UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS

Tatiana Pereira Filgueiras

RT-JADE: *MIDDLEWARE* COM SUPORTE PARA ESCALONAMENTO DE AGENTES MÓVEIS EM TEMPO REAL

Dissertação submetida ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas da Universidade Federal de Santa Catarina para a obtenção do Grau de Mestrado em Engenharia de Automação e Sistemas
Orientador: Prof. Dr. Lau Cheuk Lung

Florianópolis
2011

Catalogação na fonte pela Biblioteca Universitária da Universidade Federal de Santa Catarina

F481r Filgueiras, Tatiana Pereira
RT-JADE [dissertação] : Middleware com suporte para escalonamento de agentes móveis em tempo real / Tatiana Pereira Filgueiras ; orientador, Lau Cheuk Lung. – Florianópolis, SC, 2011.
121 p.: il., grafs., tabs.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Engenharia de Automação e Sistemas.

Inclui referências

1. Engenharia de sistemas. 2. Agentes móveis (software). 3. Sistemas de reconhecimento de padrões. 4. Escalonamento. 5. Java (Linguagem de programação de computador). I. Lung, Lau Cheuk. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Automação e Sistemas. III. Título.

CDU 621.3-231.2(021)

Tatiana Pereira Filgueiras

RT-JADE: *MIDDLEWARE* COM SUPORTE PARA ESCALONAMENTO DE AGENTES MÓVEIS EM TEMPO REAL

Esta Dissertação foi julgada adequada para obtenção do Título de "Mestre" e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia de Automação e Sistemas

Florianópolis, 23 de maio de 2011.

_____
Prof. José Eduardo Ribeiro Cury, Dr.
Coordenador do Curso

Banca Examinadora:

_____
Prof. Lau Cheuk Lung, Dr.
Orientador
Universidade Federal de Santa Catarina

_____
Profª. Luciana de Oliveira Rech, Dra.
Co-Orientador
Universidade Federal de Santa Catarina

_____
Prof. Carlos Barros Montez, Dr.
Universidade Federal de Santa Catarina

_____
Prof. Ricardo Alexandre Moraes, Dr.
Universidade Federal de Santa Catarina

_____
Prof. Rômulo Silva de Oliveira, Dr.
Universidade Federal de Santa Catarina

Dedico este trabalho a Jeová Deus e seu filho Jesus Cristo, por terem dado não somente a mim, mas a toda a humanidade, a maravilhosa esperança que me impulsiona a continuar seguindo em frente.

AGRADECIMENTOS

Agradeço, primeiramente, a Jeová Deus, pelas bênçãos da vida e capacidade intelectual, que me permitiram concluir este trabalho; além de muitas outras que, se fossem escritas, a própria Terra não as comportaria.

À minha amada mãe, Florice, que sempre me impulsionou a continuar seguindo em frente, mesmo sob pressões e desgaste físico e emocional, e por ter praticamente feito o mestrado comigo, sofrendo com meu sofrimento e se alegrando com minha alegria.

A meu pai, Antonio, pela ajuda financeira e moradia.

A minha irmã, Jeane, por sempre dizer ter orgulho de mim, mesmo sem eu saber por quê.

Ao meu orientador, Lau Cheuk Lung, e à professora Luciana de Oliveira Rech, pelo apoio dado durante o trabalho e também pela amizade, carinho e principalmente paciência demonstrados.

Ao professor Eugênio, pela oportunidade de cursar o mestrado e por toda a ajuda provida durante o curso.

Aos professores Carlos Barros Montez e Jomi Hubner, pela instrução no âmbito prático da docência.

A todos os professores do DAS, pela instrução que me capacitou a desenvolver este trabalho.

Ao CAPES, pelo apoio financeiro à pesquisa.

Enfim, agradeço a todos os envolvidos, quer direta, quer indiretamente, que contribuíram para que eu pudesse alcançar mais este estágio em minha vida.

A todos, muito obrigada!

"Somente um principiante que não sabe nada sobre ciência diria que a ciência descarta a fé. Se você realmente estudar a ciência, ela certamente o levará para mais perto de Deus."

(James Tour, data desconhecida)

RESUMO

Com o crescimento das redes de computadores mundiais e o aumento do uso de aplicações distribuídas, surge um aumento significativo no tráfego de dados no enlace, demonstrando, assim, a necessidade de aprimoramento – ou substituição das atuais técnicas de transmissão de dados – do modelo tradicional cliente-servidor. A tecnologia de agentes móveis tem sido alvo de diversas pesquisas na área, por permitir que apenas um código mova-se entre os nós da rede e retorne com os resultados, diminuindo, assim, a carga na rede. Para que um agente móvel possa cumprir sua missão, é necessário que se atenda a um *deadline*. Entretanto, em um sistema distribuído, há a possibilidade de concorrência por um mesmo recurso. Tratar de forma adequada tal concorrência é de suma importância, especialmente em um ambiente em tempo real. Em face disso, um modelo de execução em que agentes móveis disputam um mesmo recurso em um mesmo *host* é proposto, o RT-JADE: uma extensão de *middleware* que possibilita agentes móveis concorrentes cumprirem suas missões, usando métodos de escalonamento em tempo real sobre a plataforma JADE. A eficácia do modelo proposto foi demonstrada através de simulações e comparações entre diversas políticas de escalonamento, com distintas cargas de trabalho (quantidade de agentes concorrentes).

Palavras-chave: Agente Móveis. Concorrência. Tempo Real. JADE.

ABSTRACT

With the growth of worldwide computer networks and increased use of distributed applications, there is a significant increase in data traffic in the link, thus demonstrating the need for improvement – or replacement of existing techniques for data transmission – of the traditional client-server one. The mobile agent technology has been the subject of several researches in area, because it only allows a code to move between network nodes and return with the results, reducing the network's load. For a mobile agent to accomplish its mission, it is necessary for it to meet a deadline. However, in a distributed system there is the possibility of competition for the same resource. Treating such competition adequately is very important, especially in a real-time environment. In this view, an execution model in which mobile agents compete for the same resource in the same host is proposed – RT-JADE: a middleware extension that allows concurrent mobile agents to achieve their missions, using real-time scheduling methods on the JADE platform. We demonstrate the effectiveness of the proposed model through simulation and comparison of different scheduling policies under different workloads (number of competing agents).

Keywords: Mobile Agents. Concurrency. Real Time. JADE.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACL	Agent Comunication Language
AID	Agent Identification
AMS	Agent Management System
BA	Broker Agent
BDI	Belief-Desire-Intention
BMA	Broker level Mobile Agent
CMIP	Common Management Information Protocol
CORBA	Common Object Request Broker Architecture
DB	DataBase/Banco de dados
DM	Deadline Monotonic
DPS	Distributed Problem Solving
EDF	Earliest Deadline First
FIFO	First In, First Out
FIPA	Foundation for Intelligent Physical Agents
IA	Inteligência Artificial
IAD	Inteligência Artificial Distribuída
ID	Identificador
JADE	Java Agent Development Framework
JSP	JavaServer Pages
LAN	Local Area Network/Rede local
LIFO	Last In, First Out
MA	Mobile Agent/Agente móvel
MAS	Multi Agent System/Sistema multiagente
MBS	Mobile Behaviour Scheme
ORB	Object Request Broker
PDA	Personal Digital Assistant
PDM	Preemptive Deadline Monotonic
PEDF	Preemptive EDF
PPRIO	Preemptive Priority Based Scheduler
PRIO	Priority-based scheduler
REV	Remote Evaluation
RM	Rate Monotonic
RPC	Remote Procedure Call
RR	Round Robin
RTT	Round-Trip Time
SA	Server Agent

SJF	Shortest Job First
SM	Smart Message
SMA	Supplier level Mobile Agent
SNMP	Simple Network Management Protocol
SRTF	Shortest Remaining Time First
STR	Sistema de Tempo Real
TO	Task Objective
UCP	Unidade Central de Processamento
UI	User Interface
WAN	Wide Area Network/Rede global

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Atualmente, as redes de computadores têm sido criadas para suportar uma série de aplicações distribuídas, podendo fazer com que o modelo tradicional cliente-servidor torne-se inadequado, devido à problemática do congestionamento de informações na rede (DOTTI; DUARTE, 2001). Para solucionar este problema, faz-se necessária a criação de um novo modelo que atenda às mesmas premissas compreendidas pelo modelo tradicional, porém minimizando o tráfego de dados.

A mobilidade de código é uma área emergente de pesquisa, que pode ser considerada uma solução alternativa para o projeto e implementação de sistemas distribuídos. Entre os benefícios que essa mobilidade traz em sistemas distribuídos, estão: alto grau de flexibilidade, escalabilidade e customização; melhor utilização da infraestrutura de comunicação; e provisão de serviços de maneira autônoma, sem a necessidade de conexão permanente (FOU, 2001). Além disso, alguns domínios de aplicação em que tem se mostrado promissora são: acesso e gerenciamento de informações, comércio eletrônico, controle de processos, gerenciamento de sistemas, entre outros (SALIM; JAVED; AKBAR, 2010).

No contexto de mobilidade de código, uma importante área de pesquisa é a tecnologia de agentes móveis (SHEMSHADI; SOROOR; TAROKH, 2008; BAEK; KIM; YEOM, 2002; SAHINGOZ; ERDOGAN, 2004; ARUNACHALAN; LIGHT, 2008). Um agente móvel (*Mobile Agent* – MA) é um componente de *software* independente e autocontido, que encapsula protocolos que facilitam a interoperabilidade entre plataformas, além de ter sua execução de forma assíncrona e autônoma, sendo, assim, capaz de executar tarefas.

Um MA tem também como vantagem a total independência do sistema em que iniciou a sua execução, ou seja, é capaz de migrar de forma autônoma através dos nodos de um sistema distribuído, continuando sua execução, mantendo seu estado e gerando ou coletando resultados e, com isso, tornando-se um importante representante do conceito da mobilidade de código. Ainda, por migrar para a localização do recurso desejado, um MA pode responder a estímulos rapidamente e continuar sua interação com o recurso se a conexão de rede cair temporariamente. Essas características fazem com que os MAs tornem-se atrativos para o desenvolvimento de

aplicações móveis, as quais frequentemente devem tratar com baixa largura de banda, alta latência e enlaces de rede não confiáveis (PICCO; FUGGETA; VIGNA, 1998).

Outro fator que tem se tornado um desafio na adequação de um novo modelo para aplicações distribuídas é a questão temporal. Sistemas de Tempo Real (STRs) são aqueles que possuem a capacidade de acompanhar o estado de um dado processo físico e atuar "a tempo" sobre ele, ou seja, dentro de um prazo específico (*deadline*).

Quando se trata do uso de MAs em STR, pesquisas recentes têm sido realizadas em diversas áreas, tais como: cuidados médicos (ARUNACHALAN; LIGHT, 2008), *e-commerce* (SAHINGOZ; ERDOGAN, 2004), manufatura (WANG et al., 2006) e sistemas distribuídos (CAO et al., 2002). Além disso, com o objetivo de facilitar o desenvolvimento de sistemas distribuídos utilizando MAs, vários grupos de pesquisa desenvolveram plataformas de *middleware* específicas para esse fim (CHEN; CHENG; PALEN, 2006; WIERLEMANN; KASSING; HARMER, 2002).

Os primeiros *middlewares* para MA foram desenvolvidos para redes locais (*Local Area Network* – LAN) e globais (*Wide Area Network* – WAN), como Grasshopper (BÄUMER; MAGEDANZ, 1999), Aglets (LANGE et al., 1997) e *Java Agent Development Framework* (JADE) (BELLIFEMINE; CAIRE; GREENWOOD, 2007). Após isso, o interesse em dispositivos móveis aumentou, gerando a necessidade de propostas de *middlewares* para redes de sensores sem fio (FOK; ROMAN; LU, 2005) e dispositivos embarcados (CHEN; CHENG; PALEN, 2006), esses últimos projetados para dar suporte à mobilidade e comunicação entre MAs em redes sem fio, considerando as limitações técnicas (bateria, processamento, memória etc.) desses dispositivos.

Apesar de a utilização do modelo de agentes móveis em STR já ser amplamente difundida e pesquisada, o uso de políticas de escalonamento para MAs em aplicações de tempo real ainda não foi abordado na literatura; dessa forma, não existem propostas de *middleware* que dê suporte ao escalonamento local de MAs.

Assim como acontece com processos, quando um determinado recurso é requisitado simultaneamente por vários MAs, soluções para tratar este problema de forma justa são necessárias, porém não há estudos que indiquem a preocupação com tal acontecimento ou o levantamento de uma abordagem que permita o escalonamento em tempo real para MAs concorrentes. Em aplicações em tempo real, os MAs devem executar suas tarefas de acordo com seus *deadlines* e, baseado nisso, é necessário atribuir priori-

dades para a utilização dos recursos, permitindo, dessa forma, aos agentes mais prioritários utilizar os recursos mais rapidamente.

A busca por soluções para desenvolver um sistema de escalonamento de MAs é importante para aplicações que necessitam cumprir *deadline*, pois tais agentes são normalmente assíncronos em relação à chegada e saída dos *hosts* devido à sua característica autônoma, o que faz com que o sistema não tenha um controle prévio sobre prioridades, alocando as entradas e saídas dos agentes somente em política *First In, First Out* (FIFO) (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

Um exemplo de possível cenário é uma oficina de produção, onde todos os recursos de manufatura de peças são frequentemente representados como MA, enquanto outros recursos de produção, como máquinas, instrumentos e ferramentas de corte, são representados como agentes estáticos (WANG et al., 2006). A função de cada agente depende de sua colocação dentro da produção em chão de fábrica, podendo negociar, ofertar ou votar em contratos de diversas áreas, porém, se houver prazos ou ordem de prioridade, o agente precisa atender a essas restrições temporais ou poderá prejudicar toda a linha de produção.

1.2 OBJETIVOS DA DISSERTAÇÃO

O objetivo geral desta dissertação é propor a arquitetura RT-JADE, que tem como função unir a mobilidade de código com a questão temporal, tratando a questão de escalonamento de MA com restrições temporais, visando à questão de concorrência entre estes sobre os recursos de um *host*. Esta dissertação apresenta uma proposta de extensão da plataforma para MA JADE, para dar suporte ao escalonamento de MA. Nesse sentido, as extensões de *middleware* propostas comportam diferentes tipos de políticas de escalonamento, para diferentes necessidades de aplicações em tempo real usando MA.

Visando a atender a este objetivo geral, os seguintes objetivos específicos foram definidos:

- pesquisar na literatura o estado da arte de MAs em STR;
- apresentar um estudo do comportamento de MAs concorrentes com restrições temporais utilizando a política FIFO, empregada pelo JADE;

- desenvolver uma arquitetura que dê suporte para diferentes políticas de escalonamento de MAs em um *host* com múltiplos recursos;
- propor e implementar algoritmos de escalonamento (preemptivos e não preemptivos) eficientes, que permitam ao maior número possível de MAs concluir suas missões, respeitando seus *deadlines*;
- avaliar o comportamento das políticas de escalonamento empregadas nos algoritmos elaborados considerando diferentes faixas de *deadline*, comparando-os com a política FIFO do JADE.

1.3 METODOLOGIA

1.3.1 Classificação da Pesquisa

Com base na classificação proposta por Silva e Menezes (2005), esta dissertação enquadra-se da seguinte forma.

1.3.1.1 Com Base nos Objetivos

Exploratória e explicativa, pois, com o objetivo de aprimorar ideias, foram realizadas uma pesquisa bibliográfica e a análise de exemplos, apresentando ao final características explicativas.

1.3.1.2 Com Base nos Procedimentos Técnicos

Bibliográfica e experimental, pois foi elaborada inicialmente com base em material já existente (artigos e livros) e avaliada experimentalmente através de simulações.

1.3.1.3 Com Base na Natureza da Pesquisa

Aplicada, uma vez que a pesquisa gera conhecimentos com finalidades imediatas, como, por exemplo, o escalonamento de agentes móveis em um sistema de manufatura.

1.3.1.4 Com Base na Abordagem do Problema

Quantitativa, visto que se utilizaram os parâmetros estatísticos das simulações para análise e qualificação do modelo proposto.

1.3.2 Roteiro da Pesquisa

A partir das classificações listadas na seção anterior, o roteiro da pesquisa a ser apresentada nesta dissertação pode ser descrito da seguinte forma:

- revisão bibliográfica da literatura relacionada;
- estudo das propostas que envolvem aspectos relacionados aos objetivos desta dissertação;
- proposição, implementação e avaliação (por meio de simulações) da arquitetura proposta;
- escrita da dissertação, com base nos dados obtidos.

1.4 ORGANIZAÇÃO DO TRABALHO

Este texto está dividido em sete capítulos, sendo que este primeiro capítulo descreveu o contexto geral ao qual a dissertação está inserida.

O capítulo 2 introduz uma série de conceitos sobre sistemas de tempo real, bem como algumas políticas de escalonamento para o problema de concorrência em tempo real são descritas; ainda neste capítulo, também se discute a questão de restrição temporal.

O capítulo 3 apresenta uma caracterização de agentes inteligentes, a qual é necessária para uma melhor compreensão de todo o trabalho. No texto, são descritos: a definição e classificação de agentes inteligentes estáticos e móveis, os aspectos dos sistemas baseados em agentes, as principais ferramentas para o desenvolvimento e implementação de agentes e, por fim, as atuais aplicações dessa tecnologia.

Os trabalhos relacionados com esta pesquisa são apresentados no capítulo 4, enquanto o capítulo 5 contém a descrição da arquitetura proposta e desenvolvida, sendo essa a contribuição desta dissertação. Além disso, são apresentados os principais problemas que justificam a escolha do tema e, também, a formulação da possível solução para eles.

Já no capítulo 6, são discutidos os resultados obtidos com a simulação e avaliação da proposta, sendo que as conclusões e os trabalhos futuros encontram-se no capítulo 7.

2 SISTEMAS DE TEMPO REAL

2.1 INTRODUÇÃO

Na medida em que o uso de dispositivos móveis (*tablets*, *Personal Digital Assistants* – PDAs etc.) torna-se comum, aplicações que necessitem cumprir um determinado prazo de tempo são mais utilizadas, desde controladores inteligentes embutidos em utilidades domésticas até controladores de tráfego aéreo, de modo que a complexidade dessas aplicações varia de acordo com as necessidades de garantia de atendimento às restrições temporais (FARINES; FRAGA; OLIVEIRA, 2000).

A respeito da rigorosidade no atendimento às restrições temporais, há aplicações que exigem mais critério, como sistemas responsáveis pelo monitoramento de pacientes em hospitais, e aplicações que exigem menos rigor, como é o caso de aplicações multimídia em geral, estando ambas sujeitas a restrições temporais e incluídas, assim, na categoria de STR.

O restante do capítulo está organizado da seguinte forma: na seção 2.2, é apresentada a caracterização de um STR; a definição de tarefas e concorrência é brevemente descrita na seção 2.3; a seção 2.4 introduz o conceito de escalonamento em STR e, por fim, as conclusões encontram-se na seção 2.5.

2.2 CARACTERIZAÇÃO DE TEMPO REAL e STR

Tempo real é a não capacidade de controlar externamente o ritmo de evolução com o qual o sistema computacional interage com o ambiente, porém não significa rapidez e sim um ritmo de evolução próprio de um dado processo físico (FARINES; FRAGA; OLIVEIRA, 2000).

O ambiente impõe ao sistema requisitos temporais e este deve atuar de acordo com essa restrição, porém vale ressaltar que há situações de antagonismo, em que o ritmo de evolução pode ser controlado, como no caso de sistema de reservas de passagens aéreas (PEDREIRAS; ALMEIDA, 2005). Nessas situações, quando ocorre uma sobrecarga de solicitações, o sistema passa a operar da melhor forma possível (*best effort*), mas sem garantias quanto ao cumprimento de prazos (*deadline*), geralmente ocasionando formação de filas de espera.

Por sua vez, um STR é um sistema computacional que possui a capacidade de acompanhar o estado de um dado processo físico e atuar "a tempo" sobre ele.

Nesse contexto, os *sistemas reativos* reagem continuamente a cada estímulo originado do ambiente, sendo que, a cada reação, o STR deve efetivar a entrega de um resultado correto dentro do *deadline*. Quanto à corretude do resultado, esta depende não somente da integridade dos dados (*correctness*), mas também dos valores de tempo em que são produzidos (*timeless*) (FARINES; FRAGA; OLIVEIRA, 2000). A maior parte dos STRs é, portanto, sistemas reativos que gerenciam os recursos de um sistema computacional, com o objetivo de garantir com que todos os eventos sejam atendidos dentro de suas restrições temporais, o mais eficientemente possível. Porém, para garantir a eficiência, um STR deve não somente prover a integridade dos dados, mas também sua segurança (*security*).

Em se tratando de *safety*, os STRs classificam-se em duas categorias: críticos e não críticos. Um STR crítico precisa obrigatoriamente cumprir um *deadline*; quando o seu descumprimento excede de forma crítica o benefício do sistema, geralmente incorre em falhas catastróficas ao ambiente. Exemplos de sistemas críticos de tempo real são: sistemas de controle de planta nuclear, sistema de controle de voo, sistema de sinalização de ferrovias, entre outros.

Os STRs críticos podem ser subdivididos em (KOPETZ, 1997):

- STR crítico em caso de falha ("*fail safe*"): quando ocorre uma falha, o sistema pode evitar a catástrofe por atingir um estado seguro, como, por exemplo, a parada obrigatória de todos os trens em caso de falha na sinalização;
- STR crítico operacional em caso de falha ("*failoperation*"): quando ocorre uma falha parcial, o sistema pode degradar-se oferecendo o mínimo necessário para continuar funcionando de forma segura, como, por exemplo, um sistema de controle de voo com comportamento degradado, mas ainda seguro.

Ao contrário dos STRs críticos, o descumprimento do *deadline* no STR não crítico afeta o benefício do sistema, mas não traz falhas catastróficas ao ambiente. Exemplos de STRs não críticos são: sistemas multimídia, sistemas de comutação telefônica etc.

2.3 TAREFAS E CONCORRÊNCIA

Tarefas são unidades de execução sequencial que compartilham um ou mais recursos computacionais (SHAW, 2003). Há casos em que uma tarefa depende de dados ou resultados de outra, o que chamamos relação de precedência (AUDSLEY et al., 1993). Outra relação importante entre tarefas é a exclusão mútua, que ocorre quando duas tarefas compartilham o mesmo recurso – sem ser o processador – e uma não pode iniciar sua execução enquanto a outra estiver em andamento (COULOURUS; DOLLIMORE; KINDBERG, 2007).

Em STR, as tarefas devem não somente apresentar corretude lógica na execução, mas também ter seus requisitos temporais individuais satisfeitos. Segundo Stankovic et al. (1996), as tarefas em STR podem ser periódicas, aperiódicas ou esporádicas, sendo:

- tarefa periódica: tarefas cuja ativação dá-se a cada "*t*" unidades de tempo. O seu limite de execução para cada ativação pode ser menor, igual ou maior do que o período "*t*";
- tarefa aperiódica: tarefas cuja ativação dá-se em períodos indeterminados;
- tarefa esporádica: tarefa aperiódica com a restrição adicional de possuir um intervalo mínimo entre ativações de tarefas.

Quanto ao uso de recursos, várias tarefas podem disputar o direito de execução, de forma que os STRs devem lidar com essa disputa, denominada concorrência (SHAW, 2003). Define-se concorrência quando dois ou mais processos solicitam a utilização de um mesmo recurso, simultaneamente (BARLAND; GREINER; VARDI, 2005), sendo um elemento muito importante da programação distribuída (McHALE, 1994).

