Uma Arquitetura Baseada em CORBA para Workflow de Larga Escala

Roberto Silveira Silva Filho

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Uma Arquitetura Baseada em CORBA para Workflow de Larga Escala

Roberto Silveira Silva Filho

Agosto de 2000

Banca Examinadora

- Dr. Jacques Wainer (Orientador)
  IC - UNICAMP – Universidade Estadual de Campinas
- Dr. Otto Carlos Muniz Bandeira Duarte
  COPPE/EE - UFRJ – Universidade Federal do Rio de Janeiro
- Dr. Luiz Eduardo Buzato
  IC- UNICAMP – Universidade Estadual de Campinas
- Dra. Maria Beatriz Felgar de Toledo (Suplente)
  IC - UNICAMP – Universidade Estadual de Campinas

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Silva Filho, Roberto Silveira

Si38a Uma arquitetura baseada em CORBA para workflow de larga escala / Roberto Silveira Silva Filho - Campinas, [S.P. :s.n.], 2000.

Orientador : Jacques Wainer

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

1. Sistemas distribuídos operacionais (Computadores). 2. CORBA (Arquitetura de computador). 3. Software - Arquitetura. I. Wainer, Jacques. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Uma Arquitetura Baseada em CORBA para Workflow de Larga Escala

Este exemplar corresponde à redação final da dissertação, devidamente corrigida e defendida por Roberto Silveira Silva Filho, aprovada pela banca examinadora.

Campinas, 21 de Agosto de 2000

Jacques Wainer
(Orientador)

Edmundo Roberto Mauro Madeira
(Co-orientador)

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

© Roberto Silveira Silva Filho 2000.
Todos os direitos reservados.

Ao único Deus soberano, criador,
doador de toda a sabedoria, de todo
o conhecimento e da salvação em
Jesus Cristo

Agradecimentos

Aos meus pais, Ana e Roberto, pelo seu apoio, especialmente durante estes sete anos e meio de graduação e mestrado.

A D. Lourdes por ter me acolhido e recebido em Campinas durante todos estes anos.

Aos meus bons amigos durante estes sete anos de UNICAMP, em especial a Christian, Marlon, Katlin, e Adriane, não esquecendo do Akira e do Fábio e todos os outros que foram verdadeiros companheiros durante todos estes anos em Campinas, tornando este período de minha vida certamente inesquecível.

Aos meus orientadores Jacques Wainer e Edmundo R. M. Madeira pelas críticas, sugestões e idéias que tornaram este trabalho possível, inclusive permitindo a colaboração com o professor Clarence (Skip) Ellis da Universidade do Colorado, o que resultou em uma viagem aos Estados Unidos e algumas publicações em conjunto.

A FAPESP (processo número 98/06648-0) e ao CNPq pelo apoio financeiro durante a vigência deste projeto.

Resumo

Sistemas de Gerenciamento de Workflow (SGWFs) tradicionais possuem uma limitação intrínseca de escalabilidade, o servidor central, que representa um gargalo de desempenho e um único ponto de falhas em sistemas onde um grande número de casos simultâneos precisa ser executado. Com base nesta deficiência dos SGWFs tradicionais, é proposto o projeto e a especificação de uma arquitetura distribuída, utilizando as funcionalidades do ambiente aberto de distribuição CORBA, de forma a suportar, em primeiro lugar, os requisitos de escalabilidade, disponibilidade e confiabilidade dos SGWFs de larga escala. Esta arquitetura utiliza a idéia de casos (instâncias de processos) móveis que migram pelos nós do sistema, seguindo o plano do processo, conforme as atividades do workflow são realizadas. A arquitetura é estendida de maneira a satisfazer outros requisitos de SGWFs de larga escala como segurança, recuperação de falhas, interoperabilidade, segurança e outros. Questões relacionadas ao mapeamento desta arquitetura para CORBA e sua implementação são discutidas, juntamente com suas limitações, vantagens e decisões de projeto. Foi realizada a análise dos custos de migração, configuração e criação dos agentes móveis da arquitetura. Testes de desempenho, envolvendo configurações totalmente centralizadas e distribuídas foram definidos e realizados. Nestes testes, configurações distribuídas tiveram maior desempenho que as centralizadas para instâncias envolvendo a execução simultânea de mais de 5 casos concorrentes.

Abstract

Standard client-server workflow management systems have an intrinsic scalability limitation, the centralized server, which represents a bottleneck for large-scale applications. This server also is a single-failure point that may disable the whole system. In this work, it is proposed a fully distributed architecture for workflow management systems. It is based on the idea that the case (an instance of the process) migrates from host to host, following a process definition, while the case corresponding activities are executed. This basic architecture is improved so that other requirements for Workflow Management Systems, such as fault tolerance, monitoring, interoperability, security and others, are also contemplated. A CORBA-based implementation of such architecture is discussed, with its limitations, advantages and project decisions described. The mobile agent migration, creation and configuration costs were computed. Performance tests, ranging from full centralization to distribution, were defined and performed. In these tests, the distributed configuration performed better than the centralized configuration for instances with more than 5 concurrent cases.

Conteúdo

Conteúdo .... **i**
Lista de Figuras v
Lista de Tabelas vi
Lista de Gráficos vii

Capítulo 1 Introdução **1**
1.1 Motivação 1
1.1.1 Principais Limitações dos Sistemas Existentes 2
1.2 Objetivos 3
1.2.1 Hipótese 3
1.2.2 Estratégia 3
1.3 Estrutura do Trabalho 4

Capítulo 2 Workflow **5**
2.1 Histórico 5
2.2 Origens 7
2.2.1 Automação de Escritórios 7
2.2.2 Gerenciamento de Bancos de Dados 8
2.2.3 Modelos de Transações Avançadas 8
2.2.4 E-Mail 9
2.2.5 Gerenciamento de Documentos 9
2.2.6 Processo de Software 10
2.2.7 Modelagem de Processos Empresariais e Modelagem de Arquitetura de Empresas 10
2.2.8 Aplicações de *Groupware* 11
2.3 Aplicações 11
2.4 Definição de Sistemas de Gerenciamento de Workflow 13
2.5 Classificação 14
2.6 Estrutura de um Sistema de Gerenciamento de Workflow (SGWF) 15
2.7 Modelo de Referência da WfMC 18
2.7.1 Terminologia 19
2.8 Principais Requisitos dos SGWFs Tradicionais 20
2.9 Workflow Distribuído e de Larga Escala 22
2.9.1 Problemas dos SGWFs Convencionais 22
2.9.2 Requisitos Adicionais de SGWFs de Larga Escala 23

Capítulo 3 Fundamentos de CORBA e Agentes Móveis **25**

3.1 OMA e CORBA 25
3.1.1 Estrutura de um ORB 27
3.1.2 O Futuro da arquitetura OMA 31
3.1.3 Principais Vantagens de CORBA....32
3.1.4 Principais Desvantagens de CORBA 32
3.1.5 OrbixWeb 33
3.2 Agentes Móveis....35
3.2.1 Paradigma de Agentes Móveis 35
3.2.2 Aplicações 36
3.2.3 Vantagens 37
3.2.4 Requisitos 38
3.2.5 Sistemas de Agentes Móveis....39
3.2.6 Propostas do OMG 40
3.2.7 Principais Requisitos dos Sistemas de Agentes Móveis 40
3.2.8 Linguagens de Programação 43

Capítulo 4 Arquitetura WONDER 45
4.1 Modelo Distribuído 45
4.2 Componentes da Arquitetura....48
4.2.1 Coordenador de Processo 48
4.2.2 Coordenador de Caso 49
4.2.3 Coordenador de Papel 50
4.2.4 Atividade de Sincronização (Synchronization Activity)....51
4.2.5 Atividades Roteadora (*Gateway Activity*) 52
4.2.6 Lista de Tarefas (*TaskList*) 53
4.2.7 Servidor de Histórico (*HistoryServer*) 53
4.2.8 Servidor de Backup (*Backup Server*) 54
4.2.9 Gerenciador de Atividade (*Activity Manager*) 55
4.2.10 Interpretador de Planos (*PlanInterpreter*)....56
4.2.11 Gerenciadores de Aplicações (*Wrappers*)....56
4.2.12 Ambiente de Suporte a Objetos do Workflow 56
4.3 Cenários de Execução 58
4.3.2 Seqüenciamento de Atividades 59
4.3.3 Criação de Um Caso....61
4.3.4 Atividades AND-Split....61
4.3.5 Atividades de Sincronização 62
4.3.6 Finalização do Caso....63
4.3.7 Recuperação de Falhas 63

Capítulo 5 Mapeamento para CORBA e Java 65
5.1 CORBA e Java 65
5.1.1 Serviços CORBA 66
5.1.2 Referência a Objetos CORBA....68
5.1.3 Ambiente de Suporte a Workflow....70
5.2 Hierarquia de Interfaces 70

5.2.1 Data Container ... 72
5.2.2 Compilador PLISP ... 73
5.2.3 Suporte a conversão de Links em Dados ... 74
5.2.4 Compilador WStarter ... 74
5.3 Máquinas de Estado ... 75

Capítulo 6 Paralelo entre o Paradigma de Agentes Móveis e a Arquitetura WONDER .. 81
6.1 Transportabilidade ... 81
6.2 Autonomia ... 82
6.3 Navegabilidade ... 82
6.4 Segurança ... 83
6.5 Tolerância a falhas ... 83
6.6 Desempenho ... 84
6.7 Suporte multiplataforma ... 85
6.8 Adaptabilidade ... 85
6.9 Comunicação ... 85
6.10 Serviço de Nomes ... 86
6.11 Persistência de Objetos ... 86
6.12 Espaço de nomes de objetos WONDER ... 86
6.13 Conclusões da Seção de Agentes Móveis e WONDER ... 87

Capítulo 7 Implementação ... 89
7.1 Simplificações do Modelo Implementado ... 89
7.2 Soluções de Implementação ... 92
7.2.1 Workflow Object Factory ... 92
7.2.2 Persistência de Objetos e Timeout ... 92
7.2.3 Utilização do Espaço de Nomes ... 93
7.2.4 LockManager ... 93
7.2.5 OrbixWeb e Escalabilidade ... 94
7.2.6 Peculiaridades do OrbixWeb ... 96

Capítulo 8 Testes de Desempenho ... 97
8.1 Objetivos ... 97
8.2 Descrição dos Testes ... 98
8.2.1 Ruídos ... 101
8.3 Relação dos Testes ... 101
8.3.1 Análise do *Overhead* da Arquitetura ... 101
8.3.2 Estudo da Variação do Número de Casos Concorrentes - Sem Processamento ... 102
8.3.3 Estudo da Variação do Número de Casos Concorrentes - Com Processamento .. 103
8.3.4 Estudo da Variação do Volume de Dados Trocado ... 103
8.4 Medidas Empregadas ... 103
8.5 Análise dos Resultados ... 106
8.6 Testes ... 107
8.6.1 Análise do *Overhead* da Arquitetura ... 107
8.6.2 Análise da Variação do Número de Casos Concorrentes - Sem Processamento .. 112

8.6.3 Análise da Variação do Número de Casos Concorrentes - Com Processamento..120
8.6.4 Análise da Variação do Volume de Dados Trocado 123
8.7 Comentários e Conclusões 125

Capítulo 9 Trabalhos Relacionados.. 129
9.1 IBM Flowmark..129
9.2 Exotica/FMQM 130
9.2.1 Modelo Distribuído 130
9.2.2 Principais Características 131
9.2.3 Exotica/FMQM X WONDER...131
9.3 IBM MQSeries Workflow...132
9.3.1 MQSeries Workflow X WONDER...133
9.4 Mentor 133
9.4.1 Mentor X WONDER...134
9.5 INCA (INformation CArriers) Workflow 134
9.5.1 INCA X WONDER...136
9.6 $METEOR_2$ 137
9.6.1 Modelo $METEOR_2$...137
9.6.2 ORBWork...138
9.6.3 $METEOR_2$ X WONDER...139
9.7 Proposta da Nortel para o OMG...140
9.7.1 Nortel X WONDER 142
9.8 Proposta do OMG...142
9.9 Outros Sistemas..143

Capítulo 10Conclusões... 145
10.1 Principais Características e Contribuições 146
10.2 Requisitos da Distribuição 147
10.3 Discussão...148
10.4 Trabalhos Futuros...149

Referências.. 151

Apêndice A 160
A.1 Notação da BNF (*Backus Normal Form*):...160
A.2 Gramática da linguagem PLISP 161
A.3 Exemplo de um plano escrito em PLISP..162
A.4 Gramática do compilador WStarter...163
A.5 Exemplo de um ambiente de testes a ser montado pelo compilador WStarter 164

Apêndice B 165

Apêndice C 166

Apêndice D 174

Lista de Figuras

Figura 1: Processo de Revisão de Padrões do OMG 12
Figura 2: Características de um Sistema de Gerenciamento de Workflow 16
Figura 3: Estrutura Genérica de um Workflow Management System 17
Figura 4: Modelo de Referência de Workflow - WfMC 18
Figura 5: Relacionamento entre os elementos da terminologia básica 19
Figura 6: Arquitetura OMA 26
Figura 7: Componentes do Modelo de Referência CORBA 28
Figura 8: Processo de Geração de Stubs e Skeletons a partir de uma interface IDL. 28
Figura 9: Passagem de uma requisição remota de um cliente a um servidor. 29
Figura 10: Estrutura de uma IOR (Interoperable Object Reference) 31
Figura 11: Os principais Componentes da arquitetura e seus relacionamentos 48
Figura 12: Diagrama de Interação Cliente-LOA 58
Figura 13: Diagrama de seqüenciamento de atividades. 60
Figura 14: Diagrama de criação de um caso 61
Figura 15: Sincronização AND-Join 62
Figura 16: Diagrama de seqüência do procedimento de finalização 63
Figura 17: Ambiente de Suporte a Workflow da Arquitetura WONDER 70
Figura 18: Hierarquia de Classes e Interfaces IDL da arquitetura WONDER 71
Figura 19: Hierarquia de dados e objetos armazenados no DataContainer 72
Figura 20: DTE da classe Activity 76
Figura 21: DTE da calasse ActivityManager 77
Figura 22: DTE da classe SynchronizationActivity 77
Figura 23: DTE da classe CaseCoordinator 78
Figura 24: DTE da classe ProcessCoordinator 79
Figura 25: Exemplo de um ambiente de testes com vários casos em paralelo. 99
Figura 26: Execução alternada de atividades consecutivas em testes distribuídos. 100
Figura 27: Execução alternada de atividades em nós distribuídos. Os coordenadores são criados em um nó à parte 102
Figura 28: Intervalos de Tempos Medidos 104
Figura 29: Tempos associados a duas atividades consecutivas de um mesmo caso 105
Figura 30: Tarefa composta 140
Figura 31 Controladores para tarefas compostas 141
Figura 32: Representação Gráfica do Workflow do Item A.3 usando a notação da WfMC 163

Lista de Tabelas

Tabela 1: CRC do Coordenador de processo 49
Tabela 2: CRC do Coordenador de caso 50
Tabela 3: CRC do Coordenador de papel..51
Tabela 4: CRC da Atividade de Sincronização...52
Tabela 5: CRC da Atividade Roteadora..52
Tabela 6: CRC da Lista de Tarefas 53
Tabela 7: CRC do Servidor de Histórico...54
Tabela 8: CRC do Servidor de Backup 54
Tabela 9: CRC do Activity Manager...55
Tabela 10: CRC do Interpretador de Planos..56
Tabela 11: CRC do Gerenciador de Aplicações..56
Tabela 12: CRC do Ativador de Objetos Local 57
Tabela 13: CRC do Repositório de Objetos..57
Tabela 14: Dados de Execução do ambiente de teste centralizado, sem atividades invocadas e sem dados trocados. Máquina iguacu...107
Tabela 15: Dados médios de execução dos testes envolvendo a execução de casos em ambientes centralizados, sem processamento e sem troca de dados do caso. Máquinas: araguaia, iguacu, anhumas e tigre. 108
Tabela 16: Percentual relativo dos dados médios de execução dos testes envolvendo a execução de casos em ambientes centralizados, sem processamento e sem troca de dados do caso. Máquinas: araguaia, iguacu, anhumas e tigre...108
Tabela 17: Dados de médios de execução de testes envolvendo a execução de casos em ambientes distribuídos, sem processamento e sem troca de dados do caso. Máquinas: anhumas, araguaia e iguacu, e araguaia, tigre e anhumas. 109
Tabela 18: Percentual relativo dos dados de médios de execução de testes envolvendo a execução de casos em ambientes distribuídos, sem processamento e sem troca de dados do caso. Máquinas: anhumas, araguaia e iguacu, e araguaia, tigre e anhumas. 109
Tabela 19: Dados de Execução do ambiente de teste centralizado, sem processamento. Máquina: araguaia.ic.unicamp.br 112
Tabela 20: Dados de Execução dos testes envolvendo 1 a 20 casos concorrentes em ambiente centralizado. Máquina: araguaia.ic.unicamp.br..113
Tabela 21: Dados de execução dos testes envolvendo 1 a 20 casos concorrentes em ambiente distribuído. Máquinas: araguaia.ic.unicamp.br e iguacu.ic.unicamp.br. Sem processamento 115
Tabela 22: Máquinas utilizadas durante os testes da arquitetura WONDER..........................165

Lista de Gráficos

Gráfico 1: Comparação entre os tempos médios das atividades em execução centralizada e distribuída. Máquinas araguaia, iguacu, anhumas e tigre. 110
Gráfico 2: Comparação entre os tempos médios das atividades em execução centralizada e distribuída. Percentuais relativos. Máquinas araguaia, iguacu, anhumas e tigre. 110
Gráfico 3: Tempo médio de execução do caso X Número de casos concorrentes. 1 a 20 casos em ambiente centralizado. Máquina araguaia 113
Gráfico 4: Tempo médio de seqüenciamento do caso X Número de casos concorrentes. 1 a 20 casos em ambiente centralizado. Máquina araguaia 114
Gráfico 5: Tempo médio de execução do wrapper X Número de casos concorrentes. 1 a 20 casos em ambiente centralizado. 114
Gráfico 6: Tempo médio de execução dos casos X Número de casos concorrentes. Execução de 1 a 20 casos concorrentes em ambiente distribuído: nós araguaia e iguacu. Sem Processamento 116
Gráfico 7: Tempo médio de seqüenciamento dos casos X Número de casos concorrentes. Execução de 1 a 20 casos concorrentes em ambiente distribuído: nós araguaia e iguacu. Sem processamento 116
Gráfico 8: Tempo médio de execução dos wrappers X Número de casos concorrentes. Execução de 1 a 20 casos concorrentes em ambiente distribuído: nós araguaia e iguacu. Sem processamento 117
Gráfico 9: Tempo médio de execução dos casos X Número de casos concorrentes. Comparação de execuções de 1 a 20 casos concorrentes em ambientes distribuído e centralizado. 118
Gráfico 10: Tempo médio de execução dos casos X Número de casos concorrentes. Comparação de execuções de 1 a 20 casos concorrentes em ambientes distribuído e centralizado 118
Gráfico 11: Tempo médio de seqüenciamento do caso X Número de casos concorrentes. Comparação de execuções de 1 a 20 casos concorrentes em ambientes distribuído e centralizado. 119
Gráfico 12: Tempo médio de execução dos wrappers X Número de casos concorrentes. Comparação de execuções de 1 a 20 casos concorrentes em ambientes distribuído e centralizado. 119
Gráfico 13: Tempo médio de execução dos casos X Número de casos concorrentes. Comparação das execuções de 1 a 20 casos concorrentes, em 2 ambientes distribuídos e 1 centralizado, executando atividades de ordenação de 1000 números. 121
Gráfico 14: Tempo médio de seqüenciamento dos casos X Número de casos concorrentes. Comparação das execuções de 1 a 20 casos concorrentes, em 2 ambientes distribuídos e 1 centralizado, executando atividades de ordenação de 1000 números. 121
Gráfico 15: Tempo médio de execução dos wrappers X Número de casos concorrentes. Comparação das execuções de 1 a 20 casos concorrentes, em 2 ambientes distribuídos e 1 centralizado, executando atividades de ordenação de 1000 números. 122

Gráfico 16: Tempo médio de execução dos casos X Volume de dados trocado entre as atividades do caso. Execuções sucessivas, com incremento de 3 unidades de dados a partir da segunda iteração. Comparação dos cenários distribuído e centralizado......................124
Gráfico 17: Tempo médio de seqüenciamento dos casos X Volume de dados trocado entre as atividades do caso. Execuções sucessivas, com incremento de 3 unidades de dados a partir da segunda iteração. Comparação dos cenários distribuído e centralizado......................124
Gráfico 18: Tempo médio de execução dos wrappers X Volume de dados trocado entre as atividades do caso. Execuções sucessivas, com incremento de 3 unidades de dados a partir da segunda iteração. Comparação dos cenários distribuído e centralizado......................125

Capítulo 1

Introdução

1.1 Motivação

O Gerenciamento de Workflow (*Workflow Management*) é uma das áreas que, nos últimos anos, tem atraído a atenção de muitos pesquisadores, desenvolvedores e usuários. Conceitos como Trabalho Cooperativo Suportado por Computador (*Computer Supported Cooperative Work* - CSCW), escritórios sem papel, processamento de formulários e automação de escritório, tiveram sua implementação adiada por décadas, principalmente por falta de tecnologia e *know-how*. Essa tecnologia foi finalmente provida através de avanços nas áreas de redes de computadores e sistemas distribuídos, com surgimento de computadores cada vez mais rápidos e baratos, além de ter sido impulsionada por novas técnicas de reestruturação empresarial como a reengenharia. Enquanto esses conceitos tornavam-se realidade, a demanda por soluções capazes de integrar os diversos recursos de informação de uma empresa tornava-se cada vez maior. Os sistemas de informação de uma corporação moderna, em sua maioria, são compostos por um conjunto fracamente acoplado, heterogêneo e distribuído de ambientes computacionais. A descentralização das corporações e do processo de tomada de decisões, a necessidade de dados detalhados diários sobre o andamento das diversas atividades empresariais, assim como a ênfase em arquiteturas cliente/servidor e a crescente disponibilidade de tecnologias de processamento distribuído (a exemplo de WWW, CORBA, ActiveX e Java) constituem um conjunto de tendências que indicam o fim dos dias de processamento de informação monolítico e centralizado.

De maneira a concretizar este conjunto de mudanças, surge a necessidade de formas de implementar e integrar ambientes de execução heterogêneos, onde conjuntos de tarefas interrelacionadas possam ser desempenhadas de forma eficiente e segura, permitindo um acompanhamento detalhado de sua execução. É nesse contexto que sistemas de Gerenciamento de Workflow de larga escala são empregados.

Sistemas de Gerenciamento de Workflow (SGWFs) são usados para coordenar e seqüenciar processos empresariais (*business processes*). Exemplos típicos de tais processos são: aprovação de empréstimos, processamento de propostas de seguro e pagamento de contas. Tais processos são representados por *Workflows*, ou seja, modelos computadorizados de processos empresariais que especificam todos os parâmetros envolvidos na sua realização. Esses parâmetros envolvem a definição de passos (atividades) individuais (como entrada de dados do comprador, consultar banco de dados ou verificar uma assinatura), ordens e condições sob as quais os passos devem ser executados, incluindo aspectos como fluxo de dados entre as atividades e o desígnio de pessoas ou processos responsáveis pela execução de cada tarefa. Assim como aplicações (planilhas eletrônicas, processadores de texto, ferramentas de CAD/CAM, bancos de dados) que irão auxiliar no desempenho de cada atividade.

1.1.1 Principais Limitações dos Sistemas Existentes

O estado da arte dos Sistemas de Workflow tem sido determinado, até o momento, por funcionalidades providas em sistemas comerciais. Inicialmente concebidos para coordenar atividades cooperativas, SGWFs foram desenvolvidos tendo em vista o seu uso por pequenos grupos de usuários. Com a atual introdução desses sistemas nas corporações e seu conseqüente uso em ambientes e tarefas maiores e mais complexos, as limitações e restrições iniciais desses sistemas tornaram-se mais evidentes. Suas deficiências quanto a arquitetura (base de dados centralizada, sistema de comunicação ineficiente, incapacidade de representar processos complexos e heterogêneos) tornaram-se um limitante ao seu avanço no campo corporativo, o que em muitas aplicações, como as envolvendo sistemas bancários e de grandes empresas, mostra-se insuficiente.

Uma das principais deficiências dos sistemas atuais é sua inadequação no suporte a aplicações de larga escala. Por serem construídos, em sua maioria, sobre uma base central de dados que, em muitos casos, é também utilizada para armazenar informações de controle interno do SGWF (estado dos processos em execução, históricos, binários das aplicações), muitos dos sistemas atuais possuem gargalos de desempenho que limitam seus aspectos de escala e concorrência. Adicionalmente, a utilização de um banco de dados central representa um único ponto de falha que, em grande parte dos casos, abre brechas para que uma falha paralise todo o sistema, tornando-o, dessa forma, indisponível por um intervalo de tempo muitas vezes inaceitável em aplicações de grande porte. Dessa forma, os produtos atuais pecam em prover a flexibilidade, escala e tolerância a falhas necessárias aos requisitos de desempenho e disponibilidade de um grande número de aplicações.

SGWFs de larga escala são sistemas capazes de suportar a execução simultânea de um grande número de casos e atividades concorrentes, envolvendo um grande número de atores e programas tipicamente dispersos em um sistema distribuído. São alguns exemplos de aplicações que utilizam workflows de larga escala: sistemas bancários, de reserva de passagens aéreas, aplicações que automatizam processos governamentais e outros.

O conceito de "grande" é relativo e varia com o desenvolvimento tecnológico. Assume-se, neste trabalho, que um SGWF de larga escala é aquele que, em configurações centralizadas tradicionais, normalmente necessitaria de recursos computacionais com capacidade de processamento e banda passante de rede acima da média. Em geral, são sistemas que demandam o uso de supercomputadores, como *mainframes*, e conexões de rede de alta velocidade, utilizando recursos do estado da arte corrente.

1.2 Objetivos

O presente trabalho tem como objetivo o desenvolvimento de uma arquitetura de software para um SGWF que satisfaça, primeiramente, os requisitos de escalabilidade (suporte a à execução de um grande volume de instâncias de processos concorrentes com a menor degradação de desempenho possível) e disponibilidade (capacidade de prover um determinado serviço quando este é requerido) demandados por ambientes de execução de workflows de larga escala. Os outros requisitos de SGWFs tradicionais e de larga escala (apresentados nas sessões 2.8 e 2.9.2) devem também ser atendidos.

1.2.1 Hipótese

O presente trabalho utiliza a hipótese de que o problema de escalabilidade de SGWFs pode ser atacado através da distribuição de controle e processamento em entidades autônomas, que executam por vários nós de um sistema distribuído. Esta autonomia é provida pelo transporte dos dados e do plano do caso junto com estas entidades. O rompimento com o paradigma centralizado dos SGWFs tradicionais deve prover, desta forma, a escalabilidade, disponibilidade e tolerância a falhas requeridas por workflows de larga escala.

1.2.2 Estratégia

De maneira a testar a hipótese anterior, foi desenvolvida e implementada uma arquitetura distribuída para a execução simultânea de várias instâncias de um grande número de processos empresariais. Esta arquitetura utilizou o conceito de casos móveis autônomos, que carregam consigo os dados e a informação de controle do workflow. A implementação utilizou os recursos de distribuição do *framework* de comunicação CORBA (*Common Object Request Broker Architecture*). A arquitetura proposta utiliza uma abordagem radical, onde o controle e a execução do workflow são realizados de forma completamente descentralizada.

Uma vez definida e implementada a arquitetura, esta foi testada em configurações totalmente centralizadas e distribuídas, tendo seus resultados comparados, de maneira a estudar, comparar e determinar o comportamento do sistema em diversos cenários, em especial em cenários distribuídos.

1.3 Estrutura do Trabalho

A dissertação está organizada de acordo com a estrutura de capítulos a seguir:

Capítulo 2. Apresenta o conceito de workflow, suas origens, requisitos e problemas. Ao seu final são discutidos os requisitos de SGWFs de larga escala.

Capítulo 3. Apresenta os conceitos fundamentais da arquitetura CORBA e do paradigma de Agentes Móveis necessários à compreensão deste trabalho.

Capítulo 4. Descreve a arquitetura WONDER, seus componentes e inter-relacionamentos, apresentando suas características e decisões de projeto frente aos requisitos dos SGWFs de larga escala.

Capítulo 5. Discute o mapeamento dos elementos da arquitetura WONDER para o *framework* de comunicação CORBA, implementado em Java.

Capítulo 6. Realiza um paralelo entre a arquitetura WONDER e o paradigma de agentes móveis.

Capítulo 7. Descreve aspectos relacionados à implementação do protótipo desenvolvido para a realização dos testes.

Capítulo 8. Descreve os testes realizados com o protótipo da arquitetura, discutindo os resultados obtidos.

Capítulo 9. Apresenta alguns trabalhos relacionados, destacando e comparando suas soluções para os requisitos de SGWFs de larga escala.

Capítulo 10. Apresenta algumas conclusões e discute alguns trabalhos futuros.

Capítulo 2

Workflow

Neste capítulo, é introduzido o conceito de Workflow e Sistemas de Gerenciamento de Workflow. É apresentado um breve histórico desses sistemas, descrevendo sua evolução, suas principais características e aplicações. Ao final, é apresentado o conceito de Workflow de Larga Escala, o domínio do problema abordado nesta dissertação, juntamente com seus requisitos.

2.1 Histórico

Workflows são modelos computadorizados de processos empresariais que especificam todos os parâmetros envolvidos em sua execução. O termo gerenciamento de workflow é anterior aos atuais sistemas de gerenciamento de workflow (SGWFs). Este termo refere-se ao domínio de aplicação que envolve logística e processos empresariais. Esta área de estudo é também conhecida como logística de escritórios. O principal objetivo dos SGWFs é assegurar que atividades apropriadas sejam executadas pela(s) pessoa(s) certa(s), no tempo correto. Apesar de ser possível realizar gerenciamento de workflow sem o uso de um sistema de gerenciamento de workflow, a maioria das pessoas associa este conceito com tais sistemas. A *Workflow Management Coalition* (WfMC) [WfMC-TC00-1003] define "sistemas de gerenciamento de workflow" como sendo: "Sistemas que definem, executam e gerenciam completamente workflows através da execução de um software cuja ordem de execução é dirigida por uma representação lógica e computadorizada de um workflow". Outros sinônimos de SGWFs são: 'sistemas operacionais de negócios' (*business operating system*), 'gerenciadores de workflow', 'gerenciadores de casos' e 'sistemas de controle de logística'. Descreveremos a seguir um breve histórico de tais sistemas. Uma definição mais precisa pode ser encontrada ao final desta seção.

O advento de Sistemas de Gerenciamento de Workflow (SGWFs) foi adiado durante vários anos devido, principalmente, à falta de tecnologia que permitisse o seu desenvolvimento. No

final da década de 80 e início de 90, vários fatores permitiram sua adoção e expansão. Listaremos a seguir alguns destes fatores.

Segundo Jablonski e Bussler [JB96] duas tendências fundamentais contribuíram para o advento da tecnologia de sistemas de gerenciamento de workflow. O primeiro foi o enorme avanço tecnológico ocorrido nas últimas décadas; e o segundo foi caracterizado pela mudança do objetivo central no desenvolvimento de sistemas computacionais. Este deixou de ser o desenvolvimento de programas que automatizam pequenas tarefas corporativas, passando a objetivar o desenvolvimento de soluções integradas completas. Discutiremos a seguir, em mais detalhe, estes dois fatores.

O progresso tecnológico foi basicamente representado pelos constantes avanços nas áreas de hardware e software. Na área de hardware, podemos destacar o grande desenvolvimento da microeletrônica, com o crescente aumento da capacidade de processamento dos computadores. Tal crescimento pode ser descrito pela lei de Moore. Esta preconiza que a capacidade de processamento dos computadores deve dobrar a cada 18 meses. Nesta mesma direção, a lei de Hogland prevê o aumento multiplicativo, por um fator de 10, da capacidade de armazenamento de discos e fitas magnéticos a cada década. Tais avanços possibilitaram a migração da capacidade computacional dos antigos *mainframes*, alojados em obscuros CPDs, para as mesas de trabalho das corporações. Desta forma, as capacidades computacionais e de armazenamento de dados que antes eram caros, restritos e portanto centralizados, tornam-se cada vez mais baratos e ubíquos, alcançando não somente os locais de trabalho mas, de maneira crescente, todos os vários aspectos de nosso dia-a-dia. São exemplos destes avanços, os computadores pessoais, os telefones celulares, os *notebooks*, os computadores portáteis (*hand-held PCs*), os consoles para *WebTVs* e muitos outros aparelhos.

Nas últimas décadas, juntamente com o avanço da tecnologia dos microcomputadores, observou-se também um grande crescimento da conectividade deste aparato computacional. Tal crescimento pode ser evidenciado pelos avanços na tecnologia de redes de computadores, abrangendo não somente a velocidade de transmissão de dados, mas também sua confiabilidade e seu alcance geográfico, seja por meio físico ou por novas aplicações das redes sem fio. Tais fatos culminaram no atual fenômeno da popularização da Internet e das redes locais corporativas.

Outra conseqüência dos avanços alcançados nos campos de hardware e redes de computadores foi a “revolução do software”. O aumento da conectividade dos sistemas de hardware foi pré-requisito básico para o desenvolvimento da conectividade dos sistemas de software. Tal fato pode ser evidenciado pelo atual desenvolvimento de ambientes computacionais integrados tanto no âmbito local como no âmbito global. Tal avanço possibilitou o desenvolvimento das atuais aplicações de *Groupware* [EGR91]. Enquanto que, no início da história da computação, os computadores eram utilizados apenas como potentes máquinas de calcular, o que aliás originou o nome “computador”, ou máquina de computar; nos dias atuais, a computação tem sido empregada nas em um crescente leque de aplicações, em especial, nas áreas de telecomunicações e comunicação de dados. Esta expansão transformou o computador em uma poderosa ferramenta de comunicação e colaboração.

Este conjunto de mudanças fez com que o desenvolvimento de software modificasse sua perspectiva e abordagem. Se, no passado, o desenvolvimento de sistemas utilizava uma abordagem orientada a tarefas ou a dados, objetivando o desenvolvimento de aplicações de escritório, tipicamente independentes e isoladas entre si, destinadas a aplicações específicas; no presente, busca-se o desenvolvimento de sistemas que permitam modelar, controlar e espelhar os processos corporativos. O trabalho, sua execução e modelagem passaram a ser o centro da questão.

2.2 Origens

Após a apresentação de um breve histórico e pré-requisitos que permitiram o desenvolvimento dos SGWFs atuais, serão relacionados alguns "ancestrais conceituais" dos SGWFs. São descritos, a seguir, algumas dentre as várias tecnologias de software que, segundo Jablonski e Bussler [JB96], contribuíram de várias maneiras para o desenvolvimento destes sistemas. São estas tecnologias:

- Automação de escritórios
- Gerenciamento de Bancos de Dados
- Modelos de Transações Avançadas
- E-mail
- Gerenciamento de Documentos
- Gerenciamento de Desenvolvimento de Software
- Modelagem de processos e arquiteturas empresariais
- Aplicações de *Groupware*

São descritos, a seguir, de forma resumida, cada uma destas tecnologias, destacando suas principais contribuições aos SGWFs.

2.2.1 Automação de Escritórios

A tecnologia de automação de escritórios é considerada um dos principais "ancestrais" dos sistemas de gerenciamento de workflow. Esta tecnologia permitiu a incorporação de requisitos como agendamento de atividades, integração de funções, sistemas de informações pessoais e gerenciamento de tarefas. Contudo, este tipo de aplicação está intimamente relacionado à automação de tarefas individuais. Gerenciamento de workflow deve ser diferenciado de automação de workflow. Este último visa execução automática de tarefas e aplicações individuais, enquanto que o primeiro está relacionado à coordenação e controle de processos envolvidos em uma organização. Desta forma, é o controle, e não a realização da tarefa, o objetivo central dos SGWFs.

2.2.2 Gerenciamento de Bancos de Dados

Vários SGWFs têm sua origem em pesquisas realizadas na área de Sistemas de Gerenciamento de Bancos de Dados (SGBDs). Bancos de dados convencionais são passivos, ou seja, apenas armazenam informações de forma estruturada. Esta informação é recuperada e atualizada através de operações de consultas e atualizações. Estas operações precisam ser invocadas explicitamente e duram, normalmente, alguns segundos. Operações sobre um banco de dados geralmente ocorrem na forma de transações. Uma transação é uma unidade independente de computação. Transações atendem aos requisitos conhecidos como ACID, significando Atomicidade, Consistência, Independência e Durabilidade. Estes requisitos são normalmente viabilizados através do emprego de técnicas bem conhecidas como o uso de históricos (*logs*), *timestamps* e *locks* de escrita e leitura. Estas técnicas são empregadas em operações de fazer/desfazer (*redo*/*undo*), implementadas como parte do protocolo de transações em duas fases (*two-phase commit*).

Aplicações mais complexas podem envolver a execução ordenada de transações cujas execuções dependem de eventos do sistema ou modificações ocorridas no banco de dados. Surge então o conceito de Bancos de Dados ativos, cuja principal idéia é enriquecer um banco de dados convencional como o conceito de regras ECA (Evento Condição Ação). Toda vez que um evento ocorre, envolvendo num determinado conjunto de dados, um conjunto de condições associadas a estes dados é avaliado. O resultado desta avaliação dispara um conjunto de ações realizadas neste banco de dados. Regras ECA são, normalmente, armazenadas no próprio banco de dados. Estas regras são empregadas no controle das transações de longa duração, que são agora implementadas como um conjunto de sub-transações de menor duração, casualmente relacionadas. Estas regras são também utilizadas na implementação de vários SGWFs atuais.

2.2.3 Modelos de Transações Avançadas

Requisitos de atomicidade (tudo ou nada) não são desejáveis em transações que podem permanecer por horas ou até dias em execução. A recuperação de uma falha, nestes casos, implica em um custo alto: uma falha não pode resultar em desfazer tudo o que foi feito durante horas de trabalho. Frente a esses novos requisitos, extensões e relaxamentos do modelo ACID tradicional tornaram-se necessários [Elmargamid92; GHKM94].

Várias pesquisas foram e vêm sendo realizadas no desenvolvimento de modelos de transações de longa duração. Estas pesquisas deram origem a modelos de transações estendidas. São exemplos destes modelos: Sagas, Transações aninhadas e os Workflows Transacionais [Elmargamid92; WR93; RS95; SR93].

Workflows transacionais permitem representar transações complexas, como um conjunto de várias tarefas (sub-transações), ordenadas através do uso de estruturas de controle como laços (*loops*) e comandos condicionais. O tratamento de falhas destas transações envolve regras de

compensação e operações de desfazer semânticas, que são associadas a cada tarefa, de forma a compor o Workflow Transacional. Uma tarefa é um conjunto de estados de execução, um conjunto de transições entre estados válidos e um conjunto de condições que disparam a execução desta tarefa.

Workflows transacionais são amplamente dependentes de bancos de dados e destinam-se a aplicações que manipulam de forma complexa e estruturada estes dados [MAAEGK95]. Estes sistemas diferem, contudo, das aplicações de gerenciamento de workflow. Seu principal objetivo é estruturar transações complexas de maneira a garantir sua execução de modo transacional, através do tratamento adequado de suas falhas, e do uso de pré e pós condições de controle. É por este motivo que workflows transacionais não são classificados como uma abordagem para o problema de gerenciamento de workflow, mas como uma abordagem à estruturação e execução de aplicações (tarefas). Estes sistemas podem ser, portanto, empregados no contexto maior de gerenciamento de workflow.

São alguns exemplos de sistemas de workflow transacionais o sistema Contracts [RS95; Schwenkreis93], da Universidade de Stuttgard, na Alemanha, e o FlowMark [MAGKR95; MAAEGK95] da IBM.

SGWFs baseados em modelos transacionais são, em geral, pouco flexíveis, sendo empregados em processos muito bem definidos e, em sua maioria, compostos por atividades automáticas, com pouca interação com o usuário final, a exemplo de processos bancários e de reservas de passagens aéreas. O controle de execução destes processos requer monitoramento constante o que é, normalmente, realizado de maneira centralizada, com o auxílio de monitores de transação (TP-Monitors).

2.2.4 E-Mail

A idéia de podermos enviar uma mensagem para várias pessoas, assim como a capacidade de anexar arquivos, figuras, textos e formulários HTML em uma mensagem fornece um sistema básico de roteamento de informação. Estes mecanismos estão presentes nos SGWFs atuais. Em especial, em sistemas como o Teamroute da DEC (*Digital Equipment Corporation*), que permitia incluir informações de roteamento em uma mensagem, de maneira que esta seguisse seqüencialmente pela lista de usuários fornecida em seu cabeçalho [DEC1992]. A idéia de migração de dados através dos nós da rede também é utilizada na arquitetura proposta neste trabalho.

2.2.5 Gerenciamento de Documentos

A penetração da informática no meio corporativo levou a substituição de documentos em papel, por documentos eletrônicos. Sistemas de Gerenciamento de Documentos são empregados na organização, indexação e recuperação destes dados eletrônicos. Os primeiros sistemas eram

passivos, isto é, apenas permitiam organizar, unir, classificar e consultar documentos sob o comando dos usuários. Estes sistemas foram substituídos por sistemas ativos, que incorporavam funções de gerenciamento do ciclo de vida dos documentos. Por exemplo, *triggers* podem ser configurados de forma a enviarem um determinado documento para revisão antes de um determinado prazo. Estes sistemas deram origem aos SGWFs centrados em documentos que serão discutidos na próxima seção.

2.2.6 Processo de Software

O desenvolvimento de software costuma seguir uma metodologia específica, que define várias etapas a serem cumpridas, do início ao fim do projeto. O conjunto destas etapas é conhecido como processo de software (*software process*). Este processo precisa ser previamente analisado e testado, de maneira a determinar erros e inconsistências, antes que este seja usado para reger o desenvolvimento de sistemas. Os processos atuais de desenvolvimento costumam envolver um grande número de programadores, equipes de teste e engenheiros de software. A necessidade de coordenar todos estes esforços, de maneira eficiente, promoveu o desenvolvimento de várias técnicas e conceitos que hoje são utilizados na área de gerenciamento de workflow. A criação de um processo de desenvolvimento de software engloba três grandes fases:

1. O modelo de processo é desenvolvido;
2. O processo proposto precisa ser analisado de forma a detectar erros e inconsistências antes que o modelo seja utilizado para guiar o desenvolvimento de software;
3. O modelo é utilizado (executado), ou seja, um software é desenvolvido usando as regras e recomendações deste modelo.

Durante estas fases, a coordenação do trabalho em grupo e o gerenciamento das etapas do projeto constituem um fator essencial ao adequado desenvolvimento do modelo.

2.2.7 Modelagem de Processos Empresariais e Modelagem de Arquitetura de Empresas

Nos últimos anos, conceitos como reestruturação empresarial, reengenharia e horizontalização de empresas foram amplamente empregados e utilizados, em especial nas grandes empresas. Estas metodologias compreendem o planejamento e modelagem das corporações como um conjunto de processos. Tal abordagem visa modelar uma empresa com o intuito de compreender, otimizar e reestruturar seus processos, tanto no âmbito da produção como em seus vários setores a exemplo do financeiro, logístico e pessoal. Estes modelos podem ser reestudados, executados, testados, e novamente iterados. Esta metodologia, juntamente com as metodologias empregadas na confecção de seus modelos e na sua documentação, podem ser facilmente aplicados no desenvolvimento de SGWFs que irão automatizar estes processos.

2.2.8 Aplicações de *Groupware*

A indústria de *Groupware* tem introduzido uma grande gama de aplicações projetadas para suportar e facilitar a interação, cooperação e o trabalho de grupos de pessoas. Como o escopo de tais processos se estendeu ao ambiente empresarial, tais aplicações passaram a necessitar de regras mais formais de coordenação da cooperação. Workflow provê um ambiente que permite o controle e a modelagem de tais interações.

2.3 Aplicações

Podemos listar vários cenários onde SGWFs são empregados. São algumas destas aplicações: processos de desenvolvimento de software, pedidos de financiamento de casa própria, controle de manufatura, controle de empréstimos, automação de cartórios e de fóruns de justiça, acompanhamento de pacientes em hospitais, automação de processos logísticos, planejamento da produção, e muitos mais.

Na Figura 1, é descrito um exemplo de workflow usado pelo OMG (*Object Management Group*) para aprovar novas RFPs (*Request for Proposals*) para seus padrões.

Este workflow descreve o trabalho envolvendo os grupos: Times de Revisão, Força Tarefas (*Task Forces*) e Força Tarefas de Revisão (*RTF – Revision Task Forces*) [Kobryn99]. Retângulos em cinza destacam o fluxo de documentos entre atividades. Textos entre colchetes indicam condições de controle. As tarefas realizadas pelos times de Submissão o RFP e o RTF são agrupadas em colunas separadas por linhas tracejadas. dentro de caixas tracejadas.

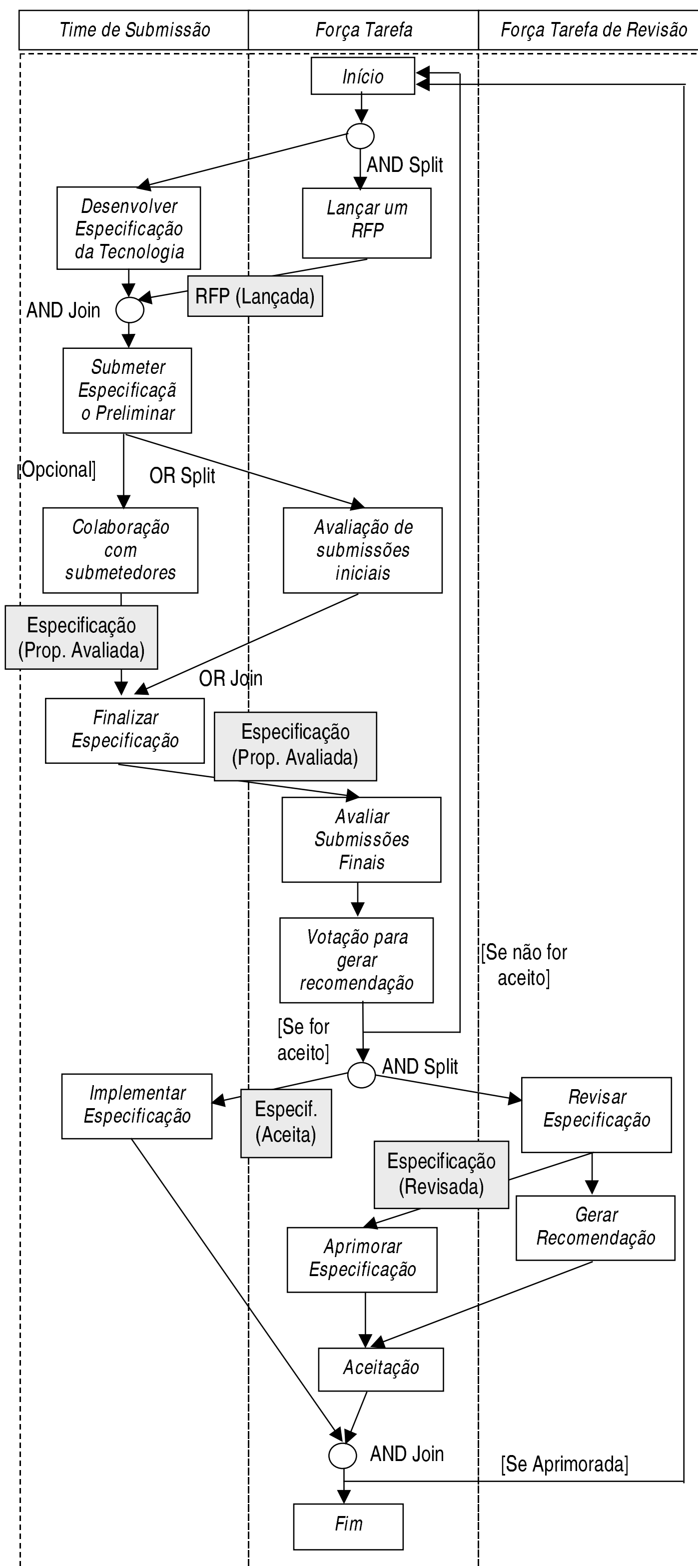

Figura 1: Processo de Revisão de Padrões do OMG

O processo de uma RFP, descrito na Figura 1, é o mecanismo primário de aceitação de novas especificações e de aprimoramento de especificações existentes no OMG. Este processo envolve a confecção e revisão de vários documentos. A força tarefa de um determinado domínio de aplicações lança uma RFP inicial. Um ou mais times de submissão respondem com propostas iniciais (*drafts*) para esta RFP. A força tarefa designada para este padrão avalia estas propostas iniciais, provendo pareceres para os autores que são encorajados a debaterem entre si antes de gerarem a(s) proposta(s) final(is). Esta(s) proposta(s) é(são) levada(s) para votação pela força tarefa após ser(em) avaliada(s). Uma proposta é selecionada e encaminhada para a Banca de Arquitetura e a Força Tarefa de Tecnologias para sua aprovação.

Se uma proposta final recebe todas as aprovações requeridas, esta torna-se uma tecnologia adotada pelo OMG. De outra maneira, a Força Tarefa tem a opção de re-submeter a RFP com alterações que idealmente refletem lições aprendidas. Logo após a adoção de uma especificação, uma Força Tarefa de Revisão é formada para revisar a especificação e recomendar alterações.

2.4 Definição de Sistemas de Gerenciamento de Workflow

Após esta breve introdução definiremos mais precisamente o conceito de Sistemas de Gerenciamento de Workflow. Fornecemos duas definições, uma do ponto de vista empresarial e outra do ponto de vista do sistema:

Do ponto de vista *empresarial*, **Sistemas de Gerenciamento de Workflow** (SGWFs) são usados para coordenar e seqüenciar processos empresariais (*business processes*). Tais processos são representados por **workflows**. Esses parâmetros envolvem a definição de passos (atividades) individuais (como entrada de dados do comprador, consultar banco de dados ou verificar uma assinatura), o estabelecimento de uma determinada ordem de condições sob as quais os passos devem ser executados, incluindo aspectos como fluxo de dados entre as atividades, o desígnio de pessoas ou processos responsáveis por cada tarefa e aplicações que irão auxiliar no desempenho de cada atividade.

Do ponto de vista de *projeto e implementação*, Sistemas de Gerenciamento de Workflow definem um conjunto de interfaces para usuários e aplicações, através de APIs (*Application Programming Interfaces*) envolvidos nos processos de Workflow. Um SGWF é, dessa forma, um conjunto de ferramentas e aplicações de controle usadas para projetar, definir, executar e monitorar processos empresariais.

2.5 Classificação

Existem várias classificações de SGWF, dentre elas selecionamos duas classificações que explicaremos a seguir.

Segundo Alonso et. al [AAEM97], SGWFs podem ser agrupados, de acordo com a *estrutura e complexidade dos processos envolvidos* em:

Administrativos. Estes SGWFs são usados na automação de processos onde regras são bem definidas e conhecidas por todos os atores do processo. São exemplos de Workflows Administrativos: processos de matrícula em universidades, registro de veículos, e várias outras atividades que normalmente envolvem um conjunto de formulários que são roteados por diversas atividades, representando um processo burocrático bem conhecido e que não costuma variar com o passar do tempo.

Ad Hoc (Excepcionais ou Adaptativos). Workflows são, normalmente, utilizados na modelagem de processos bem conhecidos e estabelecidos, que não mudam freqüentemente. Workflows *ad-hoc* são similares a Workflows Administrativos possuindo, contudo, características que permitam o tratamento de exceções ou situações únicas, não rotineiras. Um exemplo são os processos de submissão de artigos em revistas, tais processos são normalmente demorados e seguem protocolos diferentes, que dependem da revista em questão. O número de revisões e autores varia de acordo com cada caso.

De Colaboração. Constitui uma classe de SGWF que permite modelar e coordenar processos de colaboração envolvendo vários participantes. Um exemplo deste tipo de processo é a escrita colaborativa de artigos. Ao contrário dos outros tipos de workflows, onde há sempre um seqüenciamento de atividades, um workflow de colaboração pode conter/modelar várias iterações em uma mesma atividade até que um consenso seja estabelecido, assim como pode requerer a retomada de atividades anteriores. SGWFs que coordenam a execução de workflows de colaboração tendem a ser dinâmicos de forma que várias atividades são determinadas durante o transcorrer do processo.

Workflows de colaboração formam a classe que mais se distancia do conceito clássico de workflow, sendo questionável sua classificação como tal visto que, em muitos casos, a coordenação é realizada pelos próprios participantes, já que o plano (definição de processo) em execução precisa ser, via de regra, extremamente flexível ou mesmo inexistente. Tais sistemas restringem-se, normalmente, a apenas fornecer uma interface que permita registrar as decisões tomadas, tipicamente semelhante a interfaces de correio eletrônico.

De Produção. Correspondem a SGWFs Administrativos que satisfazem requisitos de escalabilidade, complexidade e heterogeneidade de ambiente, assim como variedade de atores (pessoas/processos), organizações e tarefas. Tendem a ser executados em grandes corporações, envolvendo ambientes e aplicativos heterogêneos, muitos deles legados. Estes SGWFs são normal-

mente empregados em processos de missão crítica de uma empresa como controle de pedidos de compra e venda.

Em uma outra classificação, Jablonski e Bussler [JB96] agrupam SGWFs de acordo com a *tecnologia e estrutura empregadas* em:

Centrados em E-mail. São SGWFs baseados em sistemas de correio eletrônico. Podem ser associados a workflows de Colaboração e Ad Hoc. Por utilizarem correio eletrônico, não são adequados à execução de Workflows de Produção, ou que pressuponham a execução de um grande número de processos concorrentes.

Centrados em documentos. Baseiam-se na troca de documentos entre as atividades. Possuem pouca habilidade de integração como outras aplicações. Vários SGWFs Administrativos baseados em formulários podem ser implementados dessa forma.

Centrados em processos. Correspondem a SGWFs de Produção. Geralmente implementam seu próprio sistema de comunicação, são geralmente construídos utilizando Sistemas de Gerenciamento de Bancos de Dados (SGBDs), provendo interfaces de comunicação com sistemas heterogêneos e legados.

2.6 Estrutura de um Sistema de Gerenciamento de Workflow (SGWF)

Fundada em 1993, a WfMC (*Workflow Management Coalition*) é uma organização internacional formada por vendedores, fabricantes e usuários de Sistemas de Workflow. Reúne mais de 200 membros em 25 países. Visa promover o uso de sistemas de Workflow através do estabelecimento de terminologias, descritas em [WfMC-TC1011], e padrões de software, de interoperabilidade e de conectividade entre os Sistemas de Workflow.

A seguir são descritos, de forma mais detalhada e do ponto de vista de implementação, os principais componentes de um SGWF. Esta seção segue o modelo proposto *pela Workflow Management Coalition* (WfMC) [WfMC-TC2101].

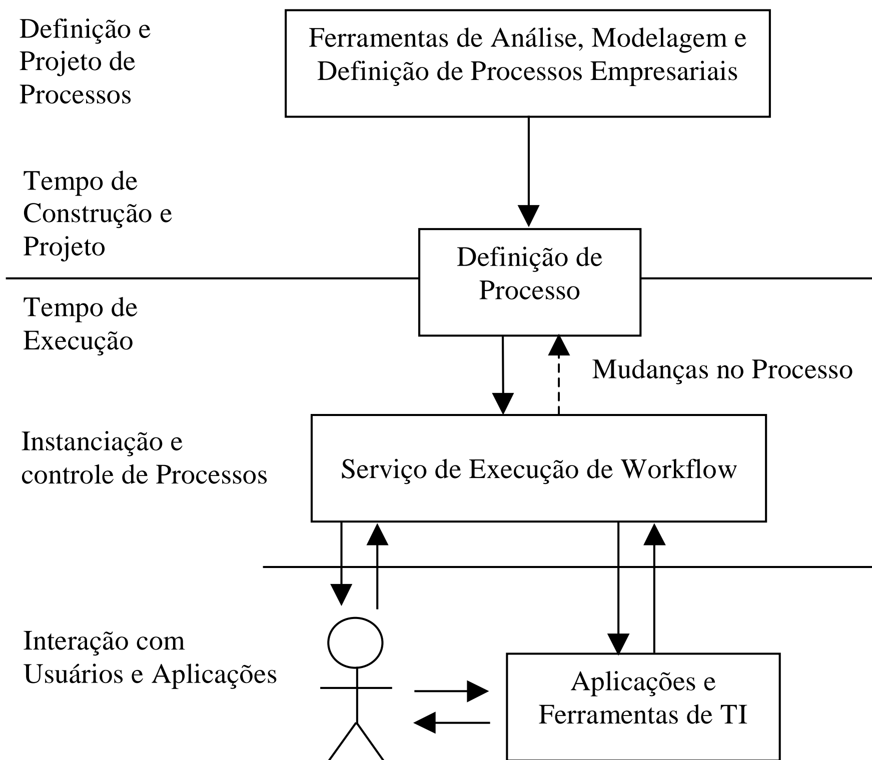

Figura 2: Características de um Sistema de Gerenciamento de Workflow

Um SGWF pode ser dividido em dois subconjuntos de ferramentas e aplicações como descrito na Figura 2 [WfMC-TC00-1003 - *Figure 1*]. São estas as ferramentas de tempo de definição (*build time*) e de tempo de execução (*run time*). Em tempo de definição, questões como análise e modelagem dos processos a serem executados são consideradas. É normalmente neste momento que são produzidas as definições de processo (ou planos). Estas definições são normalmente compiladas para uma linguagem intermediária, de maneira a permitir sua interpretação pelo núcleo do sistema de tempo de execução (*Workflow Enactment Service*). Durante o tempo de execução, o núcleo do sistema de workflow interage com aplicações externas e usuários de forma a desempenhar as atividades descritas na definição de processo fornecida.

