ORB. Tais implementações não consideravam funcionalidades como persistência de objetos e *multithreading* de servidores. A implementação mais recente da SUN já implementa essas funcionalidades, facilitando o desenvolvimento de novas aplicações que a utilizem.

5.2 Simplificações

O protótipo do WorkToDo não teve todas as suas funcionalidades e características implementadas devido ao limite de tempo para conclusão do projeto. Com isso, sua implementação focou os aspectos referentes à operação dos usuários, com o objetivo de fornecer facilidades de cooperação para os usuários de um grupo. Apesar disso, muitos outros aspectos relevantes com relação a um SGWF não foram implementados, como: ferramentas para definição de processos, políticas de escalonamento e sincronização de tarefas, políticas de notificação de usuários, interoperabilidade com outros SGWFs e ferramentas de supervisão e administração. A implementação do protótipo reduziu-se apenas aos aspectos considerados mais importantes para a interação e cooperação entre os usuários, proporcionando um funcionamento satisfatório do sistema.

A seguir, são descritas as simplificações realizadas no desenvolvimento do protótipo, bem como as características implementadas.

Gerenciador de Usuários e Papéis, de Definições e de Instâncias - Esses gerenciadores foram implementados como servidores *multithreaded*, podendo receber várias requisições ao mesmo tempo e permitindo que seus repositórios sejam acessados simultaneamente. Tais vantagens podem diminuir consideravelmente o tempo de resposta das requisições feitas pelas aplicações clientes.

Devido ao fato de poder existir um grande número de processos em execução num dado instante, esses gerenciadores poderiam prejudicar a performance do sistema. Uma solução de implementação encontrada para esse problema é a replicação desses servidores, eliminando assim a sobrecarga sobre um nó da rede. Além disso, o sistema se torna mais tolerante a falhas, pois no caso de falha de um componente,

os outros podem dispor das mesmas operações. No caso do protótipo, optou-se por não replicar esses componentes, devido às limitações de tempo.

Gerenciador de Processo - O PM foi um dos componentes com mais funcionalidades implementadas, pois tais funcionalidades são essenciais para a execução das instâncias de processos e, portanto, para o funcionamento correto do sistema.

O Escalonador foi completamente implementado. Assim que ocorre a mudança de estado de uma tarefa, o Escalonador reavalia as árvores de dependência de todas as tarefas que se encontram na lista de dependentes da tarefa. Caso alguma dessas árvores seja avaliada para `true`, o Escalonador insere sua tarefa correspondente em uma fila de tarefas que estão prontas para execução. O Despachante consulta esta fila de tempos em tempos para despachar as tarefas para execução.

Na implementação do Despachante, só foi considerada a política de notificação de usuários "todos os usuários do papel". O Despachante também não considera a prioridade das tarefas, despachando-as na ordem em que ficam prontas para executar.

A recuperação de falhas dos Gerenciadores de Tarefa não foi implementada, pois considerou-se que estes componentes sempre informam o resultado da execução das tarefas que monitoram. Já a recuperação de falhas dos Gerenciadores de Processo foi implementada através do componente Gerenciador de Recuperação. A recuperação de falhas dos Gerenciadores de Lista de Trabalho é tratada em outro trabalho [39].

O PM precisa se comunicar com os usuários do sistema constantemente, a fim de enviar notificações de tarefas prontas para execução, remover tarefas que já tenham sido selecionadas por outro usuário e assim por diante. Para isso, cada PM mantém uma lista com os usuários ativos de cada papel (como uma *cache*), para evitar consultas ao URM sempre que precisar notificar algum usuário (pois é o URM que contém a referência ao WM do usuário). A lista que o PM mantém é inicializada no momento da criação do mesmo e é atualizada periodicamente junto ao URM.

Gerenciador de Tarefa - Ao iniciar a execução de uma tarefa, o TM invoca sua aplicação correspondente e aguarda o término da execução desta tarefa, para então repassar o resultado dessa execução ao PM a que está vinculado. Quando a tarefa falha, caso ela tenha um número de novas tentativas definido (*retries*) maior do que zero, ela pode ser re-executada esse número de vezes. Para isso, esse mecanismo de verificação de *retries* foi implementado como um contador que é decrementado a cada nova tentativa de execução até que atinja o valor zero.

Gerenciador de Tarefas Cooperativas - Quando uma tarefa cooperativa é iniciada, o GTM invoca a interface para gerenciamento de tarefas cooperativas e gerencia a troca controlada de informações entre os usuários, a fim de se obter cooperação.

Dentre os recursos do serviço de transações cooperativas [38], foram implementados até o momento:

- O gerenciamento de transações cooperativas (descrito na Seção 4.6.1), incluindo o tratamento de uma tarefa cooperativa como uma transação hierárquica em forma de árvore, onde os nós internos são transações de grupo e os nós folhas são transações de usuário;
- O gerenciamento de transações de grupo (descrito na Seção 4.6.2), incluindo a área de trabalho do grupo, gerenciada pelo usuário coordenador do grupo e as operações que este pode realizar;
- O gerenciamento de transações de usuário (descrito na Seção 4.6.3), incluindo a área de trabalho de um usuário pertencente ao grupo e as operações que ele pode realizar.

As interfaces do gerenciamento de transações de grupo e de usuário foram implementadas utilizando-se componentes do pacote AWT de Java [40]. Essas interfaces são mostradas na Seção 5.4.

A implementação da recuperação de falhas baseada em *logs* e *checkpoints* não foi finalizada até o momento.

Interpretador da Linguagem de Definição de Processo - A partir de uma definição de processo, o PDLI interpreta cada definição de tarefa instanciada nesse processo e também a definição da aplicação correspondente à tarefa, se houver. O PDLI constrói uma *task definition* para cada definição de tarefa interpretada e, no final, tem-se uma *task list* contendo todas as *task definitions* da instância.

O PDLI assume que existe pelo menos uma tarefa que não possui dependências, a fim de que o processo possa ser iniciado. Além disso, assume também que não existem dependências mútuas, ou seja, que não há ciclos de dependências. Por exemplo, se uma tarefa t_1 depende da tarefa t_2 e, ao mesmo tempo, t_2 depende de t_1, há um ciclo que faz com que o processo seja dado por concluído antes mesmo que alguma tarefa seja executada. De forma análoga, se t_1 depende de t_2, t_2 de t_3 e t_3 de t_1, há um ciclo de dependências que impede a execução das tarefas e o processo é dado por concluído. O PM assume que o processo está concluído quando não existem tarefas prontas para executar.

Prazo das Tarefas - Apenas a opção do prazo das tarefas em horas foi implementada no protótipo, apesar de a PDL permitir os formatos em horas ou dias. Portanto, o valor do prazo é sempre considerado em horas, ainda que na definição da tarefa esteja expresso em dias.

Tipos de Dados - Dentre os tipos de dados descritos na Seção 4.1.5, somente o tipo Arquivo foi considerado. Os demais tipos não são interpretados corretamente quando o PDLI interpreta a definição do processo.

Atividades - Na implementação do protótipo, somente tarefas são consideradas, sem incluir outras atividades dentro de uma atividade (sub-atividades).

5.3 Considerações de Implementação

Os componentes da arquitetura do WorkToDo foram implementados como servidores CORBA e, para os testes do protótipo foi estabelecido um tempo de ativação de 10 minutos para cada servidor. A interface desses servidores está descrita no Apêndice B.

O protótipo foi implementado contendo uma interface gráfica que utiliza menus, construída a partir de componentes do pacote AWT de Java [40]. Através dessa interface, o usuário pode interagir com o SGWF e com seu Gerenciador de Lista de Trabalho. O usuário é capaz de visualizar, selecionar, trancar[1] e executar *workitems*. Ele também pode consultar os estados dos processos em execução, criar novas instâncias e assim por diante. Caso o usuário seja do papel Administrador, ele usufrui de uma interface especial que oferece opções adicionais de menu, para que ele execute as operações de Administrador (descritas na Seção 4.5.1).

A Tabela 5.1 (situada no final deste Capítulo) mostra, para cada componente da arquitetura, a quantidade de linhas de código Java do protótipo, juntamente com o número de classes. A tabela contém também informações a respeito do interpretador da Linguagem de Definição de Processo (PDLI), da interface gráfica do sistema e de outras classes utilitárias compartilhadas por todos os componentes (declarações de constantes, nomes de servidores e assim por diante). A coluna GA indica as linhas e classes geradas automaticamente pelo compilador IDL do OrbixWeb. Não foram implementados como parte dessa tese o WM e o GTM.

5.3.1 Nomes de Instâncias de Processos

O nome de uma instância de processo é formado por dois termos, separados por um sublinhado (_). O primeiro termo contém o nome do tipo de processo e o segundo termo contém o número correspondente à quantidade de instâncias desse mesmo tipo que estão em execução. Esse número é composto por três dígitos.

[1] somente quando se trata de uma tarefa semi-automática comum para permitir operação desconectada [39]

Para exemplificar a formação desse nome, tem-se `wf1_001` e `wf2_023`. O primeiro nome teve essa formação porque, quando a instância `wf1_001` foi criada, não havia nenhuma instância do tipo `wf1` em execução. Por este motivo, recebeu o sufixo `001`. O segundo nome foi assim formado porque, no momento da criação da instância `wf2_023`, já havia vinte e duas instâncias do tipo `wf2` em execução.

Com essa maneira de formar os nomes das instâncias de processo, garante-se que não podem existir duas instâncias com o mesmo nome, ao mesmo tempo. Essa atribuição de nomes é suficiente para identificar as instâncias no sistema de forma única. Assim que uma instância de processo é terminada, seu nome já pode ser reutilizado pelo sistema para novas instâncias desse mesmo tipo.

5.3.2 Nomes dos Servidores

Para referenciar um objeto servidor pelo seu nome, o prótotipo faz uso da operação `bind()`, que utiliza como parâmetro um nome (*marker*) que deve ser único para cada servidor que se deseja localizar.

Gerenciadores de Processo - O *marker* de um PM é o próprio nome da instância que gerencia pois, conforme visto na Seção anterior, os nomes das instâncias são únicos.