Há dois tipos de interações entre processos concorrentes (SHRIVASTAVA; BANATRE, 1978), sendo eles:

- competição: quando processos são independentes, mas têm de partilhar os recursos comuns de um computador. Por exemplo, tempo de Unidade Central de Processamento (UCP), espaço de memória e acesso a periféricos;
- cooperação: quando processos são interdependentes, fazendo parte de uma mesma aplicação ou interagindo explicitamente

uns com os outros. Por exemplo, para a solicitação de serviços entre si, para efetuar o processamento concorrente das entradas, tratamento e saída de dados ou para executar partes de um programa em paralelo, em processadores reais distintos.

Além disso, a concorrência pode ser usada para aumentar o desempenho do sistema, usando-se processadores em paralelo. Para garantir que todas as tarefas de um sistema atendam às suas restrições temporais, é necessário ordená-las da melhor forma possível; este processo de ordenação denomina-se escalonamento.

2.4 ESCALONAMENTO EM SISTEMAS DE TEMPO REAL

O escalonador é uma rotina que gerencia o compartilhamento de processadores ou sistemas distribuídos entre tarefas, dando a cada *task* em execução uma fatia de tempo do processador, priorizando determinados tipos de processos (BARLAND; GREINER; VARDI, 2005). Basicamente, o escalonador é que define qual processo deve ser executado primeiro.

Os principais objetivos de um algoritmo de escalonamento são: justiça, *policy enforcement* e equilíbrio. Entende-se por justiça que cada processo deve receber a sua parte da UCP; já *policy enforcement* é o respeito às políticas estabelecidas; por fim, equilíbrio é o que define que todas as partes do sistema devem estar operando (CHANG et al., 2010).

Ainda, o escalonador incorpora uma política de escalonamento, podendo esta ser do tipo preemptiva ou não preemptiva. Define-se como escalonamento preemptivo quando um processo em estado de pronto tem precedência sobre o que está usando a UCP, logo, um escalonamento preemptivo das tarefas provoca uma interrupção forçada – ou preempção – de um processo para que outro possa usar a UCP. Já no escalonamento não preemptivo, o escalonador não interrompe os processos em execução.

Tais escalonamentos podem adotar os seguintes critérios (CHOI et al., 2004):

- orientados a desempenho: tempo de retorno (*turnaround time*), tempo de resposta, *deadline* ou preditibilidade;
- orientados ao sistema: utilização da UCP, *throughput*, *fairness*, prioridades, balanceamento de recursos ou tempo de espera.

Um escalonamento é tido como ideal se possui máximo uso de UCP e *throughput*, e mínimo tempo de retorno, resposta e espera (COULOURUS; DOLLIMORE; KINDBERG, 2007). Por sua vez, diz-se praticável (*feasible schedule*) se cumpre as restrições associadas ao conjunto de tarefas, sejam elas temporais, não preemptivas, recursos partilhados ou precedências. Também, um conjunto de tarefas diz-se escalonável (*schedulable task set*) se existe pelo menos um escalonamento praticável para ele.

Nesse contexto, o escalonamento de tarefas periódicas é geralmente dirigido por níveis de prioridade, apresentando um bom grau de desempenho e flexibilidade (CHENG; STANKOVIC; RAMAMRITHAM, 1998), enquanto, para tarefas aperiódicas, não é possível fazer nenhuma previsão quanto a intervalos de lançamento de suas instâncias ou sobre o tempo necessário para a sua execução, cabendo aos algoritmos de escalonamento determinar em que momento deve-se executá-las.

2.4.1 Algoritmos de Escalonamento

Os algoritmos de escalonamento são comumente utilizados em qualquer sistema que contenha competição por recursos (concorrência). Para aplicações com requisitos de tempo real, um algoritmo de escalonamento pode fornecer garantia absoluta, porém, em aplicações *best effort,* não é necessário qualquer garantia, mas torna-se desejável, para que a partilha de recursos entre as aplicações seja feita de forma equitativa (*fair*).

Em geral, pretende-se que os algoritmos de escalonamento sejam simples, ou seja, fáceis de implementar; tratem de forma equitativa o tráfego *best effort*; garantam ao restante do tráfego os níveis de desempenho negociados; e facilitem os mecanismos associados de controle de admissão de conexões.

A atribuição equitativa de recursos é habitualmente baseada no critério designado por *max-min fairness*, que atribui recursos por ordem ascendente dos valores solicitados, na qual nenhum fluxo obtém uma quota de utilização superior ao solicitado e os fluxos cujos requisitos não podem ser satisfeitos são tratados de forma idêntica, ou seja, recebem a mesma quota de utilização (*fair share*) (FRACHTENBERG; SCHWIEGELSHOHN, 2010).

Cabe ressaltar que o algoritmo de escalonamento necessita decidir a ordem pela qual as tarefas devem ser servidas, em cada nível de prioridade.

2.4.1.1 Classificação

Os algoritmos de escalonamento podem ser classificados em dois tipos:

- estáticos: quando o cálculo da escala é feito tomando como base parâmetros atribuídos às tarefas do conjunto em tempo de projeto;
- dinâmicos: quando são baseados em parâmetros que mudam em tempo de execução, com a evolução do sistema.

Nas políticas de escalonamento dinâmico, as decisões de escalonamento são tomadas em tempo de execução (*on-line*), podendo haver mudanças nas prioridades dos processos ao longo de sua execução, sendo que os critérios para essas decisões variam de algoritmo para algoritmo.

2.4.1.2 Políticas de Escalonamento

Algumas políticas de escalonamento são:

- **FIFO** (PANICO, 1969; ALLEN, 1978; KRUSE, 1987): consiste em servir as tarefas por ordem de chegada, sem oferecer quaisquer garantias no que diz respeito a atrasos e/ou perdas de *deadline*;
- ***Last In, First Out* (LIFO) (**PANICO, 1969; ALLEN, 1978; LIPSCHUTZ, 1986): trata-se do oposto do FIFO, ou seja, a primeira tarefa a chegar é a última a ser atendida. Assim como o FIFO, não fornece quaisquer garantias referentes a atrasos e/ou perda de *deadline*;
- ***Rate Monotonic* (RM)** (LIU; LAYLAND, 1973): trata-se de um algoritmo ótimo para sistemas monoprocessadores, pois, se um conjunto qualquer de tarefas (periódicas e independentes) pode ser escalonado por uma política de escalonamento dinâmica com prioridades fixas e se *deadline* é igual ao período, então se diz que tal conjunto também pode ser escalonado por RM. O critério de atribuição de prioridades do RM baseia-se na ativação periódica de tais tarefas, com uma meta temporal igual ao seu período (ou seja, qualquer execução deve ser finalizada antes da próxima data de ativação da tarefa);

- ***Earliest Deadline First* (EDF)** (LIU; LAYLAND, 1973): é uma política de escalonamento dinâmico, podendo ser aplicada tanto em tarefas periódicas quanto em aperiódicas. As prioridades das tarefas são associadas de acordo com seu *deadline*; aquelas que possuem o *deadline* absoluto mais próximo recebem prioridade mais alta e vice-versa. No que diz respeito ao *fairness*, a política EDF é intrinsecamente mais justa (*fair*) que a RM, no sentido de que as tarefas veem a sua prioridade elevada à medida que se aproximam do *deadline*, independentemente do seu período ou de outro parâmetro estático;
- ***Deadline Monotonic* (DM)** (LEUNG; WHITEHEAD, 1982): assim como o RM, é um algoritmo ótimo de prioridades fixas, porém atribui às tarefas prioridades de acordo com seu *deadline*, ou seja, menor *deadline* relativo é igual a maior prioridade e vice-versa;
- ***Priority-based scheduling* (PRIO)** (PANICO, 1969; ALLEN, 1978): cada tarefa recebe uma prioridade, a qual pode ser atribuída estaticamente – no momento de sua criação – ou dinamicamente – o escalonador decide o valor da prioridade de acordo com estatísticas sobre a execução anterior. Nesse contexto, nem todas as tarefas podem ser tratadas de maneira idêntica, devido aos fatores externos que podem estabelecer que uma tarefa é mais urgente em relação à outra. Um problema comum dessa política é a monopolização do processador, ou seja, pode ocorrer de sempre chegar uma tarefa mais prioritária que aquelas em espera e estas nunca serem servidas. Para solucionar esse problema, usa-se o método de decremento de prioridade a cada interrupção de tempo;
- ***Shortest Job First* (SJF):** política de escalonamento que atribui às tarefas mais curtas (com menor tempo de execução) as maiores prioridades. Sua filosofia está próxima do ótimo, por reduzir o tempo médio de espera com relação ao FIFO;
- ***Round Robin* (RR)** (PANICO, 1969; ALLEN, 1978): propõe que as tarefas revezem o uso da UCP através de uma unidade de tempo denominada *quantum*, sendo que muitos escalonadores RR usam valores de *quantum* de 20 a 50 ms (TANENBAUM, 2001). As tarefas ficam organizadas em uma fila circular do tipo FIFO e o escalonador tem a responsabilidade de percorrer esta fila e revezar a execução dos processos até todos eles acabarem.

Cada tarefa é executada até o *quantum* estipulado e, após isso, volta para a última posição da fila. Ainda, neste algoritmo, nenhum processo recebe mais de um *quantum* consecutivo; dessa forma, o tempo para o próximo *quantum* de um processo será de *n*q*, onde *n* é o número de processos que há na fila e *q* é o tamanho do *quantum*. Por fim, os tempos de retorno e de resposta aumentam de acordo com o número de processos na fila.

2.5 CONCLUSÕES

Este capítulo apresentou a conceituação de STR, tempo real, concorrência e políticas de escalonamento. A ideia destes conceitos, aliada à definição de agentes inteligentes, serviu de base para o modelo proposto, que será apresentado posteriormente.

3 AGENTES EM SISTEMAS COMPUTACIONAIS

3.1 INTRODUÇÃO

O estudo sobre agentes tem sido um tópico de interesse em pesquisas recentes por apresentar uma estruturação para a construção de sistemas proativos de fácil extensão, quando comparados com as tecnologias atualmente disponíveis para o desenvolvimento de sistemas, devido às suas características de decomposição, abstração e organização (JENNINGS et al., 2000). Algumas áreas dessas pesquisas incluem arquiteturas orientadas a objeto, arquiteturas de objetos distribuídos, engenharia de *software*, sistemas de aprendizado adaptativos, inteligência artificial, sistemas especialistas, algoritmos genéticos, processamento distribuído, algoritmos distribuídos, segurança, entre outros.

Neste capítulo, o paradigma de agentes móveis é mostrado na seguinte sequência: na seção 3.2, é apresentada uma breve fundamentação teórica sobre agentes; a definição de inteligência artificial distribuída é descrita na seção 3.3; na seção 3.4, são apresentados exemplos de aplicações; e finalizando este capítulo, na seção 3.5, são feitos os comentários finais.

3.2 DEFINIÇÕES

Do ponto de vista arquitetônico, agentes são entidades compostas por um identificador único e três componentes principais: código, dados e estado, além de possuírem um ciclo de vida associado ao seu estado de execução (PATTIE, 1995). Esses componentes são mantidos pelo *mobile agent system* na plataforma de agentes, tendo o *agent middleware* como *software* que implementa a plataforma de agentes e a administra.

No contexto da literatura, o termo ‘agente’ não possui uma definição unanimemente aceita; em parte, devido à multiplicidade de enfoques sob os quais é estudado. Algumas das definições fornecidas são:

> Agentes inteligentes são entidades-software que manipulam conjuntos de operações em benefício de um usuário ou de outro programa com certo grau de independência ou autonomia, e de alguma forma, empregam conhecimento ou representação das metas e anseios do

> usuário. (GROSOF et al., 1995, p. 2-4).
>
> Agentes Inteligentes realizam continuamente três funções: percepção de condições dinâmicas no ambiente; ação para afetar condições no ambiente; e raciocínio para interpretar percepções, solucionar problemas, redigir inferências e determinar ações. (HAYES-ROTH, 1995, p. 3).

Partindo dessas definições, Franklin e Graesser (1996, p. 3-4) sugerem um novo conceito do termo 'agente':

> Um agente atua por concessão de um indivíduo ou de outro agente (GROSOF, 1995), dependendo crucialmente do que se considera um ambiente e do que significam os termos 'sentir' e 'agir', onde deve-se restringir, pelo menos, algumas noções referentes a esta dependência (RUSSEL, 1995), possuindo também raciocínio durante o processo de selecionar ações (HAYES-ROTH, 1995).

3.2.1 Classificação de Agentes

3.2.1.1 Quanto às Propriedades

Entre os diversos tipos de agentes encontrados na tipologia (Quadro 3.1), duas importantes classificações podem ser destacadas: agentes reativos e agentes cognitivos (RUSSEL; NORVIG, 2009).

Os agentes reativos não possuem objetivo explicitamente representado e a ação é baseada em estímulo-resposta. Esses tipos de agentes não raciocinam sobre o mundo, não têm "memória" e interagem pouco com outros agentes, agindo, assim, de forma reativa. Também não requerem mecanismos sofisticados de coordenação e protocolos, mas, em uma comunidade, podem produzir um comportamento global inteligente.

Por seu turno, os agentes cognitivos (ou comunicativos) têm como objetivo "cooperar", ou seja, executar sua missão da melhor forma possível, sendo sua ação baseada em intenções (racionais), utilizando-se de memória e interação com outros agentes e ambientes. Além disso, esses agentes possuem um plano de ações construído ou adaptado dinamicamente, planejando futuras ações, e, por se tratarem de entidades de *software* ou

hardware que possuem de média a alta complexidade, requerem sofisticados mecanismos de coordenação e protocolos de suporte à interação de alto nível.

Um modelo complexo que aborda um agente cognitivo é o modelo *Belief-Desire-Intention* (BDI), cuja modelagem é feita via lógica formal com bases filosóficas, baseadas em atitudes mentais: crenças (*beliefs*), desejos (*desires*) e intenções (*intentions*).

Propriedade	Significado
Reativo	Responde, em tempo hábil, a mudanças no ambiente.
Autônomo	Exercita controle sobre suas ações.
Orientado a metas	Não age simplesmente em resposta ao ambiente.
Tempo contínuo	Processo de execução contínua.
Comunicativo	Comunica-se com outros agentes, inclusive, talvez com pessoas.
Aprendizado	Altera o comportamento diante de experiência prévia.
Mobilidade	Apto a se locomover de uma máquina para outra.
Flexível	Ações não seguem um roteiro.
Caráter	Personalidade e estado emocional confiáveis.

Quadro 3.1 – Classificação de agentes por propriedade.
Fonte: Franklin e Graesser (1996).

3.2.1.2 Quanto à Mobilidade

3.2.1.2.1 Agentes Móveis

Aplicações distribuídas são tradicionalmente desenvolvidas utilizando chamadas de procedimento remoto (*Remote Procedure Call* –RPC) ou arquitetura cliente-servidor, na qual os processos comunicam-se através de troca de mensagens (TANENBAUM; STEEN, 2007). Outro conceito é o de objetos remotos, que encapsulam dados e métodos a fim de manipulá-los e transportá-los de um *host* para outro. Há, também, a *Remote Evalu-*

ation (REV), na qual o cliente envia um procedimento ao servidor e este o executa e retorna os resultados.

A partir dessas arquiteturas, um novo paradigma de comunicação em aplicações distribuídas foi desenvolvido, baseando-se na troca não somente de mensagens, procedimentos ou objetos, mas de códigos completos. A ideia principal desse paradigma é utilizar o conceito de agentes inteligentes, tornando-os móveis (WEYNS, 2010).

A comparação entre os protocolos citados pode ser observada na Figura 3.1, na qual é possível perceber, no protocolo RPC, uma comunicação bidirecional para envio de mensagens, enquanto, no REV, o código é enviado pelo cliente ao servidor e apenas os resultados são retornados. Além disso, tratando-se de um agente inteligente móvel, nota-se, pela Figura 3.1, que o programa foi enviado do cliente para o servidor e que, diferentemente de uma chamada de procedimento, o agente pode apresentar autonomia para migrar a outros servidores antes de retornar ao cliente (WEYNS, 2010).

Uma vez que os MAs compreendem a definição de objetos que possuem comportamento, estado e localização, uma das características desejáveis é a autonomia em decidir a rota, sendo que esse comportamento pode ser definido implícita (através do código do agente) ou alternativamente (especificado por um itinerário modificável em tempo de execução) (WONG et al., 1997). Outras características dos MAs são a manutenção do estado de execução, a estratégia de migração utilizada, o momento de iniciar uma nova viagem e/ou uma nova execução (COSTA, 1993), além de não ser necessário saber o local onde o MA foi criado, devido às suas características de abstração e flexibilidade (FOU, 2001).

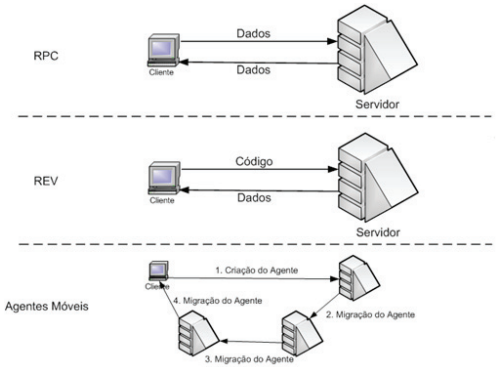

Figura 3.1 – Diferenças entre RPC, REV e MA.
Fonte: Braun e Rossak (2005).

3.2.1.2.2 Agentes Estáticos/Estacionários

Agentes estáticos (ou estacionários) são agentes executáveis somente na máquina em que foram iniciados, não podendo realizar tarefas remotas, porém tendo a capacidade de interagir com agentes remotos (RUSSEL; NORVIG, 2009).

3.2.1.3 Quanto à Capacidade Comunicativa

Os agentes possuem a capacidade de se comunicar independentemente do conjunto de capacidades funcionais que o caracterizam, ou seja, qualquer agente pode enviar e receber mensagens (assertivas ou questionadoras) (HUHNS; STEPHENS, 1999).

Quanto à capacidade de comunicação, um agente pode ser classificado como (Quadro 3.2):

- **agente básico:** recebe asserções, mas não possui quaisquer outras capacidades de comunicação;
- **agente passivo:** mantém um diálogo assertivo, podendo aceitar e responder asserções, além de ter a capacidade de aceitar questões (perguntas), podendo respondê-las somente na forma assertiva;
- **agente ativo:** recebe asserções e as responde em forma assertiva ou questionadora, porém não possui a capacidade de aceitar questões (perguntas);
- **agente interlocutor:** pode assumir o papel de interlocutor entre outros agentes, pois possui a capacidade de manter diálogos tanto assertivos quanto questionadores.

Básico		**Agente**			
		Passivo	**Ativo**	**Interlocutor**	
Receber	Asserções	Sim	Sim	Sim	Sim
	Questões	Não	Sim	Não	Sim
Enviar	Asserções	Não	Sim	Sim	Sim
	Questões	Não	Não	Sim	Sim

Quadro 3.2 – Classificação de agentes quanto à capacidade comunicativa.
Fonte: Huhns e Stephens (1999).

3.3 INTELIGÊNCIA ARTIFICIAL DISTRIBUÍDA

A Inteligência Artificial Distribuída (IAD) é a classe de sistemas que permite a vários processos denominados agentes interagirem, distribuindo-se logicamente ou, algumas vezes, espacialmente, podendo ser chamados autônomos e inteligentes; é, também, a interseção da computação distribuída e da Inteligência Artificial (IA) (BOND; GASSER, 1988) (Figura 3.2).

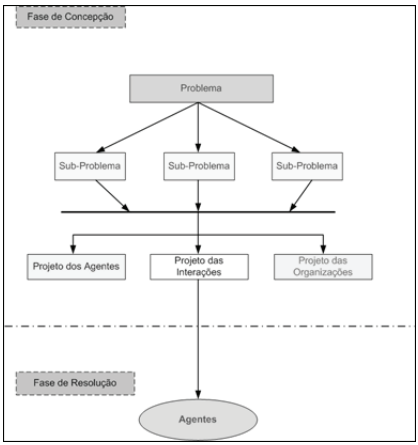

Figura 3.2 – Conceito de IAD.

A IAD divide-se em duas áreas: resolução distribuída de problemas (*Distributed Problem Solving* – DPS) e sistema multiagente (*Multi Agent System* – MAS).

3.3.1 Resolução Distribuída de Problemas

As DPS trabalham com o gerenciamento de informações, que consiste na decomposição de tarefas e na síntese de solução, na qual o resultado de diferentes subtarefas é combinado (BOND; GASSER, 1988).

Nesse sentido, o método de resolução de problemas de uma DPS baseia-se em um conjunto de agentes distribuídos fisicamente em diversos *hosts* da rede (Figura 3.3). Os agentes interagem entre si por mensagens ou por compartilhamento de dados comuns e são executados de modo concorrente, a fim de aumentar a velocidade da resolução do problema. Além dis-

so, eles cooperam entre si, dividindo a resolução de diferentes subproblemas, ou podem decidir, autonomamente, por aplicar diferentes estratégias de resolução a uma mesma tarefa.

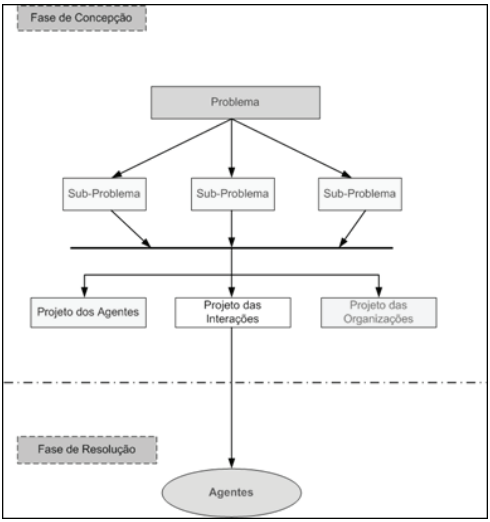

Figura 3.3 – Estrutura de um sistema DPS.

3.3.2 Sistema Multiagente

Um MAS consiste em uma aplicação distribuída, composta por um conjunto de processos autônomos, heterogêneos, distribuídos e inteligentes, que cooperam entre si para a solução de um problema complexo que está além das suas capacidades individuais (SAJJA, 2008).

O MAS busca definir formas de coordenação dos seus conhecimentos, objetivos, habilidades e planos, de forma que, conjuntamente, os processos possam realizar ações ou resolver problemas (BOND; GASSER, 1988). Para tanto, a missão dos agentes pode ser centralizada – na qual todos os agentes trabalham em direção a um único objetivo global – ou distribuída – na qual há vários objetivos individuais, podendo haver interação entre eles. A Figura 3.4 ilustra esse cenário.

Quanto à mobilidade, os agentes migram entre os *hosts* do MAS para utilizar os recursos destes *hosts* remotos de forma local. Nesse contexto, em um MAS, cada máquina é mapeada utilizando o conceito de localização, ou seja, possui uma identificação única de um lugar no espaço de endereçamento distribuído.