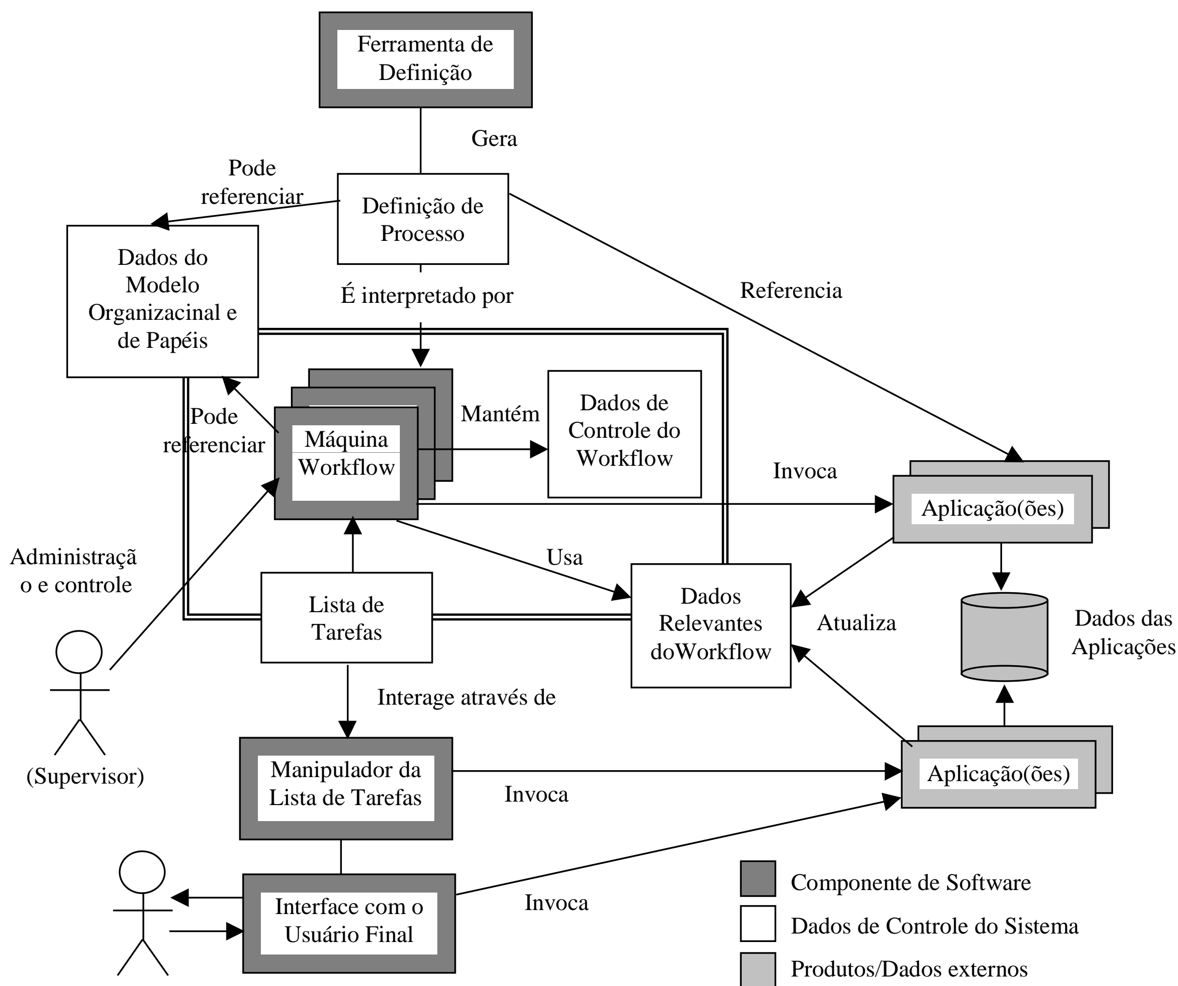

Figura 3: Estrutura Genérica de um Workflow Management System

Um modelo mais detalhado da Figura 2 é mostrado na Figura 3 [WfMC-TC00-1003 - *Figure 3*], onde podemos dividir o sistema em 3 principais tipos de componentes: componentes de software, provendo suporte a várias funcionalidades do SGWF (parte mais escura da Figura 3); vários tipos de dados de controle e definição do sistema (mostrados sem preenchimento); e aplicações e bancos de dados (com preenchimento mais claro), que não fazem parte do SGWF, mas que podem ser invocadas de forma a facilitar a realização de tarefas e atividades do sistema.

A habilidade de distribuir tarefas e informações entre seus participantes é a principal função do sistema de tempo de execução de um SGWF. Para tal, este sistema pode estar geograficamente distribuído. Esta distribuição pode ser tanto local, no caso de um grupo de trabalho local ou inter-organizacional, de forma a permitir a execução de processos envolvendo atividades e atores espalhados em mais de uma organização. Para tal, um SGWF pode utilizar várias tecnologias como correio eletrônico, troca de mensagens e, em particular tecnologias como CORBA, que fornecem um suporte à comunicação e distribuição de objetos que compõem este sistema.

2.7 Modelo de Referência da WfMC

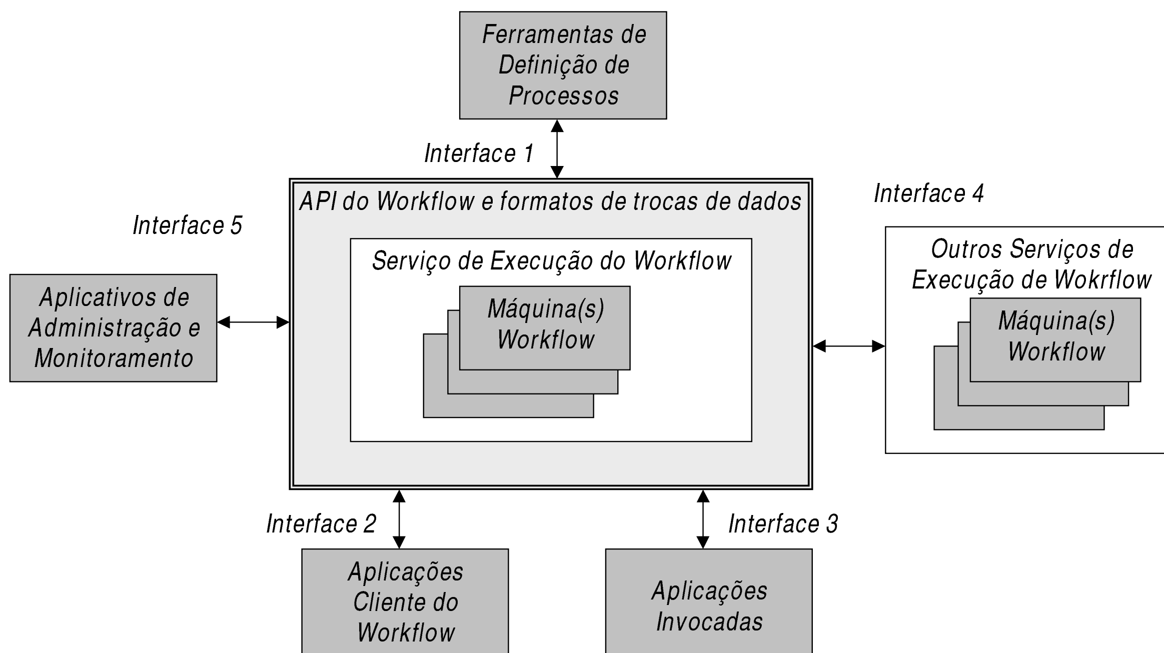

Figura 4: Modelo de Referência de Workflow - WfMC

A WfMC define um Modelo de Referência de Workflow (*Workflow Reference Model*) [WfMC-TC00-1003] que especifica uma conjunto de cinco interfaces que abrangem três áreas de funcionalidade entre o SGWF e seu ambiente (Vide Figura 4 [WfMC-TC00-1003 - *Figure 6*]): tempo de definição (*bulild time*), de controle de execução (*runtime control*) e de interação de execução *(runtime interaction*). Estas 5 interfaces são: importação e exportação de definições de processo (Interface 1); Interação com aplicações do cliente e o sistema (Interface 2); Invocação de ferramentas de software e aplicações (Interface 3); Interoperabilidade entre diferentes SGWFs (Interface 4); e Funções de administração e monitoramento (Interface 5) [WfMC-TC1009, WfMC-TC1012, WfMC-TC1016, WfMC-TC1015, WfMC-TC2101]. Esse esforço aborda principalmente questões relacionadas aos requisitos de interoperabilidade e integração entre o SGWF e aplicações existentes, padronização da terminologia empregada e das representações envolvidas na modelagem de processos. Por ser genérica, essa arquitetura não entra em detalhes de implementação do núcleo funcional de um SGWF (Quadrado central da Figura 4), especificando apenas a interface entre esse serviço de execução e os demais componentes do sistema (Parte cinza mais clara da Figura 4).

2.7.1 Terminologia

A Figura 5 a seguir, correspondente à [WfMC-TC1011 Figure 1.0], mostra os elementos da terminologia básica adotados pela WfMC e seus relacionamentos.

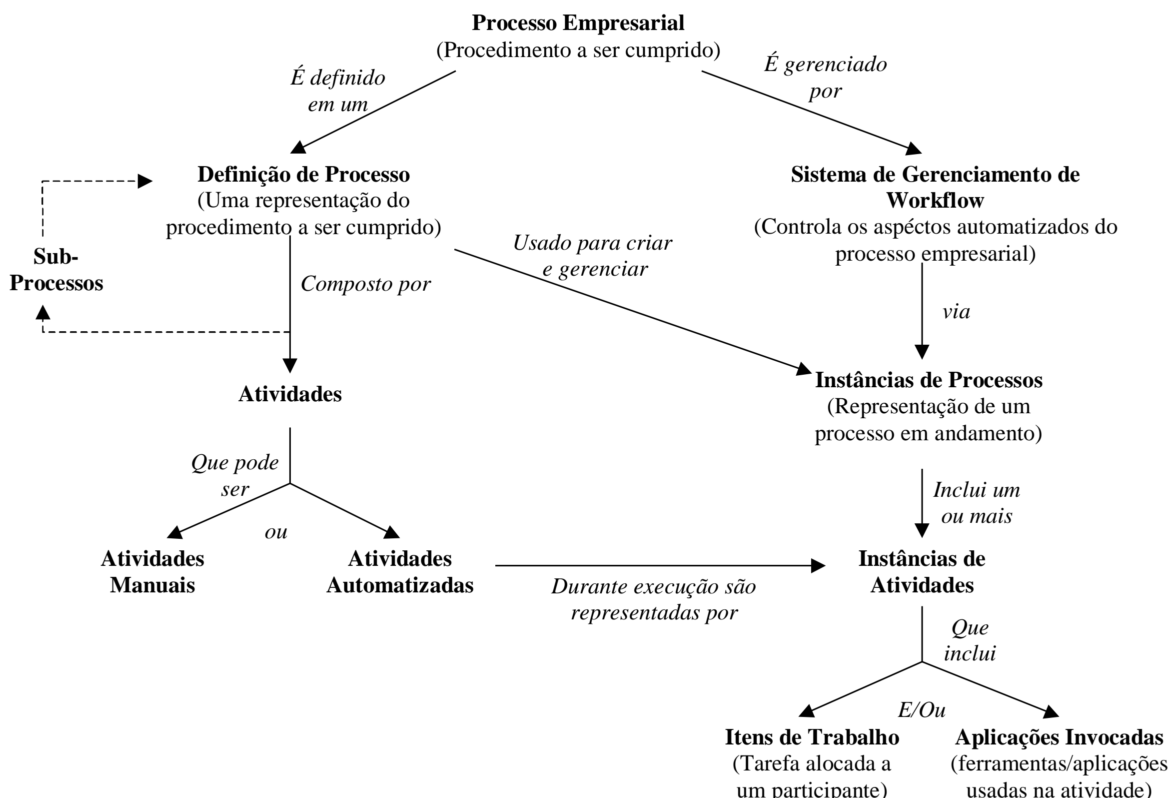

Figura 5: Relacionamento entre os elementos da terminologia básica

Um **processo** é uma descrição de uma seqüência de **tarefas** ou **atividades** que devem ser executadas para a realização de um determinado objetivo. Graficamente, um processo pode ser descrito como um grafo orientado (lido da cima para baixo no exemplo da Figura 1) onde atividades são representadas por retângulos. Atividades representam tarefas, que são atribuídas a determinados **papéis** (Funções ou cargos associados a pessoas ou programas).

Uma atividade pode ser composta por uma ou mais **(sub)tarefas**. Atividades podem ser desempenhadas de forma automática ou manual por **atores** ou **usuários.** Estes podem ser programas ou pessoas. Atividades manuais são realizadas por pessoas desempenhando determinados papéis (Secretária, Gerente, Projetista, por exemplo). Estas podem utilizar uma ou mais **aplicações invocadas** (processadores de texto, planilhas, ferramentas de CAD/CAM) no desempenho destas tarefas. Atividades automáticas são desempenhadas por programas previamente definidos. Uma atividade mais complexa pode ser representada por um processo, neste caso, chamado de **sub-processo** do processo que contém esta atividade.

Atividades consecutivas como a 'Avaliar submissões finais' e 'Votação para gerar recomendação' do exemplo da Figura 1, denotam uma relação de dependência (seta ou linha do grafo): a primeira só pode ocorrer a segunda haver terminado. Atividades 'Desenvolver especificação da tecnologia' e 'Lançar uma RFP' ocorrem de forma concorrente, independentemente uma da outra. A atividade 'Submeter especificação preliminar' depende do término de duas atividades. Tal dependência é expressa por um ***Join*** (ponto de sincronização) denotado por uma convergência de arestas. Um *Join* pode ser do tipo ***Or-Join*** onde a atividade seguinte depende do término de alguma das atividades dessa convergência, a exemplo de 'Finalizar especificação'; ou um ***And-Join***, onde o início de uma atividade, no nosso exemplo, a atividade 'Submeter especificação preliminar' é iniciado somente ao final de suas atividades anteriores. De forma análoga, o fim das atividade 'Votação para gerar recomendação' nos permite iniciar a execução em paralelo das atividades 'Revisar especificação' e 'Implementar especificação'. Essa dependência é expressa por um ***And-Split***. De maneira análoga, temos ***Or-Splits*** como 'Submeter Especificação preliminar' que especifica a escolha de uma de duas opções, no exemplo, um ramo é opcional. Variações como um *Xor* podem ainda ser também representadas.

2.8 Principais Requisitos dos SGWFs Tradicionais

De forma a coordenar e modelar atividades de longa duração do mundo real, SGWFs devem satisfazer vários requisitos. Listamos a seguir os principais requisitos destes sistemas:

Capacidade de Rotear documentos. Grande parte das operações realizadas em Sistemas de Workflow consistem do envio e recebimento de documentos entre os diferentes usuários do sistema, classicamente formulários, que são preenchidos e modificados durante diversas atividades realizadas em seu percurso.

Suporte a compartilhamento de dados. Várias aplicações cooperativas necessitam compartilhar dados e informações, sejam estes artefatos (Objetos em CAD/CAM) em aplicações colaborativas ou dados amplamente divulgados em sistemas como os bancários (cotações financeiras e taxas de juros, por exemplo).

Composição de Atividades. De forma a facilitar a reutilização e a implementação de novos processos, atividades podem ser agrupadas, compondo sub-processos que podem ser utilizados como atividades de outros processos.

Suporte a execução automática e manual de atividades. Um workflow pode ser composto por atividades completamente manuais, como o preenchimento de formulários, ou aplicações automáticas como consultar e calcular a movimentação mensal de vendas um determinado produto, em uma base de dados.

Capacidade de integração com diferentes aplicações existentes. Durante a realização de muitas tarefas, são normalmente utilizadas aplicações auxiliares como editores de texto ou planilhas eletrônicas. O SGWF deve fornecer aos usuários do sistema meios de utilização dessas ferramentas, permitindo o intercâmbio, entre o SGWF e as aplicações, de documentos e dados gerados através desses aplicativos.

Interoperabilidade entre diferentes sistemas de gerenciamento de Workflow. Para que possa haver uma maior integração entre SGWFs de fabricantes diferentes, de forma a aproveitar melhor as características de cada produto, padrões de interoperabilidade entre SGWFs devem ser suportados.

Suporte a recuperação de falhas. O SGWF deve ser capaz de lidar com falhas de software e hardware, de forma transparente, maximizando a disponibilidade do sistema, escondendo, dentro do possível, o tratamento de erros dos usuários.

Monitoramento. Ao lidarmos com processos longos e muitas vezes complexos, mecanismos que possibilitem determinar o estado corrente dos processos em execução constituem a base para ações de gerenciamento, fornecendo dados para detecção de falhas e gargalos de desempenho no sistema.

Auditoria. A manutenção do histórico das execuções dos processos em SGWFs são dados importantes, servindo de base para processos de otimização, avaliação e reestruturação dos processos de Workflow de uma empresa.

Reconfiguração Estática. Embora workflows sejam essencialmente estáticos, processos do mundo real tendem a sofrer mudanças e melhoramentos constantes. O SGWF deve prever mudanças nos processos que este executa.

Muitos destes requisitos, como capacidade de rotear documentos, composição de atividades e suporte a compartilhamento de dados, dependem de aspectos relacionados a arquitetura e ao tipo de aplicação ao qual se destina o SGWF. Os requisitos descritos acima, portanto, portanto, não são obrigatórios.

Atualmente, várias pesquisas vêm sendo realizadas nas áreas de Workflow dinâmico e *ad-hoc* (tratamento de exceções), em especial, no desenvolvimento de linguagens e algoritmos que contemplem a mudança dinâmica de processos [EKR95; ABVVV00]. Estas pesquisas introduzem os requisitos a seguir:

Reconfiguração Dinâmica. Devido à característica de longa duração de muitas atividades e à necessidade de tratamento de exceções em processos, os SGWFs devem permitir a mudança dinâmica das definições de processos em andamento. A reconfiguração dinâmica (*dynamic change*) consiste em alterar o plano de um processo enquanto suas instâncias estão em execução. Esta mudança normalmente requer políticas de atualização destes casos [EKR95].

Tratamento de Exceções. Exceções costumam ocorrer com certa freqüência em SGWFs. Para certas aplicações, as exceções são a regra. Por exemplo, um ator selecionado para realizar uma atividade que dura dias, pode ter que se ausentar, necessitando ser substituído durante este período.

2.9 Workflow Distribuído e de Larga Escala

SGWFs operam em ambientes que são, por sua natureza, distribuídos. Seu principal objetivo é coordenar os esforços de agentes que estão, em princípio, dispersos. O trabalho corporativo é, em sua maioria, realizado através da colaboração entre unidades semi-autônomas de uma organização. Desta forma, é natural que a estrutura dos SGWFs reflita esta distribuição, especialmente quando estes sistemas destinam-se à execução de workflows de larga escala. Para estes SGWFs, a distribuição representa também uma forma de obter maior desempenho, evitando gargalos de centralização.

O conceito de larga escala é relativo e depende do estado da arte da tecnologia atual. SFWFs de larga escala são aqueles que, em configurações centralizadas tradicionais, demandam capacidade computacional e de banda passante de rede acima da média. Esta demanda é normalmente requerida pela necessidade de execução e controle simultâneo de um grande número de casos e atividades, normalmente envolvendo um elevado número de atores, potencialmente geograficamente dispersos.

2.9.1 Problemas dos SGWFs Convencionais

Durante todo o desenvolvimento deste projeto, foram analisados vários trabalhos que, de forma semelhante ao nosso, buscaram solucionar os problemas relacionados a workflows de larga escala. Dentre eles podemos citar o INCA, Exotica/FMQM, Mentor e Meteor. Estes trabalhos são descritos em mais detalhe no Capítulo 9.

Vários problemas se tornam eminentes quando o número de usuários, processos ou casos concorrentes é muito grande. De maneira geral, as arquiteturas de SGWFs distribuídas buscam solucionar os seguintes problemas:

Gargalos de desempenho. O servidor central e seus enlaces de comunicação representam limites ao aumento do número de casos concorrentes que este servidor pode gerenciar. A solução adotada pela maioria das arquiteturas existentes na literatura (vide Capítulo 9) é a distribuição, em maior ou menor grau, dos recursos e do controle envolvidos no workflow.

Uma solução alternativa, contudo, seria o aumento da capacidade de processamento e de comunicação (banda passante) do servidor central. Esta alternativa é mais custosa porém mais

simples, sendo, na prática, adotada por grande parte das corporações como bancos, empresas de crédito e companhias aéreas.

Descentralização do trabalho. Um outro motivo que leva a adoção da descentralização é a própria natureza do trabalho nas grandes corporações. Este é normalmente distribuído entre escritórios, filias, ou mesmo departamentos, fisicamente (ou administrativamente) distantes e, em certo grau, autônomos. Um servidor central único é inadequado para corporações dispersas por diversos países do globo ou mesmo, em menor escala, dispersos em países continentais como o Brasil, onde mensagens como "Terminal momentaneamente inoperante" ou "Serviço temporariamente indisponível" são freqüentemente geradas nos sistemas das grandes corporações nacionais. Neste exemplo, requisitos de disponibilidade e tolerância a falhas são mais evidentes.

Autonomia administrativa. Outra característica, também relacionada como o problema anterior, é a autonomia de certos setores, filiais ou departamentos de grandes corporações. Estas unidades normalmente operam com objetivos comuns, mas em ambientes operacionais distintos, podendo abordar problemas em perspectivas diferentes, de acordo com a cultura ou leis locais. Em um banco multinacional, por exemplo, são esperadas práticas administrativas, legislativas e fiscais diferentes, em cada país em que este esteja presente. A centralização de controle e dados destas empresas, se realizado por um SGWF centralizado, torna-se extremamente complexa.

Heterogeneidade. A diversidade dos sistemas e ambientes operacionais das grandes corporações é outro fator a ser considerado. Unidades empresarias autônomas precisam trocar, rotineiramente, dados e informações. A homogeneidade de sistemas computacionais, ou mesmo dos SGWFs destas corporações não pode ser assumida. Este aspecto é o principal problema que o modelo de referência de workflow da WfMC propõe-se a solucionar.

Mobilidade. Outro fator, em crescente demanda, é o suporte a clientes móveis, fracamente conectados, como *notebooks*, computadores *handheld*, telefones celulares, *pagers* e outros dispositivos móveis. Para estes sistemas, arquiteturas que permitam a descentralização de controle e de dados, dando-lhes certo grau de autonomia, são necessárias.

2.9.2 Requisitos Adicionais de SGWFs de Larga Escala

Com base nos problemas descritos anteriormente, além dos requisitos de SGWFs tradicionais, SGWFs que se destinam a execução de workflows de larga escala devem prover as características a seguir:

Escalabilidade. Grandes empresas como bancos e companhias aéreas envolvem milhares de usuários, centenas de milhares de processos concorrentes distribuídos em milhares de sites. SGWFs devem suportar o crescimento e as exigências de tais sistemas satisfazendo seus requi-

sitos de desempenho, permitindo atender a um número cada vez maior de processos, casos e atividades simultâneas.

Disponibilidade. Para que possam ser utilizados em aplicações de missão crítica, além de garantir a consistência dos dados e da aplicação, SGWFs devem prover mecanismos de tolerância a falhas (como replicação e distribuição de dados e controle) de forma a impedir que falhas no sistema paralisem os processos em andamento, suspendendo sua execução por longos períodos de tempo.

Tolerância a Falhas. A distribuição do wokrflow normalmente introduz vários pontos de falha no, exigindo que os mecanismos de tolerância a falhas em SGWFs distribuídos sejam mais elaborados.

Segurança. A descentralização introduz problemas de segurança. Em sistemas descentralizados, dados e controle do workflow são normalmente distribuídos entre máquinas (potencialmente) menos confiáveis e menos seguras que as normalmente empregadas em soluções centralizadas. Desta forma, se no caso centralizado o gerenciamento dos dados e o controle de execução eram responsabilidade de um único servidor, em workflows descentralizados, esta responsabilidade é compartilhada pelos nós por onde o sistema está distribuído. As questões de segurança são agora multiplicadas, sendo diretamente proporcionais ao grau de distribuição e descentralização utilizado.

Mobilidade e Operação Desconectada. De maneira a permitir sua execução em sistemas móveis, o SGWF deve ser desenvolvido de maneia a satisfazer os requisitos destes ambientes. Sistemas móveis normalmente executam em ambientes onde os recursos de memória e processamento são escassos; adicionalmente, o enlace de rede possui banda passante muito pequena, sendo ainda, em muitos casos, intermitente (alternando entre períodos de conexão e desconexão).

Capítulo 3

Fundamentos de CORBA e Agentes Móveis

É apresentado, neste capítulo, de maneira resumida, os principais conceitos de CORBA e da arquitetura OMA necessários à compreensão do presente trabalho. Após a leitura deste capítulo, o leitor deverá ser capaz de responder às seguintes perguntas: O que é CORBA? A que tipos de aplicações se destina? Quais as principais vantagens em seu uso? Como esta arquitetura é utilizada? Quais as principais dificuldades/desvantagens de sua utilização?

É apresentado, também, o conceito de agentes móveis, descrevendo seu paradigma e seus requisitos. Espera-se que, ao final deste capítulo, o leitor possa responder às seguintes questões: O que é um agente móvel? O que é um sistema de agentes móveis? Qual a principal diferença entre o paradigma de agentes móveis e o paradigma cliente-servidor? Quais os principais requisitos do paradigma de agentes móveis e como são normalmente atendidos? Quais seus principais problemas e suas principais vantagens?

3.1 OMA e CORBA

O *Object Management Group* (OMG) é um consórcio, criado em 1989, composto por mais de 800 empresas, dentre elas Netscape, IBM, Sun, Motorola, Nokia e Boeing. Seu objetivo inicial é a definição de padrões abertos que permitam a interoperabilidade entre componentes de software de forma independente de localização, plataforma, sistema operacional, linguagem de programação e protocolos de comunicação, em um ambiente distribuído [Seetharaman98; OH98]. Seu trabalho deu origem, em 1996, à arquitetura CORBA 2.0 (*Common Object Request Broker Architecture 2.0*), um padrão que definiu um *framework* de comunicação que implementa esta interoperabilidade. Atualmente, o OMG trabalha na extensão e aprimoramento deste modelo através do desenvolvimento da arquitetura OMA (*Object Management Architecture*) e

da linguagem UML (*Unified Modeling Language*). Estes modelos serão mais detalhados a seguir.

A *Unified Modeling Language* define uma notação gráfica usada no projeto e documentação de software, em especial, programas que utilizam o paradigma de orientação a objetos. A UML define notações para a representação de diagramas de classe, casos de uso e de comportamento. Estes últimos incluem diagramas de estado (*state charts)*, de atividade e de interação; define ainda notações que descrevem a implementação de sistemas, como diagramas de componentes e de interação de objetos [FS97].

A Arquitetura OMA, apresentada na Figura 6, padroniza as interfaces de vários componentes, de maneira a facilitar o desenvolvimento de aplicações distribuídas que utilizam a tecnologia de objetos distribuído [Siegel98]. A OMA define um ORB (*Object Request Broker*) e dois conjuntos básicos de componentes. O ORB é uma camada de software, conhecida como *middleware*, normalmente implementada utilizando a API (*Application Program Interface*) da camada de transporte. O ORB fornece serviços de comunicação e localização de objetos distribuídos. A arquitetura OMA define seu próprio barramento de objetos, descritos na especificação CORBA [CORBA98]. Além do ORB, a OMA padroniza vários outros serviços, classificando-os em três grupos, são estes: os Serviços de Objetos, as Facilidades Comuns e a Interface de Domínios. O principal objetivo destes serviços é prover um conjunto de componentes que forneçam soluções prontas, para problemas freqüentemente encontrados no desenvolvimento de ambientes de serviços integrados, permitindo o rápido e eficaz desenvolvimento dos Objetos de Aplicação.

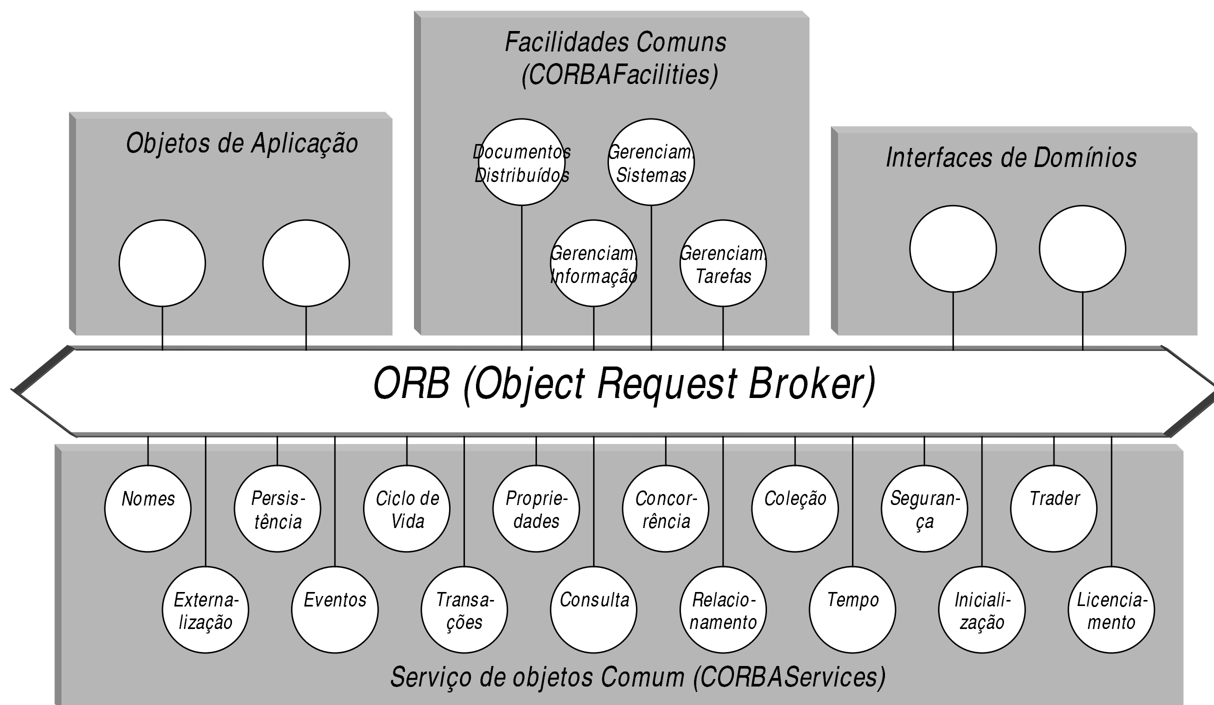

Figura 6: Arquitetura OMA

O **Serviço de Objetos (*CORBAServices*)** define um conjunto de componentes (objetos e interfaces) que estendem as funcionalidades básicas do ORB. Este conjunto de serviços é utilizado por grande parte das aplicações distribuídas, podendo ser usados tanto pelos Objetos de Aplicação como pelas Facilidades Comuns. São alguns exemplos desses objetos: Serviço de Nomes, de Ciclo de Vida, de Persistência, de Transações, de Eventos, de Notificação dentre outros [COSSpec97]. A inclusão destes serviços em implementações da arquitetura OMA é opcional mas, quando incluídos, devem ser implementados em conformidade com as especificações padronizadas pelo OMG.

As **Facilidades Comuns (*CORBAFacilities*)** definem um conjunto de serviços que podem ser utilizados por várias aplicações, mas que não são considerados fundamentais, como é o caso do Serviço de Objetos. Exemplos destes serviços são: a Facilidade de Bancos de Dados, de Impressão, de Correio Eletrônico e O *Workflow Management Facility*, parte da Facilidade de Gerenciamento de Tarefas. Esta última será discutida em maior profundidade no Capítulo 9, onde são descritos os trabalhos relacionados.

A **Interface de Domínios** é composta por objetos usados em áreas de aplicação específicas como Finanças, Saúde, Manufatura, Telecomunicações, Comércio Eletrônico e Transporte.

Finalmente, os **Objetos de Aplicação**, nada mais são que as aplicações criadas pelos usuários da arquitetura OMA. Estes objetos não estão sujeitos a padronização pelo OMG. Geralmente utilizam as funcionalidades do Serviço de Objetos e das Facilidades Comuns. Formalmente, os Objetos CORBA são aplicações ou componentes de software cujos serviços estão especificados através de interfaces IDL, permitindo que estes utilizem o barramento de comunicação CORBA para prover e utilizar serviços.

3.1.1 Estrutura de um ORB

Descrevemos a, seguir, em mais detalhes, a arquitetura e o funcionamento do padrão CORBA. A *Figura 7* apresenta o Modelo de Referência CORBA. Este modelo é formado por um conjunto de componentes que, juntos, permitem prover as transparências de acesso e localização desta arquitetura.

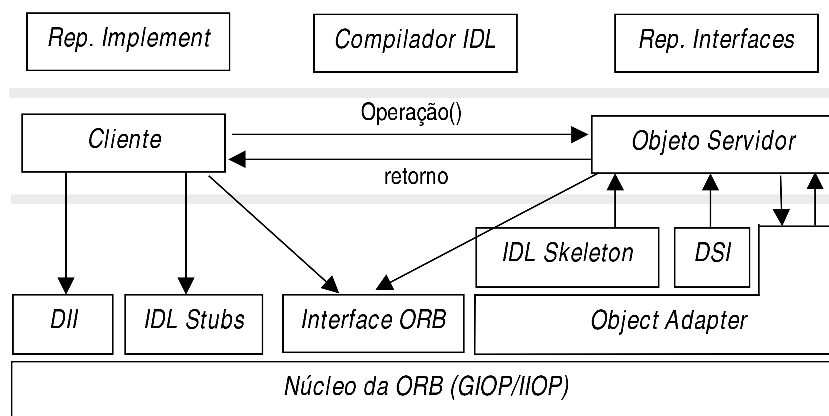

Figura 7: Componentes do Modelo de Referência CORBA

O elemento chave desta arquitetura é seu modelo de objetos. O **Modelo de Objetos CORBA** utiliza um conjunto de conceitos apoiados no paradigma de orientação a objetos, que facilitam a definição e mapeamento de objetos CORBA para diversas linguagens de programação. Estes conceitos incluem herança, polimorfismo, encapsulamento e exceções. Estes conceitos e definições foram implementadas na forma de uma linguagem, a IDL (*Interface Definition Language*).

A **IDL** (*Interface Definition Language*) é uma linguagem puramente declarativa, com sintaxe herdada de C++, que provê um meio comum para o uso e a exportação de operações e serviços por objetos CORBA. Esta linguagem permite estabelecer "contratos" entre clientes e servidores, padronizando suas ofertas e usos de serviços. A IDL define uma linguagem comum de comunicação entre objetos implementados em linguagens de programação diferentes, que podem estar executando em plataformas de hardware heterogêneas.

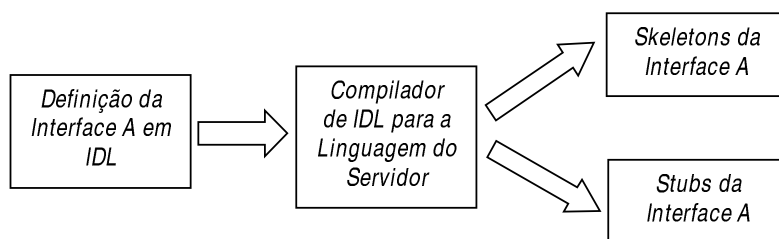

Figura 8: Processo de Geração de Stubs e Skeletons a partir de uma interface IDL.

O uso da linguagem IDL está associado a seu mapeamento específico para cada linguagem de programação. Este mapeamento está descrito no Modelo de Objetos CORBA que, atualmente, padroniza o mapeamento de IDL para as linguagens: C, C++, Java, Smalltalk, ADA e COBOL.

Tipos e interfaces definidos em IDL são convertidos para uma determinada linguagem de programação por um compilador específico. Este compilador gera um conjunto de objetos chama-

dos *stubs* e um conjunto de interfaces que os servidores devem implementar chamados *skeletons,* como descrito na *Figura 8*.

Servidores, cujas interfaces são descritas em IDL, devem fornecer a implementação, em sua linguagem nativa, para cada operação de sua interface. Clientes realizam invocações remotas usando os *stubs*. Os *stubs*, e seu lado servidor, *skeletons*, utilizam bibliotecas, APIs e pacotes que implementam o barramento CORBA e seu protocolo de comunicação. Toda a comunicação é realizada de forma transparente para os servidores e clientes que, respectivamente, fornecem e utilizam serviços remotos sem que haja preocupação com aspectos de comunicação e localização. Do ponto de vista de padrões de projeto (*design patterns*) [GHJV95], *stubs* e *skeletons* são *proxies* (representantes) locais de objetos remotos. O processo de invocação é descrito na *Figura 9*.

Do ponto de vista de implementação, as funcionalidades do ORB são providas por bibliotecas que são ligadas, dinâmica ou estaticamente, aos módulos executáveis que implementam o cliente e o servidor. Isto permite que clientes se comuniquem com servidores usando invocações realizadas em um mesmo processo (*in-process*), ou entre processos diferentes, que podem estar sendo executados em máquinas diferentes. Os *stubs* são também conhecidos como SIIs (*Static Invocation Interfaces*) ou interfaces estáticas de invocação.

O uso conjugado de compilador IDL e do ORB permite isolar cliente e servidor. Esta característica permite ao servidor prover várias implementações diferentes para uma mesma interface IDL. Este isolamento permite que s*tubs* e *skeletons* sejam gerados para linguagens de programação diferentes. Os *stubs* e os *skeletons* realizam, ainda, o empacotamento (ou serialização) e desempacotamento (ou des-serialização) de parâmetros (*marshaling* e *unmarshaling*) de maneira transparente para cliente e servidor.

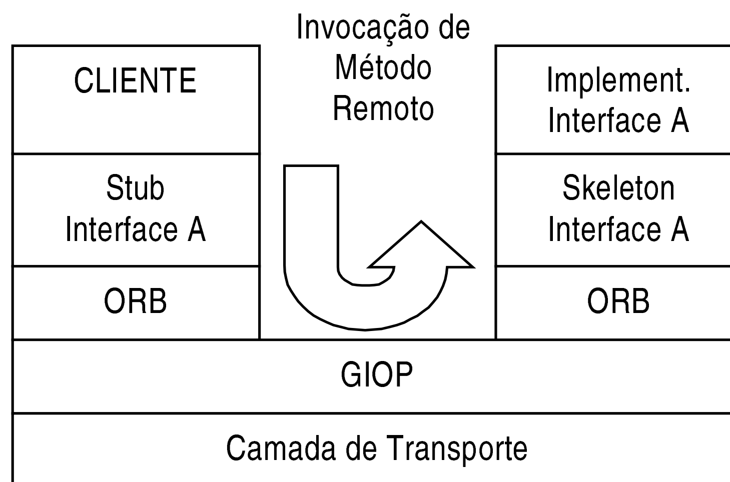

Figura 9: Passagem de uma requisição remota de um cliente a um servidor.

3.1.1.1 Mecanismo de Invocação Remota de Operações

As requisições de operações, feitas por clientes, podem ser realizadas de forma estática ou dinâmica. O repasse de invocações de operações locais para servidores remotos é realizado pelos s*tubs*. Estas invocações podem ocorrer de maneira síncrona ou assíncrona. Clientes utilizam os *stubs* como representantes (*proxies*) dos servidores remotos, criando a ilusão de estar-se usando um objeto local. Requisições assíncronas são implementadas pelos *stubs* através do uso de *threads*, permitindo que o cliente continue sua execução logo após a realização da invocação remota. O uso dos *stubs* é, desta maneira, definido em tempo de compilação, de forma estática. Esta prática é utilizada sempre que há um conhecimento prévio de todos os servidores que um cliente irá utilizar.

Em determinadas aplicações, não se conhece, de antemão, quais objetos servidores serão utilizados. Em uma aplicação de comércio eletrônico, por exemplo, onde um agente deve pesquisar vários sites de forma a achar a melhor oferta para um determinado produto, o agente precisa consultar a interface de vendas de cada site de maneira a encontrar o melhor preço de uma mercadoria. Esta interface de vendas pode não ser conhecida pelo cliente, necessitando ser determinada em tempo de execução.

Quando um servidor, e portanto sua interface, não são conhecidos em tempo de execução, a invocação estática de operações não pode ser utilizada. Para estes casos, CORBA define uma **Interface de Invocação Dinâmica**: um conjunto de operações na API do ORB (Interface ORB da *Figura 7*) que permite aos clientes descobrir, em tempo de execução, a interface (operações e atributos) de um determinado objeto ou serviço. De posse desta informação, o cliente pode, então, através de chamadas do ORB, compor e enviar uma requisição (e receber respostas) ao servidor em questão.

Este recurso do ORB é implementado com o auxílio do **Repositório de Interfaces** que armazena as definições de interfaces, em IDL, para cada servidor implementado. De maneira análoga, do lado servidor, temos a **Interface Skeleton Dinâmica** (*Dynamic Skeleton Interface*) que permite tratar invocações de operações feitas a servidores CORBA que não possuem *skeletons* registrados no repositório de implementação. Estas invocações são então repassadas para a Interface *Skeleton* Dinâmica, uma interface genérica que, dentre várias opções, pode localizar um servidor para responder às invocações recebidas.

Neste contexto, o **Repositório de Implementação** provê informações que permitem ao ORB ativar os objetos servidores tanto dinamicamente quanto estaticamente. Este repositório armazena ainda informações relacionadas ao modo de ativação dos objetos, à alocação de recursos, à segurança e aos parâmetros de administração.

3.1.1.2 Protocolos de Comunicação

A comunicação entre objetos cliente e servidor é realizada, de forma transparente, entre seus ORBs. Cada ORB fornece uma API de serviços, utilizada pelos *Stubs* e *Skeletons* destes objetos. Os *stubs* e *skeletons* funcionam como uma "cola" entre os clientes, servidores e os ORBs, que realizam enviam as requisições de um nó a outro.

O protocolo empregado na comunicação entre ORBs é padronizado pelo OMG. O **GIOP** (*General Inter-ORB Protocol*) define um conjunto de primitivas de comunicação, independentes das camadas de transporte e de rede, que permite a comunicação entre ORBs. O protocolo **IIOP** (*Internet Inter-ORB Protocol*) é a implementação deste protocolo para redes baseadas nos protocolos TCP/IP, sendo obrigatório para as implementações CORBA.

Os tipos de dados especificados em IDL são representados no formato CDR (*Common Data Representation*), antes de serem transportados pela rede. O CDR normaliza a transferência de dados entre clientes e servidores que executam em plataformas de hardware diferentes.

Iiop:1.0//	*WONDER.pos.dcc.unicamp.br:2015*	*P353bccdb00094ac8*	*FirstPOA*	*Marker*
Protocol ID	Communication Endpoint	Timestamp	Object Adapter ID	Object ID

Figura 10: Estrutura de uma IOR (Interoperable Object Reference)

O modelo de objetos CORBA é baseado no conceito de referências de objetos. Uma referência de objeto identifica unicamente um objeto local ou remoto. Clientes só podem realizar invocações de operações nestes objetos estando de posse destas referências [Henning98]. Instâncias de servidores CORBA são representados por *Interoperable Object References* (IORs), estas referências são geradas pelo adaptador de objetos e permanecem opacas para clientes e servidores. Internamente, o formato de uma IOR é extensível, possuindo uma estrutura composta por espaços reservados para informações de cada protocolo, estes dados são usados por ORBs de clientes para localizar servidores em ORBs locais ou remotos. Em algumas implementações, a IOR inclui o endereço de rede de um objeto CORBA. Um exemplo de uma IOR é descrita na Figura 10.

O **Adaptador de Objetos** (*Object Adapter*) é o componente responsável pela criação de IORs. Este adaptador implementa a ativação e desativação de servidores; sendo responsável por direcionar (ou multiplexar) requisições de operações estes servidores, colaborando com os *skeletons* na inovação de operações remotas [PS98].

3.1.2 O Futuro da arquitetura OMA

A arquitetura OMA está em constante revisão e extensão. Novas funcionalidades, serviços e objetos para esta arquitetura estão em constante desenvolvimento. O modelo básico de CORBA, em sua versão 3.0, está sendo estendido de maneira a incluir múltiplas interfaces por objeto, objetos passados por valor, um modelo de componentes semelhante ao adotado por Java

Beans, suporte a aplicações de tempo real, tolerância a falhas e suporte a software básico (*embeded* CORBA) [OMG00].

3.1.3 Principais Vantagens de CORBA

A principal vantagem do uso de CORBA, e dos outros padrões OMA, é sua capacidade de permitir o desenvolvimento de aplicações que são interoperáveis em um ambiente de programação de alto nível. Isto é feito de maneira independente de sistema operacional, linguagem de programação, protocolos e topologia de rede. CORBA utiliza recursos de orientação a objetos como polimorfismo, herança e encapsulamento, facilitando a implementação de componentes reutilizáveis de software. Esta arquitetura provê ainda um conjunto de serviços abertos e padronizados (*CORBAServices* e *CORBAFacilities*) que facilitam a implementação de aplicações distribuídas.

CORBA é empregado também na integração de sistemas legados, através da definição de interfaces IDL para aplicações destes sistemas. Isto permite a utilização de aplicações legadas por clientes escritos em linguagens modernas, sem que haja a necessidade de rescrita total destes sistemas legados para plataformas de hardware e software diferentes.

3.1.4 Principais Desvantagens de CORBA

Podemos citar as principais desvantagens de CORBA como sendo:

Seu elevado custo de aprendizagem e utilização. CORBA é uma tecnologia que envolve conceitos avançados de programação orientada a objetos, sofrendo dos problemas encontrados no uso de *frameworks*. A experiência de implementação da arquitetura descrita nesta dissertação, exigiu um conhecimento prévio de conceitos relacionados à programação orientada a objetos, assim como experiência no desenvolvimento de aplicações distribuídas em um ambiente que dificulta a depuração de programas, visto que servidor e cliente executam em processos independentes.

A compreensão dos conceitos de ORB e dos serviços de objetos com sua posterior utilização bem sucedida constitui um processo demorado e custoso de aprendizado, normalmente baseado em sucessivas tentativas e erros.

Os produtos comerciais existentes como o OrbixWeb [OrbixWeb] e o Visibroker [Visibroker] disponibilizam apenas um pequeno subconjunto de serviços CORBA, basicamente o serviço de nomes e, em alguns casos os de eventos e notificação, segurança e transação. A Prism Technologies provê um conjunto quase completo de serviços CORBA, o Openfusion [Openfusion] para Java, que pode ser integrado com os dois produtos comerciais descritos acima. Esta inte-

gração, entretanto, é ainda não trivial, sofrendo de problemas de configuração e adaptação a versões anteriores destes produtos.

Durante a realização deste trabalho, foram pesquisadas várias implementações de domínio público. A maioria das implementações, contudo, provêm apenas as funcionalidades básicas do ORB e, em alguns casos, serviços como o de nomes e o de eventos [FreeCORBA].

A integração de clientes CORBA com servidores implementados para ORBs de fabricantes diferentes é ainda uma tarefa não trivial. Isto deve-se principalmente a diferenças relacionadas aos padrões implementados por cada ORB. Padronizações como a POA (*Portable Object Adapter*) [POA97], contudo, estão sendo realizadas no sentido de reduzir esta limitação. No POA, os *skeletons* gerados por diferentes compiladores IDL, de diferentes fabricantes, assim como sua interação com o ORB, é padronizado. Isto permite que os stubs assim gerados possam ser usados em ORBs diferentes.

3.1.5 OrbixWeb

Este trabalho foi desenvolvido utilizando-se a implementação de CORBA para Java da Iona [Iona], o OrbixWeb 3.1c. Este ORB apresenta os seguintes recursos [OrbixPrg98]:

- Compilador IDL com mapeamento para Java
- Suporte à interface de invocação dinâmica (DII) e à DSI *(Dynamic Skeleton Interface*), além do conjunto básico de funcionalidades do ORB.
- Gerente de Ativação. O *orbixd* (*OrbixWeb daemon* escrito em C++) é um processo que executa em cada nó do sistema. Este *daemon* implementa vários componentes do ORB, dentre eles o Repositório de Interfaces, que permite o uso da interface de invocação dinâmica de servidores.
- Implementação do padrão CORBA 2.0 [CORBA98] do OMG.
- Comunicação padrão entre os servidores ocorre utilizando o protocolo IIOP.

O ORB implementado no OrbixWeb é estendido através de vários serviços adicionais como:

Filtros. O OrbixWeb permite especificar interceptores de invocações de operações que podem ser executados antes ou após o acesso a atributos ou métodos de um servidor. Este recurso permite inspecionar e modificar parâmetros de invocações remotas. Pode ser usado para depuração, auditoria, checagens de segurança e outros.

Smart Proxies. Permite implementar *proxies* (que operam em conjunto com *stubs* e *skeletons*) de forma manual. Este recurso permite otimizar operações do lado cliente, ou implementar políticas de balanceamento de carga.

Loader. Permite implementar a persistência de objetos. Classes derivadas desta *interface (Loader.class*) são *hooks* (pontos adaptáveis opcionais em *frameworks*) no *orbixd*. Objetos do tipo

Loader podem ser associados a servidores CORBA, na hora em que servidores são criados pelo gerente de ativação (*orbixd*). Quando isto ocorre, invocações associadas a este tipo de servidor são interceptadas pelo seu *loader* correspondente. Este objeto é responsável por salvar e recuperar o estado de servidores CORBA anteriormente salvos ou, no caso destes não existirem, criar novos objetos. Estes objetos, recuperados ou criados, são repassados para o *orbixd* que os disponibiliza para uso, por quem os requisitou, atribuindo a este servidor uma porta livre no sistema.

Locator. Este recurso do OrbixWeb, implementado no orbixd, permite associar um nome a cada objeto criado (*marker*). Este *marker* identifica unicamente cada objeto no nó corrente. O *Locator* nada mais é que uma espécie de servidor de nomes local que associa *markers* a referências de objetos. Desta forma, o *orbixd* permite a realização de operações *bind()* (Conexões cliente-servidor) utilizando um nome (*marker*) como parâmetro.

Criação Automática de Servidores. O *orbixd* utiliza o Repositório de Interfaces juntamente com o *Locator* e o *Loader* para instanciar objetos de maneira automática. Ao receber um pedido de *bind()*, especificando um *marker* e o tipo do servidor a ser criado ou conectado, o *orbixd* verifica se tal objeto está ativo no nó corrente. Se a resposta for negativa, o *orbixd* direciona a chamada para o *Loader* que pesquisa a existência do objeto em sua base de dados. Caso este objeto não exista, o *Loader* cria automaticamente uma nova instância, do tipo do servidor especificado no *bind*, atribuindo a este objeto o *marker* passado como parâmetro. A criação automática de novas instâncias de servidores é implementada pelo *orbixd* com o auxílio do repositório de implementação.

O Repositório de Implementação do OrbixWeb é um banco de dados que permite associar a cada servidor, um conjunto de parâmetros de ativação (passados à máquina virtual Java), a localização da implementação das interfaces IDL, o modo de ativação (*shared* ou *unshared*), permissões de acesso (criação e invocação), dentre outros atributos.

Modos de Ativação. O OrbixWeb permite especificar modos de ativação diferentes para cada servidor. Estes modos de ativação são armazenados no repositório de implementação. Servidores podem executar no modo *shared* (compartilhado) ou *unshared* (não compartilhado). Servidores *shared* só podem ter uma única instância por nó, podendo ser utilizados por vários clientes simultaneamente. Para que várias instâncias de um servidor possam ser criados em um mesmo nó, estes precisam ser registrados com *unshared.*

No modo *unshared*, o *orbixd* cria um processo independente, com sua própria máquina virtual, para cada servidor Java instanciado. Desta forma, cada servidor responde em uma porta diferente. A comunicação entre servidores em uma mesma máquina, ou em máquinas diferentes, é implementada da mesma forma, utilizando a porta TCP deste processo. Servidores em um mesmo nó comunicam-se através da interface lógica de rede do sistema operacional. No caso do sistema UNIX, esta interface é conhecida como *loopback*.

3.2 Agentes Móveis

Esta sessão apresenta o paradigma de agentes móveis, descrevendo seus principais requisitos, aplicações, conceitos, linguagens, relacionando suas principais vantagens e desvantagens.

3.2.1 Paradigma de Agentes Móveis

Em termos gerais, um agente de software pode ser definido como um programa que age em favor de seu dono (*agent owner*) [GHNCSE97]. Um agente móvel é, desta forma, um software que representa um usuário em uma rede de computadores, podendo migrar de forma autônoma, de nó em nó, realizando tarefas em favor de seu dono. Ao final deste processo um agente pode retornar ao seu nó de origem (*home site*) e reportar-se a quem o injetou na rede [KT98b]. Em outra definição, um agente móvel é descrito como um objeto que migra por vários nós de uma rede heterogênea, sob seu próprio controle, de maneira a realizar tarefas utilizando recursos destas máquinas [IH99; RGK97].

No paradigma cliente-servidor, detentores de um determinado recurso estão fisicamente distantes de seus clientes (usuários). A comunicação entre as partes ocorre através de uma rede de computadores. No paradigma de agentes móveis, recursos e seus usuários movem-se de maneira a interagirem localmente. Através de sua mudança de localização, agentes podem mudar a qualidade de interação dinamicamente, reduzindo seu custo [CPV97]. São alguns exemplos de aplicações do paradigma de agentes móveis: o lançamento e atualização de aplicações, feita de maneira distribuída, a customização de serviços e a implementação de políticas de tolerância a falhas.

O modelo cliente-servidor utiliza-se de infra-estruturas de comunicação, a exemplo de CORBA, para realizar a troca de informações entre processos/objetos normalmente localizados em nós diferentes. Neste paradigma, dados são transportados entre servidor e cliente utilizando invocações de métodos remotas ou trocas de mensagens. Neste paradigma, a confiabilidade dos enlaces de comunicação e a sincronicidade das chamadas de procedimento remoto (RPC) são requisitos importantes.

O paradigma de agentes móveis destina-se, principalmente, a ambientes distribuídos, executando em uma rede (lenta) composta por um grande número de máquinas. Neste contexto, os agentes móveis são utilizados para reduzir o tráfego de dados na rede. Normalmente, o agente migra para o nó que fornece os serviços (ou dados) a serem consultados ou utilizados [CHK94]. Em aplicações que utilizam bancos de dados, por exemplo, agentes podem mover-se para o local onde os dados se encontram, realizando consultas e filtrando a informação relevante antes de repassá-la para o cliente. Neste caso, é mais barato enviar um pequeno agente até a origem dos dados do que trazer resultados de consultas até o nó onde o agente se encontra.

Os fundamentos metodológicos e tecnológicos utilizados no desenvolvimento de sistemas distribuídos convencionais falham quando aplicados a problemas de distribuição envolvidos em aplicações de larga escala [CPV97]. De uma maneira geral, no projeto de arquiteturas em sistemas distribuídos, a interação entre seus componentes é definida de forma independente de sua localização. *Middlewares* baseados em CORBA, por exemplo, permitem tornar transparente a localização dos componentes. Neste *framework*, não há distinção entre as interações local ou remota. Em contra partida, o paradigma de agentes móveis surge como uma nova abordagem para o projeto e especificação de sistemas distribuídos. Este paradigma é normalmente empregado onde os conceitos de localização e mobilidade precisam ser levados em conta devido a requisitos de desempenho e confiabilidade. Em problemas que utilizam o paradigma de agentes móveis, estes requisitos são tão importantes que afetam a estrutura conceitual da aplicação quando esta é definida na fase de projeto.

