Um *Middleware* para Execução de Processos Estruturados em Grades Computacionais

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Fábio Rodrigo de Lima Cicerre e aprovada pela Banca Examinadora.

Campinas, 30 de junho de 2010.

Prof. Dr. Luiz Eduardo Buzato (Orientador)

Prof. Dr. Edmundo Roberto Mauro Madeira
(Coorientador)

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

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP
Bibliotecária: Maria Fabiana Bezerra Müller – CRB8 / 6162

Cicerre, Fábio Rodrigo de Lima
C485m Um middleware para execução de processos estruturados em grades computacionais/Fábio Rodrigo de Lima Cicerre -- Campinas, [S.P. : s.n.], 2010.

Orientadores : Luiz Eduardo Buzato ; Edmundo Roberto Mauro Madeira
Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1.Middleware. 2.Computação em grade (Sistemas de computador). 3.Fluxo de trabalho. 4.Sistemas operacionais distribuídos (Computadores). 5.Tolerância a falha (Computação). I. Buzato, Luiz Eduardo. II. Madeira, Edmundo Roberto Mauro. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Título em inglês: A middleware for execution of structured processes in computer grids

Palavras-chave em inglês (Keywords): 1. Middleware. 2. Grid computing. 3. Workflow. 4. Distributed operating systems (Computers). 5. Fault-tolerant computing.

Área de concentração: Sistemas Distribuídos e Redes de Computadores

Titulação: Doutor em Ciência da Computação

Banca examinadora: Prof. Dr. Luiz Eduardo Buzato (IC - UNICAMP)
Prof. Dr. Bruno Schulze (LNCC - MCT)
Prof. Dr. Delano Medeiros Beder (USP)
Prof. Dr. Jó Ueyama (IC - UNICAMP)
Profa. Dra. Maria Beatriz Felgar de Toledo (IC - UNICAMP)

Data da defesa: 07/12/2007

Programa de Pós-Graduação: Doutorado em Ciência da Computação

TERMO DE APROVAÇÃO

Tese Defendida e Aprovada em 07 de dezembro de 2007, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Bruno Richard Schulze
UFRJ.

Prof. Dr. Delano Medeiros Beder
USP.

Prof. Dr. Jó Ueyama
IC - UNICAMP.

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

Prof. Dr. Luiz Eduardo Buzato
IC - UNICAMP.

Instituto de Computação – UNICAMP
Instituto de Computação
Universidade Estadual de Campinas

Um *Middleware* para Execução de Processos Estruturados em Grades Computacionais

Fábio Rodrigo de Lima Cicerre

Dezembro de 2007

Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato (Orientador)
- Prof. Dr. Bruno Schulze
- Prof. Dr. Delano Medeiros Beder
- Prof. Dr. Jó Ueyama
- Profª. Drª. Maria Beatriz Felgar de Toledo

Resumo

O conceito de grade surgiu com a necessidade crescente de se aproveitar recursos computacionais disponíveis em uma ou mais organizações para resolver problemas que exigem compartilhamento de dados e um grande poder de processamento. Uma grade computacional tem como objetivo principal permitir a execução distribuída e paralela de tarefas em recursos compartilhados. Uma grade é constituída de infra-estrutura física, composta de uma ou mais redes autônomas de computadores, e de um sistema de suporte (*middleware*), que provê serviços de gerenciamento de informações sobre os recursos da grade, controle de acesso e execução de tarefas sobre esse recursos e mecanismos de comunicação.

Atualmente existem diversos sistemas que suportam a execução de tarefas independentes em uma grade computacional, mas poucos consideram a execução de processos de *workflow*, que permitem a definição de dependência explícita de dados e controle entre tarefas, o que impede um melhor aproveitamento de recursos, escalabilidade, desempenho de execução e recuperação automática de processos com manutenção de consistência.

O sistema Xavantes, proposto e descrito nesse trabalho, procura suprir essas deficiências, tendo como principal objetivo suportar a execução distribuída de processos de *workflow* em máquinas heterogêneas, em uma ou mais organizações autônomas e dinâmicas, provendo um *middleware* que forneça uma melhor escalabilidade, desempenho e confiabilidade para a execução de aplicações em grades computacionais.

Abstract

The grid concept has emerged from the increasing necessity of using available computational resources in one or more organizations in order to solve problems that require data sharing and large processing power. The main goal of a computational grid is to allow the distributed and parallel execution of tasks in shared resources. A grid is composed of a physical infra-structure, with one or more autonomous networks of computers, and a middleware, which provides services of information management about the grid resources, access control and tasks execution in these resources and communication mechanisms.

Nowadays, there are some systems that support the execution of independent tasks in a computational grid, but only ones consider the execution of workflow processes, which allow the explicit definition of data and control dependencies among tasks, and this restricts a better use of available resources, scalability, execution performance, and automatic recovery of processes with the correct consistency maintenance.

The Xavantes system, proposed and described in this work, is designed to reduce these deficiencies, having as its main goal the supports to the distributed execution of workflows in heterogeneous resources of one or more autonomous and dynamic organizations, providing a middleware that delivers a better scalability, performance and reliability to the application execution in grid computing.

Agradecimentos

Agradeço a todos que me ajudaram de alguma forma nessa árdua tarefa de pesquisar um tema original para a ciência dos homens e defender uma tese sobre uma matéria importante e prática para esse tema, avançando o conhecimento na área relacionada. Especialmente, agradeço aos meus orientadores, Luiz Eduardo Buzato e Edmundo Roberto Mauro Madeira, por indicarem os caminhos promissores de pesquisa e colaborarem nas soluções propostas, ora criticando os erros, ora oferecendo idéias e modificações, sempre muito valiosas. Mais do que isso, agradeço a esses dois grandes homens pelo seu caracter humano, por trilharem sempre ao meu lado, com muita dedicação e paciência. Agradeço aos meus pais, Miguel Roberto Cicerre e Isabel de Lima Cicerre, à minha irmã, Déborah Maria de Lima Cicerre, e à minha namorada, Elisângela Iódice, por acreditarem em mim e me apoiarem nos momentos mais difíceis desta jornada. Agradeço ao meus colegas de trabalho, da Secretaria da Fazenda do Estado de São Paulo, por me incentivarem e ajudarem a terminar esse trabalho, em especial ao Paulo César de Oliveira, que revisou artigos escritos em língua inglesa, e aos Delegados Regionais Tributários de Campinas, antigo e atual, Glauco Honório e José Eduardo de Paula Saran, por me darem especial atenção nos momentos em que mais precisei me dedicar às pesquisas e à redação da tese. Agradeço também ao Estado de São Paulo e ao seu virtuoso povo por financiarem a minha educação, desde o ensino fundamental até o final deste doutorado. Para estes retribuirei, nos papéis de cidadão e de funcionário público, com todos os meus esforços, conhecimento e dedicação. Agradeço por último, mas com a mais elevada atenção, ao bom Deus que há de existir e que indiretamente, ou diretamente, permitiu que eu tivesse forças para dar mais esse importante passo em direção ao bom e ao belo.

Sumário

Resumo **vii**

Abstract **ix**

Agradecimentos **xi**

1 Introdução **1**

2 Conceitos Básicos **7**
2.1 *Middlewares* . . . 9
2.2 Grades Computacionais . . . 12
2.2.1 Grades de Dados . . . 13
2.2.2 Grades de Processamento . . . 14
2.3 Sistemas de Gerenciamento de *Workflow* . . . 17
2.3.1 Modelo de Especificação de Workflows . . . 18
2.3.2 Arquitetura de Sistemas de Gerenciamento de Workflow . . . 20
2.4 Sistemas de Suporte a *Workflows* em Grades Computacionais . . . 23
2.4.1 Modelo de Programação de Processos . . . 23
2.4.2 Arquitetura de Sistemas de Suporte a Processos . . . 25
2.4.3 Tolerância a Falhas . . . 29
2.5 Resumo . . . 37

3 Trabalhos Relacionados **39**
3.1 Grade Computacional . . . 39
3.1.1 *Middlewares* Precursores . . . 39
3.1.2 *Middlewares* Tradicionais . . . 40
3.1.3 *Middlewares* Oportunistas . . . 41
3.2 Suporte a Processos em Grades Computacionais . . . 42
3.2.1 Controle de Execução no Cliente . . . 42
3.2.2 Suporte a *Workflow* em Grades de Dados . . . 43

3.2.3 Coordenação Centralizada de Processos . . . 43
3.2.4 Suporte a Infra-estruturas *Peer-to-Peer* . . . 44
3.2.5 Recuperação de Processos . . . 45
3.2.6 Escalonamento Otimizado e Distribuído . . . 46
3.2.7 Escalonamento e Execução em *Pipeline* . . . 46
3.3 Resumo . . . 47

4 Especificação de Processos Estruturados 51
4.1 Descrição da Linguagem de Especificação . . . 51
4.1.1 Processos . . . 53
4.1.2 Atividades . . . 53
4.1.3 Controladores . . . 53
4.2 Características da Linguagem de Especificação . . . 55
4.3 Gramática da Linguagem . . . 62
4.3.1 Aplicação . . . 62
4.3.2 Processo . . . 62
4.3.3 Atividade . . . 65
4.3.4 Elemento de Processo . . . 66
4.3.5 Controladores . . . 67
4.3.6 Invocação de Processo e Atividade . . . 78
4.3.7 Código . . . 80
4.3.8 Invocação de Serviço e de Objeto Remoto . . . 81
4.3.9 Requisitos de Execução . . . 82
4.3.10 Estimativas de Execução . . . 85
4.3.11 Atributos para Tolerância a Falhas . . . 87
4.4 Exemplos de Processos . . . 89
4.4.1 Busca de Números Primos . . . 89
4.4.2 Cruzamento de Transações de Venda entre Contribuintes . . . 92
4.5 Resumo . . . 96

5 *Middleware* para a Execução de Processos Estruturados 99
5.1 Arquitetura do *Middleware* . . . 100
5.1.1 Gerenciador de Grupo . . . 101
5.1.2 Gerenciador de Nó . . . 118
5.1.3 Gerenciador de Cliente . . . 131
5.2 Relacionamento entre Grupos . . . 134
5.3 Modelo Dinâmico . . . 135
5.3.1 Autenticação de Usuários e Servidores . . . 135
5.3.2 Gerenciamento de Informações sobre Recursos da Grade . . . 137

5.3.3 Implantação de Aplicações . . . 139
5.3.4 Escalonamento e Execução de Processos . . . 139
5.3.5 Execução de Atividades . . . 142
5.4 Resumo . . . 143

6 Execução de Processos e Tolerância a Falhas 147
6.1 Execução de Processos . . . 147
6.1.1 Especificação e Implantação de Processos e Atividades . . . 150
6.1.2 Invocação e Instanciação de Processos e Controladores . . . 151
6.1.3 Invocação e Instanciação de Atividades . . . 152
6.1.4 Escalonamento de Elementos Filhos em Processos e Controladores . 152
6.1.5 Coordenação de Processos e Controladores . . . 154
6.2 Escalonamento . . . 154
6.2.1 Algoritmos de Escalonamento . . . 156
6.2.2 Seleção de Recursos Computacionais . . . 163
6.3 Tolerância a Falhas . . . 166
6.3.1 *Middleware* . . . 167
6.3.2 Atividades . . . 175
6.3.3 Processos . . . 177
6.4 Resumo . . . 179

7 Implementação e Resultados Experimentais 183
7.1 Implementação . . . 183
7.2 Resultados Experimentais . . . 185
7.3 Resumo . . . 187

8 Conclusão 191

Bibliografia 195

Lista de Tabelas

1 Introdução **1**

2 Conceitos Básicos **7**

3 Trabalhos Relacionados **39**

4 Especificação de Processos Estruturados **51**
4.1 Classificação dos controladores em função do tipo do fluxo de controle e do paralelismo permitido. 54

5 *Middleware* para a Execução de Processos Estruturados **99**

6 Execução de Processos e Tolerância a Falhas **147**
6.1 Tabela CLIENT_REGISTRY 168
6.2 Tabela NODE_SERVICE_REGISTRY 169
6.3 Tabela GROUP_SERVICE_REGISTRY 170
6.4 Tabela ACTIVITY_INFORMATION 172
6.5 Tabela PROCESS_INFORMATION 173

7 Implementação e Resultados Experimentais **183**

8 Conclusão **191**

Lista de Figuras

1 Introdução **1**

2 Conceitos Básicos **7**
2.1 Exemplo de um Processo *Workflow* de Compra de Livros. 19

3 Trabalhos Relacionados **39**

4 Especificação de Processos Estruturados **51**
4.1 Diagrama de Classes do Modelo de Programação. 52

5 *Middleware* para a Execução de Processos Estruturados **99**
5.1 Arquitetura do *Middleware* Xavantes. 100
5.2 Estrutura do Gerenciador de Grupo (GM). 103
5.3 Estrutura do Gerenciador de Nó (NM). 119
5.4 Estrutura do Gerenciador de Cliente (CM). 132
5.5 Relacionamento entre Gerenciadores de Grupo. 134

6 Execução de Processos e Tolerância a Falhas **147**
6.1 Execução de um Processo no *Middleware* Xavantes. 148

7 Implementação e Resultados Experimentais **183**
7.1 Comparação de tempo. 186
7.2 Comparação de desempenho. 187
7.3 Uso do poder de processamento disponível. 188

8 Conclusão **191**

Capítulo 1

Introdução

Com o advento da Internet, um número cada vez maior de serviços e de dados se tornou disponível, assim, para o bem da humanidade, a informação ganhou uma grande importância em nossos dias. Atividades que eram relevantes para a sociedade ou forneciam uma vantagem competitiva, mas eram impossíveis de serem realizadas ou muito custosas em tempo ou dinheiro, hoje, com a Internet, se tornaram possíveis. Agora é simples comprar bens e produtos de qualquer lugar do mundo com segurança e rapidez, consultar e movimentar contas bancárias, declarar e pagar impostos, e ainda buscar e acessar notícias, artigos, livros, músicas, filmes, aplicativos e outros conteúdos digitalizados imediatamente, pagando ou não por esses serviços.

Entretanto, mesmo com todas essas vantagens, a sociedade está muito distante de conseguir usufruir de todos os benefícios que podem ser providos por essa enorme rede de milhões de computadores e de fontes de dados. Em um computador pessoal, um usuário consegue acessar arquivos de dados e serviços básicos, mas além disso, consegue administrar o seu próprio computador e executar qualquer aplicativo, interativo ou não, que ele obtenha ou construa para satisfazer suas necessidades específicas e imediatas. Em contraposição, em uma rede, seja ela local ou global, um usuário ou organização raramente consegue utilizar os recursos e serviços de que ele necessita de forma integrada e plena e, mesmo nesses casos, raramente ele, ou a organização, tem meios de construir aplicativos que utilizem os recursos distribuídos e compartilhados de maneira simples, distribuindo, compondo e coordenando os serviços existentes e pequenos utilitários criados por ele ou por um grupo de trabalho. É exatamente nesse nicho que os conceitos de sistemas distribuídos do tipo grade e do tipo *peer-to-peer* se estabelecem, com o objetivo de permitir a completa utilização de recursos heterogêneos e distribuídos, compartilhados por uma ou mais organizações autônomas, ou grupos de usuários, de forma homogênea e simples.

Uma grade computacional é formada de recursos computacionais e de dados compartilhados de forma transparente entre uma ou mais organizações e tem como intuito permitir

a realização de computações crescentemente complexas, não viáveis de serem realizadas em um único supercomputador tradicional ou aglomerado, pela dificuldade de acesso, custo ou tempo.

Um sistema de suporte a grade, ou *middleware* de grade [FKT01], tem por objetivo fornecer à grade computacional serviços para facilitar o gerenciamento e a utilização de seus recursos computacionais e de dados. Dessa forma, ele permite a integração de dados e serviços largamente dispersos e fornece poder computacional não disponível em um único computador ou aglomerado isolado, o que possibilita a resolução de problemas computacionalmente complexos, que necessitam de acesso a dados ou instrumentos distribuídos, em um tempo factível, impossíveis de serem resolvidos de outro modo. O uso desses sistemas para grades computacionais auxilia na resolução de problemas científicos complexos, como o descobrimento da cura de doenças, o mapeamento de genomas, e a simulação mais precisa de experimentos físicos. Complementarmente, eles provêem o aumento do poder computacional de empresas, instituições e indivíduos para realizar, de forma mais eficiente, por exemplo, mineração de dados e conhecimento, simulação de projetos e produtos de engenharia e arquitetura, e também para produzir filmes e criar mundos virtuais mais realistas com uso massivo de usuários [GRI07a, Ros05].

Atualmente existem diversos *middlewares* para grades computacionais: Globus [Fos05], Legion [GW96] e Condor-G [FTL+01]. Esses *middlewares* fornecem serviços de registro e monitoramento de recursos disponíveis, compartilhamento de recursos computacionais e de dados, segurança no acesso aos recursos e suporte a execução distribuída de tarefas.

Embora os *middlewares* tradicionais para grades computacionais facilitem o gerenciamento de recursos distribuídos e o acesso a esses recursos para execução de tarefas isoladas, eles não fornecem suporte direto a especificação, execução e monitoramento de processos de longa duração, que compõem e coordenam a invocação e a troca de dados entre tarefas interdependentes. Dessa forma, eles deixam a cargo do usuário a programação da coordenação das tarefas, dificultando o desenvolvimento de aplicações distribuídas e de longa duração. Ademais, neles não há nem coordenação automática, nem monitoramento e recuperação transparente de processos. No aspecto de tolerância a falhas, não são permitidos a descrição e o tratamento específico de exceções em etapas determinadas do processo. Por não considerar as dependências entre as atividades, eles também não permitem a utilização de algoritmos eficientes de escalonamento e alocação de atividades, que visam a minimizar o custo de troca de dados, reduzir o tempo de execução da aplicação e otimizar a utilização compartilhada de recursos.

Para suprir algumas dessas deficiências, recentemente foram desenvolvidos alguns *middlewares* para execução de processos sobre grades computacionais. Nesses *middlewares*, os processos são geralmente especificados por interconexões de tarefas na forma de um grafo dirigido e acíclico (DAG), no qual os vértices designam as tarefas e as arestas, os

fluxos de controle e de dados. Exemplos desses *middlewares* são: WebFlow [AFFH98], que dispõe de uma arquitetura para composição e coordenação de processos distribuídos; Opera-G [Bau04], que permite recuperação e manipulação dinâmica de processos; Swin-Dew [YYR03], que executa processos DAG de forma distribuída em uma infra-estrutura *peer-to-peer*; e GridFlow [CJSN03], que contém um algoritmo de escalonamento que utiliza as informações de dependências entre tarefas para minimizar o tempo de execução de processos. Entretanto, os *middlewares* existentes contêm ainda muitas limitações, tal como serviços de escalonamento totalmente centralizado ou totalmente distribuído, não havendo um equilíbrio entre esses dois extremos. Isso implica facilidade de monitoramento e controle de processos, no caso de serviço centralizado, ou eficiência na execução de processos, no caso distribuído, mas nunca as duas vantagens. Outros problemas encontrados são: linguagens de especificação não muito simples, nem expressivas; construtores da linguagem que não explicitam as características dos processos para haver uma otimização de uso dos recursos da grade, tal como localização de variáveis compartilhadas e criação dinâmica de tarefas; algoritmos de escalonamento que são muito custosos e que deixam de considerar os melhores recursos no ambiente dinâmico da grade; tolerância a falhas inadequada para a execução de processos de longa duração; e *middlewares* sem adaptação necessária para se ajustar, juntamente com as aplicações em execução, às condições extremamente dinâmicas de uma grade. Todos esses problemas restringem o completo uso da grade de forma fácil e transparente e impedem o desenvolvimento de aplicações consistentes e que necessitam de execução eficiente num ambiente escalável, disponível e confiável.

Esse trabalho defende a tese de que, para se utilizar efetivamente os recursos computacionais disponíveis na Internet, é necessário o desenvolvimento de um *middleware* que permita a especificação e monitoramento de processos de longa duração em grades computacionais com um modelo de programação simples e expressivo e que utilize os recursos disponíveis de forma homogênea, eficiente e escalável para execuções tolerantes a falhas, consistentes e adaptáveis desses processos. Uma forma de atingir esse intento, conforme demonstraremos, é a utilização de uma linguagem estruturada e expressiva de especificação de processos e de um *middleware* para execução de processos que distribui hierarquicamente o controle dos processos, com mecanismos de escalonamento contextualizado, adaptável e tolerante a falhas.

Para alcançar o objetivo acima exposto, a contribuição desse trabalho pode ser resumida em:

- projeto de uma linguagem de especificação estruturada de processos, baseada na composição de controladores seqüenciais e paralelos que permitem a fácil modelagem de aplicações distribuídas em grade;

- utilização de variáveis compartilhadas próximas à computação, para a troca de dados de forma eficiente e confiável entre elementos de processo e otimização dos recursos computacionais;

- projeto de um *middleware* para execução de processos que utiliza a composição estruturada dos controladores para distribuir hierarquicamente a execução de aplicações (processos *workflow*) sobre a grade computacional, de forma a garantir a escalabilidade do *middleware* e a adaptabilidade e o desempenho das aplicações;

- proposta de um modelo de tolerância a falhas que permite a recuperação automática e distribuída do *middleware* e dos processos (aplicações), o tratamento de exceções da aplicação e a manutenção de consistência dos dados do processo e compartilhados;

- proposta de algoritmos dinâmicos de escalonamento que permitem a adaptação dinâmica de aplicações para a utilização otimizada de recursos disponíveis e para o reescalonamento de partes de processos falhos;

- resultados experimentais que comprovam a eficácia da linguagem e a eficiência do *middleware* proposto na execução adaptável de processos e na recuperação de falhas sobre grades computacionais dinâmicas.

Para realizar essas contribuições práticas, importantes e originais na área de computação em grade, o sistema Xavantes [CMB04] foi desenvolvido. Ele consiste de uma linguagem de especificação de processos estruturados, que permite a composição hierárquica de construtores de fluxo de controle, e de um *middleware* para a execução de processos estruturados de forma totalmente distribuída e hierárquica, que pode envolver vários domínios autônomos de recursos. A linguagem de programação estruturada permite criar facilmente fluxos de controle complexos e paralelos em processos, combinando e aninhando os construtores de fluxos disponíveis, de forma similar a linguagens estruturadas de programação. O *middleware*, por sua vez, aproveita a composição dos construtores de fluxo de controle para hierarquicamente escaloná-los, distribuí-los e executá-los pela grade, de forma que atividades e controladores fortemente dependentes fiquem em nós próximos, quando possível. Dessa forma, devido à distribuição controlada, ele provê melhor desempenho de execução, tolerância a falhas e escalonamento adaptável e minimiza o custo de comunicação entre atividades. Como resultado o sistema suporta a execução distribuída da aplicação em grades, mas de forma organizada e controlada, com um alto nível de confiabilidade, escalabilidade e desempenho.

Implementamos um protótipo do Xavantes e realizamos experimentos que evidenciam a validade das decisões do projeto, que avançam no caminho para se obter um *middleware* de grade mais transparente, flexível, confiável, escalável e eficiente, que torne, enfim, a grade

disponível à sociedade para responder às necessidades crescentes de aplicações complexas e eficientes, com acesso a recursos distribuídos. Demonstraremos resultados que comprovam que esse sistema permite: a adaptação própria e das aplicações às condições dinâmicas da grade [CMB06]; que o sub-sistema de escalonamento distribuído permite um maior desempenho de execução de processos e utilização de recursos; e que falhas e exceções podem ser tratadas transparentemente e de forma flexível, de modo a tornar a execução de processos mais eficiente, confiável e consistente.

O Capítulo 2 dessa tese descreve os conceitos básicos relacionados às áreas de grades computacionais e de sistemas de gerenciamento de processos. O Capítulo 3 apresenta os trabalhos relacionados. Ele contém uma avaliação de sistemas desenvolvidos que procuram resolver problemas similares aos abordados por essa tese. O Capítulo 4 descreve a linguagem de especificação estruturada de processos e o Capítulo 5, a arquitetura do protótipo do *middleware* Xavantes. O Capítulo 6 detalha o serviço de execução de processos, descrevendo algoritmos de escalonamentos desenvolvidos para o modelo estruturado de programação, a adaptação dinâmica do escalonamento e distribuição de elementos do processo, além dos mecanismos de tolerância a falhas do modelo, com recuperação automática dos serviços do *middleware*, e dos processos, atividades e dados das aplicações. O Capítulo 7 descreve a implementação do sistema Xavantes e os resultados que comprovam as asserções supracitadas sobre esse modelo por meio dos experimentos que foram realizados. A conclusão da tese está no Capítulo 8, que resume as contribuições dessa tese e procura indicar desdobramentos futuros para a pesquisa na área de ambientes de programação de processos para grades computacionais.

Capítulo 2

Conceitos Básicos

Com o aumento da demanda por serviços mais eficientes e abrangentes pela sociedade, universidades, governos e empresas, que exigem alto nível de desempenho dos recursos distribuídos e de comunicação, por um lado, e o aumento da capacidade computacional e redes cada vez mais rápidas, por outro lado, tornou-se clara a necessidade do desenvolvimento de *software* intermediário (também conhecido como *middleware*) que permita o uso do potencial desses recursos disponíveis pela Internet de forma simples, transparente e eficaz pelas aplicações. Para suprir essa lacuna surgiram os *middlewares* para grades computacionais, conforme descreveremos nessa seção.

A Internet trouxe a possibilidade de se acessar remotamente dados e serviços existentes de qualquer lugar do mundo, conectando milhões de computadores distribuídos entre governos, empresas e residências. Logo se notou a possibilidade de compartilhar esses mesmos recursos entre instituições a fim de se realizar não somente o acesso aos dados contidos nesses, mas aos próprios recursos computacionais para executar serviços próprios e específicos, com a finalidade de aumentar o desempenho e a confiabilidade das aplicações desenvolvidas. Surgiu dessa idéia o conceito de grade computacional, na qual o acesso completo a recursos heterogêneos compartilhados entre várias instituições autônomas deve ser idealmente tão transparente e simples quanto a utilização da rede elétrica (*grid* em inglês) para conectar aparelhos eletrodomésticos. O *middleware* para grade computacional é o software intermediário, que se estabelece entre as aplicações e os recursos compartilhados, que provê serviços para tornar o acesso transparente e simples aos recursos compartilhados.

Até recentemente o acesso a dados pela Internet era realizado quase exclusivamente por meio de páginas HTML e serviços de obtenção (*download*) de arquivos hospedados em poucas instituições, restringindo, assim, o potencial e o desempenho da troca de informações entre os computadores ligados a ela. Visando a superar essa limitação, deixando os dados mais disponíveis e o acesso a eles independente de serviços providos por essas insti-

tuições e empresas, foram criadas as infra-estruturas de troca de dados *peer-to-peer*, tais como Napster [CG01] e Gnutella [Bab07], que permitem o compartilhamento e a troca de arquivos entre quaisquer usuários, domésticos ou profissionais. Seguindo paralelamente o desenvolvimento dos *middlewares* para grades computacionais, estão surgindo sistemas que permitem, além do compartilhamento de arquivos, o compartilhamento do poder de processamento de computadores de usuários domésticos utilizando essa topologia *peer-to-peer*. Exemplos de tais sistemas são o BOINC (*Berkeley Open Infrastructure for Network Computing*) [And04] e o OurGrid [CBA+06].

As diferenças entre os *middlewares* para grades computacionais e os sistemas *peer-to-peer* estão se tornando cada vez mais tênues, podendo, num futuro próximo, tornarem-se nulas com a origem de sistemas mais completos de compartilhamento de recursos, que envolverão tanto máquinas especializadas de organizações, quanto máquinas de mesa em empresas ou nas residências [FI03]. Atualmente, os *middlewares* para grades computacionais envolvem recursos com alto poder de processamento e grandes repositórios de dados de poucas organizações e fornecem serviços de autenticação, acesso e gerenciamento de recursos autônomos. Comparativamente, os sistemas *peer-to-peer* permitem um maior acesso a recursos distribuídos, incluindo máquinas residenciais, e lidam melhor com controle de disponibilidade de recursos e tolerância a falhas.

Entretanto, mesmo havendo uma certa variedade de *middlewares* hábeis de executar várias tarefas de forma distribuída e paralela em máquinas disponíveis e compartilhadas, tanto em grades computacionais, como em redes *peer-to-peer*, eles não permitem a especificação de aplicações complexas e de longa duração, que necessitam executar de forma consistente e confiável. A vasta maioria dos *middlewares* executam grupos de tarefas independentes em máquinas distribuídas, mas não coordenam automaticamente a execução dessas de acordo com um plano definido, tampouco fornecem recuperação automática e modificação dinâmica da ordem de execução entre elas. Ademais, por falta de conhecimento da dependência de dados e de execução entre essas tarefas, não fazem um melhor uso de recursos disponíveis e perdem em eficiência por não considerar essas informações em seu algoritmo de escalonamento de tarefas e alocação de recursos. A manutenção da consistência dos dados é outro problema, uma vez que esses *middlewares* também não fornecem propriedades transacionais para controlar a concorrência de acesso a dados, a atomicidade das modificações e o desfazimento de operações mal-sucedidas.

Pelo aspecto da modelagem, com a necessidade de qualquer usuário especificar facilmente aplicações de longa duração, compondo e coordenando tarefas e serviços, pré-existentes ou criados especificamente para o problema, surgiram os paradigmas de meta-programação e de processos de alto-nível do tipo *workflow*. Ambos utilizam o conceito de processos para compor e coordenar tarefas, o que permite especificar o fluxo de dados e a ordem de execução entre essas tarefas. Por ser um modelo de programação de alto

nível, em sua maior parte declarativo, mesmo usuários com pouco conhecimento de programação podem especificar e modificar aplicações e monitorar e analisar sua execução em tempo real [Hol94, Coa05]. Existem vários sistemas desenvolvidos pelo mercado e pelas universidades que suportam a execução de processos de *workflow*, mas poucos têm a máquina de execução distribuída, comportam mais de uma organização ou servem para descrever aplicações computacionais, pela própria natureza dos processos [GHS95, H+04]. Sendo assim, eles não são adequados para descobrir e utilizar máquinas disponíveis por várias organizações e nem têm como objetivo maximizar a utilização de recursos e otimizar o escalonamento de tarefas para executar processos que necessitam de alto poder de processamento.

Somente há poucos anos o conceito de *workflow* começou a ser empregado para uso em grades computacionais de forma a permitir a construção de aplicações de alto-desempenho para serem executadas de forma distribuída sobre estas. Os *middlewares* existentes inicialmente suportavam somente a especificação de processos simples e o escalonamento e execução centralizada desses processos para a composição de serviços existentes [AFFH98]. Eles têm como objetivo selecionar os melhores recursos para executar os processos o mais eficientemente possível, de forma a terminá-los no menor tempo. Mais recentemente, os *middlewares* começaram a tolerar falhas de processos [Bau04] e a permitir a execução distribuída de tarefas [CJSN03], além de possuírem melhores algoritmos de escalonamento [YYR03]. Entretanto, eles não permitem a utilização efetiva dos recursos da grade, contendo várias organizações; não são escaláveis, por não possuírem mecanismos de distribuição de tarefas eficiente; não são adaptáveis, com escalonamento dinâmico, em caso de falhas ou disponibilização de novos recursos; possuem tolerância a falhas limitada, com pontos únicos de falha no sistema e ausência de mecanismos de manutenção de consistência de dados e de processo em caso de falhas; não possuem mecanismos de escalonamento com alocação de recursos e distribuição de tarefas de forma antecipada; e nem possuem linguagem expressiva que permite especificação e monitoramento de processos complexos.

Nesse capítulo descreveremos primeiramente os conceitos básicos de *middlewares*, grades computacionais e sistemas de gerenciamento de *workflows*. Depois, apresentaremos os fundamentos dos *middlewares* com suporte a execução de *workflows* em grades computacionais.

2.1 *Middlewares*

Um *middleware* é uma camada de *software* de conectividade que se aloja entre o sistema operacional e as aplicações de cada nó de um sistema distribuído. Ele contém um conjunto de serviços com a finalidade de permitir que componentes de aplicação situados em um ou mais computadores distribuídos e heterogêneos interajam [Ber96].

Os *middlewares* podem ser classificados quanto ao seu tipo de suporte a invocação e distribuição de componentes de aplicação em *middleware* para:

Acesso a dados por SQL – se estabelece entre aplicações e gerenciadores de bancos de dados relacionais e permite o armazenamento e a recuperação consistentes de dados em diversos tipos de bases de dados por meio de comandos SQL. Como exemplo, o JDBC *Java Database Connectivity* [And06] permite que aplicações escritas em Java invoquem bases de dados relacionais por meio de comandos SQL padrão.

Chamada Remota a Procedimentos (*Remote Procedure Call*) – um dos primeiros *middlewares* para comunicação entre processos remotos. Ele promove a comunicação entre processos distribuídos em uma ou mais máquinas por meio de invocações remotas a procedimentos externalizados. Ele provê transparência de acesso, em que o processo cliente invoca um procedimento no processo servidor como se este procedimento fosse local. Para prover esse mecanismos, ele utiliza um *proxy* no cliente para serializar os parâmetros da invocação do procedimento local e um *stub* no servidor, onde o procedimento será invocado, para desserializar esses parâmetros e invocar o procedimento. O *middleware* baseado em RPC mais utilizado é o Sun ONC RPC [Sri95].

Troca de Mensagens (*Message Oriented Middleware*) – utilizado para a comunicação em sistemas distribuídos altamente paralelos. A comunicação nesse *middleware* é realizada por meio de trocas de mensagens entre máquinas da rede. No modelo MPI (*Message Passing Interface*) [DW95], cada processo pode ser emissor e receptor de mensagens, sendo que o emissor utiliza primitivas de envio de mensagens para um receptor, com tipo de dados e marcador, em canais especificados, e o receptor utiliza primitivas de recebimento de mensagens advindas de um determinado canal e emissor, enviadas para este receptor. O modo de envio e de recebimento de mensagens podem ser síncrono ou assíncrono. Há também primitivas para comunicação em grupo. Complementarmente, existem modelos de troca de mensagens persistentes, por exemplo o MSMQ (*Microsoft Message Queuing*) [Cor07b], em que os processos emissores e receptores não precisam estar ativos simultaneamente, sendo as mensagens armazenadas nos canais persistentes para posterior recuperação na seqüência pelos receptores.

Invocação a Objetos – com a popularização do paradigma de orientação a objetos, surgiram na década de 1990 *middlewares* que permitem criar, buscar e invocar objetos remotos, que são instanciados a partir de um tipo determinado, possuem estado interno, implementam determinadas interfaces e são unicamente identificados. Podem existir vários objetos remotos de um determinado tipo em uma máquina, cada um

com seu identificador e estado. Exemplos dessa classe de *middlewares* são CORBA [Obj08] e COM [Cor08a]. Estendendo esse conceito, atualmente os *middlewares* suportam o paradigma de componentes de software, que contêm estrutura interna mais complexa e operações especiais. Esses *middlewares* fornecem atributos transacionais, persistência e segurança aos componentes de forma facilitada. Exemplos são servidores de aplicações EJB [Gro06] e .NET *Framework* [Cor08b].

Invocação a Serviços *Web* – recentemente, com a proliferação de diferentes *middlewares* que não se comunicam entre si e devido à necessidade da construção de aplicações interoperáveis em nível global, surgiu uma arquitetura padrão de *middleware* baseada em padrões da Internet. Nessa arquitetura, também conhecida como arquitetura baseada em serviços (*Service Oriented Architecture – SOA*), as aplicações são implementadas utilizando-se serviços disponíveis ou construídos especificamente para cada aplicação, instanciados em sítios da rede global. O transporte da comunicação entre os serviços e os clientes (que podem ser outros serviços) se baseia comumentemente no protocolo HTTP (*Hypertext Transfer Protocol*), largamente utilizado para a obtenção de documentos na Internet. As invocações são realizadas por meio de mensagens em XML (*Extensible Markup Language*) [BP04], estruturadas conforme o padrão SOAP (*Simple Object Access Protocol*) [Con03]. Os serviços podem implementar uma ou mais portas, conhecidas como interfaces, contendo uma ou mais operações cada. Os serviços são descritos pelo padrão WSDL (*Web Service Description Language*) [Con01]. Esses serviços são referenciados diretamente pelo cliente, por meio de URLs (*Uniform Resource Locators*), ou descobertos por meio do serviço de registro e busca UDDI (*Universal Description, Discovery and Integration*). Adicionalmente, há padrões que permitem que a invocação desses serviços seja transacional e segura.

Grade Computacional – os *middlewares* para grade computacional são os mais abrangentes entre os apresentados para a construção de aplicações distribuídas. Eles têm por finalidade fornecer serviços de compartilhamento de recursos computacionais e de dados entre as diversas aplicações. Em detalhes, esses *middlewares* gerenciam as meta-informações de propriedades e estado dos recursos e grupos; suportam a distribuição e a comunicação entre os componentes instanciados em diversos computadores da rede; e controlam o acesso a esses recursos [FKT01]. Portanto, eles comportam a comunicação entre os componentes, como os demais *middlewares*, mas também gerenciam as informações sobre o estado dos recursos computacionais e controlam a distribuição dos componentes entre os recursos. Adicionalmente, há *middlewares* que controlam a execução distribuída de aplicações.

O presente trabalho foca na especificação de um *middleware* para a execução distribuída de aplicações, descritas como processos de longa-duração, em grades computacionais. As grades computacionais são adequadas para promover o que recentemente a mídia está denominando computação em nuvens (*cloud computing*) [Bak07], a utilização de diversos computadores distribuídos para a realização de computação em larga escala. Na próxima seção descreveremos mais detalhadamente os tipos de grades computacionais existentes, suas características e os serviços do *middleware* que permitem a execução de aplicações sobre essas grades.

2.2 Grades Computacionais

Segundo Foster [Fos02], uma grade computacional é uma infra-estrutura de *hardware* composta de recursos computacionais dispersos em uma ou mais organizações e, sobre esta, camadas de *software* para controlar o compartilhamento desses recursos entre aplicações de diferentes usuários com a finalidade de resolver problemas específicos. No mínimo, para uma infra-estrutura ser considerada uma grade ela deve possibilitar o compartilhamento e acesso a recursos de forma não centralizada, entre várias organizações; fornecer serviços de valor agregado, com controle de qualidade desse serviço, tal como desempenho, armazenamento e largura de banda de comunicação; e ser baseada em padrões abertos. Comparativamente, um sistema *peer-to-peer* [FI03] é geralmente baseado em padrões fechados e oferece serviços mais específicos, entretanto atualmente provê uma melhor tolerância a falhas e escalabilidade, lidando com a participação de um maior número de recursos e usuários, em ambientes mais heterogêneos, dinâmicos e menos confiáveis.

Na definição de grade, Foster [FKT01] afirma que devem existir protocolos e serviços comuns, principalmente, e interfaces de programação e ambientes de desenvolvimento, complementarmente, que permitam o compartilhamento de recursos computacionais e de dados entre organizações. Ele divide a arquitetura de uma grade em camadas: *componentes físicos*; *serviços de comunicação*, que incluem o transporte seguro de mensagens, e as funcionalidades de autenticação e autorização; *serviços de acesso a recursos*, que incluem acesso, monitoramento e gerenciamento tanto de recursos de dados quanto computacionais, provendo escalonamento local de tarefas; e *serviços coletivos*, que fornecem uma visão global dos recursos, com serviços de diretório e escalonamento global de tarefas. Resumindo, uma infra-estrutura para grade deve fornecer, nos níveis mais básicos, protocolos e serviços de comunicação, segurança de mensagens, autenticação e autorização de usuários, mecanismos de acesso aos recursos, registro e monitoramento do estado dos recursos, escalonamento de tarefas, e acesso a dados distribuídos. Serviços mais avançados, que estão começando a surgir, envolvem a utilização coletiva de recursos, e um exemplo é o serviço de suporte a execução de processos workflow em máquinas distribuídas.

Com o objetivo de definir um padrão para a especificação e a integração de infra-estruturas de grades, surgiu a Open Grid Service Architecture (OGSA) [FKNT02, Fea05]. A arquitetura proposta é baseada em Web Services, um padrão recente para disponibilização de serviços por meio da Internet que utiliza: o padrão *Web Service Description Language* (WSDL [Con01]) para descrever serviços; e a passagem de mensagens XML, geralmente sobre o protocolo HTTP, formatados segundo o padrão *Simple Object Access Protocol* (SOAP [Con03]), para a sua invocação. A arquitetura utiliza o padrão *Web Service Resource Framework* (WSRF [OAS06]), que estende os Web Services com a descrição de recursos, que são instâncias de objetos mantidos pelos serviços, permitindo a identificação e manutenção do estado desses objetos pelo servidor. Para receber notificações sobre o estado dos recursos, a OGSA utiliza o padrão *Web Service Notification* (WS-Notification [GHM06]). A arquitetura genérica propõe um conjunto de serviços a serem implementados para prover as funcionalidades de alto-nível esperadas de uma grade. Esses serviços são agrupados em: gerenciamento de execução; dados; gerenciamento de recursos; segurança; auto-gerenciamento; e meta-informação.

Pode-se classificar basicamente dois tipos de grades, de acordo com o seu uso: grade de processamento e grade de dados. Uma grade de processamento tem como prioridade prover o compartilhamento de recursos heterogêneos e dinâmicos para a solução de problemas intensivos em computação, não possíveis de serem resolvidos por computadores ou agregados (*clusters*) tradicionais de maneira custo-efetiva [FK99]. Por sua vez, a grade de dados [Cas02] tem por finalidade principal compartilhar recursos dispersos de dados, capacidade de armazenamento e largura de banda em rede para resolver problemas intensivos no uso e na transformação de dados.

2.2.1 Grades de Dados

As grades de dados são compostas de infra-estrutura e *middleware* de controle e de compartilhamento de recursos computacionais que contêm mecanismos de transporte de dados e execução de processos, projetadas para facilitar a criação e a execução de aplicações que realizam computações sobre grandes volumes de dados. Elas geralmente suportam projetos científicos ou de engenharia de grande porte, que envolvem várias universidades. As experiências e simulações nas áreas de física, biologia, medicina, química e astronomia são grandes usuárias dessas grades. Diversos países estão investindo em grades de dados, tanto para fins científicos, quanto comerciais. Os países da União Européia têm como maior projeto em grades de dados o Enabling Grids for E-SciencE (EGEE) [fEs06], também conhecido como EuroGrid, que em 2006 possuía 33.000 processadores e 5 petabytes de armazenamento. Os Estados Unidos possuem dois grandes projetos: Open Science Grid (OSG) [Gri06] e TeraGrid.

O seu funcionamento baseia-se no armazenamento, transformação e transporte de dados entre os vários nós, dos produtores até os consumidores de dados. Os dados brutos são geralmente obtidos de instrumentos científicos, espalhados por uma ou mais instituições, e armazenados em grandes arquivos. Esses arquivos são publicados e qualquer usuário pode descobri-los e utilizá-los para suas pesquisas. A partir desses, um cientista pode realizar computações, em computadores da grade, e gerar dados derivados, também chamados de dados virtuais. Esses dados virtuais, por sua vez, também podem ser publicados, descobertos e gerar novos dados virtuais, e assim sucessivamente, com a finalidade de implementar processos e procedimentos científicos definidos. Meta-dados são geralmente associados aos dados virtuais ou reais com o objetivo de informar sua origem, transformações realizadas e procedimentos de extração.

Um exemplo de sistema para grade de dados é o GriPhyN (Grid Physical Network) Virtual Data System [ZWF+06], em que os usuários podem publicar, descobrir e utilizar dados virtuais, bem como realizar derivações sobre esses e gerar novos dados virtuais a serem publicados e descobertos. As derivações são realizadas por meio de transformações sobre os dados, realizadas por programas que são executados em máquinas da grade. Os dados virtuais são armazenados em nós intermediários da grade, podendo haver réplicas do mesmo dado espalhadas pelos nós da grade.

Outro exemplo dessa categoria de *middleware* é o UNICORE (UNiform Interface for COmputing REsources) [ES01], desenvolvido por várias empresas e instituições e utilizado no EuroGrid. Nesse sistema é dada especial ênfase na segurança de acesso e autonomia para as organizações que participam da grade, de forma transparente ao usuário. Ele também provê mecanismos de submissão e controle de tarefas e *workflows* de dados (aplicações baseadas em tarefas inter-dependentes em dados) executados em um centro de computação da grade. A transferência de dados e arquivos entre as tarefas é feita de maneira automática.

Há vários exemplos importantes de uso dessas grades em aplicações científicas: análise de genoma [SRD+05], no qual bancos de genomas de diferentes espécies são comparados para se descobrir o significado de certas porções do genoma quanto a modificações do fenótipo ou metabolismo nessas espécies; busca de ondas gravitacionais em rede de instrumentos [Dat06]; e consultas sobre banco de imagens anatômicas do cérebro de diversos pacientes, obtidas por ressonância magnética, para análise científica [ZWF+06].

2.2.2 Grades de Processamento

As grades de processamento são infra-estruturas de hardware e software utilizadas para gerenciar o acesso e o compartilhamento de recursos distribuídos com o objetivo de utilizar o poder computacional de tais recursos disponíveis para executar aplicações paralelas

intensivas em processamento, não baseadas somente na transformação e no armazenamento de grande volume de dados em máquinas intermediárias. Existem vários tipos de *middleware* para grades de processamento. Eles podem ser classificados, de acordo com o seu modelo computacional, em sistemas que suportam aplicações compostas por: tarefas independentes; objetos distribuídos; módulos paralelos com comunicação baseada em troca de mensagens; varredura de coleções; e processos do tipo *workflow*.

Modelos de Programação

Tarefas Independentes A maioria dos *middlewares* para grades de processamento é baseada no conceito de execução de coleções de tarefas independentes (*bag of tasks*), ou seja, tarefas nos quais a suas inter-dependências em dados e em controle são desconhecidas pelo sistema. Nesses sistemas não há modelagem de dependências entre tarefas e os serviços existentes basicamente fornecem: informações de registro e estado dos recursos disponíveis, acesso seguro a recursos computacionais, e mecanismos de localização e transporte de dados distribuídos. Alguns suportam o monitoramento e o controle das tarefas executadas nos recursos computacionais.

Uma vez que não há dependências explícitas entre tarefas, esses sistemas não suportam a coordenação da execução de tarefas e da recuperação automática do estado da aplicação. Assim os usuários devem escrever programas de controle em linguagem de programação genérica que coordenam as tarefas e tratam da recuperação em caso de falhas, reiniciando o controle da execução e gerenciando quais tarefas foram executadas e quais ainda não foram.

Os sistemas mais conhecidos nessa categoria são Condor [TTL02] e Globus [Fos05]. Condor foi o pioneiro e teve como objetivo inicial compartilhar máquinas disponíveis em um laboratório para executar aplicações diversas baseadas em tarefas paralelas. Seu serviço *matchmaker* é responsável por localizar recursos computacionais na grade que satisfaçam os requisitos desejados das tarefas a serem executadas. Ele suporta o controle e o monitoramento de tarefas, simulação de ambiente virtual no recurso computacional e recuperação de falhas de tarefas, com re-execução. Globus, embora seja mais considerado como um conjunto de ferramentas, procurou uniformizar esses conceitos com a finalidade de facilitar a criação de qualquer aplicação sobre grades computacionais e está colaborando com a definição da OGSA (Open Grid Service Architecture) no desenvolvimento de padrões de protocolos e serviços de grade. Ele é composto de serviços de informação e monitoramento de recursos, transmissão de dados e execução de tarefas independentes, e provê protocolos de comunicação e acesso seguro. No capítulo 3 descreveremos de forma mais detalhada esses importantes *middlewares*.

Há vários exemplos de aplicações executadas sobre esses sistemas com o intuito de utilizar o poder computacional e de transporte de dados da grade, tais como aplicações

de previsão de tempo (Earth Systems Grid [Gri07b]), otimização numérica (MetaNEOS [Pro02]), simulação de terremotos (NEESgrid [Cen07]), pesquisa de novos medicamentos, análise estrutural de construções e renderização final de animações gráficas em 3D (GRIA – Grid Resources for Industrial Applications [GRI07a]).

Objetos Distribuídos Os *middlewares* para computação em grade baseados em objetos distribuídos, tais como Legion [GW96] e InteGrade [GKGF04], provêem uma abstração de objetos para a exposição dos serviços e para a composição de aplicações distribuídas. Os serviços existentes permitem descobrir máquinas para executar objetos, controlar a disponibilidade dos recursos e buscar objetos para compor a aplicação. Não há uma dependência explícita, na visão do *middleware*, da ordem de invocação entre objetos, o que dificulta o monitoramento da execução e a recuperação de falhas das aplicações distribuídas.

Esses sistemas dispõem das principais funcionalidades dos *middlewares* para aplicações compostas por objetos distribuídos (e. g. CORBA [OH98] e COM+ [Raj98]), dentre as quais localização e invocação de objetos. Entretanto, eles adicionalmente possuem funcionalidades de monitoramento e descoberta de máquinas disponíveis, distribuição de objetos para execução, e, no caso do InteGrade, um certo nível de recuperação da aplicação em caso de falhas, sendo que nos *middlewares* tradicionais um objeto é instanciado em uma máquina pré-determinada e não há recuperação automática do estado de execução.

Troca de Mensagens Os sistemas que gerenciam e executam aplicações paralelas com comunicação realizada por troca de mensagens, que utilizam a interface de programação *Message Passing Interface* (MPI) [For94], são parecidos com sistemas de suporte a tarefas independentes, mas provêem a facilidade da comunicação explícita entre elas. A principal vantagem é essa explicitação da comunicação entre tarefas, o que permite utilizar melhor os recursos computacionais para minimizar os tempos das trocas de mensagens. Entretanto, tais aplicações são difíceis de se monitorar, pouco previsíveis, e o suporte a recuperação de falhas, embora exista, é custoso por se basear em algoritmos de *checkpointing* distribuídos.

A ferramenta MPICH-G2 [KTF03] é um exemplo desse modelo. Ele é baseado na implementação MPICH do modelo MPI e utiliza os serviços do Globus para grades computacionais. Os serviços Globus são utilizados para descobrir os recursos computacionais e para controlar a execução das tarefas, ou módulos paralelos, nesses recursos. A comunicação entre os módulos é realizada por troca de mensagens por meio da interface MPI.

Varredura de Coleções O modelo de programação de varredura de coleções para grades permite que o usuário especifique uma coleção de elementos em cada parâmetro de entrada da aplicação e uma tarefa para ser executada para cada combinação dos elementos dessas coleções. O sistema varre as coleções de elementos, escalona, aloca e distribui instâncias dessa tarefa pelos recursos disponíveis na grade, passando como entrada, para cada tarefa, uma combinação dos elementos. A aplicação termina quando todas as coleções forem varridas, todas as tarefas associadas terminarem, e todos resultados forem agregados.

Nimrod/G [AGK00, BAG00] é um exemplo desse tipo de sistema. Ele contém um escalonador e controlador, denominado *Nimrod Resource Broker*, que utiliza os serviços do Globus para descobrir e acessar recursos disponíveis na grade e controlar a execução das tarefas nesses recursos, obedecendo um certo *deadline* e custo máximo definido para a execução da aplicação.

Processos Workflow O modelo baseado em processos do tipo *workflow* é o mais recente e avançado. Ele é utilizado para especificar aplicações genéricas, complexas e de longa duração, científicas ou comerciais. Por esse motivo, o modelo de programação do *middleware* proposto nesta tese é baseado em processos *workflow*.

Nesse modelo, uma aplicação é descrita como um processo composto de atividades interdependentes. Ele permite que o programador do processo especifique a ordem de execução das atividades e os tipos de dados trocados entre elas, bem como as características dos recursos necessários para executá-las. A partir dessa especificação, o *middleware* pode coordenar automaticamente a execução das atividades na ordem definida e utilizar os recursos adequados da grade computacional. Na seção 2.3 apresentaremos os conceitos básicos dos processos *workflow* e na seção 2.4 mostraremos como esse modelo é suportado atualmente nas grades computacionais.

2.3 Sistemas de Gerenciamento de *Workflow*

O conceito de processos *workflow* já está bastante sedimentado na área de modelagem e gerenciamento de processos de negócio, empregado principalmente na área administrativa de empresas e governos. O modelo de referência para os sistemas de gerenciamento de *workflow* (WFMSs) [Hol94], que define os elementos básicos constituintes de processos workflow, bem como a arquitetura e as interfaces padrões para esses sistemas, foi desenvolvido pela *Workflow Management Coalition* (WfMC) [Coa06].

O modelo de processos workflow baseia-se na idéia de que um processo tem por objetivo resolver um problema complexo pela coordenação de várias atividades que resolvem problemas menores. Nesse contexto, um processo é composto por atividades que definem

quais trabalhos devem ser executados, por quais agentes (humanos ou automáticos), em qual ordem e com quais objetos de dados.

2.3.1 Modelo de Especificação de Workflows

Para a área de modelagem de processos *workflow*, também chamados de processos de negócio, foram criadas várias linguagens de especificação. Em geral estas comportam a especificação: das atividades a serem executadas em um processo (aspecto funcional); das dependências de execução entre as atividades (aspecto comportamental); das dependências de dados entre as atividades (aspecto informacional); e dos agentes responsáveis pela execução de cada atividade (aspecto organizacional).

A dependências de execução entre as atividades especifica o fluxo de controle de um processo. Essas dependências indicam quais atividades serão executadas após o término de uma determinada atividade, se esta não for final. A execução paralela de atividades é especificada por meio de construtores de fluxo que disparam várias atividades simultaneamente após o término de uma atividade, denominados *fork*. Para sincronizar o término de atividades paralelas para depois executar a próxima atividade, utiliza-se o construtor de fluxo de junção, denominado *join*. Há dois tipos básicos de junção: alternativa (*or join*), onde a atividade seguinte é executada após o término de qualquer uma das anteriores ligadas à junção; e total (*and join*), em que a atividade seguinte é executada após o término de todas as anteriores ligadas à junção. A maioria das linguagens possibilita a definição de dependência de controle de forma condicional, em que uma atividade somente é executada após a anterior terminar e se a condição associada à dependência for verdadeira.

A Figura 2.1 apresenta um exemplo de processo *workflow* de compra de livros em uma loja virtual. A primeira atividade, seleção de livros, é realizada pelo cliente da loja. Se algum livro selecionado para compra faltar no estoque, o processo então invoca a atividade de encomenda de livros, que será realizada pela editora. Concorrentemente, o processo invoca a atividade de débito de cartão de crédito, que será executada pela administradora do cartão, utilizando o número do cartão, senha e o valor dos livros e do frete. A atividade de entrega de livros somente será executada pela empresa transportadora quando as atividades de débito do cartão e encomenda dos livros terminarem, utilizando os livros entregues e o valor monetário debitado para frete.

Duas linguagens se destacam por serem definidas por órgãos de padronização: XPDL [Coa05] e WS-BPEL [AC03, TC05]. A linguagem XPDL, acrônimo de *XML Process Definition Language*, foi definida pela Workflow Management Coalition (WfMC) [Coa06]. Ela é baseada em redes de Petri e utiliza documentos no formato XML para especificar processos. Contém como principais elementos: processos, atividades, aplicações, transições,

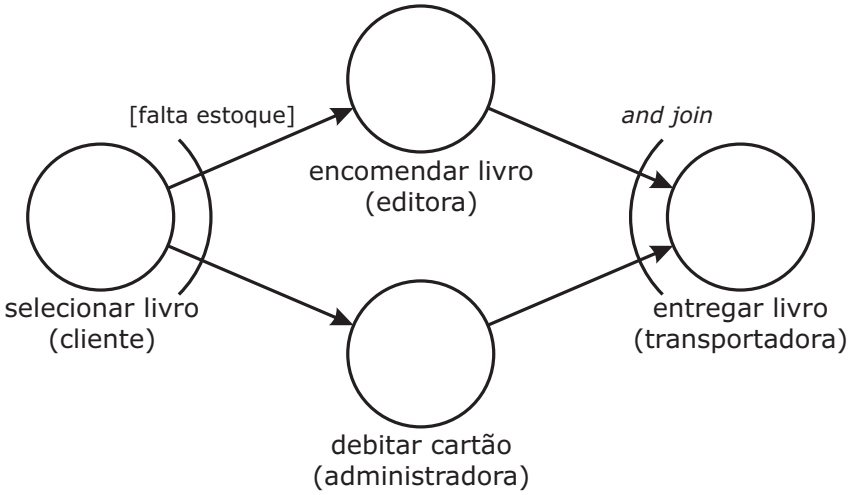

Figura 2.1: Exemplo de um Processo *Workflow* de Compra de Livros.

tipos de dados, campos e participantes. Um processo contém a definição de atividades, atribuição de participantes e aplicações às atividades, transições e campos. Em geral, as atividades são unidades de trabalho que podem ser executadas pela combinação de aplicações e participantes. As atividades também podem se referir a subprocessos ou serem vazias, simplesmente para rotear o fluxo de controle. As transições são condicionais e definem a precedência de execução entre atividades. A junção de fluxos de controle (*join*), ou seja, quais atividades devem terminar antes que esta execute, é descrita na própria atividade, antes de seu corpo. O disparo de vários fluxos (*fork*), ou seja, quais atividades devem executar após o término desta, é especificado após o seu corpo. O fluxo de dados entre as atividades ocorre por meio de objetos de dados que são trocados utilizando-se os campos do processo. Cada participante atribuído para desempenhar uma certa atividade pode se referir a uma pessoa em particular ou a um grupo de seres humanos ou agentes automatizados com certos conhecimentos, habilidades ou que desempenham um certo papel na organização. Em tempo de execução, as atividades prontas são alocadas às listas de atividades dos participantes associados, para ser executada.

A linguagem WS-BPEL, *Business Process Execution Language for Web Services*, foi padronizada pelo consórcio OASIS [Cor06] e é amplamente aceita para a especificação de processos de negócios com a utilização da tecnologia *Web Services* [Con01]. Essa linguagem também define processos, abstratos ou concretos, em XML e é estruturada, de forma parecida às linguagens de programação estruturadas, mas permite a dependência de controle entre ramos distintos da hierarquia de controle e tratamento de mensagens assíncronas. Um processo basicamente contém: referências a parceiros de negócios, que

endereçam indiretamente *web services*; definição de atividades compostas ou simples; ligações de controle; e definição de tipos de dados e variáveis. As atividades compostas são construtores de controle estruturados que permitem o agrupamento de atividades e se assemelham aos construtores de linguagem de programação genérica, mas permitem a definição de fluxos paralelos. As atividades compostas definidas na linguagem são: *sequence* (seqüência), *flow* (paralelo), *while* (iterativo), *switch* (condicional) e *pick* (espera de mensagem). As atividades simples permitem a interação síncrona ou assíncrona com os *web services*. As atividades simples definidas na linguagem são: *invoke* (invocação síncrona), *reply* (envio de mensagem) e *receive* (recepção de mensagem). Há também atividades para tratamento de exceções, eventos e compensação, além de temporização. É possível especificar dependências entre atividades de ramos diferentes da hierarquia, ultrapassando a estrutura definida pelas atividades compostas, por meio de *links* (ligações). Essas ligações podem ser condicionais e podem ligar uma atividade a várias outras, sendo esta destino ou fonte de ligações. Quando uma atividade é fonte de ligações, tem-se uma distribuição de fluxos de controle paralelos (*fork*) a partir desta; por outro lado, quando uma atividade é destino de ligações, tem-se uma junção de fluxos de controle (*join*) nesta. Na atividade destino de ligação de controle, é possível especificar junções exclusivas (*or join*) ou inclusivas (*and join*). O fluxo de dados dentro de um processo ocorre por meio do uso de variáveis compartilhadas e cópia desses valores. Entre um processo e os serviços *web*, o fluxo de dados ocorre por meio de mensagens.

Embora a linguagem WS-BPEL seja bastante aceita para se modelar processos de negócio, ela é inadequada para se modelar processos de longa-duração e que necessitem de alto-desempenho de execução em ambientes como grades computacionais. O suporte limitado ao paralelismo não permite, por exemplo, criar um número de atividades paralelas determinado em tempo de execução. Tampouco há possibilidade de se especificar recursos computacionais mais adequados para executar cada tipo específico de atividade, distribuída dinamicamente de acordo com as condições variáveis de disponibilidade de recursos na grade. Esta linguagem também não permite descrever um processo em vários níveis, o que impede a distribuição dinâmica e execução de partes deste processo em grupos de máquinas mais adequados, centralizando, pois, o controle do processo.

2.3.2 Arquitetura de Sistemas de Gerenciamento de Workflow

Segundo o modelo de referência da WfMC [Hol94, H+04], um sistema de gerenciamento de *workflow* contém: ferramentas de especificação de processos *workflow*; máquina de execução de processos; e interfaces para administração e monitoramento de processos, para interação de participantes com as suas listas de atividades, e para invocação de aplicações externas usadas pelas atividades. Também é possível haver uma interface de

interação entre máquinas de execução distintas, para os casos em que a execução seja distribuída.

A ferramenta de especificação permite aos usuários descrever processos na linguagem utilizada pelo sistema, que sempre contém a noção básica de processos, atividades, transições, participantes e aplicações utilizadas. Objetos de dados podem ser passados entre atividades e também utilizados pelo próprio processo para avaliar as condições das transições, determinar quais atividades serão executadas e os participantes a serem atribuídos a estas.