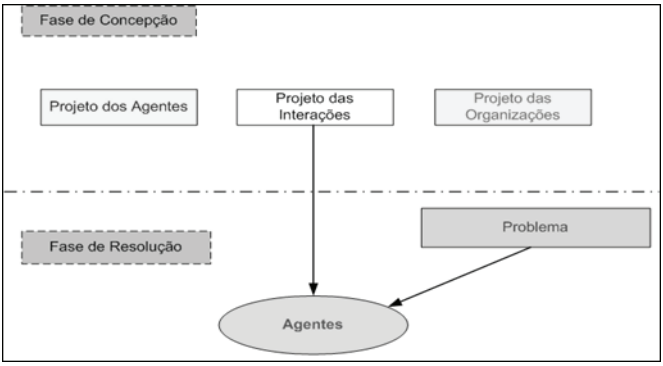

Figura 3.4 – Estrutura de um sistema MAS.

3.4 APLICAÇÕES

As aplicações atuais de agentes são de natureza experimental, porém grandes esforços têm sido feitos na área de pesquisa com essa tecnologia (MULLER, 1999; CAO et al., 2005). Nesse contexto, as aplicações como tecnologia de agentes têm sido agrupadas da seguinte forma (OMG, 2011):

- **acesso e gerenciamento da informação:** um ambiente de intranet corporativa possui restrições no que diz respeito à capacidade de a maioria dos usuários recuperarem efetivamente informações úteis. Agentes inteligentes podem ser empregados na filtragem e busca de informações em nível de domínio, bem como obter noções sobre segurança e precisão das fontes de informação;
- **comércio eletrônico:** vendedores, gerentes e clientes podem ser representados por agentes móveis em um sistema de comércio eletrônico. Esses agentes têm autonomia para: negociar, buscar sugestões de produtos através de especificações fornecidas pelo usuário, atuar como assistentes de vendas e concluir uma compra;
- **controle de processos:** atribuição de agentes ao controle de processos em edifícios inteligentes (aquecimento/refrigeração e segurança inteligentes – *smart*), em gerenciamento de aparelhagem (refinarias etc.), em robôs e outras aplicações autônomas;
- **correio eletrônico e troca de mensagens:** agentes inteligentes podem facilitar funcionalidades como *anti-spam* e organização

automática do correio de usuários, atribuindo prioridades a determinados tipos de mensagens recebidas e enviadas. Tais funcionalidades podem ser definidas por regras preestabelecidas ou por aprendizado, conforme comportamento do usuário;

- **gerenciamento de sistemas e redes:** devido à capacidade de colaboração e mobilidade dos agentes, a implementação de um sistema de gerenciamento com capacidade de resposta em tempo real e tráfego de dados reduzido é viável em nível de alterações ocorridas no domínio da rede.

3.5 CONCLUSÕES

Este capítulo teve por objetivo esclarecer os conceitos que envolvem a definição de um agente de *software*, suas propriedades e capacidades e seu uso em sistemas distribuídos. Através das características que foram relacionadas e dos serviços disponíveis apresentados, justifica-se o interesse em unir a tecnologia de agentes móveis a sistemas com requisitos de tempo real, uma vez que MAs são uma alternativa viável para a construção de sistemas distribuídos, possuindo vantagens sobre as demais alternativas mencionadas neste capítulo.

O próximo capítulo abordará os trabalhos relacionados à proposta desta dissertação.

4 TRABALHOS RELACIONADOS

Partindo dos resumos teóricos apresentados nos capítulos anteriores, buscou-se o estado da arte para a justificação da proposta desta dissertação. Assim, através do levantamento bibliográfico realizado, pôde-se observar que, no campo da computação distribuída inteligente em redes de computadores dinâmicas, uma questão que tem atraído o interesse da comunidade é o uso de MA e MAS para solucionar problemas do cotidiano.

Entre as muitas pesquisas existentes nesta área, podemos citar o uso de *middlewares* para MAs, os quais visam a dar suporte a diversas aplicações, com o objetivo de que estas englobem MAs em sua arquitetura.

Este capítulo apresenta as principais características de alguns *middlewares* para MAs encontrados na literatura.

4.1 AGILLA

Agilla (FOK; ROMAN; LU, 2005) é um *middleware* que visa a facilitar a comunicação interagentes em redes de sensores sem fio, permitindo ainda que usuários criem e enviem MAs pela rede. Ainda, fornece um mecanismo de troca de contexto que permite a execução simultânea e independente de cada um dos MAs.

A comunicação interagente dá-se por meio de um espaço de tuplas, que é a memória da arquitetura compartilhada por todos os agentes em execução no *host*-sensor, sendo acessível remotamente (Figura 4.1); assim, os agentes não se comunicam diretamente com outros agentes.

Cabe informar que uma tupla é uma sequência de objetos e/ou dados que é inserida no espaço de tuplas. Nesse sentido, mesmo que o MA que inseriu os dados migre, estes permanecem no espaço, permitindo que, posteriormente, outro agente possa recuperar as tuplas mediante a emissão de uma consulta com a mesma sequência de campos.

Agilla permite, ainda, a implantação do código, que é enviado através dos MAs na rede, de forma que os agentes podem migrar ou clonar-se de forma autônoma para locais apropriados, a depender da aplicação.

Para tanto, assume-se que cada *host*-sensor conheça a sua localização física; já a quantidade de MAs que podem executar em um mesmo *host* é variável e determinada pela quantidade de memória disponível do *host*. Além disso, os MAs podem compartilhar seu código com outros agentes

do sistema, pois, em cada *host*-sensor, há a lista dos nodos vizinhos e um espaço de tuplas, conforme mencionado anteriormente. Antes de migrar ou clonar-se, o MA verifica na lista de nodos vizinhos os que estão alcançáveis e decide por um deles.

Entretanto, o *TinyOS* – sistema no qual o Agilla foi implementado – não fornece gerenciamento de memória dinâmica. Com isso, os autores do *middleware* elaboraram um mecanismo de gerenciamento de memória dinâmica próprio para as instruções dos MAs e do espaço de tuplas.

Devido à pequena dimensão física e dependência de bateria, a utilização da largura de banda é baixa e não confiável, acarretando a degradação do código e dados dos MAs, devido à perda de pacotes. Para solucionar esse problema, os MAs são divididos em pequenos pacotes de 41 bytes. Dessa forma, a memória do *host*-sensor é compartilhada entre os MAs que estão em execução, porém cada MA possui uma pilha de execução de cerca de 50 bytes, alocada dinamicamente de acordo com o tamanho dos dados. Note que as instruções do agente não são armazenadas dentro do espaço de endereçamento reservado para o agente, pois são gerenciadas pelo Agilla.

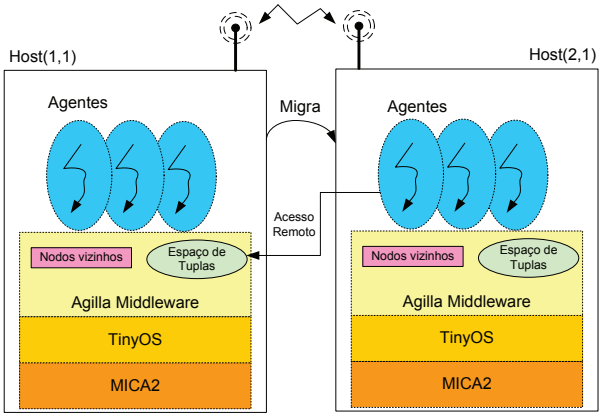

Figura 4.1 – Modelo do *middleware* Agilla.

4.2 FTS

O trabalho apresentado por Leung (2010) propõe um módulo que permite ao JADE fornecer tolerância a faltas e segurança em MAS. Para

tanto, o autor cita alguns recursos, como a detecção de ação maliciosa no *host* remoto e a recriação do agente suspeito de ser malicioso através de suas características anteriores (*rollback*), utilizando o mecanismo de clonagem fornecido pelo JADE e tratando cada passo do *host* como uma transação. No entanto, o autor não dá detalhes adicionais sobre o projeto em sua página.

4.3 MOBIGRID

Grande parte dos computadores pessoais permanece parcialmente ociosa durante longos períodos de tempo. Nesse sentido, o InteGrade (GOLDCHLEGER et al., 2004) é um *middleware* que constrói uma grade computacional em ambientes heterogêneos, utilizando computadores comuns ao invés de máquinas dedicadas, com o objetivo principal de implantar grades sobre recursos computacionais que seriam desperdiçados.

Para este sistema, os autores dividiram os *hosts* em tipos distintos de nodos, denominando-os aglomerado InteGrade. Entre eles, podem-se citar: o nodo dedicado, que é uma máquina reservada à computação em grade, podendo seus recursos ser integrados à grade, se preciso; o nodo compartilhado, que pertence ao usuário que disponibiliza recursos ociosos de sua máquina à grade; o nodo usuário, que submete aplicações para serem executadas na grade; e o nodo gerenciador de aglomerado, no qual são executados os módulos responsáveis pela coleta de informações, entre outros.

Por sua vez, o MobiGrid (BARBOSA; GOLDMAN, 2004) é um *middleware* que visa à criação de uma infraestrutura de MAs para o ambiente de grade InteGrade, provendo um ambiente de programação para aplicações com longo tempo de processamento encapsuladas em um MA.

O papel dos MAs, neste *middleware*, é prover a tolerância a falhas nas aplicações. Assim, o nodo usuário lança uma aplicação à grade por meio do MA (Figura 4.2), que migra com essa aplicação a diversos *hosts* distintos, realizando periodicamente o *checkpointing* e salvando seu estado atual. Caso ocorra alguma falha na aplicação, a restauração é dada a partir do último *checkpoint* e o MA migra com essa aplicação restaurada para outra máquina, a fim de continuar a execução do ponto de parada.

Ressalte-se que a migração exercida pelos MAs é forte (s*trong migration*), ou seja, ao migrar, não apenas os dados migram, mas também o estado do agente.

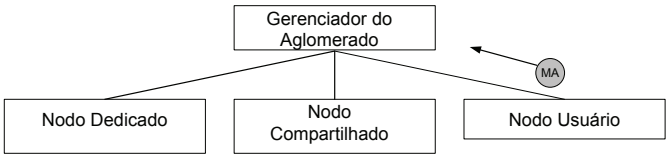

Figura 4.2 – Arquitetura MobiGrid.

4.4 ECOMOBILE

O projeto EcoMobile (ROSSIER-RAMUZ; SCHEURER, 2002) propõe uma infraestrutura de MAs baseada em princípios do ecossistema, em que cada nodo é considerado um ecossistema, servindo como um *middleware* para aplicações distribuídas baseadas em redes, como, por exemplo, algoritmos de gerenciamento de rede. O modelo do sistema consiste em dois aspectos: o *Mobile Behaviour Scheme* (MBS – esquema de comportamento móvel) e o *Task Objective* (TO – objetivo da tarefa).

O MBS implementa tanto um modelo de migração quanto o modelo de coordenação, descrevendo o comportamento do agente do ponto de vista da migração e interação interagente, por meio de comportamentos reativos (φ-comportamentos). Por sua vez, os φ-comportamentos caracterizam atividades de indivíduos do ecossistema, como movimento, espera, reprodução, ação etc., ou ainda atividades que implicam interações com outros indivíduos, como a comunicação ou a concorrência, porém os autores não especificaram nenhuma abordagem de tratamento desta concorrência.

Além disso, o MBS pode ser um, entre dois tipos: o MBS-*low* e o MBS-*high*. No MBS-*low*, o MA inicia a migração imediatamente após sua criação (denominada "parto", pelos autores). Quando o MA chega a um novo *host,* ativa o comportamento de ação, que consiste em carregar os objetivos da tarefa presente no nodo ao seu contexto. Ainda, se um MA (*A*) consegue contatar outro MA (*B*) no nodo, *B* passa para *A* todos os objetivos da tarefa presente no nodo, independentemente da sua natureza – os autores, porém, não deixam claro o porquê dessa abordagem. Caso *B* "morra" enquanto *A* recebe seu conhecimento, *A* migra para outro *host,* a fim de "conquistá-lo" e, se um MA não consegue contato com outro agente, este se clona e continua a migrar pela rede.

O MBS-*high* tem o mesmo comportamento do MBS-*low*, sendo diferenciado pelo processo de clonagem antes de deixar o nodo. Antes de partir, o MA clona-se em quantas réplicas forem necessárias (diretamente depen-

dente do número de nodos interligados ao atual) e envia um clone seu para cada nodo interligado ao *host* atual; isso se dá para que todos os nodos de saída sejam investigados pelo MA. Esse comportamento, que é semelhante a uma busca em largura paralela, permite a exploração exaustiva da rede, levando a uma difusão mais rápida de MAs.

O TO consiste no comportamento operacional do agente, ou seja, a descrição de uma tarefa específica, como, por exemplo, gerenciamento de rede, roteamento, acompanhamento, alocação de recursos, detecção de falhas etc. Para tanto, os TOs são depositados no ambiente e carregados dinamicamente pelos MAs do ecossistema.

Ressalte-se que a tarefa não tem efeito direto sobre o comportamento do agente. Assim, no caso de ocorrer um destino inadequado ao seu objetivo, o TO pode ser retirado do MA proprietário e carregado de volta ao ambiente para ser utilizado por outro MA.

Em detalhes, o sistema EcoMobile possui os seguintes elementos em cada nodo (Figura 4.3):

- ***M-agent*:** MA do ecossistema;
- ***agency*:** agente *Foundation for Intelligent Physical Agents* (FIPA), que implementa os componentes do EcoMobile e as políticas de segurança, podendo coexistir com outros agentes estacionários. Tem como responsabilidades gerenciar o ambiente, regular as atividades do *M-agent* e gerenciar a interação interagente e agente-ambiente;
- ***place* (lugar):** é o ambiente dos *M-agents* e também um espaço de coordenação entre eles. Os *places* são interligados para formar a conectividade que permite aos *M-agents* migrarem de um nodo para outro;
- ***blackboard* (quadro negro):** cada *place* está associado a um *blackboard* passivo, que é parte do ambiente do ecossistema e permite que *M-agents* depositem informações destinadas a outros agentes. Contudo, cada *M-agent* possui acesso somente ao *blackboard* associado ao *place* em que reside. O *blackboard* também é usado como um repositório de TOs;
- **MIB:** contém objetos gerenciados, acessíveis aos *M-agents*;
- **agente *Simple Network Management Protocol/Common Management Information Protocol* (SNMP/CMIP):** manipula os objetos gerenciados do MIB, através de um protocolo padronizado (CMIP/SNMP).

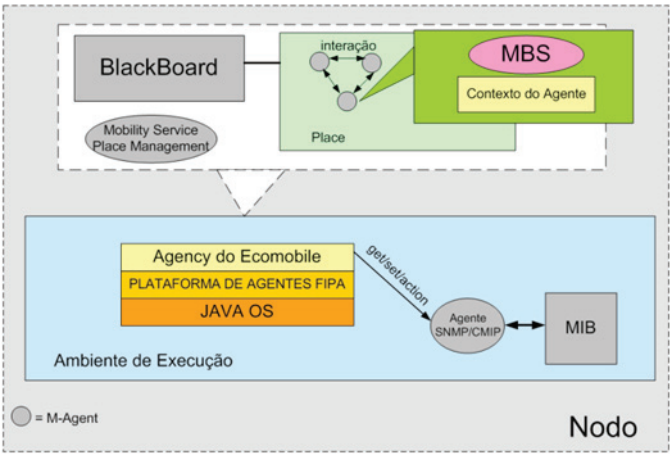

Figura 4.3 – Ambiente do EcoMobile.

4.5 TLMAS

O modelo de Sahingoz e Erdogan (2004) propõe um *middleware* utilizando clonagem de agentes, com o objetivo de reduzir o tempo de completude de transações comerciais. O trabalho consiste não somente em prover suporte a clientes e servidores, mas também facilitar a computação paralela de MAs. Assim, o cenário divide-se em três agentes estáticos (*buyers, suppliers* e *dispatcher/broker*) e dois tipos de MAs (*Broker level Mobile Agent* – BMA – e *Supplier level Mobile Agent* – SMA).

O agente estático *buyer* tem como objetivo buscar serviços fornecidos pelo agente *supplier*, que, por sua vez, oferece serviços e produtos. Já o *dispatcher* – ou *broker* – tem como função facilitar a comunicação interagente, por centralizá-la.

Nesse contexto, o usuário comprador inicia o *buyer subsystem* em sua máquina, o *buyer agent* especifica o critério para a função desejada (compra ou pesquisa) e cria uma instância do BMA, que migra para o *broker*. Por seu turno, a parte vendedora inicia em sua máquina o *supplier subsystem* e une-se aos demais *suppliers,* se houverem. Quando um *supplier subsystem* é executado pela primeira vez, deve inscrever-se no sistema, fornecendo endereço de *host* e nome dos produtos disponíveis para transações, e aguardar requisições dos compradores. Se um *supplier* inicia uma venda de um novo produto, deve inscrevê-lo (*publish*) e, se encerrar, subscrevê-lo (*subscribe*).

No *broker*, o BMA verifica a base de conhecimento e seleciona os *suppliers* que têm os produtos requisitados, sendo que, para cada nodo *supplier*, uma réplica (clone) do BMA é enviada, a qual é denominada SMA (Figura 4.4). Os SMAs migram concorrentemente para seus respectivos *suppliers*, beneficiando-se, assim, das vantagens do processamento paralelo.

Esse modelo tem como vantagem a possibilidade de mais *suppliers* poderem ser pesquisados em curto espaço de tempo.

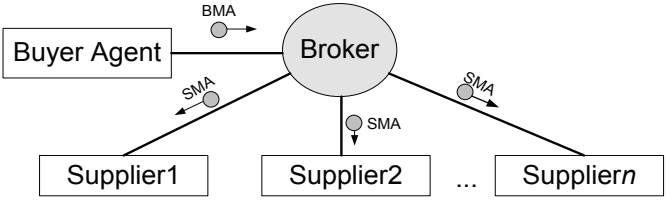

Figura 4.4 – Modelo de clonagem de agentes do *middleware*.

Na sequência, cada SMA faz uma busca na base de dados de seu respectivo *supplier*, negociando com este e retornando via mensagem os resultados ao BMA. Cabe ressaltar que os SMAs podem acessar a base de dados de produtos apenas em modo *read only*.

Para reduzir o tráfego de informações desnecessárias de uma intermediação de produto, os autores desenvolveram um *filtering by rules*. O filtro de critérios dos *buyers* é iniciado no BMA no momento de criação pelo *buyer agent*, de modo que o vendedor faz um *upload* do critério de filtragem do produto, como uma regra, na mensagem de inscrição do processo de registro de um produto. Essa regra é armazenada na tabela de inscrição e o BMA utiliza-a por examinar essa tabela.

Por fim, após receber os resultados de todos os SMAs (Figura 4.5) ou ocorrer *timeout*, o BMA escolhe o melhor resultado e envia uma mensagem de aprovação ao SMA correspondente e uma mensagem de reprovação aos demais SMAs. Aqueles que receberam a mensagem de reprovação autodestroem e o SMA aceito finaliza a compra do produto, retornando o resultado ao BMA e autodestruindo. O BMA, por sua vez, envia o histórico da negociação e resultados ao *buyer agent*.

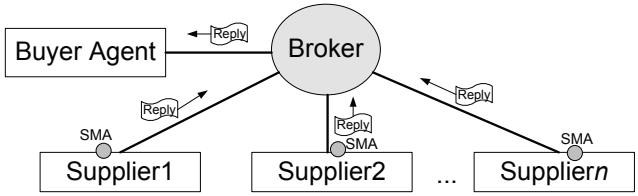

Figura 4.5 – Modelo de troca de mensagens do *middleware*.

4.6 MRSCC

O projeto MRSCC (SHEMSHADI, SOROOR; TAROKH, 2008) visa a produzir uma arquitetura de tempo real integrada ao JADE para membros da cadeia de abastecimento que possibilite a criação de produtos utilizando a tecnologia de MAs. Para fazer com que agentes JADE estejam acessíveis em *JavaServer Pages* (JSP), um *middleware* denominado *gateway agent* foi adicionado ao sistema proposto, bem como ACL *messages* foram utilizadas dentro do MAS e objetos *bean*, denominados *board beans,* fora do MAS. Assim, cada vez que um usuário loga-se, um MA dedicado é criado para atender a seu pedido (Figura 4.6), o qual migra dentro do MAS, à procura de produtos e do melhor preço, e tem a capacidade de efetivar a compra.

Para os agentes, os autores definiram quatro classes distintas, quais sejam: *procurement-agent, supplier agent, trade-gateway-agent* e *ontology*. Os agentes definidos na *procurement-agent* têm como responsabilidades a negociação com agentes fornecedores (*supplier agents*) e a busca por itens-alvo com o melhor preço ou a melhor qualidade, de acordo com um padrão de análise interna do sistema. Para poupar energia e recursos do sistema, os MAs enviam solicitações somente para os agentes fornecedores dos serviços relacionados, baseando-se em ontologias.

Os *supplier agents* têm como responsabilidade a negociação com os agentes de compras (MAs de compras) e, após fornecer as informações necessárias àquela negociação, autobloqueiam, até que um novo pedido de negociação seja feito. Já os *trade-gateway-agents* (implementados pelo *gateway agent*) têm como função interagir com os *board beans* e manipular as *ACL messages* que entram e saem do MAS.

Por fim, a *ontology class* fornece a ontologia necessária para os MAs de compras, uma vez que eles utilizam seus métodos para identificar o que os fornecedores podem oferecer entre seus itens favoritos. Dependendo da

complexidade do ambiente do *software* de negócios, essa classe pode ser estendida para usar estruturas mais complicadas.

Ainda, os autores demonstraram em seu artigo, através de *print screen*, que cada *supplier agent* encontra-se em um contêiner distinto no JADE, porém não apresenta a utilização de um mecanismo de tratamento para requisições concorrentes.

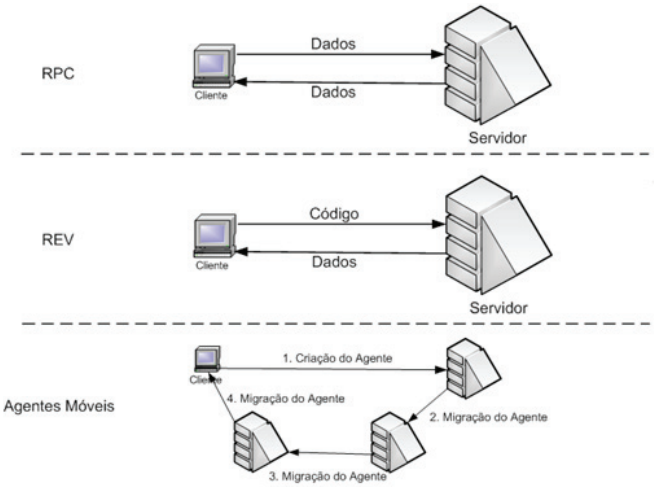

Figura 4.6 – Arquitetura MRSCC baseada na tecnologia JADE.

4.7 REMMOC

A plataforma ReMMoC (GRACE; BLAIR; SAMUEL, 2003) é um *middleware* reflexivo configurável dinamicamente, que suporta o desenvolvimento de aplicações móveis e elimina as propriedades heterogêneas de um ambiente móvel. O projeto propõe o uso de reflexão e tecnologia de componentes para resolver o problema de comunicação entre *middlewares* heterogêneos no ambiente de computação móvel.

As aplicações móveis podem ser do tipo RPC, orientado a mensagens, orientado a agentes etc. Dessa forma, ganham flexibilidade através de *middlewares* reflexivos, no sentido de poderem interagir com diferentes tecnologias de *middleware* e protocolos de descoberta de serviços.