3.2.2 Aplicações

Detalhamos a seguir algumas aplicações e requisitos de sistemas distribuídos que se beneficiam do paradigma de agentes móveis.

Computação Móvel. Em aplicações envolvendo mobilidade, a presença de uma conexão de rede é intermitente, ou possui taxas de transmissão de dados que variam ou são muito baixas [RGK97]. Agentes móveis são entidades autônomas e independentes em relação à aplicação que os gerou. O agente leva consigo seu estado, seu código fonte e alguns dados necessários a sua execução. O cliente (*agent owner*) que lançou o agente na rede não precisa manter conexões com estes objetos. Uma vez hospedado em um nó, um agente pode ser facilmente configurado para esperar até que a conexão de rede seja restabelecida antes migrar para outro nó. Devido a estas características, o paradigma de agentes móveis permite a implementação de aplicações envolvendo computação móvel.

Tolerância a Falhas. Tarefas e aplicações complexas podem ser quebradas em pequenas subtarefas na forma de agentes móveis. Estes agentes podem migrar por nós de uma rede ou podem ser duplicados de maneia a não dependerem de um único nó. Aplicações que utilizam controle ou monitoramento de recursos remotos, por exemplo, podem evitar atrasos ou falhas de rede, enviando agentes para o nó contendo o recurso a ser monitorado. Neste cenário, agentes notificam seus *owners* apenas quando uma determinada condição de controle for satisfeita, por exemplo, quando o recurso monitorado apresentar um comportamento anormal. A assincronicidade é uma característica intrínseca do paradigma de agentes móveis. O agente não precisa estar conectado com quem o lançou durante sua vida. Isto promove uma maior tolerância a falhas de rede na comunicação entre o agente e seu *owner*.

Balanceamento de Carga. Agentes podem ser transportados de um nó a outro de maneira a implementar políticas de balanceamento de carga ou tolerância a falhas, migrando através de

nós do sistema distribuído de maneira a distribuir a carga de processamento ou comunicação de maneira mais uniforme.

Comércio eletrônico. Agentes podem ser configurados, por exemplo, para percorrerem diversos nós de uma rede de forma a intermediar transações comerciais, procurar melhores ofertas de produtos e serviços, ou realizar compras e pedidos. Ao final de sua missão, o agente retorna a seu nó de origem (*home site*).

Gerência de Sistemas Distribuídos e Redes de Telecomunicação. Agentes podem ser configurados para percorrer nós em um sistema distribuído, coletando dados (gerenciamento passivo) e/ou reconfigurando nós (gerenciamento ativo), de maneira a implementar diversas políticas de gerenciamento. Redes de gerenciamento de telecomunicações, normalmente detentoras de enlaces onde há baixa banda passante, se beneficiam do paradigma de agentes móveis.

3.2.3 Vantagens

Harrison et. al [CHK94] descreve as principais vantagens individuais do paradigma de agentes móveis como sendo: a interação local agente-servidor, resultando em maior banda passante; a adequação com a operação desconectada em computação móvel; o suporte a clientes fracos/magros, com pouca capacidade computacional ou como recursos escassos; a facilidade de implementação de roteamento semântico, como em aplicações de workflow; a escalabilidade; e a interação remota assíncrona, esta última sendo mais tolerante a falhas de links de rede que a interação síncrona. Nenhuma destas vantagens, contudo, é exclusiva do paradigma de agentes móveis. Todas elas podem ser implementadas em outros paradigmas como o cliente-servidor.

Em contrapartida, as principais desvantagens individuais do paradigma de agentes móveis segundo Harrison et. al são: a necessidade de ambientes de execução seguros, com restrições de acesso mais severas; limitações de desempenho geradas por políticas de segurança, como o uso de linguagens interpretadas e o emprego de limitações de acesso a recursos; a necessidade de mecanismos de detecção de agentes mal intencionados (vírus); e, finalmente, o *overhead* de comunicação e processamento envolvido na migração do agente.

Harrison et. al argumenta, contudo, que o uso do paradigma de agentes móveis, se levado em conta todos os seus pontos positivos e negativos, permite prover um *framework* aberto e genérico, para a customização e o desenvolvimento de aplicações de rede. Além do mais, o conjunto de vantagens agregadas, provido por tal paradigma, é dificilmente implementado em paradigmas como o cliente-servidor. Harrison et. al conclui que, enquanto as vantagens individuais do paradigma de agentes móveis não representam um fator que justifique a sua adoção, a criação de um *framework* de agentes móveis facilita o desenvolvimento de um grande conjunto de aplicações e serviços de rede.

3.2.4 Requisitos

Sistemas que suportam o uso do paradigma de agentes móveis devem, por tanto, prover um conjunto básico de serviços e características, satisfazendo os requisitos listados a seguir:

Transportabilidade: Um agente deve ser capaz de mover-se, sob seu próprio comando, de uma máquina a outra em uma rede heterogênea. Ou seja, o programa deve ser capaz de suspender sua execução em um nó, mover-se para outro nó e reiniciar sua computação a partir do ponto onde parou, usando seus próprios recursos. Esta migração deve ocorrer de forma independente da plataforma de hardware e software existentes na rede de computadores. O transporte do agente (ou seu estado) de um nó a outro pode ser auxiliado por entidades externas como serviços de mensagens e servidores de e-mail, ou pode ser provido, de forma transparente, por um sistema de agentes móveis como Voyager [ObjectSpace97] ou Aglets [KLO97] e outros.

Autonomia: O agente deve ser capaz de decidir para onde e quando migrar durante o cumprimento de sua missão. Esta movimentação pela rede pode ser totalmente reativa, baseada em dados dinâmicos relacionados a parâmetros de uso da rede, ou pode seguir um plano preestabelecido (seqüência não estática de recursos/nós que devem/podem ser visitados, juntamente com tarefas a serem realizadas em cada site). A comunicação com seu *home site* deve ser evitada ao máximo. Para tal, o agente deve utilizar recursos e mecanismos que o permitam tomar decisões relacionadas a seu comportamento e migração. Estes mecanismos são normalmente providos por sensores, que permitem ao agente coletar dados sobre seu ambiente, assim como por interpretadores de planos, algoritmos ou outros mecanismos *internos* do agente, que lhe atribuem um certo grau de "inteligência".

Navegabilidade. De forma a tomar decisões quanto a sua migração, agentes devem ter conhecimento de seus objetivos e planos assim como de parâmetros relacionados a seu meio ambiente. Este conhecimento do meio pode ser auxiliado por mecanismos *externos* aos agentes a exemplo de *traders*, serviços de nomes, páginas amarelas virtuais, dentre outros.

Segurança. Agentes são geralmente implementados como código e dados que se movimentam entre máquinas de um sistema distribuído. Neste aspecto, um agente pode ser comparado a um vírus de computador. Esta característica introduz dois problemas: os agentes precisam ser protegidos da ação de máquinas "maliciosas" e as máquinas precisam ser protegidas contra agentes "maliciosos" ou vírus. A segurança dos dados transportados é outro fator importante a ser tratado por estes sistemas. Um exemplo do mau uso do paradigma de agentes móveis foi o vírus "I LOVE YOU" que danificou milhares de computadores em maio de 2000 [Freedman00; IloveYou00].

Tolerância a falhas. Agentes executam em vários nós de um sistema distribuído, migrando através de máquinas e conexões de rede pouco confiáveis. O agente deve ser capaz de detectar erros de hardware e software, tomando as providências necessárias para contornar estes erros. Tais providências, por exemplo, podem compreender: avisar outros agentes sobre falhas, mo-

ver-se para um recurso alternativo, esperar até que um dado recurso fique ativo novamente, dentre outros.

Desempenho. O processo de movimentação de um agente deve ser eficiente, de forma a compensar seu uso, quando comparado a outros paradigmas como o cliente/servidor. O agente precisa ser pequeno, favorecendo sua rápida transferência entre os nós da rede, além de ser capaz de executar em máquinas com possíveis restrições de memória e processamento, a exemplo de computadores móveis e portáteis (*handheld*).

Suporte multiplataforma. Agentes devem ser capazes de migrar entre máquinas de diversas arquiteturas de hardware, executando em sistemas operacionais heterogêneos.

Adaptabilidade. A sensibilidade às diversas condições de tráfego, conexão e topologia de uma rede, assim como aos recursos disponíveis em cada nó, é outra característica que um agente deve possuir. Esta informação é normalmente utilizada na tomada de decisões relacionadas à migração, tolerância a falhas, modo de operação (conectado/desconectado) e outros. Este recurso é auxiliado pelo uso de sensores que coletam informações do ambiente.

Comunicação. Outra característica necessária aos agentes móveis é a capacidade de comunicar-se de maneira independente de localização. Agentes migram constantemente, não possuindo desta forma, uma localização (endereço) fixa na rede. Para tal, os agentes normalmente utilizam-se de mecanismos de rastreamento e localização, a exemplo de serviços de nomes (mantidos atualizados) ou serviços de mensagens. A comunicação pode ser implementada de forma assíncrona (baseada em datagramas), síncrona, através de chamadas de procedimentos ou de métodos remotos, ou através de arquivos ou dados compartilhados (via arquivos no NFS por exemplo). Primitivas de comunicação de grupo podem ainda ser utilizadas.

3.2.5 Sistemas de Agentes Móveis

Um Sistema de Agentes Móveis (SAM), ou Agência, é uma infra-estrutura computacional que implementa o paradigma de agentes móveis, provendo serviços e primitivas que auxiliam na implementação e migração destes agentes. De forma a hospedar os agentes móveis, cada máquina hospedeira precisa fornecer um ou mais ambientes básicos de suporte a estes objetos/processos, provendo mecanismos que permitam aos agentes migrarem, comunicarem-se entre si, obter acesso aos recursos da máquina em questão, devendo prover mecanismos que permitam satisfazer os requisitos listados anteriormente. Mecanismos adicionais como persistência de dados e de objetos (usados em políticas de tolerância a falhas), e mecanismos de localização de objetos podem ainda ser providos.

O sistema de agentes móveis Telescript, que utiliza uma linguagem de programação de mesmo nome [White94], desenvolvido pela General Magic no início de 1990, foi o primeiro sistema especialmente projetado para o desenvolvimento de aplicações comerciais utilizando o paradigma de agentes móveis. O exemplo deste sistema foi logo seguido por vários outros como o

Tacoma [JRS95] e Agent Tcl [Gray96], nos quais os agentes são descritos em linguagens de *script* proprietárias. O advento da linguagem Java, com seu suporte a código móvel, incentivou o crescimento da pesquisa e desenvolvimento de novos SAMs como Aglets [KLO97] da IBM, Voyager [ObjectSpace97] da ObjectSpace, Concordia [Concordia97] e Ajanta [KT98a], todos utilizando os recursos da linguagem Java. Uma comparação entre estes sistemas é descrita em [KT98a].

SAMs são específicos para determinadas linguagens como TCL, Java e Telescript. Os sistemas mais antigos implementam seus próprios protocolos e mecanismos de migração, aqueles baseados em Java, normalmente utilizam a passagem de objetos por valor, um recurso provido pela API de RMI (*Remote Method Invocation*) desta linguagem, como mecanismo de migração. Alguns sistemas como o Aglets e o Voyager são compatíveis com o protocolo IIOP da CORBA, permitindo que agentes escritos para estes sistemas comuniquem-se com servidores CORBA. A integração destes sistemas com CORBA, contudo, limita-se a esta comunicação unilateral.

3.2.6 Propostas do OMG

Como parte das Facilidades Comuns da arquitetura OMA, o OMG define uma Facilidade de Interoperabilidade de Sistemas de Agentes Móveis (MASIF – *Mobile Agents System Interoperable Facility*) [OMG-MASIF98] Seu principal objetivo é definir um *framework* comum, baseado em CORBA, para a interoperabilidade entre os diferentes SAMs. Esta especificação é bastante genérica e não trata dos agentes como objetos CORBA, assim como não define mecanismos de transporte de destes objetos através de um ORB. Este último requisito é abordado em uma outra especificação do OMG, o mecanismo de passagem de objetos CORBA por valor (*Object by Value RFP*) [OMG-OBV96]. Esta especificação contudo, provê apenas um mecanismo que permite a implementação de um SAM em CORBA. Este recurso estará disponível no padrão CORBA 3.0 [Vinoski 98].

Uma facilidade de agentes móveis usando CORBA [VM98] foi implementada como trabalho de mestrado no Instituto de Computação da UNICAMP, e trata da implementação de um SAM em CORBA usando a linguagem Java. Posteriormente, um mecanismo de migração transparente de agentes móveis usando CORBA [Schulze99]., também foi definido como um trabalho de doutorado, deste mesmo instituto. Este trabalho apresenta detalhes de implementação e os resultados de alguns testes de desempenho.

3.2.7 Principais Requisitos dos Sistemas de Agentes Móveis

Descrevemos, a seguir, em mais detalhe, os requisitos mais comuns que um SAM deve satisfazer.

3.2.7.1 Mecanismo de Migração

O processo de migração de um agente pode ser implementado de duas formas: o agente pode criar uma outra cópia de si mesmo (*fork*), que segue executando em um outro nó da rede, de maneira independente, ou pode suspender sua execução e mover-se para um outro nó onde a execução é retomada.

De maneira geral, para mover-se de uma máquina a outra, o agente precisa: parar seu processamento, salvar seu estado corrente, mover-se para o próximo nó (juntamente com seu estado e seu código) e reiniciar o processamento de a partir do ponto onde este foi interrompido. Para tal, o agente pode ser visto como a composição de 2 partes básicas: o código do agente e seu estado de execução (no caso de objetos, os valores de seus atributos; em alguns casos a pilha de execução do processo). A maior parte das implementações de agentes, contudo, implementa o que a literatura chama de "migração fraca" (*weak migration*), uma migração de granulosidade grossa onde o agente move-se apenas com seu estado de execução e código, não levando sua pilha de execução. Na migração fraca, o a gente precisa ser parado em um estado consistente antes de ser transportado [IH99].

Como agentes móveis são autônomos, sua migração ocorre sob seu próprio comando. Desta forma, mecanismos mais avançados que permitem a captura do estado de execução, usando granulosidade fina ou seja, guardando o estado da pilha do *thread* de execução corrente, são normalmente desnecessários [KT98b]. Esta última abordagem é conhecida como "migração forte" (*strong migration*). Definições mais precisas de migração forte e fraca podem ser encontrados em [CPV97].

Outro ponto importante na implementação do processo de migração de agentes é a sua mobilidade de código. Alguns sistemas carregam seu próprio código, que é transportado juntamente com seu estado de execução. Outros sistemas, contudo, não implementam esta funcionalidade. Neste último caso, o código do agente precisa estar pré-instalado no ambiente de suporte a agentes móveis das máquinas que o hospedarão, ou estar disponível através de sistemas de arquivos compartilhados (NFS por exemplo). Em uma terceira abordagem, o agente não carrega seu código, mas sim uma referência a uma base de dados de códigos (*code base*) de onde este pode ser copiado e instalado no ambiente hospedeiro, sob demanda (*code on demand*).

3.2.7.2 Espaço de Nomes

Um sistema de agentes móveis deve prover meios de identificar unicamente os agentes, servidores e recursos disponíveis num sistema distribuído. Para cada recurso do sistema é normalmente atribuído um nome. Estes nomes são usados para localizar e identificar estes recursos durante a vida errante dos agentes, de maneira mais amigável e legível (*user friendly e human readable*). Recursos são unicamente identificados por referências dependentes de protocolo e de localização, seqüências de números e caracteres, com significado aderente aos detalhes de

implementação do sistema. De forma a facilitar o uso destas referências por usuários comuns, a cada referência é associado um nome de mais fácil compreensão. Um exemplo é a associação de nomes a endereços IP (*Internet Protocol*), implementado no DNS (*Domain Name Service*). Da mesma forma, um SAM deve prover mecanismos que permitam a correta conversão de nomes para referências de recursos e agentes do ambiente distribuído. Este processo é conhecido como resolução de nomes.

Nomes podem ser dependentes ou independentes de localização. O uso de nomes dependentes de localização em referências a agentes móveis, permite que estes objetos sejam localizados sem que haja etapas adicionais em sua resolução. Isto ocorre em detrimento de um maior comprimento destes nomes, que passam a incluir dados como o nó onde o agente se encontra. Nomes independentes de localização são menores mas, requerem etapas de comunicação adicionais na resolução de referências a objetos que migraram. Por outro lado, nomes dependentes de localização precisam ser atualizados toda vez que um objeto movimenta-se. No caso de nomes independentes de localização, a atualização dos endereços a eles associados é realizada de forma transparente para quem os utiliza. Isto ocorre através de adoções de políticas de atualização de servidores de nomes, ou através do uso de *proxies* locais, cujas referências são mantidas atualizadas pelo sistema [DLMM93].

3.2.7.3 Segurança

Em uma rede local completamente isolada, contida inteiramente em uma organização, é possível confiar em todas as máquinas e no software nelas instalado [KT98b]. Nestes sistemas, agentes podem migrar livremente entre as máquinas. Em redes abertas, como a Internet, entretanto, agentes podem pertencer a domínios administrativos diferentes, não tão confiáveis. Agentes podem ser maliciosos, ou possuir erros de programação que poderiam interferir no adequado funcionamento dos sistemas que os hospedam. De maneira a lidar com estes problemas, um SAM deve satisfazer os seguintes requisitos de segurança, cujos detalhes são descritos em [KT98b]:

- Privacidade e integridade dos dados e do código fonte transportado
- Autenticação de agentes e servidores
- Autorização e controle de acesso a recursos como processamento, disco e memória
- Auditoria e métricas que permitam monitorar o comportamento dos agentes

3.2.7.4 Tolerância a Falhas

Um mecanismo de tolerância a falhas muito usado em SAMs é a realização de *checkpoints* (pontos de sincronização) onde o estado do agente é salvo em uma memória não volátil. Esta memória pode ser um repositório especial em um hospedeiro onde o agente esteve, ou pode ser um servidor dedicado, separado especialmente para coletar este tipo de dado.

Se um agente (ou hospedeiro) falha, o proprietário do agente (*agent owner*) pode realizar o procedimento de recuperação de falhas utilizando os dados coletados durante o último *checkpoint* do agente. Estes dados podem ser usados na criação de um outro agente. Os servidores que compõem o sistema de agentes podem ainda manter uma trilha de auditoria, de maneira a rastrear o progresso deste objeto, provendo meios de determinar as possíveis causas de sua falha [KT98b].

Falhas ou exceções que não podem ser tratadas pelo próprio agente são normalmente repassadas para o *agent owner*, de maneira que este último tome as devidas providências. Uma outra alternativa é enviar o agente para seu *owner* para que este o inspecione e determine a causa de sua falha [KT98b].

O paradigma de agentes móveis permite encapsular o estado envolvido na computação distribuída em um único componente, o agente, que pode ser facilmente rastreado, sincronizado (*checkpointed*) e, eventualmente, recuperado. Tudo isso realizado localmente, sem o conhecimento do estado global do sistema [CPV97]. Esta característica faz com que o paradigma de agentes móveis seja singularmente tolerante a falhas, permitindo que haja uma maior disponibilidade de recursos no sistema.

3.2.8 Linguagens de Programação

Um agente deve trafegar em uma rede composta (possivelmente) por nós pertencentes a arquiteturas de hardware e sistema operacional diferentes. Linguagens de programação para estas aplicações devem ser capazes de executar em diferentes plataformas de hardware e software. Devido a este requisito, as linguagens de programação interpretadas são as preferidas. Estas executam em máquinas virtuais ou interpretadores portados para cada plataforma que integra a rede.

Segurança é outro fator importante. Linguagens que suportam checagem de tipos, encapsulamento, e acesso restrito à memória e a sistemas de arquivos são preferidas por ajudarem a isolar o sistema de erros de programação e acessos não permitidos. São exemplos de linguagens interpretadas usadas em sistemas de agentes móveis: TCL, *shell scripts*, Phyton, Perl, Telescript e Java.

Linguagens como Java e Telescript, que provêm mecanismos para a implementação de sistemas de agentes móveis são classificadas por alguns atores como linguagens de código móvel (*mobile code languages* -MCLs) [CPV97].

3.2.8.1 Java

Java é uma linguagem orientada a objetos desenvolvida pela Sun com o intuito de ser portátil entre várias plataformas de hardware e software, além de permitir sua integração com a Internet

[Flanagan99]. De maneira a ser portátil, o código Java é compilado para uma linguagem intermediária chamada *byte code* que, nada mais é, que a linguagem *assembly* da máquina virtual Java (*JVM – Java Virtual Machine*). Esta máquina virtual representa uma camada de software independente de plataforma, onde programas escritos em *byte code* podem ser executados. A JVM está disponível para várias plataformas de hardware e software, permitindo a execução de aplicações Java em diversas arquiteturas. A integração com a Internet é possível através da geração de um código objeto compacto, que pode ser carregado de forma dinâmica, e sob demanda, a partir de vários servidores distribuídos. O ambiente de execução Java (*Java Runtime Environment*) é hoje disponível para as vários sistemas operacionais e arquiteturas, a exemplo de Windows32, Linux, Solaris, Macintoch e outros.

Java incorpora também grande parte dos recursos de programação orientada a objetos disponíveis em linguagens como C++. Estes recursos foram incorporados a uma linguagem fortemente tipada, livre de operações de mais baixo nível, como operações com ponteiros. A linguagem Java incorpora ainda uma API rica em estruturas de dados e bibliotecas reutilizáveis e genéricas, facilitando a implementação de aplicações de rede (TCP/IP, CORBA e RMI), interfaces com o usuário final (Swing e AWT), criptografia, compressão de dados, manipulação de imagens, som, e muitos outros.

3.2.8.2 Java e Agentes Móveis

A plataforma Java facilita bastante a implementação de mobilidade e portabilidade de código. A portabilidade é alcançada através do uso dos *byte codes e* da JVM. A mobilidade de código é implementada através do uso do conceito de *dynamic binding* (Alocação dinâmica de objetos em tempo de execução). Em Java, cada classe é compilada em um arquivo *.class* separado. Estes arquivos são carregados na memória, sob demanda, no momento em que um objeto é instanciado (usando comando *new* da linguagem). Arquivos *.class* podem estar em máquinas diferentes e podem ser acessados usando o *class loader,* um mecanismo do ambiente de execução Java responsável por carregar código objeto (arquivos .class) de diferentes localizações.

O recurso de serialização, na linguagem Java, permite salvar objetos no formato de *byte streams.* Estes *streams* podem ser salvos em arquivos e posteriormente transportados e convertidos novamente em objetos através do processo de des-serialização. *Byte streams* podem, ainda, ser enviados para outro nó, através de uma conexão de rede.

Uma forma alternativa de enviar um objeto Java, contendo informações sobre o estado do agente corrente, através de uma conexão de rede é passá-lo como parâmetro de uma operação RMI (*Remote Method Invocation*). Esta operação, contudo, não permite a passagem por valor do próprio objeto (*self-reference*), no nosso caso o agente, o que torna a serialização de objetos a forma preferencial de implementação deste recurso na linguagem Java. Em sistemas que utilizam migração fraca, entretanto, o uso de RMI é uma alternativa atraente.

Capítulo 4

Arquitetura WONDER

Descrevemos neste capítulo os principais elementos da arquitetura de software WONDER (*Workflow ON Distributed EnviRonment*), destacando suas características. Os elementos são descritos com base nos requisitos de SGWFs de workflows convencionais e de larga escala listados no capítulo anterior. Os elementos da arquitetura WONDER são descritos de forma independente de hardware e plataforma. O mapeamento destes elementos para a plataforma CORBA é descrita no capítulo de implementação.

4.1 Modelo Distribuído

A arquitetura WONDER [FWME00; FWME99a; FWME99b] foi desenvolvida tendo como base a idéia de que cada caso é um agente móvel, que migra de um nó a outro conforme as atividades vão sendo desempenhadas. Este agente encapsula os dados das atividades e o plano do caso corrente. Uma cópia do caso migra para um nó de um ator (usuário) particular quando este é selecionado para o desempenho da atividade corrente. Ao final de cada atividade, o agente seleciona o usuário que irá desempenhar a próxima atividade, migrando, em seguida, para seu nó preferencial. Esta arquitetura atende aos requisitos de escalabilidade e disponibilidade, descritos no capítulo de Workflow. O caso móvel não necessita de um servidor central que, tradicionalmente armazenaria os dados das atividades e seria responsável por eleger um usuário para desempenhar a próxima atividade. Esta abordagem elimina gargalos de desempenho associados a este servidor central.

Alguns componentes foram adicionados a este modelo de forma a lidar com os outros requisitos dos sistemas de gerenciamento de workflow, em especial, SGWFs de larga escala. De maneira geral, planos costumam especificar papéis, e não usuários, para desempenhar uma determinada atividade. Seja o exemplo de um processo de checagem de crédito. O plano especificará

que um avaliador de crédito, e não um usuário específico, deverá desempenhar esta atividade. De forma a permitir esta associação dinâmica de atores (*actor dynamic binding*), um Coordenador de papel, contendo informações que associam atores a papéis específicos, foi definido. Desta forma, no exemplo anterior, o caso consulta o Coordenador de Avaliadores de Crédito e requisita um usuário para desempenhar esta atividade. Uma vez determinado um usuário, o caso move-se para o nó preferencial deste ator (especificado em suas configurações).

Monitoramento é um outro requisito atendido pela arquitetura WONDER. Como descobrir, sem realizar *broadcast*, o estado corrente de um caso (Atividades que estão em execução em um determinado momento, assim como seus estados internos)? O caso pode estar sendo executado em qualquer um dos nós da rede. Foi definido, desta forma, um Coordenador de Caso. Este componente monitora a execução das atividades do caso, através do recebimento de notificações, durante sua movimentação pelos nós da rede. Cada vez que o caso move-se para um novo nó, este envia pequenas notificações assíncronas ao seu coordenador de caso associado. Desta forma, o coordenador do caso armazena informação a respeito das atividades em execução no momento, assim como onde estas atividades estão sendo executadas.

Outra questão importante, para esta arquitetura baseada em casos móveis, é a recuperação de falhas. O caráter distribuído da arquitetura introduz vários possíveis pontos de falhas. Como os casos são independentes entre si, a falha que afeta um caso, normalmente, não afeta a execução de outras instâncias de processos, permitindo, desta forma, o isolamento das falhas da rede. O que acontece com o caso se o nó onde ele está sendo executado falha? De forma a tratar destes problemas, algumas políticas de redundância foram utilizadas. Para uma eventualidade de uma falha (de hardware ou software) em um nó, o caso armazena, durante sua execução, cópias persistentes do seu estado nos nós por onde este passou (*checkpointing*). Assim que a falha é detectada pelo coordenador de caso, este elege um novo usuário/nó para refazer ou continuar a atividade interrompida, valendo-se do estado do caso armazenado no nó anterior. Além disso, de maneira a evitar o armazenamento desnecessário de estados antigos do caso, o coordenador do caso pode indicar nós para transferir estes estados persistentes para um servidor de backup, liberando o espaço ocupado por estes dados nas máquinas do sistema. Falhas de enlace de comunicação são tratadas através do uso de transferência de informações transacional entre os nós da rede. Em caso de perda de dados, estes são retransmitidos de maneira a garantir a atomicidade desta transferência.

A detecção de falhas é realizada pelo coordenador de caso e pelas atividades que estejam em execução, tipicamente migrando para o próximo nó da rede. Esta detecção utiliza mecanismos de *timeout*, ou *pings* periódicos. Falhas de coordenadores de casos são detectadas por coordenadores de processos e falhas de coordenadores de processos são detectadas por coordenadores de atividades. A resolução destas falhas pode ser feita de forma automática ou através de intervenção humana (gerente do sistema).

A interoperabilidade com outros SGWFs é realizada por atividades *Gateway* que realizam a conversão bidirecional de planos e dados entre a arquitetura WONDER e outros SGWFs. Semanticamente, esta atividade permite a execução de sub-workflows, executando em outros SGWFs, como atividades da arquitetura WONDER.

A interação com o usuário final ocorre através de interfaces gráficas que utilizam componentes *TaskList*. Estes componentes operam como ponte entre o estas aplicações e a arquitetura WONDER.

ActivityManagers controlam a execução das tarefas e das aplicações utilizadas durante cada atividade.

A migração do caso ocorre de forma dinâmica, durante a execução das atividades correntes, levando em conta fatores como carga de atividades e disponibilidade de recursos e atores. Esta característica facilita a implementação de políticas de reconfiguração dinâmica e tratamento de exceções. Em determinadas instâncias de processo, que estejam em sua fase inicial, a reconfiguração pode ser feita simplesmente alterando o plano do caso corrente. Esta tarefa é realizada pelo coordenador de caso. Alterações mais profundas no plano podem, contudo, envolver operações mais complexas como a criação de atividades de compensação ou anulação de atividades anteriores. O tratamento de exceções é realizado de forma semelhante, muitas vezes envolvendo a mudança do plano, para o caso corrente, através da adição de atividades de compensação a este caso.

Operação desconectada é uma conseqüência secundária na arquitetura proposta. Por carregar o plano e os dados do processo, o caso pode ser configurado para suportar períodos de desconexão. Este pode ser configurado para esperar até que a uma conexão de rede esteja presente, de maneira a enviar suas notificações ao coordenador de caos, consultar outros coordenadores, assim como migrar para o nó da próxima atividade. Nestes cenários, uma cópia dos binários do sistema deve estar instalada no sistema.

O compartilhamento de dados, ou a integração do sistema com um banco de dados central ou distribuído, não é responsabilidade da arquitetura aqui proposta. O WONDER supõe que a responsabilidade do controle e da utilização de dados compartilhados é feita pelas próprias aplicações invocadas, chamadas pelo caso móvel, durante a realização de cada atividade. O mesmo ocorre com atividades colaborativas como teleconferências, edições de texto colaborativas, sistemas de apoio a discussões de grupos e outras aplicações de *Groupware*. Nestes casos o controle é também responsabilidade das aplicações utilizadas durante as atividades, externas ao SGWF.

Em sua essência, a arquitetura WONDER é composta por uma hierarquia de servidores e responsabilidades, onde cada tipo de servidor (coordenador) gerencia um subconjunto de objetos. Estes servidores compreendem os coordenadores de papéis, de caso e de processo, e os servidores de backup e de histórico, de forma a dar apoio à execução e migração dos casos móveis. Esta abordagem, baseada em servidores descentralizados, elimina os gargalos de rede e de processamento dos workflows centralizados tradicionais.

Em contrapartida, a descentralização é conhecida por aumentar a comunicação (via rede) entre os servidores distribuídos, um problema que precisa ser investigado com mais detalhe. Por exemplo, um coordenador de caso representa uma instância de processo e recebe apenas pe-

quenas notificações, assíncronas, dos "agentes móveis". Estas notificações informam apenas o estado corrente do objeto. Por outro lado, o servidor de backup recebe uma grande quantidade de dados. Esta transferência de dados, entretanto, é realizada em horários de baixo uso da rede e das estações de trabalho, ou quando a carga neste servidor permite sua realização. O único servidor, no ponto de vista tradicional (cliente-servidor), é o coordenador de papel. Este servidor recebe uma consulta e precisa retornar uma resposta para que o processo de migração possa continuar. O volume de dados trocado, entretanto, é pequeno, compreendendo uma *string* de consulta e uma lista de usuários como resposta. Dessa forma, como a maioria das mensagens trocadas são curtas e assíncronas, a comunicação não é um problema.

4.2 Componentes da Arquitetura

A arquitetura é composta por objetos distribuídos autônomos descritos a seguir. Suas principais colaborações e seus relacionamentos são mostrados na Figura 11 a seguir.

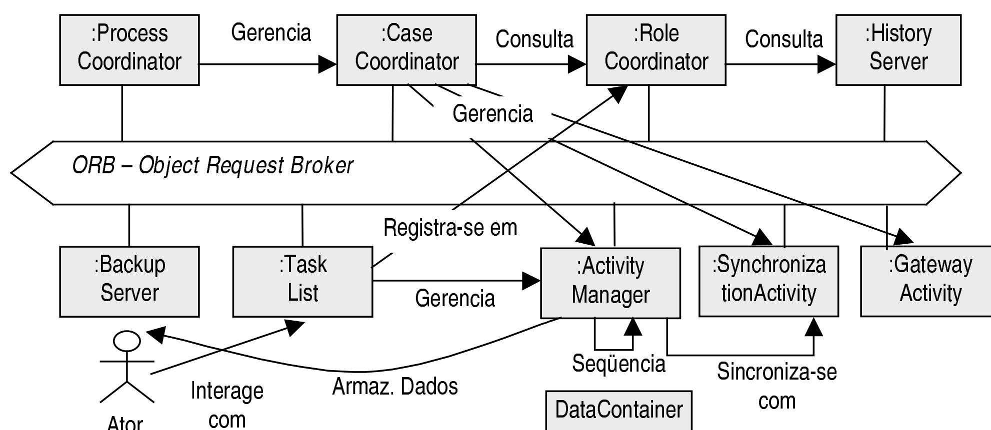

Figura 11: Os principais Componentes da arquitetura e seus relacionamentos

Na próxima seção são apresentados, em mais detalhes, os principais componentes da arquitetura. São utilizados para isso diagramas CRC (*Case Responsibility Colaboration*) [FS97 pp. 65], uma notação que permite identificar de maneira resumida, as principais responsabilidades de cada componente, destacando seus relacionamentos com a outras entidades da arquitetura. Sempre que possível, serão descritos os requisitos que estes componentes pretendem satisfazer.

4.2.1 Coordenador de Processo

O coordenador de processo é responsável pela criação e gerenciamento de coordenadores de casos. Em resposta a uma requisição de compra de um novo suprimento para escritório, por

exemplo, o "coordenador de processo de compra de suprimentos de escritório" criará um novo coordenador de caso para este pedido, transferindo o plano e uma cópia dos dados iniciais do processo para este novo servidor. Para cada caso, um novo coordenador de caso é criado.

Para ocasiões onde torna-se necessário a localização de todas as instâncias de um processo, o coordenador de processo mantém informações sobre e monitora a execução de todos os coordenadores de casos em execução, isto é realizado através do recebimento de notificações periódicas dos coordenadores de casos. Por exemplo, se uma definição de processos é alterada, digamos, para introduzir uma nova atividade, o coordenador de processo pode propagar esta alteração para todos os casos ativos.

Falhas envolvendo coordenadores de casos são tratadas pelo coordenador de processo que podem, abortar o caso corrente ou criar um novo coordenador de caso usando dados armazenados no servidor de backup. Para casos onde há falhas do coordenador de processo, políticas de replicação podem ser utilizadas. A replicação dos coordenadores de processos não afetariam o desempenho dos coordenadores de caso visto que estes enviam apenas pequenas notificações assíncronas a estes coordenadores.

Coordenador de Processo	
Responsabilidades	*Colaborações*
Criação de coordenadores de casos; Alteração dinâmica e estática da definição de processo dos Coordenadores de Casos; Gerenciamento e monitoramento de coordenadores de casos; Receber eventos de coordenadores de casos	Servidor de Backup; Coordenadores de casos; LOA

Tabela 1: CRC do Coordenador de processo

4.2.2 Coordenador de Caso

O coordenador de caso centraliza todas as informações relacionadas a uma instância de processo em execução. Este componente é responsável pela detecção de falhas nas atividades do caso, coordenando os procedimentos de recuperação de suas falhas. O coordenador de caso é também responsável pela execução do procedimento de finalização de uma instância de processo, realizando a coleta (*garbage collection*) de dados e estados de execução de atividades finalizadas que foram armazenados, durante a execução do caso, nos nós do sistema. Durante este procedimento, estas informações são filtradas e armazenadas no servidor de histórico apropriado. O coordenador de caso provê informações relacionadas ao estado corrente do caso, notificando o coordenador de processo ao final do caso, dando acesso às atividades em execução corrente e outras ações de gerenciamento.

Semelhante aos coordenadores de processo, os coordenadores de caso podem ser replicados em nós diferentes da rede de maneira a aumentar sua tolerância a falhas. Esta replicação afeta muito pouco o desempenho dos casos móveis do sistema visto que as notificações recebidas por estes coordenadores são assíncronas e de tamanho reduzido.

Por questões de desempenho, podem ser utilizadas várias políticas de monitoramento do caso. Podem ser utilizados um coordenador de caso por instância de processo, ou um coordenador de caso por conjuntos de instâncias de um mesmo processo. O uso de um coordenador por caso aumenta a tolerância a falhas mas utiliza mais recursos do sistema, ao passo que o uso de um coordenador por conjunto de instâncias de processo requer uma estrutura de controle interna mais complexa e *multithreading*.

Coordenadores de casos podem ser criados em quaisquer nós do sistema distribuído contudo, por se tratar de um componente de gerenciamento importante, devem ser configurados para serem executados em máquinas mais confiáveis, distribuídas de maneira estratégica pelo sistema distribuído. A escolha destas máquinas pode obedecer políticas de balanceamento de carga.

Coordenador de Caso	
Responsabilidades	*Colaborações*
Criação e seqüenciamento da primeira atividade de um caso e das atividades de sincronização; Alteração dinâmica e estática da definição de processo dos ActivityManagers; Gerenciamento e monitoramento de atividades; Recuperação de falhas; Garbage Collection de atividades e finalização do caso, armazenando dados no Servidor de Histórico	Servidor de Backup; Coordenadores de papéis; Coordenador de processo; *TaskLists* dos usuários; Atividades (sincronização, gerenciador, *gateway*); LOA; Servidor de Histórico

Tabela 2: CRC do Coordenador de caso

4.2.3 Coordenador de Papel

Este coordenador é responsável pela localização de atores (pessoas ou serviços que desempenham um determinado papel) de acordo com um critério fornecido. Funciona de maneira semelhante a um *trader*. Tal critério de busca pode refletir dados dinâmicos ou históricos do sistema.

Cada coordenador de papel mantém informações dinâmicas relacionadas a usuários que podem desempenhar um determinado papel. A estes dados são adicionadas informações sobre a carga de trabalho de um ator, em um determinado momento. Para tal, os coordenadores de papéis consultam regularmente os *TaskLists* dos usuários a eles associados, inspecionando sua lista de tarefas pendentes. De posse destes dados, o "coordenador de programadores", por exemplo, pode responder perguntas como "qual o programador com menos carga de trabalho?" ou "Quais são os programadores disponíveis no momento?".

Coordenadores de papéis podem ainda ter acesso a informações relacionadas a casos já encerrados. Estes dados são armazenados nos Servidores de Histórico e em bancos de dados corporativos. Com o auxílio destes servidores, coordenadores de papéis podem responder perguntas como: "Quem é o programador com mais experiência naquele tipo de sistema?" ou "Quem era o programador que implementou a versão anterior daquele programa?".

Cada coordenador de papel pode ainda ser replicado de maneira a prover maior tolerância a falhas ou melhor distribuição da carga de processamento de consultas. Algumas resoluções de nomes envolvendo consultas a dados históricos podem levar algum tempo, podendo atrasar um pouco o processo de migração do caso.

Coordenador de Papel	
Responsabilidades	*Colaborações*
Resolver atores e papéis; Coletar dados de carga dos atores nos TaskLists; Realizar consultas avançadas no Servidor de Histórico	Servidor de Backup; Servidor de Histórico; TaskLists dos usuários; Coordenadores de Casos; Atividades (sincronização, gerenciador, *gateway*); LOA

Tabela 3: CRC do Coordenador de papel

4.2.4 Atividade de Sincronização (Synchronization Activity)

Or-joins e *and-joins* são um problema particular na arquitetura baseada em agentes móveis. Cada *join* de um caso precisa ser criado antes que este inicie. De outra maneira, o agente móvel não poderia saber aonde ir quando este precisasse sincronizar-se com outros agentes móveis que estivessem sendo executados em paralelo num mesmo plano. A atividade de sincronização é responsável por esperar por todas as notificações (*and-join*) ou por uma ou mais *notificações (or-join*), vindas de suas atividades incidentes, antes de criar a atividade que o sucede. Por exemplo, durante um *and-join*, uma vez que todos os agentes móveis tenham enviado notificações a uma atividade de sincronização, este objeto coleta todos os dados de entrada, fornecidos pelas atividades incidentes, e compõe um único agente que é movido ao nó onde a próxima atividade será executada. Em uma sincronização do tipo *or-join*, o número de notificações necessárias para criar a próxima atividade dependerá da expressão lógica a ser avaliada durante a sincronização. De uma maneira geral, o primeiro caso a chegar irá disparar o processo de seqüenciamento da próxima atividade.

Uma atividade de sincronização pode ainda esperar por sinais externos ao workflow. Uma atividade que envolva o uso de um determinado recurso, por exemplo, uma sala de reunião ou um supercomputador, deve esperar até ser notificada da liberação deste recurso. Este evento é externo ao sistema, independendo do desempenho anterior das atividades do caso.

Atividades de sincronização podem ser criadas em quaisquer máquinas do sistema entretanto, de maneira a tornar o sistema mais tolerante a falhas, devem ser criadas em máquinas mais confiáveis. Esta atividade deve ser monitorada constantemente pelo coordenador de caso de maneira a detectar falhas. A falha de uma atividade deste tipo pode implicar na criação de outra atividade de sincronização e atualização dos planos de todos os agentes móveis de um caso.

Atividade de Sincronização	
Responsabilidades	*Colaborações*
Sincronizar Atividades (de sincronização, gateway e de gerenciamento); Receber eventos externos; Seqüenciar atividade seguinte; Notificar Coordenador de caso; Recuperar dados de atividades anteriores; Fornecer dados para atividades posteriores e servidores de backup	Servidor de Backup; Atividades (gateway, sincronização e de gerenciamento); Coordenador de caso; Coordenadores de papéis; *TaskList;* LOA

Tabela 4: CRC da Atividade de Sincronização

4.2.5 Atividades Roteadora (*Gateway Activity*)

A capacidade de integração entre SGWFs diferentes é um problema a ser considerado por estes sistemas. Cada SGWF possui sua própria arquitetura e linguagem de especificação de planos, o que torna sua integração um problema não trivial. A *GatewayActivity* permite integrar processos em execução na arquitetura WONDER com outros SGWFs. Este objeto deve converter dados e a definição do processo para os formatos nativos de outros SGWFs de maneira a utilizar estes sistemas como sub-workflows que são integrados ao sistema WONDER. O processo de conversão inversa pode ainda ser suportado.

Atividade Roteadora (GatewayActivity)	
Responsabilidades	*Colaborações*
Converter dados do caso de/para outros formatos; Seqüenciar atividade seguinte; Notificar Coordenador de caso; Recuperar dados de atividades anteriores; Fornecer dados para atividades posteriores e servidores de backup	Atividades (gateway, sincronização e de gerenciamento); Coordenador de caso; Coordenadores de papéis; *TaskList*; Servidor de Backup; LOA

Tabela 5: CRC da Atividade Roteadora

4.2.6 Lista de Tarefas (*TaskList*)

Em geral, a interface dos usuários finais com um SGWF é implementada na forma de uma lista de tarefas, similar a um leitor de e-mails. A lista de tarefas informa ao usuário sobre as novas atividades que ele deverá desempenhar. Isto permite ao usuário aceitar ou rejeitar as novas atividades a ele sugeridas, de acordo com a política de alocação de atividades corrente. Além do mais, como a lista de tarefas representa a principal interface dos usuários com o SGWF em si, esta deve permitir a configuração de parâmetros do sistema. São exemplos de alguns destes parâmetros: aplicações preferidas para executar determinado tipo de tarefas, nó preferencial, seleção de políticas de ordenação de tarefas e outros.

O componente *TaskList* definido na arquitetura WONDER tem como propósito ser usado por estas aplicações/GUIs com o usuário final como forma de integração destes sistemas com a arquitetura WONDER. Este componente realiza uma ponte entre a arquitetura e a interface com o usuário final. O *TaskList* é responsável também por coletar informações relacionadas à carga de trabalho do usuário, de forma a ser consultada pelos coordenadores de papéis.

Lista de Tarefas	
Responsabilidades	*Colaborações*
Gerenciar lista de tarefas de um ator; Negociar aceitação de atividade; Informar estado da lista e dados do usuário; Iniciar atividades	Atividades (sincronização, gateway e de gerenciamento); Coordenador de caso; Coordenadores de papéis; Atores; LOA

Tabela 6: CRC da Lista de Tarefas

4.2.7 Servidor de Histórico (*HistoryServer*)

O servidor de histórico (ou servidores) é um *front-end* para o repositório de casos já completados. Quando um coordenador de caso finaliza seu trabalho (um caso chega ao final), todos os dados relevantes, usados pelo caso, são armazenados no servidor de histórico. Este procedimento permite que casos possam sofrer auditoria, fazendo com que a memória do caso seja armazenada para posterior consulta.

Servidor de Histórico	
Responsabilidades	*Colaborações*
Armazenar dados de histórico dos casos; Realizar consultas a bancos de dados	Servidor de Backup; Coordenadores de papéis; Coordenador de caso; Coordenador de processo; LOA

Tabela 7: CRC do Servidor de Histórico

4.2.8 Servidor de Backup (*Backup Server*)

De forma a tornar o sistema mais tolerante a falhas, são utilizados servidores especiais chamados Servidores de Backup. Tais servidores são escolhidos dinamicamente, ao início de cada caso, de acordo com informações relacionadas à disponibilidade, carga, confiabilidade e outros dados destes servidores, que estejam ativos no sistema.

O servidor de backup (ou servidores) é um *front-end* para o repositório de dados intermediários dos casos em execução no sistema. Como mencionado anteriormente, a informação sobre os estados passados dos "agentes móveis" são armazenados em alguns dos nós onde o caso executou. Estes nós não são confiáveis para armazenar este estado de execução, de forma indefinida, nem pode ser assumido que estas máquinas estejam ativas quando esta informação for necessária. O servidor de backup é um objeto que executa em um servidor mais confiável e como maior capacidade de processamento. Este servidor recebe dados e estados de execução das atividades já terminadas de um caso, sob o comando do coordenador de caso. Dependendo da política de tolerância a falhas vigente, uma vez que o backup é realizado, estes estados podem (ou não) ser excluídos dos nós dos usuários.

Servidores de Backup podem ainda armazenar dados de Servidores de Casos e Servidores de processos, introduzindo uma redundância no sistema que pode ser necessária durante o processo de recuperação de falhas destes servidores.

Servidor de Backup	
Responsabilidades	*Colaborações*
Armazenar dados de coordenadores, atividades e servidores; Recuperar dados durante recuperação de falhas	Coordenadores (caso, processo, papéis); Atividades (de gerenciamento, gateway, sincronização); LOA

Tabela 8: CRC do Servidor de Backup

4.2.9 Gerenciador de Atividade (*Activity Manager*)

Até o momento, foi usada a idéia de um agente móvel como uma descrição intuitiva da natureza distribuída do caso. O caso, contudo, não é realmente implementado como um único agente móvel, mas como um conjunto de dados e estado de execução que é trocado entre objetos (servidores CORBA) no sistema. O *ActivityManager* ou Gerenciador de Atividade é o servidor CORBA que implementa a mobilidade do caso na arquitetura WONDER. É um agente móvel que utiliza migração fraca, ou seja, transporta apenas seu estado de execução de um servidor a outro.

Cada gerenciador de atividade coordena a execução de uma atividade, dentre as que compõem cada caso. Quando uma nova atividade necessita ser desempenhada, um novo gerenciador de atividade é criado no nó preferencial do usuário que irá executa-la. Este novo gerenciador é então configurado com os dados necessários para seu desempenho e com o estado do caso vindo da atividade anterior. A interpretação do plano é então ser retomada e a atividade é então executada utilizando as aplicações apropriadas, através do uso dos *wrappers*. O gerenciador de atividade aguarda até ser notificado da finalização do trabalho do ator, para então calcular quem deverá executar a próxima atividade (através da interpretação do plano que vem junto com o estado do caso, e através da consulta do coordenadores de papéis apropriados). Se a próxima atividade deverá ser executada por um usuário humano, o gerenciador de atividade envia a informação apropriada para a lista de tarefas deste ator, notificando o coordenador de caso que a atividade terminou e quem será o usuário selecionado para desempenhar a próxima atividade. Após isso, a informação do caso é transferida para o gerenciador de atividade criado. Atividades automáticas não passam por fase de negociação como o *TaskList*. São criadas diretamente no nó onde serão executadas, como especificado no plano.

A transferência de dados entre atividades consecutivas é realizada de forma transacional, de maneira a evitar erros de transmissão de dados durante este processo, além de garantir a consistência dos dados trocados.

É ainda responsabilidade do gerenciador de atividade receber requisições do coordenador de caso para transferir seu estado ara o servidor de backup.

Gerenciador de atividade (Activity Manager)	
Responsabilidades	*Colaborações*
Gerenciar execução de atividades; Disparar Wrappers; Notificar Coordenador de caso; Seqüenciar novas atividades; Recuperar dados de atividades anteriores; Fornecer dados para atividades posteriores e servidores de backup	Coordenador de caso; Atividades (sincronização, gateway e de gerenciamento); TaskList; Coordenadores de papéis; LOA

Tabela 9: CRC do Activity Manager

4.2.10 Interpretador de Planos (*PlanInterpreter*)

Internamente, o gerenciador de atividade utiliza um interpretador de plano para determinar seu estado corrente e as próximas atividades a serem realizadas. Este interpretador também é responsável por armazenar dados de execução e de desempenho do caso, durante toda sua execução. Esta informação é utilizada tanto pelos coordenadores como pelo agente (atividades). É importante durante procedimentos de recuperação de falhas.

Interpretador de Planos (PlanInterpreter)	
Responsabilidades	*Colaborações*
Gerenciar o estado corrente do workflow; Armazenar dados de execução do caso	Atividades (sincronização, gateway e de gerenciamento); Coordenador de caso;

Tabela 10: CRC do Interpretador de Planos

4.2.11 Gerenciadores de Aplicações (*Wrappers*)

Wrappers são componentes que controlam a execução de uma aplicação invocada em particular. Eles disparam as aplicações com seus dados iniciais apropriados, especificados no plano, e coletam dados de retorno, repassando-os aos gerenciadores de atividades. São, desta forma, uma ponte entre os programas específicos, que auxiliam na realização de cada atividade, e o gerenciador de atividade. Quando uma aplicação finaliza, os *wrappers* notificam seu gerenciador de atividade correspondente.

Gerenciador de Aplicações (Wrappers)	
Responsabilidades	*Colaborações*
Gerenciar execução de aplicações invocadas; Coletar e fornecer dados das aplicações	Activity Manager

Tabela 11: CRC do Gerenciador de Aplicações

4.2.12 Ambiente de Suporte a Objetos do Workflow

De maneira a atender os requisitos de tolerância a falhas e a facilitar a migração das atividades durante a execução do caso, cada nó habilitado para a execução do WONDER possui um ambiente de suporte aos objetos deste SGWF. Este ambiente é composto pelo ativador de objetos, que executa localmente em cada nó, e pelo repositório de objetos. A seguir são descritos, de maneira genérica, estes dois elementos. No capítulo de implementação, estes objetos serão descritos em maior detalhe.

4.2.12.1 Ativador de Objetos Local (LOA – *Local Object Activator*)

O Ativador de objetos tem um papel duplo na arquitetura. Este agente gerencia o ciclo de vida das atividades e coordenadores em execução num determinado nó, implementando sua persistência, criação e localização, provendo um ambiente seguro de execução. Este componente é responsável por guardar o estado destes objetos, em memória não volátil, no nó local. Este trabalho é feito em conjunto com o Repositório de Objetos.

LOA (Local Object Activator)	
Responsabilidades	*Colaborações*
Criar novos objetos; Resolver nomes de objetos em referências; Implementar persistência de objetos;	Atividades (gateway, de sincronização e de gerenciamento); Coordenadores (processo, caso, papéis)

Tabela 12: CRC do Ativador de Objetos Local

4.2.12.2 Repositório de Objetos (*ObjectRepository*)

O caso, durante sua migração entre os nós do sistema, deve poder armazenar dados e seu estado de execução em um local seguro, de maneira a satisfazer os requisitos de segurança do sistema. O Repositório de Objetos desempenha este papel, restringindo o acesso a estes dados, de maneira que somente atores autorizados possam ter acesso a esta informação. Este repositório armazena também o estado de execução dos servidores, coordenadores e das atividades em execução num determinado nó.

Repositório de Objetos	
Responsabilidades	*Colaborações*
Armazenar estados de servidores e coordenadores; Armazenar dados das atividades e coordenadores; Cuidar da segurança dos dados armazenados	LOA Servidor de Backup; Servidor de Histórico

Tabela 13: CRC do Repositório de Objetos

4.3 Cenários de Execução

Nesta sessão, são apresentados alguns exemplos de execução envolvendo os componentes da arquitetura WONDER. Estes exemplos mostram o funcionamento básico do sistema, enfatizando a comunicação e interação entre os principais componentes da arquitetura, em situações genéricas, encontradas em aplicações reais. Por simplicidade, não será apresentada a interação como o objeto LOA nos diagramas da Figura 13 em diante. Esta interação, descrita na Figura 12, ocorre toda vez que um objeto é criado ou reiniciado. Os cenários são descritos utilizando diagramas de seqüência em UML (*Unified Modeling Language*) [BJR97], uma linguagem de modelagem de sistemas orientados a objetos adotada como padrão pelo OMG, amplamente aceita na comunidade de Engenharia de Software.

4.3.1.1 Interação com o LOA (*Local Object Activator*)

A Figura 12 mostra três tipos de interação entre um cliente, a atividade AM3, o Gerente de Ativação (*orbixd*) e o LOA. Esta interação ocorre sempre que a operação *bind()* é invocada através dos *stubs* dos clientes. Esta requisição é, então, repassada para o *orbixd*.

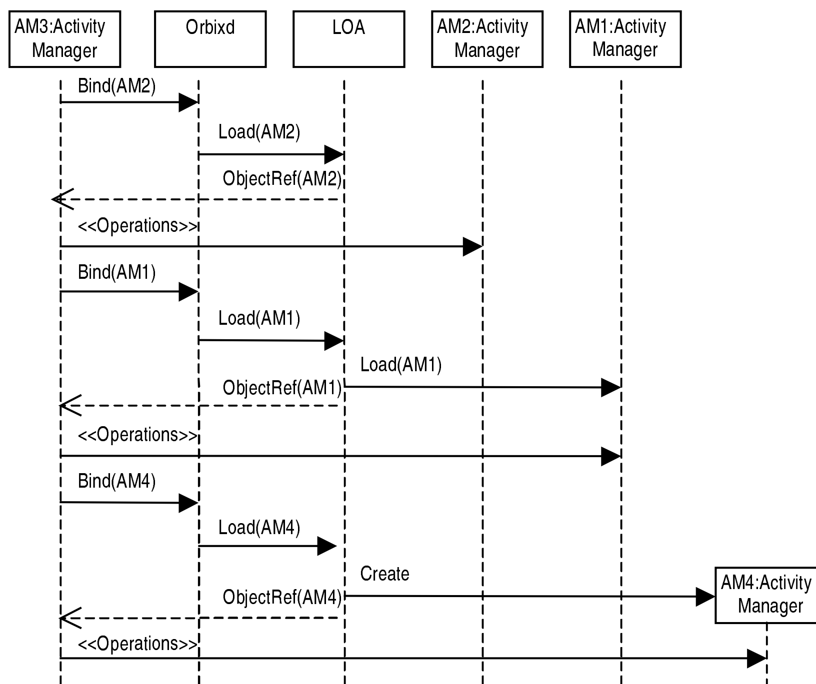

Figura 12: Diagrama de Interação Cliente-LOA

No primeiro caso, AM3 conecta-se a um objeto já existente, o AM2. Neste momento, o *orbixd* redireciona a chamada para o LOA que simplesmente repassa a referência de objetos para quem a requisitou (AM3). No segundo caso, AM3 requisita a conexão com AM1, um objeto que não está ativo na memória, mas que possui seu estado armazenado em disco (Repositório de Objetos). Neste momento, AM1 é acordado pelo LOA e tem sua referência retornada para AM3. Finalmente, no último caso, AM3 requisita a criação de um novo objeto, a atividade AM4. O LOA verifica os objetos ativos na memória e salvos em disco (Repositório de Objetos), como o objeto respondendo pelo nome AM4 não foi encontrado, um novo objeto, com esta nome, é criado. Sua referência é, então, retornada a quem o requisitou (AM3).