Gerenciadores de Tarefa - O *marker* de um TM é composto pelo nome da instância que contém a tarefa e pelo nome da tarefa, separados por um sublinhado. Por exemplo, o *marker* do TM vinculado à instância `wf3_004`, que monitora a execução da tarefa `t9` é `wf3_004_t9`.

Gerenciadores de Lista de Trabalho - Como cada WM está relacionado a um e somente um usuário e cada nome de usuário é único no sistema, o *marker* de um WM é o nome de seu usuário correspondente. Por exemplo, `Bruno` e `Elaine`.

Gerenciador de Usuários e Papéis, de Definições e de Instâncias - Por serem únicos, possuem *markers* fixos, a saber, `urm`, `dm` e `im`, respectivamente.

Gerenciador de Recuperação - Por ser único, também possui *marker* fixo: `rm`.

Gerenciador de Tarefas Cooperativas - O *marker* de um GTM, assim como o do TM, é composto pelo nome da instância que contém a tarefa cooperativa e pelo nome dessa tarefa, separados por um sublinhado. Por exemplo, o *marker* do GTM vinculado à instância `wf8_005`, que monitora a execução da tarefa cooperativa `t7` é `wf8_005_t7`.

Para executar a operação `bind()` corretamente, é necessário também passar como parâmetro dessa operação o nome da máquina (*host*) no qual se deseja invocar o servidor, já que o *orbixd* é executado em cada máquina que hospeda um servidor. Desta forma, para referenciar um servidor WorkToDo, deve-se utilizar o formato `<nome-da-máquina> <nome-do-servidor> <marker>`, sendo esta a referência utilizada nas invocações da operação `bind()`.

5.3.3 Serialização de Objetos

Algumas operações remotas de um servidor precisam retornar um objeto de um tipo que não é pré-definido por CORBA[2] como resultado. Quando isto ocorre, a solução geralmente adotada é definir esse tipo na IDL, para que os objetos desse tipo se tornem objetos CORBA e possam ser utilizados em retornos de operações invocadas remotamente. O cliente invoca a operação e tem como retorno da mesma uma referência a esse objeto remoto. Dessa forma, futuras invocações a métodos desse objeto serão repassadas (através do *stub*, ORB e *skeleton*) para o objeto remoto.

No protótipo, essa solução não foi adotada. A solução adotada no protótipo foi a **serialização**[3], que se trata de um recurso da linguagem Java para converter qualquer objeto no formato *string*. Essa *string* gerada pode posteriormente ser convertida novamente para a classe original, criando um novo objeto e atribuindo ao mesmo o estado do objeto que foi serializado. Assim, sempre que uma operação remota precisa retornar um objeto que não é pré-definido por CORBA, esse objeto é serializado e a *string* correspondente é retornada pela operação. Na aplicação cliente, a *string* é convertida e o objeto criado é utilizado normalmente.

A vantagem desta solução é que as invocações aos métodos do objeto retornado são feitas localmente e não utilizam a camada ORB. Assim sendo, as invocações se tornam mais rápidas e menos propensas a falhas.

Duas desvantagens desta solução são apontadas abaixo.

1. As alterações realizadas no objeto remoto após sua serialização não são identificadas pela aplicação cliente e;

2. As alterações realizadas na aplicação cliente não são refletidas diretamente no objeto remoto, podendo gerar conflitos com outras escritas desse objeto.

A utilização desta solução é viável somente quando o objeto remoto é utilizado para consultas rápidas, como foram os casos do protótipo.

[2]os tipos pré-definidos são `string`, `int` e demais tipos básicos

[3]Do termo em inglês *serialization*

5.3.4 OrbixWeb

Observou-se durante os testes do protótipo que eram freqüentes os erros nas invocações das operações `bind()` para localização dos servidores WorkToDo. O *orbixd* apresentava mensagens de erro tais como a seguinte:

```
org.omg.CORBA.COMM_FAILURE: araguaia.ic.unicamp.br/2025
```

Após alguns testes, descobriu-se que o motivo que levava a esses erros era uma sobrecarga na máquina onde a operação `bind()` era chamada. Tal sobrecarga é caracterizada pelo uso excessivo da memória da máquina ou pelo processo estar muito freqüentemente ocupado. Devido a esta sobrecarga na máquina, o OrbixWeb não conseguia instanciar um servidor ou, na prática, criar um novo processo Java. Caso a mesma chamada sobre a operação `bind()` fosse executada novamente em mais algumas tentativas, geralmente se conseguia obter sucesso.

Devido a isto, todas as chamadas sobre a operação `bind()` foram substituídas pelo trecho de código mostrado abaixo:

```
int retries = Servers.RETRIES;
String reason = "No reason";
while (retries > 0) {
            try {
                return RecoveryManagerHelper.bind (Servers.RECOVERY_MANAGER,
                    Servers.RECOVERY_MANAGER_HOST);
            }
            catch (org.omg.CORBA.SystemException exc) {
                reason = exc.toString ();
                retries -;
            }
}
System.out.println ("Can't connect to Recovery Manager:  " + reason);
return null;
```

Desta forma, a operação `bind()` é chamada `RETRIES` vezes (valor definido na classe `Servers`). Caso não haja sucesso nas chamadas da operação `bind()`, um erro na ativação dos servidores é gerado. No caso do protótipo, o atributo `RETRIES` teve seu valor fixado em 5 tentativas, apesar de se ter observado que normalmente bastavam duas ou três tentativas para o sucesso da operação.

5.3.5 Repositórios

Os repositórios utilizados no protótipo foram implementados através de uma tabela *hash* (um objeto da classe *Hashtable* de Java). Essa tabela armazena um elemento em cada posição. O recurso de serialização de objetos de Java foi novamente utilizado com o fim de tornar os repositórios do WorkToDo persistentes.

A tabela é serializada e então gravada em um arquivo cujo nome é o nome do repositório acrescido da extensão `.rep` como, por exemplo, `ProcessManagers.rep`. Caso se queira recuperar o repositório, a tabela *hash* é reconstruída e utiliza o conteúdo lido do arquivo em questão.

5.4 Exemplo

Nesta Seção, é descrito um exemplo de *workflow* no WorkToDo. Neste exemplo, é criado um processo para especificação e desenvolvimento de um projeto de *software.*

A Figura 5.1 mostra uma representação do processo em forma de diagrama, em que os retângulos representam as tarefas e os nomes abaixo deles representam suas respectivas entidades processadoras. As dependências entre as tarefas são expressas por meio de setas nomeadas com `SUCCEEDED` ou `FAILED`.

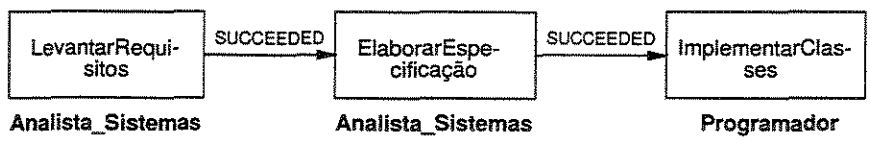

Figura 5.1: Diagrama do processo do exemplo

O WorkToDo utiliza-se da especificação de um processo na Linguagem de Definição de Processo (PDL), como se observa a seguir:

```
WORKFLOW CriacaoSistemaSoftware {
    FILE Requisitos {
        NAME "/processos/arquivos/Requisitos.txt";
    }
    FILE Especificacao {
        NAME "/processos/arquivos/Especificacao.doc";
    }
    FILE Classes {
        NAME "/processos/arquivos/Classes.html";
    }
    TASK LevantarRequisitos:  RedigirDocumento {
```

```
        DEPENDS;
        IN_CONTEXT;
        OUT_CONTEXT Requisitos;
    }
    TASK ElaborarEspecificacao: RedigirDocumento {
        DEPENDS LevantarRequisitos → SUCCEEDED;
        IN_CONTEXT Requisitos;
        OUT_CONTEXT Especificacao;
    }
    TASK ImplementarClasses: CriarClasse {
        DEPENDS ElaborarEspecificacao → SUCCEEDED;
        IN_CONTEXT Especificacao;
        OUT_CONTEXT Classes;
    }
}
```

O processo CriacaoSistemaSoftware contém 3 tarefas, identificadas por LevantarRequisitos, ElaborarEspecificacao e ImplementarClasses. Cada uma dessas tarefas instancia um modelo de tarefa (RedigirDocumento e CriarClasse).

A tarefa ElaborarEspecificacao é instância do modelo de tarefa RedigirDocumento. Ela somente será iniciada se a tarefa LevantarRequisitos for executada com sucesso. O dado de entrada de ElaborarEspecificacao é o arquivo Requisitos.txt e o dado de saída é o arquivo Especificacao.doc. Esses dados são identificados, respectivamente, por Requisitos e Especificacao e estão localizados no diretório /processos/arquivos.

O modelo de tarefa RedigirDocumento é definido da seguinte forma:

```
TASK RedigirDocumento {
    TYPE Semi-automatic;
    ROLE Analista_Sistemas;
    DESCRIPTION "Redigir um documento usando um
        editor de texto";
    APPLICATION EditorTexto;
    PRIORITY 20;
    DEADLINE 48 HOURS;
}
```

A tarefa é do tipo semi-automática e a responsabilidade por sua execução é dos usuários do papel Analista_Sistemas. A tarefa contém uma descrição textual de como o usuário deve proceder para executá-la. A aplicação relacionada a ela é EditorTexto. As

características dessa aplicação estão definidas separadamente. São também indicados: a prioridade (20) e o prazo para execução da tarefa (48 horas, ou seja, dois dias).

O modelo de tarefa `CriarClasse` é definido abaixo:

```
TASK CriarClasse {
     TYPE Cooperative;
     ROLE Programador;
     DESCRIPTION "Implementar uma classe do projeto do
          sistema";
     APPLICATION;
     PRIORITY 30;
     DEADLINE 168 HOURS;
     USERS "Paulo", "Joao";
}
```

A tarefa é do tipo cooperativa e a responsabilidade por sua execução é dos usuários do papel `Programador`. A tarefa contém uma descrição textual de como o usuário deve proceder para executá-la. Sua prioridade é 30 e o prazo máximo para sua execução é de 168 horas (uma semana). A cláusula `USERS` indica que a tarefa tem um grupo definido para executá-la, composto pelos usuários `Paulo` e `Joao`, além do usuário `Hudo`, que é o coordenador da transação de grupo.