Como exemplo, pode-se citar um ambiente no qual alguns serviços estão disponíveis para usuários em duas localidades. As instâncias de

cada serviço são implementadas utilizando diferentes tipos de *middleware*, como, por exemplo, serviços oferecidos por uma plataforma de MAs. Digamos que o mesmo serviço esteja disponível em outra localidade, porém utilizando *Common Object Request Broker Architecture* (CORBA); o ReMMoC torna-o disponível para o usuário nas duas localidades, embora implementados com diferentes tecnologias de *middleware*, protocolos de descoberta de serviços e protocolos de troca de mensagens (Figura 4.7).

Caber lembrar que dispositivos móveis geralmente possuem limitações quanto à quantidade de recursos disponíveis para dar suporte a uma variedade de tecnologias de *middleware*, protocolos de troca de mensagens e protocolos de descoberta de serviço no mesmo dispositivo. Portanto, um *middleware* num dispositivo móvel deve estar apto a mudar dinamicamente sua estrutura e comportamento, de modo que possa descobrir e interoperar com todos os serviços disponíveis no ambiente corrente.

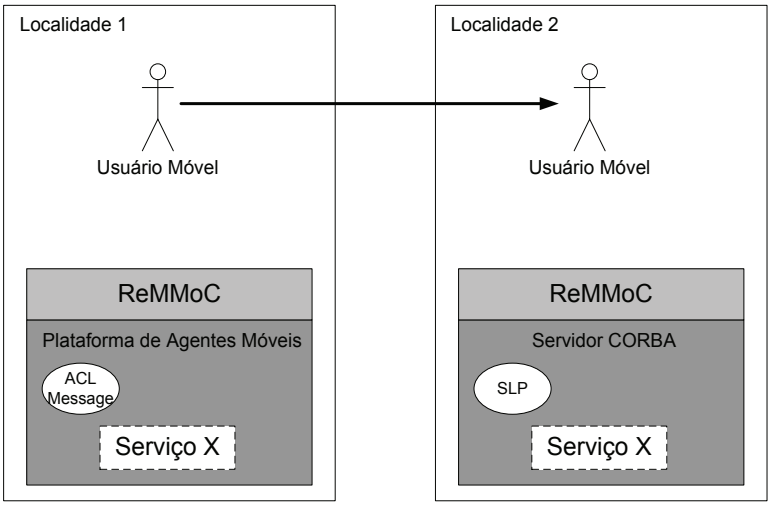

Figura 4.7 – Exemplo de uso do ReMMoC.

4.8 SMART MESSAGES

O *Smart Messages* (SMs) (KANG et al., 2004) é uma plataforma de computação distribuída para sistemas embarcados baseada em migração, execução e autorroteamento para troca de mensagens entre aplicações. Para tanto, os autores assumem uma arquitetura descentralizada, na qual

os nodos na rede agem como pares (*peers*). Essa arquitetura possui um *tag space*, que oferece uma memória baseada em nomes, persistente entre as execuções do SM, e é constituída por duplas (nome, data) denominadas *tags*, que são usadas para a troca de dados entre SMs.

Um exemplo dado pelos autores é de um grupo de pessoas que sai de uma conferência e quer um táxi. Em vez de chamar uma companhia de táxi ou esperar na rua por um táxi livre, um deles usa seu dispositivo de bolso (por exemplo, um PDA) para injetar um SM na rede e reservar certo número de táxis livres. Uma vez que cada cabina de táxi oferece suporte para a execução de SMs e é identificada por uma *tag* chamada *free cab*, o SM migra para os *free cabs* e muda suas *tags* para *occupied cab.*

O ponto-chave do SM é o modelo de programação com migração baseada em nomes, que possibilita o roteamento de MAs utilizando *tags*. Dessa forma, um SM nomeia os nodos de interesse por *tags* próprias e invoca uma função de migração de alto nível para rotear a si mesmo para os *hosts* de interesse através das *tags*.

No exemplo da Figura 5.8, o código de aplicação tem como objetivo permitir ao SM adaptar-se rapidamente às mudanças na rede. Nesse sentido, durante as migrações entre os nodos de interesse, o código de roteamento pode ser instruído a retornar o controle para a aplicação, caso a rota não seja encontrada, sendo que, a cada nodo que o SM migra, recebe em seus dados a quantidade de saltos efetuada (dada pela variável i da Figura 4.8).

Por sua vez, a função *migrate* (“*FreeCab”*) roteia o SM para os *hosts* de interesse (*hosts* com *tags = free cab*), usando os *occupied cabs* como nodos intermediários. Embora o código de roteamento seja executado em cada nodo que o SM migra através da rede, o controle do aplicativo dá-se apenas nos nodos de interesse (por exemplo, nos *free cabs*). Vale ressaltar que a migração dos SMs é feita de forma explícita, como, por exemplo, quando um programador solicita a migração, quando esta é necessária.

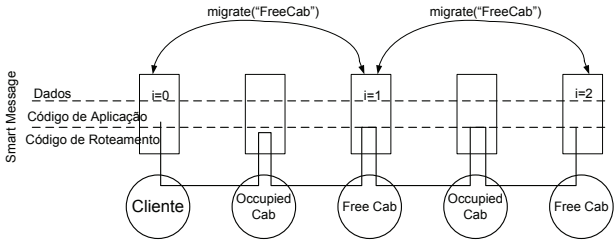

Figura 4.8 – Modelo de execução do SM.

4.9 CONSIDERAÇÕES FINAIS SOBRE AS ABORDAGENS ENVOLVENDO *MIDDLEWARES* PARA AGENTES MÓVEIS

Neste capítulo, foram apresentadas abordagens que mostram *middlewares* sendo utilizados para auxiliar, aperfeiçoar e/ou inserir o uso de MAs em diversas aplicações, quer preconcebidas ou não. O objetivo principal foi apresentar propostas encontradas na literatura que tratassem a questão de *middlewares* aliados à mobilidade de código. Nem todas as propostas têm interesses, características e objetivos comuns a esta dissertação, mas o seu estudo serviu de base para o desenvolvimento da proposta presente nesta dissertação.

Por exemplo, o *middleware* Agilla (FOK; ROMAN; LU, 2005) tem como objetivo facilitar a comunicação interagentes em redes de sensores sem fio, por utilizar um espaço de tuplas, o qual também permite o acesso remoto aos dados nele contidos. Pela possibilidade de perda de pacotes durante a transmissão, o Agilla divide os MAs em pequenos dados, para depois transmiti-los pela rede, sendo que cada *host* é identificado com base em sua localização.

Por sua vez, Barbosa e Goldman (2004) propõem um *middleware* que utiliza MAs para fornecer ao InteGrade tolerância a falhas nas aplicações enviadas pelos usuários à grade. Assim, os MAs migram entre os nodos, a fim de executar a aplicação, realizando periodicamente o *checkpointing* desta para, caso a aplicação venha a falhar, possa ser executada do ponto de parada em outro nodo.

O EcoMobile (ROSSIER-RAMUZ; SCHEURER, 2002) é um *middleware* para aplicações distribuídas, que utiliza o conceito de ecossistema, possuindo definições de comportamento para os MAs e definições de objetivos para as tarefas. Cada nodo é tido como um ecossistema com tarefas a serem realizadas; dessa forma, os MAs migram entre os ecossistemas com o objetivo de "conquistá-los", ou seja, realizar as tarefas contidas nesses *hosts*. Para isso, os MAs interagem com outros MAs e com o próprio ambiente.

Em Sahingoz e Erdogan (2004), foi proposto um algoritmo utilizando agentes centralizados para auxiliarem dois níveis distintos de MAs: BMA e SMA. O objetivo da proposta é a redução do tempo de completude da missão por parte dos MAs através do uso de clonagem de agentes: um clone é enviado a cada nodo *supplier* para executar uma missão em comum; ao completar sua missão, encaminha o resultado ao BMA, que, por

sua vez, elege a melhor entre as respostas recebidas e a encaminha ao agente centralizador. O clone proprietário da melhor resposta conclui a compra do produto, enquanto os demais autodestroem. Ao concluir a compra, o clone encaminha o histórico de operações ao BMA, que o reencaminha ao agente centralizador.

Já o projeto MRSCC (SHEMSHADI; SOROOR; TAROKH, 2008) visa a produzir uma arquitetura de tempo real integrada ao JADE, para possibilitar a criação de produtos com MAs para membros da cadeia de abastecimento. Assim, cada vez que um usuário loga-se, um agente dedicado é criado para atender ao seu pedido, o qual migra à procura de produtos e do melhor preço e tem a capacidade de efetivar a compra.

Ainda, Grace, Blair e Samuel (2003) propõem um *middleware* que possibilite a um usuário móvel usar um mesmo serviço disponível em duas localidades distintas, mas utilizando tecnologias diferentes. Nesse contexto, se um usuário faz uso de um serviço X disponível numa localidade L1 utilizando uma plataforma de MAs e vai para uma localidade L2, na qual o mesmo serviço está disponível em CORBA, o ReMMoC permite que o usuário continue a utilizar o serviço, de forma transparente.

Por fim, o projeto *Smart Messages* (KANG et al., 2004) visa a fornecer suporte à migração, autorroteamento, execução e troca de mensagens entre aplicações. Para a migração, os autores utilizaram o conceito baseado em nomes (*tags*), no qual os nodos em que o MA irá executar uma tarefa recebe *tags* próprias, que definem esse comportamento para o MA, e os demais nodos recebem outro tipo de *tags,* definindo-os como nodos a serem utilizados somente para chegar ao *host* proprietário do recurso desejado.

Diante do exposto, um breve comparativo dos trabalhos apresentados é demonstrado no Quadro 4.1, o que possibilitou verificar, partindo da ideia inicial do problema tratado nesta dissertação, que existe uma lacuna na literatura.

Plataforma	Tipo de migração	Tratamento de concorrência	Tolerância a falhas	Uso de clonagem	Restrição temporal	Comunicação remota
Agilla	Fraca	Não	Sim (dados)	Sim	Não	Sim
FTS	Fraca	Não	Sim (MA malicioso)	Sim	Não	Sim
MobiGrid	Forte	Não	Sim (aplicação)	Não	Não	Sim
EcoMobile	Fraca	Não	Não	Sim	Não	Não
TLMAS	Fraca	Não	Sim (regras)	Sim	Sim	Sim
MRSCC	Fraca	Não	Sim (*suppliers*)	Não	Não	Sim
ReMMoC	Depende da aplicação	Não	Não	Depende da aplicação	Não	Sim
SM	Explícita	Não	Não	Não	Não	Sim

Quadro 4.1 – Comparativo entre as propostas apresentadas.

Analisando o Quadro 4.1, é possível perceber que nenhuma das abordagens leva em conta a questão do tratamento de MAs concorrentes. Visando a atacar esse problema, a presente dissertação traz a proposta de uma extensão de *middleware* que possibilite escalonamento em tempo real de MAs concorrentes, de modo que o tema desta dissertação é inédito na literatura.

Seguindo a visão de Leung (2010) e de Shemshadi, Soroor e Tarokh (2008), nesta dissertação propõe-se uma melhoria à plataforma JADE, que permita o tratamento de requisições de MAs concorrentes em tempo real em um mesmo *host*, por meio de *middleware*. Para a simulação desse *middleware*, montou-se uma arquitetura baseada no modelo de Sahingoz e Erdogan (2004), com utilização de um *broker* e agentes estáticos em cada nodo para atender às requisições dos MAs e gerenciar os recursos disponíveis no nodo.

Além disso, outro objetivo deste trabalho é a atribuição dinâmica de prioridades para os MAs através de algoritmos de escalonamento.

4.10 CONCLUSÕES

Este capítulo teve por objetivo discutir estudos que propusessem a utilização de *middlewares* para MAs, comprovando, assim, o expressivo interesse nesta área. Para tanto, buscou-se, na bibliografia, artigos que dessem suporte aos objetivos desta dissertação, que serão considerados no próximo capítulo.

Com relação ao escalonamento de MAs em tempo real, objetivo de estudo desta dissertação, algumas abordagens foram estudadas com o propósito de servir como inspiração ao desenvolvimento dos algoritmos propostos.

O próximo capítulo contém o modelo computacional desenvolvido, que descreve o comportamento de MAs com restrições temporais que concorrem por um mesmo recurso em um mesmo *host*. Já nos capítulos 5 e 6, serão apresentados e avaliados os algoritmos que auxiliam a plataforma JADE na tomada de decisão sobre prioridades dos MAs.

5 RT-JADE

5.1 INTRODUÇÃO

Apesar de a utilização de MAs em sistemas de tempo real já ser amplamente difundida e estudada, ainda não há pesquisas voltadas para o escalonamento de MAs em aplicações de tempo real, mais especificamente, o desenvolvimento de um *middleware* com suporte para escalonamento de MAs.

Nesse sentido, um importante problema não tratado nas plataformas de *middleware* existentes é o aspecto da concorrência dos recursos em um *host* remoto. Quando vários agentes visitam o mesmo *host* simultaneamente, para utilizar um determinado recurso, não há mecanismos de escalonamento em tempo real para o uso desse recurso por parte dos agentes visitantes, ou seja, o que normalmente se faz é o atendimento desses agentes segundo a ordem de chegada. Uma vez que, em aplicações de tempo real, os MAs devem executar suas tarefas de acordo com suas restrições temporais, é necessário atribuir prioridades no escalonamento para a utilização dos recursos, desse modo, permitindo aos agentes com maior prioridade terem acesso aos recursos mais rapidamente.

A busca por soluções para desenvolver um sistema de escalonamento de MAs é importante para aplicações que necessitam cumprir restrições temporais, pois tais agentes são normalmente assíncronos em relação à chegada e saída dos *hosts*, devido à sua característica autônoma, o que faz com que o sistema não tenha um controle prévio sobre prioridades, alocando as entradas e saídas dos agentes somente em política FIFO (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

Um exemplo é o contexto de uma linha de produção, em que todos os recursos de manufatura de peças são frequentemente representados como MA, enquanto outros recursos de produção, como máquinas, instrumentos e ferramentas de corte, são representados como agentes estáticos (WANG et al., 2006). Assim, a função de cada agente depende de sua colocação dentro da produção em chão de fábrica, podendo negociar, ofertar ou votar em contratos de diversas áreas, porém, se houver prazos ou ordem de prioridade, o agente precisa atender a essas restrições temporais ou poderá prejudicar toda a linha de produção.

Como principal contribuição desta dissertação, foi proposta a arquitetura RT-JADE, que tem como objetivo tratar o problema de escalonamen-

to de MAs com restrições temporais, visando à questão de concorrência entre estes. Nesse sentido, neste capítulo, é apresentada uma proposta de extensão da plataforma JADE para dar suporte ao escalonamento de MAs; as extensões de *middleware* propostas comportam diferentes tipos de políticas de escalonamento, tais como: FIFO, LIFO, EDF, EDF preemptivo, prioridade, prioridade preemptivo, DM, DM preemptivo, SJF e *Shortest Remaining Time First* (SRTF), oferecendo uma ampla gama de políticas de escalonamento para diferentes necessidades de aplicações em tempo real usando MA.

Por fim, para a validação da proposta, foram feitas simulações numa rede local para cada tipo de escalonamento adotado, permitindo, assim, uma análise de quais seriam as melhores escolhas considerando diferentes tipos de cenários de aplicação.

5.2 MODELO

A arquitetura proposta visa à criação de uma camada de *software* que possibilite ao *middleware* JADE dar suporte a escalonamento em tempo real, utilizando a política *best effort*. Para tanto, nesta seção, serão apresentados o modelo de execução de MA, a arquitetura proposta e os algoritmos suportados para esse *middleware*.

5.2.1 Modelo de Execução de Agentes Móveis em Tempo Real

Neste trabalho, considerou-se um conjunto de computadores conectado em uma rede, sendo que cada *host* dessa rede possui o *middleware* RT-JADE, no qual os MAs podem migrar, executar sua missão e, por fim, partir para outro *host* (Figura 5.1).

Nesse contexto, uma missão é composta por um conjunto de recursos que devem ser executados e um itinerário é definido como uma sequência de *host*s que um agente deve visitar para consumir esses recursos e, assim, cumprir a missão de uma aplicação. Uma vez que cada *host* é capaz de receber a quantidade de MAs que puder, limitada apenas pela sua capacidade de memória e processamento, para aplicações de tempo real, é necessário colocar esses agentes em uma fila, seguindo alguma política de escalonamento, para definir a ordem de utilização do recurso (de uso exclusivo) por parte dos agentes. Por exemplo, na política EDF, agentes com

deadlines absolutos mais próximos, mesmo chegando mais tarde, podem ser posicionados no início da fila.

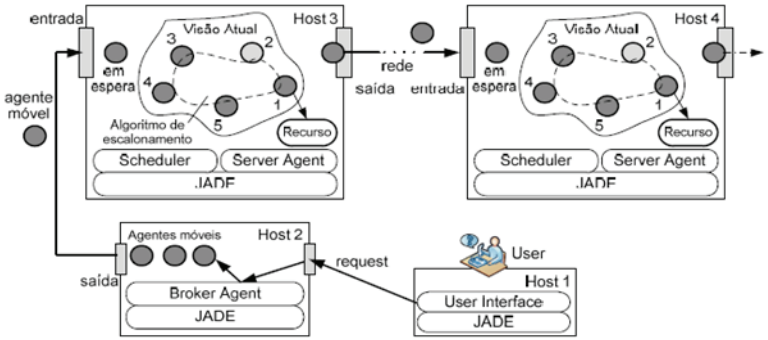

Figura 5.1 – Modelo de execução da arquitetura RT-JADE.

Por sua vez, a interação entre cliente e servidor é efetuada através da utilização de MA, porém de forma transparente ao usuário. Ainda, assume-se, por simplicidade, que cada *host* possui um único recurso e que tal recurso seja de uso exclusivo ao MA que o está utilizando no momento, podendo este ser somente utilizado por outro agente após o anterior o liberar (seja por conclusão da missão ou por preempção do escalonador). Devido à dinâmica de entradas e saídas de agentes em um *host*, a solução de escalonamento de agentes não é trivial.

Nesta proposta, adotou-se o uso de "visões", que indicam o conjunto de MAs em um *host* em um determinado instante. Assim, os agentes em uma $visão_i$ (visão atual) são escalonados de acordo com um algoritmo de escalonamento para definir a ordem de utilização do recurso por parte desses agentes. Durante a $visão_i$, os agentes escalonados podem desistir de esperar pelo recurso e, como consequência, desistir da fila, deixando o *host* (ou migrar para outro *host* menos "sobrecarregado" ou voltar para o *host* origem), sendo que a desistência de um *host* ocorre quando o agente verifica a impossibilidade de concluir sua missão dentro do seu *deadline*. Também durante a $visão_i$, pode ocorrer a entrada de novos MAs no *host* (agente em espera no *host* 3 da Figura 5.1); esses agentes ficam fora da $visão_i$ e, portanto, não são escalonados para usar o recurso no momento.

Além disso, o algoritmo de escalonamento (*host* 3 da Figura 5.1) é executado uma única vez para uma determinada visão, ocorrendo uma mudança de visão (para $visão_{i+1}$) quando um agente libera o recurso; assim,

após esse agente deixar o *host*, verifica-se se houve a entrada de novos agentes no *host* e uma nova visão (a $visão_{i+1}$) é iniciada. Nesse ponto, se houver pelo menos um agente em espera, a nova visão ($visão_{i+1}$) é estabelecida e o algoritmo de escalonamento é executado para definir a nova ordem nessa nova visão. Dependendo da política de escalonamento adotada, a ordem pode ser mantida para os agentes membros da visão anterior, colocando os novos agentes ao final dessa fila. Já para o caso de preempção, quando um MA é pausado, ele é inserido no início da visão ordenada e uma nova visão é criada a partir desta.

O uso de "visões" tem como vantagem permitir o dinamismo no escalonamento dos MAs. Por exemplo, supondo que um *host* possua *n* recursos e que migram para esse *host* diferentes MAs – cada um com o objetivo de utilizar um determinado recurso –, podendo alguns deles terem um objetivo em comum (executar um mesmo recurso), o uso de visões permite separar tais MAs em grupos de recurso, ou seja, os MAs que desejam utilizar o recurso 1 estarão no grupo 1; os que desejam utilizar o grupo 2, no grupo 2 e assim por diante; permitindo, ainda, que cada grupo seja escalonado com uma política de escalonamento diferente.

Conforme ilustrado na Figura 5.2, possui-se *n* recursos e *n* grupos de MAs, todos agrupados de acordo com o recurso R_i que desejam acessar. Cada grupo faz o controle de suas próprias visões, inclusive com a possibilidade de um grupo utilizar um algoritmo de escalonamento distinto ao de outro grupo. No entanto, se, ao invés dos membros da visão, o escalonamento fosse executado por um líder centralizado, ele teria a responsabilidade pelo escalonamento de todos os grupos e, caso esse líder viesse a falhar, todos os grupos estariam comprometidos.

Portanto, ao chegar ao *host*, o MA fica em estado de espera (Figura 5.2), para que, na próxima mudança de visão, possa ser alocado em um dos *n* grupos, de acordo com o recurso R_i que deseja utilizar.

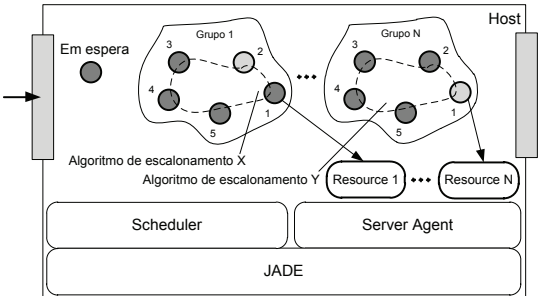

Figura 5.2 – Conceito de visão por política de escalonamento.

5.3 ARQUITETURA

A arquitetura consiste, basicamente, de três agentes estacionários (*user interface*, *broker agent* e *server agent*) e dos MAs visitantes (Figura 5.3). Ainda, como um todo, o sistema possui apenas um *broker* e até ***n*** *server agents*, sendo que os agentes, bem como o *scheduler*, rodam sobre a plataforma JADE.

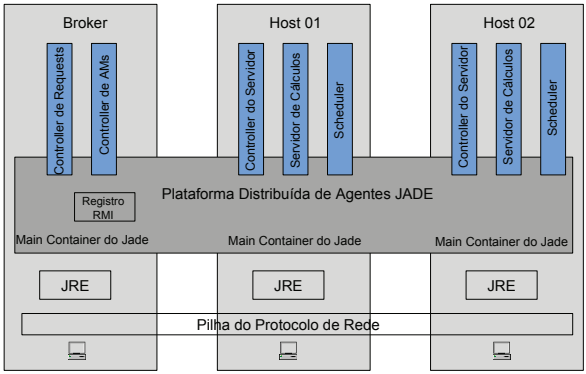

Figura 5.3 – Arquitetura proposta sobre a plataforma JADE.

5.3.1 Interface para Usuários (*User Interface*)

Trata-se de um agente estático que simula o comportamento de até *n* usuários distintos, cada um com missões distintas; denomina-se missão um conjunto de tarefas requisitadas pelo usuário para serem cumpridas por um único MA. Nesse sentido, alguns usuários possuem uma requisição e outros, duas ou mais requisições dentro de uma mesma missão, existindo casos em que pode haver mais de um usuário com uma determinada quantidade *n* de requisições idênticas, porém suas missões sempre serão distintas.