Um processo pode ser executado de forma centralizada, com a utilização de uma máquina de execução, ou distribuída, por meio da cooperação de várias máquinas. Na execução de uma instância de um determinado processo, denominado caso, a máquina de execução lê a especificação desse processo e mantém dados de controle pertinentes a esta instância, tais como estado de execução e variáveis internas. Podem existir várias instâncias (ou casos) de uma mesma especificação (ou tipo) de processo sendo executadas por uma máquina, cada uma com seus respectivos estado de execução e variáveis internas. Durante a interpretação de uma instância, a máquina de *workflow* utiliza as variáveis internas para decidir qual é a próxima atividade a ser executada, com que dados, quem a executará e utilizando qual aplicação. Uma vez decidida qual atividade será executada, a máquina utiliza a interface com as listas de atividades dos participantes selecionados para notificá-los que esta atividade está pronta para ser executada por um deles. A máquina de *workflow* passa os dados de entrada da atividade para esta ser executada pelo participante selecionado. A interface com a aplicação é utilizada também para a invocar a aplicação associada à atividade.

Um participante do processo pode ser escolhido de acordo com o seu papel dentro da organização ou seus atributos, conhecimento e habilidades. Quando há mais de um participante que satisfaz os requisitos para executar uma atividade, a máquina de *workflow* pode enviar a atividade para a lista de atividades de um deles ou de todos eles. No último caso, o primeiro a selecionar a atividade se torna o responsável por executá-la.

A interface do usuário permite que este se autentique para o sistema, visualize as atividades associadas a ele e selecione qual delas deseja executar. Ao selecionar uma atividade, se houver uma aplicação associada, a interface é responsável por invocá-la passando os dados necessários para realizar a atividade. Quando o participante termina de executar a atividade, a interface repassa os dados resultantes da sua execução para a máquina de *workflow*. A máquina então utiliza esses dados para continuar a interpretação e navegação pela execução do processo.

A interface de administração permite monitorar os processos sendo executados, o estado da execução desses e o valor de suas variáveis internas. É possível, também, administrar o modelo organizacional que associa as unidades da organização e seus membros aos participantes de um processo, bem como registrar aplicações a serem utilizadas.

Os sistemas tradicionais de gerenciamento de *workflow*, entretanto, não são adequados para executar aplicações de longa duração intensivas em computação, isso porque:

- sua linguagem de especificação e máquina de execução são focados em processos de negócio, que envolvem principalmente humanos, não máquinas, e portanto não tem como objetivo o desempenho da execução, nem o melhor escalonamento, distribuição e controle das atividades;

- a máquina de execução de processo é geralmente centralizada e, portanto, não tem a preocupação de distribuir o processo de forma a prover uma melhor escalabilidade e disponibilidade do sistema e das aplicações;

- a máquina de execução distribuída se preocupa em distribuir os processos entre as organizações de acordo com a responsabilidade desta no processo, não distribuí-los de forma a aproveitar os recursos disponíveis do ambiente para prover um maior desempenho da aplicação;

- a máquina de execução desconsidera o ambiente da grade computacional, que fornece um número crescente de recursos dinamicamente disponíveis, entre várias organizações autônomas e heterogêneas, para executar aplicações de forma adaptativa e eficiente.

Em suma, tais sistemas de gerenciamento de *workflow* não são eficientes e confiáveis para a execução de aplicações de longa duração intensivas em computação, pois não foram projetados para otimizar o escalonamento e a distribuição de partes de processos e atividades, de forma adaptativa, nos recursos dinâmicos e autônomos disponíveis. Entretanto, o modelo de processos *workflow*, no que tange as informações de dependências entre atividades e controle do estado do processo, é fundamental para a otimização do escalonamento, distribuição e controle das atividades, bem como para a recuperação automática do processo. Para aproveitar esse modelo computacional, mas de forma a executar eficientemente aplicações de longa duração, recentemente estão surgindo sistemas de grades computacionais que permitem a execução de processos do tipo *workflow*. Apresentaremos as principais características dos sistemas existentes na seção a seguir.

2.4 Sistemas de Suporte a *Workflows* em Grades Computacionais

Um estudo foi desenvolvido recentemente para classificar os diversos tipos de sistemas de suporte a processos *workflow* em grades [YB05]. Esse estudo apresenta e classifica as características desses sistemas nos aspectos: modelo de especificação de processos, mecanismo de escalonamento, tolerância a falhas e movimentação de dados. Nessa seção, estendemos esse trabalho para incluir o aspecto de coordenação de execução de processos e detalhar e aprofundar os demais aspectos.

2.4.1 Modelo de Programação de Processos

Os modelos de programação dos sistemas de suporte a processos em grades computacionais estendem o modelo de programação encontrado em sistemas de gerenciamento de workflows a fim de permitir a especificação dos recursos necessários para executar uma certa atividade e aumentar o paralelismo de execução. De modo geral, eles desconsideram a visão de participantes e descrevem melhor os recursos necessários para executar uma atividade, tais como memória e disco a serem utilizados, software e bibliotecas necessárias.

Formas de Especificação Um processo é composto de atividades (também conhecidas como tarefas), dependências de execução e de dados entre atividades, e atribuições de recursos e software necessários para executar cada atividade. Em geral pode-se dizer que os modelos de programação são declarativos e definem processos como grafos orientados acíclicos (DAGs – *Directed Acyclic Graphs*), nos quais os nós definem as atividades a serem executadas e as arestas definem as dependências de dados e de controle. Uma dependência de controle entre duas atividades pode ser condicional, sendo esta ativada quando a condição associada for verdadeira. Uma atividade pode ser destino de algumas dependências e origem de outras dependências de controle e dados. Uma atividade que contém várias dependências de saída pode disparar (*fork*) a execução de várias atividades simultaneamente. Uma atividade somente tem a sua execução iniciada quando qualquer ou todas as dependências de controle estiverem ativadas, dependendo se a definição da junção de controle for exclusiva (*or join*) ou inclusiva (*and join*), respectivamente.

De acordo com um estudo recente sobre esses sistemas de suporte [YB05], há modelos não baseados em DAGs, tais como os baseados em grafos orientados cíclicos (DCGs – *Directed Cyclic Graphs*), mas também há os baseados em linguagens imperativas estruturadas (e.g. WS-BPEL [TC05]). Ambos permitem a especificação de iteração (laços) no fluxo de controle, ou seja, permite que uma certa parte do processo possa ser executada várias vezes. Os modelos baseados em DCGs têm como efeito colateral tornar

não-estruturada a especificação de processos, sendo o seu comportamento durante a execução de difícil previsão, como em redes de Petri, em que as dependências podem gerar ativações indesejadas em saltos para atividades anteriores [EN94]. Por outro lado, os poucos modelos baseados em linguagens estruturadas somente recentemente começaram a surgir. Esses simplificam a especificação e o monitoramento de processos, mas ainda não permitem o paralelismo e a expressividade necessários para as aplicações em grade. Por exemplo, o WS-BPEL, utilizado em processos de negócio, ainda não foi adaptado para aplicações científicas e de alto-desempenho em grade, e não suporta a criação de um número arbitrário de atividades em tempo de execução, tampouco permite especificar transformações de dados complexas e selecionar recursos adequados em tempo de execução.

Processos Concretos e Abstratos Os processos especificados podem ser considerados concretos ou abstratos. Nos processos concretos os recursos necessários para executar uma atividade são associados em tempo de especificação. Por outro lado, nos processos abstratos há somente a especificação das características dos recursos que uma certa atividade requer e somente em tempo de execução o sistema seleciona, entre os recursos que satisfazem essas características, qual serão os recursos que executarão a atividade, utilizando informações de disponibilidade e balanceamento de carga fornecidas pela grade.

Modelagem Gráfica e Textual Os sistemas geralmente aceitam a especificação de processos por meio de linguagens textuais ou ferramentas gráficas. As ferramentas gráficas são normalmente mais fáceis de usar, pois permitem aos usuários descrever processos simplesmente arrastando e ligando os componentes disponíveis. Esses componentes podem tanto representar atividades ou controles de fluxo já especificados, quanto repositórios de dados e recursos computacionais existentes. As ligações, por sua vez, representam as dependências de dados e de controle entre atividades, e associação das atividades aos recursos. Entretanto, a construção de processos de forma gráfica é bastante limitada, pois permite somente descrever processos declarativos, sendo necessária a utilização de caixas textuais para a especificação de condições, transformações, fórmulas e controles mais complexos. Assim, a linguagem textual, mesmo que mesclada com a ferramenta gráfica, apesar de ser menos intuitiva, permite a especificação de processos complexos, que necessitam de realizar transformações de dados e controle fino de sua execução.

2.4.2 Arquitetura de Sistemas de Suporte a Processos

Os sistemas de suporte a processos existentes têm arquiteturas distintas, mas basicamente fornecem os seguintes serviços: interpretação de processos workflow; escalonamento, distribuição e coordenação da execução de atividades; manutenção de informação sobre as características e o estado dos recursos computacionais; e controle de acesso e execução de atividades nos recursos computacionais [AFFH98, Bau04]. Alguns sistemas fornecem adicionalmente os seguintes serviços: escalonamento, distribuição e coordenação da execução de sub-processos [YYR03]; pré-alocação de recursos [CJSN03]; localização e transferência de dados; recuperação de processos e tolerância a falhas [Bau04]; e suporte a autenticação e segurança de informação.

Em comparação com os sistemas de gerenciamento de *workflow*, os sistemas para grades computacionais se preocupam mais com a escalabilidade, desempenho e confiabilidade da execução distribuída de processos em organizações com administração autônoma e com recursos heterogêneos de disponibilidade variável. Os algoritmos de escalonamento e distribuição, por exemplo, devem consideram o poder de processamento dos recursos disponíveis e possivelmente a largura de banda de comunicação entre nós da grade, o que não é necessário em aplicações genéricas de *workflow*.

Um sistema de suporte a processos necessita da informação do estado e das características de cada recurso computacional onde serão executadas as atividades do processo, bem como do acesso a esses recursos para alocação e execução das atividades. A execução de um processo compreende a interpretação do processo e a coordenação das atividades do processo. A coordenação envolve as fases de escalonamento de atividades, obedecendo a ordem de execução determinada pelo fluxo de controle do processo; a distribuição das atividades e dos dados para os recursos selecionados no escalonamento; e o controle da execução de cada atividade.

Distribuição da Execução de Processos

As arquiteturas dos sistemas de suporte a processos em grade são distribuídas, algumas abrangendo recursos de uma rede local em uma instituição, outras abrangendo várias redes, em várias instituições autônomas e potencialmente toda a Internet. Entretanto, mesmo o sistema sendo distribuído, o serviço de interpretação e execução de processos pode ser centralizado, descentralizado ou hierárquico.

O serviço de execução centralizado é largamente predominante, devido à facilidade de controlar a obtenção de informações da grade e os escalonamento e distribuição de atividades. A deficiência desse modelo é sua baixa escalabilidade, desempenho e confiabilidade na execução de processos. Na extremidade oposta há serviços de execução totalmente distribuídos, em que cada nó participa de uma rede *peer-to-peer*. Nesse modelo, cada nó

conhece exatamente a parte do processo que deve executar e quais são os nós que contêm atividades anteriores e posteriores no fluxo, dos quais deve esperar dados e para os quais deve enviar resultados, respectivamente. Esse modelo distribuído resolve o problema de escalabilidade e confiabilidade dos sistemas centralizados. Por outro lado, dificulta a coordenação e o monitoramento de atividades do processos, bem como impede uma adaptação dinâmica com conhecimento global das condições de disponibilidades dinâmicas dos recursos das grades computacionais. O modelo hierárquico é o intermediário, que busca um melhor compromisso com a escalabilidade e a confiabilidade do sistema, mas sem perder o controle do escalonamento e da adaptação dinâmica eficaz. Existe um ou outro sistema [CJSN03] que se classifica como hierárquico, mas nenhum deles é adaptável e lida com distribuição em vários níveis.

Escalonamento de Atividades

O escalonamento de atividades, que é a fase mais complexa da execução de um processo, envolve a seleção de atividades a serem escalonadas e a escolha dos recursos mais adequados para a execução de cada atividade, considerando a ordem de execução dessas, de forma a reduzir o tempo de execução do processo. O escalonamento de atividades inter-dependentes de tamanho variável em recursos de poder computacional variável é um problema NP-completo [ERAL95]. Dessa forma, os algoritmos de escalonamento existentes utilizam heurísticas para a busca de soluções quase-ótimas de forma que o tempo de execução do algoritmo seja aceitável, isto é, apenas um fator do tempo previsto de execução do processo. Esse problema se torna ainda mais difícil e desafiador ao se considerar que a grade pode conter milhões de recursos a disposição em diversas organizações, cada um com capacidade e disponibilidade variável no tempo. Para o problema ficar completo, pode ser necessário também considerar o esforço de comunicação entre atividades inter-dependentes juntamente com o custo da comunicação entre os nós a alocados a essas atividades de forma a reduzir o tempo de comunicação do processo, que pode não ser desprezível em relação ao tempo total previsto de execução do processo.

Estratégias de Escalonamento Diferentes algoritmos de escalonamento podem visar objetivos distintos, entre esses podemos destacar: buscar o melhor desempenho de execução de um processo, reduzir o custo total dos recursos alocados, e aumentar a confiabilidade da execução. Também é possível que um algoritmo vise a todos os objetivos, mas considerando cada um com um certo peso para cada atividade ou processo. Os algoritmos que buscam o melhor desempenho do processo focam na escolha de recursos mais eficientes para alocar atividades mais pesadas em termos de esforço de processamento necessário. Os algoritmos que buscam reduzir o custo total da alocação de recursos procuram utilizar os recursos mais caros somente quando necessário para satisfazer alguma

outra restrição, como, por exemplo, o tempo máximo de execução de processo. Por fim, recentemente estão surgindo algoritmos que buscam alocar atividades em recursos mais confiáveis, a fim de aumentar a previsibilidade e reduzir as falhas de execução de atividades nos processos. Há poucos algoritmos que buscam mesclar mais de um desses objetivos no escalonamento de atividades de um processo, devido à complexidade da especificação de restrições e escolha de recursos.

Podemos dividir os algoritmos que buscam um melhor desempenho para a execução de processos em dois tipos: o tipo que somente considera o poder computacional do nó para alocar atividades, e o que considera tanto o poder computacional do nó quanto a largura de banda de comunicação entre os nós. O primeiro é mais simples, pois se baseia em uma informação pontual de desempenho de um nó, e basicamente aloca primeiro as atividades mais críticas, de acordo com o critério do algoritmo, para os nós mais eficientes. Ele utiliza essa heurística de forma a reduzir o tempo de processamento total, sem considerar a comunicação entre as atividades. Complementarmente, o segundo algoritmo visa a sanar essa deficiência considerando também o tempo da troca de mensagem entre nós diferentes que contêm atividades adjacentes. Essa abordagem provê um melhor resultado para processos com troca de grandes volumes de dados, pois considera tanto o tempo de execução de cada atividade nos recursos alocados, quanto como o tempo de comunicação entre atividades adjacentes, determinado pela largura do canal de comunicação entre os nós alocados a essas atividades e o tamanho total estimado das mensagens trocadas. Algoritmos desse tipo tendem, por exemplo, a alocar recursos menos eficientes, mas mais próximos ou com maior largura de banda, visando a reduzir o custo da comunicação entre atividades adjacentes com troca de grandes volumes de dados.

Decisão Local e Global O serviço de escalonamento de atividades de um processo pode realizar decisões locais ou globais, tanto na seleção de atividades a serem escalonadas, quanto na escolha de recursos para executá-las. Na seleção de atividades, um algoritmo de escalonamento local seleciona somente atividades que estão próximas de executar, tomando decisões rápidas, mas que podem ter resultados inferiores a algoritmos globais, que consideram todas as atividades do processo para realizar um escalonamento. Um exemplo são os algoritmos de escalonamento que buscam uma menor duração do processo (*makespan*). No modelo de seleção local de atividades para escalonamento, o algoritmo pode desprezar atividades pesadas e distantes, e alocar recursos mais potentes para atividades leves e próximas e, conseqüentemente, deixar que máquinas menos eficientes executem as atividades mais pesadas posteriormente, aumentando o tempo de execução do processo como um todo. Os algoritmos de seleção local de atividades geralmente provêem resultados inferiores, porém possuem a vantagem de serem mais ágeis do que os correspondentes globais.

No aspecto de escolha de recursos para executar as atividades selecionadas, os algoritmos locais escolhem somente recursos computacionais próximos do nó que está executando o serviço de escalonamento, tipicamente na mesma rede local ou em redes próximas. Comparativamente aos algoritmos de escolha global de recursos, os primeiros são mais simples e rápidos, pois utilizam um número menor de recursos para o escalonamento, mas podem desprezar recursos mais eficientes e redes com maior largura de banda de outras instituições. Utilizando o exemplo de algoritmos que buscam uma menor duração do processo, os algoritmos de escolha global de recursos podem considerar máquinas mais rápidas em redes remotas e alocar atividades pesadas para essas redes, reduzindo a duração do processo.

Planejamento Estático e Dinâmico O escalonamento pode ser baseado em um planejamento estático ou dinâmico de alocação de atividades a recursos. No planejamento estático, todo o escalonamento de atividades é realizado anteriormente à execução do processo, alocando-se recursos para executar cada atividade previamente. De modo diverso, o algoritmo de escalonamento com planejamento dinâmico aloca recursos para cada atividade durante a execução do processo. O modelo estático tem a vantagem de permitir um melhor resultado para grades com pouca variabilidade de disponibilidade de recursos, pois previamente calcula, utilizando heurísticas, qual é o melhor recurso para cada atividade e o aloca. O modelo estático geralmente utiliza a seleção global de atividades, obtendo um bom resultado para todo o processo, tal como menor duração do processo, no caso de estratégia por desempenho.

Por outro lado, o algoritmo com planejamento dinâmico é mais flexível do que o algoritmo com planejamento estático, se adaptando melhor a grades em que há uma variação da capacidade e disponibilidade dos recursos. A grande maioria dos algoritmos dinâmicos é executada por demanda (*just-in-time*), entretanto, considerando qual é o melhor recurso somente para as atividades que estão prontas para executar. A desvantagem dessa abordagem é a visão local para a alocação de atividades, que provê um resultado sub-ótimo para o escalonamento. Recentemente estão surgindo algoritmos de escalonamento com previsão dinâmica [BM06], os quais realizam uma alocação mais abrangente de atividades não-escalonadas do processo em tempo de execução e prevêem quais serão os melhores recursos para executá-las naquele momento. De tempos em tempos, a previsão de alocação é refeita e as atividades são reescalonadas e realocadas se necessário. Nesse caso, atividades também devem ser abortadas e recriadas nos novos recursos alocados, ou migradas para esses novos recursos. Esse último algoritmo é mais complexo do que os anteriores, mas representa um compromisso com a qualidade do escalonamento e com a adaptabilidade às características dinâmicas da grade.

2.4.3 Tolerância a Falhas

Atualmente utilizamos os sistemas computacionais para a maior parte de nossas atividades diárias, em todos os ambientes, desde a residência, até a empresa, a universidade e o Estado. A medida de quanto alguém pode depender de um sistema, ou a *dependability* de um sistema, é conferida pela capacidade deste em atender aos requisitos do usuário ou do domínio para o qual foi desenvolvido. Laprie [Lap95] define quatro características que um sistema deve possuir para dependerem dele, ou seja, para ser considerado *dependable*: confiabilidade (capacidade de executar o serviço corretamente), disponibilidade (capacidade de manter o serviço operante), manutenibilidade (possibilidade de fácil manutenção) e segurança (provisão de privacidade e integridade às informações e de autenticidade dos pares). No caso de sistemas de suporte a grade, para um *middleware* de grade computacional ser *dependable*, ele deve também possuir alto desempenho, escalabilidade e adaptabilidade. Trataremos nessa tese de todas essas características, em maior ou menor grau, de acordo com o escopo do trabalho.

Focando nas características de confiabilidade e disponibilidade, para um sistema computacional possuir essas propriedades ele deve tratar as falhas e erros corretamente de forma a impedir que estes erros prejudiquem a consistência dos dados ou tornem o sistema inacessível, manifestando defeitos no serviço prestado.

Laprie [Lap95] define falha como um problema que ocorre externamente a um componente do sistema computacional, mas que pode ser percebido por esse, tal como falhas físicas ou de especificação do componente. Uma vez percebida, essa falha pode resultar um erro latente ou efetivo na execução do componente. Um exemplo de erro efetivo é a impossibilidade de uma aplicação enviar dados pela rede por causa de uma falha de comunicação. Esse problema interno ao componente, ou erro, é apresentado ao usuário ou componente superior como um defeito. No caso de uma falha de comunicação, que ocasiona um erro no componente, este apresenta um defeito que pode ser presenciado, por exemplo, como uma mensagem ao usuário ou queda da aplicação.

Há basicamente quatro formas de impedir que ocorra um defeito no serviço prestado: isolamento de falhas, tolerância a falhas, correção de erros e previsão de erros. Como o isolamento de falhas é impossibilitado no ambiente dinâmico da grade, e a correção e previsão de erros está relacionado a técnicas de engenharia de software para o desenvolvimento de aplicações, não a sistemas de suporte, focamos a tolerância a falhas, nessa tese, como forma de prevenir a ocorrência de defeitos no sistema e na aplicação, de forma a mantê-los confiáveis e disponíveis.

A tolerância a falhas em *middlewares* de suporte a processos workflow em grades computacionais visa a manter a disponibilidade e a consistência da execução e dos dados dos processos e das atividades. Hwang [HK03] classifica as várias formas existentes de tolerância a falhas de processos, em nível de atividade e de processos. Nessa seção des-

creveremos esses mecanismos de tolerância a falhas e estenderemos o seu trabalho com outros modos de tratamento de falhas, incluindo modos de manutenção de consistência dos dados utilizados, não considerados em seu trabalho.

As falhas podem ser ocasionadas pelo *hardware*, pelo sistema operacional local, pelo *middleware* da grade, denominadas falhas de sistema, ou podem ser ocasionadas pela especificação da aplicação ou interações com outros componentes e dados, denominadas falhas de aplicação. Em nível de atividade, as falhas de sistema são geralmente mascaradas pelo sistema por meio da reexecução ou da restauração de atividades atingidas, de forma que este erro não seja percebido pelo processo ou pelo usuário como defeito na execução da atividade. Falhas de aplicação, entretanto, não são tratadas pelo sistema e se transformam em erros na execução e levam a atividade a falhar. Um exemplo é o erro ocorrido na divisão por zero. Em nível de processo, as falhas de sistemas, geralmente transientes, são tratadas por meio da restauração automática do estado do processo. Para as falhas de aplicação, geralmente permanentes e mais complexas, alguns sistemas levantam o erro como exceção e permitem que o processo trate-a por meio de ações específicas, evitando que o processo falhe. As falhas de atividade, tipo específico de falha de aplicação percebida em nível de processo, podem ser tratadas pela execução de atividades alternativas ou replicadas. No caso de utilização de dados compartilhados por atividades ou processos, alguns sistemas oferecem mecanismos para a manutenção da consistência desses dados, por meio de controle de acesso e restauração do seu estado anterior.

Nível de Atividade

As falhas percebidas em nível de atividade, quando causadas por erros no sistema operacional, no recurso computacional ou na rede de comunicação, consideradas falhas de sistema, são geralmente tratadas transparentemente pelo sistema de suporte. A forma mais básica de tratamento dessas falhas, provavelmente causadas por erros transientes nos próprios recursos, tais como falta de memória, é a re-execução da atividade defeituosa no mesmo recurso um certo número de vezes. A segunda forma de tratamento de falhas, muito encontrada nos sistemas, é a re-execução da atividade em um recurso alternativo. A vantagem dessa abordagem é o tratamento imediato de falhas de sistema, que dificilmente ocorrem simultaneamente em recursos distintos. Complementando essa abordagem, alguns sistemas permitem a execução paralela da mesma atividade replicada em diversos recursos, aceitando o resultado da primeira atividade concluída com sucesso. Ela permite desconsiderar imediatamente todas as atividades defeituosas, por falhas do tipo *fail-safe*, e aproveitar o resultado do recurso mais rápido, mas com o custo de utilizar vários recursos para realizar a mesma tarefa. Alternativamente, para tratar de falhas não *fail-safe*, em que as atividades não deixam de executar, mas fornecem resultados inválidos, os sistemas podem executar atividades em mais de um recurso simultaneamente, obter

todos os resultados, compará-los e aceitar o resultado retornado por um número maior de atividades. Esse tipo de abordagem é utilizado pelo sistema OurGrid [CBA+06], que considera que atividades maliciosas podem ser executadas em recursos não controlados da grade.

Há sistemas que suportam gravar periodicamente o estado de execução da atividade em meio persistente, em determinados pontos consistentes (*checkpoints*), para posterior recuperação da atividade no último ponto salvo, em caso de falhas. A vantagem dessa alternativa é a recuperação de atividades longas em um ponto da execução próximo do ponto da falha, considerando toda a computação já realizada, o que reduz a perda causada pela falha. Essa recuperação pode ser realizada na mesma ou em outra máquina. No último caso, há uma migração da execução da atividade.

Nível de Processo

As falhas percebidas em nível de processo são as falhas de sistema ou de aplicação não-mascaradas no nível de atividade e que foi lançada para cima, ou do próprio processo. Poucos sistemas de suporte permitem o tratamento das falhas ocorridas em atividades ou processos. Nos sistemas em que é permitido esse tratamento, as falhas não-mascaradas em nível de atividade ou de processo são geralmente percebidas e tratadas na própria especificação do processo, por meio de tratamento explícito de exceções ocorridas no fluxo de controle, levantadas por erros efetivos de execução gerados por tais falhas.

Atividades Alternativas A forma mais simples de tratamento de falhas de atividades é a especificação de uma atividade alternativa: caso uma atividade falhe, uma atividade alternativa é executada para recuperar o processo. Essa abordagem é útil, por exemplo, quando se pretende executar uma atividade mais eficiente, mas que consome muita memória primeiro, e, se não for possível, executar uma que demande menos memória e seja mais lenta. Similarmente, um processo pode especificar várias atividades redundantes (e alternativas) a serem executadas simultaneamente, todas com a mesma função, mas com implementações distintas, e aproveitar o resultado da atividade que terminar com sucesso primeiro. Como a execução de atividades replicadas, essa abordagem consome mais recursos da grade, mas permite que a execução de uma alternativa termine mais rápido e com mais chances de sucesso. Há também a possibilidade de especificar que uma atividade não é essencial, e portanto, se ela falhar, pode-se desconsiderá-la.

Tratamento de Exceções Alguns sistemas permitem que se especifique no processo ações específicas para tratar cada tipo de erro gerado por falhas de aplicação ou de sistema. Para isso, esses erros, conhecidos como exceções, são tipificados e identificam algum desvio que pode ocorrer durante a execução do processo, tanto em nível de sistema quanto de

aplicação, tais como arquivo inacessível ou divisão por zero. Os sistemas que permitem o tratamento de exceções fornecem maior flexibilidade ao usuário que especifica o processo, pois permite a ele tratar de cada tipo de exceção, ocorrida dentro de um determinado contexto, de uma forma diferente. A especificação do tratamento de falhas, portanto, fica mais precisa. Por exemplo, no caso de uma exceção de arquivo inexistente, o usuário poderia especificar uma ação para executar uma atividade alternativa que obtenha os dados necessários de outra fonte ou que requisite ao usuário a cópia do arquivo.

Compensação de Atividades Quando não há atividades alternativas, nem tratamento adequado de exceções, e o processo não pode descartar essa atividade e continuar executando a partir desse ponto (*forward recovery*) para tratar de um erro ocasionado por uma falha, faz-se necessário o desfazimento da atividade (ou do bloco de atividades) para que o processo prossiga por um caminho alternativo a partir de um ponto anterior (*backward recovery*) ou para que se desfaça totalmente. Para desfazer atividades que não foram concluídas, pode-se simplesmente abortá-la. Entretanto, para desfazer atividades concluídas, que alteraram dados compartilhados e que possivelmente já foram lidos ou sobrescritos por outras atividades, é preciso executar uma atividade reversa, denominada atividade de compensação. Por exemplo, para um processo com duas atividades, uma que debita uma conta A em x e a próxima que credita a conta B com esse valor x, é necessário executar a compensação da primeira atividade caso não seja possível executar a segunda por um erro persistente, a fim de manter a consistência dos dados das contas. Até o momento não se conhece algum sistema de grade que contenha o suporte explícito a esse tipo de tratamento de falhas, mas ele pode ser conseguido indiretamente por meio do tratamento de exceções tipadas, que executa atividades compensatórias no caso da falha de uma ou mais atividades e depois retenta as mesmas atividades ou prossegue por caminho diverso após a referida compensação.

Modificação de Estrutura e Estado de Processos Infelizmente nem todas as falhas são mascaradas pelo sistema ou são previstas para serem tratadas na especificação do processo. No caso de uma falha não tratada, todo o processo é suspenso ou abortado, dependendo do sistema, e o erro gerado sobe até o usuário, que deve tratá-lo. O erro, nesse caso, pode ser causado por uma simples falta de um recurso específico para executar uma atividade ou até pela necessidade de se adicionar uma atividade para realizar uma computação não prevista em tempo de especificação. O tratamento mais comum desse tipo de exceção, nos sistemas existentes, é o cancelamento da execução do processo. Infelizmente, esse não é um tratamento adequado para processos que podem levar dias para terminar. Para superar esse problema, alguns sistemas estão tentando prover mecanismos de modificação dinâmica do estado e da estrutura do processo em tempo de execução,

sem precisar abortá-lo. Dessa forma, um usuário pode adicionar, trocar ou excluir uma atividade determinada, modificando os fluxos de controle e dados pertinentes, e depois continuar a execução do processo a partir desse ponto.

Recuperação do Sistema

Além de falhas de sistema e de aplicação, que podem ser percebidas por processos ou atividades, há falhas nas máquinas, nos sistemas operacionais e na comunicação, percebidas pelos serviços do *middleware* de grade computacional.

As falhas do tipo *fail-stop* nos serviços de informação sobre o estado dos recursos da grade não trazem resultados graves, somente indisponibilidade momentânea de tais serviços, e normalmente a recuperação destes ocorrem simplesmente com a sua reinicialização e leitura dos dados persistentes. Falhas no serviço de comunicação podem acarretar erros no envio ou na recepção de mensagens. Esses erros podem ser recuperados com a reinicialização da conexão; retransmissão, pelo emissor, de mensagens não confirmadas; e descarte, pelo receptor, de mensagens já recebidas.

O problema mais grave ocorre na falha do serviço de execução de processos, que deve recuperar o estado de execução e variáveis dos processos (ou partes deles) que estavam executando sob sua responsabilidade. Muitos sistemas experimentais e acadêmicos simplesmente ignoram esse problema, fazendo com que processos longos, de vários dias, sejam abortados ou simplesmente inacabados em estado inconsistente. Os poucos sistemas que tratam desse problema [Bau04] mantêm a localização e o estado de cada atividade em um repositório de dados, gravados a cada modificação, para que seja possível recuperar o estado dos processos e continuar os executando. Entretanto, esses sistemas possuem execução centralizada de processos e como conseqüência podem ocasionar problemas de disponibilidade e escalabilidade.

Manutenção da Consistência dos Dados

Compartilhamento de Dados Grande parte dos *middlewares* para grade lida simplesmente com a manipulação de arquivos replicados de dados. Para executar uma atividade da aplicação, primeiramente o arquivo de entrada e o executável da atividade são copiados para a máquina selecionada para a execução, depois a atividade é executada e os resultados são escritos em outro arquivo. Nesse caso, a atividade tem acesso exclusivo aos arquivos de entrada e de saída e, portanto, a manutenção de consistência é trivial. Entretanto, há aplicações, tais como as comerciais, que necessitam de ter acesso a dados compartilhados durante a execução de cada atividade, de forma a aumentar o nível de concorrência de leituras e escritas, e portanto o desempenho do processo e a escalabilidade do sistema. Um exemplo é uma aplicação de cruzamento de dados de venda e compra

de mercadorias por fornecedores e destinatários a fim de verificar se ambos registraram corretamente suas operações para fins de tributação. Copiar o banco de dados entre todas as empresas para as respectivas atividades de cruzamento seria impraticável. Para o compartilhamento de dados, podemos utilizar tanto bancos de dados relacionais e orientados a objetos, quanto objetos distribuídos e serviços *web*.

Distribuição de Dados Quanto à distribuição, os dados a serem acessados podem estar centralizados em um servidor, distribuídos ou locais às máquinas em que se encontram as atividades que utilizá-los-ão. Os dados compartilhados para leitura e escrita são geralmente centralizados ou distribuídos de forma controlada, nos sistemas distribuídos tradicionais. O acesso a esses dados ocorre remotamente pelas atividades que desejam utilizar ou modificar tais dados, mas de forma sincronizada, com o intuito de manter a consistência desses dados. Por outro lado, os dados de uso exclusivo para leitura e escrita se encontram localmente nas máquinas, como arquivos, no caso de aplicações e *middlewares* para a grade [FK97].

Os dados compartilhados podem ser distribuídos de diversas formas: por tipo (particionamento vertical), por grupo de objetos relacionados ou domínio (particionamento horizontal), por hierarquia ou de forma mista. Um exemplo de particionamento por tipo é a distribuição dos dados de funcionários de uma empresa em um banco de dados e dos dados de vendas em outro. Um exemplo de particionamento por domínio é a distribuição dos dados das vendas de cada estabelecimento em um banco de dados local do estabelecimento. O particionamento hierárquico é um tipo específico de particionamento horizontal, no qual cada nó da estrutura contém objetos relacionados e pode fazer referência a nós inferiores, como no exemplo de serviços de diretórios distribuído. O objetivo da distribuição dos dados é aproximá-los dos processos e atividades que necessitam de utilizá-los, permitindo, assim, um acesso mais rápido e confiável a estes.

Replicação de Dados Um determinado dado pode existir de forma exclusiva, sendo todos os acessos realizados diretamente a ele, ou pode ser replicado em vários nós, sendo os acessos direcionados a alguma das réplicas, de acordo com a localização, visibilidade e versão dessas ou outro critério. A finalidade da replicação vai além da simples distribuição de dados, para fornecer acesso rápido e confiável a cópias, com a mesma versão ou não, de um mesmo dado. Um exemplo de dado geralmente não replicado é conta bancária, por outro lado, um exemplo muito comum de dado replicado em grades são arquivos de dados científicos, tais como informações espaciais de recepção de raios cósmicos. Os dados mais replicados são acessados principalmente para leitura, ou são dados modificáveis, mas com versões inconsistentes aceitáveis e propagadas lentamente. Exemplos de dados desse último tipo são arquivos com código fonte de aplicações.

A replicação dos dados pode ser feita estaticamente ou dinamicamente, assim como o escalonamento de atividades. Na replicação estática, o usuário ou sistema distribui as réplicas de acordo com o seu possível uso pelos processos e atividades nesses locais. Por outro lado, na replicação dinâmica os dados são replicados de acordo com o escalonamento ou durante a execução dos processos, a fim de implantar essas réplicas próximas dos nós em que as atividades serão executadas, para assim reduzir o custo de acesso a estas. A réplica criada para uso exclusivo de uma determinada atividade também é conhecida como *cache* de dados.

Escopo de Acesso O escopo de um acesso a dados pode compreender uma coleção de objetos ou de registros (no caso de bancos de dados relacionais), um objeto ou um registro, ou um campo do objeto ou do registro. Portanto, em um banco de dados relacionais pode haver esses três tipos de escopo de acesso. O mais utilizado, entretanto, é o escopo de registro. No caso de arquivos, o escopo de acesso geralmente é um arquivo inteiro. No acesso a objetos, por meio de *middleware* orientado a objetos (e.g. Enterprise Java Beans), o escopo geralmente é um objeto. Alternativamente, a implementação de um objeto pode definir quais operações necessitam ser executadas isoladamente, podendo envolver vários outros objetos no mesmo escopo de acesso. Esse último modo de acesso é útil para implementar operações sobre coleções de objetos, tal como inclusão de um novo objeto ou listagem consistente de todos objetos da coleção.

Isolamento de Acesso Para manter a consistência dos dados da grade de acordo com as exigências do próprio dado e das atividades que o utilizam, mas mesmo assim permitir o acesso concorrente quando possível, o sistema deve fornecer mecanismos de isolamento de acesso aos dados. Há basicamente dois tipos de isolamento de acesso a arquivos fornecidos pelos sistemas: exclusivo e concorrente, tanto para leitura quanto para escrita. No caso de objetos compartilhados, o isolamento de acesso a esses objetos pode ser tanto controlado pelo *middleware*, com os mesmos modos de isolamento fornecidos pelos bancos de dados relacionais, quanto controlado pelo usuário que implementou o objeto, por meio de mecanismos básicos de controle de acesso, tais como monitores e semáforos.

Para acesso a dados relacionais, obtidos em forma de registros de tabelas de bancos de dados relacionais, as interfaces atuais com bases de dados SQL (e.g. JDBC [And06] e .NET Framework [Cor08b]) oferecem os quatro tipos de isolamento de acesso definidos pelo padrão ANSI-92 SQL. Na leitura inconsistente (*read uncommitted*), a atividade que requisita esse tipo de acesso pode ler registros, mesmo que esses estejam inconsistentes e sendo modificados por outra atividade (ou transação). Na leitura consistente (*read committed*), a atividade pode ler somente registros cujos dados estão consistentes, ou seja, cujas atividades que os modificaram, terminaram com sucesso (*committed*). Entretanto, pode haver

modificações subseqüentes desses registros, quando outras atividades forem terminando de forma concorrente, o que pode causar o problema de leituras não-repetitíveis. Na leitura repetitível (*repeatable read*), é garantido o acesso exclusivo aos registros lidos, não sendo permitido que outra atividade os modifiquem até que a atividade requisitante termine. Entretanto, novos registros, mesmo dentro do escopo de acesso da primeira atividade, podem ser incluídos por outras atividades, o que pode ocasionar obtenção de registros fantasmas na próxima leitura pela primeira atividade. Por fim, na leitura serializável (*serializable*) é garantido o acesso exclusivo aos registros e também às faixas de registros determinadas pelas consultas, prevenindo-se a criação de outros registros na mesma faixa enquanto essa atividade estiver sendo executada. Expressando de outro modo, a escrita concorrente a um determinado registro é permitida sempre que nenhuma outra atividade (ou transação) estiver com um acesso *repeatable read* ou *serializable* a este registro, sendo que no último caso não é permitida a criação de nenhum registro na faixa de registros lidos pela atividade requisitante – a escrita também não é permitida durante o tempo de uma consulta de leitura com isolamento *read committed*. Berenson [BBG+95] fornece uma definição exata e uma crítica a esses modos de isolamento.

Atomicidade das Atividades Existem dois tipos de atividades quanto aos seus efeitos externos e sobre os dados compartilhados: atômicas e não-atômicas. As atividades atômicas, também conhecidas como transacionais, são do tipo "tudo-ou-nada", ou seja, na visão dos dados acessados, todas ou nenhuma das modificações realizadas pela atividade sobre esses dados são bem sucedidas (*committed*). Como conseqüência, no caso de falha de uma atividade atômica, ou de um cancelamento de sua execução, todas as modificações sobre os dados externos são desfeitas (desde que os recursos contendo esses dados sejam transacionais). As atividades não-atômicas, pelo contrário, não exibem esse comportamento e, quando falham, não desfazem as modificações, parciais, sobre os dados externos. Um exemplo de atividade que pode ser declarada não-atômica, sem nenhum prejuízo de consistência, é a que retorna o cálculo de multiplicação de matrizes, onde o resultado somente é obtido no final e nenhum dado compartilhado é acessado para escrita. Por outro lado, um exemplo clássico de atividade que deve ser atômica é a que realiza uma transação financeira entre duas contas, debitando uma conta de um certo valor e creditando outra conta com este valor. Se houver um erro na execução da atividade, ou esta for cancelada, a alteração intermediária de débito da conta deve ser desfeita, recuperando o valor anterior.

Cada atividade transacional pode acessar um banco de dados centralizado ou vários distribuídos. No primeiro caso, o controle do cancelamento (*rollback*) ou conclusão (*commit*) das alterações é realizado pelo gerenciador do banco de dados. No segundo caso, de transações distribuídas, como são acessados vários bancos de dados, o controle é re-

alizado por um gerenciador que executa um protocolo específico, que deve garantir que as alterações nos diversas bancos sejam todas concluídas ou todas canceladas, ou seja, haja um consenso. O protocolo mais usado para transações distribuídas é o *two-phase commit*, que requer uma fase de votação para a conclusão, e outra de efetiva conclusão ou cancelamento.

Além do aspecto tudo-ou-nada da atividade, existem dois tipos de atividades quanto a sua possibilidade de executar totalmente mais de uma vez, sem prejudicar a consistência dos dados: as atividades idempotentes e as não-idempotentes. As atividades idempotentes são as que sempre que executadas produzem os mesmos efeitos visíveis. Um exemplo é a atividade que adiciona sempre o mesmo número primo em um conjunto de primos. Por outro lado, os efeitos visíveis de atividades não-idempotentes podem ser distintos se executadas mais de uma vez. Por exemplo, uma atividade que adiciona um certo valor a uma conta bancária.

Persistência dos Dados Todos os dados, sejam eles compartilhados ou isolados, podem permanecer somente em memória transiente (RAM), ou podem ser armazenados em memória persistente, tais como discos rígidos (HDs) ou unidades de estado sólido (SSDs) (geralmente implementadas com memórias *flash*), para posterior recuperação para a memória RAM. Os dados transientes são lidos e escritos rapidamente, em poucos nanosegundos, em velocidades próximas às dos processadores. Entretanto, eles podem ser perdidos caso haja uma falha no nó que os contêm, diferentemente dos dados persistentes. Esses últimos são recuperados em sua última versão consistente a partir dos meios persistentes e podem continuar sendo utilizados. O custo, entretanto, é o seu acesso bem mais lento, na ordem de milisegundos em média para dados aleatórios, ou de microsegundos em média para dados em bloco. Um perda na ordem de 1.000.000 (um milhão) na ordem de grandeza de velocidade para acessos aleatórios, ou na ordem de 1.000 (mil) para acessos em bloco. Os dados persistentes são mais adequados para armazenar informações com ciclo de vida relativamente longo e quando representam um grande volume, mas não dados cujo acesso deve ser rápido e de vida curta. Por exemplo, o valor de variáveis internas às atividades curtas podem ser transientes, enquanto os valores de contas bancárias devem ser persistentes.

2.5 Resumo

Esse capítulo introduziu os conceitos básicos de *middlewares* para grades computacionais, de sistemas de gerenciamento de workflow e de *middlewares* de suporte a processos workflow em grades computacionais que serão utilizados no restante dessa tese. Para esses três tipos de sistemas, foram apresentados modelos de especificação de aplicações e

arquiteturas utilizadas para especificar, executar e monitorar tais aplicações.

O modelo tradicional de aplicações para grades computacionais é o de tarefas independentes. Neste, o programador é o responsável por escalonar, controlar a execução de cada atividade e restaurar a aplicação em caso de falhas. Para liberar o programador destas tarefas complexas e fornecer um melhor desempenho e confiabilidade às aplicações desenvolvidas, surgiram os *middlewares* de suporte a processos workflow em grades. Nesses, a aplicação é desenvolvida como um processo que contém um conjunto de atividades e dependências de controle e dados entre essas. Os sistemas utilizam essas informações de dependência para prover um escalonamento automático e adaptativo, possivelmente distribuído, e recuperação automática em caso de falhas.

Os *middlewares* de grades computacionais contêm serviços de informação e monitoramento dos recursos disponíveis, de acesso aos recursos e execução de atividades e de comunicação. Os *middlewares* que suportam processos workflow contêm adicionalmente serviços de execução de processos workflow e poucos provêem tolerância a falhas. Como a grade é um ambiente dinâmico, o escalonamento de processos necessita ser adaptativo, selecionando os melhores recursos para as atividades do processo dinamicamente, em tempo de execução. A tolerância a falhas é necessária para prover confiabilidade e disponibilidade às aplicações de grade.

Capítulo 3

Trabalhos Relacionados

Este capítulo descreve os trabalhos relacionados a grades computacionais e suporte a execução distribuída de processos. Na Seção 3.1 são descritos quatro dentre os principais *middlewares* de grades computacionais existentes, seja pelo seu pioneirismo, seja pela sua importância atual; e por último dois *middlewares* que podem utilizar máquinas de uso comum, quando disponíveis, para aplicações em grades computacionais, também chamados de *middlewares* oportunistas. Na Seção 3.2 são descritos dez *middlewares* de grades, entre os mais conhecidos, que suportam a execução de processos workflow, com escalonamento centralizado ou distribuído, estático ou dinâmico, com recuperação ou não de processos.

3.1 Grade Computacional

3.1.1 *Middlewares* Precursores

Condor [TTL02] é um dos *middlewares* precursores no controle e na execução de tarefas distribuídas em recursos dedicados e máquinas de mesa disponíveis. Inicialmente ele apoiava somente a execução de tarefas em um *pool* local de máquinas, administrado por uma única instituição, mas posteriormente incorporou o conceito de grade computacional, aceitando a execução de tarefas em várias instituições e redes autônomas, com sua extensão denominada Condor-G [FTL+01]. A infra-estrutura do Condor conta com três serviços principais: escalonador, gerenciador de grupo (também conhecido como *matchmaker*) e gerenciador de recursos. O usuário utiliza o escalonador para requisitar a execução de tarefas, e o administrador da instituição utiliza gerenciadores de recurso para disponibilizar os recursos. Ao requisitar a execução, o escalonador solicita ao *matchmaker* os recursos disponíveis e depois interage com os gerenciadores de recurso associados para requisitar a execução de cada tarefa. Para executar uma tarefa, são criados no lado do escalonador e do recurso, simultaneamente, ambientes para providenciar todos os requisitos necessários

para a execução da tarefa no recurso e impedir que esta utilize áreas não permitidas neste, como se estivesse sendo executada no ambiente local do requisitante. Esse ambiente duplicado, por exemplo, redireciona o acesso a arquivos locais para o ambiente original, do lado da máquina requisitante, provendo a uniformidade necessária para a execução em máquinas heterogêneas. Durante a execução, o Condor suporta armazenar o estado da tarefa em alguns *checkpoints* para que esta possa ser restaurada no mesmo ponto de execução caso haja falha ou necessite haver uma migração desta tarefa. Sobre os escalonadores de tarefas podem ser utilizados ainda gerenciadores de tarefas implantados pelo usuário, tais como controladores mestre-escravo e gerenciadores de processos (por exemplo, DagMan [Tea02]).

3.1.2 *Middlewares* Tradicionais

Legion [GW96] é uma arquitetura de software pioneira, projetada com o objetivo ambicioso de prover todas as funcionalidades básicas para o suporte à implementação e à execução de aplicações num ambiente multi-institucional com milhões de computadores. O principal conceito por trás dessa arquitetura é a orientação a objetos: o próprio sistema e as aplicações escritas sobre ele são todos compostos por objetos distribuídos. Uma aplicação pode ter seus objetos dispersos por várias máquinas da grade, conforme a política de distribuição utilizada, a fim de obter uma maior escalabilidade e desempenho de sua execução. Os principais objetivos dessa arquitetura são: autonomia das instituições no uso de suas máquinas; núcleo extensível; escalabilidade; facilidade de uso; computação transparente de localização e acesso; alto-desempenho por meio de paralelismo; espaço único de objetos persistentes; segurança, tanto para o usuário quanto para os recursos; gerenciamento de recursos e exploração da heterogeneidade; suporte a múltiplas linguagens; e tolerância a falhas.

Globus [FK97, Fos05], atualmente em sua versão 4.0 e conhecido como *Globus Toolkit (GT4)*, é um *middleware* e um conjunto de ferramentas de suporte a grade, que implementam protocolos e serviços que permitem a definição de mecanismos de mais alto-nível e aplicações a serem executadas sobre uma grade, composta de recursos de uma ou mais organizações. Ele é baseado no padrão *Web Services* e suporta as linguagens C, Java e Python. A comunicação entre clientes e serviços, camada inferior da arquitetura, é basicamente suportada pela tecnologia *Web Services*, baseada em HTTP/TCP, XML, SOAP e WSDL, com a finalidade de prover um baixo acoplamento entre serviços e universalizar o acesso a estes. A segurança na troca de mensagens, com autenticação e autorização, nessa mesma camada, é baseada no padrão *WS-Security*. Com a finalidade de suportar o acesso e o monitoramento de serviços e o acesso a recursos de dados, com estrutura e estado, são utilizados os padrões *WS-Addressing*, *Web Service Resource Framework (WSRF)* e *WS-*

Notify. Os serviços de monitoramento e busca de recursos da grade, de forma geral, são suportados pelos serviços de consulta e notificação, respectivamente *Index-MDS* e *Trigger-MDS*, componentes do *Monitoring and Discovery System (MDS)*. O serviço *Grid Resource Access and Management (GRAM)* permite que sejam acessados recursos computacionais para controle de tarefas e arquivos utilizados, e monitoramento desses recursos computacionais e tarefas. Existem diversos serviços de acesso a arquivos e dados, tais como *GridFTP*, *Reliable File Transfer (RTF)* e *Replica Location Service (RLS)* para transporte e localização de dados, e *Data Access and Integration (OGSA-DAI)*, que permite acesso a dados relacionais e XML. O serviço de controle de acesso, que autentica e autoriza usuários e processos a usarem recursos da grade, é suportado pelo padrão WS-Security, na camada inferior da infra-estrutura, e também por serviços de manutenção de certificados X.509 e delegação de autorização no *Grid Service Infrastructure (GSI)*. Finalmente, para criação de novos serviços, o *middleware* provê *containers* que facilitam a criação de Web Services com ou sem estado (recursos de dados), com suporte a comunicação, segurança de informação e monitoramento; e bibliotecas de acesso aos demais serviços (e.g. Java CoG (Commodity Grid Kit)).

O Condor-G [FTL+01] é uma evolução do Condor para a execução em grades e utiliza os serviços do Globus para descobrir e acessar recursos e distribuir gerenciadores de recursos entre várias máquinas selecionadas na grade. O escalonador do Condor-G cria um gerenciador de grupo para cada grupo de tarefas. O gerenciador de grupo (ou grade) tem como objetivo selecionar os recursos, dentre os que contêm gerenciadores de recursos distribuídos anteriormente, para executar essas tarefas. Dessa forma o escalonador pode distribuir tarefas para serem executadas pelos gerenciadores de recursos selecionados, da mesma forma que o Condor original. Todo o sistema é tolerante a falhas e utiliza o protocolo *two phase commit* e reexecução para garantir que uma única execução ocorreu.

3.1.3 *Middlewares* Oportunistas

InteGrade [GKGF04] é um *middleware* para grades que propõe reservar recursos disponíveis nos computadores de mesa espalhados pela organização, permitindo ao usuário de cada máquina (*peer*) definir a quantidade máxima de recursos que ele deseja compartilhar quando a sua máquina está sendo usada. O *middleware* emprega um modelo estatístico de previsões de utilização de recursos, permitindo que o escalonador tenha uma estimativa de quais computadores estarão livres em um certo instante e por quanto tempo, reduzindo o esforço de reescalonamento e armazenamento do estado das tarefas em *checkpoints*. Ele utiliza o *middleware* orientado a objetos CORBA para disponibilizar o acesso aos seus serviços e auxiliar na construção de aplicações paralelas. Além disso, ele se baseia numa estrutura hierárquica de organização de grupos de máquinas, com gerenciadores de gru-

pos intermediários que armazenam informações sobre recursos disponíveis dos grupos de máquinas que ele gerencia. Também está prevista a possibilidade de o sistema recuperar aplicações em caso de falhas, por meio de armazenamento do estado das atividades sendo executadas em determinados *checkpoints*.

OurGrid [CBA+06] é um *middleware* para grades com suporte a execução de conjuntos independentes de tarefas (*bag of tasks*) e cujo objetivo é permitir o compartilhamento de recursos computacionais ociosos entre organizações. O sistema é baseado no modelo *peer-to-peer* de forma a ser escalável e de fácil acesso, sendo cada organização participante considerada um par (*peer*) para essa infra-estrutura. Para garantir que haja colaboração entre as organizações, os pares, o *middleware* contém um mecanismo distribuído, conhecido como *Network of Favors*, que prioriza o uso de recursos a quem mais colabora para a grade, de forma semelhante às redes de compartilhamento de dados *peer-to-peer* atuais. O mecanismo compara a colaboração histórica para um determinado recurso, realizada pelas organizações que competem por esse recurso, e permite o acesso a quem mais colaborou para esse. Para garantir a segurança a aplicações e recursos em uma grade em que várias organizações desconhecidas podem se juntar dinamicamente, o sistema utiliza o mecanismo de *sandboxing* denominado SWAN, baseado na máquina virtual Xen [BDF+03], para proteger o recurso contra acessos indevidos, e a replicação de tarefas em organizações distintas para garantir que os resultados não sejam perturbados intencionalmente. Existem dois algoritmos de escalonamento de tarefas: o primeiro distribui e replica tarefas nas máquinas disponíveis, escolhidas aleatoriamente, e o segundo, mais recente, utiliza o critério de afinidade de dados necessários por diferentes tarefas, armazenados em cada máquina, para reduzir a troca de dados entre máquinas.

3.2 Suporte a Processos em Grades Computacionais

3.2.1 Controle de Execução no Cliente

DagMan [Tea02] foi uma das primeiras ferramentas a controlar a execução ordenada de tarefas, mas na porção cliente do sistema. Ela é empregada no sistema Condor [TTL02] de suporte a execução de tarefas em grades computacionais. O objetivo dela é funcionar como um comando *make* do Unix, mas para a execução de tarefas, em cujo *script* são especificadas as dependências entre tarefas ao invés de regras. As tarefas suportadas são limitadas a execução de processos Unix remotos e os dados são comunicados por meio de arquivos, que devem ser explicitamente copiados entre as máquinas executantes. Além disso, os processos workflow criados mediante a especificação dessas dependências são concretos, nos quais é necessário definir previamente a máquina onde cada tarefa será executada. Esta ferramenta não suporta os conceitos de sessão e monitoramento

de processos. Se alguma tarefa retornar erro, a ferramenta retorna um arquivo com a especificação do processo que resta executar. Desta forma o usuário pode corrigir a tarefa e executar o restante do processo. Essa recuperação é bastante primária, depende da intervenção do usuário e não inclui a recuperação no caso de haver problemas na máquina cliente.

GridAnt [vLAH+04] é uma outra ferramenta para a execução de processos workflow na máquina cliente, por meio do controle e monitoramento de tarefas na grade computacional. Essa ferramenta é utilizada no Globus e estende a ferramenta Ant [Pro07], usada na automação de processos de geração de *softwares*, similar ao comando *make* do Unix. Entre as extensões ao Ant está a manipulação de variáveis internas ao processo e a inclusão de regras específicas para a execução de tarefas remotas e para a transferência de arquivos entre máquinas de uma grade. Existem várias deficiências nesse projeto: o processo workflow definido é concreto, não abstrato, atribuindo-se logo na especificação as máquinas que devem executar as tarefas; a execução é centralizada e realizada na máquina cliente, impedindo um melhor desempenho e tolerância a falhas na execução de processos; os construtores de fluxo são inadequados para a especificação de workflows expressivos; e os processos resultantes são difíceis de se monitorar.

3.2.2 Suporte a *Workflow* em Grades de Dados

Pegasus [DBG+04] é um sistema para mapeamento e execução de processos *workflow* científicos sobre grades, utilizados em aplicações intensivas em dados. Ele é disponibilizado juntamente com o *GriPhyN Virtual Data System*, sendo uma extensão do sistema de representação e derivação automatizada de dados virtuais *Chimera* [FVWZ02]. O sistema proposto utiliza como entrada uma especificação de *workflow* abstrato representada por dependências de dados. Essa especificação contém os dados, como nós fontes, e as transformações encadeadas, como nós intermediários, necessárias para se obter um resultado desejado. O sistema Pegasus mapeia esse *workflow* abstrato para os recursos disponíveis numa infra-estrutura de grade, utilizando os serviços do sistema Globus, gerando assim um workflow concreto. O workflow concreto é executado sobre a grade utilizando o sistema de suporte a execução de *workflow* DagMan/Condor [Tea02], apresentado anteriormente.

3.2.3 Coordenação Centralizada de Processos

Symphony [LK02] é um arcabouço (*framework*) para composição e manipulação visual de meta-programas e execução desses sobre recursos computacionais de uma grade. Os componentes dos meta-programas são programas e arquivos pré-existentes, representados por componentes *JavaBeans*, que são compostos e ligados visualmente por arcos, que representam fluxos de dados entre os componentes. Entre os atributos dos componentes

do tipo programa estão a sua localização, o recurso a ser utilizado para sua execução e os parâmetros de invocação. O arcabouço permite a utilização do serviço GRAM (*Grid Resource Allocation and Management*) do Globus para alocar recursos e executar programas. A sua principal contribuição é o desenvolvimento de um arcabouço mínimo para a composição e execução de meta-programas em grades computacionais.

WebFlow [AFFH98] é um ambiente utilizado para especificar e executar facilmente meta-programas. Os meta-programas são especificados por meio da construção e interligação de módulos, que representam tarefas, de acordo com o fluxo de dados entre eles. A arquitetura é composta por três camadas. A primeira camada é o *front-end*, na qual o meta-programa, contendo módulos e conectores de dados, é representado e construído por meio de um *applet* Java executado no navegador do cliente. A camada intermediária é definida sobre o *middleware* CORBA por meio de serviços providos pelo WebFlow, tais como serviços de controle de sessão e comunicação, e módulos do meta-programa. Cada módulo deve ser implementado pelo usuário, especificando-se os métodos *initialize*, *run* e *destroy*, utilizados, respectivamente, para inicializar o módulo e conectá-lo ao arcabouço; executá-lo; e destruí-lo. Por outro lado, os serviços oferecidos pelo WebFlow são utilizados para interligar os módulos e gerenciar sua execução. Por fim, o *back-end* é composto de repositórios e computadores da grade, suportados pelo toolkit de meta-computação Globus. Nessa camada, os serviços do Globus são utilizados para descobrir recursos (*Monitoring and Discovery System - MDS*), para alocar recursos (*Grid Resource Allocation and Management - GRAM*), autenticar o cliente e autorizar o acesso a recursos (*Generic Security Service - GSS*) e fornecer o acesso global aos repositórios persistentes de dados (*Global Access to Secondary Storage - GASS*).

3.2.4 Suporte a Infra-estruturas *Peer-to-Peer*

SwinDew [YYR03] suporta a instanciação e execução de processos *workflow* em redes *peer-to-peer*. A instanciação é feita percorrendo-se as tarefas do processo na ordem de execução e buscando-se os nós mais disponíveis para executar cada tarefa. Em detalhes, o nó com uma tarefa busca os melhores nós para executar as tarefas sucessoras, considerando que determinada tarefa pode ter sido instanciada anteriormente, retornando, nesse caso, o nó em que ela se encontra. Na fase de execução, cada nó somente executa a tarefa atribuída a ele depois que uma ou mais tarefas predecessoras tenham sido executadas, segundo o tipo de junção especificada, e se as condições atribuídas às dependências de controle ativadas tenham sido avaliadas como verdadeira. Dessa forma, a execução é totalmente distribuída, não tendo nenhuma máquina de *workflow* ou banco de dados centralizado como gargalo ou ponto único de falhas. As desvantagens desse sistema são claras, entretanto: o escalonamento é feito somente no início, desconsiderando-se falhas

em nós, problemas de disponibilidade e balanceamento de carga em ambientes totalmente dinâmicos, como as grades; para cada tarefa a seleção de um nó é realizada consultando-se todos os nós do grupo, e somente deste grupo, limitando a escalabilidade do sistema; e a linguagem de especificação não é expressiva por não considerar controles iterativos, por exemplo.

Triana [CGH+05, TSWR03] é um ambiente para a programação de aplicações científicas, baseado nos conceitos de composição, coreografia e distribuição de processos *workflow*. Ele suporta a execução de processos *workflow* sobre os *middlewares peer-to-peer* P2PS e JXTA, e arquiteturas baseadas em *web services*, por meio de uma camada própria com funcionalidades de publicação, de busca de serviços e de comunicação, denominada GAP (*Grid Application Prototype Interface*). Um processo *workflow* é composto de tarefas e ligações tipadas entre essas tarefas e cada tarefa pode se referir a um serviço disponível em uma rede *peer-to-peer* ou grade. As ligações entre tarefas são feitas por meio de suas portas de entrada e de saída. Construtores de controle condicional e de controle iterativo devem ser implementados pelo programador do processo, pois o modelo de especificação não provê suporte direto a eles; tarefas devem ser implementadas para realizar conversões de dados no processo. Para o controle de execução de tarefas paralelas e sua distribuição, o programador deve implementar uma tarefa especial chamada de "tarefa de controle", que, entre outras funções deve: distribuir as mensagens de dados e controle aos serviços descobertos em tempo de execução; conectar os canais de comunicação às portas de entrada e de saída dos serviços descobertos; e sincronizar as mensagens recebidas dos serviços para depois enviar os dados resultantes por sua porta de saída quando algumas ou todas as tarefas que ele estiver controlando terminarem. Há a possibilidade de se compor partes de um processo dentro de outro processo para realizar uma distribuição hierárquica limitada de processos. As desvantagens dessa abordagem são: necessidade do programador desenvolver todo o fluxo de controle por meio de tarefas, utilizando uma ferramenta visual somente depois para ligar as tarefas; e, por ser baseado em mensagens, de forma similar a petri-nets, o monitoramento, recuperação e modificação de processos em tempo de execução são complicados, conforme foi discutido anteriormente.