A definição da aplicação `EditorTexto` é mostrada a seguir.

```
APPLICATION EditorTexto {
     NAME "/n/gnu/emacs/bin/emacs";
     SIZE 3125;
     OS "Linux";
     CPU "486";
     INSTALLER;
     HOSTS "iguacu.ic.unicamp.br", "tuiuiu.ic.unicamp.br";
}
```

O nome do arquivo executável da aplicação, localizado no diretório `/n/gnu/emacs/bin` é `emacs` e seu tamanho é 3125 Kb. A aplicação é executada no sistema operacional `Linux` e o processador mínimo necessário para sua execução é um 486. Nenhum instalador foi especificado (cláusula `INSTALLER`) então, o único arquivo necessário para executar a aplicação é o seu próprio executável. A aplicação está disponível em duas máquinas (*hosts*), a saber: `{iguacu, tuiuiu}.ic.unicamp.br`.

A seguir, é dada a explicação passo a passo de um exemplo de execução de uma instância desse processo:

1. Um usuário do papel criador do tipo de processo `CriacaoSistemaSoftware` cria uma instância do processo. Devido a isto, um Gerenciador de Processo é criado para gerenciar a execução da instância;

2. A tarefa `LevantarRequisitos` não possui dependências. Assim sendo, logo que a instância é iniciada, o Escalonador já considera essa tarefa como pronta para executar. Como o modelo de tarefa correspondente (`RedigirDocumento`) define que essa tarefa é semi-automática, o Despachante notifica os Gerenciadores de Lista de Trabalho de todos os usuários do papel `Analista_Sistemas` (ou os usuários que se enquadram na política de notificação adotada) sobre a disponibilidade dessa tarefa para execução;

3. Os WMs notificados adicionam um *workitem* correspondente à tarefa nas *worklists* de seus respectivos usuários;

4. Um dos usuários do papel `Analista_Sistemas` seleciona o *workitem* correspondente à tarefa `LevantarRequisitos` e o executa, informando em seguida o resultado da execução (`SUCCEEDED`);

5. Quando a tarefa `LevantarRequisitos` atinge o estado `SUCCEEDED`, o Escalonador detecta que as dependências da tarefa `ElaborarEspecificacao` foram satisfeitas e, portanto, ela já pode ser executada. O Despachante notifica os WMs dos usuários do papel `Analista_Sistemas` e adiciona um *workitem* para a tarefa nas *worklists* dos usuários;

6. Um dos usuários do papel `Analista_Sistemas` seleciona o *workitem* e o executa. Assim que conclui a tarefa, o usuário informa ao PM o resultado da execução (`SUCCEEDED`);

7. Quando a tarefa `ElaborarEspecificacao` atinge o estado `SUCCEEDED`, o Escalonador detecta que as dependências da tarefa `ImplementarClasses` foram satisfeitas e, portanto, ela já pode ser executada;

8. Como `ImplementarClasses` é uma tarefa cooperativa, o Despachante notifica somente o WM do usuário responsável pelo processo, inicialmente. Assim que o usuário responsável pelo processo seleciona a tarefa cooperativa para execução, o WM cria um GTM para gerenciar a execução dessa tarefa cooperativa e o GTM, por sua vez, cria uma transação de grupo para o usuário coordenador (que é o usuário responsável pelo processo);

9. Após isso, o Despachante notifica os WMs dos demais usuários do grupo (definidos no modelo de tarefa `CriarClasse`) sobre a disponibilidade da tarefa para execução;

10. Para cada usuário do grupo que seleciona a tarefa cooperativa para execução, o GTM cria uma transação de usuário;

11. Conforme os usuários do grupo vão terminando de executar suas transações de usuário, eles informam o resultado da execução de sua transação (`COMMIT` ou `ABORT`) ao GTM, que por sua vez, repassa este resultado ao WM. Caso o resultado da execução da transação seja `COMMIT`, o WM associa a esse resultado o estado `SUCCEEDED`. Caso o resultado seja `ABORT`, é associado ao resultado o estado `FAILED`;

12. Assim que o usuário coordenador recebe todos os resultados das transações de usuário vinculadas à sua transação de grupo, ele informa o resultado da transação cooperativa ao GTM, que verifica se o resultado informado é válido. Caso haja alguma transação vital abortada, o resultado da transação cooperativa é `FAILED`;

13. Após a execução dessas tarefas, o Escalonador detecta que o processo foi completado;

14. O PM notifica o IM, que armazena o estado final da instância no repositório de processos terminados. Após essa operação, o PM é desativado;

15. O sistema envia uma notificação para o usuário responsável pelo processo, informando que o processo foi concluído.

A Figura 5.2 mostra a *worklist* de um usuário que assume os papéis `Analista_Sistemas` e `Programador`. A *worklist* pertence ao usuário `Paulo` que, no momento, está conectado ao SGWF. Sua *worklist* contém 3 *workitems* de diferentes instâncias do processo `CriacaoSistemaSoftware`. O segundo *workitem* está selecionado (estado `SELECTED`) e o primeiro e o terceiro estão prontos para serem executados (estado `READY`). No caso de existirem tarefas com mesmo nome, elas são diferenciadas pelo nome da instância à qual estão vinculadas.

A *worklist* contém diversas informações a respeito dos *workitems*, tais como o nome da instância de processo e da tarefa correspondente, o tipo do *workitem* e seu estado atual. O usuário pode interagir com os *workitems* através da *worklist* e com o SGWF como um todo a partir dos menus da interface do WorkToDo.

A Figura 5.3 mostra a lista de instâncias em execução em um determinado momento. Essa lista contém informações como: o nome da instância em execução, o tipo de processo ao qual ela pertence, o nome e endereço IP da máquina onde seu respectivo PM está sendo executado.

A Figura 5.4 mostra em detalhes o estado de uma instância em particular (`CriacaoSistemaSoftware_002`). A instância foi criada em `03/10/2002` às `15:02` hs e seu usuário responsável (*User in charge*) é `Hudo`. São mostradas também as informações detalhadas

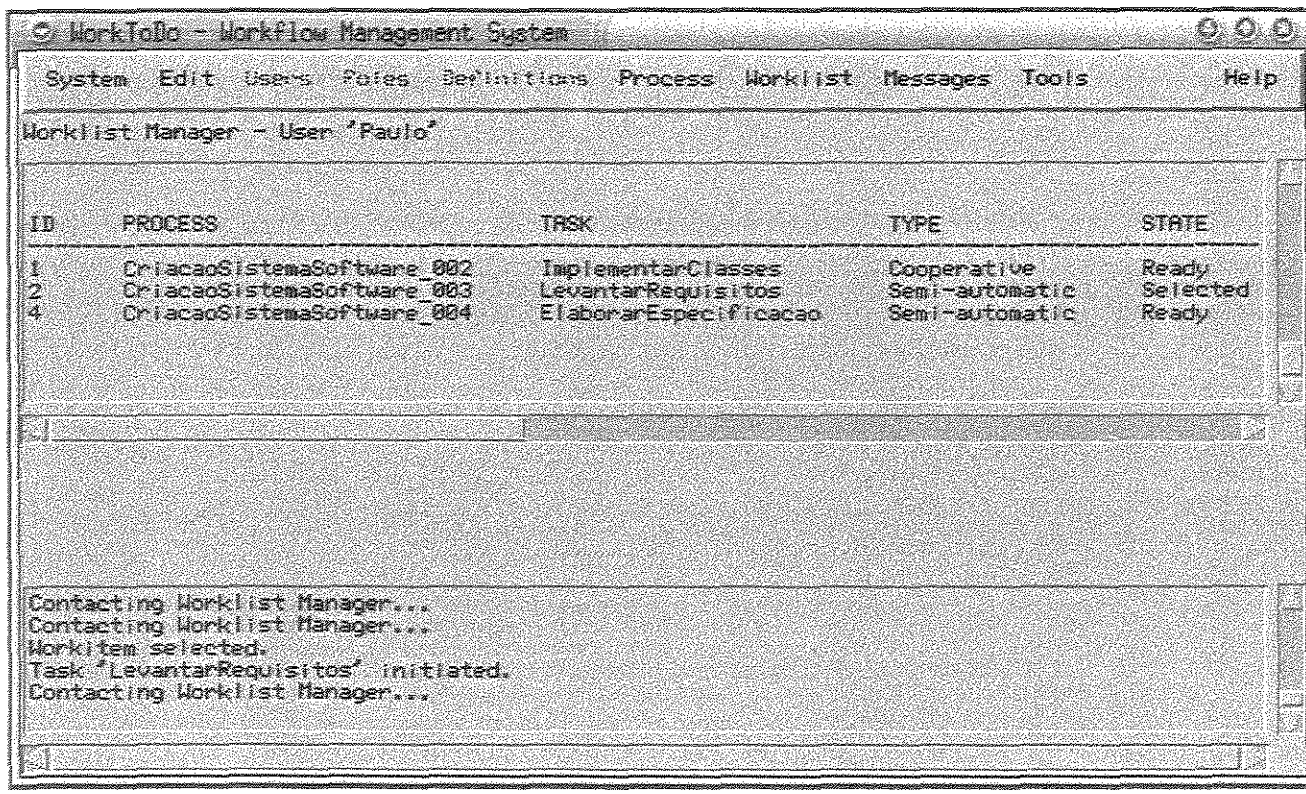

Figura 5.2: Exemplo de *worklist*

sobre cada uma das tarefas do processo, tais como: tipo, estado atual, usuário que a executou ou a está executando, tempo limite para execução, entre outras.