5.3.1.1 Funcionamento

A principal função da *User Interface* (UI) é enviar missões e receber respostas *ACLMessage*[1], no padrão FIPA ACL (FIPA, 2002), do agente estacionário *broker agent*.

[1] Tipo de mensagem dos agentes JADE baseado nas especificações FIPA ACL.

Ao iniciar a interface, a primeira coisa a se escolher é o número de usuários e o tipo de escalonamento a serem simulados. Ao efetivar a escolha, a interface gera *n* conjuntos de requisições (missões) para os *n* usuários escolhidos e associa o(s) tipo(s) de recurso(s) necessário(s) para a completude de cada requisição, juntamente ao escalonamento a ser empregado na simulação corrente.

A UI envia as missões ao *broker agent* no formato de tupla <*CREQ(n), REC(CREQ(n)), SCH*>, sendo:

- *CREQ(n):* conjunto de *n* requisições;
- *REC(CREQ(n))*: recursos necessários para a completude da requisição na posição *n;*
- *SCH*: tipo de escalonamento a ser empregado na simulação.

Na sequência, aguarda o resultado da missão e grava em um arquivo txt: o tipo de escalonamento, o número de usuários da simulação (consequentemente, o número de MAs), o *start time* (momento em que lançou a missão), o *deadline*, o *Round-Trip Time* (RTT, que é o tempo total que o MA levou para fornecer a resposta) e o cumprimento do *deadline* (*accomplish*, que pode ser *YES* ou *NO*).

5.3.2 Agente Centralizador (*Broker Agent*)

O *Broker Agent* (BA) é um agente estático que tem como função receber requisições dos usuários (*host* 1 e *host* 2 da Figura 5.1) e criar um MA para cada tupla de requisições (missão) recebida. Note que um usuário pode ter mais de uma requisição dentro da missão, porém nunca mais de um MA (salvo casos em que o usuário faz outra requisição após a primeira, já enviada).

Tem, também, como função definir os elementos de prioridade (*deadline* e credencial) dos MAs, conforme o tipo de escalonamento utilizado. No caso do *deadline*, é possível configurá-lo para que escolha valores dentro de uma determinada faixa ou valores fixos. Já para o caso de prioridade por credencial, as credenciais podem variar de 0 a 7, sendo 7 a maior prioridade e 0 a menor. Assim, após criar o MA, o BA fornece a ele uma lista de nodos interligados diretamente ao *host* em que o BA está sendo executado; com isso, não há planejamento prévio de itinerário total por parte do MA – o que chamamos de agente míope.

Por fim, o BA tem também por finalidade aguardar o retorno dos MAs e informar os resultados da missão ao usuário da aplicação através da UI.

5.3.3 Agente Servidor (*Server Agent*)

O *Server Agent* (SA) é um agente estático para a administração dos recursos oferecidos pelo *host* e tem como funções: o recebimento de MAs, a comunicação com esses MAs (para o fornecimento de tempo estimado de espera para utilizar o recurso), a escolha de um líder para a *visão,* entre os MAs contidos nela, e o provimento de acesso ao recurso exclusivo.

Sua estrutura interna (Figura 5.4) é composta de: um Identificador (ID) remoto, provida pelo JADE, que permite a agentes remotos comunicarem-se; um ID local, provida pelo JADE, que permite a agentes locais comunicarem-se; uma interface de comunicação com especificações da FIPA (2002), provida pelo JADE, que permite que agentes que seguem as especificações FIPA comuniquem-se, independentemente da plataforma ou linguagem de programação; a lista de recursos disponíveis no *host;* uma fila de agentes para comportar os MAs no *host;* um *membership service* para fins de tratamento de *visões*; e a lista de *hosts* interligados ao *host* corrente.

Além disso, o SA possui um servidor de cálculos instanciado, que contém a identificação do banco de dados (*DataBase* – DB), para fins de acesso (leitura, escrita, atualização etc.) ao histórico e outros tipos de dados.

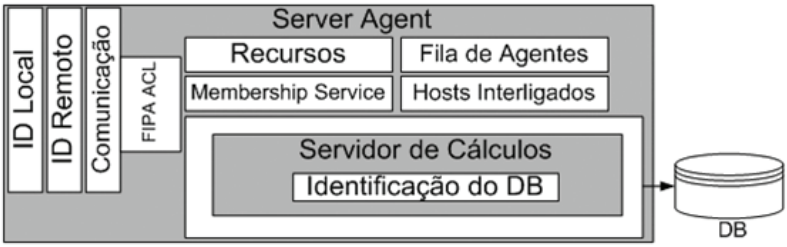

Figura 5.4 – Estrutura interna do SA.

5.3.3.1 Funcionamento

5.3.3.1.1 Quanto ao tratamento de MAs

Ao receber um MA no *host*, se o recurso exclusivo estiver livre, o MA recebe a autorização a seu acesso imediatamente, sendo que os demais MAs que chegarem ao *host* são enfileirados pelo SA na fila de agentes e informados sobre o(s) recurso(s) disponível(eis) no nodo. Já no servidor de cálculos, a função *getCimed()* é invocada, recebendo como parâmetros o endereço e a posição na fila do MA que foi recebido no nodo naquele instante, bem como o número de requisições e tipos de recursos a serem utilizados por ele, permitindo ao *host*, com isso, informar aos MAs um tempo de computação estimativo a ser gasto nos recursos necessários e disponíveis no nodo (dado por (1)) e a estimativa de tempo total a ser gasto pelo MA no *host* (incluindo o tempo de espera na fila), fornecido por (2). Assim, tem-se:

$$Tc_k = \sum_{i=0}^{n} Tc_R[i] * N_R[i] \qquad (1)$$

Sendo:

- Tc_k: tempo estimado de computação total para o agente;
- n: tamanho total da fila de recursos disponíveis no nodo;
- i: posição do recurso na fila de recursos;
- $Tc_R[i]$: tempo de computação estimado (dado por históricos anteriores) do recurso R na posição i;
- $N_R[i]$: quantidade de requisições a serem realizadas pelo recurso R na posição i da fila.

$$W_{k'} = \sum_{i=0}^{Pos(k)-1} W_i \qquad (2)^2$$

$$W_k = W_{k'} + Tc_k$$

Sendo:

- $W_{k'}$: tempo estimado que o agente k terá que esperar na fila para

[2] A expressão (2) pode ser descrita da seguinte forma: quando um MA chega ao *host* e é enfileirado, o servidor de cálculos soma o tempo estimado de espera na fila de todos os MAs que já estavam enfileirados antes dele. O resultado final dessa soma é adicionado ao tempo de computação estimado que esse MA gastará ao utilizar o recurso (baseado em suas requisições).

poder utilizar o recurso;

- W_k: tempo total estimado que o agente *k* ficará no *host*;
- *Pos(k)*: posição do agente móvel *k* na fila;
- *i*: demais MAs na fila;
- W_i: tempo estimado de espera na fila dos demais MAs;
- Tc_k: tempo de computação estimado para que o agente *k* utilize todos os recursos necessários disponíveis no nodo – dado em (1).

Há, também, a possibilidade de um MA deixar o nodo antes de utilizar o recurso (a ser explicado no funcionamento dos MAs); caso isso ocorra, seu tempo é desprezado em (2).

5.3.3.1.2 Quanto ao Escalonamento

Após informar o tempo estimado a todos os MAs da fila, o SA seleciona aleatoriamente um líder entre os MAs da *visão* (*leader*), o qual será responsável pelo escalonamento dos MAs naquela *visão* e pelo fornecimento da visão ordenada ao SA. Dessa forma, o SA recebe a *visão* já ordenada e libera o recurso para o MA mais prioritário. Quando esse MA termina, se a *visão* não mudou, o SA libera o acesso ao recurso para o próximo MA; quando a *visão* muda, o SA elege um novo *leader* para a próxima *visão* e o procedimento descrito repete-se.

Para uma melhor visualização, a Figura 5.5 ilustra o diagrama de colaboração das entidades, desde a escolha do *leader* até o envio do MA mais prioritário ao recurso.

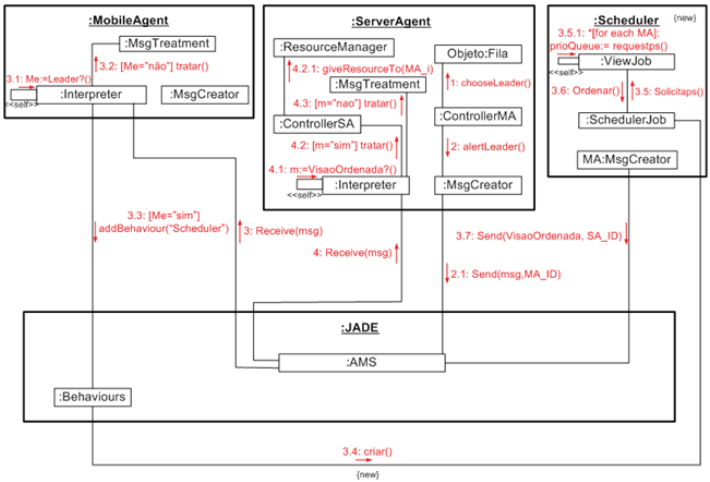

Figura 5.5 – Diagrama de colaboração para escalonamento.

Na Figura 5.5, o SA é representado pelo bloco **:ServerAgent**; o MA, pelo bloco **:MobileAgent**; o *scheduler*, pelo bloco **:Scheduler**; e a plataforma JADE, pelo bloco **:JADE**. Por sua vez, as funções necessárias para cada fase do processo são representadas pelos blocos internos; as mensagens, por um número de sequência que indica a ordem e profundidade em que cada uma ocorre; e as setas representam a direção das mensagens.

Seguindo a ordem das mensagens, a descrição do funcionamento é a seguinte: no bloco **:ServerAgent**, o :ControllerMA escolhe aleatoriamente um MA da fila de agentes – representada por Objeto:Fila – através da função *chooseLeader()*. Após isso, o :MsgCreator cria uma mensagem nos padrões FIPA ACL (FIPA, 2002) e envia ao agente centralizador do JADE – denominado *Agent Management System* (AMS) –, para que seja encaminhada ao MA escolhido como *leader*.

Cabe ressaltar que o AMS possui responsabilidades na plataforma JADE, como registro de novos agentes, contêineres e entrega de mensagens para seus respectivos destinatários, de modo que não é necessário que todos os agentes saibam o endereço uns dos outros. Dessa forma, o AMS encaminha a mensagem, que é interceptada pelo :Interpreter do MA correspondente, que verifica se a mensagem contém padrões de "alerta de liderança", ou seja, se o MA está sendo comunicado de que é o *leader*. Caso o MA não seja *leader*, a mensagem é encaminhada ao :MsgTreatment para que seja tratada, mas, caso o MA seja o *leader*, o :Interpreter envia uma requisição ao :Behaviours do JADE para a adição de um novo comportamento ao MA; nesse caso, o comportamento *scheduler*.

Nesse sentido, cada agente JADE possui uma série de comportamentos (*behaviours*), que podem ser estáticos ou dinâmicos; comportamentos estáticos são aqueles definidos na criação do agente, enquanto comportamentos dinâmicos são aqueles adicionados ao agente em determinado momento. Para que um comportamento seja adicionado a um agente e funcione corretamente, deve ser reconhecido pela plataforma JADE; portanto, :Behaviours possui essa função.

Uma vez requisitado, :Behaviours cria uma nova instância do comportamento requisitado – nesse caso, uma nova instância de **:Scheduler**, com a política de escalonamento a ser empregada – e **:Scheduler** passa a ser um novo *behaviour* do MA *leader*. Ainda, **:Scheduler** possui a função :SchedulerJob, que solicita que :ViewJob obtenha os elementos de prioridade de todos os membros da visão, sendo esses elementos a política de escalonamento empregada.

Ao obter os elementos de prioridade, :ViewJob requisita a :SchedulerJob a ordenação dos membros da visão, com base nesses elementos. :SchedulerJob, por sua vez, atribui a prioridade respectiva a cada MA. O

:Scheduler, então, utiliza o :MsgCreator do :MobileAgent que o instanciou para enviar ao :ServerAgent a visão ordenada. Da mesma forma que todas as mensagens, esta mensagem passa pelo AMS do JADE, para então ser redirecionada ao **:ServerAgent**.

Por fim, o :Interpreter do **:ServerAgent** verifica se a mensagem equivale a uma visão ordenada ou não. Se a mensagem recebida for de outra natureza, é encaminhada ao :MsgTreatment para ser tratada; se não, o :ControllerSA identifica o MA mais prioritário e encaminha ao :Resource-Manager para que o MA receba a autorização para uso do recurso.

Vale ressaltar que cada comportamento JADE possui características definidas pelo programador. Entre elas, estão as de definição de tempo de vida do comportamento, podendo ser periódico (executado a cada *n* tempos), infinito (até a destruição do agente) ou único (executando apenas uma vez e deixando de existir). No caso do **:Scheduler**, ele é um *behaviour* do tipo único, ou seja, após a tarefa de escalonamento ser completada, o MA deixa de possuir o comportamento.

As Figuras 5.6 e 5.7 ilustram o diagrama de classes do RT-JADE e suas ligações com a plataforma JADE.

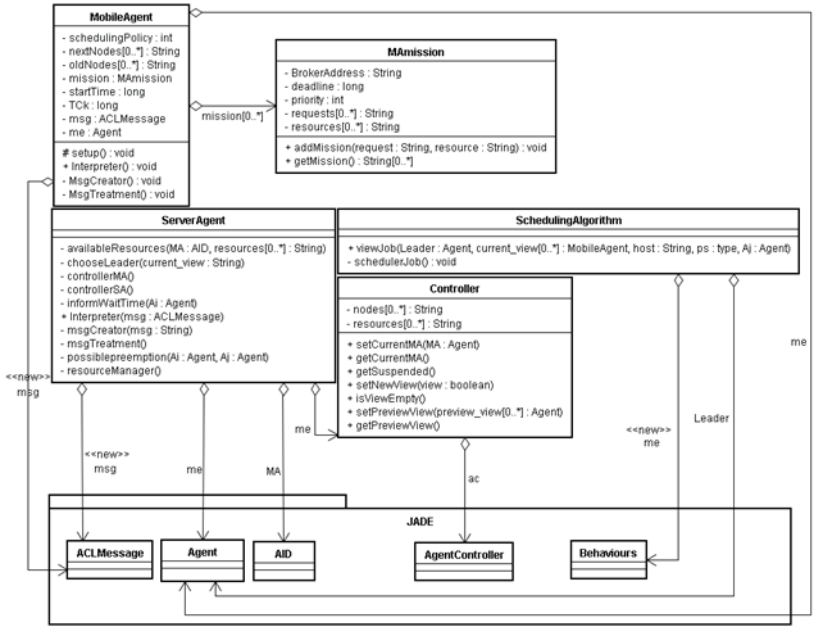

Figura 5.6 – Diagrama de classe.

O MA é ilustrado pelo bloco **MobileAgent**; o SA, pelo bloco **ServerAgent**; o *scheduler*, pelo bloco **SchedulingAlgorithm**; e a plataforma JADE, pelo bloco **JADE**. Já as classes **MobileAgent** e **MAmission** são pertencentes ao MA.

Na classe **MobileAgent**, o atributo *schedulingPolicy* identifica o índice numérico referente à política de escalonamento a ser empregada, permitindo, assim, que conjuntos de MAs com uma mesma política de escalonamento sejam alocados em um mesmo grupo dentro de uma visão. O atributo *nextNodes[0..*]* contém o endereço dos próximos nodos para os quais o MA pode migrar a partir do *host* corrente; *oldNodes[0..*]* contém os nodos que o MA já passou e também os nodos que poderiam ter sido escolhidos anteriormente, mas não foram utilizados; e o atributo *mission[0..*]* engloba a classe **MAmission**, que contém: o endereço do *host* do BA, o *deadline* para a completude da missão, a credencial de prioridade (se for o caso da política de escalonamento ser *priority-based*), o conjunto de requisições a serem atendidas, bem como o conjunto de recursos a serem utilizados para que cada requisição seja satisfeita.

Para fins de controle de *deadline*, a variável *start time* é iniciada assim que o MA parte do *broker*; já a variável *TCk* guarda o tempo previsto de permanência no *host* (fornecido pelo **ServerAgent**). Ainda, a comunicação do MA com outros agentes necessita da criação de uma instância da classe *ACLMessage* do **JADE** para cada mensagem que for transmitir. Assim, uma vez que todo agente JADE (quer móvel ou não) deve estender (*extends*) a classe *Agent* do **JADE**, o **MobileAgent** tem sua identidade salva na variável *me*. Por sua vez, a função *setup()* é necessária para todo MA JADE, pois é nela que se encontram os comportamentos e ações do agente.

O bloco **ServerAgent** possui uma classe **Controller**, responsável pela administração do *host*, recursos, visões e MAs no *host*. Nesse contexto, a função *availableResources()* retorna ao MA os recursos disponíveis no nodo; a *chooseLeader()*, conforme explicada anteriormente, escolhe o *leader* dentro da visão para efetivar o escalonamento; e as funções *controllerSA()*, *controllerMA()* e *resourceManager()* são utilizadas como "ponte" para a classe **Controller**. Já o tempo estimado de permanência de um MA no *host* é informado pela função *informWaitTime()* e, quando o escalonamento é preemptivo, o **ServerAgent** informa ao MA recém-chegado no *host* a identificação do MA através da função *possiblepreemption()*, assim como o MobileAgent.

Por sua vez, o **SchedulerAlgorithm** fica na máquina do **ServerAgent**, sendo instanciado como um novo JADE *behaviour* pelo *leader*.

A classe *Agent Identification* (AID) do **JADE** é utilizada para a identificação de agentes, como nome, local etc., sendo que todo agente JADE estende *Agent* e possui um AID. Ainda, para que o **Controller** possa ter acesso aos MAs, é necessário que instancie a classe *AgentController* do JADE.

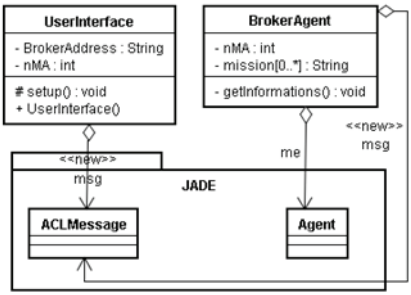

Figura 5.7 – Diagrama de classe da UI e BA.

Conforme ilustrado na Figura 5.7, a **UserInterface** possui como atributos o endereço do nodo *broker* e a quantidade de MAs que irá simular, fornecida pelo usuário no momento em que roda a aplicação. Para que se comunique com o **BrokerAgent**, é necessário que a **UserInterface** crie uma nova instância da classe *ACLMessage* para cada mensagem que deseja transmitir.

Já o **BrokerAgent** recebe da **UserInterface** a quantidade de MAs a ser criada, bem como suas respectivas missões, separadas pela função *getInformations()*. Da mesma forma que a **UserInterface**, o **BrokerAgent** necessita criar uma nova instância da classe *ACLMessage* para cada mensagem que deseja transmitir.

5.3.3.1.3 Quanto à Administração do(s) Recurso(s) e do Histórico

No que diz respeito ao acesso ao recurso, este é feito por exclusão mútua – então chamar-se-á recurso exclusivo –, sendo que a(s) identificação(ões) do(s) recurso(s) exclusivo(s) disponível(is) no nodo é(são) enviada(s) ao MA numa estrutura de dados do tipo fila.

Cada vez que um recurso é utilizado por um MA, seu tempo de computação, bem como o número de requisições atendidas, é salvo no DB[3], para que, posteriormente, sirva de base para os cálculos estimativos representados por (1) e (2).

Além disso, o DB também possui o histórico de todos os MAs que estiveram no nodo e permaneceram na *visão*, ou seja, não desistiram de usar o(s) recurso(s) (Quadro 5.1).

Variável	Definição
ArrivalTime	Tempo em que o MA chegou ao *host* do SA.
StartTime	Tempo em que o MA iniciou o uso do recurso.
EndTime	Tempo em que o MA deixou o recurso.
CompTime	Tempo de computação, calculado por *StartTime - EndTime*. A média de todos os *CompTime* gravados é enviada ao MA como tempo estimado de utilização do recurso.
ConcTime	Tempo de espera total no SA, calculado por *ArrivalTime - EndTime*.

Quadro 5.1 – Histórico dos MAs.

5.3.4 Agente Móvel (*Mobile Agent*)

Cada MA recebe do BA uma missão a ser executada e um tempo máximo para o seu cumprimento (*deadline*) e, caso o tipo de escalonamento exija, recebe também uma credencial. Após isso, o MA migra para um dos *hosts* interligados diretamente ao BA; ao chegar ao *host*, recebe do SA o tempo estimado de espera na fila, juntamente ao tempo estimado de utilização do recurso exclusivo. Se o tempo estimado de espera na fila for maior ou igual a 75%[4] de seu tempo restante para retornar ao BA sem ultrapassar seu *deadline*, e no nodo anterior ao corrente havia(m) outro(s) nodo(s)

[3] No caso do RT-JADE, utilizou-se o PostgreSQL.

[4] Esse valor foi estipulado baseando-se na observação do RTT dos MAs em sucessivas simulações. Notou-se que, para a arquitetura proposta, o tempo de espera máximo para que um agente cumpra seu *deadline* deve ser inferior ou igual a 75% do tempo restante até a sua expiração.

interligado(s) além do nodo escolhido, o MA solicita a este(s) nodo(s) o estado de sua fila. Ao receber esse estado, constata a mesma condição verificada no nodo corrente; se for menor ou igual a 75% de seu tempo restante, migra para o nodo com melhor resultado e, caso contrário, retorna ao BA com missão parcialmente cumprida (caso já tenha utilizado algum recurso anteriormente) ou não cumprida (caso seja a primeira utilização de um recurso).

Vale ressaltar que o MA deve aguardar a autorização por parte do SA para utilizar o recurso. Ao obter a autorização, informa ao SA a missão correspondente ao recurso disponível no nodo e aguarda pelo resultado. Quando o MA conclui a missão, o SA informa a ele a lista de *hosts* interligados diretamente ao nodo, se houverem, mas, se ainda não tiver completado sua missão, ele escolhe aleatoriamente um *host* da lista recebida e migra para ele, repetindo o processo até a finalização da missão ou a aproximação do *deadline*. Após isso, retorna ao BA e informa o resultado obtido, autodestruindo em seguida.

É digno de nota que sua missão é tida como cumprida apenas se for concluída antes do *deadline* estipulado, podendo ser cumprida totalmente (quando todos os recursos necessários foram utilizados) ou parcialmente (quando apenas parte dos recursos foram utilizados). Já quando um MA é selecionado para ser o *leader*, ele instancia o *scheduler* da política de escalonamento escolhida e comunica ao SA o primeiro MA a ser liberado para a utilização do recurso na *visão* corrente.

5.3.5 Escalonador (*Scheduler*)

É um comportamento (JADE *behaviour*) que pode ser inicializado por qualquer agente JADE, móvel ou não, através da chamada *new Scheduler(<agentID>, <current view>,<Server Host>)*, para escalonamentos não preemptivos, e *new Scheduler(<agentID>, <current agent>, <current view>, <Server Host>)*, para escalonamentos preemptivos; sendo:

- *agentID:* a identificação local do *leader*;
- *current agent:* o MA que está utilizando o recurso;
- *current view:* a *visão* atual dos MAs;
- *Server Host:* o endereço do nodo corrente.