3.2.5 Recuperação de Processos

Opera-G [Bau04] é um *middleware* para execução de processos *workflow* sobre grades computacionais que provê a execução paralela, distribuída e confiável de processos e mecanismos de controle de execução de processos com suporte a visualização e modificação da estrutura e estado desses processos. A linguagem de especificação de processos utilizada, OCR-G, é bastante simplista, contendo somente os conceitos de tarefa, subprocesso e guarda condicional para a definição de fluxos de controle. A estrutura dos processos,

o estado de suas instâncias, o estado dos recursos computacionais e o método de acesso a programas externos são armazenados em um banco de dados relacional, utilizado para o controle da execução dos processos. O *middleware* é composto de módulos de escalonamento, acesso a dados, distribuição e controle de execução de tarefas, e controle de recursos disponíveis, mas de forma monolítica e de difícil distribuição. Ele não provê escalonamento distribuído de forma automática, tornando necessária a intervenção manual para a definição de subprocessos e para ativação de outros sistemas escravos.

3.2.6 Escalonamento Otimizado e Distribuído

GridFlow [CJSN03] é um dos *middlewares* pioneiros de execução e controle de processos *workflow* sobre grades. Ele tem como característica principal a simulação, execução e monitoramento de processos em dois níveis: global e local. Um *workflow* global é dividido em subworkflows a serem executados em redes locais ou aglomerados de computadores. Por sua vez, um *subworkflow* é dividido em tarefas a serem executadas em cada máquina. Paralelamente, o *middleware* propõe gerenciadores de recursos distintos para o nível global e para o local, responsáveis por prever o início e o tempo de execução de cada *subworkflow* ou tarefa, respectivamente. Em nível local, o gerenciador de recursos, Titan, emprega heurísticas com lógica *fuzzy* para prever o tempo de início e fim de cada tarefa, considerando conflitos de alocação entre tarefas de diferentes *workflows* e usando o serviço PACE de predição de tempo de execução de tarefas. Com base em simulações de tempo de execução de cada *subworkflow*, o gerenciador de recursos ARMS pode decidir, então, qual é a melhor rede para executar cada um dos *subworkflows* de um *workflow*. A desvantagem desse modelo é a subdivisão de *workflows* somente em dois níveis e a impossibilidade de responder à variação das disponibilidades dos recursos após a simulação prévia da execução de um *workflow*, por falta de adaptabilidade do mecanismo de escalonamento.

3.2.7 Escalonamento e Execução em *Pipeline*

ICENI [MYA+04] é um *middleware* para a execução de processos *workflow* em grades computacionais com escalonamento e execução de tarefas de processos realizados em *pipeline*. O *workflow* é abstrato e descrito em XML, sendo constituído de componentes e ligações de dados entre eles. Cada componente descreve uma tarefa a ser executada no contexto do *workflow* e é composto de significado, comportamento e implementações. O significado descreve o objetivo do componente e os dados necessários para a sua execução; o comportamento descreve o fluxo de controle interno; e as implementações, que pode haver mais de uma para um comportamento, contêm o código para realizar o comportamento com os dados descritos. Para executar um *workflow*, o ICENI obtém a especificação

deste e depois, em seqüência para cada componente, escalona-o, distribui-o ao recurso selecionado, e executa-o. O escalonamento utiliza a melhor combinação de implementação e recurso que a heurística em uso fornecer para cada componente, utilizando um banco de dados de desempenho de implementações em recursos e também reservas avançadas, quando possível. A distribuição dos componentes é feita de acordo com o escalonamento realizado. Depois de distribuído, cada serviço de execução associado a um recurso e que contém um componente do *workflow* é responsável por descobrir os demais serviços de execução, associados a outros recursos, e que contêm os demais componentes do *workflow*. Após um serviço de execução com um componente do *workflow* descobrir os serviços adjacentes, esse também é responsável por: obter dados de entrada necessários dos serviços anteriores; executar o componente (tarefa) na ordem prevista; e enviar os resultados para o próximo serviço. Esse *middleware* realiza cada um desses estágios da execução do *workflow* em momentos distintos para cada componente, numa forma de linha de produção (*pipeline*), escalonando, distribuindo e executando componentes independentemente, assim que os pré-requisitos forem sendo satisfeitos. As técnicas de escalonamento atrasado e reserva adiantada permitem que se tenha um melhor conhecimento do estado da grade no momento da seleção dos recursos para um componente e se garanta um tempo de execução mínimo, respectivamente. Esse sistema tem como deficiências a falta de um mecanismo de especificação de fluxo de controle e paralelismo, além da dificuldade da adaptação à variação da disponibilidade de recursos e ausência de mecanismos de tolerância a falhas [MLD05].

3.3 Resumo

Nesse capítulo apresentamos os principais *middlewares* para grades computacionais, classificados em *middlewares* para execução de conjuntos de tarefas independentes (*bag of tasks*) e *middlewares* para execução de processos *workflow*. Os primeiros são mais básicos e utilizados como fundação para a construção de outros serviços e aplicações mais completos, os segundos, originados recentemente, por sua vez, contêm um modelo de programação mais expressivo, baseado em processos, e fornecem desde escalonamentos otimizados para desempenho e utilização de recursos, distribuição completa de execução de aplicações em redes *peer-to-peer*, até controle e recuperação automática de aplicações em caso de falhas.

Condor [TTL02] foi o primeiro *middleware* projetado para escalonar e controlar a execução de tarefas em máquinas distribuídas, com suporte a recuperação de tarefas baseado em *checkpoints*, mas operava em uma rede local, sendo precursor das grades computacionais. Legion [GW96] foi o primeiro *middleware* que se propôs a execução de aplicações, construídas com base em objetos distribuídos, sobre milhões de computadores, pertencen-

tes à várias instituições independentes. Ele visava uma maior escalabilidade e desempenho de aplicações por meio da utilização de escalonamento de objetos distribuídos em máquinas disponíveis e execução paralela. Globus [FK97, Fos05] foi o primeiro *middleware* que surgiu com a finalidade de permitir a criação de aplicações que utilizassem todos os recursos disponíveis de uma grade computacional, sendo composto de vários serviços básicos para localização e monitoramento de recursos, acesso a máquinas e controle de execução de tarefas, transferência de dados e segurança. Condor-G [FTL+01] é uma evolução do Condor para grade, que escalona, distribui e controla a execução de tarefas pela grade, utilizando o protocolo *two-phase commit* e reexecução para garantir a recuperação de tarefas. Integrade [GKGF04] avança na possibilidade de o usuário especificar qual é o poder de processamento que ele deseja compartilhar de suas máquinas, e na maior previsibilidade da utilização dos recursos, permitindo um escalonamento mais eficiente devido a essas informações. OurGrid [CBA+06] se destaca por ser baseado em uma rede *peer-to-peer* de compartilhamento de máquinas entre instituições e por ter um mecanismo de crédito no uso de recursos, conhecido como *Network of Favors*, que permite a usuários que colaboram mais com a grade, utilizar mais os recursos desta. A segurança é baseada na técnica de *sandboxing* por meio da máquina virtual Xen.

Na classe de *middlewares* que oferecem suporte a processos, DagMan [Tea02] e GridAnt [vLAH+04] são ferramentas básicas, executadas na máquina cliente da grade, que simplesmente controlam a ordem da execução de tarefas independentes sobre os recursos da grade. A troca de dados é realizada pela cópia de arquivos entre máquinas, os processos são concretos, no qual as máquinas executantes são previamente definidas, e, por executarem na porção cliente, limitam o desempenho e a confiabilidade das aplicações. Pegasus [DBG+04] permite a criação de processos abstratos, no qual máquinas com melhores características para cada tarefa são descobertas e então as tarefas são distribuídas. Entretanto ele é projetado somente para execução de *workflows* de transformação de dados. Symphony [LK02] e WebFlow [AFFH98] são os primeiros *middlewares* projetados para se construir processos genéricos, chamados de meta-programas, de forma que esses sejam executados em uma máquina da grade responsável por localizar os melhores recursos para as tarefas, distribuí-las e controlar a sua execução. Suas linguagens contém simplesmente tarefas e dependências entre essas. SwinDew [YYR03] e Triana [CGH+05, TSWR03] são os primeiros a executar processos de forma distribuída em redes *peer-to-peer*. As tarefas são distribuídas para máquinas da grade e cada uma conhece suas predecessoras e sucessoras, e, em tempo de execução, cada tarefa envia mensagens para suas sucessoras com os resultados desta. A principal vantagem é a distribuição total da execução e as deficiências são a dificuldade de monitoramento e adaptação dinâmica. Opera-G [Bau04] é um dos primeiros a oferecer recuperação automática de processos, por meio do armazenamento do estado de execução e dos dados em um base de dados relacional, mas sua linguagem

é simplista, não permitindo a criação de processos estruturados e iterações, e a máquina de execução é centralizada. GridFlow [CJSN03] tem como principais vantagens a execução de processos em dois níveis, global e local, compreendendo mais de uma instituição, e escalonamento otimizado por meio de estimativas de tempo de execução de tarefas e subprocessos e seleção dos melhores recursos disponíveis para a execução dessas tarefas e subprocessos. Entretanto ele não admite adaptação dinâmica e é limitado a dois níveis. ICENI [MYA+04] tem como principais características a reserva antecipada de recursos, que permite que máquinas sejam alocadas previamente para as tarefas, e execução em forma de *pipeline*, na qual as tarefas são concorrentemente escalonadas, distribuídas e executadas. Isso permite um escalonamento mais preciso, sem interferência no tempo de execução processo. A desvantagem é a falta de adaptação dinâmica às condições de disponibilidade dos recursos.

Capítulo 4

Especificação de Processos Estruturados

Visando a sanar algumas deficiências dos *middlewares* para grades computacionais com suporte a processos *workflow* existentes e possibilitar que a grade seja efetivamente utilizada, foi desenvolvido o sistema Xavantes. Ele é composto de dois componentes principais: uma linguagem de especificação de processos estruturados e um *middleware* cujos serviços apóiam a execução e o monitoramento de processos estruturados em grupos de máquinas distribuídas. Nesse capítulo descreveremos a linguagem de especificação de processos do sistema Xavantes e no próximo, o seu *middleware*.

4.1 Descrição da Linguagem de Especificação

A linguagem de especificação do Xavantes foi criada para permitir ao usuário do sistema o rápido desenvolvimento de aplicações baseadas em processos do tipo *workflow*, mas estruturados. Esses processos estruturados são compostos por construtores intermediários, para controlar o fluxo de controle e de dados dos processos; e de atividades, nas folhas, existentes ou criadas pela aplicação, para executar as tarefas básicas dos processos. Ademais, a linguagem permite a especificação de parâmetros para definir o tipo e a qualidade dos recursos que essas atividades necessitam para serem executadas; prever o custo de execução das atividades, determinado pelo tamanho das mensagens e do processamento requerido; tratar exceções levantadas pelas aplicações; e controlar o paralelismo entre os elementos do processo.

A linguagem do Xavantes, denominada *Xavantes Structured Process Language* (XSPL) é similar à linguagem WS-BPEL [TC05], proposta para executar processos de negócios no ambiente de serviços *web*, mas é bem mais simples de ser utilizada e adequada para grades computacionais. Ambas descrevem processos usando o padrão XML [BP04] de

estruturação de documentos, mas XSPL simplifica a especificação de processos por ser estritamente síncrona, minimalista e estruturada. Ela também é mais poderosa para construir aplicações paralelas, por possuir um maior número de construtores de controles paralelos e estruturados. Ademais, diferentemente do WS-BPEL, ela é projetada para promover a distribuição de controle e tarefas e não somente para compor e coordenar tarefas e serviços pré-existentes.

Os princípios que nortearam o seu desenvolvimento foram:

- simplicidade de especificação, depuração, monitoramento e modificação de processos *workflow* por meio do uso de poucos tipos de construtores estruturados simples e expressivos que lembram construtores de linguagens genéricas de programação, como Java;
- construção de processos *workflow* complexos por meio da composição aninhada de construtores de fluxo seqüenciais e paralelos que podem compartilhar variáveis comuns e que podem tratar exceções advindas de construtores inferiores;
- necessidade de disponibilização de informações de dependência e agregação de controles em seu meta-modelo de modo a permitir que um serviço de escalonamento e execução em grade interprete processos e facilmente distribua hierarquicamente o controle de processos, caso haja necessidade;
- simplicidade do meta-modelo da linguagem para que instâncias de processos possam ser manipuladas e modificadas dinamicamente, tanto no seu estado quanto na sua estrutura, de especial importância para tratar os casos em que ocorram exceções não previstas.

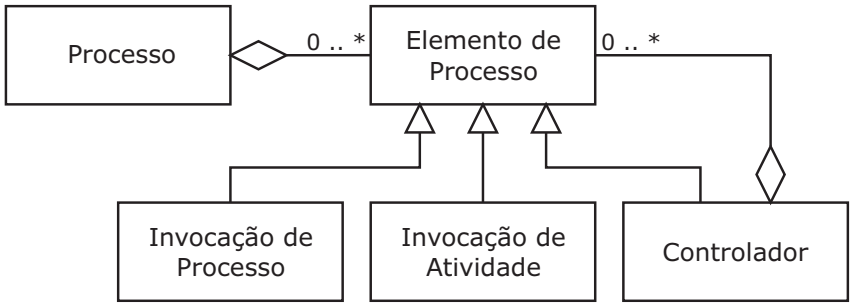

Figura 4.1: Diagrama de Classes do Modelo de Programação.

Em XSPL, uma aplicação é especificada por meio de processos e atividades. A estrutura de uma aplicação é definida como uma árvore, iniciada com um processo raiz que

invoca outros processos, e assim sucessivamente, até a invocação das atividades, folhas dessa árvore. Como pode ser observado na Figura 4.1, que especifica o diagrama de classes do modelo de programação, em notação UML [BRJ99], um processo é composto por elementos de processo, que podem ser invocações a atividades, invocações a processos, ou controladores. Controladores, por sua vez, também podem possuir esses elementos de processo, inclusive outros controladores aninhados.

4.1.1 Processos

Processos são utilizados para compor e coordenar atividades e outros processos, invocados por esse na ordem definida pelos controladores, a fim de resolver um certo problema complexo, interagindo com o meio e retornando resultados. Nesse contexto, processos e atividades, especificados para a aplicação ou pré-existentes, são utilizados para resolver partes menores do problema da aplicação. Controladores, nós intermediários dessa estrutura hierárquica de especificação, são utilizados para controlar a ordem de execução e o fluxo de dados entre os elementos que ele contém. Atividades são os nós mais básicos da estrutura e são utilizadas para realizar tarefas atômicas, sob a ótica do *middleware* de suporte. Atividades podem ser definidas por código Java [GJSB05], ou invocação a objetos remotos e serviços *web* pré-existentes.

4.1.2 Atividades

As atividades realizam as tarefas básicas de uma aplicação especificada como processo *workflow*. Elas são utilizadas principalmente para realizar computações intensivas, transformações de dados ou são *wrappers* para invocar serviços remotos. Todo o restante do processo especifica em que ordem essas atividades são invocadas, quais são os dados recebidos por elas e onde são utilizados os dados retornados por elas. O corpo da atividade deve ser codificado em Java 6.0 [GJSB05]. As atividades são compiladas para que haja um melhor desempenho em sua execução. Atividades podem ser utilizadas, por exemplo, para realizar a multiplicação de matrizes, renderização de partes de gráficos 3D e cruzamento de dados obtidos de bases distribuídas.

4.1.3 Controladores

Os controladores especificam a ordem de execução dos elementos de um processo. Um controlador é responsável por determinar quando e com que dados os seus elementos imediatamente controlados serão executados, sejam eles invocações a processos, a atividades ou outros controladores. Eles podem, portanto, coordenar a execução de outros controladores e a invocação de processos e atividades, promovendo uma especificação estruturada

e hierárquica de fluxo de controle. Eles são síncronos no sentido que sua execução somente é terminada quando as execuções de todos os seus elementos filhos são terminadas.

De forma a estender os construtores de controle das linguagens de programação genéricas para incluir o tratamento simples do paralelismo, a linguagem XSPL contém todos os tipos básicos de controladores seqüenciais e paralelos, que permitem, por simples composição, a descrição de qualquer forma de controle síncrono e paralelo. Os controladores podem ser classificados quanto ao tipo fluxo de controle especificado e ao paralelismo permitido. A Tabela 4.1 apresenta tal classificação cruzada. Em função do tipo de fluxo de controle especificado, um controlador pode ser classificado em simples, condicional, iterativo ou iterativo-condicional. Em função do paralelismo permitido, um controlador pode ser classificado em seqüencial ou paralelo.

Tabela 4.1: Classificação dos controladores em função do tipo do fluxo de controle e do paralelismo permitido.

tipo / paralelismo	**simples**	**condicional**	**iterativo**	**iterativo condicional**
seqüencial	block	switch	for(each)	while
paralelo	par	parswitch	parfor(each)	parwhile

Os controladores seqüenciais são block, switch, for, foreach e while. O controlador block é seqüencial simples, que executa cada elemento após o término do anterior. O switch é condicional e executa o primeiro elemento cuja condição associada for verdadeira. O for é um controlador iterativo, que executa os elementos filhos um determinado número de vezes, seqüencialmente, uma iteração após a outra, e mantém um contador dessas iterações. O foreach é uma especialização do for utilizado para simplificar a iteração sobre objetos de uma coleção. Ele executa o mesmo conjunto de ações sobre cada objeto da coleção, uma iteração após a outra, referenciando um objeto por iteração. O while, por sua vez, é iterativo-condicional, e executa o mesmo conjunto de ações, uma vez após a outra, enquanto a condição associada a ele for satisfeita. Todos esses controladores têm o comportamento similar aos construtores de fluxo de controle correspondentes das linguagens estruturadas de programação.

Paralelismo estruturado Os controladores paralelos reproduzem o comportamento dos respectivos controladores seqüenciais, mas de forma que vários elementos de processo possam ser executados concorrentemente. O controlador par é o paralelo simples, que executa todos os seus elementos em paralelo. O parswitch é o correspondente condicional, que executa concorrentemente todos os elementos cuja condição associada for verdadeira. O controlador parfor é o paralelo iterativo, que dispara a execução simultânea de um certo

número de caminhos (*threads*), cada um executando o mesmo conjunto de elementos e mantendo o valor do contador do disparo. O parforeach é a especialização do parfor, que executa um certo número de caminhos em paralelo, um para cada elemento da coleção passada. O parwhile é o paralelo iterativo-condicional. Ele dispara caminhos de execução em paralelo enquanto a condição associada for verdadeira. A cada disparo ele verifica se será disparado o próximo caminho. Todos esses controladores paralelos iterativos, por continuarem disparando a execução de caminhos paralelos de acordo com o estado interno atual do processo, eles provêem a flexibilidade de determinar em tempo de execução o número de caminhos a serem executados concorrentemente.

Sincronização flexível Todos os controladores paralelos disparam a execução de um ou mais elementos de processo e os esperam terminar. Efetivamente, esses controladores contêm um disparador (*fork*) e uma junção (*join*) de execução pareados, o que permite a estruturação hierárquica e síncrona do fluxo de controle dos processos. De forma a implementar uma junção condicional, além do disparo condicional, todos os controladores paralelos contêm um condição de saída, avaliada todas as vezes que uma execução termina. Essa condição especifica quando um controlador deve terminar, abortando todos os elementos ainda em execução.

4.2 Características da Linguagem de Especificação

A linguagem XSPL, por meio da descrição de processos, controladores e atividades, permite especificar, de forma simples, aplicações que são estruturadas, paralelas e síncronas e que podem ser facilmente particionadas e distribuídas pelo *middleware* Xavantes. As principais características e vantagens dessa linguagem são apresentadas nessa seção.

Fluxo de dados Para uma efetiva composição de atividades é necessário haver formas de comunicação de dados entre elas. No XSPL, os dados são trocados entre processos, atividades e controladores por meio de parâmetros de entrada e saída e por meio de objetos compartilhados. Os dados trocados são representados por objetos Java, classificados em locais e remotos. Objetos locais são criados na mesma máquina virtual Java (*Java Virtual Machine*) [LY99], doravante denominado ambiente, que o elemento que o criou se encontra. Eles são de uso exclusivo do elemento e são transmitidos por valor toda vez que são passados por parâmetros, o que produz uma cópia no ambiente destino do objeto exclusivo do ambiente fonte. Objetos remotos, por sua vez, são criados em um determinado ambiente e são geralmente compartilhados por vários elementos. Eles são transmitidos por referência quando passados pelos parâmetros dos elementos. Isso permite

que os elementos que recebem essa referência invoquem remotamente os métodos disponibilizados do objeto remoto e troquem dados entre eles. Essas invocações remotas são realizadas de forma transparente por meio do protocolo *Java Remote Method Invocation (RMI)* [Mic04]. Dados externos ao processo, compartilhados por mais de um processo ou aplicação, também podem ser utilizados. Eles são acessados por meio de fontes de dados, tais como banco de dados ou objetos distribuídos.

Todos os elementos de processo contêm a declaração de parâmetros formais de entrada e de saída. Os parâmetros de entrada são utilizados para receber objetos de seus pais (invocadores), por valor ou referência, antes da execução do elemento. Os parâmetros de saída são utilizados para retornar objetos para seus pais (invocadores), por valor ou referência, contendo os resultados da execução do elemento. Todos os parâmetros, sejam de entrada ou saída, têm um nome específico, que o identifica, e um tipo associado, que restringe a atribuição de valores (ou referências) a esse parâmetro a somente objetos do tipo determinado. A declaração de parâmetros formais de entrada e saída na especificação dos elementos permite deixar explícito, na invocação, quais são os objetos trocados entre os invocadores e seus elementos por meio da passagem dos parâmetros atuais. Dessa forma, a passagem de dados entre elementos de um processo por meio de parâmetros é sempre realizada por intermédio de controladores (o tipo de elemento invocador). A vantagem é que os controladores podem determinar quais são as transformações necessárias sobre objetos do tipo valor e qual é o fluxo de objetos na comunicação de dados entre os elementos.

Todos os processos e controladores podem conter a declaração de variáveis internas, também com nome associado e tipo de objetos aceitos, assim como os parâmetros. Essas variáveis aceitam objetos por valor ou referência. Da mesma forma que nos parâmetros, quem determina se uma variável contém um objeto valor ou uma referência ao objeto é o tipo do objeto: se objeto local, a variável armazena por valor; se objeto remoto, armazena por referência. As variáveis internas são utilizadas para armazenar objetos para serem passados entre os elementos inferiores (invocados) ou para serem utilizados em expressões e *scripts* internos, sejam de computação de dados relevantes ou de controle de fluxo.

Objetos compartilhados Objetos remotos podem ser criados pelos controladores e passados por referência para os seus elementos, para que eles acessem um mesmo objeto compartilhado e troquem informações por meio deste. As vantagens da utilização deste objeto compartilhado são: acesso a um objeto remoto compartilhado somente no escopo do controlador, ou seja, pelos elementos do controlador e seus descendentes, o que permite replicar somente os dados necessários a esses e limitar o acesso concorrente, aumentando a disponibilidade da aplicação; e proximidade entre os elementos do controlador e a variável compartilhada, que se encontra no mesmo ambiente do controlador, o que reduz o tempo

de comunicação e aumenta o desempenho da aplicação.

Scriptlets A linguagem XSPL permite a inserção de comandos e expressões descritos em uma linguagem *script* baseada em Java, denominada BeanShell [Nie07], dentro de processos e controladores em blocos pré-determinados com funções específicas, com o objetivo de se manter tão expressiva quanto uma linguagem de programação genérica e mesmo assim não perder a sua simplicidade. Esses *scriptlets* são muito úteis para especificar, de modo detalhado, transformações e manipulações de dados, e condições de execução de elementos de processos. Na prática, os *scriptlets* são usados para: declarar expressões numéricas e booleanas para especificar condições nos controladores; converter valores entre tipos de dados; preencher estruturas de dados; manipular coleções e outros objetos; e para utilizar funções matemáticas e invocar métodos quaisquer das bibliotecas Java referenciadas pelo processo e da *Java Runtime Class Library*. Esse tipo de abordagem facilita muito a integração de atividades e serviços pré-existentes, pois permite a manipulação de dados entre esses elementos no próprio processo.

Reflexão Computacional Todos os elementos de processo e os caminhos paralelos de execução (*threads*) de controladores paralelos são identificados. A linguagem permite que os *scriptlets* sejam utilizados para consultar o estado de cada elemento e caminho de execução e atuar sobre esses, como se fossem objetos de dados. Esse é um tipo de reflexão computacional sobre o comportamento do processo. Ela torna possível, por exemplo, um código de tratamento de exceção em um controlador abortar um determinado elemento inferior, ou um código de junção consultar quantos caminhos de execução já terminaram no escopo do controlador, promovendo um grande controle sobre a execução dos processos.

Especificação de recursos A linguagem permite a descrição, em cada atividade, de requisitos mínimos necessários para a execução da atividade. O objetivo é permitir que o escalonador somente selecione os recursos que satisfaçam os requisitos determinados para executar a atividade. Entre os requisitos necessários que podem ser declarados estão: existência de arquivos específicos, acesso a fontes de dados específicas, quantidade mínima de memória volátil, tamanho mínimo de memória persistente e poder de processamento mínimo.

Estimativa de uso de recursos Por meio da linguagem XSPL é permitido declarar a estimativa de uso de recursos por elemento de processo dentro da especificação desse elemento. Para os processos, controladores e atividades é possível declarar o uso estimado da rede para a troca de dados, destes com o seu controlador pai, em *kilobytes*, seja por meio de parâmetros, variáveis compartilhadas ou acesso a bancos de dados externos.

Para as atividades é possível declarar o uso estimado do esforço de processamento de uma máquina, em milhares de instruções básicas, para executar cada atividade. Também é possível declarar o tempo estimado em que um elemento pode ficar ocioso, nem processando, nem transmitindo ou recebendo dados, mas esperando por uma resposta externa. Dadas essas estimativas de esforço de processamento e de troca de dados, juntamente com a estimativa de tempo ocioso, o sistema pode calcular o tempo total de execução de um certo elemento, dado o meio de comunicação e a máquina alocada para executar esse elemento. Com isso, um escalonador pode selecionar o melhor recurso para executar cada elemento, alocando-o próximo ao controlador pai, se há um grande potencial de troca de dados com este e com os elementos do mesmo nível, ou em uma máquina de alto desempenho, mas distante, para o caso de um elemento intensivo em computação.

Temporização A junção dos controladores paralelos podem conter um atributo para a definição de um período de espera, em milissegundos. Assim, toda vez que o tempo especificado for transcorrido, o código contido na junção é executado, permitindo que o controlador continue esperando pela finalização dos elementos; seja terminado, abortando a execução de todos os elementos sendo executados; ou seja executada uma ação específica para tratar desse evento.

Nos controladores paralelos iterativos é possível especificar o tempo de espera em milissegundos entre uma iteração e a próxima de disparo de execução paralela de elementos. Essa funcionalidade é útil para continuar disparando periodicamente a execução de uma determinada atividade enquanto os resultados desejados não forem obtidos. Um exemplo de uso dessa funcionalidade é a especificação de algoritmos não-determinísticos do tipo *Las Vegas*, que podem executar um certo número de atividades simultâneas enquanto os resultados da computação não estiverem com a qualidade desejada.

Tolerância a falhas Infelizmente os processos nem sempre são executados da maneira prevista, ou por falhas de sistema, ou por falhas de usuário na interação ou na especificação da aplicação, que podem produzir, por exemplo, erros de divisão por zero, acesso fora dos limites de vetores e abertura de arquivos inexistentes. As falhas de sistema, ocorridas nos recursos ou nas camadas inferiores de *software*, devem ser tratadas, na maioria das vezes, de forma transparente para o processo; os erros gerados por falhas de interação, por sua vez, devem ser tratados por código específico de tratamento de erro ou por atividades alternativas; e as falhas de especificação ou implementação, como se tratam de falhas no próprio código da aplicação, devem ser tratadas por códigos alternativos ou modificação dinâmica do estado ou da estrutura do processo, enquanto esse estiver em execução. Caso uma falha não seja tratada, ela pode se manifestar como um defeito na aplicação, a ser corrigido pelo usuário pela correção da interação ou da especificação da aplicação, ou pela

interrupção da aplicação. Como as aplicações são processos de longa duração, deve-se evitar a sua interrupção, de forma a torná-la confiável e disponível.

A linguagem permite a especificação de vários modos de tratamento de falhas descritos na Seção 2.4.3. Em nível de atividade, as falhas de sistemas podem ser tratadas por retentativas de execução de uma atividade em diversos recursos distintos, onde o número de tentativas, descrita em um atributo da invocação da atividade, é fixa, infinita ou determinada por tempo. As atividades podem ser declaradas como recuperáveis e assim o sistema pode tentar restaurar a sua execução a partir do último *checkpoint* gravado, no mesmo recurso computacional ou em outro que acesse a mesma base onde foi armazenado o estado das variáveis internas da atividade. Entretanto, é responsabilidade do usuário escrever o código para executar a atividade do ponto salvo após a recuperação. Há ainda um atributo que permite especificar que a atividade pode ser replicada, e um atributo, especificado na invocação da atividade, que determina o número de réplicas a serem criadas.

Em nível de processo, o tratamento de falhas de sistema e de aplicação é realizado por meio do tratamento das exceções de sistema ou de aplicação originadas a partir destas falhas, respectivamente. Pode-se, com os mecanismos de tratamento de exceções contidos na linguagem, muito semelhante aos da linguagem Java, executar ações específicas para responder a cada tipo de exceção. Detalhadamente, todos os controladores e processos têm blocos para que o usuário especifique como devem ser tratadas exceções geradas por falhas específicas, não mascaradas pelo sistema ou ocorridas no código da aplicação. O corpo do controlador especifica o escopo em que as exceções serão tratadas pelo controlador; em seqüência há blocos catch, nos quais se especificam quais são os tipos de exceções a serem tratadas pelo controlador e como elas devem ser tratadas; e no final há um bloco finally, cujo corpo é sempre executado, tendo havido ou não exceção na execução dos elementos contidos no controlador, usado geralmente para liberar recursos. Caso não seja possível tratar a falha no bloco catch correspondente, o usuário deve relançar a exceção para o controlador superior. Para esse caso existem duas possibilidades distintas: a ação pode ser simplesmente relançar a exceção sem o aborto da execução do controlador, para posterior continuação; ou pode ser a compensação da execução do controlador, com o aborto da execução do controlador. Assim, com o uso dos mecanismos de tratamento de exceções, dos processos e controladores, é possível: ignorar uma atividade defeituosa não relevante para a execução do processo; executar uma atividade alternativa, com outras características e requisitos, mas que produz um resultado semelhante; compensar uma ou mais atividades falhas, para retornar a um ponto anterior da execução do processo e seguir um caminho alternativo; corrigir o estado interno do processo para continuar a execução; além de qualquer outra ação mais específica, como notificar o usuário sobre o erro e pedir a sua intervenção.

Adicionalmente, utilizando o controlador paralelo simples, que será descrito na próxima seção, é possível executar simultaneamente várias atividades alternativas e obter o resultado da primeira, abortando-se a execução das demais. Todo processo ou controlador é automaticamente recuperável pelo sistema, que inicia a execução de cada um a partir do último ponto salvo da máquina falha que o estava executando, como veremos na Seção 6.3. Entretanto, esse é um mecanismo do sistema, não da linguagem. O sistema Xavantes permite a modificação dinâmica das variáveis e do estado de execução dos processos e controladores, e também da especificação desses elementos, pela modificação de registros em base de dados relacionais, para tratar com exceções não previstas, conforme veremos na descrição do *middleware*. Entretanto, o sistema não contém nenhum mecanismo para verificar e validar a integridade e a consistência dessas modificações.

A linguagem não fornece mecanismos de manutenção da consistência dos dados em caso de falhas, sendo responsabilidade do usuário ao acessar dados externos compartilhados utilizar serviços transacionais distribuídos que garantam a propriedade "tudo-ou-nada" na modificação desses dados, como por exemplo o *Java Transaction API (JTA)* [CM02] que implementa o protocolo *two phase commit*. É interessante que os gerenciadores de recursos de dados compartilhados também suportem mecanismos flexíveis de controle de acesso, com bloqueios de leituras inconsistentes e escritas concorrentes, conforme requerido pelo usuário. Para garantir a consistência dos objetos compartilhados internos aos processos, suas operações devem controlar o acesso reentrante por vários elementos distintos, bem como bloquear leituras e escritas inconsistentes. No futuro, pretende-se implementar um tipo de objeto tabela que forneça as mesmas operações de consulta e atualização, controle de acesso e propriedades transacionais que um banco de dados fornece às suas tabelas, mas em memória volátil e sincronizada com o repositório persistente nos términos das atividades transacionais e nos *checkpoints* dos controladores.

Fluxo de controle assíncrono Embora a linguagem seja simples e de monitoramento fácil, por ser síncrona e estruturada, ela é muito expressiva e permite a especificação de qualquer tipo de ordenação de elementos de processo. Entretanto, o sincronismo e a estruturação dos processos especificados podem retardar a execução de alguns elementos de processo que não têm dependência de dados em relação aos anteriores, mas que mesmo assim devem esperar o término desses elementos no fluxo principal pelo sincronismo exigido pela linguagem. Por exemplo, uma atividade B_i em um controlador par que depende somente dos dados da atividade A_j em um controlador par anterior no fluxo, mas que pela sincronização estruturada tem que esperar o término desse controlador paralelo anterior. Para lidar com essas deficiências, nos casos em que o usuário deseja sacrificar os benefícios dos processos síncronos e estruturados para construir processos complexos e que requeiram maior eficiência ou reatividade, foi prevista uma futura extensão dessa

linguagem para permitir quebrar a sincronia dos controladores e ainda assim manter a estruturação da linguagem. A linguagem poderá ser estendida por meio da criação do conceito de "portas" emissoras e receptoras nos controladores e também de blocos internos nesses controladores que tratam de mensagens advindas de cada uma dessas portas e que decidem quando o corpo do controlador deve executar. Dessa forma, mensagens entre controladores de diferentes ramos da estrutura hierárquica do processo, e mesmo eventos externos ao processo, poderiam ser tratados assincronamente pelos controladores, aumentando a reatividade da linguagem e permitindo a execução antecipada de elementos de processos, que antes teriam que esperar o fluxo principal.

Comparação As linguagens para *workflows* são geralmente baseadas em grafos orientados acíclicos (*Directed Acyclic Graphs – DAGs*), nos quais os nós especificam as tarefas, e as arestas descrevem os fluxos de dados e controle entre elas. Quando a tarefa de um nó termina, são enviadas mensagens de dados e de controle por todos os fluxos descritos nas arestas de saída, o que representa um disparo (*fork*) completo. Os fluxos de controle podem ser diretos, sendo seguidos sempre, ou condicionais, sendo seguidos se e somente se a condição associada for verdadeira. Em cada nó, antes da tarefa, há um tipo de junção (*join*), que determina se a tarefa será executada quando todos os fluxos de controle forem seguidos, *and join*, ou apenas um deles, *or join*. Dessa forma, é permitido descrever todos os fluxos possíveis de dados e controle entre um número fixo de tarefas. Entretanto o comportamento dos seus processos podem ser imprevisíveis em tempo de execução, como qualquer algoritmo baseado em redes de Petri, tornando-a inadequada para especificar e monitorar processos complexos. Comparativamente, elas equivalem a linguagens de programação não-estruturadas, com saltos de execução, mas paralelas, o que torna o entendimento e a previsibilidade do comportamento dos processos um desafio ainda maior.

Quando comparados com linguagens para *workflows*, os controladores da linguagem XSPL representam uma versão estruturada dos controles de fluxo daquelas, porque ambas contêm disparadores (*fork*) e junções (*join*) de execução, com fluxos tanto simples quanto condicionais, mas XSPL estrutura o uso destes. Para lidar com fluxos de dados ou de controle que perpassam a estrutura imposta, a linguagem permite o uso de variáveis compartilhadas bloqueantes, o que permite uma junção total (*and join*) entre atividades contidas em ramos distintos da hierarquia de controladores. Ademais, pelo fato de existirem controladores que determinam em tempo de execução o número de elementos a serem disparados, bem como controladores que tratam explicitamente de exceções, sem nunca perder de vista a estruturação, a linguagem XSPL é certamente mais poderosa do que as linguagens para *workflows* existentes para a especificação de controles de execução, e, mesmo assim, mais simples de usar, seja na definição ou no monitoramento de processos.

4.3 Gramática da Linguagem

Nessa seção descreveremos a sintaxe da *Xavantes Structured Process Language* (XSPL) em uma variante da notação *Extended Backus-Naur Form* (EBNF) [Gar04] que acomoda as especificidades da linguagem XML. A semântica da linguagem é apresentada via comentários sobre o significado de cada sentença em EBNF.

4.3.1 Aplicação

```
APPLICATION ::=
  ( PROCESS_TYPE | ACTIVITY_TYPE )*
```

No modelo de especificação de processos do Xavantes, uma aplicação é especificada por meio da definição de processos e atividades. Os processos são utilizados para se resolver um certo problema complexo, geralmente de longa duração, que demanda um alto grau de processamento paralelo e distribuído e que utiliza dados armazenados em repositórios e fontes dispersas. Um processo compõe e coordena a execução de atividades e invocações de serviços em nós distribuídos da grade para realizar a computação desejada. As atividades do processo realizam tarefas simples a fim de resolver uma parte do problema associado ao processo.

4.3.2 Processo

```
PROCESS_TYPE ::=
  <PROCESS_TYPE NAME="IDENT">
    [<DESCRIPTION>TEXT</DESCRIPTION>]
    [TRANSFERENCE_ESTIMATION]
    (<IMPORT (PACKAGE="PACKAGE" | CLASS="TYPE")/>)*
    (<IN_PARAMETER TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    (<OUT_PARAMETER TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    <BODY>
      ( PROCESS_ELEMENT )*
    </BODY>
    (<CATCH TYPE="EXCEPTION_TYPE" NAME="IDENT">
      ( PROCESS_ELEMENT )*
    </CATCH>)*
    [<FINALLY> ( PROCESS_ELEMENT )* </FINALLY>]
  </PROCESS_TYPE>
```

Um tipo de processo especifica um *template* a partir do qual instâncias podem ser criadas e executadas. Este contém parâmetros de entrada e saída, que definem quais são os tipos de dados recebidos e produzidos pelo processo; descrição de sua função; e elementos de processo em seu corpo, que especificam controladores e atividades que serão executados, determinando o comportamento e a funcionalidade deste. O processo compõe e coordena a execução desses elementos em um ou mais nós distribuídos da grade. Este processo pode especificar variáveis, como forma de utilizar objetos para cálculos intermediários ou compartilhamento de dados entre os seus elementos.

Os elementos de um processo são especificados em seu bloco body. Os elementos incluídos no processo podem ser tanto invocações de atividades, processos e controladores, quanto blocos de código em BeanShell [Nie07], uma linguagem *script* baseada em Java. Esses elementos são executados em seqüência, por definição. Fluxos de controle mais complexos devem ser especificados por meio da composição de controladores, descritos mais abaixo. Para a inclusão de código, em *script* Java, o usuário pode importar pacotes e classes das bibliotecas Java locais do ambiente de execução ou incluídos nos arquivos de biblioteca Java associados ao processo, implementadas pelo usuário ou por terceiros. Para isso, ele deve declarar quais são os pacotes a serem utilizados com um marcador import por pacote ou classe.

Nos processos, assim como nos controladores, a comunicação de dados é realizada por meio de parâmetros de entrada e de saída, que trocam objetos locais ou remotos, por valor ou por referência. Objetos locais representam dados primitivos ou estruturados e são passados por valor, objetos remotos padrão *Java RMI* [Mic04] representam dados compartilhados e são passados por referência.

Os parâmetros de entrada são declarados utilizando-se o marcador in_parameter. Ele contém o tipo do objeto aceito, definido pelo atributo type, e o nome do parâmetro, definido pelo atributo name. Opcionalmente é possível definir o valor padrão do parâmetro de entrada por meio do atributo value, utilizado caso não seja passado nenhum objeto no momento da invocação do processo. O valor a ser atribuído por padrão pode ser descrito por qualquer expressão que retorne o tipo especificado, como por exemplo uma expressão numérica resultando em um objeto valor ou um identificador de variável. De modo similar, os parâmetros de saída são definidos com o marcador out_parameter.

Tanto os processos quantos os controladores podem conter variáveis internas utilizadas para compartilhar objetos entre os seus elementos ou armazenar resultados intermediários. Variáveis internas podem ser declaradas de duas formas: como variáveis de um bloco especificado em *script* Java ou como variáveis explícitas do processo ou controlador. As variáveis explícitas são declaradas por meio do marcador var, que tem como atributos o tipo de dado aceito (type), o nome da variável (name) e valor inicial (value), da mesma forma que os parâmetros do processo. Ambos os tipos de variáveis tem como escopo todo

o corpo do processo ou controlador, bem como os blocos de tratamento de exceção e sincronização. A diferença é que a variável explícita é mantida diretamente pelo processo e tem seu estado persistido periodicamente para posterior recuperação; enquanto que a variável do *script* Java é utilizada geralmente para controles internos, tais como contadores, e tem seu estado transiente.

Podem ocorrer falhas de sistema ou de aplicação durante a execução dos elementos de um processo ou controlador. Para lidar com esses problemas, os processos suportam o tratamento de exceções como uma técnica de tolerância a falhas não-mascaradas. As exceções são levantadas devido a falhas no sistema e a falhas de especificação ou de execução do processo ou de suas atividades, assim elas podem ser classificadas em exceções de sistema e de aplicação, respectivamente. Todas as exceções levantadas pelo sistema ou pela aplicação têm tipo e conteúdo, que as definem. As exceções são objetos Java do tipo *Throwable*. As exceções são tratadas por blocos catch. Cada bloco trata exceções de um certo tipo, levantadas pelos elementos do corpo do controlador ou processo, de forma semelhante ao comportamento da linguagem Java. O tipo de exceção que um determinado bloco catch pode tratar é determinado pelo seu atributo type e a instância do objeto que representa a exceção é obtida da variável identificada pelo atributo name. Para tratar exceções, é possível utilizar qualquer conjunto de elementos de processo nesse bloco. Nele, o programador pode especificar elementos que testem o estado em que o processo estava quando houve a exceção e decida se o usuário deve ser notificado ou se atividades alternativas devem ser executadas, a fim de manter a consistência dos dados e do estado do processo.

O bloco finally é sempre executado no final da execução de um controlador, seja na execução normal, ou havendo exceções. Nesse bloco também é possível utilizar qualquer conjunto de elementos para tratar da finalização do controlador. Entretanto, a utilização mais comum desse tipo de bloco, tal como em programas Java, é a liberação de recursos e a restauração da execução normal do processo, mesmo na ocorrência de exceções.

Na seção transference_estimation, contida tanto em processos quanto em controladores, podem ser inseridas estimativas de transferência total de dados entre o processo ou o controlador e o seu elemento invocador, em *kilobytes*. A transferência de dados estimada considera tanto mensagens trocadas pelos parâmetros de entrada e saída, quanto acessos remotos realizados a objetos compartilhados durante a sua execução. Essa estimativa, por ser baseada em execuções anteriores, é utilizada pelo serviço de escalonamento para escolher o melhor nó para alocar o processo ou controlador, a fim de diminuir o tempo de transferência de dados entre esse elemento e seu invocador, e portanto, aumentar o desempenho de execução da aplicação.

4.3.3 Atividade

```
ACTIVITY_TYPE ::=
  <ACTIVITY_TYPE NAME="IDENT">
    [<DESCRIPTION>TEXT</DESCRIPTION>]
    [EXECUTION_REQUIREMENTS]
    [EXECUTION_ESTIMATIONS]
    [FAULT_TOLERANCE_ATTRIBUTES]
    (<IMPORT (PACKAGE="PACKAGE" | CLASS="TYPE")/>)*
    (<IN_PARAMETER TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    (<OUT_PARAMETER TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    (CODE | INVOKE_WEB_SERVICE | INVOKE_OBJECT)
    (<CATCH TYPE="EXCEPTION_TYPE" NAME="IDENT">
      ( PROCESS_ELEMENT )*
    </CATCH>)*
    [<FINALLY> ( PROCESS_ELEMENT )* </FINALLY>]
  </ACTIVITY_TYPE>
```

A especificação de um tipo de atividade contém o nome da atividade, uma descrição opcional da sua função e um corpo que define sua implementação. O corpo pode conter código Java, ou uma invocação a um objeto remoto, ou uma invocação a um serviço *web*. Os dados de entrada para a execução da atividade, bem como os resultados, são passados por parâmetros tipados de entrada e de saída, como nos controladores e nos processos. Os tipos aceitos são tipos de objeto local ou remoto Java. Para a execução de código Java, o programador pode importar pacotes e classes das bibliotecas locais do ambiente de execução ou de bibliotecas enviadas com a atividade, como em processos e controladores. O código Java é inserido no bloco code.

Uma especificação de atividade pode definir, na região execution_requirement, os requisitos para a execução de suas instâncias, tais como tamanhos de memória volátil e persistente exigidos, e bibliotecas e arquivos necessários . Na região execution_estimation podem ser inseridas estimativas de esforço de processamento necessário para executar a atividade, em número de milhares de instruções básicas; e de transferência total de dados entre o invocador e a atividade, em *kilobytes*, realizada por mensagens trocadas pelos parâmetros de entrada e saída e acessos a objetos compartilhados durante a sua execução. Essa estimativa, que pode ser baseada em execuções anteriores, é utilizada pelo serviço de escalonamento para selecionar o melhor tempo e lugar para alocar essa atividade, a fim de otimizar o desempenho da execução do processo.

As atividades definidas por código Java podem ser alocadas em qualquer recurso computacional da grade que satisfaça os requisitos necessários para executar a atividade.

Entretanto, as atividades que contém invocações a objetos remotos ou serviços *web* devem preferencialmente ser executadas próximas aos objetos invocados. A duração da computação, nesses casos, é calculada pelo tempo de comunicação entre a atividade e o objeto invocado, bem como o tempo de execução do serviço. Nesses dois casos, o tempo de duração estimado pode ser explicitado no bloco idle_time_estimation descrito na Seção 4.3.10.

A região fault_tolerance_attributes contém atributos que podem ser obtidos pelo *middleware* para saber quais são as requisitos e características da atividade para que este trate de falhas de sistema ocorridas na atividade, interagindo adequadamente com a atividade falha e evitando que essas falhas sejam percebidas pelo processo ou controlador invocador. Atualmente há nessa região a possibilidade de se especificar, como veremos adiante, se uma atividade é: recuperável, o que permite ao sistema notificar a atividade para que esta se restaure a partir do ponto em que ela falhou (*resume*); transacional, que define se uma atividade tem a propriedade "tudo ou nada" e se precisa ser notificada pelo sistema se esta deve ser concluída (*commit*) ou abortada (*abort*).

De forma semelhante ao processo e ao controlador, a atividade pode conter blocos catch e um bloco finally para tratar exceções não mascaradas pelo *middleware*. A finalidade do bloco catch é obter a exceção levantada e tratar essa exceção. Entre os usos desse bloco está: mapear a exceção para uma nova e levantá-la para o controlador invocador tratar; abortar a execução da atividade dependendo do tipo da exceção capturada, se a atividade for recuperável; ignorar a exceção capturada terminando a execução da atividade ou restaurando essa execução, se a atividade for recuperável nesse último caso. O bloco finally pode ser utilizado para liberar recursos ao final da execução da atividade.

4.3.4 Elemento de Processo

```
PROCESS_ELEMENT ::=
  CONTROLLER | INVOKE_PROCESS | INVOKE_ACTIVITY | CODE

CONTROLLER ::=
  BLOCK | SWITCH | FOR | FOREACH | WHILE |
  PAR | PARSWITCH | PARFOR | PARFOREACH | PARWHILE
```

Os tipos de elemento que os processos e controladores podem conter são: controlador, invocação de processo, invocação de atividade e código em BeanShell, linguagem *script* Java. Tanto um processo quanto um controlador pode conter uma combinação qualquer desses elementos. Em um processo, os elementos são executados em seqüência, enquanto que em um controlador, a ordem de execução depende de seu tipo.

4.3.5 Controladores

Existem vários tipos de controladores na linguagem, conforme descrito anteriormente. Eles são divididos em seqüenciais e paralelos, conforme o paralelismo; e simples, condicionais, iterativo e iterativos-condicionais, quanto ao tipo de controle de fluxo. Os seqüenciais são: block, switch, for, foreach e while; enquanto os paralelos, que correspondem aos seqüenciais, são, respectivamente: par, parswitch, parfor, parforeach e parwhile.

Controlador Seqüencial Simples

```
BLOCK ::=
  <BLOCK NAME="IDENT">
    [TRANSFERENCE_ESTIMATION]
    (<IMPORT (PACKAGE="PACKAGE" | CLASS="TYPE")/>)*
    (<IN_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="EXPRESSION"/>)*
    (<OUT_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="IDENT"/>)*
    (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    <BODY>
      ( PROCESS_ELEMENT )*
    </BODY>
    (<CATCH TYPE="EXCEPTION_TYPE" NAME="IDENT">
      ( PROCESS_ELEMENT )*
    </CATCH>)*
    [<FINALLY> ( PROCESS_ELEMENT )* </FINALLY>]
  </BLOCK>
```

O controlador block é utilizado para executar um conjunto de elementos de processo, cada um após o término do anterior. Esses elementos, por definição, podem ser invocações a atividades e a processos, controladores e códigos escritos em BeanShell, uma linguagem *script* Java. Os elementos a serem executados em seqüência são inseridos entre os marcadores body.

Assim como um processo, um controlador também: troca dados com o elemento invocador, que o contém, por meio de seus parâmetros de entrada e saída; pode conter variáveis internas para armazenamento de resultados intermediários da computação e dados de controle; pode tratar exceções de sistemas e de aplicação, e conter bloco de finalização; e pode conter estimativa de transferência total de dados entre o controlador e o elemento invocador, a fim de otimizar o seu escalonamento. Para facilitar o monitoramento de instâncias de processos e possibilitar a modificação dinâmica dessas, todos os controladores contêm uma identificação única em relação aos elementos do mesmo nível, dada pelo valor do seu atributo name. Essa identificação também pode ser utilizada para

acessar o estado e atuar sobre o objeto controlador, realizando-se reflexão computacional de comportamento.

Existe uma diferença fundamental quanto ao uso dos marcadores de declaração de parâmetros de entrada e saída de um controlador em relação ao uso desses marcadores em processos e atividades: os marcadores de parâmetros do controlador servem tanto para definir os objetos aceitos pelo parâmetros quanto para atribuir valores de entrada ou variáveis de saída a estes, enquanto os marcadores de parâmetros dos processos e atividades servem apenas para declarar os objetos aceitos pelos parâmetros. Em um processo ou atividade, apenas os elementos invoke_process e invoke_activity atribuem valores de entrada e variáveis de saída aos parâmetros, nas chamadas de instâncias de processos e atividades, respectivamente, como veremos adiante.

Em um marcador de declaração de parâmetro de entrada de um controlador, além da definição do tipo de objeto aceito em type e do nome do parâmetro em name, deve-se definir o valor de entrada do parâmetro em value por meio de uma expressão em *script* Java que crie um objeto de dados. Em um marcador de parâmetro de saída, além da definição do tipo do objeto retornado e do nome do parâmetro nos respectivos atributos, deve-se atribuir ao parâmetro value uma expressão que retorne uma variável ou campo de objeto para receber o objeto retornado.

Controlador Seqüencial Condicional

```
SWITCH ::=
  <SWITCH NAME="IDENT">
    [TRANSFERENCE_ESTIMATION]
    (<IMPORT (PACKAGE="PACKAGE" | CLASS="TYPE")/>)*
    (<IN_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="EXPRESSION"/>)*
    (<OUT_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="IDENT"/>)*
    (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    (<CASE COND="BOOL_EXPRESSION" [SEL_RATE="FLOAT_EXPRESSION"]>
      ( PROCESS_ELEMENT )*
    </CASE>)+
    [<ELSE [SEL_RATE="FLOAT_EXPRESSION"]>
        ( PROCESS_ELEMENT )*
    </ELSE>]
    (<CATCH TYPE="EXCEPTION_TYPE" NAME="IDENT">
      ( PROCESS_ELEMENT )*
    </CATCH>)*
    [<FINALLY> ( PROCESS_ELEMENT )* </FINALLY>]
  </SWITCH>
```

O controlador switch é seqüencial e condicional. Ele é utilizado para selecionar um caminho de execução do processo dependendo da condição associada ao caminho: o primeiro caminho que tiver a condição verdadeira é executado. Se todos os caminhos tiverem condição falsa, então o controlador pode executar um caminho de contingência, caso ele seja declarado.

Cada caminho de execução alternativo é declarado utilizando-se um bloco case e a condição associada a esse caminho é definida pela expressão booleana inclusa no seu atributo cond. Se a expressão resultar verdadeira, e se nenhum outro caminho foi selecionado antes, então o controlador determina a execução seqüencial dos elementos contidos neste bloco. Caso nenhum bloco case seja executado, o controlador pode executa seqüencialmente os elementos contidos no bloco else, caso ele tenha sido declarado.

O atributo sel_rate dos blocos case e else suporta uma expressão numérica em ponto flutuante de 0 a 100, que provê uma estimativa percentual ao escalonador sobre a chance de um certo caminho ser executado, em tempo de instanciação do processo. A expressão pode ser baseada em parâmetros de entrada do processo. Essa informação é útil para priorizar mais um caminho do que outro no escalonamento e reserva de recursos, de forma a não reservar recursos desnecessariamente. Outra vantagem é prover antecipadamente um cálculo mais preciso do tempo de início e fim da execução dos elementos do processo, permitindo realizar um escalonamento mais eficaz e, portanto, melhorar o desempenho da execução do processo.

Controladores Seqüenciais Iterativos

```
FOR ::=
  <FOR NAME="IDENT">
    [TRANSFERENCE_ESTIMATION]
    (<IMPORT (PACKAGE="PACKAGE" | CLASS="TYPE")/>)*
    (<IN_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="EXPRESSION"/>)*
    (<OUT_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="IDENT"/>)*
    (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    <ITERATE VAR="IDENT" BEGIN="INT_EXPRESSION" END="INT_EXPRESSION"
      [NUM_ITERATIONS_EST="INT_EXPRESSION"]>
      ( PROCESS_ELEMENT )*
    </ITERATE>
    (<CATCH TYPE="EXCEPTION_TYPE" NAME="IDENT">
      ( PROCESS_ELEMENT )*
    </CATCH>)*
    [<FINALLY> ( PROCESS_ELEMENT )* </FINALLY>]
  </FOR>
```

O controlador for é seqüencial e iterativo. Ele é utilizado para executar uma seqüência de elementos de processo um determinado número de vezes, uma iteração após a outra. A seqüencia de elementos a ser executada é declarada no bloco iterate do controlador. O atributo var desse marcador é utilizado para definir o nome da variável que armazena o contador da iteração, que pode ser utilizada pelos elementos do controlador para realizar computações dependentes do número da iteração atual. O controlador itera sobre os valores de contador definidos na faixa do valor do atributo begin até o valor do atributo end, incrementando o contador a cada iteração, tal como o construtor de controle for da linguagem *Pascal*.

O atributo num_iterations_est do bloco iterate é utilizado para estimar, em tempo de instanciação do processo, o número de iterações a serem realizadas pelo controlador. Ele é definido por uma expressão inteira em *script* Java, que pode ser baseada nos parâmetros de entrada do processo. O objetivo desse atributo é fornecer uma estimativa ao escalonador do processo para que ele possa reservar recursos antecipadamente, até um certo limite, e determinar com maior precisão o tempo de início e fim de execução de cada elemento e do próprio controlador, a fim de realizar um escalonamento mais eficaz e uma execução mais eficiente do processo.

```
FOREACH ::=
  <FOREACH NAME="IDENT">
    [TRANSFERENCE_ESTIMATION]
    (<IMPORT (PACKAGE="PACKAGE" | CLASS="TYPE")/>)*
    (<IN_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="EXPRESSION"/>)*
    (<OUT_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="IDENT"/>)*
    (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    <ITERATE VAR="IDENT" COLLECTION="EXPRESSION"
      [NUM_ITERATIONS_EST="INT_EXPRESSION"]>
      ( PROCESS_ELEMENT )*
    </ITERATE>
    (<CATCH TYPE="EXCEPTION_TYPE" NAME="IDENT">
      ( PROCESS_ELEMENT )*
    </CATCH>)*
    [<FINALLY> ( PROCESS_ELEMENT )* </FINALLY>]
  </FOREACH>
```

O controlador foreach é uma especialização do for com o objetivo de facilitar a iteração sobre objetos de uma coleção, executando uma seqüência de elementos especificados para cada objeto desta coleção. Ele tem um comportamento semelhante ao construtor de controle homônimo da linguagem de programação C# [Cor07a]. O seu bloco iterate define

a coleção de objetos sobre a qual o controlador iterará no atributo collection e a variável que conterá o objeto de cada iteração no atributo var. O atributo collection deve conter uma expressão que retorna um objeto do tipo *Collection* da linguagem *Java*. O atributo var deve conter um identificador de variável para armazenar cada objeto da coleção. O corpo do bloco contém os elementos que poderão utilizar a variável com o elemento de cada iteração para realizar a computação pretendida. O atributo num_iterations_est contém o número estimado de iterações em tempo de instanciação do processo, que, neste caso, é o tamanho estimado da coleção atribuída, para fins de cálculos de escalonamento.

Controlador Seqüencial Iterativo-Condicional

```
WHILE ::=
  <WHILE NAME="IDENT">
    [TRANSFERENCE_ESTIMATION]
    (<IMPORT (PACKAGE="PACKAGE" | CLASS="TYPE")/>)*
    (<IN_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="EXPRESSION"/>)*
    (<OUT_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="IDENT"/>)*
    (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    <ITERATE COND="BOOL_EXPRESSION" [STEP="SCRIPT"]
      [NUM_ITERATIONS_EST="INT_EXPRESSION"]>
      ( PROCESS_ELEMENT )*
    </ITERATE>
    (<CATCH TYPE="EXCEPTION_TYPE" NAME="IDENT">
      ( PROCESS_ELEMENT )*
    </CATCH>)*
    [<FINALLY> ( PROCESS_ELEMENT )* </FINALLY>]
  </WHILE>
```

O controlador while também é iterativo, como o for, mas a quantidade de iterações é determinada por uma condição. Detalhadamente, os elementos contidos no bloco iterate são executados em seqüência por

um certo número de vezes, uma iteração após a outra, enquanto a condição do atributo cond resultar verdadeira. Essa condição é definida por uma expressão booleana, que pode utilizar parâmetros e variáveis do controlador.

O atributo step pode conter qualquer *script* Java. Ele tem como função alterar o valor das variáveis utilizadas pela condição e pelos elementos do controlador a cada iteração. Um exemplo de sua utilização é a obtenção de cada um dos objetos de uma coleção, um por iteração, de modo semelhante ao foreach. A condição, nesse caso, serviria para testar se ainda resta elementos a serem obtidos e terminar as iterações quando não houver mais.

Controlador Paralelo Simples

```
PAR ::=
  <PAR NAME="IDENT">
    [TRANSFERENCE_ESTIMATION]
    (<IMPORT (PACKAGE="PACKAGE" | CLASS="TYPE")/>)*
    (<IN_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="EXPRESSION"/>)*
    (<OUT_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="IDENT"/>)*
    (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    (<START [NAME="IDENT"]>
      (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
      ( PROCESS_ELEMENT )*
    </START>)+
    [<JOIN [WAIT="INT_EXPRESSION"]> SCRIPT </JOIN>]
    [<TERMINATE COND="BOOL_EXPRESSION"/>]
    (<CATCH TYPE="EXCEPTION_TYPE" NAME="IDENT">
      ( PROCESS_ELEMENT )*
    </CATCH>)*
    [<FINALLY> ( PROCESS_ELEMENT )* </FINALLY>]
  </PAR>
```

Deve-se ressaltar que a linguagem XSPL oferece meios de se especificar qualquer fluxo de controle paralelo, mas síncrono e estruturado, por meio de seus controladores. O mais simples de todos os controladores paralelos é o par, que permite a execução paralela de um conjunto de caminhos (*threads*) especificados. Cada caminho é especificado em um bloco start e contém todos os elementos de processo que serão executados seqüencialmente por esse caminho. Pelo menos um caminho deve ser especificado. O seu atributo name pode ser definido para identificar facilmente um caminho para monitoramento e reflexão computacional.