A Figura 5.5 mostra o recebimento de mensagens por um usuário. A primeira e a terceira mensagens se referem ao término da execução das instâncias de processo `CriacaoSistemaSoftware_002` e `CriacaoSistemaSoftware_004`; a segunda mensagem alerta sobre a falha na execução da tarefa `ElaborarEspecificacao`, pertencente à instância `CriacaoSistemaSoftware_004` e é dirigida ao responsável pelo processo, para que este tome as devidas providências. Para cada mensagem, é indicado também o horário de recebimento.

As Figuras 5.6 e 5.7 mostram as interfaces para execução de uma transação de grupo e de uma transação de usuário, respectivamente. Ao requisitar um objeto para execução (uma classe no caso do processo `CriacaoSistemaSoftware`), a interface da Figura 5.8 é disponibilizada para que o usuário selecione o objeto (classe `Interface.java`) para leitura (*read*), para escrita (*write*), para cópia (*copy*), para empréstimo (*loan*) ou para concessão (*concession*). Conforme visto na Figura 5.7, a requisição de um objeto pode ser:

Não bloqueante (*Non-blocking*) - Caso o objeto já esteja sendo usado por outro

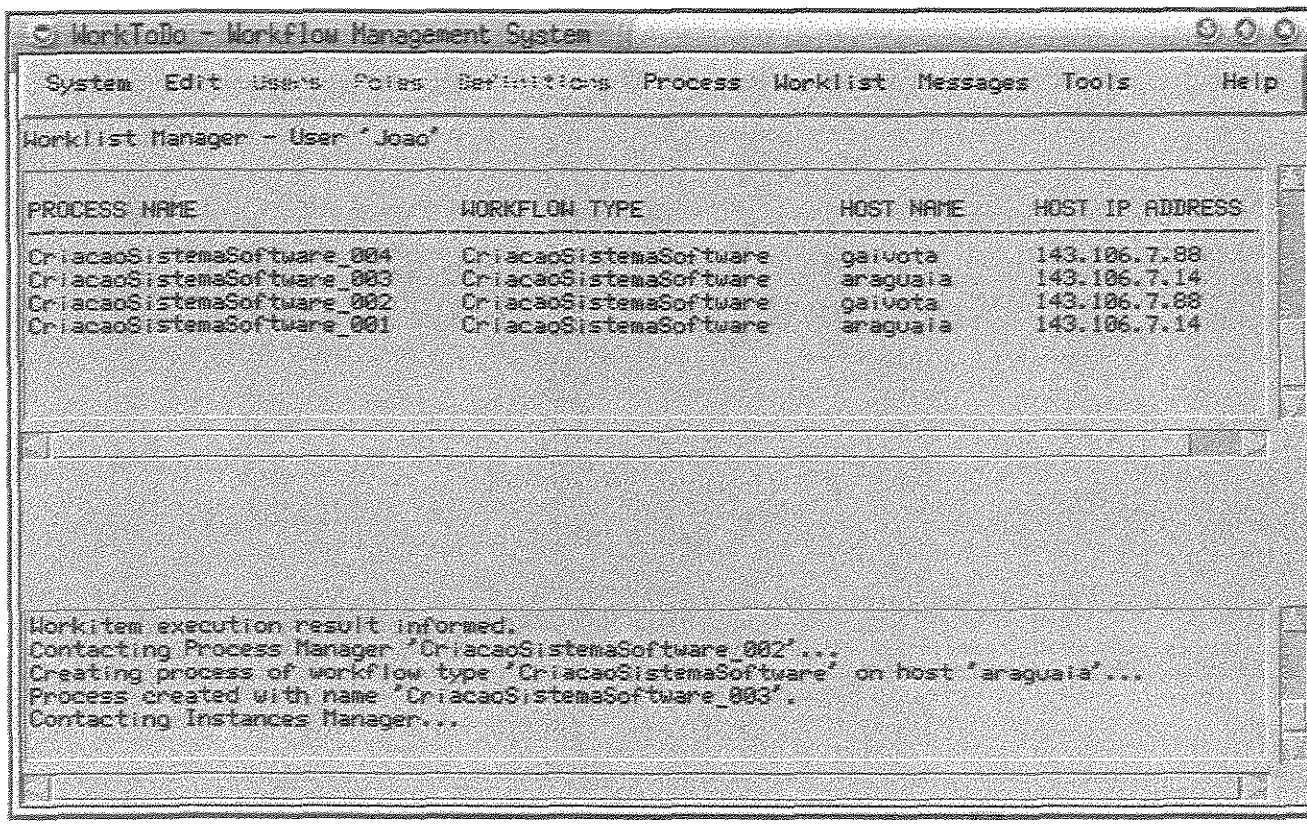

Figura 5.3: Exemplo de listagem de instâncias em execução

usuário, a interface informa ao usuário requisitante que o objeto já está sendo utilizado, sem bloquear a requisição.

Bloqueante (*Blocking*) - A requisição fica bloqueada até que o usuário que está usando o objeto requisitado libere esse objeto.

Através da opção `Edit`, o usuário pode realizar alterações sobre o objeto como desejar. Assim que o usuário termina a edição sobre o objeto, ele deve liberá-lo (opção `Release`) para que os outros usuários do grupo possam utilizá-lo, viabilizando-se assim a troca controlada de informações entre os participantes do grupo. A interface disponibilizada pelo sistema para a liberação do objeto é mostrada na Figura 5.9. Através dela, o usuário pode liberar o objeto com dois tipos de liberação: `COMMIT`, validando as alterações que realizou sobre o objeto e `ABORT`, abortando essas alterações e fazendo com que sejam desconsideradas. Nesse último caso, o objeto assume o estado anterior às alterações do usuário.

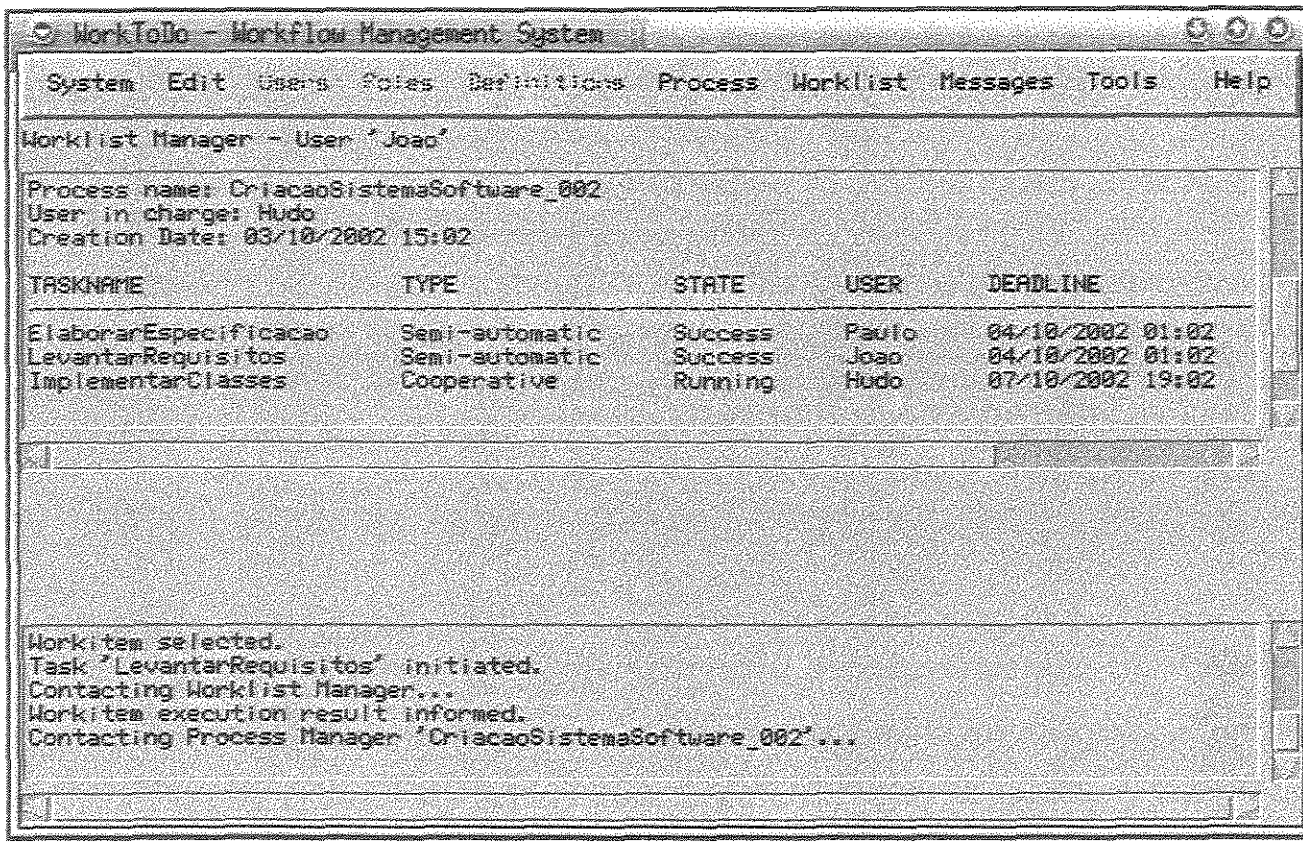

Figura 5.4: Exemplo de consulta ao estado de uma instância de processo

5.5 Resultados

O sistema distribuído utilizado nos testes do protótipo é composto por estações de trabalho (PC's) que contêm o sistema operacional Linux e por máquinas Sparc que contêm o sistema operacional Solaris, conectados através de uma rede local de 10 Mb.

Na realização dos testes, o URM, o DM, o IM, o GTM e o RM foram executados um em cada máquina diferente do sistema distribuído. A máquina em que o PM seria executado era escolhida pelo usuário que criava a instância em tempo de execução, no momento da criação da instância. Quanto às máquinas em que os TMs eram executados, estas eram escolhidas de acordo com as definições de suas respectivas aplicações, pois o TM deve ser executado na máquina em que a aplicação está disponível para execução. Quanto às máquinas em que os WMs eram executados, estas eram as mesmas em que os seus respectivos usuários estavam conectados ao SGWF. A Tabela 5.2 apresenta as máquinas escolhidas para cada gerenciador.