O *scheduler* (*host* 3 da Figura 5.1) ordena os MAs da *visão*, de acordo com a política de escalonamento empregada, e fornece ao SA, através do

leader, a identificação do MA mais prioritário para utilizar o recurso. Além disso, utiliza a comunicação FIPA ACL do JADE e possui como ID remoto e local os mesmos do MA que o instanciou (Figura 5.8).

O *scheduler* recebe do SA a *visão* corrente e a insere em sua fila de agentes, de modo que cada política de escalonamento possui um elemento de prioridade distinto (por exemplo, DM = *deadline*; PRIO = credencial), que servirá de base para a ordenação, guardando-se a visão ordenada para informá-la ao servidor posteriormente.

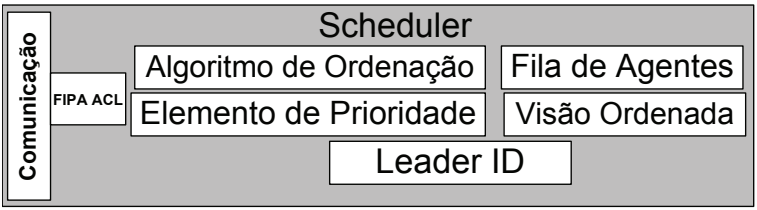

Figura 5.8 – Estrutura interna do *scheduler*.

5.3.5.1 Algoritmos de Escalonamento

Conforme explicado na seção 0, o escalonamento é feito apenas para os MAs pertencentes à *visão* atual ($visão_i$). Além disso, assume-se a sincronização de relógios.

Na *visão* atual, o SA elege aleatoriamente um dos MAs contidos na *visão* para ser o seu líder, o qual passa a ter a responsabilidade sobre o escalonamento de si e dos demais MAs da *visão* atual, requisitando a cada um deles o elemento de prioridade correspondente ao tipo de escalonamento definido pela aplicação.

No exemplo da Figura 5.9, o *leader* requisita o elemento de prioridade a **todos** os MAs (A1, A2, ..., An) contidos na *visão* atual e, após receber as respostas, seleciona o MA com maior prioridade, baseada na política de escalonamento empregada. Então, o agente mais prioritário recebe autorização para utilizar o recurso exclusivo (neste exemplo, o MA A2 é o mais prioritário) e o *leader* informa ao SA a sua identidade. Após o MA terminar sua missão no *host* corrente, um novo *leader* é escolhido para a próxima visão.

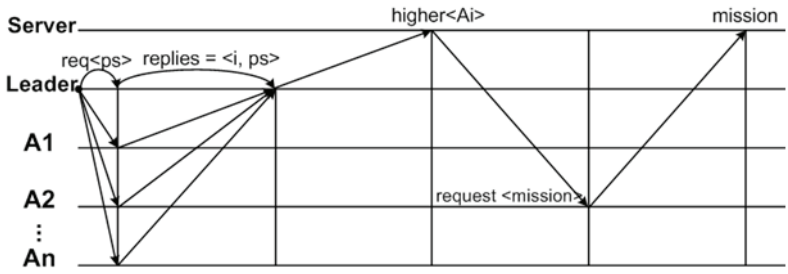

Figura 5.9 – Escalonamento baseado em visões.

Caso ocorra de um *leader* sofrer um *crash*, uma nova *visão* é iniciada e um novo *leader* é escolhido para essa nova *visão*, evitando, assim, *crash* em todo o sistema. A Figura 5.10 ilustra o comportamento do *server* no qual o *leader* sofreu um *crash*. Se até o *timeout* o *server* não receber a resposta do *leader* atual, o SA elege aleatoriamente um novo *leader* para iniciar uma nova *visão*. Caso a *visão* não mude, ou seja, não houver a entrada/preempção de MA no *host*, o escalonamento não será necessário. Então, o *leader* da *visão* apenas comunica ao *server* o próximo MA a ter permissão de utilizar o recurso.

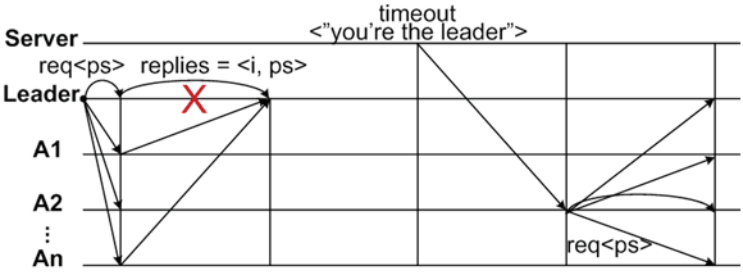

Figura 5.10 – Escolha de novo *leader* após *crash* do anterior.

O algoritmo de escalonamento não preemptivo é apresentado na Figura 5.11, sendo que as notações utilizadas estão no Quadro 5.2. Assim, pode-se observar que o funcionamento do algoritmo executado pelo SA é descrito nas linhas 5 a 19, de modo que, quando o *host* recebe um MA, o SA o enfileira e define como verdadeira a variável *newView* (linhas 6 a 8). No entanto, se o MA deixar o *host* antes de utilizar o recurso, é retirado da fila e *newView* passa a ser verdadeira (linhas 9 a 11). Ainda, caso um recurso

já esteja sendo utilizado, quando liberado, o SA guarda na variável *preview_view* a visão anterior e atualiza a variável *current_view* com a *visão* atual, ou seja, a fila atual de MAs (linhas 12 a 14). Após isso, é realizada uma nova escolha de *leader* para a próxima visão (linha 15).

Como o escalonamento será executado, a variável *newView* é definida como falsa, até que um próximo MA chegue ao *host* (linha 16). Com isso, sempre que um MA liberar o recurso e nenhum outro agente tiver migrado ao *host*, a variável será falsa, garantindo assim, no código do MA, que a função *newView( )* retorne falsa. Mas, caso a visão seja a primeira, ou seja, o algoritmo esteja sendo executado pela primeira vez e ainda não haja *leader*, ele será eleito (linhas 17 a 19).

Por seu turno, o algoritmo do MA (linhas 20 a 26) consiste em verificar se ele é o *leader*, através do recebimento da mensagem "*you're the leader*" (linha 21) enviada pelo SA através da função *chooseLeader*. Caso seja o *leader*, é verificado se há uma nova *visão* (uma *visão* diferente da anterior) ou se não há *visão* anterior (o que indica que é a primeira vez que o algoritmo está sendo executado) (linha 22). Se uma das duas situações anteriores for verdadeira, o MA cria uma instância da classe *scheduler* (linha 23), mas, se ambas forem falsas, o escalonamento não será necessário, por não ter havido alterações no escalonamento anterior; então, se o recurso estiver livre, o MA com maior prioridade é enviado à sessão crítica (linhas 24 a 26).

Já o *scheduler* (linhas 27 a 36) requisita de cada MA da fila (inclusive o que o instanciou) seus respectivos objetos de prioridade p_s (linhas 28 e 29), que podem ter somente o *deadline; deadline* e credencial; ou somente credencial, para o caso do escalonamento de prioridades em que o *deadline* só é utilizado para fins de avaliação de desempenho. Após isso, o *scheduler* aguarda as respostas dos agentes por um determinado período t_1 e essas respostas são inseridas em uma fila (linhas 30 a 32). Caso haja *timeout* ($t_1 > t_0$), a variável *replies* terá valor nulo, o qual estará enfileirado na posição correspondente ao MA que deixou de responder dentro do tempo previsto.

O escalonamento é, então, executado ignorando os valores nulos (se houverem) e o MA com maior prioridade é enviado ao recurso exclusivo (linhas 33 a 36). Ainda, a ordenação dos valores recebidos, realizada pela função *schedule()* (linha 35), utiliza o algoritmo *Fast Quick Sort* (AHRENS, 1998) e, para o escalonamento FIFO, o *leader* requisita apenas ao SA que libere o acesso do recurso exclusivo ao próximo MA da fila.

```
Variáveis compartilhadas:
1. Queue = Ø        {fila que comportará os MAs}
2. Leader = Ø       {MA líder na visão}
3. preview_view = Ø  {visão anterior}
4. current_view  = Ø {visão atual}

{Parte Server Agent}
Variáveis locais:
5. newView = false   {indica quando uma visão muda}
Tarefa principal
6. On arrival of an Ai
7. enqueue(Queue, Ai)
8. newView = true
9. On Ai leave before resource use
10. dequeue(Queue, Ai)
11. newView = true
12. On Ai leave after resource use
13. preview_view = current_view
14. current_view = Queue
15. chooseLeader(current_view)
16. newView = false
17. IF (preview_view is empty && Leader is empty) THEN
18. chooseLeader(Queue)
19. ENDIF
{Parte Mobile Agent}
Variáveis locais:
20. sch = Ø          {tipo de escalonamento}
subroutine check(Leader)
21. On receive message “you’re the leader”
22. IF (newView() || preview_view is empty) THEN
23. sch.runScheduler()
24. ELSIF resource is free
25. send higher priority Aj  in Queue to critical section
26. ENDIF
```

```
{Parte Scheduler}
Variáveis locais:
27. prioQueue=Ø      {fila de elementos de prioridade}
subroutine runScheduler()
28. FOR each Aj on current_view DO
29. request ps  for Aj
30. DO
31. replies← wait_replies(Aj)
32. UNTIL (t1> t0)
33. enqueue(prioQueue, replies)
34. ENDFOR
35. schedule()
36. send higher priority Aj  in Queue to exclusive resource
```

Figura 5.11 – Algoritmo de escalonamento não preemptivo.

O algoritmo preemptivo (Figura 5.12) tem funcionamento semelhante ao do não preemptivo, com exceção das linhas:

- 4 a 6 do SA: verifica, no momento da chegada de um MA, se há outro utilizando o recurso. Em caso afirmativo, envia o endereço do agente que utiliza o recurso (A_j) para o agente que chegou (A_i), para que este tente uma possível preempção;
- 19 a 20 do MA: recebe o endereço de A_j no momento e tenta a preempção;
- 36 a 38 do *scheduler*: é realizada a mesma verificação das linhas 24 a 26 do MA.

Nas linhas 39 a 46, o algoritmo de preempção é descrito, de forma que A_i comunica-se com A_j solicitando seu objeto de prioridade p_s (linha 39). Caso A_j seja menos prioritário que A_i, é suspenso e reinserido no início da *visão*, ou seja, da fila *current_view* (linhas 40 a 42). Após isso, A_i passa a utilizar o recurso (linha 43) e, ao terminar, o agente suspenso A_j é reativado, voltando a utilizar o recurso do ponto de onde parou (linhas 45 e 46). As funções de suspensão e reativação de um MA são fornecidas pelo JADE, através dos métodos *suspend()* e *activate()*.

```
Variáveis compartilhadas:
1. Queue = Ø        {fila que comportará os MAs}
2. Leader = Ø       {MA líder na visão}
3. preview_view = Ø  {visão anterior}
4. current_view  = Ø {visão atual}

{Parte Server Agent}
Variáveis locais:
5. newView = false   {indica quando uma visão muda}
Tarefa principal
1. On arrival of an Ai
2. enqueue(Queue, Ai)
3. newView = true
4. IF has an Aj in resource THEN
5. send Aj address to Ai
6. ENDIF
7. On Ai host change
8. dequeue(Queue, Ai)
9. newView = true
10. On leave of an Ai
11. preview_view = current_view
12. current_view = Queue
13. chooseLeader(current_view)
14. newView = false
15. IF (preview_view is empty && Leader is empty) THEN
16. chooseLeader(Queue)
17. ENDIF
{Parte Mobile Agent}
Variáveis locais:
18. sch = Ø          {tipo de escalonamento}
Tarefa principal
19. On receive Aj address
20. sch.tryPreemption(Aj)
subroutine check(Leader)
21. On receive message "you're the leader"
```

```
22. IF (newView() || preview_view is empty) THEN
23. sch.runScheduler()
24. ELSIF resource is free
25. send higher priority Aj  in Queue to critical section
26. ENDIF
{Parte Scheduler}
Variáveis locais:
27. prioQueue=Ø       {fila de elementos de prioridade}
subroutine runScheduler()
28. FOR each Aj on current_view DO
29. request ps  for Aj
30. DO
31. replies← wait_replies(Aj)
32. UNTIL (t1> t0)
33. enqueue(prioQueue, replies)
34. ENDFOR
35. schedule()
36. IF resource is free THEN
37. send higher priority Aj  in Queue to exclusive resource
38. ENDIF

subroutine tryPreemption(Aj)
39. request ps  for Aj
40. IF Aj has lower priority than Ai THEN
41. suspend(Aj)
42. enqueue(current_view(0), Aj)
43. send Ai to exclusive resource
44. ENDIF
45. When Ai leave resource
46. activate(Aj)
```

Figura 5.12 – Algoritmo de escalonamento preemptivo.

Símbolo	Descrição	Símbolo	Descrição
A_i	MA local	t_1	Tempo máximo de uma tarefa
A_j	Demais MAs	***enqueue(q,e)***	Insere na fila *q* um elemento *e*
e	Elemento	***dequeue(q,e)***	Retira da fila *q* um elemento *e*
ps	Elemento de prioridade do escalonamento	***chooseLeader(q)***	Escolhe um líder para a visão
q	Fila	***wait_replies(A_j)***	Aguarda p_s de outros agentes
s	Identificação do escalonamento	***suspend()***	Função JADE
t_0	Tempo mínimo de uma tarefa	***activate()***	Função JADE

Quadro 5.2 – Notações.

5.4 CONCLUSÕES

Este capítulo apresentou uma visão geral da arquitetura da proposta desta dissertação, tendo sido descrito o modelo do sistema, bem como suas funções e os algoritmos de escalonamento implementados. Ainda, discutiu-se a motivação para o uso de *visões* nos algoritmos de escalonamento.

Os testes e análises para a validação da proposta são descritos no próximo capítulo.

6 IMPLEMENTAÇÃO

Neste trabalho, foram analisados dez algoritmos de escalonamento, sendo nove implementados e um (FIFO) provido pelo JADE: FIFO; EDF não preemptivo; *Preemptive* EDF (PEDF); LIFO; PRIO; escalonamento baseado em prioridade preemptivo (*Preemptive Priority Based Scheduler* – PPRIO); DM; *Preemptive Deadline Monotonic* (PDM); SJF; e SRTF ou SJF preemptivo. A descrição das políticas de escalonamento é descrita no Capítulo 2 desta dissertação e também se encontra em Farines, Fraga e Oliveira (2000) e Stankovic et al. (1996).

Toda a arquitetura (MA, BA, SA e algoritmos de escalonamento) foi implementada na linguagem JAVA (JDK 1.6.0_19), utilizando o *framework* JADE (versão 4.0.1). Ainda, o ambiente de teste é constituído de três máquinas, sendo elas: (*i*) Intel Core2Duo 2.4GHz, 1GB RAM, Windows XP Professional 32 bits; (*ii*) Intel Core2Duo 1.6GHz, 1.5GB RAM, Windows XP Professional 32 bits; e (*iii*) Intel Core Quad 3.0GHZ, 4GB RAM, Windows 7 64 bits; as quais foram interligadas com um *hub* em uma rede LAN, e assume-se ambiente controlado.

Foram simulados dois SAs, cada um contendo apenas um único recurso exclusivo; o SA do *host* 1 possui o recurso de busca em base de dados, enquanto o SA do *host* 2 possui o recurso de cálculos gerais, como preço de peças, dimensões, orçamento etc. Já a configuração dos nodos é por ordem de precedência, não sendo possível utilizar o recurso R2 sem antes executar R1 (Figura 6.1), e as faixas de *deadline* foram escolhidas baseadas em históricos de simulações anteriores – conforme descrito em 0 – e carga computacional, o que justifica a escolha distinta de *deadline*s por quantidade de MA concorrentes. Assim, para cinco MAs concorrentes, escolheu-se as faixas de 300, 500, 700 e 1100 ms, enquanto para vinte MAs concorrentes, faixas de 700, 1500, 2000 e 2500 ms.

Além disso, cada MA possui de 1 a 3 requisições distintas em uma missão, para serem utilizadas por ambos os recursos. Tais requisições são escolhidas aleatoriamente, portanto, há a possibilidade de mais de um MA possuir uma mesma requisição.

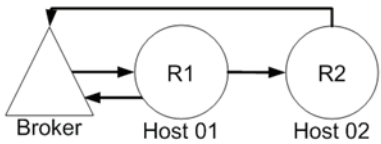

Figura 6.1 – Configuração dos nodos para simulação da arquitetura.

6.1 TESTES

Nesta seção, serão apresentadas as descrições dos testes e análises realizados para validar a proposta desta dissertação. Para fins de esclarecimento:

- utiliza-se a sigla IT para referir-se à iteração;
- cada simulação possui um número *n* de iterações; se em uma simulação foram executados ao todo 400 MAs, então essa simulação denominar-se-á IT400.

6.1.1 Por Cumprimento de Missão (Benefício)

Para este teste, foram executadas 100 iterações para cada política de escalonamento em cada faixa de *deadline*, totalizando 500 MAs para cada escalonamento por faixa de *deadline* em IT500 e 2000 MAs em IT2k, sendo 2k representativo de 2000. Em IT500, 5 MAs concorrentes são lançados por iteração, enquanto, em IT2k, 20 MAs concorrentes são lançados. Cabe ressaltar que o intervalo de cada iteração não ultrapassa 1 segundo após o BA ter recebido todos os MAs da iteração anterior.

Nesse contexto, foram contabilizados o número de MAs de cada escalonamento que cumpriu total (consumindo R1 e R2) ou parcialmente (consumindo apenas R1) suas missões e, também, o número dos que não cumpriram. Para tanto, um consumo é considerado parcial quando um MA, que terminou de consumir o recurso R1, percebe que seu *deadline* está muito próximo e desiste de R2, retornando ao BA e, assim, completando parcialmente sua missão.

Quanto à medida do tempo de viagem do MA, ela inicia no momento em que ele sai do BA (Figura 5.1 e 6.1), sendo que o MA deve deixar o BA, ir para o nodo do recurso R1, aguardar sua vez na fila, usar o recurso, migrar para o nodo do recurso R2, aguardar na fila, usar o recurso e, por fim, retornar ao BA antes do *deadline*.

Por fim, os desempenhos dos algoritmos foram comparados entre si – em especial com o algoritmo FIFO –, para averiguar quais faixas de *deadline* são melhores para cada caso e qual algoritmo atende melhor ao requisito de cumprimento de missão.

6.1.1.1 Resultados Obtidos

O código de cada MA foi compactado pelo JADE em um *Java Archive* (JAR) com 13 Kbytes de tamanho, cuja transmissão efetua a migração de um MA de um *host* para outro. No JADE, o tempo de transmissão desse arquivo ficou em torno de 83,5 ms; já em relação ao consumo dos recursos R1 e R2, os agentes levaram, em média, 6,2 ms para consumir o recurso R1 e 9,2 ms para consumir o recurso R2, sendo que essa média não leva em conta o desvio padrão.

Para as simulações IT500, os resultados são demonstrados na Figura 6.2. Analisando-os, foi possível concluir que, para *deadline*s mais apertados (300 a 500 ms), o algoritmo de escalonamento LIFO apresentou melhor benefício em ambos os casos, comparando completude total da missão, sendo que, na faixa de 300 ms, o escalonamento menos indicado foi o SJF, que ocasionou a perda total da missão para todos os MAs. Já para a faixa de 700 ms, os algoritmos PEDF, PRIO e PPRIO apresentaram os melhores benefícios, sendo o SJF e seu preemptivo SRTF os piores, enquanto, com *deadline*s mais folgados (1100 ms), todos os algoritmos apresentaram resultados satisfatórios, sendo os piores o DM e o FIFO.

É importante salientar que, com base na população de MAs que não cumpriu sua missão (total ou parcialmente), o algoritmo FIFO ficou entre os piores resultados na maioria das faixas de *deadline* escolhidas.

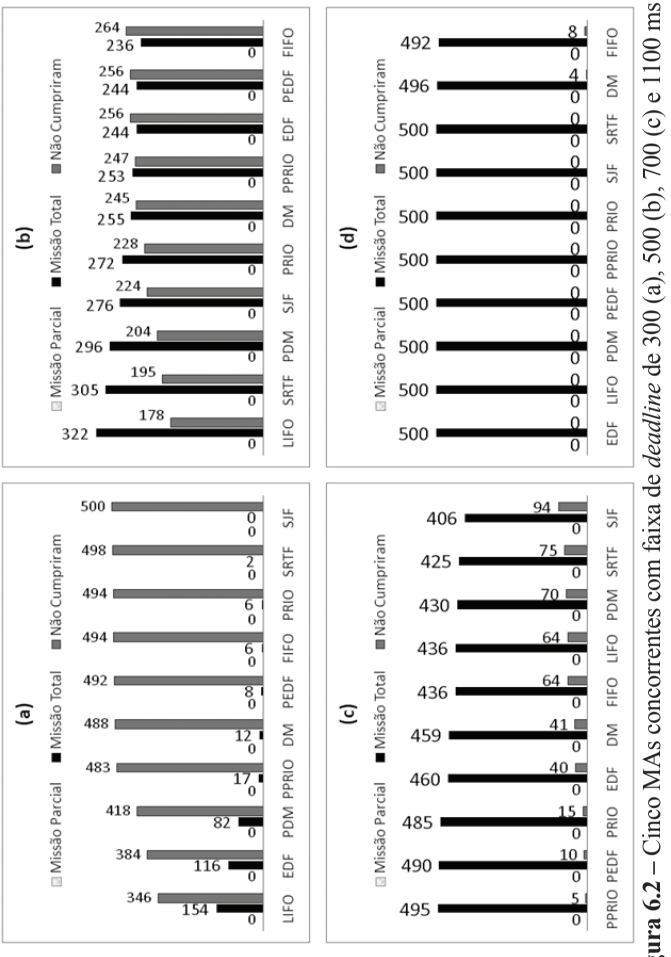

Figura 6.2 – Cinco MAs concorrentes com faixa de *deadline* de 300 (a), 500 (b), 700 (c) e 1100 ms (d).

Para as simulações IT2k, os resultados obtidos são demonstrados na Figura 6.3. Para a faixa de *deadline* de 300 ms, nenhum agente conseguiu cumprir sua missão; por essa razão, os resultados foram desprezados e foi escolhida uma nova faixa. Para a faixa de *deadline* de 700 ms, todos os algoritmos apresentaram baixo desempenho, enquanto, para as faixas de 1500 a 2000 ms, os algoritmos apresentaram uma grande melhora, sendo os mais indicados para 1500 ms PRIO, EDF e PPRIO e, para 2000 ms, PEDF, PDM e EDF, podendo-se, assim, observar que o algoritmo EDF está entre os mais indicados para ambos os casos. Já na faixa mais folgada (2500 ms), os algoritmos que permitiram que todos os MAs cumprissem suas missões foram o DM e o PEDF, sendo o SRTF o segundo e o EDF o terceiro quanto à completude de missão.

Logo após o EDF, surgem os algoritmos PDM e PPRIO; com isso, pode-se observar que, entre os cinco melhores resultados (com cumprimento de mais de 1900 MAs), três foram de algoritmos preemptivos (PEDF, PDM e SRTF) e quatro de políticas de escalonamento diretamente relacionadas com o *deadline*, ou seja, os algoritmos DM e EDF e seus preemptivos PDM e PEDF. Além disso, para essa simulação, a política FIFO obteve o pior resultado em três das quatro faixas escolhidas.