Cada caminho de execução paralela, definido no bloco start, pode conter variáveis locais, utilizadas somente por elementos especificados nesse caminho. Cada variável local ao caminho é declarada por meio de um marcador var internamente ao bloco start que define o caminho em que esta variável será utilizada. A declaração das variáveis locais a um caminho é realizada da mesma forma que a declaração de uma variável local a um controlador. Deste modo, o atributo type define o tipo de objeto aceito pela variável, name, o nome da variável, e value pode conter o valor inicial da variável. A variável local de caminho é utilizada para armazenar computações internas dos elementos de cada caminho, de forma que os valores computados em um caminho não interfiram nos valores computados em outro caminho.

Com o objetivo de permitir a declaração de junções condicionais de fluxos paralelos, foram definidos dois elementos nos controladores paralelos: join e terminate. O *script* Java especificado no corpo do bloco join é executado toda vez que um dos caminhos paralelos termina ou que um determinado tempo é transcorrido. O tempo máximo de espera pelo término de um caminho pode ser especificado pelo atributo wait, em milissegundos. O bloco join pode ser usado para agregar resultados das computações realizadas pelos elementos de processo do caminho terminado e para atualizar o valor das variáveis utilizadas posteriormente, pelos demais marcadores do controlador. A definição do tempo máximo de espera é opcional e pode servir para que o bloco join verifique periodicamente o estado dos caminhos de execução e tome uma decisão, com por exemplo o término do controlador.

O marcador terminate contém uma condição que define se o controlador deve terminar. Essa condição é definida pelo atributo cond, que contém uma expressão booleana. Caso a condição resulte verdadeira, o controlador paralelo é terminado e todos os caminhos em execução são abortados. Para especificar o término do controlador somente após a execução de todos os caminho, *and-join*, basta não especificar o elemento terminate ou atribuir falso à sua condição. Para terminar o controlador após a execução de qualquer caminho, *or-join*, basta atribuir verdadeiro à condição do elemento terminate. Para especificar um término condicional, *conditional or-join*, a condição do elemento terminate pode utilizar as variáveis atualizadas pelo código incluso no elemento join. Dessa forma, por exemplo, é fácil de especificar se o controlador deve terminar após um certo número de caminhos terminar, um caminho específico terminar ou após transcorrido um certo tempo de execução do controlador.

O tratamento de exceções dos controladores paralelos é similar ao dos controladores seqüenciais. Uma vez que uma exceção ocorre em um dos caminhos paralelos do controlador e não é tratada pelos elementos de processos internos, a exceção é repassada para o controlador paralelo que pode tratá-la ou não, em um dos seus elementos catch. No tratamento da exceção, o controlador pode corrigir o problema do elemento defeituoso, abortar a execução do elemento defeituoso, ou abortar a execução de todo o controlador. Quando a execução do controlador é abortada, todos os caminhos também são abortados e uma exceção de aborto é levantada para o elemento superior, que o invocou. Após tratar a exceção, o código especificado como *script* Java pode lançar uma exceção com o comando *throw*. Se a exceção não for tratada ou se for lançada uma nova exceção, esta é repassada para o elemento superior na hierarquia de controladores, que pode tratá-la dentro de seu escopo. Se a exceção não for tratada por nenhum elemento superior e nem pelo processo, esta é apresentada para o usuário que deve tratá-la ou abortar a aplicação. O bloco finally é sempre executado no término regular do controlador ou no aborto da execução do controlador, caso este bloco seja declarado, e serve principalmente para liberar recursos utilizados.

Controlador Paralelo Condicional

```
PARSWITCH ::=
  <PARSWITCH NAME="IDENT">
    [TRANSFERENCE_ESTIMATION]
    (<IMPORT (PACKAGE="PACKAGE" | CLASS="TYPE")/>)*
    (<IN_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="EXPRESSION"/>)*
    (<OUT_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="IDENT"/>)*
    (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    (<CASE COND="BOOL_EXPRESSION" [NAME="EXPRESSION"]
      [SEL_RATE="FLOAT_EXPRESSION"]>
      (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
      ( PROCESS_ELEMENT )*
    </CASE>)+
    [<ELSE [NAME="EXPRESSION"] [SEL_RATE="FLOAT_EXPRESSION"]>
      (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
      ( PROCESS_ELEMENT )*
    </ELSE>]
    [<JOIN [WAIT="INT_EXPRESSION"]> SCRIPT </JOIN>]
    [<TERMINATE COND="BOOL_EXPRESSION"/>]
    (<CATCH TYPE="EXCEPTION_TYPE" NAME="IDENT">
      ( PROCESS_ELEMENT )*
    </CATCH>)*
    [<FINALLY> ( PROCESS_ELEMENT )* </FINALLY>]
  </PARSWITCH>
```

O controlador parswitch é a versão paralela do controlador condicional switch. Em ambos, cada elemento case especifica um caminho a ser executado caso a condição associada, definida por uma expressão booleana no atributo cond, seja verdadeira. Diferentemente do controlador switch, todos os caminhos que têm a condição verdadeira são executados em paralelo. Caso nenhum tenha a condição verdadeira, então o caminho definido pelo bloco else é executado, se ele for definido. Como pode haver vários caminhos sendo executados concorrentemente, para identificar facilmente cada caminho o usuário pode especificar o nome deste caminho dentro do controlador utilizando o atributo name do marcador case ou else responsável pelo seu disparo. Cada caminho paralelo pode conter suas variáveis locais, declaradas pelos marcadores var, permitindo que as computações internas de um caminho não interfiram nas de outro caminho. A junção de fluxos paralelos ocorre até que todos os caminhos paralelos disparados terminem. Assim, se somente um caminho for disparado, quando este terminar, o controlador também é terminado.

O atributo sel_rate dos marcadores case e else são utilizados, assim como no switch, para fornecer uma estimativa percentual ao serviço de escalonamento, em tempo de instanciação do processo, da chance de um certo caminho ser executado. Esse atributo recebe um valor de 0 a 100, sendo que 100 determina que o caminho será executado com certeza e 0 o inverso, i.e., não será executado. Diferentemente do switch, a somatória desses atributos de todos os marcadores case pode ultrapassar 100, uma vez que vários caminhos podem ter a chance de serem executados em paralelo. Esse atributo no elemento else deve especificar a chance de nenhum dos caminhos definidos por case serem executados.

Controladores Paralelos Iterativos

```
PARFOR ::=
  <PARFOR NAME="IDENT">
    [TRANSFERENCE_ESTIMATION]
    (<IMPORT (PACKAGE="PACKAGE" | CLASS="TYPE")/>)*
    (<IN_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="EXPRESSION"/>)*
    (<OUT_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="IDENT"/>)*
    (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    <ITERATE VAR="IDENT" BEGIN="INT_EXPRESSION" END="INT_EXPRESSION"
      [SLEEP="INT_EXPRESSION"] [MAX_THREADS="INT_EXPRESSION"]
      [NUM_ITERATIONS_EST="INT_EXPRESSION"]>
      (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
      [<PRE_START> SCRIPT </PRE_START>]
      ( PROCESS_ELEMENT )*
    </ITERATE>
    [<JOIN [WAIT="INT_EXPRESSION"]> SCRIPT </JOIN>]
    [<TERMINATE COND="BOOL_EXPRESSION"/>]
    (<CATCH TYPE="EXCEPTION_TYPE" NAME="IDENT">
      ( PROCESS_ELEMENT )*
    </CATCH>)*
    [<FINALLY> ( PROCESS_ELEMENT )* </FINALLY>]
  </PARFOR>
```

O controlador parfor é o correspondente paralelo do for, mas ao invés de executar uma seqüência de elementos um certo número de vezes, uma vez após a outra, o parfor executa em paralelo um certo número de instâncias, determinado em tempo de execução, da mesma seqüência de elementos. Efetivamente, o controlador executa várias iterações de disparo de execução de caminho paralelo (*thread*), sendo um caminho paralelo para cada seqüência de elementos descritos no bloco iterate.

No controlador parfor há uma variável utilizada para armazenar o valor do contador de iterações, declarada pelo atributo var do marcador iterate. O número de iterações de instanciação e disparo é definido pela diferença de seus atributos end e begin mais um. Na primeira iteração de disparo, o valor do contador é igual ao valor de begin e depois ele é incrementado a cada iteração, até que se torne igual ao valor de end. A cada iteração é disparada a execução de um caminho e para cada caminho é criado um escopo com uma variável contador independente, iniciada com o valor da iteração, e com outras variáveis declaradas por marcadores var no corpo do bloco iterate. O valor inicial destas variáveis internas ao bloco de interação é calculado na iteração, antes do disparo do caminho. Deste modo, pode-se executar paralelamente o mesmo conjunto de elementos, mas utilizando dados diferentes, permitindo-se realizar computações no modelo *Single Instruction Multiple Data (SIMD)* em grades computacionais.

O programador pode determinar o tempo de espera em milissegundos entre uma iteração e a próxima utilizando o atributo sleep do bloco iterate para reduzir a taxa de disparos de execução paralela. O atributo max_threads permite indicar o número máximo de execuções simultâneas em um controlador, útil quando o número de instâncias paralelas a serem criadas puder ser muito alto e se desejar limitar o uso de recursos.

Nos controladores paralelos iterativos, o programador pode especificar um *script* Java para ser executado antes do disparo de um caminho paralelo, ocorrido em cada iteração. Esse código deve ser especificado no bloco pre_start. Ele é útil para iniciar as variáveis do escopo do caminho paralelo a ser disparado, dependentes da iteração, bem como para interromper as iterações ou deixar de executar algumas iterações dependendo do valor das variáveis internas do controlador.

O atributo num_iterations_est permite que o serviço de escalonamento, em tempo de instanciação do processo, tenha uma estimativa do número de iterações de disparo de caminho paralelo para um determinado controlador iterativo paralelo. Desse modo, o escalonador pode reservar *a priori*, de acordo com a estimativa feita, um certo número de recursos para executar as instâncias desses caminhos paralelos. Como resultado, é possível fornecer uma melhor qualidade de serviço e desempenho de execução de processos. Se, durante a execução não houver recursos suficientes alocados, o escalonador pode reservar dinamicamente mais recursos para executar as instâncias remanescentes de caminhos paralelos.

```
PARFOREACH ::=
  <PARFOREACH NAME="IDENT">
    [TRANSFERENCE_ESTIMATION]
    (<IMPORT (PACKAGE="PACKAGE" | CLASS="TYPE")/>)*
    (<IN_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="EXPRESSION"/>)*
    (<OUT_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="IDENT"/>)*
```

```
    (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    <ITERATE VAR="IDENT" COLLECTION="EXPRESSION"
      [SLEEP="INT_EXPRESSION"] [MAX_THREADS="INT_EXPRESSION"]
      [NUM_ITERATIONS_EST="INT_EXPRESSION"]>
      (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
      [<PRE_START> SCRIPT </PRE_START>]
      ( PROCESS_ELEMENT )*
    </ITERATE>
    [<JOIN [WAIT="INT_EXPRESSION"]> SCRIPT </JOIN>]
    [<TERMINATE COND="BOOL_EXPRESSION"/>]
    (<CATCH TYPE="EXCEPTION_TYPE" NAME="IDENT">
      ( PROCESS_ELEMENT )*
    </CATCH>)*
    [<FINALLY> ( PROCESS_ELEMENT )* </FINALLY>]
  </PARFOREACH>
```

O controlador parforeach é uma especialização do parfor e uma versão paralela do controlador foreach. Ele foi projetado para executar em paralelo vários caminhos (*threads*) com seqüência de elementos de processo, sendo um caminho para cada objeto de uma coleção. Da mesma forma que o parfor, ele itera um determinado número de vezes, mas sobre cada objeto de uma coleção ao invés de sobre cada número de uma faixa, disparando um caminho paralelo de execução (*thread*) a cada iteração. O caminho paralelo disparado executa a seqüência de elementos definida no corpo do bloco iterate. Esses elementos podem utilizar a variável identificada pelo atributo var para acessar o objeto da coleção associado ao caminho paralelo disparado.

Controlador Paralelo Iterativo-Condicional

```
PARWHILE ::=
  <PARWHILE NAME="IDENT">
    [TRANSFERENCE_ESTIMATION]
    (<IMPORT (PACKAGE="PACKAGE" | CLASS="TYPE")/>)*
    (<IN_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="EXPRESSION"/>)*
    (<OUT_PARAMETER TYPE="TYPE" NAME="IDENT" VALUE="IDENT"/>)*
    (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
    <ITERATE COND="BOOL_EXPRESSION" STEP="SCRIPT"
      [SLEEP="INT_EXPRESSION"] [MAX_THREADS="INT_EXPRESSION"]
      [NUM_ITERATIONS_EST="INT_EXPRESSION"]>
      (<VAR TYPE="TYPE" NAME="IDENT" [VALUE="EXPRESSION"]/>)*
```

```
    [<PRE_START> SCRIPT </PRE_START>]
    ( PROCESS_ELEMENT )*
  </ITERATE>
  [<JOIN [WAIT="INT_EXPRESSION"]> SCRIPT </JOIN>]
  [<TERMINATE COND="BOOL_EXPRESSION"/>]
  (<CATCH TYPE="EXCEPTION_TYPE" NAME="IDENT">
    ( PROCESS_ELEMENT )*
  </CATCH>)*
  [<FINALLY> ( PROCESS_ELEMENT )* </FINALLY>]
</PARWHILE>
```

O controlador parwhile é o correspondente paralelo do while. Ele executa um certo número, determinado em tempo de execução, de caminhos paralelos (*threads*) de seqüência de elementos. Efetivamente, assim como o parfor, o controlador itera um determinado número de vezes e, em cada uma, ele instancia e dispara um caminho paralelo (*thread*) para a execução paralela da mesma seqüência de elementos. A seqüência de elementos a ser executada em paralelo é definida pelo bloco iterate. As iterações de disparo de execução paralela ocorrem enquanto a condição, definida pelo valor do atributo cond do marcador iterate, for verdadeira. No final de cada iteração, o *script* Java especificado pelo atributo step é executado com o intuito de alterar o valor das variáveis para a próxima iteração, inclusive a utilizada na condição de iteração.

Internamente ao bloco iterate do parwhile podem ser definidas variáveis, assim como no do parfor. As variáveis definidas nesse escopo têm um valor específico para cada caminho paralelo de execução da seqüência de elementos (*thread*), diferentemente das variáveis definidas fora do bloco iterate, que são compartilhadas por todos os caminhos paralelos. A finalidade dessas variáveis é armazenar valores particulares, a serem utilizados somente pelo caminho paralelo, tal como o número da iteração ou os dados sobre os quais devem ser realizados a computação nesse caminho. Um exemplo de uso é o armazenamento do objeto de uma coleção que deve ser utilizado pelo caminho. Com todos esses mecanismos, o controlador parwhile é flexível o suficiente para executar qualquer tipo de disparo de *threads* de execução sobre qualquer conjunto de dados.

4.3.6 Invocação de Processo e Atividade

```
INVOKE_PROCESS ::=
  <INVOKE_PROCESS TYPE="TYPE" NAME="IDENT">
    (<IN_PARAMETER NAME="IDENT" VALUE="EXPRESSION"/>)*
    (<OUT_PARAMETER NAME="IDENT" VALUE="IDENT"/>)*
  </INVOKE_PROCESS>
```

```
INVOKE_ACTIVITY ::=
  <INVOKE_ACTIVITY TYPE="TYPE" NAME="IDENT"
    [RETRIES="INT_EXPRESSION"] [RETRY_TIME="INT_EXPRESSION"]
    [REPLICAS="INT_EXPRESSION"]>
    (<IN_PARAMETER NAME="IDENT" VALUE="EXPRESSION"/>)*
    (<OUT_PARAMETER NAME="IDENT" VALUE="IDENT"/>)*
  </INVOKE_ACTIVITY>
```

Existem dois outros elementos de processo além dos controladores: invocação de processo (invoke_process) e invocação de atividade (invoke_activity). Para invocar um processo no contexto de um processo pai ou controlador, é necessário passar à invoke_process o nome do tipo do processo no seu atributo type. Ao executar a invocação de processo, o sistema cria uma instância desse processo, identifica a instância pelo valor do atributo name, e invoca a instância. O mesmo ocorre na invocação de uma atividade por um processo ou controlador ao utilizar o marcador invoke_activity. Nesse caso, ao executar a invocação de atividade o sistema cria uma instância da atividade do tipo fornecido em type, identifica a instância com o nome dado por name e invoca a instância.

Da mesma forma que os controladores, os elementos de invocação de processo e de invocação de atividade trocam dados com o elemento invocador por meio de parâmetros atuais de entrada (in_parameter) e saída (out_parameter). Os parâmetros de entrada e saída devem ter o mesmo nome, dado pelos seus respectivos atributos name, que os respectivos parâmetros formais declarados nos tipos de processos e atividades invocados. Da mesma forma, os objetos, passados ou recebidos, por valor ou referência, para a instância de processo ou atividade invocada, por meio de cada parâmetro de entrada ou saída, devem ter o mesmo tipo declarado no correspondente parâmetro formal de entrada ou saída do tipo de processo ou atividade referenciada. Um dado a ser passado por um parâmetro atual de entrada é definido pelo valor da expressão do seu atributo value. Um dado é recebido pelo parâmetro atual de saída pela variável identificada no seu atributo value, que pode ser uma variável local ou campo de objeto.

Na invocação de uma atividade, é possível definir no atributo retries o número máximo de retentativas de invocação de uma atividade para que a execução desta seja bem sucedida, sem levantar exceções. Uma atividade é considerada falha quando ocorre uma exceção de sistema ou de aplicação na execução desta atividade e esta exceção não é tratada pela própria atividade. Entretanto, esse mecanismo de tolerância a falhas é utilizado para retentar a invocação da atividade até o limite dado pelo atributo apenas em casos de exceções de sistema. Nas exceções de aplicação, o mecanismo deixa de retentar a invocação da atividade e repassa a exceção para cima na hierarquia do processo. A finalidade desse mecanismo é lidar com falhas transientes, nas quais o sistema pode au-

tomaticamente se restaurar e permitir a execução da atividade. As falhas de aplicação normalmente necessitam de intervenção humana para serem eliminadas e por isso são repassadas. A retentativa de execução pode ocorrer tanto na máquina onde ela foi executada anteriormente ou em outra máquina selecionada pelo sistema. De forma alternativa, o programador pode especificar o tempo em milisegundos que o sistema deve ficar retentando executar a atividade por meio do atributo retry_time. Se os dois atributos forem especificados, devem ser interrompidas as retentativas no momento em que a primeira das condições for satisfeita.

No caso de a execução da atividade não ser bem sucedida, antes de ser repassada a exceção para o elemento invocador, a atividade somente é abortada se não for definida como recuperável (sem o marcador recoverable na definição da atividade). Caso a atividade seja recuperável ela não é automaticamente abortada, assim o usuário pode tentar sanar a falha ocorrida, por exemplo copiando um arquivo faltante para o domínio do processo, ou abortá-la manualmente.

O atributo replicas do marcador de invocação de atividades possibilita ao programador especificar o número máximo de réplicas de uma atividade que devem ser executadas em diferentes máquinas da grade, utilizando os mesmos parâmetros de entrada. Combinado com o atributo retries faz com que o sistema execute o número desejado de réplicas, se existirem recursos suficientes, até o número especificado de tentativas, reiniciando a execução das réplicas quando todas tiverem falhado. O sistema deve considerar a invocação bem sucedida quando uma réplica é executada integralmente e retorna os seus resultados, da mesma forma que em [Yer03]. As vantagens das réplicas são duas: provável diminuição do tempo de invocação de uma atividade, pois as réplicas são executadas em recursos diversos com disponibilidade variável, conhecida precisamente somente em tempo de execução; e redução da possibilidade de falha da invocação, uma vez que falhas locais são ignoradas, bastando somente uma réplica retornar o resultado. Entretanto, as réplicas podem não prover a redução do tempo de invocação quando elas acessarem repositórios ou máquinas em comum e quando houver falhas de aplicação. Ademais, as réplicas de atividades transacionais em execução após o retorno da primeira réplica devem ser abortadas, o que pode ocasionar uma sobrecarga no sistema.

4.3.7 Código

```
CODE ::=
  <CODE>
    SCRIPT
  </CODE>
```

O último elemento de processo, também encontrado em atividades, é o code. No contexto de um processo ou controlador, esse elemento serve para especificar código (*script*) Java para realizar computações simples; obtenção, distribuição, agregação e transformações de dados; invocações a métodos de bibliotecas importadas; e reflexão computacional. Por exemplo, ele pode servir para calcular faixas de números para distribuir para atividades responsáveis por encontrar números primos nestas faixas. O código (*script*) Java é especificado no bloco code. No contexto de um processo ou controlador, ele é atualmente interpretado por uma biblioteca BeanShell [Nie07]. O código pode utilizar variáveis do controlador ou do processo, no qual ele está especificado, para realizar as computações e trocar dados.

No contexto de uma atividade, o código Java pode ser utilizado para realizar qualquer computação ou interação com serviços e banco de dados. Ele é compilado diretamente para *bytecode* a fim de o sistema conseguir um maior desempenho na execução da atividade. A execução de uma atividade, dessa forma, pode ser tão eficiente quanto executáveis compilados a partir da linguagem C.

4.3.8 Invocação de Serviço e de Objeto Remoto

```
INVOKE_WEB_SERVICE ::=
  <INVOKE_WEB_SERVICE URL="URL" PORT="IDENT" OPERATION="IDENT"/>
    (<IN_PARAMETER NAME="IDENT" VALUE="EXPRESSION"/>)*
    (<OUT_PARAMETER NAME="IDENT" VALUE="IDENT"/>)*
  </INVOKE_WEB_SERVICE>
```

A linguagem XSPL permite a invocação explícita de operações de serviços *web* por meio do elemento invoke_web_service. Para isso, deve-se passar a referência URL ao serviço por meio do atributo url, a identificação da porta, ou interface, que contém a operação por meio do atributo port, e o nome da operação a ser invocada nessa porta por meio do atributo operation. Por padrão deve ser utilizado SOAP (*Simple Object Access Protocol*) [Con03] sobre HTTP (*HyperText Transfer Protocol*) para invocar os serviços *web*. Os parâmetros de entrada e saída da operação são passados e obtidos, respectivamente, por meio dos elementos in_parameter e out_parameter. Para isso, na invocação o sistema deve criar uma mensagem SOAP com os dados da operação e parâmetros atuais de entrada e invocar o serviço *web* na referência especificada; e depois esperar a resposta como uma mensagem de resultado SOAP, passando os valores retornados para as variáveis identificadas nos parâmetros de saída. A especificação da mensagem SOAP a ser gerada é obtida por meio do documento WSDL (*Web Service Description Language*) da referência URL. Se houver erro na especificação da referência URL, da operação, dos parâmetros de entrada ou de

saída, ou no tipo dos objetos passados ou retornados, a invocação de serviço *web* levanta uma exceção correspondente.

```
INVOKE_OBJECT ::=
  <INVOKE_OBJECT URL="URL" TYPE="TYPE" OPERATION="IDENT">
    (<IN_PARAMETER NAME="IDENT" VALUE="EXPRESSION"/>)*
    [<OUT_PARAMETER NAME="IDENT" VALUE="IDENT"/>]
  </INVOKE_OBJECT>
```

A linguagem também permite especificar a invocação explícita de objetos remotos utilizando o protocolo RMI (*Remote Method Interface*) do padrão Java. A referência ao objeto é obtida por meio da URL, definida no atributo url do elemento, que identifica unicamente essa instância no serviço de diretórios do domínio (grupo) local, acessado por meio do LDAP (*Lightweight Directory Access Protocol*) [Zei06]. O tipo do objeto invocado, previamente conhecido, deve ser definido no atributo type, para que o sistema possa identificá-lo e invoque suas instâncias. O nome da operação a ser invocada no objeto referenciado é definido pelo atributo operation. Uma operação RMI pode receber vários parâmetros de entrada, mas pode retornar somente um resultado. Por isso, podem ser utilizados, no elemento invoke_object), zero ou mais parâmetros de entrada (in_parameter), mas no máximo um parâmetro de saída (out_parameter), para receber o resultado. Caso o objeto não seja encontrado, ou haja erro na especificação da operação, dos parâmetros ou do retorno, ou no tipo dos objetos passados ou retornados, a invocação de objeto remoto levanta uma exceção correspondente.

4.3.9 Requisitos de Execução

```
EXECUTION_REQUIREMENTS ::=
  [PROCESSING_REQUIREMENT]
  [MEMORY_REQUIREMENT]
  [DISK_REQUIREMENT]
  [TRANSFERENCE_REQUIREMENT]
  [FILE_REQUIREMENTS]
  [DATASOURCE_REQUIREMENTS]
  [GENERIC_REQUIREMENTS]

PROCESSING_REQUIREMENT ::=
  <PROCESSING_REQUIREMENT MIN_POWER="INT_EXPRESSION"
    [MAX_POWER="INT_EXPRESSION"]/>
```

```
MEMORY_REQUIREMENT ::=
  <MEMORY_REQUIREMENT MIN_SIZE="INT_EXPRESSION"
    [MAX_SIZE="INT_EXPRESSION"]/>

DISK_REQUIREMENT ::=
  <DISK_REQUIREMENT MIN_SIZE="INT_EXPRESSION"
    [MAX_SIZE="INT_EXPRESSION"]/>

TRANSFERENCE_REQUIREMENT ::=
  <TRANSFERENCE_REQUIREMENT MIN_SPEED="INT_EXPRESSION"
    [MAX_SPEED="INT_EXPRESSION"]/>

FILE_REQUIREMENTS ::=
  <FILE_REQUIREMENTS>
    (<FILE_REQUIREMENT NAME="STRING_EXPRESSION"/>)*
  </FILE_REQUIREMENTS>

DATASOURCE_REQUIREMENTS ::=
  <DATASOURCE_REQUIREMENTS>
    (<DATASOURCE_REQUIREMENT NAME="STRING_EXPRESSION"/>)*
  </DATASOURCE_REQUIREMENTS>

GENERIC_REQUIREMENTS ::=
  <GENERIC_REQUIREMENTS>
    (<GENERIC_REQUIREMENT TYPE="IDENT" VALUE="STRING_EXPRESSION"/>)*
  </GENERIC_REQUIREMENTS>
```

Uma atividade geralmente necessita de alguns recursos específicos para a sua execução, sejam eles arquivos, fontes de dados, capacidade computacional, de transferência, de armazenamento ou outros. Essas necessidades são observadas nas atividades constituídas de código para realização de computação, ao invés de atividades para invocação de serviço *web* ou objeto. Os requisitos para a execução de uma atividade podem ser classificados em: requisitos por capacidade computacional mínima, tamanho mínimo de memória volátil ou de disco, velocidade mínima de transferência, acesso a determinados arquivos e fonte de dados, ou outros recursos definidos pelo usuário. Na especificação de requisitos, pode-se definir também a capacidade computacional máxima, tamanho máximo de memória volátil ou de disco, e velocidade máxima de transferência, de forma a permitir que o *middleware* não reserve recursos em excesso para a execução da atividade.

Uma atividade para executar com o tempo esperado pode necessitar de uma certa capacidade mínima de processamento, velocidade de comunicação mínima e quantidades de memória volátil rápida, RAM, e de disco disponíveis na máquina na qual ela será alocada. O requisito de poder mínimo de processamento pode ser especificado no atributo min_power do marcador processing_requirement, em milhares de instruções básicas por segundo. O requisito de quantidade de memória mínima pode ser declarado utilizando-se o marcador memory_requirement. O valor mínimo em kilobytes é definido no atributo min_size. Da mesma forma, uma atividade pode necessitar de espaço em disco local, ou outra forma de memória persistente local, para criar certos arquivos temporários, e para armazenar arquivos de entrada e de resultados. O espaço mínimo necessário de disco, ou de memória persistente, por uma atividade pode ser declarado utilizando-se o marcador disk_requirement. O atributo min_size define o tamanho mínimo em megabytes. Por último, o requisito de velocidade mínima de comunicação pode ser especificado no atributo min_speed do marcador transference_requirement, em *kilobytes* por segundo.

De forma complementar, para evitar a alocação em excesso de recursos para uma atividade, pode-se especificar a capacidade de processamento, a velocidade de comunicação e as quantidades de memória volátil e de disco (persistente) máximas necessárias para a execução da atividade. O poder máximo de processamento requisitado é determinado pelo atributo max_power do marcador processing_requirement; o tamanho máximo de memória volátil é especificado pelo atributo max_size do marcador memory_requirement; o tamanho máximo de memória de disco (persistente), de forma similar, é determinado pelo atributo correspondente do marcador disk_requirement; e a velocidade máxima de comunicação é definida pelo atributo max_speed do marcador transference_requirement. A unidade utilizada para a determinação das grandezas de cada atributo é a correspondente do atributo de requisitos mínimos de cada marcador.

O requisito de acesso a arquivos locais ou remotos é declarado por meio do bloco file_requirements. Os nomes do arquivos a serem acessados são definidos pelos marcadores file_requirement no seu corpo. Cada arquivo necessário é especificado pelo atributo name deste marcador. Os arquivos a serem utilizados podem ser tanto de dados quanto de código, tais como bibliotecas com classes e executáveis a serem utilizados pela computação a ser realizada. Se um arquivo não for encontrado no sistema de arquivos local do recurso onde a atividade será executada, o sistema pode copiar esse arquivo do domínio da especificação do processo, diretamente da máquina origem que contém a descrição do processo para a máquina selecionada para executar a atividade.

O bloco datasource_requirements é utilizado para declarar o requisito de acesso a fontes de dados. O nome de cada fonte de dados a ser acessada é definido pelo atributo name de cada marcador datasource_requirement contido neste bloco. Uma vez dado o nome de uma fonte de dados, o sistema de suporte do domínio (grupo) local pode encontrá-la por meio

de seu serviço de diretórios, utilizando LDAP. A fonte de dados geralmente se refere a uma base de um banco de dados relacional contido na instituição que executará a atividade. Uma vez com uma referência para a fonte, a atividade pode obter conexões para essa base e realizar consultas de leitura e modificação com esta por meio da interface de programação JDBC (*Java Database Connectivity*) [And06]. Se a atividade for declarada transacional (atômica), é responsabilidade desta delimitar o início e o final da transação com *begin* e *commit*, bem como realizar o aborto da transação com o *rollback* automaticamente ou quando requisitado pelo sistema. Se a atividade transacional acessar mais de uma fonte de dados, é necessário utilizar JTA (*Java Transaction API*) [CM02] para delimitar e abortar as transações. Essa interface de programação é responsável por executar o protocolo *two phase commit* sobre as fontes de dados para garantir que todos ou nenhum dos dados sejam atualizados.

Além dos requisitos acima enumerados, uma atividade pode necessitar de que o recurso a ser utilizado possua outras características, tal como pacotes de software, agente humano com habilidades ou papel determinado, disponibilidade média da máquina, confiabilidade e segurança. Todas essas características podem ser definidas de maneira genérica, pelo bloco generic_requirements. Cada característica é definida por um marcador generic_requirement contido no corpo deste bloco. O atributo type de cada marcador é utilizado para especificar o tipo da característica necessária e o atributo value do marcador, para especificar o valor dessa característica. Esses tipos de características são definidas por recurso e o sistema não possui nenhum dicionário de meta-dados que enumera as características existentes.

4.3.10 Estimativas de Execução

```
EXECUTION_ESTIMATIONS ::=
  [PROCESSING_ESTIMATION]
  [TRANSFERENCE_ESTIMATION]
  [IDLE_TIME_ESTIMATION]

PROCESSING_ESTIMATION ::=
  <PROCESSING_ESTIMATION NUM_INSTRUCTIONS="INT_EXPRESSION"/>

TRANSFERENCE_ESTIMATION ::=
  <TRANSFERENCE_ESTIMATION SIZE="INT_EXPRESSION"/>

IDLE_TIME_ESTIMATION ::=
  <IDLE_TIME_ESTIMATION TIME="INT_EXPRESSION"/>
```

A fim de auxiliar o serviço de escalonamento no cálculo de tempo de execução das atividades, a especificação de cada atividade pode fornecer uma estimativa de qual é o esforço de computação necessário para executá-la inteiramente, de qual é o tamanho total das mensagens trocadas por essa com o processo ou controlador que a invocou, e de quanto tempo ela fica ociosa esperando por respostas na comunicação com outras entidades.

O esforço estimado de processamento para executar uma certa atividade é declarado utilizando-se o marcador processing_estimation. O valor do atributo num_instructions deve fornecer quantas mil instruções básicas são necessárias para se executar a fim de concluir a computação descrita pelo código da atividade. Esse valor pode ser obtido por meio do resultado de uma expressão inteira em Java, calculada no momento da instanciação do processo que contém a atividade. Dessa forma, é possível estimar mais precisamente o tamanho da atividade, dados os parâmetros de entrada do processo. Com base nessa informação e sabendo qual é o poder de processamento dos recursos computacionais em número de milhares de instruções básicas executadas por segundo, o serviço de escalonamento pode determinar qual é a duração esperada de uma atividade em cada recurso.

A estimativa de tamanho total das mensagens utilizadas para a comunicação entre uma atividade e o controlador ou processo que a invocou pode ser declarada utilizando-se o marcador transference_estimation. O valor do atributo size deve conter quantos *kilobytes* são trocados entre a atividade e o seu invocador durante toda a sua execução, seja pelos parâmetros de entrada e saída (na invocação e no retorno dos resultados, respectivamente), seja pelo uso de objetos compartilhados do invocador. Esse valor também pode ser calculado por uma expressão inteira em Java no momento de instanciação do processo que contém a atividade, permitindo-se estimar com maior precisão o tamanho total das mensagens na comunicação dados os parâmetros iniciais do processo. Com base nessa informação e na velocidade do canal de comunicação entre o recurso que contém a atividade e o que contém o invocador, em *kilobytes* por segundo, o serviço de escalonamento pode estimar o tempo de comunicação entre a atividade e o seu invocador.

Na execução de qualquer atividade, além do tempo gasto com a execução das instruções e com a comunicação, há o tempo ocioso na espera pelo acesso a um arquivo local ou a uma fonte de dados, assim como por outras operações do sistema operacional ou do *middleware*. Para fornecer uma estimativa desse tempo, o usuário pode utilizar o marcador idle_time_estimation. O valor do atributo time, contendo o tempo ocioso em segundos, pode ser calculado como uma expressão inteira, da mesma forma que o realizado nos elementos anteriores. Para a previsão do tempo de duração dessa atividade, o escalonador simplesmente adiciona esse tempo aos demais calculados, dependentes dos recursos e dos canais de comunicação. No caso de uma atividade estimar que ela será executada durante um determinado tempo fixo, em vez de o programador especificar processing_estimation e

tranference_estimation, ele pode declarar simplesmente todo o tempo previsto de execução em idle_time_estimation.

4.3.11 Atributos para Tolerância a Falhas

```
FAULT_TOLERANCE_ATTRIBUTES ::=
  [RECOVERABLE]
  [REPLICABLE]
  [TRANSACTIONAL]

RECOVERABLE ::=
  <RECOVERABLE/>

REPLICABLE ::=
  <REPLICABLE />

TRANSACTIONAL ::=
  <TRANSACTIONAL/>
```

Os atributos para tolerância a falhas são declarados na especificação de uma atividade. Eles servem para informar ao *middleware* como ele deve cooperar com a atividade para tratar das falhas ocorridas nesta atividade de forma a aproveitar a computação anteriormente realizada e manter a consistência dos dados utilizados, aumentando o desempenho da aplicação e tornando-a confiável.

O marcador recoverable é utilizado para indicar que a atividade é recuperável, ou seja, que, em caso de falhas de sistema ou de aplicação não tratadas pela atividade, ela pode se recuperar, a partir do último ponto consistente antes da falha, e continuar a sua execução. É responsabilidade da atividade manter o último estado consistente dos dados e da execução da atividade (*checkpoint*), que pode ser armazenado em memória volátil ou persistente, e depois continuar a sua execução a partir deste ponto. Para a atividade se recuperar de falhas da máquina ou do gerenciador de atividades que a estavam executando e que depois precisem de reinicialização, é interessante manter o estado consistente da execução em memória persistente de tempos em tempos, que depois pode ser lido na reativação da atividade pelo *middleware*, após a sua própria reativação na máquina. O sistema pode notificar a atividade recuperável a suspender a sua execução por meio da operação *suspend* do objeto que representa a atividade, em caso de exceção não tratada pela atividade, levantada para o invocador. Nesse momento, a atividade pode armazenar o estado de seus dados e de execução em memória, volátil ou persistente. Depois de tratada a exceção, o *middleware* pode solicitar que ela continue a execução por

meio da operação *resume*. Essa última operação invoca o método *run* da atividade que contém o código compilado da atividade. Esse código deve sempre testar em que ponto de execução a atividade está, de forma iterativa, e continuar a partir desse ponto. Por último, o *middleware* ou a própria aplicação pode invocar a operação *abort* da atividade recuperável, de modo a interromper a sua execução. Nesse caso, a atividade descarta o seu estado armazenado em memória e recomeça a sua execução no caso de invocação de sua operação *run*.

As atividades recuperáveis, assim como os controladores e processos, que são por definição recuperáveis, suspendem a sua execução automaticamente após repassar a exceção, gerada pela falha ocorrida, para o elemento de processo invocador. Assim, após resolvidas as falhas transientes, esses elementos podem continuar sendo executados, na mesma máquina ou em outra. As falhas podem ser resolvidas, por exemplo, reiniciando a máquina ou o processo falho, copiando o arquivo faltante para o domínio da aplicação ou para a máquina da atividade, ou até modificando a atividade e recompilando-a para posterior recarregamento dos seus *bytecodes* na máquina virtual do gerenciador de atividades. A vantagem da atividade recuperável é o aproveitamento da computação já realizada por uma atividade longa, reduzindo o tempo da execução do processo em caso de falhas e, portanto, aumentando o desempenho da aplicação e disponibilidade do sistema.

O marcador replicable indica que instâncias dessa atividade podem ser criadas em mais de uma máquina pela grade, tomando-se os mesmos parâmetros de entrada. Esse atributo de tolerância a falhas garante que as instâncias estão conscientes da possível existência de réplicas e que devem realizar a sua execução de forma consistente mesmo na presença dessas réplicas. Por exemplo, se uma atividade for idempotente, ela pode ser replicada de forma trivial. Se não for, somente uma pode ser terminada com seus resultados sendo considerados. O uso das propriedades transacionais de serialização de execução, mesmo que relaxadas, dependendo da necessidade de cada atividade, permite que atividades não idempotentes sejam replicáveis, desde que somente uma termine de forma bem sucedida (*commited*).

O marcador transactional é utilizado para indicar que a atividade é transacional, i.e., tem a propriedade atômica "tudo ou nada" na qual todas as modificações realizadas são concluídas ou nenhuma delas. No sistema, é responsabilidade do programador delimitar a transação que corresponde à atividade, iniciando a transação no início da atividade e concluindo-a de forma bem sucedida (*commit*) caso a atividade execute corretamente. Se somente uma fonte de dados relacional é utilizada, por meio de JDBC (*Java Database Connectivity*), pode-se utilizar a operação *begin* da transação no seu início e *commit* no final, que o *driver* correspondente persistirá corretamente as modificações dos dados utilizados. Para mais de uma fonte de dados ou recurso transacional, faz-se necessário utilizar a interface de programação JTA (*Java Transaction API*) para as mesmas operações. Se

a atividade não for recuperável e ocorrer uma exceção não tratada, o programador deve abortar as alterações com a operação *abort* da transação. Para o caso de uma atividade recuperável, quem decide se a atividade deve ser abortada ou não é o sistema ou o usuário, em última instância. No caso de decisão por abortar, o sistema invoca a operação *abort* do objeto atividade, modificando o estado da atividade para *aborting*. Caso isso ocorra, esta atividade deve abortar a transação vigente por meio de um código do usuário que testa essa condição.

4.4 Exemplos de Processos

Nessa seção apresentaremos dois exemplos de processos especificados com a linguagem XSPL com objetivo de ilustrar sua expressividade e simplicidade. O primeiro exemplo apresentado é de uma aplicação científica para cálculo de números primos em uma determinada faixa de números. O segundo exemplo descreve uma aplicação comercial para verificação da consistência dos dados registrados, nos fornecedores e nos compradores, sobre transações de venda de produtos dos primeiros para os segundos.

4.4.1 Busca de Números Primos

Nessa seção é apresentada uma aplicação matemática (científica) de busca de números primos. Esse aplicação exemplo recebe uma certa faixa de número inteiros e retorna um conjunto de números primos contidos nessa faixa. Ela é implementada em XSPL como um processo que usa um controlador paralelo para disparar várias atividades que buscam números primos em subfaixas determinadas. O documento XML que descreve ambos o processo e a atividade da aplicação é apresentado abaixo.

```
1  <PROCESS_TYPE NAME="FindPrimes">
2    <IN_PARAMETER TYPE="int" NAME="min"/>
3    <IN_PARAMETER TYPE="int" NAME="max"/>
4    <IN_PARAMETER TYPE="int" NAME="numPrimes"/>
5    <IN_PARAMETER TYPE="int" NAME="numActs"/>
6    <OUT_PARAMETER TYPE="RemoteCollection" NAME="primes"/>
7    <BODY>
8      <CODE>
9        primes = new RemoteHashSet();
10     </CODE>
11     <PARFOR NAME="parfor">
12       <IN_PARAMETER TYPE="int" NAME="min" VALUE="min"/>
```

```
13      <IN_PARAMETER TYPE="int" NAME="max" VALUE="max"/>
14      <IN_PARAMETER TYPE="int" NAME="numPrimes" VALUE="numPrimes"/>
15      <IN_PARAMETER TYPE="int" NAME="numActs" VALUE="numActs"/>
16      <IN_PARAMETER TYPE="RemoteCollection" NAME="primes"
17        VALUE="primes"/>
18      <ITERATE VAR="counter" BEGIN="0" END="numActs - 1">
19        <CODE>
20          int rangeSize = (max - min + 1) / numActs;
21          int minNum = rangeSize * counter + min;
22          int maxNum = minNum + rangeSize - 1;
23          if (counter == numActs - 1) maxNum = max;
24        </CODE>
25        <INVOKE_ACTIVITY TYPE="FindPrimes" NAME="findPrimes">
26          <IN_PARAMETER NAME="min" VALUE="minNum"/>
27          <IN_PARAMETER NAME="max" VALUE="maxNum"/>
28          <IN_PARAMETER NAME="numPrimes" VALUE="numPrimes"/>
29          <IN_PARAMETER NAME="primes" VALUE="primes"/>
30        </INVOKE_ACTIVITY>
31      </ITERATE>
32    </PARFOR>
33  </BODY>
34 </PROCESS_TYPE>
35
36 <ACTIVITY_TYPE NAME="FindPrimes">
37  <IN_PARAMETER TYPE="int" NAME="min"/>
38  <IN_PARAMETER TYPE="int" NAME="max"/>
39  <IN_PARAMETER TYPE="int" NAME="numPrimes"/>
40  <IN_PARAMETER TYPE="RemoteCollection" NAME="primes"/>
41  <CODE>
42    for (int num = min; num <= max; num++) {
43      // stop if required number of primes was found
44      if (primes.size() >= numPrimes)
45        break;
46      boolean prime = true;
47      for (int i = 2; i <= (int)Math.sqrt(num); i++) {
48        if (num % i == 0) { prime = false; break; }
49      }
50      if (prime) {
```

```
51          primes.add(new Integer(num));
52        }
53      }
54    </CODE>
55 </ACTIVITY_TYPE>
```

No início do documento é definido o processo que encontra números primos, denominado FindPrimes (linhas 1 – 34). Ele recebe, por meio dos seus parâmetros de entrada: uma faixa de inteiros, definida de min até max, na qual devem ser buscados os números primos; a quantidade de números primos a serem obtidos, em numPrimes; e o número de atividades a serem executadas para resolver esse problema, em numActs (linhas 2 – 5). No final da execução, os números primos são retornados em uma coleção a ser referenciada pelo parâmetro de saída denominado primes (linha 6).

Esse processo contém um controlador parfor que tem a função de disparar o número determinado de atividades paralelas para encontrar números primos (linhas 11 – 32). O controlador itera de 0 até numActs - 1, onde numActs determina o número de atividades, e, em cada iteração, ele divide a faixa de primos em subfaixas de mesmo tamanho e dispara uma atividade paralela para encontrar números primos nessa subfaixa específica (linhas 18 – 31). Na invocação de cada atividade a ser executada em paralelo são passados por parâmetro: a subfaixa de números inteiros sobre a qual a atividade deve encontrar números primos, definida de minNum até maxNum; a quantidade global de números primos a serem encontrados em numPrimes; e um objeto compartilhado do tipo coleção, por referência, em primes, para armazenar os primos encontrados e controlar se a quantidade requerida de números primos foi atingida globalmente (linhas 25 – 30).

Depois do processo é definida a atividade FindPrimes, usada para encontrar primos em cada subfaixa (linhas 36 – 55). Cada instância recebe, por meio de seus parâmetros de entrada, a faixa de números sobre o qual deve encontrar os números primos, a quantidade de números primos a serem encontrados pelo processo e, por referência, o objeto compartilhado usado para armazenar os números primos encontrados (linhas 37 – 40), passados pela invocação descrita acima. Dentro do bloco code há um código Java, relativamente simples, para encontrar números primos, que itera sobre a faixa passada e verifica se o número corrente é primo (linhas 41 – 54). Em cada iteração o código verifica se a quantidade de números primos, inseridos na coleção por todas as atividades concorrentes, foi alcançada e termina a atividade se verdadeiro.

Uma das vantagens da adoção de controladores na linguagem é a explicitação da estrutura e do comportamento do processo para que o *middleware* Xavantes possa determinar o ponto atual de execução do processo e implementar recuperação automática, monitoramento e modificação dinâmica de processos; e escalonar, alocar e distribuir elementos de processo previamente, quando há recursos disponíveis, promovendo o aumento do de-

sempenho da execução. Ademais, devido à sua natureza estruturada, os controladores podem ser compostos e o *middleware* pode tirar vantagem disso para distribuir hierarquicamente controladores aninhados para máquinas diferentes da grade, permitindo uma maior escalabilidade, desempenho e tolerância a falhas.

4.4.2 Cruzamento de Transações de Venda entre Contribuintes

Nessa seção é apresentada uma aplicação tributária (de negócio) de cruzamento entre informações de venda, obtidas dos contribuintes fornecedores, e informações de compra, obtidas dos contribuintes clientes, para fins de verificação de fraudes nas escriturações dos respectivos contribuintes. A aplicação é composta de um processo, que controla a obtenção de informações de venda e compra dos contribuintes fornecedores e clientes, respectivamente, e o cruzamento dessas informações; e três atividades que realizam essas tarefas. O processo, descrito em XML, é apresentado abaixo.

```
1  <PROCESS_TYPE NAME="MatchTransactions">
2    <IN_PARAMETER TYPE="List" NAME="suppliers"/>
3    <IN_PARAMETER TYPE="List" NAME="clients"/>
4    <IN_PARAMETER TYPE="int" NAME="year"/>
5    <OUT_PARAMETER TYPE="TransTable" NAME="diffTable"/>
6    <VAR TYPE="TransTable" NAME="sellTable"/>
7    <VAR TYPE="TransTable" NAME="buyTable"/>
8    <BODY>
9      <CODE>
10       sellTable = new TransTable();
11       buyTable = new TransTable();
12       diffTable = new TransTable();
13     </CODE>
14     <PAR NAME="par1">
15       <IN_PARAMETER TYPE="List" NAME="suppliers"
16         VALUE="suppliers"/>
17       <IN_PARAMETER TYPE="List" NAME="clients"
18         VALUE="clients"/>
19       <IN_PARAMETER TYPE="int" NAME="year"
20         VALUE="year"/>
21       <IN_PARAMETER TYPE="TransTable" NAME="sellTable"
22         VALUE="sellTable"/>
23       <IN_PARAMETER TYPE="TransTable" NAME="buyTable"
24         VALUE="buyTable"/>
```

```
25      <START>
26        <PARFOREACH NAME="parforSup">
27          <IN_PARAMETER TYPE="List" NAME="suppliers"
28            VALUE="suppliers"/>
29          <IN_PARAMETER TYPE="int" NAME="year"
30            VALUE="year"/>
31          <IN_PARAMETER TYPE="TransTable" NAME="sellTable"
32            VALUE="sellTable"/>
33          <ITERATE VAR="supplier" COLLECTION="suppliers">
34            <INVOKE_ACTIVITY TYPE="GetSellTransactions"
35              NAME="getSellTrans">
36              <IN_PARAMETER NAME="supplier" VALUE="supplier"/>
37              <IN_PARAMETER NAME="year" VALUE="year"/>
38              <IN_PARAMETER NAME="sellTable" VALUE="sellTable"/>
39            </INVOKE_ACTIVITY>
40          </ITERATE>
41        </PARFOREACH>
42      </START>
43      <START>
44        <PARFOREACH NAME="parforCli">
45          <IN_PARAMETER TYPE="List" NAME="clients"
46            VALUE="clients"/>
47          <IN_PARAMETER TYPE="int" NAME="year"
48            VALUE="year"/>
49          <IN_PARAMETER TYPE="TransTable" NAME="buyTable"
50            VALUE="buyTable"/>
51          <ITERATE VAR="client" COLLECTION="clients">
52            <INVOKE_ACTIVITY TYPE="GetBuyTransactions"
53              NAME="getBuyTrans">
54              <IN_PARAMETER NAME="client" VALUE="client"/>
55              <IN_PARAMETER NAME="year" VALUE="year"/>
56              <IN_PARAMETER NAME="buyTable" VALUE="buyTable"/>
57            </INVOKE_ACTIVITY>
58          </ITERATE>
59        </PARFOREACH>
60      </START>
61    </PAR>
62    <PARFOREACH NAME="parforDiff">
```

```
63        <IN_PARAMETER TYPE="List" NAME="suppliers"
64          VALUE="suppliers"/>
65        <IN_PARAMETER TYPE="List" NAME="clients"
66          VALUE="clients"/>
67        <IN_PARAMETER TYPE="TransTable" NAME="sellTable"
68          VALUE="sellTable"/>
69        <IN_PARAMETER TYPE="TransTable" NAME="buyTable"
70          VALUE="buyTable"/>
71        <IN_PARAMETER TYPE="TransTable" NAME="diffTable"
72          VALUE="diffTable"/>
73        <ITERATE VAR="supplier" COLLECTION="suppliers">
74          <INVOKE_ACTIVITY TYPE="CompareTransactions"
75            NAME="compTrans">
76            <IN_PARAMETER NAME="supplier" VALUE="supplier"/>
77            <IN_PARAMETER NAME="clients" VALUE="clients"/>
78            <IN_PARAMETER NAME="sellTable" VALUE="sellTable"/>
79            <IN_PARAMETER NAME="buyTable" VALUE="buyTable"/>
80            <IN_PARAMETER NAME="diffTable" VALUE="diffTable"/>
81          </INVOKE_ACTIVITY>
82        </ITERATE>
83      </PARFOREACH>
84    </BODY>
85 </PROCESS_TYPE>
```

O processo de cruzamento entre informações de venda e de compra é denominado MatchTransactions (linhas 1 – 85). Ele recebe uma lista de fornecedores pelo parâmetro suppliers, uma de clientes pelo parâmetro clients e um ano pelo parâmetro year, e retorna ao final de sua execução, pelo parâmetro diffTable, uma tabela com todas as informações de notas fiscais escrituradas de forma divergente entre cada par fornecedor-cliente no ano especificado (linhas 2 – 5). Além dos parâmetros, são declaradas duas variáveis no processo: sellTable, contendo todas as informações de venda pelos fornecedores listados, e buyTable, contendo todas as informações de compra pelos clientes listados (linhas 6 – 7). Tanto o parâmetro diffTable, quanto as variáveis sellTable e buyTable, referenciam objetos compartilhados do tipo TransTable, que são tabelas transacionais e remotas utilizadas para armazenar registros de escrituração de notas fiscais. Esses objetos compartilhados são utilizados remotamente pelos controladores e atividades do processo.

O corpo do processo contém um bloco de código Java para a criação das variáveis compartilhadas no contexto do processo (linhas 9 – 13), um controlador paralelo simples (*par*) para a obtenção das informações de venda e de compra por contribuintes, referentes

ao ano determinado, de forma paralela e distribuída (linhas 14 – 61), e um controlador paralelo iterativo (*parforeach*) para o cruzamento de informações efetivamente (linhas 62 – 83). Esses elementos de processo são executados em seqüência.

O primeiro controlador paralelo, denominado par1, recebe a lista de fornecedores, a lista de clientes, o ano, a tabela vazia de informações de venda e a tabela vazia de informações de compra, nessa ordem, nos seus parâmetros de entrada (linhas 15 – 24). No seu corpo ele dispara dois controladores paralelos iterativos: um para obter as informações de venda dos contribuintes fornecedores no ano determinado, denominado parforSup (linhas 26 – 41); e outro para obter as informações de compra dos contribuintes clientes no mesmo período, denominado parforCli (linhas 44 – 59).

O controlador parforSup dispara várias invocações paralelas, uma para cada fornecedor da lista passada, à atividade GetSellTransactions (linhas 34 – 39). Essa atividade tem por finalidade obter as transações de venda de um determinado contribuinte, referentes a um determinado ano, e inserir essas informações em uma tabela de registros de escrituração de notas fiscais. Ela pode ser uma atividade comum a várias aplicações e implementada como uma invocação a um *web service* mais um código para inserção de informações obtidas na tabela passada. De forma similar, o controlador parforCli dispara várias invocações paralelas à atividade GetBuyTransactions (linhas 52 – 57), uma para cada cliente da lista passada. Essa atividade obtém as transações de compra de um determinado contribuinte, referentes a um determinado ano e insere essas informações em uma tabela de registros de escrituração de notas fiscais. Após a execução desses dois controladores, as variáveis sellTable e buyTable contêm, respectivamente, as informações de venda e de compra de todos os contribuintes fornecedores e clientes contidos nas listas suppliers e clients, referentes ao ano especificado.

O controlador parforDiff invoca paralelamente a execução de atividades de cruzamento de transações, CompareTransactions, uma para cada fornecedor (linhas 74 – 81). Essa atividade compara os registros da tabela de venda sellTable, relacionados a um determinado fornecedor supplier e diversos clientes clients, com os registros da tabela de compra buyTable relacionados a esses mesmos pares fornecedor-cliente. Os registros da tabela sellTable foram obtidos dos fornecedores e os registros da tabela buyTable, dos clientes, como visto anteriormente. A comparação é feita com base nos registros de cada nota fiscal escriturada no fornecedor e no cliente. A atividade compara o total do valor contábil dos registros de uma nota fiscal escriturada no fornecedor com o total do valor contábil dos registros de uma nota fiscal escriturada no cliente e faz o mesmo com o valor do imposto. Se o total do valor contábil for diferente ou se o total do valor do imposto for maior no cliente, esses registros são inseridos na tabela diffTable. Dessa forma, essa tabela de divergências contém todos os registros de notas fiscais com problemas de escrituração, que podem ter sido produzidas por fraudes na escrituração no fornecedor ou no cliente.

Esse exemplo demonstra como é simples especificar aplicações reais para a grade computacional, de forma a utilizar o poder de processamento e de armazenamento e comunicação de dados disponíveis nesta. O *middleware* Xavantes obtém essa especificação e utiliza as máquinas disponíveis para executar as atividades de obtenção de informações de transação e cruzamentos por fornecedor de forma distribuída, utilizando as informações contidas nas variáveis compartilhadas. Além disso, como será apresentado mais adiante, ele também controla a execução de forma a redistribuir atividades para obter melhor balanceamento de carga e restaura o processo em caso de falhas.

4.5 Resumo

A linguagem de especificação de processos XSPL (*Xavantes Structured Process Language*) permite descrever facilmente aplicações para grades computacionais em forma de processos estruturados. Os processos especificados com essa linguagem, como veremos no próximo capítulo, são executados pelo *middleware* Xavantes, que escalona, aloca, distribui e controla a execução dos elementos de processos pela grade computacional, de forma hierárquica, segundo a forma estruturada de cada processo. Os processos compõem e coordenaram atividades, que realizam as tarefas mais básicas da aplicação. Para controlar a ordem de execução das atividades e o fluxo de dados, são utilizados construtores estruturados conhecidos como controladores. Os controladores podem conter outros controladores, e invocações a atividades e a processos, de forma hierárquica, sendo que um controlador ordena a execução e repassa dados aos controladores, atividades e processos contidos neste. Existem controladores seqüenciais e paralelos, quanto ao paralelismo permitido, e simples, condicionais, iterativos e iterativo-condicionais, quanto ao tipo de fluxo de controle suportado. Os dados são passados do controlador para os seus elementos (controladores, atividades e processos invocados) por meio de parâmetros. Durante a execução, atividades podem trocar dados por meio de objetos compartilhados do controlador pai ou outro ascendente.

As principais vantagens dessa linguagem são:

simplicidade — por ser estruturada, a linguagem permite especificar aplicações paralelas de forma simples, como ocorre com linguagens genéricas de programação imperativas e estruturadas, mas de modo a aproveitar toda a informação e os recursos disponíveis em uma grade computacional ao invés de um simples computador, e de modo a permitir a execução de atividades paralelas ao invés de comandos em seqüência. Essa simplicidade se traduz numa fácil manutenção e monitoração das aplicações construídas, diferentemente das linguagens de workflow existentes, baseadas em *petri-nets*.

expressividade — a linguagem permite a especificação de qualquer tipo de aplicação paralela, porque ela contém construtores que permitem a especificação de qualquer tipo de fluxo de controle estruturado e porque há formas de mapear fluxos com dependências cruzadas, por meio de variáveis compartilhadas bloqueantes. A linguagem permite qualquer tipo de disparo e junção condicional, por meio de seus blocos join e terminate, e contém controladores em que é possível determinar o número de caminhos de execução (*threads*) a serem criados em tempo de execução. Novamente, essa abordagem suplanta a expressividade das linguagens de programação de *workflow*, que não permitem disparar um número dinamicamente determinado de atividades e não fornecem controle fino de disparo e sincronização de atividades.

modularização do controle — que permite ao *middleware*, por meio de reflexão computacional estrutural, conhecer a estrutura da aplicação, para poder, hierarquicamente, escalonar, distribuir, monitorar, controlar e recuperar, em caso de falhas, aplicações executadas sobre as grades computacionais.

controle fino — por meio do uso de *scripts* escritos em Java, o programador pode facilmente transformar dados advindos de uma atividade para a outra; inspecionar e alterar o estado das variáveis de um controlador ou processo, para controlar o seu fluxo de forma condicional expressiva; executar ações para tratar exceções ou junções; acessar bibliotecas escritas em Java; ou executar qualquer código que supra as necessidades da especificação fina do fluxo de controle e de dados.

reflexão computacional — que permite aos *scriptlets* conhecer o estado dos dados e do controle de cada processo ou controlador, e invocar operações sobre estes, como se fossem objetos regulares de primeira ordem, para executar ações correspondentes. Um exemplo é o tratamento de falhas, onde é necessário abortar automaticamente o controlador caso uma atividade essencial, controlada por esse, falhar.

temporização — em todos os controladores paralelos é possível especificar um tempo para executar os blocos de junção (*join*), requisito importante para verificar o estado do controlador, por meio de reflexão computacional, e decidir se este deve continuar sendo executado ou abortado. É permitido também controlar o tempo entre as iterações de controladores paralelos iterativos e o tempo de retentativa de invocação de atividades.

objetos compartilhados — os elementos de processo podem se comunicar por meio de parâmetros, com objetos passados por valor ou por referência, ou por meio de objetos compartilhados, instanciados em um controlador ou processo, e passados por referência. Um objeto compartilhado pode ser utilizado por atividades ou controladores

abaixo do elemento que o criou para transmitir dados entre esses elementos. Um objeto compartilhado é lido e modificado por meio de invocações de suas operações. A vantagem do uso do objeto compartilhado é a proximidade deste, instanciado no controlador, com os elementos que o utilizam, o que permite um maior desempenho da aplicação.

especificação de recursos necessários — nas atividades é possível especificar quais são as características de um recurso para que este execute uma atividade, tais como poder de processamento, velocidade de comunicação, memória volátil e persistente, mínimos e máximos, e arquivos e fonte de dados necessários.

estimativa de uso de recursos — em todos os elementos de processo é possível inserir uma estimativa de custo da comunicação entre esses e seus controladores. Nas atividades também é possível declarar uma estimativa do uso de esforço de processamento para a sua execução completa e do tempo de espera dessa atividade sem computação. Com essas meta-informações, o *middleware* pode estimar o tempo total de execução de cada elemento e escalonar esses elementos em cada recurso disponível de forma a otimizar o tempo de execução do processo.

tolerância a falhas — a linguagem contém elementos que especificam o modo de tratamento automático de falhas ocorridas em nível de atividade, tal como número de retentativas, para falhas mascaráveis, e blocos de tratamento de exceções geradas a partir de falhas de sistema e de aplicação, no caso destas não poderem ser mascaradas. Esses blocos podem ser especificados com *scriptlets* que verificam a exceção levantada e o estado atual do controlador, e tomam a ação correspondente para tratar a exceção. Em último caso, a exceção é levantada para usuário, que pode corrigir a falha ou modificar o processo para que a aplicação continue sendo executada. Todos os acessos a dados podem ser feitos por meio de transações, que garantem a integridade e consistência desses.