4.3.2 Seqüenciamento de Atividades

A Figura 13 mostra um exemplo típico de um procedimento de seqüenciamento. Quando a execução de uma atividade termina, ou seja, quando todas suas tarefas foram completadas (seus *Wrappers* terminaram a execução normalmente), o *ActivityManger* AM2 notifica o *CaseCoordinator* e o *TaskList* (envio das mensagens 5 e 6) e inicia o procedimento de seqüenciamento de uma nova atividade. O coordenador de caso CC1, executando em um nó diferente, recebe uma notificação do tipo "*end of the activity*" (6). A atividade AM2 continua a interpretação do plano do processo corrente e "descobre" qual a próxima atividade a ser realizada e por qual papel. O AM2 consulta o RC1 (o coordenador de papeis correspondente ao papel da atividade seguinte – mensagem 8), o qual seleciona um usuário para realizar a próxima atividade. O AM2 sugere uma nova atividade ao *TaskList*, TL2, do ator correspondente (10). Se o usuário em questão aceita esta oferta de tarefa, o processo de criação da atividade inicia-se (10 e 13). O *ActivityManager* AM2 cria o próximo servidor do tipo *ActivityManager*, AM3, no nó preferencial do ator selecionado (13), transferindo todos os dados necessários sua execução (16).

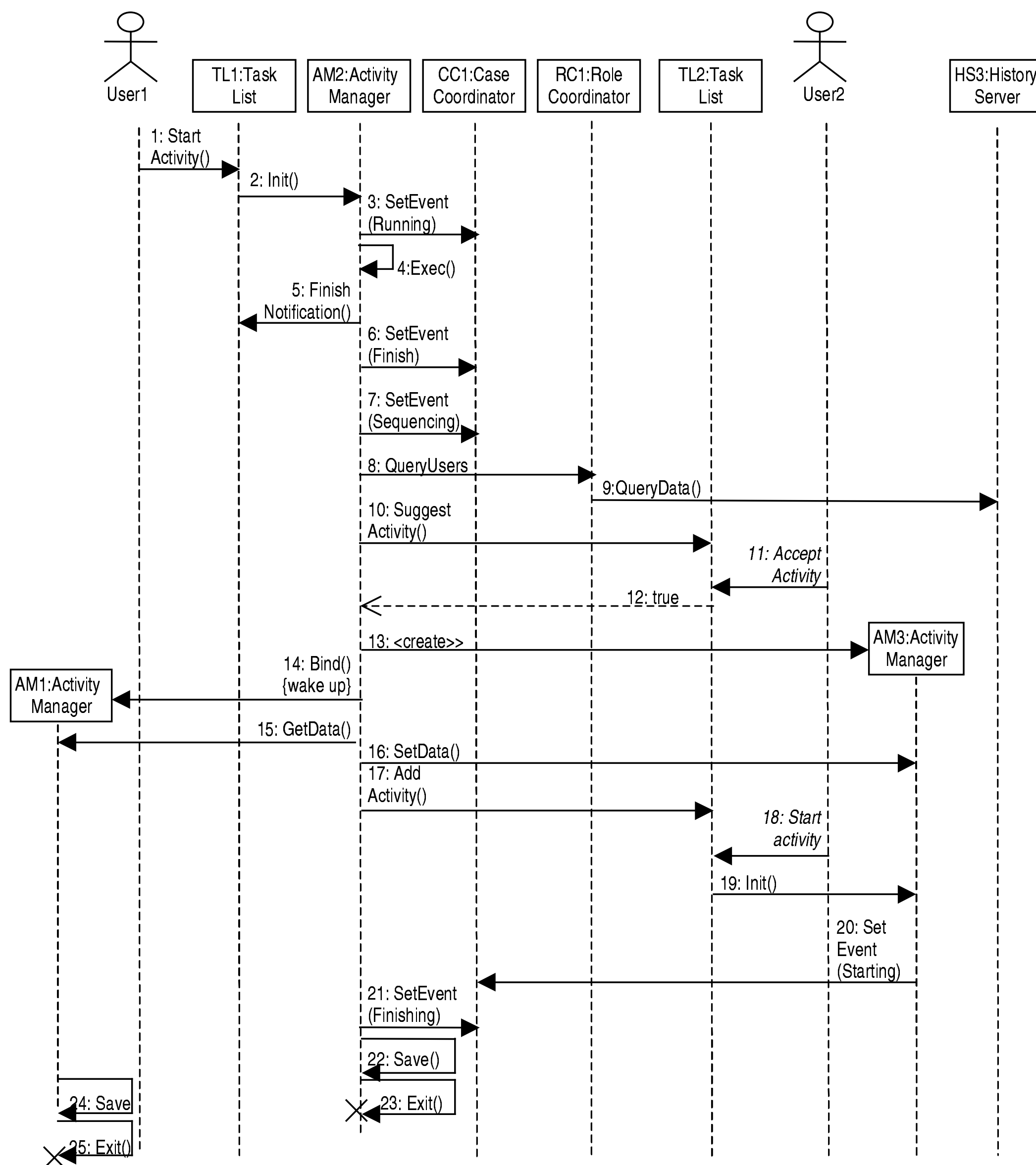

Figura 13: Diagrama de seqüenciamento de atividades.

No exemplo da Figura 13, o AM2 não tem todos os dados necessários à execução de AM3 armazenados localmente. De maneira a fornecer estes dados, o AM2 resolve as referências para estes dados remotos, descobrindo sua localização e os coletando da atividade anterior AM1 (14 e 15). Os dados são “embalados” em um *DataContainer*, juntamente com o estado do caso. Finalmente, a atividade AM3 é inserida na lista de tarefas do User2 (17). AM3 é inicializada e a atividade AM2 é finalizada (21 e 23).

Por questões de desempenho, somente os dados necessários são transferidos. O restante é passado na forma de links, de maneira a serem coletados por atividades subsequentes.

4.3.3 Criação de Um Caso

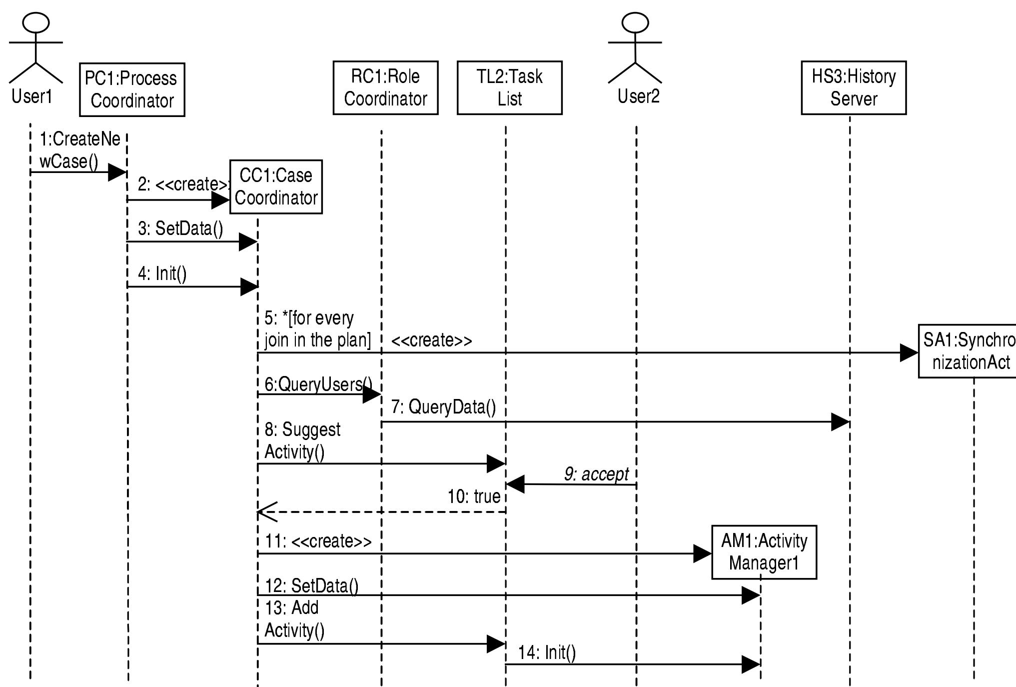

Figura 14: Diagrama de criação de um caso

O procedimento de criação de um caso, apresentado na Figura 14. Este procedimento é iniciado por um usuário responsável pelo caso (User1), que realiza uma requisição ao coordenador de processo PC1 (1). Esta requisição resulta na criação e configuração do coordenador de caso CC1 (2 e 3). O processo de configuração é, então, inicializado e o CC1 cria as atividades de sincronização para seu caso (5). o coordenador de caso CC1 cria a primeira atividade AM1 (11 a 14), após consultar o coordenador de papel RC1, requisitando um usuário que realizar a primeira atividade do caso (User2) e após a aceitação da atividade por este usuário (8 a 10). Neste momento, o caso é então iniciado. AM1 e suas atividades posteriores seguem de maneira independente, executando o caso em questão.

4.3.4 Atividades AND-Split

O *and-split* é implementado como um seqüenciamento paralelo de atividades. O procedimento descrito na Figura 13 é iterado pelo *ActivityManager* corrente para cada atividade do *split*. As atividades assim criadas seguem, então, caminhos independentes até sua posterior sincronização junto a um objeto do tipo *and-join* previamente estabelecido e especificado em seus planos.

4.3.5 Atividades de Sincronização

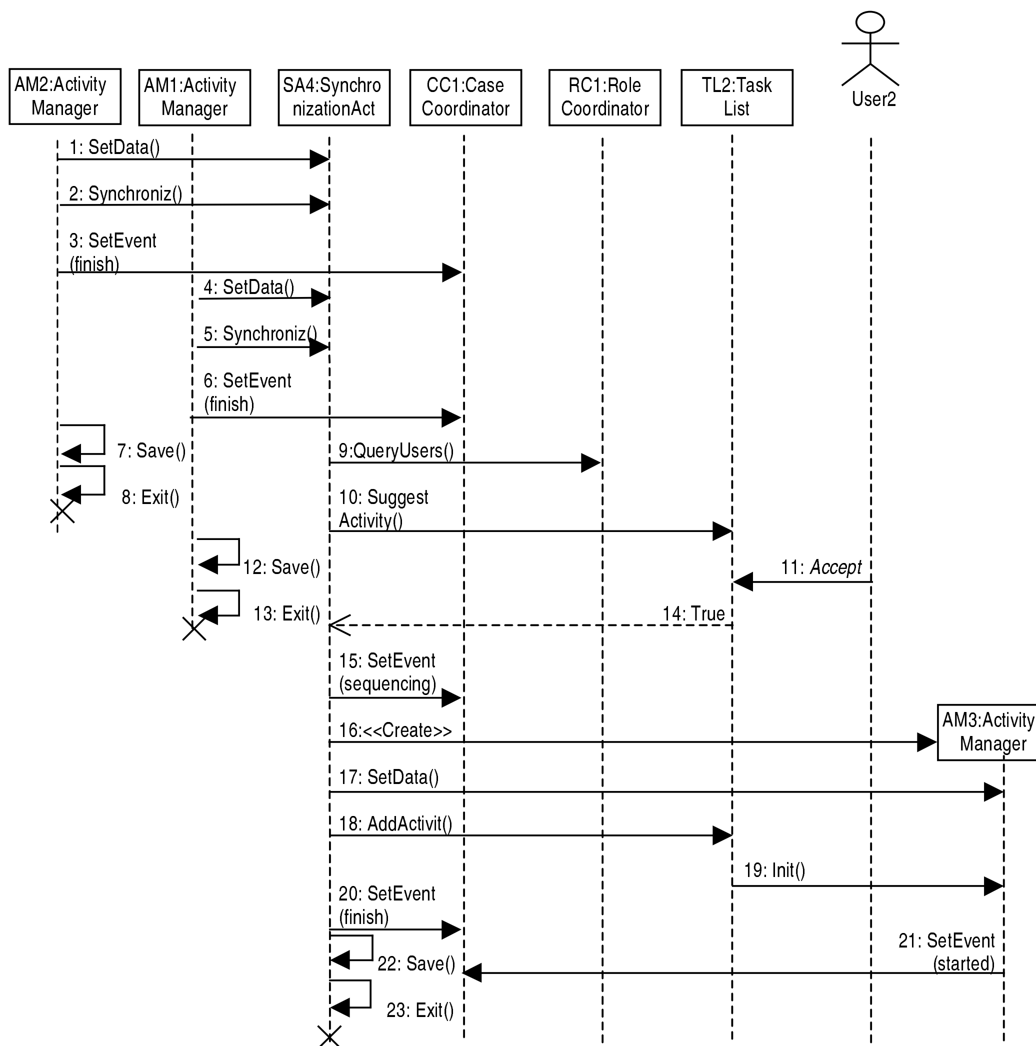

Figura 15: Sincronização AND-Join

As atividades de sincronização são criadas por coordenadores de casos ao início de cada caso. Sua localização é então armazenada como dado do plano que é passado para a primeira atividade do caso. Desta forma, a localização destes servidores é um parâmetro bem conhecido, descrito no plano.

O procedimento de sincronização envolvendo as atividades AM1, AM2 e SA4 é descrito na *Figura 15*. Durante o procedimento de sincronização, cada *ActivityManager* notifica a atividade de sincronização SA4 e o coordenador de caso CC1 (2 e 3). Após ambos os *ActivityManagers* (AM2 e AM1) terem notificado o SA4, este inicializa a atividade seguinte da maneira convencional, como descrito no item Seqüenciamento de Atividades anterior. Como no se-

qüenciamento convencional, o CC1 é mantido informado do progresso do caso, gerenciando e resolvendo as (possíveis) falhas ocorridas.

4.3.6 Finalização do Caso

A Figura 16 apresenta o procedimento de finalização de um caso. Ao final de cada caso, dados armazenados em cada nó, onde pelo menos uma das atividades deste caso executou, e todos os dados armazenados no(s) servidor(es) de backup são removidos pelo coordenador de caso CC3 (9,11 e 13). Um sumário da execução contendo dados de relevantes, possivelmente usados em futuras consultas, são armazenados no Servidor de Histórico HS2 (12).

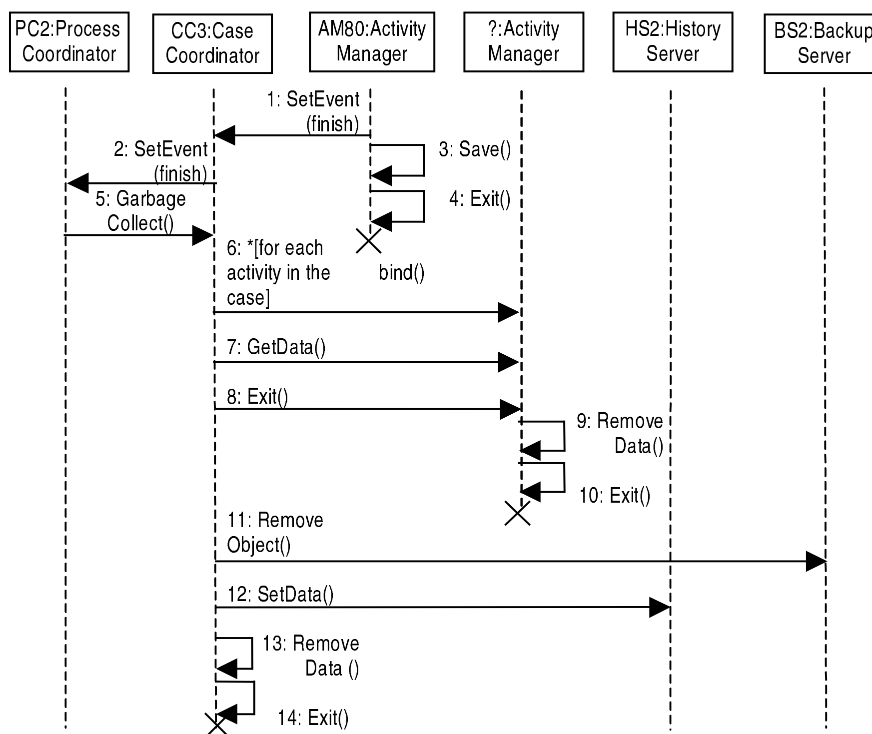

Figura 16: Diagrama de seqüência do procedimento de finalização

4.3.7 Recuperação de Falhas

O procedimento de recuperação de falhas consiste em parar o processo corrente (as atividades de um caso que estão em execução no momento), restaurar o sistema a um estado anterior estável, modificar a definição de processo (adicionando atividades de compensação se for o caso) e finalmente continuar o caso. Esta rotina é gerenciada pelo coordenador do caso em questão, usando dados armazenados no repositório de objetos se cada nó envolvido (*checkpoints*) , assim como também os dados armazenados nos servidores de backup espalhados pelo sistema.

Falhas de atividades de sincronização são detectadas por atividades que estão em fase de sincronização ou pelo coordenador de caso. A recuperação deste componente consiste em criar outra atividade de sincronização, em um nó diferente da rede, e ajustar os planos das atividades correntes para incluírem a nova localização deste servidor.

Capítulo 5

Mapeamento para CORBA e Java

Descrevemos neste capítulo os principais aspectos do mapeamento dos elementos da arquitetura WONDER para o ambiente CORBA, implementado em Java. Destacando os principais fatores e decisões de projeto que permearam este mapeamento.

5.1 CORBA e Java

A arquitetura WONDER foi implementada utilizando os recursos do *framework* de comunicação CORBA (*Common Object Request Broker Architecture*) [OMG95]. Este *framework* foi escolhido por prover um conjunto de funcionalidades e transparências que facilitam o desenvolvimento de aplicações distribuída, como descrito no Capítulo 3.

Embora proveja transparências de acesso (independência de hardware, linguagem de programação e sistema operacional) e localização (independência de onde o objeto será criado), esta última transparência não é muito empregada na arquitetura WONDER. A localização de um componente é um dado importante para a movimentação e o gerenciamento do caso que migra pela rede.

A capacidade de integração de aplicações legadas é outra característica importante de CORBA, utilizado na implementação de SGWFs. Esta característica é implementada pelos *wrappers* da arquitetura WONDER.

A linguagem Java foi escolhida para o desenvolvimento da arquitetura WONDER por suas vantagens relacionadas a implementação de aplicações que utilizam do paradigma de agentes móveis. Dentre suas vantagens podemos citar: suporte a serialização de objetos, independência de plataforma de software e hardware, facilidade de aprendizado da linguagem e de desenvol-

vimento de aplicações, além da previa experiência do autor no desenvolvimento de sistemas utilizando Java.

Para a implementação da arquitetura WONDER foi utilizado o OrbixWeb3.1c da Iona [OrbixPrg98], suas características são descritas no Capítulo 3. O OrbixWeb foi escolhido por suportar o mapeamento IDL para Java, implementando o padrão CORBA 2.0 [CORBA98], além de estar disponível para uso no Instituto de Computação, sendo ainda familiar ao autor. Aliás, a familiaridade no uso deste tipo de *framework* é um fator decisivo para seu uso bem sucedido. Cada implementação possui suas peculiaridades, erros (*bugs*) e facilidades, o que torna seu aprendizado um processo relativamente custoso e demorado.

No início deste projeto, em meados de 1998, existiam outras alternativas implementações de ORBs comerciais como o Visibroker for Java [Visibroker], da Inprise, ou mesmo implementações livres (*free*) como o JacORB [CH97], JavaORB [JavaORB] e o MICO (uma implementação para C++) [MICO]. O Visibroker é um forte concorrente do OrbixWeb da Iona oferecendo, em sua versão atual 4.0, virtualmente os mesmos recursos que sua concorrente, além de serviços de eventos e de segurança, qualidade de serviço (*QoS – Quality of Service*), suporte a migração de objetos, balanço de carga, passagem de objetos por valor e outros. Sua utilização, contudo, requeria a compra de uma licença de utilização.

Na época em que foi iniciado o desenvolvimento do protótipo WONDER, as implementações livres de CORBA para Java ainda não estavam suficientemente maduras, ou não proviam todos os recursos encontrados nas implementações comerciais. Em parte, isto deve-se ao fato destas distribuições livres implementarem somente o núcleo básico do ORB, não provendo facilidades adicionais como os agentes de ativação de objetos (*object activators*) e persistência. Estes recursos adicionais, apesar de não serem padronizados, facilitam imensamente o desenvolvimento de aplicações.

Atualmente, contudo, como o crescimento do movimento de código aberto (e livre) [Opensource; FSF], estas implementações livres tendem a alcançar sua maturidade. As implementações JacORB e, em especial a JavaORB, nos 2 anos que se passaram desde então, foram bastante aprimoradas e incorporaram serviços como: Nomes, Eventos, Propriedades, Persistência e Notificação dentre outros, além de aderência ao padrão CORBA 2.3.

5.1.1 Serviços CORBA

Várias arquiteturas de Workflow baseadas em CORBA utilizam um subconjunto dos Serviços de Objetos definidos na arquitetura OMA [OMG99; WWWD97]. Os serviços mais usados são o Serviço de Nomes, de Eventos, de Notificação, de Segurança e de Transação. Devido aos requisitos de larga escala da arquitetura WONDER, assim como seu uso de objetos móveis, algumas características destes serviços tornam-se inadequadas ou desnecessárias e serão discutidos a seguir.

Algumas implementações de workflow baseados em CORBA utilizam o serviço de transações da arquitetura OMA para coordenar o fluxo de dados entre diferentes servidores [SRD99, WSR98]. Esta abordagem permite criar um protocolo tolerante a falhas de transferência de dados entre atividades, implementando um conjunto de “canais de comunicação transacionais”. Apesar disto, segundo Stewart et. al, “Sistemas de larga escala geralmente necessitam de semântica transacional, mas nem sempre precisam utilizar transações distribuídas” [SRD99].

Na arquitetura WONDER, pares de Gerenciadores de Atividades (*ActivityManagers*) gerenciam a consistência dos dados e notificações que são entre eles trocados. Durante *splits*, este processo é iterado para cada atividade pertencente a seu *fan-out*. Dados são passados de uma atividade a outra usando uma única invocação de operação (*setData()*). Erros são tratados utilizando políticas de retransmissão destes dados. Desta forma, o mecanismo de invocação remota de operações, provido pelo barramento CORBA é suficiente para a implementação da arquitetura WONDER.

Se algum erro ocorre durante a invocação de uma operação remota, devido a um *crash* temporário de um link, por exemplo, o ORB gera uma *SystemException*. Esta exceção é tratada pelo objeto que enviou os dados o qual, de acordo com o motivo da falha, realiza uma nova invocação quando a conexão for restabelecida. Se a falha persistir, o coordenador de caso desempenha o procedimento de recuperação de falhas, criando um caminho alternativo a ser seguido. Esta abordagem dispensa o uso de um servidor de transações.

Os serviços de eventos e notificação permitem desacoplar os servidores produtores e o consumidores de eventos, implementando uma fila de mensagens. Estas mensagens podem ser persistentes em algumas implementações do *COSEvents* e *COSNotification* [Web-1]. Estes sistemas de envio e recebimento de mensagens e eventos são serviços genéricos e, relativamente complexos para os requisitos da arquitetura WONDER. Estes serviços são normalmente utilizados em aplicações complexas de gerenciamento de redes de telecomunicações. O serviço de Notificação, por exemplo, permite definir múltiplos canais de eventos, aos quais podem ser associados filtros. O isolamento entre produtor e consumidor de eventos, provido por tais serviços aumenta a complexidade dos algoritmos de detecção de falhas. Como detectar, usando estes serviços, a falha dos coordenadores de casos e processos, que recebem notificações das atividades?

A arquitetura WONDER não utiliza nenhum serviço de nomes padrão da arquitetura OMA. Este serviço, quando usado de forma centralizada, sem replicação, constitui um único ponto de falhas. Como localizar um objeto se o serviço de nomes falhar? Políticas de federação de servidores de nomes, ou mesmo o uso de vários destes servidores, de maneira descentralizada, poderiam ser adotados. Contudo, uma solução mais simples, porém não padronizada, foi utilizada: Cada nó executa um *locator* ou agente de ativação, que resolve *markers* (nomes associados a objetos utilizados pelo OrbixWeb) para IORs. Este *locator (orbixd)*, operando juntamente com o LOA, é também usado para implementar a persistência e a ativação e desativação de objetos. Optou-se pelo uso destes recursos por estarem disponíveis no OrbixWeb, além de permitirem implementar uma solução completamente descentralizada, estando de acordo com a filosofia da arquitetura WONDER.

5.1.2 Referência a Objetos CORBA

Devido a requisitos de persistência de objetos e mobilidade do caso, o principal problema na utilização de CORBA como suporte ao desenvolvimento de workflow distribuído são suas referências de objetos. O uso das referências CORBA padrão, IORs *(Interoperable Object References)* em conjunto com o BOA (*Basic Object Adapter*), como forma de localização de objetos, não são completamente adequadas à arquitetura proposta. Estas referências devem ser únicas em todo o sistema e, no caso especial da arquitetura WONDER, são normalmente alocadas dinamicamente, várias vezes, para um mesmo objeto. Dentre os vários dados que compõem uma IOR são destacados o endereço IP (ou o nome DNS do nó) e o número da porta que, respectivamente, permitem localizar o nó e o processo correspondente ao servidor CORBA em execução nesta máquina. Como o tempo de execução de uma atividade pode durar de horas a dias, ou até mesmo meses, não pode ser assumido que, durante todo o tempo de vida de um caso, um objeto estará ativo, em execução, na mesma porta onde este foi criado e, portanto, sendo referenciado pela mesma IOR. Na arquitetura WONDER, servidores são constantemente ativados e desativados, cada vez que isto ocorre, uma nova IOR é associada a este objeto.

Recentemente, o OMG aprovou um novo adaptador de objetos, o POA (*Portable Object Adpapter*) [POA97], para substituir seu padrão antigo o BOA. Este novo padrão permite, dentre outros novos recursos, tornar as referências de objeto imunes a diversas ativações e desativações. Este padrão, por ser muito recente, ainda não está disponível na versão do OrbixWeb utilizada para implementação do protótipo WONDER, o que forçou a implementação de um mecanismo alternativo de nomes e localização de objetos.

Durante a fase de projeto e implementação da arquitetura, meados de 1998 e 1999, a especificação CORBA vigente, mantida pelo OMG (*Object Management Group*), ainda não havia especificado um serviço de persistência de objetos. Para contornar este problema, permitindo que um mesmo objeto fosse ativado/desativado durante sua vida, assumindo IORs possivelmente diferentes, foi criado um espaço de nomes proprietário para a arquitetura WONDER. Este espaço de nomes está associado ao mecanismo de persistência de objetos. Neste mecanismo, o estado dos objetos é armazenado localmente em cada nó. Para tal, são identificados de forma única através de um espaço de nomes descrito a seguir:

- (nome da máquina, processo, caso, ator, atividade, arquivo) para arquivos;
- (nome da máquina, processo, caso, ator, atividade) para atividades;
- (nome da máquina, processo, caso) para coordenadores de casos;
- (nome da máquina, processo) para coordenadores de processos;
- (nome da máquina, backup server) para servidores de backup;
- e assim por diante.

De forma a prover persistência de objetos de maneira transparente, cada nó tem um Ativador de Objetos Local (*LOA – Local Object Activator*). O LOA executa como um *hook* (ponto adaptá-

vel opcional em um *framework*), uma classe derivada da interface *LoaderClass* do OrbixWeb. Este *hook* é usado pelo orbixd – *daemon locator* do OrbixWeb. O orbixd executa em uma porta bem conhecida, parametrizada no sistema. Este *daemon* permite ativar objetos dinamicamente, intermediando seu processo de ativação e desativação. Um objeto é criado ou ativado/despertado através de chamadas à operação *bind()* que são repassadas ao orbixd executando no nó onde o objeto será utilizado. O LOA realiza o salvamento do estado dos objetos em uma área local reservada (Repositório de Objetos – *ObjectRepository*). Por exemplo, o coordenador de caso para uma requisição de compra de 500 clipes de papel (caso C4375), do processo “compra de suprimentos para escritório” (Processo P12), no nó abc.def.com é identificado por (abc.def.com, C, P12, C4375). A letra ‘C’ nesta referência, indica ao LOA, que este nome é referente a um objeto do tipo *CaseCoordinator.*

De forma a ter acesso a objetos gerenciados pelo LOA (ou formalmente, realizar um *bind*, estabelecendo uma conexão), um processo precisa enviar a referência (abc.def.com, C, P12, C4375) para o orbixd que, imediatamente, o repassa para o LOA da máquina abc.def.com. O LOA restaura o estado do objeto colocando o servidor CORBA novamente de pé. Esta ativação usa informações previamente armazenada no repositório de objetos. O LOA então retorna a IOR do objeto recém restaurado para que este seja imediatamente utilizado pelo cliente que o requisitou.

De maneira a termos várias instâncias de um mesmo tipo de atividade, os nomes são acrescidos de um contador de instância. Desta forma, se a atividade AT01 se repete mais de uma vez em um mesmo caso, seus contadores de instâncias são diferentes. O mesmo ocorre para o coordenador de caso e o de processos, que podem ter várias instâncias em um mesmo sistema, assim como para *wrappers*.

5.1.2.1 Hierarquia de Nomes

A hierarquia de nomes é descrita a seguir.

CaseCoordinator: hostname,C,<ProcessName>,<CaseName>-<InstanceCounter>
RoleCoordinator: hostname,R,<RoleName>-<InstanceCounter>
ActivityManager: hostname,M,<ProcessName>,<CaseName>,<ActivityName>-<InstanceCounter>
SynchroniationActivity: hostname,S,<ProcessName>,<CaseName>,<ActivityName>-<InstanceCounter>
GatewayActivity: hostname,G,<ProcessName>,<CaseName>,<ActivityName>-<InstanceCounter>
Wrapper: hostname,W,<ProcessName>,<CaseName>,<ActivityName>,<WrapperName>-<InstanceCounter>
TaskList: hostname,T,<UserName>
BackupServer: hostname,B,<Name>
HistoryServer: hostname,H,<Name>

5.1.3 Ambiente de Suporte a Workflow

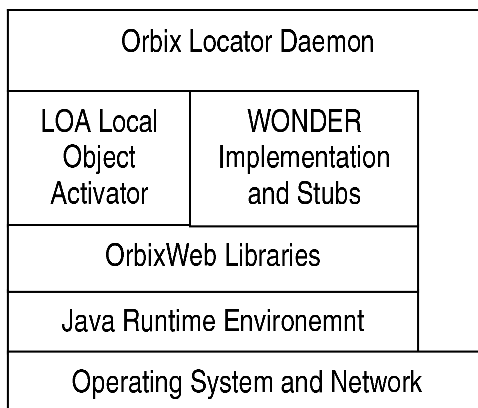

Figura 17: Ambiente de Suporte a Workflow da Arquitetura WONDER

A Figura 17 descreve o ambiente de suporte a workflow que deve estar presente em cada nó do sistema. O orbixd, escrito em C++, é responsável pelo gerenciamento e criação de servidores CORBA no nó local. O orbixd repassa as operações de criação e conexão de servidores para o LOA que se encarrega de criar novos objetos ou reutilizar os já existentes. As bibliotecas do OrbixWeb implementam o barramento de comunicação CORBA usados pelos servidores da arquitetura WONDER. Estes elementos são descritos em mais detalhe nas próximas sessões.

5.2 Hierarquia de Interfaces

A seguir são descritas as principais interfaces resultantes do mapeamento dos componentes da arquitetura para o ambiente CORBA. A hierarquia de interfaces IDL (e algumas classes) é apresentada na Figura 18 a seguir.

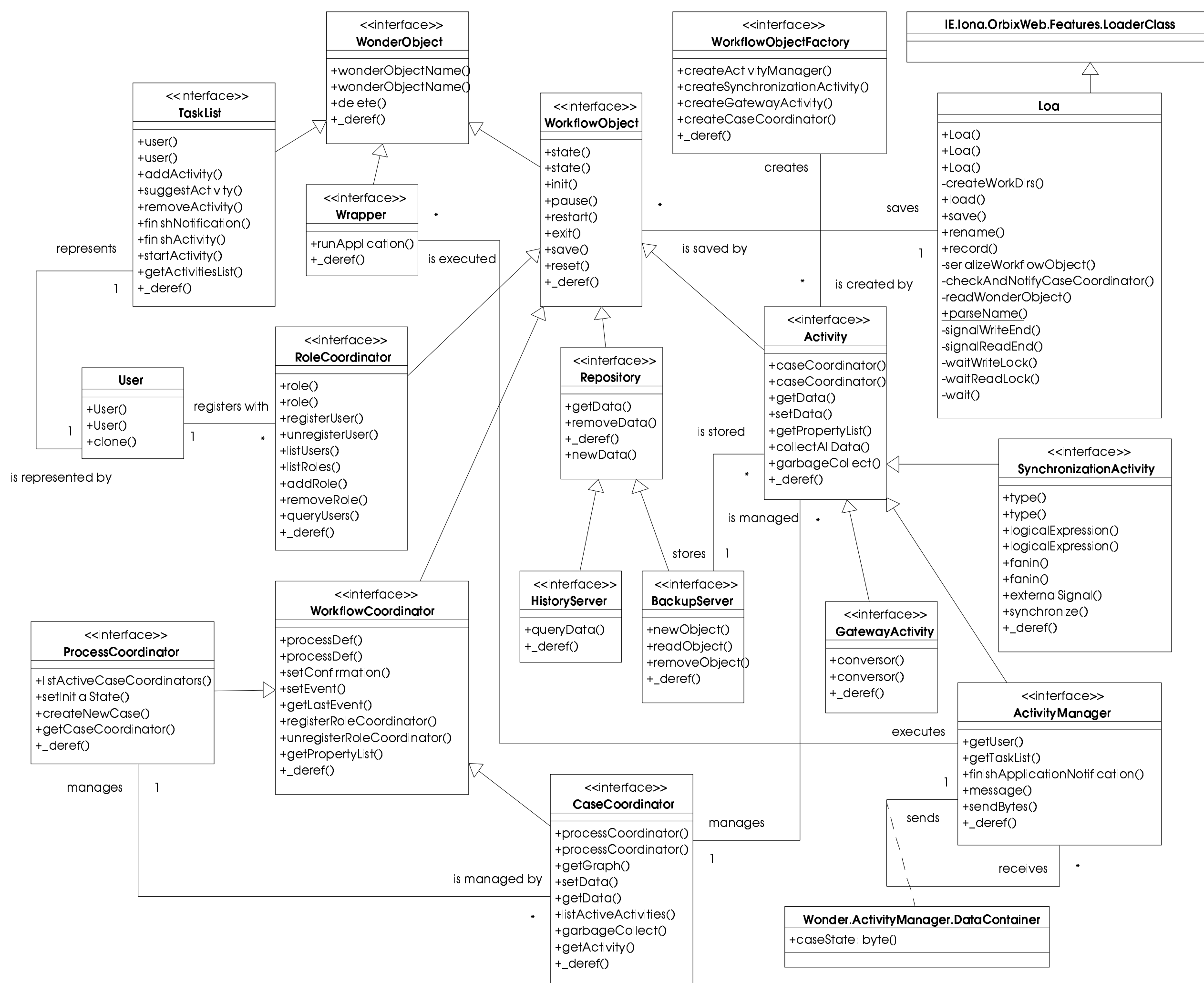

Figura 18: Hierarquia de Classes e Interfaces IDL da arquitetura WONDER

A hierarquia de interfaces (e classes), da Figura 18, é composta de 3 grupos: *Repository*, *WorkflowCoordinator* e *Activity*. *Repository* representam interfaces que devem ser implementadas por objetos que gerenciam e armazenam dados. Existem dois tipos de repositórios: *BackupServer* e *HistoryServer*. *Coordinators* gerenciam a execução de conjuntos de objetos do sistema. São estes os coordenadores de processo e de caso (*ProcessCoordinator* e *CaseCoordinator*). Instâncias de *Activity* são objetos que controlam a execução de tarefas (atividades). Estes servidores são gerenciados por instâncias de *CaseCoordinator*. São definidas duas sub-interfaces de *Activity*: uma para controlar a execução de tarefas por atores (pessoas/programas) – *ActivityManager* – e um para realizar a sincronização de processos (*and/or-joins*) – *SynchronizationActivity*. O *ActivityManager* é também responsável pelo seqüenciamento e execução de atividades, sendo o agente móvel da arquitetura. O procedimento de seqüenciamento utiliza instâncias do tipo *DataContainer* para transportar dados e definições de processos. Estes containers são trocados entre as objetos do tipo *Activity* e *WorkflowCoordinator*. Os grupos encabeçados por *WorkflowCoordinator* e *Activity*, juntamente com *RoleCoordiantor*, são interfaces derivadas de *WorkflowObject*. Objetos do tipo *RoleCoordinator* gerenciam informações dinâ-

micas e de histórico relacionadas aos atores do sistema (Objetos do tipo *User*). Cada ator (ou usuário) possui um ou mais papéis associados. Instancias de *WonderObject* são unicamente identificadas, utilizando o espaço de nomes da arquitetura WONDER, podendo ser controladas, localizadas ou restauradas pelo LOA. Instâncias de *LocalObjectActivator* (LOA) são responsáveis por implementar a persistência e ativação de objetos. Objetos do tipo *TaskList* armazenam informações relacionadas com atividades alocadas para cada usuário.

Uma descrição detalhada das interfaces e seus parâmetros é listada no Apêndice C.

5.2.1 Data Container

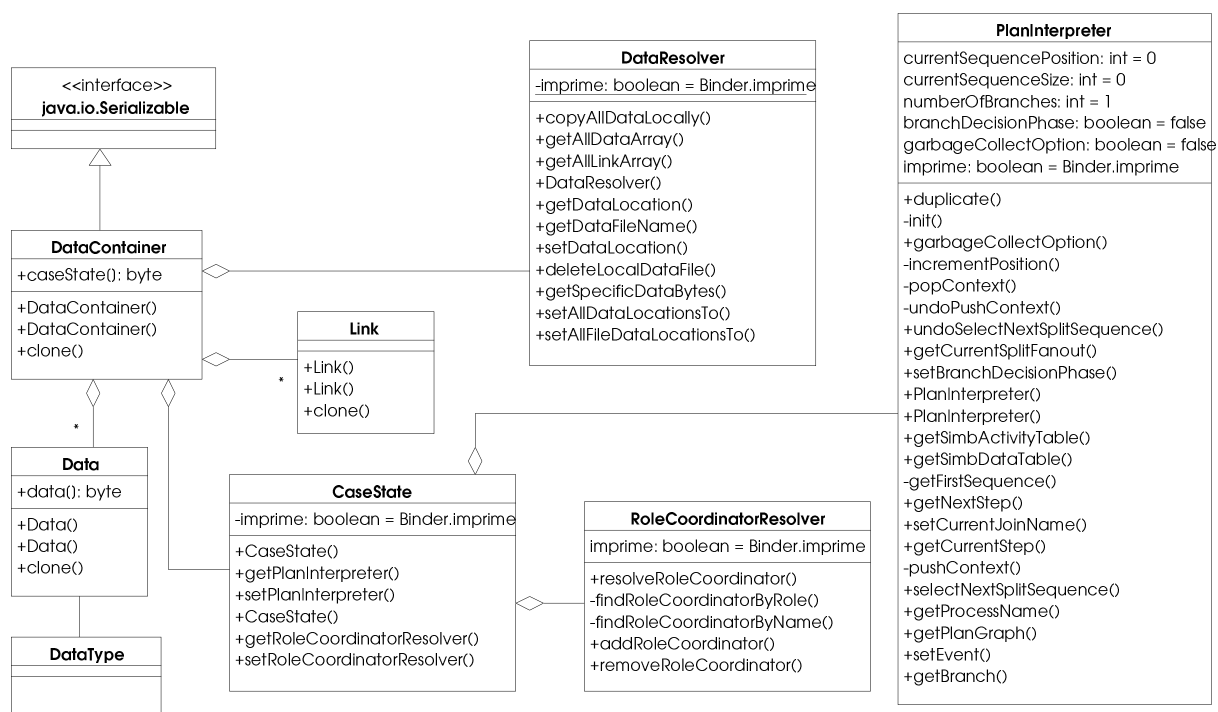

Figura 19: Hierarquia de dados e objetos armazenados no DataContainer

A Figura 19 apresenta o diagrama de classes associadas ao *DataContainer*. Instâncias destas classes são trocadas entre atividades adjacentes e entre o coordenador de caso e a primeira atividade do caso. Objetos do tipo *CaseState*, representam o estado do caso, são compostos pelo interpretador de plano e o resolvedor de coordenadores de papéis (*RoleCoordinatorResolver*). Este último objeto relaciona papéis com seus respectivos servidores. O interpretador de plano (*PlanInterpreter*) possui uma representação do plano em execução (árvore de programa), gerada pelo compilador PLISP, permitindo que esta seja navegada (interpretada) usando, dentre outras, a operações *getNextStep()*, que retorna o símbolo correspondente á próxima atividade, e a operação *getCurrentStep()* que retorna o símbolo relativo à atividade corrente.

O *DataResolver* relaciona dados a suas respectivas localizações (nome do dado e da atividade onde este dado foi modificado pela última vez). Antes de serem transportados entre atividades, dados são armazenados como *byte streams* (byte[]) em objetos do tipo *Data*, que por sua vez, são inseridos em vetores, atributos do *DataContainer*. O objeto do tipo *DataResolver* é mantido sempre atualizado, sendo utilizado durante o seqüenciamento de atividades, onde links (nome dos dados) são resolvidos em dados. Por sua vez, dados desnecessários para a próxima atividade são passados na forma de links (objetos do tipo Link).

Classes do tipo *DataContainer* são trocados entre atividades no formato de *byte streams*. Para tal, precisam ser serializados. Durante este processo, seus objetos agregados (cujas referências fazem parte de seus atributos) são serializados recursiva e automaticamente. Após seu envio para a atividade seguinte, o *DataContainer* é des-serializado, juntamente com todos seus objetos associados. Estes objetos, representando os dados e o estado do caso e, quando recuperados, são "desempacotados" e usados pela próxima atividade.

5.2.2 Compilador PLISP

A representação do plano (ou definição do processo) que será executado e interpretado pelos agentes do sistema (*ActivityManagers*) está intimamente relacionada com o modelo de execução de um workflow. Estas representações devem permitir expressar suas estruturas básicas como *splits*, *joins* e seqüenciamento, assim como atribuições de atores a tarefas, aplicações invocadas, dados a serem utilizados por cada atividade e as demais características do SGWF. São alguns exemplos de linguagens usadas para representar workflow: Redes de Petri e suas variações, regras Evento-Condição-Ação (*ECA rules*), fluxos de controle, linguagens procedimentais e orientadas a objeto, linguagens formais e outros.

Para cada tipo de SGWF é definida, normalmente, uma linguagem diferente. Esta linguagem costuma ser fortemente aderente à arquitetura, recursos, políticas e entidades de cada sistema. Esforços de padronização destas linguagens são desconhecidos para o autor deste trabalho. É importante, ainda, que planos descritos nesta linguagem possam ser facilmente modificados em tempo de execução de forma a suportar *dynamic change* [EKR95] e políticas de recuperação de falhas da arquitetura representada.

Para a arquitetura WONDER optou-se por implementar uma linguagem puramente declarativa cuja sintaxe assemelha-se muito com o paradigma funcional da linguagem LISP [Graham95]. A esta linguagem foi dada o nome de PLISP (*Process LISP*). A gramática, expressa em BNF (*Bacus Normal Form*), da linguagem PLISP é descrita no item A.2 do Apêndice A. Esta linguagem permite expressar as estruturas de controle: *sequence, and-split, or-split*; os tipos de dados: *file e link*, e as atividades da arquitetura (*Activity*). Um plano escrito em PLISP é composto por 3 blocos básicos: o bloco de opções, onde são configurados (ativados/desativados) opções como a de realização de *garbage collection;* o bloco de declarações de elementos onde são especificados: dados, atividades, seqüências e *splits*; e o bloco de declaração do workflow propriamente dito: um *sequence* principal que pode conter outros *sequences* e *splits*, de forma

aninhada. Os elementos declarados no bloco de declarações são usados neste bloco. Semelhante à linguagem LISP, estes elementos podem ser ainda declarados *in loco* nesta última seção. Um exemplo de processo na linguagem PLISP é descrito no item A.3 do Apêndice A.

Esta linguagem é compilada (com realização de análise sintática e semântica) pelo *parser* PLISP. Este *parser* gera uma RIP (Representação Intermediária de Programa), que basicamente é uma árvore de programa, representando a estrutura funcional do plano. Esta RIP é usada durante o tempo de execução para a interpretação do plano. Os nós desta árvore representam *sequences*, *splits*, atividades e dados. Nestes nós são armazenadas informações de desempenho e de execução, usados, dentre outros fins, em procedimentos de recuperação de falhas do WONDER. Na RIP são também armazenados dados de execução, coletados durante o caso, que serão armazenados no servidor de histórico.

O *parser* foi completamente implementado em Java usando o JavaCC (*Java Compiler Compiler*) [JavaCC], gerador de compiladores de domínio público que realiza análise ascendente, permitindo a realização da análises sintática e semântica e a geração da árvore de programa em uma única passada do código fonte. Por ser escrito em Java, o compilador é perfeitamente integrável com a aplicação WONDER. A árvore de programa gerada é passada diretamente para o interpretador de plano (objeto que corresponde ao *workflow engine* da arquitetura) e é passado como estado do agente móvel.

O uso de uma linguagem cuja sintaxe é semelhante à de LISP, facilitou bastante a construção desta árvore de programa. Por ser uma linguagem funcional, permitindo declaração de funções *in loco*, dentro de escopos aninhados, a linguagem PLISP possui uma gramática sem ambigüidades.

5.2.3 Suporte a conversão de Links em Dados

A Linguagem PLISP foi especificada de maneira a prover estruturas que fornecem informações relacionadas aos dados que são modificados, criados ou somente lidos por cada aplicação invocada. A declaração de *wrappers*, em PLISP, fornece esta informação. Cada atividade pode ser configurada para disparar mais de uma aplicação. Desta forma, é possível saber, de antemão quais dados são necessários à execução de cada atividade, quais dados foram modificados e sua última localização. Esta informação é fornecida para o *DataResolver*, sendo utilizada durante o seqüenciamento das atividades.

5.2.4 Compilador WStarter

O compilador WStarter foi criado para montar a configuração de servidores (de processos, de casos, papéis e *task lists*) inicial do sistema, criando um ambiente para a realização dos testes

de desempenho da arquitetura. Seguindo o modelo de sintaxe adotado pelo compilador PLISP, o WStarter também utiliza uma linguagem derivada de LISP para especificar este ambiente.

A definição de um ambiente de execução (*environment*), descrito nesta linguagem WStarter, contém três blocos fundamentais: declarações (*declarations*), configurações (*configure*) e execução (*run*). Nas declarações são especificados coordenadores de processos, com suas respectivas configurações e o seu plano associado. Nesta seção são configurados também os servidores de papéis e os usuários. Estes últimos serão representados por servidores CORBA do tipo *TaskList*. Em todas as configurações, o nó onde cada servidor será criados pode ser especificado. Caso a informação seja omitida, através do uso de *null* ou vazio, o nó corrente é selecionado. No bloco de configuração, os usuários são associados a seus respectivos servidores de papéis. Um usuário pode estar associado a mais de um papel. No bloco de execução, são descritos, em ordem de criação, os casos a serem criados, com seus respectivos coordenadores de processo. Um tempo de espera (em ms) pode ser especificado de maneira a retardar a criação do caso. Este último recurso foi criado especialmente para controlar a taxa de criação de casos nos testes. Um exemplo de um ambiente é descrito no item A.5 do Apêndice A.

5.3 Máquinas de Estado

A seguir, são descritas as máquinas de estados (DTE – Diagrama de Transição de Estados) dos principais elementos implementados no protótipo da arquitetura WONDER. Os diagramas representam o comportamento básico destes objetos, da maneira em que foram implementados no protótipo utilizado na realização dos testes deste trabalho. As figuras usam a notação UML para *Statecharts* [BJR97].

Todos os objetos da arquitetura WONDER, derivados de *WorkflowObject* são passíveis de serialização pelo LOA. Esta serialização ocorre de maneira automática quando o orbixd detecta um período de inatividade (*timeout*) maior ou igual ao configurado, tipicamente 5 minutos para atividades e 10 minutos para coordenadores, ou quando a operação *save()* é invocada explicitamente na interface do objeto.

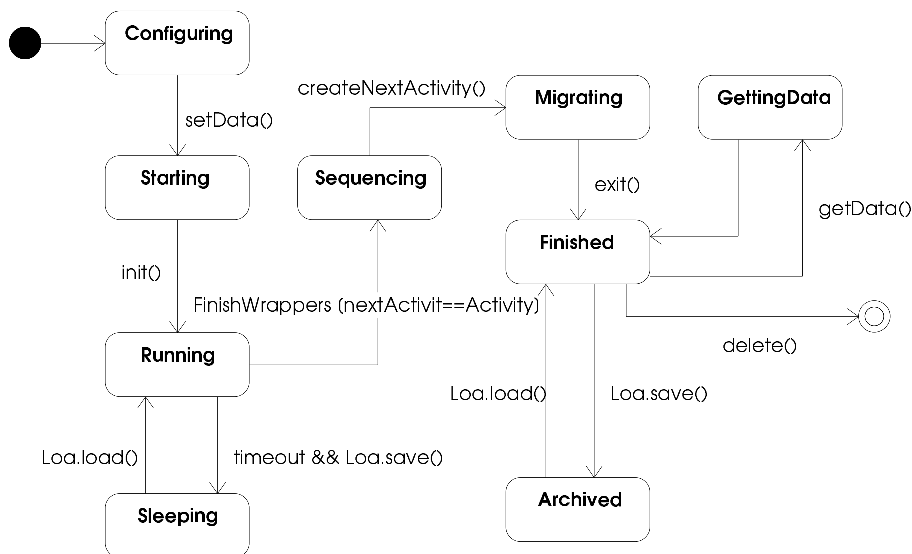

Figura 20: DTE da classe Activity

Ao ser criada, a atividade (cujo DTE é apresentado na Figura 20) é configurada, recebendo o estado de execução e os dados necessários a sua execução através da operação *setData()* de sua interface. Após a inicialização, a atividade é colocada em operação com o comando *init()*. Os servidores *wrappers*, independentes da atividade, são então criados. Cada *wrapper*, ao terminar sua execução, notifica seu *ActivityManager* associado. Ao final de todas as aplicações invocadas, o processo de *seqüenciamento* é iniciado (*Sequencing*).

Caso ocorra um erro durante a invocação de *setData(),* o ORB gera uma exceção. O comando *setData()* pode então ser novamente chamado.

O seqüenciamento consiste em determinar o nó da próxima atividade, criando um servidor CORBA, do tipo *Activity* para desempenha-la. Este processo envolve a "descoberta" do ator para a próxima atividade, feito junto ao *RoleCoordinator* juntamente com o *TaskList* do(s) ator(es) (candidatos) selecionados.

Após a criação da próxima atividade, o processo de migração inicia-se. Neste estado, dados para a próxima atividade são coletados e, junto com o interpretador de plano e os resolvedores de papel e de arquivos, são "embrulhados" no *DataContainer* (inseridos como atributos), serializados e enviados como *byte stream* à próxima atividade.

O processo de coleta de dados consiste em despertar atividades anteriores, contendo as últimas versões dos arquivos e dados, e requisita-los com o comando *getData()*. Esta operação leva as atividades para o estado *GettingData*. O processo é implementado, no lado cliente, pelo objeto do tipo *DataResolver* que armazena referências (links) para a última localização destes dados.

Se o tempo de execução da atividade ultrapassa um determinado *timeout*, tipicamente 5 minutos, na implementação do protótipo, este servidor é parado, serializado e removido da memória.

Esta política permite liberar a memória da estação de trabalho na eventualidade de atividades muito demoradas.

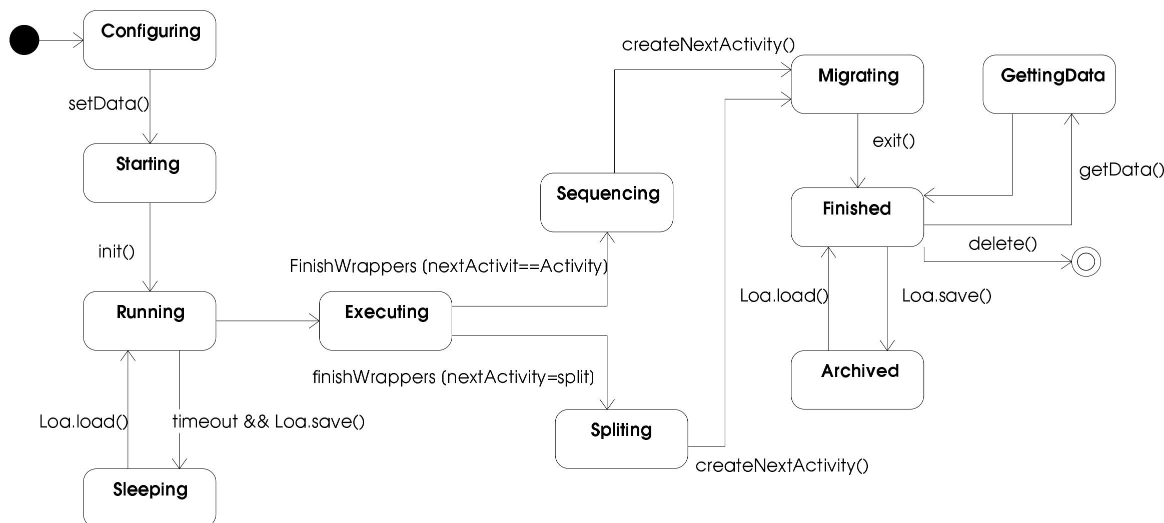

Figura 21: DTE da calasse ActivityManager

A Figura 21 mostra o DTE da classe *ActivityManager*. Esta classe é uma especialização de *Activity*, comportando-se de maneira semelhante a sua superclasse, quando na fase de inicialização e configuração. O seqüenciamento da próxima atividade pode, por sua vez, ser realizado de duas maneiras: o seqüenciamento simples, semelhante ao realizado em *Activity*, ou o seqüenciamento múltiplo, feito durante *splits*.

Durante *and-splits* (estado *Splitting* da Figura 21), as atividades consecutivas são seqüencializadas, uma a uma, em ordem de declaração; durante *or-splits*, uma expressão lógica é avaliada e, de acordo com seu resultado, uma ou mais atividades são seqüencializas.

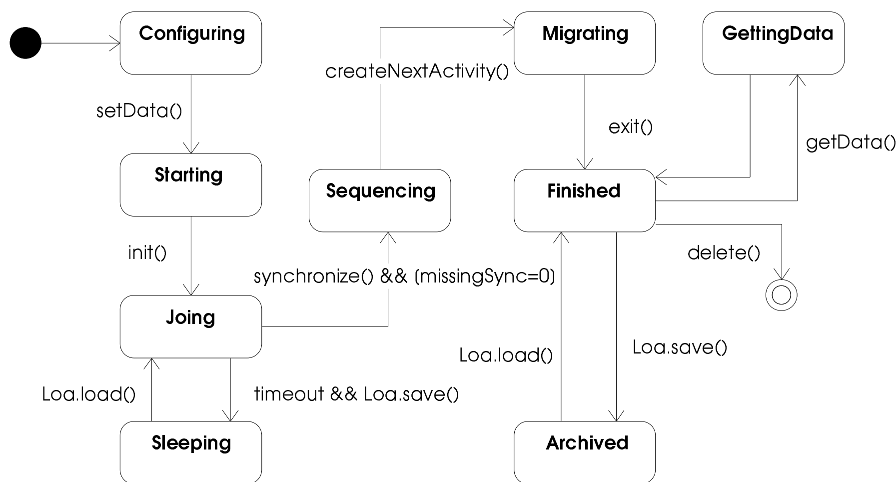

Figura 22: DTE da classe SynchronizationActivity

A atividade de sincronização (*SynchronizationActivity*), descrita na Figura 22, comporta-se como um concentrador de notificações. Estas atividades são criadas pelos coordenadores de caso (*CaseCoordinators*), sendo configurados com o valor de seu tipo (*and/or-join*) e a expressão lógica a ser avaliada. Atividades de sincronização recebem notificações (*synchronize()*) e dados (*setData()*) das atividades que a antecedem. Estes dados podem ser processados, mesclados e filtrados de maneira a compor os dados da(s) atividade(s) que seguintes. Ao receber todas as notificações esperadas, de acordo com a expressão avaliada, a atividade de sincronização inicializa o processo de seqüenciamento da próxima atividade (se houver).

Quando a atividade de sincronização implementa um *and-join*, todas as notificações precisam ser recebidas antes que o seqüenciamento se inicie; para *or-joins*, o número de notificações recebidas, de maneira a iniciar o seqüenciamento da próxima atividade, depende da expressão a ser avaliada. Notificações recebidas durante os estados *Migrating* ou posteriores são ignoradas.