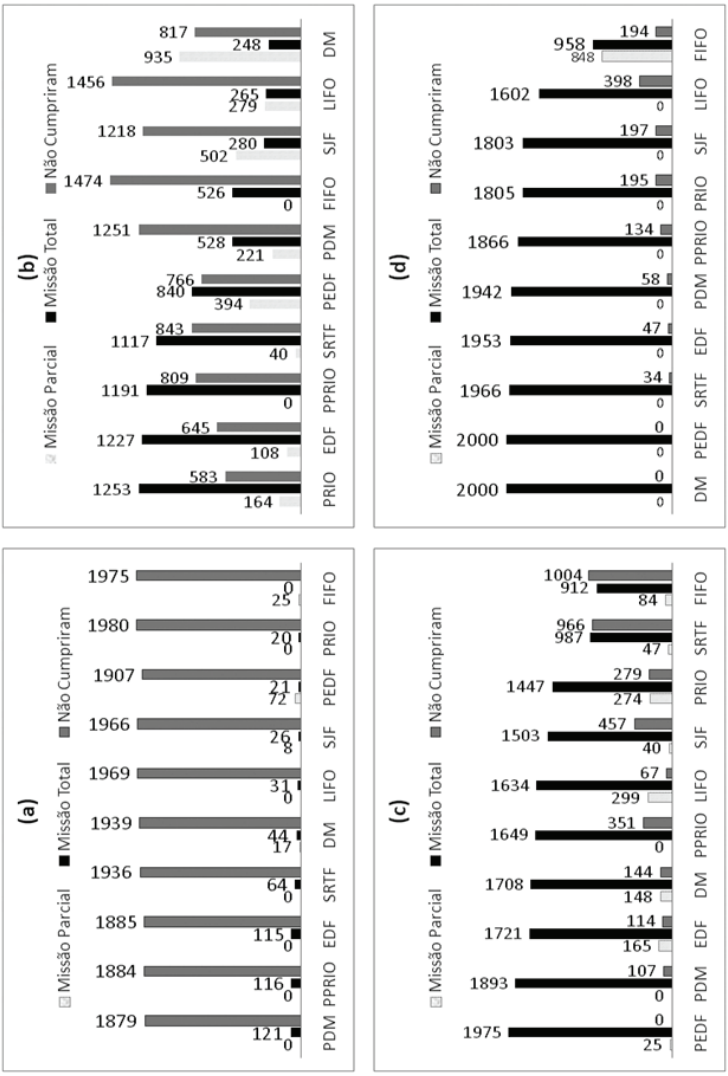

Figura 6.3 – Vinte MAs concorrentes com faixa de *deadline* de 700 (a), 1500 (b), 2000(c) e 2500 ms (d).

6.1.2 Por Carga da Unidade Central de Processamento (Custo)

Para este teste, analisou-se a carga da UCP de cada política de escalonamento, para cinco e vinte MAs concorrentes, com o objetivo de comparar o custo de processamento de cada algoritmo implementado com o algoritmo FIFO.

Nesse contexto, alguns aspectos importantes a serem considerados são:

- a ordenação do algoritmo LIFO é feita através da função *Collections.reverse()* do JAVA, a qual é executada em tempo linear com custo O(n), ou seja, a fila é percorrida por completo;
- os demais algoritmos utilizam *FastQuickSort,* que possui complexidade temporal de $O(n\log n)$, para os casos melhor e médio, e de O(n^2), para o pior caso. Como os MAs são autônomos, sua ordenação na fila não é previsível, portanto não é possível saber qual caso o algoritmo de ordenação tratará.

6.1.2.1 Resultados Obtidos

As medidas de carga de UCP são apresentadas nas Figura 6.4 e 6.5.

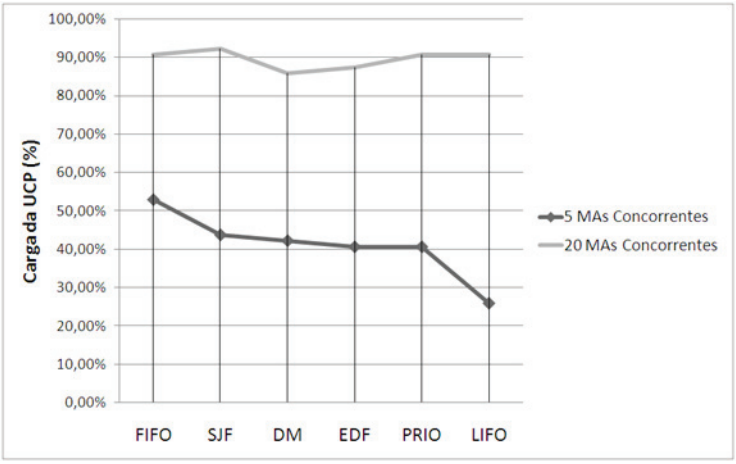

Figura 6.4 – Carga de UCP para algoritmos não preemptivos.

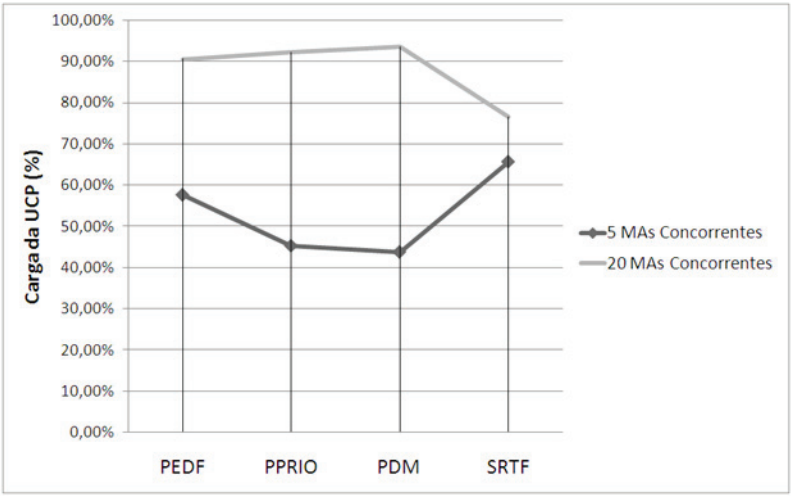

Figura 6.5 – Carga de UCP para algoritmos preemptivos.

Com relação ao custo-benefício dos algoritmos não preemptivos para cinco MAs concorrentes, o algoritmo FIFO apresentou maior uso de UCP, enquanto o LIFO apresentou o menor consumo. Comparando esses resultados ao teste realizado em 0, é possível concluir que o algoritmo LIFO ficou entre os que permitiram a um maior número de MAs cumprir seu *deadline* com a taxa de consumo mais baixa.

Quanto aos algoritmos preemptivos para cinco MAs concorrentes, os escalonamentos utilizando as políticas PDM e PPRIO foram os que apresentaram menor carga de UCP, enquanto o SRTF apresentou a maior carga. Analisando esse resultado com os obtidos em 0, pode-se observar que PDM e PPRIO ficaram entre os melhores em três das quatro faixas de *deadline* escolhidas, enquanto o PEDF e o SRTF ficaram entre os melhores em apenas duas das quatro faixas de *deadline* aplicadas; com isso, conclui-se que, na relação custo-benefício, o algoritmo PDM é o mais indicado, seguido do PPRIO.

Para vinte MAs concorrentes, os algoritmos não preemptivos DM e EDF apresentaram menor consumo de processador, enquanto o maior consumo ficou a cargo do SJF. Comparando esse resultado com o apresentado em 0, nota-se que a melhor relação custo-benefício ocorreu com o uso do algoritmo EDF.

Dos algoritmos preemptivos, o SRTF e o PEDF apresentam menor carga de UCP, sendo o PEDF o mais indicado na comparação custo-be-

nefício, pois representa o melhor resultado em duas das quatro faixas de *deadline*. Já o algoritmo PDM, apesar de apresentar a maior carga de UCP, está em segundo lugar na questão de cumprimento de *deadline*.

Por fim, com relação à complexidade temporal, ao verificarmos os escalonamentos com cinco e vinte MAs, é possível perceber que o consumo do LIFO foi o mais baixo na menor quantidade de MAs concorrentes utilizadas para o teste e ficou entre os mais altos para a maior quantidade de MAs; enquanto, para os algoritmos preemptivos, o SRTF consumiu maior carga para cinco MAs concorrentes, porém obteve a menor carga para vinte. Essas observações talvez tenham se dado pelos aspectos de complexidade algorítmica mencionados no início desta subseção.

6.1.3 Por *Throughput* (Taxa de Transferência)

Para este teste, estipulou-se uma métrica denominada *deadline* interno. Quando um MA chega ao *host*, este recebe um *deadline* interno, que significa o tempo máximo que o MA tem para concluir sua missão no nodo atual e que é atribuído somente para comparar quais algoritmos de escalonamento permitem a um maior número de MAs sair do *host* – após utilização do recurso – em um período de 1 segundo.

Para medir o *throughput*, *deadlines* internos foram estipulados para o servidor e realizou-se a simulação do comportamento de cinco e dez MAs concorrentes. Para cinco MAs concorrentes, os *deadlines* aplicados foram 5, 10, 15, 20 e 25 ms, enquanto, para dez MAs, foram de 2 a 22,5 ms (aumentando em 0,5 ms cada valor, até totalizar 10 valores).

6.1.3.1 Resultados Obtidos

Analisando a Figura 6.6, para cinco MAs concorrentes, tanto o algoritmo FIFO quanto o DM não permitiram que nenhum MA cumprisse sua missão, enquanto as demais políticas de escalonamento permitiram que pelo menos 20% dos MAs cumprissem.

No segundo teste, realizado com dez MAs concorrentes, ao atribuir-lhes *deadlines* de 5 a 50 ms, o algoritmo FIFO permitiu que 40% dos MAs concluíssem a missão. Ao reduzir essa faixa – até que o FIFO não obtivesse mais desempenho –, atribuindo os *deadlines* de 2 a 22,5 ms, conseguiu-se o resultado ilustrado na Figura 6.6, na qual o eixo x representa as políticas de escalonamento empregadas e o eixo y, a porcentagem de MAs que conseguiu deixar o *host* antes do *deadline* interno estipulado.

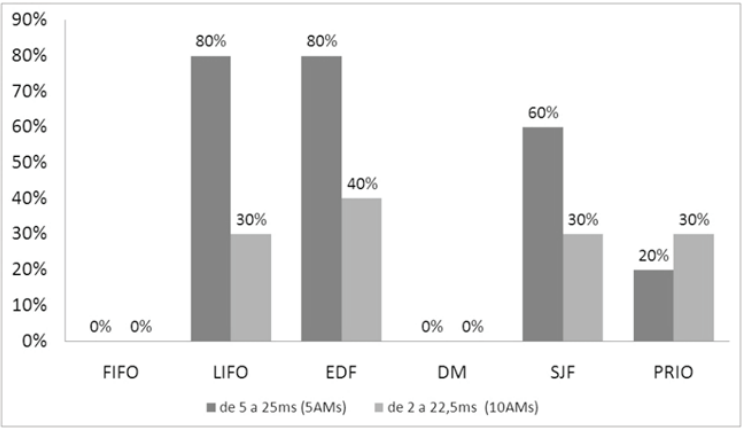

Figura 6.6 – Comportamento das políticas de escalonamento não preemptivo.

Avaliando o cumprimento de missões, é possível perceber que, enquanto a política FIFO não permitiu a nenhum MA completar sua missão, os demais algoritmos (com exceção do DM) permitiram que, no mínimo, 20% dos MAs concluíssem – o mesmo resultado mínimo comparado ao resultado com 5 MAs concorrentes.

A Tabela 6.1 contém o tempo de conclusão (em milissegundos) de cada MA para cinco MAs concorrentes, bem como os respectivos *throughputs* – dados pelo número de MAs por segundo – de cada algoritmo de escalonamento. Para dez MAs concorrentes, o tempo de conclusão de cada MA é demonstrado na Tabela 6.2, sendo que, quando o tempo de conclusão é inferior a 1 ms, este é representado pelo número 0 (zero).

Tabela 6.1 – Tempo de conclusão para 5 MAs concorrentes.

	Políticas de escalonamento (não preemptivo)					
***Deadline* interno (ms)**	***FIFO***	***LIFO***	***EDF***	***DM***	***SJF***	***PRIO***
5	15	16	0	16	16	16
10	31	0	0	47	15	16
15	47	0	15	62	0	32
20	78	15	16	78	0	32
25	31	16	32	78	16	0
Tempo total gasto (ms)	**202**	**47**	**63**	**281**	**47**	**96**
***Throughput* (MAs/s)**	**0**	**85**	**63**	**0**	**64**	**10**

Tabela 6.2 – Tempo de conclusão para 10 MAs concorrentes.

	Políticas de escalonamento (não preemptivo)					
***Deadline* interno (ms)**	***FIFO***	***LIFO***	***EDF***	***DM***	***SJF***	***PRIO***
2	47	15	16	16	15	16
2,5	78	16	16	16	16	16
5	94	15	16	16	16	15
7,5	125	15	32	16	16	15
10	63	16	14	15	31	15
12,5	15	16	31	15	15	15
15	16	16	15	31	15	31
17,5	32	16	15	47	16	15
20	63	16	16	31	32	16
22,5	31	16	15	31	16	0
Tempo total gasto (ms)	**564**	**157**	**186**	**234**	**188**	**154**
***Throughput* (MAs/s)**	**0**	**19**	**22**	**0**	**16**	**19**

Ao analisar a Tabela 6.1, nota-se que, para o algoritmo FIFO permitir que pelo menos um MA cumpra sua missão, a faixa de *deadline* deve ter valor mínimo de 15 ms, enquanto os demais algoritmos (com exceção do DM) permitem que pelo menos um MA cumpra a missão com as faixas atribuídas e com tempo de resposta na ordem dos µs.

Por sua vez, analisando a Tabela 6.2, é possível perceber que o *deadline* mínimo requerido para o algoritmo FIFO é de 15 ms – o mesmo da Tabela 6.1 – enquanto, para o algoritmo EDF, é de 14 ms. Ainda, o algoritmo PRIO permitiu que um MA cumprisse sua missão na ordem dos µs. Apesar de o algoritmo DM não ter permitido a nenhum MA cumprir sua missão, seu tempo total de resposta foi 41,49% menor quando comparado ao tempo total de resposta da política FIFO.

A Figura 6.7, a seguir, ilustra o comportamento dos algoritmos preemptivos em relação ao FIFO, sendo que o eixo x representa as políticas de escalonamento empregadas, enquanto o eixo y representa a porcentagem de MAs que conseguiu deixar o *host* antes do *deadline* interno estipulado.

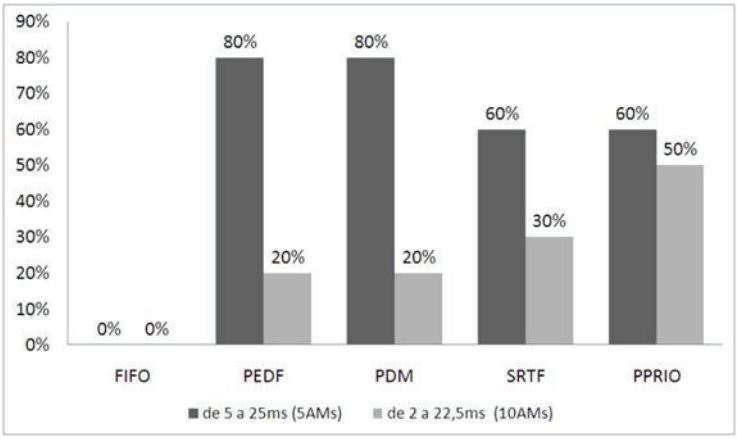

Figura 6.7 – Comportamento dos algoritmos de escalonamento preemptivos.

Analisando a Figura 6.7, é possível observar que, para 5 MAs concorrentes, os algoritmos preemptivos permitiram que um mínimo de 60% dos MAs completasse sua missão, enquanto, para 10 MAs concorrentes, um mínimo de 20% foi obtido.

A Tabela 6.3 contém o tempo gasto por cada MA para cinco MAs concorrentes, enquanto a Tabela 6.4 possui esse tempo para dez MAs concorrentes. Ambas as tabelas contêm os *throughputs* respectivos a cada algoritmo de escalonamento.

Tabela 6.3 – Tempo de conclusão de cada MA – Algoritmos preemptivos com 5 MAs concorrentes.

	Políticas de escalonamento (preemptivo)			
***Deadline* interno(ms)**	***PEDF***	***PDM***	***SRTF***	***PPRIO***
5	0	0	0	16
10	0	15	0	0
15	32	0	16	15
20	15	0	32	31
25	0	15	15	16
Tempo total gasto (ms)	**47**	**30**	**63**	**78**
***Throughput* (MAs/s)**	**8 5**	**133**	**48**	**38**

Tabela 6.4 – Tempo de conclusão de cada MA – Algoritmos preemptivos com 10 MAs concorrentes.

	Políticas de escalonamento (preemptivo)			
***Deadline* interno(ms)**	***PEDF***	***PDM***	***SRTF***	***PPRIO***
2	0	16	16	15
2,5	15	16	15	16
5	15	15	31	16
7,5	15	16	15	31
10	328	15	15	0
12,5	344	16	15	16
15	78	16	31	0
17,5	47	31	16	0
20	62	15	15	0
22,5	15	16	16	16
Tempo total gasto (ms)	**919**	**172**	**185**	**110**
***Throughput*(MAs/s)**	**2**	**12**	**16**	**45**

O *throughput* dos algoritmos de escalonamento é graficamente demonstrado na Figura 6.8.

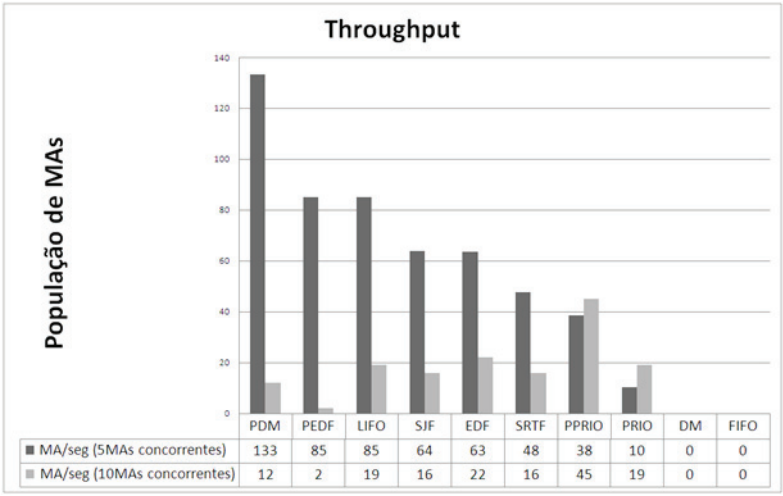

Figura 6.8 – *Throughput* dos algoritmos de escalonamento (MAs/segundo).

Ao observar o gráfico do *throughput*, observa-se que o algoritmo PDM apresentou melhor vazão para cinco MAs concorrentes, enquanto o algoritmo PPRIO apresentou o melhor resultado para dez MAs concorrentes.

Ao comparar esses valores com os testes realizados em 0 e 0, é possível concluir que o algoritmo LIFO apresentou melhor custo-benefício e melhor taxa de transferência dos algoritmos não preemptivos para cinco MAs, enquanto, para os algoritmos preemptivos, o PDM apresentou os melhores resultados na análise conjunta dos três testes. Por sua vez, para dez MAs concorrentes, os algoritmos não preemptivo EDF e preemptivo PPRIO são os de melhor performance no quesito taxa de transferência/custo-benefício.

6.1.4 Por Tempo de Resposta ao Usuário

Para este experimento, foram utilizados os resultados obtidos em 0 e calculadas as médias do tempo de resposta de cada escalonamento para cinco MAs concorrentes. É importante enfatizar que as médias dos tempos adquiridos nesta seção foram somente para os MAs que cumpriram – por completo – suas missões em 0, tendo sido adquiridas através dos valores de RTT contidos no arquivo txt gerado pela UI.

6.1.4.1 Resultados Obtidos

Os resultados obtidos pelos algoritmos não preemptivos são demonstrados na Figura 6.9, enquanto a Figura 6.10 ilustra os resultados para os algoritmos preemptivos.

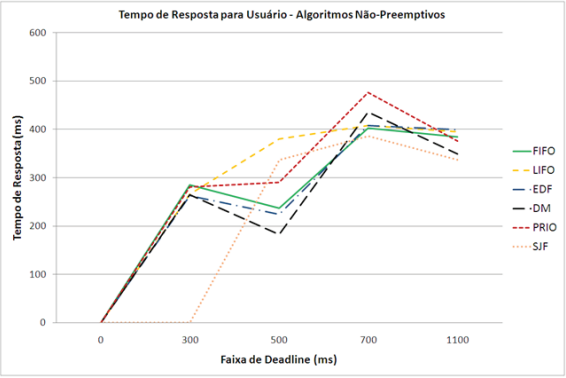

Figura 6.9 – Tempo médio de resposta ao usuário – Algoritmos não preemptivos.

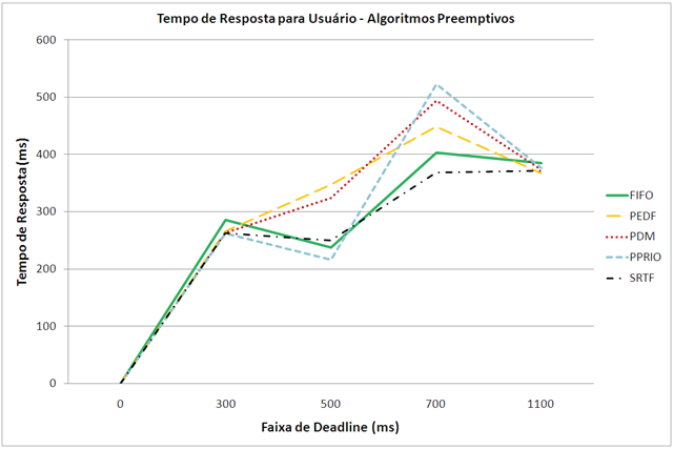

Figura 6.10 – Tempo médio de resposta ao usuário – Algoritmos preemptivos + FIFO.

Em relação ao tempo de resposta dos algoritmos não preemptivos, na faixa de *deadline* de 300 ms, o DM e o EDF apresentaram o melhor resultado e o FIFO, o pior. Para a faixa de 500 ms, houve uma melhora no tempo de resposta do algoritmo FIFO e uma piora no LIFO; e, para as faixas de 700 e 1100 ms, os algoritmos começaram a aproximar seus valores, sendo o SJF o de melhor desempenho para esses valores de faixa.

Para fins de comparação, na análise dos algoritmos preemptivos, foram inseridos os valores de tempo de resposta da política FIFO. Assim, na faixa de 300 ms, todos os algoritmos preemptivos obtiveram um valor muito próximo; na faixa de 500 ms, os valores começaram a se distanciar e foi possível perceber que o escalonamento PPRIO apresentou melhor desempenho; e, para 700 ms, houve um pico em três dos quatro algoritmos preemptivos, talvez devido ao número de preempções realizadas, o que costuma gerar atraso na chegada dos MAs que são suspensos e dos que decidem aguardar na fila. Na última faixa de *deadline*, os valores aproximaram-se muito, sendo possível observar que o algoritmo PEDF apresentou melhor desempenho.