Uma vez programado o processo, é responsabilidade do *middleware* executá-lo conforme a especificação. A linguagem oferece uma série de elementos que devem ser interpretados conforme a semântica descrita nesse capítulo. O projeto e a implementação atual do *middleware* suportam grande parte dos elementos descritos nessa linguagem e oferecem contribuições importantes na execução dos processos em máquinas de vários grupos da grade, no escalonamento adaptativo para balanceamento de carga, e nos mecanismos de tolerância a falhas, conforme veremos nos capítulos posteriores.

Capítulo 5

Middleware para a Execução de Processos Estruturados

No capítulo anterior descrevemos a linguagem para a especificação de processos estruturados *Xavantes Structured Process Language (XSPL)*. Nesse capítulo apresentaremos os componentes do *middleware* Xavantes para a execução hierárquica de processos.

O *middleware* Xavantes é responsável por executar, monitorar e controlar processos estruturados especificados em XSPL (*Xavantes Structured Process Language*) sobre a grade computacional. Ele utiliza a árvore sintática do processo, formada pelo próprio processo na raiz, controladores no ramos e invocação a atividades e a processos nas folhas, para escalonar, distribuir, e executar *hierarquicamente* o processo. O processo é alocado a uma máquina, que escalona e distribui os seus elementos-filho para serem executados na mesma ou em outra máquina. Se o elemento filho é um controlador, ele escalona e distribui, também, seus elementos-filho para serem executados na mesma ou em outra máquina, e assim recursivamente, até que todos os elementos-filho sejam atividades, escalonadas e distribuídas para execução.

O *middleware* compreende ferramentas de especificação e monitoramento de processos e serviços de suporte a execução de processos em grades computacionais, para execução e monitoramento de processos em ambientes distribuídos, heterogêneos e autônomos. Os serviços de suporte a execução de processos na grade podem ser classificados em: serviço de informação e monitoramento de recursos na grade; serviço de execução de processos, controladores e atividades, com interpretação de processos, e escalonamento e distribuição de atividades e controladores; e serviço cliente de acesso aos recursos da grade e invocação de processos e atividades.

5.1 Arquitetura do *Middleware*

O *middleware* Xavantes se estabelece acima do nível dos recursos computacionais e de armazenamento de dados da grade, com a função de prover a comunicação e o acesso a esses recursos, manter a informação dinâmica sobre o estado dos recursos, e gerenciar a execução de processos e atividades. Na visão do *middleware*, a grade é divida em inúmeros grupos de recursos, cuja administração é autônoma — cada grupo pode controlar a disponibilidade de certos recursos para a grade, bem como o acesso a esses por outros grupos. Um grupo é definido geralmente como uma rede local ou aglomerado (*cluster*) de computadores, interligados por um meio rápido de comunicação e com repositórios de dados compartilhados, cada qual executando um gerenciador de banco de dados relacionais.

Todo grupo contém um gerenciador de grupo (*Group Manager – GM*), vários gerenciadores de nó (*Node Manager – NM*) e vários gerenciadores de cliente (*Client Manager – CM*) (Figura 5.1).

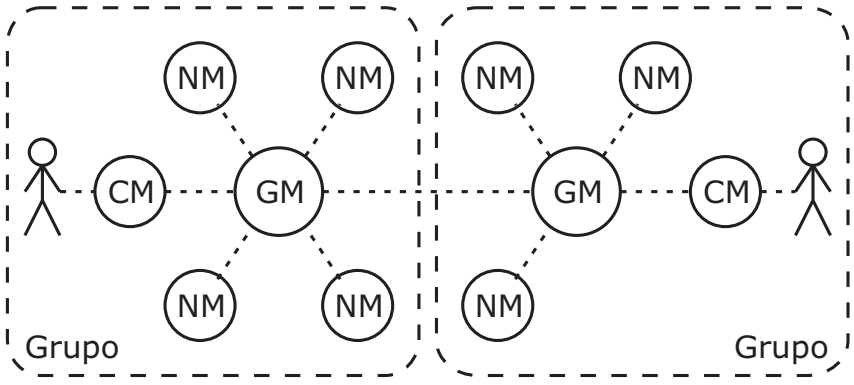

Figura 5.1: Arquitetura do *Middleware* Xavantes.

Gerenciador de grupo (GM) mantém informações atuais sobre os gerenciadores de nó do grupo, tais como localização, disponibilidade e alocação, e sobre os gerenciadores de cliente do grupo. Ele troca informações de disponibilidade com outros grupos, e repassa e recebe requisições de alocação de recursos para executar elementos de processo entre grupos. Ademais, controla o *status* de execução e a localização dos elementos executados no grupo e invocados pelo grupo; e gerencia domínios de aplicação, que contêm as especificações das aplicações do grupo.

Gerenciador de nó (NM) é composto de um gerenciador de processos (*Process Manager – PM*) e um gerenciador de atividades (*Activity Manager – AM*). Um gerenciador de nó pode ser acrescentado ou excluído do grupo a qualquer momento, em

tempo de execução. O gerenciador de processos permite a alocação de recursos para a execução de processos e controladores, e cria e executa esses elementos. Ele contém o escalonador de processos que requisita a alocação de recursos e distribui os elementos de um processo durante a sua execução. O gerenciador de atividades permite a alocação de recursos para a execução de atividades, e cria e executa essas atividades, aproveitando os processadores e demais recursos da máquina. Um gerenciador de nó pode desabilitar o seu gerenciador de processos ou de atividades, fazendo com que o servidor nó somente execute atividades ou processos, respectivamente.

Gerenciador de cliente (CM) é utilizado pelos usuários para acessar os recursos da grade computacional e controlar a execução de aplicações. Ele permite especificar, invocar e monitorar processos e atividades, ao interagir diretamente com os serviços da grade.

Todos os três serviços descritos são executados sobre máquinas virtuais Java [LY99] e utilizam a biblioteca de classes Java, que provê acesso aos recursos locais e comunicação entre nós distribuídos por meio do protocolo RMI (*Remote Method Invocation*) [Mic04]. Para a persistência de dados e tolerância a falhas, esses serviços utilizam bases de dados relacionais *MySQL* [MyS07] compartilhadas pelo grupo, por meio das operações disponibilizadas pela interface de programação JDBC (*Java Database Connectivity*) [And06]. O gerenciador de grupo utiliza um repositório local para persistir dados sobre usuários, os recursos do grupo, registro de grupos pares e localização dos elementos de processo. O gerenciador de nó utiliza um repositório local ou de grupo para persistir o estado das instâncias de processos e atividades. O gerenciador de cliente armazena os dados de invocação de elementos e os resultados em um repositório local. O gerenciador de grupo e os gerenciadores de nó também utilizam o sistema de arquivos local para armazenar a especificação das aplicações e bibliotecas utilizadas.

5.1.1 Gerenciador de Grupo

O gerenciador de grupo (GM) é responsável por manter informações sobre o grupo e alocar e controlar a execução de elementos de processo nos servidores nó. As informações mantidas sobre o grupo são: estado dos gerenciadores de nó registrados no grupo, com estado de ativação, localização e agendas de alocação de cada nó; estado, localização e agendas agregadas de alocação dos gerenciadores de grupo pares (peer GMs), que servem requisições para a execução de atividades e processos nesses grupos adjacentes; estado de ativação e localização dos clientes (*front-ends*) que servem os usuários; estado de execução e localização dos processos, controladores e atividades em execução no grupo ou invocados por clientes locais; e conteúdo dos domínios de aplicação, com arquivos XML

de especificação de processos e atividades, arquivos de bibliotecas compiladas contendo classes Java da aplicação, e arquivos específicos utilizados pela aplicação.

O gerenciador de grupo registra gerenciadores de nó (NM), monitora o seu estado de ativação e localização e contém cópia da agenda de alocação de processos de cada gerenciador de processos (PM) e da agenda de alocação de atividades de cada gerenciador de atividades (AM) associados aos gerenciadores de nó locais. O gerenciador de grupo também registra gerenciadores de grupo pares (peer GMs), mantém informações sobre os seus estados de ativação e localização e contém agenda agregada de disponibilidade de recursos dos servidores nó de cada grupo par, atualizada por meio de troca de informações entre os gerenciadores de grupo. Com base nessas informações, todas as requisições de busca dos melhores recursos desse grupo, ou de outro grupo, para se executar uma certa atividade ou processo, com determinadas características e requisitos, advindas do grupo local, são respondidas por ele. Ele é responsável por alocar servidores nó locais (do grupo) para executar atividades e processos de acordo com a seleção realizada e controlar quando e quais elementos devem ser executados nesses servidores em um determinado momento. Ademais, ele monitora o estado de ativação e localização dos gerenciadores de cliente dos usuários do grupo, para receber requisições desses e notificar alterações no estado de execução dos processos e atividades invocados por esses.

Para se recuperar em caso de falhas, o gerenciador de grupo utiliza um banco de dados relacional para persistir o registro, localização, estado de ativação e agendas de alocação dos serviços gerenciadores de nó; registro, localização e estado de ativação dos serviços gerenciadores de grupo dos grupos pares; registro, estado de ativação e localização dos gerenciadores de cliente; identidade de usuários; e estado de execução e localização de atividades e processos executados no grupo ou invocados por esse. Ele armazena as especificações originais das aplicações em XML, atividades compiladas, bibliotecas de classes utilizadas e outros arquivos específicos em diretórios relacionados às aplicações (domínios de aplicação) no sistema de arquivo local.

O serviço gerenciador de grupo é iniciado por meio da execução do comando *xavantes_gs* (*Xavantes Group Server*). Ao iniciar, o serviço carrega todas as informações sobre o grupo armazenadas no repositório do grupo e reinicia a sua interação com os demais gerenciadores de nó do grupo, gerenciadores de cliente e gerenciadores de grupo pares.

A Figura 5.2 apresenta a estrutura interna do gerenciador de grupo. As funcionalidades de cada componente são descritas nas próximas seções.

Ambiente de Execução (*Runtime Environment*)

O gerenciador de grupo é executado sobre a plataforma *Java Standard Edition 6.0 (Java SE 6.0)* [Mic06], assim como os demais serviços Xavantes (gerenciador de nó e de cliente), que pode ser instalada em inúmeros sistema operacionais existentes nos computadores de

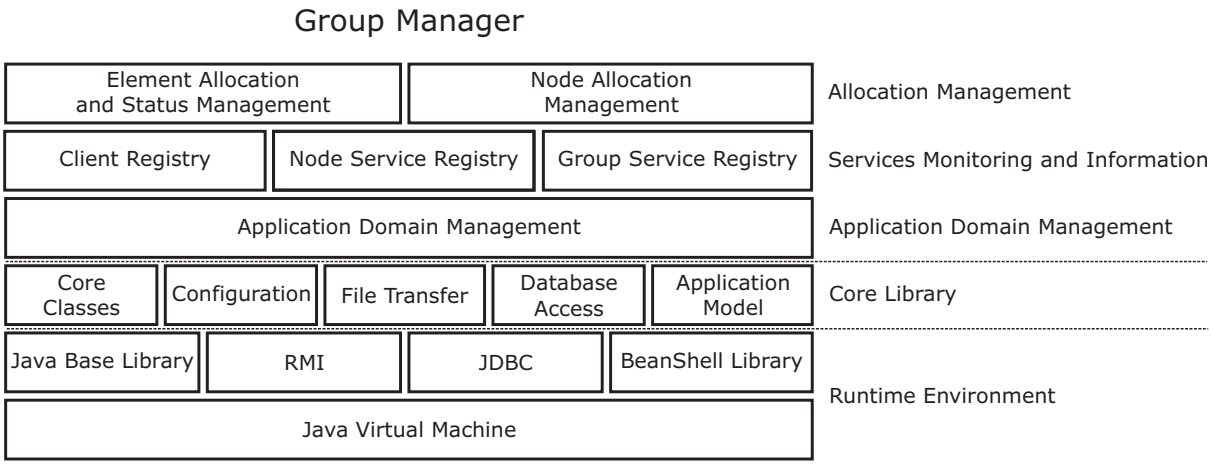

Figura 5.2: Estrutura do Gerenciador de Grupo (GM).

ambientes domésticos, comerciais, universitários e governamentais, incluindo Windows, Mac OS, Linux e Solaris, residentes em mais de 99,5% dos computadores do mundo [App07].

Cada serviço Xavantes é compilado como Java *bytecodes* e executado sobre uma máquina virtual Java (*Java Virtual Machine — JVM*) [LY99]. Eles utilizam as funcionalidades presentes na biblioteca básica do Java (*Java Base Library*) da plataforma Java SE 6 para o acesso aos recursos da máquina, disparo e controle de *threads*, comunicação remota, acesso a bases de dados remotas e sistemas de arquivo, entre outras tarefas básicas. Detalhadamente, a comunicação remota é realizada por meio do protocolo de invocação remota de objetos (*Remote Method Invocation — RMI*) [Mic04] e o acesso às bases de dados relacionais ocorre por meio da interface de programação Java de conexão a bases de dados (*Java Database Connectivity — JDBC*) [And06].

As bases de dados relacionais são controladas pelo gerenciador de banco de dados *MySQL 5.0* [MyS07], que deve ser instalado em máquinas locais ou em mais máquinas do grupo, servindo de repositórios locais ou compartilhados, respectivamente. Um desses repositórios deve ser utilizado pelo gerenciador de grupo, para persistência de seus dados. Todos os repositórios compartilhados, inclusive o do gerenciador de grupo, podem ser distribuídos para serem utilizados pelos gerenciadores de nó e de cliente. Cada um desses serviços deve acessar um repositório local ou compartilhado para persistir seus dados de controle. A vantagem de acessar uma base de dados local, pelo gerenciador de nó ou de cliente, é o acréscimo da velocidade de acesso e da escalabilidade do sistema.

O ambiente de execução inclui a biblioteca de terceiros *BeanShell* (*BeanShell Library*), que contém classes para interpretar *scripts* Java. A biblioteca permite criar dinamica-

mente classes e funções Java internas ao interpretador, instanciar objetos a partir dessas classes, e obter e alterar os valores das variáveis no contexto do interpretador destes *scripts*. Essa biblioteca é utilizada para executar os *scripts* Java que podem ser inseridos nos elementos de processo.

Biblioteca Básica Xavantes (*Core Library*)

Os serviços Xavantes utilizam um conjunto de classes da biblioteca básica, conhecida como *Xavantes Core Library*, para acessar estruturas de dados comuns, realizar operações básicas, e acessar os serviços remotos. Entre as estruturas de dados comuns estão classes que especificam o modelo de uma aplicação, tais como processos, controladores e atividades, e detalhes de cada um desses elementos. As classes comuns também definem exceções, eventos, informações sobre serviços e elementos. Entre as operações básicas realizadas por essa biblioteca estão a leitura do arquivo de configuração, criação de conexão à base de dados, criação e exportação de referência a um serviço e transferência de arquivos. Todas as interfaces remotas aos serviços estão disponíveis nessa biblioteca.

Essa biblioteca pode ser dividida em funções, conforme a classificação abaixo:

Classes Base (*Core Classes*) — classes que contêm operações básicas para os serviços Xavantes, tais como geração de identificadores globais únicos (*UIDs*), exportação e obtenção de referências de objeto remoto de interface *facade* de serviço e suporte a objetos compartilhados em controladores. Há também classes de informações sobre serviços e elementos de processo, exceções e eventos comuns. As interfaces de acesso remoto aos serviços Xavantes por *RMI* são definidas nessa biblioteca.

Configuração (*Configuration*) — operações de acesso aos dados do arquivo de configuração, tais como endereço da máquina e porta do repositório persistente de dados, endereço da máquina e porta do gerenciador de grupo, porta de exportação do serviço e nome do serviço.

Transferência de Arquivos (*File Transfer Manager*) — operações de listagem de arquivos em um diretório domínio de aplicação, de obtenção de sua data de atualização e de transferência de arquivos atualizados.

Acesso a Base de Dados (*Database Access*) — operações de obtenção de conexões de um *pool* para uma certa base de dados gerenciada pelo *MySQL* e de leitura e escrita de objetos do tipo valor em campos da base.

Modelo de Aplicação (*Application Model*) — classes que definem a estrutura dos processos, controladores e atividades da aplicação. Por exemplo, por meio de uma

instância da classe *ProcessType*, que corresponde a um processo especificado, é possível acessar a estrutura desse processo, obter os nomes de seus parâmetros de entrada e de saída, e os elementos de seu corpo. Da mesma forma, por meio de uma instância da classe *Code*, que corresponde a um código especificado em um processo, é possível obter o *script* Java desse elemento código. Essas estruturas são utilizadas para especificar processos e controladores, bem como visualizar e interpretar instâncias desse processos e controladores, quando usadas para representar instâncias.

Gerenciamento de Domínios de Aplicação (*Application Domain Management*)

As aplicações Xavantes são compostas de arquivos XML que especificam processos e atividades, arquivos de bibliotecas de classes Java que definem estrutura de dados e funções (ou regras) de negócio, arquivos de bibliotecas de classes de terceiros, e arquivos de dados. As atividades também podem ser definidas por classes Java que derivam de *Activity* do *framework* Xavantes, depois compiladas e inseridas nas bibliotecas da aplicação.

Implantação de Aplicações Cada aplicação definida tem um diretório correspondente no servidor do grupo, chamado de domínio da aplicação. Para implantar uma aplicação, os arquivos que definem a aplicação e os arquivos de dados devem ser transferidos (implantados) para esse diretório. Os arquivos componentes da aplicação são os arquivos XML que definem os processos e atividades, as bibliotecas de classes da aplicação, bibliotecas de terceiros e arquivos de dados.

Compilação Automática de Arquivos XML de Atividade Todos os arquivos XML de especificação de atividades são automaticamente compilados quando atualizados no diretório do domínio da aplicação. A compilação gera um fonte Java com uma classe derivada de *Activity* do *framework* Xavantes com o código correspondente da especificação, sendo depois compilada para uma classe em *bytecodes* Java e inserida na biblioteca de classes da aplicação. O objetivo é transformar as atividades em um código mais eficiente de ser executado, uma vez que elas representam a parcela mais intensiva em computação da aplicação.

Distribuição de Arquivos da Aplicação Após implantados os arquivos da aplicação, qualquer nó que desejar executar uma instância de um processo ou de uma atividade da aplicação, acessa o gerenciador de grupo e obtém os arquivos correspondentes. Os arquivos XML que define o processo, então, no ambiente do nó, podem ser carregados e analisados para gerar a estrutura do processo a ser interpretado. Já os arquivos com bibliotecas de classes, que incluem atividades compiladas, classes com regras de negócio e código de

terceiros, são utilizados para se obter localmente classes previamente compiladas em Java *bytecodes* a serem instanciadas e utilizadas no contexto da aplicação.

Registro de Gerenciadores de Cliente (*Client Registry*)

Um gerenciador de cliente é o *front end* de um usuário de um grupo para o uso da grade. O usuário, para acessar os serviços da grade, por meio de um grupo, deve utilizar o gerenciador de cliente associado. O usuário pode estar cadastrado em vários grupos, mas somente tem um gerenciador de cliente por grupo. O serviço de registro de gerenciadores de cliente tem como objetivo registrar os gerenciadores de cliente (CM) de um determinado grupo, com dados do usuário associado, e monitorar o seu estado de ativação e localização.

Registro de Gerenciadores de Cliente Antes de ser utilizado, o gerenciador de cliente deve ser cadastrado no gerenciador de grupo. As informações mantidas pelo gerenciador de grupo em relação aos gerenciadores de cliente são: identificador único do gerenciador de cliente (*UID*), nome, login e senha do usuário associado, *status* do gerenciador de cliente, localização (máquina e porta onde uma referência à interface *callback* desse serviço pode ser obtida) e sua referência atual.

Monitoramento de Gerenciadores de Cliente Na inicialização do gerenciador de cliente, ele se autentica ao gerenciador de grupo utilizando usuário e senha. Periodicamente o gerenciador de cliente envia um evento para o gerenciador de grupo informando que ele continua ativo, sua localização e referência. Se após um período maior que um determinado limite ele não enviar o evento, o gerenciador de grupo altera o seu *status* para desconectado e descarta a sua referência.

Registro de Gerenciadores de Nó (*Node Service Registry*)

Os gerenciadores de nó têm por objetivo executar processos e atividades na grade computacional. O serviço de registro de gerenciadores de nó do gerenciador de grupo registra os gerenciadores de nó existentes em um grupo e monitora estado de ativação desses gerenciadores de nó.

Registro de Gerenciadores de Nó Um gerenciador de nó, para se tornar disponível e utilizável pela grade, para executar processos e atividades, deve se registrar no gerenciador de grupo correspondente. Para manter esses registros, cada gerenciador de grupo contém um cadastro de gerenciadores de nó que compõem o grupo, com um identificador único (*UID*), nome, senha de autenticação, estado de ativação, localização e informações adicionais para cada gerenciador de nó, tais como descrição da arquitetura, número de

processadores, poder de processamento agregado (soma do poder de processamento de todos os processadores), quantidade de memória volátil e de disco do nó gerenciado, bibliotecas disponíveis e campos que indicam se o gerenciador de nó pode executar processos, atividades, ou ambos.

Monitoramento de Gerenciadores de Nó O gerenciador de grupo monitora o estado de ativação, localização e os recursos disponíveis nos gerenciadores de nó do grupo. Um gerenciador de nó ao iniciar se autentica utilizando o seu nome e senha, indicando ao gerenciador de grupo que está ativo. Desde a sua inicialização, periodicamente o gerenciador de nó envia um evento ao gerenciador de grupo com informações, tais como referência remota, localização e recursos disponíveis, e.g. poder de processamento e quantidade de memória e de disco. Se um gerenciador de nó cessa o envio de eventos por um período maior, seu estado de ativação é alterado para desconectado e sua referência remota é descartada. Caso continue não enviando eventos, seu estado de ativação é alterado para desativado. No estado desativado, todos os processos, controladores e atividades alocados para esse gerenciador de nó são realocados pelo gerenciador de grupo.

Registro de Gerenciadores de Grupo (*Group Service Registry*)

Um gerenciador de grupo pode se relacionar com grupos pares para requisitar ou fornecer recursos computacionais a esses grupos para executar processos e atividades, formando, assim, uma grade computacional plana *peer-to-peer*. Um grupo que requisita recursos computacionais de outro é denominado grupo cliente em relação a esse; enquanto que um grupo que fornece recursos a outro é denominado grupo servidor.

Registro de Gerenciadores de Grupos Pares O relacionamento entre o grupo local e um grupo par é construído inserindo-se os dados desse grupo par no cadastro de grupos pares de um certo grupo. Um relacionamento com um grupo par contém o identificador global, nome, estado de ativação, localização do servidor do grupo par correspondente, a referência a esse, informações de número de nós, poder computacional agregado, velocidade de comunicação da rede interna e da rede externa, entre o grupo par e o local, e o tipo de relacionamento do grupo par com o grupo local (cliente, servidor ou cliente e servidor).

Monitoramento de Gerenciadores de Grupos Pares Uma vez inicializado o gerenciador de grupo, ele procura todos os grupos pares servidores e se registra nos gerenciadores de grupo correspondentes para receber periodicamente informações de número de nós e poder computacional disponíveis nesses grupos. Além dessas informações, ele recebe, desses grupos servidores, e dos demais grupos com que esses estão ligados diretamente,

agendas agregadas com poder computacional médio disponível por tempo de cada grupo, como veremos no serviço "Gerenciamento de Alocação de Nós". Se um gerenciador de grupo cessa o envio de eventos por um período maior que um determinado limite, seu estado de ativação é alterado para desconectado e sua referência remota é descartada. Caso continue não enviando eventos após um segundo limite de tempo, seu estado de ativação é alterado para desativado. No estado desativado, todos os processos, controladores e atividades alocados a partir do grupo local para esse gerenciador de grupo são realocados pelo gerenciador de grupo local.

Gerenciamento de Alocação de Nós (*Node Allocation Management*)

O gerenciador de grupo é responsável por selecionar heuristicamente o melhor nó da grade para executar um elemento de processo (atividade, processo ou controlador) com determinados requisitos e estimativas de execução, invocados a partir do grupo local. Para realizar essa tarefa, o gerenciador de grupo mantém, para cada nó do grupo, uma agenda de alocação de períodos para execução de atividades, denominada agenda de atividades, e uma agenda de alocação de períodos para execução de processos ou controladores, denominada agenda de processos. Ele também gerencia, para cada grupo servidor conectado, uma agenda agregada de disponibilidade média de poder de processamento e do total de memória dos nós daquele grupo servidor, divida por tempo e por faixas de prioridade dos elementos.

Manutenção de Agendas de Alocação de Atividades para Cada Nó Para selecionar o melhor servidor nó do grupo para executar uma atividade, o gerenciador de grupo gerencia agendas detalhadas de alocação de atividades para cada nó do grupo. Cada agenda contém tempos de início e de fim de execução previstos para cada atividade alocada, identificador da atividade, sua prioridade, bem como poder de processamento e quantidade de memória volátil reservados para a sua execução. Atividades podem ser executadas concorrentemente em um determinado nó. A prioridade é formada por um vetor de dois inteiros: o primeiro indica a prioridade absoluta, da aplicação; e o segundo indica a prioridade relativa, entre os elementos da mesma aplicação.

Manutenção de Agendas de Alocação de Processos para Cada Nó Para selecionar o melhor servidor nó do grupo para executar um processo ou controlador, o gerenciador do grupo mantém agendas detalhadas de alocação de processos e controladores para cada nó do grupo. Cada agenda contém tempos de início e de fim de execução previstos para cada processo ou controlador alocado, seu identificador e prioridade. Processos e controladores podem ser executados concorrentemente em um determinado nó. A prioridade é formada por um vetor de dois inteiros: o primeiro indica a prioridade

absoluta, da aplicação; e o segundo indica a prioridade relativa, entre os elementos da mesma aplicação.

Seleção de Gerenciador de Nó Local para Executar uma Atividade A seleção de um nó para executar uma atividade considera o tempo mínimo de início, tempo máximo de fim, esforço total de processamento, poder de processamento e memória mínimos requeridos e bibliotecas necessárias pela atividade. Pode-se indicar a prioridade da atividade, sendo esta alocada com preferência sobre outras atividades com menor prioridade. A comparação entre prioridades verifica se as atividades pertencem à mesma aplicação: se não pertencer, as prioridades absolutas são comparadas; se pertencer, as prioridades relativas são comparadas.

A seleção itera sobre todas agendas de atividades dos nós do grupo verificando se o nó contém as bibliotecas necessárias, e simulando se a atividade pode ser alocada naquela agenda, considerando se o poder de processamento disponível do período entre o tempo mínimo de início e o tempo máximo de fim é suficiente para executar a atividade que exige o esforço total de processamento fornecido, com a prioridade dada, sempre acima do poder de processamento mínimo e memória mínima requerida. Deve ser notado que o poder máximo de processamento que pode ser alocado a uma atividade é o de um único processador, mesmo se o nó possuir mais de um, uma vez que cada atividade é executada por um único caminho de execução (*thread*). O nó selecionado é o que conseguir alocar a atividade no período especificado e que resultar no menor tempo de término, considerando atividades já alocadas de prioridade igual ou superior. Dessa forma, aproveita-se o nó que possui o maior poder de processamento disponível no grupo e que satisfaz os requisitos da atividade, promovendo um balanceamento de carga e uma execução mais eficiente possível no grupo.

Pode-se também fornecer o tamanho total das mensagens trocadas entre a atividade e o elemento pai da atividade e o identificador do nó em que esse elemento pai está alocado. Essas informações são utilizadas para se calcular o tempo de comunicação da atividade e acrescentá-lo ao tempo total de processamento desta no momento da simulação da alocação da atividade nas agendas. Para o cálculo do tempo de comunicação: se o elemento invocador estiver localizado em outro grupo, é considerada a velocidade da rede externa entre os dois grupos; se o elemento estiver no mesmo grupo, é considerada a velocidade da rede interna; e se estiver no mesmo nó, o tempo de comunicação é considerado nulo.

Seleção de Gerenciador de Nó Local para Executar um Processo ou Controlador A seleção de um nó para executar um processo ou controlador considera o tempo mínimo de início e o tempo máximo de fim para a execução desse elemento. Pode-se indicar a prioridade do processo ou controlador, sendo este alocado com preferência sobre

os outros elementos com menor prioridade. A comparação entre prioridades verifica se os elementos pertencem à mesma aplicação: se não pertencer, as prioridades absolutas são comparadas; se pertencer, as prioridades relativas são comparadas.

A seleção itera sobre todas as agendas de processos dos nós do grupo simulando se o elemento pode ser alocado naquela agenda, considerando se o número limite máximo de elementos, entre o tempo mínimo de início e o tempo máximo de término, com a prioridade dada, não foi atingido. O nó selecionado é o que alocar o processo no período especificado e que resultar no menor tempo de término, considerando elementos já alocados de prioridade igual ou superior. Dessa forma, aproveita-se o nó que possui o maior poder de processamento disponível no grupo e que satisfaz os requisitos do processo ou controlador, promovendo um balanceamento de carga.

Manutenção de Agendas Agregadas de Disponibilidade de Recursos em Grupos Pares Para selecionar o melhor grupo para executar uma atividade, um processo ou um controlador, o gerenciador de grupo local mantém agendas agregadas de disponibilidade média, em poder de processamento e de memória volátil médios nas máquinas, por faixas de tempo e prioridade de atividades, em todos os grupos servidores relacionados. Cada agenda agregada contém informações de um grupo. Ela é dividida em períodos fixos e faixas fixas de prioridade contendo qual é o poder de processamento e memória volátil médios disponíveis nos nós do grupo para aquele período e faixa de prioridades de atividades específicos. Cada agenda contém o número de nós do grupo, para ser possível calcular o poder total de processamento e de memória total do grupo. Essa agenda é atualizada constantemente por meio de troca de informações entre o grupo servidor e o grupo cliente. Ela é utilizada para verificar quais são os melhores grupos para executar uma atividade com determinados requisitos e prioridade no período de tempo indicado.

Uma agenda similar é mantida para apresentar quantos processos e controladores ainda podem ser alocados em média nos nós do grupo por faixas de tempo e por faixas de prioridade absoluta. Ela é utilizada para verificar quais são os melhores grupos para executar um processo ou controlador com determinada prioridade no período de tempo indicado.

Seleção de Gerenciador de Nó da Grade para Executar uma Atividade A seleção de um nó da grade para executar uma atividade ocorre em três fases. Na primeira fase é escolhido um certo número de grupos servidores, incluindo o próprio grupo local, que pode executar a atividade com o menor tempo de término, considerando os requisitos de tempo mínimo de início, tempo máximo de fim, esforço total de processamento, poder de processamento e memória mínimos, bibliotecas necessárias e prioridade da atividade. Para essa seleção inicial de grupos, são utilizadas as informações das agendas de poder

de processamento e memória médios disponíveis para atividades de uma determinada prioridade nos servidores nó do grupo relacionado. Na segunda fase, é requisitado aos grupos selecionados o melhor servidor nó para executar a atividade, que permite executar a atividade com as restrições definidas no menor tempo de término previsto, utilizando-se o serviço "Seleção de Gerenciador de Nó Local para Executar uma Atividade". Na terceira fase, desses nós obtidos, um de cada grupo escolhido, é selecionado o nó que retornar o menor tempo de término previsto para a atividade. Dessa forma, consegue-se um balanceamento de carga entre os nós e um menor tempo de execução, considerando todos os nós visíveis da grade.

É possível fornecer o tamanho total das mensagens trocadas entre a atividade e o seu elemento invocador, bem como o identificador do nó onde o elemento invocador foi alocado. Essas informações são úteis para calcular o tempo previsto de comunicação entre a atividade e o seu elemento invocador. Esse tempo é adicionado ao total previsto para o processamento da atividade, sem comunicação, para se obter o tempo total de execução da atividade. O tempo total de execução então é utilizado para calcular qual é o melhor nó da grade para executar a atividade, utilizando o método descrito acima. O tempo de comunicação é calculado baseando-se na velocidade da comunicação entre o grupo local e o grupo servidor, se o elemento superior estiver em um grupo distinto do grupo da atividade; ou na velocidade da rede interna do grupo, se o elemento estiver no mesmo grupo da atividade. Na segunda fase, a seleção do nó é realizada utilizando a informação do custo de comunicação passado, segundo o descrito no serviço "Seleção de Gerenciador de Nó Local para Executar uma Atividade".

Seleção de Gerenciador de Nó da Grade para Executar um Processo ou Controlador A seleção de um nó da grade para executar um processo ou controlador ocorre em três fases. Na primeira fase é escolhido um certo número de grupos servidores, incluindo o próprio grupo local, que pode executar o processo ou o controlador com o menor tempo de término, considerando os requisitos de tempo mínimo de início, tempo máximo de fim, esforço total de processamento e prioridade absoluta do elemento. O esforço total de processamento de um processo ou controlador é definido pela somatória do esforço estimado de processamento necessário para executar todos os seus elementos filhos, recursivamente. Para essa seleção inicial de grupos, são utilizadas as informações das agendas de poder de processamento total disponível para atividades de uma determinada prioridade absoluta nos servidores nó do grupo relacionado. Na segunda fase, é requisitado aos grupos selecionados o servidor nó que retornar o menor tempo de término para executar um processo ou controlador utilizando-se o serviço "Seleção de Gerenciador de Nó Local para Executar um Processo". Na terceira fase, desses nós obtidos, um de cada grupo, é selecionado o nó que retornar o menor tempo de término para o processo ou controlador.

Dessa forma, aproveita-se o grupo que tem maior capacidade de executar os elementos filhos, recursivamente, sem que esses precisem ser alocados em outros grupos, reduzindo o tempo de comunicação total interno ao elemento, promovendo um balanceamento de carga entre todos os grupos pares e reduzindo o tempo total de execução do elemento.

Gerenciamento de Alocação e Estado de Elementos (*Element Allocation and Status Management*)

O gerenciador de grupo é responsável por alocar, e realocar se necessário, um gerenciador de nó da grade para executar um elemento de processo (atividade, processo ou controlador); informar aos clientes sobre o estado de execução dos elementos; e controlar a execução dos elementos nos gerenciadores de nó, informando quais elementos devem ser executados, abortados, suspensos e reexecutados. Para isso ele mantém um tabela de elementos de processo invocados, além de conter as agendas de atividades e de processos de cada nó, como foi visto no serviço anterior. Os elementos podem ser invocados pelos gerenciadores de cliente e de processo do grupo, além de gerenciadores de grupos clientes.

Consulta sobre o Estado de Execução e Alocação de um Elemento O gerenciador de grupo controla uma tabela de elementos que contém informações sobre cada elemento invocado pelos gerenciadores de cliente e de processo do grupo, sendo executados no grupo local ou em outro grupo, incluindo seu estado de execução e sua alocação atual. Com base nessa tabela, o gerenciador responde o estado de um certo elemento para o cliente que requisitou a sua alocação. Em detalhes, cada registro da tabela contém a identificação do elemento invocado, com o tipo do elemento, sua prioridade, seu estado de execução, a identificação do elemento invocador, a identificação do cliente invocador (gerenciador de cliente ou de processo), seus requisitos de execução, tais como tempo mínimo de início, tempo máximo de fim, esforço total de processamento, tamanho total das mensagens de comunicação com o elemento invocador, poder de processamento mínimo e tamanho de memória mínima requeridos, bibliotecas necessárias, e dados de alocação, se estiver alocado, tais como o gerenciador de nó alocado ao elemento e o tempo de início e de fim de execução alocados.

Um elemento pode passar por vários estados de execução durante o seu ciclo de vida:

não alocado — não foi selecionado e alocado um nó para a execução do elemento;

não pronto — o elemento foi alocado, mas sua especificação e bibliotecas não foram carregados na máquina alocada ou os parâmetros iniciais não foram passados pelo invocador;

pronto — o elemento foi alocado e todos os pré-requisitos para a sua execução foram satisfeitos, estando em estado de espera pelo tempo de execução;

em execução — o elemento se encontra em execução no período alocado a ele no servidor nó;

terminado — sua execução foi concluída com sucesso e ele enviou os resultados para o invocador;

abortado — a execução do elemento foi abortada por ordem, por falha ou porque ele ultrapassou o tempo de término estabelecido;

suspenso — o elemento é recuperável e sua execução foi suspensa por ordem, por falha ou porque ele ultrapassou o tempo de término estabelecido.

Alocação de Gerenciador de Nó do Grupo para Executar uma Atividade O gerenciador de grupo é responsável por alocar os nós do grupo para executar atividades. O gerenciador de grupo aloca a atividade no servidor nó previamente selecionado ou no requisitado diretamente pelo escalonador de processos. Ele realiza essa tarefa marcando na sua agenda de atividades para este nó que a atividade pode executar desde o tempo de início até o tempo de fim que couber em um *slot* disponível desta agenda, respeitando o período definido entre o tempo mínimo de início e o tempo máximo de fim da atividade, utilizando um poder de processamento médio correspondente ao esforço total de execução da atividade dividido pelo período agendado de sua execução, nunca inferior ao poder de processamento mínimo requisitado e com memória não inferior à quantidade mínima requisitada. O poder de processamento alocado a uma atividade nunca pode ser superior ao de um processador individual do servidor nó. Se não for possível alocar o poder de processamento ou memória para a atividade no período determinado, uma ou mais atividades de menor prioridade devem ser realocadas para liberar os recursos necessários. Após a alocação da atividade na agenda, o registro correspondente é atualizado na tabela de elementos do gerenciador de grupo e o gerenciador de nó correspondente é invocado para adicionar a atividade em sua lista (*pool*) de atividades a espera de execução. Ambos colocam a atividade no estado "não pronto".

Alocação de Gerenciador de Nó do Grupo para Executar um Processo O gerenciador de grupo é responsável por alocar os nós do grupo para executar processos e controladores. O gerenciador de grupo aloca o processo ou controlador no servidor nó previamente selecionado ou no requisitado diretamente pelo escalonador de processos. Ele realiza essa tarefa marcando na agenda de processos deste servidor nó que o elemento pode executar desde o tempo de início que couber em um *slot* disponível desta agenda,

respeitando-se o período definido entre o tempo mínimo de início e o tempo máximo de fim do elemento. Se não for possível alocar o período determinado, por excesso de elementos alocados ao período, um ou mais elementos de menor prioridade devem ser realocados para liberar os recursos necessários. Após a alocação do elemento na agenda, o registro correspondente é atualizado na tabela de elementos do gerenciador de grupo e o gerenciador nó correspondente é invocado para adicionar o processo ou controlador em sua lista (*pool*) de elementos a espera de execução. Ambos colocam o elemento no estado "não pronto".

Alocação de Gerenciador de Nó da Grade para Executar uma Atividade O gerenciador de grupo é responsável por alocar servidores nó da grade, do grupo local ou de outros grupos servidores, obedecendo a seleção previamente realizada ou por ordem direta do escalonador de processos, para executar atividades. Se o servidor nó se encontra no grupo local, é utilizado o serviço "Alocação de Gerenciador de Nó do Grupo para Executar uma Atividade" do gerenciador de grupo local para se alocar a atividade no servidor nó, atualizando-se a agenda do servidor nó e a tabela de elementos do gerenciador de grupo local. Se o nó se encontra em outro grupo servidor, o gerenciador de grupo local solicita ao gerenciador de grupo remoto que contém o nó para que ele aloque a atividade naquele servidor nó, por meio da invocação do serviço "Alocação de Gerenciador de Nó do Grupo para Executar uma Atividade" do último gerenciador de grupo. Após a alocação da agenda de atividades do servidor nó para executar a atividade e da inserção do elemento na tabela de elementos do gerenciador do grupo remoto, o gerenciador de grupo local insere o registro de alocação correspondente da atividade na tabela de elementos do gerenciador de grupo local, indicando que a alocação foi realizada em um grupo remoto. A inserção na tabela local serve para manter uma referência ao elemento no gerenciador de grupo invocador, o que é importante para rastrear o elemento caso o servidor nó alocado mude de grupo, por uma realocação ou um reescalonamento do elemento alocado.

Alocação de Gerenciador de Nó da Grade para Executar um Processo O gerenciador de grupo é responsável por alocar servidores nó da grade, do grupo local ou de outros grupos servidores, obedecendo a seleção previamente realizada ou por ordem direta do escalonador de processos, para executar processos e controladores. Se o servidor nó se encontra no grupo local, é utilizado o serviço "Alocação de Gerenciador de Nó do Grupo para Executar um Processo" do gerenciador de grupo local para se alocar o processo ou controlador no servidor nó, atualizando-se a agenda do servidor nó e a tabela de elementos do gerenciador de grupo local. Se o servidor nó se encontra em outro grupo, o gerenciador de grupo local solicita ao gerenciador de grupo remoto que contém o servidor nó para ele alocar o processo ou controlador naquele servidor nó, por meio da invocação

do serviço "Alocação de Gerenciador de Nó do Grupo para Executar um Processo" do último gerenciador de grupo. Após a alocação da agenda de processos do servidor nó para executar o elemento e a inserção desse elemento na tabela de elementos do gerenciador de grupo remoto, o gerenciador de grupo local insere o registro de alocação do processo ou controlador na tabela de elementos do gerenciador de grupo local, indicando que a alocação foi realizada em um grupo remoto. A inserção na tabela local serve para manter uma referência ao elemento no gerenciador de grupo invocador, o que é importante para rastrear o elemento caso o servidor nó alocado mude de grupo, por uma realocação ou um reescalonamento do elemento alocado.

Solicitação de Execução de Elementos ao Gerenciador de Nó O gerenciador de grupo monitora constantemente o tempo de início dos próximos elementos da agenda de cada nó do seu grupo para verificar quais execuções de elementos devem ser disparadas em um certo momento. Ele também recebe eventos dos gerenciadores de nó que indicam quando um elemento está pronto para a execução. Se o tempo de início de um determinado elemento for atingido e ele estiver no estado "pronto", o gerenciador de grupo solicita ao gerenciador de nó correspondente para iniciar a execução desse elemento. Se o elemento estiver no estado "não pronto", ele é realocado, liberando recursos para que outros elementos prontos possam ser executados antes dele de forma que sempre se utilize o máximo possível da capacidade computacional disponível. Um determinado servidor nó pode executar várias atividades concorrentemente, mesmo tendo somente um processador, se os seus períodos de alocação tiverem pontos de intersecção. Processos e controladores também podem ser executados concorrentemente na mesma condição.

Solicitação de Aborto ou Suspensão de Elementos ao Gerenciador de Nó O gerenciador de grupo monitora constantemente o tempo de término dos próximos elementos da agenda de cada nó do seu grupo para verificar quais execuções devem ser abortadas ou suspensas em um certo momento. Se o tempo de término de um determinado elemento for passado por um determinado fator e ele estiver no estado "em execução", o gerenciador de grupo solicita ao gerenciador de nó correspondente para abortá-lo ou suspendê-lo. O elemento é abortado se for uma atividade não recuperável; e suspenso se for uma atividade recuperável, processo ou controlador. O elemento abortado ou suspenso é realocado para execução, neste nó ou em outro nó do grupo ou da grade. Se for recuperável, ele é preferencialmente realocado para um nó do grupo com acesso ao mesmo repositório compartilhado para possibilitar a recuperação do seu estado e execução a partir do ponto mais próximo ao que ele foi suspenso.

Tratamento de Término ou Aborto na Execução de Elementos O gerenciador de grupo recebe eventos dos gerenciadores de nó que notificam que um elemento foi terminado ou abortado por falha. Em ambos casos, ele remove a alocação do elemento da agenda de atividades ou de processos correspondente ao nó, modifica o estado do elemento na tabela de elementos e realoca os próximos elementos, que não estiverem em execução, por ordem de prioridade e alocação, na agenda do nó para aproveitar os espaços (*slots*) liberados. Com a agenda alterada, o gerenciador de grupo pode verificar quais são os próximos elementos que podem ser executados no servidor nó aproveitando os recursos liberados e solicita a execução desses elementos utilizando o serviço "Solicitação de Execução de Elementos ao Gerenciador de Nó".

Realocação de Elementos O gerenciador de grupo é responsável por realocar elementos de processo (atividades, processos ou controladores). Um elemento deve ser realocado por diversos motivos: necessidade de liberação de recursos para a execução de elementos com maior prioridade; elemento não pronto para execução no tempo de início alocado; período para término de execução do elemento excedido; término antecipado de elementos anteriores, como liberação de recursos; servidor nó ou grupo, para o qual o elemento estava alocado, desativado; e redistribuição de carga por ativação de novos servidores nó ou grupos.

No início da realocação, se o elemento estiver com o estado "em execução", o gerenciador de nó é invocado, se estiver ativo, para suspender ou abortar a execução do elemento, dependendo se ele é recuperável ou não, respectivamente. O elemento somente é realocado com o estado "em execução" nos casos de tempo de término excedido ou servidor nó ou grupo desativado. O elemento a ser realocado é então removido da agenda do gerenciador de nó correspondente e o estado desse elemento é alterado para "não alocado". Se a solicitação de alocação inicial adveio de um gerenciador de grupo cliente, o responsável pela realocação é esse último gerenciador, não o gerenciador de grupo do servidor nó alocado. Nesse caso o primeiro é notificado da alteração do estado do seu elemento para "não alocado", para realocá-lo. Se a solicitação da alocação inicial adveio do próprio grupo, o elemento é reinserido na fila (*pool*) de elementos para alocação desse grupo. Os elementos dessa fila são alocados na ordem da prioridade e do tempo da solicitação original de alocação. De qualquer forma, se não houver recursos para alocar o elemento com as restrições originais dadas, o cliente, solicitador da alocação, é notificado.

Se o elemento realocado for um processo, controlador ou atividade recuperável e esse tiver sido suspenso, na realocação, como critério adicional, é procurado um servidor nó que acesse o mesmo repositório compartilhado em que foi persistido o estado de execução do elemento suspenso pelo servidor nó anterior. Se não for encontrado, a execução do elemento é abortada e este pode ser realocado em qualquer nó de qualquer grupo, como

os demais elementos.

Realocação de Elementos por Desativação de Gerenciadores de Nó ou de Grupo Servidores de nó podem ter o seu estado passado para desativado no gerenciador de grupo por ordem do usuário ou por falta de comunicação com o gerenciador de grupo por um período determinado. Nesse caso, o gerenciador de grupo remove todas as atividades e processos, ou controladores, alocados a esse gerenciador de nó das respectivas agendas de atividades e de processos do nó e insere esses elementos na fila (*pool*) de elementos a serem alocados. Ao mesmo tempo, os elementos constantes nessas agendas têm o seu estado alterado para "não alocado". No caso de elementos invocados por grupos pares, esses grupos são informados sobre a alteração de *status* de seus elementos e os respectivos gerenciadores tratam de sua realocação. Dessa forma, o gerenciador de grupo automaticamente trata de remoções de nós do grupo, promovendo uma realocação automática de elementos.

Um procedimento semelhante é realizado quando um determinado grupo é desativado. Todos os elementos alocados a esse grupo servidor são marcados como "não alocado" e inseridos na fila de alocação. Como um gerenciador de grupo somente pode solicitar a alocação dos elementos invocados no seu grupo local (não invocados por grupos clientes) em grupos servidores, nesse caso não existe gerenciadores de grupos clientes a serem comunicados sobre a realocação dos seus elementos.

Realocação de Elementos por Ativação de Gerenciadores de Nó ou de Grupo De tempos em tempos, o gerenciador de grupo tenta selecionar servidores de nó para a realocação dos elementos que não estão em execução, de forma a aproveitar ao máximo os recursos disponíveis no grupo e na grade. Se o tempo de término previsto para o elemento no novo recurso for menor, por um determinada diferença, o gerenciador de grupo tenta realocar o elemento nesse servidor de nó. A diferença limite para haver a realocação pode ser determinada pelo estado do elemento: se ele estiver em "pronto" esse valor pode ser maior do que se o elemento estiver em "não pronto". Esse mecanismo permite aproveitar servidores de nó e grupo recentemente ativados para executar elementos que já estavam alocados, além de recursos que ficaram disponíveis por erros acumulados de estimativas. Dessa forma, o gerenciador de grupo balanceia o uso de recursos, sempre considerando a ordem definida das prioridades e dos tempos originais de solicitação de alocação, sem gerar uma sobrecarga na grade por movimentações de elementos.

5.1.2 Gerenciador de Nó

Os gerenciadores de nó (NM) são responsáveis por executar instâncias de atividades, processos e controladores das aplicações de grade. Cada gerenciador de nó contém um executor de atividades e um interpretador de processos e controladores. O interpretador escalona a execução dos elementos de processo, requisita a seleção e a alocação de nós para esses elementos, e invoca, controla e monitora a execução remota desses elementos.

Cada gerenciador de nó mantém informações de quais atividades, processos e controladores devem ser executados no servidor nó, qual o poder de processamento e memória necessários para cada atividade, qual o estado de execução dos elementos, parâmetros de entrada e resultados produzidos por elementos executados, o estado das variáveis internas e ponto de execução atual de cada elemento. Ele mantém em seu sistema de arquivos os documentos XML da especificação dos elementos, classes compiladas das atividades e arquivos de bibliotecas e de dados utilizados pelos elementos.

De forma a prover a recuperação do estado dos dados e o ponto de execução dos processos e controladores, na mesma ou em outra máquina, no caso de falhas do servidor nó, o gerenciador de nó persiste regularmente o ponto de execução e as variáveis internas do processos e controladores interpretados em bancos de dados relacionais compartilhados por servidores nós do grupo. As atividades sinalizadas como recuperáveis, executadas pelo gerenciador de nó, armazenam o seu ponto de execução periodicamente em um desses banco de dados para uma posterior restauração ou migração para outro servidor nó do grupo. O estado dos objetos compartilhados, instanciados nos controladores, armazenam as suas alterações nesses mesmos bancos de dados, para que eles sejam recuperados posteriormente no mesmo ou em outro servidor nó.

O serviço gerenciador de nó é iniciado por meio da execução do comando *xavantes_ns (Xavantes Node Server)*. Ao iniciar, o serviço comunica ao gerenciador de grupo local sua disponibilidade, recebe as atividades e os processos alocados a ele, obtém dados sobre esses elementos do repositório compartilhado e continua a sua operação normal.

A Figura 5.3 apresenta a estrutura interna do gerenciador de nó. As duas camadas inferiores são as mesmas para todos os serviços: a máquina virtual Java (*Java Virtual Machine*) é utilizada para executar o serviço; e o ambiente de execução (*Runtime Environment*) contém bibliotecas básicas, do *Java Runtime Library*, utilizadas para acesso aos recursos da máquina local, acesso aos repositórios de dados, e para a comunicação com outras máquinas. O ambiente de execução inclui a biblioteca *BeanShell* (*BeanShell Library*), que contém classes para executar *scripts* Java e para obter e alterar os valores das variáveis no contexto destes *scripts*. Ela é utilizada para executar os *scripts* Java que podem ser inseridos nos elementos de processo. As funcionalidades de cada componente das camadas superiores são descritas nas próximas seções.

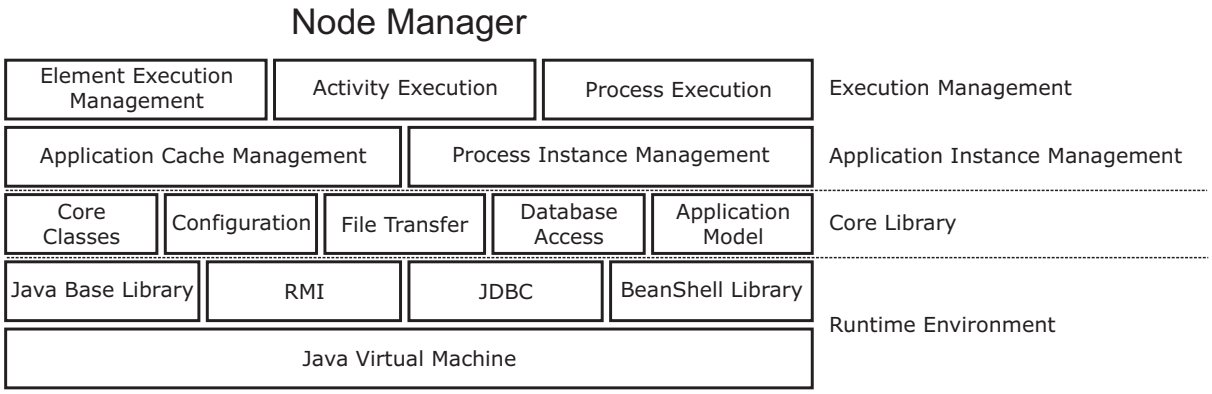

Figura 5.3: Estrutura do Gerenciador de Nó (NM).

Gerenciamento de *Cache* da Aplicação (*Application Cache Management*)

O gerenciador de nó, por meio do serviço de gerenciamento de *cache* da aplicação, é responsável por atualizar todos os arquivos necessários para executar os elementos alocados ao servidor nó. Para isso, ele mantém um diretório por aplicação relacionada aos elementos alocados, com o mesmo nome do diretório domínio da aplicação no gerenciador de grupo, que contém todos os arquivos da aplicação; e um subdiretório por instância dessa aplicação no servidor nó, para gravar arquivos temporários e de dados dessa instância.

Atualização do *Cache* da Aplicação Quando algum elemento de uma aplicação é alocado para executar no servidor nó, o gerenciador de nó verifica se os arquivos que especificam a aplicação estão atualizados no cache local da aplicação em relação ao diretório do domínio da aplicação no grupo que a especificou. Os arquivos que precisam ser atualizados são obtidos do domínio da aplicação e sobrescritos no diretório do cache da aplicação. Essa atualização ocorre assincronamente, antes do período alocado de execução do elemento, de forma a não impactar o tempo total de execução desse.

Gerenciamento de Instâncias de Processo (*Process Instance Management*)

Quando o gerenciador de grupo requisita a inserção de um processo ou controlador na lista (*pool*) de elementos a serem executados pelo gerenciador de nó e após a especificação em XML do processo, ou controlador, estar atualizada no *cache*, o serviço de gerenciamento de instâncias de processo lê o conteúdo desse arquivo e gera uma árvore sintática que o representa o elemento, formada por instâncias das classes do modelo de aplicação. Essa árvore é iniciada com valores padrão nas variáveis que representam o estado de uma

instância de processo, ou controlador, e depois é armazenada em tabelas correspondentes de um dos repositórios compartilhados do grupo. Ela é posteriormente utilizada pelo interpretador, com estado recuperável, para realizar o comportamento previsto do elemento. Esse serviço também gerencia as modificações do estado dessa árvore do elemento no repositório associado, contendo funções de gravação e leitura de cada elemento filho do processo, ou controlador, tais como controladores e invocação de processos e atividades, de forma a persistir o estado atual do processo em repositórios compartilhados e recuperá-lo quando necessário.

Criação de Instâncias de Processos e de Controladores O gerenciador de nó é responsável por criar uma árvore sintática em memória que representa uma instância de processo ou controlador em memória, interpretável pelo executor de processos, para cada tipo de processo ou controlador alocado para esse gerenciador de nó. A árvore sintática, que representa uma instância do processo ou do controlador, é criada a partir da análise sintática (*parsing*) do arquivo XML que contém a definição desse elemento. O arquivo XML que contém a definição do processo tem o mesmo nome que o processo, e o que tem o controlador, tem o nome do processo que o contém. Durante o *parsing*, são criadas instâncias de classes, do modelo de aplicação, que representam cada um dos marcadores da definição do processo, ou controlador, mas com os valores iniciais. Por exemplo, no *parsing* de um processo, ao ser encontrado um controlador filho, é criada uma instância de *InvokeController*, para invocar o controlador em outro, ou no mesmo, gerenciador de nó. Após ser criada essa árvore sintática em memória, ela é persistida em um dos repositórios compartilhados utilizando o serviço de "Persistência de Instâncias de Processos e de Controladores".

Persistência de Instâncias de Processos e de Controladores Durante a interpretação de uma instância de processo ou de controlador, são armazenados os estados dos dados e dos pontos de execução da instância, tal como elementos filhos sendo executados e suas variáveis internas, para posterior recuperação em caso de falhas de sistema do gerenciador de nó. O gerenciador de nó permite que o executor de processos insira um novo processo ou controlador no repositório compartilhado, criando registros para representar os objetos da árvore sintática nas tabelas correspondente às classes do modelo; e armazene o estado atual de um determinado processo ou controlador em um repositório compartilhado, alterando os campos que correspondentes aos valores dos objetos do modelo que foram modificados em relação à última persistência desse elemento.

Recuperação de Instâncias de Processos e de Controladores Ao ser iniciado, após uma falha ou reintegração como recurso disponível do grupo, o gerenciador de nó

obtém do gerenciador de grupo quais são os processos e controladores alocados a ele e recupera o estado de execução desses elementos para continuar a sua interpretação. Iniciando do objeto que representa o processo ou o controlador, são recuperados o valor das variáveis internas e o ponto de execução desse elemento. Cada linha de execução (*thread*), representando cada caminho que estava ativo, é recriada, com seu ponto de execução e estado das variáveis internas. As invocações de atividades, controladores e processos em execução são recuperadas, para esperar os elementos invocados terminarem, ou obter os resultados dos gerenciadores de nó alocados a eles. Pode haver necessidade de um reescalonamento de elementos invocados. O estado de execução armazenado sempre é consistente, não contendo valores sendo modificados por elementos filhos (*code*) em execução, por exemplo. O estado dos objetos compartilhados são consistentes quando recuperados, pois o acesso a esse é controlado e o seu estado é guardado em momentos independentes. Detalhes da recuperação de processos serão discutidos em detalhes na Seção 6.3. Após recuperado o estado de execução, a interpretação prossegue processando os próximos elementos filhos e esperando o término dos ativos.

Gerenciamento de Execução de Elemento (*Element Execution Management*)

O gerenciador de nó é responsável por gerenciar a execução dos elementos que estão alocados a ele, desde a inserção na lista (*pool*) de elementos, passando pela invocação da atualização dos arquivos necessários, recepção dos parâmetros iniciais, execução, até o envio de resultados. Ele informa o estado atual do gerenciador de nó para o gerenciador de grupo local, os eventos de alteração de estado de execução dos elementos, armazena os parâmetros iniciais e resultados no repositório compartilhado, e inicia e controla a execução de processos, controladores e atividades. Para isso o serviço mantém listas (*pools*) de processos e controladores por estado de execução, listas (*pools*) de atividades por estado de execução, e uma tabela com as informações sobre esses elementos.

Informação de Disponibilidade do Gerenciador de Nó O gerenciador de nó se autentica no gerenciador do grupo local e informa periodicamente a disponibilidade dos recursos do nó associado, ao iniciar. Caso não seja informada a disponibilidade dos recursos na periodicidade especificada, o gerenciador de nó pode ser desconectado ou desativado, não sendo mais alocado para executar processos, controladores ou atividades. As informações de disponibilidade dos recursos repassadas são poder de processamento e memória volátil e persistente disponíveis para o gerenciador de nó, bibliotecas presentes, e outros recursos específicos.

Consulta sobre Dados de Execução de Elementos Os gerenciadores de cliente da grade, bem como os invocadores de elementos, podem consultar os dados agregados e es-

pecíficos dos elementos alocados a um gerenciador de nó. O gerenciado nó pode informar quais processos, controladores e atividades foram alocados para ele e dados específicos sobre cada um desses elementos. Entretanto, a fonte primária sobre informações de alocação de elementos a gerenciadores de nó e de estado de execução desses elementos é o gerenciador de grupo. Os dados mantidos por elemento são identificador (UID), nome e tipo do elemento, classificação (processo, controlador ou atividade), estado de execução, indicador de atualização dos arquivos da aplicação, identificador do elemento superior, identificador do invocador, esforço de processamento e tamanho total das mensagens trocadas, poder de processamento e memória mínimo requeridos, bibliotecas necessárias, tempo previsto de início e fim de execução e valores dos parâmetro de entrada e de saída.

Controle de *Pipeline* de Execução Para um elemento de processo ser completamente executado, ele deve passar por uma seqüência de etapas de execução, na forma de uma linha de execução (*pipeline*). Em cada etapa uma ação específica da execução do elemento é realizada para que seja possível passar para a próxima etapa. Um exemplo de *pipeline* de execução, com várias etapas e ações correspondentes, é a linha de montagem de um carro, em que primeiro o motor é incorporado ao chassi, depois a carroceria fixada por cima, o painel, e assim por diante. O estado de execução do elemento determina qual é a etapa corrente de execução desse no *pipeline*. Especificamente, as etapas do *pipeline* de execução de um elemento são: atualização de arquivos da aplicação no *cache*, espera de mensagem com parâmetros de entrada, execução de código do elemento, envio de mensagem com os parâmetros de saída (ou exceção), e remoção do elemento do servidor nó. O gerenciador de nó controla qual evento esperar e qual ação realizar para cada elemento alocado a ele, dependendo do estado de execução dos elementos, bem como a ordem do processamento desses eventos e ações, por meio de listas de processos por estado de execução e listas de atividades por estado de execução. Os eventos esperados, as ações realizadas e as mudanças de estado de execução para os elementos de cada lista são apresentados abaixo:

- Após ser alocado no gerenciador de grupo o gerenciador de nó para o elemento, esse gerenciador de nó é comunicado e ele insere o elemento na lista de elementos "não-prontos".