No WorkToDo, foram testadas as execuções de tarefas automáticas, semi-automáticas comuns e semi-automáticas cooperativas. Em todas obteve-se o resultado esperado.

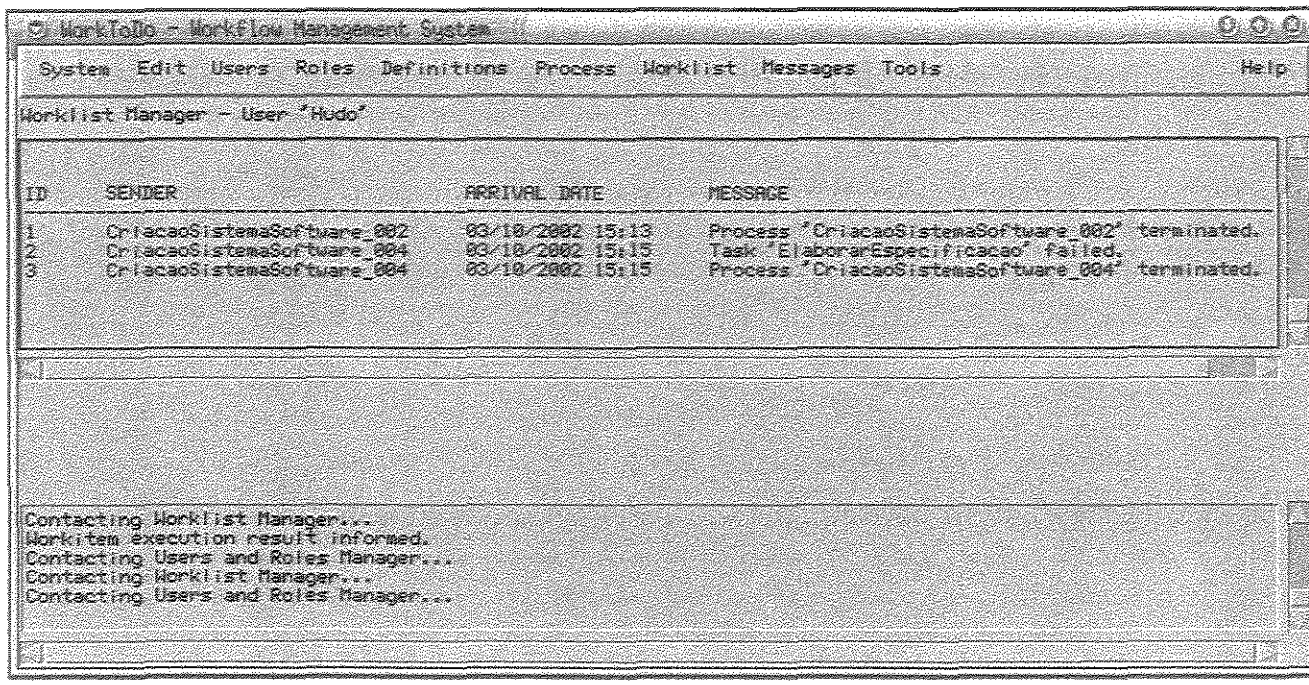

Figura 5.5: Exemplo da lista de mensagens de um usuário

A execução de tarefas automáticas teve êxito, pois o PM criou os TMs para essas tarefas corretamente e os TMs gerenciaram a execução das mesmas de forma satisfatória, em todos os casos observados. Conforme visto na Seção 5.2, a recuperação de TMs não foi implementada.

A execução de tarefas semi-automáticas comuns também foi bem sucedida e, mesmo nos casos de queda de conexão, não foram geradas inconsistências. Como o estado do WM é persistente, ele pode ser recuperado após uma falha, a partir da mesma máquina ou de outra máquina, conforme visto na Seção 4.7.2.

O trabalho cooperativo se mostrou possível de ser executado e o sistema se mostrou estável mediante quedas de conexão, sem gerar nenhum tipo de inconsistência. As transações da hierarquia têm seus estados persistentes. Estes estados podem ser restaurados em caso de falha e a execução pode ser retomada após a falha, através do uso de *checkpoints* e do *log*, conforme visto na Seção 4.7.2.

Nos casos acima, a recuperação de falhas (exceto em relação aos TMs, pois não foi implementada) foi satisfatória e as mensagens de erro auxiliaram na identificação e resolução dos problemas gerados pelas falhas ocorridas nos testes.

	DM			IM		
	GA	Escritas	Total	GA	Escritas	Total
Com comentários	1103	495	1598	775	463	1238
Sem comentários	766	247	1013	546	244	790
Número de classes	9	6	15	9	3	12

	URM			PM		
	GA	Escritas	Total	GA	Escritas	Total
Com comentários	1252	837	2089	1127	3225	4352
Sem comentários	869	415	1284	798	1949	2747
Número de classes	9	7	16	18	16	34

	TM			WM		
	GA	Escritas	Total	GA	Escritas	Total
Com comentários	2067	794	2861	1306	1766	3072
Sem comentários	1475	496	1971	912	1005	1917
Número de classes	42	10	52	17	9	26

	RM			GTM		
	GA	Escritas	Total	GA	Escritas	Total
Com comentários	569	372	941	11285	7159	18444
Sem comentários	402	238	640	8165	3484	11649
Número de classes	9	4	13	148	26	174

	INTERFACE			PDLI		
	GA	Escritas	Total	GA	Escritas	Total
Com comentários	-	1821	1821	-	1668	1668
Sem comentários	-	1354	1354	-	1286	1286
Número de classes	-	5	5	-	14	14

	UTIL			TOTAL		
	GA	Escritas	Total	GA	Escritas	Total
Com comentários	-	1754	1754	19484	20354	39838
Sem comentários	-	1126	1126	13933	11844	25777
Número de classes	-	12	12	261	112	373