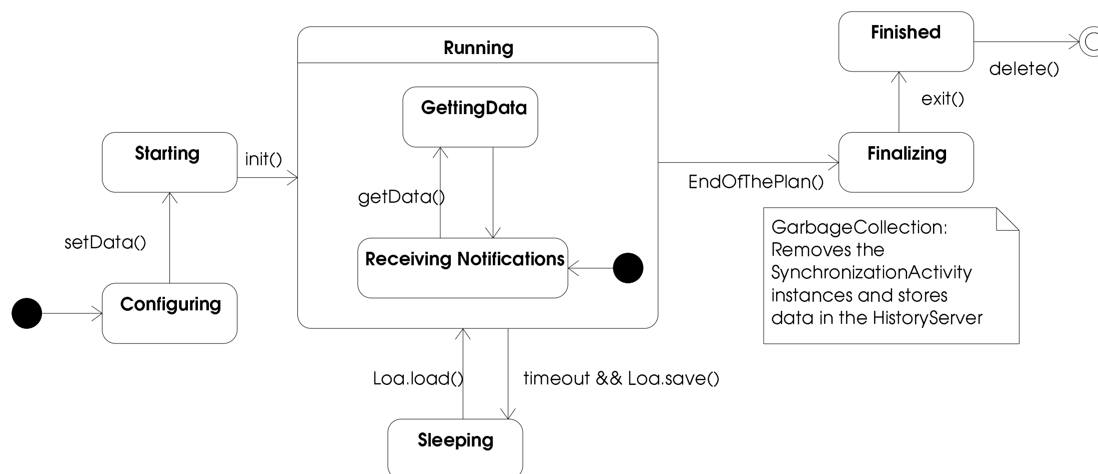

Figura 23: DTE da classe CaseCoordinator

Ao ser criado, o coordenador de caso (*CaseCoordinator*), cuja máquina de estados é descrita na Figura 23, entra em estado de configuração (*Configuring*) e recebe, de uma única vez, todo o conjunto de dados iniciais do caso. Após sua configuração, ser invocada a operação *init()*, este servidor inicializa o sua atividade. Neste momento, é criada a configuração inicial do caso: são instanciadas as atividades de sincronização, adicionando suas localizações no plano; em seguida, é criada a primeira atividade, recebendo o plano e os dados necessários a sua execução. A primeira atividade do caso é, então inicializada e segue de forma autônoma, interpretando o plano recebido. Durante todo o caso, seu coordenador recebe notificações cada vez que há uma transição de estado dos objetos do tipo *Activity*. Durante a resolução de dados das primeiras atividades, dados são geralmente coletados diretamente do coordenador de caso, usando o comando *getData()* da interface deste servidor. Neste último aspecto, o coordenador se comporta de maneira semelhante a qualquer outra atividade que contém as versões mais atuais dos dados do caso.

Ao receber a notificação de final de caso, o coordenador de caso inicia o procedimento de *garbage collection*. Este procedimento remove as atividades de sincronização, coleta e apaga os dados de *checkpointing* deixados pelas atividades nos nós da rede e nos servidores de backup.

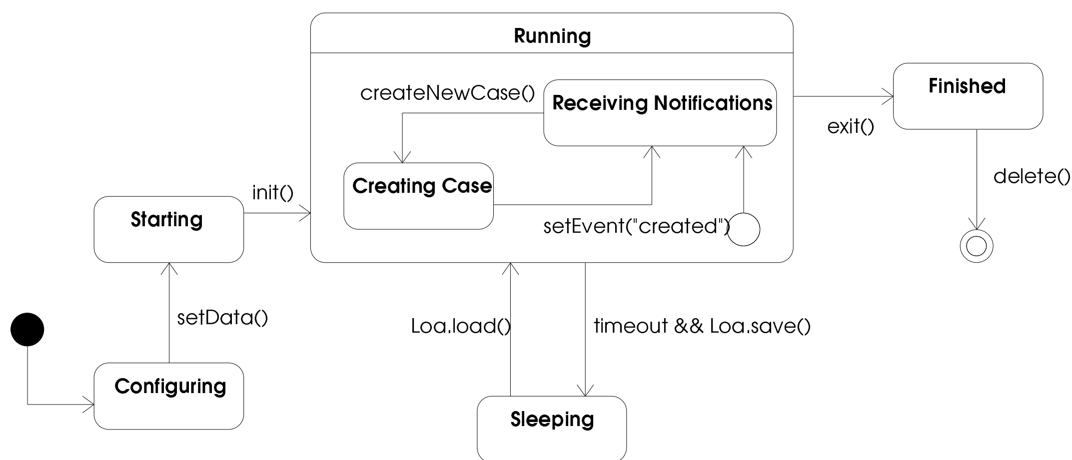

Figura 24: DTE da classe ProcessCoordinator

O coordenador de processo (Figura 24), de forma análoga ao coordenador de caso, armazena todos os dados iniciais do processo e os recebe, no momento de sua criação, através do comando *setData()*. Estes objetos coordenadores recebem notificações dos coordenadores de casos por ele criados, através da operação *setEvent()* de sua interface. Novos casos são criados através da invocação do comando *createNewCase()*.

Capítulo 6

Paralelo entre o Paradigma de Agentes Móveis e a Arquitetura WONDER

Embora tenha sido desenvolvida em CORBA e não com o auxílio de um sistema de agentes móveis (SAMs), a arquitetura WONDER implementa muitas das características providas por estes sistemas, sofrendo de problemas e possuindo requisitos semelhantes.

O objetivo principal do projeto WONDER foi o desenvolvimento de uma arquitetura em CORBA, que satisfizesse os requisitos de SGWFs de larga escala. O uso do conceito de casos móveis foi, desta maneira, a solução escolhida para abordar os requisitos de tolerância a falhas, disponibilidade e escalabilidade. A arquitetura não foi implementada utilizando sistema de agentes móveis, a exemplo do Voyager [ObjectSpace97], Aglets [KLO97] ou outros, pois sua integração com CORBA é parcial.

Desta forma, o WONDER implementa conceitos de um SAM sem contudo pretender prover todas as facilidades de um sistema como este.

É apresentado, a seguir, um paralelo entre o paradigma de agentes móveis e as características da arquitetura WONDER, descrevendo suas correspondências, tomando como base aos principais requisitos dos sistemas de suporte a agentes móveis descritos no Capítulo 3.

6.1 Transportabilidade

O processo de migração de agentes móveis na arquitetura WONDER é bastante simplificado. Ao invés de transportar todo o objeto de um nó a outro, são movidos apenas, o plano (contido no interpretador de plano), o estado atual do interpretador de planos, o resolvedor de dados e os

dados da atividade seguinte. Estes dados são serializados e transportados através de uma única invocação de operação.

A arquitetura permite que o código dos servidores do SGWF esteja previamente instalado nos nós do ambiente de execução ou que estes sejam compartilhados via NFS. Neste último caso, o código binário dos agentes é compartilhado via sistema de arquivos distribuído e, portanto, movido sob demanda toda vez que um objeto de um objeto de uma determinada classe (arquivo .class) é instanciado. Como esta mobilidade não é implementada explicitamente, mas sim provida indiretamente pelo NFS (ou similar), pode-se afirmar que o WONDER utiliza mobilidade fraca de agentes.

O uso de NFS simplifica a administração configuração dos nós do sistema. Em contra partida, qualquer falha de comunicação ou do próprio nó onde os binários estão instalados pode paralisar a execução do sistema. Para evitar este possível ponto único de falhas, diminuindo ainda o *overhead* de comunicação associado ao transporte de código binário via rede, a instalação dos binários em cada máquina do sistema é preferível.

6.2 Autonomia

A autonomia do "caso móvel" é provida pelo interpretador de plano carregado por cada atividade. Este objeto, que é trocado entre as atividades como parte do estado do caso, fornece informações necessárias à execução do caso corrente. O interpretador de plano também fornece o itinerário a ser seguido pelo agente. Informações contidas no plano são atualizadas durante a execução do caso.

6.3 Navegabilidade

O plano não contém todas as informações necessárias à execução das atividades. Parte destes dados são determinados dinamicamente, durante a execução do caso, através do auxílio de servidores externos ao agente (*ActivityManager*). Estes servidores auxiliares são: o servidor de papéis, que fornece ao agente uma lista de atores candidatos para a realização da próxima atividade, de acordo com o papel especificado no plano; os *task lists* dos usuários, que intermediam o processo de aceitação/rejeição da próxima atividade; e o servidor de histórico que é consultado pelo servidor de papéis na determinação de atores com base em históricos dos casos.

6.4 Segurança

Os aspectos de segurança da arquitetura WONDER envolvem basicamente dois fatores: a transferência de dados (potencialmente) confidenciais para nós pouco confiáveis; e a execução de atividades em máquinas não autorizadas.

De maneira a prover segurança aos dados e aos agentes do caso, estes são serializados e armazenados em uma área protegida do sistema de arquivos de cada nó (Repositório de Objetos). Este repositório é acessível somente a objetos do tipo *Activity* e *WorkflowCoordinator*. O protótipo WONDER implementado assume que esta proteção é provida pelo sistema operacional, no caso Windows NT ou Solaris, devidamente configurados para controlar o acesso dos usuários aos binários e dados da arquitetura.

Na arquitetura WONDER não há transferência do código binário do agente entre os nós do ambiente distribuído. Esta política previne o sistema de ataques de vírus (agentes móveis mal intencionados) que possam tentar se passar por atividades.

Para que o agente possa migrar entre os nós, em cada máquina do sistema deve estar instalado e configurado o ambiente de execução WONDER, que inclui o OrbixWeb e os binários dos agentes. Este ambiente corresponde a uma *agência* de agentes móveis que hospeda as atividades. O uso de NFS, como forma de prover e compartilhar o código binário dos agentes, também é possível mas, para maior eficiência e segurança, recomenda-se a instalação prévia dos binários nos nós da rede.

Há sempre a possibilidade, entretanto, que pessoas mal intencionadas substituam, invadam ou roubem estações de trabalho contendo dados críticos dos casos. Além do mais, ataques a estações de trabalho são bem mais fáceis de serem realizados que ataques a servidores centrais, que via de regra são mais bem protegidos. Vírus de computadores, ou mesmo descuidos de segurança dos usuários podem facilmente danificar estes valiosos dados. Uma alternativa contra acesso indevido a informações, seria criptografar estes dados com chaves conhecidas apenas por servidores de autenticação especialmente protegidos (e potencialmente centralizados). Estas alternativas, contudo, não foram exploradas pois fogem ao escopo primário deste trabalho.

6.5 Tolerância a falhas

O uso do paradigma de agentes (casos) móveis introduz diversos pontos de falhas no sistema. Se no paradigma cliente-servidor centralizado, os servidores são máquinas mais confiáveis, possuindo recursos como *no-breaks*, memória com correção de dados, discos duplicados, backups periódicos, servidores redundantes, links de rede alternativos e outros recursos de redundância e segurança; na arquitetura completamente distribuída usada no WONDER, os nós envolvidos são meras estações de trabalho.

Várias políticas de tolerância a falhas foram implementadas no sistema. As políticas adotadas na arquitetura WONDER são específicas para o problema de workflow de larga escala. Em sua maioria, são implementadas por servidores auxiliares como coordenadores de casos e servidores de backup. O agente está programado apenas para escolher nós que estejam disponíveis no momento do seqüenciamento. Falhas envolvendo o nó onde um agente estava sendo executado são tratadas pelos servidores auxiliares como o coordenador de caso.

De maneira semelhante às utilizadas em SAMs, algumas medidas/políticas de redundância são adotadas como forma de dar suporte a ações que permitam a recuperação de falhas do sistema. Cópias do estado do caso são mantidas pelo ambiente de suporte a workflow da arquitetura nos nós onde o caso executou (*checkpointing*). Estes dados funcionam de maneira semelhante a pontos de sincronização, permitindo aos coordenadores de casos restaurar o estado do caso a partir destes dados;

Também são especificados servidores de backup especiais que, em períodos de pouco uso do sistema, coletam estas informações deixadas nos nós e as armazenam em meios mais confiáveis. O coordenador de cada caso mantém o estado corrente da execução de sua instância de processo, detectando erros e tomando as devidas medidas de compensação.

6.6 Desempenho

O estado do caso trocado entre as atividades da arquitetura WONDER é bastante compacto, ocupando (quando serializado e comprimido) o volume de, no máximo, poucos quilobytes. O processo de criação de um servidor CORBA costuma demorar no máximo alguns segundos. Estes valores serão discutidos no Capítulo 8.

Tanto no paradigma cliente-servidor, quando no paradigma de agentes móveis, este último empregado na arquitetura WONDER, os dados utilizados por uma atividade (dados do caso) precisam ser copiados para o nó do ator que a desempenha, seja através do uso do NFS, ou transportados pelo agente. A diferença é que no paradigma de agentes móveis, é necessário mover o agente para o nó onde a atividade será realizada.

De forma a evitar que dados que não serão utilizados em uma determinada atividade sejam transferidos desnecessariamente para um nó, uma política de uso de *links* foi utilizada. Referências aos dados do processo são carregadas junto com o estado do caso. Dados são copiados para o nó local, a partir dos nós onde estes foram deixados/atualizados pela última vez, somente quando são necessários. Desta forma, os dados do caso permanecem no último local onde foram alterados até serem necessários por uma atividade posterior.

O uso de Java e OrbixWeb contudo, demandam uma grande quantidade de memória, o que o torna incompatível com sistemas móveis portáteis.

Parte do problema relacionado ao excesso de agentes em um mesmo nó é resolvido com a implementação de um mecanismo de persistência e *timeout* de servidores. Objetos que não são usados por um determinado período de tempo são removidos da memória principal, permanecendo inativos em disco.

6.7 Suporte multiplataforma

O suporte a multiplataforma de hardware e software é alcançado através do uso da linguagem Java que permite ao sistema ser executado em todas as plataformas suportadas pelo OrbixWeb. Atualmente o OrbixWeb está disponível para as plataformas Windows NT, Solaris e HP-UX. O JDK (*Java Development Kit*) da SUN, utilizado pelo OrbixWeb, está disponível para mais plataformas, como Linux, por exemplo.

6.8 Adaptabilidade

A adaptabilidade, na WONDER, está relacionada à alocação dinâmica de atores e atividades. Servidores de papéis e *TaskLists* auxiliam na escolha de atores que satisfaçam políticas como “ator menos carregado”, “ator mais experiente” e outras políticas relacionadas a alocação (*binding*) dinâmica de papéis.

A uma determinada atividade (servidor ActivityManager) é criada somente no momento em que esta será desempenhada. Esta política adia a criação de servidores, permitindo que a alocação de atividades siga a atual disponibilidade de recursos do sistema. Máquinas indisponíveis num determinado momento e atores muito ocupados ou ausentes não são geralmente evitados.

6.9 Comunicação

A comunicação entre agentes (*ActivityManager*, *SynchronizationActivity* e *GatewayActivity*) ocorre basicamente em dois momentos: no momento do seqüenciamento de atividades, quando a atividade seguinte é criada e o estado do caso é entre eles transferido; e no momento que antecede o seqüenciamento, onde a atividade corrente coleta (a partir de outras atividades) os dados que a atividade seguinte irá utilizar.

A maior parte da comunicação de uma atividade ocorre no sentido atividade – coordenador de caso. Esta comunicação, realizada através de envio de eventos (pequenas mensagens) assíncro-

nos, permite o monitoramento do caso corrente. Em todo o processo de comunicação são utilizados nomes dependentes de localização.

6.10 Serviço de Nomes

O OrbixWeb implementa um serviço de nomes distribuído integrado com um seu gerente de ativação (orbixd). O orbixd mantém uma tabela de nomes e referências IOR, estes nomes são chamados de *markers*. No modo de ativação *unshared* do OrbixWeb, cada vez que um servidor é criado, um *marker* pode ser atribuído a este objeto. Referências a este objeto podem agora ser obtidas, apresentando o seu *marker* ao *orbixd*, tipicamente durante a chamada da operação *bind*(). Vários clientes podem usar um mesmo objeto simultaneamente. O *orbixd* se encarrega de ativar objetos que não estejam em execução na memória (cujo *marker* não esteja em sua tabela de nomes), criando uma nova instância destes servidores, ou lendo objetos serializados através do uso de *Loaders*.

Este mecanismo de ativação é usado para criar novos objetos na arquitetura WONDER, assim como para implementar a persistência de objetos locais em um nó. A criação de um espaço de nomes para identificar unicamente um objeto (em especial os do tipo *ActivityManager*) da arquitetura WONDER, valeu-se do uso destas facilidades do OrbixWeb e permitiu a não utilização de um servidor de nomes central padrão OMA.

6.11 Persistência de Objetos

O ambiente de suporte aos casos (agentes) móveis e a persistência de objetos é implementado pelo LOA (*Local Object Activator*), juntamente com o orbixd. O LOA é integrado com o orbixd através da implementação da interface *LoaderClass*. O LOA é então registrado junto ao orbixd, na forma de um *hook* (ponto adaptável de um *framework* cuja implementação é opcional), durante a criação de cada servidor CORBA. Todas as chamadas de criação ou conexão (*bind()*) de objetos passam pelo LOA que é responsável por criação, persistência e gerenciamento de objetos de um determinado nó. Os objetos são identificados utilizando o espaço de nomes de objetos da arquitetura WONDER.

6.12 Espaço de nomes de objetos WONDER

Um dos requisitos básicos de sistemas de agentes móveis é a comunicação transparente entre estes agentes. Para que esta comunicação possa ocorrer de maneira independente da localização

dos objetos, os SAMs podem utilizar políticas como *forwarding addresses* [Fowler85] e outras atualizações do serviço de nomes, feitas pelos agentes que migraram, ou pelo próprio SAM.

A cada nova atividade desempenhada pelo caso móvel da arquitetura, um novo servidor ActivityManager é criado. A estes novos servidores, representando as tarefas do caso, são atribuídos novos nomes. Desta forma, apesar do estado do agente ser mantido durante sua migração, seu nome muda conforme novas atividades são realizadas. Um servidor ActivityManager representa somente uma única atividade do workflow. Uma vez criado em um nó, o estado persistente do objeto (servidor CORBA) permanece armazenado no repositório de objetos desta máquina até o momento de sua destruição, ao final do caso.

Como o agente móvel possui vários nomes durante sua vida, não há necessidade de atualização de um servidor de nomes central. O plano da atividade corrente e do coordenador de caso, em contra partida, é atualizado passando a incluir o nome desta nova atividade criada.

O único caso onde é necessária a atualização de uma referência de objetos ocorre durante a recuperação de falhas de um objeto, ou seus dados, armazenados em um nó que falhou. Se o objeto foi devidamente copiado para o servidor de backup, este passa ser agora a localização oficial deste objeto. Desta forma, todas as referências do plano corrente precisam ser atualizadas para referenciar esta nova localização.

O espaço de nomes de objetos WONDER permite localizar um objeto diretamente, sem consultas a serviços de nomes globais. Uma referência de objetos é retornada, através da chamada *bind(nome da máquina, marker:serverType)* feita diretamente no *orbixd* do nó onde este objeto se encontra.

6.13 Conclusões da Seção de Agentes Móveis e WONDER

O uso de objetos móveis permite que haja a movimentação de dados e processamento para o nó do ator que realizará uma tarefa.

Os binários dos objetos e do ambiente de suporte da arquitetura WONDER podem ser disponibilizados de duas maneiras. Na primeira, estes arquivos são compartilhados via NFS (ou equivalente), na segunda, o NFS está ausente e cópias destes arquivos precisam ser instalados em cada máquina do sistema. No primeiro caso, há mobilidade de código, feito sob demanda, de forma transparente, através do uso do NFS. No segundo caso, como não há mobilidade de código, o ambiente de execução de workflow, que inclui o *daemon* do OrbixWeb e os binários que implementam a arquitetura WONDER, deve ser replicado em cada nó que participa do SGWF. Esta última abordagem permite tornar os nós independentes da conexão de rede, durante todo o tempo de execução de uma atividade. Este ambiente de execução deve ter acesso restrito apenas aos usuários do sistema em um ambiente corporativo.

Comparado com sistemas centralizados, que utilizam o paradigma cliente-servidor, o uso de Atividades (agentes) distribuídas, que movem-se carregando seus próprios dados, não reduz o tráfego geral destes dados pela rede. Em ambos os casos (distribuído e centralizado), dados ou parte dos dados, precisam ser copiados localmente (sob demanda ou não), na máquina cliente, para serem usados pelas aplicações. O modelo descentralizado utilizado no paradigma de agentes móveis, contudo, distribui o tráfego de dados de maneira mais uniforme pelos enlaces da rede, evitando a sobrecarga dos enlaces de conexão com o servidor central. O tráfego não é mais cliente-servidor central mas ponto-a-ponto. A descentralização de dados e de controle permite desta forma distribuir a carga de processamento e de comunicação, de maneira mais uniforme, pelos nós do sistema.

É importante também ressaltar que, quando é usado o NFS para compartilhar os binários da arquitetura WONDER, estes dados também aumentam o tráfego na rede. Em ambientes onde a latência da rede é expressiva, o uso de cópias dos binários da arquitetura pelos nós da rede é preferível.

Capítulo 7

Implementação

Descrevemos, neste capítulo, os principais aspectos e problemas relacionados com a implementação do protótipo da arquitetura WONDER criado para a realização de testes de desempenho.

O sistema foi desenvolvido no Instituto de Computação da UNCIAMP. Foi escrito inteiramente em Java (JDK1.1 da SUN) utilizando a implementação de CORBA da Iona, OrbixWeb3.1c. O sistema distribuído utilizado para a implementação e os testes é composto de estações de trabalho NCD X-Terminals, estações de trabalho Windows NT/LINUX e máquinas Sparc rodando o sistema operacional Solaris da SUN. Estes computadores estão conectados por uma rede local de 10Mb.

O WONDER possui 21970 linhas de código Java (LOC – já descontados os comentários e as linhas em branco). Deste total, 7325 LOCs foram implementados e 14645 LOCs foram geradas automaticamente.

Das 21970 LOC do projeto, 4805 LOC são usadas na implementação dos objetos da arquitetura; 8283 LOCs correspondem aos *stubs* e *skeletons* gerados automaticamente pelo compilador IDL do OrbixWeb e 5918 LOC são utilizadas na implementação do compilador PLISP e 2964 LOC implementam o compilador WStarter. Do total de LOCs do compilador PLISP, 4001 são geradas automaticamente pelo JavaCC. Da mesma forma, do total de LOCs do compilador WStarter, 2361 são geradas pelo JavaCC.

7.1 Simplificações do Modelo Implementado

O presente trabalho envolveu a implementação de um protótipo da arquitetura WONDER. Devido a limitações de tempo do mestrado, o principal objetivo do protótipo foi implementar um subconjunto da arquitetura WONDER, fornecendo um ambiente de testes que provesse as funcionalidades básicas da arquitetura, de maneira a verificar seu comportamento perante os requisitos de workflow de larga escala, em especial o de escalabilidade. Não foram implementadas, desta maneira, políticas de tolerância a falhas, de segurança, de escolha de usuários (avançadas) e de operação desconectada. O núcleo básico da arquitetura, seu agente móvel, foi implementado com a funcionalidade básica de migração e a política de transporte de dados e links. Os outros componentes da arquitetura foram implementados provendo seu comportamento básico, de maneira a prover um pseudo comportamento de um ambiente real.

De maneira a proverem maior escalabilidade, os coordenadores deveriam ter sido implementados como objetos *multithreading*. Por simplicidade, estes objetos foram implementados como objetos simples. Requisições e eventos recebidos são enfileirados pela própria pilha TCP/IP usada pelos skeletons CORBA.

A seguir são descritos os principais componentes da arquitetura e o que foi, ou o que não foi, implementado em cada objeto. Todos os servidores a seguir foram registrados, no repositório de implementação do OrbixWeb no modo de ativação *unshared*.

ActivityManager*:* Foi implementado o sistema básico de migração, interpretação de plano, coleta de dados de execução e ativação de *wrappers*.

Durante o seqüenciamento, o *ActivityManager* implementado para o protótipo consulta o coordenador de papel, que fornece a lista de todos os usuários registrados para este papel. O *ActivityManager* então seleciona aleatoriamente um ator para realizar a próxima atividade. A próxima atividade é então sugerida ao *TaskList* deste ator, que foi programado para sempre aceitar qualquer atividade. Esta atividade é criada, configurada e registrada no *TaskList*. O *TaskList* então simula a seleção da atividade pelo ator (o ator escolhe realizar a atividade neste momento). Com esta seleção, o *TaskList* inicia a atividade (chamando o comando *init()* do *ActivityManager* em questão).

Durante o período de execução, os *wrappers* são disparados de maneira concorrente. Se o tempo de execução dos *wrappers* for muito longo, o mecanismo de *timeout* é ativado e a atividade que os disparou é salva e removida da memória. Ao final de sua execução, cada wrapper desperta o *ActivityManager* e o notifica. Ao colecionar todas as notificações, o *ActivityManager* inicia o de seqüenciamento da próxima atividade.

Wrapper. Executa aplicações especificadas no plano, esperando por seu término. Notifica o *ActivityManager* ao final da execução da aplicação. O *wrapper* foi testado com *shell scripts* usados nos testes da arquitetura. Não foram implementados a coleta nem o envio de dados para as aplicações invocadas.

Wrappers possuem *timeout* = –1 (infinito), desta forma, nunca são removidos da memória, não sendo portanto objetos persistentes. Para cada aplicação invocada existe um *wrapper* associado.

LOA. Foi totalmente implementado, usando o mecanismo de persistência de objetos, utilizando os recursos de *object serialization* da linguagem Java, assim como a persistência de referências através do uso do espaço de nomes da arquitetura WONDER.

Os estados de execução dos objetos são armazenados pelo LOA no subdiretório /WonderData diretório /tmp local de cada máquina. Este diretório assume, desta forma, o papel de ObjectRepository.

PlanInterpreter. Foi implementado de forma integrada com o compilador PLISP. Armazena e informa dados sobre as atividades corrente e futura. Embora a sintaxe permita especificar consultas SQL e passagem de links, estes recursos são ignorados pelo *ActivityManager* e coordenadores de caso e processo. Apenas arquivos, existentes desde o início do caso, são trocados entre as atividades.

ObjectRepository. Implementado na forma de um diretório especial (criado sob /tmp, para a realização dos testes), utilizando os recursos de segurança do sistema de arquivos do sistema operacional (Solaris ou NT).

SynchronizationActivity. Realiza *OR-Joins* e *AND-Joins* simplificados. Para sincronizações do tipo *OR-Join* não foi implementado a avaliação de expressões lógicas. Neste tipo de sincronização, a próxima atividade é criada ao primeiro comando *synchronize()*, os outros são ignorados. Ao invés de serem criadas no início do caso, como previsto na especificação, estas atividades são criadas, por *ActivityManagers*, durante a execução de *splits*. Neste momento, o servidor de sincronização é criado e seu nome é acrescentado ao plano. Este plano é passado para as atividades seguintes ao *split*. O recebimento de eventos externos também não foi implementado.

CaseCoordinator. Apenas cria e recebe notificações de atividades do caso, gerando um *log* de notificações recebidas e um relatório ao final do caso. Ao detectar o final do plano, realiza *garbage collection*, sem que haja a integração como o *HistoryServer* (passagem de dados de execução para este servidor); Não instancia as atividades de sincronização no início do caso. Este último trabalho é feito pelos *ActivityManagers*. Não foi implementada a detecção e resolução de erros.

ProcessCoordinator. Cria coordenadores de casos, passando para estes objetos os dados iniciais do processo. Recebe notificações destes coordenadores, gerando *logs* e relatórios de execução. Não foi implementada a detecção e resolução de erros.

RoleCoordinator. Armazena apenas usuários e papéis. Realiza consultas que listam apenas os usuários associados a um determinado papel, aquele que o objeto *RoleCoordinator* está representando no momento; Não realiza consultas mais complexas, envolvendo dados históricos, por exemplo.

TaskList. Implementa o protocolo de escolha da atividade corrente, realizado junto com o *ActivityManager*. Implementa a política de aceitação de atividades onde todas as atividades atribuídas a um determinado usuário são aceitas.

Não foram implementados, desta forma, os objetos: *HistoryServer, BackupServer* e *GatewayActivity.* A interação entre estes objetos e os outros servidores da arquitetura não ocorreu.

7.2 Soluções de Implementação

Descrevemos nesta sessão, os principais problemas encontrados durante a implementação do protótipo da arquitetura WONDER, descrevendo as soluções adotadas no contorno destas dificuldades.

7.2.1 Workflow Object Factory

Este objeto auxiliar foi criado de maneira a implementar a mobilidade do agente na arquitetura WONDER. Restrições internas do OrbixWeb impedem que um servidor crie uma outro servidor de seu mesmo tipo. Quando isto é realizado de maneira direta, usando o comando *bind()* a partir do próprio servidor, ao invés de ser criado um novo objeto, o orbixd retorna uma referência para o servidor corrente (*self-reference*). De maneira a contornar este problema, foi criado um servidor a parte chamado *WorkflowObjectFactory*.

O *WorkflowObjectFactory* nada mais é que um criador de servidores, possuindo operações como *createActivityManager()* que simplesmente cria uma nova instância do tipo *ActivityManager*, retornando esta nova referência como resposta. O código é mostrado abaixo.

```
public WONDER.ActivityManager createActivityManager
       (String host, String basename) {

       WONDER.ActivityManager activityMan;
activityMan = Binder.bindActivityManager(nó, basename);
       return activityMan;
  }
```

7.2.2 Persistência de Objetos e Timeout

O OrbixWeb permite especificar um tempo de inatividade (*timeout*) a partir do qual um servidor é removido da memória. Se um loader foi associado a este objeto, este pode ser salvo em um meio não volátil, no caso da arquitetura WONDER, no Repositório de Objetos local. O

timeout é contabilizado a partir da última invocação de uma operação na interface do servidor em questão.

Durante os testes, foram utilizados os valores de 5 minutos para atividades (*SynchronizationActivity* e *ActivityManager*) e 10 minutos para coordenadores. Por algum motivo ainda não muito bem compreendido, foi observado que o OrbixWeb contabiliza este *timeout* a partir do momento que o servidor é criado. Em testes de carga do sistema as atividades eram removidas da memória antes de completarem o processo de seqüenciamento da próxima atividade quando: o tempo de seqüenciamento e de execução ultrapassava estes valores, devido a maiores atrasos da máquina, ou durante testes onde o tempo de duração dos *wrappers* associados a cada atividade superava este *timeout*. Nestas ocasiões, o caso nunca chegava ao seu final.

Após várias tentativas sem sucesso, a alternativa encontrada foi aumentar o valor do *timeout* destas atividades. A atividade foi configurada para ser removida da memória por si só, invocando a operação *delete(),* de sua própria interface, quando esta chega ao final. Neste momento, o estado da atividade é salvo em disco para posterior consulta. Logo após, a atividade é removida da memória.

7.2.3 Utilização do Espaço de Nomes

Como não pode ser assumido que um servidor estará ativo no nó onde foi criado indefinidamente, antes de ser realizada qualquer invocação remota de operação em um servidor da arquitetura WONDER, é realizado um *bind()* para este objeto. Para tal, o nome do servidor é passando como parâmetro. Desta maneira, não são armazenadas referências de objetos (IORs) mas o nome do servidor em questão. Desta forma, a invocação de uma operação só é realizada após a realização da (re)conexão.

Outro fator que obriga a realização de um *bind()* antes da invocação de operações remotas é incapacidade de salvar referências de objetos remotos. A persistência dos servidores foi implementada utilizando o *object serialization* da API Java. Este recurso não permite a serialização de referências remotas, gerando uma exceção caso isto seja tentado. Desta forma, ao invés de armazenar referências, que são marcadas como *transient*, cada servidor da arquitetura armazena uma lista de nomes de objetos WONDER com os quais este se relaciona (invoca operações). Quando um objeto é serializado, todas suas referências a servidores CORBA são, então, perdidas, restando somente o nome destes servidores.

7.2.4 LockManager

Outro problema encontrado durante os testes é descrito a seguir. Atividades que foram desativadas da memória são normalmente “acordadas” após o término de sua execução, por atividades posteriores que coletam dados durante o processo de seqüenciamento da próxima atividade.

Em alguns casos, durante o processo de salvamento do estado de um objeto, devido a *timeout* ou finalização de execução, uma requisição de *bind()* para este servidor, feita por um cliente, era recebida. O orbixd iniciava, desta forma, o processo de leitura do arquivo contendo o estado do objeto. Isto ocorria no exato momento em que este arquivo (.ser) estava sendo criado pela serialização do objeto requisitado. Nestas ocasiões, erros de leitura e exceções de criação de servidores eram gerados, certamente provocados por leituras inconsistentes de estado dos servidores em serialização.

Na tentativa de resolver este problema, foi criado um gerente de *locks*, executando em cada nó do sistema, que centralizava e controlava este processo. Durante uma serialização, um *lock* de escrita deveria ser conseguido, junto com o *LockManager* local. De forma análoga, um *lock* de leitura deveria ser obtido junto a este servidor, antes que um objeto local pudesse ser lido. Foi adotada a política de múltiplos leitores e um escritor. Uma instância local de *LockManager* gerenciava locks de todos os objetos de um nó.

O *LockManager* não funcionou. O erro continuou persistindo. A solução adotada foi aumentar o *timeout* dos servidores de maneira a tornar estas colisões em eventos raros. Isso foi suficiente para a realização dos testes no sistema sem que este erro ocorresse.

7.2.5 OrbixWeb e Escalabilidade

Durante a realização dos testes ocorreram vários erros no sistema. A seguir são descritos os principais erros e as medidas adotadas para contorna-los, quando possível.

7.2.5.1 Loop de Espera e Nova Tentativa

Quando o número de casos e atividades concorrentes crescia acima de um determinado patamar, tipicamente 10 casos concorrentes, com 10 atividades cada, tentativas de criar um novo objeto, usando o comando *bind()*, resultavam no seguinte erro.

```
ERRO: org.omg.CORBA.COMM_FAILURE
```

Por algum motivo interno ocasionado à carga do sistema, o OrbixWeb não conseguia instanciar um novo servidor, o que na prática implica na criação de um novo processo no sistema. De maneira a contornar este problema, para toda tentativa de realização de *bind* do programa, forma feitas as alterações descritas a seguir.

```
int retryTurn = 0;
int retryTimes = 10;
do {
   try {
      // Não somente este mas para todos os servidores do WONDER
      server = LockManagerHelper.bind(":LockManagerSrv");
   } catch (SystemException ex) {
        System.out.println(ex.toString());
        server = null;
        retryTurn ++;
        wait(waitTimeout);
   }
} while (doRetry && (server == null) && (retryTurn < retryTimes));
```

Desta maneira, são realizadas até *retryTimes* tentativas de conexão antes de ser gerado um erro de criação de servidores. O *retryTimes* utilizado foi de 10 tentativas. Durante as execuções do WONDER, observou-se que, na maioria das vezes, a conexão ou a criação de um novo objeto era bem sucedida a partir da segunda tentativa, ou seja, *retryTurn*=1. O *waitTimeout* escolhido foi de 500ms.

O uso deste artifício fez com que fossem criados até 30 casos concorrentes contendo 30 atividades cada. Este número foi limitado pelo problema descrito a seguir.

7.2.5.2 Esgotamento de Recursos do Ambiente

Ao ser utilizada a arquitetura WONDER na execução de um número relativamente grande de atividades concorrentes em um mesmo nó, simulando um sistema centralizado, em uma das máquinas mais potentes do Instituto de Computação, a anhumas, (Vide Apêndice B), os recursos deste sistema esgotaram. O OrbixWeb não conseguia mais criar servidores. O seguinte erro foi gerado.

```
org.omg.CORBA.INTERNAL: remote exception - ORB internal error:

    Internal Error in activator
    Actual system exception is `(unknown)'
    (please contact Iona Technologies)
```

Este erro ocorria quando tentava-se executar 30 casos concorrentes, iguais, contendo 30 atividades cada em um mesmo nó (caso centralizado). O caso nada mais era que uma seqüência linear de atividades. Sua incidência era crítica a partir do caso número 28, ou seja, havia 28 casos concorrentes, no momento em que o primeiro estava quase terminando (vide capítulo de testes).

Por ser um problema interno do OrbixWeb, não foi possível resolve-lo. Este fato serviu como limitante do número máximo de casos concorrentes usados no teste do protótipo.

Outro problema interno do OrbixWeb é a o limite do intervalo de números de portas que podem ser alocadas dinamicamente. Quando este limite era excedido, tipicamente da porta 2000 à 3000, exceções eram geradas. Tentativas de aumentar este limite, mudando parâmetros de configuração do ORB não foram bem sucedidas.

7.2.6 Peculiaridades do OrbixWeb

Os principais erros de programação e de utilização do OrbixWeb são descritos no Apêndice D. Dentre eles é destacado o seguinte erro.

7.2.6.1 Tamanho do Marker do Objeto

O OrbixWeb permite associar a cada objeto criado no modo *unshared* uma *string* que permite identifica-lo. Esta *string* é conhecida como *marker* e é utilizada pelo orbixd para implementar um de serviço de nomes local, permitindo a obtenção de uma referência de objetos através do uso deste nome em comandos *bind()*.

Ao criarmos servidores com *marker* mais longos que 59 caracteres obtínhamos o erro seguinte erro:

```
ERROR: org.omg.CORBA.COMM_FAILURE
```

Por algum motivo interno, o orbixd não conseguia instanciar tais objetos. A solução adotada foi reduzir o comprimento dos nomes utilizados para os servidores do sistema.

Capítulo 8

Testes de Desempenho

São descritos neste capítulo os principais testes de desempenho realizados com o protótipo da arquitetura WONDER, discutindo e comparando os resultados obtidos.

8.1 Objetivos

Foram definidos e realizados vários testes de desempenho de maneira a extrair as características da arquitetura WONDER. Para tal, foram utilizados vários cenários de execução onde os componentes da arquitetura são dispostos em diferentes configurações de distribuição e centralização.

Nos testes aqui descritos, são estudas as variações dos tempos das várias etapas de execução das instâncias de processo, frente ao aumento gradual dos seguintes parâmetros do ambiente de teste: o número de casos concorrentes, na presença e na ausência de processamento, e o volume de dados trocados entre atividades dos casos de teste. Estas variações são estudadas em diversas configurações de distribuição dos servidores, abrangendo ambientes totalmente centralizados a ambientes distribuídos. Dados destes cenários são posteriormente comparados entre si.

Por restrições relacionadas ao tempo disponível para a implementação do projeto, assim como devido à falta de disponibilidade de dados de desempenho de outros sistemas, não foram realizados testes comparativos envolvendo outras arquiteturas com funcionalidades similares. Também não foram realizados testes utilizando processos de workflows reais, que modelam tarefas rotineiras de empresas ou entidades governamentais.

Nos testes aqui descritos, foram utilizadas definições de processos bastante simples, possuindo uma seqüência única de atividades que executam, cada uma, por um período relativamente

curto de tempo, tipicamente de apenas alguns segundos. Em processos reais, estes tempos são em geral mais longos e com durações aleatórias, podendo variar de minutos a dias. Este maior tempo de duração das atividades reais faz com que, na prática, haja uma menor sobrecarga do sistema distribuído já que grande parte do tempo de execução da atividade é gasto em operações de E/S do ator (digitação de textos, preenchimento de formulários, leitura de documentos, e outros), operações que normalmente utilizam pouca CPU e banda passante.

Desta forma, apesar de não serem usadas definições de processos reais, os testes aqui empregados imprimem uma carga maior ao ambiente de testes, permitindo estudar seus limites com maior facilidade, obtendo um melhor controle dos parâmetros estudados. Apesar de utilizar um conjunto de máquinas reduzido, envolvendo um número de casos concorrente não muito grande, os dados aqui obtidos permitem estimar o comportamento do sistema em situações reais de larga escala.

8.2 Descrição dos Testes

Foram realizados 3 tipos básicos de testes onde é estudado o comportamento dos objetos da arquitetura frente à variação do número de servidores concorrentes, da carga de processamento e do volume de dados do caso. Nos três cenários, a arquitetura foi testada em 2 configurações básicas: totalmente centralizada e distribuída. Em cada teste, uma variável foi selecionada, fixando-se os outros parâmetros do ambiente de testes. Cada bateria de testes compreendeu a execução de subtestes, cada um com um valor diferente para o parâmetro estudado. Os resultados de cada instância foram representados graficamente, permitindo gerar curvas características do sistema, de acordo com a variação do parâmetro escolhido.

Foram realizados ainda teses com o objetivo de determinar o *overhead* da arquitetura na ausência de aplicações invocadas e de dados do caso.

Os valores das variáveis foram escolhidos de maneira a levar determinados aspectos do sistema ao seu limite, sempre que possível. Os limites do número de casos concorrentes, com ou sem processamento, só puderam ser determinados para os testes em ambientes centralizados, ou seja, todos os objetos servidores executavam em um mesmo nó. Estes limites centralizados foram usados em testes envolvendo configurações distribuídas, por motivo de comparação, de maneira a observar o impacto da distribuição no desempenho do sistema.

Os limites da arquitetura em configurações distribuídas não foram testados já que, nestes casos, como os testes mostrarão, é sempre possível aumentar a capacidade de processamento do sistema, adicionando-se mais um nó ao ambiente de testes. Isto é iterado até o ponto de haver apenas um objeto servidor do tipo ActivityManager, ativo na memória, por máquina, que representaria um cenário típico, onde os atores realizam apenas uma tarefa por vez.

Em cada bateria de testes, são executados casos simples, compostos por uma única seqüência de atividades iguais, sem *splits* e *joins*. Como o objetivo dos testes é determinar o impacto da arquitetura no sistema, medindo a variação dos tempos de execução dos elementos do caso, o uso de *AND* ou *OR-splits* não traz contribuições a estas medidas. Desta forma, um *OR-Split-Join* é revertido em uma única seqüência linear (a escolhida dentre os disponíveis ramos do OR), tornando sua execução semelhante a uma seqüência. De maneira semelhante, um *AND-split-join*, embora gere uma carga maior no sistema, devido à execução de seqüências paralelas, possui um tempo de execução igual a da seqüência paralela mais lenta. Em última análise, é equivalente a uma seqüência única, onde ruídos (processamento, criação de processos, realização de E/S e uso de CPU) provocados pelas atividades dos outros ramos paralelos do AND, são introduzidos no ambiente. O uso de uma seqüência única permite, desta forma, gerar casos de teste mais previsíveis e melhor comportados, facilitando o controle dos parâmetros da arquitetura.

Nos testes onde são estudados os impactos da variação do número de casos concorrentes na arquitetura, o volume de dados trocado entre atividades consecutivas é mantido fixo (121KB nos testes). Este conjunto de dados é composto por 3 documentos *.doc* de 4, 6 e 8 páginas respectivamente. Este valor foi escolhido arbitrariamente. É importante lembrar que, durante a transferência de dados de um nó a outro, este volume torna-se menor, visto que o *stream* de bytes serializado é comprimido, durante a serialização, com o pacote *Java.util.zip* da API Java.

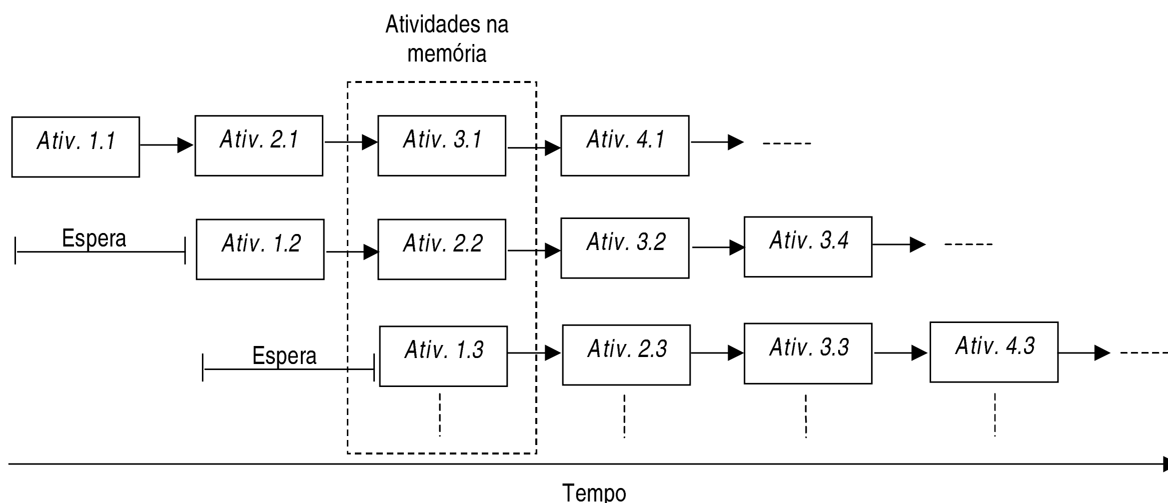

Figura 25: Exemplo de um ambiente de testes com vários casos em paralelo.

Em todos os testes aqui descritos, os casos são criados um após o outro, observando um tempo de espera de algumas dezenas de segundos. Cada caso possui um número fixo de atividades. Este número é, em geral, igual ao número máximo de casos paralelos a serem gerados na bateria de testes. O objetivo é executar várias seqüências de testes, variando o número de casos paralelos em cada teste, sem contudo alterar o número de atividades de cada caso. Um atraso fixo, entre a criação de casos concorrentes, é definido de maneira a evitar um pico de carga inicial, que seria gasto na criação de todos os casos concorrentes ao mesmo tempo. O uso desta estratégia permite levar o sistema a seu ponto de operação, de maneira gradual. Neste ponto de ope-

ração, o número de atividades ativas na memória é, em média, igual ao número de casos concorrentes. Um exemplo é mostrado na Figura 25.

No exemplo da Figura 25, cada atividade dispara somente um único *wrapper*. Este *wrapper* executa um *shell script* que, por sua vez, dispara uma determinada aplicação que executa durante um determinado período de tempo.

O valor do tempo de espera entre a criação de cada caso foi determinado de maneira a permitir a inicialização do coordenador de caso de cada instância de processo, juntamente com sua primeira atividade. Nos testes, empregou-se 20 e 30 segundos para este valor.

As baterias de testes são realizadas sempre em duas configurações. Na primeira, todos os objetos da arquitetura executam em um mesmo nó; na segunda configuração, as atividades do caso padrão são configuradas para executarem, de maneira alternada, em um subconjunto de dois ou mais nós. Esta estratégia permite a execução de atividades consecutivas de um mesmo caso em nós diferentes. Isto faz com que a carga de execução das atividades do caso seja dividida entre estas duas máquinas. Os coordenadores são criados em um dos dois nós utilizados, a que possuir maior memória. Um exemplo de execução de um caso, de maneira distribuída, é mostrado na Figura 26.

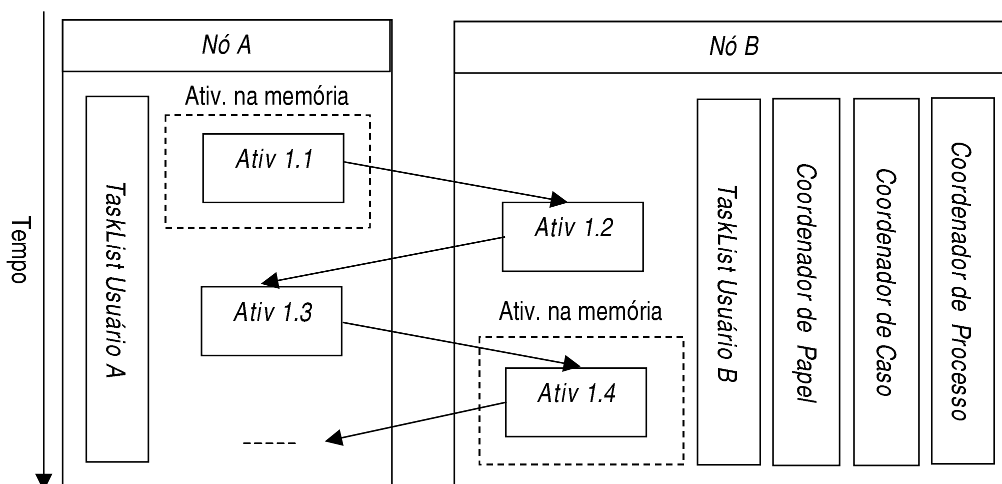

Figura 26: Execução alternada de atividades consecutivas em testes distribuídos.

A alternância de nós onde as atividades executam obriga a realização da troca de dados através da rede. Este tipo de teste permite simular o comportamento da arquitetura em um ambiente distribuído, já que todas as mensagens e dados trocados entre as atividades consecutivas trafegam através da rede. Este teste não simula, contudo, a execução totalmente distribuída já que, para testes onde há vários casos em execução simultânea, um mesmo nó executa de forma concorrente atividades de casos diferentes. Isto faz com que o uso dos recursos de um nó por uma atividade influencie no tempo de execução de outras atividades.

Do ponto de vista do sistema como um todo, o que existe é, portanto, uma distribuição de carga entre dois ou mais nós. Do ponto de vista de cada caso, a execução é realizada de maneira a

simular uma execução totalmente distribuída, visto que somente uma atividade de cada caso está ativa, em um único nó, ao mesmo tempo.

Em todos os testes realizados, o procedimento de *garbage collection* não foi realizado. Este procedimento apaga os arquivos de dados, de estado dos objetos, de relatórios e de *log,* gerados pelos coordenadores e pelas atividades do caso. O que não é desejável, já que esta informação é utilizada para medir os tempos de execução das atividades e coordenadores dos casos ao final da bateria de testes. A ativação deste procedimento não influencia nos testes comparativos, pois seu uso representa apenas uma constante adicionada ao tempo de execução de todos os casos.

8.2.1 Ruídos

A maioria dos testes foi realizada nas máquinas anhumas e tigre, araguaia e iguacu. Uma descrição mais detalhada de suas configurações é apresentada no Apêndice B. Estas máquinas, com exceção da anhumas, são de livre acesso aos alunos de pós-graduação do IC. Desta forma, durante a realização dos testes, não foi possível obter um sistema completamente livre da influência de outros usuários.

Os testes foram, executados, sempre que possível, em momentos onde o uso da CPU destas máquinas estava em menos de 10%, aproximadamente. O que não impediu que, durante estes testes, alguma influência aleatória ocasionada por processos disparados por outros usuários ocorresse.

Nos testes apresentados neste capítulo, a presença de ruídos pode ser mais facilmente observada nos gráficos onde são apresentados os tempos médios de execução dos *wrappers*. Como as operação executada pela atividade invocada é apenas um *sleep*, a variação do tempo de execução destes processos está intimamente relacionada com a carga do sistema, já que o processamento envolvido em sua criação e destruição é realizada pelo sistema operacional. Desta forma, de uma maneira indireta, as variações dos tempos médios de execução dos *wrappers* representam a medida das diferenças de carga do sistema operacional.

8.3 Relação dos Testes

A seguir são descritos, em maior detalhe, os tipos de testes realizados.

8.3.1 Análise do *Overhead* da Arquitetura

Este conjunto de testes objetiva estudar o impacto dos objetos da arquitetura nos tempos de execução das atividades do SGWF, na ausência de aplicações invocadas e do volume de dados trocado entre as atividades. Este conjunto de testes permite determinar o tempo mínimo necessário para a execução das atividades e dos caso nas máquinas utilizadas nos testes. Mede-se desta forma, somente os tempos associados à cada fase de execução e migração do agente móvel nas máquinas utilizadas nos testes.

São realizados testes centralizados, onde todos os objetos da arquitetura são executados no mesmo nó, e testes distribuídos, onde as atividades alternam entre duas máquinas. Nestes últimos testes, os coordenadores executam em uma terceira máquina (vide Figura 27), fazendo com que toda a comunicação entre atividades e coordenadores ocorra através da rede.

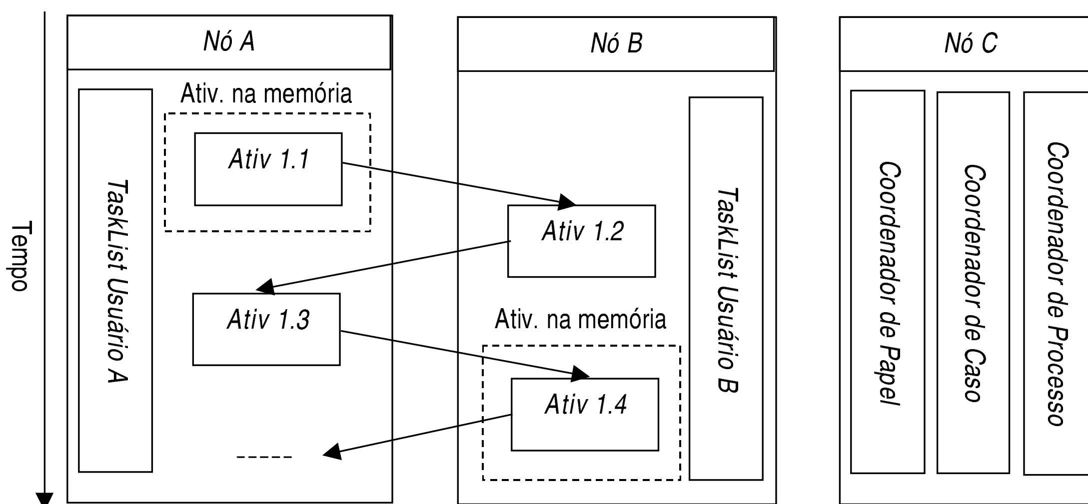

Figura 27: Execução alternada de atividades em nós distribuídos. Os coordenadores são criados em um nó à parte

8.3.2 Estudo da Variação do Número de Casos Concorrentes - Sem Processamento

Este conjunto de testes tem como objetivo medir a variação os tempos de execução dos objetos da arquitetura frente ao aumento do número de casos concorrentes. Nestes testes, cada atividade dispara um único *wrapper*. Este *wrapper* invoca um *shell script* que executa o comando "*sleep 20*" (espera por 20 segundos), simulando uma aplicação que não realiza processamento.

O uso de aplicações invocadas que não realizavam processamento objetiva destacar o impacto da arquitetura no sistema, sem que isto provoque a degradação de desempenho promovida por aplicações externas ao SGWF, ao mesmo tempo que mantém as atividades em estado de execução por um período maior de tempo. Estas aplicações geralmente consomem recursos de processamento e E/S, que reduzem o desempenho do sistema como um todo. Devido a ruídos provocados por outros usuários, contudo, isto não pôde ser completamente evitado.

8.3.3 Estudo da Variação do Número de Casos Concorrentes - Com Processamento

Este conjunto de testes objetiva medir os tempos de execução da arquitetura frente a situações extremas, onde as aplicações invocadas utilizam uma carga excessiva de processamento. Para tal, é utilizada uma aplicação invocada que realiza a ordenação de 1000 números aleatórios, usando um algoritmo pouco eficiente, o *bubblesort*, cuja complexidade é $O(n^2)$. Esta ordenação é realizada por uma aplicação Java disparada por um *shell script*, executado pelo *wrapper*. A aplicação gera os 1000 números aleatórios e, em seguida, os ordena. A semente (*seed*) utilizada no gerador aleatório é o relógio do sistema. O teste foi executado em cenários centralizados e distribuídos.

8.3.4 Estudo da Variação do Volume de Dados Trocado

O objetivo deste conjunto de testes é medir a influência do volume de dados, trocados entre atividades consecutivas, no tempo de execução total dos casos da arquitetura, em especial no tempo de seqüenciamento destas atividades. Os testes são realizados usando sucessivas execuções de um único caso, composto por 20 atividades iguais. Estas atividades disparam *wrappers* que executam *shell scripts* com o comando "*sleep 10*". A cada iteração (subteste), o volume de dados utilizado pelas atividades dos casos é aumentado.

8.4 Medidas Empregadas

Durante os testes foram coletadas várias medidas temporais. Um diagrama temporal mostrando as principais medidas é apresentado na Figura 28. Neste exemplo é mostrado um caso formado por três atividades. Os tempo de vida das atividades, com seus estados internos, são representados na forma de retângulos. Notificações e chamadas de operações são representadas como setas cheias. Setas tracejadas representam operações invocadas por *TaskLists*, não mostrados na figura. Somente um *wrapper* é disparado por atividade neste exemplo.