- Para cada elemento "não-pronto" da lista, são atualizados os arquivos de aplicação no *cache*, se necessário, e esperada a mensagem inicial, com os parâmetros de entrada, vinda do invocador. Seu estado é alterado para "pronto" e o gerenciador de grupo é notificado sobre essa alteração.

- Para cada elemento "pronto" é esperado, do gerenciador de grupo, o evento indicando que este pode ser executado. A sua execução é iniciada utilizando um executor

de atividades, se o elemento for uma atividade, ou um executor de processos, se o elemento for um processo ou um controlador. Seu estado é alterado para "em execução".

- Para cada elemento "em execução" é esperado o seu término ou uma exceção não tratada. Se o elemento terminar com sucesso, seu estado é alterado para "terminado". Se for levantada uma exceção e o elemento for recuperável, o executor suspende a sua execução e seu estado é alterado para "suspenso". Se for levantada uma exceção e o elemento não for recuperável, e.g. uma atividade não recuperável, o executor aborta a sua execução e seu estado é alterado para "abortado". O gerenciador do grupo pode requerer explicitamente a suspensão ou aborto do elemento, fazendo com que o gerenciador de nó execute a mesma ação para o respectivo caso. O gerenciador de nó tenta enviar a mensagem final com os parâmetros de saída ou possível exceção ao invocador no caso de término ou exceção levantada, respectivamente. Em qualquer caso, o gerenciador de grupo é notificado sobre essa alteração.

- Para cada elemento "terminado" é esperado que o gerenciador de grupo solicite a sua remoção do servidor nó, após o invocador solicitá-la ao gerenciador de grupo.

- Para cada elemento "suspenso" é esperado que o gerenciador de grupo invoque a sua execução, acionando o executor e alterando o seu estado para "em execução"; ou que o gerenciador de grupo cancele a execução local após um realocação, removendo-o do servidor nó; ou que ele aborte a sua execução, acionando o executor e alterando o estado do elemento para "abortado".

- Para cada elemento "abortado" é esperado que o gerenciador de grupo solicite a sua remoção do servidor, após o invocador solicitá-la ao gerenciador de grupo.

Execução de Atividades (*Activity Execution*)

O gerenciador de nó executa as atividades alocadas ao servidor nó e permite que as variáveis internas e do ponto de execução das atividades sejam armazenados, para que essas atividades depois sejam recuperadas no mesmo servidor nó ou em outro. Várias instâncias de atividades do mesmo ou de diferentes tipos podem ser executadas simultaneamente no servidor nó, distribuídas em um ou mais processadores da máquina. O momento de início de execução de cada atividade é determinado pela agenda de atividades do servidor nó mantida pelo gerenciador de grupo. As atividades em execução podem terminar com sucesso, ou serem suspensas ou abortadas, por falha ou por ordem do gerenciador de grupo ou do invocador. Somente atividades recuperáveis podem armazenar e recuperar as suas variáveis internas e o seu ponto de execução para serem suspensas e continuadas, respectivamente.

As atividades, no presente modelo computacional, são vistas como instâncias dos tipos de atividades especificados na aplicação. Elas são implementadas como objetos criados a partir de classes específicas, derivadas de *Activity* do modelo de programação. Essas classes específicas de atividade são geradas a partir da compilação da especificação de um tipo de atividade em XML ou escritas diretamente pelo programador, conforme descrito na Seção 5.1.1.

O executor de atividades é o serviço responsável por executar as atividades pelo gerenciador de nó. Existe um executor por domínio de aplicação que possuem atividades alocadas ao servidor nó. Isso por causa de uma necessidade técnica de espaços distintos de nomes de classes por domínio de aplicação. Assim cada domínio possui um *Class Loader* [Mic06] específico para carregar o executor de atividades e as classes do domínio. Esse executor pode ser reinicializado para recarregar classes atualizadas, caso o domínio da aplicação seja atualizado na origem.

Obtenção dos Parâmetros de Entrada Antes de executar uma atividade, o executor da atividade deve receber uma mensagem com os parâmetros iniciais da atividade. Após recebida a mensagem, advinda do invocador da atividade, seja ele um cliente ou um gerenciador de processos, os valores desses parâmetros são armazenados no repositório compartilhado associado, para posterior recuperação. O serviço "Gerenciamento de Execução de Elemento" é informado de que foi recebido os parâmetros de entrada da atividade para dar prosseguimento ao *pipeline* de execução.

Disparo de Execução de Atividades O executor dispara a execução de uma atividade quando requisitado pelo serviço "Gerenciamento de Execução de Elementos", que controla o *pipeline* de execução. Para isso, ele cria uma nova linha de execução (*thread*) que recupera os valores dos parâmetros de entrada; inicia os campos correspondentes do objeto que implementa a atividade; e invoca o método *run* desse objeto para efetivamente executar o código da atividade.

Retorno dos Resultados ou Exceção Há três eventos que interrompem a execução de uma atividade: o seu término bem sucedido, o aborto e a suspensão de sua execução. Os dois últimos podem ocorrer quando é levantada uma exceção de sistema ou de aplicação não tratada internamente, ou quando o sistema ou o usuário os requisitam explicitamente. Se a atividade é recuperável e ocorre uma exceção, a sua execução pode ser suspensa ou abortada, dependendo do tipo de exceção; por outro lado, se a atividade não é recuperável, ela é necessariamente abortada.

No término bem sucedido da atividade, os parâmetros de saída são armazenados no repositório compartilhado; no aborto ou suspensão, a exceção não tratada ou que identifica

a requisição dessa ação é armazenada no repositório. O executor tenta enviar a mensagem com os parâmetros de saída (resultados) ou com a exceção para o invocador da atividade. Se não for possível enviar a mensagem, o resultado ou a exceção fica armazenada no repositório para posterior consulta pelo invocador. O serviço "'Gerenciador de Execução de Elemento" é então informado sobre o término, aborto ou suspensão da atividade.

Persistência de Atividades O executor de atividades permite a persistência das variáveis internas e do ponto de execução das atividades recuperáveis em repositórios compartilhados, para que esses valores sejam restaurados em caso de falhas do servidor nó ou migração de atividades e a atividade possa continuar a sua execução a partir do ponto de execução salvo, com os valores recuperados das suas variáveis internas. O serviço não suporta a persistência automática das variáveis internas e do ponto de execução da atividade em *checkpoints*, por esse motivo o programador da atividade deve explicitamente invocar uma operação do serviço, em pontos chaves, para armazenar esses valores. Nesses pontos, a atividade também deve persistir as alterações realizadas nos objetos compartilhados por meio de uma operação de sincronização (*commit*) das transações correntes. Pode-se implementar atividades recuperáveis construindo-se uma ciclo (*loop*) interno de verificação do ponto de execução para se identificar qual é o próximo passo a realizar na atividade. As persistências não devem ocorrer muito freqüentemente, para não sobrecarregar o sistema, e nem devem ser demasiadamente raras, para que uma falha não ocasione uma grande perda de tempo de computação na recuperação e nem bloqueiem registros de objetos compartilhados por um longo período.

Recuperação de Atividades Em caso de falha do servidor nó ou migração de atividade, o gerenciador de nó automaticamente restaura os valores das variáveis internas e o ponto de execução das atividades recuperáveis. Posteriormente ele invoca o método *run* de cada atividade. Na especificação desse método, a atividade recuperável recarrega as variáveis internas, verifica qual é o ponto atual da execução e executa os próximos passos correspondentes.

Execução de Processos (*Process Execution*)

O gerenciador de nó, por meio do executor de processos, executa instâncias de processos e controladores. Instâncias de processos são criadas a partir de tipos de processo especificados nos documentos XML, de mesmo nome do tipo, atualizados no *cache* da aplicação no servidor local, utilizando-se o serviço "Criação de Instâncias de Processos e de Controladores". Instâncias de controladores são criadas do mesmo modo. Uma instância de processo ou controlador é representada em memória por uma árvore sintática de objetos. A execução de um processo ou controlador utiliza a árvore sintática de objetos que representa

uma instância desse elemento. A execução de um processo ou controlador ocorre com o escalonamento e a execução de seus elementos filhos. O escalonamento dos elementos filhos ocorre paralelamente à execução desses mesmos elementos. No escalonamento dos elementos filhos, o executor calcula uma prioridade relativa para cada elemento; escolhe elementos a alocar; e seleciona e aloca servidores nós para esses elementos. A execução dos elementos filhos segue os fluxos de controle paralelos, iterativos e condicionais definidos pelo processo ou controlador; executa *scripts* em Java para modificar variáveis de dados e de controle e obter resultados de expressões; distribui mensagens com dados para cada elemento filho alocado previamente, espera a resposta desses elementos e trata exceções. O escalonamento e a execução dos elementos filhos ocorrem respeitando-se a semântica específica do processo ou do tipo do controlador, conforme definido pela linguagem Xavantes, mas realizando tarefas complementares, realizando ações sobre os elementos filhos diretos. O escalonador, que roda na mesma máquina do elemento, monitora a execução dos elementos filhos e realiza reescalonamentos, se forem necessários para otimizar o uso de recursos e a velocidade de execução desse elemento, no ambiente dinâmico da grade. O executor persiste as modificações ocorridas na execução dos processos e controladores, em etapas definidas, para uma posterior recuperação do elemento em caso de falhas de sistema ou migração. É permitida a modificação dinâmica do processo para tratar de falhas de aplicação não previstas no código.

O escalonamento e a execução dos elementos filhos (atividades, processos e controladores) de um elemento ocorrem em paralelo, sendo cada elemento filho processado por um *pipeline* de tarefas básicas:

- um elemento filho não escalonado é selecionado, segundo a ordem de execução, e para ele é atribuída uma prioridade relativa entre os demais elementos filhos;
- o melhor servidor nó da grade é selecionado e alocado para a execução do elemento filho, utilizando o gerenciador de grupo local;
- o código do elemento filho é atualizado no servidor nó, se necessário, por demanda do último;
- a mensagem inicial, com os parâmetros iniciais, é enviada do elemento para o elemento filho;
- o elemento filho é executado no servidor nó alocado, quando requisitado pelo gerenciador de grupo, segundo a ordenação da agenda de alocação mantida para o nó;
- uma mensagem com resultados ou uma exceção é retornada do elemento filho para o elemento;

- se o elemento filho for suspenso, a execução pode ser continuada após ser resolvida a falha;
- se o elemento filho for terminado ou abortado, o processamento é finalizado.

Escalonamento de Elementos Filhos de Processos e Controladores O escalonamento básico de elementos filhos de processos e de controladores é local, compreendendo somente a visão dos elementos filhos (atividades, controladores e processos) imediatos. Entretanto, o mecanismo de prioridades de execução permite considerar cálculos globais, que levam em conta todos os elementos do processo *. O início do escalonamento de elementos filhos de um processo ou controlador ocorre tão logo ele é alocado em um servidor nó, diferentemente da execução desse elemento, que se inicia somente quando o gerenciador de grupo requisita. Inicialmente, o escalonador cria uma lista de elementos filhos a escalonar e calcula a prioridade relativa para cada elemento filho desta lista. O escalonamento efetivo de cada elemento filho desta lista ocorre em duas fases: seleção do elemento filho a alocar; e seleção e alocação de servidor nó da grade para executá-lo. A seleção do elemento filho a alocar respeita a ordem de execução entre eles: um elemento filho somente pode ser alocado após o anterior de um caminho sê-lo. Entretanto, elementos filhos sem dependência de execução entre eles podem ser alocados em paralelo, e.g. os pertencentes a caminhos distintos de um controlador *par*. Isso permite aumentar a eficiência do algoritmo de escalonamento, pois não é necessário esperar o resultado de uma alocação para realizar a próxima. Ademais, o escalonamento dos elementos filhos de cada processo ou controlador filho pode se iniciar assim que ele for alocado em um servidor nó, e este, por sua vez, pode alocar elementos filhos de processos e controladores netos, e assim recursivamente, promovendo-se, portanto, um escalonamento distribuído, altamente paralelo e mais disponível do que o centralizado, pois não há ponto único de falha.

Para requisitar a seleção e a alocação de servidor nó para o elemento filho ao gerenciador de grupo, o escalonador calcula a prioridade relativa, o tempo mínimo de início e o tempo máximo de fim desse elemento. Se o elemento for uma atividade, ele também repassa as informações de requisitos necessários, tais como poder de processamento e memória mínimos, e fontes de dados e bibliotecas necessárias. A partir desse ponto, independentemente da ordem efetiva de alocação, os serviços do *middleware* respeitam a prioridade relativa de execução e o tempo de início atribuídos a cada elemento dentro da aplicação na seleção do servidor nó e na alocação do elemento na agenda do servidor nó, de forma que a ordem de execução mais eficiente entre os elementos seja alcançada de acordo com o algoritmo de escalonamento utilizado.

*Na Seção 6.2.1 são especificados algoritmos que utilizam o sistema de prioridades para realizar escalonamentos globais

Criação da Lista de Elementos Filhos a Alocar O escalonador aloca somente alguns elementos filhos inicialmente, ao invés de todos. Depois, conforme a requisição do executor de processos, ele aloca os demais, em uma ou mais iterações. O objetivo dessa funcionalidade é reduzir a utilização de agendas de atividades e de processos e evitar a alocação de recursos para elementos com execução incerta, e.g. o número de elementos filhos em um controlador *parfor* somente é conhecido em tempo de execução. De qualquer forma, o sistema de prioridades garante que mesmo que os elementos filhos mais importantes para a redução do tempo da aplicação sejam alocados posteriormente, eles são sempre executados antes de elementos filhos menos importante, pela realocação de atividades.

Cálculo da Prioridade de Execução de Elementos no Escalonamento A flexibilidade no mecanismo de escalonamento está no suporte à alocação de servidores nó baseada na prioridade relativa entre elementos de uma aplicação (macro processo) a serem executados, de forma a executar elementos mais críticos para o tempo de execução do processo em servidores nó com maior poder de processamento. No mecanismo básico de escalonamento, a prioridade relativa é calculada como a distância entre o elemento (atividade, controlador ou processo) em questão e o final da aplicação, em tempo estimado de execução, do maior caminho que parte desse elemento até o final do processo principal. O cálculo do tempo estimado de execução de cada elemento é calculado como o esforço total necessário de processamento do elemento dividido pelo poder computacional de uma máquina padrão, mais o tamanho total das mensagens trocadas entre esse elemento e o seu elemento pai dividido pela velocidade de comunicação de uma rede padrão. Se o elemento é um controlador ou processo, o esforço de processamento é calculado como a somatória dos esforços de processamento dos elementos filhos, recursivamente. Como o processo é estruturado e o esforço total de processamento de cada elemento é suficiente para calcular o maior caminho de um elemento até o final do elemento pai, não é necessário o cálculo recursivo dos maiores caminhos, somente um pré-cálculo recursivo do esforço de total de processamento dos elementos compostos (processos e controladores). Uma vez que um elemento composto tem um tempo calculado entre ele e o final do processo, pode-se calcular para cada elemento filhos deste o tempo entre o elemento filho e o final da aplicação (macro processo) por meio do cálculo do tempo entre o elemento filho e o final do elemento pai (composto) mais o tempo do elemento pai até o final do processo, previamente calculado. O maior caminho entre um elemento filho e o final do seu elemento pai pode ser obtido localmente, regredindo-se do último elemento filho deste, até o elemento filho em questão. O maior caminho (tempo) de cada elemento filho é calculado como o maior tempo dos imediatamente próximos elementos filhos até o final do elemento pai comum, mais o próprio tempo do elemento filho. Como esse grafo é acíclico, e um elemento com-

posto somente pode ter elementos filhos em seqüência ou em paralelo, não há repetição de seleção de elementos e o tempo de um elemento até o final do seu elemento pai pode ser calculado em complexidade computacional $O(n)$ — onde n é o número de elementos filhos. O cálculo do esforço de processamento de um controlador ou de um processo, que é a soma do esforço de processamento de todos os seus elementos filhos, também é realizado em complexidade computacional $O(n)$.

Alocação de Elementos Filhos no Escalonamento A alocação de cada elemento filho ocorre em diversas etapas. Um elemento filho da lista de elementos a alocar que possui todos os antecessores alocados é escolhido para a alocação. O tempo mínimo de início do elemento a alocar é igual ao maior tempo de término entre os elementos antecessores. Os tempos de término são obtidos da alocação realizada. Se não houver antecessor, esse tempo é igual ao tempo de início do elemento pai. A prioridade relativa é obtida da forma descrita no parágrafo "Cálculo da Prioridade de Execução de Elementos no Escalonamento". O tempo máximo de execução de elemento filho é proporcional ao tempo máximo de execução do elemento pai, na razão do tempo de execução do elemento filho, previsto utilizando-se uma máquina padrão e uma rede padrão, com relação ao tempo de execução do caminho crítico do elemento pai, previsto utilizando-se os mesmo recursos. O tempo máximo de fim do elemento filho é calculado como tempo mínimo de início mais o tempo máximo de execução deste.

Após calculados a prioridade relativa, o tempo mínimo de início e o tempo máximo de fim de um elemento filho, é solicitado ao gerenciador de grupo a sua alocação. Se o elemento filho for uma atividade, o gerenciador de grupo retorna o tempo de início e o tempo de fim previstos para a execução da atividade, bem como o servidor nó alocado. Se o elemento filho for um controlador ou um processo, o gerenciador de grupo requisita a alocação de um servidor nó para este imediatamente. No servidor nó, o controlador ou o processo escalona todos os elementos netos, e assim recursivamente, para obter o tempo final de sua alocação. O tempo de início do controlador ou do processo é determinado pelo tempo de início do elemento neto com menor tempo de início, e o tempo de fim é determinado pelo tempo de fim do elemento neto com maior tempo de fim. O escalonador do elemento filho é notificado desses tempos para o prosseguimento do escalonamento. Enquanto o tempo final não é retornado, provisoriamente é calculado um tempo final estimado para o controlador ou processo, utilizando-se como base a máquina e a rede padrão de um lado, e o esforço de processamento e de comunicação estimado recursivamente para o elemento, de outro lado. Em qualquer um dos casos, o elemento filho seguinte é escolhido para a alocação. Alocações de vários elementos filhos sem dependência de execução podem ser realizadas paralelamente.

Execução de Processos e Controladores Para executar um processo ou um controlador, o gerenciador de processos inicia uma *thread* e invoca o interpretador Xavantes passando uma árvore de objetos que representa a especificação do elemento correspondente com a invocação a seus elementos filhos. O interpretador executa o processo ou o controlador de acordo com sua semântica (fluxo condicional ou não, iterativo ou não, seqüência ou paralelo), criando uma *thread* para cada caminho de execução do controlador paralelo, ou somente uma *thread* para o processo ou controlador seqüencial. Ele utiliza os parâmetros iniciais para o início da execução. Durante a interpretação do elemento, ele altera as variáveis internas do elemento de acordo com o fluxo de dados, e invoca e controla os elementos filhos de acordo com o fluxo de controle. Os valores da variáveis e os pontos de execução de cada caminho do elemento são mantidos no objeto correspondente da árvore sintática. Esses dados são armazenados periodicamente na base de dados para garantir a recuperação automática do elemento. No final da execução, os parâmetros de saída são lidos da árvore de objetos e são armazenados para posterior envio ao invocador do elemento.

Distribuição de Elementos de Processo Uma vez alocado os servidores nó para executar os elementos filhos, esses elementos são distribuídos para execução. Como a atribuição de prioridades e as alocações são realizadas previamente, os elementos podem ser distribuídos bem antes de ter a sua execução iniciada, juntamente com os arquivos e bibliotecas associados. Dessa forma o ambiente é preparado antecipadamente para executar imediatamente um elemento quando: for enviada a mensagem contendo os parâmetros de entrada a partir do gerenciador de processo invocador, que contém o elemento pai; e o elemento for escolhido para ser executado, de acordo com a agenda do servidor nó, mantida pelo gerenciador de grupo.

A distribuição é realizada por demanda, assim, logo que é alocado um servidor nó para um elemento, o gerenciador de processos ou o gerenciador de atividades requisita, ao gerenciador do grupo em que é definido o elemento, os arquivos contendo as especificações em XML, as bibliotecas e os dados necessários para se construir o ambiente de execução para esse elemento. Nesse passo é criado um canal de comunicação para a cópia desses arquivos do gerenciador de grupo requisitado ao sistema de arquivos do servidor nó alocado.

Recuperação de Processos e Controladores O executor de processos armazena as variáveis internas e os pontos de execução dos processos e controladores sob seu controle para posterior recuperação, no mesmo servidor ou em outro, em caso de falhas do servidor nó. Os objetos criados em memória que representam os elementos de processo tem um correspondente conjunto de tabelas no repositório compartilhado do gerenciador de nó

para armazenar um estado consistente do seu ponto atual de execução. Em caso de falhas ou migração de processos ou controladores, essas tabelas podem ser utilizadas para recriar essa mesma estrutura de objetos em memória e reiniciar a execução do processo ou controlador no ponto em que foi salvo o seu último estado consistente.

5.1.3 Gerenciador de Cliente

Os gerenciadores de cliente (CM) são responsáveis por especificar, implantar e invocar processos e atividades, bem como monitorar a execução desses elementos e recursos da grade. Cada gerenciador de cliente contém um gerenciador de aplicações, um monitor de recursos e um controlador de execução de elementos. O gerenciador de aplicações é utilizado para implantar aplicações no domínio específico do grupo, transferindo arquivos da aplicação para o gerenciador de grupo. O monitor de recursos consulta o gerenciador de grupo para apresentar ao usuário (cliente) o estado dos recursos da grade. O controlador de execução de elementos, instancia, invoca e monitora a execução dos elementos pelo cliente. Acima do gerenciador de cliente podem existir ambientes gráficos de desenvolvimento de aplicações em grade que auxiliam a especificação e o monitoramento de processos e atividades, bem como o monitoramento dos recursos da grade.

Cada gerenciador de cliente mantém informações de quais atividades e processos foram invocados pelo usuário; qual é a alocação e o estado de execução de cada elemento; e quais foram os parâmetros enviados para os elementos e os resultados produzidos, se terminados. Ele mantém em seu sistema de arquivos os documentos XML da especificação dos elementos, classes compiladas das atividades, arquivos de bibliotecas de classes e arquivos de dados utilizados pelos elementos.

De forma a prover a recuperação das informações sobre os elementos invocados, no caso de falhas da máquina cliente, o gerenciador de cliente persiste a identificação, tipo, os parâmetros iniciais, alocação, estado de execução e resultados toda vez que um elemento é invocado ou seu estado é alterado. Ele persistente essas informações no banco de dados local.

O serviço gerenciador de cliente básico é iniciado por meio da execução do comando *xavantes_client (Xavantes Client)*. A sua classe principal, *Client*, também pode ser utiliza por ambientes de desenvolvimento referenciando as bibliotecas de classe Java *xavantes_client.jar* e *xavantes_core.jar*. A classe *Client* retorna uma instância da classe *ClientManager*. Esta instância contém todas as operações de interação com os serviços do sistema Xavantes, servindo de *front-end* para a grade. Entre essas, existem operações para: implantação da definição de processos e atividades no grupo local; invocação de processos e atividades, com a passagem de parâmetros iniciais de execução e informações para alocação; obtenção de resultados de atividades e processos invocados; controle e

monitoramento da execução de processos; e monitoramento dos recursos da grade.

Ao iniciar, o serviço gerenciador de cliente comunica ao gerenciador de grupo local sua disponibilidade, passa uma referência a sua interface *callback*, obtém dados sobre os elementos invocados do repositório local e continua a sua operação normal. A interface *callback* do gerenciador de cliente, passada para o gerenciador de grupo, é invocada assincronamento quando as informações relevantes sobre os elementos gerenciados são alteradas, quando o executor de algum elemento necessita obter novamente os parâmetros iniciais de invocação, ou quando algum elemento é terminado, para notificar sobre o seu resultado.

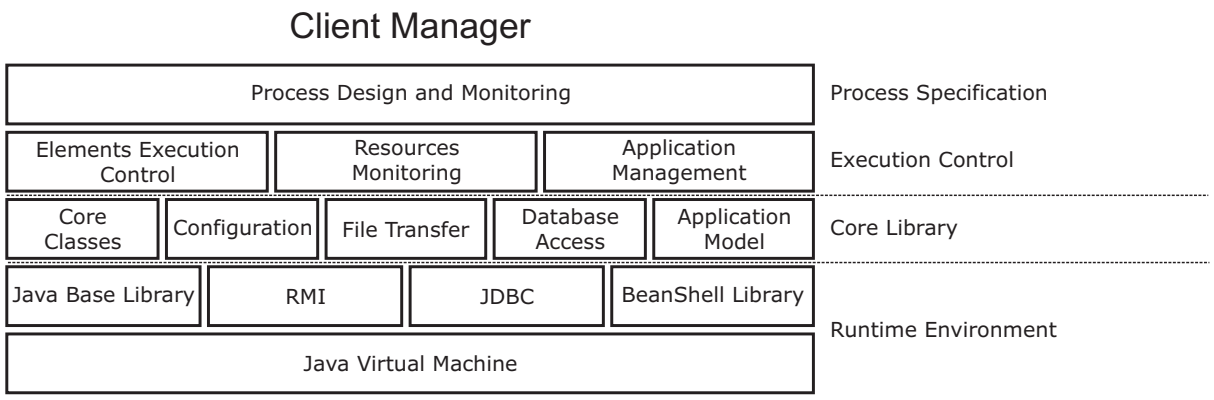

Figura 5.4: Estrutura do Gerenciador de Cliente (CM).

A Figura 5.4 apresenta a estrutura interna do gerenciador de cliente. As duas camadas inferiores são as mesmas para todos os serviços: a máquina virtual Java (*Java Virtual Machine*) é utilizada para executar o serviço; e o ambiente de execução (*Runtime Environment*) contém bibliotecas básicas, do *Java Runtime Library*, utilizadas para acesso aos recursos da máquina local, acesso aos repositórios de dados, e para a comunicação com outras máquinas. As funcionalidades de cada componente das camadas superiores são descritas nas próximas seções.

Gerenciamento de Aplicações (*Application Management*)

O serviço (ou módulo) de gerenciamento de aplicações tem por funcionalidade listar e implantar aplicações no gerenciador de grupo local. Para uma aplicação, ele obtém os arquivos da aplicação do diretório local da aplicação na máquina cliente, incluindo biblioteca de classes, especificação de processos em XML e arquivos de dados, e copia esses arquivos para o domínio de aplicação correspondente do grupo, que é um diretório para

a aplicação no gerenciador de grupo. Esse serviço permite listar os arquivos da aplicação no domínio da aplicação, no gerenciador de grupo.

Além da aplicação, cada instância da aplicação contém um diretório específico no gerenciador de grupo que pode conter arquivos de dados de entrada e saída específicos daquela instância. Esse serviço também implanta e lista os arquivos das instâncias de aplicação.

Monitoramento de Recursos (*Resources Monitoring*)

O monitoramento de recursos permite que usuários possam listar os grupos conhecidos do grupo local e os recursos existentes em cada grupo. Para cada grupo ou recurso é possível obter suas propriedades (atributos) e seu estado atual. Para fornecer esse serviço, o gerenciador de cliente comporta-se como um *front-end* para que o usuário consulte as informações mantidas pelo gerenciador de grupo local, no caso de recursos locais, ou mantidas pelos gerenciadores de grupos remotos conhecidos, no caso de recursos remotos.

Controle de Execução de Elementos (*Elements Execution Control*)

O serviço de controle de execução de elementos é responsável por fornecer uma interface ao usuário para invocação, monitoramento e controle de instâncias de atividades e processos. O usuário que deseja invocar uma atividade ou processo informa o nome do tipo do elemento, informações para alocação e os parâmetros iniciais de invocação. O serviço mantém essas informações em uma memória persistente, aloca um recurso utilizando o gerenciador de grupo e requisita a criação de uma instância desse elemento no recurso alocado, utilizando o gerenciador de nó. Durante a existência da instância, o gerenciador de cliente monitora as possíveis alterações de recurso para essa instância, informadas pelo gerenciador de grupo, e depois recebe assincronamente a resposta da execução ou a exceção resultante. Essa resposta ou exceção também é armazenada em memória persistente para posterior consulta pelo usuário.

As alterações de estado e de alocação das instâncias monitoradas são informadas ao gerenciador de cliente por meio da interface *callback* de monitoramento, cuja referência é passada para o gerenciador de grupo. Essas alterações também são armazenadas em memória persistente local. Cada usuário pode conter várias instâncias de elementos sendo executadas na grade paralelamente. Ao receber a resposta da execução do elemento, o usuário deve remover esse elemento do gerenciador de cliente.

Além da invocação e do monitoramento, esse módulo permite controlar a execução de cada instância. Um usuário pode requerer a suspensão, a continuação ou o aborto das instâncias sendo executadas pela grade. Nesses casos, o gerenciador de cliente também é informado do atendimento da requisição e do novo estado de execução do elemento.

Especificação e Monitoramento de Processos (*Process Design and Monitoring*)

A camada de mais alto nível do gerenciador de cliente fornece uma interface para especificação de processos e atividades diretamente no gerenciador do cliente, bem como para o monitoramento do estado de execução e variáveis de cada elemento do processo, após esse processo raiz ser invocado. Essa camada pode ser acoplada a uma IDE (*Integrated Development Environment*) para a especificação e monitoramento gráfico de processos pelos usuários.

Após especificado, o processo pode ser armazenado como um arquivo de especificação em XML ou pela inserção de registros de metadados que especificam um tipo de processo no repositório de aplicações do grupo. Um processo gravado como registros de metadados em repositórios pode ser facilmente alterado.

Para o monitoramento, o gerenciador de cliente pode requisitar informações de estrutura, estado de execução e variáveis internas de cada elemento do processo aos gerenciadores de nó alocados para a sua execução. Essas informações podem então ser visualizadas utilizando um ambiente gráfico.

5.2 Relacionamento entre Grupos

De forma a modelar uma arquitetura de grade de larga escala, o *middleware* deve permitir incorporar várias, potencialmente todas, redes locais, na forma de uma "Internet" de recursos compartilhados. Com essa finalidade, no *middleware* Xavantes todos os grupos locais são interligados entre si, diretamente ou indiretamente, por meio de gerenciadores de grupos (GMs), em uma topologia *peer-to-peer*.

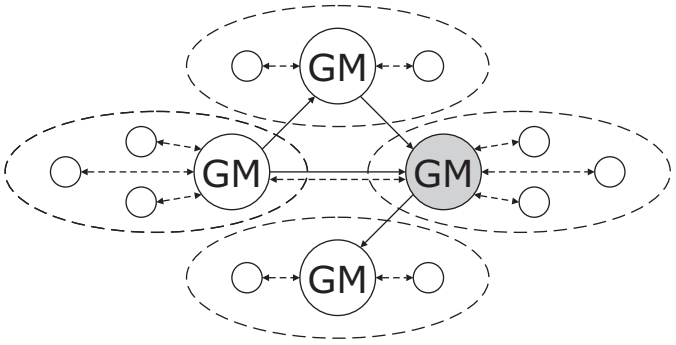

Figura 5.5: Relacionamento entre Gerenciadores de Grupo.

Como pode ser visto na Figura 5.5, cada gerenciador de grupo (GM) lida com requisições do seu grupo, a fim de registrar máquinas locais e monitorar sua disponibilidade.

Adicionalmente, gerenciadores de grupo se comunicam com gerenciadores de grupos pares. Nessas comunicações, cada gerenciador de grupo servidor exporta informações agregadas de disponibilidade média de poder de processamento e de memória, do seu grupo e de outros grupos dos quais obteve essa informação, para gerenciadores de grupos adjacentes (setas contínuas). Pelo caminho inverso, um gerenciador de grupo servidor recebe requisições de grupos pares para fornecer informações detalhadas de disponibilidade e de alocação (setas tracejadas). Dessa forma, se há recursos disponíveis em grupos externos, é possível que um gerenciador de grupo cliente, intermediando solicitações de gerenciadores de processo ou de cliente locais, requisite a grupos servidores pares a alocação de máquinas locais a esses para executar processos ou atividades pelo grupo requisitante. Depois de alocadas essas máquinas, então os arquivos com a especificação dos elementos de processo podem ser enviados do gerenciador de grupo local para essas máquinas, de forma que essas executem os elementos alocados de forma distribuída.

5.3 Modelo Dinâmico

As principais funções do *middleware* Xavantes de suporte a execução de processos estruturados em grades computacionais são: autenticação de usuários e de servidores, gerenciamento de informações sobre os recursos da grade, implantação de aplicações, escalonamento e execução de processos, execução de atividades nos servidores nó, acesso a dados compartilhados e recuperação do sistema e dos processos. Existem serviços no *middleware* para realizar cada uma dessas funções, apresentados na Seção 5.1. Esses serviços estão distribuídos entre os dois principais componentes: gerenciador de grupo e gerenciador de nó. Nessa seção descreveremos as principais funções do *middleware* e apresentaremos o seu comportamento dinâmico, mostrando como os vários serviços interagem para realizá-las.

5.3.1 Autenticação de Usuários e Servidores

Autenticação de Usuários Os dados dos usuários da grade são armazenados na tabela *Client_ Registry* do repositório do grupo local. Cada grupo local tem um conjunto de usuários e esses podem utilizar os recursos da grade por meio das permissões do grupo e prioridade máxima atribuída às suas aplicações. Esse cadastro contém campos de identificação do usuário, nome, senha, localização do gerenciador de cliente (máquina e porta de comunicação) e estado de ativação do cliente. Para utilizar a grade, o usuário deve se autenticar ao gerenciador de grupo local fornecendo nome, senha e localização do gerenciador de cliente (máquina e porta). No futuro, pretende-se inserir certificado digital do usuário aos registros, com chaves públicas distribuídas, para permitir um mecanismo de acesso seguro à grade, provendo ao sistema comunicação segura, com privacidade e

integridade de dados, autenticação e não-repúdio.

Autenticação de Servidores Os dados dos nós de um grupo são armazenados na tabela *Node_Service_Registry* do repositório desse grupo. Esse cadastro contém campos de identificação do nó, nome, senha e localização do gerenciador de nó (máquina e porta de comunicação), descrição da máquina, número de processadores, poder de processamento (em quantidade de milhares de instruções processadas por segundo), tamanho das memórias volátil e persistente e seu estado de ativação. Como extensão, associado a cada gerenciador de nó pode haver informações sobre a quais fontes de dados o servidor nó tem acesso e que bibliotecas ele possui. No aspecto de segurança de informação, esse registro poderia armazenar o certificado digital do gerenciador de nó, com chave pública distribuída, para permitir autenticação e comunicação segura na grade.

Para se registrar na grade, o gerenciador de nó deve se autenticar no gerenciador de grupo local fornecendo nome, senha e sua localização (máquina e porta) e uma referência ao objeto remoto que representa a interface do gerenciador de nó. Como resultado, o gerenciador do grupo autentica o gerenciador de nó e retorna a identificação única para esse.

Registro entre Grupos Os dados dos grupos pares da grade são armazenados na tabela *Group_Service_Registry* do repositório desse grupo. Esse cadastro contém campos de identificação do grupo, nome, senha e localização do serviço gerenciador de grupo (máquina e porta de comunicação), descrição do grupo, número de servidores nós, poder de processamento total, largura de banda interna ao grupo e largura de banda externa entre o grupo local e o referido grupo, indicador do relacionamento entre o gerenciador de grupo par e o grupo local (cliente, servidor ou ambos) e seu estado de ativação. É possível informar características adicionais do grupo, tais como bibliotecas, serviços e fontes de dados acessíveis no grupo. Para permitir comunicação segura e autenticação mais confiável entre grupos, esse registro poderia armazenar o certificado digital do gerenciador de grupo registrado, com chave pública, para permitir autenticação e comunicação segura na grade.

Um grupo par é registrado manualmente. O administrador do grupo local pode inserir registros no cadastro de grupos indicando alguns grupos servidores e clientes deste. Pretende-se especificar um serviço de diretório de grupos, contendo as informações seguras de todos os grupos conhecidos da grade, para que esse cadastramento recíproco, entre cada par de grupos, seja automatizado.

Uma vez cadastrado um gerenciador de grupo cliente em um gerenciador de grupo servidor, o último deve periodicamente informar as características do seu grupo, tais como nome, localização de seu gerenciador de grupo (máquina e porta), descrição, número de nós, poder de processamento total, largura de banda interna e referência ao objeto remoto

que representa a interface do gerenciador de grupo. A largura de banda entre os dois grupos pode ser obtida por meio de medições. O uso de certificados digitais, nesse caso, também permitiria a autenticação bilateral entre grupos desconhecidos, desde que esses certificados fossem assinados por uma entidade confiável.

5.3.2 Gerenciamento de Informações sobre Recursos da Grade

As informações relevantes do grupo mantidas pelo gerenciador de grupo são: características e disponibilidade dos servidores nó, agenda de atividades e de processos dos servidores nó, e características, disponibilidade e agenda agregadas de poder médio de processamento e tamanho médio de memória dos servidores nó dos grupos servidores conhecidos. As informações sobre os servidores nó são mantidas pelo gerenciador de grupo local de forma persistente para possível recuperação em caso de falhas. As características dos servidores nó e dos grupos são fornecidas no momento do seu registro, quando os servidores nó são inseridos ou quando os grupos são ativados na grade. Periodicamente, após registrados, os servidores nó enviam informações sobre o seu estado para os gerenciadores de grupo local, e os servidores de grupo pares enviam informações sobre o seu estado atual para os gerenciadores de grupo cliente registrados.

Informações sobre os Servidores Nó Os servidores nó são considerados ativos na sua autenticação. Após a autenticação, eles informam o seu estado periodicamente. Se depois um determinado tempo limite, o gerenciador de grupo não receber a mensagem periódica de algum servidor nó, ele o considera como *desconectado*. Realmente, pode ser o caso de desconexão intermitente ou reinicialização de rotina ocorrida em alguns sistemas operacionais. Após um tempo maior, se ainda nenhuma mensagem for recebida desse servidor nó, o gerenciador de grupo o considera *inativo*. Em ambos os casos, se alguma mensagem for recebida a partir dele ou se ele for registrado novamente, seu estado volta a ser *ativo*. Quando desconectado, nenhuma outra atividade é alocada ou enviada para esse nó; e quando inativo, todas as atividades alocadas devem ser realocadas para outro servidor nó.

Informações sobre Atividades e Processos Alocados para o Grupo O gerenciador de grupo contém informações sobre atividades e processos alocados para o grupo, invocados por clientes do grupo local ou de grupos clientes. As informações sobre atividades e processos, inclusive controladores, alocados são armazenados nas tabelas *Activity_Information* e *Process_Information*, respectivamente. Na tabela de atividades, cada registro contém a identificação da atividade, identificação do elemento pai, nome único da atividade em relação ao pai, tipo da atividade (com grupo, domínio e nome do tipo),

identificação do cliente, características de tempo mínimo de início e tempo máximo de fim de execução e tamanho em processamento e em mensagens trocadas com o elemento pai, poder de processamento e memória mínimos requeridos, identificação do executor (servidor nó) alocado à atividade, início e término esperado para a atividade no executor alocado, poder de processamento alocado, prioridade e estado de execução. Na tabela de processos, e controladores, cada registro contém a identificação do elemento, identificação do elemento pai, nome único do elemento em relação ao pai, tipo do elemento (com grupo, domínio e nome do tipo), características de tempo mínimo de início e tempo máximo de fim de execução e tamanho em mensagens trocadas com o elemento pai, identificação do executor (servidor nó) alocado ao elemento, início e término esperado para o elemento no executor alocado, prioridade e estado de execução.

Informações sobre as Agendas de Atividades e de Processos de cada Servidor Nó O gerenciador de grupo contém as agendas, em memória, de atividades e de processos para todos os servidores nós locais. As agendas de atividades e de processos são derivadas das tabelas que contém informações sobre as atividades e os processos alocados para o grupo, *Activity_Information* e *Process_Information*, respectivamente, conforme descrito no parágrafo anterior. Cada item das agendas de alocação de atividades contém a identificação da atividade alocada, tempo de início e tempo de fim previstos de execução, prioridade da atividade, e poder de processamento alocado para essa atividade. Cada item das agendas de alocação de processos contém a identificação do processo ou controlador alocado, tempo de início e tempo de fim previstos de execução e prioridade do elemento.

Os gerenciadores de grupo respondem pela seleção e alocação de servidor nó para executar processos, controladores e atividades. Os servidores nó alocados podem ser do mesmo grupo ou de um grupo servidor, em relação ao requisitante. Se o requisitante estiver no mesmo grupo, na tabela de elementos invocados deste grupo é inserida a informação de alocação do elemento. Se o requisitante for de um grupo cliente, nas tabelas de elementos invocados tanto do grupo cliente, quando do grupo servidor, são inseridas as informações sobre a alocação realizada. Em qualquer um dos casos, a alocação de um servidor nó de um grupo altera a agenda correspondente do gerenciador de grupo do servidor nó que foi alocado.

Informações sobre os Grupos Externos Cada gerenciador de grupo tem informações sobre as características e o estado dos grupos conhecidos. As principais características são trocadas no registro inicial entre grupos clientes e servidores. Posteriormente, dados agregados sobre o estado dos servidores nó de um grupo servidor e sobre a agenda agregada de disponibilidade dos seus servidores nó são enviados pelo gerenciador desse grupo

para o gerenciador dos grupos clientes, periodicamente. Essa agenda contém a disponibilidade média de poder computacional e de memória no grupo relacionado para cada período de tempo e faixa de prioridades. Essa informação é utilizada pelos algoritmos de escalonamento para a seleção prévia de grupos candidatos para a alocação de atividades e processos, conforme será explicado posteriormente.

5.3.3 Implantação de Aplicações

O gerenciador de grupo controla o acesso às especificações em XSPL e às bibliotecas de classes das aplicações implantadas no grupo local. As aplicações são compostas de processos, controladores e atividades especificadas em XSPL (*Xavantes Structured Process Language*), mais bibliotecas contendo classes e interfaces utilizadas para computação e invocação de serviços externos. As atividades de uma aplicação também podem ser implantadas como classes especializadas de *Activity* do Xavantes *framework*, compiladas em Java, agregadas em um arquivo compactado no formato de biblioteca de classes.

Implantação de uma Aplicação Depois de implementada a aplicação, como um conjunto de arquivos em XSPL e de bibliotecas de classes, esses são copiadas para um diretório específico do domínio da aplicação no gerenciador de grupo. Depois de implantada a aplicação, por meio do uso do cliente Xavantes é possível criar instâncias e executar processos e atividades codificadas, se referindo ao grupo e domínio dessa aplicação e ao nome do processo ou atividade.

Acesso aos Componentes da Aplicação Durante a execução do processo, quando um dos servidores nó da grade necessita de obter a especificação de um processo ou controlador em XSPL ou uma atividade compilada, ou classes auxiliares, para executar esse elemento, ele se comunica com o gerenciador de grupo da aplicação para atualizar as cópias dos arquivos da aplicação no seu diretório local correspondente ao domínio da aplicação, chamado de *cache* do domínio. Com esses arquivos em XSPL e bibliotecas de classe no *cache* de domínio, é possível instanciar e executar os elementos e criar e se referir a objetos específicos da aplicação.

5.3.4 Escalonamento e Execução de Processos

Os processos têm os seus elementos escalonados e executados paralelamente e recursivamente. Em cada elemento da árvore hierárquica que define o processo, os seus elementos filhos são escalonados em paralelo e recursivamente e depois executados recursivamente, em forma de um *pipeline* de processamento de elementos, por elemento pai, compreendido de várias etapas: definição de prioridades de execução dos elementos filhos; escolha dos

elementos filhos a alocar; seleção e alocação de servidores nó para executar cada elemento filho, seja ele controlador, atividade ou processo; envio dos parâmetros iniciais; atualização dos arquivos de definição da aplicação; seleção do elemento para execução pelo gerenciador de grupo do servidor nó; execução; envio de resultados ou exceção; continuação da execução, se necessário pelo tratamento da falha ou erro; e remoção da alocação.

Invocação de Processos Um processo é invocado por um usuário por meio da utilização do gerenciador de cliente (CM), que requisita ao gerenciador de grupo a alocação de nó para a execução do processo. Para invocar um processo, o usuário deve se referir ao grupo e domínio da aplicação e ao nome do tipo do processo a ser instanciado. Depois de alocar o servidor nó para executar o processo, o gerenciador de grupo retorna uma referência ao gerenciador de nó associado. Com a referência, o gerenciador de cliente passa os valores dos parâmetros iniciais a serem utilizados por essa instância do processo, por meio de uma mensagem, ao gerenciador de nó alocado. No final da execução do processo, o gerenciador de nó que executou o processo retorna uma mensagem com os valores dos parâmetros de saída do processo ou uma exceção. Finalmente, o gerenciador de cliente requisita ao gerenciador de grupo a remoção da alocação do processo.

O gerenciador de cliente monitora o estado de execução do processo e do gerenciador de nó alocado. Se houver uma realocação, por falha no servidor nó inicial ou balanceamento de carga, o gerenciador de grupo repassa a referência ao novo gerenciador de nó alocado, para o gerenciador de cliente, e o gerenciador de cliente reenvia a mensagem com os valores de parâmetros de entrada para o novo gerenciador de nó. O gerenciador de cliente armazena os parâmetros de entrada e os resultados dos processos no repositório compartilhado para se recuperar de falhas na máquina cliente.

Etapas Prévias para a Execução de Processos No lado do gerenciador de nó, existe um protocolo que é seguido na execução de processos. Primeiro, como foi explicado acima, o processo deve ser alocado no servidor nó, se houver disponibilidade, por meio do uso de gerenciadores de grupo. Depois de alocado, o arquivo em XSPL da aplicação, que contém o processo, é atualizado no servidor nó, assincronamente à execução do processo, de forma a evitar o aumento do tempo para a execução do elemento e diluir o custo de transferência de dados. Simultaneamente, o gerenciador de nó pode receber a mensagem com os valores dos parâmetros de entrada e armazená-la no repositório compartilhado. Uma vez carregado o arquivo do processo, e os auxiliares, e recebidos os parâmetros iniciais, o processo está pronto para ser executado. A partir desse momento, o gerenciador de grupo é notificado do estado *pronto* do processo. Ele invoca a execução do processo no tempo previsto de início do processo na agenda de processos mantida para esse servidor nó no próprio gerenciador de grupo. A execução do processo será descrita em detalhes na

Seção 6.1, sendo dividida em fases de escalonamento, alocação e invocação dos elementos do processo, de forma paralela, em um *pipeline* determinado pela dependência entre esses elementos. O resultado do processo, obtido no final de sua execução, é retornado em forma de uma mensagem ao gerenciador de cliente, seja por notificação ou por requisição. Todos os passos dessa execução são armazenados em meio persistente para posterior recuperação em caso de falhas do servidor de processos, como será visto na Seção 6.3, de tolerância a falhas.

Escalonamento de Elementos de Processos No escalonamento dos elementos de um processo, são atribuídas prioridades para os elementos, eles são escolhidos de acordo com a ordem de execução, e servidores nó são selecionados e alocados para cada elemento. A alocação dos servidores nó para executar cada elemento é realizada pelo gerenciador de grupo. Mais detalhadamente, o processo é visto de forma hierárquica, com o próprio processo principal como raiz, controladores nos nós intermediários, e invocações a atividades e processos nas folhas. Nessa hierarquia, cada elemento escalona os seus elementos filhos imediatos, fornecendo prioridades e requisitando a alocação de servidores nó para executá-los. O escalonamento interno a um controlador ou processo pode se iniciar assim que este for alocado. Na alocação é considerada a prioridade absoluta, da aplicação, e a relativa, do elemento em relação a outros elementos da aplicação: elementos de maior prioridade são alocados para executar antes de elementos de menor prioridade na mesma agenda de elementos, mesmo que a sua alocação seja requisitada posteriormente. Por isso o escalonamento de vários elementos pode ser executado em paralelo, tanto internamente, quanto entre elementos de níveis distintos. Ademais, pode-se escalonar alguns elementos filhos de um controlador ou processo, até um número limite, e depois escalonar o restante por requisição da execução do elemento. Para cada elemento escolhido, é selecionado o melhor recurso para executá-lo, de acordo com uma heurística, e este é alocado.

Execução de Elementos de Processos Depois do escalonamento de um elemento de processo, ele pode ser executado. O *middleware* escalona e executa os elementos de um processo, ou controlador, em forma de *pipeline*: ao mesmo tempo que elementos são escalonados, os já escalonados podem ser executados. A execução é realizada pela invocação dos elementos do processo, ou controlador, por meio da passagem da mensagem inicial correspondente para cada elemento. Essas invocações respeitam a ordem seqüencial, se forem elementos filhos imediatos de um processo, ou respeitam a ordem imposta pelos controladores, de acordo com o seu tipo. O tipo do controlador determina quais elementos deve ser invocados após o término do anterior, definindo, assim, o fluxo de controle e de dados entre os seus elementos filhos imediatos. Para cada invocação ou recebimento de resultados (ou exceção) dos elementos filhos invocados, são armazenados no repositó-

rio compartilhado o valor das variáveis internas modificadas e os valores dos parâmetros iniciais enviados ou os resultados obtidos, para posterior recuperação em caso de falha do servidor nó executor do elemento.

5.3.5 Execução de Atividades

Clientes e servidores nó executando processos podem invocar atividades, sendo denominados invocadores. Para a execução de uma atividade: primeiro é selecionado e alocado o melhor servidor nó para executá-la, segundo exigências da requisição do invocador; a biblioteca que contém as classes da aplicação é atualizada no servidor nó, se necessário; os parâmetros iniciais são enviados pelo invocador; o gerenciador de grupo notifica o gerenciador nó para executá-la, no tempo previsto pela sua agenda de atividades; a atividade é executada; os resultados ou exceção são retornados ao invocador; sua execução é reestabelecida, se necessário, no caso de exceção e se for recuperável; sua alocação é removida dos gerenciadores de grupo envolvidos e do servidor nó após o seu término.

Invocação de Atividades Uma atividade por ser invocada por um usuário utilizando-se o gerenciador de cliente Xavantes, que requisita ao gerenciador de grupo a alocação desta. Para invocar uma atividade, o usuário deve se referir ao grupo e domínio da aplicação e ao nome do tipo da atividade a ser instanciada. Depois de alocar o servidor nó para executar a atividade, o gerenciador de grupo retorna uma referência a seu gerenciador de nó. Com a referência, o cliente passa os valores dos parâmetros iniciais a serem utilizados por essa instância da atividade, por meio de uma mensagem, ao gerenciador de nó. No final da execução da atividade, o servidor nó retorna uma mensagem com os valores dos parâmetros de saída ou uma exceção. Finalmente, o cliente requisita ao gerenciador de grupo a remoção da alocação da atividade.

O gerenciador de cliente monitora o estado de execução da atividade. Se houver uma realocação, por falha no servidor nó inicial ou balanceamento de carga, o gerenciador de grupo local notifica ao gerenciador de cliente esse fato, que reenvia a mensagem com os valores de parâmetros de entrada para o novo servidor nó alocado. Ele armazena os parâmetros de entrada e os resultados das atividades no repositório compartilhado para se recuperar de falhas na máquina cliente.

Execução de Atividades No lado do servidor de atividades, existe também um protocolo que é seguido pelo gerenciador de nó na execução de atividades. Primeiro, como foi explicado acima, a atividade deve ser alocada na agenda de atividades para o servidor nó, localizada no gerenciador de grupo do servidor nó, se houver disponibilidade. Depois de alocada, as bibliotecas da aplicação que contêm a atividade são copiadas do geren-

ciador de grupo associado à aplicação para o servidor, paralelamente, de forma a evitar o aumento do tempo para a execução do elemento e diluir o custo de transferência de dados. Simultaneamente, o gerenciador de atividades pode receber a mensagem com os parâmetros iniciais advinda do invocador e armazená-la no repositório persistente. Uma vez carregadas as bibliotecas e arquivos da aplicação e recebidos os parâmetros iniciais, a atividade está pronta para ser executada. O gerenciador de grupo inicia sua execução no tempo previsto de início na agenda de atividades para o servidor nó. A atividade é executada por meio da invocação do método *run* da classe Java que a define, contida na biblioteca carregada. O resultado da atividade, computado no final da execução, é retornado em forma de uma mensagem ao cliente, seja por notificação ou por requisição. Todos os passos dessa execução são armazenados em meio persistente para posterior recuperação em caso de falhas do servidor de atividades, como será visto na Seção 6.3.

5.4 Resumo

O *middleware* Xavantes é responsável por monitorar e executar processos estruturados especificados em XSPL (*Xavantes Structured Process Language*) na grade computacional. Ele utiliza a árvore sintática do processo, formada pelo próprio processo na raiz, controladores nos ramos e invocação a atividades e a processos nas folhas, para executar *hierarquicamente* o processo, escalonando, distribuindo e invocando os seus elementos recursivamente. O processo é alocado a uma máquina, que escalona, distribui e invoca os seus elementos filhos para serem executados na mesma ou em outra máquina. Se o elemento filho é um controlador, ele escalona, distribui e invoca também seus elementos filhos para serem executados na mesma ou em outra máquina, e assim recursivamente, até que todos os elementos filhos sejam atividades, escalonadas e distribuídas para execução.

O *middleware* executa sobre o sistema operacional de máquinas de uma ou mais organizações, redes locais autônomas ou aglomerados (*clusters*), denominados grupos. Cada grupo é formado por máquinas, bancos de dados relacionais compartilhados e uma rede relativamente rápida que os interliga. Para realizar o monitoramento e a execução hierárquica de processos na grade computacional, o *middleware* conta com três serviços principais: um serviço gerenciador de grupo por grupo, um serviço gerenciador de nó por máquina, e um gerenciador de cliente por usuário ativo, executado na máquina do usuário.

O serviço gerenciador de grupo (GM) controla as informações e os serviços coletivos. Ele mantém as informações de características e disponibilidade dos servidores nó do grupo e agendas de atividades e de processos a serem executados em cada um deles, com prioridade, tempo de início, fim e poder de processamento reservado (no caso de atividade); tabelas com informações sobre os elemento de processos (processos, controladores e atividades) que foram invocados pelo grupo, ou em execução no grupo mas invocados por um

grupo cliente; registros dos gerenciadores de grupo pares, servidores ou clientes; além de arquivos que compõem uma aplicação. Ele responde por seleção de melhores servidores nós; aloca servidores nós para executar elementos de processo; realoca, se conveniente ou necessário; requisita a execução de cada elemento aos servidores nó; informa o estado de elementos de processo em execução; e provê acesso aos arquivos das aplicações.

O serviço gerenciador de nó (NM) executa processos, controladores e atividades requisitadas pelo grupo. Ele mantém uma tabela de elementos alocados, com características, estado de ativação dos elementos e parâmetros iniciais e resultados da execução; arquivos de cada aplicação alocada em *cache*; e a árvore sintática que representa os controladores e os processos em memória. Ele atualiza ou cria o *cache* da aplicação por demanda; executa os elementos de processo (processos, controladores e atividades); informa sobre a execução dos elementos; recebe parâmetros iniciais e retorna resultados ou exceções; cria instâncias de processos e controladores em memória; escalona, distribui (requisita alocação), e invoca elementos filhos em outros servidores nó; reescalona, para aumentar o desempenho de execução; e persiste e recupera o estado de execução dos elementos de processo em caso de falhas ou migração.

O serviço gerenciador de cliente (CM) invoca e monitora processos e atividades na grade computacional. Ele mantém uma tabela de elementos invocados; e parâmetros iniciais e resultados de execução. Tem como função especificar processos; implantar processos e atividades no grupo; invocar e monitorar a execução de processos; enviar parâmetros iniciais; receber resultados de execução; e requisitar a desalocação de processos e atividades terminadas.

As vantagens da execução hierárquica de processos estruturados com o suporte do *middleware* Xavantes são:

interoperabilidade — todas as bibliotecas de classes, as atividades, os processos e o próprio sistema rodam sobre a máquina virtual Java (JVM), que pode ser instalada em mais de 99% das máquinas interligadas à Internet. Portanto, a grande maioria das máquinas, independentemente da arquitetura ou sistema operacional, pode instalar o JVM, depois esse *middleware* e começar a participar da grade.

transparência de acesso — as atividades e os processos especificados são distribuídos e executados transparentemente na maior parte das arquiteturas de máquinas, sem o programador necessitar de se preocupar em codificar funções específicas para acessar determinado tipo de recurso.

execução distribuída hierárquica — a execução hierárquica permite distribuir controladores a vários nós da grade, executando-se processos de forma totalmente distribuída, mas sem perder o controle dos elementos filhos, o que facilita o escalonamento adaptativo dos elementos filhos e a recuperação da execução em caso de

falhas, além de possibilitar a comunicação localizada entre o controlador e os seus elementos filhos.

desempenho — o *middleware* confere um alto desempenho para a execução de aplicação pois: as atividades são compiladas para *bytecode* Java; o processo escalona as atividades de forma a alocá-las nos recursos mais poderosos de acordo com o seu peso para o término mais cedo deste; aloca o controlador próximo à atividade, quando possível, para diminuir o tempo de comunicação; e o escalonamento se adapta agilmente à variabilidade da disponibilidade de recursos, inerente à grade, para aproveitar melhor os recursos.

escalabilidade — a escalabilidade se classifica em dois tipos: das aplicações e da grade. A escalabilidade das aplicações é atingida pela possível distribuição completa de seus controladores, o que permite a execução das partes do processo em vários grupos de máquinas, se não houver capacidade suficiente, local ou próxima. A escalabilidade da grade suportada é obtida pela adição de novas máquinas nos grupos existentes ou de novos grupos na grade, com as interligações necessárias, o que aumenta a capacidade da grade imediatamente.

adaptabilidade — a execução distribuída de elementos de processo é adaptável em dois pontos: alocação e escalonamento. A alocação é controlada pelo gerenciador de grupo, que pode automaticamente realocar elementos para outros servidores nó de acordo com as prioridades e requisitos dos elementos, para aproveitar novos recursos ou espaços vazios na agenda por término mais cedo de elementos. O escalonamento é controlado pelo gerenciador de nó que está executando o controlador ou o processo pai. Ele pode ser adaptável no sentido de monitorar os recursos e requisitar realocações ou modificar as prioridades do elementos conforme seu peso aumente para o término mais cedo do elemento pai ou processo.

confiabilidade — o sistema é considerado confiável pois todos os seus serviços podem ser restaurados após uma falha; as aplicações são confiáveis pois as atividades recuperáveis, os processos e os controladores são suspensos e depois restaurados em caso de falhas de sistema; as atividades regulares são abortadas e reexecutadas; as falhas de aplicação geram exceções que podem ser tratadas na especificação ou pelo usuário, com posterior continuidade da execução, aborto ou compensação. Os dados compartilhados podem ser acessados por transações, se o seu acesso não for controlado explicitamente, para serem restaurados de forma consistente.

disponibilidade — o sistema e as aplicações são disponíveis pois suas execuções são reiniciadas rapidamente quando a máquina falha é restaurada. No caso de atividades

e processos, esses podem ser migrados no caso de desativação manual ou por tempo da máquina atual, desde que as duas máquinas compartilhem o mesmo repositório.

otimização do uso de recursos — por meio do uso de controladores é possível realizar a alocação de recursos iterativamente, conforme eles forem sendo necessários pelos elementos filhos a serem executados. Isso permite uma redução nas agendas de elementos dos gerenciadores grupos e das alocações de recursos dos servidores nó. Entretanto, esse medida não prejudica o algoritmo de escalonamento, pois o mecanismo de prioridades relativa garante a alocação do espaço adequado, mesmo que a alocação seja tardia.

localidade de comunicação — o uso de objetos compartilhados nos controladores permite que atividades filhas utilizem um objeto próximo para se comunicarem, reduzindo o custo da comunicação entre elas e otimizando o desempenho das aplicações.

Capítulo 6

Execução de Processos e Tolerância a Falhas

Nesse capítulo descreveremos como o sistema Xavantes executa processos, escalonando e distribuindo adaptativamente e recursivamente controladores e atividades, e como ele tolera falhas em nível de atividades e processos, conferindo, com isso, confiabilidade, disponibilidade, escalabilidade e alto desempenho para aplicações em grades computacionais.

6.1 Execução de Processos

O sistema Xavantes executa aplicações definidas por processos estruturados na grade computacional distribuindo e executando controladores e atividades hierarquicamente pelos servidores nó da grade. Cada controlador de um processo, seja seqüencial ou paralelo, tem uma semântica própria que permite facilmente definir processos complexos por meio da composição hierárquica desses, de forma semelhante à estruturação de construtores de controle nas linguagens estruturadas genéricas, mas que somente permitem fluxos seqüenciais. O *middleware* Xavantes utiliza a estrutura do processo para distribuir hierarquicamente os controladores pelos servidores nó da grade. Os controladores, uma vez alocados a cada servidor nó, são instanciados e iniciam a execução de seus elementos filhos recursivamente, escalonando, alocando e coordenando a execução deles. Como o controlador tem todo o controle sobre os elementos filhos, pode-se promover uma coordenação, execução adaptativa e recuperação de falhas local. Com isso, a execução do processo é realizada de forma eficiente, escalável, confiável e adaptável.