Tabela 5.1: Linhas de código do protótipo

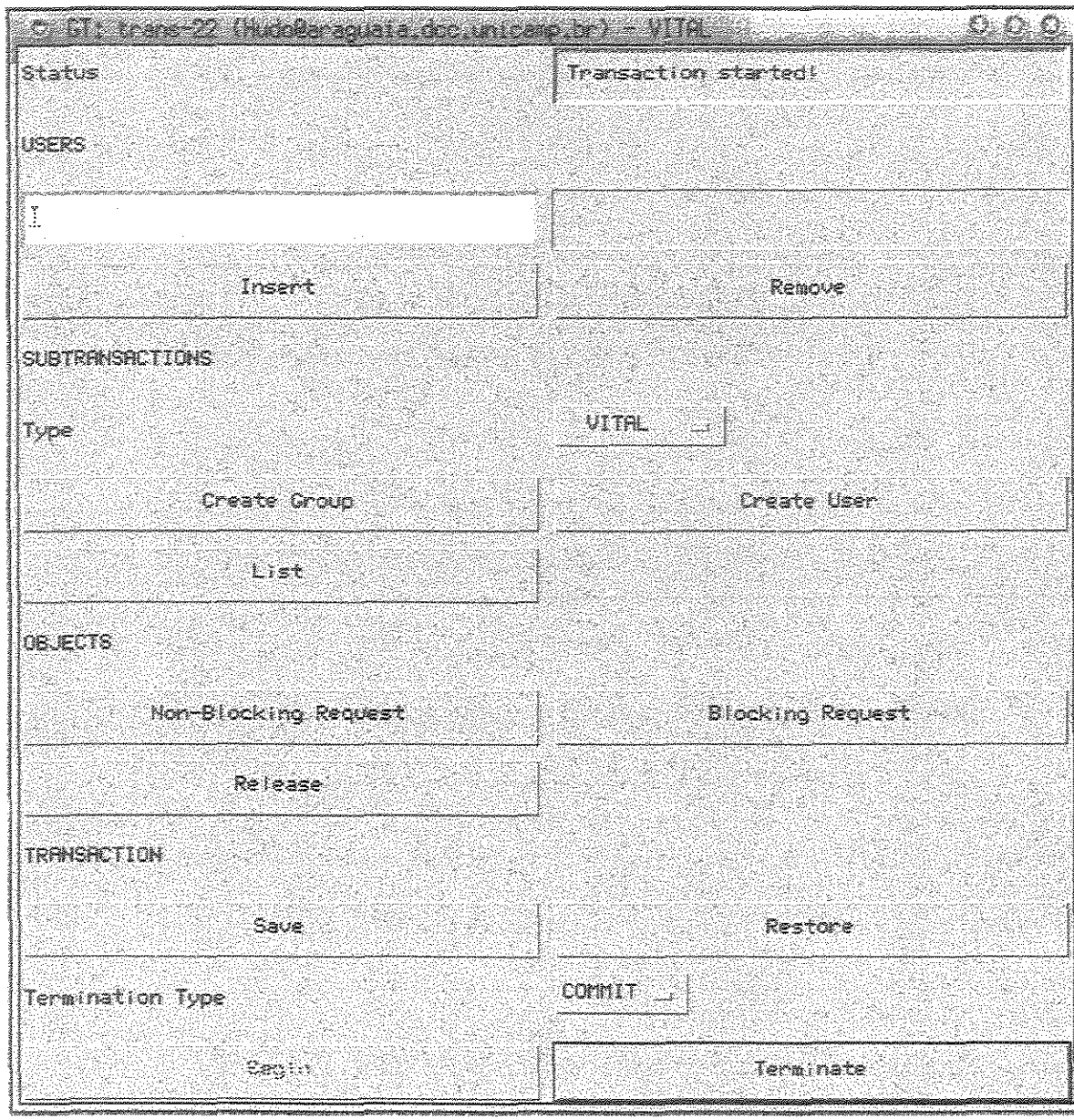

Figura 5.6: Exemplo da interface da transação de grupo

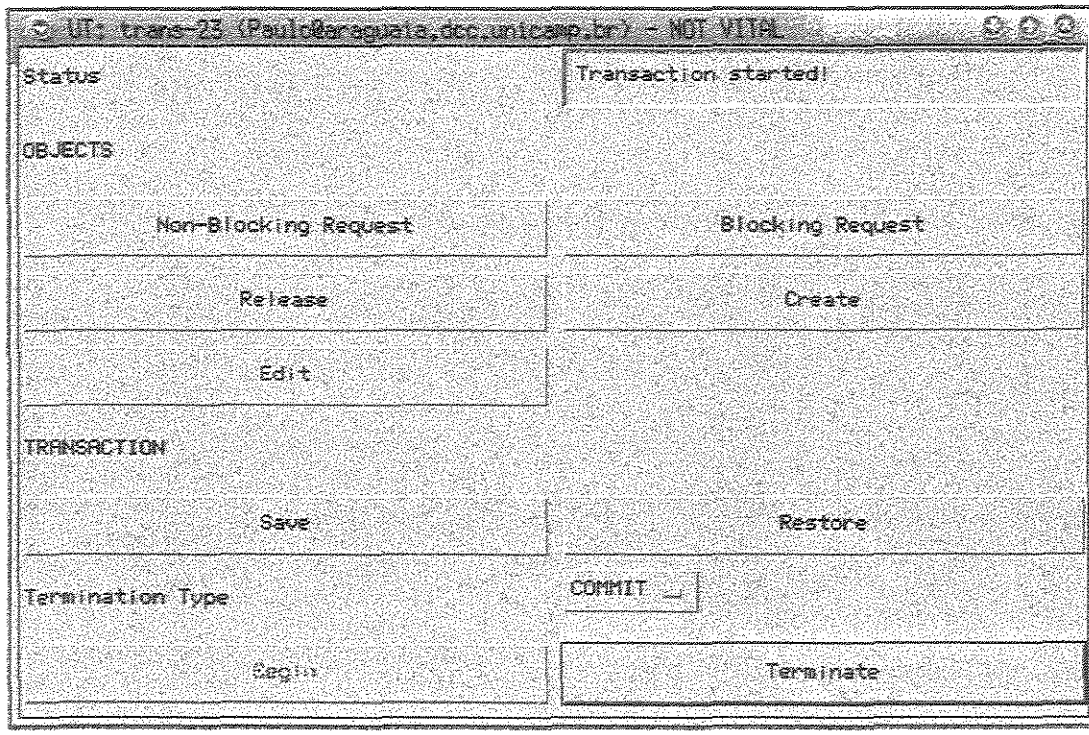

Figura 5.7: Exemplo da interface da transação de usuário

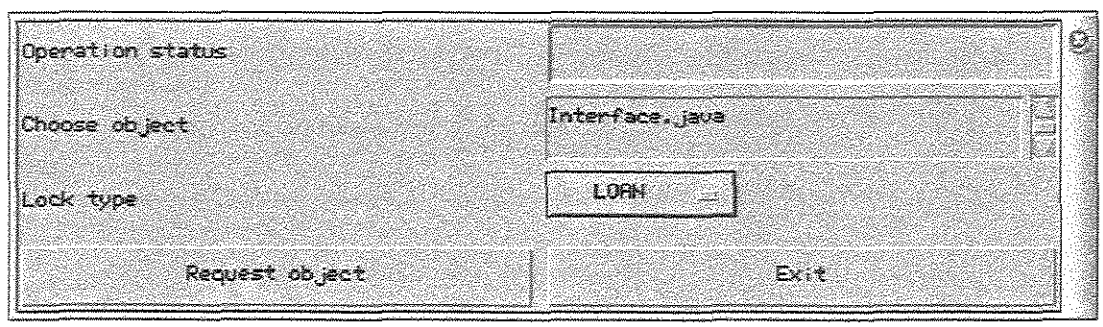

Figura 5.8: Exemplo de requisição de um objeto

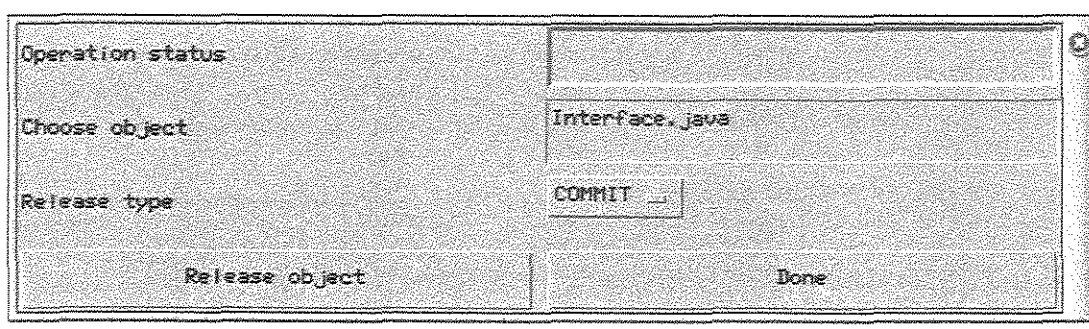

Figura 5.9: Exemplo de liberação de um objeto

Componente	Máquina
URM	`araguaia.ic.unicamp.br`
DM	`xingu.ic.unicamp.br`
IM	`iguacu.ic.unicamp.br`
GTM	`tuiuiu.ic.unicamp.br`
RM	`gaivota.ic.unicamp.br`
PM	Escolhida pelo usuário no momento da criação da instância
TM	Escolhida de acordo com a definição da aplicação correspondente à tarefa
WM	Mesma em que o usuário correspondente está conectado ao SGWF

Tabela 5.2: Máquinas utilizadas nos testes do protótipo

Capítulo 6

Conclusões

Nesta dissertação, foi apresentado o WorkToDo, que é um SGWF capaz de atender a requisitos de aplicações para desenvolvimento de projeto como CAD, CASE, por exemplo. O sistema é capaz de executar tarefas automáticas, tarefas manuais, tarefas semi-automáticas comuns e, especialmente, tarefas semi-automáticas cooperativas, dentro das quais são oferecidas maiores facilidades de colaboração entre usuários de um grupo.

As tarefas automáticas são executadas através da invocação de uma aplicação. No exemplo da criação de um sistema de *software*, uma tarefa automática seria a geração automática da documentação do *software* (Java API) através da invocação da aplicação `javadoc`.

As tarefas manuais e semi-automáticas comuns são executadas através da visualização, seleção e execução de *workitems*, que são criados nas *worklists* dos usuários conectados ao SGWF conforme a política de notificação adotada. No exemplo da criação de um sistema de *software*, uma tarefa manual seria um telefonema para o cliente confirmando os dados coletados na execução da tarefa `LevantarRequisitos`. Já uma tarefa semi-automática seria a execução da tarefa `ElaborarEspecificacao`, em que o analista de sistemas conta com o auxílio de um editor de texto.

As tarefas semi-automáticas cooperativas são visualizadas e selecionadas da mesma forma que as tarefas semi-automáticas comuns, com a diferença de que, na execução de tarefas cooperativas, são admitidas seleções simultâneas e múltiplas de um mesmo *workitem*, para que se alcance a cooperação entre os usuários do grupo. A execução destas tarefas utiliza um serviço para gerenciamento de transações [38], que permite a troca controlada de resultados das transações de grupo e de usuário. No exemplo da criação de um sistema de *software*, uma tarefa cooperativa seria a implementação das diversas classes que compõem um sistema de *software* orientado a objetos, contando com o trabalho cooperativo realizado por uma equipe de programadores.

O tratamento de falhas desse sistema é realizado através do monitoramento e recupera-

ção dos componentes que falham, garantindo a execução correta de processos (*workflows*), mesmo quando falhas ocorrem.

A utilização de regras de dependência que definem o fluxo de controle de um processo permite a recuperação de processos com tarefas que falham. É possível definir para um processo tanto recuperação com atraso, como recuperação com avanço. No caso de recuperação com atraso, tarefas de compensação são executadas para desfazer tarefas já executadas antes da falha. No caso de recuperação com avanço, um caminho alternativo é executado.

O sistema leva em conta a distribuição dos componentes e usuários, já que *workflows* envolvem vários usuários e aplicativos localizados em diferentes pontos de um sistema distribuído. Para atender tal necessidade, o sistema foi implementado sobre uma plataforma distribuída (CORBA), incorporando algumas das vantagens desta infra-estrutura de comunicação. Além disso, o sistema foi implementado utilizando-se uma linguagem multiplataforma (Java), pois os usuários que participam na execução das tarefas de um processo podem utilizar diferentes computadores em diferentes ambientes. Com isso, o SGWF pode ser executado em diferentes sistemas operacionais e plataformas de *hardware*, aumentando sua flexibilidade de uso.

6.1 Comparação entre o WorkToDo e os Trabalhos Relacionados

Tanto o WorkToDo como *Sagas* utilizam a noção de **recuperação com atraso** e **recuperação com avanço**, como forma de tratar falhas. Contudo, a recuperação com avanço no *Sagas* permite que um *Saga* seja reiniciado do ponto em que parou com o cancelamento de passos que não foram terminados e a re-execução do caminho interrompido. No WorkToDo, é possível definir caminhos alternativos em caso de falha de tarefa, através da especificação de regras de execução.

Tanto o WAMO como o WorkToDo incorporaram a estruturação de atividades em forma de árvore. O WorkToDo e o WAMO utilizam o conceito de atividades vitais e não vitais. Finalmente, o WorkToDo faz uso da noção de regras de dependência, assim como o WAMO.

O WorkToDo, assim como o CoAct, aborda especialmente tarefas cooperativas. Em ambos a cooperação envolve a troca de resultados entre espaços de trabalho. No CoAct, as formas de cooperação são previamente definidas na especificação da atividade cooperativa através de regras de entrelaçamento. No WorkToDo, a cooperação ocorre através da troca de resultados entre áreas de trabalho, realizada através de requisições de usuários conforme as necessidades da tarefa cooperativa. Os entrelaçamentos possíveis não fazem parte da

definição do processo, sendo estabelecidos em tempo de execução.

Assim como o modelo ConTract, o WorkToDo possui um serviço de gerenciamento de transações para gerenciar uma transação cooperativa, que é uma estrutura complexa, assim como um ConTract. Além disso, o WorkToDo também trata recuperação de falhas, permitindo a reinicialização de componentes que falharam e a continuação de processos a partir do ponto de falha.