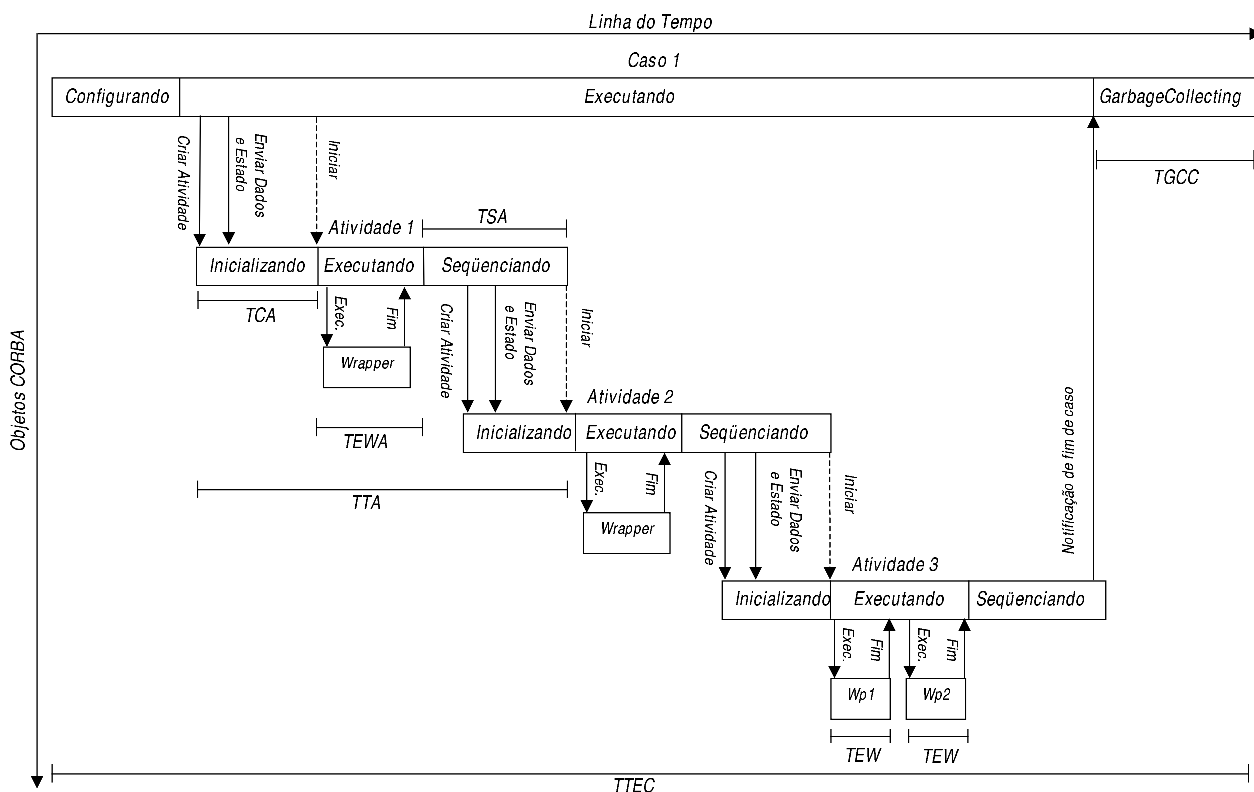

Figura 28: Intervalos de Tempos Medidos

Estas medidas são coletadas pelo coordenador de casos através do recebimento de notificações enviadas pelas atividades durante cada fase de sua execução.

Uma descrição das medidas coletadas nos testes é feita a seguir.

Tempo de seqüenciamento de uma atividade (TSA). É o tempo compreendido entre o recebimento do evento SEQUENCING e do evento FINISHED, por um coordenador de caso. Corresponde ao tempo transcorrido em uma atividade, desde o final da execução do último *wrapper* até sua finalização. Desta forma, corresponde à soma do tempo negociação que precede o seqüenciamento (TNA) somado ao tempo de criação do próximo *ActivityManager* (TCrA), somado ao tempo de configuração da próxima atividade (TCA), alem de contar a inclusão da nova atividade no *TaskList* do ator. (Vide Figura 29).

Tempo de negociação de uma atividade (TNA). Representa o tempo necessário para que um *ActivityManager* determine o tipo da próxima atividade, recupere a lista de usuários do coordenador de papel, escolha o usuário para realizar a próxima atividade, negocie a escolha com o *TaskList* deste ator.

Tempo de criação de uma atividade (TCrA). Corresponde ao tempo necessário para a atividade anterior criar um novo *ActivityManager.*

Tempo de configuração de uma atividade (TCA). Corresponde ao tempo necessário para a atividade anterior: coletar e serializar dados da próxima atividade juntamente com o estado do caso, enviar este data container à nova atividade, invocando *setData*().

Tempo de inicialização de uma atividade (TIA): É o tempo que engloba a criação e configuração de uma atividade, ou seja, o tempo transcorrido entre o *bind()* (criação de um novo *ActivityManager*) e a invocação do comando *init().* $TIA_i = TCrA_i + TCA_i$ aproximadamente.

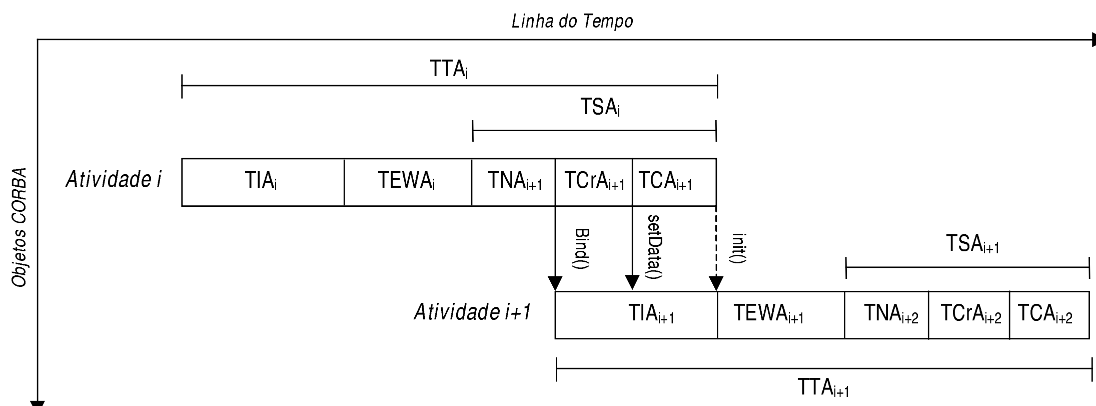

Figura 29: Tempos associados a duas atividades consecutivas de um mesmo caso

Tempo total de seqüenciamento de um caso (TTSC). É a soma dos tempos de seqüenciamento de todas as atividades de um caso. TTSC = soma (todos os TSA do caso).

Tempo médio de seqüenciamento das atividades de um caso (TMSC). Corresponde à média dos tempos de seqüenciamento de todas as suas atividades. TMSC = Média (todos os TTSC).

Tempo total de seqüenciamento (TTS). É a soma dos tempos de seqüenciamento de todos os casos de um teste. TTS = Soma(todos os TTSC).

Tempo médio de seqüenciamento (TMS). É a média aritmética dos tempos totais de seqüenciamento dos casos de um teste. TMS = Média (todos os TSA).

Tempo de execução dos *wrappers* de uma atividade (TEWA). Corresponde à soma dos tempos de execução dos *wrappers* (TEW_i) de uma atividade. É a soma dos tempos transcorridos entre os eventos EXECUTING e FINISH_EXEC enviados pelos *wrappers* ao coordenador de caso. TEWA = Soma (todos os TEW_i).

Tempo total de execução dos wrappers (TTEW). Representa a soma dos tempos de execução dos *wrappers* de todas as atividades em um conjunto de casos. TTEW = Soma (todos os TEWA).

Tempo médio de execução dos *wrappers* (TMEW). Corresponde à média aritmética de todos tempos de execução dos *wrappers* das atividades de um conjunto de casos. TMEW = Média (todos os TEWA).

Tempo total da atividade (TTA). Tempo transcorrido desde a criação até a finalização e salvamento de uma atividade. É o tempo compreendido entre o recebimento dos eventos STARTING e FINISHED, enviados pelas atividades ao coordenador de caso. Esta medida não leva em conta o tempo de execução das atividades quando estas são despertadas por atividades posteriores para a coleta de dados.

Tempo médio da atividade (TMA). É a média dos TTAs de um caso.

Tempo de realização de *Garbage Collection* de um caso (TGCC). Tempo computado entre o início e o fim deste processo. É computado pelo coordenador de processo pelo recebimento dos eventos GARBAGE_COLLECTING e GARBAGE_COLLECTED.

Tempo de execução de um caso (TTEC). Compreende o tempo transcorrido desde a criação do coordenador de caso até sua finalização. É computado pelo coordenador de processo como sendo o tempo entre o recebimento dos eventos CREATED/AWAKED e o FINISHED, enviado pelos coordenadores de casos. TTEC = a (TC+TEWA) +TGCC. Onde 'a' é o número de atividades do caso.

Tempo total dos casos de um teste (TTC). Representa a soma dos tempos totais de execução dos casos deste teste. TTC = Soma(todos os TTEC).

Tempo total de seqüenciamento (TTS). É a soma do tempo de seqüenciamento de todas as atividades de um conjunto de casos, incluindo os tempos de seqüenciamento iniciais, que ocorrem durante a criação da primeira atividade de cada caso. TTS = Soma (todos os TSA) = Soma (todos os TTSC).

8.5 Análise dos Resultados

Os dados dos testes foram coletados através da análise dos *logs* e dos relatórios de eventos gerados pelos coordenadores de processo e de caso. Atividades enviam eventos a seus coordenadores de casos. Coordenadores de casos enviam eventos para os coordenadores de processos. A cada tipo de evento é associado um horário de recebimento. Estes eventos são armazenados nos interpretadores de planos dos coordenadores, ao mesmo tempo em que são escritos em *logs* no sistema de arquivos.

Dados de desempenho para ambientes distribuídos e centralizados, coletados durante os testes, são apresentados e comparados graficamente. Para cada valor de x fixado durante o experi-

mento, um valor y foi medido. Nesta comparação foi feita a regressão linear dos pontos medidos utilizando-se o método dos quadrados mínimos.

Para as retas de aproximação, foram geradas a fórmula da reta juntamente com o coeficiente de determinação (R^2) da regressão. R^2 é assume valores entre 0 e 1, inclusive, e representa o quão bem a função de aproximação explica os pontos medidos. Ou seja, o quão próximo está a reta aproximada das medidas reais. Desta forma, R^2=1 implica que a curva obtida pela regressão aproxima (ou passa por) 100% os pontos (x,y) medidos. Um valor de R^2 igual a 0,90 por exemplo, informa que a regressão linear reduz a variabilidade (erro) dos valores de y de 90%. [HL87, pp.280] Quanto mais próximo de 1 (100%) é o valor de R^2, mais fiel é a aproximação linear, ou seja, a probabilidade da função real ser linear é maior.

8.6 Testes

Apresentamos, a seguir, os testes realizados, seus parâmetros, resultados e suas comparações em configurações centralizadas e distribuídas.

8.6.1 Análise do *Overhead* da Arquitetura

Os parâmetros utilizados nos testes distribuído e centralizado são apresentados na Tabela 14.

Timeout das atividades (minutos)	5
Timeout dos coordenadores (Case, Process and Role) (minutos)	10
Timeout dos servidores *TaskList* (minutos)	5
Número de atividades no caso (seqüência única)	20
Número de usuários (*TaskLists*)	2
Número de coordenadores de processos	1
Linha de comando da aplicação Invocada	Não há
Volume de dados trocado entre atividades (Bytes)	1338 a 1342
Realiza *garbage collection* ?	Não

Tabela 14: Dados de Execução do ambiente de teste centralizado, sem atividades invocadas e sem dados trocados. Máquina iguacu.

O único dado trocado entre as atividades é o estado do caso, cujo volume é de 1338B na primeira atividade e 1346B na última. Este volume aumenta gradualmente, conforme dados de desempenho são adicionados ao interpretador de plano, trocado entre atividades.

8.6.1.1 Testes Centralizados

Teste: Estudo do tempo de execução das atividades da arquitetura na ausência de chamada de aplicações invocadas e de transferência de dados do caso em ambiente centralizado. Todos os objetos executam na máquina iguacu.

Tabela de Resultados

Para cada fase da execução de uma atividade (vide Figura 29), foi calculada a média dos valores obtidos para as 20 atividades executadas. Estes dados médios, são apresentados na Tabela 15 a seguir. Os percentuais destes valores, relativos ao tempo de execução total, são apresentados na Tabela 16.

Máquina	TTA (ms)	TSA(ms)	TIA (ms)	TEWA (ms)	TNA (ms)	TCrA (ms)	TCA (ms)
Araguaia	10917	6252	4665	0	403	4496	1093
Iguacu	12558	7337	5221	0	788	4718	1541
Anhumas	8610	4726	3884	0	385	3158	944
Tigre	14116	7872	6244	0	641	5393	1530
Tuiuiu	23137	13945	9192	0	1372	8533	3288
Gaivota	22659	13502	9157	0	1372	8175	3141

Tabela 15: Dados médios de execução dos testes envolvendo a execução de casos em ambientes centralizados, sem processamento e sem troca de dados do caso. Máquinas: araguaia, iguacu, anhumas e tigre.

Máquina	TTA (ms)	TSA(ms)	TIA (ms)	TEWA (ms)	TNA (ms)	TCrA (ms)	TCA (ms)
Araguaia	100,00%	57,27%	42,73%	0%	3,69%	41,18%	10,01%
Iguacu	100,00%	58,42%	41,58%	0%	4,57%	40,83%	10,38%
Anhumas	100,00%	59,28%	40,72%	0%	4,70%	38,86%	12,78%
Tigre	100,00%	55,77%	44,23%	0%	4,54%	38,20%	10,84%
Tuiuiu	100,00%	60,27%	39,73%	0%	5,93%	36,88%	14,21%
Gaivota	100,00%	59,59%	40,41%	0%	6,05%	36,08%	13,86%
MÉDIA	**100,00%**	**57,68%**	**42,32%**	**0%**	**4,38%**	**39,77%**	**11,00%**

Tabela 16: Percentual relativo dos dados médios de execução dos testes envolvendo a execução de casos em ambientes centralizados, sem processamento e sem troca de dados do caso. Máquinas: araguaia, iguacu, anhumas e tigre

8.6.1.2 Testes Distribuídos

Teste: Estudo do tempo de execução das atividades da arquitetura na ausência de chamada de aplicações invocadas e de transferência de dados do caso em ambiente distribuído.

Descrição: Um caso contendo 20 atividades iguais é executado de maneira distribuída em dois testes. O primeiro, realizado nas máquinas araguaia, iguacu e anhumas, e o segundo nas máquinas anhumas, tigre e araguaia. No primeiro teste, os coordenadores de caso, processo e papel são criados na máquina anhumas. No segundo, estes objetos são criados na máquina araguaia. Em ambos os testes, as atividades executam, de maneira alternada, nas máquinas iguacu e araguaia, no primeiro teste, e nas máquinas anhumas e tigre, no segundo teste, como apresentado na Figura 27. Um *TaskList* é criado em cada uma das máquinas onde as atividades executam.

Tabela de Resultados

As médias dos tempos coletados no teste distribuído são apresentados na Tabela 17 a seguir. Os percentuais destes valores para cada teste são apresentados na Tabela 18.

Máquina	TTA (ms)	TSA (ms)	TIA (ms)	TEWA (ms)	TNA (ms)	TCrA (ms)	TCA (ms)
Araguaia-iguacu-anhumas	10334	5683	4651	0	399	4037	1009
Anhumas-tigre-araguaia	11681	7078	4603	0	481	5075	1243
Tuiuiu-gaivota-araguaia	19121	10975	8146	0	981	7270	2251

Tabela 17: Dados de médios de execução de testes envolvendo a execução de casos em ambientes distribuídos, sem processamento e sem troca de dados do caso. Máquinas: anhumas, araguaia e iguacu, e araguaia, tigre e anhumas.

Máquina	TTA (ms)	TSA(ms)	TIA (ms)	TEWA (ms)	TNA (ms)	TCrA (ms)	TCA (ms)
Araguaia-iguacu-anhumas	100,00%	54,99%	45,01%	0%	3,86%	39,07%	9,76%
Anhumas-tigre-araguaia	100,00%	63,03%	36,97%	0%	5,12%	44,39%	11,03%
Tuiuiu-gaivota-araguaia	100,00%	57,40%	42,60%	0%	5,13%	38,02%	11,77%
MÉDIA	**100,00%**	**59,01%**	**40,99%**	**0%**	**4,49%**	**41,73%**	**10,40%**

Tabela 18: Percentual relativo dos dados de médios de execução de testes envolvendo a execução de casos em ambientes distribuídos, sem processamento e sem troca de dados do caso. Máquinas: anhumas, araguaia e iguacu, e araguaia, tigre e anhumas.

Gráficos

Os dados da Tabela 15 e da Tabela 17 são apresentados, lado a lado, no Gráfico 1 a seguir.

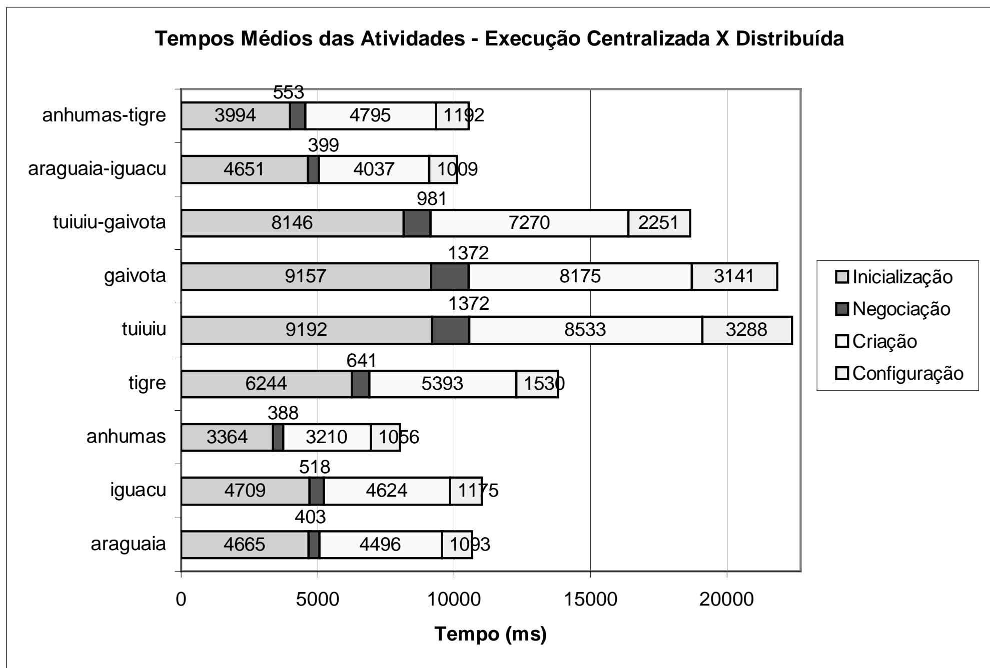

Gráfico 1: Comparação entre os tempos médios das atividades em execução centralizada e distribuída. Máquinas araguaia, iguacu, anhumas e tigre.

Os dados da Tabela 16 e da Tabela 18 são apresentados, lado a lado, no Gráfico 2.

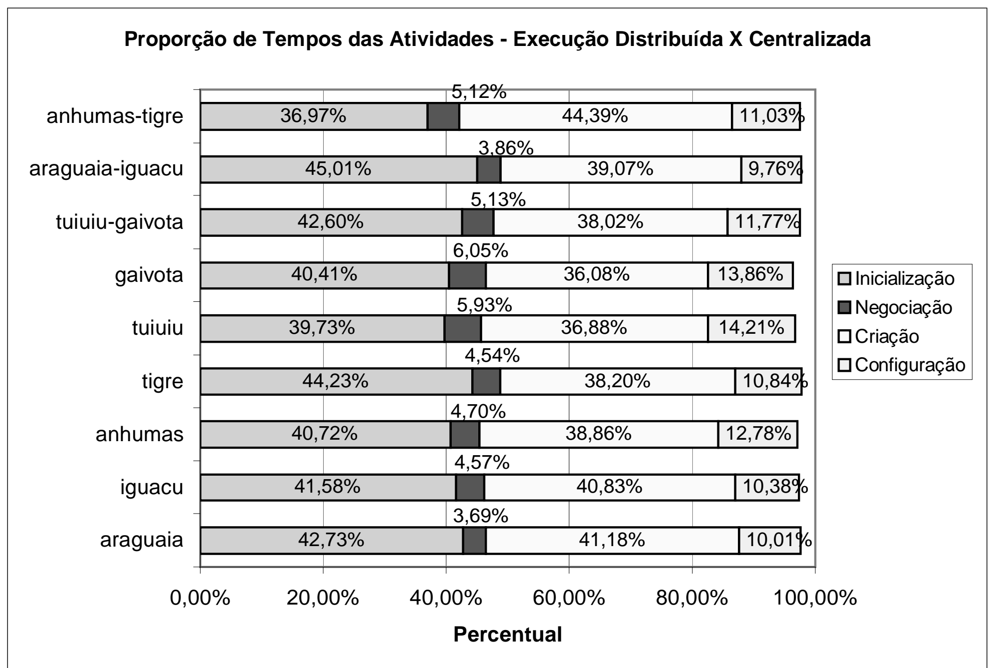

Gráfico 2: Comparação entre os tempos médios das atividades em execução centralizada e distribuída. Percentuais relativos. Máquinas araguaia, iguacu, anhumas e tigre.

Análise

Durante a realização dos testes, a máquina tigre estava com cerca de 15% de tempo de CPU livre (*idle*), o que explica os valores relativamente superiores do tempo de execução de suas atividades, se comparado com os tempos das máquinas iguacu e araguaia, nominalmente mais lentas, ou mesmo com sua "máquina irmã", a anhumas.

Como não foi possível isolar totalmente as máquinas iguacu, araguaia, tigre e anhumas, garantindo que somente um usuário estivesse acesso a estes sistemas, o mesmo conjunto de testes foi realizado em duas máquinas menos potentes, gaivota e tuiuiu, onde garantiu-se o isolamento.

Se comparados os tempos médios de execução dos casos medidos para os testes distribuídos nas máquinas araguaia-iguacu, com os tempos de suas respectivas execuções centralizadas (vide Gráfico 1), observa-se que os testes distribuídos apresentam melhor desempenho. O mesmo pode ser observado com as máquinas tuiuiu e gaivota.

O melhor desempenho das execuções distribuídas pode estar associado ao processamento realizado pelos coordenadores durante o recebimento dos eventos. Embora sejam assíncronos para quem os envia, o tratamento dos eventos enviados aos coordenadores implica em processamento e E/S extra. Ao receber os eventos, os coordenadores atualizam seus arquivos de *log*. Este acesso a disco adicional aliado ao consumo de memória e de CPU requerido por estes objetos, pode influenciar indiretamente os tempos das atividades em execução em uma mesma máquina. Nos testes realizados em ambiente distribuído, entretanto, os coordenadores são executados um nó à parte, o que isola as atividades do ruído provocado pela execução destes coordenadores. Esta influência é maior conforme é menor a capacidade de processamento das máquinas envolvidas.

O Gráfico 2 permite constatar que as proporções dos tempos envolvidos no seqüenciamento são praticamente iguais. As diferenças entre o envio de dados local ou via rede são muito pequenas. Isto é decorrente do uso do mesmo mecanismo (IIOP sobre *sockets* implementado pelo OrbixWeb) para realizar a transferência de dados e a comunicação entre as atividades. A latência de rede, no ambiente de testes (rede local do Instituto de Computação), não é muito expressiva. Desta forma, o tempos de criação de objetos e envio de mensagens são muito próximos em ambientes centralizados e distribuídos.

Observa-se também que, o tempo médio máximo de migração e inicialização de uma atividade, medido na máquina tuiuiu (SPARCStation 4), foi de 23 segundos. Este *overhead* de migração é relativamente grande para aplicações que necessitem de menor tempo de espera, ou que realizam atividades de curta duração. Entretanto, para aplicações de escritório convencionais, onde atividades podem durar de minutos a horas, esta latência de migração é aceitável.

O tempo gasto com a troca de mensagens, tempos de negociação e configuração, não representa mais que 20% do tempo total da atividade. A maior latência está associada à criação de servi-

dores CORBA, em especial, à máquinas virtuais Java que executam os objetos que implementam estes servidores.

8.6.2 Análise da Variação do Número de Casos Concorrentes - Sem Processamento

Para esta bateria de testes, utilizou-se os seguintes parâmetros, apresentados na Tabela 19 .

Timeout das atividades (minutos)	5
Timeout dos coordenadores (Case, Process and Role) (minutos)	10
Timeout dos servidores *TaskList* (minutos)	5
Tempo de espera antes da criação do próximo caso concorrente (segundos)	30
Número de atividades no caso	20
Número de usuários por máquina (*TaskLists*)	1
Número de coordenadores de processos	1
Linha de comando da aplicação invocada	Sleep 20
Realiza *garbage collection* ?	Não
Volume total de dados trocado entre atividades consecutivas (arquivos) (KB):	121

Tabela 19: Dados de Execução do ambiente de teste centralizado, sem processamento. Máquina: araguaia.ic.unicamp.br

8.6.2.1 Testes Centralizados

Teste: Estudo da influência do número de casos concorrentes, cujas atividades não realizam processamento, no tempo total de execução dos casos. Execução centralizada no nó: *araguaia.ic.unicamp.br*.

Descrição: Instâncias concorrentes de um mesmo processo, contendo 20 atividades idênticas, são executadas na máquina araguaia.

Tabela de Resultados

Foram executados 5 testes onde o número de casos concorrentes era aumentado, gradualmente, de 5 casos, a cada iteração. Os dados coletados durante a execução destes testes são mostrados na Tabela 20.

# Casos Paralelos	TMC (seg.)	TMC (ms)	TTC (ms)	TTEW (ms)	TMEW (ms)	TTS (ms)	TMS (ms)
1	651,96	651956,00	651956,00	440810,00	22040,00	115893,00	6099,00
5	962,68	962676,20	4813381,00	2441485,00	24414,40	1432518,00	15078,80
10	1177,59	1177591,50	11775915,00	4710790,00	23553,50	4165767,00	21924,60
15	1683,79	1683786,87	25256803,00	7113017,00	23709,60	10043184,00	35238,80
20	2268,99	2268989,50	45379790,00	9593112,00	23982,30	19049509,00	50129,85

Tabela 20: Dados de Execução dos testes envolvendo 1 a 20 casos concorrentes em ambiente centralizado. Máquina: araguaia.ic.unicamp.br.

Gráficos

Os dados da Tabela 20 são representados nos gráficos a seguir.

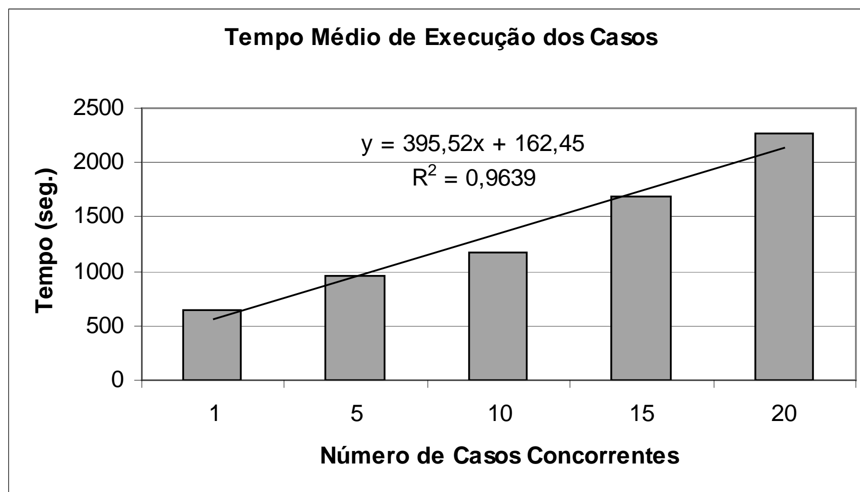

Gráfico 3: Tempo médio de execução do caso X Número de casos concorrentes. 1 a 20 casos em ambiente centralizado. Máquina araguaia

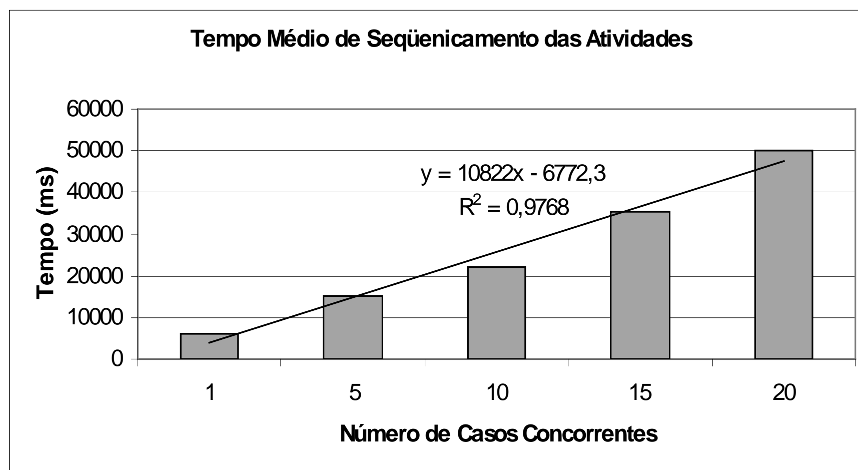

Gráfico 4: Tempo médio de seqüenciamento do caso X Número de casos concorrentes. 1 a 20 casos em ambiente centralizado. Máquina araguaia

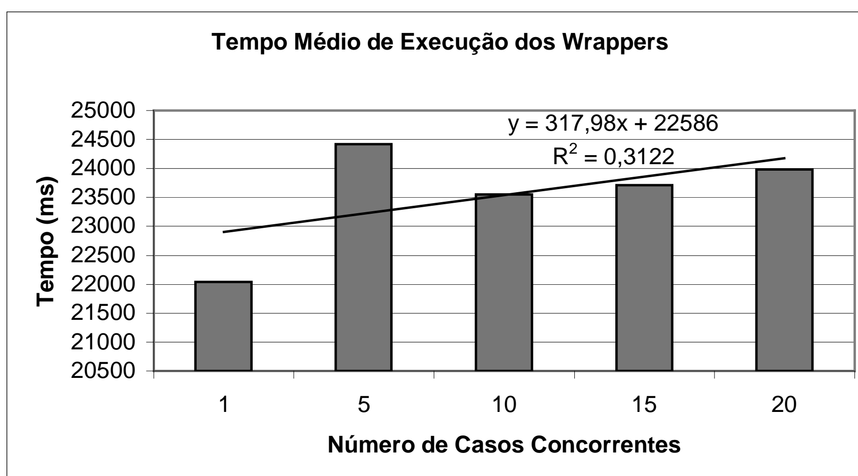

Gráfico 5: Tempo médio de execução do wrapper X Número de casos concorrentes. 1 a 20 casos em ambiente centralizado.

Análise

O Gráfico 3 apresenta o crescimento com tendência linear do tempo médio de execução dos casos conforme o número de instâncias concorrentes é aumentado (R^2 = 0,9639, muito próximo a 1). O Gráfico 4 apresenta a mesma tendência de crescimento linear (R^2 = 0,9768) para o tempo médio de seqüenciamento das atividades.

No Gráfico 5, o teste onde 5 casos concorrentes formam executados, apresentou um valor maior que seus vizinhos. Este ponto incomum é atribuído à presença de ruídos externos no momento em que o teste foi realizado para esta instância.

Analisando-se a primeira medida do Gráfico 5 observa-se que o tempo médio de execução dos *wrappers* durante a execução de um único caso (primeira coluna) é de aproximadamente 22 segundos. Neste teste há apenas 1 caso em execução, e portanto um único *ActivityManager* por vez na memória. Desta forma, vê-se que o tempo médio necessário para criar um objeto *wrapper,* criar e destruir um *command shell,* enviando uma notificação ao coordenador de casos localmente é de aproximadamente 2 segundos. Conforme o número de casos concorrentes aumenta, esta diferença sobe para 4 segundos (20 casos concorrentes), refletindo o aumento do uso de recursos da máquina araguaia.

8.6.2.2 Testes Distribuídos

O mesmo conjunto de casos de testes, realizados anteriormente, de forma centralizada no nó araguaia, é agora executado de maneira distribuída nos nós araguaia e iguacu. Atividades consecutivas executam em nós diferentes. Os coordenadores de processo, de caso e de papel executam na araguaia. Em cada nó é criado um *TaskList*.

Teste: Estudo da influência do número de casos concorrentes, cujas atividades não realizam processamento, nos tempos das atividades dos casos. Execução distribuída nos nós: *araguaia.ic.unicamp.br* e *iguacu.ic.unicamp.br.*

Descrição: Várias instâncias de um mesmo processo simples, contendo uma seqüência de 20 atividades iguais, são executadas de forma concorrente nas máquinas araguaia e iguacu. O número de casos concorrentes é incrementado de 5 a cada iteração do teste. O processo é iniciado, invocando-se WStarter, a partir da máquina iguacu.

Tabela de Resultados

Os dados de execução da bateria de testes distribuídos são relacionados na Tabela 21.

# Casos Paralelos	TMC (seg.)	TMC (ms)	TTC (ms)	TTEW (ms)	TMEW (ms)	TTS (ms)	TMS (ms)
1	673,67	673674,00	673674,00	440735,00	22036,00	132629,00	6980,00
5	717,70	717699,20	3588496,00	2239645,00	22396,00	791664,00	8332,80
10	856,10	856100,50	8561005,00	4683440,00	23416,70	2412284,00	12695,60
15	1103,61	1103612,33	16554185,00	7150401,00	23834,20	5591900,00	19620,27
20	1352,17	1352168,00	27043360,00	9595248,00	23987,60	9925649,00	26119,65

Tabela 21: Dados de execução dos testes envolvendo 1 a 20 casos concorrentes em ambiente distribuído. Máquinas: araguaia.ic.unicamp.br e iguacu.ic.unicamp.br. Sem processamento

Gráficos

Os dados da Tabela 21 são apresentados nos gráficos que se seguem.

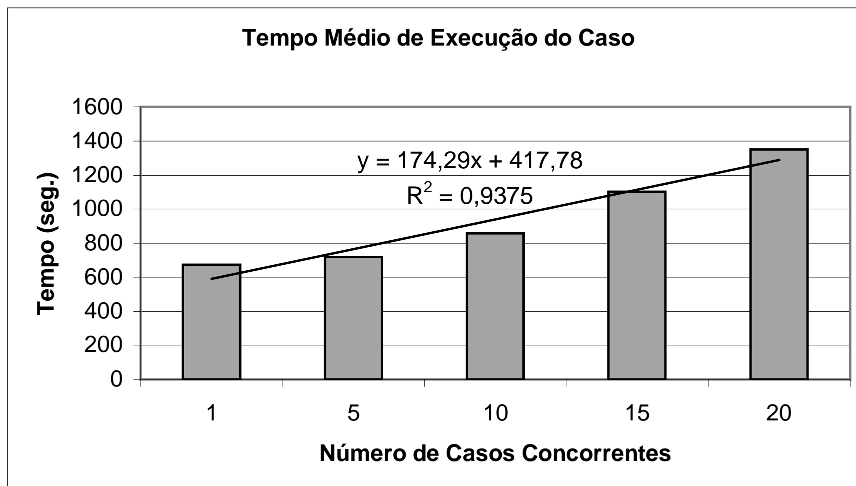

Gráfico 6: Tempo médio de execução dos casos X Número de casos concorrentes. Execução de 1 a 20 casos concorrentes em ambiente distribuído: nós araguaia e iguacu. Sem Processamento

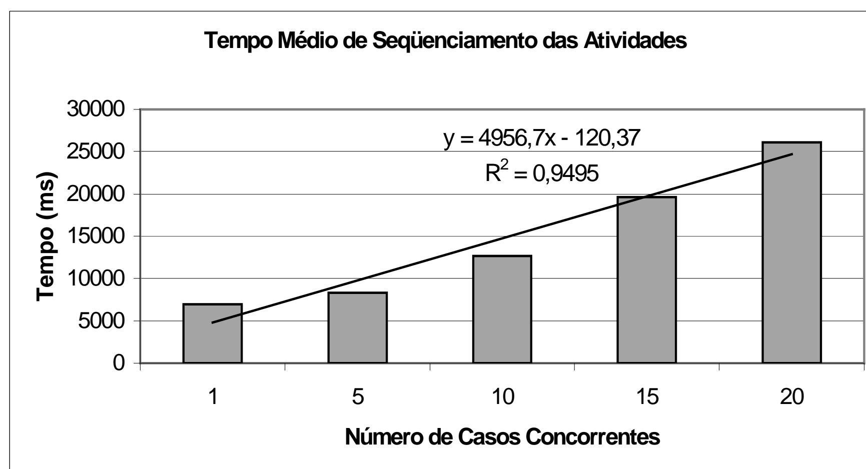

Gráfico 7: Tempo médio de seqüenciamento dos casos X Número de casos concorrentes. Execução de 1 a 20 casos concorrentes em ambiente distribuído: nós araguaia e iguacu. Sem processamento

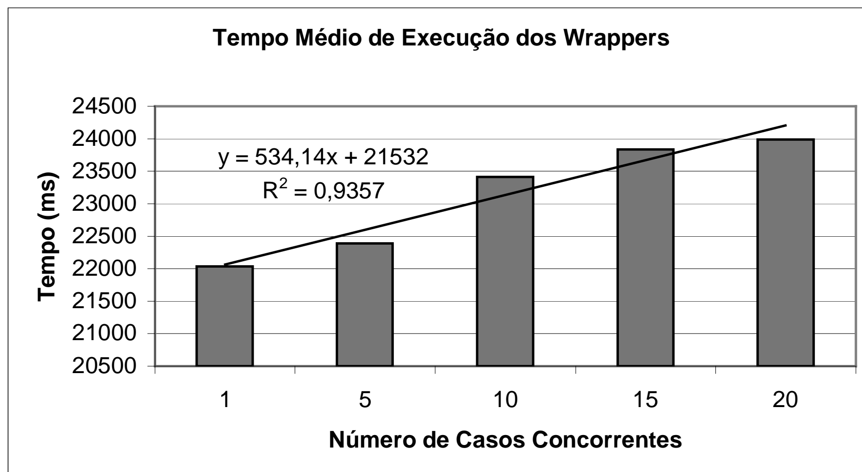

Gráfico 8: Tempo médio de execução dos wrappers X Número de casos concorrentes. Execução de 1 a 20 casos concorrentes em ambiente distribuído: nós araguaia e iguacu. Sem processamento

Análise

Da mesma forma que o caso centralizado, o Gráfico 6 apresenta um crescimento muito próximo ao linear (R^2 = 0,9375) do tempo médio de execução dos casos concorrentes conforme o número destes casos aumenta. O Gráfico 7 reflete esta mesma tendência linear (R^2 = 0,9495) no tempo de seqüenciamento das atividades. O Gráfico 8 apresenta uma tendência semelhante. Neste último gráfico, observa-se um ruído menor, se comparado com o Gráfico 5, do ambiente centralizado.

8.6.2.3 Comparação de Teses Distribuído e Centralizado

Os dados de execução coletados nos testes centralizado e distribuído, descritos anteriormente, são agora dispostos, lado a lado, de maneira a serem comparados.

Gráficos

Os gráficos a seguir apresentam os dados da Tabela 20 e da Tabela 21 lado a lado.

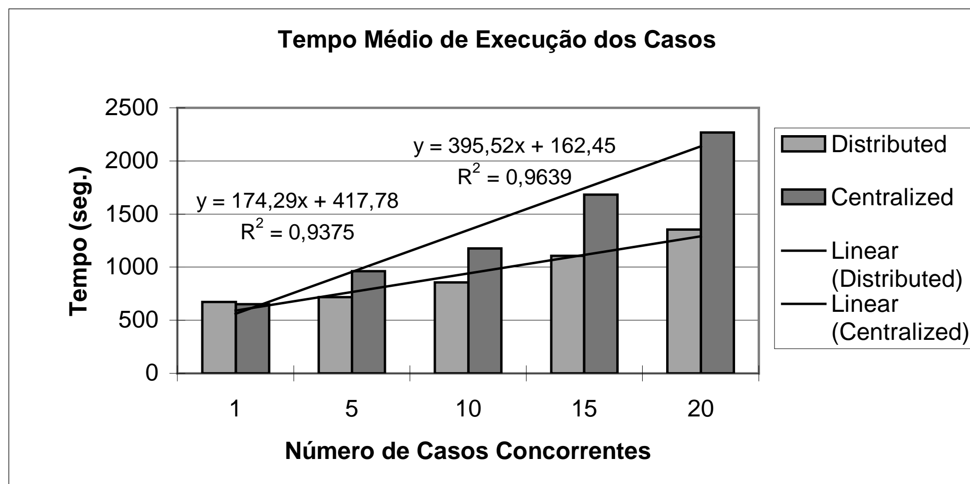

Gráfico 9: Tempo médio de execução dos casos X Número de casos concorrentes. Comparação de execuções de 1 a 20 casos concorrentes em ambientes distribuído e centralizado.

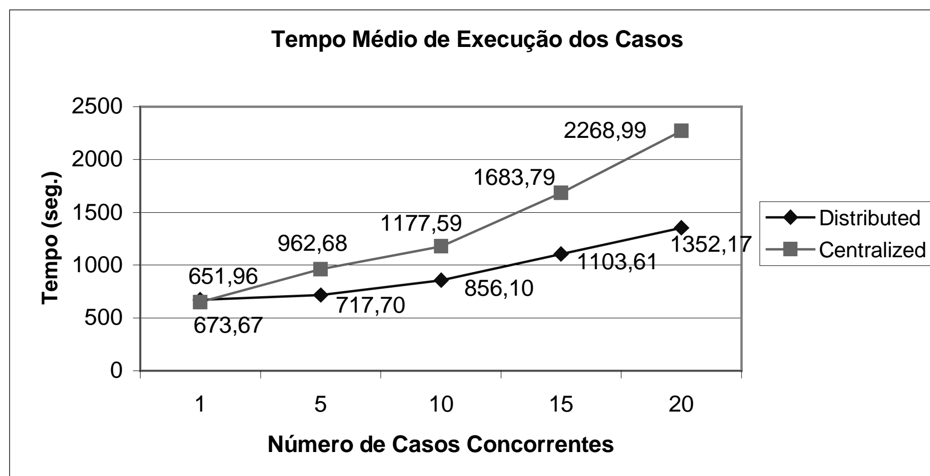

Gráfico 10: Tempo médio de execução dos casos X Número de casos concorrentes. Comparação de execuções de 1 a 20 casos concorrentes em ambientes distribuído e centralizado

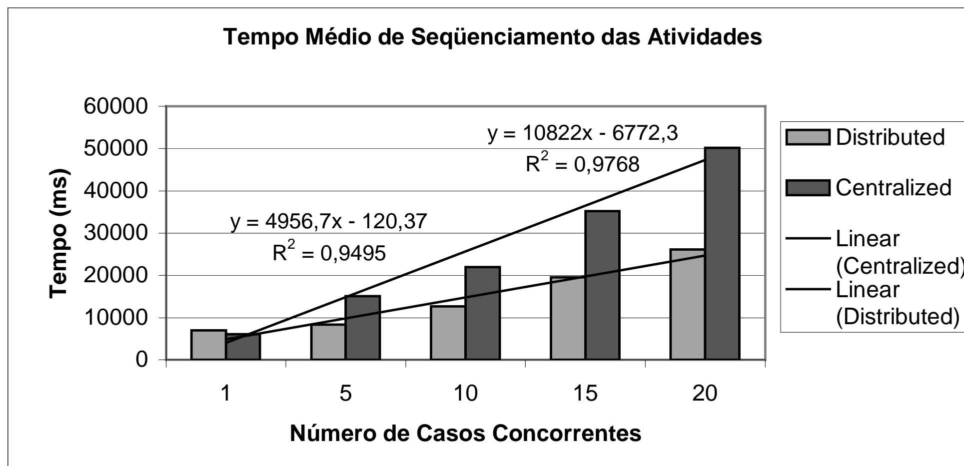

Gráfico 11: Tempo médio de seqüenciamento do caso X Número de casos concorrentes. Comparação de execuções de 1 a 20 casos concorrentes em ambientes distribuído e centralizado.

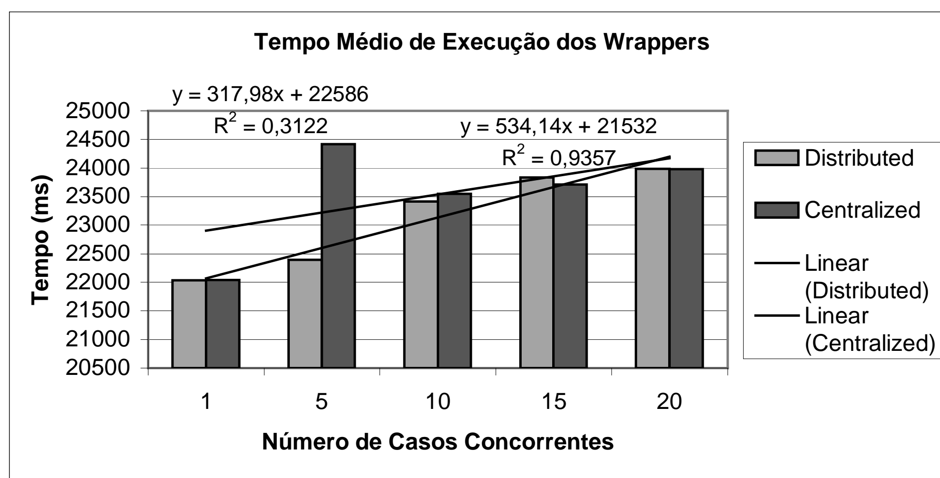

Gráfico 12: Tempo médio de execução dos wrappers X Número de casos concorrentes. Comparação de execuções de 1 a 20 casos concorrentes em ambientes distribuído e centralizado.

Análise

O Gráfico 9 e o Gráfico 10 permitem comparar os tempos médios dos casos executados de forma concorrente em ambientes centralizado e distribuído. Os tempos médios para a configuração distribuída são menores, superando em desempenho, os valores médios da configuração centralizada. Isto ocorre logo a partir das primeiras instâncias (em um ponto entre 1 e 5 casos concorrentes). Além do mais, a curva de tendência dos testes centralizados possui um coeficiente angular maior que o caso distribuído (174,59, distribuído, versos 395,52, centralizado), apontando uma maior taxa de crescimento com o aumento do número de casos concorrentes. Percebe-se então que a distribuição suaviza o impacto do aumento do número de casos concorrentes.

Os tempos médios máximo e mínimo de execução dos *wrappers*, dos testes distribuídos, foram muito semelhantes aos dos testes centralizados (Gráfico 12). Desta forma, os efeitos da descentralização foram mais sentidos nos tempos médios de seqüenciamento, que foram reduzidos à metade (vide coluna com 20 casos concorrentes), passando de 50 segundos, nos testes centralizados, para 25 segundos, nos testes distribuídos. Esta variação é contudo, relativamente pequena, cerca de 10%, ou 2 segundos entre os testes com 1 caso concorrente e com 20 casos concorrentes.

No Gráfico 11, o tempo de seqüenciamento das atividades acompanha a tendência do tempo médio de execução dos casos.

O aumento do número de casos concorrentes é mais sentido no tempo de seqüenciamento (Gráfico 11). Quando são comparados a primeira e a última bateria de testes, observa-se que o tempo médio de seqüenciamento é cerca de 4 vezes maior nos testes distribuídos (de 6,9s a 26,1s). e 8 vezes maior nos testes centralizados (de 6,0s a 50,1s). Na última bateria de testes, a execução centralizada foi cerca de duas vezes mais lenta que a distribuída.

8.6.3 Análise da Variação do Número de Casos Concorrentes - Com Processamento

O mesmo conjunto de testes, executados anteriormente sem realização de processamento, são agora realizados utilizando aplicações invocadas que realizam processamento. As mesmas configurações são empregadas. Um conjunto de testes adicional, envolvendo 4 máquinas é também realizado.

Nos testes distribuídos realizados em 4 máquinas, diferentemente dos outros testes, a escolha do próximo nó ocorre de maneira pseudo-aleatória. Os nós não são previamente especificados no plano. Desta forma, atividades são criadas nos nós preferenciais dos atores que, neste conjunto de testes, é o nó onde o *TaskList* do ator está executando. Todos os atores desempenham o mesmo papel, sendo registrados no mesmo coordenador de papel. Isto faz com que um dentre eles seja escolhido durante o processo de seqüenciamento, através de um sorteio realizado durante o seqüenciamento. A semente empregada é o relógio do sistema. Esta função fornece um espalhamento uniforme e diferente para cada iteração.

Deste ponto em diante, não serão mais apresentados, individualmente, os dados e os gráficos e dados coletados para os testes centralizados e distribuídos. Apenas a comparações gráficas dos resultados nestes dois cenários serão feitas.

8.6.3.1 Comparação dos Teses Distribuídos e Centralizados

Os dados de execução coletados nos testes centralizado e distribuído são dispostos, lado a lado, de maneira a serem comparados.

Gráficos

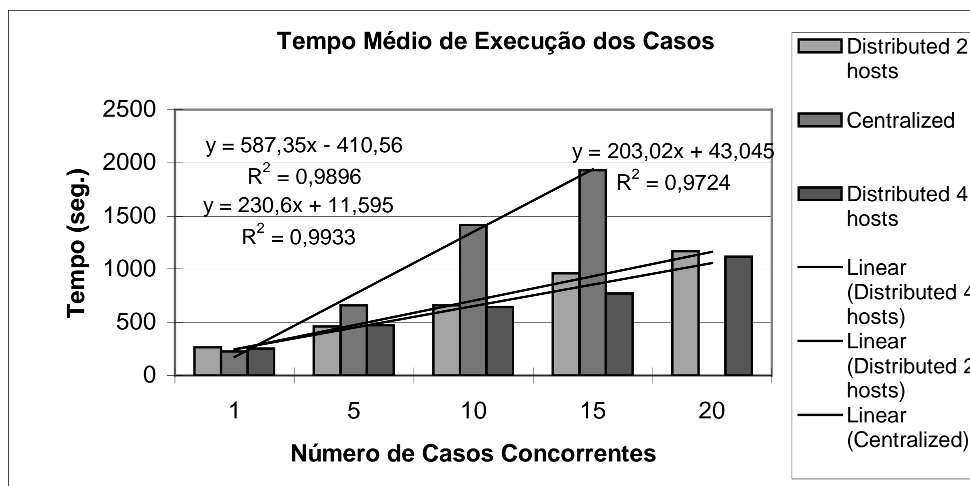

Gráfico 13: Tempo médio de execução dos casos X Número de casos concorrentes. Comparação das execuções de 1 a 20 casos concorrentes, em 2 ambientes distribuídos e 1 centralizado, executando atividades de ordenação de 1000 números.

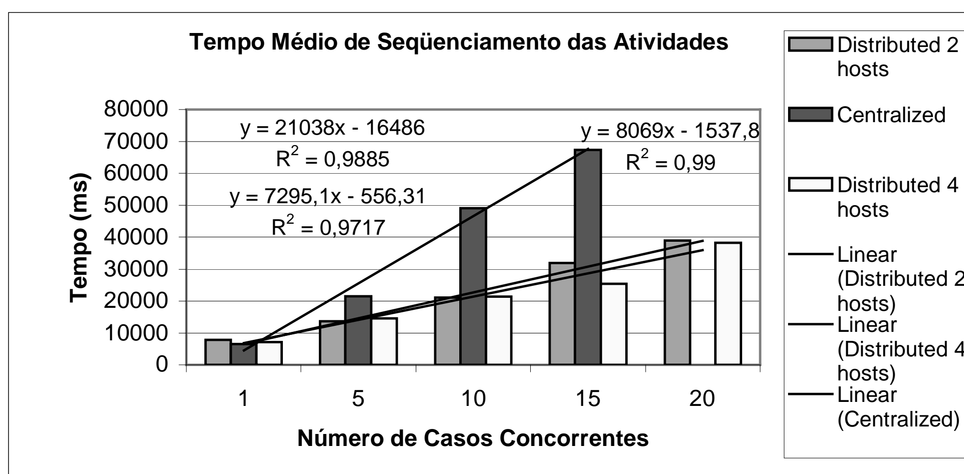

Gráfico 14: Tempo médio de seqüenciamento dos casos X Número de casos concorrentes. Comparação das execuções de 1 a 20 casos concorrentes, em 2 ambientes distribuídos e 1 centralizado, executando atividades de ordenação de 1000 números.

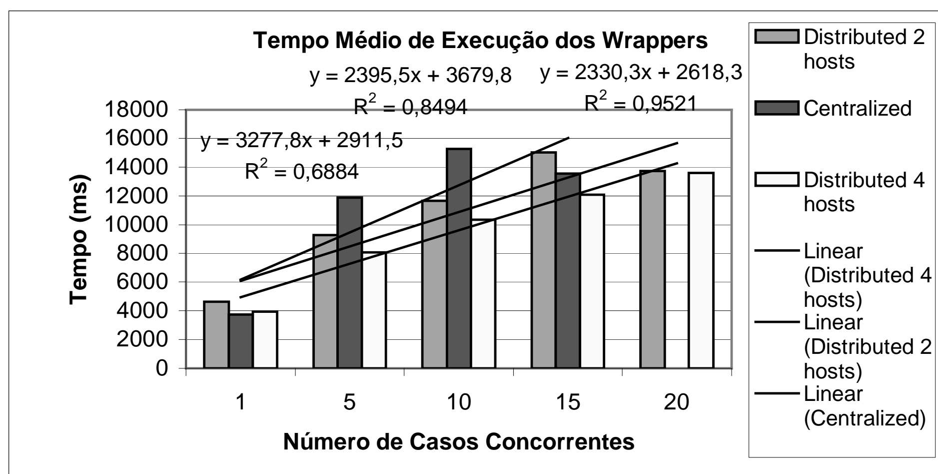

Gráfico 15: Tempo médio de execução dos wrappers X Número de casos concorrentes. Comparação das execuções de 1 a 20 casos concorrentes, em 2 ambientes distribuídos e 1 centralizado, executando atividades de ordenação de 1000 números.

Análise

Nos testes centralizados, instâncias com 20 casos concorrentes não puderam ser completadas com êxito por sobrecarregarem demasiadamente o ambiente centralizado dos testes. Desta forma, não foram coletadas medidas para esta instância.

O comportamento do sistema na presença de processamento é aproximadamente linear, dado os valores de R^2 observados no Gráfico 13 e no Gráfico 14, muito próximos a 1. As execuções distribuídas superaram a centralizada, em desempenho, a partir das primeiras instâncias (entre 2 e 5 casos concorrentes), possuindo ainda taxas de crescimento (coeficientes angulares das retas) muito menores que a apresentada nos testes centralizados.

A diferença entre os tempos médios de execução dos casos, mostrados no Gráfico 13, para os testes distribuídos, não é muito expressiva para o tamanho das instâncias testadas. Para maiores instâncias, contudo (a partir de 15 casos concorrentes), esta diferença tende a tornar-se maior, o que é observado pelo pequeno afastamento das curvas de tendência destes dois testes distribuídos.

Se forem comparados os tempos de seqüenciamento em testes com e sem processamento em ambiente centralizado (vide Gráfico 14 e Gráfico 11), observa-se que a relação dos tempos de seqüenciamento entre o primeiro e o último teste de cada bateria foi ampliado de 8 vezes (de 6s para 50s), no caso sem processamento, para 11 vezes (6s para 67s), no caso com processamento.

Como o Gráfico 14 mostra, o tempo de seqüenciamento para o ambiente de testes distribuído envolvendo 2 máquinas foi um pouco menor que o envolvendo 4 máquinas, nos casos onde haviam 5 e 10 casos concorrentes. Esta diferença é atribuída a ruídos externos, que afetaram

uma das 4 máquinas utilizadas. Já nos testes com 15 e 20 casos concorrentes, a execução em 4 nós superou a execução em 2 nós. Esta diferença tende a ser maior conforme há o aumento do número de casos concorrentes. Estes limites não foram explorados nestes testes, cujo limitante foi estabelecido tomando como base os testes centralizados.

No Gráfico 15, as diferenças nos tempo de execução dos *wrappers,* entre instâncias com 10 e com 15 casos concorrentes, estão invertidas para o caso centralizado. Isso é atribuído a interferências de ruídos externos aos sistema e à escolha aleatória dos números a serem ordenados.

O tempo médio de execução dos *wrappers*, mostrado no Gráfico 15, é, em sua maioria, menor para o ambiente envolvendo 4 máquinas pois a carga de processamento é melhor, estando distribuída entre os 3 nós que executam atividades.