Os usuários da grade utilizam gerenciadores de cliente (*front-ends*) para invocar, monitorar e controlar a execução de processos. O usuário deve identificar o tipo do processo e passar os parâmetros iniciais ao gerenciador de cliente para criar uma instância do processo. A Figura 6.1 apresenta as interações entre os serviços Xavantes para a execução

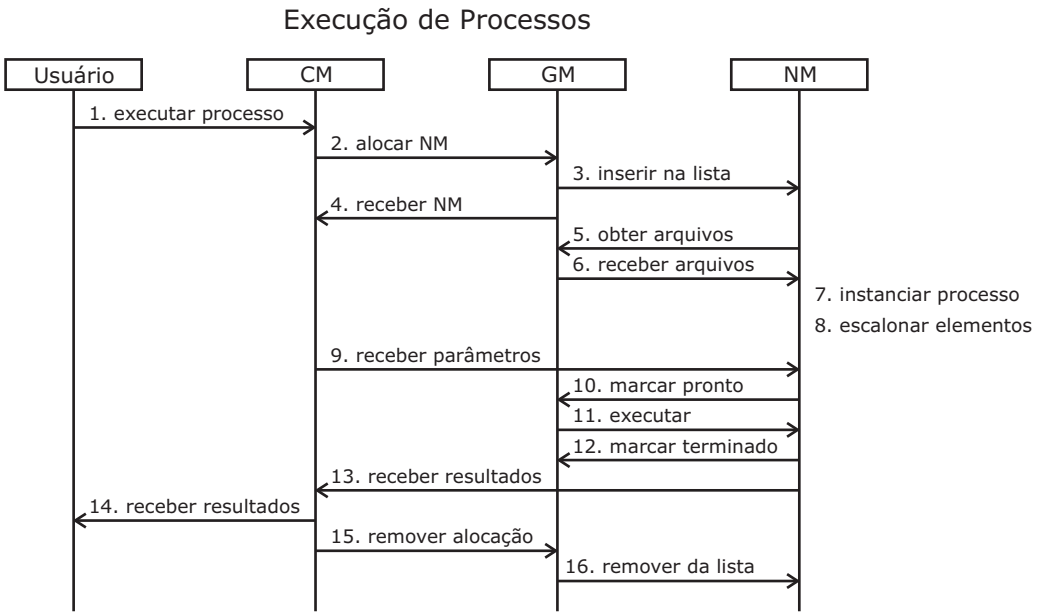

Figura 6.1: Execução de um Processo no *Middleware* Xavantes.

do processo criado pelo usuário:

1. **executar processo** — o usuário invoca o processo passando o seu tipo e parâmetros iniciais para o gerenciador de cliente (CM);

2. **alocar NM** — o gerenciador de cliente (CM) solicita ao gerenciador de grupo (GM) a alocação de um gerenciador de nó (NM) para executar o processo;

3. **inserir na lista** o gerenciador de grupo aloca o melhor gerenciador nó para o processo, utilizando as agendas de processo, e solicita ao gerenciador nó para inserir o processo na sua lista de execução;

4. **receber NM** — o gerenciador de grupo retorna as informações de alocação do processo com a referência ao gerenciador de nó;

5. **obter arquivos** — o gerenciador de nó solicita ao gerenciador de grupo os arquivos da aplicação atualizados ou criados a partir de uma data;

6. **receber arquivos** — o gerenciador de grupo retorna os arquivos da aplicação atualizados;

7. **instanciar processo** — o gerenciador de nó instancia o processo conforme a sua definição constante nos arquivos obtidos;

8. **escalonar elementos** — o gerenciador de nó inicia o escalonamento dos elementos filhos do processo, que solicita a alocação de outros gerenciadores de nó para executar esses elementos filhos, iniciando o passo 2 para cada um desses elementos, e assim recursivamente se o elemento filho for um controlador ou um processo;

9. **receber parâmetros** — o gerenciador de cliente repassa os parâmetros iniciais para o gerenciador de nó alocado executar o processo;

10. **marcar pronto** — o gerenciador de nó informa ao gerenciador de grupo que o processo está pronto;

11. **executar** — o gerenciador de grupo solicita ao gerenciador de nó alocado para executar o processo, no momento alocado de início de execução. Este último coordena a execução do processo, invocando e repassando os parâmetros iniciais para cada elemento filho na ordem definida no processo, iniciando o passo 9 para cada um dos elementos, e assim recursivamente se o elemento for um controlador ou processo;

12. **marcar terminado** — o gerenciador de nó informa ao gerenciador de grupo que o processo terminou;

13. **receber resultados (gerenciador de nó)** — o gerenciador de nó retorna os resultados ao gerenciador de cliente no final da execução do processo;

14. **receber resultados (gerenciador de cliente)** — o gerenciador de cliente informa ao usuário os resultados obtidos;

15. **remover alocação** — o gerenciador de cliente requisita a remoção da alocação do elemento ao gerenciador de grupo;

16. **remover da lista** — o gerenciador de grupo solicita ao gerenciador de nó para remover o elemento da sua lista.

No escalonamento e na coordenação de elementos filhos de um processo ou controlador, o gerenciador de nó (NM) requisita a alocação de servidor nó para executar esses elementos filhos (passo 2), recebe informações de alocação (passo 4), envia parâmetros iniciais para a execução dos elementos filhos de acordo com o fluxo de controle (passo 9), recebe resultados dos elementos filhos (passo 13) e remove a alocação desses elementos (passo 15). Assim ele realiza o mesmo papel de invocador do gerenciador de cliente para processos, mas para cada um dos elementos filhos.

Na invocação de uma controlador são executados os mesmos passos que na invocação de um processo, mas o invocador é um gerenciador de nó. Na invocação de uma atividade, não existe o escalonamento do elemento (passo 8), pois atividades são atômicas do ponto

de vista do sistema e não são escalonadas. O invocador nesse caso pode ser um gerenciador de cliente ou um gerenciador de nó.

Nas próximas seções são detalhadas as principais fases da especificação e da execução do processo.

6.1.1 Especificação e Implantação de Processos e Atividades

Uma aplicação Xavantes é construída por meio de especificações e de invocações a processos, controladores, atividades e bibliotecas de classes Java. Os processos podem ser especificados, de forma flexível e expressiva, utilizando-se a linguagem XSPL (*Xavantes Structured Process Language*), que contém controladores para determinar todo tipo de fluxo de controle estruturado seqüencial ou paralelo e que permite introduzir *scripts* em Java para realizar controle fino da execução e da transformação de dados. As atividades podem ser especificadas em XSPL ou podem ser implementadas diretamente como uma classe Java, derivada de *Activity* do arcabouço Xavantes (*Xavantes Grid Library*). Pode-se, no futuro, criar uma interface gráfica para o usuário do sistema Xavantes especificar e monitorar processos em XSPL, o que facilitará muito essas duas tarefas.

Os processos especificados no sistema Xavantes contêm controladores que determinam, segundo a sua semântica, a ordem de invocação a elementos filhos (outros controladores, processos e atividades). O fluxo de dados direto é definido pela passagem de objetos, por valor ou por referência, pelos parâmetros de entrada e de saída dos elementos de processo. Dados podem ser trocados entre elementos filhos de um controlador por meio do uso de objetos compartilhadas, que contêm operações de leitura e alteração de dados, possivelmente com controle de acesso e transações, instanciados no contexto desse controlador.

As especificações de processos e atividades da aplicação, juntamente com as possíveis bibliotecas utilizadas, são enviadas para o gerenciador de grupo local, que implanta os arquivos correspondentes em um domínio de nomes da aplicação, persistindo-os em um diretório específico do sistema de arquivos local. Todos os processos e atividades da aplicação podem ser especificados graficamente ou por meio de editor de texto, gerando arquivos XML correspondentes. Todas as atividades da aplicação construídas por meio de documentos XML são compiladas para classes Java por meio de ferramentas específicas. Essa compilação resulta nas mesmas classes que o especificador pode implementar diretamente utilizando o arcabouço Xavantes. Posteriormente, as classes Java que representam as atividades são compiladas para *bytecode* Java, para tornar a execução destas tão eficiente quanto qualquer código escrito em C. A partir desse ponto, os processos, controladores e atividades podem ser instanciados e executados por gerenciadores de nó nos servidores nó da grade.

6.1.2 Invocação e Instanciação de Processos e Controladores

Para invocar e monitorar um processo raiz, o gerenciador de cliente (*front-end*) do Xavantes é utilizado. O usuário requisita a execução de um processo ao gerenciador de cliente passando o tipo do processo a ser criado e os parâmetros iniciais a serem utilizados. O tipo do processo é identificado pelo nome do grupo onde foi implantada a aplicação, nome da aplicação que contém o processo e nome do tipo do processo, definido na especificação do processo. O gerenciador de cliente se registra e se autentica no gerenciador de grupo local, como requisito de acesso aos recursos da grade computacional. Após essa fase, o gerenciador de cliente pede ao gerenciador de grupo para selecionar e alocar um gerenciador de nó, local ou externo, para executar o processo, passando os requisitos desse processo. Depois, o gerenciador de cliente repassa os parâmetros iniciais ao gerenciador de nó que contém o elemento alocado.

Um processo ou controlador pode se invocado por um gerenciador de nó executando o elemento pai que contém referência a esse filho. O gerenciador de nó também identifica o tipo do processo ou do controlador a ser executado. Todo controlador é definido dentro de um processo, e têm o nome do tipo igual ao do tipo do processo mais o caminho até a sua definição, separado por pontos. Antes de invocar, o gerenciador de nó requisita ao gerenciador de grupo para selecionar e alocar um gerenciador de nó, local ou externo, para executar o processo (ou controlador), passando os seus requisitos. Depois, o gerenciador de nó invocador repassa os parâmetros iniciais ao gerenciador de nó que contém o elemento alocado.

Quando um gerenciador de nó é alocado para executar um processo ou controlador, ele atualiza os arquivos necessários da aplicação, instancia o elemento e espera os parâmetros iniciais e a ordem de execução do elemento oriunda do gerenciador de grupo local. O gerenciador de nó atualiza no diretório local, se necessário, os arquivos XML de especificação de processos e de atividades e as bibliotecas de classes da aplicação, lendo-os por meio de um canal de comunicação com o gerenciador de grupo correspondente.

O processo ou o controlador é instanciado em memória segundo a estrutura hierárquica do arquivo XSPL com a definição do processo. O processo (ou controlador) é representado em memória como uma árvore de execução, onde cada nó é um objeto que corresponde a um marcador do arquivo, instanciado da classe correspondente ao tipo do marcador, e com valor para cada campo correspondente ao valor do atributo de mesmo nome no marcador. Assim, a raiz da árvore é um objeto do tipo *Process*, correspondente ao marcador *ProcessType*, que possui um objeto filho do tipo *Body*, entre outros; e o valor do atributo *typeName* do objeto raiz é igual ao nome do tipo do processo, definido por *name*. Os objetos nós da árvore são inseridos na mesma ordem dos marcadores, para um dado marcador pai. Entretanto, alguns nós são suprimidos por otimização para a execução e outros são transformados, tal como a tabela de parâmetros de entrada, utilizada para

acessar rapidamente os valores dos parâmetros de um processo.

A construção da árvore de execução ocorre da raiz para baixo e pára nas folhas, continuando em outro ramo até ser finalizada. Exemplos de folhas são códigos, controladores, e invocações a atividades e a processos. O controlador tem um tratamento específico: ele é considerado folha, sendo transformado em uma invocação ao controlador, com os mesmos nomes de parâmetros e valores. Isso ocorre pois o controlador pode ser distribuído para ser executado em outra máquina, tal como atividades e processos invocados.

Além dos campos correspondentes aos atributos dos tipos dos marcadores, que armazena uma informação do tipo (especificação) do elemento, os objetos contêm campos para armazenar o estado do dados da instância do elemento e informações de controle, tal como o ponto de execução. Por exemplom existe o campo *typeName* em *Process* que representa uma informação de tipo (o nome do tipo do processo, no caso) e existe o campo *name* que representa uma informação de instância (o nome da instância do processo, no caso).

6.1.3 Invocação e Instanciação de Atividades

Uma atividade pode ser invocada pelo cliente ou gerenciador de nó executando o elemento pai que contém referência a essa atividade. O invocador identifica o tipo da atividade a ser executada. O tipo da atividade é identificado pelo nome do grupo onde foi implantada a atividade, nome da aplicação que contém a atividade e nome do tipo da atividade. O invocador solicita ao gerenciador de grupo para selecionar e alocar um gerenciador de nó, local ou externo, para executar a atividade, passando os requisitos dessa atividade. Com a referência do gerenciador nó, o invocador repassa os parâmetros iniciais para a atividade alocada.

Quando um gerenciador de nó é alocado para executar uma atividade, ele atualiza os arquivos necessários da aplicação, instancia a atividade e espera os parâmetros iniciais e a ordem de execução da atividade oriunda do gerenciador de grupo local. O gerenciador de nó atualiza no diretório local, se necessário, a biblioteca de classes da aplicação que contém a classe compilada que representa a atividade a ser executada, obtendo-a por meio de um canal de comunicação com o gerenciador de grupo correspondente. A atividade é instanciada criando-se um objeto Java, no executor de atividades, a partir da classe que a representa na biblioteca.

6.1.4 Escalonamento de Elementos Filhos em Processos e Controladores

Depois da instanciação, o processo ou controlador pode começar a escalonar os elementos filhos. Para escalonar esses elementos filhos, o executor de processos invoca o módulo

de escalonamento repassando a árvore de escalonamento do processo. Essa é uma árvore de objetos, como a árvore de execução, mas somente contém objetos importantes para o escalonamento e sua construção não pára nos controladores, chegando até a folhas da especificação do processo. Desta forma a árvore de escalonamento permite uma visão mais específica e global do processo para o escalonamento.

O módulo de escalonamento pode aceitar diferentes algoritmos de escalonamento com seleção de elementos de maior prioridade primeiro. Alguns desses algoritmos serão descritos na Seção 6.2. O módulo de escalonamento repassa a árvore para o algoritmo ativo de escalonamento e este inicia o seu trabalho. Ele pode utilizar várias funcionalidades do sistema para realizar o escalonamento dos elementos filhos, tal como consulta à disponibilidade dos recursos, e pode requerer que o gerenciador de grupo selecione e aloque o melhor recurso da grade para um elemento filho com certas características e requisitos.

De modo geral, cada algoritmo de escalonamento realiza os seguintes passos:

1. atribui prioridades de execução aos elementos filhos, de acordo com a ordem de execução estabelecida e um critério próprio de peso de cada elemento;

2. inicia a seleção e alocação de recursos para executar os primeiros elementos filhos do processo, por ordem de execução, levando em conta suas características: prioridades atribuídas, tempo mínimo de início, tempo máximo de fim, estimativa de esforço de processamento, no caso de atividades, e tamanho total das mensagens trocadas entre o elemento filho e o próprio elemento;

3. realizada a alocação de recursos para esses elementos filhos, o escalonador obtém os tempos estimados de início e término desses de acordo com o recurso alocado e calcula o tempo mínimo de início dos próximos elementos filhos do processo, de acordo com a ordem de execução;

4. os elementos filhos seguintes são alocados e assim sucessivamente até que todos os elementos filhos tenham sido escalonados.

O escalonamento pode ser centralizado ou distribuído. No primeiro caso, todo o escalonamento e alocação de elementos filhos são realizados no nível do processo, centralizadamente, na mesma máquina que executa o processo raiz. No segundo caso, o escalonamento e a alocação são realizados no nível do elemento, somente considerando os elementos filhos imediatos. Nesse caso, se o elemento filho for um controlador ou um processo, após a sua alocação, o escalonamento e a alocação dos elementos filhos deste também são iniciados no servidor nó alocado. O escalonamento então é realizado recursiva, distribuída e hierarquicamente até que só existam atividades a serem escalonadas nos elementos.

6.1.5 Coordenação de Processos e Controladores

A coordenação de um processo ou controlador corresponde à execução do elemento com a invocação de processos, controladores e atividades filhos. A coordenação ocorre depois da recepção dos parâmetros iniciais pelo gerenciador de nó e da notificação do gerenciador de grupo para executá-lo. A coordenação, realizada pelo executor de processos, é local, ocorrendo somente no nível do elemento. A coordenação utiliza a árvore de execução do elemento, construída na instanciação do processo ou do controlador.

O escalonamento e a coordenação de elementos filhos podem ocorrer concorrentemente, em forma de *pipeline*, onde cada elemento filho deve ser escalonado, invocado e executado. Assim um elemento filho inicial pode estar executando enquanto outro, posterior, sendo escalonado.

A coordenação utiliza a árvore de execução para saber qual é o tipo do controlador a ser executado, obter a sua estrutura interna, e ler e modificar valores dos objetos relacionados à instância do controlador. Ele executa o controlador seguindo a semântica definida, executando os *scripts* Java para devolver resultados de expressões e alterar dados, invocando os elementos filhos e esperando o seu término, e modificando variáveis de controle conforme o andamento da execução. Na invocação do elemento filho são enviados os parâmetros iniciais, e no término deste, quando ocorre a junção, são recebidos os resultados e alterados os valores das variáveis internas correspondentes. O executor também persiste o estado de execução do controlador em passos definidos, para posterior restauração da execução ou migração, se necessário, por tolerância a falhas.

Em controladores paralelos, podem ser criadas várias linhas de execução (*threads*) para executar os caminhos definidos. Por exemplo, para executar um *parfor*, é criada uma *thread* em cada iteração para executar o único caminho definido, mas com variáveis de dados e de controle distintas para cada *thread*, quando não compartilhadas, e uma outra *thread* de sincronização associada ao código do marcador *join*. No caso de controlador paralelos iterativos, e.g. *parfor*, os caminhos de execução (*threads*) são criados conforme necessário e dependendo dos recursos disponíveis.

6.2 Escalonamento

O problema da distribuição de tarefas de um processo do tipo *workflow* aos recursos de um ambiente multiprocessado e heterogêneo de forma a se obter um escalonamento com o menor tempo possível de execução do processo é um problema NP-completo [ERAL95]. Dessa forma, foram criadas diversas heurísticas para tentar resolver esse problema. Elas podem ser classificadas basicamente em heurísticas que utilizam: seleção de tarefas com maior peso, por exemplo em esforço computacional exigido, a serem alocadas aos recursos

mais eficientes; agregação de tarefas relacionadas para redução do custo de comunicação; e replicação de tarefas, para aproveitamento de recursos mais disponíveis não previstos em tempo de escalonamento. É possível perceber que se nas heurísticas com seleção de tarefas for considerado o custo da comunicação na seleção do elementos, elas fornecem resultados similares ou melhores que as heurísticas com agregação, por considerarem adicionalmente as informações de esforço de processamento das tarefas. Por esse motivo, o Xavantes suporta algoritmos de escalonamento com seleção de tarefas com maior peso. Esses algoritmos podem utilizar os serviços do *middleware* Xavantes para obter informações sobre os elementos (processos, controladores e atividades) da estrutura de um processo; obter informações agregadas e detalhadas sobre os recursos de cada grupo conhecido; e requisitar a seleção e alocação dos melhores recursos para executar elementos de processo com características próprias e prioridades fornecidas. No futuro, pensa-se em suportar também a replicação de tarefas para reduzir o tempo de execução do processo, caso haja alguma falha num recurso alocado ou caso ele se comporte de forma inesperada.

O escalonamento, quanto à abrangência, pode ser total ou parcial. O escalonamento total considera todo o processo antes de sua execução ser iniciada. O parcial, entretanto, pode escalonar e executar partes do processo de forma iterativa e concorrente, em turnos, conforme for sendo necessário, permitindo que os elementos do processo iniciem sua execução antes do término do escalonamento de todo o processo. No limite, um elemento de processo pode ser escalonado e ter a sua execução iniciada imediatamente. A vantagem da abordagem parcial é a execução imediata do processo, sem a necessidade de esperar o seu escalonamento total, que pode ser relativamente demorado.

O escalonamento, quanto à seleção dos elementos a serem escalonados, pode ser classificado em local ou global. Algoritmos globais levam em consideração todo o processo, do nível dos controladores até o nível das atividades, para decidir qual será o próximo elemento a ser escalonado. Os locais, pelo contrário, observam somente o nível atual para escalonar o elemento, como por exemplo os controladores e atividades que estão abaixo do controlador considerado. Os algoritmos locais geralmente são mais eficientes, rápidos, mas menos eficazes do que os globais, pois retornam um escalonamento em que provavelmente o processo demorará mais para terminar, ao desconsiderar o peso dos elementos externos em relação aos internos de um nível.

O escalonamento, quanto ao espalhamento de sua execução, pode ser centralizado ou distribuído. O escalonamento centralizado é o mais comum, executado em uma única máquina por processo, permite uma rápida visão global do processo, mas pode ficar distante das tarefas executadas. O distribuído é executado em várias máquinas, próximas às tarefas, resultando em uma execução mais rápida, adaptativa e tolerante a falhas.

O escalonamento, quanto à visão dos recursos dos elementos da grade, pode ser local ou global. Algoritmos locais somente alocam recursos que estão em uma única rede ou

grupo para a tarefa a ser escalonada. Por outro lado, algoritmos globais nesse aspecto alocam recursos de qualquer rede ou grupo da grade. Algoritmos globais são mais eficazes na alocação do melhor recurso, pois consideram os recursos de toda a grade. Entretanto, esses algoritmos exigem a manutenção das informações de disponibilidade dos recursos de toda a grade, ou de um grande número de grupos, e o controle da alocação desses recursos.

O escalonamento, quanto à adaptação da alocação dos elementos ao estado dos recursos computacionais, pode ser estático ou dinâmico. O escalonamento estático é realizado uma única vez para cada elemento do processo, considerando somente o estado atual dos recursos computacionais para realizar a seleção e a alocação de recursos para o elemento corrente. Diferentemente, os algoritmos de escalonamento com adaptação dinâmica monitoram o estado dos recursos computacionais alocados para cada elemento, de acordo com o seu peso, e alocam os elementos para outros recursos com maior capacidade ou disponibilidade, se resultar em um menor tempo de execução.

6.2.1 Algoritmos de Escalonamento

Os algoritmos de escalonamento com seleção de tarefas utilizam diferentes critérios para atribuir pesos (prioridades) às tarefas do processo para posterior alocação dessas às máquinas disponíveis, visando a reduzir o tempo total de execução. Entre os algoritmos baseados em seleção de tarefas podemos citar LMT [IOF95], HEFT [THW02], CPOP [THW02] e PETS [IT07].

O algoritmo LMT (*Levelized Min-Time*) é um dos pioneiros nessa especialidade de escalonamento. Ele subdivide o grafo dirigido (DAG) de dependências entre tarefas em níveis formados pelas dependências. Cada nível é formado por tarefas sucessoras de tarefas de níveis anteriores e sucessora de pelo menos uma tarefa do nível atual. Após a divisão das tarefas em níveis, ele aloca as tarefas de cada nível em processadores disponíveis. Se houver mais tarefas em um nível do que processadores, o algoritmo agrupa tarefas a serem escalonadas até o número de agrupamentos se tornar igual ao número de processadores. O algoritmo HEFT (*Heterogeneous Earliest Finish Time*) atribui um peso a cada tarefa, calculado da tarefa final até a tarefa inicial, como o tempo previsto do caminho crítico desta atividade até o final do processo. Os tempos de processamento e de comunicação entre as tarefas são considerados no cômputo do caminho crítico, sendo que para o cálculo do tempo de processamento considera-se um processador médio da grade, e para o do tempo de comunicação considera-se um canal de capacidade média da grade. O algoritmo CPOP (*Critical Path On Processors*) é semelhante ao HEFT, mas considera o tempo do caminho crítico entre o início do processo e a tarefa corrente, bem como o caminho crítico entre o final do processo e a tarefa corrente para o cálculo do peso da tarefa para a

alocação. Além disso, o CPOP tenta manter as tarefas de um caminho crítico no mesmo processador, a fim de reduzir o tempo de comunicação entre as tarefas. O algoritmo PETS (*Performance Effective Task Scheduling*) utiliza a técnica de classificação de tarefas em níveis (como o LMT), mais o cálculo de um peso para cada tarefa que considera o maior peso entre as tarefas sucessoras, o seu tempo médio de processamento e tempo de comunicação total com as tarefas sucessoras. Posteriormente, as tarefas são alocadas nível a nível, considerando os pesos calculados. Em todos esses algoritmos, as tarefas de maior peso são alocadas primeiro.

Os algoritmos descritos acima são utilizados para escalonar um grafo dirigido acíclico (DAG) de tarefas. Entretanto, a linguagem desenvolvida para o Xavantes produz processos que contém controladores que estruturam hierarquicamente o grafo dirigido de tarefas, e que permitem a criação dinâmica de instâncias de tarefas em seqüência ou em paralelo, por meio dos construtores par e parfor, por exemplo. O grafo resultante, se achatado e se previsto o número de instâncias de atividades a serem criadas durante a execução, pode ser reduzido a um grafo dirigido acíclico (DAG). Portanto, pode se utilizar pequenas variações desses algoritmos para realizar o escalonamento no Xavantes. Entretanto, devido à especificidade dos grafos resultantes dos processos criados no Xavantes, existem algoritmos que podem prover melhores resultados para o escalonamento desses processos, e que aproveitam todos as vantagens do *middleware*, tais como recuperação, escalonamento distribuído e alocação adaptativa que consideram prioridades de execução. Exemplos desses algoritmos são apresentados a seguir.

Escalonamento Global baseado na Alocação Conjunta de Atividades em Caminhos Críticos

Um exemplo de algoritmo que escalona todo o processo utilizando o modelo de programação e serviços fornecidos pelo Xavantes é o algoritmo PCH (*Path Clustering Heuristic*), baseado na alocação conjunta de tarefas pertencentes a cada caminho crítico remanescente em um mesmo processador. Ele pode ser encontrado em Bittencourt [BMCB05]. O escalonamento feito por esse algoritmo é total, global e estático. Testes realizados utilizando processos estruturados com controladores demonstram que esse algoritmo é melhor do que os existentes na obtenção de menor tempo de execução. De forma resumida, ele primeiramente escalona as atividades de todo o processo e depois os controladores, de baixo para cima na estrutura do processo. O escalonamento das atividades é feito em várias iterações, sendo que em cada uma é descoberto o caminho crítico para o grafo de dependências formado pelas atividades não escalonadas, e as atividades desse caminho são agrupadas e alocadas no melhor processador disponível, de forma a diminuir o tempo de processamento e de comunicação. Depois, cada controlador é alocado próximo à atividade (ou controlador filho) com maior tempo previsto de comunicação entre os dois. O

objetivo dessa alocação de controladores é diminuir o tempo da comunicação entre esses e seus elementos filhos. Como conseqüência, tem-se um escalonamento heurístico que tenta reduzir o tempo total de execução do processo, pela redução do tempo de execução dos caminhos mais críticos e da comunicação com os controladores.

Uma extensão desse algoritmo para o escalonamento dinâmico de atividades e controladores foi descrita em [BM06]. O algoritmo proposto baseia-se no conceito de turnos de escalonamento, em que o número de turnos é definido pelo usuário. Em cada turno, parte do processo é escalonada utilizando o algoritmo PCH, as atividades e controladores escalonados são enviados para execução, são recomputados o tempo previsto de início e término das atividade não executadas para o próximo turno, e é novamente utilizado o algoritmo PCH para realizar o escalonamento nesse novo turno. Segundo os testes realizados, o algoritmo dinâmico obtém melhores resultados do que o estático para o caso de variações de desempenho dos recursos, como ocorre na grade, mas sem aumentar a complexidade computacional do algoritmo.

Escalonamento Global baseado no Maior Tempo Remanescente entre os Controladores

Nessa seção descreveremos o algoritmo padrão de escalonamento do Xavantes, que utiliza a estrutura hierárquica de um processo e se baseia no princípio de priorizar a atividade do controlador com maior tempo remanescente de execução para a alocação de recurso, no qual o tempo remanescente de um controlador é a somatória dos tempos de execução previstos de suas atividades não alocadas, recursivamente. Esse algoritmo realiza um escalonamento global, parcial (concorrente com a execução), dinâmico (adaptativo conforme a disponibilidade de recursos) e distribuído.

Um processo no Xavantes contém elementos de processos, que são controladores, invocações a atividades e invocações a processos. Os controladores definem a estrutura de controle e de fluxo de dados entre os seus elementos. As invocações a atividades e a processos definem as tarefas a serem realizadas pelo processo. A especificação de um processo é estruturada e hierárquica, uma vez que cada controlador do processo pode conter outros controladores, invocações a instâncias de processos e de atividades. Existem controladores de diversos tipos, que definem fluxos paralelos ou seqüenciais, condicionais ou fixos, e iterativos ou diretos. Durante a execução, cada controlador gerencia e controla seus elementos filhos diretos, requisitando o escalonamento, a alocação e a execução desses.

O tempo previsto de execução de um controlador é a somatória dos tempos previstos das trocas de mensagens entre ele e seus elementos filhos e os tempos previstos da execução desses elementos filhos, recursivamente. O tempo remanescente previsto de execução de um controlador é a somatória dos tempos previstos das trocas de mensagens e dos tempos remanescentes previsto da execução dos elementos ainda não alocados ou parcialmente

alocados, recursivamente. O tempo remanescente de execução de um controlador é o critério utilizado para definir a sua prioridade na alocação de suas atividades, imediatas ou mediatas, e de seus controladores aninhados.

Tempo Remanescente Previsto de Execução O tempo remanescente previsto de execução de uma atividade não alocada é o produto entre o número previsto de instruções a serem executadas e o tempo da execução de uma instrução em uma máquina mediana estabelecida. O tempo remanescente previsto de comunicação entre quaisquer dois elementos, em que pelos menos um não está alocado, da mesma forma, é o produto entre o tamanho total previsto das mensagens trocadas em *bytes* e o tempo da troca de um *byte* em uma rede mediana estabelecida, desconsiderando a latência para a chegada do primeiro *byte* de cada mensagem.

O tempo remanescente de execução de um controlador ou processo é a somatória dos tempos remanescentes previstos de execução de todas instâncias de seus elementos mais o tempo previsto de comunicação entre esse controlador e as instâncias de seus elementos. Se a instância do elemento é um controlador ou processo, o tempo remanescente previsto de sua execução deve ser determinado utilizando essa definição recursivamente, tendo como condição de parada o cálculo dos tempos remanescente previstos das atividades folhas, determinados diretamente. No caso de um controlador iterativo, por exemplo, o tempo remanescente previsto de execução é o produto do número de vezes previsto de execução de seus elementos pelo tempo remanescente previsto de execução e de comunicação com esses elementos. No caso de um controlador condicional, o tempo remanescente previsto de execução desse é a somatória ponderada pela chance de executar cada um dos elementos multiplicado pelo tempo remanescente previsto de execução e de comunicação com esse elemento. Não é considerado o esforço computacional próprio do controlador ou processo, pois geralmente o código *script* contido nesses são utilizados para simples transformação de dados e de controle, não para a computação.

Os tempos remanescente de execução previstos são adequados para a seleção dos elementos que devem ter maior prioridade na utilização de recursos computacionais mais poderosos. Elementos com maior tempo remanescente previsto de execução entre os elementos prontos a serem executados têm uma maior change de reduzir o tempo total do processo se utilizar o esforço dos melhores recursos disponíveis. Assim, para reduzir o tempo de execução do processo é imprescindível priorizar a seleção de elementos com maior tempo remanescente previsto de execução para a alocação desses nos recursos mais poderosos, seja em processamento ou em capacidade de troca de dados. Porém, pelo fato de os tempos serem previstos, os seus valores absolutos não são muito significativos, o que importa são os valores relativos para a seleção de elementos.

Seleção de Elementos a Serem Alocados Esse algoritmo de escalonamento tem como objetivo minimizar o tempo total de execução do processo. Assim, após calculado o tempo remanescente de cada elemento, sejam eles controladores, processos ou atividades, o algoritmo de escalonamento seleciona o elemento mais crítico, ou seja, o que mais influi na diminuição do tempo global do processo, para ser alocado. Esse elemento então será alocado no recurso computacional mais poderoso da grade dentre os que contêm as características desejadas pelo elemento, de forma a reduzir o tempo de execução do processo.

A seleção de um elemento é realizada de cima para baixo. Partindo-se do processo, o algoritmo obtém o elemento com todas as dependências satisfeitas, ainda não alocado, com o maior tempo remanescente de execução, considerando o tempo de comunicação com o elemento pai. No caso dos controladores e processos, o tempo remanescente é calculado recursivamente somando-se somente os tempos remanescentes dos elementos descendentes não alocados. Se o elemento de maior tempo remanescente tem alguns elementos filhos não alocados, então o algoritmo de seleção é repetido recursivamente, até alcançar uma atividade não alocada de maior tempo remanescente ou um controlador (ou processo) não alocado de maior tempo remanescente que tem todos os seus elementos filhos alocados, considerando o tempo de comunicação. Esse elemento é selecionado. O algoritmo requisita a alocação do elemento selecionado para o melhor recurso disponível da grade e depois recalcula os tempos remanescentes do ramo percorrido, de baixo para cima.

O principal objetivo deste algoritmo é reduzir, a cada iteração, o tempo remanescente dos ramos da árvore sintática, que contém toda a estrutura hierárquica de controladores do processo, de forma balanceada, permitindo que os melhores recursos sejam distribuídos para os ramos de controladores com maiores tempos remanescentes primeiro, fazendo com que, idealmente, todos os ramos consigam o melhor tempo de execução possível, de forma a se obter o menor tempo possível para a execução do processo.

Escalonamento Distribuído baseado no Maior Caminho até o Término

O algoritmo de escalonamento distribuído com pesos dos elementos baseados no comprimento do maior caminho destes até o fim do processo é outro exemplo de algoritmo que pode ser utilizado no sistema Xavantes. Ele atribui pesos aos elementos de acordo com o comprimento total de processamento e de comunicação do maior caminho que parte do elemento até o final do processo. Após a atribuição de pesos aos elementos filhos de um controlador, o algoritmo requisita ao gerenciador de grupo a seleção e a alocação de servidores nós a esses elementos filhos de acordo com a ordem decrescente de seus pesos, respeitando a ordem do processo. Dessa forma, o peso define a prioridade de execução de um elemento. Uma vez alocado, cada controlador pode iniciar o escalonamento dos seus elementos filhos, e assim recursivamente, produzindo um algoritmo de escalonamento dis-

tribuído e paralelo. Na prática ele é um algoritmo HEFT (*Heterogeneous Earliest Finish Time*) recursivo, paralelo, distribuído e adaptável, para processos estruturados.

A vantagem deste algoritmo é que ele privilegia globalmente a execução de elementos que mais contribuem para o término antecipado do processo, portanto aumentando a velocidade de execução do processo, mesmo realizando o escalonamento em nível de controlador e de forma distribuída. Esse algoritmo realiza um escalonamento com visão global do processo, parcial (concorrente com a execução), dinâmico (adaptativo conforme a disponibilidade de recursos) e distribuído.

Atribuição de Pesos aos Elementos O peso de uma elemento é calculado como o comprimento do maior caminho a partir dele até o final do processo, incluindo o seu próprio comprimento. Assim, para a atribuição de pesos aos elementos, o algoritmo começa dos últimos elementos do processo até os primeiros, das atividades (folhas) até o processo raiz (raiz), respeitando a estrutura do processo formado pelo aninhamento dos controladores e a semântica de controle dos controladores.

O comprimento total de uma atividade é definido como o seu comprimento de processamento mais o seu comprimento de comunicação. O comprimento de processamento de uma atividade é calculado como o esforço necessário de processamento da atividade dividido por um poder computacional padrão, que representa uma média do poder de processamento disponível por nó nos grupos medianos. O comprimento de comunicação de um elemento, seja ele atividade, controlador ou processo, é calculado pelo esforço de comunicação entre o elemento e o seu pai dividido por uma velocidade de comunicação padrão, que representa uma média das velocidades disponíveis (por nó) das redes internas dos grupos medianos.

O comprimento de um caminho, seqüência de elementos, é igual à soma do comprimento de todos os seus elementos. O comprimento de um controlador é igual ao comprimento de seu maior caminho interno, calculado recursivamente, mais o seu comprimento de comunicação.

O peso de um elemento é igual ao comprimento do maior caminho desde esse elemento até o final do processo mais o comprimento do próprio elemento. Dessa forma, o peso dos últimos elementos do processo é o próprio comprimento desses elementos. A partir desse ponto, dos últimos elementos até os primeiros, o peso de um elemento (maior caminho até o fim do processo) é calculado como o maior peso entre os seus elementos sucessores mais o seu próprio comprimento. Para calcular o peso dos elementos filhos de um controlador, o sucessor dos últimos elementos filhos de um controlador é o sucessor desse controlador.

Em detalhes, se o elemento filho é um controlador, é atribuído um peso para o fim do controlador igual ao peso calculado para o próximo elemento da execução no mesmo nível. O peso do fim do controlador é utilizado para calcular o peso de seus elementos filhos.

O peso de um elemento filho de um controlador é igual ao comprimento desse elemento ao fim do controlador mais o peso do fim do controlador. Dessa forma, é possível obter localmente o peso dos elementos filhos, e portanto a sua prioridade relativa, global ao processo.

Para o nível do processo, que é seqüencial, o cálculo do peso de cada elemento filho é direto: o peso de um elemento filho é igual ao peso do elemento filho sucessor mais o seu comprimento. Os pesos dos elementos filhos de um controlador seqüencial são obtidos de forma parecida, mas considerando o peso do fim do controlador para calcular o peso do último elemento do controlador. Para um controlador iterativo seqüencial, é obtida a estimativa do número de iterações, e o peso de cada elemento filho depende da iteração. Em controladores paralelos, em que existem vários caminhos, são calculados os pesos dos elementos de cada caminho, de forma independente.

Requisição de Seleção e Alocação de Recursos O peso de um elemento é utilizado para determinar a prioridade de execução desse elemento: quanto maior o peso, maior a prioridade de execução. Após calculada a prioridade de um elemento, a partir de seu peso, o algoritmo pode requisitar, ao gerenciador de grupo, a seleção e a alocação de recursos para esse elemento. A alocação respeita naturalmente a ordem de execução dos elementos filhos e o tempo mínimo de início de um elemento a ser alocado é igual ao tempo de término previsto do anterior. No mesmo nível é possível alocar elementos filhos paralelamente, se não há dependências de execução entre eles.

Quando uma atividade é encontrada, ela é alocada com a prioridade atribuída, tempo mínimo de início, tempo máximo de fim e demais requisitos da atividade. Quando um controlador é encontrado, para a sua alocação também é utilizada a prioridade, tempo mínimo de início e tempo máximo de fim calculados. No entanto, quando um controlador é alocado a um servidor nó, ele pode iniciar a escalonar os elementos filhos, atribuindo prioridades e requisitando a alocação de cada um, e assim recursivamente. Assim o escalonamento pode ser distribuído e paralelizado. As vantagens são maior desempenho no escalonamento, tolerância a falhas e escalabilidade do escalonamento, mas sem perder uma visão global do processo.

Esse escalonamento pode ser parcial, em que o executor de processos requisita ao escalonador local para alocar elementos iterativamente, em turnos. Pode ser adaptativo, no sentido de o gerenciador de grupo realocar outros servidores nós para executar os elementos, caso haja nova disponibilidade de processamento ou máquinas, ou atraso na execução dos elementos nas máquinas atuais. O tempo mínimo de início também pode ser recalculado para os elementos seguintes de cada caminho: se um elemento filho terminar a sua execução antecipadamente ou tardiamente, com uma diferença acima de um certo fator em relação ao comprimento do elemento, o novo tempo mínimo de início dos sucessores é

calculado e há uma realocação de elementos.

6.2.2 Seleção de Recursos Computacionais

O gerenciador de grupos fornece informações sobre os recursos do grupo e responde à seleções de melhores recursos para executar atividades ou controladores (e processos). A seleção de recursos para as atividades considera os requisitos de execução, prioridade, tempo mínimo de início, tempo máximo de fim e esforço de processamento necessário da atividade e esforço de comunicação desta com o seu elemento pai. Existem dois tipos de seleção de recursos para controladores (e processos): a seleção que considera prioridade, tempo mínimo de início, tempo máximo de fim e esforço agregado de execução do controlador e esforço de comunicação deste com o seu elemento pai; e a seleção com base no tempo de comunicação com os seus elementos filhos.

Seleção de Recursos para Execução de Atividades

A seleção de um recurso para executar uma atividade leva em consideração o tempo mínimo de início, o tempo máximo de fim, a prioridade e o esforço computacional exigido pela atividade, em número de milhares de instruções, por um lado, e a agenda de execução e poder computacional do recurso, por outro lado. É possível especificar requisitos para a execução da atividade, tais como tamanho mínimo da memória volátil e da persistente, além de bibliotecas de classes e fontes de dados necessários. A seleção de recurso pode considerar o esforço de comunicação entre a atividade e o seu controlador pai. Nesse caso, o tamanho total das mensagens trocadas pela atividade com o controlador pai, bem como a velocidade da rede entre o recurso que contém a atividade e o recurso (ou grupo) que contém o controlador são considerados.

Seleção com base no Esforço de Processamento Para selecionar o melhor recurso do grupo local para executar uma atividade, o gerenciador de grupo realiza testes de alocação dessa atividade nas agendas dos servidores nó do grupo, mantidas na memória do servidor de grupo. Realizando essa simulação, ele pode obter qual é o melhor recurso local para executar a atividade, ou seja, o recurso que permite a atividade terminar no menor tempo. Para realizar esse teste, ele obtém da atividade a sua prioridade, o esforço estimado de processamento, em número de milhares de instruções, e o menor tempo de início que ela pode executar, definido pelas dependências de execução. Para cada agenda de atividades de servidor nó, o algoritmo tenta alocar essa atividade no tempo mais cedo que estiver disponível para a prioridade dada e poder de processamento mínimo requerido. Com base no esforço de processamento da atividade e no poder de processamento disponível do recurso, ele calcula a duração da atividade. Com esses dois

dados para um determinado servidor nó, o menor tempo de início e duração da atividade, o algoritmo calcula o tempo esperado de seu término e verifica se é menor que o tempo máximo de fim. Realizando esse cálculo para todos os servidores nó, o servidor nó em que o tempo de término for menor é selecionado para executar a atividade.

O algoritmo para selecionar o melhor recurso da grade, globalmente, tem três fases. Na primeira fase são selecionados os grupos conhecidos mais promissores para executar a atividade, na segunda, o melhor recurso de cada um desses grupos é retornado e, na última fase, o melhor recurso dentre estes retornados é selecionado. Para selecionar os grupos mais promissores, o gerenciador de grupo consulta agendas agregadas de disponibilidade média de poder de processamento dos servidores nó de todos os grupos que ele conhece, inclusive dele próprio. As agendas agregadas consideram o poder de processamento médio disponível para uma série de intervalos de tempo e de prioridades de execução. Com essa agenda agregada, para cada grupo o algoritmo calcula qual é o menor tempo de término previsto para a atividade e seleciona os n grupos que fornecerem os melhores resultados. Depois, esses n grupos são consultados para retornar o recurso com menor tempo de término para essa atividade. Na última fase, o recurso que retornar o menor de todos os tempos de término é selecionado.

Seleção com base no Esforço de Processamento mais o Esforço de Comunicação com o Elemento Pai A seleção global de recursos computacionais para executar uma certa atividade pode considerar também o esforço de comunicação com o controlador (ou processo) pai. Nesse caso, a duração da atividade é calculada como o tempo de processamento mais o tempo de comunicação para selecionar grupos e servidores nó que produzem o menor tempo de término para uma determinada atividade. Para calcular o tempo de comunicação considera-se, por um lado, o tamanho total das mensagens trocadas entre a atividade e o controlador pai e, por outro lado, a velocidade da rede entre o recurso testado e um certo recurso passado como parâmetro, que contém a possível localização do controlador pai. Dessa forma, para um recurso ser selecionado globalmente, na primeira fase adiciona-se o tempo da comunicação desta atividade com o grupo que contém seu controlador pai ao tempo de execução esperado para a atividade, para se calcular o tempo de término desta no grupo: os n grupos que retornarem o menor tempo de término são escolhidos. Na segunda fase, são requisitados a esses n grupos para que retornem o seu melhor recurso, o que produz o menor tempo de término, considerando o esforço de comunicação da atividade com o elemento pai. Na última fase, dentre os recursos retornado, o recurso com o menor tempo de término, considerando comunicação, é selecionado.

Seleção de Recursos para Coordenação de Processos e Controladores

O *middleware* oferece dois tipos de seleção de recursos para controladores (e processos): a seleção que considera prioridade, tempo mínimo de início, tempo máximo de fim e esforço agregado de processamento do controlador e esforço de comunicação deste com o elemento pai; ou a seleção com base no tempo de comunicação deste com os seus elementos filhos.

Seleção com base no Esforço de Processamento mais o Esforço de Comunicação com o Elemento Pai A seleção do melhor recurso para executar um controlador (ou processo) pode ser baseada no esforço total estimado de processamento do controlador mais o esforço de comunicação com o seu elemento pai. O esforço total estimado de processamento de um controlador é o esforço total estimado de processamento de seus elementos filhos, recursivamente.

A seleção do melhor recurso utiliza a prioridade, tempo mínimo de início, tempo máximo de fim e o esforço total estimado de processamento do controlador e o esforço de comunicação deste com o seu elemento pai, de um lado, e as agendas agregadas de disponibilidade total de poder de processamento por grupo, divididas por faixas de tempo e prioridades, e a velocidade da rede entre os dois grupos, do outro lado, para selecionar os n melhores grupos onde o controlador pode ser executado. Os melhores grupos são os que retornam os menores tempos de término para o controlador. O tempo de término é calculado como o tempo de início mais o tempo de execução. O tempo de execução é igual ao tempo de processamento mais o tempo de comunicação. O tempo de processamento é calculado dividindo-se o esforço de processamento pelo poder de processamento disponível na agenda agregada do grupo, para o tempo de início e prioridade do controlador. O tempo de comunicação é calculado dividindo-se o tamanho total das mensagens trocadas entre o controlador com o elemento pai pela velocidade da rede entre os dois grupos (se for o mesmo grupo, é utilizada velocidade da rede interna). Depois é requisitado a cada grupo para que esse selecione o melhor servidor nó, que é o mais disponível para o período de execução calculado do controlador. Dentre os servidores nó retornados, o com menor tempo de término retornado pelo grupo é o selecionado.

Esse método permite alocar o controlador antes de seus elementos filhos, recursivamente, da raiz para baixo, possibilitando um escalonamento distribuído e eficiente. Entretanto, ele não obtém o tempo de término preciso do controlador imediatamente, somente o tempo de término com base no valor agregado do grupo, pois o primeiro depende do tempo de término do último elemento filho, que não foi escalonado. Esse método também não considera o custo da comunicação do controlador com os elementos filhos imediatamente, embora os elementos filhos possam ser alocados depois considerando o tempo de comunicação entre eles e o controlador.

Seleção com base no Tempo de Comunicação com os Elementos Filhos A seleção do melhor recurso para executar um controlador (ou processo) também pode ser baseada no tempo de comunicação entre esse e os seus elementos filhos. O tempo da comunicação entre um controlador e seus elementos filhos é dado pela soma dos tempos de comunicação entre esse e cada um de seus elementos filhos. Cada tempo de comunicação entre um controlador e um elemento filho é igual à razão entre o tamanho total das mensagens trocadas entre o controlador e esse filho, em *bytes*, pela velocidade de comunicação, em *bytes* por segundo, entre o recurso que contém o controlador e o recurso computacional que contém esse filho.

Essa abordagem é útil para o modelo de baixo para cima, em que o escalonador aloca o controlador posteriormente à alocação de seus elementos filhos, no grupo ou na máquina em que o tempo de comunicação com os seus filhos for o menor possível, a fim de reduzir o tempo total de execução do processo. Essa abordagem dificulta o escalonamento distribuído, pois não é possível antecipadamente selecionar o melhor recurso para o controlador para que este realize o escalonamento de seus elementos filhos. Entretanto, como vantagem, considera o tempo de comunicação com os elementos filhos para ser alocado, otimizando a seleção do melhor recurso.

6.3 Tolerância a Falhas

Durante a execução de um processo podem ocorrer falhas na infra-estrutura física da grade e no *middleware*, além de erros na execução dos elementos de processo, tais como controladores e atividades. As falhas na infra-estrutura física e no *middleware* são denominadas falhas de sistema, uma vez que são ocasionadas pelo mal funcionamento dos seus componentes. Os erros na execução dos elementos de processo, conseqüências de falhas na lógica de negócio ou na especificação de processos, são considerados erros de execução, gerados por falhas da aplicação. Para maiores explicações sobre a definição dos termos *falha*, *erro* e *defeito*, utilizados na área de tolerância a falhas em sistemas distribuídos confiáveis, consultar Laprie [Lap95].

As falhas de sistema podem ser mascaradas do usuário e tratadas automaticamente, ou levantadas como exceção de sistema, no nível de processo. Os erros de execução são sempre explicitados no nível de processo como exceções. As exceções podem ser tratadas pelo processo ou apresentadas como defeito para o usuário, sendo que, no último caso, ele deve tratá-lo manualmente ou abortar o processo.

Hwang [HK03] classifica as formas de tolerância a falhas nos níveis de atividade e de processo, descrevendo os modos de percepção e recuperação de falhas nesses níveis. No nível de atividade, os sistemas existentes podem mascarar as falhas de sistema de diversas formas: retentando a atividade na mesma ou em outra máquina; replicando a sua

execução em máquinas da grade; ou salvando periodicamente o estado de uma atividade para restaurá-la no último ponto salvo. No nível de processo, a aplicação pode tratar explicitamente da falha de sistema ou de aplicação: executando atividades alternativas, em seqüência ou em paralelo; ignorando a execução da atividade; ou tratando da exceção gerada pela falha por meio de ações específicas. Esse trabalho não tratou da recuperação dos dados em caso de falhas e da manutenção de sua consistência em execuções concorrentes, pois os sistemas existentes utilizam dados isolados, geralmente arquivos, o que limita a concorrência e o tipo de aplicações possíveis. Além do mais, não foi descrito no trabalho como tratar de exceções não previstas, modificando a estrutura do processo e restaurando-o do ponto em que sua execução se encontrava no momento da falha.

Nessa seção descreveremos os mecanismos de tolerância a falhas do *middleware* Xavantes no nível de sistemas, atividades e processos; o tratamento de exceções; e a manutenção de consistência de dados.

6.3.1 *Middleware*

Para um sistema distribuído ser confiável, tanto ele próprio quanto as aplicações executadas sobre ele devem ser tolerantes a falhas e consistentes (livre de erros de lógica e de dados) [Gra86]. O *middleware* Xavantes, nesse sentido, possui todos os seus componentes, servidores de grupo e de nó, tolerantes a falhas do tipo *fail-stop* e de comunicação, com preservação da consistência dos dados do sistema. Os componentes recuperam seu estado automaticamente, a partir dos repositórios de dados compartilhados, e restauram os canais de comunicação, continuando sua execução a partir do ponto da falha.

Gerenciador de Grupo

O gerenciador de grupo mantém, de forma persistente, utilizando a base *xavantes_gs* de um repositório compartilhado, informações sobre o próprio grupo; o registro, estado de ativação e a localização de cada um dos servidores nó do grupo, bem como a agenda de alocação destes para a execução de processos, controladores e atividades. Mantém o registro, estado de ativação e a localização de cada um dos grupos pares, sejam eles servidores ou clientes, e a informação agregada da disponibilidade média dos recursos dos servidores nó dos grupos servidores. Contém a lista de clientes do grupo, com registro, estado de ativação e localização. Por último, contém as tabelas de processos, controladores e atividades invocadas a partir desse grupo ou em execução nesse grupo. Os arquivos de definição de processos e atividades das aplicações, bem como suas bibliotecas de classes, são armazenados em diretórios específicos do sistema de arquivo, chamados de domínios de aplicação. Assim, os principais dados do funcionamento do gerenciador de grupo, bem como informações sobre objetos do grupo, podem ser recuperados em caso de falhas.

Nome	Descrição
groud_id	identificação do cliente da grade
server_id	
object_id	
name	nome único no grupo
passwd	senha de autenticação
host	endereço da máquina
port	porta para obtenção da referência remota
active	indica se está ativo

Tabela 6.1: Tabela CLIENT_REGISTRY

A comunicação com os servidores nó do grupo e com os servidores de grupo pares pode ser restaurada de forma transparente, porque a localização armazenada fornece dados para a obtenção de uma nova referência RMI (*Remote Method Interface)* para o objeto remoto que faz a interface com esses serviços. Ademais, essas referências são atualizadas constantemente pelo monitoramento de estado dos serviços.

Abaixo apresentaremos detalhes das principais tabelas da base *xavantes_gs* no repositório compartilhado, que mantêm as informações acima descritas sobre o gerenciador de grupo.

Client_Registry A tabela *Client_Registry* (6.1) contém informações sobre os clientes do grupo e o estado de ativação e localização de sua interface *callback* para recepção de informações da grade. A identificação (*group_id*, *server_id*, *object_id*) do cliente é gerada no seu cadastro no grupo. O nome e a senha (*passwd*) são utilizados para autenticar o cliente na grade e para autorizar o seu acesso aos recursos da infra-estrutura. Os campos máquina (*host*) e a porta (*port*) são utilizados para se obter uma referência à interface *callback* do gerenciador de cliente, para que esse possa receber informações das atividades e dos processos executados e seus resultados. O campo ativo (*active*) indica se o cliente está correntemente ativo na grade. Com essas informações armazenadas de forma persistente, o cliente da grade pode se registrar e acessar a grade em caso de falha e posterior restauração do gerenciador de grupo.

Node_Service_Registry A tabela *Node_Service_Registry* (6.2) contém informações sobre os servidores nó do grupo. A identificação única (*group_id*, *server_id*, *object_id*) do servidor nó é gerada pelo gerenciador de grupo no momento do cadastro daquele no grupo. O nome e a senha são utilizados para ele se autenticar ao gerenciador de grupo, indicando que está ativo e pronto para executar atividades e processos pelo grupo. Os campos máquina (*host*) e a porta (*port*) são utilizados para se obter uma referên-

Nome	Descrição
groud_id	identificação do servidor nó
server_id	
object_id	
name	nome único no grupo
passwd	senha de autenticação
host	endereço da máquina
port	porta para obtenção da referência remota
arch_description	descrição da arquitetura da máquina
num_processors	número de processadores
processing_power	poder de processamento de cada processador
memory_size	tamanho da memória da máquina
disk_size	tamanho do disco da máquina
active	indica se está ativo

Tabela 6.2: Tabela NODE_SERVICE_REGISTRY

cia à interface de invocação do gerenciador de nó em caso de falha de comunicação. O campo *arch_ description* contém uma descrição textual e resumida da máquina que executa o gerenciador de nó. Os campos *num_ processors*, *processing_ power*, *memory_ size* e *disk_ size* contêm detalhes da máquina, especificamente, número de processadores (ou núcleos), poder de processamento agregado e tamanhos das memórias volátil e persistente, respectivamente. O campo ativo, conhecido como estado de ativação, indica se o servidor está ativo, ou seja, pronto para receber processos e atividades.

As referências aos servidores nó são mantidas pelo gerenciador de grupo e podem ser obtidas deste gerenciador toda vez que a comunicação com um servidor nó for perdida, como no caso de falha intermitente do canal de comunicação ou queda do servidor. Ademais, se o gerenciador de nó está ativo, ele constantemente informa a sua referência para o gerenciador de grupo. O estado de ativação do servidor nó é utilizado pelo gerenciador de grupo para saber se o servidor nó relacionado pode receber requisições. Em caso de inatividade por um determinado período, o servidor nó é marcado como inativo e as atividades são realocadas para outros servidores da grade. Os campos são mantidos de forma persistente, portanto em caso de falha do gerenciador de grupo, seus dados podem ser recuperados e a operação normal do grupo pode ser restaurada.

Group_ Service_ Registry A tabela *Group_ Service_ Registry* (6.3) contém informações sobre os grupos conhecidos do grupo local, sejam eles grupos clientes ou servidores para este grupo local. A identificação única (*group_ id*, *server_ id*, *object_ id*) do servidor de grupo é inserida de acordo com o número de cada grupo cadastrado na grade. O nome

Nome	Descrição
groud_id	identificação do servidor de grupo
server_id	
object_id	
name	nome único no grupo
passwd	senha de autenticação
host	endereço da máquina
port	porta para obtenção da referência remota
group_description	descrição do grupo
num_nodes	número de servidores nó do grupo
processing_power	poder de processamento total do grupo
intra_bandwidth	largura de banda da rede interna
inter_bandwidth	largura de banda entre o grupo local e o atual
client	indica se o grupo faz requisições para o grupo local
server	indica se o grupo recebe requisições do grupo local
active	indica se o servidor está ativo

Tabela 6.3: Tabela GROUP_SERVICE_REGISTRY

e a senha são utilizados pelos gerenciadores de grupo cliente se autenticarem perante o grupo local e indicarem que estão ativos. Os campos máquina (*host*) e a porta (*port*) são utilizados para se obter uma referência à interface de invocação do gerenciador de grupo em caso de falha de comunicação. O campo *group_ description* contém uma descrição textual e resumida do grupo registrado. Os campos *num_ nodes*, *processing_ power*, *intra_ bandwidth* e *inter_ bandwidth* contêm detalhes do grupo, especificamente, número de nós do grupo, poder de processamento total do grupo, e larguras de banda interna e externa com o grupo local, respectivamente. Os campos *client* e *server* indicam se o grupo registrado é cliente e servidor, respectivamente, do grupo local – ou seja, indicam se o grupo relacionado pode enviar e se pode receber requisições de alocação do grupo local. O campo ativo, conhecido como estado de ativação, indica se o gerenciador de grupo registrado está ativo, ou seja, pronto para receber requisições de alocação de processos e atividades.

As referências aos servidores de grupo remotos são mantidas pelo gerenciador de grupo local e podem ser obtidas deste gerenciador toda vez que a comunicação com um servidor de grupo for perdida, como no caso de falha intermitente do canal de comunicação ou queda do servidor. O estado de ativação do gerenciador de grupo é utilizado pelo gerenciador de grupo local para saber se o gerenciador de grupo remoto pode receber requisições. Em caso de inatividade por um determinado período, o gerenciador de grupo remoto é marcado como inativo e os elementos alocados a esses são realocados para outros grupos. Os campos são mantidos de forma persistente, portanto em caso de falha do gerencia-

dor de grupo, seus dados podem ser recuperados e a operação normal do grupo pode ser restaurada.

Activity_ Information A tabela *Activity_ Information* (6.4) contém informações sobre atividades invocadas por serviços locais (gerenciadores de cliente e gerenciadores de nó), ou em execução no grupo e invocadas por serviços locais ou externos. A identificação única da atividade (*group_ id*, *server_ id*, *object_ id*) é criada do momento da invocação e contém o número do grupo criador. O tipo da atividade é definida pelos campos *type_ group_ id*, *type_ domain* e *type_ name*. Os campos (*process_ group_ id*, *process_ server_ id*, *process_ object_ id*) identificam o processo pai da atividade. Os campos (*parent_ group_ id*, *parent_ server_ id*, *parent_ object_ id*) e *name* identificam o elemento pai da atividade e o nome da atividade, único no escopo do elemento pai. O invocador da atividade, que pode ser um gerenciador de cliente ou gerenciador de nó, é identificado por (*caller_ group_ id*, *caller_ server_ id* e *caller_ object_ id*).

Os campos *invocation_ time*, *absolute_ priority*, *relative_ priority*, *min_ start_ time*, *max_ end_ time*, *proc_ size*, *transfer_ size*, *min_ proc_ power* e *min_ memory_ size* armazenam, respectivamente, o momento da alocação, a prioridade absoluta do processo, a prioridade relativa aos elementos do processo, o tempo mínimo de início, o tempo máximo de fim, o esforço total de processamento em milhares de instruções, a transferência total de dados entre a atividade e o elemento pai em *kilobytes*, o poder mínimo de processamento requerido em milhares de instruções por segundo e a memória mínima requerida em *kilobytes*. O estado de execução da atividade é armazenado no campo *status*. Depois de alocado um servidor de nó para executar a atividade, os campos *executor_ group_ id*, *executor_ server_ id* e *executor_ object_ id* identificam o servidor nó. O identificador de alocação *allocation_ id* é utilizado para indicar qual é o número da alocação válida para essa atividade. O tempo de início, o tempo de fim e o poder de processamento médio alocado são armazenados em *start_ time*, *end_ time* e *proc_ power*, respectivamente.

As informações das agendas de atividades dos servidores nó locais, mantidas em memória, são obtidas dessa tabela, mas filtrando-se os executores locais pelo campo *executor_ group_ id*. Cada registro da agenda de atividades contém dados de uma atividade, com o identificador da atividade, o identificador do processo, as prioridades absoluta e relativa, o identificador do executor, os tempo de início e de término previstos e o poder de processamento alocado. A agenda é mantida em memória em uma estrutura que permite saber para um determinado tempo e prioridade, quanto poder de processamento há disponível, de forma a facilitar as simulações de agendamento de atividades para cada servidor nó do grupo.