Similar ao METEOR, os processos no WorkToDo podem possuir tarefas transacionais e não transacionais. Entretanto, no WorkToDo, as tarefas não transacionais são tarefas automáticas que não satisfazem as propriedades ACID e as tarefas transacionais são transações de grupo. Além disso, o WorkToDo também usa a camada ORB de CORBA como infra-estrutura de comunicação.

O Exotica oferece apenas recuperação com avanço, enquanto que o WorkToDo oferece recuperação com avanço e recuperação com atraso. Além disso, o Exotica usa replicação de componentes para aumentar a tolerância a falhas e a escalabilidade. O WorkToDo possui um componente especial (RM) para monitorar os PMs e re-instanciá-los quando necessário. Para aumentar a escalabilidade, um PM é criado para controlar cada nova instância de processo.

6.2 Contribuições

Este trabalho teve, como principais contribuições:

- A apresentação de um modelo e de uma arquitetura flexíveis, que não se restringem ao mapeamento para uma linguagem de programação e uma infra-estrutura únicas, apesar da escolha de Java e CORBA para a implementação de seu protótipo;
- O sistema viabiliza a execução de tarefas cooperativas, cujo gerenciamento é complexo, sem apresentar restrições quanto à localização dos usuários;
- A troca controlada dos resultados intermediários dos trabalhos dos usuários na execução de uma tarefa cooperativa;
- A recuperação na ocorrência de falhas de tarefa e de componente;
- A validação do modelo através da implementação de um protótipo.

6.3 Trabalhos Futuros

Como trabalhos futuros, foram identificados alguns aspectos, descritos abaixo:

- O sistema proposto pode ser estendido para oferecer tarefas transacionais ACID;
- A recuperação de falhas de outros componentes do sistema de *workflow*, tais como o Gerenciador de Usuários e Papéis, o Gerenciador de Definições e o Gerenciador de Instâncias;
- O armazenamento dos repositórios em um banco de dados ao invés de arquivos, a fim de aumentar a confiabilidade e segurança das informações;
- O desenvolvimento de ferramentas gráficas para a especificação e visualização das definições de tipos de processos, tarefas e aplicações;
- O desenvolvimento de ferramentas de administração que facilitem o gerenciamento do estado do sistema;
- A implementação de alguns aspectos que não foram implementados no protótipo, tais como o cancelamento de um processo em execução, a utilização de dados do tipo *Query*, entre outros;
- A implementação das demais políticas de notificação de usuários para tarefas semi-automáticas comuns;
- A implementação de sub-atividades como atividades válidas no protótipo;
- A implementação dos prazos de tarefas expressos em dias, pois neste protótipo, só podem ser expressos em horas;
- A implementação de políticas de prioridades de tarefas no Despachante;
- A sincronização de tarefas, tanto para controlar o compartilhamento de recursos como para garantir requisitos de ordenação entre tarefas.

Além desses aspectos, outras melhorias podem ser introduzidas no WorkToDo, devido à flexibilidade de sua arquitetura.

Referências Bibliográficas

[1] H. R. Almeida and M. B. F. Toledo. Cooperation and Failure Recovery in a Workflow Management System. In *Proceedings of First International Seminar on Advanced Research in Electronic Business (EBR'2002)*, pages 85–93, Rio de Janeiro, RJ, Brazil, November 2002. PUC-Rio and UFMG.

[2] G. Alonso, D. Agrawal, A. El Abbadi, R. Günthör, M. Kamath, and C. Mohan. Exotica: A Project on Advanced Transaction Management and Workflow Systems. *ACM SIGOIS Bulletin*, 16(1), August 1995.

[3] G. Alonso, D. Agrawal, El Abbadi, R. Günthör, M. Kamath, and C. Mohan. Exotica/FMQM: A Persistent Message-based Architecture for Distributed Workflow Management. Technical Report RJ9912, IBM Almaden Research Center, 1994.

[4] G. Alonso, D. Agrawal, El Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced Transaction Models in Workflow Contexts. In *Proc. of the 12th International Conference on Data Engineering.* New Orleans, USA, February 1996.

[5] G. Alonso, D. Agrawal, El Abbadi, and C. Mohan. Functionality and Limitations of Current Workflow Management Systems. Technical report, IBM Almaden Research Center, 1997.

[6] G. Alonso, R. Günthör, M. Kamath, D. Agrawal, El Abbadi, and C. Mohan. Exotica/FMDC: Handling Disconnected Clients in a Workflow Management System. In *Proc. of the 3rd International Conference on Cooperative Information Systems*, pages 99–100. Vienna, Austria, 1995.

[7] G. Alonso, M. Kamath, D. Agrawal, El Abbadi, R. Günthör, and C. Mohan. Failure Handling in Large Scale Workflow Management Systems. Technical Report RJ9913, IBM Almaden Research Center, 1994.

[8] D. Barbara, S. Mehrota, and M. Rusinkiewicz. INCAS: A Computation Model for Dynamic Workflows in Autonomous Distributed Environments. Technical report, Matsushita Information Technology Laboratory, 1994.

[9] G. Booch, J. Rumbaugh, and I. Jacobson. *UML – Guia do Usuário.* Editora Campus, 2000.

[10] Y. Breitbart, A. Deacon, H. Schek, A. Sheth, and G. Weikum. Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multi-system Workflows. 22(3), September 1993.

[11] J. Eder and W. Liebhart. The Workflow Activity Model WAMO. In *Proc. of the 3rd International Conference on Cooperative Information Systems*, pages 87–98. Vienna, Austria, 1995.

[12] A. K. Elmagarmid, editor. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, 1992.

[13] H. Garcia-Molina and K. Salem. Sagas. In *Proc. of ACM SIGMOD Conference on Management of Data*, pages 249–259, May 1987.

[14] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3:119–153, 1995.

[15] S. Jablonski. Functional and Behavioral Aspects of Process Modelling in Workflow Systems. In A. Benczur G. Chroust, editor, *CON 94 Workflow Management: Challenges, Paradigms and Products*, pages 113–133. R. Oldenburg, 1994.

[16] M. Kamath and K. Ramamritham. Bridging the Gap Between Transaction Management and Workflow Management. In *NSF Workshop on Workflow and Process Automation in Information Systems.* Athens, Georgia, 1996.

[17] K. Kochut, A. Sheth, and J. Miller. Orbwork: A CORBA-based Fully Distributed, Scalable and Dynamic Workflow Enactment Service for METEOR2. Technical report, Department of Computer Science, University of Georgia, 1998.

[18] N. Krishnakumar and A. Sheth. Specification of Workflows with Heterogeneous Tasks in METEOR. Technical report, Bell Communications Research Report, 1994.

[19] J. Miller, D. Palaniswami, A. Sheth, K. Kochut, and H. Singh. Webwork: METEOR2's Web-based Workflow Management System. *Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems*, 10(2):185–215, March 1998.

[20] J. E. B. Moss. Nested Transactions and Reliable Distributed Computing. pages 33–39, July 1982.

[21] OMG. The Common Object Request Broker: Architecture and Specification. Technical Report OMG Revision 2.0, Object Management Group, 1996.

[22] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. John Wiley & Sons, Inc., Second edition, 1998.

[23] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wäsch, and P. Muth. Towards a Cooperative Transaction Model – The Cooperative Activity Model. In *VLDB Conference*, Zurich, Switzerland, 1995.

[24] J. Veijalainen, A. Lehtola, and O. Pihlajamaa. Research Issues in Workflow Systems. Technical report, VTT Information Technology, Finland, 1995.

[25] H. Wächter and A. Reuter. The ConTract Model. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 7, pages 219–263. Morgan Kaufmann Publishers, 1992.

[26] WfMC. The Workflow Reference Model. Technical Report TC00-1003, Workflow Management Coalition Group, January 1995.

[27] D. Worah and A. Sheth. Transactions in Transactional Workflows. In L. Kerschberg S. Jajodia, editor, *Advanced Transaction Models and Architectures*, pages 3–34. Kluwer Academic Publishers, 1997.

[28] X. Wang. Implementation and Performance Evaluation of CORBA-based Centralized Workflow Schedulers. University of Georgia, 1995.

[29] Workflow Management Coalition – http://www.wfmc.org, Agosto de 2002.

[30] The Workflow Portal – http://www.e-workflow.org, Agosto de 2002.

[31] Oracle FAQ – http://www.orafaq.com/glossary/faqglosa.htm, Outubro de 2002.

[32] Object Management Group – http://www.omg.org, Agosto de 2002.

[33] Common Object Request Broker Architecture – http://www.corba.org, Setembro de 2002.

[34] OrbixWeb Programmers's Guide - IONA Technologies PLC – http://www.iona.com, Setembro de 2002.

[35] OrbixWeb Programmers's Reference - IONA Technologies PLC – http://www.iona.com, Setembro de 2002.

[36] OrbixWeb - IONA Technologies Java ORB Implementation – http://www.iona.com/products/orbixweb/index.html, Setembro de 2002.

[37] IONA – http://www.iona.com, Abril de 2002.

[38] A. Hervella. Um Serviço de Transações Cooperativas Baseado em CORBA. Tese de Mestrado – Universidade Estadual de Campinas (Unicamp), Dezembro de 2002.

[39] L. H. Reinehr. WorkToDo – Um Sistema de Gerenciamento de Workflows para Ambientes de Comunicação sem Fio. Tese de Mestrado – Universidade Estadual de Campinas (Unicamp), Julho de 2002.

[40] Java Development Kit, version 1.2 API Specification – http://java.sun.com/products/jdk/1.2/docs/api, Setembro de 2002.

Apêndice A

Gramática da Linguagem de Definição de Processo

A.1 Notação BNF

Esta Seção apresenta a notação BNF (*Backus Normal Form*), utilizada para descrever a gramática da Linguagem de Definição de Processo.