8.6.4 Análise da Variação do Volume de Dados Trocado

Vários testes são realizados, aumentando-se gradualmente o volume de dados trocado entre as atividades para cada iteração. O plano utilizado nestes testes é composto por uma seqüência única formada por 20 atividades iguais. Não há execução de casos concorrentes. *Wrappers* executam o comando "*sleep 10*", resultando em uma espera de 10 segundos durante cada atividade. Os outros parâmetros do sistema são iguais aos dos testes anteriores.

O limite do volume de dados, estabelecido como sendo de aproximadamente 14MB nos testes, foi determinado de maneira arbitrária, de maneira a não forçar os servidores Java a seu limite de memória. Nos testes, o tamanho do *heap* da máquina virtual Java foi estendido de 16MB para 64MB. Não foram testados, portanto, os limites da arquitetura para o volume de dados que pode ser trocado entre *ActivityManagers* já que este fator depende das configurações das máquinas virtuais Java utilizadas.

Os testes foram realizados um após o outro, de maneira a, pelo menos, minimizar a variação das influências de outras aplicações. Desta forma, um teste pode ser realizado do começo ao fim, sofrendo uma carga mais ou menos estável de ruídos.

8.6.4.1 Comparação de Testes Distribuídos com Centralizados

Os dados de execução coletados nos testes centralizado e distribuído, são dispostos, lado a lado nos gráficos a seguir, de maneira a serem comparados.

Gráficos

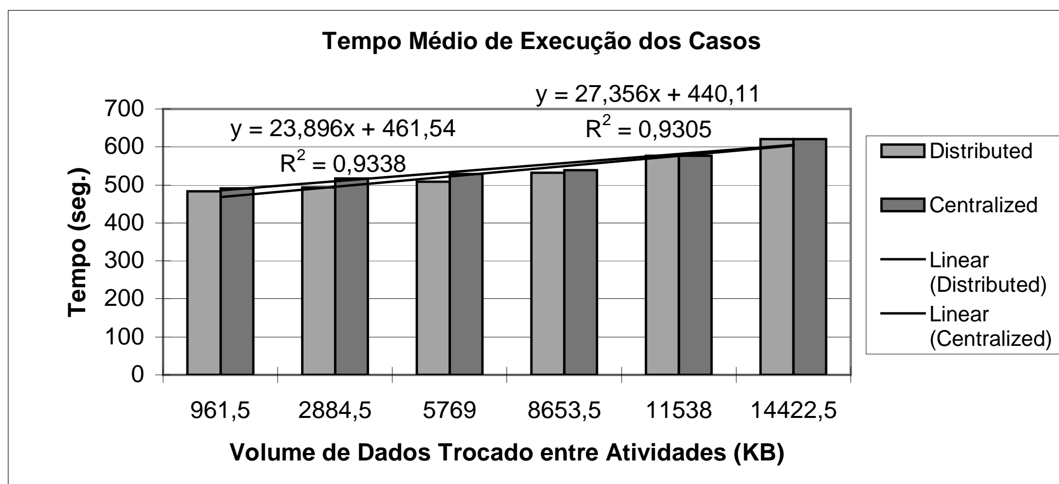

Gráfico 16: Tempo médio de execução dos casos X Volume de dados trocado entre as atividades do caso. Execuções sucessivas, com incremento de 3 unidades de dados a partir da segunda iteração. Comparação dos cenários distribuído e centralizado.

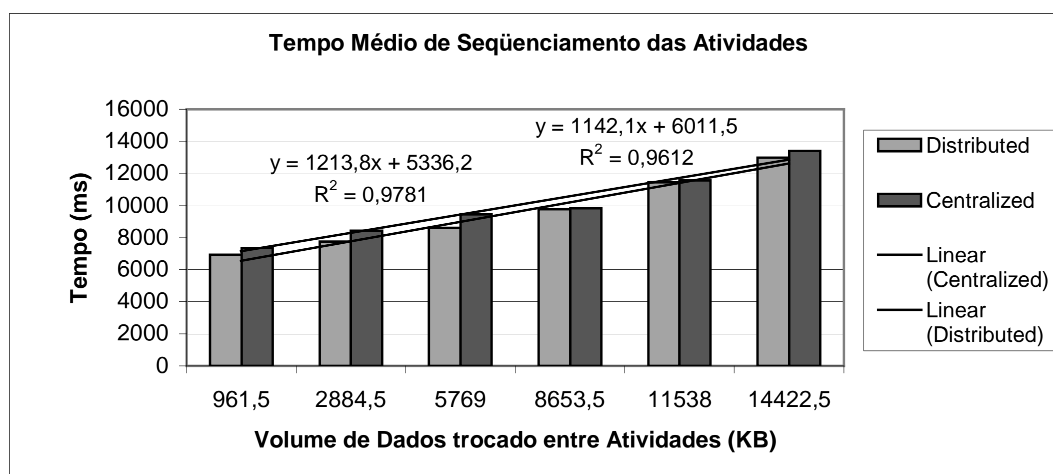

Gráfico 17: Tempo médio de seqüenciamento dos casos X Volume de dados trocado entre as atividades do caso. Execuções sucessivas, com incremento de 3 unidades de dados a partir da segunda iteração. Comparação dos cenários distribuído e centralizado.

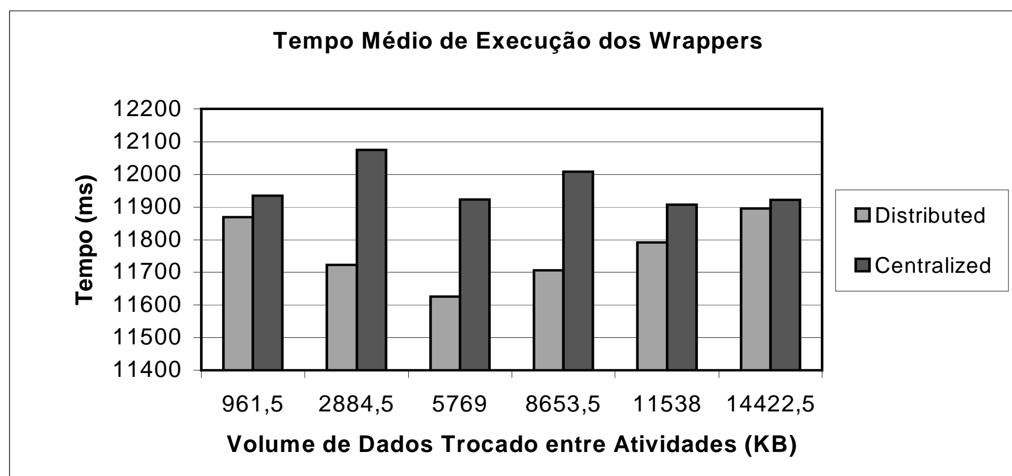

Gráfico 18: Tempo médio de execução dos wrappers X Volume de dados trocado entre as atividades do caso. Execuções sucessivas, com incremento de 3 unidades de dados a partir da segunda iteração. Comparação dos cenários distribuído e centralizado.

Análise

Nos testes centralizados, como havia apenas uma única seqüência executando de maneira centralizada, a carga imposta ao sistema pela arquitetura era muito pequena. As variações dos tempos de execução dos *wrappers* observados no Gráfico 18 são decorrentes de ruídos externos. Todavia, sua faixa de variação média nas baterias de testes realizadas possui amplitude de menos de 1 segundo, (11,6s a 11,9s nos testes distribuídos, e de 12,8s 1 11,9s nos testes centralizados) como pode ser observado no Gráfico 18.

Nos testes distribuído e centralizado, a influência da variação do volume de dados mostrou-se pequena. Observou-se um crescimento suave do tempo de execução médio do caso (Gráfico 16) com relação ao aumento do volume de dados trocado. Ao ser aumentado o volume de dados transferido em 15 vezes, o tempo de execução aumentou em aproximadamente 20% (de 500s a 600s). Como visto anteriormente na sessão 8.3.1, o tempo de transferência de dados possui pouca influência relativa no tempo total de execução de cada atividade.

8.7 Comentários e Conclusões

A arquitetura apresenta um comportamento linear quanto ao atraso de execução dos casos, quando o número de casos concorrentes, ou o volume de dados trocado, ou a carga de processamento das atividades são aumentados. O aumento do tempo médio de execução dos casos (o coeficiente angular das retas de aproximação) reduz-se conforme são introduzidos novos nós no sistema.

Nos testes realizados, não houve atrasos significativos associados às notificações (assíncronas) enviadas por atividades aos coordenadores de casos. O mesmo pode ser dito dos coordenadores de processos.

As execuções dos testes em configurações centralizadas foram logo superadas em desempenho, a partir das primeiras instâncias concorrentes, por configurações distribuídas. Desta forma, o uso de objetos CORBA escritos em Java, por executarem em máquinas virtuais distintas, não apresenta um bom desempenho em um ambiente centralizado onde o número de casos concorrentes é grande. O grande número de máquinas virtuais Java criadas nestes testes sobrecarregam facilmente um sistema centralizado. Em configurações distribuídas, contudo, onde o número de servidores da arquitetura WONDER e, consequentemente de máquinas virtuais Java, executando em um único nó, é menor, seu desempenho é aceitável, promovendo uma menor carga no sistema. Estes aspectos são evidenciados pelo melhor desempenho de configurações distribuídas frente as centralizadas.

Desta forma, o maior atraso associado ao agente móvel da arquitetura é o de criação destes servidores, um processo que consome memória e CPU, influenciando ainda no desempenho das outras atividades em execução.

Esta sobrecarga em configurações centralizadas é explicada, em parte, pela forma como a arquitetura foi implementada. O *framework* CORBA não diferencia entre operações invocadas localmente ou remotamente. Desta forma, independente da localização dos objetos, a troca de dados entre os estes servidores ocorre através de mensagens que trafegam através da pilha TCP/IP, sobre a qual está é implementado o IIOP. A arquitetura WONDER não implementa otimizações de transferência de dados em configurações centralizadas: Ao invés de transferir dados via invocações de operações, estes poderiam ser compartilhados localmente, quando as atividades consecutivas executassem em um mesmo nó. Outra otimização que não ocorre é a execução de dois ou mais servidores (*ActivityManager*) em um mesmo processo do sistema operacional, compartilhando código. Estes dois motivos foram cruciais no baixo desempenho das configurações centralizadas.

Instâncias centralizadas, executando um número maior de casos concorrentes não puderam ser executadas devido a limitações do OrbixWeb. Isto dificultou a exploração dos limites de desempenho das configurações empregadas nos testes centralizados, fazendo com que os testes não fossem muito conclusivos quanto ao comportamento do sistema, do ponto de vista de atrasos, nestes ambientes. Contudo, comparações entre configurações centralizadas e distribuídas puderam ser realizadas.

A arquitetura WONDER não foi projetada para ser executada de maneira totalmente centralizada. A partir de 5 casos concorrentes, nos testes realizados, a arquitetura apresenta maior desempenho em ambientes distribuídos, possuindo um comportamento linear para o número de instâncias empregado. A escalabilidade é, desta forma, proporcional ao grau de distribuição empregado, podendo ser aumentada adicionando-se novas máquinas ao ambiente distribuído, conforme a demanda por distribuição ou processamento aumente.

Se fossem implementadas otimizações que permitissem a comunicação entre servidores locais de forma diferenciada, utilizando memória compartilhada por exemplo, os testes centralizados certamente teriam um desempenho melhor que o distribuído (talvez para instâncias com algumas dezenas de casos paralelos). Entretanto, para instâncias maiores, seu desempenho seria superado pela execução distribuída pois outros fatores como o uso de memória ou consumo de CPU degradariam o desempenho do sistema centralizado, visto que um dos maiores custos da arquitetura é o de criação de objetos. Estes testes centralizados otimizados, contudo, não foram realizados, e os valores exatos destes tempos não são conhecidos.

Apesar de terem sido feitos testes com um protótipo simplificado da arquitetura, seu comportamento não sofreria alterações significativas se os outros componentes estivessem implementados na íntegra. As notificações enviadas aos coordenadores são assíncronas, o processamento destes objetos não introduz atrasos à migração normal do caso. Servidores de Backup seriam ativados apenas em momentos de baixa utilização do sistema.

O único componente que poderia introduzir atrasos à migração do agente seria o Coordenador de Papéis. Consultas mais elaboradas, baseadas em dados de execuções anteriores do workflow, realizadas ao coordenador de papéis poderiam atrasar em alguns segundos a fase de negociação das atividades. Este atraso, contudo, é dependente da consulta e do banco de dados do Servidor de Históricos e está relacionada como estes dados estão armazenados e organizados no banco de dados. Estas questões estão fora do escopo deste trabalho.

Capítulo 9

Trabalhos Relacionados

Neste capítulo são descritos vários trabalhos e arquiteturas que provêm soluções para a execução de workflows de larga escala. São descritos os principais aspectos destas arquiteturas, destacando as soluções empregadas para os requisitos de tolerância a falhas e escalabilidade. Algumas arquiteturas são comparadas com as soluções propostas no presente trabalho.

9.1 IBM Flowmark

O Flowmark é um SGWF comercial da IBM, baseado no modelo de referência da WfMC. Apesar de ser um SGWF centralizado, este sistema provê a infra-estrutura básica sobre a qual são construídos os outros dois projetos da IBM, o FMQM (*Flowmark on Message Queue Manager*) e o MQSeries (*Message Queues Series*) Workflow, que buscam formas de estender este modelo centralizado para atender aos requisitos de workflows de larga escala.

O Flowmark é um SGWF transacional que utiliza um banco de dados orientado a objetos centralizado (o ObjectStore da ODI). Os elementos do workflow como atividades, processos e listas de tarefas (*TaskLists*) não são processos autônomos independentes, mas sim entidades passivas, armazenadas no bancos de dados. Estes elementos assim como todos os dados de execução e definição dos processos são armazenados em um banco de dados central.

No Flowmark, atividades podem ser de dois tipos: atividades de programa ou atividades de processo. Neste último caso, uma atividade é um sub-processo (descrita por um sub-workflow). A interface com o usuário final é implementada através de listas de tarefas. A cada atividade descrita na definição de processos está associado um papel. O modelo define containers(de entrada e saída) de dados e de controle, associados a cada atividade. Containers de entrada de uma atividade são interligados com containers de saída da atividade seguinte através de entidades

conhecidas como conectores de dados e de controle. A cada container pode ser associado um conjunto de predicados lógicos conhecidos como condições de transição. Atividades são iniciadas quando suas condições de início são avaliadas como verdadeiras (AND), ou quando pelo menos uma delas é verdadeira (OR). Ao término das tarefas da atividade corrente, as condições de saída são avaliadas. A atividade termina quando estas condições são avaliadas para verdadeiro. Se esta condição é avaliada para falso, a atividade é novamente escalonada para execução.

9.2 Exotica/FMQM

Exotica/FMQM (*Flowmark on Message Queue Manager*) [AAAM95; MAGK95; MAGKR95] é um projeto desenvolvido por Mohan et. al no IBM Almaden Research Center. Este projeto é uma extensão do Flowmark que implementa um SGWF transacional, distribuído, baseado no conceito de filas de mensagens persistentes. Os principais problemas abordados neste projeto são a tolerância a falhas e a escalabilidade. O Exotica/FMQM utiliza filas persistentes como meio de comunicação entre as atividades, objetivando a substituição do banco de dados centralizado por este paradigma. O Exotica/FMQM é, desta forma, um SGWF distribuído onde um conjunto de nós autônomos (atividades) cooperam na execução de um processo comum.

O Exotica/FMQM provê suporte a clientes móveis através do conceito de *locked activity*. Um usuário pode selecionar atividades que deseja realizar em modo desconectado. O sistema marca estas atividades, com seus respectivos dados, como *locked*. Neste momento, a execução passa a ser responsabilidade dos usuários do sistema (abordagem otimista). Atividades executadas desta maneira residem na máquina dos usuários que iram executa-la, permitindo a desconexão temporária do nó em questão da rede [AGK+95].

O MQSeries é um *middleware* da IBM, orientado a mensagens, que fornece uma API, a MQI (*Message Queue Interface*). Este sistema foi utilizado como base para a implementação das filas persistentes do protótipo Exótica/FMQM.

9.2.1 Modelo Distribuído

Ao início de cada caso, atividades são criadas nos nós do sistema. Cada nó executa um processo chamado *node manager*. *Process threads* são processos que executam sob o controle de cada *node manager* e são responsáveis pela criação de *activity threads* locais, estes últimos representando atividades. *Activity threads* de vários processos concorrentes podem ser criadas em um mesmo nó. Estes *threads* monitoram seus conectores de entrada (para dados e para controle) de maneira a iniciarem sua execução quando a condição de entrada for satisfeita. *Activity threads* são responsáveis por executar as diversas tarefas de uma atividade e, ao seu final, enviar dados e informação de controle para a próxima atividade, através de seus conectores de saí-

da. Se o conjunto das pré-condições é avaliado para false, a atividade propaga esta condição, sendo em seguida encerrada. Este processo permite a eliminação de caminhos que não são utilizados (*dead paths elimination*).

9.2.2 Principais Características

A escalabilidade é implementada através da execução das atividades, de maneira distribuída, por nós do sistema. Esta distribuição favorece também o esquema de tolerância a falhas. A comunicação acontece através de filas nomeadas (*named queues*), de forma assíncrona: os participantes (que enviam e recebem mensagens dessa fila) não precisam estar ativos ao mesmo tempo no sistema. Cada site opera de forma independente um do outro, sendo a única interação entre esses nós conduzida através de mensagens persistentes, que informam sobre eventos ocorridos no sistema e transmitem dados entre atividades.

A interação entre as atividades e as filas que as conectam ocorrem de forma transacional: uma mensagem não pode ser lida até que sua operação de postagem (*Put*) tenha sido decidida (*commited*) e permanece armazenada na fila até que a operação de leitura (*Get*) decida. Essa abordagem aumenta a tolerância a falhas do sistema assim como sua escalabilidade. As filas permitem isolar as atividades. A falha de um nó pode ser compensada pela entrada de outro nó que, utilizando os dados contidos na fila de mensagens persistente, consegue levar adiante a atividade interrompida.

Recursos são geralmente alocados em tempo de definição do workflow, sendo compilados juntamente como o plano. Atividades são instanciadas em nós preestabelecidos assim que o caso é criado. A atribuição de atores a atividades, contudo, é realizada assim que uma atividade muda para o estado "pronta para execução". Somente um ator é escolhido para a realização da atividade. Não há negociação entre o sistema e um conjunto selecionado de atores. Dados e aplicações utilizadas durante a execução do workflow são especificadas, em tempo de definição, na definição do processo a ser executada.

Por ser uma extensão do Flowmark, a interface gráfica do sistema é provida pelo RTC (*Runtime Client*) deste SGWF.

9.2.3 Exotica/FMQM X WONDER

Semelhante ao Exotica/FMQM, o WONDER aborda o problema da escalabilidade criando atividades em nós do sistema distribuído. A semântica transacional na comunicação entre atividades também é empregada.

No WONDER, as atividades são criadas sob demanda, durante a execução do caso, desta forma, não há necessidade da manutenção filas persistentes de dados, como é feito no Exoti-

ca/FMQM. Se uma atividade ou link falhar durante a transferência de dados, o processo é repetido utilizando-se outro nó, onde é criada outra atividade.

No WONDER, não há necessidade de especificação de guardas de entrada e de saída nas atividades. As atividades são elementos ativos, autônomos que, ao interpretarem o plano localmente, sabem quando migrar para um outro nó.

9.3 IBM MQSeries Workflow

Em contraste com a proposta mais radical do Exotica/FMQM, que propõe a substituição do banco de dados central por filas de mensagens persistentes, os trabalhos [AKA+94; IBM99] adotam uma abordagem mais centralizada, estudando o uso de vários agrupamentos (*clusters*) de servidores executando o Flowmark. No MQSeries Workflow, para cada agrupamento é utilizada uma mesma base de dados, compartilhada entre vários servidores Flowmark.

Um cliente identifica-se (*log*) para o *Execution Server* de um agrupamento e não para um servidor Flowmark. Neste momento, o *Execution Server* seleciona um dos servidores do agrupamento para executar o workflow corrente. Este esquema utiliza a replicação de servidores de maneira a prover maior desempenho e disponibilidade. Instâncias de processos não podem ser executadas por mais de um servidor contudo, diferentes instâncias de um mesmo processo podem ser executadas em máquinas diferentes. Servidores podem ser ainda replicados por diferentes domínios administrativos. Esta abordagem permite a implementação de balanceamento de carga. A escalabilidade é alcançada através da adição de mais servidores e agrupamentos neste esquema de distribuição.

Em caso de falha de um servidor, o cliente é movido, de forma transparente, para outro servidor do agrupamento, que utiliza os dados do banco de dados compartilhado por este conjunto de servidores. Este procedimento é realizado pelo *Execution Server*.

De maneira a solucionar problemas relacionados à falha do banco de dados compartilhado de um agrupamento, procedimentos de backup são realizados de maneira cruzada entre SGBDs de agrupamentos distintos. Durante a execução de uma instância de processo, ao final de cada tarefa, dados de execução são trocados entre servidores, usando filas persistentes providas pelo *middleware* MQSeries. Estes dados são armazenados em áreas reservadas de backup do servidor. Desta forma, em um determinado instante, cada banco de dados associado a um agrupamento de servidores armazena uma cópia dos dados de outro agrupamento. Ao final do caso, estes dados de backup são eliminados por procedimentos de *garbage collection*.

A interação entre cliente (ator) e servidor é realizada através de mensagens trocadas via MQSeries ou através do *middleware* CORBA. A interface com o usuário final (atores do workflow) é provida pelo *MQ Workflow Client.* A execução de aplicações é realizada pelo processo *Program Execution Agent.* Além do servidor de execução, outros servidores são providos. São eles

o *Administration Server*, o *Cleanup Server* e o *Scheduling Server*, que são responsáveis, respectivamente, por realizar a recuperação de falhas, procedimento de finalização de casos e envio de notificações baseadas em temporizadores (eventos externos).

9.3.1 MQSeries Workflow X WONDER

Apesar de adotar uma abordagem centralizada, o MQSeries é aqui apresentado como um exemplo de solução para o problema de escalabilidade que não utiliza a abordagem totalmente descentralizada da arquitetura WONDER.

9.4 Mentor

O *Middleware for Enterprise-Wide Workflow Management* (MENTOR) [WWWD97; WWWK96] é um projeto da Universidade de Saarland na Alemanha. Este projeto objetiva o desenvolvimento de um ambiente para execução, monitoramento e controle de workflows, provendo escalabilidade e disponibilidade. MENTOR é um workflow transacional, baseado na arquitetura cliente-servidor onde servidores de workflow controlam a execução das atividades. O controle e os dados do processo são centralizados nos servidores do sistema contudo, a execução das aplicações é distribuída. A escalabilidade é conseguida através da distribuição de servidores por vários domínios organizacionais. Estes servidores são autônomos, possuindo seu próprio banco de dados. Um workflow pode ser dividido em sub-workflows que são controlados por vários destes servidores autônomos. A disponibilidade é alcançada através da cópia da base de dados destes servidores de domínio para servidores de backup.

Mentor permite a execução de planos descritos em linguagens diferentes, assim como o uso gerenciadores de tarefas (*TaskList Managers*) heterogêneos. Estes gerenciadores podem implementar várias políticas de alocação de atores. Esta interoperabilidade de definições de processo é conseguida através do mapeamento da diferentes definições para *state charts*, realizada em tempo de definição do workflow. Esta definição de processo é interpretada, em tempo de execução, por uma máquina workflow proprietária utilizada pelo Mentor.

O *Communication Manager* é responsável pelo compartilhamento de dados armazenados em servidores diferentes. Dados do workflow são passados, entre atividades consecutivas, como referências, sendo recuperados pelas atividades quando necessários. Esta abordagem evita tráfego desnecessário na rede e permite o compartilhamento desta informação por vários casos concorrentes em execução no SGWF. Problemas de segurança e gerenciamento de dados também são simplificados com esta abordagem. A comunicação entre servidores é realizada de maneira transacional, tendo sua consistência controlada por *TP-Monitors* (monitores de transação).

CORBA é empregado como um meio de integração entre aplicações heterogêneas. Para cada aplicação é especificado um *wrapper*, possuindo uma interface IDL, que funciona como uma ponte entre a aplicação invocada e o SGWF. Esta abordagem permite a troca de parâmetros e dados entre a máquina workflow e as aplicações.

A recuperação de falhas é auxiliada pelo *Log Manager* um processo executando em cada servidor que armazena informações de execução do servidor corrente em meio estável. Históricos de execução de cada instância de processo são armazenados em um servidor de histórico gerenciado pelo *Workflow History Manager*, um servidor que mantém informações de execuções passadas e correntes dos casos. Este servidor também armazena informações sobre tarefas desempenhadas por atores do sistema, fornecendo informações usadas na alocação de atores a atividades.

O *Worklist Manager* gerencia as listas de tarefas dos usuários do sistema permitindo implementar políticas de alocação de atores e papéis. Estas políticas de alocação podem usar informação armazenada no *History Manager* para realizar políticas de alocação de atores mais sofisticadas. O *Worklist Manager* permite ainda a aceitação ou rejeição de atividades pelos atores.

9.4.1 Mentor X WONDER

Os elementos *Worklist Manager* e *History Manager* da arquitetura Mentor possuem funções semelhantes ao *Worklist* e *History Server* da arquitetura WONDER, permitindo a implementação de políticas de alocação de atores de acordo com dados de execuções anteriores. O *Wrapper* dos dois modelos são também muito parecidos, embora a definição de uma interface para integração com aplicações invocadas não tenha sido implementada no protótipo WONDER.

9.5 INCA (INformation CArriers) Workflow

Rusinkiewicz et. al, da Universidade de Houston nos EEUU, desenvolveu o modelo de workflow baseado em INCAs [BMR96] (carregadores de informações). Este modelo foi desenvolvido para suportar a execução de processos dinâmicos de workflow, e especifica unidades de execução que são relativamente autônomas. Nesta arquitetura, a definição do processo (plano), juntamente com os dados necessários a sua execução, são encapsulados em um objeto distribuído chamado INCA. Em tempo de execução, o INCA é roteado entre várias entidades de processamento distribuído, de acordo com a definição de processo (conjunto de regras) do workflow corrente, que também é carregada pelo INCA. Nesta abordagem, a execução e o controle do workflow são descentralizados.

Segundo seus autores, o INCA Workflow destina-se a workflows que executam em ambientes autônomos e que evoluem dinamicamente, a exemplo de ambientes de escritório. Nestes ambi-

entes, os processos podem ser levemente alterados durante sua execução e o ambiente de execução é parcialmente automatizado (algumas tarefas são manuais) e fracamente conectado (estações móveis como *notebooks* podem existir). Estações de trabalho são autônomas e as atividades parcialmente automatizadas, em contraste com workflows transacionais convencionais. INCAs implementam o paradigma de formulários em escritórios, que trafegam de mesa em mesa conforme o trabalho vai sendo realizado.

Neste modelo, workflows são descritos usando uma linguagem de regras proprietária do tipo ECA (Evento condição Ação). Um INCA é um container de informações (ou seja, uma entidade passiva) que carrega dados da atividade a ser realizada, histórico de execução no formato de *logs*, e informações de controle (regras) relacionados à tarefa a ser executada em um nó (estação de processamento). Ao ser terminada uma atividade, processos executando no nó corrente determinam o próximo conjunto de nós que receberão o(s) INCA(s) com base nas regras (definição de processo) contidas este INCA. Estas regras assim como os dados transportados pelo INCA podem ser modificados (adicionados, excluídos ou alterados) durante este processo. INCAs armazenam também informações de requisitos de execução, como necessidade de atomicidade, da atividade corrente. Os *logs* carregados pelos INCAs são empregados durante o procedimento de recuperação de falhas, de maneira a garantir a atomicidade das atividades do workflow.

Estações de processamento (*processing stations* – PS) são responsáveis por receber INCAs, executar seus serviços requisitados, e passa-los a diante para os destinos corretos, usando os meios de comunicação adequados. Como meio de comunicação podem ser utilizados: e-mail, FTP, *pipes*, FAX ou mesmo *smart cards* (No protótipo implementado, foi utilizado e-mail). Internamente, estas estações são compostas por um INCA *shell*. Este processo compreende um interpretador de regras e um agente que realiza a recepção, execução e roteamento do INCA.

Estações de processamento podem ser completamente automatizadas, gerenciando um SGBD, ou podem ser estações de trabalho convencionais, operadas por atores humanos. Estas máquinas disponibilizam dados, regras locais e procedimentos. Ao chegarem a estas estações, os INCAs podem chamar estes procedimentos, modificando seus dados locais. Por exemplo, uma estação de processamento pode conter uma base de dados de passagens aéreas (dados locais), que são modificadas pelos procedimentos (APIs) providas por esta estação. Desta forma, estações provêm serviços (automáticos – feitos por procedimentos, ou manuais – feitos por atores humanos) aos INCAs, ao mesmo tempo que funcionam como roteadores destes objetos pela rede, com base nas regras descritas nos próprios INCAs ou mesmo localmente, na estação de processamento. Todos os procedimentos providos por uma estação obedecem a modelos transacionais locais. Estes modelos abrangem deste os tradicional modelo ACID, para bancos de dados, até modelos transacionais relaxados ou mesmo não transacionais, estes últimos usados na interação com atores humanos.

Splits e *joins* são implementados como chamadas aninhadas a (sub)INCAs. Em um procedimento de submissão de artigos para congressos, por exemplo, um INCA é replicado (*split*) e enviado a vários revisores. Ao final das revisões, estes INCAs retornam ao seu nó original (*join*). Neste momento, os dados e regras resultantes, possivelmente alterados, são então agru-

pados (*merged*) de acordo com uma política preestabelecida. O INCA original segue, então, seu caminho.

INCAs podem ser alterados cada vez que passam por um nó da rede. Esta capacidade de modificação dinâmica das regras, que regem a migração dos INCAs, e dos dados, carregados por estes containers, permite a utilização deste SGWF em workflows *ad-hoc*.

As falhas de nós e dos INCAs são tratadas com o uso de ações de compensação, especificadas nos próprios INCAs ou nas estações de processamento. Erros de enlace são detectados por mecanismos de *timeout*. Caso um enlace permaneça por muito tempo indisponível, rotas alternativas podem ser encontradas ou mesmo especificadas por regras carregadas pelos INCAs. Estas recuperações são auxiliadas por *logs* e por cópias dos INCAs mantidas em estações anteriores (*checkpointing*). A recuperação de falhas utiliza o algoritmo de sagas [GMS87] aplicado para aplicações envolvendo nós de um sistema distribuído. Este algoritmo assume a execução de procedimentos com semântica transacional pelas estações de processamento. As atividades de compensação são programadas usando a linguagem de eventos proprietária, o que permite adotar diversas políticas de tratamento de falhas.

A segurança dos dados que trafegam pela rede é provida por uso de criptografia e autenticação. Esta funcionalidade, contudo, não foi implementada no protótipo, que é escrito em C, para a plataforma Windows 3.1.

9.5.1 INCA X WONDER

A arquitetura WONDER estende o conceito dos INCAs utilizando o paradigma de agentes móveis, criando entidades ativas, autônomas, que interpretam o plano em questão. INCAs são entidades passivas. No WONDER, as estações que recebem os agentes são passivas, não provendo serviços, ao contrário das estações de processamento do modelo INCAs.

Ao invés de serem usadas ações de compensação na recuperação de falhas, o WONDER permite criar atividades de compensação, mantendo contudo uma semântica semelhante à adotada pelo modelo INCA. O conceito de *checkpointing/logs*, armazenando cópias do agente em nós anteriormente percorridos, também é utilizado no WONDER.

Requisitos como rastreamento, auditoria, alocação de atores não são abordados no modelo INCA.

9.6 METEOR$_2$

O METEOR$_2$ (*Managing End-To-End OpeRations*) [MPS+97; DKM+97; SKM+96; MSKW96], projeto desenvolvido por Sheth et. al na Universidade da Geórgia, tem com objetivo investigar aspectos tecnológicos relacionados ao suporte a SGWFs distribuídos. Cinco diferentes arquiteturas, desde a totalmente descentralizada até a completamente centralizada foram propostas. Três implementações diferentes existem, explorando a tecnologia WWW (WEBWork), CORBA (ORBWork) e o NEO da Sun (NEOWork). Nesta sessão, será descrita a proposta totalmente distribuída do ORBWork.

9.6.1 Modelo METEOR$_2$

No METEOR$_2$ Processos são especificados graficamente com o auxílio de uma aplicação gráfica chamada *designer component*. Workflows são compostos por tarefas que podem ser dos tipos: transacionais, não transacionais ou compostas. Tarefas são responsáveis por realizar sincronizações do tipo AND e OR, podendo ter sua execução dependente de eventos ou dados provenientes de tarefas anteriores (dependência de ativação) ou de acordo com regras de transição adotada pela atividade corrente (transição de entrada). Tarefas possuem transições de saída (predicados lógicos) que podem incluir dependências de dados, envolvendo atributos de entrada (vindos de outras tarefas), e/ou de saída (produzidos na tarefa corrente).

Dados podem ser estruturas complexas, sendo modelados através da linguagem OMT (*Object Modeling Technique*), o que permite a representação de classes, atributos, operações e relacionamentos entre conjuntos de informações. A modelagem de dados é realizada por uma ferramenta com interface gráfica chamada *data design component*.

Após a modelagem da estrutura de processo, que compreende a estrutura de dados, as tarefas e suas interdependências, esta estrutura é representada em uma linguagem intermediária chamada WIL (*Workflow Intermediate Language*).

Processos descritos em WIL são compilados, pelo gerador de código (*code generator component*) para um dos sistemas de execução do METEOR. Este sistema de execução que pode ser o WEBWork ou ORBWork. Neste processo, dados são encapsulados em objetos, de maneira a serem manipulados pelo sistema de tempo de execução (*runtime system*) destino. A estrutura do workflow é então gerada dispondo seus objetos componentes de maneira estática por nós do sistema.

O sistema de tempo de execução é formados por gerenciadores de tarefas (*Task Managers*), que controlam a execução de suas respectivas tarefas, pela interface com o usuário final (via Web,

usando HTML e CGI), pelo mecanismo de recuperação de falhas distribuído, pelo escalonador (*sheduler*), que é distribuído pelos gerenciadores de tarefas, além de vários componentes de monitorização.

No METEOR, falhas são tratadas nas próprias tarefas. Seu tratamento, específico para cada tarefa, é descrito através do uso de linguagens de definição de workflow e de tarefas do *sistemas (TSL – Task Specification Language e WFSL – Workflow Specification Language*). O METEOR utiliza *forward recovering* (desfazer semântico) valendo-se de *logs*, tarefas alternativas e de compensação. Sempre que possível, os erros são tratados de maneira automática pelo sistema. Falhas e conflitos mais complexos são feitos manualmente, por um administrador, usando o *Workflow Monitor*.

9.6.2 ORBWork

O ORBWork é a implementação do modelo $METEOR_2$ para CORBA. Neste sistema, todos os componentes do execução do $METEOR_2$ são mapeados para objetos servidores CORBA. Sempre que possível, são usados os serviços CORBA, como o de transações. Os objetos são gerados automaticamente por um compilador que converte a especificação intermediária, descrita na linguagem WIL, para objetos CORBA, representados por interfaces IDL, *stubs*, *skeletons*, e implementação. Tarefas são "envolvidas" por interfaces IDL (*wrappers*), o que permite que estas representem aplicações legadas.

Gerenciadores de Tarefas controlam a execução de tarefas. Estes objetos possuem componentes que realizam a ativação de atividades *(task activator*), a invocação de tarefas ou aplicações externas (*task invocation and observer*) e o tratamento e recuperação de erros. Gerenciadores de tarefas podem ser de três tipos, de acordo com a natureza das tarefas a serem monitoradas: transacionais, não transacionais e baseados em usuários. Os dois primeiros monitoram a execução de procedimentos e tarefas automáticas, enquanto que o último monitora a execução de tarefas manuais, feitas por usuários (atores).

Ao serem gerados, os gerenciadores de tarefas são programados com a lógica necessária para a execução do workflow. Estes gerenciadores são criados em nós predefinidos da rede, formando uma estrutura de controle estática. Desta forma, cada gerenciador de tarefas conhece, de ante mão, as próximas tarefas que deverão ser ativadas ao final de sua execução. Gerenciadores de tarefas são ativados por seus predecessores e, uma vez ativados, monitoram um conjunto de pré-condições que "guardam" a ativação da tarefa associada a este gerenciador. Quando estas condições são satisfeitas, a tarefa é acionada. Ao final da tarefa, o próximo gerenciador de tarefas é ativado com base nas pós-condições da tarefa corrente. A estrutura do workflow é, desta forma, previamente definida, tendo o controle da máquina workflow (*scheduler*) "dividida" por cada gerenciador de tarefas do processo corrente. O ORBWork não permite, com esta política de alocação de objetos, a execução de workflows *ad-hoc*, nem a mudança dinâmica do workflow.

Ao final de cada execução de uma tarefa, seu gerenciador realiza um *chechpoint*, salvando o estado de execução, juntamente com os dados gerados na atividade e um *log* de execução, no *Local Persistent Storage*, um banco de dados replicado, compartilhado por objetos do domínio onde a aplicação executa.

Não há suporte direto para alocação de atores a tarefas. A interface com o usuário é implementada através de um conjunto de listas de tarefas que podem monitorar determinadas tarefas. Uma lista de tarefas é associada a cada papel, servindo como uma fila de tarefas. Atores localizam e interagem com as listas de tarefas associadas a seus papeis. A tarefa é apresentada aos atores que irão desempenha-la na forma de uma página HTML, que apresenta links e dados necessários a sua execução, assim como elementos de controle necessários ao seu desempenho.

A escalabilidade é conseguida através da criação de gerenciadores de tarefas em nós dispersos do sistema, juntamente com a descentralização de controle de execução, que também é realizada por estes gerenciadores. A localização destes servidores CORBA é predefinida, em tempo de compilação. Tarefas, entretanto, podem ser criadas em nós diferentes dos seus gerenciadores e sua localização pode ser especificada em tempo de execução. Dados manipulados pelas tarefas do workflow são representados como objetos CORBA e são passados entre controladores de tarefas como referências, ao invés de serem diretamente transportados.

Erros são classificados em três categorias: erros de tarefas, relacionados a execução de aplicações; erros de gerenciadores de tarefas, relacionados a ativação/desativação e controle de execução de tarefas; e erros do workflow, incluindo problemas de comunicação e interdependências entre os gerenciadores de tarefas. Cada um destes erros pode ser subdividido em duas categorias: erros lógicos ou erros de sistema. O ORBWork provê mecanismos transacionais para a resolução destes erros usando regras de compensações especificadas em tempo de projeto, de acordo com a semântica de cada tarefa.

A detecção de erros é realizada por objetos servidores chamados *recovery managers*. Cada nó da rede, onde tarefas são executadas, executa um *Local Recovery Manager* (LRM) que, através de um *wathdog*, periodicamente verifica cada componente (objeto) local, registrado em sua *watch-list.* Ao detectar uma falha, esta é corrigida localmente com o uso de *logs* e de *checkpoints* dos gerenciadores de tarefas locais. Globalmente, erros são detectados pelo *Global Recovery Manager* (GRM), que executa em nós mais confiáveis. Este servidor monitora os LRMs, detectando a falha destes servidores, também através de *watchdogs,* que periodicamente verificam os servidores contidos em sua *watch-list*. GRMs podem ser replicados para distribuir a carga de monitoramento ou por questões de tolerância a falhas.

9.6.3 METEOR$_2$ X WONDER

A realização de *checkpoints*, feita pelo METEOR corresponde ao estado de execução deixado por atividades da arquitetura WONDER pelos nós onde estas executaram. Semelhante ao GRM do METEOR, o Coordenador de caso do WONDER é responsável pela detecção dos erros en-

volvendo atividades de um caso. A implementação desta detecção, não foi explorada no projeto WONDER mas, o uso de *watchdogs* do $METEOR_2$ pode ser uma boa escolha.

Ao contrário do modelo INCA, que transporta os dados do workflow entre atividades consecutivas, o ORBWork transporta referências a dados, permitindo a obtenção desta informação somente quando necessário. O WONDER utiliza o melhor dos dois mundos: transfere links juntamente com dados entre atividades consecutivas. Dados necessários a execução da atividade seguinte são coletados por sua atividade anterior. Diferentemente do METEOR, os dados do WONDER ficam armazenados pelos nós onde o caso executou.

9.7 Proposta da Nortel para o OMG

Em resposta a um RFP do OMG para a Facilidade de Gerenciamento de Workflow para a arquitetura OMA, Weather et. al [WSR98], da Universidade de Newcastle upon Tyne (*Arjuna group*), propuseram um SGWF Transacional, tolerante a falhas, baseado em CORBA. A arquitetura proposta busca satisfazer os requisitos de escalabilidade, composição flexível de tarefas, dependência (transações, ou tarefas, dependentes entre si) e reconfiguração dinâmica.

Workflows são modelados como conjuntos de tarefas transacionais. Estas tarefas possuem conectores de entrada de dados (*inputs*) e de saída (*outputs*), representados por círculos na Figura 30. Instâncias de processos são implementadas como conjuntos de tarefas cujos conectores estão interligados. Tarefas podem ainda ser compostas por várias sub-tarefas, organizadas em sub-processos, o que pode ser visto como tarefas aninhadas na Figura 30 (t1, t2 e t3). O SGWF possui dois níveis lógicos: o nível de fluxo de dados, composto pelas tarefas interligadas da forma descrita anteriormente, e o (meta)nível de controle do workflow, ligado separadamente mas obedecendo a esta mesma topologia. O SGWF é controlado (e programado) utilizando reflexão computacional. A transferência de dados e de notificações de controle são transacionais.

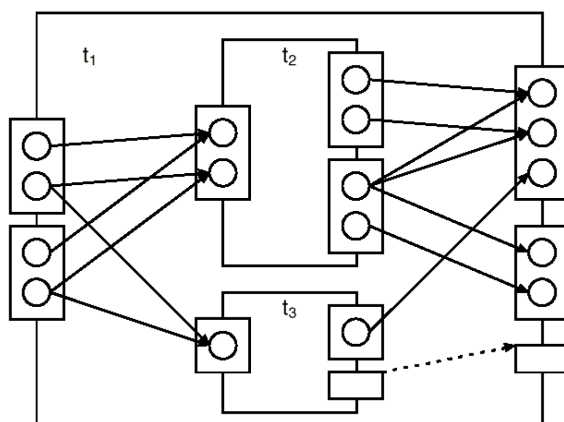

Figura 30: Tarefa composta

O sistema é subdividido em serviço de execução e serviço de dados. As atividades são realizadas por *tasks* (nível de trabalho - *work*). Estes objetos são *wrappers* que controlam a execução das aplicações invocadas. O fluxo de dados e controle é implementado por *task controllers* (meta-nível de fluxo - *flow*), vide Figura 31. Os *task controllers* são instanciados no início da execução do workflow de forma a serem notificados quando a tarefa deve ser executada, além de receber os dados necessários a sua execução.

O sistema é um *framework*, programado em CORBA e Java. Utiliza os serviços OTS – o Serviço de Transações de Objetos (*Object Transaction Service)*, a Facilidade de Meta-Objetos (*Meta-Object Facility*) e o Serviço de persistência da arquitetura OMA. É programado usando Reflexão Computacional, através da Facilidade de Meta-Objetos CORBA. Não possui, portanto, uma linguagem de definição de workflow convencional. A cada tarefa é associado um controlador. Controladores, assim como tarefas, podem executar em diversas configurações distribuídas.

Tarefas podem ser criadas em nós diferentes de seus controladores e vice-versa. O que torna a disposição destes objetos nos nós na rede bastante flexível, permitindo o uso de inúmeras configurações distribuídas. As tarefas e controladores, contudo, tem suas localizações previamente determinadas. Os *task controllers* são instanciados no início da execução do sistema por uma tarefa especial chamada *gênesis*. Os controladores de um caso podem ser criados todos de uma única vez ou, em pequenos conjuntos, durante a execução do caso.

O sistema é destinado a aplicações de comércio eletrônico, *home banking*, corretoras e outros, onde os recursos são pré definidos.

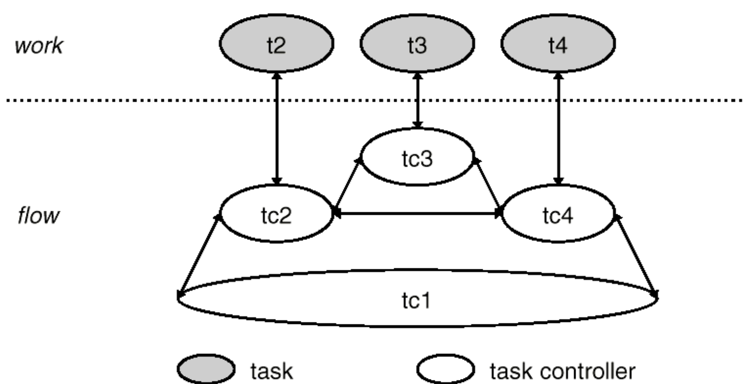

Figura 31 Controladores para tarefas compostas

O plano pode ser reconfigurado durante a execução do workflow (*dynamic change*). Para tal, *os task controllers* são reorganizados de maneira a refletir estas mudanças. O controle é, desta forma, distribuído.

A escalabilidade é alcançada através da distribuição de controle e de tarefas entre vários nós do sistema. *task controllers* podem ser instanciados em um mesmo nó, de forma a minimizar o

tráfego de dados de controle pela rede, ou podem ser criados em nós separados, de forma a distribuir a carga de processamento, além de permitir que a falha de um nó tenha pouco impacto no sistema como um todo.

Instâncias de processos são geradas por tarefas especiais chamadas *gênesis tasks*. A execução do workflow pode ser monitorada consultando-se os *task controllers*. A interoperabilidade com outros SGWFs é implementadas por tarefas do tipo *Adapter Tasks*.

A tolerância a falhas é implementada de diversas maneiras. Objetos de controle (*task controllers*) podem executar em nós diferentes dos objetos que realizam atividades (*tasks*); ou podem ser replicados, introduzindo redundância ao sistema. A tolerância a falhas de enlace de rede é provida pelos canais transacionais: os dados são transmitidos entre *task controllers* de forma transacional, garantindo sua consistência. As tarefas também são executadas de forma transacional.

9.7.1 Nortel X WONDER

O conceito de *AdapterTasks* empregado no projeto da Nortel é semelhante ao *GatewayActivity* do WONDER.

Na arquitetura WONDER não há separação entre controle e execução da atividade, como feito no sistema da Nortel, ambos são desempenhados pelo *ActivityManager.* Adicionalmente, no WONDER, dados da atividade e informações de controle são enviados conjuntamente. A proposta da Nortel não especifica a maneira como os dados são transportados entre tarefas.

Na proposta da Nortel, a estrutura de controle do workflow (servidores CORBA distribuídos) é montada, totalmente ou por partes, antes da execução das tarefas, ao contrário do WONDER que cria a atividade sob demanda, conforme o caso se movimenta. Esta última abordagem torna o WONDER mais flexível e tolerante a falhas e mudanças no plano, requerendo um procedimento de reconfiguração mais simples.

9.8 Proposta do OMG

A atual proposta do OMG para a *Business Object Domain Facility* [jFlow98] foi submetida inicialmente por um conjunto por 20 empresas, dentre elas Oracle, IBM e DEC. A proposta é centrada no conceito de *Workflow Object.* A partir deste objeto, são especializados a maioria dos objetos que compõem o modelo (atividades, processos, bases de dados, eventos e especificações de processo). Estes objetos são servidores CORBA que, a princípio podem ser criados em qualquer nó do sistema.

Suporte a escalabilidade, tolerância a falhas, disponibilidade, transações, capacidade de alteração dinâmica do plano (workflow dinâmico), políticas de alocação de recursos (atores ou pro-

cessos automáticos) a atividades e outros requisitos não são responsabilidade desta especificação que objetiva apenas prover um conjunto de interfaces genérico para os principais elementos que compõem um SGWF.

9.9 Outros Sistemas

São também exemplos de sistemas que propõem soluções para o problema de execução de workflow de larga escala:

O **EVE Workflow** [GT98; GKT98] é um SGWF baseado no middleware de suporte a eventos persistentes EVE. *middleware* EVE foi desenvolvido por Geppert et al. na Universidade de Zürich e é baseado em um modelo B/SM (*Broker/Service Model*) [TGD97a; TGD97b] de ativação de serviços por eventos. *Brokers* distribuídos por nós da rede implementam as atividades do workflow, comunicando-se através de eventos armazenados em EVE-Servers distribuídos.

O projeto **Wide** (*Workflow on Intelligent Distributed database Environment*) [CGP+96; BCC+99; CGS96; Sn97; Sn99; VGBA99], desenvolvido por um consórcio internacional, cujo objetivando era o desenvolvimento de um SGWF baseado em uma tecnologia de bancos de dados ativos, utilizando modelos de transações avançadas, provida por servidores de bancos de dados distribuídos.

O **BPAFrame2** [SM98], desenvolvido por três universidades da Alemanha, Fakultät Informatik, TU Dresden; Institt für Informatik, e TU Bergakademie Freiberg. Define um *framework* orientado a objetos, para ambientes distribuídos. Destaca-se principalmente no encapsulamento de recursos em EJBs (*Enterprise Java Beans*), no mapeamento de tarefas, feito de forma flexível, e na descentralização do controle.

Forté Conductor + Fusion. Conductor é um SGWF comercial produzido pela Forté Software Inc. [But97; Mann99]. Este sistema permite o uso de servidores replicados. A administração do sistema pode ser realizada de maneira distribuída, ou centralizada. Fusion é uma camada de software que provê uma infra-estrutura XML permitindo a integração de sistemas heterogêneos.

Capítulo 10

Conclusões

Este trabalho descreve o projeto e a implementação do WONDER *(Workflow ON Distributed EnviRonment*), uma arquitetura de software distribuída para a execução de workflow de larga escala. A arquitetura utiliza o conceito de casos móveis, onde agentes, que carregam o plano do workflow juntamente com seus dados, trafegam pelos nós da rede conforme o workflow é realizado. Estes agentes utilizam o paradigma de agentes móveis. Diferente das demais arquiteturas centralizadas tradicionais e algumas distribuídas, como o IBM MQSeries e o MENTOR e EVE, o WONDER é composto por servidores CORBA de granulosidade bastante fina, no nível de atividades, característica que provê maior flexibilidade e controle de distribuição. A arquitetura utiliza-se de vários servidores auxiliares, definidos de maneira a satisfazer requisitos como tolerância a falhas, auditoria e monitoramento, além de auxiliarem o caso móvel em sua migração.

Um mapeamento da arquitetura e do paradigma de agentes móveis para CORBA foi proposto, com seus problemas e decisões de projeto discutidos. As similaridades da arquitetura WONDER com modelos propostos pelos projetos $METEOR_2$, INCAS, Nortel, Mentor e Exotica/FMQM foram apresentados e discutidos.

Um protótipo com as funcionalidades básicas do sistema, incluindo o suporte a agentes móveis, persistência de objetos e as principais políticas de tolerância a falhas como *checkpointing*, foi implementado. Detalhes, decisões de projeto e principais dificuldades relacionados à sua implementação foram também apresentados e discutidos.

Para a especificação dos workflows a serem executados pela arquitetura foram definidas e implementadas a linguagem PLISP, de especificação de workflow, e o aplicativo WStarter, que cria e executa ambientes de testes e um sistema de coleta de dados de desempenho e de geração de relatórios.

Resultados de testes de desempenho envolvendo a execução de SGWFs são ausentes na literatura. O presente trabalho propõe e realiza um conjunto de testes que permite comparar os tempos de execução de SGWFs em diferentes configurações de distribuição. Este conjunto e testes

definidos e realizados neste trabalho são, desta forma, uma contribuição importante para a pesquisa na área de testes envolvendo SGWFs, em especial, sistemas de larga escala.

Dentre os testes de desempenho realizados, uma análise detalha dos tempos médios de execução das atividades em valores absolutos e relativos foi realizada. Estes testes permitiram contabilizar os custos, em termos de atrasos, associados à migração, criação e configuração do agente móvel da arquitetura.

O protótipo foi testado em várias configurações distribuídas e centralizadas. Por ter sido projetado para ser executado em um sistema completamente distribuído, a execução centralizada da arquitetura sobrecarrega as máquinas utilizadas nos testes a partir da execução de algumas dezenas de casos concorrentes. Entretanto, os testes mostraram que, para um grande volume de casos concorrentes, a distribuição do processo entre vários servidores resulta em menores atrasos e em uma menor carga de processamento e E/S nas máquinas envolvidas.

A hipótese inicial, de que o uso da descentralização de controle e de processamento permite diminuir a carga de processamento do servidor central, provendo maior disponibilidade, foi comprovada pelos testes de desempenho. O aumento da distribuição, reduz a taxa de crescimento dos tempos médios de execução dos casos concorrentes. Desta forma, a escalabilidade pode ser provida aumentando-se o número de máquinas utilizadas na execução do workflow. O uso do paradigma de agentes móveis permite descentralizar dados e controle, distribuindo a carga de processamento e de comunicação, de maneira mais uniforme, pelos nós do sistema, além de torna-lo mais tolerante a falhas, aumentando a disponibilidade do SGWF.

10.1 Principais Características e Contribuições

Várias soluções de projeto e implementação foram empregadas na arquitetura WONDER. São listados a seguir, estas principais contribuições e características.

Alocação Dinâmica de Atores. Atores são alocados a tarefas em tempo de execução, com base em políticas que podem utilizar informações relacionadas à carga de trabalho destes atores ou dados históricos de execuções anteriores dos casos. Esta característica torna o sistema mais flexível e tolerante a falhas.

Otimização na passagem de Dados e Referências. Diferentemente de outras arquiteturas que passam somente links ou somente dados, o WONDER adota uma política mista, onde dados são passados entre atividades consecutivas somente quando necessários.

Sistema Hierárquico de Gerenciamento. São definidos coordenadores de caso e de processo, permitindo o gerenciamento distribuído da execução de workflows. Esta abordagem permite o monitoramento do sistema sem degradação de desempenho, permitindo a detecção de falhas e sua correção.

Utilização de um Serviço de Nomes Descentralizado. Foi definido e implementado um espaço de nomes proprietário, baseado em servidores de nomes locais a cada nó. Esta abordagem elimina o servidor de nomes central e torna o sistema mais tolerante à falhas.

Suporte a Workflow Dinâmico. O plano das atividades em execução pode ser alterado de maneira a refletir mudanças dinâmicas no processo. O sistema pode ser modificado, em tempo de execução, de maneira a contornar gargalos de desempenho e falhas de enlaces de comunicação ou de nós, permitindo um melhor balanço de carga e uma melhor tolerância a falhas.

Programação de alto nível através da linguagem PLISP. As definições de processo podem ser definidas através de uma linguagem de definição de workflow semelhante a LISP onde todos os parâmetros do sistema podem ser especificados.

Implementação em CORBA de um SGWF. A comunicação do sistema utiliza o *framework* de comunicação CORBA, definindo IDLs para cada componente da arquitetura, fornecendo uma especificação genérica. Esta especificação permite a implementação destes componentes em diferentes linguagens de programação que podem ser portadas para várias plataformas de hardware e software.

Além das características acima, a granulosidade muito fina de objetos (definição de um servidor para cada atividade), provê uma grande flexibilidade de configuração destes servidores quanto a sua localização. Esta característica permite ajustar o sistema para atender a diferentes requisitos de distribuição, balanceamento de carga, segurança, tolerância a falhas e gerenciamento. A configuração destes parâmetros é provida pela linguagem PLISP, que permite especificar a localização de cada servidor, além de ser auxiliada por configurações dos *TaskLists* e dos servidores de papéis.

10.2 Requisitos da Distribuição

O paradigma distribuído, baseado em agentes móveis, introduz ou altera alguns requisitos dos SGWFs convencionais. São estes:

Segurança. Políticas de segurança adicionais devem ser adotadas. Dados agora trafegam por todos os nós, em máquinas que são potencialmente menos confiáveis que o servidor central, tanto do ponto de vista de controle de acesso, como da vulnerabilidade a falhas de software e hardware. Este problema é responsabilidade do repositório de objetos do WONDER.

Configuração. Por executar em um ambiente distribuído, a configuração do sistema torna-se mais custosa. Cada máquina deve ter o ambiente WONDER instalado com suas configurações de segurança e acesso previamente definidas. Cada máquina deve ainda possuir espaço em disco necessário para receber os agentes e seus dados. A arquitetura WONDER permite, contudo, amenizar este problema utilizando do NFS.

Requisitos de Memória dos Servidores Java. No protótipo implementado, a granulosidade muito fina dos componentes da arquitetura, que ocorre no nível de atividades, requer um maior uso de memória. Isto ocorre devido ao uso de servidores que executam em máquinas virtuais Java independentes.