Na relação benefício-tempo de resposta, os resultados apresentados nesta seção podem ser comparados com a média da quantidade de MAs que cumpriram sua missão (Quadro 6.1). Essa comparação permite concluir, com base na análise dos algoritmos não preemptivos, que a política de escalonamento EDF é a mais indicada, pois atende a uma quantidade satisfatória de MAs, com tempo de resposta também satisfatório. Por sua

vez, o algoritmo LIFO, apesar de permitir que uma maioria de MAs cumpra sua missão, possui tempo de resposta alto entre as faixas de 300 a 700 ms.

Quanto aos algoritmos preemptivos, ao realizar essa mesma análise, pode-se perceber que a política SRTF permitiu que uma quantidade satisfatória de MAs cumprisse sua missão com tempo de resposta mais estável, em comparação com os demais algoritmos que sofreram um pico em 700 ms.

Algoritmo de escalonamento	Média de MAs que cumpriram a missão (população = 500)
FIFO	292,5
LIFO	353
EDF	330
PEDF	310,5
DM	305,5
PDM	327
PRIO	315,75
PPRIO	316,25
SJF	295,5
SRTF	308

Quadro 6.1 – Média de MAs que cumpriram sua missão.

6.2 DIFICULDADES ENCONTRADAS NA IMPLEMENTAÇÃO

Quanto à plataforma JADE, algumas características foram observadas:

- o número de mensagens trocadas pelos agentes não deve ultrapassar 1.000.000 (como um todo) e o *flush* demora cerca de 1 segundo, o que impossibilitou a simulação de um número maior de MAs;
- as funções *activate()* e *suspend()* do JADE estavam disponíveis, porém continham erro, tendo sido necessário entrar em contato com o desenvolvedor do projeto para que tal erro fosse solucionado;

- ao solicitar a posição de um agente, o JADE varre todos os contêineres para localizá-lo. No caso de vários contêineres, o processo torna-se demorado; portanto, para este trabalho, foi necessário utilizar um único contêiner de agentes (salvo a UI, por estar em uma máquina remota distinta da arquitetura);
- ao solicitar a ativação de um agente, o JADE busca-o nos contêineres e ressincroniza o cache com o servidor; já ao solicitar a suspensão, o agente é pausado e seu contexto salvo. A forma como isso é feito não é claramente explicada pelos autores da plataforma;
- a migração de MAs é feita pela transferência de arquivos JAR, cuja execução é demorada; por isso, obteve-se um tempo de migração médio superior a 80 ms.

6.3 CONCLUSÕES

Este capítulo apresentou os tipos de testes e análise de resultados da proposta descrita no Capítulo 5, a qual foi testada no próprio *framework* JADE, sendo que as medidas de desempenho mostraram que a quantidade de MAs com restrições temporais que cumpriu sua missão dentro do *deadline* estipulado utilizando o RT-JADE foi de 1,6 a 96,1% maior para cinco MAs concorrentes e de 47 a 100% maior para vinte MAs concorrentes, em comparação com a atual política de escalonamento (FIFO) fornecida pelo JADE.

Analisando ainda mais o teste de benefício, é possível observar que, dos três melhores resultados, dois foram de algoritmos preemptivos em três das quatro faixas de *deadline* atribuídas para 5 e 20 MAs concorrentes, enfatizando, assim, a importância da preempção em algoritmos de escalonamento. Também, é possível analisar que o algoritmo EDF esteve entre os três melhores resultados em todas as faixas de *deadline* para 20 MAs concorrentes e entre os três melhores resultados em duas das quatro faixas de *deadline* atribuídas para 5 MAs concorrentes.

Analisando ainda mais os gráficos para 20 MAs concorrentes, dos três melhores resultados em três das quatro faixas de *deadline* aplicadas, pelo menos dois algoritmos de escalonamento foram de políticas diretamente relacionadas ao *deadline* (DM, EDF e seus preemptivos) e, na última faixa (2500 ms), dos cinco melhores resultados (políticas de escalonamen-

to que permitiram a um mínimo de 1900 MAs de um total de 2000 MAs cumprir seu *deadline*), quatro foram de algoritmos diretamente ligados ao *deadline*, sendo eles: DM, PDM, EDF e PEDF. Ainda, vale enfatizar que o algoritmo PEDF na faixa de 2000 ms para 20 MAS concorrentes permitiu que nenhum MA perdesse seu *deadline*, mas sim que cumprisse total ou parcialmente sua missão.

Também é possível concluir que os algoritmos com melhor custo-benefício e melhor taxa de transferência foram: LIFO e PDM, para cinco MAs concorrentes, e EDF e PPRIO, para dez MAs concorrentes; porém, na relação benefício-tempo de resposta por parte dos usuários, EDF e SRTF obtiveram melhor desempenho.

Partindo da análise dos quatro testes realizados, conclui-se que, para o cenário simulado, o algoritmo de escalonamento com melhor desempenho foi o EDF.

7 CONCLUSÕES E TRABALHOS FUTUROS

7.1 REVISÃO DAS MOTIVAÇÕES E OBJETIVOS

Apesar de haver um grande número de pesquisas voltadas para MAs, foi possível perceber muitas lacunas nas pesquisas que tratam do atendimento destes em um mesmo *host.*

Para a realização deste trabalho, foram estudadas diversas áreas, tais como: agentes inteligentes móveis, tempo real, concorrência de tarefas, plataformas de *middlewares* para desenvolvimento de agentes e a arquitetura JADE, sendo que este estudo serviu como base para a proposição da arquitetura apresentada nesta dissertação. O objetivo geral foi apresentar uma extensão de *middleware* que fornecesse à plataforma JADE suporte ao escalonamento de MAs em tempo real.

Visando a atender este objetivo geral, os seguintes objetivos específicos foram alcançados:

- pesquisa na literatura sobre o estado da arte;
- estudo do comportamento de MAs concorrentes com restrições temporais utilizando a política FIFO, empregada pelo JADE;
- desenvolvimento de uma arquitetura que desse suporte para diferentes políticas de escalonamento de MAs em um *host* com múltiplos recursos;
- desenvolvimento de algoritmos de escalonamento (preemptivos e não preemptivos) eficientes que permitissem ao maior número possível de MAs concluir suas missões, respeitando seus *deadlines*;
- análise comparativa do desempenho das políticas de escalonamento no RT-JADE.

7.2 VISÃO GERAL DO TRABALHO

A proposta de um sistema distribuído no qual um MA percorre *hosts* com o objetivo de cumprir uma missão dentro de um prazo estipulado foi implementada. Nesse sentido, entende-se que a missão é composta por um conjunto de requisições e recursos a ser utilizado para atender a essas re-

quisições, sendo que cada recurso representa um benefício adicional para a missão do agente.

Portanto, o objetivo de construir uma abordagem em que um MA conseguisse utilizar todos os recursos necessários em *hosts* distintos para cumprir sua missão de forma completa, sem ultrapassar seu *deadline*, foi alcançado. Com base neste objetivo, observou-se a possibilidade de cumprimento parcial da missão, em caso de *deadline* próximo, a qual foi também inserida na arquitetura.

Por sua vez, a definição dinâmica do itinerário para o MA foi feita de forma que ele só tenha conhecimento dos *hosts* interligados ao nodo corrente após a utilização do recurso, o que possibilitou uma melhor avaliação das políticas de escalonamento e sua influência sobre o resultado final com respeito ao cumprimento de missões. Já o uso de históricos, com o objetivo de fornecer aos MAs o tempo estimativo de permanência e uso do recurso no *host* corrente, permitiu a estes a decisão por continuar no nodo ou migrar para outro em busca do mesmo e/ou de outro recurso, levando em conta o *deadline* para o cumprimento da missão.

Por fim, o comportamento dos algoritmos de escalonamento propostos foi analisado por meio de simulações e os resultados foram comparados aos resultados do algoritmo FIFO, presente na plataforma JADE, com o intuito de determinar a política de escalonamento com melhor rendimento em diferentes situações. Os resultados apresentados após as simulações das abordagens mostraram desempenho bastante satisfatório, permitindo que um grupo maior de MAs com restrições temporais cumprisse sua missão, em comparação com o que se possui atualmente no ferramental.

7.3 PRINCIPAIS CONTRIBUIÇÕES DESTA DISSERTAÇÃO

Dentro dos objetivos traçados para este trabalho e das atividades desenvolvidas durante este período, pode-se citar que o principal destaque desta dissertação foi a proposição de um modelo inédito de associação de MAs com algoritmos de escalonamento.

De forma a divulgar os conceitos e resultados obtidos com a arquitetura proposta e, principalmente, submeter seus resultados para uma avaliação crítica da comunidade científica que se ocupa das questões discutidas nesta dissertação, dois documentos em forma de artigo foram produzidos, dos quais um resultou em aceite para publicação. Ainda, as revisões e dis-

cussões provenientes desses artigos contribuíram para elucidar algumas limitações e lacunas dos resultados preliminares e para motivar a superação desses problemas na arquitetura proposta final.

Conforme citado anteriormente, o documento científico produzido e aceito para publicação foi um artigo em evento nacional:

1. FILGUEIRAS, T. P.; LUNG, L. C.; RECH, L. O. RT-JADE: *middleware* com suporte para escalonamento de agentes móveis em tempo real. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES E SISTEMAS DISTRIBUÍDOS (SBRC), 29., 2011, Campo Grande. **Anais...** Campo Grande: [s.n.], 2011.

7.4 TRABALHOS FUTUROS

Considerando o atual estágio deste projeto, algumas possibilidades para a sua continuidade são apresentadas a seguir:

- utilizar a teoria das filas para o cálculo de tempo de espera dos agentes móveis por média ponderada;
- otimização dos algoritmos de escalonamento preemptivos (como ajuste dos picos dos tempos de resposta para usuários);
- adição de novas políticas de escalonamento, como RR;
- adição de heurísticas de comportamento capazes de auxiliar no cumprimento da missão.

REFERÊNCIAS

AHRENS, D. Fast quick sort for JAVA. **Sorting algorithms**, 1998. Disponível em: <http://people.cs.ubc.ca/~harrison/Java>. Acesso em: 10 jan. 2011.

ALLEN, A. O. **Probability, statistics, and queuing theory with computer science application**. New York: Academic Press, 1978.

ARUNACHALAN, B.; LIGHT, J. Agent-based Mobile Middleware Architecture (AMMA) for patient-care clinical data messaging using wireless networks. In: IEEE/ACM INTERNATIONAL SYMPOSIUM ON DISTRIBUTED SIMULATION AND REAL-TIME APPLICATIONS, 12., 2008, Vancouver. **Proceedings…** Vancouver: IEEE, 2008.

AUDSLEY, N. C. et al. Applying new scheduling theory to static priority pre-emptive scheduling. **Software Engineering Journal**, v. 8, n. 5, p. 284-292, 1993.

BAEK, J.; KIM, G.; YEOM, H. Y. Cost-effective planning of timed mobile agents. In: INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY: CODING AND COMPUTING (ITCC), 3., 2002, Las Vegas. **Proceedings…** Las Vegas: IEEE, 2002. p. 536-554.

BARBOSA, R. M.; GOLDMAN, A. Mobigrid – Framework for mobile agents on computer grid environments. In: KARMOUCH, A.; KORBA, L; MADEIRA, E. R. M. (Eds.). **Mobility aware technologies and applications**. Berlin: Springer, 2004. (LNCS 3284). p. 147-157.

BARLAND, I.; GREINER, J.; VARDI, M. Concurrent processes: basic issues. **The Connexions Project**. Oct. 6 2005. Disponível em: <http://cnx.org/content/m12312/1.16/content_info>. Acesso em: 10 jan. 2011.

BÄUMER, C.; MAGEDANZ, T. Grasshopper – A mobile agent platform for active telecommunication. In: ALBAYRAK, S. (Ed.). **Intelligent agents for telecommunication applications.** Berlin: Springer, 1999. (LNCS 1699). p. 19-32.

BELLIFEMINE, F.; CAIRE, G.; GREENWOOD, D. **Developing multi-agent systems with JADE**. West Sussex: John Wiley & Sons, 2007. (Wiley Series in Agent Technology).

BOND, A. H.; GASSER, L. **Readings in distributed artificial intelligence**. San Mateo: Morgan Kaufmann, 1988.

BRAUN, P.; ROSSAK, W. R. **Mobile agents:** basic concepts, mobility models, and the tracy toolkit. San Francisco: Morgan Kaufmann, 2005.

CAO, J. et al. A consensus algorithm for synchronous distributed systems using mobile agent. In: PACIFIC RIM INTERNATIONAL SYMPOSIUM ON DEPENDABLE COMPUTING (PRDC), 9., 2002, Tsukuba-City. **Proceedings…** Tsukuba-City: IEEE, 2002. p. 229-238.

CAO, J. et al. Grid load balancing using intelligent agents. **Future Generation Computer Systems**, v. 21, n. 1, p. 135-149, 2005.

COSTA, A. C. R. **Some principles for a functionalist account of complex systems.** Unpublished. 1993.

COULOURUS, G.; DOLLIMORE, J.; KINDBERG, T. **Sistemas distribuídos conceitos e projeto**. Porto Alegre: Bookman, 2007.

CHANG, J. et al. Scheduling algorithm of load balancing based on dynamic policies. In: INTERNATIONAL CONFERENCE ON NETWORK-

ING AND SERVICES (ICNS), 6., 2010, Cancun. **Proceedings…** Cancun: IARIA, 2010.

CHEN, B.; CHENG, H.; PALEN, J. Mobile-C: a mobile agent platform for mobile C/C++ code. **Software – Practice & Experience**, New Jersey, v. 36, n. 15, p. 1711-1733, 2006.

CHENG, S.; STANKOVIC, J. A.; RAMAMRITHAM, K. Scheduling algorithms for hard real-time systems: a brief survey. In: STANKOVICK, J. A.; RAMAMRITHAM, K. (Eds.). **Hard real-time systems**. Washington: IEEE CS Press, 1998. p. 150-173.

CHOI, S. et al. A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching. **ACM Transactions on Computer Systems**, v. 22, n. 2, p. 214-280, 2004.

DOTTI, F. L.; DUARTE, L. M. Desenvolvimento de aplicações móveis corretas. In: WORKSHOP DE COMUNICAÇÃO SEM FIO E COMPUTAÇÃO MÓVEL, III., 2001, Recife. **Anais...** Recife: [s.n.], 2001. p. 10-17.

FARINES, J. M.; FRAGA, J. S.; OLIVEIRA, R. S. **Sistemas de tempo real.** São Paulo: Escola de Computação USP, 2000. 1 v.

FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS (FIPA). **FIPA ACL message structure specification**. Geneva: FIPA, 2002. Disponível em: <http://www.fipa.org/specs/fipa00061/SC00061G.pdf>. Acesso em: 20 jan. 2011.

FOU, J. Web services and mobile intelligent agents – Combining intelligence with mobility. **Web Services Architect**. 10 out. 2001. Disponível em: <http://www.webservicesarchitect.com/content/articles/fou02.asp>.

Acesso em: 10 jan. 2011.

FOK, C.; ROMAN, G.; LU, C. Mobile agent middleware for sensor networks: an application case study. In: INTERNATIONAL CONFERENCE ON INFORMATION PROCESSING IN SENSOR NETWORKS (IPSN), 4., 2005, Los Angeles. **Proceedings…** Los Angeles: IEEE, 2005. p. 382-387.

FRACHTENBERG, E.; SCHWIEGELSHOHN, U. (Eds.). **Job scheduling strategies for parallel processing**. Berlin: Springer, 2010. (LNCS 6253).

FRANKLIN, S.; GRAESSER, A. Is it an agent, or just a program?: a taxonomy for autonomous agents. In: MÜLLER, J. P.; WOOLDRIDGE, M. J.; JENNINGS, N. R. (Eds.). **Intelligent agents III:** agent theories, architectures, and languages. Berlin: Springer, 1996. p. 21-35.

GOLDCHLEGER, A. et al. InteGrade: object-oriented grid middleware leveraging idle computing power of desktop machines. **Concurrency and Computation: Practice & Experience – Middleware for Grid Computing**, Chichester, v. 16, n. 5, p. 449-459, Apr. 2004.

GRACE, P.; BLAIR, S. G.; SAMUEL, S. ReMMoC: a reflective middleware to support mobile client interoperability. In: MEERSMAN, R. et al. (Eds.). **On the move to meaningful internet systems 2003:** CoopIS, DOA, and ODBASE. Berlin: Sringer, 2003. (LNCS 2888). p. 1170-1187.

GROSOF, B. N. et al. Reusable architecture for embedding rule-based intelligence in information agents. In: ACM CONFERENCE ON INFORMATION AND KNOWLEDGE MANAGEMENT (CIKM), 4., 1995, Baltimore. **Proceedings of the workshop on intelligent information agents**. Baltimore: ACM, 1995.

HAYES-ROTH, B. An architecture for adaptive intelligent systems. **Artificial intelligence:** special issue on agents and interactivity, n. 72, p. 329-365, 1995.

HUHNS, M.; STEPHENS, L. M. Multiagent systems and societies of agents. In: WEISS, G. (Ed.). **Multiagent systems:** a modern approach to distributed artificial intelligence. Massachusetts: MIT Press, 1999. p. 121-164.

JENNINGS, N. et al. Automated negotiation. In: INTERNATIONAL CONFERENCE ON THE PRACTICAL APPLICATION OF INTELLIGENT AGENTS AND MULTI-AGENT SYSTEMS (PAAMS), 5., 2000, Manchester. **Proceedings**… Manchester: ACM, 2000. p. 23-30.

KANG, P. et. al. Smart messages: a distributed computing platform for networks of embedded systems. **The Computer Journal**, Oxford, v. 47, n. 4, p. 475-494, 2004.

KOPETZ, H. **Real-time systems:** design principles for distributed embedded applications. Boston: Kluwer Academic Publishers, 1997.

KRUSE, R. L. **Data structures & program design.** 2. ed. New Jersey: Prentice-Hall, 1987.

LANGE, D. B. et al. Aglets: programming mobile agents in Java. In: MASUDA, T.; MASUNAGA, Y.; TSUKAMOTO, M. (Eds). **Worldwide computing and its applications**. Berlin: Springer, 1997. (LNSC 1274). p. 253-266.

LEUNG, J.Y.T.; WHITEHEAD, J. On the complexity of fixed-priority scheduling of periodic, real-time tasks. **Performance Evaluation**, North-Holland, v. 2, n. 4, p. 237-250, Dec. 1982.

LEUNG, K. K. **FTS framework for JADE**. 2010. Disponível em: <http://www.cse.cuhk.edu.hk/~kwng/FTS.html>. Acesso em: 10 jan. 2011.

LIPSCHUTZ, S. **Schaum's outline of 'theory and problems of data structures'**. 1. ed. Columbus: McGraw-Hill, 1986.

LIU, C. L.; LAYLAND, J. Scheduling algorithms for multiprogramming in a hard real-time environment. **Journal of the ACM**, v. 20, n. 1, p. 46-61, 1973.

McHALE, C. **Synchronization in concurrent object-oriented languages.** 1994. Thesis (PhD) – Trinity College, Dublin, 1994.

MULLER, J. P. Architectures and applications of intelligent agents: a survey. **The Knowledge Engineering Review**, Cambridge, v. 13, p. 353-380, 1999.

OBJECT MANAGEMENT GROUP (OMG). MAF – Mobile Agent Facility specification. **Catalog of OMG specifications**. 2011. Disponível em: <http://www.omg.org/technology/documents/spec_catalog.htm>. Acesso em: 15 jan. 2011.

PANICO, J. A. **Queuing theory**. New Jersey: Prentice-Hall, 1969.

PATTIE, M. Artificial life meets entertainment: life like autonomous agents. **Communications of the ACM**, v. 38, n. 11, p. 108-114, 1995.

PEDREIRAS, P.; ALMEIDA, L. Approaches to enforce real-time behavior in ethernet. In: ZURAWSKI, R. (Ed.). **The industrial communication systems handbook**. Boca Raton: CRC Press, 2005.

PICCO, G. P.; FUGGETA, A.; VIGNA, G. Understanding code mobility. **IEEE Transaction on Software Engineering**, Washington, v. 24, n. 5, p. 342-361, May 1998.

ROSSIER-RAMUZ, D.; SCHEURER, R. An ecosystem-inspired mobile agent middleware for active network management. In: KARMOUCH, A.; MAGEDANZ, T.; DELGADO, J. (Eds.). **Mobile agents for telecommunication applications**. Berlin: Springer, 2002. (LNCS 2521). p. 73-82.

RUSSEL, S.; NORVIG, P. **Artificial intelligence:** a modern approach. New Jersey: Prentice Hall, 2009. v. 3.

SAHINGOZ, O. K.; ERDOGAN, N. A two-leveled mobile agent system for e-commerce with constraint-based filtering. In: BUBAK, M. et al. (Eds.). **Computational science** – ICCS 2004. Berlin: Springer, 2004. (LNCS 3036). p. 437-440.

SAJJA, P. S. Multi-agent system for knowledge-based access to distributed databases. **Interdisciplinary Journal of Information, Knowledge, and Management**, Santa Rosa, v. 3, p. 1-9, 2008.

SALIM, S.; JAVED, M.; AKBAR, A. H. **A mobile agent-based architecture for fault tolerance in wireless sensor networks**. In: ANNUAL COMMUNICATION NETWORKS AND SERVICES RESEARCH CONFERENCE (CNSR), 8., 2010, Montreal. **Proceedings...** Montreal: IEEE, 2010. p. 276-283.

SHAW, A. C. **Sistemas e software de tempo real**. Porto Alegre: Bookman, 2003.

SHEMSHADI, A.; SOROOR, J.; TAROKH, M. J. Implementing a multi-agent system for the real-time coordination of a typical supply chain based

on the JADE technology. In: IEEE INTERNATIONAL CONFERENCE ON SYSTEM OF SYSTEMS ENGINEERING (SoSE), 3., 2008, Monterey. **Proceedings…** Monterey: IEEE, 2008.

SHRIVASTAVA, S. K.; BANATRE, J. P. Reliable resource allocation between unreliable processes. **IEEE Transactions on Software Engineering**, Washington, v. 4, n. 3, p. 230-241, May 1978.

SILVA, E. L.; MENEZES, E. M. **Metodologia da pesquisa e elaboração de dissertação**. Florianópolis: Universidade Federal de Santa Catarina, 2005.

STANKOVIC, J. A. et al. Strategic directions in real-time and embedded systems. **ACM Computing Surveys**, v. 25, n. 4, p. 751-763, Dec. 1996.

TANENBAUM, A. **Modern operating systems**. New Jersey: Prentice Hall, 2001.

TANENBAUM, A.; STEEN, M. **Distributed systems:** principles and paradigms. 2. ed. New Jersey: Prentice Hall, 2007.

WANG, S. et al. An intelligent manufacturing system: agent lives in adhesive slice. **International Journal of Computer Science and Network Security**, Seoul, v. 6, n. 5A, p. 73-80, May 2006.

WEYNS, D. **Architecture-based design of multi-agent systems**. 1. ed. Berlin: Springer, 2010.

WIERLEMANN, T.; KASSING, T.; HARMER, J. The on the move project: description of mobile middleware and experimental results. In: HOLTZMAN, J. M.; ZORZI, M. (Eds.). **Advances in wireless commu-**

nications. New York: Kluwer, 2002. (The Kluwer International Series in Engineering and Computer Science 435). p. 21-35.

WONG, D. et al. Concordia: an infrastructure for collaborating mobile agents. In: ROTHERMEL, K.; POPESCU-ZELETIN, R. (Eds). **Mobile agents**. Berlin: Springer, 1997. (LNCS 1219). p. 86-97.