Nome	Descrição
groud_id	identificação da atividade alocada
server_id	
object_id	
type_group_id	grupo de definição da atividade
type_domain	domínio da definição da atividade
type_name	nome da definição da atividade
process_groud_id	identificação do processo raiz
process_server_id	
process_object_id	
parent_groud_id	identificação do elemento pai (processo ou controlador)
parent_server_id	
parent_object_id	
name	nome único no escopo do elemento pai
caller_groud_id	identificação do invocador
caller_server_id	
caller_object_id	
invocation_time	tempo da invocação
absolute_priority	prioridade absoluta
relative_priority	prioridade relativa
min_start_time	tempo mínimo de início
max_end_time	tempo máximo de fim
proc_size	esforço de processamento (em kilo instruções)
transfer_size	quantidade de octetos transferidos (em kilobytes)
min_proc_power	poder de processamento mínimo requerido
min_memory_size	tamanho mínimo de memória requerido
status	estado de execução
executor_groud_id	identificação do servidor nó executor
executor_server_id	
executor_object_id	
allocation_id	identificador da alocação
start_time	tempo de início de execução previsto
end_time	tempo de término de execução previsto
proc_power	poder de processamento alocado

Tabela 6.4: Tabela ACTIVITY_INFORMATION

Nome	Descrição
groud_id	identificação do processo ou controlador alocado
server_id	
object_id	
type_group_id	grupo de definição do elemento
type_domain	domínio da definição do elemento
type_name	nome da definição do elemento
process_groud_id	identificação do processo raiz
process_server_id	
process_object_id	
parent_groud_id	identificação do elemento pai (processo ou controlador)
parent_server_id	
parent_object_id	
name	nome único no escopo do elemento pai
caller_groud_id	identificação do invocador
caller_server_id	
caller_object_id	
invocation_time	tempo da invocação
absolute_priority	prioridade absoluta
relative_priority	prioridade relativa
min_start_time	tempo mínimo de início
max_end_time	tempo máximo de fim
transfer_size	quantidade de octetos transferidos (em kilobytes)
status	estado de execução
executor_groud_id	identificação do servidor nó executor
executor_server_id	
executor_object_id	
allocation_id	identificador da alocação
start_time	tempo de início de execução previsto
end_time	tempo de término de execução previsto

Tabela 6.5: Tabela PROCESS_INFORMATION

Process_ Information A tabela *Process_ Information* (6.5) contém informações sobre processos e controladores invocados por serviços locais (gerenciadores de cliente e gerenciadores de nó) ou em execução no grupo e invocados por serviços locais ou externos. A identificação única do processo ou controlador (*group_ id*, *server_ id*, *object_ id*) é criada do momento da invocação e contém o número do grupo criador. O tipo do elemento é definido pelos campos *type_ group_ id*, *type_ domain* e *type_ name*. O campo *element_ type* indica se o elemento é um processo ou controlador. Os campos (*process_ group_ id*, *process_ server_ id*, *process_ object_ id*) identificam o processo pai do elemento, se o próprio elemento não for o processo raiz. Os campos (*parent_ group_ id*, *parent_ server_ id*, *parent_ object_ id*) e *name* identificam o elemento pai e o nome do elemento, único no escopo do elemento pai. O invocador do elemento, que pode ser um gerenciador de cliente, se o elemento for um processo raiz, ou gerenciador de nó, é identificado por (*caller_ group_ id*, *caller_ server_ id*, *caller_ object_ id*).

Os campos *invocation_ time*, *absolute_ priority*, *relative_ priority*, *min_ start_ time*, *max_ end_ time* e *transfer_ size* armazenam, respectivamente, o tempo da alocação, a prioridade absoluta do processo, a prioridade relativa aos elementos do processo, o tempo mínimo de início, o tempo máximo de fim e a transferência total de dados entre o elemento e o elemento pai em *kilobytes*. O estado de execução do elemento é armazenado no campo *status*. Depois de alocado um servidor de nó para executar o elemento, os campos *executor_ group_ id*, *executor_ server_ id* e *executor_ object_ id* identificam o servidor nó. O identificador de alocação *allocation_ id* é utilizado para indicar qual é o número da alocação válida para esse elemento. O tempo de início e tempo de fim são armazenados em *start_ time* e *end_ time*, respectivamente.

As informações das agendas de processos dos servidores nó, mantidas em memória, são obtidas dessa tabela, mas filtrando-se os executores locais pelo campo *executor_ group_ id*. Cada registro da agenda de processos contém dados de um processo ou controlador, com o identificador do elemento, o identificador do processo, as prioridades absoluta e relativa, o identificador do executor e os tempos de início e de término previstos. A agenda é mantida em memória em uma estrutura que permite saber para um determinado tempo e prioridade, quantos elementos ainda podem ser alocados para esse executor, de forma a facilitar as simulações de agendamento de processos e controladores para cada servidor nó do grupo.

Gerenciador de Nó

O gerenciador de nó mantém, de forma persistente, utilizando a base *xavantes_ ns* de um repositório compartilhado, informações sobre o próprio servidor nó; informações de alocação, estado de execução, parâmetros iniciais e resultados dos processos, controladores e atividades alocados; e estado da execução, com variáveis de dados e de controle, dos

elementos recuperáveis. Ademais, os arquivos de definição de processos e atividades das aplicações, bem como suas bibliotecas de classes, e outros arquivos de dados de cada instância da aplicação são armazenados em diretórios específicos do sistema de arquivo local, chamados de *caches* de aplicação. Assim, os dados para a restauração do gerenciador de nó podem ser recuperados em caso de falhas.

Para recuperação automática da comunicação entre o gerenciador de nó e o gerenciador de grupo, ambos mantêm referências ao outro. Cada gerenciador de nó contém uma referência persistente ao gerenciador de grupo local, utilizada para restaurar a comunicação com este no caso de falha e posterior recuperação deste. Ademais, o gerenciador de nó envia ao gerenciador de grupo periodicamente informações de disponibilidade juntamente com a referência ao objeto remoto que representa a sua interface. Dessa forma, o gerenciador de grupo contém uma referência atualizada para o gerenciador de nó caso haja uma falha e posterior restauração do servidor nó.

O gerenciador de nó sincroniza a sua lista de atividades e de processos com as mantidas pelo o gerenciador de grupo periodicamente para remover elementos e liberar recursos não mais necessários.

O executor de processos e o executor de atividades, contidos no servidor nó, são invocados para se restaurarem em caso de falhas. Esses, por sua vez, possuem acesso a informações do repositório compartilhado para recuperarem o estado dos processos, controladores e atividades alocados a essa máquina. Mais detalhes sobre a restauração de atividades e processos serão descritos nas Seções 6.3.2 e 6.3.3.

6.3.2 Atividades

O sistema Xavantes fornece alguns mecanismos para tolerar falhas de sistema em nível de atividades: retentativas de execução por número de vezes ou tempo especificado, na mesma ou em outra máquina. Se a atividade for recuperável, é retentada a restauração de sua execução, ao invés de sua reexecução, na mesma ou em outra máquina. A replicação da execução de atividades está prevista em trabalhos futuros, para prover maior disponibilidade e desempenho na execução de aplicações. Caso a falha não seja mascarada pelo *middleware*, uma exceção de sistema é levantada e passada para o nível de processo. Falhas de usuário geram exceções de aplicação que também são repassadas para serem tratadas em nível de processo.

Retentativas No nível de atividades, o *middleware* Xavantes retenta executar uma atividade um certo número de vezes ou por um tempo determinado, no caso da ocorrência de falhas de sistema. A cada falha, a atividade não-recuperável é abortada pelo sistema para a liberação de recursos correspondentes. A retentativa pode ocorrer na mesma máquina

ou em outra, posteriormente alocada pelo gerenciador de grupo. Um novo identificador de alocação é atribuído à atividade a cada tentativa. Se houver falha de comunicação com o gerenciador de grupo por um determinado período e o servidor nó alocado for reconhecido como inativo, a atividade é abortada por este e removida de sua lista de atividades. A falha da máquina é tratada localmente, por meio de sua reinicialização e reexecução da atividade. Nesse caso, se a atividade não terminar no tempo previsto, o gerenciador de grupo a realoca.

Caso a atividade seja recuperável, o *middleware* tenta restaurar a sua execução um certo número de vezes ou por um tempo determinado, enquanto houver falhas de sistema. A cada falha, a atividade é suspensa pelo sistema, sendo sua última transação interna, antes do próximo *checkpoint*, abortada pela definição da atividade, para liberação de recursos (dados compartilhados). A tentativa de restauração pode ocorrer na mesma máquina ou em outra, posteriormente alocada pelo gerenciador de grupo. Um novo identificador de alocação é atribuído à atividade a cada tentativa. Para que uma atividade seja restaurada em uma outra máquina, esta deve ter acesso ao repositório compartilhado da máquina anterior, para obter o estado de execução da atividade. Se houver falha de comunicação com o gerenciador de grupo por um determinado período e o servidor nó alocado for reconhecido como inativo, a atividade é suspensa por este e removida de sua lista de atividades. A falha da máquina é tratada localmente, por meio de sua reinicialização e restauração da atividade. Nesse caso, se a atividade não terminar no tempo previsto, o gerenciador de grupo a realoca.

Recuperação O gerenciador de nó armazena o estado de cada atividade, bem como parâmetros iniciais e resultados na tabela *Acitivity_Information* da base *xavantes_ns* em um repositório compartilhado. Cada recepção de parâmetros iniciais, mudança de estado da atividade, ou resposta da execução é persistida nesse repositório. Se a atividade for recuperável, em pontos determinados por sua especificação, o estado de execução desta, composto variáveis de dados e de controle, é persistido na tabela *Activity* da base *xavantes_ns* no repositório compartilhado como um objeto serializado. Dessa forma, o gerenciador de nó pode recuperar, em caso de continuação ou realocação, o estado de cada atividade para, automaticamente, reiniciar a execução das atividades iniciadas e não terminadas, ou enviar as respostas não entregues, no caso das terminadas.

Lançamento de Exceções As falhas de sistema que não foram possíveis de se mascarar com retentativas ou recuperação automática, e as falhas de usuário, geram exceções de sistema ou de aplicação, respectivamente. Nesse caso, as atividade não recuperáveis são abortadas e as recuperáveis, suspensas, se já não tiverem sido abortadas. Depois, as exceções são lançadas para o nível do processo para serem tratadas por seus mecanismos

de tolerância a falhas, descritos na Seção 6.3.3.

6.3.3 Processos

No nível de processos, o sistema Xavantes fornece dois mecanismos de tolerância a falhas: tratamento de exceções e recuperação de execução. As exceções podem ser lançadas por falhas de sistema e de usuário, sendo denominadas de exceções de sistema e de aplicação, respectivamente. As exceções também podem ser repassadas por elementos filhos que não as trataram, que podem iniciar nas atividades (folhas) e chegar até o processo raiz, passando pelos controladores. A recuperação da execução de processos é automática, podendo ser realizada na mesma máquina ou em outra que tenha acesso ao mesmo repositório compartilhado.

Tratamento de Exceções A linguagem XSPL fornece um mecanismo flexível para o tratamento de exceções de sistema ou de aplicação que podem ocorrer no corpo de um processo ou controlador ou serem repassadas por elementos filhos. Esse mecanismo permite ao usuário especificar a execução de qualquer tipo de atividade ou um *script* Java para tratar cada um dos tipos de exceção declarados, tal como ocorre em linguagens modernas de programação (vide Java e C#). As exceções são representadas por objetos Java instanciados de classes específicas que correspondem aos tipos das exceções. Com esse mecanismo é possível programar as formas de tolerância a falhas conhecidas, tais como atividades alternativas, atividades opcionais e compensação, além de possibilitar o tratamento fino de cada tipo de exceção, seja ela de sistema ou de aplicação. A lógica da compensação, entretanto, deve ser implementada pelo usuário, seja para compensar uma atividade ou um controlador completo, da mesma forma como ocorre em sistemas que explicitam o mecanismo de compensação.

Quando ocorre uma exceção em um controlador ou processo, o primeiro bloco *catch* com o tipo declarado igual ao tipo da exceção é executado. O corpo desse bloco pode conter qualquer seqüência de elementos, inclusive blocos *code* com *script* Java. Esse *script* Java pode modificar variáveis do controlador ou executar ações para restaurar a execução desse controlador ou abortá-lo. Essas ações podem ser condicionais à fonte da exceção (elemento filho) e ao estado do controlador (ou processo). Algumas ações possíveis são: abortar o elemento filho defeituoso (correspondente à ignorar um elemento filho opcional); solicitar a continuação da execução do elemento filho defeituoso após uma modificação dos dados utilizados; abortar o elemento filho e modificar uma variável do controlador indicando para executar um elemento filho alternativo na próxima iteração; abortar o controlador, o que faz o sistema abortar todos os elementos filhos em execução e suspensos; e relançar a exceção. A execução de blocos *catch* é serializada, de forma que

somente um é executado por vez, caso ocorram exceções em vários elementos filhos.

O bloco *finally* pode ser utilizado para liberar recursos e para compensar a execução de um controlador (ou processo) caso ele seja abortado. Esse bloco é o último a ser executado antes do término ou aborto do controlador. O corpo desse bloco pode conter qualquer seqüência de elementos, inclusive blocos *code* com *script* Java. O corpo desse bloco pode executar ações condicionais. A mais importante é verificar se o controlador foi abortado para compensar as suas ações e retornar o estado dos objetos compartilhados e fontes de dados para um anterior consistente, como se o controlador nunca tivesse sido executado.

Exceções não tratadas ou relançadas são repassadas para o elemento pai que deve tratá-la ou repassá-la, e assim sucessivamente, até alcançar o nível do usuário. Este é o último que decide como tratar a falha, podendo abortar todo o processo, ou modificar seus dados ou a sua definição dinamicamente e solicitar a continuação de sua execução.

Recuperação O gerenciador de nó armazena o estado de cada processo e controlador, bem como parâmetros iniciais e resultados na tabela *Process_ Information* da base *xavantes_ ns* em um repositório compartilhado. Cada recepção de parâmetros iniciais, mudança de estado do controlador (ou processo), ou resposta da execução é persistida nesse repositório. Ademais, as modificações do estado de execução de cada controlador (ou processo), composto de variáveis de dados e de controle em cada objeto da árvore de execução, é persistido em determinados pontos em tabelas correspondentes da base *xavantes_ ns* do repositório compartilhado. Dessa forma, o gerenciador de nó pode recuperar, em caso de reinicialização ou realocação, o estado de cada controlador para, automaticamente, reiniciar a execução dos iniciados e não terminados, ou enviar as respostas não entregues, no caso dos terminados.

A persistência do estado de um controlador ou processo é controlada pelo gerenciador de nó. Em pontos específicos, somente as modificações em relação ao último armazenamento são persistidos, evitando salvar toda a estrutura constantemente. Os pontos específicos são: após a inicialização do elemento, invocação de um elemento filho (processo, controlador ou atividade), término da execução de cada um desses elementos filhos, e antes do término do elemento. Em cada persistência são salvos os objetos modificados da árvore de execução do elemento em tabelas correspondentes na base *xavantes_ ns*. Por exemplo, as variáveis de um controlador paralelo e os identificadores dos caminhos disparados são salvos em *ParController*; e o estado de cada caminhos paralelo, com variáveis exclusivas dos caminhos e último elemento filho executado, são salvos em *Controller_ Thread*.

Objetos Compartilhados Os objetos compartilhados controlam o acesso a seus dados e a persistência de suas modificações. Cada objeto compartilhado persistente é salvo em

tabelas na base com o nome da aplicação no repositório compartilhado. A criação de tabelas, e a inserção, alteração e a remoção de registros é de responsabilidade do programador. O controle de acesso também é de responsabilidade do programador, permitindo acesso a leitura de dados concorrente e alteração serializada com o uso de mecanismos da linguagem Java (*synchronized*, *wait* e *notify*).

Um objeto compartilhado persistente pode ser implementado como uma camada de acesso transacional à uma tabela da base da aplicação no repositório compartilhado, com limites das operações definidos por transações iniciadas pelo usuário. Dessa forma, conexões Java obtidas do objeto compartilhado e contendo transações podem controlar o acesso e a persistência de dados automaticamente, sem a necessidade de o programador se preocupar com a implementação desses mecanismos. Esse objeto compartilhado, conhecido como tabela compartilhada, pode suportar consultas com filtros que retornam um objeto tabela em memória, por valor, com os registros filtrados e obtidos da tabela no repositório; e atualizações que recebem esse objeto tabela em memória, por valor, modificam os registros correspondentes da tabela no repositório compartilhado. Assim implementa-se acesso a dados relacionais, com *caches* distribuídos, em grades computacionais.

As referências aos objetos compartilhados são persistentes. A biblioteca básica implementa objetos *proxies* que mantêm a identificação do controlador em que reside o objeto compartilhado, a identificaçâo única (UID) do objeto compartilhado e o seu tipo; e intermediam as invocações aos objetos compartilhados remotos. Em caso de falhas do servidor nó em que está o controlador em que reside o objeto compartilhado, essas referências persistentes localizam o controlador e se comunicam com o gerenciador de nó correspondente para obter o objeto RMI que realiza a interface remota com o objeto compartilhado, recuperando a comunicação. Em caso de falhas do servidor nó do elemento que está usando o objeto compartilhado, as identificações do controlador, do objeto compartilhado e seu tipo são recuperadas do repositório compartilhado e assim o objeto *proxy* que implementa a referência persistente pode restaurar a comunicação com o objeto compartilhado remoto. Dessa forma, com referências persistentes, o *middleware* recupera a comunicação com objetos compartilhados em caso de falhas em qualquer uma das pontas.

6.4 Resumo

O *middleware* Xavantes executa processos estruturados definidos em XSPL de forma hierárquica, distribuindo controladores pelos recursos da grade, que controlam o escalonamento, alocação, distribuição e coordenação de outros controladores e assim recursivamente até o nível de atividade. Essa distribuição por controladores de elementos permite um escalonamento adaptativo, recuperação e comunicação localizada, proporcio-

nando um melhor desempenho, escalabilidade, confiabilidade e adaptabilidade à execução distribuída de processos. Os elementos são somente distribuídos a outros grupos caso os recursos locais sejam piores para executá-los, considerando o poder de processamento e a comunicação, em relação a recursos de outros grupos. Dessa forma, a seleção aproveita os melhores recursos da grade, elevando o desempenho ao melhor possível, dada a disponibilidade da grade.

A execução de processos compreende o escalonamento de seus elementos filhos, diretos e indiretos, e a coordenação da invocação dos elementos filhos alocados, de acordo com a ordem de execução definida. A execução de um processo ou controlador basicamente compreende os passos 2 a 13 e 15 da Figura 6.1. O escalonamento pode ser centralizado ou distribuído entre os controladores, que escalonam os seus elementos filhos recursivamente, até o nível de atividades. A coordenação do processo é distribuída entre os controladores, que coordenam os seus elementos filhos recursivamente.

No escalonamento, em nível local, são atribuídas prioridades para os elementos filhos e depois esses são alocados em ordem de execução, considerando a prioridade atribuída e de forma que o tempo mínimo de início seja igual ao tempo de término do anterior. O *middleware* Xavantes permite que algoritmos de escalonamento sejam acoplados ao módulo de escalonamento (escalonador). Os escalonamentos podem ser globais ou locais; totais ou parciais; estáticos ou adaptativos; e centralizados ou distribuídos. Na Seção 6.2.1 foram estudados algoritmos de escalonamento para processos especificados por grafos orientados acíclicos (DAGs), que é uma superclasse dos grafos estruturados, e foram propostos três algoritmos para os processos estruturados: um por Bittencourt [BMCB05] e dois originais para o trabalho. O *middleware* contém funções de auxílio aos algoritmos de escalonamento para obter a árvore de escalonamento dos processos, obter a disponibilidade prevista dos servidores nó e dos grupos, e selecionar e alocar os melhores recursos da grade para executar um elemento com os seus requisitos, tempo mínimo de início, tempo máximo de fim e prioridade atribuída. Os escalonamentos são automaticamente adaptativos pois o gerenciador de grupo realoca os elementos, respeitando os seus requisitos e prioridade, caso aumente o poder de processamento disponível na grade ou a máquina que cada elemento estiver falhar.

Os mecanismos de tolerância a falhas existentes no sistema Xavantes são essenciais para que as aplicações sejam confiáveis e disponíveis. Os serviços do sistema, gerenciador de grupo, de nó e de cliente podem se recuperar em caso de falhas, restaurando todos os seus dados de controle para que a sua execução continue a partir do ponto falho (*forward recovery*). O estado de ativação dos gerenciadores de nó e dos grupos servidores é monitorado pelo gerenciador de grupo, que se adapta e realoca os elementos em caso de desativação. A comunicação com o gerenciador de grupo é restaurada por meio de uma referência persistente à sua interface; e o gerenciador de grupo mantém referências atuais

para cada um dos gerenciadores de nó e de cliente do grupo e para os gerenciadores de grupos externos, que podem ser obtidas em caso de falhas.

No gerenciador de nó, em determinados pontos o estado de execução dos processos, controladores e atividades recuperáveis são persistidos. Portanto, se ocorrer uma falha no gerenciador de nó, esses elementos podem ser recuperados no próprio servidor nó onde eles estavam executando ou em um outro que tem acesso ao mesmo repositório compartilhado. A recuperação da execução ocorre no último ponto persistido (*forward recovery*).

As falhas de sistema em nível de atividade são mascaradas pelo sistema por meio de retentativas. Falhas de sistema ou de usuário não mascaradas em nível de atividade ou de sistema geram exceções correspondentes: de sistema ou de aplicação. Existem marcadores na linguagem XSPL que permitem tratar de exceções por meio de ações correspondentes, ou lançar exceções. Quando uma atividade recuperável, processo ou controlador não trata uma exceção, ele é suspenso para posterior correção. Nesse caso, se a falha for corrigida, a sua execução é retomada (*forward recovery)*, senão o elemento filho ou próprio elemento deve ser abortado (*backward recovery*). A atividade não recuperável é automaticamente abortada.

Capítulo 7

Implementação e Resultados Experimentais

Esse capítulo descreve a implementação do sistema Xavantes, responsável pela especificação e execução de processos estruturados, e os resultados obtidos de testes que validam a sua eficiência na execução de processos e na adaptação dinâmica à variabilidade do ambiente da grade.

7.1 Implementação

O sistema Xavantes foi escrito em *Java 6.0* [GJSB05], em sua última versão, com a utilização das facilidades providas pelas bibliotecas de classes padrão do pacote *Java Standard Edition 6.0 (Java SE 6.0)* [Mic06]. Para a comunicação remota entre os serviços foi utilizado o protocolo RMI (*Remote Method Invocation*) [Mic04], implementado pela biblioteca padrão, estendido com mecanismos de referência persistente para a comunicação com o gerenciador de grupo. Para a persistência de dados, nos repositórios compartilhados por grupo, foi utilizado o gerenciador de banco de dados *MySQL 5.0* [MyS07], acessado por meio de um *driver* do próprio desenvolvedor, que implementa o protocolo JDBC 2.0 [And06]. A execução de *scripts* Java contidos nos elementos de processo é realizada pelo interpretador *BeanShell 2.0* [Nie07], invocado diretamente por meio de uma biblioteca de classes fornecida.

O ambiente de desenvolvimento *NetBeans IDE 5.5* [Net07] foi utilizado para auxiliar a implementação do sistema. O código fonte do sistema foi dividido em quatro projetos principais e um de teste. Os quatro projetos principais após compilados e empacotados originam bibliotecas de mesmo nome. O sistema Xavantes está em implementação, atualmente em sua versão 0.2. As características das bibliotecas principais na versão atual do sistema são listadas a seguir:

xavantes_gs.jar — gerenciador de grupo: 45 classes e 7.182 linhas de código;

xavantes_ns.jar — gerenciador de nó: 61 classes e 10.516 linhas de código;

xavantes_client.jar — gerenciador de cliente: 13 classes e 1.561 linhas de código;

xavantes_core.jar — biblioteca comum: 77 classes e 3.478 linhas de código.

A biblioteca de classes do núcleo comum, *xavantes_core.jar*, é utilizada pelos três serviços Xavantes. Ela contém definições básicas de estruturas de dados, modelo de programação de processos XSPL (*Xavantes Structured Process Language*), e interfaces para invocações dos serviços remotos do *middleware* Xavantes por meio do protocolo RMI (*Remote Method Invocation*).

O sistema Xavantes é composto por três serviços principais: gerenciador de grupo (GM), gerenciador de nó (NM) e gerenciador de cliente (CM). O serviço gerenciador de grupo (GM) deve ser instanciado apenas na máquina servidora do grupo. Esse serviço é iniciado pela invocação do *script* `xavantes_gs`, que cria uma instância da máquina virtual Java (JVM) [LY99], com referências às bibliotecas *xavantes_gs.jar* e *xavantes_core.jar*, para executar o código do serviço a partir da classe *GroupServer*.

O serviço gerenciador de nó (NM) deve ser instanciado em todas as máquinas que estiverem disponíveis para executar processos ou atividades em um grupo. Este serviço pode executar uma instância do gerenciador de processos (PM), uma do gerenciador de atividades (AM), ou uma de cada, dependendo da finalidade da máquina sobre a qual ele está sendo executado (servidora de processos, servidora de atividades (*worker*), ou ambas, respectivamente). O serviço gerenciador de nó é iniciado pela invocação do *script* `xavantes_ns`, que cria uma instância da máquina virtual Java (JVM), com referências às bibliotecas *xavantes_ns.jar* e *xavantes_core.jar*, para executar o código do serviço a partir da classe *NodeServer*.

O serviço gerenciador de cliente (CM) é utilizado como *front end* para acessar o sistema Xavantes. Aplicações de usuário e interfaces gráficas podem interagir diretamente com ele, a partir de sua interface de programação, para utilizar os serviços de grade do sistema Xavantes. Para interagir com esse serviço cliente deve-se referenciar as bibliotecas de classes *xavantes_client.jar* e *xavantes_core.jar*. O serviço cliente é iniciado por meio da classe *Client*, que exporta a interface *callback* para monitoramento da execução de elementos de processo e retorna uma instância de *ClientManager*, que, por sua vez, contém todas as operações de interação com os serviços do sistema Xavantes. Entre essas, existem operações para: implantação da definição de processos e atividades no grupo local; invocação de processos e atividades, com a passagem de parâmetros iniciais de execução e informações para alocação; obtenção de resultados de atividades e processos invocados; e

controle e monitoramento da execução de processos. O serviço cliente também é um servidor na medida em que aceita invocações do tipo *callback* para, por exemplo, ser notificado do término de atividades de forma assíncrona. É possível construir uma interface gráfica que utilize esse serviço cliente para especificar e invocar processos, bem como monitorar e controlar suas instâncias, de forma simples e dinâmica.

A versão 0.2 tem implementadas, até o momento (*build 08/08/2007*), as funções de gerenciamento de recursos de grupo, alocação de recursos para elementos, execução de atividades e de processos, e recuperação do sistema. A versão 0.1, mais simples, sem alocação de recursos para elementos, porém com implementação mais estável, contendo funções de gerenciamento de recursos de grupo, execução de atividades e de processos, e escalonamento adaptativo de processos, foi utilizada para realizar os testes de conceito, apresentados na Seção 7.2.

7.2 Resultados Experimentais

Essa seção demonstra, por meio de experimentos, a alta adaptabilidade do *middleware* proposto para a coordenação de processos de forma a lidar com o ambiente dinâmico da grade. Todos esses experimentos rodam o exemplo de número primos (`FindPrimes`) do Capítulo 4 sobre configurações distintas da grade. Primeiramente, essa aplicação exemplo é executada sobre diferentes números de máquinas disponíveis, com o intuito de mostrar a eficiência do *middleware* no uso do poder computacional disponível e também revelar algumas características da aplicação. Depois, a aplicação `FindPrimes` é executada sobre um ambiente dinâmico, em que as máquinas são incluídas e excluídas da grade durante a sua execução. Esse exemplo mostra a eficiência do *middleware* em tratar modificações ocorridas em tempo de execução, ocasionadas por falhas e variação de disponibilidade das máquinas, com o uso de reescalonamento e redistribuição de atividades.

Todos os experimentos executam a aplicação `FindPrimes` para encontrar 100.000 números primos entre 2.000.000 e 10.000.000 usando 80 atividades que buscam primos independentemente sobre faixas distintas de números. Os primeiros quatro experimentos usam, respectivamente, 1, 2, 4 e 8 computadores como servidores nó (*workers*) e um computador distinto como o servidor de grupo, todos utilizando o Linux 2.6.x como sistema operacional. O poder de processamento total disponível nos servidores nó para cada um desses experimentos é 5.585, 11.076, 18.604 e 22.174 milhões de instruções por segundo, em MIPS, respectivamente. Esses valores foram extraídos do diretório *proc* de cada máquina rodando Linux, para validação, e depois somados para obter o poder computacional total em cada experimento. Na Figura 7.1 é apresentado quanto tempo, em segundos, cada experimento (exp1, exp2, exp3 e exp4) precisou para calcular um certo número de primos. Para encontrar os 100.000 números primos, os experimentos de 1 a 4 precisaram

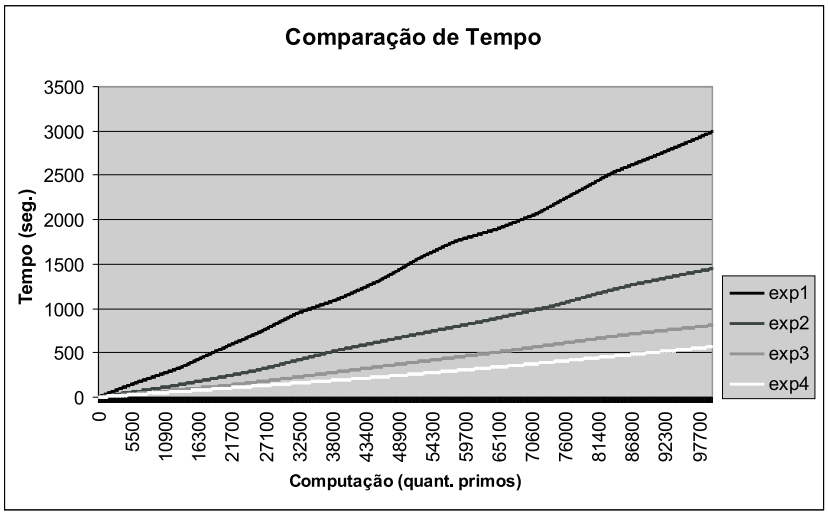

Figura 7.1: Comparação de tempo.

de 2.998, 1.448, 812 e 569 segundos, respectivamente. Dessa forma é possível observar, por meio desses experimentos, que o *middleware* utiliza proporcionalmente todo o poder computacional disponível para escalonar, distribuir e executar as atividades, uma vez que o tempo para obter um certo número de primos é praticamente inversamente proporcional à soma do poder computacional total dos recursos disponíveis.

Pelo motivo de a aplicação `FindPrimes` ter atividades com diferentes necessidades computacionais, pois a obtenção de números primos em faixas contendo números maiores requer a execução de um número maior de instruções, é possível perceber algumas variações de desempenho, em quantidade de primos obtidos por segundo, durante a execução. A Figura 7.2 apresenta essas variações de desempenho no tempo para cada experimento. Nota-se por meio desse gráfico que, mesmo com as variações apresentadas, os experimentos que usam mais poder computacional da grade são proporcionalmente mais rápidos na média, conforme esperado.

Diferentemente dos primeiros quatro experimentos, em que o poder computacional disponível é constante durante a execução, o experimento 5 testa o *middleware* proposto em um ambiente de grade real, inerentemente dinâmico, executando a aplicação `FindPrimes`. Nesse experimento, o servidor de grupo é iniciado em uma máquina do grupo, e 10 servidores nó (*workers*) são iniciados e parados durante a execução da aplicação em 10 máquinas distintas da grade com diferentes poderes computacionais. A Figura 7.3 mostra a variação de desempenho da aplicação de acordo com as variações do poder computacional disponível no tempo: a linha preta representa o poder computacional disponível percebido em

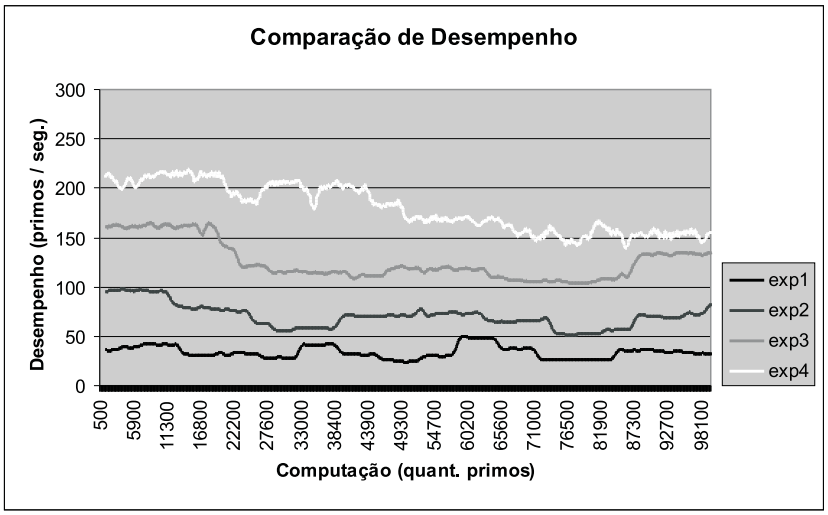

Figura 7.2: Comparação de desempenho.

um certo tempo e a linha cinza representa a variação de desempenho, em primos por segundo, dependendo dos recursos disponíveis, no tempo.

Nesse experimento é possível verificar que há uma grande correlação entre a variação do poder computacional disponível e a variação do desempenho da aplicação, demonstrando, portanto, que o *middleware* é dinamicamente adaptável à variabilidade do ambiente da grade, com o objetivo de usar todos os recursos disponíveis. O *middleware* adapta a distribuição de atividades aos melhores recursos por meio de reescalonamento e redistribuição de atividades não terminadas, com execução iniciada ou não, para novos servidores nó. Particularmente, a aplicação `FindPrimes` aproveita a vantagem das variáveis compartilhadas do controlador para persistir o estado das atividades regularmente, e assim elas podem ser recuperadas a partir do ponto salvo em outros trabalhadores.

Outrossim, pode-se observar pela Figura 7.3 que o *middleware* prontamente utiliza o poder computacional adicionado para executar atividades, isso devido à proximidade do gerenciador de processos em relação aos servidores nó, que as executam. Como conseqüência, tem-se uma adaptação eficiente para o ambiente dinâmico da grade.

7.3 Resumo

Um protótipo do sistema Xavantes foi construído implementando-se em *Java 6.0* os seus três principais serviços (gerenciador de grupo, gerenciador de nó e gerenciador de cliente) para executar sobre máquinas virtuais Java (JVM). Esses serviços utilizam as bibliotecas

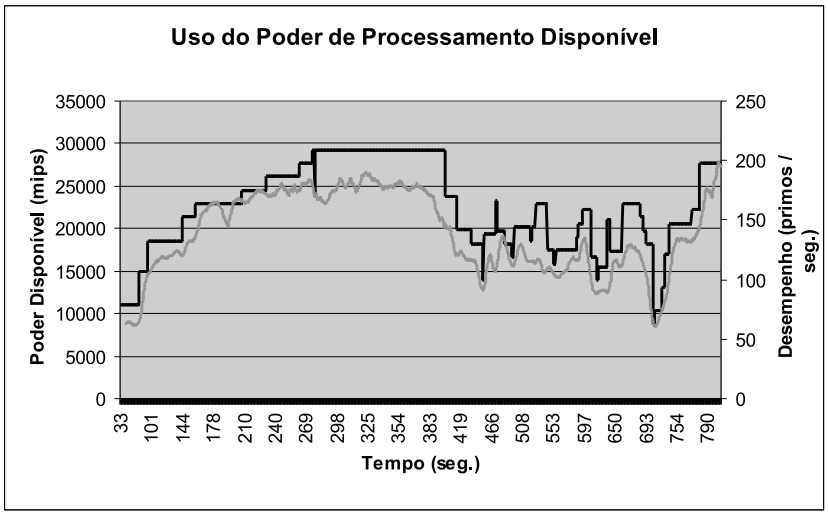

Figura 7.3: Uso do poder de processamento disponível.

de classes padrão do pacote *Java Standard Edition 6.0* para realizar tarefas comuns, utilizar os recursos da máquina local, comunicar-se (protocolo RMI), transferir dados e acessar bancos de dados relacionais (interface JDBC). Os repositórios compartilhados são controlados por sistemas gerenciadores de banco de dados *MySQL 5.0*.

Foram implementados 4 pacotes de classes para construir o protótipo do sistema Xavantes, versão 0.2: xavantes_gs.jar (gerenciador de grupo); xavantes_ns.jar (gerenciador de nó); xavantes_client.jar (gerenciador de cliente) e xavantes_core.jar (núcleo comum).

Essa versão contém implementadas, até o momento (*build 08/08/2007*), as funções de gerenciamento de recursos de grupo, alocação de recursos para elementos, execução de atividades e de processos, e recuperação do sistema. A versão 0.1, mais simples, sem alocação de recursos para elementos, porém com implementação mais completa, contendo funções de gerenciamento de recursos de grupo, execução de atividades e de processos, e escalonamento adaptativo de processos, foi utilizada para realizar os testes de conceito.

Os testes realizados com o protótipo do sistema Xavantes validaram as principais colaborações dessa tese. Para os testes foi utilizada a aplicação *FindPrimes*, definida na Seção 4.4.1, que busca números primos em uma faixa especificada por meio da execução paralela de atividades na grade computacional.

Os testes mostraram que o sistema permite construir aplicações para grade facilmente, utiliza eficientemente os recursos disponíveis para executar aplicações, adapta-se às variações de disponibilidade rapidamente, recupera-se de falhas parciais (nos servidores nó) e restaura a execução de atividades. Assim, prova-se que o escalonamento adaptável e

a execução hierárquica de processos estruturados, realizados pelo *middleware* Xavantes, permite desenvolver aplicações eficientes, escaláveis, adaptáveis e confiáveis, abrindo caminho para que desenvolvedores domésticos ou de grandes instituições utilizem o sistema Xavantes para explorar o potencial computacional da Internet.

Capítulo 8

Conclusão

A área de computação em grade pretende suportar a execução distribuída e paralela de tarefas sobre uma ampla área de domínios autônomos e heterogêneos, por meio do compartilhamento transparente de recursos computacionais e de dados em nível global, estendendo o uso da Internet, que hoje somente promove compartilhamento de documentos e alguns serviços, e aproveitando, assim, de forma muito mais abrangente, o seu poder para a evolução da humanidade.

O uso de recursos da grade para resolver problemas da sociedade, seja de governos, empresas, universidades ou indivíduos, é uma necessidade crescente, assim como é a do uso da Internet. O uso das grades computacionais auxilia na resolução de problemas científicos complexos, como o descobrimento da cura de doenças, mapeamento de genomas, e simulações mais precisas de experimentos físicos. Elas também provêem o aumento do poder computacional de empresas, instituições e indivíduos para realizar, de forma mais eficiente, por exemplo, mineração de dados e conhecimento, e simulação de projetos e produtos de engenharia e arquitetura, além de produzir filmes e criar mundos virtuais, com o uso massivo de usuários, mais realistas [GRI07a, Ros05]. Tudo isso, ainda, facilitando a tarefa dos usuários, desenvolvedores e administradores na grade, devido à transparência de acesso e aos serviços de gerenciamento de recursos providos pela sua infra-estrutura.

Entretanto, a maioria dos sistemas existentes de grades computacionais apóia somente a execução de tarefas independentes, não suportando diretamente a especificação, execução e monitoramento de processos de longa duração que compõem e coordenam a invocação e a troca de dados entre várias tarefas, deixando a cargo do usuário a programação da coordenação de tarefas, o que dificulta o seu uso para aplicações de longa duração. Ademais, por não reconhecerem o conceito de processos diretamente, não há suporte à recuperação transparente de processos, não é permitido o tratamento de exceções em etapas determinadas do processo e, por não considerar as dependências entre as atividades, esses sistemas não permitem a utilização de algoritmos eficientes de escalonamento de

atividades que minimizam o custo de troca de dados, reduzem o tempo de execução da aplicação e otimizam a utilização compartilhada de recursos.

Para resolver esses problemas, recentemente surgiram alguns sistemas que introduziram o suporte a processos que coordenam a execução de tarefas. Entretanto, eles ainda não provêem as características e funcionalidades necessárias para a especificação simples e expressiva, e para a execução eficiente, escalável e confiável de processos. Nenhum desses sistemas facilita a construção, o monitoramento e a modificação de processos expressivos ou contém uma infra-estrutura totalmente distribuída e controlável, com recuperação de falhas, tratamento de exceções, otimização de uso de recursos (máquinas e redes) e algoritmos adaptativos e eficientes para execução de processos.

Esse trabalho apresentou o sistema Xavantes, nos Capítulos 4, 5 e 6, que tem por objetivo suprir as deficiências acima expostas. Ele é formado pela linguagem de programação XSPL (*Xavantes Strutured Process Language*) e pelo *middleware* Xavantes. A linguagem de programação estruturada permite criar facilmente fluxos de controles complexos e paralelos em processos, combinando e aninhando os controladores de fluxos disponíveis, de forma similar a linguagens estruturadas de programação. O *middleware*, por sua vez, aproveita a composição dos controladores de fluxo para hierarquicamente escalonar, distribuir e executar os elementos de processo pelos grupos da grade, de forma que atividades e controladores fortemente dependentes fiquem em nós próximos, ou no mesmo, quando possível. Dessa forma, devido à distribuição controlada, é possível prover melhor desempenho de execução, tolerância a falhas, escalonamento adaptável, otimizar o uso de recursos e minimizar o custo de comunicação entre atividades. Como resultado o sistema provê especificação, monitoramento e execução distribuída da aplicação, mas de forma organizada e controlada, com um alto nível de desempenho, escalabilidade, confiabilidade e adaptabilidade.

Esse trabalho defende a tese de que o sistema Xavantes cumpriu os seus objetivos e portanto obteve-se um avanço na definição de um ambiente eficaz de desenvolvimento e de execução de aplicações de longa duração para grades computacionais. Para obtermos evidências que validam a tese, implementamos um protótipo do sistema Xavantes de execução de processos estruturados em grades computacionais [CMB04] e o utilizamos em experimentos de adaptação dinâmica, desempenho e recuperação na ocorrência de falhas parciais [CMB06]. Os detalhes da implementação e dos experimentos foram descritos no Capítulo 7.

Os experimentos demonstram a validade das principais contribuições do sistema Xavantes e, portanto, da tese:

- a linguagem de especificação estruturada de processos, baseada na composição de controladores seqüenciais e paralelos, permitiu a fácil especificação de uma aplicação paralela genérica;

- a utilização de uma variável compartilhada, próxima à computação, tornou possível a troca de dados entre as atividades de forma eficiente e confiável, otimizando o uso da rede de dados;

- o *middleware* executou o processo estruturado de forma hierárquica, permitindo que o controlador escalonasse, distribuísse e coordenasse a execução de atividades conforme máquinas fossem disponibilizadas, provendo escalabilidade e aumento de desempenho relativo para a aplicação genérica;

- o *middleware* tolerou falhas dos servidores nó, de modo que estes executassem atividades a partir de sua ativação;

- o *middleware* permitiu que o controlador realocasse as atividades para máquinas distintas quando as anteriores falharam ou quando novas surgiram, responsivamente, possibilitando a adaptação do processo às condições da grade.

Entretanto, ainda há muito para implementar e testar para que todos os conceitos e contribuições dessa tese se tornem parte da realidade. Ademais, esse trabalho de implementação e de testes pode proporcionar novas pesquisas, desenvolvimentos e contribuições, originando novas dissertações e teses, ampliando-se mais o conhecimento sobre essa área e originando um sistema cada vez mais completo e importante para os usuários e instituições que desejam não somente obter documentos da Internet, mas utilizá-la para executar suas aplicações desde domésticas até as de grande importância para o desenvolvimento de outros campos da ciência humana ou essenciais para os negócios privados ou de interesse público (governamentais).

No nível de projeto do sistema Xavantes, ainda há várias desafios a serem superados para o seu uso mais efetivo na grade, objetos de trabalhos futuros:

- definição precisa e simples de políticas e mecanismos de segurança de acesso aos recursos dos grupos autônomos, por meio de certificados digitais com chaves públicas distribuídas e criação de listas de acesso por grupo, papel e usuário;

- mecanismos de definição de prioridades entre usuários e grupos, derivados dos utilizados em redes *peer-to-peer* para compartilhamento de arquivos, semelhantes ao do *Network of Favors* do OurGrid [CBA+06], para permitir um compartilhamento justo dos recursos da grade, mas que levem em consideração a colaboração acumulada (esforço computacional fornecido, pecuniária e outras) mais os adicionais definidos;

- desenvolvimento de melhores serviços de informação e de monitoramento de recursos globais, que retornem mais precisamente a previsão do poder de processamento disponível em cada máquina ou grupo por período de tempo;

- previsão mais precisa do custo de processamento e de comunicação dos elementos de processo, sem a necessidade da intervenção do usuário, por meio de simulações;
- replicação consistente de dados, atividades e controladores em grades, para garantir maior tolerância a falhas, desempenho e disponibilidade;
- projeto de uma interface gráfica que permita especificar, invocar, monitorar e modificar em tempo de execução as aplicações baseadas em processos estruturados.

Referências

[AC03] T. Andrews and F. Curbera. *Specification: Business Process Execution Language for Web Services Version 1.1.* IBM DeveloperWorks, 2003. Available at http://www-106.ibm.com/developerworks/library/ws-bpel.

[AFFH98] E. Akarsu, G. C. Fox, W. Furmanski, and T. Haupt. WebFlow: High-Level Programming Environment and Visual Authoring Toolkit for High Performance Distributed Computing. In *Proceedings of Supercomputing (SC98)*, 1998.

[AGK00] D. Abramson, J. Giddy, and L. Kotler. High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid? In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 520–528, Cancun, Mexico, May 2000.

[And04] D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Proceeding of the 5th IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 4–10, 2004.

[And06] L. Andersen. *JDBC 4.0 Specification - JSR 221.* Sun Microsystems, Palo Alto, CA, USA, final release edition, November 2006.

[App07] Net Applications. Market Share for Operating Systems. http://marketshare.hitslink.com/report.aspx?qprid=2, September 2007.

[Bab07] A. Babenhauserheide. GNUFu: Gnutella for Users Wiki. http://rakjar.de/gnufu/index.php/GnuFU_en, June 2007.

[BAG00] R. Buyya, D. Abramson, and J. Giddy. Nimrod/G: An Architecture of a Resource Management and Scheduling System in a Global Computational Grid. In *Proceedings of the HPC Asia 2000*, pages 283–289, Beijing, China, May 2000.

[Bak07] S. Baker. Google and the Wisdom of Clouds. *Business Week*, December 2007.

[Bau04] W. Bausch. *OPERA-G: A Microkernel for Computational Grids*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 2004.

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1 – 10, 1995.

[BDF+03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, 2003.

[Ber96] P. A. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM*, 39(2):86–98, 1996.

[BM06] L. F. Bittencourt and E. R. M. Madeira. A Dynamic Approach for Scheduling Dependent Tasks on the Xavantes Grid Middleware. In *Proceedings of the 4th ACM International Workshop on Middleware for Grid Computing*, Melbourne, Australia, 2006.

[BMCB05] L. F. Bittencourt, E. R. M. Madeira, F. R. L. Cicerre, and L. E. Buzato. A Path Clustering Heuristic for Scheduling Task Graphs onto a Grid. In *Proceedings of the 3rd ACM International Workshop on Middleware for Grid Computing*, Grenoble, France, 2005.

[BP04] T. Bray and J. Paoli. *Extensible Markup Language (XML) 1.0*. XML Core WG, W3C, 2004. Available at http://www.w3.org/TR/2004/REC-xml-20040204.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Reading Mass, 1999.

[Cas02] H. Casanova. Distributed Computing Research Issues in Grid Computing. *ACM SIGAct News*, 33(3):50–70, 2002.

[CBA+06] W. Cirne, F. V. Brasileiro, N. Andrade, L. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the World, Unite!!! *Journal of Grid Computing*, 3(4):225–246, 2006.

[Cen07] NEES Cyberinfrastructure Center. NEES Cyberinfrastructure Center Portal. http://it.nees.org/, August 2007.

[CG01] B. Carlsson and R. Gustavsson. The Rise and Fall of Napster - An Evolutionary Approach. In *Proceedings of the 6th International Computer Science Conference on Active Media Technology*, volume 2252 of *Lecture Notes in Computer Science*, pages 347–354, Hong Kong, China, December 2001. Springer.

[CGH+05] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming Scientific and Distributed Workflow with Triana Services. *Grid Workflow 2004 Special Issue of Concurrency and Computation: Practice and Experience*, To be published, 2005.

[CJSN03] J. Cao, S. A. Jarvis, S. Saini, and G. R. Nudd. GridFlow: Workflow Management for Grid Computing. In *Proceedings of the International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 2003.

[CM02] S. Cheung and V. Matena. *Java Transaction API (JTA) Specification, Version 1.1*. Sun Microsystems, Palo Alto, CA, USA, november 2002.

[CMB04] F. R. L. Cicerre, E. R. M. Madeira, and L. E. Buzato. A Hierarchical Process Execution Support for Grid Computing. In B. Schulze, R. Nandkumar, and F. Kon, editors, *Proceedings of the 2nd ACM International Workshop on Middleware for Grid Computing*, pages 87 – 92, Toronto, Ontario, Canada, 2004. ACM Press.

[CMB06] F. R. L. Cicerre, E. R. M. Madeira, and L. E. Buzato. Structured Process Execution Middleware for Grid Computing. *Concurrency and Computation: Practice and Experience*, 18(6):581 – 594, 2006.

[Coa05] WfMC Workflow Management Coalition. Workflow Process Definition Interface – XML Process Definition Language (XPDL) Version 1.14. Technical Report WFMC-TC-1025, WfMC Workflow Management Coalition, Brussels, Belgium, October 2005.

[Coa06] WfMC Workflow Management Coalition. WfMC Workflow Management Coalition Home Page. http://www.wfmc.org/, May 2006.

[Con01] World Wide Web Consortium. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, March 2001.

[Con03] World Wide Web Consortium. SOAP Version 1.2 Part 0: Primer. http://www.w3.org/TR/soap12-part0, June 2003.

[Cor06] OASIS Corsortium. OASIS Corsortium Home Page. http://www.oasis-open.org/, May 2006.

[Cor07a] Microsoft Corporation. *C# Language Specification, Version 3.0*. Microsoft Corporation, 2007.

[Cor07b] Microsoft Corporation. Queues in Windows Communication Foundation. http://msdn.microsoft.com/en-us/library/ms731089.aspx, 2007.

[Cor08a] Microsoft Corporation. COM: Component Object Model Technologies. http://www.microsoft.com/com/default.mspx, 2008.

[Cor08b] Microsoft Corporation. .Net Framework Developer Center. http://msdn.microsoft.com/en-us/netframework/default.aspx, August 2008.

[Dat06] LSC DataGrid. LSC DataGrid Home Page. http://www.lsc-group.phys.uwm.edu/lscdatagrid/, May 2006.

[DBG+04] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, and M. Livny. Pegasus: Mapping Scientific Workflows onto the Grid. In *European Across Grids Conference*, pages 11–20, 2004.

[DW95] J. Dongarra and D. Walker. MPI: A Message-Passing Interface Standard. Technical report, Message Passing Interface Forum, June 1995.

[EN94] J. Esparza and M. Nielsen. Decidability Issues for Petri Nets. *BRICS Report Series – Department of Computer Science – University of Aarhus*, RS-94-8:1 – 23, May 1994.

[ERAL95] H. El-Rewini, H. H. Ali, and T. Lewis. Task Scheduling in Multiprocessing Systems. *Computer*, 28(12):27–37, 1995.

[ES01] D. W. Erwin and D. F. Snelling. UNICORE: A Grid Computing Environment. In *Euro-Par '01: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, pages 825–834, London, UK, 2001. Springer-Verlag.

[Fea05] I. Foster and H. Kishimoto et al. *The Open Grid Services Architecture, Version 1.0*. Global Grid Forum (GGF), 2005.

[fEs06] Enabling Grids for E-sciencE. Enabling Grids for E-sciencE Home Page. http://www.eu-egee.org/, May 2006.

[FI03] I. Foster and A. Imanitchi. On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. In *Peer-To-Peer Systems II: 2nd International Workshop (IPTPS 03)*, 2003.

[FK97] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl. J. Supercomputer Applications*, 11(2):115–128, 1997.

[FK99] I. Foster and C. Kesselman. Computational Grids. *The Grid: Blueprint for a New Computing Infrastructure*, pages 15–51, 1999.

[FKNT02] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration*. Open Grid Service Infrastructure WG, Global Grid Forum, 2002.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organization. *The Intl. Journal of High Performance Computing Applications*, 15(3):200–222, 2001.

[For94] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. Technical report, Computer Science Dept., University of Tennessee, Knoxville, TN, 1994.

[Fos02] I. Foster. *What is the Grid: A Three Point Checklist*, 2002. Available at http://www-fp.mcs.anl.gov/ foster/Articles/WhatIsTheGrid.pdf.

[Fos05] I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems. In *IFIP International Conference on Network and Parallel Computing*, pages 2–13, 2005.

[FTL+01] J. Frey, T. Tannenbaum, M. Livny, I. Foster, and S. Tuecke. Condor-G: A Computational Management Agent for Multi-institutional Grids. In *Proceedings of the Tenth Intl. Symposium on High Performance Distributed Computing (HPDC-10)*. IEEE, 2001.

[FVWZ02] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *Proceedings of the 14th Conference on Scientific and Statistical Database Management*, 2002.

[Gar04] L. M. Garshol. BNF and EBNF: What Are They and How Do They Work? http://www.massey.ac.nz/ kahawick/159331/BNF+EBNF-Garshol.pdf, 2004.

[GHM06] S. Graham, D. Hull, and B. Murray. *Web Services Base Notification 1.3 (WS-BaseNotification)*. OASIS, October 2006.

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[GJSB05] J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Prentice Hall PTR, 2005.

[GKGF04] A. Goldchleger, F. Kon, A. Goldman, and M. Finger. InteGrade: Object-Oriented Grid Middleware Leveraging Idle Computing Power of Desktop Machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, 2004.

[Gra86] J. Gray. Why Do Computers Stop and What Can Be Done about It? *Proceedings of the 1986 Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[Gri06] Open Science Grid. Open Science Grid Home Page. http://www.opensciencegrid.org/, May 2006.

[GRI07a] GRIA. Grid Resources for Industrial Applications Portal. http://www.gria.org/, August 2007.

[Gri07b] Earth Systems Grid. Earth Systems Grid Portal. http://www.earthsystemgrid.org/, August 2007.

[Gro06] EJB 3.0 Expert Group. JSR-000220 Enterprise JavaBeans 3.0. http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html, May 2006.

[GW96] A. S. Grimshaw and W. A. Wulf. Legion - A View from 50,000 Feet. In *Proceedings of the Fifth Intl. Symposium on High Performance Distributed Computing*. IEEE, 1996.

[H+04] D. Hollingsworth et al. The Workflow Reference Model: 10 Years On. *Workflow Handbook*, pages 295–311, 2004.

[HK03] S. Hwang and C. Kesselman. GridWorkflow: A Flexible Failure Handling Framework for the Grid. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13)*, pages 126 – 137, Seattle, Washington, USA, 2003.

[Hol94] D. Hollinsworth. The Workflow Reference Model. Technical Report WFMC-TC-1003, WfMC Workflow Management Coalition, Brussels, Belgium, December 1994.

[IOF95] M. Iverson, F. Ozguner, and G. Follen. Parallelizing Existing Applications in a Distributed Heterogeneous Environment. In *Proceedings of the 4th Heterogeneous Computing Workshop (HCW'95)*, pages 93–100, April 1995.

[IT07] E. Ilavarasan and P. Thambidurai. Low Complexity Performance Effective Task Scheduling Algorithm for Heterogeneous Computing Environments. *Journal of Computer Science*, 3(2):94–103, 2007.

[KTF03] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 2003.

[Lap95] J.C. Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. *Twenty-Fifth International Symposium on Fault-Tolerant Computing*, 1995.

[LK02] M. Lorch and D. G. Kafura. Symphony - A Java-Based Composition and Manipulation Framework for Computational Grids. In *2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2002), 22-24 May 2002, Berlin, Germany*, pages 136–143, 2002.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, Second Edition edition, 1999.

[Mic04] Sun Microsystems. *Java Remote Method Invocation (RMI) Specification, Version 1.5*. Sun Microsystems, Palo Alto, CA, USA, 2004.

[Mic06] Sun Microsystems. *Java Development Kit 6 Manual*. Sun Microsystems, Palo Alto, CA, USA, 2006.

[MLD05] S. McGough, W. Lee, and J. Darlington. ICENI II Architecture. In *Proceedings of the UK e-Science All Hands Meeting 2005*, Nottingham, UK, September 2005.

[MYA+04] S. McGough, L. Young, A. Afzal, S. Newhouse, and J. Darlington. Workflow Enactment in ICENI. In *Proceedings of the UK e-Science All Hands Meeting 2004*, pages 894 – 900, Nottingham, UK, September 2004.

[MyS07] MySQL AB. *MySQL 5.0 Reference Manual*, 2007.

[Net07] NetBeans Organization. *NetBeans 5.5 Documentation*, 2007.

[Nie07] P. Niemeyer. BeanShell - Simple Java Scripting Documentation. http://www.beanshell.org/docs.html, 2007.

[OAS06] OASIS. Web Services Resource Framework (WSRF) – Primer v1.2. http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf, May 2006.

[Obj08] Object Management Group – OMG. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1*, January 2008.

[OH98] R. Orfali and D. Harkey. *Client/Server Programming with Java and CORBA*. John Wiley & Sons, Inc. New York, NY, USA, 1998.

[Pro02] MetaNEOS Project. MetaNEOS Portal. http://www-unix.mcs.anl.gov/metaneos/, February 2002.

[Pro07] Apache Ant Project. Apache Ant Online Manual. http://ant.apache.org/manual/index.html, September 2007.

[Raj98] G. S. Raj. A Detailed Comparison of CORBA, DCOM and Java/RMI. Technical report, Object Management Group (OMG), September 1998.

[Ros05] D. Rosenberg. Building the Business Case for Grid. *IBM Developer Website*, June 2005.

[SRD+05] D. Sulakhe, A. Rodriguez, M. D'Souza, M. Wilde, V. Nefedova, I. Foster, and N. Maltsev. GNARE: An Environment for Grid-Based High-Throughput Genome Analysis. In *Proceedings of Third International Workshop on Biomedical Computations on the Grid, Cardiff, UK, 2005*, 2005.

[Sri95] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. Technical Report RFC 1831, The Internet Engineering Task Force – IETF, August 1995.

[TC05] OASIS Web Services Business Process Execution Language (WSBPEL) TC. Web Services Business Process Execution Language Version 2.0 – Committee Draft. http://www.oasis-open.org/committees/download.php/16024/wsbpel-specification-draft-Dec-22-2005.htm, December 2005.

[Tea02] Condor Team. *The Directed Acyclic Graph Manager*, 2002. Available at www.cs.wisc.edu/condor/dagman.

[THW02] H. Topcuouglu, S. Hariri, and M. Wu. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Trans. Parallel Distrib. Syst.*, 13(3):260–274, 2002.

[TSWR03] I. Taylor, M. Shields, I. Wang, and O. Rana. Triana Applications within Grid Computing and Peer to Peer Environments. *Journal of Grid Computing*, 1(2):199 – 217, 2003.

[TTL02] D. Thain, T. Tannenbaum, and M. Livny. Condor and the Grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., December 2002.

[vLAH+04] G. van Laszewski, K. Amin, M. Hategan, N. J. Zaluzec, S. Hampton, and A. Rossi. GridAnt: A Client-Controllable Grid Workflow System. In *Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS-37)*, pages 126 – 137, Hilton Waikoloa Village, Island of Hawaii, USA, 2004.

[YB05] J. Yu and R. Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. *SIGMOD Rec.*, 34(3):44 – 49, 2005.

[Yer03] E. J. H. Yero. *Estudo sobre Processamento Maciçamente Paralelo na Internet*. PhD thesis, Faculdade de Engenharia Elétrica e de Computação, Universidade Estadual de Campinas, July 2003.

[YYR03] J. Yan, Y. Yang, and G. K. Raikundalia. Enacting Business Processes in a Decentralised Environment with P2P-Based Workflow Support. In *Proceedings of the Fourth Intl. Conference on Web-Age Information Management (WAIM 2003)*, 2003.

[Zei06] K. D. Zeilenga. *Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map*. OpenLDAP Foundation, June 2006.

[ZWF+06] Y. Zhao, M. Wilde, I. Foster, J. Voeckler, J. Dobson, E. Gilbert, T. Jordan, and E. Quigg. Virtual Data Grid Middleware Services for Data-Intensive Science. *Concurrency and Computation: Practice and Experience*, 18(6):595 – 608, 2006.