Modificação de Nomes em Tempo de Execução. O uso de nomes dependentes de localização torna mais complexa a implementação de operações que necessitam mudar a localização de servidores, como é o caso de balanceamento de carga dinâmico e alguns procedimentos de resolução de falhas. A escalabilidade alcançada com o uso da arquitetura distribuída, contudo, compensa este custo adicional.

Otimização da Comunicação Local dos Servidores. A comunicação entre processos locais deve ser otimizada de forma a utilizar meios mais eficientes como memória compartilhada, chamada de operações locais ou *pipes*, por exemplo. No protótipo implementado, a comunicação local entre processos é feita via IIOP sobre *sockets*. Este problema é, contudo, específico do protótipo implementado, podendo ser contornado através da implementação destas políticas de comunicação locais ou do uso de ORBs que diferenciam chamadas locais de remotas, a exemplo do protocolo IPC SAP [Schmidt92] utilizado no ORB TAO.

10.3 Discussão

Uma aplicação importante da arquitetura WONDER é o seu uso no isolamento de diferentes departamentos ou filiais de uma grande corporação. Deseja-se executar casos em domínios (sub-redes) específicas, geograficamente distantes, sem que haja a dependência de um computador central que, em última instância, implicaria em acessos remotos, utilizando enlaces de WANs. O sistema precisa ser imune a falhas ou atrasos mais comuns em redes de longa distância. Neste tipo de aplicações quer-se poder explicitamente especificar onde executar determinadas atividades e coordenadores de casos e processos.

Neste contexto, o uso de referências dependentes de localização, como são as da arquitetura WONDER são ideais. A capacidade de especificar onde cada coordenador será executado permite implementar domínios administrativos independentes, com o mínimo de comunicação através de WANs.

Outra aplicação importante é o uso de SGWFs que suportem o uso de clientes móveis. Devido a sua granulosidade muito fina de objetos (atividades são executadas em nós específicos do sistema, levando consigo os dados que necessitam), o uso da arquitetura WONDER em ambientes móveis é uma possibilidade interessante, bastando fazer alguns ajustes no protocolo de envio de eventos entre a atividade e o coordenador do caso, e no protocolo de migração de agentes: o envio de eventos deve ser retardado, sendo realizado apenas quando o enlace de comunicação estiver operante. A migração do *ActivityManager* deve obedecer esta mesma políti-

ca, sendo somente realizada quando a conexão de rede estiver disponível, ou quando a banda passante permitir.

10.4 Trabalhos Futuros

Como futuras extensões e trabalhos relacionados propõe-se:

Realizar a implementação dos componentes e das características da arquitetura não disponíveis no protótipo. A exemplo do Coordenador de Histórico, do Servidor de Backup e da atividade de Gateway; Realização de estudo mais detalhado das questões de segurança, envolvendo controle do acesso a dados que trafegam pelos nós do sistema, e autenticação de atividades.

Estudar a possibilidade de da utilização da linguagem XML, seguindo os padrões propostos pela WfMC [WfMC-TC-1023], na implementação da integração do WONDER com outras arquiteturas, provido pela atividade *Gateway.*

Estudar o uso dos serviços CORBA, como o de transações, notificação, segurança e eventos, e seu impacto, na arquitetura WONDER, considerando também os novos serviços disponíveis no padrão CORBA 3.0, a exemplo do serviço de persistência de servidores CORBA e passagem de objetos por valor, ou mesmo características da CORBA 2.3 como o POA.

Estudar algumas otimizações que podem ser feitas na arquitetura. Alguns dos servidores, podem ser reescritos em C++, de maneira a prover maior desempenho e menor consumo de memória. Coordenadores são bons candidatos para esta otimização; O único componente que necessita de recursos específicos da linguagem Java, que dão suporte à mobilidade, são as Atividades. O uso de CORBA permite que a rescrita de servidores em C++ possa ser feita de maneira transparente, não afetando a interação entre servidores escritos em Java e C++.

Implementar a comunicação local entre processos de forma diferenciada da comunicação remota. Outra otimização seria a execução de dois ou mais servidores (*ActivityManager*) em um mesmo processo do sistema operacional, compartilhando código. Estas alterações reduziriam o consumo local de memória e aumentariam a velocidade de seqüenciamento de atividades.

Implementar políticas de balanceamento de carga usando os coordenadores. O paradigma de agentes móveis pode ser estendido a estes servidores CORBA. Desta forma, os coordenadores de caso e de processo podem ser movidos entre os nós do sistema, em momentos onde a carga da máquina onde o servidor está executando torna-se crítica. Servidores de casos podem ser ainda replicados. Neste caso, ao invés de haver um nome para cada servidor, haveria um nome para o conjunto destes servidores. Em ambos os casos, como os nomes do WONDER são dependentes de localização, será necessário introduzir uma indireção, um servidor de nomes, que mantenha a referência de objeto consistente conforme o coordenador se move, de maneira que os eventos gerados pelas atividades (*ActivityManagers*) possam chegar ao seu destino. Políticas como *proxies* inteligentes podem ainda ser usadas.

Referências

[AAAM95] G. Alonso, A. Agrawal, C. El Abbadi, and R. Mohan. Exotica/FMQM: A persistent message-based architecture for distributed workflow management. In Proceedings of the IFIP WG8.1 Working Conference on Information Systems for Decentralized Organizations, Trondheim, August 1995. Also available as IBM Research Report RJ9912, IBM Almaden Research Center, November 1994.
http://www.almaden.ibm.com/cs/exotica/exotica_distributed_workflow_ifipc95/

[AAEM97] G. Alonso, D. Agrawal, A. El Abbadi, and C. Mohan. Functionality and Limitations of Current Workflow Management Systems. IEEE Expert – Special Issue on Cooperative Information Systems, 1999. Also available as a Technical Report at IBM, 1997.
http://www.inf.ethz.ch/personal/alonso/PAPERS/IEEE-Expert.ps.Z

[ABVVV00] W. M. P. van der Aalst, T. Baesten, H. M. W. Verbeek, P. A. C. Verkoulen, and M. Voorhoeve. Adaptative Workflow. Enterprise Information Systems. Joaquim Felipe (Ed.). Kluwer Academic Publishers. Netherlands, May 2000, pp. 63-70. ISBN 0-7923-6239-X.

[AGK+95] G. Alonso, R. Güthör, M. Kamath, D. Agrawal, A. El Abbadi, and C. Mohan. Exotica/FMDC: Handling disconnected clients in a workflow management system. In Proc. 3rd International Conference on Cooperative Information Systems, Vienna, May 1995.
http://www.almaden.ibm.com/cs/exotica/exotica_mobile_computing_coopis95.ps

[AKA+94] G. Alonso, M. Kamath, D. Agrawal, A. El Abbadi, R. Günthör, and C. Mohan. Failure Handling in Large Scale Workflow Management Systems. Research Report RJ9913, IBM Almaden Research Center, November 1994.

[BCC+99] L. Baresi, F. Casati, S. Castano, M. G. Fugini, I. Mirbel, and B. Pernici. Wide workflow design methodology. In Proceedings of the WACC'99, San Francisco, CA, February 1999. http://dis.sema.es/projects/WIDE/Documents/3027-6.pdf

[BJR97] G. Booch, I. Jacobson, J. Rumbaugh, UML Distilled Applying the Standard Object Modeling Language, Addison-Wesley, Reading, MA, 1997.

[BMR96] D. Barbara, S. Mehrotra, and M. Rusinkiewicz. INCAs: Managing Dynamic Workflows in Distributed Environments. Journal of Database Management, Vol. 7, No. 1, 1996. http://www.buva.sowi.uni-bamberg.de/ps-Sammlung/literatur/workflowUnterlagen/vortel/barbara.ps.gz

[But97] P. Butterworth. Automating the business process of mission-critical distributed applications. Technical Report. Forté Software Inc., 1997. http://www.forte.com.

[CGP+96] F. Casati, P. Grefen, B. Pernici, G. Pozzi, and G. Snchez. Wide workflow model and architecture. Technical Report, Dipartimento di Elettronica e Informazione, Politecnico di Milano, 1996. http://dis.sema.es/projects/WIDE/Documents/ase30_4.ps.gz

[CGS96] S. Ceri, P. Grefen, and G. Sánchez. Wide – A distributed architecture for workflow management. Technical Report, Wide Consortium, December 1996. http://dis.sema.es/projects/WIDE/Documents/ridepap.ps.gz

[CH97] Chapman & Hall. JacORB: Implementation and Design of a Java ORB. Procs. of DAIS'97, IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems, September 30 - October 2, Cottbus, Germany, 1997. http://www.inf.fu-berlin.de/~brose/jacorb/

[CHK94] D. Chess, C. Harrison, and A. Kershenbaum. Mobile Agents: are they a good idea?. IBM Research Report, IBM T. J. Watson Research Center, Yourktown Heights, N.Y. RC 19887, December 1994. http://www.research.ibm.com/massdist/mobag.ps

[Concordia97] Mitsubishi Electric. Concordia: An Infrastructure for Collaborating Mobile Agents. In Proceedings of the 1st International Workshop on Mobile Agents (MA '97), April 1997.

[CORBA98] The Common Object Request Broker: Architecture and Specification. Object Management Group, Framingham, MA, 1998.

[CORBAfac] OMG - CORBAfacilities: Common Facilities Architecture - rev. 4.0. June 1997.

[CORBAserv] OMG - CORBAservices: Common Object Services Specification. June 1997.

[CORBAspec] OMG - The Common Object Request Broker: Architecture and Specification - rev. 2.2. 1998.

[COSSpec97] Common Object Services Specification. Object Management Group, Framingham, MA, 1997.

[CPV97] A. Carzaniga, G. P. Picco, and G. Vigna. Designing Distributed Applications With Mobile Code Paradigms. Proceedings of the 1997 International Conference on Software Engineering, 1997, pp. 22 – 32.

[DEC1992] Teamroute Programming Guide, Order Numver: AA-PM6FA-TE, Digital Equipment Corporation, Maynard, June 1992.

[DKM+97] S. Das, K. Kochut, J. Miller, A. Sheth, and D. Worah. ORBWork: A reliable distributed CORBA-based workflow enactment system for METEOR. Technical Report. Department of Computer Science, University of Georgia, 1997. UGA-CS-TR97001. http://lsdis.cs.uga.edu/lib/download/SD+96.ps

[DLMM93] M. Day, B. Liskov, U. Maheshwari, and A. C. Myers. References to remote mobile objects in Thor. ACM Letters on Programming Languages and Systems 2, 1-4, March. 1993, pp. 115- 126.

[EGR91] C. A. Ellis, S. J. Gibbs, and G. L. Rein. Groupware: Some Issues and Experiences. CACM 34:1, January 1991, pp. 38-58.

[EKR95] C. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. Proceedings of conference on Organizational computing systems. 1995, pp. 10 – 21.

[Elmargamid92] A. Elmargamid (Ed.). Database Transaction Models for Advanced Application. Morgan Kaufmann, San Mateo, CA, 1992.

[Emmerich97] W. Emmerich. An introduction to OMG/CORBA (tutorial). In. Proceedings of the 1997 international conference on Software engineering , 1997, pp. 641 – 642.

[Flanagan99] D. Flanagan. Java in a Nutshell. O'Reilly & Associates, 3rd. Edition. December 1999. ISBN 1565924878.

[Fowler85] R. J. Fowler. Decentralized object finding using forwarding addresses. Tech. Rep. 85-12-1, Dept. of Computer Science, Univ. of Washington, December 1985.

[FreeCORBA] The Free CORBA Page. http://adams.patriot.net/~tvalesky/freeCORBA.html

[Freedman00] D. H. Freedman. Attack of the Killer Viruses. The New York Times - Editorial Desk. May 6, 2000, Saturday Edition.

[FS97] Martin Fowler and Kendall Scott. UML Distilled - Applying the Standard Object Modeling Language. Object Technology Series. Addison-Wesley, June 1997.

[FSF] Free Software Foundation. http://www.fsf.org/

[FWME00] R. S. Silva Filho, J. Wainer, E. R. M. Madeira, and C. Ellis. CORBA Based Architecture for Large Scale Workflow. Special Issue on Autonomous Decentralized Systems. IEICE Transactions on Communications. Tokyo, Japan, Vol. E83-B, No. 5. May 2000, pp.988-998.

[FWME99a] R. S. Silva Filho, J. Wainer, E. R. M. Madeira, and C. Ellis. CORBA Based Architecture for Large Scale Workflow - 4th International Symposium on Autonomous Decentralized Systems (ISADS '99), March 20-23, 1999 - Tokyo, JAPAN. pp. 276-283. ISBN 0-7695-0137-0. IEEE Computer Society Eds.

[FWME99b] R. S. Silva Filho, J. Wainer, E. R. M. Madeira, and C. Ellis. WONDER: A Distributed Architecture for Large Scale Workflow Using CORBA - 17th Brazilian Symposium on Computer Networks - (SBRC'99) – Salvador, BA, Brazil - May 25-28, 1999. pp. 379-380.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns—Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[GHKM94] D. Georgakopoulos, M. Hornick, P. Krychniak, and F. Manola. Specification and Management of Extended Transactions in a Programmable Transaction Environment. In

Proceedings of the 10th. Intl. Conference on Data Engineering, Houston, TX, February 1994, pp. 462-473.

[GHNCSE97] S. Green, L. Hurst , B. Nangle, P. Cunningham, F. Somers, and R. Evans. Software Agents: A review. Trinity College Dublin. May 1997.

[GKT98] A. Geppert, M. Kradolfer, and D. Tombros. Workflow specification and event-driven workflow execution. Technical report, University of Zürich, 1998. ftp://ftp/ifi.unizh.ch/dbtg/projects/EXE/si.os.gz

[GMS87] H. Gargcia-Molina, and K. Salem. Sagas. In Proceedings of ACM-SIGMOD 1987, International Conference on Management of Data. San Francisco, 1987.

[Graham95] P. Graham. ANSI Common LISP. Prentice Hall. November, 1995, ISBN: 0133708756.

[Gray96] R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In Proceedings of the 4th Annual Tcl/Tk Workshop (TCL '96), July 1996.

[GT98] A. Geppert, D. Tombros. Event-based Distributed Workflow Execution with EVE, Proceedings Middleware'98, The Lake District, England, September 1998, pp. 427-442.

[Henning98] M. Henning. Binding, migration, and scalability in CORBA. Communications of the ACM, Volume 41, Num. 10. ACM Press. October 1998, pp. 34 – 36.

[HL87] R. V. Hogg, J. ledolter. Engineering Statistics. Macmillan Publishing, 1987. ISBN 0-02-355790-7.

[IBM99] IBM. IBM MQSeries Workflow – Concepts and Architecture. Edition 3.2, June 1999. ftp://ftp.software.ibm.com/software/mqseries/pdf/fmcg0mst.pdf

[IH99] L. Ismail,, D. Hagimont. A Performance Evaluation of the Mobile Agent Paradigm. Proc. Of the 1999 ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. November 1-5, 1999, Denver, CO USA.

[IIOP98] OMG doc formal/98-02-33-CORBA / IIOP 2.2 Specification - Full version - 1998.

[IloveYou00] I Love You Virus in News. CNET.com News – Enterprise Computing. May 16, 2000. http://news.cnet.com/news/0-1003-201-1826257-0.html?tag=st.cn.sr1.ssr.ne_virus .

[Iona] Iona Technologies. http://www.iona.com/

[JavaCC] Java Compiler Compiler. Metamata Tech. http://www.metamata.com/JavaCC/

[JavaORB] Distributed Object Group. http://www.multimania.com/dogweb/

[JB96] S. Jablonski and C. Bussler. Workflow Management: Modeling Concepts, Architecture and Implementation. International Tompson Computer Press Eds., 1996.

[Jflow98] OMG Business Object Domain Task Force BOTF-RFP 2 join Submission – jFlow. Workflow Management Facility, revised submission. OMG bom/98-03-04.

[JRS95] D. Johansen, R. van Renesse, and F. B. Schneider. Operating System Support for Mobile Agents. In Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems (HotOS-V), May 1995, pp. 42 - 45.

[KLO97] G. Karjoth, D. Lange, and M. Oshima. A Security Model for Aglets. IEEE Internet Computing, July-August 1997, pp. 68 - 77.

[Kobryn99] Cris Kobryn. UML 2001: a standardization odyssey. Communications of the ACM 42, 10 (Oct. 1999), pp. 29 –37.

[KT98a] N. M. Karnik and A. R. Tripathi. Agent Server Architecture for the Ajanta Mobile Agent System. In Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98), July 1998.

[KT98b] N. M. Karnik and A. R. Tripathi. Design Issues in Mobile-Agent Programming Systems. IEEE Concurrency, July-September 1998.

[MAAEGK95] C. Mohan, D. Agrawal, G. Alonso, A. El Abbadi, R. Güthör, M. Kamath. Exotica: A project on Advanced Transaction Management and Workflow Systems. ACM SIGOIS Bulletin, Vol. 16, No. 1, August 1995.

[MAGK95] C. Mohan, G. Alonso, R. Güthör, and M. Kamath. Exotica: A research perspective for workflow management systems. IEEE Data Engineering Bulletin, Vol. 18, No.1, March 1995. pp.19-26. http://www.almaden.ibm.com/cs/exotica/exotica_overview_de03_95.ps

[MAGKR95] C. Mohan, G. Alonso, R. Günthör, M. Kamath, B. Reinwald. An Overview of the Exotica Research Project on Workflow Management Systems. Proc 6th Int'l Workshop on High Performance Transaction Systems, Asilomar, September 1995.

[Mann99] J. Mann. Forté fusion product profile. Technical Report. Forté Software Inc., June 1999. http://www.forte.com

[MICO] Mico Is CORBA. http://www.mico.org

[MPS+97] J. Miller, D. Palaniswani, A. Sheth, K. Kochut, and H. Singh. Webwork: METEOR's web-based workflow management system. Journal of Intelligent Information Systems, pp. 185-215, 1997.

[MSKW96] J. Miller, A. Sheth, K. Kochut, and X. Wang. CORBA-Based runtime architectures for workflow management systems. Journal of Database Management, Special Issue on Multidatabases, Vol. 7 No.1, 1996, pp. 16-27. http://lsdis.cs.uga.edu/lib/download/MS+96.ps

[ObjectSpace97] ObjectSpace: ObjectSpace Voyager Core Package Technical Overview. Tech. Repot. ObjectSpace Inc. Dallas, 1997. http://www.objectspace.com.

[OH98] R. Orfali and D. Harkey. Client/Server Programming with Java and CORBA. Second Edition. John Wiley & Sons, 1998

[OHE97] Robert Orfali, Dan Harkey, and Jeri Edwards. Instant CORBA. John Wiley & Sons, 1997.

[OMG00] Object Management Group. OMG TC Work in Progress. www.omg.org/library/shedule.htm

[OMG95] The Common Object Request Broker: Architecture and Specification - OMG - Revision 2.0 July 1995.

[OMG99] Joint submission. OMG Workflow Management Facility. OMG dtc/99-07-05, Jul., 30, 1999.

[OMG-MASIF98] OMG doc orbos/98-03-09 - Evaluation Report on the Mobile Agent Facility Joint Submission. Revised Version.

[OMG-OBV96] OMG doc orbos/96-06-14 - Objects-By-Value RFP

[OpenFusion] PrismTech Openfusion CORBA Services. http://www.prismtechnologies.com/products/openfusion/

[Opensource] Opensource organization. http://www.opensource.org

[OrbixPrg98] OrbixWeb Programmer's Guide. Iona Technologies PLC, September 1998.

[OrbixWeb] Iona Technologies Java ORB Implementation. http://www.iona.com/products/orbixweb/index.html

[POA97] Object Management Group. Specification of the Portable Object Adapter (POA). OMG Document orbos/97-05-15 ed., June 1997.

[PS98] I. Pyarali and D. C. Schmidt. An Overview of the CORBA Portable Object Adapter, Special Issue on CORBA in the ACM StandardView magazine, Vol. 6, No. 1, March 1998.

[RGK97] D. Rus, R. Gray, and D. Kotz. Transportable Information Agents. Proceedings of the first ACM international conference on Autonomous agents, 1997, pp. 228 – 236.

[RS95] A. Reuter and F. Schwenkreis. Contracts - A Low Level Mechanism for Building General-Purpose Workflow Management Systems. Bulletin of the Technical Committee on Data Engineering, Vol. 18, No. 1, 1995. IEEE Computer Society.

[Schmidt92] D. C. Schmidt. IPC SAP: An Object-Oriented Interface to Interprocess Communication Services. C++ Report, vol. 4, November/December 1992.

[Schulze99] B. R. Schulze. Migração transparente em sistemas de agentes móveis usando CORBA. Tese de Doutorado, Instituto de Computação. UNICAMP, July 1999.

[Schwenkreis93] F. Schwenkreis. APPRICOTS - A Prototype Implementation of ConTract System - Management of the Control Flow and the Communication System. Proc. of the 12th Symposium on Reliable Distributed Systems, Princeton (NJ), 1993. IEEE Computer Society press.

[Seetharaman98] K Seetharaman. The CORBA Connection. Communications of the ACM, Vol. 41, No. 10. ACM Press. October 1998, pp. 34 – 36.

[Siegel98] J Siegel. OMG overview: CORBA and the OMA in enterprise computing. Communications of the ACM, Vol. 41, No. 10. ACM Press. October 1998. pp. 37 – 43.

[SKM+96] A. Sheth, K. Kochut, J. Miller, D. Worah, S. Das, C. Lin, D. Palaniswami, J. Lynch, and I. Shvchenko. Supporting state-wide immunization tracking using multi-paradigm workflow technology. In Proc. Of the 22nd . Intl. Conf. On Very Large Databases (VLDB96), September, 1996.

[SM98] A. Schill, and C. Mittasch. A Generic Workflow Environment based on CORBA Business Objects, Proc. of Middleware'1998, pp. 19-34.

[Sn97] G. Sánchez. The WIDE Workflow Model and Language. Technical Report. Wide Consortium. October 1997. http://dis.sema.es/projects/WIDE/Documents/4080-2.pdf

[Sn99] G. Sánchez. The WIDE Project: Final Report. Technical Report. Wide Consortium, October 1997. http://dis.sema.es/projects/WIDE/4111-2.pdf

[SR93] A. Sheth and M. Rusinkiewicz. On Transactional Workflows. Data Engineering Bulletin, Vol. 16, No. 2, June, 1993. http://lsdis.cs.uga.edu/lib/download/SR93.ps

[SRD99] R. Stewart, D. Rai, S. Dalal – "Building Large-Scale CORBA-Based Systems" - Component Strategies, January 1999, pp.34-44, 59.

[TGD97a] D. Tombros, A. Geppert, and K. Dittrich. The broker/service model for the design of cooperative process-oriented environments. Technical report. University of Zürich, 1997. ftp://ftp.ifi.unizh.ch/pub/techreports/TR-97/ifi-97.o6.os.gz

[TGD97b] D. Tombros, A. Geppert, and K. Dittrich. Semantics of reactive components in event-driven workflow execution. In Proc. 9th. Int'l Conf. On Advanced Information Systems Engineering, Barcelona, Spain, June 1997. ftp://ftp.ifi.unizh.ch/pub/techreports/other_docs/CaiSE97.os.gz

[VGBA99] J. Vonk, P. Grefen, E. Boertjes, and P. Apers. Distributed global transaction support for workflow management applications. Technical Report. University of Ternte, May 1999.

[Vinoski 98] S. Vinoski. New features for CORBA 3.0. Communications of the ACM, Vol. 41, No. 10. ACM Press. October 1998, pp. 44 - 52.

[Visibroker] Inprise Visibroker for Java ORB. http://www.inprise.com/visibroker/

[VM98] F. J. S. Vasconcellos, E. R. M. Madeira. Projeto e Desenvolvimento de Um Suporte a Agentes Móveis Baseado em CORBA. Relatório Técnico, Instituto de Computação, Universidade Estadual de Campinas, 1998.

[WfMC-TC00-1003] D. Hollingsworth. Reference Model - Revision 1.1. Workflow Management Coalition. November 1994. WfMC-TC00-1003. http://www.aiim.org/WfMC/standards/docs/tc003v11.pdf

[WfMC-TC1009] Workflow Client Application Programming Interface (WAPI) Specification, Version 1.2. Workflow Management Coalition, October 1996, WMC-TC-1009. http://www.aiim.org/WfMC/standards/docs/if21009v11.PDF

[WfMC-TC1011] Terminology & Glossary, Version 2.0. Workflow Management Coalition, June 1996, WfMC-TC-1011. http://www.aiim.org/WfMC/standards/docs/glossy3.pdf

[WfMC-TC1012] Workflow Interoperability – Abstract Specification, Version 1.0. Workflow Management Coalition, October 1996, WfMC-TC1012.

[WfMC-TC1015] Audit Data Specification. Workflow Management Coalition – Version 1.0. November 1996. WfMC-TC-1015.

[WfMC-TC1016] Process Definition Interface. Workflow Management Coalition, WfMC TC-1016-X. http://www.aiim.org/WfMC/standards/docs/If19808xb1.PDF

[WfMC-TC-1023] Interoperability Wf-XML Binding. FINAL SPECIFICATION (May 8, 2000) Workflow Standard - Interoperability Wf-XML Binding Document Number WFMC-TC-1023. http://www.aiim.org/WfMC/standards/docs/Wf-XML-1.0.pdf

[WfMC-TC2101] Workflow Facility Specification, Draft. Workflow Management Coalition, WfMC-TC2101.

[White94] J.E. White. Telescript Technology: The Foundation for the Electronic Marketplace. Whitepaper by General Magic, Inc, Sunnyvale, CA, USA, 1994.

[WR93] H. Wächter and A. Reuter. The Contract Model. Ahmed Elmagarmid. Database Transaction Models for Advanced Applications. Morgan Kaufmann, 1993, pp.219-263.

[WSR98] S. M. Weather, S. K. Shrivastava, and F. Ranno. A CORBA Compliant Transactional Workflow System for Internet Applications, Proc. Middleware'1998, pp. 3-17.

[WWWD97] J. Weissenfels, D. Wodtke, G. Weikum, and A. Dittrich. The Mentor Architecture for Enterprise-wide Workflow Management. University of Saarland, Department of Computer Science, 1997.

[WWWD97] J. Weissenfels, D. Wodtke, G. Weikum, and A. K. Dittrich. The Mentor Architecture for Enterprise-wide Workflow Management. Technical Report. University of Saarland, Germany, 1997. http://www-dbs.cs.uni-sb.de/public_html/papers/mentor.html

[WWWK96] D. Wodtke, J. Weissenfels, G. Weikum, and A. Kotz Dittrich. The Mentor Project: Steps Towards Enterprise-Wide Workflow Management. IEEE International Conference on Data Engineering, New Orleans, 1996.

Apêndice A

A.1 Notação da BNF (*Backus Normal Form*):

Listamos a seguir a notação empregada pelo compilador JavaCC [JavaCC] e o gerador de árvores de programa JJTree, integrante desta ferramenta de programação. Esta notação difere um pouco da BNF tradicional.

```
<   >         - Tokens. O texto entre < e > é uma palavra reservada
" "           - Strings. O texto entre "" é considerado como um token
(...)?        - Opcional
(...)*        - Repetição 0 ou mais vezes
(...)+        - Repetição 1 ou mais vezes
...|...       - Alternativa
[... , ... ] - Alternativa entre tokens
... ::= ...   - Regra de produção (... = lados direito e esquerdo da regra)
<EOF>         - Caracter especial de final de arquivo
```

A.2 Gramática da linguagem PLISP

Esta gramática não é sensível à caixa das letras (não é *case sensitive*).

```
Process                  ::= "(" <WORKFLOW> identifier block ")" <EOF>
Block                    ::= ( option_switches )? ( declarations )? ( identifier |
                             sequence_declaration )?
Option_switches          ::= "(" <OPTIONS> ( garbagecollect_option )? ")"
Garbagecollect_option    ::= "(" <GARBAGECOLLECT> identifier ")"
Declarations             ::= "(" <DECLARATIONS> ( simple_activity |
                             gateway_activity | wrapper_declaration |
                             sequence_declaration | andsplit_declaration |
                             orsplit_declaration | data_declaration |
                             application_declaration )+ ")"
Sequence_declaration     ::= "(" <SEQUENCE> identifier ( identifier | activity |
                             split | sequence_declaration )+ ")"
Sequence_list            ::= ( identifier | sequence_declaration )+
Split                    ::= ( identifier | andsplit_declaration |
                             orsplit_declaration )
Andsplit_declaration     ::= "(" <ANDSPLIT> identifier ( string | <NULL> )?
                             sequence_list ")"
Orsplit_declaration      ::= "(" <ORSPLIT> identifier ( string | <NULL> )?
                             sequence_list ")"
Activity                 ::= ( identifier | simple_activity | gateway_activity )
Simple_activity          ::= "(" <ACTIVITY> identifier ( string | <NULL> )? string
                             ( string | <NULL> )? ( string | <NULL> )? ( string |
                             <NULL> )? ( string | <NULL> )? ( wrapper_list | <NULL>
                             )? ")"
Gateway_activity         ::= "(" <GATEWAY> identifier ( string | <NULL> )? string (
                             wrapper_list | <NULL> )? ")"
Wrapper_list             ::= ( identifier | wrapper_declaration )+
Wrapper_declaration      ::= "(" <WRAPPER> identifier application ( read_data |
                             <NULL> )? ( create_data | <NULL> )? ( modify_data |
                             <NULL> )? ")"
Application              ::= ( identifier | application_declaration )
Application_declaration  ::= "(" <APPLICATION> identifier string ")"
Read_data                ::= "(" <READ> data_list ")"
Modify_data              ::= "(" <MODIFY> data_list ")"
Create_data              ::= "(" <CREATE> data_list ")"
data_declaration         ::= "(" <DATA> identifier string identifier ")"
data_list                ::= ( identifier | data_declaration )+
String                   ::= <STRING_LITERAL>
Identifier               ::= <ID>
Identifiers_list         ::= ( identifier )+
```

A.3 Exemplo de um plano escrito em PLISP

```
(workflow st1

  (options
    (garbageCollect false)
  )

// Declaração de identificadores
  (declarations
    (data data1
       "/home/msc98/931680/Data/teste1.doc" reference) // Lido quando necessário
    (data data2
       "/home/msc98/931680/Data/teste2.doc" file)
    (data data3
       "/home/msc98/931680/Data/teste3.doc" reference) // Lido quando necessário
    (data data4
       "/home/msc98/931680/Data/teste4.doc" file)
    (data data5
       "/home/msc98/931680/Data/teste5.doc" file)

    (application ap1
       "/n/dtp/StarOffice5.1/bin/soffice")
    (wrapper wp1
       ap1 (read data1 data2 ) (create data4) (modify data3) )
    (wrapper wp2
       ap1 (read data3 data2 ) (create data5) (modify data1) )
    (wrapper wp3
       ap1 (read data1 data2 data3 ) null (modify data5) )

    (activity act01
       "iguacu" "role1" "role query" "priority" "deadline" "description1" wp1)
    (activity act02
       "iguacu" "role2" "role query" "priority" "deadline" "description1"
       wp1 wp2)
    (activity act03
       "iguacu" "role3" "role query" "priority" "deadline" "description1"
        wp1 wp2 wp3)

  ) // declarations

// Sequencia contendo todas as atividades do workflow
  (sequence sq1
     act01
    (andsplit split1
       (sequence sq2 act01 act02)
       (sequence sq3 act01 act03)
     )
     act02
   )
)
```

Dados podem ser de 3 tipos: "*reference*", "*file*" ou "*query*", representando referências a dados, arquivos e consultas a bancos de dados, respectivamente.

De maneira a indicar quais dados são criados, apenas lidos ou modificados por cada aplicação, o *wrapper* especifica 3 subconjuntos de dados: *modify*, *create* e *read*. Com base nesta informação é possível saber onde o dado foi modificado pela última vez e quais dados são necessários para cada atividade.

O Workflow descrito acima corresponde ao diagrama a seguir:

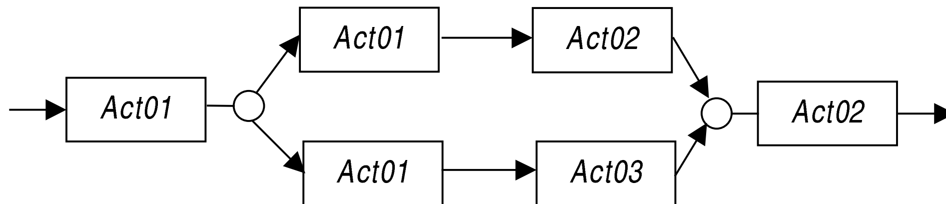

Figura 32: Representação Gráfica do Workflow do Item A.3 usando a notação da WfMC

A.4 Gramática do compilador WStarter

```
Environment                          ::= "(" <ENVIRONMENT> identifier declarations configure run ")" <EOF>
Declarations                         ::= "(" <DECLARATIONS> ( roleCoordinatorDeclaration | processCoordinatorDeclaration | userDeclaration | backupServerDeclaration | historyServerDeclaration )+ ")"
RoleCoordinatorDeclaration           ::= "(" <ROLECOORDINATOR> identifier string string ")"
ProcessCoordinatorDeclaration        ::= "(" <PROCESSCOORDINATOR> identifier string string ")"
UserDeclaration                      ::= "(" <USER> identifier ( string | <NULL> )? ( string | <NULL> )? ( string | <NULL> )? ( string | <NULL> )? ( string | <NULL> )? ")"
HistoryServerDeclaration             ::= "(" <HISTORYSERVER> identifier string ")"
BackupServerDeclaration              ::= "(" <BACKUPSERVER> identifier string ")"
Configure                            ::= "(" <CONFIGURE> ( roleAssign )+ ")"
RoleAssign                           ::= "(" <SETROLE> identifier identifiers_list ")"
Run                                  ::= "(" <RUN> ( newCase )+ ")"
NewCase                              ::= "(" <NEWCASE> identifier identifier string ( numero | <NULL> )? ")"
String                               ::= <STRING_LITERAL>
Identifier                           ::= <ID>
Numero                               ::= <NUM>
Identifiers_list                     ::= ( identifier )+
```

A.5 Exemplo de um ambiente de testes a ser montado pelo compilador WStarter

```
(Environment TestEnvironmesolaris

  ( Declarations
    (RoleCoordinator rc1 "role1" "anhumas.dcc.unicamp.br ")
    (RoleCoordinator rc2 "role2" "tigre.dcc.unicamp.br")
    (ProcessCoordinator pc1 "teste.pls" "anhumas")
    (ProcessCoordinator pc2 "teste2.pls" "anhumas")


    // One TaskList is created for each user
    (User user1 "human" "name1" "homedir" "profile" "araguaia.dcc.unicamp.br ")
    (User user2 "human" "name2" "homedir" "profile" "iguacu.dcc.unicamp.br ")
    (User user3 "human" "name3" "homedir" "profile" "tigre.dcc.unicamp.br ")
  ) // Declarations

  ( Configure
    // Register the users in the role coordinator.
    (SetRole rc1 user1)
    (SetRole rc2 user2)
    (SetRole rc1 user3)
    (SetRole rc2 user3)
  ) // Configure

  ( Run
    (NewCase case1 pc1 "anhumas.dcc.unicamp.br " 0)
    (NewCase case2 pc2 "iguacu.dcc.unicamp.br " 30000)
  ) // Run

)
```

No ambiente descrito acima, determinamos a criação de 2 coordenadores de processos no nó anhumas, cada um seguindo um plano diferente, respectivamente *teste.pls* e *teste2.pls* (arquivos contendo a descrição do plano em PLISP); São declarados ainda dois coordenadores de papéis diferentes nos nós anhumas e tigre.

TaskLists são criados para cada usuário do bloco de declarações nos nós especificados. Usuários podem ser registrados em mais de um servidor de papéis, como é o caso do user3.

O exemplo determina a criação de dois casos, coordenados por PC1 e PC2, nos nós *anhumas* e *iguacu* respectivamente. Antes da criação do case2, o ambiente espera por 30 segundos (30000 ms).

Apêndice B

Tabela de máquinas utilizadas durante os testes da arquitetura WONDER:

<table>
<tr>
<td>Host Name : **araguaia**
Manufacturer : Sun (Sun Microsys-
tems)
System Model : Ultra 2
Main Memory : 380 MB
Virtual Memory : 526 MB</td>
<td>Host Name : **iguacu**
Manufacturer : Sun (Sun Microsys-
tems)
System Model : Ultra 2
Main Memory : 252 MB
Virtual Memory : 531 MB</td>
</tr>
<tr>
<td>Host Name : **tuiuiu**
Manufacturer : Sun (Sun Microsys-
tems)
System Model : SPARCstation 4
Main Memory : 64 MB
Virtual Memory : 110 MB
ROM Version : 2.24
CPU Type : sparc
Number of CPUs : 1
App Architecture : sparc
Kernel Architecture : sun4m
OS Name : SunOS
OS Version : 5.5
Kernel Version : SunOS Release 5.5
Version Generic_103093-27 [UNIX(R) Sys-
tem V Release 4.0]</td>
<td>Host Name : **gaivota**
Manufacturer : Sun (Sun Microsys-
tems)
System Model : SPARCstation 4
Main Memory : 64 MB
Virtual Memory : 102 MB
ROM Version : 2.24
CPU Type : sparc
Number of CPUs : 1
App Architecture : sparc
Kernel Architecture : sun4m
OS Name : SunOS
OS Version : 5.5
Kernel Version : SunOS Release 5.5
Version Generic_103093-27 [UNIX(R) Sys-
tem V Release 4.0]</td>
</tr>
<tr>
<td>Host Name : **anhumas**
Manufacturer : Sun (Sun Microsys-
tems)
System Model : Ultra 4
Main Memory : 4096 MB
Virtual Memory : 5200 MB
CPU Type : sparc
Number of CPUs : 4
App Architecture : sparc
Kernel Architecture : sun4u
OS Name : SunOS
OS Version : 5.7
Kernel Version : SunOS Release 5.7
Version Generic_106541-08 [UNIX(R) Sys-
tem V Release 4.0]</td>
<td>Host Name : **tigre**
Manufacturer : Sun (Sun Microsys-
tems)
System Model : Ultra Enterprise
Main Memory : 512 MB
Virtual Memory : 870 MB
CPU Type : sparc
Number of CPUs : 2
App Architecture : sparc
Kernel Architecture : sun4u
OS Name : SunOS
OS Version : 5.7
Kernel Version : SunOS Release 5.7
Version Generic_106541-08 [UNIX(R) Sys-
tem V Release 4.0]</td>
</tr>
</table>

Tabela 22: Máquinas utilizadas durante os testes da arquitetura WONDER.

Apêndice C

Descrição em IDL dos servidores da arquitetura WONDER:

```
module WONDER {

	/**
	 * Types of confirmation received by coordinators
	 */
	enum ConfirmationType { conf_ok, conf_cancel, conf_other };

	/**
	 * Allowed execution states for a Workflow oject
	 * Associated numbers: 0 1 2 3 4 5 ...
	 */
	enum WorkflowObjectState { wfo_starting, wfo_running, wfo_migrating,
	 wfo_finalized, wfo_stopped, wfo_reseting };

	/**
	 * Events received by coordinators
	 * Associated numbers: 0 1 2 3 4 5 6 7 8 9 ...
	 */
	enum EventType { evn_configuring, evn_starting, evn_running, evn_spliting,
evn_joing, evn_sleeping, evn_sequencing, evn_migrating, evn_finished, evn_archived,
evn_endofplan, evn_executing, evn_finishexec, evn_saving, evn_garbage_collecting,
evn_garbage_collected, evn_negociating, evn_negociated, evn_creating, evn_created,
evn_configured};

	/**
	 * External signal received by synchronization activities
	 */
	enum SignalType { sig_ready };

	/**
	 * Type of a synchronization activity
	 */
	enum SyncActivityType { and, or, xor };

	/**
	 * Type of the user: human or process
	 */
	enum UserType { human, process };

	/**
	 * Type of the data exchanged
	 */
	enum DataType { textfile, binaryfile };

	/**
	 * Array of bytes
	 */
	typedef sequence <octet> ByteArray;

	/**
	 * List of Workflow objects names
	 */
	typedef sequence <string> NamesList;
```

```
/**
 * Stores information about one data element
 */
struct Data {
      ByteArray data;  // Data stream
      string idName;   // Id associated with the data (declared in the plan)
      //string pathname; // Directory name of the data
      string filename; // Name of the file
};

/**
 * List of data srtructures exchanged and stored by servers
 */
typedef sequence <Data> DataList;

/**
 * Stores information about a link to a data element
 */
struct Link {
      string address; // Full address of the link
      string idName;  // Id name associated to the link (declared in the plan)
      string type;  // Type of the file which the link reffers to.
      //DataType type;  // Type of the file which the link reffers to.
};

/**
 * List of links to Data location exchanged by servers
 */
typedef sequence <Link> LinksList;

/**
 * List of roles stored in a role coordinator
 */
typedef sequence <string> RolesList;

interface TaskList; // forward declaration

/**
 * User data used by the application
 */
struct User {

      UserType type;         // Type of the user
      string taskListName;   // Name of the user tasklist
      string name;           // User name
      string host;           // User preferential host
      string homedir;        // User homedir in the distriuted system
      string profile;        // Full pathname of user profile

};

/*
* Execution log of one activity. Data sent to the visualization application.
*/
struct ExecutionLogInformation {
      string activityName;
      string hostname;
      string executionStatus;
      long startTime;
      long finishTime;
```

```
        User user;
};

/**
 * Stores the case data exchanged by activities
 */
struct DataContainer {

        DataList data;        // List of data
        LinksList links;      // Links list
        ByteArray caseState;  // Set of calsses, including process definition

};

/**
 * Unit of data stored by the reposiory objects
 */
struct RepositoryData {

        DataContainer data;
        string name;

};

/**
 * List of user data, usually answear of queries.
 */
typedef sequence <User> UserList;


/** ----------------------------- Interfaces ----------------------------- */

/**
 * General WONDER interface for objects.
 */
interface WONDERObject {

        attribute string WONDERObjectName; // Name (marker) of the object
        oneway void delete(); // Deletes the current object.
};

/**
 * General control interface for objects.
 */
interface WorkflowObject : WONDERObject {

        attribute WorkflowObjectState state; // Current object state

        oneway void init();     // Starts object after a pause
        oneway void pause();    // Pauses object execution
        oneway void restart();  // Continues a paused object
        oneway void exit();     // Finalizes object before saving and disposing
        oneway void save();     // Saves (serializes) the current object state
        oneway void reset();    // Restarts object from the begining
};

/**
 * Defines the role coordinator interface used during activity sequencing
 * Each role coordinator is responsible for one role
 */
interface RoleCoordinator: WorkflowObject {
```

```
        attribute string role;  // Role associated to the current server

        boolean registerUser (in User user, in string role);
        boolean unregisterUser (in User user, in string role);
        void listUsers (out UserList userlist);
        void listRoles (out RolesList roles);

        oneway void addRole (in string role);
        oneway void removeRole (in string role);

        /**
         * Returns user (or users) associated to the current activity
         */
        boolean queryUsers( in string role_query, out UserList users);
};


/**
 * General coordinator interface
 */
interface WorkflowCoordinator: WorkflowObject {

        attribute ByteArray processDef; // Process definition stream

        // Metodo possivelmente desnecessario
        oneway void setConfirmation
            ( in ConfirmationType confirmation, in string objectname) ;

        /**
         * Receives events comming from servers under coordinator responsibility
         */
        oneway void setEvent ( in EventType event, in string senderName);

        /**
         * Returns an event stored in the intern event stack
         */
        void getLastEvent( out EventType event, out string objectname );

        oneway void registerRoleCoordinator (in RoleCoordinator roleCoord);
        oneway void unregisterRoleCoordinator (in RoleCoordinator roleCoord);

        /**
        * Returns a list of management property pairs: name, value
        */
        PropertySequence getPropertyList();

};

// Forward interface definition
interface CaseCoordinator;

interface Activity: WorkflowObject {

        attribute CaseCoordinator caseCoordinator;

        /**
         * Returs a specified data in a container. Used when converting
         * links to data.
         * @param idNames a list of idNames of the data to be retrieved.
         */
```

```
        DataList getData ( in NamesList idNames );

        /**
         * Sends a data container to the activity manager
         */
        void setData ( in DataContainer data, in string previousActivityName );

        /**
        * Returns a list of management property pairs: name, value
        */
        PropertySequence getPropertyList();

        /**
        * Removes all locally stored data and return its contents in a container
        */
        DataContainer collectAllData(in boolean remove);

        /**
        * Removes the current activity with all its data and loa state.
        */
        void garbageCollect();
};


/**
 * Returns user (or users) associated to the current activity
 */
interface Repository: WorkflowObject {

        /**
         * Data is stores and receives an index string
         */
        boolean newData (in RepositoryData data, out string dataindex);
        void getData (in string dataindex, out RepositoryData data);
        boolean removeData (in string dataindex);

};

/**
 * Defines the process coordinator interface. It is a Case coordinator factory
 * and manager.
 */
interface ProcessCoordinator: WorkflowCoordinator {

        void listActiveCaseCoordinators (out NamesList caseCoordinatorNames);

        /**
      * Sends a ByteArray with initial state: PlanInterpreter and RoleResolver.
      */
        void setInitialState ( in ByteArray caseState );
        CaseCoordinator createNewCase (in string hostName, in string caseName);
        CaseCoordinator getCaseCoordinator (in string caseCoordinatorName);
};


/**
 * Defines the case coordinator interface. It is an Activity factory
 * and manager. However, it only creates the first activiy manager
 * and synchronization activities.
 */
interface CaseCoordinator: WorkflowCoordinator {
```

```
        attribute ProcessCoordinator processCoordinator;

        void getGraph(out ProcessGraph graph);

        /**
         * Sends a data container to the case coordinator.
         * used by the ProcessCoordinator
         */
        void setData(in DataContainer data, in string processCoordinatorName );

        /**
         * Returs a specified data in a container
         * @param idNames a list of idNames of the data to be retrieved.
         */
        DataList getData (in NamesList idNames);

        void listActiveActivities (out NamesList names);

        /**
        * Explicitely starts the garage collection process
        * when Activity data are removed from hosts.
        */
        void garbageCollect();

        /**
        * Returns a reference to an activity managed by the coordinator
        * null if it is not found
        */
        Activity getActivity (in string activityName);
};

 /**
  * Defines the activity that implements the workflow conversion
  * between different WFMSs.
  */
interface GatewayActivity: Activity {

        attribute string conversor; // Conversor path location
};

/**
 * Defines the synchonization activity interface
 */
interface SynchronizationActivity: Activity {

        attribute SyncActivityType type;
        attribute string logicalExpression; // Associated logical expression
        attribute long fanin; // Number of insident activities

        /**
         * Receives signals from exter applications
         */
         oneway void externalSignal( in SignalType signal );

        /**
         * Sends a synchronization signal
         */
         oneway void synchronize ( in string previousActivityName );

};
```

```
/**
 * Defines the interface of the mobile object that implements the
 * workflow engine and wrapper activities
 */
interface ActivityManager: Activity {

      /**
       * Returns user (or users) associated to the current activity
       */
      User getUser();

      /**
       * Returns a reference to the associated user task list
       */
      TaskList getTaskList();

      /**
      * Used by wrappers to send a finish notification to the ActivityManager
      * @param wrapperName name of the wrapperServer (marker)
      */
      oneway void finishApplicationNotification(in string wrapperName);
};

/**
 * Defines the interface of the runtime objects and datatabase frontend
 */
interface BackupServer: Repository {

      boolean newObject (in WorkflowObject object);
      void readObject (in string objectname, out WorkflowObject object);
      boolean removeObject (in string objectname);
};

/**
 * Defines the interface of the already finished workflows database frontend
 */
interface HistoryServer: Repository {

      boolean queryData(in string query, out any data);
};

/**
 * Callback interface of the user's GUI.
 */
interface TaskList : WONDERObject {

      attribute User user; // Associated user

      oneway void addActivity (in ActivityManager activity,
             in string activityName, in string description);
      boolean suggestActivity (in string activityname, in string description);
      boolean removeActivity (in string activityname);

      // Informs the tasklist that the activty finished.
      oneway void finishNotification(in string activityname);

      // These methods may be private (????)
      oneway void finishActivity(in string activityname);
      oneway void startActivity(in string activityname);
      NamesList getActivitiesList();
```

```
        };

        /**
        * Interface that executes the invoked application in a host
        */
        interface Wrapper : WONDERObject {
                /**
                * @param applicationName name of the application as declared in the plan
                * @param commandLine full pathname of the application to be executed
                * @param activityName name of the invoking application - for callback.
                */
                oneway void runApplication(in string commandLine,
                        in string activityName);

        };

        /**
         * Interface used to create new activities. One activity cannot create
         * another activity of the same type without the use of a Factory.
         */
        interface WorkflowObjectFactory  {

                ActivityManager createActivityManager
                        ( in string host, in string basename );
                SynchronizationActivity createSynchronizationActivity
                        ( in string host, in string basename );
                GatewayActivity createGatewayActivity
                        ( in string host, in string basename );
                CaseCoordinator createCaseCoordinator
                        ( in string host, in string basename );
        };
};
```

Apêndice D

Lista dos erros mais comuns encontrados durante a implementação do protótipo WONDER juntamente com suas causas.

ERRO: Parada anormal na janela do *orbixd*. (UNKNOWN ERROR):
CAUSA: invocação de métodos em uma variável cujo valor era *null* no cliente.
ERRO: *Null pointer exception* durante a invocação de métodos remotos passando *sequences*
CAUSA: passagem de *sequences* não inicializados: variáveis do tipo *sequence* passadas como *null*.
SOLUÇÃO: Resolvemos o problema iniciando o *sequence* com tamanho zero.

ERRO: Ao darmos um *bind* para um servidor de uma determinada classe, dentro desta mesma classe, por exemplo, um servidor do tipo X tenta criar outro servidor do tipo X, o OrbixWeb retorna uma referência para o servidor que invocou o *bind* (*self-reference*). Deveria criar um novo servidor e retornar uma referência para este objeto.
CAUSA: Desconhecida
SOLUÇÃO: Criar um FACTORY (Um outro servidor registrado no OrbixWeb, de um tipo diferente) contendo os *factory methods*: operações que criam os servidores desejados, devolvendo referências para estes objetos. Criar o servidor a partir de invocações neste *factory*.

ERRO: org.omg.CORBA.UNKNOWN: remote exception - Unknown error. The server encountered a Java exception while dispatching the request - (unknown)
CAUSA: Passagem de uma referência a um servidor CORBA como *null*, seguido de uma posterior tentativa de invocação de métodos usando esta referência, no servidor que a obteve.
SOLUÇÃO: Checar se a referência é ou não nula antes de usa-la.

ERRO: org.omg.CORBA.UNKNOWN: remote exception - Unknown error. The server encountered a Java exception while dispatching the request - (unknown)
CAUSA: Tentativa de invocação de método em uma referência *null* no lado servidor
SOLUÇÃO: Fazer checagem de referências nulas antes de as utilizar.

ERRO: org.omg.CORBA.UNKNOWN: remote exception – Unknown error The server encountered a Java exception while dispatching the request - (unknown)
CAUSA: Passagem de um *struct* contendo campos *null*: ou o *struct* era *null* ou seus campos eram.
SOLUÇÃO: Preencher as variáveis do *struct* com instâncias vazias exemplo: "" para *string*, byte[0] para *arrays*, etc.

ERRO: Variable not initialized OR null variable
CAUSA: Re-declaração de uma mesma variável existente na classe pai. Desta forma, métodos da classe pai que, inicializam esta variável, não possuem efeito pois, na classe filha, a variável está inicializada como *null* (padrão do Java).

SOLUÇÃO: Excluir a declaração da variável da classe derivada de forma a não sobrescrever (*overrride*) a declaração da super classe.

ERRO: org.omg.CORBA.INV_OBJREF: remote exception – Invalid object reference
CAUSA: Tentativa de dar um *bind* num objeto de um tipo X usando um HELPER de um tipo Y.

ERRO: org.omg.CORBA.INV_OBJREF: remote exception – Invalid object reference
CAUSA: Tentativa de dar um *bind* num objeto do tipo X usando um *marker* qualquer, sem que o *marker* seja fornecido pelo *Loader.class*. O *marker* precisa ser consenso tanto do lado cliente quanto do lado servidor. Precisamos usar o *Loader*, registrando-o quando o objeto é criado.

ERRO: org.omg.CORBA.UNKNOWN: remote exception – Unknown error The server encountered a Java exception while dispatching the request - (unknown)
CAUSA: Erro durante a passagem de um tipo estruturado, contendo um dos atributos como sendo um tipo enumerado (*enum*), para o servidor.
SOLUÇÃO: Não usar o tipo enumerado, substituir por um *string*.

ERRO: Binder: Exception during binding to WorkflowManager at host: WONDER org.omg.CORBA.INV_OBJREF: remote exception - Invalid object reference
CAUSA: uso de Binder.bindDefaultWorkflowManager ao invés de Binder.bindLocalWorkflowManager. A mesma coisa vale para o Binder.WONDERObjectFactory.

ERRO: Binder: Exception during binding to WorkflowObjectFactory org.omg.CORBA.INITIALIZE: Invalid operation OrbixWeb orb is not initialised, check ORB.init call minor code: 12256 completed: No
CAUSA: Tentativa de utilização do ORB do OrbixWeb sem que esta esteja devidamente inicializada OU uso do ORB do JDK1.2 ao invés do OrbixWeb.
SOLUÇÃO: Inicializar o ORB ou rodar o programa usando JDK1.1 (através do *wrapper* 'owjava')

ERRO: Binder: Exception during binding to ActivityManager. Org.omg.CORBA.COMM_FAILURE: WONDER.pos.ic.unicamp.br/1570
CAUSA: Tentativa de criação de um servidor *unshared* usando um *marker* com comprimento maior que 59 caracteres na máquina WONDER
SOLUÇÃO: Usar nomes menores. O comprimento destes nomes parede depender da plataforma (Solaris/NT), de acordo com a versão do OrbixWeb.

ERRO: LOA: Error while serializing to file: ./WONDERData/Loa/WONDER,P,seqPC.loa Java.io.NotSerializableException: WONDER._CaseCoordinatorStub
CAUSA: Tentativa de serializar um objeto contendo um atributo não serializavel. No caso, uma referência a um objeto CORBA. Referências a objetos CORBA não são serializáveis já que os objetos não são locais.

SOLUÇÃO: Tornar o atributo do tipo servidor CORBA *transient* e armazenar, como alternativa, o nome do objeto, não a sua referência. Usar o nome do objeto para fazer um *bind* posterior, após o objeto ser lido da base de dados serializada.

ERRO: Servidor trava durante a invocação de um método remoto. O programa pára mas o *daemon* continua respondendo a invocações do comando 'psit' por exemplo.
CAUSA: A variável usada para invocar o método não foi devidamente inicializada ou aponta para outro servidor.
SOLUÇÃO: Iniciar a variável corretamente, possivelmente usando *bind* e obtendo a referência desejada antes de invocar o método.

ERRO: DataResolver: WONDER,M,seqPC,seqCC-0,act03-1: Error while getting data !
org.omg.CORBA.UNKNOWN: remote exception - Unknown error
The server encountered a Java exception while dispatching the request - (unknown)
CAUSA: Envio de campos *null* dentro de um *struct*
SOLUÇÃO: Structs são mapeadas para Classes Java, contendo seus respectivos atributos, pelo compilador IDL. Enviar objetos vazios ou *strings* de tamanho zero, mas não *null*.

ERRO: org.omg.CORBA.COMM_FAILURE do lado cliente.
CAUSA: provocado por excesso de carga na máquina: falta de recursos de memória ou swap devido ao grande número de servidores carregados em paralelo.
SOLUÇÃO: Tratar a exceção e tentar a conexão, novamente, até conseguir usando o trecho de código semelhante ao descrito a seguir:

```
int retryTurn = 0;
do {

      try {

         // Não somente este mas para todos os servidores do WONDER
         server = LockManagerHelper.bind(":LockManagerSrv");
      } catch (SystemException ex) {
      System.out.println(ex.toString());
      server = null;
      retryTurn ++;
      wait(waitTimeout);
   }

} while (doRetry && (server == null) && (retryTurn < retryTimes));
```

ERRO: org.omg.CORBA.INTERNAL: remote exception - ORB internal error:

```
Internal Error in activator
Actual system exception is '(unknown)'
(please contact Iona Technologies)
```

CAUSA: O aplicativo utiliza excesso de CPU ou recursos da máquina o que a impossibilita de atender requisições ou criar novos objetos.