Notação	Significado
<...>	Símbolos não-terminais. Os caracteres que não são escritos entre < e > são símbolos terminais (tokens)
... ::= ...	Regra de produção. O lado esquerdo contém um símbolo não terminal e o lado direito sua expansão, que pode conter símbolos terminais e não terminais
... | ...	Alternativa
[...]	Opcional. O(s) símbolo(s) entre colchetes podem ou não aparecer na produção
x*	Repetição. O símbolo x (terminal ou não terminal) pode ser repetido 0 ou mais vezes

Tabela A.1: Descrição da notação BNF.

A.2 Tipo de Processo

```
<WORKFLOW>     ::= WORKFLOW <IDENT> { <DEFINITION>* }
<DEFINITION>   ::= FILE <IDENT> { <FILE_DEF> } |
                   QUERY <IDENT> { <QUERY_DEF> } |
                   STRING <IDENT> { <STRING_DEF> } |
                   NUMBER <IDENT> { <NUMBER_DEF> } |
                   TASK <IDENT> :<IDENT> { <TASK_DEF> }
<FILE_DEF>     ::= NAME <STRING> ;
<QUERY_DEF>    ::= DATABASE <STRING> ; EXPRESSION <STRING> ;
<STRING_DEF>   ::= VALUE <STRING>;
<NUMBER_DEF>   ::= VALUE <INTEGER>;
<TASK_DEF>     ::= [<DEPENDS> ;] [<IN_CONTEXT> ;] [<OUT_CONTEXT> ;]
<DEPENDS>      ::= DEPENDS DEPENDENCE_RULE
<IN_CONTEXT>   ::= IN_CONTEXT <IDENT> <IDENT_LIST>*
<OUT_CONTEXT>  ::= OUT_CONTEXT <IDENT> <IDENT_LIST>*
<IDENT_LIST>   ::= , <IDENT>
<IDENT>        ::= ID
<STRING>       ::= STR
<INTEGER>      ::= INT
```

A.3 Modelo de Tarefa

```
<TASK>            ::= TASK <IDENT> { <DEFINITION> }
<DEFINITION>      ::= <TYPE> ; [<APPLICATION> ;] <PRIORITY> ;
                      [<DEADLINE> ;] <DISCONNECTED_OP> ; [<RETRIES> ;]
                      [<USERS> ;]
<TYPE>            ::= TYPE [AUTOMATIC | SEMI_AUTOMATIC | COOPERATIVE | MANUAL]
<ROLE>            ::= ROLE <IDENT>
<DESCRIPTION>     ::= DESCRIPTION <STRING>
<APPLICATION>     ::= APPLICATION <IDENT>
<PRIORITY>        ::= PRIORITY <INTEGER>
<DEADLINE>        ::= DEADLINE <INTEGER> [HOURS | DAYS]
<DISCONNECTED_OP> ::= DISCONNECTED_OPERATION <BOOLEAN>
<RETRIES>         ::= RETRIES <INTEGER>
<USERS>           ::= USERS <IDENT> <IDENT_LIST>*
<IDENT_LIST>      ::= , <IDENT>
<IDENT>           ::= ID
<INTEGER>         ::= INT
<BOOLEAN>         ::= TRUE | FALSE
```

A.4 Aplicação

```
<APPLICATION> ::= APPLICATION <IDENT> { <DEFINITION> }
<DEFINITION>  ::= <FILENAME> ; <SIZE> ; <OS> ; <CPU> ; [<INSTALLER> ;]
                  <HOSTS> ;
<FILENAME>    ::= FILENAME <STRING>
<SIZE>        ::= SIZE <INTEGER>
<OS>          ::= OS <STRING>
<CPU>         ::= CPU <STRING>
<INSTALLER>   ::= INSTALLER <STRING>
<HOSTS>       ::= HOSTS <STRING> <STRING_LIST>*
<STRING_LIST> ::= , <STRING>
<IDENT>       ::= ID
<STRING>      ::= STR
<INTEGER>     ::= INT
```

A.5 Regra de Dependência

```
<EXPRESSION>   ::= <OPERATOR> ( <EXPRESSION_1> <EXPRESSION_2>* )
<EXPRESSION_1> ::= <ITEM> | <EXPRESSION>
<EXPRESSION_2> ::= , <EXPRESSION_1>
<ITEM>         ::= <IDENT> → <STATE>
<OPERATOR>     ::= AND | OR
<STATE>        ::= READY | RUNNING | FAILED | SUCCEEDED
<IDENT>        ::= ID
```

Apêndice B

Interface dos Servidores WorkToDo

Este Apêndice contém a interface, descrita em IDL (*Interface Definition Language*), dos servidores da arquitetura WorkToDo.

```
module sgwf {
   typedef sequence<string> StringList;
   typedef sequence<octet> ByteList;

   module dMgr {
      interface DefinitionManager {
         // Operações sobre Tipos de Workflow
         boolean addWorkflowType (in string typeName, in string fileName,
                                  in string roleName);
         boolean removeWorkflowType (in string typeName);
         StringList listWorkflowTypes ();
         string getWorkflowTypeInfo (in string typeName);
         string getWorkflowTypeDefinition (in string typeName);

         // Operações sobre Modelos de Tarefas
         boolean addTaskModel (in string modelName, in string fileName);
         boolean removeTaskModel (in string modelName);
         StringList listTaskModels ();
         string getTaskModelInfo (in string modelName);
         string getTaskModelDefinition (in string modelName);

         // Operações sobre Aplicações
         boolean addApplication (in string applName, in string fileName);
         boolean removeApplication (in string applName);
         StringList listApplications ();
```

```
      string getApplicationInfo (in string applName);
      string getApplicationDefinition (in string applName);

      // Operações sobre arquivos
      ByteList readFile (in string fileName);
      boolean writeFile (in string fileName, in ByteList fileContents);
      boolean copyFile (in string source, in string destination);
   };
};


module iMgr {
   interface InstanceManager {
      // Operações sobre Instâncias de Workflow
      string createProcess (in string workflowType, in string hostName,
                            in string creator, in string userInCharge);
      boolean removeProcess (in string processName);
      StringList listProcesses ();
      StringList listProcessesByType (in string workflowType);
      StringList listTerminatedProcesses ();
      StringList listTerminatedProcessesByType (in string workflowType);
      string getHostName (in string processName);
      string getHostAddress (in string processName);
      boolean updateProcessHost (in string processName,
                            in string hostName, in string hostAddress);
      boolean processTerminated (in string processName,
                            in string processState);
   };
};


module urMgr {
   interface UserRoleManager {
      // Operações sobre Papeis
      boolean addRole (in string roleName);
      boolean removeRole (in string roleName);
      StringList listRoles ();

      // Operações sobre Usuarios
      boolean addUser (in string userName, in string roleName);
      boolean addUserToRole (in string userName, in string roleName);
```

```
      boolean removeUser (in string userName);
      boolean removeUserFromRole (in string userName,
                                  in string roleName);
      StringList listUsers ();
      StringList listUsersByRole (in string roleName);
      StringList listActiveUsers ();
      StringList listActiveUsersByRole (in string roleName);
      StringList userRoles (in string userName);
      boolean userInRole (in string userName, in string roleName);
      boolean userActive (in string userName);
      string getUserInfo (in string userName);
      boolean connectUser (in string userName, in string hostName,
                           in string hostAddress);
      boolean disconnectUser (in string userName);
      boolean addMessage (in string userName, in string sender,
                          in string message);
      boolean removeMessage (in string userName, in long messageID);
      boolean removeAllMessages (in string userName);
      StringList getMessages (in string userName);
   };
};


module pMgr {
   interface ProcessManager {
      boolean init (in string typeName);
      boolean isAlive ();
      boolean remove ();
      string retrieveState ();
      void newTaskState (in string userName, in string taskName,
                         in long newState);
      boolean selectTask (in string userName, in string taskName);
      boolean unselectTask (in string userName, in string taskName);
      boolean confirmTaskExecution (in string userName,
                                    in string taskName);
      boolean updateState (in string newState);
   };

   interface ProcessManagerFactory {
      string createProcessManager (in string processName,
                                   in string userInCharge);
```

```
      string createSubstitutiveProcessManager (in string processName,
                                               in string state);
      void deleteProcessManager (in string processName);
   };
};


module wlMgr {
   interface WorklistManager {
      boolean connect ();
      boolean disconnect ();
      boolean addWorkitem (in string processName,
                           in string serialTaskDefinition);
      boolean removeWorkitem (in long workitemID);
      StringList listWorkitems ();
      long getWorkitemID (in string processName, in string taskName);
      boolean selectWorkitem (in long workitemID);
      boolean unselectWorkitem (in long workitemID);
      boolean lockWorkitem (in long workitemID);
      boolean unlockWorkitem (in long workitemID);
      boolean executionCompleted (in long workitemID,
                                  in boolean success);
      void prefetch (in boolean enable);
      void changePolicy (in string policy);
      long timeToDisconnect ();
      void exchangeWorkitems (in long workitemID1, in long workitemID2);
   };

   interface WorklistManagerFactory {
      string createWorklistManager (in string userName);
      void deleteWorklistManager (in string userName);
   };
};


module tMgr {
   interface TaskManager {
      void initiate ();
      void execute (in string taskDef);
      boolean isAlive ();
      void remove ();
```

```
    };

    interface NTTaskManager :  TaskManager {
       void failed ();
       void done ();
    };

    interface NTTaskManagerFactory {
       NTTaskManager createNTTaskManager (in string taskName,
                                          in string processName);
       void deleteNTTaskManager (in string taskManagerName);
    };

    interface CoopTaskManager :  TaskManager {
       void abort ();
       void commit ();
    };

    interface CoopTaskManagerFactory {
       CoopTaskManager createCoopTaskManager (in string taskName,
                                              in string processName);
       void deleteCoopTaskManager (in string taskManagerName);
    };
  };


  module rMgr {
    interface RecoveryManager {
       void registerPM (in string processName, in string hostName,
                        in string hostAddress);
       boolean unregisterPM (in string processName);
       StringList listPMs ();
       boolean savePMContext (in string processName,
                              in string processState);
       void checkPMs ();
       boolean check (in string processName, in long version);
    };
  };
};
```